BEDROCK ARCHITECTURE

Programmer's Reference Manual

CONTENTS

LIST OF TABLES

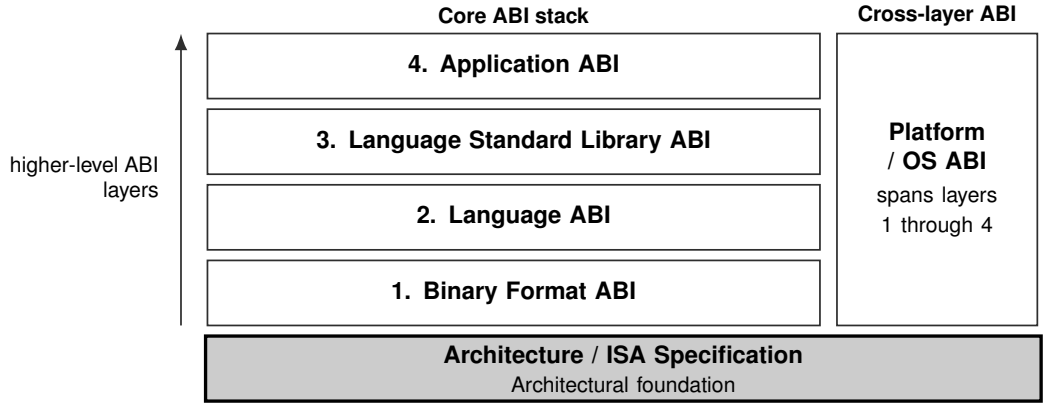
LIST OF FIGURES

PART I

Architectural Foundations

1 OVERVIEW

1.1 Architecture and ABI Layers



Current document: Bedrock Programmer's Reference Manual. The shaded block identifies its primary position in the architecture and ABI stack. The Platform / OS ABI spans and constrains the four core ABI layers; the ISA specification defines the architecture beneath them.

1.2 Manual Scope

This manual defines programmer-visible architectural state, operand encoding, execution semantics, and instruction behavior for the Bedrock instruction set architecture.

1.3 Architectural Profile

Bedrock defines sixteen 64-bit general-purpose registers, R0 through R15. SP and PC are special architectural registers outside the Rn namespace, with SP-relative addressing tied to SS and PC-relative addressing tied to CS.

- The PC names a byte in the instruction stream; sequential execution advances it by the encoded instruction length determined during decoding.
- Encoded instruction length is explicit and bounded; the base profile defines encoded instruction lengths from 1 to 18 bytes.
- Four integer operation sizes: B, W, L, and Q.
- The optional scalable-vector extension provides 32 VLEN-bit vector registers and 16 packed predicate registers; VLEN is a reset-stable power of two from 128 through 2048 bits.
- The effective-address model covers register, memory, immediate, absolute, segment-qualified, indexed, and auto-update operands.
- Segment pre-translation precedes optional page-table translation.
- User and supervisor modes share the instruction set, with privileged instructions and checked user-access moves defining supervisor-only behavior.

1.4 Normative Material and Specificity

Architecture prose, ordered steps, architecture tables, instruction encoding diagrams, and the Operation and Detailed Semantics fields of instruction entries are normative unless a section explicitly identifies material as a summary, example, or validation status.

Rule set or provider	Covered material
Instruction definitions	Concrete instruction encodings, operand and effective-address grammar, extension wiring, and document ordering.
Instruction Execution Model common rules and instruction-entry Operation and Detailed Semantics	Executable instruction behavior, architectural state transitions, fault and commit ordering, repeat and architectural-event behavior, and single-logical-processor memory-access sequencing.
Formal memory model	Permitted cross-logical-processor ordering and visibility.
ABI specifications	ELF, C ABI, and calling-convention contracts.
Compiler-interface specifications	Source-language and compiler-facing target-interface contracts.
External numerical floating-point providers	Numerical results and certificates only; input validation and the resulting architectural trap or commit behavior follow the applicable Instruction Execution Model common rules and instruction entry's Operation and Detailed Semantics.

Presented material remains normative where declared and follows the applicable rule set. For executable behavior, readers apply the Instruction Execution Model common rules together with the applicable instruction entry's Operation and Detailed Semantics. The following specificity rules govern narrowing among applicable rules:

1. Opcode-space length, required instruction length, encoded instruction length, field values, and decoding validity follow the selected instruction encoding, its field constraints, and the Instruction Encoding and Effective Addressing chapters.
2. Execution of a decoded instruction follows its Operation and Detailed Semantics. An instruction-specific rule narrows the common execution, operand, flag, memory, or event rule that it names.
3. Common architectural behavior follows the chapter that defines that behavior. A structure-specific rule narrows only the common rule that it names.
4. Summary tables and examples provide navigation or explanatory context and do not replace a normative rule.

Two normative statements at the same specificity are required to agree. A discrepancy at that level is a specification defect and is resolved by correcting the normative sources and their derived tables together.

2 TERMINOLOGY

2.1 Address Values

An *effective address (EA)* is the address value or operand designator produced by an addressing mode before segment pre-translation. A memory EA produces an address, while register and immediate EAs name register and immediate operands.

A *pre-segment virtual address* is a virtual-address value before segment pre-translation. Segment pre-translation either converts it to a linear address or reports the architecturally defined fault.

A *linear address* is the result of segment pre-translation. The page-table walker consumes it when paging is enabled; otherwise the memory system uses it directly.

A *physical address* is the byte address produced by page-table translation. With paging disabled, the linear address is used directly after the implementation PABITS check.

A *memory-system address* is the byte address presented to the memory subsystem after every active architectural translation stage.

Normal memory is the coherent memory-location model.

Device memory is the platform-owned physical class that requires MMIO execution rules. A leaf AM selects Normal or MMIO access semantics, while the orthogonal CP field selects one of four cache policies.

Address generation is the instruction-local calculation that combines registers, displacement, index, scale, and immediate fields into an effective address. It does not by itself read memory.

2.2 Memory Behavior

A *memory access* is an individual architectural instruction-fetch, read, or write attempt.

A *memory operation* is an instruction-level semantic operation that may contain one or more memory accesses or memory events.

A *memory event* is a read or write unit that participates in coherence or memory ordering.

A *memory effect* is a result that becomes observable to another logical processor or an external system.

Ordinary is the classification of an operation, such as an ordinary load or store, within the operation taxonomy. Memory type is classified separately.

2.3 Segmentation and Paging

Segment pre-translation is the stage between effective-address calculation and paging. A disabled segment passes the EA through. A translated window checks an offset and adds its base, while a bounds-only window checks an already-linear address without adding the base.

A *segment register image* is the 64-bit architectural value containing the segment base page, size exponent, size mantissa, and bounds-only enable bit.

A *disabled segment* is a segment with a zero size mantissa. It performs no segment-window check or base addition and passes the effective address through as the linear address.

A *translated segment* is an enabled segment with bounds-only mode clear. It checks the effective address as an offset within the segment span and then adds the segment base.

A *bounds-only segment* is an enabled segment with bounds-only mode set. It treats the effective address as already linear and checks only that it lies inside the segment window.

Page-table translation is the optional PTCR.PE-controlled stage that maps a canonical linear address to a physical frame and applies effective permissions, access class, cache policy, and A/D state. Bit 7 is A in both descriptor layouts.

A *physical frame number (PFN)* is a 16-KiB frame identifier used by leaf page-table entries. An L1, L2, L3, or L4 leaf PTE combines its naturally aligned PFN base with the low 14, 23, 34, or 45 linear-address bits respectively to form a byte address.

2.4 Instruction Encoding

An *opcode* is the value that selects an instruction operation.

The *opcode space* is the contiguous leading region of an encoded instruction, from byte 0 through the byte immediately before the first payload. It includes the header and may contain operand or control fields in addition to the opcode.

The *header* is the leading byte range within the opcode space that determines the encoded instruction length and opcode-space selection. It is the first byte for extrashort and short instructions and the first two bytes for medium and longer instructions.

A *payload* is an independently determined unit after the opcode space that supplies additional information and increases the required instruction length. Adjacent units are separate payloads when their presence, length, or meaning is determined independently.

A *field* is a defined bit region within the opcode space or a particular payload. An undivided payload may carry its value without defining a separate field.

The *required instruction length* is the opcode-space length plus the length of every payload. It excludes padding.

The *encoded instruction length* is the required instruction length plus any trailing padding in the encoded record.

Padding is trailing encoded-record content that carries no information. Payload accounting excludes padding.

2.5 Control Flow

A *near control transfer* is a CALL, RET, JMP, or conditional branch that changes only PC within the current code-segment context.

A *segment-changing control transfer* is a transfer that updates CS and PC in one architectural commit. Long jumps, long calls, and long returns belong to this category.

Long Call is the instruction family that creates a two-value return context while changing the code segment and program counter.

Long Jump is the instruction family that changes the code segment and program counter without creating a return context.

Long Return is the instruction family that consumes a two-value return context to restore the code segment and program counter.

A *long return frame* is the two-value return context created by Long Call and consumed by Long Return.

2.6 Tracing Terms

The *TRACE instruction* is the marker instruction that records its immediate marker and current instruction boundary in the implementation trace stream when that stream is enabled.

A *trace unit* is the execution-completion unit sampled by STATUS.TF.

SINGLE_STEP is the architectural event delivered after successful completion of a TF-selected trace unit when RF does not suppress the automatic debug stop.

2.7 Architectural State

A *logical processor* is an architecturally independent execution context that owns its architectural state and remains the architectural execution subject.

A *thread* is a software execution context that may execute on a logical processor.

A *general-purpose register* is one of the sixteen 64-bit registers R0 through R15.

An *Rn register* is a general-purpose register selected through the four-bit Rn operand namespace. It may hold integers, addresses, counters, selectors, or temporary data according to the instruction encoding.

The *stack pointer register* is SP. It lies outside the four-bit Rn namespace and uses SS as the fixed segment for SP-relative addressing.

A *segment register operand* is a 64-bit segment register exposed through the SREG operand class. The SREG selector table maps its eight encodings to DS, SS, and GS0 through GS5. RDSEG CS and PUSH CS read the current CS image through their fixed CS encodings.

A *state register* is named architectural state such as FLAGS or STATUS. RDFLAGS, WRFLAGS, RDSTATUS, and WRSTATUS access these registers.

A *control register* is a privileged register exposed through the CR operand class, such as PTCR, ASCR, ECR, EPC, UCTL, and UINFO.

Architectural state is programmer-visible current state.

A *register image* is the complete bit representation read from or written to one register.

A *processor-state image* is an aggregate representation of multiple architectural-state components, such as a reset-state table.

Processor state is the complete architectural state set of a logical processor.

A *temporary operand-evaluation state* is a working copy used during operand evaluation.

The *program counter* is the architectural state object that selects instruction execution.

An *instruction address* is the numeric address of an instruction.

A *continuation* is a saved or selected resumption point.

2.8 Architectural Exceptions

An *architectural exception* is a synchronous architectural event delivered as a fault or trap.

2.9 Floating-Point and Architectural Exceptions

A *floating-point exception condition* is an IEEE-754 condition such as NV, DZ, OF, UF, or NX. An enabled condition raises the FLOATING_POINT_EXCEPTION architectural event before the instruction commits the effects suppressed by that event.

A *floating-point exception flag* is the accrued FFLAGS bit corresponding to a floating-point exception condition.

2.10 Vector Architectural State

A *vector register* is one of the 32 VLEN-bit registers V0 through V31.

A *predicate register* is one of P0 through P15 and contains one governing bit for each byte of a vector register.

A *logical vector lane* is one selected-width element. Predicate bit iE governs lane i for an E-byte element. VLEN is the reset-stable implementation-selected vector register width in bits.

2.11 Normative Qualifiers

Reserved is a classification meaning that software must not depend on the field's value or behavior.

Must be zero is a requirement that software write zero.

Read-as-zero is a rule under which reads return zero while the field remains reserved.

Ignored is a rule under which hardware accepts the field without using it for the current operation.

Preserved is a requirement that software retain the previous value when rewriting the containing object.

Software-defined state is state reserved for software use. Hardware interpretation is limited to behavior explicitly defined by the containing structure.

Implementation-defined behavior is behavior selected by the implementation and published through CPUID, platform documentation, or firmware.

An *illegal use* is use of an opcode, effective-address form, selector, or operation that raises the exception named by the defining rule.

3 PROGRAMMING MODEL

3.1 Register Model

The general-purpose register set contains sixteen 64-bit registers, R0 through R15. Instruction encodings use these registers for integer values, addresses, counters, selectors, and temporary data.

SP and PC are special architectural registers outside the four-bit Rn namespace. SP-relative memory forms use SS by default; PC-relative memory forms use CS by default.

CS, DS, SS, and GS0 through GS5 are 64-bit architectural segment registers. Segment-qualified EA forms select them explicitly, while SP and PC forms omit the segment field because their segment association is fixed.

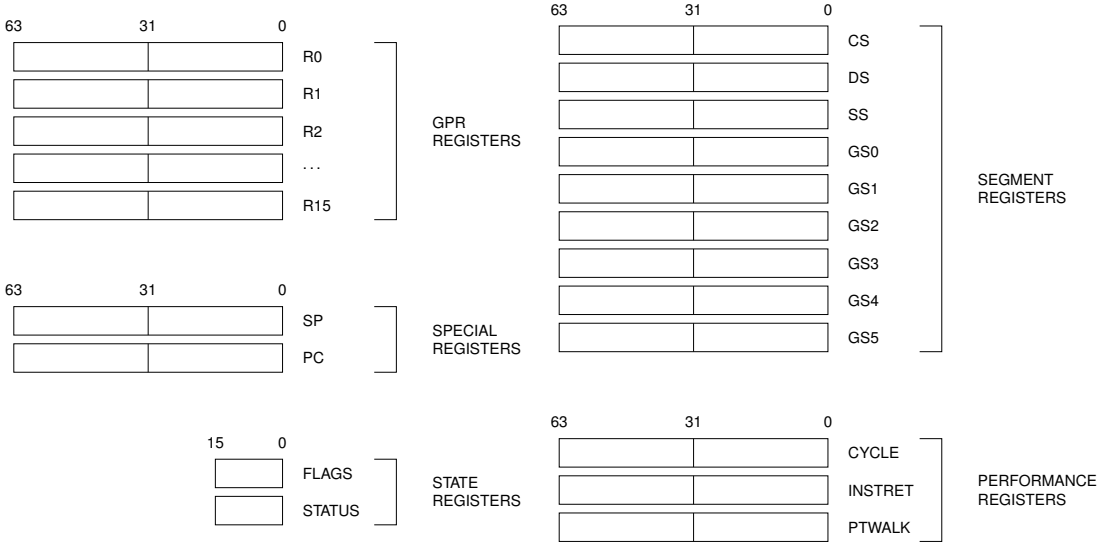


Figure 3-1. Base Register Model

3.2 FLAGS and STATUS Registers

FLAGS and STATUS are accessed through dedicated read/write instructions.

Figure 3-2. FLAGS Register Format

Table 3-1. FLAGS Fields

Field	Bits	Meaning
reserved	15..4	reads as zero and must be zero when written
Z	3	result value is zero
N	2	most significant result bit at the operand size
C	1	unsigned carry out for addition or unsigned borrow for subtraction
V	0	signed overflow or an explicitly reported exceptional condition

RDFLAGS and WRFLAGS are unprivileged. RDFLAGS reads and zero-extends the 16-bit image; WRFLAGS writes the low 16 bits and requires every reserved bit to be zero. Instruction-specific flag production and conditional use are defined in Flags and Condition Codes.

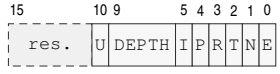


Figure 3-3. STATUS Register Format

In the diagram, U, DEPTH, I, P, R, T, N, and E abbreviate STATUS.UO, STATUS.EDEPTH, STATUS.IE, STATUS.PM, STATUS.RF, STATUS.TF, STATUS.NI, and STATUS.EA.

Table 3-2. STATUS Fields

Field	Bits	Meaning
reserved	15..11	reads as zero and must be zero when written
UO	10	outer active event originated in user mode
EDEPTH	9..6	current architectural event depth
IE	5	permits maskable-interrupt admission when the other admission conditions hold
PM	4	selects user mode when zero and supervisor mode when one
RF	3	one-shot suppression of automatic single-step, execution-breakpoint, and memory-watchpoint stops
TF	2	requests post-success SINGLE_STEP for a trace unit when STATUS.RF is clear
NI	1	inhibits NMI admission while preserving the NMI pending
EA	0	hardware-managed architectural event-active state

In user mode, STATUS.PM, STATUS.EA, STATUS.EDEPTH, and STATUS.UO are always observed as zero.

RDSTATUS is unprivileged; WRSTATUS is supervisor-only. WRSTATUS atomically updates STATUS.IE, STATUS.NI, STATUS.TF, and STATUS.RF; reserved bits must be zero, and supplied STATUS.

PM, STATUS.EA, STATUS.EDEPTH, and STATUS.UO must equal their current values. Privilege transitions are defined in the Privileged Execution Model; event admission, tracing, and event-active transitions are defined in the Architectural Event Processing Model.

3.3 Floating-Point Register Model

The floating-point extension exposes F0 through F15 as 64-bit registers when the floating-point instruction group is implemented.

H-valued conversion operands occupy bits 15..0, S values occupy bits 31..0, and D values occupy all 64 bits. Writing H or S writes zero to the unused upper bits. The presence of an H conversion value in Fn does not enable scalar H arithmetic.

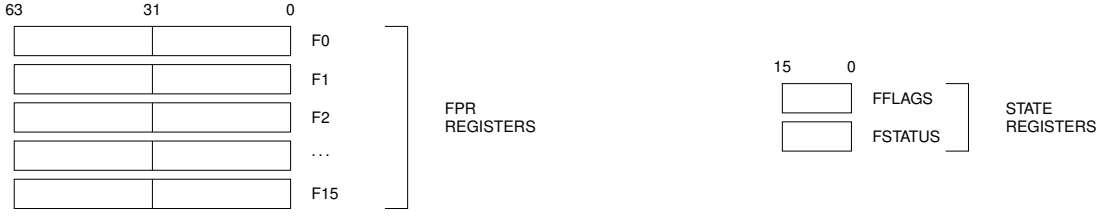


Figure 3-4. FP Register Model

3.4 FFLAGS and FSTATUS Registers

FFLAGS holds accrued IEEE-754 floating-point exception flags. FSTATUS holds the matching exception-condition enables, rounding mode, and optional NaN and subnormal handling modes. Both registers are present only with the floating-point extension and reset to zero.

Figure 3-5. FFLAGS Register Format

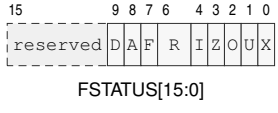


Figure 3-6. FSTATUS Register Format

In the diagrams, I, Z, O, U, and X denote the NV, DZ, OF, UF, and NX flag or enable bits. In FSTATUS, D, A, F, and R denote DN, DAZ, FTZ, and RM.

Table 3-3. FFLAGS Fields

Field	Bits	Meaning
reserved	15..5	reads as zero and must be zero when written
NV..NX	4..0	accrued invalid, divide-by-zero, overflow, underflow, and inexact flags

Table 3-4. FSTATUS Fields

Field	Bits	Meaning
reserved	15..10	reads as zero and must be zero when written
DN	9	produce the architectural default NaN
DAZ	8	treat subnormal inputs as signed zero
FTZ	7	flush tiny results to signed zero
RM	6..5	0 nearest-even; 1 toward zero; 2 toward negative; 3 toward positive
NV_EN..NX_EN	4..0	floating-point exception-condition enables corresponding to FFLAGS.NV through FFLAGS.NX

3.5 Vector Register Model

The scalable-vector extension provides 32 vector registers, V0 through V31, and 16 predicate registers, P0 through P15. A vector register contains VLEN bits. VLEN is an implementation-selected power of two from 128 through 2048 bits and remains stable until reset. A predicate register contains one bit for each byte of a vector register; its packed image therefore occupies VLEN/64 bytes. Reset clears every vector and predicate bit.

The VECTOR discovery predicate and the VECTOR_PARAMETERS CPUID leaf report the extension and VLEN. Integer and raw-bit vector forms require VECTOR. Floating-point vector operations are provided by the separate VECTORFP extension.

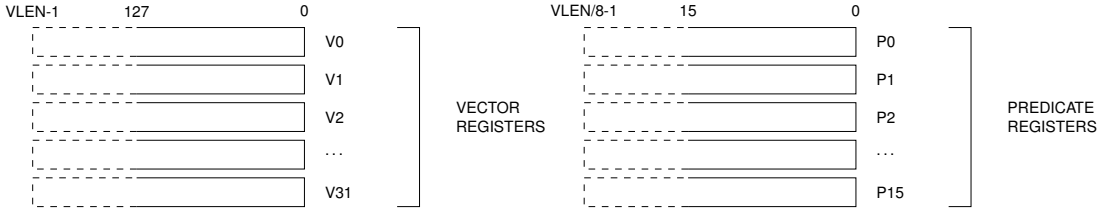


Figure 3-7. VECTOR Register Model

3.6 Segment Registers

Figure 3-8. Segment Register Format

Table 3-5. Segment Register Fields

Field	Bits	Meaning
base_page	63..12	unsigned 4-KiB page number from which $base$ is derived
e	11..7	unsigned exponent used to derive $span$
m	6..1	unsigned mantissa used to derive $span$; zero selects the disabled image
b	0	for an enabled image, zero selects translated-window mode and one selects bounds-only mode

The following quantities are evaluated mathematically without 64-bit truncation:

$$base = base\_page \times 4096,$$
$$span = m \times 2^{e} \times 4096,$$
$$limit = base + span.$$

A segment image with SEGMENT.M=0 is a valid disabled image. An enabled image is valid only when $limit \leq 2^{64}$. The disabled, translated-window, and bounds-only address checks and their fault rules are defined in Segment Pre-Translation.

3.7 Segment Register Operand Class

Instructions that take an explicit segment-register operand use the three-bit SREG encoding below.

SP-relative EA forms do not carry this field and use SS. PC-relative EA forms do not carry this field and use CS.

SREG selects DS, SS, and GS0 through GS5. Instruction fetch and fixed PC-relative addressing select CS implicitly. RDSEG CS and PUSH CS read the current CS image. Segment-changing control transfers update CS and PC together.

Table 3-6. Segment Register Operand Encoding

Segment	Bits	Role	Use
DS	000	data	default data segment
SS	001	stack	stack segment; fixed for SP-relative forms
GS0	010	general	general segment register 0
GS1	011	general	general segment register 1
GS2	100	general	general segment register 2
GS3	101	general	general segment register 3
GS4	110	general	general segment register 4

Table 3-6 (continued)

Segment	Bits	Role	Use
GS5	111	general	general segment register 5

3.8 Resident-Link State

LPC and LPA are 64-bit members of the L architectural-state family. They are not control registers, have no RDCR or WRCR selectors, and are not general-purpose instruction operands. LPC holds a resident continuation PC. LPA bit 63 is the active bit A and bits 62 through 0 are an opaque pointer-authentication field PA.

When A is zero, no resident continuation is active. When A is one and PA is zero, LPC holds an unauthenticated FCALL continuation. When A is one and PA is nonzero, LPC:PA holds an authenticated FPCALL continuation. Base instructions interpret A and preserve the complete LPA image through SAVE and RESTORE; the CFI extension owns the meaning of PA.

Every taken call and every long return requires A to be zero before target or stack access. A false CALLcc does not perform this check. CLRLINK atomically clears LPC and LPA in either privilege mode without accessing memory or changing the CFI key. Event entry and ERET preserve all L state. A call-capable event prologue remains call-free until it has saved the interrupted LPC and LPA image and executed CLRLINK. After restoring an active image, the epilogue executes ERET without an intervening taken call. Supervisor-origin, nested, and non-maskable events follow the same ordering.

3.9 Control-Flow Integrity Key State

LKL and LKH are supervisor-only members of the L architectural-state family and form one 128-bit per-context pointer-authentication key. They are not control registers and have no RDCR, WRCR, or individual read/write instruction. The all-zero combined value is unconfigured; any nonzero value enables CFI indirect-landing checks and permits protected call and return instructions.

Supervisor software transfers both halves atomically with CFISSAVE and CFISRESTORE. Changing the combined key invalidates all protected continuations made with the previous key. CLRLINK does not change the key.

3.10 Control Registers

RDCR and WRCR use a 16-bit selector and transfer a 64-bit register image. Both instructions are supervisor-only. Selectors not listed below raise `INVALID_CONTROL_SELECTOR`. Reserved register bits read as zero, and a write with a nonzero reserved bit raises `RESERVED_CONTROL_BITS` without changing the register. The entry, stack, and user-return registers are per-logical-processor architectural state.

Table 3-7. Control-Register Selector Assignments

Selector	Register	Use
0x0000	PTCR	Page-table root and translation control.
0x0001	ASCR	Address-space identifier control.
0x0002	ECR	Architectural event-delivery control.
0x0108	UPC	User-return program counter.
0x0109	USP	User-return stack pointer.
0x010A	UCS	User-return code-segment image.
0x010B	UDS	User-return data-segment image.
0x010C	USS	User-return stack-segment image.
0x010D	UCTL	User-return FLAGS, STATUS, and validity state.
0x010E	UINFO	User-origin outer-event code.
0x0110	EPC	Common architectural event-entry program counter.
0x0111	ECS	Architectural event-entry code-segment image.
0x0112	EDS	Architectural event-entry data-segment image.
0x0200	SSS	User-origin system-call stack-segment image.
0x0201	SSP	User-origin system-call initial stack top.
0x0210	ISS	Maskable-interrupt stack-segment image.
0x0211	ISP	Maskable-interrupt stack top.
0x0220	FSS	Fault and exception stack-segment image.
0x0221	FSP	Fault and exception stack top.
0x0230	DSS	Double-fault stack-segment image.
0x0231	DSP	Double-fault stack top.
0x0300	ICAP	Implemented interrupt-identity capability.
0x0301	ITHRESH	Interrupt-identity priority threshold.
0x0302	ITOP	Highest-priority eligible interrupt identity.
0x0303	ISEL	Indirect interrupt-file bank and word selector.
0x0304	IDATA	Selected interrupt-file pending or enable word.
0x0400	DTRSEL	Runtime debug-trigger slot and bank selector.
0x0401	DTRDATA	Selected debug-trigger configuration or address-range word.
0x1000	BOOTPC	Warm-reset target supplied by the platform at cold reset.
0x1001	BOOTCFG	Opaque platform boot configuration preserved by warm reset.
0x1100	PMC	Performance-monitor enable.

The selector table is complete for the architectural control registers defined by this manual and is ordered by selector value. The selected register or the subsystem that owns it defines the accepted WRCR source image, validation conditions, and successful-commit effects.

PTCR[63:0]

PTCR Format

In the diagram, P abbreviates PTCR.PE.

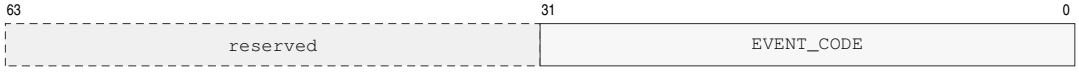
Table 3-8. PTCR Fields

Field	Bits	Meaning
reserved	63..56	must be zero in this page-table format
ROOT_PAGE	55..14	physical frame number of the selected page-table root; when PTCR.PE is one, must fit PABITS
reserved	13..4	must be zero
TT	3..1	complete translation-table format selector
PE	0	page-table translation enable

Table 3-9. PTCR TT Encodings

Encoding	Translation-Table Format
010	LA45 three-level paging with 16-KiB L3/L2 and 4-KiB L1 table objects
011	LA56 four-level paging with 16-KiB L4/L3/L2 and 4-KiB L1 table objects
all others	reserved

The reset PTCR image is zero. When PTCR.PE is zero, ordinary memory accesses do not interpret PTCR.TT or PTCR.ROOT_PAGE. When PTCR.PE is one, PTCR.TT must be 010 or 011 and PTCR.ROOT_PAGE must satisfy the implementation PABITS constraint. A successful WRCR of PTCR commits the complete image and the translation-cache transition selected by its changed fields atomically.

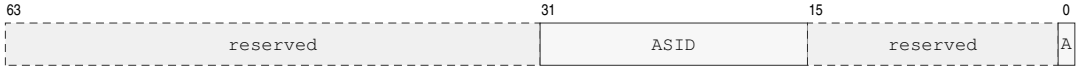


UINFO[63:0]

UINFO Format

Table 3-10. UINFO Fields

Field	Bits	Meaning
reserved	63..32	must be zero
EVENT_CODE	31..0	code for the outer user-origin event and its optional payload shape



ASCR[63:0]

ASCR Format

A abbreviates ASCR.AE in the diagram.

Table 3-11. ASCR Fields

Field	Bits	Meaning
reserved	63..32	must be zero
ASID	31..16	current address-space identifier
reserved	15..1	must be zero
AE	0	address-space identifier matching enable

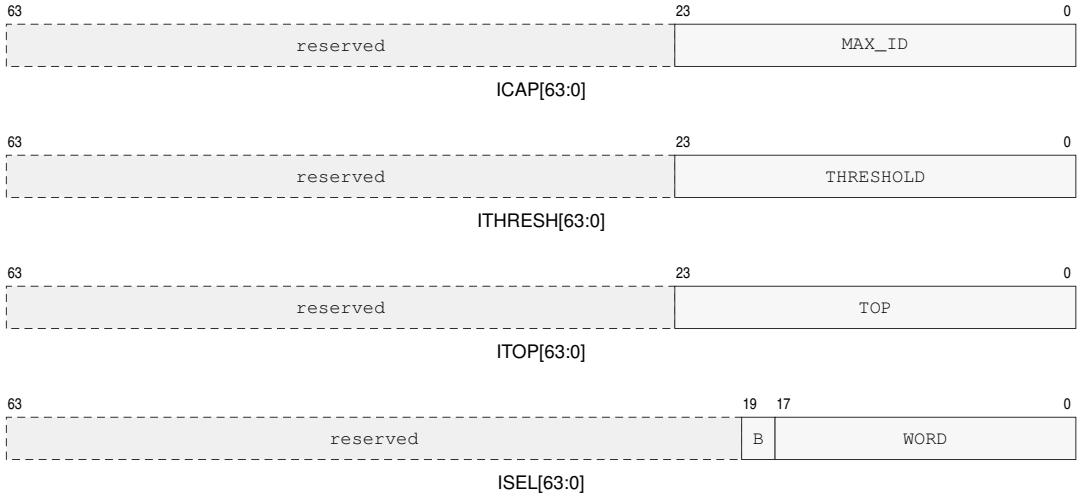
A successful WRCR validates the complete ASCR image, then commits it and the translation-cache transition selected by the changed ASCR.ASID or ASCR.AE field atomically.

ECR[63:0]

ECR Format

Table 3-12. ECR Fields

Field	Bits	Meaning
reserved	63..12	must be zero
MAX_EDEPTH	11..8	maximum depth at which a maskable interrupt may be admitted
NMI_P	7	hardware-managed coalesced pending-NMI state while entry is invalid or STATUS.NI is set; WRCR ignores the supplied bit
reserved	6..1	must be zero
V	0	architectural event-entry state is valid



Interrupt-File Control Formats

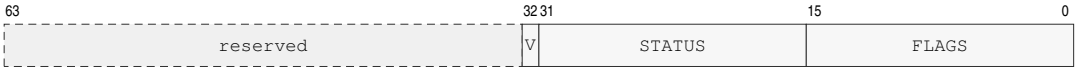
In the ISEL row, `B` abbreviates `ISEL.BANK`.

Each logical processor owns an interrupt file for identities 1 through `ICAP.MAX_ID`; identity zero is reserved. Each implemented identity has one pending bit and one enable bit. Posting a valid identity sets its pending bit, repeated postings coalesce, and posting zero or an identity above `ICAP.MAX_ID` has no effect.

An identity is eligible when both bits are set and `ITHRESH.THRESHOLD` is zero or the identity is numerically less than the threshold. Lower identity numbers have higher priority. `ITOP.TOP` reads as the lowest eligible identity, or zero when none is eligible. `ICAP` and `ITOP` are read-only.

`ISEL.WORD` selects a 64-identity word and `ISEL.BANK` selects its view through `IDATA`. Bank zero reads pending bits and is read-only. Bank one reads zero and writes pending bits with write-one-to-set semantics. Bank two reads zero and clears pending bits with write-one-to-clear semantics. Bank three reads enable bits and replaces the selected enable word on write. A selected word is valid only when it contains at least one implemented nonzero identity. Bits for identity zero or identities above `ICAP.MAX_ID` are reserved.

Writing `ICAP`, `ITOP`, pending bank zero, or an invalid `ISEL.WORD` raises `INVALID_CONTROL_SELECTOR`. Nonzero reserved bits in `ISEL`, `ITHRESH`, or a writable `IDATA` bank raise `RESERVED_CONTROL_BITS`. Validation precedes the write, so a fault changes neither the interrupt file nor its selector. A successful WRCR of `ITHRESH` or `ISEL` atomically replaces the selected threshold or selector. A successful WRCR of `IDATA` atomically applies the selected bank's write-one-to-set, write-one-to-clear, or enable-word replacement operation.

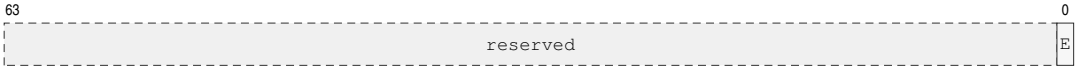


UCTL[63:0]

UCTL Format

Table 3-13. UCTL Fields

Field	Bits	Meaning
reserved	63..33	must be zero
V	32	complete user-return bank contains a valid return context
STATUS	31..16	saved user STATUS image
FLAGS	15..0	saved user FLAGS image



PMC[63:0]

PMC Format

E abbreviates PMC.EN in the diagram.

Table 3-14. PMC Fields

Field	Bits	Meaning
reserved	63..1	must be zero
EN	0	enable performance-counter increments when set

WRCR accepts PMC.EN with every reserved bit zero and atomically replaces PMC. Clearing the field freezes the current performance-counter values without clearing them. Selectors 0x0100 through 0x0102 are reserved.

4 DATA FORMATS

4.1 Integer Data Sizes

Integer operands use size suffixes to select the low-order subfield of an Rn register and the number of bytes transferred by memory operands.

Byte, word, and long operations use the low-order subfield of the selected register. Q-sized operations use the full 64-bit register.

Every Rn destination write defines all 64 bits. A Q-sized write uses the complete result. For a B-, W-, or L-sized write, `ABS`, `DIVS`, `MODS`, `DIVMODS`, `MINS`, `MAXS`, `SAR`, and the explicit `EXTS` operations sign-extend the selected-width result to 64 bits. Every other integer operation, including `MOV`, neutral modular arithmetic, logical and bit operations, count and Boolean results, address results, atomic return values, and state-register reads, zero-extends the selected-width result to 64 bits. This result extension does not change FLAGS calculation, which continues to use the selected operation size.

Memory destination writes transfer only the selected number of bytes. Extension-store instructions instead write the explicitly named W, L, or Q destination width. SP has only Q-sized writable forms. The result-extension rule applies to Rn destinations.

Table 4-1. Integer Data Sizes

Code	Suffix	Bits	Bytes	Name
`B`	`.B`	8	1	Byte
`W`	`.W`	16	2	Word
`L`	`.L`	32	4	Long
`Q`	`.Q`	64	8	Quad

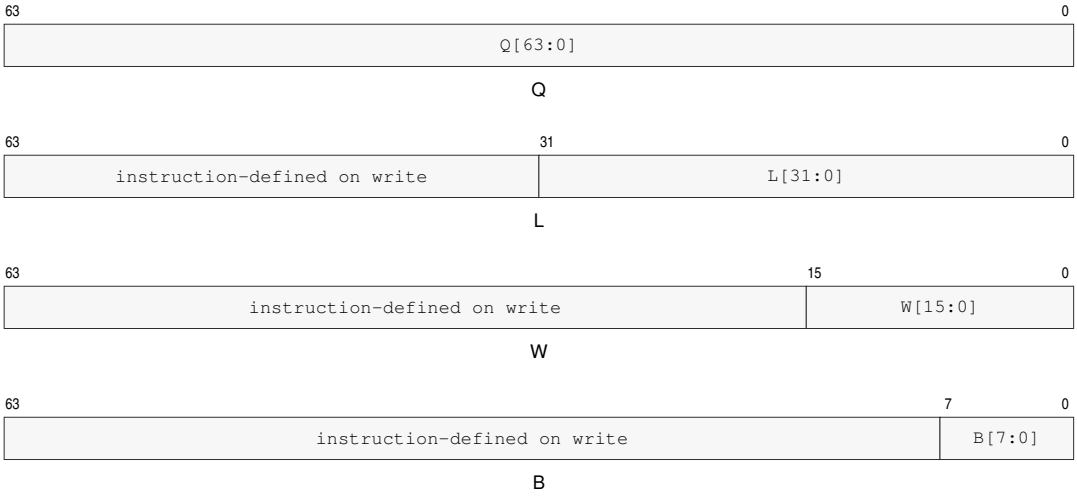


Figure 4-1. Rn Register Operand Subfields

4.2 Little-Endian Memory Organization

Integer data, immediates, displacements, and absolute address payloads are little-endian. Multi-byte numeric payloads are stored least significant byte first. The decoder consumes instruction-header fields byte by byte in instruction-stream order.

For an N-byte integer value stored at byte address A, byte A contains bits 7..0, byte A+1 contains bits 15..8, and so on.

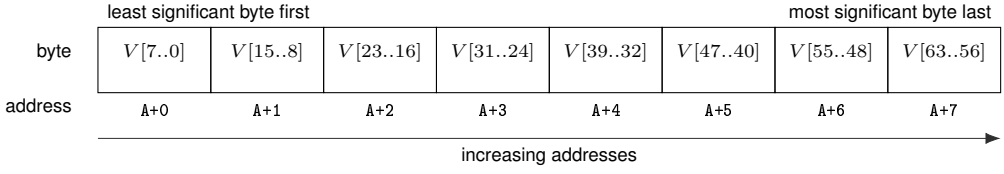


Figure 4-2. Little-Endian Byte Order for a 64-Bit Value

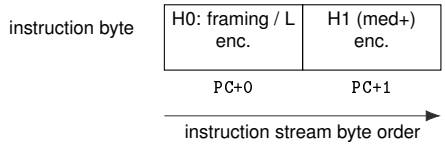


Figure 4-3. Instruction Start Bytes

H0 and H1 denote header bytes 0 and 1; H1 belongs to the header only for medium and longer instructions.

4.3 Floating-Point Data Formats

Table 4-2. Floating-Point Scalar Sizes

Code	Suffix	Bits	Bytes	Name
`H`	`.H`	16	2	Half (conversion and storage)
`S`	`.S`	32	4	Single
`D`	`.D`	64	8	Double

Floating-point scalar formats are little-endian in memory as well. H is IEEE-754 binary16 with 11 bits of significand precision, minimum normal exponent -14, and minimum subnormal quantum 2^{-24}. H is available to the scalar instruction set only as a conversion and storage format; scalar arithmetic remains S/D. S is IEEE-754 binary32 with 24 bits of significand precision, minimum normal exponent -126, and minimum subnormal quantum 2^{-149}. D is IEEE-754 binary64 with 53 bits of significand precision, minimum normal exponent -1022, and minimum subnormal quantum 2^{-1074}.

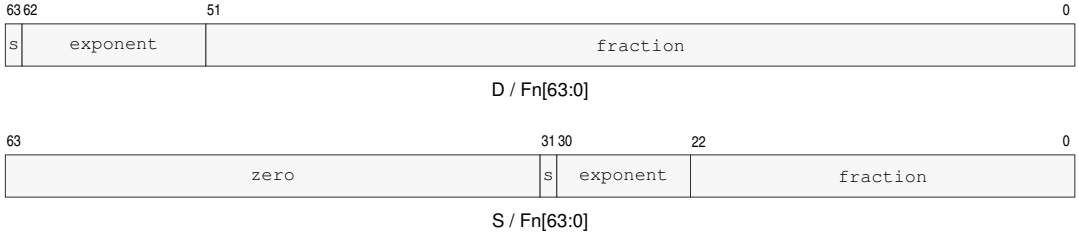


Figure 4-4. Floating-Point Register Encodings

For a NaN, the most significant fraction bit is the quiet bit: bit 22 for S and bit 51 for D. `s` denotes the sign bit. An S result occupies Fn bits 31..0 and writes zero to Fn bits 63..32.

The bit-level floating-point encoding, NaN policy, exception flags, and rounding behavior are defined by the floating-point instruction and state descriptions; their memory byte order follows the same least-significant-byte-first rule as integer data.

PART II

BEDROCK ARCHITECTURE

Encoding, Addressing, and Execution

5 INSTRUCTION ENCODING

5.1 Instruction Stream and Record Boundary

The byte addressed by PC is byte 0 of the current instruction. Instruction decoding determines the complete encoded instruction length before operand evaluation begins.

The opcode space begins at byte 0, includes the header, and ends immediately before the first payload. Any appended EA descriptor, displacement, immediate, bitmap, or instruction-specific value is a payload. Independently selected appended units are separate payloads. Bytes after the required instruction length constitute padding.

The header is byte 0 for extrashort and short instructions and the first two bytes for medium and longer instructions. It determines the encoded instruction length and selects the opcode space before operand evaluation begins. An encoded instruction length that cannot contain the selected encoding raises `TRUNCATED_INSTRUCTION` before any architectural state is changed.

The required instruction length is the opcode-space length plus every required payload length. The encoded instruction length must be at least that large. For an extended instruction, every trailing byte after the required instruction length is uninterpreted padding; every byte value is valid. Padding never extends an operand, payload, descriptor, or opcode space.

Before decode validation or operand evaluation, instruction fetch and permission checks cover the complete encoded record, including padding. An inaccessible padding byte therefore reports the applicable instruction-fetch fault. An undersized record raises `TRUNCATED_INSTRUCTION` before architectural state changes.

5.2 Instruction Header Formats

Instruction framing is byte-oriented. Byte 0 identifies extrashort and short instructions. Extended instructions use byte 0 and byte 1: byte 0 supplies the framing and encoded length, while the remaining bits of those bytes carry the first fixed bits that select the opcode and the first operand-selector fields.

For an extended instruction, byte0[5:2] is `L` and the encoded instruction length is $3 + L$ bytes. The opcode space contains fixed bits that select the opcode and operand-selector fields; it occupies three bytes for medium instructions, four bytes for long instructions, five bytes for extralong instructions, and six bytes for xxlong instructions. An extended instruction is invalid when its encoded length is shorter than the opcode space selected by its header.

`LEN n, <instruction>` requests an encoded extended instruction length of `n` bytes, where `3 <= n <= 18`. The requested length must cover the required instruction length. Without `LEN`, the assembler emits that required length. It fills requested trailing padding with zero. A disassembler emits `LEN n,` when the encoded instruction length exceeds the required instruction length so reassembly preserves the length; padding byte values are not preserved. Extrashort and short encodings retain their fixed lengths.

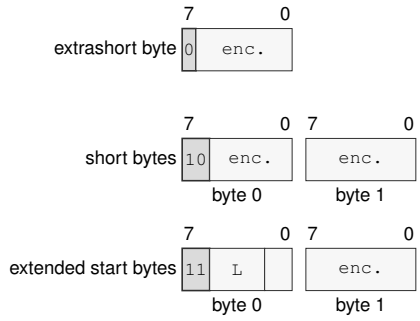


Figure 5-1. Instruction Start Byte Formats

Figure 5-1 is read from high to low bit. `enc.` denotes instruction-specific bits within the opcode space: fixed opcode bits and operand-selector fields. In byte 0, bit 7 selects extrashort framing; bits 7..6 select short (`10`) or extended (`11`) framing. For extended instructions, `EXTENDED_HEADER.L` directly encodes the encoded instruction length as $3 + L$ bytes.

The extended class selector is byte0[1:0] followed by byte1[7:4]. Values 0 through 59 select a medium opcode space, and values 60 through 62 select a long opcode space. When the selector is 63, byte1[3:2] completes an eight-bit class prefix: `1111110x` selects extralong and `11111111` selects xxlong. The `11111110` prefix retains extralong framing but is reserved for custom extensions and is not assigned by the standard ISA.

Table 5-1. Encoded Instruction Length Truth Table

Byte 0 Pattern	Byte 1 Pattern	Bytes
`0xxxxxxx`	–	1
`10xxxxxx`	`xxxxxxxx`	2
`110000xx`	`xxxxxxxx`	3
`110001xx`	`xxxxxxxx`	4
`110010xx`	`xxxxxxxx`	5
`110011xx`	`xxxxxxxx`	6
`110100xx`	`xxxxxxxx`	7
`110101xx`	`xxxxxxxx`	8
`110110xx`	`xxxxxxxx`	9
`110111xx`	`xxxxxxxx`	10
`111000xx`	`xxxxxxxx`	11
`111001xx`	`xxxxxxxx`	12
`111010xx`	`xxxxxxxx`	13
`111011xx`	`xxxxxxxx`	14
`111100xx`	`xxxxxxxx`	15
`111101xx`	`xxxxxxxx`	16
`111110xx`	`xxxxxxxx`	17
`111111xx`	`xxxxxxxx`	18

Table 5-2. Opcode-Space Class Summary

Class	Opcode-Space Bytes	Header Selector	Minimum Bytes	Validity
`extrashort`	byte 0	`0xxxxxxx`	1	exactly 1 byte
`short`	bytes 0–1	`10xxxxxx`	2	exactly 2 bytes
`medium`	bytes 0–2	extended selector 0–59	3	encoded length at least 3 bytes
`long`	bytes 0–3	extended selector 60–62	4	encoded length at least 4 bytes
`extralong`	bytes 0–4	extended selector 63, prefix `1111110x`	5	encoded length at least 5 bytes
`xxlong`	bytes 0–5	all-one eight-bit prefix	6	encoded length at least 6 bytes

5.3 Unavailable and Unassigned Encodings

An unassigned encoding or an encoding marked reserved does not define alternative architectural behavior. Instruction opcodes, extension opcodes, and effective-address forms with that designation, together with encodings for an optional instruction group whose CPUID feature bit is clear, raise `UNAVAILABLE_INSTRUCTION_EXTENSION` during decode.

5.4 Instruction Payload Ordering

The instruction header is byte 0 for extrashort and short instructions and the first two bytes for medium and longer instructions. The opcode space may continue after the header. If a descriptor, immediate, displacement, or bitmap follows the opcode space, each independently determined unit is a separate payload, and the payloads occupy increasing addresses in decode order. All extended EA descriptors precede every other payload, and multiple extended EA descriptors appear in declared instruction operand order. The remaining payloads then appear in declared instruction operand order. An extended EA descriptor and a following displacement are therefore separate payloads, and the displacement follows every extended EA descriptor in the instruction.

Signed displacement and immediate payloads use two's-complement representation at their declared width before any sign extension performed by the consuming instruction or effective-address encoding.

Table 5-3. Immediate Operand Interpretation

Operand	Bits	Value	Range	Extension	Applies When
`PAIRn`	3	identifier	0..7	none	PUSHP/POPP canonical register-pair selector
`pt_level`	3	enumeration	1..4	none	PTQUERY page-table level selector
`flags_bitmap`	4	bitmap	0..15	none	SETF integer FLAGS image
`imm6`	6	unsigned	0..63	zero-extend	integer encodings with an explicit B/W/L/Q operation size
`imm7`	7	unsigned	0..127	zero-extend	EXTRACT register-pair encodings

PUSHP and POPP interpret their 3-bit operand as a canonical register-pair index, so every encoding from 0..7 is valid. PTQUERY uses a 3-bit page-table level; assigned levels are 1..4, while encodings 0 and 5..7 are reserved. SETF interprets its 4-bit bitmap range 0..15 as a complete

FLAGS image: bit 3 writes `FLAGS.Z`, bit 2 writes `FLAGS.N`, bit 1 writes `FLAGS.C`, and bit 0 writes `FLAGS.V`. Each one writes the corresponding flag to one, and each zero writes it to zero.

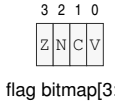


Figure 5-2. SETF Flag Bitmap

A 6-bit `imm6` value in 0..63 is zero-extended to the selected operation size before use when that size is wider than 6 bits. EXTRACT uses the 7-bit `imm7` range 0..127 as the offset into its concatenated register pair.

5.5 Representative Encodings

The following opcode space byte layouts show framing, encoded length, fixed opcode bits, and operand-selector fields in instruction-stream order. Each row is read from bit 7 to bit 0.

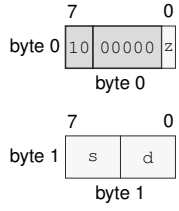


Figure 5-3. Short Encoding for MOV.X(z:L/Q) Rn(s), Rn(d)

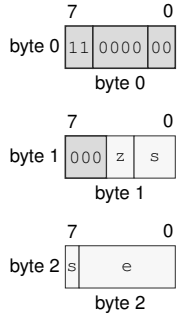


Figure 5-4. Medium Encoding for MOV.X(z:B/W/L/Q) Rn(s), <ea>(e)

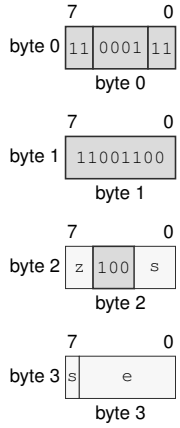


Figure 5-5. Long Encoding for ADC.X(z:B/W/L/Q) Rn(s), <ea>(e)

In these examples, `z` selects the operation size, `s` and `d` select integer registers, and `e` selects an effective-address form. A field may cross a byte boundary, as `s` does in the medium and long examples.

6 EFFECTIVE-ADDRESS PROFILES

EA is the base seven-bit compact effective-address profile selected by the consuming instruction's operand type. Extensions may define additional profiles that reuse the memory address calculation rules and EXT1/EXT2 descriptor grammar in this section while assigning profile-specific compact forms. An effective-address field is an operand descriptor with an instruction-defined role and interpretation width. A value role reads or writes the selected register, immediate, or memory value. An address role uses a register or immediate as an address value and uses a memory-form EA as the calculated address without an initial data read. A control-target role uses a register or immediate value directly and reads an interpretation-width target from a memory form. Memory forms continue into segment pre-translation and paging.

Each compact field is embedded in the instruction's opcode space, and all compact fields precede any appended effective-address payload bytes. Any displacement, absolute address, immediate, or extended descriptor bytes are appended as separate payloads. All extended descriptor bytes appear first, ordered according to the declared instruction operand order of the operands that select them. All remaining effective-address payloads follow, also in declared instruction operand order. For two payload-bearing operands, the encoded stream contains the opcode space for both operands, any extended descriptor for operand 0, any extended descriptor for operand 1, the remaining operand 0 payloads, the remaining operand 1 payloads, and any trailing padding, in that order.

Multi-byte payloads, such as displacements, absolute addresses, and immediates, are little-endian. Payload names ending in s are signed and are sign-extended before use; payload names without s are unsigned unless the consuming instruction explicitly says otherwise. Extended descriptor bytes appear in stream order, byte 0 first. Each descriptor byte is independently numbered from bit 7 through bit 0.

Indexed scale is the EA interpretation width declared by the consuming instruction encoding. B, W, L, and Q widths select scales 1, 2, 4, and 8. An encoding without a fixed width uses its selected instruction size. LEA and SEGLEA use their suffix; size-less instructions with address or control-target roles use Q.

For every compact or extended-descriptor PC-based EA, the PC term is the architectural PC naming byte 0 of the current instruction. The PC term remains that instruction-start value throughout operand evaluation.

6.1 Compact Profile Encodings

The scalar EA profile encodes common Rn/SP/PC memory operands, absolute memory operands, integer immediate operands, and the EXT1 and EXT2 escapes. An extension-defined profile consists only of the forms explicitly declared by that extension; an unassigned selector has no form in that profile.

Compact memory forms that do not name SP or PC use the operation default data segment. SP memory forms are fixed to SS, and PC memory forms are fixed to CS.

Table 6-1. Compact Effective-Address Profile Encoding

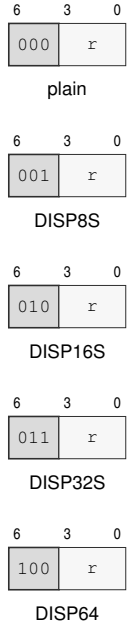
Encoding	EA form
000rrrr (0x00--0x0F)	[Rn(r)]
001rrrr (0x10--0x1F)	[Rn(r) + disp8s]
010rrrr (0x20--0x2F)	[Rn(r) + disp16s]
011rrrr (0x30--0x3F)	[Rn(r) + disp32s]
100rrrr (0x40--0x4F)	[Rn(r) + disp64]
1010000 (0x50)	[SP + disp8s]
1010001 (0x51)	[SP + disp16s]
1010010 (0x52)	[SP + disp32s]
1010011 (0x53)	[SP + disp64]
1010100 (0x54)	[PC + disp8s]
1010101 (0x55)	[PC + disp16s]
1010110 (0x56)	[PC + disp32s]
1010111 (0x57)	[PC + disp64]
1011000 (0x58)	[SP]
1011001 (0x59)	[abs32s]
1011010 (0x5A)	[abs64]
1011011 (0x5B)	imm8s
1011100 (0x5C)	imm16s
1011101 (0x5D)	imm32s
1011110 (0x5E)	imm64
1011111 (0x5F)	EXT1/disp8s
1100000 (0x60)	EXT1/disp16s
1100001 (0x61)	EXT1/disp32s
1100010 (0x62)	EXT1/disp64
1100011 (0x63)	EXT1
1100100 (0x64)	EXT2/disp8s
1100101 (0x65)	EXT2/disp16s
1100110 (0x66)	EXT2/disp32s
1100111 (0x67)	EXT2/disp64
1101000 (0x68)	EXT2
0x69--0x7F	reserved

Scalar compact mode details. The following diagrams define the scalar compact forms selected by the preceding encoding table.

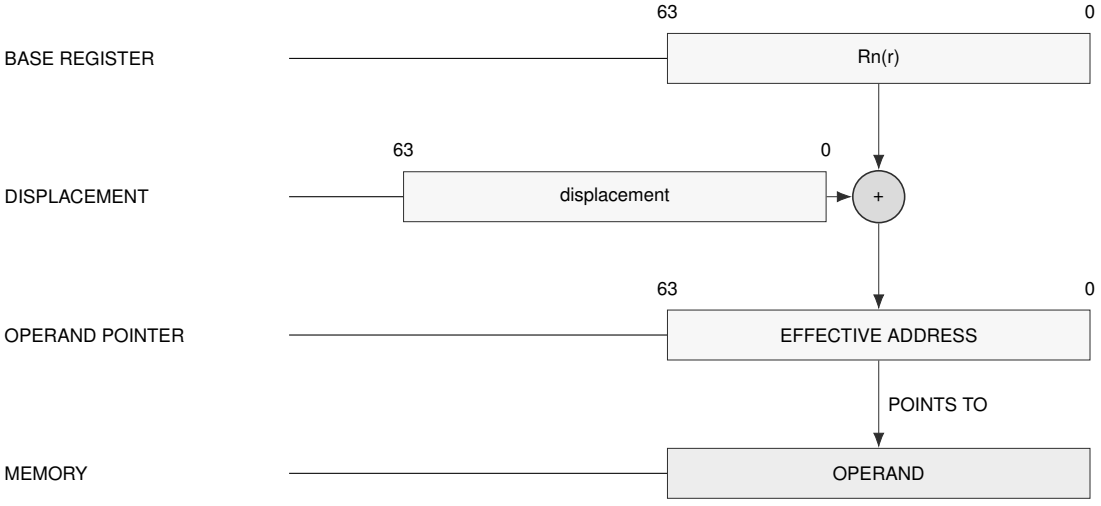
Compact Register Memory

`[Rn(r)]`
`[Rn(r) + disp8s]`
`[Rn(r) + disp16s]`
`[Rn(r) + disp32s]`
`[Rn(r) + disp64]`

EA encoding: `000rrrr`; `001rrrr` (disp8s); `010rrrr` (disp16s); `011rrrr` (disp32s); `100rrrr` (disp64).
Segment: Uses the operation default data segment.
Payload: Appends `disp8s` (displacement), `disp16s` (displacement), `disp32s` (displacement), `disp64` (displacement) as selected by the encoding.



Register Memory Encodings

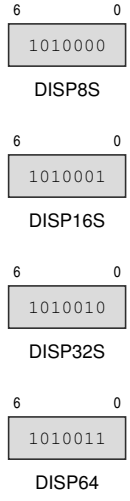


Register Memory Address Generation

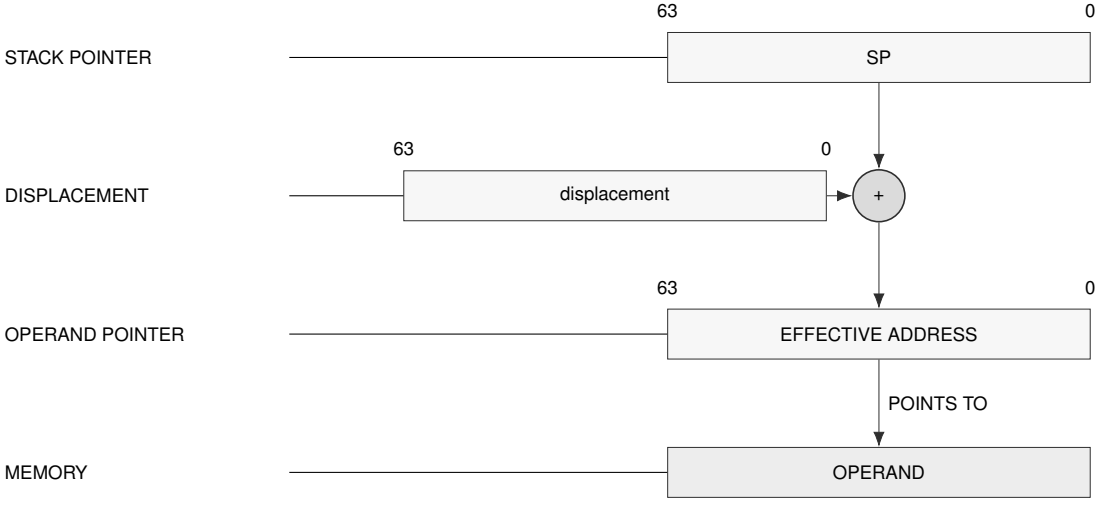
Compact Stack-Pointer Displaced Memory

`[SP + disp8s]`
`[SP + disp16s]`
`[SP + disp32s]`
`[SP + disp64]`

EA encoding: `1010000` (disp8s); `1010001` (disp16s); `1010010` (disp32s); `1010011` (disp64).
Segment: Fixed to SS.
Payload: Appends `disp8s` (displacement), `disp16s` (displacement), `disp32s` (displacement), `disp64` (displacement) as selected by the encoding.



Stack-Pointer Displaced Memory Encodings

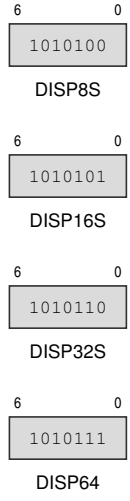


Stack-Pointer Displaced Memory Address Generation

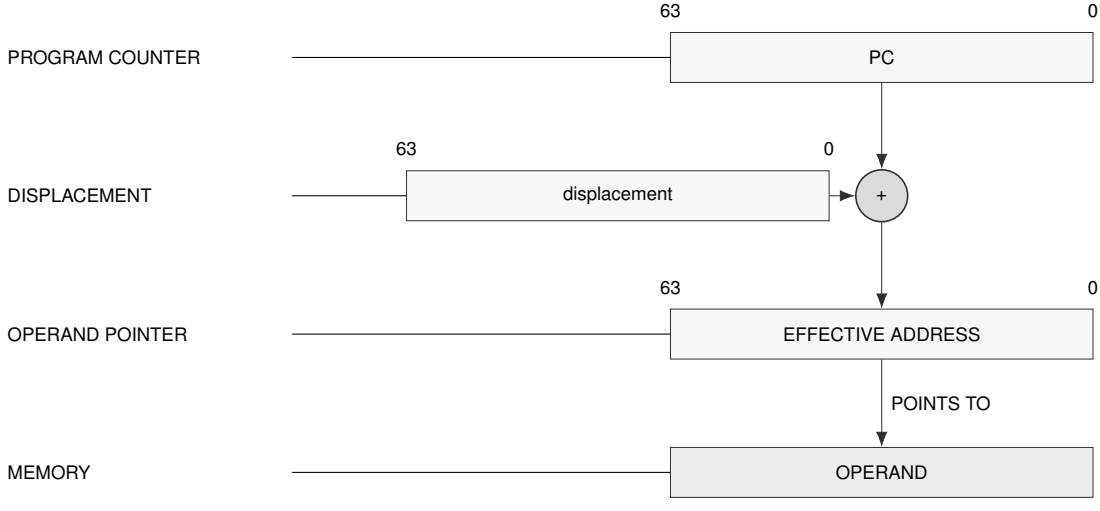
Compact Program-Counter Displaced Memory

`[PC + disp8s]`
`[PC + disp16s]`
`[PC + disp32s]`
`[PC + disp64]`

EA encoding: `1010100` (disp8s); `1010101` (disp16s); `1010110` (disp32s); `1010111` (disp64).
Segment: Fixed to CS.
Payload: Appends `disp8s` (displacement), `disp16s` (displacement), `disp32s` (displacement), `disp64` (displacement) as selected by the encoding.



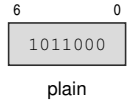
Program-Counter Displaced Memory Encodings



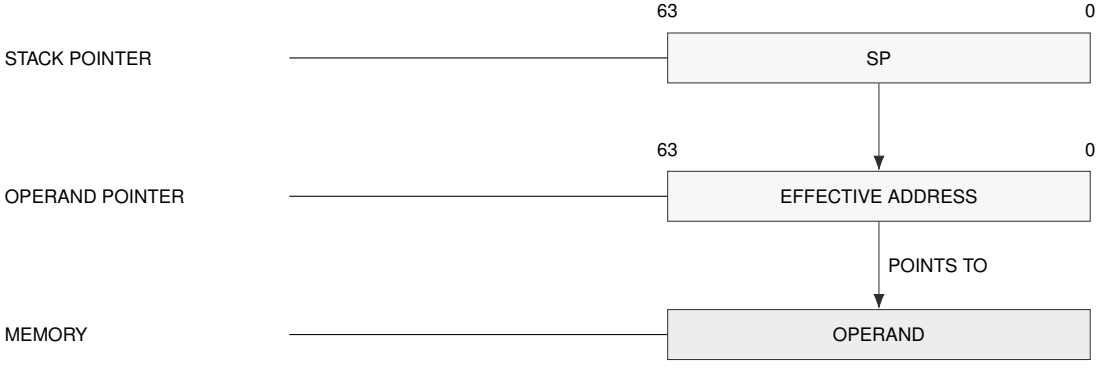
Program-Counter Displaced Memory Address Generation

Compact Stack Pointer Indirect

`[SP]`
EA encoding: `1011000`.
Segment: Fixed to SS.
Payload: No appended payload bytes.



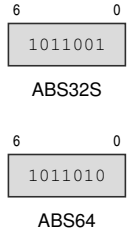
Stack Pointer Indirect Encodings



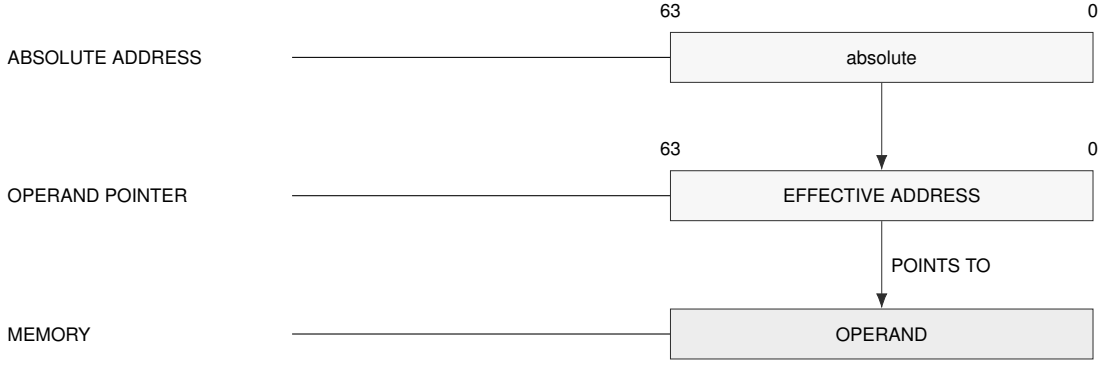
Stack Pointer Indirect Address Generation

Compact Absolute Memory

`[abs32s]`
`[abs64]`
EA encoding: `1011001` (abs32s); `1011010` (abs64).
Segment: Uses the operation default data segment.
Payload: Appends `abs32s` (absolute), `abs64` (absolute) as selected by the encoding.



Absolute Memory Encodings

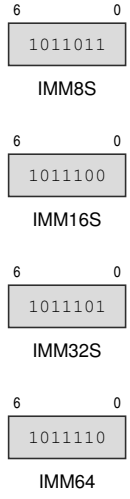


Absolute Memory Address Generation

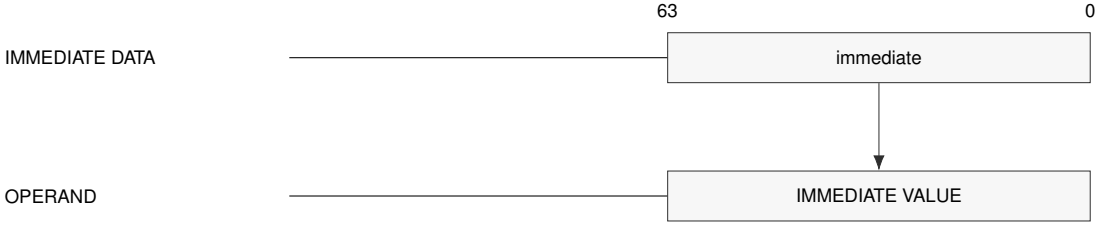
Compact Integer Immediate

`imm8s`
`imm16s`
`imm32s`
`imm64`

EA encoding: `1011011` (imm8s); `1011100` (imm16s); `1011101` (imm32s); `1011110` (imm64).
Segment: Uses the operation default data segment.
Payload: Appends `imm8s` (immediate), `imm16s` (immediate), `imm32s` (immediate), `imm64` (immediate) as selected by the encoding.



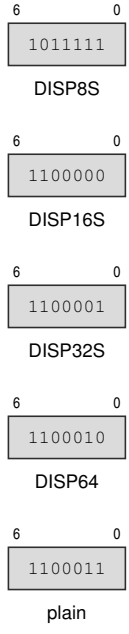
Integer Immediate Encodings



Integer Immediate Operand Generation

Compact EXT1

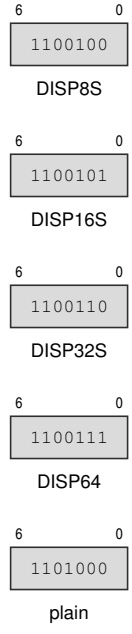
EA encoding: `1011111` (disp8s); `1100000` (disp16s); `1100001` (disp32s); `1100010` (disp64); `1100011`.
Segment: Uses the operation default data segment.
Payload: Appends `disp8s` (displacement), `disp16s` (displacement), `disp32s` (displacement), `disp64` (displacement) as selected by the encoding.



EXT1 Encodings

Compact EXT2

EA encoding: `1100100` (disp8s); `1100101` (disp16s); `1100110` (disp32s); `1100111` (disp64); `1101000`.
Segment: Uses the operation default data segment.
Payload: Appends `disp8s` (displacement), `disp16s` (displacement), `disp32s` (displacement), `disp64` (displacement) as selected by the encoding.



EXT2 Encodings

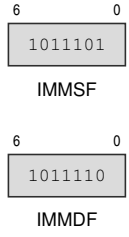
6.1.1 Floating-Point FEA Immediate Forms

FEA directly declares the register, SP-displaced, PC-displaced, absolute, EXT1, and EXT2 selectors defined for this profile. Selectors `0x58`, `0x5B`, and `0x5C` are reserved. Selectors `0x5D` and `0x5E` select `immsf` and `immdf`. The former appends four bytes and the latter eight bytes.

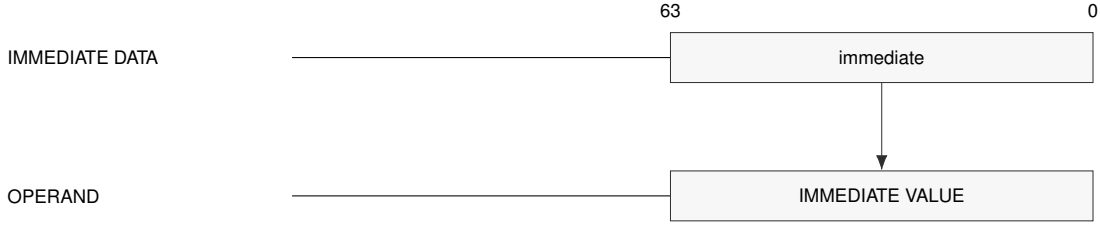
`immsf` carries the exact IEEE 754 binary32 interchange encoding and `immdf` carries the exact binary64 interchange encoding. Either source form is valid for either an S or D instruction. The payload format defines the source format and the instruction suffix defines the operation format. When they differ, operand evaluation converts the immediate to the operation format under the architectural IEEE 754 environment before the instruction operation. The conversion uses `FSTATUS.RM`; its floating-point exception causes are combined with the operation's causes and are tested and committed atomically under the ordinary precise floating-point fault rules. An FEA floating-point immediate is never a valid destination.

FP FEA Compact Floating-Point Immediate

`immsf`
`immdf`
EA encoding: `1011101` (immsf); `1011110` (immdf).
Segment: Uses the operation default data segment.
Payload: Appends `immsf` (immediate), `immdf` (immediate) as selected by the encoding.



Floating-Point Immediate Encodings



Floating-Point Immediate Operand Generation

6.1.2 Vector VEA Lane Addressing

VEA directly declares the register, SP-displaced, PC-displaced, absolute, EXT1, and EXT2 selectors defined for this profile. Selectors `0x58` and `0x5B` through `0x5E` are reserved.

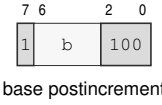
For an ordinary compact or EXT VEA memory form, address calculation first produces one scalar anchor using the shared base, scalar Rn index, scale, displacement, segment, and auto-update grammar. If E is the vector element size in bytes, lane n accesses $\mathrm{anchor} + nE$. Separate vector instructions define scatter/gather addressing outside the VEA descriptor forms.

For a VEA auto-update form, a base-register update changes the base by $\pm\mathrm{VLEN}/8$ bytes. A scalar Rn index update changes the index by $\pm$the instruction's number of elements before the element-size scale is applied. Predicate activity does not change either update quantity. Descriptor encodings and update ordering otherwise follow the base extended-EA rules.

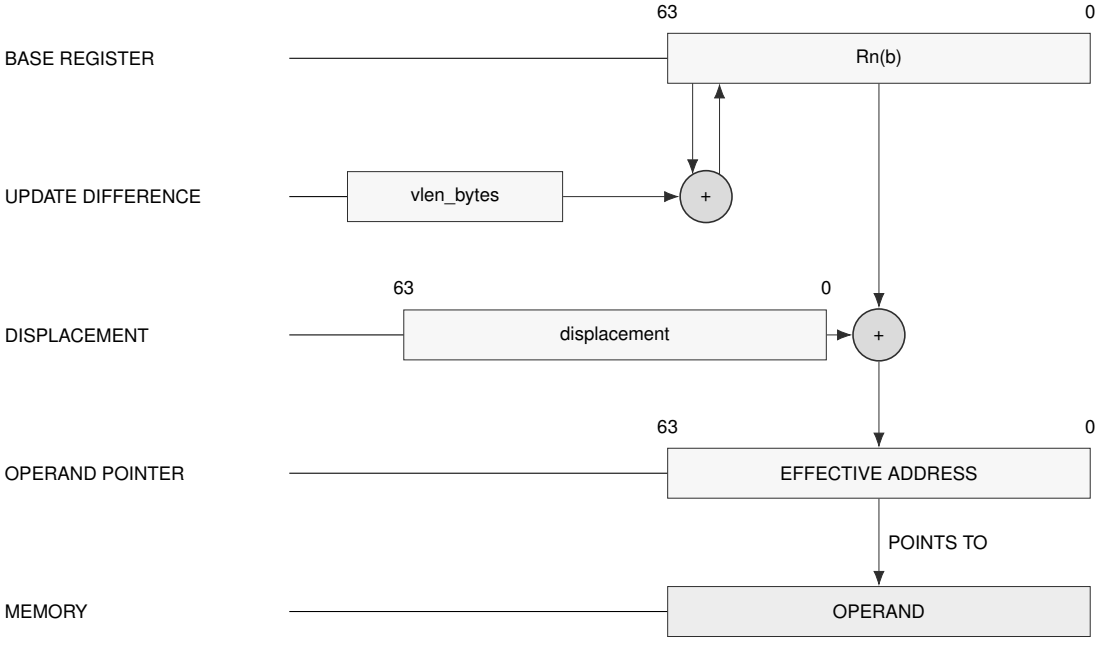
VEA EXT1 Mode

VECTOR VEA EXT1 Default-Segment Base with Autoupdate / Postincrement

`[Rn(b)++ + displacement]`
Descriptor: `1bbbb100` (base postincrement).
Segment: Uses the operation default data segment.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Postincrement uses the current temporary base register, then adds `vlen_bytes`.



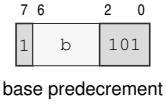
Default-Segment Base with Autoupdate / Postincrement Encodings



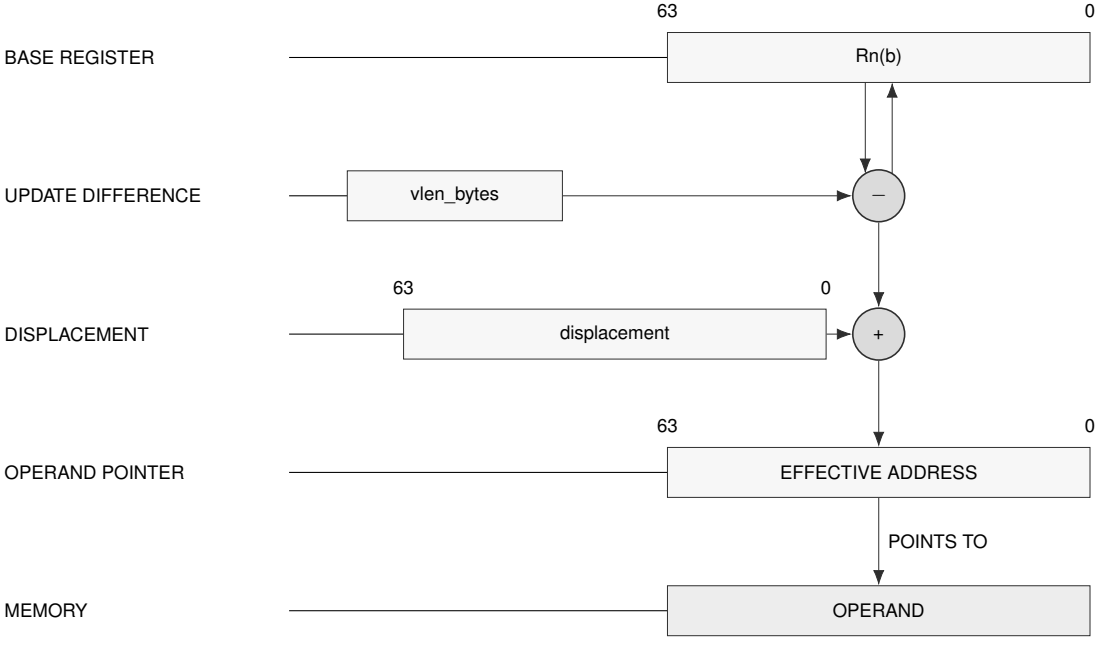
Default-Segment Base with Autoupdate / Postincrement Address Generation

VECTOR VEA EXT1 Default-Segment Base with Autoupdate / Predecrement

`[--Rn(b) + displacement]`
Descriptor: `1bbbb101` (base predecrement).
Segment: Uses the operation default data segment.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Predecrement subtracts `vlen_bytes` from the temporary base register before use.



Default-Segment Base with Autoupdate / Predecrement Encodings

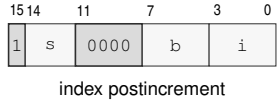


Default-Segment Base with Autoupdate / Predecrement Address Generation

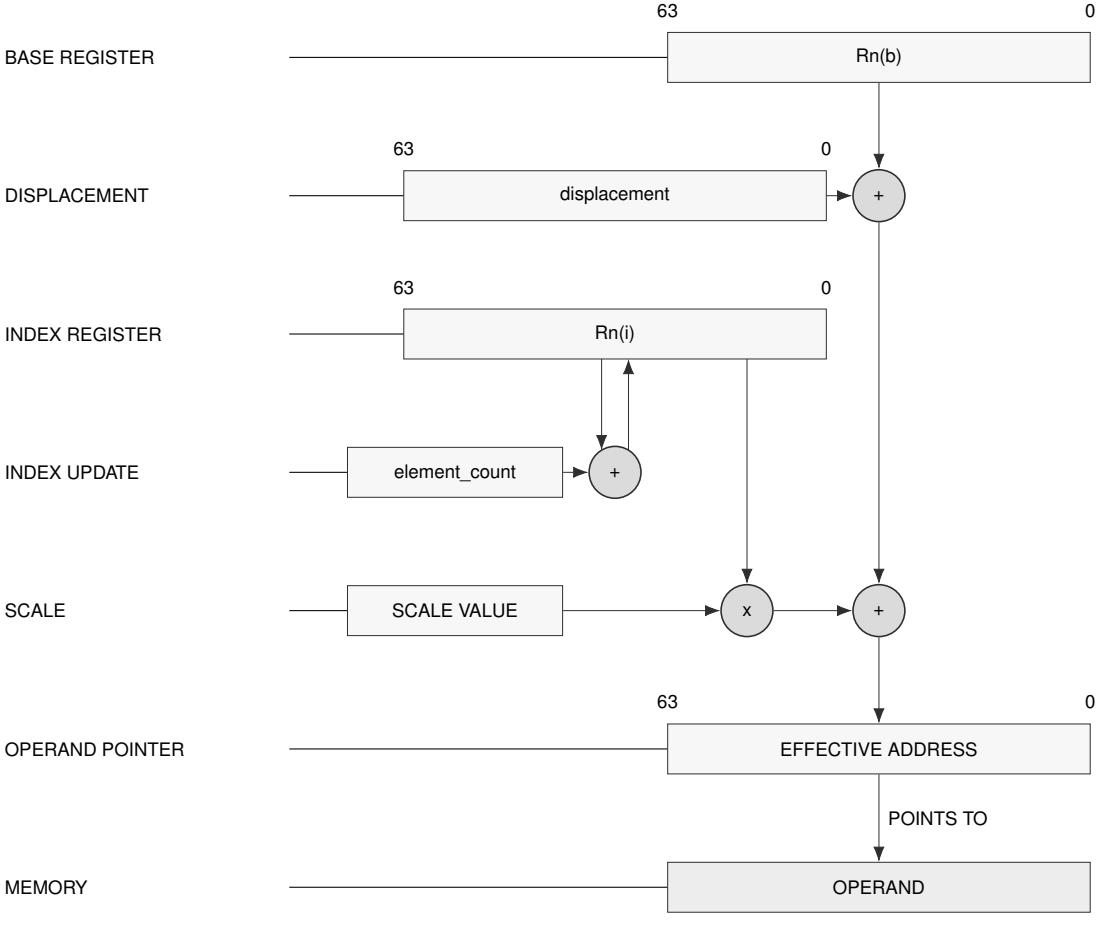
VEA EXT2 Modes

VECTOR VEA EXT2 Explicit-Segment Indexed / Postincrement

`[SEG(s):Rn(b) + Rn(i)++ * scale + displacement]`
Descriptor: `1sss0000 bbbbiiii` (index postincrement).
Segment: SEG(s) selects the encoded segment register.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Postincrement uses the current temporary index register, then adds `element_count`.



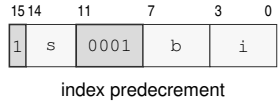
Explicit-Segment Indexed / Postincrement Encodings



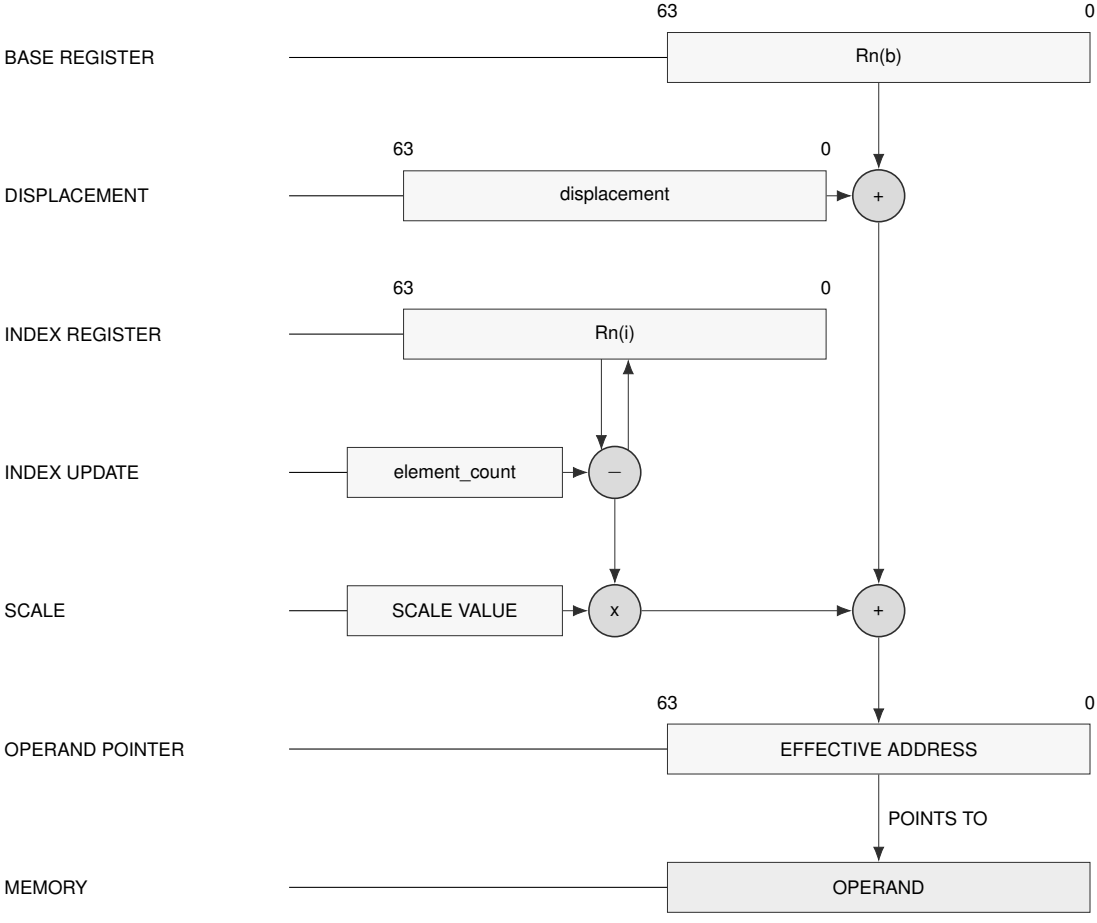
Explicit-Segment Indexed / Postincrement Address Generation

VECTOR VEA EXT2 Explicit-Segment Indexed / Predecrement

`[SEG(s):Rn(b) + --Rn(i) * scale + displacement]`
Descriptor: `1sss0001 bbbbiiii` (index predecrement).
Segment: SEG(s) selects the encoded segment register.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Predecrement subtracts `element_count` from the temporary index register before use.



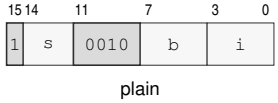
Explicit-Segment Indexed / Predecrement Encodings



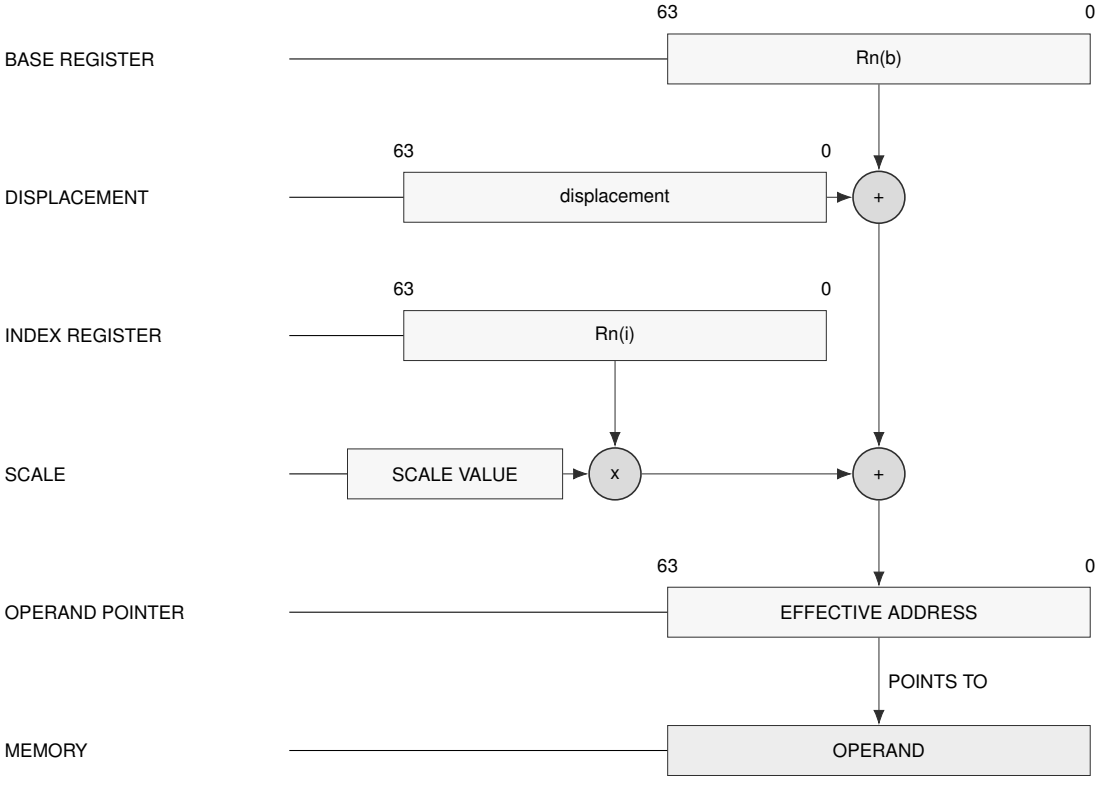
Explicit-Segment Indexed / Predecrement Address Generation

VECTOR VEA EXT2 Explicit-Segment Indexed / Plain

`[SEG(s):Rn(b) + Rn(i) * scale + displacement]`
Descriptor: `1sss0010 bbbbiiii.`
Segment: SEG(s) selects the encoded segment register.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.



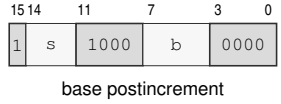
Explicit-Segment Indexed / Plain Encodings



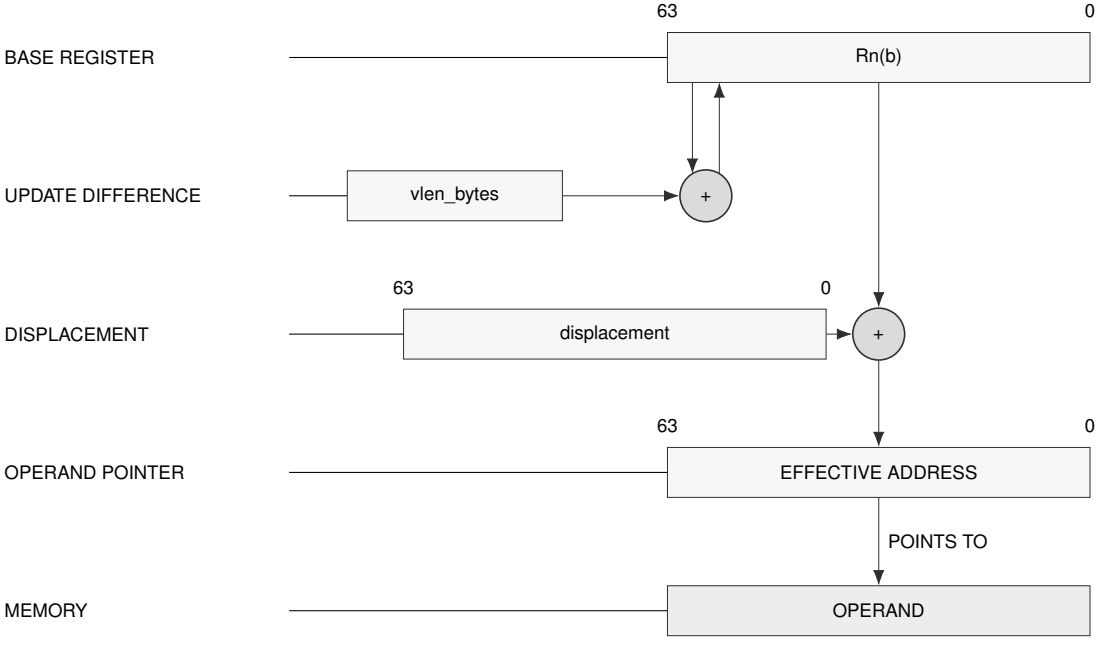
Explicit-Segment Indexed / Plain Address Generation

VECTOR VEA EXT2 Explicit-Segment Base with Autoupdate / Postincrement

`[SEG(s):Rn(b)++ + displacement]`
Descriptor: `1sss1000 bbbb0000` (base postincrement).
Segment: SEG(s) selects the encoded segment register.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Postincrement uses the current temporary base register, then adds `vlen_bytes`.



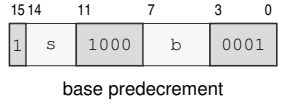
Explicit-Segment Base with Autoupdate / Postincrement Encodings



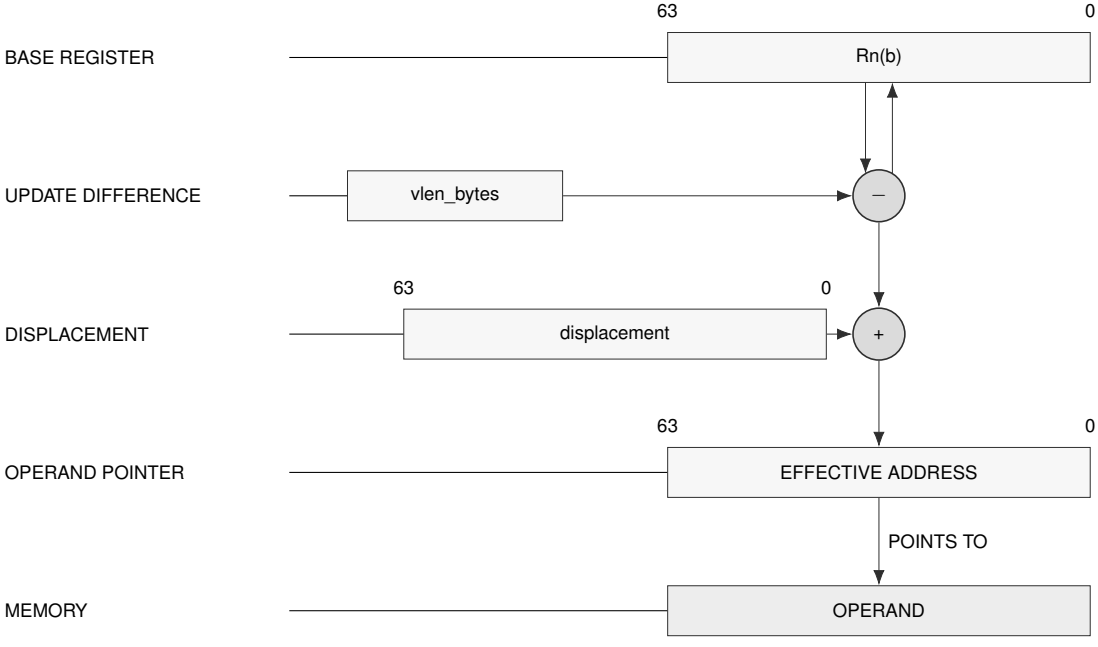
Explicit-Segment Base with Autoupdate / Postincrement Address Generation

VECTOR VEA EXT2 Explicit-Segment Base with Autoupdate / Predecrement

`[SEG(s):--Rn(b) + displacement]`
Descriptor: `1sss1000 bbbb0001` (base predecrement).
Segment: SEG(s) selects the encoded segment register.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Predecrement subtracts `vlen_bytes` from the temporary base register before use.



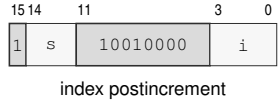
Explicit-Segment Base with Autoupdate / Predecrement Encodings



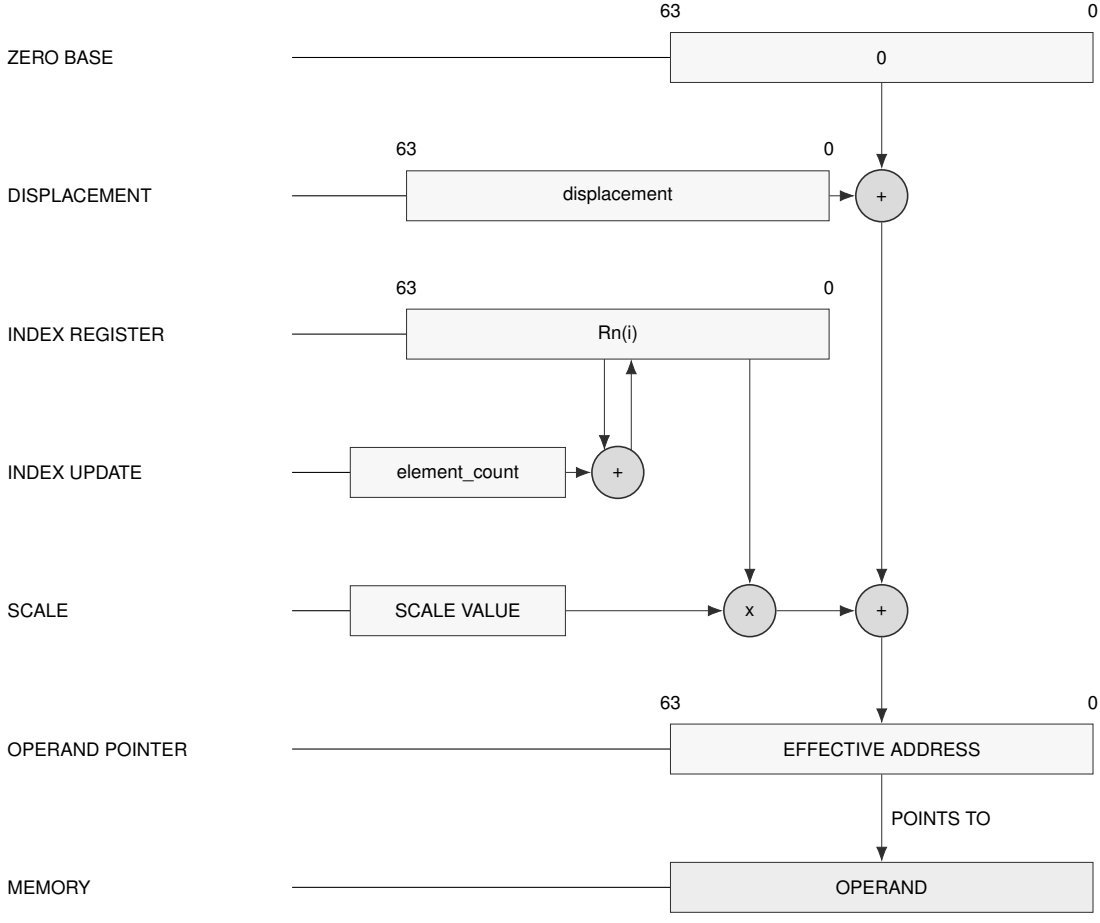
Explicit-Segment Base with Autoupdate / Predecrement Address Generation

VECTOR VEA EXT2 Explicit-Segment Zero-Base Indexed / Postincrement

`[SEG(s):0 + Rn(i)++ * scale + displacement]`
Descriptor: `1sss1001 0000iiii` (index postincrement).
Segment: SEG(s) selects the encoded segment register.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Postincrement uses the current temporary index register, then adds `element_count`.



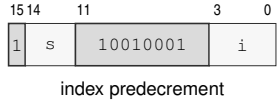
Explicit-Segment Zero-Base Indexed / Postincrement Encodings



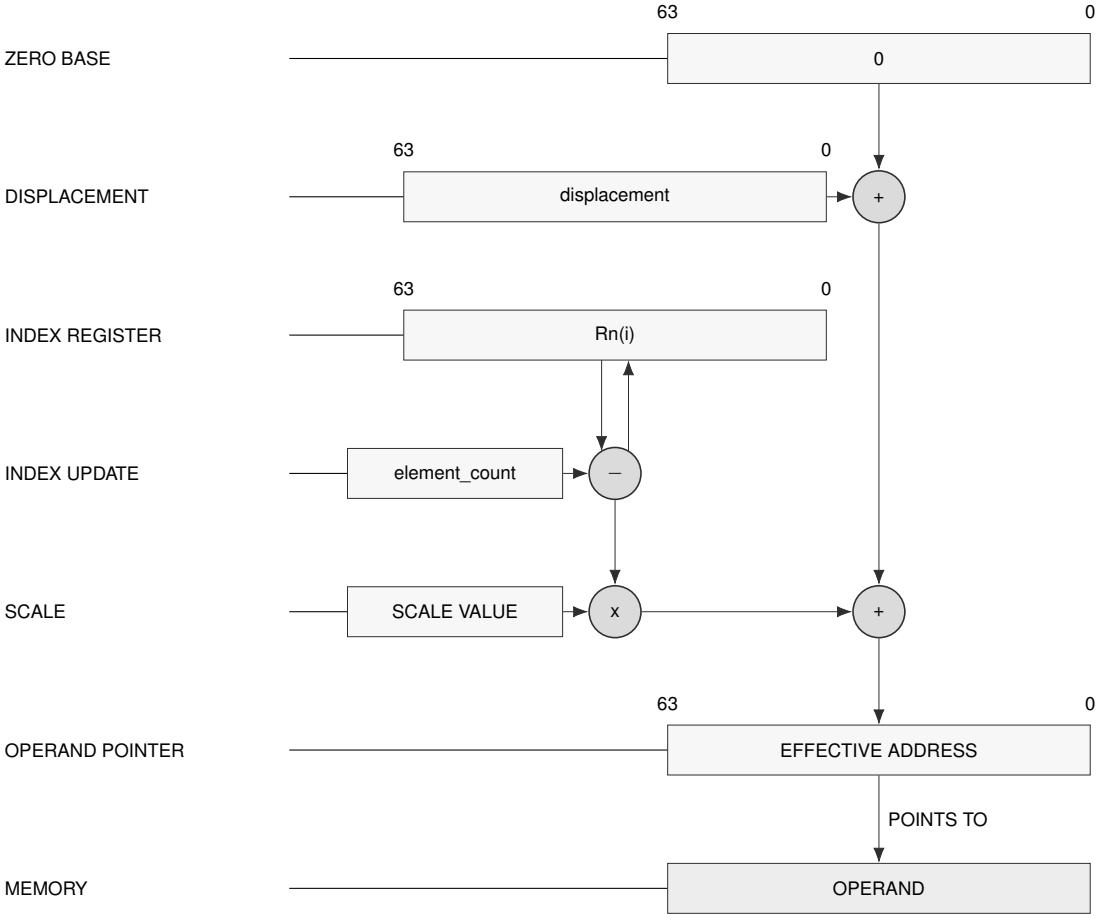
Explicit-Segment Zero-Base Indexed / Postincrement Address Generation

VECTOR VEA EXT2 Explicit-Segment Zero-Base Indexed / Predecrement

`[SEG(s):0 + --Rn(i) * scale + displacement]`
Descriptor: `1sss1001 0001iiii` (index predecrement).
Segment: SEG(s) selects the encoded segment register.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Predecrement subtracts `element_count` from the temporary index register before use.



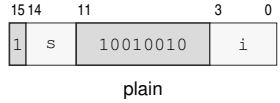
Explicit-Segment Zero-Base Indexed / Predecrement Encodings



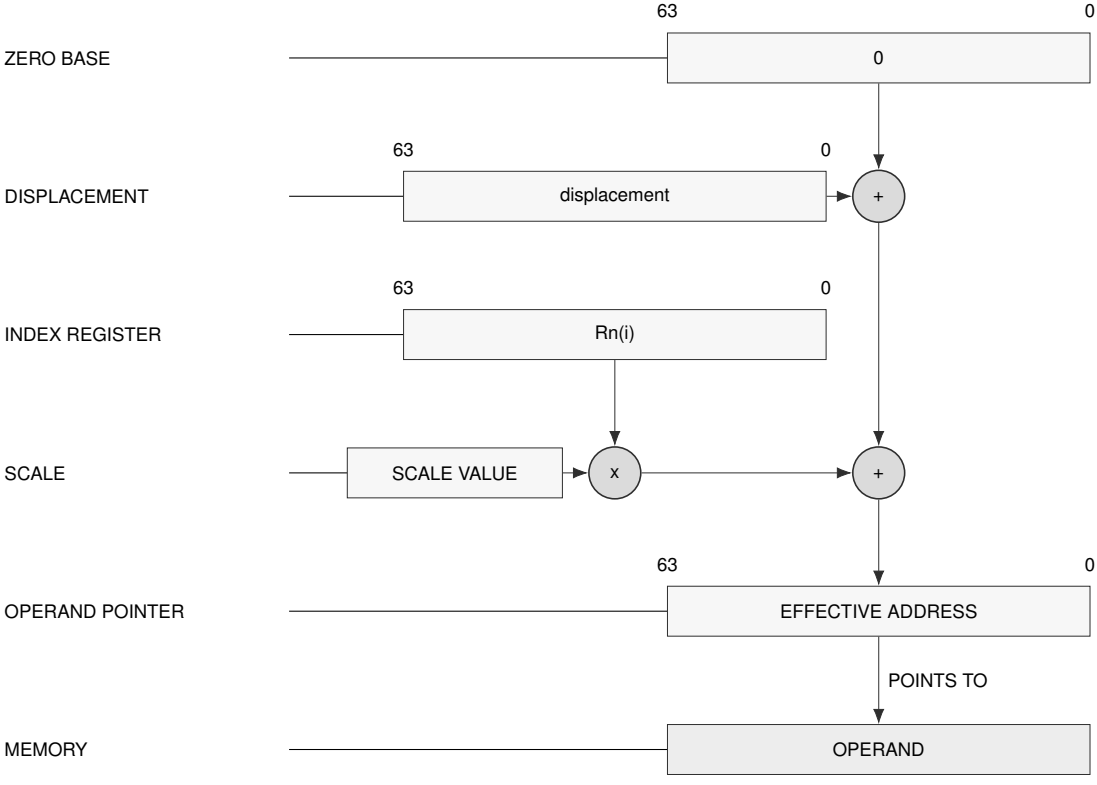
Explicit-Segment Zero-Base Indexed / Predecrement Address Generation

VECTOR VEA EXT2 Explicit-Segment Zero-Base Indexed / Plain

`[SEG(s):0 + Rn(i) * scale + displacement]`
Descriptor: `1sss1001 0010iiii`.
Segment: SEG(s) selects the encoded segment register.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.



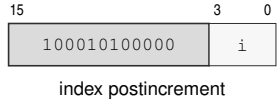
Explicit-Segment Zero-Base Indexed / Plain Encodings



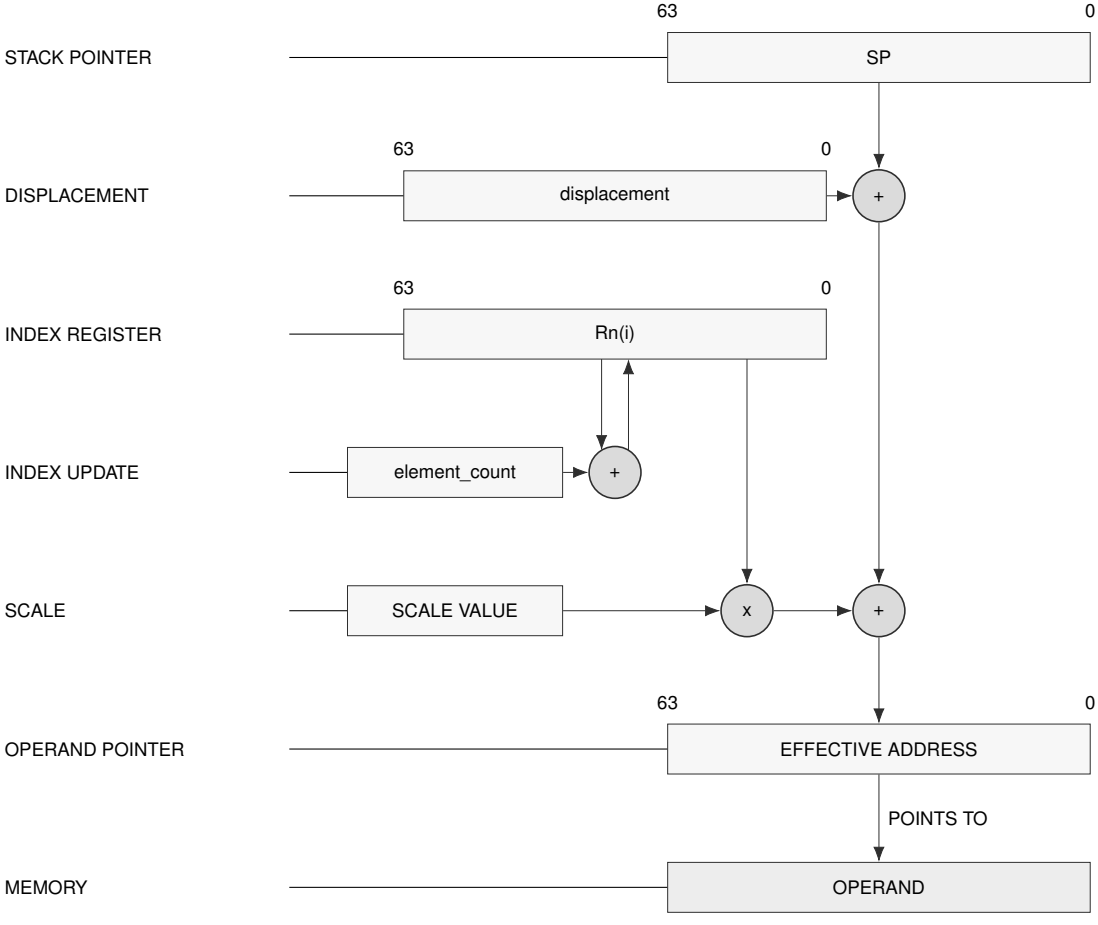
Explicit-Segment Zero-Base Indexed / Plain Address Generation

VECTOR VEA EXT2 Stack-Pointer Indexed / Postincrement

`[SP + Rn(i)++ * scale + displacement]`
Descriptor: `10001010 0000iiii` (index postincrement).
Segment: Fixed to SS.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Postincrement uses the current temporary index register, then adds `element_count`.



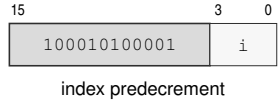
Stack-Pointer Indexed / Postincrement Encodings



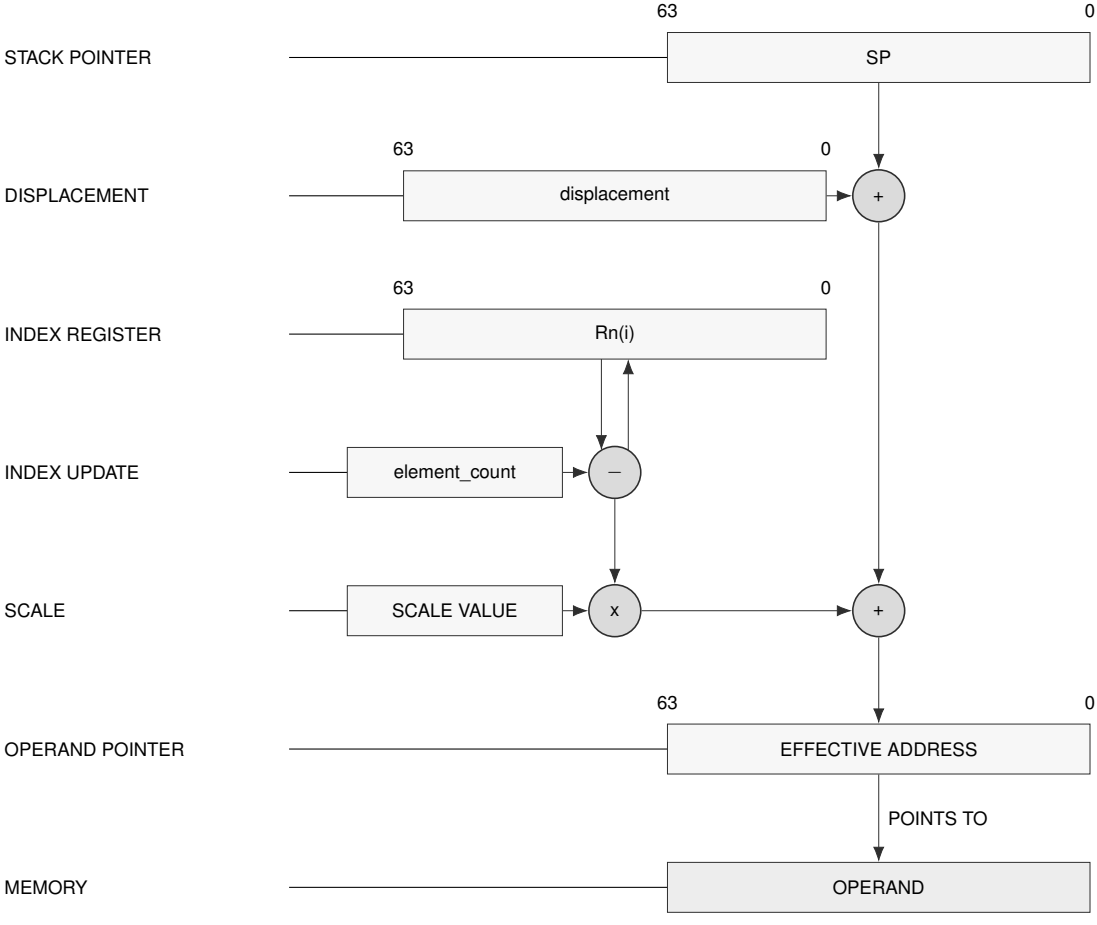
Stack-Pointer Indexed / Postincrement Address Generation

VECTOR VEA EXT2 Stack-Pointer Indexed / Predecrement

`[SP + --Rn(i) * scale + displacement]`
Descriptor: `10001010 0001iiii` (index predecrement).
Segment: Fixed to SS.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Predecrement subtracts `element_count` from the temporary index register before use.



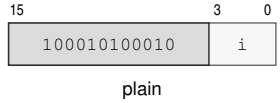
Stack-Pointer Indexed / Predecrement Encodings



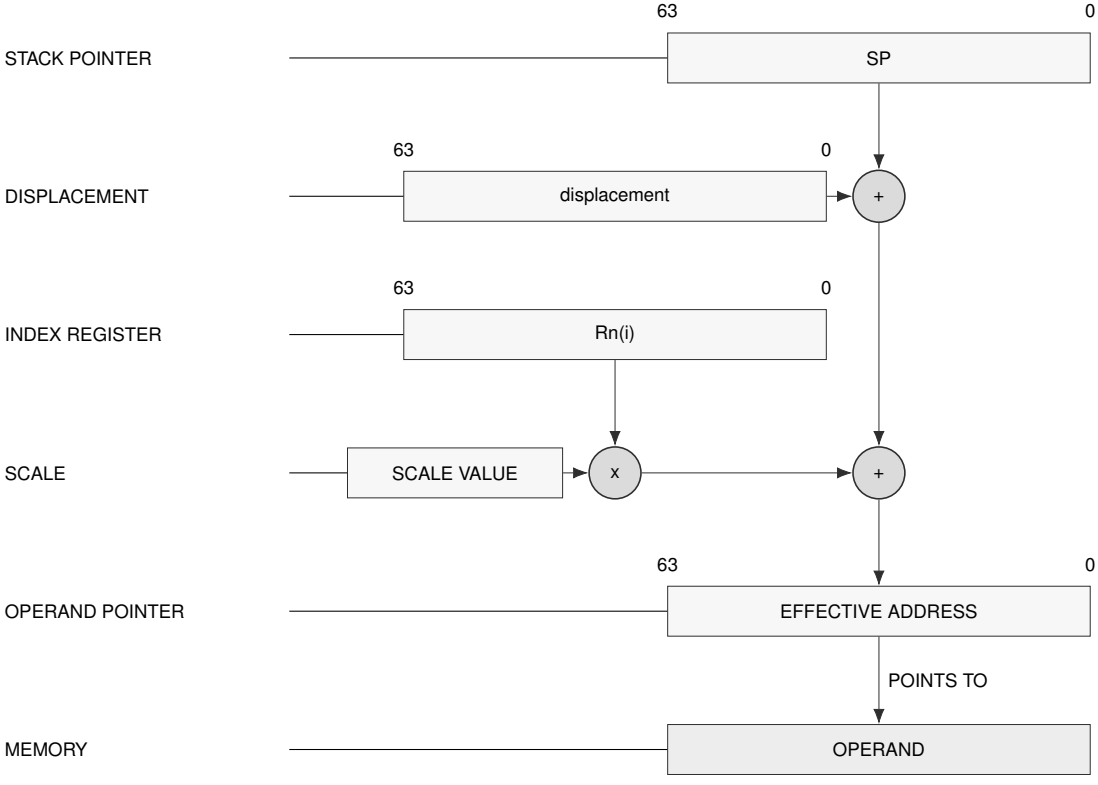
Stack-Pointer Indexed / Predecrement Address Generation

VECTOR VEA EXT2 Stack-Pointer Indexed / Plain

`[SP + Rn(i) * scale + displacement]`
Descriptor: `10001010 0010iiii`.
Segment: Fixed to SS.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.



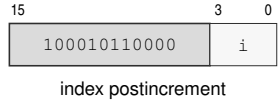
Stack-Pointer Indexed / Plain Encodings



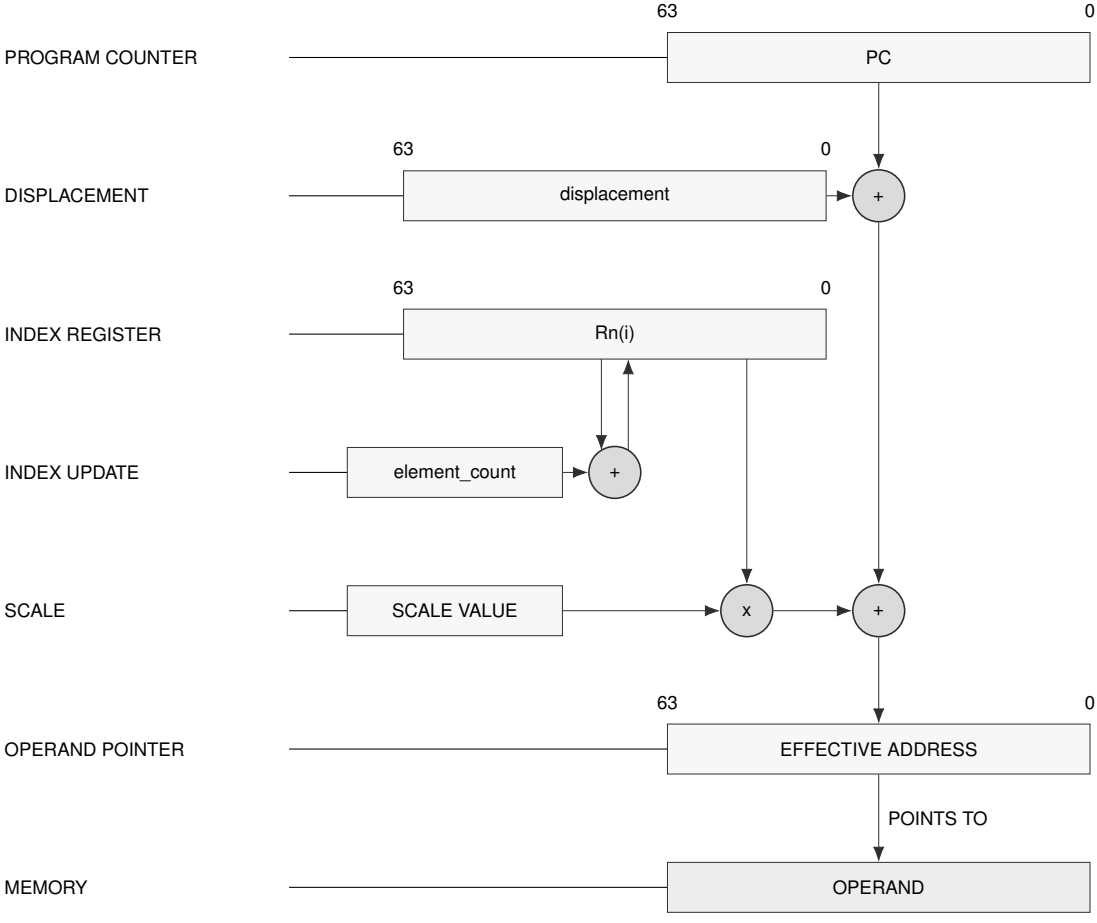
Stack-Pointer Indexed / Plain Address Generation

VECTOR VEA EXT2 Program-Counter Indexed / Postincrement

`[PC + Rn(i)++ * scale + displacement]`
Descriptor: `10001011 0000iiii` (index postincrement).
Segment: Fixed to CS.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Postincrement uses the current temporary index register, then adds `element_count`.



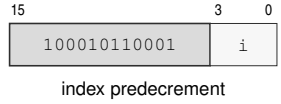
Program-Counter Indexed / Postincrement Encodings



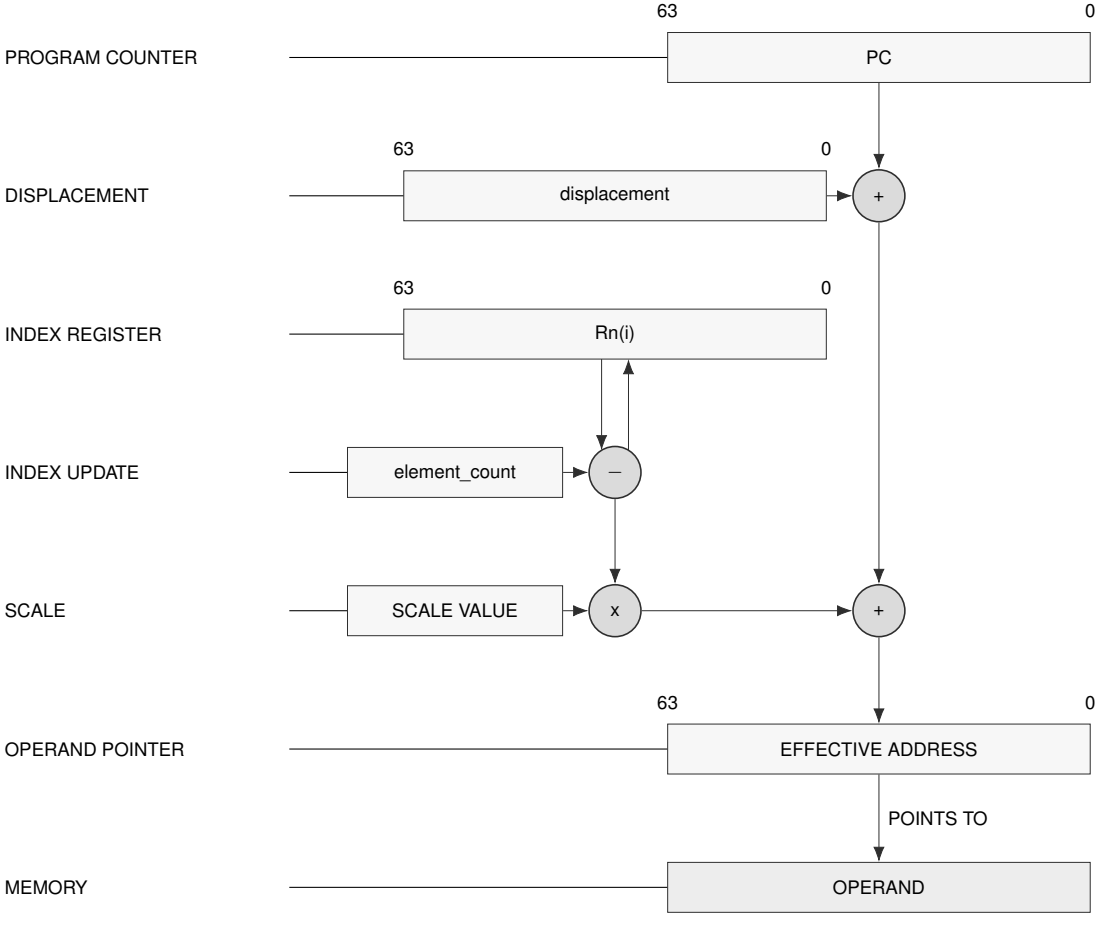
Program-Counter Indexed / Postincrement Address Generation

VECTOR VEA EXT2 Program-Counter Indexed / Predecrement

`[PC + --Rn(i) * scale + displacement]`
Descriptor: `10001011 0001iiii` (index predecrement).
Segment: Fixed to CS.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Predecrement subtracts `element_count` from the temporary index register before use.



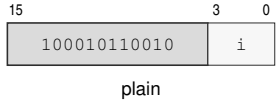
Program-Counter Indexed / Predecrement Encodings



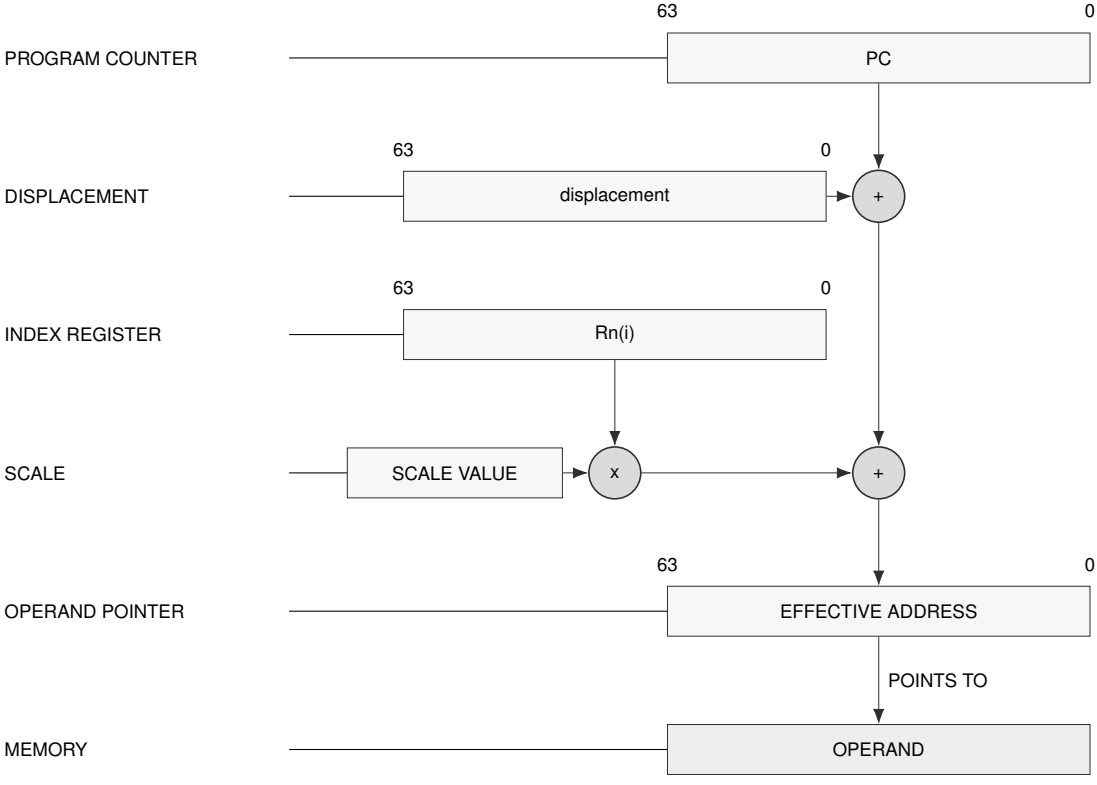
Program-Counter Indexed / Predecrement Address Generation

VECTOR VEA EXT2 Program-Counter Indexed / Plain

`[PC + Rn(i) * scale + displacement]`
Descriptor: `10001011 0010iiii`.
Segment: Fixed to CS.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.



Program-Counter Indexed / Plain Encodings



Program-Counter Indexed / Plain Address Generation

6.2 Shared Extended Descriptor

A compact EXT1 or EXT2 value in any profile selects both an exact descriptor length and the displacement width. The descriptor bytes then select segment, base, index, zero-base, SP/PC indexed, and auto-update behavior within that family.

Each extended descriptor selects its segment from the encoded SREG field SEG(s), the fixed SS segment for SP, the fixed CS segment for PC, or the operation's default data segment.

For a compact displacement, absolute address, or immediate, the selected payload follows the compact field at that operand's payload position. Extended-descriptor construction begins with the compact escape value in the opcode space and exactly one EXT1 or two EXT2 descriptor bytes; a displaced form places its selected displacement after all extended descriptor bytes in the instruction. When an instruction has more than one extended EA operand, the decoder consumes their complete descriptors in declared instruction operand order. It then consumes every remaining appended EA payload, including any selected displacement, in declared instruction operand order.

6.3 Extended EA Addressing Modes

EXT1 and EXT2 extend EA when a compact form cannot express the operand: explicit segment selection, indexed addressing, zero-base addressing, SP/PC indexed addressing, or auto-update addressing. EXT1 selects exactly one descriptor byte; EXT2 selects exactly two descriptor bytes.

Each extended-descriptor form selects its segment from the encoded SREG field SEG(s), the fixed SS segment for SP, the fixed CS segment for PC, or the operation's default data segment.

EA evaluation begins by copying the architectural registers into temporary operand-evaluation state. Operands are then processed in instruction order. Within one EA, the base term is evaluated before the index term and then the displacement. Auto-update changes only the temporary operand-evaluation state while operands are being evaluated. Memory reads and writes are collected as instruction side effects, and neither those effects nor the register updates become architecturally visible until the instruction commits. A fault before commit therefore leaves both register and memory state unchanged apart from the exception-entry transition.

Postincrement forms use the current temporary value and update it afterward. Predecrement forms update the temporary value first and then use the result. A base register changes by the EA interpretation width in bytes; an index register changes by one element before scaling. Each REP iteration applies and commits these rules independently.

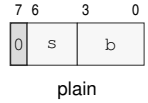
Size-less address-role instructions INVPAGE, FLSHDCACHE, INVDCACHE, INVICACHE, WRBKDCACHE, SYNCCACHE, PREFETCH, PREFETCHNT, and PTQUERY have Q interpretation width. Size-less control-target instructions DJcc, IJcc, LCALL, and LJMP also have Q interpretation width. Their extended-descriptor index scale is eight and their base predecrement or postincrement amount is eight. Cache range loops advance their explicit address by the CPUID maintenance granule, independently of this per-EA update amount.

When one register supplies both terms in `[R1 + R1++ * scale]`, the base and index terms use the same current temporary R1 before the index update. In `[R1 + --R1 * scale]`, the base uses the current value; the index term then decrements temporary R1 and uses the updated value.

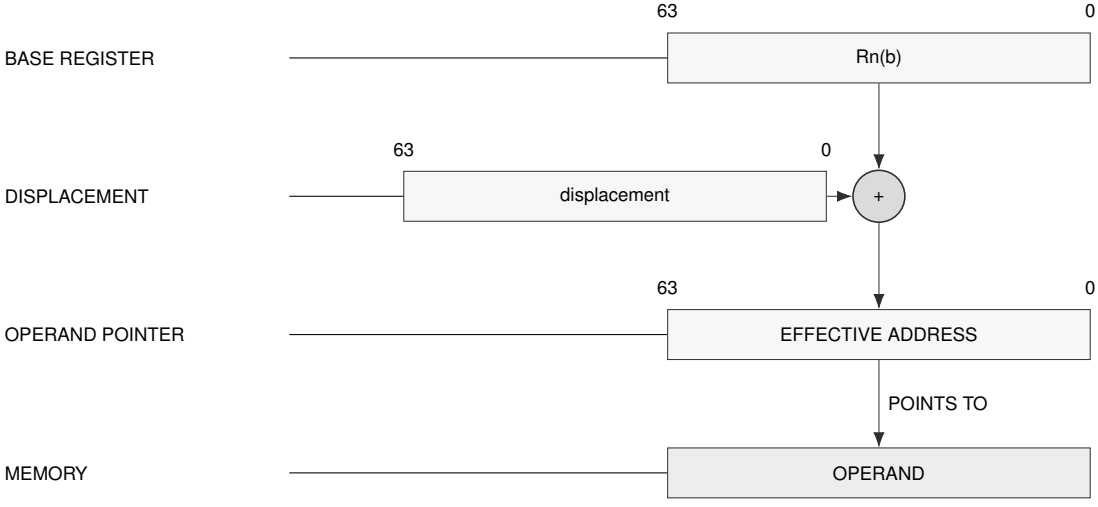
Scalar EXT1 Modes

EXT1 Explicit Segment with Base

`[SEG(s):Rn(b) + displacement]`
Descriptor: `0sssbbbb`.
Segment: SEG(s) selects the encoded segment register.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.



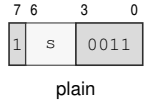
Explicit Segment with Base Encodings



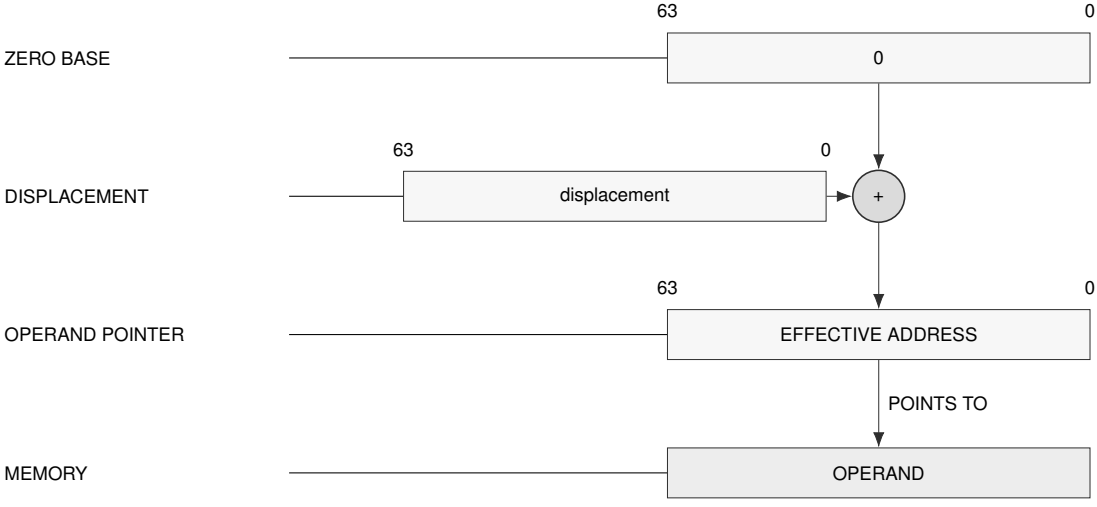
Explicit Segment with Base Address Generation

EXT1 Explicit Segment with Zero Base

`[SEG(s):0 + displacement]`
Descriptor: `1sss0011`.
Segment: SEG(s) selects the encoded segment register.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.



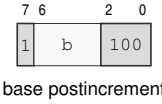
Explicit Segment with Zero Base Encodings



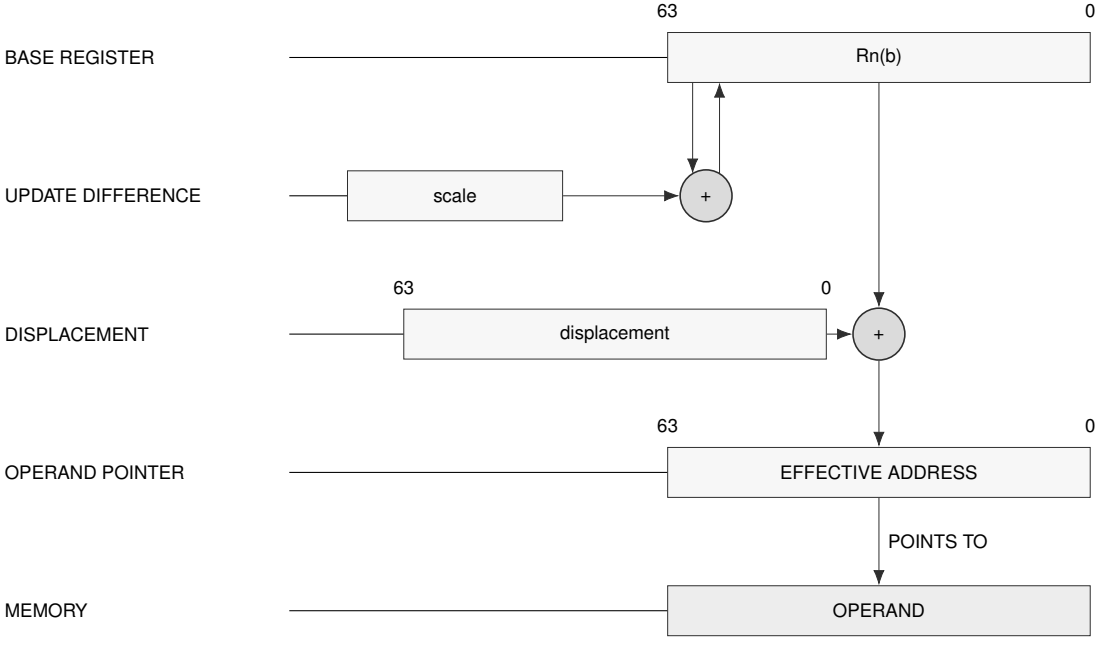
Explicit Segment with Zero Base Address Generation

EXT1 Default-Segment Base with Autoupdate / Postincrement

`[Rn(b)++ + displacement]`
Descriptor: `1bbbb100` (base postincrement).
Segment: Uses the operation default data segment.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Postincrement uses the current temporary base register, then adds `scale`.



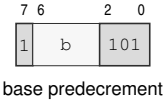
Default-Segment Base with Autoupdate / Postincrement Encodings



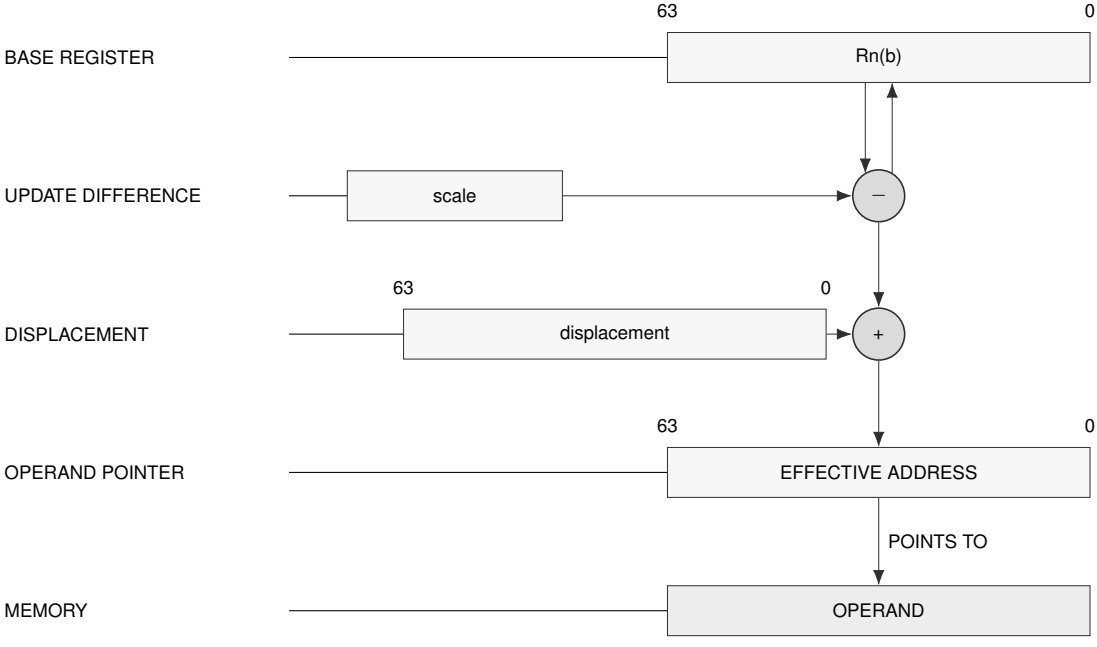
Default-Segment Base with Autoupdate / Postincrement Address Generation

EXT1 Default-Segment Base with Autoupdate / Predecrement

`[--Rn(b) + displacement]`
Descriptor: `1bbbb101` (base predecrement).
Segment: Uses the operation default data segment.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Predecrement subtracts `scale` from the temporary base register before use.



Default-Segment Base with Autoupdate / Predecrement Encodings

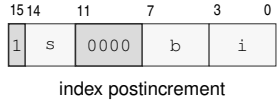


Default-Segment Base with Autoupdate / Predecrement Address Generation

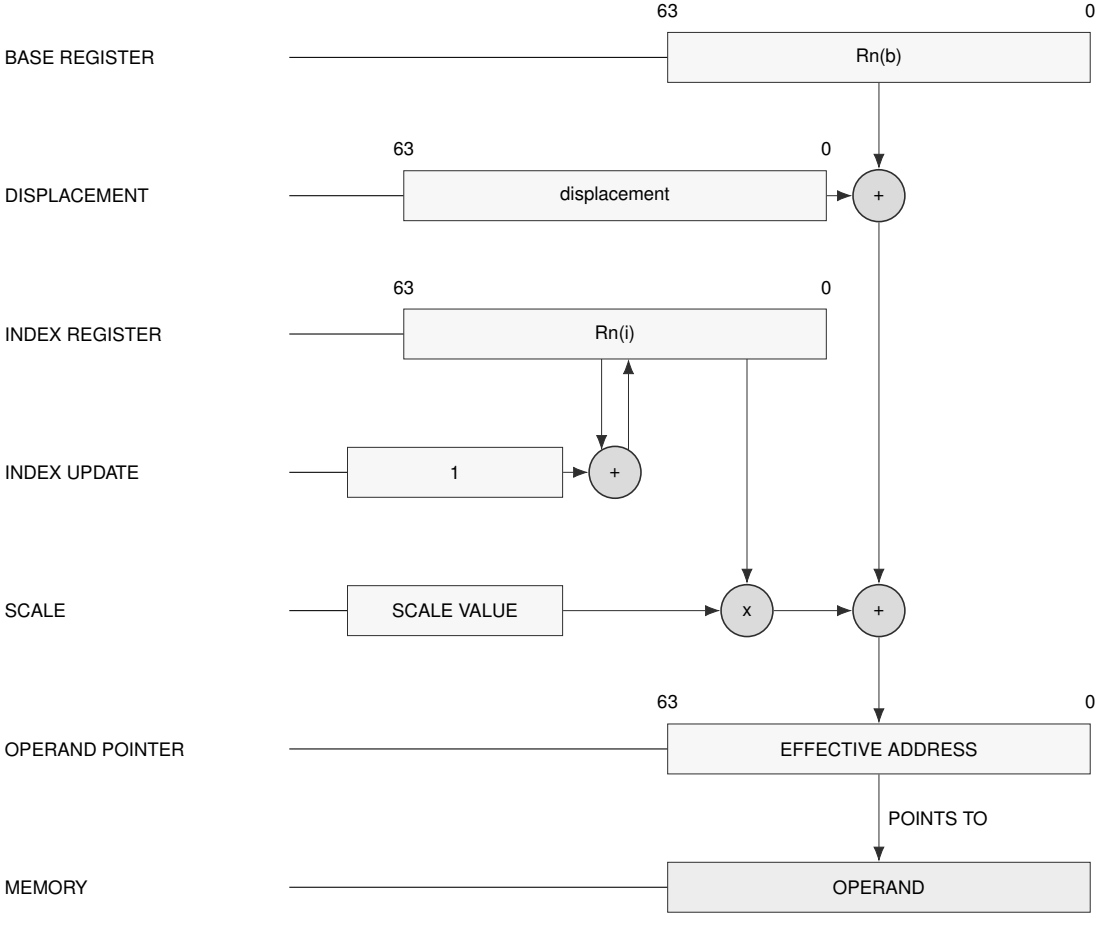
Scalar EXT2 Modes

EXT2 Explicit-Segment Indexed / Postincrement

`[SEG(s):Rn(b) + Rn(i)++ * scale + displacement]`
Descriptor: `1sss0000 bbbbiiii` (index postincrement).
Segment: SEG(s) selects the encoded segment register.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Postincrement uses the current temporary index register, then adds 1.



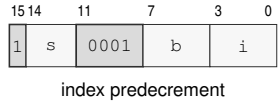
Explicit-Segment Indexed / Postincrement Encodings



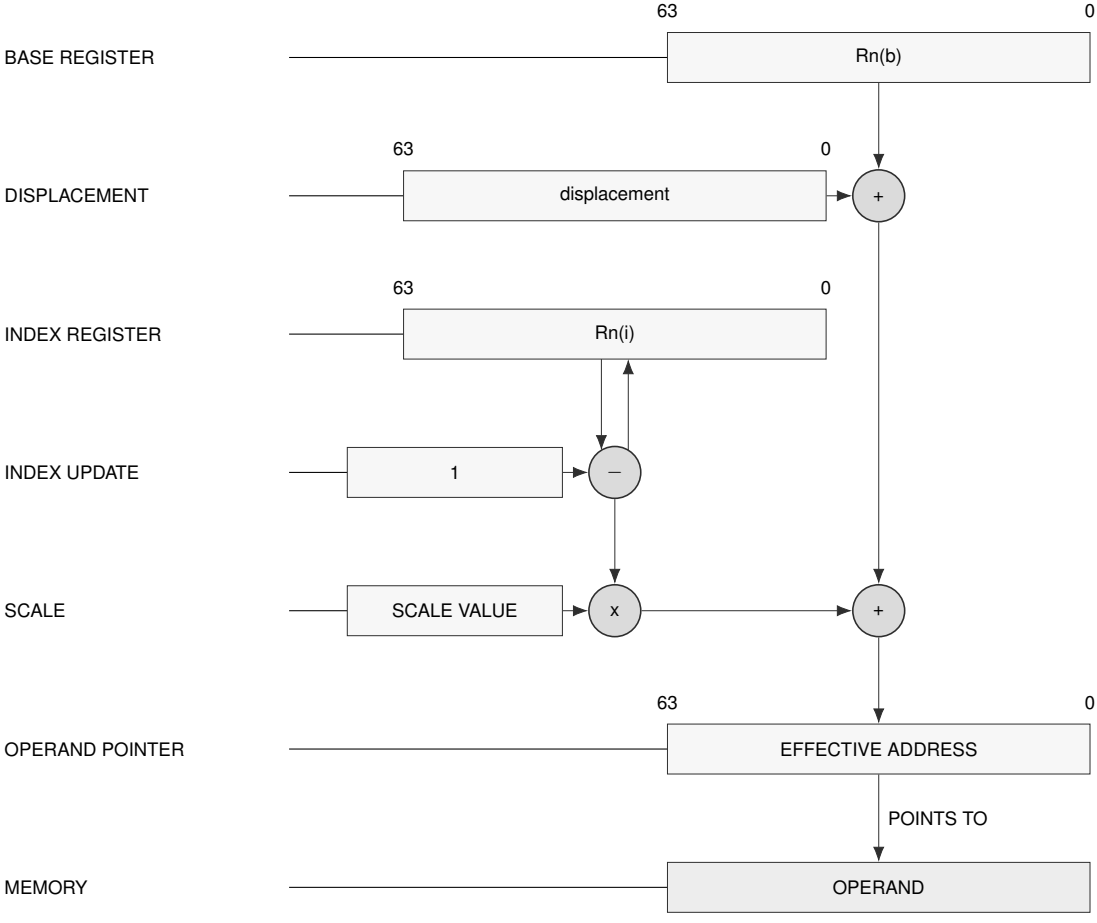
Explicit-Segment Indexed / Postincrement Address Generation

EXT2 Explicit-Segment Indexed / Predecrement

`[SEG(s):Rn(b) + --Rn(i) * scale + displacement]`
Descriptor: `1sss0001 bbbbiiii` (index predecrement).
Segment: SEG(s) selects the encoded segment register.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Predecrement subtracts 1 from the temporary index register before use.



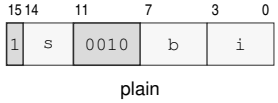
Explicit-Segment Indexed / Predecrement Encodings



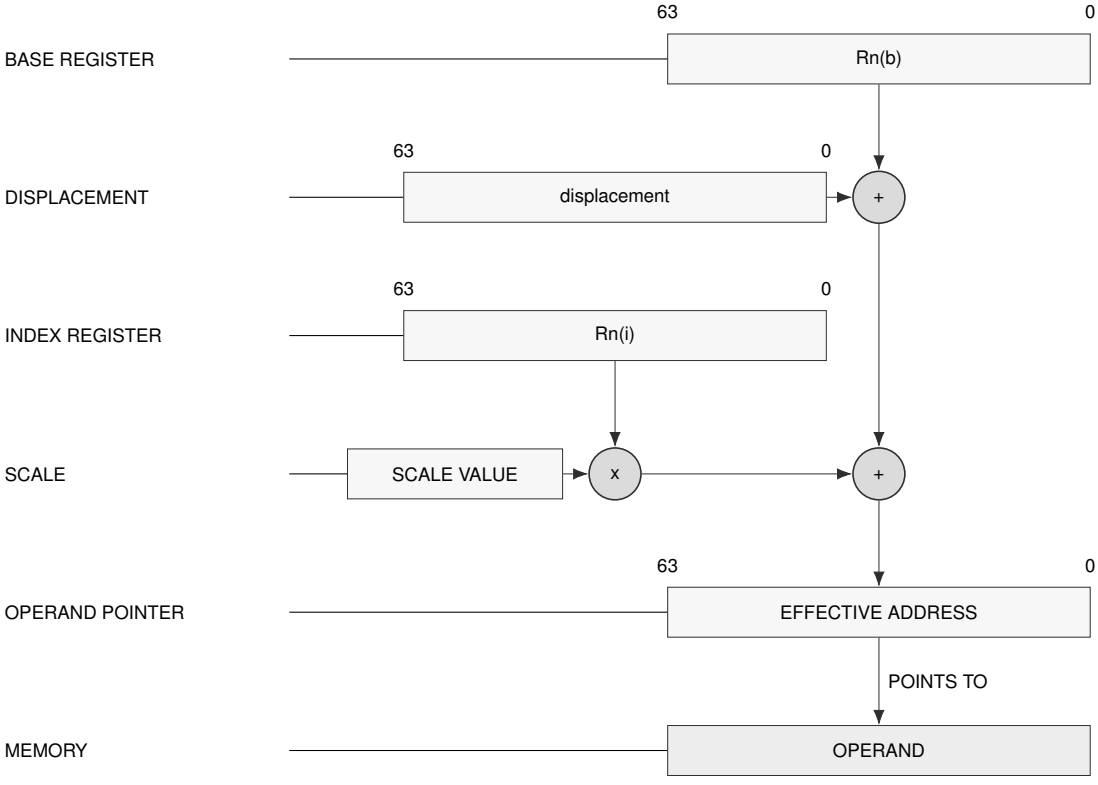
Explicit-Segment Indexed / Predecrement Address Generation

EXT2 Explicit-Segment Indexed / Plain

`[SEG(s):Rn(b) + Rn(i) * scale + displacement]`
Descriptor: `1sss0010 bbbbiiii`.
Segment: SEG(s) selects the encoded segment register.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.



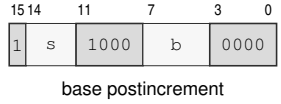
Explicit-Segment Indexed / Plain Encodings



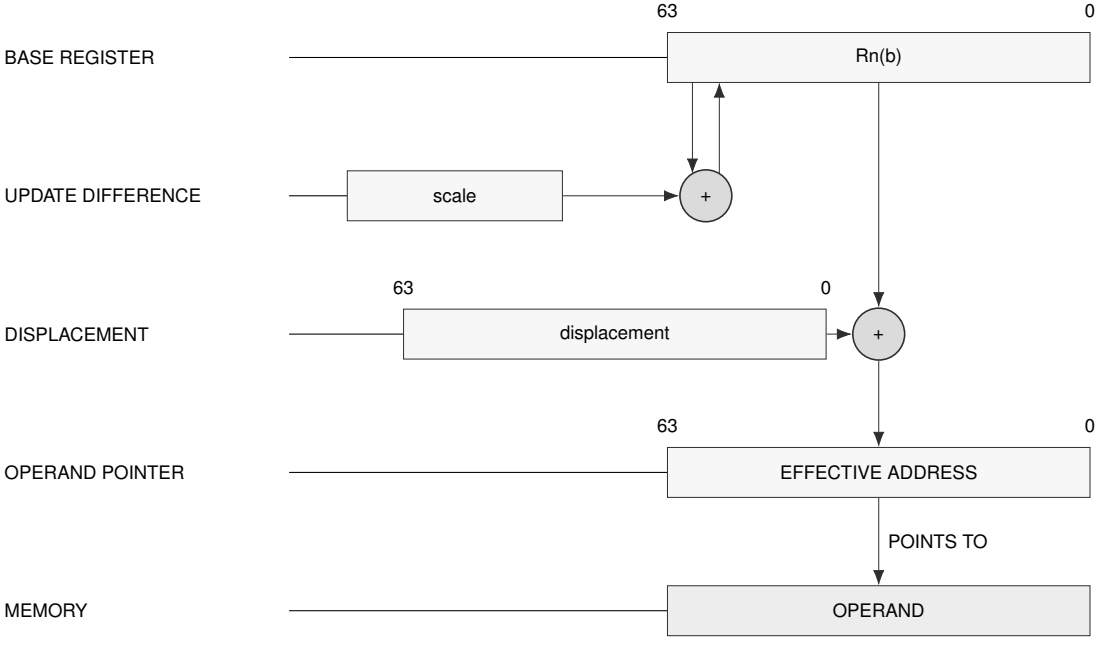
Explicit-Segment Indexed / Plain Address Generation

EXT2 Explicit-Segment Base with Autoupdate / Postincrement

`[SEG(s):Rn(b)++ + displacement]`
Descriptor: `1sss1000 bbbb0000` (base postincrement).
Segment: SEG(s) selects the encoded segment register.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Postincrement uses the current temporary base register, then adds `scale`.



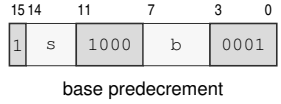
Explicit-Segment Base with Autoupdate / Postincrement Encodings



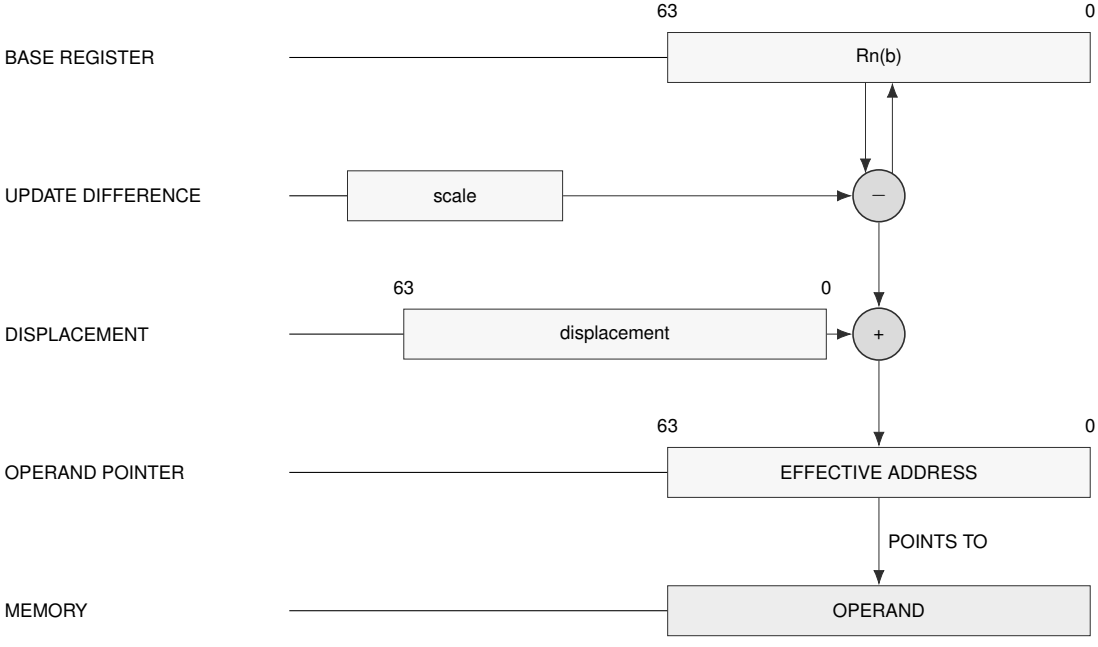
Explicit-Segment Base with Autoupdate / Postincrement Address Generation

EXT2 Explicit-Segment Base with Autoupdate / Predecrement

`[SEG(s):--Rn(b) + displacement]`
Descriptor: `1sss1000 bbbb0001` (base predecrement).
Segment: SEG(s) selects the encoded segment register.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Predecrement subtracts `scale` from the temporary base register before use.



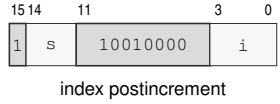
Explicit-Segment Base with Autoupdate / Predecrement Encodings



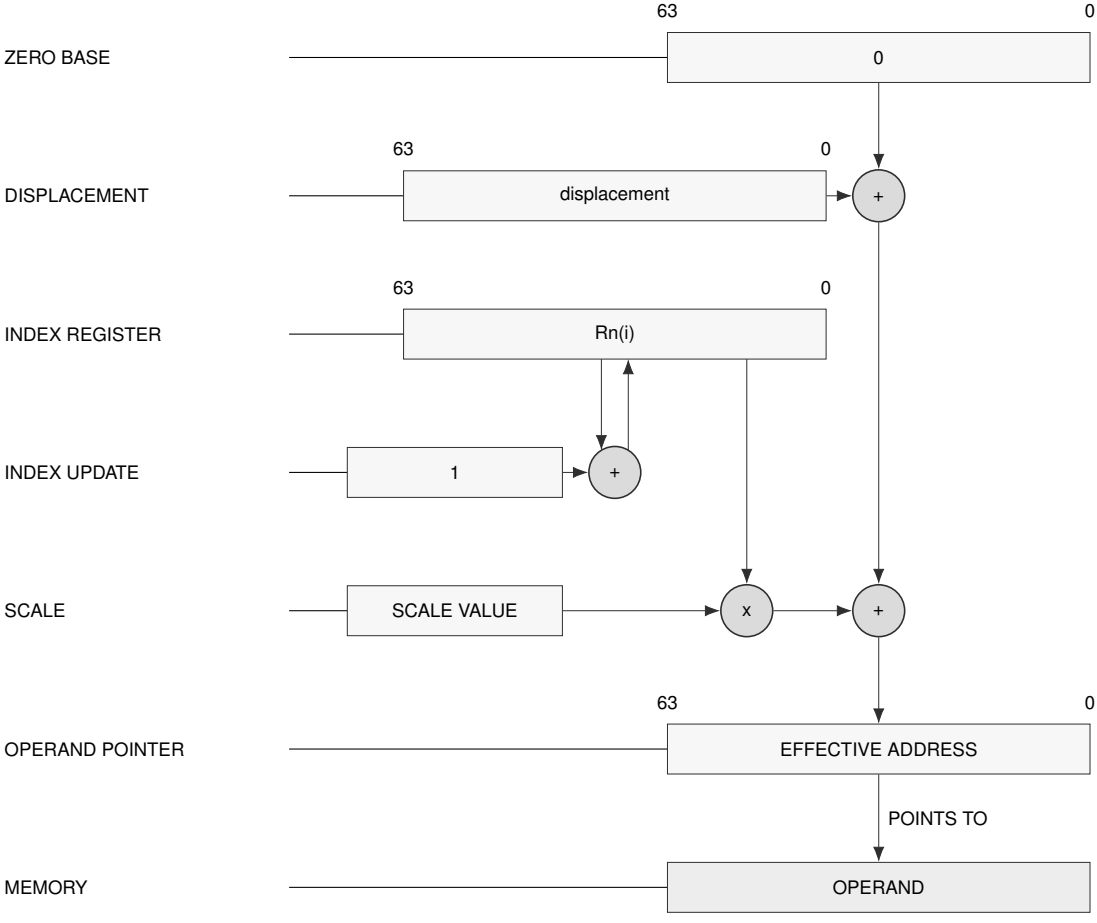
Explicit-Segment Base with Autoupdate / Predecrement Address Generation

EXT2 Explicit-Segment Zero-Base Indexed / Postincrement

`[SEG(s):0 + Rn(i)++ * scale + displacement]`
Descriptor: `1sss1001 0000iiii` (index postincrement).
Segment: SEG(s) selects the encoded segment register.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Postincrement uses the current temporary index register, then adds 1.



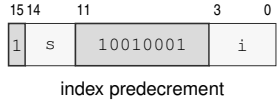
Explicit-Segment Zero-Base Indexed / Postincrement Encodings



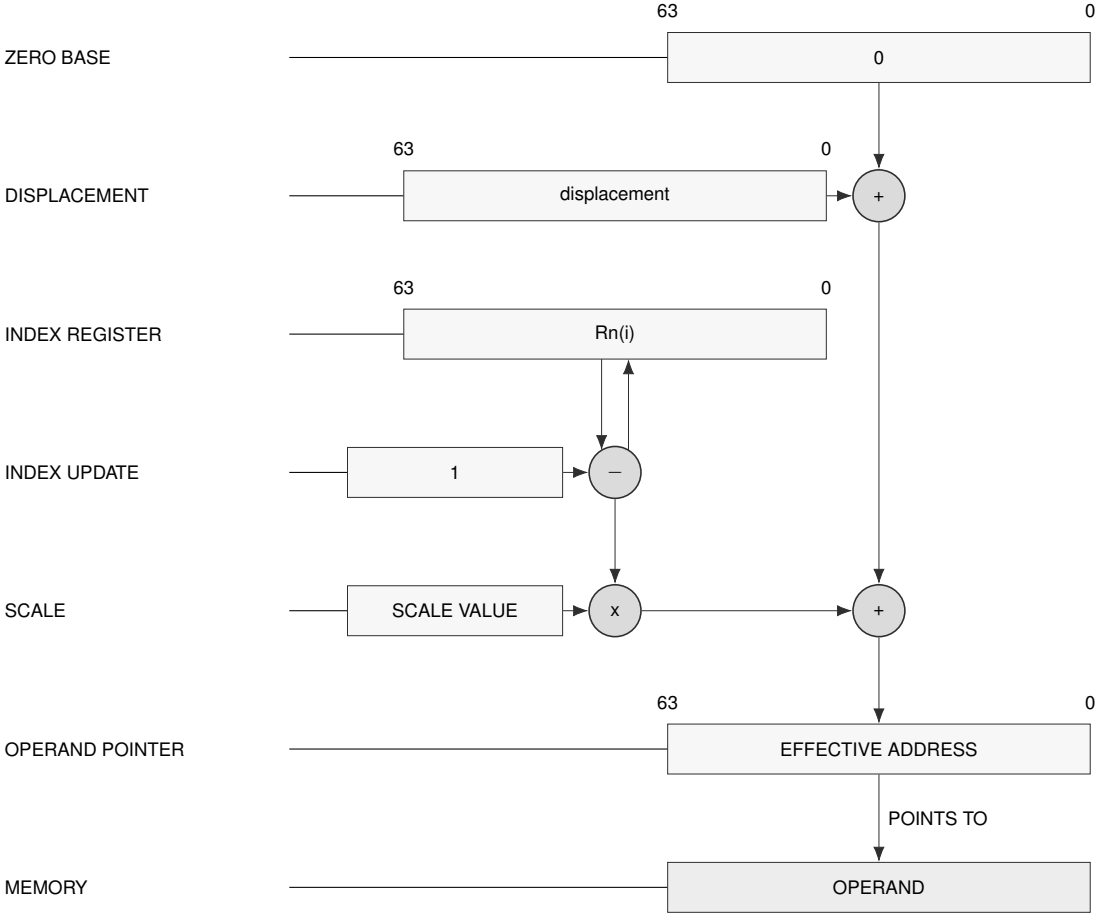
Explicit-Segment Zero-Base Indexed / Postincrement Address Generation

EXT2 Explicit-Segment Zero-Base Indexed / Predecrement

`[SEG(s):0 + --Rn(i) * scale + displacement]`
Descriptor: `1sss1001 0001iiii` (index predecrement).
Segment: SEG(s) selects the encoded segment register.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Predecrement subtracts 1 from the temporary index register before use.



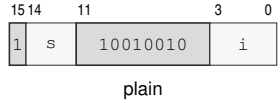
Explicit-Segment Zero-Base Indexed / Predecrement Encodings



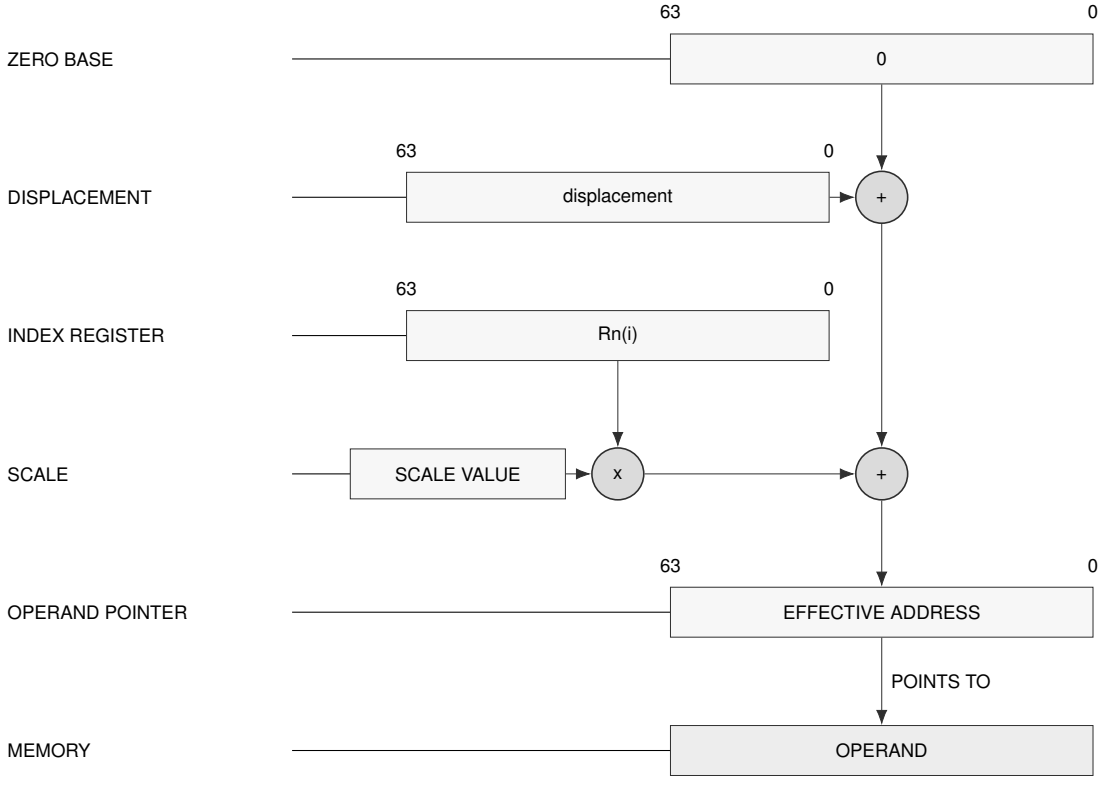
Explicit-Segment Zero-Base Indexed / Predecrement Address Generation

EXT2 Explicit-Segment Zero-Base Indexed / Plain

`[SEG(s):0 + Rn(i) * scale + displacement]`
Descriptor: `1sss1001 0010iiii`.
Segment: SEG(s) selects the encoded segment register.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.



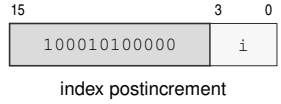
Explicit-Segment Zero-Base Indexed / Plain Encodings



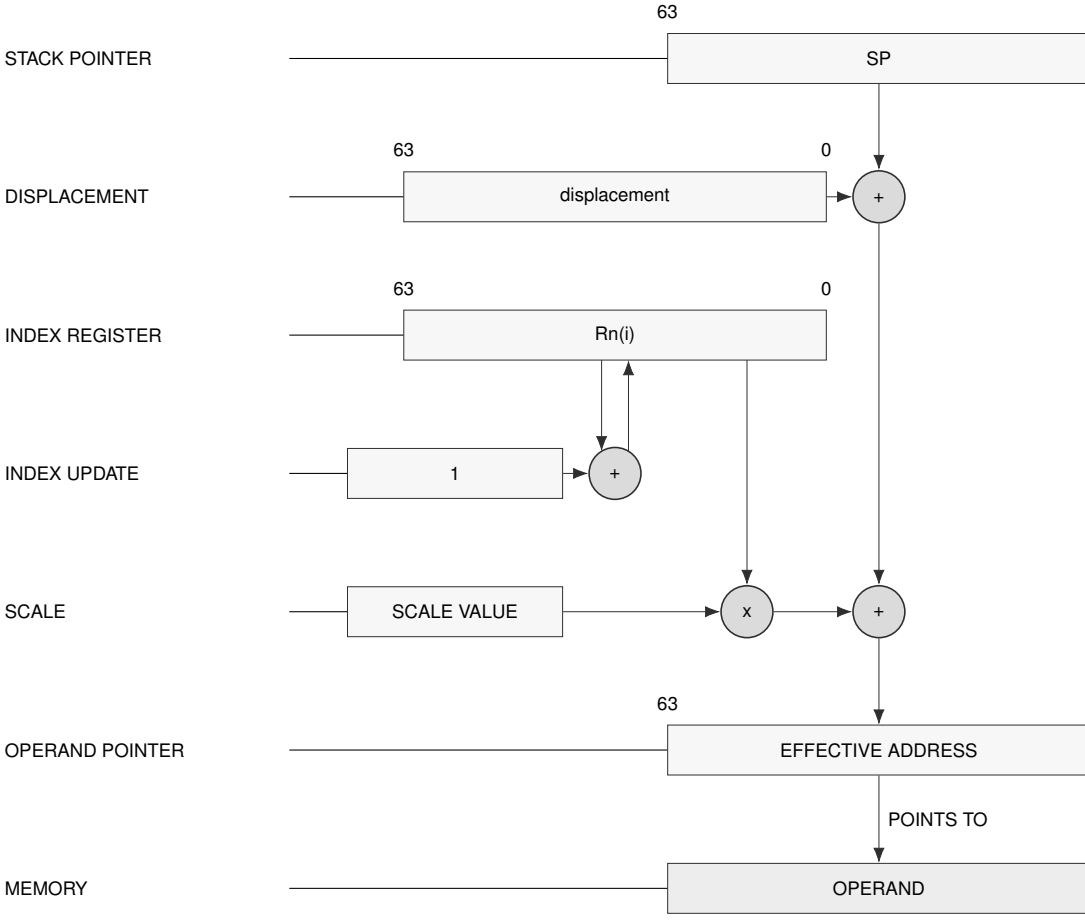
Explicit-Segment Zero-Base Indexed / Plain Address Generation

EXT2 Stack-Pointer Indexed / Postincrement

`[SP + Rn(i)++ * scale + displacement]`
Descriptor: `10001010 0000iiii` (index postincrement).
Segment: Fixed to SS.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Postincrement uses the current temporary index register, then adds 1.



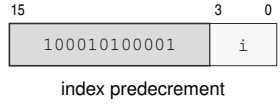
Stack-Pointer Indexed / Postincrement Encodings



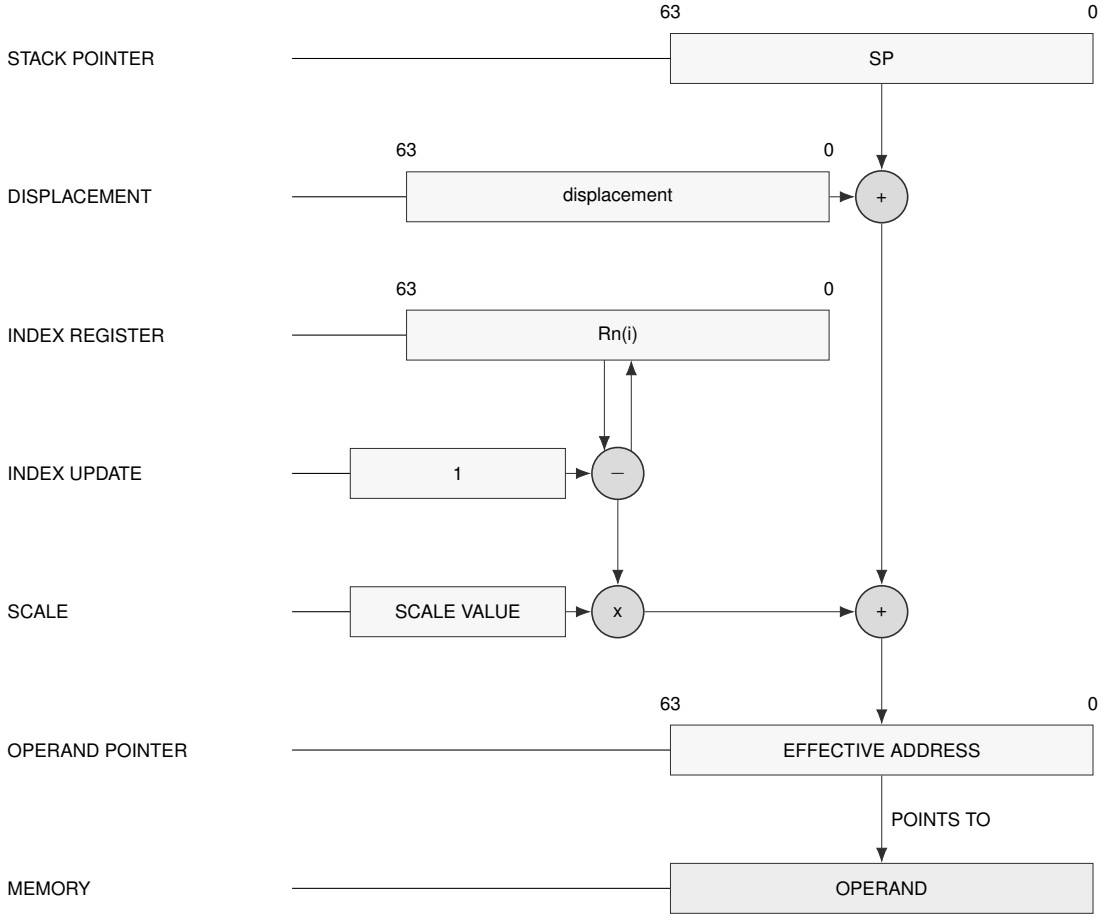
Stack-Pointer Indexed / Postincrement Address Generation

EXT2 Stack-Pointer Indexed / Predecrement

`[SP + --Rn(i) * scale + displacement]`
Descriptor: `10001010 0001iiii` (index predecrement).
Segment: Fixed to SS.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Predecrement subtracts 1 from the temporary index register before use.



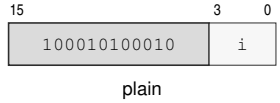
Stack-Pointer Indexed / Predecrement Encodings



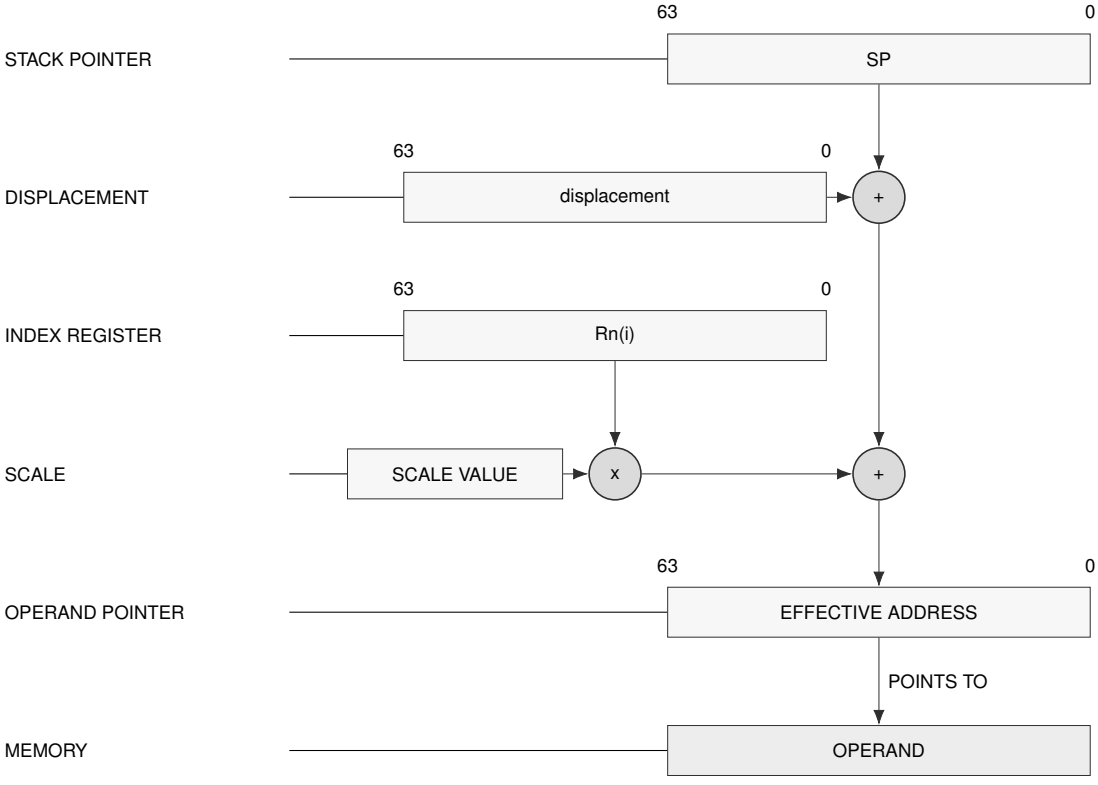
Stack-Pointer Indexed / Predecrement Address Generation

EXT2 Stack-Pointer Indexed / Plain

`[SP + Rn(i) * scale + displacement]`
Descriptor: `10001010 0010iiii`.
Segment: Fixed to SS.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.



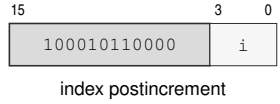
Stack-Pointer Indexed / Plain Encodings



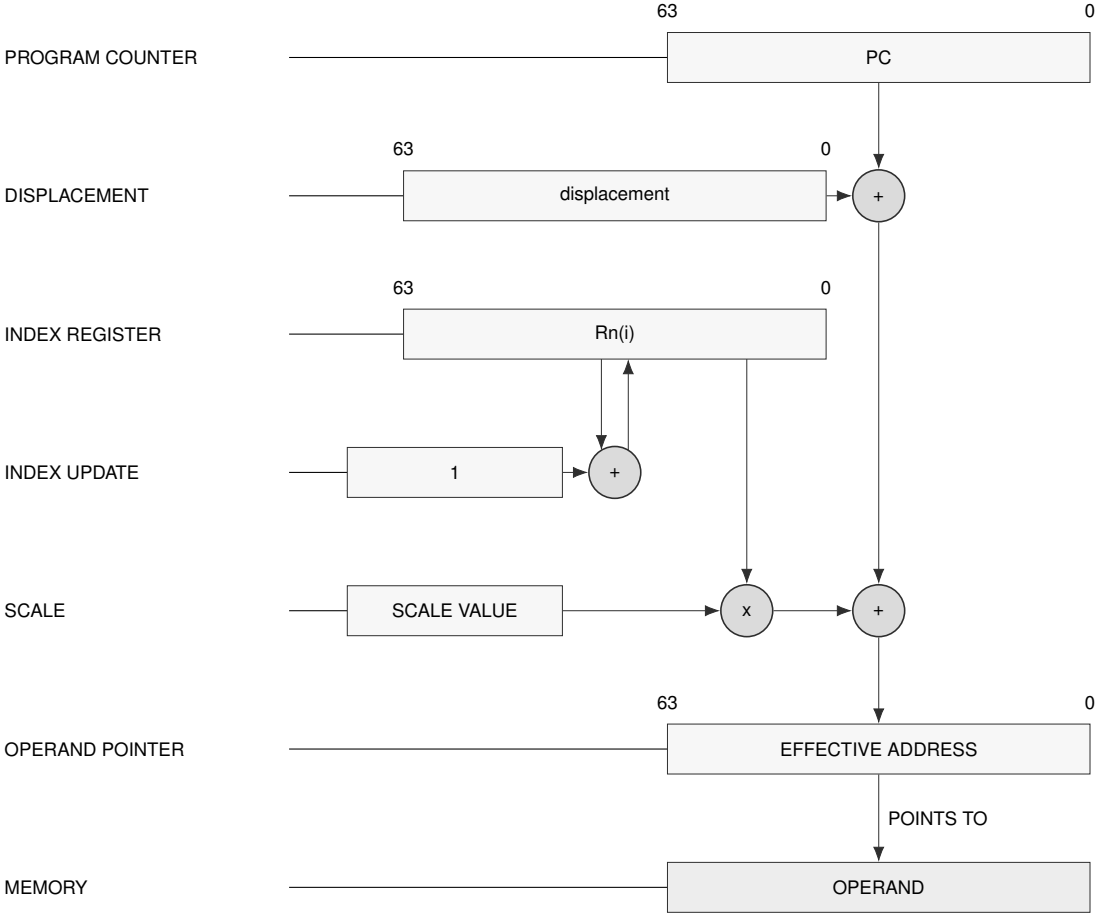
Stack-Pointer Indexed / Plain Address Generation

EXT2 Program-Counter Indexed / Postincrement

`[PC + Rn(i)++ * scale + displacement]`
Descriptor: `10001011 0000iiii` (index postincrement).
Segment: Fixed to CS.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Postincrement uses the current temporary index register, then adds 1.



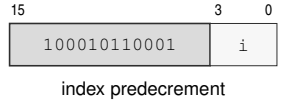
Program-Counter Indexed / Postincrement Encodings



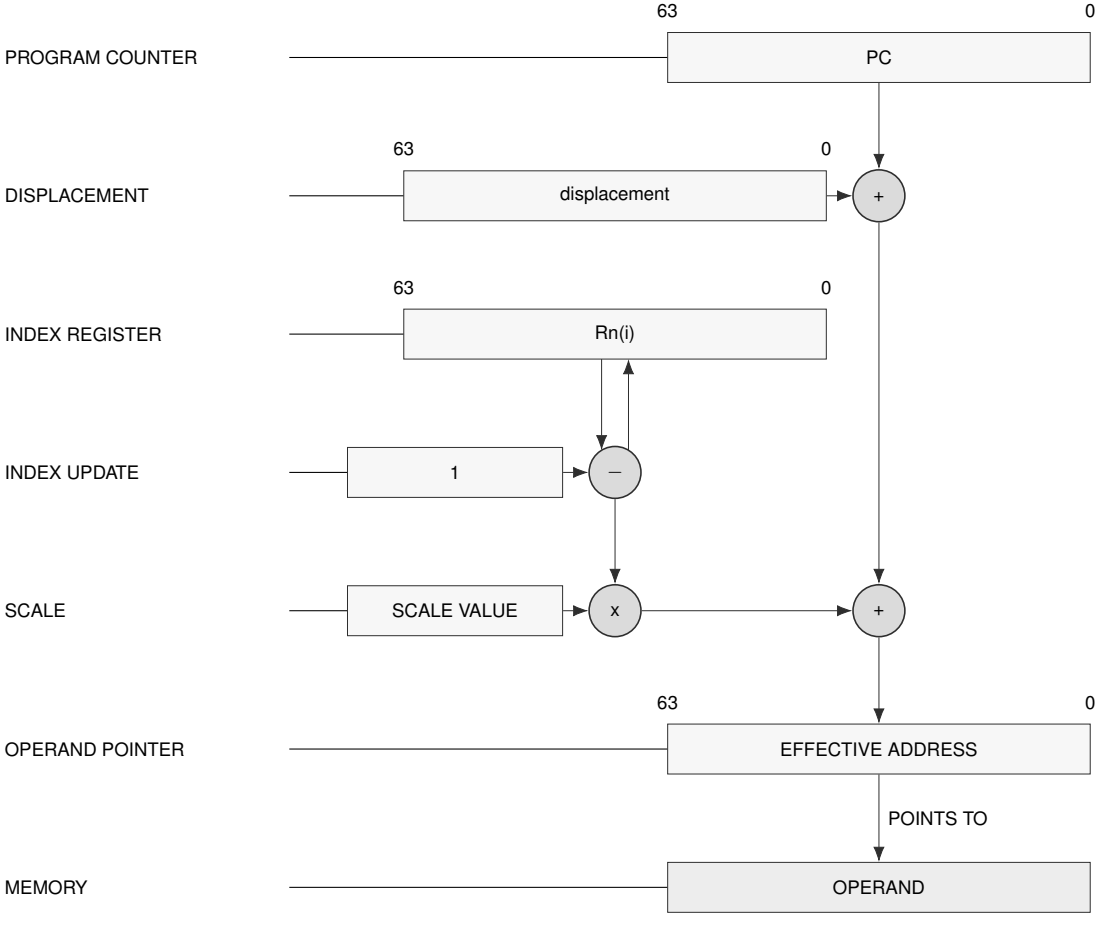
Program-Counter Indexed / Postincrement Address Generation

EXT2 Program-Counter Indexed / Predecrement

`[PC + --Rn(i) * scale + displacement]`
Descriptor: `10001011 0001iiii` (index predecrement).
Segment: Fixed to CS.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.
Update: Predecrement subtracts 1 from the temporary index register before use.



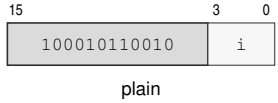
Program-Counter Indexed / Predecrement Encodings



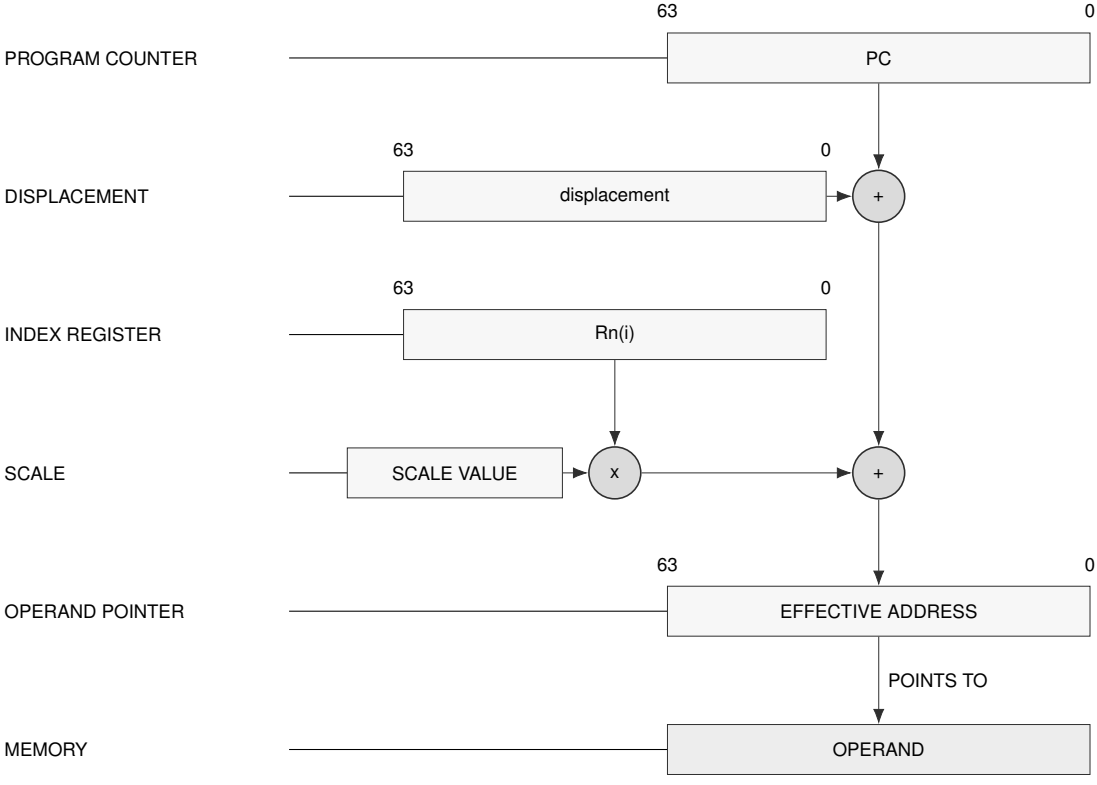
Program-Counter Indexed / Predecrement Address Generation

EXT2 Program-Counter Indexed / Plain

`[PC + Rn(i) * scale + displacement]`
Descriptor: `10001011 0010iiii`.
Segment: Fixed to CS.
Payload: The descriptor is followed by the displacement selected by the compact EXT escape.



Program-Counter Indexed / Plain Encodings



Program-Counter Indexed / Plain Address Generation

7 MEMORY ADDRESS TRANSLATION

Memory address translation is a separate pipeline after effective-address calculation. Effective-address evaluation produces an address value or operand designator; segmentation optionally turns that value into a linear address, and paging optionally turns the linear address into a memory-system address.

Instruction fetches use CS, stack accesses use SS, and ordinary data accesses use the operation default data segment unless an effective-address form explicitly selects another segment. SP and PC forms use their fixed segments.

`SEGLEA` exposes the linear address produced by segment pre-translation when software explicitly needs it. Its instruction description defines the point check, FLAGS result, destination suppression, and auto-update commit behavior. SEGLEA applies segment pre-translation to the computed starting point.

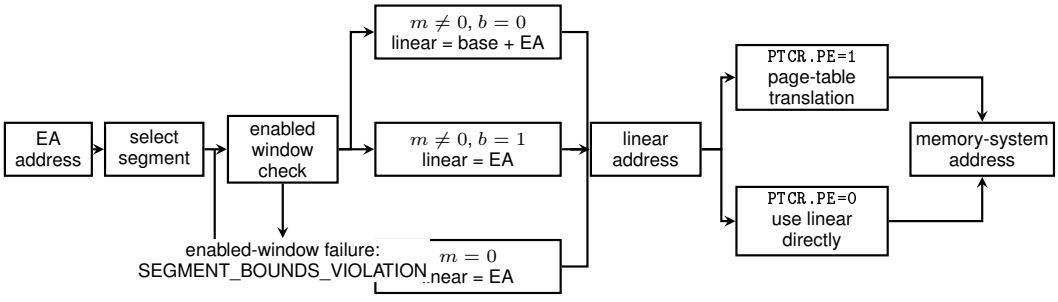


Figure 7-1. Address Translation Pipeline

7.1 Segment Pre-Translation

Segment pre-translation interprets the selected 64-bit segment image before paging. The segment fields, derived quantities, and image-validity condition are defined in Segment Registers. A disabled segment passes the effective address through. An enabled segment either checks the effective address as an offset and adds the base, or checks an already-linear address in bounds-only mode.

Table 7-1. Segment Pre-Translation Modes

Mode	Segment State	Check	Linear Address
disabled	`SEGMENT.M=0`	no segment-window check	linear = EA
translated window	`SEGMENT.M!=0`, `SEGMENT.B=0`	complete byte range lies within $0 \le \mathrm{EA} < \mathrm{span}$	linear = base + EA
bounds-only window	`SEGMENT.M!=0`, `SEGMENT.B=1`	complete byte range lies within $\mathrm{base} \le \mathrm{EA} < \mathrm{limit}$	linear = EA

Address validation begins by forming the starting EA and its segment-pretranslated linear value. The starting point must be within the selected segment. With paging enabled, the complete byte range is translated in increasing linear-address order after canonicality checks. With paging disabled, the same byte-range rules apply directly.

Complete range additions are mathematical: 64-bit wraparound never makes an out-of-range access valid. A range failure raises `SEGMENT_BOUNDS_VIOLATION` before the instruction's target memory access begins.

The segment result is the linear address. With `PTCR.PE` set, the page-table walker consumes that address and applies the selected PTE frame and attributes. With paging disabled, the linear address is the physical byte address and must fit `PABITS`. An out-of-range direct address raises `PHYSICAL_ADDRESS_FAULT`. An invalid segment image raises `INVALID_CONTROL_IMAGE` when written; a failed segment bound or canonical-address check raises `SEGMENT_BOUNDS_VIOLATION` or `NONCANONICAL_ADDRESS`, respectively, during access.

7.2 Paging Stage

Paging is enabled by `PTCR.PE` and translates the linear address produced by segment pre-translation. Before the walk can produce a translation, the linear address must be canonical for the translation-table format selected by `PTCR.TT`. A noncanonical address raises `NONCANONICAL_ADDRESS`. With paging disabled, the memory system uses the linear address directly and does not interpret `PTCR.TT` or `PTCR.ROOT_PAGE`.

Every page-table entry is one naturally aligned 64-bit descriptor. L4, L3, and L2 are 16-KiB-aligned table objects containing 2048 entries selected by an 11-bit address field. L1 is a 4-KiB-aligned table object containing exactly 512 entries selected by a nine-bit address field. For a table-object base address B and index i, the selected entry is always at $B + 8i$. PTEs are a dense array of eight-byte descriptors; an L1 object has no architecturally unreachable trailing storage.

The walker starts at the 16-KiB-aligned physical table object named by `PTCR.ROOT_PAGE` for the selected format. At each active level, a present selected entry with `PTE.T=1` supplies `PTE.NEXT_TABLE`, the exact base address of the next table object; this form is valid only above L1. An L4-to-L3 or L3-to-L2 transition requires a 16-KiB-aligned target. An L2-to-L1 transition requires only a 4-KiB-aligned target, so the walker preserves `PTE.NEXT_TABLE[13:12]` for that transition. A selected entry with `PTE.T=0` terminates the walk as a byte-addressed leaf. Define the number of low linear-address bits supplied by a leaf at level L as $o(L1) = 14$, $o(L2) = 23$, $o(L3) = 34$, and $o(L4) = 45$. The leaf `PTE.PFN` supplies a naturally aligned physical base, and the final byte address is:

$$byte\_address = (leaf\_PFN \ll 14) \mid linear[o(L) - 1 : 0].$$

The resulting L1 through L4 leaf sizes are 16 KiB, 8 MiB, 16 GiB, and 32 TiB. Bit 7 is `PTE.A` in both descriptor layouts and does not participate in address formation.

Software may place up to four independent L1 objects in one 16-KiB physical backing frame, including objects belonging to different roots or ASIDs in the same supervisor-owned physical-memory domain. Such co-location does not combine address-space permissions: each table pointer and leaf continues to govern its own subtree or mapping. A backing frame is used only for L1 storage while any of its four slots is live. Slot allocation, ownership, zeroing before linking, reference counting, and reclamation are software responsibilities. Independently protected physical ownership domains may use separate allocator pools without changing the descriptor format. Software may instead allocate every L1 object at a 16-KiB-aligned address by keeping `PTE.NEXT_TABLE[13:12]=00`; every conforming walker must nevertheless accept all four 4-KiB-aligned L1 positions.

The walker intersects table-pointer `PTE.R`/`PTE.W`/`PTE.X`/`PTE.U` fields while descending. The terminating leaf supplies decoded `PTE.AM` permissions, `PTE.U`, the final `PTE.PFN`, `PTE.G`, `PTE.CP`, and the Normal/MMIO class. The detailed entry-validity, permission, accessed, and dirty rules are specified under Page-Table Entry Format. At each consumed level, result priority is not-present first, structural validity second, and effective permission third; the walker resolves those results before consuming a lower level.

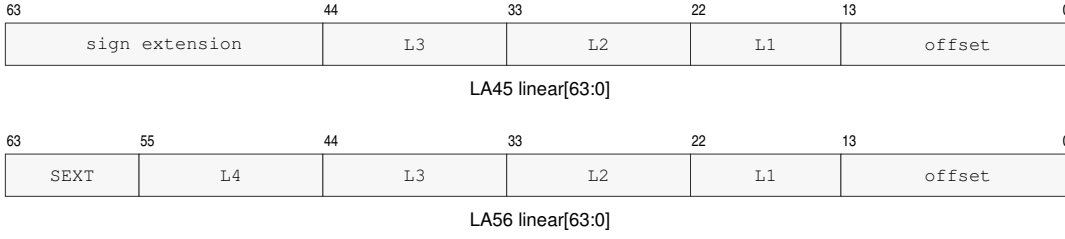


Figure 7-2. Linear-Address Paging Fields

7.3 LA45 Three-Level Paging

When `PTCR.TT` is `010`, bits `63..45` of the linear address must equal the sign extension of bit `44`. Bits `44..34`, `33..23`, and `22..14` select the L3, L2, and L1 entries respectively. A byte-addressed leaf may terminate the walk at any of those levels and uses the level-specific offset defined by the Paging Stage. The L3 table is the root table named by `PTCR.ROOT_PAGE`.

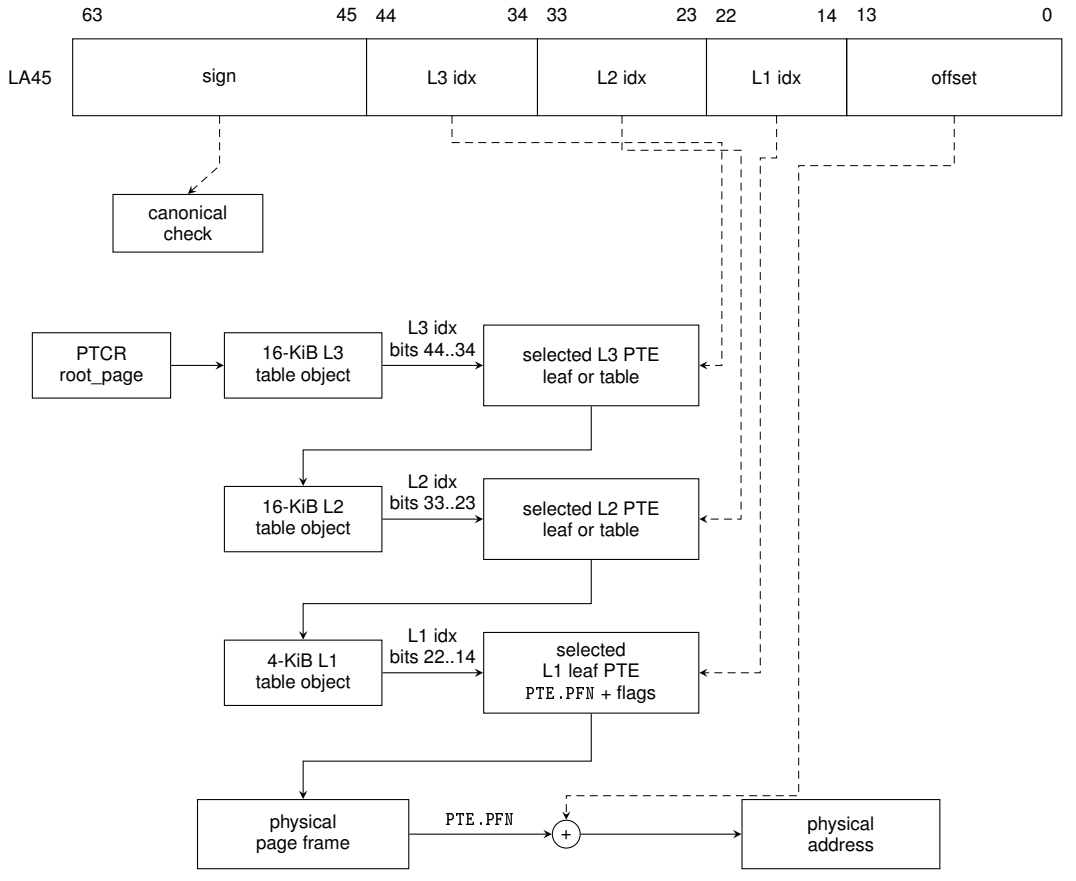
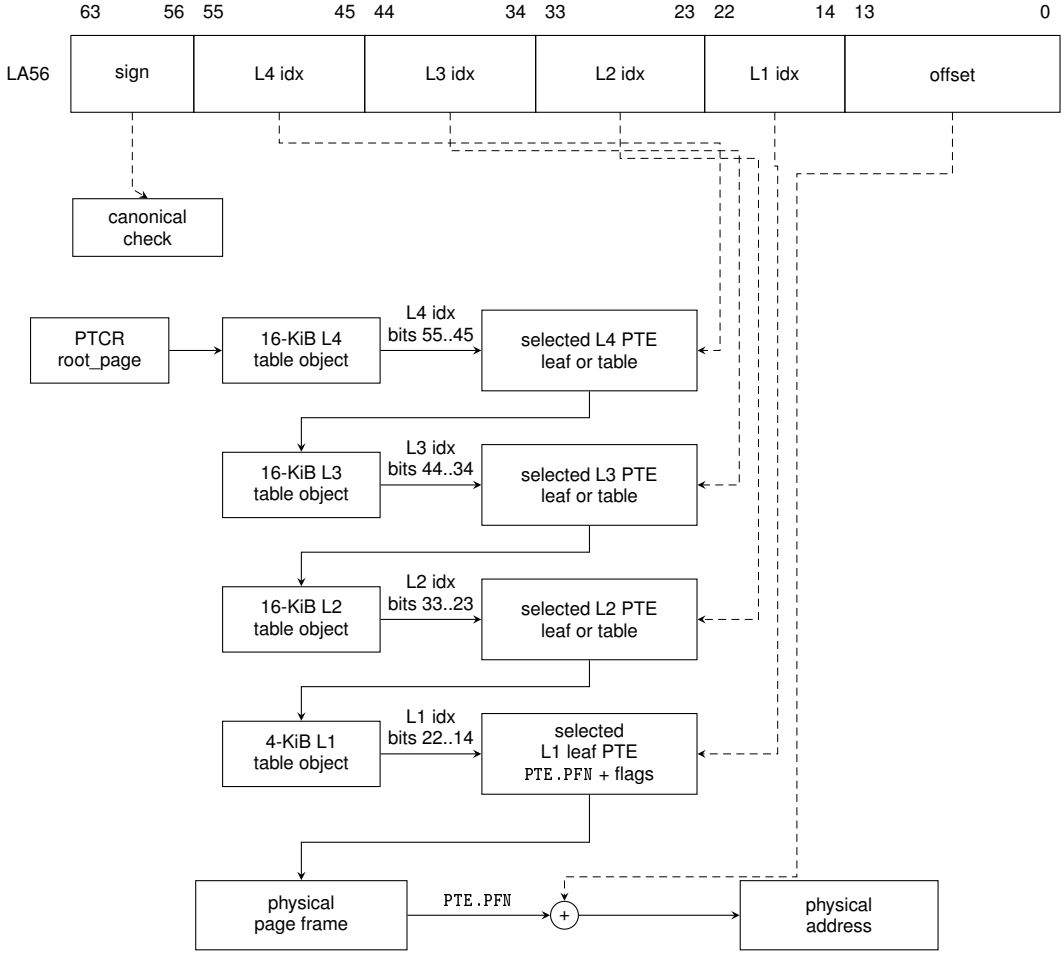


Figure 7-3. LA45 Three-Level Page Walk

7.4 LA56 Four-Level Paging

When PTCR.TT is 011, bits 63..56 of the linear address must equal the sign extension of bit 55. Bits 55..45 select an additional L4 entry. A table pointer continues into the same L3-through-L1 hierarchy used by LA45, while a byte-addressed leaf at any selected level terminates the walk and uses the level-specific offset defined by the Paging Stage. The L4 table is the root table named by PTCR.ROOT_PAGE.



Maximum-depth walk shown. At L4 through L2, PTE.T=1 continues and PTE.T=0 terminates at an aligned byte leaf; upper table transitions require 16-KiB alignment, while L2-to-L1 consumes the full 4-KiB-aligned PTE.NEXT_TABLE.

Figure 7-4. LA56 Four-Level Page Walk

7.5 Page-Table Entry Format

Every page-table entry is one naturally aligned 64-bit descriptor. A leaf owns `PTE.PFN[55:14]`, while a table pointer owns `PTE.NEXT_TABLE[55:12]`. Both layouts therefore have a 56-bit architectural physical-address ceiling. An implementation reports its exact `PABITS` value, from 32 through 56, through CPUID `ADDRESS_WIDTHS`. In a present descriptor, bits of the layout-selected physical-address field at or above that value must be zero. Bits `57..56` are `PTE.SW[1:0]` and are ignored by hardware; bits `63..58` are reserved and must be zero.

`P` in bit 0 is common. When `PTE.P=0`, the remaining bits are ignored and the entry is not present. When `PTE.P=1`, `T` in bit 1 selects the leaf or table-pointer layout. A nonzero reserved bit in a present descriptor raises `MALFORMED_PAGE_TABLE_ENTRY`.

Table 7-2. Leaf Descriptor Fields (`PTE.P=1`, `PTE.T=0`)

Field	Bits	Meaning
`PTE.P`	`0`	present, one
`PTE.T`	`1`	leaf selector, zero
`AM`	`4..2`	permission and Normal/MMIO access class
`U`	`5`	user-domain permission
`G`	`6`	global translation
`A`	`7`	accessed state
`D`	`8`	exact dirty state
`CP`	`10..9`	cache policy
reserved	`13..11`	must be zero
`PFN`	`55..14`	mapped physical frame number
`SW[1:0]`	`57..56`	software-defined, ignored by hardware
reserved	`63..58`	must be zero

Table 7-3. Leaf `PTE.AM` Encodings

`PTE.AM`	Permission	Class	Name
`000`	read	Normal	`R`
`001`	write	Normal	`W`
`010`	execute	Normal	`X`
`011`	read/write	Normal	`RW`
`100`	read/execute	Normal	`RX`
`101`	read	MMIO	`MMIO_R`
`110`	write	MMIO	`MMIO_W`
`111`	read/write	MMIO	`MMIO_RW`

No leaf encoding grants both write and execute, and no MMIO leaf grants execute. This W^X property applies to one selected translation. The architecture does not compare aliases: distinct

descriptors for the same physical page may have different permissions, PTE.AM classes, or PTE.CP values.

Table 7-4. Table-Pointer Descriptor Fields (PTE.P=1, PTE.T=1)

Field	Bits	Meaning
PTE.P	0	present, one
PTE.T	1	table-pointer selector, one
R/W/X	2/3/4	subtree maximum permissions
PTE.U	5	subtree maximum user-domain permission
reserved	6	must be zero
PTE.A	7	accessed state
reserved	11..8	must be zero
NEXT_TABLE	55..12	physical base address of the next table object
PTE.SW[1:0]	57..56	software-defined, ignored by hardware
reserved	63..58	must be zero

A table pointer with PTE.R=PTE.W=PTE.X=0, or a table pointer at L1, is malformed. An L4-to-L3 or L3-to-L2 pointer targets a 16-KiB table object and is malformed when PTE.NEXT_TABLE[13:12] is nonzero. An L2-to-L1 pointer targets a 4-KiB table object and may use all four values of those bits. Every walker, PTQUERY implementation, and page-walk cache preserves the complete PTE.NEXT_TABLE value while descending. The walker begins with PTE.R, PTE.W, PTE.X, and PTE.U enabled, intersects each table bitmap and PTE.U bit, then intersects the terminating leaf's decoded PTE.AM permissions and PTE.U bit. The leaf alone supplies PTE.G, PTE.CP, and Normal/MMIO class. An L1, L2, L3, or L4 leaf must have a physical base aligned to 2^{14}, 2^{23}, 2^{34}, or 2^{45} bytes respectively.

The PTE.CP field is orthogonal to PTE.AM: all 32 PTE.AM/PTE.CP combinations are structurally valid. PTE.CP values 0 through 3 mean cacheable write-back, coherent uncacheable, coherent write-through, and coherent write-combining respectively. PTE.CP never weakens an MMIO execution rule.

Hardware sets PTE.A on each structurally valid descriptor consumed by a completed walk and may also set PTE.A during a speculative walk whose instruction later faults or is squashed. In that latter case, PTE.A is an imprecise, PTE-owned usage hint; it is not instruction progress, restart state, or generic continuation state. PTE.D is set only when the corresponding store or atomic update commits, and every required PTE.D update succeeds before its memory update becomes visible. Hardware PTE.A/PTE.D changes are conditional atomic updates of the complete 64-bit PTE and preserve every other field. A concurrent change causes a re-read and complete revalidation; stale fields are never written back. PTQUERY and VTOP do not modify PTE.A or PTE.D.

7.6 Translation-Cache Identity and Invalidation

A translation-cache entry is identified by the components below, including the leaf-aligned linear range that it maps. Translation-cache identity excludes PTCR.ROOT_PAGE. While ASCR.AE is one, system software binds one ASCR.ASID to one PTCR root and translation-table format; it completes the required shootdown before reusing that ASCR.ASID for a different PTCR image. While ASCR.AE is zero, the current untagged context is invalidated when PTCR changes.

No translation-cache entry may be interpreted under a different PTCR.TT value from the one that produced it.

Table 7-5. Translation-Cache Entry Identity

Identity Component	Applies To
leaf-aligned linear mapping range	every entry
PTCR.TT	every entry
ASCR.ASID	non-global entry while ASCR.AE is one

Table 7-6. Local Translation-Cache Transitions

Operation	Non-Global Entries	Global Entries
WRCR with changed PTCR.ROOT_PAGE	invalidate all current untagged entries when ASCR.AE is zero; when ASCR.AE is one software must invalidate the ASCR.ASID before rebinding it	preserve only mappings whose complete leaf attributes are identical in every context
WRCR with changed PTCR.TT	invalidate entries using the old translation-table format	invalidate entries using the old translation-table format
WRCR changing ASCR.ASID while ASCR.AE remains one	select the new ASCR.ASID-tagged entries	preserve
WRCR changing ASCR.AE	invalidate the old untagged context or select the new tagged context as applicable	preserve
SWPT	install PTCR and invalidate the current untagged context	preserve only identical global mappings
SWPTA	install PTCR and ASCR and select the new ASCR.ASID-tagged context	preserve only identical global mappings
INVPAGE	invalidate every mapping containing the addressed linear value for the current ASCR.ASID, or the current untagged context when ASCR.AE is zero	invalidate every addressed global mapping containing that linear value
INVASID	invalidate every entry tagged with the selected ASCR.ASID	preserve
INVTLB	invalidate every local entry	invalidate every local entry

Table 7-7. Remote Translation Shootdown Protocol

Step	Required Action
1	PTE store
2	AFENCE
3	local invalidate
4	release shootdown request
5	target acquire and local invalidate
6	target release acknowledgement
7	updater acquire of every acknowledgement
8	AFENCE
9	reclaim or reuse

A global leaf ignores ASID matching. Its PTE.PFN, effective permissions, PTE.AM, and PTE.CP attributes must be identical in every context in which that global entry can be used. Changing any of those attributes requires invalidating the global entry on every target logical processor.

The page-table walker reads page tables as PTE.CP=0 coherent normal memory. Each 64-bit PTE read observes one coherence version; it does not combine fields from different writes. An in-progress walk may complete with the old or new coherence version, so the updater does not reclaim the old mapping until the complete shootdown protocol finishes. The local invalidation instructions affect only the executing logical processor; the request and acknowledgement steps are ordinary shared-memory communication ordered by the stated release, acquire, and AFENCE operations.

7.7 Leaf Byte Addresses and Access Ranges

Every leaf at level L forms a byte address from PTE.PFN, where $o(L1) = 14$, $o(L2) = 23$, $o(L3) = 34$, and $o(L4) = 45$:

$$byte\_address = (\texttt{PTE.PFN} \ll 14) \mid linear[o(L) - 1 : 0].$$

A width-n access covers the half-open byte interval $[a, a + n)$. Bit 7 records PTE.A and may be set by an ordinary walk. Atomics require natural byte alignment; a misaligned operand raises ATOMIC_ALIGNMENT before the memory read or write. PTQUERY returns the raw selected descriptor, while VTOP reports the translated byte address; neither instruction modifies PTE.A or PTE.D.

7.8 Physical Class and MMIO Accesses

The platform assigns each physical address an authoritative Normal or Device class. A Normal leaf selecting Device physical memory raises MEMORY_TYPE_FAULT. A MMIO leaf may select either physical class and imposes MMIO rules; paging-disabled Device accesses also use those rules. Paging-disabled Normal addresses are Normal accesses.

A permitted MMIO operation is one naturally aligned scalar load or store of 1, 2, 4, or 8 bytes. All complete-range and translation checks finish before one architectural MMIO event is produced, and that event is not split. A permitted scalar that is misaligned raises MMIO_ALIGNMENT_FAULT. Atomic, read-modify-write, repeated, and every other operation that is not one permitted scalar load or store raise UNSUPPORTED_MMIO_OPERATION before alignment is considered and before any target effect. A prefetch to MMIO is a no-op. Instruction fetch is never permitted from MMIO.

Within one memory operand, failures are selected in this order: segment range and wrap, canonicality, complete translation of all covered mappings in increasing address order, physical memory class, MMIO operation class, MMIO or atomic alignment, then the target access and bus fault. Thus a later mapping's translation fault precedes an MMIO condition already visible in an earlier mapping.

7.9 Multi-Page Accesses

For a byte range that intersects more than one distinct leaf mapping, translation and accumulated permission checks are resolved once per mapping in increasing linear-address order. Each mapping's walk reaches its existing not-present, structural-validity, and permission result before the next mapping is consumed. The first mapping in that order that fails supplies the architectural fault and walk level.

Every covered leaf mapping of a store is validated before any byte of the store becomes visible. A later-mapping translation, permission or synchronous data-access fault leaves every destination byte and every leaf `PTE.D` bit unchanged. `PTE.A` updates completed while consuming an earlier valid entry retain the speculative-A rule and may remain set when a later mapping faults.

Instruction-record fetch uses the same increasing-address rule for all bytes required by the encoded record. A fault in a later mapping completes as an instruction-fetch fault before decode or operand evaluation, while earlier instruction-cache and `PTE.A` effects retain their ordinary rules.

8 INSTRUCTION EXECUTION MODEL

This section defines the common execution rules used by the instruction descriptions. An individual instruction may add stricter rules, but it does not silently replace the rules below.

8.1 Architectural Result Priority

When one instruction could encounter more than one exceptional condition, the architecturally reported result is the first applicable condition in the following order.

1. Instruction fetch and fetch translation.
2. Instruction framing and acquisition of the complete instruction record.
3. Opcode recognition.
4. Availability of every extension required by the decoded operation.
5. Fixed-field, reserved-field, and effective-address-form legality.
6. Privilege checks.
7. Selector, control-image, and other architectural-state validation.
8. Predicate evaluation for a conditional operation.
9. Every enabled operand access in architectural access order.
10. Execution-stage faults defined by the decoded operation.

A false predicate suppresses the conditional operation's target-memory and implicit stack accesses. It does not suppress instruction framing, opcode and extension checks, encoding legality, privilege checks, or control-state validation. Explicit operands are accessed in instruction operand order unless an instruction states another order. For a memory-to-memory operation, the complete source read precedes access to the destination, so a source-access failure has priority over a destination-access failure.

Implicit accesses use the order stated by the instruction. In particular, PUSH captures its source before accessing the destination stack slot; POP reads the stack slot before validating or writing its destination; PUSHP uses listed register order and POPP uses reverse listed order; CALL validates its target before accessing the return stack; RET reads the return stack before validating its target; and Long Call, Long Return, Event Return, and supervisor-entry instructions use the step order in their instruction descriptions.

8.2 Implicit Stack Operations

Ordinary implicit stack operations capture the current SS and SP before their first stack access. Every stack slot is 64 bits and is addressed through the captured SS. The ranges below are expressed in terms of the captured SP_{old}; the applicable segment, translation, and complete-range checks finish before the operation commits.

Table 8-1. Ordinary Implicit Stack Operations

Operation	Stack range	Committed SP	Slot layout or value order
`PUSH`, `CALL`	$[SP_{\text{old}} - 8, SP_{\text{old}})$	$SP_{\text{old}} - 8$	one source value or the following instruction's continuation PC at $SP_{\text{old}} - 8$
`POP`, `RET`	$[SP_{\text{old}}, SP_{\text{old}} + 8)$	$SP_{\text{old}} + 8$	one destination value or continuation PC read at SP_{old}
`PUSHP`	$[SP_{\text{old}} - 16, SP_{\text{old}})$	$SP_{\text{old}} - 16$	listed pair's second register at $SP_{\text{old}} - 16$ and first register at $SP_{\text{old}} - 8$
`POPP`	$[SP_{\text{old}}, SP_{\text{old}} + 16)$	$SP_{\text{old}} + 16$	listed pair's second register from SP_{old}, then first register from $SP_{\text{old}} + 8$
`LCALL`	$[SP_{\text{old}} - 16, SP_{\text{old}})$	$SP_{\text{old}} - 16$	continuation PC at $SP_{\text{old}} - 16$ and return CS image at $SP_{\text{old}} - 8$
`LRET`	$[SP_{\text{old}}, SP_{\text{old}} + 16)$	$SP_{\text{old}} + 16$	continuation PC at SP_{old} and return CS image at $SP_{\text{old}} + 8$

The complete range and all source values are established before a push operation changes memory or SP. A pop operation reads its complete range and completes any applicable destination validation before changing a destination or SP. A successful operation commits all listed stack and register effects together; a preceding fault exposes none of them.

Architectural event entry and ERET use the return source selected by the Architectural Event Processing Model. SYSCALL is the SYSTEM_CALL event; user-origin entry keeps its continuation in the user context bank described by the Privileged Execution Model and creates no memory frame.

8.3 Floating-Point Implicit Stack Operations

`FPUSHP` and `FPOPP` follow the base implicit-stack rules for register-pair operations. FPUSHP writes the listed pair's second register at $SP_{\text{old}}-16$ and first register at $SP_{\text{old}}-8$, then commits $SP_{\text{old}}-16$. FPOPP reads the listed pair's second register at SP_{old} and first register at $SP_{\text{old}}+8$, then commits $SP_{\text{old}}+16$. Both operations establish the complete 16-byte range and all values before committing any register or SP effect.

8.4 Operand Evaluation and EA Defaults

Operands are decoded in instruction order. Source reads complete before the final destination write unless an atomic instruction says otherwise. Effective-address calculation may produce an address without reading memory.

Auto-update EA forms update temporary operand-evaluation state. The architectural register update becomes visible only when the instruction commits.

Ordinary instructions accept at most one memory operand. The explicitly encoded memory-memory exceptions are CMP, EXTSL, EXTSQ, EXTSW, EXTZL, EXTZQ, EXTZW, MOV, MOVCU, MOVUC, MOVUU, FBNDII, FBNDIX, FBNDXI, FBNDXX. Repeat behavior is available only through the repeat instructions and body-eligibility rules defined in Repeated Scalar Execution.

An instruction either replaces the complete `FLAGS.Z`, `FLAGS.N`, `FLAGS.C`, and `FLAGS.V` FLAGS image or preserves the complete image; partial FLAGS writes do not exist. Operand evaluation follows instruction operand order, and an instruction has one architectural commit point unless its description explicitly defines partial completion.

8.5 Shared Side-Effect Rules

An instruction that faults before commit leaves destination registers, destination memory, and auto-update register effects unchanged unless the instruction explicitly defines partial completion.

Atomic read-modify-write instructions have a single architectural commit point for the memory update and any associated register result. Cache, TLB, and system-state instructions describe their own completion and visibility rules.

Ordinary integer ALU instructions use the common one-memory-operand rule. Compare and test instructions may read two operands and update FLAGS without storing their temporary result. EXTS and EXTZ additionally provide explicitly encoded register-memory and memory-memory conversion stores; EXTSQ also provides register-register and memory-register sign extension. These operations preserve FLAGS unless a repeat rule observes a derived value. Data movement, LEA, and segment-address operations also preserve FLAGS unless their descriptions say otherwise.

Control transfers make PC and CS changes at a single control-transfer commit point. Atomic operations require a naturally aligned addressable memory operand and commit the read-modify-write as one architectural update. TLB, page-table context, and privileged control operations require supervisor privilege. Cache-maintenance operations preserve FLAGS and define their own visibility rules.

9 FLAGS AND CONDITION CODES

9.1 Condition Encoding

Conditional instructions encode a four-bit condition code. The zero-valued condition is T, which is also used for canonical aliases such as JMP for Jcc.T.

The condition-code bits are the low four meaningful bits of FLAGS, as shown in the Programming Model section.

Table 9-1. Condition Code Encoding

Bits	Name	Aliases	Expression
0000	T	-	true
0001	F	-	false
0010	EQ	Z	FLAGS.Z == 1
0011	NE	NZ	FLAGS.Z == 0
0100	ULT	C	FLAGS.C == 1
0101	UGE	NC	FLAGS.C == 0
0110	MI	N	FLAGS.N == 1
0111	PL	NN	FLAGS.N == 0
1000	VS	V	FLAGS.V == 1
1001	VC	NV	FLAGS.V == 0
1010	ULE	-	FLAGS.C == 1 || FLAGS.Z == 1
1011	UGT	-	FLAGS.C == 0 && FLAGS.Z == 0
1100	LT	-	FLAGS.N != FLAGS.V
1101	GE	-	FLAGS.N == FLAGS.V
1110	LE	-	FLAGS.Z == 1 || FLAGS.N != FLAGS.V
1111	GT	-	FLAGS.Z == 0 && FLAGS.N == FLAGS.V

9.2 Flag Meanings

Integer condition codes are held in FLAGS. Integer instructions leave FLAGS unchanged unless their instruction description explicitly defines a FLAGS write. Every FLAGS-writing instruction replaces the complete `FLAGS.Z`, `FLAGS.N`, `FLAGS.C`, and `FLAGS.V` image; partial FLAGS writes do not exist. An instruction that does not write FLAGS preserves the complete image.

For an operand width of n bits, ordinary integer ALU results are evaluated modulo 2^n unless the instruction explicitly defines a wider intermediate.

Explicit FLAGS-writing integer instructions include CMP, TEST, ADC, SBB, INCF, DECF, BTEST, BSET, BCLR, BCHG, SETF, WRFLAGS, CMPXCHG, SEGLEA, and the bounds-check instructions. CMPJcc and TESTJcc compute temporary condition flags for their branch decision and do not update architectural FLAGS. ADD, SUB, logic operations, INC, DEC, shifts, rotates, moves, extensions, min/max, and multiply/divide instructions leave FLAGS unchanged.

Table 9-2. Integer Condition-Code Bits

Bit	Name	Meaning
`FLAGS.Z`	zero	Set when the result value is zero.
`FLAGS.N`	negative	Set from the most significant result bit at the operand size.
`FLAGS.C`	carry/borrow	Set on unsigned carry out for addition or unsigned borrow for subtraction.
`FLAGS.V`	overflow	Set on signed overflow or an explicitly reported exceptional condition.

9.3 Conditional Consumers

Conditional branches, calls, traps, and sets consume the condition-code table without recomputing FLAGS. An ordinary conditional consumer observes the FLAGS image produced by the last committed FLAGS-writing instruction. REPcc instead tests a temporary flag image derived from the body instruction's repeat-observed value, as defined in Repeated Scalar Execution. That test leaves architectural FLAGS unchanged.

A false condition suppresses the conditional action unless the instruction description explicitly defines a different commit rule, such as a repeat instruction's terminating iteration rule.

9.4 Floating-Point Interactions

Floating-point comparison instructions may update integer condition codes only when their instruction definition says so. FFLAGS holds floating-point exception flags, and FSTATUS holds the corresponding exception-condition enables and rounding state. IEEE-754 FFLAGS causes use independent accrued semantics. The complete-image rule applies to integer FLAGS.

Integer conditional consumers do not read FFLAGS directly; floating-point instructions that want integer control-flow consumers must materialize the selected condition into FLAGS or an Rn value.

10 MEMORY MODEL

The memory model defines instruction fetches, loads, stores, atomic read-modify-write operations, cache maintenance, and translation-cache maintenance after EA calculation and address translation.

10.1 Baseline Ordering

Normal memory is coherent. Every logical processor observes writes to a given byte in one common coherence order, and writes by one logical processor to that byte enter the order in program order. A naturally aligned tear-free scalar store enters that order as one memory write event.

Normal-memory writes are multi-copy atomic. When a memory write event becomes observable by a logical processor other than the writer, it becomes observable to all such logical processors at the same architectural point. An implementation may still forward a logical processor's own pending store to its later loads before that store is globally observable.

The baseline ordering is weak. Except for same-byte coherence or ordering imposed by an atomic memory-order selector or fence, loads and stores to different locations may be issued, completed, and observed in an order different from program order. Coherence and multi-copy atomicity do not by themselves order accesses to different locations.

10.2 Ordinary Load Values and Preserved Order

Each byte returned by a load comes from the initial value of that physical byte or from an actual store or successful atomic read-modify-write to that byte. A load does not obtain a value solely from a cyclic chain in which the load value is needed to produce the write that would supply it.

Among writes already globally observable when a load takes its value, the load observes the latest write in the byte's coherence order. A load may instead forward a byte from an earlier store by the same logical processor before that store is globally observable. A store later than the load in program order cannot supply the load.

Program order is preserved from an earlier read or write to a later write when their byte ranges overlap. Two loads of the same byte, with no intervening write by the same logical processor to that byte, do not observe the byte's coherence order moving backward. These rules apply independently to every byte of an access, including each visible portion of an unaligned access.

A load is ordered before a later memory access whose effective address depends on the loaded value. A load is also ordered before a later store when the stored value depends on the load or when execution of that store is control-dependent on the load. Control dependence alone does not order a later load; software uses an appropriate fence when that ordering is required.

10.3 PTE Cache Policies

The two-bit `PTE.CP` selects a cache policy for byte-addressed normal memory. Every policy uses the same coherence order, multi-copy atomicity, tearing rules, atomic operations, and fence semantics defined in this chapter. The policy changes allocation, retained cache state, and store combination; it does not create a separate value or ordering domain.

Table 10-1. Normal-Memory Cache Policies

PTE.CP	Policy	Reads and Fetches	Stores	Retained State
0	Write-back cacheable	may allocate a coherent cache copy	may retire into a coherent dirty cache copy	clean or dirty coherent cache copies
1	Coherent uncached	enter coherence without retaining a cache copy	enter coherence before retirement without retaining a dirty copy	no retained cache copy
2	Coherent write-through	may allocate a coherent clean cache copy	enter coherence before retirement without retaining a dirty copy	clean coherent cache copies only
3	Coherent write-combining	enter coherence without retaining a cache copy	may combine adjacent byte writes before entering coherence	pending combined stores, but no retained cache copy

Aliases of one physical byte with different PTE.CP values participate in the same coherence order. A load, instruction fetch, or atomic operation through any alias observes the same physical value subject to ordinary ordering, and atomic operations serialize at the physical location. PTE.CP2 stores enter coherence before retirement and leave no dirty cache copy. PTE.CP3 combined stores preserve the byte writes and their per-location program order.

WFENCE and AFENCE complete earlier pending PTE.CP3 stores. WRBKDCACHE, FLSHDCACHE, and SYNCCACHE complete pending PTE.CP3 stores for their selected block before performing their named cache action. The data-write, cache-maintenance, rendezvous, and local instruction-cache sequence below applies without regard to the PTE.CP value used by the modified code mapping.

10.4 Access Atomicity and Alignment

Unaligned ordinary normal-memory accesses are architecturally permitted. A completed unaligned load may observe bytes from multiple memory events in coherence order, and a completed unaligned store may enter coherence as separately visible portions. Each visible store portion is coherent and multi-copy atomic.

An unaligned store is nevertheless fault-atomic for synchronous faults. If any byte of the complete store range would raise a synchronous fault, no byte of the store becomes architecturally visible.

Atomic read-modify-write instructions require natural byte alignment. A misaligned atomic operand raises ATOMIC_ALIGNMENT_FAULT before any memory update or architectural result commits; it must not complete as a non-atomic update.

For every scalar size, a naturally aligned normal-memory load or store is tear-free. Unaligned accesses retain the successful-access and synchronous-fault rules above.

Natural alignment equals the operand width in bytes: B is one byte and may use any byte address; W, L, and Q are two, four, and eight bytes and require addresses divisible by two, four, and eight respectively.

Stores to memory that is later executed as instructions are ordinary data stores until an explicit synchronization sequence makes the modified bytes visible to instruction fetch.

10.5 Cache-Maintenance Blocks

CPUID `CACHE_TOPOLOGY` reports a cache-maintenance granule G. A cache-maintenance instruction computes and translates its address-role EA without reading an operand value, then selects the physical byte interval $[\lfloor A/G \rfloor G, \lfloor A/G \rfloor G + G)$ containing the translated physical address A. One instruction operates on exactly that block. Because G is at most 4096 bytes, the selected physical block does not cross a 16 KiB page boundary.

FLSHDCACHE, INVDCACHE, and WRBKDCACHE apply to every data or unified cache copy of the selected block in its normal-memory coherence domain. FLSHDCACHE writes dirty bytes into normal-memory coherence and invalidates the copies. INVDCACHE discards the copies without writeback. WRBKDCACHE writes dirty bytes into normal-memory coherence and retains clean copies. Each operation completes the selected block before retirement.

INVICACHE invalidates the selected block in the executing logical processor's instruction cache, decoded instruction state, and instruction-prefetch state. SYNCCACHE first completes the data writeback for the block throughout the coherence domain, then performs the same local instruction-side invalidation, and finally serializes later instruction fetch so that it observes the synchronized bytes. To publish modified code to another logical processor, software completes SYNCCACHE on the publisher, performs a release/acquire rendezvous, and executes INVICACHE for the block on every receiving logical processor before branching to the modified bytes.

Address translation and permission checks precede a block's cache effects. A fault leaves that instruction's selected block unchanged.

10.6 Atomic Memory-Order Selectors

Table 10-2. Atomic Memory-Order Selectors

Selector	Code	Architectural Ordering Effect
`RELAXED`	`0`	Atomicity only. No ordering is imposed on unrelated memory operations.
`ACQUIRE`	`1`	Later memory operations by the same logical processor are ordered after the atomic operation.
`RELEASE`	`2`	Earlier memory operations by the same logical processor are ordered before the atomic operation.
`ACQREL`	`3`	Applies both acquire and release ordering.
`SEQCST`	`4`	Applies acquire-release ordering and participates in the single global sequentially consistent order.

`CMPXCHG`, `FETCHADD`, `FETCHSUB`, `FETCHAND`, `FETCHOR`, and `FETCHXOR` are the atomic read-modify-write instructions. Each performs its memory read, successful memory write, and architectural register result as one indivisible operation in the coherence order of the addressed physical location.

A successful release or stronger atomic write heads a release sequence. The sequence continues through immediately following successful atomic read-modify-write operations to the same location in coherence order and ends before any other write. An acquire or stronger atomic operation that takes its value from any member of the sequence observes the effects ordered before its head before effects ordered after the acquiring operation.

One encoded selector governs the complete CMPXCHG operation on both success and failure. A failed CMPXCHG contributes a memory read event. Its acquire component orders later memory operations after that read, and its release component orders earlier memory operations before that read. On success, the memory write event carries the selected release effect to observers of that write; on failure, the read is the operation's memory event. A failed CMPXCHG contributes no write and therefore creates no release sequence. A failed CMPXCHG.SEQCST additionally participates in the global sequentially consistent order as an SC load.

10.7 Fence Instructions

The table gives every ordered-before pair imposed by a fence on memory operations issued by the same logical processor. An entry marked ordered places every earlier operation of the row kind before every later operation of the column kind.

Table 10-3. Fence Ordered-Before Pairs

Fence	read to read	read to write	write to read	write to write
RFENCE	ordered	baseline	baseline	baseline
WFENCE	baseline	baseline	baseline	ordered
AFENCE	ordered	ordered	ordered	ordered

A baseline entry retains the weak-ordering rules of this chapter. AFENCE is cumulative: memory effects observed before the fence are propagated before effects ordered after it, including through another logical processor. The linked instruction entries define the matching completion behavior.

10.8 Global Sequentially Consistent Order

All AFENCE operations and all SEQCST atomic operations participate in one global total order that is consistent with each logical processor's ordered-before relation and with per-location coherence. A failed CMPXCHG.SEQCST participates in that order as an SC load even though it performs no write.

A naturally aligned tear-free scalar load or store also participates as an SC operation when it is the only memory access between its immediately surrounding AFENCE operations in memory-event order. Thus AFENCE; access; AFENCE is the architectural sequence for an SC scalar load or store. The access and both fences occupy their required positions in the same global order.

Acquire and release operations may be strengthened with AFENCE; the resulting instruction sequence has the union of the ordering constraints imposed by its operations. Leaf PTE.AM and PTE.CP are orthogonal. An MMIO access retains its non-speculative, operation-class, alignment, and ordering rules under every PTE.CP value. MMIO accesses from one logical processor are observed in program order relative to other MMIO accesses from that logical processor. Normal-memory and MMIO accesses retain the baseline weak ordering between classes; AFENCE orders earlier accesses before later accesses across that boundary.

11 REPEATED SCALAR EXECUTION

`REP Rn, (<instruction>)` and `REPcc Rn, (<instruction>)` repeat one scalar instruction. The body is decoded and checked for repeat eligibility before a zero-count annul is applied, then remains fixed for that repeat context. A body may not be a repeat operation, a repeat-control instruction, or a control transfer.

Entering repeat state captures the selected Rn value as an unsigned remaining count. A valid zero count performs no body side effects. Otherwise each iteration evaluates operands and commits the body's register, memory, and auto-update effects according to its ordinary scalar rules. A body read of the selected counter register observes the iteration-start architectural value; a body write to that register is ignored.

Each successfully committed iteration decrements the captured count before an event may be admitted at the next iteration boundary. A condition-false `REPcc` iteration still commits its body and decrements the count before repeat state is cleared. If the body faults, that iteration commits nothing and does not decrement the count; earlier iterations remain committed. Event entry then saves the repeat-prefix address in `SAVED_PC` and clears hidden repeat state, so ERET returns to the prefix and re-executes it normally with the unchanged counter.

The `REPcc observation` attribute identifies the value tested after the body executes. A `result:operand` or `source:operand` observation names an operand; `computed` names the derived value defined by that instruction. The temporary condition image sets Z exactly when the observed value is zero, sets N from its most significant bit at the observation width, and clears C and V. It does not sample or overwrite architectural FLAGS, although the body's own FLAGS effects commit normally. `REP` is the unconditional T spelling; condition F is reserved.

While repeat state is active, architectural PC identifies the body instruction. After a successful body commit, the counter decrement completes before an event may be admitted between iterations. Both that event path and the fault path above save the repeat-prefix address in `SAVED_PC` and clear hidden repeat state. ERET restores the ordinary saved architectural state and begins execution at that prefix address; it does not restore hidden repeat continuation state. The prefix is therefore decoded and executed normally, using the already-decremented architectural Rn value. `STATUS.TF` and `STATUS.RF` apply once to each committed body iteration; a zero-count annul is also one trace unit.

PART III

BEDROCK ARCHITECTURE

System Programming

12 PRIVILEGED EXECUTION MODEL

All exceptions, interrupts, NMI, and explicit synchronous traps enter supervisor execution through the architectural event mechanism. SYSCALL is the explicit SYSTEM_CALL event (0x00000003); it uses the common EPC/ECS/EDS entry and returns with ERET.

12.1 Privilege and Event State

STATUS.PM is zero in user mode and one in supervisor mode. STATUS.EA indicates active event processing. STATUS.EDEPTH in bits 9..6 is the active event depth, and STATUS.UO in bit 10 records that the outer event originated in user mode. STATUS.PM, STATUS.EA, STATUS.EDEPTH, and STATUS.UO are hardware managed; WRSTATUS must preserve them. User execution always observes STATUS.PM=STATUS.EA=STATUS.EDEPTH=STATUS.UO=0.

STATUS.IE enables maskable interrupts, STATUS.TF requests single-step tracing, STATUS.RF suppresses one trace boundary, and STATUS.NI inhibits nested NMI admission. RDCR, WRCR, translation-state changes, event-control changes, and page-table context switches require supervisor privilege unless an instruction explicitly states otherwise.

12.2 User Context Bank

The user context bank consists of UPC, USP, UCS, UDS, USS, UCTL, and UINFO. UCTL contains saved UCTL.FLAGS in bits 15..0, saved UCTL.STATUS in bits 31..16, and UCTL.V in bit 32. UINFO contains UINFO.EVENT_CODE; its upper half is zero. A saved user STATUS image has STATUS.PM, STATUS.EA, STATUS.EDEPTH, and STATUS.UO clear.

While UCTL.V is zero, WRCR to UPC, USP, UCS, UDS, USS, or UINFO atomically stages the selected 64-bit value without validating the complete bank; UINFO still requires bits 63 through 32 to be zero. Writing UCTL.V=1, or writing another bank member while the resulting UCTL.V remains one, validates the complete candidate bank before atomically replacing the selected register. Complete-bank validation includes the event-code-dependent UINFO shape, valid user segment images and pointers, an executable UPC, and the required zero saved STATUS fields.

Software stages a context by writing UPC, USP, UCS, UDS, USS, and UINFO before writing UCTL.V=1. The bank belongs to the interrupted user thread and must be saved with its supervisor context across preemption or migration.

12.3 Initial Event Stacks and Payloads

A first user-origin SYSTEM_CALL selects SSS:SSP, a maskable interrupt selects ISS:ISP, and an exception or NMI selects FSS:FSP. BASIC events, including SYSTEM_CALL, have no payload and perform no stack-memory access. ERROR events store 16 bytes; PAGE and AUXILIARY events store 32 bytes. The event code in UINFO defines the payload shape.

A supervisor-origin event and every nested event push a complete event frame on the current SS:SP. They do not promote to another configured event stack. The saved SS:SP in the frame identifies the interrupted stack.

12.4 Event Entry and Failure

User-origin entry atomically saves the user context bank, optionally stores the payload, loads EPC/ECS/EDS and the selected initial stack, sets STATUS.PM and STATUS.EA, sets STATUS.EDEPTH to one and STATUS.UO to one, and clears STATUS.IE, STATUS.TF, and STATUS.RF. Supervisor and nested entry save the prior context in the full frame and increment STATUS.EDEPTH while preserving STATUS.UO.

Delivery failure selects DSS:DSP and attempts the corresponding EVENT_DELIVERY_FAILURE leaf event. The hidden current-DFA bit is set when that entry commits. Failure while current DFA is set, or failure of that delivery attempt, enters shutdown. A full frame saves the prior DFA state in FRAME_CONTROL.SAVED_DFA.

12.5 ERET Routing and Atomicity

ERET classifies its source before any stack access. STATUS.PM=1, STATUS.EA=0, STATUS.EDEPTH=0, STATUS.UO=0, and UCTL.V=1 performs direct initial user entry from the bank. STATUS.PM=1, STATUS.EA=1, STATUS.EDEPTH=1, STATUS.UO=1, and UCTL.V=1 returns the outer user event from the bank. Other supervisor states with STATUS.EA=1 and nonzero STATUS.EDEPTH restore a full frame from the current SS:SP. Every other combination raises INVALID_CONTROL_TRANSITION.

A U-bank return validates UCTL, UINFO, the user segment images, USP, and the executable UPC before committing. It restores user execution, clears UCTL.V, STATUS.EDEPTH, STATUS.UO, STATUS.EA, and current DFA, and never reads the stack. A stacked return validates the entire frame, requires saved STATUS.EDEPTH plus one to equal the current depth and saved STATUS.UO to match the current chain, then restores FRAME_CONTROL.STATUS and FRAME_CONTROL.SAVED_DFA atomically. A failed validation or target probe leaves the current context and return source unchanged.

Normative instruction-local behavior is defined by SYSCALL and ERET.

13 ARCHITECTURAL TIME

Each logical processor owns one 64-bit invariant timebase. The value advances modulo 2^{64} at the exact, nonzero integer number of ticks per second reported by CPUID TICKS_PER_SECOND. Its rate is independent of processor clock-frequency changes and the implementation-cycle counter. In particular, PMC.EN does not affect the timebase, and CYCLE is not an elapsed-time substitute.

The timebase continues to advance while the logical processor is halted or blocked by WAIT. RDTIME returns one atomic snapshot and is available in user and supervisor modes. Both modes observe the same value. The snapshot is elapsed-time state, not wall-clock or calendar time, and RDTIME is neither a memory access nor a memory fence.

Warm RESET preserves the timebase value and frequency. Cold reset initializes the value to zero and begins a new epoch; the architecture does not order values from different cold-reset epochs. The timebase and its deadline timer are per-logical-processor runtime resources, not task computational state, and no state-record instruction transfers them.

13.1 One-Shot Deadline Timer

Each logical processor owns one one-shot deadline timer consisting of armed state, an absolute timebase deadline, and a 24-bit interrupt identity. TARM and TCANCEL are the only architectural programming operations. The timer state has no control-register or state-record projection.

TARM observes the current time and classifies its absolute deadline by signed modular distance:

$$distance = \mathrm{signed64}(deadline - time).$$

A positive distance arms the timer and atomically replaces any previous arm. Software can therefore schedule at most $2^{63} - 1$ ticks into the future with one operation. A zero or negative distance is already reached: TARM posts the supplied identity to the local interrupt file and leaves the timer disarmed. The supplied 64-bit identity value must be from 1 through `ICAP.MAX_ID`. An invalid identity raises `INVALID_CONTROL_SELECTOR` without changing timer state.

When time reaches an armed deadline, posting the configured identity and clearing armed state are one atomic transition. Expiry is one-shot; periodic behavior requires software to install another absolute deadline. TCANCEL atomically clears armed state and is a no-op when already disarmed. Neither TARM nor TCANCEL clears a pending bit created by an earlier expiry.

Expiry posts pending state regardless of the identity's enable bit, ITHRESH, `STATUS.IE`, `ECR.V`, or event depth. Those gates remain owned by the interrupt file and event-admission model. A halted logical processor leaves HALT only if the resulting interrupt becomes admissible; otherwise the identity remains pending. A WAIT may return spuriously so that an admissible interrupt can be processed, while time advancement and posting remain independent of that return.

Time advancement, expiry, TARM, TCANCEL, interrupt-file pending operations, and event-entry commit are observed in one of their legal serial orders. A cancellation ordered before expiry prevents the post; a cancellation ordered after expiry cannot remove it. An expiry ordered before replacement may leave the old identity pending, whereas a replacement ordered first prevents the superseded deadline from posting. No operation loses a post that has already committed. These operations do not by themselves order memory operations.

Warm and cold reset disarm the timer. Reset does not clear interrupt-file pending state except through the interrupt file's own reset transition.

14 ARCHITECTURAL EVENT PROCESSING MODEL

An *architectural event* is a synchronous or asynchronous condition delivered through the common event-entry mechanism. Synchronous exceptions are faults or traps. Asynchronous events are maskable interrupts or NMIs. Fault and trap are the two exception subtypes. SYSCALL produces the explicit synchronous `SYSTEM_CALL` exception (event code `0x00000003`) and uses this same mechanism.

Every event transfers to the exact program counter in EPC after loading ECS and EDS. Hardware supplies an event code and either a user-bank payload or a full supervisor frame whose layout is fixed by the event source. Software begins at one common entry point and dispatches by the event code.

Related normative instruction entries are `BKPT`, `SYSCALL`, `ERET`, and `RESET`.

14.1 Event Code and Sources

Every delivered event carries a 32-bit code. The high byte is the event class and the low 24 bits are an ID in that class's namespace.

Figure 14-1. Architectural Event Code

Table 14-1. Architectural Event Classes

Value	Class	Name	Selector policy
`0x00`	`EXCEPTION`	Synchronous Exception	fixed 24-bit selector
`0x01`	`INTERRUPT`	Maskable Interrupt	identity 24-bit selector
`0x02`	`NMI`	Non-Maskable Interrupt	source 24-bit selector

Class values not listed above are reserved. Fixed selectors are assigned by this specification; platform and source selectors are supplied by the corresponding event source.

The following fixed event codes are assigned by this specification. Each row answers only which symbolic event is assigned to a fixed code; the event's behavior, frame, and payload are defined by the topic that owns that behavior.

Table 14-2. Fixed Architectural Event-Code Assignments

Code	Event
`0x00000000`	`SINGLE_STEP`
`0x00000001`	`EXPLICIT_BREAKPOINT`
`0x00000006`	`EXECUTION_BREAKPOINT`
`0x00000007`	`MEMORY_READ_WATCHPOINT`
`0x00000008`	`MEMORY_WRITE_WATCHPOINT`
`0x00000009`	`ATOMIC_MEMORY_WATCHPOINT`

Table 14-2 (continued)

Code	Event
0x00000002	PRIVILEGE_VIOLATION
0x00000003	SYSTEM_CALL
0x00000004	DIVIDE_BY_ZERO
0x00000005	SIGNED_DIVIDE_OVERFLOW
0x00000010	INVALID_OPCODE
0x00000011	INVALID_ADDRESSING_FORM
0x00000012	RESERVED_INSTRUCTION_ENCODING
0x00000013	UNAVAILABLE_INSTRUCTION_EXTENSION
0x00000014	EXPLICIT_ILLEGAL_INSTRUCTION
0x00000015	TRUNCATED_INSTRUCTION
0x00000016	INVALID_OPERAND_RELATION
0x00000020	PAGE_NOT_PRESENT
0x00000021	PAGE_PERMISSION_VIOLATION
0x00000022	MALFORMED_PAGE_TABLE_ENTRY
0x00000023	NONCANONICAL_ADDRESS
0x00000024	SEGMENT_BOUNDS_VIOLATION
0x00000025	ATOMIC_ALIGNMENT_FAULT
0x00000026	MEMORY_TYPE_FAULT
0x00000030	PHYSICAL_ADDRESS_FAULT
0x00000031	MMIO_ALIGNMENT_FAULT
0x00000032	UNSUPPORTED_MMIO_OPERATION
0x00000040	INVALID_CONTROL_SELECTOR
0x00000041	RESERVED_CONTROL_BITS
0x00000042	INVALID_CONTROL_IMAGE
0x00000043	INVALID_CONTROL_TRANSITION
0x00000044	CONTROL_FLOW_INTEGRITY_VIOLATION
0x00000050	VECTOR_LOOP_OFFSET_OUT_OF_RANGE
0x00000051	VECTOR_LANE_INDEX_OUT_OF_RANGE
0x00000060	FLOATING_POINT_EXCEPTION
0x00000070	BUS_NO_RESPONDER
0x00000071	BUS_ACCESS_DENIED
0x00000072	BUS_TIMEOUT
0x00000073	BUS_DATA_ERROR
0x00000074	BUS_OTHER_ERROR
0x00000078	EVENT_ENTRY_STATE_FAILURE
0x00000079	EVENT_STACK_STATE_FAILURE
0x0000007A	EVENT_FRAME_ADDRESS_FAILURE
0x0000007B	EVENT_FRAME_STORE_FAILURE
0x0000007C	MACHINE_CHECK

Unassigned base-profile exception IDs are reserved. Interrupt identities do not share a namespace with exception IDs. Interrupt-controller routing, native-ID mapping, ownership, trigger mode, and any external acknowledgement or end-of-interrupt protocol belong to the platform ABI. A valid posted identity is recorded in the target logical processor's architectural interrupt file; the delivered event code has class `0x01` and that identity in its low 24 bits.

Deadline-timer expiry uses the same posting operation with the identity supplied by TARM. It does not define a fixed timer identity or a separate interrupt-source catalog member.

A restartable fault saves the faulting instruction or its architecturally defined restart point. A trap saves the following instruction. An interrupt or NMI saves the next instruction that would otherwise execute. A faulting repeat iteration rolls back without decrementing its counter, then saves the repeat-prefix address. After a successful repeat iteration, the counter decrement commits before an event may be admitted between iterations; that event also saves the repeat-prefix address. Event entry clears hidden repeat state, and ERET executes the saved prefix normally without restoring repeat continuation.

14.2 Event Priority and Debug Stops

When events compete at one architectural boundary, delivery failure is selected before machine check. Instruction and operand validation, address formation, translation, permission, alignment, and other higher-priority synchronous faults precede debug-trigger evaluation. A pre-effect execution breakpoint or memory watchpoint is selected before any result of the stopped instruction and before NMI or a maskable interrupt at the same boundary. After successful completion, an explicit `EXPLICIT_BREAKPOINT` is selected before `SINGLE_STEP`; NMI and maskable interrupts follow automatic debug events. An asynchronous event that is not selected remains pending.

A trace unit is one ordinary committed instruction or one committed REP or REPcc body iteration. Completion of a zero-count REP or REPcc is also one trace unit. `STATUS.TF` is sampled at the start of the unit. If sampled `STATUS.TF` is one and sampled `STATUS.RF` is zero, successful completion commits the unit and then delivers `SINGLE_STEP` with the following trace-unit boundary as its saved PC.

The `TRACE` marker instruction is an ordinary instruction for this rule. Its marker is recorded before a pending SINGLE_STEP is delivered. The Tracing Terms section distinguishes the marker instruction, trace unit, and SINGLE_STEP event.

`STATUS.RF` suppresses SINGLE_STEP and every automatic execution or memory trigger for one trace unit. It does not suppress `EXPLICIT_BREAKPOINT` or the TRACE marker. If sampled `STATUS.RF` is one, successful trace-unit completion clears `STATUS.RF`. A fault or asynchronous event before successful completion preserves it. Event entry saves the old STATUS image and clears live `STATUS.TF` and `STATUS.RF`. Entry for a pre-effect hardware-trigger event first sets `STATUS.RF` in the saved return STATUS image, allowing a normal return to retry the instruction once without an immediate retrigger. A handler may clear that saved `STATUS.RF` bit to request deliberate retriggering.

14.3 Entry Admission and Event Depth

The event-entry state is usable only while `ECR.V` is set. A maskable interrupt is admitted only when `ECR.V`, `STATUS.IE`, and the depth limit all permit it: the current event depth must be less than `ECR.MAX_EDEPTH`. The interrupt file additionally requires a pending and enabled identity that passes ITHRESH; ITOP chooses the numerically lowest such identity. Otherwise the identity remains pending in the interrupt file. A synchronous exception that requires delivery while `ECR.V` is clear cannot be deferred and enters SHUTDOWN. An NMI observed while `ECR.V` is clear sets the hardware-managed `ECR.NMI_P` latch and is not admitted. After `ECR.V` becomes one, hardware

admits that pending NMI before the next instruction boundary when STATUS.NI is clear. While STATUS.NI remains set, the NMI remains pending. Multiple NMI observations coalesce in ECR.NMI_P.

Event depth is zero outside event handling. A successful entry saves STATUS in the selected return context and increments STATUS.EDEPTH. ECR.MAX_EDEPTH equal to zero prevents maskable interrupt admission; values 1 through 15 limit only maskable interrupts and do not suppress synchronous exceptions or NMI. Depth 15 is representable but terminal: any further event requirement enters SHUTDOWN. At depth 14 or less, a delivery failure can still create an EVENT_DELIVERY_FAILURE family frame; at depth 15 there is no representable child frame.

14.4 Supervisor Event Stacks and Nesting

The configured supervisor stack pairs select only the first user-origin entry and delivery-failure recovery.

Table 14-3. Supervisor Stack Roles

Pair	Role	Use
SSS:SSP	system call	first user-origin SYSTEM_CALL
ISS:ISP	interrupt	first user-origin maskable interrupt
FSS:FSP	fault	first user-origin exception or NMI
DSS:DSP	recovery	EVENT_DELIVERY_FAILURE family event

A first user-origin entry loads the applicable configured pair. Supervisor-origin and all nested entries retain the current SS:SP and push a full frame there; event class never promotes an active handler to another configured stack. Delivery failure uses DSS:DSP when the hidden current-DFA bit is clear. Failure with current DFA set enters shutdown.

For user-origin entry, hardware rounds only the event payload to 16 bytes and subtracts it from the configured top. BASIC events have zero payload and perform no stack access. For supervisor or nested entry, hardware subtracts the complete full-frame size from current SP. Each nonzero range is validated and stored atomically before the new SP commits; configured tops are never modified.

Events in the EVENT_DELIVERY_FAILURE family use DSS:DSP. Supervisor-origin and nested events retain the current SS:SP; ERET validates and restores the saved STATUS depth and origin chain.

WRCR to SSS, ISS, FSS, or DSS accepts a complete valid segment image. WRCR to SSP, ISP, FSP, or DSP accepts a complete 64-bit stack top aligned to 64 bytes. A successful write atomically replaces the selected register; event entry and return do not modify the configured segment or top.

14.5 Event Entry State

Before making an event visible, hardware validates EPC, ECS, EDS, the selected stack state, and the complete frame storage range. It then stores the frame, loads the entry segments and PC, sets STATUS.PM and STATUS.EA, and clears STATUS.IE, STATUS.TF, and STATUS.RF as one architectural commit. R0 through R15 and GS0 through GS5 are unchanged; the common entry code preserves whatever its software ABI requires.

WRCR to EPC accepts a complete 64-bit entry address and requires it to be 16-byte aligned, canonical, and executable under ECS. WRCR to ECS or EDS accepts a complete valid segment image. A successful write atomically replaces the selected register.

NMI entry additionally sets STATUS.NI. Other event classes preserve STATUS.NI. A further NMI observed while STATUS.NI is set is recorded by setting ECR.NMI_P. ECR.NMI_P is hardware managed, so WRCR ignores the supplied value for that bit. When entry of an NMI admitted from ECR.NMI_P commits, hardware clears ECR.NMI_P as part of the same architectural transition. A newly observed NMI may set the latch again. An entry failure does not clear the latch before committing the specified delivery-failure result. When ERET successfully restores STATUS.NI to zero, a pending NMI is delivered before the next instruction boundary.

WRCR to ECR takes ECR.MAX_EDEPTH and ECR.V from the source image and preserves the hardware-managed ECR.NMI_P value. A transition that changes ECR.V from zero to one validates EPC, ECS, EDS, and every configured event-stack segment and top before atomically replacing the writable ECR image.

When maskable-interrupt entry commits, hardware clears exactly the delivered identity's pending bit in the same architectural transition. Entry validation, target probing, frame-address, or frame-store failure preserves that bit; a later posting of the same identity continues to coalesce while it remains pending.

STATUS.EA, STATUS.EDEPTH, STATUS.UO, and STATUS.PM are hardware-managed transition state. WRSTATUS must preserve their current values. An attempted change raises RESERVED_CONTROL_BITS without modifying STATUS. WRSTATUS may change STATUS.IE, STATUS.NI, STATUS.TF, and STATUS.RF. Clearing STATUS.NI while ECR.NMI_P is one admits the pending NMI at the next instruction boundary. ERET classifies direct U-bank, outer U-bank, and stacked returns before any stack access; an invalid combination raises INVALID_CONTROL_TRANSITION without consuming either return source.

14.6 Architectural Event Frame

Every event frame begins with exactly eight 64-bit slots. `EVENT_INFO` contains `EVENT_INFO.EVENT_CODE` in bits 31 through 0; its upper half is zero. `FRAME_CONTROL` contains only frame shape, saved nesting state, saved `FRAME_CONTROL.FLAGS`, and saved `FRAME_CONTROL.STATUS`. Event classification is carried exclusively by `EVENT_INFO`.

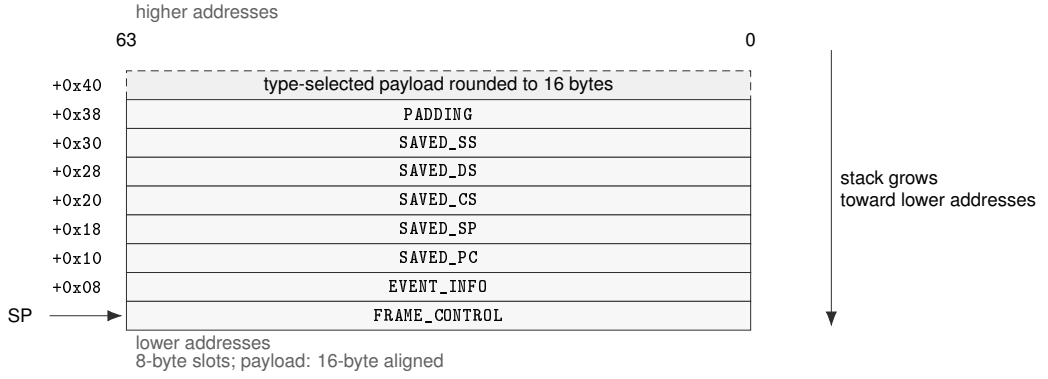


Figure 14-2. Architectural Event Frame

Table 14-4. Architectural Event Frame Fields

Offset	Slot	Meaning
+0x00	`FRAME_CONTROL`	frame shape, nesting metadata, saved FLAGS, and saved STATUS
+0x08	`EVENT_INFO`	`EVENT_CODE` in bits 31..0; bits 63..32 are zero
+0x10	`SAVED_PC`	saved program counter
+0x18	`SAVED_SP`	saved stack pointer
+0x20	`SAVED_CS`	saved code-segment image
+0x28	`SAVED_DS`	saved data-segment image
+0x30	`SAVED_SS`	saved stack-segment image
+0x38	`PADDING`	ignored by ERET; software may write any value
+0x40	`ERROR_CODE`	exception-specific error code; unassigned bits are zero
+0x48	`FAULT_EA`	numeric effective address before segment pre-translation
+0x50	`FAULT_LINEAR`	address after segment pre-translation
+0x58	`EVENT_AUX`	delivery-failure, machine-check, or bus-failure auxiliary information

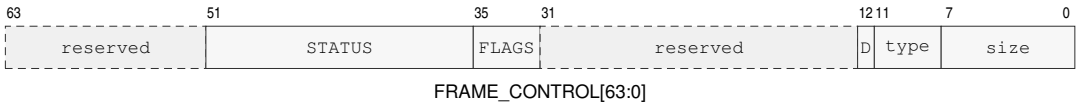


Figure 14-3. FRAME_CONTROL Format

In the diagram, D abbreviates FRAME_CONTROL.SAVED_DFA.

Table 14-5. FRAME_CONTROL Fields

Field	Bits	Meaning
FRAME_SIZE	0..7	total allocated frame size in 8-byte units
FRAME_TYPE	8..11	optional payload layout code; it does not classify the event
SAVED_DFA	12	prior hidden delivery-failure-active state
reserved	13..31	reserved; must be zero
FLAGS	32..35	saved FLAGS image
STATUS	36..51	saved STATUS image, including STATUS.EDEPTH and STATUS.UO
reserved	52..63	reserved; must be zero

ERET first validates the frame shape and nesting metadata. BASIC requires FRAME_CONTROL.FRAME_SIZE=8, ERROR requires 10, and PAGE and AUXILIARY require 12. It requires saved STATUS.PM=1, saved STATUS.EDEPTH plus one to equal the current depth, and saved STATUS.UO to match the active outer-origin chain.

Only after those checks does ERET validate the saved control state, segment images, SP, and PC. On success it restores FRAME_CONTROL.FLAGS, FRAME_CONTROL.STATUS, CS, DS, SS, SP, PC, and current DFA as one commit. It ignores the fixed-header PADDING slot at +0x38 and restores no hidden repeat state. EVENT_INFO and the event-specific payload remain available to software.

14.7 Event Payloads

Payload starts at offset +0x40. Hardware allocates only the defined payload prefix, rounds it up to a 16-byte boundary, and zeros the padding. The frame-size unit remains one 8-byte slot.

Table 14-6. Architectural Event Frame Types and Sizes

Code	Type	Payload	Allocated	Total	Size	Last slot
0x0	BASIC	0	0	64	8	none
0x1	ERROR	8	16	80	10	ERROR_CODE
0x2	PAGE	24	32	96	12	FAULT_LINEAR
0x3	AUXILIARY	32	32	96	12	EVENT_AUX

The delivered leaf event determines the frame type. ERROR_CODE is defined separately by each event that uses it and has zero in every unassigned bit. PAGE frames extend that slot with effective-address and linear-address context. AUXILIARY includes all four named slots and permits EVENT_AUX to describe delivery-failure, machine-check, or bus-failure state.

A misaligned atomic memory operand raises ATOMIC_ALIGNMENT_FAULT and uses the PAGE frame. FAULT_EA identifies the misaligned effective-address operand and FAULT_LINEAR contains its segment-pretranslated starting address. The fault is reported before the atomic instruction performs a memory read, memory write, or architectural result update.

Execution-breakpoint and memory-watchpoint payloads are defined in the debug-event payload format. Their pre-effect timing is part of the Architectural Debug Triggers contract.

14.7.1 Address-Context Payload

Events in the ADDRESS_TRANSLATION and PHYSICAL_ACCESS families use the PAGE frame. The leaf event code identifies the failure; ERROR_CODE carries only the access context common to those events.

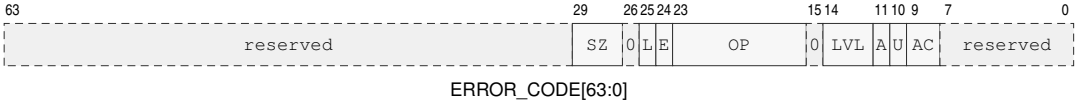


Figure 14-4. PAGE Event ERROR_CODE Format

SZ, L, E, OP, LVL, and AC denote PAGE_ERROR_CODE.ACCESS_SIZE, PAGE_ERROR_CODE.FAULT_LINEAR_VALID, PAGE_ERROR_CODE.FAULT_EA_VALID, PAGE_ERROR_CODE.OPERAND, PAGE_ERROR_CODE.WALK_LEVEL, and PAGE_ERROR_CODE.ACCESS; A and U denote PAGE_ERROR_CODE.ATOMIC and PAGE_ERROR_CODE.USER_DOMAIN.

Table 14-7. PAGE Event ERROR_CODE Fields

Bits	Field	Meaning
7..0	reserved	zero; failure identity is carried by EVENT_INFO.EVENT_CODE
9..8	ACCESS	0 none, 1 read, 2 write, 3 execute
10	USER_DOMAIN	access selected the user-domain permission path
11	ATOMIC	access was an atomic read-modify-write
14..12	WALK_LEVEL	0 before or without a walk; 1 through 4 identify the PTE level
15	reserved	zero
23..16	OPERAND	zero-based explicit operand ordinal; 0xff for implicit fetch or stack access
24	FAULT_EA_VALID	FAULT_EA was produced
25	FAULT_LINEAR_VALID	FAULT_LINEAR was produced
26	reserved	zero
29..27	ACCESS_SIZE	0 unavailable or inapplicable; 1 B; 2 W; 3 L; 4 Q
63..30	reserved	zero

Saved STATUS.PM records the originating privilege. PAGE_ERROR_CODE.USER_DOMAIN distinguishes the ordinary user permission path from the checked user-access path used by supervisor instructions such as MOVUC, MOVCU, and MOVUU. An atomic read-modify-write reports PAGE_ERROR_

CODE.ACCESS=write. Events detected during a page-table walk report the applicable level; events detected before a walk and every event in the PHYSICAL_ACCESS family report level zero.

PAGE_ERROR_CODE.ACCESS_SIZE records the architectural transfer width when one of the B, W, L, or Q widths applies. It is zero when no such width is available or applicable. FAULT_EA is the numeric effective address before segment pre-translation. FAULT_LINEAR is the result after segment pre-translation. A value not produced is zero and has its corresponding validity bit clear.

14.7.2 Other Exception Payloads

Integer-divide, instruction-validation, control-state, and privilege failures are represented by distinct leaf events. Their event code identifies the precise condition, so events without a defined payload use BASIC frames and do not repeat a cause number in ERROR_CODE.

The bounds instructions report comparison failure through FLAGS.V. Segment bounds failures raise SEGMENT_BOUNDS_VIOLATION. Debug-event payloads are defined separately for their distinct event identities.

14.7.3 Debug Event Payloads

SINGLE_STEP and EXPLICIT_BREAKPOINT use BASIC frames and carry no address payload. EXECUTION_BREAKPOINT uses an ERROR frame; ERROR_CODE bits 15 through 0 contain the TRIGGER_SLOT number and all other bits are zero. Its saved PC is the matched address.

The three memory-watchpoint events use PAGE frames. FAULT_EA contains the matched pre-segment effective address and FAULT_LINEAR contains its segment-pretranslated linear address. ERROR_CODE has the following event-specific format.

Figure 14-5. Memory-Watchpoint ERROR_CODE Format

L, M, and U abbreviate MEMORY_WATCHPOINT_ERROR_CODE.LINEAR_VALID, MEMORY_WATCHPOINT_ERROR_CODE.MMIO, and MEMORY_WATCHPOINT_ERROR_CODE.USER_DOMAIN.

Table 14-8. Memory-Watchpoint ERROR_CODE Fields

Bits	Field	Meaning
15..0	TRIGGER_SLOT	zero-based winning trigger-slot number
23..16	OPERAND	zero-based explicit memory-operand ordinal; 0xff for an implicit data access
39..24	ACCESS_SIZE	number of bytes in the matched architectural access
40	USER_DOMAIN	access selected the user permission domain
41	MMIO	translated access was classified as MMIO rather than Normal

Table 14-8 (continued)

Bits	Field	Meaning
42	LINEAR_VALID	FAULT_LINEAR contains the segment-pretranslated address
63..43	reserved	zero

14.7.4 Floating-Point Exception Payload

The FLOATING_POINT_EXCEPTION event uses an ERROR frame because one operation may raise several enabled floating-point conditions together. ERROR_CODE[4:0] is a bitmap: bits 4 through 0 are NV, DZ, OF, UF, and NX; bits 63 through 5 are zero.

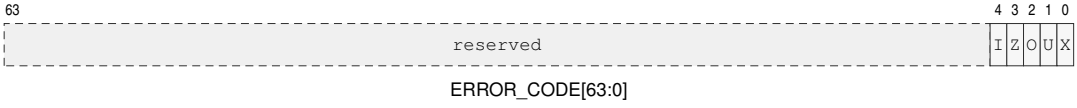


Figure 14-6. FLOATING_POINT_EXCEPTION ERROR_CODE Format

I, Z, O, U, and X abbreviate NV, DZ, OF, UF, and NX.

14.8 Control-Flow Integrity Violation

The CONTROL_FLOW_INTEGRITY_VIOLATION event uses an error frame. ERROR_CODE value 1 identifies an indirect-landing failure and value 2 identifies return-authentication failure. The payload does not expose a candidate continuation, supplied or expected PA, or key-derived value. A failed check commits no stack, resident-state, or control-transfer change.

14.8.1 Auxiliary Payloads

For an AUXILIARY frame, ERROR_CODE bit 26 is the applicable DELIVERY_FAILURE_ERROR_CODE.EVENT_AUX_VALID, MACHINE_CHECK_ERROR_CODE.EVENT_AUX_VALID, or BUS_FAILURE_ERROR_CODE.EVENT_AUX_VALID field. Bits 24 and 25 are the corresponding address-validity fields when the event can supply those addresses. Every unassigned bit is zero, and an auxiliary or address value not produced is zero with its validity bit clear.

Delivery-failure stages are represented by distinct leaf events. Their EVENT_INFO.EVENT_CODE identifies the failed stage; the EVENT_AUX payload records the event whose delivery failed.

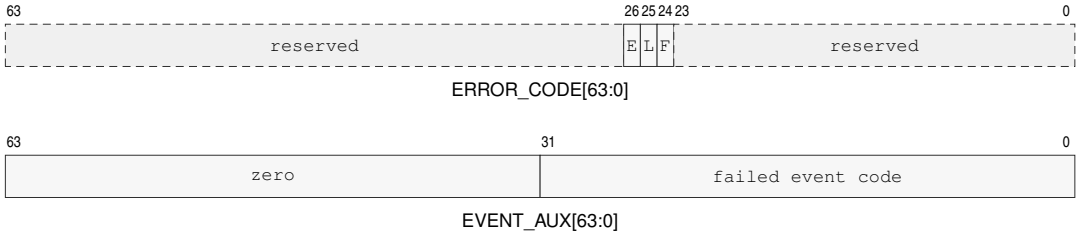


Figure 14-7. Delivery-Failure Auxiliary Format

E, L, and F are EVENT_AUX_VALID, FAULT_LINEAR_VALID, and FAULT_EA_VALID.

When DELIVERY_FAILURE_ERROR_CODE.EVENT_AUX_VALID is one, EVENT_AUX[31:0] contains the event code whose delivery failed and bits 63..32 are zero. A EVENT_FRAME_ADDRESS_FAILURE or EVENT_FRAME_STORE_FAILURE supplies FAULT_EA and FAULT_LINEAR when those values were produced.

For MACHINE_CHECK, the error code has this layout:

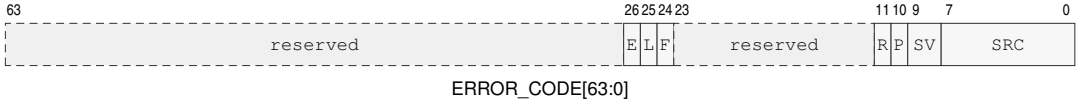


Figure 14-8. MACHINE_CHECK ERROR_CODE Format

E, L, and F are MACHINE_CHECK_ERROR_CODE.EVENT_AUX_VALID, MACHINE_CHECK_ERROR_CODE.FAULT_LINEAR_VALID, and MACHINE_CHECK_ERROR_CODE.FAULT_EA_VALID; P, R, SV, and SRC denote MACHINE_CHECK_ERROR_CODE.PRECISE, MACHINE_CHECK_ERROR_CODE.RETRY_SAFE, MACHINE_CHECK_ERROR_CODE.SEVERITY, and MACHINE_CHECK_ERROR_CODE.SOURCE.

Table 14-9. MACHINE_CHECK ERROR_CODE Fields

Bits	Field	Meaning
7..0	SOURCE	0 unknown; 1 core; 2 cache; 3 translation; 4 memory; 5 interconnect
9..8	SEVERITY	0 corrected; 1 recoverable; 2 fatal; 3 reserved
10	PRECISE	saved PC identifies the responsible instruction and saved architectural state precedes it
11	RETRY_SAFE	retry at that precise boundary cannot duplicate an external effect
23..12	reserved	zero
24	FAULT_EA_VALID	FAULT_EA was produced
25	FAULT_LINEAR_VALID	FAULT_LINEAR was produced
26	EVENT_AUX_VALID	EVENT_AUX was produced
63..27	reserved	zero

When MACHINE_CHECK_ERROR_CODE.EVENT_AUX_VALID is one, EVENT_AUX contains an implementation-defined syndrome. Associated effective and linear addresses are supplied when available.

MACHINE_CHECK_ERROR_CODE.RETRY_SAFE=1 requires MACHINE_CHECK_ERROR_CODE.PRECISE=1. The valid MACHINE_CHECK_ERROR_CODE.SEVERITY and continuation combinations are:

Table 14-10. MACHINE_CHECK Valid Continuation Combinations

MACHINE_CHECK_ERROR_CODE.SEVERITY	MACHINE_CHECK_ERROR_CODE.PRECISE	MACHINE_CHECK_ERROR_CODE.RETRY_SAFE	Saved PC and continuation meaning
corrected	0	0	Trap; saved PC is the instruction following the corrected operation.
recoverable	0	0	Saved PC is the next instruction that would execute at the delivery boundary; saved architectural state is the state at that boundary. It does not identify the responsible operation.
recoverable	1	0	Saved PC identifies the responsible instruction and architectural state precedes it; retry may duplicate an external effect.
recoverable	1	1	Saved PC identifies the responsible instruction and architectural state precedes it; retry cannot duplicate an external effect.
fatal	0	0	Saved PC is the next instruction at the delivery boundary and is diagnostic only.
fatal	1	0	Saved PC identifies the responsible instruction and pre-instruction architectural state for diagnosis only.

All other combinations are invalid and are not generated. In particular, corrected machine checks never set either bit, MACHINE_CHECK_ERROR_CODE.PRECISE=0, MACHINE_CHECK_ERROR_CODE.RETRY_SAFE=1 is invalid for every severity, fatal machine checks never set MACHINE_CHECK_ERROR_CODE.RETRY_SAFE, and MACHINE_CHECK_ERROR_CODE.SEVERITY 3 is reserved. A fatal event provides no reliable continuation even when its diagnostic state is precise.

For an event in the BUS_FAILURE family, the error code has this layout:

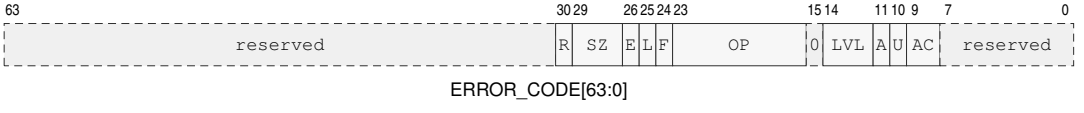


Figure 14-9. BUS_FAILURE ERROR_CODE Format

E, L, and F are BUS_FAILURE_ERROR_CODE.EVENT_AUX_VALID, BUS_FAILURE_ERROR_CODE.FAULT_LINEAR_VALID, and BUS_FAILURE_ERROR_CODE.FAULT_EA_VALID; SZ, OP, LVL, and AC denote BUS_FAILURE_ERROR_CODE.ACCESS_SIZE, BUS_FAILURE_ERROR_CODE.OPERAND, BUS_FAILURE_ERROR_CODE.WALK_LEVEL, and BUS_FAILURE_ERROR_CODE.ACCESS; A, U, and R denote BUS_FAILURE_ERROR_CODE.ATOMIC, BUS_FAILURE_ERROR_CODE.USER_DOMAIN, and BUS_FAILURE_ERROR_CODE.RETRY_SAFE.

Table 14-11. BUS_FAILURE ERROR_CODE Fields

Bits	Field	Meaning
7..0	reserved	zero; failure identity is carried by EVENT_INFO.EVENT_CODE
9..8	ACCESS	0 none, 1 read, 2 write, 3 execute

Table 14-11 (continued)

Bits	Field	Meaning
10	USER_DOMAIN	access selected the user-domain permission path
11	ATOMIC	access was an atomic read-modify-write
14..12	WALK_LEVEL	0 outside a walk; 1 through 4 identify the PTE level
15	reserved	zero
23..16	OPERAND	zero-based explicit operand ordinal; 0xff for implicit fetch or stack access
24	FAULT_EA_VALID	FAULT_EA was produced
25	FAULT_LINEAR_VALID	FAULT_LINEAR was produced
26	EVENT_AUX_VALID	EVENT_AUX was produced
29..27	ACCESS_SIZE	0 unavailable or inapplicable; 1 B; 2 W; 3 L; 4 Q
30	RETRY_SAFE	retry cannot duplicate an external effect
63..31	reserved	zero

A bus error during a page walk reports the level whose PTE access failed. When BUS_FAILURE_ERROR_CODE.EVENT_AUX_VALID is one, EVENT_AUX contains the final byte address available for the failed access.

14.9 Machine-Check and Bus-Error Continuation

A corrected MACHINE_CHECK is a trap, has both MACHINE_CHECK_ERROR_CODE.PRECISE and MACHINE_CHECK_ERROR_CODE.RETRY_SAFE clear, and saves the following instruction. A recoverable machine check uses the valid combinations in the auxiliary-payload table. When MACHINE_CHECK_ERROR_CODE.PRECISE is one, the saved PC identifies the responsible instruction boundary and logical-processor architectural state is the state from before that instruction. When it is zero, the saved PC is the next instruction that would execute at the delivery boundary and architectural state corresponds to that boundary; neither identifies the responsible operation. MACHINE_CHECK_ERROR_CODE.RETRY_SAFE may be one only with MACHINE_CHECK_ERROR_CODE.PRECISE=1 and additionally guarantees that returning to the responsible boundary cannot duplicate an external effect. A fatal machine check always clears MACHINE_CHECK_ERROR_CODE.RETRY_SAFE; MACHINE_CHECK_ERROR_CODE.PRECISE then describes diagnostic location quality and never guarantees reliable continuation.

Every event in the BUS_FAILURE family is a synchronous precise fault. Logical-processor architectural state remains uncommitted and the saved PC identifies the responsible instruction boundary. Its separate BUS_FAILURE_ERROR_CODE.RETRY_SAFE bit states whether an external effect might already have occurred. Page-walk and ordinary memory bus errors are retry-safe.

Software interprets the machine-check continuation fields before choosing a return or recovery action.

14.10 Delivery Failure and Shutdown

Failure to validate the common entry context, establish the selected stack, or store a complete event frame is delivered as the corresponding `EVENT_DELIVERY_FAILURE` leaf event on DSS:DSP. Failure while such an event is already being delivered enters SHUTDOWN. SHUTDOWN retires no further instructions and can be exited only by a platform reset. The frame save and all architectural entry state remain atomic, so a failed attempt never exposes a partially stored frame or partially changed execution context.

14.11 Event Reset and Context State

Warm RESET is supervisor-only and affects only the executing logical processor. Before the reset-state transition commits, all earlier memory effects and pending write-combining stores from that logical processor complete. Other logical processors are unchanged.

The transition clears R0 through R15, SP, and FLAGS. It sets PC to BOOTPC and sets `STATUS.PM` to one while clearing every other STATUS bit. It clears CS, DS, SS, GS0 through GS5, PTCR, ASCR, ECR, EPC, ECS, EDS, the complete user context bank (UPC, USP, UCS, UDS, USS, UCTL, and UINFO), and the configured event stacks and segments (SSS, SSP, ISS, ISP, FSS, FSP, DSS, and DSP). It also clears PMC, CYCLE, INSTRET, and PTWALK. Clearing ECR clears its hardware-managed `ECR.NMI_P` as part of the same transition. It clears all interrupt-file pending and enable bits, ITHRESH, and ISEL while preserving the implementation capability reported by `ICAP.MAX_ID`. It resets DTRSEL and disables every architectural debug-trigger slot. It disarms the deadline timer while preserving the invariant timebase value and frequency. Timer reset does not independently clear interrupt-file state.

Warm RESET invalidates hidden repeat state, the load reservation, local translation and page-walk caches, decoded instruction state, and local prefetch state. It clears the hidden current DFA. Coherent data- and instruction-cache contents remain coherent. Instruction-fetch synchronization completes before the first instruction at BOOTPC is fetched, at which point the processor is running at BOOTPC.

BOOTPC and BOOTCFG are preserved by warm RESET. Cold reset initializes them from platform-supplied values before applying the same processor-state image to each logical processor selected by the platform reset event. Software initializes entry segments, exact entry PCs, and configured stack tops before setting `ECR.V` or entering user mode. WRCR atomically replaces the complete 64-bit BOOTPC or opaque BOOTCFG image without transformation. A later warm RESET uses BOOTPC as its exact resumed PC and preserves both images.

ECR, entry and stack registers, `STATUS.EDEPTH`/`STATUS.UO`, hidden current DFA, the interrupt file, the debug-trigger array, and the user context bank are per-logical-processor state. Event entry preserves the debug-trigger array. System software includes the applicable processor state listed above when switching supervisor contexts and manages the debug-trigger array separately. Payloads and full event frames reside in byte-addressed normal memory owned by the interrupted context.

15 ARCHITECTURAL DEBUG TRIGGERS

The Base debug facility defines stopping events, trigger configuration, and retry behavior for one logical processor. External debugger transport, authentication, forced halt and resume, multicore coordination, and platform policy are outside the processor ISA.

15.1 Debug Event Identities

SINGLE_STEP is the post-success event selected by STATUS.TF. EXPLICIT_BREAKPOINT is the post-success event produced by BKPT. Both save the following architectural boundary.

EXECUTION_BREAKPOINT, MEMORY_READ_WATCHPOINT, MEMORY_WRITE_WATCHPOINT, and ATOMIC_MEMORY_WATCHPOINT are fault-like, pre-effect events. They save the current instruction for retry. The event identity itself states the stopping reason; there is no generic debug-cause event. TRACE records a trace-stream marker and permits execution to continue.

15.2 Trigger-Slot Array and Programming Model

Each logical processor owns an array of at least four trigger slots. The implementation reports the exact count through CPUID SLOT_COUNT. Trigger state is accessed by the supervisor-only DTRSEL/DTRDATA selector-data pair. DTRSEL.SLOT selects a slot and DTRSEL.BANK selects the word read or written through DTRDATA. WRCR to DTRSEL requires bits 63 through 18 to be zero, a slot below SLOT_COUNT, and a bank from zero through two; a successful write atomically replaces the selector.

Table 15-1. DTRSEL Bank Selection

DTRSEL.BANK	Word	Meaning
0	CONFIG	validity, access-kind, and privilege-domain controls
1	LOW	inclusive low effective-address or PC bound
2	HIGH	exclusive high effective-address or PC bound
3	reserved	selection raises INVALID_CONTROL_SELECTOR

Figure 15-1. Debug-Trigger CONFIG Word

V, X, R, W, U, and S abbreviate CONFIG.VALID, CONFIG.MATCH_EXECUTE, CONFIG.MATCH_READ, CONFIG.MATCH_WRITE, CONFIG.MATCH_USER, and CONFIG.MATCH_SUPERVISOR.

Table 15-2. Debug-Trigger CONFIG Fields

Field	Bit	Meaning
VALID	0	slot participates in matching
MATCH_EXECUTE	1	match instruction execution at the architectural PC
MATCH_READ	2	match instruction-owned data reads
MATCH_WRITE	3	match instruction-owned data writes
MATCH_USER	4	match accesses in the user permission domain
MATCH_SUPERVISOR	5	match accesses in the supervisor permission domain
reserved	63..6	reads as zero and must be zero when written

A slot is staged while CONFIG.VALID=0. Software writes CONFIG, LOW, and HIGH in any order, then writes CONFIG.VALID=1 as the final operation. Activation atomically validates that LOW is less than HIGH, at least one of CONFIG.MATCH_EXECUTE/CONFIG.MATCH_READ/CONFIG.MATCH_WRITE is set, at least one of CONFIG.MATCH_USER/CONFIG.MATCH_SUPERVISOR is set, and every reserved bit is zero. To replace an active slot, software first writes its current CONFIG image with only CONFIG.VALID cleared. While active, no other CONFIG change and no LOW or HIGH write is permitted. A successful WRCR to DTRDATA atomically replaces the selected staged word or activates the complete staged slot. A rejected write changes neither the slot nor the selector.

15.3 Matching Coordinates and Selection

Each slot describes one non-wrapping half-open range $[\mathrm{LOW}, \mathrm{HIGH})$ using LOW and HIGH. An execution trigger matches when the pre-segment architectural PC is in that range. A data trigger matches when the byte range of an instruction-owned memory operation overlaps it; the compared coordinate is the pre-segment effective address. An exact one-byte range is expressed by HIGH=LOW+1.

Read and write matching may both be enabled. An atomic read-modify-write qualifies for both bits, but any winning match raises ATOMIC_MEMORY_WATCHPOINT. Trigger matching considers only range, access kind, and user or supervisor permission domain. Data values, masks, and value-change conditions are not part of the Base trigger model. If multiple slots match one operation, the lowest-numbered slot wins.

15.4 Pre-Effect Memory Evaluation

For a candidate memory operation, the processor validates the instruction and operands, resolves its condition or predicate, computes the effective address, and completes segmentation, translation, permission, and alignment validation before evaluating triggers. A winning trigger is reported before the memory or device access and before any architectural result commits. Consequently, earlier synchronous faults take priority; a matching load performs no read, a matching store is not visible, a matching atomic performs no read-modify-write action, and a matching MMIO operation does not access the device. A bus or data fault can still occur after event return when the operation is retried.

Explicit and implicit instruction data accesses participate. Instruction fetch, page-table walks, event-entry frame accesses, prefetch hints, and cache-maintenance accesses do not. False-conditioned operations and inactive vector lanes do not participate. REP and REPcc evaluate before each iteration's memory effect and preserve completed prior iterations. Gather and scatter evaluate before the next active lane access and preserve completed lane and predicate progress.

15.5 Trigger State and Retry

Reset disables every slot and resets DTRSEL. Event entry does not modify the array. The array is per-processor debug state rather than computational context and is not part of any owner state record. `STATUS.TF` and `STATUS.RF` remain execution-context state. `STATUS.RF` suppression, automatic saved-image `STATUS.RF` for pre-effect hardware events, and event priority are specified in Event Priority and Debug Stops.

16 CPUID FEATURE DISCOVERY

16.1 Overview

CPUID is the architectural discovery interface for optional instruction groups, state-record revisions, implementation properties, and feature-specific parameters.

The CPUID instruction uses a selected Rn register as both query selector and result register. Software writes the selector before execution and reads the returned 64-bit value after execution.

CPUID exposes program-visible architectural information.

The normative instruction entry is `CPUID`.

16.2 Query Model

A query selector identifies a discovery class, leaf, and optional result index. The selector is a Q-sized value held in the selected Rn register.

CPUID is unprivileged. For a logical processor, CPUID results remain stable until that logical processor is reset. A discovery result applies to execution on the logical processor that returned it. Software associates cached discovery state with that logical processor and uses results obtained on each logical processor before selecting its optional architectural behavior.

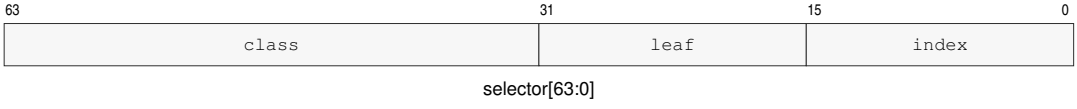


Figure 16-1. CPUID Query Selector Format

Table 16-1. CPUID Query Selector

Bits	Field	Meaning
`0..15`	`index`	result index within the selected leaf
`16..31`	`leaf`	information leaf within the selected class
`32..63`	`class`	CPUID information class

Index zero of every defined leaf is a common discovery header. Leaf-specific payload begins at index one. An unknown class, leaf, or index returns zero.

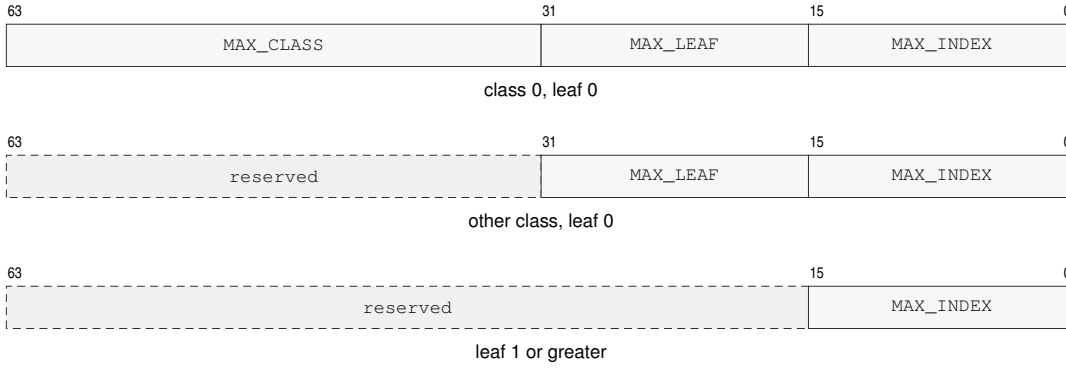


Figure 16-2. CPUID Common Leaf Header Formats

Table 16-2. CPUID Common Leaf Header

Query	Bits 63..32	Bits 31..16	Bits 15..0
class 0, leaf 0, index 0	MAX_CLASS	MAX_LEAF	MAX_INDEX
other class, leaf 0, index 0	reserved, zero	COMMON_HEADER.MAX_LEAF	COMMON_HEADER.MAX_INDEX
any class, leaf 1 or greater, index 0	reserved, zero	reserved, zero	COMMON_HEADER.MAX_INDEX

COMMON_HEADER.MAX_CLASS is the greatest class recognized by the implementation, COMMON_HEADER.MAX_LEAF is the greatest leaf recognized within the selected class, and COMMON_HEADER.MAX_INDEX is the greatest index recognized within the selected leaf. Each field gives the greatest recognized selector value. A leaf may be sparse below COMMON_HEADER.MAX_INDEX. Software ignores reserved bits in every returned value.

CPUID bit fields are feature predicates unless the leaf description says they encode a numeric value. Reserved result bits are ignored by software.

A set feature bit only permits software to use the corresponding architectural behavior; the instruction's normal privilege, operand, and availability checks still apply.

Numeric fields use names that describe their unit. State-record revision fields are unsigned format identifiers, not byte counts.

16.3 Discovery Classes

Discovery classes partition architectural information by subject. Software first checks the class and leaf directory before consuming optional feature bits.

The base class describes the mandatory scalar ISA and implementation identity. Optional groups are reported in the optional-extension directory.

16.4 Leaf Directory

Each discovery class may expose a leaf directory. A directory leaf reports which subordinate leaves are defined and how indexed leaves should be enumerated.

Classes 0x00000000, 0x00000001, and 0x00000002 respectively contain the Bedrock base ISA, optional extensions, and implementation properties.

Table 16-3. CPUID Class and Leaf Directory

Class	Leaf	Name	Indexes	Summary
0x00000000	0x0000	BASE_IDENTITY	0..18	common header, identity, revisions, vendor name, and processor name
0x00000000	0x0001	BASE_INSTRUCTIONS	0..1	common header and optional base-instruction bits
0x00000000	0x0002	BASE_STATE_RECORDS	0..1	common header and base user/supervisor record revisions
0x00000001	0x0000	EXTENSION_DIRECTORY	0..1	common header and optional architectural-group bits
0x00000002	0x0000	IMPLEMENTATION_DIRECTORY	0..0	common implementation-class header
0x00000002	0x0001	CACHE_TOPOLOGY	0..n	common header, maintenance granule, and cache-sharing descriptors
0x00000002	0x0002	PERFORMANCE_COUNTERS	0..3	common header and standard RDPMC-counter presence
0x00000002	0x0003	ADDRESS_WIDTHS	0..1	common header and the implementation physical-address width
0x00000002	0x0006	DEBUG_TRIGGERS	0..1	common header and implemented debug-trigger slot count

16.5 Base Identity

Class 0x00000000, leaf 0x0000 reports MAX_LEAF=0 and MAX_INDEX=18 in its index-zero header. An implementation recognizing only the classes defined by this specification reports MAX_CLASS=2; an implementation that recognizes a higher implementation-defined class reports the highest such selector. The base-identity payload is:

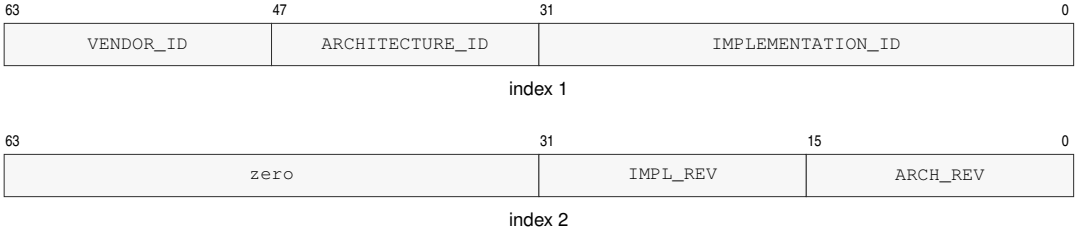


Figure 16-3. CPUID Base Identity Result Formats

Table 16-4. Base Identity Indexes

Index	Contents
1	bits 31..0 IMPLEMENTATION_ID; bits 47..32 ARCHITECTURE_ID; bits 63..48 VENDOR_ID
2	bits 15..0 ARCHITECTURE_REVISION; bits 31..16 IMPLEMENTATION_REVISION; bits 63..32 zero
3–10	64-byte vendor name
11–18	64-byte processor name

VENDOR_ID selects the vendor namespace in which IMPLEMENTATION_ID is interpreted. IMPLEMENTATION_ID is assigned within that VENDOR_ID namespace. ARCHITECTURE_ID identifies the base ISA definition implemented by the processor, while the (VENDOR_ID, IMPLEMENTATION_ID) pair identifies the implementation.

ARCHITECTURE_REVISION is the unsigned 16-bit revision of the base ISA definition selected by ARCHITECTURE_ID; it does not imply the behavior of any other revision. Optional instruction groups, processor state, and feature-specific parameters are reported separately by their CPUID leaves and feature bits. IMPLEMENTATION_REVISION is an unsigned revision assigned by the vendor and is compared only when both VENDOR_ID and IMPLEMENTATION_ID match.

The name fields are UTF-8 byte strings in increasing index order. Within each 64-bit result, the first byte occupies bits 7..0 and subsequent bytes occupy successively higher bits. A name shorter than 64 bytes is terminated by a zero byte and all remaining bytes are zero. A 64-byte name has no terminator. Producers must not split a multi-byte UTF-8 code point at the field boundary. Consumers stop at that boundary and replace any invalid sequence.

16.6 Optional-Extension Directory

Software treats the optional-extension directory as the first availability gate. Before using an optional architectural behavior or consuming one of its parameter leaves, software establishes every extension predicate required by that extension. A result returned by a parameter leaf does not by itself establish extension availability.

Each advertised extension owns exactly one discovery leaf in class 0x00000001. If its availability field is bit m of directory index n, where $1 \leq n \leq 1023$ and $0 \leq m \leq 63$, its discovery leaf is $64(n-1)+m+1$. This relation is one-to-one; an extension with no additional discovery payload still provides the leaf with its index-zero header.

Class 0x00000001, leaf 0x0000 uses the 64-bit query selector 0x0000000100000000 | $index$, where $index$ is the 16-bit query index below.

Table 16-5. Extension Directory CPUID Query-Index Assignments

Index	Query
0x0000	HEADER
0x0001	FEATURES

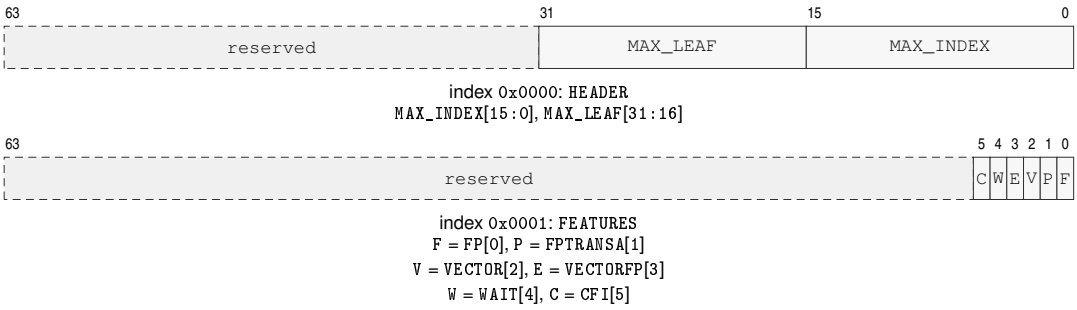


Figure 16-4. Extension Directory CPUID Result Formats

16.7 Floating-Point Extension Discovery

When FP is available, the floating-point registers, state, event, addressing forms, and instructions defined available subject to their ordinary legality checks and any feature-specific predicates below.

When FP16_CONVERT is available, binary16 conversion is available through every form of FCVTH and FCVTUH, the H forms of FCVTS, and the H forms of FCVTD. It does not advertise binary16 arithmetic. The reported value is stable until reset. Executing one of those forms while FP16_CONVERT is clear raises UNAVAILABLE_INSTRUCTION_EXTENSION before operand or floating-point-state access.

USER_RECORD_REVISION reports the 8-bit floating-point user-state-record revision. When FP is available, this field returns 1. Revision 1 identifies the fixed 192-byte record defined by this specification. Revision values are opaque identifiers: software must not infer ordering or compatibility from their numeric values and must reject an unrecognized revision before using the record format.

Class 0x00000001, leaf 0x0001 uses the 64-bit query selector 0x0000000100010000 | $index$, where $index$ is the 16-bit query index below.

Table 16-6. Floating-Point Features CPUID Query-Index Assignments

Index	Query
0x0000	HEADER
0x0001	FEATURES

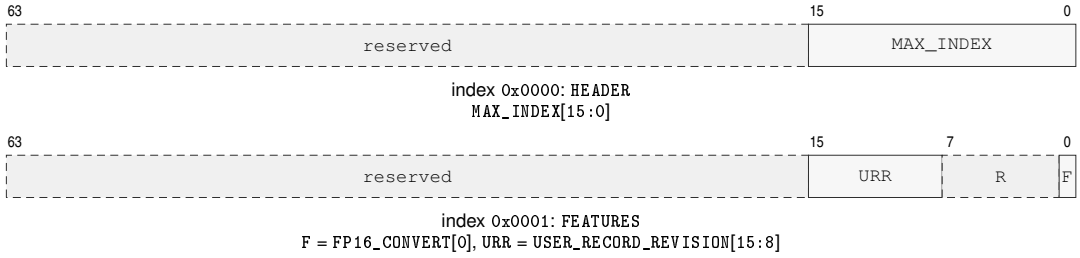


Figure 16-5. Floating-Point Features CPUID Result Formats

16.8 Vector-Parameter Discovery

After establishing VECTOR availability, software uses this leaf to obtain the architectural VLEN reported by VLEN_LOG2_BITS. VLEN is $2^{\texttt{VLEN\_LOG2\_BITS}}$ bits and is an implementation-selected power of two from 128 through 2048 bits. It determines vector-register width, predicate storage, the number of lanes for a selected element width, and the size of each vector register and predicate register image. USER_RECORD_REVISION reports the 8-bit vector user-state-record revision. When VECTOR is available, this field returns 1. Revision 1 identifies the VLEN-dependent record defined by this specification. Revision values are opaque identifiers: software must not infer ordering or compatibility from their numeric values and must reject an unrecognized revision before using the record format. This leaf does not establish availability of VECTORFP or any other extension that consumes vector state.

Class 0x00000001, leaf 0x0003 uses the 64-bit query selector $\texttt{0x0000000100030000} \mid index$, where $index$ is the 16-bit query index below.

Table 16-7. Vector Parameters CPUID Query-Index Assignments

Index	Query
0x0000	HEADER
0x0001	PARAMETERS

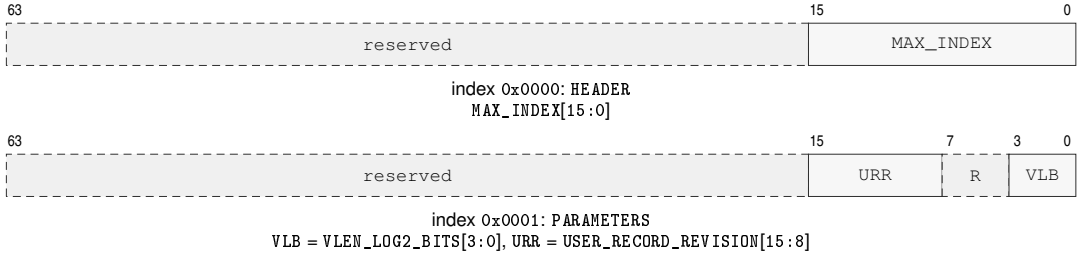


Figure 16-6. Vector Parameters CPUID Result Formats

16.9 Vector Floating-Point Discovery

Software establishes VECTORFP, VECTOR, and FP through the Optional-Extension Directory before using this extension. This extension defines no additional discovery payload, so its discovery leaf contains only the common header.

Class `0x00000001`, leaf `0x0004` uses the 64-bit query selector `0x0000000100040000` | $index$, where $index$ is the 16-bit query index below.

Table 16-8. Vector Floating-Point Discovery CPUID Query-Index Assignments

Index	Query
`0x0000`	`HEADER`

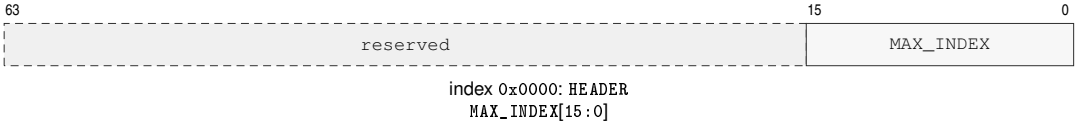


Figure 16-7. Vector Floating-Point Discovery CPUID Result Formats

16.10 FPTRANSA Accuracy Discovery

The `FPTRANSA` predicate requires `FP` and reports that at least one approximate transcendental contract may be available. Software determines each operation's availability separately from its accuracy-contract result. Class `0x00000001`, leaf `0x0002` uses indexes 1 through `0x0044` as a sparse space of stable 16-bit accuracy-contract IDs. Index zero is the common header and reports `MAX_INDEX=0x0044`. A contract ID identifies the reference function, input domain, special-value behavior, error metric, exact anchors, and mathematical properties; it is independent of instruction encoding placement. A different tuple of those properties uses a different ID, and a retired ID is not reused. The contracts defined here use `CONTRACT_REVISION=1`. A changed guarantee, domain, or error metric names a different contract ID. An implementation that advertises a smaller certified ULP bound strengthens the existing contract and retains the same ID and revision.

The sparse ID space reserves `0x0001..0x000F` for reduced-domain circular trigonometric functions, `0x0010..0x001F` for inverse trigonometric functions, `0x0020..0x002F` for hyperbolic and inverse hyperbolic functions, `0x0030..0x003F` for exponential functions, and `0x0040..0x004F` for logarithmic functions. The low-nibble-zero selector of each family is unassigned, and actual contracts in each family begin at low nibble one.

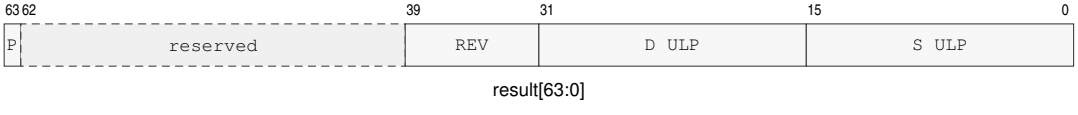


Figure 16-8. FPTRANSA Accuracy Result Format

`P` is `PRESENT`; `REV` is `CONTRACT_REVISION`; the D and S ULP fields use unsigned Q8.8.

Table 16-9. FPTRANSA Accuracy Result

Bits	Field	Meaning
0..15	S_MAX_ULP_Q8_8	certified worst-case S-format ULP bound, rounded upward to unsigned Q8.8
16..31	D_MAX_ULP_Q8_8	certified worst-case D-format ULP bound, rounded upward to unsigned Q8.8
32..39	CONTRACT_REVISION	value 1 for the contracts defined here
40..62	reserved	zero
63	PRESENT	the instruction and both S and D encodings are available

For a certified bound K, the field value is $\lceil 256K \rceil$ and the advertised bound is the field divided by 256. Thus 0.5, 1, 1.5, 2, and 4 ULP encode as 0x0080, 0x0100, 0x0180, 0x0200, and 0x0400. A present contract has both fields in 0x0001..0x0400. An unrecognized returned revision does not satisfy the contracts defined here.

Software first checks FP and FPTRANSA in EXTENSION_DIRECTORY, then queries the intended contract ID in FPTRANSA_ACCURACY. It uses the instruction only when PRESENT is set, the returned CONTRACT_REVISION is 1, and the S or D bound required by the call site is no larger than its quality threshold; otherwise it does not use the instruction under that accuracy contract. For example, a D-format FLOG2A path requiring at most 2 ULP accepts selector 0x0000000100020043 only when the returned D field is at most 0x0200.

The selector is $(\texttt{0x00000001} \ll 32) \,|\, (\texttt{0x0002} \ll 16) \,|$ *contract-id*. For example, FSINA uses selector 0x0000000100020001, and FLOG2A uses 0x0000000100020043. An implementation that certifies FSINA at 1.5 ULP for S and 2 ULP for D returns 0x8000000102000180.

PRESENT=1 establishes the accuracy contract for both S and D encodings but does not waive the instruction's ordinary legality checks. The S and D bounds are worst-case bounds over every input in the contract domain and apply to every execution on the logical processor that returned the CPUID result until reset. Executing the instruction with PRESENT=0 raises INVALID_OPERAND_RELATION before operand or floating-point-state access.

Table 16-10. FPTRANSA Accuracy Contracts

Contract ID	Mnemonic	Reference	Domain	ISA maximum
0x0001	FSINA	sin(x) in radians	finite x with abs(x) $\leq$ pi / 4 after DAZ	S $\leq$ 4 ULP; D $\leq$ 4 ULP
0x0002	FCOSA	cos(x) in radians	finite x with abs(x) $\leq$ pi / 4 after DAZ	S $\leq$ 4 ULP; D $\leq$ 4 ULP
0x0003	FTANA	tan(x) in radians	finite x with abs(x) $\leq$ pi / 4 after DAZ	S $\leq$ 4 ULP; D $\leq$ 4 ULP
0x0004	FSINCOSA	paired sin(x) and cos(x) in radians	finite x with abs(x) $\leq$ pi / 4 after DAZ	S $\leq$ 4 ULP; D $\leq$ 4 ULP
0x0011	FASINA	asin(x) in radians	finite x with abs(x) $\leq$ 1 after DAZ	S $\leq$ 4 ULP; D $\leq$ 4 ULP
0x0012	FACOSA	acos(x) in radians	finite x with abs(x) $\leq$ 1 after DAZ	S $\leq$ 4 ULP; D $\leq$ 4 ULP
0x0013	FATANA	atan(x) in radians	all finite values and signed infinity after DAZ	S $\leq$ 4 ULP; D $\leq$ 4 ULP
0x0021	FSINHA	sinh(x)	all finite values and signed infinity after DAZ	S $\leq$ 4 ULP; D $\leq$ 4 ULP

Table 16-10 (continued)

Contract ID	Mnemonic	Reference	Domain	ISA maximum
0x0022	FCOSHA	cosh(x)	all finite values and signed infinity after DAZ	S $\leq$ 4 ULP; D $\leq$ 4 ULP
0x0023	FTANHA	tanh(x)	all finite values and signed infinity after DAZ	S $\leq$ 4 ULP; D $\leq$ 4 ULP
0x0024	FATANHA	atanh(x)	finite x with abs(x) $\leq$ 1 after DAZ	S $\leq$ 4 ULP; D $\leq$ 4 ULP
0x0031	FETOXA	e^x	all finite values and signed infinity after DAZ	S $\leq$ 4 ULP; D $\leq$ 4 ULP
0x0032	FETOXM1A	$e^x - 1$ without an intermediate rounded exponential	all finite values and signed infinity after DAZ	S $\leq$ 4 ULP; D $\leq$ 4 ULP
0x0033	FTWOTOXA	2^x	all finite values and signed infinity after DAZ	S $\leq$ 4 ULP; D $\leq$ 4 ULP
0x0034	FTENTOXA	10^x	all finite values and signed infinity after DAZ	S $\leq$ 4 ULP; D $\leq$ 4 ULP
0x0041	FLOGNA	ln(x)	positive finite values and positive infinity after DAZ; signed zero is the DZ boundary	S $\leq$ 4 ULP; D $\leq$ 4 ULP
0x0042	FLOGNP1A	ln(1 + x) without an intermediate rounded addition	finite x ≥ -1 and positive infinity after DAZ; -1 is the DZ boundary	S $\leq$ 4 ULP; D $\leq$ 4 ULP
0x0043	FLOG2A	log2(x)	positive finite values and positive infinity after DAZ; signed zero is the DZ boundary	S $\leq$ 4 ULP; D $\leq$ 4 ULP
0x0044	FLOG10A	log10(x)	positive finite values and positive infinity after DAZ; signed zero is the DZ boundary	S $\leq$ 4 ULP; D $\leq$ 4 ULP

16.11 WAIT Extension Discovery

Software establishes the WAIT extension predicate before executing WAIT or WAKE. The extension has no implementation-varying architectural parameters.

Class 0x00000001, leaf 0x0005 uses the 64-bit query selector 0x0000000100050000 | $index$, where $index$ is the 16-bit query index below.

Table 16-11. Address Wait Discovery CPUID Query-Index Assignments

Index	Query
0x0000	HEADER

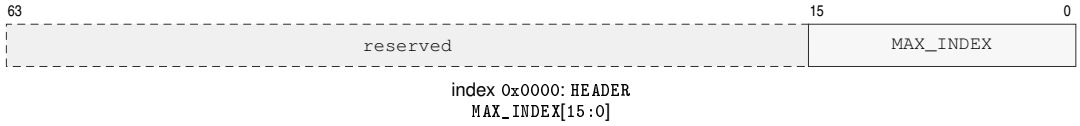


Figure 16-9. Address Wait Discovery CPUID Result Formats

16.12 CFI Extension Discovery

Software establishes the CFI extension predicate before using CFILAND or another CFI instruction. SUPERVISOR_RECORD_REVISION reports the 8-bit CFI supervisor-state-record revision. When CFI is available, this field returns 1. Revision 1 identifies the fixed 16-byte key record. Revision values are opaque identifiers and software must reject an unrecognized revision before using the record format.

Class 0x00000001, leaf 0x0006 uses the 64-bit query selector 0x0000000100060000 | $index$, where $index$ is the 16-bit query index below.

Table 16-12. Control-Flow Integrity Discovery CPUID Query-Index Assignments

Index	Query
0x0000	HEADER
0x0001	STATE_RECORDS

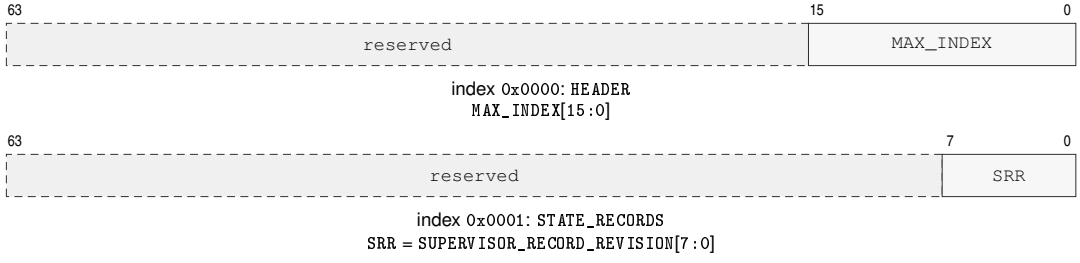


Figure 16-10. Control-Flow Integrity Discovery CPUID Result Formats

16.13 Implementation-Class Directory

Class 0x00000002, leaf 0x0000 uses the 64-bit query selector 0x0000000200000000 | $index$, where $index$ is the 16-bit query index below.

Table 16-13. Implementation Directory CPUID Query-Index Assignments

Index	Query
0x0000	HEADER

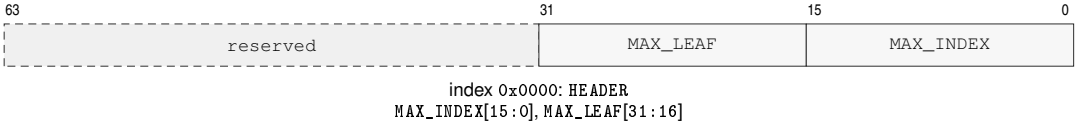


Figure 16-11. Implementation Directory CPUID Result Formats

16.14 Cache-Topology Discovery

Class 0x00000002, leaf 0x0001 reports the cache-maintenance granule and the cache instances visible to the current logical processor. Index zero is the common header. Index one is the maintenance-properties result, and indexes 2 through MAX_INDEX are a dense array of cache descriptors. With N descriptors, MAX_INDEX=1+N.

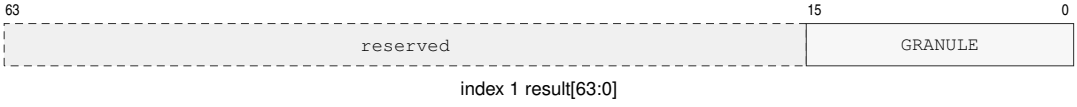


Figure 16-12. Cache-Maintenance Properties Result

GRANULE is MAINTENANCE_GRANULE_BYTES.

Table 16-14. Cache-Maintenance Properties

Index	Bits	Field	Meaning
1	15..0	MAINTENANCE_GRANULE_BYTES	one-block cache-maintenance granule in bytes
1	63..16	reserved	zero

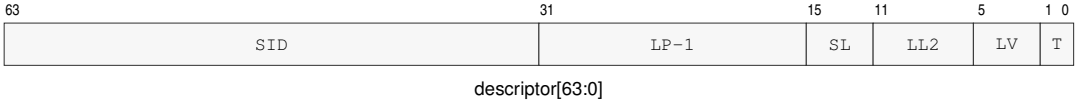


Figure 16-13. Cache-Topology Descriptor Format

SID, LP-1, SL, LL2, LV, and T denote SHARING_ID, SHARING_LP_COUNT_MINUS1, SHARING_LEVEL, LINE_LOG2, LEVEL, and TYPE.

Table 16-15. Cache-Topology Descriptor

Bits	Field	Meaning
1..0	TYPE	1 data; 2 instruction; 3 unified
5..2	LEVEL	cache level, in the range 1 through 15
11..6	LINE_LOG2	cache-line bytes are 1 shifted left by this value
15..12	SHARING_LEVEL	nested cache-sharing scope; zero is one logical processor
31..16	SHARING_LP_COUNT_MINUS1	logical-processor count in the sharing scope, minus one
63..32	SHARING_ID	identifier of the processor set at the reported sharing level

MAINTENANCE_GRANULE_BYTES is a nonzero power of two no greater than 4096 and has the same value on every logical processor in one normal-memory coherence domain. Every reported cache-line size divides the maintenance granule. A descriptor has TYPE 1, 2, or 3 and a nonzero LEVEL. Descriptors are ordered by increasing cache level, cache type, sharing level, and sharing ID.

SHARING_LEVEL values describe strictly nested processor sets. The pair (SHARING_LEVEL, SHARING_ID) is equal exactly when two descriptors name the same logical-processor set. At sharing level zero, SHARING_LP_COUNT_MINUS1 is zero. At a greater sharing level, the count field equals the number of logical processors in that set minus one. The tuple of cache type, cache level, sharing level, and sharing ID is unique within the leaf.

16.15 Performance-Counter Discovery

This leaf determines whether an RDPMC counter ID is available on the current logical processor. Counter IDs form a sparse space: the header bounds the selectors that software may examine but does not imply that every intervening ID is present. Software checks the selected ID before issuing RDPMC.

The standard counter events are defined by RDPMC. An implementation-defined counter additionally requires implementation documentation that names the event counted by its ID.

Class 0x00000002, leaf 0x0002 uses the 64-bit query selector 0x0000000200020000 | $index$, where $index$ is the 16-bit query index below.

Table 16-16. Performance Counters CPUID Query-Index Assignments

Index	Query
0x0000	HEADER
0x0001	CYCLE
0x0002	INSTRET
0x0003	PTWALK
0x0004-0xFFFF	IMPLEMENTATION_DEFINED

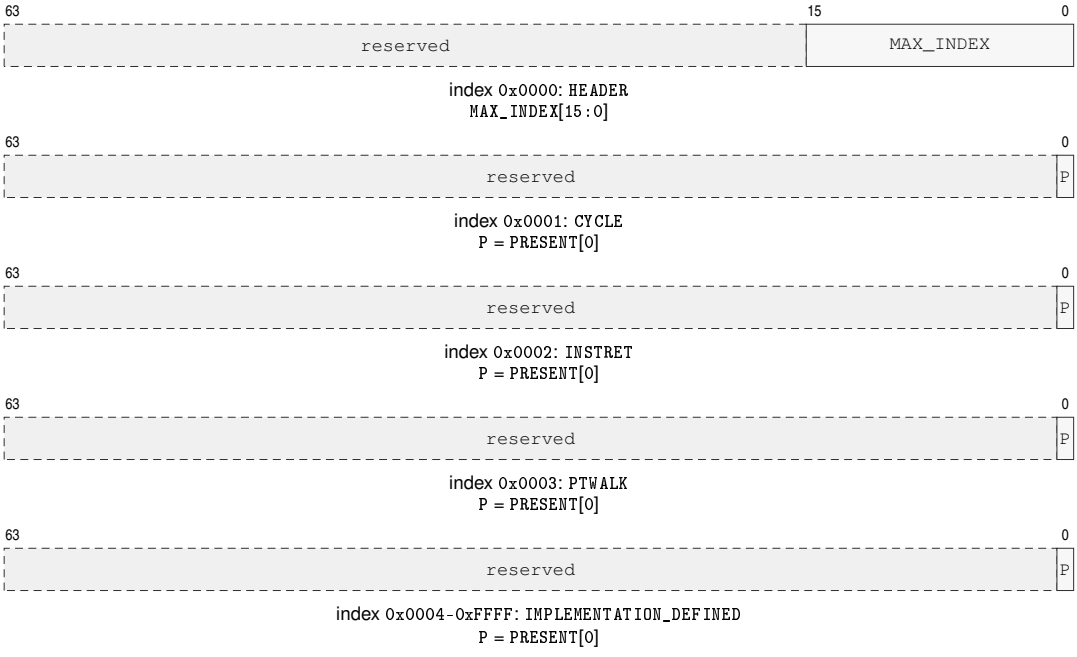


Figure 16-14. Performance Counters CPUID Result Formats

16.16 Address-Width Discovery

`PABITS` is the authoritative physical-address bound for the current logical processor and has a value from 32 through 56. A physical address is representable only when it is less than $2^{\texttt{PABITS}}$. Page-table roots, every present leaf `PTE.PFN`, every present table-pointer `PTE.NEXT_TABLE`, translated results, and paging-disabled physical addresses are subject to that bound.

`PABITS` does not select a translation-table format or translation-cache identity. `PTCR.TT` selects the complete translation-table format, while the translation-cache contract separately defines entry identity.

Class `0x00000002`, leaf `0x0003` uses the 64-bit query selector `0x0000000200030000` | $index$, where $index$ is the 16-bit query index below.

Table 16-17. Address Widths CPUID Query-Index Assignments

Index	Query
`0x0000`	`HEADER`
`0x0001`	`PARAMETERS`

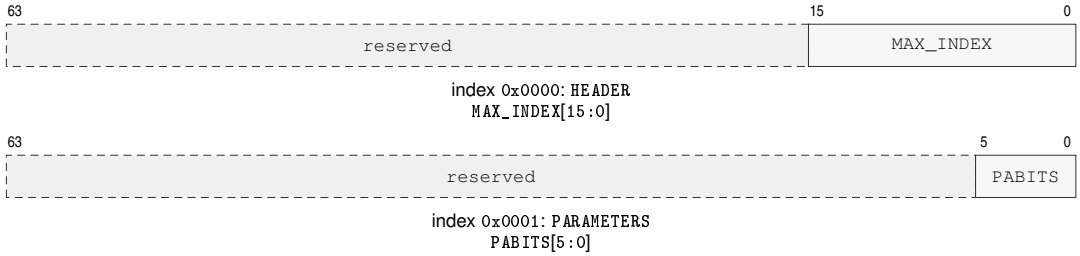


Figure 16-15. Address Widths CPUID Result Formats

16.17 Base State-Record Revisions

The BASE_STATE_RECORDS leaf reports independent 8-bit format revisions for the base user and base supervisor records. This specification provides revision 1 for both records, so both fields return 1. Revision 1 identifies the fixed layouts defined by this specification.

State-record revision values are opaque identifiers: software must not infer ordering or compatibility from their numeric values. Software must reject an unrecognized revision before using the corresponding record format.

Record sizes and alignments are architectural properties of the reported revision and are not repeated in CPUID.

Class 0x00000000, leaf 0x0002 uses the 64-bit query selector $\texttt{0x0000000000020000} \mid index$, where $index$ is the 16-bit query index below.

Table 16-18. Base State Records CPUID Query-Index Assignments

Index	Query
0x0000	HEADER
0x0001	REVISIONS

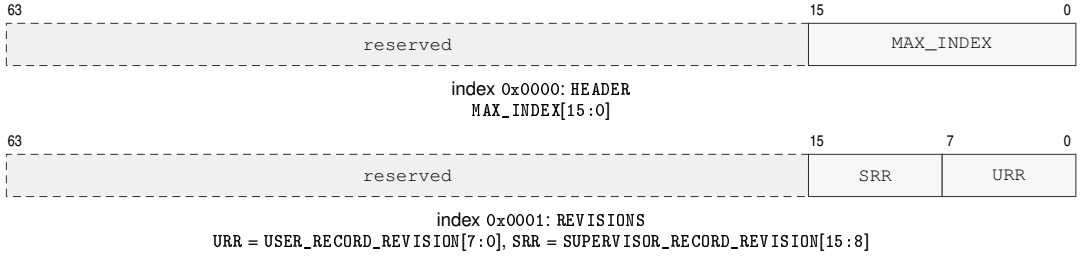


Figure 16-16. Base State Records CPUID Result Formats

16.18 Timebase Discovery

The mandatory architectural timebase advances at the exact nonzero integer rate reported by TICKS_PER_SECOND. The value is invariant for one cold-reset epoch and is preserved by warm

RESET. Software uses this rate to convert differences between RDTIME results and absolute deadline values into elapsed time.

Class 0x00000002, leaf 0x0005 uses the 64-bit query selector 0x0000000200050000 | $index$, where $index$ is the 16-bit query index below.

Table 16-19. Architectural Timebase CPUID Query-Index Assignments

Index	Query
0x0000	HEADER
0x0001	PARAMETERS

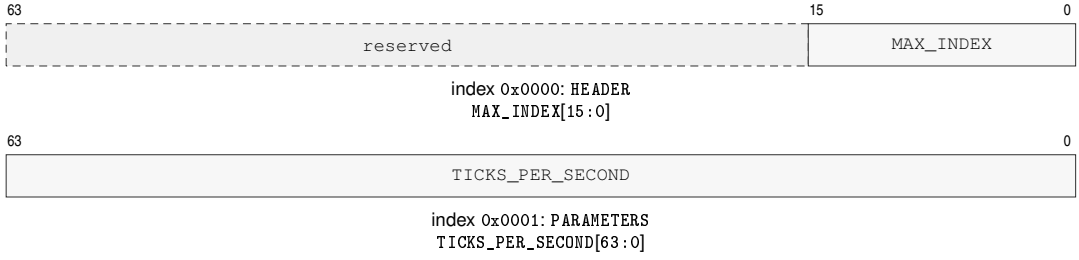


Figure 16-17. Architectural Timebase CPUID Result Formats

16.19 Debug-Trigger Discovery

The mandatory Base trigger array has at least four slots per logical processor. SLOT_COUNT reports the exact number accepted by DTRSEL.SLOT; software does not probe selector values beyond that count.

Class 0x00000002, leaf 0x0006 uses the 64-bit query selector 0x0000000200060000 | $index$, where $index$ is the 16-bit query index below.

Table 16-20. Architectural Debug Triggers CPUID Query-Index Assignments

Index	Query
0x0000	HEADER
0x0001	PARAMETERS

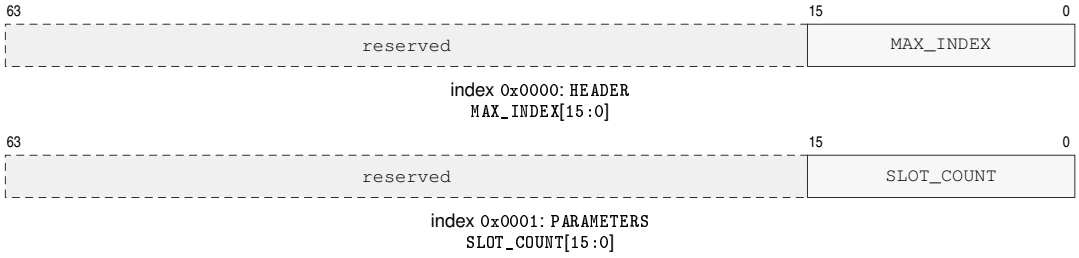


Figure 16-18. Architectural Debug Triggers CPUID Result Formats

17 OWNER-SPECIFIC STATE RECORDS

State-transfer instructions operate on one architecturally owned record at a time. SAVE and RESTORE own the base user record; SSAVE and SRESTORE own the base supervisor record; FP owns its floating-point user record; and VECTOR owns its vector user record. FPTRANSA, VECTORFP, and WAIT define no state-record instruction.

Each instruction has exactly one address form, `[Rn]`. It captures Rn before executing, uses DS as the default data segment, and accepts no displacement, direct, update, or generic effective-address form. Each instruction accesses only its owning state. Software that composes several owner records chooses their placement and synchronization; the architecture provides no cross-owner atomicity.

17.1 Base User-State Record

This record is 200 bytes and 8-byte aligned.

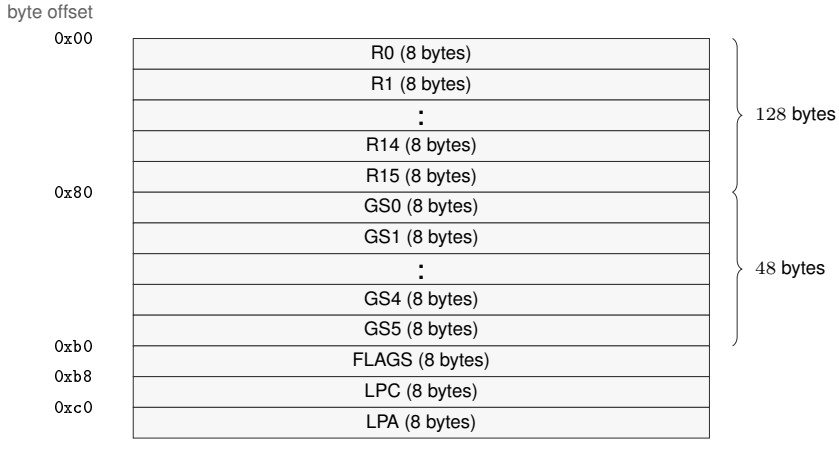


Figure 17-1. Base User-State Record

Figure 17-2. Base User-State Record Bit Layouts

SAVE writes the complete record. RESTORE validates all six segment-register images and the FLAGS image before atomically replacing R0 through R15, GS0 through GS5, FLAGS, LPC, and the complete opaque LPA image. The record is 200 bytes with 8-byte alignment. No field is optional and no base supervisor or extension-owned key state appears in this record.

17.2 Base Supervisor-State Record

This record is 8 bytes and 8-byte aligned.

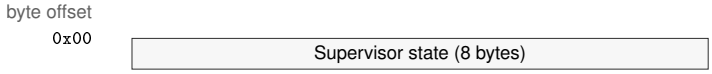


Figure 17-3. Base Supervisor-State Record

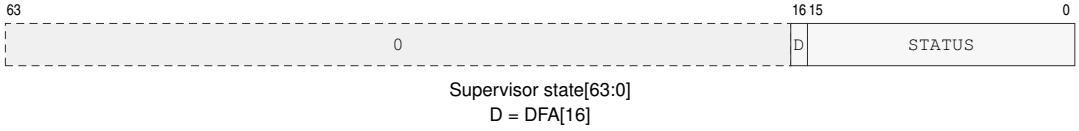


Figure 17-4. Base Supervisor-State Record Bit Layouts

SRESTORE requires the reserved bits to be zero and validates the complete event-state relation before changing state. The saved STATUS must select privileged mode. DFA requires an active event; inactive state requires EDEPTH, UO, and DFA to be zero; and UO requires a valid live U bank. SRESTORE atomically replaces STATUS and DFA only after these checks succeed.

17.3 Floating-Point User-State Record

This record is 192 bytes and 64-byte aligned.

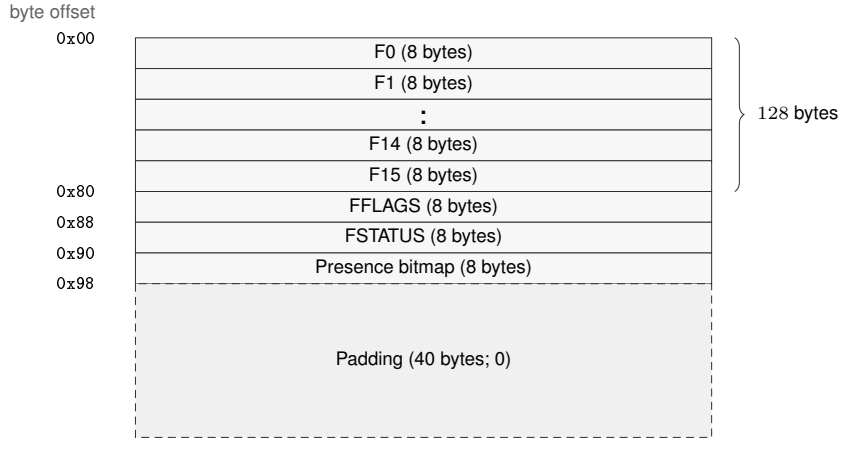


Figure 17-5. Floating-Point User-State Record

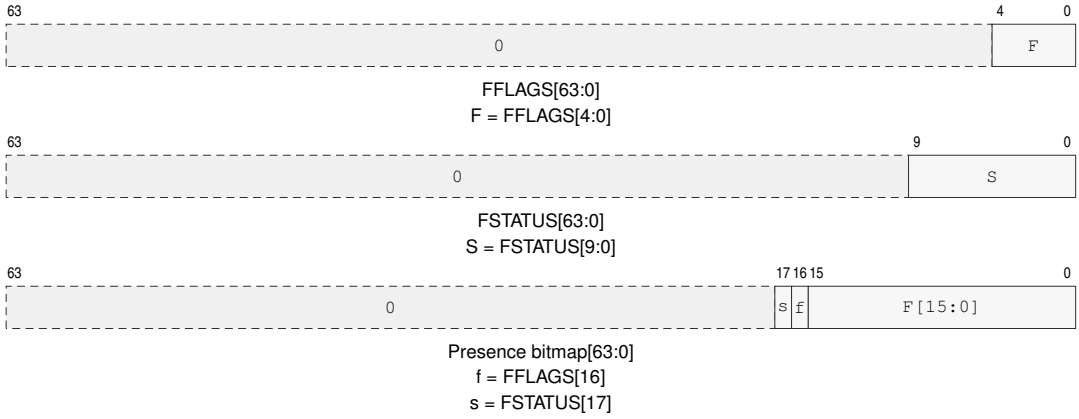


Figure 17-6. Floating-Point User-State Record Bit Layouts

Its initial state is positive zero in F0 through F15 and zero in FFLAGS and FSTATUS.

For FRESTORE, a clear presence bit installs the selected slot's initial value and its payload bytes are not read. All selected images and reserved fields are validated before any floating-point state is committed. An all-clear bitmap makes floating-point state clean; any set bit makes it modified. FSAVE does not change this tracking state and may conservatively set presence bits.

17.4 Vector User-State Record

Let B be VLEN in bytes, with $B \in \{16, 32, 64, 128, 256\}$. This record is 64-byte aligned and has total size $\texttt{0x40} + 64\lceil 34B/64 \rceil$ bytes.

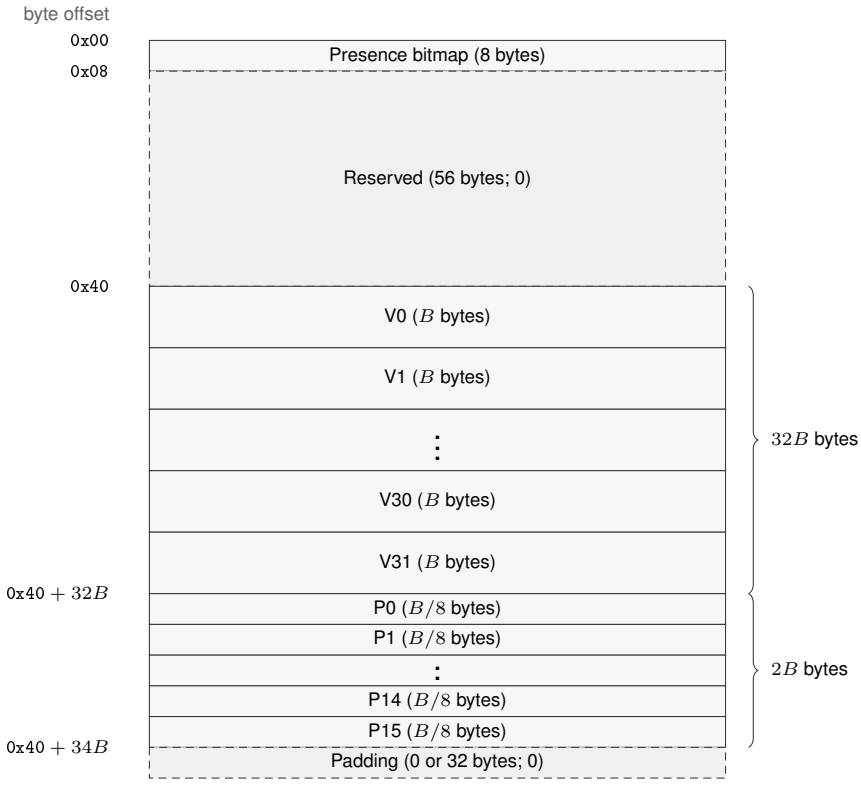


Figure 17-7. Vector User-State Record

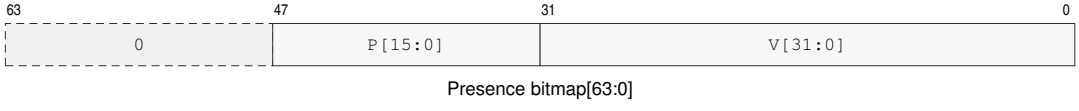


Figure 17-8. Vector User-State Record Bit Layouts

Its initial state is all zero.

For VRESTORE, a clear presence bit installs the selected slot's initial value and its payload bytes are not read. The complete header and all selected images are validated before any vector state is committed. An all-clear bitmap makes vector state clean; any set bit makes it modified. VSAVE does not change this tracking state and may conservatively set presence bits.

17.5 Control-Flow Integrity Supervisor-State Record

This record is 16 bytes and 8-byte aligned.

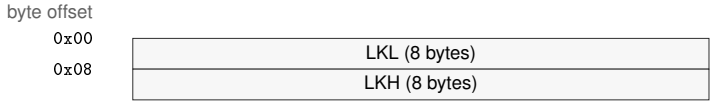


Figure 17-9. Control-Flow Integrity Supervisor-State Record

CFISSAVE writes this complete record without changing key state. CFISRESTORE validates the complete memory access and then replaces LKL and LKH together. Both instructions are supervisor-only, use the ordinary DS addressing context, and leave LPC and LPA unchanged. The all-zero image is valid and selects the unconfigured CFI state.

17.6 State-Record Transfer Semantics

Every SAVE-class instruction is a pure snapshot: it does not change the captured owner state or its clean/modified tracking. The complete destination range is validated before a record is written. Every RESTORE-class instruction validates the complete source range and all record invariants before committing any owner state. An address, access, or validation fault leaves architectural state unchanged.

FP and VECTOR records use owner-local presence bitmaps. A clear slot means initial state, not preserved state, and the corresponding payload bytes are not read. An all-clear bitmap installs the complete initial owner state and makes that owner state clean. Any set bit makes the restored owner state modified, even when the selected payload happens to encode an initial value. SAVE may conservatively set presence bits, but it may clear a bit only for state known to be initial.

State-record instructions do not imply acquire, release, fence, or cross-owner ordering. The normative instruction entries are `SAVE`, `RESTORE`, `SSAVE`, `SRESTORE`, `FSAVE`, `FRESTORE`, `VSAVE`, and `VRESTORE`.

PART IV

BEDROCK ARCHITECTURE

Instruction Set Reference

READING AN INSTRUCTION DESCRIPTION

Entry Organization

Each instruction entry presents its Operation and Assembler Syntax before applicability, repeat, event, and status information relevant to that instruction. Detailed Semantics supplies the conditions and effects needed to complete the Operation. Illustrative diagrams follow that explanation and precede the concrete encodings. Each mnemonic begins on a new page so its encoding diagrams, field explanations, and side-effect text stay together as one reference unit.

Assembler Source Grammar

The assembler source language is a sequence of newline-terminated statements. Whitespace may separate tokens except where an instruction spelling joins a mnemonic, size or condition suffix, or modifier. A label may precede a statement on the same line, and a line may contain labels without an instruction or directive.

```
source              ::= { line }
line                ::= { label ":" } [ statement ] end-of-line
statement           ::= instruction | directive
instruction         ::= instruction-form [ ":" "LEN" integer
                        [ "," padding-byte-list ] ]
instruction-form    ::= mnemonic-form [ operand-list ]
operand-list        ::= operand { "," operand }
padding-byte-list   ::= padding-byte { "," padding-byte }
padding-byte        ::= integer | "ILLEGAL" | "NOP"
operand             ::= register-name
                      | expression
                      | address-expression
                      | register-mask
                      | "(" instruction ")"
address-expression  ::= "[" ea-expression "]"
symbol-reference    ::= bare-identifier | quoted-identifier
quoted-identifier   ::= "‘" { quoted-character | escape } "‘"
escape              ::= "\‘" | "\\"
```

This grammar is an envelope, not a second declaration of the instruction set. The instruction encoding catalog is the sole definition of the accepted mnemonic forms, suffixes, modifiers, operand counts, operand ordering, and operand classes. A parenthesized instruction is accepted only where an instruction form declares an instruction operand. The effective-address mode catalog is the sole definition of the accepted `ea-expression` forms. The brackets delimit address expressions only and never enclose another operand class. Whether a consuming instruction reads memory, writes memory, computes the address only, or applies another address-directed operation is defined by that instruction's operand role.

The optional `: LEN n` annotation requests an encoded extended instruction length of `n` bytes. If no padding-byte list follows, the assembler fills every trailing padding byte with `ILLEGAL`, whose canonical one-byte encoding is zero. If a list is present, each item specifies one trailing padding byte in instruction-stream order; its item count must equal the requested length minus the required instruction length. `NOP` denotes its canonical one-byte encoding, and an integer padding byte must be an absolute value from 0 through 255. Padding bytes do not accept expressions or relocations.

Register names are recognized case-insensitively. Symbol names are case-sensitive. A bare identifier that is a register name denotes that register in a context that accepts it; every other bare identifier in an expression denotes a symbol. Backtick quoting may be applied to any symbol reference and forces the enclosed name to be interpreted as a symbol, so `R1` denotes the register while ‘`R1`‘ denotes the distinct symbol named `R1`. Within a quoted identifier, `\`` denotes a literal backtick and `\\` denotes a literal backslash. No other escape is defined, and a quoted identifier cannot cross a newline.

A relocation annotation follows the complete symbol reference and is not part of the quoted name. For example, ‘`R1`‘`@abs64` applies `abs64` to the symbol `R1`, whereas ‘`R1@abs64`‘ names a symbol whose name contains the characters `@abs64`. An assembler emits an unquoted symbol when its name is a valid bare identifier that does not collide with a register name; otherwise it emits the canonical backtick-quoted form.

Mnemonics and Suffixes

The B, W, L, and Q suffixes select 8-, 16-, 32-, and 64-bit integer sizes.

When an instruction encoding includes a size selector z, its definition maps z to the suffix set shown for that encoding. A mnemonic without a size suffix has a single architecturally defined size or no data-size operand. Conditional mnemonic variants use the condition names listed in the Flags and Condition Codes chapter.

CMPJcc, DJcc, IJcc, Jcc, MOVcc, SETcc, and TESTJcc place the condition suffix in the mnemonic. An encoding with only one architectural size omits the size suffix.

Operand and Status Notation

Rn names an Rn register selected from the general-purpose registers R0 through R15; SP and PC are special registers outside that encoding namespace. An `<ea>` operand names a compact effective-address field and any appended EA payload.

FLAGS and STATUS are architectural state registers. Instruction descriptions state whether an encoding updates, preserves, reads, or ignores each status register. A FLAGS update always names and replaces the complete ZNCV image.

In a transposed flag-effects table, `-` marks an unchanged flag and `0` or `1` marks a constant result. `*` marks the single table-defined effect described by that table's local legend. When a table contains multiple table-defined effects, letters `a` through `z` distinguish their local legend entries.

In instruction pseudocode, Operand Read applies the addressing mode and operation size. Operand Write stores the selected-size result; an Rn write defines all 64 bits and extends a sub-Q result according to the common integer result-extension rule, while a memory write changes only the selected bytes. EA Address computes an address without reading memory. A control-target memory operand reads its declared-width target, while a control-target register or immediate operand supplies the target value directly. A selector may name an Rn register, a segment register, or an encoded immediate. FLAGS ZNCV refers to the integer `FLAGS.Z`, `FLAGS.N`, `FLAGS.C`, and `FLAGS.V` bits.

Encodings

In an encoding diagram, fixed 0 and 1 fields identify framing, opcode-selection, or reserved values, and the notes below decode lettered fields. An <ea> field is the compact effective-address field and may select one or more independently determined EA payloads in operand order.

In each instruction entry, Encodings presents the encoding variants, operand and effective-address grammar, and size selectors.

For an extended encoding, the four-bit L field selects a 3+L-byte encoded instruction length. The required instruction length is the opcode-space length plus the selected operand payload lengths. Every larger encoded instruction length through 18 bytes is valid, and the trailing bytes are uninterpreted padding. The `LEN n,` spelling records an explicit encoded instruction length.

18 GENERAL INSTRUCTIONS

18.1 Summary

Table 18-1. General Instructions Summary (Informative)

Mnemonic	Brief description
ABS	Replaces the destination with its selected-width two's-complement absolute value.
ADC	Adds source and carry-in to the destination, writes the modular sum, and updates ZNCV.
ADD	Adds the source operand to the destination operand.
AFENCE	Order prior memory operations before later memory operations.
AND	Computes the bitwise AND of the source and destination operands.
BCHG	Toggles the indexed destination bit, sets Z when it was clear, and clears NCV.
BFEXTS	Extracts a contiguous bitfield and sign-extends it.
BFEXTU	Extracts a contiguous bitfield and zero-extends it.
BFINS	Inserts low source bits into a contiguous destination bitfield.
BCLR	Clears the indexed destination bit, sets Z when it was clear, and clears NCV.
BIC	Clears destination bits selected by the source operand.
BKPT	Raise the breakpoint exception.
BNDSII	Checks signed value membership in the inclusive-inclusive interval [lo, hi].
BNDSIX	Checks signed value membership in the inclusive-exclusive interval [lo, hi).
BNDSXI	Checks signed value membership in the exclusive-inclusive interval (lo, hi].
BNDSXX	Checks signed value membership in the exclusive-exclusive interval (lo, hi).
BNDUII	Checks unsigned value membership in the inclusive-inclusive interval [lo, hi].
BNDUIX	Checks unsigned value membership in the inclusive-exclusive interval [lo, hi).
BNDUXI	Checks unsigned value membership in the exclusive-inclusive interval (lo, hi].
BNDUXX	Checks unsigned value membership in the exclusive-exclusive interval (lo, hi).
BSET	Sets the indexed destination bit, sets Z when it was clear, and clears NCV.
BTEST	Tests the indexed destination bit without writing it, sets Z when it is clear, and clears NCV.
CALL	Pushes the continuation PC and atomically transfers control to the validated target.
CALLcc	Performs a call when the condition is true.
CLMUL	Writes the low selected-width half of the carryless source-by-destination product.
CLMULH	Writes the high selected-width half of the carryless source-by-destination product.
CLR	Writes zero to the selected-width destination.
CLRLINK	Atomically discards the resident continuation without changing the CFI key.
CLS	Counts leading one bits in the selected-width source and writes the count.
CLZ	Counts leading zero bits in the selected-width source and writes the count.
CMP	Computes a comparison result and updates condition codes without storing the temporary value.
CMPJcc	Compares two Rn registers and branches on the selected condition without writing FLAGS.
CMPXCHG	Atomically compares memory with the expected register, conditionally stores, and returns the observed value.
CMPXCHGP	Atomically compares and conditionally exchanges a 128-bit memory value using two register pairs.
CPUID	Queries architectural discovery information through a selected Rn register.
CRC32	Updates a 32-bit CRC accumulator with selected source bytes.
CRC32C	Updates a 32-bit CRC-32C accumulator with selected source bytes.

Table 18-1 (continued)

Mnemonic	Brief description
CTS	Counts trailing one bits in the selected-width source and writes the count.
CTZ	Counts trailing zero bits in the selected-width source and writes the count.
DEC	Decrements the selected-width destination modulo its width without changing FLAGS.
DECF	Decrements the destination operand and updates integer FLAGS.
DIVMODS	Divides a signed quotient-register dividend by source and writes quotient and remainder registers.
DIVMODU	Divides an unsigned quotient-register dividend by source and writes quotient and remainder registers.
DIVS	Divides the signed destination by source with truncation toward zero and writes the quotient.
DIVU	Divides the unsigned destination by source and writes the quotient.
DJcc	Decrements the counter and branches only when its new value is nonzero and the condition is true.
ERET	Return through a staged user context or from an architectural event.
EXTRACT	Extracts a selected-width value from a concatenated pair of Rn registers.
FCALL	Installs a resident continuation and transfers without return-record memory traffic.
EXTSL	Sign-extends the selected source width to a longword EA destination.
EXTSQ	Sign-extends the selected source width to a quadword destination.
EXTSW	Sign-extends a byte source to a word EA destination.
EXTZL	Zero-extends the selected source width to a longword EA destination.
EXTZQ	Zero-extends the selected source width to a quadword destination.
EXTZW	Zero-extends a byte source to a word EA destination.
FETCHADD	Atomically adds source to memory and returns the old memory value in the source register.
FETCHAND	Atomically ANDs source with memory and returns the old memory value in the source register.
FETCHANDN	Atomically clears memory bits selected by a source and returns the old value.
FETCHMAXS	Atomically stores the signed maximum of memory and source and returns the old value.
FETCHMAXU	Atomically stores the unsigned maximum of memory and source and returns the old value.
FETCHMINS	Atomically stores the signed minimum of memory and source and returns the old value.
FETCHMINU	Atomically stores the unsigned minimum of memory and source and returns the old value.
FETCHOR	Atomically ORs source with memory and returns the old memory value in the source register.
FETCHSUB	Atomically subtracts source from memory and returns the old memory value in the source register.
FETCHXCHG	Atomically exchanges a register with memory and returns the old memory value.
FETCHXOR	Atomically XORs source with memory and returns the old memory value in the source register.
FLSHDCACHE	Write back and invalidate one cache-maintenance block.
HALT	Halt the executing logical processor until an asynchronous architectural event is admitted.
IJcc	Increments the index and branches only when it differs from the bound and the condition is true.
ILLEGAL	Raise illegal-instruction exception.
INC	Increments the selected-width destination modulo its width without changing FLAGS.
INCF	Increments the destination operand and updates integer FLAGS.

Table 18-1 (continued)

Mnemonic	Brief description
INVASID	Invalidate TLB entries for ASID.
INVDCACHE	Invalidate one data-cache maintenance block.
INVICACHE	Invalidate one local instruction-cache maintenance block.
INVPAGE	Invalidate TLB mapping containing a linear address.
INVTLB	Invalidate all TLB entries.
Jcc	Transfers control when the encoded condition is true.
JMP	Transfers control to the target address.
LCALL	Pushes a long return frame and atomically transfers control to a new CS:PC pair.
LDP	Loads one 128-bit memory value into a canonical register pair.
LEA	Computes the source effective address without reading memory and writes it to the destination.
LJMP	Atomically transfers control to a new CS:PC pair.
LRET	Pops a long return frame and atomically restores CS:PC.
MADD	Adds the low product of two sources to the destination.
MAXS	Writes the signed greater of source and destination to the destination.
MAXU	Writes the unsigned greater of source and destination to the destination.
MINS	Writes the signed lesser of source and destination to the destination.
MINU	Writes the unsigned lesser of source and destination to the destination.
MODS	Writes the signed remainder of destination-by-source truncation-toward-zero division.
MODU	Writes the unsigned remainder of destination-by-source division.
MOV	Copies the source operand to the destination operand.
MOVACQ	Loads from memory with acquire ordering.
MOVcc	Copies source to destination only when the encoded condition is true.
MOVCU	Copies from a current-domain source to user-domain memory.
MOVNT	Stores an Rn register value to memory with a non-temporal access hint.
MOVREL	Stores to memory with release ordering.
MOVUC	Copies from user-domain memory to a current-domain destination.
MOVUU	Copies from user-domain memory to user-domain memory.
MSUB	Subtracts the low product of two sources from the destination.
MUL	Multiplies the destination by the source and keeps the low half.
MULHS	Writes the high half of the signed 64-bit source-by-destination product.
MULHSU	Writes the high half of the signed-destination by unsigned-source 64-bit product.
MULHU	Writes the high half of the unsigned 64-bit source-by-destination product.
NEG	Writes the selected-width two's-complement negation of the destination.
NOP	No architectural state change.
NOT	Writes the selected-width bitwise complement of the destination.
OR	Computes the bitwise OR of the source and destination operands.
PARITY	Computes operand parity and writes the predicate to an Rn register.
POP	Pops one 64-bit stack value into an Rn or SREG destination.
POPCNT	Counts set bits in the selected-width source and writes the count.
POPP	Pops one canonical Rn register pair selected by an inline pair index.
PREFETCH	Hints that EA will be read soon; faults only under the configured prefetch-fault policy.
PREFETCHNT	Hint that EA will be read with non-temporal locality.
PTQUERY	Reads the selected raw page-table descriptor without changing translation state.
PUSH	Pushes one Rn register value, SREG image, or the current CS image onto the stack.

Table 18-1 (continued)

Mnemonic	Brief description
PUSHP	Pushes one canonical Rn register pair selected by an inline pair index.
RDCR	Read a privileged control register.
RDFLAGS	Read the integer condition-code flags into an Rn register.
RDPMC	Read the 64-bit performance counter selected by a constant counter ID.
RDTIME	Read one atomic snapshot of the invariant architectural timebase.
RDSEG	Read an SREG or the current CS image into an Rn register.
RDSTATUS	Read the processor STATUS register into an Rn register.
REPcc	Repeats the next instruction while the counter and condition remain active.
RESET	Perform warm reset; preserve BOOTPC and BOOTCFG.
RESTORE	Restores the base user-state record.
RET	Pops and validates a return PC, then atomically advances SP and transfers control.
REVBIT	Reverses bit order within the selected operand width.
REVBYTE	Reverses byte order within the selected operand width and writes the result.
RFENCE	Order prior reads before later reads.
ROL	Rotates the destination left by the count modulo the selected width without changing FLAGS.
ROR	Rotates the destination right by the count modulo the selected width without changing FLAGS.
SAR	Arithmetic-right-shifts the destination by the count modulo its selected width without changing FLAGS.
SAVE	Saves the base user-state record.
SBB	Subtracts source and C borrow from destination, writes the modular difference, and updates ZNCV.
SRESTORE	Restores the base supervisor-state record.
SSAVE	Saves the base supervisor-state record.
SEGLEA	Computes segment pre-translation of the effective-address point and reports bounds failure in FLAGS.
SET	Writes integer one to the destination operand.
SETcc	Writes one or zero to the destination according to the encoded condition.
SETF	Replaces integer FLAGS from an immediate bitmap.
SPECFENCE	Prevents younger instructions from executing speculatively across the fence.
SHL	Left-shifts the destination by the count modulo its selected width without changing FLAGS.
SHR	Logical-right-shifts the destination by the count modulo its selected width without changing FLAGS.
SUB	Subtracts the source operand from the destination operand.
STP	Stores one canonical register pair as a 128-bit memory value.
SWPT	Atomically installs PTCR and clears ASCR as a page-table switch.
SWPTA	Switches the page table and enables address-space context matching.
SYNCCACHE	Synchronize one cache-maintenance block for local instruction fetch.
SYSCALL	Raise the SYSTEM_CALL architectural event.
TARM	Atomically install an absolute deadline and interrupt identity.
TCANCEL	Atomically disarm the current deadline timer.
TEST	Computes a bitwise conjunction and updates condition codes without storing the temporary value.
TESTJcc	Tests two Rn registers and branches on the selected condition without writing FLAGS.

Table 18-1 (continued)

Mnemonic	Brief description
TRACE	Records a marker and instruction boundary in the implementation trace stream.
VTOP	Queries the current translation of a linear address without changing translation state.
RELAX	Permits a finite implementation-selected delay, including zero.
WFENCE	Order prior writes before later writes.
WRBKDCACHE	Write back one data-cache maintenance block.
WRCR	Write a privileged control register from an Rn register.
WRFLAGS	Write the integer condition-code flags from an Rn register.
WRSEG	Write an SREG segment image from an Rn register.
WRSTATUS	Write the processor STATUS register from an Rn register.
XCHG	Exchanges selected-width operand values, committing both destinations together.
XOR	Computes the bitwise XOR of the source and destination operands.
YIELD	Hints that the current logical processor may be deprioritized.

ABS — Absolute Value — ABS

Operation: Replaces the destination with its selected-width two's-complement absolute value.

Assembler Syntax:
```
ABS.{L|Q}(z) Rn(r)
ABS.{B|W|L|Q}(z) <ea>(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

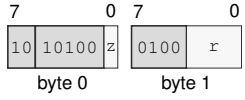
Detailed Semantics

Replaces the selected-width destination value with its two's-complement absolute value. The most-negative value wraps to itself and does not trap. A sub-Q Rn result is sign-extended to 64 bits.

Encodings:

`ABS.{L|Q}(z) Rn(r)`

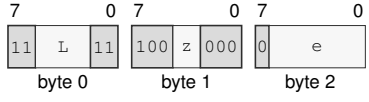
Instruction Format:



Format: Instruction format for ABS.{L|Q}(z) Rn(r)

`ABS.{B|W|L|Q}(z) <ea>(e)`

Instruction Format:



Format: Instruction format for ABS.{B|W|L|Q}(z) <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

ADC — Add With Carry — ADC

Operation: Adds source and carry-in to the destination, writes the modular sum, and updates ZNCV.

Assembler Syntax:

```
ADC.{B|W|L|Q}(z) Rn(s), Rn(d)
ADC.{B|W|L|Q}(z) Rn(s), <ea>(e)
ADC.{B|W|L|Q}(z) <ea>(e), Rn(s)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

Detailed Semantics

Adds the selected-width source value and the incoming `FLAGS.C` flag to the destination, writes the truncated sum, and updates `FLAGS.Z`, `FLAGS.N`, `FLAGS.C`, and `FLAGS.V` from the complete addition. The written destination result supplies the REPcc observed value.

Let n be the selected width, d and s the unsigned operand bit patterns, and c the incoming `FLAGS.C`. The written result is

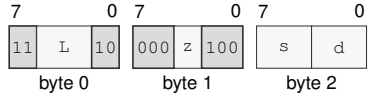
$$r = (d + s + c) \bmod 2^n.$$

`FLAGS.Z` is set when $r = 0$, and `FLAGS.N` receives the most-significant bit of r. `FLAGS.C` reports an unsigned carry out of either addition. `FLAGS.V` reports signed overflow for the complete $d + s + c$ operation.

Encodings:

`ADC.{B|W|L|Q}(z) Rn(s), Rn(d)`

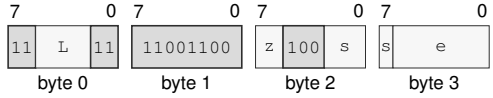
Instruction Format:



Format: Instruction format for ADC.{B|W|L|Q}(z) Rn(s), Rn(d)

`ADC.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



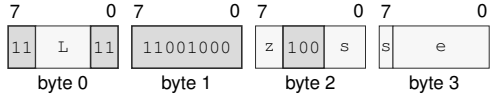
Format: Instruction format for ADC.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`ADC.{B|W|L|Q}(z) <ea>(e), Rn(s)`

Instruction Format:



Format: Instruction format for ADC.{B|W|L|Q}(z) <ea>(e), Rn(s)

ADD — Add — ADD

Operation: Adds the source operand to the destination operand.

Assembler Syntax:

```
ADD.Q 8, SP
ADD.{L|Q}(z) Rn(s), Rn(d)
ADD.Q imm8(i), SP
ADD.Q <imm16s>, SP
ADD.Q <imm32s>, SP
ADD.{B|W|L|Q}(z) Rn(s), <ea>(e)
ADD.{B|W|L|Q}(z) <ea>(e), Rn(d)
ADD.{B|W|L|Q}(z) <imm8s>, <ea>(e)
ADD.{W|L|Q}(z) <imm16s>, <ea>(e)
ADD.{L|Q}(z) <imm32s>, <ea>(e)
ADD.Q <imm64>, <ea>(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

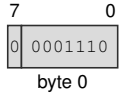
Detailed Semantics

ADD reads the source and destination operands at the selected integer width, adds their values modulo that width, and writes the low result bits to the destination. The common integer result-extension rule defines the upper bits of an Rn destination. The operation leaves FLAGS unchanged.

Encodings:

`ADD.Q 8, SP`

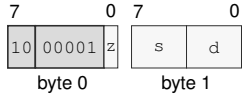
Instruction Format:



Format: Instruction format for ADD.Q 8, SP

`ADD.{L|Q}(z) Rn(s), Rn(d)`

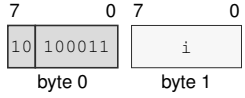
Instruction Format:



Format: Instruction format for ADD.{L|Q}(z) Rn(s), Rn(d)

`ADD.Q imm8(i), SP`

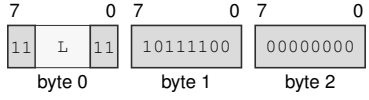
Instruction Format:



Format: Instruction format for ADD.Q imm8(i), SP

`ADD.Q <imm16s>, SP`

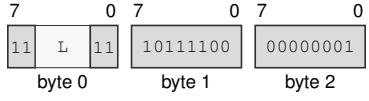
Instruction Format:



Format: Instruction format for ADD.Q <imm16s>, SP

`ADD.Q <imm32s>, SP`

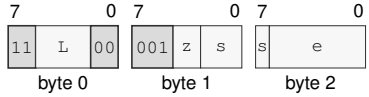
Instruction Format:



Format: Instruction format for ADD.Q <imm32s>, SP

`ADD.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



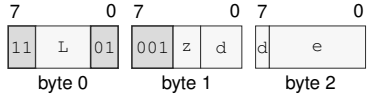
Format: Instruction format for ADD.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`ADD.{B|W|L|Q}(z) <ea>(e), Rn(d)`

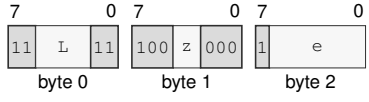
Instruction Format:



Format: Instruction format for ADD.{B|W|L|Q}(z) <ea>(e), Rn(d)

`ADD.{B|W|L|Q}(z) <imm8s>, <ea>(e)`

Instruction Format:



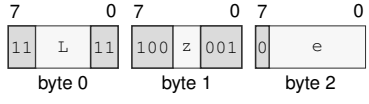
Format: Instruction format for ADD.{B|W|L|Q}(z) <imm8s>, <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`ADD.{W|L|Q}(z) <imm16s>, <ea>(e)`

Instruction Format:



Format: Instruction format for ADD.{W|L|Q}(z) <imm16s>, <ea>(e)

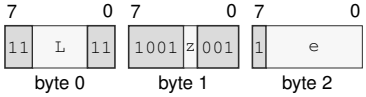
Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `immediate_wider_than_operation_size`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`ADD.{L|Q}(z) <imm32s>, <ea>(e)`

Instruction Format:



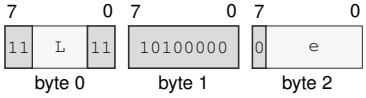
Format: Instruction format for ADD.{L|Q}(z) <imm32s>, <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`ADD.Q <imm64>, <ea>(e)`

Instruction Format:



Format: Instruction format for ADD.Q <imm64>, <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

AFENCE — Address Fence — AFENCE

Operation: Order prior memory operations before later memory operations.

Assembler Syntax: `AFENCE`

Privilege: Unprivileged

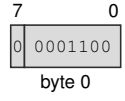
Detailed Semantics

Acts as a full cumulative fence. Every earlier memory read and write is ordered before every later memory read and write on the same logical processor, and memory effects observed before the fence are propagated before effects ordered after it. Every AFENCE participates in the single global sequentially consistent order. It performs no memory access and changes no register or flag.

Encodings:

`AFENCE`

Instruction Format:



Format: Instruction format for AFENCE

AND — Bitwise And — AND

Operation: Computes the bitwise AND of the source and destination operands.

Assembler Syntax:

```
AND.{L|Q}(z) Rn(s), Rn(d)
AND.{B|W|L|Q}(z) Rn(s), <ea>(e)
AND.{B|W|L|Q}(z) <ea>(e), Rn(d)
AND.{B|W|L|Q}(z) <imm8s>, <ea>(e)
AND.{W|L|Q}(z) <imm16s>, <ea>(e)
AND.{L|Q}(z) <imm32s>, <ea>(e)
AND.Q <imm64>, <ea>(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

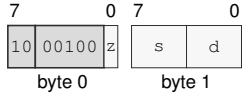
Detailed Semantics

Writes the selected-width bitwise conjunction of the source and destination values to the destination.

Encodings:

`AND.{L|Q}(z) Rn(s), Rn(d)`

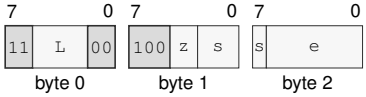
Instruction Format:



Format: Instruction format for AND.{L|Q}(z) Rn(s), Rn(d)

`AND.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



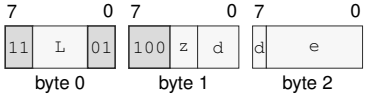
Format: Instruction format for AND.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`AND.{B|W|L|Q}(z) <ea>(e), Rn(d)`

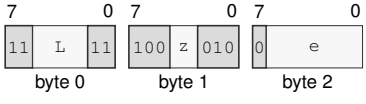
Instruction Format:



Format: Instruction format for AND.{B|W|L|Q}(z) <ea>(e), Rn(d)

`AND.{B|W|L|Q}(z) <imm8s>, <ea>(e)`

Instruction Format:



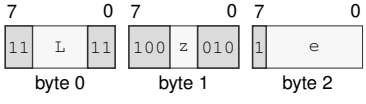
Format: Instruction format for AND.{B|W|L|Q}(z) <imm8s>, <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`AND.{W|L|Q}(z) <imm16s>, <ea>(e)`

Instruction Format:



Format: Instruction format for AND.{W|L|Q}(z) <imm16s>, <ea>(e)

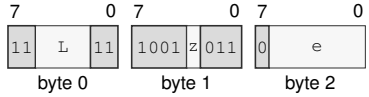
Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `immediate_wider_than_operation_size`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`AND.{L|Q}(z) <imm32s>, <ea>(e)`

Instruction Format:



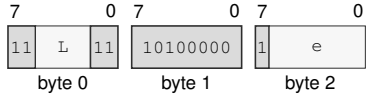
Format: Instruction format for AND.{L|Q}(z) <imm32s>, <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`AND.Q <imm64>, <ea>(e)`

Instruction Format:



Format: Instruction format for AND.Q <imm64>, <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

BCHG Bit Change BCHG

Operation: Toggles the indexed destination bit, sets Z when it was clear, and clears NCV.

Assembler Syntax:

```
BCHG Rn(b), <ea>(e)
BCHG imm6(i), <ea>(e)
BCHG Rn(b), Rn(e)
BCHG imm6(i), Rn(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `computed`

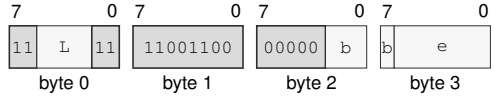
Detailed Semantics

Complements the destination bit selected by the low six bits of the bit index, writes `FLAGS.Z` from whether that bit was previously clear, and clears `FLAGS.N`, `FLAGS.C`, and `FLAGS.V`. A memory destination is read by an ordinary selected-width load and updated by an ordinary selected-width store. The B-sized 0 or 1 value of the old tested bit supplies the REPcc observed value.

Encodings:

`BCHG Rn(b), <ea>(e)`

Instruction Format:



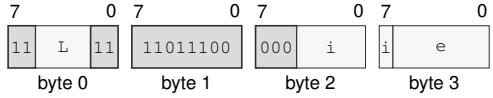
Format: Instruction format for BCHG Rn(b), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`BCHG imm6(i), <ea>(e)`

Instruction Format:



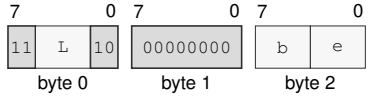
Format: Instruction format for BCHG imm6(i), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`BCHG Rn(b), Rn(e)`

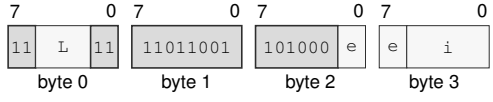
Instruction Format:



Format: Instruction format for BCHG Rn(b), Rn(e)

`BCHG imm6(i), Rn(e)`

Instruction Format:



Format: Instruction format for BCHG imm6(i), Rn(e)

BFEXTS Bitfield Extract Signed BFEXTS

Operation: Extracts a contiguous bitfield and sign-extends it.

Assembler Syntax: `BFEXTS.{B|W|L|Q}(z) imm6(l), imm6(w), Rn(s), Rn(d)`

Privilege: Unprivileged

Required CPUID flags: `BFEXTS`

Repeat eligibility: REP eligible; REPcc observes `dst`

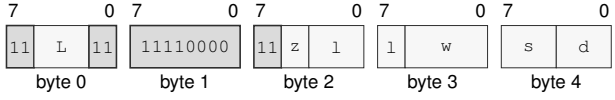
Detailed Semantics

Extracts extttimm6(w)+1 bits beginning at bit extttimm6(l) of the selected-width source and writes the sign-extended field to the destination. An encoding is invalid when the selected field extends past the operand width.

Encodings:

`BFEXTS.{B|W|L|Q}(z) imm6(l), imm6(w), Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for BFEXTS.{B|W|L|Q}(z) imm6(l), imm6(w), Rn(s), Rn(d)

BFEXTU Bitfield Extract Unsigned BFEXTU

Operation: Extracts a contiguous bitfield and zero-extends it.

Assembler Syntax: `BFEXTU.{B|W|L|Q}(z) imm6(l), imm6(w), Rn(s), Rn(d)`

Privilege: Unprivileged

Required CPUID flags: `BFEXTU`

Repeat eligibility: REP eligible; REPcc observes `dst`

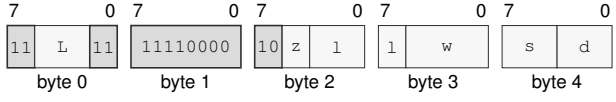
Detailed Semantics

Extracts extttimm6(w)+1 bits beginning at bit extttimm6(l) of the selected-width source and writes the zero-extended field to the destination. An encoding is invalid when the selected field extends past the operand width.

Encodings:

`BFEXTU.{B|W|L|Q}(z) imm6(l), imm6(w), Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for BFEXTU.{B|W|L|Q}(z) imm6(l), imm6(w), Rn(s), Rn(d)

BFINS Bitfield Insert BFINS

Operation: Inserts low source bits into a contiguous destination bitfield.

Assembler Syntax: `BFINS.{B|W|L|Q}(z) imm6(l), imm6(w), Rn(s), Rn(d)`

Privilege: Unprivileged

Required CPUID flags: `BFINS`

Repeat eligibility: REP eligible; REPcc observes `dst`

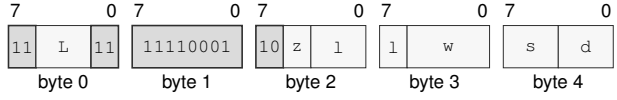
Detailed Semantics

Replaces extttimm6(w)+1 destination bits beginning at bit extttimm6(l) with the same number of low source bits. Destination bits outside the field are preserved. An encoding is invalid when the selected field extends past the operand width.

Encodings:

`BFINS.{B|W|L|Q}(z) imm6(l), imm6(w), Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for BFINS.{B|W|L|Q}(z) imm6(l), imm6(w), Rn(s), Rn(d)

BCLR — Bit Clear — BCLR

Operation: Clears the indexed destination bit, sets Z when it was clear, and clears NCV.

Assembler Syntax:

```
BCLR Rn(b), <ea>(e)
BCLR imm6(i), <ea>(e)
BCLR Rn(b), Rn(e)
BCLR imm6(i), Rn(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `computed`

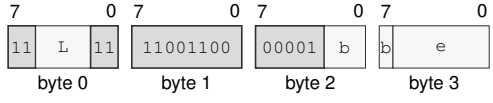
Detailed Semantics

Clears the destination bit selected by the low six bits of the bit index, writes `FLAGS.Z` from whether that bit was previously clear, and clears `FLAGS.N`, `FLAGS.C`, and `FLAGS.V`. A memory destination is read by an ordinary selected-width load and updated by an ordinary selected-width store. The B-sized 0 or 1 value of the old tested bit supplies the REPcc observed value.

Encodings:

`BCLR Rn(b), <ea>(e)`

Instruction Format:



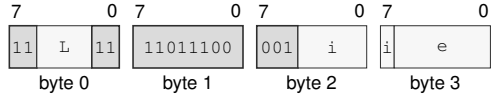
Format: Instruction format for BCLR Rn(b), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`BCLR imm6(i), <ea>(e)`

Instruction Format:



Format: Instruction format for BCLR imm6(i), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

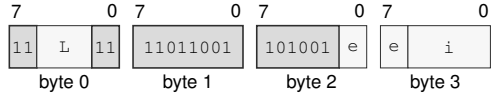
`BCLR Rn(b), Rn(e)`

Instruction Format:

Format: Instruction format for BCLR Rn(b), Rn(e)

`BCLR imm6(i), Rn(e)`

Instruction Format:



Format: Instruction format for BCLR imm6(i), Rn(e)

BIC — Bit Clear — BIC

Operation: Clears destination bits selected by the source operand.

Assembler Syntax: `BIC.{B|W|L|Q}(z) Rn(s), Rn(d)`

Privilege: Unprivileged

Required CPUID flags: `BIC`

Repeat eligibility: REP eligible; REPcc observes `dst`

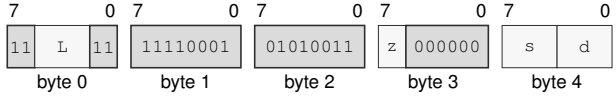
Detailed Semantics

Writes the selected-width value $destination \ \& \ \neg source$ to the destination.

Encodings:

`BIC.{B|W|L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for BIC.{B|W|L|Q}(z) Rn(s), Rn(d)

BKPT Breakpoint BKPT

Operation: Raise the breakpoint exception.

Assembler Syntax: `BKPT`

Privilege: Unprivileged

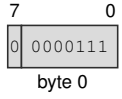
Detailed Semantics

Raises `EXPLICIT_BREAKPOINT` after BKPT completes and saves the next instruction address without executing that instruction before event entry commits.

Encodings:

`BKPT`

Instruction Format:



Format: Instruction format for BKPT

BNDSII — Signed Bounds Check Inclusive-Inclusive — BNDSII

Operation: Checks signed value membership in the inclusive-inclusive interval [lo, hi].

Assembler Syntax:
```
BNDSII.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)
BNDSII.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `computed`

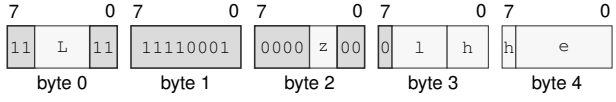
Detailed Semantics

BNDSII treats both bounds as inclusive. It sets `FLAGS.V` when signed(value) is outside [signed(lo), signed(hi)] and clears `FLAGS.Z`, `FLAGS.N`, and `FLAGS.C`. The B-sized 0 or 1 violation result supplies the REPcc observed value.

Encodings:

`BNDSII.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)`

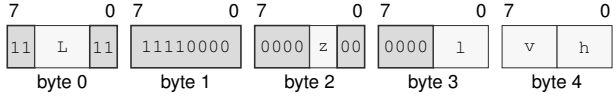
Instruction Format:



Format: Instruction format for BNDSII.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)

`BNDSII.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)`

Instruction Format:



Format: Instruction format for BNDSII.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)

BNDSIX Signed Bounds Check Inclusive-Exclusive BNDSIX

Operation: Checks signed value membership in the inclusive-exclusive interval [lo, hi).

Assembler Syntax:
```
BNDSIX.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)
BNDSIX.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `computed`

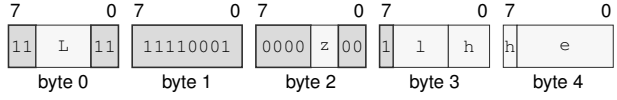
Detailed Semantics

BNDSIX treats the low bound as inclusive and the high bound as exclusive. It sets `FLAGS.V` when signed(value) is outside [signed(lo), signed(hi)) and clears `FLAGS.Z`, `FLAGS.N`, and `FLAGS.C`. The B-sized 0 or 1 violation result supplies the REPcc observed value.

Encodings:

`BNDSIX.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)`

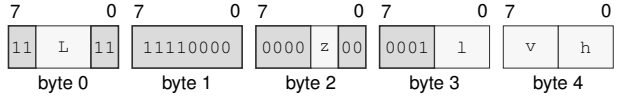
Instruction Format:



Format: Instruction format for BNDSIX.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)

`BNDSIX.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)`

Instruction Format:



Format: Instruction format for BNDSIX.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)

BNDSXI — Signed Bounds Check Exclusive-Inclusive — BNDSXI

Operation: Checks signed value membership in the exclusive-inclusive interval (lo, hi].

Assembler Syntax:
```
BNDSXI.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)
BNDSXI.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `computed`

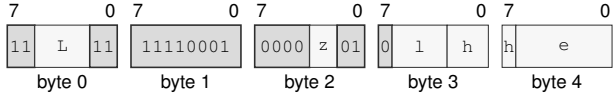
Detailed Semantics

BNDSXI treats the low bound as exclusive and the high bound as inclusive. It sets `FLAGS.V` when signed(value) is outside (signed(lo), signed(hi)] and clears `FLAGS.Z`, `FLAGS.N`, and `FLAGS.C`. The B-sized 0 or 1 violation result supplies the REPcc observed value.

Encodings:

`BNDSXI.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)`

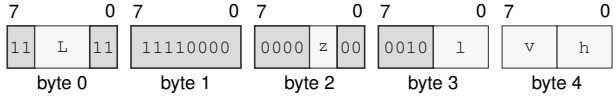
Instruction Format:



Format: Instruction format for BNDSXI.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)

`BNDSXI.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)`

Instruction Format:



Format: Instruction format for BNDSXI.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)

BNDSXX Signed Bounds Check Exclusive-Exclusive BNDSXX

Operation: Checks signed value membership in the exclusive-exclusive interval (lo, hi).

Assembler Syntax:
```
BNDSXX.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)
BNDSXX.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `computed`

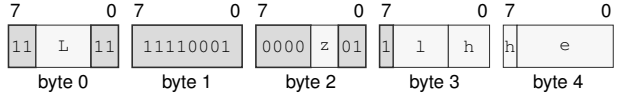
Detailed Semantics

BNDSXX treats both bounds as exclusive. It sets `FLAGS.V` when signed(value) is outside (signed(lo), signed(hi)) and clears `FLAGS.Z`, `FLAGS.N`, and `FLAGS.C`. The B-sized 0 or 1 violation result supplies the REPcc observed value.

Encodings:

`BNDSXX.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)`

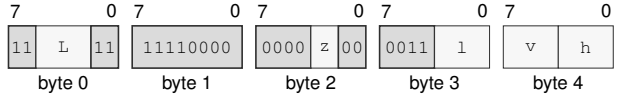
Instruction Format:



Format: Instruction format for BNDSXX.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)

`BNDSXX.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)`

Instruction Format:



Format: Instruction format for BNDSXX.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)

BNDUII — Unsigned Bounds Check Inclusive-Inclusive — BNDUII

Operation: Checks unsigned value membership in the inclusive-inclusive interval [lo, hi].

Assembler Syntax:
```
BNDUII.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)
BNDUII.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `computed`

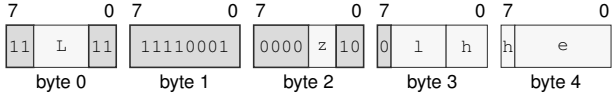
Detailed Semantics

BNDUII treats both bounds as inclusive. It sets `FLAGS.V` when unsigned(value) is outside [unsigned(lo), unsigned(hi)] and clears `FLAGS.Z`, `FLAGS.N`, and `FLAGS.C`. The B-sized 0 or 1 violation result supplies the REPcc observed value.

Encodings:

`BNDUII.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)`

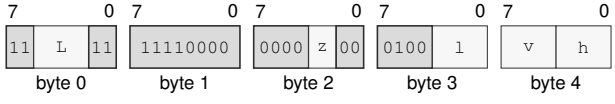
Instruction Format:



Format: Instruction format for BNDUII.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)

`BNDUII.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)`

Instruction Format:



Format: Instruction format for BNDUII.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)

BNDUIX — Unsigned Bounds Check Inclusive-Exclusive — BNDUIX

Operation: Checks unsigned value membership in the inclusive-exclusive interval [lo, hi).

Assembler Syntax:
```
BNDUIX.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)
BNDUIX.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `computed`

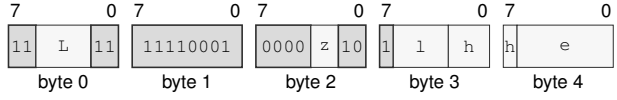
Detailed Semantics

BNDUIX treats the low bound as inclusive and the high bound as exclusive. It sets `FLAGS.V` when unsigned(value) is outside [unsigned(lo), unsigned(hi)) and clears `FLAGS.Z`, `FLAGS.N`, and `FLAGS.C`. The B-sized 0 or 1 violation result supplies the REPcc observed value.

Encodings:

`BNDUIX.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)`

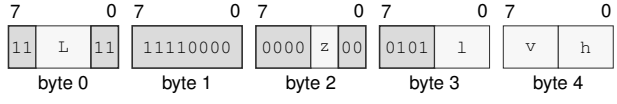
Instruction Format:



Format: Instruction format for BNDUIX.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)

`BNDUIX.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)`

Instruction Format:



Format: Instruction format for BNDUIX.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)

BNDUXI — Unsigned Bounds Check Exclusive-Inclusive — BNDUXI

Operation: Checks unsigned value membership in the exclusive-inclusive interval (lo, hi].

Assembler Syntax:
```
BNDUXI.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)
BNDUXI.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `computed`

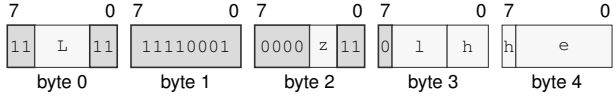
Detailed Semantics

BNDUXI treats the low bound as exclusive and the high bound as inclusive. It sets `FLAGS.V` when unsigned(value) is outside (unsigned(lo), unsigned(hi)] and clears `FLAGS.Z`, `FLAGS.N`, and `FLAGS.C`. The B-sized 0 or 1 violation result supplies the REPcc observed value.

Encodings:

`BNDUXI.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)`

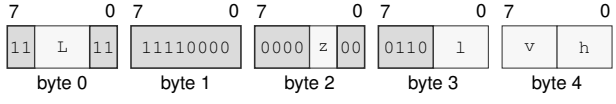
Instruction Format:



Format: Instruction format for BNDUXI.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)

`BNDUXI.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)`

Instruction Format:



Format: Instruction format for BNDUXI.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)

BNDUXX Unsigned Bounds Check Exclusive-Exclusive BNDUXX

Operation: Checks unsigned value membership in the exclusive-exclusive interval (lo, hi).

Assembler Syntax:
```
BNDUXX.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)
BNDUXX.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `computed`

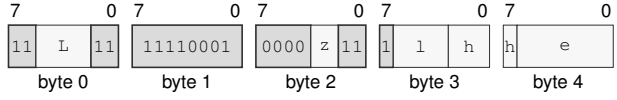
Detailed Semantics

BNDUXX treats both bounds as exclusive. It sets `FLAGS.V` when unsigned(value) is outside (unsigned(lo), unsigned(hi)) and clears `FLAGS.Z`, `FLAGS.N`, and `FLAGS.C`. The B-sized 0 or 1 violation result supplies the REPcc observed value.

Encodings:

`BNDUXX.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)`

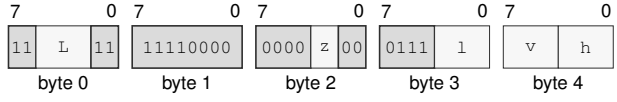
Instruction Format:



Format: Instruction format for BNDUXX.{B|W|L|Q}(z) Rn(l), <ea>(e), Rn(h)

`BNDUXX.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)`

Instruction Format:



Format: Instruction format for BNDUXX.{B|W|L|Q}(z) Rn(l), Rn(v), Rn(h)

BSET — Bit Set — BSET

Operation: Sets the indexed destination bit, sets Z when it was clear, and clears NCV.

Assembler Syntax:

```
BSET Rn(b), <ea>(e)
BSET imm6(i), <ea>(e)
BSET Rn(b), Rn(e)
BSET imm6(i), Rn(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `computed`

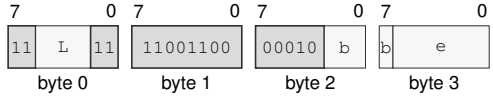
Detailed Semantics

Sets the destination bit selected by the low six bits of the bit index, writes `FLAGS.Z` from whether that bit was previously clear, and clears `FLAGS.N`, `FLAGS.C`, and `FLAGS.V`. A memory destination is read by an ordinary selected-width load and updated by an ordinary selected-width store. The B-sized 0 or 1 value of the old tested bit supplies the REPcc observed value.

Encodings:

`BSET Rn(b), <ea>(e)`

Instruction Format:



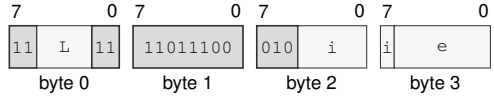
Format: Instruction format for BSET Rn(b), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`BSET imm6(i), <ea>(e)`

Instruction Format:



Format: Instruction format for BSET imm6(i), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

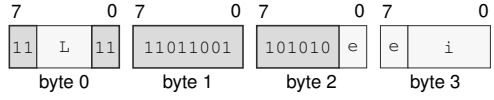
`BSET Rn(b), Rn(e)`

Instruction Format:

Format: Instruction format for BSET Rn(b), Rn(e)

`BSET imm6(i), Rn(e)`

Instruction Format:



Format: Instruction format for BSET imm6(i), Rn(e)

BTEST — Bit Test — BTEST

Operation: Tests the indexed destination bit without writing it, sets Z when it is clear, and clears NCV.

Assembler Syntax:

```
BTEST Rn(b), <ea>(e)
BTEST imm6(i), <ea>(e)
BTEST Rn(b), Rn(e)
BTEST imm6(i), Rn(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `computed`

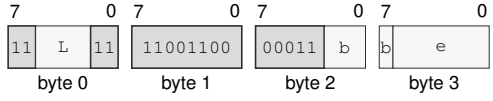
Detailed Semantics

Tests the destination bit selected by the low six bits of the bit index and writes `FLAGS.Z` from whether that bit was clear while clearing `FLAGS.N`, `FLAGS.C`, and `FLAGS.V`, without changing the destination. The B-sized 0 or 1 value of the old tested bit supplies the REPcc observed value.

Encodings:

`BTEST Rn(b), <ea>(e)`

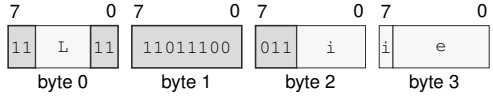
Instruction Format:



Format: Instruction format for BTEST Rn(b), <ea>(e)

`BTEST imm6(i), <ea>(e)`

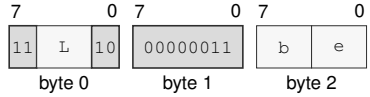
Instruction Format:



Format: Instruction format for BTEST imm6(i), <ea>(e)

`BTEST Rn(b), Rn(e)`

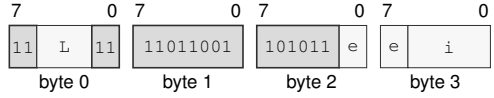
Instruction Format:



Format: Instruction format for BTEST Rn(b), Rn(e)

`BTEST imm6(i), Rn(e)`

Instruction Format:



Format: Instruction format for BTEST imm6(i), Rn(e)

CALL — Call — CALL

Operation: Pushes the continuation PC and atomically transfers control to the validated target.

Assembler Syntax:

```
CALL <imm16s>
CALL <imm32s>
CALL <ea>(e)
CALL Rn(r)
```

Privilege: Unprivileged

Detailed Semantics

Pushes the following instruction's continuation PC as one 64-bit stack value and transfers control to the target. An EA memory operand reads a 64-bit target; an EA register operand supplies its 64-bit value directly. The stack update and control transfer use the ordinary SS:SP and target-validation rules. An active resident link raises `INVALID_CONTROL_TRANSITION` before target or stack access.

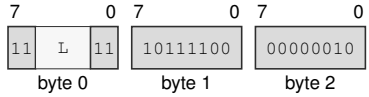
1. Require LPA.A to be zero, then evaluate the target operand in the old architectural context and validate the target under the current CS.
2. Probe the 8-byte return-address store through the old SS:SP context.
3. Commit the return-address store, decremented SP, and new PC as one successful control-transfer operation.

For CALLcc, a false condition is evaluated before target-memory or stack access and suppresses both. Encoding, extension, privilege, and control-state validation still occur. A fault before the final commit changes neither the stack nor PC.

Encodings:

`CALL <imm16s>`

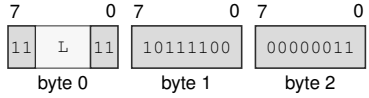
Instruction Format:



Format: Instruction format for CALL <imm16s>

`CALL <imm32s>`

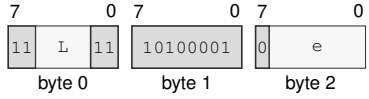
Instruction Format:



Format: Instruction format for CALL <imm32s>

`CALL <ea>(e)`

Instruction Format:



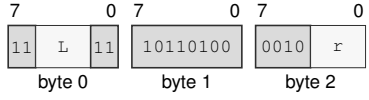
Format: Instruction format for CALL <ea>(e)

Restrictions:

Operand constraint: Role `target`; relation `excluded`; values `immediate`; reason `canonical_form_reclaim`.

`CALL Rn(r)`

Instruction Format:



Format: Instruction format for CALL Rn(r)

CALLcc — Call Conditional — CALLcc

Operation: Performs a call when the condition is true.

Assembler Syntax:

```
CALLcc <imm16s>
CALLcc <imm32s>
CALLcc <ea>(e)
CALLcc Rn(r)
```

Privilege: Unprivileged

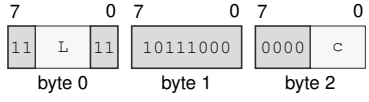
Detailed Semantics

When the encoded condition is true, requires LPA.A to be zero, pushes the following instruction's continuation PC as one 64-bit stack value, and transfers control to the target. When the condition is false, execution continues without resident-link checking, target access, or stack access.

Encodings:

`CALLcc <imm16s>`

Instruction Format:



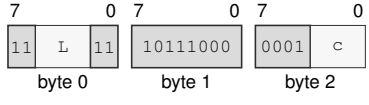
Format: Instruction format for CALLcc <imm16s>

Restrictions:

Operand constraint: Role `cc`; relation `allowed`; values `0x2..0xf`; reason `condition_true_false_reclaimed`.

`CALLcc <imm32s>`

Instruction Format:



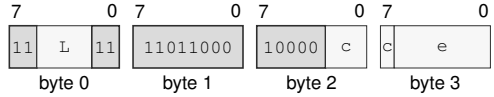
Format: Instruction format for CALLcc <imm32s>

Restrictions:

Operand constraint: Role `cc`; relation `allowed`; values `0x2..0xf`; reason `condition_true_false_reclaimed`.

`CALLcc <ea>(e)`

Instruction Format:



Format: Instruction format for CALLcc <ea>(e)

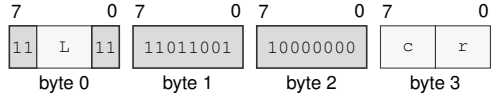
Restrictions:

Operand constraint: Role `cc`; relation `allowed`; values `0x2..0xf`; reason `condition_true_false_reclaimed`.

Operand constraint: Role `target`; relation `excluded`; values `immediate`; reason `canonical_form_reclaim`.

`CALLcc Rn(r)`

Instruction Format:



Format: Instruction format for CALLcc Rn(r)

Restrictions:

Operand constraint: Role `cc`; relation `allowed`; values `0x2..0xf`; reason `condition_true_false_reclaimed`.

CLMUL — Carryless Multiply — CLMUL

Operation: Writes the low selected-width half of the carryless source-by-destination product.

Assembler Syntax:
```
CLMUL.{B|W|L|Q}(z) <ea>(e), Rn(d)
CLMUL.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

Detailed Semantics

Forms the carryless polynomial product of the selected-width source and destination values and writes its low half to the destination.

Treat each selected-width operand as a polynomial over $\mathrm{GF}(2)$, with operand bit i as the coefficient of x^i. Their double-width product is

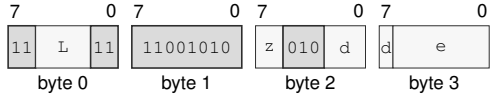
$$P = \bigoplus_{i=0}^{n-1} s_i(d \ll i).$$

Addition in this expression is exclusive-or, so no carry propagates between bit positions. CLMUL writes the low n bits of P to the destination.

Encodings:

`CLMUL.{B|W|L|Q}(z) <ea>(e), Rn(d)`

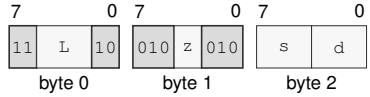
Instruction Format:



Format: Instruction format for CLMUL.{B|W|L|Q}(z) <ea>(e), Rn(d)

`CLMUL.{B|W|L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for CLMUL.{B|W|L|Q}(z) Rn(s), Rn(d)

CLMULH — Carryless Multiply High — CLMULH

Operation: Writes the high selected-width half of the carryless source-by-destination product.

Assembler Syntax:
```
CLMULH.Q <ea>(e), Rn(d)
CLMULH.Q Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

Detailed Semantics

Forms the carryless polynomial product of the selected-width source and destination values and writes its high half to the destination.

Treat each selected-width operand as a polynomial over $\mathrm{GF}(2)$, with operand bit i as the coefficient of x^i. Their double-width product is

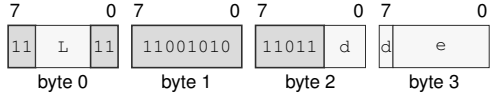
$$P = \bigoplus_{i=0}^{n-1} s_i(d \ll i).$$

Addition in this expression is exclusive-or, so no carry propagates between bit positions. CLMULH writes the high n bits of P to the destination.

Encodings:

```
CLMULH.Q <ea>(e), Rn(d)
```

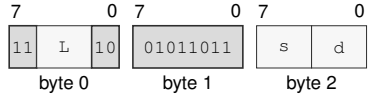
Instruction Format:



Format: Instruction format for CLMULH.Q <ea>(e), Rn(d)

`CLMULH.Q Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for CLMULH.Q Rn(s), Rn(d)

CLR — Clear — CLR

Operation: Writes zero to the selected-width destination.

Assembler Syntax:
```
CLR.Q Rn(r)
CLR.L Rn(r)
CLR.{B|W|L|Q}(z) <ea>(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

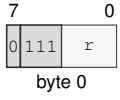
Detailed Semantics

Writes zero to the selected-width destination.

Encodings:

`CLR.Q Rn(r)`

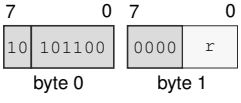
Instruction Format:



Format: Instruction format for CLR.Q Rn(r)

`CLR.L Rn(r)`

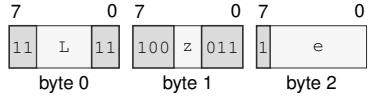
Instruction Format:



Format: Instruction format for CLR.L Rn(r)

`CLR.{B|W|L|Q}(z) <ea>(e)`

Instruction Format:



Format: Instruction format for CLR.{B|W|L|Q}(z) <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

CLRLINK — Clear Resident Link — CLRLINK

Operation: Atomically discards the resident continuation without changing the CFI key.

Assembler Syntax: `CLRLINK`

Privilege: Unprivileged

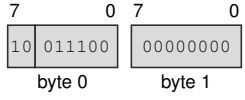
Detailed Semantics

Atomically sets LPC and LPA to zero. CLRLINK is unprivileged, idempotent, performs no memory access, and leaves LKL and LKH unchanged. It abandons a resident continuation without authenticating or returning through it.

Encodings:

`CLRLINK`

Instruction Format:



Format: Instruction format for CLRLINK

CLS Count Leading Set Bits CLS

Operation: Counts leading one bits in the selected-width source and writes the count.

Assembler Syntax:
```
CLS.{B|W|L|Q}(z) <ea>(e), Rn(d)
CLS.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

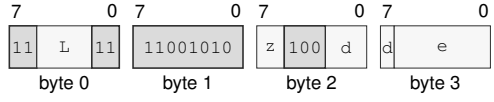
Detailed Semantics

Counts the leading one bits within the selected operand size and writes the count to the destination register. A zero operand returns 0; an operand whose selected bits are all one returns the operand width in bits.

Encodings:

`CLS.{B|W|L|Q}(z) <ea>(e), Rn(d)`

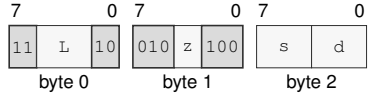
Instruction Format:



Format: Instruction format for CLS.{B|W|L|Q}(z) <ea>(e), Rn(d)

`CLS.{B|W|L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for CLS.{B|W|L|Q}(z) Rn(s), Rn(d)

CLZ — Count Leading Zeros — CLZ

Operation: Counts leading zero bits in the selected-width source and writes the count.

Assembler Syntax:
```
CLZ.{B|W|L|Q}(z) <ea>(e), Rn(d)
CLZ.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

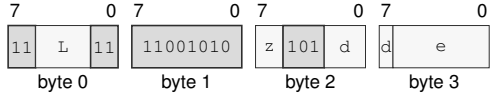
Detailed Semantics

Counts the leading zero bits within the selected operand size and writes the count to the destination register. A zero operand returns the operand width in bits.

Encodings:

`CLZ.{B|W|L|Q}(z) <ea>(e), Rn(d)`

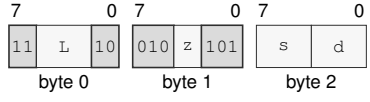
Instruction Format:



Format: Instruction format for CLZ.{B|W|L|Q}(z) <ea>(e), Rn(d)

`CLZ.{B|W|L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for CLZ.{B|W|L|Q}(z) Rn(s), Rn(d)

CMP Compare CMP

Operation: Computes a comparison result and updates condition codes without storing the temporary value.

Assembler Syntax:
```
CMP.{L|Q}(z) Rn(s), Rn(d)
CMP.{B|W|L|Q}(z) Rn(s), <ea>(e)
CMP.{B|W|L|Q}(z) <ea>(e), Rn(d)
CMP.{B|W|L|Q}(z) <ea>(s), <ea>(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `computed`

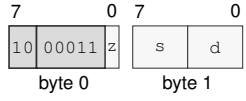
Detailed Semantics

Subtracts the source from the destination only to derive `FLAGS.Z`, `FLAGS.N`, `FLAGS.C`, and `FLAGS.V`; the temporary selected-width difference is not written to either operand. The temporary difference supplies the REPcc observed value.

Encodings:

`CMP.{L|Q}(z) Rn(s), Rn(d)`

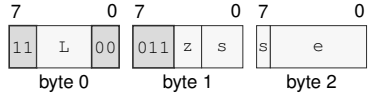
Instruction Format:



Format: Instruction format for CMP.{L|Q}(z) Rn(s), Rn(d)

`CMP.{B|W|L|Q}(z) Rn(s), <ea>(e)`

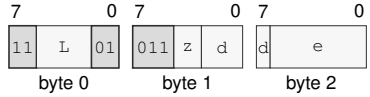
Instruction Format:



Format: Instruction format for CMP.{B|W|L|Q}(z) Rn(s), <ea>(e)

`CMP.{B|W|L|Q}(z) <ea>(e), Rn(d)`

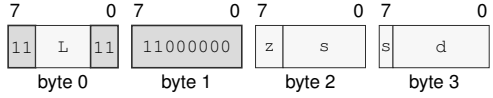
Instruction Format:



Format: Instruction format for CMP.{B|W|L|Q}(z) <ea>(e), Rn(d)

`CMP.{B|W|L|Q}(z) <ea>(s), <ea>(d)`

Instruction Format:



Format: Instruction format for CMP.{B|W|L|Q}(z) <ea>(s), <ea>(d)

CMPJcc — Compare And Jump Conditional — CMPJcc

Operation: Compares two Rn registers and branches on the selected condition without writing FLAGS.

Assembler Syntax:
```
CMPJcc.{B|W|L|Q}(z) Rn(s), Rn(d), <imm8s>
CMPJcc.{B|W|L|Q}(z) Rn(s), Rn(d), <imm16s>
```

Privilege: Unprivileged

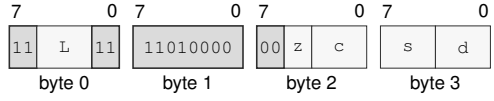
Detailed Semantics

CMPJcc computes the same temporary compare result as CMP for the selected operand size, evaluates the encoded condition from that temporary result, and branches to the relative target when the condition is true. The computed condition is consumed only by the branch decision; architectural FLAGS are not read or written. Conditions T and F are reserved and are not valid encodings.

Encodings:

`CMPJcc.{B|W|L|Q}(z) Rn(s), Rn(d), <imm8s>`

Instruction Format:



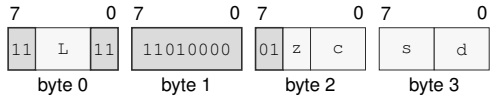
Format: Instruction format for CMPJcc.{B|W|L|Q}(z) Rn(s), Rn(d), <imm8s>

Restrictions:

Operand constraint: Role `cc`; relation `allowed`; values `0x2..0xf`; reason `condition_true_false_reclaimed`.

`CMPJcc.{B|W|L|Q}(z) Rn(s), Rn(d), <imm16s>`

Instruction Format:



Format: Instruction format for CMPJcc.{B|W|L|Q}(z) Rn(s), Rn(d), <imm16s>

Restrictions:

Operand constraint: Role `cc`; relation `allowed`; values `0x2..0xf`; reason `condition_true_false_reclaimed`.

CMPXCHG Compare And Exchange CMPXCHG

Operation: Atomically compares memory with the expected register, conditionally stores, and returns the observed value.

Assembler Syntax: `CMPXCHG.{B|W|L|Q}(z)/order(o) Rn(x), Rn(d), <ea>(e)`

Privilege: Unprivileged

Detailed Semantics

Compares memory with the expected Rn register and conditionally stores the desired Rn register. After the operation, the expected register contains the observed old memory value. `FLAGS.Z=1` means the store happened; `FLAGS.Z=0` means memory was left unchanged.

The memory read, optional memory write, expected-register update, and flag update form one atomic read-modify-write operation. The encoded memory-order selector governs the complete operation on both comparison outcomes. On failure, the operation contributes the observed memory read as its ordering event: acquire orders later memory operations after that read, release orders earlier memory operations before that read, and sequential consistency places that read in the global SC order. On success, the memory write carries the same selected order to observers of that write. A failed comparison contributes no memory write and creates no release sequence.

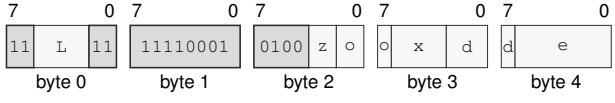
- If memory equals the original expected value, the desired value is stored and `FLAGS.Z` is set.
- Otherwise memory is unchanged and `FLAGS.Z` is cleared.
- In both cases the expected register receives the value originally read from memory, and `FLAGS.N`, `FLAGS.C`, and `FLAGS.V` are cleared.

A memory-access fault exposes none of these updates.

Encodings:

`CMPXCHG.{B|W|L|Q}(z)/order(o) Rn(x), Rn(d), <ea>(e)`

Instruction Format:



Format: Instruction format for CMPXCHG.{B|W|L|Q}(z)/order(o) Rn(x), Rn(d), <ea>(e)

Restrictions:

Operand constraint: Role `order`; relation `allowed`; values `0x0..0x4`; reason `memory_order_reserved_reclaimed`.
Operand constraint: Role `memory`; relation `excluded`; values `immediate`; reason `memory_operand_required`.

CMPXCHGP Compare And Exchange Pair CMPXCHGP

Operation: Atomically compares and conditionally exchanges a 128-bit memory value using two register pairs.

Assembler Syntax: `CMPXCHGP/order(o) PAIRn(x), PAIRn(d), <ea>(e)`

Privilege: Unprivileged

Required CPUID flags: `CMPXCHGP`

Detailed Semantics

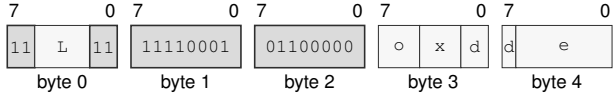
Treats each canonical register pair as a 128-bit little-endian integer: the even register supplies bits 63:0 and the following odd register supplies bits 127:64. The instruction atomically reads a naturally aligned 16-byte memory value, compares it with the original expected pair, and stores the original desired pair only when both halves compare equal. In both cases the expected pair receives the old memory value. `FLAGS.Z` is set exactly when the store occurs; `FLAGS.N`, `FLAGS.C`, and `FLAGS.V` are cleared.

The encoded memory-order selector governs the complete operation on both outcomes under the same success and failure rules as `CMPXCHG`. An alignment or memory-access fault exposes no register, flag, or memory update.

Encodings:

`CMPXCHGP/order(o) PAIRn(x), PAIRn(d), <ea>(e)`

Instruction Format:



Format: Instruction format for CMPXCHGP/order(o) PAIRn(x), PAIRn(d), <ea>(e)

Restrictions:

Operand constraint: Role `order`; relation `allowed`; values `0x0..0x4`; reason `memory_order_reserved_reclaimed`.
Operand constraint: Role `memory`; relation `excluded`; values `immediate`; reason `memory_operand_required`.

CPUID — CPU Identification — CPUID

Operation: Queries architectural discovery information through a selected Rn register.

Assembler Syntax: `CPUID Rn(r)`

Privilege: Unprivileged

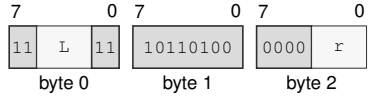
Detailed Semantics

Queries architectural discovery information. The selected Rn register contains the CPUID query selector before execution and receives the selected 64-bit result after execution.

Encodings:

`CPUID Rn(r)`

Instruction Format:



Format: Instruction format for CPUID Rn(r)

CRC32 CRC-32 Update CRC32

Operation: Updates a 32-bit CRC accumulator with selected source bytes.

Assembler Syntax: `CRC32.{B|W|L|Q}(z) <ea>(e), Rn(d)`

Privilege: Unprivileged

Required CPUID flags: `CRC32`

Repeat eligibility: REP eligible; REPcc observes `dst`

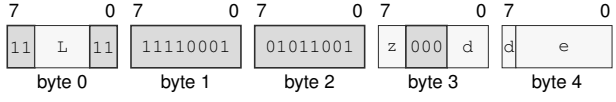
Detailed Semantics

Updates the low 32 bits of the destination as a reflected CRC-32 accumulator using polynomial exttt0xEDB88320. Source bytes are consumed from least-significant to most-significant; the instruction applies no implicit initial or final XOR. The result is zero-extended to 64 bits.

Encodings:

`CRC32.{B|W|L|Q}(z) <ea>(e), Rn(d)`

Instruction Format:



Format: Instruction format for CRC32.{B|W|L|Q}(z) <ea>(e), Rn(d)

CRC32C CRC-32C Update CRC32C

Operation: Updates a 32-bit CRC-32C accumulator with selected source bytes.

Assembler Syntax: `CRC32C.{B|W|L|Q}(z) <ea>(e), Rn(d)`

Privilege: Unprivileged

Required CPUID flags: `CRC32C`

Repeat eligibility: REP eligible; REPcc observes `dst`

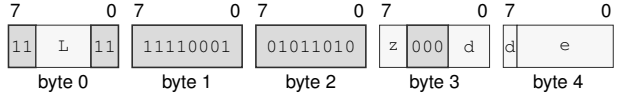
Detailed Semantics

Updates the low 32 bits of the destination as a reflected CRC-32C accumulator using Castagnoli polynomial exttt0x82F63B78. Source bytes are consumed from least-significant to most-significant; the instruction applies no implicit initial or final XOR. The result is zero-extended to 64 bits.

Encodings:

`CRC32C.{B|W|L|Q}(z) <ea>(e), Rn(d)`

Instruction Format:



Format: Instruction format for CRC32C.{B|W|L|Q}(z) <ea>(e), Rn(d)

CTS — Count Trailing Set Bits — CTS

Operation: Counts trailing one bits in the selected-width source and writes the count.

Assembler Syntax:
```
CTS.{B|W|L|Q}(z) <ea>(e), Rn(d)
CTS.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

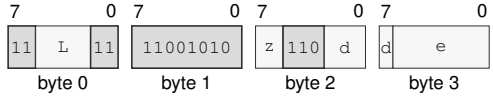
Detailed Semantics

Counts the trailing one bits within the selected operand size and writes the count to the destination register. A zero operand returns 0; an operand whose selected bits are all one returns the operand width in bits.

Encodings:

`CTS.{B|W|L|Q}(z) <ea>(e), Rn(d)`

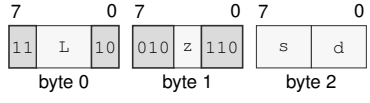
Instruction Format:



Format: Instruction format for CTS.{B|W|L|Q}(z) <ea>(e), Rn(d)

`CTS.{B|W|L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for CTS.{B|W|L|Q}(z) Rn(s), Rn(d)

CTZ — Count Trailing Zeros — CTZ

Operation: Counts trailing zero bits in the selected-width source and writes the count.

Assembler Syntax:
```
CTZ.{B|W|L|Q}(z) <ea>(e), Rn(d)
CTZ.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

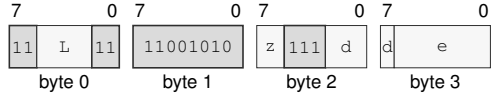
Detailed Semantics

Counts the trailing zero bits within the selected operand size and writes the count to the destination register. A zero operand returns the operand width in bits.

Encodings:

`CTZ.{B|W|L|Q}(z) <ea>(e), Rn(d)`

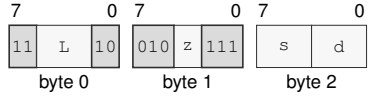
Instruction Format:



Format: Instruction format for CTZ.{B|W|L|Q}(z) <ea>(e), Rn(d)

`CTZ.{B|W|L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for CTZ.{B|W|L|Q}(z) Rn(s), Rn(d)

DEC — Decrement — DEC

Operation: Decrements the selected-width destination modulo its width without changing FLAGS.

Assembler Syntax:
```
DEC.{L|Q}(z) Rn(r)
DEC.{B|W|L|Q}(z) <ea>(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

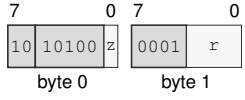
Detailed Semantics

Subtracts one from the selected-width destination and writes the modular result without changing FLAGS.

Encodings:

`DEC.{L|Q}(z) Rn(r)`

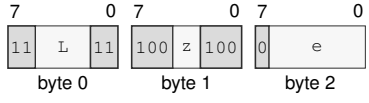
Instruction Format:



Format: Instruction format for DEC.{L|Q}(z) Rn(r)

`DEC.{B|W|L|Q}(z) <ea>(e)`

Instruction Format:



Format: Instruction format for DEC.{B|W|L|Q}(z) <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

DECF — Decrement And Set Flags — DECF

Operation: Decrements the destination operand and updates integer FLAGS.

Assembler Syntax: `DECF.{B|W|L|Q}(z) Rn(r)`

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

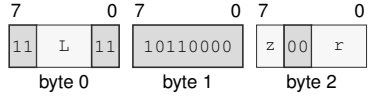
Detailed Semantics

Subtracts one from the selected-width destination, writes the modular result, and updates `FLAGS.Z`, `FLAGS.N`, `FLAGS.C`, and `FLAGS.V`. The written destination result supplies the REPcc observed value.

Encodings:

`DECF.{B|W|L|Q}(z) Rn(r)`

Instruction Format:



Format: Instruction format for DECF.{B|W|L|Q}(z) Rn(r)

DIVMODS — Signed Divide With Remainder — DIVMODS

Operation: Divides a signed quotient-register dividend by source and writes quotient and remainder registers.

Assembler Syntax:
```
DIVMODS.{B|W|L|Q}(z) <ea>(e), Rn(q), Rn(r)
DIVMODS.{B|W|L|Q}(z) Rn(e), Rn(q), Rn(r)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `quotient`

Detailed Semantics

The source operand supplies the signed divisor, and the quotient register Rn(q) supplies the signed dividend before the operation. On success, Rn(q) receives their quotient truncated toward zero and the remainder register Rn(r) receives the signed remainder satisfying dividend = quotient * divisor + remainder. Sub-Q quotient and remainder results are each sign-extended to 64 bits.

Let n be the selected operand width, d the signed two's-complement dividend, and s the signed two's-complement divisor. The quotient and remainder are defined by

$$q = \mathrm{trunc}(d/s), \qquad r = d - qs.$$

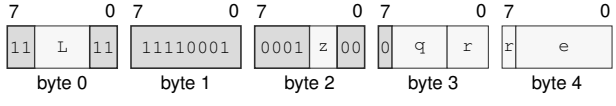
Thus a nonzero remainder has the sign of the dividend. The original quotient operand supplies the dividend. On success the quotient operand receives q, and the separate remainder operand receives r.

A zero divisor raises `DIVIDE_BY_ZERO`; the most-negative value divided by -1 raises `SIGNED_DIVIDE_OVERFLOW`. No destination is changed.

Encodings:

`DIVMODS.{B|W|L|Q}(z) <ea>(e), Rn(q), Rn(r)`

Instruction Format:



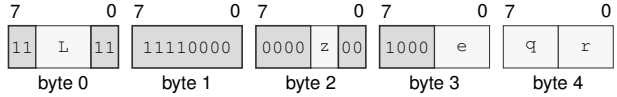
Format: Instruction format for DIVMODS.{B|W|L|Q}(z) <ea>(e), Rn(q), Rn(r)

Restrictions:

Operand overlap: Operands `quotient`, `remainder`; rule `illegal_instruction`.

`DIVMODS.{B|W|L|Q}(z) Rn(e), Rn(q), Rn(r)`

Instruction Format:



Format: Instruction format for DIVMODS.{B|W|L|Q}(z) Rn(e), Rn(q), Rn(r)

Restrictions:

Operand overlap: Operands `quotient`, `remainder`; rule `illegal_instruction`.

DIVMODU — Unsigned Divide With Remainder — DIVMODU

Operation: Divides an unsigned quotient-register dividend by source and writes quotient and remainder registers.

Assembler Syntax:
```
DIVMODU.{B|W|L|Q}(z) <ea>(e), Rn(q), Rn(r)
DIVMODU.{B|W|L|Q}(z) Rn(e), Rn(q), Rn(r)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `quotient`

Detailed Semantics

The source operand supplies the unsigned divisor, and the quotient register Rn(q) supplies the unsigned dividend before the operation. On success, Rn(q) receives their unsigned quotient and the remainder register Rn(r) receives the unsigned remainder satisfying dividend = quotient * divisor + remainder.

Let n be the selected operand width, d the unsigned dividend, and s the unsigned divisor. The quotient and remainder are defined by

$$q = \lfloor d/s \rfloor, \qquad r = d - qs.$$

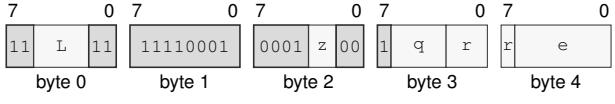
For a nonzero divisor, the remainder satisfies $0 \le r < s$. The original quotient operand supplies the dividend. On success the quotient operand receives q, and the separate remainder operand receives r.

A zero divisor raises `DIVIDE_BY_ZERO`; no destination is changed.

Encodings:

`DIVMODU.{B|W|L|Q}(z) <ea>(e), Rn(q), Rn(r)`

Instruction Format:



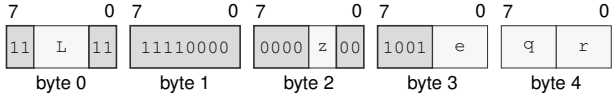
Format: Instruction format for DIVMODU.{B|W|L|Q}(z) <ea>(e), Rn(q), Rn(r)

Restrictions:

Operand overlap: Operands `quotient`, `remainder`; rule `illegal_instruction`.

`DIVMODU.{B|W|L|Q}(z) Rn(e), Rn(q), Rn(r)`

Instruction Format:



Format: Instruction format for DIVMODU.{B|W|L|Q}(z) Rn(e), Rn(q), Rn(r)

Restrictions:

Operand overlap: Operands `quotient`, `remainder`; rule `illegal_instruction`.

DIVS — Signed Divide — DIVS

Operation: Divides the signed destination by source with truncation toward zero and writes the quotient.

Assembler Syntax:
```
DIVS.{B|W|L|Q}(z) <ea>(e), Rn(d)
DIVS.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

Detailed Semantics

Interprets both selected-width operands as signed integers, divides the destination by the source with truncation toward zero, and writes the quotient. A sub-Q quotient is sign-extended to 64 bits. Division by zero raises DIVIDE_BY_ZERO; the most-negative value divided by minus one raises SIGNED_DIVIDE_OVERFLOW.

Let n be the selected operand width, d the signed two's-complement dividend, and s the signed two's-complement divisor. The quotient and remainder are defined by

$$q = \mathrm{trunc}(d/s), \qquad r = d - qs.$$

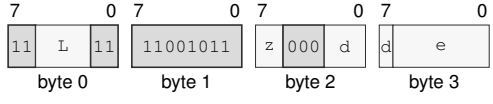
Thus a nonzero remainder has the sign of the dividend. The destination supplies the dividend and receives q.

A zero divisor raises `DIVIDE_BY_ZERO`; the most-negative value divided by -1 raises `SIGNED_DIVIDE_OVERFLOW`. No destination is changed.

Encodings:

`DIVS.{B|W|L|Q}(z) <ea>(e), Rn(d)`

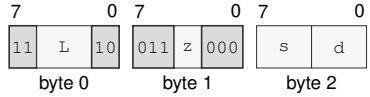
Instruction Format:



Format: Instruction format for DIVS.{B|W|L|Q}(z) <ea>(e), Rn(d)

```
DIVS.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Instruction Format:



Format: Instruction format for DIVS.{B|W|L|Q}(z) Rn(s), Rn(d)

DIVU — Unsigned Divide — DIVU

Operation: Divides the unsigned destination by source and writes the quotient.

Assembler Syntax:
```
DIVU.{B|W|L|Q}(z) <ea>(e), Rn(d)
DIVU.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

Detailed Semantics

Divides the selected-width unsigned destination value by the unsigned source value and writes the quotient. A zero divisor raises DIVIDE_BY_ZERO.

Let n be the selected operand width, d the unsigned dividend, and s the unsigned divisor. The quotient and remainder are defined by

$$q = \lfloor d/s \rfloor, \qquad r = d - qs.$$

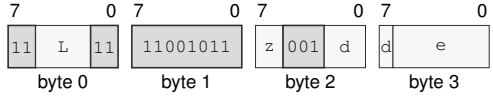
For a nonzero divisor, the remainder satisfies $0 \leq r < s$. The destination supplies the dividend and receives q.

A zero divisor raises `DIVIDE_BY_ZERO`; no destination is changed.

Encodings:

`DIVU.{B|W|L|Q}(z) <ea>(e), Rn(d)`

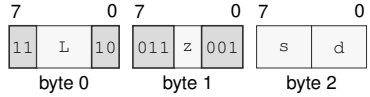
Instruction Format:



Format: Instruction format for DIVU.{B|W|L|Q}(z) <ea>(e), Rn(d)

`DIVU.{B|W|L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for DIVU.{B|W|L|Q}(z) Rn(s), Rn(d)

DJcc — Decrement And Jump Conditional — DJcc

Operation: Decrements the counter and branches only when its new value is nonzero and the condition is true.

Assembler Syntax:
```
DJcc Rn(r), <ea>(e)
DJcc Rn(r), Rn(e)
```

Privilege: Unprivileged

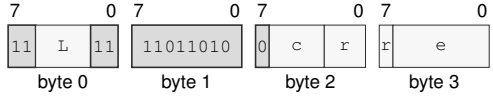
Detailed Semantics

Decrements the selected counter and transfers control only when the new count is nonzero and the encoded condition is true. The target EA has Q interpretation width: a memory form reads a 64-bit target and a register or immediate form supplies its value directly.

Encodings:

`DJcc Rn(r), <ea>(e)`

Instruction Format:



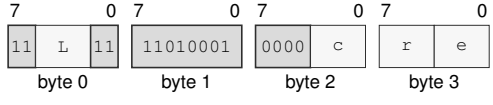
Format: Instruction format for DJcc Rn(r), <ea>(e)

Restrictions:

Operand constraint: Role `cc`; relation `allowed`; values `0, 0x2..0xf`; reason `condition_false_reclaimed`.

`DJcc Rn(r), Rn(e)`

Instruction Format:



Format: Instruction format for DJcc Rn(r), Rn(e)

Restrictions:

Operand constraint: Role `cc`; relation `allowed`; values `0, 0x2..0xf`; reason `condition_false_reclaimed`.

ERET — Event Return — ERET

Operation: Return through a staged user context or from an architectural event.

Assembler Syntax: `ERET`

Privilege: Supervisor only

Detailed Semantics

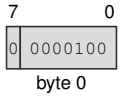
Classify the return source before stack access, then atomically return through a valid staged or outer-event U bank, or restore a valid active supervisor event frame.

1. Classify the return source before any stack access. A staged direct user entry requires supervisor mode, `STATUS.EA` clear, `STATUS.EDEPTH` zero, `STATUS.UO` clear, and `UCTL.V` set. An outer user-event return requires `STATUS.EA` set, `STATUS.EDEPTH` one, `STATUS.UO` set, and `UCTL.V` set. Other active supervisor returns use the memory frame.
2. For a U-bank return, validate UCTL, UINFO, the user segments and pointers, and the executable UPC, then atomically restore user state, clear `UCTL.V`, `STATUS.EDEPTH`, `STATUS.UO`, `STATUS.EA`, and current DFA, and perform no stack access.
3. For a stacked return, read and decode `FRAME_CONTROL` at SS:SP without changing SP. Accept only the defined BASIC, ERROR, PAGE, and AUXILIARY sizes, with all reserved bits zero.
4. Validate that saved `STATUS.EDEPTH` plus one equals the current depth and that saved and current `STATUS.UO` preserve the outer-origin chain.
5. Read the complete selected frame and validate `FRAME_CONTROL.FLAGS`, `FRAME_CONTROL.STATUS`, CS, DS, SS, SP, and an executable canonical PC without changing architectural state.
6. Ignore the value in the fixed-header `PADDING` slot at offset +0x38.
7. Atomically restore the validated architectural state and current DFA from `FRAME_CONTROL.SAVED_DFA`. Hidden repeat state remains clear; if `SAVED_PC` identifies a repeat prefix, that prefix is decoded and executed normally after return.
8. After restoration, admit a pending NMI before the next instruction boundary when `STATUS.NI` is clear and `ECR.NMI_P` is set.

Encodings:

```
ERET
```

Instruction Format:



Format: Instruction format for ERET

EXTRACT — Extract From Register Pair — EXTRACT

Operation: Extracts a selected-width value from a concatenated pair of Rn registers.

Assembler Syntax: `EXTRACT.{B|W|L|Q}(z) imm7(i), Rn(h), Rn(l)`

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `low_dst`

Detailed Semantics

Concatenates Rn(high) as the upper half and the original Rn(low/dst) value as the lower half, shifts the concatenated value right by an unsigned 7-bit immediate offset, and writes the low selected-size result back to Rn(low/dst).

Let n be the selected operand width and form the $2n$-bit concatenation

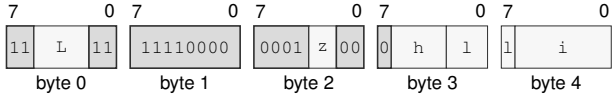
$$T = \mathrm{concat}\big(Rn(\mathrm{high})[n-1:0], Rn(\mathrm{low})[n-1:0]\big).$$

The immediate is an unsigned seven-bit shift count. The destination receives the low n bits of T after a logical right shift by that count. A count greater than or equal to $2n$ produces zero. The high register is not changed.

Encodings:

`EXTRACT.{B|W|L|Q}(z) imm7(i), Rn(h), Rn(l)`

Instruction Format:



Format: Instruction format for EXTRACT.{B|W|L|Q}(z) imm7(i), Rn(h), Rn(l)

FCALL — Fast Call — FCALL

Operation: Installs a resident continuation and transfers without return-record memory traffic.

Assembler Syntax:

```
FCALL <imm16s>
FCALL <imm32s>
FCALL <ea>(e)
FCALL Rn(r)
```

Privilege: Unprivileged

Detailed Semantics

FCALL reserves the ordinary eight-byte near-call stack footprint without accessing a return-record memory location. After validating the target, it commits SP minus eight, installs the following instruction address in LPC with LPA.A one and LPA.PA zero, and transfers within the current CS. A live resident continuation raises `INVALID_CONTROL_TRANSITION` before target or stack access.

Encodings:

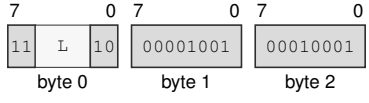
`FCALL <imm16s>`

Instruction Format:

Format: Instruction format for FCALL <imm16s>

`FCALL <imm32s>`

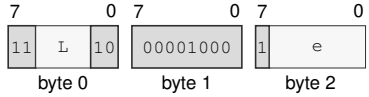
Instruction Format:



Format: Instruction format for FCALL <imm32s>

`FCALL <ea>(e)`

Instruction Format:



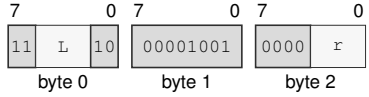
Format: Instruction format for FCALL <ea>(e)

Restrictions:

Operand constraint: Role `target`; relation `excluded`; values `immediate`; reason `canonical_form_reclaim`.

`FCALL Rn(r)`

Instruction Format:



Format: Instruction format for FCALL Rn(r)

EXTSL — Sign Extend To Long — EXTSL

Operation: Sign-extends the selected source width to a longword EA destination.

Assembler Syntax:

```
EXTSL.B Rn(s), <ea>(e)
EXTSL.W Rn(s), <ea>(e)
EXTSL.{B|W}(z) <ea>(s), <ea>(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

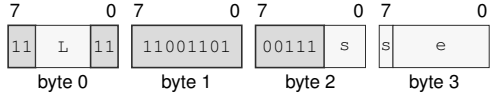
Detailed Semantics

Sign-extends the source operand from the instruction size to long size and stores the four-byte result to an EA destination. The low source-size bytes are copied into the result, and the remaining high result bits are filled with the source sign bit. EXTSL has no Rn destination form; use EXTSQ.B or EXTSQ.W for a sign-extended Rn result.

Encodings:

`EXTSL.B Rn(s), <ea>(e)`

Instruction Format:



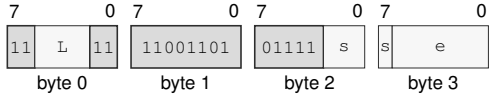
Format: Instruction format for EXTSL.B Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`EXTSL.W Rn(s), <ea>(e)`

Instruction Format:



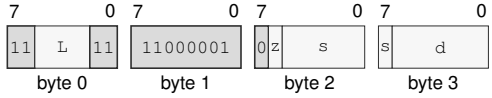
Format: Instruction format for EXTSL.W Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`EXTSL.{B|W}(z) <ea>(s), <ea>(d)`

Instruction Format:



Format: Instruction format for EXTSL.{B|W}(z) <ea>(s), <ea>(d)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

EXTSQ — Sign Extend To Quad — EXTSQ

Operation: Sign-extends the selected source width to a quadword destination.

Assembler Syntax:

```
EXTSQ.B Rn(s), Rn(d)
EXTSQ.W Rn(s), Rn(d)
EXTSQ.L Rn(s), Rn(d)
EXTSQ.B Rn(s), <ea>(e)
EXTSQ.B <ea>(e), Rn(d)
EXTSQ.W Rn(s), <ea>(e)
EXTSQ.W <ea>(e), Rn(d)
EXTSQ.L Rn(s), <ea>(e)
EXTSQ.L <ea>(e), Rn(d)
EXTSQ.B <ea>(s), <ea>(d)
EXTSQ.W <ea>(s), <ea>(d)
EXTSQ.L <ea>(s), <ea>(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

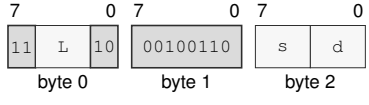
Detailed Semantics

Sign-extends the source operand from the instruction size to quad size. The low source-size bytes are copied into the result, and the remaining high result bits are filled with the source sign bit. An Rn destination receives the complete 64-bit result; an EA destination stores all eight result bytes.

Encodings:

`EXTSQ.B Rn(s), Rn(d)`

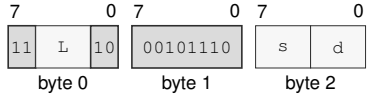
Instruction Format:



Format: Instruction format for EXTSQ.B Rn(s), Rn(d)

`EXTSQ.W Rn(s), Rn(d)`

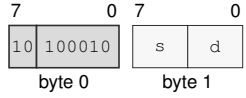
Instruction Format:



Format: Instruction format for EXTSQ.W Rn(s), Rn(d)

`EXTSQ.L Rn(s), Rn(d)`

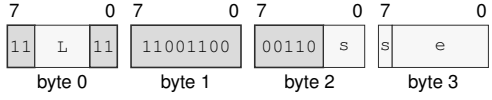
Instruction Format:



Format: Instruction format for EXTSQ.L Rn(s), Rn(d)

`EXTSQ.B Rn(s), <ea>(e)`

Instruction Format:



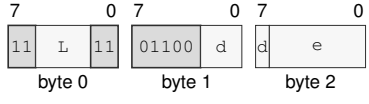
Format: Instruction format for EXTSQ.B Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`EXTSQ.B <ea>(e), Rn(d)`

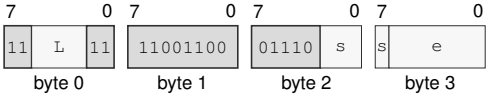
Instruction Format:



Format: Instruction format for EXTSQ.B <ea>(e), Rn(d)

`EXTSQ.W Rn(s), <ea>(e)`

Instruction Format:



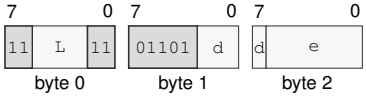
Format: Instruction format for EXTSQ.W Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`EXTSQ.W <ea>(e), Rn(d)`

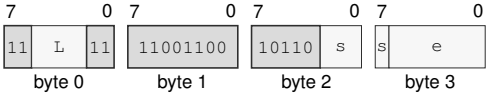
Instruction Format:



Format: Instruction format for EXTSQ.W <ea>(e), Rn(d)

`EXTSQ.L Rn(s), <ea>(e)`

Instruction Format:



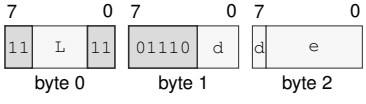
Format: Instruction format for EXTSQ.L Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`EXTSQ.L <ea>(e), Rn(d)`

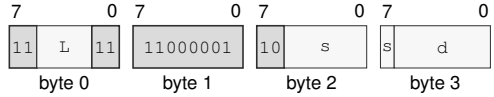
Instruction Format:



Format: Instruction format for EXTSQ.L <ea>(e), Rn(d)

`EXTSQ.B <ea>(s), <ea>(d)`

Instruction Format:



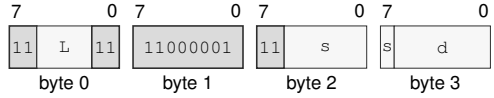
Format: Instruction format for EXTSQ.B <ea>(s), <ea>(d)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`EXTSQ.W <ea>(s), <ea>(d)`

Instruction Format:



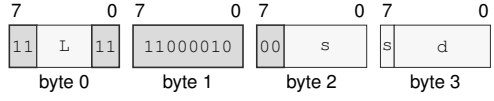
Format: Instruction format for EXTSQ.W <ea>(s), <ea>(d)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`EXTSQ.L <ea>(s), <ea>(d)`

Instruction Format:



Format: Instruction format for EXTSQ.L <ea>(s), <ea>(d)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

EXTSW — Sign Extend To Word — EXTSW

Operation: Sign-extends a byte source to a word EA destination.

Assembler Syntax:
```
EXTSW.B Rn(s), <ea>(e)
EXTSW.B <ea>(s), <ea>(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

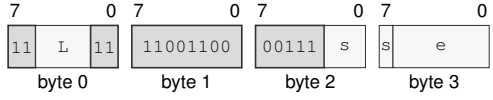
Detailed Semantics

Sign-extends the source operand from byte size to word size and stores the two-byte result to an EA destination. The low byte of the source is copied into the result, and the remaining high result bits are filled with the source sign bit. EXTSW has no Rn destination form; use EXTSQ.B for a sign-extended Rn result.

Encodings:

`EXTSW.B Rn(s), <ea>(e)`

Instruction Format:



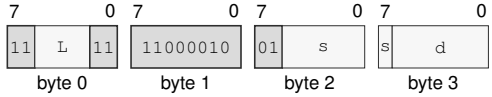
Format: Instruction format for EXTSW.B Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`EXTSW.B <ea>(s), <ea>(d)`

Instruction Format:



Format: Instruction format for EXTSW.B <ea>(s), <ea>(d)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

EXTZL — Zero Extend To Long — EXTZL

Operation: Zero-extends the selected source width to a longword EA destination.

Assembler Syntax:

```
EXTZL.B Rn(s), <ea>(e)
EXTZL.W Rn(s), <ea>(e)
EXTZL.{B|W}(z) <ea>(s), <ea>(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

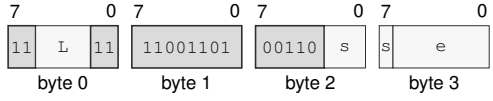
Detailed Semantics

Zero-extends the source operand from the instruction size to long size and stores the four-byte result to an EA destination. The low source-size bytes are copied into the result, and the remaining high result bits are filled with zero. EXTZL has no Rn destination form; use MOV.B or MOV.W for a zero-extended Rn result.

Encodings:

`EXTZL.B Rn(s), <ea>(e)`

Instruction Format:



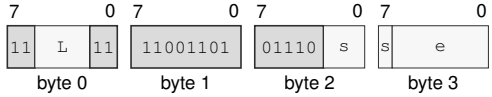
Format: Instruction format for EXTZL.B Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`EXTZL.W Rn(s), <ea>(e)`

Instruction Format:



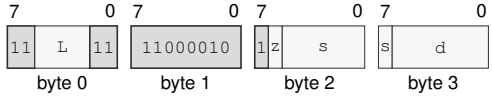
Format: Instruction format for EXTZL.W Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`EXTZL.{B|W}(z) <ea>(s), <ea>(d)`

Instruction Format:



Format: Instruction format for EXTZL.{B|W}(z) <ea>(s), <ea>(d)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

EXTZQ — Zero Extend To Quad — EXTZQ

Operation: Zero-extends the selected source width to a quadword destination.

Assembler Syntax:

```
EXTZQ.B Rn(s), <ea>(e)
EXTZQ.W Rn(s), <ea>(e)
EXTZQ.L Rn(s), <ea>(e)
EXTZQ.B <ea>(s), <ea>(d)
EXTZQ.W <ea>(s), <ea>(d)
EXTZQ.L <ea>(s), <ea>(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

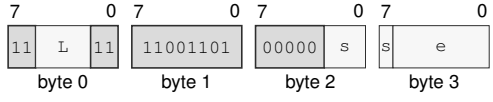
Detailed Semantics

Zero-extends the source operand from the instruction size to quad size and stores the eight-byte result to an EA destination. The low source-size bytes are copied into the result, and the remaining high result bits are filled with zero. EXTZQ has no Rn destination form; use MOV.B, MOV.W, or MOV.L for a zero-extended Rn result.

Encodings:

`EXTZQ.B Rn(s), <ea>(e)`

Instruction Format:



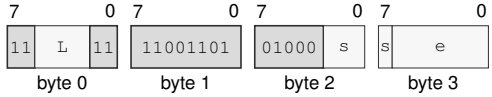
Format: Instruction format for EXTZQ.B Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`EXTZQ.W Rn(s), <ea>(e)`

Instruction Format:



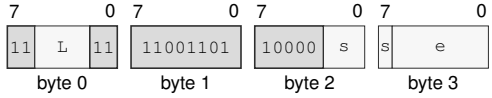
Format: Instruction format for EXTZQ.W Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`EXTZQ.L Rn(s), <ea>(e)`

Instruction Format:



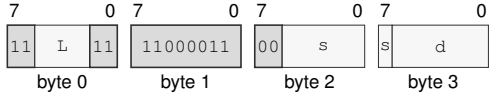
Format: Instruction format for EXTZQ.L Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`EXTZQ.B <ea>(s), <ea>(d)`

Instruction Format:



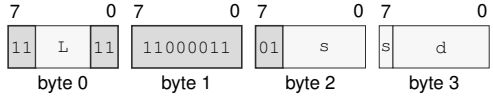
Format: Instruction format for EXTZQ.B <ea>(s), <ea>(d)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`EXTZQ.W <ea>(s), <ea>(d)`

Instruction Format:



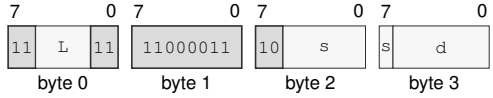
Format: Instruction format for EXTZQ.W <ea>(s), <ea>(d)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`EXTZQ.L <ea>(s), <ea>(d)`

Instruction Format:



Format: Instruction format for EXTZQ.L <ea>(s), <ea>(d)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

EXTZW — Zero Extend To Word — EXTZW

Operation: Zero-extends a byte source to a word EA destination.

Assembler Syntax:
```
EXTZW.B Rn(s), <ea>(e)
EXTZW.B <ea>(s), <ea>(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

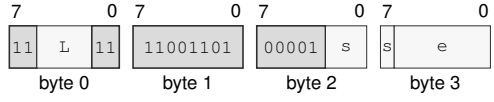
Detailed Semantics

Zero-extends the source operand from byte size to word size and stores the two-byte result to an EA destination. The low byte of the source is copied into the result, and the remaining high result bits are filled with zero. EXTZW has no Rn destination form; use MOV.B for a zero-extended Rn result.

Encodings:

`EXTZW.B Rn(s), <ea>(e)`

Instruction Format:



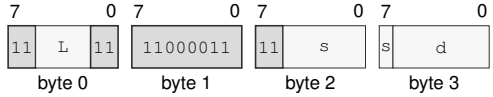
Format: Instruction format for EXTZW.B Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`EXTZW.B <ea>(s), <ea>(d)`

Instruction Format:



Format: Instruction format for EXTZW.B <ea>(s), <ea>(d)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

FETCHADD Atomic Fetch Add FETCHADD

Operation: Atomically adds source to memory and returns the old memory value in the source register.

Assembler Syntax: `FETCHADD.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Privilege: Unprivileged

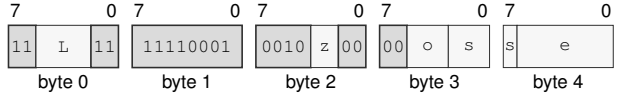
Detailed Semantics

Atomically replaces the memory operand with the selected-width sum of the old memory value and the source, and returns the old memory value in the source register.

Encodings:

`FETCHADD.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FETCHADD.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `order`; relation `allowed`; values `0x0..0x4`; reason `memory_order_reserved_reclaimed`.

Operand constraint: Role `memory`; relation `excluded`; values `immediate`; reason `memory_operand_required`.

FETCHAND Atomic Fetch And FETCHAND

Operation: Atomically ANDs source with memory and returns the old memory value in the source register.

Assembler Syntax: `FETCHAND.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Privilege: Unprivileged

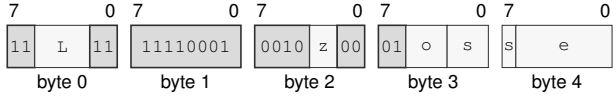
Detailed Semantics

Atomically replaces the memory operand with the selected-width bitwise conjunction of the old memory value and the source, and returns the old memory value in the source register.

Encodings:

`FETCHAND.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FETCHAND.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `order`; relation `allowed`; values `0x0..0x4`; reason `memory_order_reserved_reclaimed`.

Operand constraint: Role `memory`; relation `excluded`; values `immediate`; reason `memory_operand_required`.

FETCHANDN Atomic Fetch And Not FETCHANDN

Operation: Atomically clears memory bits selected by a source and returns the old value.

Assembler Syntax: `FETCHANDN.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Privilege: Unprivileged

Required CPUID flags: `FETCHANDN`

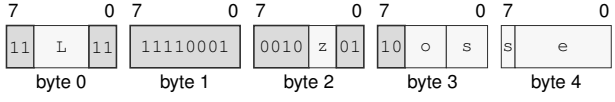
Detailed Semantics

Atomically replaces memory with $oldMemory \ \& \ \neg source$ and returns the old memory value in the source register. The encoded memory-order selector governs the complete operation.

Encodings:

`FETCHANDN.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FETCHANDN.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `order`; relation `allowed`; values `0x0..0x4`; reason `memory_order_reserved_reclaimed`.

Operand constraint: Role `memory`; relation `excluded`; values `immediate`; reason `memory_operand_required`.

FETCHMAXS Atomic Fetch Signed Maximum FETCHMAXS

Operation: Atomically stores the signed maximum of memory and source and returns the old value.

Assembler Syntax: `FETCHMAXS.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Privilege: Unprivileged

Required CPUID flags: `FETCHMAXS`

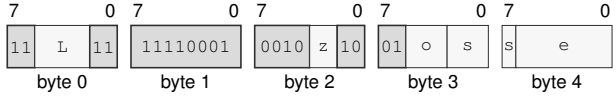
Detailed Semantics

Atomically compares memory and source as signed selected-width integers, stores the greater value, and returns the old memory value in the source register. The encoded memory-order selector governs the complete operation.

Encodings:

`FETCHMAXS.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FETCHMAXS.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `order`; relation `allowed`; values `0x0..0x4`; reason `memory_order_reserved_reclaimed`.
Operand constraint: Role `memory`; relation `excluded`; values `immediate`; reason `memory_operand_required`.

FETCHMAXU Atomic Fetch Unsigned Maximum FETCHMAXU

Operation: Atomically stores the unsigned maximum of memory and source and returns the old value.

Assembler Syntax: `FETCHMAXU.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Privilege: Unprivileged

Required CPUID flags: `FETCHMAXU`

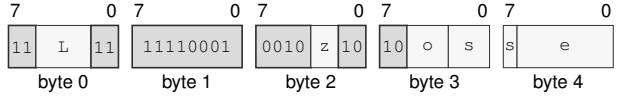
Detailed Semantics

Atomically compares memory and source as unsigned selected-width integers, stores the greater value, and returns the old memory value in the source register. The encoded memory-order selector governs the complete operation.

Encodings:

`FETCHMAXU.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FETCHMAXU.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `order`; relation `allowed`; values `0x0..0x4`; reason `memory_order_reserved_reclaimed`.

Operand constraint: Role `memory`; relation `excluded`; values `immediate`; reason `memory_operand_required`.

FETCHMINS Atomic Fetch Signed Minimum FETCHMINS

Operation: Atomically stores the signed minimum of memory and source and returns the old value.

Assembler Syntax: `FETCHMINS.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Privilege: Unprivileged

Required CPUID flags: `FETCHMINS`

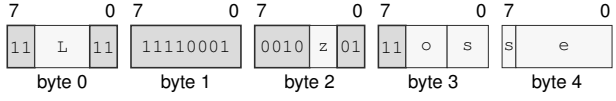
Detailed Semantics

Atomically compares memory and source as signed selected-width integers, stores the lesser value, and returns the old memory value in the source register. The encoded memory-order selector governs the complete operation.

Encodings:

`FETCHMINS.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FETCHMINS.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `order`; relation `allowed`; values `0x0..0x4`; reason `memory_order_reserved_reclaimed`.
Operand constraint: Role `memory`; relation `excluded`; values `immediate`; reason `memory_operand_required`.

FETCHMINU Atomic Fetch Unsigned Minimum FETCHMINU

Operation: Atomically stores the unsigned minimum of memory and source and returns the old value.

Assembler Syntax: `FETCHMINU.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Privilege: Unprivileged

Required CPUID flags: `FETCHMINU`

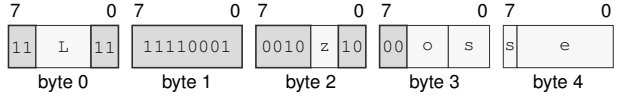
Detailed Semantics

Atomically compares memory and source as unsigned selected-width integers, stores the lesser value, and returns the old memory value in the source register. The encoded memory-order selector governs the complete operation.

Encodings:

`FETCHMINU.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FETCHMINU.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `order`; relation `allowed`; values `0x0..0x4`; reason `memory_order_reserved_reclaimed`.

Operand constraint: Role `memory`; relation `excluded`; values `immediate`; reason `memory_operand_required`.

FETCHOR — Atomic Fetch Or — FETCHOR

Operation: Atomically ORs source with memory and returns the old memory value in the source register.

Assembler Syntax: `FETCHOR.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Privilege: Unprivileged

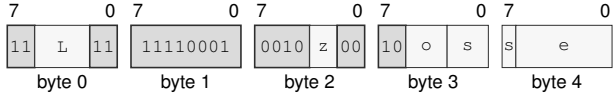
Detailed Semantics

Atomically replaces the memory operand with the selected-width bitwise disjunction of the old memory value and the source, and returns the old memory value in the source register.

Encodings:

`FETCHOR.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FETCHOR.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `order`; relation `allowed`; values `0x0..0x4`; reason `memory_order_reserved_reclaimed`.
Operand constraint: Role `memory`; relation `excluded`; values `immediate`; reason `memory_operand_required`.

FETCHSUB Atomic Fetch Subtract FETCHSUB

Operation: Atomically subtracts source from memory and returns the old memory value in the source register.

Assembler Syntax: `FETCHSUB.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Privilege: Unprivileged

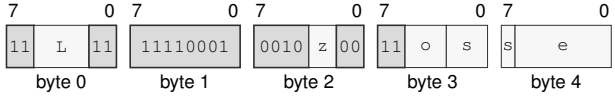
Detailed Semantics

Atomically replaces the memory operand with the selected-width result of old memory minus the source, and returns the old memory value in the source register.

Encodings:

`FETCHSUB.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FETCHSUB.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `order`; relation `allowed`; values `0x0..0x4`; reason `memory_order_reserved_reclaimed`.

Operand constraint: Role `memory`; relation `excluded`; values `immediate`; reason `memory_operand_required`.

FETCHXCHG — Atomic Fetch Exchange — FETCHXCHG

Operation: Atomically exchanges a register with memory and returns the old memory value.

Assembler Syntax: `FETCHXCHG.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Privilege: Unprivileged

Required CPUID flags: `FETCHXCHG`

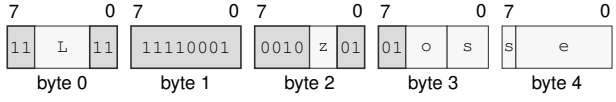
Detailed Semantics

Atomically replaces the selected-width memory value with the original source-register value and returns the old memory value in that register. The encoded memory-order selector governs the complete read-modify-write operation. Unlike the memory form of `XCHG`, this instruction is one atomic memory operation.

Encodings:

`FETCHXCHG.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FETCHXCHG.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `order`; relation `allowed`; values `0x0..0x4`; reason `memory_order_reserved_reclaimed`.
Operand constraint: Role `memory`; relation `excluded`; values `immediate`; reason `memory_operand_required`.

FETCHXOR — Atomic Fetch Xor — FETCHXOR

Operation: Atomically XORs source with memory and returns the old memory value in the source register.

Assembler Syntax: `FETCHXOR.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Privilege: Unprivileged

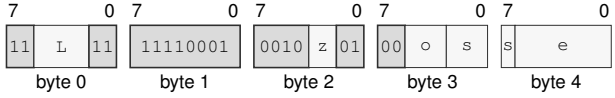
Detailed Semantics

Atomically replaces the memory operand with the selected-width bitwise exclusive-or of the old memory value and the source, and returns the old memory value in the source register.

Encodings:

`FETCHXOR.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FETCHXOR.{B|W|L|Q}(z)/order(o) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `order`; relation `allowed`; values `0x0..0x4`; reason `memory_order_reserved_reclaimed`.

Operand constraint: Role `memory`; relation `excluded`; values `immediate`; reason `memory_operand_required`.

FLSHDCACHE Flush Data Cache FLSHDCACHE

Operation: Write back and invalidate one cache-maintenance block.

Assembler Syntax: `FLSHDCACHE <ea>(e)`

Privilege: Supervisor only

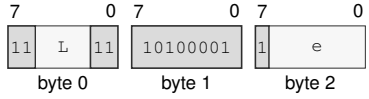
Detailed Semantics

Computes and translates the address-role EA without reading an operand value, selects its CPUID-sized maintenance block, writes dirty bytes from every data or unified cache copy in the coherence domain into normal-memory coherence, invalidates those copies, and retires after the block is complete.

Encodings:

`FLSHDCACHE <ea>(e)`

Instruction Format:



Format: Instruction format for FLSHDCACHE <ea>(e)

HALT Halt HALT

Operation: Halt the executing logical processor until an asynchronous architectural event is admitted.

Assembler Syntax: `HALT`

Privilege: Supervisor only

Detailed Semantics

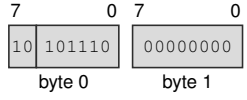
HALT retires with the following instruction as its continuation, then applies the Architectural Event Processing Model's event priority and admission rules at that instruction boundary. If the model selects an event, it delivers that event with the continuation as its saved PC before entering the halted state.

Otherwise, the executing logical processor stops instruction fetch. When an asynchronous architectural event later becomes admissible, the logical processor leaves the halted state by delivering that event with the continuation as its saved PC. Event delivery precedes execution of the continuation. Only the executing logical processor enters the halted state.

Encodings:

`HALT`

Instruction Format:



Format: Instruction format for HALT

IJcc — Increment And Jump Conditional — IJcc

Operation: Increments the index and branches only when it differs from the bound and the condition is true.

Assembler Syntax:
```
IJcc Rn(i), Rn(b), <ea>(e)
IJcc Rn(i), Rn(b), Rn(e)
```

Privilege: Unprivileged

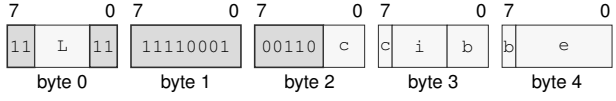
Detailed Semantics

Increments the index and transfers control only when the new index differs from the bound and the encoded condition is true. The target EA has Q interpretation width: a memory form reads a 64-bit target and a register or immediate form supplies its value directly.

Encodings:

`IJcc Rn(i), Rn(b), <ea>(e)`

Instruction Format:



Format: Instruction format for IJcc Rn(i), Rn(b), <ea>(e)

Restrictions:

Operand constraint: Role `cc`; relation `allowed`; values `0, 0x2..0xf`; reason `condition_false_reclaimed`.

`IJcc Rn(i), Rn(b), Rn(e)`

Instruction Format:

Format: Instruction format for IJcc Rn(i), Rn(b), Rn(e)

Restrictions:

Operand constraint: Role `cc`; relation `allowed`; values `0, 0x2..0xf`; reason `condition_false_reclaimed`.

ILLEGAL — Illegal Instruction — ILLEGAL

Operation: Raise illegal-instruction exception.

Assembler Syntax: `ILLEGAL`

Privilege: Unprivileged

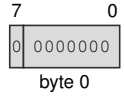
Detailed Semantics

Raises EXPLICIT_ILLEGAL_INSTRUCTION at the current instruction boundary without retiring or changing architectural destination state before event entry.

Encodings:

`ILLEGAL`

Instruction Format:



Format: Instruction format for ILLEGAL

INC — Increment — INC

Operation: Increments the selected-width destination modulo its width without changing FLAGS.

Assembler Syntax:
```
INC.{L|Q}(z) Rn(r)
INC.{B|W|L|Q}(z) <ea>(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

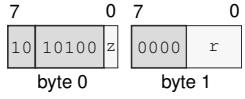
Detailed Semantics

Adds one to the selected-width destination and writes the modular result without changing FLAGS.

Encodings:

`INC.{L|Q}(z) Rn(r)`

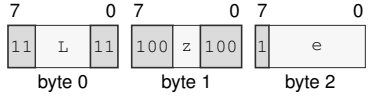
Instruction Format:



Format: Instruction format for INC.{L|Q}(z) Rn(r)

`INC.{B|W|L|Q}(z) <ea>(e)`

Instruction Format:



Format: Instruction format for INC.{B|W|L|Q}(z) <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

INCF — Increment And Set Flags — INCF

Operation: Increments the destination operand and updates integer FLAGS.

Assembler Syntax: `INCF.{B|W|L|Q}(z) Rn(r)`

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

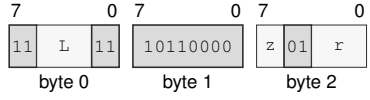
Detailed Semantics

Adds one to the selected-width destination, writes the modular result, and updates `FLAGS.Z`, `FLAGS.N`, `FLAGS.C`, and `FLAGS.V`. The written destination result supplies the REPcc observed value.

Encodings:

`INCF.{B|W|L|Q}(z) Rn(r)`

Instruction Format:



Format: Instruction format for INCF.{B|W|L|Q}(z) Rn(r)

INVASID — Invalidate TLB By ASID — INVASID

Operation: Invalidate TLB entries for ASID.

Assembler Syntax: `INVASID <imm16>`

Privilege: Supervisor only

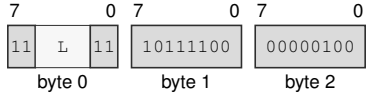
Detailed Semantics

Invalidates every non-global cached translation for the selected 16-bit ASID on the current logical processor, discards matching in-progress results, and prevents later use of stale entries.

Encodings:

`INVASID <imm16>`

Instruction Format:



Format: Instruction format for INVASID <imm16>

INVDCACHE Invalidate Data Cache INVDCACHE

Operation: Invalidate one data-cache maintenance block.

Assembler Syntax: `INVDCACHE <ea>(e)`

Privilege: Supervisor only

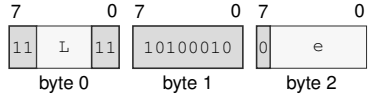
Detailed Semantics

Computes and translates the address-role EA without reading an operand value, selects its CPUID-sized maintenance block, discards every data or unified cache copy in the coherence domain without writeback, and retires after the block is complete.

Encodings:

`INVDCACHE <ea>(e)`

Instruction Format:



Format: Instruction format for INVDCACHE <ea>(e)

INVICACHE — Invalidate Instruction Cache — INVICACHE

Operation: Invalidate one local instruction-cache maintenance block.

Assembler Syntax: `INVICACHE <ea>(e)`

Privilege: Supervisor only

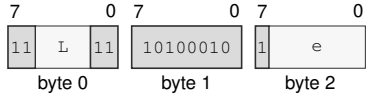
Detailed Semantics

Computes and translates the address-role EA without fetching from it, selects its CPUID-sized maintenance block, invalidates the executing logical processor's matching instruction-cache, decoded, and prefetch state, and serializes later instruction fetch.

Encodings:

`INVICACHE <ea>(e)`

Instruction Format:



Format: Instruction format for INVICACHE <ea>(e)

INVPAGE — Invalidate TLB Page — INVPAGE

Operation: Invalidate TLB mapping containing a linear address.

Assembler Syntax:
```
INVPAGE <ea>(e)
INVPAGE Rn(r)
```

Privilege: Supervisor only

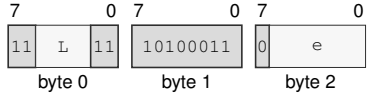
Detailed Semantics

Computes the source linear address without reading memory and invalidates every matching global or current-ASID cached leaf-aligned mapping range that contains it, plus matching in-progress results, before retirement.

Encodings:

`INVPAGE <ea>(e)`

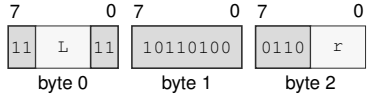
Instruction Format:



Format: Instruction format for INVPAGE <ea>(e)

`INVPAGE Rn(r)`

Instruction Format:



Format: Instruction format for INVPAGE Rn(r)

INVTLB — Invalidate TLB — INVTLB

Operation: Invalidate all TLB entries.

Assembler Syntax: `INVTLB`

Privilege: Supervisor only

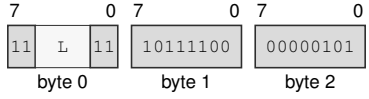
Detailed Semantics

Invalidates all cached instruction and data translations and discards in-progress results on the current logical processor before later accesses may proceed.

Encodings:

`INVTLB`

Instruction Format:



Format: Instruction format for INVTLB

Jcc — Jump Conditional — Jcc

Operation: Transfers control when the encoded condition is true.

Assembler Syntax:

```
Jcc imm8s(i)
Jcc <imm16s>
Jcc <imm32s>
Jcc.{L|Q}(z) <ea>(e)
Jcc.{L|Q}(z) Rn(r)
```

Privilege: Unprivileged

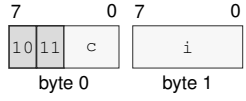
Detailed Semantics

Transfers execution to the target when the encoded condition is true and otherwise continues at the next instruction. Compact direct-relative encodings select a W- or L-width displacement. Extended EA encodings use L/Q. Jcc.L reads the low 32-bit target from its register or memory EA and zero-extends it to the 64-bit near target; Jcc.Q uses the complete 64-bit target.

Encodings:

`Jcc imm8s(i)`

Instruction Format:



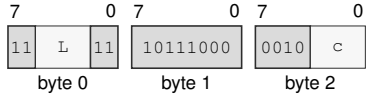
Format: Instruction format for Jcc imm8s(i)

Restrictions:

Operand constraint: Role `cc`; relation `allowed`; values `0x2..0xf`; reason `condition_true_false_reclaimed`.

`Jcc <imm16s>`

Instruction Format:



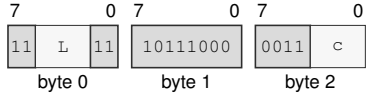
Format: Instruction format for Jcc <imm16s>

Restrictions:

Operand constraint: Role `cc`; relation `allowed`; values `0x2..0xf`; reason `condition_true_false_reclaimed`.

`Jcc <imm32s>`

Instruction Format:



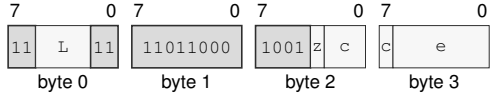
Format: Instruction format for Jcc <imm32s>

Restrictions:

Operand constraint: Role `cc`; relation `allowed`; values `0x2..0xf`; reason `condition_true_false_reclaimed`.

`Jcc.{L|Q}(z) <ea>(e)`

Instruction Format:



Format: Instruction format for Jcc.{L|Q}(z) <ea>(e)

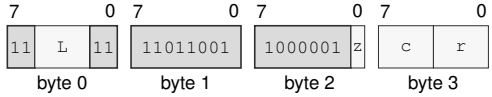
Restrictions:

Operand constraint: Role `cc`; relation `allowed`; values `0x2..0xf`; reason `condition_true_false_reclaimed`.

Operand constraint: Role `target`; relation `excluded`; values `immediate`; reason `canonical_form_reclaim`.

`Jcc.{L|Q}(z) Rn(r)`

Instruction Format:



Format: Instruction format for Jcc.{L|Q}(z) Rn(r)

Restrictions:

Operand constraint: Role `cc`; relation `allowed`; values `0x2..0xf`; reason `condition_true_false_reclaimed`.

JMP — Jump — JMP

Operation: Transfers control to the target address.

Assembler Syntax:

```
JMP imm8s(i)
JMP <imm16s>
JMP <imm32s>
JMP.{L|Q}(z) <ea>(e)
JMP.{L|Q}(z) Rn(r)
```

Privilege: Unprivileged

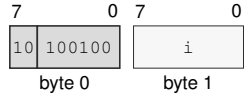
Detailed Semantics

Validates the target under the current control context and transfers execution to it. For the extended EA encoding, JMP.L reads the low 32-bit target from its register or memory EA and zero-extends it to the 64-bit near target; JMP.Q uses the complete 64-bit target.

Encodings:

`JMP imm8s(i)`

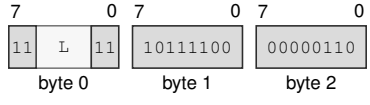
Instruction Format:



Format: Instruction format for JMP imm8s(i)

`JMP <imm16s>`

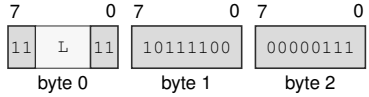
Instruction Format:



Format: Instruction format for JMP <imm16s>

`JMP <imm32s>`

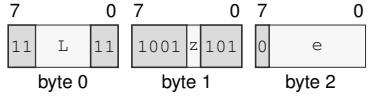
Instruction Format:



Format: Instruction format for JMP <imm32s>

`JMP.{L|Q}(z) <ea>(e)`

Instruction Format:



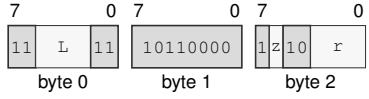
Format: Instruction format for JMP.{L|Q}(z) <ea>(e)

Restrictions:

Operand constraint: Role `target`; relation `excluded`; values `immediate`; reason `canonical_form_reclaim`.

`JMP.{L|Q}(z) Rn(r)`

Instruction Format:



Format: Instruction format for JMP.{L|Q}(z) Rn(r)

LCALL — Long Call — LCALL

Operation: Pushes a long return frame and atomically transfers control to a new CS:PC pair.

Assembler Syntax:

```
LCALL Rn(r), <ea>(e)
LCALL Rn(s), Rn(d)
```

Privilege: Unprivileged

Detailed Semantics

Long Call is a segment-changing control transfer. It first constructs the long return frame on SS:SP, then commits the new CS and PC together. The two-operand encoding takes the new CS image from an Rn register. Its target EA has Q interpretation width: a memory operand reads a 64-bit offset and a register or immediate operand supplies its value directly.

1. Require LPA.A to be zero, then evaluate the target operand completely in the old architectural context.
2. Validate the complete new CS image.
3. Validate the target offset against the new CS and probe executable translation at the resulting linear address.
4. Probe both 8-byte long-return-frame stores through the old SS context.
5. Commit the two frame stores, decremented SP, new CS, and new PC as one successful control-transfer operation.

Fault atomicity. No frame word, SP, CS, or PC update is visible when any prior step faults.

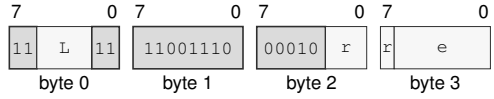
An active resident link raises `INVALID_CONTROL_TRANSITION` before target or stack access.

The long return frame is 16 bytes in the old SS context: the continuation PC occupies offset 0 and the return CS image occupies offset 8. An invalid new CS image raises `INVALID_CONTROL_IMAGE`; target translation or either frame-store translation failure raises the applicable `ADDRESS_TRANSLATION` family event.

Encodings:

`LCALL Rn(r), <ea>(e)`

Instruction Format:



Format: Instruction format for LCALL Rn(r), <ea>(e)

`LCALL Rn(s), Rn(d)`

Instruction Format:

Format: Instruction format for LCALL Rn(s), Rn(d)

LDP — Load Pair — LDP

Operation: Loads one 128-bit memory value into a canonical register pair.

Assembler Syntax: `LDP <ea>(e), PAIRn(d)`

Privilege: Unprivileged

Required CPUID flags: `LDP`

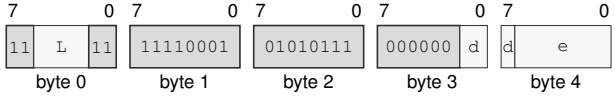
Detailed Semantics

Loads one naturally aligned 16-byte memory value into the selected canonical register pair. Bits 63:0 go to the even register and bits 127:64 go to the following odd register. Both registers commit together; a preceding alignment or memory-access fault changes neither register.

Encodings:

`LDP <ea>(e), PAIRn(d)`

Instruction Format:



Format: Instruction format for LDP <ea>(e), PAIRn(d)

Restrictions:

Operand constraint: Role `memory`; relation `excluded`; values `immediate`; reason `memory_operand_required`.

LEA — Load Effective Address — LEA

Operation: Computes the source effective address without reading memory and writes it to the destination.

Assembler Syntax:
```
LEA.{B|W|L|Q}(z) <ea>(e), Rn(d)
LEA.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

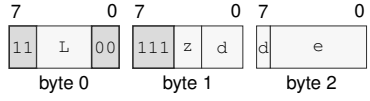
Detailed Semantics

Computes the source effective address without reading the addressed memory and writes that address to the destination.

Encodings:

`LEA.{B|W|L|Q}(z) <ea>(e), Rn(d)`

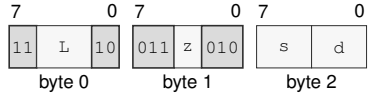
Instruction Format:



Format: Instruction format for LEA.{B|W|L|Q}(z) <ea>(e), Rn(d)

`LEA.{B|W|L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for LEA.{B|W|L|Q}(z) Rn(s), Rn(d)

LJMP — Long Jump — LJMP

Operation: Atomically transfers control to a new CS:PC pair.

Assembler Syntax:
```
LJMP Rn(r), <ea>(e)
LJMP Rn(s), Rn(d)
```

Privilege: Unprivileged

Detailed Semantics

Long Jump is a segment-changing control transfer. It commits the new CS and PC as a single architectural control-transfer step. The two-operand encoding takes the new CS image from an Rn register. Its target EA has Q interpretation width: a memory operand reads a 64-bit offset and a register or immediate operand supplies its value directly.

1. Evaluate the target operand completely in the old architectural context.
2. Validate the complete new CS image.
3. Validate the target offset against the new CS and probe executable translation at the resulting linear address.
4. Commit new CS and new PC as one successful control-transfer operation.

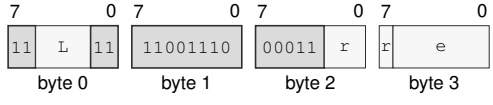
Fault atomicity. CS and PC remain unchanged when operand evaluation or target validation faults.

An invalid new CS image raises `INVALID_CONTROL_IMAGE`; target translation failure raises the applicable `ADDRESS_TRANSLATION` family event.

Encodings:

`LJMP Rn(r), <ea>(e)`

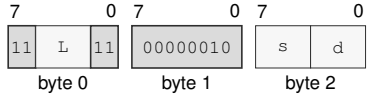
Instruction Format:



Format: Instruction format for LJMP Rn(r), <ea>(e)

`LJMP Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for LJMP Rn(s), Rn(d)

LRET — Long Return — LRET

Operation: Pops a long return frame and atomically restores CS:PC.

Assembler Syntax: `LRET`

Privilege: Unprivileged

Detailed Semantics

Long Return is a segment-changing control transfer that requires LPA.A to be zero, reads the long return frame from SS:SP, and commits CS and PC together. An active resident link raises `INVALID_CONTROL_TRANSITION` before stack access. The frame contains the continuation PC Q value at the current SP and the return CS Q value at SP+8; after both values are fetched, SP is advanced by 16 bytes and the control state is committed.

1. Read both 8-byte long-return-frame words through the old SS context without changing SP.
2. Validate the complete return CS image.
3. Validate the continuation PC offset against the return CS and probe executable translation at the resulting linear address.
4. Commit advanced SP, return CS, and continuation PC as one successful control-transfer operation.

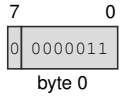
Fault atomicity. SP, CS, and PC remain unchanged when a frame read or return-target validation faults.

The 16-byte long return frame is read through the old SS context: the continuation PC occupies offset 0 and the return CS image occupies offset 8. A frame translation or return-target translation failure raises the applicable `ADDRESS_TRANSLATION` family event; an invalid return CS image raises `INVALID_CONTROL_IMAGE`.

Encodings:

LRET

Instruction Format:



Format: Instruction format for LRET

MADD — Multiply Add — MADD

Operation: Adds the low product of two sources to the destination.

Assembler Syntax: `MADD.{B|W|L|Q}(z) Rn(l), Rn(r), Rn(d)`

Privilege: Unprivileged

Required CPUID flags: `MADD`

Repeat eligibility: REP eligible; REPcc observes `dst`

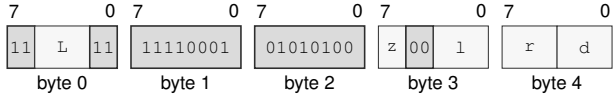
Detailed Semantics

Multiplies the selected-width source values, adds the old destination value, and writes the low selected-width result. The instruction does not update flags.

Encodings:

`MADD.{B|W|L|Q}(z) Rn(l), Rn(r), Rn(d)`

Instruction Format:



Format: Instruction format for MADD.{B|W|L|Q}(z) Rn(l), Rn(r), Rn(d)

MAXS — Signed Maximum — MAXS

Operation: Writes the signed greater of source and destination to the destination.

Assembler Syntax:
```
MAXS.{B|W|L|Q}(z) <ea>(e), Rn(d)
MAXS.{B|W|L|Q}(z) Rn(s), <ea>(e)
MAXS.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

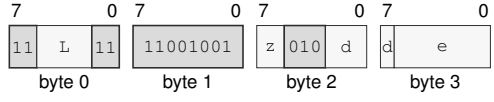
Detailed Semantics

Compares the selected-width operands as signed integers and writes the greater value to the destination. A sub-Q Rn result is sign-extended to 64 bits.

Encodings:

`MAXS.{B|W|L|Q}(z) <ea>(e), Rn(d)`

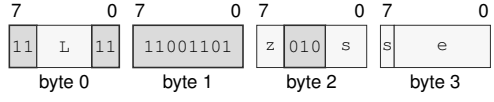
Instruction Format:



Format: Instruction format for MAXS.{B|W|L|Q}(z) <ea>(e), Rn(d)

`MAXS.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



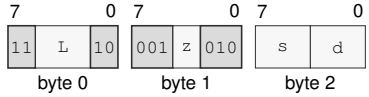
Format: Instruction format for MAXS.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`MAXS.{B|W|L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for MAXS.{B|W|L|Q}(z) Rn(s), Rn(d)

MAXU — Unsigned Maximum — MAXU

Operation: Writes the unsigned greater of source and destination to the destination.

Assembler Syntax:
```
MAXU.{B|W|L|Q}(z) <ea>(e), Rn(d)
MAXU.{B|W|L|Q}(z) Rn(s), <ea>(e)
MAXU.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

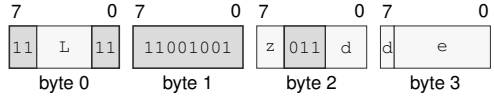
Detailed Semantics

Compares the selected-width operands as unsigned integers and writes the greater value to the destination.

Encodings:

`MAXU.{B|W|L|Q}(z) <ea>(e), Rn(d)`

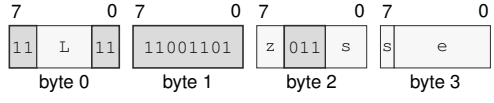
Instruction Format:



Format: Instruction format for MAXU.{B|W|L|Q}(z) <ea>(e), Rn(d)

`MAXU.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



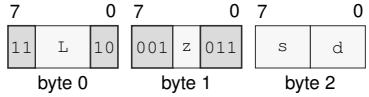
Format: Instruction format for MAXU.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`MAXU.{B|W|L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for MAXU.{B|W|L|Q}(z) Rn(s), Rn(d)

MINS — Signed Minimum — MINS

Operation: Writes the signed lesser of source and destination to the destination.

Assembler Syntax:
```
MINS.{B|W|L|Q}(z) <ea>(e), Rn(d)
MINS.{B|W|L|Q}(z) Rn(s), <ea>(e)
MINS.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

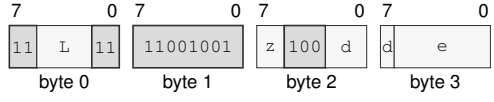
Detailed Semantics

Compares the selected-width operands as signed integers and writes the lesser value to the destination. A sub-Q Rn result is sign-extended to 64 bits.

Encodings:

`MINS.{B|W|L|Q}(z) <ea>(e), Rn(d)`

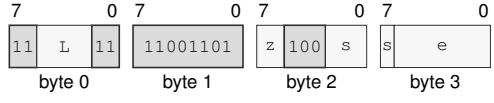
Instruction Format:



Format: Instruction format for MINS.{B|W|L|Q}(z) <ea>(e), Rn(d)

`MINS.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



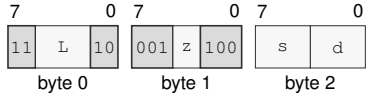
Format: Instruction format for MINS.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`MINS.{B|W|L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for MINS.{B|W|L|Q}(z) Rn(s), Rn(d)

MINU — Unsigned Minimum — MINU

Operation: Writes the unsigned lesser of source and destination to the destination.

Assembler Syntax:

```
MINU.{B|W|L|Q}(z) <ea>(e), Rn(d)
MINU.{B|W|L|Q}(z) Rn(s), <ea>(e)
MINU.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

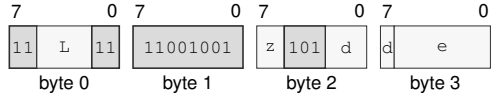
Detailed Semantics

Compares the selected-width operands as unsigned integers and writes the lesser value to the destination.

Encodings:

`MINU.{B|W|L|Q}(z) <ea>(e), Rn(d)`

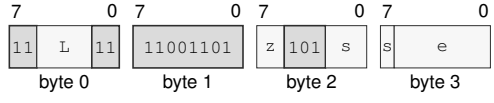
Instruction Format:



Format: Instruction format for MINU.{B|W|L|Q}(z) <ea>(e), Rn(d)

`MINU.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



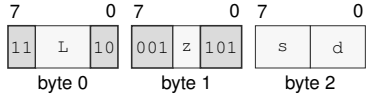
Format: Instruction format for MINU.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`MINU.{B|W|L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for MINU.{B|W|L|Q}(z) Rn(s), Rn(d)

MODS — Signed Modulo — MODS

Operation: Writes the signed remainder of destination-by-source truncation-toward-zero division.

Assembler Syntax:
```
MODS.{B|W|L|Q}(z) <ea>(e), Rn(d)
MODS.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

Detailed Semantics

Interprets both selected-width operands as signed integers and writes the remainder associated with truncation-toward-zero division. A sub-Q remainder is sign-extended to 64 bits. A zero divisor raises DIVIDE_BY_ZERO; the most-negative value divided by minus one raises SIGNED_DIVIDE_OVERFLOW.

Let n be the selected operand width, d the signed two's-complement dividend, and s the signed two's-complement divisor. The quotient and remainder are defined by

$$q = \mathrm{trunc}(d/s), \qquad r = d - qs.$$

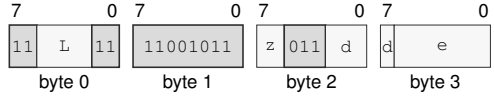
Thus a nonzero remainder has the sign of the dividend. The destination supplies the dividend and receives r.

A zero divisor raises `DIVIDE_BY_ZERO`; the most-negative value divided by -1 raises `SIGNED_DIVIDE_OVERFLOW`. No destination is changed.

Encodings:

`MODS.{B|W|L|Q}(z) <ea>(e), Rn(d)`

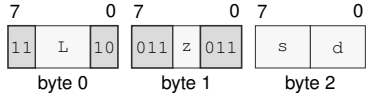
Instruction Format:



Format: Instruction format for MODS.{B|W|L|Q}(z) <ea>(e), Rn(d)

`MODS.{B|W|L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for MODS.{B|W|L|Q}(z) Rn(s), Rn(d)

MODU — Unsigned Modulo — MODU

Operation: Writes the unsigned remainder of destination-by-source division.

Assembler Syntax:
```
MODU.{B|W|L|Q}(z) <ea>(e), Rn(d)
MODU.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

Detailed Semantics

Writes the remainder of selected-width unsigned destination-by-source division. A zero divisor raises DIVIDE_BY_ZERO.

Let n be the selected operand width, d the unsigned dividend, and s the unsigned divisor. The quotient and remainder are defined by

$$q = \lfloor d/s \rfloor, \qquad r = d - qs.$$

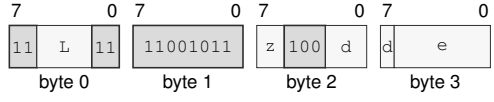
For a nonzero divisor, the remainder satisfies $0 \leq r < s$. The destination supplies the dividend and receives r.

A zero divisor raises `DIVIDE_BY_ZERO`; no destination is changed.

Encodings:

`MODU.{B|W|L|Q}(z) <ea>(e), Rn(d)`

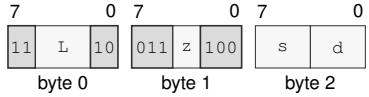
Instruction Format:



Format: Instruction format for MODU.{B|W|L|Q}(z) <ea>(e), Rn(d)

`MODU.{B|W|L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for MODU.{B|W|L|Q}(z) Rn(s), Rn(d)

MOV — Move — MOV

Operation: Copies the source operand to the destination operand.

Assembler Syntax:

```
MOV.Q Rn(r), SP
MOV.Q SP, Rn(r)
MOV.{L|Q}(z) Rn(s), Rn(d)
MOV.B Rn(s), Rn(d)
MOV.W Rn(s), Rn(d)
MOV.{B|W|L|Q}(z) Rn(s), <ea>(e)
MOV.{B|W|L|Q}(z) <ea>(e), Rn(d)
MOV.{B|W|L|Q}(z) <ea>(s), <ea>(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

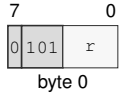
Detailed Semantics

Reads the source using its addressing mode and selected size and writes that value to the destination. A sub-Q Rn destination receives the selected source bits zero-extended to 64 bits.

Encodings:

`MOV.Q Rn(r), SP`

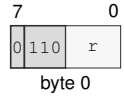
Instruction Format:



Format: Instruction format for MOV.Q Rn(r), SP

`MOV.Q SP, Rn(r)`

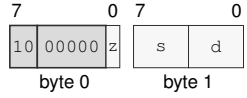
Instruction Format:



Format: Instruction format for MOV.Q SP, Rn(r)

`MOV.{L|Q}(z) Rn(s), Rn(d)`

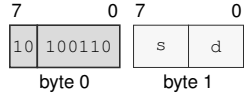
Instruction Format:



Format: Instruction format for MOV.{L|Q}(z) Rn(s), Rn(d)

`MOV.B Rn(s), Rn(d)`

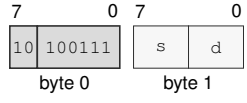
Instruction Format:



Format: Instruction format for MOV.B Rn(s), Rn(d)

`MOV.W Rn(s), Rn(d)`

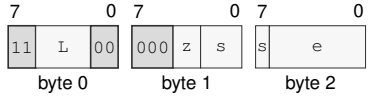
Instruction Format:



Format: Instruction format for MOV.W Rn(s), Rn(d)

`MOV.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



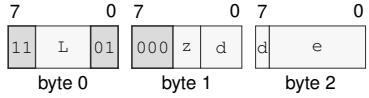
Format: Instruction format for MOV.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`MOV.{B|W|L|Q}(z) <ea>(e), Rn(d)`

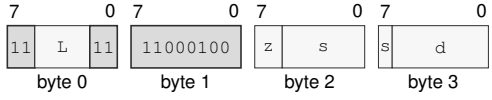
Instruction Format:



Format: Instruction format for MOV.{B|W|L|Q}(z) <ea>(e), Rn(d)

`MOV.{B|W|L|Q}(z) <ea>(s), <ea>(d)`

Instruction Format:



Format: Instruction format for MOV.{B|W|L|Q}(z) <ea>(s), <ea>(d)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

MOVACQ — Move Acquire — MOVACQ

Operation: Loads from memory with acquire ordering.

Assembler Syntax: `MOVACQ.{B|W|L|Q}(z) <ea>(e), Rn(d)`

Privilege: Unprivileged

Required CPUID flags: `MOVACQ`

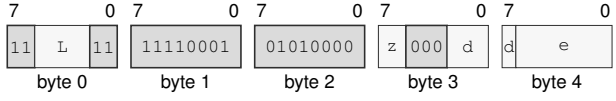
Detailed Semantics

Loads the selected-width memory value into the destination register. The load has acquire ordering: later memory operations by the same logical processor are ordered after it. A sub-Q destination is zero-extended to 64 bits. The instruction does not impose release or global sequentially consistent ordering.

Encodings:

`MOVACQ.{B|W|L|Q}(z) <ea>(e), Rn(d)`

Instruction Format:



Format: Instruction format for MOVACQ.{B|W|L|Q}(z) <ea>(e), Rn(d)

Restrictions:

Operand constraint: Role `memory`; relation `excluded`; values `immediate`; reason `memory_operand_required`.

MOVcc — Move Conditional — MOVcc

Operation: Copies source to destination only when the encoded condition is true.

Assembler Syntax:

```
MOVcc.{B|W|L|Q}(z) Rn(s), <ea>(e)
MOVcc.{B|W|L|Q}(z) <ea>(e), Rn(d)
MOVcc.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible

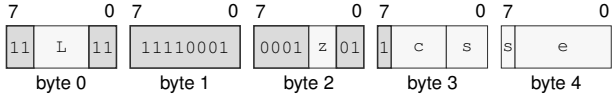
Detailed Semantics

Copies the source value to the destination only when the encoded condition is true. When the condition is true, a sub-Q Rn destination receives the selected source bits zero-extended to 64 bits.

Encodings:

`MOVcc.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



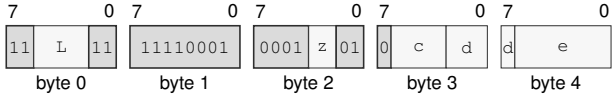
Format: Instruction format for MOVcc.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`MOVcc.{B|W|L|Q}(z) <ea>(e), Rn(d)`

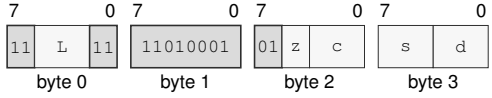
Instruction Format:



Format: Instruction format for MOVcc.{B|W|L|Q}(z) <ea>(e), Rn(d)

`MOVcc.{B|W|L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for MOVcc.{B|W|L|Q}(z) Rn(s), Rn(d)

MOVCU — Move Current To User — MOVCU

Operation: Copies from a current-domain source to user-domain memory.

Assembler Syntax:

```
MOVCU.{B|W|L|Q}(z) <ea>(s), <ea>(d)
MOVCU.{B|W|L|Q}(z) Rn(s), Rn(d)
MOVCU.{B|W|L|Q}(z) Rn(s), <ea>(d)
MOVCU.{B|W|L|Q}(z) <ea>(s), Rn(d)
```

Privilege: Supervisor only

Repeat eligibility: REP eligible; REPcc observes `dst`

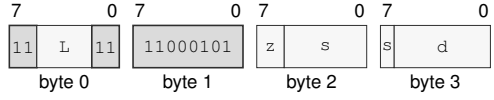
Detailed Semantics

MOVCU is a supervisor-only checked user-access move. The source is a current-domain memory operand, register, or immediate, and the destination is always user-domain memory.

Encodings:

`MOVCU.{B|W|L|Q}(z) <ea>(s), <ea>(d)`

Instruction Format:



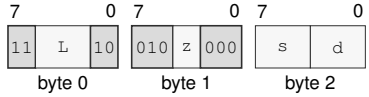
Format: Instruction format for MOVCU.{B|W|L|Q}(z) <ea>(s), <ea>(d)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`MOVCU.{B|W|L|Q}(z) Rn(s), Rn(d)`

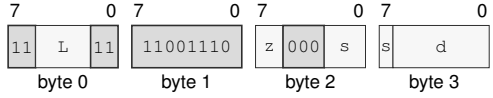
Instruction Format:



Format: Instruction format for MOVCU.{B|W|L|Q}(z) Rn(s), Rn(d)

`MOVCU.{B|W|L|Q}(z) Rn(s), <ea>(d)`

Instruction Format:



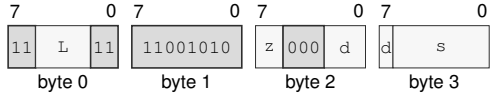
Format: Instruction format for MOVCU.{B|W|L|Q}(z) Rn(s), <ea>(d)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`MOVCU.{B|W|L|Q}(z) <ea>(s), Rn(d)`

Instruction Format:



Format: Instruction format for MOVCU.{B|W|L|Q}(z) <ea>(s), Rn(d)

MOVNT Move Non-Temporal MOVNT

Operation: Stores an Rn register value to memory with a non-temporal access hint.

Assembler Syntax: `MOVNT.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `src`

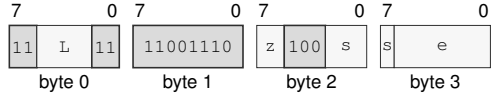
Detailed Semantics

MOVNT is a store-only non-temporal move. The source is an Rn register and the destination must be a memory operand.

Encodings:

`MOVNT.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for MOVNT.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

MOVREL — Move Release — MOVREL

Operation: Stores to memory with release ordering.

Assembler Syntax: `MOVREL.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Privilege: Unprivileged

Required CPUID flags: `MOVREL`

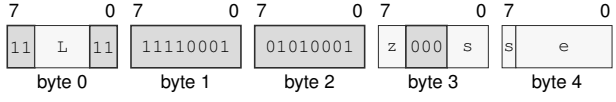
Detailed Semantics

Stores the selected-width source register value to memory. The store has release ordering: earlier memory operations by the same logical processor are ordered before it. The instruction does not impose acquire or global sequentially consistent ordering.

Encodings:

`MOVREL.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for MOVREL.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `memory`; relation `excluded`; values `immediate`; reason `memory_operand_required`.

MOVUC — Move User To Current — MOVUC

Operation: Copies from user-domain memory to a current-domain destination.

Assembler Syntax:

```
MOVUC.{B|W|L|Q}(z) <ea>(s), <ea>(d)
MOVUC.{B|W|L|Q}(z) Rn(s), Rn(d)
MOVUC.{B|W|L|Q}(z) Rn(s), <ea>(d)
MOVUC.{B|W|L|Q}(z) <ea>(s), Rn(d)
```

Privilege: Supervisor only

Repeat eligibility: REP eligible; REPcc observes `dst`

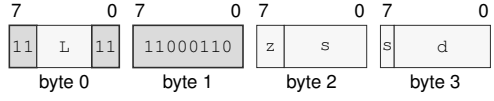
Detailed Semantics

MOVUC is a supervisor-only checked user-access move. If the source operand is memory, that access uses the user domain. The destination is a current-domain memory operand or an Rn register.

Encodings:

`MOVUC.{B|W|L|Q}(z) <ea>(s), <ea>(d)`

Instruction Format:



Format: Instruction format for MOVUC.{B|W|L|Q}(z) <ea>(s), <ea>(d)

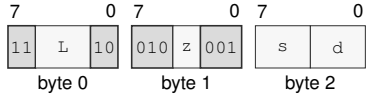
Restrictions:

Operand constraint: Role `src`; relation `excluded`; values `immediate`; reason `user_source_memory_required`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`MOVUC.{B|W|L|Q}(z) Rn(s), Rn(d)`

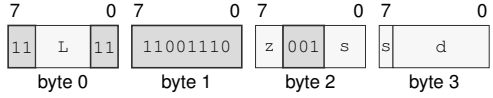
Instruction Format:



Format: Instruction format for MOVUC.{B|W|L|Q}(z) Rn(s), Rn(d)

`MOVUC.{B|W|L|Q}(z) Rn(s), <ea>(d)`

Instruction Format:



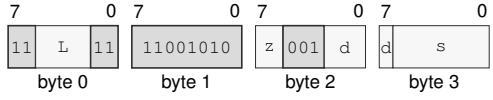
Format: Instruction format for MOVUC.{B|W|L|Q}(z) Rn(s), <ea>(d)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`MOVUC.{B|W|L|Q}(z) <ea>(s), Rn(d)`

Instruction Format:



Format: Instruction format for MOVUC.{B|W|L|Q}(z) <ea>(s), Rn(d)

Restrictions:

Operand constraint: Role `src`; relation `excluded`; values `immediate`; reason `user_source_memory_required`.

MOVUU — Move User To User — MOVUU

Operation: Copies from user-domain memory to user-domain memory.

Assembler Syntax: `MOVUU.{B|W|L|Q}(z) <ea>(s), <ea>(d)`

Privilege: Supervisor only

Repeat eligibility: REP eligible; REPcc observes `dst`

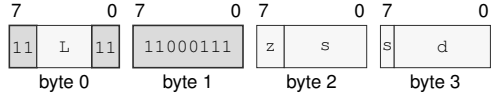
Detailed Semantics

MOVUU is a supervisor-only checked user-access move. Both source and destination operands must be memory operands, and both memory accesses use the user domain.

Encodings:

`MOVUU.{B|W|L|Q}(z) <ea>(s), <ea>(d)`

Instruction Format:



Format: Instruction format for MOVUU.{B|W|L|Q}(z) <ea>(s), <ea>(d)

Restrictions:

Operand constraint: Role `src`; relation `excluded`; values `immediate`; reason `user_source_memory_required`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

MSUB — Multiply Subtract — MSUB

Operation: Subtracts the low product of two sources from the destination.

Assembler Syntax: `MSUB.{B|W|L|Q}(z) Rn(l), Rn(r), Rn(d)`

Privilege: Unprivileged

Required CPUID flags: `MSUB`

Repeat eligibility: REP eligible; REPcc observes `dst`

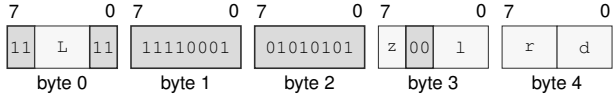
Detailed Semantics

Multiplies the selected-width source values, subtracts the low product from the old destination value, and writes the low selected-width result. The instruction does not update flags.

Encodings:

`MSUB.{B|W|L|Q}(z) Rn(l), Rn(r), Rn(d)`

Instruction Format:



Format: Instruction format for MSUB.{B|W|L|Q}(z) Rn(l), Rn(r), Rn(d)

MUL — Multiply Low — MUL

Operation: Multiplies the destination by the source and keeps the low half.

Assembler Syntax:
```
MUL.{B|W|L|Q}(z) <ea>(e), Rn(d)
MUL.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

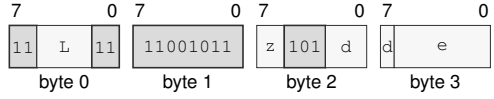
Detailed Semantics

MUL multiplies the selected-width operands and writes the low N bits of the product.

Encodings:

`MUL.{B|W|L|Q}(z) <ea>(e), Rn(d)`

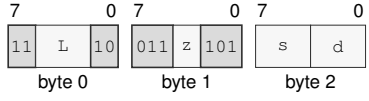
Instruction Format:



Format: Instruction format for MUL.{B|W|L|Q}(z) <ea>(e), Rn(d)

`MUL.{B|W|L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for MUL.{B|W|L|Q}(z) Rn(s), Rn(d)

MULHS — Signed Multiply High — MULHS

Operation: Writes the high half of the signed 64-bit source-by-destination product.

Assembler Syntax: `MULHS.Q Rn(s), Rn(d)`

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

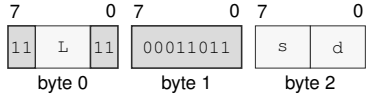
Detailed Semantics

Multiplies the 64-bit operands as signed integers and writes the high 64 bits of the 128-bit product.

Encodings:

`MULHS.Q Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for MULHS.Q Rn(s), Rn(d)

MULHSU — Signed By Unsigned Multiply High — MULHSU

Operation: Writes the high half of the signed-destination by unsigned-source 64-bit product.

Assembler Syntax: `MULHSU.Q Rn(s), Rn(d)`

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

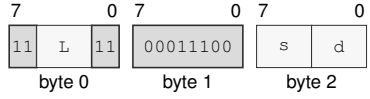
Detailed Semantics

Multiplies the signed 64-bit destination by the unsigned 64-bit source and writes the high 64 bits of the 128-bit product.

Encodings:

`MULHSU.Q Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for MULHSU.Q Rn(s), Rn(d)

MULHU — Unsigned Multiply High — MULHU

Operation: Writes the high half of the unsigned 64-bit source-by-destination product.

Assembler Syntax: `MULHU.Q Rn(s), Rn(d)`

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

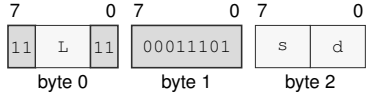
Detailed Semantics

Multiplies the 64-bit operands as unsigned integers and writes the high 64 bits of the 128-bit product.

Encodings:

`MULHU.Q Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for MULHU.Q Rn(s), Rn(d)

NEG — Negate — NEG

Operation: Writes the selected-width two's-complement negation of the destination.

Assembler Syntax:
```
NEG.{L|Q}(z) Rn(r)
NEG.{B|W|L|Q}(z) <ea>(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

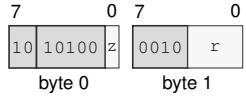
Detailed Semantics

Writes the selected-width two's-complement negation of the destination value.

Encodings:

`NEG.{L|Q}(z) Rn(r)`

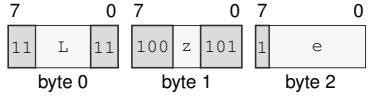
Instruction Format:



Format: Instruction format for NEG.{L|Q}(z) Rn(r)

`NEG.{B|W|L|Q}(z) <ea>(e)`

Instruction Format:



Format: Instruction format for NEG.{B|W|L|Q}(z) <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

NOP — No Operation — NOP

Operation: No architectural state change.

Assembler Syntax: `NOP`

Privilege: Unprivileged

Detailed Semantics

Retires normally without changing architectural state.

Encodings:

`NOP`

Instruction Format:

Format: Instruction format for NOP

NOT — Bitwise Not — NOT

Operation: Writes the selected-width bitwise complement of the destination.

Assembler Syntax:
```
NOT.{L|Q}(z) Rn(r)
NOT.{B|W|L|Q}(z) <ea>(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

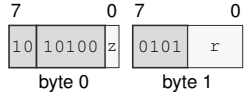
Detailed Semantics

Writes the selected-width bitwise complement of the destination value.

Encodings:

`NOT.{L|Q}(z) Rn(r)`

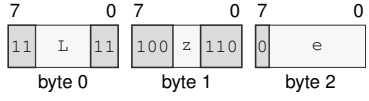
Instruction Format:



Format: Instruction format for NOT.{L|Q}(z) Rn(r)

`NOT.{B|W|L|Q}(z) <ea>(e)`

Instruction Format:



Format: Instruction format for NOT.{B|W|L|Q}(z) <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

OR — Bitwise Or — OR

Operation: Computes the bitwise OR of the source and destination operands.

Assembler Syntax:

```
OR.{L|Q}(z) Rn(s), Rn(d)
OR.{B|W|L|Q}(z) Rn(s), <ea>(e)
OR.{B|W|L|Q}(z) <ea>(e), Rn(d)
OR.{B|W|L|Q}(z) <imm8s>, <ea>(e)
OR.{W|L|Q}(z) <imm16s>, <ea>(e)
OR.{L|Q}(z) <imm32s>, <ea>(e)
OR.Q <imm64>, <ea>(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

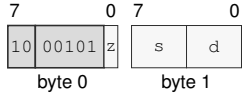
Detailed Semantics

Writes the selected-width bitwise disjunction of the source and destination values to the destination.

Encodings:

`OR.{L|Q}(z) Rn(s), Rn(d)`

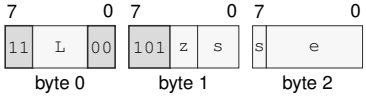
Instruction Format:



Format: Instruction format for OR.{L|Q}(z) Rn(s), Rn(d)

`OR.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



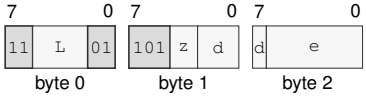
Format: Instruction format for OR.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`OR.{B|W|L|Q}(z) <ea>(e), Rn(d)`

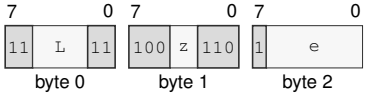
Instruction Format:



Format: Instruction format for OR.{B|W|L|Q}(z) <ea>(e), Rn(d)

`OR.{B|W|L|Q}(z) <imm8s>, <ea>(e)`

Instruction Format:



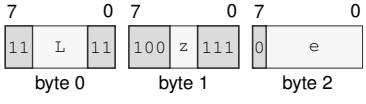
Format: Instruction format for OR.{B|W|L|Q}(z) <imm8s>, <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`OR.{W|L|Q}(z) <imm16s>, <ea>(e)`

Instruction Format:



Format: Instruction format for OR.{W|L|Q}(z) <imm16s>, <ea>(e)

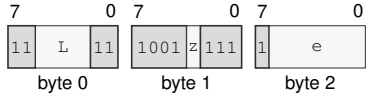
Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `immediate_wider_than_operation_size`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`OR.{L|Q}(z) <imm32s>, <ea>(e)`

Instruction Format:



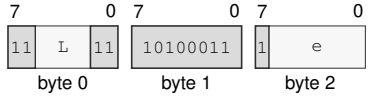
Format: Instruction format for OR.{L|Q}(z) <imm32s>, <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`OR.Q <imm64>, <ea>(e)`

Instruction Format:



Format: Instruction format for OR.Q <imm64>, <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

PARITY — Parity Test — PARITY

Operation: Computes operand parity and writes the predicate to an Rn register.

Assembler Syntax:
```
PARITY.{B|W|L|Q}(z) <ea>(e), Rn(d)
PARITY.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

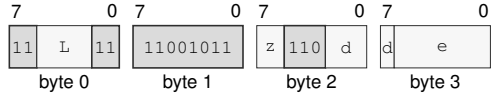
Detailed Semantics

Computes the parity of the selected operand size and writes popcount(src) & 1 to the destination register. The written value is 1 for odd parity and 0 for even parity. FLAGS are unchanged.

Encodings:

`PARITY.{B|W|L|Q}(z) <ea>(e), Rn(d)`

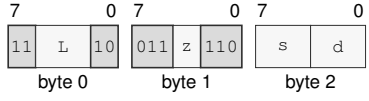
Instruction Format:



Format: Instruction format for PARITY.{B|W|L|Q}(z) <ea>(e), Rn(d)

`PARITY.{B|W|L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for PARITY.{B|W|L|Q}(z) Rn(s), Rn(d)

POP — Pop — POP

Operation: Pops one 64-bit stack value into an Rn or SREG destination.

Assembler Syntax:
```
POP Rn(r)
POP SREG(s)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `reg`

Detailed Semantics

POP loads one 64-bit stack value into an Rn or SREG destination and advances SP. The destination and updated SP commit together.

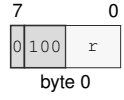
POP reads and validates the stack value without changing architectural state. For an SREG destination, segment-image validation completes before commit. It then updates the destination and SP together; any preceding access or validation fault leaves both unchanged.

POP SS performs the read using the old SS and exposes the new SS only at the final commit.

Encodings:

`POP Rn(r)`

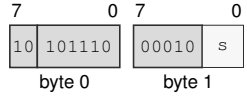
Instruction Format:



Format: Instruction format for POP Rn(r)

`POP SREG(s)`

Instruction Format:



Format: Instruction format for POP SREG(s)

POPCNT Population Count POPCNT

Operation: Counts set bits in the selected-width source and writes the count.

Assembler Syntax:
```
POPCNT.{B|W|L|Q}(z) <ea>(e), Rn(d)
POPCNT.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

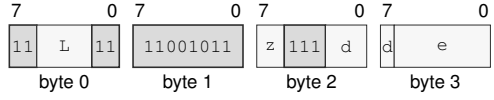
Detailed Semantics

Counts set bits within the selected B, W, L, or Q source width and writes that count to the destination.

Encodings:

`POPCNT.{B|W|L|Q}(z) <ea>(e), Rn(d)`

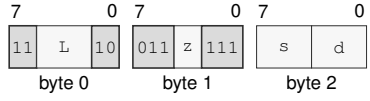
Instruction Format:



Format: Instruction format for POPCNT.{B|W|L|Q}(z) <ea>(e), Rn(d)

`POPCNT.{B|W|L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for POPCNT.{B|W|L|Q}(z) Rn(s), Rn(d)

POPP — Pop Register Pair — POPP

Operation: Pops one canonical Rn register pair selected by an inline pair index.

Assembler Syntax: `POPP PAIRn(i)`

Privilege: Unprivileged

Repeat eligibility: REP eligible

Detailed Semantics

Pops the canonical register pair selected by the pair ID. The canonical pair sequence is pair 0 (R0, R1), pair 1 (R2, R3), pair 2 (R4, R5), pair 3 (R6, R7), pair 4 (R8, R9), pair 5 (R10, R11), pair 6 (R12, R13), and pair 7 (R14, R15). All eight pair IDs are valid and each selects one pair.

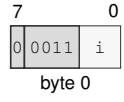
POPP selects one canonical pair using the unsigned 3-bit pair index and reads the complete two-slot stack range through the old SS:SP context before either destination register or SP is changed. For a listed pair (*first*, *second*), the second register is read from old SP and the first register is read from old SP plus 8. Both destination registers and the SP update to old SP plus 16 commit together. A preceding fault changes neither destination register nor SP.

Multiple POPP instructions can be used in reverse canonical epilogue order to restore multiple pairs.

Encodings:

`POPP PAIRn(i)`

Instruction Format:



Format: Instruction format for POPP PAIRn(i)

PREFETCH Prefetch PREFETCH

Operation: Hints that EA will be read soon; faults only under the configured prefetch-fault policy.

Assembler Syntax: `PREFETCH <ea>(e)`

Privilege: Unprivileged

Repeat eligibility: REP eligible

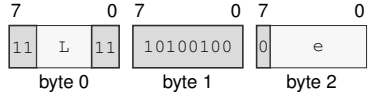
Detailed Semantics

Computes the byte-addressed source without an architectural read and issues an optional temporal read-prefetch hint. Ignoring the hint has no architectural effect except under an explicitly configured prefetch-fault policy.

Encodings:

`PREFETCH <ea>(e)`

Instruction Format:



Format: Instruction format for PREFETCH <ea>(e)

PREFETCHNT Prefetch Non-Temporal PREFETCHNT

Operation: Hint that EA will be read with non-temporal locality.

Assembler Syntax: `PREFETCHNT <ea>(e)`

Privilege: Unprivileged

Repeat eligibility: REP eligible

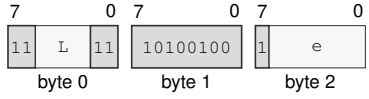
Detailed Semantics

Computes the byte-addressed source without an architectural read and issues an optional non-temporal read-prefetch hint. Ignoring the hint has no architectural effect except under an explicitly configured prefetch-fault policy.

Encodings:

`PREFETCHNT <ea>(e)`

Instruction Format:



Format: Instruction format for PREFETCHNT <ea>(e)

PTQUERY Page Table Query PTQUERY

Operation: Reads the selected raw page-table descriptor without changing translation state.

Assembler Syntax:
```
PTQUERY pt_level(i), <ea>(e), Rn(d)
PTQUERY pt_level(i), Rn(s), Rn(d)
```

Privilege: Supervisor only

Detailed Semantics

PTQUERY reads the raw descriptor selected by a linear address and requested page-table level, or returns zero when the requested entry cannot be read. `FLAGS.Z` reports structural validity under the layout selected by `PTE.T`; `FLAGS.N`, `FLAGS.C`, and `FLAGS.V` are zero.

The requested level is the low three bits of the level operand. Values outside 1 through 4 raise `INVALID_CONTROL_SELECTOR`. With `PTCR.TT=010`, an L4 query is reported as an ordinary unsuccessful query before reading a PTE. With an unassigned `PTCR.TT` encoding, no geometry is selected and every query is reported as an ordinary unsuccessful query before reading a PTE.

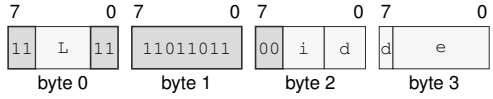
1. Compute the linear address without reading the source memory and reject a noncanonical address as an unsuccessful query.
2. Starting at the PTCR root, read one 64-bit PTE per level. Every preceding entry must be a valid table pointer; a valid earlier leaf prevents traversal to a lower requested level.
3. Stop after reading the requested level. A structurally malformed entry or valid leaf terminates the query at that level.
4. Return the requested raw PTE and set `FLAGS.Z` only when it is structurally valid under the table-pointer or leaf layout selected by `PTE.T` at that level. Table-pointer validity uses the current source level, and traversal preserves the complete `PTE.NEXT_TABLE` value. `FLAGS.N`, `FLAGS.C`, and `FLAGS.V` are zero.

Failure before the requested entry is read returns zero and clears all four flags. The walk ignores `PTCR.PE`, does not set `PTE.A` or `PTE.D`, does not modify a TLB, and does not raise an `ADDRESS_TRANSLATION` family event for walk failure.

Encodings:

```
PTQUERY pt_level(i), <ea>(e), Rn(d)
```

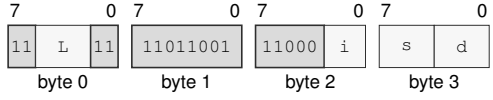
Instruction Format:



Format: Instruction format for PTQUERY pt_level(i), <ea>(e), Rn(d)

```
PTQUERY pt_level(i), Rn(s), Rn(d)
```

Instruction Format:



Format: Instruction format for PTQUERY pt_level(i), Rn(s), Rn(d)

PUSH — Push — PUSH

Operation: Pushes one Rn register value, SREG image, or the current CS image onto the stack.

Assembler Syntax:

```
PUSH CS
PUSH Rn(r)
PUSH SREG(s)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `reg`

Detailed Semantics

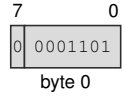
PUSH stores one 64-bit Rn register value, selected SREG image, or current CS image on the stack. The SREG encoding accepts DS, SS, and GS0-GS5. PUSH SS captures the old SS image and uses that same old SS as the stack context for the store.

PUSH captures the source value and current SS:SP before changing architectural state. After the destination stack slot has been validated, it commits the 64-bit store and decremented SP together. A fault before commit leaves memory and SP unchanged. For PUSH SS, both the captured value and the stack access use the old SS image.

Encodings:

`PUSH CS`

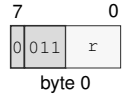
Instruction Format:



Format: Instruction format for PUSH CS

`PUSH Rn(r)`

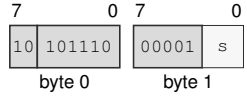
Instruction Format:



Format: Instruction format for PUSH Rn(r)

`PUSH SREG(s)`

Instruction Format:



Format: Instruction format for PUSH SREG(s)

PUSHP Push Register Pair PUSHP

Operation: Pushes one canonical Rn register pair selected by an inline pair index.

Assembler Syntax: `PUSHP PAIRn(i)`

Privilege: Unprivileged

Repeat eligibility: REP eligible

Detailed Semantics

Pushes the canonical register pair selected by the pair ID. The canonical pair sequence is pair 0 (R0, R1), pair 1 (R2, R3), pair 2 (R4, R5), pair 3 (R6, R7), pair 4 (R8, R9), pair 5 (R10, R11), pair 6 (R12, R13), and pair 7 (R14, R15). All eight pair IDs are valid and each selects one pair.

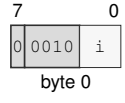
PUSHP selects one canonical pair using the unsigned 3-bit pair index and captures both register values and the old SS:SP context. It validates the complete two-slot stack range before either slot or SP is changed. For a listed pair (*first*, *second*), the second register is stored at old SP minus 16 and the first register is stored at old SP minus 8. Both stores and the SP update to old SP minus 16 commit together. A preceding fault changes neither slot nor SP.

Multiple PUSHP instructions can be used in canonical prologue order to save multiple pairs.

Encodings:

`PUSHP PAIRn(i)`

Instruction Format:



Format: Instruction format for PUSHP PAIRn(i)

RDCR Read Control Register RDCR

Operation: Read a privileged control register.

Assembler Syntax: `RDCR <imm16>, Rn(d)`

Privilege: Supervisor only

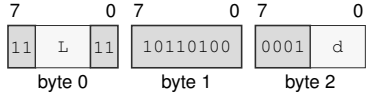
Detailed Semantics

RDCR reads a selected control register in supervisor mode.

Encodings:

`RDCR <imm16>, Rn(d)`

Instruction Format:



Format: Instruction format for RDCR <imm16>, Rn(d)

RDFLAGS Read Flags RDFLAGS

Operation: Read the integer condition-code flags into an Rn register.

Assembler Syntax: `RDFLAGS Rn(d)`

Privilege: Unprivileged

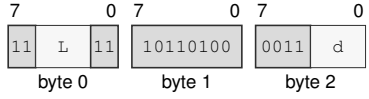
Detailed Semantics

RDFLAGS reads the architectural 16-bit FLAGS register, zero-extends it to 64 bits, and writes the result to the destination Rn register.

Encodings:

`RDFLAGS Rn(d)`

Instruction Format:



Format: Instruction format for RDFLAGS Rn(d)

RDPMC — Read Performance Counter — RDPMC

Operation: Read the 64-bit performance counter selected by a constant counter ID.

Assembler Syntax: `RDPMC <imm16>, Rn(d)`

Privilege: Unprivileged

Detailed Semantics

Performance-Counter IDs

RDPMC selects a 64-bit counter with its 16-bit immediate ID. The three standard counters are mandatory and wrap modulo 2^{64}. Cold reset and warm RESET clear them to zero. `PMC.EN` controls whether they increment.

Table 18-2. Standard Performance Counters

ID	Name	Increment event
1	`CYCLE`	each implementation cycle while `PMC.EN` is set
2	`INSTRET`	each successfully committed architectural instruction
3	`PTWALK`	start of a hardware page walk after a paging-enabled TLB miss

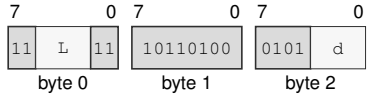
Each successfully committed repeat-body iteration contributes one INSTRET event. A faulting instruction does not. PTWALK includes a walk that later reports a page fault, but excludes a segment-bound or canonical-address failure detected before a walk begins. ID zero is unavailable because index zero of the CPUID performance-counter leaf is its common discovery header. Other IDs beyond the standard set are implementation-defined and are discovered through that leaf.

Every privilege level may read every implemented counter. Clearing `PMC.EN` freezes the current counter values without clearing them.

Encodings:

```
RDPMC <imm16>, Rn(d)
```

Instruction Format:



Format: Instruction format for RDPMC <imm16>, Rn(d)

RDTIME — Read Time — RDTIME

Operation: Read one atomic snapshot of the invariant architectural timebase.

Assembler Syntax: `RDTIME Rn(d)`

Privilege: Unprivileged

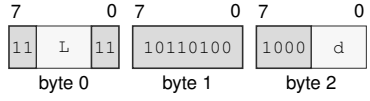
Detailed Semantics

RDTIME writes one atomic snapshot of the executing logical processor's 64-bit invariant timebase to the destination Rn register. User and supervisor execution observe the same timebase. RDTIME does not access memory, modify FLAGS, or order memory operations.

Encodings:

`RDTIME Rn(d)`

Instruction Format:



Format: Instruction format for RDTIME Rn(d)

RDSEG Read Segment Register RDSEG

Operation: Read an SREG or the current CS image into an Rn register.

Assembler Syntax:
```
RDSEG SREG(s), Rn(d)
RDSEG CS, Rn(d)
```

Privilege: Unprivileged

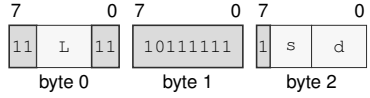
Detailed Semantics

RDSEG reads the 64-bit image of an SREG-class segment register into the destination Rn register. The RDSEG CS encoding reads the current CS image.

Encodings:

`RDSEG SREG(s), Rn(d)`

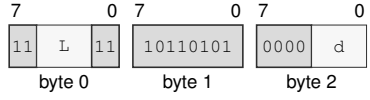
Instruction Format:



Format: Instruction format for RDSEG SREG(s), Rn(d)

`RDSEG CS, Rn(d)`

Instruction Format:



Format: Instruction format for RDSEG CS, Rn(d)

RDSTATUS — Read Status — RDSTATUS

Operation: Read the processor STATUS register into an Rn register.

Assembler Syntax: `RDSTATUS Rn(d)`

Privilege: Unprivileged

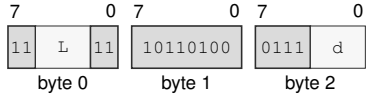
Detailed Semantics

RDSTATUS reads the architectural 16-bit STATUS register, zero-extends it to 64 bits, and writes the result to the destination Rn register. Reading STATUS is unprivileged; writing STATUS remains supervisor-only.

Encodings:

`RDSTATUS Rn(d)`

Instruction Format:



Format: Instruction format for RDSTATUS Rn(d)

REPcc — Repeat Conditional — REPcc

Operation: Repeats the next instruction while the counter and condition remain active.

Assembler Syntax: `REPcc Rn(r), (<instruction>)`

Privilege: Unprivileged

Detailed Semantics

REPcc installs repeat state for the following single instruction. The selected counter register is interpreted as an unsigned remaining count. REP is the unconditional spelling and uses condition T. Condition F is reserved and is not a valid encoding. A faulting iteration does not decrement the count. A successful iteration decrements it before an event may be admitted. An event between iterations saves the repeat-prefix address and clears hidden repeat state; ERET returns to the prefix and restores no repeat continuation.

The body instruction is fetched, decoded, and checked for repeatability before the zero-count rule is applied. If the counter is zero, the decoded body is annulled and has no architectural side effects.

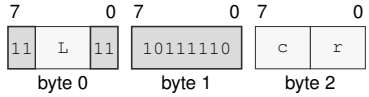
If the counter is nonzero, the first body iteration executes before any condition check. The repeat condition is tested against a temporary ZNCV image derived from the body instruction's repeat-observed value: Z reports zero, N is the most significant bit at the observation width, and C and V are zero. REPcc never uses architectural FLAGS as the condition input and does not write architectural FLAGS beyond any ordinary FLAGS effect of the body itself.

Body completion, observation, temporary-condition construction, architectural commit, counter decrement, and continuation PC selection are one precise atomic architectural commit step. An interrupt handler therefore cannot alter the continuation decision of an iteration that reached this step. Every completed body iteration decrements the counter by one, including the terminating condition-false iteration. Writes by the body to the selected counter register have no architectural effect while repeat state is active.

Encodings:

`REPcc Rn(r), (<instruction>)`

Instruction Format:



Format: Instruction format for REPcc Rn(r), (<instruction>)

Restrictions:

Operand constraint: Role `cc`; relation `allowed`; values `0, 0x2..0xf`; reason `condition_false_reclaimed`.

RESET Reset RESET

Operation: Perform warm reset; preserve BOOTPC and BOOTCFG.

Assembler Syntax: `RESET`

Privilege: Supervisor only

Detailed Semantics

RESET serializes earlier memory effects, applies the complete architectural warm-reset state to the executing logical processor, and resumes execution at BOOTPC.

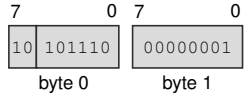
RESET is supervisor-only and affects only the executing logical processor. Before the reset-state transition commits, all earlier memory effects and pending write-combining stores from that logical processor complete. No younger instruction effect becomes visible.

The complete register, hidden-state, and cache effects of the transition are defined under Event Reset and Context State. Instruction-fetch synchronization completes before the first instruction at BOOTPC is fetched.

Encodings:

`RESET`

Instruction Format:



Format: Instruction format for RESET

RESTORE — Restore Base User State — RESTORE

Operation: Restores the base user-state record.

Assembler Syntax: `RESTORE [Rn(r)]`

Privilege: Unprivileged

Detailed Semantics

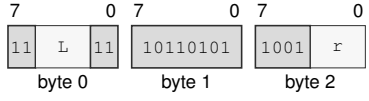
RESTORE reads one 200-byte base user-state record from the address captured from Rn before the instruction executes. The address must be 8-byte aligned. It validates every GS image and the FLAGS image before atomically replacing R0 through R15, GS0 through GS5, FLAGS, LPC, and the complete opaque LPA image.

RESTORE does not read or change supervisor or extension state. Any address, access, or validation fault leaves architectural state unchanged.

Encodings:

`RESTORE [Rn(r)]`

Instruction Format:



Format: Instruction format for RESTORE [Rn(r)]

RET Return RET

Operation: Pops and validates a return PC, then atomically advances SP and transfers control.

Assembler Syntax: `RET`

Privilege: Unprivileged

Detailed Semantics

When LPA.A is one, RET validates LPC as a near return target without authenticating LPA.PA. It restores eight bytes of stack footprint when PA is zero and 16 bytes when PA is nonzero, clears LPC and LPA, and transfers within the current CS. No return-record memory location is accessed on this path.

When LPA.A is zero, RET reads one 64-bit return address from SS:SP, advances SP by eight bytes, validates the target, and transfers control to it.

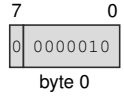
1. Read the 8-byte return address through the old SS:SP context without changing SP.
2. Validate that address as an executable target under the current CS.
3. Commit advanced SP and the new PC as one successful control-transfer operation.

A stack-read or target-validation fault leaves SP, PC, LPC, and LPA unchanged.

Encodings:

`RET`

Instruction Format:



Format: Instruction format for RET

REVBIT — Reverse Bits — REVBIT

Operation: Reverses bit order within the selected operand width.

Assembler Syntax: `REVBIT.{B|W|L|Q}(z) Rn(d)`

Privilege: Unprivileged

Required CPUID flags: `REVBIT`

Repeat eligibility: REP eligible; REPcc observes `dst`

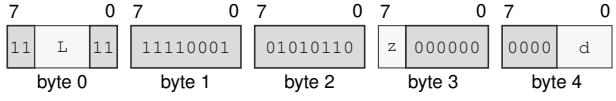
Detailed Semantics

Reverses the order of all bits within the selected width. Source bit (i) becomes destination bit (n-1-i), where (n) is the selected width in bits.

Encodings:

`REVBIT.{B|W|L|Q}(z) Rn(d)`

Instruction Format:



Format: Instruction format for REVBIT.{B|W|L|Q}(z) Rn(d)

REVBYTE Reverse Bytes REVBYTE

Operation: Reverses byte order within the selected operand width and writes the result.

Assembler Syntax:

```
REVBYTE.W Rn(r)
REVBYTE.L Rn(r)
REVBYTE.Q Rn(r)
REVBYTE.W <ea>(e)
REVBYTE.L <ea>(e)
REVBYTE.Q <ea>(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

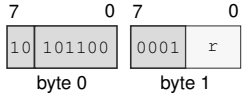
Detailed Semantics

Reverses the byte order within the selected operand width and writes the result to the destination.

Encodings:

`REVBYTE.W Rn(r)`

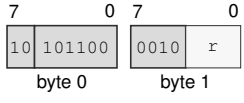
Instruction Format:



Format: Instruction format for REVBYTE.W Rn(r)

`REVBYTE.L Rn(r)`

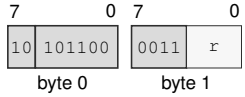
Instruction Format:



Format: Instruction format for REVBYTE.L Rn(r)

`REVBYTE.Q Rn(r)`

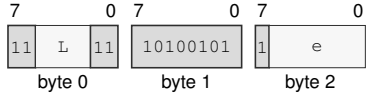
Instruction Format:



Format: Instruction format for REVBYTE.Q Rn(r)

`REVBYTE.W <ea>(e)`

Instruction Format:



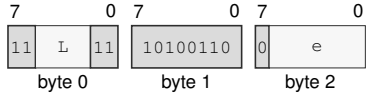
Format: Instruction format for REVBYTE.W <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`REVBYTE.L <ea>(e)`

Instruction Format:



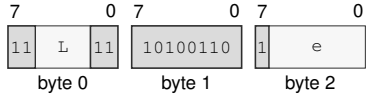
Format: Instruction format for REVBYTE.L <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`REVBYTE.Q <ea>(e)`

Instruction Format:



Format: Instruction format for REVBYTE.Q <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

RFENCE — Read Fence — RFENCE

Operation: Order prior reads before later reads.

Assembler Syntax: `RFENCE`

Privilege: Unprivileged

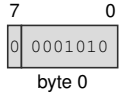
Detailed Semantics

Prevents retirement until every earlier memory read is ordered before every later memory read on the same logical processor. It performs no memory access and changes no register or flag.

Encodings:

`RFENCE`

Instruction Format:



Format: Instruction format for RFENCE

ROL — Rotate Left — ROL

Operation: Rotates the destination left by the count modulo the selected width without changing FLAGS.

Assembler Syntax:

```
ROL.{L|Q}(z) Rn(s), Rn(d)
ROL.{B|W|L|Q}(z) Rn(s), <ea>(e)
ROL.{B|W|L|Q}(z) imm6(i), <ea>(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

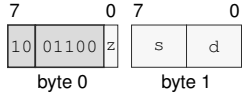
Detailed Semantics

Rotates the selected-width destination value left by the effective count and writes the result to the destination. The effective count is the count operand modulo the selected operand width: B uses mod 8, W uses mod 16, L uses mod 32, and Q uses mod 64. If the effective count is zero, the selected-width result equals the original selected-width value; an Rn destination is still written and a sub-Q result is zero-extended to 64 bits, while a memory destination rewrites the same selected bytes. FLAGS are unchanged for all counts.

Encodings:

`ROL.{L|Q}(z) Rn(s), Rn(d)`

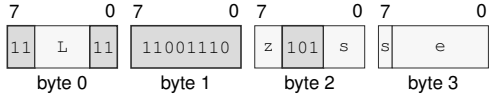
Instruction Format:



Format: Instruction format for ROL.{L|Q}(z) Rn(s), Rn(d)

`ROL.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



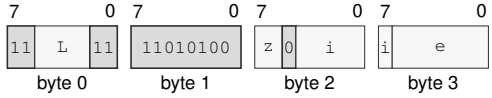
Format: Instruction format for ROL.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`ROL.{B|W|L|Q}(z) imm6(i), <ea>(e)`

Instruction Format:



Format: Instruction format for ROL.{B|W|L|Q}(z) imm6(i), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

ROR — Rotate Right — ROR

Operation: Rotates the destination right by the count modulo the selected width without changing FLAGS.

Assembler Syntax:

```
ROR.{L|Q}(z) Rn(s), Rn(d)
ROR.{B|W|L|Q}(z) Rn(s), <ea>(e)
ROR.{B|W|L|Q}(z) imm6(i), <ea>(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

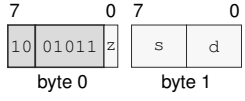
Detailed Semantics

Rotates the selected-width destination value right by the effective count and writes the result to the destination. The effective count is the count operand modulo the selected operand width: B uses mod 8, W uses mod 16, L uses mod 32, and Q uses mod 64. If the effective count is zero, the selected-width result equals the original selected-width value; an Rn destination is still written and a sub-Q result is zero-extended to 64 bits, while a memory destination rewrites the same selected bytes. FLAGS are unchanged for all counts.

Encodings:

`ROR.{L|Q}(z) Rn(s), Rn(d)`

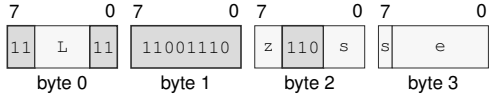
Instruction Format:



Format: Instruction format for ROR.{L|Q}(z) Rn(s), Rn(d)

`ROR.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



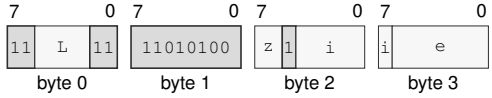
Format: Instruction format for ROR.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`ROR.{B|W|L|Q}(z) imm6(i), <ea>(e)`

Instruction Format:



Format: Instruction format for ROR.{B|W|L|Q}(z) imm6(i), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

SAR — Shift Arithmetic Right — SAR

Operation: Arithmetic-right-shifts the destination by the count modulo its selected width without changing FLAGS.

Assembler Syntax:

```
SAR.{L|Q}(z) Rn(s), Rn(d)
SAR.{B|W|L|Q}(z) Rn(s), <ea>(e)
SAR.{B|W|L|Q}(z) imm6(i), <ea>(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

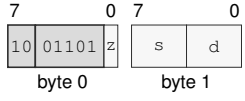
Detailed Semantics

Shifts the selected-width destination value right by the effective count, sign-fills the vacated high bits, and writes the result to the destination. The effective count is the count operand modulo the selected operand width: B uses mod 8, W uses mod 16, L uses mod 32, and Q uses mod 64. If the effective count is zero, the selected-width result equals the original selected-width value; an Rn destination is still written and a sub-Q result is sign-extended to 64 bits, while a memory destination rewrites the same selected bytes. FLAGS are unchanged for all counts.

Encodings:

`SAR.{L|Q}(z) Rn(s), Rn(d)`

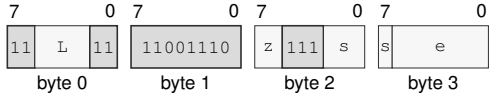
Instruction Format:



Format: Instruction format for SAR.{L|Q}(z) Rn(s), Rn(d)

`SAR.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



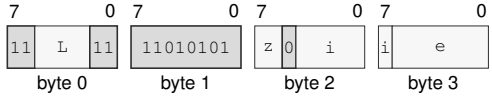
Format: Instruction format for SAR.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`SAR.{B|W|L|Q}(z) imm6(i), <ea>(e)`

Instruction Format:



Format: Instruction format for SAR.{B|W|L|Q}(z) imm6(i), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

SAVE — Save Base User State — SAVE

Operation: Saves the base user-state record.

Assembler Syntax: `SAVE [Rn(r)]`

Privilege: Unprivileged

Detailed Semantics

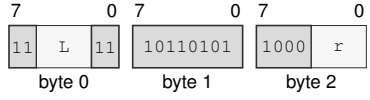
SAVE writes one base user-state record at the address captured from Rn before the instruction executes. The address must be 8-byte aligned. The 200-byte record contains R0 through R15, GS0 through GS5, the FLAGS image, LPC, and the complete opaque LPA image; it contains no supervisor or extension-owned key state.

The complete record range is validated before any byte is written. SAVE is a pure snapshot and does not change architectural state. Any address or access fault leaves architectural state and the destination record unchanged.

Encodings:

`SAVE [Rn(r)]`

Instruction Format:



Format: Instruction format for SAVE [Rn(r)]

SBB — Subtract With Borrow — SBB

Operation: Subtracts source and C borrow from destination, writes the modular difference, and updates ZNCV.

Assembler Syntax:

```
SBB.{B|W|L|Q}(z) Rn(s), Rn(d)
SBB.{B|W|L|Q}(z) Rn(s), <ea>(e)
SBB.{B|W|L|Q}(z) <ea>(e), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

Detailed Semantics

Subtracts the selected-width source value and incoming `FLAGS.C` borrow from the destination, writes the truncated difference, and updates `FLAGS.Z`, `FLAGS.N`, `FLAGS.C`, and `FLAGS.V`. The written destination result supplies the REPcc observed value.

Let n be the selected width, d and s the unsigned operand bit patterns, and b the incoming borrow represented by `FLAGS.C`. The written result is

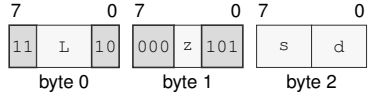
$$r = (d - s - b) \bmod 2^n.$$

`FLAGS.Z` is set when $r = 0$, and `FLAGS.N` receives the most-significant bit of r. `FLAGS.C` reports an unsigned borrow from either subtraction. `FLAGS.V` reports signed overflow for the complete $d - s - b$ operation.

Encodings:

`SBB.{B|W|L|Q}(z) Rn(s), Rn(d)`

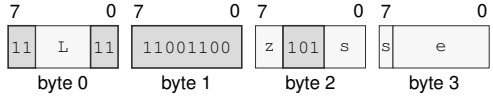
Instruction Format:



Format: Instruction format for SBB.{B|W|L|Q}(z) Rn(s), Rn(d)

`SBB.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



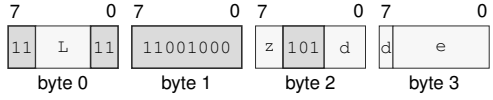
Format: Instruction format for SBB.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`SBB.{B|W|L|Q}(z) <ea>(e), Rn(d)`

Instruction Format:



Format: Instruction format for SBB.{B|W|L|Q}(z) <ea>(e), Rn(d)

SRESTORE — Restore Base Supervisor State — SRESTORE

Operation: Restores the base supervisor-state record.

Assembler Syntax: `SRESTORE [Rn(r)]`

Privilege: Supervisor only

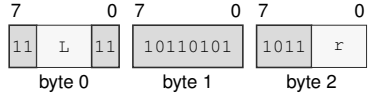
Detailed Semantics

SRESTORE reads one 8-byte base supervisor-state record from the 8-byte-aligned address captured from Rn. It requires bits 63 through 17 to be zero and validates the complete STATUS and DFA event-state combination before atomically replacing them. A fault leaves architectural state unchanged.

Encodings:

`SRESTORE [Rn(r)]`

Instruction Format:



Format: Instruction format for SRESTORE [Rn(r)]

SSAVE — Save Base Supervisor State — SSAVE

Operation: Saves the base supervisor-state record.

Assembler Syntax: `SSAVE [Rn(r)]`

Privilege: Supervisor only

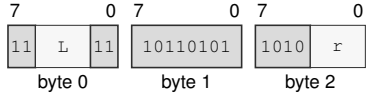
Detailed Semantics

SSAVE writes one 8-byte base supervisor-state record at the 8-byte-aligned address captured from Rn. Bits 15 through 0 contain STATUS, bit 16 contains DFA, and bits 63 through 17 are zero. SSAVE is a pure snapshot and does not change architectural state.

Encodings:

`SSAVE [Rn(r)]`

Instruction Format:



Format: Instruction format for SSAVE [Rn(r)]

SEGLEA Segment Load Effective Address SEGLEA

Operation: Computes segment pre-translation of the effective-address point and reports bounds failure in FLAGS.

Assembler Syntax:
```
SEGLEA.{B|W|L|Q}(z) <ea>(e), Rn(d)
SEGLEA.{B|W|L|Q}(z) Rn(s), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `computed`

Detailed Semantics

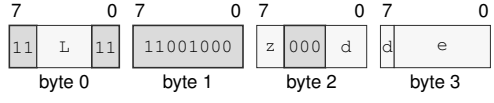
SEGLEA computes the source effective-address point and applies the selected segment's pre-translation to that point. The size suffix selects the index scale used during effective-address calculation. On success, SEGLEA writes the segment result to the destination and clears FLAGS. A failed segment-bound check sets `FLAGS.V` while clearing `FLAGS.Z`, `FLAGS.N`, and `FLAGS.C`, suppresses the destination write, commits effective-address auto-update effects, and completes without an exception. The B-sized 0 or 1 failure result supplies the REPcc observed value.

Let a be the source effective address before a memory read. The effective-address form selects a segment register image, which supplies the $base$, $span$, and $limit$ quantities used by segment pre-translation. A zero mantissa disables bounds and returns a. In translated mode, a must satisfy $0 \leq a < span$ and the result is $base + a$. In bounds-only mode, a must satisfy $base \leq a < limit$ and the result remains a. An out-of-bounds address writes FLAGS as `FLAGS.Z=FLAGS.N=FLAGS.C=0` and `FLAGS.V=1`, in which case the destination is not written. A successful address writes FLAGS as `FLAGS.Z=FLAGS.N=FLAGS.C=FLAGS.V=0`.

Encodings:

`SEGLEA.{B|W|L|Q}(z) <ea>(e), Rn(d)`

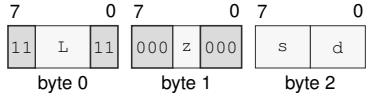
Instruction Format:



Format: Instruction format for SEGLEA.{B|W|L|Q}(z) <ea>(e), Rn(d)

`SEGLEA.{B|W|L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for SEGLEA.{B|W|L|Q}(z) Rn(s), Rn(d)

SET — Set True — SET

Operation: Writes integer one to the destination operand.

Assembler Syntax: `SET Rn(r)`

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

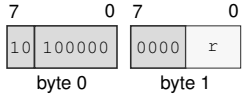
Detailed Semantics

Writes integer one to the selected-width destination.

Encodings:

`SET Rn(r)`

Instruction Format:



Format: Instruction format for SET Rn(r)

SETcc — Set Conditional — SETcc

Operation: Writes one or zero to the destination according to the encoded condition.

Assembler Syntax: `SETcc Rn(r)`

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

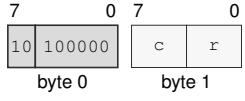
Detailed Semantics

Writes one when the encoded condition is true and zero when it is false.

Encodings:

`SETcc Rn(r)`

Instruction Format:



Format: Instruction format for SETcc Rn(r)

Restrictions:

Operand constraint: Role `cc`; relation `allowed`; values `0x2..0xf`; reason `condition_true_false_reclaimed`.

SETF Set Flags Image SETF

Operation: Replaces integer FLAGS from an immediate bitmap.

Assembler Syntax: `SETF flags_bitmap(m)`

Privilege: Unprivileged

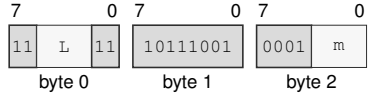
Detailed Semantics

SETF replaces the complete architectural integer FLAGS image from its encoded bitmap. Bit 3 writes `FLAGS.Z`, bit 2 writes `FLAGS.N`, bit 1 writes `FLAGS.C`, and bit 0 writes `FLAGS.V`; a one writes the corresponding flag to one and a zero writes it to zero. The assembler may provide symbolic bitmap names such as `Z`, `N`, `C`, `V`, and combinations using |.

Encodings:

`SETF flags_bitmap(m)`

Instruction Format:



Format: Instruction format for SETF flags_bitmap(m)

SPECFENCE Speculation Fence SPECFENCE

Operation: Prevents younger instructions from executing speculatively across the fence.

Assembler Syntax: `SPECFENCE`

Privilege: Unprivileged

Required CPUID flags: `SPECFENCE`

Detailed Semantics

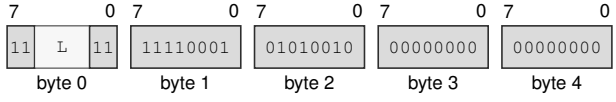
No architecturally younger instruction may issue an externally observable or transiently data-dependent operation until SPECFENCE retires. No architecturally older conditional control transfer or faulting instruction may remain unresolved when SPECFENCE retires.

SPECFENCE changes no register, flag, or memory value and is not a memory-ordering fence. Programs use `RFENCE`, `WFENCE`, or `AFENCE` when they need architectural memory ordering.

Encodings:

`SPECFENCE`

Instruction Format:



Format: Instruction format for SPECFENCE

SHL — Shift Left — SHL

Operation: Left-shifts the destination by the count modulo its selected width without changing FLAGS.

Assembler Syntax:
```
SHL.{L|Q}(z) Rn(s), Rn(d)
SHL.{B|W|L|Q}(z) Rn(s), <ea>(e)
SHL.{B|W|L|Q}(z) imm6(i), <ea>(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

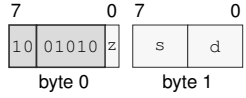
Detailed Semantics

Shifts the selected-width destination value left by the effective count, zero-fills the vacated low bits, and writes the result to the destination. The effective count is the count operand modulo the selected operand width: B uses mod 8, W uses mod 16, L uses mod 32, and Q uses mod 64. If the effective count is zero, the selected-width result equals the original selected-width value; an Rn destination is still written and a sub-Q result is zero-extended to 64 bits, while a memory destination rewrites the same selected bytes. FLAGS are unchanged for all counts.

Encodings:

`SHL.{L|Q}(z) Rn(s), Rn(d)`

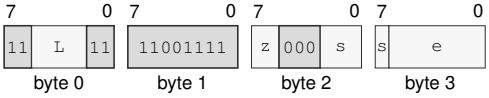
Instruction Format:



Format: Instruction format for SHL.{L|Q}(z) Rn(s), Rn(d)

`SHL.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



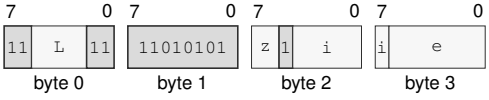
Format: Instruction format for SHL.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`SHL.{B|W|L|Q}(z) imm6(i), <ea>(e)`

Instruction Format:



Format: Instruction format for SHL.{B|W|L|Q}(z) imm6(i), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

SHR — Shift Right — SHR

Operation: Logical-right-shifts the destination by the count modulo its selected width without changing FLAGS.

Assembler Syntax:

```
SHR.{L|Q}(z) Rn(s), Rn(d)
SHR.{B|W|L|Q}(z) Rn(s), <ea>(e)
SHR.{B|W|L|Q}(z) imm6(i), <ea>(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

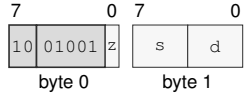
Detailed Semantics

Shifts the selected-width destination value right by the effective count, zero-fills the vacated high bits, and writes the result to the destination. The effective count is the count operand modulo the selected operand width: B uses mod 8, W uses mod 16, L uses mod 32, and Q uses mod 64. If the effective count is zero, the selected-width result equals the original selected-width value; an Rn destination is still written and a sub-Q result is zero-extended to 64 bits, while a memory destination rewrites the same selected bytes. FLAGS are unchanged for all counts.

Encodings:

`SHR.{L|Q}(z) Rn(s), Rn(d)`

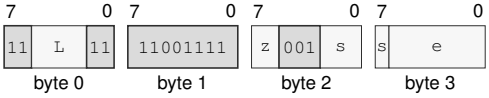
Instruction Format:



Format: Instruction format for SHR.{L|Q}(z) Rn(s), Rn(d)

`SHR.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



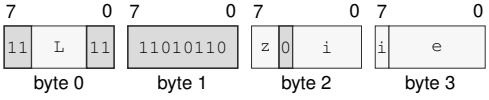
Format: Instruction format for SHR.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`SHR.{B|W|L|Q}(z) imm6(i), <ea>(e)`

Instruction Format:



Format: Instruction format for SHR.{B|W|L|Q}(z) imm6(i), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

SUB — Subtract — SUB

Operation: Subtracts the source operand from the destination operand.

Assembler Syntax:

```
SUB.Q 8, SP
SUB.{L|Q}(z) Rn(s), Rn(d)
SUB.Q imm8(i), SP
SUB.Q <imm16s>, SP
SUB.Q <imm32s>, SP
SUB.{B|W|L|Q}(z) Rn(s), <ea>(e)
SUB.{B|W|L|Q}(z) <ea>(e), Rn(d)
SUB.{B|W|L|Q}(z) <imm8s>, <ea>(e)
SUB.{W|L|Q}(z) <imm16s>, <ea>(e)
SUB.{L|Q}(z) <imm32s>, <ea>(e)
SUB.Q <imm64>, <ea>(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

Detailed Semantics

Performs selected-width modular subtraction of the source from the destination and writes the low result bits back to the destination.

Encodings:

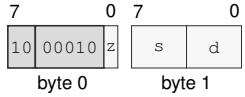
`SUB.Q 8, SP`

Instruction Format:

Format: Instruction format for SUB.Q 8, SP

`SUB.{L|Q}(z) Rn(s), Rn(d)`

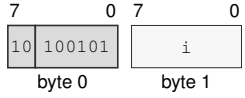
Instruction Format:



Format: Instruction format for SUB.{L|Q}(z) Rn(s), Rn(d)

`SUB.Q imm8(i), SP`

Instruction Format:



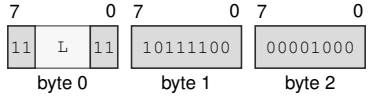
Format: Instruction format for SUB.Q imm8(i), SP

Restrictions:

Operand constraint: Role `imm`; relation `allowed`; values `0x1..0xff`; reason `zero_immediate_reclaimed`.

`SUB.Q <imm16s>, SP`

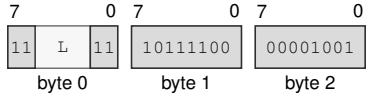
Instruction Format:



Format: Instruction format for SUB.Q <imm16s>, SP

`SUB.Q <imm32s>, SP`

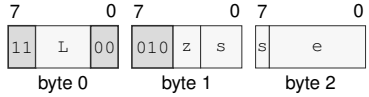
Instruction Format:



Format: Instruction format for SUB.Q <imm32s>, SP

`SUB.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



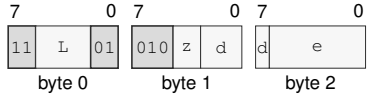
Format: Instruction format for SUB.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`SUB.{B|W|L|Q}(z) <ea>(e), Rn(d)`

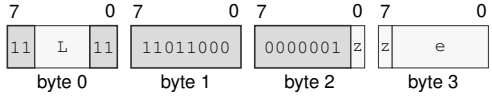
Instruction Format:



Format: Instruction format for SUB.{B|W|L|Q}(z) <ea>(e), Rn(d)

`SUB.{B|W|L|Q}(z) <imm8s>, <ea>(e)`

Instruction Format:



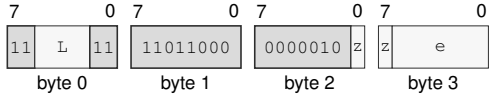
Format: Instruction format for SUB.{B|W|L|Q}(z) <imm8s>, <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`SUB.{W|L|Q}(z) <imm16s>, <ea>(e)`

Instruction Format:



Format: Instruction format for SUB.{W|L|Q}(z) <imm16s>, <ea>(e)

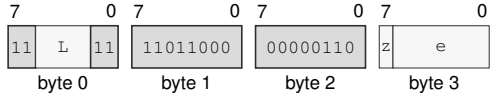
Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `immediate_wider_than_operation_size`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`SUB.{L|Q}(z) <imm32s>, <ea>(e)`

Instruction Format:



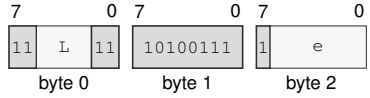
Format: Instruction format for SUB.{L|Q}(z) <imm32s>, <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`SUB.Q <imm64>, <ea>(e)`

Instruction Format:



Format: Instruction format for SUB.Q <imm64>, <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

STP Store Pair STP

Operation: Stores one canonical register pair as a 128-bit memory value.

Assembler Syntax: `STP PAIRn(s), <ea>(e)`

Privilege: Unprivileged

Required CPUID flags: `STP`

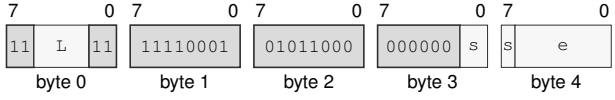
Detailed Semantics

Stores the selected canonical register pair as one naturally aligned 16-byte memory value. The even register supplies bits 63:0 and the following odd register supplies bits 127:64. The two halves form one compound store; a preceding alignment or memory-access fault exposes no partial write.

Encodings:

`STP PAIRn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for STP PAIRn(s), <ea>(e)

Restrictions:

Operand constraint: Role `memory`; relation `excluded`; values `immediate`; reason `memory_operand_required`.

SWPT — Switch Page Table — SWPT

Operation: Atomically installs PTCR and clears ASCR as a page-table switch.

Assembler Syntax: `SWPT Rn(p)`

Privilege: Supervisor only

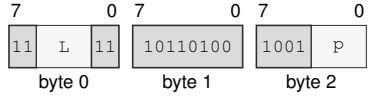
Detailed Semantics

Validates and replaces PTCR with the supplied page-table control value and clears ASCR as one architectural page-table switch. A PTCR validation failure leaves both registers unchanged.

Encodings:

`SWPT Rn(p)`

Instruction Format:



Format: Instruction format for SWPT Rn(p)

SWPTA — Switch Page Table With Context — SWPTA

Operation: Switches the page table and enables address-space context matching.

Assembler Syntax: `SWPTA Rn(p), Rn(a)`

Privilege: Supervisor only

Detailed Semantics

Validates and updates PTCR from the first Rn register, writes the low 16 bits of the second Rn register into ASCR[31:16], and sets `ASCR.AE`. A PTCR validation failure leaves both registers unchanged.

Encodings:

`SWPTA Rn(p), Rn(a)`

Instruction Format:

Format: Instruction format for SWPTA Rn(p), Rn(a)

SYNCCACHE — Synchronize Caches — SYNCCACHE

Operation: Synchronize one cache-maintenance block for local instruction fetch.

Assembler Syntax: `SYNCCACHE <ea>(e)`

Privilege: Supervisor only

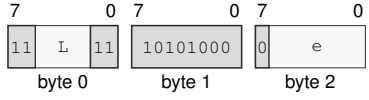
Detailed Semantics

Computes and translates the address-role EA, writes dirty data for its CPUID-sized maintenance block into normal-memory coherence throughout the coherence domain, invalidates the executing logical processor's matching instruction, decoded, and prefetch state, and serializes later instruction fetch.

Encodings:

`SYNCCACHE <ea>(e)`

Instruction Format:



Format: Instruction format for SYNCCACHE <ea>(e)

SYSCALL System Call SYSCALL

Operation: Raise the SYSTEM_CALL architectural event.

Assembler Syntax: `SYSCALL`

Privilege: Unprivileged

Detailed Semantics

Raise the explicit synchronous SYSTEM_CALL event (`0x00000003`), saving the following instruction as UPC and entering the common EPC/ECS/EDS event handler on SSS:SSP without a memory frame.

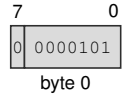
Execution is valid only in user mode with no active architectural event. It selects the explicit synchronous `SYSTEM_CALL` event, whose event code is `0x00000003`, priority is 4, frame class is BASIC, and saved PC is the following instruction.

On successful entry the common event mechanism atomically saves the user continuation in UPC, USP, UCS, UDS, USS, UCTL, and UINFO; loads CS, DS, and PC from ECS, EDS, and EPC; selects SSS:SSP; and sets `STATUS.PM`, `STATUS.EA`, `STATUS.EDEPTH=1`, and `STATUS.UO`. Because `SYSTEM_CALL` has no payload, entry performs no stack-memory access. Return uses ERET. Software restarts the one-byte instruction by subtracting one from UPC before ERET.

Encodings:

`SYSCALL`

Instruction Format:



Format: Instruction format for SYSCALL

TARM — Arm Deadline Timer — TARM

Operation: Atomically install an absolute deadline and interrupt identity.

Assembler Syntax: `TARM Rn(d), Rn(i)`

Privilege: Supervisor only

Detailed Semantics

TARM atomically installs the absolute deadline from the first Rn register and the interrupt identity from the second. The identity must be in the implemented range 1 through `ICAP.MAX_ID`; otherwise TARM raises `INVALID_CONTROL_SELECTOR` without changing timer state.

The signed modular distance from the current timebase to the supplied deadline determines whether the deadline is future. A positive distance arms the timer and replaces any prior arm. Zero or a negative distance immediately posts the identity to the executing logical processor's interrupt file and leaves the timer disarmed. TARM does not clear pending state posted by an earlier expiry and does not order memory operations.

Encodings:

`TARM Rn(d), Rn(i)`

Instruction Format:

Format: Instruction format for TARM Rn(d), Rn(i)

TCANCEL — Cancel Deadline Timer — TCANCEL

Operation: Atomically disarm the current deadline timer.

Assembler Syntax: `TCANCEL`

Privilege: Supervisor only

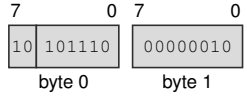
Detailed Semantics

TCANCEL atomically disarms the executing logical processor's deadline timer. Executing TCANCEL while the timer is already disarmed is a valid no-op. TCANCEL does not clear or claim any pending interrupt-file identity and does not order memory operations.

Encodings:

`TCANCEL`

Instruction Format:



Format: Instruction format for TCANCEL

TEST Test TEST

Operation: Computes a bitwise conjunction and updates condition codes without storing the temporary value.

Assembler Syntax:
```
TEST.{L|Q}(z) Rn(s), Rn(d)
TEST.{B|W|L|Q}(z) Rn(s), <ea>(e)
TEST.{B|W|L|Q}(z) <ea>(e), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `computed`

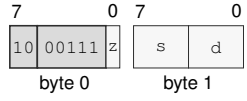
Detailed Semantics

Computes the selected-width bitwise conjunction only to set `FLAGS.Z` and `FLAGS.N`, clears `FLAGS.C` and `FLAGS.V`, and does not write the temporary result. The temporary conjunction supplies the REPcc observed value.

Encodings:

`TEST.{L|Q}(z) Rn(s), Rn(d)`

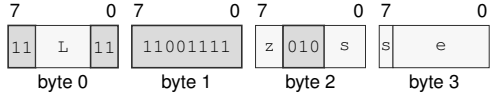
Instruction Format:



Format: Instruction format for TEST.{L|Q}(z) Rn(s), Rn(d)

`TEST.{B|W|L|Q}(z) Rn(s), <ea>(e)`

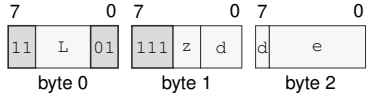
Instruction Format:



Format: Instruction format for TEST.{B|W|L|Q}(z) Rn(s), <ea>(e)

`TEST.{B|W|L|Q}(z) <ea>(e), Rn(d)`

Instruction Format:



Format: Instruction format for TEST.{B|W|L|Q}(z) <ea>(e), Rn(d)

TESTJcc — Test And Jump Conditional — TESTJcc

Operation: Tests two Rn registers and branches on the selected condition without writing FLAGS.

Assembler Syntax:
```
TESTJcc.{B|W|L|Q}(z) Rn(s), Rn(d), <imm8s>
TESTJcc.{B|W|L|Q}(z) Rn(s), Rn(d), <imm16s>
```

Privilege: Unprivileged

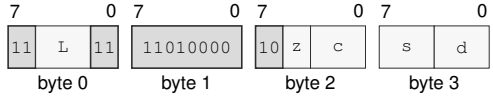
Detailed Semantics

TESTJcc computes the same temporary logical-test result as TEST for the selected operand size, evaluates the encoded condition from that temporary result, and branches to the relative target when the condition is true. The computed condition is consumed only by the branch decision; architectural FLAGS are not read or written. Conditions T and F are reserved and are not valid encodings.

Encodings:

`TESTJcc.{B|W|L|Q}(z) Rn(s), Rn(d), <imm8s>`

Instruction Format:



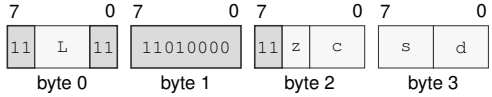
Format: Instruction format for TESTJcc.{B|W|L|Q}(z) Rn(s), Rn(d), <imm8s>

Restrictions:

Operand constraint: Role `cc`; relation `allowed`; values `0x2..0xf`; reason `condition_true_false_reclaimed`.

`TESTJcc.{B|W|L|Q}(z) Rn(s), Rn(d), <imm16s>`

Instruction Format:



Format: Instruction format for TESTJcc.{B|W|L|Q}(z) Rn(s), Rn(d), <imm16s>

Restrictions:

Operand constraint: Role `cc`; relation `allowed`; values `0x2..0xf`; reason `condition_true_false_reclaimed`.

TRACE — Trace Stream Marker — TRACE

Operation: Records a marker and instruction boundary in the implementation trace stream.

Assembler Syntax: `TRACE <imm16>`

Privilege: Unprivileged

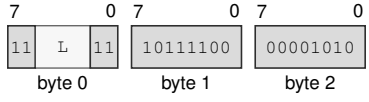
Detailed Semantics

When the implementation trace stream is enabled, appends the low 16-bit marker and current instruction boundary to that stream; otherwise it completes without changing architectural state. The tracing terminology distinguishes this marker instruction from a `STATUS.TF`-selected trace unit and the `SINGLE_STEP` architectural event. `STATUS.RF` does not suppress the marker.

Encodings:

`TRACE <imm16>`

Instruction Format:



Format: Instruction format for TRACE <imm16>

VTOP — Virtual To Physical Query — VTOP

Operation: Queries the current translation of a linear address without changing translation state.

Assembler Syntax: `VTOP Rn(v), Rn(p)`

Privilege: Supervisor only

Detailed Semantics

With paging disabled, VTOP returns the supplied address when it fits `PABITS`. With paging enabled, it returns the current physical byte-address translation without accessing the translated target. `FLAGS.Z` reports success; `FLAGS.N`, `FLAGS.C`, and `FLAGS.V` are zero. The query does not modify `PTE.A` or `PTE.D`.

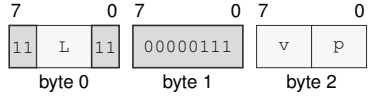
The source address is checked for canonicality before the page-table walk. The walk begins at the PTCR root and follows the LA45 or LA56 hierarchy selected by `PTCR.TT` using the virtual page number.

Every continuing entry must be a valid table pointer. `PTE.R`, `PTE.W`, `PTE.X`, and `PTE.U` are intersected across table pointers and the terminating leaf `PTE.AM`. A valid leaf at L1, L2, L3, or L4 combines its naturally aligned physical base with the original low 14, 23, 34, or 45 address bits respectively to form a byte address. Success sets `FLAGS.Z` and clears `FLAGS.N`, `FLAGS.C`, and `FLAGS.V`. Any canonicality, `PABITS`, or walk failure returns zero and clears all four flags without raising an access exception, setting `PTE.A` or `PTE.D`, or modifying the TLB.

Encodings:

`VTOP Rn(v), Rn(p)`

Instruction Format:



Format: Instruction format for VTOP Rn(v), Rn(p)

RELAX — Relax — RELAX

Operation: Permits a finite implementation-selected delay, including zero.

Assembler Syntax: `RELAX`

Privilege: Unprivileged

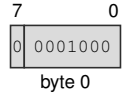
Detailed Semantics

RELAX is a PAUSE-like execution hint that retires normally to the following instruction. The implementation may delay subsequent execution by any finite duration, including zero.

Encodings:

`RELAX`

Instruction Format:



Format: Instruction format for RELAX

WFENCE — Write Fence — WFENCE

Operation: Order prior writes before later writes.

Assembler Syntax: `WFENCE`

Privilege: Unprivileged

Detailed Semantics

Prevents retirement until every earlier memory write is ordered before every later memory write on the same logical processor. It performs no memory access and changes no register or flag.

Encodings:

`WFENCE`

Instruction Format:

Format: Instruction format for WFENCE

WRBKDCACHE Write Back Data Cache WRBKDCACHE

Operation: Write back one data-cache maintenance block.

Assembler Syntax: `WRBKDCACHE <ea>(e)`

Privilege: Supervisor only

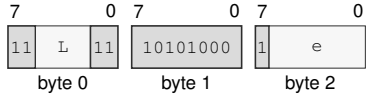
Detailed Semantics

Computes and translates the address-role EA without reading an operand value, selects its CPUID-sized maintenance block, writes dirty bytes from every data or unified cache copy in the coherence domain into normal-memory coherence, retains clean copies, and retires after the block is complete.

Encodings:

`WRBKDCACHE <ea>(e)`

Instruction Format:



Format: Instruction format for WRBKDCACHE <ea>(e)

WRCR — Write Control Register — WRCR

Operation: Write a privileged control register from an Rn register.

Assembler Syntax: `WRCR Rn(s), <imm16>`

Privilege: Supervisor only

Detailed Semantics

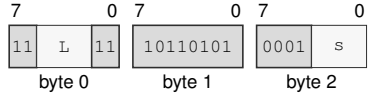
WRCR is supervisor-only and resolves its 16-bit selector in the architectural control register space. An unassigned selector, a read-only register, or a read-only indirect view raises `INVALID_CONTROL_SELECTOR`.

For a writable target, WRCR forms the candidate image according to the selected register's definition and validates every target-owned image and transition condition before changing architectural state. A rejected write leaves the register and every associated selector, bank, cache, and transition state unchanged. A successful write commits the candidate image and all target-defined side effects atomically. The selected register or its owning subsystem defines writable and ignored fields, image constraints, transition constraints, and commit effects.

Encodings:

`WRCR Rn(s), <imm16>`

Instruction Format:



Format: Instruction format for WRCR Rn(s), <imm16>

WRFLAGS Write Flags WRFLAGS

Operation: Write the integer condition-code flags from an Rn register.

Assembler Syntax: `WRFLAGS Rn(s)`

Privilege: Unprivileged

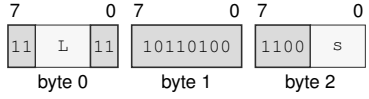
Detailed Semantics

WRFLAGS writes the low 16 bits from the source Rn register into the architectural FLAGS register. Reserved FLAGS bits must be zero.

Encodings:

`WRFLAGS Rn(s)`

Instruction Format:



Format: Instruction format for WRFLAGS Rn(s)

WRSEG Write Segment Register WRSEG

Operation: Write an SREG segment image from an Rn register.

Assembler Syntax: `WRSEG Rn(d), SREG(s)`

Privilege: Unprivileged

Detailed Semantics

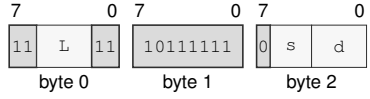
WRSEG validates a 64-bit segment register image from the source Rn register and writes it to the selected SREG.

The complete source image is validated before architectural state changes. An invalid image raises `INVALID_CONTROL_IMAGE`; the selected segment register remains unchanged when validation fails.

Encodings:

`WRSEG Rn(d), SREG(s)`

Instruction Format:



Format: Instruction format for WRSEG Rn(d), SREG(s)

WRSTATUS — Write Status — WRSTATUS

Operation: Write the processor STATUS register from an Rn register.

Assembler Syntax: `WRSTATUS Rn(s)`

Privilege: Supervisor only

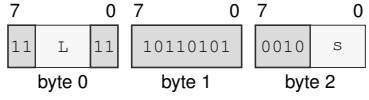
Detailed Semantics

WRSTATUS validates the low 16 bits from the source Rn register and atomically updates `STATUS.IE`, `STATUS.NI`, `STATUS.TF`, and `STATUS.RF`. Reserved bits must be zero, and the supplied `STATUS.PM`, `STATUS.EA`, `STATUS.EDEPTH`, and `STATUS.UO` values must equal the current values. Clearing `STATUS.NI` with a pending NMI admits that NMI at the next instruction boundary.

Encodings:

`WRSTATUS Rn(s)`

Instruction Format:



Format: Instruction format for WRSTATUS Rn(s)

XCHG Exchange XCHG

Operation: Exchanges selected-width operand values, committing both destinations together.

Assembler Syntax:
```
XCHG.{L|Q}(z) Rn(s), Rn(d)
XCHG.{B|W|L|Q}(z) Rn(s), <ea>(e)
XCHG.{B|W|L|Q}(z) <ea>(e), Rn(d)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible

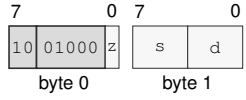
Detailed Semantics

Exchanges the two selected-width operand values as one instruction. A memory form performs an ordinary selected-width load followed by an ordinary selected-width store, with both architectural destinations committed by the instruction.

Encodings:

`XCHG.{L|Q}(z) Rn(s), Rn(d)`

Instruction Format:



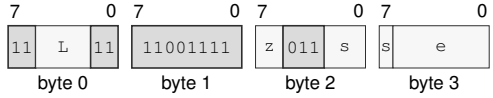
Format: Instruction format for XCHG.{L|Q}(z) Rn(s), Rn(d)

Restrictions:

Operand overlap: Operands `lhs`, `rhs`; rule `same_value`.

`XCHG.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



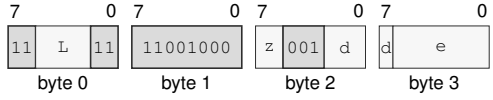
Format: Instruction format for XCHG.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `rhs`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`XCHG.{B|W|L|Q}(z) <ea>(e), Rn(d)`

Instruction Format:



Format: Instruction format for XCHG.{B|W|L|Q}(z) <ea>(e), Rn(d)

Restrictions:

Operand constraint: Role `lhs`; relation `excluded`; values `immediate`; reason `invalid_destination`.

XOR — Bitwise Xor — XOR

Operation: Computes the bitwise XOR of the source and destination operands.

Assembler Syntax:

```
XOR.{L|Q}(z) Rn(s), Rn(d)
XOR.{B|W|L|Q}(z) Rn(s), <ea>(e)
XOR.{B|W|L|Q}(z) <ea>(e), Rn(d)
XOR.{B|W|L|Q}(z) <imm8s>, <ea>(e)
XOR.{W|L|Q}(z) <imm16s>, <ea>(e)
XOR.{L|Q}(z) <imm32s>, <ea>(e)
XOR.Q <imm64>, <ea>(e)
```

Privilege: Unprivileged

Repeat eligibility: REP eligible; REPcc observes `dst`

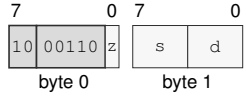
Detailed Semantics

Writes the selected-width bitwise exclusive-or of the source and destination values to the destination.

Encodings:

`XOR.{L|Q}(z) Rn(s), Rn(d)`

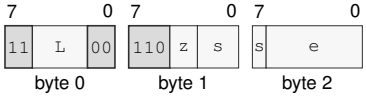
Instruction Format:



Format: Instruction format for XOR.{L|Q}(z) Rn(s), Rn(d)

`XOR.{B|W|L|Q}(z) Rn(s), <ea>(e)`

Instruction Format:



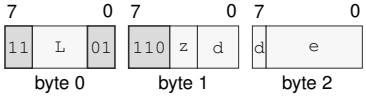
Format: Instruction format for XOR.{B|W|L|Q}(z) Rn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`XOR.{B|W|L|Q}(z) <ea>(e), Rn(d)`

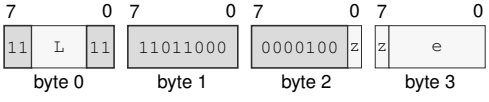
Instruction Format:



Format: Instruction format for XOR.{B|W|L|Q}(z) <ea>(e), Rn(d)

`XOR.{B|W|L|Q}(z) <imm8s>, <ea>(e)`

Instruction Format:



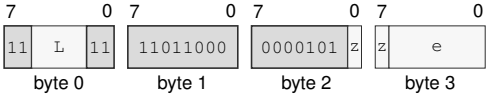
Format: Instruction format for XOR.{B|W|L|Q}(z) <imm8s>, <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`XOR.{W|L|Q}(z) <imm16s>, <ea>(e)`

Instruction Format:



Format: Instruction format for XOR.{W|L|Q}(z) <imm16s>, <ea>(e)

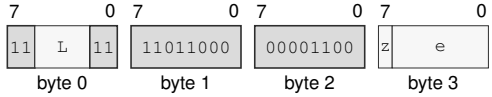
Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `immediate_wider_than_operation_size`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`XOR.{L|Q}(z) <imm32s>, <ea>(e)`

Instruction Format:



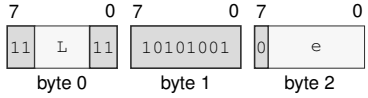
Format: Instruction format for XOR.{L|Q}(z) <imm32s>, <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`XOR.Q <imm64>, <ea>(e)`

Instruction Format:



Format: Instruction format for XOR.Q <imm64>, <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

YIELD Yield YIELD

Operation: Hints that the current logical processor may be deprioritized.

Assembler Syntax: `YIELD`

Privilege: Unprivileged

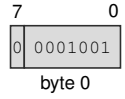
Detailed Semantics

Notifies the implementation that the current logical processor may be deprioritized or descheduled. YIELD affects scheduling only; memory accesses retain the ordinary instruction-ordering rules, and execution resumes at the next instruction.

Encodings:

`YIELD`

Instruction Format:



Format: Instruction format for YIELD

19 ADDRESS WAIT INSTRUCTIONS

19.1 Summary

Table 19-1. Address Wait Instructions Summary (Informative)

Mnemonic	Brief description
`WAIT`	Waits while a memory value equals the expected register value.
`WAKE`	Wakes all waiters armed on the addressed location.

19.2 Address Wait Protocol

WAIT and WAKE form an address-scoped blocking protocol. A waiter supplies an expected value and a memory address. The comparison read and monitor arm are indivisible: the logical processor blocks only when the selected memory value still equals the expected value at the instant the monitor becomes visible to WAKE.

WAKE broadcasts to every armed waiter whose resolved physical start address is the same as its operand address. It does not modify memory. An ordinary store is not required to wake a waiter; synchronization code first changes the condition with a release operation and then executes WAKE. A waiter always rechecks its condition because WAIT may return spuriously. These rules prevent a wake from being lost between the comparison and monitor arm without turning WAKE itself into a memory-ordering operation.

Blocking in WAIT does not stop the base architectural timebase. A deadline-timer interrupt may become pending while the monitor remains armed; ordinary interrupt admission determines when it is delivered.

WAIT — Wait On Address — WAIT

Operation: Waits while a memory value equals the expected register value.

Assembler Syntax: `WAIT.{B|W|L|Q}(z) Rn(x), <ea>(e)`

Privilege: Unprivileged

Required CPUID flags: `WAIT`

Detailed Semantics

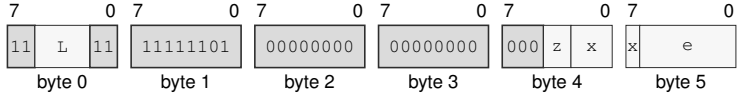
WAIT performs an acquire-ordered read of the selected-width memory operand. The comparison read and monitor arm are indivisible. If the value differs from the expected register value, the instruction returns without blocking. If the values are equal, the logical processor may block until a matching WAKE or a spurious return. Software therefore rechecks the condition after every return.

The monitor is identified by the resolved physical start address of the memory operand. WAIT does not modify the memory value, the expected register, or FLAGS. Ordinary stores are not required to make the instruction return.

Encodings:

`WAIT.{B|W|L|Q}(z) Rn(x), <ea>(e)`

Instruction Format:



Format: Instruction format for WAIT.{B|W|L|Q}(z) Rn(x), <ea>(e)

Restrictions:

Operand constraint: Role `memory`; relation `excluded`; values `immediate`; reason `memory_operand_required`.

WAKE — Wake Address Waiters — WAKE

Operation: Wakes all waiters armed on the addressed location.

Assembler Syntax: `WAKE <ea>(e)`

Privilege: Unprivileged

Required CPUID flags: `WAIT`

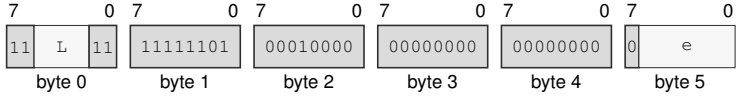
Detailed Semantics

WAKE broadcasts to every waiter whose armed monitor has the same resolved physical start address as the operand. The instruction does not read or modify memory and does not order memory operations. A producer normally publishes the changed condition with release ordering before executing WAKE.

Encodings:

`WAKE <ea>(e)`

Instruction Format:



Format: Instruction format for WAKE <ea>(e)

Restrictions:

Operand constraint: Role `address`; relation `excluded`; values `immediate`; reason `memory_operand_required`.

20 CONTROL-FLOW INTEGRITY INSTRUCTIONS

20.1 Summary

Table 20-1. Control-Flow Integrity Instructions Summary (Informative)

Mnemonic	Brief description
CFILAND	Marks a valid destination for an enforced indirect control transfer.
CFISRESTORE	Atomically restores the complete CFI supervisor-state record.
CFISSAVE	Saves the complete CFI supervisor-state record.
FPCALL	Installs an authenticated resident continuation without return-record memory traffic.
PCALL	Materializes an authenticated near-return record and transfers control.
PLCALL	Materializes an authenticated long-return record and transfers to a new CS:PC.
PLRET	Authenticates a long-return record and restores CS:PC.
PRET	Authenticates and consumes a near continuation.

20.2 Control-Flow Integrity Model

The combined LKH:LKL value selects the CFI context. Zero is unconfigured: protected call and return instructions raise INVALID_CONTROL_TRANSITION before operand or stack access, while ordinary indirect transfers do not require a landing marker. A nonzero key permits protected instructions and requires every taken indirect JMP or call-family target to begin with CFILAND. Direct transfers do not perform the landing check.

PCALL and PRET use a 16-byte ordinary-stack record containing a continuation PC and a nonzero 63-bit PA. PLCALL and PLRET use a 24-byte record containing a continuation PC, continuation CS, and PA. FPCALL stores its authenticated continuation in LPC:LPA while reserving the same 16-byte stack footprint without return-record memory traffic. Fast calls validate the exact SP subtraction without probing or accessing the reserved return-record range, and resident returns validate the corresponding SP addition. PRET authenticates either source before target validation; PLRET has no resident form.

Every PA authenticates the complete near or long continuation, uses distinct near and long domains, and binds the continuation to its post-allocation SS:SP. Changing the key invalidates prior tags, so software changes it only after discarding or making unreachable every outstanding protected continuation. The construction and serialization are implementation-defined, must generate nonzero tags, and must provide no greater than one chance in $2^{63} - 1$ of accepting a chosen forgery per independent attempt. A construction must remain stable across every supported processor migration and suspend/resume of the same architectural context. Any additional privilege, address-space, or execution-context inputs must have the same stability, or software must install a new key after discarding outstanding protected continuations. Same-depth temporal replay and return-target confidentiality are not provided.

An active resident link prevents every taken child call. The violation is detected before target or stack access and raises INVALID_CONTROL_TRANSITION. A false CALLcc is unaffected. RET may deliberately consume a protected resident continuation without authentication and is therefore outside the protected ABI; protected code uses PRET or PLRET.

The well-formed pairs are CALL or FCALL with RET, PCALL or FPCALL with PRET, LCALL with LRET, and PLCALL with PLRET. CALL followed by PRET applies the ordinary protected-record

load and authentication rules. A materialized PCALL followed by RET consumes only the continuation word and leaves the PA word on the stack; a resident FPCALL followed by RET restores the 16-byte footprint without authenticating PA. Selecting an unprotected return for a protected continuation opts out of return integrity. The protected-return guarantee therefore assumes trusted code generation and executable-code integrity preserve the selected protected return path.

Compiler output uses CALL and RET by default. FCALL is valid only for an activation that executes no taken child call before its RET, and the protected forms are selected only when the compiler enables the CFI extension. Stack alignment and stack-argument placement remain ABI responsibilities. Unwind metadata distinguishes resident from materialized return state and identifies the applicable stack footprint.

Setjmp and longjmp facilities, language exceptions, coroutines, and user-level context switches use the decomposed base and CFI context interfaces to preserve the applicable resident and key state; they do not assume whole-context atomicity. An event-entry stub remains call-free until it has saved the interrupted resident state and executed CLRLINK. If the CFI context may change, it also snapshots the key before that change. After restoring an active resident image, the stub executes ERET without an intervening taken call. Supervisor-origin, nested, and non-maskable events follow the same ordering.

CFILAND — Control-Flow Integrity Landing Pad — CFILAND

Operation: Marks a valid destination for an enforced indirect control transfer.

Assembler Syntax: `CFILAND`

Privilege: Unprivileged

Required CPUID flags: `CFI`

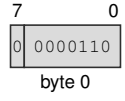
Detailed Semantics

CFILAND marks its own address as a valid destination for an indirect control transfer while LKH:LKL is nonzero. Direct transfers, sequential execution, and indirect transfers in an unconfigured CFI context do not require a landing marker. Executing CFILAND itself has no architectural effect other than ordinary instruction completion.

Encodings:

`CFILAND`

Instruction Format:



Format: Instruction format for CFILAND

CFISRESTORE Restore Control-Flow Integrity Supervisor State CFISRESTORE

Operation: Atomically restores the complete CFI supervisor-state record.

Assembler Syntax: `CFISRESTORE [Rn(r)]`

Privilege: Supervisor only

Required CPUID flags: `CFI`

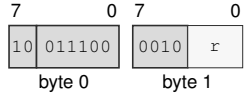
Detailed Semantics

CFISRESTORE is supervisor-only. It reads a complete 16-byte CFI supervisor-state record and replaces LKL and LKH together at one instruction commit point. The all-zero image is valid and disables CFI; LPC and LPA are unchanged.

Encodings:

`CFISRESTORE [Rn(r)]`

Instruction Format:



Format: Instruction format for CFISRESTORE [Rn(r)]

CFISSAVE Save Control-Flow Integrity Supervisor State CFISSAVE

Operation: Saves the complete CFI supervisor-state record.

Assembler Syntax: `CFISSAVE [Rn(r)]`

Privilege: Supervisor only

Required CPUID flags: `CFI`

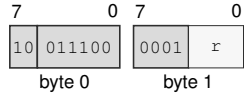
Detailed Semantics

CFISSAVE is supervisor-only and writes the complete 16-byte CFI supervisor-state record containing LKL followed by LKH. It is a pure snapshot and changes no L architectural state.

Encodings:

`CFISSAVE [Rn(r)]`

Instruction Format:



Format: Instruction format for CFISSAVE [Rn(r)]

FPCALL — Fast Protected Call — FPCALL

Operation: Installs an authenticated resident continuation without return-record memory traffic.

Assembler Syntax:

```
FPCALL <imm16s>
FPCALL <imm32s>
FPCALL <ea>(e)
FPCALL Rn(r)
```

Privilege: Unprivileged

Required CPUID flags: `CFI`

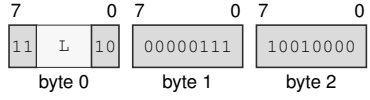
Detailed Semantics

FPCALL requires a configured CFI key and reserves the 16-byte protected near-call footprint without accessing that memory range. It installs the continuation PC and a nonzero PA in LPC:LPA. The PA uses the same near-return domain and post-allocation SS:SP binding as PCALL.

Encodings:

`FPCALL <imm16s>`

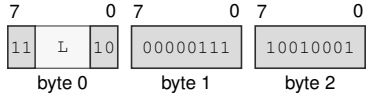
Instruction Format:



Format: Instruction format for FPCALL <imm16s>

`FPCALL <imm32s>`

Instruction Format:



Format: Instruction format for FPCALL <imm32s>

`FPCALL <ea>(e)`

Instruction Format:

Format: Instruction format for FPCALL <ea>(e)

Restrictions:

Operand constraint: Role `target`; relation `excluded`; values `immediate`; reason `canonical_form_reclaim`.

`FPCALL Rn(r)`

Instruction Format:

Format: Instruction format for FPCALL Rn(r)

PCALL — Protected Call — PCALL

Operation: Materializes an authenticated near-return record and transfers control.

Assembler Syntax:

```
PCALL <imm16s>
PCALL <imm32s>
PCALL <ea>(e)
PCALL Rn(r)
```

Privilege: Unprivileged

Required CPUID flags: `CFI`

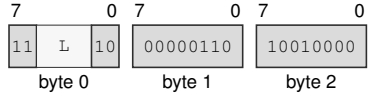
Detailed Semantics

PCALL requires a configured CFI key, reserves 16 bytes on SS:SP, and writes the continuation PC at offset zero followed by its nonzero 63-bit PA at offset eight. The PA binds the continuation to the post-allocation SS:SP and the near-return domain. A live resident continuation faults before target or stack access.

Encodings:

`PCALL <imm16s>`

Instruction Format:



Format: Instruction format for PCALL <imm16s>

`PCALL <imm32s>`

Instruction Format:

Format: Instruction format for PCALL <imm32s>

`PCALL <ea>(e)`

Instruction Format:

Format: Instruction format for PCALL <ea>(e)

Restrictions:

Operand constraint: Role `target`; relation `excluded`; values `immediate`; reason `canonical_form_reclaim`.

`PCALL Rn(r)`

Instruction Format:

Format: Instruction format for PCALL Rn(r)

PLCALL — Protected Long Call — PLCALL

Operation: Materializes an authenticated long-return record and transfers to a new CS:PC.

Assembler Syntax:
```
PLCALL Rn(r), <ea>(e)
PLCALL Rn(s), Rn(d)
```

Privilege: Unprivileged

Required CPUID flags: `CFI`

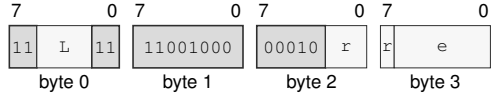
Detailed Semantics

PLCALL requires a configured CFI key and writes a 24-byte long-return record containing the continuation PC, continuation CS, and nonzero PA at offsets zero, eight, and 16. The PA binds both continuation values to the post-allocation SS:SP in a domain distinct from protected near returns.

Encodings:

`PLCALL Rn(r), <ea>(e)`

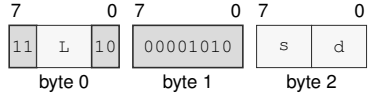
Instruction Format:



Format: Instruction format for PLCALL Rn(r), <ea>(e)

`PLCALL Rn(s), Rn(d)`

Instruction Format:



Format: Instruction format for PLCALL Rn(s), Rn(d)

PLRET — Protected Long Return — PLRET

Operation: Authenticates a long-return record and restores CS:PC.

Assembler Syntax: `PLRET`

Privilege: Unprivileged

Required CPUID flags: `CFI`

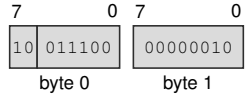
Detailed Semantics

PLRET requires a configured CFI key and no active resident continuation. It loads the 24-byte PLCALL record, authenticates the continuation PC and CS in the long-return domain, validates the target, advances SP by 24 bytes, and commits CS:PC. Failure leaves the pre-instruction state intact.

Encodings:

`PLRET`

Instruction Format:



Format: Instruction format for PLRET

PRET — Protected Return — PRET

Operation: Authenticates and consumes a near continuation.

Assembler Syntax: `PRET`

Privilege: Unprivileged

Required CPUID flags: `CFI`

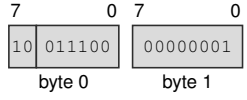
Detailed Semantics

PRET requires a configured CFI key. With an active resident link it authenticates LPC:LPA against the current SS:SP, restores 16 bytes of stack footprint, clears LPC:LPA, and returns. Otherwise it loads and authenticates the 16-byte PCALL record at SS:SP. Authentication precedes target validation and any architectural change.

Encodings:

`PRET`

Instruction Format:



Format: Instruction format for PRET

21 FLOATING-POINT INSTRUCTIONS

21.1 Summary

Table 21-1. Floating-Point Instructions Summary (Informative)

Mnemonic	Brief description
FABS	Clears the selected floating-point source sign and writes its magnitude.
FADD	Adds the source operand to the destination operand.
FBNDII	Checks floating-point value membership in the inclusive-inclusive interval [lo, hi].
FBNDIX	Checks floating-point value membership in the inclusive-exclusive interval [lo, hi).
FBNDXI	Checks floating-point value membership in the exclusive-inclusive interval (lo, hi].
FBNDXX	Checks floating-point value membership in the exclusive-exclusive interval (lo, hi).
FCEIL	Rounds a floating-point source toward positive infinity in the selected format.
FCLASS	Classifies an S- or D-format source into a one-hot integer bitmap without changing FFLAGS.
FCLR	Writes positive zero to the complete 64-bit floating-point destination.
FCMP	Compares two floating-point operands and writes their relation to integer condition flags.
FCOPYSIGN	Combines a magnitude source with the sign bit of a separate floating-point source.
FCVTD	Converts between D and signed B/W/L/Q integer or H/S floating-point representations.
FCVTH	Converts between H and signed B/W/L/Q integer or S/D floating-point representations.
FCVTS	Converts between S and signed B/W/L/Q integer or H/D floating-point representations.
FCVTUD	Converts between D and unsigned B/W/L/Q integers.
FCVTUH	Converts between H and unsigned B/W/L/Q integers.
FCVTUS	Converts between S and unsigned B/W/L/Q integers.
FDIV	Divides the destination value by the source under the floating-point environment.
FFLOOR	Rounds a floating-point source toward negative infinity in the selected format.
FGETEXP	Converts the source encoding's signed exponent to the selected floating-point format.
FGETMAN	Produces the normalized significand view of a floating-point source.
FINT	Rounds a floating-point source to an integral value using FSTATUS.RM.
FINTRZ	Rounds a floating-point source toward zero to an integral value.
FMADD	Computes the source product plus the old destination as one fused operation and rounds once.
FMAX	Selects the greater numeric source or destination value using IEEE NaN and zero rules.
FMIN	Selects the lesser numeric source or destination value using IEEE NaN and zero rules.
FMOD	Computes the destination modulo the source using a quotient truncated toward zero.
FMOV	Copies the source operand to the destination operand.
FMOVcc	Copies a floating-point source only when the encoded integer condition is true.
FMOVCR	Loads a named architectural binary64 constant selected by an unsigned 16-bit ID.
FMSUB	Computes the source product minus the old destination as one fused operation and rounds once.
FMUL	Multiplies the source and destination values under the floating-point environment.
FNEG	Complements the selected floating-point source sign bit.
FNMADD	Computes the negated source product minus the old destination as one fused operation and rounds once.
FNMSUB	Computes the old destination minus the source product as one fused operation and rounds once.
FPOPP	Pops one canonical floating-point register pair selected by an inline pair index.
FPUSHP	Pushes one canonical floating-point register pair selected by an inline pair index.

Table 21-1 (continued)

Mnemonic	Brief description
FREM	Computes the destination remainder using a nearest-even integral quotient.
FRESTORE	Restores the floating-point user-state record.
FROUND	Rounds a floating-point source to a nearest-even integral value independently of FSTATUS.RM.
FSAVE	Saves the floating-point user-state record.
FSCALE	Scales the destination by a power of two derived from the rounded source.
FSQRT	Computes the selected-format square root of a floating-point source.
FSUB	Subtracts the source operand from the destination operand.
FTEST	Compares a floating-point source with positive zero and writes integer condition flags.
FTRUNC	Rounds a floating-point source toward zero in the same format.
FXCHG	Exchanges the complete 64-bit images of two floating-point registers.
RDFFLAGS	Read the accrued floating-point exception flags into an Rn register.
RDFSTATUS	Read the FPU status/control register into an Rn register.
WRFFLAGS	Write the accrued floating-point exception flags from an Rn register.
WRFSTATUS	Write the FPU status/control register from an Rn register.

21.2 Common Floating-Point Semantics

The S and D suffixes select 32- and 64-bit floating-point scalar arithmetic sizes. H selects the 16-bit scalar conversion and storage format only. Conditional floating-point moves place the integer condition suffix in the mnemonic.

The H, S, and D formats are IEEE-754 binary16, binary32, and binary64. H uses sign bit 15, exponent bits 14..10, and fraction bits 9..0. S uses sign bit 31, exponent bits 30..23, and fraction bits 22..0. D uses sign bit 63, exponent bits 62..52, and fraction bits 51..0. An all-ones exponent and nonzero fraction is a NaN. The most significant fraction bit distinguishes a quiet NaN from a signaling NaN: zero is signaling and one is quiet. The architectural default quiet NaNs are 0x7e00 for H, 0x7fc00000 for S, and 0x7ff8000000000000 for D. Figure 4-4 shows both encodings in their 64-bit Fn register view.

An H conversion reads bits 15..0 of an H-valued Fn operand and an H result write writes zero to bits 63..16. An S operation reads bits 31..0 of each Fn operand. Writing an S result to Fn writes the result to bits 31..0 and writes zero to bits 63..32. A D operation reads and writes all 64 bits. Operations use the selected format for their intermediate and final values. FMADD, FMSUB, FNMADD, and FNMSUB form the complete multiply-add expression with unbounded intermediate range and precision and round only the final result. Other arithmetic operations round once after computing their exact mathematical result.

An encoded FEA source immsf is an exact IEEE 754 binary32 payload, and immdf is an exact binary64 payload. Either immediate may be used by either an S or D instruction. If the payload format differs from the operation format selected by the suffix, the source is converted to the operation format using the current FSTATUS.RM before the operation. All conversion causes, including causes from NaN handling and narrowing, are combined with the operation's causes. The combined cause set follows the same enabled-cause fault test, FFLAGS accrual, and all-or-nothing destination commit rules below. Floating-point immediate forms are source-only.

21.2.1 Input, NaN, Rounding, and Result Order

Each ordinary floating-point operation applies the following order.

1. Read all operands and classify zero, subnormal, normal, infinity, quiet NaN, and signaling NaN.
2. If `FSTATUS.DAZ` is set, replace each subnormal input with a zero having the same sign.
3. Determine signaling-NaN and operation-specific floating-point exception causes and form the exact mathematical result.
4. For a NaN result, select the first signaling NaN in source-operand order, or the first quiet NaN when there is no signaling NaN, and set its quiet bit. If the operation produces a NaN without a NaN operand, use the default quiet NaN. When `FSTATUS.DN` is set, replace the selected NaN with the default quiet NaN.
5. Round a numeric result to the selected format using `FSTATUS.RM`, except where an instruction names a fixed rounding direction.
6. Detect overflow and tininess after rounding. UF is generated when the rounded result is tiny and inexact.
7. If `FSTATUS.FTZ` is set and the rounded result is a nonzero subnormal, replace it with a zero of the same sign and generate both UF and NX.
8. Test all generated causes against their FSTATUS enable bits and then perform the architectural commit.

A signaling NaN generates NV. A quiet NaN does not generate NV unless the operation itself is invalid. When two NaN operands have the same priority, source-operand order determines the selected sign and payload. Quieting sets the quiet bit and preserves the remaining selected sign and payload bits.

Overflow generates OF and NX. Nearest-even produces signed infinity. Toward zero produces the signed maximum finite value. Toward positive produces positive infinity for a positive result and the negative maximum finite value for a negative result. Toward negative produces negative infinity for a negative result and the positive maximum finite value for a positive result.

21.2.2 Floating-Point Exception Causes and Commit

Cause	Generated when
`NV`	an input is a signaling NaN; an arithmetic operation has no defined numeric result, including infinity minus the same infinity, zero times infinity, zero divided by zero, infinity divided by infinity, or square root of a negative nonzero value; or a floating-to-integer conversion is invalid
`DZ`	a finite nonzero value is divided by zero
`OF`	the rounded numeric magnitude exceeds the selected format's finite range
`UF`	the result is tiny after rounding and inexact, or FTZ flushes a nonzero subnormal result
`NX`	rounding or a valid conversion changes the exact value, or OF or FTZ generates NX

If any generated floating-point exception cause has its matching FSTATUS enable bit set, the instruction raises the FLOATING_POINT_EXCEPTION architectural event. Its error-code bitmap contains every cause generated by the operation. The destination, integer FLAGS, and FFLAGS remain unchanged, and arithmetic instructions never modify FSTATUS.

If no generated floating-point exception cause is enabled, the instruction commits its destination and any complete integer FLAGS image defined by the instruction, and ORs every generated cause into FFLAGS. FFLAGS fields not named by the instruction remain unchanged. An instruction-specific rule may exclude a cause.

21.2.3 Comparison, Minimum, and Maximum

FCMP and FTEST write Z,N,C,V as follows.

Relation	Z	N	C	V
greater	0	0	0	0
equal	1	0	0	0
less	0	1	1	0
unordered	0	0	0	1

A quiet NaN produces unordered without NV. A signaling NaN produces unordered and NV. FTEST compares its operand with positive zero under these rules.

For FMIN and FMAX, when exactly one operand is a NaN the numeric operand is the result. When both operands are NaNs, the common NaN-selection rule supplies the result. A signaling NaN generates NV even when the other operand is numeric. Between positive and negative zero, FMIN selects negative zero and FMAX selects positive zero.

21.2.4 Conversions

The final letter of FCVTH, FCVTS, and FCVTD names the H, S, or D format associated with the Fn side of a conversion. The suffix names the other representation. For Fn-to-Fn forms, the source has the suffix format and the destination has the mnemonic format; for example, FCVTS.D converts D to S and FCVTD.H converts H to D. For a form containing one Fn and one Rn or EA operand, Fn has the mnemonic format and Rn or EA has the suffix representation. Source-first operand order therefore permits both directions. The H/D, S/D, or H/S conversion is direct and does not pass through an intermediate floating format.

The .B, .W, .L, and .Q suffixes select 8-, 16-, 32-, and 64-bit integers. FCVTH, FCVTS, and FCVTD interpret those integers as signed; FCVTUH, FCVTUS, and FCVTUD interpret them as unsigned. An Rn integer source uses the selected number of low bits. A signed Rn result is sign-extended to 64 bits and an unsigned result is zero-extended. An EA access transfers exactly the selected number of bytes. Floating-format EA conversion operands are memory operands; floating immediates are not accepted by these forms.

Integer-to-floating and floating-to-floating conversion uses FSTATUS.RM. Floating-to-integer conversion truncates toward zero. A valid conversion that discards a fractional part generates NX. An invalid conversion generates NV without OF or NX. When NV is not enabled, a signed negative

overflow or negative infinity produces the least value of the selected signed width, a signed positive overflow or positive infinity produces the greatest value, and NaN produces zero. An unsigned negative value or negative infinity produces zero, positive overflow or positive infinity produces the greatest value of the selected unsigned width, and NaN produces zero.

Conversions generate only NV, OF, UF, and NX; DZ is unchanged. An enabled generated cause raises FLOATING_POINT_EXCEPTION and suppresses the destination write and newly accrued FFLAGS effects. A memory fault likewise suppresses the destination and floating-point state changes.

FABS — Floating-Point Absolute Value — FABS

Operation: Clears the selected floating-point source sign and writes its magnitude.

Assembler Syntax:
```
FABS.{S|D}(z) Fn(s), Fn(d)
FABS.{S|D}(z) <ea>(e), Fn(d)
FABS.{S|D}(z) Fn(s), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: FP

Repeat eligibility: REP eligible

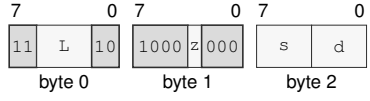
Detailed Semantics

Clears the sign of the selected floating-point source value and writes the resulting magnitude to the destination.

Encodings:

`FABS.{S|D}(z) Fn(s), Fn(d)`

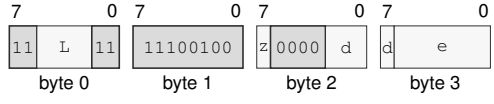
Instruction Format:



Format: Instruction format for FABS.{S|D}(z) Fn(s), Fn(d)

`FABS.{S|D}(z) <ea>(e), Fn(d)`

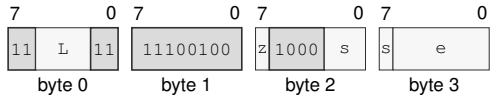
Instruction Format:



Format: Instruction format for FABS.{S|D}(z) <ea>(e), Fn(d)

`FABS.{S|D}(z) Fn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FABS.{S|D}(z) Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

FADD — Floating-Point Add — FADD

Operation: Adds the source operand to the destination operand.

Assembler Syntax:
```
FADD.{S|D}(z) Fn(s), Fn(d)
FADD.{S|D}(z) <ea>(e), Fn(d)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

Detailed Semantics

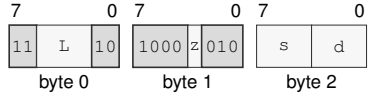
FADD adds the selected-format floating-point source to the destination and rounds the result according to the floating-point environment. A completed operation writes the rounded value to the destination and accrues any generated NV, OF, UF, and NX causes in FFLAGS; DZ is unchanged.

If a generated floating-point exception is enabled, FADD raises FLOATING_POINT_EXCEPTION. The faulting operation commits neither its destination write nor its accrued FFLAGS causes.

Encodings:

`FADD.{S|D}(z) Fn(s), Fn(d)`

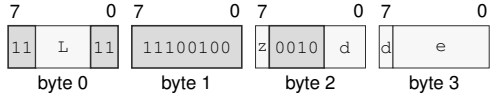
Instruction Format:



Format: Instruction format for FADD.{S|D}(z) Fn(s), Fn(d)

`FADD.{S|D}(z) <ea>(e), Fn(d)`

Instruction Format:



Format: Instruction format for FADD.{S|D}(z) <ea>(e), Fn(d)

FBNDII Floating-Point Bounds Check Inclusive-Inclusive FBNDII

Operation: Checks floating-point value membership in the inclusive-inclusive interval [lo, hi].

Assembler Syntax:
```
FBNDII.{S|D}(z) Fn(l), Fn(v), Fn(h)
FBNDII.{S|D}(z) Fn(l), <ea>(v), Fn(h)
FBNDII.{S|D}(z) <ea>(l), Fn(v), <ea>(h)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

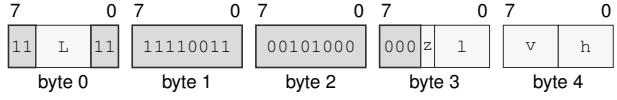
Detailed Semantics

FBNDII treats both bounds as inclusive. It sets `FLAGS.V` when the value is unordered or outside [lo, hi] and clears `FLAGS.Z`, `FLAGS.N`, and `FLAGS.C`. IEEE-754 FFLAGS causes continue to accrue independently.

Encodings:

`FBNDII.{S|D}(z) Fn(l), Fn(v), Fn(h)`

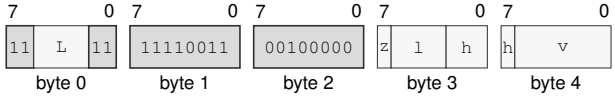
Instruction Format:



Format: Instruction format for FBNDII.{S|D}(z) Fn(l), Fn(v), Fn(h)

`FBNDII.{S|D}(z) Fn(l), <ea>(v), Fn(h)`

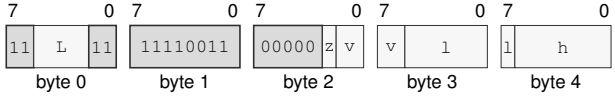
Instruction Format:



Format: Instruction format for FBNDII.{S|D}(z) Fn(l), <ea>(v), Fn(h)

`FBNDII.{S|D}(z) <ea>(l), Fn(v), <ea>(h)`

Instruction Format:



Format: Instruction format for FBNDII.{S|D}(z) <ea>(l), Fn(v), <ea>(h)

FBNDIX Floating-Point Bounds Check Inclusive-Exclusive FBNDIX

Operation: Checks floating-point value membership in the inclusive-exclusive interval [lo, hi).

Assembler Syntax:
```
FBNDIX.{S|D}(z) Fn(l), Fn(v), Fn(h)
FBNDIX.{S|D}(z) Fn(l), <ea>(v), Fn(h)
FBNDIX.{S|D}(z) <ea>(l), Fn(v), <ea>(h)
```

Privilege: Unprivileged

Required CPUID flags: FP

Repeat eligibility: REP eligible

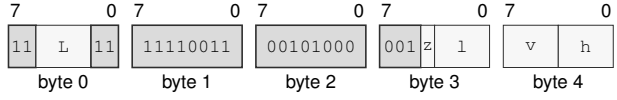
Detailed Semantics

FBNDIX treats the low bound as inclusive and the high bound as exclusive. It sets FLAGS.V when the value is unordered or outside [lo, hi) and clears FLAGS.Z, FLAGS.N, and FLAGS.C. IEEE-754 FFLAGS causes continue to accrue independently.

Encodings:

FBNDIX.{S|D}(z) Fn(l), Fn(v), Fn(h)

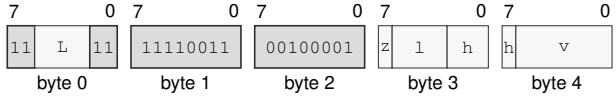
Instruction Format:



Format: Instruction format for FBNDIX.{S|D}(z) Fn(l), Fn(v), Fn(h)

`FBNDIX.{S|D}(z) Fn(l), <ea>(v), Fn(h)`

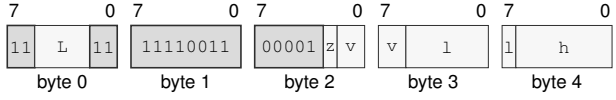
Instruction Format:



Format: Instruction format for FBNDIX.{S|D}(z) Fn(l), <ea>(v), Fn(h)

`FBNDIX.{S|D}(z) <ea>(l), Fn(v), <ea>(h)`

Instruction Format:



Format: Instruction format for FBNDIX.{S|D}(z) <ea>(l), Fn(v), <ea>(h)

FBNDXI Floating-Point Bounds Check Exclusive-Inclusive FBNDXI

Operation: Checks floating-point value membership in the exclusive-inclusive interval (lo, hi].

Assembler Syntax:
```
FBNDXI.{S|D}(z) Fn(l), Fn(v), Fn(h)
FBNDXI.{S|D}(z) Fn(l), <ea>(v), Fn(h)
FBNDXI.{S|D}(z) <ea>(l), Fn(v), <ea>(h)
```

Privilege: Unprivileged

Required CPUID flags: FP

Repeat eligibility: REP eligible

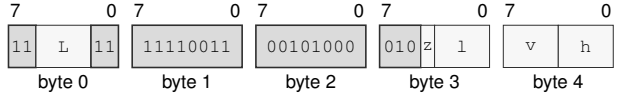
Detailed Semantics

FBNDXI treats the low bound as exclusive and the high bound as inclusive. It sets FLAGS.V when the value is unordered or outside (lo, hi] and clears FLAGS.Z, FLAGS.N, and FLAGS.C. IEEE-754 FFLAGS causes continue to accrue independently.

Encodings:

FBNDXI.{S|D}(z) Fn(l), Fn(v), Fn(h)

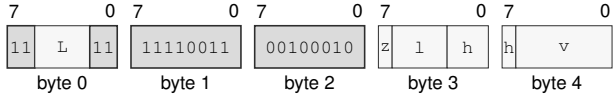
Instruction Format:



Format: Instruction format for FBNDXI.{S|D}(z) Fn(l), Fn(v), Fn(h)

`FBNDXI.{S|D}(z) Fn(l), <ea>(v), Fn(h)`

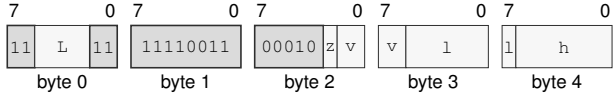
Instruction Format:



Format: Instruction format for FBNDXI.{S|D}(z) Fn(l), <ea>(v), Fn(h)

`FBNDXI.{S|D}(z) <ea>(l), Fn(v), <ea>(h)`

Instruction Format:



Format: Instruction format for FBNDXI.{S|D}(z) <ea>(l), Fn(v), <ea>(h)

FBNDXX Floating-Point Bounds Check Exclusive-Exclusive FBNDXX

Operation: Checks floating-point value membership in the exclusive-exclusive interval (lo, hi).

Assembler Syntax:
```
FBNDXX.{S|D}(z) Fn(l), Fn(v), Fn(h)
FBNDXX.{S|D}(z) Fn(l), <ea>(v), Fn(h)
FBNDXX.{S|D}(z) <ea>(l), Fn(v), <ea>(h)
```

Privilege: Unprivileged

Required CPUID flags: FP

Repeat eligibility: REP eligible

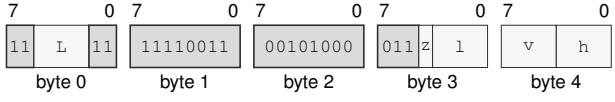
Detailed Semantics

FBNDXX treats both bounds as exclusive. It sets FLAGS.V when the value is unordered or outside (lo, hi) and clears FLAGS.Z, FLAGS.N, and FLAGS.C. IEEE-754 FFLAGS causes continue to accrue independently.

Encodings:

FBNDXX.{S|D}(z) Fn(l), Fn(v), Fn(h)

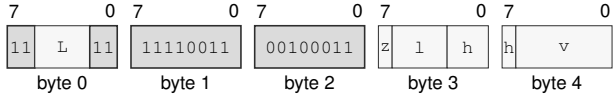
Instruction Format:



Format: Instruction format for FBNDXX.{S|D}(z) Fn(l), Fn(v), Fn(h)

`FBNDXX.{S|D}(z) Fn(l), <ea>(v), Fn(h)`

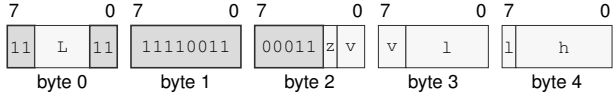
Instruction Format:



Format: Instruction format for FBNDXX.{S|D}(z) Fn(l), <ea>(v), Fn(h)

`FBNDXX.{S|D}(z) <ea>(l), Fn(v), <ea>(h)`

Instruction Format:



Format: Instruction format for FBNDXX.{S|D}(z) <ea>(l), Fn(v), <ea>(h)

FCEIL — Floating-Point Ceiling — FCEIL

Operation: Rounds a floating-point source toward positive infinity in the selected format.

Assembler Syntax:
```
FCEIL.{S|D}(z) Fn(s), Fn(d)
FCEIL.{S|D}(z) <ea>(e), Fn(d)
FCEIL.{S|D}(z) Fn(s), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: FP

Repeat eligibility: REP eligible

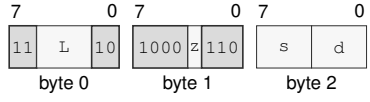
Detailed Semantics

Rounds the selected floating-point source toward positive infinity to an integral floating-point value.

Encodings:

`FCEIL.{S|D}(z) Fn(s), Fn(d)`

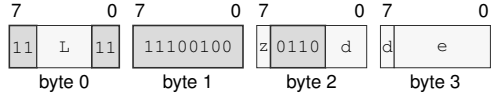
Instruction Format:



Format: Instruction format for FCEIL.{S|D}(z) Fn(s), Fn(d)

`FCEIL.{S|D}(z) <ea>(e), Fn(d)`

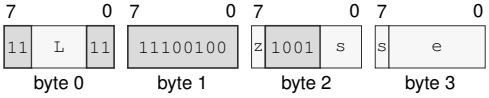
Instruction Format:



Format: Instruction format for FCEIL.{S|D}(z) <ea>(e), Fn(d)

`FCEIL.{S|D}(z) Fn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FCEIL.{S|D}(z) Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

FCLASS Floating-Point Classify FCLASS

Operation: Classifies an S- or D-format source into a one-hot integer bitmap without changing FFLAGS.

Assembler Syntax: `FCLASS.{S|D}(z) Fn(s), Rn(d)`

Privilege: Unprivileged

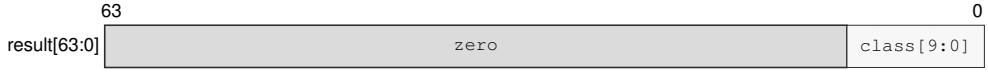
Required CPUID flags: `FP`

Repeat eligibility: REP eligible

Detailed Semantics

Examines the selected floating-point encoding without generating a floating-point exception condition and writes a one-hot class bitmap to the destination Rn register.

The destination is a one-hot bitmap selected directly from the sign, exponent, fraction, and NaN quiet bit of the S or D source encoding.



FCLASS Result Bitmap

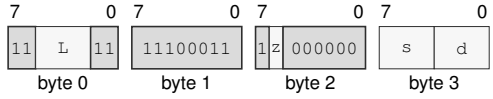
Bit	Class
0	negative infinity
1	negative normal
2	negative subnormal
3	negative zero
4	positive zero
5	positive subnormal
6	positive normal
7	positive infinity
8	signaling NaN
9	quiet NaN

Bits 63..10 of the destination Rn register are zero. No floating-point exception flag is changed.

Encodings:

```
FCLASS.{S|D}(z) Fn(s), Rn(d)
```

Instruction Format:



Format: Instruction format for FCLASS.{S|D}(z) Fn(s), Rn(d)

FCLR — Floating-Point Clear — FCLR

Operation: Writes positive zero to the complete 64-bit floating-point destination.

Assembler Syntax: `FCLR Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

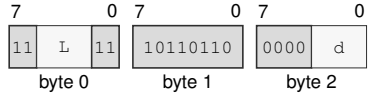
Detailed Semantics

Writes all zero bits to the complete 64-bit Fn destination, representing positive zero in both S and D interpretations.

Encodings:

`FCLR Fn(d)`

Instruction Format:



Format: Instruction format for FCLR Fn(d)

FCMP — Floating-Point Compare — FCMP

Operation: Compares two floating-point operands and writes their relation to integer condition flags.

Assembler Syntax:
```
FCMP.{S|D}(z) Fn(s), Fn(d)
FCMP.{S|D}(z) <ea>(e), Fn(d)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

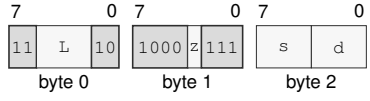
Detailed Semantics

Compares the selected-format operands and writes `FLAGS.Z`, `FLAGS.N`, `FLAGS.C`, and `FLAGS.V` as greater=0000, equal=1000, less=0110, or unordered=0001. A quiet NaN is unordered; a signaling NaN is unordered and generates NV.

Encodings:

`FCMP.{S|D}(z) Fn(s), Fn(d)`

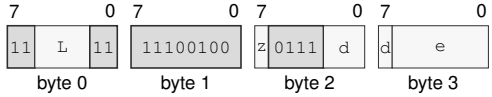
Instruction Format:



Format: Instruction format for FCMP.{S|D}(z) Fn(s), Fn(d)

`FCMP.{S|D}(z) <ea>(e), Fn(d)`

Instruction Format:



Format: Instruction format for FCMP.{S|D}(z) <ea>(e), Fn(d)

FCOPYSIGN Floating-Point Copy Sign FCOPYSIGN

Operation: Combines a magnitude source with the sign bit of a separate floating-point source.

Assembler Syntax: `FCOPYSIGN.{S|D}(z) Fn(s), Fn(m), Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

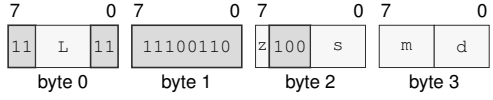
Detailed Semantics

Combines the magnitude bits of the magnitude source with the sign bit of the sign source and writes the selected-format result.

Encodings:

`FCOPYSIGN.{S|D}(z) Fn(s), Fn(m), Fn(d)`

Instruction Format:



Format: Instruction format for FCOPYSIGN.{S|D}(z) Fn(s), Fn(m), Fn(d)

FCVTD — Convert to Double Precision — FCVTD

Operation: Converts between D and signed B/W/L/Q integer or H/S floating-point representations.

Assembler Syntax:

```
FCVTD.H Fn(s), Fn(d)
FCVTD.S Fn(s), Fn(d)
FCVTD.{B|W|L|Q}(z) Fn(s), Rn(d)
FCVTD.{B|W|L|Q}(z) Rn(s), Fn(d)
FCVTD.{B|W|L|Q}(z) <ea>(e), Fn(d)
FCVTD.{B|W|L|Q}(z) Fn(s), <ea>(e)
FCVTD.H <ea>(e), Fn(d)
FCVTD.S <ea>(e), Fn(d)
FCVTD.H Fn(s), <ea>(e)
FCVTD.S Fn(s), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

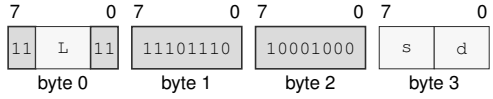
Detailed Semantics

Converts between the D format in an Fn operand and the representation selected by the suffix. The integer suffixes are signed B/W/L/Q; `.H` and `.S` select binary16 and binary32. For Fn-to-Fn conversion, the source has the suffix format and the destination has D format. With one Fn and one Rn or memory operand, the Fn has D format and the other operand has the suffix representation. Floating-format EA sources are memory-only.

Encodings:

`FCVTD.H Fn(s), Fn(d)`

Instruction Format:

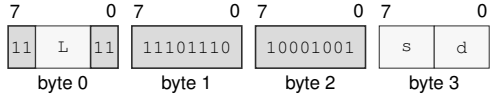


Format: Instruction format for FCVTD.H Fn(s), Fn(d)

Additional CPUID flags: `FP16_CONVERT`

`FCVTD.S Fn(s), Fn(d)`

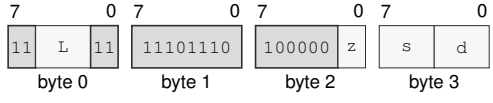
Instruction Format:



Format: Instruction format for FCVTD.S Fn(s), Fn(d)

`FCVTD.{B|W|L|Q}(z) Fn(s), Rn(d)`

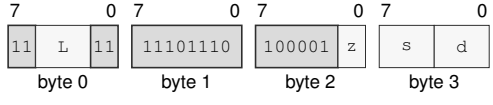
Instruction Format:



Format: Instruction format for FCVTD.{B|W|L|Q}(z) Fn(s), Rn(d)

`FCVTD.{B|W|L|Q}(z) Rn(s), Fn(d)`

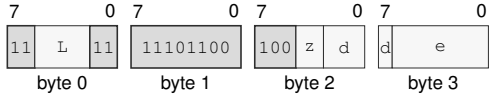
Instruction Format:



Format: Instruction format for FCVTD.{B|W|L|Q}(z) Rn(s), Fn(d)

`FCVTD.{B|W|L|Q}(z) <ea>(e), Fn(d)`

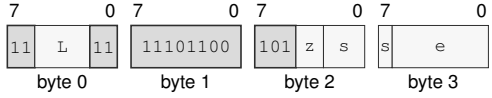
Instruction Format:



Format: Instruction format for FCVTD.{B|W|L|Q}(z) <ea>(e), Fn(d)

`FCVTD.{B|W|L|Q}(z) Fn(s), <ea>(e)`

Instruction Format:



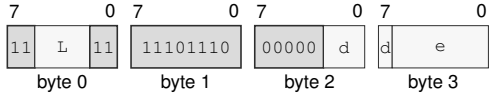
Format: Instruction format for FCVTD.{B|W|L|Q}(z) Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`FCVTD.H <ea>(e), Fn(d)`

Instruction Format:



Format: Instruction format for FCVTD.H <ea>(e), Fn(d)

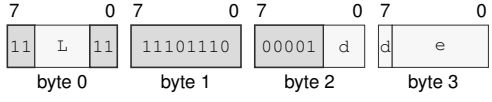
Additional CPUID flags: `FP16_CONVERT`

Restrictions:

Operand constraint: Role `src`; relation `excluded`; values `immediate`; reason `invalid_source`.

`FCVTD.S <ea>(e), Fn(d)`

Instruction Format:



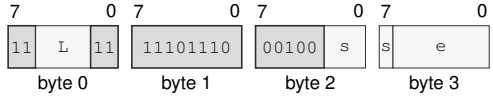
Format: Instruction format for FCVTD.S <ea>(e), Fn(d)

Restrictions:

Operand constraint: Role `src`; relation `excluded`; values `immediate`; reason `invalid_source`.

`FCVTD.H Fn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FCVTD.H Fn(s), <ea>(e)

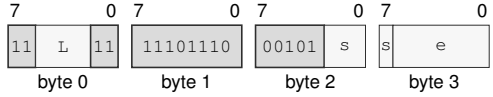
Additional CPUID flags: `FP16_CONVERT`

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`FCVTD.S Fn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FCVTD.S Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

FCVTH — Convert to Half Precision — FCVTH

Operation: Converts between H and signed B/W/L/Q integer or S/D floating-point representations.

Assembler Syntax:
```
FCVTH.{S|D}(z) Fn(s), Fn(d)
FCVTH.{B|W|L|Q}(z) Fn(s), Rn(d)
FCVTH.{B|W|L|Q}(z) Rn(s), Fn(d)
FCVTH.{B|W|L|Q}(z) <ea>(e), Fn(d)
FCVTH.{B|W|L|Q}(z) Fn(s), <ea>(e)
FCVTH.{S|D}(z) <ea>(e), Fn(d)
FCVTH.{S|D}(z) Fn(s), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `FP`
`FP16_CONVERT`

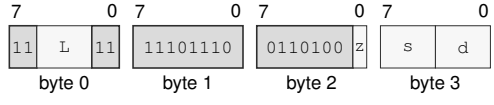
Repeat eligibility: REP eligible

Detailed Semantics

Converts between the H conversion format in an Fn operand and the representation selected by the suffix. The integer suffixes are signed B/W/L/Q; `.S` and `.D` select binary32 and binary64. For Fn-to-Fn conversion, the source has the suffix format and the destination has H format. With one Fn and one Rn or memory operand, the Fn has H format and the other operand has the suffix representation. Floating-format EA sources are memory-only. H is a conversion and storage format; this instruction does not add scalar H arithmetic.

Encodings:

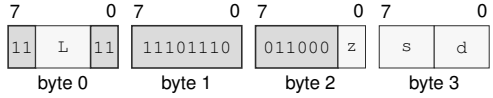
`FCVTH.{S|D}(z) Fn(s), Fn(d)`
Instruction Format:



Format: Instruction format for FCVTH.{S|D}(z) Fn(s), Fn(d)

`FCVTH.{B|W|L|Q}(z) Fn(s), Rn(d)`

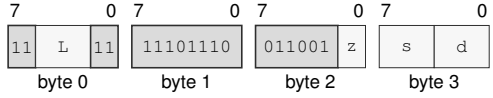
Instruction Format:



Format: Instruction format for FCVTH.{B|W|L|Q}(z) Fn(s), Rn(d)

`FCVTH.{B|W|L|Q}(z) Rn(s), Fn(d)`

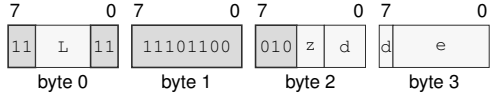
Instruction Format:



Format: Instruction format for FCVTH.{B|W|L|Q}(z) Rn(s), Fn(d)

`FCVTH.{B|W|L|Q}(z) <ea>(e), Fn(d)`

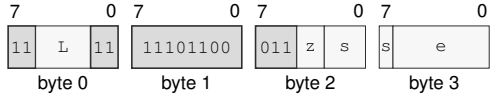
Instruction Format:



Format: Instruction format for FCVTH.{B|W|L|Q}(z) <ea>(e), Fn(d)

`FCVTH.{B|W|L|Q}(z) Fn(s), <ea>(e)`

Instruction Format:



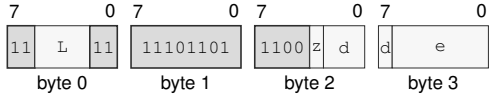
Format: Instruction format for FCVTH.{B|W|L|Q}(z) Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`FCVTH.{S|D}(z) <ea>(e), Fn(d)`

Instruction Format:



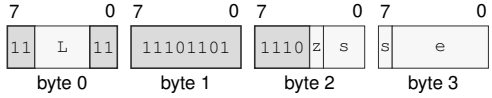
Format: Instruction format for FCVTH.{S|D}(z) <ea>(e), Fn(d)

Restrictions:

Operand constraint: Role `src`; relation `excluded`; values `immediate`; reason `invalid_source`.

`FCVTH.{S|D}(z) Fn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FCVTH.{S|D}(z) Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

FCVTS — Convert to Single Precision — FCVTS

Operation: Converts between S and signed B/W/L/Q integer or H/D floating-point representations.

Assembler Syntax:

```
FCVTS.H Fn(s), Fn(d)
FCVTS.D Fn(s), Fn(d)
FCVTS.{B|W|L|Q}(z) Fn(s), Rn(d)
FCVTS.{B|W|L|Q}(z) Rn(s), Fn(d)
FCVTS.{B|W|L|Q}(z) <ea>(e), Fn(d)
FCVTS.{B|W|L|Q}(z) Fn(s), <ea>(e)
FCVTS.H <ea>(e), Fn(d)
FCVTS.D <ea>(e), Fn(d)
FCVTS.H Fn(s), <ea>(e)
FCVTS.D Fn(s), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

Detailed Semantics

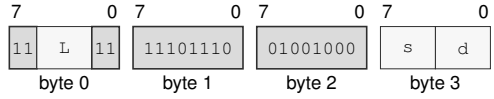
Converts between the S format in an Fn operand and the representation selected by the suffix. The `.B`, `.W`, `.L`, and `.Q` forms use signed 8-, 16-, 32-, and 64-bit integers; the `.H` and `.D` forms use binary16 and binary64.

For Fn-to-Fn conversion, the source has the suffix format and the destination has S format. With one Fn and one Rn or memory operand, the Fn has S format and the other operand has the suffix representation. Operand order selects the conversion direction. Floating-format EA sources are memory-only.

Encodings:

`FCVTS.H Fn(s), Fn(d)`

Instruction Format:

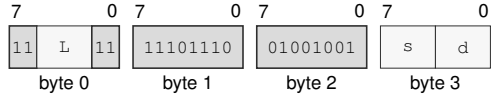


Format: Instruction format for FCVTS.H Fn(s), Fn(d)

Additional CPUID flags: `FP16_CONVERT`

`FCVTS.D Fn(s), Fn(d)`

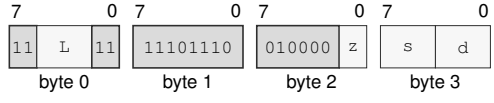
Instruction Format:



Format: Instruction format for FCVTS.D Fn(s), Fn(d)

`FCVTS.{B|W|L|Q}(z) Fn(s), Rn(d)`

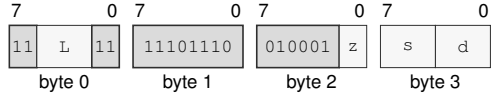
Instruction Format:



Format: Instruction format for FCVTS.{B|W|L|Q}(z) Fn(s), Rn(d)

`FCVTS.{B|W|L|Q}(z) Rn(s), Fn(d)`

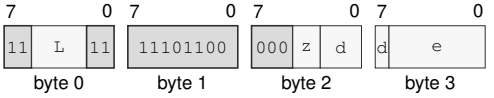
Instruction Format:



Format: Instruction format for FCVTS.{B|W|L|Q}(z) Rn(s), Fn(d)

`FCVTS.{B|W|L|Q}(z) <ea>(e), Fn(d)`

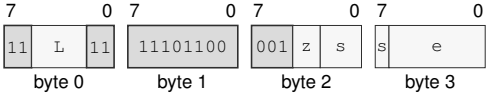
Instruction Format:



Format: Instruction format for FCVTS.{B|W|L|Q}(z) <ea>(e), Fn(d)

`FCVTS.{B|W|L|Q}(z) Fn(s), <ea>(e)`

Instruction Format:



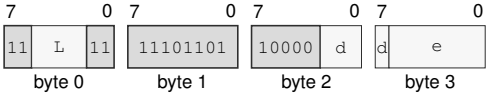
Format: Instruction format for FCVTS.{B|W|L|Q}(z) Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`FCVTS.H <ea>(e), Fn(d)`

Instruction Format:



Format: Instruction format for FCVTS.H <ea>(e), Fn(d)

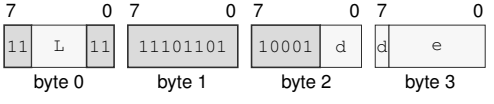
Additional CPUID flags: `FP16_CONVERT`

Restrictions:

Operand constraint: Role `src`; relation `excluded`; values `immediate`; reason `invalid_source`.

`FCVTS.D <ea>(e), Fn(d)`

Instruction Format:



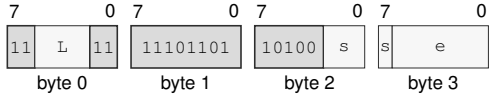
Format: Instruction format for FCVTS.D <ea>(e), Fn(d)

Restrictions:

Operand constraint: Role `src`; relation `excluded`; values `immediate`; reason `invalid_source`.

`FCVTS.H Fn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FCVTS.H Fn(s), <ea>(e)

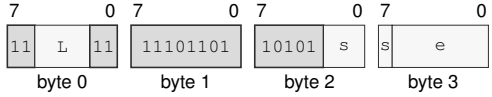
Additional CPUID flags: `FP16_CONVERT`

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`FCVTS.D Fn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FCVTS.D Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

FCVTUD — Convert Unsigned and Double Precision — FCVTUD

Operation: Converts between D and unsigned B/W/L/Q integers.

Assembler Syntax:

```
FCVTUD.{B|W|L|Q}(z) Fn(s), Rn(d)
FCVTUD.{B|W|L|Q}(z) Rn(s), Fn(d)
FCVTUD.{B|W|L|Q}(z) <ea>(e), Fn(d)
FCVTUD.{B|W|L|Q}(z) Fn(s), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

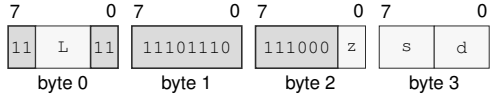
Detailed Semantics

Converts between D in an Fn operand and an unsigned integer of the suffix width in an Rn or memory operand. Integer-to-D conversion uses `FSTATUS.RM`; D-to-integer conversion truncates toward zero.

Encodings:

`FCVTUD.{B|W|L|Q}(z) Fn(s), Rn(d)`

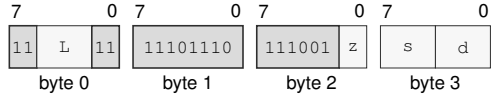
Instruction Format:



Format: Instruction format for FCVTUD.{B|W|L|Q}(z) Fn(s), Rn(d)

`FCVTUD.{B|W|L|Q}(z) Rn(s), Fn(d)`

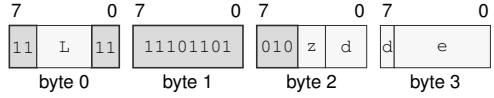
Instruction Format:



Format: Instruction format for FCVTUD.{B|W|L|Q}(z) Rn(s), Fn(d)

`FCVTUD.{B|W|L|Q}(z) <ea>(e), Fn(d)`

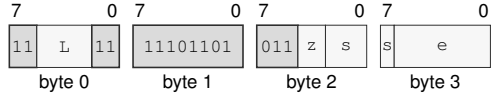
Instruction Format:



Format: Instruction format for FCVTUD.{B|W|L|Q}(z) <ea>(e), Fn(d)

`FCVTUD.{B|W|L|Q}(z) Fn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FCVTUD.{B|W|L|Q}(z) Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

FCVTUH — Convert Unsigned and Half Precision — FCVTUH

Operation: Converts between H and unsigned B/W/L/Q integers.

Assembler Syntax:

```
FCVTUH.{B|W|L|Q}(z) Fn(s), Rn(d)
FCVTUH.{B|W|L|Q}(z) Rn(s), Fn(d)
FCVTUH.{B|W|L|Q}(z) <ea>(e), Fn(d)
FCVTUH.{B|W|L|Q}(z) Fn(s), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `FP`
`FP16_CONVERT`

Repeat eligibility: REP eligible

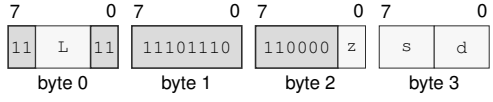
Detailed Semantics

Converts between the H conversion format in an Fn operand and an unsigned integer of the suffix width in an Rn or memory operand. Integer-to-H conversion uses `FSTATUS.RM`; H-to-integer conversion truncates toward zero.

Encodings:

`FCVTUH.{B|W|L|Q}(z) Fn(s), Rn(d)`

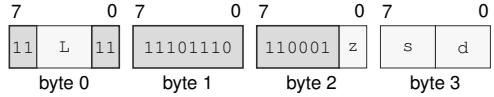
Instruction Format:



Format: Instruction format for FCVTUH.{B|W|L|Q}(z) Fn(s), Rn(d)

`FCVTUH.{B|W|L|Q}(z) Rn(s), Fn(d)`

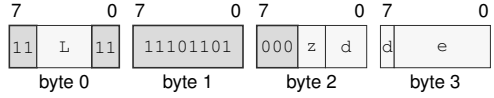
Instruction Format:



Format: Instruction format for FCVTUH.{B|W|L|Q}(z) Rn(s), Fn(d)

`FCVTUH.{B|W|L|Q}(z) <ea>(e), Fn(d)`

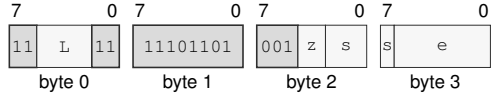
Instruction Format:



Format: Instruction format for FCVTUH.{B|W|L|Q}(z) <ea>(e), Fn(d)

`FCVTUH.{B|W|L|Q}(z) Fn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FCVTUH.{B|W|L|Q}(z) Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

FCVTUS — Convert Unsigned and Single Precision — FCVTUS

Operation: Converts between S and unsigned B/W/L/Q integers.

Assembler Syntax:

```
FCVTUS.{B|W|L|Q}(z) Fn(s), Rn(d)
FCVTUS.{B|W|L|Q}(z) Rn(s), Fn(d)
FCVTUS.{B|W|L|Q}(z) <ea>(e), Fn(d)
FCVTUS.{B|W|L|Q}(z) Fn(s), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

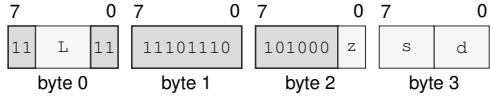
Detailed Semantics

Converts between S in an Fn operand and an unsigned integer of the suffix width in an Rn or memory operand. Integer-to-S conversion uses `FSTATUS.RM`; S-to-integer conversion truncates toward zero.

Encodings:

`FCVTUS.{B|W|L|Q}(z) Fn(s), Rn(d)`

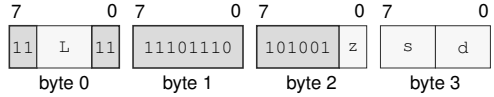
Instruction Format:



Format: Instruction format for FCVTUS.{B|W|L|Q}(z) Fn(s), Rn(d)

`FCVTUS.{B|W|L|Q}(z) Rn(s), Fn(d)`

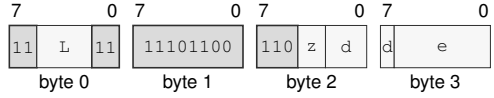
Instruction Format:



Format: Instruction format for FCVTUS.{B|W|L|Q}(z) Rn(s), Fn(d)

`FCVTUS.{B|W|L|Q}(z) <ea>(e), Fn(d)`

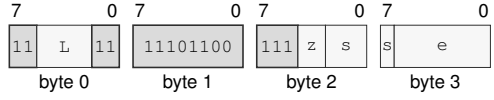
Instruction Format:



Format: Instruction format for FCVTUS.{B|W|L|Q}(z) <ea>(e), Fn(d)

`FCVTUS.{B|W|L|Q}(z) Fn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FCVTUS.{B|W|L|Q}(z) Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

FDIV — Floating-Point Divide — FDIV

Operation: Divides the destination value by the source under the floating-point environment.

Assembler Syntax:
```
FDIV.{S|D}(z) Fn(s), Fn(d)
FDIV.{S|D}(z) <ea>(e), Fn(d)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

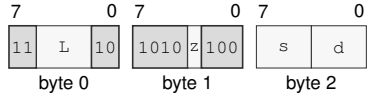
Detailed Semantics

Divides the selected-format floating-point destination value by the source and writes the rounded result under the architectural floating-point environment.

Encodings:

`FDIV.{S|D}(z) Fn(s), Fn(d)`

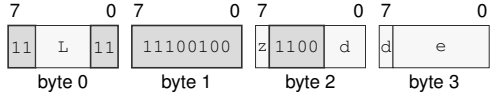
Instruction Format:



Format: Instruction format for FDIV.{S|D}(z) Fn(s), Fn(d)

`FDIV.{S|D}(z) <ea>(e), Fn(d)`

Instruction Format:



Format: Instruction format for FDIV.{S|D}(z) <ea>(e), Fn(d)

FFLOOR — Floating-Point Floor — FFLOOR

Operation: Rounds a floating-point source toward negative infinity in the selected format.

Assembler Syntax:
```
FFLOOR.{S|D}(z) Fn(s), Fn(d)
FFLOOR.{S|D}(z) <ea>(e), Fn(d)
FFLOOR.{S|D}(z) Fn(s), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: FP

Repeat eligibility: REP eligible

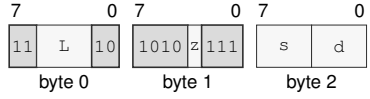
Detailed Semantics

Rounds the selected floating-point source toward negative infinity to an integral floating-point value.

Encodings:

`FFLOOR.{S|D}(z) Fn(s), Fn(d)`

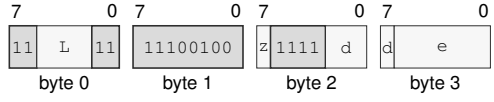
Instruction Format:



Format: Instruction format for FFLOOR.{S|D}(z) Fn(s), Fn(d)

`FFLOOR.{S|D}(z) <ea>(e), Fn(d)`

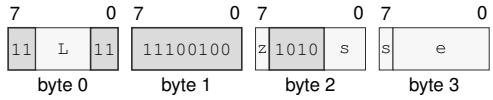
Instruction Format:



Format: Instruction format for FFLOOR.{S|D}(z) <ea>(e), Fn(d)

`FFLOOR.{S|D}(z) Fn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FFLOOR.{S|D}(z) Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

FGETEXP Floating-Point Get Exponent FGETEXP

Operation: Converts the source encoding's signed exponent to the selected floating-point format.

Assembler Syntax:
```
FGETEXP.{S|D}(z) <ea>(e), Fn(d)
FGETEXP.{S|D}(z) Fn(s), Fn(d)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

Detailed Semantics

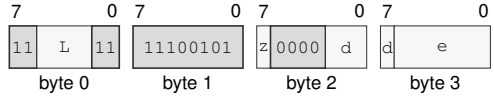
Derives a signed exponent from the selected source's exponent field and converts that integer to the selected floating-point destination format.

For S format, exponent field zero is treated as biased exponent one and the bias 127 is subtracted. For D format, exponent field zero is likewise treated as one and bias 1023 is subtracted. The resulting signed integer exponent is converted to the selected floating-point destination format. This rule gives zero and subnormal inputs the minimum normal exponent and leaves infinity and NaN encodings at the exponent produced by their all-ones field.

Encodings:

`FGETEXP.{S|D}(z) <ea>(e), Fn(d)`

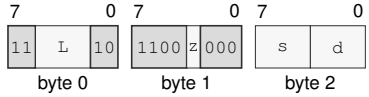
Instruction Format:



Format: Instruction format for FGETEXP.{S|D}(z) <ea>(e), Fn(d)

`FGETEXP.{S|D}(z) Fn(s), Fn(d)`

Instruction Format:



Format: Instruction format for FGETEXP.{S|D}(z) Fn(s), Fn(d)

FGETMAN Floating-Point Get Mantissa FGETMAN

Operation: Produces the normalized significand view of a floating-point source.

Assembler Syntax:
```
FGETMAN.{S|D}(z) <ea>(e), Fn(d)
FGETMAN.{S|D}(z) Fn(s), Fn(d)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

Detailed Semantics

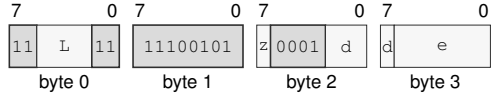
Extracts the normalized significand of the selected floating-point source and writes the specified floating-point result for finite and special values.

For a normal finite source, the result preserves the sign and fraction and replaces the exponent with the bias, producing a magnitude in $[1, 2)$. A zero, subnormal, infinity, or NaN source retains its source encoding, subject to the architectural signaling-NaN and default-NaN rules. The S and D encodings use their native exponent and fraction widths.

Encodings:

`FGETMAN.{S|D}(z) <ea>(e), Fn(d)`

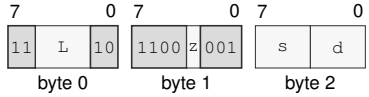
Instruction Format:



Format: Instruction format for FGETMAN.{S|D}(z) <ea>(e), Fn(d)

`FGETMAN.{S|D}(z) Fn(s), Fn(d)`

Instruction Format:



Format: Instruction format for FGETMAN.{S|D}(z) Fn(s), Fn(d)

FINT — Floating-Point Integral Value — FINT

Operation: Rounds a floating-point source to an integral value using FSTATUS.RM.

Assembler Syntax:
```
FINT.{S|D}(z) <ea>(e), Fn(d)
FINT.{S|D}(z) Fn(s), Fn(d)
FINT.{S|D}(z) Fn(s), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: FP

Repeat eligibility: REP eligible

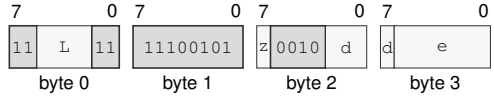
Detailed Semantics

Rounds the selected floating-point source to an integral floating-point value using the active FSTATUS rounding mode.

Encodings:

`FINT.{S|D}(z) <ea>(e), Fn(d)`

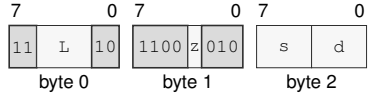
Instruction Format:



Format: Instruction format for FINT.{S|D}(z) <ea>(e), Fn(d)

`FINT.{S|D}(z) Fn(s), Fn(d)`

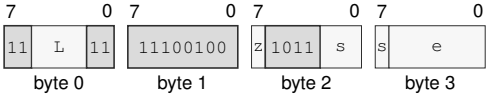
Instruction Format:



Format: Instruction format for FINT.{S|D}(z) Fn(s), Fn(d)

`FINT.{S|D}(z) Fn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FINT.{S|D}(z) Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

FINTRZ — Floating-Point Integral Toward Zero — FINTRZ

Operation: Rounds a floating-point source toward zero to an integral value.

Assembler Syntax:
```
FINTRZ.{S|D}(z) <ea>(e), Fn(d)
FINTRZ.{S|D}(z) Fn(s), Fn(d)
FINTRZ.{S|D}(z) Fn(s), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

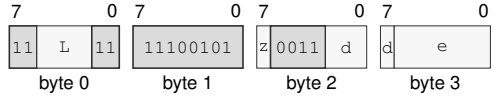
Detailed Semantics

Rounds the selected floating-point source toward zero to an integral floating-point value.

Encodings:

`FINTRZ.{S|D}(z) <ea>(e), Fn(d)`

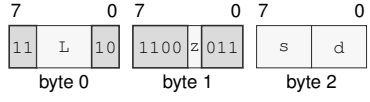
Instruction Format:



Format: Instruction format for FINTRZ.{S|D}(z) <ea>(e), Fn(d)

`FINTRZ.{S|D}(z) Fn(s), Fn(d)`

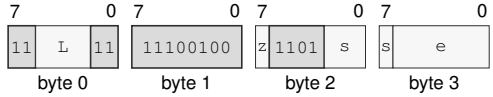
Instruction Format:



Format: Instruction format for FINTRZ.{S|D}(z) Fn(s), Fn(d)

`FINTRZ.{S|D}(z) Fn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FINTRZ.{S|D}(z) Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

FMADD — Floating-Point Fused Multiply Add — FMADD

Operation: Computes the source product plus the old destination as one fused operation and rounds once.

Assembler Syntax:
```
FMADD.{S|D}(z) Fn(l), Fn(r), Fn(d)
FMADD.{S|D}(z) <ea>(l), Fn(r), Fn(d)
FMADD.{S|D}(z) Fn(l), <ea>(r), Fn(d)
```

Privilege: Unprivileged

Required CPUID flags: FP

Repeat eligibility: REP eligible

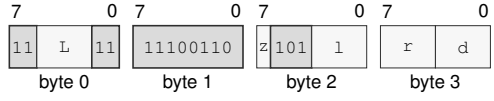
Detailed Semantics

Computes the selected-format source product plus the old destination as one fused operation, rounds once, and writes the result to the destination.

Encodings:

`FMADD.{S|D}(z) Fn(l), Fn(r), Fn(d)`

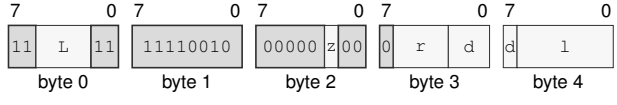
Instruction Format:



Format: Instruction format for FMADD.{S|D}(z) Fn(l), Fn(r), Fn(d)

`FMADD.{S|D}(z) <ea>(l), Fn(r), Fn(d)`

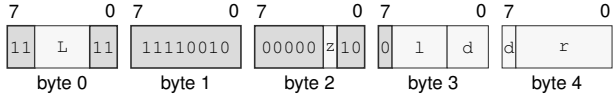
Instruction Format:



Format: Instruction format for FMADD.{S|D}(z) <ea>(l), Fn(r), Fn(d)

`FMADD.{S|D}(z) Fn(l), <ea>(r), Fn(d)`

Instruction Format:



Format: Instruction format for FMADD.{S|D}(z) Fn(l), <ea>(r), Fn(d)

FMAX — Floating-Point Maximum — FMAX

Operation: Selects the greater numeric source or destination value using IEEE NaN and zero rules.

Assembler Syntax:
```
FMAX.{S|D}(z) Fn(s), Fn(d)
FMAX.{S|D}(z) <ea>(e), Fn(d)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

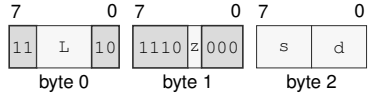
Detailed Semantics

Selects the greater numeric operand; one NaN selects the numeric operand, two NaNs use the common NaN-selection rule, and a positive-zero/negative-zero pair selects positive zero. A signaling NaN generates NV.

Encodings:

`FMAX.{S|D}(z) Fn(s), Fn(d)`

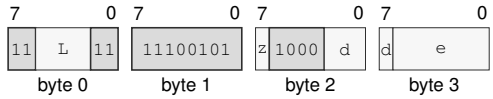
Instruction Format:



Format: Instruction format for FMAX.{S|D}(z) Fn(s), Fn(d)

`FMAX.{S|D}(z) <ea>(e), Fn(d)`

Instruction Format:



Format: Instruction format for FMAX.{S|D}(z) <ea>(e), Fn(d)

FMIN — Floating-Point Minimum — FMIN

Operation: Selects the lesser numeric source or destination value using IEEE NaN and zero rules.

Assembler Syntax:
```
FMIN.{S|D}(z) Fn(s), Fn(d)
FMIN.{S|D}(z) <ea>(e), Fn(d)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

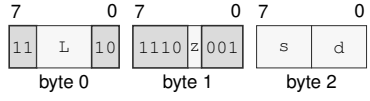
Detailed Semantics

Selects the lesser numeric operand; one NaN selects the numeric operand, two NaNs use the common NaN-selection rule, and a positive-zero/negative-zero pair selects negative zero. A signaling NaN generates NV.

Encodings:

`FMIN.{S|D}(z) Fn(s), Fn(d)`

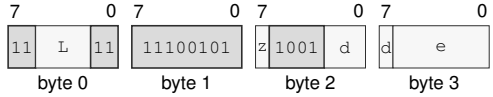
Instruction Format:



Format: Instruction format for FMIN.{S|D}(z) Fn(s), Fn(d)

`FMIN.{S|D}(z) <ea>(e), Fn(d)`

Instruction Format:



Format: Instruction format for FMIN.{S|D}(z) <ea>(e), Fn(d)

FMOD — Floating-Point Modulo — FMOD

Operation: Computes the destination modulo the source using a quotient truncated toward zero.

Assembler Syntax:
```
FMOD.{S|D}(z) <ea>(e), Fn(d)
FMOD.{S|D}(z) Fn(s), Fn(d)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

Detailed Semantics

Computes the IEEE-style floating-point modulus defined by a quotient rounded toward zero and writes the selected-format remainder.

After DAZ processing, let x be the original destination and y the source. For finite valid operands, choose q as x/y truncated toward zero and compute

$$r = x - qy$$

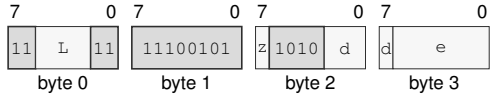
with exact intermediate arithmetic. A zero result has the sign of x.

A signaling NaN generates NV; a NaN input produces the selected quiet NaN. Infinite x or zero y generates NV and produces the architectural default quiet NaN. Infinite y with finite x returns x. An enabled floating-point exception condition raises the `FLOATING_POINT_EXCEPTION` architectural event before the destination write.

Encodings:

`FMOD.{S|D}(z) <ea>(e), Fn(d)`

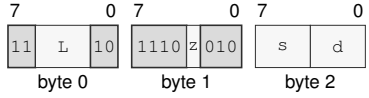
Instruction Format:



Format: Instruction format for FMOD.{S|D}(z) <ea>(e), Fn(d)

`FMOD.{S|D}(z) Fn(s), Fn(d)`

Instruction Format:



Format: Instruction format for FMOD.{S|D}(z) Fn(s), Fn(d)

FMOV — Floating-Point Move — FMOV

Operation: Copies the source operand to the destination operand.

Assembler Syntax:
```
FMOV.{S|D}(z) Fn(s), Fn(d)
FMOV.{S|D}(z) <ea>(e), Fn(d)
FMOV.{S|D}(z) Fn(s), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: FP

Repeat eligibility: REP eligible

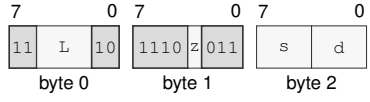
Detailed Semantics

Copies the selected-format floating-point source value to the destination.

Encodings:

`FMOV.{S|D}(z) Fn(s), Fn(d)`

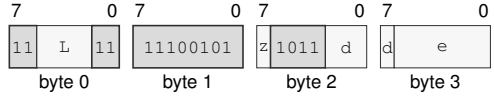
Instruction Format:



Format: Instruction format for FMOV.{S|D}(z) Fn(s), Fn(d)

`FMOV.{S|D}(z) <ea>(e), Fn(d)`

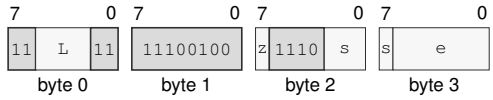
Instruction Format:



Format: Instruction format for FMOV.{S|D}(z) <ea>(e), Fn(d)

`FMOV.{S|D}(z) Fn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FMOV.{S|D}(z) Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

FMOVcc — Floating-Point Conditional Move — FMOVcc

Operation: Copies a floating-point source only when the encoded integer condition is true.

Assembler Syntax:
```
FMOVcc Fn(s), Fn(d)
FMOVcc.{S|D}(z) <ea>(e), Fn(d)
FMOVcc.{S|D}(z) Fn(s), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: FP

Repeat eligibility: REP eligible

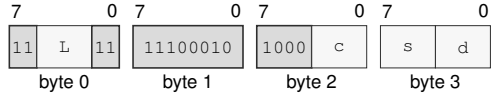
Detailed Semantics

Copies the selected-format floating-point source value to the destination only when the encoded integer condition is true.

Encodings:

FMOVcc Fn(s), Fn(d)

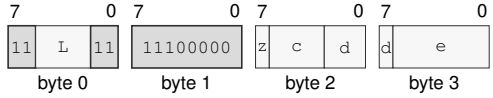
Instruction Format:



Format: Instruction format for FMOVcc Fn(s), Fn(d)

`FMOVcc.{S|D}(z) <ea>(e), Fn(d)`

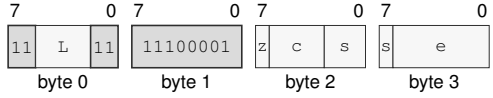
Instruction Format:



Format: Instruction format for FMOVcc.{S|D}(z) <ea>(e), Fn(d)

`FMOVcc.{S|D}(z) Fn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FMOVcc.{S|D}(z) Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

FMOVCR Floating-Point Move Constant FMOVCR

Operation: Loads a named architectural binary64 constant selected by an unsigned 16-bit ID.

Assembler Syntax: `FMOVCR.{S|D}(z) <fconst_id>, Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

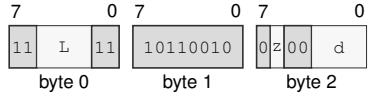
Detailed Semantics

FMOVCR maps an unsigned 16-bit constant ID to a fixed IEEE-754 binary64 bit pattern and writes that pattern to the destination F register. It is independent of FSTATUS rounding, DAZ, FTZ, and default-NaN modes. Loading a signaling NaN is a bitwise move and does not itself raise NV; a later consuming floating-point operation applies its normal NaN rules.

Encodings:

`FMOVCR.{S|D}(z) <fconst_id>, Fn(d)`

Instruction Format:



Format: Instruction format for FMOVCR.{S|D}(z) <fconst_id>, Fn(d)

FMSUB — Floating-Point Fused Multiply Subtract — FMSUB

Operation: Computes the source product minus the old destination as one fused operation and rounds once.

Assembler Syntax:
```
FMSUB.{S|D}(z) Fn(l), Fn(r), Fn(d)
FMSUB.{S|D}(z) <ea>(l), Fn(r), Fn(d)
FMSUB.{S|D}(z) Fn(l), <ea>(r), Fn(d)
```

Privilege: Unprivileged

Required CPUID flags: FP

Repeat eligibility: REP eligible

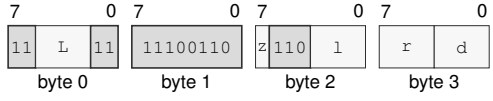
Detailed Semantics

Computes the selected-format source product minus the old destination as one fused operation, rounds once, and writes the result to the destination.

Encodings:

`FMSUB.{S|D}(z) Fn(l), Fn(r), Fn(d)`

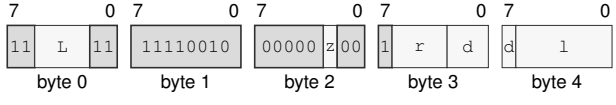
Instruction Format:



Format: Instruction format for FMSUB.{S|D}(z) Fn(l), Fn(r), Fn(d)

`FMSUB.{S|D}(z) <ea>(l), Fn(r), Fn(d)`

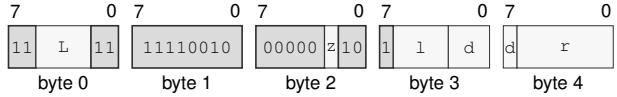
Instruction Format:



Format: Instruction format for FMSUB.{S|D}(z) <ea>(l), Fn(r), Fn(d)

`FMSUB.{S|D}(z) Fn(l), <ea>(r), Fn(d)`

Instruction Format:



Format: Instruction format for FMSUB.{S|D}(z) Fn(l), <ea>(r), Fn(d)

FMUL — Floating-Point Multiply — FMUL

Operation: Multiplies the source and destination values under the floating-point environment.

Assembler Syntax:
```
FMUL.{S|D}(z) Fn(s), Fn(d)
FMUL.{S|D}(z) <ea>(e), Fn(d)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

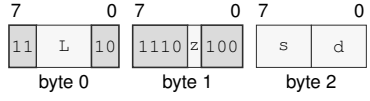
Detailed Semantics

Multiplies the selected-format floating-point source and destination values and writes the rounded result.

Encodings:

`FMUL.{S|D}(z) Fn(s), Fn(d)`

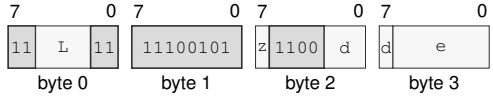
Instruction Format:



Format: Instruction format for FMUL.{S|D}(z) Fn(s), Fn(d)

`FMUL.{S|D}(z) <ea>(e), Fn(d)`

Instruction Format:



Format: Instruction format for FMUL.{S|D}(z) <ea>(e), Fn(d)

FNEG — Floating-Point Negate — FNEG

Operation: Complements the selected floating-point source sign bit.

Assembler Syntax:
```
FNEG.{S|D}(z) Fn(s), Fn(d)
FNEG.{S|D}(z) <ea>(e), Fn(d)
FNEG.{S|D}(z) Fn(s), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

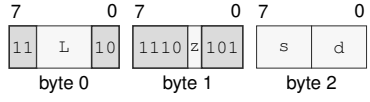
Detailed Semantics

Complements the sign bit of the selected floating-point source value and writes the result.

Encodings:

`FNEG.{S|D}(z) Fn(s), Fn(d)`

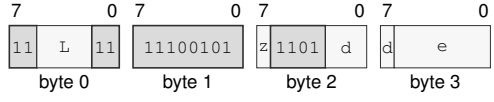
Instruction Format:



Format: Instruction format for FNEG.{S|D}(z) Fn(s), Fn(d)

`FNEG.{S|D}(z) <ea>(e), Fn(d)`

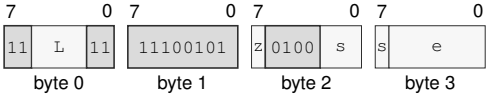
Instruction Format:



Format: Instruction format for FNEG.{S|D}(z) <ea>(e), Fn(d)

`FNEG.{S|D}(z) Fn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FNEG.{S|D}(z) Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

FNMADD Floating-Point Fused Negative Multiply Add FNMADD

Operation: Computes the negated source product minus the old destination as one fused operation and rounds once.

Assembler Syntax:
```
FNMADD.{S|D}(z) Fn(l), Fn(r), Fn(d)
FNMADD.{S|D}(z) <ea>(l), Fn(r), Fn(d)
FNMADD.{S|D}(z) Fn(l), <ea>(r), Fn(d)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

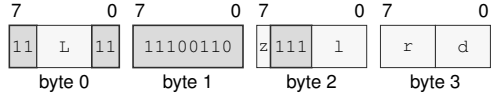
Detailed Semantics

Computes the negated selected-format source product minus the old destination as one fused operation, rounds once, and writes the result to the destination.

Encodings:

`FNMADD.{S|D}(z) Fn(l), Fn(r), Fn(d)`

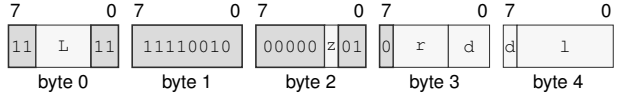
Instruction Format:



Format: Instruction format for FNMADD.{S|D}(z) Fn(l), Fn(r), Fn(d)

`FNMADD.{S|D}(z) <ea>(l), Fn(r), Fn(d)`

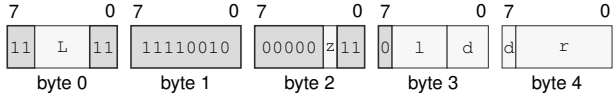
Instruction Format:



Format: Instruction format for FNMADD.{S|D}(z) <ea>(l), Fn(r), Fn(d)

`FNMADD.{S|D}(z) Fn(l), <ea>(r), Fn(d)`

Instruction Format:



Format: Instruction format for FNMADD.{S|D}(z) Fn(l), <ea>(r), Fn(d)

FNMSUB Floating-Point Fused Negative Multiply Subtract FNMSUB

Operation: Computes the old destination minus the source product as one fused operation and rounds once.

Assembler Syntax:
```
FNMSUB.{S|D}(z) Fn(l), Fn(r), Fn(d)
FNMSUB.{S|D}(z) <ea>(l), Fn(r), Fn(d)
FNMSUB.{S|D}(z) Fn(l), <ea>(r), Fn(d)
```

Privilege: Unprivileged

Required CPUID flags: FP

Repeat eligibility: REP eligible

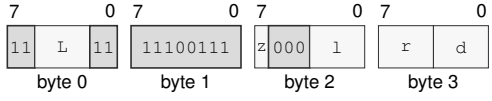
Detailed Semantics

Computes the selected-format old destination minus the source product as one fused operation, rounds once, and writes the result to the destination.

Encodings:

`FNMSUB.{S|D}(z) Fn(l), Fn(r), Fn(d)`

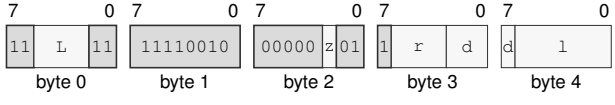
Instruction Format:



Format: Instruction format for FNMSUB.{S|D}(z) Fn(l), Fn(r), Fn(d)

`FNMSUB.{S|D}(z) <ea>(l), Fn(r), Fn(d)`

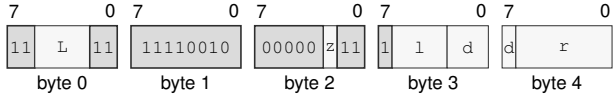
Instruction Format:



Format: Instruction format for FNMSUB.{S|D}(z) <ea>(l), Fn(r), Fn(d)

`FNMSUB.{S|D}(z) Fn(l), <ea>(r), Fn(d)`

Instruction Format:



Format: Instruction format for FNMSUB.{S|D}(z) Fn(l), <ea>(r), Fn(d)

FPOPP Floating-Point Pop Register Pair FPOPP

Operation: Pops one canonical floating-point register pair selected by an inline pair index.

Assembler Syntax: `FPOPP FPAIRn(i)`

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

Detailed Semantics

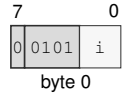
Pops the canonical floating-point register pair selected by the pair ID. The canonical pair sequence is pair 0 (F0, F1), pair 1 (F2, F3), pair 2 (F4, F5), pair 3 (F6, F7), pair 4 (F8, F9), pair 5 (F10, F11), pair 6 (F12, F13), and pair 7 (F14, F15). All eight pair IDs are valid and each selects one pair.

FPOPP selects one canonical floating-point pair using the unsigned 3-bit pair index and processes that pair's registers in reverse listed order. The complete two-slot stack range is validated and read before either floating-point register or SP is changed. Each register pop loads the 64-bit stack slot at SP and then increments SP by 8.

Encodings:

`FPOPP FPAIRn(i)`

Instruction Format:



Format: Instruction format for FPOPP FPAIRn(i)

FPUSHP Floating-Point Push Register Pair FPUSHP

Operation: Pushes one canonical floating-point register pair selected by an inline pair index.

Assembler Syntax: `FPUSHP FPAIRn(i)`

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

Detailed Semantics

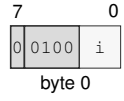
Pushes the canonical floating-point register pair selected by the pair ID. The canonical pair sequence is pair 0 (F0, F1), pair 1 (F2, F3), pair 2 (F4, F5), pair 3 (F6, F7), pair 4 (F8, F9), pair 5 (F10, F11), pair 6 (F12, F13), and pair 7 (F14, F15). All eight pair IDs are valid and each selects one pair.

FPUSHP selects one canonical floating-point pair using the unsigned 3-bit pair index and processes that pair's registers in the listed order. The complete two-slot stack range is validated before either slot or SP is changed. Each register push decrements SP by 8 and then stores the 64-bit floating-point register value at the new SP.

Encodings:

`FPUSHP FPAIRn(i)`

Instruction Format:



Format: Instruction format for FPUSHP FPAIRn(i)

FREM — Floating-Point Remainder — FREM

Operation: Computes the destination remainder using a nearest-even integral quotient.

Assembler Syntax:
```
FREM.{S|D}(z) <ea>(e), Fn(d)
FREM.{S|D}(z) Fn(s), Fn(d)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

Detailed Semantics

Computes and writes the IEEE floating-point remainder to the destination using a quotient rounded to the nearest integer with ties to even.

After DAZ processing, let x be the original destination and y the source. For finite valid operands, choose q as x/y rounded to the nearest integer with ties to even and compute

$$r = x - qy$$

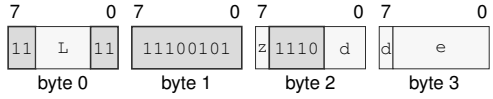
with exact intermediate arithmetic. A zero result has the sign of x.

A signaling NaN generates NV; a NaN input produces the selected quiet NaN. Infinite x or zero y generates NV and produces the architectural default quiet NaN. Infinite y with finite x returns x. An enabled floating-point exception condition raises the `FLOATING_POINT_EXCEPTION` architectural event before the destination write.

Encodings:

`FREM.{S|D}(z) <ea>(e), Fn(d)`

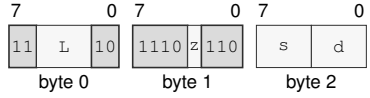
Instruction Format:



Format: Instruction format for FREM.{S|D}(z) <ea>(e), Fn(d)

`FREM.{S|D}(z) Fn(s), Fn(d)`

Instruction Format:



Format: Instruction format for FREM.{S|D}(z) Fn(s), Fn(d)

FRESTORE Restore Floating-Point State FRESTORE

Operation: Restores the floating-point user-state record.

Assembler Syntax: `FRESTORE [Rn(r)]`

Privilege: Unprivileged

Required CPUID flags: `FP`

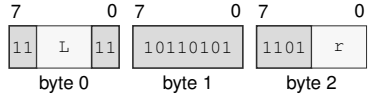
Detailed Semantics

FRESTORE validates the complete 192-byte source range and its bitmap before atomically replacing floating-point user state. A clear bit installs the corresponding initial value without reading that payload slot. An all-clear bitmap makes the floating-point state clean; any set bit makes it modified.

Encodings:

`FRESTORE [Rn(r)]`

Instruction Format:



Format: Instruction format for FRESTORE [Rn(r)]

FROUND — Floating-Point Round — FROUND

Operation: Rounds a floating-point source to a nearest-even integral value independently of FSTATUS.RM.

Assembler Syntax:
```
FROUND.{S|D}(z) Fn(s), Fn(d)
FROUND.{S|D}(z) <ea>(e), Fn(d)
FROUND.{S|D}(z) Fn(s), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: FP

Repeat eligibility: REP eligible

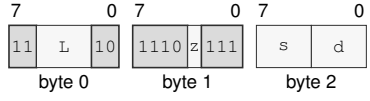
Detailed Semantics

Rounds the selected floating-point source to the nearest integral value in the same format using nearest-even, independently of FSTATUS.RM.

Encodings:

FROUND.{S|D}(z) Fn(s), Fn(d)

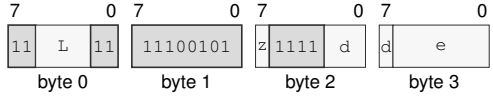
Instruction Format:



Format: Instruction format for FROUND.{S|D}(z) Fn(s), Fn(d)

FROUND.{S|D}(z) <ea>(e), Fn(d)

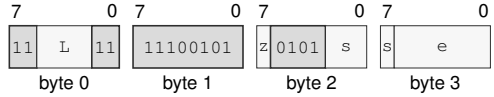
Instruction Format:



Format: Instruction format for FROUND.{S|D}(z) <ea>(e), Fn(d)

`FROUND.{S|D}(z) Fn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FROUND.{S|D}(z) Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

FSAVE — Save Floating-Point State — FSAVE

Operation: Saves the floating-point user-state record.

Assembler Syntax: `FSAVE [Rn(r)]`

Privilege: Unprivileged

Required CPUID flags: `FP`

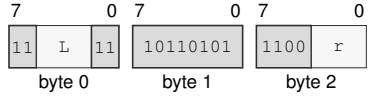
Detailed Semantics

FSAVE writes one 192-byte, 64-byte-aligned floating-point user-state record. A clean initial floating-point state may use an all-clear presence bitmap; otherwise FSAVE may conservatively mark and write every floating-point slot. FSAVE does not change clean or modified state.

Encodings:

`FSAVE [Rn(r)]`

Instruction Format:



Format: Instruction format for FSAVE [Rn(r)]

FSCALE — Floating-Point Scale — FSCALE

Operation: Scales the destination by a power of two derived from the rounded source.

Assembler Syntax:
```
FSCALE.{S|D}(z) <ea>(e), Fn(d)
FSCALE.{S|D}(z) Fn(s), Fn(d)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

Detailed Semantics

Scales the selected floating-point destination by an integral power of two derived from the source and writes the rounded result.

Let x be the destination value after DAZ processing and let k be the source value rounded to an integer using the current rounding mode. The mathematical result before format rounding is

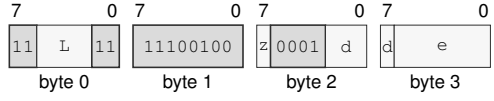
$$x \times 2^k.$$

A NaN operand is propagated according to the common floating-point rules. Multiplying zero or infinity by the power of two preserves that value and its sign. Conversion of the scale operand to k can generate NX; final rounding can generate OF, UF, or NX. An enabled floating-point exception condition raises the `FLOATING_POINT_EXCEPTION` architectural event before the destination is written.

Encodings:

`FSCALE.{S|D}(z) <ea>(e), Fn(d)`

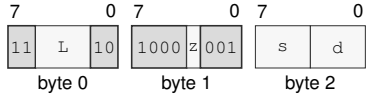
Instruction Format:



Format: Instruction format for FSCALE.{S|D}(z) <ea>(e), Fn(d)

`FSCALE.{S|D}(z) Fn(s), Fn(d)`

Instruction Format:



Format: Instruction format for FSCALE.{S|D}(z) Fn(s), Fn(d)

FSQRT — Floating-Point Square Root — FSQRT

Operation: Computes the selected-format square root of a floating-point source.

Assembler Syntax:
```
FSQRT.{S|D}(z) Fn(s), Fn(d)
FSQRT.{S|D}(z) <ea>(e), Fn(d)
FSQRT.{S|D}(z) Fn(s), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: FP

Repeat eligibility: REP eligible

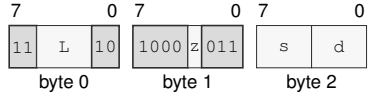
Detailed Semantics

Computes the selected-format square root of the source and writes the rounded result.

Encodings:

`FSQRT.{S|D}(z) Fn(s), Fn(d)`

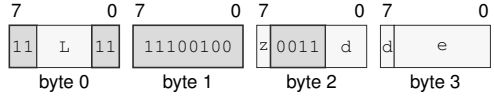
Instruction Format:



Format: Instruction format for FSQRT.{S|D}(z) Fn(s), Fn(d)

`FSQRT.{S|D}(z) <ea>(e), Fn(d)`

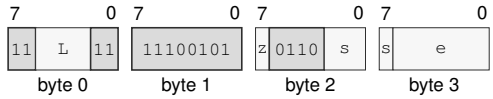
Instruction Format:



Format: Instruction format for FSQRT.{S|D}(z) <ea>(e), Fn(d)

`FSQRT.{S|D}(z) Fn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FSQRT.{S|D}(z) Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

FSUB — Floating-Point Subtract — FSUB

Operation: Subtracts the source operand from the destination operand.

Assembler Syntax:
```
FSUB.{S|D}(z) Fn(s), Fn(d)
FSUB.{S|D}(z) <ea>(e), Fn(d)
```

Privilege: Unprivileged

Required CPUID flags: FP

Repeat eligibility: REP eligible

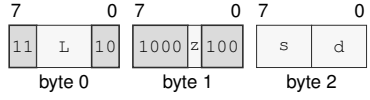
Detailed Semantics

Subtracts the selected-format floating-point source from the destination and writes the rounded result.

Encodings:

`FSUB.{S|D}(z) Fn(s), Fn(d)`

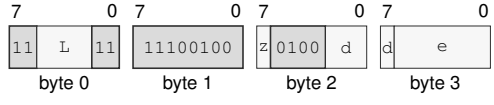
Instruction Format:



Format: Instruction format for FSUB.{S|D}(z) Fn(s), Fn(d)

`FSUB.{S|D}(z) <ea>(e), Fn(d)`

Instruction Format:



Format: Instruction format for FSUB.{S|D}(z) <ea>(e), Fn(d)

FTEST — Floating-Point Test — FTEST

Operation: Compares a floating-point source with positive zero and writes integer condition flags.

Assembler Syntax:
```
FTEST.{S|D}(z) <ea>(e)
FTEST.{S|D}(z) Fn(s)
```

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

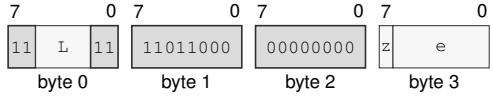
Detailed Semantics

Compares the selected-format operand with positive zero and writes `FLAGS.Z`, `FLAGS.N`, `FLAGS.C`, and `FLAGS.V` as greater=0000, equal=1000, less=0110, or unordered=0001. A quiet NaN is unordered; a signaling NaN is unordered and generates NV.

Encodings:

`FTEST.{S|D}(z) <ea>(e)`

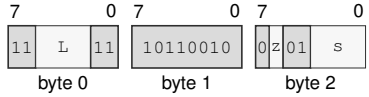
Instruction Format:



Format: Instruction format for FTEST.{S|D}(z) <ea>(e)

`FTEST.{S|D}(z) Fn(s)`

Instruction Format:



Format: Instruction format for FTEST.{S|D}(z) Fn(s)

FTRUNC — Floating-Point Truncate — FTRUNC

Operation: Rounds a floating-point source toward zero in the same format.

Assembler Syntax:
```
FTRUNC.{S|D}(z) Fn(s), Fn(d)
FTRUNC.{S|D}(z) <ea>(e), Fn(d)
FTRUNC.{S|D}(z) Fn(s), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: FP

Repeat eligibility: REP eligible

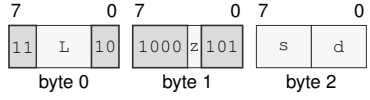
Detailed Semantics

Rounds the selected floating-point source toward zero to an integral value in the same floating-point format.

Encodings:

`FTRUNC.{S|D}(z) Fn(s), Fn(d)`

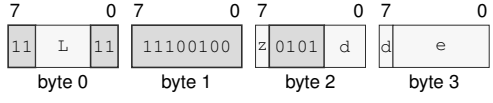
Instruction Format:



Format: Instruction format for FTRUNC.{S|D}(z) Fn(s), Fn(d)

`FTRUNC.{S|D}(z) <ea>(e), Fn(d)`

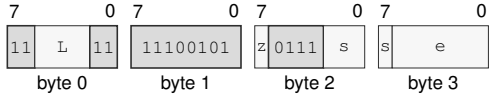
Instruction Format:



Format: Instruction format for FTRUNC.{S|D}(z) <ea>(e), Fn(d)

`FTRUNC.{S|D}(z) Fn(s), <ea>(e)`

Instruction Format:



Format: Instruction format for FTRUNC.{S|D}(z) Fn(s), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

FXCHG — Floating-Point Exchange — FXCHG

Operation: Exchanges the complete 64-bit images of two floating-point registers.

Assembler Syntax: `FXCHG Fn(l), Fn(r)`

Privilege: Unprivileged

Required CPUID flags: `FP`

Repeat eligibility: REP eligible

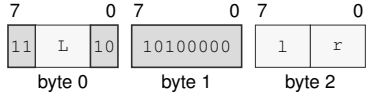
Detailed Semantics

Exchanges the complete 64-bit images of two floating-point registers.

Encodings:

`FXCHG Fn(l), Fn(r)`

Instruction Format:



Format: Instruction format for FXCHG Fn(l), Fn(r)

Restrictions:

Operand overlap: Operands `lhs`, `rhs`; rule `same_value`.

RDFFLAGS Read Floating-Point Flags RDFFLAGS

Operation: Read the accrued floating-point exception flags into an Rn register.

Assembler Syntax: `RDFFLAGS Rn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`

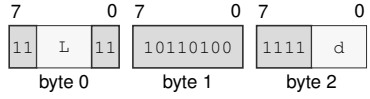
Detailed Semantics

RDFFLAGS reads the architectural FFLAGS register, zero-extends it to the destination register, and writes only that destination. RDFFLAGS requires the floating-point extension.

Encodings:

`RDFFLAGS Rn(d)`

Instruction Format:



Format: Instruction format for RDFFLAGS Rn(d)

RDFSTATUS — Read Floating-Point Status — RDFSTATUS

Operation: Read the FPU status/control register into an Rn register.

Assembler Syntax: `RDFSTATUS Rn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`

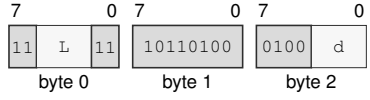
Detailed Semantics

RDFSTATUS reads the architectural 16-bit FSTATUS register, zero-extends it to 64 bits, and writes the result to the destination Rn register. FSTATUS contains floating-point exception-condition enables, rounding mode, and NaN and subnormal handling bits. FFLAGS holds accrued floating-point exception flags. RDFSTATUS requires the floating-point extension.

Encodings:

`RDFSTATUS Rn(d)`

Instruction Format:



Format: Instruction format for RDFSTATUS Rn(d)

WRFFLAGS Write Floating-Point Flags WRFFLAGS

Operation: Write the accrued floating-point exception flags from an Rn register.

Assembler Syntax: `WRFFLAGS Rn(s)`

Privilege: Unprivileged

Required CPUID flags: `FP`

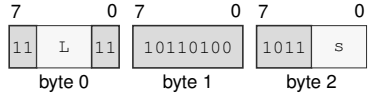
Detailed Semantics

WRFFLAGS writes the architectural FFLAGS register from the low source bits. Reserved FFLAGS bits must be zero. WRFFLAGS requires the floating-point extension.

Encodings:

`WRFFLAGS Rn(s)`

Instruction Format:



Format: Instruction format for WRFFLAGS Rn(s)

WRFSTATUS Write Floating-Point Status WRFSTATUS

Operation: Write the FPU status/control register from an Rn register.

Assembler Syntax: `WRFSTATUS Rn(s)`

Privilege: Unprivileged

Required CPUID flags: `FP`

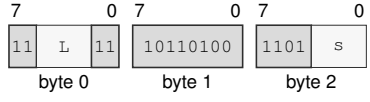
Detailed Semantics

WRFSTATUS writes only the architectural 16-bit FSTATUS register from the low 16 bits of the source Rn register. Reserved FSTATUS bits must be zero. WRFSTATUS requires the floating-point extension.

Encodings:

`WRFSTATUS Rn(s)`

Instruction Format:



Format: Instruction format for WRFSTATUS Rn(s)

22 APPROXIMATE FLOATING-POINT TRANSCENDENTAL INSTRUCTIONS

22.1 Summary

Table 22-1. Approximate Floating-Point Transcendental Instructions Summary (Informative)

Mnemonic	Brief description
FACOSA	Approximates arccosine in radians for finite source magnitudes at most one.
FASINA	Approximates arcsine in radians for finite source magnitudes at most one.
FATANA	Approximates arctangent in radians for finite values and signed infinity.
FATANHA	Approximates hyperbolic arctangent for finite source magnitudes at most one.
FCOSA	Approximates cosine in radians over the reduced interval from minus pi/4 to pi/4.
FCOSHA	Approximates hyperbolic cosine for finite values and signed infinity.
FETOXA	Approximates e raised to the source for finite values and signed infinity.
FETOXM1A	Approximates e raised to the source minus one without an intermediate rounded exponential.
FLOG10A	Approximates base-10 logarithm for positive values, with zero as the divide-by-zero boundary.
FLOG2A	Approximates base-2 logarithm for positive values, with zero as the divide-by-zero boundary.
FLOGNA	Approximates natural logarithm for positive values, with zero as the divide-by-zero boundary.
FLOGNP1A	Approximates natural log of one plus the source, with minus one as the divide-by-zero boundary.
FSINA	Approximates sine in radians over the reduced interval from minus pi/4 to pi/4.
FSINCOSA	Atomically writes paired sine and cosine approximations over the reduced pi/4 interval.
FSINHA	Approximates hyperbolic sine for finite values and signed infinity.
FTANA	Approximates tangent in radians over the reduced interval from minus pi/4 to pi/4.
FTANHA	Approximates hyperbolic tangent for finite values and signed infinity.
FTENTOXA	Approximates ten raised to the source for finite values and signed infinity.
FTWOTOXA	Approximates two raised to the source for finite values and signed infinity.

22.2 FPTRANSA Common Model

For an approximate transcendental instruction, x' is the source after FSTATUS.DAZ, $y = f(x')$ is the exact real reference value named by its accuracy contract, and a is the approximate result before FSTATUS.FTZ. For a format with precision p and minimum normal exponent $e_{\min}$, define

$$\operatorname{ulp}_F(y) = \begin{cases} 2^{\max(\lfloor \log_2 |y| \rfloor - p + 1,\ e_{\min} - p + 1)}, & y \neq 0, \\ 2^{e_{\min} - p + 1}, & y = 0. \end{cases}$$

The measured error is $|a - y| / \operatorname{ulp}_F(y)$. The bound applies only to finite reference values within the selected format's finite range; NaN, infinity, and overflow follow the instruction's special-value rules. Every present contract guarantees at most 4 ULP for both S and D, corresponding to at least 21 and 50 worst-case relative precision bits respectively for nonzero normal results. CPUID may advertise a smaller certified bound. Each instruction names its required FPTRANSA_ACCURACY

contract. A clear PRESENT for that contract raises INVALID_OPERAND_RELATION before the instruction reads operands or floating-point state. An approximate transcendental operation is available only when CPUID reports FPTRANSA and marks that instruction's accuracy-contract ID present.

FPTRANSA applies DAZ before forming the exact reference input and measures its advertised ULP error before FTZ. It ignores FSTATUS.RM and formats its implementation-defined approximation with nearest-even rounding. The same implementation, input, format, and relevant FSTATUS mode produce a deterministic approximation. Different implementations may produce different bit patterns within the applicable accuracy contract.

A signaling NaN generates NV. NaN results are quieted and then processed by FSTATUS.DN. FSTATUS.FTZ replaces a subnormal result with the required signed zero and generates UF. Each instruction generates NV and DZ according to its special-value rules, and generates OF and UF when required by its exact result; NX is never generated, remains unchanged in FFLAGS, and cannot cause the FLOATING_POINT_EXCEPTION architectural event. Overflow means that the exact result magnitude exceeds the selected format's maximum finite value. Underflow means that the exact nonzero result magnitude is below the selected format's minimum normal value. If any generated floating-point exception cause has its matching FSTATUS enable bit set, the instruction raises the FLOATING_POINT_EXCEPTION architectural event before writing any destination. Otherwise it commits its result or results and accrues the generated causes in FFLAGS as one architectural commit.

FACOSA — Approximate Floating-Point Arc Cosine — FACOSA

Operation: Approximates arccosine in radians for finite source magnitudes at most one.

Assembler Syntax: `FACOSA.{S|D}(z) Fn(s), Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`FPTRANSA`

Repeat eligibility: REP eligible

Detailed Semantics

Computes the $\mathrm{acos}(x)$ reference in radians over finite x with $|x| \leq 1$ after DAZ and writes an implementation-bounded S- or D-format approximation under the selected FPTRANSA accuracy contract.

The required FPTRANSA_ACCURACY contract is `0x0012`.

Special values

- Source $+1$: the result is positive zero.
- A finite source with magnitude greater than one, or either infinity, generates NV and produces the architectural default quiet NaN.

Exact anchors

- $\mathrm{acos}(+1) = +0$

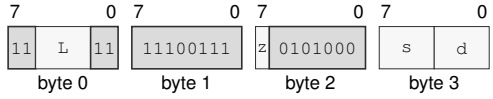
Required properties

- monotonically decreasing on $[-1, 1]$
- result is in $[0, \pi]$

Encodings:

```
FACOSA.{S|D}(z) Fn(s), Fn(d)
```

Instruction Format:



Format: Instruction format for FACOSA.{S|D}(z) Fn(s), Fn(d)

FASINA — Approximate Floating-Point Arc Sine — FASINA

Operation: Approximates arcsine in radians for finite source magnitudes at most one.

Assembler Syntax: `FASINA.{S|D}(z) Fn(s), Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`FPTRANSA`

Repeat eligibility: REP eligible

Detailed Semantics

Computes the $\mathrm{asin}(x)$ reference in radians over finite x with $|x| \leq 1$ after DAZ and writes an implementation-bounded S- or D-format approximation under the selected FPTRANSA accuracy contract.

The required FPTRANSA_ACCURACY contract is `0x0011`.

Special values

- Zero source: the result is the source.
- A finite source with magnitude greater than one, or either infinity, generates NV and produces the architectural default quiet NaN.

Exact anchors

- $\mathrm{asin}(+0) = +0$
- $\mathrm{asin}(-0) = -0$

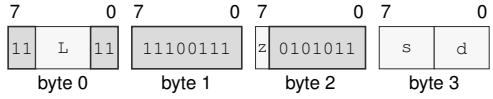
Required properties

- odd
- monotonically increasing on $[-1, 1]$
- result is in $[-\pi/2, \pi/2]$

Encodings:

```
FASINA.{S|D}(z) Fn(s), Fn(d)
```

Instruction Format:



Format: Instruction format for FASINA.{S|D}(z) Fn(s), Fn(d)

FATANA — Approximate Floating-Point Arc Tangent — FATANA

Operation: Approximates arctangent in radians for finite values and signed infinity.

Assembler Syntax: `FATANA.{S|D}(z) Fn(s), Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`FPTRANSA`

Repeat eligibility: REP eligible

Detailed Semantics

Computes the $\mathrm{atan}(x)$ reference in radians over all finite values and signed infinity after DAZ and writes an implementation-bounded S- or D-format approximation under the selected FPTRANSA accuracy contract.

The required FPTRANSA_ACCURACY contract is `0x0013`.

Special values

- Zero source: the result is the source.

Exact anchors

- $\mathrm{atan}(+0) = +0$
- $\mathrm{atan}(-0) = -0$

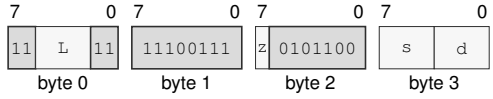
Required properties

- odd
- monotonically increasing
- finite results are in $[-\pi/2, \pi/2]$

Encodings:

```
FATANA.{S|D}(z) Fn(s), Fn(d)
```

Instruction Format:



Format: Instruction format for FATANA.{S|D}(z) Fn(s), Fn(d)

FATANHA Approximate Floating-Point Hyperbolic Arc Tangent FATANHA

Operation: Approximates hyperbolic arctangent for finite source magnitudes at most one.

Assembler Syntax: `FATANHA.{S|D}(z) Fn(s), Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`FPTRANSA`

Repeat eligibility: REP eligible

Detailed Semantics

Computes the $\operatorname{atanh}(x)$ reference over finite x with $|x| \leq 1$ after DAZ and writes an implementation-bounded S- or D-format approximation under the selected FPTRANSA accuracy contract.

The required FPTRANSA_ACCURACY contract is `0x0024`.

Special values

- Zero source: the result is the source.
- Source $+1$ or -1 generates DZ and returns infinity with the source sign.
- A finite source with magnitude greater than one, or either infinity, generates NV and produces the architectural default quiet NaN.

Exact anchors

- $\operatorname{atanh}(+0) = +0$
- $\operatorname{atanh}(-0) = -0$
- $\operatorname{atanh}(\pm 1) = \pm\infty$

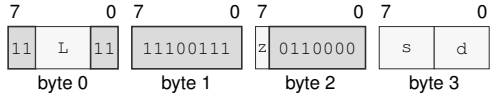
Required properties

- odd
- monotonically increasing on $(-1, 1)$

Encodings:

```
FATANHA.{S|D}(z) Fn(s), Fn(d)
```

Instruction Format:



Format: Instruction format for FATANHA.{S|D}(z) Fn(s), Fn(d)

FCOSA — Approximate Floating-Point Cosine — FCOSA

Operation: Approximates cosine in radians over the reduced interval from minus pi/4 to pi/4.

Assembler Syntax: `FCOSA.{S|D}(z) Fn(s), Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`FPTRANSA`

Repeat eligibility: REP eligible

Detailed Semantics

Computes the $\cos(x)$ reference in radians over finite x with $|x| \leq \pi/4$ after DAZ and writes an implementation-bounded S- or D-format approximation under the selected FPTRANSA accuracy contract.

The required FPTRANSA_ACCURACY contract is `0x0002`.

Special values

- Infinite source or source magnitude above $\pi/4$: NV is generated and the result is the architectural default quiet NaN.
- Zero source: the result is positive one.

Exact anchors

- $\cos(+0) = +1$
- $\cos(-0) = +1$

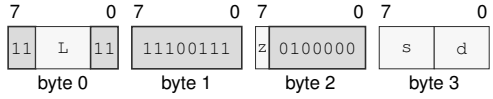
Required properties

- even
- nondecreasing on $[-\pi/4, 0]$ and nonincreasing on $[0, \pi/4]$
- result is in $[\sqrt{2}/2, 1]$

Encodings:

`FCOSA.{S|D}(z) Fn(s), Fn(d)`

Instruction Format:



Format: Instruction format for FCOSA.{S|D}(z) Fn(s), Fn(d)

FCOSHA Approximate Floating-Point Hyperbolic Cosine FCOSHA

Operation: Approximates hyperbolic cosine for finite values and signed infinity.

Assembler Syntax: `FCOSHA.{S|D}(z) Fn(s), Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`FPTRANSA`

Repeat eligibility: REP eligible

Detailed Semantics

Computes the cosh(x) reference over all finite values and signed infinity after DAZ and writes an implementation-bounded S- or D-format approximation under the selected FPTRANSA accuracy contract.

The required FPTRANSA_ACCURACY contract is `0x0022`.

Special values

- Infinite source: the result is +infinity.
- Zero source: the result is positive one.

Exact anchors

- cosh(+0) = +1
- cosh(-0) = +1
- cosh(infinity) = +infinity

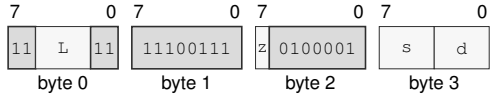
Required properties

- even
- nonincreasing below zero and nondecreasing above zero
- result is at least 1

Encodings:

```
FCOSHA.{S|D}(z) Fn(s), Fn(d)
```

Instruction Format:



Format: Instruction format for FCOSHA.{S|D}(z) Fn(s), Fn(d)

FETOXA — Approximate Floating-Point e To x — FETOXA

Operation: Approximates e raised to the source for finite values and signed infinity.

Assembler Syntax: `FETOXA.{S|D}(z) Fn(s), Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`FPTRANSA`

Repeat eligibility: REP eligible

Detailed Semantics

Computes the e^x reference over all finite values and signed infinity after DAZ and writes an implementation-bounded S- or D-format approximation under the selected FPTRANSA accuracy contract.

The required FPTRANSA_ACCURACY contract is `0x0031`.

Special values

- Positive infinity: the result is +infinity.
- Negative infinity: the result is positive zero.
- Zero source: the result is positive one.

Exact anchors

- $e^{\pm 0} = +1$
- $e^{+\infty} = +\infty$
- $e^{-\infty} = +0$

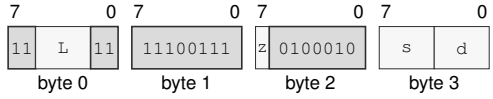
Required properties

- strictly increasing for finite inputs
- result is nonnegative

Encodings:

```
FETOXA.{S|D}(z) Fn(s), Fn(d)
```

Instruction Format:



Format: Instruction format for FETOXA.{S|D}(z) Fn(s), Fn(d)

FETOXM1A Approximate Floating-Point e To x Minus One FETOXM1A

Operation: Approximates e raised to the source minus one without an intermediate rounded exponential.

Assembler Syntax: `FETOXM1A.{S|D}(z) Fn(s), Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`FPTRANSA`

Repeat eligibility: REP eligible

Detailed Semantics

Computes the $e^x - 1$ reference without first rounding e^x over all finite values and signed infinity after DAZ and writes an implementation-bounded S- or D-format approximation under the selected FPTRANSA accuracy contract.

The required FPTRANSA_ACCURACY contract is `0x0032`.

Special values

- Positive infinity: the result is +infinity.
- Negative infinity: the result is -1.
- Zero source: the result is the source.

Exact anchors

- $\mathrm{expm1}(+0) = +0$
- $\mathrm{expm1}(-0) = -0$
- $\mathrm{expm1}(-\infty) = -1$

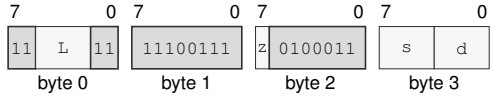
Required properties

- strictly increasing for finite inputs
- result is at least -1

Encodings:

```
FETOXM1A.{S|D}(z) Fn(s), Fn(d)
```

Instruction Format:



Format: Instruction format for FETOXM1A.{S|D}(z) Fn(s), Fn(d)

FLOG10A Approximate Floating-Point Log Base 10 FLOG10A

Operation: Approximates base-10 logarithm for positive values, with zero as the divide-by-zero boundary.

Assembler Syntax: `FLOG10A.{S|D}(z) Fn(s), Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`FPTRANSA`

Repeat eligibility: REP eligible

Detailed Semantics

After DAZ, computes an implementation-bounded S- or D-format approximation of the log10(x) reference under the selected FPTRANSA accuracy contract. The reference domain includes positive finite values and positive infinity; signed zero is the DZ boundary.

The required FPTRANSA_ACCURACY contract is `0x0044`.

Special values

- Zero source: DZ is generated and the result is -infinity.
- Negative source: NV is generated and the result is the architectural default quiet NaN.
- Positive infinity: the result is +infinity.
- Unit source: the result is positive zero.

Exact anchors

- log10(+1) = +0
- log10(signed 0) = -infinity
- log10(+infinity) = +infinity

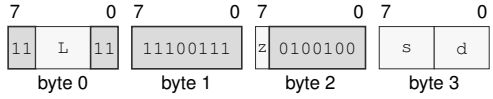
Required properties

- strictly increasing on the positive domain

Encodings:

```
FLOG10A.{S|D}(z) Fn(s), Fn(d)
```

Instruction Format:



Format: Instruction format for FLOG10A.{S|D}(z) Fn(s), Fn(d)

FLOG2A Approximate Floating-Point Log Base 2 FLOG2A

Operation: Approximates base-2 logarithm for positive values, with zero as the divide-by-zero boundary.

Assembler Syntax: `FLOG2A.{S|D}(z) Fn(s), Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`FPTRANSA`

Repeat eligibility: REP eligible

Detailed Semantics

After DAZ, computes an implementation-bounded S- or D-format approximation of the log2(x) reference under the selected FPTRANSA accuracy contract. The reference domain includes positive finite values and positive infinity; signed zero is the DZ boundary.

The required FPTRANSA_ACCURACY contract is `0x0043`.

Special values

- Zero source: DZ is generated and the result is -infinity.
- Negative source: NV is generated and the result is the architectural default quiet NaN.
- Positive infinity: the result is +infinity.
- Unit source: the result is positive zero.

Exact anchors

- log2(+1) = +0
- log2(signed 0) = -infinity
- log2(+infinity) = +infinity

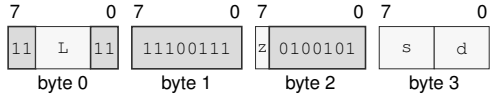
Required properties

- strictly increasing on the positive domain

Encodings:

```
FLOG2A.{S|D}(z) Fn(s), Fn(d)
```

Instruction Format:



Format: Instruction format for FLOG2A.{S|D}(z) Fn(s), Fn(d)

FLOGNA Approximate Floating-Point Natural Log FLOGNA

Operation: Approximates natural logarithm for positive values, with zero as the divide-by-zero boundary.

Assembler Syntax: `FLOGNA.{S|D}(z) Fn(s), Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`FPTRANSA`

Repeat eligibility: REP eligible

Detailed Semantics

After DAZ, computes an implementation-bounded S- or D-format approximation of the ln(x) reference under the selected FPTRANSA accuracy contract. The reference domain includes positive finite values and positive infinity; signed zero is the DZ boundary.

The required FPTRANSA_ACCURACY contract is `0x0041`.

Special values

- Zero source: DZ is generated and the result is -infinity.
- Negative source: NV is generated and the result is the architectural default quiet NaN.
- Positive infinity: the result is +infinity.
- Unit source: the result is positive zero.

Exact anchors

- ln(+1) = +0
- ln(signed 0) = -infinity
- ln(+infinity) = +infinity

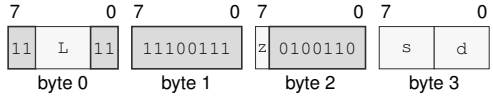
Required properties

- strictly increasing on the positive domain

Encodings:

```
FLOGNA.{S|D}(z) Fn(s), Fn(d)
```

Instruction Format:



Format: Instruction format for FLOGNA.{S|D}(z) Fn(s), Fn(d)

FLOGNP1A Approximate Floating-Point Natural Log Plus One FLOGNP1A

Operation: Approximates natural log of one plus the source, with minus one as the divide-by-zero boundary.

Assembler Syntax: `FLOGNP1A.{S|D}(z) Fn(s), Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`FPTRANSA`

Repeat eligibility: REP eligible

Detailed Semantics

After DAZ, computes an implementation-bounded S- or D-format approximation of the $\ln(1 + x)$ reference without an intermediate rounded addition under the selected FPTRANSA accuracy contract. The reference domain includes finite $x \geq -1$ and positive infinity; -1 is the DZ boundary.

The required FPTRANSA_ACCURACY contract is `0x0042`.

Special values

- Source equal to -1: DZ is generated and the result is -infinity.
- Source below -1: NV is generated and the result is the architectural default quiet NaN.
- Positive infinity: the result is +infinity.
- Zero source: the result is the source.

Exact anchors

- $\mathrm{log1p}(+0) = +0$
- $\mathrm{log1p}(-0) = -0$
- $\mathrm{log1p}(-1) = -\infty$

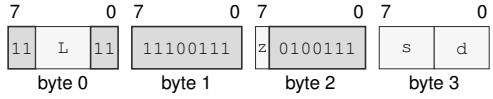
Required properties

- strictly increasing on $(-1, +\infty)$

Encodings:

```
FLOGNP1A.{S|D}(z) Fn(s), Fn(d)
```

Instruction Format:



Format: Instruction format for FLOGNP1A.{S|D}(z) Fn(s), Fn(d)

FSINA — Approximate Floating-Point Sine — FSINA

Operation: Approximates sine in radians over the reduced interval from minus pi/4 to pi/4.

Assembler Syntax: `FSINA.{S|D}(z) Fn(s), Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`FPTRANSA`

Repeat eligibility: REP eligible

Detailed Semantics

Computes the $\sin(x)$ reference in radians over finite x with $|x| \leq \pi/4$ after DAZ and writes an implementation-bounded S- or D-format approximation under the selected FPTRANSA accuracy contract.

The required FPTRANSA_ACCURACY contract is `0x0001`.

Special values

- Infinite source or source magnitude above $\pi/4$: NV is generated and the result is the architectural default quiet NaN.
- Zero source: the result is the source.

Exact anchors

- $\sin(+0) = +0$
- $\sin(-0) = -0$

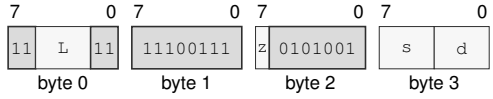
Required properties

- odd
- monotonically increasing on the contract domain
- result is in $[-\sqrt{2}/2, \sqrt{2}/2]$

Encodings:

```
FSINA.{S|D}(z) Fn(s), Fn(d)
```

Instruction Format:



Format: Instruction format for FSINA.{S|D}(z) Fn(s), Fn(d)

FSINCOSA Approximate Floating-Point Sine And Cosine FSINCOSA

Operation: Atomically writes paired sine and cosine approximations over the reduced pi/4 interval.

Assembler Syntax: `FSINCOSA.{S|D}(z) Fn(s), Fn(d), Fn(c)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`FPTRANSA`

Repeat eligibility: REP eligible

Detailed Semantics

Computes the paired $\sin(x)$ and $\cos(x)$ references in radians over finite x with $|x| \leq \pi/4$ after DAZ and writes implementation-bounded S- or D-format approximations under the selected FPTRANSA accuracy contract.

The required FPTRANSA_ACCURACY contract is `0x0004`.

Special values

- Matching destination registers: `INVALID_OPERAND_RELATION` is raised before operands or floating-point state are read.
- Infinite source or source magnitude above $\pi/4$: NV is generated and both results are the architectural default quiet NaN.
- Zero source: the sine result preserves the source zero and the cosine result is positive one.

Exact anchors

- $\sin(+0) = +0$ and $\cos(+0) = +1$
- $\sin(-0) = -0$ and $\cos(-0) = +1$

Required properties

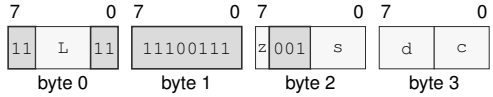
- the ULP bound applies independently to both results
- separate FSINA and FCOSA executions may produce different results

For nontrapping execution, the sine and cosine destinations are written atomically.

Encodings:

`FSINCOSA.{S|D}(z) Fn(s), Fn(d), Fn(c)`

Instruction Format:



Format: Instruction format for FSINCOSA.{S|D}(z) Fn(s), Fn(d), Fn(c)

Restrictions:

Operand overlap: Operands `sin_dst`, `cos_dst`; rule `illegal_instruction`.

FSINHA Approximate Floating-Point Hyperbolic Sine FSINHA

Operation: Approximates hyperbolic sine for finite values and signed infinity.

Assembler Syntax: `FSINHA.{S|D}(z) Fn(s), Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`FPTRANSA`

Repeat eligibility: REP eligible

Detailed Semantics

Computes the sinh(x) reference over all finite values and signed infinity after DAZ and writes an implementation-bounded S- or D-format approximation under the selected FPTRANSA accuracy contract.

The required FPTRANSA_ACCURACY contract is `0x0021`.

Special values

- Infinite or zero source: the result is the source.

Exact anchors

- sinh(+0) = +0
- sinh(-0) = -0
- sinh(infinity) = infinity with the input sign

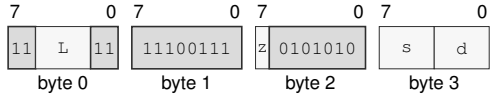
Required properties

- odd
- monotonically increasing

Encodings:

```
FSINHA.{S|D}(z) Fn(s), Fn(d)
```

Instruction Format:



Format: Instruction format for FSINHA.{S|D}(z) Fn(s), Fn(d)

FTANA — Approximate Floating-Point Tangent — FTANA

Operation: Approximates tangent in radians over the reduced interval from minus pi/4 to pi/4.

Assembler Syntax: `FTANA.{S|D}(z) Fn(s), Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`FPTRANSA`

Repeat eligibility: REP eligible

Detailed Semantics

Computes the $\tan(x)$ reference in radians over finite x with $|x| \leq \pi/4$ after DAZ and writes an implementation-bounded S- or D-format approximation under the selected FPTRANSA accuracy contract.

The required FPTRANSA_ACCURACY contract is `0x0003`.

Special values

- Infinite source or source magnitude above $\pi/4$: NV is generated and the result is the architectural default quiet NaN.
- Zero source: the result is the source.

Exact anchors

- $\tan(+0) = +0$
- $\tan(-0) = -0$

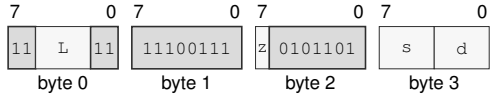
Required properties

- odd
- monotonically increasing on the contract domain
- result is in [-1, 1]

Encodings:

```
FTANA.{S|D}(z) Fn(s), Fn(d)
```

Instruction Format:



Format: Instruction format for FTANA.{S|D}(z) Fn(s), Fn(d)

FTANHA Approximate Floating-Point Hyperbolic Tangent FTANHA

Operation: Approximates hyperbolic tangent for finite values and signed infinity.

Assembler Syntax: `FTANHA.{S|D}(z) Fn(s), Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`FPTRANSA`

Repeat eligibility: REP eligible

Detailed Semantics

Computes the tanh(x) reference over all finite values and signed infinity after DAZ and writes an implementation-bounded S- or D-format approximation under the selected FPTRANSA accuracy contract.

The required FPTRANSA_ACCURACY contract is `0x0023`.

Special values

- Infinite source: the result is one with the sign of the source.
- Zero source: the result is the source.

Exact anchors

- tanh(+0) = +0
- tanh(-0) = -0
- tanh(signed infinity) = signed 1

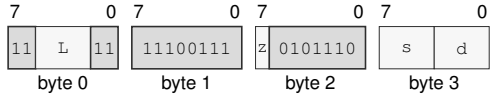
Required properties

- odd
- monotonically increasing
- result is in [-1, 1]

Encodings:

```
FTANHA.{S|D}(z) Fn(s), Fn(d)
```

Instruction Format:



Format: Instruction format for FTANHA.{S|D}(z) Fn(s), Fn(d)

FTENTOXA Approximate Floating-Point 10 To x FTENTOXA

Operation: Approximates ten raised to the source for finite values and signed infinity.

Assembler Syntax: `FTENTOXA.{S|D}(z) Fn(s), Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`FPTRANSA`

Repeat eligibility: REP eligible

Detailed Semantics

Computes the 10^x reference over all finite values and signed infinity after DAZ and writes an implementation-bounded S- or D-format approximation under the selected FPTRANSA accuracy contract.

The required FPTRANSA_ACCURACY contract is `0x0034`.

Special values

- Positive infinity: the result is +infinity.
- Negative infinity: the result is positive zero.
- Zero source: the result is positive one.

Exact anchors

- $10^{\pm 0} = +1$
- $10^{+\infty} = +\infty$
- $10^{-\infty} = +0$

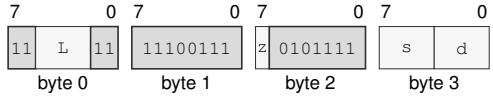
Required properties

- strictly increasing for finite inputs
- result is nonnegative

Encodings:

`FTENTOXA.{S|D}(z) Fn(s), Fn(d)`

Instruction Format:



Format: Instruction format for FTENTOXA.{S|D}(z) Fn(s), Fn(d)

FTWOTOXA Approximate Floating-Point 2 To x FTWOTOXA

Operation: Approximates two raised to the source for finite values and signed infinity.

Assembler Syntax: `FTWOTOXA.{S|D}(z) Fn(s), Fn(d)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`FPTRANSA`

Repeat eligibility: REP eligible

Detailed Semantics

Computes the 2^x reference over all finite values and signed infinity after DAZ and writes an implementation-bounded S- or D-format approximation under the selected FPTRANSA accuracy contract.

The required FPTRANSA_ACCURACY contract is `0x0033`.

Special values

- Positive infinity: the result is +infinity.
- Negative infinity: the result is positive zero.
- Zero source: the result is positive one.

Exact anchors

- $2^{\pm 0} = +1$
- $2^{+\infty} = +\infty$
- $2^{-\infty} = +0$

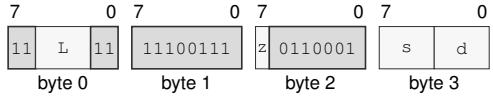
Required properties

- strictly increasing for finite inputs
- result is nonnegative

Encodings:

`FTWOTOXA.{S|D}(z) Fn(s), Fn(d)`

Instruction Format:



Format: Instruction format for FTWOTOXA.{S|D}(z) Fn(s), Fn(d)

23 SCALABLE-VECTOR INSTRUCTIONS

23.1 Summary

Table 23-1. Scalable-Vector Instructions Summary (Informative)

Mnemonic	Brief description
VRESTORE	Restores the vector user-state record.
VSAVE	Saves the vector user-state record.
VDUP	Broadcasts an integer scalar register or immediate value to every vector lane.
VMOV	Copies or merges vector elements between registers and contiguous memory.
PHEAD	Sets predicate positions from the count remainder through the final lane.
PTAIL	Set significant positions before unsigned(Rcount) mod lane_count.
PFIRST	Return the first selected element index at a significant position.
PLAST	Return the last selected element index at a significant position.
PCOUNT	Count set significant positions for the selected element size.
PAND	AND corresponding bits of Ps and the old Pd into the complete Pd image.
POR	OR corresponding bits of Ps and the old Pd into the complete Pd image.
PXOR	XOR corresponding bits of Ps and the old Pd into the complete Pd image.
PUNPKLO	Unpacks the lower half of source predicate positions into wider destination positions.
PUNPKHI	Unpacks the upper half of source predicate positions into wider destination positions.
PPACKLO	Packs source predicate positions into the lower half of narrower destination positions.
PPACKHI	Packs source predicate positions into the upper half of narrower destination positions.
VCLR	Writes zero to every byte of the complete destination vector image.
VINDEX	Writes each logical lane number to its destination element.
VLCNT	Write the number of lanes of the selected element width.
VLCADD	Add a signed immediate multiple of the selected lane count to Rn.
VGATHER	Gather governed vector lanes while journaling completed lanes in a predicate.
VSCATTER	Scatter governed vector lanes while journaling completed lanes in a predicate.
PTRUE	Set every significant position for the selected size.
PFALSE	Clear the complete predicate image.
PNOT	Complements every bit of the complete Pd image.
BPANY	Branch if any bit is set.
BPNONE	Branch if no bit is set.
BPALL	Branch if every significant position for the selected size is set.
VNEG	Negates active integer elements.
VABS	Computes the absolute value of active integer elements.
VNOT	Complements every bit of each active element.
VCLZ	Counts consecutive zero bits from the high end of each active element.
VCTZ	Counts consecutive zero bits from the low end of each active element.
VCLS	Counts consecutive one bits from the high end of each active element.
VCTS	Counts consecutive one bits from the low end of each active element.
VPOPCNT	Counts the set bits in each active element.
VREVBYTE	Reverses the byte order within each active element.
PPERM	Select predicate lanes using unsigned vector indices.
PSLIDEUP	Move predicate lanes toward higher indices and insert zero at the low end.
PSLIDEDN	Move predicate lanes toward lower indices and insert zero at the high end.
VCMPcc	Compares active integer lanes and writes a complete predicate image.

Table 23-1 (continued)

Mnemonic	Brief description
VTESTZ	Sets significant result bits where the governed element-wise AND is zero and clears all other bits.
VTESTNZ	Sets significant result bits where the governed element-wise AND is nonzero and clears all other bits.
VADD	Adds corresponding active integer vector elements.
VSUB	Subtracts corresponding active integer vector elements.
VMUL	Multiplies corresponding active integer elements.
VAND	Applies bitwise AND to corresponding active vector elements.
VOR	Applies bitwise OR to corresponding active vector elements.
VXOR	Applies bitwise XOR to corresponding active vector elements.
VMINS	Selects the lesser signed value from each active destination/source element pair.
VMINU	Selects the lesser unsigned value from each active destination/source element pair.
VMAXS	Selects the greater signed value from each active destination/source element pair.
VMAXU	Selects the greater unsigned value from each active destination/source element pair.
VMULHS	Writes the high half of each active signed destination/source product.
VMULHU	Writes the high half of each active unsigned destination/source product.
VMULHSU	Writes the high half of each active signed-destination by unsigned-source product.
VSHL	Logically shifts each active integer element left by its selected-width count.
VSHR	Logically shifts each active integer element right by its selected-width count.
VSAR	Arithmetically shifts each active integer element right by its selected-width count.
VROL	Rotates each active integer element left by its selected-width count.
VROR	Rotates each active integer element right by its selected-width count.
VEXTZW	Zero-extends B source elements to W destination elements.
VEXTSW	Sign-extends B source elements to W destination elements.
VEXTZL	Zero-extends B or W source elements to L destination elements.
VEXTSL	Sign-extends B or W source elements to L destination elements.
VEXTZQ	Zero-extends B, W, or L source elements to Q destination elements.
VEXTSQ	Sign-extends B, W, or L source elements to Q destination elements.
VTRUNCB	Copies the low B field from each W, L, or Q source container.
VTRUNCW	Copies the low W field from each L or Q source container.
VTRUNCL	Copies the low L field from each Q source container.
VPERM	Select source lanes with vector indices; out-of-range indices produce zero.
VZIPLO	Interleave the lower halves of the old destination and source.
VZIPHI	Interleave the upper halves of the old destination and source.
VUZIPLO	Gather even lanes from the old destination and source into separate result halves.
VUZIPHI	Gather odd lanes from the old destination and source into separate result halves.
VTRNLO	Interleave even lanes from the old destination and source by adjacent result pairs.
VTRNHI	Interleave odd lanes from the old destination and source by adjacent result pairs.
VSLICE	Extract a lane window from the old destination followed by a source vector.
VSLIDEUP	Move old destination lanes toward higher indices and zero vacated lanes.
VSLIDEDN	Move old destination lanes toward lower indices and zero vacated lanes.
VEXTRACT	Copies one indexed integer vector lane to an integer scalar register.
VINSERT	Replaces one indexed vector lane with an integer scalar value.
VREDADD	Reduces active integer elements to one scalar sum.
VREDMINS	Finds the signed minimum of selected integer lanes and writes the scalar result to Rn.

Table 23-1 (continued)

Mnemonic	Brief description
VREDMINU	Finds the unsigned minimum of selected integer lanes and writes the scalar result to Rn.
VREDMAXS	Finds the signed maximum of selected integer lanes and writes the scalar result to Rn.
VREDMAXU	Finds the unsigned maximum of selected integer lanes and writes the scalar result to Rn.
VREDAND	Reduces selected integer lanes with bitwise AND and writes the scalar result to Rn.
VREDOR	Reduces selected integer lanes with bitwise OR and writes the scalar result to Rn.
VREDXOR	Reduces selected integer lanes with bitwise XOR and writes the scalar result to Rn.
PSEL	Replace bits of Pd selected by Pc with bits from Ps.
PZIPLO	Interleave the lower halves of two predicate-lane sequences.
PZIPHI	Interleave the upper halves of two predicate-lane sequences.
PUZIPLO	Concatenate even-position predicate lanes from two sources.
PUZIPHI	Concatenate odd-position predicate lanes from two sources.
PTRNLO	Interleave even-position predicate lanes from two sources.
PTRNHI	Interleave odd-position predicate lanes from two sources.
PSLICE	Extract a predicate-lane window from the old destination followed by a source predicate.
VMOVZ	Loads active memory elements and clears inactive destination lanes.
PLOOP	Builds the next predicate chunk from a remaining count and lane offset.
PMOV	Transfers a complete packed predicate image between a predicate register and memory.

23.2 Element Types and Assembly Spelling

A size-bearing VECTOR instruction has one suffix. The integer element suffixes `B`, `W`, `L`, and `Q` select 1-, 2-, 4-, and 8-byte elements.

The three-bit `tzz` selector values zero through three select `B/W/L/Q`; value four is invalid. The two-bit `zz` selector in integer families assigns all four values to `B/W/L/Q`. A reserved selector does not name another data format.

The suffix is the source width for a width-changing instruction; the destination width and signedness are part of the mnemonic. For example, `VEXTZQ.B` zero-extends a byte into a quadword. Predicate operations, `VLCNT`, `VLCADD`, `VGATHER`, `VSCATTER`, `PLOOP`, `VMOV`, `VMOVZ`, and the permute, slide, slice, and zip families use `B/W/L/Q`; `H/S/D` width-only aliases are not defined.

23.3 Predicates and Destination Images

For an element size of E bytes, logical lane i is governed by predicate bit iE. All other predicate bits are outside the typed predicate image for that operation. Unless an instruction explicitly produces a complete predicate image or uses zeroing movement, a false governing bit annuls that lane: it performs no memory access, produces no exception, and preserves the old destination lane.

An instruction that writes a predicate result writes a complete packed predicate image. It sets or clears each significant result bit and clears all bits outside the typed predicate image. Integer comparisons support `EQ`, `NE`, `ULT`, `UGE`, `ULE`, `UGT`, `LT`, `GE`, `LE`, and `GT`. VECTORFP defines its own floating-point comparison forms. `VTESTZ` and `VTESTNZ` are distinct integer test operations. Vector integer operations do not update scalar `FLAGS`.

Integer reductions return a zero-extended result in an Rn. For an empty predicate, ADD, OR, and XOR return zero; AND and unsigned MIN return the all-ones value of the selected width; signed MIN returns the signed maximum; signed MAX returns the signed minimum; and unsigned MAX returns zero.

PLOOP consumes a remaining-count register and an offset register. If the remaining count is zero, it takes its encoded branch target and does not change the predicate or either register. Otherwise, an offset greater than or equal to the selected lane count raises VECTOR_LANE_INDEX_OUT_OF_RANGE with error code zero and leaves the instruction uncommitted. A valid nonzero iteration activates the next $\min(remaining, lane\_count - offset)$ lanes, subtracts that number from the remaining count, clears the offset register, and falls through. Software advances the offset between chunks when it needs a nonzero subsequent chunk origin.

23.4 Lane Mapping and Scalar Bridges

Vector bytes use increasing-address little-endian order. A same-width operation treats a vector as VLEN divided by the selected element width lanes. For widening and narrowing, the operation container is the larger of the source and destination widths. Each active container reads the source value from its low source-width part and writes the result into its low destination-width part. Other bits of an active destination container and every inactive container remain unchanged.

VDUP broadcasts a scalar register or immediate into every lane. VEXTRACT and VINSERT use a scalar lane index. An index at or above the lane count raises VECTOR_LANE_INDEX_OUT_OF_RANGE with error code one and commits no state. VLCNT returns VLEN in bytes divided by the selected element width in bytes. VLCADD adds its signed eight-bit immediate multiplied by that lane count to its scalar destination modulo 2^{64}. These operations preserve FLAGS and use the integer B/W/L/Q suffixes. VECTORFP provides the corresponding floating-point scalar bridges.

23.5 Vector Memory Operations

A vector memory operand uses the ordinary compact, EXT1, and EXT2 descriptor formats in vector context. The descriptor computes one scalar anchor address. A contiguous access uses

$$address[i] = anchor + iE,$$

All arithmetic is modulo 2^{64}. Existing effective-address auto-update is applied once to the scalar anchor in the ordinary architectural order. That single update covers all lanes.

Inactive lanes issue no memory transaction. VMOV merges a memory source into its destination, while VMOVZ writes zero to inactive destination lanes. A memory-destination arithmetic form uses non-atomic, lane-wise read-modify-write semantics. A vector instruction nevertheless has one architectural commit point: address evaluation, all active reads, computation, and all active writes must succeed before any register, predicate, memory, auto-update, FFLAGS, or PC effect becomes visible. After range, wrap, and canonicality checks, the implementation completes translation of every mapping covered by every active lane in increasing linear-address order. Only after all such translations succeed does it apply physical-class checks, the vector/gather/scatter MMIO-operation prohibition, alignment checks, and target or bus accesses, in that order. Consequently, a translation fault in a higher-address active lane takes precedence over an MMIO condition discovered in a lower-address active lane. When logical lane order differs from address order, translation

and translation-fault selection still follow address order. The reported fault address identifies the lane whose range supplies the selected failure.

23.5.1 Resumable Gather and Scatter

`VGATHER` and `VSCATTER` are full-vector memory instructions with an explicit completion predicate. Their canonical operand orders are illustrated below; T denotes the selected `B/W/L/Q` element width.

```
VGATHER.T  Pg, Ps, [address], Vd
VSCATTER.T Pg, Ps, Vs, [address]
```

Typed completion image. For element width E bytes, let $N = VLEN/(8E)$. Define $G = typed_E(Pg)$ by copying `Pg[iE]` to bit iE for every $0 \le i < N$ and clearing every other predicate bit. At each entry or re-entry, the instruction performs the architectural normalization

$$Ps \leftarrow typed_E(Ps)\ \&\ G, \qquad pending = G\ \&\ {\sim}Ps.$$

`Pg` and `Ps` shall be distinct. The normalization is committed before a lane memory transaction is issued and remains visible if that transaction faults. Consequently inactive completion bits and all bits outside the typed predicate image are zero. If `pending` is empty, the instruction retires without a memory transaction and its normal postcondition is `Ps == G`.

Otherwise the implementation selects any pending lane i. VGATHER performs one E-byte load and, on success, writes only `Vd[i]` before setting `Ps[iE]`. VSCATTER performs one E-byte store of `Vs[i]` and, on success, sets `Ps[iE]`. Every successful lane therefore has one completion bit that journals its externally visible work. VGATHER requires its vector address register to differ from `Vd`; VSCATTER permits its vector address register and `Vs` to be the same register.

Lane-completion boundary. After a successful lane, destination or memory state and the corresponding completion bit commit together. If another lane is pending, the PC remains at the gather or scatter instruction and an asynchronous event may be admitted. This progress boundary is not architectural instruction retirement, does not increment an architectural retired-instruction count, and does not itself generate a debug-trace event. The final empty-pending transition advances the PC and forms the ordinary retirement and trace boundary.

An event handler may emulate or waive a faulting lane by setting its completion bit. For VGATHER, emulation shall first write the lane's destination value; for VSCATTER, it shall first perform or explicitly waive the store. Returning to the saved PC is sufficient to continue: no hidden cursor or implementation token is part of architectural state.

Address forms. The lane number i is implicit and does not appear in assembly syntax. Before default-data-segment translation, the address forms are:

Canonical address	Constraint	Lane offset
`[Rb + Rs]`	any T	$Rb + i \cdot signed64(Rs)$
`[Va]`	`L`	$zero\_extend_{64}(Va.L[i])$
`[Va]`	`Q`	$Va.Q[i]$
`[Rb + Va]`	any T	$Rb + sign\_extend_{64}(Va.T[i])$
`[Rb + Va * scale]`	any T	$Rb + zero\_extend_{64}(Va.T[i])E$
`[Rb + Va * scale + disp]`	any T	preceding indexed address plus `disp`

In the scalar-stride form, `Rs` supplies the signed lane stride even though the canonical spelling is the familiar-looking `[Rb + Rs]`. In vector-index forms, the assembler token `scale` is fixed by T to E; it is not an additional operand. The `disp8s`, `disp16s`, and `disp32s` payloads are sign-extended, while `disp64` contributes its exact 64-bit bit pattern. All address arithmetic is modulo 2^{64}. The absolute forms are still offsets through the default data segment and do not bypass segmentation.

Ordering, overlap, and failures. Lane selection and memory-access order are not architecturally specified. Each successful lane access is nevertheless one ordinary, naturally sized E-byte normal-memory access and obeys the baseline memory-ordering rules. If two VSCATTER lanes target overlapping physical bytes, which lane supplies the final value of each overlapping byte is unspecified. This does not merge, drop, or widen either access: each successful lane still performs its own E-byte store and sets its own completion bit.

A load fault or a scatter failure reported before the store point of no return leaves the selected lane pending and raises the ordinary precise synchronous fault. An ordinary memory bus failure in this state is retry-safe. If a scatter store has become irrevocable, the implementation sets the lane's completion bit and shall not report a restartable synchronous failure for that lane. Any later report is an imprecise `MACHINE_CHECK` delivered at a subsequent lane or retirement boundary after the store becomes irrevocable. These instructions are not permitted for MMIO, including when only one lane is active. Both preserve FLAGS.

23.6 Execution and Saved State

The vector user-state record contains the complete scalable vector and packed predicate images selected by its presence bitmap. A committed `Vn` or `Pn` write marks vector state modified. Architectural event entry does not otherwise change vector state.

An implementation may internally coalesce eligible scalar `REP` or `REPcc` iterations into vector-sized work. This permission creates no vector instruction, predicate, or additional repeat state. The observable counter, condition, `FLAGS`, memory, auto-update, fault, event, and restart behavior must equal the scalar iteration sequence, and only the scalar prefix preceding the first fault or terminating `REPcc` observation may commit.

VRESTORE Restore Vector State VRESTORE

Operation: Restores the vector user-state record.

Assembler Syntax: `VRESTORE [Rn(r)]`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

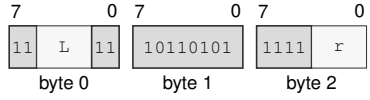
Detailed Semantics

VRESTORE validates the complete VLEN-dependent source range and the 64-byte header before atomically replacing vector user state. A clear bit installs the corresponding initial value without reading that payload slot. An all-clear bitmap makes vector state clean; any set bit makes it modified.

Encodings:

`VRESTORE [Rn(r)]`

Instruction Format:



Format: Instruction format for VRESTORE [Rn(r)]

VSAVE — Save Vector State — VSAVE

Operation: Saves the vector user-state record.

Assembler Syntax: `VSAVE [Rn(r)]`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

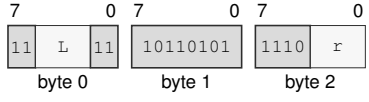
Detailed Semantics

VSAVE writes one 64-byte-aligned vector user-state record sized from VLEN. A clean initial vector state may use an all-clear presence bitmap; otherwise VSAVE may conservatively mark and write every vector and predicate slot. VSAVE does not change clean or modified state.

Encodings:

`VSAVE [Rn(r)]`

Instruction Format:



Format: Instruction format for VSAVE [Rn(r)]

VDUP — Vector Duplicate — VDUP

Operation: Broadcasts an integer scalar register or immediate value to every vector lane.

Assembler Syntax:

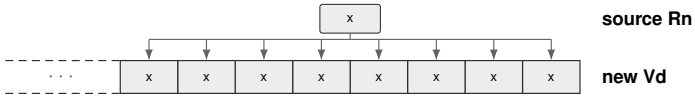
```
VDUP.{B|W|L|Q}(z) Rn(r), Vn(v)
VDUP.B <imm8>, Vn(v)
VDUP.W <imm16>, Vn(v)
VDUP.L <imm32>, Vn(v)
VDUP.Q <imm64>, Vn(v)
```

Privilege: Unprivileged

Required CPUID flags: VECTOR

Detailed Semantics

VDUP fills every selected-width destination lane with the same source value. The integer-register forms use the low B, W, L, or Q field of Rn; the immediate forms broadcast their complete encoded B, W, L, or Q value. The operation replaces the complete destination vector image.

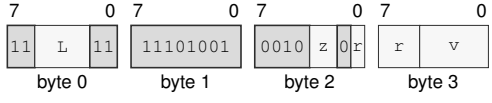


VDUP.W scalar broadcast

Encodings:

VDUP.{B|W|L|Q}(z) Rn(r), Vn(v)

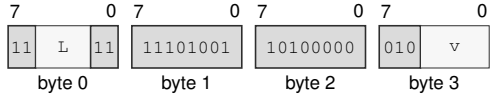
Instruction Format:



Format: Instruction format for VDUP.{B|W|L|Q}(z) Rn(r), Vn(v)

`VDUP.B <imm8>, Vn(v)`

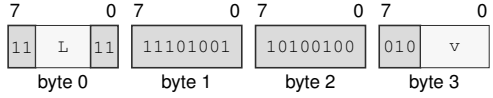
Instruction Format:



Format: Instruction format for VDUP.B <imm8>, Vn(v)

`VDUP.W <imm16>, Vn(v)`

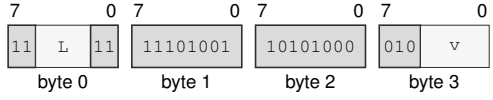
Instruction Format:



Format: Instruction format for VDUP.W <imm16>, Vn(v)

`VDUP.L <imm32>, Vn(v)`

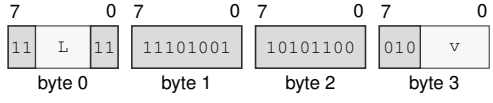
Instruction Format:



Format: Instruction format for VDUP.L <imm32>, Vn(v)

`VDUP.Q <imm64>, Vn(v)`

Instruction Format:



Format: Instruction format for VDUP.Q <imm64>, Vn(v)

VMOV — Vector Move — VMOV

Operation: Copies or merges vector elements between registers and contiguous memory.

Assembler Syntax:

```
VMOV Vn(v), Vn(w)
VMOV.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)
VMOV.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VMOV.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
```

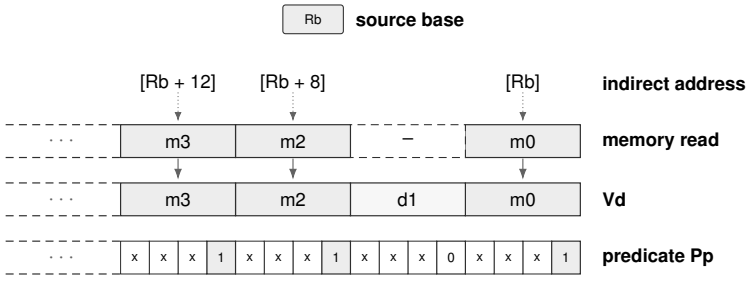
Privilege: Unprivileged

Required CPUID flags: VECTOR

Detailed Semantics

The unsuffixed register form copies the complete source vector image to Vd. The predicated register form replaces each active destination lane with the corresponding Vn lane and retains every inactive destination lane.

For a memory source, the effective address supplies the contiguous anchor and each active lane reads its corresponding element into Vd; inactive lanes issue no access and retain their old values. For a memory destination, each active lane stores its corresponding Vn element and each inactive lane issues no access.

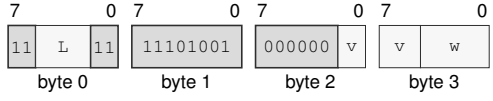


VMOV.L memory load with merge

Encodings:

`VMOV Vn(v), Vn(w)`

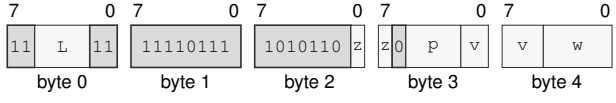
Instruction Format:



Format: Instruction format for VMOV Vn(v), Vn(w)

`VMOV.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

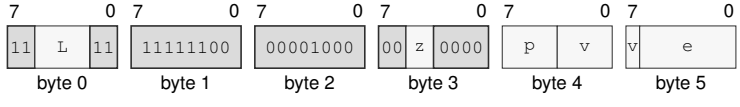
Instruction Format:



Format: Instruction format for VMOV.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

`VMOV.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

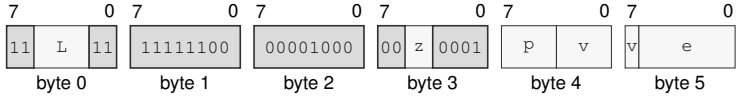
Instruction Format:



Format: Instruction format for VMOV.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VMOV.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VMOV.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

PHEAD — Predicate Head — PHEAD

Operation: Sets predicate positions from the count remainder through the final lane.

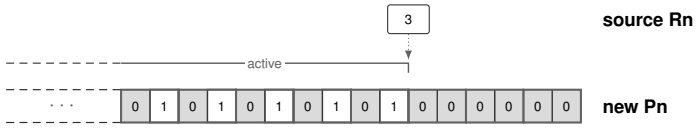
Assembler Syntax: `PHEAD.{B|W|L|Q}(z) Rn(r), Pn(p)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

PHEAD computes the selected-element lane count and starts at the unsigned Rcount remainder modulo that count. It clears predicate positions below the start position and sets positions from the start through the final lane.

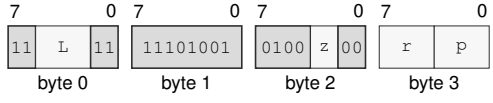


PHEAD.W upper-range predicate fill

Encodings:

`PHEAD.{B|W|L|Q}(z) Rn(r), Pn(p)`

Instruction Format:



Format: Instruction format for PHEAD.{B|W|L|Q}(z) Rn(r), Pn(p)

PTAIL — Predicate Tail — PTAIL

Operation: Set significant positions before unsigned(Rcount) mod lane_count.

Assembler Syntax: `PTAIL.{B|W|L|Q}(z) Rn(r), Pn(p)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

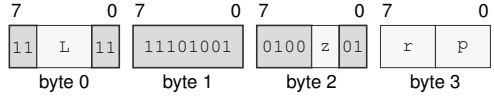
Detailed Semantics

The selected element size determines the logical lane count and the spacing of significant predicate positions. Reducing unsigned Rcount modulo that lane count produces a boundary. PTAIL writes a new Pd image with significant positions below the boundary set and all remaining positions clear. A zero boundary therefore produces an all-clear image.

Encodings:

`PTAIL.{B|W|L|Q}(z) Rn(r), Pn(p)`

Instruction Format:



Format: Instruction format for PTAIL.{B|W|L|Q}(z) Rn(r), Pn(p)

PFIRST First Set Predicate Lane PFIRST

Operation: Return the first selected element index at a significant position.

Assembler Syntax: `PFIRST.{B|W|L|Q}(z) Pn(p), Rn(r)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

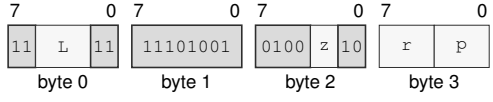
Detailed Semantics

The selected element size defines one significant position in Pn for each logical element. Scanning from element index zero upward, the first set significant position supplies its logical element index to Rn. If the scan finds no set position, Rn receives the full 64-bit all-ones value `UINT64_MAX`.

Encodings:

`PFIRST.{B|W|L|Q}(z) Pn(p), Rn(r)`

Instruction Format:



Format: Instruction format for PFIRST.{B|W|L|Q}(z) Pn(p), Rn(r)

PLAST — Last Set Predicate Lane — PLAST

Operation: Return the last selected element index at a significant position.

Assembler Syntax: `PLAST.{B|W|L|Q}(z) Pn(p), Rn(r)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

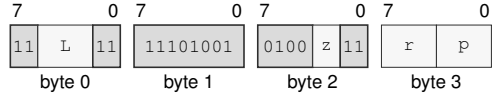
Detailed Semantics

The selected element size defines one significant position in Pn for each logical element. Scanning from the final element toward index zero, the first set significant position encountered supplies its logical element index to Rn. If the scan finds no set position, Rn receives the full 64-bit all-ones value `UINT64_MAX`.

Encodings:

`PLAST.{B|W|L|Q}(z) Pn(p), Rn(r)`

Instruction Format:



Format: Instruction format for PLAST.{B|W|L|Q}(z) Pn(p), Rn(r)

PCOUNT — Predicate Population Count — PCOUNT

Operation: Count set significant positions for the selected element size.

Assembler Syntax: `PCOUNT.{B|W|L|Q}(z) Pn(p), Rn(r)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

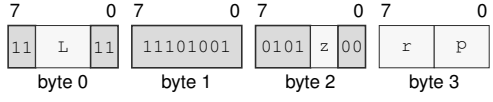
Detailed Semantics

The selected element size maps Pn to one significant predicate position per logical element. PCOUNT scans those positions and writes the number of set significant positions to Rn. Bits between significant positions lie outside the count.

Encodings:

`PCOUNT.{B|W|L|Q}(z) Pn(p), Rn(r)`

Instruction Format:



Format: Instruction format for PCOUNT.{B|W|L|Q}(z) Pn(p), Rn(r)

PAND — Predicate Bitwise AND — PAND

Operation: AND corresponding bits of Ps and the old Pd into the complete Pd image.

Assembler Syntax: `PAND Pn(p), Pn(q)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

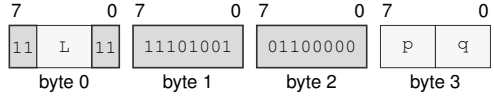
Detailed Semantics

Ps and the previous value of Pd are read as complete packed predicate images. Each result bit is the logical AND of the corresponding source and destination bits. The result replaces the complete Pd image, including both significant and intervening bit positions.

Encodings:

`PAND Pn(p), Pn(q)`

Instruction Format:



Format: Instruction format for PAND Pn(p), Pn(q)

POR — Predicate Bitwise OR — POR

Operation: OR corresponding bits of Ps and the old Pd into the complete Pd image.

Assembler Syntax: `POR Pn(p), Pn(q)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

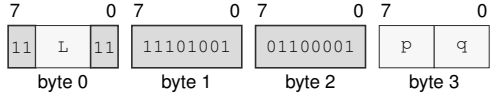
Detailed Semantics

Ps and the previous value of Pd are read as complete packed predicate images. Each result bit is the logical OR of the corresponding source and destination bits. The result replaces the complete Pd image, including both significant and intervening bit positions.

Encodings:

`POR Pn(p), Pn(q)`

Instruction Format:



Format: Instruction format for POR Pn(p), Pn(q)

PXOR — Predicate Bitwise XOR — PXOR

Operation: XOR corresponding bits of Ps and the old Pd into the complete Pd image.

Assembler Syntax: `PXOR Pn(p), Pn(q)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

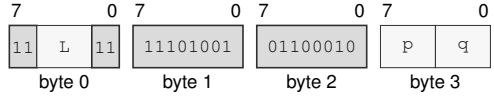
Detailed Semantics

Ps and the previous value of Pd are read as complete packed predicate images. Each result bit is the logical exclusive OR of the corresponding source and destination bits. The result replaces the complete Pd image, including both significant and intervening bit positions.

Encodings:

`PXOR Pn(p), Pn(q)`

Instruction Format:



Format: Instruction format for PXOR Pn(p), Pn(q)

PUNPKLO — Predicate Unpack from Lower Half — PUNPKLO

Operation: Unpacks the lower half of source predicate positions into wider destination positions.

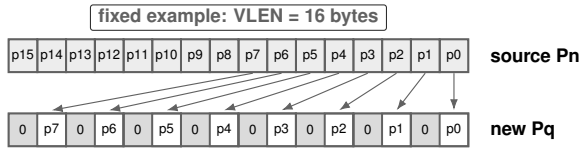
Assembler Syntax: `PUNPKLO.{B|W|L}(z) Pn(p), Pn(q)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

PUNPKLO reads the significant predicate positions for the selected B, W, or L source element width. It selects the lower half of those positions and copies them in lane order to the significant destination positions for elements twice that width. The complete destination predicate image is written, and every bit outside the significant destination positions is cleared. The operation leaves FLAGS unchanged.

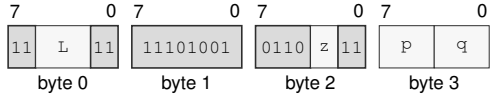


PUNPKLO.B lower-half predicate unpacking

Encodings:

`PUNPKLO.{B|W|L}(z) Pn(p), Pn(q)`

Instruction Format:



Format: Instruction format for PUNPKLO.{B|W|L}(z) Pn(p), Pn(q)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x2`; reason `reserved_vector_element_type`.

PUNPKHI Predicate Unpack from Upper Half PUNPKHI

Operation: Unpacks the upper half of source predicate positions into wider destination positions.

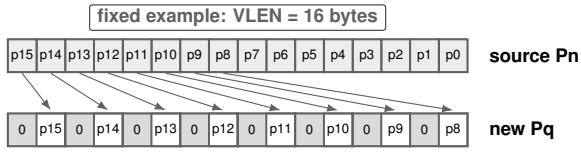
Assembler Syntax: `PUNPKHI.{B|W|L}(z) Pn(p), Pn(q)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

PUNPKHI reads the significant predicate positions for the selected B, W, or L source element width. It selects the upper half of those positions and copies them in lane order to the significant destination positions for elements twice that width. The complete destination predicate image is written, and every bit outside the significant destination positions is cleared. The operation leaves FLAGS unchanged.

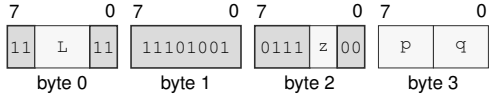


PUNPKHI.B upper-half predicate unpacking

Encodings:

`PUNPKHI.{B|W|L}(z) Pn(p), Pn(q)`

Instruction Format:



Format: Instruction format for PUNPKHI.{B|W|L}(z) Pn(p), Pn(q)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x2`; reason `reserved_vector_element_type`.

PPACKLO — Predicate Pack to Lower Half — PPACKLO

Operation: Packs source predicate positions into the lower half of narrower destination positions.

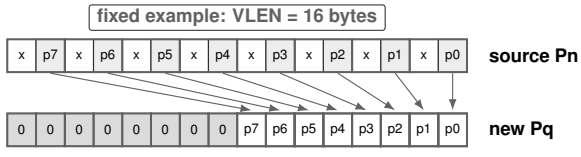
Assembler Syntax: `PPACKLO.{W|L|Q}(z) Pn(p), Pn(q)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

PPACKLO reads the significant predicate positions for the selected W, L, or Q source element width. It copies those positions in lane order to the lower half of the significant destination positions for elements half that width. The complete destination predicate image is written: the upper half of its significant positions and every bit outside its significant positions are cleared. The operation leaves FLAGS unchanged.

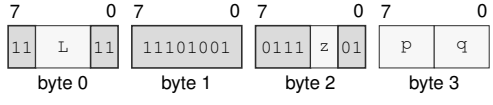


PPACKLO.W lower-half predicate packing

Encodings:

`PPACKLO.{W|L|Q}(z) Pn(p), Pn(q)`

Instruction Format:



Format: Instruction format for PPACKLO.{W|L|Q}(z) Pn(p), Pn(q)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

PPACKHI — Predicate Pack to Upper Half — PPACKHI

Operation: Packs source predicate positions into the upper half of narrower destination positions.

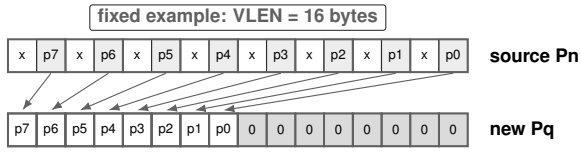
Assembler Syntax: `PPACKHI.{W|L|Q}(z) Pn(p), Pn(q)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

PPACKHI reads the significant predicate positions for the selected W, L, or Q source element width. It copies those positions in lane order to the upper half of the significant destination positions for elements half that width. The complete destination predicate image is written: the lower half of its significant positions and every bit outside its significant positions are cleared. The operation leaves FLAGS unchanged.

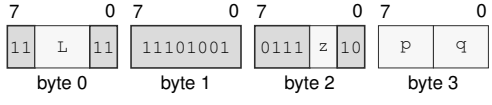


PPACKHI.W upper-half predicate packing

Encodings:

`PPACKHI.{W|L|Q}(z) Pn(p), Pn(q)`

Instruction Format:



Format: Instruction format for PPACKHI.{W|L|Q}(z) Pn(p), Pn(q)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

VCLR — Vector Clear — VCLR

Operation: Writes zero to every byte of the complete destination vector image.

Assembler Syntax: `VCLR Vn(v)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

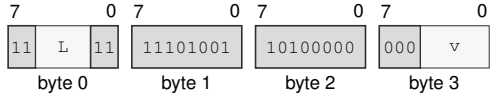
Detailed Semantics

The operation writes zero to every byte of the destination vector register for the current VLEN.

Encodings:

`VCLR Vn(v)`

Instruction Format:



Format: Instruction format for VCLR Vn(v)

VINDEX — Vector Lane Indices — VINDEX

Operation: Writes each logical lane number to its destination element.

Assembler Syntax: `VINDEX.{B|W|L|Q}(z) Vn(v)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

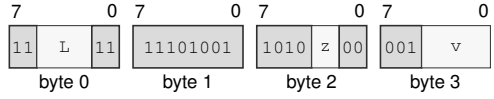
Detailed Semantics

For an element width of E bytes, the destination contains $\mathrm{VLEN}/(8E)$ logical lanes. Lane i receives the unsigned integer i encoded in its E-byte element.

Encodings:

`VINDEX.{B|W|L|Q}(z) Vn(v)`

Instruction Format:



Format: Instruction format for VINDEX.{B|W|L|Q}(z) Vn(v)

VLCNT — Vector Lane Count — VLCNT

Operation: Write the number of lanes of the selected element width.

Assembler Syntax: `VLCNT.{B|W|L|Q}(z) Rn(r)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

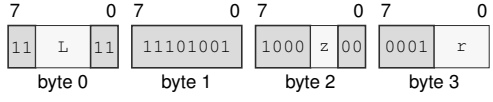
Detailed Semantics

VLCNT writes the selected vector lane count to the complete 64-bit `Rn` destination. The count is VLEN in bytes divided by the selected B, W, L, or Q element width in bytes. The operation leaves FLAGS unchanged.

Encodings:

`VLCNT.{B|W|L|Q}(z) Rn(r)`

Instruction Format:



Format: Instruction format for VLCNT.{B|W|L|Q}(z) Rn(r)

VLCADD Vector Lane-Count Add VLCADD

Operation: Add a signed immediate multiple of the selected lane count to Rn.

Assembler Syntax: `VLCADD.{B|W|L|Q}(z) <imm8s>, Rn(r)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

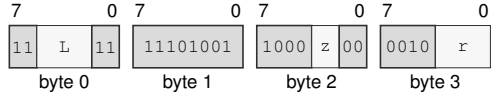
Detailed Semantics

VLCADD reads the old `Rn` value and adds the signed eight-bit immediate multiplied by the selected vector lane count. The lane count is VLEN in bytes divided by the selected B, W, L, or Q element width in bytes, and the addition is modulo 2^{64}. The operation leaves FLAGS unchanged.

Encodings:

`VLCADD.{B|W|L|Q}(z) <imm8s>, Rn(r)`

Instruction Format:



Format: Instruction format for VLCADD.{B|W|L|Q}(z) <imm8s>, Rn(r)

VGATHER — Resumable Vector Gather — VGATHER

Operation: Gather governed vector lanes while journaling completed lanes in a predicate.

Assembler Syntax:

```
VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Rn(s)], Vn(v)
VGATHER.L Pn(p), Pn(c), [Vn(x)], Vn(v)
VGATHER.Q Pn(p), Pn(c), [Vn(x)], Vn(v)
VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Vn(x)], Vn(v)
VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Vn(x) * <scale>],
Vn(v)
VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Vn(x) * <scale> +
<disp8s>], Vn(v)
VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Vn(x) * <scale> +
<disp16s>], Vn(v)
VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Vn(x) * <scale> +
<disp32s>], Vn(v)
VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Vn(x) * <scale> +
<disp64>], Vn(v)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

Let E be the selected element width in bytes, $N = VLEN/(8E)$, and G be the typed governing-predicate image whose significant bits are `Pg[iE]` for $0 \leq i < N$. `Pg` and `Ps` shall name different predicate registers. Before attempting a lane, VGATHER replaces `Ps` by its typed image `Ps AND G`; all bits outside the typed predicate image are cleared. Thus completed lanes that are no longer governed are discarded.

If `G AND NOT Ps` is empty, the instruction completes normally with `Ps == G`. Otherwise it selects one pending lane i, calculates that lane's effective address from the selected address form, and issues exactly one E-byte normal-memory load through the default data segment. On success it replaces only destination lane i, sets `Ps[iE]`, and commits those effects at a lane-completion boundary. If more lanes remain pending, the PC continues to identify this VGATHER and execution may admit an asynchronous event before another lane is attempted. A lane-completion boundary is not an instruction retirement or a debug-trace boundary.

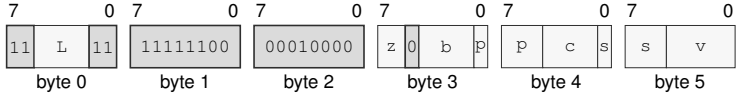
A synchronous memory fault commits no destination value and does not set the faulting lane's completion bit; the typed normalization of `Ps` remains committed, so retrying at the saved PC does not repeat an already completed lane. Software emulation of a faulting lane shall write the destination lane before setting its completion bit. The vector address and destination operands

shall not name the same vector register. VGATHER preserves FLAGS and is not permitted for MMIO.

Encodings:

```
VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Rn(s)], Vn(v)
```

Instruction Format:



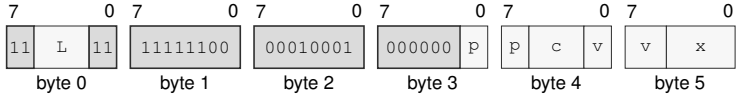
Format: Instruction format for VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Rn(s)], Vn(v)

Restrictions:

Operand overlap: Operands `govern`, `completion`; rule `illegal_instruction`.

```
VGATHER.L Pn(p), Pn(c), [Vn(x)], Vn(v)
```

Instruction Format:



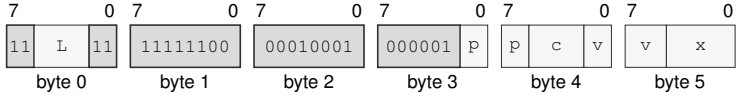
Format: Instruction format for VGATHER.L Pn(p), Pn(c), [Vn(x)], Vn(v)

Restrictions:

Operand overlap: Operands `govern`, `completion`; rule `illegal_instruction`.
Operand overlap: Operands `address`, `dst`; rule `illegal_instruction`.

```
VGATHER.Q Pn(p), Pn(c), [Vn(x)], Vn(v)
```

Instruction Format:



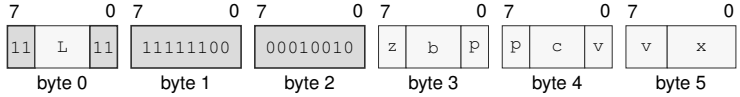
Format: Instruction format for VGATHER.Q Pn(p), Pn(c), [Vn(x)], Vn(v)

Restrictions:

Operand overlap: Operands `govern`, `completion`; rule `illegal_instruction`.
Operand overlap: Operands `address`, `dst`; rule `illegal_instruction`.

`VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Vn(x)], Vn(v)`

Instruction Format:



Format: Instruction format for VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Vn(x)], Vn(v)

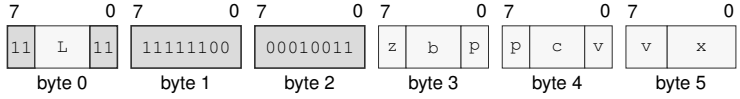
Restrictions:

Operand overlap: Operands `govern`, `completion`; rule `illegal_instruction`.

Operand overlap: Operands `address`, `dst`; rule `illegal_instruction`.

`VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Vn(x) * <scale>], Vn(v)`

Instruction Format:



Format: Instruction format for VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Vn(x) * <scale>], Vn(v)

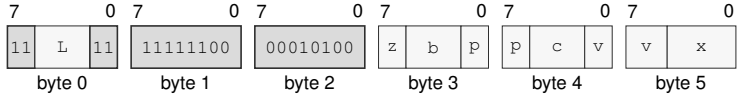
Restrictions:

Operand overlap: Operands `govern`, `completion`; rule `illegal_instruction`.

Operand overlap: Operands `address`, `dst`; rule `illegal_instruction`.

`VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Vn(x) * <scale> + <disp8s>], Vn(v)`

Instruction Format:



Format: Instruction format for VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Vn(x) * <scale> + <disp8s>], Vn(v)

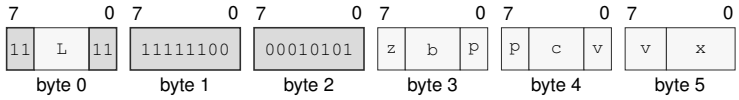
Restrictions:

Operand overlap: Operands `govern`, `completion`; rule `illegal_instruction`.

Operand overlap: Operands `address`, `dst`; rule `illegal_instruction`.

`VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Vn(x) * <scale> + <disp16s>], Vn(v)`

Instruction Format:



Format: Instruction format for VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Vn(x) * <scale> + <disp16s>], Vn(v)

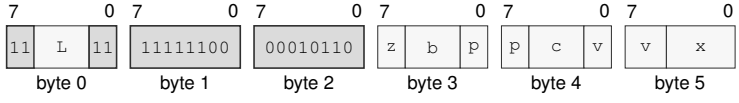
Restrictions:

Operand overlap: Operands `govern`, `completion`; rule `illegal_instruction`.

Operand overlap: Operands `address`, `dst`; rule `illegal_instruction`.

`VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Vn(x) * <scale> + <disp32s>], Vn(v)`

Instruction Format:



Format: Instruction format for VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Vn(x) * <scale> + <disp32s>], Vn(v)

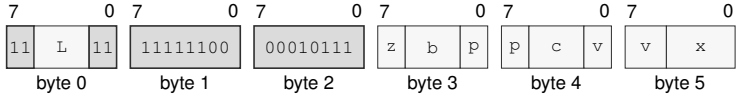
Restrictions:

Operand overlap: Operands `govern`, `completion`; rule `illegal_instruction`.

Operand overlap: Operands `address`, `dst`; rule `illegal_instruction`.

`VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Vn(x) * <scale> + <disp64>], Vn(v)`

Instruction Format:



Format: Instruction format for VGATHER.{B|W|L|Q}(z) Pn(p), Pn(c), [Rn(b) + Vn(x) * <scale> + <disp64>], Vn(v)

Restrictions:

Operand overlap: Operands `govern`, `completion`; rule `illegal_instruction`.

Operand overlap: Operands `address`, `dst`; rule `illegal_instruction`.

VSCATTER — Resumable Vector Scatter — VSCATTER

Operation: Scatter governed vector lanes while journaling completed lanes in a predicate.

Assembler Syntax:

```
VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Rn(s)]
VSCATTER.L Pn(p), Pn(c), Vn(v), [Vn(x)]
VSCATTER.Q Pn(p), Pn(c), Vn(v), [Vn(x)]
VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Vn(x)]
VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Vn(x) *
<scale>]
VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Vn(x) *
<scale> + <disp8s>]
VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Vn(x) *
<scale> + <disp16s>]
VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Vn(x) *
<scale> + <disp32s>]
VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Vn(x) *
<scale> + <disp64>]
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

Let E be the selected element width in bytes, $N = VLEN/(8E)$, and G be the typed governing-predicate image whose significant bits are `Pg[iE]` for $0 \le i < N$. `Pg` and `Ps` shall name different predicate registers. Before attempting a lane, VSCATTER replaces `Ps` by its typed image `Ps AND G`; all bits outside the typed predicate image are cleared.

If `G AND NOT Ps` is empty, the instruction completes normally with `Ps == G`. Otherwise it selects one pending lane i, calculates that lane's effective address from the selected address form, and issues exactly one E-byte normal-memory store of source lane i through the default data segment. On success it sets `Ps[iE]` and commits the store and completion bit at a lane-completion boundary. If more lanes remain pending, the PC continues to identify this VSCATTER and execution may admit an asynchronous event before another lane is attempted. A lane-completion boundary is not an instruction retirement or a debug-trace boundary.

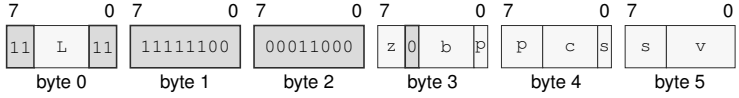
Lane selection order is not architecturally specified. When stores from two lanes overlap in physical memory, the final value of each overlapping byte is therefore unspecified; every successful lane nevertheless performs exactly one E-byte store and sets its completion bit. A failure reported before the store's point of no return leaves that bit clear and is retry-safe. A failure reported after the store becomes irrevocable sets the bit and defers an imprecise machine check to a subsequent boundary. Software emulation shall perform or explicitly waive the store before setting the

completion bit. The address and source vector may name the same vector register. VSCATTER preserves FLAGS and is not permitted for MMIO.

Encodings:

```
VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Rn(s)]
```

Instruction Format:



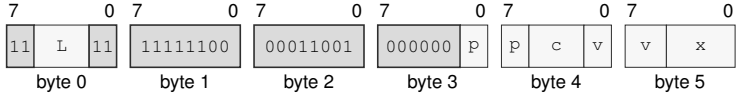
Format: Instruction format for VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Rn(s)]

Restrictions:

Operand overlap: Operands `govern`, `completion`; rule `illegal_instruction`.

```
VSCATTER.L Pn(p), Pn(c), Vn(v), [Vn(x)]
```

Instruction Format:



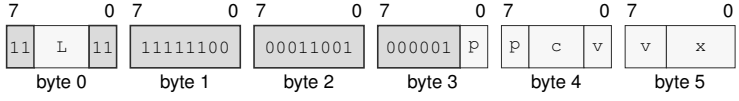
Format: Instruction format for VSCATTER.L Pn(p), Pn(c), Vn(v), [Vn(x)]

Restrictions:

Operand overlap: Operands `govern`, `completion`; rule `illegal_instruction`.

```
VSCATTER.Q Pn(p), Pn(c), Vn(v), [Vn(x)]
```

Instruction Format:



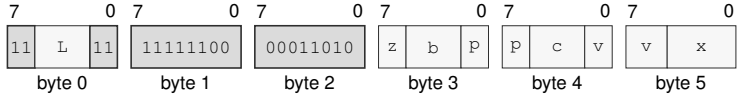
Format: Instruction format for VSCATTER.Q Pn(p), Pn(c), Vn(v), [Vn(x)]

Restrictions:

Operand overlap: Operands `govern`, `completion`; rule `illegal_instruction`.

```
VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Vn(x)]
```

Instruction Format:



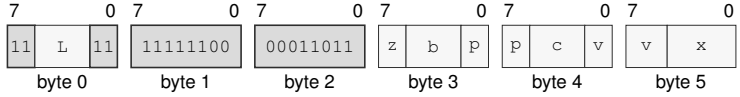
Format: Instruction format for VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Vn(x)]

Restrictions:

Operand overlap: Operands `govern`, `completion`; rule `illegal_instruction`.

```
VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Vn(x) * <scale>]
```

Instruction Format:



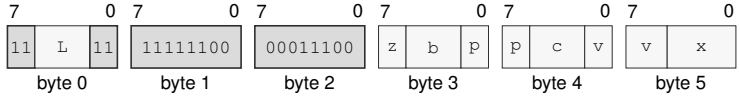
Format: Instruction format for VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Vn(x) * <scale>]

Restrictions:

Operand overlap: Operands `govern`, `completion`; rule `illegal_instruction`.

```
VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Vn(x) * <scale> + <disp8s>]
```

Instruction Format:



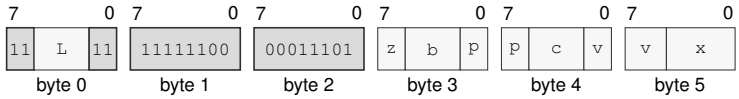
Format: Instruction format for VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Vn(x) * <scale> + <disp8s>]

Restrictions:

Operand overlap: Operands `govern`, `completion`; rule `illegal_instruction`.

`VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Vn(x) * <scale> + <disp16s>]`

Instruction Format:



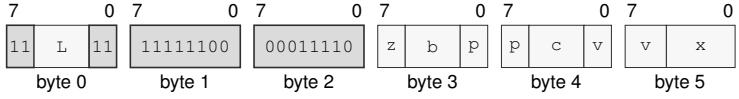
Format: Instruction format for VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Vn(x) * <scale> + <disp16s>]

Restrictions:

Operand overlap: Operands `govern`, `completion`; rule `illegal_instruction`.

`VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Vn(x) * <scale> + <disp32s>]`

Instruction Format:



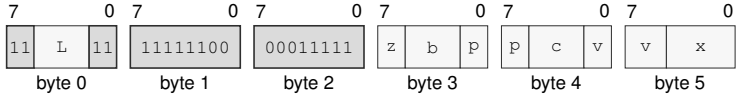
Format: Instruction format for VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Vn(x) * <scale> + <disp32s>]

Restrictions:

Operand overlap: Operands `govern`, `completion`; rule `illegal_instruction`.

`VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Vn(x) * <scale> + <disp64>]`

Instruction Format:



Format: Instruction format for VSCATTER.{B|W|L|Q}(z) Pn(p), Pn(c), Vn(v), [Rn(b) + Vn(x) * <scale> + <disp64>]

Restrictions:

Operand overlap: Operands `govern`, `completion`; rule `illegal_instruction`.

PTRUE — Predicate Set All — PTRUE

Operation: Set every significant position for the selected size.

Assembler Syntax: `PTRUE.{B|W|L|Q}(z) Pn(p)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

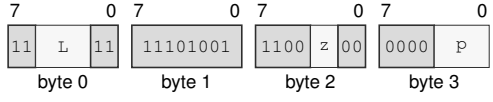
Detailed Semantics

The selected B, W, L, or Q element size spaces one significant predicate position at the start of each logical lane. PTRUE writes a new Pd image with those positions set and every intervening position clear.

Encodings:

`PTRUE.{B|W|L|Q}(z) Pn(p)`

Instruction Format:



Format: Instruction format for PTRUE.{B|W|L|Q}(z) Pn(p)

PFALSE — Predicate Clear — PFALSE

Operation: Clear the complete predicate image.

Assembler Syntax: `PFALSE Pn(p)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

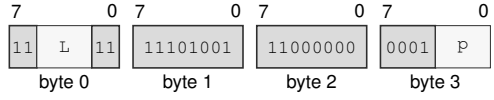
Detailed Semantics

Pd names the destination packed predicate register. Every bit position in its image is written with zero, covering the significant positions for every element size and all intervening positions.

Encodings:

`PFALSE Pn(p)`

Instruction Format:



Format: Instruction format for PFALSE Pn(p)

PNOT — Predicate Bitwise NOT — PNOT

Operation: Complements every bit of the complete Pd image.

Assembler Syntax: `PNOT Pn(p)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

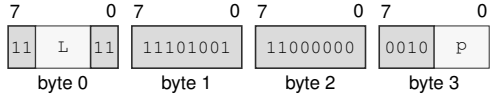
Detailed Semantics

The previous value of Pd is read as a complete packed predicate image. Every bit is complemented and the resulting image replaces Pd, covering both significant and intervening bit positions.

Encodings:

`PNOT Pn(p)`

Instruction Format:



Format: Instruction format for PNOT Pn(p)

BPANY — Branch on Any Predicate Bit — BPANY

Operation: Branch if any bit is set.

Assembler Syntax:
```
BPANY Pn(p), <imm32s>
BPANY Pn(p), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: VECTOR

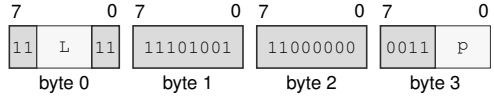
Detailed Semantics

Pn is read as a complete packed predicate image, so every stored bit participates in the branch decision. A set bit selects the target operand; an all-zero image selects the next instruction address.

Encodings:

`BPANY Pn(p), <imm32s>`

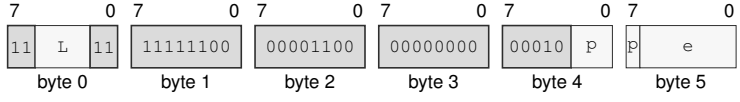
Instruction Format:



Format: Instruction format for BPANY Pn(p), <imm32s>

`BPANY Pn(p), <ea>(e)`

Instruction Format:



Format: Instruction format for BPANY Pn(p), <ea>(e)

BPNONE — Branch on No Predicate Bits — BPNONE

Operation: Branch if no bit is set.

Assembler Syntax:
```
BPNONE Pn(p), <imm32s>
BPNONE Pn(p), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

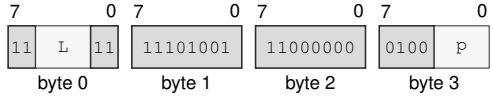
Detailed Semantics

Pn is read as a complete packed predicate image, with every stored bit included in the branch decision. An all-zero image selects the target operand; any set bit selects the next instruction address.

Encodings:

`BPNONE Pn(p), <imm32s>`

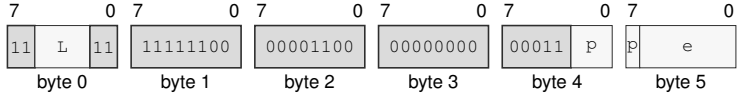
Instruction Format:



Format: Instruction format for BPNONE Pn(p), <imm32s>

`BPNONE Pn(p), <ea>(e)`

Instruction Format:



Format: Instruction format for BPNONE Pn(p), <ea>(e)

BPALL Branch on All Significant Predicate Positions BPALL

Operation: Branch if every significant position for the selected size is set.

Assembler Syntax:
```
BPALL.{B|W|L|Q}(z) Pn(p), <imm32s>
BPALL.{B|W|L|Q}(z) Pn(p), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: VECTOR

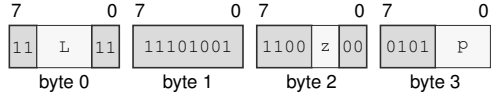
Detailed Semantics

Pn supplies one significant predicate position for each lane of the selected B, W, L, or Q element size. The target operand is selected when all of those positions are set; otherwise execution continues at the next instruction. The branch decision is derived solely from the significant positions.

Encodings:

`BPALL.{B|W|L|Q}(z) Pn(p), <imm32s>`

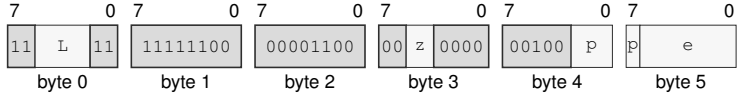
Instruction Format:



Format: Instruction format for BPALL.{B|W|L|Q}(z) Pn(p), <imm32s>

`BPALL.{B|W|L|Q}(z) Pn(p), <ea>(e)`

Instruction Format:



Format: Instruction format for BPALL.{B|W|L|Q}(z) Pn(p), <ea>(e)

VNEG — Vector Negate — VNEG

Operation: Negates active integer elements.

Assembler Syntax:

```
VNEG.{B|W|L|Q}(x) Pn(p), Vn(v)
VNEG.{B|W|L|Q}(x) Pn(p), <ea>(e), Vn(v)
VNEG.{B|W|L|Q}(x) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

For each active B, W, L, or Q element, VNEG replaces the element with its selected-width two's-complement negation.

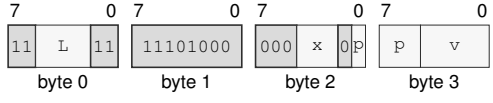
The register form reads and overwrites each active destination lane. The memory-source form reads each active memory element and writes the result to the corresponding destination lane. An inactive destination lane retains its previous value.

The memory-destination form transforms each active source-register element and writes the result to the corresponding memory element. Inactive memory elements retain their previous values, making the destination transaction a lane-wise read-modify-write.

Encodings:

`VNEG.{B|W|L|Q}(x) Pn(p), Vn(v)`

Instruction Format:



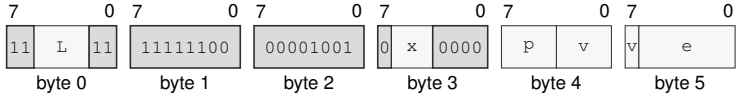
Format: Instruction format for VNEG.{B|W|L|Q}(x) Pn(p), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x3`; reason `reserved_vector_element_type`.

`VNEG.{B|W|L|Q}(x) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



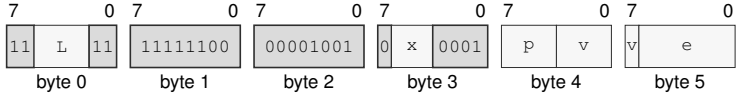
Format: Instruction format for VNEG.{B|W|L|Q}(x) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x3`; reason `reserved_vector_element_type`.

`VNEG.{B|W|L|Q}(x) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VNEG.{B|W|L|Q}(x) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x3`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VABS — Vector Absolute Value — VABS

Operation: Computes the absolute value of active integer elements.

Assembler Syntax:

```
VABS.{B|W|L|Q}(x) Pn(p), Vn(v)
VABS.{B|W|L|Q}(x) Pn(p), <ea>(e), Vn(v)
VABS.{B|W|L|Q}(x) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

For each active B, W, L, or Q element, VABS replaces the element with its selected-width signed absolute value. The result is reduced modulo the selected width, so the most-negative value retains its bit pattern.

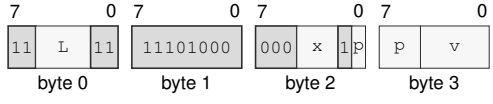
The register form reads and overwrites each active destination lane. The memory-source form reads each active memory element and writes the result to the corresponding destination lane. An inactive destination lane retains its previous value.

The memory-destination form transforms each active source-register element and writes the result to the corresponding memory element. Inactive memory elements retain their previous values, making the destination transaction a lane-wise read-modify-write.

Encodings:

`VABS.{B|W|L|Q}(x) Pn(p), Vn(v)`

Instruction Format:



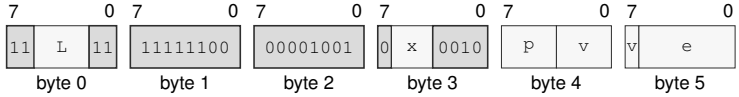
Format: Instruction format for VABS.{B|W|L|Q}(x) Pn(p), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x3`; reason `reserved_vector_element_type`.

```
VABS.{B|W|L|Q}(x) Pn(p), <ea>(e), Vn(v)
```

Instruction Format:



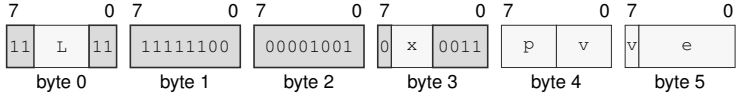
Format: Instruction format for VABS.{B|W|L|Q}(x) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x3`; reason `reserved_vector_element_type`.

```
VABS.{B|W|L|Q}(x) Pn(p), Vn(v), <ea>(e)
```

Instruction Format:



Format: Instruction format for VABS.{B|W|L|Q}(x) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x3`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VNOT — Vector Bitwise NOT — VNOT

Operation: Complements every bit of each active element.

Assembler Syntax:

```
VNOT.{B|W|L|Q}(z) Pn(p), Vn(v)
VNOT.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VNOT.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

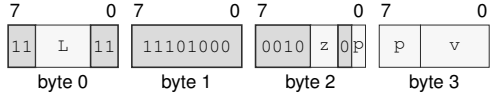
The register form reads and overwrites each active destination lane. The memory-source form complements each active memory element and writes it to the corresponding destination lane. An inactive destination lane retains its previous value.

The memory-destination form complements each active source-register lane and writes it to the corresponding memory element. An inactive memory element remains unchanged. The complement is confined to the selected element width.

Encodings:

`VNOT.{B|W|L|Q}(z) Pn(p), Vn(v)`

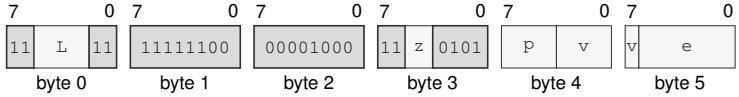
Instruction Format:



Format: Instruction format for VNOT.{B|W|L|Q}(z) Pn(p), Vn(v)

`VNOT.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

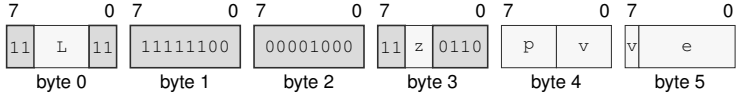
Instruction Format:



Format: Instruction format for VNOT.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VNOT.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VNOT.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VCLZ — Vector Count Leading Zeros — VCLZ

Operation: Counts consecutive zero bits from the high end of each active element.

Assembler Syntax:

```
VCLZ.{B|W|L|Q}(z) Pn(p), Vn(v)
VCLZ.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VCLZ.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

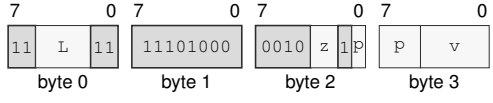
The register form replaces each active destination lane with the count of consecutive zero bits at the high end of its old selected-width value. The memory-source form counts leading zero bits in each active memory element and writes the result to the corresponding destination lane. An inactive destination lane retains its previous value.

The memory-destination form counts leading zero bits in each active source-register lane and writes the result to the corresponding memory element. An inactive memory element remains unchanged. If every bit in an element is zero, the result equals the element width in bits.

Encodings:

`VCLZ.{B|W|L|Q}(z) Pn(p), Vn(v)`

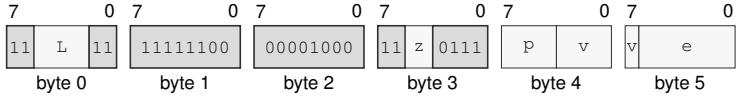
Instruction Format:



Format: Instruction format for VCLZ.{B|W|L|Q}(z) Pn(p), Vn(v)

`VCLZ.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

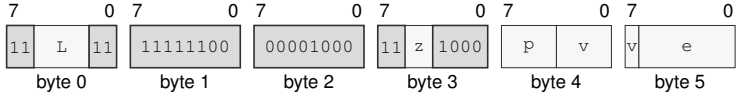
Instruction Format:



Format: Instruction format for VCLZ.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VCLZ.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VCLZ.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VCTZ — Vector Count Trailing Zeros — VCTZ

Operation: Counts consecutive zero bits from the low end of each active element.

Assembler Syntax:

```
VCTZ.{B|W|L|Q}(z) Pn(p), Vn(v)
VCTZ.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VCTZ.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

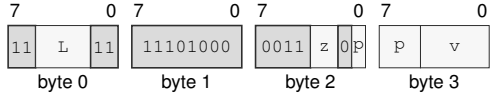
The register form replaces each active destination lane with the count of consecutive zero bits at the low end of its old selected-width value. The memory-source form counts trailing zero bits in each active memory element and writes the result to the corresponding destination lane. An inactive destination lane retains its previous value.

The memory-destination form counts trailing zero bits in each active source-register lane and writes the result to the corresponding memory element. An inactive memory element remains unchanged. If every bit in an element is zero, the result equals the element width in bits.

Encodings:

`VCTZ.{B|W|L|Q}(z) Pn(p), Vn(v)`

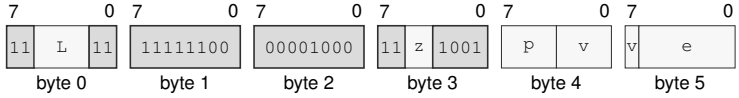
Instruction Format:



Format: Instruction format for VCTZ.{B|W|L|Q}(z) Pn(p), Vn(v)

`VCTZ.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

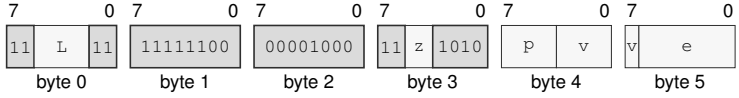
Instruction Format:



Format: Instruction format for VCTZ.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VCTZ.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VCTZ.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VCLS — Vector Count Leading Ones — VCLS

Operation: Counts consecutive one bits from the high end of each active element.

Assembler Syntax:

```
VCLS.{B|W|L|Q}(z) Pn(p), Vn(v)
VCLS.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VCLS.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

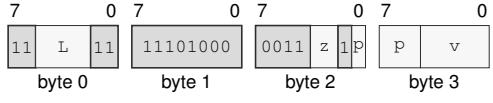
The register form replaces each active destination lane with the count of consecutive one bits at the high end of its old selected-width value. The memory-source form counts leading one bits in each active memory element and writes the result to the corresponding destination lane. An inactive destination lane retains its previous value.

The memory-destination form counts leading one bits in each active source-register lane and writes the result to the corresponding memory element. An inactive memory element remains unchanged. If every bit in an element is one, the result equals the element width in bits.

Encodings:

`VCLS.{B|W|L|Q}(z) Pn(p), Vn(v)`

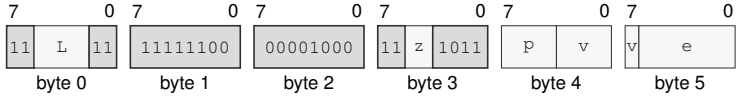
Instruction Format:



Format: Instruction format for VCLS.{B|W|L|Q}(z) Pn(p), Vn(v)

`VCLS.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

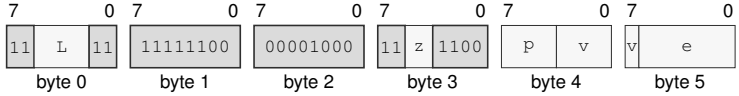
Instruction Format:



Format: Instruction format for VCLS.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VCLS.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VCLS.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VCTS — Vector Count Trailing Ones — VCTS

Operation: Counts consecutive one bits from the low end of each active element.

Assembler Syntax:
```
VCTS.{B|W|L|Q}(z) Pn(p), Vn(v)
VCTS.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VCTS.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

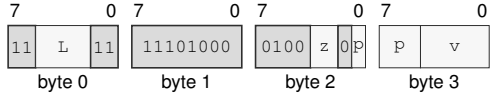
The register form replaces each active destination lane with the count of consecutive one bits at the low end of its old selected-width value. The memory-source form counts trailing one bits in each active memory element and writes the result to the corresponding destination lane. An inactive destination lane retains its previous value.

The memory-destination form counts trailing one bits in each active source-register lane and writes the result to the corresponding memory element. An inactive memory element remains unchanged. If every bit in an element is one, the result equals the element width in bits.

Encodings:

`VCTS.{B|W|L|Q}(z) Pn(p), Vn(v)`

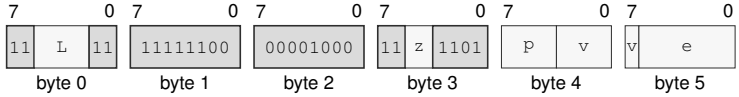
Instruction Format:



Format: Instruction format for VCTS.{B|W|L|Q}(z) Pn(p), Vn(v)

`VCTS.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

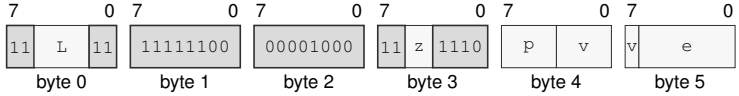
Instruction Format:



Format: Instruction format for VCTS.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VCTS.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VCTS.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VPOPCNT — Vector Population Count — VPOPCNT

Operation: Counts the set bits in each active element.

Assembler Syntax:

```
VPOPCNT.{B|W|L|Q}(z) Pn(p), Vn(v)
VPOPCNT.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VPOPCNT.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

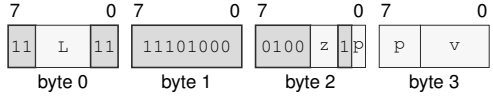
The register form replaces each active destination lane with the population count: the number of one bits in its old selected-width value. The memory-source form counts the set bits in each active memory element and writes the result to the corresponding destination lane. An inactive destination lane retains its previous value.

The memory-destination form counts the set bits in each active source-register lane and writes the result to the corresponding memory element. An inactive memory element remains unchanged. The count covers exactly the bits in the selected element width.

Encodings:

`VPOPCNT.{B|W|L|Q}(z) Pn(p), Vn(v)`

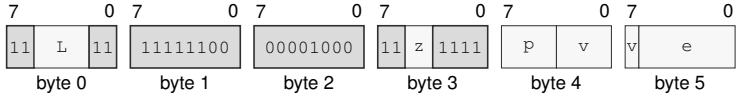
Instruction Format:



Format: Instruction format for VPOPCNT.{B|W|L|Q}(z) Pn(p), Vn(v)

`VPOPCNT.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

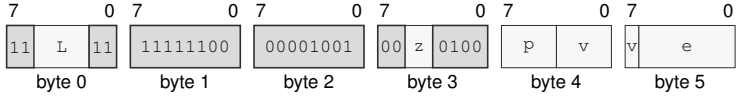
Instruction Format:



Format: Instruction format for VPOPCNT.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VPOPCNT.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VPOPCNT.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VREVBYTE — Vector Byte-Order Reverse — VREVBYTE

Operation: Reverses the byte order within each active element.

Assembler Syntax:

```
VREVBYTE.{W|L|Q}(z) Pn(p), Vn(v)
VREVBYTE.{W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VREVBYTE.{W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

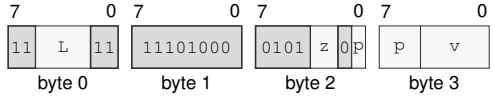
The register form reads and overwrites each active destination lane. The memory-source form transforms each active memory element and writes it to the corresponding destination lane. An inactive destination lane retains its previous value.

The memory-destination form transforms each active source-register lane and writes it to the corresponding memory element. An inactive memory element remains unchanged. The low-order byte becomes the high-order byte, the next pair exchanges places, and so on; each byte retains its internal bit order.

Encodings:

`VREVBYTE.{W|L|Q}(z) Pn(p), Vn(v)`

Instruction Format:



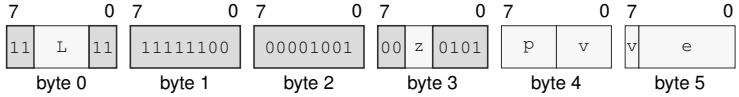
Format: Instruction format for VREVBYTE.{W|L|Q}(z) Pn(p), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VREVBYTE.{W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



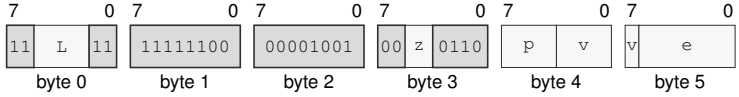
Format: Instruction format for VREVBYTE.{W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VREVBYTE.{W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VREVBYTE.{W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

PPERM — Predicate Permute — PPERM

Operation: Select predicate lanes using unsigned vector indices.

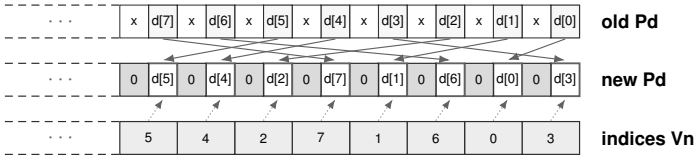
Assembler Syntax: `PPERM.{B|W|L|Q}(z) Vn(v), Pn(p)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

PPERM reads an unsigned index from each selected B, W, L, or Q vector lane. Each index selects the old destination's significant predicate position whose typed-lane index has that value. An index outside the architectural typed-lane range produces zero. The complete destination predicate image is written, and every bit outside its significant positions is cleared. The operation leaves FLAGS unchanged.

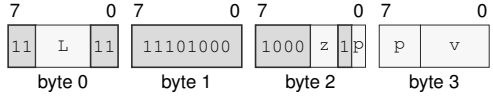


PPERM.W indexed predicate selection

Encodings:

`PPERM.{B|W|L|Q}(z) Vn(v), Pn(p)`

Instruction Format:



Format: Instruction format for PPERM.{B|W|L|Q}(z) Vn(v), Pn(p)

PSLIDEUP Predicate Slide Up PSLIDEUP

Operation: Move predicate lanes toward higher indices and insert zero at the low end.

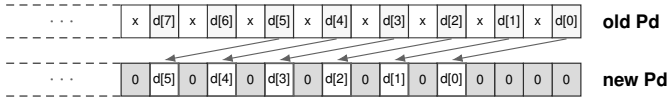
Assembler Syntax: `PSLIDEUP.{B|W|L|Q}(z) imm6(i), Pn(p)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

PSLIDEUP moves the significant predicate positions for the selected B, W, L, or Q width toward higher typed-lane indices by the unsigned immediate count. Vacated positions at the low end become zero. A count at least as large as the architectural typed-lane count clears every significant position. The complete destination predicate image is written, and every bit outside its significant positions is cleared. The operation leaves FLAGS unchanged.

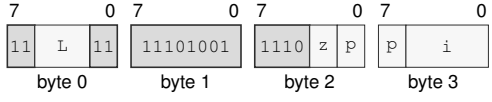


PSLIDEUP.W slide up by two predicate lanes

Encodings:

`PSLIDEUP.{B|W|L|Q}(z) imm6(i), Pn(p)`

Instruction Format:



Format: Instruction format for PSLIDEUP.{B|W|L|Q}(z) imm6(i), Pn(p)

PSLIDEDN — Predicate Slide Down — PSLIDEDN

Operation: Move predicate lanes toward lower indices and insert zero at the high end.

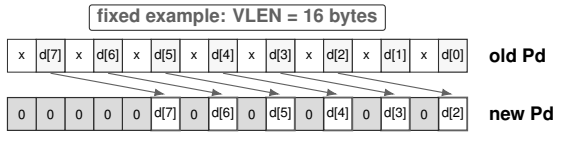
Assembler Syntax: `PSLIDEDN.{B|W|L|Q}(z) imm6(i), Pn(p)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

PSLIDEDN moves the significant predicate positions for the selected B, W, L, or Q width toward lower typed-lane indices by the unsigned immediate count. Vacated positions at the high end become zero. A count at least as large as the architectural typed-lane count clears every significant position. The complete destination predicate image is written, and every bit outside its significant positions is cleared. The operation leaves FLAGS unchanged.

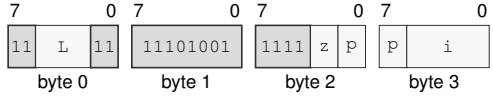


PSLIDEDN.W slide down by two predicate lanes

Encodings:

`PSLIDEDN.{B|W|L|Q}(z) imm6(i), Pn(p)`

Instruction Format:



Format: Instruction format for PSLIDEDN.{B|W|L|Q}(z) imm6(i), Pn(p)

VCMPcc — Vector Conditional Compare — VCMPcc

Operation: Compares active integer lanes and writes a complete predicate image.

Assembler Syntax:
```
VCMPcc.{B|W|L|Q}(x) Pn(p), Vn(v), Vn(w), Pn(q)
VCMPcc.{B|W|L|Q}(x) Pn(p), Vn(v), <ea>(e), Pn(q)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

VCMPcc compares corresponding active elements under the encoded condition. The register form reads both elements from vector registers. The memory form reads the right-hand element from the corresponding active VEA location.

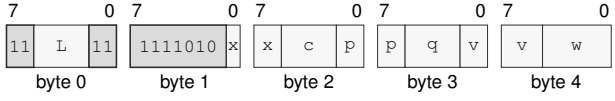
B, W, L, and Q forms apply the selected integer condition.

The destination is a complete predicate image. A significant result bit is set when its governing bit is set and the selected relation is true. Every other predicate bit is clear.

Encodings:

`VCMPcc.{B|W|L|Q}(x) Pn(p), Vn(v), Vn(w), Pn(q)`

Instruction Format:



Format: Instruction format for VCMPcc.{B|W|L|Q}(x) Pn(p), Vn(v), Vn(w), Pn(q)

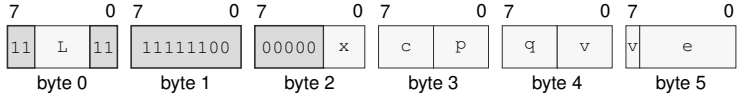
Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x3`; reason `reserved_vector_element_type`.

Operand constraint: Role `cc`; relation `allowed`; values `0x2..0x5, 0xa..0xf`; reason `reserved_vector_comparison_condition`.

`VCMPcc.{B|W|L|Q}(x) Pn(p), Vn(v), <ea>(e), Pn(q)`

Instruction Format:



Format: Instruction format for VCMPcc.{B|W|L|Q}(x) Pn(p), Vn(v), <ea>(e), Pn(q)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x3`; reason `reserved_vector_element_type`.

Operand constraint: Role `cc`; relation `allowed`; values `0x2..0x5, 0xa..0xf`; reason `reserved_vector_comparison_condition`.

VTESTZ — Vector Bitwise Test Zero — VTESTZ

Operation: Sets significant result bits where the governed element-wise AND is zero and clears all other bits.

Assembler Syntax:
```
VTESTZ.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w), Pn(q)
VTESTZ.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e), Pn(q)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

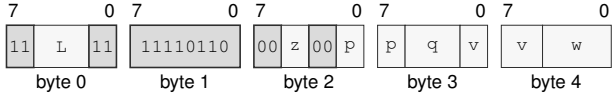
At the selected B, W, L, or Q width, the operation computes the bitwise AND of each left-hand Vn element and its corresponding right-hand element. The register form reads the right-hand element from the corresponding Vn lane. The memory form reads it from the corresponding element of the complete vector-length VEA source.

The complete result image replaces the destination predicate. A bit at an element's significant predicate position is set only when its governing bit is set and the element result is zero; every other predicate bit is clear.

Encodings:

`VTESTZ.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w), Pn(q)`

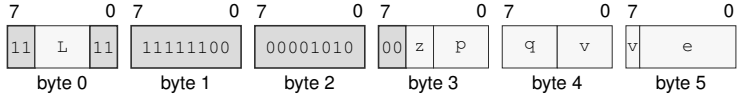
Instruction Format:



Format: Instruction format for VTESTZ.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w), Pn(q)

`VTESTZ.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e), Pn(q)`

Instruction Format:



Format: Instruction format for VTESTZ.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e), Pn(q)

VTESTNZ Vector Bitwise Test Nonzero VTESTNZ

Operation: Sets significant result bits where the governed element-wise AND is nonzero and clears all other bits.

Assembler Syntax:
```
VTESTNZ.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w), Pn(q)
VTESTNZ.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e), Pn(q)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

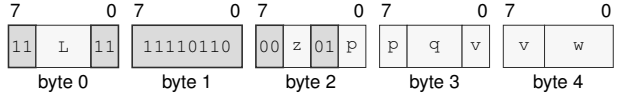
At the selected B, W, L, or Q width, the operation computes the bitwise AND of each left-hand Vn element and its corresponding right-hand element. The register form reads the right-hand element from the corresponding Vn lane. The memory form reads it from the corresponding element of the complete vector-length VEA source.

The complete result image replaces the destination predicate. A bit at an element's significant predicate position is set only when its governing bit is set and the element result is nonzero; every other predicate bit is clear.

Encodings:

`VTESTNZ.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w), Pn(q)`

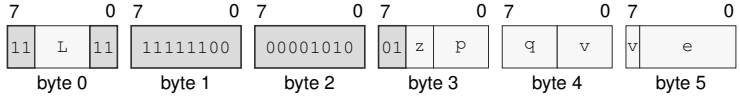
Instruction Format:



Format: Instruction format for VTESTNZ.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w), Pn(q)

`VTESTNZ.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e), Pn(q)`

Instruction Format:



Format: Instruction format for VTESTNZ.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e), Pn(q)

VADD Vector Add VADD

Operation: Adds corresponding active integer vector elements.

Assembler Syntax:
```
VADD.{B|W|L|Q}(x) Pn(p), Vn(v), Vn(w)
VADD.{B|W|L|Q}(x) Pn(p), <ea>(e), Vn(v)
VADD.{B|W|L|Q}(x) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

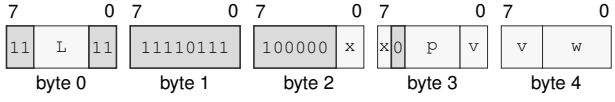
VADD applies selected integer addition independently to each active vector lane. The governing predicate selects active lanes; an inactive destination lane retains its previous value.

For B, W, L, and Q elements, each active result is the lane-width modular sum and no flag bank changes.

Encodings:

`VADD.{B|W|L|Q}(x) Pn(p), Vn(v), Vn(w)`

Instruction Format:



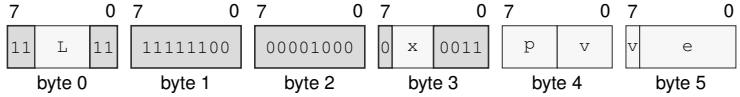
Format: Instruction format for VADD.{B|W|L|Q}(x) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x3`; reason `reserved_vector_element_type`.

`VADD.{B|W|L|Q}(x) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



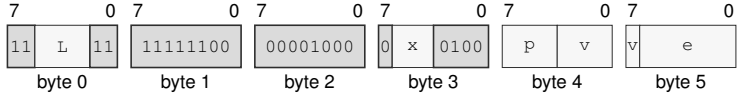
Format: Instruction format for VADD.{B|W|L|Q}(x) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x3`; reason `reserved_vector_element_type`.

`VADD.{B|W|L|Q}(x) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VADD.{B|W|L|Q}(x) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x3`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VSUB Vector Subtract VSUB

Operation: Subtracts corresponding active integer vector elements.

Assembler Syntax:
```
VSUB.{B|W|L|Q}(x) Pn(p), Vn(v), Vn(w)
VSUB.{B|W|L|Q}(x) Pn(p), <ea>(e), Vn(v)
VSUB.{B|W|L|Q}(x) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

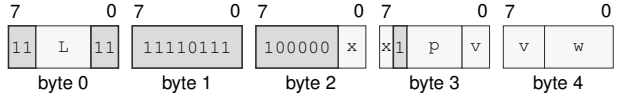
VSUB subtracts each corresponding active source element from its destination element. For B, W, L, and Q elements, the selected-width modular difference replaces the destination element. The register form and the memory-source form write the result to the destination vector lane. An inactive destination lane retains its previous value.

The memory-destination form reads each active destination element, subtracts the corresponding source-register element, and writes the result back to that memory element. Inactive memory elements retain their previous values, making the destination transaction a lane-wise read-modify-write.

Encodings:

`VSUB.{B|W|L|Q}(x) Pn(p), Vn(v), Vn(w)`

Instruction Format:



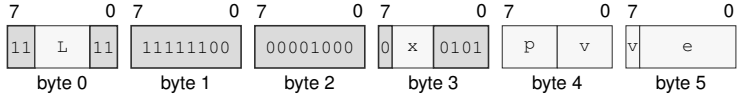
Format: Instruction format for VSUB.{B|W|L|Q}(x) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x3`; reason `reserved_vector_element_type`.

`VSUB.{B|W|L|Q}(x) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



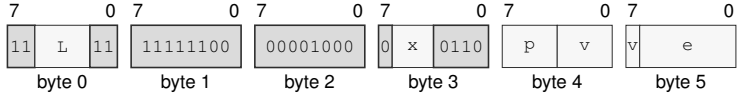
Format: Instruction format for VSUB.{B|W|L|Q}(x) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x3`; reason `reserved_vector_element_type`.

`VSUB.{B|W|L|Q}(x) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VSUB.{B|W|L|Q}(x) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x3`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VMUL Vector Multiply VMUL

Operation: Multiplies corresponding active integer elements.

Assembler Syntax:
```
VMUL.{B|W|L|Q}(x) Pn(p), Vn(v), Vn(w)
VMUL.{B|W|L|Q}(x) Pn(p), <ea>(e), Vn(v)
VMUL.{B|W|L|Q}(x) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

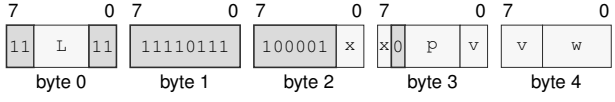
VMUL multiplies each active destination element by its corresponding source element. For B, W, L, and Q elements, the low selected-width product replaces the destination element. The register form and the memory-source form write the result to the destination vector lane. An inactive destination lane retains its previous value.

The memory-destination form reads each active destination element, multiplies it by the corresponding source-register element, and writes the result back to that memory element. Inactive memory elements retain their previous values, making the destination transaction a lane-wise read-modify-write.

Encodings:

`VMUL.{B|W|L|Q}(x) Pn(p), Vn(v), Vn(w)`

Instruction Format:



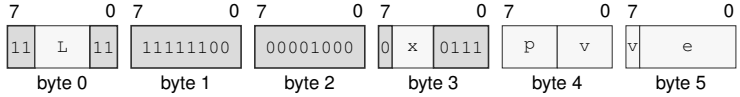
Format: Instruction format for VMUL.{B|W|L|Q}(x) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x3`; reason `reserved_vector_element_type`.

`VMUL.{B|W|L|Q}(x) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



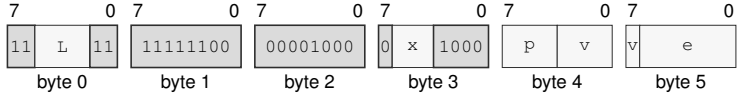
Format: Instruction format for VMUL.{B|W|L|Q}(x) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x3`; reason `reserved_vector_element_type`.

`VMUL.{B|W|L|Q}(x) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VMUL.{B|W|L|Q}(x) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x3`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VAND — Vector Bitwise AND — VAND

Operation: Applies bitwise AND to corresponding active vector elements.

Assembler Syntax:
```
VAND.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)
VAND.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VAND.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: VECTOR

Detailed Semantics

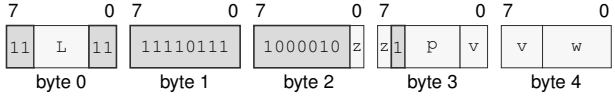
The register form computes the selected-width bitwise AND of each active source lane and corresponding old destination lane, then writes the result to the destination. When memory is the source, each active memory element is ANDed with the corresponding old destination lane. In both forms, an inactive destination lane retains its previous value.

The memory-destination form reads each active destination element, computes its selected-width bitwise AND with the corresponding source-register lane, and writes the result back to the same element. An inactive memory element remains unchanged.

Encodings:

`VAND.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

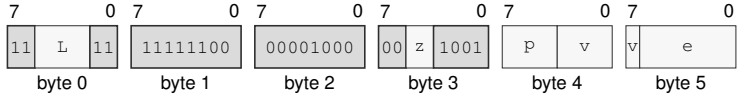
Instruction Format:



Format: Instruction format for VAND.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

`VAND.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

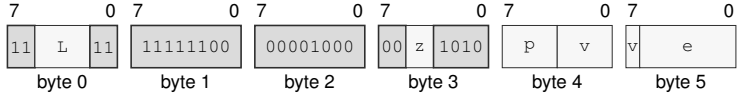
Instruction Format:



Format: Instruction format for VAND.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VAND.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VAND.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VOR — Vector Bitwise OR — VOR

Operation: Applies bitwise OR to corresponding active vector elements.

Assembler Syntax:
```
VOR.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)
VOR.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VOR.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

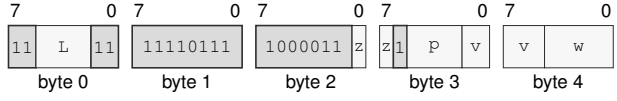
The register form computes the selected-width bitwise OR of each active source lane and corresponding old destination lane, then writes the result to the destination. When memory is the source, each active memory element is ORed with the corresponding old destination lane. In both forms, an inactive destination lane retains its previous value.

The memory-destination form reads each active destination element, computes its selected-width bitwise OR with the corresponding source-register lane, and writes the result back to the same element. An inactive memory element remains unchanged.

Encodings:

`VOR.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

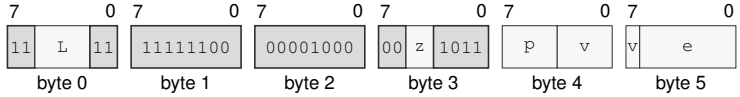
Instruction Format:



Format: Instruction format for VOR.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

`VOR.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

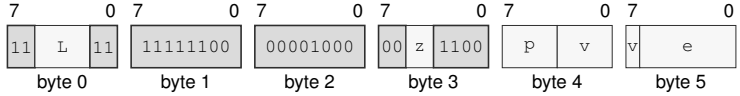
Instruction Format:



Format: Instruction format for VOR.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VOR.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VOR.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VXOR — Vector Bitwise XOR — VXOR

Operation: Applies bitwise XOR to corresponding active vector elements.

Assembler Syntax:
```
VXOR.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)
VXOR.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VXOR.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

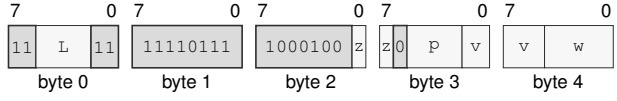
The register form computes the selected-width bitwise exclusive OR of each active source lane and corresponding old destination lane, then writes the result to the destination. When memory is the source, each active memory element is XORed with the corresponding old destination lane. In both forms, an inactive destination lane retains its previous value.

The memory-destination form reads each active destination element, computes its selected-width bitwise exclusive OR with the corresponding source-register lane, and writes the result back to the same element. An inactive memory element remains unchanged.

Encodings:

`VXOR.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

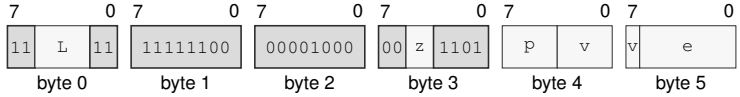
Instruction Format:



Format: Instruction format for VXOR.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

`VXOR.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

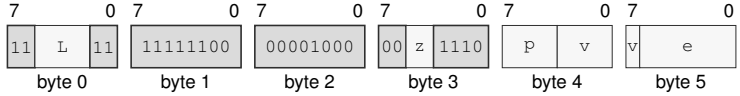
Instruction Format:



Format: Instruction format for VXOR.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VXOR.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VXOR.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VMINS — Vector Signed Minimum — VMINS

Operation: Selects the lesser signed value from each active destination/source element pair.

Assembler Syntax:
```
VMINS.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)
VMINS.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VMINS.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: VECTOR

Detailed Semantics

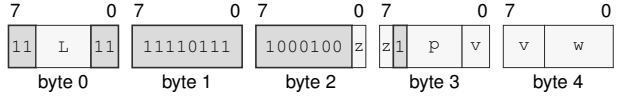
The governing predicate selects destination elements for update. For each selected element, the old destination and corresponding source are interpreted as two's-complement signed B, W, L, or Q values. The lesser value's bit pattern replaces the destination element. An unselected destination element retains its previous bit pattern.

The register form reads both operands from corresponding vector lanes. The memory-source form reads the source elements from memory and updates destination register lanes. The memory-destination form reads each selected destination element, compares it with the corresponding source-register lane, and writes the selected bit pattern back to that memory element.

Encodings:

`VMINS.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

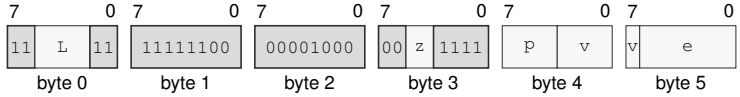
Instruction Format:



Format: Instruction format for VMINS.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

`VMINS.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

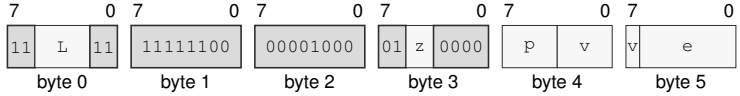
Instruction Format:



Format: Instruction format for VMINS.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VMINS.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VMINS.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VMINU — Vector Unsigned Minimum — VMINU

Operation: Selects the lesser unsigned value from each active destination/source element pair.

Assembler Syntax:
```
VMINU.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)
VMINU.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VMINU.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: VECTOR

Detailed Semantics

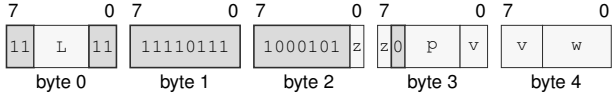
The governing predicate selects destination elements for update. For each selected element, the old destination and corresponding source are interpreted as unsigned B, W, L, or Q values. The lesser value's bit pattern replaces the destination element. An unselected destination element retains its previous bit pattern.

The register form reads both operands from corresponding vector lanes. The memory-source form reads the source elements from memory and updates destination register lanes. The memory-destination form reads each selected destination element, compares it with the corresponding source-register lane, and writes the selected bit pattern back to that memory element.

Encodings:

`VMINU.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

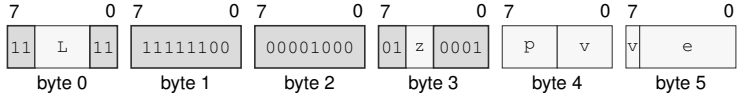
Instruction Format:



Format: Instruction format for VMINU.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

`VMINU.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

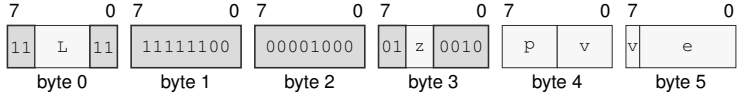
Instruction Format:



Format: Instruction format for VMINU.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VMINU.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VMINU.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VMAXS Vector Signed Maximum VMAXS

Operation: Selects the greater signed value from each active destination/source element pair.

Assembler Syntax:
```
VMAXS.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)
VMAXS.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VMAXS.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: VECTOR

Detailed Semantics

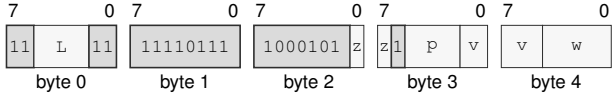
The governing predicate selects destination elements for update. For each selected element, the old destination and corresponding source are interpreted as two's-complement signed B, W, L, or Q values. The greater value's bit pattern replaces the destination element. An unselected destination element retains its previous bit pattern.

The register form reads both operands from corresponding vector lanes. The memory-source form reads the source elements from memory and updates destination register lanes. The memory-destination form reads each selected destination element, compares it with the corresponding source-register lane, and writes the selected bit pattern back to that memory element.

Encodings:

`VMAXS.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

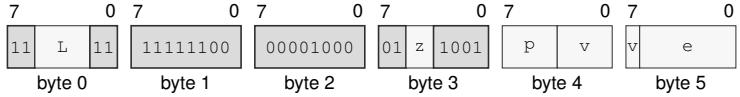
Instruction Format:



Format: Instruction format for VMAXS.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

`VMAXS.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

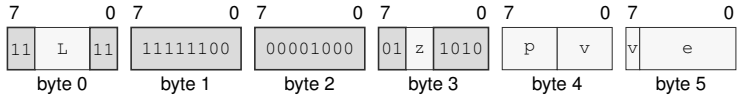
Instruction Format:



Format: Instruction format for VMAXS.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VMAXS.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VMAXS.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VMAXU — Vector Unsigned Maximum — VMAXU

Operation: Selects the greater unsigned value from each active destination/source element pair.

Assembler Syntax:
```
VMAXU.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)
VMAXU.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VMAXU.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: VECTOR

Detailed Semantics

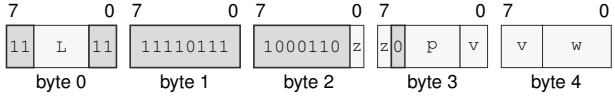
The governing predicate selects destination elements for update. For each selected element, the old destination and corresponding source are interpreted as unsigned B, W, L, or Q values. The greater value's bit pattern replaces the destination element. An unselected destination element retains its previous bit pattern.

The register form reads both operands from corresponding vector lanes. The memory-source form reads the source elements from memory and updates destination register lanes. The memory-destination form reads each selected destination element, compares it with the corresponding source-register lane, and writes the selected bit pattern back to that memory element.

Encodings:

`VMAXU.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

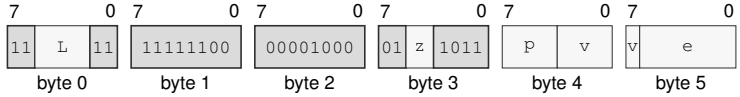
Instruction Format:



Format: Instruction format for VMAXU.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

`VMAXU.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

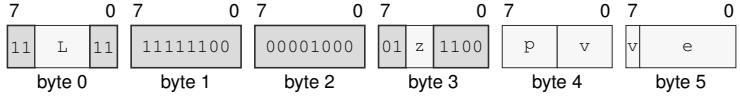
Instruction Format:



Format: Instruction format for VMAXU.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VMAXU.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VMAXU.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VMULHS Vector Signed Multiply High VMULHS

Operation: Writes the high half of each active signed destination/source product.

Assembler Syntax:
```
VMULHS.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)
VMULHS.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VMULHS.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

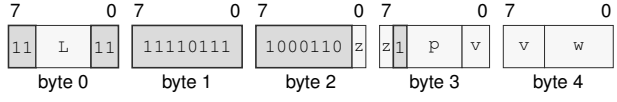
The governing predicate selects destination elements for update. The old destination and corresponding source are interpreted as two's-complement signed values at the selected B, W, L, or Q width. Their product is formed at twice that width, and its upper element-width half replaces the destination element. An unselected destination element retains its previous bit pattern.

The register form reads both operands from corresponding vector lanes. The memory-source form reads source elements from memory and updates destination register lanes. The memory-destination form treats each selected memory element as the old destination, multiplies it by the corresponding source-register lane, and writes the upper half back to that memory element.

Encodings:

`VMULHS.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

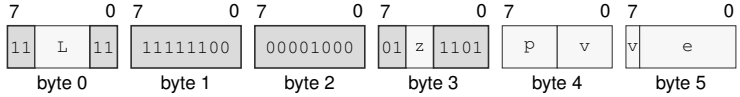
Instruction Format:



Format: Instruction format for VMULHS.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

`VMULHS.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

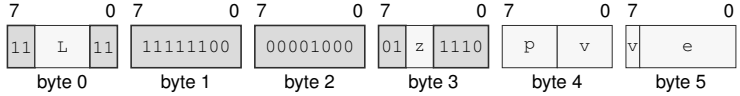
Instruction Format:



Format: Instruction format for VMULHS.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VMULHS.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VMULHS.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VMULHU Vector Unsigned Multiply High VMULHU

Operation: Writes the high half of each active unsigned destination/source product.

Assembler Syntax:
```
VMULHU.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)
VMULHU.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VMULHU.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: VECTOR

Detailed Semantics

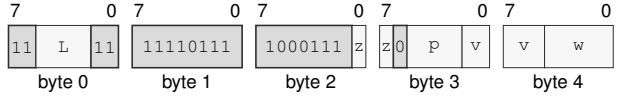
The governing predicate selects destination elements for update. The old destination and corresponding source are interpreted as unsigned values at the selected B, W, L, or Q width. Their product is formed at twice that width, and its upper element-width half replaces the destination element. An unselected destination element retains its previous bit pattern.

The register form reads both operands from corresponding vector lanes. The memory-source form reads source elements from memory and updates destination register lanes. The memory-destination form treats each selected memory element as the old destination, multiplies it by the corresponding source-register lane, and writes the upper half back to that memory element.

Encodings:

`VMULHU.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

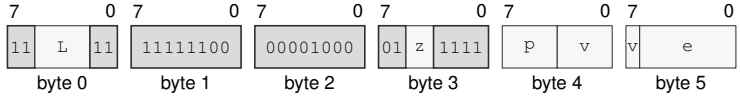
Instruction Format:



Format: Instruction format for VMULHU.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

`VMULHU.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

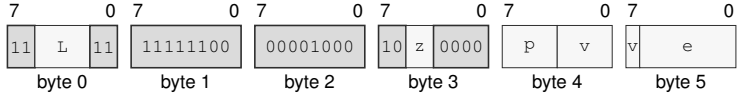
Instruction Format:



Format: Instruction format for VMULHU.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VMULHU.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VMULHU.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VMULHSU Vector Signed-by-Unsigned Multiply High VMULHSU

Operation: Writes the high half of each active signed-destination by unsigned-source product.

Assembler Syntax:
```
VMULHSU.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)
VMULHSU.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VMULHSU.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: VECTOR

Detailed Semantics

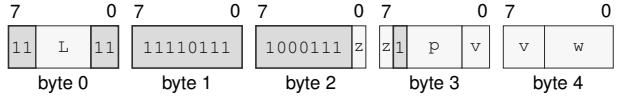
The governing predicate selects destination elements for update. At the selected B, W, L, or Q width, the old destination is interpreted as a two's-complement signed value and the corresponding source is interpreted as an unsigned value. Their product is formed at twice that width, and its upper element-width half replaces the destination element. An unselected destination element retains its previous bit pattern.

In the register form, the old destination-register lane is signed and the source-register lane is unsigned. In the memory-source form, the old destination-register lane is signed and the corresponding memory source is unsigned. In the memory-destination form, the old memory destination is signed and the corresponding source-register lane is unsigned; the upper half is written back to that memory element.

Encodings:

`VMULHSU.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

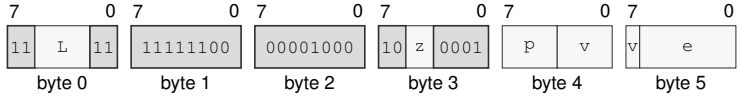
Instruction Format:



Format: Instruction format for VMULHSU.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

`VMULHSU.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

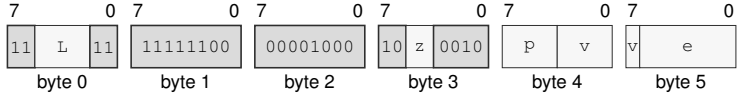
Instruction Format:



Format: Instruction format for VMULHSU.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VMULHSU.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VMULHSU.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VSHL — Vector Logical Shift Left — VSHL

Operation: Logically shifts each active integer element left by its selected-width count.

Assembler Syntax:

```
VSHL.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)
VSHL.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)
VSHL.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VSHL.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
VSHL.{B|W|L|Q}(z) Pn(p), imm6(i), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

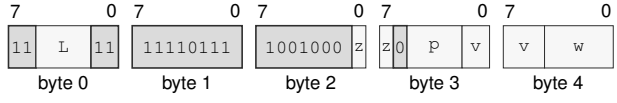
The governing predicate selects active destination elements. An immediate count applies to every active destination element. A vector or memory count source supplies the count from the lane corresponding to that element. VSHL takes the low six bits of each supplied count and reduces the value modulo the selected B, W, L, or Q element width in bits. The resulting count shifts the old destination element to the left and fills vacated low bits with zeros. An effective count of zero writes the original selected-width value for that active element.

With a vector-register destination, active lanes are updated and inactive lanes retain their previous values. A memory destination is updated as an encoded lane-wise read-modify-write: each active element is read, shifted, and written back, while inactive elements retain their previous values.

Encodings:

`VSHL.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

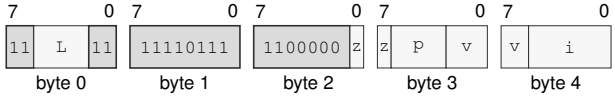
Instruction Format:



Format: Instruction format for VSHL.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

`VSHL.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)`

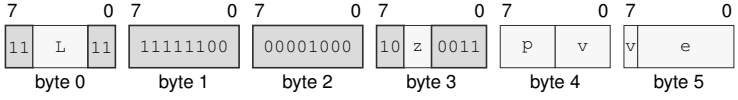
Instruction Format:



Format: Instruction format for VSHL.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)

`VSHL.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

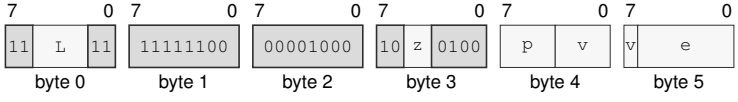
Instruction Format:



Format: Instruction format for VSHL.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VSHL.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



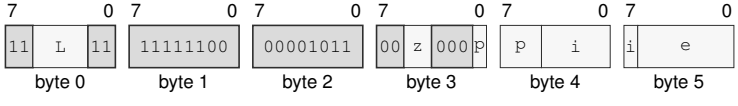
Format: Instruction format for VSHL.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`VSHL.{B|W|L|Q}(z) Pn(p), imm6(i), <ea>(e)`

Instruction Format:



Format: Instruction format for VSHL.{B|W|L|Q}(z) Pn(p), imm6(i), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VSHR — Vector Logical Shift Right — VSHR

Operation: Logically shifts each active integer element right by its selected-width count.

Assembler Syntax:

```
VSHR.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)
VSHR.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)
VSHR.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VSHR.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
VSHR.{B|W|L|Q}(z) Pn(p), imm6(i), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: VECTOR

Detailed Semantics

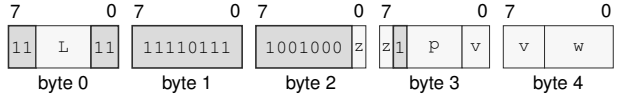
The governing predicate selects active destination elements. An immediate count applies to every active destination element. A vector or memory count source supplies the count from the lane corresponding to that element. VSHR takes the low six bits of each supplied count and reduces the value modulo the selected B, W, L, or Q element width in bits. The resulting count shifts the old destination element to the right and fills vacated high bits with zeros. An effective count of zero writes the original selected-width value for that active element.

With a vector-register destination, active lanes are updated and inactive lanes retain their previous values. A memory destination is updated as an encoded lane-wise read-modify-write: each active element is read, shifted, and written back, while inactive elements retain their previous values.

Encodings:

`VSHR.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

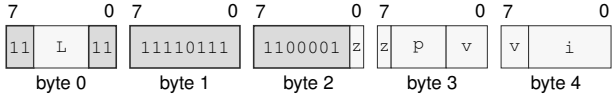
Instruction Format:



Format: Instruction format for VSHR.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

`VSHR.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)`

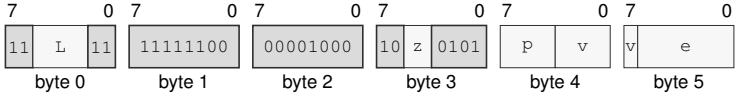
Instruction Format:



Format: Instruction format for VSHR.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)

`VSHR.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

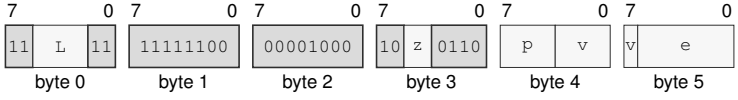
Instruction Format:



Format: Instruction format for VSHR.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VSHR.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



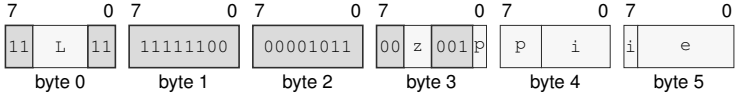
Format: Instruction format for VSHR.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`VSHR.{B|W|L|Q}(z) Pn(p), imm6(i), <ea>(e)`

Instruction Format:



Format: Instruction format for VSHR.{B|W|L|Q}(z) Pn(p), imm6(i), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VSAR — Vector Arithmetic Shift Right — VSAR

Operation: Arithmetically shifts each active integer element right by its selected-width count.

Assembler Syntax:

```
VSAR.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)
VSAR.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)
VSAR.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VSAR.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
VSAR.{B|W|L|Q}(z) Pn(p), imm6(i), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: VECTOR

Detailed Semantics

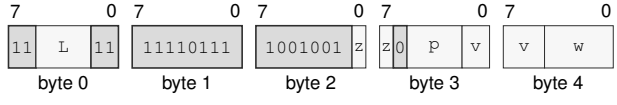
The governing predicate selects active destination elements. An immediate count applies to every active destination element. A vector or memory count source supplies the count from the lane corresponding to that element. VSAR takes the low six bits of each supplied count and reduces the value modulo the selected B, W, L, or Q element width in bits. The resulting count shifts the old destination element to the right and replicates its sign bit into vacated high positions. An effective count of zero writes the original selected-width value for that active element.

With a vector-register destination, active lanes are updated and inactive lanes retain their previous values. A memory destination is updated as an encoded lane-wise read-modify-write: each active element is read, shifted, and written back, while inactive elements retain their previous values.

Encodings:

`VSAR.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

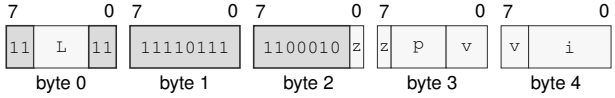
Instruction Format:



Format: Instruction format for VSAR.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

`VSAR.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)`

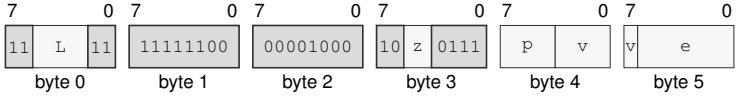
Instruction Format:



Format: Instruction format for VSAR.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)

`VSAR.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

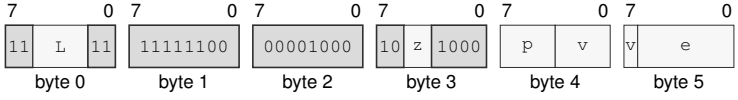
Instruction Format:



Format: Instruction format for VSAR.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VSAR.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



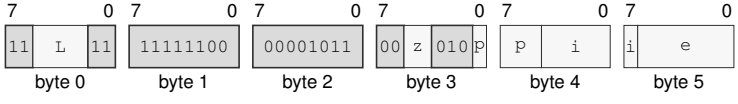
Format: Instruction format for VSAR.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`VSAR.{B|W|L|Q}(z) Pn(p), imm6(i), <ea>(e)`

Instruction Format:



Format: Instruction format for VSAR.{B|W|L|Q}(z) Pn(p), imm6(i), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VROL — Vector Rotate Left — VROL

Operation: Rotates each active integer element left by its selected-width count.

Assembler Syntax:

```
VROL.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)
VROL.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)
VROL.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VROL.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
VROL.{B|W|L|Q}(z) Pn(p), imm6(i), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: VECTOR

Detailed Semantics

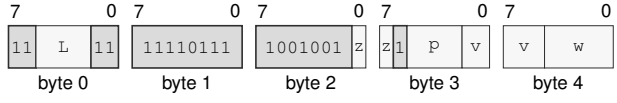
The governing predicate selects active destination elements. An immediate count applies to every active destination element. A vector or memory count source supplies the count from the lane corresponding to that element. VROL takes the low six bits of each supplied count and reduces the value modulo the selected B, W, L, or Q element width in bits. The resulting count rotates the old destination element to the left, wrapping bits shifted out of the high end into the low end of the same element. An effective count of zero writes the original selected-width value for that active element.

With a vector-register destination, active lanes are updated and inactive lanes retain their previous values. A memory destination is updated as an encoded lane-wise read-modify-write: each active element is read, rotated, and written back, while inactive elements retain their previous values.

Encodings:

`VROL.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

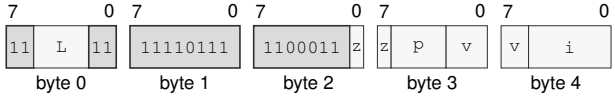
Instruction Format:



Format: Instruction format for VROL.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

`VROL.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)`

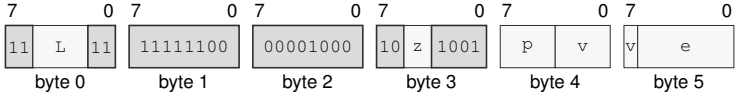
Instruction Format:



Format: Instruction format for VROL.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)

`VROL.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

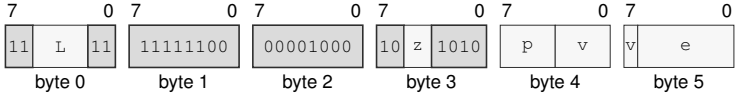
Instruction Format:



Format: Instruction format for VROL.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VROL.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



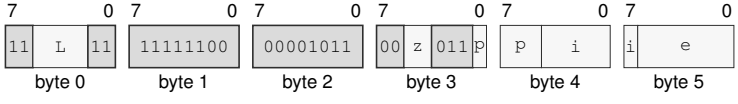
Format: Instruction format for VROL.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`VROL.{B|W|L|Q}(z) Pn(p), imm6(i), <ea>(e)`

Instruction Format:



Format: Instruction format for VROL.{B|W|L|Q}(z) Pn(p), imm6(i), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VROR — Vector Rotate Right — VROR

Operation: Rotates each active integer element right by its selected-width count.

Assembler Syntax:

```
VROR.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)
VROR.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)
VROR.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)
VROR.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)
VROR.{B|W|L|Q}(z) Pn(p), imm6(i), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: VECTOR

Detailed Semantics

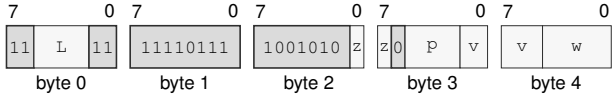
The governing predicate selects active destination elements. An immediate count applies to every active destination element. A vector or memory count source supplies the count from the lane corresponding to that element. VROR takes the low six bits of each supplied count and reduces the value modulo the selected B, W, L, or Q element width in bits. The resulting count rotates the old destination element to the right, wrapping bits shifted out of the low end into the high end of the same element. An effective count of zero writes the original selected-width value for that active element.

With a vector-register destination, active lanes are updated and inactive lanes retain their previous values. A memory destination is updated as an encoded lane-wise read-modify-write: each active element is read, rotated, and written back, while inactive elements retain their previous values.

Encodings:

`VROR.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

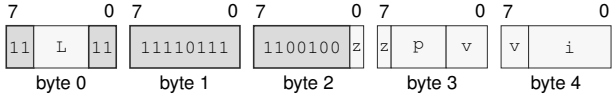
Instruction Format:



Format: Instruction format for VROR.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

`VROR.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)`

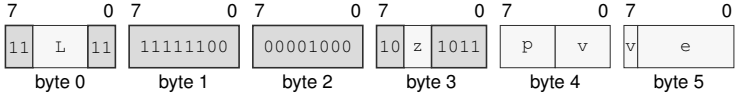
Instruction Format:



Format: Instruction format for VROR.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)

`VROR.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

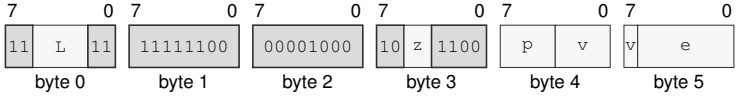
Instruction Format:



Format: Instruction format for VROR.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

`VROR.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



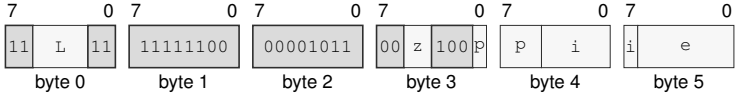
Format: Instruction format for VROR.{B|W|L|Q}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

`VROR.{B|W|L|Q}(z) Pn(p), imm6(i), <ea>(e)`

Instruction Format:



Format: Instruction format for VROR.{B|W|L|Q}(z) Pn(p), imm6(i), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VEXTZW Vector Zero Extension to Word VEXTZW

Operation: Zero-extends B source elements to W destination elements.

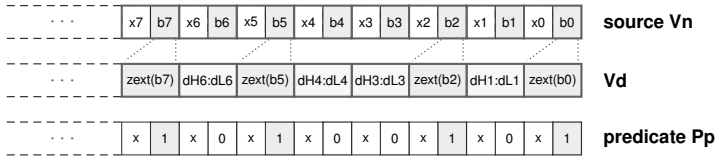
Assembler Syntax: `VEXTZW.{B}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

VEXTZW zero-extends the low B source element in each W container across the destination W element. The governing predicate position associated with each W container controls that container. An inactive container retains its complete previous destination value. The operation leaves FLAGS unchanged.

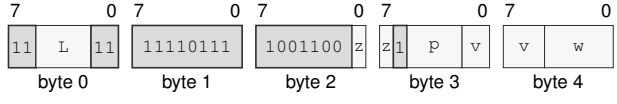


VEXTZW.B unsigned byte-to-word extension

Encodings:

`VEXTZW.{B}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VEXTZW.{B}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0`; reason `reserved_vector_element_type`.

VEXTSW — Vector Signed Extension to Word — VEXTSW

Operation: Sign-extends B source elements to W destination elements.

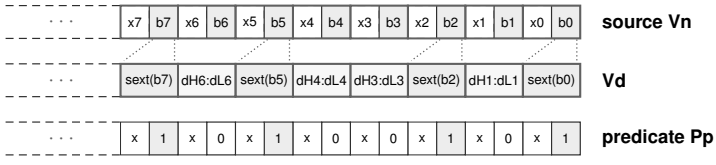
Assembler Syntax: `VEXTSW.{B}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

VEXTSW treats the low B source element in each W container as a signed two's-complement value and sign-extends it across the destination W element. The governing predicate position associated with each W container controls that container. An inactive container retains its complete previous destination value. The operation leaves FLAGS unchanged.

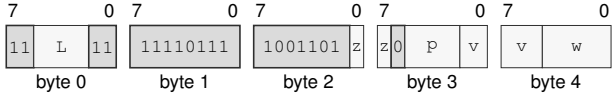


VEXTSW.B signed byte-to-word extension

Encodings:

`VEXTSW.{B}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VEXTSW.{B}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0`; reason `reserved_vector_element_type`.

VEXTZL — Vector Zero Extension to Long — VEXTZL

Operation: Zero-extends B or W source elements to L destination elements.

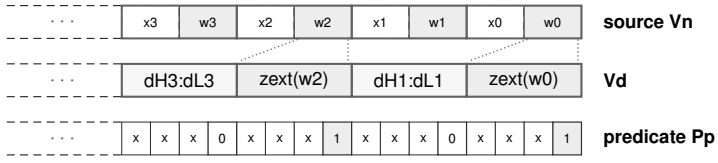
Assembler Syntax: `VEXTZL.{B|W}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

VEXTZL zero-extends the selected B or W source element into its L destination container. The governing predicate position associated with each L container controls that container. An inactive container retains its complete previous destination value. The operation leaves FLAGS unchanged.

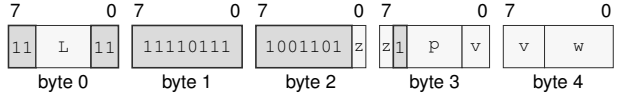


VEXTZL.W unsigned word-to-long extension

Encodings:

`VEXTZL.{B|W}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VEXTZL.{B|W}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x1`; reason `reserved_vector_element_type`.

VEXTSL Vector Signed Extension to Long VEXTSL

Operation: Sign-extends B or W source elements to L destination elements.

Assembler Syntax: `VEXTSL.{B|W}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

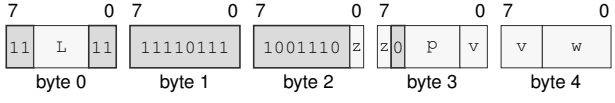
VEXTSL sign-extends the selected B or W source element into its L destination container for each active governing-predicate position. An inactive destination container retains its previous value. The operation leaves FLAGS unchanged.

VEXTSL.W signed word-to-long extension

Encodings:

`VEXTSL.{B|W}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VEXTSL.{B|W}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x1`; reason `reserved_vector_element_type`.

VEXTZQ — Vector Zero Extension to Quad — VEXTZQ

Operation: Zero-extends B, W, or L source elements to Q destination elements.

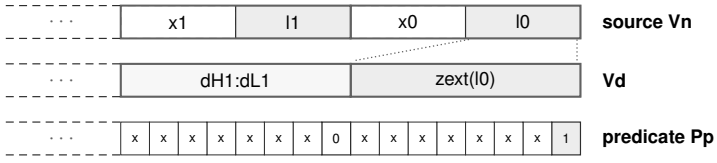
Assembler Syntax: `VEXTZQ.{B|W|L}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

VEXTZQ zero-extends the selected B, W, or L source element into its Q destination container. The governing predicate position associated with each Q container controls that container. An inactive container retains its complete previous destination value. The operation leaves FLAGS unchanged.

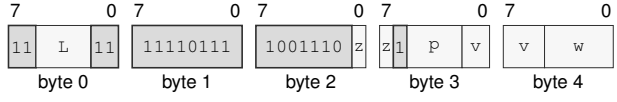


VEXTZQ.L unsigned long-to-quad extension

Encodings:

`VEXTZQ.{B|W|L}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VEXTZQ.{B|W|L}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x2`; reason `reserved_vector_element_type`.

VEXTSQ — Vector Signed Extension to Quad — VEXTSQ

Operation: Sign-extends B, W, or L source elements to Q destination elements.

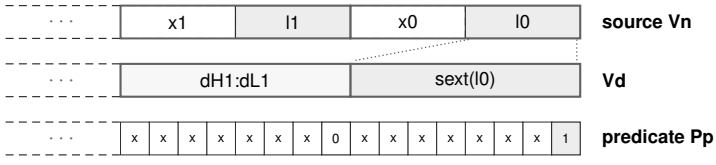
Assembler Syntax: `VEXTSQ.{B|W|L}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

VEXTSQ treats each selected B, W, or L source element as a signed two's-complement value and sign-extends it into the corresponding Q destination container. The governing predicate position associated with each Q container controls that container. An inactive container retains its complete previous destination value. The operation leaves FLAGS unchanged.

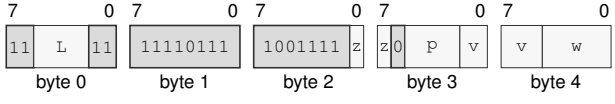


VEXTSQ.L signed long-to-quad extension

Encodings:

`VEXTSQ.{B|W|L}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VEXTSQ.{B|W|L}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x0..0x2`; reason `reserved_vector_element_type`.

VTRUNCB Vector Truncation to Byte VTRUNCB

Operation: Copies the low B field from each W, L, or Q source container.

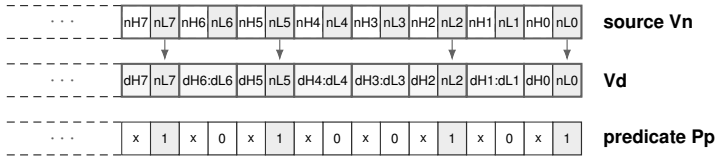
Assembler Syntax: `VTRUNCB.{W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

VTRUNCB copies the low B field of each selected W, L, or Q source container into the low B field of the corresponding destination container. For an active container, the destination bytes above that field retain their previous values. An inactive container retains its complete previous destination value. The operation leaves FLAGS unchanged.

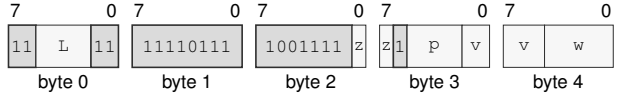


VTRUNCB.W low-byte truncation

Encodings:

`VTRUNCB.{W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VTRUNCB.{W|L|Q}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

VTRUNCW — Vector Truncation to Word — VTRUNCW

Operation: Copies the low W field from each L or Q source container.

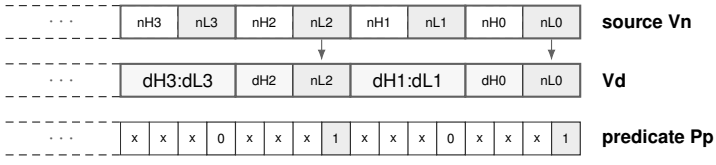
Assembler Syntax: `VTRUNCW.{L|Q}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

VTRUNCW copies the low W field of each selected L or Q source container into the low W field of the corresponding destination container. For an active container, the destination fields above that word retain their previous values. An inactive container retains its complete previous destination value. The operation leaves FLAGS unchanged.

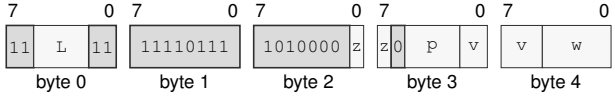


VTRUNCW.L low-word truncation

Encodings:

`VTRUNCW.{L|Q}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VTRUNCW.{L|Q}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x2..0x3`; reason `reserved_vector_element_type`.

VTRUNCL Vector Truncation to Long VTRUNCL

Operation: Copies the low L field from each Q source container.

Assembler Syntax: `VTRUNCL.{Q}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

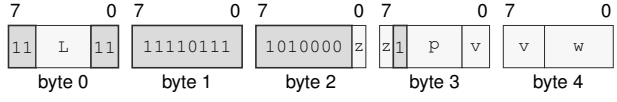
VTRUNCL copies the low L field of each Q source container into the low L field of the corresponding destination container. For an active container, the high L field of the destination retains its previous value. An inactive container retains its complete previous destination value. The operation leaves FLAGS unchanged.

VTRUNCL.Q low-longword truncation

Encodings:

`VTRUNCL.{Q}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VTRUNCL.{Q}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `3`; reason `reserved_vector_element_type`.

VPERM — Vector Permute — VPERM

Operation: Select source lanes with vector indices; out-of-range indices produce zero.

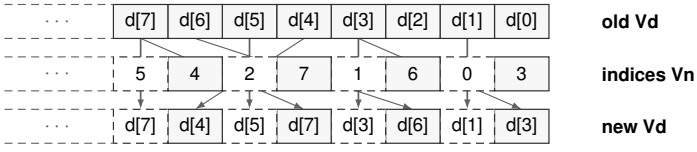
Assembler Syntax: `VPERM.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

VPERM reads an element index from each active source lane. An index within the selected vector lane count copies the correspondingly numbered old destination lane to that destination lane; an out-of-range index writes zero. The governing predicate retains the old destination value in each inactive lane.

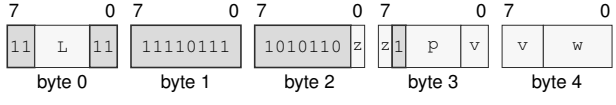


VPERM.W indexed lane selection

Encodings:

`VPERM.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VPERM.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

VZIPLO — Vector Lower-Half Interleave — VZIPLO

Operation: Interleave the lower halves of the old destination and source.

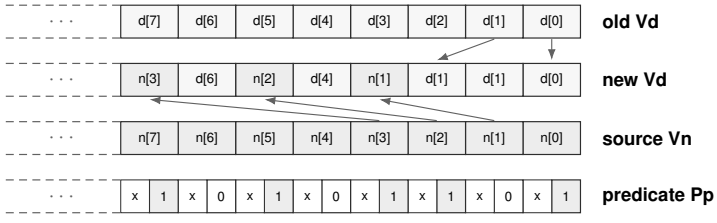
Assembler Syntax: `VZIPLO.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

VZIPLO forms each adjacent result-lane pair from one lane in the lower half of each input. In increasing order through those input halves, the even result lane receives the old destination value and the odd result lane receives the corresponding Vn value. The governing predicate retains the corresponding old destination value in each inactive result lane.

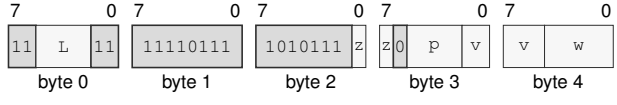


VZIPLO.W lower-half interleave

Encodings:

`VZIPLO.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VZIPLO.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

VZIPHI — Vector Upper-Half Interleave — VZIPHI

Operation: Interleave the upper halves of the old destination and source.

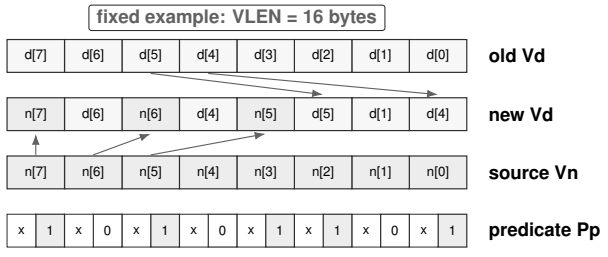
Assembler Syntax: `VZIPHI.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

VZIPHI forms each adjacent result-lane pair from one lane in the upper half of each input. In increasing order through those input halves, the even result lane receives the old destination value and the odd result lane receives the corresponding Vn value. The governing predicate retains the corresponding old destination value in each inactive result lane.

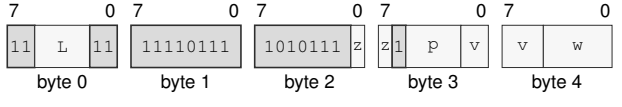


VZIPHI.W upper-half interleave

Encodings:

`VZIPHI.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VZIPHI.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

VUZIPLO Vector Even-Lane Deinterleave VUZIPLO

Operation: Gather even lanes from the old destination and source into separate result halves.

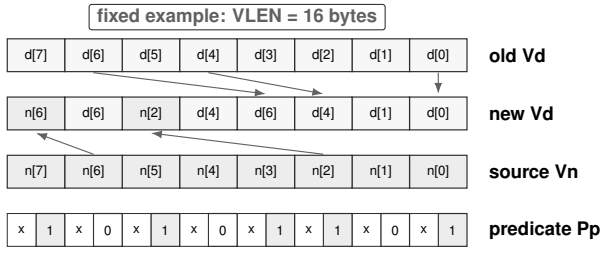
Assembler Syntax: `VUZIPLO.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

VUZIPLO gathers the even-numbered lanes from the old destination into the lower half of the result, in increasing lane order. It gathers the even-numbered Vn lanes into the upper half in the same order. The governing predicate retains the corresponding old destination value in each inactive result lane.

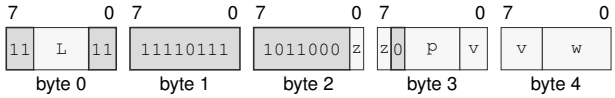


VUZIPLO.W even-lane extraction

Encodings:

```
VUZIPLO.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)
```

Instruction Format:



Format: Instruction format for VUZIPLO.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

VUZIPHI — Vector Odd-Lane Deinterleave — VUZIPHI

Operation: Gather odd lanes from the old destination and source into separate result halves.

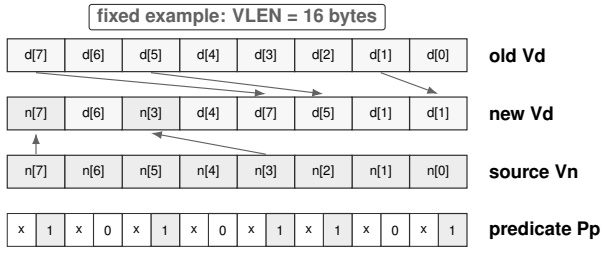
Assembler Syntax: `VUZIPHI.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

VUZIPHI gathers the odd-numbered lanes from the old destination into the lower half of the result, in increasing lane order. It gathers the odd-numbered Vn lanes into the upper half in the same order. The governing predicate retains the corresponding old destination value in each inactive result lane.

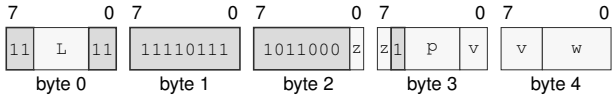


VUZIPHI.W odd-lane extraction

Encodings:

```
VUZIPHI.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)
```

Instruction Format:



Format: Instruction format for VUZIPHI.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

VTRNLO — Vector Even-Lane Transpose — VTRNLO

Operation: Interleave even lanes from the old destination and source by adjacent result pairs.

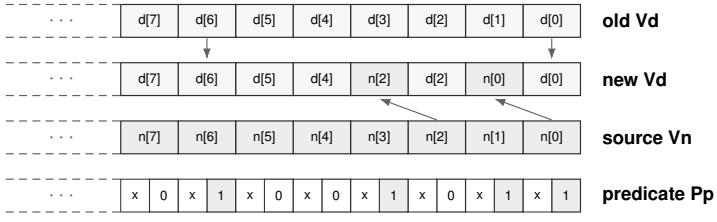
Assembler Syntax: `VTRNLO.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

For each adjacent result-lane pair, VTRNLO selects the even lane of the corresponding pair in both inputs. The even result lane receives that old destination value, and the odd result lane receives the value from Vn. The governing predicate retains the corresponding old destination value in each inactive result lane.

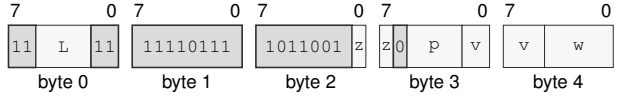


VTRNLO.W even-lane transpose

Encodings:

`VTRNLO.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VTRNLO.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

VTRNHI — Vector Odd-Lane Transpose — VTRNHI

Operation: Interleave odd lanes from the old destination and source by adjacent result pairs.

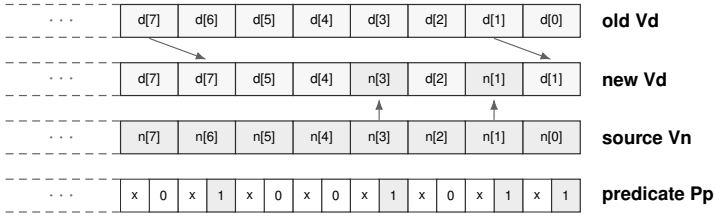
Assembler Syntax: `VTRNHI.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

For each adjacent result-lane pair, VTRNHI selects the odd lane of the corresponding pair in both inputs. The even result lane receives that old destination value, and the odd result lane receives the value from Vn. The governing predicate retains the corresponding old destination value in each inactive result lane.

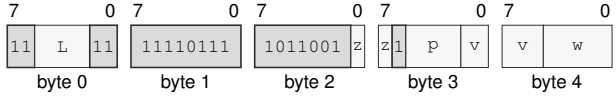


VTRNHI.W odd-lane transpose

Encodings:

`VTRNHI.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VTRNHI.{B|W|L|Q}(z) Pn(p), Vn(v), Vn(w)

VSLICE Vector Slice VSLICE

Operation: Extract a lane window from the old destination followed by a source vector.

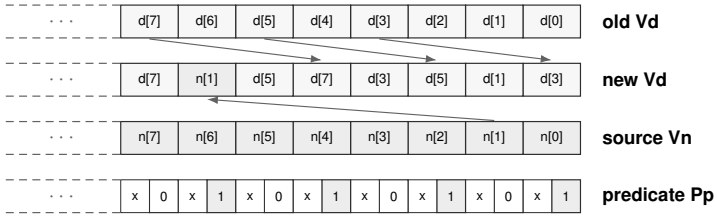
Assembler Syntax: `VSLICE.{B|W|L|Q}(z) Pn(p), Vn(v), imm6(i), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

Number the lanes of the old destination followed by the lanes of Vn as one sequence. For each active destination lane, VSLICE selects the sequence lane whose index is the destination-lane index plus the unsigned count. A sequence index beyond both vectors produces zero. The governing predicate retains the corresponding old destination value in each inactive lane.

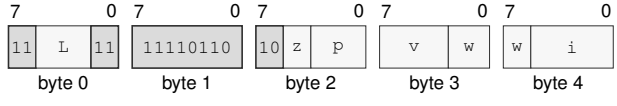


VSLICE.W slice by three lanes

Encodings:

`VSLICE.{B|W|L|Q}(z) Pn(p), Vn(v), imm6(i), Vn(w)`

Instruction Format:



Format: Instruction format for VSLICE.{B|W|L|Q}(z) Pn(p), Vn(v), imm6(i), Vn(w)

VSLIDEUP — Vector Slide Up — VSLIDEUP

Operation: Move old destination lanes toward higher indices and zero vacated lanes.

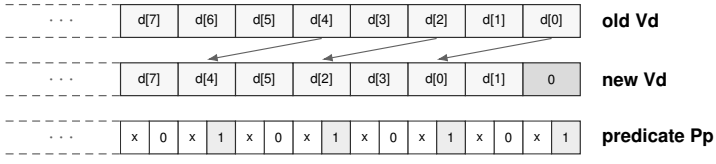
Assembler Syntax: `VSLIDEUP.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

For each active destination lane whose index is at least the unsigned count, VSLIDEUP reads the old destination lane at the index reduced by that count. It writes zero to an active lane with a lower index. The governing predicate retains the corresponding old destination value in each inactive lane.

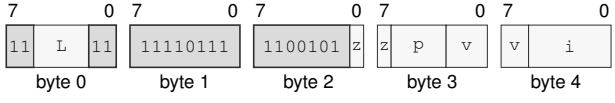


VSLIDEUP.W slide up by two lanes

Encodings:

`VSLIDEUP.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)`

Instruction Format:



Format: Instruction format for VSLIDEUP.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)

VSLIDEDN Vector Slide Down VSLIDEDN

Operation: Move old destination lanes toward lower indices and zero vacated lanes.

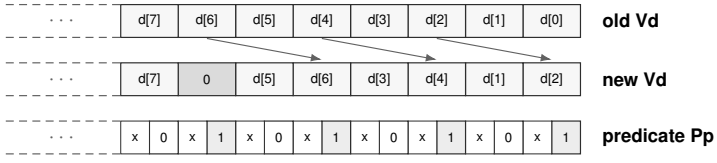
Assembler Syntax: `VSLIDEDN.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

For each active destination lane, VSLIDEDN reads the old destination lane whose index is the destination-lane index plus the unsigned count. It writes zero when that index reaches or exceeds the vector lane count. The governing predicate retains the corresponding old destination value in each inactive lane.

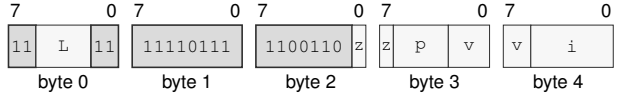


VSLIDEDN.W slide down by two lanes

Encodings:

`VSLIDEDN.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)`

Instruction Format:



Format: Instruction format for VSLIDEDN.{B|W|L|Q}(z) Pn(p), imm6(i), Vn(v)

VEXTRACT Vector Lane Extract VEXTRACT

Operation: Copies one indexed integer vector lane to an integer scalar register.

Assembler Syntax: `VEXTRACT.{B|W|L|Q}(z) Vn(v), Rn(r), Rn(s)`

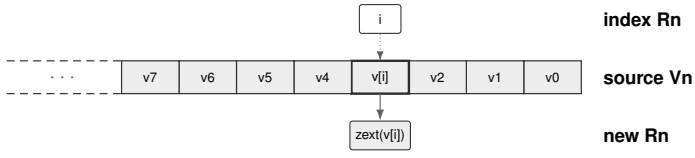
Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

VEXTRACT interprets the scalar index as unsigned and selects that lane from Vn. The B, W, L, and Q forms zero-extend the selected lane to the complete Rn result.

An index at or above the selected vector element count raises VECTOR_LANE_INDEX_OUT_OF_RANGE. The faulting instruction commits no scalar result.

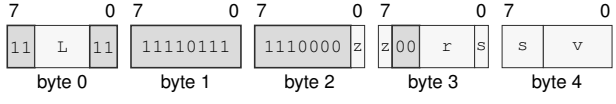


VEXTRACT.W indexed scalar result

Encodings:

`VEXTRACT.{B|W|L|Q}(z) Vn(v), Rn(r), Rn(s)`

Instruction Format:



Format: Instruction format for VEXTRACT.{B|W|L|Q}(z) Vn(v), Rn(r), Rn(s)

VINSERT — Vector Lane Insert — VINSERT

Operation: Replaces one indexed vector lane with an integer scalar value.

Assembler Syntax: `VINSERT.{B|W|L|Q}(z) Rn(r), Rn(s), Vn(v)`

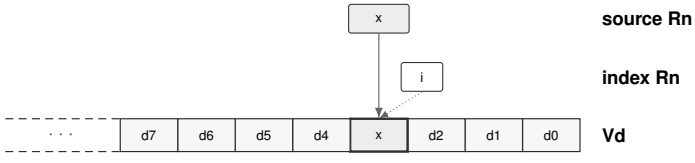
Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

VINSERT interprets the scalar index as unsigned and replaces only that lane of Vd. The B, W, L, and Q forms use the corresponding low field of Rn. Every lane at another index retains its previous value.

An index at or above the selected vector element count raises VECTOR_LANE_INDEX_OUT_OF_RANGE. The faulting instruction commits no destination change.

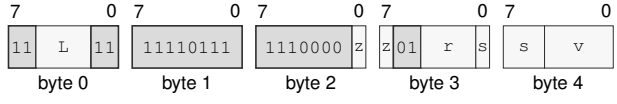


VINSERT.W indexed lane replacement

Encodings:

`VINSERT.{B|W|L|Q}(z) Rn(r), Rn(s), Vn(v)`

Instruction Format:



Format: Instruction format for VINSERT.{B|W|L|Q}(z) Rn(r), Rn(s), Vn(v)

VREDADD — Vector Add Reduction — VREDADD

Operation: Reduces active integer elements to one scalar sum.

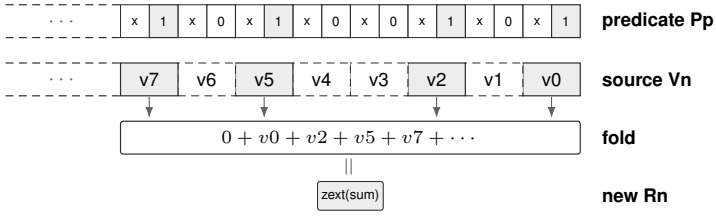
Assembler Syntax:
```
VREDADD.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)
VREDADD.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)
```

Privilege: Unprivileged

Required CPUID flags: VECTOR

Detailed Semantics

The governing predicate selects the source elements from Vn or the corresponding contiguous memory locations. The B, W, L, and Q forms add the selected elements modulo the element width and zero-extend the result to Rn. An empty integer selection produces zero.

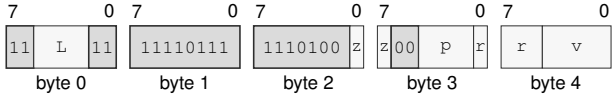


VREDADD.W integer reduction

Encodings:

VREDADD.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)

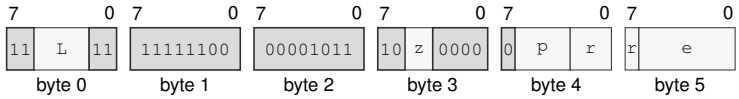
Instruction Format:



Format: Instruction format for VREDADD.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)

```
VREDADD.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)
```

Instruction Format:



Format: Instruction format for VREDADD.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)

VREDMINS Vector Signed Minimum Reduction VREDMINS

Operation: Finds the signed minimum of selected integer lanes and writes the scalar result to Rn.

Assembler Syntax:
```
VREDMINS.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)
VREDMINS.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

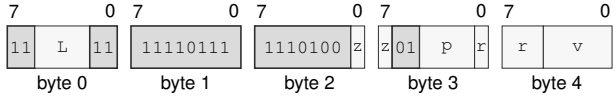
The governing predicate chooses which source elements participate. The Vn form reads those elements from the vector source. The VEA form first reads a complete vector-length source from memory and applies the same predicate selection.

Selected elements are compared as signed B, W, L, or Q values, and the least value becomes the reduction result. Its selected-width bit pattern occupies the low bits of Rn, and all higher bits are zero. If the predicate selects no elements, the result is the greatest signed value at the selected width.

Encodings:

`VREDMINS.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)`

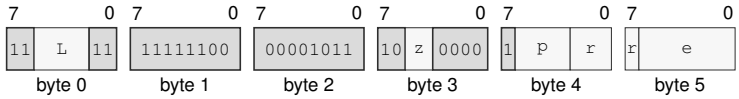
Instruction Format:



Format: Instruction format for VREDMINS.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)

```
VREDMINS.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)
```

Instruction Format:



Format: Instruction format for VREDMINS.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)

VREDMINU Vector Unsigned Minimum Reduction VREDMINU

Operation: Finds the unsigned minimum of selected integer lanes and writes the scalar result to Rn.

Assembler Syntax:
```
VREDMINU.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)
VREDMINU.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

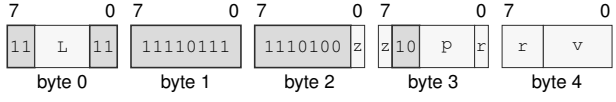
The governing predicate chooses which source elements participate. The Vn form reads those elements from the vector source. The VEA form first reads a complete vector-length source from memory and applies the same predicate selection.

Selected elements are compared as unsigned B, W, L, or Q values, and the least value becomes the reduction result. Its selected-width bit pattern occupies the low bits of Rn, and all higher bits are zero. If the predicate selects no elements, the low result bits are all ones at the selected width.

Encodings:

`VREDMINU.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)`

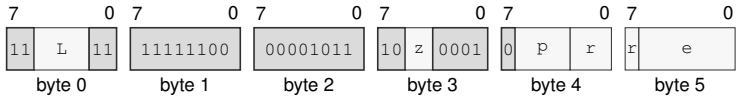
Instruction Format:



Format: Instruction format for VREDMINU.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)

```
VREDMINU.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)
```

Instruction Format:



Format: Instruction format for VREDMINU.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)

VREDMAXS Vector Signed Maximum Reduction VREDMAXS

Operation: Finds the signed maximum of selected integer lanes and writes the scalar result to Rn.

Assembler Syntax:
```
VREDMAXS.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)
VREDMAXS.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

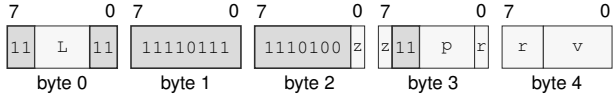
The governing predicate chooses which source elements participate. The Vn form reads those elements from the vector source. The VEA form first reads a complete vector-length source from memory and applies the same predicate selection.

Selected elements are compared as signed B, W, L, or Q values, and the greatest value becomes the reduction result. Its selected-width bit pattern occupies the low bits of Rn, and all higher bits are zero. If the predicate selects no elements, the result is the least signed value at the selected width.

Encodings:

`VREDMAXS.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)`

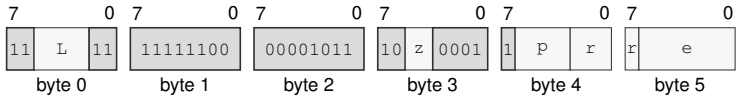
Instruction Format:



Format: Instruction format for VREDMAXS.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)

```
VREDMAXS.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)
```

Instruction Format:



Format: Instruction format for VREDMAXS.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)

VREDMAXU Vector Unsigned Maximum Reduction VREDMAXU

Operation: Finds the unsigned maximum of selected integer lanes and writes the scalar result to Rn.

Assembler Syntax:
```
VREDMAXU.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)
VREDMAXU.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

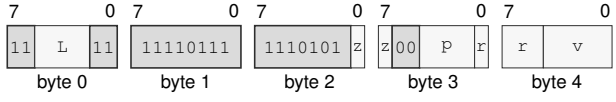
The governing predicate chooses which source elements participate. The Vn form reads those elements from the vector source. The VEA form first reads a complete vector-length source from memory and applies the same predicate selection.

Selected elements are compared as unsigned B, W, L, or Q values, and the greatest value becomes the reduction result. Its selected-width bit pattern occupies the low bits of Rn, and all higher bits are zero. If the predicate selects no elements, Rn receives zero.

Encodings:

`VREDMAXU.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)`

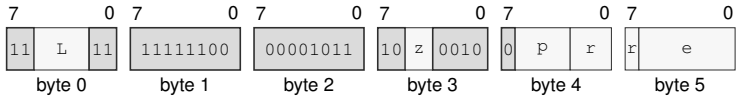
Instruction Format:



Format: Instruction format for VREDMAXU.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)

```
VREDMAXU.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)
```

Instruction Format:



Format: Instruction format for VREDMAXU.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)

VREDAND — Vector Bitwise AND Reduction — VREDAND

Operation: Reduces selected integer lanes with bitwise AND and writes the scalar result to Rn.

Assembler Syntax:

```
VREDAND.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)
VREDAND.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

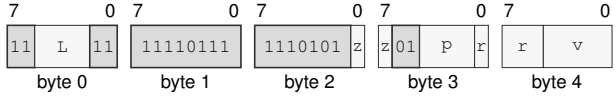
The governing predicate chooses which source elements participate. The Vn form reads those elements from the vector source. The VEA form first reads a complete vector-length source from memory and applies the same predicate selection.

Selected elements are combined with bitwise AND at the B, W, L, or Q element width. The selected-width result bit pattern occupies the low bits of Rn, and all higher bits are zero. If the predicate selects no elements, the low result bits are all ones at the selected width.

Encodings:

`VREDAND.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)`

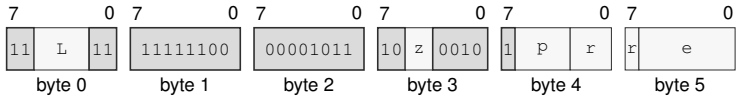
Instruction Format:



Format: Instruction format for VREDAND.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)

```
VREDAND.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)
```

Instruction Format:



Format: Instruction format for VREDAND.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)

VREDOR — Vector Bitwise OR Reduction — VREDOR

Operation: Reduces selected integer lanes with bitwise OR and writes the scalar result to Rn.

Assembler Syntax:
```
VREDOR.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)
VREDOR.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

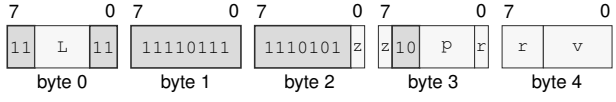
The governing predicate chooses which source elements participate. The Vn form reads those elements from the vector source. The VEA form first reads a complete vector-length source from memory and applies the same predicate selection.

Selected elements are combined with bitwise OR at the B, W, L, or Q element width. The selected-width result bit pattern occupies the low bits of Rn, and all higher bits are zero. If the predicate selects no elements, Rn receives zero.

Encodings:

`VREDOR.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)`

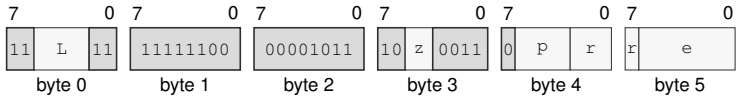
Instruction Format:



Format: Instruction format for VREDOR.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)

```
VREDOR.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)
```

Instruction Format:



Format: Instruction format for VREDOR.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)

VREDXOR — Vector Bitwise XOR Reduction — VREDXOR

Operation: Reduces selected integer lanes with bitwise XOR and writes the scalar result to Rn.

Assembler Syntax:

```
VREDXOR.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)
VREDXOR.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

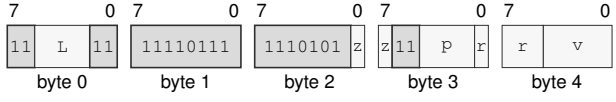
The governing predicate chooses which source elements participate. The Vn form reads those elements from the vector source. The VEA form first reads a complete vector-length source from memory and applies the same predicate selection.

Selected elements are combined with bitwise XOR at the B, W, L, or Q element width. The selected-width result bit pattern occupies the low bits of Rn, and all higher bits are zero. If the predicate selects no elements, Rn receives zero.

Encodings:

`VREDXOR.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)`

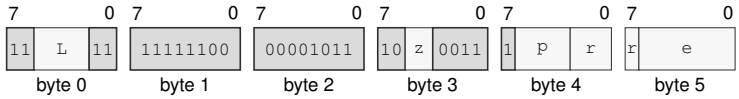
Instruction Format:



Format: Instruction format for VREDXOR.{B|W|L|Q}(z) Pn(p), Vn(v), Rn(r)

```
VREDXOR.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)
```

Instruction Format:



Format: Instruction format for VREDXOR.{B|W|L|Q}(z) Pn(p), <ea>(e), Rn(r)

PSEL — Predicate Select — PSEL

Operation: Replace bits of Pd selected by Pc with bits from Ps.

Assembler Syntax: `PSEL Pn(p), Pn(q), Pn(h)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

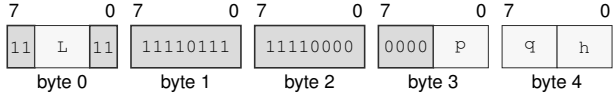
Detailed Semantics

Pc, Ps, and the previous value of Pd are read as complete packed predicate images. At each bit position, a set control bit writes the corresponding Ps bit to Pd, while a clear control bit preserves the previous Pd bit. Selection spans every bit of the packed image.

Encodings:

`PSEL Pn(p), Pn(q), Pn(h)`

Instruction Format:



Format: Instruction format for PSEL Pn(p), Pn(q), Pn(h)

PZIPLO Predicate Lower-Half Interleave PZIPLO

Operation: Interleave the lower halves of two predicate-lane sequences.

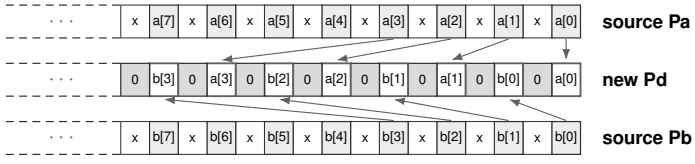
Assembler Syntax: `PZIPLO.{B|W|L|Q}(z) Pn(p), Pn(q), Pn(h)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

PZIPLO selects the lower half of the significant predicate lanes for the chosen B, W, L, or Q width from each source. In increasing order through those halves, each even result lane receives a left-source value and the following odd result lane receives the corresponding right-source value. The complete destination predicate image is written, and every bit outside its significant positions is cleared. The operation leaves FLAGS unchanged.

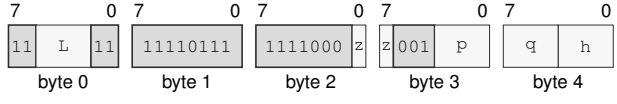


PZIPLO.W lower-half predicate interleave

Encodings:

`PZIPLO.{B|W|L|Q}(z) Pn(p), Pn(q), Pn(h)`

Instruction Format:



Format: Instruction format for PZIPLO.{B|W|L|Q}(z) Pn(p), Pn(q), Pn(h)

PZIPHI — Predicate Upper-Half Interleave — PZIPHI

Operation: Interleave the upper halves of two predicate-lane sequences.

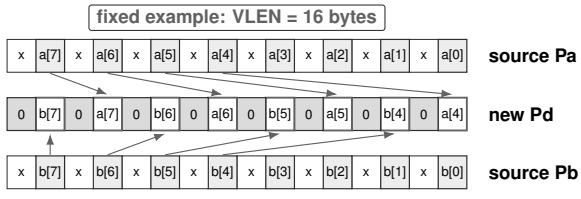
Assembler Syntax: `PZIPHI.{B|W|L|Q}(z) Pn(p), Pn(q), Pn(h)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

PZIPHI selects the upper half of the significant predicate lanes for the chosen B, W, L, or Q width from each source. In increasing order through those halves, each even result lane receives a left-source value and the following odd result lane receives the corresponding right-source value. The complete destination predicate image is written, and every bit outside its significant positions is cleared. The operation leaves FLAGS unchanged.

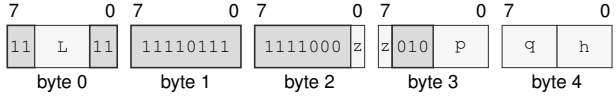


PZIPHI.W upper-half predicate interleave

Encodings:

`PZIPHI.{B|W|L|Q}(z) Pn(p), Pn(q), Pn(h)`

Instruction Format:



Format: Instruction format for PZIPHI.{B|W|L|Q}(z) Pn(p), Pn(q), Pn(h)

PUZIPLO Predicate Even-Lane Extraction PUZIPLO

Operation: Concatenate even-position predicate lanes from two sources.

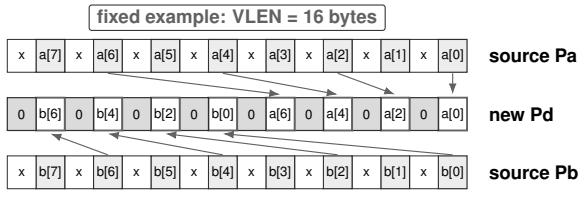
Assembler Syntax: `PUZIPLO.{B|W|L|Q}(z) Pn(p), Pn(q), Pn(h)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

PUZIPLO selects the even-position significant predicate lanes for the chosen B, W, L, or Q width. The selected lanes from the left source form the lower half of the result, and those from the right source form the upper half, with each half retaining source order. The complete destination predicate image is written, and every bit outside its significant positions is cleared. The operation leaves FLAGS unchanged.

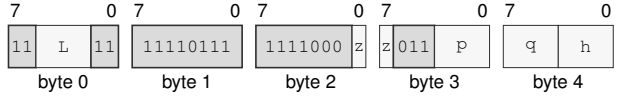


PUZIPLO.W even predicate-lane extraction

Encodings:

`PUZIPLO.{B|W|L|Q}(z) Pn(p), Pn(q), Pn(h)`

Instruction Format:



Format: Instruction format for PUZIPLO.{B|W|L|Q}(z) Pn(p), Pn(q), Pn(h)

PUZIPHI — Predicate Odd-Lane Extraction — PUZIPHI

Operation: Concatenate odd-position predicate lanes from two sources.

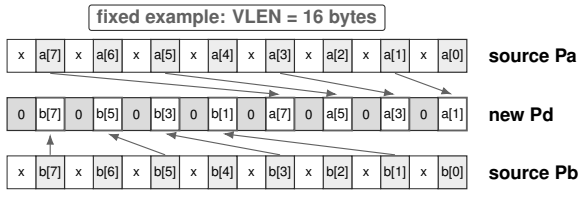
Assembler Syntax: `PUZIPHI.{B|W|L|Q}(z) Pn(p), Pn(q), Pn(h)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

PUZIPHI selects the odd-position significant predicate lanes for the chosen B, W, L, or Q width. The selected lanes from the left source form the lower half of the result, and those from the right source form the upper half, with each half retaining source order. The complete destination predicate image is written, and every bit outside its significant positions is cleared. The operation leaves FLAGS unchanged.

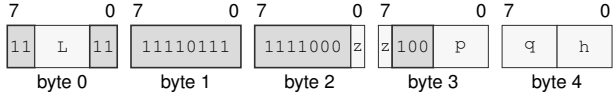


PUZIPHI.W odd predicate-lane extraction

Encodings:

`PUZIPHI.{B|W|L|Q}(z) Pn(p), Pn(q), Pn(h)`

Instruction Format:



Format: Instruction format for PUZIPHI.{B|W|L|Q}(z) Pn(p), Pn(q), Pn(h)

PTRNLO Predicate Even-Lane Transpose PTRNLO

Operation: Interleave even-position predicate lanes from two sources.

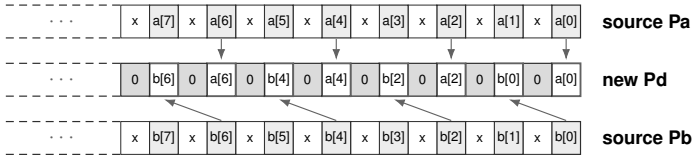
Assembler Syntax: `PTRNLO.{B|W|L|Q}(z) Pn(p), Pn(q), Pn(h)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

PTRNLO selects the even-position significant predicate lanes for the chosen B, W, L, or Q width from each source. For each selected source position in increasing order, the corresponding even result lane receives the left source value and the following odd result lane receives the right source value. The complete destination predicate image is written, and every bit outside its significant positions is cleared. The operation leaves FLAGS unchanged.

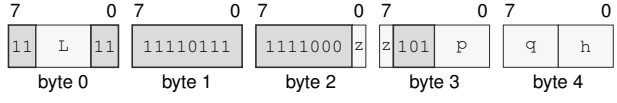


PTRNLO.W even predicate-lane transpose

Encodings:

`PTRNLO.{B|W|L|Q}(z) Pn(p), Pn(q), Pn(h)`

Instruction Format:



Format: Instruction format for PTRNLO.{B|W|L|Q}(z) Pn(p), Pn(q), Pn(h)

PTRNHI — Predicate Odd-Lane Transpose — PTRNHI

Operation: Interleave odd-position predicate lanes from two sources.

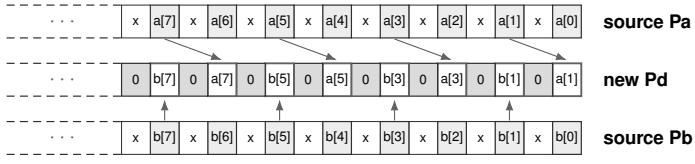
Assembler Syntax: `PTRNHI.{B|W|L|Q}(z) Pn(p), Pn(q), Pn(h)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

PTRNHI selects the odd-position significant predicate lanes for the chosen B, W, L, or Q width from each source. For each selected source position in increasing order, the corresponding even result lane receives the left source value and the following odd result lane receives the right source value. The complete destination predicate image is written, and every bit outside its significant positions is cleared. The operation leaves FLAGS unchanged.

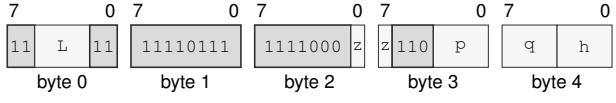


PTRNHI.W odd predicate-lane transpose

Encodings:

`PTRNHI.{B|W|L|Q}(z) Pn(p), Pn(q), Pn(h)`

Instruction Format:



Format: Instruction format for PTRNHI.{B|W|L|Q}(z) Pn(p), Pn(q), Pn(h)

PSLICE — Predicate Slice — PSLICE

Operation: Extract a predicate-lane window from the old destination followed by a source predicate.

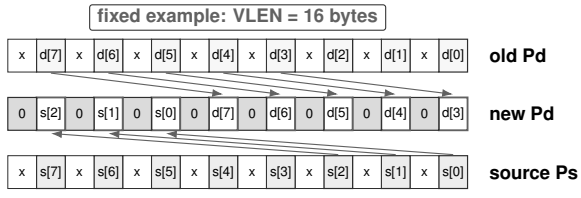
Assembler Syntax: `PSLICE.{B|W|L|Q}(z) Pn(p), imm6(i), Pn(q)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

PSLICE treats the significant predicate positions of the old destination followed by those of the source as one sequence of selected B, W, L, or Q typed lanes. It writes the destination from a window of the architectural typed-lane count beginning at the unsigned immediate count. Positions beyond that concatenated sequence become zero. The complete destination predicate image is written, and every bit outside its significant positions is cleared. The operation leaves FLAGS unchanged.

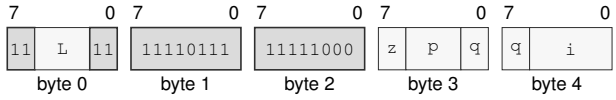


PSLICE.W slice by three predicate lanes

Encodings:

```
PSLICE.{B|W|L|Q}(z) Pn(p), imm6(i), Pn(q)
```

Instruction Format:



Format: Instruction format for PSLICE.{B|W|L|Q}(z) Pn(p), imm6(i), Pn(q)

VMOVZ — Vector Zeroing Move — VMOVZ

Operation: Loads active memory elements and clears inactive destination lanes.

Assembler Syntax: `VMOVZ.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

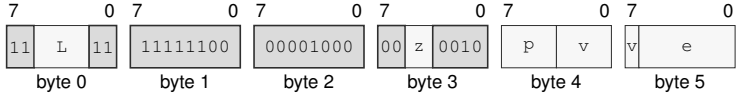
Detailed Semantics

The governing predicate selects the contiguous memory elements to read. Each active lane receives its corresponding memory element, and each inactive lane receives zero. The resulting elements replace the destination vector image at the selected width.

Encodings:

`VMOVZ.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



Format: Instruction format for VMOVZ.{B|W|L|Q}(z) Pn(p), <ea>(e), Vn(v)

PLOOP — Predicate Loop Chunk — PLOOP

Operation: Builds the next predicate chunk from a remaining count and lane offset.

Assembler Syntax: `PLOOP.{B|W|L|Q}(z) Rn(r), Rn(s), Pn(p), <ea>(e)`

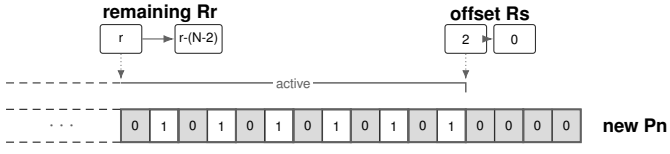
Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

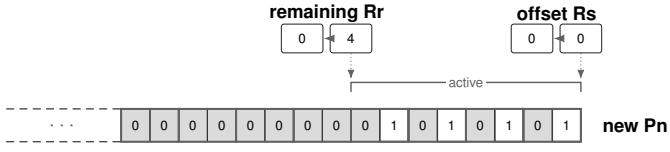
PLOOP reads an unsigned remaining count and an unsigned lane offset. Let N denote the number of lanes for the selected element width. If the remaining count is zero, PLOOP transfers control to the encoded target and leaves the predicate and both scalar registers unchanged.

For a nonzero remaining count, an offset at or above the selected element count raises VECTOR_LOOP_OFFSET_OUT_OF_RANGE and commits no state. A valid iteration sets the next $\min(\text{remaining},\ N - \text{offset})$ significant predicate positions, clears every other predicate bit, subtracts the number of set positions from the remaining count, clears the offset register, and falls through.

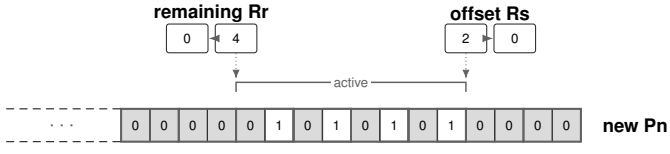


PLOOP.W head iteration

PLOOP.W body iteration



PLOOP.W tail iteration

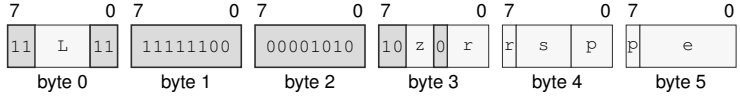


PLOOP.W head-tail iteration

Encodings:

```
PLOOP.{B|W|L|Q}(z) Rn(r), Rn(s), Pn(p), <ea>(e)
```

Instruction Format:



Format: Instruction format for PLOOP.{B|W|L|Q}(z) Rn(r), Rn(s), Pn(p), <ea>(e)

Restrictions:

Operand overlap: Operands `remaining`, `offset`; rule `illegal_instruction`.

PMOV — Predicate Memory Move — PMOV

Operation: Transfers a complete packed predicate image between a predicate register and memory.

Assembler Syntax:
```
PMOV <ea>(e), Pn(p)
PMOV Pn(p), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags: `VECTOR`

Detailed Semantics

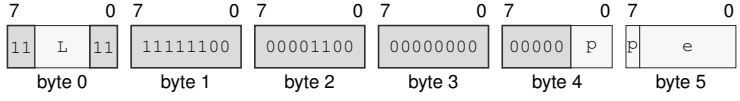
PMOV transfers the complete packed image of one predicate register between `Pn` and contiguous memory. The image contains VLEN/64 bytes. The memory-source form replaces every bit of the destination predicate image; the memory-destination form stores every bit of the source predicate image. No element predicate applies to either form.

The effective-address calculation, complete image transfer, and any effective-address auto-update commit as one vector memory operation. A fault commits none of those effects. The operation leaves FLAGS unchanged.

Encodings:

`PMOV <ea>(e), Pn(p)`

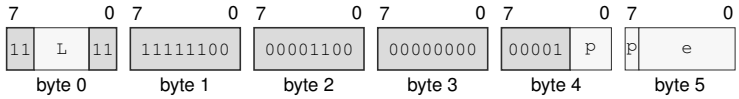
Instruction Format:



Format: Instruction format for PMOV <ea>(e), Pn(p)

`PMOV Pn(p), <ea>(e)`

Instruction Format:



Format: Instruction format for PMOV Pn(p), <ea>(e)

Restrictions:

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

24 SCALABLE-VECTOR FLOATING-POINT INSTRUCTIONS

24.1 Summary

Table 24-1. Scalable-Vector Floating-Point Instructions Summary (Informative)

Mnemonic	Brief description
VFDUP	Broadcasts a floating-point scalar register to every vector lane.
VFNEG	Toggles the sign of active floating-point elements.
VFABS	Clears the sign of active floating-point elements.
VFSQRT	Computes the square root of each active floating-point element.
VFROUND	Rounds each active floating-point element to an integral value in the same format using the current rounding mode.
VFTRUNC	Rounds each active floating-point element toward zero to an integral value in the same format.
VFFLOOR	Rounds each active floating-point element downward to an integral value in the same format.
VFCEIL	Rounds each active floating-point element upward to an integral value in the same format.
VFCLASS	Writes a floating-point classification bitmap for each active element.
VFCMPcc	Compares active floating-point lanes and writes a complete predicate image.
VFADD	Adds corresponding active floating-point vector elements.
VFSUB	Subtracts corresponding active floating-point vector elements.
VFMUL	Multiplies corresponding active floating-point elements.
VFMIN	Selects the floating-point minimum in each active lane.
VFMAX	Selects the floating-point maximum in each active lane.
VFDIV	Divides corresponding active floating-point elements.
VFCOPYSIGN	Combines each active destination magnitude with the corresponding source sign bit.
VFCVTS	Converts active half-precision, double-precision, or signed integer elements to single precision.
VFCVTD	Converts active half-precision, single-precision, or signed integer elements to double precision.
VFCVTUS	Converts active unsigned integer elements to single-precision floating point.
VFCVTUD	Converts active unsigned integer elements to double-precision floating point.
VFCVTL	Converts active floating-point elements to signed longwords.
VFCVTUL	Converts active floating-point elements to unsigned longwords.
VFCVTQ	Converts active floating-point elements to signed quadwords.
VFCVTUQ	Converts active floating-point elements to unsigned quadwords.
VFCVTH	Converts active single-precision, double-precision, or signed integer elements to half precision.
VFCVTUH	Converts active unsigned integer elements to half-precision floating point.
VFMADD	Performs fused multiply-add on corresponding active floating-point elements.
VFMSUB	Performs fused multiply-subtract on corresponding active floating-point elements.
VFNMADD	Performs fused negative multiply-add on corresponding active floating-point elements.
VFNMSUB	Performs fused negative multiply-subtract on corresponding active floating-point elements.
VFEXTRACT	Copies one indexed floating-point vector lane to a floating-point scalar register.
VFINSERT	Replaces one indexed vector lane with a floating-point scalar value.
VFREDADD	Reduces active floating-point elements to one scalar sum.

Table 24-1 (continued)

Mnemonic	Brief description
VFREDMIN	Reduces active floating-point elements to one minimum value.
VFREDMAX	Reduces active floating-point elements to one maximum value.

24.2 Vector Floating-Point Element Types and Semantics

VECTORFP requires VECTOR and FP. Its mnemonics use the VF prefix and their element suffixes H, S, and D select IEEE 754 binary16, binary32, and binary64 elements. Binary16 has default quiet NaN 0x7e00. The three-bit tzz selector values five through seven select H/S/D; value four remains invalid. In FP-only families, the two-bit zz selector reserves zero and assigns one through three to H/S/D.

Floating-Point Exceptions, Conversions, and Reductions. Each active FP lane applies the corresponding scalar rounding, NaN, signed-zero, default-NaN, denormal, and exception rules. Inactive lanes generate no cause. The instruction unions all active-lane causes before testing the enables in the original FSTATUS. If any generated cause is enabled, one FLOATING_POINT_EXCEPTION occurs and neither a destination effect nor newly accrued FFLAGS becomes visible. Otherwise the union is accrued once and all destination effects commit together. FP-to-integer conversion rounds toward zero and follows the scalar saturation and invalid result rules.

FP reductions return an Fn. Empty FP ADD returns positive zero, empty FP MIN returns positive infinity, and empty FP MAX returns negative infinity. FP ADD visits active lanes in increasing logical-lane order and may not be reassociated.

VFDUP — Vector Floating-Point Duplicate — VFDUP

Operation: Broadcasts a floating-point scalar register to every vector lane.

Assembler Syntax: `VFDUP.{H|S|D}(z) Fn(r), Vn(v)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`VECTOR`
`VECTORFP`

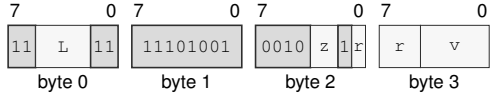
Detailed Semantics

VFDUP fills every selected-width H, S, or D destination lane with the selected floating-point value from Fn. The operation replaces the complete destination vector image and leaves FFLAGS unchanged.

Encodings:

`VFDUP.{H|S|D}(z) Fn(r), Vn(v)`

Instruction Format:



Format: Instruction format for VFDUP.{H|S|D}(z) Fn(r), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

VFNEG — Vector Floating-Point Negate — VFNEG

Operation: Toggles the sign of active floating-point elements.

Assembler Syntax:

```
VFNEG.{H|S|D}(x) Pn(p), Vn(v)
VFNEG.{H|S|D}(x) Pn(p), <ea>(e), Vn(v)
VFNEG.{H|S|D}(x) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags:

```
FP
VECTOR
VECTORFP
```

Detailed Semantics

For each active H, S, or D element, VFNEG toggles the sign bit and preserves the remaining bits.

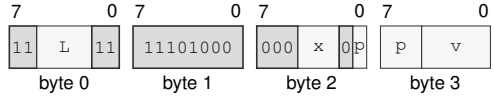
The register form reads and overwrites each active destination lane. The memory-source form reads each active memory element and writes the result to the corresponding destination lane. An inactive destination lane retains its previous value.

The memory-destination form transforms each active source-register element and writes the result to the corresponding memory element. Inactive memory elements retain their previous values, making the destination transaction a lane-wise read-modify-write.

Encodings:

`VFNEG.{H|S|D}(x) Pn(p), Vn(v)`

Instruction Format:



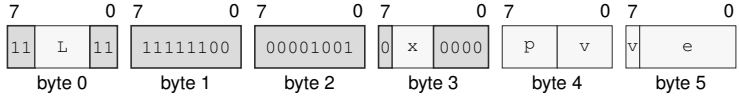
Format: Instruction format for VFNEG.{H|S|D}(x) Pn(p), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x5..0x7`; reason `reserved_vector_element_type`.

`VFNEG.{H|S|D}(x) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



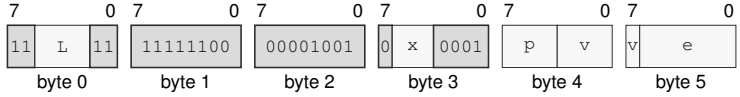
Format: Instruction format for VFNEG.{H|S|D}(x) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x5..0x7`; reason `reserved_vector_element_type`.

`VFNEG.{H|S|D}(x) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VFNEG.{H|S|D}(x) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x5..0x7`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VFABS — Vector Floating-Point Absolute Value — VFABS

Operation: Clears the sign of active floating-point elements.

Assembler Syntax:
```
VFABS.{H|S|D}(x) Pn(p), Vn(v)
VFABS.{H|S|D}(x) Pn(p), <ea>(e), Vn(v)
VFABS.{H|S|D}(x) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

Detailed Semantics

For each active H, S, or D element, VFABS clears the sign bit and preserves the remaining bits.

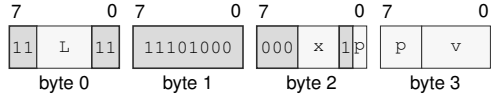
The register form reads and overwrites each active destination lane. The memory-source form reads each active memory element and writes the result to the corresponding destination lane. An inactive destination lane retains its previous value.

The memory-destination form transforms each active source-register element and writes the result to the corresponding memory element. Inactive memory elements retain their previous values, making the destination transaction a lane-wise read-modify-write.

Encodings:

`VFABS.{H|S|D}(x) Pn(p), Vn(v)`

Instruction Format:



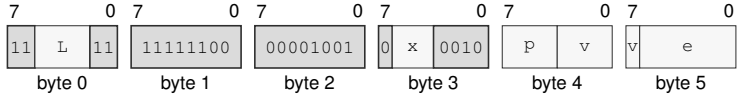
Format: Instruction format for VFABS.{H|S|D}(x) Pn(p), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x5..0x7`; reason `reserved_vector_element_type`.

`VFABS.{H|S|D}(x) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



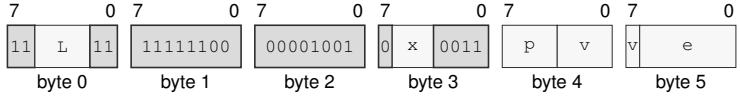
Format: Instruction format for VFABS.{H|S|D}(x) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x5..0x7`; reason `reserved_vector_element_type`.

`VFABS.{H|S|D}(x) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VFABS.{H|S|D}(x) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x5..0x7`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VFSQRT — Vector Floating-Point Square Root — VFSQRT

Operation: Computes the square root of each active floating-point element.

Assembler Syntax:
```
VFSQRT.{H|S|D}(z) Pn(p), Vn(v)
VFSQRT.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)
VFSQRT.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

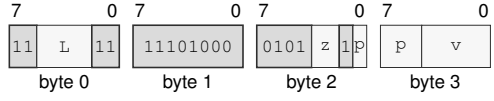
Detailed Semantics

VFSQRT computes the square root of each active H, S, or D element and rounds the result according to the floating-point environment. The register form reads and overwrites each active destination lane. The memory-source form reads each active memory element and writes the result to the corresponding destination lane. An inactive destination lane retains its previous value.

The memory-destination form computes the square root of each active source-register element and writes the result to the corresponding memory element. Inactive memory elements retain their previous values, making the destination transaction a lane-wise read-modify-write.

Encodings:

`VFSQRT.{H|S|D}(z) Pn(p), Vn(v)`
Instruction Format:



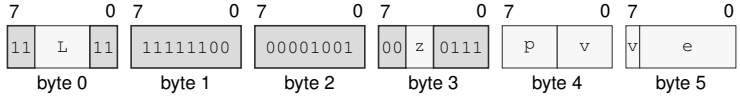
Format: Instruction format for VFSQRT.{H|S|D}(z) Pn(p), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFSQRT.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



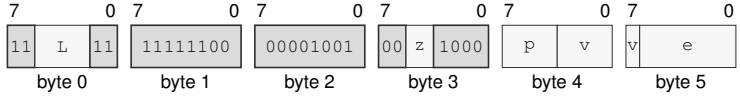
Format: Instruction format for VFSQRT.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFSQRT.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VFSQRT.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VFROUND Vector Floating-Point Round to Integral VFROUND

Operation: Rounds each active floating-point element to an integral value in the same format using the current rounding mode.

Assembler Syntax:
```
VFROUND.{H|S|D}(z) Pn(p), Vn(v)
VFROUND.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)
VFROUND.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

Detailed Semantics

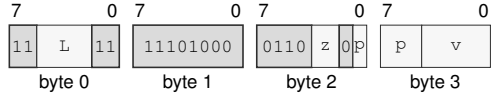
Each active H, S, or D element is rounded to an integral value in the same floating-point format using the current floating-point rounding mode. The register form reads and overwrites each active destination lane. The memory-source form reads each active memory element and writes the result to the corresponding destination lane. An inactive destination lane retains its previous value.

The memory-destination form rounds each active source-register element and writes the result to the corresponding memory element. Inactive memory elements retain their previous values, making the destination transaction a lane-wise read-modify-write.

Encodings:

`VFROUND.{H|S|D}(z) Pn(p), Vn(v)`

Instruction Format:



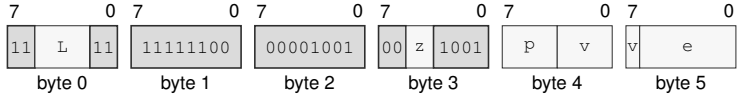
Format: Instruction format for VFROUND.{H|S|D}(z) Pn(p), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFROUND.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



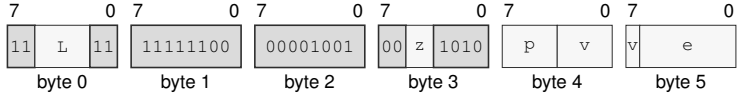
Format: Instruction format for VFROUND.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFROUND.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VFROUND.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VFTRUNC Vector Floating-Point Truncate to Integral VFTRUNC

Operation: Rounds each active floating-point element toward zero to an integral value in the same format.

Assembler Syntax:
```
VFTRUNC.{H|S|D}(z) Pn(p), Vn(v)
VFTRUNC.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)
VFTRUNC.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

Detailed Semantics

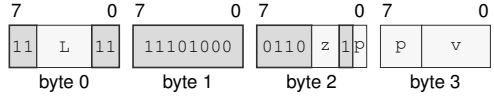
Each active H, S, or D element is rounded toward zero to an integral value in the same floating-point format. The register form reads and overwrites each active destination lane. The memory-source form reads each active memory element and writes the result to the corresponding destination lane. An inactive destination lane retains its previous value.

The memory-destination form rounds each active source-register element and writes the result to the corresponding memory element. Inactive memory elements retain their previous values, making the destination transaction a lane-wise read-modify-write.

Encodings:

`VFTRUNC.{H|S|D}(z) Pn(p), Vn(v)`

Instruction Format:



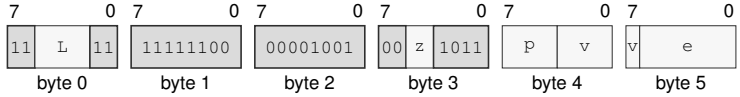
Format: Instruction format for VFTRUNC.{H|S|D}(z) Pn(p), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFTRUNC.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



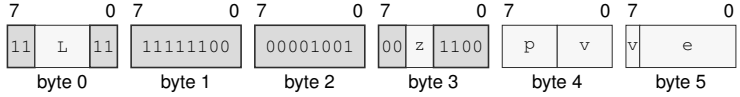
Format: Instruction format for VFTRUNC.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFTRUNC.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VFTRUNC.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VFFLOOR — Vector Floating-Point Floor — VFFLOOR

Operation: Rounds each active floating-point element downward to an integral value in the same format.

Assembler Syntax:
```
VFFLOOR.{H|S|D}(z) Pn(p), Vn(v)
VFFLOOR.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)
VFFLOOR.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

Detailed Semantics

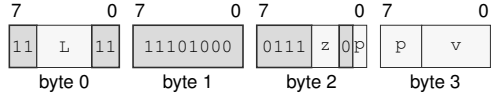
Each active H, S, or D element is rounded downward to an integral value in the same floating-point format. The register form reads and overwrites each active destination lane. The memory-source form reads each active memory element and writes the result to the corresponding destination lane. An inactive destination lane retains its previous value.

The memory-destination form rounds each active source-register element and writes the result to the corresponding memory element. Inactive memory elements retain their previous values, making the destination transaction a lane-wise read-modify-write.

Encodings:

`VFFLOOR.{H|S|D}(z) Pn(p), Vn(v)`

Instruction Format:



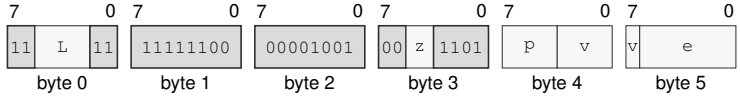
Format: Instruction format for VFFLOOR.{H|S|D}(z) Pn(p), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFFLOOR.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



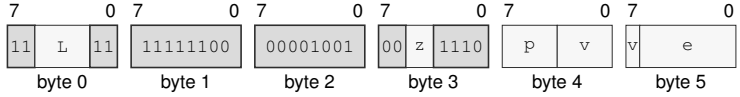
Format: Instruction format for VFFLOOR.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFFLOOR.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VFFLOOR.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VFCEIL — Vector Floating-Point Ceiling — VFCEIL

Operation: Rounds each active floating-point element upward to an integral value in the same format.

Assembler Syntax:
```
VFCEIL.{H|S|D}(z) Pn(p), Vn(v)
VFCEIL.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)
VFCEIL.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

Detailed Semantics

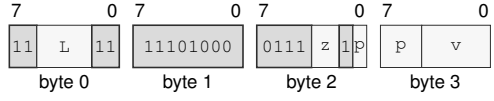
Each active H, S, or D element is rounded upward to an integral value in the same floating-point format. The register form reads and overwrites each active destination lane. The memory-source form reads each active memory element and writes the result to the corresponding destination lane. An inactive destination lane retains its previous value.

The memory-destination form rounds each active source-register element and writes the result to the corresponding memory element. Inactive memory elements retain their previous values, making the destination transaction a lane-wise read-modify-write.

Encodings:

`VFCEIL.{H|S|D}(z) Pn(p), Vn(v)`

Instruction Format:



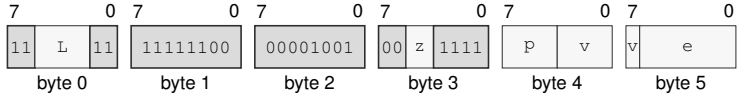
Format: Instruction format for VFCEIL.{H|S|D}(z) Pn(p), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFCEIL.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



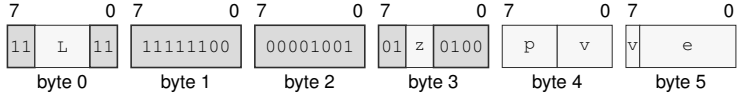
Format: Instruction format for VFCEIL.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFCEIL.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VFCEIL.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VFCLASS Vector Floating-Point Classify VFCLASS

Operation: Writes a floating-point classification bitmap for each active element.

Assembler Syntax:
```
VFCLASS.{H|S|D}(z) Pn(p), Vn(v)
VFCLASS.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)
VFCLASS.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

Detailed Semantics

For each active H, S, or D element, bits 0 through 9 of the result form a one-hot bitmap for negative infinity, negative normal, negative subnormal, negative zero, positive zero, positive subnormal, positive normal, positive infinity, signaling NaN, and quiet NaN, respectively. All higher bits of the destination element are zero.

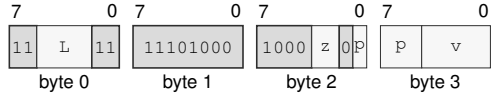
The register form classifies each old destination lane and writes the result back to that lane. The memory-source form classifies each active memory element and writes the result to the corresponding destination lane. An inactive destination lane retains its previous value.

The memory-destination form classifies each active source-register lane and writes the result to the corresponding memory element. An inactive memory element retains its previous value, making the destination transaction a lane-wise read-modify-write.

Encodings:

`VFCLASS.{H|S|D}(z) Pn(p), Vn(v)`

Instruction Format:



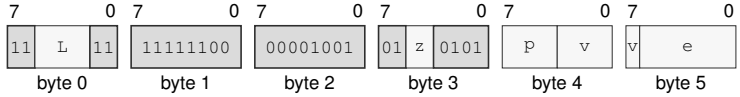
Format: Instruction format for VFCLASS.{H|S|D}(z) Pn(p), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFCLASS.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



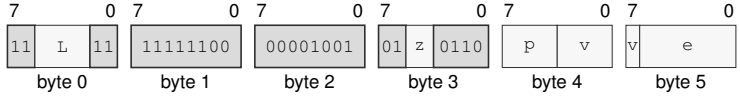
Format: Instruction format for VFCLASS.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFCLASS.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VFCLASS.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VFCMPcc Vector Floating-Point Conditional Compare VFCMPcc

Operation: Compares active floating-point lanes and writes a complete predicate image.

Assembler Syntax:
```
VFCMPcc.{H|S|D}(x) Pn(p), Vn(v), Vn(w), Pn(q)
VFCMPcc.{H|S|D}(x) Pn(p), Vn(v), <ea>(e), Pn(q)
```

Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

Detailed Semantics

VFCMPcc compares corresponding active floating-point elements under the encoded condition. The register form reads both elements from vector registers. The memory form reads the right-hand element from the corresponding active VEA location.

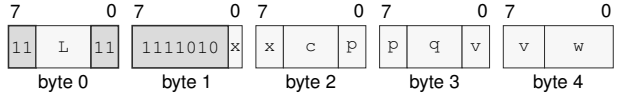
H, S, and D forms apply the selected floating-point condition under the common floating-point comparison rules.

The destination is a complete predicate image. A significant result bit is set when its governing bit is set and the selected relation is true. Every other predicate bit is clear.

Encodings:

`VFCMPcc.{H|S|D}(x) Pn(p), Vn(v), Vn(w), Pn(q)`

Instruction Format:



Format: Instruction format for VFCMPcc.{H|S|D}(x) Pn(p), Vn(v), Vn(w), Pn(q)

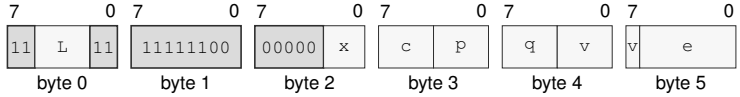
Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x5..0x7`; reason `reserved_vector_element_type`.

Operand constraint: Role `cc`; relation `allowed`; values `0x2..0x3, 0x8..0x9, 0xc..0xf`; reason `reserved_vector_comparison_condition`.

`VFCMPcc.{H|S|D}(x) Pn(p), Vn(v), <ea>(e), Pn(q)`

Instruction Format:



Format: Instruction format for VFCMPcc.{H|S|D}(x) Pn(p), Vn(v), <ea>(e), Pn(q)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x5..0x7`; reason `reserved_vector_element_type`.

Operand constraint: Role `cc`; relation `allowed`; values `0x2..0x3, 0x8..0x9, 0xc..0xf`; reason `reserved_vector_comparison_condition`.

VFADD — Vector Floating-Point Add — VFADD

Operation: Adds corresponding active floating-point vector elements.

Assembler Syntax:
```
VFADD.{H|S|D}(x) Pn(p), Vn(v), Vn(w)
VFADD.{H|S|D}(x) Pn(p), <ea>(e), Vn(v)
VFADD.{H|S|D}(x) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

Detailed Semantics

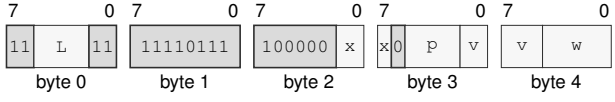
VFADD applies selected floating-point addition independently to each active vector lane. The governing predicate selects active lanes; an inactive destination lane retains its previous value.

For H, S, and D elements, each active result is rounded according to the floating-point environment, and a completed operation accrues any generated NV, OF, UF, and NX causes in FFLAGS; DZ is unchanged. If a generated floating-point exception is enabled in any active lane, VFADD raises FLOATING_POINT_EXCEPTION and commits neither destination lanes nor accrued FFLAGS causes from the faulting operation.

Encodings:

`VFADD.{H|S|D}(x) Pn(p), Vn(v), Vn(w)`

Instruction Format:



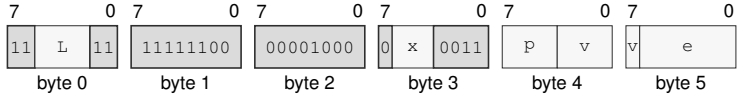
Format: Instruction format for VFADD.{H|S|D}(x) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x5..0x7`; reason `reserved_vector_element_type`.

`VFADD.{H|S|D}(x) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



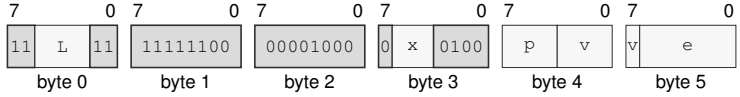
Format: Instruction format for VFADD.{H|S|D}(x) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x5..0x7`; reason `reserved_vector_element_type`.

`VFADD.{H|S|D}(x) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VFADD.{H|S|D}(x) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x5..0x7`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VFSUB Vector Floating-Point Subtract VFSUB

Operation: Subtracts corresponding active floating-point vector elements.

Assembler Syntax:
```
VFSUB.{H|S|D}(x) Pn(p), Vn(v), Vn(w)
VFSUB.{H|S|D}(x) Pn(p), <ea>(e), Vn(v)
VFSUB.{H|S|D}(x) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

Detailed Semantics

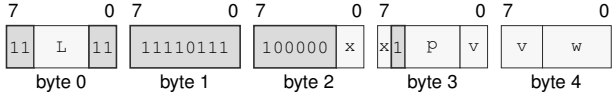
VFSUB subtracts each corresponding active source element from its destination element. For H, S, and D elements, the exact difference is rounded according to the floating-point environment. The register form and the memory-source form write the result to the destination vector lane. An inactive destination lane retains its previous value.

The memory-destination form reads each active destination element, subtracts the corresponding source-register element, and writes the result back to that memory element. Inactive memory elements retain their previous values, making the destination transaction a lane-wise read-modify-write.

Encodings:

`VFSUB.{H|S|D}(x) Pn(p), Vn(v), Vn(w)`

Instruction Format:



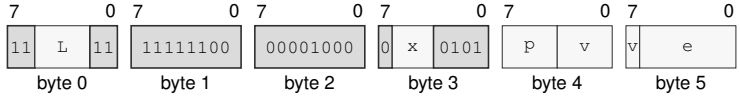
Format: Instruction format for VFSUB.{H|S|D}(x) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x5..0x7`; reason `reserved_vector_element_type`.

`VFSUB.{H|S|D}(x) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



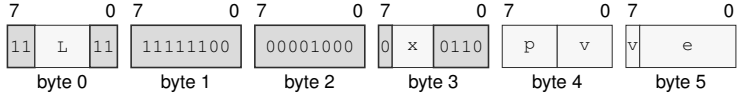
Format: Instruction format for VFSUB.{H|S|D}(x) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x5..0x7`; reason `reserved_vector_element_type`.

`VFSUB.{H|S|D}(x) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VFSUB.{H|S|D}(x) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x5..0x7`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VFMUL — Vector Floating-Point Multiply — VFMUL

Operation: Multiplies corresponding active floating-point elements.

Assembler Syntax:
```
VFMUL.{H|S|D}(x) Pn(p), Vn(v), Vn(w)
VFMUL.{H|S|D}(x) Pn(p), <ea>(e), Vn(v)
VFMUL.{H|S|D}(x) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

Detailed Semantics

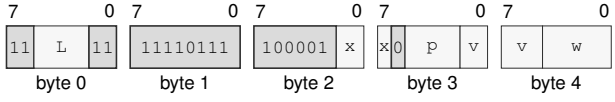
VFMUL multiplies each active destination element by its corresponding source element. For H, S, and D elements, the exact product is rounded according to the floating-point environment. The register form and the memory-source form write the result to the destination vector lane. An inactive destination lane retains its previous value.

The memory-destination form reads each active destination element, multiplies it by the corresponding source-register element, and writes the result back to that memory element. Inactive memory elements retain their previous values, making the destination transaction a lane-wise read-modify-write.

Encodings:

`VFMUL.{H|S|D}(x) Pn(p), Vn(v), Vn(w)`

Instruction Format:



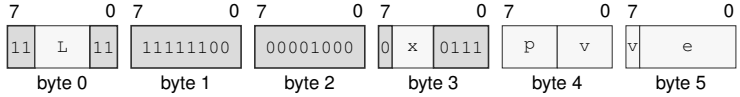
Format: Instruction format for VFMUL.{H|S|D}(x) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x5..0x7`; reason `reserved_vector_element_type`.

`VFMUL.{H|S|D}(x) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



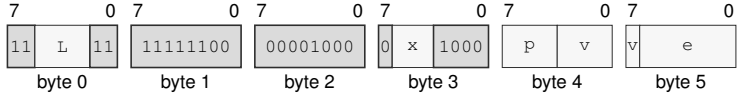
Format: Instruction format for VFMUL.{H|S|D}(x) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x5..0x7`; reason `reserved_vector_element_type`.

`VFMUL.{H|S|D}(x) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VFMUL.{H|S|D}(x) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x5..0x7`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VFMIN — Vector Floating-Point Minimum — VFMIN

Operation: Selects the floating-point minimum in each active lane.

Assembler Syntax:

```
VFMIN.{H|S|D}(z) Pn(p), Vn(v), Vn(w)
VFMIN.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)
VFMIN.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags:

```
FP
VECTOR
VECTORFP
```

Detailed Semantics

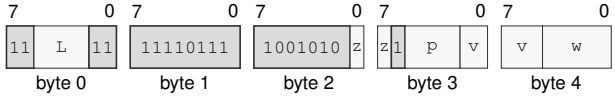
VFMIN applies the scalar floating-point minimum rules to each active H, S, or D destination element and its corresponding source element. The register form and the memory-source form write the selected value to the destination vector lane. An inactive destination lane retains its previous value.

The memory-destination form combines each active destination element with the corresponding source-register element and writes the selected value back to that memory element. Inactive memory elements retain their previous values, making the destination transaction a lane-wise read-modify-write.

Encodings:

`VFMIN.{H|S|D}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



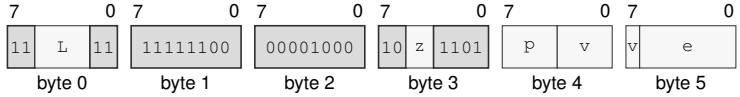
Format: Instruction format for VFMIN.{H|S|D}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFMIN.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



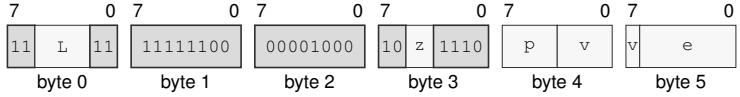
Format: Instruction format for VFMIN.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFMIN.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VFMIN.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VFMAX — Vector Floating-Point Maximum — VFMAX

Operation: Selects the floating-point maximum in each active lane.

Assembler Syntax:
```
VFMAX.{H|S|D}(z) Pn(p), Vn(v), Vn(w)
VFMAX.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)
VFMAX.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

Detailed Semantics

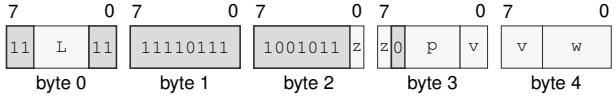
VFMAX applies the scalar floating-point maximum rules to each active H, S, or D destination element and its corresponding source element. The register form and the memory-source form write the selected value to the destination vector lane. An inactive destination lane retains its previous value.

The memory-destination form combines each active destination element with the corresponding source-register element and writes the selected value back to that memory element. Inactive memory elements retain their previous values, making the destination transaction a lane-wise read-modify-write.

Encodings:

`VFMAX.{H|S|D}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



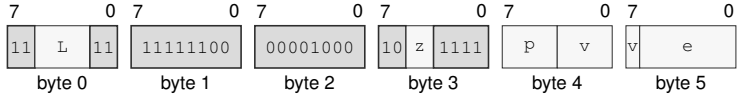
Format: Instruction format for VFMAX.{H|S|D}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFMAX.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



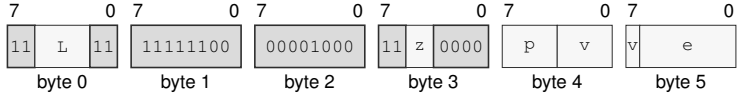
Format: Instruction format for VFMAX.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFMAX.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VFMAX.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VFDIV — Vector Floating-Point Divide — VFDIV

Operation: Divides corresponding active floating-point elements.

Assembler Syntax:
```
VFDIV.{H|S|D}(z) Pn(p), Vn(v), Vn(w)
VFDIV.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)
VFDIV.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

Detailed Semantics

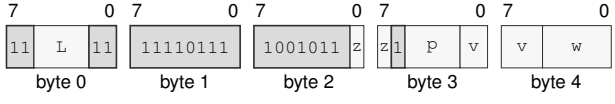
VFDIV divides each active H, S, or D destination element by its corresponding source element and rounds the result according to the floating-point environment. The register form and the memory-source form write the result to the destination vector lane. An inactive destination lane retains its previous value.

The memory-destination form reads each active destination element, divides it by the corresponding source-register element, and writes the result back to that memory element. Inactive memory elements retain their previous values, making the destination transaction a lane-wise read-modify-write.

Encodings:

`VFDIV.{H|S|D}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



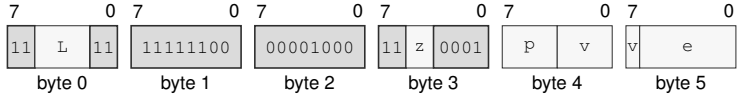
Format: Instruction format for VFDIV.{H|S|D}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFDIV.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



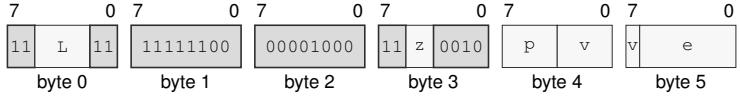
Format: Instruction format for VFDIV.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFDIV.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VFDIV.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VFCOPYSIGN Vector Floating-Point Copy Sign VFCOPYSIGN

Operation: Combines each active destination magnitude with the corresponding source sign bit.

Assembler Syntax:
```
VFCOPYSIGN.{H|S|D}(z) Pn(p), Vn(v), Vn(w)
VFCOPYSIGN.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)
VFCOPYSIGN.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)
```

Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

Detailed Semantics

For each active H, S, or D element, the result takes its magnitude bits from the old destination element and its sign bit from the corresponding source element.

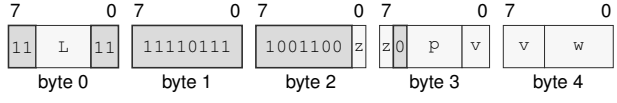
In the register form, the source and old destination come from corresponding vector lanes. With a memory source, the sign bit comes from the active memory element and the magnitude bits come from the old destination lane. In both forms, an inactive destination lane retains its previous value.

The memory-destination form reads the magnitude bits from each active destination memory element and the sign bit from the corresponding source-register lane, then writes the combined result back to that element. An inactive memory element retains its previous value.

Encodings:

`VFCOPYSIGN.{H|S|D}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



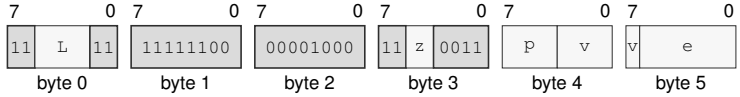
Format: Instruction format for VFCOPYSIGN.{H|S|D}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFCOPYSIGN.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)`

Instruction Format:



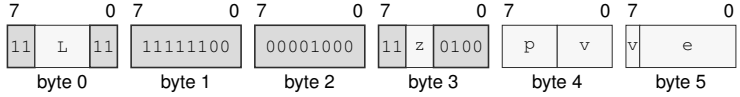
Format: Instruction format for VFCOPYSIGN.{H|S|D}(z) Pn(p), <ea>(e), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFCOPYSIGN.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)`

Instruction Format:



Format: Instruction format for VFCOPYSIGN.{H|S|D}(z) Pn(p), Vn(v), <ea>(e)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

Operand constraint: Role `dst`; relation `excluded`; values `immediate`; reason `invalid_destination`.

VFCVTS Vector Floating-Point Convert to Single Precision VFCVTS

Operation: Converts active half-precision, double-precision, or signed integer elements to single precision.

Assembler Syntax:
```
VFCVTS.{H|D}(z) Pn(p), Vn(v), Vn(w)
VFCVTS.{L|Q}(z) Pn(p), Vn(v), Vn(w)
```

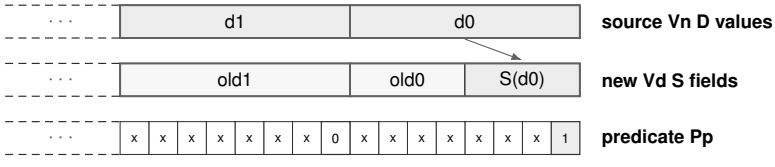
Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

Detailed Semantics

Each active H or D source value is converted to S. The signed-integer forms convert each active L or Q source value to S. The operation container is the larger of the source and 32-bit destination widths. The S result replaces the low 32-bit field; the other bits of an active destination container and every inactive container retain their previous values.

A completed conversion accrues generated NV, OF, UF, and NX causes in FFLAGS; DZ is unchanged. An enabled generated cause raises FLOATING_POINT_EXCEPTION and suppresses destination and newly accrued FFLAGS effects.

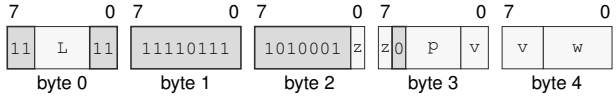


VFCVTS.D double-to-single conversion containers

Encodings:

`VFCVTS.{H|D}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



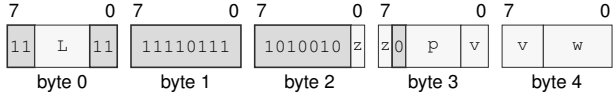
Format: Instruction format for VFCVTS.{H|D}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `1, 3`; reason `reserved_vector_element_type`.

`VFCVTS.{L|Q}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VFCVTS.{L|Q}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x2..0x3`; reason `reserved_vector_element_type`.

VFCVTD Vector Floating-Point Convert to Double Precision VFCVTD

Operation: Converts active half-precision, single-precision, or signed integer elements to double precision.

Assembler Syntax:
```
VFCVTD.{H|S}(z) Pn(p), Vn(v), Vn(w)
VFCVTD.{L|Q}(z) Pn(p), Vn(v), Vn(w)
```

Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

Detailed Semantics

Each active H or S source value is converted to D. The signed-integer forms convert each active L or Q source value to D. The operation uses 64-bit containers, reads the source from the low source-width field, and writes the D result to the complete container. Every inactive container retains its previous value.

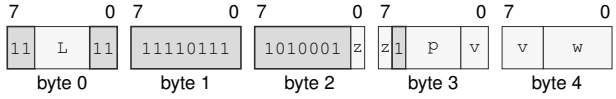
A completed conversion accrues generated NV, OF, UF, and NX causes in FFLAGS; DZ is unchanged. An enabled generated cause raises FLOATING_POINT_EXCEPTION and suppresses destination and newly accrued FFLAGS effects.

VFCVTD.H half-to-double conversion containers

Encodings:

`VFCVTD.{H|S}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



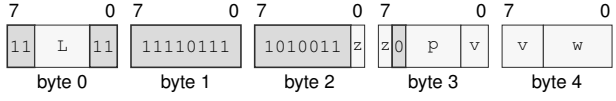
Format: Instruction format for VFCVTD.{H|S}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x2`; reason `reserved_vector_element_type`.

`VFCVTD.{L|Q}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VFCVTD.{L|Q}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x2..0x3`; reason `reserved_vector_element_type`.

VFCVTUS Vector Floating-Point Convert Unsigned to Single Precision VFCVTUS

Operation: Converts active unsigned integer elements to single-precision floating point.

Assembler Syntax: `VFCVTUS.{L|Q}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`VECTOR`
`VECTORFP`

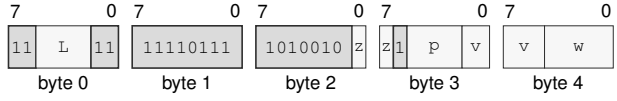
Detailed Semantics

Each active unsigned L or Q source element is converted to an S element and rounded according to the floating-point environment. The result is written into the low single-precision field of its destination container. Bits outside that result field and every inactive destination container retain their previous values.

Encodings:

`VFCVTUS.{L|Q}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VFCVTUS.{L|Q}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x2..0x3`; reason `reserved_vector_element_type`.

VFCVTUD Vector Floating-Point Convert Unsigned to Double Precision VFCVTUD

Operation: Converts active unsigned integer elements to double-precision floating point.

Assembler Syntax: `VFCVTUD.{L|Q}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`VECTOR`
`VECTORFP`

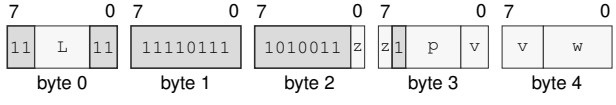
Detailed Semantics

Each active unsigned L or Q source element is converted to a D element and rounded according to the floating-point environment. The result is written into the low double-precision field of its destination container. Every inactive destination container retains its previous value.

Encodings:

`VFCVTUD.{L|Q}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VFCVTUD.{L|Q}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x2..0x3`; reason `reserved_vector_element_type`.

VFCVTL Vector Floating-Point Convert to Signed Longword VFCVTL

Operation: Converts active floating-point elements to signed longwords.

Assembler Syntax: `VFCVTL.{H|S|D}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags:
`FP`
`VECTOR`
`VECTORFP`

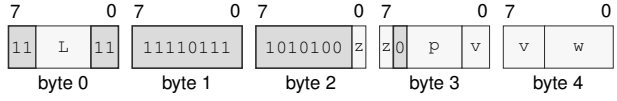
Detailed Semantics

Each active H, S, or D source element is converted toward zero to a signed 32-bit longword under the common floating-point conversion rules. The result is written into the low longword of its destination container. Bits outside that result field and every inactive destination container retain their previous values.

Encodings:

`VFCVTL.{H|S|D}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VFCVTL.{H|S|D}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

VFCVTUL Vector Floating-Point Convert to Unsigned Longword VFCVTUL

Operation: Converts active floating-point elements to unsigned longwords.

Assembler Syntax: `VFCVTUL.{H|S|D}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`VECTOR`
`VECTORFP`

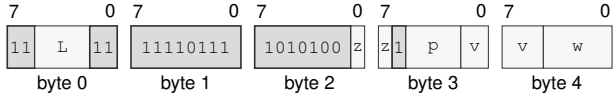
Detailed Semantics

Each active H, S, or D source element is converted toward zero to an unsigned 32-bit longword under the common floating-point conversion rules. The result is written into the low longword of its destination container. Bits outside that result field and every inactive destination container retain their previous values.

Encodings:

`VFCVTUL.{H|S|D}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VFCVTUL.{H|S|D}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

VFCVTQ Vector Floating-Point Convert to Signed Quadword VFCVTQ

Operation: Converts active floating-point elements to signed quadwords.

Assembler Syntax: `VFCVTQ.{H|S|D}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags:
`FP`
`VECTOR`
`VECTORFP`

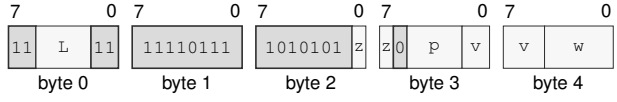
Detailed Semantics

Each active H, S, or D source element is converted toward zero to a signed 64-bit quadword under the common floating-point conversion rules. The result is written into the low quadword of its destination container. Every inactive destination container retains its previous value.

Encodings:

`VFCVTQ.{H|S|D}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VFCVTQ.{H|S|D}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

VFCVTUQ Vector Floating-Point Convert to Unsigned Quadword VFCVTUQ

Operation: Converts active floating-point elements to unsigned quadwords.

Assembler Syntax: `VFCVTUQ.{H|S|D}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`VECTOR`
`VECTORFP`

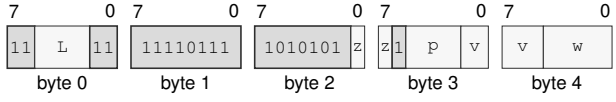
Detailed Semantics

Each active H, S, or D source element is converted toward zero to an unsigned 64-bit quadword under the common floating-point conversion rules. The result is written into the low quadword of its destination container. Every inactive destination container retains its previous value.

Encodings:

`VFCVTUQ.{H|S|D}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VFCVTUQ.{H|S|D}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

VFCVTH Vector Floating-Point Convert to Half Precision VFCVTH

Operation: Converts active single-precision, double-precision, or signed integer elements to half precision.

Assembler Syntax:
```
VFCVTH.{S|D}(z) Pn(p), Vn(v), Vn(w)
VFCVTH.{L|Q}(z) Pn(p), Vn(v), Vn(w)
```

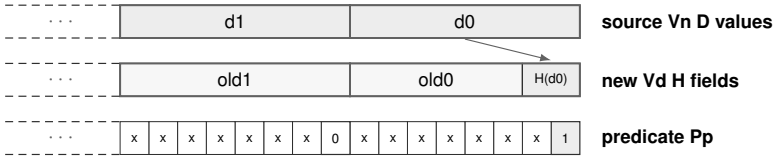
Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

Detailed Semantics

Each active S or D source value is converted to H. The signed-integer forms convert each active L or Q source value to H. The operation container is the larger of the source and 16-bit destination widths. The H result replaces the low 16-bit field; the other bits of an active destination container and every inactive container retain their previous values.

A completed conversion accrues generated NV, OF, UF, and NX causes in FFLAGS; DZ is unchanged. An enabled generated cause raises FLOATING_POINT_EXCEPTION and suppresses destination and newly accrued FFLAGS effects.

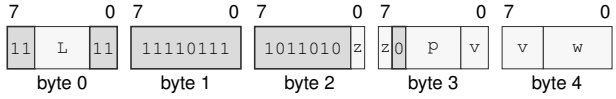


VFCVTH.D double-to-half conversion containers

Encodings:

`VFCVTH.{S|D}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



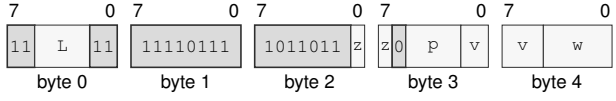
Format: Instruction format for VFCVTH.{S|D}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x2..0x3`; reason `reserved_vector_element_type`.

`VFCVTH.{L|Q}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VFCVTH.{L|Q}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x2..0x3`; reason `reserved_vector_element_type`.

VFCVTUH Vector Floating-Point Convert Unsigned to Half Precision VFCVTUH

Operation: Converts active unsigned integer elements to half-precision floating point.

Assembler Syntax: `VFCVTUH.{L|Q}(z) Pn(p), Vn(v), Vn(w)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`VECTOR`
`VECTORFP`

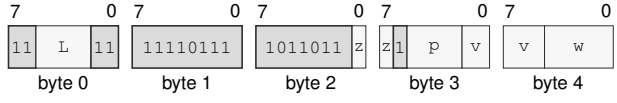
Detailed Semantics

Each active unsigned L or Q source element is converted to an H element and rounded according to the floating-point environment. The result is written into the low half-precision field of its destination container. Bits outside that result field and every inactive destination container retain their previous values.

Encodings:

`VFCVTUH.{L|Q}(z) Pn(p), Vn(v), Vn(w)`

Instruction Format:



Format: Instruction format for VFCVTUH.{L|Q}(z) Pn(p), Vn(v), Vn(w)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x2..0x3`; reason `reserved_vector_element_type`.

VFMADD — Vector Floating-Point Fused Multiply-Add — VFMADD

Operation: Performs fused multiply-add on corresponding active floating-point elements.

Assembler Syntax: `VFMADD.{H|S|D}(z) Pn(p), Vn(v), Vn(w), Vn(y)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`VECTOR`
`VECTORFP`

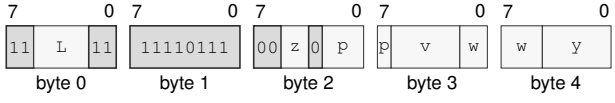
Detailed Semantics

Each active H, S, or D lane forms the fused multiply-add expression from its two source elements and old accumulator destination element. The complete expression has unbounded intermediate range and precision and is rounded only once when written to the destination. An inactive accumulator lane retains its previous value.

Encodings:

`VFMADD.{H|S|D}(z) Pn(p), Vn(v), Vn(w), Vn(y)`

Instruction Format:



Format: Instruction format for VFMADD.{H|S|D}(z) Pn(p), Vn(v), Vn(w), Vn(y)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

VFMSUB Vector Floating-Point Fused Multiply-Subtract VFMSUB

Operation: Performs fused multiply-subtract on corresponding active floating-point elements.

Assembler Syntax: `VFMSUB.{H|S|D}(z) Pn(p), Vn(v), Vn(w), Vn(y)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`VECTOR`
`VECTORFP`

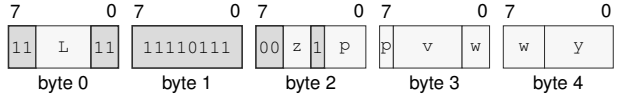
Detailed Semantics

Each active H, S, or D lane forms the fused multiply-subtract expression from its two source elements and old accumulator destination element. The complete expression has unbounded intermediate range and precision and is rounded only once when written to the destination. An inactive accumulator lane retains its previous value.

Encodings:

`VFMSUB.{H|S|D}(z) Pn(p), Vn(v), Vn(w), Vn(y)`

Instruction Format:



Format: Instruction format for VFMSUB.{H|S|D}(z) Pn(p), Vn(v), Vn(w), Vn(y)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

VFNMADD Vector Floating-Point Fused Negative Multiply-Add VFNMADD

Operation: Performs fused negative multiply-add on corresponding active floating-point elements.

Assembler Syntax: `VFNMADD.{H|S|D}(z) Pn(p), Vn(v), Vn(w), Vn(y)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`VECTOR`
`VECTORFP`

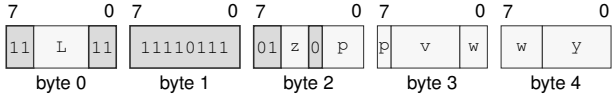
Detailed Semantics

Each active H, S, or D lane forms the negated fused multiply-add expression from its two source elements and old accumulator destination element. The complete expression has unbounded intermediate range and precision and is rounded only once when written to the destination. An inactive accumulator lane retains its previous value.

Encodings:

`VFNMADD.{H|S|D}(z) Pn(p), Vn(v), Vn(w), Vn(y)`

Instruction Format:



Format: Instruction format for VFNMADD.{H|S|D}(z) Pn(p), Vn(v), Vn(w), Vn(y)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

VFNMSUB Vector Floating-Point Fused Negative Multiply-Subtract VFNMSUB

Operation: Performs fused negative multiply-subtract on corresponding active floating-point elements.

Assembler Syntax: `VFNMSUB.{H|S|D}(z) Pn(p), Vn(v), Vn(w), Vn(y)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`VECTOR`
`VECTORFP`

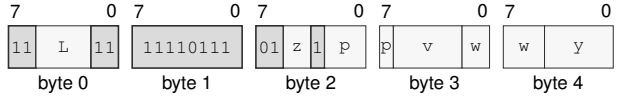
Detailed Semantics

Each active H, S, or D lane forms the negated fused multiply-subtract expression from its two source elements and old accumulator destination element. The complete expression has unbounded intermediate range and precision and is rounded only once when written to the destination. An inactive accumulator lane retains its previous value.

Encodings:

`VFNMSUB.{H|S|D}(z) Pn(p), Vn(v), Vn(w), Vn(y)`

Instruction Format:



Format: Instruction format for VFNMSUB.{H|S|D}(z) Pn(p), Vn(v), Vn(w), Vn(y)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

VFEXTRACT Vector Floating-Point Lane Extract VFEXTRACT

Operation: Copies one indexed floating-point vector lane to a floating-point scalar register.

Assembler Syntax: `VFEXTRACT.{H|S|D}(z) Vn(v), Rn(r), Fn(s)`

Privilege: Unprivileged

Required CPUID flags: `FP`
`VECTOR`
`VECTORFP`

Detailed Semantics

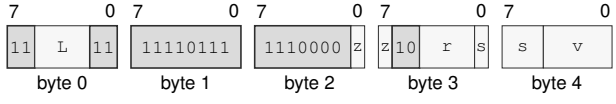
VFEXTRACT interprets the scalar index as unsigned and selects that lane from Vn. The H, S, and D forms copy the selected floating-point value to Fn and leave FFLAGS unchanged.

An index at or above the selected vector element count raises VECTOR_LANE_INDEX_OUT_OF_RANGE. The faulting instruction commits no scalar result.

Encodings:

`VFEXTRACT.{H|S|D}(z) Vn(v), Rn(r), Fn(s)`

Instruction Format:



Format: Instruction format for VFEXTRACT.{H|S|D}(z) Vn(v), Rn(r), Fn(s)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

VFINSERT Vector Floating-Point Lane Insert VFINSERT

Operation: Replaces one indexed vector lane with a floating-point scalar value.

Assembler Syntax: `VFINSERT.{H|S|D}(z) Fn(r), Rn(s), Vn(v)`

Privilege: Unprivileged

Required CPUID flags:
`FP`
`VECTOR`
`VECTORFP`

Detailed Semantics

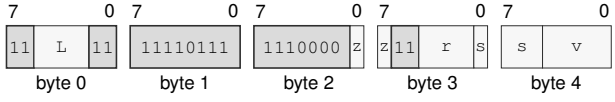
VFINSERT interprets the scalar index as unsigned and replaces only that lane of Vd. The H, S, and D forms copy the selected floating-point value from Fn and leave FFLAGS unchanged. Every lane at another index retains its previous value.

An index at or above the selected vector element count raises VECTOR_LANE_INDEX_OUT_OF_RANGE. The faulting instruction commits no destination change.

Encodings:

`VFINSERT.{H|S|D}(z) Fn(r), Rn(s), Vn(v)`

Instruction Format:



Format: Instruction format for VFINSERT.{H|S|D}(z) Fn(r), Rn(s), Vn(v)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

VFREDADD Vector Floating-Point Add Reduction VFREDADD

Operation: Reduces active floating-point elements to one scalar sum.

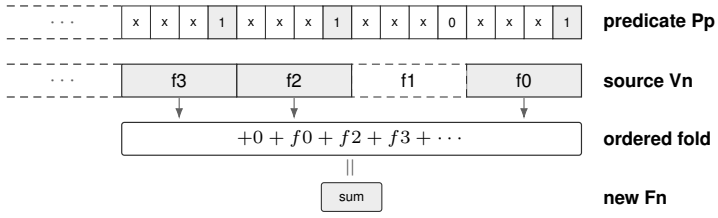
Assembler Syntax:
```
VFREDADD.{H|S|D}(z) Pn(p), Vn(v), Fn(r)
VFREDADD.{H|S|D}(z) Pn(p), <ea>(e), Fn(r)
```

Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

Detailed Semantics

The governing predicate selects the source elements from Vn or the corresponding contiguous memory locations. The H, S, and D forms begin with positive zero and add active elements in increasing logical-lane order without reassociation. An empty floating-point selection therefore produces positive zero. A completed floating-point reduction accrues generated NV, OF, UF, and NX causes in FFLAGS; DZ is unchanged. An enabled generated cause raises FLOATING_POINT_EXCEPTION and suppresses the scalar result and newly accrued causes.

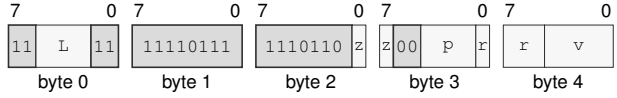


VFREDADD.S ordered floating-point reduction

Encodings:

`VFREDADD.{H|S|D}(z) Pn(p), Vn(v), Fn(r)`

Instruction Format:



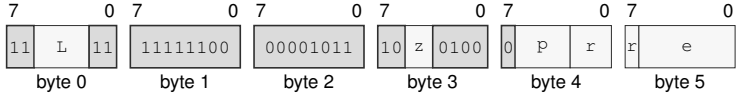
Format: Instruction format for VFREDADD.{H|S|D}(z) Pn(p), Vn(v), Fn(r)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFREDADD.{H|S|D}(z) Pn(p), <ea>(e), Fn(r)`

Instruction Format:



Format: Instruction format for VFREDADD.{H|S|D}(z) Pn(p), <ea>(e), Fn(r)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

VFREDMIN Vector Floating-Point Minimum Reduction VFREDMIN

Operation: Reduces active floating-point elements to one minimum value.

Assembler Syntax:
```
VFREDMIN.{H|S|D}(z) Pn(p), Vn(v), Fn(r)
VFREDMIN.{H|S|D}(z) Pn(p), <ea>(e), Fn(r)
```

Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

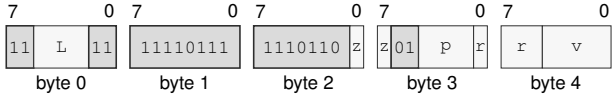
Detailed Semantics

The governing predicate selects the active H, S, or D source elements. The register form reads them from Vn, and the memory form reads them from the corresponding active VEA locations. VFREDMIN reduces the selected elements under the scalar floating-point minimum rules and writes the selected-format result to Fn. If no element is selected, the result is positive infinity.

Encodings:

`VFREDMIN.{H|S|D}(z) Pn(p), Vn(v), Fn(r)`

Instruction Format:



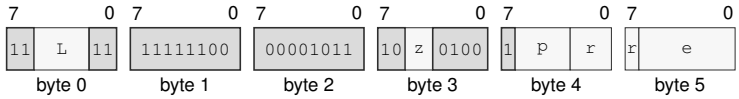
Format: Instruction format for VFREDMIN.{H|S|D}(z) Pn(p), Vn(v), Fn(r)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFREDMIN.{H|S|D}(z) Pn(p), <ea>(e), Fn(r)`

Instruction Format:



Format: Instruction format for VFREDMIN.{H|S|D}(z) Pn(p), <ea>(e), Fn(r)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

VFREDMAX Vector Floating-Point Maximum Reduction VFREDMAX

Operation: Reduces active floating-point elements to one maximum value.

Assembler Syntax:
```
VFREDMAX.{H|S|D}(z) Pn(p), Vn(v), Fn(r)
VFREDMAX.{H|S|D}(z) Pn(p), <ea>(e), Fn(r)
```

Privilege: Unprivileged

Required CPUID flags:
```
FP
VECTOR
VECTORFP
```

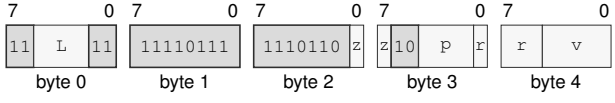
Detailed Semantics

The governing predicate selects the active H, S, or D source elements. The register form reads them from Vn, and the memory form reads them from the corresponding active VEA locations. VFREDMAX reduces the selected elements under the scalar floating-point maximum rules and writes the selected-format result to Fn. If no element is selected, the result is negative infinity.

Encodings:

`VFREDMAX.{H|S|D}(z) Pn(p), Vn(v), Fn(r)`

Instruction Format:



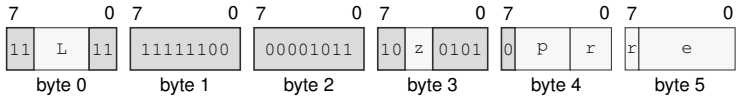
Format: Instruction format for VFREDMAX.{H|S|D}(z) Pn(p), Vn(v), Fn(r)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

`VFREDMAX.{H|S|D}(z) Pn(p), <ea>(e), Fn(r)`

Instruction Format:



Format: Instruction format for VFREDMAX.{H|S|D}(z) Pn(p), <ea>(e), Fn(r)

Restrictions:

Operand constraint: Role `size`; relation `allowed`; values `0x1..0x3`; reason `reserved_vector_element_type`.

A REFERENCE INDEXES

This appendix provides the canonical field-name lookup and the list of implementation-defined disclosure points defined by this manual.

A.1 Canonical Field Names

The following spellings are canonical within the named owner. A one-letter field is interpreted in its owner or instruction-encoding context.

Table A-1. Canonical Field Names

Owner	Canonical fields	Normative definition
`FLAGS`	`FLAGS.Z`, `FLAGS.N`, `FLAGS.C`, `FLAGS.V`	definition (p. 8)
`STATUS`	`STATUS.IE`, `STATUS.PM`, `STATUS.RF`, `STATUS.TF`, `STATUS.NI`, `STATUS.EA`	definition (p. 8)
`PTCR`	`PTCR.ROOT_PAGE`, `PTCR.TT`, `PTCR.PE`	definition (p. 13)
`ASCR`	`ASCR.ASID`, `ASCR.AE`	definition (p. 13)
`ECR`	`ECR.MAX_EDEPTH`, `ECR.NMI_P`, `ECR.V`	definition (p. 13)
`UCTL`	`UCTL.V`, `UCTL.STATUS`, `UCTL.FLAGS`	definition (p. 13)
`PMC`	`PMC.EN`	definition (p. 13)
`PTE`	`PTE.P`, `PTE.T`, `PTE.AM`, `PTE.R`, `PTE.W`, `PTE.X`, `PTE.U`, `PTE.G`, `PTE.A`, `PTE.D`, `PTE.CP`, `PTE.SW`, `PTE.PFN`, `PTE.NEXT_TABLE`	definition (p. 82)
`instruction_header`	`EXTENDED_HEADER.L`	definition (p. 23)

A.2 Implementation-Defined Disclosures

An implementation-defined selection is stable for the implementation revision named by CPUID and is published through the channel named by its defining rule. The publication identifies the selected value or behavior and the processor revisions to which it applies.

Table A-2. Implementation-Defined Disclosure Register

Item	Defining rule	Publication
Higher CPUID classes	CPUID Feature Discovery, Base Identity, Leaf Directory	CPUID class and leaf directories plus implementation documentation for the class payload
Performance-counter identifiers	CPUID Performance-Counter Discovery, RDPMC	CPUID PERFORMANCE_COUNTERS presence results plus implementation documentation naming each counter event
Auxiliary event syndrome	Machine-check EVENT_AUX payload	Platform documentation or firmware interface documentation for the event source
Maximum implemented interrupt identity	Interrupt File, ICAP	ICAP.MAX_ID
Architectural timebase frequency	Architectural Timebase, Timebase Discovery	CPUID TIMEBASE.TICKS_PER_SECOND
FPTRANSA approximation contract	FPTRANSA Common Model, CPUID accuracy contracts	CPUID accuracy-contract discovery plus implementation documentation for the deterministic approximation
CFI pointer-authentication construction	Control-Flow Integrity Model, Protected call and return instructions	Implementation documentation naming the MAC construction, input serialization, tag mapping, and processor revisions to which they apply