Base Instruction Set

The base instruction set perform operations on the 16 general purpose registers within the ExVM machine. No memory direct operations, other than load and store are implemented.

OPERATION

Summary description of the operation.

generalised syntactical form

Detailed description of the operation.

List of types or variants supported.

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

Type	Extension Word
Applicable type	Extension word interpretation

Additional notes, where relevant.

Inline C Macros
Macro forms for statically declaring VM code inside C source.

ADD

Addition.

add.type rS, rD

The value stored in in the source register is added to the value stored in the destination 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 integer types are supported.
32 and 64-bit floating point types are supported.

Type	**Opcode**	**Operand**	
i8	ADD_I8	S	D
i16	ADD_I16	S	D
i32	ADD_I32	S	D
i64	ADD_I64	S	D
f32	ADD_F32	S	D
f64	ADD_F64	S	D

Inline C Macros

```
_add_8(src reg, dst reg)
_add_16(src reg, dst reg)
_add_32(src reg, dst reg)
_add_64(src reg, dst reg)
_add_f32(src reg, dst reg)
_add_f64(src reg, dst reg)
```

ADDI

Add integer immediate.

addi.type #N, rD

The integer immediate stored in the extension word(s) is added to the value in the destination 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 and 32-bit integer immediates are supported.

Type	**Opcode**	**Operand**	
i8	ADDI_I8	0x0	D
i16	ADDI_I16	0x0	D
i32	ADDI_I32	0x0	D

Type	**Extension Word 1**	
i8	ignored	N
i16	N	
i32	N (host native half)	

Type	**Extension Word 2**
i32	N (host native half)

Inline C Macros

    _addi_8(int literal, dst reg)
    _addi_16(int literal, dst reg)
    _addi_32(int literal, dst reg)

AND

Bitwise AND.

and.type rS, rD

The value stored in in the source register is logically ANDed with the value stored in the destination 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 integer types are supported.

Type	**Opcode**	**Operand**	
i8	AND_8	S	D
i16	AND_16	S	D
i32	AND_32	S	D
i64	AND_64	S	D

Inline C Macros

```
_and_8(src reg, dst reg)
_and_16(src reg, dst reg)
_and_32(src reg, dst reg)
_and_64(src reg, dst reg)
```

ASR

Arithmetic shift right, sign bits are preserved.

asr.type rS, rD

The value stored in in the destination register is arithmetically right shifted by the value stored in the source register (modulo the operation size) 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 integer types are supported.

Type	**Opcode**	**Operand**	
s8	ASR_S8	S	D
s16	ASR_S16	S	D
s32	ASR_S32	S	D
s64	ASR_S64	S	D

Inline C Macros

```
_asr_s8(src reg, dst reg)
_asr_s16(src reg, dst reg)
_asr_s32(src reg, dst reg)
_asr_s64(src reg, dst reg)
```

BCALL

Program counter relative function call.

bcall @location
bcall #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 16-bit displacements are supported. When the displacement is 8-bit, the operand byte contains the signed displacement.

Offsets are measured in instruction words from the location of the call instruction.

Displacement	**Opcode**	**Operand**
8	BCALL_8	Displacement
16	BCALL_16	ignored

Displacement	**Extension Word 1**
16	Displacement

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

Inline C Macros

_bcall_8_unresolved(dummy symbol)
_bcall_8(symbol)
_bcall_unresolved(dummy symbol)
_bcall(symbol)

BEQ

Branch if values compare equal.

beq.type rS, rD, @location
beq.type rS, rD, #displacement

Compare the values in rS and rD. If the values are equal, 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.

Offsets are measured in instruction words from the location of the call instruction.

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

Type	**Opcode**	**Operand**	
i8	BEQ_8	S	D
i16	BEQ_16	S	D
i32	BEQ_32	S	D
i64	BEQ_64	S	D
f32	BEQ_F32	S	D
f64	BEQ_F64	S	D

Extension Word 1
Displacement

Inline C Macros

_beq_8(src reg, dst reg, displacement)
_beq_16(src reg, dst reg, displacement)
_beq_32(src reg, dst reg, displacement)
_beq_64(src reg, dst reg, displacement)
_beq_f32(src reg, dst reg, displacement)
_beq_f64(src reg, dst reg, displacement)

BGREQ

Branch if values compare greater or equal.

bgreq.type rS, rD, @location
bgreq.type rS, rD, #displacement

Compare the values in rS and rD. If the values are equal, 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.

Offsets are measured in instruction words from the location of the call instruction.

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

Type	**Opcode**	**Operand**	
i8	BGREQ_8	S	D
i16	BGREQ_16	S	D
i32	BGREQ_32	S	D
i64	BGREQ_64	S	D
f32	BGREQ_F32	S	D
f53	BGREQ_F64	S	D

Extension Word 1
Displacement

There is no corresponding “branch if less than or equal” opcode. These are modelled by performing BGREQ with the operands inverted.

Inline C Macros

_bgreq_8(src reg, dst reg, displacement)
_bgreq_16(src reg, dst reg, displacement)
_bgreq_32(src reg, dst reg, displacement)
_bgreq_64(src reg, dst reg, displacement)
_bgreq_f32(src reg, dst reg, displacement)
_bgreq_f64(src reg, dst reg, displacement)
_blseq_8(src reg, dst reg, displacement)
_blseq_16(src reg, dst reg, displacement)
_blseq_32(src reg, dst reg, displacement)
_blseq_64(src reg, dst reg, displacement)
_blseq_f32(src reg, dst reg, displacement)
_blseq_f64(src reg, dst reg, displacement)

BGR

Branch if value compares greater than.

bgr.type rS, rD, @location
bgr.type rS, rD, #displacement

Compare the values in rS and rD. If the value in rS is greater than that in rD, 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.

Offsets are measured in instruction words from the location of the call instruction.

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

Type	**Opcode**	**Operand**	
i8	BGR_8	S	D
i16	BGR_16	S	D
i32	BGR_32	S	D
i64	BGR_64	S	D
f32	BGR_F32	S	D
f53	BGR_F64	S	D

Extension Word 1
Displacement

There is no corresponding “branch if less than” opcode. These are modelled by performing BGR with the operands inverted.

Inline C Macros

_bgr_8(src reg, dst reg, displacement)
_bgr_16(src reg, dst reg, displacement)
_bgr_32(src reg, dst reg, displacement)
_bgr_64(src reg, dst reg, displacement)
_bgr_f32(src reg, dst reg, displacement)
_bgr_f64(src reg, dst reg, displacement)
_bls_8(src reg, dst reg, displacement)
_bls_16(src reg, dst reg, displacement)
_bls_32(src reg, dst reg, displacement)
_bls_64(src reg, dst reg, displacement)
_bls_f32(src reg, dst reg, displacement)
_bls_f64(src reg, dst reg, displacement)

BNZ

Branch if value tests non-zero.

bnz.type rS, @location
bnz.type rS, #displacement

Compare the values in rS 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.

Offsets are measured in instruction words from the location of the call instruction.

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	Operand	
i8	BNZ_8	0x0	S
i16	BNZ_16	0x0	S
i32	BNZ_32	0x0	S
i64	BNZ_64	0x0	S

Extension Word 1
Displacement

Inline C Macros

_bnz_8(src reg, displacement)
_bnz_16(src reg, displacement)
_bnz_32(src reg, displacement)
_bnz_64(src reg, displacement)

BRA

Jump to new program counter location.

bra @location
bra #displacement

The program counter is offset by the signed displacement. Execution then proceeds from the new program counter.

8 and 16-bit displacements are supported. When the displacement is 8-bit, the operand byte contains the signed displacement.

Offsets are measured in instruction words from the location of the call instruction.

Displacement	**Opcode**	**Operand**
8	BRA_8	Displacement
16	BRA_16	ignored

Displacement	**Extension Word 1**
16	Displacement

Inline C Macros

    _bra_8(displacement)
    _bra(displacement)

BRK

Halt at breakpoint

brk

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

Opcode	Operand
BRK	ignored

Inline C Macros

_brk

BSWP

Byte swap (endian conversion).

bswp.type rS, rD

The value stored in in the source register is byte-swapped 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.

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

Type	**Opcode**	**Operand**	
i16	BSWP_16	S	D
i32	BSWP_32	S	D
i64	BSWP_64	S	D

Inline C Macros

```
_bswap_16(src reg, dst reg)
_bswap_32(src reg, dst reg)
_bswap_64(src reg, dst reg)
```

CALL

Call a function.

call @symbol

The address of the next instruction is pushed onto the return stack. The 20-bit ID to which the code symbol was resolved at link time is read by combining the lower nybble of the operand byte 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 new program counter address. Execution then resumes from this location. On return, execution will proceed from the following instruction.

Opcode	**Operand**	
CALL	0x0	Code Symbol ID [19:16]

Extension Word 1
Code Symbol ID [15:0]

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.

Inline C Macros

_call_unresolved(dummy symbol)
_call(symbol)

CALLN

Call a host-native function.

calln @symbol

The address of the next instruction is pushed onto the return stack. The 20-bit ID to which the host-native code symbol was resolved at link time is read by combining the lower nybble of the operand byte 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 entry point for the host native function. Execution of virtual code stops and the host native function is invoked. The host native function is passed a pointer to the current Interpreter instance and is able to read and write the GPR.

Assuming the native call returns, on return, execution will proceed from the following instruction.

Opcode	**Operand**	
CALLN	0x0	Native Symbol ID [19:16]

Extension Word 1
Native Symbol ID [15:0]

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

Inline C Macros

_calln_unresolved(dummy symbol)
_calln(symbol)

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	**Operand**	
CASE	0x0	S

Extension Word 1
Table Size

Extension Word 2
First Displacement

Extension Word X
Last Displacement (always the default case)

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

Inline C Macros

_case(src reg, table entries)

DIV

Division.

div.type rS, rD

The value stored in 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**	**Operand**	
u8	DIV_U8	S	D
u16	DIV_U16	S	D
u32	DIV_U32	S	D
u64	DIV_U64	S	D
s8	DIV_S8	S	D
s16	DIV_S16	S	D
s32	DIV_S32	S	D
s64	DIV_S64	S	D
f32	DIV_F32	S	D
f64	DIV_F64	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.

Inline C Macros

```
_div_u8(src reg, dst reg)
_div_u16(src reg, dst reg)
_div_u32(src reg, dst reg)
_div_u64(src reg, dst reg)
_div_s8(src reg, dst reg)
_div_s16(src reg, dst reg)
_div_s32(src reg, dst reg)
_div_s64(src reg, dst reg)
_div_f32(src reg, dst reg)
_div_f64(src reg, dst reg)
```

EXG

Exchange registers.

exg rS, rD

Exchange the contents of the source and destination registers. The full contents of the registers are exchanged.

Opcode	**Operand**	
EXG	S	D

Inline C Macros

_exg(src reg, dst reg)

ICALL

Call a function indirectly.

call (rX)

The address of the next instruction is pushed onto the return stack. The value in rX is interpreted as the the 20-bit ID to which the code symbol was resolved at link time.

The Symbol ID is then dereferenced by the VM to derive the new program counter address. Execution then resumes from this location. On return, execution will proceed from the following instruction.

Opcode	**Operand**	
ICALL	0x0	X

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.

Inline C Macros

_icall(dst reg)

ICALLN

Call a host-native function indirectly.

calln (rX)

The address of the next instruction is pushed onto the return stack. The value in rX is interpreted as the the 20-bit ID to which the host-native code symbol was resolved at link time.

The Symbol ID is then dereferenced by the VM to derive the entry point for the host native function. Execution of virtual code stops and the host native function is invoked. The host native function is passed a pointer to the current Interpreter instance and is able to read and write the GPR.

Assuming the native call returns, on return, execution will proceed from the following instruction.

Opcode	**Operand**	
ICALLN	0x0	X

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

Inline C Macros

_icalln(dst reg)

INV

Bitwise inversion.

inv.type rS, rD

The value stored in in the source register is bitwise inverted and the resulting value 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 integer types are supported.

Type	**Opcode**	**Operand**	
i8	INV_8	S	D
i16	INV_16	S	D
i32	INV_32	S	D
i64	INV_64	S	D

Inline C Macros

```
_inv_8(src reg, dst reg)
_inv_16(src reg, dst reg)
_inv_32(src reg, dst reg)
_inv_64(src reg, dst reg)
```

LDQ

Load immediate

ldq #N, rD

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

Opcode	**Operand**	
LDQ	N	D

Inline C Macros

_ldq(int literal, dst reg)

LD

Load global data.

ld.type @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 to be loaded. The value at the address is then loaded into 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 data types are supported.

Type	**Opcode**	**Operand**	
8	LD_8	Symbol ID [19:16]	D
16	LD_16	Symbol ID [19:16]	D
32	LD_32	Symbol ID [19:16]	D
64	LD_64	Symbol ID [19:16]	D

Extension Word 1
Symbol ID [15:0]

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.

Inline C Macros

_ld_8_unresolved(dummy symbol, dst reg)
_ld_16_unresolved(dummy symbol, dst reg)
_ld_32_unresolved(dummy symbol, dst reg)
_ld_64_unresolved(dummy symbol, dst reg)
_ld_8(symbol, dst reg)
_ld_16(symbol, dst reg)
_ld_32(symbol, dst reg)
_ld_64(symbol, dst reg)

LD_ADDR

Load global 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 to be loaded. This address is then stored in the destination register. All bits of the register are affected.

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

Opcode	**Operand**	
LD_8	Symbol ID [19:16]	D

Extension Word 1
Symbol ID [15:0]

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.

Inline C Macros

_lda_unresolved(dummy symbol, dst reg)
_lda(symbol, dst reg)

LD_CSYM

Load callable function address.

ldc @symbol, rD

The 20-bit ID to which the code 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 function to be loaded. This address is then stored in the destination register. All bits of the register are affected.

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 ICALL operation.

Opcode	**Operand**	
LD_CSYM	Symbol ID [19:16]	D

Extension Word 1
Symbol ID [15:0]

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.

Inline C Macros

_ldc_unresolved(dummy symbol, dst reg)
_ldc(symbol, dst reg)

LD_NSYM

Load host native callable function address.

ldn @symbol, rD

The 20-bit ID to which the host native code 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 function to be loaded. This address is then stored in the destination register. All bits of the register are affected.

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 ICALLN operation.

Opcode	**Operand**	
LD_NSYM	Symbol ID [19:16]	D

Extension Word 1
Symbol ID [15:0]

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.

Inline C Macros

_ldn_unresolved(dummy symbol, dst reg)
_ldn(symbol, dst reg)

LD_I16

Load immediate 16-bit integer.

ld.type #N, rD

The 16-bit integer value in the extension word is loaded into 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 integer types are supported.

Type	**Opcode**	**Operand**	
i8	LD_I16_8	0x0	D
i16	LD_I16_16	0x0	D
i32	LD_I16_32	0x0	D
i64	LD_I16_64	0x0	D

Type	**Extension Word 1**	
i8	ignored	N
I16, i32, i64	N	

Inline C Macros

```
_ld_16_i8(src reg, dst reg)
_ld_16_i16(src reg, dst reg)
_ld_16_i32(src reg, dst reg)
_ld_16_i64(src reg, dst reg)
```

LD_I32

Load immediate 32-bit value.

ld.type #N, rD

The 32-bit value in the extension word is loaded into 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 integer types are supported.
32-bit floating point types are also supported

Type	**Opcode**	**Operand**	
I32, f32	LD_I32_32	0x0	D
i64	LD_I32_64	0x0	D

Type	**Extension Word 1**
All	N (host native half)

Type	**Extension Word 2**
All	N (host native half)

Inline C Macros

_ld_32_i32(src reg, dst reg)
_ld_32_i64(src reg, dst reg)
_ld_32_f32(src reg, dst reg)
_ld_32_f64(src reg, dst reg)

LD_RID

Load register indirect value, with displacement.

ld.type #d(rS), rD

The address in the source register is offset by the signed 16-bit integer displacement in the extension word. The value at the resulting address is loaded into 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 data types are supported.

Type	**Opcode**	**Operand**	
i8	LD_RID_8	S	D
i16	LD_RID_16	S	D
I32, f32	LD_RID_32	S	D
I64, f64	LD_RID_64	S	D

Type	**Extension Word 1**
All	#d

LD_RI

Load register indirect value.

ld.type (rS), rD

The value at the address indicated by the source register is loaded into 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 data types are supported.

Type	**Opcode**	**Operand**	
i8	LD_RI_8	S	D
i16	LD_RI_16	S	D
I32, f32	LD_RI_32	S	D
I64, f64	LD_RI_64	S	D

Inline C Macros

_ld_ri_8(src reg, dst reg)
_ld_ri_16(src reg, dst reg)
_ld_ri_32(src reg, dst reg)
_ld_ri_64(src reg, dst reg)

LD_RIPD

Load register indirect value, pre-decremented.

ld.type -(rS), rD

The address in the source register is pre-decremented by the type’s size. The value at the new address in the source register is loaded into 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 data types are supported.

Type	**Opcode**	**Operand**	
i8	LD_RIPD_8	S	D
i16	LD_RIPD_16	S	D
I32, f32	LD_RIPD_32	S	D
I64, f64	LD_RIPD_64	S	D

Inline C Macros

_ld_ripd_8(src reg, dst reg)
_ld_ripd_16(src reg, dst reg)
_ld_ripd_32(src reg, dst reg)
_ld_ripd_64(src reg, dst reg)

LD_RIPI

Load register indirect value, post-incremented.

ld.type (rS)+, rD

The value at the address in the source register is loaded into the destination register. The address in the source register 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	Operand	
i8	LD_RIPI_8	S	D
i16	LD_RIPI_16	S	D
I32, f32	LD_RIPI_32	S	D
I64, f64	LD_RIPI_64	S	D

Inline C Macros

_ld_ripi_8(src reg, dst reg)
_ld_ripi_16(src reg, dst reg)
_ld_ripi_32(src reg, dst reg)
_ld_ripi_64(src reg, dst reg)

LSL

Logical shift left, sign is not preserved.

lsl.type rS, rD

The value in the destination register is shifted left by the value in the source register, 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.

Type	**Opcode**	**Operand**	
i8	LSL_8	S	D
i16	LSL_16	S	D
i32	LSL_32	S	D
i64	LSL_64	S	D

Inline C Macros

```
_lsl_8(src reg, dst reg)
_lsl_16(src reg, dst reg)
_lsl_32(src reg, dst reg)
_lsl_64(src reg, dst reg)
```

LSR

Logical shift right, sign is not preserved.

lsr.type rS, rD

The value in the destination register is shifted right by the value in the source register, 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.

Type	**Opcode**	**Operand**	
i8	LSR_8	S	D
i16	LSR_16	S	D
i32	LSR_32	S	D
i64	LSR_64	S	D

Inline C Macros

```
_lsr_8(src reg, dst reg)
_lsr_16(src reg, dst reg)
_lsr_32(src reg, dst reg)
_lsr_64(src reg, dst reg)
```

MAX

Select maximum value.

max.type rS, rD

The value stored in in the source register is compared with the value stored in the destination register and the larger of the two 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 integer types are supported.
32 and 64-bit floating point types are supported.

Type	**Opcode**	**Operand**	
s8	MAX_S8	S	D
s16	MAX_S16	S	D
s32	MAX_S32	S	D
s64	MAX_S64	S	D
f32	MAX_F32	S	D
f64	MAX_F64	S	D

Inline C Macros

    _max_s8(src reg, dst reg)
    _max_s16(src reg, dst reg)
    _max_s32(src reg, dst reg)
    _max_s64(src reg, dst reg)
    _max_f32(src reg, dst reg)
    _max_f64(src reg, dst reg)

MIN

Select minimum value.

min.type rS, rD

The value stored in in the source register is compared with the value stored in the destination register and the smaller of the two 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 integer types are supported.
32 and 64-bit floating point types are supported.

Type	**Opcode**	**Operand**	
s8	MIN_S8	S	D
s16	MIN_S16	S	D
s32	MIN_S32	S	D
s64	MIN_S64	S	D
f32	MIN_F32	S	D
f64	MIN_F64	S	D

Inline C Macros

    _min_s8(src reg, dst reg)
    _min_s16(src reg, dst reg)
    _min_s32(src reg, dst reg)
    _min_s64(src reg, dst reg)
    _min_f32(src reg, dst reg)
    _min_f64(src reg, dst reg)

MOD

Modulus.

mod.type rS, rD

The value stored in in the destination register is divided by the value stored in the source register and the remainder 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**	**Operand**	
u8	MOD_U8	S	D
u16	MOD_U16	S	D
u32	MOD_U32	S	D
u64	MOD_U64	S	D
s8	MOD_S8	S	D
s16	MOD_S16	S	D
s32	MOD_S32	S	D
s64	MOD_S64	S	D
f32	MOD_F32	S	D
f64	MOD_F64	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.

Inline C Macros

```
_mod_u8(src reg, dst reg)
_mod_u16(src reg, dst reg)
_mod_u32(src reg, dst reg)
_mod_u64(src reg, dst reg)
_mod_s8(src reg, dst reg)
_mod_s16(src reg, dst reg)
_mod_s32(src reg, dst reg)
_mod_s64(src reg, dst reg)
_mod_f32(src reg, dst reg)
_mod_f64(src reg, dst reg)
```

MUL

Multiplication.

mul.type rS, rD

The value stored in in the destination register is multiplied 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	Operand	
u8	MUL_U8	S	D
u16	MUL_U16	S	D
u32	MUL_U32	S	D
u64	MUL_U64	S	D
s8	MUL_S8	S	D
s16	MUL_S16	S	D
s32	MUL_S32	S	D
s64	MUL_S64	S	D
f32	MUL_F32	S	D
f64	MUL_F64	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.

Inline C Macros

```
_mul_u8(src reg, dst reg)
_mul_u16(src reg, dst reg)
_mul_u32(src reg, dst reg)
_mul_u64(src reg, dst reg)
_mul_s8(src reg, dst reg)
_mul_s16(src reg, dst reg)
_mul_s32(src reg, dst reg)
_mul_s64(src reg, dst reg)
_mul_f32(src reg, dst reg)
_mul_f64(src reg, dst reg)
```

MV

Move register to register.

mv.type rS, rD

The value stored in in the source register is copied to the destination 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 integer types are supported.

Type	Opcode	Operand	
i8	MV_8	S	D
i16	MV_16	S	D
I32, f32	MV_32	S	D
I64, f64	MV_64	S	D

Inline C Macros

```
_mv_8(src reg, dst reg)
_mv_16(src reg, dst reg)
_mv_32(src reg, dst reg)
_mv_64(src reg, dst reg)
```

NEG

Negate value.

neg.type rS, rD

The value stored in in the source register is negated 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 integer types are supported.
32 and 64-bit floating point types are supported.

Type	**Opcode**	**Operand**	
s8	NEG_S8	S	D
s16	NEG_S16	S	D
s32	NEG_S32	S	D
s64	NEG_S64	S	D
f32	NEG_F32	S	D
f64	NEG_F64	S	D

Inline C Macros

```
_neg_s8(src reg, dst reg)
_neg_s16(src reg, dst reg)
_neg_s32(src reg, dst reg)
_neg_s64(src reg, dst reg)
_neg_f32(src reg, dst reg)
_neg_f64(src reg, dst reg)
```

OR

Bitwise OR.

or.type rS, rD

The value stored in in the source register is logically ORed with the value stored in the destination 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 integer types are supported.

Type	Opcode	Operand	
i8	OR_8	S	D
i16	OR_16	S	D
i32	OR_32	S	D
i64	OR_64	S	D

Inline C Macros

_or_8(src reg, dst reg)
_or_16(src reg, dst reg)
_or_32(src reg, dst reg)
_or_64(src reg, dst reg)

POP

Retrieve elements from the data stack.

pop.size reg_mask

For each register implied by a set bit in the extension word, load the value stored at the next stack location, decrementing the data stack position accordingly,

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

Size	**Opcode**	**Operand**	
8	POP_8	0x0	0x0
16	POP_16	0x0	0x0
32	POP_32	0x0	0x0
64	POP_64	0x0	0x0

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.

Inline C Macros

```
_pop_8(reg mask)
_pop_16(reg mask)
_pop_32(reg mask)
_pop_64(mask)
```

PUSH

Push elements to the data stack.

pop.size reg_mask

For each register implied by a set bit in the extension word, save the value stored at the next stack location, incrementing the data stack position accordingly.

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

Size	Opcode	Operand	
8	PUSH_8	0x0	0x0
16	PUSH_16	0x0	0x0
32	PUSH_32	0x0	0x0
64	PUSH_64	0x0	0x0

Size	Extension Word
all	Register mask

Registers are loaded in ascending order. If the data stack size is reached, execution halts and the status register is set to DATA_STACK_OVERFLOW.

Inline C Macros

_push_8(reg mask)
_push_16(reg mask)
_push_32(reg mask)
_push_64(mask)

RET

Return from function.

ret

The address of the next instruction is popped from the return stack. Execution then resumes from this location.

Opcode	**Operand**	
RET	0x0	0x0

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

Inline C Macros

_ret

RS

Load entire registers from the register stack.

restore reg_mask

For each register implied by a set bit in the extension word, load the full register contents at the next stack location, decrementing the register stack position accordingly,

Opcode	**Operand**	
RS	0x0	0x0

Extension Word
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.

Inline C Macros

_restore(reg mask)

ROL

Bitwise rotate left.

rol.type rS, rD

The value stored in in the destination register is bitwise rotated to the left 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 integer types are supported.

Type	**Opcode**	**Operand**	
i8	ROL_8	S	D
i16	ROL_16	S	D
i32	ROL_32	S	D
i64	ROL_64	S	D

Inline C Macros

_rol_8(src reg, dst reg)
_rol_16(src reg, dst reg)
_rol_32(src reg, dst reg)
_rol_64(src reg, dst reg)

ROR

Bitwise rotate right.

ror.type rS, rD

The value stored in in the destination register is bitwise rotated to the right 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 integer types are supported.

Type	**Opcode**	**Operand**	
i8	ROR_8	S	D
i16	ROR_16	S	D
i32	ROR_32	S	D
i64	ROR_64	S	D

Inline C Macros

```
_ror_8(src reg, dst reg)
_ror_16(src reg, dst reg)
_ror_32(src reg, dst reg)
_ror_64(src reg, dst reg)
```

SALLOC

Allocate storage on the data stack.

salloc #size, rD

The current data stack address is loaded into the destination register 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 SFREE.

Opcode	**Operand**	
SALLOC	0x0	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.

Inline C Macros

_salloc(size, dst reg)

SFREE

Deallocate storage on the data stack.

sfree 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 SALLOC call was made.

Opcode	**Operand**	
SFREE	0x0	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.

Inline C Macros

_sfree(src reg)

SUB

Subtraction.

sub.type rS, rD

The value stored in in the source register is subtracted from the value stored in the destination 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 integer types are supported.
32 and 64-bit floating point types are supported.

Type	**Opcode**	**Operand**	
i8	SUB_I8	S	D
i16	SUB_I16	S	D
i32	SUB_I32	S	D
i64	SUB_I64	S	D
f32	SUB_F32	S	D
f64	SUB_F64	S	D

Inline C Macros

```
_sub_8(src reg, dst reg)
_sub_16(src reg, dst reg)
_sub_32(src reg, dst reg)
_sub_64(src reg, dst reg)
_sub_f32(src reg, dst reg)
_sub_f64(src reg, dst reg)
```

SUBI

Subtract integer immediate.

subi.type #N, rD

The integer immediate stored in the extension word(s) is subtracted from the the value stored in the destination 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 and 32-bit integer immediates are supported.

Type	**Opcode**	**Operand**	
i8	SUBI_I8	0x0	D
i16	SUBI_I16	0x0	D
i32	SUBI_I32	0x0	D

Type	**Extension Word 1**	
i8	ignored	N
i16	N	
i32	N (host native half)	

Type	**Extension Word 2**
i32	N (host native half)

Inline C Macros

_subi_8(int literal, dst reg)
_subi_16(int literal, dst reg)
_subi_32(int literal, dst reg)

SV

Push entire registers to the register stack.

save reg_mask

For each register implied by a set bit in the extension word, save the full register contents at the next stack location, incrementing the register stack position accordingly,

Opcode	**Operand**	
SV	0x0	0x0

Extension Word
Register mask

Registers are saved in ascending order. If the register stack size is reached, execution halts and the status register is set to REGISTER_STACK_OVERFLOW.

Inline C Macros

_save(reg mask)

XOR

Bitwise Exclusvive OR.

xor.type rS, rD

The value stored in in the source register is logically XORed with the value stored in the destination 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 integer types are supported.

Type	**Opcode**	**Operand**	
i8	XOR_8	S	D
i16	XOR_16	S	D
i32	XOR_32	S	D
i64	XOR_64	S	D

Inline C Macros

```
_xor_8(src reg, dst reg)
_xor_16(src reg, dst reg)
_xor_32(src reg, dst reg)
_xor_64(src reg, dst reg)
```

XX_2_F32

Convert type to 32-bit floating point.

tof32.type rS, rD

Convert type to 32-bit floating point.

Type	Opcode	Operand	
u8	U8_2_F32	S	D
u16	U16_2_F32	S	D
u32	U32_2_F32	S	D
u64	U64_2_F32	S	D
s8	S8_2_F32	S	D
s16	S16_2_F32	S	D
s32	S32_2_F32	S	D
s64	S64_2_F32	S	D
f64	F64_2_F32	S	D

Inline C Macros

```
_u8to_f32(src reg, dst reg)
_u16to_f32(src reg, dst reg)
_u32to_f32(src reg, dst reg)
_u64to_f32(src reg, dst reg)
_s8to_f32(src reg, dst reg)
_s16to_f32(src reg, dst reg)
_s32to_f32(src reg, dst reg)
_s64to_f32(src reg, dst reg)
_f64to_f32(src reg, dst reg)
```

XX_2_F64

Convert type to 64-bit floating point.

tof64.type rS, rD

Convert type to 64-bit floating point.

Type	**Opcode**	**Operand**	
u8	U8_2_F64	S	D
u16	U16_2_F64	S	D
u32	U32_2_F64	S	D
u64	U64_2_F64	S	D
s8	S8_2_F64	S	D
s16	S16_2_F64	S	D
s32	S32_2_F64	S	D
s64	S64_2_F64	S	D
f32	F32_2_F64	S	D

Inline C Macros

```
_u8to_f64(src reg, dst reg)
_u16to_f64(src reg, dst reg)
_u32to_f64(src reg, dst reg)
_u64to_f64(src reg, dst reg)
_s8to_f64(src reg, dst reg)
_s16to_f64(src reg, dst reg)
_s32to_f64(src reg, dst reg)
_s64to_f64(src reg, dst reg)
_f32to_f64(src reg, dst reg)
```

XX_2_S8

Convert type to signed 8-bit integer.

tos8.type rS, rD

Convert type to signed 8-bit integer.

Type	**Opcode**	**Operand**	
f32	F32_2_S8	S	D
f64	F64_2_S8	S	D

Inline C Macros

_f32to_s8(src reg, dst reg)
_f64to_s8(src reg, dst reg)

XX_2_S16

Convert type to signed 16-bit integer.

tos16.type rS, rD

Convert type to signed 16-bit integer.

Type	**Opcode**	**Operand**	
s8	S8_2_S16	S	D
f32	F32_2_S16	S	D
f64	F64_2_S16	S	S

Inline C Macros

```
_s8to_s16(src reg, dst reg)
_f32to_s16(src reg, dst reg)
_f64to_s16(src reg, dst reg)
```

XX_2_S32

Convert type to signed 32-bit integer.

tos32.type rS, rD

Convert type to signed 32-bit integer.

Type	Opcode	Operand	
s8	S8_2_S32	S	D
s16	S16_2_S32	S	D
f32	F32_2_S32	S	S
f64	F64_2_S32	S	D

Inline C Macros

```
_s8to_s32(src reg, dst reg)
_s16to_s32(src reg, dst reg)
_f32to_s32(src reg, dst reg)
_f64to_s32(src reg, dst reg)
```

XX_2_S64

Convert type to signed 64-bit integer.

tos64.type rS, rD

Convert type to signed 64-bit integer.

Type	**Opcode**	**Operand**	
s8	S8_2_S64	S	D
s16	S16_2_S64	S	D
s32	S32_2_S64	S	S
f32	F32_2_S64	S	D
f64	F64_2_S64	S	D

Inline C Macros

```
_s8to_s64(src reg, dst reg)
_s16to_s64(src reg, dst reg)
_s32to_s64(src reg, dst reg)
_f32to_s64(src reg, dst reg)
_f64to_s64(src reg, dst reg)
```