Base Instruction Set

This document describes the instruction opcode layout for the base instruction set of the ExVM2 machine. These perform operations on the 16 general purpose registers and 16 floating point registers. No memory direct operations, other than load and store are implemented.

Notes

This document does not formally describe the assembler syntax but proposes the generalised syntactical form for each described operation:

- Operations which imply signed integer arithmetic should use i<size> as their type suffix.
- Operations which imply unsigned integer arithmetic should use u<size> as their type suffix.
- Operations which imply floating point arithmetic should use f<size> as their type suffix.
- Operations which imply pure bitwise logic only should use <size> as their type suffix.
- Labels refer to PC relative identifiers.
- Symbols refer to imported/exported identifiers that are resolved to a 20-bit enumerated ID and resolved at load/link time.

There are 4 implementation levels. Each level includes all features of all preceding levels:

- Level 0 supports operations on 8, 16 and 32-bit integer types only.
- Level 1 adds support for operations on 32-bit floating point types.
- Level 2 adds support for operations on 64-bit integer and floating point types.
- Level 3 adds the requirement for 64-bit memory support.

Opcode Structure

Instructions are composed of one or more 16-bit words.

- The upper 8-bits of the first word define the basic operation.
- The lower 8-bits of the first word usually encode a source and destination register pair, but may also contain:
    - Single 8 bit operand.
    - Short branch displacement.
    - Lowest 8 bits of a 24-bit signed branch displacement.
    - Extension opcode identifier.
- Common arithmetic/logic operations require 1 word. Extension words words may contain:
    - Immediate operand data.
    - Branch displacements.

    - Register masks.
    - Symbol ID values.
    - Additional register definitions or other field values.
- Monadic instructions may utilise the source (or destination) nybble of the first instruction opcode to encode some other data:
    - Lowest 4 bits of a signed 20-bit branch displacement.
    - Lowest 4 bits of an unsigned 20-bit resolved symbol ID.
    - An operation mode.
- Branch displacements can be 8, 16, 20 or 24 bits, depending on the instruction, but are always signed and measured in 16 bit words.
- Opcodes 240-255 are reserved for extension instruction sets.

Enumerations

Opcodes are grouped first by implementation level:

- Applies to first level instruction opcodes.
- Applies to second-level type selection fields.
- Ensures all ranges are contiguous for a given implementation level.

Where multiple operand types are supported:

- Unsigned integer types are positioned first.
- Signed integer types are positioned second.
- Floating point types are positioned last.

Overall order of enumeration precedence for type is given below:

Type	**Level**	**Order**
u8	0	0
u16	0	1
u32	0	2
i8	0	3
i16	0	4
i32	0	5
f32	1	6
u64	2	7
i64	2	8
f64	2	9

Where an operation supports only a subset of the above types, those types shall be enumerated in the order they occur above.

Machine Model

An overview of the machine model is shown below.

General Purpose Registers

GPR	63	31	15	7
r0				
r1				
,,,				
r15				

Floating Point Registers

Floating point registers are only present in level 1 implementations and above.

FPR	63	31
f0		
f1		
,,,		
f15		

Special Purpose Registers

These registers are not directly accessible.

Register	Description	VM Access	Host access
Program Counter	Pointer, immutable, 16-bit aligned	N/A	Read/write
Register Stack	Pointer, mutable, 64-bit aligned	Save/Restore	Read/Write
Data Stack	Pointer, mutable, 8-bit aligned	Push/Pop	Read/Write
Return Stack	Pointer, host address size aligned	Call/Return	Read
Status	Integer	Set on error	Read/write

- The data stack is intended for arbitrary argument passing and local storage allocation.
    - Alignment is managed in code.
- The register stack is intended for register spilling. It is always aligned to the full register width:
    - 32-bit on Level 0
    - 64-bit on Level 2 and above.
- Return stack is used for the call return address only.

Conditional Comparison

ExVM does not maintain a condition code register. Instead there are a number of instructions that accept a comparison type. The comparison type is an enumeration that describes both the operation and the implied operand types.

As with regular opcodes, some operations require level 1 or level 2 implementation.

Arithmetic Comparisons

The full set of arithmetic equality and inequality tests are provided for all unsigned integer, signed integer and floating point operand types:

CC	Comparison	Type / Size	Level
EQ_8	rA = rB, bitwise identical	8	0
EQ_16	rA == rB, bitwise identical	16	0
EQ_32	rA == rB, bitwise identical	32	0
EQ_64	rA == rB, bitwise identical	64	2
EQ_F32	fA == fB, bitwise identical	f32	1
EQV_F32	fA == fB, within epsilon	f32	1
EQ_F64	fA == fB, bitwise identical	f64	2
EQV_F64	fA == fB, within epsilon	f64	2
NE_8	rA != rB	8	0
NE_16	rA != rB	16	0
NE_32	rA != rB	32	0
NE_64	rA != rB	64	2
NE_F32	fA != fB	f32	1
NE_F64	fA != fB	f64	2

CC	Comparison	Type / Size	Level
LT_U8	rA < rB, unsigned	u8	0
LT_U16	rA < rB, unsigned	u16	0
LT_U32	rA < rB, unsigned	u32	0
LT_U64	rA < rB, unsigned	u64	2
LT_I8	rA < rB	i8	0
LT_I16	rA < rB	i16	0
LT_I32	rA < rB	i32	0
LT_I64	rA < rB	i64	2
LT_F32	fA < fB	f32	1
LT_F64	fA < fB	f64	2
LTEQ_U8	rA <= rB, unsigned	u8	0
LTEQ_U16	rA <= rB, unsigned	u16	0
LTEQ_U32	rA <= rB, unsigned	u32	0
LTEQ_U64	rA <= rB, unsigned	u64	2
LTEQ_I8	rA <= rB	i8	0
LTEQ_I16	rA <= rB	i16	0
LTEQ_I32	rA <= rB	i32	0
LTEQ_I64	rA <= rB	i64	2
LTEQ_F32	fA <= fB	f32	1
LTEQ_F64	fA <= fB	f64	2
GREQ_U8	rA >= rB, unsigned	u8	0
GREQ_U16	rA >= rB, unsigned	u16	0
GREQ_U32	rA >= rB, unsigned	u32	0
GREQ_U64	rA >= rB, unsigned	u64	2
GREQ_I8	rA >= rB	i8	0
GREQ_I16	rA >= rB	i16	0
GREQ_I32	rA >= rB	i32	0
GREQ_I64	rA >= rB	i64	2
GREQ_F32	fA >= fB	f32	1
GREQ_F64	fA >= fB	f64	2
GR_U8	rA > rB, unsigned	u8	0
GR_U16	rA > rB, unsigned	u16	0

CC	Comparison	Type / Size	Level
GR_U32	rA > rB, unsigned	u32	0
GR_U64	rA > rB, unsigned	u64	2
GR_I8	rA > rB	i8	0
GR_I16	rA > rB	i16	0
GR_I32	rA > rB	i32	0
GR_I64	rA > rB	i64	2
GR_F32	fA > fB	f32	1
GR_F64	fA > fB	f64	2

Logical Comparisons

A set of logical comparisons are included that allow the testing of individual bits and other logical intersections of the source and destination operand. Where indicated, some bit tests interpret the source operand as an immediate value in the range 0 - 15:

CC	Comparison	Type / Size	Level
BSET_32	0 != rB & (1 << rA), rA mod 32	32	0
BSET_64	0 != rB & (1 << rA), rA mod 64	64	2
BSETI_0	0 != rB & (1 << A), A #0-15	16	0
BSETI_1	0 != rB & (1 << (A + 16)), A #0-15	32	0
BSETI_2	0 != rB & (1 << (A + 32)), A #0-15	64	2
BSETI_3	0 != rB & (1 << (A + 48)), A #0-15	64	2
BCLR_32	0 == (rB & (1 << rA)), rA mod 32	32	0
BCLR_64	0 == (rB & (1 << rA)), rA mod 64	64	2
BCLRI_0	0 == (rB & (1 << A)), A #0-15	16	0
BCLRI_1	0 == (rB & (1 << (A + 16))), A #0-15	32	0
BCLRI_2	0 == (rB & (1 << (A + 32))), A #0-15	64	2
BCLRI_3	0 == (rB & (1 << (A + 48))), A #0-15	64	2
AND_8	0 != rA & rB	8	0
AND_16	0 != rA & rB	16	0
AND_32	0 != rA & rB	32	0
AND_64	0 != rA & rB	64	2

Instruction Description Format

Each opcode, or group of related opcodes, are documented in the format shown below. Where the opcode name contains a lower case x, this indicates the opcode has multiple forms, usually implying the data type or word size.

OPCODE

Summary description of the operation.

generalised syntactical form(s)

Detailed description of the operation.

List of types or variants supported.

Type	**Opcode**	**Level**	**Operand**	
First type	Enumerated opcode	Level	Upper nybble	Lower nybble
Second type	Enumerated opcode	Level		
...	...		...	...

Type	**Extension Word**
Applicable type	Extension word interpretation

Additional notes, where relevant.

ABS_x

Absolute value.

abs.i<size> rS, rD

abs.f<size> fS, fD

The positive magnitude of the value in S is stored in D. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit integer types are supported.
32 and 64-bit floating point types are supported.

Type	Opcode	Level	Operand	
i8	ABS_I8	0	S	D
i16	ABS_I16	0	S	D
i32	ABS_I32	0	S	D
f32	ABS_F32	1	S	D
i64	ABS_I64	2	S	D
f64	ABS_F64	2	S	D

ADD_x

Arithmetic sum.

add.i<size> rS, rD

add.f<size> fS, fD

The value in S is added to the value in D and the result stored in D. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit integer types are supported.
32 and 64-bit floating point types are supported.

Type	Opcode	Level	Operand	
i8	ADD_I8	0	S	D
i16	ADD_I16	0	S	D
i32	ADD_I32	0	S	D
f32	ADD_F32	1	S	D
i64	ADD_I64	2	S	D
f64	ADD_F64	2	S	D

ADDI

Add immediate value.

addi.i<size> #N, rD

add.f<size> #N, fD

subi.i<size> #N, rD

sub.f<size> #N, fD

The signed immediate stored in the extension word(s) is added to the value in D and the result stored in D. Where the operation size is less than the register size, the upper bits of the register are not affected.

Type	Opcode	Level	Variant / Operand	
i8	ADDI	0	0	D
i16	ADDI	0	1	D
I32 (16-bit range)	ADDI	0	2	D
I32 (32-bit range)	ADDI	0	3	D
f32 (32-bit range)	ADDI	1	4	D
I64 (16-bit range)	ADDI	2	5	D
I64 (32-bit range)	ADDI	2	6	D
I64 (64-bit range)	ADDI	2	7	D
f64 (32-bit range)	ADDI	2	8	D
f64 (64-bit range)	ADDI	2	9	D

Variant	Extension Word 1	
0	N	N
All other	N (host native 16-bit fragment)	

Variant	Extension Word 2 / 3 / 4
3, 5, 6, 7, 8, 9	N (host native 16-bit fragment)

Immediate values are always encoded into the least number of words that can represent the value without loss of precision or truncation. Immediate values are stored in host-native endian form.

Where the operation is 8 bit, the immediate is duplicated in both halves of the extension word allowing big and little endian host implementations to access without shifting.

There is no immediate subtract operation, this is achieved by adding a negative immediate.

ADDQ

Add small immediate value.

addq #N, rD

Add the small immediate value to D. All bits of the register are affected.

Opcode	Level	Operand	
ADDQ	0	N - 1	D

The immediate value is in the range 1-16.

AND_x

Bitwise logical AND.

and.<size> rS, rD

The value stored in S is logically ANDed with the value stored D and the result stored in the D. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit integer types are supported.

Size	Opcode	Level	Operand	
8	AND_8	0	S	D
16	AND_16	0	S	D
32	AND_32	0	S	D
64	AND_64	2	S	D

ANDI

Bitwise logical AND with immediate value.

and.<size> #N, rD

The immediate value stored in the extension word(s) is logically ANDed with the value in D and the result stored in D. Where the operation size is less than the register size, the upper bits of the register are not affected.

Size	**Opcode**	**Level**	**Variant / Operand**	
8	ANDI	0	0	D
16	ANDI	0	1	D
32 (16-bit range)	ANDI	0	2	D
32 (32-bit range)	ANDI	0	3	D
64 (16-bit range)	ANDI	2	4	D
64 (32-bit range)	ANDI	2	5	D
64 (64-bit range)	ANDI	2	6	D

Variant	**Extension Word 1**	
0	N	N
All other	N (host native 16-bit fragment)	

Variant	**Extension Word 2 / 3 / 4**
3,5,6	N (host native 16-bit fragment)

Immediate values are always encoded into the least number of words that can represent the value by omitting leading zeros. Immediate values are stored in host-native endian form.

Where the operation is 8 bit, the immediate is duplicated in both halves of the extension word allowing big and little endian host implementations to access without shifting.

ASR_x

Arithmetic shift-right, sign bits are preserved.

asr.i<size> rS, rD

The value stored in D is arithmetically right shifted by the value stored in S (modulo the operation size) and the result stored in D. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit signed integer types are supported.

Size	Opcode	Level	Operand	
i8	ASR_I8	0	S	D
i16	ASR_I16	0	S	D
i32	ASR_I32	0	S	D
i64	ASR_I64	2	S	D

BCALL_x

Program counter relative function call.

bcall <label | #displacement>

The address of the next instruction is pushed onto the return stack and the program counter is offset by the signed displacement. Execution then proceeds from the new program counter. On return, execution will proceed from the following instruction.

8 and 24-bit signed displacements are supported.

Displacement	Opcode	Level	Operand
8	BCALL_8	0	Displacement
24	BCALL_24	0	Displacement [7:0]

Displacement	Extension Word 1
24	Displacement [23:16]

When the displacement is 8-bit, the operand byte contains the signed displacement.

When the displacement is 24-bit, the most significant 16-bit portion of a 24-bit displacement is stored in the extension word as this can be sign expanded with branchless logic.

Offsets are measured in instruction words from the end of the entire instruction, e.g. a displacement of zero is the next instruction opcode.

If the call stack reaches the currently defined limit, execution halts and the machine status is set to CALL_STACK_OVERFLOW.

BCC

Compare two operands and branch on condition true.

b<cc>.<u|i><size> rS, rD, <label | displacement>

b<cc>.f<size> fS, fD, <label | displacement>

The condition to be evaluated is encoded into the upper byte of the first extension word. This also determines the operand type. Refer to the Conditional Comparison section for the appropriate enumeration.

8 and 24-bit signed displacements are supported.

Displacement	**Opcode**	**Level**	**Operand**	
8	BCC_8	0	S	D
24	BCC_24	0	S	D

Displacement	**Extension Word 1**	
8	CC	Displacement
24	CC	Displacement [7:0]

Displacement	**Extension Word 2**
24	Displacement [23:16]

Offsets are measured in instruction words from the end of the entire instruction, e.g. a displacement of zero is the next instruction opcode.

BNZ

Branch if value tests non-zero.

bnz.<size> rS, <label | #displacement>

bnz.f<size> fS, <label | #displacement>

Compare the values stored in S with all bits zero. If the value is not all bits zero, apply the 16-bit signed displacement to the program counter. Where the operation size is less than the register size, the upper bits of the register are not compared.

8, 16, 32 and 64-bit integer types are supported.
Floating point values will only test as zero in the case when all bits are zero.

Type	**Opcode**	**Level**	**Variant / Operand**	
8	BNZ	0	0	S
16	BNZ	0	1	S
32	BNZ	0	2	S
f32	BNZ	1	3	S
64	BNZ	2	4	S
f64	BNZ	2	5	S

Extension Word 1
Displacement

Offsets are measured in instruction words from the end of the entire instruction, e.g a displacement of zero is the next instruction opcode.

BRA_x

Jump to a new program counter location.

bra <label | #displacement>

The program counter is offset by the signed displacement. Execution then proceeds from the new program counter. 8 and 24-bit signed displacements are supported.

Displacement	Opcode	Level	Operand
8	BRA_8	0	Displacement
24	BRA_24	0	Displacement [7:0]

Displacement	Extension Word 1
24	Displacement [23:16]

When the displacement is 8-bit, the operand byte contains the signed displacement.

When the displacement is 24-bit, the most significant 16-bit portion of a 24-bit displacement is stored in the extension word as this can be sign expanded with branchless logic.

Offsets are measured in instruction words from the end of the entire instruction, e.g a displacement of zero is the next instruction opcode.

BRK

Halt at breakpoint.

brk #N

Execution of the VM halts at this instruction and the status register is set to BREAKPOINT. Intended to support interactive debugging tools.

Opcode	Level	Operand
BRK	0	N

Note that this should be opcode zero, such that any unintended branch to zero filled memory will be interpreted as a break #0.

BSWP_x

Byte swap (endian conversion).

bswp.<size> rS, rD

The value stored in S is byte-swapped and the result stored in the D. Where the operation size is less than the register size, the upper bits of the register are not affected.

16, 32 and 64-bit integer types are supported.

Type	Opcode	Level	Operand	
16	BSWP_16	0	S	D
32	BSWP_32	0	S	D
64	BSWP_64	2	S	D

CALL

Call a function.

call <@symbol | (rS)>
calln <@symbol | (rS)>

Direct and indirect calls to normal and host native functions are supported:

- For normal calls, the address of the next instruction is pushed onto the return stack.
- For native calls, interpretation suspends and execution resumes from the dereferenced native entry point.

If the call is indirect, the lower 20 bits of the source register are interpreted as the run-time resolved function enumeration to use. If the call is direct, the 20 bit symbol ID is constructed from the nybble and extension word. On return, execution resumes from the next instruction.

Variant	**Opcode**	**Level**	**Variant / Operand**	
Direct	CALL	0	0	Code Symbol ID [3:0]
Indirect	CALL	0	1	S
Native	CALL	0	2	Code Symbol ID [3:0]
Native Indirect	CALL	0	3	S

Variant	**Extension Word 1**
0, 2	Symbol ID [19:4]

If the Symbol ID is outside the range of those known to the VM, execution halts and the machine status is set to UNKNOWN_CODE_SYMBOL or UNKNOWN_NATIVE_CODE_SYMBOL accordingly.

CASE

Perform a register indexed jump to a new program counter position.

case rS

The unsigned 16-bit value in the source register is used to index a table of signed 16-bit offsets following the instruction. The corresponding offset is added to the program counter and execution resumes from that location.

Opcode	**Level**	**Operand**	
CASE	0	0	S

Extension Word 1
Table Size

Extension Word 2 …
First Displacement

Extension Word N
Last Displacement (always the default case)

Due to the potential length of the table, signed offsets are measured differently depending on the direction:

- Positive displacements are measured from the end of the table, i.e. an offset of zero refers to the next instruction immediately following the last extension word.
- Negative displacements are measured from the location of the case instruction, i.e an offset of -1 is one word before the case instruction.

If the value in the register exceeds the table size in the first extension word, the default offset stored in the last table entry is used.

CEIL_x

Ceiling function.

ceil.f<size> fS, fD

The smallest whole number not less than the value in S is calculated and stored in D.

32 and 64-bit floating point data types are supported.

Type	Opcode	Level	Operand	
f32	CEIL_F32	1	S	D
f64	CEIL_F64	2	S	D

CFB

Classify float and branch.

cfb.f<size> fS, <label | #displacement>

Classifies the floating point value in S and if the classification matches, the displacement is added to the program counter. Where the operation size is less than the register size, the upper bits of the register are not tested:

Type	Class	Opcode	Level	Variant / Operand	
f32	Zero	CFB	1	0	S
f32	Normal	CFB	1	1	S
f32	Subnormal	CFB	1	2	S
f32	Infinite	CFB	1	3	S
f32	NaN	CFB	1	4	S
f32	Custom	CFB	1	5	S
f64	Zero	CFB	2	6	S
f64	Normal	CFB	2	7	S
f64	Subnormal	CFB	2	8	S
f64	Infinite	CFB	2	9	S
f64	NaN	CFB	2	10	S
f64	Custom	CFB	2	11	S

Extension Word 1
Displacement

Offsets are measured in instruction words from the end of the entire instruction, e.g a displacement of zero is the next instruction opcode.

DBNZ

Decrement and Branch while non-zero.

dbnz rD, <label | #displacement>

Decrement the full width signed integer value in D. If the value after decrement is non-zero, the signed 16-bit displacement is added to the program counter.

Opcode	**Level**	**Operand**	
DBNZ	0	Reserved	D

Extension Word 1
Displacement

Offsets are measured in instruction words from the end of the entire instruction, e.g a displacement of zero is the next instruction opcode.

DBNN

Dereference and branch if not null.

dbnn #d(rD), <label | #displacement>

The address in D, plus a short 4-bit offset, d, is used to determine the location of a new address that is loaded into D. If the loaded address is not null, the signed 16-bit displacement in the extension word is added to the program counter. All bits of D are affected. The intention of this operation is to allow for simple traversal of linked lists.

Opcode	**Level**	**Operand**	
DBNN	0	d	D

Extension Word 1
Displacement

The short offset, d, is measured in 32-bit words as this is the minimum expected alignment.

Offsets are measured in instruction words from the end of the entire instruction, e.g a displacement of zero is the next instruction opcode.

IDIV

Integer arithmetic division.

div.<u|i><size> rS, rD [, rQ]

mod.<u|i><size> rS, rD [, rR]

divmod.<u|i><size> rS, rD [, rQ [, rR]]

Calculates the quotient, modulo or both, of the integer value in D divided by the value in S. Optional separate outputs can be specified for the quotient and remainder. The specific operation and output assignments are encoded into an extension word.

8, 16, 32 and 64-bit signed and unsigned integer types are supported.

Type	**Opcode**	**Level**	**Operand**	
all <= 32	IDIV	0	S	D
I64, u64	IDIV	2	S	D

Type	**Extension Word**			
i8	0, 1, 2	0	R, S	Q, D
i16	0, 1 ,2	1	R, S	Q, D
i32	0, 1, 2	2	R, S	Q, D
u8	0, 1, 2	3	R, S	Q, D
u16	0, 1, 2	4	R, S	Q, D
u32	0, 1, 2	5	R, S	Q, D
i64	0, 1, 2	6	R, S	Q, D
u64	0, 1, 2	7	R, S	Q, D
div	0	0-7	R, S	Q, D
mod	1	0-7	R, S	Q, D
divmod	2	0-7	R, S	Q, D

Unless otherwise indicated by the assembler, Q = D, R = S.

When the value in S is zero, execution halts and the machine status is set to ZERO_DIVIDE.

FDIV

Arithmetic division.

div.f<size> fS, fD

The value stored in the destination register is divided by the value stored in the source register and the result stored in the destination register. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit signed and unsigned integer types are supported.
32 and 64-bit floating point types are supported.

Type	Opcode	Level	Operand	
f32	DIV_F32	1	S	D
f64	DIV_F64	2	S	D

For integer types, when the value in the source register is zero, execution halts and the machine status is set to ZERO_DIVIDE.

DSA

Allocate storage on the data stack.

dsa #size, rD

The current data stack address is loaded into D and space for #size bytes of storage is reserved by incrementing the data stack position accordingly. The destination register now represents the base address of the stack allocated storage and must be preserved for any later call to DSF.

Opcode	**Level**	**Operand**	
DSA	0	0	D

Extension Word
#size

If the allocation would result in the data stack size being exceeded, execution halts and the status register is set to DATA_STACK_OVERFLOW.

DSF

Deallocate storage on the data stack.

dsf rS

The previous data stack address stored in the source register is used to restore the data stack pointer to the address it had before the DSA call was made.

Opcode	**Level**	**Operand**	
DSF	0	0	S

If the address in the source register is lower than the known data stack base address, execution halts and the status register is set to DATA_STACK_UNDERFLOW.

If the address in the source register is higher than the known data stack ceiling address, execution halts and the status register is set to DATA_STACK_OVERFLOW.

DSX

Transfer elements between the data stack.

push.<size> gpr_reg_mask

push.f<size> fpr_reg_mask

pop.<size> gpr_reg_mask

pop.f<size> gpr_reg_mask

For each register implied by a set bit in the extension word, transfer the element size to or from the data stack as indicated by the operand lower nybble in the direction indicated by the upper operand nybble.

8, 16, 32 and 64-bit sizes are supported.

Size	Opcode	Level	Operand	
8	DSX (push)	0	0	0
16	DSX (push)	0	0	1
32	DSX (push)	0	0	2
f32	DSX (push)	1	0	3
64	DSX (push)	2	0	4
f64	DSX (push)	2	0	5
8	DSX (pop)	0	1	0
16	DSX (pop)	0	1	1
32	DSX (pop)	0	1	2
f32	DSX (pop)	1	1	3
64	DSX (pop)	2	1	4
f64	DSX (pop)	2	1	5

Size	Extension Word
all	Register mask

Registers are loaded in descending order.

If the data stack is emptied, execution halts and the status register is set to DATA_STACK_UNDERFLOW.

If the data stack is exceeded, execution halts and the status register is set to DATA_STACK_OVERFLOW.

EXG_x

Exchange registers.

exg rS, rD

exg fS, fD

Exchange the contents of S and D. The full contents of the registers are exchanged.

Opcode	Level	Operand	
EXG_I	0	S	D
EXG_F	1	S	D

Registers can only be exchanged with other registers of the same type.

FLOOR_x

Floor function.

floor.f<size> fS, fD

The largest whole number not greater than the value in S is calculated and stored in D.

32 and 64-bit floating point data types are supported.

Type	Opcode	Level	Operand	
f32	FLOOR_F32	1	S	D
f64	FLOOR_F64	2	S	D

INV_x

Inversion.

inv.<size> rS, rD

For integer operands, the value stored in S is bitwise inverted and the resulting value stored in D. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit integer types are supported.

Type	Opcode	Level	Operand	
8	INV_8	0	S	D
16	INV_16	0	S	D
32	INV_32	0	S	D
64	INV_64	2	S	D

LDQ

Load a small immediate integer.

ldq #N, rD

Load the immediate small integer N (0-15) into D. All upper bits of the register are cleared.

Opcode	Level	Operand	
LDQ	0	N	D

The immediate value is interpreted as an unsigned value.

LD_x

Load global data.

ld.<size> @symbol, rD
ld.f<size> @symbol, fD

The 20-bit ID to which the data symbol was resolved at link time is read by combining the source operand nybble and 16-bit value in the extension word to give a 20 bit unsigned value.

The Symbol ID is then dereferenced by the VM to derive the host native address of the data to be loaded. The value at the address is then loaded into D. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit integer data types are supported.
32 and 64-bit floating point data types are supported.

Type	Opcode	Level	Operand	
8	LD_8	0	Symbol ID [3:0]	D
16	LD_16	0	Symbol ID [3:0]	D
32	LD_32	0	Symbol ID [3:0]	D
f32	LD_F32	1	Symbol ID [3:0]	D
64	LD_64	2	Symbol ID [3:0]	D
f64	LD_F64	2	Symbol ID [3:0]	D

Extension Word 1
Symbol ID [19:4]

If the Symbol ID is outside the range of those known to the VM, execution halts and the machine status is set to UNKNOWN_DATA_SYMBOL.

LD

Load data address

lda @symbol, rD

The 20-bit ID to which the data symbol was resolved at link time is read by combining the source operand nybble and 16-bit value in the extension word to give a 20 bit unsigned value. The Symbol ID is then dereferenced by the VM to derive the host native address of the data. If successful, this address is then stored in D. All bits of the register are affected.

Opcode	**Level**	**Operand**	
LD_DADR	0	Symbol ID [3:0]	D

Extension Word 1
Symbol ID [19:4]

If the Symbol ID is outside the range of those known to the VM, execution halts and the machine status is set to UNKNOWN_DATA_SYMBOL.

LD

Load callable symbol.

ldc @symbol, rD
ldn @symbol, rD

The 20-bit ID to which the callable symbol was resolved at link time is read by combining the source operand nybble and 16-bit value in the extension word to give a 20 bit unsigned value and loaded into the destination register. No dereferencing is performed. The intention of this operation is to allow the address of a function to be taken so that the function can be invoked indirectly via the CALL operation.

Variant	**Opcode**	**Level**	**Operand**	
Function	LD_CSYM	0	Symbol ID [3:0]	D
Native	LD_NSYM	0	Symbol ID [3:0]	D

Extension Word 1
Symbol ID [19:4]

If the Symbol ID is outside the range of those known to the VM, execution halts and the machine status is set to UNKNOWN_CODE_SYMBOL.

LDI

Load immediate value.

ld.i<size> #N, rD
ld.f<size> #N, fD

The 16-bit integer value in the extension word is loaded into D. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit integer types are supported.
32 and 64-bit floating point types are supported

Type	Opcode	Level	Variant / Operand	
i8	LDI	0	0	D
i16	LDI	0	1	D
I32 (16-bit range)	LDI	0	2	D
I32 (32-bit range)	LDI	0	3	D
f32 (32-bit range)	LDI	1	4	D
I64 (16-bit range)	LDI	2	5	D
I64 (32-bit range)	LDI	2	6	D
I64 (64-bit range)	LDI	2	7	D
f64 (32-bit range)	LDI	2	8	D
f64 (64-bit range)	LDI	2	9	D

Variant	Extension Word 1	
0	N	N
All other	N (host native 16-bit fragment)	

Variant	Extension Word 2 / 3 / 4
3,5,6,7,8,9	N (host native 16-bit fragment)

Immediate values are always encoded into the least number of words that can represent the value without loss of precision or truncation.

Immediate words are stored in host-native endian form.

LD_I_x

Load register indirect value.

ld.<size> (rS), rD

ld.<size>, #d(rS), rD

ld.f<size> (rS), fD

ld.f<size> #d(rS), fD

The value at the address in S, plus the optional signed 16-bit displacement in the extension word, is loaded into D. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit data types are supported.

Type	**Opcode**		**Operand**	
8	LD_I_8	0	S	D
16	LD_I_16	0	S	D
32	LD_I_32	0	S	D
f32	LD_I_F32	1	S	D
64	LD_I_64	2	S	D
f64	LD_I_F64	2	S	D

With displacements:

Type	**Opcode**	**0**	**Operand**	
8	LD_ID_8	0	S	D
16	LD_ID_16	0	S	D
32	LD_ID_32	0	S	D
f32	LD_ID_F32	1	S	D
64	LD_ID_64	2	S	D
f64	LD_ID_F64	2	S	D

Type	**Extension Word 1**
LD_ID_x	displacement

LD_IPD_x

Load register indirect value, pre-decremented.

ld.<size> -(rS), rD

ld.f<size> -(rS), fD

The address in the S is pre-decremented by the type's size. The value at the new address in S is loaded into the D register. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit data types are supported.

Type	Opcode	Level	Operand	
8	LD_IPD_8	0	S	D
16	LD_IPD_16	0	S	D
32	LD_IPD_32	0	S	D
f32	LD_IPD_F32	1	S	D
64	LD_IPD_64	2	S	D
f64	LD_IPD_F64	2	S	D

LD_IPI_x

Load register indirect value, post-incremented.

ld.<size> (rS)+, rD

ld.f<size> (rS)+, fD

The value at the address in the S into the D. The address in S is then incremented by the type's size. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit data types are supported.

Type	Opcode	Level	Operand	
8	LD_IPI_8	0	S	D
16	LD_IPI_16	0	S	D
32	LD_IPI_32	0	S	D
f32	LD_IPI_F32	1	S	D
64	LD_IPI_64	2	S	D
f64	LD_IPI_F64	2	S	D

LD_II

Load register indirect with index.

ld.<size> (rB, rI, #N), rD

ld.f<size> (rB, rI, #N), fD

The value at the effective address computed by taking the base address in B, offset by the 32-bit signed index value in I, scaled by shifting with integer value N, is loaded into D.

8, 16, 32 and 64-bit sizes are supported.

Type	Opcode	Level	Operand	
8	LD_II	0	0	0
16	LD_II	0	0	1
32	LD_II	0	0	2
f32	LD_II	1	0	3
64	LD_II	2	0	4
f64	LD_II	2	0	5

Type	Extension Word			
all	N	I	B	D

Scale value is applied as a shift, meaning the largest scale factor possible is 32768.

The operand size is not factored into the scaling, such that ld.8 (rB, rI, 0) and ld.64 (rB, rI, 0) will result in identical effective addresses.

LSL_x

Logical shift left, sign is not preserved.

lsl.<size> rS, rD

The value in D is shifted left by the value in the S, modulo the size of the type. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit integer types are supported.

Size	Opcode	Level	Operand	
8	LSL_8	0	S	D
16	LSL_16	0	S	D
32	LSL_32	0	S	D
64	LSL_64	2	S	D

LSLQ

Logical left shift bysmall immediate value.

lslq #N, rD

The value in D is left shifted by the small immediate. All bits of the register are affected.

Opcode	Level	Operand	
LSLQ	0	N - 1	D

The immediate is in the range 1-16.

LSR_x

Logical shift right, sign is not preserved.

lsr.<size> rS, rD

The value in D is shifted right by the value in S, modulo the size of the type. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit integer types are supported.

Size	Opcode	Level	Operand	
8	LSR_8	0	S	D
16	LSR_16	0	S	D
32	LSR_32	0	S	D
64	LSR_64	2	S	D

LSRQ

Logical right shift by small immediate value.

lsrq #N, rD

The value in D is right shifted by the small immediate. All bits of the register are affected.

Opcode	Level	Operand	
LSRQ	0	N - 1	D

The immediate is in the range 1-16.

MADD

Multiply add.

madd.<u|i><size> rA, rB, rC, rD

madd.f<size> fA, fB, fC, fD

The value in A, multiplied by the value in B and summed with the value in C is stored in D.
Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit signed and unsigned integer types are supported.
32 and 64-bit floating point types are supported.

Type	Opcode	Level	Operand	
u8	MADD	0	0	0
u16	MADD	0	0	1
u32	MADD	0	0	2
i8	MADD	0	0	3
i16	MADD	0	0	4
i32	MADD	0	0	5
f32	MADD	1	0	6
u64	MADD	2	0	7
i64	MADD	2	0	8
f64	MADD	2	0	9

Type	Operand Word			
all	A	B	C	D

FMOD_x

Arithmetic modulus.

mod.f<size> fS, fD

The value in D is divided by the value in S and the remainder stored in D. Where the operation size is less than the register size, the upper bits of the register are not affected.

32 and 64-bit floating point types are supported.

Type	Opcode	Level	Operand	
f32	MOD_F32	1	S	D
f64	MOD_F64	2	S	D

For integer types, when the value in the source register is zero, execution halts and the machine status is set to ZERO_DIVIDE.

For floating point types, the divisor can be any arbitrary value. The operation performed is equivalent to the fmod() standard library function.

MUL_x

Multiplication.

mul.<u|i><size> rS, rD

mul.f<size> fS, fD

The value in D is multiplied by the value in S and the result stored in the destination register. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit signed and unsigned integer types are supported.
32 and 64-bit floating point types are supported.

Type	Opcode	Level	Operand	
u8	MUL_U8	0	S	D
u16	MUL_U16	0	S	D
u32	MUL_U32	0	S	D
i8	MUL_I8	0	S	D
i16	MUL_I16	0	S	D
i32	MUL_I32	0	S	D
f32	MUL_F32	1	S	D
u64	MUL_U64	2	S	D
i64	MUL_I64	2	S	D
f64	MUL_F64	2	S	D

There are no versions of the instruction that produce a widened result. Where a widened result is required, the two source operands should first be widened.

MULI

Multiplication by immediate.

mul.i<size> #N, rD

mul.f<size> #N, fD

The immediate stored in the extension word(s) is multiplied by the value in D and the result stored in D. Where the operation size is less than the register size, the upper bits of the register are not affected.

Type	Opcode	Level	Variant / Operand	
u16	MULI	0	0	D
u32 (16-bit range)	MULI	0	1	D
u32 (32-bit range)	MULI	0	2	D
i16	MULI	0	3	D
i32 (16-bit range)	MULI	0	4	D
i32 (32-bit range)	MULI	0	5	D
f32 (32-bit range)	MULI	1	6	D
u64 (16-bit range)	MULI	2	7	D
u64 (32-bit range)	MULI	2	8	D
u64 (64-bit range)	MULI	2	9	D
i64 (16-bit range)	MULI	2	10	D
i64 (32-bit range)	MULI	2	11	D
i64 (64-bit range)	MULI	2	12	D
f64 (32-bit range)	MULI	2	13	D
f64 (64-bit range)	MULI	2	14	D

Variant	Extension Word 1/2/3/4
all	N (host native 16-bit fragment)

Immediate values are always encoded into the least number of words that can represent the value without loss of precision or truncation. Immediate values are stored in host-native endian form.

There are no immediate 8-bit forms of this operation.

MV_x

Move register to register.

mv.<size> rS, rD

mv.f<size> fS, fD

The value in S is copied to the D. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit integer types are supported.

Type	Opcode	Level	Operand	
8	MV_8	0	S	D
16	MV_16	0	S	D
32	MV_32	0	S	D
f32	MV_F32	1	S	D
64	MV_64	2	S	D
f64	MV_F64	2	S	D

MVCC

Compare two operands and move the third operand to the fourth when comparison is true.

mv<cc>.<size> rS, rD

mv<cc>.<size> rS, rD, rA, rB

The condition to be evaluated is encoded into the upper byte of the first extension word. This also determines the operand type. Refer to the Conditional section for the appropriate enumeration.

Opcode	Operand	
MVCC	S	D

NE_		
CC	A	B

Where the third and fourth operands are omitted, they are encoded as rS and rD respectively, implying the source is moved to the destination.

NEG_x

Negate value.

neg.i<size> rS, rD
neg.f<size> fS, fD

The value in S is negated and the result stored in D. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit signed integer types are supported.
32 and 64-bit floating point types are supported.

Type	Opcode	Level	Operand	
i8	NEG_I8	0	S	D
i16	NEG_I16	0	S	D
i32	NEG_I32	0	S	D
f32	NEG_F32	1	S	D
i64	NEG_I64	2	S	D
f64	NEG_F64	2	S	D

OR_x

Bitwise OR.

or.<size> rS, rD

The value in S is logically ORed with the value stored in D and the result stored in D. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit integer types are supported.

Type	Opcode	Level	Operand	
8	OR_8	0	S	D
16	OR_16	0	S	D
32	OR_32	0	S	D
64	OR_64	2	S	D

ORI

Bitwise OR with immediate value.

or.<size> #N, rD

The immediate stored in the extension word(s) is logically ORed with the value in D and the result stored in D. Where the operation size is less than the register size, the upper bits of the register are not affected.

Size	**Opcode**	**Level**	**Variant / Operand**	
8	ORI	0	0	D
16	ORI	0	1	D
32 (16-bit range)	ORI	0	2	D
32 (32-bit range)	ORI	0	3	D
64 (16-bit range)	ORI	2	4	D
64 (32-bit range)	ORI	2	5	D
64 (64-bit range)	ORI	2	6	D

Variant	**Extension Word 1**	
0	N	N
All other	N (host native 16-bit fragment)	

Variant	**Extension Word 2 / 3 / 4**
3,5,6	N (host native 16-bit fragment)

Immediate values are always encoded into the least number of words that can represent the value by omitting leading zeros.

Immediate values are stored in host-native endian form.

ORD_x

Order two values.

ord.i<size> rS, rD

ord.f<size> fS, fD

The value S is compared with the value in D and the larger of the two stored in D and the smaller is retained in S. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit signed integer types are supported.
32 and 64-bit floating point types are supported.

Type	Opcode	Level	Operand	
i8	ORD_I8	0	S	D
i16	ORD_I16	0	S	D
i32	ORD_I32	0	S	D
f32	ORD_F32	1	S	D
i64	ORD_I64	2	S	D
f64	ORD_F64	2	S	D

POW_x

Power function.

pow.f<size> fS, fD

The value in D is raised to the power of the value in S. Where the operation size is less than the register size, the upper bits of the register are not affected.

32 and 64-bit floating point types are supported.

Type	Opcode	Level	Operand	
f32	POW_F32	1	S	D
f64	POW_F64	2	S	D

RCP_x

Reciprocal.

rcp.f<size> fS, fD

inv.f<size> fS, fD

The reciprocal of the value in S is stored in D. Where the operation size is less than the register size, the upper bits of the register are not affected.

32 and 64-bit floating point types are supported.

Type	Opcode	Level	Operand	
f32	RCP_F32	1	S	D
f64	RCP_F64	2	S	D

RET

Return from function.

ret

ret #depth

The address of the next instruction is popped from the return stack. It is possible to return from an entire call tree up to a maximum depth of 256 by specifying the optional depth parameter.

Opcode	Level	Operand
RET	0	depth - 1

The operand holds one less than the return depth, i.e for normal return to immediate caller, the operand value is zero.

When the return stack is empty or would underflow, return from the entry point is assumed to have completed normally and the status register is set to COMPLETED.

RSX

Transfer entire registers to/from register stack.

restore.i gpr_reg_mask

restore.f fpr_reg_mask

restore gpr_reg_mask, fpr_reg_mask

save.i gpr_reg_mask

save.f fpr_reg_mask

save gpr_reg_mask, fpr_reg_mask

For each register implied by a set bit in the extension word(s), save or restore the data from the register stack as implied by the operand lower nybble.

Opcode	Level	Variant Operand	
RSX (gpr save)	0	0	0
RSX (gpr restore)	0	0	1
RSX (fpr save)	1	0	2
RSX (fpr restore)	1	0	3
RSX (gpr + fpr save)	1	0	4
RSX (gpr + fpr restore)	1	0	5

Variant	Extension Word 1
0, 1, 4, 5	GPR Register Mask
2, 3	FPR Register Mask

Variant	Extension Word 2
4, 5	FPR Register Mask

Registers are saved in ascending order.

If the register stack is emptied, execution halts and the status register is set to REGISTER_STACK_UNDERFLOW.

If the register stack is exceeded, execution halts and the status register is set to REGISTER_STACK_OVERFLOW.

ROL_x

Bitwise rotate left.

rol.<size> rS, rD

The value stored in D is bitwise rotated to the left by the value stored in S and the result stored in D. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit integer types are supported.

Size	Opcode	Level	Operand	
8	ROL_8	0	S	D
16	ROL_16	0	S	D
32	ROL_32	0	S	D
64	ROL_64	2	S	D

ROR_x

Bitwise rotate right.

ror.<size> rS, rD

The value stored in D is bitwise rotated to the right by the value stored in S and the result stored in D. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit integer types are supported.

Size	Opcode	Level	Operand	
8	ROR_8	0	S	D
16	ROR_16	0	S	D
32	ROR_32	0	S	D
64	ROR_64	2	S	D

SQRT_x

Square root.

sqrt.f<size> fS, fD

The positive square root of the floating point value in S is calculated and the result stored in D. Where the operation size is less than the register size, the upper bits of the register are not affected.

32 and 64-bit floating point data types are supported.

Type	Opcode	Level	Operand	
f32	SQRT_F32	1	S	D
f64	SQRT_F64	2	S	D

ST_x

Store global data.

st.<size> rS, @symbol
st.f<size> fS, @symbol

The 20-bit ID to which the data symbol was resolved at link time is read by combining the source operand nybble and 16-bit value in the extension word to give a 20 bit unsigned value.

The Symbol ID is then dereferenced by the VM to derive the host native address of the data to be written. The value in S at the address is then written to the address.

8, 16, 32 and 64-bit integer data types are supported.
32 and 64-bit floating point data types are supported.

Type	Opcode	Level	Operand	
8	ST_8	0	Symbol ID [3:0]	D
16	ST_16	0	Symbol ID [3:0]	D
32	ST_32	0	Symbol ID [3:0]	D
f32	ST_F32	1	Symbol ID [3:0]	D
64	ST_64	2	Symbol ID [3:0]	D
f64	ST_F64	2	Symbol ID [3:0]	D

Extension Word 1
Symbol ID [19:4]

If the Symbol ID is outside the range of those known to the VM, execution halts and the machine status is set to UNKNOWN_DATA_SYMBOL.

ST_I_x

Store register indirect value.

st.<size> rS, (rD)

st.<size>, rS, #d(rD)

st.f<size> fS, (rD)

st.f<size> fS, #d(rD)

The value in S is saved to the address in D, plus the optional signed 16-bit displacement in the extension word.

8, 16, 32 and 64-bit data types are supported.

Type	Opcode	Level	Operand	
8	ST_I_8	0	S	D
16	ST_I_16	0	S	D
32	ST_I_32	0	S	D
f32	ST_I_F32	1	S	D
64	ST_I_64	2	S	D
f64	ST_I_F64	2	S	D

With displacements:

Type	Opcode	Level	Operand	
8	ST_ID_8	0	S	D
16	ST_ID_16	0	S	D
32	ST_ID_32	0	S	D
f32	ST_ID_F32	1	S	D
64	ST_ID_64	2	S	D
f64	ST_ID_F64	2	S	D

Type	Extension Word 1
ST_ID_x	displacement

ST_IPD_x

Store register indirect value, pre-decremented.

st.<size> rS, -(rD)

st.f<size> fS, -(rD)

The value in S is written to the address in the D, pre-decremented by the type's size.

8, 16, 32 and 64-bit data types are supported.

Type	Opcode	Level	Operand	
8	ST_IPD_8	0	S	D
16	ST_IPD_16	0	S	D
32	ST_IPD_32	0	S	D
f32	ST_IPD_F32	1	S	D
64	ST_IPD_64	2	S	D
f64	ST_IPD_F64	2	S	D

ST_IPI_x

Store register indirect value, post-incremented.

st.<size> rS, (rD)+,

st.f<size> fS, (rD)+

The value in S is stored at the address in D. The address in D is then incremented by the type's size.

8, 16, 32 and 64-bit data types are supported.

Type	Opcode	Level	Operand	
8	ST_IPI_8	0	S	D
16	ST_IPI_16	0	S	D
32	ST_IPI_32	0	S	D
f32	ST_IPI_F32	1	S	D
64	ST_IPI_64	2	S	D
f64	ST_IPI_F64	2	S	D

ST_II

Store register indirect with index.

st.<size> rS, (rB, rI, #N)

st.f<size> fS, (rB, rI, #N)

The value in S is written to the effective address computed by taking the base address in B, offset by the 32-bit signed index value in I, scaled by shifting with integer value N.

8, 16, 32 and 64-bit sizes are supported.

Size	Opcode	Level	Operand	
8	ST_II	0	0	0
16	ST_II	0	0	1
32	ST_II	0	0	2
f32	ST_II	1	0	3
64	ST_II	2	0	4
f64	ST_II	2	0	5

Type	Operand Word			
all	#N	I	B	S

Scale value is applied as a shift, meaning the largest scale factor possible is 32768.

The operand size is not factored into the scaling, such that ld.8 (rB, rI, 0) and ld.64 (rB, rI, 0) will result in identical effective addresses.

SUB_x

Subtraction.

sub.i<size> rS, rD

sub.f<size> fS, fD

The value stored in S is subtracted from the value stored in D and the result stored in D. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit integer types are supported.
32 and 64-bit floating point types are supported.

Type	Opcode	Level	Operand	
i8	SUB_I8	0	S	D
i16	SUB_I16	0	S	D
i32	SUB_I32	0	S	D
f32	SUB_F32	1	S	D
i64	SUB_I64	2	S	D
f64	SUB_F64	2	S	D

SUBQ

Subtract small immediate value.

subq #N, rD

Subtract the small immediate value from D. All bits of the register are affected.

Opcode	Level	Operand	
SUBQ	0	N - 1	D

The immediate is in the range 1-16.

XOR_x

Bitwise Exclusive OR.

xor.<size> rS, rD

The value stored in S is logically XORed with the value stored in D and the result stored in D. Where the operation size is less than the register size, the upper bits of the register are not affected.

8, 16, 32 and 64-bit integer types are supported.

Size	Opcode	Level	Operand	
8	XOR_8	0	S	D
16	XOR_16	0	S	D
32	XOR_32	0	S	D
64	XOR_64	2	S	D

XORI

Bitwise Exclusive OR immediate value.

xor.<size> #N, rD

The immediate stored in the extension word(s) is logically XORed with the value in D and the result stored in D. Where the operation size is less than the register size, the upper bits of the register are not affected.

Size	Opcode	Level	Variant / Operand	
8	XORI	0	0	D
16	XORI	0	1	D
32 (16-bit range)	XORI	0	2	D
32 (32-bit range)	XORI	0	3	D
64 (16-bit range)	XORI	2	4	D
64 (32-bit range)	XORI	2	5	D
64 (64-bit range)	XORI	2	6	D

Variant	Extension Word 1	
0	N	N
All other	N (host native 16-bit fragment)	

Variant	Extension Word 2 / 3 / 4
3,5,6	N (host native 16-bit fragment)

Immediate values are always encoded into the least number of words that can represent the value by omitting leading zeros.

Immediate values are stored in host-native endian form.

x_2_F32

Convert varying types to 32-bit floating point.

tof32.i<size> rS, fD

tof32.u<size> rS, fD

tof32.f64 fS, fD

Convert the typed value in S to a 32-bit floating point value in D.

Type	Opcode	Level	Operand	
u8	U8_2_F32	1	S	D
u16	U16_2_F32	1	S	D
u32	U32_2_F32	1	S	D
i8	I8_2_F32	1	S	D
i16	I16_2_F32	1	S	D
i32	I32_2_F32	1	S	D
u64	U64_2_F32	2	S	D
i64	I64_2_F32	2	S	D
f64	F64_2_F32	2	S	D

Conversion of 32 and 64-bit integer values to floating point may result in loss of precision.

x_2_F64

Convert varying types to 64-bit floating point.

tof64.i<size> rS, fD

tof64.u<size> rS, fD

tof64.f32 fS, fD

Convert the typed value in S to a 64-bit floating point value in D.

Type	Opcode	Level	Operand	
u8	U8_2_F64	2	S	D
u16	U16_2_F64	2	S	D
u32	U32_2_F64	2	S	D
u64	U64_2_F64	2	S	D
i8	I8_2_F64	2	S	D
i16	I16_2_F64	2	S	D
i32	I32_2_F64	2	S	D
i64	I64_2_F64	2	S	D
f32	F32_2_F64	2	S	D

Conversion of 64-bit integer values to floating point may result in loss of precision.

x_2_I8

Convert varying types to signed 8-bit integer.

toi8.f<size> fS, rD

Convert floating point type to signed 8-bit integer.

Type	Opcode	Level	Operand	
f32	F32_2_I8	1	S	D
f64	F64_2_I8	2	S	D

Conversion of floating point values within the destination range round towards zero, e.g.

Source	Destination
-2.99	-2
-0.75	0
0.75	0
1.25	1
1.75	1
2.99	2

Conversion of floating point values outside the destination range are truncated.

x_2_I16

Convert varying types to signed 16-bit integer.

toi16.i<size> rS, rD

toi16.f<size> fS, rD

Convert type to signed 16-bit integer.

Type	Opcode	Level	Operand	
i8	I8_2_I16	0	S	D
f32	F32_2_I16	1	S	D
f64	F64_2_I16	2	S	S

Conversion of floating point values within the destination range round towards zero, e.g.

Source	Destination
-2.99	-2
-0.75	0
0.75	0
1.25	1
1.75	1
2.99	2

Conversion of floating point values outside the destination range are truncated.

x_2_I32

Convert varying types to signed 32-bit integer.

toi32.i<size> rS, rD

toi32.f<size> fS, rD

Convert type to signed 32-bit integer.

Type	Opcode	Level	Operand	
i8	I8_2_I32	0	S	D
i16	I16_2_I32	0	S	D
f32	F32_2_I32	1	S	S
f64	F64_2_I32	2	S	D

Conversion of floating point values within the destination range round towards zero, e.g.

Source	Destination
-2.99	-2
-0.75	0
0.75	0
1.25	1
1.75	1
2.99	2

Conversion of floating point values outside the destination range are truncated.

x_2_I64

Convert varying types to signed 64-bit integer.

toi64.i<size> rS, rD

toi64.f<size> fS, rD

Convert type to signed 64-bit integer.

Type	Opcode	Level	Operand	
i8	I8_2_I64	2	S	D
i16	I16_2_I64	2	S	D
i32	I32_2_I64	2	S	S
f32	F32_2_I64	2	S	D
f64	F64_2_I64	2	S	D

Conversion of floating point values within the destination range round towards zero, e.g.

Source	Destination
-2.99	-2
-0.75	0
0.75	0
1.25	1
1.75	1
2.99	2

Conversion of floating point values outside the destination range are truncated.