The XMOS XS1 Architecture

David May
# Contents

1 Background 1

2 Interconnect 1
   2.1 XMOS Link Ports 3
   2.2 Serial XMOS Link 3
   2.3 Fast XMOS Link 4

3 Concurrent Threads 5

4 The XCore Instruction Set 6

5 Instruction Issue and Execution 8
   5.1 Scheduler Implementation 9

6 Instruction Set Notation and Definitions 11
   6.1 Instruction Prefixes 11

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 18

12 Communication 21

13 Locks 24

14 Timers and Clocks 24

15 Ports, Input and Output 26
   15.1 Input and Output 26
   15.2 Port Configuration 27
   15.3 Configuring Ready and Clock Signals 29
   15.4 NOREADY mode 29
   15.5 HANDSHAKEN mode 29
   15.6 STROBED mode 30
   15.7 The Port Timer 30
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.8</td>
<td>Conditions</td>
<td>31</td>
</tr>
<tr>
<td>15.9</td>
<td>Synchronised Transfers</td>
<td>31</td>
</tr>
<tr>
<td>15.10</td>
<td>Buffered Transfers</td>
<td>32</td>
</tr>
<tr>
<td>15.11</td>
<td>Partial Transfers</td>
<td>34</td>
</tr>
<tr>
<td>15.12</td>
<td>Changing Direction</td>
<td>34</td>
</tr>
<tr>
<td>16</td>
<td>Events, Interrupts and Exceptions</td>
<td>35</td>
</tr>
<tr>
<td>17</td>
<td>Initialisation and Debugging</td>
<td>41</td>
</tr>
<tr>
<td>18</td>
<td>Specialised Instructions</td>
<td>42</td>
</tr>
<tr>
<td>19</td>
<td>Instruction Details</td>
<td>45</td>
</tr>
<tr>
<td>19.1</td>
<td>Instructions</td>
<td>45</td>
</tr>
<tr>
<td>19.2</td>
<td>Instruction Format Specification</td>
<td>226</td>
</tr>
<tr>
<td>19.3</td>
<td>Exceptions</td>
<td>247</td>
</tr>
</tbody>
</table>
1 Background

An XS1 combines 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 XMOS Links are provided so that systems can easily be constructed from multiple chips.

The XS1 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.

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 XMOS Links 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.

The four XMOS Links from each XCore connect directly to an on-chip switch which provides non-blocking communication between the XCores. The switch also provides 16 off-chip XMOS Links allowing multiple XS1 chips to be combined in a system. The structure and performance of the XMOS Link connections in a system can be varied to meet the needs of applications.

The links between XCores and switches and the XMOS Links 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 through the XMOS Links 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 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. Otherwise it uses the number of the first non-matching pair of bits to select an outgoing direction. The direction of each XMOS Link is set when the switch is configured and it is possible for several XMOS Links to share the same direction thereby providing several independent routes between the same 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 Interconnect

2.1 XMOS Link Ports

The ports used for inter-chip XMOS Link communication use a transition-based non return-to-zero signalling scheme. Bits are sent at a rate derived from the XS1 clock; this rate can be programmed to meet applications requirements.

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

2.2 Serial XMOS Link

The serial XMOS Link uses two data wires in each direction. A transition on one wire represents a one bit and a transition on the other wire 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 LRESET token is used to cause the destination link to reset and the CRESET is used to reset the issued credit to 0.

<table>
<thead>
<tr>
<th>token</th>
<th>use</th>
</tr>
</thead>
<tbody>
<tr>
<td>224</td>
<td>CREDIT8</td>
</tr>
<tr>
<td>225</td>
<td>CREDIT64</td>
</tr>
<tr>
<td>226</td>
<td>LRESET</td>
</tr>
<tr>
<td>227</td>
<td>CRESET</td>
</tr>
</tbody>
</table>
2.3 Fast XMOS Link

The fast XMOS Link 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:

<table>
<thead>
<tr>
<th>symbol</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001</td>
<td>value 00</td>
</tr>
<tr>
<td>00010</td>
<td>value 01</td>
</tr>
<tr>
<td>00100</td>
<td>value 10</td>
</tr>
<tr>
<td>01000</td>
<td>value 11</td>
</tr>
<tr>
<td>10000</td>
<td>escape</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>token</th>
<th>use</th>
</tr>
</thead>
<tbody>
<tr>
<td>v v v v</td>
<td>256 data tokens</td>
</tr>
<tr>
<td>e v v v</td>
<td>64 control tokens 192-255</td>
</tr>
<tr>
<td>v e v v</td>
<td>64 control tokens 128-191</td>
</tr>
<tr>
<td>v v e v</td>
<td>64 control tokens 64-127</td>
</tr>
<tr>
<td>v v v e</td>
<td>64 control tokens 0-63</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>token</th>
<th>use</th>
</tr>
</thead>
<tbody>
<tr>
<td>e e v v</td>
<td>END tokens</td>
</tr>
<tr>
<td>v v e e</td>
<td>PAUSE tokens</td>
</tr>
<tr>
<td>e v v e</td>
<td>NOP (return to zero) tokens</td>
</tr>
<tr>
<td>e 11 11 v</td>
<td>NOP (return to zero) tokens</td>
</tr>
<tr>
<td>e 00 e 00</td>
<td>CREDIT8</td>
</tr>
<tr>
<td>e 01 e 01</td>
<td>CREDIT64</td>
</tr>
<tr>
<td>e 10 e 10</td>
<td>LRESET</td>
</tr>
<tr>
<td>e 11 e 11</td>
<td>CRESET</td>
</tr>
</tbody>
</table>

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

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.

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 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 \( r_0 \ldots r_{11} \) are used by instructions which perform arithmetic and logical operations, access data structures, and call subroutines.

The access registers are:

<table>
<thead>
<tr>
<th>register</th>
<th>number</th>
<th>use</th>
</tr>
</thead>
<tbody>
<tr>
<td>cp</td>
<td>12</td>
<td>constant pool pointer</td>
</tr>
<tr>
<td>dp</td>
<td>13</td>
<td>data pointer</td>
</tr>
<tr>
<td>sp</td>
<td>14</td>
<td>stack pointer</td>
</tr>
<tr>
<td>lr</td>
<td>15</td>
<td>link register</td>
</tr>
</tbody>
</table>

The control registers are:

<table>
<thead>
<tr>
<th>register</th>
<th>number</th>
<th>use</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>16</td>
<td>program counter</td>
</tr>
<tr>
<td>sr</td>
<td>17</td>
<td>status register</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>register</th>
<th>number</th>
<th>use</th>
</tr>
</thead>
<tbody>
<tr>
<td>spc</td>
<td>18</td>
<td>saved pc</td>
</tr>
<tr>
<td>ssr</td>
<td>19</td>
<td>saved status</td>
</tr>
<tr>
<td>et</td>
<td>20</td>
<td>exception type</td>
</tr>
<tr>
<td>ed</td>
<td>21</td>
<td>exception data</td>
</tr>
<tr>
<td>sed</td>
<td>22</td>
<td>saved exception data</td>
</tr>
<tr>
<td>kep</td>
<td>23</td>
<td>kernel entry pointer</td>
</tr>
<tr>
<td>ksp</td>
<td>24</td>
<td>kernel stack pointer</td>
</tr>
</tbody>
</table>
The status register \( sr \) contains the following information:

<table>
<thead>
<tr>
<th>bit</th>
<th>use</th>
</tr>
</thead>
<tbody>
<tr>
<td>eeble</td>
<td>event enable</td>
</tr>
<tr>
<td>ieble</td>
<td>interrupt enable</td>
</tr>
<tr>
<td>inenb</td>
<td>thread is enabling events</td>
</tr>
<tr>
<td>inint</td>
<td>thread is in interrupt mode</td>
</tr>
<tr>
<td>ink</td>
<td>thread is in kernel mode</td>
</tr>
<tr>
<td>sink</td>
<td>saved ( ink )</td>
</tr>
<tr>
<td>waiting</td>
<td>thread waiting to execute current instruction</td>
</tr>
<tr>
<td>fast</td>
<td>thread enabled for fast input-output</td>
</tr>
</tbody>
</table>

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.

Typically over 80% of instructions executed are 16-bit, so that the XS1 processors fetch two instructions every cycle. As typically less than 30% of instructions require a memory access, each processor can run at full speed using a unified memory system.
5 Instruction Issue and Execution

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 \( \frac{1}{n} \) 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 taken by other threads. Further, the time taken to re-start a waiting thread is always at most one thread cycle.

The set of \( n \) threads can therefore be thought of as a set of virtual processors each with clock rate at least \( \frac{1}{n} \) 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 \( \frac{1}{p} \).

Each thread has a 64-bit instruction buffer which is able to hold four short instructions or two 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).
- 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. A ready request to a thread must be received and an instruction issued rapidly in order to support a high rate of input and output.

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 re-started, 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.
6 Instruction Set Notation and Definitions

In the following description

$Bpw$ 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

$r0 \ldots r11$ represent specific operand registers

$x$ (a single small letter) represents one of $r0 \ldots r11$

$X$ (a single large letter) represents one of $r0 \ldots r11$, $sp$, $dp$, $cp$ or $lr$

$u_s$ is a small unsigned source operand in the range $0 \ldots 11$

$bitp$ is one of $bpw$, $1$, $2$, $3$, $4$, $5$, $6$, $7$, $8$, $16$, $24$, $32$ encoded as a $u_s$

$u_{16}$ is a 16-bit source operand in the range $0 \ldots 65535$

$u_{20}$ is a 20-bit source operand in the range $0 \ldots 1048575$ which

Some useful functions are

$zext(x, n) = x \land (2^n - 1)$ zero extend

$sext(x, n) = -(2^{n-1} \land x) \lor x$ sign extend

6.1 Instruction Prefixes

If the most significant 10 bits of a $u_{16}$ or $u_{20}$ 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

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

- **LDWSP**: $D \leftarrow \text{mem}[\text{sp} + u_{16} \times \text{Bpw}]$ — load word from stack
- **STWSP**: $\text{mem}[\text{sp} + u_{16} \times \text{Bpw}] \leftarrow S$ — store word to stack
- **LDAWSP**: $D \leftarrow \text{sp} + u_{16} \times \text{Bpw}$ — load address of word in stack

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

- **LDWDP**: $D \leftarrow \text{mem}[\text{dp} + u_{16} \times \text{Bpw}]$ — load word from data
- **STWDP**: $\text{mem}[\text{dp} + u_{16} \times \text{Bpw}] \leftarrow S$ — store word to data
- **LDAWDP**: $D \leftarrow \text{dp} + u_{16} \times \text{Bpw}$ — 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**: $D \leftarrow u_{16}$ — load constant
- **LDWCP**: $D \leftarrow \text{mem}[\text{cp} + u_{16} \times \text{Bpw}]$ — load word from constant pool
- **LDAWCP**: $r_{11} \leftarrow \text{cp} + u_{16} \times \text{Bpw}$ — load word address in constant pool
- **LDWCPL**: $r_{11} \leftarrow \text{mem}[\text{cp} + u_{20} \times \text{Bpw}]$ — load word address in constant pool long
- **LDAPF**: $r_{11} \leftarrow \text{pc} + u_{20} \times 2$ — load address in program forward
- **LDAPB**: $r_{11} \leftarrow \text{pc} - u_{20} \times 2$ — 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**: $d \leftarrow \text{mem}[b + u_{8} \times \text{Bpw}]$ — load word
- **STWI**: $\text{mem}[b + u_{8} \times \text{Bpw}] \leftarrow s$ — store word
- **LDAWF**: $d \leftarrow b + u_{8} \times \text{Bpw}$ — load address of word forward
- **LDAWB**: $d \leftarrow b - u_{8} \times \text{Bpw}$ — load address of word backward
- **LDW**: $d \leftarrow \text{mem}[b + i \times \text{Bpw}]$ — load word
- **STW**: $\text{mem}[b + i \times \text{Bpw}] \leftarrow s$ — store word
- **LDAWF**: $d \leftarrow b + i \times \text{Bpw}$ — load address of word forward
- **LDAWB**: $d \leftarrow b - i \times \text{Bpw}$ — load address of word backward
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 32-bit value.

$$LD16S\quad d \leftarrow \text{sext}(mem[b + i \times 2], 16)$$ load 16-bit signed item

$$ST16\quad \text{mem}[b + i \times 2] \leftarrow s$$ store 16-bit item

$$LDA16F\quad d \leftarrow b + i \times 2$$ load address of 16-bit item forward

$$LDA16B\quad d \leftarrow b - i \times 2$$ 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 32-bit value.

$$LD8U\quad d \leftarrow \text{zext}(mem[b + i], 8)$$ load byte unsigned

$$ST8\quad \text{mem}[b + i] \leftarrow s$$ 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.

$$MKMSK\quad d \leftarrow 2^s - 1$$ make mask

$$MKMSKI\quad d \leftarrow 2^{\text{bitp}} - 1$$ make mask immediate

$$SEXT\quad d \leftarrow \text{sext}(d, s)$$ sign extend

$$SEXTI\quad d \leftarrow \text{sext}(d, \text{bitp})$$ sign extend immediate

$$ZEXT\quad d \leftarrow \text{zext}(d, s)$$ zero extend

$$ZEXTI\quad d \leftarrow \text{zext}(d, \text{bitp})$$ zero extend immediate

$$\text{ANDNOT}\quad d \leftarrow d \& \neg s$$ 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.
8 Expression Evaluation

ADDI \( d \leftarrow l + u_s \) 
add immediate

ADD \( d \leftarrow l + r \) 
add

SUBI \( d \leftarrow l - u_s \) 
subtract immediate

SUB \( d \leftarrow l - r \) 
subtract

NEG \( d \leftarrow -s \) 
negate

EQI \( d \leftarrow l = u_s \) 
equal immediate

EQ \( d \leftarrow l = r \) 
equal

LSU \( d \leftarrow l < r \) 
less than unsigned

LSS \( d \leftarrow l <_{sgn} r \) 
less than signed

AND \( d \leftarrow l \& r \) 
and

OR \( d \leftarrow l \lor r \) 
or

XOR \( d \leftarrow l \oplus r \) 
exclusive or

NOT \( d \leftarrow (-1) \oplus s \) 
not

SHLI \( d \leftarrow l << \text{bitp} \) 
logical shift left immediate

SHL \( d \leftarrow l << r \) 
logical shift left

SHRI \( d \leftarrow l >> \text{bitp} \) 
logical shift right immediate

SHR \( d \leftarrow l >> r \) 
logical shift right

ASHRI \( d \leftarrow l >>_{sgn} \text{bitp} \) 
arithmetic shift right immediate

ASHR \( d \leftarrow l >>_{sgn} r \) 
arithmetic shift right

MUL \( d \leftarrow l \times r \) 
multiply

DIVU \( d \leftarrow l \div r \) 
divide unsigned

DIVS \( d \leftarrow l \div_{sgn} r \) 
divide signed

REMU \( d \leftarrow l \mod r \) 
remainder unsigned

REMS \( d \leftarrow l \mod_{sgn} r \) 
remainder signed

BITREV \( d : \forall_{ix} d[\text{bit } ix] = s[\text{bit } bpw - ix - 1] \) 
bit reverse

BYTEREV \( d : \forall_{ix} d[\text{byte } ix] = s[\text{byte } Bpw - ix - 1] \) 
byte reverse

CLZ \( d : \text{first } d : s[\text{bit } bpw - d] = 1 \) 
count leading zeros
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 \[ \text{if } c \text{ then } pc \leftarrow pc + u_{16} \times 2 \] branch relative forward true
BRFF \[ \text{if } \neg c \text{ then } pc \leftarrow pc + u_{16} \times 2 \] branch relative forward false
BRBT \[ \text{if } c \text{ then } pc \leftarrow pc - u_{16} \times 2 \] branch relative backward true
BRBF \[ \text{if } \neg c \text{ then } pc \leftarrow pc - u_{16} \times 2 \] branch relative backward false

BRFU \[ pc \leftarrow pc + u_{16} \times 2 \] branch relative forward unconditional
BRBU \[ pc \leftarrow pc - u_{16} \times 2 \] branch relative backward unconditional
BRU \[ pc \leftarrow pc + s \times 2 \] branch relative unconditional (via register)

BAU \[ pc \leftarrow s \] branch absolute unconditional (via register)

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 \[ lr \leftarrow pc; \]
\[ pc \leftarrow pc + u_{20} \times 2 \]
branch and link relative forward

BLRB \[ lr \leftarrow pc; \]
\[ pc \leftarrow pc - u_{20} \times 2 \]
branch and link relative backward

BLACP \[ lr \leftarrow pc; \]
\[ pc \leftarrow mem[cp + u_{20} \times Bpw] \]
branch and link absolute via constant pool

BLAT \[ lr \leftarrow pc; \]
\[ pc \leftarrow mem[r11 + u_{16} \times Bpw] \]
branch and link absolute via table

BLA \[ lr \leftarrow pc; \]
\[ pc \leftarrow s \]
branch and link absolute (via register)

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.

\[
\begin{align*}
\text{EXTSP} & \quad sp \leftarrow sp - u_{16} \times Bpw & \text{extend stack} \\
\text{EXTDP} & \quad dp \leftarrow dp - u_{16} \times Bpw & \text{extend data} \\
\text{ENTSP} & \quad \text{if } u_{16} > 0 \quad \{\text{mem}[sp] \leftarrow lr; sp \leftarrow sp - u_{16} \times Bpw\} & \text{entry and extend stack} \\
\text{RETSP} & \quad \text{if } u_{16} > 0 \text{ then} \quad \{sp \leftarrow sp + u_{16} \times Bpw; lr \leftarrow \text{mem}[sp]\}; \text{ and return} & \text{contract stack} \\
\end{align*}
\]

Notice that the stack and data area can be contracted using the LDAWSP and LDAWD DP 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.

\[
\begin{align*}
\text{SETSP} & \quad sp \leftarrow s & \text{set stack pointer} \\
\text{SETDP} & \quad dp \leftarrow s & \text{set data pointer} \\
\text{SETCP} & \quad cp \leftarrow s & \text{set pool pointer} \\
\end{align*}
\]

\section{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.

\[
\begin{align*}
\text{GETR} & \quad r \leftarrow \text{first res \in setof}(us) : \neg\text{inuse}_{\text{res}}; \quad \text{get resource} \\
& \quad \text{inuse}_r \leftarrow \text{true} \\
\text{FREER} & \quad \text{inuse}_r \leftarrow \text{false} & \text{free resource} \\
\end{align*}
\]

In the above \text{setof}(r) returns the set corresponding to the source operand of \(r\).
The resources are:

<table>
<thead>
<tr>
<th>resource name</th>
<th>set</th>
<th>use</th>
</tr>
</thead>
<tbody>
<tr>
<td>THREAD</td>
<td>threads</td>
<td>concurrent execution</td>
</tr>
<tr>
<td>SYNC</td>
<td>synchronisers</td>
<td>thread synchronisation</td>
</tr>
<tr>
<td>CHANEND</td>
<td>channel ends</td>
<td>thread communication</td>
</tr>
<tr>
<td>TIMER</td>
<td>timers</td>
<td>timing</td>
</tr>
<tr>
<td>LOCK</td>
<td>locks</td>
<td>mutual exclusion</td>
</tr>
</tbody>
</table>

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

\[
\text{SETC } \text{control}_r \leftarrow u_{16} \quad \text{set resource control}
\]

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:

<table>
<thead>
<tr>
<th>mode</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFF</td>
<td>resource off</td>
</tr>
<tr>
<td>ON</td>
<td>resource on</td>
</tr>
<tr>
<td>START</td>
<td>resource active</td>
</tr>
<tr>
<td>STOP</td>
<td>resource inactive</td>
</tr>
<tr>
<td>EVENT</td>
<td>port will cause events</td>
</tr>
<tr>
<td>INTERRUPT</td>
<td>port will raise interrupts</td>
</tr>
</tbody>
</table>
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, it is in a paused state and its access registers can be initialised using the following instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TINITPC</td>
<td>$pc_t ← s$ set thread pc</td>
</tr>
<tr>
<td>TINITSP</td>
<td>$sp_t ← s$ set thread stack</td>
</tr>
<tr>
<td>TINITDP</td>
<td>$dp_t ← s$ set thread data</td>
</tr>
<tr>
<td>TINITCP</td>
<td>$cp_t ← s$ set thread pool</td>
</tr>
<tr>
<td>TINITLR</td>
<td>$lr_t ← s$ set thread link</td>
</tr>
</tbody>
</table>
These instructions can only be used when the thread is paused. The TINITLR instruction is intended primarily to support debugging.

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.

\[
\text{TSETR} \quad d_t \leftarrow s \\
\text{TSETMR} \quad d_{\text{mstr}(\text{tid})} \leftarrow s
\]

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.

\[
\text{GETST} \quad d \leftarrow \text{first thread} \in \text{threads} : \neg \text{inuse}_{\text{thread}}; \quad \text{get synchronised thread} \\
\quad \text{inuse}_d \leftarrow \text{true}; \\
\quad \text{spaused} \leftarrow \text{spaused} \cup \{d\}; \\
\quad \text{slaves}_s \leftarrow \text{slaves}_s \cup \{d\} \\
\quad \text{mstr}_s \leftarrow \text{tid}
\]

\[
\text{MSYNC} \quad \text{if } (\text{slaves}_s \setminus \text{spaused} = \emptyset) \quad \text{master synchronise} \\
\quad \text{then} \{ \\
\quad \quad \text{spaused} \leftarrow \text{spaused} \setminus \text{slaves}_s \\
\quad \text{else} \{ \\
\quad \quad \text{mpaused} \leftarrow \text{mpaused} \cup \{\text{tid}\}; \\
\quad \quad \text{msyn}_s \leftarrow \text{true} \}
\]

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 unpaued and continue executing from the next instruction. The processor maintains a set of paused master threads \(\text{mpaused}\) and a set of paused slave threads \(\text{spaused}\) from which it derives the set of runnable threads \(\text{run}\):

\[
\text{run} = \{\text{thread} \in \text{threads} : \text{inuse}_{\text{thread}}\} \setminus (\text{spaused} \cup \text{mpaused})
\]

Each synchroniser also maintains a record \(\text{msyn}_s\) of whether its master has reached a synchronisation point.
SSYNC if \( (slaves_{syn(tid)} \setminus spaused = \{ tid \}) \land msyn_{syn(tid)} \) slave synchronise

then {
  if mjoin_{syn(tid)}
  then {
    forall thread \in slaves_{syn(tid)} : inuse_{thread} \leftarrow false;
    mjoin_{syn(tid)} \leftarrow false
  }
  else
    spaused \leftarrow spaused \setminus slaves_{syn(tid)};
    mpaused \leftarrow mpaused \setminus \{ mstr_{syn(tid)} \};
    msyn_{syn(tid)} \leftarrow false
  }
else
  spaused \leftarrow spaused \cup \{ 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 \( (slaves_s \setminus spaused = \emptyset) \) master join

then {
  forall thread \in slaves_s : inuse_{thread} \leftarrow false;
  mjoin_{syn(tid)} \leftarrow false
}
else {
  mpaused \leftarrow mpaused \cup \{ tid \};
  mjoin_s \leftarrow true;
  msyn_s \leftarrow 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 \leftarrow spaused \setminus \{ tid \} start thread

FREET inuse_{tid} \leftarrow false; free thread

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

GETID \( t \leftarrow tid \) 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.

\[
\text{SETD } r_{\text{dest}} \leftarrow s \quad \text{set destination}
\]

\[
\text{GETD } d \leftarrow r_{\text{dest}} \quad \text{get destination}
\]

The identifier of the channel end \(c_1\) is used to initialise the channel end for thread \(c_2\), 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.

\[
\text{SETN } c_{\text{net}} \leftarrow s \quad \text{set network}
\]

\[
\text{GETN } d \leftarrow c_{\text{net}} \quad \text{get network}
\]
The XMOS XS1 Architecture

In the following, \( c \triangleleft s \) represents an output of \( s \) to channel \( c \) and \( c \triangleright d \) represents an input from channel \( c \) to \( d \).

\[
\begin{align*}
\text{OUTT} & \quad c \triangleleft \text{dtoken}(s) & \text{output token} \\
\text{OUTCT} & \quad c \triangleleft \text{ctoken}(s) & \text{output control token} \\
\text{OUTCTI} & \quad c \triangleleft \text{ctoken}(us) & \text{output control token immediate} \\
\text{INT} & \quad \text{if hasctoken}(c) \text{ then } \text{trap} \\
& \quad \text{else } c \triangleright d \\
\text{INCT} & \quad \text{if hasctoken}(c) \text{ then } c \triangleright d \\
& \quad \text{else } \text{trap} \\
\text{CHKCT} & \quad \text{if hasctoken}(c) \land (s = \text{token}(c)) \text{ then } \text{skiptoken}(c) \\
& \quad \text{else } \text{trap} \\
\text{CHKCTI} & \quad \text{if hasctoken}(c) \land (s = \text{token}(c)) \text{ then } \text{skiptoken}(c) \\
& \quad \text{else } \text{trap} \\
\text{OUT} & \quad c \triangleleft s & \text{output data word} \\
\text{IN} & \quad \text{if containsctoken}(c) \text{ then } \text{trap} \\
& \quad \text{else } c \triangleright d \\
\text{TESTCT} & \quad d \leftarrow \text{hasctoken}(c) & \text{test for control token} \\
\text{TESTWCT} & \quad d \leftarrow \text{containsctoken}(c) & \text{test word for control token}
\end{align*}
\]

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 $c_2$ stored in channel end $c_1$. 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 $c_2$ stored in channel end $c_1$ is on the same processor as $c_1$; this makes it possible to optimise communication of large data structures where the two communicating threads are executed by the same processor.

$$\text{TESTLCL } d \gets \text{islocal}(c) \quad \text{test destination local}$$

## 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 and Clocks

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.
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 \texttt{GETR t, TIMER} instruction. It can be configured using the \texttt{SETC} instruction; the only two modes which can be set are UNCOND and AFTER.

<table>
<thead>
<tr>
<th>mode</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNCOND</td>
<td>timer always ready; inputs complete immediately</td>
</tr>
<tr>
<td>AFTER</td>
<td>timer ready when its current time is after its DATA value</td>
</tr>
</tbody>
</table>

In unconditional mode, an \texttt{IN} instruction reads the current value of the timer. In AFTER mode, the \texttt{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 \texttt{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 \texttt{SETCLK} instruction.

\texttt{SETCLK} \hspace{1em} \texttt{clock} \leftarrow \texttt{s} \hspace{1em} set clock source

Each port \( p \) which is to be clocked from a clock \( c \) can be connected to it by executing a \texttt{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 \texttt{SETCLK} \( c, p \) instruction. Alternatively, a clock may use the reference clock as its clock source (by \texttt{SETCLK} \( c, \text{REF} \)) and in this case the clock can be configured to divide the reference frequency using an 8-bit divider. When this is set to 0, the reference 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 \( f/2^n \). The division factor is set using the \texttt{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 \texttt{SETC} \( c, \text{START} \) 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 \texttt{SETC} \( c, \text{STOP} \) 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.

\[
\begin{align*}
\text{OUTSHR} & \quad p \leftarrow s[\text{bits 0 for } \text{trwidth}(p)]; \\
& \quad s \leftarrow s \gg \text{trwidth}(p) \quad \text{output to port and shift} \\
\text{INSHR} & \quad s \leftarrow s \gg \text{trwidth}(p); \\
& \quad p \leftarrow s[\text{bits } (\text{bpw} - \text{trwidth}(p)) \text{ for } \text{trwidth}(p)] \quad \text{input from port}
\end{align*}
\]

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 pull down for 0 bits, are high impedance otherwise
PULLUP | pins pull up for 1 bits, but are high impedance otherwise
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.

\[
\text{SETRDY } \text{ready}_p \leftarrow s \quad \text{set source of port ready input}
\]

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 \( \text{SETC } r, \text{READYPORT} \). This ready port \( r \) can then be associated with a port \( p \) by \( \text{SETRDY } r, p \).

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 timestamping as this allows precise control of response times.

\[
\text{SETPT } \quad \text{porttime}_p \leftarrow s \quad \text{set port time}
\]

\[
\text{CLRPT } \quad \text{clearporttime}(p) \quad \text{clear port time}
\]

\[
\text{GETTS } \quad d \leftarrow \text{timestamp}_p \quad \text{get port timestamp}
\]

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.

<table>
<thead>
<tr>
<th>mode</th>
<th>port ready condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>no condition</td>
</tr>
<tr>
<td>EQ</td>
<td>value on pins equal to port data register value</td>
</tr>
<tr>
<td>NEQ</td>
<td>value on pins not equal to port data register value</td>
</tr>
</tbody>
</table>

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.

\[
\text{SETTW } \text{width}_p \leftarrow s \quad \text{set port transfer width}
\]

For a 32-bit wordlength, 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 \( n \times p \) 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 \( n \times p \) 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 an 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 \( shiftcount_p \leftarrow s \) 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 \( d \leftarrow buffercount_p \) end input

To optimise the transfer of partwords two further instructions are provided:

OUTPW \( shiftcount_p \leftarrow bitp; \) output part word

INPW \( shiftcount_p \leftarrow bitp; \) input part word

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.
Events and interrupts allow timers, ports and channel ends to automatically transfer control to a pre-defined event handler. 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.

\[
\text{SETSR} \quad sr \leftarrow sr \lor \text{u6} \quad \text{set thread state}
\]
\[
\text{CLRSR} \quad sr \leftarrow sr \land \neg \text{u6} \quad \text{clear thread state}
\]
\[
\text{GETSR} \quad r11 \leftarrow sr \land \text{u6} \quad \text{get thread state}
\]

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
- INK    determine if thread is in kernel mode
- SINK   determine if thread was in kernel mode
- WAITING determine if thread is waiting to execute the current instruction
- FAST  determine if thread is in fast mode

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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETV</td>
<td>vector ← s  set event vector</td>
</tr>
<tr>
<td>SETEV</td>
<td>envector ← s set event environment vector</td>
</tr>
<tr>
<td>SETD</td>
<td>data ← s   set resource data</td>
</tr>
<tr>
<td>GETD</td>
<td>d ← data   get resource data</td>
</tr>
<tr>
<td>SETC</td>
<td>cond ← s   set event condition</td>
</tr>
<tr>
<td>EET</td>
<td>enb ← c; thread, ← tid event enable true</td>
</tr>
<tr>
<td>EEF</td>
<td>enb ← ¬c; thread, ← tid event enable false</td>
</tr>
<tr>
<td>EDU</td>
<td>enb ← false; thread, ← tid event disable</td>
</tr>
<tr>
<td>EEU</td>
<td>enb ← true; thread, ← tid event enable</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAITET</td>
<td>if c then eebleid ← true event wait if true</td>
</tr>
<tr>
<td>WAITEF</td>
<td>if ¬c then eebleid ← true event wait if false</td>
</tr>
<tr>
<td>WAITEU</td>
<td>eebleid ← true event wait</td>
</tr>
</tbody>
</table>

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 ← evres;
   pc ← vres;
   sr[bit inenb] ← false;
   sr[bit eeble] ← false;
   sr[bit 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 EEABLE 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.

\[
\begin{align*}
\text{CLRE} & \quad eeb\text{le}_{\text{tid}} \leftarrow \text{false}; \quad \text{disable all events} \\
& \quad i\text{nenb}_{\text{tid}} \leftarrow \text{false}; \quad \text{for thread} \\
& \quad \text{forall } res \\
& \quad \quad \text{if } (\text{thread}_{res} = \text{tid} \land \text{event}_{res}) \text{ then } \text{enb}_{res} \leftarrow \text{false}
\end{align*}
\]

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.

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. This is done using the spc and ssr registers. On an interrupt generated by resource \( r \) the following occurs automatically:

\[
\begin{align*}
& \text{int } \quad \text{spc} \leftarrow \text{pc}; \\
& \quad \text{ssr} \leftarrow \text{sr}; \\
& \quad \text{pc} \leftarrow \text{v}_{res}; \\
& \quad \text{sed} \leftarrow \text{ed}; \\
& \quad \text{ed} \leftarrow \text{ev}_{res} \\
& \quad \text{sr}[\text{bit inint}] \leftarrow \text{true} \\
& \quad \text{sr}[\text{bit ink}] \leftarrow \text{true}; \\
& \quad \text{sr}[\text{bit eeble}] \leftarrow \text{false}; \\
& \quad \text{sr}[\text{bit ieble}] \leftarrow \text{false} \\
& \quad \text{sr}[\text{bit waiting}] \leftarrow \text{false}
\end{align*}
\]
When the handler has completed, execution of the interrupted thread can be performed by a KRET instruction.

```
KRET  pc ← spc;  return from interrupt
       sr ← ssr
       ed ← sed
```

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;
       ssr ← sr;
       et ← traptype;
       sed ← ed;
       ed ← trapdata;
       pc ← kep;
       sr[bit ink] ← true;
       sr[bit eeble] ← false;
       sr[bit ieble] ← false
```

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

```
ECALLT if e then {  error on true
       spc ← pc;
       ssr ← sr;
       et ← error;
       sed ← ed;
       ed ← s;
       pc ← kep;
       sr[bit ink] ← true;
       sr[bit eeble] ← false;
       sr[bit ieble] ← false }
```
ECALLF  if ¬e then { error on false
spc ← pc;
ssr ← sr;
et ← error;
osed ← ed;
ed ← s
pc ← kep;
sr[bit ink] ← true;
sr[bit eeble] ← false;
sr[bit ieble] ← false
}

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.

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  spc ← pc; kernel call immediate
ssr ← sr;
et ← kernelcall
osed ← ed
ed ← u6;
pc ← kep + 64;
sr[bit ink] ← true;
sr[bit ieble] ← false;
sr[bit eeble] ← false

KCALL  spc ← pc; kernel call
ssr ← sr;
ed ← s;
pc ← kep + 64;
sr[bit ink] ← true;
sr[bit ieble] ← false;
sr[bit eeble] ← false

The spc, ssr, et and sed registers can be saved and restored directly to the stack.
In addition, the et and ed registers can be transferred directly to a register.

GETET \( r_{11} \leftarrow et \) get exception type
GETED \( r_{11} \leftarrow ed \) 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.

\[
\text{KENTSP } n \quad \text{mem}[ksp] \leftarrow sp; \quad \text{switch to kernel stack}
\quad sp \leftarrow ksp - n \times Bpw
\]

\[
\text{KRESTSP } n \quad ksp \leftarrow sp + n \times Bpw; \quad \text{switch from kernel stack}
\quad sp \leftarrow \text{mem}[ksp]
\]

A handler can detect whether or not it has been entered from kernel mode using GETSR SINK.

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

SETKEP \( kep \leftarrow r_{11} \) set kernel entry point
GETKSP \( r_{11} \leftarrow ksp \) 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.
17 Initialisation and Debugging

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

<table>
<thead>
<tr>
<th>register</th>
<th>use</th>
</tr>
</thead>
<tbody>
<tr>
<td>dspc</td>
<td>debug save pc</td>
</tr>
<tr>
<td>dsrs</td>
<td>debug save sr</td>
</tr>
<tr>
<td>dssp</td>
<td>debug save sp</td>
</tr>
<tr>
<td>dtype</td>
<td>debug cause</td>
</tr>
<tr>
<td>dtid</td>
<td>thread identifier used to access thread state</td>
</tr>
<tr>
<td>dtreg</td>
<td>register identifier used to access thread state</td>
</tr>
</tbody>
</table>

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

- GETPS: $d \leftarrow state[s]$ get processor state
- SETPS: $state[d] \leftarrow s$ 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: $d \leftarrow dtreg,dtid$ get thread register

The debugging state is entered by either executing a DCALL instruction, or by an external DEBUG event (such as a breakpoint or watchpoint). During debug, only thread 0 executes, 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 to an interrupt:

```
dspc ← pc0;
dssr ← sr0;
pc0 ← debugentry
dtype ← cause
sr0[bit init] ← true
sr0[bit ink] ← true;
sr0[bit eble] ← false;
sr0[bit ieble] ← false
sr0[bit waiting] ← false
```
The DCALL instruction has the same effect:

DCALL  \[ \text{dscp} \leftarrow \text{pc}_0; \]
\[ \text{dssr} \leftarrow \text{sr}_0; \]
\[ \text{pc}_0 \leftarrow \text{debugentry} \]
\[ \text{dtype} \leftarrow \text{dcallcause} \]
\[ \text{sr}_0[\text{bit inint}] \leftarrow \text{true} \]
\[ \text{sr}_0[\text{bit ink}] \leftarrow \text{true}; \]
\[ \text{sr}_0[\text{bit eeble}] \leftarrow \text{false}; \]
\[ \text{sr}_0[\text{bit ieble}] \leftarrow \text{false} \]

DRET  \[ \text{pc}_0 \leftarrow \text{dscp}; \]
\[ \text{sr}_0 \leftarrow \text{dssr}; \]

DENTSP  \[ \text{dssp} \leftarrow \text{sp}; \]
\[ \text{sp} \leftarrow \text{ramend} \]

DRESTSP  \[ \text{sp} \leftarrow \text{dssp} \]

18 Specialised Instructions

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  \[ d \leftarrow l + r + c[\text{bit 0}]; \]
\[ e \leftarrow \text{carry}(l + r + c[\text{bit 0}]) \]

LSUB  \[ d \leftarrow l - r - b[\text{bit 0}]; \]
\[ e \leftarrow \text{borrow}(l - r - b[\text{bit 0}]) \]

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 \((B - 1) \times (B - 1) + (B - 1) + (B - 1) = B^2 - 1\)
where $B = 2^{b pw}$. The two carry-in operands allow the component results of multi-length multiplications to be formed directly without the need for extra addition steps.

\[
\text{LMUL } d \leftarrow ((l \times r) + s + t)[\text{bits } b pw \text{ for } b pw]; \quad \text{long multiply}
\]
\[
e \leftarrow ((l \times r) + s + t)[\text{bits } 0 \text{ for } b pw]
\]

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.

\[
\text{LDIV } d \leftarrow (l \ll bpw + m) \div r; \quad \text{long divide unsigned}
\]
\[
e \leftarrow (l \ll bpw + m) \mod r
\]

The instruction traps if the result cannot be represented as a single word value; this occurs when $l \leq r$. 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 multiply-accumulate instructions perform a double length accumulation of products of single length operands:

\[
\text{MACCU } s \leftarrow ((l \times r) + s \ll bpw + t)[\text{bits } bpw \text{ for } bpw]; \quad \text{long multiply}
\]
\[
t \leftarrow ((l \times r) + t)[\text{bits } 0 \text{ for } bpw] \quad \text{accumulate unsigned}
\]

\[
\text{MACCS } s \leftarrow ((l \times_{sgn} r) + s \ll bpw + t)[\text{bits } bpw \text{ for } bpw]; \quad \text{long multiply}
\]
\[
t \leftarrow ((l \times_{sgn} r) + t)[\text{bits } 0 \text{ for } bpw] \quad \text{accumulate signed}
\]

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.
Cyclic redundancy check is performed using:

**CRC** for \( \text{step} = 0 \) for \( \text{bpw} \)

\[
\text{if } (r[\text{bit 0}] = 1) \text{ redundancy check}
\]
\[
\text{then } r \leftarrow (s[\text{bit step}] : r[\text{bits } (\text{bpw} - 1) \ldots 1]) \oplus p
\]
\[
\text{else } r \leftarrow (s[\text{bit step}] : r[\text{bits } (\text{bpw} - 1) \ldots 1])
\]

**CRC8** for \( \text{step} = 0 \) for 8

\[
\text{if } (r[\text{bit 0}] = 1) \text{ redundancy check}
\]
\[
\text{then } r \leftarrow (s[\text{bit step}] : r[\text{bits } 31 \ldots 1]) \oplus p
\]
\[
\text{else } r \leftarrow (s[\text{bit step}] : r[\text{bits } 31 \ldots 1]);
\]
\[
d \leftarrow s \gg 8
\]

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.
19  Instruction Details

This section details the semantics and encoding of all instructions of the XCore instruction set architecture. The meaning and assembly syntax of each instruction is documented in alphabetical order in Section 19.1. Section 19.2 presents the encoding of each instruction; the information in this chapter is needed for the construction of low-level tools such as assemblers and debuggers. Section 19.3 presents all exceptions, and lists which instructions can trigger each specific exception.

The instructions use the following registers:

- \( r_0 \ldots r_{11} \) operand registers
- \( pc \) program counter. The program counter is pre incremented, that is, it contains the address of the next instruction in the program. All instructions that use an address offset relative to the program counter (such as relative branches, load address relative, etc) use an offset of ‘0’ to address the next instruction.
- \( sr \) status register
- \( sp \) stack pointer
- \( dp \) data pointer
- \( cp \) constant pool pointer
- \( lr \) link register

19.1 Instructions

This section presents the instructions in alphabetical order. Each instruction is presented with 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.
The following notation is used in the description to describe operands and constants:

- \( \text{bitp} \) denotes a bit-position - one of \( b_{pw} \), 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, and 32; these are encoded using numbers 0...11.
- \( b \) register used as a base address.
- \( c \) 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.
- \( u_x \) a small unsigned constant in the range 0...11
- \( u_x \) 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 (\( \mathbb{Z} \)) and, unless otherwise stated, bit patterns found in registers are interpreted \emph{unsigned}. Signed numbers are represented using two’s complement, and if an operand is interpreted as a signed number, this is denoted by a subscript \emph{signed}. In addition to the standard numerical operators following bitwise operators are assumed:

- \( \lor \text{bit} \) Bitwise or.
- \( \land \text{bit} \) Bitwise and.
- \( \oplus \text{bit} \) Bitwise xor.
- \( \neg \text{bit} \) Bitwise complement.

Square brackets are used for two purposes. When preceded with the word \emph{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 \( \text{mem}[x] \) represents word-based access to memory, and the address \( x \) must be word-aligned (that is, the address must be a multiple of \( B_{pw} \)). 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 \emph{opcode} bits of the encoding - the way that the operands are encoded is specified on the corresponding page in the instruction formats section. Each operand in the instruction section maps positionally on an operand in the format section.
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:

\[
\begin{align*}
\text{op1} & \quad \text{d} \quad \text{Operand register, one of r0...r11} \\
\text{op2} & \quad \text{x} \quad \text{Operand register, one of r0...r11} \\
\text{op3} & \quad \text{y} \quad \text{Operand register, one of r0...r11}
\end{align*}
\]

Mnemonic and operands:

\[
\text{ADD } d, x, y
\]

Operation:

\[
d \leftarrow (x + y) \mod 2^{bpw}
\]

Encoding:

\[
3r \quad 0 0 0 1 0 \ldots \ldots 
\]
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:

\[ \begin{align*}
  op1 & \quad d \quad \text{Operand register, one of } r0\ldots r11 \\
  op2 & \quad x \quad \text{Operand register, one of } r0\ldots r11 \\
  op3 & \quad u_s \quad \text{An integer in the range } 0\ldots 11
\end{align*} \]

Mnemonic and operands:

\[ \text{ADDI } d, x, u_s \]

Operation:

\[ d \leftarrow (x + u_s) \mod 2^\text{bpw} \]

Encoding:

\[ \text{2rus } 10010\ldots\ldots\ldots\ldots \]
AND

Bitwise and

Produces the bitwise AND of two words.

The instruction has three operands:

\[
\begin{align*}
\text{op} & \ 1 & d & \text{Operand register, one of } r0...r11 \\
\text{op} & \ 2 & x & \text{Operand register, one of } r0...r11 \\
\text{op} & \ 3 & y & \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{AND} \quad d, x, y
\]

Operation:

\[
d \leftarrow x \land y
\]

Encoding:

\[
3r \quad \begin{array}{cccccccccc}
0 & 0 & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{array}
\]
ANDNOT

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:

\[
\begin{align*}
\text{op1} & \quad d & \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad s & \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{ANDNOT } d, s
\]

Operation:

\[
d \leftarrow d \land \neg \text{bit s}
\]

Encoding:

\[
2r \quad 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0
\]
**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.

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:

\[
\text{ASHR} \ d, x, y
\]

Operation:

\[
d \leftarrow \begin{cases} 
0 < y < \text{bpw}, & x[\text{bpw} - 1] : \ldots : x[\text{bpw} - 1 - y] \\
y = 0, & x \\
y \geq \text{bpw}, & x[\text{bpw} - 1] : \ldots : x[\text{bpw} - 1]
\end{cases}
\]

Encoding:

\[
\begin{array}{c}
\text{I3r} \\
1 1 1 1 1 0 \ldots 0 \ldots 0 \ldots 0 \\
0 0 0 0 1 0 1 1 1 1 1 1 0 1 1 0 0
\end{array}
\]
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.

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:

\[
d \leftarrow \begin{cases} 
0 < bitp < bpw, & x[bpw - 1] : ... : x[bpw - 1 - bitp] \\
bitp = 0, & x \\
bitp \geq bpw, & x[bpw - 1] : ... : x[bpw - 1]
\end{cases}
\]

Encoding:

<table>
<thead>
<tr>
<th>l2rus</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1</td>
</tr>
<tr>
<td>1 0 0 1 0</td>
</tr>
<tr>
<td>1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 0</td>
</tr>
<tr>
<td>1 1 0 0</td>
</tr>
</tbody>
</table>
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:

\[
\text{BAU} \quad s
\]

Operation:

\[ pc \leftarrow s \]

Encoding:

\[
1r \quad 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ \cdots
\]

Conditions that raise an exception:

\[ \text{ET_ILLEGAL_PC} \]  
The address specified was not 16-bit aligned or did not point to a memory location.
BITREVER

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:

\[
\begin{align*}
op_1 & \quad d \quad \text{Operand register, one of } r0...r11 \\
op_2 & \quad s \quad \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{BITREVER } d, s
\]

Operation:

\[
d[bpw - 1...0] \leftarrow s[0] : s[1] : s[2] : ... : s[bpw - 1]
\]

Encoding:

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c}
& 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
\hline
l2r & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0
\end{array}
\]
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:

\[ op1 \quad s \quad \text{Operand register, one of } r0...r11 \]

Mnemonic and operands:

\[ \text{BLA} \quad s \]

Operation:

\[ lr \leftarrow pc \]
\[ pc \leftarrow s \]

Encoding:

\[ 1r \quad 0 0 1 0 0 1 1 1 1 1 1 0 \ldots \ldots \]

Conditions that raise an exception:

\[ \text{ET,ILLEGAL,PC} \quad \text{The address specified was not 16-bit aligned or did not point to a memory location.} \]
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 ($u_{20}$) 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:

$$op1 \quad u_{20} \quad \text{A 20-bit immediate in the range 0...1048575.}$$

If $u_{20} < 1024$, the instruction requires no prefix.

Mnemonic and operands:

\[
\text{BLACP} \quad u_{20}
\]

Operation:

\[
lr \leftarrow pc \\
pc \leftarrow \text{mem}[cp + u_{20} \times Bpw]
\]

Encoding:

\[
\begin{array}{c}
\text{u10} \\
\text{or prefixed for long immediates:}
\end{array}
\]

\[
\begin{array}{c}
111000 \cdots \\
111000 \cdots \\
111000 \cdots \\
\end{array}
\]

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.
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 (u16) 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 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:

\[
\text{op1 \hspace{5mm} u}_{16} \hspace{5mm} \text{A 16-bit immediate in the range 0...65535.} \\
\text{If } u_{16} < 64, \text{ the instruction requires no prefix}
\]

Mnemonic and operands:

\[
\text{BLAT \hspace{5mm} u}_{16}
\]

Operation:

\[
lr \leftarrow pc \\
\text{pc} \leftarrow \text{mem}[r11 + u_{16} \times Bpw]
\]

Encoding:

\[
\text{u6} \hspace{5mm} 011101101\ldots\ldots
\]

or prefixed for long immediates:

\[
\text{lu6} \hspace{5mm} 111100\ldots\ldots
\]

Conditions that raise an exception:

\[
\text{ET.ILLEGAL_PC} \hspace{5mm} \text{Loaded value was not 16-bit aligned or did not point to a memory location (trapped during the next cycle).}
\]

\[
\text{ET.LOAD_STORE} \hspace{5mm} \text{Register r11 points to an unaligned address, or the indexed address does not point to a valid memory address.}
\]
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:

\[ \text{op1 } u_{20} \quad \text{A 20-bit immediate in the range } 0...1048575. \]

If \( u_{20} < 1024 \), the instruction requires no prefix

Mnemonic and operands:

\[ \text{BLRB } u_{20} \]

Operation:

\[ lr \leftarrow pc \]
\[ pc \leftarrow pc - u_{20} \times 2 \]

Encoding:

\[ \text{u10 } \begin{array}{c} 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \end{array} \]

or prefixed for long immediates:

\[ \text{lu10 } \begin{array}{c} 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \end{array} \]

Conditions that raise an exception:

\text{ET_ILLEGAL_PC} \quad \text{The new PC is not pointing to a valid memory location.}
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:

\[ op1 \quad u_{20} \quad \text{A 20-bit immediate in the range } 0...1048575. \]
\[ \text{If } u_{20} < 1024, \text{ the instruction requires no prefix} \]

Mnemonic and operands:

\[
\text{BLRF} \quad u_{20}
\]

Operation:

\[
lr \leftarrow pc \\
p_{c} \leftarrow pc + u_{20} \times 2
\]

Encoding:

\[
u10 \quad 110100\ldots\ldots\ldots\ldots
\]

or prefixed for long immediates:

\[
lu10 \quad 111100\ldots\ldots\ldots\ldots \\
110100\ldots\ldots\ldots\ldots
\]

Conditions that raise an exception:

\text{ET_ILLEGAL_PC} \quad \text{The new PC is not pointing to a valid memory location.}
**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 \( r0...r11 \)

- \( op2 \)  
  \( u_{16} \)  
  A 16-bit immediate in the range 0...65535.
  
  If \( u_{16} < 64 \), the instruction requires no prefix

Mnemonic and operands:

\[
\text{BRBF} \quad c, u_{16}
\]

Operation:

\[
\text{if } c = 0 \text{ then } pc \leftarrow pc - u_{16} \times 2
\]

Encoding:

\[
\text{ru6} \quad 0 \ 1 \ 1 \ 1 \ 1 \ 1 | \quad \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot
\]

or prefixed for long immediates:

\[
\text{lru6} \quad 1 \ 1 \ 1 \ 1 \ 0 \ 0 | \quad \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot
\]

\[
0 \ 1 \ 1 \ 1 \ 1 \ 1 | \quad \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot
\]

Conditions that raise an exception:

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

\[
\begin{align*}
\text{op1} & \quad c & \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad u_{16} & \text{A 16-bit immediate in the range } 0...65535. \\
\notag & \quad \text{If } u_{16} < 64, \text{ the instruction requires no prefix}
\end{align*}
\]

Mnemonic and operands:

\[
\text{BRBT } c, u_{16}
\]

Operation:

\[
\text{if } c \neq 0 \text{ then } pc \leftarrow pc - u_{16} \times 2
\]

Encoding:

\[
\text{ru6} 011101\ldots\ldots
\]

or prefixed for long immediates:

\[
\text{lru6} 111100\ldots\ldots \\
011101\ldots\ldots
\]

Conditions that raise an exception:

\[
\text{ET.ILLEGAL_PC} \quad \text{The new PC is not pointing to a valid memory location.}
\]
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:

\[ \text{op1 } u_{16} \]

A 16-bit immediate in the range 0...65535.

If \( u_{16} < 64 \), the instruction requires no prefix

Mnemonic and operands:

\[
\begin{align*}
\text{BRBU} & \quad u_{16} \\
\end{align*}
\]

Operation:

\[
\text{pc} \leftarrow \text{pc} - u_{16} \times 2
\]

Encoding:

\[
\begin{array}{c}
u_6 \quad 0 \ 1 \ 1 \ 1 \ 0 \ | \ 1 \ 1 \ 0 \ 0 \ | \ \cdots \ \cdots \\
\end{array}
\]

or prefixed for long immediates:

\[
\begin{array}{c}
l_u6 \quad 1 \ 1 \ 1 \ 1 \ 0 \ | \ \cdots \ \cdots \\
0 \ 1 \ 1 \ 1 \ 0 \ | \ 1 \ 1 \ 0 \ 0 \ | \ \cdots \\
\end{array}
\]

Conditions that raise an exception:

\[
\text{ET_ILLEGAL_PC} \quad \text{The new PC is not pointing to a valid memory location.}
\]
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:

\[
\begin{align*}
op1 & \quad c \\
op2 & \quad u_{16}
\end{align*}
\]

Operand register, one of \( r0...r11 \)
A 16-bit immediate in the range 0...65535.
If \( u_{16} < 64 \), the instruction requires no prefix

Mnemonic and operands:

\[ \text{BRFF} \quad c, u_{16} \]

Operation:

\[
\text{if } c = 0 \text{ then } pc \leftarrow pc + u_{16} \times 2
\]

Encoding:

\[
\begin{align*}
ru6 & \quad 0 1 1 1 1 0 \ldots \\
\end{align*}
\]

or prefixed for long immediates:

\[
\begin{align*}
lru6 & \quad 1 1 1 1 0 0 \ldots \\
0 1 1 1 1 0 \ldots \\
\end{align*}
\]

Conditions that raise an exception:

\[ \text{ET\_ILLEGAL\_PC} \quad \text{The new PC is not pointing to a valid memory location.} \]
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 (uₐ) 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 \( r0...r11 \)
- \( op2 \)  \( uₐ \)  A 16-bit immediate in the range 0...65535.
  If \( uₐ < 64 \), the instruction requires no prefix

Mnemonic and operands:

\[
\text{BRFT} \quad c, uₐ
\]

Operation:

\[
\text{if } c \neq 0 \text{ then } pc \leftarrow pc + uₐ \times 2
\]

Encoding:

\[
\text{ru6} \quad 0 1 1 1 0 \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \\
\]

or prefixed for long immediates:

\[
\text{lru6} \quad 1 1 1 1 0 \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \\
    \quad 0 1 1 1 0 \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \\
\]

Conditions that raise an exception:

\text{ET.ILLEGAL.PC}  \text{ The new PC is not pointing to a valid memory location.}
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:

\[ op_1 \ u_{16} \]

A 16-bit immediate in the range 0...65535.

If \( u_{16} < 64 \), the instruction requires no prefix.

Mnemonic and operands:

\[
\text{BRFU} \ u_{16}
\]

Operation:

\[
pc \leftarrow pc + u_{16} \times 2
\]

Encoding:

\[
\begin{array}{cccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
\end{array}
\]

or prefixed for long immediates:

\[
\begin{array}{cccccccccccc}
1 & 1 & 1 & 1 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
0 & 1 & 1 & 1 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\end{array}
\]

Conditions that raise an exception:

\texttt{ET_ILLEGAL_PC}  The new PC is not pointing to a valid memory location.
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:

\[
\textit{op} \quad s \quad \text{Operand register, one of } r0...r11
\]

Mnemonic and operands:

\[
\text{BRU} \quad s
\]

Operation:

\[
\text{pc} \leftarrow \text{pc} + s_{\text{signed}} \times 2
\]

Encoding:

\[
1r \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \ldots
\]

Conditions that raise an exception:

\texttt{ET\_ILLEGAL\_PC}  The new PC is not pointing to a valid memory location.
BYTEREV

Byte reverse

This instruction reverses the bytes of a word.

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

The instruction has two operands:

\[
\begin{align*}
\text{op1} & \quad d & \text{Operand register, one of } r0 \ldots r11 \\
\text{op2} & \quad s & \text{Operand register, one of } r0 \ldots r11 \\
\end{align*}
\]

Mnemonic and operands:

\[
\text{BYTEREV } d, s
\]

Operation:

\[
d[bwp - 1...0] \leftarrow s[7...0] : s[15...8] : \ldots : s[bwp - 1 : bwp - 8]
\]

Encoding:

\[
\begin{array}{c}
\text{l2r} \\
1 1 1 1 1 1 \ldots 1 \\
0 0 0 0 0 1 1 1 1 0 1 1 0 0
\end{array}
\]
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:

\[
\begin{align*}
op_1 & \quad r & \text{Operand register, one of } r0...r11 \\
op_2 & \quad s & \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{CHKCT } r, s
\]

Operation:

\[
\text{if } \text{hasctoken}(r) \land (s = \text{token}(r)) \quad \text{then } \text{skiptoken}(r) \\
\text{else } \text{raiseexception}
\]

Encoding:

\[
\begin{array}{cccccccc}
2r & 1 & 1 & 0 & 0 & 1 & \cdots & \cdots & 0 & \cdots
\end{array}
\]

Conditions that raise an exception:

- **ETRESOURCEDEP** Resource illegally shared between threads
- **ETILLEGALRESOURCE** \( r \) is not pointing to a channel resource, or the resource is not in use.
- **ETILLEGALRESOURCE** \( r \) contains a data token.
- **ETILLEGALRESOURCE** \( r \) contains a control token different to \( s \).
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:

\[ op1 \quad r \quad \text{Operand register, one of } r0...r11 \]
\[ op2 \quad us \quad \text{An integer in the range } 0...11 \]

Mnemonic and operands:

\[
\text{CHKCTI } r, us
\]

Operation:

\[
\begin{align*}
\text{if } \text{hasctoken}(r) \land (us = \text{token}(r)) \\
\text{then skipctoken}(r) \\
\text{else raiseexcept}
\end{align*}
\]

Encoding:

\[
\text{rus } \begin{array}{cccccccccccc}
1 & 1 & 0 & 0 & 1 & \cdots & \cdots & 1 & \cdots & \cdots
\end{array}
\]

Conditions that raise an exception:

\begin{itemize}
\item \texttt{ET\_RESOURCE\_DEP} \quad \text{Resource illegally shared between threads}
\item \texttt{ET\_ILLEGAL\_RESOURCE} \quad r \text{ is not pointing to a channel resource, or the resource is not in use.}
\item \texttt{ET\_ILLEGAL\_RESOURCE} \quad r \text{ contains a data token.}
\item \texttt{ET\_ILLEGAL\_RESOURCE} \quad r \text{ contains a control token different to } us.
\end{itemize}
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 then enb\_res ← 0
```

Encoding:

```
0 0 0 0 0 1 1 1 1 0 1 1 0 1
```
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:

\[ op \quad r \quad \text{Operand register, one of } r0 \ldots r11 \]

Mnemonic and operands:

\[
\text{CLRPT} \quad r
\]

Operation:

\[
clearporttime(r)
\]

Encoding:

\[
1r \quad 100001111110 \ldots
\]

Conditions that raise an exception:

- \text{ET\_RESOURCE\_DEP}  \quad \text{Resource illegally shared between threads}
- \text{ET\_ILLEGAL\_RESOURCE}  \quad r \text{ is not pointing to a port resource, or the resource is not in use.}
**CLRSR**

Clear bits in the thread’s status register \((sr)\). The mask supplied specifies which bits should be cleared.

SETSR is used to set bits in the status register.

The instruction has one operand:

\[ op1 \quad u_{16} \]

A 16-bit immediate in the range 0...65535.

If \(u_{16} < 64\), the instruction requires no prefix.

Mnemonic and operands:

\[
\text{CLRSR} \quad u_{16}
\]

Operation:

\[
sr \leftarrow sr \land \lnot \text{bit}_{u_{16}}
\]

Encoding:

<table>
<thead>
<tr>
<th>u6</th>
<th>0 1 1 1 1 0 1 1 0</th>
<th>\ldots</th>
<th>\ldots</th>
<th>\ldots</th>
</tr>
</thead>
</table>

or prefixed for long immediates:

<table>
<thead>
<tr>
<th>lu6</th>
<th>1 1 1 1 0</th>
<th>\ldots</th>
<th>\ldots</th>
<th>\ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 1 1 1 0</td>
<td>1 1 0</td>
<td>\ldots</td>
<td>\ldots</td>
</tr>
</tbody>
</table>
**CLZ**

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 (i.e., 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 $r0\ldots r11$
- $op2\ s$  Operand register, one of $r0\ldots r11$

Mnemonic and operands:

\[
\text{CLZ} \quad d, s
\]

Operation:

\[
d \leftarrow \begin{cases} 
  s = 0 & \text{bpw} \\
  s[bpw - 1] = 0, & bpw - 1 - \lfloor \log_2 s \rfloor \\
  s[bpw - 1] = 1, & 0
\end{cases}
\]

Encoding:

\[
\begin{array}{c}
\text{l2r} \\
1 1 1 1 1 0 0 0 0 1 1 1 1 1 0 0
\end{array}
\]
The XMOS XS1 Architecture

**CRC**

Incorporates a word into a Cyclic Redundancy Checksum. The instruction has three operands. The first operand (\(r\)) 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 (\(d\)) and the polynomial to use when computing the CRC (\(p\)).

Note - this instruction may not be available in cores where \(bpw\) exceeds 32. A CRC32 instruction may be provided with four arguments and a structure identical to CRC8.

The instruction has three operands:

- \(op1\) \(r\) Operand register, one of \(r0...r11\)
- \(op2\) \(d\) Operand register, one of \(r0...r11\)
- \(op3\) \(p\) Operand register, one of \(r0...r11\)

Mnemonic and operands:

\[
\text{CRC} \quad r, d, p
\]

Operation:

\[
\begin{align*}
&\text{for } step = 0 \text{ for } bpw \\
&\quad \text{if } (r[0] = 1) \\
&\quad \quad \text{then } r \leftarrow (d[step] : r[bpw − 1..1]) \oplus bit p \\
&\quad \quad \text{else } r \leftarrow (d[step] : r[bpw − 1..1])
\end{align*}
\]

Encoding:

\[
\begin{array}{c}
<table>
<thead>
<tr>
<th>l3r</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1</td>
</tr>
<tr>
<td>1 0 1 0 1</td>
</tr>
</tbody>
</table>
\end{array}
\]
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 polynomial \( p \) to use when computing the CRC, and the data \( d \) 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 \( r \). 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 CRC8 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:

- \( op_1 \) \( o \): Operand register, one of \( r_0 \ldots r_{11} \)
- \( op_4 \) \( r \): Operand register, one of \( r_0 \ldots r_{11} \)
- \( op_2 \) \( d \): Operand register, one of \( r_0 \ldots r_{11} \)
- \( op_3 \) \( p \): Operand register, one of \( r_0 \ldots r_{11} \)

Mnemonic and operands:

\[
\text{CRC8} \quad o, r, d, p
\]

Operation:

\[
\text{for } step = 0 \text{ for } 8 \\
\text{if } (r[0] = 1) \\
\quad \text{then } r \leftarrow (d[step] : r[31...1]) \oplus p \\
\quad \text{else } r \leftarrow (d[step] : r[31...1]) \\
\quad o[bpw - 1...0] \leftarrow 0 : 0 : 0 : 0 : 0 : 0 : 0 : d[bpw - 1 : 8]
\]

Encoding:

\[
\begin{align*}
l4r & \quad \begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & . & . & . & \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & . & . & .
\end{array}
\end{align*}
\]
**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 ← pc0

dssr ← sr0

pc0 ← debugentry

dtype ← dcallcause

sr0[linik] ← 1

sr0[like] ← 1

sr0[leble] ← 0

sr0[lieble] ← 0

sr0[linreb] ← 0

sr0[levant] ← 0

dbg[in] ← 1
```

Encoding:

```
0 0 0 0 0 1 1 1 1 1 1 1 0 0
```
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:

\[ \text{dssp} \leftarrow \text{sp} \]
\[ \text{sp} \leftarrow \text{ramend} \]

Encoding:

\[
\begin{array}{c}
0r \\
0 0 0 1 0 1 1 1 1 1 0 1 0 0
\end{array}
\]

Conditions that raise an exception:

ET_ILLEGAL_INSTRUCTION not in debug mode.
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:

\[ \text{op1 \ s} \quad \text{Operand register, one of r0...r11} \]

Mnemonic and operands:

\[ \text{DGETREG \ s} \]

Operation:

\[ s \leftarrow dtreg_{\text{dtid}} \]

Encoding:

\[ 1r \quad 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ \ldots \] 

Conditions that raise an exception:

ET_ILLEGAL_INSTRUCTION not in debug mode.
DIVS

Signed division

Produces the result of dividing two signed words, rounding the result towards zero. For example $5 \div 3$ is 1, $-5 \div 3$ is $-1$, $-5 \div 3$ is 1, and $5 \div -3$ 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_{\text{signed}} \leftarrow x_{\text{signed}} \div y_{\text{signed}}
```

Encoding:

```
| l3r | 1 1 1 1 1 \cdot \cdot \cdot | 1 1 1 0 0 |
```

Conditions that raise an exception:

- $ET_{ARITHMETIC}$ Division by 0.
- $ET_{ARITHMETIC}$ Division of $-2^{bpw-1}$ by $-1$
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:

\[
\begin{align*}
\text{op1} & \quad d & \text{Operand register, one of } r_0 \ldots r_{11} \\
\text{op2} & \quad x & \text{Operand register, one of } r_0 \ldots r_{11} \\
\text{op3} & \quad y & \text{Operand register, one of } r_0 \ldots r_{11}
\end{align*}
\]

Mnemonic and operands:

\[
\text{DIVU} \quad d, x, y
\]

Operation:

\[
d \leftarrow x \div y
\]

Encoding:

\[
\begin{array}{c}
l3r \quad 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 0 \\
0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0
\end{array}
\]

Conditions that raise an exception:

\[
\text{ET,ARITHMETIC} \quad \text{Division by 0.}
\]
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:

\[ sp \leftarrow dssp \]

Encoding:

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

Conditions that raise an exception:

ET_ILLEGAL_INSTRUCTION not in debug mode.
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:

\[
p_{c_{t0}} \leftarrow \text{dspc} \\
\sr_{r_{t0}} \leftarrow \text{dssr}
\]

Encoding:

```
0 0 0 0 0|1 1 1 1 1|1 1 1 1 0
```

Conditions that raise an exception:

- \text{ET_ILLEGAL_INSTRUCTION} not in debug mode.
- \text{ET_ILLEGAL_PC} The return address is invalid.
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:

\[ \text{op1 } c \quad \text{Operand register, one of } r0...r11 \]

Mnemonic and operands:

\[ \text{ECALLF } c \]

Operation:

\[ \text{nop} \]

Encoding:

1r \[ \begin{array}{cccccccccc}
0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & \cdots \\
\end{array} \]

Conditions that raise an exception:

\[ \text{ET\_ECALL } c = 0. \]
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:

\[ op \ c \ \text{Operand register, one of } r0...r11 \]

Mnemonic and operands:

\[ \text{ECALLT } c \]

Operation:

\[ \text{nop} \]

Encoding:

\[ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ \cdots \]

Conditions that raise an exception:

\[ \text{ET.ECALL } c \neq 0. \]
**EDU**  

Unconditionally disable event

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

The instruction has one operand:

\[ op1 \quad r \quad \text{Operand register, one of } r0...r11 \]

Mnemonic and operands:

\[
\text{EDU} \quad r
\]

Operation:

\[
enb_r \leftarrow 0 \\
\text{thread}_r \leftarrow \text{tid}
\]

Encoding:

\[
1r \quad 0000111110\ldots
\]

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.
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:

\[
\begin{align*}
\text{op1} & \quad d & \text{Operand register, one of } & \text{r0,...,r11} \\
\text{op2} & \quad r & \text{Operand register, one of } & \text{r0,...,r11}
\end{align*}
\]

Mnemonic and operands:

\[
\text{EEF} \quad d, r
\]

Operation:

\[
\begin{align*}
\text{enb}_r & \quad \leftarrow \quad d = 0 \\
\text{thread}_r & \quad \leftarrow \quad \text{tid}
\end{align*}
\]

Encoding:

\[
2r \quad \begin{array}{cccccccccccccccccccc}
0 & 0 & 1 & 0 & 1 & \cdots & \cdots & 1 & \cdots & \cdots
\end{array}
\]

Conditions that raise an exception:

- \text{ET\_RESOURCE\_DEP} Resource illegally shared between threads
- \text{ET\_ILLEGAL\_RESOURCE} \( r \) is not referring to a legal resource, or the resource is not in use.
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:

\[ \text{op1} \quad d \quad \text{Operand register, one of } r0...r11 \]
\[ \text{op2} \quad r \quad \text{Operand register, one of } r0...r11 \]

Mnemonic and operands:

\[ \text{EET} \quad d, r \]

Operation:

\[ \text{enb}_r \leftarrow d \neq 0 \]
\[ \text{thread}_r \leftarrow \text{tid} \]

Encoding:

\[ 2r \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad \cdots \cdots \quad \underbrace{1} \quad \cdots \cdots \]

Conditions that raise an exception:

- \text{ET\_RESOURCE\_DEP} \quad \text{Resource illegally shared between threads}
- \text{ET\_ILLEGAL\_RESOURCE} \quad \text{r is not referring to a legal resource, or the resource is not in use.}
EEU  

Unconditionally enable event

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

The instruction has one operand:

\[
op1 \ r \quad \text{Operand register, one of } r0...r11
\]

Mnemonic and operands:

\[
\text{EEU} \quad r
\]

Operation:

\[
enb_r \leftarrow 1 \\
\text{thread}_r \leftarrow \text{tid}
\]

Encoding:

\[
1r \quad \begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{array} \ldots
\]

Conditions that raise an exception:

- **ET RESOURCE DEP**: Resource illegally shared between threads
- **ET ILLEGAL RESOURCE**: \( \text{op2} \) is not referring to a legal resource, or the resource is not in use.
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; i.e., 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:

\[ \text{op1} \quad d \quad \text{Operand register, one of } r0...r11 \]
\[ \text{op2} \quad r \quad \text{Operand register, one of } r0...r11 \]

Mnemonic and operands:

\[
\text{ENDIN} \quad d, r
\]

Operation:

\[ d \leftarrow \text{buffercount}, \]

Encoding:

\[ 2r \quad 10010 \ldots \ldots \ldots \ldots 1 \ldots \ldots \]

Conditions that raise an exception:

- \text{ET RESOURCE DEP} \quad \text{Resource illegally shared between threads}
- \text{ET ILLEGAL RESOURCE} \quad r \text{ is not referring to a legal resource, or the resource is not in use.}
- \text{ET ILLEGAL RESOURCE} \quad r \text{ is referring to a port which is not in BUFFERS mode.}
- \text{ET ILLEGAL RESOURCE} \quad r \text{ is referring to a port which is not in INPUT mode.}
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:

\[
\text{op1 } u_{16} \quad \text{A 16-bit immediate in the range 0...65535.}
\]

If \( u_{16} < 64 \), the instruction requires no prefix

Mnemonic and operands:

\[
\text{ENTSP } u_{16}
\]

Operation:

\[
\text{if } u_{16} > 0 \\
\quad \text{mem}[sp] \leftarrow lr \\
\quad sp \leftarrow sp - u_{16} \times Bpw
\]

Encoding:

\[
u_6 \quad 0 1 1 1 0 1 1 1 0 1 \cdots \cdots 
\]

or prefixed for long immediates:

\[
\text{lu}_6 \\
0 1 1 1 0 1 1 1 0 1 \cdots \cdots 
\]

Conditions that raise an exception:

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

Perform 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:

- $\text{op}1 \ c$  Operand register, one of $r0...r11$
- $\text{op}2 \ x$  Operand register, one of $r0...r11$
- $\text{op}3 \ y$  Operand register, one of $r0...r11$

Mnemonic and operands:

\[
\text{EQ} \ c, x, y
\]

Operation:

\[
c \leftarrow \begin{cases} 
  x = y, & 1 \\
  x \neq y, & 0 
\end{cases}
\]

Encoding:

\[
\begin{array}{cccccccccc}
3r & 0 & 0 & 1 & 1 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\end{array}
\]
The XMOS XS1 Architecture

**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:

- \( op_1 \)  
  Operand register, one of \( r_0 \ldots r_{11} \)
- \( op_2 \)  
  Operand register, one of \( r_0 \ldots r_{11} \)
- \( op_3 \)  
  \( u_s \)  
  An integer in the range \( 0 \ldots 11 \)

Mnemonic and operands:

\[
\text{EQI} \quad c, x, u_s
\]

Operation:

\[
c \leftarrow \begin{cases} 
  x = u_s, & 1 \\
  x \neq u_s, & 0
\end{cases}
\]

Encoding:

2rus  
\[
1 0 1 1 0 \ldots \ldots \ldots \ldots \ldots
\]
EXTDP

Extend data

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

The instruction has one operand:

\[ op1 \quad u_{16} \]

A 16-bit immediate in the range 0...65535.

If \( u_{16} \leq 64 \), the instruction requires no prefix.

Mnemonic and operands:

\[ \text{EXTDP} \quad u_{16} \]

Operation:

\[ dp \leftarrow dp - u_{16} \times B_{pw} \]

Encoding:

\[
\begin{align*}
u_6 & \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad \ldots \quad \ldots \\
\text{or prefixed for long immediates:} & \\
lu_6 & \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad \ldots \quad \ldots \quad \ldots \\
& \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad \ldots \quad \ldots \\
\end{align*}
\]
EXTSP

Extend stack

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

The instruction has one operand:

\[ op1 \quad u_{16} \quad \text{A 16-bit immediate in the range 0...65535.} \]

If \( u_{16} < 64 \), the instruction requires no prefix

Mnemonic and operands:

\[ \text{EXTSP} \quad u_{16} \]

Operation:

\[ sp \leftarrow sp - u_{16} \times Bpw \]

Encoding:

u6

01111011110\ldots

or prefixed for long immediates:

lu6

11110\ldots

011101110\ldots
FREER

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:

\[ \text{opt} \ r \quad \text{Operand register, one of } r0...r11 \]

Mnemonic and operands:

\[ \text{FREER } r \]

Operation:

\[ \text{inuse}_r \leftarrow 0 \]

Encoding:

\[
\begin{array}{cccccccc}
1r & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & \cdot & \cdot & \cdot
\end{array}
\]

Conditions that raise an exception:

- ETRESOURCE 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.
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 0 0 0 0 1 1 1 1 1 1 0 1 1 1 1
```
**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:

\[
\text{GETD} \quad d, r
\]

Operation:

\[
d \leftarrow \text{data}_r
\]

Encoding:

\[
i2r \quad \begin{array}{cccccccccc}
1 & 1 & 1 & 1 & 1 & \cdots & 1 & \cdots & 1 & \cdots \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0
\end{array}
\]

Conditions that raise an exception:

- \( \text{ET\_RESOURCE\_DEP} \)  Resource illegally shared between threads
- \( \text{ET\_ILLEGAL\_RESOURCE} \)  \( d \) is not referring to a legal resource, or a resource which doesn’t have a DATA register.
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 Chapter 19.3.

The instruction has no operands.

Mnemonic and operands:

```
GETED
```

Operation:

```
r11 ← ed
```

Encoding:

```
0r  0 0 0 0 1 1 1 1 1 1 1 1 1 0
```
**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 0 0 0 1 1 1 1 1 1 1
```
GETID

Get the thread ID of this thread into r11.
The instruction has no operands.
Mnemonic and operands:

```
GETID
```

Operation:

```
r11 ← tid
```

Encoding:

```
0r  0 0 0 1 0 1 1 1 1 1 0 1 1 1 0
```
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:

```
0 0 0 1 0 1 1 1 1 1 0 1 1 1
```

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:

\[
\begin{align*}
\text{LDAWSP} & \quad r1, \; sp[0] \quad // \quad \text{Save SP into R1} \\
\text{SETSP} & \quad r0 \quad \quad \quad \quad // \quad \text{Set SP, and place old SP...} \\
\text{STW} & \quad r1, \; sp[0] \quad \quad \quad \quad ...\text{where KRESTSP expects it} \\
\text{KRESTSP} & \quad 0 \quad \quad \quad \quad \quad \quad \quad // \quad \text{Set KSP, restore SP}
\end{align*}
\]

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:

\[
\begin{align*}
\text{GETKSP} \\
\text{Operation:} \\
\quad r_{11} & \leftrightarrow \ ksp \\
\text{Encoding:} \\
0 & \quad 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0
\end{align*}
\]
**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:

\[
\begin{align*}
\text{op1} & \quad d \quad \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad r \quad \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{GETN} \quad d, r
\]

Operation:

\[
d \leftarrow \text{net}_r
\]

Encoding:

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot \\
& & & & & & & 1 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 1
\end{array}
\]

Conditions that raise an exception:

- **ET_RESOURCE_DEP**: Resource illegally shared between threads
- **ET_ILLEGALRESOURCE**: \(d\) is not referring to a legal channel end, or the channel end is not in use.
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:

\[
\begin{align*}
\text{op1} & \quad d & \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad r & \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{GETPS} \quad d, r
\]

Operation:

\[
d \leftarrow PS[r]
\]

Encoding:

\[
\begin{array}{cccccccccccc}
1 & 1 & 1 & 1 & . & . & . & . & 1 & . & . & . & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0
\end{array}
\]

Conditions that raise an exception:

\[
\text{ET_ILLEGAL PS} \quad d \text{ is not referring to a legal processor state register}
\]
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:

<table>
<thead>
<tr>
<th>RES_TYPE_PORT</th>
<th>Ports</th>
<th>0</th>
<th>cannot be allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES_TYPE_TIMER</td>
<td>Timers</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>RES_TYPE_CHANEND</td>
<td>Channel ends</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RES_TYPE_SYNC</td>
<td>Synchronisers</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>RES_TYPE_THREAD</td>
<td>Threads</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>RES_TYPE_LOCK</td>
<td>Lock</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>RES_TYPE_CLKBLK</td>
<td>Clock source</td>
<td>6</td>
<td>cannot be allocated</td>
</tr>
<tr>
<td>RES_TYPE_PS</td>
<td>Processor state</td>
<td>11</td>
<td>cannot be allocated</td>
</tr>
<tr>
<td>RES_TYPE_CONFIG</td>
<td>Configuration messages</td>
<td>12</td>
<td>cannot be allocated</td>
</tr>
</tbody>
</table>

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:

\[
\begin{align*}
\text{op1} & \quad d \quad \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad u_s \quad \text{An integer in the range } 0...11
\end{align*}
\]

Mnemonic and operands:

\[
\text{GETR} \quad d, u_s
\]

Operation:

\[
\begin{align*}
d & \leftarrow \text{first } res \in \text{setof}(u_s) : \neg \text{inuse}_{res} \\
\text{inuse}_d & \leftarrow 1
\end{align*}
\]

Encoding:

\[
rus \quad 1 \, 0 \, 0 \, 0 \, 0 \, \cdot \, \cdot \, \cdot \, \cdot \, \cdot \, \cdot \, \cdot \, 0 \, \cdot \, \cdot \, \cdot 
\]
GETSR

Get bits from the thread’s Status Register. The mask supplied specifies which bits should be extracted.

The instruction has one operand:

\[ \text{op1} \quad u_{16} \quad \text{A 16-bit immediate in the range 0...65535.} \]
If \( u_{16} < 64 \), the instruction requires no prefix

Mnemonic and operands:

\[
\text{GETSR } \quad u_{16}
\]

Operation:

\[
r_{11} \leftarrow sr \land bit_{u_{16}}
\]

Encoding:

\[
u_{6} \quad \begin{array}{cccccccccc}
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot
\end{array}
\]

or prefixed for long immediates:

\[
l_{u_{6}} \quad \begin{array}{ccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{array}
\]

\[
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot
\]
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:

\[\begin{align*}
\text{op1} & \quad d \quad \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad r \quad \text{Operand register, one of } r0...r11
\end{align*}\]

Mnemonic and operands:

\[
\text{GETST} \quad d, r
\]

Operation:

\[
\begin{align*}
d & \leftarrow \text{first thread } \in \text{threads} : \neg \text{inuse}_{\text{thread}} \\
inuse_d & \leftarrow 1 \\
\text{spaus}ed & \leftarrow \text{spaus}ed \cup \{d\} \\
\text{slaves}_r & \leftarrow \text{slaves}_r \cup \{d\} \\
\text{mstr}_r & \leftarrow \text{tid}
\end{align*}
\]

Encoding:

\[
2r \quad 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1
\]

Conditions that raise an exception:

\[
\begin{align*}
\text{ET\_RESOURCE\_DEP} & \quad \text{Resource illegally shared between threads} \\
\text{ET\_ILLEGAL\_RESOURCE} & \quad r \text{ is not referring to a synchroniser that is in use}
\end{align*}
\]
GETTS

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 WAIT 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:

\[ \text{op1} \quad d \quad \text{Operand register, one of r0...r11} \]
\[ \text{op2} \quad r \quad \text{Operand register, one of r0...r11} \]

Mnemonic and operands:

\[
\text{GETTS} \quad d, r
\]

Operation:

\[
d \leftarrow \text{timestamp}_r
\]

Encoding:

\[
2r \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad \ldots \quad \ldots \quad 0 \quad \ldots 
\]

Conditions that raise an exception:

- ETRESOURCE_DEP Resource illegally shared between threads
- ET_ILLEGAL RESOURCE \( r \) is not referring to a port, or the port is not in use.
IN

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 \(Bp\)w 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:

\[
\begin{array}{ll}
\text{op1} & d \quad \text{Operand register, one of } r0...r11 \\
\text{op2} & r \quad \text{Operand register, one of } r0...r11 \\
\end{array}
\]

Mnemonic and operands:

\begin{align*}
\text{IN} & \quad d, r \\
\text{Operation:} & \quad r \, \Delta \, d \\
\text{Encoding:} & \quad 2r \, \begin{array}{c}1 \, 0 \, 1 \, 1 \, 0 \, \ldots \, \ldots \, | \, 0 \, \ldots \, .
\end{array}
\end{align*}

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 4 tokens in its input buffer.
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:

\[
\begin{align*}
\text{op}_1 & \quad \text{d} & \text{Operand register, one of } r_0 \ldots r_{11} \\
\text{op}_2 & \quad r & \text{Operand register, one of } r_0 \ldots r_{11}
\end{align*}
\]

Mnemonic and operands:

\[\text{INCT} \quad d, r\]

Operation:

\[
\begin{align*}
\text{if } \text{hasctoken}(r) \\
\text{then } r \leftarrow d \\
\text{else raiseexception}
\end{align*}
\]

Encoding:

\[
2r \quad 1 \ 0 \ 0 \ 0 \ 0 \ \cdots \ \cdots \ \cdot \ 1 \ \cdots
\]

Conditions that raise an exception:

- \text{ET\_RESOURCE\_DEP}  
  Resource illegally shared between threads
- \text{ET\_ILLEGAL\_RESOURCE}  
  \(r\) is not pointing to a channel resource, or the resource is not in use.
- \text{ET\_ILLEGAL\_RESOURCE}  
  \(r\) is a channel end which contains a data token in the first entry in its input buffer.
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:

- \( op_1 \) \( d \) Operand register, one of \( r0...r11 \)
- \( op_2 \) \( r \) Operand register, one of \( r0...r11 \)
- \( op_3 \) \( bitp \) A bit position; one of \( bpw, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32 \)

Mnemonic and operands:

\[
\text{INPW } \ d, \ r, \ bitp
\]

Operation:

\[
\text{shiftcount, } \leftarrow \ bitp \\
\quad \quad \quad \quad \ r \triangleright \ d
\]

Encoding:

\[
\begin{array}{cccccccccccc}
1 & 1 & 1 & 1 & 1 & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0
\end{array}
\]

Conditions that raise an exception:

- \text{ET\_RESOURCE\_DEP} Resource illegally shared between threads
- \text{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.
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 \textit{port-width} bits of the destination register are lost.

The instruction has two operands:

\begin{align*}
  \textit{op}_1 & \quad d \quad \text{Operand register, one of } r0...r11 \\
  \textit{op}_2 & \quad r \quad \text{Operand register, one of } r0...r11
\end{align*}

Mnemonic and operands:

\begin{align*}
  \text{INSHR} & \quad d, r
\end{align*}

Operation:

\begin{align*}
  r & \to x \\
  d & \leftarrow x : d[bpw−1...\textit{portwidth}] 
\end{align*}

Encoding:

\begin{align*}
  2r & \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad \ldots \quad \ldots \quad 1 \quad \ldots \quad \ldots
\end{align*}

Conditions that raise an exception:

\begin{align*}
  \text{ET\_RESOURCE\_DEP} & \quad \text{Resource illegally shared between threads} \\
  \text{ET\_ILLEGAL\_RESOURCE} & \quad r \text{ is not pointing to a port resource, or the resource is not in use.}
\end{align*}
**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:

\[
\begin{align*}
\text{op1} & \quad d \quad \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad r \quad \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{INT} \quad d, r
\]

Operation:

\[
\begin{align*}
\text{if } \text{hastoken}(r) \\
\text{then } r \triangleright d \\
\text{else } \text{raiseexception}
\end{align*}
\]

Encoding:

\[
2r \quad \begin{array}{cccccccccccc}
1 & 0 & 0 & 0 & 1 & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot
\end{array}
\]

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.
KCALL

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:

\[
\text{op1} \quad s \quad \text{Operand register, one of r0...r11}
\]

Mnemonic and operands:

\[
\text{KCALL} \quad s
\]

Operation:

\[
\begin{align*}
spc & \leftarrow pc \\
ssr & \leftarrow sr \\
et & \leftarrow ET_KCALL \\
\text{sed} & \leftarrow \text{ed} \\
ed & \leftarrow s \\
\text{pc} & \leftarrow \text{kep + 64} \\
\text{sr[ink]} & \leftarrow 1 \\
\text{sr[ieble]} & \leftarrow 0 \\
\text{sr[eeble]} & \leftarrow 0
\end{align*}
\]

Encoding:

\[
1r \quad \begin{array}{cccccccccccc}
0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & \cdots
\end{array}
\]

Conditions that raise an exception:

\[
\text{ET_KCALL} \quad \text{Kernel call.}
\]
Perform 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:

\[ op1 \quad u_{16} \] A 16-bit immediate in the range 0...65535.

If \( u_{16} < 64 \), the instruction requires no prefix

Mnemonic and operands:

\[ \text{KCALLI} \ u_{16} \]

Operation:

\[
\begin{align*}
\text{spc} & \leftarrow \text{pc} \\
\text{ssr} & \leftarrow \text{sr} \\
\text{et} & \leftarrow \text{ET.KCALL} \\
\text{sed} & \leftarrow \text{ed} \\
\text{ed} & \leftarrow u_{16} \\
\text{pc} & \leftarrow \text{kep} + 64 \\
\text{sr}[\text{ink}] & \leftarrow 1 \\
\text{sr}[\text{ieble}] & \leftarrow 0 \\
\text{sr}[\text{eeble}] & \leftarrow 0
\end{align*}
\]

Encoding:

\[ \text{u6} \quad \begin{array}{c}0 \ \ 1 \ \ 1 \ \ 1 \ \ 0 \ \ | \ \ 1 \ \ 1 \ \ 1\ \ |
\end{array}
\]

or prefixed for long immediates:

\[ \text{lu6} \quad \begin{array}{c}1 \ \ 1 \ \ 1 \ \ 1 \ \ 0 \ \ 0 \ \ |
\end{array}
\]

Conditions that raise an exception:

\[ \text{ET.KCALL} \quad \text{Kernel call.} \]
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:

\[ \text{op1} \quad u_{16} \ \ \text{A 16-bit immediate in the range 0...65535.} \]
\[ \text{If } u_{16} < 64, \text{ the instruction requires no prefix} \]

Mnemonic and operands:

\[ \text{KENTSP } u_{16} \]

Operation:

\[ \text{mem}[ksp] \leftarrow sp \]
\[ sp \leftarrow ksp - n \times Bpw \]

Encoding:

\[ \text{u6} \quad 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ | \ 1 \ 1 \ 1 \ 0 \ | \cdot \cdot \cdot \cdot \cdot \cdot \cdot \]

or prefixed for long immediates:

\[ \text{lu6} \quad 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ | \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \]
\[ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ | \ 1 \ 1 \ 1 \ 0 \ | \cdot \cdot \cdot \cdot \cdot \cdot \cdot \]

Conditions that raise an exception:

\[ \text{ET_LOAD_STORE} \quad \text{Register } ksp \text{ points to an unaligned address, or does not point to a valid memory location.} \]
KRESTSP

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:

\[
\text{op1 } u_{16} \quad \text{A 16-bit immediate in the range 0...65535.}
\]

If \( u_{16} < 64 \), the instruction requires no prefix

Mnemonic and operands:

KRESTSP \( u_{16} \)

Operation:

\[
\begin{align*}
ksp & \leftarrow sp + n \times Bpw \\
sp & \leftarrow \text{mem}[ksp]
\end{align*}
\]

Encoding:

\[
u_{6} \quad 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ ...
\]

or prefixed for long immediates:

\[
l_u{6} \quad 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ ...
\]

Conditions that raise an exception:

\text{ET_LOAD_STORE} The indexed address points to an unaligned address, or the indexed address does not point to a valid memory location.
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:

```
0 0 0 0 0 1 1 1 1 1 1 1 1 0 1
```

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.
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:

- \( op_1 d \) Operand register, one of \( r_0 \ldots r_{11} \)
- \( op_4 e \) Operand register, one of \( r_0 \ldots r_{11} \)
- \( op_2 x \) Operand register, one of \( r_0 \ldots r_{11} \)
- \( op_3 y \) Operand register, one of \( r_0 \ldots r_{11} \)
- \( op_5 v \) Operand register, one of \( r_0 \ldots r_{11} \)

Mnemonic and operands:

\[
\text{LADD } d, e, x, y, v
\]

Operation:

\[
\begin{align*}
d & \leftarrow r[\text{bpw} - 1 \ldots 0] \\
e & \leftarrow r[\text{bpw}] \\
\text{where } r & \leftarrow x + y + v[0]
\end{align*}
\]

Encoding:

\[
\begin{array}{c}
\text{l5r} \\
1 1 1 1 1 \cdot \cdot \cdot \cdot \cdot \cdot \cdot \\
0 0 0 0 0 \cdot \cdot \cdot \cdot 1 \cdot \cdot \cdot \\
\end{array}
\]
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:

- \( op1 \) \texttt{d}  Operand register, one of \( r0...r11 \)
- \( op2 \) \texttt{b}  Operand register, one of \( r0...r11 \)
- \( op3 \) \texttt{i}  Operand register, one of \( r0...r11 \)

Mnemonic and operands:

\[
\text{LD16S} \quad d, b, i
\]

Operation:

\[
d \leftarrow \text{word}[\text{bnum} + 15] : \ldots : \text{word}[\text{bnum} + 15] : \text{word}[\text{bnum} + 15...\text{bnum}]
\]

where \( ea \leftarrow b + i \times 2 \)
\[
bytenum \leftarrow ea \mod Bpw
\]
\[
\text{bnum} \leftarrow 16 \times (bytenum \div 2)
\]
\[
\text{word} \leftarrow \text{mem}[ea - bytenum]
\]

Encoding:

\[
3r \quad \begin{array}{cccccccccccc}
1 & 0 & 0 & 0 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{array}
\]

Conditions that raise an exception:

\[
\text{ET, LOAD, STORE} \quad b \text{ is not 16-bit aligned (unaligned load), or does not point to a valid memory location.}
\]
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:

- \(op_1\)  \(d\)  Operand register, one of \(r0\)...
- \(op_2\)  \(b\)  Operand register, one of \(r0\)...
- \(op_3\)  \(i\)  Operand register, one of \(r0\)...

Mnemonic and operands:

\[
\text{LD8U}\quad d, b, i
\]

Operation:

\[
d \leftarrow 0 : \ldots : 0 \text{ word}[bnum + 7...bnum]
\]

where \(ea \leftarrow b + i\)

\[
bytenum \leftarrow ea \mod Bpw
\]
\[
bnum \leftarrow 8 \times bytenum
\]
\[
word \leftarrow \text{mem}[ea – bytenum]
\]

Encoding:

\[
3r \quad 1 0 0 0 1 \quad \ldots \quad \ldots \quad \ldots \quad \ldots
\]

Conditions that raise an exception:

**ET_LOAD_STORE**  The indexed address does not point to a valid memory location.
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:

\[
\begin{align*}
\text{op1} & \quad d \quad \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad b \quad \text{Operand register, one of } r0...r11 \\
\text{op3} & \quad i \quad \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{LDA16B } d, b, i
\]

Operation:

\[
d \leftarrow b - i \times 2
\]

Encoding:

\[
\begin{array}{cccccccccccccccc}
\text{l3r} & 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 & 1 & 0
\end{array}
\]

\[
0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0
\]
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:

\[
\text{LDA16F } d, b, i
\]

Operation:

\[
d \leftarrow b + i \times 2
\]

Encoding:

<table>
<thead>
<tr>
<th>l3r</th>
<th>1 1 1 1 1 . . . . . .</th>
<th>0 0 1 0 1 1 1 1 1 1 0 1 1 0 0</th>
</tr>
</thead>
</table>

The XMOS XS1 Architecture
LDAPB  

Load backward pc-relative address

Load effective address relative to the program counter. This operation scales the index \((u_{20})\) so that it counts 16-bit entities.

The instruction has one operand:

\[
op1 \quad u_{20} \quad \text{A 20-bit immediate in the range 0...1048575.}
\]

\[
\text{If } u_{20} < 1024, \text{ the instruction requires no prefix}
\]

Mnemonic and operands:

\[
\text{LDAPB} \quad u_{20}
\]

Operation:

\[
r11 \leftarrow pc - u_{20} \times 2
\]

Encoding:

\[
u10 \quad 1 1 0 1 1 | 1 \ldots \ldots \ldots \ldots
\]

or prefixed for long immediates:

\[
lu10 \quad 1 1 1 1 1 0 | 0 \ldots \ldots \ldots \ldots
\]
\[
1 1 0 1 1 | 1 \ldots \ldots \ldots \ldots
\]
LDAPF

Load forward pc-relative address

Load effective address relative to the program counter. This operation scales the index \((u_{20})\) so that it counts 16-bit entities.

The instruction has one operand:

\[
op1 \quad u_{20} \quad \text{A 20-bit immediate in the range 0...1048575.}
\]

If \(u_{20} < 1024\), the instruction requires no prefix

Mnemonic and operands:

\[
\text{LDAPF} \quad u_{20}
\]

Operation:

\[
r11 \leftarrow pc + u_{20} \times 2
\]

Encoding:

\[
\begin{array}{c}
\text{u10} \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad \ldots \quad \ldots \quad \ldots \\
\end{array}
\]

or prefixed for long immediates:

\[
\begin{array}{c}
\text{lu10} \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad \ldots \quad \ldots \quad \ldots \\
1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad \ldots \quad \ldots \quad \ldots \\
\end{array}
\]
LDAWB

Subtract from word address

Load effective address for word given a base-address \( (b) \) and an index \( (i) \)

The instruction has three operands:

\[
\begin{align*}
\text{op}_1 & \quad d & \text{Operand register, one of } r0...r11 \\
\text{op}_2 & \quad b & \text{Operand register, one of } r0...r11 \\
\text{op}_3 & \quad i & \text{Operand register, one of } r0...r11 \\
\end{align*}
\]

Mnemonic and operands:

\[
\text{LDAWB} \quad d, b, i
\]

Operation:

\[
d \leftarrow b - i \times Bpw
\]

Encoding:

L3r

\[
\begin{array}{cccccccccc}
1 & 1 & 1 & 1 & 1 & . & . & . & . & . \\
0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1
\end{array}
\]
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:

\[
\begin{align*}
\text{op1} & \quad d & \text{Operand register, one of } r_0...r_{11} \\
\text{op2} & \quad b & \text{Operand register, one of } r_0...r_{11} \\
\text{op3} & \quad u_s & \text{An integer in the range } 0...11
\end{align*}
\]

Mnemonic and operands:

\[\text{LDAWBI } d, b, u_s\]

Operation:

\[d \leftarrow b - u_s \times Bw\]

Encoding:

\[
\begin{array}{cccccccccccc}
1 & 1 & 1 & 1 & 1 & . & . & . & . & . & . & . \\
1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0
\end{array}
\]
LDAWCP

Load address of word in constant pool

Loads the address of a word relative to the constant pointer. The instruction has one operand:

\[ op1 \quad u_{16} \quad \text{A 16-bit immediate in the range 0...65535.} \]

If \( u_{16} < 64 \), the instruction requires no prefix

Mnemonic and operands:

\[ \text{LDAWCP } u_{16} \]

Operation:

\[ r11 \leftarrow cp + u_{16} \times Bpw \]

Encoding:

\[ u6 \quad 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ \cdots \cdots \cdot \]

or prefixed for long immediates:

\[ lu6 \quad 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ \cdots \cdots \cdot \]

\[ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ \cdots \cdots \cdot \]
**LDAWDP**

Load address of word in data pool

Loads the address of a word relative to the data pointer.

The instruction has two operands:

- \( \text{op1} \ d \) Any of \( r0...r11, \text{cp, dp, sp, lr} \)
- \( \text{op2} \ u_{16} \) A 16-bit immediate in the range 0...65535.
  If \( u_{16} < 64 \), the instruction requires no prefix

Mnemonic and operands:

\[
\text{LDAWDP} \ d, u_{16}
\]

Operation:

\[
d \leftarrow dp \ + \ u_{16} \times Bpw
\]

Encoding:

| ru6 | 0 1 1 0 0 0 | ... | ... | ... |

or prefixed for long immediates:

| lru6 | 1 1 1 1 0 0 | ... | ... | ... |
|      | 0 1 1 0 0 0 | ... | ... | ... |
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:

- \( op_1 \) \( d \) Operand register, one of \( r0...r11 \)
- \( op_2 \) \( b \) Operand register, one of \( r0...r11 \)
- \( op_3 \) \( i \) Operand register, one of \( r0...r11 \)

Mnemonic and operands:

\[
\text{LDAWF } d, b, i
\]

Operation:

\[
d \leftarrow b + i \times Bpw
\]

Encoding:

\[
\begin{array}{cccccccccccccc}
1 & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{array}
\]

\[
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0
\]
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:

\[
\begin{align*}
\text{op1} & \quad d & \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad b & \text{Operand register, one of } r0...r11 \\
\text{op3} & \quad i & \text{An integer in the range } 0...11
\end{align*}
\]

Mnemonic and operands:

\[
\text{LDAWFI} \ d, b, i
\]

Operation:

\[
d \leftarrow b + i \times Bpw
\]

Encoding:

<table>
<thead>
<tr>
<th>l2rus</th>
<th>1 1 1 1 1</th>
<th>· · · · · · · · ·</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 0 0 1</td>
<td>1 1 1 1 1</td>
</tr>
</tbody>
</table>
LDAWSP

Load address of word on stack

Loads the address of a word relative to the stack pointer.

The instruction has two operands:

\[
\begin{align*}
\text{op1} & \quad d & \quad \text{Any of } r0\ldots r11, \, cp, \, dp, \, sp, \, lr \\
\text{op2} & \quad u_{16} & \quad \text{A 16-bit immediate in the range } 0\ldots65535.
\end{align*}
\]

If \( u_{16} < 64 \), the instruction requires no prefix

Mnemonic and operands:

\[
\text{LDAWSP } d, u_{16}
\]

Operation:

\[
d \leftarrow sp + u_{16} \times Bpw
\]

Encoding:

\[
\begin{array}{c}
\text{ru6} \\
0 \ 1 \ 1 \ 0 \ 0 \ 1 \ \cdots \ \cdots \ \cdots \ \cdots \\
\end{array}
\]

or prefixed for long immediates:

\[
\begin{array}{c}
\text{lru6} \\
1 \ 1 \ 1 \ 1 \ 0 \ 0 \ \cdots \ \cdots \ \cdots \ \cdots \\
0 \ 1 \ 1 \ 0 \ 0 \ 1 \ \cdots \ \cdots \ \cdots \\
\end{array}
\]
LDC

Load constant into a register

The instruction has two operands:

\[ \begin{align*}
    \text{op1} & \quad d & \quad \text{Any of } r0...r11, \text{ cp, dp, sp, lr} \\
    \text{op2} & \quad u_{16} & \quad \text{A 16-bit immediate in the range } 0...65535.
\end{align*} \]

If \( u_{16} \leq 64 \), the instruction requires no prefix

Mnemonic and operands:

\[
\text{LDC } \quad d, u_{16}
\]

Operation:

\[ d \leftarrow u_{16} \]

Encoding:

\[
\begin{array}{c}
\text{ru6} \\
\text{lru6}
\end{array}
\]

\[
\begin{array}{c}
011010 \ldots \\
111100 \\
011010 \ldots
\end{array}
\]

or prefixed for long immediates:
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 LDPC, LDSSR, and LDSED
all or part of the state can be restored.
The instruction has no operands.
Mnemonic and operands:

\[
\text{LDET}
\]

Operation:

\[
\text{set} \leftarrow \text{mem}[sp + 4 \times Bpw]
\]

Encoding:

\[
0 0 0 1 0 | 1 1 1 1 1 | 1 1 1 0
\]

Conditions that raise an exception:

\[
\text{ET_, LOAD, STORE} \quad \text{The indexed address does not point to a valid memory location}.
\]
LDIVU

Long unsigned divide

ONLY AVAILABLE IN REVISION-B

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:

- $d \leftarrow (v : x) \div y$
- $e \leftarrow (v : x) \mod y$

Encoding:

\[ \begin{array}{c}
\text{l5r} \\
1 1 1 1 1 . . . . . . . . . . \\
0 0 0 0 0 . . . . . 0 . . . . 
\end{array} \]

Conditions that raise an exception:

ET_ARITHMETIC $y = 0 \lor v \geq y$. 
**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:

\[
\text{LDSED}
\]

Operation:

\[
\text{sed} \gets \text{mem}[\text{sp} + 3 \times Bpw]
\]

Encoding:

\[
0 0 0 1 0\mid 1 1 1 1 1\mid 1 1 0 1
\]

Conditions that raise an exception:

\[
\text{ET, LOAD, STORE}\quad \text{The indexed address does not point to a valid memory location.}
\]
**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:

```
0 0 0 0 1 1 1 1 1 0 1 1 0 0
```

Conditions that raise an exception:

- **ET, LOAD, STORE** The indexed address does not point to a valid memory location.
**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, LDSED, and LDET all or part of the state can be restored.

The instruction has no operands.

Mnemonic and operands:

\[
\text{LDSSR}
\]

Operation:

\[
\text{ssr} \leftarrow \text{mem}[sp + 2 \times Bpw]
\]

Encoding:

\[
0 0 0 0 1 1 1 1 0 1 1 0
\]

Conditions that raise an exception:

**ET_LOAD_STORE** The indexed address does not point to a valid memory location.
The XMOS XS1 Architecture

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:

- \( \text{op1} \ d \)  Operand register, one of \( r0...r11 \)
- \( \text{op2} \ b \)  Operand register, one of \( r0...r11 \)
- \( \text{op3} \ i \)  Operand register, one of \( r0...r11 \)

Mnemonic and operands:

\[
\text{LDW} \quad d, b, i
\]

Operation:

\[
d \leftarrow \text{mem}[b + i \times Bpw]
\]

Encoding:

\[
3r \quad 01001\cdots\cdots\cdots\cdots\cdots\cdots
\]

Conditions that raise an exception:

\[
\text{ET_LOAD_STORE} \quad b \text{ is not word aligned, or the indexed address does not point to a valid memory location.}
\]
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:

\[
\begin{align*}
\text{op}_1 & \quad d \quad \text{Operand register, one of } r0 \ldots r11 \\
\text{op}_2 & \quad b \quad \text{Operand register, one of } r0 \ldots r11 \\
\text{op}_3 & \quad i \quad \text{An integer in the range } 0 \ldots 11
\end{align*}
\]

Mnemonic and operands:

\[
\text{LDWI} \quad d, b, i
\]

Operation:

\[
d \leftarrow \text{mem}[b + i \times Bpw]
\]

Encoding:

\[
\begin{array}{c}
2\text{rus} \\
\end{array}
\begin{array}{c}
0 \quad 0 \quad 0 \quad 0 \quad 1 \quad \cdots \quad \cdots \quad \cdots \quad \cdots
\end{array}
\]

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.
LDWCP Load word from constant pool

Loads a word relative to the constant pool pointer.

The instruction has two operands:

\[
\begin{align*}
\text{op1} & \quad d & \text{Any of } r0\ldots r11, cp, dp, sp, lr \\
\text{op2} & \quad u_{16} & \text{A 16-bit immediate in the range } 0\ldots 65535. \\
& & \text{If } u_{16} < 64, \text{ the instruction requires no prefix}
\end{align*}
\]

Mnemonic and operands:

\[
\text{LDWCP } d, u_{16}
\]

Operation:

\[
d \leftarrow \text{mem}[cp + u_{16} \times \text{Bpw}]
\]

Encoding:

\[
\begin{array}{c}
\text{ru6} \\
011011 \ldots \ldots \ldots \ldots \\
\end{array}
\]

or prefixed for long immediates:

\[
\begin{array}{c}
\text{lru6} \\
111100 \ldots \ldots \ldots \ldots \\
011011 \ldots \ldots \ldots \ldots \\
\end{array}
\]

Conditions that raise an exception:

\[
\text{ET_LOAD.Store } cp \text{ is not word aligned, or the indexed address does not point to a valid memory location.}
\]
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:

\[ \text{op1} \quad u_{20} \quad \text{A 20-bit immediate in the range 0...1048575.} \]
\[ \text{If } u_{20} < 1024, \text{ the instruction requires no prefix} \]

Mnemonic and operands:

\[
\text{LDWCPL } u_{20}
\]

Operation:

\[ r_{11} \leftarrow \text{mem}[cp + u_{20} \times Bpw] \]

Encoding:

\[
\text{u10} \quad 111001\ldots\ldots\ldots\\ldots
\]

or prefixed for long immediates:

\[
\text{lu10} \quad 111001\ldots\ldots\ldots\ldots\quad 111001\ldots\ldots\ldots\ldots
\]

Conditions that raise an exception:

\[ \text{ET_LOAD_STORE } cp \text{ is not word aligned, or the indexed address does not} \]
\[ \text{point to a valid memory location.} \]
LDWDP

Load word form data pool

Loads a word relative to the data pointer.

The instruction has two operands:

\[
\begin{align*}
\text{op1} & \quad d & \text{Any of r0...r11, cp, dp, sp, lr} \\
\text{op2} & \quad u_{16} & \text{A 16-bit immediate in the range 0...65535.}
\end{align*}
\]

If \( u_{16} < 64 \), the instruction requires no prefix

Mnemonic and operands:

\[
\text{LDWDP} \quad d, u_{16}
\]

Operation:

\[
d \leftarrow \text{mem}[dp + u_{16} \times Bpw]
\]

Encoding:

ru6

\[
\begin{array}{cccccc}
0 & 1 & 0 & 1 & 1 & 0 \\
& & & & & \cdot \cdot \cdot \cdot \cdot \cdot \\
\end{array}
\]

or prefixed for long immediates:

lru6

\[
\begin{array}{cccccc}
1 & 1 & 1 & 1 & 0 & 0 \\
& & & & & \cdot \cdot \cdot \cdot \cdot \cdot \\
0 & 1 & 0 & 1 & 1 & 0 \\
& & & & & \cdot \cdot \cdot \cdot \cdot \cdot \\
\end{array}
\]

Conditions that raise an exception:

\text{ET\_LOAD\_STORE} \quad dp \text{ is not word aligned, or the indexed address does not point to a valid memory location.}
**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** \( u_{16} \) A 16-bit immediate in the range 0...65535.
  - If \( u_{16} < 64 \), the instruction requires no prefix

Mnemonic and operands:

\[
\text{LDWSP } d, u_{16}
\]

Operation:

\[
d \leftarrow \text{mem}[sp + u_{16} \times Bpw]
\]

Encoding:

- **ru6**

\[
0 1 0 1 1 1 . . . . . .
\]

or prefixed for long immediates:

- **lru6**

\[
1 1 1 1 0 0 . . . . . .
0 1 0 1 1 1 . . . . . .
\]

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.
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:

\[
\begin{align*}
\text{op1} & \quad d & \text{Operand register, one of } r0 \ldots r11 \\
\text{op4} & \quad e & \text{Operand register, one of } r0 \ldots r11 \\
\text{op2} & \quad x & \text{Operand register, one of } r0 \ldots r11 \\
\text{op3} & \quad y & \text{Operand register, one of } r0 \ldots r11 \\
\text{op5} & \quad v & \text{Operand register, one of } r0 \ldots r11 \\
\text{op6} & \quad w & \text{Operand register, one of } r0 \ldots r11 
\end{align*}
\]

Mnemonic and operands:

\[
\text{LMUL} \quad d, e, x, y, v, w
\]

Operation:

\[
\begin{align*}
e & \leftarrow r[bpw - 1 ... 0] \\
d & \leftarrow r[2bpw - 1 ... bpw] \\
\text{where } r & \leftarrow x \times y + v + w
\end{align*}
\]

Encoding:

\[
\begin{array}{c}
l6r \quad \begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & \cdots & \cdots & \cdots \\0 & 0 & 0 & 0 & \cdots & \cdots & \cdots 
\end{array}
\end{array}
\]
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:

- $\text{op1} = c$  Operand register, one of $r0...r11$
- $\text{op2} = x$  Operand register, one of $r0...r11$
- $\text{op3} = y$  Operand register, one of $r0...r11$

Mnemonic and operands:

$$\text{LSS } c, x, y$$

Operation:

$$c \leftarrow \begin{cases} 
  x_{\text{signed}} < y_{\text{signed}}, & 1 \\
  x_{\text{signed}} \geq y_{\text{signed}}, & 0
\end{cases}$$

Encoding:

$$3r 1100\ldots\ldots\ldots\ldots$$
LSU

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:

\[
\begin{align*}
\text{op1} & \quad c & \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad x & \text{Operand register, one of } r0...r11 \\
\text{op3} & \quad y & \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{LSU } c, x, y
\]

Operation:

\[
c \leftarrow \begin{cases} 
x < y, & 1 \\
x \geq y, & 0 
\end{cases}
\]

Encoding:

\[
3r \quad 11001\ldots\ldots\ldots\ldots\ldots\ldots
\]
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:

- \( op_1 \) \( d \) Operand register, one of \( r0...r11 \)
- \( op_4 \) \( e \) Operand register, one of \( r0...r11 \)
- \( op_2 \) \( x \) Operand register, one of \( r0...r11 \)
- \( op_3 \) \( y \) Operand register, one of \( r0...r11 \)
- \( op_5 \) \( v \) Operand register, one of \( r0...r11 \)

Mnemonic and operands:

\[
\text{LSUB} \quad d, e, x, y, v
\]

Operation:

\[
\begin{align*}
  d & \leftarrow r[bpw - 1...0] \\
  e & \leftarrow r[bpw] \\
  & \quad \text{where } r \leftarrow x - y - v[0]
\end{align*}
\]

Encoding:

\[
\begin{array}{cccccccccccc}
1 & 1 & 1 & 1 & 1 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \\
0 & 0 & 0 & 1 & \cdots & \cdots & 0 & \cdots & \cdots & \cdots & \cdots & \\
\end{array}
\]
MACCS
Multiply and accumulate signed

ONLY AVAILABLE IN REVISION-B

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:

- \textit{op1} \textit{d} Operand register, one of \textit{r0...r11}
- \textit{op4} \textit{e} Operand register, one of \textit{r0...r11}
- \textit{op2} \textit{x} Operand register, one of \textit{r0...r11}
- \textit{op3} \textit{y} Operand register, one of \textit{r0...r11}

Mnemonic and operands:

\begin{align*}
\text{MACCS} & \quad d, e, x, y
\end{align*}

Operation:

\begin{align*}
e & \leftarrow r[bpw - 1...0] \\
d & \leftarrow r[2bpw - 1...bpw] \\
& \text{where } r \leftarrow ((d_{\text{signed}} : e) + x_{\text{signed}} \times y_{\text{signed}}) \mod 2^{2bpw}
\end{align*}

Encoding:

\begin{align*}
\text{l4r} & \quad 11111\ldots1 \ldots0 \ldots0 \\
000011111110 & \ldots1
\end{align*}
MACCU

Multiply and accumulate unsigned

*Only available in revision-B. In revision-A use MACC h, l, x, y, hi, lo which computes \((h : l) = x \times y + (hi : lo)\).*

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:

\[
\text{MACCU } d, e, x, y
\]

Operation:

\[
\begin{align*}
e & \leftarrow r[bpw - 1...0] \\
d & \leftarrow r[2bpw - 1...bpw]
\end{align*}
\]

where \(r \leftarrow ((d : e) + x \times y) \mod 2^{2bpw}\)

Encoding:

<table>
<thead>
<tr>
<th>l4r</th>
<th>1 1 1 1 1</th>
<th>⋯</th>
<th>⋯</th>
<th>⋯</th>
<th>⋯</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 0 0 0 0</td>
<td>1 1 1 1 1</td>
<td>1</td>
<td>⋯</td>
<td>⋯</td>
</tr>
</tbody>
</table>
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:

\[
\text{op1 } r \quad \text{Operand register, one of } r0...r11
\]

Mnemonic and operands:

\[
\text{MJOIN } r
\]

Operation:

\[
sr[\text{eeble}] \leftarrow 0 \\
\text{if } (\text{slaves}_r \setminus \text{spausd} = \emptyset) \quad \text{then} \\
\text{forall } \text{thread} \in \text{slaves}_r : \inuse_{\text{thread}} \leftarrow 0 \\
\text{mjoin}_{\text{syn}(tid)} \leftarrow 0 \\
\text{else} \\
\text{mpausd} \leftarrow \text{mpausd} \cup \{\text{tid}\} \\
\text{mjoin}_r \leftarrow 1 \\
\text{msyn}_r \leftarrow 1
\]

Encoding:

\[
1r \quad 0001011111111\ldots
\]

Conditions that raise an exception:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>ET_RESOURCE_DEP</td>
<td>Resource illegally shared between threads</td>
</tr>
<tr>
<td>ET_ILLEGAL_RESOURCE</td>
<td>(r) is not a synchroniser resource, or the resource is not in use.</td>
</tr>
</tbody>
</table>
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 \( s1 \) 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:

\[
\text{MKMSK} \quad d, s
\]

Operation:

\[
d \leftarrow \begin{cases} 
    s < \text{bpw}, & 2^s - 1 \\
    s \geq \text{bpw}, & 1 : 1 : ... : 1
\end{cases}
\]

Encoding:

\[
2r \quad 1 \, 0 \, 1 \, 0 \, 0 \, \cdot \, \cdot \, \cdot \, \cdot \, \cdot \, \cdot \, \cdot \, \cdot \, \cdot
\]
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 \( bitp \) 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:

\[
\text{MKMSKI} \ d, \ bitp
\]

Operation:

\[
d \leftarrow \begin{cases} 
    bitp < bpw, & 2^{bitp} - 1 \\
    bitp \geq bpw, & 1 : 1 : ... : 1
\end{cases}
\]

Encoding:

\[
rus \quad 1 \ 0 \ 1 \ 0 \ 0 \ \cdot \ \cdot \ \cdot \ \cdot \ 1 \ \cdot \ \cdot \ \cdot
\]
MSYNC

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:

\[ \text{op} \quad r \quad \text{Operand register, one of } r0...r11 \]

Mnemonic and operands:

\[ \text{MSYNC} \quad r \]

Operation:

\[
\begin{align*}
\text{sr}[\text{eeble}] & \leftarrow 0 \\
\text{if } (\text{slaves}_r \setminus \text{spausd} = \emptyset) & \text{ then } \\
\text{spausd} & \leftarrow \text{spausd} \setminus \text{slaves}_r \\
\text{else } & \\
\text{mpausd} & \leftarrow \text{mpausd} \cup \{ \text{tid} \} \\
\text{msyn}_r & \leftarrow 1
\end{align*}
\]

Encoding:

\[
1r \quad 00011111111111 \cdots
\]

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.
The XMOS XS1 Architecture

**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:

- \( op_1 \) \( d \) Operand register, one of \( r0...r11 \)
- \( op_2 \) \( x \) Operand register, one of \( r0...r11 \)
- \( op_3 \) \( y \) Operand register, one of \( r0...r11 \)

Mnemonic and operands:

\[
\text{MUL} \quad d, x, y
\]

Operation:

\[
d \leftarrow (x \times y) \mod 2^{bpw}
\]

Encoding:

\[
\begin{array}{cccccccccccccccc}
\text{l3r} & 1 & 1 & 1 & 1 & \cdots & \cdots & \cdots & \cdots & \cdots & \\
\hline
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0
\end{array}
\]
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:

\[
\begin{align*}
\text{op}1 & \quad d \quad \text{Operand register, one of } r0...r11 \\
\text{op}2 & \quad s \quad \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{NEG} \quad d, s
\]

Operation:

\[
d_{\text{signed}} \leftarrow 2^{bpw} - s
\]

Encoding:

\[
2r \quad 1001000000000000
\]
NOT

Bitwise not

Produces the bitwise not of its source operand.

The instruction has two operands:

\[
\begin{align*}
\text{op1} & \quad d \quad \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad s \quad \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

```
NOT    d, s
```

Operation:

```
d \leftarrow \neg s;
```

Encoding:

```
2r  1 0 0 0 1 \ldots\ldots\ldots\ldots 0 \ldots\ldots
```
**OR**  

Produces the bitwise or of its two source operands.

The instruction has three operands:

\[
\begin{align*}
\text{op1} & \quad d & \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad x & \text{Operand register, one of } r0...r11 \\
\text{op3} & \quad y & \text{Operand register, one of } r0...r11 \\
\end{align*}
\]

Mnemonic and operands:

\[
\text{OR} \quad d, x, y
\]

Operation:

\[
d \leftarrow x \lor y
\]

Encoding:

\[
3r \quad \begin{array}{cccccccccc}
0 & 1 & 0 & 0 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{array}
\]
OUT

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:

\[ \text{op}1 \quad r \quad \text{Operand register, one of } r0...r11 \]
\[ \text{op}2 \quad s \quad \text{Operand register, one of } r0...r11 \]

Mnemonic and operands:

\[ \text{OUT} \quad r, s \]

Operation:

\[ r \triangleleft s \]

Encoding:

\[ r2r \quad \begin{array}{cccccccccccc}
1 & 0 & 1 & 0 & 1 & \cdot & \cdot & \cdot & \cdot & 0 & \cdot & \cdot & \cdot \\
\end{array} \]

Conditions that raise an exception:

- **ETRESOURCE_Dep**  Resource illegally shared between threads
- **ET_ILLEGALRESOURCE**  \( r \) 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.
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:

\[ \text{op}1 \quad r \quad \text{Operand register, one of } r0...r11 \]
\[ \text{op}2 \quad s \quad \text{Operand register, one of } r0...r11 \]

Mnemonic and operands:

\[ \text{OUTCT} \quad r, s \]

Operation:

\[ r \triangleleft \text{ctoken}(s) \]

Encoding:

\[ 2r \quad 01001\ldots\ldots0\ldots\ldots \]

Conditions that raise an exception:

- \text{ET\_RESOURCE\_DEP} (Resource illegally shared between threads)
- \text{ET\_ILLEGAL\_RESOURCE} (r is not a channel end, or not in use.)
- \text{ET\_LINK\_ERROR} (r is a channel end, and the destination has not been set.)
- \text{ET\_LINK\_ERROR} (r is a channel end, and the control token is a reserved hardware token.)
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:

\[
\begin{align*}
\text{op1} & \quad r & \text{Operand register, one of } r_0 \ldots r_{11} \\
\text{op2} & \quad u_s & \text{An integer in the range } 0 \ldots 11
\end{align*}
\]

Mnemonic and operands:

\[
\text{OUTCTI } r, u_s
\]

Operation:

\[
r \gets \text{ctoken}(u_s)
\]

Encoding:

\[
rus \quad 0 1 0 0 1 \ldots 1 \ldots
\]

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

Mnemonic and operands:

\[
\text{OUTPW } s, r, bitp
\]

Operation:

\[
\text{shiftcount, } \leftarrow \text{bitp} \\
\text{r } \leftarrow \text{s}
\]

Encoding:

<table>
<thead>
<tr>
<th>l2rus</th>
<th>1 1 1 1 1</th>
<th>· · ·</th>
<th>· · ·</th>
<th>· · ·</th>
<th>· · ·</th>
</tr>
</thead>
</table>
|       | 1 0 0 1 0 | 1 1 1 1 | 1 | 0 | 1 0 | 1

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.
OUTSHR

Output data and shift

Outs 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:

\[ \text{op1} \quad r \quad \text{Operand register, one of } r0...r11 \]
\[ \text{op2} \quad d \quad \text{Operand register, one of } r0...r11 \]

Mnemonic and operands:

\[ \text{OUTSHR } r, d \]

Operation:

\[ r \leftarrow d[portwidth_r - 1...0] \]
\[ d \leftarrow 0 : ... : 0 : d[bpw - 1...portwidth_r] \]

Encoding:

\[ r2r \quad \begin{array}{cccccccccc}
1 & 0 & 1 & 0 & 1 & \cdots & \cdots & 1 & \cdots
\end{array} \]

Conditions that raise an exception:

- \text{ET\_RESOURCE\_DEP}  
  Resource illegally shared between threads
- \text{ET\_ILLEGAL\_RESOURCE}  
  \( r \) is not pointing to a port resource, or the resource is not in use.
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:

\[ \begin{align*}
  \text{op1} & \quad r & \text{Operand register, one of } r0...r11 \\
  \text{op2} & \quad s & \text{Operand register, one of } r0...r11
\end{align*} \]

Mnemonic and operands:

\[ \text{OUTT} \quad r, s \]

Operation:

\[ r \leftarrow \text{dtoken}(s) \]

Encoding:

\[ r2r \quad 0 \quad 0 \quad 0 \quad 1 \quad \cdot \cdot \cdot \quad \cdot \quad 1 \quad \cdot \cdot \cdot \]

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.
PEEK

Peek at port data

Looks at the value of the port pins, by-passing all input logic. Peek will not pause.

The instruction has two operands:

\[
\begin{align*}
\text{op1} & \quad d & \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad r & \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{PEEK } \quad d, r
\]

Operation:

\[
d \leftarrow \text{pins}(r)
\]

Encoding:

\[
2r \quad 10111\ldots0\ldots
\]

Conditions that raise an exception:

\begin{align*}
\text{ET\_RESOURCE\_DEP} & \quad \text{Resource illegally shared between threads} \\
\text{ET\_ILLEGAL\_RESOURCE} & \quad r \text{ is not a port resource, or the resource is not in use.}
\end{align*}
REMS

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:

- $op_1 \ d$ Operand register, one of $r0...r11$
- $op_2 \ x$ Operand register, one of $r0...r11$
- $op_3 \ y$ Operand register, one of $r0...r11$

Mnemonic and operands:

REMS $d, x, y$

Operation:

$$d_{\text{signed}} \leftarrow x_{\text{signed}} \mod y_{\text{signed}}$$

Encoding:

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c}
 & 1 & 1 & 1 & 1 & . & . & . & . & . & . & . \\
\hline
l3r & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0
\end{array}
\]

Conditions that raise an exception:

- $ET$ ARITHMETIC Remainder of $X$ by 0.
- $ET$ ARITHMETIC Remainder of $-2^{bpw-1}$ by $-1$
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:

\[
\begin{align*}
\text{op1} & \ d & \text{Operand register, one of } r0...r11 \\
\text{op2} & \ x & \text{Operand register, one of } r0...r11 \\
\text{op3} & \ y & \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{REMU} \quad d, x, y
\]

Operation:

\[
d \leftarrow x \mod y
\]

Encoding:

\[
\begin{array}{c}
\text{l3r:} \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0
\end{array}
\]

Conditions that raise an exception:

\text{ET\_ARITHMETIC} \quad \text{Remainder of } X \text{ by 0.}
RETSP

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:

\[ \text{op} \quad u_{16} \]

A 16-bit immediate in the range 0...65535.

If \( u_{16} < 64 \), the instruction requires no prefix

Mnemonic and operands:

\[
\text{RETSP} \quad u_{16}
\]

Operation:

\[
\begin{align*}
\text{if } u_{16} > 0 \text{ then } \\
sp &\leftarrow sp + u_6 \times Bpw \\
lr &\leftarrow \text{mem}[sp] \\
pc &\leftarrow lr
\end{align*}
\]

Encoding:

\[
u_{6} \quad 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1
\]

or prefixed for long immediates:

\[
l_{u_{6}} \quad 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1
\]

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.
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**
  - Value: 0x0000
- **CTRL_INUSE_ON**
  - Value: 0x0008
- **CTRL_COND_NONE**
  - Value: 0x0001
- **CTRL_COND_FULL**
  - Value: 0x0001
- **CTRL_COND_AFTER**
  - Value: 0x0009
- **CTRL_COND_EQ**
  - Value: 0x0011
- **CTRL_COND_NEQ**
  - Value: 0x0019
- **CTRL_COND_GREATER**
  - Value: 0x0021
- **CTRL_COND_LESS**
  - Value: 0x0029
- **CTRL_IE_MODE_EVENT**
  - Value: 0x0002
- **CTRL_IE_MODE_INTERRUPT**
  - Value: 0x000a
- **CTRL_DRIVE_DRIVE**
  - Value: 0x0003
- **CTRL_DRIVE_PULL_DOWN**
  - Value: 0x000b
- **CTRL_DRIVE_PULL_UP**
  - Value: 0x0013
- **CTRL_RUN_STOPR**
  - Value: 0x0007
- **CTRL_RUN_STARTR**
  - Value: 0x000f

The precise effect depends on the resource type:

**Port** 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:

- \(op_1\ r\)  
  - **Operand register**, one of \(r0...r11\)
- \(op_2\ u_{16}\)  
  - A 16-bit immediate in the range 0..65535.
  - If \(u_{16} < 64\), the instruction requires no prefix.
Mnemonic and operands:

\[
\text{SETCI} \quad r, u_{16}
\]

Operation:

\[
\text{control}_r \leftarrow u_{16}
\]

Encoding:

ru6  1 1 1 0 1 0 . . . .

or prefixed for long immediates:

lr6  1 1 1 1 0 0 . . . .
     1 1 1 0 1 0 . . . .

Conditions that raise an exception:

- \text{ET\_RESOURCE\_DEP}  Resource illegally shared between threads
- \text{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
- \text{ET\_ILLEGAL\_RESOURCE}  \(op2\) is not a valid mode, or not a mode that can be used on \(op1\).
SETC

Set resource control bits

Sets the resource control bits. The control bits that can be set with SETC are the following:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTRL_INUSE_OFF</td>
<td>0x0000</td>
</tr>
<tr>
<td>CTRL_INUSE_ON</td>
<td>0x0008</td>
</tr>
<tr>
<td>CTRL_COND_NONE</td>
<td>0x0001</td>
</tr>
<tr>
<td>CTRL_COND_FULL</td>
<td>0x0001</td>
</tr>
<tr>
<td>CTRL_COND_AFTER</td>
<td>0x0009</td>
</tr>
<tr>
<td>CTRL_COND_EQ</td>
<td>0x0011</td>
</tr>
<tr>
<td>CTRL_COND_NEQ</td>
<td>0x0019</td>
</tr>
<tr>
<td>CTRL_COND_GREATER</td>
<td>0x0021</td>
</tr>
<tr>
<td>CTRL_COND_LESS</td>
<td>0x0029</td>
</tr>
<tr>
<td>CTRL_IE_MODE_EVENT</td>
<td>0x0029</td>
</tr>
<tr>
<td>CTRL_IE_MODE_INTERRUPT</td>
<td>0x00a</td>
</tr>
<tr>
<td>CTRL_DRIVE_DRIVE</td>
<td>0x0003</td>
</tr>
<tr>
<td>CTRL_DRIVE_PULL_DOWN</td>
<td>0x000b</td>
</tr>
<tr>
<td>CTRL_DRIVE_PULL_UP</td>
<td>0x0013</td>
</tr>
<tr>
<td>CTRL_RUN_STOPR</td>
<td>0x0007</td>
</tr>
<tr>
<td>CTRL_RUN_STARTR</td>
<td>0x000f</td>
</tr>
</tbody>
</table>

The precise effect depends on the resource type:

**Port**  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^{bw-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:

\[
\text{SETC} \quad r, s
\]

Operation:

\[
\text{control}_r \leftarrow s
\]

Encoding:

<table>
<thead>
<tr>
<th>l2r</th>
<th>1 1 1 1</th>
<th>· · · ·</th>
<th>1</th>
<th>· · · ·</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 0 1 0 1 1 1 1 1 1 0 1 1 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Conditions that raise an exception:

- **ETRESOURCE_DEPEND** Resource illegally shared between threads
- **ET_ILLEGALRESOURCE** \( r \) is not a valid resource, or the resource is not in use, or not a resource on which SETC can be used
- **ET_ILLEGALRESOURCE** \( s \) is not a valid mode, or not a mode that can be used on \( r \).
**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:

\[
\begin{align*}
op1 & \quad r & \text{Operand register, one of } r0...r11 \\
op2 & \quad s & \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{SETCLK} \ r, \ s
\]

Operation:

\[
clk_r \leftarrow s
\]

Encoding:

\[
\begin{array}{cccccccccccc}
    & 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & 1 & 0 & 0 & 0 & 0
\end{array}
\]

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.
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:

\[ \text{op1} \ s \ \text{Operand register, one of } r0...r11 \]

Mnemonic and operands:

\[
\text{SETCP } s
\]

Operation:

\[
\text{cp } \leftarrow s
\]

Encoding:

\[
1r \ 0011011 \ 111111 \ \cdot \cdot \cdot
\]
SETD

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:

\[
\begin{align*}
\text{op}_1 & \quad r & \text{Operand register, one of } r0...r11 \\
\text{op}_2 & \quad s & \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{SETD} \quad r, s
\]

Operation:

\[
data_r \leftarrow s
\]

Encoding:

\[
r2r \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad \cdot \quad \cdot \quad \cdot \quad 1 \quad \cdot \quad \cdot
\]

Conditions that raise an exception:

- **ETRESOURCEDEP** Resource illegally shared between threads
- **ET_ILLEGALRESOURCE** \(r\) is not a channel, timer, port or clock resource, or the resource is not in use.
- **ET_ILLEGALRESOURCE** \(r\) is a running clock-block.
- **ET_ILLEGALRESOURCE** \(r\) is a channel-end, and \(s\) is not a channel-end or a configuration resource.
SETDP

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:

\[ \text{op1} \quad s \quad \text{Operand register, one of } r0...r11 \]

Mnemonic and operands:

\[
\text{SETDP} \quad s
\]

Operation:

\[ dp \leftarrow s \]

Encoding:

\[
\begin{array}{c}
1r \\
0011011101111000... \\
\end{array}
\]
**SETEV**

Set environment vector

Sets the environment vector related to a resource. When a resource issues an event to a thread, this environment vector will overwrite \( ed \). SETEV can be used to pass data specific to a resource to the event handler. SETEV can be used to share a single handler between multiple resources. The event handlers can be set-up once when all event handlers are installed.

SETEV is used in conjunction with SETV, and any of the WAITEU instructions.

The instruction has one operand:

\[
\text{op} \quad r
\]

Operand register, one of \( r0...r11 \)

Mnemonic and operands:

\[
\text{SETEV} \quad r
\]

Operation:

\[
ev_r \leftarrow r11
\]

Encoding:

\[
1r \quad 0 \, 0 \, 1 \, 1 \, 1 \, 1 \, 1 \, 1 \, 1 \, 1 \, 1 \, 1 \, \ldots
\]

Conditions that raise an exception:

- **ET_Resource_Det**: Resource illegally shared between threads
- **ET_Illegal_Resource**: \( r \) is not a port, timer or channel resource, or the resource is not in use.
SETKEP

Set the kernel entry point

Sets the kernel entry point. The kernel entry point should be aligned on a 64-byte boundary.

The instruction has no operands.

Mnemonic and operands:

```
SETKEP
```

Operation:

```
kep ← r11
```

Encoding:

```
0r 0 0 0 0 1 1 1 | 1 1 | 1 1 1 1
```
SETN

Set network

Sets the logical network over which a channel should communicate.

The instruction has two operands:

\[
\begin{align*}
\text{op1} & \quad r & \text{Operand register, one of } r_0 \ldots r_{11} \\
\text{op2} & \quad s & \text{Operand register, one of } r_0 \ldots r_{11}
\end{align*}
\]

Mnemonic and operands:

\[
\text{SETN} \quad r, s
\]

Operation:

\[
\text{net}_r \leftarrow s
\]

Encoding:

\[
\begin{array}{cccccccccccc}
1 & 1 & 1 & 1 & \ldots & \cdot & 0 & \ldots & \cdot \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0
\end{array}
\]

Conditions that raise an exception:

\[
\begin{align*}
\text{ET\_RESOURCE\_DEP} & \quad \text{Resource illegally shared between threads} \\
\text{ET\_ILLEGAL\_RESOURCE} & \quad r \text{ is not a channel end or not in use.}
\end{align*}
\]
**SETPS** Set processor state

Sets a processor internal register. Only used when configuring the core.

The instruction has two operands:

\[
\begin{align*}
\text{op1} & \quad r & & \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad s & & \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{SETPS} \quad r, s
\]

Operation:

\[
ps[r] \leftarrow s
\]

Encoding:

\[
\begin{array}{cccccccccccc}
1 & 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot & 0 & \cdot & \cdot & \cdot \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0
\end{array}
\]

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
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 INPW/IN.
The instruction has two operands:

\[
\begin{align*}
op1 & \quad r & \text{Operand register, one of } r0...r11 \\
op2 & \quad s & \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{SETPSC } r, s
\]

Operation:

\[
\text{shiftcount}_r \leftarrow s
\]

Encoding:

\[
r2r \quad 1\ 1\ 0\ 0\ \cdot\ \cdot\ \cdot\ \cdot\ \cdot\ \cdot
\]

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.
- **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.
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:

\[
\begin{align*}
  op_1 & \quad r & \quad \text{Operand register, one of } r0 \ldots r11 \\
  op_2 & \quad s & \quad \text{Operand register, one of } r0 \ldots r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{SETPT } r, s
\]

Operation:

\[
\text{porttimer}_r \leftarrow s
\]

Encoding:

| r2r | 0 0 1 1 1 | . . . . . . | 1 | . . . |

Conditions that raise an exception:

- \text{ET\_RESOURCE\_DEP} \quad \text{Resource illegally shared between threads}
- \text{ET\_ILLEGAL\_RESOURCE} \quad r \text{ is not pointing to a port resource, or the resource is not in use.}
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:

\[
\begin{align*}
\text{op1} & \quad r & & \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad s & & \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{SETRDY} \quad r, s
\]

Operation:

\[
\text{rdy}_r \leftarrow s
\]

Encoding:

\[
\begin{array}{c}
\text{lr2r} \\
1 1 1 1 1 1 0 1 1 1 1 0 1 1 0 1 1 0
\end{array}
\]

Conditions that raise an exception:

- \( \text{ET\_RESOURCE\_DEP} \): Resource illegally shared between threads
- \( \text{ET\_ILLEGAL\_RESOURCE} \): \( r \) is not pointing to a port or clock resource, or the resource is not in use.
- \( \text{ET\_ILLEGAL\_RESOURCE} \): \( s \) is not pointing to a port resource, or the port is not a 1-bit port.
SETSP

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:

\[ op_1 \ s \]

Operand register, one of \( r0...r11 \)

Mnemonic and operands:

\[
\text{SETSP} \quad s
\]

Operation:

\[
sp \leftarrow s
\]

Encoding:

\[
1r \quad 00101111111\ldots
\]

Conditions that raise an exception:

\[ \text{ET_ILLEGAL_PC} \quad \text{The address was not 16-bit aligned or did not point to a memory location.} \]
SETSR

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.

CLRSR is used to clear bits in the status register.

The instruction has one operand:

\[ op1 \quad u_{16} \]

A 16-bit immediate in the range 0...65535.
If \( u_{16} < 64 \), the instruction requires no prefix

Mnemonic and operands:

\[
\text{SETSR} \quad u_{16}
\]

Operation:

\[
sr \leftarrow sr \lor bit \ u_{16}
\]

Encoding:

\[
u6 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad \ldots \quad \ldots \quad \ldots \quad \ldots
\]

or prefixed for long immediates:

\[
u_{6} \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad \ldots \quad \ldots \quad \ldots \quad \ldots
\]

\[
u_{6} \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad \ldots \quad \ldots \quad \ldots \quad \ldots
\]
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:

\[ \text{op1} \quad r \quad \text{Operand register, one of } r0...r11 \]
\[ \text{op2} \quad s \quad \text{Operand register, one of } r0...r11 \]

Mnemonic and operands:

\[
\text{SE} \text{TTW} \quad r, s
\]

Operation:

\[ \text{transferwidth}_r \leftarrow s \]

Encoding:

\[
\begin{array}{c}
\text{lr2r} \\
\begin{array}{cccccccccccc}
1 & 1 & 1 & 1 & \cdots & \cdots & | & 1 & \cdots & \cdots \\
0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0
\end{array}
\end{array}
\]

Conditions that raise an exception:

- \text{ET_ILLEGAL\_RESOURCE}  \quad r \text{ is not pointing to a port resource, or the port is not in use.}
- \text{ET\_RESOURCE\_DEP}  \quad \text{Resource illegally shared between threads}
- \text{ET\_ILLEGAL\_RESOURCE}  \quad s \text{ is not legal width for the port, or the port is not in BUFFERS mode.}
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:

\[ \text{op1} \quad r \quad \text{Operand register, one of} \quad r0...r11 \]

Mnemonic and operands:

\[
\begin{align*}
\text{SETV} & \quad r \\
\text{Operation:} & \\
\quad v_r & \leftarrow r11
\end{align*}
\]

Encoding:

\[ \begin{array}{c|c}
1 & 1000011111111111 \cdots \\
\end{array} \]

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 resource, or the resource is not in use.
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:

\[
\begin{align*}
\text{op1} & \quad d & \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad s & \text{Operand register, one of } r0...r11 
\end{align*}
\]

Mnemonic and operands:

\[
\text{SEXT} \quad d, s
\]

Operation:

\[
d \leftarrow \begin{cases} 
  s \leq 0 \lor s \geq \text{bpw}, & d \\
  s > 0 \land s < \text{bpw}, & d[s-1] : \ldots : d[1] : d[0]
\end{cases}
\]

Encoding:

\[
2r \quad 0 0 1 1 \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot 
\]
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:

```plaintext
SEXTI d, bitp
```

Operation:

```plaintext
d ← \{ bitp ≤ 0 ∨ bitp ≥ bpw, d \\
    bitp > 0 ∧ bitp < bpw, d[bitp − 1] : ... : d[bitp − 1] : d[bitp − 1...0]
```

Encoding:

```plaintext
rus 0 0 1 1 0 · · · · · · · · 1 · · · ·
```
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 $2^y$.

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:

$$\text{SHL } d, x, y$$

Operation:

$$d \leftarrow \begin{cases} y < bpw, & x[bpw - y...0] : 0 : ... : 0 \\ y \geq bpw, & 0 \end{cases}$$

Encoding:

$$3r \quad 0 \ 0 \ 1 \ 0 \ 0 \ \cdot \ \cdot \ \cdot \ \cdot \ \cdot \ \cdot \ \cdot$$
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 \( 2^{bitp} \).

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:

\[
\text{SHLI } d, x, bitp
\]

Operation:

\[
d \leftarrow \begin{cases} 
    bitp < bpw, & \text{x[bpw} - bitp\text{...0]} : 0 : \ldots : 0 \\
    bitp \geq bpw, & 0 
\end{cases}
\]

Encoding:

\[
2\text{rus } 1\,0\,1\,0\,0\,\ldots\ldots\ldots\ldots\ldots\ldots\ldots
\]
Shift right

Shifts a word right by \( y \) positions, filling the most significant \( y \) bits with zeros. This implements an unsigned divide by \( 2^y \).

For signed shifts, use ASHR.

The instruction has three operands:

- \( op1 \) \( d \): Operand register, one of \( r0 \ldots r11 \)
- \( op2 \) \( x \): Operand register, one of \( r0 \ldots r11 \)
- \( op3 \) \( y \): Operand register, one of \( r0 \ldots r11 \)

Mnemonic and operands:

\[
\text{SHR} \quad d, x, y
\]

Operation:

\[
d \leftarrow \begin{cases} 
y < \text{bpw}, & 0 : \ldots : 0 : x[\text{bpw} - 1..y] 
y \geq \text{bpw}, & 0 
\end{cases}
\]

Encoding:

\[
3r \quad 0 0 1 0 1 \ldots \ldots \ldots \ldots \ldots
\]
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 \( 2^{bitp} \).

For signed shifts, use ASHR.

The instruction has three operands:

\[
\begin{align*}
op1 & \quad \text{Operand register, one of } r0...r11 \\
op2 & \quad \text{Operand register, one of } r0...r11 \\
op3 & \quad \text{A bit position; one of } bpw, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32
\end{align*}
\]

Mnemonic and operands:

\[
\text{SHRI} \quad d, x, \text{bitp}
\]

Operation:

\[
d \leftarrow \begin{cases} \text{bitp} < \text{bpw}, & 0 : \ldots : 0 : x[\text{bpw} - 1...\text{bitp}] \\
\text{bitp} \geq \text{bpw}, & 0 \end{cases}
\]

Encoding:

\[
\text{2rus} \quad 10101\ldots\ldots\ldots\ldots\ldots\ldots\]

SSYNC

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:

\[ sr[\text{eeble}] \leftarrow 0 \]
if \((\text{slaves}_{\text{syn}(\text{tid})} \setminus \text{spausd} = \{ \text{tid} \}) \land \text{msyn}_{\text{syn}(\text{tid})})\]
then
if \(\text{mjoin}_{\text{syn}(\text{tid})}\)
then
forall thread \(\in \text{slaves}_{\text{syn}(\text{tid})} : \text{inuse}_{\text{thread}} \leftarrow 0\)
\(\text{mjoin}_{\text{syn}(\text{tid})} \leftarrow 0\)
else
\(\text{spausd} \leftarrow \text{spausd} \setminus \text{slaves}_{\text{syn}(\text{tid})}\)
\(\text{mpausd} \leftarrow \text{mpausd} \setminus \{ \text{mstr}_{\text{syn}(\text{tid})} \}\)
\(\text{msyn}_{\text{syn}(\text{tid})} \leftarrow 0\)
else
\(\text{spausd} \leftarrow \text{spausd} \cup \{ \text{tid} \}\)

Encoding:

0 0 0 0 0 0 1 1 1 1 1 0
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 mod Bpw
bitnum ← 16 × (bytenum ÷ 2)
```

Encoding:

```
<table>
<thead>
<tr>
<th>l3r</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1</td>
</tr>
<tr>
<td>1 0 0 0 0</td>
</tr>
</tbody>
</table>
```

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.
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:

\[
\begin{align*}
\text{op}_1 & \quad s & \text{Operand register, one of } r0...r11 \\
\text{op}_2 & \quad b & \text{Operand register, one of } r0...r11 \\
\text{op}_3 & \quad i & \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{ST8} \quad s, b, i
\]

Operation:

\[
\text{mem}[\text{ea} - \text{bytenum}][\text{bitnum} + 7...\text{bitnum}] \leftarrow s
\]

where \(\text{ea} \leftarrow b + i \times 2\)

\[
\begin{align*}
\text{bytenum} & \leftarrow \text{ea mod Bpw} \\
\text{bitnum} & \leftarrow 8 \times \text{bytenum}
\end{align*}
\]

Encoding:

\[
\begin{array}{l|cccccccccccc}
 & 1 & 1 & 1 & 1 & . & . & . & . & . & . & . & . & . & . & . \\
\text{l3r} & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0
\end{array}
\]

Conditions that raise an exception:

\[
\text{ET, LOAD, STORE} \quad \text{The indexed address does not point to a valid memory location.}
\]
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] ← set
```

Encoding:

```
0 0 0 0 1 1 1 1 1 1 1 0 1
```

Conditions that raise an exception:

```
ET_LOAD_STORE  The indexed address does not point to a valid memory location.
```
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:

\[
\text{STSED}
\]

Operation:

\[\text{mem}[sp + 3 \times Bpw] \leftarrow \text{sed}\]

Encoding:

\[
0 0 0 0 1 1 1 1 1 1 1 0 0
\]

Conditions that raise an exception:

\text{ET, LOAD, STORE}  
The indexed address does not point to a valid memory location.
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:

\[ \text{mem}[sp + 1 \times Bpw] \leftarrow \text{spc} \]

Encoding:

0r \[ \begin{array}{cccccccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
\end{array} \]

Conditions that raise an exception:

\text{ET, LOAD, STORE} The indexed address does not point to a valid memory location.
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:

\[
\text{STSSR}
\]

Operation:

\[
\text{mem} [sp + 2 \times Bpw] \leftarrow \text{ssr}
\]

Encoding:

0r 0 0 0 0 1 1 1 1 1 1 0 1 1 1 1

Conditions that raise an exception:

\text{ET, LOAD, STORE} The indexed address does not point to a valid memory location.
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:

\[
\begin{align*}
\text{op1} & \quad s & \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad b & \text{Operand register, one of } r0...r11 \\
\text{op3} & \quad i & \text{Operand register, one of } r0...r11 \\
\end{align*}
\]

Mnemonic and operands:

\[
\text{STW} \quad s, b, i
\]

Operation:

\[
\text{mem}[b + i \times B\text{pw}] \leftarrow s
\]

Encoding:

\[
\begin{array}{l}
\text{l3r} \\
1 1 1 1 1 \cdot \cdot \cdot \cdot \cdot \\
0 0 0 0 1 1 1 1 1 0 1 1 0 0
\end{array}
\]

Conditions that raise an exception:

\[
\text{ET, LOAD, STORE} \quad b \text{ is not word aligned, or the indexed address does not point to a valid memory location.}
\]
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:

\[
\begin{align*}
\textit{op1} & \quad s & \text{Operand register, one of } r0...r11 \\
\textit{op2} & \quad b & \text{Operand register, one of } r0...r11 \\
\textit{op3} & \quad i & \text{An integer in the range } 0...11
\end{align*}
\]

Mnemonic and operands:

\[
\text{STWI} \quad s, b, i
\]

Operation:

\[
\text{mem}[b + i \times Bpw] \leftarrow s
\]

Encoding:

\[
\begin{array}{cccccccccccccccc}
2\text{rus} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Conditions that raise an exception:

\*ET_LOAD\_STORE\*  
\*\textit{b} is not word aligned, or the indexed address does not point to a valid memory location.*
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:

\[
\begin{align*}
\text{op1} & \quad s \\
\text{op2} & \quad u_{16}
\end{align*}
\]

- \( s \) Any of \( r0...r11, cp, dp, sp, lr \)
- \( u_{16} \) A 16-bit immediate in the range 0...65535.
  If \( u_{16} < 64 \), the instruction requires no prefix

Mnemonic and operands:

\[
\text{STWDP} \quad s, u_{16}
\]

Operation:

\[
\text{mem}[dp + u_{16} \times Bpw] \leftarrow s
\]

Encoding:

\[
\begin{array}{c}
\text{ru6} \\
0 \ 1 \ 0 \ 1 \ 0 \ 0 \ \ldots \ \ldots \ \ldots
\end{array}
\]

or prefixed for long immediates:

\[
\begin{array}{c}
\text{lru6} \\
1 \ 1 \ 1 \ 1 \ 0 \ 0 \ \ldots \ \ldots \ \ldots \ \\
0 \ 1 \ 0 \ 1 \ 0 \ 0 \ \ldots \ \ldots \ \ldots
\end{array}
\]

Conditions that raise an exception:

\[
\text{ET_LOAD\_STORE} \quad dp \text{ is not word aligned, or the indexed address does not point to a valid memory location.}
\]
STWSP

Stores a word on the stack, using a constant offset from the stack pointer. The offset is specified in words. STWSP used to write to stack variables.

The instruction has two operands:

<table>
<thead>
<tr>
<th>op1</th>
<th>s</th>
<th>Any of r0...r11, cp, dp, sp, lr</th>
</tr>
</thead>
<tbody>
<tr>
<td>op2</td>
<td>u16</td>
<td>A 16-bit immediate in the range 0...65535. If u16 &lt; 64, the instruction requires no prefix</td>
</tr>
</tbody>
</table>

Mnemonic and operands:

```
STWSP  s, u16
```

Operation:

```
mem[sp + u16 × Bpw] ← s
```

Encoding:

```
ru6    0 1 0 1 0|1|· · · · · · ·
```

or prefixed for long immediates:

```
lru6   1 1 1 1 0|0|· · · · · · ·
        0 1 0 1 0|1|· · · · · · ·
```

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.
```
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 \leftarrow (2^{bpw} + x - y) \mod 2^{bpw}
```

Encoding:

```
3r 0 0 0 1 1 \cdot \cdot \cdot \cdot \cdot \cdot \cdot
```
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:

- $op_1$ $d$  Operand register, one of $r0...r11$
- $op_2$ $x$  Operand register, one of $r0...r11$
- $op_3$ $u_s$ An integer in the range 0...11

Mnemonic and operands:

```
SUBI  d, x, u_s
```

Operation:

$$d \leftarrow (2^{bpw} + x) - u_s \mod 2^{bpw}$$

Encoding:

```
2rus 1 0 0 1 1 . . . . . . . . . .
```
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:

\[ \text{opt \ } r \text{ \ \Operand register, one of } r0...r11 \]

Mnemonic and operands:

\[ \text{SYNCR } r \]

Operation:

\[ \text{syncr}(r) \]

Encoding:

\[ 1r \quad \begin{array}{cccccccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{array} \]

Conditions that raise an exception:

\[ \text{ETRESOURCEDEP} \quad \text{Resource illegally shared between threads} \]
\[ \text{ETILLEGALRESOURCE} \quad r \text{ is not a port resource, or the resource is not in use.} \]
TESTLCL

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:

\[ \text{op1 } d \quad \text{Operand register, one of } r0...r11 \]
\[ \text{op2 } r \quad \text{Operand register, one of } r0...r11 \]

Mnemonic and operands:

\[ \text{TESTLCL } d, r \]

Operation:

\[ d \leftarrow \begin{cases} d_{[bpw - 1..16]} = r_{[bpw - 1..16]}, & 1 \\ d_{[bpw - 1..16]} \neq r_{[bpw - 1..16]}, & 0 \end{cases} \]

Encoding:

| l2r | 1 1 1 1 | ... | 0 | ... | 0 | 0 | 0 | 0 | 0 |

Conditions that raise an exception:

\[ \text{ET RESOURCE DEP} \quad \text{Resource illegally shared between threads} \]
\[ \text{ET ILLEGAL RESOURCE} \quad r \text{ is not pointing to a channel resource, or the resource is not in use.} \]
\[ \text{ET ILLEGAL RESOURCE} \quad r \text{ is a channel end, and the destination has not been set.} \]
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:

\[
\begin{align*}
op_1 & \quad \text{Operand register, one of } r_0...r_{11} \\
op_2 & \quad \text{Operand register, one of } r_0...r_{11}
\end{align*}
\]

Mnemonic and operands:

\[
\text{TESTCT } d,r
\]

Operation:

\[
d \leftarrow \begin{cases} 
\text{hasctoken}(r), & 1 \\ 
\neg \text{hasctoken}(r), & 0 
\end{cases}
\]

Encoding:

\[
2r \quad 1 \quad 0 \quad 1 \quad 1 \quad \cdot \ldots \cdot \quad 1 \quad \cdot \ldots
\]

Conditions that raise an exception:

\[
\begin{align*}
\text{ET RESOURCE DEP} & \quad \text{Resource illegally shared between threads} \\
\text{ET ILLEGAL RESOURCE} & \quad r \text{ is not pointing to a channel resource, or the resource is not in use.}
\end{align*}
\]
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:

\[ \text{op}_1 \quad d \quad \text{Operand register, one of r0...r11} \]
\[ \text{op}_2 \quad r \quad \text{Operand register, one of r0...r11} \]

Mnemonic and operands:

\[ \text{TESTWCT \ } d, r \]

Operation:

\[ d \leftarrow \begin{cases} \neg \text{hasctoken}(r), & 0 \\ \text{firsttokenisctoken}, & 1 \\ \text{secondtokenisctoken}, & 2 \\ \text{thirdtokenisctoken}, & 3 \\ \text{fourthtokenisctoken}, & 4 \end{cases} \]

Encoding:

\[ 2r \quad 1100 \cdot \cdot \cdot \cdot \cdot \cdot 1 \cdot \cdot \cdot \]

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.
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:

\[
\begin{align*}
\text{op1} & \quad s \quad \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad t \quad \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{TINITCP } s, t
\]

Operation:

\[
\text{cp}_s \leftarrow t
\]

Encoding:

\[
2r \quad 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0
\]

Conditions that raise an exception:

\[
\begin{align*}
\text{ETRESOURCE\_DEP} & \quad \text{Resource illegally shared between threads} \\
\text{ETILLEGAL\_RESOURCE} & \quad t \text{ is not pointing to a thread resource, or the thread is not in use, or the thread is not SSYNC.}
\end{align*}
\]
**TINITDP**

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:

\[
\begin{align*}
\text{op}_1 &\quad \text{s} & \text{Operand register, one of r0...r11} \\
\text{op}_2 &\quad \text{t} & \text{Operand register, one of r0...r11}
\end{align*}
\]

Mnemonic and operands:

\[\text{TINITDP } s, t\]

Operation:

\[dp_s \leftarrow t\]

Encoding:

\[\text{2r } 0 0 0 0 1 \cdot \cdot \cdot \cdot 0 \cdot \cdot \cdot \cdot\]

Conditions that raise an exception:

- **ETRESOURCEDEP**: Resource illegally shared between threads
- **ETILLEGALRESOURCE**: \(t\) is not pointing to a thread resource, or the thread is not in use, or the thread is not SSYNC.
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:

\[
\begin{align*}
\text{op1} & \quad s \quad \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad t \quad \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{TINITLR } s, t
\]

Operation:

\[
lr_s \leftarrow t
\]

Encoding:

\[
\begin{array}{cccccccccccc}
1 & 1 & 1 & 1 & 1 & \ldots & 0 & \ldots & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & \ldots & 1 & 1 & 0 & 0
\end{array}
\]

Conditions that raise an exception:

\begin{itemize}
\item ET\_RESOURCE\_DEP \quad \text{Resource illegally shared between threads}
\item ET\_ILLEGAL\_RESOURCE \quad t \text{ is not pointing to a thread resource, or the thread is not in use, or the thread is not SSYNC.}
\end{itemize}
**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:

\[
\begin{align*}
\text{op1} & \quad s & \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad t & \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[\text{TINITPC } s, t\]

Operation:

\[
\text{pc}_s \leftarrow t
\]

Encoding:

\[
2r \quad 0 \ 0 \ 0 \ 0 \ \cdots \ \cdots \ 0 \ \cdots
\]

Conditions that raise an exception:

- **ET_RESOURCE_DEP** Resource illegally shared between threads
- **ET_ILLEGALRESOURCE** \( t \) is not pointing to a thread resource, or the thread is not in use, or the thread is not SSYNC.
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:

\[ \text{op1} \quad s \quad \text{Op operand register, one of } r0...r11 \]
\[ \text{op2} \quad t \quad \text{Op operand register, one of } r0...r11 \]

Mnemonic and operands:

\[
\text{TINITSP} \quad s,t
\]

Operation:

\[ sp_s \leftarrow t \]

Encoding:

\[
2r \quad 000010 \cdot \cdot \cdot \cdot \cdot 0 \cdot \cdot \cdot \cdot
\]

Conditions that raise an exception:

\begin{align*}
\text{ET\_RESOURCE\_DEP} & \quad \text{Resource illegally shared between threads} \\
\text{ET\_ILLEGAL\_RESOURCE} & \quad t \text{ is not pointing to a thread resource, or the thread is not in use, or the thread is not SSYNC.}
\end{align*}
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:

\[
\begin{align*}
\text{op1} & \quad d & \text{Operand register, one of } & \text{r0...r11} \\
\text{op2} & \quad s & \text{Operand register, one of } & \text{r0...r11}
\end{align*}
\]

Mnemonic and operands:

\[
\text{TSETMR } d, s
\]

Operation:

\[
mtid_d \leftarrow s
\]

Encoding:

\[
2r \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad \cdots \cdots \quad 1 \quad \cdots \cdots
\]

Conditions that raise an exception:

- ET_RESOURCE_DEP Resource illegally shared between threads
- ET_ILLEGAL_RESOURCE Master thread is not in use.
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:

\[
\begin{align*}
\text{op1} & \quad d & \text{Operand register, one of } r0...r11 \\
\text{op2} & \quad s & \text{Operand register, one of } r0...r11 \\
\text{op3} & \quad t & \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{TSETR } d, s, t
\]

Operation:

\[
d_t \leftarrow s
\]

Encoding:

\[
3r \quad 10111\ldots\ldots\ldots\ldots\ldots
\]

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.
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:

\[ op1 \ t \quad \text{Operand register, one of r0...r11} \]

Mnemonic and operands:

\[
\text{TSTART } t
\]

Operation:

\[
\begin{align*}
\text{spau} & \quad \leftarrow \quad \text{spau} \setminus \{t\} \\
\text{waiting}_t & \quad \leftarrow \quad 0
\end{align*}
\]

Encoding:

\[
1r \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad \cdots
\]

Conditions that raise an exception:

- ET_RESOURCE_DEP: Resource illegally shared between threads
- ET_ILLEGALRESOURCE: \( t \) is not pointing to a thread, or the thread is not in use, or the thread is not SSYNC.
- ET_ILLEGALPC: Thread \( t \) does not have a legal program counter.
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:

$$\text{op1 c}$$  

Operand register, one of \(r0...r11\)

Mnemonic and operands:

$$\text{WAITEF } c$$

Operation:

$$\text{if } c = 0 \text{ then } sr_{id}[\text{eeble}] \leftarrow 1$$

Encoding:

$$1r \quad 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ \cdot \ \cdot \ \cdot$$
WAITET

If true wait for event

Waits for an event when a condition is true. If the condition is 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:

\[ \text{op1 } c \quad \text{Operand register, one of } r0...r11 \]

Mnemonic and operands:

\[ \text{WAITET } c \]

Operation:

\[ \text{if } c \neq 0 \text{ then } s_{rid}[\text{eeble}] \leftarrow 1 \]

Encoding:

\[ 1r \quad 0 0 0 0 1 1 1 1 1 0 \cdots \cdots \]
The XMOS XS1 Architecture

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:

```
src[eeble] ← 1
```

Encoding:

```
0r 0 0 0 0 1 1 1 1 1 1 1 0 1 0 0
```
XOR

Bitwise exclusive or

Produces the bitwise exclusive-or of two words.

The instruction has three operands:

- \( op_1 \) \( d \) Operand register, one of \( r_0...r_{11} \)
- \( op_2 \) \( x \) Operand register, one of \( r_0...r_{11} \)
- \( op_3 \) \( y \) Operand register, one of \( r_0...r_{11} \)

Mnemonic and operands:

\[
\text{XOR} \quad d, x, y
\]

Operation:

\[
d \leftarrow x \oplus y
\]

Encoding:

\[
\begin{array}{ccccccccc}
1 & 1 & 1 & 1 & 1 & \ldots & \ldots & \ldots & \ldots \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0
\end{array}
\]
ZEXT

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:

\[
\begin{align*}
op1 & \quad d & \text{Operand register, one of } r0...r11 \\
op2 & \quad s & \text{Operand register, one of } r0...r11
\end{align*}
\]

Mnemonic and operands:

\[
\text{ZEXT } d, s
\]

Operation:

\[
d \leftarrow \begin{cases} 
  s \leq 0 \lor s \geq bpw, & d \\
  s > 0 \land s < bpw, & 0 : ... : d[s-1...0]
\end{cases}
\]

Encoding:

\[
2r \quad 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ ...
\]
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:

\begin{align*}
  op1 & \quad s \quad \text{Operand register, one of } r0...r11 \\
  op2 & \quad \text{bitp} \quad \text{A bit position; one of } bpw, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32
\end{align*}

Mnemonic and operands:

\[
\text{ZEXTI } s, \text{bitp}
\]

Operation:

\[
s \leftarrow \begin{cases} 
  \text{bitp} \leq 0 \lor \text{bitp} \geq \text{bpw}, & s \\
  \text{bitp} > 0 \land \text{bitp} < \text{bpw}, & 0 : \ldots : 0 : s[\text{bitp} - 1...0] 
\end{cases}
\]

Encoding:

\[
\text{rus} \quad 01000\cdots1\cdots
\]
19.2 Instruction Format Specification

This chapter presents 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 comprises two or four bytes, presented as one or two groups of 16 bits. For each of them, bits are numbered from 15 down to 0. If a bit value depends on the opcode, then this is marked with a “×” 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.

All “long” formats comprise either a prefix instruction to specify an extra 10 bits of immediate operand and a prefixable instruction, or they comprise two instruction words allowing instructions with up to six operands to be represented.
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** $op_1, op_2, op_3$

Instructions in this format are encoded in one word:

```
×××××××××××××××××××××××××
    op3[1...0]
    op2[1...0]
    op1[1...0]
    op1[3...2] × 9 + op2[3...2] × 3 + op3[3..2]
```

This format is used by the following instructions:

- ADD
- LDW
- SHR
- AND
- LSS
- SUB
- EQ
- LSU
- TSETR
- LD16S
- OR
- LD8U
- SHL
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 two words:

This format is used by the following instructions:

ASHR  LDA16F  REMU
CRC   LDAWB   ST16
DIVS  LDAWF   ST8
DIVU  MUL     STW
LDA16B REMS     XOR
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 one word:

```
0000 0000 0000 0000 0000 0000 0000 0000
```

$op3[1...0]$

$op2[1...0]$

$op1[1...0]$

$op1[3...2] \times 9 + op2[3...2] \times 3 + op3[3...2]$

**Opcode**

This format is used by the following instructions:

- ADDI
- SHLI
- SUBI
- EQI
- SHRI
- LDWI
- STWI
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 \textit{op1, op2, op3}

Instructions in this format are encoded in two words:

This format is used by the following instructions:

ASHRI LDAWBI OUTPW
INPW LDAWFI
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** \( op_1, op_2 \)

Instructions in this format are encoded in one word:

```
×××××××××××××××
  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
```

- \( op_2[5...0] \)
- \( op_1[3...0] \)
- Opcode
- Opcode

This format is used by the following instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRBF</td>
<td>LDAWSP</td>
</tr>
<tr>
<td>BRBT</td>
<td>LDC</td>
</tr>
<tr>
<td>BRFF</td>
<td>LDWCP</td>
</tr>
<tr>
<td>BRFT</td>
<td>LDWDP</td>
</tr>
<tr>
<td>LDAWDP</td>
<td>LDWSP</td>
</tr>
</tbody>
</table>
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**  \( op_1, op_2 \)

Instructions in this format are encoded in two words:

\[
\begin{array}{c}
11110\\
\end{array}
\]

\[
\begin{array}{c}
\times\times\times\times\\
\end{array}
\]

- **op2** [15...6]
- **op2** [5...0]
- **op1** [3...0]
- **Opcode**
- **Opcode**

This format is used by the following instructions:

- BRBF  LDAWSP  SETCI
- BRBT  LDC  STWDP
- BRFF  LDWCP  STWSP
- BRFT  LDWDP
- LDAWDP  LDWSP
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 one word:

```
×××××××××××××××
o1[5...0]
```

This format is used by the following instructions:

- BLAT
- EXTDP
- KRESTSP
- BRBU
- EXTSP
- LDAWCP
- BRFU
- GETSR
- RETSP
- CLRSP
- KCALLI
- SETSR
- ENTSP
- KENTSP
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**  \texttt{op1}

Instructions in this format are encoded in two words:

\[\begin{array}{c|c}\hline
\text{Op1[15...6]} & \text{Op1[5...0]} \\
\hline
\text{Opcode} & \text{Opcode} \\
\text{Opcode} & \text{Opcode} \\
\end{array}\]

This format is used by the following instructions:

- BLAT
- EXTDP
- KRESTSP
- BRBU
- EXTP
- LDAWCP
- BRFU
- GETSR
- RETSP
- CLRSPR
- KCALLI
- SETSP
- ENTSP
- KENTSP
10-bit immediate

Instructions with a single operand encoding a 10-bit integer.

The syntax for this instruction is:

MNEMONIC \textit{op1}

Instructions in this format are encoded in one word:

This format is used by the following instructions:

\text{BLACP, BLRF, LDAPF, BLRB, LDAPB, LDWCPL}
20-bit immediate  \hspace{1cm} \text{lu10}

Instructions with a single operand encoding a 20-bit integer. This instruction is a prefixed version of u10.

The syntax for this instruction is:

\textbf{MNEMONIC} \hspace{0.5cm} \text{op1}

Instructions in this format are encoded in two words:

\begin{center}
\begin{tabular}{|c|}
\hline
1 1 1 1 0 | 0 . . . . . . . . . . . . . . . . . . . . . . . . . \hspace{0.5cm} \text{op1}[19\ldots10] \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|c|}
\hline
\times\times\times|\times . . . . . . . . . . . . . . . . . . . . . . . . \hspace{0.5cm} \text{op1}[9\ldots0] \\
\hline
\end{tabular}
\end{center}

- Opcode
- Opcode

This format is used by the following instructions:

- BLACP
- BLRF
- LDAPF
- BLRB
- LDAPB
- LDWCPL

\textbf{The XMOS XS1 Architecture}
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 \( op_1, op_2 \)

Instructions in this format are encoded in one word:

\[
\begin{array}{cccccc}
\times \times \times \times \times & \cdots & \cdots & \times \cdots \cdots \\
\end{array}
\]

- \( op_2[1...0] \)
- \( op_1[1...0] \)
- Opcode
- \((op_1[3...2] \times 3 + op_2[3...2] + 27)[5]\)
- \((op_1[3...2] \times 3 + op_2[3...2] + 27)[4...0]\)
- Opcode

This format is used by the following instructions:

- ANDNOT
- INSHR
- TESTWCT
- CHKCT
- INT
- TINITCP
- EEF
- MKMSK
- TINITDP
- EET
- NEG
- TINITPC
- ENDIN
- NOT
- TINITSP
- GETST
- OUTCT
- TSETMR
- GETTS
- PEEK
- ZEXT
- IN
- SEXT
- INCT
- TESTCT
Two register reversed \( r2r \)

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:

\[
\text{MNEMONIC} \quad op1, op2
\]

Instructions in this format are encoded in one word:

\[
\begin{array}{cccccccc}
\times & \times & \times & \times & \times & \times & \times & \times \\
\end{array}
\]

\[
\begin{array}{c}
op1[1...0] \\
op2[1...0] \\
\text{Opcode} \\
(op2[3...2] \times 3 + op1[3...2] + 27)[5] \\
(op2[3...2] \times 3 + op1[3...2] + 27)[4...0] \\
\text{Opcode}
\end{array}
\]

This format is used by the following instructions:

\[
\begin{array}{c}
\text{OUT} \quad \text{OUTT} \quad \text{SETPSC} \\
\text{OUTSHR} \quad \text{SETD} \quad \text{SETPT}
\end{array}
\]
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:

\[ \text{MNEMONIC} \quad op1, \; op2 \]

Instructions in this format are encoded in two words:

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot \\
\end{array}
\]

This format is used by the following instructions:

BITREV \quad GETD \quad SETC
BYTEREV \quad GETN \quad TESTLCL
CLZ \quad GETPS \quad TINITLR
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**  \( op_1, op_2 \)

Instructions in this format are encoded in two words:

\[
\begin{array}{c}
11111\cdots11\cdots11\cdots11\cdots1
\end{array}
\]

\[ op_1[1...0] \]

\[ op_2[1...0] \]

\[ \text{Opcode} \]

\[ (op_2[3...2] \times 3 + op_1[3...2] + 27)[5] \]

\[ (op_2[3...2] \times 3 + op_1[3...2] + 27)[4...0] \]

\[
\begin{array}{c}
\times\times\times\times\times\times11110\times\times\times
\end{array}
\]

\[ \text{Opcode} \]

\[ \text{Opcode} \]

This format is used by the following instructions:

**SETCLK**  **SETPS**  **SETTW**

**SETN**  **SETRDY**
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 one word:

\[
\begin{array}{cccccccc}
\times & \times & \times & \times & \cdot & \cdot & \cdot & \cdot \\
\times & \times & \times & \times & \cdot & \cdot & \cdot & \cdot \\
\end{array}
\]

- op2[1...0]
- op1[1...0]
- Opcode
- \((op1[3...2] \times 3 + op2[3...2] + 27)[5]\)
- \((op1[3...2] \times 3 + op2[3...2] + 27)[4...0]\)

This format is used by the following instructions:

CKCTI  MKMSKI  SEXTI
GETR  OUTCTI  ZEXTI
Register

Instructions with one operand register.

The syntax for this instruction is:

\[
\text{MNEMONIC } op1
\]

Instructions in this format are encoded in one word:

```
+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   | 1 | 1 | 1 | 1 | 1 |   |
+---+---+---+---+---+---+---+---+---+---+
```

- `op1[3...0]`:
- `Opcode`
- `Opcode`

This format is used by the following instructions:

<table>
<thead>
<tr>
<th>BAU</th>
<th>EEU</th>
<th>SETSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLA</td>
<td>FREER</td>
<td>SETV</td>
</tr>
<tr>
<td>BRU</td>
<td>KCALL</td>
<td>SYNCR</td>
</tr>
<tr>
<td>CLRPT</td>
<td>MJOIN</td>
<td>TSTART</td>
</tr>
<tr>
<td>DGETREG</td>
<td>MSYNC</td>
<td>WAITEF</td>
</tr>
<tr>
<td>ECALLF</td>
<td>SETCP</td>
<td>WAITET</td>
</tr>
<tr>
<td>ECALLT</td>
<td>SETDP</td>
<td></td>
</tr>
<tr>
<td>EDU</td>
<td></td>
<td>SETEV</td>
</tr>
</tbody>
</table>
No operands

These instructions operate on implicit operands.

The syntax for this instruction is:

\[ \text{MNEMONIC} \]

Instructions in this format are encoded in one word:

\[
\begin{array}{cccccccc}
\times & \times & \times & \times & 1 & 1 & 1 & 1 & \times & \times & \times
\end{array}
\]

This format is used by the following instructions:

- CLRE
- GETID
- SETKEP
- DCALL
- GETKEP
- SSYNC
- DENTSP
- GETKSP
- STET
- DRESTSP
- KRET
- STSED
- DRET
- LDET
- STSPC
- FREET
- LDSED
- STSSR
- GETED
- LDSPC
- WAITEU
- GETET
- LDSSR
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 \( op_1, op_4, op_2, op_3 \)

Instructions in this format are encoded in two words:

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
\cdot & \cdot & \cdot & \cdot \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
\times & \times & \times & \times \\
1 & 1 & 1 & 1 \\
\times & \cdot & \cdot & \cdot \\
\end{array}
\]

\[
\begin{aligned}
op_3[1...0] \\
op_2[1...0] \\
op_1[1...0] \\
op_1[3...2] \times 9 + op_2[3...2] \times 3 + op_3[3..2] \\
\end{aligned}
\]

\[
\begin{aligned}
op_4[3...0] \\
Opcode \\
Opcode
\end{aligned}
\]

This format is used by the following instructions:

CRC8  MACCS  MACCU
Five register long

Operations on five registers - the last three operands are source registers, the first two may be used as source or destination registers.

The syntax for this instruction is:

MNEMONIC \( \text{op}_1, \text{op}_4, \text{op}_2, \text{op}_3, \text{op}_5 \)

Instructions in this format are encoded in two words:

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
\times & \times & \times & \times \\
\end{array}
\]

\[
\begin{array}{c}
\text{op}_3[1...0] \\
\text{op}_2[1...0] \\
\text{op}_1[1...0] \\
\text{op}_1[3...2] \times 9 + \text{op}_2[3...2] \times 3 + \text{op}_3[3..2] \\
\text{op}_5[1...0] \\
\text{op}_4[1...0] \\
\text{Op code} \\
\text{Op code} \\
\text{Op code} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Op code} \\
(op_4[3...2] \times 3 + op_5[3...2] + 27)[5] \\
(op_4[3...2] \times 3 + op_5[3...2] + 27)[4...0] \\
\end{array}
\]

This format is used by the following instructions:

LADD  LDIVU  LSUB
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 two words:

This format is used by the following instructions:

LMUL
19.3 Exceptions

Exceptions change the normal flow of control on an XS1; 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.

The different types of exception are listed in this section, together with their representation, their meaning, and the instructions that may cause them.
ET\_LINK\_ERROR

A reserved 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:

\begin{itemize}
\item $et$ will be set to 1.
\item $ed$ will be set to the resource ID of the channel end which generated the exception.
\end{itemize}

This exception may be raised by the following instructions:

\begin{verbatim}
OUT OUTCT OUTT
\end{verbatim}
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:

<table>
<thead>
<tr>
<th>BAU</th>
<th>BRBF</th>
<th>BRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLA</td>
<td>BRBT</td>
<td>DRET</td>
</tr>
<tr>
<td>BLACP</td>
<td>BRBU</td>
<td>KRET</td>
</tr>
<tr>
<td>BLAT</td>
<td>BRFF</td>
<td>MSYNC</td>
</tr>
<tr>
<td>BLRB</td>
<td>BRFT</td>
<td>SETSP</td>
</tr>
<tr>
<td>BLRF</td>
<td>BRFU</td>
<td>TSTART</td>
</tr>
</tbody>
</table>
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
- DGETREG
- DRET
ET_ILLEGALRESOURCE

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_ILLEGALRESOURCE 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:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHKCT</td>
<td>INT</td>
<td>SETRDY</td>
</tr>
<tr>
<td>CLRPT</td>
<td>MJOIN</td>
<td>SETTW</td>
</tr>
<tr>
<td>EDU</td>
<td>MSYNC</td>
<td>SETV</td>
</tr>
<tr>
<td>EEF</td>
<td>OUT</td>
<td>SYNCR</td>
</tr>
<tr>
<td>EET</td>
<td>OUTCT</td>
<td>TESTLCL</td>
</tr>
<tr>
<td>EEU</td>
<td>OUTPW</td>
<td>TESTCT</td>
</tr>
<tr>
<td>ENDIN</td>
<td>OUTSHR</td>
<td>TESTWCT</td>
</tr>
<tr>
<td>FREER</td>
<td>OUTT</td>
<td>TINITCP</td>
</tr>
<tr>
<td>GETD</td>
<td>PEEK</td>
<td>TINITDP</td>
</tr>
<tr>
<td>GETN</td>
<td>SETC</td>
<td>TINITLR</td>
</tr>
<tr>
<td>GETST</td>
<td>SETCLK</td>
<td>TINITPC</td>
</tr>
<tr>
<td>GETTS</td>
<td>SETD</td>
<td>TINITSP</td>
</tr>
<tr>
<td>IN</td>
<td>SETEV</td>
<td>TSETMR</td>
</tr>
<tr>
<td>INCT</td>
<td>SETN</td>
<td>TSETR</td>
</tr>
<tr>
<td>INPW</td>
<td>SETPSC</td>
<td>TSTART</td>
</tr>
<tr>
<td>INSHR</td>
<td>SETPT</td>
<td></td>
</tr>
</tbody>
</table>
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 \( \log_2 Bpw \) 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
- LDSPC
- ST8
- BLAT
- LDSSR
- STET
- ENTSP
- LDW
- STSED
- KENTSP
- LDWCP
- STSPC
- KRESTSP
- LDWCPL
- STSSR
- LD16S
- LDWDP
- STW
- LD8U
- LDWSP
- STWDP
- LDET
- RETSP
- STWSP
- LDSED
- ST16
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
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  REMU
DIVU   REMS
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
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:

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Set to</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHKCT</td>
<td>INT</td>
</tr>
<tr>
<td>CLRPT</td>
<td>MJOIN</td>
</tr>
<tr>
<td>EDU</td>
<td>MSYNC</td>
</tr>
<tr>
<td>EEF</td>
<td>OUT</td>
</tr>
<tr>
<td>EET</td>
<td>OUTCT</td>
</tr>
<tr>
<td>EEU</td>
<td>OUTPW</td>
</tr>
<tr>
<td>ENDIN</td>
<td>OUTSHR</td>
</tr>
<tr>
<td>FREER</td>
<td>OUTT</td>
</tr>
<tr>
<td>GETD</td>
<td>PEEK</td>
</tr>
<tr>
<td>GETN</td>
<td>SETC</td>
</tr>
<tr>
<td>GETST</td>
<td>SETCLK</td>
</tr>
<tr>
<td>GETTS</td>
<td>SETD</td>
</tr>
<tr>
<td>IN</td>
<td>SETEV</td>
</tr>
<tr>
<td>INCT</td>
<td>SETN</td>
</tr>
<tr>
<td>INPW</td>
<td>SETPSC</td>
</tr>
<tr>
<td>INSHR</td>
<td>SETPT</td>
</tr>
</tbody>
</table>
```
**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
Index

Branching, Jumping and Calling
- Adjust stack and save link register, 90
- Branch absolute unconditional register, 53
- Branch and link absolute via constant pool, 56
- Branch and link absolute via register, 55
- Branch and link absolute via table, 57
- Branch and link relative backwards, 58
- Branch and link relative forwards, 59
- Branch relative backwards false, 60
- Branch relative backwards true, 61
- Branch relative backwards unconditional, 62
- Branch relative forward false, 63
- Branch relative forward true, 64
- Branch relative forward unconditional, 65
- Branch relative unconditional register, 66
- Extend data, 93
- Extend stack, 94
- Return, 169
- Set constant pool, 175
- Set the data pointer, 177
- Set the stack pointer, 185

Concurrency and Thread Synchronisation
- Free unsynchronised thread, 96
- Get a synchronised thread, 108
- Get the thread's ID, 100
- Initialise a thread's CP, 212
- Initialise a thread's DP, 213
- Initialise a thread's LR, 214
- Initialise a thread's PC, 215
- Initialise a thread's SP, 216
- Master synchronise, 155
- Set register in thread, 218
- Set the master's register, 217
- Slave synchronise, 195
- Start thread, 219
- Synchronise and join, 152

Test for control token, 68, 210
Test for control token immediate, 69
Test local, 209

Data Access
- 16-bit store, 196
- 8-bit store, 197
- Add to a 16-bit address, 124
- Add to a word address, 131
- Add to a word address immediate, 132
- Load address of word in constant pool, 129
- Load address of word in data pool, 130
- Load address of word on stack, 133
- Load backward pc-relative address, 125
- Load constant, 134
- Load ET from the stack, 135
- Load forward pc-relative address, 126
- Load SED from stack, 137
- Load signed 16 bits, 121
- Load SSR from stack, 139
- Load the SPC from the stack, 138
- Load unsigned 8 bits, 122

Communication
- Get network, 103
- Input a token of data, 114
- Input control tokens, 111
- Input data, 110
- Output a control token, 161
- Output a control token immediate, 162
- Output a token, 165
- Output data, 160
- Set network, 180
Load word, 140
Load word form data pool, 144
Load word from constant pool, 142
Load word from large constant pool, 143
Load word from stack, 145
Load word immediate, 141
Make n-bit mask, 153
Make n-bit mask immediate, 154
Set constant pool, 175
Set the data pointer, 177
Set the stack pointer, 185
Sign extend an n-bit field, 189
Sign extend an n-bit field immediate, 190
Store ET on the stack, 198
Store SED on the stack, 199
Store SPC on the stack, 200
Store the SSR to the stack, 201
Store word, 202
Store word immediate, 203
Store word in data pool, 204
Store word on stack, 205
Subtract from 16-bit address, 123
Subtract from word address, 127
Subtract from word address immediate, 128
Zero extend, 224
Zero extend immediate, 225

Data Manipulation
8-step CRC, 75
And not, 50
Arithmetic shift right, 51
Arithmetic shift right immediate, 52
Bit reverse, 54
Bitwise and, 49
Bitwise exclusive or, 223
Bitwise not, 158
Bitwise or, 159
Byte reverse, 67
Count leading zeros, 73
Equal, 91
Equal immediate, 92
Integer unsigned add, 47
Integer unsigned add immediate, 48
Integer unsigned subtraction, 206
Integer unsigned subtraction immediate, 207
Less than signed, 147
Less than unsigned, 148
Long multiply, 146
Long unsigned add with carry, 120
Long unsigned divide, 136
Long unsigned subtract, 149
Make n-bit mask, 153
Make n-bit mask immediate, 154
Multiply and accumulate signed, 150
Multiply and accumulate unsigned, 151
Shift left, 191
Shift left immediate, 192
Shift right, 193
Shift right immediate, 194
Sign extend an n-bit field, 189
Sign extend an n-bit field immediate, 190
Signed division, 79
Signed remainder, 167
Two's complement negate, 157
Unsigned divide, 80
Unsigned multiply, 156
Unsigned remainder, 168
word CRC, 74
Zero extend, 224
Zero extend immediate, 225

Debugging
Call a debug interrupt, 76
Debug read of another thread's register, 78
Get processor state, 104
Restore non debug stack pointer, 81
Return from debug interrupt, 82
Save and modify stack pointer for debug, 77
Set processor state, 181

Event Handling
Clear all events, 70
Clear bits SR, 72
Enable events conditionally, 87
Enables events conditionally, 86
Get bits from SR, 107
If false wait for event, 220
If true wait for event, 221
Set bits in SR, 186
Unconditionally disable event, 85
Unconditionally enable event, 88
Wait for event, 222

Exceptions
ET_ARITHMETIC, 254
ET_ECALL, 255
ET_ILLEGAL_INSTRUCTION, 250
ET_ILLEGAL_PC, 249
ET_ILLEGAL_PS, 253
ET_ILLEGALRESOURCE, 251
ET_KCALL, 257
ET_LINK_ERROR, 248
ET_LOAD_STORE, 252
ETRESOURCE_DEP, 256

Interrupts, Exceptions and Kernel Calls
Clear bits SR, 72
Get bits from SR, 107
Get ED into r11, 98
Get ET into r11, 99
Get Kernel Stack Pointer, 102
Get the Kernel Entry Point, 101
Kernel call, 115
Kernel call immediate, 116
Kernel Return, 119
Restore stack pointer from kernel stack, 118
Set bits in SR, 186
Set the kernel entry point, 179
Switch to kernel stack, 117
Throw exception if non-zero, 84
Throw exception if zero, 83

Resource Operations
Clear the port time, 71
End a current input, 89
Free a resource, 95
Get a resource, 105
Get resource data, 97
Get the time stamp, 109
Input a part word, 112
Input and shift right, 113
Input data, 110
Output a part word, 163
Output data, 160
Output data and shift, 164
Peek at port data, 166
Set clock for a resource, 174
Set environment vector, 178
Set event data, 176
Set event vector, 188
Set ready input for a port, 184
Set resource control bits, 172
Set resource control bits immediate, 170
Set the port shift count, 182
Set the port time, 183
Set transfer width for a port, 187
Synchronise a resource, 208
Test for position of control token, 211