The XMOS XS3 Architecture

Publication Date: 2024/10/10
Document Number: XM-014007-PS v2.0.0

IN THIS DOCUMENT

1 Introduction . . . 2
2 Interconnect . . . 2
3 Concurrent Threads . . . 5
4 The XCORE Instruction Set . . . 6
5 Instruction Issue and Execution . . . 8
6 Instruction Set Notation and Definitions . . . 10
7 Data Access . . . 12
8 Expression Evaluation . . . 14
9 Branching, Jumping and Calling . . . 15
10 Resources and the Thread Scheduler . . . 16
11 Concurrency and Thread Synchronisation . . . 17
12 Communication . . . 19
13 Locks . . . 21
14 Timers . . . 21
15 Ports, Input and Output . . . 22
16 Memory model . . . 29
17 Events, Interrupts and Exceptions . . . 30
18 Initialisation and Debugging . . . 34
19 Specialised Instructions . . . 35
20 Floating point arithmetic . . . 37
21 Vector unit . . . 39
22 XCore XS3 Instructions . . . 46
23 XCore XS3 Instruction Format Specification . . . 301
24 XCore XS3 Exceptions . . . 325
25 XCore XS3 Lanes . . . 337

1 Introduction

xcore.ai products combine a number of XCORE processors, each with its own memory, on a single chip. The programmable processors are *general purpose* in the sense that they can execute languages such as C; they also have direct support for concurrent processing (multi-threading), communication and input-output. A high-performance *switch* supports communication between the processors, and inter-chip XLINKs are provided so that systems can easily be constructed from multiple chips.

xcore.ai products are intended to make it practical to use software to perform many functions which would normally be done by hardware; an important example is interfacing and input-output controllers.

xcore.ai products are based on the XS3 architecture. The XS3 architecture is a evolution of the XS2 architecture. The main differences with the XS2 architecture are:

- Vector operations (*Vector unit*).
- Floating point operations (*Floating point arithmetic*).
- External memory (*External memory* and *Software defined memory*).

There are also extra instructions for counting leading sign bits, and defining shift operations with negative values..

2 Interconnect

The interconnect provides communication between all XCORES on the chip (or system if there is more than one chip). In conjunction with simple programs, it can also be used to

support access to the memory on any XCORE from any other XCORE, and to allow any XCORE to initiate programs on any other XCORE.

The interface between an XCORE and the interconnect is a group of XLINKs which carry *control tokens* and *data tokens*. The data tokens are simply bytes of data; the control tokens are as follows.

- Tokens 0-127 (*Application tokens*). These are intended for use by compilers or applications software to implement streamed, packetised and synchronised communications, to encode data-structures and to provide run-time type-checking of channel communications.
- Tokens 128-191 (*Special tokens*) are architecturally defined and may be interpreted by hardware or software. They are used to give standard encodings of common data types and structures.
- Tokens 192-223 (*Privileged tokens*) are architecturally defined and may be interpreted by hardware or privileged software. They are used to perform system functions including hardware resource sharing, control, monitoring and debugging. An attempt to transfer one of these tokens to or from unprivileged software will cause an exception.
- Tokens 224-255 (*Hardware tokens*) are only used by hardware; they control the physical operation of the link. An attempt to transfer one of these tokens using an output instruction will cause an exception.

Four links connect each XCORE directly to an on-chip switch which provides non-blocking communication between the XCOREs. The switch also provides off-chip XLINKs allowing multiple XS3, XS2, and XS1 chips to be combined in a system. The structure and performance of the XLINK connections in a system can be varied to meet the needs of applications.

The links between XCORES and switches and the XLINKs can be partitioned into independent networks. This can be used, for example, to provide independent networks carrying long and short messages or to provide independent networks for control and data messages.

Messages are routed to *channel-ends* on a specific processor through the XLINKs using a *message header* which contains the number of the destination chip, the number of the destination processor and the number of a destination channel-end within the processor. These can be encoded using either 24 bits (16 bits chip and processor address, 8 bits channel address) or 8 bits (3 bits chip and processor address, 5 bits channel address).

Each switch has a configurable identifier and can also be configured to route messages according to the first component of each message header. It compares this bit-by-bit with its own switch identifier; if all bits match it then uses the second component to route the message to the destination XCORE. If the bits do not match, then it uses the number of the first non-matching bit to select an outgoing direction. The direction of each XLINK is set when the switch is configured and it is possible for several XLINKs to share the same direction thereby providing several independent routes between two switches.

The header establishes a route through the interconnect and subsequent tokens will follow the same route until one of two special control tokens is sent: these are end-of-message (END) and pause (PAUSE).

2.1 XLINK Ports

The ports used for inter-chip XLINK communication use a transition-based non return-to-zero signalling scheme. Bits are sent at a rate derived from the XS3 clock; this rate can be programmed to meet applications requirements.

The XLINKs can be switched between a fast, wide mode and a slower, serial mode. Two encoding schemes are used.

2.2 Serial XLINK

The serial XLINK uses two data wires in each direction. A transition on Wire 1 represents a one bit and a transition on Wire 0 represents a zero bit. The first bit of a *control* token is a one; the first bit of a *data* token is a zero; the next 8 bits are the token value. The two signal wires are both at rest between tokens and the final bit of each token is chosen to return the non-zero signal wire to the rest state; one of the signal wires must be non-zero at this point as nine bits have been sent.

On the serial link, the END and PAUSE tokens are coded directly as application tokens 1 and 2.

The link also uses several hardware tokens. The credit tokens are transmitted by the receiver to control the flow of data; each CREDIT*n* token issues credit to the sender to allow it to send *n* tokens. The HELLO token solicits initial credits, setting up a half-duplex link. To bring up a link, both sides have to issue a HELLO, and both sides have to respond to the HELLO with a CREDIT*n* token.

token	use
224	CREDIT8
225	CREDIT64
228	CREDIT16
230	HELLO

2.3 Fast XLINK

The fast XLINK uses 1-of-5 codes with five data wires in each direction; a *symbol* is transmitted by changing the state of one of the wires. Each symbol has the following meaning:

symbol	meaning
Wire 0 changes	value 00
Wire 1 changes	value 01
Wire 2 changes	value 10
Wire 3 changes	value 11
Wire 4 changes	escape

A sequence of four symbols are used to encode each token. In the following *e* is an escape and *v* is one of the values *00*, *01*, *10*, *11*.

symbol sequence	use
v v v v	256 data tokens
e v v v	64 control tokens 192-255
v e v v	64 control tokens 128-191
v v e v	64 control tokens 64-127
v v v e	64 control tokens 0-63

There are some additional codes in which more than one symbol is an escape. These are used to code certain control tokens.

symbol sequence	use
e e v v	END tokens
v v e e	PAUSE tokens
e v v e	NOP (return to zero) tokens
e 11 11 v	NOP (return to zero) tokens
e 00 e 00	CREDIT8
e 01 e 01	CREDIT64
e 10 e 10	HELLO
e 11 e 11	CREDIT16

Because each token contains four symbols, at the end of each token there are always an even number of signal wires in a non-zero state. To send an END or PAUSE, one of the END or PAUSE tokens is chosen to leave at most two signal wires in a non-zero state; this can be followed by a NOP token which is chosen to leave all of the signal wires in a zero state.

The encoding of the credit and reset tokens has been chosen so that the state of the signal wires after the token is the same as it was before the token.

3 Concurrent Threads

A single XCore enables a number of tasks to execute concurrently in *threads*. Each thread executes a series of instructions that follow a conventional three register operand model. Threads have access to *resources* that enable a thread to interact with other threads or the outside world.

Each XCORE has hardware support for executing a number of concurrent threads. This includes:

- a set of registers for each thread.
- a thread scheduler which dynamically selects which thread to execute.
- a set of synchronisers to synchronise thread execution.
- a set of channels used for communication with other threads.
- a set of ports used for input and output.
- a set of timers to control real-time execution.

- a set of clock generators to enable synchronisation of the input-output with an external time domain.
- a set of hardware locks to enable low level locking

Instructions are provided to support initialisation, termination, starting, synchronising and stopping threads; also there are instructions to provide input-output and inter-thread communication.

The set of threads on each XCORE can be used:

- to implement input-output controllers executed concurrently with applications software.
- to allow communications or input-output to progress together with processing.
- to allow latency hiding in the interconnect by allowing some threads to continue whilst others are waiting for communication to or from remote XCOREs.

The instruction set includes instructions that enable the threads to communicate and perform input and output. These:

- provide event-driven communications and input-output with waiting threads automatically descheduled.
- support streamed, packetised or synchronised communication between threads anywhere in a system.
- enable the processor to idle with clocks disabled when all of its threads are waiting so as to save power.
- allow the interconnect to be pipelined and input-output to be buffered.

4 The XCORE Instruction Set

The main features of the instruction set used by the XCORE processors are as follows.

- Short instructions are provided to allow efficient access to the stack and other data regions allocated by compilers; these also provide efficient branching and subroutine calling. The short instructions have been chosen on the basis of extensive evaluation to meet the needs of modern compilers.
- The memory is byte addressed; however all accesses must be aligned on natural boundaries so that, for example, the addresses used in 32-bit loads and stores must have the two least significant bits zero. The memory is little endian.
- The processor supports a number of threads each of which has its own set of registers. Some registers are used for specific purposes such as accessing the stack, the data region or large constants in a constant pool.
- Input and output instructions allow very fast communications between threads within an XCORE and between XCOREs. They also support high speed, low-latency, input and output. They are designed to support high-level concurrent programming techniques.

Most instructions are 16-bit. Many instructions use operands in the range *0 ... 11* as this allows sufficient three-address instructions to be encoded using 16 bit instructions. Instruction prefixes are used to extend the range of immediate operands and to provide more inter-register operations (and inter-register operations with more operands). The prefixes are:

- PFIX which concatenates its 10-bit immediate with the immediate operand of the next 16-bit instruction.
- EOPR which concatenates its 11-bit operation set with the following instruction.

The prefixes are inserted automatically by compilers and assemblers.

The normal state of a thread is represented by 12 operand registers, 4 access registers and 2 control registers.

The twelve operand registers *r0* … *r11* are used by instructions which perform arithmetic and logical operations, access data structures, and call subroutines.

The access registers are:

register	number	use
cp	12	constant pool pointer
dp	13	data pointer
sp	14	stack pointer
lr	15	link register

The control registers are:

register	number	use
pc	16	program counter
sr	17	status register

Each thread has seven additional registers which have very specific uses:

register	number	use
spc	18	saved pc
ssr	19	saved status
et	20	exception type
ed	21	exception data
sed	22	saved exception data
kep	23	kernel entry pointer
ksp	24	kernel stack pointer

The status register *sr* contains the following information:

bit	number	use
eeble	0	event enable
ieble	1	interrupt enable
inenb	2	thread is enabling events
inint	3	thread is in interrupt mode
ink	4	thread is in kernel mode
reserved	5	do not use
waiting	6	thread waiting to execute current instruction
fast	7	thread enabled for fast input-output
di	8	thread is running in dual issue mode
kedi	9	thread switches to dual issue on kernel entry
hipri	10	thread is in high priority mode

5 Instruction Issue and Execution

The processor is implemented using a short pipeline to maximise responsiveness. It is optimised to provide deterministic execution of multiple threads. There is no need for forwarding between pipeline stages and no need for speculative instruction issue and branch prediction. The memory is 128-bit wide, enabling sufficient instructions to be fetched simultaneously to enable the processor to run at full speed using a unified memory system. Long sequences of memory accesses require an occasional instruction fetch, consuming one extra thread cycle.

5.1 Scheduler Implementation

The threads in an XCORE are intended to be used to perform several simultaneous real-time tasks such as input-output operations, so it is important that the performance of an individual thread can be guaranteed. The scheduling method used allows any number of threads to share a single unified memory system and input-output system whilst guaranteeing that with *n* threads able to execute, each will get at least *1/n* processor cycles. In fact, it is useful to think of a *thread cycle* as being *n* processor cycles.

From a software design standpoint, this means that the minimum performance of a thread can be calculated by counting the number of concurrent threads at a specific point in the program. In practice, performance will almost always be higher than this because individual threads will sometimes be delayed waiting for input or output and their unused processor cycles can be taken by other threads. Further, the time taken to re-start a waiting thread is always at most one thread cycle. (Note that the use of priority threads will cause a slightly different but still predictable performance pattern, see *High priority threads*.)

The set of *n* threads can therefore be thought of as a set of virtual processors each with clock rate at least *1/n* of the clock rate of the processor itself. The only exception to this is that if the number of threads is less than the pipeline depth *p*, the clock rate is at most *1/p*.

Each thread has a 256-bit instruction buffer which is able to hold sixteen short instructions or eight long ones. Instructions are issued from the runnable threads in a round-robin manner, ignoring threads which are not in use or are paused waiting for a synchronisation or input-output operation.

The pipeline has a memory access stage which is available to *all* instructions. The rules for performing an instruction fetch are as follows.

- Any instruction which requires data-access performs it during the memory access stage.
- Branch instructions fetch their branch target instructions during the memory access stage unless they also require a data access (in which case they will leave the instruction buffer empty).
- Conditional branches only ever fetch instructions around the target address.
- Any other instruction (such as ALU operations) uses the memory access stage to perform an instruction fetch. This is used to load the thread's own instruction buffer unless it is full.
- If the instruction buffer is empty when an instruction should be issued, a special *fetch no-op* is issued; this will use its memory access stage to load the issuing thread's instruction buffer.

There are very few situations in which a *fetch no-op* is needed, and these can often be avoided by simple instruction scheduling in compilers or assemblers. An obvious example is to break long sequences of loads or stores by interspersing ALU operations.

Certain instructions cause threads to become non-runnable because, for example, an input channel has no available data. When the data becomes available, the thread will continue from the point where it paused.

To achieve this, each thread has an individual ready request signal. The thread identifier is passed to the resource (port, channel, timer etc) and used by the resource to select the correct ready request signal. The assertion of this will cause the thread to be restarted, normally by re-entering it into the round-robin sequence and re-issuing the input instruction. In most situations this latency is acceptable, although it results in a response time which is longer than the virtual cycle time because of the time for the re-issued instruction to pass through the pipeline.

To enable the virtual processor to perform one input or output per virtual cycle, a *fast-mode* is provided. When a thread is in fast-mode, it is not de-scheduled when an instruction can not complete; instead the instruction is re-issued until it completes.

Events and interrupts are slightly different from normal input and output, because a vector must also be supplied and the target instruction fetched before execution can proceed. However, the same ready request system is used. The result will be to make the thread runnable but with an empty instruction buffer.

A variation on the *fetch no-op* is the *event no-op*; this is used to access the resource which generated the event (or interrupt) using the thread identifier; the resource can then supply the appropriate vector in time for it to be used for instruction fetch during the event no-op memory access stage. This means that at most one virtual cycle is used to process the vector, so there will be at most two virtual cycles before instruction issue following an event or interrupt.

The XCORE scheduler therefore allows threads to be treated as virtual processors with performance predicted by tools. There is no possibility that the performance can be reduced below these predicted levels when virtual processors are combined.

5.2 Single and Dual Issue

An XS3 pipeline has two *lanes*: the memory lane can execute all memory instructions, branches, and basic arithmetic, and the resource lane can execute all resource instructions and basic arithmetic. Each thread can chose to execute in *dual issue mode*, in which case the processor will execute two 16-bit instructions or a single 32-bit instruction in a

single thread cycle. In dual issue mode, all instructions must be aligned: 32-bit instructions must be 32-bit aligned and pairs of 16-bit instructions must be aligned on a 32-bit boundary. The program counter is always aligned to a 32-bit boundary and points to an issue slot rather than to an individual instruction. The 16 bit value stored at addresses *4n+2* and *4n+3* encodes an instruction for the memory lane. The 16-bit value stored at at addresses *4n+0* and *4n+1* encodes an instruction for the resource lane. Long instructions are stored in a word at addresses *4n+0...4n+3*.

Where two instructions are executed simultaneously, any destination operands should be disjoint. If they are not disjoint, an exception will be raised.

When the resource lane stalls a thread, the other lane will be stalled also. This is normally not observable, except when an interrupt or an exception is raised. On an interrupt or exception, no registers will be overwritten, and the PC will point to the instruction to be reexecuted.

If an instruction in one of the two lanes causes an exception, then this exception is reported. If the other lane is executing an instruction then this second instruction is aborted. If the instructions in both lanes cause an exception, then only one exception is reported, and both instructions are aborted, but any memory store which is in progress will complete. On an exception, the savedPC value is set to the instruction that caused the exception.

A single bit in the status register, DI, enables dual-issue. If this bit is not set, then instructions flow through one lane at a time, and mis-aligned 32-bit instructions are allowed. The dual-issue-bit is set and cleared on a per function basis. The bit is saved in the lowest bit of LR when a function call is taken. It is restored on a RETSP instruction. The dual-issue-bit is set on executing a DUALENTSP x instruction, and cleared on executing an ENTSP x instruction. This enables functions to be dual or single issue.

5.3 High priority threads

Threads can be set to be *high priority*. If no high priority threads are runnable, then a low priority thread will be scheduled if one is runnable. If high priority threads are runnable, then they will be scheduled, but at least one low priority thread will be executed on every iteration of the high priority queue. This means that all threads are always guaranteed progress.

Threads start as low-priority and only threads that require a very short turn around time or maximum throughput will be high priority.

6 Instruction Set Notation and Definitions

In the following description

Identifier	Meaning
Bpw	is the number of bytes in a word
bpw	is the number of bits in a word
mem	represents the memory
pc	represents the program counter
sr	represents the status register
sp	represents the stack pointer
dp	represents the data pointer
cp	represents the constant pool pointer
lr	represents the link register
r0 … r11	represent specific operand registers
x	(a single small letter) represents one of *r0 … r11*
X	(a single large letter) represents one of *r0 … r11*, *sp*, *dp*, *cp*, *lr*
u_s	is a small unsigned source operand in the range *0 … 11*
bitp	is one of *bpw*, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32 encoded as a *u_s*
u16	is a 16-bit source operand in the range *0 … 65535*
u20	is a 20-bit source operand in the range *0 … 1048575*
iw	is the issue-width in bytes, 2 (for single issue) or 4 (for dual issue)

Note that when the program counter (*pc*) is used by an instruction, it is always pointing to the next instruction. Instructions that access the location of the current instruction use *pc_old*.

The operators used in this manual are:

Operator	Meaning
||	logical or
|	bitwise or
&&	logical and
&	bitwise and
+, -, x , //, %	arithmetic operations; full precision unsigned integer, unless specified as *signed*
2^n	integer power
l <- r	assignment of *r* to *l*; if *r* has more bits than *l*, then the most significant bits of *r* will be ignored
!	logical not
~	bitwise not
@	bitwise xor
mem[x]	An entity at memory address x
y[bit x]	A single bit of *y*
y[bits x..z]	A slice of *y* comprising *x-z+1* bits; *x >= z*
x:y	Concatenates *x* and *y*, ie, *x « bpw | y*
forall x in y	for each value *x* in the set *y*

Some useful functions are

- *zext(x, n) = x & (2^n - 1)*: zero extend
- *sext(x, n) = -(2^(n-1) & x) | x*: sign extend

6.1 Instruction Prefixes

If the most significant *10* bits of a *u16* or *u20* instruction operand are non-zero, a 16-bit prefix (PFIX) preceding the instruction is used to encode them. The least significant bits are encoded within the instruction itself.

A different kind of 16-bit prefix (EOPR) is used to encode instructions with more than three operands, or to encode the less common instructions.

7 Data Access

7.1 Access to words

The data access instructions fall into several groups. One of these provides access via the stack pointer.:

```
LDWSP   D <- mem[sp + u16  x  Bpw]      load word from stack
STWSP   mem[sp + u16  x  Bpw] <- S      store word to stack
LDAWSP  D <- sp + u16  x  Bpw           load address of word in stack
```

Another is similar, but provides access via the data pointer.:

```
LDWDP   D <- mem[dp + u16  x  Bpw]      load word from data
STWDP   mem[dp + u16  x  Bpw] <- S      store word to data
LDAWDP  D <- dp + u16  x  Bpw           load address of word in data
```

Access to constants and program addresses is provided by instructions which either load values directly or load them from the constant pool.:

```
LDC     D <- u16                    load constant
LDWCP   D <- mem[cp + u16  x  Bpw]  load word from constant pool
LDAWCP  r11 <- cp + u16  x  Bpw]    load word address in constant pool
LDWCPL  r11 <- mem[cp + u20  x  Bpw] load word from constant pool long
LDAPF   r11 <- pc + u20  x  iw      load address in program forward
LDAPB   r11 <- pc - u20  x  iw      load address in program backward
```

Access to data structures is provided by instructions which use any of the operand registers as a base address, and combine this with a scaled offset. In the case of word accesses, the operand may be a small constant or another operand register, and the instructions are as follows:

```
LDWI    d <- mem[b + u_s  x  Bpw]      load word
STWI    mem[b + u_s  x  Bpw] <- s      store word
LDAWFI  d <- b + u_s  x  Bpw           load address of word forward
LDAWBI  d <- b - u_s  x  Bpw           load address of word backward

LDW     d <- mem[b + i  x  Bpw]        load word
STW     mem[b + i  x  Bpw] <- s        store word
LDAWF   d <- b + i  x  Bpw             load address of word forward
LDAWB   d <- b - i  x  Bpw             load address of word backward
```

7.2 Access to sub-words

In the case of access to 16-bit quantities, the base address is combined with a scaled operand, which must be an operand register. The least significant bit of the resulting address must be zero. The 16-bit item is loaded and sign extended into a word:

```
LD16S   d <- sext(mem[b+i x  2],16)    load 16-bit signed item
ST16    mem[b + i  x  2] <- s          store 16-bit item
LDA16F  d <- b + i  x  2               load address of 16-bit item forward
LDA16B  d <- b - i  x  2               load address of 16-bit item backward
```

In the case of access to 8-bit quantities, the base address is combined with an unscaled operand, which must be an operand register. The 8-bit item is loaded and zero extended into a word.:

```
LD8U    d <- zext(mem[b + i], 8)       load byte unsigned
ST8     mem[b + i] <- s                store byte
```

Access to part words, including bit-fields, is provided by a small set of instructions which are used in conjunction with the shift and bitwise operations described below. These instructions provide for mask generation of any length up to 32 bits, sign extension and zero-extension from any bit position, and clearing fields within words prior to insertion of new values.:

```
MKMSK   d <- 2^s - 1                   make mask
MKMSKI  d <- 2^bitp - 1                make mask immediate
SEXT    d <- sext(d, s)                sign extend
SEXTI   d <- sext(d, bitp)             sign extend immediate
ZEXT    d <- zext(d, s)                zero extend
ZEXTI   d <- zext(d, bitp)             zero extend immediate
ANDNOT  d <- d && ~ s                  and not (clear field)
```

The SEXTI and ZEXTI instructions can also be used in conjunction with the LD16S and LD8U instructions to load unsigned 16-bit and signed 8-bit values.

7.3 Access to double words

Pairs of words can be accessed in a single instruction. This requires the address to be aligned on a two-word boundary; it must be a multiple of Bpw x 2. For store operations two destination registers must be specified, for load operations two source registers must be specified.:

```
LDDSP   d <- mem[sp + u_s x Bpw x 2]        load two words from stack
        e <- mem[sp + u_s x Bpw x 2 + Bpw]
STDSP   mem[sp + u_s x Bpw x 2] <- x        store two words to stack
        mem[sp + u_s x Bpw x 2 + Bpw] <- y
LDDI    d <- mem[b + u_s x Bpw x 2]         load two words
        e <- mem[b + u_s x Bpw x 2 + Bpw]
STDI    mem[b + u_s x Bpw x 2] <- x         store two words
        mem[b + u_s x Bpw x 2 + Bpw] <- y
LDD     d <- mem[b + i x Bpw x 2]           load two words
        e <- mem[b + i x Bpw x 2 + Bpw]
STD     mem[b + i x Bpw x 2] <- x           store two words
        mem[b + i x Bpw x 2 + Bpw] <- y
```

Note that the stack pointer must be double word aligned if double loads and double stores are used. The LDDSP and STDSP instructions can be used for saving context efficiently.

8 Expression Evaluation

```
ADDI    d <- l + u_s                      add immediate
ADD     d <- l + r                        add
SUBI    d <- l - u_s                      subtract immediate
SUB     d <- l - r                        subtract
NEG     d <- - s                          negate

EQI     d <- l = u_s                      equal immediate
EQ      d <- l = r                        equal
LSU     d <- l < r                        less than unsigned
LSS     d <- l <_sgn r                    less than signed

AND     d <- l & r                        and
OR      d <- l | r                        or
XOR     d <- l @ r                        exclusive or
XOR4    d <- l @ r @ s @ t                exclusive or
NOT     d <- ~ s                          not

SHLI    d <- l << bitp                    logical shift left immediate
SHL     d <- l << r                       logical shift left
SHRI    d <- l >> bitp                    logical shift right immediate
SHR     d <- l >> r                       logical shift right
ASHRI   d <- l >>_sgn bitp                arithmetic shift right immediate
ASHR    d <- l >>_sgn r                   arithmetic shift right

MUL     d <- l  x  r                      multiply
DIVU    d <- l // r                       divide unsigned
DIVS    d <- l //_sgn r                   divide signed
REMU    d <- l % r                        remainder unsigned
REMS    d <- l %_sgn  r                   remainder signed

NOP                                       no operation
```

The shift instructions SHL, SHR, and ASHR interpret their third operand (the shift distance) as a signed number, and accept negative operands. When presented with a negative shift-value:

- SHR will perform a shift left with the negative value.
- ASHR will perform a shift left with the negative value.
- SHL will perform an arithmetic shift right with the negative value.

Note that for positive shifts, these instructions are backwards compatible with the XS2 architecture.

```
BITREV   d : for x in bpw     d[bit  ix] = s[bit  bpw-ix-1]  bit reverse
BYTEREV  d : for ix in bpw/8  d[byte ix] = s[byte Bpw-ix-1]  byte reverse
CLZ      d <- 32,                               if s = 0     count leading zeros
              lowest x : s[bpw-1-x] = 1,        otherwise

CLS      d <- 32,                               if s = 0     count leading signbits
              32,                               if s = -1
              lowest x : s[bpw-1-x] != s[bpw-1], otherwise

ZIP      w = 2^s                                             zip double word
         z = d[bpw-1..bpw-w-1]:
             e[bpw-1..bpw-w-1]:
             d[bpw-w-1..bpw-2  x  w-1]:
             e[bpw-w-1..bpw-2  x  w-1]: ... :
             d[w-1..0]:
             e[w-1..0]:
```

(continues on next page)

(continued from previous page)

```
        d <- z[2 bpw-1..bpw]
        e <- z[bpw-1..0]

UNZIP   w = 2^s                                       unzip double word
        z = d:e
        d <- z[2  x  bpw-1..2  x  bpw-w-1]:
             z[2  x  bpw-2w-1..2  x  bpw-3w-1]:...:
             z[2w-1..w]
        e <- z[2  x  bpw-w-1..2  x  bpw-2w-1]:
             z[2  x  bpw-3w-1..2  x  bpw-4w-1]:...:
             z[w-1..0]
```

9 Branching, Jumping and Calling

The branch instructions include conditional and unconditional relative branches. A branch using the address in a register is provided; a relative branch which adds a scaled register operand to the program counter is provided to support jump tables:

```
BRFT    if  c  then  pc <- pc + u16 x iw     branch relative forward true
BRFF    if !c  then  pc <- pc + u16 x iw     branch relative forward false
BRBT    if  c  then  pc <- pc - u16 x iw     branch relative backward true
BRBF    if !c  then  pc <- pc - u16 x iw     branch relative backward false

BRFU    pc <- pc + u16 x iw                  branch relative forward unconditional
BRBU    pc <- pc - u16 x iw                  branch relative backward unconditional
BRU     pc <- pc + s x iw                    branch relative unconditional via reg

BAU     pc <- s                              branch absolute unconditional via reg
```

In some cases, the calling instructions described below can be used to optimise branches; as they overwrite the link register they are not suitable for use in leaf procedures which do not save the link register.

The procedure calling instructions include relative calls, calls via the constant pool, indexed calls via a dedicated register (r11) and calls via a register. Most calls within a single program module can be encoded in a single instruction; inter-module calling requires at most two instructions.:

```
BLRF    lr <- pc || sr[bit  di];              branch and link relative forward
        pc <- pc + u20  x  iw
BLRB    lr <- pc || sr[bit  di];              branch and link relative backward
        pc <- pc - u20  x  iw

BLACP   lr <- pc || sr[bit  di];              branch and link absolute via CP
        pc <- mem[cp + u20  x  Bpw]
BLAT    lr <- pc || sr[bit  di];              branch and link absolute via table
        pc <- mem[r11 + u16  x  Bpw]
BLA     lr <- pc || sr[bit  di]               branch and link absolute via register
        pc <- s
```

Notice that control transfers which do not affect the link (required for tail calls to procedures) can be performed using one of the LDWCP, LDWCPL, LDAPF or LDAPB instructions followed by BAU r11.

Calling may require modification of the stack. Typically, the stack is extended on procedure entry and contracted on exit. The instructions to support this are shown below:

```
EXTSP   sp <- sp - u16  x  Bpw              extend stack
EXTDP   dp <- dp - u16  x  Bpw              extend data

ENTSP   if  u16 > 0  then                   SI entry and extend stack
            mem[sp] <- lr;
            sp <- sp - u16  x  Bpw
        sr[bit  di]<- false

DUALENTSP if  u16 > 0   then                DI entry and extend stack
            mem[sp] <- lr;
            sp <- sp - u16  x  Bpw
        sr[bit  di]<- true

RETSP   if  u16 > 0  then                   contract stack
            sp <- sp + u16  x  Bpw;         and return
            lr <- mem[sp];
        sr[bit  di] <- lr & 1
        pc <- lr & ~ 1
```

Functions can be made that can be entered in either single or dual issue:

- A single issue function must start with either a 32-bit aligned, long ENTSP instruction, or a short 32-bit aligned instruction that is paired with a dual-issuable instruction. This enables the function to be called from both single and dual issue contexts.
- A DUALENTSP instruction must either be a long instruction that is 32-bit aligned, or it must be a short DUALENTSP that is stored in the third and fourth byte of the word, together with an instruction that can be executed in the resource lane.

A short DUALENTSP executed in single issue stored in the lower 16-bits of a word will raise an exception in the following instruction, since the PC will be misaligned.

Notice that the stack and data area can be contracted using the LDAWSP and LDAWDP instructions.

In some situations, it is necessary to change to a new stack pointer, data pointer or pool pointer on entry to a procedure. Saving or restoring any of the existing pointers can be done using normal STWS, STWD, LDWS or LDWD instructions; loading them from another register can be optimised using the following instructions.:

```
SETSP   sp <- s                 set stack pointer
SETDP   dp <- s                 set data pointer
SETCP   cp <- s                 set pool pointer
```

10 Resources and the Thread Scheduler

Each XCORE manages a number of different types of *resource*. These include threads, synchronisers, channel ends, timers and locks. For each type of resource a set of available items is maintained. The names of these sets are used to identify the type of resource to be allocated by the GETR (get resource) instruction. When the resource is no longer needed, it can be released for subsequent use by a FREER (free resource) instruction.:

```
GETR    r <- first res in setof(us): !inuse(res)       get resource
        inuse(r) <- true

FREER   inuse(r) <- false                              free resource
```

In the above `setof(r)` returns the set corresponding to the source operand of `r`. The resources are:

resource name	set	use
THREAD	threads	concurrent execution
SYNC	synchronisers	thread synchronisation
CHANEND	channel ends	thread communication
TIMER	timers	timing
LOCK	locks	mutual exclusion

Some resources have associated control *modes* which are set using the SETC instruction.:

```
SETC    control(r) <- u16            set resource control
```

Many of the mode settings are defined only for a specific kind of resource and are described in the appropriate section; the ones which are used for several different kinds of resource are:

mode	effect
OFF	resource off
ON	resource on
START	resource active
STOP	resource inactive
EVENT	resource will cause events
INTERRUPT	resource will raise interrupts

Execution of instructions from each thread is managed by the *thread scheduler*. This maintains a set of runnable threads, run, from which it takes instructions in turn. When a thread is unable to continue, it is *paused* by removing it from the run set. The reason for this may be any of the following.

- Its registers are being initialised prior to it being able to run.
- It is waiting to synchronise with another thread before continuing.
- It is waiting to synchronise with another thread and terminate (a *join*).
- It has attempted an input from a channel which has no data available, or a port which is not ready, or a timer which has not reached a specified time.
- It has attempted an output to a channel or a port which has no room for the data.
- It has executed an instruction causing it to wait for one of a number of events or interrupts which may be generated when channels, ports or timers become ready for input.

The thread scheduler manages the threads, thread synchronisation and timing (using the synchronisers and timers). It is directly coupled to resources such as the ports and channels so as to minimise the delay when a thread becomes runnable as a result of a communication or input-output.

11 Concurrency and Thread Synchronisation

A thread can initiate execution on one or more newly allocated threads, and can subsequently synchronise with them to exchange data or to ensure that all threads have completed before continuing. Thread synchronisation is performed using hardware *synchronisers*, and threads using a synchroniser will move between running states and paused states. When a thread is first created, its status register is initialised as follows:

```
sr[bit eeble]    <-     0
sr[bit ieble]    <-     0
sr[bit inenb]    <-     0
sr[bit inint]    <-     0
sr[bit hipri]    <-     0
sr[bit fast]     <-     0
sr[bit kedi]     <-     0
sr[bit waiting]  <-     1                  the thread is paused
sr[bit di]       <-     0
```

The access registers of the newly created thread can be initialised using the following instructions.:

```
TINITPC     pc(t) <- s                set thread pc
TINITSP     sp(t) <- s                set thread stack
TINITDP     dp(t) <- s                set thread data
TINITCP     cp(t) <- s                set thread pool
TINITLR     lr(t) <- s                set thread link
```

These instructions can only be used when the thread is paused. The TINITLR instruction is intended primarily to support debugging. On thread initialisation, the PC must be initialised. DP, SP, and CP will retain their value on freeing and allocating threads, so they may not have to be reinitialised.

Data can be transferred between the operand registers of two threads using TSETR and TSETMR instructions, which can be used even when the destination thread is running.:

```
TSETR      d(t) <- s                  set thread operand register
TSETMR     d(mstr(tid)) <- s          set master thread operand register
```

To start a *synchronised* slave thread a master must first acquire a synchroniser. This is done using a GETR SYNC instruction. If there is a synchroniser available its resource ID is returned, otherwise the invalid resource ID is returned. The GETST instruction is then used to get a synchronised thread. It is passed the synchroniser ID and if there is a free thread it will be allocated, attached to the synchroniser and its ID returned, otherwise the invalid resource ID is returned.

The master thread can repeat this process to create a group of threads which will all synchronise together. To start the slave threads the master executes an MSYNC instruction using the synchroniser ID.:

```
GETST  d <- first res in setof(threads): !inuse(res)    get synchronised thread
       inuse(d) <- true
       spaused <- spaused union {d};
       slaves(s) <- slaves(s) union {d}
       mstr(s) <- tid

MSYNC  if  (slaves(s) setminus spaused = {})           master synchronise
       then
            spaused <- spaused setminus slaves(s)
       else
           mpaused <- mpaused union {tid};
           msyn(s) <- true
```

The group of threads can synchronise at any point by the slaves executing the SSYNC and the master the MSYNC. Once all the threads have synchronised they are unpaused and continue executing from the next instruction. The processor maintains a set of paused master threads mpaused and a set of paused slave threads spaused from which it derives the set of runnable threads `run`

```
run = {thread in threads : inuse(thread)} setminus (spaused union mpaused)
```

Each synchroniser also maintains a record msyn(s) of whether its master has reached a synchronisation point.:

```
SSYNC if  (slaves(syn(tid)) setminus spaused = {tid}) && msyn(syn(tid))   slave
      then                                                                   synchronise
           if  mjoin(syn(tid))
           then
               for t in slaves(syn(tid)) : inuse(t) <- false;
               mjoin(syn(tid)) <- false
           else
               spaused <- spaused setminus slaves(syn(tid));
           mpaused <- mpaused setminus {mstr(syn(tid))};
           msyn(syn(tid)) <- false
      else
           spaused <- spaused union {tid}
```

To terminate all of the slaves and allow the master to continue the master executes an MJOIN instruction instead of an MSYNC. When this happens, the slave threads are all freed and the master continues.:

```
MJOIN if  (slaves(s) = spaused)                        master join
      then
           for t in slaves(s) : inuse(t) <- false;
           mjoin(syn(tid)) <- false
      else
          mpaused <- mpaused union {tid};
          mjoin(s) <- true;
          msyn(s) <- true
```

A master thread can also create threads which can terminate themselves. This is done by the master executing a GETR THREAD instruction. This instruction returns either a thread ID if there is a free thread or the invalid resource ID. The unsynchronised thread can be initialised in the same way as a synchronised thread using the TINITPC, TINITSP, TINITDP, TINITCP, TINITLR and TSETR instructions.

The unsynchronised thread is then started by the master executing a TSTART instruction specifying the thread ID. Once the thread has completed its task it can terminate itself with the FREET instruction.:

```
TSTART   spaused <- spaused setminus {tid}    start thread
FREET    inuse(tid) <- false;                 free thread
```

The identifier of an executing thread can be accessed by the GETID instruction.:

```
GETID     t <- tid                              get thread identifier
```

12 Communication

Communication between threads is performed using *channels*, which provide full-duplex data transfer between *channel ends*, whether the ends are both in the same XCORE, in different XCORES on the same chip or in XCORES on different chips. Channels carry messages constructed from data and control *tokens* between the two channel ends. The control tokens are used to encode communication protocols. Although most control tokens are available for software use, a number are reserved for encoding the protocol used by the interconnect hardware, and can not be sent and received using instructions.

A channel end can be used to generate events and interrupts when data becomes available as described below. This allows a thread to monitor several channels, ports or timers, only servicing those that are ready.

To communicate between two threads, two channel ends need to be allocated, one for each thread. This is done using the GETR c , CHANEND instruction. Each channel end has a *destination* register which holds the identifier of the destination channel end; this is initialised with the SETD instruction. It is also possible to use the identifier of a channel end to determine its destination channel end.:

```
SETD                 r(dest) <- s              set destination
GETD                 d <- r(dest)              get destination
```

The identifier of the channel end c1 is used to initialise the channel end for thread c2 , and vice versa. Each thread can then use the identifier of its own channel end to transfer data and messages using output and input instructions.

The interconnect can be partitioned into several independent networks. This makes it possible, for example, to allocate channels carrying short control messages to one network whilst allocating channels carrying long data messages to another. There are instructions to allocate a channel to a network and to determine which network a channel is using.:

```
SETN      c(net) <- s                 set network
GETN      d <- c(net)                 get network
```

In the following, **c** `<:` **s** represents an output of **s** to channel **c** and **c** `:>` **d** represents an input from channel **c** into **d**.:

```
OUTT      c <: dtoken(s)                output token
OUTCT     c <: ctoken(s)                output control token
OUTCTI    c <: ctoken(us)               output control token immediate

INT       if hasctoken(c)               input token
          then trap
```

(continues on next page)

(continued from previous page)

```
         else  c :> d
INCT     if hasctoken(c)            input control token
         then c :> d
         else trap

CHKCT    if hasctoken(c)&&(s=token(c))    check control token
         then skiptoken(c)
         else trap
CHKCTI   if hasctoken(c)&&(s=token(c))    check control token immediate
         then skiptoken(c)
         else trap

OUT      c <: s                     output data word
IN       if containsctoken(c)       input token
         then trap
         else  c :> d

TESTCT   d <- hasctoken(c)          test for control token
TESTWCT  d <- containsctoken(c)     test word for control token
```

The channel connection is established when the first output is executed. If the destination channel end is on another XCORE, this will cause the destination identifier to be sent through the interconnect, establishing a route for the subsequent data and control tokens. The connection is terminated when an END control token is sent. If a subsequent output is executed using the same channel end, the destination identifier will be used again to establish a new route which will again persist until another END control token is sent.

A destination channel end can be shared by any number of outputting threads; they are served in a round-robin manner. Once a connection has been established it will persist until an END is received; any other thread attempting to establish a connection will be queued. In the case of a shared channel end, the outputting thread will usually transmit the identifier of its channel end so that the inputting thread can use it to reply.

The OUT and IN instructions are used to transmit words of data through the channel; to transmit bytes of data the OUTT and INT instructions are used. Control tokens are sent using OUTCT or OUTCTI and received using INCT. To support efficient runtime checks that the type, length or structure of output data matches that expected by the inputer, CHKCT and CHKCTI instructions are provided. The CHKCT instruction inputs and discards a token provided that the input token matches its operand; otherwise it traps. The normal IN and INT instructions trap if they encounter a control token. To input a control token INCT is used; this traps if it encounters a data token.

The END control token is one of the 12 tokens which can be sent using OUTCTI and checked using CHKCTI. By following each message output with an OUTCTI *c*, END and each input with a CHKCTI *c*, END it is possible to check that the size of the message is the same as the size of the message expected by the inputting thread. To perform synchronised communication, the output message should be followed with (OUTCTI *c*, END; CHKCTI *c*, END) and the input with (CHKCTI *c*, END; OUTCTI *c*, END).

Another control token is PAUSE. Like END, this causes the route through the interconnect to be disconnected. However the PAUSE token is not delivered to the receiving thread. It is used by the outputting thread to break up long messages or streams, allowing the interconnect to be shared efficiently. The remaining control tokens are used for runtime checking and for signalling the type of message being received; they have no effect on the interconnect. Note that in addition to END and PAUSE, ten of these can be efficiently handled using OUTCTI and CHKCTI.

A control token takes up a single byte of storage in the channel. On the receiving end the software can test whether the next token is a control token using the TESTCT instruction, which waits until at least one token is available. It is also possible to test whether the next word contains a control token using the TESTWCT instruction. This waits until a whole word of data tokens has been received (in which case it returns 0) or until a control token

has been received (in which case it returns the byte position after the position of the byte containing the control token).

Channel ends have a buffer able to hold sufficient tokens to allow at least one word to be buffered. If an output instruction is executed when the channel is too full to take the data then the thread which executed the instruction is paused. It is restarted when there is enough room in the channel for the instruction to successfully complete. Likewise, when an input instruction is executed and there is not enough data available then the thread is paused and will be restarted when enough data becomes available.

Note that when sending long messages to a shared channel, the sender should send a short request and then wait for a reply before proceeding as this will minimise interconnect congestion caused by delays in accepting the message.

When a channel end c is no longer required, it can be freed using a FREER c instruction. Otherwise it can be used for another message.

It is sometimes necessary to determine the identifier of the destination channel end $c2$ stored in channel end $c1$. For example, this enables a thread to transmit the identifier of a destination channel end it has been using to a thread on another processor. This can be done using the GETD instruction. It is also useful to be able to determine quickly whether a destination channel end $c2$ stored in channel end $c1$ is on the same processor as $c1$; this makes it possible to optimise communication of large data structures where the two communicating threads are executed by the same processor.:

```
TESTLCL    d <- islocal(c)           test destination local
```

13 Locks

Mutual exclusion between a number of threads can be performed using *locks*. A lock is allocated using a GETR l, LOCK instruction. The lock is initially *free*. It can be *claimed* using an IN instruction and freed using an OUT instruction.

When a thread executes an IN on a lock which is already claimed, it is paused and placed in a queue waiting for the lock. Whenever a lock is freed by an OUT instruction and the lock's queue is not empty, the next thread in the queue is unpaused; it will then succeed in claiming the lock.

When inputting from a lock, the IN instruction always returns the lock identifier, so the same register can be used as both source and destination operand. When outputting to a lock, the data operand of the OUT instruction is ignored.

When the lock is no longer needed, it can be freed using a FREER *l* instruction.

14 Timers

Each XCORE executes instructions at a speed determined by its own clock input. In addition, it provides a reference clock output which ticks at a standard frequency of 100MHz. A set of programmable timers is provided and all of these can be used by threads to provide timed program execution relative to the reference clock.

14.1 Using timers

The processor has a set of timers that can be used to wait for a time. The current time can be input from any timer, or it can be obtained by using GETTIME:

```
GETTIME    d <- current time              get current time
```

Each timer can be used by a thread to read its current time or to wait until a specified time. A timer is allocated using the GETR t, TIMER instruction. It can be configured using the SETC instruction; the only two modes which can be set are UNCOND and AFTER.

mode	effect
UNCOND	timer *always* ready; inputs complete immediately
AFTER	timer ready when its current time is *after* its DATA value

In unconditional mode, an IN instruction reads the current value of the timer. In AFTER mode, the IN instruction waits until the value of its current time is after (later than) the value in its DATA register. The value can be set using a SETD instruction. Timers can also be used to generate events as described below.

A set of programmable clocks is also provided and each can be used to produce a clock output to control the action of one or more ports and their associated port timers. The ports are connected to a clock using the SETCLK instruction.:

```
SETCLK     clock(d) <- s                set clock source
```

Each port *p* which is to be clocked from a clock *c* can be connected to it by executing a SETCLK *p*, *c* instruction.

Each clock can use a one bit port as its clock source. A clock *c* which is to use a port *p* as its clock source can be connected to it by executing a SETCLK *p*, *c* instruction. Alternatively, a clock may use the reference clock as its clock source (by SETCLK *p*, REF). In either case the clock can be configured to divide the frequency using an 8-bit divider. When this is set to 0, the clock passes directly to the output. The falling edge of the clock is used to perform the division. Hence a setting of 1 will result in an output from the clock which changes each falling edge of the input, halving the input frequency *f*; and a setting of *n* will produce an output frequency of *f/2n*. The division factor is set using the SETD instruction. The lowest eight bits of the operand are used and the rest ignored.

To ensure that the timers in the ports which are attached to the same clock all record the same time, the clock should be started using a SETC *c*, START instruction *after* the ports have all been attached to the clock. All of the clocks are initially stopped and a clock can be stopped by a SETC *c*, STOP instruction.

The data output on the pins of an output port changes state synchronously with the port clock. If several output ports are driven from the same clock, they will appear to operate as a single output port, provided that the processor is able to supply new data to all of them during each clock cycle. Similarly, the data input by an input port from the port pins is sampled synchronously with the port clock. If several input ports are driven from the same clock they will appear to operate as a single input port provided that the processor is able to take the data from all of them during each clock cycle.

The use of clocked ports therefore decouples the internal timing of input and output program execution from the operation of synchronous input and output interfaces.

15 Ports, Input and Output

Ports are interfaces to physical pins. A port can be used for *input* or *output*. It can use the reference clock as its port clock or it can use one of the programmable clocks. Transfers to and from the pins can be *synchronised* with the execution of input and output instructions, or the port can be configured to *buffer* the transfers and to convert automatically between serial and parallel form. Ports can also be *timed* to provide precise

timing of values appearing on output pins or taken from input pins. When inputting, a *condition* can be used to delay the input until the data in the port meets the condition. When the condition is met the captured data is *time stamped* with the time at which it was captured.

The port clock input is initially the reference clock. It can be changed using the SETCLK instruction with a clock ID as the clock operand. This port clock drives the port timer and can also be used to determine when data is taken from or presented to the pins.

A port can be used to generate events and interrupts when input data becomes available as described below. This allows a thread to monitor several ports, channels or timers, only servicing those that are ready.

15.1 Input and Output

Each port has a *transfer register*. The input and output instructions used for channels, IN and OUT, can also be used to transfer data to and from a port transfer register. The IN instruction zero-extends the contents of a port transfer register and transfers the result to an operand register. The OUT instruction transfers the least significant bits from an operand register to a port transfer register.

Two further instructions, INSHR and OUTSHR, optimise the transfer of data. The INSHR instruction shifts the contents of its destination register right, filling the left-most bits with the data transferred from the port. The OUTSHR instruction transfers the least significant bits of data from its source register to the port and shifts the contents of the source register right.:

```
OUTSHR  p <: s[0..trwidth(p)];               output to port
        s <- s >> trwidth(p)                 and shift

INSHR   s <- s >> trwidth(p);                shift and
        p :> s[(bpw-trwidth(p))..bpw-1]      input from port
```

The transfer register is accessed by the processor; it is also accessed by the port when data is moved to or from the pins. When the processor writes data into the transfer register it *fills* the transfer register; when the processor takes data from the transfer register it *empties* the transfer register.

15.2 Port Configuration

A port is initially OFF with its pins in a high impedance state. Before it is used, it must be configured to determine the way it interacts with its pins, and set ON, which also has the effect of starting the port. The port can subsequently be stopped and started using SETC p, STOP and SETC p, START; between these the port configuration can be changed.

The port configuration is done using the SETC instruction which is used to define several independent settings of the port. Each of these has a default mode and need only be configured if a different mode is needed. The effect of the SETC mode settings is described below. The **bold** entry in each setting is the default mode.

mode	effect
NOREADY	no ready signals are used
HANDSHAKEN	both ready input and ready output signals are used
STROBED	one ready signal is used (output on master, input on slave)
SYNCHRONISED	processor synchronises with pins
BUFFERED	port buffers data between pins and processor
SLAVE	port acts as a slave
MASTER	port acts as a master
NOSDELAY	input sample not delayed
SDELAY	input sample delayed half a clock period
DATAPORT	port acts as normal
CLOCKPORT	the port outputs its source clock
READYPORT	the port outputs a ready signal
DRIVE	pins are driven both high and low
PULLDOWN	pins have a weak pull-down on input, on output pins are driven high only, high impedance otherwise
PULLUP	pins have a weak pull-up on input, on output pins are driven low only, high impedance otherwise
KEEP	pins keep their value on input
NOINVERT	data is not inverted
INVERT	data is inverted

The DRIVE, PULLDOWN and PULLUP modes determine the way the pins are driven when outputting, and the way they are pulled when inputting. The CLOCKPORT, READYPORT and INVERT settings can only be used with 1-bit ports.

Initially, the port is ready for input. Subsequently, it may change to output data when an output instruction is executed; after outputting it may change back to inputting when an input instruction is executed.

It is sometimes useful to read the data on the pins when the port is outputting; this can be done using the PEEK instruction:

```
PEEK       d <- pins(p)        read port pins
```

15.3 Configuring Ready and Clock Signals

A port can be configured to use *ready input* and *ready output* signals.

A port's ready input signal is input by an associated one-bit port. This association is made using the SETRDY instruction.:

```
SETRDY      ready(p) <- s      set source of port ready input
```

A port's ready output signal is output by another associated one-bit port. A one-bit port *r* which is to be used as a ready output must first be configured in READYPORT mode by

SETC *r*, READYPORT. This ready port *r* can then be associated with a port *p* by SETRDY *r*, *p*.

A one-bit port can be used to output a clock signal by setting it into CLOCKPORT mode; its clock source is set using the SETCLK instruction.

When a 1-bit port is configured to be in CLOCKPORT or READYPORT mode, the drive mode and invert mode are configurable as normal.

15.4 NOREADY mode

If the port is in NOREADY mode, no ready signals are used and data is moved to and from the pins either asynchronously (at times determined by the execution of input and output instructions) or synchronously with the port clock, irrespective of whether the port is in MASTER or SLAVE mode.

At most one input or output is performed per cycle of the port clock.

15.5 HANDSHAKEN mode

In HANDSHAKEN mode, ready signals are used to control when data is moved to or from a port's pins.

A port in MASTER HANDSHAKEN mode initiates an output cycle by moving data to the pins and asserting the ready output (request); it then waits for the ready input (reply) to be asserted. It initiates an input cycle by asserting the ready output (request) and waiting for the ready input (reply) to be asserted along with the data; it then takes the data.

A port in SLAVE HANDSHAKEN mode waits for the ready input (request) to be asserted. It performs an input cycle by taking the data and asserting the ready output (reply); it performs an output cycle by moving data to the pins and asserting the ready output (reply).

The ready signals accompany the data in each cycle of the port clock. The *falling edge* of the port clock initiates the set up of data or a change of port direction; the port timer also advances on this edge. On output, the data and the ready output will be valid on the *rising edge* of the port clock. On input, data and the ready input will be sampled on the rising edge of the port clock unless the port is configured as SDELAY, in which case they are sampled on the falling edge.

15.6 STROBED mode

In STROBED mode only one ready signal is used and the port can be in MASTER or SLAVE mode. A MASTER port asserts its ready output and the slave has to keep up; a SLAVE port has to keep up with the ready input.

Note that a port in NOREADY mode behaves in the same way as a port in STROBED mode which is always ready.

15.7 The Port Timer

A port has a timer which can be used to cause the transfer of data to or from the pins to take place at a specified time. The time at which the transfer is to be performed is set using the SETPT (set port time) instruction. Timed ports are often used together with time-stamping as this allows precise control of response times.:

```
SETPT     porttime(p) <- s              set port time
CLRPT     clearporttime(p)              clear port time
GETTS     d <- timestamp(p)             get port timestamp
```

The CLRPT instruction can be used to cancel a timed transfer.

The timestamp which is set when a port becomes ready for input can be read using the GETTS instruction.

15.8 Conditions

A port has an associated *condition* which can be used to prevent the processor from taking input from the port when the condition is not met. The conditions are set using the SETC instruction. The value used for comparison in some of the conditions is held in the port data register, which can be set using the SETD instruction.

mode	port ready condition
NONE	no condition
EQ	value on pins *equal to* port data register value
NEQ	value on pins *not equal to* port data register value

The simplest condition is NONE. The other conditions all involve comparing the value from the pins with the value in the port data register.

When the condition is met a timestamp is set and the port becomes ready for input.

When a port is used to generate an event, the data which satisfied the condition is held in the transfer register and the timestamp is set. The value returned by a subsequent input on the port is guaranteed to meet the condition and to correspond to the timestamp even if the value on the port has changed.

15.9 Synchronised Transfers

A port in SYNCHRONISED mode ensures that the signalling operation of the port pins is synchronised with the processor instruction execution.

When a SETPT instruction is used, the movement of data between the pins and the transfer register takes place when the current value of the port timer matches the time specified with the SETPT instruction.

If the port is used for output and the transfer register is full, the SETPT instruction will pause until the transfer register is empty. This ensures that the port time is not changed until the pending output has completed.

If a condition other than NONE is used the port will only be ready for input when the data in the transfer register matches the condition. If an input instruction is executed and the specified condition is not met, the thread executing the input will be paused until the condition is met; the thread then resumes and completes the input. The value of the port timer corresponding to the data in the transfer register when a port condition is met is recorded in the port timestamp register. The timestamp register is read at any time using the GETTS instruction.

15.10 Buffered Transfers

A port in BUFFERED mode buffers the transfer of data between the processor and the pins through the use of a *shift register*, which is situated between the transfer register and the pins. A buffered port can be used to convert between parallel and serial form using its shift register. The number of bits in the transfer register and the shift register determines the width of the transfers (the *transfer width*) between the processor and the port; this is a multiple of the *port width* (the number of pins) and can be set by the SETTW instruction.:

```
SETTW     width(p) <- s       set port transfer width
```

For a 32-bit word-length, the transfer width is normally 32, 8, 4 or 1 bit.

Note that in contrast to a synchronised transfer, where the transfer width and the port width are equal, the transfer width of a buffered transfer can differ from the port width.

On input, the shift register is full when *n* values have been taken from the *p* pins, where *n x p* is the transfer width; it will then be emptied to the transfer register ready for an input instruction. On output the shift register is filled from the transfer register and will be empty when *n* values have been moved to the *p* pins, where *n x p* is the transfer width.

The port operates as follows:

- **HANDSHAKEN**: A handshaken transfer only shifts data from the pins to the shift register on input when the shift register is not full; on output it only shifts data from the shift register to the pins when the shift register is not empty. On input, the shift register will become full if the processor does not input data to empty the transfer register; when the processor inputs the data, the transfer register is filled from the shift register and the shift register will start to be re-filled from the pins. On output, the shift register will become empty if the processor does fill the transfer register; when the processor outputs data to fill the transfer register, the shift register will be filled from the transfer register and the shift register will then start to be emptied to the pins.
- **STROBED SLAVE Input**: Data is shifted into the shift register from the pins whenever the ready input is asserted. Provided that the transfer register is empty, when the shift register is full the transfer register is filled from the shift register. When the processor executes an input instruction to take data from the transfer register, the transfer register is emptied.

  If the processor does not take the data from the transfer register by the time the shift register is next full, data will continue to be shifted into the shift register and only the most recent values will be kept; as soon as an input instruction empties the transfer register the transfer register will be filled from the shift register.
- **STROBED SLAVE Output**: Data is shifted out to the pins whenever the ready input is asserted. Provided that the transfer register is full, when the shift register is empty, it is filled from the transfer register. When the processor executes an output instruction it fills the transfer register.

  If the processor has not filled the transfer register by the time the shift register is next empty, the data is held on the pins. As soon as the processor executes and output instruction it fills the transfer register; the shift register is then filled from the transfer register and the it will start to be emptied to the pins.
- **STROBED MASTER**: The transfer operates in the same way as a handshaken transfer in which the ready input is always asserted.

The SETPT instruction can be used to delay the movement of data between the shift register and the transfer register until the current value of the port timer matches the time specified.

Note that this can be used to provide synchronisation with a stream of data in a BUFFERED port in NOREADY mode, because exactly one item will be shifted to or from the pins in each clock cycle.

If the port is outputting and the transfer register is full the SETPT instruction will pause until it is empty. This ensures that the port time is not changed until the pending output has completed.

The port condition can be used to locate the first item of data on the pins that matches a condition. If the condition is different from NONE, data will be held in the shift register until the data meets the condition; the data is then moved to the transfer register, the timestamp is set and the port changes the condition to NONE so that data can continue to fill the shift register in the normal way. Only the top port-width bits of the shift register are used for comparison when the condition is checked.

15.11 Partial Transfers

Buffered transfers permit data of less than the transfer width to be moved between the shift register and the transfer register. The length of the items in a buffered transfer can be set by a SETPSC instruction, which sets the port shift register count. On input, this will cause the shift register contents to be moved to the transfer register when the specified amount of data has been shifted in; on output it will cause only the specified amount of data to be shifted out before the shift register is ready to be re-loaded. This is useful for handling the first and last items in a long transfer.:

```
SETPSC       shiftcount(p) <- s       set port shift register count
```

A buffered input can be terminated by executing an ENDIN instruction which returns the number of items buffered in the port (which will include the shift register and transfer register contents) and also sets the port shift register count to the amount of data remaining in the shift register, enabling a following input to complete.:

```
ENDIN        d <- buffercount(p)      end input
```

To optimise the transfer of part-words two further instructions are provided:

```
OUTPW        shiftcount(p) <- q;      output part word
             p <: s
OUTPWI       shiftcount(p) <- bitp;   output part word
             p <: s
INPW         shiftcount(p) <- bitp;   input part word
             p :> d
```

These encode their immediate operand in the same way as the shift instructions.

15.12 Changing Direction

A SYNCHRONISED port can change from input to output, or from output to input. The direction changes at the start of the next setup period. For a transfer initiated by a SETPT instruction, the direction will be input unless an output is executed before the time specified by the SETPT instruction.

A BUFFERED port can change direction only after it has completed a transfer. This is done by stopping and re-starting the port using SETC p, STOP and SETC p, START instructions.

16 Memory model

The XS3 architecture supports three forms of memory:

1. Internal memory that is normally used to store code and data. Accesses to this memory incur no latency. In addition to an internal RAM there may be an internal ROM for booting and debugging
2. An optional software programmable memory. Accesses to this memory may incur a delay.
3. An optional external memory. Accesses to this memory may incur a delay.

The address space is a single linear address space that is mapped onto the above memories. The precise locations of each of the three types of memory are implementation dependent, and may be settable from software. More details are in the product datasheet. An example memory map may be as follows:

- Internal ROM: 0xfff0 0000 … 0xfff0 07ff
- Software defined memory: 0x4000 0000 … 0x7fff ffff
- External RAM: 0x1000 0000 … 0x1fff ffff
- Internal RAM: 0x0008 0000 … 0x000f ffff

16.1 Internal memory

The internal RAM location can be set using the SETPS instruction. The implementation may require the start address to be aligned with, for example, the size of the memory.

Internal RAM is the primary location where code and data are stored, and programs typically are stored in the lower locations of memory, the stacks in the higher locations of RAM, and the data and any heap in between.

The internal ROM is used for booting and debugging. It has a fixed location and fixed entry points for the boot-rom and debug-rom. These entry points are implementation dependent.

16.2 External memory

The external memory is an optional interface that can be used to connect large memories. The details on how the memory are connected physically, and how it is enabled are implementation dependent and documented in the product datasheet. When enabled, load/store operations and instruction fetches to that address range will be served from external memory.

The external memory is served through a small cache. This cache is guaranteed to have room for at least *2 x bpv* bits. The memory cannot be shared between processors, and care should be taken that memory performance will be much lower than internal memory, especially when multiple threads use external memory simultaneously.

Three instructions manage the cache:

```
FLUSH       Flush all cache lines
            All memory is updated with the cache contents
INVALIDATE  Invalidate all cache lines
PREFETCH    Starts a fetch of address r11
```

16.3 Software defined memory

Software defined memory is an optional part of the address range that can be used to emulate other forms of memory, for example, it can be used to execute-from-flash, or to map data from a remote tile. Software defined memory uses the same small cache that serves the external memory.

The software defined memory has to be enabled by enabling the SWMEM resource. When enabled, load/store operations and instruction fetches to that address range will be served from the software defined memory.

Accesses to the software defined memory are served by the cache. If the cache contains the requested data, then it be provided to the thread without any delay. Otherwise, the thread will be paused, and the Software Defined Memory (SWMEM) resource will become ready. Any thread can be waiting to take an event or interrupt on the SWMEM resource, and fill the cache. The filling thread must input the missing memory address from the SWMEM resource, locate the data belonging to this address, and refill the cache by writing the data to the address. When the refill is complete, the SWMEM resource is restarted to signal that the paused thread can safely continue:

```
LDC  r0, SWMEM // The SWMEM resource ID
IN   r0, r1    // Input the address on which the program missed
...            // Locate the data belonging to that address
STW            // Store the data (this may be multiple stores)
SETC r0, START // Start the SWMEM resource ID.
```

17 Events, Interrupts and Exceptions

Events and interrupts allow timers, ports and channel ends to automatically transfer control to a pre-defined event handler. The resources generate events by default and must be reconfigured using a SETC instruction in order to generate interrupts. The ability of a thread to accept events or interrupts is controlled by information held in the thread status register (sr), and may be explicitly controlled using SETSR and CLRSR instructions with appropriate operands.:

```
SETSR      sr  <- sr || u16          set thread state
CLRSR      sr  <- sr && ~ u16        clear thread state
GETSR      r11 <- sr && u16          get thread state
```

The operand of these instructions should be one (or more) of:

```
EEBLE  enable events
IEBLE  enable interrupts
INENB  determine if thread is enabling events
ININT  determine if thread is in interrupt mode
HIPRI  set thread to high priority mode
FAST   set thread to fast mode
KEDI   set thread to switch to dual issue on kernel entry
```

17.1 Events

A thread normally enables one or more events and then waits for one of them to occur. Hence, on an event all the thread's state is valid, allowing the thread to respond rapidly to the event. The thread can perform input and output operations using the port, channel or timer which gave rise to an event whilst leaving some or all of the event information unchanged. This allows the thread to complete handling an event and immediately wait for another similar event.

Timers, ports and channel ends all support events, the only difference being the ready conditions used to trigger the event. The program location of the event handler must be set prior to enabling the event using the SETV instruction. The SETEV instruction can be used to set an environment for the event handler; this will often be a stack address containing data used by the handler. Timers and ports have conditions which determine

when they will generate an event; these are set using the SETC and SETD instructions. Channel ends are considered ready as soon as they contain enough data.

Event generation by a specific port, timer or channel can be enabled using an event enable unconditional (EEU) instruction and disabled using an event disable unconditional (EDU) instruction. The event enable true (EET) instruction enables the event if its condition operand is true and disables it otherwise; conversely the event enable false (EEF) instruction enables the event if its condition operand is false, and disables it otherwise. These instructions are used to optimise the implementation of guarded inputs.:

```
SETV  vector(r) <- s                    set event vector
SETEV envector(r) <- s                  set event environment vector

SETD  data(r) <- s                      set resource data
GETD  d <- data(r)                      get resource data
SETC  cond(r) <- s                      set event condition

EET   enb(r) <- c; thread(r) <- tid     event enable true
EEF   enb(r) <- ! c; thread(r) <- tid   event enable false
EDU   enb(r) <- false; thread(r) <- tid event disable
EEU   enb(r) <- true; thread(r) <- tid  event enable
```

Having enabled events on one or more resources, a thread can use a WAITEU, WAITET or WAITEF instruction to wait for at least one event. The WAITEU instruction waits unconditionally; the WAITET instruction waits only if its condition operand is true, and the WAITEF waits only if its condition operand is false.:

```
WAITET  if c  then  eeble(tid) <- true  event wait if true
WAITEF  if !c then  eeble(tid) <- true  event wait if false
WAITEU  eeble(tid) <-  true             event wait
```

This may result in an event taking place immediately with control being transferred to the event handler specified by the corresponding event vector with events disabled by clearing the thread's \$eeble\$ flag. Alternatively the thread may be paused until an event takes place with the \$eeble\$ flag enabled; in this case the \$eeble\$ flag will be cleared when the event takes place, and the thread resumes execution.:

```
event   ed <- ev(res);
        pc <- v(res);
        sr[inenb] <- false;
        sr[eeble] <- false;
        sr[waiting] <- false
```

Note that the environment vector is transferred to the event data register, from where it can be accessed by the GETED instruction. This allows it to be used to access data associated with the event, or simply to enable several events to share the same event vector.

To optimise the responsiveness of a thread to high priority resources the SETSR EEBLE instruction can be used to enable events before starting to enable the ports, channels and timers. This may cause an event to be handled immediately, or as soon as it is enabled. An enabling sequence of this kind can be followed either by a WAITEU instruction to wait for one of the events, or it can simply be followed by a CLRSR EEBLE to continue execution when no event takes place. The WAITET and WAITEF instructions can also be used in conjunction with a CLRSR EEBLE to conditionally wait or continue depending on a guarding condition. The WAITET and WAITEF instructions can also be used to optimise the common case of repeatedly handling events from multiple sources until a terminating condition occurs.

All of the events which have been enabled by a thread can be disabled using a single CLRE instruction. This disables event generation in all of the ports, channels or timers which have had events enabled by the thread. The CLRE instruction also clears the thread's `eeble` flag.:

```
CLRE  eeble(tid) <-  false;          disable all events
      inenb(tid) <-  false;          for thread
      forall res
          if  (thread(res) == tid && event(res))
              then  enb(res) <- false
```

Where enabling sequences include calls to input subroutines, the SETSR INENB instruction can be used to record that the processor is in an enabling sequence; the subroutine body can use GETSR INENB to branch to its enabling code (instead of its normal inputting code). INENB is cleared whenever an event occurs, or by the CLRE instruction.

17.2 Interrupts

In contrast to events, interrupts can occur at any point during program execution, and so the current pc and sr (and potentially also some or all of the other registers) must be saved prior to execution of the interrupt handler. Interrupts are taken *between* instructions, which means that in an interrupt handler the previous instruction will have been completed, and the next instruction is yet to be executed on return from the interrupt. This is done using the spc and ssr registers. Any interrupt and exception causes the pc and sr registers to be saved into spc and ssr, and the status register to be modified to indicate that the processor is running in kernel mode:

```
kernelentry
          ssr <- sr;
          sed <- ed;
          sr[bit  di] <- sr[bit  kedi];
          sr[bit  eeble] <- false;
          sr[bit  ieble] <- false;
```

On an interrupt generated by resource `r` the following occurs automatically:

```
interrupt
          spc <- pc
          kernelentry ;                kernelentry is defined above
          ed <- ev(res)
          pc <- v(res);
          sr[bit  inint] <- true;
          sr[bit  waiting] <- false;
```

On kernel entry the DI bit is saved in the `ssr` register, whereupon DI is set according to the KEDI (dual-issue-in-kernel) bit in the status register. This enables exception handlers to be written in either SI or DI code as required. When in kernel mode, the kernel can switch between SI and DI mode as usual using DUALENTSP/ENTSP. On return from the kernel call, KRET, the DI bit is restored from `ssr`.

When the handler has completed, execution of the interrupted thread can be performed by a KRET instruction, this restores the DI bit from `spc`:

```
KRET   pc <- spc && ~ 1;         return from interrupt
       sr <- ssr
       ed <- sed
```

17.3 Exceptions

Exceptions which occur when an error is detected during instruction execution are treated in the same way as interrupts except that they transfer control to a location defined relative to the thread's kernel entry point kep register:

```
except spc <- pc_old;            any exception
       kernelentry ;
       pc <- kep;
       et <- exceptiontype;
       ed <- exceptiondata;
```

kernelentry is defined in *Interrupts* The exception handler resides on the address stored in **kep**. The handler can run in dual or single issue mode, depending on the **kedi** bit in the status register. Exception types are listed below:

Exception	et	Meaning
ET_LINK_ERROR	1	Incorrect use of channel
ET_ILLEGAL_PC	2	Unaligned program counter
ET_ILLEGAL_INSTRUCTION	3	Illegal opcode
ET_ILLEGAL_RESOURCE	4	Illegal use of resource
ET_LOAD_STORE	5	Unaligned memory access
ET_ILLEGAL_PS	6	Undefined PS register
ET_ARITHMETIC	7	Arithmetic error
ET_ECALL	8	Assertion failed
ET_RESOURCE_DEP	9	Illegal resource use
ET_KCALL	15	KCALL executed

When in dual issue mode, an exception in one lane will abort any instruction in the other lane. If two instructions would both cause an exception, then only one exception is taken. The *et* register will hold the data as specified above, but the lane in which the exception occurred is encoded in bit *4* of *et*. If the thread is in dual-issue mode, and two instructions were issued, and the exception occurred in the resource lane, then this bit is set to *1*. In all other cases bit *4* of *et* will be set to *0*.

A program can force an exception as a result of a software detected error condition using ECALLT, ECALLF, or ELATE:

```
ECALLT  if  e  then                       error on true
          except(ET_ECALL,e)

ECALLF  if  !e  then                      error on true
          except(ET_ECALL,e)

ELATE  if  s  after  t(current) then      error if late
          except(ET_ECALL,s)
```

except is defined on *Exceptions*. These have the same effect as hardware detected exceptions, transferring control to the same location and indicating that an error has occurred in the exception type (et) register. If in dual issue mode, any instruction in the other lane will be aborted on taking an exception.

A program can explicitly cause entry to a handler using one of the kernel call instructions. These have a similar effect to exceptions, except that they transfer control to a location defined relative to the thread's **kep** register:

```
KCALLI              kernelentry ;
                    spc <- pc
                    et <- ET_KCALL;
                    ed <- u6;
                    pc <- kep + 64;
KCALL               kernelentry ;
                    spc <- pc
                    et <- ET_KCALL;
                    ed <- s;
                    pc <- kep + 64;
```

In dual issue mode KCALL will complete as normal; it is safe to dual-issue a KCALL instruction with any other instruction. If the instruction in the other lane causes an exception, the thread will continue with the exception and abort the KCALL.

The **spc**, **ssr**, **et** and **sed** registers can be saved and restored directly to the stack.:

```
LDSPC    spc <- mem[sp + 1 x  Bpw]        load exception pc
STSPC    mem[sp + 1 x  Bpw] <- spc        store exception pc
LDSSR    ssr <- mem[sp + 2 x  Bpw]        load exception sr
STSSR    mem[sp + 2 x  Bpw] <- ssr        store exception sr
LDSED    sed <- mem[sp + 3 x  Bpw]        load exception data
STSED    mem[sp + 3 x  Bpw] <- sed        store exception data
STET     mem[sp + 4 x  Bpw] <- et         store exception type
```

In addition, the `et` and `ed` registers can be transferred directly to a register.:

```
GETET               r11 <- et                get exception type
GETED               r11 <- ed                get exception data
```

A handler can use the KENTSP instruction to save the current stack pointer into word 0 of the thread's kernel stack (using the kernel stack pointer `ksp`) and change stack pointer to point at the base of the thread's kernel stack. KRESTSP can then be used to restore the stack pointer on exit from the handler.:

```
KENTSP  n    mem[ksp] <- sp;            switch to kernel stack
             sp <- ksp - n x  Bpw
KRESTSP n    ksp <- sp + n x  Bpw;      switch from kernel stack
             sp <- mem[ksp]
```

The `kep` can be initialised using the SETKEP instruction; the `ksp` can be read using the GETKSP instructions.:

```
SETKEP        kep <- r11       set kernel entry point
GETKSP        r11 <- ksp       get kernel stack pointer
```

The kernel stack pointer is initialised by the boot-ROM to point to a safe location near the last location of RAM - the last few locations are used by the JTAG debugging interface. `ksp` can be modified by using a sequence of SETSP followed by KRESTSP.

18 Initialisation and Debugging

The state of the processor includes additional registers to those used for the threads.

register	use
dspc	debug save pc
dssr	debug save sr
dssp	debug save sp
dtype	debug cause
dtid	thread identifier used to access thread state
dtreg	register identifier used to access thread state
DEBUG	flag that indicates that processor is in debug mode

All of the processor state can be accessed using the GETPS and SETPS instructions:

```
GETPS    d <- state[s]      get processor state
SETPS    state[d] <- s      set processor state
```

To access the state of a thread, first SETPS is used to set `dtid` and `dtreg` to the thread identifier and register number within the thread state. The contents of the register can then be accessed by:

```
DGETREG  d <- dtreg(dtid)  get thread register
```

The debugging state is entered by executing a DCALL instruction, by an instruction that triggers a watchpoint or a breakpoint, or by an external asynchronous DEBUG event (for

example caused by asserting a DEBUG pin). During debug, thread `0` executes the debug handler, all other threads are frozen. The debugging state is exited on DRET, which causes thread `0` to resume at its saved PC, and all other threads to start where they were stopped. Entry to a debug handler operates in a manner similar manner to an interrupt:

```
debugentry dspc <- pc(t0);
           dssr <- sr(t0);
           pc(t0) <- debugentrypoint
           sr(t0)[bit  inint] <- true
           sr(t0)[bit  di] <- false;
           sr(t0)[bit  eeble] <- false;
           sr(t0)[bit  ieble] <- false
           sr(t0)[bit  waiting] <- false
           DEBUG <- 1
```

On an external, asynchronous, DEBUG event, the processor will always enter the debug state as follows:

```
DEBUG event     debugentry
                dtype <- debugcause
```

The DCALL instruction has the same effect:

```
DCALL   debugentry
        dtype <- dcallcause        debug call (breakpoint)

DRET    pc(t0) <- dspc;           return from debug
        sr(t0) <- dssr;
        DEBUG <- 0

DENTSP  dssp <- sp;                debug save stack pointer
        sp <- ramend

DRESTSP sp <- dssp                 debug restore stack pointer
```

On entering debug mode the DI bit is saved in the `dspc` register, and it is cleared. Debug mode is always entered in single issue mode, but the debugger can switch to dual-issue mode if required using DUALENTSP. On return from the debugger, DI is restored from the `dspc`

Watchpoints and instruction breakpoints are supported by means of SETPS and GETPS instructions. An instruction breakpoint is an address that triggers a *DCALL* on a PC being equal to the value in the instruction break point. A data watchpoint is a pair of addresses `l` and `h`, and a condition that triggers a *DCALL* on stores and or loads to specific memory addresses. If the condition is set to `INRANGE`, then a debug is triggered if a thread access address `x` where `l <= x <= h`. If the condition is set to `NOTINRANGE`, then a debug is triggered if a thread access address `x` where `x <= l || x >= h`.

- When the processor is not in *debug-mode*, none of the debug information is writable, except for the DEBUG registers that brings the processor into debug mode.
- When the processor is not in *debug-mode*, none of the debug values can be read except the PC and SR values, in order to support profiling.

19 Specialised Instructions

19.1 Long arithmetic

The long arithmetic instructions support signed and unsigned arithmetic on multi-word values. The long subtract instruction (LSUB) enables conversion between long signed and long unsigned values by subtracting from long `0`. The long multiply and long divide operate on unsigned values.

The long add instruction is intended for adding multi-word values. It has a carry-in operand and a carry-out operand. Similarly, the long subtract instruction is intended for subtracting multi-word values and has a borrow-in operand and a borrow-out operand.:

```
LADD    d <- l + r + c[bit  0];            add with carry
        e <- carry(l + r + c[bit  0])

LSUB    d <- l - r - b[bit  0];            subtract with borrow
        e <- borrow(l - r - b[bit  0])
```

The long multiply instruction multiplies two of its source operands, and adds two more source operands to the result, leaving the unsigned double length result in its two destination operands. The result can always be represented within two words because the largest value that can be produced is `(B-1) x (B-1) + (B-1) +(B-1) = B^2 - 1` where `B = 2^bpw`. The two carry-in operands allow the component results of multi-length multiplications to be formed directly without the need for extra addition steps.:

```
LMUL     d <- ((l  x  r) + s + t)[bits 2 x bpw-1..bpw]    long multiply
         e <- ((l  x  r) + s + t)[bits bpw-1..0]
```

The long division instruction (LDIV) is very similar to the short unsigned division instruction, except that it returns the remainder as well as the result; it also allows the remainder from a previous step of a multi-length division to be loaded as the high part of the dividend.:

```
LDIV     d <- (l:m) // r                long divide unsigned
         e <- (l:m)  mod  r
```

An ET_ARITHMETIC exception is raised if the result cannot be represented as a single word value; this occurs when `l <= r`. Note that this instruction operates correctly if the most significant bit of the divisor is 1 and the initial high part of the dividend is non-zero. A (fairly) simple algorithm can be used to deal with a double length divisor. One method is to normalise the divisor and divide first by the top 32 bits; this produces a very close approximation to the result which can then be corrected.

The long extract and insert instructions perform long shift and mask operations. LEXTRACT extracts a selection of bits from two words at a given offset; a sequence of LEXTRACT instructions can be used to implement a rotate, long shift, and misaligned loads. An LSATS followed by an LEXTRACT can be used to extract a word from the result of a MACCS (see the next subsection). LINSERT performs the inverse operation of LEXTRACT and inserts a bit pattern into a double word.:

```
LEXTRACT d <- (l:r)[bit  bitp+x-1..x]           extract  word
LINSERT  m = ((1 << bitp) - 1) << s             insert word
         d:e <- ((d:e) & ~ m) |  ((x << s) & m)
```

LEXTRACT assumes that `x` is a signed value. Where `bitp+x` exceeds the double word length, the top most bit `l` will be replicated (ie, this is assumed to be a signed value). Where `x` is negative, zero bits are used.

19.2 Long Integer multiply accumulate

The multiply-accumulate instructions perform a double length accumulation of products of single length operands:

```
MACCU    s <- ((l  x  r) + (s:t))[bits  2 x  bpw-1..bpw];     long multiply
         t <- ((l  x  r) + t)[bits  bpw-1..0]                acc unsigned

MACCS    s <- ((l  x  r) + (s:t))[bits  2 x  bpw-1..bpw];     long multiply
         t <- ((l  x  r) + t)[bits  bpw-1..0]                acc signed
LSATS    if s:t > 2^(l+bpw)-1                          Saturate signed
         then s:t <- 2^(l+bpw) - 1;
         elif s:t < -2^(l+bpw)
         then s:t <- -2^(l+bpw);
LSATSI   if s:t > 2^(bitp+bpw)-1                       Saturate signed immediate
         then s:t <- 2^(bitp+bpw) - 1;
         elsif s:t < -2^(bitp+bpw)
         then s:t <- -2^(bitp+bpw);
```

The MACCU instruction multiplies two unsigned source operands to produce a double length result which it adds to its unsigned double length accumulator operand held in two other operands. Similarly, the MACCS instruction multiplies two signed source operands to produce a double length result which it adds to its signed double length accumulator operand held in two other operands. The LSATS instruction saturates a number that is outside the range `-2^(l+bpw)..2^(l+bpw)-1`.

19.3 Cyclic redundancy check

Cyclic redundancy check is performed using:

```
CRC32   for  step  =  0  for  bpw                    word cyclic
            if  (r[bit  0] = 1)                     redundancy check
            then r <- (s[bit  step] : r[bits  (bpw-1) ... 1]) @ p
            else r <- (s[bit  step] : r[bits  (bpw-1) ... 1])

CRC8    for  step  =  0 for  8                     8 step cyclic
            if  (r[bit  0] = 1)                     redundancy check
            then r <- (s[bit  step] : r[bits  31 ... 1]) @ p
            else r <- (s[bit  step] : r[bits  31 ... 1]);
        d <- s >> 8

CRCN   if  n > 32 then  cnt = 32                   n step cyclic
       else  cnt = n                                redundancy check
       for  step = 0  for  cnt
           if  (r[bit  0] = 1)
           then r <- (s[bit  step] : r[bits  31 ... 1]) @ p
           else r <- (s[bit  step] : r[bits  31 ... 1]);
```

The CRC8 instruction operates on the least significant 8 bits of its data operand, ignoring the most significant 24 bits. It is useful when operating on a sequence of bytes, especially where these are not word-aligned in memory. The CRCN instruction operates on the least significant bytes of its data operand; the fourth operand of CRCN, `t`, determines the number of bytes to fold into the CRC. If `t > 32` then 32 bits are be processed. This enables CRCN to be passed a bit count in a loop, and overrun in an unrolled loop.

The CRC32_INC instruction performs a CRC32 and a simultaneous increment on the second parameter:

```
CRC32_INC for  step  =  0  for  bpw              word cyclic redundancy check
             if  (r[bit  0] = 1)                 and increment register
             then r <- (s[bit step] : r[bits (bpw-1) ... 1]) @ p
             else r <- (s[bit step] : r[bits (bpw-1) ... 1])
          a <- a + bitp
```

20 Floating point arithmetic

Floating point instructions support four arithmetic instruction (FMACC, FMUL, FADD, and FSUB) and a number of logical instructions that support efficient implementation of other floating point operations. The instructions below operate on IEEE 754 floating point numbers that are represented in **bpw** bits (ie, single precision for **bpw=32**):

```
FMACC   d <- l + (r  *  s)              Floating-point MACC
FMUL    d <- l * r                      Floating-point multiply
FADD    d <- l+r                        Floating-point addition
FSUB    d <- l-r                        Floating-point subtraction
FSPEC   d <- 0, if IsANumber(s) && s > 0 Test on special values
        d <- 1, if s = +0.0
        d <- 2, if s = infinity
        d <- 3, if IsSignallingNAN(s)
        d <- 4, if IsANumber(s) && s < 0
        d <- 5, if s = -0.0
        d <- 6, if s = -infinity
        d <- 7, if IsQuietNAN(s)
FENAN   f  isNaN(s)  then               exception on NaN
           except(ET_ARITHMETIC,s)
FMANT    d <- mantissa(s)               Extract normalised mantissa
FSEXP    d <- exponent(s)               Extract unbiased exponent and sign
         e <- sign(s)
FMAKE    d <- (-1)^s * 2^e * x:y        Build a FP number
FGT      d <- x > y                     True if x greater than y
FLT      d <- x < y                     True if x less than y
FEQ      d <- x = y                     True if x equal to y
FUN      d <- unordered(x, y)           True if x and y are unordered
```

All instructions use the *roundTiesToEven* rounding direction as defined in IEEE 754: numbers are rounded to the nearest value, and if both values are equally near, the representation that ends in a zero bit is chosen. The FMACC instruction performs a multiply-accumulate rounding the result only after the addition. The FMUL instruction calculates the product of two numbers, FADD calculates the sum, and FSUB calculates the difference.

Not a number values are propagated through these operations with the Quiet bit set. If two or more input operands are not-a-number, then the first operand (from left to right) that is NaN is propagated through. Instructions that generate not-a-number will fill the least significant 22 bits of the NaN value with the least significant 22 bits of the address of the next instruction (the same value that is used for calculating the offset of a branch), and a zero sign bit, and the signalling bit set.

The FSPEC instruction can be used to efficiently deal with special floating point numbers (such as zero or NaN), prior to using FMANT and FSEXP to dissect a floating point number. The FSEXP and FMANT instructions raise a ET_ARITHMETIC trap if the input operand is not a number or infinite. For *normal* floating point numbers, the FMANT and FSEXP instruction will return the mantissa with the implicit one bit set, and FSEXP will return the exponent with the bias subtracted. For *subnormal* floating point numbers (non-zero), the FMANT and FSEXP instruction will normalise the mantissa by shifting it left by `k` bits, so that the mantissa has its `implicit` one bit set, and FSEXP will subtract `k` in addition to the bias. FMANT will produce a zero mantissa if the input is `+0.0` or `-0.0`.

The FMAKE instruction is provided to reassemble a floating point number. It has five operands: an output register and four input operands. The output register will on completion contain a floating point value. The first input register contains the sign bit in bit 0; the other bits are ignored. The second operand contains the exponent, it is a (typically small) signed value; FMAKE will add the bias term to this value prior to building a floating point value. The third and fourth input operands represent a double length unsigned mantissa. The FMAKE instruction is exactly the opposite of FSEXP/FMANT, except that an extra word of significant bits may be passed in, enabling large mantissas to be rounded. Infinity is produced if the result does not fit

Suppose we have a floating point value `f` represented with a sign `s`, exponent `e` and mantissa `m`. As per the IEEE standard, the exponent is stored with a bias term added, and the mantissa is stored without its leading '1'-bit. FSEXP will return `s` and `e`, and FMANT will return `m` including the implicit leading '1'-bit. Conversely, passing `s`, `e`, and `m` into FMAKE will reconstruct `f`. Subnormal floating point numbers (that in accordance with the standard have a minimal exponent and no implicit leading '1' bit) are normalised by FSEXP/FMANT, by reducing the exponent to below its nominal minimum value, and shifting the mantissa up to create the leading '1' bit in the correct place. Conversely, FMAKE will normalise its input mantissa values. Ie, the bit associated with the implicit leading one has a value of `1.0`, but if extra higher order bits are present, or the first '1' bit is lower, then the mantissa will be shifted (and rounded), and the exponent adjusted accordingly. The IEEE representation of infinite is created if the number cannot be represented by the limited range of the exponent. Underflow will cause 0 to be created.

The IEEE standard specifies a set of 20 relational operators. In hardware, these are recreated by means of five operations: four comparison instructions (FGT, FLT, FEQ, FUN) and an Exception on `Not A Number` instruction (FENAN).

The comparison instructions never raise an exception, and calculate the value of the predicate as per the IEEE standard: true yields 1 in the destination operand, false yields 0. If an exception is required, FENAN can be used.

21 Vector unit

The vector unit contains three vector registers (vC, vD, and vR) and a subsidiary register for maintaining the headroom and status (vCTRL). The vector registers are bpv (bits per vector) bits wide. Inside it are epv elements, each bpe bits wide; these depend on the data type being used, and they are defined as epv = bpv / bpe. The vector unit is controlled by instructions that execute in the memory lane. Many of the instructions are encoded in short form and execute in the memory lane, enabling dual issue to perform pointer management using the scalar registers r0..r11 in the resource lane.

Vector instructions go through the following stages:

1. Load values from the vector registers,
2. Perform a optional vector load or vector store operation,
3. Perform a point wise multiplication,
4. Perform pointwise or reducing additions,
5. Store the answer in the vector registers

Each of these steps is optional, ie, not all vector instructions perform a load from memory. The most intensive operation computes an inner product of a vector in memory and a vector in a register. The vector unit is optimised for fast convolution (FIR), complex multiplication, and headroom computation. The bpv parameter is an implementation defined parameter; a typical value is 256. The three registers of the vector unit are not general purpose registers as they appear as implicit operands in almost all instructions. In general vC is used to store some kernel or coefficients; vD is used to store data; and vR is used to hold output, prior to storing the output.

21.1 Control

The vCTRL register controls the width and type of the vector elements and also stores the available headroom. It contains 12 bist as follows:

Name	Bits	Constant	Val	Meaning
Magnitude	5..0			Maximum number of significant bits
Shift	7..6	VEC_SH0 VEC_SHL VEC_SHR	0 1 2	Do not shift on VLADSB and VFT* Shift left on VLADSB and VFT* Shift right on VLADSB and VFT*
Type	11..8	VSETCTRL_TYPE_INT32 VSETCTRL_TYPE_INT16 VSETCTRL_TYPE_INT8	0 1 2	Signed 32-bit integer Signed 16-bit integers Signed 8-bit integers

- The *magnitude* field contains the highest number of significant bits that has been stored using any of the VSTR, VSTD, or VSTC operations. Typically, the *magnitude* field is initialised to 0, and on each store operation it is updated to reflect the number of significant bits that has been stored. The number of significant bits is defined as the number of bits that are not equal to the sign bit.
- The *type* field defines the number of bits, number of elements, and data type of the operations. Three values are defined on the XS3 architecture for signed 32-bit opera-

tions (the default), signed 16-bit operations, and signed 8-bit operations. Signed 1-bit arithmetic is supported by a special instruction.

- The *shift* field is used to maintain optimal headroom in an FFT computation, and it enables an optional implicit shift left or right by one bit. The default is not to shift the data.

The `vCTRL` register can be read using VGETC and set using VSETC. The register is set to change the *type* or *shift* values, or to clear the *magnitude*. The register is typically read to inspect the *magnitude* field. Both operations use use `r11` as the implicit source and destination operand:

```
VSETC     vCTRL <- r11[11:0]          Set vector control
VGETC     r11 <- vCTRL                Get vector control
```

In the description of the operations below, we will use `[x]` to denote that we use element `x` of the vector register. Wherever elements are used, they will be `bpe` bits wide (as set per the VSETC instruction), and unless specified otherwise, the operation will iterate over all `0..bpv/bpe-1` elements of the vector register. Empty brackets `[]` are used to indicate that an operation is mapped over the whole vector. In the descriptions we also use a temporary vector `t` that is not holding any state across instructions; the only state held is in the registers specified earlier.

21.2 Vector Memory operations

These instructions load a vector from memory, and store a vector to memory, without performing any mathematical operation. The three vector registers `vC`, `vD`, and `vR` can all be load from memory and stored to memory. Loading and storing from memory has to be word-aligned.:

```
VLDC      vC <- mem[s]                          Load vC from memory
VLDD      vD <- mem[s]                          Load vD from memory
VLDR      vR <- mem[r11]                        Load vR from memory
VSTC      mem[r11..r11+bpv-1] <- vC             Store vC to memory
          vCTRL[5:0] <- max(vCTRL[5:0], csb(vC[]))
VSTD      mem[s..s+bpv-1] <- vD                 Store vD to memory
          vCTRL[5:0] <- max(vCTRL[5:0], csb(vD[]))
VSTR      mem[s..s+bpv-1] <- vR                 Store vR to memory
          vCTRL[5:0] <- max(vCTRL[5:0], csb(vR[]))
VSTRPV    mem[k] <- vR[k], forall k in t        Store part of vR to memory
                                                t is a bytemask
DVST      mem[s] <- register x thread t         Store any vector (debug)
DVLD      register x thread t <- mem[s]         Load any vector (debug)
```

The operations that store a whole vector update the *magnitude* field of the `vCTRL` register. They count the number of significant bits in each element of the vector and store the maximum in the *magnitude* field. In order to calculate the magnitude of a very long vector, one should first clear the headroom register using `VSETC`, then perform all operations, storing the result, and finally reading the headroom register using `VGETC`.

DVST and DVLD are used in DEBUG-mode only. DVST stores a specific vector register of a specific thread in memory. The parameters are three source registers: the first source register carries the address, the second the thread number, and the final register carries the register to store: 0 for `vC`, 1 for `vD`, 2 for `vR`, and 16 for `vCTRL`. DVLD loads a vector register from memory, and uses the same three register parameters. These instructions can only be used in DEBUG mode, an exception is raised otherwise.

21.3 Vector Arithmetic

Arithmetic operations facilitate both complex and ordinary arithmetic. For complex numbers, pairs of real and imaginary values are stored in subsequent elements of a vector, so a vector that contains `epv` elements can hold `epv/2` complex values. For normal arithmetic, the values are stored in `bpe` bits, for complex arithmetic the real and imagi-

nary part have **bpe** bits each. All operations saturate; in the case of complex arithmetic saturation is individual on each component, and saturation will result in a rotation and magnitude reduction.

The detailed semantics of each operation depends on the type. XS3 defines the following types: VSETCTRL_TYPE_INT32, VSETCTRL_TYPE_INT16, VSETCTRL_TYPE_INT8. Multiplications and additions are performed at double precision (`2 x bpe` bits). The bits that are kept depend on the operation.

- For additive operations (VLADD, VLSUB, VLADSB) the bottom **bpe** bits are saturated to a number in the range `[-2^(bpe-1)+1..2^(bpe-1)-1]`.
- For multiplicative operations (VLMUL, VCMR, VCMI, VCMCR, VCMCI), bits `[2 x bpe-3..bpe-2]` are saturated.
- For multiply accumulate operations on 32-bit values, the same bits `[2 x bpe-3..bpe-2]` are saturated
- For MACC operations on smaller values, a full precision 32-bit result is kept.

For a multiplication this works on the assumption that bit `bpe-2` has a magnitude of 1.0 on one of the input operands; and the result has the same magnitude as the other input operand. When both operands use bit `bpe-2` as magnitude 1, then all numbers in the unit square can be multiplied with each other without saturation, which enables block-floating-point complex vector arithmetic to be implemented efficiently.

For example, for VSETCTRL_TYPE_INT32, the bit pattern 0x4000 0000 represents 1.0, and the largest value that can be represented is very close to 2.0. All operations perform saturation to deal with the special cases `(1+j)^2` which will have an answer saturated to `1.9999999991j` In the text below we use the functions msat"()" and ssat"()" to to define the precise operation, and rnd"()" to denote rounding:

```
ssat(x)    =    -2^(bpe-1)+1,            if  x <= -2^(bpe-1)+1
                 2^(bpe-1)-1,            if  x >= 2^(bpe-1)-1
                 x,                      otherwise

 msat(x)   =    ssat( (x + 2^bpe-3) >> (bpe-2) )
 rnd(x)    =    floor( x + 0.5 )
```

The VLADD, VLSUB, and VLMUL operations perform element-wise addition, subtraction, and multiplication on the data type as set with VSETC. The operations are explained graphically in Fig. 1 and Fig. 2:

```
VLADD       vR[i] <- ssat(t[i] + vR[i])      Vector load and add
            where t = mem[s]
            for all i in 0..epv-1
VLSUB       vR[i] <- ssat(t[i] - vR[i])      Vector load and add
            where t = mem[s]
            for all i in 0..epv-1
VLMUL       vR[i] <- msat(t[i] * vR[i])      Vector load and multiply
            where t = mem[s]
            for all i in 0..epv-1
```

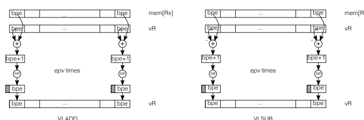


Fig. 1: Vector load and add/subtract operations

For complex arithmetic, the ISA supports a family of operations that implements complex multiplications (VCMR and VCMI) and multiplications with conjugate values (VCMCR and VCMCI). For adding and subtracting numbers, the VLADD and VLSUB oper-

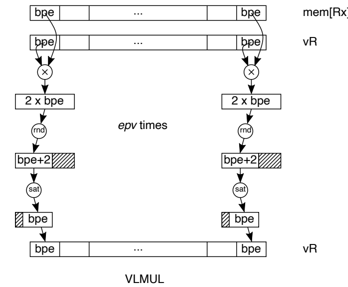


Fig. 2: Vector load and multiply operation

ations can be used. The complex multiplies are useful for Fourier transforms, complex FIRs and for performing a element wise complex vector multiplication. The operations are explained graphically in Fig. 3 and Fig. 4:

```
VCMR  vR[2 i] <- msat(t)                          Complex Multiply Real
      where t = rnd(vC[2 i] * vD[2 i]) - rnd(vC[2 i + 1] * vD[2 i + 1])
      for all i in 0..epv/2-1
VCMI  vR[2 i + 1] <- msat(t)                      Complex Multiply Imaginary
      where t = rnd(vC[2 i + 1] * vD[2 i])+ rnd(vC[2 i] * vD[2 i + 1])
      for all i in 0..epv/2-1
VCMCR vR[2 i] <- msat(t)                          Complex Multiply Conjugate Real
      where rnd(t = vC[2 i] * vD[2 i]) + rnd(vC[2 i + 1] * vD[2 i + 1])
      for all i in 0..epv/2-1
VCMCI vR[2 i + 1] <- msat(t)                      Complex Multiply Conjugate Imaginary
      where t=rnd(vC[2 i + 1] * vD[2 i]) - rnd(vC[2 i] * vD[2 i + 1])
      for all i in 0..epv/2-1
```

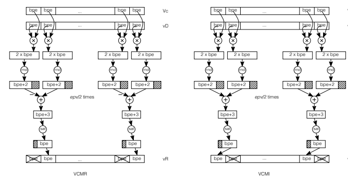


Fig. 3: Vector complex multiply operations

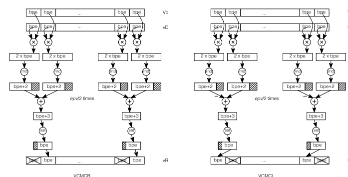


Fig. 4: Vector complex multiply conjugate operations

The pair of VCMR/VCMI instructions executed one after the other will calculate `epv/2` complex products of the form `vD x vC`. A pair of VCMCR/VCMCI instructions will calculate `epv/2` complex conjugate products of the form `vD x conjugate(vC)`.

For implementing FFTs efficiently, an operation is provided that can simultaneously add and subtract values (VLADSB), and a set of instructions that compute Fourier transforms on a small vector. Note that all arithmetic operators below denote complex arithmetic:

```
VLADSB   t = mem[s]                             Load, add, and subtract
         vD[i] <- ssat((t[i] - vR[i]) * x)
         vR[i] <- ssat((t[i] + vR[i]) * x)
         for all i in 0..epv/2-1
VFTFF    vD <- ssat(fft_dif(vD[i]) * x)         DIF FFT Forwards
VFTFB    vD <- ssat(fft_dif^-1(vD[i]) * x)      DIF FFT Backwards
VFTTF    vR <- ssat(fft_dit(vR[i]) * x)         DIT FFT Forwards
VFTTB    vR <- ssat(fft_dit^-1(vR[i]) * x)      DIT FFT Backwards
         where
           x = 2,                 if VEC_SHL
               1,                 if VEC_SH0
               1/2,               if VEC_SHR
```

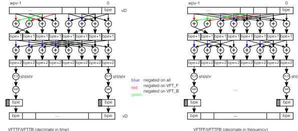


Fig. 5: Vector Fourier transforms

The VLADSB operation performs the butterfly part of the FFT computation. The VFT instructions calculate a Fourier transforms on the vector without bit-reversing. The four versions of this instruction calculate forwards and backwards (inverse) Fourier transforms that are decimate in time or in frequency. All have an optional shift component that can add or remove one bit of headroom based on the amount of headroom that was available after the previous round. This allows FFT implementations to dynamically maintain optimal headroom.

21.4 Multiply accumulate

Two convolution operations enable element-wise convolutions of vectors (VLMACC), or complete inner products (VLMACCR, for MACC and Reduce). Both operate on a vector of **bpv** data elements, each of which **bpe** bits wide. All convolutions maintain the accumulated result in a pair of registers with extended precision. **vR** store the least significant bits as normal and **vD** stores headroom. Depending on the precision data storage is as follows:

- VEC_INT_8 and VEC_INT_16: Both **vD** and **vR** hold a 16-bit part of the accumulator, for a total of 32 bits. All multiply accumulate operations are full precision integer arithmetic with saturation.
- VEC_INT_32: **vR** holds a 32-bit part of the accumulator, **vD** holds 8 bits of headroom. Arithmetic is performed by shifting the multiplied number down by 30 (`bpe-2`) bits as is the case with VLMUL. The unused 24 bits of **vD** replicate the sign bit.

The double result in **vD** and **vR** can be reduced to a normal vector using VLSAT, or they can be accumulated to a single number using VADDDR. The operations are explained graphically in Fig. 6 and Fig. 7:

```
VCLRDR    vD <- 0                               Clear D and R
          vR <- 0

VLMACC    t = mem[s]                            outer product of vC and mem[s]
          for all  i in 0..epv-1
          x = (t[i] *  vC[i])[2 bpe-1 : bpe - 2]
          s = vD[i]:vR[i] + x,                  vD stores 8 bits only
          p = min(max(s, -2^bpe+vac-1+1), 2^bpe+vac-1-1)
          vR[i] <- p[bpe-1..0]
          vD[i] <- p[2 bpe-1..bpe]

VLMACCR   t = mem[s]                            inner product of vC and mem[s]
          bpe16 = max(bpe,16)
          sR = vR[bpv - bpe16 - 1 .. 0]
          sD = vD[bpv - bpe16 - 1 .. 0]
          o = vD[bpv-1..bpv - bpe16]:vR[bpv-1..bpv - bpe16]
          s = o + sum(t[i]  *  vC[i], for i in range(bpv/bpe))
          p = min(max(s, -2^bpe16+vac-1+1), 2^bpe16+vac-1-1)
```

(continues on next page)

(continued from previous page)

```
          vR <- sR : p[bpe16-1..0]
          vD <- sD : p[2 bpe16-1..bpe16]

VLMACCR1  t = mem[s]                            inner product of vC and mem[s]
          bpe16 = max(bpe,16)
          sR = vR[bpv - bpe16 - 1 .. 0]
          sD = vD[bpv - bpe16 - 1 .. 0]
          o = vD[bpv-1..bpv - bpe16]:vR[bpv-1..bpv - bpe16]
          s = o + sum(i=0..bpv-1: (1-2 * t[i]) *  (1-2 * vC[i]))/2
          p = min(max(s, -2^bpe16+vac-1+1), 2^bpe16+vac-1-1)
          vR <- sR : p[bpe16-1..0]
          vD <- sD : p[2 bpe16-1..bpe16]

VLSAT     t = mem[s]                            Saturate double answer into 16/32 bits
          bpe16 = max(bpe,16)
          forall i in 0..bpv/bpe16-1
          vR[i] <- sat(vD[i]:vR[i] >> t[i])
          vD    <- 0

VADDDR    sD = vD[bpv - bpe - 1 .. 0]           Add double and reduce
          s = sum(vD[16*i+15..16*i]:vR[16*i+15..16*i], for i in range(bpv/16))
          p = min(max(s, -2^31+1), 2^31-1)
          vR[bpv - 1 .. 16] <- 0
          vD[bpv - 1 .. 16] <- 0
          vR[15 .. 0]       <- p[bpe-1..0]
          vD[15 .. 0]       <- p[2 bpe-1..bpe]
```

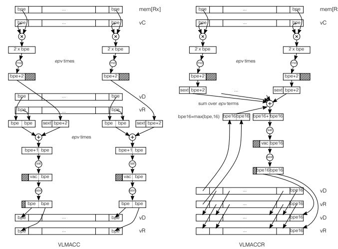


Fig. 6: Vector load and multiply accumulate operations

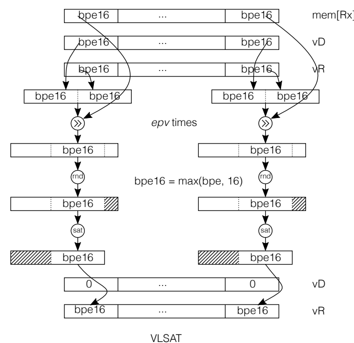


Fig. 7: Vector load and saturate operation

Both VLMACC and VLMACCR instructions are designed to be chained. When a FIR has to be computed that is longer than the vector length, a sequence of VLMACC instructions can be issued on the data, whilst loading the coefficients in between; at the end, a VADDDR is needed to sum all the results together (for 8- and 16-bits), or a VLSAT followed by a VLMACC. When many FIRs need to be computed on identical data (say, whilst multiplying two matrices), the data can be loaded once, and a sequence of up to **epv** VLMACCR operations can be issued to compute up to **epv** intermediate operations, this can be repeated on subsequent rows of the matrix. The VLMACCR instruction can be

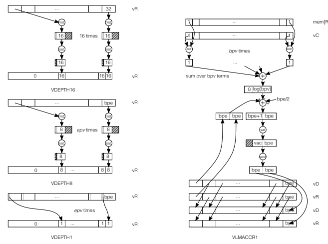


Fig. 8: Bitwise Vector load, multiply, and accumulate

issued for computing a series of up to **epv** inner products in parallel, requiring a load of the coefficients in between.

The VLMACCR1 instruction is very similar to the VLMACCR instruction, except it assumes that each vector contains **bpv** elements each with a value of +1 (encoded as 0) or -1 (encoded as 1). Otherwise, it is the normal multiply accumulate instruction. This will yield a value formatted in the currently selected data format. For example, if the control register is set to VSETCTRL_TYPE_INT8, then up to 32768 values can be accumulated before an overflow can occur; at which time the value will be saturated. The VDEPTH* instructions (see below) can be used to convert the value of **bpe** bits to any bit-depth, for example 1-bit.

The VSIGN instruction extracts the sign of all the vector elements. The VPOS instruction replaces all negative values with 0. The VLASHR operation performs an arithmetic shift right on each element of the vector. It can be used to normalise a vector:

```
VSIGN       vR[] <- vR[] < 0 ? -2^bpe-2 : 2^bpe-2       Compute signs of vector
VPOS        vR[] <- max(0, vR[])                        Vector positive
VDEPTH1     t[bpv-1..epv] = 0                           Binarises a vector
            for i in 0..epv-1
               if vR[i] < 0: t[i] = 1
               else: t[i] = 0
            vR <- t
VDEPTH8     t[bpv-1..epv * 8] = 0                       Converts a vector depth to 8
            for i in 0..epv-1
               t[i  x  8 + 8-1..i x  8] = rnd(vR[i] >> (bpe - 8))
            vR <- t
VDEPTH16    t[bpv-1..epv * 16] = 0                      Reduces a vector depth to 16
            for i in 0..epv-1
               t[i  x  16 + 16-1..i * 16] = rnd(vR[i] >> (bpe - 16))
            vR <- t
VLASHR      p = mem[s]                                  Vector arithmetic shift right mem[s] by t
            vR[] <- p >> t,             if  t in 0..bpe-1
            vR[] <- p[bpe-1]..p[bpe-1], if  t in bpe..
            vR[] <- ssat(p << t),       if  t in -bpe+1..-1
            vR[] <- ssat(p << bpe),     if  t in ..-bpe
VEQDR       r10 <- vD = vR                              Equality test on vD and vR
VEQCR       r10 <- vC = vR                              Equality test on vC and vR
```

22 XCore XS3 Instructions

This section presents the instructions in alphabetical order. For each instruction we present a short textual description, followed by the assembly syntax, its meaning in a more formal notation, its encoding(s) and potential exceptions that can be raised by this exception.

The processor operates on words - registers are one-word wide, data can be transferred to ports and channels in words, and most memory operations operate on words. A word is `bpw` bits long, or `Bpw` bytes long.

In the description we use the following notation to describe operands and constants:

b
: denotes a bit-pattern - one of `bpw`, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, and 32; these are encoded using numbers 0…11.

c
: register used as a conditional.

d,e
: register used as a destination.

r
: register used as a resource identifier.

s
: register used as a source.

t
: register used as a thread identifier.

us
: a small unsigned constant in the range `0...11`

ux
: an unsigned constant in the range `0...(2^x-1)`

v,w,x,y
: registers used for two or more sources.

All mathematical operators are assumed to work on Integers (**Z**) and, unless otherwise stated, bit patterns found in registers are interpreted *unsigned*. Signed numbers are represented using two's complement, and if an operand is interpreted as a signed number, this is denoted by a keyword `signed`. In addition to the standard numerical operators we assume the following bitwise operators:

|
: Bitwise or.

&
: Bitwise and.

@
: Bitwise xor.

~
: Bitwise complement.

Square brackets are used for two purposes. When preceded with the word `mem` square brackets address a memory location. Otherwise, they indicate that one or more bits are sliced out of a bit pattern. Bits can be spliced together using a `:` operator. The bit pattern `x:y` is a pattern where `x` are the higher order bits and `y` are the lower order bits.

The notation `mem[x]` represents word-based access to memory, and the address `x` must be word-aligned (that is, the address must be a multiple of `Bpw`). Instructions that read or write data to memory that is not a word in size (such as a byte or a 16-bit value) explicitly specify which bits in memory are accessed.

The instruction encoding specifies the *opcode* bits of the encoding - the way that the operands are encoded is specified by the corresponding page in the chapter on instruction formats (if you access this document electronically there should be a hyperlink). Each operand in the instruction chapter maps positionally on an operand in the format chapter.

22.1 ADD: Integer unsigned add

Adds two unsigned integers together. There is no check for overflow. Where it occurs, overflow is ignored.

To add with carry the *LADD* instruction should be used instead.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
ADD d,x,y
```

Operation:

```
d <- (x + y) % 2^bpw
```

Encoding:

3r: Three register

0 0 0 1 0	. . . . .	. .	. .	. .	(*M or R*)

22.2 ADDI: Integer unsigned add immediate

Adds two unsigned integers together. There is no check for overflow. Where it occurs, overflow is ignored.

To add with carry the *LADD* instruction should be used instead.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: us, An integer in the range 0…11

Mnemonic and operands:

```
ADDI d,x,u_s
```

Operation:

```
d <- (x + us) % 2^bpw
```

Encoding:

2rus: Two register with immediate

1 0 0 1 0	. . . . .	. .	. .	. .

(*M or R*)

22.3 AND: Bitwise and

Produces the bitwise AND of two words.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
AND d,x,y
```

Operation:

```
d <- x & y
```

Encoding:

3r: Three register

0 0 1 1 1	. . . . .	. .	. .	. .	(*M or R*)

22.4 ANDNOT: And not

ANDNOT clears bits in a word. Given the bits set a bit pattern (s), ANDNOT clears the equivalent bits in the destination operand (d). ANDNOT is a two operand instruction where the first operand acts as both source and destination.

ANDNOT can be used to efficiently operate on bit patterns that span a non-integral number of bytes.

See *MKMSK* for how to build masks efficiently.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
ANDNOT d,s
```

Operation:

```
d <-s & ~ s
```

Encoding:

2r: Two register

0 0 1 0 1	. . . . .	.	0	. .	. .

(*M or R*)

22.5 ASHR: Arithmetic shift right

Right shifts a signed integer and performs sign extension. The shift distance (y) is an unsigned integer. If the shift distance is larger than the size of a word, the result will only be the sign extension.

If sign extension is not required, the *SHR* instruction should be used instead. Note that ASHR is not the same as a *DIVS* by `2^{y}` because ASHR rounds towards minus infinity, whereas DIVS rounds towards zero.

ASHR will perform an shift left with a negative value.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
ASHR d,x,y
```

Operation:

```
d <- x[bpw-1]: ... :x[bpw-1]:x[bpw-1...y], if 0 < y < bpw
     x,                                   if y = 0
     x[bpw-1]: ... :x[bpw-1],             if y >= bpw
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .
0 0 0 1 0	1 1 1 1 1	1 0	1 1	0 0

(*M and R*)

22.6 ASHRI: Arithmetic shift right immediate

Right shifts a signed integer and performs sign extension. The shift distance (`bitp`) is an unsigned integer. If the shift distance is larger than the size of a word, the result will only be the sign extension.

If sign extension is not required, the *SHR* instruction should be used instead. Note that ASHR is not the same as a *DIVS* by `2^{bitp}` because ASHR rounds towards minus infinity, whereas DIVS rounds towards zero.

ASHR will perform an shift left with a negative value.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: bitp, A bit position; one of `bpw`, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32

Mnemonic and operands:

```
ASHRI d,x,bitp
```

Operation:

```
d <- x[bpw-1]: ... :x[bpw-1]:x[bpw-1...bitp], if 0 < bitp < bpw
     x,                                       if bitp = 0
     x[bpw-1]: ... :x[bpw-1],                 if bitp >= bpw
```

Encoding:

l2rus: Two register with immediate long

1 1 1 1 1	. . . . . .	. .	. .	. .
1 0 0 1 0	1 1 1 1 1 1	1 0	1 1	0 0

(*M and R*)

22.7 BAU: Branch absolute unconditional register

Branches to the address given in a general purpose register. The register value must be even, and should point to a valid memory location.

The instruction has one operand:

op1
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
BAU s
```

Operation:

```
pc <- s
```

Encoding:

1r: Register

0 0 1 0 0	1 1 1 1 1	1	1	. . . .

(*M*)

Conditions that raise an exception:

ET_ILLEGAL_PC
: The address specified was not 16-bit aligned or did not point to a memory location.

22.8 BITREV: Bit reverse

Reverses the bits in a word; the most significant bit of the source operand will be produced in the least significant bit of the destination operand, the value of the least significant bit of the source operand will be produced in the most significant bit of the destination operand.

This instruction can be used in conjunction with *BYTEREV* in order to translate between different ordering conventions such as big-endian and little-endian.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
BITREV d,s
```

Operation:

```
d[bpw-1...0] <- s[0] : s[1] : s[2] : ... : s[bpw-1]
```

Encoding:

2r: Two register

0 0 0 1 0	. . . . .	.	0	. .	. .

(*M or R*)

22.9 BLA: Branch and link absolute via register

This instruction implements a procedure call to an absolute address. The program counter is saved in the link-register `lr` and the program counter is set to the given address. This address must be even and point to a valid memory address, otherwise an exception is raised. On execution of BLA, the processor will read the target instruction so that the invoked procedure will start without delay.

On entry to the procedure, the Link Register can be saved on the stack using the *ENTSP* instruction. *RETSP* performs the opposite of this instruction, returning from a procedure call.

The instruction has one operand:

op1
s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
BLA s
```

Operation:

```
lr <- pc
pc <- s
```

Encoding:

1r: Register

0 0 1 0 0	1 1 1 1 1	1	0	. . . . .

(*M*)

Conditions that raise an exception:

ET_ILLEGAL_PC
The address specified was not 16-bit aligned or did not point to a memory location.

22.10 BLACP: Branch and link absolute via constant pool

This instruction implements a call to a procedure via the constant pool lookup table. The program counter is saved in the link-register (`lr`). The program counter is loaded from the constant pool table. The constant pool register (`cp`) is used as the base address for the table. An offset (`u10`) specifies which word in the table to use. Because the instruction requires access to memory, the execution of the target instruction may be delayed by one instruction in order to fetch the target instruction.

On entry to the procedure, the Link Register can be saved on the stack using the *ENTSP* instruction. *RETSP* performs the opposite of this instruction, returning from a procedure call.

The instruction has one operand:

op1
u10, A 20-bit immediate in the range 0…1048575. If `u20 < 1024`, the instruction requires no prefix

Mnemonic and operands:

```
BLACP u10
```

Operation:

```
lr <- pc
pc <- mem[cp + u10 * Bpw]
```

Encoding:

u10: 10-bit immediate

1 1 1 0 0	0	. . . .

(*M*)

lu10: 20-bit immediate

1 1 1 1 0	0	. . . .
1 1 1 0 0	0	. . . .

(*M and R*)

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_ILLEGAL_PC
Loaded value was not 16-bit aligned or did not point to a memory location (trapped during next cycle).

ET_LOAD_STORE
Register `cp` points to an unaligned address, or the indexed address does not point to a valid memory address.

22.11 BLAT: Branch and link absolute via table

This instruction implements a call to a procedure via a lookup table. The program counter is saved in the link-register (`lr`). The program counter is loaded from the lookup table. The lookup table base address is taken from `r11`. An offset (`u_{16}`) specifies which word in the table to use. Because the instruction requires access to memory, the execution of the target instruction may be delayed by one instruction in order to fetch the target instruction.

On entry to the procedure, the Link Register can be saved on the stack using the *ENTSP* instruction. *RETSP* performs the opposite of this instruction, returning from a procedure call.

The instruction has one operand:

op1
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
BLAT u_{16}
```

Operation:

```
lr <- pc
pc <- mem[r11 + u16 * Bpw]
```

Encoding:

u6: 6-bit immediate

0 1 1 1 0	0	1 1 0 1	. . . . . .

(*M*)

lu6: 16-bit immediate

1 1 1 1 0	0	. . . .

0 1 1 1 0	0	1 1 0 1	. . . . . .

(*M and R*)

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_ILLEGAL_PC
: Loaded value was not 16-bit aligned or did not point to a memory location (trapped during the next cycle).

ET_LOAD_STORE
: Register `r11` points to an unaligned address, or the indexed address does not point to a valid memory address.

22.12 BLRB: Branch and link relative backwards

This instruction performs a call to a procedure: the address of the next instruction is saved in the link-register (`lr`) An unsigned offset is subtracted from the program counter. This implements a relative jump.

On entry to the procedure, the Link Register can be saved on the stack using the *ENTSP* instruction. *RETSP* performs the opposite of this instruction, returning from a procedure call. The counterpart forward call is called *BLRF*.

The instruction has one operand:

op1
: u10, A 20-bit immediate in the range 0…1048575. If `u20 < 1024`, the instruction requires no prefix

Mnemonic and operands:

```
BLRB u10
```

Operation:

```
lr <- pc
pc <- pc - u10 * iw
```

Encoding:

u10: 10-bit immediate

1 1 0 1 0	1	. . . .

(*M*)

lu10: 20-bit immediate

1 1 1 1 0	0	. . . .
1 1 0 1 0	1	. . . .

(*M and R*)

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_ILLEGAL_PC
: The new PC is not pointing to a valid memory location.

22.13 BLRF: Branch and link relative forwards

This instruction performs a call to a procedure: the address of the next instruction is saved in the link-register (`lr`) An unsigned offset is added to the program counter. This implements a relative jump.

On entry to the procedure, the Link Register can be saved on the stack using the *ENTSP* instruction. *RETSP* performs the opposite of this instruction, returning from a procedure call. The counterpart backward call is called *BLRB*.

The instruction has one operand:

op1
: u10, A 20-bit immediate in the range 0…1048575. If `u20 < 1024`, the instruction requires no prefix

Mnemonic and operands:

```
BLRF u10
```

Operation:

```
lr <- pc
pc <- pc + u10 * iw
```

Encoding:

u10: 10-bit immediate

1 1 0 1 0	0	. . . .

(*M*)

lu10: 20-bit immediate

1 1 1 1 0	0	. . . .
1 1 0 1 0	0	. . . .

(*M and R*)

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_ILLEGAL_PC
: The new PC is not pointing to a valid memory location.

22.14 BRBF: Branch relative backwards false

This instruction implements a conditional relative jump backwards. A condition (`c`) is tested whether it represents 0 (false) and if this is the case an offset (`u_{16}`) is subtracted from the program counter.

This instruction is part of a group of four instructions that conditionally jump forwards or backwards on true or false conditions: *BRBF*, *BRBT*, *BRFF*, and *BRFT*.

The instruction has two operands:

op1
: c, Operand register, one of `r0`… `r11`

op2
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
BRBF c,u_{16}
```

Operation:

```
if c = 0:
  pc <- pc - u16 * iw
```

Encoding:

ru6: Register with 6-bit immediate

| 0 1 1 1 1 | 1 | . . . . | . . . . . . | (*M*)

lru6: Register with 16-bit immediate

| 1 1 1 1 0 | 0 | . . . . | (*M and R*)

| 0 1 1 1 1 | 1 | . . . . | . . . . . . |

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_ILLEGAL_PC
: The new PC is not pointing to a valid memory location.

22.15 BRBT: Branch relative backwards true

This instruction implements a conditional relative jump backwards. A condition (`c`) is tested whether it is not 0 (true) and if this is the case an offset (`u_{16}`) is subtracted from the program counter.

This instruction is part of a group of four instructions that conditionally jump forwards or backwards on true or false conditions: *BRBF*, *BRBT*, *BRFF*, and *BRFT*.

The instruction has two operands:

op1
: c, Operand register, one of `r0`… `r11`

op2
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
BRBT c,u_{16}
```

Operation:

```
if c != 0:
  pc < pc - u16 * iw
```

Encoding:

ru6: Register with 6-bit immediate

0 1 1 1 0	1	. . . .	. . . . . .

(*M*)

lru6: Register with 16-bit immediate

1 1 1 1 0	0	. . . .

0 1 1 1 0	1	. . . .	. . . . . .

(*M and R*)

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_ILLEGAL_PC
: The new PC is not pointing to a valid memory location.

22.16 BRBU: Branch relative backwards unconditional

This instruction implements a relative jump backwards. The operand specifies the offset that should be subtracted from the program counter.

The counterpart forward relative jump is *BRFU*.

The instruction has one operand:

op1
u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
BRBU u_{16}
```

Operation:

```
pc <- pc - u16 * iw
```

Encoding:

u6: 6-bit immediate

0 1 1 1 0	1	1 1 0 0	. . . . . .

(*M*)

lu6: 16-bit immediate

1 1 1 1 0	0	. . . .

0 1 1 1 0	1	1 1 0 0	. . . . . .

(*M and R*)

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_ILLEGAL_PC
The new PC is not pointing to a valid memory location.

22.17 BRFF: Branch relative forward false

This instruction implements a conditional relative jump forwards. A condition (`c`) is tested whether it represents 0 (false) and if this is the case an offset (`u_{16}`) is added to the program counter.

This instruction is part of a group of four instructions that conditionally jump forwards or backwards on true or false conditions: *BRBF*, *BRBT*, *BRFF*, and *BRFT*.

The instruction has two operands:

op1
: c, Operand register, one of `r0`… `r11`

op2
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
BRFF c,u_{16}
```

Operation:

```
if c == 0:
  pc <- pc + u16 * iw
```

Encoding:

ru6: Register with 6-bit immediate

0 1 1 1 1	0	. . . .	. . . . . .

(*M*)

lru6: Register with 16-bit immediate

1 1 1 1 0	0	. . . .

0 1 1 1 1	0	. . . .	. . . . . .

(*M and R*)

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_ILLEGAL_PC
: The new PC is not pointing to a valid memory location.

22.18 BRFT: Branch relative forward true

This instruction implements a conditional relative jump forwards. A condition (`c`) is tested whether it is not 0 (true) and if this is the case an offset (`u_{16}`) is added to the program counter.

This instruction is part of a group of four instructions that conditionally jump forwards or backwards on true or false conditions: *BRBF*, *BRBT*, *BRFF*, and *BRFT*.

The instruction has two operands:

op1
: c, Operand register, one of `r0`... `r11`

op2
: u16, A 16-bit immediate in the range 0...65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
BRFT c,u_{16}
```

Operation:

```
if c != 0:
  pc <- pc + u16 * iw
```

Encoding:

ru6: Register with 6-bit immediate

0 1 1 1 0	0	. . . .	. . . . . .

(*M*)

lru6: Register with 16-bit immediate

1 1 1 1 0	0	. . . .

0 1 1 1 0	0	. . . .	. . . . . .

(*M and R*)

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_ILLEGAL_PC
: The new PC is not pointing to a valid memory location.

22.19 BRFU: Branch relative forward unconditional

This instruction implements a relative jump forwards. The operand specifies the offset that should be added to the program counter.

The counterpart backward relative jump is *BRBU*.

The instruction has one operand:

op1
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
BRFU u_{16}
```

Operation:

```
pc <- pc + u16 * iw
```

Encoding:

u6: 6-bit immediate

0 1 1 1 0	0	1 1 0 0	. . . . . .

(*M*)

lu6: 16-bit immediate

1 1 1 1 0	0	. . . .

0 1 1 1 0	0	1 1 0 0	. . . . . .

(*M and R*)

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_ILLEGAL_PC
: The new PC is not pointing to a valid memory location.

22.20 BRU: Branch relative unconditional register

This instruction implements a jump using a signed offset stored in a register. Because instructions are aligned on 16-bit boundaries, the offset in the register is multiplied by 2. Negative values cause backwards jumps.

The instruction has one operand:

op1
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
BRU s
```

Operation:

```
pc <- pc + s_{signed} * iw
```

Encoding:

1r: Register

0 0 1 0 1	1 1 1 1 1	1	0	. . . .

(*M*)

Conditions that raise an exception:

ET_ILLEGAL_PC
: The new PC is not pointing to a valid memory location.

22.21 BYTEREV: Byte reverse

This instruction reverses the bytes of a word.

Together with the *BITREV* instruction this can be used to resolve requirements of different ordering conventions such as little-endian and big-endian.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
BYTEREV d,s
```

Operation:

```
d[bpw-1...0] <- s[7...0] : s[15...8] : ... : s[bpw-1:bpw-8]
```

Encoding:

2r: Two register

0 0 0 0 0	. . . . .	.	0	. .	. .

(*M or R*)

22.22 CHKCT: Test for control token

If the next token on a channel is the specified control token, then this token is discarded from the channel. If not, the instruction raises an exception.

This instruction pauses if the channel does not have a token available to be read.

This instruction can be used together with *OUTCT* in order to implement robust protocols on channels; each OUTCT must have a matching CHKCT or *INCT*. *TESTCT* tests for a control token without trapping, and does not discard the control token.

The instruction has two operands:

op1
r, Operand register, one of `r0`… `r11`

op2
s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
CHKCT r,s
```

Operation:

```
if hasctoken(r) && s == token(r):
  skiptoken(r)
else:
  raise exception
```

Encoding:

2r: Two register

1 1 0 0 1	. . . . .	.	0	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
`r` is not pointing to a channel resource, or the resource is not in use.

ET_ILLEGAL_RESOURCE
`r` contains a data token.

ET_ILLEGAL_RESOURCE
`r` contains a control token different to `s`.

22.23 CHKCTI: Test for control token immediate

If the next token on a channel is the specified control token, then this token is discarded from the channel. If not, the instruction raises an exception.

This instruction pauses if the channel does not have a token available to be read.

This instruction can be used together with *OUTCT* in order to implement robust protocols on channels; each OUTCT must have a matching CHKCT or *INCT*. *TESTCT* tests for a control token without trapping, and does not discard the control token.

The instruction has two operands:

op1
r, Operand register, one of `r0`… `r11`

op2
us, An integer in the range 0…11

Mnemonic and operands:

```
CHKCTI r,u_s
```

Operation:

```
if hasctoken(r) && us == token(r):
  skiptoken(r)
else:
  raise exception
```

Encoding:

rus: Register with immediate

1 1 0 0 1	. . . . .	.	1	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
`r` is not pointing to a channel resource, or the resource is not in use.

ET_ILLEGAL_RESOURCE
`r` contains a data token.

ET_ILLEGAL_RESOURCE
`r` contains a control token different to `u_s`.

22.24 CLRE: Clear all events

Clears the thread's Event-Enable and In-Enabling flags, and disables all individual events for the thread. Any resource (port, channel, timer) that was enabled for this thread will be disabled.

The instruction has no operands.

Mnemonic and operands:

```
CLRE
```

Operation:

```
sr[eeble] <- 0
sr[inenb] <- 0
forall res:
  if thread(res) == tid && event(res):
    enb(res) <- 0
```

Encoding:

0r: No operands

0 0 0 0 0	1 1 1 1 1	1	0	1 1 0 1	(*R*)

22.25 CLRPT: Clear the port time

Clears the timer that is used to determine when the next output on a port will happen.

The instruction has one operand:

op1
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
CLRPT r
```

Operation:

```
clearporttime(r)
```

Encoding:

1r: Register

1 0 0 0 0	1 1 1 1 1	1	0	. . . .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not pointing to a port resource, or the resource is not in use.

22.26 CLS: Count leading sign bits

Counts the number of leading sign bits in `s` If the `s` is zero, then `bpw` is produced. The instruction can never produce 0 as there is always at least one sign-bit. This instruction can be used to efficiently compute the headroom of signed integers.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
CLS d,s
```

Operation:

```
d <- bpw,                              if s == 0
     bpw - 1 - floor(log2 s),        if s[bpw-1] == 0
     bpw - 1 - floor(log2 ~s),       if s[bpw-1] == 1
```

Encoding:

2r: Two register

0 0 1 0 0	. . . . .	.	0	. .	. .

(*M or R*)

22.27 CLRSR: Clear bits SR

Clear bits in the thread's status register (`sr`). The mask supplied specifies which bits should be cleared. CLRSR can only be used to clear the EEBLE, IEBLE, INENB, ININT and INK bits.

SETSR is used to set bits in the status register. The value of these bits are documented on the SETSR page

The instruction has one operand:

op1
u16, A 16-bit immediate in the range 0...65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
CLRSR u_{16}
```

Operation:

```
sr <-sr & ~ u16
```

Encoding:

u6: 6-bit immediate

0 1 1 1 1	0	1 1 0 0	. . . . . .

(*R*)

lu6: 16-bit immediate

1 1 1 1 0	0	. . . .

0 1 1 1 1	0	1 1 0 0	. . . . . .

(*M and R*)

The latter is prefixed for long immediates.

22.28 CLZ: Count leading zeros

Counts the number of leading zero bits in its operand. If the operand is zero, then bpw is produced. If the operand starts with a '1' bit (ie, a negative signed integer, or a large unsigned integer), then 0 is produced. This instruction can be used to efficiently normalise integers.

The instruction has two operands:

op1
: d, Operand register, one of r0... r11

op2
: s, Operand register, one of r0... r11

Mnemonic and operands:

```
CLZ d,s
```

Operation:

```
d <- bpw,                        if s == 0
     bpw - 1 - floor(log2 s),    if s[bpw-1] == 0
     0,                          if s[bpw-1] = 1
```

Encoding:

2r: Two register

0 0 0 0 1	. . . . .	.	0	. .	. .	(*M or R*)

22.29 CRC8: 8-step CRC

Incorporates the CRC over 8-bits of a 32-bit word into a Cyclic Redundancy Checksum. The instruction has four operands. Similar to *CRC* the first operand is used both as a source to read the initial value of the checksum and a destination to leave the updated checksum, and there are operands to specify the the polynomial (p) to use when computing the CRC, and the data (e) to compute the CRC over. Since on completion of the instruction the part of the data that has not yet been incorporated into the CRC, the most significant 24-bits of the data are stored in a second destination register (x). This enables repeated execution of CRC8 over a part-word. Executing Bpw CRC8 instructions in a row is identical to executing a single *CRC* instruction.

The instruction has four operands:

op1
: d, Operand register, one of `r0`… `r11`

op4
: x, Operand register, one of `r0`… `r11`

op2
: e, Operand register, one of `r0`… `r11`

op3
: p, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
CRC8 d,x,e,p
```

Operation:

```
for step in range(8):
   if r[0] == 1:
      r <- (d[step] : r[bpw-1 ... 1]) @ p
   else
      r <- (d[step] : r[bpw-1 ... 1])
d[bpw-1...0] <- 0:0:0:0:0:0:0:0:e[bpw-1:8]
```

Encoding:

l4r: Four register long

1 1 1 1 1	. . . . .	. .	. .	. .
0 0 0 0 0	1 1 1 1 1	1 0	. .	. .

(*M and R*)

22.30 CRC: Word CRC

Incorporates a word into a Cyclic Redundancy Checksum. The instruction has three operands. The first operand (**d**) is used both as a source to read the initial value of the checksum and a destination to leave the updated checksum. The other operands are the data to compute the CRC over (**x**) and the polynomial to use when computing the CRC (**p**).

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: p, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
CRC d,x,p
```

Operation:

```
for step in range(bpw):
   if r[0] == 1:
     r <- (d[step] : r[bpw-1 ... 1]) @ p
   else
     r <- (d[step] : r[bpw-1 ... 1])
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .	(*M and R*)
1 0 1 0 1	1 1 1 1 1	1 0	1 1	0 0	

22.31 CRC32_INC: Word CRC with address increment

Incorporates a word into a Cyclic Redundancy Checksum. The instruction has three operands. The first operand (**d**) is used both as a source to read the initial value of the checksum and a destination to leave the updated checksum. The other operands are the data to compute the CRC over (**x**) and the polynomial to use when computing the CRC (**p**).

Simultaneously, the instruction increments a register with the specified value.

The instruction has five operands:

op1
: d, Operand register, one of `r0`… `r11`

op4
: a, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: p, Operand register, one of `r0`… `r11`

op5
: bitp, A bit position; one of **bpw**, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32

Mnemonic and operands:

```
CRC32_INC d,a,x,p,bitp
```

Operation:

```
for step in range(bpw):
   if r[0] == 1:
      r <- (d[step] : r[bpw-1 ... 1]) @ p
   else
      r <- (d[step] : r[bpw-1 ... 1])
a <- a + bitp
```

Encoding:

l4rus: Four registers with immediate long

1 1 1 1 1	. . . . .	. .	. .	. .	(*M and R*)
0 0 1 0 1	x x x x x	. 1	. .	. .	

22.32 CRCN: Variable step CRC

Incorporates the CRC over N-bits of a 32-bit word into a Cyclic Redundancy Checksum. The instruction has four operands. Similar to *CRC* the first operand is used both as a source to read the initial value of the checksum and a destination to leave the updated checksum, and there are operands to specify the the polynomial (p) to use when computing the CRC, the data (d) to compute the CRC over, and the number of bits (n).

The CRCN instruction is provided to complete the checksum over messages that have a number of bytes that is not a multiple of Bpw, or for messages where the start is not aligned.

The instruction has four operands:

op1
: x, Operand register, one of r0… r11

op4
: d, Operand register, one of r0… r11

op2
: p, Operand register, one of r0… r11

op3
: n, Operand register, one of r0… r11

Mnemonic and operands:

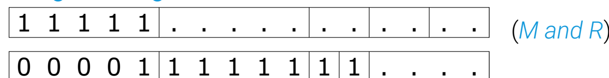
```
CRCN x,d,p,n
```

Operation:

```
for step in range(if n < bpw then n else bpw):
   if r[0] == 1:
     r <- (d[step] : r[bpw-1 ... 1]) @ p
   else
     r <- (d[step] : r[bpw-1 ... 1])
```

Encoding:

l4r: Four register long



(*M and R*)

22.33 DCALL: Call a debug interrupt

Switches to debug mode, saving the current program counter and stack pointer of thread 0 in debug registers. Thread 0 is deemed to have taken an interrupt and is therefore removed from the multicycle unit and lock resources, and all of its resources are informed such that it is removed from any resources it was inputting/outputting/eventing on.

DRET returns from a debug interrupt. *DENTSP* and *DRESTSP* instructions are used to switch to and from the debug SP.

The instruction has no operands.

Mnemonic and operands:

```
DCALL
```

Operation:

```
dspc <- pc(t0)
dssr <- sr(t0)
pc(t0) <- debugentry
dtype <- dcallcause
sr(t0)[inint] <- 1
sr(t0)[ink] <- 1
sr(t0)[eeble] <- 0
sr(t0)[ieble] <- 0
sr(t0)[inenb] <- 0
sr(t0)[waiting] <- 0
in_debug <- 1
```

Encoding:

0r: No operands

0 0 0 0 0	1 1 1 1 1	1	1	1 1 0 0

(*M or R*)

22.34 DENTSP: Save and modify stack pointer for debug

Causes thread 0 to use the Debug SP rather than the SP in debug mode. Saves the SP in debug saved stack pointer (DSSP), and loads the SP with the top word location in RAM.

DRESTSP is used to use the restore the original SP from the DSSP.

The instruction has no operands.

Mnemonic and operands:

```
DENTSP
```

Operation:

```
dssp <- sp
sp <- ramend
```

Encoding:

l0r: No operands

1 1 1 1 1	1 1 1 1 1	1	0	1 1 0 0
0 0 0 1 0	1 1 1 1 1	1	0	1 1 0 0

(*M and R*)

Conditions that raise an exception:

ET_ILLEGAL_INSTRUCTION
not in debug mode.

22.35 DGETREG: Debug read of another thread's register

The contents of any thread's register can then be accessed for debugging purpose. To access the state of a thread, first used *SETPS* to set **dtid** and **dtreg** to the thread identifier and register number within the thread state.

The instruction has one operand:

op1
s, Operand register, one of `r0`... `r11`

Mnemonic and operands:

```
DGETREG s
```

Operation:

```
s <- register(dtid, dtreg)
```

Encoding:

1r: Register

0 0 1 1 1	1 1 1 1 1	1	0	. . . .

(*M*)

Conditions that raise an exception:

ET_ILLEGAL_INSTRUCTION
not in debug mode.

22.36 DIVS: Signed division

Produces the result of dividing two signed words, rounding the result towards zero. For example `5 // 3` is `1`, `-5 // 3` is `-1`, `-5 // -3` is `1`, and `5 // -3` is `-1`.

This instruction does not execute in a single cycle, and multiple threads may share the same division unit. The division may take up to `bpw` thread-cycles.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
DIVS d,x,y
```

Operation:

```
d <- x // y       # signed
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .
0 1 0 0 0	1 1 1 1 1	1 0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_ARITHMETIC
: Division by 0.

ET_ARITHMETIC
: Division of `-2^{bpw-1}` by `-1`

22.37 DIVU: Unsigned divide

Computes an unsigned integer division, rounding the answer down to 0. For example `5// 3` is `1`.

This instruction does not execute in a single cycle, and multiple threads may share the same division unit. The division may take up to `bpw` thread-cycles.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
DIVU d,x,y
```

Operation:

```
d <- x // y
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .
0 1 0 0 1	1 1 1 1 1	1 0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_ARITHMETIC
: Division by 0.

22.38 DRESTSP: Restore non debug stack pointer

Causes thread 0 to use the original SP rather than the debug SP. Restores the SP from the debug saved stack pointer (DSSP)

DENTSP is used to use the save the original SP to the DSSP.

The instruction has no operands.

Mnemonic and operands:

```
DRESTSP
```

Operation:

```
sp <- dssp
```

Encoding:

l0r: No operands

1 1 1 1 1	1 1 1 1 1	1	0	1 1 0 0
0 0 0 1 1	1 1 1 1 1	1	0	1 1 0 0

(*M and R*)

Conditions that raise an exception:

ET_ILLEGAL_INSTRUCTION
not in debug mode.

22.39 DRET: Return from debug interrupt

Exits debug mode, restoring thread 0's program counter and stack pointer from the start of the debug interrupt.

DCALL calls a debug interrupt. *DENTSP* and *DRESTSP* instructions are used to switch to and from the debug SP.

The instruction has no operands.

Mnemonic and operands:

```
DRET
```

Operation:

```
pc(t0) <- dspc
sr(t0) <- dssr
```

Encoding:

l0r: No operands

1 1 1 1 1	1 1 1 1 1	1	0	1 1 0 0
0 0 0 0 1	1 1 1 1 1	1	0	1 1 0 0

(*M and R*)

Conditions that raise an exception:

ET_ILLEGAL_INSTRUCTION
not in debug mode.

ET_ILLEGAL_PC
The return address is invalid.

22.40 DUALENTSP: Adjust stack and save link register

Stores the link register on the stack then adjusts the stack pointer creating enough space for the procedure call that has just been entered.

See *RETSP* for the operation that restores the link-register.

The instruction has one operand:

op1
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
DUALENTSP u_{16}
```

Operation:

```
if u16 > 0:
  mem[sp] <- lr
  sp <- sp - u16 * Bpw
sr[di]<- 1
```

Encoding:

u6: 6-bit immediate

0 1 1 1 1	1	1 1 1 0	. . . . . .

(*M*)

lu6: 16-bit immediate

1 1 1 1 0	0	. . . .

0 1 1 1 1	1	1 1 1 0	. . . . . .

(*M and R*)

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_LOAD_STORE
: The indexed address is unaligned, or does not point to a valid memory address.

22.41 DVLD: Debug Loads vector

Loads a vector of any thread from memory: `r` must be 0, 1, or 2 (for C, D, and O), and `t` must be a thread number.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: t, Operand register, one of `r0`… `r11`

op3
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
DVLD d,t,r
```

Operation:

```
for k in range(bpv // 8):
  if r == 0:
    vC(t)[k* 8+7..k* 8] <- mem[d + k]
  if r == 1:
    vD(t)[k* 8+7..k* 8] <- mem[d + k]
  if r == 2:
    vR(t)[k* 8+7..k* 8] <- mem[d + k]
  if r == 16
    vSR(t) <- mem[d]
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .
1 0 1 0 1	1 1 1 1 1	1 0	1 1	1 0

(*M and R*)

Conditions that raise an exception:

ET_ILLEGAL_INSTRUCTION
: not in debug mode.

ET_LOAD_STORE
: `d` is not word aligned, or a complete vector cannot be loaded from this address.

22.42 DVST: Debug Store vector

Stores a vector from any thread in memory: `r` must be 0, 1, or 2 (for C, D, and O), and `t` must be a thread number.

The instruction has three operands:

op1
: d, Operand register, one of `r0`... `r11`

op2
: t, Operand register, one of `r0`... `r11`

op3
: r, Operand register, one of `r0`... `r11`

Mnemonic and operands:

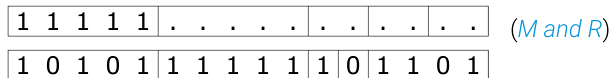
```
DVST d,t,r
```

Operation:

```
for k in range(bpv // 8):
  if r = 0:
    mem[d + k] <- vC(t)[k* 8+7..k* 8]
  if r = 1:
    mem[d + k] <- vD(t)[k* 8+7..k* 8]
  if r = 2:
    mem[d + k] <- vR(t)[k* 8+7..k* 8]
  if r = 16:
    mem[d] <- vSR(t)
```

Encoding:

l3r: Three register long



1 1 1 1 1	. . . . .	. .	. .	. .
1 0 1 0 1	1 1 1 1 1	1 0	1 1	0 1

(*M and R*)

Conditions that raise an exception:

ET_ILLEGAL_INSTRUCTION
: not in debug mode.

ET_LOAD_STORE
: `d` is not word aligned, or a complete vector cannot be stored at this address.

22.43 ECALLF: Throw exception if zero

This instruction checks whether the operand is 0 (false) and raises an exception if it is the case. It can be used to implement assertions, and to implement array bound checks together with the LSU instruction.

The instruction has one operand:

op1
: c, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
ECALLF c
```

Operation:

```
if c == 0:
  raise exception
```

Encoding:

1r: Register

0 1 0 0 1	1 1 1 1 1	1	0	. . . .

(*M or R*)

Conditions that raise an exception:

ET_ECALL
: `c = 0`.

22.44 ECALLT: Throw exception if non-zero

This instruction checks whether a condition is not 0, and raises an exception if it is the case. It can be used to implement assertions.

The instruction has one operand:

op1
: c, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
ECALLT c
```

Operation:

```
if c != 0:
  raise exception
```

Encoding:

1r: Register

0 1 0 0 1	1 1 1 1 1	1	1	. . . .

(*M or R*)

Conditions that raise an exception:

ET_ECALL
: `c != 0`.

22.45 EDU: Unconditionally disable event

Clears the event enabled status of a resource, disabling events and interrupts from that resource.

The instruction has one operand:

op1
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
EDU r
```

Operation:

```
enb(r) <- 0
thread(r) <- tid
```

Encoding:

1r: Register

0 0 0 0 0	1 1 1 1 1	1	0	. . . .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not referring to a legal resource, or the resource is not in use.

22.46 EEF: Enables events conditionally

Sets or clears the enabled event status of a resource. If the condition is 0 (false), events and interrupts are enabled, if the condition is not 0, events and interrupts are disabled.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
EEF d,r
```

Operation:

```
enb(r) <- d = 0
thread(r) <- tid
```

Encoding:

2r: Two register

0 0 1 0 1	. . . . .	.	1	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not referring to a legal resource, or the resource is not in use.

22.47 EET: Enable events conditionally

Sets or clears the enabled event status of a resource. If the condition is 0 (false), events and interrupts are disabled, if the condition is not 0, events and interrupts are enabled.

The instruction has two operands:

op1
d, Operand register, one of `r0`… `r11`

op2
r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
EET d,r
```

Operation:

```
enb(r) <- d != 0
thread(r) <- tid
```

Encoding:

2r: Two register

0 0 1 0 0	. . . . .	.	1	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
`r` is not referring to a legal resource, or the resource is not in use.

22.48 EEU: Unconditionally enable event

Sets the event enabled status of a resource, enabling events and interrupts from that resource.

The instruction has one operand:

op1
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
EEU r
```

Operation:

```
enb(r) <- 1
thread(r) <- tid
```

Encoding:

1r: Register

0 0 0 0 0	1 1 1 1 1	1	1	. . . .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `op2` is not referring to a legal resource, or the resource is not in use.

22.49 ELATE: Throw exception if too late

This instruction checks whether the operand is in the past, and raises an exception if it is the case. It can be used to implement timing assertions.

The instruction has one operand:

op1
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
ELATE s
```

Operation:

```
if !(s after current_time):
  raise exception
```

Encoding:

1r: Register

1 0 0 0 1	1 1 1 1 1	1	1	. . . .

(*M or R*)

Conditions that raise an exception:

ET_ECALL
: `s` is in the past.

22.50 ENDIN: End a current input

Allows any remaining input bits to be read of a port, and produces an integer stating how much data is left. The produced integer is the number of bits of data remaining; ie, This assumes that the port is buffering and shifting data.

The port-shift-count is set to the number of bits present, so an ENDIN instruction can be followed directly by an IN instruction without having to perform a *SETPSC*.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
ENDIN d,r
```

Operation:

```
d <- buffercount(r)
```

Encoding:

2r: Two register

1 0 0 1 0	. . . . .	.	1	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not referring to a legal resource, or the resource is not in use.

ET_ILLEGAL_RESOURCE
: `r` is referring to a port which is not in BUFFERS mode.

ET_ILLEGAL_RESOURCE
: `r` is referring to a port which is not in INPUT mode.

22.51 ENTSP: Adjust stack and save link register

Stores the link register on the stack then adjusts the stack pointer creating enough space for the procedure call that has just been entered.

See *RETSP* for the operation that restores the link-register.

The instruction has one operand:

op1
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
ENTSP u_{16}
```

Operation:

```
if u16 > 0:
  mem[sp] <- lr
  sp <- sp - u16 * Bpw
sr[di]<- 0
```

Encoding:

u6: 6-bit immediate

0 1 1 1 0	1	1 1 0 1	. . . . . .

(*M*)

lu6: 16-bit immediate

1 1 1 1 0	0	. . . .

0 1 1 1 0	1	1 1 0 1	. . . . . .

(*M and R*)

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_LOAD_STORE
: The indexed address is unaligned, or does not point to a valid memory address.

22.52 EQ: Equal

Performs a test on whether two words are equal. If the two operands are equal, `1` is produced in the destination register, otherwise `0` is produced.

The instruction has three operands:

op1
: c, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
EQ c,x,y
```

Operation:

```
c <- 1,      if x == y
      0,      if x != y
```

Encoding:

3r: Three register

0 0 1 1 0	. . . . .	. .	. .	. .

(*M or R*)

22.53 EQI: Equal immediate

Performs a test on whether two words are equal. If the two operands are equal, `1` is produced in the destination register, otherwise `0` is produced.

The instruction has three operands:

op1
: c, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: us, An integer in the range 0…11

Mnemonic and operands:

```
EQI c,x,u_s
```

Operation:

```
c <- 1,     if x == us
      0,     if x != us
```

Encoding:

2rus: Two register with immediate

1 0 1 1 0	. . . . .	. .	. .	. .

(*M or R*)

22.54 EXTDP: Extend data

Extends the data area by moving the data pointer to a lower address

The instruction has one operand:

op1
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
EXTDP u_{16}
```

Operation:

```
dp<- dp - u16 * Bpw
```

Encoding:

u6: 6-bit immediate

0 1 1 1 0	0	1 1 1 0	. . . . . .

(*M or R*)

lu6: 16-bit immediate

1 1 1 1 0	0	. . . .

0 1 1 1 0	0	1 1 1 0	. . . . . .

(*M and R*)

The latter is prefixed for long immediates.

22.55 EXTSP: Extend stack

Extends the stack by moving the stack pointer to a lower address.

The instruction has one operand:

op1
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
EXTSP u_{16}
```

Operation:

```
sp<- sp - u16 * Bpw
```

Encoding:

u6: 6-bit immediate

0 1 1 1 0	1	1 1 1 0	. . . . . .

(*M or R*)

lu6: 16-bit immediate

1 1 1 1 0	0	. . . .

0 1 1 1 0	1	1 1 1 0	. . . . . .

(*M and R*)

The latter is prefixed for long immediates.

22.56 FADD: Floating point addition

Adds two floating point numbers. The result is rounded according to the IEEE 754 roundTiesToEven mode.

If NaN is presented as **x**, then this value with bit 22 set will be passed on as the result into **d**. If **x** is a number, and **y** is NaN, then the value of **y** with bit 22 set will be used as the result for **d**. Otherwise, if the operation results in not a number, then it will set bits 22..30, and fill bits 0..21 with the bottom 23 bits of the next PC.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
FADD d,x,y
```

Operation:

```
d <- x + y    # Float
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .
1 0 1 1 0	1 1 1 1 1	1	0	1 1 1 0

(*M and R*)

22.57 FENAN: Exception if Not-a-number

Traps if the floating point value is not a number

The instruction has one operand:

op1
s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
FENAN s
```

Operation:

```
nop
```

Encoding:

1r: Register

1 0 1 1 0	1 1 1 1 1	1	1	. . . .

(*M*)

Conditions that raise an exception:

ET_ARITHMETIC
`s` is not-a-number.

22.58 FEQ: Floating point comparison

Compares two floating point numbers according to IEEE 754 and returns 1 if the two input operands are equal, and 0 otherwise

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: l, Operand register, one of `r0`… `r11`

op3
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
FEQ d,l,r
```

Operation:

```
d <- l = r
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .	(*M and R*)
1 1 0 0 0	1 1 1 1 1	1 0	1 1	1 0	

22.59 FGT: Floating point comparison

Compares two floating point numbers according to IEEE 754 and returns 1 if `d` is greater than `l`, and 0 otherwise

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: l, Operand register, one of `r0`… `r11`

op3
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
FGT d,l,r
```

Operation:

```
d <- l > r
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .
1 0 1 1 1	1 1 1 1 1	1 0	1 1	1 0

(*M and R*)

22.60 FLT: Floating point comparison

Compares two floating point numbers according to IEEE 754 and returns 1 if `d` is less than `l`, and 0 otherwise

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: l, Operand register, one of `r0`… `r11`

op3
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
FLT d,l,r
```

Operation:

```
d <- l < r
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .
1 0 1 1 1	1 1 1 1 1	1 0	1 1	1 1

(*M and R*)

22.61 FLUSH: Flushes the entire contents of the cache

The instruction has no operands.

Mnemonic and operands:

```
FLUSH
```

Operation:

```
nop
```

Encoding:

0r: No operands

0 0 1 1 0	1 1 1 1 1	1	0	1 1 1 1

(*M*)

22.62 FMACC: Floating point fused Multiply Accumulate

Computes the product of the last two operands, added in full precision to the first operand. The final number is rounded according to the IEEE 754 roundTiesToEven mode.

If NaN is presented as **x**, then this value with bit 22 set will be passed on as the result into **d**. Otherwise, if **y** is NaN, then the value of **y** with bit 22 set will be used as the result for **d**. Otherwise, if **z** is NaN, then the value of **z** with bit 22 set will be used as the result for **d**. Otherwise, if the operation results in not a number, then it will set bits 22..30, and fill bits 0..21 with the bottom 23 bits of the next PC.

The instruction has four operands:

op1
: d, Operand register, one of `r0`… `r11`

op4
: z, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

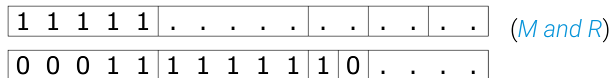
```
FMACC d,z,x,y
```

Operation:

```
z <- d + x * y    # Float
```

Encoding:

l4r: Four register long



(*M and R*)

22.63 FMAKE: Build a floating point number

Creates a floating point number given a sign, exponent and double length mantissa. The number produced wil have a value closest to `(-1)^(s &1) * (m_h:m_l) * 2^-23 * 2^e`. The `2^-23` value is specific for single precision floating point, for `bpw=32`.

If `m_h` and `m_l` are zero, then zero with the appropriate sign is produced in `d`.

Otherwise, for `bpw=32`, the exponent is calculated as `e - 127 + 40 - clz(m_h:m_l)`. That is, assuming that `m_h` is zero, and `m_l` contains a mantissa with an implicit one bit in location 23, then the exponent is set to `e-127`, otherwise the magnitude of the mantissa is used to adjust the exponent. An exponent of less than 1 indicates that a subnormal number has to be created, an exponent larger than `254` indicates that inifinity has to be generated.

Once the exponent is calculated, the mantissa is shifted so that bit 23 of `m_l` is one. The sign-bit (bit 0 of `s`), exponent (as computed above), and the bottom half of the mantissa (`m_l`, shifted with bits of `m_h` as appropriate) are now used to form a floating point number. Whilst making a floating point number, the lower bits of the mantissa are used for rounding, which may cause an adjustment to the exponent, which may cause infinity.

Observe that a sequence FSEXP, FMANT, FMAKE is a no-op if the input is an actual floating point number.

The instruction has five operands:

op1
: d, Operand register, one of `r0`… `r11`

op4
: s, Operand register, one of `r0`… `r11`

op2
: e, Operand register, one of `r0`… `r11`

op3
: m_h, Operand register, one of `r0`… `r11`

op5
: m_l, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
FMAKE d,s,e,m_h,m_l
```

Operation:

```
d <- (-1)^(s&1) * (m_h:m_l) * 2^-23 * 2^e   # Float
                 Assuming bpw=32
```

Encoding:

l5r: Five register long

1 1 1 1 1	. . . . .	. .	. .	. .
0 0 0 1 0	. . . . .	. 0	. .	. .

(*M and R*)

22.64 FMANT: Extract mantissa

Extracts the mantissa from a single precision floating point number, assuming it is represented according to IEEE 754. If the number represents plus or minus 0, then zero is produced. Otherwise, if the number is *normal* then the mantissa is produced including its implicit one bit in the bit value associated with `1.0`. Finally, if the number is *subnormal*, then the mantissa is shifted up so that the first '1' bit appears in the bit value associated with `1.0`

This instruction will trap if the operand is not-a-number or infinite. Use FSPEC to check on special cases prior to using FMANT.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

Mnemonic and operands:

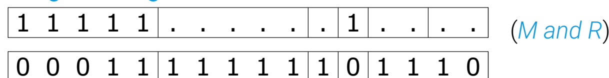
```
FMANT d,x
```

Operation:

```
d <-  x[22...0] << (23-k),   if x[30:23] == 0
      x[22...0] + 0x800000,        if x[30:23] != 0
      k is the highest k < 23 such that x[k]==1
```

Encoding:

l2r: Two register long



(*M and R*)

Conditions that raise an exception:

ET_ARITHMETIC
: `x` is infinite or not-a-number.

22.65 FMUL: Floating point multiply

Multiplies two floating point numbers. The result is rounded according to the IEEE 754 roundTiesToEven mode.

If NaN is presented as **x**, then this value with bit 22 set will be passed on as the result into **d**. If **x** is a number, and **y** is NaN, then the value of **y** with bit 22 set will be used as the result for **d**. Otherwise, if the operation results in not a number, then it will set bits 22..30, and fill bits 0..21 with the bottom 23 bits of the next PC.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
FMUL d,x,y
```

Operation:

```
d <- x * y   # Float
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .
1 0 1 1 0	1 1 1 1 1	1 0	1 1	0 1

(*M and R*)

22.66 FREER: Free a resource

Frees a resource so that it can be reused. Only resources that have been previously allocated with *GETR* can be freed; in particular, ports and clock-blocks cannot be freed since they are not allocated.

FREER pauses when freeing a channel end that has outstanding transmit data.

The instruction has one operand:

op1
r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
FREER r
```

Operation:

```
inuse(r) <- 0
```

Encoding:

1r: Register

0 0 0 1 0	1 1 1 1 1	1	0	. . . .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
`r` is not referring to a legal resource

ET_ILLEGAL_RESOURCE
`r` is referring to a resource that cannot be freed

ET_ILLEGAL_RESOURCE
`r` is referring to a running thread

ET_ILLEGAL_RESOURCE
`r` is referring to a channel end on which no terminating CT_END token has been input and/or output, or which has data pending for input, or which has a thread waiting for input or output.

22.67 FREET: Free unsynchronised thread

Stops the thread that executes this instruction, and frees it. This must not be used by synchronised threads, which should terminate by using a combination of an *SSYNC* on the slave and an *MJOIN* on the master.

The instruction has no operands.

Mnemonic and operands:

```
FREET
```

Operation:

```
sr[inuse] <- 0
```

Encoding:

0r: No operands

0 0 0 0 0	1 1 1 1 1	1	0	1 1 1 1

(*R*)

22.68 FSEXP: Extract sign and exponent

Extracts the sign and exponent from a floating point number: `s` becomes the sign, `e` becomes the exponent. If `x` equals zero, then `e` is set to 0, and `s` is set to the sign of the zero. If the number represents a *subnormal* number, then the exponent is adjusted to match the normalised mantissa that FMANT will produce.

This instruction will trap if the operand is not-a-number or infinite. Use FSPEC to check on special cases prior to using FEXP.

The instruction has three operands:

op1
: s, Operand register, one of `r0`… `r11`

op2
: e, Operand register, one of `r0`… `r11`

op3
: x, Operand register, one of `r0`… `r11`

Mnemonic and operands:

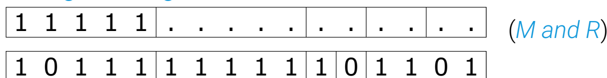
```
FSEXP s,e,x
```

Operation:

```
s <- 1,   if x < 0
      0,    if x >= 0
        e <- e[30...23] - 127,   if x != 0
              0,                if x == 0
              -127-(23-k),      if e[30...23] = 0
              k is the highest k < 23 such that x[k]==1
```

Encoding:

l3r: Three register long



1 1 1 1 1	. . . . .	. .	. .	. .
1 0 1 1 1	1 1 1 1 1	1 0	1 1	0 1

(*M and R*)

Conditions that raise an exception:

ET_ARITHMETIC
: `x` is infinite or not-a-number.

22.69 FSPEC: Identify floating point type

Identifies whether the floating point number is special or not. This instruction followed by a BRU and a jump table can quickly deal with all special cases, such as square-root of -1, divisions by zero or the sine of NaN.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
FSPEC d,x
```

Operation:

```
d <- 0,    if x > 0
         1,     if x == +0.0
         2,     if x == infinite
         3,     if IsSignallingNaN(s)
         4,     if x < 0
         5,     if x == -0.0
         6,     if x == -infinite
         7,     if IsQuietNaN(x)
```

Encoding:

l2r: Two register long

1 1 1 1 1	. . . . .	.	1	. .	. .
0 0 0 1 1	1 1 1 1 1	1	0	1 1	0 1

(*M and R*)

22.70 FSUB: Floating point subtraction

Subtracts two floating point numbers. The result is rounded according to the IEEE 754 roundTiesToEven mode.

If NaN is presented as **x**, then this value with bit 22 set will be passed on as the result into **d**. If **x** is a number, and **y** is NaN, then the value of **y** with bit 22 set will be used as the result for **d**. Otherwise, if the operation results in not a number, then it will set bits 22..30, and fill bits 0..21 with the bottom 23 bits of the next PC.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
FSUB d,x,y
```

Operation:

```
d <- x - y    # Float
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .
1 0 1 1 0	1 1 1 1 1	1	0	1 1 1 1

(*M and R*)

22.71 FUN: Floating point comparison

Compares two floating point numbers according to IEEE 754 and returns 1 if the two input operands are unordered, and 0 otherwise

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: l, Operand register, one of `r0`… `r11`

op3
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
FUN d,l,r
```

Operation:

```
d <- unordered(l, r)
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .	(*M and R*)
1 1 0 0 0	1 1 1 1 1	1 0	1 1	0 1	

22.72 GETD: Get resource data

Gets the contents of the data/dest/divide register of a resource. This data register is set using *SETD*. The way that a resource depends on its data register is resource dependent and described at *SETD*.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
GETD d,r
```

Operation:

```
d <- data(r)
```

Encoding:

l2r: Two register long

1 1 1 1 1	. . . . .	.	1	. .	. .
0 0 0 1 1	1 1 1 1 1	1	0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: **d** is not referring to a legal resource, or a resource which doesn't have a DATA register.

22.73 GETED: Get ED into r11

Obtains the value of `ed`, exception data, into `r11`. In the case of an event, `ed` is set to the environment vector stored in the resource by *SETEV*. The data that is stored in `ed` in the case of an exception is given in *XCore XS3 Exceptions*.

The instruction has no operands.

Mnemonic and operands:

```
GETED
```

Operation:

```
r11 <- ed
```

Encoding:

0r: No operands

0 0 0 0 1	1 1 1 1 1	1	1	1 1 1 0

(*M or R*)

22.74 GETET: Get ET into r11

Obtains the value of ET (exception type) into `r11`.

The instruction has no operands.

Mnemonic and operands:

```
GETET
```

Operation:

```
r11 <- et
```

Encoding:

0r: No operands

0 0 0 0 1	1 1 1 1 1	1	1	1 1 1 1

(*M or R*)

22.75 GETID: Get the thread's ID

Get the thread ID of this thread into `r11`.

The instruction has no operands.

Mnemonic and operands:

```
GETID
```

Operation:

```
r11 <- tid
```

Encoding:

0r: No operands

0 0 0 1 0	1 1 1 1 1	1	0	1 1 1 0

(*M or R*)

22.76 GETKEP: Get the Kernel Entry Point

Get the kernel entry point of this thread into `r11`.

The instruction has no operands.

Mnemonic and operands:

```
GETKEP
```

Operation:

```
r11 <- kep
```

Encoding:

0r: No operands

0 0 0 1 0	1 1 1 1 1	1	0	1 1 1 1

(*M or R*)

22.77 GETKSP: Get Kernel Stack Pointer

Gets the thread's Kernel Stack Pointer `ksp` into `r11`. There is no instruction to set `ksp` directly since it is normally not moved. SETSP followed by KRESTSP will set both `sp` and `ksp`. By saving `sp` beforehand, `ksp` can be set to the value found in `r0` by using the following code sequence:

```
LDAWSP r1, sp[0] // Save SP into R1

SETSP r0 // Set SP, and place old SP...

STW r1, sp[0] // ...where KRESTSP expects it

KRESTSP 0 // Set KSP, restore SP
```

The kernel stack pointer is initialised by the boot-ROM to point to a safe location near the last location of RAM - the last few locations are used by the JTAG debugging interface. If debugging is not required, then the KSP can safely be moved to the top of RAM.

The instruction has no operands.

Mnemonic and operands:

```
GETKSP
```

Operation:

```
r11 <- ksp
```

Encoding:

0r: No operands

0 0 0 1 0	1 1 1 1 1	1	1	1 1 0 0

(*M or R*)

22.78 GETN: Get network

Gets the network identifier that this channel-end belongs to.

The network identifier is set using *SETN*.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
GETN d,r
```

Operation:

```
d <- net(r)
```

Encoding:

l2r: Two register long

1 1 1 1 1	. . . . .	.	1	. .	. .
0 0 1 1 0	1 1 1 1 1	1	0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `d` is not referring to a legal channel end, or the channel end is not in use.

22.79 GETPS: Get processor state

Obtains internal processor state; used for low level debugging. The operand is a processor state resource; the register to be read is encoded in bits 15…8, and bits 7…0 should contain the resource type associated with processor state.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
GETPS d,r
```

Operation:

```
d <- PS[r]
```

Encoding:

l2r: Two register long

1 1 1 1 1	. . . . .	.	1	. .	. .
0 0 0 1 0	1 1 1 1 1	1	0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_ILLEGAL_PS
: d is not referring to a legal processor state register

22.80 GETR: Get a resource

Gets a resource of a specific type. This instruction dynamically allocates a resource from the pools of available resources. Not all resources are dynamically allocated; resources that refer to physical objects (IO pins, clock blocks) are used without allocating. The resource types are:

RES_TYPE_PORT	Ports	0	cannot be allocated
RES_TYPE_TIMER	Timers	1	
RES_TYPE_CHANEND	Channel ends	2	
RES_TYPE_SYNC	Synchronisers	3	
RES_TYPE_THREAD	Threads	4	
RES_TYPE_LOCK	Lock	5	
RES_TYPE_CLKBLK	Clock source	6	cannot be allocated
RES_TYPE_SWMEM	S/W mem	8	cannot be allocated
RES_TYPE_PS	Proc state	11	cannot be allocated
RES_TYPE_CONFIG	Config	12	cannot be allocated

The returned identifier comprises a 32-bit word, where the most significant 16-bits are resource specific data, followed by an 8-bit resource counter, and 8-bits resource-type. The resource specific 16 bits have the following meaning:

- Port: The width of the port.
- Timer: Reserved, returned as 0.
- Channel end: The node id (8-bits) and the core id (8-bits).
- Synchroniser: Reserved, returned as 0.
- Thread: Reserved, returned as 0.
- Lock: Reserved, returned as 0.
- Clock source: Reserved, should be set to 0.
- Processor state: Reserved, should be set to 0.
- Configuration: Reserved, should be set to 0.

If no resource of the requested type is available, then the destination operand is set to zero, otherwise the destination operand is set to a valid resource id .

If a channel end is allocated, a local channel end is returned. In order to connect to a remote channel end, a program normally receives a channel-end over an already connected channel, which is stored using *SETD*. To connect the first remote channel, a channel-end identifier can be constructed (by concatenating a node id, core id, channel-end and the value '2').

When allocated, resources are freed using *FREER* to allow them to be available for reallocation.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: us, An integer in the range 0…11

Mnemonic and operands:

```
GETR d,u_s
```

Operation:

```
d        <- first res in setof(us): !inuse(res)
inuse(d) <- 1
```

Encoding:

rus: Register with immediate

1 0 0 0 0	. . . . .	.	0	. .	. .

(*R*)

22.81 GETSR: Get bits from SR

Get bits from the thread's Status Register. The mask supplied specifies which bits should be extracted.

SETSR is used to set bits in the status register. The value of these bits are documented on the SETSR page.

The instruction has one operand:

op1
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
GETSR u_{16}
```

Operation:

```
r11 <-sr & u16
```

Encoding:

u6: 6-bit immediate

0 1 1 1 1	1	1 1 0 0	. . . . . .	(*M or R*)

lu6: 16-bit immediate

1 1 1 1 0	0	. . . .	(*M and R*)

0 1 1 1 1	1	1 1 0 0	. . . . . .

The latter is prefixed for long immediates.

22.82 GETST: Get a synchronised thread

Gets a new thread and binds it to a synchroniser. The synchroniser ID is passed as an operand to this instruction, and the destination register is set to the resulting thread ID. If no threads are available then the destination register is set to 0.

The thread is started on execution of *MSYNC* by the master thread.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
GETST d,r
```

Operation:

```
d         <- first thread in threads: ! inuse(thread)
inuse(d)  <- 1
spaused     <- spaused union {d}
slaves(r) <- slaves(r) union {d}
mstr(r)   <- tid
```

Encoding:

2r: Two register

0 0 0 0 0	. . . . .	.	1	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not referring to a synchroniser that is in use

22.83 GETTIME: Get the reference time

Gets the current value of the reference time and loads it into the specified register

The instruction has one operand:

op1
: d, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
GETTIME d
```

Operation:

```
d <- reference-time
```

Encoding:

1r: Register

1 0 0 0 1	1 1 1 1 1	1	0	. . . .

(*M or R*)

22.84 GETTS: Get the time stamp

Gets the time stamp of a port. This is the value of the port timer at which the previous transfer between the Shift and Transfer registers for input or output occurred. The port timer counts ticks of the clock associated with this port, and returns a 16-bit value. In the case of a conditional input, this instruction should be executed between a *WAITEU* and its associated *IN* instruction; the value returned by GETTS will be the timestamp of the data that will be input using the IN instruction.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
GETTS d,r
```

Operation:

```
d <- timestamp(r)
```

Encoding:

2r: Two register

0 0 1 1 1	. . . . .	.	0	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not referring to a port, or the port is not in use.

22.85 IN: Input data

Inputs data from a resource (`r`) into a destination register (`d`). The precise effect depends on the resource type:

- Port: Read data from the port. If the port is buffered, a whole word of data is returned. If the port is unbuffered, the most significant bits of the data will be set to 0. The thread pauses if the data is not available.
- Timer: Reads the current time from the timer, or pauses until after a specific time returning that time.
- Channel end: Reads `Bpw` data tokens from the channel, and concatenate them to a single word of data. The bytes are assumed to be transmitted most significant byte first. The thread pauses if there are not enough data tokens available.
- Lock: Lock the resource. The instruction pauses if the lock has been taken by another thread, and is released when the out is released.

This instruction may pause.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
IN d,r
```

Operation:

```
r :> d
```

Encoding:

2r: Two register

1 0 1 1 0	. . . . .	.	0	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not a valid resource, not in use, or it does not support IN.

ET_ILLEGAL_RESOURCE
: `r` is a channel end which contains a Control Token in the first `Bpw` tokens in its input buffer.

22.86 INCT: Input control tokens

If the next token on a channel is a control token, then this token is input to the destination register. If not, the instruction raises an exception.

This instruction pauses if the channel does not have a token of data available to input.

This instruction can be used together with *OUTCT* in order to implement robust protocols on channels.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
INCT d,r
```

Operation:

```
if hasctoken(r):
  r :> d
else:
  raise exception
```

Encoding:

2r: Two register

1 0 0 0 0	. . . . .	.	1	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not pointing to a channel resource, or the resource is not in use.

ET_ILLEGAL_RESOURCE
: `r` is a channel end which contains a data token in the first entry in its input buffer.

22.87 INPW: Input a part word

Inputs an incomplete word that is stored in the input buffer of a port. Used in conjunction with *ENDIN*. ENDIN is used to determine how many bits are left on the port, and this number is passed to INPW in order to read those remaining bits.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: r, Operand register, one of `r0`… `r11`

op3
: bitp, A bit position; one of `bpw`, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32

Mnemonic and operands:

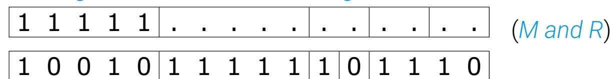
```
INPW d,r,bitp
```

Operation:

```
shiftcount(r) <- bitp
r :> d
```

Encoding:

l2rus: Two register with immediate long



(*M and R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not pointing to a port resource, or the resource is not in use, or `bitp` is an unsupported width, or the port is not in BUFFERS mode.

22.88 INSHR: Input and shift right

Inputs a value from a port, and shifts the data read into the most significant bits of the destination register. The bottom *port-width* bits of the destination register are lost.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
INSHR d,r
```

Operation:

```
r :> x
d <- x : d[bpw - 1...portwidth(r)]
```

Encoding:

2r: Two register

1 0 1 1 0	. . . . .	.	1	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not pointing to a port resource, or the resource is not in use.

22.89 INT: Input a token of data

If the next token on a channel is a data token, then this token is input into the destination register. If not, the instruction raises an exception.

This instruction pauses if the channel does not have a token of data available to input.

The instruction has two operands:

op1
d, Operand register, one of `r0`… `r11`

op2
r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
INT d,r
```

Operation:

```
if hasctoken(r):
  raise exception
else
  r :> d
```

Encoding:

2r: Two register

1 0 0 0 1	. . . . .	.	1	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
`r` is not pointing to a channel resource, or the resource is not in use.

ET_ILLEGAL_RESOURCE
`r` contains a control token in the first entry in its input buffer.

22.90 INVALIDATE: Invalidates the entire contents of the cache

The instruction has no operands.

Mnemonic and operands:

```
INVALIDATE
```

Operation:

```
nop
```

Encoding:

0r: No operands

0 0 1 1 0	1 1 1 1 1	1	1	1 1 1 1

(*M*)

22.91 KCALL: Kernel call

Performs a kernel call. The program counter, status register and exception data are stored in save-registers `spc`, `ssr`, and `sed` and the program continues at the kernel entry point. Similar to exceptions, the program counter that is saved on KCALL is the program counter of this instruction - hence an kernel call handler using KRET has to adjust `spc` prior to returning.

The instruction has one operand:

op1
: s, Operand register, one of `r0`... `r11`

Mnemonic and operands:

```
KCALL s
```

Operation:

```
spc <- pc
ssr <- sr
et <- ET_KCALL
sed <- ed
ed <- s
pc <- kep + 64
sr[ink] <- 1
sr[ieble] <- 0
sr[eeble] <- 0
```

Encoding:

1r: Register

0 1 0 0 0	1 1 1 1 1	1	0	. . . .

(*M*)

Conditions that raise an exception:

ET_KCALL
: Kernel call.

22.92 KCALLI: Kernel call immediate

Performs a kernel call. The program counter, status register and exception data are stored in save-registers `spc`, `ssr`, and `sed` and the program continues at the kernel entry point. Similar to exceptions, the program counter that is saved on KCALL is the program counter of this instruction - hence an kernel call handler using KRET has to adjust `spc` prior to returning.

The instruction has one operand:

op1
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
KCALLI u_{16}
```

Operation:

```
spc <- pc
ssr <- sr
et <- ET_KCALL
sed <- ed
ed <- u16
pc <- kep + 64
sr[ink] <- 1
sr[ieble] <- 0
sr[eeble] <- 0
```

Encoding:

u6: 6-bit immediate

0 1 1 1 0	0	1 1 1 1	. . . . . .

(*M*)

lu6: 16-bit immediate

1 1 1 1 0	0	. . . .

0 1 1 1 0	0	1 1 1 1	. . . . . .

(*M and R*)

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_KCALL
: Kernel call.

22.93 KENTSP: Switch to kernel stack

Saves the stack pointer on the kernel stack, then sets the stack pointer to the kernel stack.

KRESTSP is used to use the restore the original stack pointer from the kernel stack.

The instruction has one operand:

op1
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
KENTSP u_{16}
```

Operation:

```
mem[ksp] <- sp
sp <- ksp - n * Bpw
```

Encoding:

u6: 6-bit immediate

0 1 1 1 1	0	1 1 1 0	. . . . . .

(*M*)

lu6: 16-bit immediate

1 1 1 1 0	0	. . . .

0 1 1 1 1	0	1 1 1 0	. . . . . .

(*M and R*)

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_LOAD_STORE
: Register `ksp` points to an unaligned address, or does not point to a valid memory location.

22.94 KRESTSP: Restore stack pointer from kernel stack

Restores the stack pointer from the address saved on entry to the kernel by *KENTSP*. This instruction is also used to initialise the kernel-stack-pointer.

KENTSP is used to save the stack pointer on entry to the kernel.

The instruction has one operand:

op1
u16, A 16-bit mask

Mnemonic and operands:

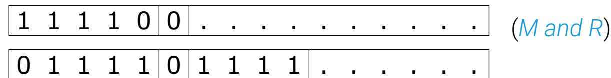
```
KRESTSP u16
```

Operation:

```
ksp <- sp + n * Bpw
sp <- mem[ksp]
```

Encoding:

lu6: 16-bit immediate



1 1 1 1 0	0	. . . .
0 1 1 1 1	0	1 1 1 1	. . . . . .

(*M and R*)

Conditions that raise an exception:

ET_LOAD_STORE
The indexed address points to an unaligned address, or the indexed address does not point to a valid memory location.

22.95 KRET: Kernel Return

Returns from the kernel after an interrupt, kernel call, or exception.

The instruction has no operands.

Mnemonic and operands:

```
KRET
```

Operation:

```
pc <- spc
sr <- ssr
ed <- sed
```

Encoding:

l0r: No operands

1 1 1 1 1	1 1 1 1 1	1	0	1 1 0 0
0 0 0 0 0	1 1 1 1 1	1	0	1 1 0 0

(*M and R*)

Conditions that raise an exception:

ET_ILLEGAL_PC
The register **spc** was not 16-bit aligned or did not point to a valid memory location.

22.96 LADD: Long unsigned add with carry

Adds two unsigned integers and a carry, and produces both the unsigned result and the possible carry. For this purpose, the instruction has five operands, two registers that contain the numbers to be added (`x` and `y`); the carry which is stored in the last bit of a third source operand (`v`); one destination register which is used to store the carry (`e`), and a destination register for the sum (`d`).

The instruction has five operands:

op1
: d, Operand register, one of `r0`… `r11`

op4
: e, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

op5
: v, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
LADD d,e,x,y,v
```

Operation:

```
r = x + y + v[0]
d <- r[bpw-1...0]
e <- r[bpw]
```

Encoding:

l5r: Five register long

1 1 1 1 1	. . . . .	. .	. .	. .	(*M and R*)
0 0 0 0 0	. . . . .	. 1	. .	. .	

22.97 LD8U: Load unsigned 8 bits

Loads an unsigned 8-bit value from memory. The address is computed using a base address (`b`) and index (`i`).

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: b, Operand register, one of `r0`… `r11`

op3
: i, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
LD8U d,b,i
```

Operation:

```
d <- 0:...:0:word[bnum + 7... bnum]
where ea = b + i
      bytenum = ea % Bpw
      bnum = 8 * bytenum
      word = mem[ea - bytenum]
```

Encoding:

3r: Three register

| 1 0 0 0 1 | . . . . . | . . | . . | . . | (*M*)

Conditions that raise an exception:

ET_LOAD_STORE
: The indexed address does not point to a valid memory location.

22.98 LD16S: Load signed 16 bits

Loads a signed 16-bit integer from memory extending the sign into the whole word. The address is computed using a base address (b) and index (i). The base address should be word-aligned.

The instruction has three operands:

op1
: d, Operand register, one of r0… r11

op2
: b, Operand register, one of r0… r11

op3
: i, Operand register, one of r0… r11

Mnemonic and operands:

```
LD16S d,b,i
```

Operation:

```
d <- word[bnum + 15]:...:word[bnum + 15]:word[bnum + 15... bnum]
where ea = b + i * 2
      bytenum = ea % Bpw
      bnum = 16 * (bytenum // 2)
      word = mem[ea-bytenum]
```

Encoding:

3r: Three register

| 1 0 0 0 0 | . . . . . | . . | . . | . . | (*M*)

Conditions that raise an exception:

ET_LOAD_STORE
: b is not 16-bit aligned (unaligned load), or does not point to a valid memory location.

22.99 LDA16B: Subtract from 16-bit address

Load effective address for a 16-bit value based on a base-address (`b`) and an index (`i`)

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: b, Operand register, one of `r0`… `r11`

op3
: i, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
LDA16B d,b,i
```

Operation:

```
d <- b - i * 2
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	.	.	. .	. .
0 0 1 1 0	1 1 1 1 1	1	0	1 1	0 0

(*M and R*)

22.100 LDA16F: Add to a 16-bit address

Load effective address for a 16-bit value based on a base-address (`b`) and an index (`i`)

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: b, Operand register, one of `r0`… `r11`

op3
: i, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
LDA16F d,b,i
```

Operation:

```
d <- b + i * 2
```

Encoding:

l3r: Three register long

1	1	1	1	1	.	.	.	.	.	.	.	.	.	.	.
0	0	1	0	1	1	1	1	1	1	1	0	1	1	0	0

(*M and R*)

22.101 LDAPB: Load backward pc-relative address

Load effective address relative to the program counter. This operation scales the index (`u10`) so that it counts 16-bit entities.

The instruction has one operand:

op1
: u10, A 20-bit immediate in the range 0…1048575. If `u20 < 1024`, the instruction requires no prefix

Mnemonic and operands:

```
LDAPB u10
```

Operation:

```
r11 <- pc - u10 * iw
```

Encoding:

u10: 10-bit immediate

1 1 0 1 1	1	. . . .

(*M or R*)

lu10: 20-bit immediate

1 1 1 1 0	0	. . . .

(*M and R*)

1 1 0 1 1	1	. . . .

The latter is prefixed for long immediates.

22.102 LDAPF: Load forward pc-relative address

Load effective address relative to the program counter. This operation scales the index (`u10`) so that it counts 16-bit entities.

The instruction has one operand:

op1
: u10, A 20-bit immediate in the range 0…1048575. If `u20 < 1024`, the instruction requires no prefix

Mnemonic and operands:

```
LDAPF u10
```

Operation:

```
r11 <- pc + u10 * iw
```

Encoding:

u10: 10-bit immediate

1 1 0 1 1	0	. . . .

(*M or R*)

lu10: 20-bit immediate

1 1 1 1 0	0	. . . .
1 1 0 1 1	0	. . . .

(*M and R*)

The latter is prefixed for long immediates.

22.103 LDAWB: Subtract from word address

Load effective address for word given a base-address (`b`) and an index (`i`)

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: b, Operand register, one of `r0`… `r11`

op3
: i, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
LDAWB d,b,i
```

Operation:

```
d <- b - i * Bpw
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	.	.	. .	. .
0 0 1 0 0	1 1 1 1 1	1	0	1 1	0 0

(*M and R*)

22.104 LDAWBI: Subtract from word address immediate

Load effective address for word given a base-address (b) and an index (u_s)

The instruction has three operands:

op1
: d, Operand register, one of r0… r11

op2
: b, Operand register, one of r0… r11

op3
: us, An integer in the range 0…11

Mnemonic and operands:

```
LDAWBI d,b,u_s
```

Operation:

```
d <- b - us * Bpw
```

Encoding:

l2rus: Two register with immediate long

1 1 1 1 1	. . . . .	. .	. .	. .

(*M and R*)

1 0 1 0 0	1 1 1 1 1	1	0	1 1 0 0

22.105 LDAWCP: Load address of word in constant pool

Loads the address of a word relative to the constant pointer.

The instruction has one operand:

op1
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
LDAWCP u_{16}
```

Operation:

```
r11 <- cp + u16 * Bpw
```

Encoding:

u6: 6-bit immediate

0 1 1 1 1	1	1 1 0 1	. . . . . .

(*M or R*)

lu6: 16-bit immediate

1 1 1 1 0	0	. . . .

0 1 1 1 1	1	1 1 0 1	. . . . . .

(*M and R*)

The latter is prefixed for long immediates.

22.106 LDAWDP: Load address of word in data pool

Loads the address of a word relative to the data pointer.

The instruction has two operands:

op1
: D, Any of `r0`… `r11`, `cp`, `dp`, `sp`, `lr`

op2
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
LDAWDP D,u_{16}
```

Operation:

```
D <- dp + u16 * Bpw
```

Encoding:

ru6: Register with 6-bit immediate

0 1 1 0 0	0	. . . .	. . . . . .	(*M or R*)

lru6: Register with 16-bit immediate

1 1 1 1 0	0	. . . .	(*M and R*)

0 1 1 0 0	0	. . . .	. . . . . .

The latter is prefixed for long immediates.

22.107 LDAWF: Add to a word address

Load effective address for word given a base-address (`b`) and an index (`i`).

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: b, Operand register, one of `r0`… `r11`

op3
: i, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
LDAWF d,b,i
```

Operation:

```
d <- b + i * Bpw
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .
0 0 0 1 1	1 1 1 1 1	1 0	1 1	0 0

(*M and R*)

22.108 LDAWFI: Add to a word address immediate

Load effective address for word given a base-address (`b`) and an index (`i`).

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: b, Operand register, one of `r0`… `r11`

op3
: i, An integer in the range 0…11

Mnemonic and operands:

```
LDAWFI d,b,i
```

Operation:

```
d <- b + i * Bpw
```

Encoding:

l2rus: Two register with immediate long

1 1 1 1 1	. . . . .	. .	. .	. .
1 0 0 1 1	1 1 1 1 1	1 0	1 1	0 0

(*M and R*)

22.109 LDAWSP: Load address of word on stack

Loads the address of a word relative to the stack pointer.

The instruction has two operands:

op1
: D, Any of `r0`… `r11`, `cp`, `dp`, `sp`, `lr`

op2
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
LDAWSP D,u_{16}
```

Operation:

```
D <- sp + u16 * Bpw
```

Encoding:

ru6: Register with 6-bit immediate

0 1 1 0 0	1	. . . .	. . . . . .	(*M or R*)

lru6: Register with 16-bit immediate

1 1 1 1 0	0	. . . .	(*M and R*)

0 1 1 0 0	1	. . . .	. . . . . .

The latter is prefixed for long immediates.

22.110 LDC: Load constant

Load a constant into a register

The instruction has two operands:

op1
: D, Any of `r0`… `r11`, `cp`, `dp`, `sp`, `lr`

op2
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
LDC D,u_{16}
```

Operation:

```
D <- u16
```

Encoding:

ru6: Register with 6-bit immediate

0 1 1 0 1	0	. . . .	. . . . . .	(*M or R*)

lru6: Register with 16-bit immediate

1 1 1 1 0	0	. . . .	(*M and R*)

0 1 1 0 1	0	. . . .	. . . . . .

The latter is prefixed for long immediates.

22.111 LDD: Load double word

Loads two words from memory, using a base and an index. The index is scaled in order to translate the double-word-index into a byte-index. The base address must be double-word-aligned. The immediate version, *LDDI*, implements a load from a structured data type; the version with registers only, *LDD*, implements a load from an array.

The instruction has four operands:

op1
: d, Operand register, one of `r0`... `r11`

op4
: e, Operand register, one of `r0`... `r11`

op2
: b, Operand register, one of `r0`... `r11`

op3
: i, Operand register, one of `r0`... `r11`

Mnemonic and operands:

```
LDD d,e,b,i
```

Operation:

```
d <- mem[b + i * Bpw * 2]
e <- mem[b + i * Bpw * 2 + Bpw]
```

Encoding:

l4r: Four register long

1 1 1 1 1	. . . . .	. .	. .	. .	(*M and R*)
0 0 1 0 0	1 1 1 1 1	1	0	. . . .	

Conditions that raise an exception:

ET_LOAD_STORE
: b is not double word aligned, or the indexed address does not point to a valid memory location.

22.112 LDDI: Load double word immediate

Loads two words from memory, using a base and an index. The index is scaled in order to translate the double-word-index into a byte-index. The base address must be double-word-aligned. The immediate version, *LDDI*, implements a load from a structured data type; the version with registers only, *LDD*, implements a load from an array.

The instruction has four operands:

op1
: d, Operand register, one of `r0`… `r11`

op4
: e, Operand register, one of `r0`… `r11`

op2
: b, Operand register, one of `r0`… `r11`

op3
: i, An integer in the range 0…11

Mnemonic and operands:

```
LDDI d,e,b,i
```

Operation:

```
d <- mem[b + i * Bpw * 2]
e <- mem[b + i * Bpw * 2 + Bpw]
```

Encoding:

l3rus: Three register with immediate long

1 1 1 1 1	. . . . .	. .	. .	. .	(*M and R*)
0 0 1 0 0	1 1 1 1 1	1 1	. . . .		

Conditions that raise an exception:

ET_LOAD_STORE
: b is not double word aligned, or the indexed address does not point to a valid memory location.

22.113 LDDSP: Load double word from stack

Loads two words relative to the stack pointer. The stack pointer must be double-word aligned.

The instruction has three operands:

op1
: d, Operand register, one of `r0`... `r11`

op2
: e, Operand register, one of `r0`... `r11`

op3
: us, An integer in the range 0...11

Mnemonic and operands:

```
LDDSP d,e,u_s
```

Operation:

```
d <- mem[sp + us * Bpw * 2]
e <- mem[sp + us * Bpw * 2 + Bpw]
```

Encoding:

l2rus: Two register with immediate long

1 1 1 1 1	. . . . .	. .	. .	. .	(*M and R*)
1 1 1 0 1	1 1 1 1 1	1 0	1 1	0 0	

Conditions that raise an exception:

ET_LOAD_STORE
: `sp` is not double-word aligned, or the indexed address does not point to a valid memory location.

22.114 LDET: Load ET from the stack

Restores the value of ET from the stack from offset 4.

The value was typically saved using *STET*. Together with *LDSPC*, *LDSSR*, and *LDSED* all or part of the state can be restored.

The instruction has no operands.

Mnemonic and operands:

```
LDET
```

Operation:

```
et <- mem[sp + 4 * Bpw]
```

Encoding:

0r: No operands

0 0 0 1 0	1 1 1 1 1	1	1	1 1 1 0

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
The indexed address does not point to a valid memory location.

22.115 LDIVU: Long unsigned divide

Divides a double word operand by a single word operand. This will result in a single word quotient and a single word remainder. This instruction has three source operands and two destination operands. The LDIVU instruction can take up to `bpw` thread-cycles to complete; the divide unit is shared between threads.

The operation only works if the division fits in a 32-bit word, that is, if the higher word of the double word input is less than the divisor. This operation is intended to be used for the implementation of long division.

The instruction has five operands:

op1
: d, Operand register, one of `r0`… `r11`

op4
: e, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

op5
: v, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
LDIVU d,e,x,y,v
```

Operation:

```
d <- (v : x) // y
e <- (v : x) % y
```

Encoding:

l5r: Five register long

1 1 1 1 1	. . . . .	. .	. .	. .	(*M and R*)
0 0 0 0 0	. . . . .	. 0	. .	. .	

Conditions that raise an exception:

ET_ARITHMETIC
: y = 0 || v >= y

22.116 LDSED: Load SED from stack

Restores the value of SED from the stack from offset 3.

The value was typically saved using *STSED*. Together with *LDSPC*, *LDSSR*, and *LDET* all or part of the state can be restored.

The instruction has no operands.

Mnemonic and operands:

```
LDSED
```

Operation:

```
sed <- mem[sp + 3 * Bpw]
```

Encoding:

0r: No operands

0 0 0 1 0	1 1 1 1 1	1	1	1 1 0 1

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
The indexed address does not point to a valid memory location.

22.117 LDSPC: Load the SPC from the stack

Restores the value of SPC from the stack from offset 1.

The value was typically saved using *STSPC*. Together with *LDSED*, *LDSSR*, and *LDET* all or part of the state can be restored.

The instruction has no operands.

Mnemonic and operands:

LDSPC

Operation:

```
spc <- mem[sp + 1 * Bpw]
```

Encoding:

0r: No operands

0 0 0 0 1	1 1 1 1 1	1	0	1 1 0 0

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
The indexed address does not point to a valid memory location.

22.118 LDSSR: Load SSR from stack

Restores the value of SSR from the stack from offset 2.

The value was typically saved using *STSSR*. Together with *LDSED*, *LDSPC*, and *LDET* all or part of the state can be restored.

The instruction has no operands.

Mnemonic and operands:

LDSSR

Operation:

```
ssr <- mem[sp + 2 * Bpw]
```

Encoding:

0r: No operands

0 0 0 0 1	1 1 1 1 1	1	0	1 1 1 0

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
The indexed address does not point to a valid memory location.

22.119 LDW: Load word

Loads a word from memory, using two registers as a base register and an index register. The index register is scaled in order to translate the word-index into a byte-index. The base address must be word-aligned. The immediate version, *LDWI*, implements a load from a structured data type; the version with registers only, *LDW*, implements a load from an array.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: b, Operand register, one of `r0`… `r11`

op3
: i, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
LDW d,b,i
```

Operation:

```
d <- mem[b + i * Bpw]
```

Encoding:

3r: Three register

0 1 0 0 1	. . . . .	. .	. .	. .

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
: b is not word aligned, or the indexed address does not point to a valid memory location.

22.120 LDWI: Load word immediate

Loads a word from memory, using two registers as a base register and an index register. The index register is scaled in order to translate the word-index into a byte-index. The base address must be word-aligned. The immediate version, *LDWI*, implements a load from a structured data type; the version with registers only, *LDW*, implements a load from an array.

The instruction has three operands:

op1
: d, Operand register, one of `r0`... `r11`

op2
: b, Operand register, one of `r0`... `r11`

op3
: i, An integer in the range 0...11

Mnemonic and operands:

```
LDWI d,b,i
```

Operation:

```
d <- mem[b + i * Bpw]
```

Encoding:

2rus: Two register with immediate

| 0 0 0 0 1 | . . . . . | . . | . . | . . | (*M*)

Conditions that raise an exception:

ET_LOAD_STORE
: b is not word aligned, or the indexed address does not point to a valid memory location.

22.121 LDWCP: Load word from constant pool

Loads a word relative to the constant pool pointer.

The instruction has two operands:

op1
: D, Any of `r0`... `r11`, `cp`, `dp`, `sp`, `lr`

op2
: u16, A 16-bit immediate in the range 0...65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
LDWCP D,u_{16}
```

Operation:

```
D <- mem[cp + u16 * Bpw]
```

Encoding:

ru6: Register with 6-bit immediate

0 1 1 0 1	1	. . . .	. . . . . .	(*M*)

lru6: Register with 16-bit immediate

1 1 1 1 0	0	. . . .	(*M and R*)

0 1 1 0 1	1	. . . .	. . . . . .

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_LOAD_STORE
: `cp` is not word aligned, or the indexed address does not point to a valid memory location.

22.122 LDWCPL: Load word from large constant pool

Loads a word relative to the constant pool pointer into `r11`. The offset can be larger than the offset specified in *LDWCP*.

The instruction has one operand:

op1
: u10, A 20-bit immediate in the range 0…1048575. If `u20 < 1024`, the instruction requires no prefix

Mnemonic and operands:

```
LDWCPL u10
```

Operation:

```
r11 <- mem[cp + u10 * Bpw]
```

Encoding:

u10: 10-bit immediate

1 1 1 0 0	1	. . . .

(*M*)

lu10: 20-bit immediate

1 1 1 1 0	0	. . . .

1 1 1 0 0	1	. . . .

(*M and R*)

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_LOAD_STORE
: `cp` is not word aligned, or the indexed address does not point to a valid memory location.

22.123 LDWDP: Load word form data pool

Loads a word relative to the data pointer.

The instruction has two operands:

op1
: D, Any of `r0`… `r11`, `cp`, `dp`, `sp`, `lr`

op2
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
LDWDP D,u_{16}
```

Operation:

```
D <- mem[dp + u16 * Bpw]
```

Encoding:

ru6: Register with 6-bit immediate

0 1 0 1 1	0	. . . .	. . . . . .

(*M*)

lru6: Register with 16-bit immediate

1 1 1 1 0	0	. . . .

0 1 0 1 1	0	. . . .	. . . . . .

(*M and R*)

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_LOAD_STORE
: `dp` is not word aligned, or the indexed address does not point to a valid memory location.

22.124 LDWSP: Load word from stack

Loads a word relative to the stack pointer.

The instruction has two operands:

op1
: D, Any of `r0`… `r11`, `cp`, `dp`, `sp`, `lr`

op2
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
LDWSP D,u_{16}
```

Operation:

```
D <- mem[sp + u16 * Bpw]
```

Encoding:

ru6: Register with 6-bit immediate

0 1 0 1 1	1	. . . .	. . . . . .

(*M*)

lru6: Register with 16-bit immediate

1 1 1 1 0	0	. . . .

0 1 0 1 1	1	. . . .	. . . . . .

(*M and R*)

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_LOAD_STORE
: `sp` is not word aligned, or the indexed address does not point to a valid memory location.

22.125 LEXTRACT: Bitfield extraction from register pair

Extracts a bitfield at position **x** in a pair of registers **l** and **r** into **d**. A mask **bitp** is applied allowing a bitfield of less than **bpw** bits to be extracted.

The instruction has five operands:

op1
: d, Operand register, one of **r0**… **r11**

op4
: l, Operand register, one of **r0**… **r11**

op2
: r, Operand register, one of **r0**… **r11**

op3
: x, Operand register, one of **r0**… **r11**

op5
: bitp, A bit position; one of **bpw**, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32

Mnemonic and operands:

```
LEXTRACT d,l,r,x,bitp
```

Operation:

```
d <- (l:r)[bpw+x-1..x] & (2^bitp-1);
```

Encoding:

l4rus: Four registers with immediate long

1 1 1 1 1	. . . . .	. .	. .	. .	(*M and R*)
0 0 0 1 1	x x x x x	. 0	. .	. .	

22.126 LINSERT: Inserts a bitfield into a pair of registers

Inserts a bitfield into a pair of registers `d` and `e`. The bitfield is stored in register `x`, the location of the bitfield is stored in register `s` (which must be between 0 and `bpw-1` inclusive), and the length of the bitfield is a short immediate operand `bitp`.

The instruction has five operands:

op1
: d, Operand register, one of `r0`... `r11`

op4
: e, Operand register, one of `r0`... `r11`

op2
: x, Operand register, one of `r0`... `r11`

op3
: s, Operand register, one of `r0`... `r11`

op5
: bitp, A bit position; one of `bpw`, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32

Mnemonic and operands:

```
LINSERT d,e,x,s,bitp
```

Operation:

```
m = ((1 << bitp) - 1)  << s
d:e <- ((d:e) & ~ m) | ((x << s) & m)
```

Encoding:

l4rus: Four registers with immediate long

1 1 1 1 1	. . . . .	. .	. .	. .	(*M and R*)
0 0 0 1 1	x x x x x	. 1	. .	. .	

22.127 LMUL: Long multiply

Multiplies two words to produce a double-word, and adds two single words. Both the high word and the low word of the result are produced. This multiplication is unsigned and cannot overflow.

The instruction has six operands:

op1
: d, Operand register, one of `r0`… `r11`

op4
: e, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

op5
: v, Operand register, one of `r0`… `r11`

op6
: w, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
LMUL d,e,x,y,v,w
```

Operation:

```
e <- r [bpw-1 ... 0]
d <- r [2 bpw-1 ... bpw]
    where r = x * y + v + w
```

Encoding:

l6r: Six register long

1 1 1 1 1	. . . . .	. .	. .	. .
0 0 0 0 0	. . . . .	. .	. .	. .

(*M and R*)

22.128 LSS: Less than signed

Tests whether one signed value is less than another signed value. The test result is produced in the destination register (`c`) as 1 (true) or 0 (false).

The instruction has three operands:

op1
: c, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
LSS c,x,y
```

Operation:

```
c <- 1,    if x < y    # signed comparison
     0,    if x >= y   # signed comparison
```

Encoding:

3r: Three register

1 1 0 0 0	. . . . .	. .	. .	. .

(*M or R*)

22.129 LSU: Less than unsigned

Tests whether one unsigned value is less than another unsigned value. The result is produced in the destination register (`c`) as 1 (true) or 0 (false). It can be used to perform efficient bound checks against values in the range `0...(y-1)`

The instruction has three operands:

op1
: c, Operand register, one of `r0`... `r11`

op2
: x, Operand register, one of `r0`... `r11`

op3
: y, Operand register, one of `r0`... `r11`

Mnemonic and operands:

```
LSU c,x,y
```

Operation:

```
c <- 1,    if x < y
     0,    if x >= y
```

Encoding:

3r: Three register

1 1 0 0 1	. . . . .	. .	. .	. .

(*M or R*)

22.130 LSUB: Long unsigned subtract

Subtracts unsigned integers and a borrow from an unsigned integer, producing both the unsigned result and the possible borrow. The instruction has five operands: two registers that contain the numbers to be subtracted (x and y), the borrow input which is stored in the last bit of a third source operand (v), one destination register which is used to store the borrow-out (e), and a destination register for the difference (d).

The instruction has five operands:

op1
: d, Operand register, one of `r0`… `r11`

op4
: e, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

op5
: v, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
LSUB d,e,x,y,v
```

Operation:

```
d <- r[bpw-1...0]
e <- r[bpw]
     where r = x - y - v[0]
```

Encoding:

l5r: Five register long

1 1 1 1 1	. . . . .	. .	. .	. .	(*M and R*)
0 0 0 0 1	. . . . .	. 0	. .	. .	

22.131 MACCS: Multiply and accumulate signed

Multiplies two signed words, and adds the double word result into a signed double word accumulator. The double word accumulator comprises two registers that are used both as a source and destination. Two other operands are the values that are to be multiplied.

The instruction has four operands:

op1
: d, Operand register, one of `r0`… `r11`

op4
: e, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

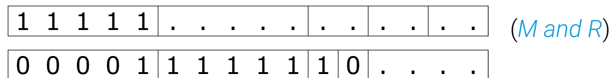
```
MACCS d,e,x,y
```

Operation:

```
e <- tmp[bpw-1...0]
d <- tmp[2 * bpw-1...bpw]
    where tmp = (d:e) + x * y    # signed
```

Encoding:

l4r: Four register long



(*M and R*)

22.132 MACCU: Multiply and accumulate unsigned

Multiplies two unsigned words, and adds the double word result into an unsigned double word accumulator. The double word accumulator comprises two registers that are used both as a source and destination. Two other operands are the values that are to be multiplied.

MACCU can be used to correct word alignment issues by repeatedly operating on words of a stream. For example, multiplying with 0x00010000 will result in the high word of the accumulator to produce the same stream of words offset by half a word.

The instruction has four operands:

op1
: d, Operand register, one of `r0`… `r11`

op4
: e, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
MACCU d,e,x,y
```

Operation:

```
e <- tmp[bpw-1...0]
d <- tmp[2 * bpw-1...bpw]
    where tmp = (d:e) + x * y
```

Encoding:

l4r: Four register long

1 1 1 1 1	. . . . .	. .	. .	. .
0 0 0 0 0	1 1 1 1 1	1	1	. . . .

(*M and R*)

22.133 MJOIN: Synchronise and join

Synchronises the master thread that executes this instruction with all the slave threads associated with its synchroniser operand (r), and frees those slave threads when the synchronisation completes. This is used to end a group of parallel threads. Note this clears the EEBLE bit. If the ININT bit is set, then MJOIN will not block; MJOIN should not be used inside an interrupt handler.

The slaves execute an *SSYNC* instruction to synchronise. The master can execute an *MSYNC* instruction to synchronise without freeing the slave threads.

The instruction has one operand:

op1
: r, Operand register, one of `r0`... `r11`

Mnemonic and operands:

```
MJOIN r
```

Operation:

```
sr[eeble] <- 0
if slaves(r) == spaused:
  for thread in slaves(r):
    inuse(thread) <- 0
  mjoin(syn(tid)) <- 0
else:
  mpaused <- mpaused UNION {tid}
  mjoin(r) <- 1
  msyn(r) <- 1
```

Encoding:

1r: Register

0 0 0 1 0	1 1 1 1 1	1	1	. . . . .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not a synchroniser resource, or the resource is not in use.

22.134 MKMSK: Make n-bit mask

Makes an n-bit mask that can be used to extract a bit field from a word. The resulting mask consists of `s 1` bits aligned to the right.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
MKMSK d,s
```

Operation:

```
d <- 2^{s}-1,         if s < bpw
     ~0,                if s >= bpw
```

Encoding:

2r: Two register

1 0 1 0 0	. . . . .	.	0	. .	. .

(*M or R*)

22.135 MKMSKI: Make n-bit mask immediate

Makes an n-bit mask that can be used to extract a bit field from a word. The resulting mask consists of `bitp 1` bits aligned to the right.

The instruction has two operands:

op1
: d, Operand register, one of `r0`... `r11`

op2
: bitp, A bit position; one of `bpw`, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32

Mnemonic and operands:

```
MKMSKI d,bitp
```

Operation:

```
d <- 2^{bitp}-1,         if bitp < bpw
     ~0,                if bitp >= bpw
```

Encoding:

rus: Register with immediate

1 0 1 0 0	. . . . .	.	1	. .	. .

(*M or R*)

22.136 MSYNC: Master synchronise

Synchronise a master thread with the slave threads associated with its synchroniser (`r`). If the slave threads have just been created (with *GETST*), then MSYNC starts all slaves. This clears the EEBLE bit. If the ININT bit is set, then MSYNC will not block; MSYNC should not be used inside an interrupt handler.

The slaves execute an *SSYNC* instruction to synchronise. The master can execute an *MJOIN* instruction to free the slave threads after synchronisation.

The instruction has one operand:

op1
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
MSYNC r
```

Operation:

```
sr[eeble] <- 0
  if slaves(r) == spaused:
    spaused <- spaused setminus slaves(r)
  else:
    mpaused <- mpaused UNION {tid}
    msyn(r) <- 1
```

Encoding:

1r: Register

0 0 0 1 1	1 1 1 1 1	1	1	. . . .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not a synchroniser resource, or the resource is not in use.

ET_ILLEGAL_PC
: One or more of the slave threads do not have a legal program counter.

22.137 MUL: Unsigned multiply

Performs a single word unsigned multiply. Any overflow is discarded, and only the last bpw bits of the result are produced.

If overflow is important, one of the *LMUL*, *MACCU* or *MACCS* instructions should be used.

The instruction has three operands:

op1
: d, Operand register, one of r0… r11

op2
: x, Operand register, one of r0… r11

op3
: y, Operand register, one of r0… r11

Mnemonic and operands:

```
MUL d,x,y
```

Operation:

```
d <- (x * y) % 2^bpw
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .
0 0 1 1 1	1 1 1 1 1	1 0	1 1	0 0

(*M and R*)

22.138 NEG: Two's complement negate

Performs a signed negation in two's complement, ie, it computes `0-s`. Overflow is ignored, ie, Negating `-2^{bpw-1}` will produce `-2^{bpw-1}`.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
NEG d,s
```

Operation:

```
d <- 2^bpw-s
```

Encoding:

2r: Two register

1 0 0 1 0	. . . . .	.	0	. .	. .

(*M or R*)

22.139 NOP: No operation

No operation.

The instruction has no operands.

Mnemonic and operands:

```
NOP
```

Operation:

```
No operation
```

Encoding:

0r: No operands

0 0 0 1 0	1 1 1 1 1	1	1	1 1 1 1

(*M or R*)

22.140 NOT: Bitwise not

Produces the bitwise not of its source operand.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
NOT d,s
```

Operation:

```
d <- ~ s
```

Encoding:

2r: Two register

1 0 0 0 1	. . . . .	.	0	. .	. .

(*M or R*)

22.141 OR: Bitwise or

Produces the bitwise or of its two source operands.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
OR d,x,y
```

Operation:

```
d <- x | y
```

Encoding:

3r: Three register

0 1 0 0 0	. . . . .	. .	. .	. .	(*M or R*)

22.142 OUT: Output data

Output data to a resource. The precise effect of this instruction depends on the resource:

- Port: Output a word to the port - if the port is buffered the data will be shifted out piece-meal, if the port is unbuffered the most significant bits of the data outputted will be ignored. The instruction pauses if the out data cannot be accepted.
- Channel end: Output Bpw data tokens to the destination associated with this channel-end (see *SETD*) - the most significant byte of the word is output first. The instruction pauses if the out data cannot be accepted.
- Lock: Releases the lock.

The instruction has two operands:

op1
r, Operand register, one of `r0`… `r11`

op2
s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
OUT r,s
```

Operation:

```
r <: s
```

Encoding:

r2r: Two register reversed

1 0 1 0 1	. . . . .	.	0	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
`r` is not a valid resource, not in use, or it does not support OUT.

ET_LINK_ERROR
`r` is a channel end, and the destination has not been set.

22.143 OUTCT: Output a control token

Outputs a control token to a channel.

The instruction pauses if the control token cannot be accepted by the channel.

Each OUTCT must have a matching *CHKCT* or *INCT*

The instruction has two operands:

op1
: r, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
OUTCT r,s
```

Operation:

```
r <: ctoken(s)
```

Encoding:

2r: Two register

0 1 0 0 1	. . . . .	.	0	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not a channel end, or not in use.

ET_LINK_ERROR
: `r` is a channel end, and the destination has not been set.

ET_LINK_ERROR
: `r` is a channel end, and the control token is a reserved hardware token.

22.144 OUTCTI: Output a control token immediate

Outputs a control token to a channel.

The instruction pauses if the control token cannot be accepted by the channel.

Each OUTCT must have a matching *CHKCT* or *INCT*

The instruction has two operands:

op1
: r, Operand register, one of `r0`… `r11`

op2
: us, An integer in the range 0…11

Mnemonic and operands:

```
OUTCTI r,u_s
```

Operation:

```
r <: ctoken(us)
```

Encoding:

rus: Register with immediate

0 1 0 0 1	. . . . .	.	1	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not a channel end, or not in use.

ET_LINK_ERROR
: `r` is a channel end, and the destination has not been set.

ET_LINK_ERROR
: `r` is a channel end, and the control token is a reserved hardware token.

22.145 OUTPW: Output a part word

Outputs a partial word to a port. This is useful to send the last few port-widths of data.

The instruction pauses if the out data cannot be accepted.

The instruction has three operands:

op1
: s, Operand register, one of `r0`… `r11`

op2
: r, Operand register, one of `r0`… `r11`

op3
: w, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
OUTPW s,r,w
```

Operation:

```
shiftcount(r) <- w
r  <: s
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .
1 1 0 0 1	1 1 1 1 1	1 0	1 1	0 1

(*M and R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not pointing to a port resource, or the resource is not in use, or `w` is an unsupported width, or the port is not in BUFFERS mode.

22.146 OUTPWI: Output a part word immediate

Outputs a partial word to a port. This is useful to send the last few port-widths of data.

The instruction pauses if the out data cannot be accepted.

The instruction has three operands:

op1
: s, Operand register, one of `r0`... `r11`

op2
: r, Operand register, one of `r0`... `r11`

op3
: bitp, A bit position; one of `bpw`, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32

Mnemonic and operands:

```
OUTPWI s,r,bitp
```

Operation:

```
shiftcount(r) <- bitp
r  <: s
```

Encoding:

l2rus: Two register with immediate long

1 1 1 1 1	. . . . .	. .	. .	. .
1 0 0 1 0	1 1 1 1 1	1 0	1 1	0 1

(*M and R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not pointing to a port resource, or the resource is not in use, or `bitp` is an unsupported width, or the port is not in BUFFERS mode.

22.147 OUTSHR: Output data and shift

Outputs the least significant *port-width* bits of a register to a port, shifting the register contents to the right by that number of bits.

The instruction pauses if the out data cannot be accepted.

The instruction has two operands:

op1
: r, Operand register, one of `r0`… `r11`

op2
: d, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
OUTSHR r,d
```

Operation:

```
r  <: d[portwidth(r)-1...0]
d <- 0 : ... : 0 : d[bpw - 1...portwidth(r
```

Encoding:

r2r: Two register reversed

1 0 1 0 1	. . . . .	.	1	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not pointing to a port resource, or the resoruce is not in use.

22.148 OUTT: Output a token

Output a data token to a channel.

The instruction pauses if the output token cannot be accepted.

The instruction has two operands:

op1
: r, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
OUTT r,s
```

Operation:

```
r <: dtoken(s)
```

Encoding:

r2r: Two register reversed

0 0 0 0 1	. . . . .	.	1	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not a channel end or not in use.

ET_LINK_ERROR
: `r` is a channel end, and the destination has not been set.

22.149 PEEK: Peek at port data

Looks at the value of the port pins, by-passing all input logic. Peek will not pause, and will not take ownership of the port.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
PEEK d,r
```

Operation:

```
d <- pins(r)
```

Encoding:

2r: Two register

1 0 1 1 1	. . . . .	.	0	. .	. .

(*R*)

Conditions that raise an exception:

ET_ILLEGAL_RESOURCE
: `r` is not a port resource, or the resource is not in use.

22.150 PREFETCH: Prefetches a load from external memory

The instruction has no operands.

Mnemonic and operands:

```
PREFETCH
```

Operation:

```
nop
```

Encoding:

0r: No operands

0 0 1 1 0	1 1 1 1 1	1	1	1 1 1 0

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
: `r11` is not word aligned, or a complete vector cannot be loaded from this address.

22.151 REMS: Signed remainder

Computes a signed integer remainder. The remainder is negative if the dividend is negative. For example 5 rem 3 is 2, -5 rem 3 is -2, -5 rem -3 is -2, and 5 rem -3 is 2.

This instruction does not execute in a single cycle, and multiple threads may share the same division unit. The remainder may take up to **bpw** thread-cycles.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

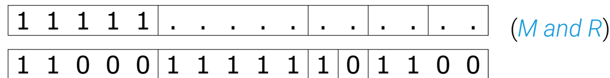
```
REMS d,x,y
```

Operation:

```
d <- x % y    # signed
```

Encoding:

l3r: Three register long



1 1 1 1 1	. . . . .	. .	. .	. .
1 1 0 0 0	1 1 1 1 1	1 0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_ARITHMETIC
: Remainder by `x` by `0`.

ET_ARITHMETIC
: Remainder by of `-2^{bpw-1}` by `-1`

22.152 REMU: Unsigned remainder

Computes an unsigned integer remainder.

This instruction does not execute in a single cycle, and multiple threads may share the same division unit. The division may take up to `bpw` thread-cycles.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
REMU d,x,y
```

Operation:

```
d <- x % y
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .

1 1 0 0 1	1 1 1 1 1	1	0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_ARITHMETIC
: Remainder of `x` by `0`.

22.153 RETSP: Return

Returns to the caller of this procedure, and (optionally) adjusts the stack. This instruction assumes that the return address is stored in LR (where call instructions leave the return address).

This instruction is used with *ENTSP*. The *BLA*, *BLACP*, *BLAT*, *BLRB* and *BLRF* instructions perform the opposite of this instruction, calling a procedure.

The instruction has one operand:

op1
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
RETSP u_{16}
```

Operation:

```
if u16 > 0:
  sp <- sp + u6 * Bpw
  lr <- mem[sp]
pc <- lr
sr[di] <- lr & 1
```

Encoding:

u6: 6-bit immediate

0 1 1 1 0	1	1 1 1 1	. . . . . .

(*M*)

lu6: 16-bit immediate

1 1 1 1 0	0	. . . .

0 1 1 1 0	1	1 1 1 1	. . . . . .

(*M and R*)

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_LOAD_STORE
: Register **sp** points to an unaligned address, or the indexed address does not point to a valid memory address.

22.154 LSATS: Saturate signed

Perform saturation on a double word value. Given a bit index this operation will check if any arithmetic has overflowed beyond this bit. If an overflow has occurred, then the double word will be set to MININT or MAXINT (shifted by the given bit location). Performing this instruction between a series of *MACCS* instructions and a *LEXTRACT* instruction will cause the extracted word to be either the correct answer or MAXINT/MININT if the result had overflowed positively or negatively.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
LSATS d,x,y
```

Operation:

```
if d:x > 2^{y+bpw}-1:
  d:x <- 2^{y+bpw} - 1
elif d:x < -2^{y+bpw}:
  d:x <- -2^{y+bpw}
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .
1 1 1 0 0	1 1 1 1 1	1	0	1 1 0 0

(*M and R*)

22.155 SETC: Set resource control bits

Sets the resource control bits. The control bits that can be set with SETC are the following:

CTRL_INUSE_OFF	0x0000	CTRL_RUN_CLRBUF	0x0017
CTRL_INUSE_ON	0x0008	CTRL_MS_MASTER	0x1007
CTRL_COND_NONE	0x0001	CTRL_MS_SLAVE	0x100f
CTRL_COND_FULL	0x0001	CTRL_BUF_NOBUFFERS	0x2007
CTRL_COND_AFTER	0x0009	CTRL_BUF_BUFFERS	0x200f
CTRL_COND_EQ	0x0011	CTRL_RDY_NOREADY	0x3007
CTRL_COND_NEQ	0x0019	CTRL_RDY_STROBED	0x300f
CTRL_COND_GREATER	0x0021	CTRL_RDY_HANDSHAKE	0x3017
CTRL_COND_LESS	0x0029	CTRL_SDELAY_NOSDELAY	0x4007
CTRL_IE_MODE_EVENT	0x0002	CTRL_SDELAY_SDELAY	0x400f
CTRL_IE_MODE_INTERRUPT	0x000a	CTRL_PORT_DATAPORT	0x5007
CTRL_DRIVE_DRIVE	0x0003	CTRL_PORT_CLOCKPORT	0x500f
CTRL_DRIVE_PULL_DOWN	0x000b	CTRL_PORT_READYPORT	0x5017
CTRL_DRIVE_PULL_UP	0x0013	CTRL_INV_NOINVERT	0x6007
CTRL_RUN_STOPR	0x0007	CTRL_INV_INVERT	0x600f
CTRL_RUN_STARTR	0x000f		

The precise effect depends on the resource type:

- Ports: See the chapter on Ports in the architecture manual for a description of the port modes.
- Timer: Only two of the modes, COND_AFTER and COND_NONE, can be used. When COND_AFTER is set, the next *IN* operation on this resource will block until the timer has reached the value set with *SETD*. Note that any value between the set time and the set time - `2^{bpw-1}` is accepted for the after condition.
- Clock source: Only the modes INUSE_ON and INUSE_OFF can be used - the resource must be switched on before it is used, and switch off when the program is finished with it.

The instruction has two operands:

op1
: r, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
SETC r,s
```

Operation:

```
control(r) <- s
```

Encoding:

l2r: Two register long

1 1 1 1 1	. . . . .	.	1	. .	. .
0 0 1 0 1	1 1 1 1 1	1	0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
`r` is not a valid resource, or the resource is not in use, or not a resource on which SETC can be used

ET_ILLEGAL_RESOURCE
`s` is not a valid mode, or not a mode that can be used on `r`.

22.156 SETCI: Set resource control bits immediate

Sets the resource control bits. The control bits that can be set with SETC are the following:

CTRL_INUSE_OFF	0x0000	CTRL_RUN_CLRBUF	0x0017
CTRL_INUSE_ON	0x0008	CTRL_MS_MASTER	0x1007
CTRL_COND_NONE	0x0001	CTRL_MS_SLAVE	0x100f
CTRL_COND_FULL	0x0001	CTRL_BUF_NOBUFFERS	0x2007
CTRL_COND_AFTER	0x0009	CTRL_BUF_BUFFERS	0x200f
CTRL_COND_EQ	0x0011	CTRL_RDY_NOREADY	0x3007
CTRL_COND_NEQ	0x0019	CTRL_RDY_STROBED	0x300f
CTRL_COND_GREATER	0x0021	CTRL_RDY_HANDSHAKE	0x3017
CTRL_COND_LESS	0x0029	CTRL_SDELAY_NOSDELAY	0x4007
CTRL_IE_MODE_EVENT	0x0002	CTRL_SDELAY_SDELAY	0x400f
CTRL_IE_MODE_INTERRUPT	0x000a	CTRL_PORT_DATAPORT	0x5007
CTRL_DRIVE_DRIVE	0x0003	CTRL_PORT_CLOCKPORT	0x500f
CTRL_DRIVE_PULL_DOWN	0x000b	CTRL_PORT_READYPORT	0x5017
CTRL_DRIVE_PULL_UP	0x0013	CTRL_INV_NOINVERT	0x6007
CTRL_RUN_STOPR	0x0007	CTRL_INV_INVERT	0x600f
CTRL_RUN_STARTR	0x000f		

The precise effect depends on the resource type:

- Ports: See the chapter on Ports in the architecture manual for a description of the port modes.
- Timer: Only two of the modes, COND_AFTER and COND_NONE, can be used. When COND_AFTER is set, the next *IN* operation on this resource will block until the timer has reached the value set with *SETD*. Note that any value between the set time and the set time - `2^{bpw-1}` is accepted for the after condition.
- Clock source: Only the modes INUSE_ON and INUSE_OFF can be used - the resource must be switched on before it is used, and switch off when the program is finished with it.

The instruction has two operands:

op1
: r, Operand register, one of `r0`… `r11`

op2
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
SETCI r,u_{16}
```

Operation:

```
control(r) <- u16
```

Encoding:

ru6: Register with 6-bit immediate

1 1 1 0 1	0	. . . .	. . . . . .

(*R*)

lru6: Register with 16-bit immediate

1 1 1 1 0	0	. . . .

1 1 1 0 1	0	. . . .	. . . . . .

(*M and R*)

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_RESOURCE_DEP
Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
`op1` is not a valid resource, or the resource is not in use, or not a resource on which SETC can be used

ET_ILLEGAL_RESOURCE
`op2` is not a valid mode, or not a mode that can be used on `op1`.

22.157 SETCLK: Set clock for a resource

Sets the clock for a resource. The precise meaning of this instruction depends on the resource.

The instruction has two operands:

op1
: r, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
SETCLK r,s
```

Operation:

```
clk(r) <- s
```

Encoding:

lr2r: Two register reversed long

1	1	1	1	1	.	.	.	.	.	.	1	.	.	.	.
0	0	0	0	1	1	1	1	1	1	1	0	1	1	0	0

(*M and R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not a port or clock source resource, or the resource is not in use.

ET_ILLEGAL_RESOURCE
: `s` is not a port or clock source resource.

ET_ILLEGAL_RESOURCE
: `r` is a running clock-block.

22.158 SETCP: Set constant pool

Sets the base address of the constant pool, held in `cp`. The value that is written into `cp` should be word-aligned, otherwise subsequent loads and stores relative to `cp` will raise an exception.

SETCP is used in conjunction with *LDWCP* and *LDAWCP*.

The instruction has one operand:

op1
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
SETCP s
```

Operation:

```
cp <- s
```

Encoding:

1r: Register

0 0 1 1 0	1 1 1 1 1	1	1	. . . .

(*M*)

22.159 SETD: Set event data

Sets the contents of the data/dest/divide register of a resource. Its data register is read using *GETD*. The way that a resource depends on the data register is resource dependent:

- Port: specifies the value for the input condition (see *SETC*)
- Timer: specifies the value to wait for (see *SETC*)
- Channel end: specifies the destination channel for *OUT* operations. The value written should be a channel identifier, constructed as specified for *GETR*.
- Clock source specifies the value to divide the clock input by.

The instruction has two operands:

op1
: r, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
SETD r,s
```

Operation:

```
data(r) <- s
```

Encoding:

r2r: Two register reversed

0 0 0 1 0	. . . . .	.	1	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not a channel, timer, port or clock resource, or the resource is not in use.

ET_ILLEGAL_RESOURCE
: `r` is a running clock-block.

ET_ILLEGAL_RESOURCE
: `r` is a channel-end, and `s` is not a channel-end or a configuration resource.

22.160 SETDP: Set the data pointer

Sets the base address of the global data area, held in **dp**. The value that is written into **dp** should be word-aligned, otherwise subsequent loads and stores relative to **dp** will raise an exception.

SETDP is used in conjunction with *LDWDP*, *STWDP*, and *LDAWDP*

The instruction has one operand:

op1
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
SETDP s
```

Operation:

```
dp <- s
```

Encoding:

1r: Register

0 0 1 1 0	1 1 1 1 1	1	0	. . . .

(*M*)

22.161 SETEV: Set environment vector

Sets the environment vector related to a resource. When a resource issues an event to a thread, any address stored in the environment vector will overwrite `ed`. If uninitialised, `ed` will be set to the resource identifier. SETEV can be used to pass an address specific to a resource to the event handler. SETEV can be used to share a single handler between multiple resources. Note that SETEV is intended to pass address information, as such it does not necessarily hold `bpw` bits.

SETEV is used in conjunction with *SETV*, and any of the *WAITEU* instructions.

The instruction has one operand:

op1
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
SETEV r
```

Operation:

```
ev(r) <- r11
```

Encoding:

1r: Register

0 0 1 1 1	1 1 1 1 1	1	1	. . . .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not a port, timer or channel resource, or the resource is not in use.

22.162 SETKEP: Set the kernel entry point

Sets the kernel entry point. The kernel entry point should be aligned on a 128-byte boundary.

The instruction has no operands.

Mnemonic and operands:

```
SETKEP
```

Operation:

```
kep <- r11
```

Encoding:

0r: No operands

0 0 0 0 0	1 1 1 1 1	1	1	1 1 1 1

(*M*)

22.163 SETN: Set network

Sets the logical network over which a channel should communicate.

The instruction has two operands:

op1
: r, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
SETN r,s
```

Operation:

```
net(r) <- s
```

Encoding:

lr2r: Two register reversed long

1 1 1 1 1	. . . . .	.	0	. .	. .
0 0 1 1 0	1 1 1 1 1	1	0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not a channel end or not in use.

22.164 SETPS: Set processor state

Sets a processor internal register. Only used when configuring the core.

The instruction has two operands:

op1
: r, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
SETPS r,s
```

Operation:

```
ps[r] <- s
```

Encoding:

lr2r: Two register reversed long

1 1 1 1 1	. . . . .	.	0	. .	. .
0 0 0 1 1	1 1 1 1 1	1	0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_ILLEGAL_PS
: `s` is not referring to a legal processor state register

ET_ILLEGAL_PS
: `s` is not referring to a read-only processor state register

ET_ILLEGAL_PS
: `s` is referring to RAMBASE and `r` is set to the ROM address

22.165 SETPSC: Set the port shift count

Sets the port shift count for input and output operations.

OUTPW and *INPW* can be used instead of a combination of SETPSC and *OUT*/*IN*.

The instruction has two operands:

op1
: r, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
SETPSC r,s
```

Operation:

```
shiftcount(r) <- s
```

Encoding:

r2r: Two register reversed

1 1 0 0 0	. . . . .	.	0	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not pointing to a port resource, or the resoruce is not in use.

ET_ILLEGAL_RESOURCE
: `s` is not a valid shift count for the transfer width of the port, or the port is not in BUFFERED mode.

22.166 SETPT: Set the port time

Specifies the time when the next port input or output will be performed. The time is specified in terms of the number of edges of the clock associated with this port. The port timer stores a 16-bit value hence the largest delay is 65535 edges of the port-clock.

The instruction has two operands:

op1
: r, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
SETPT r,s
```

Operation:

```
porttimer(r) <- s
```

Encoding:

r2r: Two register reversed

0 0 1 1 1	. . . . .	.	1	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not pointing to a port resource, or the resource is not in use.

22.167 SETRDY: Set ready input for a port

Sets ready input pin to be used by a port for strobing or handshaking.

If `r` is a clock block, then `s` should be the 1-bit port to be used as ready input. `r` should be associated with a dataport using *SETCLK*.

Otherwise, if `r` is a port, then this port should be in mode READY_OUT, and `s` is the data port from which the ready out will be generated.

The instruction has two operands:

op1
: r, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
SETRDY r,s
```

Operation:

```
rdy(r) <- s
```

Encoding:

lr2r: Two register reversed long

1 1 1 1 1	. . . . .	.	0	. .	. .
0 0 1 0 1	1 1 1 1 1	1	0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not pointing to a port or clock resource, or the resource is not in use.

ET_ILLEGAL_RESOURCE
: `s` is not pointing to a port resource, or the port is not a 1-bit port.

22.168 SETSP: Set the stack pointer

Sets the end address of the stack, held in `sp`. The value that is written into `sp` should be word-aligned, otherwise subsequent loads and stores relative to `sp` will raise an exception.

SETSP is used in conjunction with *ENTSP*, *RETSP*, *LDWSP* and *STWSP*.

The instruction has one operand:

op1
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
SETSP s
```

Operation:

```
sp <- s
```

Encoding:

1r: Register

0 0 1 0 1	1 1 1 1 1	1	1	. . . .

(*M*)

22.169 SETSR: Set bits in SR

Set bits in the thread's Status Register. The mask supplied specifies which bits should be set. Note that setting the EEBLE bit may cause an event to be issued, causing subsequent instructions to not be executed (since events do not save the program counter). Setting IEBLE may cause an interrupt to be issued. The bits are defined as follows:

Bit	Name	Number
0	EEBLE	When 1 events are enabled for the thread.
1	IEBLE	When 1 interrupts are enabled for the thread.
2	INENB	1 when in an event enabling sequence.
3	ININT	1 when in an interrupt handler.
4	INK	1 when in kernel mode.
6	WAITING	When 1 the thread is paused waiting for events.
7	FAST	When 1 the thread will continually issue.

SETSR can only be used to set the EEBLE, IEBLE and INENB bits.

CLRSR is used to clear bits in the status register.

The instruction has one operand:

op1

u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
SETSR u_{16}
```

Operation:

```
sr <- sr | u16
```

Encoding:

u6: 6-bit immediate

0 1 1 1 1	0	1 1 0 1	. . . . . .

(*R*)

lu6: 16-bit immediate

1 1 1 1 0	0	. . . .
0 1 1 1 1	0	1 1 0 1 . . . . . .

(*M and R*)

The latter is prefixed for long immediates.

22.170 SETTW: Set transfer width for a port

Sets the number of bits that is transferred on an IN or OUT operation on a port that is buffered. The buffering will shift the data.

The instruction has two operands:

op1
  r, Operand register, one of `r0`... `r11`

op2
  s, Operand register, one of `r0`... `r11`

Mnemonic and operands:

```
SETTW r,s
```

Operation:

```
transferwidth(r) <- s
```

Encoding:

lr2r: Two register reversed long

1 1 1 1 1	. . . . .	.	1	. .	. .
0 0 1 0 0	1 1 1 1 1	1	0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_ILLEGAL_RESOURCE
  `r` is not pointing to a port resource, or the port is not in use.

ET_RESOURCE_DEP
  Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
  `s` is not legal width for the port, or the port is not in BUFFERS mode.

22.171 SETV: Set event vector

Sets the vector related to a resource. When a resource issues an event to a thread, this vector is used to determine which instruction to issue. The vector is typically set up once when all event handlers are installed. Note that if an illegal vector is supplied, this will not raise an exception until an actual event is handled.

SETV is used in conjunction with *SETEV*, and any of the *WAITEU* instructions.

The instruction has one operand:

op1
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
SETV r
```

Operation:

```
v(r) <- r11
```

Encoding:

1r: Register

0 1 0 0 0	1 1 1 1 1	1	1	. . . .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not pointing to a port, timer or channel resoruce, or the resource is not in use.

22.172 SEXT: Sign extend an n-bit field

Sign extends an n-bit field stored in a register. The first operand is both a source and destination operand. The second operand contains the bit position. All bits at a position higher or equal are set to the value of the bit one position lower. In effect, the lower `n` bits are interpreted as a signed integer, and produced in the destination register.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
SEXT d,s
```

Operation:

```
d <- d,                              if s <= 0 || s >= bpw
     d[s-1]:...:d[s-1]:d[s-1...0], if s > 0 && s < bpw,
```

Encoding:

2r: Two register

0 0 1 1 0	. . . . .	.	0	. .	. .

(*M or R*)

22.173 SEXTI: Sign extend an n-bit field immediate

Sign extends an n-bit field stored in a register. The first operand is both a source and destination operand. The second operand contains the bit position. All bits at a position higher or equal are set to the value of the bit one position lower. In effect, the lower `n` bits are interpreted as a signed integer, and produced in the destination register.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: bitp, A bit position; one of `bpw`, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32

Mnemonic and operands:

```
SEXTI d,bitp
```

Operation:

```
d <- d,                            if bitp <= 0 || bitp >= bpw
     d[bitp-1]:...:d[bitp-1]:d[bitp-1...0], if bitp > 0 && bitp < bpw,
```

Encoding:

rus: Register with immediate

0 0 1 1 0	. . . . .	.	1	. .	. .

(*M or R*)

22.174 SHL: Shift left

Shifts a word left by y bits, filling the least significant y bits with zeros. Shift left multiplies signed and unsigned integers by 2^{y}.

SHL with perform an arithmetic shift right with a negative value.

The instruction has three operands:

op1
: d, Operand register, one of r0… r11

op2
: x, Operand register, one of r0… r11

op3
: y, Operand register, one of r0… r11

Mnemonic and operands:

```
SHL d,x,y
```

Operation:

```
d <- x[bpw-y...0]:0:...:0,           if 0 < y < bpw
     x[bpw]:x[bpw]:...:x[bpw-1:-y],  if y < 0
     0,                              if y >= bpw
```

Encoding:

3r: Three register

0 0 1 0 0	. . . . .	. .	. .	. .

(*M or R*)

22.175 SHLI: Shift left immediate

Shifts a word left by **bitp** bits, filling the least significant **bitp** bits with zeros. Shift left multiplies signed and unsigned integers by 2^{bitp}.

SHL with perform an arithmetic shift right with a negative value.

The instruction has three operands:

op1
d, Operand register, one of r0... r11

op2
x, Operand register, one of r0... r11

op3
bitp, A bit position; one of bpw, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32

Mnemonic and operands:

```
SHLI d,x,bitp
```

Operation:

```
d <- x[bpw-bitp...0]:0:...:0,            if 0 < bitp < bpw
     x[bpw]:x[bpw]:...:x[bpw-1:-bitp],   if bitp < 0
     0,                               if bitp >= bpw
```

Encoding:

2rus: Two register with immediate

| 1 0 1 0 0 | . . . . . | . . | . . | . . | (*M or R*)

22.176 SHR: Shift right

Shifts a word right by `y` positions, filling the most significant `y` bits with zeros. This implements an unsigned divide by `2^{y}`.

SHR will perform an shift left with a negative value.

For signed shifts, use *ASHR*.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
SHR d,x,y
```

Operation:

```
d <- 0:...:0:x[bpw-1...y], if 0 < y < bpw,
     x[bpw+y-1...0]:0:...:0, if 0 < y < 0
     0,                     if y >= bpw
```

Encoding:

3r: Three register

0 0 1 0 1	. . . . .	. .	. .	. .

(*M or R*)

22.177 SHRI: Shift right immediate

Shifts a word right by **bitp** positions, filling the most significant **bitp** bits with zeros. This implements an unsigned divide by `2^{bitp}`.

SHR will perform an shift left with a negative value.

For signed shifts, use *ASHR*.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: bitp, A bit position; one of `bpw`, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32

Mnemonic and operands:

```
SHRI d,x,bitp
```

Operation:

```
d <- 0:...:0:x[bpw-1...bitp], if 0 < bitp < bpw,
     x[bpw+bitp-1...0]:0:...:0, if 0 < bitp < 0
     0,                        if bitp >= bpw
```

Encoding:

2rus: Two register with immediate

1 0 1 0 1	. . . . .	. .	. .	. .

(*M or R*)

22.178 SSYNC: Slave synchronise

Synchronises this thread with all threads associated with a synchroniser. SSYNC is used together with *MSYNC* to implement a barrier, or together with *MJOIN* in order to terminate a group of processes. SSYNC uses the synchroniser that was used to create this process in order to establish which other processes to synchronise with.

SSYNC clears the EEBLE bit, disabling any events from being issued; this commits the thread to synchronising. If the ININT bit is set, then SSYNC will not block; SSYNC should not be used inside an interrupt handler.

The instruction has no operands.

Mnemonic and operands:

```
SSYNC
```

Operation:

```
sr[eeble] <- 0
if (slaves(syn(tid)) setminus spaused = {tid}) && msyn(syn(tid)):
  if mjoin(syn(tid)):
    for thread in slaves(syn(tid)):
      inuse_{thread} <- 0
    mjoin(syn(tid)) <- 0
  else:
    spaused <- spaused - slaves(syn(tid))
  mpaused <- mpaused - {mstr(syn(tid))}
  msyn(syn(tid)) <- 0
else:
  spaused <- spaused UNION {tid}
```

Encoding:

0r: No operands

0 0 0 0 0	1 1 1 1 1	1	0	1 1 1 0

(*R*)

22.179 ST8: 8-bit store

Stores eight bits of a register into memory. The least significant 8 bits of the register are stored into the address computed using a base address (**b**) and index (**i**).

The instruction has three operands:

op1
: s, Operand register, one of `r0`… `r11`

op2
: b, Operand register, one of `r0`… `r11`

op3
: i, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
ST8 s,b,i
```

Operation:

```
mem[ea-bytenum][bitnum+7...bitnum] <- s
     where ea = b + i
           bytenum = ea % Bpw
           bitnum = 8 * bytenum
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .
1 0 0 0 1	1 1 1 1 1	1 0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_LOAD_STORE
: The indexed address does not point to a valid memory location.

22.180 ST16: 16-bit store

Stores 16 bits of a register into memory. The least significant 16 bits of the register are stored into the address computed using a base address (**b**) and index (**i**). The base address should be word-aligned, the index is multiplied by 2.

The instruction has three operands:

op1
: s, Operand register, one of `r0`... `r11`

op2
: b, Operand register, one of `r0`... `r11`

op3
: i, Operand register, one of `r0`... `r11`

Mnemonic and operands:

```
ST16 s,b,i
```

Operation:

```
mem[ea - bytenum][bitnum+15...bitnum] <- s[15...0]
    where ea = b + i * 2
          bytenum = ea % Bpw
          bitnum = 16 * (bytenum // 2)
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .
1 0 0 0 0	1 1 1 1 1	1 0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_LOAD_STORE
: b is not 16-bit aligned (unaligned load), or does not point to a valid memory location.

22.181 STD: Store double word

Stores two words in memory, at a location specified by a base address and an index. The index is multiplied by the size of a double word, the base address must be double-word aligned.

The immediate version, *STDI*, implements a store into a structured data type, the version with registers only, *STD*, implements a store into an array.

The instruction has four operands:

op1
: d, Operand register, one of `r0`… `r11`

op4
: e, Operand register, one of `r0`… `r11`

op2
: b, Operand register, one of `r0`… `r11`

op3
: i, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
STD d,e,b,i
```

Operation:

```
mem[b + i * Bpw * 2] <- d
mem[b + i * Bpw * 2 + Bpw] <- e
```

Encoding:

l4r: Four register long

1 1 1 1 1	. . . . .	. .	. .	. .
0 0 0 1 0	**1 1 1 1 1**	**1**	**0**	**. . . .**

(*M and R*)

Conditions that raise an exception:

ET_LOAD_STORE
: b is not double word aligned, or the indexed address does not point to a valid memory location.

22.182 STDI: Store double word immediate

Stores two words in memory, at a location specified by a base address and an index. The index is multiplied by the size of a double word, the base address must be double-word aligned.

The immediate version, *STDI*, implements a store into a structured data type, the version with registers only, *STD*, implements a store into an array.

The instruction has four operands:

op1
: d, Operand register, one of `r0`… `r11`

op4
: e, Operand register, one of `r0`… `r11`

op2
: b, Operand register, one of `r0`… `r11`

op3
: i, An integer in the range 0…11

Mnemonic and operands:

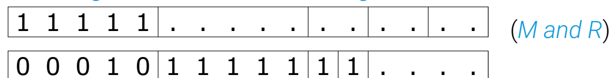
```
STDI d,e,b,i
```

Operation:

```
mem[b + i * Bpw * 2] <- d
mem[b + i * Bpw * 2 + Bpw] <- e
```

Encoding:

l3rus: Three register with immediate long



1 1 1 1 1	. . . . .	. .	. .	. .
0 0 0 1 0	1 1 1 1 1	1	1	. . . .

(*M and R*)

Conditions that raise an exception:

ET_LOAD_STORE
: b is not double word aligned, or the indexed address does not point to a valid memory location.

22.183 STDSP: Store double word on stack

Stores two words on the stack, using a constant offset from the stack pointer. The offset is specified in double words.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: e, Operand register, one of `r0`… `r11`

op3
: us, An integer in the range 0…11

Mnemonic and operands:

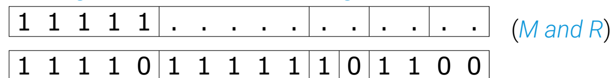
```
STDSP d,e,u_s
```

Operation:

```
mem[sp + us * Bpw * 2] <- d
mem[sp + us * Bpw * 2 + Bpw] <- e
```

Encoding:

l2rus: Two register with immediate long



1 1 1 1 1	. . . . .	. .	. .	. .
1 1 1 1 0	1 1 1 1 1	1	0	1 1 0 0

(*M and R*)

Conditions that raise an exception:

ET_LOAD_STORE
: `sp` is not double-word aligned, or the indexed address does not point to a valid memory location.

22.184 STET: Store ET on the stack

Stores the value of ET on the stack at offset 4.

The value can be restored using *LDET*. Together with *STSPC*, *STSSR*, and *STSED* all or part of the state copied during an interrupt can be placed on the stack.

The instruction has no operands.

Mnemonic and operands:

STET

Operation:

```
mem[sp + 4 * Bpw] <- et
```

Encoding:

0r: No operands

0 0 0 0 1	1 1 1 1 1	1	1	1 1 0 1

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
The indexed address does not point to a valid memory location.

22.185 STSED: Store SED on the stack

Stores the value of SED on the stack at offset 3.

The value can be restored using *LDSED*. Together with *STSPC*, *STSSR*, and *STET* all or part of the state copied during an interrupt can be placed on the stack.

The instruction has no operands.

Mnemonic and operands:

```
STSED
```

Operation:

```
mem[sp + 3 * Bpw] <- sed
```

Encoding:

0r: No operands

0 0 0 0 1	1 1 1 1 1	1	1	1 1 0 0

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
The indexed address does not point to a valid memory location.

22.186 STSPC: Store SPC on the stack

Stores the value of SPC on the stack at offset 1.

The value can be restored using *LDSPC*. Together with *STET*, *STSSR*, and *STSED* all or part of the state copied during an interrupt can be placed on the stack.

The instruction has no operands.

Mnemonic and operands:

```
STSPC
```

Operation:

```
mem[sp + 1 * Bpw] <- spc
```

Encoding:

0r: No operands

0 0 0 0 1	1 1 1 1 1	1	0	1 1 0 1

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
The indexed address does not point to a valid memory location.

22.187 STSSR: Store the SSR to the stack

Stores the value of SSR on the stack at offset 2.

The value can be restored using *LDSSR*. Together with *STET*, *STSPC*, and *STSED* all or part of the state copied during an interrupt can be placed on the stack.

The instruction has no operands.

Mnemonic and operands:

STSSR

Operation:

```
mem[sp + 2 * Bpw] <- ssr
```

Encoding:

0r: No operands

0 0 0 0 1	1 1 1 1 1	1	0	1 1 1 1

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
The indexed address does not point to a valid memory location.

22.188 STW: Store word

Stores a word in memory, at a location specified by a base address and an index. The index is multiplied by the size of a word, the base address must be word aligned.

The immediate version, *STWI*, implements a store into a structured data type, the version with registers only, *STW*, implements a store into an array.

The instruction has three operands:

op1
: s, Operand register, one of `r0`… `r11`

op2
: b, Operand register, one of `r0`… `r11`

op3
: i, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
STW s,b,i
```

Operation:

```
mem[b + i * Bpw] <- s
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .
0 0 0 0 0	1 1 1 1 1	1 0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_LOAD_STORE
: b is not word aligned, or the indexed address does not point to a valid memory location.

22.189 STWI: Store word immediate

Stores a word in memory, at a location specified by a base address and an index. The index is multiplied by the size of a word, the base address must be word aligned.

The immediate version, *STWI*, implements a store into a structured data type, the version with registers only, *STW*, implements a store into an array.

The instruction has three operands:

op1
: s, Operand register, one of `r0`… `r11`

op2
: b, Operand register, one of `r0`… `r11`

op3
: i, An integer in the range 0…11

Mnemonic and operands:

```
STWI s,b,i
```

Operation:

```
mem[b + i * Bpw] <- s
```

Encoding:

2rus: Two register with immediate

| 0 0 0 0 0 | . . . . . | . . | . . | . . | (*M*)

Conditions that raise an exception:

ET_LOAD_STORE
: b is not word aligned, or the indexed address does not point to a valid memory location.

22.190 STWDP: Store word in data pool

Stores a word in the data area, using a constant offset from the data pointer. The offset is specified in words. STWDP can be used to write to global variables.

The instruction has two operands:

op1
: S, Any of `r0`… `r11`, `cp`, `dp`, `sp`, `lr`

op2
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
STWDP S,u_{16}
```

Operation:

```
mem[dp + u16 * Bpw] <- S
```

Encoding:

ru6: Register with 6-bit immediate

0 1 0 1 0	0	. . . .	. . . . . .	(*M*)

lru6: Register with 16-bit immediate

1 1 1 1 0	0	. . . .	(*M and R*)
0 1 0 1 0	0	. . . .	. . . . . .

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_LOAD_STORE
: `dp` is not word aligned, or the indexed address does not point to a valid memory location.

22.191 STWSP: Store word on stack

Stores a word on the stack, using a constant offset from the stack pointer. The offset is specified in words. STWSP is used to write to stack variables.

The instruction has two operands:

op1
: S, Any of `r0`… `r11`, `cp`, `dp`, `sp`, `lr`

op2
: u16, A 16-bit immediate in the range 0…65535. If `u16 < 64`, the instruction requires no prefix

Mnemonic and operands:

```
STWSP S,u_{16}
```

Operation:

```
mem[sp + u16 * Bpw] <- S
```

Encoding:

ru6: Register with 6-bit immediate

0 1 0 1 0	1	. . . .	. . . . . .	(*M*)

lru6: Register with 16-bit immediate

1 1 1 1 0	0	. . . .	(*M and R*)
0 1 0 1 0	1	. . . .	. . . . . .

The latter is prefixed for long immediates.

Conditions that raise an exception:

ET_LOAD_STORE
: `sp` is not word aligned, or the indexed address does not point to a valid memory location.

22.192 SUB: Integer unsigned subtraction

Computes the difference between two words. No check on overflow is performed, and the result is produced modulo `2^{bpw}`.

If a borrow is required, then the *LSUB* instruction should be used. *LSU* and *LSS* should be used to compare signed and unsigned integers.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
SUB d,x,y
```

Operation:

```
d <- (2^bpw + x - y) % 2^bpw
```

Encoding:

3r: Three register

0 0 0 1 1	. . . . .	. .	. .	. .	(*M or R*)

22.193 SUBI: Integer unsigned subtraction immediate

Computes the difference between two words. No check on overflow is performed, and the result is produced modulo `2^{bpw}`.

If a borrow is required, then the *LSUB* instruction should be used. *LSU* and *LSS* should be used to compare signed and unsigned integers.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: us, An integer in the range 0…11

Mnemonic and operands:

```
SUBI d,x,u_s
```

Operation:

```
d <- (2^bpw + x - us) % 2^bpw
```

Encoding:

2rus: Two register with immediate

1 0 0 1 1	. . . . .	. .	. .	. .	(*M or R*)

22.194 SYNCR: Synchronise a resource

Synchronise with a port to ensure all data has been output. This instruction completes once all data has been shifted out of the port, and the last port width of data has been held for one clock period.

The instruction has one operand:

op1
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
SYNCR r
```

Operation:

```
syncr(r)
```

Encoding:

1r: Register

1 0 0 0 0	1 1 1 1 1	1	1	. . . .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not a port resource, or the resource is not in use.

22.195 TESTCT: Test for control token

Test whether the next token on a channel (r) is a control token. If the channel contains a control token, then 1 (true) will be produced in the destination register, otherwise 0 (false) will be produced.

This instruction pauses if the channel does not have a token available to be read.

In contrast to *CHKCT* this test does not trap, and does not discard the control token. TESTCT can be used to implement complex protocols over channels.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
TESTCT d,r
```

Operation:

```
d <- 1,  if hasctoken(r)
     0,  if !hasctoken(r)
```

Encoding:

2r: Two register

1 0 1 1 1	. . . . .	.	1	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not pointing to a channel resource, or the resource is not in use.

22.196 TESTLCL: Test local

Tests if a channel end is connected to a local channel end or to a remote channel end. It produces 1 (true) in the destination register if the channel end is local, and 0 (false) if the channel end is remote. The instruction will raise an exception if the resource supplied is not a channel end or an unconnected channel end.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
TESTLCL d,r
```

Operation:

```
d <- 1,  if d(r)[bpw-1..16] = r[bpw-1..16]
     0,  if d(r)[bpw-1..16] != r[bpw-1..16]
```

Encoding:

l2r: Two register long

1 1 1 1 1	. . . . .	.	0	. .	. .
0 0 1 0 0	1 1 1 1 1	1	0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not pointing to a channel resource, or the resource is not in use.

ET_ILLEGAL_RESOURCE
: `r` is a channel end, and the destination has not been set.

22.197 TESTWCT: Test for position of control token

Test whether the next word contains a control token, and produces the position (1-4) of the first control token in the word, or 0 if it contains no control tokens.

This instruction pauses if the channel has not received enough tokens to determine what value to return. So if less than four tokens have been received, but one of them is a control token, the instruction will not pause.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: r, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
TESTWCT d,r
```

Operation:

```
d <- 0, if !hasctoken(r)
     1,  if first token is ctoken
     2,  if second token is ctoken
     3,  if third token is ctoken
     4,  if fourth token is ctoken
```

Encoding:

2r: Two register

1 1 0 0 0	. . . . .	.	1	. .	. .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `r` is not pointing to a channel resource, or the resource is not in use.

22.198 TINITCP: Initialise a thread's CP

Sets the constant pool pointer for a specific thread. This operation may be used after a thread has been allocated (using *GETST* or *GETR*), but prior to the thread starting its execution.

The instruction has two operands:

op1
: s, Operand register, one of `r0`… `r11`

op2
: t, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
TINITCP s,t
```

Operation:

```
cp(t) <- s
```

Encoding:

l2r: Two register long

1 1 1 1 1	. . . . .	.	0	. .	. .
0 0 0 0 0	1 1 1 1 1	1	0	1 1	0 1

(*M and R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `t` is not pointing to a thread resource, or the thread is not in use, or the thread is not SSYNC.

22.199 TINITDP: Initialise a thread's DP

Sets the data pointer for a specific thread. This operation may be used after a thread has been allocated (using *GETST* or *GETR*), but prior to the thread starting its execution.

The instruction has two operands:

op1
: s, Operand register, one of `r0`... `r11`

op2
: t, Operand register, one of `r0`... `r11`

Mnemonic and operands:

```
TINITDP s,t
```

Operation:

```
dp(t) <- s
```

Encoding:

l2r: Two register long

1 1 1 1 1	. . . . .	.	0	. .	. .
0 0 0 0 1	1 1 1 1 1	1	0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `t` is not pointing to a thread resource, or the thread is not in use, or the thread is not SSYNC.

22.200 TINITLR: Initialise a thread's LR

Sets the link register for a specific thread. This operation may be used after a thread has been allocated (using *GETST* or *GETR*), but prior to the thread starting its execution.

The instruction has two operands:

op1
: s, Operand register, one of `r0`… `r11`

op2
: t, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
TINITLR s,t
```

Operation:

```
lr(t) <- s
```

Encoding:

l2r: Two register long

1 1 1 1 1	. . . . .	.	0	. .	. .
0 0 0 1 0	1 1 1 1 1	1	0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `t` is not pointing to a thread resource, or the thread is not in use, or the thread is not SSYNC.

22.201 TINITPC: Initialise a thread's PC

Sets the program counter for a specific thread. This operation may be used after a thread has been allocated (using *GETST* or *GETR*), but prior to the thread starting its execution.

The instruction has two operands:

op1
: s, Operand register, one of `r0`… `r11`

op2
: t, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
TINITPC s,t
```

Operation:

```
pc(t) <- s
```

Encoding:

l2r: Two register long

1 1 1 1 1	. . . . .	.	1	. .	. .
0 0 0 0 0	1 1 1 1 1	1	0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `t` is not pointing to a thread resource, or the thread is not in use, or the thread is not SSYNC.

22.202 TINITSP: Initialise a thread's SP

Sets the stack pointer for a specific thread. This operation may be used after a thread has been allocated (using *GETST* or *GETR*), but prior to the thread starting its execution.

The instruction has two operands:

op1
: s, Operand register, one of `r0`… `r11`

op2
: t, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
TINITSP s,t
```

Operation:

```
sp(t) <- s
```

Encoding:

l2r: Two register long

1 1 1 1 1	. . . . .	.	0	. .	. .
0 0 0 0 0	1 1 1 1 1	1	0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `t` is not pointing to a thread resource, or the thread is not in use, or the thread is not SSYNC.

22.203 TSETMR: Set the master's register

Writes data to a register of the master thread. This instruction should be used with care, and only when the other thread is known to be not using that register. Typically used to transfer results from a slave thread back to the master prior to a *MJOIN*.

TSETMR uses the synchroniser that was used to create this process in order to establish which thread's register to write to.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
TSETMR d,s
```

Operation:

```
register(mtid, d) <- s
```

Encoding:

l2r: Two register long

1 1 1 1 1	. . . . .	.	0	. .	. .
0 0 0 0 1	1 1 1 1 1	1	0	1 1	0 1

(*M and R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: Master thread is not in use.

22.204 TSETR: Set register in thread

Writes data to a register of another thread. This instruction should be used with care, and only when the other thread is known to be not using that register.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

op3
: t, Operand register, one of `r0`… `r11`

Mnemonic and operands:

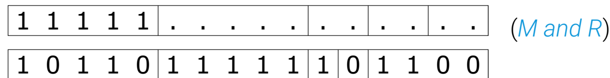
```
TSETR d,s,t
```

Operation:

```
register(t, d) <- s
```

Encoding:

l3r: Three register long



(*M and R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
: Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
: `t` is not pointing to a thread resource, or the thread is not in use.

22.205 TSTART: Start thread

Starts an unsynchronised thread. An unsynchronised thread runs independently from the starting thread.

The unsynchronised thread must have been allocated with *GETR*, and the program counter should have been initialised with *TINITPC*.

The instruction has one operand:

op1
t, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
TSTART t
```

Operation:

```
spaused <- spaused - {t}
waiting(t) <- 0
```

Encoding:

1r: Register

0 0 0 1 1	1 1 1 1 1	1	0	. . . .

(*R*)

Conditions that raise an exception:

ET_RESOURCE_DEP
Resource illegally shared between threads

ET_ILLEGAL_RESOURCE
`t` is not pointing to a thread, or the thread is not in use, or the thread is not SSYNC.

ET_ILLEGAL_PC
Thread `t` does not have a legal program counter.

22.206 UNZIP: Unzips a pair of registers

Unzips a pair of registers in bit, bit-pairs, nibbles, bytes or byte-pairs. The granularity of zipping is determined by `2^{s}`. The pair of registers is split in chunks of `2^{s}` bits. The most significant chunk and every other chunk after that are concatenated and written back to **d**. The other chunks in between are written back to **e**.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: e, Operand register, one of `r0`… `r11`

op3
: s, An integer in the range 0…11

Mnemonic and operands:

```
UNZIP d,e,s
```

Operation:

```
w = 2^s
z = d:e
d <- z[2 * bpw-1..2 * bpw-w]:
     z[2 * bpw-2w-1..2 * bpw-3w]:...:
     z[2w-1..w]
e <- z[2 * bpw-w-1..2 * bpw-2w]:
     z[2 * bpw-3w-1..2 * bpw-4w]:...:
     z[w-1..0]
```

Encoding:

l2rus: Two register with immediate long

1 1 1 1 1	. . . . .	. .	. .	. .	(*M and R*)
1 0 0 1 1	1 1 1 1 1	1 0	1 1	0 1	

22.207 VCLRDR: Clear Vectors D and R

Sets the contents of vD and vR in the vector unit to all zeroes.

The instruction has no operands.

Mnemonic and operands:

```
VCLRDR
```

Operation:

```
vD <- 0
vR <- 0
```

Encoding:

0r: No operands

0 0 1 0 0	1 1 1 1 1	1	0	1 1 1 1

(*M*)

22.208 VADDDR: Vector add double reduce

Takes a double precision result stored in two 16-bit pairs across vD and vR, and sums them into a single value stored in the lowest part of vD and vR. VLSAT can be used to reduce the values in vD and vR and saturate.

The instruction has no operands.

Mnemonic and operands:

```
VADDDR
```

Operation:

```
tmp = sum(vD[k* 16+15..k* 15] : vD[k* 16+15..k* 16]) where k in range(bpv // bpe)
vR[0] &<- & tmp[15:0]
vD[0] &<- & tmp[31:16]
```

Encoding:

0r: No operands

0 0 1 1 1	1 1 1 1 1	1	0	1 1 0 0

(*M*)

22.209 VCMCI: Vector Complex Multiply Conjugate Imaginary

This operation performs half of a complex multiply conjugate operation in the vector unit, calculating all the imaginary parts of the answer. Together with VCMCR a complete complex vector multiply can be performed on a vector.

The operation assumes that, in memory, the vector is stored as pairs of (real, imaginary) values, where the imaginary part is stored in the higher memory addresses. After a VLD instruction, this means that the real elements are stored in bytes 0..3, 8..11, etc, of each vector and the associated imaginary elements in bytes 4..7, 12..15, etc.

Bit 30 of each number is assumed to have a magnitude of 1.0, so the final result is shifted down by 30 bits. This enables two numbers in the unit square to be multiplied without overflow.

Saturation and rounding are applied to the result.

Rounding is applied first, by adding a value `2^{2-bpe}` to the result before removing bits `bpe-3..0`. Saturation is applied in order to catch numbers that are not in the unit square, and to catch the corner case of `(1+j)^2`, which will be rounded down by a single lsb.

The instruction has no operands.

Mnemonic and operands:

```
VCMCI
```

Operation:

```
for k in range(bpv/(bpe * 2)):
  t_k  =  bpe * 2 * k
  r_k  =  rnd(vD[t_k+2* bpe - 1 .. t_k+bpe] * vC[t_k+bpe - 1 .. t_k]) -
          rnd(vD[t_k+bpe - 1 .. t_k] * vC[t_k+2* bpe - 1 .. t_k+bpe])
  T_k = t_k + bpe
  vR[T_k+bpe - 1 .. T_k] <- sat(r_k[2* bpe-3 .. bpe-2] + r_k[bpe-3])
```

Encoding:

0r: No operands

0 0 0 1 1	1 1 1 1 1	1	1	1 1 0 1

(*M*)

Conditions that raise an exception:

ET_ARITHMETIC
The type was not set to VSETCTRL_TYPE_INT32

22.210 VCMCR: Vector Complex Multiply Conjugate Real

This operation performs half of a complex multiply conjugate operation in the vector unit, calculating all the real parts of the answer. Together with VCMCI a complete complex vector multiply can be performed on a vector.

The operation assumes that, in memory, the vector is stored as pairs of (real, imaginary) values, where the imaginary part is stored in the higher memory addresses. After a VLD instruction, this means that the real elements are stored in bytes 0..3, 8..11, etc, of each vector and the associated imaginary elements in bytes 4..7, 12..15, etc.

Bit 30 of each number is assumed to have a magnitude of 1.0, so the final result is shifted down by 30 bits. This enables two numbers in the unit square to be multiplied without overflow.

Saturation and rounding are applied to the result.

Rounding is applied first, by adding a value `2^{2-bpe}` to the result before removing bits `bpe-3..0`. Saturation is applied in order to catch numbers that are not in the unit square, and to catch the corner case of `(1+j)^2`, which will be rounded down by a single lsb.

The instruction has no operands.

Mnemonic and operands:

```
VCMCR
```

Operation:

```
for k in range(bpv/(bpe * 2)):
  t_k  =  bpe * 2 * k
  r_k  =  rnd(vD[t_k+bpe - 1 .. t_k] * vC[t_k+bpe - 1 .. t_k]) +
          rnd(vD[t_k+2* bpe - 1 .. t_k+bpe] * vC[t_k+2* bpe - 1 .. t_k+bpe])
  vR[t_k+bpe - 1 .. t_k] <- sat(r_k[2* bpe-3 .. bpe-2] + r_k[bpe-3])
```

Encoding:

0r: No operands

0 0 0 1 1	1 1 1 1 1	1	1	1 1 0 0

(*M*)

Conditions that raise an exception:

ET_ARITHMETIC
The type was not set to VSETCTRL_TYPE_INT32

22.211 VCMI: Vector Complex Multiply Imaginary

This operation performs half of a complex multiply operation in the vector unit, calculating all the imaginary parts of the answer. Together with VCMR a complete complex vector multiply can be performed on a vector.

The operation assumes that, in memory, the vector is stored as pairs of (real, imaginary) values, where the imaginary part is stored in the higher memory addresses. After a VLD instruction, this means that the real elements are stored in bytes 0..3, 8..11, etc, of each vector and the associated imaginary elements in bytes 4..7, 12..15, etc.

Bit 30 of each number is assumed to have a magnitude of 1.0, so the final result is shifted down by 30 bits. This enables two numbers in the unit square to be multiplied without overflow.

Saturation and rounding are applied to the result.

Rounding is applied first, by adding a value `2^{2-bpe}` to the result before removing bits `bpe-3..0`. Saturation is applied in order to catch numbers that are not in the unit square, and to catch the corner case of `(1+j)^2`, which will be rounded down by a single lsb.

The instruction has no operands.

Mnemonic and operands:

```
VCMI
```

Operation:

```
for k in range(bpv/(bpe * 2)):
  t_k  =  bpe * 2 * k
  r_k  =  rnd(vD[t_k+2* bpe - 1 .. t_k+bpe] * vC[t_k+bpe - 1 .. t_k]) +
          rnd(vD[t_k+bpe - 1 .. t_k] * vC[t_k+2* bpe - 1 .. t_k+bpe])
  T_k = t_k + bpe
  vR[T_k+bpe - 1 .. T_k] <- sat(r_k[2* bpe-3 .. bpe-2] + r_k[bpe-3])
```

Encoding:

0r: No operands

0 0 0 1 1	1 1 1 1 1	1	0	1 1 1 1

(*M*)

Conditions that raise an exception:

ET_ARITHMETIC
The type was not set to VSETCTRL_TYPE_INT32

22.212 VCMR: Vector Complex Multiply Real

This operation performs half of a complex multiply operation in the vector unit, calculating all the real parts of the answer. Together with VCMI a complete complex vector multiply can be performed on a vector.

The operation assumes that, in memory, the vector is stored as pairs of (real, imaginary) values, where the imaginary part is stored in the higher memory addresses. After a VLD instruction, this means that the real elements are stored in bytes 0..3, 8..11, etc, of each vector and the associated imaginary elements in bytes 4..7, 12..15, etc.

Bit 30 of each number is assumed to have a magnitude of 1.0, so the final result is shifted down by 30 bits. This enables two numbers in the unit square to be multiplied without overflow.

Saturation and rounding are applied to the result.

Rounding is applied first, by adding a value `2^{2-bpe}` to the result before removing bits `bpe-3..0`. Saturation is applied in order to catch numbers that are not in the unit square, and to catch the corner case of `(1+j)^2`, which will be rounded down by a single lsb.

The instruction has no operands.

Mnemonic and operands:

```
VCMR
```

Operation:

```
for k in range(bpv/(bpe * 2)):
  t_k  =  bpe * 2 * k
  r_k  =  rnd(vD[t_k+bpe - 1 .. t_k] * vC[t_k+bpe - 1 .. t_k]) -
          rnd(vD[t_k+2* bpe - 1 .. t_k+bpe] * vC[t_k+2* bpe - 1 .. t_k+bpe])
  vR[t_k+bpe - 1 .. t_k] <- sat(r_k[2* bpe-3 .. bpe-2] + r_k[bpe-3])
```

Encoding:

0r: No operands

0 0 0 1 1	1 1 1 1 1	1	0	1 1 1 0

(*M*)

Conditions that raise an exception:

ET_ARITHMETIC
The type was not set to VSETCTRL_TYPE_INT32

22.213 VDEPTH1: Vector depth conversion

This operation takes a vector with elements that are `bpe` bits wide each, and binarises the vector to `epv` bits. Negative values are represented as `1`, positive values and zero as `0`.

The instruction has no operands.

Mnemonic and operands:

```
VDEPTH1
```

Operation:

```
t[bpv-1..ve] = 0
      t[i] = 1,  if vr[i] < 0
             0,  if vr[i] >= 0
                 for i in range(ve)
      vR <- t
```

Encoding:

0r: No operands

0 0 1 0 0	1 1 1 1 1	1	0	1 1 0 0

(*M*)

22.214 VDEPTH16: Vector depth conversion

This operation takes a vector with elements that are **bpe** bits wide each, and reduces the precision to 16-bits wide. Values are shifted down and rounded as appropriate.

The instruction has no operands.

Mnemonic and operands:

```
VDEPTH16
```

Operation:

```
t[bpv-1..ve * 16] = 0
t[i * 16 + 16-1..i* 16] = rnd(vR[i] >> (bpe - 16))
                       for i in range(ve)
vR <- t
```

Encoding:

0r: No operands

0 0 1 0 0	1 1 1 1 1	1	1	1 1 0 1

(*M*)

Conditions that raise an exception:

ET_ARITHMETIC
The type was set to VSETCTRL_TYPE_INT8

ET_ARITHMETIC
The type was set to VSETCTRL_TYPE_INT16

22.215 VDEPTH8: Vector depth conversion

This operation takes a vector with elements that are **bpe** bits wide each, and reduces the precision to 8-bits wide. Values are shifted down and rounded as appropriate.

The instruction has no operands.

Mnemonic and operands:

```
VDEPTH8
```

Operation:

```
t[bpv-1..ve * 8] = 0
t[i * 8 + 8-1..i* 8] = rnd(vR[i] >> (bpe - 8))
                       for i in range(ve)
vR <- t
```

Encoding:

0r: No operands

0 0 1 0 0	1 1 1 1 1	1	0	1 1 0 1

(*M*)

Conditions that raise an exception:

ET_ARITHMETIC
The type was set to VSETCTRL_TYPE_INT8

22.216 VEQCR: Tests whether vC is equal to vR

Tests on equality between vC and vR

The instruction has no operands.

Mnemonic and operands:

```
VEQCR
```

Operation:

```
r10 <- vC == vR
```

Encoding:

0r: No operands

0 0 1 1 0	1 1 1 1 1	1	0	1 1 0 1

(*M*)

22.217 VEQDR: Tests whether vD is equal to vR

Tests on equality between vD and vR

The instruction has no operands.

Mnemonic and operands:

```
VEQDR
```

Operation:

```
r10 <- vD == vR
```

Encoding:

0r: No operands

0 0 1 1 0	1 1 1 1 1	1	0	1 1 1 0

(*M*)

22.218 VFTFB: Vector FFT decimate-in-frequency backwards

Performs a backward decimate-in-frequency FFT. This operation assumes that the data has been shuffled (bit-reversed), and performs a complete backward FFT on the complex vector stored in `vR`. To compute a decimate-in-frequency FFT on a larger vector, *VLADSB*, *VCMI*, and *VCMR* can be used to implement the first rounds, and this instruction to perform the final round.

This operation is only supported for `bpe=32`.

The instruction has no operands.

Mnemonic and operands:

```
VFTFB
```

Operation:

```
assuming bpv = 256, bpe = 32
s_0 = vR[0] + vR[2]              # All arithmetic is complex
s_1 = vR[1] + vR[3]              # With real and imaginary
s_2 = vR[0] - vR[2]              # components stored in subsequent
s_3 =(vR[1] - vR[3]) * -1j       # elements of vR
vR[0] <- sat((s_0 + s_1) * x)
vR[1] <- sat((s_0 - s_1) * x)
vR[2] <- sat((s_2 + s_3) * x)
vR[3] <- sat((s_2 - s_3) * x)
   where
       x =  2.0,  if VEC_SHL
            1.0,  if VEC_SH0
            0.5,  if VEC_SHR
```

Encoding:

0r: No operands

0 0 1 0 1	1 1 1 1 1	1	1	1 1 0 0

(*M*)

Conditions that raise an exception:

ET_ARITHMETIC
The type was not set to VSETCTRL_TYPE_INT32

22.219 VFTFF: Vector FFT decimate-in-frequency Forward

Performs a forward decimate-in-frequency FFT. This operation assumes that the data has been shuffled (bit-reversed), and performs a complete forward FFT on the complex vector stored in **vR**. To compute a decimate-in-frequency FFT on a larger vector, *VLADSB*, *VCMI*, and *VCMR* can be used to implement the first rounds, and this instruction to perform the final round.

This operation is only supported for `bpe=32`.

The instruction has no operands.

Mnemonic and operands:

```
VFTFF
```

Operation:

```
assuming bpv = 256, bpe = 32
s_0 = vR[0] + vR[2]                 # All arithmetic is complex
s_1 = vR[1] + vR[3]                 # With real and imaginary
s_2 = vR[0] - vR[2]                 # components stored in subsequent
s_3 =(vR[1] - vR[3]) * -1j          # elements of vR
vR[0] <- sat((s_0 + s_1) * x)
vR[1] <- sat((s_0 - s_1) * x)
vR[2] <- sat((s_2 + s_3) * x)
vR[3] <- sat((s_2 - s_3) * x)
   where
       x =  2.0,  if VEC_SHL
            1.0,  if VEC_SH0
            0.5,  if VEC_SHR
```

Encoding:

0r: No operands

0 0 1 0 1	1 1 1 1 1	1	0	1 1 0 0

(*M*)

Conditions that raise an exception:

ET_ARITHMETIC
The type was not set to VSETCTRL_TYPE_INT32

22.220 VFTTB: Vector FFT decimate-in-time backwards

Performs a backward decimate-in-time FFT. This operation assumes that the data has been shuffled (bit-reversed), and performs a complete backward FFT on the complex vector stored in **vR**. To compute a decimate-in-time FFT on a larger vector, this instruction can perform the first round, and *VCMR*, *VCMI*, and *VLADSB* can be used to implement the later rounds.

This operation is only supported for `bpe=32`.

The instruction has no operands.

Mnemonic and operands:

```
VFTTB
```

Operation:

```
assuming bpv = 256, bpe = 32
s_0 = vD[0] + vD[1]                  # All arithmetic is complex
s_1 = vD[0] - vD[1]                  # With real and imaginary
s_2 = vD[2] + vD[3]                  # components stored in subsequent
s_3 =(vD[2] - vD[3]) * -1j           # elements of vD
vD[0] <- sat((s_0 + s_2) * x)
vD[1] <- sat((s_1 + s_3) * x)
vD[2] <- sat((s_0 - s_2) * x)
vD[3] <- sat((s_1 - s_3) * x)
   where
       x =  2.0,  if VEC_SHL
            1.0,  if VEC_SH0
            0.5,  if VEC_SHR
```

Encoding:

0r: No operands

0 0 1 0 1	1 1 1 1 1	1	1	1 1 0 1

(*M*)

Conditions that raise an exception:

ET_ARITHMETIC
The type was not set to VSETCTRL_TYPE_INT32

22.221 VFTTF: Vector FFT decimate-in-time forwards

Performs a forward decimate-in-time FFT. This operation assumes that the data has been shuffled (bit-reversed), and performs a complete forward FFT on the complex vector stored in vR. To compute a decimate-in-time FFT on a larger vector, this instruction can perform the first round, and *VCMR*, *VCMI*, and *VLADSB* can be used to implement the later rounds.

This operation is only supported for `bpe=32`.

The instruction has no operands.

Mnemonic and operands:

```
VFTTF
```

Operation:

```
assuming bpv = 256, bpe = 32
s_0 = vD[0] + vD[1]              # All arithmetic is complex
s_1 = vD[0] - vD[1]              # With real and imaginary
s_2 = vD[2] + vD[3]              # components stored in subsequent
s_3 =(vD[2] - vD[3]) * -1j       # elements of vD
vD[0] <- sat((s_0 + s_2) * x)
vD[1] <- sat((s_1 + s_3) * x)
vD[2] <- sat((s_0 - s_2) * x)
vD[3] <- sat((s_1 - s_3) * x)
   where
       x =  2.0,  if VEC_SHL
            1.0,  if VEC_SH0
            0.5,  if VEC_SHR
```

Encoding:

0r: No operands

0 0 1 0 1	1 1 1 1 1	1	0	1 1 0 1

(*M*)

Conditions that raise an exception:

ET_ARITHMETIC
The type was not set to VSETCTRL_TYPE_INT32

22.222 VGETC: Get Vector Headroom

This operation gets the vector headroom register and the control register.

The instruction has no operands.

Mnemonic and operands:

```
VGETC
```

Operation:

```
r11[bpw:16] <- 0
r11[15:6] <- vSR
r11[5:0]  <- clz(vH)
```

Encoding:

0r: No operands

0 0 0 1 1	1 1 1 1 1	1	0	1 1 0 1

(*M*)

22.223 VLADD: Vector Load and Add

Loads a vector from memory, add the vector element by element to the vector in the output register `vR`, and store the result vector in the output register `vR`: `vR[] <- vR[] + mem[s][]`.

The addition is signed, and the signed result is saturated if it cannot be represented in `bpe` bits. That is, values less than 0b100...0 are replaced by 0b100...0, and values larger than 0b011...1 are replaced by 0b011...1.

Note that this operation can be used to add both complex numbers and normal numbers.

The instruction has one operand:

op1
: s, Operand register, one of `r0`... `r11`

Mnemonic and operands:

```
VLADD s
```

Operation:

```
t = vR[k] + mem[s + k..s + k + bpe/8-1],
vR[k] <- sat(t)
                for k in range(ve)
```

Encoding:

1r: Register

1 0 1 0 1	1 1 1 1 1	1	0	. . . .

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
: `s` is not word aligned, or a complete vector cannot be loaded from this address.

22.224 VLADDD: Vector Load and Add to vD

Loads a vector from memory, add the vector element by element to the vector in the output register `vD`, and store the result vector in the output register `vD`: `vD[] <- vD[] + mem[r10][]`.

The addition is signed, and the signed result is saturated if it cannot be represented in `bpe` bits. That is, values less than 0b100…0 are replaced by 0b100…0, and values larger than 0b011…1 are replaced by 0b011…1.

Note that this operation can be used to add both complex numbers and normal numbers.

The instruction has no operands.

Mnemonic and operands:

```
VLADDD
```

Operation:

```
t = vD[k] + mem[op1 + k..op1 + k + bpe/8-1],
vD[k] <- sat(t)
                    for k in range(ve)
```

Encoding:

0r: No operands

0 0 1 1 0	1 1 1 1 1	1	1	1 1 0 1

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
`r10` is not word aligned, or a complete vector cannot be loaded from this address.

22.225 VLADSB: Vector Load and Butterfly

Loads a vector from memory and perform a butterfly operation.

This operation is only supported for `bpe=32`.

The instruction has one operand:

op1
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
VLADSB s
```

Operation:

```
for i in range(bpv/(2*bpe):
  vR[i] <- sat((t[i] +_{c} vR[i]) * x)
  vD[i] <- sat((t[i] -_{c} vR[i]) * x)
  where
        t = mem[s]
        x =  2.0,  if VEC_SHL
             1.0,  if VEC_SH0
             0.5,  if VEC_SHR
```

Encoding:

1r: Register

1 0 1 0 0	1 1 1 1 1	1	0	. . . .

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
: s is not word aligned, or a complete vector cannot be loaded from this address.

ET_ARITHMETIC
: The type was not set to VSETCTRL_TYPE_INT32

22.226 VLASHR: Vector Load and Arithmetic Shift Right

This instruction loads a vector from memory address `s`, shifts each element by `t` positions, and stores the result in `vR`.

A arithmetic right shift is performed if `t` is positive. If the value in `t` is greater or equal to `bpe`, then the value in vR is replaced by a sequence of sign bits.

For negative values of `t` a saturating left shift is performed. That is, the value is shifted left, and if an overflow occurs, either `2^{bpe}-1` or `-2^{bpe} + 1` is stored in the vector element.

The instruction has two operands:

op1
: s, Operand register, one of `r0`… `r11`

op2
: t, Operand register, one of `r0`… `r11`

Mnemonic and operands:

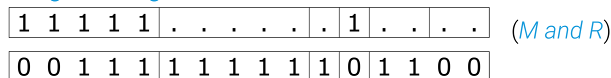
```
VLASHR s,t
```

Operation:

```
p = mem[s]
vR[i] <-  p[i] >> t,            if t in {0..bpe-1}
             p[i] >> bpe-1,        if t in {bpe..}
             ssat(p[i] << -t),     if t in {-bpe+1..-1}
             ssat(p[i] << bpe-1), if t in {..-bpe}
             for i range(ve)
```

Encoding:

l2r: Two register long



1 1 1 1 1	. . . . .	.	1	. .	. .
0 0 1 1 1	1 1 1 1 1	1	0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_LOAD_STORE
: `s` is not word aligned, or a complete vector cannot be loaded from this address.

22.227 VLDC: Load Vector vC

Loads the coefficient register of the vector unit, vC, from memory. Register s is used as the base address of the vector in memory. The vector is stored LSB first in memory. The address must be aligned with the vector width.

The instruction has one operand:

op1
: s, Operand register, one of r0… r11

Mnemonic and operands:

```
VLDC s
```

Operation:

```
for k in range(bpv // 8):
  vC[k* 8+7:k* 8] <- mem[s + k]
```

Encoding:

1r: Register

1 0 0 1 0	1 1 1 1 1	1	0	. . . .

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
: s is not word aligned, or a complete vector cannot be loaded from this address.

22.228 VLDD: Load Vector vD

Loads the data register of the vector unit, vD, from memory. Register s is used as the base address of the vector in memory. The vector is stored LSB first in memory. The address must be aligned with the vector width.

The instruction has one operand:

op1
: s, Operand register, one of r0… r11

Mnemonic and operands:

```
VLDD s
```

Operation:

```
for k in range(bpv // 8):
  vD[k* 8+7:k* 8] <- mem[s + k]
```

Encoding:

1r: Register

1 0 0 1 0	1 1 1 1 1	1	1	. . . .

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
: s is not word aligned, or a complete vector cannot be loaded from this address.

22.229 VLDR: Vector Load vR

Loads the result register of the vector unit, vR, from memory. Register r11 is used as the base address of the vector in memory. The vector is stored LSB first in memory. The address must be aligned with the vector width.

The instruction has no operands.

Mnemonic and operands:

```
VLDR
```

Operation:

```
for k in range(bpv // 8):
  vR[k* 8+7:k* 8] <- mem[r11 + k]
```

Encoding:

0r: No operands

0 0 0 1 1	1 1 1 1 1	1	1	1 1 1 0

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
`r11` is not word aligned, or a complete vector cannot be loaded from this address.

22.230 VLMACC: Vector Load and MACC reduce

Loads a a vector from memory and performs a multiply accumulate across the vector. The vector loaded from memory is elementwise multiplied with **vC**. All products are shifted down by **bpe-2** bits in order to scale the result. All products are then summed to results held in **vR** and **vD**

vR and **vD** hold an extended precision result, where **vD** is used to store up to an extra **vac** bits to avoid overflow. The **vD** bits are stored with extra sign bits to make sure that all **bpe** bits in each element of **vD** have a meaningful value. VLSAT can be used to reduce the values in **vD** and **vR** and saturate.

The instruction has one operand:

op1
: s, Operand register, one of `r0`... `r11`

Mnemonic and operands:

```
VLMACC s
```

Operation:

```
tmp = sum( x_k * y_k, for k in range(bpe))
      where
        x_k = vD[k* bpe+bpe-1:k* bpe]
        y_k = mem[s + k:s + k + bpe/8-1]
vR[0:2* bpe] <- sat(vR[0:2* bpe] + tmp)
```

Encoding:

1r: Register

1 0 1 1 1	1 1 1 1 1	1	0	. . . .

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
: **s** is not word aligned, or a complete vector cannot be loaded from this address.

22.231 VLMACCR: Vector Load and MACC reduce

Loads a a vector from memory and computes an inner-product between this vector and `vC`. The vector loaded from memory is elementwise multiplied with `vC` and all products are summed together to form an inner product. The inner product is shifted down by `bpe-2` bits in order to scale the result.

`vR` and `vD` hold a set of extended precision results with the least significant `bpe` bits of each element in `vR` and an extra `vac` bits stored in each element of `vD`. Both vecotrs are rotated left by one element on executing VLMACCR and the inner product is added to element zero stored in `vR` and `vD`.

VLSAT can be used to reduce the values in `vD` and `vR` and saturate.

The instruction has one operand:

op1
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
VLMACCR s
```

Operation:

```
tmp = sum(x_k * y_k)
where
    k in range(bpv // bpe)
    x_k = vD[k* bpe+bpe-1:k* bpe]
    y_k = mem[s + k:s + k + bpe/8-1]
vR[0:2* bpe] <- sat(vR[0:2* bpe] + tmp)
```

Encoding:

1r: Register

1 0 1 0 0	1 1 1 1 1	1	1	. . . . .

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
: `s` is not word aligned, or a complete vector cannot be loaded from this address.

22.232 VLMACCR1: Vector Load and MACC reduce

Loads a a vector from memory and computes an inner product assuming the vector holds `bpv` values: 0 represents +1 and 1 represents -1. The inner product is divided by 2 (as the result is always even), and then added to a partial results stored in `vR` and `vD` as follows.

`vR` and `vD` hold a set of extended precision results with the least significant `bpe` bits of each element in `vR` and an extra `vac` bits stored in each element of `vD`. Both vectors are rotated left by one element on executing VLMACCR1 and the inner product is added to element zero stored in `vR` and `vD`.

VLSAT can be used to reduce the values in `vD` and `vR` and saturate.

The instruction has one operand:

op1
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
VLMACCR1 s
```

Operation:

```
t = sum((1 - 2 x[k]) * (1-2 y[k]), for k in range(bpv))
    where
      x_k = vD[k* bpe+bpe-1:k* bpe]
      y_k = mem[s + k:s + k + bpe/8-1]
vR[0:2* bpe] <- sat(vR[0:2* bpe] + tmp)
```

Encoding:

1r: Register

1 1 0 0 0	1 1 1 1 1	1	0	. . . .

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
: `s` is not word aligned, or a complete vector cannot be loaded from this address.

22.233 VLMUL: Vector Load and Multiply

Loads a vector from memory, multiply the vector element by element with the vector in the result register `vR`, and store the result vector in the output register `vR`: `vR[] <- vR[] * mem[s][]`.

Note that this operation is not suitable for complex numbers, use VCMR, VCMI, VCMCR, and VCMCI for complex numbers.

The multiplication is signed, and bit 30 of all operands number is assumed to have a magnitude of 1.0. Hence, the final result is shifted down by 30 bits. This enables two numbers in the range `[-1..1]` inclusive to be multiplied without overflow.

Saturation and rounding are applied to the result.

Rounding is applied first, by adding one to bit position `bpe - 3` to the result before removing bits `bpe-3..0`. This rounds the result to the nearest representable value, with a tie rounding up. Saturation is applied in order to catch results that do not fit.

The instruction has one operand:

op1
s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
VLMUL s
```

Operation:

```
t = vR[k] * mem[s + k..s + k + bpe/8-1] * 2^(bpe-2)
vR[k] <- sat(t)
        for k in range(ve)
```

Encoding:

1r: Register

1 0 1 1 0	1 1 1 1 1	1	0	. . . .

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
s is not word aligned, or a complete vector cannot be loaded from this address.

22.234 VLSAT: Vector Load and saturate

Performs saturation of double precision results stored accross vD and vR. Each double precision result is shifted right by a number of bits that is loaded from memory. The shifted value is then rounded and saturated, and stored in vR.

The instruction has one operand:

op1
: s, Operand register, one of r0... r11

Mnemonic and operands:

```
VLSAT s
```

Operation:

```
t = mem[s]
vR[i] <- sat(vD[i]:vR[i] >> t[i]),   if t[i] >= 0
vR[i] <- sat(vD[i]:vR[i] << -t[i]),  if t[i] < 0
vD[i] <- 0
               for i range(ve)
```

Encoding:

1r: Register

1 0 1 1 1	1 1 1 1 1	1	1	. . . . .

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
: s is not word aligned, or a complete vector cannot be loaded from this address.

22.235 VLSUB: Vector Load and Subtract

Loads a vector from memory, subtract the vector element by element from the vector in the output register `vR`, and store the result vector in the output register `vR`: `vR[] <- mem[s][] - vR[]`.

The subtraction is signed, and the signed result is saturated if it cannot be represented in `bpe` bits. That is, values less than 0b100…0 are replaced by 0b100…0, and values larger than 0b011…1 are replaced by 0b011…1.

Note that this operation can be used to add both complex numbers and normal numbers.

The instruction has one operand:

op1
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
VLSUB s
```

Operation:

```
t = mem[s + k..s + k + bpe/8-1] - vR[k]
vR[k] <- sat(t)
                    for k in range(ve)
```

Encoding:

1r: Register

1 0 1 0 1	1 1 1 1 1	1	1	. . . .

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
: `s` is not word aligned, or a complete vector cannot be loaded from this address.

22.236 VPOS: Vector make positive

This operation replaces all negative values in vR with zero.

The instruction has no operands.

Mnemonic and operands:

```
VPOS
```

Operation:

```
for i in range(ve):
            if vR[i] < 0:
                vR[i] <- 0
```

Encoding:

0r: No operands

0 0 1 0 0	1 1 1 1 1	1	1	1 1 1 0

(*M*)

22.237 VSETC: Set Vector Control/Status Register

This operation sets the control register and the headroom in the vector unit. If the type is not a known type, it is set to zero.

The instruction has no operands.

Mnemonic and operands:

```
VSETC
```

Operation:

```
vSR <- r11[15:6]
vH  <- mkmsk(32-r11[5:0])
```

Encoding:

0r: No operands

0 0 0 1 1	1 1 1 1 1	1	0	1 1 0 0

(*M*)

22.238 VSIGN: Vector sign computation

This operation takes a vector in **vR** and replaces negative values with -1 (0b11000…000), and non-negative values with +1 (0b01000…000).

The instruction has no operands.

Mnemonic and operands:

```
VSIGN
```

Operation:

```
for i in range(ve):
    if R[i] < 0:
        vR[i] <- -2^(bpe-2)
    elif vR[i] > 0:
        vR[i] <- 2^(bpe-2)
```

Encoding:

0r: No operands

0 0 1 0 0	1 1 1 1 1	1	0	1 1 1 0

(*M*)

22.239 VSTC: Vector Store vC

Stores the the contents of the coefficient register of the vector unit, **vC**, in memory. Register r11 is used as the base address of the vector in memory. The vector is stored LSB first. The address must be aligned with the vector width.

The instruction has no operands.

Mnemonic and operands:

```
VSTC
```

Operation:

```
for k in range(bpv // 8):
  tmp = vC[k* 8+7:k* 8]
  mem[r11 + k] <- tmp
  vH <- vH | |tmp|
```

Encoding:

0r: No operands

0 0 0 1 1	1 1 1 1 1	1	1	1 1 1 1

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
r11 is not word aligned, or a complete vector cannot be stored at this address.

22.240 VSTD: Store Vector vD

Stores the contents of the data register of the vector unit, `vD`, in memory. Register `s` is used as the base address of the vector in memory. The vector is stored LSB first. The address must be aligned with the vector width.

The instruction has one operand:

op1
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
VSTD s
```

Operation:

```
for k in range(bpv // 8):
  tmp = vD[k* 8+7:k* 8]
  mem[s + k] <- tmp
  vH <- vH | |tmp|
```

Encoding:

1r: Register

1 0 0 1 1	1 1 1 1 1	1	0	. . . . .

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
: `s` is not word aligned, or a complete vector cannot be stored at this address.

22.241 VSTR: Store Vector R

Stores the contents of the result register of the vector unit, **vR**, in memory. Register **s** is used as the base address of the vector in memory. The vector is stored LSB first. The address must be aligned with the vector width.

The instruction has one operand:

op1
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
VSTR s
```

Operation:

```
for k in range(bpv // 8):
  tmp = vR[k* 8+7:k* 8]
  mem[s + k] <- tmp
  vH <- vH | |tmp|
```

Encoding:

1r: Register

1 0 0 1 1	1 1 1 1 1	1	1	. . . . .

(*M*)

Conditions that raise an exception:

ET_LOAD_STORE
: **s** is not word aligned, or a complete vector cannot be stored at this address.

22.242 VSTRPV: Store part of Vector R

Stores part of the contents of the result register of the vector unit, vR, in memory. Register s is used as the base address of the vector in memory. The vector is stored LSB first. s must be word aligned. t is a mask specifying which bytes must be stored. Combined with a MKMSK instruction the lower n bytes of a vector can be stored.

The instruction has two operands:

op1
: s, Operand register, one of r0… r11

op2
: t, Operand register, one of r0… r11

Mnemonic and operands:

```
VSTRPV s,t
```

Operation:

```
for k in range(bpw):
  if t[k] == 1:
    mem[s+k] <- vR[k * 8..k * 8+7]
```

Encoding:

l2r: Two register long

1 1 1 1 1	. . . . .	.	0	. .	. .
0 0 1 1 1	1 1 1 1 1	1	0	1 1	0 0

(*M and R*)

Conditions that raise an exception:

ET_LOAD_STORE
: s is not word aligned, or a complete vector cannot be stored at this address.

22.243 WAITEF: If false wait for event

Waits for an event when a condition is false. If the condition is 0 (false), then the EEBLE is set, and, if no event is ready it will suspend the thread until an event becomes ready. When an event is available, the thread will continue at the address specified by the event. If the condition is not 0, the next instruction will be executed. The current PC is not saved anywhere.

The instruction has one operand:

op1
: c, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
WAITEF c
```

Operation:

```
if c == 0:
   sr[eeble] <- 1
```

Encoding:

1r: Register

0 0 0 0 1	1 1 1 1 1	1	1	. . . .

(*R*)

22.244 WAITET: If true wait for event

Waits for an event when a condition is true. If the condition not 0, then the EEBLE is set, and, if no event is ready it will suspend the thread until an event becomes ready. When an event is available, the thread will continue at the address specified by the event. If the condition is 0 (false), the next instruction will be executed. The current PC is not saved anywhere.

The instruction has one operand:

op1
: c, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
WAITET c
```

Operation:

```
if c != 0:
   sr[eeble] <- 1
```

Encoding:

1r: Register

0 0 0 0 1	1 1 1 1 1	1	0	. . . .

(*R*)

22.245 WAITEU: Wait for event

Waits for an event. This instruction sets EEBLE and, if no event is ready it will suspend the thread until an event becomes ready. When an event is available, the thread will continue at the address specified by the event. The current PC is not saved anywhere.

The instruction has no operands.

Mnemonic and operands:

```
WAITEU
```

Operation:

```
sr[eeble] <- 1
```

Encoding:

0r: No operands

0 0 0 0 0	1 1 1 1 1	1	0	1 1 0 0

(*R*)

22.246 XOR: Bitwise exclusive or

Produces the bitwise exclusive-or of two words.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
XOR d,x,y
```

Operation:

```
d <- x ⊕ y
```

Encoding:

l3r: Three register long

1 1 1 1 1	. . . . .	. .	. .	. .

0 0 0 0 1	1 1 1 1 1	1	0	1 1 0 0

(*M and R*)

22.247 XOR4: Bitwise exclusive-or of four words

Produces the bitwise exclusive-or of four words.

The instruction has five operands:

op1
: d, Operand register, one of `r0`… `r11`

op4
: e, Operand register, one of `r0`… `r11`

op2
: x, Operand register, one of `r0`… `r11`

op3
: y, Operand register, one of `r0`… `r11`

op5
: v, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
XOR4 d,e,x,y,v
```

Operation:

```
d <- x @ y @ e @ v
```

Encoding:

l5r: Five register long

1 1 1 1 1	. . . . .	. .	. .	. .
0 0 0 0 1	. . . . .	. 1	. .	. .

(*M and R*)

22.248 ZEXT: Zero extend

Zero extends an n-bit field stored in a register. The first operand of this instruction is both a source and destination operand. The second operand contains the bit position. All bits at a position higher or equal are cleared.

The instruction has two operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: s, Operand register, one of `r0`… `r11`

Mnemonic and operands:

```
ZEXT d,s
```

Operation:

```
if s > 0 && s < bpw:
  d <- 0:...:0:d[s-1...0]
```

Encoding:

2r: Two register

0 1 0 0 0	. . . . .	.	0	. .	. .

(*M or R*)

22.249 ZEXTI: Zero extend immediate

Zero extends an n-bit field stored in a register. The first operand of this instruction is both a source and destination operand. The second operand contains the bit position. All bits at a position higher or equal are cleared.

The instruction has two operands:

op1
: s, Operand register, one of `r0`… `r11`

op2
: bitp, A bit position; one of `bpw`, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32

Mnemonic and operands:

```
ZEXTI s,bitp
```

Operation:

```
if bitp > 0 && bitp < bpw:
  s <- 0:...:0:s[bitp-1...0]
```

Encoding:

rus: Register with immediate

0 1 0 0 0	. . . . .	.	1	. .	. .

(*M or R*)

22.250 ZIP: Zips together a pair of registers

Zips a pair of registers in bit, bit-pairs, nibbles, bytes or byte-pairs. The granularity of zipping is determined by `2^{s}`. Each of **d** and **e** are chopped into chunks of `2^{s}` bits. They are then zipped together by starting with the most significant chunk of **d**, the most significant chunk of **e**, then next significant chunk of **d** and so on until the least significant chunks of **e** and **d**. This results in a bit string of `2* bpw` bits, the most significant **bpw** bits are written back to **d**, the least significant **bpw** bits to **e**.

The instruction has three operands:

op1
: d, Operand register, one of `r0`… `r11`

op2
: e, Operand register, one of `r0`… `r11`

op3
: s, An integer in the range 0…11

Mnemonic and operands:

```
ZIP d,e,s
```

Operation:

```
w = 2^s
z = d[bpw-1..bpw-w]:
    e[bpw-1..bpw-w]:
    d[bpw-w-1..bpw-2 * w]:
    e[bpw-w-1..bpw-2 * w]: ... :
    d[w-1..0]:
    e[w-1..0]:
d <- z[2 bpw-1..bpw]
e <- z[bpw-1..0]
```

Encoding:

l2rus: Two register with immediate long

1 1 1 1 1	. . . . . .	. .	. .	. .
1 0 0 1 1	1 1 1 1 1	1 0	1 1	1 0

(*M and R*)

23 XCore XS3 Instruction Format Specification

This section defines the instruction-formats. For each instruction format there is a name, a short description of its purpose, then a graphical representation of the encoding, and finally a list of instructions that use this instruction encoding. The graphical representation shows the bits of the instruction, bits are numbered from 15 down to 0. If a bit value depends on the opcode, then this is marked with a `x` symbol. If a bit value depends on an operand this is marked with a `.`, and the particular encoding for that operand is shown underneath. Otherwise, the bit will have a value of 0 or 1, in order to differentiate between formats.

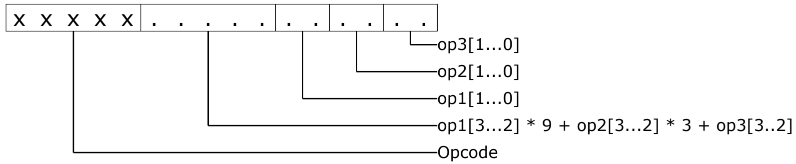
23.1 3r: Three register

Instructions with three operand registers; the last two operands are always source registers, the first operand is always a destination register

The syntax for this instruction is:

```
MNEMONIC op1, op2, op3
```

Instructions in this format are encoded in 16 bits:



This format is used by the following instructions:

ADD	*LD8U*	*LSS*	*SHL*
AND	*LD16S*	*LSU*	*SHR*
EQ	*LDW*	*OR*	*SUB*

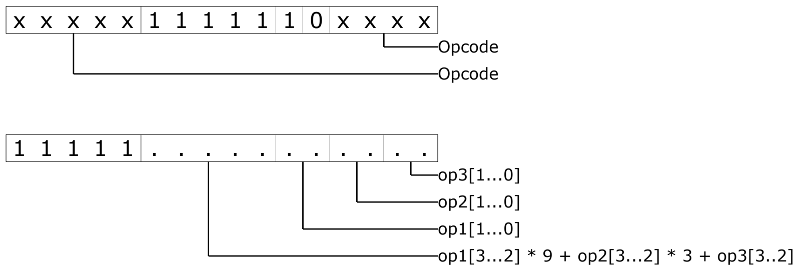
23.2 l3r: Three register long

Instructions with three operand registers; the last two operands are always source operands, the first operand usually refers to the destination register (with the exception of store instruction)

The syntax for this instruction is:

```
MNEMONIC op1, op2, op3
```

Instructions in this format are encoded in 32 bits:



This format is used by the following instructions:

ASHR	*FEQ*	*LDA16B*	*REMU*
CRC	*FGT*	*LDA16F*	*LSATS*
DIVS	*FLT*	*LDAWB*	*ST8*
DIVU	*FMUL*	*LDAWF*	*ST16*
DVLD	*FSEXP*	*MUL*	*STW*
DVST	*FSUB*	*OUTPW*	*TSETR*
FADD	*FUN*	*REMS*	*XOR*

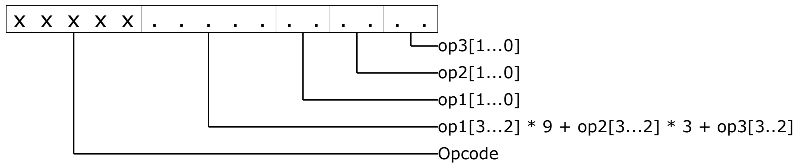
23.3 2rus: Two register with immediate

Instructions with three operands. The last operand is a small unsigned constant (0..11), the second operand is a source register, the first operand is either a destination register, or a second source register in the case of memory-store operations.

The syntax for this instruction is:

```
MNEMONIC op1, op2, op3
```

Instructions in this format are encoded in 16 bits:



This format is used by the following instructions:

ADDI	*LDWI*	*SHRI*	*SUBI*
EQI	*SHLI*	*STWI*	

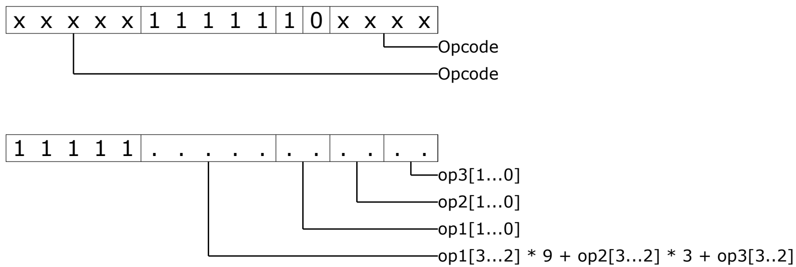
23.4 l2rus: Two register with immediate long

Instructions with three operands. The last operand is a small unsigned constant (0..11), the second operand is a source register, the first operand is either a destination register, or a second source register in the case of some resource operations.

The syntax for this instruction is:

```
MNEMONIC op1, op2, op3
```

Instructions in this format are encoded in 32 bits:



This format is used by the following instructions:

ASHRI	*LDAWFI*	*STDSP*
INPW	*LDDSP*	*UNZIP*
LDAWBI	*OUTPWI*	*ZIP*

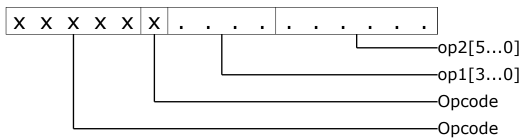
23.5 ru6: Register with 6-bit immediate

Instructions with two operands where the first operand is a register and the second operand is a 6-bit integer constant. This format used, amongst others, for load and store operations relative to the stack pointer and data pointer.

The syntax for this instruction is:

```
MNEMONIC op1, op2
```

Instructions in this format are encoded in 16 bits:



This format is used by the following instructions:

BRBF	*LDAWDP*	*LDWDP*	*STWSP*
BRBT	*LDAWSP*	*LDWSP*	
BRFF	*LDC*	*SETCI*	
BRFT	*LDWCP*	*STWDP*	

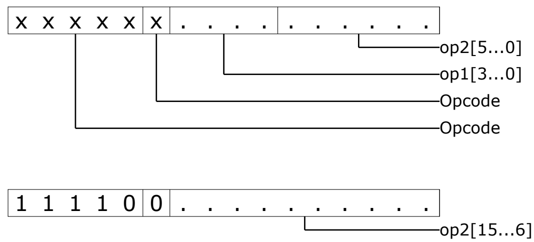
23.6 lru6: Register with 16-bit immediate

Instructions with two operands where the first operand is a register and the second operand is a 16-bit integer constant. This instruction is a prefixed version of *ru6*. This format is used, amongst others, for load and store operations relative to the stack pointer and data pointer.

The syntax for this instruction is:

```
MNEMONIC op1, op2
```

Instructions in this format are encoded in 32 bits:



This format is used by the following instructions:

BRBF	*LDAWDP*	*LDWDP*	*STWSP*
BRBT	*LDAWSP*	*LDWSP*	
BRFF	*LDC*	*SETCI*	
BRFT	*LDWCP*	*STWDP*	

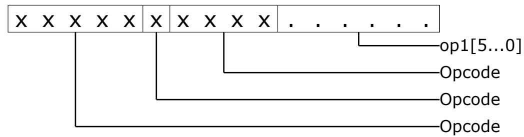
23.7 u6: 6-bit immediate

Instructions with a single operand encoding a 6-bit integer.

The syntax for this instruction is:

```
MNEMONIC op1
```

Instructions in this format are encoded in 16 bits:



This format is used by the following instructions:

BLAT	*DUALENTSP*	*GETSR*	*RETSP*
BRBU	*ENTSP*	*KCALLI*	*SETSR*
BRFU	*EXTDP*	*KENTSP*	
CLRSR	*EXTSP*	*LDAWCP*	

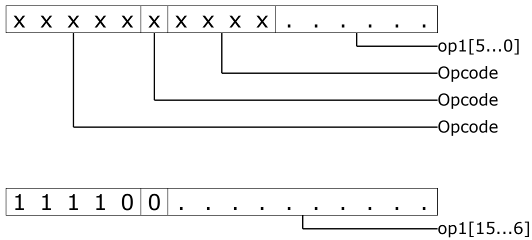
23.8 lu6: 16-bit immediate

Instructions with a single operand encoding a 16-bit integer. This instruction is a prefixed version of *u6*.

The syntax for this instruction is:

```
MNEMONIC op1
```

Instructions in this format are encoded in 32 bits:



This format is used by the following instructions:

BLAT	*DUALENTSP*	*GETSR*	*LDAWCP*
BRBU	*ENTSP*	*KCALLI*	*RETSP*
BRFU	*EXTDP*	*KENTSP*	*SETSR*
CLRSR	*EXTSP*	*KRESTSP*	

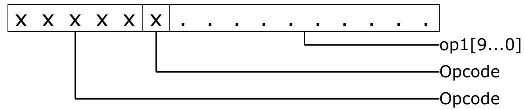
23.9 u10: 10-bit immediate

Instructions with a single operand encoding a 10-bit integer.

The syntax for this instruction is:

```
MNEMONIC op1
```

Instructions in this format are encoded in 16 bits:



This format is used by the following instructions:

BLACP	*BLRF*	*LDAPF*
BLRB	*LDAPB*	*LDWCPL*

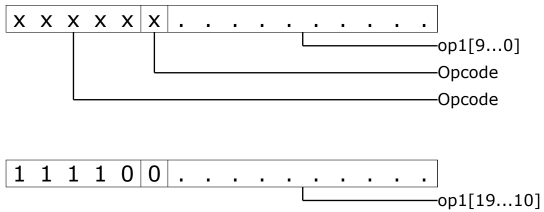
23.10 lu10: 20-bit immediate

Instructions with a single operand encoding a 20-bit integer. This instruction is a prefixed version of *u10*.

The syntax for this instruction is:

```
MNEMONIC op1
```

Instructions in this format are encoded in 32 bits:



This format is used by the following instructions:

BLACP	*BLRF*	*LDAPF*
BLRB	*LDAPB*	*LDWCPL*

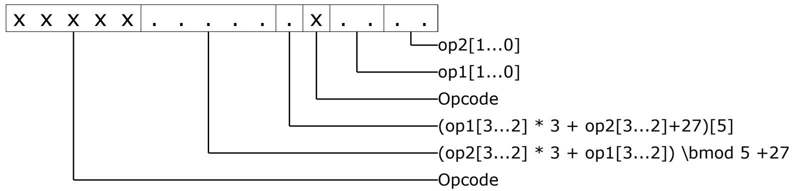
23.11 2r: Two register

Instructions with two operand registers; the last operand is always a source register, the first operand maybe a destination register.

The syntax for this instruction is:

```
MNEMONIC op1, op2
```

Instructions in this format are encoded in 16 bits:



This format is used by the following instructions:

ANDNOT	*EEF*	*INCT*	*OUTCT*
BITREV	*EET*	*INSHR*	*PEEK*
BYTEREV	*ENDIN*	*INT*	*SEXT*
CHKCT	*GETST*	*MKMSK*	*TESTCT*
CLS	*GETTS*	*NEG*	*TESTWCT*
CLZ	*IN*	*NOT*	*ZEXT*

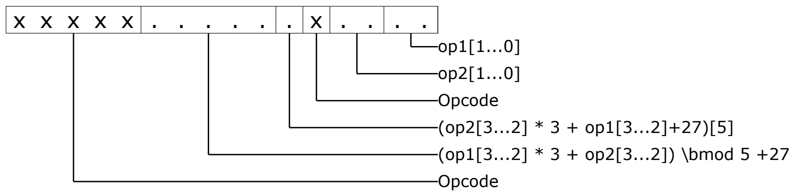
23.12 r2r: Two register reversed

Instructions with two operand registers used for resources; the first operand is always a source register containing the resource to operate on, the last operand maybe a destination register.

The syntax for this instruction is:

```
MNEMONIC op1, op2
```

Instructions in this format are encoded in 16 bits:



This format is used by the following instructions:

OUT	*OUTT*	*SETPSC*
OUTSHR	*SETD*	*SETPT*

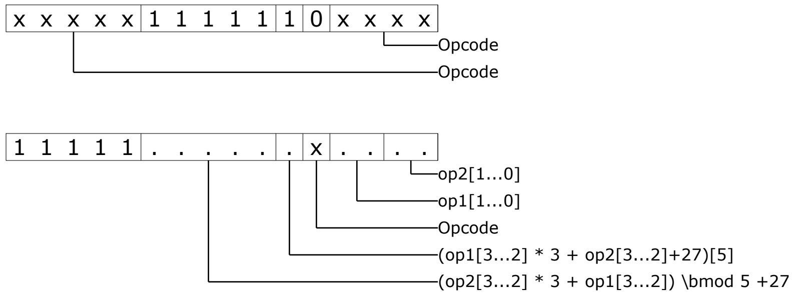
23.13 l2r: Two register long

Instructions with two operand registers; the last operand is always a source register, the first operand maybe a destination register.

The syntax for this instruction is:

```
MNEMONIC op1, op2
```

Instructions in this format are encoded in 32 bits:



This format is used by the following instructions:

FMANT	*GETPS*	*TINITDP*	*TSETMR*
FSPEC	*SETC*	*TINITLR*	*VLASHR*
GETD	*TESTLCL*	*TINITPC*	*VSTRPV*
GETN	*TINITCP*	*TINITSP*	

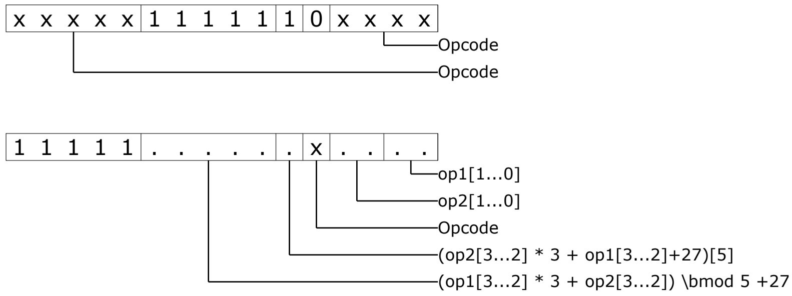
23.14 lr2r: Two register reversed long

Instructions with two operand registers; the first operand is always a source register containing a resource identifier, the last operand maybe a destination register.

The syntax for this instruction is:

```
MNEMONIC op1, op2
```

Instructions in this format are encoded in 32 bits:



This format is used by the following instructions:

SETCLK	*SETPS*	*SETTW*
SETN	*SETRDY*	

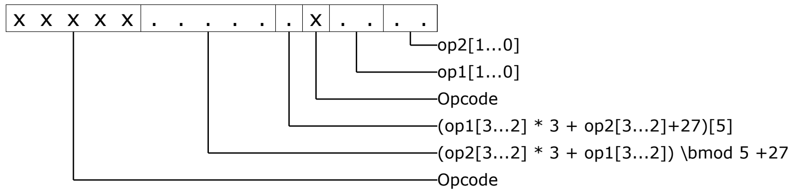
23.15 rus: Register with immediate

Instructions with two operands. The last operand is a small constant (0..11). The first operand is a register that may be used as source and or destination.

The syntax for this instruction is:

```
MNEMONIC op1, op2
```

Instructions in this format are encoded in 16 bits:



This format is used by the following instructions:

CHKCTI	*MKMSKI*	*SEXTI*
GETR	*OUTCTI*	*ZEXTI*

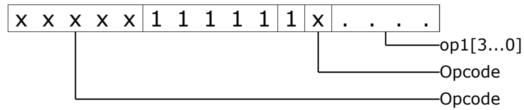
23.16 1r: Register

Instructions with one operand register.

The syntax for this instruction is:

```
MNEMONIC op1
```

Instructions in this format are encoded in 16 bits:



This format is used by the following instructions:

BAU	*FENAN*	*SETV*	*VLMUL*
BLA	*FREER*	*SYNCR*	*VLSAT*
BRU	*GETTIME*	*TSTART*	*VLSUB*
CLRPT	*KCALL*	*VLADD*	*VSTD*
DGETREG	*MJOIN*	*VLADSB*	*VSTR*
ECALLF	*MSYNC*	*VLDC*	*WAITEF*
ECALLT	*SETCP*	*VLDD*	*WAITET*
EDU	*SETDP*	*VLMACC*	
EEU	*SETEV*	*VLMACCR*	
ELATE	*SETSP*	*VLMACCR1*	

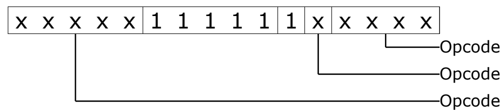
23.17 0r: No operands

These instructions operate on implicit operands.

The syntax for this instruction is:

```
MNEMONIC
```

Instructions in this format are encoded in 16 bits:



This format is used by the following instructions:

CLRE	*LDSPC*	*VCMCI*	*VFTTF*
DCALL	*LDSSR*	*VCMCR*	*VGETC*
FLUSH	*NOP*	*VCMI*	*VLADDD*
FREET	*PREFETCH*	*VCMR*	*VLDR*
GETED	*SETKEP*	*VDEPTH1*	*VPOS*
GETET	*SSYNC*	*VDEPTH16*	*VSETC*
GETID	*STET*	*VDEPTH8*	*VSIGN*
GETKEP	*STSED*	*VEQCR*	*VSTC*
GETKSP	*STSPC*	*VEQDR*	*WAITEU*
INVALIDATE	*STSSR*	*VFTFB*	
LDET	*VCLRDR*	*VFTFF*	
LDSED	*VADDDR*	*VFTTB*	

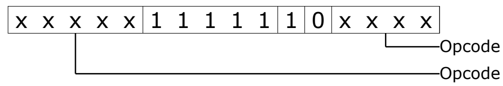
23.18 l0r: No operands

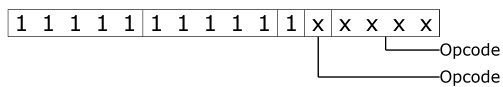
These instructions operate on implicit operands.

The syntax for this instruction is:

```
MNEMONIC
```

Instructions in this format are encoded in 32 bits:





This format is used by the following instructions:

DENTSP	DRESTSP	DRET	KRET

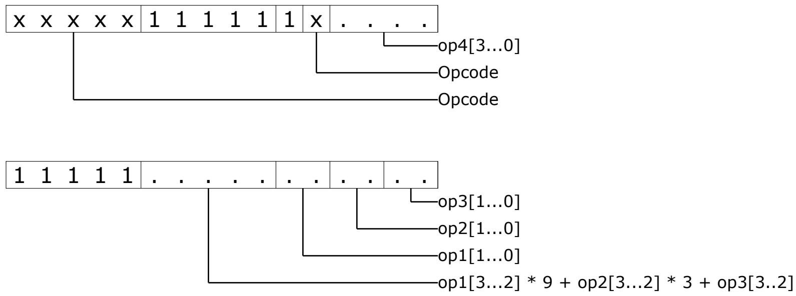
23.19 l4r: Four register long

Operations on four registers - the last two operands are source registers, the first two may be used as source and or destination registers.

The syntax for this instruction is:

```
MNEMONIC op1, op4, op2, op3
```

Instructions in this format are encoded in 32 bits:



This format is used by the following instructions:

CRC8	*FMACC*	*MACCS*	*STD*
CRCN	*LDD*	*MACCU*	

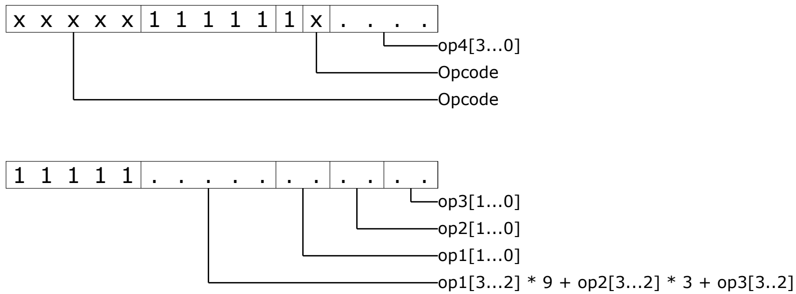
23.20 l3rus: Three register with immediate long

Operations on three registers and an immediate - the third operand is a source register, the first two may be used as source and or destination registers.

The syntax for this instruction is:

```
MNEMONIC op1, op4, op2, op3
```

Instructions in this format are encoded in 32 bits:



This format is used by the following instructions:

LDDI	*STDI*

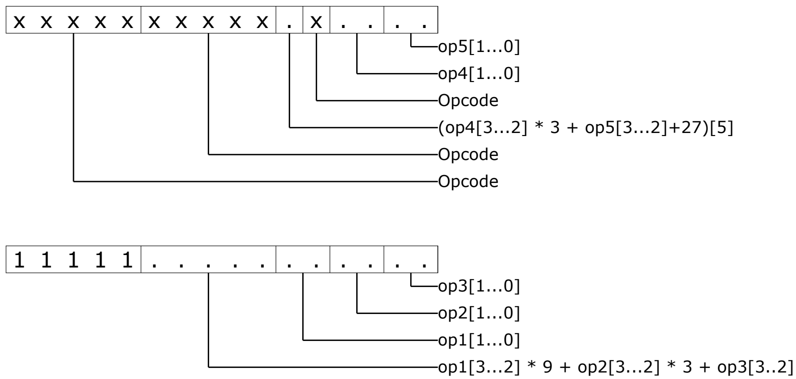
23.21 l4rus: Four registers with immediate long

Instruction with five operands. The last operand is a small unsigned constant (0..11), the third and fourth operands are source registers, the first and second operands may be used as source and or destination registers.

The syntax for this instruction is:

```
MNEMONIC op1, op4, op2, op3, op5
```

Instructions in this format are encoded in 32 bits:



This format is used by the following instructions:

CRC32_INC	LEXTRACT	LINSERT

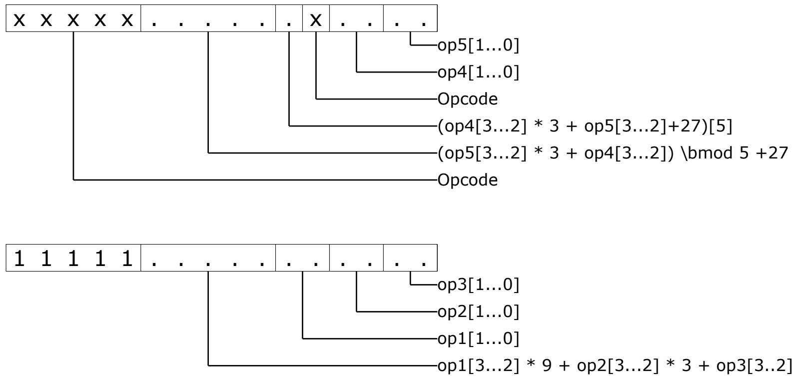
23.22 l5r: Five register long

Operations on five registers - the last three operands are source registers, the first two may be used as source and or destination registers.

The syntax for this instruction is:

```
MNEMONIC op1, op4, op2, op3, op5
```

Instructions in this format are encoded in 32 bits:



This format is used by the following instructions:

FMAKE	*LDIVU*	*XOR4*
LADD	*LSUB*	

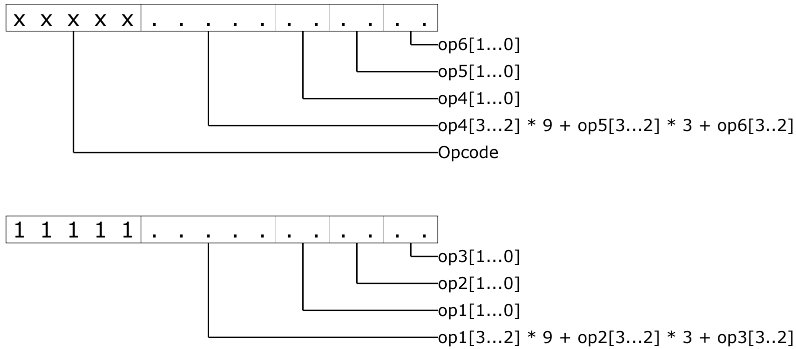
23.23 l6r: Six register long

Operations on six registers - the last four operands are source registers, the first two may be used as source and or destination registers.

The syntax for this instruction is:

```
MNEMONIC op1, op4, op2, op3, op5, op6
```

Instructions in this format are encoded in 32 bits:



This format is used by the following instructions:

LMUL

24 XCore XS3 Exceptions

Exceptions change the normal flow of control; they may be caused by interrupts, errors arising during instruction execution and by system calls. On an exception, the processor will save the `pc` and `sr` in `spc` and `ssr`, disable events and interrupts, and start executing an exception handler. The program counter that is saved normally points to the instruction that raised the exception. Two registers are also set. The exception-data (`ed`) and exception-type (`et`) will be set to reflect the cause of the exception. The exception handler can choose how to deal with the exception. In this chapter the different types of exception are listed, together with their representation, their meaning, and the instructions that may cause them.

24.1 ET_LINK_ERROR

Ad hardware control token was output to a channel end. Alternatively, a channel end was used to transmit data without its destination being set first.

When ET_LINK_ERROR is raised:

- `et` will be set to 1
- `ed` will be set to the resource ID of the channel end which generated the exception.

This exception may be raised by the following instructions:

OUT	*OUTCT*	*OUTT*

24.2 ET_ILLEGAL_PC

The program counter points to a position that could not be accessed, for example, beyond the end of memory, or a non 16-bit aligned memory location. This exception is raised on dispatch of the instruction corresponding to the illegal program counter. The program counter that is saved in `spc` *is* the illegal program counter; the memory address of the instruction that caused the program counter to become illegal is not known. Note that this exception could be caused by, for example, loading a resource with an illegal vector (*SETV*), but that this will not be known until an event happens.

When ET_ILLEGAL_PC is raised:

- `et` will be set to 2
- `ed` will be set to the PC which generated the exception.

This exception may be raised by the following instructions:

BAU	*BLRF*	*BRFT*	*MSYNC*
BLA	*BRBF*	*BRFU*	*TSTART*
BLACP	*BRBT*	*BRU*	
BLAT	*BRBU*	*DRET*	
BLRB	*BRFF*	*KRET*	

24.3 ET_ILLEGAL_INSTRUCTION

A 16-bit/32-bit word was encountered that could not be decoded. This typically indicates that the program counter was incorrect and addresses data memory. Alternatively, a binary is executed that was not compiled for this device.

When ET_ILLEGAL_INSTRUCTION is raised:

- `et` will be set to 3
- `ed` will be set to 0.

This exception may be raised by the following instructions:

DENTSP	*DRESTSP*	*DVLD*
DGETREG	*DRET*	*DVST*

24.4 ET_ILLEGAL_RESOURCE

A resource operation was performed and failed because either the resource identifier supplied was not a valid resource, it was not allocated, or the operation was not legal on that resource.

When ET_ILLEGAL_RESOURCE is raised:

- `et` will be set to 4
- `ed` will be set to the resource identifier passed to the instruction.

This exception may be raised by the following instructions:

CHKCT	*IN*	*PEEK*	*TESTCT*
CLRPT	*INCT*	*SETC*	*TESTLCL*
EDU	*INPW*	*SETCLK*	*TESTWCT*
EEF	*INSHR*	*SETD*	*TINITCP*
EET	*INT*	*SETEV*	*TINITDP*
EEU	*MJOIN*	*SETN*	*TINITLR*
ENDIN	*MSYNC*	*SETPSC*	*TINITPC*
FREER	*OUT*	*SETPT*	*TINITSP*
GETD	*OUTCT*	*SETRDY*	*TSETMR*
GETN	*OUTPW*	*SETTW*	*TSETR*
GETST	*OUTSHR*	*SETV*	*TSTART*
GETTS	*OUTT*	*SYNCR*	

24.5 ET_LOAD_STORE

A memory operation was performed that was not properly aligned. This could be a word load or word store to an address where the least significant `log2 Bpw` bits were not zero, or access to a 16-bit number using LD16S or ST16 where the least significant bit of the address was one. Many load and store operations multiply their operand by `Bpw` in order to increase the density of the encoding; even though this part of the address is guaranteed to be aligned, it is possible for one of `sp`, `cp`, or `dp` to be unaligned, causing any subsequent load or store which uses them to fail.

When ET_LOAD_STORE is raised:

- `et` will be set to 5
- `ed` will be set to the load or store address which generated the exception.

This exception may be raised by the following instructions:

BLACP	*LDSED*	*STDSP*	*VLDD*
BLAT	*LDSPC*	*STET*	*VLDR*
DUALENTSP	*LDSSR*	*STSED*	*VLMACC*
DVLD	*LDW*	*STSPC*	*VLMACCR*
DVST	*LDWCP*	*STSSR*	*VLMACCR1*
ENTSP	*LDWCPL*	*STW*	*VLMUL*
KENTSP	*LDWDP*	*STWDP*	*VLSAT*
KRESTSP	*LDWSP*	*STWSP*	*VLSUB*
LD8U	*PREFETCH*	*VLADD*	*VSTC*
LD16S	*RETSP*	*VLADDD*	*VSTD*
LDD	*ST8*	*VLADSB*	*VSTR*
LDDSP	*ST16*	*VLASHR*	*VSTRPV*
LDET	*STD*	*VLDC*	

24.6 ET_ILLEGAL_PS

Access to a non existent processor status register was requested by either GETPS or SETPS.

When ET_ILLEGAL_PS is raised:

- `et` will be set to 6
- `ed` will be set to the processor status register identifier.

This exception may be raised by the following instructions:

GETPS	*SETPS*

24.7 ET_ARITHMETIC

Signals an arithmetic error, for example a division by 0 or an overflow that was detected.

When ET_ARITHMETIC is raised:

- `et` will be set to 7
- `ed` will be set to 0.

This exception may be raised by the following instructions:

DIVS	*LDIVU*	*VCMI*	*VFTFF*
DIVU	*REMS*	*VCMR*	*VFTTB*
FENAN	*REMU*	*VDEPTH16*	*VFTTF*
FMANT	*VCMCI*	*VDEPTH8*	*VLADSB*
FSEXP	*VCMCR*	*VFTFB*	

24.8 ET_ECALL

An ECALL instruction was executed, and the associated condition caused an exception. Indicates that the application program raised an exception, for example to signal array bound errors or a failed assertion.

When ET_ECALL is raised:

- `et` will be set to 8
- `ed` will be set to 0.

This exception may be raised by the following instructions:

ECALLF	*ECALLT*	*ELATE*

24.9 ET_RESOURCE_DEP

Resources are owned and used by a single thread. If multiple threads attempt to access the same resource within 4 cycles of each other, a Resource Dependency exception will be raised.

When ET_RESOURCE_DEP is raised:

- `et` will be set to 9
- `ed` will be set to the resource identifier supplied by the instruction.

This exception may be raised by the following instructions:

CHKCT	*IN*	*SETC*	*TESTLCL*
CLRPT	*INCT*	*SETCLK*	*TESTWCT*
EDU	*INPW*	*SETD*	*TINITCP*
EEF	*INSHR*	*SETEV*	*TINITDP*
EET	*INT*	*SETN*	*TINITLR*
EEU	*MJOIN*	*SETPSC*	*TINITPC*
ENDIN	*MSYNC*	*SETPT*	*TINITSP*
FREER	*OUT*	*SETRDY*	*TSETMR*
GETD	*OUTCT*	*SETTW*	*TSETR*
GETN	*OUTPW*	*SETV*	*TSTART*
GETST	*OUTSHR*	*SYNCR*	
GETTS	*OUTT*	*TESTCT*	

24.10 ET_KCALL

Indicates that the KCALL or KCALLI instruction was executed.

When ET_KCALL is raised:

- `et` will be set to 15
- `ed` will be set to the kernel call operand.

This exception may be raised by the following instructions:

KCALL

24.11 ET_IOLANE

This value is ORed in with any of the previous exception types to indicate that the exception took place in the resource lane.

When ET_IOLANE is raised:

- `et` will be set to 16
- N/A

This exception is not related to a specific instruction

25 XCore XS3 Lanes

When executing in dual-issue mode, instructions are executed in `lanes`. Some instructions can only be executed in a specific lane, other instructions can execute in one of multiple lanes, and yet other instructions required multiple lanes for execution. In this chapter the different classes of instructions are explained, together with a list of instructions for each.

25.1 MEMORY_LANE

In dual issue mode, these instructions can only be executed in the memory lane, indicated by **M**.

Instructions:

BAU(16)	*KENTSP(16)*	*STSPC(16)*	*VLADD(16)*
BLA(16)	*LD8U(16)*	*STSSR(16)*	*VLADDD(16)*
BLACP(16)	*LD16S(16)*	*STWI(16)*	*VLADSB(16)*
BLAT(16)	*LDET(16)*	*STWDP(16)*	*VLDC(16)*
BLRB(16)	*LDSED(16)*	*STWSP(16)*	*VLDD(16)*
BLRF(16)	*LDSPC(16)*	*VCLRDR(16)*	*VLDR(16)*
BRBF(16)	*LDSSR(16)*	*VADDDR(16)*	*VLMACC(16)*
BRBT(16)	*LDW(16)*	*VCMCI(16)*	*VLMACCR(16)*
BRBU(16)	*LDWI(16)*	*VCMCR(16)*	*VLMACCR1(16)*
BRFF(16)	*LDWCP(16)*	*VCMI(16)*	*VLMUL(16)*
BRFT(16)	*LDWCPL(16)*	*VCMR(16)*	*VLSAT(16)*
BRFU(16)	*LDWDP(16)*	*VDEPTH1(16)*	*VLSUB(16)*
BRU(16)	*LDWSP(16)*	*VDEPTH16(16)*	*VPOS(16)*
DGETREG(16)	*PREFETCH(16)*	*VDEPTH8(16)*	*VSETC(16)*
DUALENTSP(16)	*RETSP(16)*	*VEQCR(16)*	*VSIGN(16)*
ENTSP(16)	*SETCP(16)*	*VEQDR(16)*	*VSTC(16)*
FENAN(16)	*SETDP(16)*	*VFTFB(16)*	*VSTD(16)*
FLUSH(16)	*SETKEP(16)*	*VFTFF(16)*	*VSTR(16)*
INVALIDATE(16)	*SETSP(16)*	*VFTTB(16)*	
KCALL(16)	*STET(16)*	*VFTTF(16)*	
KCALLI(16)	*STSED(16)*	*VGETC(16)*	

25.2 RESOURCE_LANE

In dual issue mode, these instructions can only be executed in the resource lane, indicated by **R**.

Instructions:

CHKCT(16)	*FREET(16)*	*OUTCT(16)*	*SETV(16)*
CHKCTI(16)	*GETR(16)*	*OUTCTI(16)*	*SSYNC(16)*
CLRE(16)	*GETST(16)*	*OUTSHR(16)*	*SYNCR(16)*
CLRPT(16)	*GETTS(16)*	*OUTT(16)*	*TESTCT(16)*
CLRSR(16)	*IN(16)*	*PEEK(16)*	*TESTWCT(16)*
EDU(16)	*INCT(16)*	*SETCI(16)*	*TSTART(16)*
EEF(16)	*INSHR(16)*	*SETD(16)*	*WAITEF(16)*
EET(16)	*INT(16)*	*SETEV(16)*	*WAITET(16)*
EEU(16)	*MJOIN(16)*	*SETPSC(16)*	*WAITEU(16)*
ENDIN(16)	*MSYNC(16)*	*SETPT(16)*	
FREER(16)	*OUT(16)*	*SETSR(16)*	

25.3 MEMORY_OR_RESOURCE_LANE

In dual issue mode, these instructions can be executed in either lane, indicated by **M or R**.

Instructions:

ADD(16)	*EQ(16)*	*LDAPF(16)*	*OR(16)*
ADDI(16)	*EQI(16)*	*LDAWCP(16)*	*SEXT(16)*
AND(16)	*EXTDP(16)*	*LDAWDP(16)*	*SEXTI(16)*
ANDNOT(16)	*EXTSP(16)*	*LDAWSP(16)*	*SHL(16)*
BITREV(16)	*GETED(16)*	*LDC(16)*	*SHLI(16)*
BYTEREV(16)	*GETET(16)*	*LSS(16)*	*SHR(16)*
CLS(16)	*GETID(16)*	*LSU(16)*	*SHRI(16)*
CLZ(16)	*GETKEP(16)*	*MKMSK(16)*	*SUB(16)*
DCALL(16)	*GETKSP(16)*	*MKMSKI(16)*	*SUBI(16)*
ECALLF(16)	*GETSR(16)*	*NEG(16)*	*ZEXT(16)*
ECALLT(16)	*GETTIME(16)*	*NOP(16)*	*ZEXTI(16)*
ELATE(16)	*LDAPB(16)*	*NOT(16)*	

25.4 MEMORY_AND_RESOURCE_LANE

In dual issue mode, these instructions are executed in both lanes simultaneously, indicated by **M and R**.

Instructions:

- *ASHR(32)*
- *ASHRI(32)*
- *BLACP(32)*
- *BLAT(32)*
- *BLRB(32)*
- *BLRF(32)*
- *BRBF(32)*
- *BRBT(32)*
- *BRBU(32)*
- *BRFF(32)*
- *BRFT(32)*
- *BRFU(32)*
- *CLRSR(32)*
- *CRC8(32)*
- *CRC(32)*
- *CRC32_INC(32)*
- *CRCN(32)*
- *DENTSP(32)*
- *DIVS(32)*
- *DIVU(32)*
- *DRESTSP(32)*
- *DRET(32)*
- *DUALENTSP(32)*
- *DVLD(32)*
- *DVST(32)*
- *ENTSP(32)*
- *EXTDP(32)*
- *EXTSP(32)*
- *FADD(32)*
- *FEQ(32)*
- *FGT(32)*
- *FLT(32)*
- *FMACC(32)*
- *FMAKE(32)*
- *FMANT(32)*
- *FMUL(32)*
- *FSEXP(32)*
- *FSPEC(32)*
- *FSUB(32)*
- *FUN(32)*
- *GETD(32)*
- *GETN(32)*
- *GETPS(32)*
- *GETSR(32)*
- *INPW(32)*
- *KCALLI(32)*
- *KENTSP(32)*
- *KRESTSP(32)*
- *KRET(32)*
- *LADD(32)*
- *LDA16B(32)*
- *LDA16F(32)*
- *LDAPB(32)*
- *LDAPF(32)*
- *LDAWB(32)*
- *LDAWBI(32)*
- *LDAWCP(32)*
- *LDAWDP(32)*
- *LDAWF(32)*
- *LDAWFI(32)*
- *LDAWSP(32)*
- *LDC(32)*
- *LDD(32)*
- *LDDI(32)*
- *LDDSP(32)*
- *LDIVU(32)*
- *LDWCP(32)*
- *LDWCPL(32)*
- *LDWDP(32)*
- *LDWSP(32)*
- *LEXTRACT(32)*
- *LINSERT(32)*
- *LMUL(32)*
- *LSUB(32)*
- *MACCS(32)*
- *MACCU(32)*
- *MUL(32)*
- *OUTPW(32)*
- *OUTPWI(32)*
- *REMS(32)*
- *REMU(32)*
- *RETSP(32)*
- *LSATS(32)*
- *SETC(32)*
- *SETCI(32)*
- *SETCLK(32)*
- *SETN(32)*
- *SETPS(32)*
- *SETRDY(32)*
- *SETSR(32)*
- *SETTW(32)*
- *ST8(32)*
- *ST16(32)*
- *STD(32)*
- *STDI(32)*
- *STDSP(32)*
- *STW(32)*
- *STWDP(32)*
- *STWSP(32)*
- *TESTLCL(32)*
- *TINITCP(32)*
- *TINITDP(32)*
- *TINITLR(32)*
- *TINITPC(32)*
- *TINITSP(32)*
- *TSETMR(32)*
- *TSETR(32)*
- *UNZIP(32)*
- *VLASHR(32)*
- *VSTRPV(32)*
- *XOR(32)*
- *XOR4(32)*
- *ZIP(32)*

Copyright © 2024, All Rights Reserved.

Xmos Ltd. is the owner or licensee of this design, code, or Information (collectively, the “Information”) and is providing it to you “AS IS” with no warranty of any kind, express or implied and shall have no liability in relation to its use. Xmos Ltd. makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

XMOS, xCore, xcore.ai, and the XMOS logo are registered trademarks of XMOS Ltd in the United Kingdom and other countries and may not be used without written permission. Company and product names mentioned in this document are the trademarks or registered trademarks of their respective owners.