

# XCORE XS1 Architecture Tutorial

---

(VERSION 1.1)



2009/6/22

*Authors:*

DAVID MAY  
HENK MULLER

Copyright © 2009, XMOS Ltd.  
All Rights Reserved

## 1 Introduction

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.

Each XCore has hardware support for executing several concurrent threads. Each thread has access to a private set of registers. All threads share access to all other resources available on the core.

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 by allowing some threads to continue whilst others are waiting for communication to or from remote cores.

Sequential code (Section 3) uses a standard 3-operand load-store instruction set. The instruction set has arithmetic operations on registers, can transfer data to and from memory, and has branch and procedure calling instructions. Concurrency and other features are implemented using *resources* (Section 4). Resources implement single instruction control over threads, locks and channels (Section 5), and timing and I/O (Section 6). Resources interact with the thread scheduler by means of interrupts and events (Section 7).

## 2 Data and Storage

The XCore instruction set operates on *words* of data. The instruction set is independent from the word-length, in that arithmetic, memory and I/O instructions operate on a whole word. Where required, explicit instructions deal with 8- and 16-bit values. In this document we assume that a word comprises  $bpw$  bits, or  $Bpw$  bytes;  $bpw = 8Bpw$ .

### 2.1 Memory architecture

The XCore uses a unified memory architecture; a single address space is used to address both data and program code. The address space accesses an on-chip RAM that holds user program code and user data, and a small ROM that holds the code that boots the XCore. A word of data can be accessed in a single clock cycle, and hence there are no caches needed in the system.

Input output ports are not memory mapped, and are accessed using special instructions, described in Section 6. User programs are usually read in from either a one-time programmable memory (OTP) or from a flash memory. Both are accessed using input/output ports. They are discussed in the System manual [1].

### 2.2 Registers

The normal state of a thread is represented by twelve *operand registers*, four *access registers* and the *program counter*. The twelve operand registers  $r0 \dots r11$  hold a word of data each, and are used by instructions that perform arithmetic operations, access data structures, and call subroutines. When describing instructions  $r$ ,  $s$ ,  $d$ ,  $e$ ,  $x$ , and  $y$  all denote operand registers.

The access registers store addresses in memory. There are instructions that initialise or adjust the access registers. They contain base addresses that the compiler (or assembler programmer) can use to store constants, global data, and a stack. The fourth access register holds the return address for procedure calls:

---

register	number	use
<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 program counter holds the address of the instruction that has to execute next; it is denoted *pc*. It is not manipulated other than by branch instructions.

In addition, each thread has seven additional registers which have specific uses that will be discussed in Section 7.4 on kernel calls, interrupts, and exceptions.

### 2.3 Instruction encoding

Most instructions are encoded in 16-bit instructions, with up to 3 operands. Three operand-instructions operate on either three general purpose registers, or on two general purpose registers and a small constant in the range 0 ... 11, denoted  $u_5$ . Two operand instructions may have an immediate operand that allows for slightly larger constants (0 ... 63, denoted  $u_6$ ), and one operand instructions (for example procedure calls) use 10-bit constants denoted  $u_{10}$ . The 6-bit and 10-bit immediates can be prefixed with an additional 10 bits in order to extend the range of operands to 16 and 20 bits. We use  $u_{16}$  and  $u_{20}$  to denote operands that can be extended to 16 and 20 bits.

In order to densely encode instructions, some instructions use *r11* as their source or destination operand, typically where a temporary value is used in a sequence of two instructions. Less frequently used instructions are encoded using a prefix and hence occupy 32 bits.

### 2.4 Instruction access

Each thread has a 64 bit instruction buffer which is able to hold four short instructions or two long ones. The processor pipeline allows each thread to, in turn, access memory and read or write a word of data. If the thread is not executing a load or store instruction, then the thread will use this pipeline slot to top-up the instruction buffer with the next word of instructions.

Typically over 80% of instructions executed are 16-bit; given a 32-bit wide mem-

ory the XS1 processor fetch two instructions every cycle. As typically less than 30% of instructions require a memory access, the processor can run most programs at full speed using a unified memory system.

### 3 Sequential execution

#### 3.1 Arithmetic

Most arithmetic operations execute in a single clock cycle. Operations that frequently require an immediate operand, have an immediate version that allows a small constant in the range 0 to 11. Arithmetic instructions operate on words of data, the result is the least significant word; overflow is ignored.

ADDI	$d, x, u_8$	add immediate
ADD	$d, x, y$	add
SUBI	$d, x, u_8$	subtract immediate
SUB	$d, x, y$	subtract
NEG	$d, x$	negate
MUL	$d, x, y$	multiply

If larger constants are required, or an operation is used that does not have an immediate version, then the load-constant instruction is used to load a constant into a register. This instruction accepts constants up to 16 bits long. Longer constants can be constructed arithmetically, or they can be stored in memory, for example the constant pool - discussed in Section 3.2.

LDC	$d, u_{16}$	Load constant
-----	-------------	---------------

Four comparison instructions compare two words, and result in a boolean *true* or *false*, represented by the words 1 (true) and 0 (false) . The comparison instructions are 3-operand instructions, comparing two values, and storing the result in the destination register.

EQI	$d, x, u_8$	equal immediate
EQ	$d, x, y$	equal
LSU	$d, x, y$	less than unsigned
LSS	$d, x, y$	less than signed

Bitwise operations are provided in order to manipulate bit patterns stored in a word. The first three operations can also operate on boolean values (false and

true, defined above); the NOT instruction inverts all bits in a word and is hence not suitable for a boolean negation. In the unusual case where boolean negation is required it has to be performed by two instructions, NEG followed by ADDI.

AND	$d, x, y$	and
OR	$d, x, y$	or
XOR	$d, x, y$	exclusive or
NOT	$d, x$	not

Bitwise shift instructions are supplied in both immediate and register versions. The immediate versions allow the values 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32, and  $bpw$ , enabling shifts to shift one or more bytes, or a small number of bits. The arithmetic shifts sign extend the result; the logical shifts always shift a zero in.

SHLI	$d, x, u_s$	logical shift left immediate
SHL	$d, x, y$	logical shift left
SHRI	$d, x, u_s$	logical shift right immediate
SHR	$d, x, y$	logical shift right
ASHRI	$d, x, u_s$	arithmetic shift right immediate
ASHR	$d, x, y$	arithmetic shift right

Four instructions perform division and remainder; these instructions take more than a single cycle to complete.

DIVU	$d, x, y$	divide unsigned (multi-cycle)
DIVS	$d, x, y$	divide signed (multi-cycle)
REMU	$d, x, y$	remainder unsigned (multi-cycle)
REMS	$d, x, y$	remainder signed (multi-cycle)

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	$d, e, x, y, z$	add with carry
LSUB	$d, e, x, y, z$	subtract with borrow

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  $d, e, w, x, y, z$  long unsigned multiply

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  $d, e, x, y, v$  long divide unsigned

The instruction traps if the result can not be represented as a single word value; this occurs when  $y \leq v$ . 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:

MACCU  $d, e, x, y$  long multiply accumulate unsigned  
 MACCS  $d, e, x, y$  long multiply 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  $d, x, p$  8 step cyclic redundancy check  
 CRC8  $d, e, x, p$  word cyclic redundancy check

The CRC8 instruction operates on the least significant 8 bits of its data operand, ignoring the most significant 24 bits. It is useful when operating on a sequence of bytes, especially where these are not word-aligned in memory.

The final instructions perform bit and byte manipulation. They can be used to reverse all bits in a word, all bytes in a word, or all bits in all bytes in a word:

BITREV	$d, x$	bit reverse
BYTEREV	$d, x$	byte reverse
CLZ	$d, x$	count leading zeros

use a sequence (BYTEREV; BITREV) to reverse the bits in each byte of the word. CLZ can be used to detect the first set bit.

### 3.2 Data Access

If data is to be stored in memory, load and store instructions must be used to transfer data between registers and memory. Memory access is always performed relative to some base address. This base address can be the stack-pointer, the data-pointer, the constant pointer, or a general purpose register. Operations are provided to load and store data, and to compute the address of a location in memory.

Variables that are local to a procedure are normally stored in registers, but a stack-pointer is provided to easily build a stack. The stack pointer is designed to grow downwards, with register *sp* pointing to the lowest stack item in memory; this item is at the *top* of the stack. Instructions to extend and contract the stack are discussed in Section 3.3 on procedure calls. Accesses to the stack are performed using instructions that take a destination register and a 16-bit word-offset, allowing a stack frame of up to 64 Kwords. Most stack frames (up to 64 words long) can be accessed using short 16-bit instructions.

LDWSP	$d, u_{16}$	load word from stack
STWSP	$s, u_{16}$	store word to stack
LDAWSP	$d, u_{16}$	load address of word in stack

The data pointer can be used to point to the area of memory that holds global variables for this thread. The base of this area can be held in the *dp* register. Unlike the stack pointer, the data pointer is normally not moved. Instructions to access memory relative to the data pointer are carbon copies of instructions that access data on the stack:

LDWDP	$d, u_{16}$	load word from data
STWDP	$s, u_{16}$	store word to data
LDAWDP	$d, u_{16}$	load address of word in data

A third section of memory is provided that can be used to hold large constants (larger than can be used with immediate versions of instructions or LDC). The base of the constant pool is stored in the *cp* register. There is no instruction to store data in the constant pool. Note that loading data from the constant pool involves memory access, and may be slower than loading data using an LDC instruction.

LDWCP	$d, u_{16}$	load word from constant pool
LDWCPL	$u_{20}$	load word from constant pool into <i>r11</i>
LDAWCP	$d, u_{16}$	load word address in constant pool

If constants, such as branch tables, are stored in the program itself, then their address is computed using one of the two instructions below. One instruction computes a forward address, one computes a backward address. Both take a 20-bit word offset, allowing an 8 Mbyte range to be addressed.

LDAPF	$u_{20}$	load address in program forward into <i>r11</i>
LDAPB	$u_{20}$	load address in program backward into <i>r11</i>

Access to data structures is provided by instructions which use any of the operand registers as a base address, and combine this with an offset that is scaled so that it addresses word *i* counted from the base address. The offset can either be an immediate, or it can be stored in a register. The former case is for accessing data in a struct, the latter is for accessing data in an array.

LDWI	$d, b, u_s$	load word
STWI	$s, b, u_s$	store word
LDW	$d, b, i$	load word
STW	$s, b, i$	store word
LDAWFI	$d, b, u_s$	load address of word forward immediate
LDAWBI	$d, b, u_s$	load address of word backward immediate
LDAWF	$d, b, i$	load address of word forward
LDAWB	$d, b, i$	load address of word backward

The base-addresses must be word-aligned, otherwise an exception will be raised (Section 7.3). If required, bound checks can be performed prior to accessing memory, for example when accessing arrays. The instructions to use for this are LSU (Section 3.1) and ECALLF (Section 7.3).

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 base address must be zero. The 16-bit item is loaded and sign extended into

a 32-bit value.

LD16S	$d, b, i$	load 16-bit signed item
ST16	$s, b, i$	store 16-bit item
LDA16F	$d, b, i$	load address of 16-bit item forward
LDA16B	$d, b, i$	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	$d, b, i$	load byte unsigned
ST8	$s, b, i$	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	$d, s$	make mask $2^s - 1$ , 0...01...1
MKMSKI	$d, u_s$	make mask immediate
SEXT	$d, s$	sign extend bits $s$ and higher
SEXTI	$d, u_s$	sign extend immediate
ZEXT	$d, s$	zero extend bits $s$ and higher
ZEXTI	$d, u_s$	zero extend immediate
ANDNOT	$d, s$	and not (clear field)

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

### 3.3 Branching, Jumping and Calling

The XCore branch instructions execute in a single cycle and prefetch the target instruction. One group of branches is designed for control flow, another group of branches is designed for procedure calls. The latter branches copy the program counter into the link register and are called *branch and link*, or *BL*.

Except where stated otherwise, the branch instructions prefetch the target instruction. There is no need for speculative instruction issue and branch prediction, and the branch target will be executed during the next cycle.

The branch instructions include conditional and unconditional relative branches. Conditional branches have two arguments, the register to test for a conditional value, and the offset. A conditional value of 0 is interpreted as false, and a conditional value of any other value is interpreted as true.

BRFT	$s, u_{16}$	branch relative forward true
BRFF	$s, u_{16}$	branch relative forward false
BRBT	$s, u_{16}$	branch relative backward true
BRBF	$s, u_{16}$	branch relative backward false
BRFU	$u_{16}$	branch relative forward unconditional
BRBU	$u_{16}$	branch relative backward unconditional

Branches to an instruction no further than 32 words away are encoded in 16 bits. Branches over a longer distance are encoded in 32 bits. Unconditional branches up to 512 words away may be performed using a procedure call instructions; as they overwrite the link register they are not suitable for use in leaf procedures that do not save the link register.

Two branch instructions support a branch target stored in a register; absolute and relative. The relative branch adds a scaled register operand to the program counter. This can be used to implement jump tables (if each jump target is another jump), or the register can be multiplied prior to the branch if all blocks of code have roughly the same size.

BRU	$s$	branch relative unconditional (via register)
BAU	$s$	branch absolute unconditional (via register)

The procedure calling instructions save the program counter into the link-register ( $lr$ ). The value saved is the address of the instruction that follows the procedure call. The 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	$u_{20}$	branch and link relative forward
BLRB	$u_{20}$	branch and link relative backward
BLACP	$u_{20}$	branch and link absolute via constant pool
BLAT	$u_{16}$	branch and link absolute via table
BLA	$s$	branch and link absolute (via register)

The BLACP and BLAT instructions take two cycles to complete. Control transfers that should not affect the link register (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.

ENTSP	$u_{16}$	extend stack and save <i>lr</i> ( $u_{16} > 0$ )
RETSP	0	restore <i>pc</i> from <i>lr</i>
RETSP	$u_{16}$	contract stack, and restore <i>lr</i> and <i>pc</i>
EXTSP	$u_{16}$	extend stack by $u_{16}$ words
EXTDP	$u_{16}$	extend data by $u_{16}$ words

Note that the data and stack areas can be contracted using the LDAWSP and LDAWDP instructions. Below we show three typical section of code for procedures. From left to right, they are for a leaf procedure call without any variables on the stack, a leaf procedure with five words on the stack, and a non-leaf procedure with five words on the stack (plus one word to save the link register):

```

Proc:          proc:          proc:
               EXTSP    5          ENTSP  6
               ....
               LDAWSP  SP, 5       RETSP  6
RETSP  0       RETSP    0

```

RETSP will take an extra thread cycle in order to prefetch the target instruction if it has to restore the link register from the stack. Ie, RETSP 0 takes a single thread cycle, whereas RETSP 6 takes one extra cycle in order to prefetch the target instruction.

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	<i>s</i>	set stack pointer
SETDP	<i>s</i>	set data pointer
SETCP	<i>s</i>	set pool pointer

## 4 Resources

Each XCore manages a number of different types of *resource*, that will be discussed in subsequent sections. Some of the resources are managed by the XCore, in that it maintains a set of available resources. A resource can be allocated by using 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     $r, u_s$       get a resource of type  $t$ 
FREER    $r$            free resource
```

The resources that can be allocated are:

mnemonic	value	type	use
TIMER	1	timers	timing
CHANEND	2	channel ends	thread communication
SYNC	3	synchronisers	thread synchronisation
THREAD	4	threads	concurrent execution
LOCK	5	locks	mutual exclusion

Other resources that are managed explicitly by the programmer are PORTS and CLOCKBLOCKS; these are discussed in Section 6 on I/O.

GETR will either return the resource-id of the requested resource or, if none is available, it will return the *invalid resource id*, the number 0. Thread-identifiers, port-identifiers, channel identifiers are all examples of resource identifiers. The type of the resource identifier is encoded in the least significant byte and the otherbytes identify the resource itself.

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

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

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:

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

## 5 Concurrency

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 in-core *scheduler* 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 will be 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  $1/p$ .

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.

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.

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.

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

## 5.1 Asynchronous threads

A thread can create other threads by executing a GETR THREAD instruction. This instruction returns either a thread ID if there is a free thread or 0. When a thread is first created, it is in a paused state and its access registers can be initialised using the following instructions.

TINITPC	<i>s, t</i>	set thread pc
TINITSP	<i>s, t</i>	set thread stack
TINITDP	<i>s, t</i>	set thread data
TINITCP	<i>s, t</i>	set thread pool
TINITLR	<i>s, t</i>	set thread link

These instructions can only be used when the thread is paused.

The asynchronous thread is then started by executing a TSTART instruction specifying the thread ID. Once the thread has completed its task it terminates itself with the FREET instruction.

TSTART	<i>t</i>	start thread
FREET		free current thread

Asynchronous threads are used to spawn tasks that are under their own control,

and that need not re-join with the thread that created them. After creating an asynchronous thread, the creator continues execution.

The TINITLR instruction is intended primarily to support debugging. It can also be used to set the *lr* register to point to a FREET instruction (see below), thereby allowing a thread to end with a RETSP instruction.

## 5.2 Synchronised threads

Synchronised threads allow a group of threads to be created which can subsequently perform group synchronisation, and join when all of the threads have completed execution. Thread synchronisation is performed using hardware *synchronisers*, and threads using a synchroniser will move between running states and paused states.

To start a *synchronised* slave thread a master thread 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 repeats this process to create a group of threads which will all synchronise together. The newly allocated threads are initialised in the same way as an asynchronous thread using the TINITPC, TINITSP, TINITDP, TINITCP, TINITLR and TSETR instructions.

GETST *d, s*            get synchronised thread

To start the slave threads the master executes an MSYNC instruction using the synchroniser ID.

MSYNC *s*            master synchronise

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.

SSYNC            slave synchronise

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 terminated, the thread resources freed and the master continues.

MJOIN *s* master join

A typical pattern is for the master thread to create several slaves, then execute a MSYNC to start all threads, and then for all threads to perform a MSYNC/SSYNC before they attempt to use shared resources. The identifier of an executing thread can be accessed by the GETID instruction.

GETID *d* get thread identifier

A complete example that uses MSYNC, SSYNC, GETST, and MJOIN to implement parallel threads is shown in Section [9.1](#).

## 5.3 Communication

There are three methods for threads to communicate data: through registers, through shared memory, and through channels. Registers can only be used if threads execute on the same core and belong to the same synchroniser. Shared memory only works if threads execute on the same core. Channels work both inside and across cores, subject to a suitable network layout.

### 5.3.1 Communication via registers and shared memory

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, s, t* set thread operand register  
TSETMR *d, s* set master thread operand register

In order to prevent race conditions, the TSETR instructions are normally placed between two SSYNC/MSYNC operations. A complete example that uses TSETR, TSETMR, SSYNC, and MSYNC in order to implement thread communication is shown in Section [9.2](#).

### 5.3.2 Channel 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.

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	<i>r, s</i>	set destination
GETD	<i>d, r</i>	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 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.

OUTT	<i>r, s</i>	output data token
OUT	<i>r, s</i>	output data word (4 data tokens, MSB first)
INT	<i>d, r</i>	input data token
IN	<i>d, r</i>	input data word (4 data tokens, MSB first)

In addition to data tokens, channels can carry control tokens. The data and control tokens are received in the same order as they are sent. Control tokens are sent and received using special instructions. If a control-token is input as a data token or vice versa, the thread traps. This enables the implementation of communication protocols. Two application level reserved control tokens are tokens 1 and 2, that signify END and PAUSE respectively. The application can freely use control tokens 0, and 3-127

OUTCT	$r, s$	output control token
OUTCTI	$r, u_s$	output control token immediate
INCT	$d, r$	input control token
CHKCT	$r, s$	check control token
CHKCTI	$r, u_s$	check control token immediate

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 inputter, 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.

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 in a single short instruction. 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).

The END and PAUSE tokens control the state of the channel. Channels are in one of two states, connected and disconnected. By default a channel is disconnected. 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.

Instead of END, a channel can be disconnected by outputting a PAUSE control token. In contrast with END, 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.

While two channel ends are connected, the resulting channel occupies a path between them. This is of no consequence when the channel ends are on the

same core, but if they are on different cores, this path will occupy two *internal* links and zero or more *external* links. Each core has four *internal* links connecting it to the local switch. Each switch has up to 16 external links that connect the switch to other nodes. The total number of messages that can be in flight simultaneously between any two nodes is limited by the number of available internal links and external links. More information on configuring the network is available in the XS1 System Specification [1]; this includes details on message routing.

A destination channel end can be shared by any number of outputting threads using different source channel ends. If multiple messages arrive simultaneously on a destination channel end 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.

A control token takes up a single token 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).

TESTCT	<i>d, r</i>	test for control token
TESTWCT	<i>d, r</i>	test word for 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. A channel-end

should only be freed if it is disconnected; that is, if it has been used for input the last token input should have been END, and if it has been used for output, the last output should have been END.

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  $d, r$             test destination local

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      $r, s$             set network  
GETN      $d, r$             get network

If networks are being used, then internal links and external links must be dedicated to the network. The process of setting up dedicated networks in the inter-connected is detailed in the XS1 System Specification [1].

Like any other resource, 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.

## 5.4 Locks (mutual exclusion)

Mutual exclusion between a number of threads on a core can be guaranteed using *locks*. A lock is allocated using a GETR  $r$ , 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 *r* instruction.

## 6 Input Output

External devices are not memory mapped, but are controlled using OUT and IN instructions on *ports*, resources that connect to the I/O pins of the XCore. Timing of I/O can be performed using *timer* resources, or using an external clock on the port.

### 6.1 Timers

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

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.

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

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

## 6.2 Ports

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.

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	$r, d$	output data and shift
INSHR	$d, r$	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.

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.

### 6.3 Clock blocks

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  $r, s$             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  $f/2n$ . 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.

## 7 Events, Interrupts, Exceptions, Kernel calls

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

### 7.1 Events

A thread normally enables one or more events and then waits for one of them to occur. Hence, on an event all the thread's state is in a predefined state, allowing the thread to respond rapidly to the event. The thread can perform input and output operations on 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 the next 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	<i>r</i>	set event vector from <i>r11</i>
SETEV	<i>r</i>	set event environment vector from <i>r11</i>
SETD	<i>r, s</i>	set resource data
GETD	<i>d, r</i>	get resource data
SETC	<i>r, s</i>	set event condition
SETCI	<i>r, u<sub>16</sub></i>	set event condition immediate
EET	<i>s, r</i>	event enable if condition true
EEF	<i>s, r</i>	event enable if condition false
EDU	<i>r</i>	event disable
EEU	<i>r</i>	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	<i>r</i>	event wait if true
WAITEF	<i>r</i>	event wait if false
WAITEU		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. Alternatively the thread may be paused until an event takes place; when the event takes place the thread resumes execution at the corresponding event vector. When woken up, a one-cycle delay is incurred in order to fetch the target instruction.

On wake-up the environment vector (set with SETEV) 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.

WAITEU can be used after a block of code to optimise the common case where events are dispatched continuously. 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.

CLRE                      disable all events for thread

## 7.2 Status register, save-registers

Each thread has a set of 8 registers that controls the way that the thread operates, and that can save data in the case of exceptions (Section 7.3) and kernel calls (Section 7.4). These registers are:

<b>register</b>	<b>use</b>
<i>sr</i>	the status register
<i>spc</i>	the saved pc
<i>ssr</i>	the saved status
<i>et</i>	the exception type
<i>ed</i>	the exception data
<i>sed</i>	the saved exception data
<i>kep</i>	the kernel entry pointer
<i>ksp</i>	the kernel stack pointer

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 a bitmask.

SETSR	$u_{16}$	set thread state
CLRSR	$u_{16}$	clear thread state
GETSR	$u_{16}$	get thread state

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

EEBLE	0x01	events enabled
IEBLE	0x02	interrupts enabled
INENB	0x04	thread is enabling events
ININT	0x08	thread is in interrupt mode
INK	0x10	thread is in kernel mode
SINK	0x20	thread was in kernel mode
WAITING	0x40	thread is waiting to execute the current instruction
FAST	0x80	thread is in fast mode

The ININT, INK, SINK, and WAITING are normally only set by the processor and read by the user code. The other five modes can be set as follows:

**EEBLE** 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. EEBLE is reset on CLRE and WAIT instructions.

**IEBLE** Enables interrupts. Each resource can be set to either generate events or interrupts. If the resource is set to generate interrupts, and the IEBLE bit is set, then the processor will jump to the event vector. In contrast to events, interrupts can occur at any point during program execution, and so the current *pc*, *sr*, and *ed* (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*, *ssr*, and *sed* registers. When in an interrupt INK and ININT are set in the status register, and EEBLE, IEBLE and WAITING are reset. The ED register is set to the environment data for that resource.

When the handler has completed, continuation of the interrupted thread can be performed by a KRET instruction, which restores *pc*, *sr*, and *ed* from *spc*, *ssr*, and *sed*.

**INENB** 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.

**ININT** is set to record that the thread is handling an interrupt.

**INK** is set to record that the thread is in kernel mode.

**SINK** is set to save the kernel mode bit.

**WAITING** is set if this thread is presently not ready to execute code. Usually this means that the thread is waiting for events (the EEBLE bit will be set), for

interrupts (the IEBLE bit is set), or waiting for the master thread to start it. A thread can set this bit in order to deschedule itself and wait, for example for an interrupt.

**FAST** Normally, a thread that is waiting for an event is rescheduled when the event completes, the instruction is re-executed. 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 descheduled when an instruction can not complete; instead the instruction is re-issued until it completes.

### 7.3 Exceptions

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

The *et* and *ed* registers are being set to the type of exception that occurred (eg, an unaligned memory operation) and to the data that caused the exception (the offending address).

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

ECALLT	<i>r</i>	exception on true
ECALLF	<i>r</i>	exception on false

### 7.4 Kernel call

A program can explicitly cause entry to a kernel 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; the program counter is set to *kep* + 64. The exception type register, *et*, is set to 15, and *ed* is set to the operand of KCALL.

KCALLI	$u_{16}$	kernel call immediate
KCALL	$r$	kernel call

## 7.5 Kernel, Exception, and interrupt handling

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 is then used to restore the stack pointer on exit from the handler.

KENTSP	$u_{16}$	switch to kernel stack
KRESTSP	$u_{16}$	switch from kernel stack

The  $spc$ ,  $ssr$ ,  $et$  and  $sed$  registers can be saved and restored directly to the stack. These instructions store those values at fixed offsets relative to the stack pointer. The offsets have been chosen so that the stack frame can be sized to store just the PC, or more information:

STSPC	store saved pc (offset 1)
STSSR	store saved sr (offset 2)
STSED	store saved exception data (offset 3)
STET	store exception type (offset 4)
LDSPC	load saved exception pc (offset 1)
LDSSR	load saved exception sr (offset 2)
LDSED	load saved exception data (offset 3)

In addition, the  $et$  and  $ed$  registers can be transferred directly into  $r11$ , if an exception handler is designed to just inspect the reason for an exception:

GETET	get exception type
GETED	get exception data

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. Note that the kernel entry point has to be aligned on a 64-byte boundary - the last 6 bits of the address are cleared.

SETKEP	set kernel entry point to r11
GETKSP	get kernel stack pointer into r11

## 8 Predicting and optimising performance

In order to minimise execution time, the assembly code should be designed so that it minimises the number of cycles inserted to fetch instructions. As explained in Section 2.4, the pipeline either accesses memory to execute a load/store, or it accesses the memory to prefetch a word of instructions.

- Sequences of loads and stores will be interspersed with fetch-nop cycles. In order to speed up this sequence, loads and stores should where possible be mixed with *short* instructions that do not require memory access.
- If a branch target is not word aligned, then the branch instruction will only be able to fill a small fraction of the prefetch buffer. Aligning the targets will maximise the branch target prefetch. This also goes for the return addresses of procedure calls.
- If a 32-bit instruction is not aligned, then there is a possibility that only the first half of the instruction is in the buffer; which causes a fetch-nop. Where possible, 32-bit instructions should be word aligned.
- Constants that are too large to fit in 6 bits but fit in 16 bits can be loaded either using the constant pool or a long LDC instruction. If code size is important, it may be more effective to load it via the constant pool, but if performance is important, it should be loaded using a long LDC.

Alignment can often be achieved by prefixing a short instruction. For example, “LDC r1, 0” can be encoded as 16 or 32 bits, and many of the procedure call instructions can be encoded in 16 or 32-bits in order to align the return address.

## 9 Example code sequences

In this section we give some example code sequences to show how to use combinations of instructions.

## 9.1 Implementing PAR using GETST, MSYNC, SSYNC, MJOIN

In conjunction with the SYNC instructions the GETST instruction can be used to implement fork-join concurrent threads, and to optimise on thread creation by keeping threads alive for subsequent use.

For example, the following high level sequence:

```
par {
  A;
  B;
}
E;
par {
  C;
  D;
}
F;
```

Is translated as:

```
GETR    r0,3
GETST   r1,r0
LDAPF   threadB
TINITPC r1
MSYNC   r0
... A ...
MSYNC   r0
... E ...
MSYNC   r0
... C ...
MJOIN   r0
... F ...

threadB:
... B ...
SSYNC
... D ...
SSYNC
```

## 9.2 Implementing communication using TSETR, TSETMR

The TSETR and TSETMR instructions are used to communicate data between threads. They should only be used when the threads have synchronised in order to avoid race conditions. The synchronisation can be achieved by means of a synchroniser, or locks. As an example, the high level code below:

```
int i, j;
i = 3;
j = 4;
par {
    i = i + 5;
    j = j + 6;
}
j = i;
par {
    i = i + 7;
    j = j + 8;
}
```

is translated as:

```
LDC    r6, 3      ; Variable i
LDC    r7, 4      ; Variable j
GETR   r0, 3      ; Get a synchroniser
GETST  r1,r0      ; Get a thread for this synchroniser
LDAPF  threadB
TINITPC r1        ; initialise the thread PC
TSETR  r7, r1, r7 ; Initialise j of the new process
MSYNC  r0
ADD    r6, r6, 5  ; i = i + 5;
MSYNC  r0
TSETR  r7, r6, r1 ; j = i
MSYNC  r0
ADD    r6, r6, 7  ; i = i + 7
MSYNC  r0
MJOIN  r0

threadB:
ADD    r7, r7, 6  ; j = j + 6
SSYNC
ADD    r7, r7, 8  ; j = j + 8
SSYNC
TSETMR r7, r7     ; update j of other process
SSYNC
```

## 10 Further Reading

A full listing of the individual instructions can be found in the XS1 Instruction Set Architecture Document [2].

## References

- [1] David May and Ali Dixon and Ayewin Oung and Henk Muller. XS1 System Specification. Website, 2008. <http://www.xmos.com/published/xsystem>.
- [2] 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) 2009 XMOS Limited - All Rights Reserved