

# XMOS XS1 Architecture

---

(VERSION 8.7)



2008/07/16

*Authors:*

DAVID MAY

Copyright © 2008, XMOS Ltd.  
All Rights Reserved

## 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 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 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 runtime 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 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 links allowing multiple XS4 chips to be combined in a system. The structure and performance of the system link connections can be varied to meet the needs of applications.

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

Messages are routed through the 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 (8 bit chip address, 8 bit processor address, 8 bit channel address) or 8 bits (1 bit chip address, 2 bit processor address, 5 bit 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 to select an outgoing direction. The direction of each link is set when the switch is configured and it is possible for several 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).

## 2.1 Ports

The ports used for inter-chip 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 links can be switched between between a fast, wide mode and a slower, serial mode. Two encoding schemes are used.

## 2.2 Serial Link

The serial 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.

<b>token</b>	<b>use</b>
224	CREDIT8
225	CREDIT64
226	LRESET
227	CRESET

## 2.3 Fast Link

The fast 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:

### symbol meaning

00001	value	00
00010	value	01
00100	value	10
01000	value	11

10000 escape

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.

### token use

<i>v</i>	<i>v</i>	<i>v</i>	<i>v</i>	256 data tokens
<i>e</i>	<i>v</i>	<i>v</i>	<i>v</i>	64 control tokens 192-255
<i>v</i>	<i>e</i>	<i>v</i>	<i>v</i>	64 control tokens 128-191
<i>v</i>	<i>v</i>	<i>e</i>	<i>v</i>	64 control tokens 64-127
<i>v</i>	<i>v</i>	<i>v</i>	<i>e</i>	64 control tokens 0-63

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

### token use

<i>e</i>	<i>e</i>	<i>v</i>	<i>v</i>	END tokens
<i>v</i>	<i>v</i>	<i>e</i>	<i>e</i>	PAUSE tokens
<i>e</i>	<i>v</i>	<i>v</i>	<i>e</i>	NOP (return to zero) tokens
<i>e</i>	11	11	<i>v</i>	NOP (return to zero) tokens
<i>e</i>	00	<i>e</i>	00	CREDIT8
<i>e</i>	01	<i>e</i>	01	CREDIT64
<i>e</i>	10	<i>e</i>	10	LRESET
<i>e</i>	11	<i>e</i>	11	CRESET

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 tiles.

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:

- PFI<sub>X</sub> which concatenates its 10-bit immediate with the immediate operand of the next 16-bit instruction.
- EO<sub>PR</sub> which concatenates its 11-bit operation set with the following instruction.

The prefixes are inserted automatically by compilers and assemblers.

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

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

The access registers are:

<b>register</b>	<b>number</b>	<b>use</b>
<i>cp</i>	12	the constant pool pointer
<i>dp</i>	13	the data pointer
<i>sp</i>	14	the stack pointer
<i>lr</i>	15	the link register

The control registers are:

<b>register</b>	<b>number</b>	<b>use</b>
<i>pc</i>	16	the program counter
<i>sr</i>	17	the status register

In addition, each thread has seven additional registers which have very specific uses:

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

The status register *sr* contains the following information:

<b>bit</b>	<b>use</b>
<i>eeble</i>	event enable
<i>ieble</i>	interrupt enable
<i>inenb</i>	thread is enabling events
<i>inint</i>	thread is in interrupt mode
<i>ink</i>	thread is in kernel mode
<i>sink</i>	saved <i>ink</i>
<i>waiting</i>	thread waiting to execute current instruction
<i>fast</i>	thread enabled for fast input-output

## 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.

## 5.1 Scheduler Implementation

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

From a software design standpoint, this means that the minimum performance of a thread can be calculated by counting the number of concurrent threads at a specific point in the program. In practice, performance will almost always be higher than this because individual threads will sometimes be delayed waiting for input or output and their unused processor cycles 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  $1/n$  of the clock rate of the processor itself. The only exception to this is that if the number of threads is less than the pipeline depth  $p$ , the clock rate is at most  $1/p$ .

Each thread has a 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 can be 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 ... r11* represent specific operand registers

*x* (a single small letter) represents one of *r0 ... r11*

*X* (a single large letter) represents one of *r0 ... r11*, *sp*, *dp*, *cp* or *lr*

*u<sub>s</sub>* is a small unsigned source operand in the range 0 ... 11

*bitp* is one of 1,2,3,4,5,6,7,8,16,24,32,*bpw* encoded as a *u<sub>s</sub>*

*u<sub>16</sub>* is a 16-bit source operand in the range 0 ... 65535

*u<sub>20</sub>* is a 20-bit source operand in the range 0 ... 1048575 which

Some useful functions are

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

$sext(x, n) = -(2^{n-1} \wedge x) \vee x$  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	$D \leftarrow mem[sp + u_{16} \times Bpw]$	load word from stack
STWSP	$mem[sp + u_{16} \times Bpw] \leftarrow S$	store word to stack
LDAWSP	$D \leftarrow sp + u_{16} \times Bpw$	load address of word in stack

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

LDWDP	$D \leftarrow mem[dp + u_{16} \times Bpw]$	load word from data
STWDP	$mem[dp + u_{16} \times Bpw] \leftarrow S$	store word to data
LDAWDP	$D \leftarrow dp + u_{16} \times Bpw$	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	$D \leftarrow mem[cp + u_{16} \times Bpw]$	load word from constant pool
LDAWCP	$r11 \leftarrow cp + u_{16} \times Bpw]$	load word address in constant pool
LDWCPL	$r11 \leftarrow mem[cp + u_{20} \times Bpw]$	load word from constant pool long
LDAPF	$r11 \leftarrow pc + u_{20} \times 2$	load address in program forward
LDAPB	$r11 \leftarrow pc - u_{20} \times 2$	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	$d \leftarrow mem[b + u_s \times Bpw]$	load word
STWI	$mem[b + u_s \times Bpw] \leftarrow s$	store word
LDAWFI	$d \leftarrow b + u_s \times Bpw$	load address of word forward
LDAWBI	$d \leftarrow b - u_s \times Bpw$	load address of word backward
LDW	$d \leftarrow mem[b + i \times Bpw]$	load word
STW	$mem[b + i \times Bpw] \leftarrow s$	store word
LDAWF	$d \leftarrow b + i \times Bpw$	load address of word forward
LDAWB	$d \leftarrow b - i \times Bpw$	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	$d \leftarrow sext(mem[b + i \times 2], 16)$	load 16-bit signed item
ST16	$mem[b + i \times 2] \leftarrow s$	store 16-bit item
LDA16F	$d \leftarrow b + i \times 2$	load address of 16-bit item forward
LDA16B	$d \leftarrow b - i \times 2$	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	$d \leftarrow zext(mem[b + i], 8)$	load byte unsigned
ST8	$mem[b + i] \leftarrow s$	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	$d \leftarrow 2^s - 1$	make mask
MKMSKI	$d \leftarrow 2^{bitp} - 1$	make mask immediate
SEXT	$d \leftarrow sext(d, s)$	sign extend
SEXTI	$d \leftarrow sext(d, bitp)$	sign extend immediate
ZEXT	$d \leftarrow zext(d, s)$	zero extend
ZEXTI	$d \leftarrow zext(d, bitp)$	zero extend immediate
ANDNOT	$d \leftarrow d \wedge \neg s$	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	$d \leftarrow l \wedge r$	and
OR	$d \leftarrow l \vee r$	or
XOR	$d \leftarrow l \oplus r$	exclusive or
NOT	$d \leftarrow (-1) \oplus s$	not
SHLI	$d \leftarrow l \ll bitp$	logical shift left immediate
SHL	$d \leftarrow l \ll r$	logical shift left
SHRI	$d \leftarrow l \gg bitp$	logical shift right immediate
SHR	$d \leftarrow l \gg r$	logical shift right
ASHRI	$d \leftarrow l \gg_{sgn} bitp$	arithmetic shift right immediate
ASHR	$d \leftarrow l \gg_{sgn} r$	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	$d \leftarrow l \bmod r$	remainder unsigned
REMS	$d \leftarrow l \bmod_{sgn} r$	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	if $c$ then $pc \leftarrow pc + u_{16} \times 2$	branch relative forward true
BRFF	if $\neg c$ then $pc \leftarrow pc + u_{16} \times 2$	branch relative forward false
BRBT	if $c$ then $pc \leftarrow pc - u_{16} \times 2$	branch relative backward true
BRBF	if $\neg c$ 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.

EXTSP	$sp \leftarrow sp - u_{16} \times Bpw$	extend stack
EXTDP	$dp \leftarrow dp - u_{16} \times Bpw$	extend data
ENTSP	if $u_{16} > 0$ { $mem[sp] \leftarrow lr; sp \leftarrow sp - u_{16} \times Bpw$ }	entry and extend stack
RETSP	if $u_{16} > 0$ then { $sp \leftarrow sp + u_{16} \times Bpw; lr \leftarrow mem[sp]$ }; $pc \leftarrow lr$	contract stack and return

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

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

SETSP	$sp \leftarrow s$	set stack pointer
SETDP	$dp \leftarrow s$	set data pointer
SETCP	$cp \leftarrow s$	set pool pointer

## 10 Resources and the Thread Scheduler

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

GETR  $r \leftarrow \text{first } res \in \text{setof}(us) : \neg inuse_{res};$  get resource  
 $inuse_r \leftarrow true$

FREER  $inuse_r \leftarrow false$  free resource

In the above  $setof(r)$  returns the set corresponding to the source operand of  $r$ .

The resources are:

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

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

SETC  $control_r \leftarrow u_{16}$  set resource control

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

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

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

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

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

## 11 Concurrency and Thread Synchronisation

A thread can initiate execution on one or more newly allocated threads, and can subsequently synchronise with them to exchange data or to ensure that all threads have completed before continuing. Thread synchronisation is performed using hardware *synchronisers*, and threads using a synchroniser will move between running states and paused states. When a thread is first created, it is in a paused state and its access registers can be initialised using the following instructions.

TINITPC  $pc_t \leftarrow s$  set thread pc  
 TINITSP  $sp_t \leftarrow s$  set thread stack  
 TINITDP  $dp_t \leftarrow s$  set thread data  
 TINITCP  $cp_t \leftarrow s$  set thread pool  
 TINITLR  $lr_t \leftarrow s$  set thread link

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.

TSETR  $d_t \leftarrow s$  set thread operand register  
 TSETMR  $d_{mstr(tid)} \leftarrow s$  set master thread operand register

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

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

GETST  $d \leftarrow \text{first } thread \in threads : \neg inuse_{thread};$  get synchronised thread  
 $inuse_d \leftarrow true;$   
 $spausd \leftarrow spausd \cup \{d\};$   
 $slaves_s \leftarrow slaves_s \cup \{d\}$   
 $mstr_s \leftarrow tid$

MSYNC if  $(slaves_s \setminus spausd = \emptyset)$  master synchronise  
 then {  
 $spausd \leftarrow spausd \setminus slaves_s$  }  
 else {  
 $mpausd \leftarrow mpausd \cup \{tid\};$   
 $msyn_s \leftarrow true$  }

The group of threads can synchronise at any point by the slaves executing the SSYNC and the master the MSYNC. Once all the threads have synchronised

they are unpaused and continue executing from the next instruction. The processor maintains a set of paused master threads *mpaused* and a set of paused slave threads *spaused* from which it derives the set of runnable threads *run*:

$$run = \{ thread \in threads : inuse_{thread} \} \setminus (spaused \cup mpaused)$$

Each synchroniser also maintains a record *msyn<sub>s</sub>* of whether its master has reached a synchronisation point.

```
SSYNC  if (slavessyn(tid) \ spaused = {tid}) ∧ msynsyn(tid)      slave synchronise
      then {
        if mjoinsyn(tid)
        then {
          forall thread ∈ slavessyn(tid) : inusethread ← false;
          mjoinsyn(tid) ← false }
        else
          spaused ← spaused \ slavessyn(tid);
          mpaused ← mpaused \ {mstr(syn(tid))};
          msynsyn(tid) ← false }
        else
          spaused ← spaused ∪ {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 (slavess \ spaused = ∅)      master join
      then {
        forall thread ∈ slavess : inusethread ← false;
        mjoinsyn(tid) ← false }
      else {
        mpaused ← mpaused ∪ {tid};
        mjoins ← true;
        msyns ← 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.

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

SETD  $r_{dest} \leftarrow s$  set destination

GETD  $d \leftarrow r_{dest}$  get destination

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

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

```
SETN   $c_{net} \leftarrow s$   set network
GETN   $d \leftarrow c_{net}$   get network
```

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  $d$ .

OUTT	$c \triangleleft dtoken(s)$	output token
OUTCT	$c \triangleleft ctoken(s)$	output control token
OUTCTI	$c \triangleleft ctoken(us)$	output control token immediate
INT	if $hasctoken(c)$ then <i>trap</i> else $c \triangleright d$	input token
INCT	if $hasctoken(c)$ then $c \triangleright d$ else <i>trap</i>	input control token
CHKCT	if $hasctoken(c) \wedge (s = token(c))$ then $skiptoken(c)$ else <i>trap</i>	check control token
CHKCTI	if $hasctoken(c) \wedge (s = token(c))$ then $skiptoken(c)$ else <i>trap</i>	check control token immediate
OUT	$c \triangleleft s$	output data word
IN	if $containsctoken(c)$ then <i>trap</i> else $c \triangleright d$	input token
TESTCT	$d \leftarrow hasctoken(c)$	test for control token
TESTWCT	$d \leftarrow containsctoken(c)$	test word for control token

The channel connection is established when the first output is executed. If the destination channel end is on another XCore, this will cause the destination identifier to be sent through the interconnect, establishing a route for the subsequent

data and control tokens. The connection is terminated when an END control token is sent. If a subsequent output is executed using the same channel end, the destination identifier will be used again to establish a new route which will again persist until another END control token is sent.

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

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

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

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

A control token takes up a single byte of storage in the channel. On the receiving end the software can test whether the next token is a control token using

the TESTCT instruction, which waits until at least one token is available. It is also possible to test whether the next word contains a control token using the TESTWCT instruction. This waits until a whole word of data tokens has been received (in which case it returns 0) or until a control token has been received (in which case it returns the byte position after the position of the byte containing the control token).

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

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

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

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

```
TESTLCL  $d \leftarrow islocal(c)$  test destination local
```

## 13 Locks

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

When a thread executes an IN on a lock which is already claimed, it is paused

and placed in a queue waiting for the lock. Whenever a lock is freed by an OUT instruction and the lock's queue is not empty, the next thread in the queue is unpaused; it will then succeed in claiming the lock.

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

When the lock is no longer needed, it can be freed using a FREER / 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 GETR *t*, TIMER instruction. It can be configured using the SETC instruction; the only two modes which can be set are UNCOND and AFTER.

mode	effect
UNCOND	timer <i>always</i> ready; inputs complete immediately
AFTER	timer ready when it's current time is <i>after</i> its DATA value

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

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

SETCLK  $clock_d \leftarrow s$  set clock source

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

Each clock can use a one bit port as its clock source. A clock  $c$  which is to use a port  $p$  as its clock source can be connected to it by executing a SETCLK  $c, p$  instruction. Alternatively, a clock may use the reference clock as its clock source (by SETCLK  $c, REF$ ) 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 SETD instruction. The lowest 8 bits of the operand are used and the rest ignored.

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

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

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

## 15 Ports, Input and Output

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

data in the port meets the condition. When the condition is met the captured data is *time stamped* with the time at which it was captured.

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

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

## 15.1 Input and Output

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

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

OUTSHR	$p \leftarrow s[\text{bits } 0 \text{ for } trwidth(p)];$ $s \leftarrow s \gg trwidth(p)$	output to port and shift
INSHR	$s \leftarrow s \gg trwidth(p);$ $p \triangleright s[\text{bits } (bpw - trwidth(p)) \text{ for } trwidth(p)]$	shift and input from port

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

## 15.2 Port Configuration

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

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

<b>mode</b>	<b>effect</b>
<b>NOREADY</b>	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)
<b>SYNCHRONISED</b>	processor synchronises with pins
BUFFERED	port buffers data between pins and processor
<b>SLAVE</b>	port acts as a slave
MASTER	port acts as a master
<b>NOSDELAY</b>	input sample not delayed
SDELAY	input sample delayed half a clock period
<b>DATAPORT</b>	port acts as normal
CLOCKPORT	the port outputs its source clock
READYPORT	the port outputs a ready signal
<b>DRIVE</b>	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
<b>NOINVERT</b>	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  $d \leftarrow pins(p)$  read port pins
```

### 15.3 Configuring Ready and Clock Signals

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

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

```
SETRDY  $ready_p \leftarrow s$  set source of port ready input
```

A port's ready output signal is output by another associated one-bit port. A one-bit port  $r$  which is to be used as a ready output must first be configured in READYPORT mode by SETC  $r$ , READYPORT. This ready port  $r$  can then be associated with a port  $p$  by SETRDY  $r, p$ .

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

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

### 15.4 NOREADY mode

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

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

## 15.5 HANDSHAKEN mode

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

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

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

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

## 15.6 STROBED mode

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

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

## 15.7 The Port Timer

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

SETPT	$porttime_p \leftarrow s$	set port time
CLRPT	$clearporttime(p)$	clear port time
GETTS	$d \leftarrow timestamp_p$	get port timestamp

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

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

## 15.8 Conditions

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

### mode port ready condition

NONE	no condition
EQ	value on pins <i>equal to</i> port data register value
NEQ	value on pins <i>not equal to</i> port data register value

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

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

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

## 15.9 Synchronised Transfers

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

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

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

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

### 15.10 Buffered Transfers

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

SETTW  $width_p \leftarrow s$  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 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  $d \leftarrow \text{buffercount}_p$  end input

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

OUTPW  $\text{shiftcount}_p \leftarrow \text{bitp}$ ; output part word

$p \triangleleft s$

INPW  $\text{shiftcount}_p \leftarrow \text{bitp}$ ; input part word

$p \triangleright d$

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

## 15.12 Changing Direction

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

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

## 16 Events, Interrupts and Exceptions

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.

SETSR  $sr \leftarrow sr \vee u6$  set thread state

CLRSR  $sr \leftarrow sr \wedge \neg u6$  clear thread state

GETSR  $r11 \leftarrow sr \wedge u6$  get thread state

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

EEBLE	to enable events
IEBLE	to enable interrupts
INENB	to determine if thread is enabling events
ININT	to determine if thread is in interrupt mode
INK	to determine if thread is in kernel mode
SINK	to determine if thread was in kernel mode
WAITING	to determine if thread is waiting to execute the current instruction
FAST	to 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.

SETV	$vector_r \leftarrow s$	set event vector
SETEV	$envector_r \leftarrow s$	set event environment vector
SETD	$data_r \leftarrow s$	set resource data
GETD	$d \leftarrow data_r$	get resource data
SETC	$cond_r \leftarrow s$	set event condition
EET	$enb_r \leftarrow c; thread_r \leftarrow tid$	event enable true
EEF	$enb_r \leftarrow \neg c; thread_r \leftarrow tid$	event enable false
EDU	$enb_r \leftarrow false; thread_r \leftarrow tid$	event disable
EEU	$enb_r \leftarrow true; thread_r \leftarrow tid$	event enable

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

WAITET	$if\ c\ then\ eeble_{tid} \leftarrow true$	event wait if true
WAITEF	$if\ \neg c\ then\ eeble_{tid} \leftarrow true$	event wait if false
WAITEU	$eeble_{tid} \leftarrow true$	event wait

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

```

event   $ed \leftarrow ev_{res};$ 
        $pc \leftarrow v_{res};$ 
        $sr[\text{bit } inenb] \leftarrow false;$ 
        $sr[\text{bit } eeble] \leftarrow false;$ 
        $sr[\text{bit } waiting] \leftarrow false$ 

```

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.

In order to optimise the responsiveness of a thread to high priority resources the SETSR EEBLE instruction can be used to enable events before starting to enable the ports, channels and timers. This may cause an event to be handled

immediately, or as soon as it is enabled. An enabling sequence of this kind can be followed either by a WAITEU instruction to wait for one of the events, or it can simply be followed by a CLRSR EEBLE to continue execution when no event takes place. The WAITET and WAITEF instructions can also be used in conjunction with a CLRSR EEBLE to conditionally wait or continue depending on a guarding condition. The WAITET and WAITEF instructions can also be used to optimise the common case of repeatedly handling events from multiple sources until a terminating condition occurs.

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

```
CLRE  eebletid ← false;           disable all events
      inenbtid ← false;           for thread
      forall res
        if (threadres = tid ∧ eventres) then enbres ← false
```

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

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:

```
int  spc ← pc;
      ssr ← sr;
      pc ← vres;
      sed ← ed;
      ed ← evres
      sr[bit inint] ← true
      sr[bit ink] ← true;
      sr[bit eeble] ← false;
      sr[bit ieble] ← false
      sr[bit waiting] ← false
```

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 ← traptyp;
       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  $\neg e$  then {          error on false
        spc  $\leftarrow$  pc;
        ssr  $\leftarrow$  sr;
        et  $\leftarrow$  error;
        sed  $\leftarrow$  ed;
        ed  $\leftarrow$  s;
        pc  $\leftarrow$  kep;
        sr[bit ink]  $\leftarrow$  true;
        sr[bit eeble]  $\leftarrow$  false;
        sr[bit ieble]  $\leftarrow$  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  $\leftarrow$  pc;          kernel call immediate
        ssr  $\leftarrow$  sr;
        et  $\leftarrow$  kernelcall
        sed  $\leftarrow$  ed
        ed  $\leftarrow$  u6;
        pc  $\leftarrow$  kep + 64;
        sr[bit ink]  $\leftarrow$  true;
        sr[bit ieble]  $\leftarrow$  false;
        sr[bit eeble]  $\leftarrow$  false

```

```

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

```

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

LDSPC	$spc \leftarrow mem[sp + 1 \times B_{pw}]$	load exception pc
STSPC	$mem[sp + 1 \times B_{pw}] \leftarrow spc$	store exception pc
LDSSR	$ssr \leftarrow mem[sp + 2 \times B_{pw}]$	load exception sr
STSSR	$mem[sp + 2 \times B_{pw}] \leftarrow ssr$	store exception sr
LDSED	$sed \leftarrow mem[sp + 3 \times B_{pw}]$	load exception data
STSED	$mem[sp + 3 \times B_{pw}] \leftarrow sed$	store exception data
STET	$mem[sp + 4 \times B_{pw}] \leftarrow et$	store exception type

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

GETET	$r11 \leftarrow et$	get exception type
GETED	$r11 \leftarrow ed$	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.

KENTSP	$n$	$mem[ksp] \leftarrow sp;$	switch to kernel stack
		$sp \leftarrow ksp - n \times B_{pw}$	
KRESTSP	$n$	$ksp \leftarrow sp + n \times B_{pw};$	switch from kernel stack
		$sp \leftarrow mem[ksp]$	

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 initialised and read using the SETKSP and GETKSP instructions.

SETKEP	$kep \leftarrow r11$	set kernel entry point
SETKSP	$ksp \leftarrow r11$	set kernel stack pointer
GETKSP	$r11 \leftarrow ksp$	get kernel stack pointer

## 17 Initialisation and Debugging

The state of the processor includes some registers in addition to those used for the threads.

### register use

*dspc*     debug save pc  
*dssr*     debug save sr  
*dssp*     debug save sp  
*dtype*    debug cause

*dtid*     thread identifier used to access thread state  
*dtreg*    register identifier used to access thread state

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

All debugging is handled using thread 0. The debug handler is entered automatically as a result of conditions detected by hardware (breakpoints and watchpoints), as a result of external debug requests or by debug calls (breakpoints) inserted into the program.

Automatic entry to a debug handler operates in a manner similar to an interrupt:

```

debug   $dspc \leftarrow pc_{t0};$ 
         $dssr \leftarrow sr_{t0};$ 
         $pc_{t0} \leftarrow debugentry$ 
         $dtype \leftarrow cause$ 
         $sr_{t0}[\text{bit } inint] \leftarrow true$ 
         $sr_{t0}[\text{bit } ink] \leftarrow true;$ 
         $sr_{t0}[\text{bit } eeble] \leftarrow false;$ 
         $sr_{t0}[\text{bit } ieble] \leftarrow false$ 
         $sr_{t0}[\text{bit } waiting] \leftarrow false$ 

```

The DCALL instruction has the same effect:

```

DCALL   $dspc \leftarrow pc_{t0};$           debug call (breakpoint)
         $dssr \leftarrow sr_{t0};$ 
         $pc_{t0} \leftarrow debugentry$ 
         $dtype \leftarrow dcallcause$ 
         $sr_{t0}[\text{bit } inint] \leftarrow true$ 
         $sr_{t0}[\text{bit } ink] \leftarrow true;$ 
         $sr_{t0}[\text{bit } eeble] \leftarrow false;$ 
         $sr_{t0}[\text{bit } ieble] \leftarrow false$ 

DRET    $pc_{t0} \leftarrow dspc;$           return from debug
         $sr_{t0} \leftarrow dssr;$ 

DENTSP  $dssp \leftarrow sp;$            debug save stack pointer
         $sp \leftarrow ramend$ 

DRESTSP  $sp \leftarrow dssp$           debug restore stack pointer

```

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

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

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  $B = 2^{bpw}$ . The two carry-in operands allow the component results of multi-length multiplications to be formed directly without the need for extra addition steps.

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

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

LDIV  $d \leftarrow (l_{<<bpw} + m) \div r$ ;            long divide unsigned  
 $e \leftarrow (l_{<<bpw} + m) \text{ mod } r$

The instruction traps if the result can not 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:

<p>MACCU <math>s \leftarrow ((l \times r) + s_{&lt;&lt;bpw} + t)[\text{bits } bpw \text{ for } bpw];</math>  <math>t \leftarrow ((l \times r) + t)[\text{bits } 0 \text{ for } bpw]</math></p>	<p>long multiply accumulate unsigned</p>
<p>MACCS <math>s \leftarrow ((l \times_{sgn} r) + s_{&lt;&lt;bpw} + t)[\text{bits } bpw \text{ for } bpw];</math>  <math>t \leftarrow ((l \times_{sgn} r) + t)[\text{bits } 0 \text{ for } bpw]</math></p>	<p>long multiply accumulate signed</p>

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:

<p>CRC for <math>step = 0</math> for <math>bpw</math>          if <math>(r[\text{bit } 0] = 1)</math>          then <math>r \leftarrow (s[\text{bit } step] : r[\text{bits } (bpw - 1) \dots 1]) \oplus p</math>          else <math>r \leftarrow (s[\text{bit } step] : r[\text{bits } (bpw - 1) \dots 1])</math></p>	<p>word cyclic redundancy check</p>
<p>CRC8 for <math>step = 0</math> for 8          if <math>(r[\text{bit } 0] = 1)</math>          then <math>r \leftarrow (s[\text{bit } step] : r[\text{bits } 31 \dots 1]) \oplus p</math>          else <math>r \leftarrow (s[\text{bit } step] : r[\text{bits } 31 \dots 1]);</math>  <math>d \leftarrow s \gg 8</math></p>	<p>8 step cyclic redundancy check</p>

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 Further Reading

A full listing of the individual instructions can be found in the XS1 Instruction Set Architecture Document [1].

## References

- [1] David May and Henk Muller. XMOS XS1 Instruction Set Architecture. Website, 2008. <http://www.xmos.com/published/xs1inst87>.

XMOS Ltd is the owner or licensee of this design, code, or Information (collectively, the “Information”) and is providing it to you “AS IS” with no warranty of any kind, express or implied and shall have no liability in relation to its use. XMOS Ltd makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

(c) 2008 XMOS Limited - All Rights Reserved