XS1-L System Specification

(VERSION 0.96)

2010/12/06

Authors:

DAVID MAY
ALI DIXON
AYEWIN OUNG
HENK MULLER
MARK LIPPETT

1 Introduction

This document specifies the XS1-L boot protocol, link specification, switch specification and token specifications. It is related to the L1 and L2 processors which are single and dual core. The XS1-G4, and XS1-G2 specification can be found in the *XS1-G System Specification* document. The core architecture (instruction set) specification can be found in the *XS1 Instruction Set Architecture* document.

Each XS1-L package has a datasheet that contains a pin-out and port-map. In particular, it specifies on which physical pin each *port* and *link* is bonded out, and on which pins the *MODE* signals are bonded out.

2 Booting the XS1-L

The standard boot procedure is to first boot Core 0 from either a Link, JTAG, or an external ROM or Flash memory that is connected via an SPI interface. The boot mode is selected by setting pins MODE3 and MODE2:

00 do not boot; used for booting over JTAG

10 boot from ChanEnd 0, enabling Links C-H

11 boot from SPI

A further option is to use *secure boot* for one or more of the cores. Each core can be configured to boot from a program held in its security module. This is enabled by setting a bit in the core's security module and causes the core to always use secure boot.

2.1 Boot format

When a core is booting over the SPI interface, from a Link, or from the security module, the boot ROM built into the L1 reads in a program and stores it in on-chip RAM starting at the lowest memory location. The program is then started by transferring control to the lowest location in RAM.

The boot format used for the program to boot from an SPI interface, Link or security module is represented as follows:

1. The program size *s* in words - a 32-bit value, least significant byte first.
2. Program consisting of $s \times 4$ bytes.
3. A 32-bit CRC, least significant byte first.

The CRC is calculated over the byte stream represented by the program size and the program itself. The polynomial used is 0xEDB88320 (IEEE 802.3); the CRC register is initialised with 0xFFFFFFFF and the residue is inverted to produce the CRC.

The CRC check can be disabled by setting the CRC to 0x0D15AB1E.

2.2 Boot from SPI interface

To boot from an SPI interface, an SPI slave device must be connected as follows.

Port	Use
P1A0	SPI_MISO
P1B0	SPI_SS
P1C0	SPI_SCLK
P1D0	SPI_MOSI

A READ command is issued with a 24-bit address 0x000000. Based on the 100 MHz reference clock of the XCore, an SPI clock rate of 2.5 MHz is used. The clock polarity / phase is of 0 / 0.

The XCore expects each byte to be transferred with the least-significant bit first. Many programmers write bytes into an SPI interface using the most significant bit first, so you may have to reverse the bits in each byte of the image stored in the SPI device.

If a large boot image is to be read in, it is faster to first load a small boot-loader that reads the large image using a faster SPI clock, for example 50 MHz or as fast as the flash device supports.

If field-upgradeable firmware is required, a small boot-loader should be stored in the first sector of flash memory, followed by two boot-images starting on sector boundaries. The boot-loader should be written to read the first image initially, and on CRC failure boot from the second image. On upgrade, the first image should be upgraded first, followed by the second image. If the upgrade process is interrupted at any point, there is always a working boot image.

2.3 Boot from Link

When boot from Link is selected, the boot ROM enables the Link encoded by SS_XC0_BS2, SS_XC0_BS1, and SS_XC0_BS0 (a binary number between 0 and 7). In addition, all dedicated Links that are not shared with a port are enabled. Most packages bond out only some of the pins SS_XC0_BS, and pins that are not bonded out should be interpreted as zero. A HELLO message is then sent on each enabled Link following the protocol defined in Section 3.2.

To boot a core the following procedure must be followed:

- Allocate a channel-end and connect it to the channel end of the core you want to boot using a SETD instruction. The identifier to be written with SETD typically has the form 0x*zzzz*0002, where *zzzz* is the core-identifier. By default the *allocated* channel-end is called *c*.
- Output the 32-bit identifier *c* of the channel over *c*, allowing the other side to open a back-channel.
- Send the sequence of bytes representing size, code, and CRC as specified above.
- Send an END control token.
- Receive an END control token.
- Free *c* using FREER.

Boot from Link is designed to work on systems where power-up sequences are controlled (i.e. no hot plugging). In systems where the HELLO message could be missed by the core that is trying to send out boot code, it should not send channel end 0x*zzzz*0002 but a null channel-end identifier instead. When booted, the boot-code should resend a HELLO message to open the downstream channel.

2.4 Boot from Security Module

If a core is set to use secure boot, the program in boot format is taken from address 0 of the OTP memory in the core's security module. Each core has its own individual OTP memory, and hence some cores can be booted from OTP while others are booted from SPI or the channel interface. This enables an 1L to be partially programmed, dedicating one or more cores to perform a particular function, leaving the other cores user-programmable.

3 Link specification

The interconnect provides communication between all cores on the system. A system can comprise one or more nodes, that may be physically separated. In conjunction with simple programs, the interconnect can also be used to support access to the memory on any core from any other core, and to allow any core to initiate programs on any other core.

The interconnect allows *streams* of data to be communicated with low latency. A stream comprises *data* tokens and *control* tokens, where data tokens contain 8 bits of data, and control tokens specify operations. Streams are circuit switched, but they can be set-up and terminated at low cost. This enables the network to be used as a packet switching network, where short packets are carried through the interconnect in a pipelined manner.

Each core has four links that connect the core to an on-chip switch that provides non-blocking communication between the cores on a node. The on-chip switch also provides a number off-chip Links that can be connected to Links of other nodes. The structure and performance of the Link connections can be varied to meet the needs of applications. The topology of the interconnect is not fixed, a topology appropriate to the application can be used.

An XS1-L node comprises a *single* core, connected to a switch. The switch has eight links, denoted Link A, B, C, D, E, F, G, and H; typically the datasheet abberviates them as "X0LA" which means Link A of Xcore 0. XS1-L devices with multiple cores, for example an XS1-L2, have one switch for each core. Each node may have some links bonded out (denoted for example X0LA, or X3LB), and the switches are internally connected using some links (typically links E, F, G, and H). Note that this is different from the XS1-G series where each node

comprises a switch with four cores; where the switch has 16 internal and 16 external links.

The network supports partitioning. For example, partitioning provides separation between data intensive streams and control streams. Partitioning provides real time guarantees for parts of the network that need the guarantees.

As far as a program is concerned, communication always takes place between two *channel ends*. A channel end is a physical resource that is allocated on the XCore. Channels-ends reside on a core and are identified by means of an identifier on the core, a core-identifier, and a node-identifier. Data is transmitted to a channel end by using a sequence of OUT and OUTCT instructions; when a communication is complete, an END token is transmitted by the program, which frees up any resources allocated in the network. The architecture guarantees that all data-tokens and control-tokens sent over this stream are delivered in order. Multiple streams can be set up, no guarantee is given about the ordering of data and control tokens between streams.

This document describes the stream (the transport layer), the switching method (the packet layer), the point-to-point protocol (the link layer) and the physical layer.

There are four groups of control tokens:

- Tokens 0x00-0x7f: (Application tokens). Intended for use by compilers or applications software to implement streamed, packetised and synchronised communications, to encode data-structures and to provide run-time type-checking of channel communications.
- Tokens 0x80-0xbf: (Special tokens) Architecturally defined and may be interpreted by hardware or software. They are used to give standard encodings of common data types and structures.
- Tokens 0xc0-0xdf: (Privileged tokens) 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 0xe0-0xff: (Hardware tokens) 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.

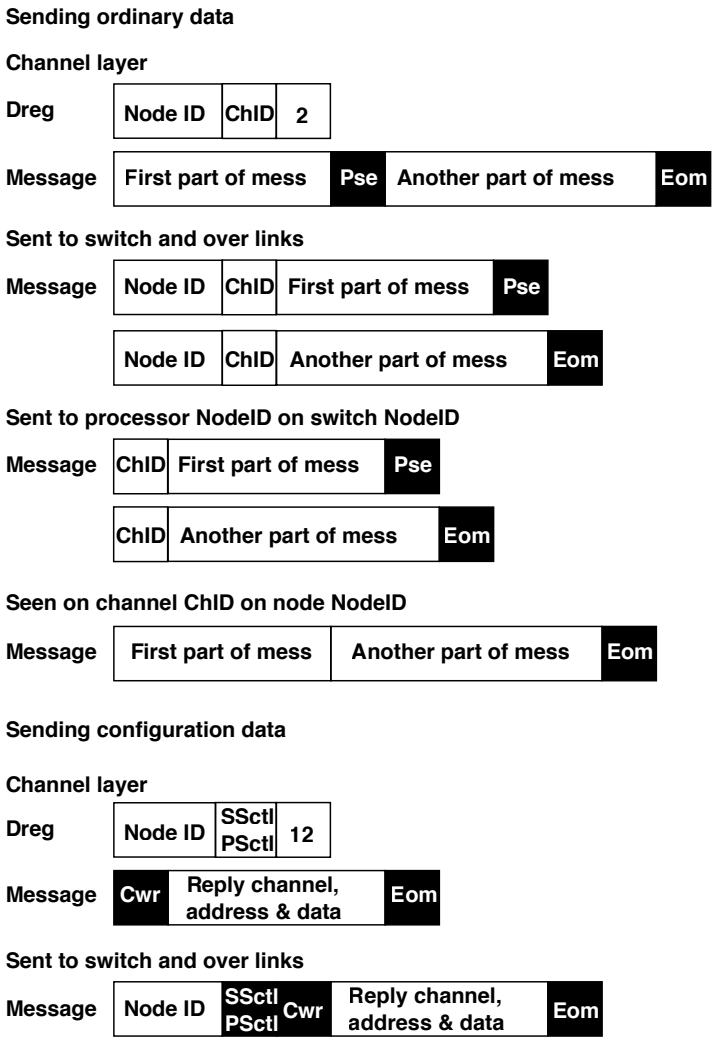

Figure 1 *Summary: tokens transferred for ordinary and configuration data. Control tokens are white on black.*

The sections below define the protocol layers bottom up: physical layer (Section 3.1), link layer (Section 3.2), switch layer (Section 3.3), physical layer (Section 3.1), processor communication (Section 3.5), and channel communication (Section 3.6). A summary is provided in Figure 1.

3.1 Physical layer

Link communication uses 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. All links have a weak pull down, but an external pull down may be required to avoid spurious transitions on reset.

The Links can be switched between between a slow serial mode that uses four wires, and a fast, wide mode that needs 10 wires. These two modes use different encoding schemes.

3.1.1 Serial Link

The serial Link uses two data wires, “0” and “1” in each direction (four wires in total). A transition on wire “0” represents a zero bit and a transition on wire “1” represents a one bit. Note that it is the transition that signals the bit; the level of the wire is irrelevant.

For each token 10 transitions are made, transmitting 10 bits. The first 8 bits are the token value. transmitted most significant bit first. The next bit signals whether the transmission is a *control* or a *data* token. A 1-bit signals a control-token, a zero-bit signals a data-token. The final bit is an even parity bit that causes both wires to go back to low state when transmitted. The two signal wires are both at rest between tokens.

For example, to send control token 0x09, transmit the following:

1. Set wire “0” to high (signals a zero in bit 7)
2. Set wire “0” to low (signals a zero in bit 6)
3. Set wire “0” to high (signals a zero in bit 5)
4. Set wire “0” to low (signals a zero in bit 4)
5. Set wire “1” to high (signals a one in bit 3)
6. Set wire “0” to high (signals a zero in bit 2)
7. Set wire “0” to low (signals a zero in bit 1)
8. Set wire “1” to low (signals a one in bit 0)

9. Set wire “1” to high (signals control token)
10. Set wire “1” to low (terminate transmission - both wires are in rest state)

3.1.2 Fast Link

The fast Link uses five wires in each direction to transmit data; 10 wires in total. The wires are called “0”, “1”, “2”, “3”, and “4”. One of fives codes are used to transmit data; changing the state of one of the five wires transmits a *symbol*. A transition on each of the wires has the following meaning:

Transition on	Symbol	Meaning
“0”	*v*	value 00
“1”	*v*	value 01
“2”	*v*	value 10
“3”	*v*	value 11
“4”	*e*	escape

A sequence of four symbols are used to encode the data and control tokens. If all four symbols are data *v* symbols, a total of 8 bits of data are transferred (a data-token). If one of the four symbols is an *e* symbol, and the other three are *v* symbols, a control-token is transmitted.

Transitions				Use
first	second	third	fourth	
v	*v*	*v*	*v*	256 data tokens
e	*v*	*v*	*v*	64 control tokens 192-255
v	*e*	*v*	*v*	64 control tokens 128-191
v	*v*	*e*	*v*	64 control tokens 64-127
v	*v*	*v*	*e*	64 control tokens 0-63

The bits of data and control tokens are always transmitted starting with the two most significant bits. In the case of control tokens, the first two bits of the control token are determined by the position of the *e* symbol.

For example, to send control token 0x09, transmit the following:

1. Set wire “0” to high (signals 00 bits in bits 5 and 4)

2. Set wire “2” to high (signals 10 bits in bits 3 and 2)
3. Set wire “1” to high (signals 01 bits in bits 1 and 0)
4. Set wire “4” to high (signals an escape, bits control token bits 6 and 7 are 0)

After transmitting one token, none, two, or four wires are high. Wires are only returned to zero when a message is completed (when data is streamed wires do not return to zero). To return to zero, a sequence of an END token and an optional return-to-zero-NOP are transmitted. They are chosen so that after them all wires are low.

Transitions				**Use**
first	second	third	fourth	
e	*e*	*v*	*v*	END tokens
e	0	*e*	0	CREDIT8 token
e	1	*e*	1	CREDIT64 token
e	2	*e*	2	HELLO token
e	3	*e*	3	CREDIT16 token
v	*v*	*e*	*e*	PAUSE tokens
e	*v*	*v*	*e*	NOPD tokens
e	3	3	*v*	NOPE tokens (control tokens 252...255)

There are sixteen possible sequences to transmit an END token on the 5-wire Link. All of them signal END; but by choosing the appropriate sequence, it can be guaranteed that none, one, or two of wires “0” to “3” are left high. After the END token, one of the NOP tokens may have to be transmitted in order to return the final wires to zero.

- If wire “4” is high after the END token, the NOPE token is transmitted, and the final transition is chosen to return the last high wire low (note that if wire “4” is high, exactly one of wires “0”... “3” must be high).
- If wire “4” is low after the END token, the NOPD token is transmitted. The NOPD token has two transitions on wire “4” and hence leaves wire “4” low. The two *v* transitions are chosen to return the final two wires to low.

For example, to send an end-of-message after the control token sent earlier (wires “0”, “1”, “2”, and “4” are high), transmit the following:

1. Set wire “4” to low (signals an escape).
2. Set wire “4” to high (signals a second escape, this is an END).
3. Set wire “0” to low.
4. Set wire “1” to low. Sends the END token, and only wires “4” and “2” are left high; hence, a NOPE token must be transmitted.
5. Set wire “4” to low (signals an escape).
6. Set wire “3” to high (transmits token value 11).
7. Set wire “3” to low (transmits token value 11).
8. Set wire “2” to low (transmits token value 10). This has transmitted token 254, which is a NOP token that is ignored by the receiver. All wires are now low.

A Linkcan be *paused* by transmitting one of the PAUSE tokens, followed by a NOP token that brings all five wires to a low state.

NOTE: The physical layer transmits tokens 0x1 and 0x2 using two escapes; they are not transmitted using the conventional single escape for control tokens less than 64. It is also the task of the physical layer to transmit a NOP after either a PAUSE or END token. Finally, on reception of a double escape END or PAUSE token, the physical layer must report this as a 0x1 or 0x2 control token, and the physical layer shall discard any NOP tokens that are received.

The encoding of the four hardware tokens operated by the physical layer is:

Name	**Value**	**Description**
RTNZ1	0xfc	NOP (return “0” to zero).
RTNZ2	0xfd	NOP (return “1” to zero).
RTNZ3	0xfe	NOP (return “2” to zero).
RTNZ4	0xff	NOP (return “3” to zero).

The PAUSE and END tokens are application level control tokens, and their encodings for higher levels are discussed in Section 3.6.1.

3.1.3 XS1-L Physical layer configuration

Bits are transmitted at a speed that is set under software control. Both speed and width are set by writing to the Link's speed register. Each of the speed registers specifies the width of the link, the gap between bits, and the gap between tokens. The addresses and contents of the speed registers are summarised in Section 9.3.

On a system-reset the link is set to a serial Link mode using two pairs. This enables boot over Link to work. The number of clock cycles spacing tokens should be reset to 400 and the number of clock cycles between symbols should be set to 400.

The speed of an Link is adjusted by changing the number of clock cycles between *tokens* and the number of clock cycles between *symbols*. Generally, these are both set to the same value. The token spacing field is encoded with an offset of 2, ie, 0x000 represents 2 cycles delay, 0x001 represents 3 cycles delay, up to 0x7ff representing 2049 cycles delay. The symbol spacing field is encoded with an offset of 1, ie, 0x000 represents a single cycle delay, 0x001 represents a two-cycle delay, etc and 0x7ff represents a 2048 cycle delay. The XS1-L cannot receive data if the transmitter does not space the symbols by at least two clock cycles. All clock cycles are relative to the *switch clock*, which clocks at 400 MHz by default, but can be set to run slower using register 7 (see Section 9.3).

For a 400 MHz system clock and bit spacing $s \geq 2$, the data rate achievable using 2 signal wires is $(160/s)$ Mbits/second; the data rate using 5 signal wires is $(400/s)$ Mbits/second. The actual speed that can be achieved depends on the electrical characteristics of the physical connection. Note that the XS1-L cannot receive bits faster than half the *switch clock* rate. When two XS1-Ls are running at the same clock, they should set their inter symbol delay to at least 2. If one of the XS1-Ls has a lower switch-clock-speed, the other one should adjust its inter symbol rate accordingly.

3.2 Link layer

The link layer protocol operates a point-to-point connection over a full-duplex Link. The link layer governs when data is transmitted, and how links start communicating. Four control tokens are used by the link layer: CREDIT8, CREDIT16, CREDIT64, and HELLO.

A link can be disabled or enabled. When disabled, no outside signals are coming through to the link state machine. When enabled, signals come through and are assembled into tokens.

When asked to transmit a HELLO, the Link resets its credits counter, and transmits a HELLO. On reception of a HELLO, the receiving Link resets its credits-issued counter, and issues at least eight credits. This sets the credits-issued counter to eight and transmits a CREDIT*X*. On reception of a CREDIT*X*, the receiving Link increments its credit-counter by *X*.

When asked to RESET, a Link resets the shift register capturing a token, clearing out any half tokens that may have been received.

3.2.1 Credits

The standard mode of operation is that a switch can *issue credits* on a link - when it does so, the switch allows the transmitter on the remote end of the link to transmit data to this switch.

The switch specifies how much credit is issued (8 to 64 bytes) using the reserved control tokens CREDIT8, CREDIT16 and CREDIT64. The transmitter will not transmit more tokens then there are credits. When multiple credit messages are issued, credits are summed together at the transmitting side; a transmitter must have a credit counter of at least 7 bits. Hence, it is illegal to send two subsequent CREDIT64 tokens, but legal to send a second CREDIT64 when one token has been received.

A transmitter should issue credits to the receiver, if it knows that the receiver is running low on credits, and if there is space in its input buffer. To save bandwidth, the transmitter should try and issue the largest possible credit token.

All data tokens require and consume credits. Most control tokens require and consume credits when transmitted, the exceptions are CREDIT*n*, HELLO, and RTNZ*n*; these tokens can be transmitted when there are no credits present because the link layer interprets them and does not insert them into the buffer. The application level tokens END and PAUSE consume credits as usual since they do end up in the buffer.

An enabled link is initialised by requesting it to send a *HELLO* token. This request can come from a local processor, or from a remote processor; possibly

even over the link itself. HELLO signals that this side is ready to receive credits. It requests that the other side clears its “issued-counter”, and issues credits.

NOTE: The HELLO token is not compatible with the XS1-G.

The definition of the three hardware control tokens used at the link layer is:

Name	Value	Description
CREDIT8	0xe0	Give additional 8 tokens of credit.
CREDIT64	0xe1	Give additional 64 tokens of credit.
CREDIT16	0xe4	Give additional 16 tokens of credit.
HELLO	0xe6	Solicit CREDIT

Note that the HELLO token is encoded in such a manner that an out-of-sync transmission over 2-wires will not result in a HELLO token, but in some other token. (A HELLO is transmitted as 1110011010.)

3.2.2 Initialising a Link comprising a single power and reset domain

On start-up, the credit counters are always zero. The boot-ROM enables all dedicated links (links E-H), and if boot-over-link is enabled it also enables link D.

If data needs to be transferred over a link, (say processor X wants to boot processor Y), processor X must ensure that it has credits. It does this by writing ’1’ to the bit that issues a HELLO, establishing an upstream (half-duplex) link. If bi-directional communication is required, processor X can initialise the other side (by using a control message over the established half-duplex link), or the booted code can initialise the other side. The register used to write a HELLO token is listed in Section 9.3

3.2.3 Initialising a Link comprising a hot plug

If hot-plugging is required, links shall be set to non-routed mode in software, a software layer shall first enable the link, then repeatedly issue HELLO until it establishes a link by reading the ”credits issued” bit. The software waits between issuing RESETs and HELLOs. The register used to write a HELLO token, and used to check for credits is listed in Section 9.3.

3.2.4 Initialising a Link comprising master and slave domains

In some designs there is a master-slave relationship between nodes; for example a “master” node that is always on controlling the power-supply of one or more “slave”-nodes. In this particular case, the master has knowledge that the slave is in a known state when booted. The master will hence wait for the slave to be booted, and then the master will enable the link and issue a HELLO.

3.2.5 Network numbers

A link can be assigned to be part of one of four “networks”. That is, the link will only carry traffic belonging to that network. For this to work, both channel ends must also be made part of this network. The intended use of this is to assign specific links to carry small control messages.

When setting up networks, no traffic should flow over the target network, as routing would be ambiguous. Network assignments are designed to be static, but if a link needs to be reassigned to, for example, the default network, the link should be disabled before the assignment is changed.

3.2.6 XS1-L Link Layer configuration

Before a link can be used it must be enabled and a HELLO must be issued. These actions are performed by writing a ’1’ to the appropriate bit in the speed registers (details are shown in Section 9.3).

On a system-reset the input FIFO is emptied, the output FIFO is emptied, and the credit and issued counters are set to zero. The link must then be enabled and a HELLO must be issued. In the case of hot-plugging, two bits of the speed-register can be read to establish whether credits have been issued or received.

3.3 Switch layer

The switch layer forwards messages from one link to another. This forwarding is either static (non-routed links) or dynamic (routed links).

3.3.1 Non-routed links

Links can be set to deliver data to a statically determined channel-end, instead of using the routing table. In non-routed mode no header is sent, and the message is sent to a specified processor and channel-end. This mode is enabled by setting bit 31 in the *Link static forwarding header register*, on both sides of the Link.

When an XCore wants to send data over a non-routed link it sets the channel-end destination register to address a core that differs in place *x* from the local core-id. Destination *x* is mapped in the lookup table to a direction that is associated with the required link. The destination channel and processor number have no relevance and are set to zero. No modifications are needed for this. The Link removes the header in non-routed mode. This saves three bytes being transmitted over the link.

On receiving data on a non-routed link, the link looks up which header to use in the Link static forwarding header register (registers 0xA0..0xA7). Each forwarding header register contains a channel identifier of up to 8 bits (in bits 7 ... 0), a core identifier of up to 8 bits (bits 15 ... 8), and an enable bit (bit 31). On the XS1-L no processor needs to be specified. The data is transmitted as usual over the internal link.

3.3.2 Routed links

Before data is transmitted on a stream, the switch sends a header to the destination core. The header establishes a route through the interconnect, and subsequent tokens follow the same route until the end-of-message (END) or pause (PAUSE) token are encountered. The header contains the identifier of the destination processor, which is encoded using either 16 bits or 3 bits. The processor address comprises a switch address and a core-number on that switch. The number of bits used to identify the core on the switch depends on the number of cores attached to the switch; on an XS1-L there is only one core and no bits are required, leaving all 16 bits to identify the switch. In the case of four cores on a switch, the lowest two bits of the address are used to identify the core, and the highest 14 bits are used to identify the switch.

The header mode can be set in software, by changing the lowest bit of configuration register 0x4 (Section 9.3). By default 3-byte mode is used; if the 1-byte

header is used it should be used on all nodes in the system.

Each node has a switch with a configurable identifier and routing table. The identifier is a bit pattern that (uniquely) identifies this node in the system. When a stream enters the switch, the destination node identifier is compared bit-by-bit with the switch-identifier. If all bits match the message is destined for this node and the message is routed to one of the local cores using the core-identifier.

If the switch-identifier is not equal to the stream's destination-node-identifier, the number of the first bit that differs specifies the dimension (direction) in which the message needs to be routed; this results in eight possible routing dimensions. The routing table associates each outgoing link with exactly one dimension, and the switch picks an available outgoing link for this dimension before forwarding the stream. This mechanism enables system designers to construct the routing tables for meshes, pipelines or hypercubes.

The node identifier of the XS1-L is initialised by writing its value in 15 ... 0 of the node identifier register. The most significant 16 bits are ignored.

Each link can be associated with one of four logical networks by writing the network number to bits 5 ... 4 of the link's configuration register. These network numbers correspond to the network numbers used when initialising channels using the SETN instruction.

A 16-entry look-up table associates a mismatch in each of the 16 node address bits with a logical direction. Each entry in this look-up table is large enough to hold an outgoing logical direction. Assuming that there are no more than 16 directions, this lookup-table logically comprises 64 bits; since there are only eight Links on an XS1-L only 48 bits need to be present.

The table is accessible via the configuration registers at addresses 0xC and 0xD in the system switch. The least significant four bits on address 0xC hold the direction for node address bit 0, the most significant four bits on address 0xD hold the direction for node address bit 15. Note that it is likely that multiple copies of this look-up table will be needed to eliminate routing latency arising from access contention.

Each Link can be associated with one of the directions by writing the direction to bits 10 ... 7 of the Link's configuration register. Four bits are sufficient for up to 16 directions. On the XS1-L only 3 bits are used.

Note: The node address is received most significant bit first, so direction 15 is

selected if the first bit received does not match bit 15 of the node address.

Two example topologies are shown below; a regular pipeline Figure 2 shows a regular pipeline and Figure 3 shows a mesh with missing wires (or pipe of pipelines). Other examples (such as hypercubes, trees, meshes, tori, and combinations of those) are easily constructed. Each node shows the node-id, the direction associated with each link, and the direction associated with each mismatching address bit.

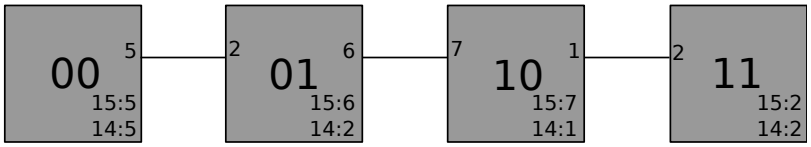

Figure 2 *Example: configuring a pipeline of four XS1-L1s*

3.3.3 XS1-L Switch Layer configuration

The core in the XS1-L is connected to the switch by four internal links, and the switch also allows connection to other chips via eight Links. The switch fully connects its 12 links (four internal links and eight external links) and can support 12 simultaneous message transfers.

The switch is configured by sending it *configuration messages*. These messages request the switch to write data to, or read data from, a bank of 32-bit configuration registers internal to the switch. These messages are used when booting to set the *node identifier* of the switch, associate *specific links with logical networks* and set the *speed and width of the Links*, and set the *routing strategy*. Section 9.3 summarises the registers (and the fields within the registers) that must be initialised in order to use the switch. The addresses used in the configuration messages are the register numbers of the 32-bit registers in the switch.

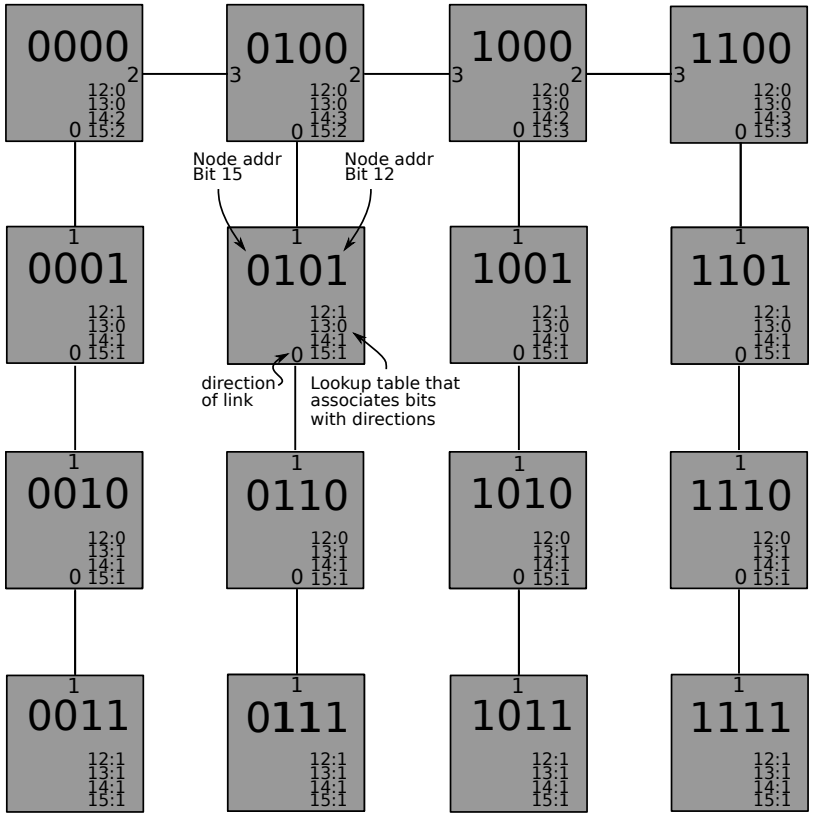

Figure 3 *Example: configuring a pipeline of four pipelines of four XS1-L1s*

3.4 Arbitration

The arbitration is fair, and gives fair bandwidth to all input links and load-balance all output links.

3.5 Processor communication

When processors communicate with each other, they transmit messages over the switches that, in addition to the 16- or 3-bit switch header include an 8- or 5-bit channel-end identifier. When a message is transmitted to the switch, it has a 24-bit (16 bits core-id + 8 bits channel-end) or 8-bit (3 bits core-id + 5 bits channel-end) header. When a message is transmitted to the processor over

an internal link, the message always has an 8-bit header which indicates the channel-end. If 1-byte switch headers is used, the first three bits of this byte will always be 0.

Processors can also communicate with switches. In this case the switch must be set to 3-byte header mode. When a message is transmitted to the switch, it contains a 2-byte header, and then a control token PSCTRL or SSCTRL. This indicates that the message is destined for the processor-control or switch-control associated with the processor addressed by the first two bytes. Messages that are transmitted to the PSCTRL or SSCTRL follow the following format:

- Two byte header identifying the destination processor/switch
- PSCTRL or SSCTRL token
- WRITEC control token
- Two bytes identifying core that reply should go to
- One byte identifying Channel-end for reply
- Two bytes identifying address within switch (*address*[15 ... 8], *address*[7 ... 0])
- Four bytes data to be written (*data*[31 ... 24], *data*[23 ... 16], *data*[15 ... 8], *data*[7 ... 0])
- END control token (value (0x01)

This results in the following reply message.

- Three bytes header (two bytes core identifier, one byte channel)
- ACK control token
- END control token

A read message is sent as follows:

- Two byte header identifying the destination processor/switch
- PSCTRL or SSCTRL token

- READC control token
- Two bytes identifying core that reply should go to
- One byte identifying Channel-end for reply
- Two bytes identifying address within switch (*address*[15 ... 8], *address*[7 ... 0])
- END control token

This results in the following reply message.

- Three bytes header (two bytes core identifier, one byte channel)
- ACK control token
- Four bytes data read (*data*[31 ... 24], *data*[23 ... 16], *data*[15 ... 8], *data*[7 ... 0])
- END control token

The four privileged tokens used to control the switch are defined as follows:

Name	**Value**	**Description**
WRITEC	0xc0	Write control register
READC	0xc1	Read control register
PSCTRL	0xc2	PSwitch configuration message
SSCTRL	0xc3	SSwitch configuration message

3.6 Channel Communication

At application level, the basic communication entity is a stream of data. A stream does not need to be limited in length, but it can be terminated after a short number of tokens has been transmitted, and can hence act as a ”packet” in a packet switched network. A stream is circuit switched and must be set up and terminated. If the destination channel end is local, data is exchanged directly. If the destination channel end is on a remote core, the switch first transmits a header to the other remote core. This header sets up a circuit for the stream. After the header is transmitted, the data-tokens and control-tokens of the stream are transmitted.

When the END token is transmitted, the switches free any resources, folding up the circuit that was used for streaming the data. The END token also returns all communication wires to a low-power state. A thread can temporarily suspend a stream by issue a PAUSE token at any time, which frees up the circuit and returns the communication wires to a low power state. Unlike the END token, the PAUSE token is invisible to the receiver, and is discarded once the final switch has freed its resources (analogous to the final switch discarding the header that was sent when the stream started).

Streams can be used to stream data such as audio or video just by opening the stream and sending volumes of data. The number of streams that can be opened simultaneously is limited by the number of physical links on the path. Each open uni-directional stream occupies one physical link in the direction of the stream. Given that an XS1-L has 4 links between the core and switch, no more than 4 streams can be open in one direction simultaneously. If switches are connected by less than 4 Links, then the number of simultaneous streams that can be used is limited to the number of Links connecting the switches.

Complex data types can be transmitted over a stream by opening a stream, and serialising the data, interspersed with user defined control tokens. This allows software to be constructed defensively by using control tokens to mark known synchronisation points in the data stream. If, at any time, the receiver were to try and input data when a control token is available or vice versa, the thread is trapped, and the program can flag or maybe recover from software errors.

By keeping streams short and synchronising often, streams can also be used to exchange packets of data. The cost of setting up a stream and terminating a stream is small, and unlike traditional packet-oriented networks, the packet is transmitted while it is being constructed; this overlaps packet creation and packet reception, reducing latency.

3.6.1 Application Tokens

Application tokens are defined by the compiler or application program. Four Application Control Tokens have been predefined, which should not be used for any other purpose:

Name	Value	Description
END	0x01	End - free up interconnect and tell target
PAUSE	0x02	Pause - free up interconnect but do not tell target
ACK	0x03	Acknowledge operation completed successfully
NACK	0x04	Acknowledge that there was an error

NOTE: These are the token values seen by the application software. When transmitted over a 5-wire Link the END and PAUSE tokens are represented by special token values. All other token values can be used by the application software in any way that it sees fit.

In addition to the application tokens, there is a block of tokens that are reserved for specific operations. These tokens have predefined meanings, and any implementation should use those meanings. Below 11 of those tokens are defined, all other 53 token values between 0x8b and 0xc0 are reserved for future use.

Name	Value	Description
READN	0x80	Read data.
READ1	0x81	Read one byte.
READ2	0x82	Read two bytes.
READ4	0x83	Read four bytes.
READ8	0x84	Read eight bytes.
WRITEN	0x85	Write data.
WRITE1	0x86	Write one byte.
WRITE2	0x87	Write two bytes.
WRITE4	0x88	Write four bytes.
WRITE8	0x89	Write eight bytes.
CALL	0x8a	Call code at the specified address.

3.6.2 XS1-L Configuration messsages over channels

To send configuration messages over a channel, the channel needs to be configured to transmit messages to the SSCTRL or PSCTRL logic. The destination of a configuration message is specified by a configuration resource identifier; this must be used to initialise a channel end using a SETD instruction in the usual way. The configuration resource identifier is a 32-bit word consisting of the following bytes:

byte	value
3,2	The Node Identifier of the switch to be configured
1	The numerical value for the PSCTRL or SSCTRL control token: 0xC2 or 0xC3
0	12

NOTE: The Node Identifier in a configuration message need not match the Node Identifier in the destination switch, allowing a configuration message to be used to initialise the switch Node Identifier.

Configuration messages follow the format specified in the previous section (remember that the header is generated by the channel-end):

- WRITEC control token (value 0xC0)
- Return channel end identifier (Node, Processor, Channel-end)
- Address within switch (*address*[15 ... 8], *address*[7 ... 0])
- Data to be written (*data*[31 ... 24], *data*[23 ... 16], *data*[15 ... 8], *data*[7 ... 0])
- END control token (value (0x01)

This results in the following reply message:

- ACK control token (value 0x03)
- END control token (value 0x01)

```
LDC    r11, 0     // preload0
LDC    r0, 0xC30C // Chan end for SSwitch ctrl on node 0
GETR   r5, 2      // Get a channel end
SETD   r5, r0     // set dest of chan end to SSwitch
OUTCTI r5, 0xC0   // WRITEC token
OUTT   r5, r11    // return address: node 0
OUTT   r5, r11    // return address: processor 0
SHR    r1, r0, 8
OUTT   r5, r1     // return address: chan-id stored in r0
```

```
OUTT   r5, r11    // high 16 bits of reg value: 0
OUTT   r5, r2     // low 16 bit of reg address, from r2
OUT    r5, r3     // value to be stored in r3
OUTCTI r5, 1      // in token
CHKCTI r5, 3      // check for ACK packet coming back
CHKCTI r5, 1      // termination of ACK packet
FREER  r5         // release channel end
```

To read data, the following sequence is sent:

- READC control token (value 0xC1)
- Return channel end identifier (Node, Processor, Channel-end)
- Address within switch (*address*[15 ... 8], *address*[7 ... 0])
- END control token (value (0x01)

This results in the following reply message:

- ACK control token (value 0x03)
- Data read (*data*[31 ... 24], *data*[23 ... 16], *data*[15 ... 8], *data*[7 ... 0])
- END control token (value 0x01)

All messages for SSCTRL must be sent on network 0, and all responses will be on network 0 too. SSCTRL messages must not be sent from other network numbers.

3.6.3 Configuring channel ends

Channel ends are configured using the SETD instruction. The SETD instruction takes a 32-bit resource-id. This resource-id must be either another channel end (type 2), or a configuration channel (type 12). The least significant 8 bits are the resouce type, the following 8 bits the number of the channel-end (or in the case of a configuration special valus 0xc2 or 0xc3 to indicate whether to control PSCTRL or SSCTRL), and the most significant 16 bits are the core and processor identifier. Channel end 0xff indicates a null channel

4 Selecting the oscillator frequency

The XS1-L needs an oscillator as a clock source. It can work using clocks in the range of 4.23 MHz to 100 MHz. Internally, a PLL is used to increase the clock frequency to 400 MHz; this is the core frequency used to run the processor data path and the switch. The 400 MHz is divided by 4 to derive the 100 MHz reference clock.

Two pins, MODE1 and MODE0, are used to select one of four standard oscillator frequencies (f_{osc}): 13, 20, 48, or 100 MHz:

F_{osc}	**MODE [1:0]**	F_{core}
13 MHz	00	399.75 MHz
20 MHz	11	400 MHz
48 MHz	10	400 MHz
100 MHz	01	400 MHz

If required, any other oscillator frequency in the range 4.22 - 100 MHz can be used as the clock source. In that case, the boot software must reprogram the PLL multiplier/dividers to get a core frequency of (close to) 400 MHz. Depending on the oscillator frequency one of the four modes is selected using MODE1 and MODE0, which results in the following power-up core frequencies:

F_{osc}	**MODE [1:0]**	F_{core}	**PLL ratio**	*OD*	*F*	*R*
4.23-13 MHz	00	130-399.75 MHz	30.75	1	122	0
13-20 MHz	11	260-400 MHz	20	2	119	0
20-48 MHz	10	166.67-400 MHz	$\approx$ 8.333	2	49	0
48-100 MHz	01	196-400 MHz	4	2	23	0

The above table also lists the values of *OD*, *F* and *R*, which are the registers that define the ratio of the core frequency to the oscillator frequency:

$$F_{core} = F_{osc} \times \frac{F+1}{2} \times \frac{1}{R+1} \times \frac{1}{OD+1}$$

OD, *F* and *R* must be chosen so that both $F_{core} \leq 400MHz$ and $260MHz \leq F_{osc} \times \frac{F+1}{2} \times \frac{1}{R+1} \leq 1.3GHz$. The *OD*, *F*, and *R* values can be modified by writing to register 6 in the interconnect. *R* must be in the range 0...63, *F* must be in the range 0...4095, and *OD* in the range 0...7.

5 Power control

The XS1-L has three modes:

- Active mode, where the core is active and all clocks run at operational frequency.
- Standby mode, where the core voltage remains present, but the chip is in a low power state.
- Sleep mode, where the core voltage is removed.

5.1 Active and standby mode

The XS1-L can be set to consume less dynamic power by reducing the clock frequency. When running at reduced clock frequency the XS1-L is in *standby mode*, when running at full clock frequency the XS1-L is in *active mode*.

The level of standby power is determined by the value of PLL_CLK_DIVIDER in the PSWITCH. A value of 0 means no power saving, a value of n means that the frequency is reduced by a factor $1/(n+1)$, reducing power by a factor $1/(n+1)$. Hence setting this register to 99 will cause the processor to run 100 times slower in standby mode and use 100 times less dynamic power.

The processor can switch automatically between active and standby modes, or it can switch when the program requests it to. This behaviour is controlled by bits 5 and 4 in PS_XCORE_CTRL0 (processor status register 2)

Bits [5:4]	mode
00	Active mode only - processor runs at full speed (400 MIPS)
01	Standby mode only - processor runs at one n-th of the speed (400/n MIPS)
11	Active mode if any thread is active, standby if there are no active threads

Timers and non-buffered ports work as usual when in standby mode, hence either can be used to wake-up a thread and switch the processor back to Active mode.

During standby mode, buffered ports using clocks faster than $400/n$ MHz cease to work properly. The table below gives some example values for the clock divider, and lists the resulting frequencies for all clocks.

Divider	Processor	Thread	Port synchr	Port transfer	Ref timer
none	400	100	400	100	100
1	200	50	200	50	100
3	100	25	100	25	100
99	4	1	4	1	100

NOTE: If the clock frequency is dynamically adapted, all clock frequencies switch between the top row and the chosen divider value, depending on the activity of any threads.

In dynamic mode, the clock frequency is set to maximum after 5-8 rising edges of the (slow) clock, and incurs a delay of $5 - 8\frac{PLL\_CLK\_DIVIDER+1}{400}\mu s$.

The switch can also be set to a lower clock speed, by sending a message to the switch requesting a change to SSWITCH_CLK_DIVIDER. Changing the clock speed of the switch affects the maximum speed at which data can be received and the speed at which data is transmitted (because the symbol-intervals and token-intervals that govern the transmission data-rate are measured in divided clock-ticks).

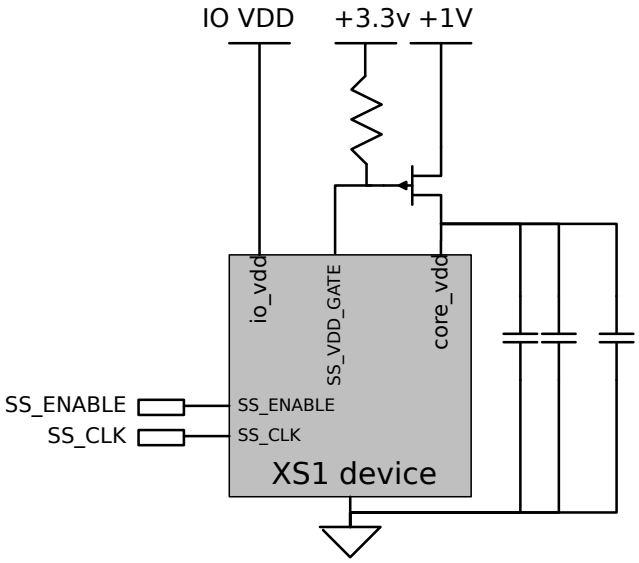

Figure 4 *PCU system diagram*

5.2 Sleep mode

The XCore can be made to completely switch off its core power, until an external signal is received or an internal timer expires. This requires an external FET that gates the core power supply (see Figure 4).

Users of the sleep mode should note the following:

- For minimal leakage, when the XCore is switched off, the internal pull-downs on IO pads cannot be relied on. The system should maintain all I/O pins at valid logic levels (pull-ups or pull-downs may be required).
- VDD at the chip IO must remain within specification, regardless of the voltage drop across the FET.
- SS_VDD_GATE must be pulled to a minimum of 3.3v.

The XCore can only be switched off co-operatively by writing a '1' into the sleep register (bit 0 of processor control register 9). It is woken up on one of two conditions: the wake-up pin was asserted for at least 0.1μs, or the wake-up

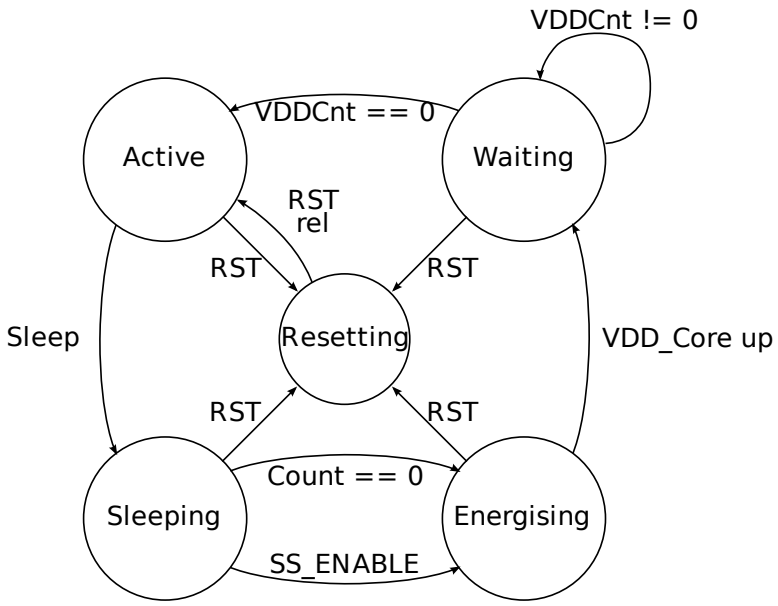

Figure 5 *PCU state diagram*

counter reached 0. The wake-up counter can be set by writing a 32-bit value into the wake-up counter register (processor control register 8). The wake-up counter counts at the external oscillator frequency. On wake-up the chip is brought up from power-on reset, which may take $x\mu s$.

Figure 5 shows the state machine of the XCore entering and leaving sleep mode.

Current state	Inputs					Outputs			
	SS_RESETB	SS_ENABLE	SLEEP	dec.zero	core VDD OK	Next state	SS_VDD_GATE	dec. enable	CORE_RESETB
Sleep (S)	active	X	X	X	X	A	0	active	active
	inactive	active	X	X	X	E			
	inactive	X	X	active	X	E			
Energise (E)	active	X	X	X	X	A	High-Z	inactive	active
	inactive	X	X	X	X	E			
Active (A)	active	X	X	X	X	A	High-Z	inactive	inactive
	inactive	X	active	X	X	S			

NOTE: The registers can only be written by this XCore. If other cores need to control the power-state of this processor, they need to activate a local thread.

It is recommended that user code tests the wakeup condition immediately before issuing the sleep signal.

A potential race condition exists between the internal assertion of sleep and the external assertion of SS_ENABLE. The race is between the test of the wakeup (or 'work available') condition by the instruction set ('ISA Test') and the test of the SS_ENABLE (external wakeup signal) from the FSM ('SS_ENABLE Test'). If the SS_ENABLE pulse is swallowed between these two points, the device will not wake up.

This is complicated by the fact that these two endpoints exist in different clock domains and the duration of the period between the ISA Test and the assertion of the sleep signal is determined by the number of instructions specified or implied by the user. In the case where the assertion of sleep and SS_ENABLE are co-incident, both signals must be synchronised into the FSM clock domain (2 cycles), the FSM state must then transition into the sleep state (1 cycle). The wakeup condition must still be valid in this state to ensure that it is not missed. It is recommended that the pulse width of SS_ENABLE is a minimum of two SS_CLK cycles plus the additional cycles to accomodate the core clock resynchronisation of the external wakeup condition signal and the user instructions between ISA test and the assertion of sleep.

6 JTAG

JTAG access to the XS1-L is exactly the same as JTAG to the XS1-G, except that the MUX has three unconnected entries. It is a two-stage process. A MUX can be used to:

- Run the scan chain through the core.
- Run the scan chain through the switch.
- Run the scan chain through neither (bypass).

The state of the MUX is programmed over JTAG. This enables an XS1-L2 to be constructed by chaining four XS1-L1s, whilst keeping the scan chain short. The MUX values are:

0000 NC The TMS signal is only connected to the MUX controller. The TDO output is taken directly from the MUX controller. In this mode the MUX controller can be interrogated without knowing the length of the DR in the devices.

0001 SSWITCH The TMS signal is connected to the SSwitch. The TDO output is connected to the SSwitch TDO.

1xxx CORE0 The TMS signal is connected to XCore0. The TDO output is connected to XCore0 TDO.

This encoding is backwards compatible with the XS1-G.

7 Free running oscillators

There are four free-running oscillators on the XS1-L. These free-running oscillators are designed to operate at different frequencies. The oscillators clock four counters. The counters and oscillators are controlled using a processor status register (using SETPS on register 6). The counter values can be read using four separate processor status registers (using GETPS on registers 7-10).

Oscillators can be enabled (started) by writing a '1' into the appropriate enable bit. The oscillator is disabled (stopped) by writing a '0'. Oscillators are disabled on a system-reset.

Counters are not cleared on system-reset, but keep their old state. Counters can be read by reading the associate counter registers using GETPS. Counters should only be read when the associated oscillator is stopped. The ring oscillator is normally used by reading the counter's initial value, starting the oscillator, stopping the osciallator, reading the counter's final value, and computing the difference as a signed short. Because the counters run asynchrounous, a reliable reading is obtained by repeating this process (eg 5 times) and taking the median value.

8 Secure boot module

The security module comprsises two parts: a configuration register (32 bits), and a one time programmable memory to store program code. The security register has the following layout:

Bits	Meaning
0	Disable JTAG debug access to this core
1	Disable access to Processor Control registers over the switch; all reads of any pswitch register return 0, writes are ignored
5	Force boot from address 0x000 of OTP, ignore the boot configuration register
7	Redundancy bit.
8	Disable programming of OTP sector 0
9	Disable programming of OTP sector 1
10	Disable programming of OTP sector 2
11	Disable programming of OTP sector 3
12	Disable OTP programming completely: disables updates to all sectors and the security register
13	Disable all (read & write) access from the JTAG interface to this OTP
14	Disable any interaction with GlobalDebug for this XCore
21..15	General purpose software accessable security register available to end-users
31..22	Core 0 - general purpose user programmable JTAG UserID code extension; Cores 1, 2, and 3 - general purpose available to end-users

The memory itself comprises four sectors containing 2KBytes each. The banks are organised contiguously and can be used as a single 8KByte bank, but each sector can be locked independently if required.

The OTP memory is programmed using three special I/O ports: the OTP address port is a 16-bit port with resource ID 0x100200, the OTP data is written via a 32-bit port with resource ID 0x200100, and the OTP control is on a 16-bit port with ID 0x100300.

9 General configuration registers

The XS1-L has three types of control registers:

- registers in the processor itself - control information private to the processor. The registers are accessed using GETPS and SETPS instructions.
- registers in the processor switch - control information specific to a processor that can also be accessed by other processors.
- registers in the interconnect switch - control information related to the interconnect and the logic shared between processors

Some information is available via multiple paths, to facilitate remote access and quick access via the instruction set, and for remote debugging purposes.

9.1 Processor status registers

The following are processor status registers that are accessed using GETPS and SETPS. The regsiter number must be translated to a resource ID by shifting the register number left 8 bits, and oring 0x0B in (the resource ID that identifies a processor control register).

Address			Contents
0x00	PS_RAM_BASE	RW	Address of RAM. Keep at 0x00010000.
0x01	PS_VECTOR_BASE	RW	Base of all 0 resource vectors. Used for both events and interrupts. Bits 31-16 should be set, bits 15-0 should be kept 0.
0x02	PS_XCORE_CTRL0	RW	General control bit 0: (verif) Reference clock from core clock bit 4: enable divider bit 5: enable divider only on WAIT
0x03		RO	Value of the boot mode pins
0x04			Reserved
0x05			Value of the OTP security register bits 0..31, see Section 8
0x06			WO Oscillator control register bit 0: enable oscillator PeCl, PeWi bit 1: enable oscillator CoCl, CoWi
0x07			bits 15..0: RO Value of counter CoCl
0x08			bits 15..0: RO Value of counter CoWi
0x09			bits 15..0: RO Value of counter PeCl
0x0A			bits 15..0: RO Value of counter PeWi
0x0B			Power control wake-up counter.
0x0C			Power control wake-up. Bit 0 indicates “go to sleep”

Address			Contents
0x10	PS_DBG_SSR	DRW	Saved SR for debug interrupts, *dssr*
0x11	PS_DBG_SPC	DRW	Saved PC for debug interrupts, *dspc*
0x12	PS_DBG_SSP	DRW	Stores the stack pointer during debug interrupts, *dssp*
0x13	PS_DBG_T_NUM	DRW	The resource ID of the thread whose state is to be read.
0x14	PS_DBG_T_REG	DRW	Register number to be accessed by DGETREG.
0x15	PS_DBG_TYPE	DRW	The type of debug interrupt.
0x16	PS_DBG_DATA	DRW	The data causing the debug interrupt.
0x18	PS_DBG_RUN_CTRL	DRW	Determine which threads are active in when not in debug mode.
0x20-0x27	PS_DBG_SCRATCH	DRW	Scratch register for debug interrupts. 0-7
0x30-0x33	PS_DBG_IBREAK_ADDR	DRW	Instruction breakpoint address 0-3
0x40-0x43	PS_DBG_IBREAK_CTRL_0	DRW	Instruction breakpoint control register 0-3.
0x50-0x53	PS_DBG_DWATCH_ADDR1	DRW	Data watchpoint address 1. 0-3
0x60-0x63	PS_DBG_DWATCH_ADDR2	DRW	Data watchpoint address 2. 0-3
0x70-0x73	PS_DBG_DWATCH_CTRL	DRW	Data breakpoint control register. 0-3
0x80-0x83	PS_DBG_RWATCH_ADDR1	DRW	Resources breakpoint address 1. 0-3
0x90-0x93	PS_DBG_RWATCH_ADDR2	DRW	Resources breakpoint address 2. 0-3
0xA0-0xA3	PS_DBG_RWATCH_CTRL	DRW	Resources breakpoint control register. 0-3

9.2 Processor switch registers (per core)

The following registers are in the processor switch. They can be accessed over JTAG or by sending a message to the processor switch. The message to be sent is specified in Section 3.6.2. These registers are specific to a processor.

Address	Contents
0x00	Device ID register bits 7..0: XCore version, 0x0 bits 15..8: XCore revision, 0x3 bits 31..16: Node/Core number, taken from SSWitch
0x01	Number of resources - I bits 7..0: Number of Threads bits 15..8: Number of Synchronisers bits 23..16: Number of Locks bits 31..24: Number of Channel Ends
0x02	Number of resources - II bits 7..0: Number of Timers bits 15..8: Number of Clock Blocks
0x04	Control PSwitch permissions to debug registsers. bit 0: Disable write access to processor registers. bit 8: Disable remote access, can only be cleared locally. bit 31: Disable further updates to any PSwitch register
0x05	Debug interrupts bit 0: Writing a 1 generates a debug interrupt.
0x06	Processor clock divider (only lower 16 bits are used)
0x07	Value of the OTP security register bits 0..31, see Section 8
0x10-0x13	Internal link status Internal LinkPA, PB, PC PD, see Section 9.4. Verification only.
0x20-0x27	Scratch register for debug software protocols 0-7 Register 0x20 can be used to set the address of a user debug handler
0x40-0x47	Copy of the PC of threads 0-7
0x60-0x67	Copy of the SR of threads 0-7
0x80-0x9F	Link status of LINK 0-31, see Section 9.4. Verification only.

9.3 Interconnect registers (per node)

The following registers are in the interconnect - they can be accessed over JTAG or by sending a message to the system switch. The message is specified in Section 3.6.2. Changing these registers has a global effect on the chip.

Address	Contents
0x00	Device identification bits 7..0: SSwitch version, 0x1 bits 15..8: SSwitch revision, 0x0 bits 23..16: The value of the SS_MODE pins, sampled on reset
0x01	Number of resources bits 7..0: Number of internal links per core bits 15..8: Number of Cores bits 23..16: Number of Links
0x04	Node configuration bit 0: Short headers (use 1-byte headers if set) bit 8: Disable PLL modifications. bit 31: Disable further updates to any SSwitch control register.
0x05	Node ID, lower 16 bits only
0x06	PLL control register. bit 5..0: PLL reference divider: *R* bit 20..8: PLL feedback divider: *F* bit 25..23: PLL output divider: *OD*
0x07	Lowest 16 bits set the switch clock frequency, $clk = pll/(n+1)$. Keep at 0 for 400 MHz.
0x08	Lowest 16 bits define the relation between the reference clock and the PLL clock, $ref = pll/(n+1)$. Keep at 3 for 100 MHz.
0x0C	Directions for bits 0 to 7 bits 3..0: Direction for bit 0 bits 7..4: Direction for bit 1 ... bits 31..28: Direction for bit 7
0x0D	bits 3..0: Direction for bit 8 ... bits 31..28: Direction for bit 15
0x10	GlobalDebug configuration for XCore0 bit 0: Drive the global debug pin on global debug events bit 1: Allow the global debug pin to generate global debug events
0x1F	global debug source. bit 0: Set if XCore0 is the source of the last global debug event bit 4: Set if the global debug pin is the source of the last global debug event

Address	Contents
0x20	Link C direction and network, see Section 9.4
0x21	Link D direction and network, see Section 9.4
0x22	Link A direction and network, see Section 9.4
0x23	Link B direction and network, see Section 9.4
0x24	Link G (X1LC) direction and network, see Section 9.4
0x25	Link H (X1LD) direction and network, see Section 9.4
0x26	Link E (X1LA) direction and network, see Section 9.4
0x27	Link F (X1LB) direction and network, see Section 9.4
0x40	PLink 0 network, see Section 9.4
0x41	PLink 1 network, see Section 9.4
0x42	PLink 2 network, see Section 9.4
0x43	PLink 3 network, see Section 9.4
0x80	Link C speed, timing, and width
0x81	Link D speed, timing, and width
0x82	Link A speed, timing, and width
0x83	Link B speed, timing, and width
0x84	Link G (X1LC) speed, timing, and width
0x85	Link H (X1LD) speed, timing, and width
0x86	Link E (X1LA) speed, timing, and width
0x87	Link F (X1LB) speed, timing, and width bit 10 ... 0: minimum number of system clock cycles between tokens bit 21 ... 11: minimum number of system clock cycles between symbols bit 23: RESET input state machine bit 24: Issue a HELLO and clear credits-counter bit 25: RO Link has credits and can transmit bit 26: RO Link has issued credits and can receive bit 27: RO (cleared by reading), Link protocol error. bit 30: Number of signal wires - 0: two pairs; 1: five pairs bit 31: Enable link
0xA0	Link C static forwarding header
0xA1	Link D static forwarding header
0xA2	Link A static forwarding header
0xA3	Link B static forwarding header
0xA4	Link G (X1LC) static forwarding header
0xA5	Link H (X1LD) static forwarding header
0xA6	Link E (X1LA) static forwarding header
0xA7	Link F (X1LB) static forwarding header bit 7 ... 0: Channel end bit 15 ... 8: Core identifier (0 on XS1-L) bit 31: enable static forwarding

9.4 Link status/control bit formats

The internal and external link registers have the following structure - where the Direction bits are 0 for internal links.

bits	use
0	SRC_INUSE
1	DST_INUSE
2	JUNK
5..4	Network - a network ID can be written in those bits
11..8	the direction to which this link belongs
24..17	SRC_TARGET_ID
26..25	SRC_TARGET_TYPE