XS1 Ports: use and specification

(VERSION 1.02)

2008/11/25

Authors:

PETER HEDINGER
ALI DIXON
ROSS OWEN
NEIL RICHARDS
DAVID MAY
HENK MULLER

1 Introduction

Ports define the interface between hardware attached to an XCore and software running on an XCore. The port logic can take care of timing, serialisation/de-serialisation, strobing and pattern matching. This document gives a number of examples of how to use ports (Sections 2) followed by a specification of how to use ports (Section 3). The examples are presented in the form of XC programs and assembly programs, in combination with waveform diagrams. Reference manuals for the libraries [1] and language [2] are documented separately.

2 Ports by Example

Below are a number of sample programs explaining how to use ports and many port features. For a specification on how to use ports, see Section 3. The simplest port toggles pins up or down, or samples the value of input pins. More complex ports use specific times, handshaking, or conditions.

2.1 Simple ports: high and low wires

A simple program that toggles a pin high and low once is shown below:

```
#include <xs1.h>

out buffered port:1 x = XS1_PORT_1P;
clock              ref = XS1_CLKBLK_REF;

int main(void) {
  configure_out_port_no_ready(x, ref, 0);
  x <: 1;          // Set pin high
  x <: 0;          // Set pin low
}
```

A port is a logical end-point (in assembly and in XC) of one or more input or output pins. A port is defined using a port declaration, and is then used by using the input (`:>`) and output (`<:`) statements. The port declaration uses an architecture port identifier, eg `XS1_PORT_1P`. This is translated into a resource identifier in assembly, eg a word 0x00010F00.

The port identifier `XS1_PORT_1P` refers to the `XS1`-architecture; a port that comprises `1` pins (the *port width*); and the `P` identifies the particular 1-bit wide port of the processor. In assembly code this is translated into 0x00010F00, where the 01 is the port width, and 0F identifies the particular 1-bit wide port. Physically, port 1P on the XS1 is connected to pin X0D39 where the 0 identifies the core that the pin is connected to (core 0) and the digits are a logical pin number between 0 and 63 inclusive. The physical position of pin X0D39 on depends on the packaging. A list of all ports and the mapping between pins and ports is listed in Table 1 near the end of this document. The mapping of pins onto locations on the package is provided in the data-sheet for your package.

This program configures its port by calling the function `configure_out_port_no_ready(x, ref, 0)`. This particular function initialises the port to be an *output* port with initial value 0. The port is dedicated to have "no ready signals" (ready signals are discussed in Section 2.4), and states that the port shall be sampled using the reference clock (the variable `ref` set to `XS1_CLKBLK_REF`, this is discussed in more detail in Section 2.2).

The program in Figure 1 shows a slightly more complex example of ports, using both a 1-pin output port, and a 4-pin input port. The 4-pin input port is called inP, and the 1-pin output port is called outP. The 4-pin port is port 4A which is connected to pins X0D02, X0D03, X0D08, and X0D09; the 1-pin port is port 1B which is connected to pin X0D01 (see Table 1 on page 30, all our programs run on core 0 and hence connect to X0D*N*). The program continuously reads the input port (it samples the 4 input pins), and outputs a "1" on the output port if the input value exceeds 9. Note that unless configured otherwise, a value of "0" is represented by a low voltage (0V) and a value of "1" is represented by a high voltage (3.3V).

Example input and output to this program are shown at the bottom of Figure 1. The input stimulus on port 4A (pins 2, 3, 8, and 9) is binary values 1000, 1010, 0010, and 0011; the program drives output values 0, 1, and 0 in response on pin 1.

2.2 Timing I/O: A square wave on a 1-bit port

Many I/O operations need to be performed at some specific time. This may be a time in relation to an external event, or a time in relation to the code being executed. Depending on which is of interest, either *timers* or *port counters* can

```
#include <xs1.h>

out buffered port:1 outP = XS1_PORT_1B;
in  buffered port:4 inP  = XS1_PORT_4A;
clock               ref  = XS1_CLKBLK_REF;

int main(void) {
  int value;
  configure_out_port_no_ready(outP, ref, 0);
  configure_in_port_no_ready(inP, ref);
  while (1) {
    inP :> value;
    if (value > 9)
      outP <: 1;
    else
      outP <: 0;
  }
}
```

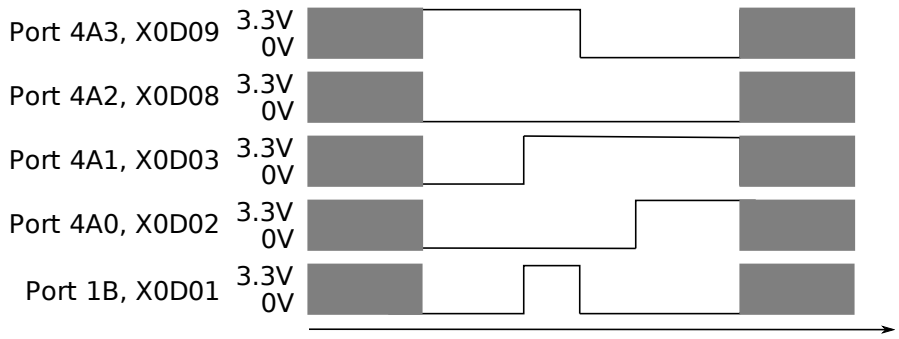

Figure 1 *Basic use of input and output ports.*

be used. Timers are 32-bit counters that are relative to the internal 100 MHz reference clock. Port counters are 16-bit counters clocked by either an external clock, or by a divided internal reference clock. Port counters are guaranteed to perform I/O operations at precisely defined moments related to an externally visible clock signal. Timers are discussed in the XC documentation[3, 2]; in this document we focus on the port counters.

A simple example that pulls a pin high on the third clock, and that lowers it on the fifth is shown in Figure 2. It uses two 1-bit ports, an output port on port 1B (pin X0D01 on an XS1), and an input clock from some external source on port 1C (pin X0D02). The output port is declared as before, but the port used for the clock signal has to be declared unbuffered; it is neither possible (nor desirable) to buffer clock signals. The example also uses a *clock block*, which is the unit that makes the pin into a clock. There are 5 clock blocks available, XS1_CLKBLK_1 .. XS1_CLKBLK_5, and you can connect each clock block to at most one input clock. Alternatively, you can run a clock block from the divided reference clock.

Figure 2 also shows the timing diagram of the input stimulus and the resulting output signal. The port counter is incremented on falling edges of the clock. Note that the port counter simply counts clock edges on the input clock. A port counter counts in the application clock domain, and is only a *measure of time* if the input clock is a signal that has clock edges at regular intervals. If the input clock is irregular (such as the one shown in Figure 2), then the port counter is not a measure of time. There is a limit to the speed at which an external clock can run, which can be found in the datasheet.

If no external clock is available, a port can be clocked by the internal reference clock, and can be divided. The functions to configure the port this way are `set_clock_ref(clk)` and `set_clock_div(clk, divide)`. This will run the clock at 100 MHz divided by two times the given 8-bit value `divide`. The clock block can then be used to drive a clock on an output pin, and the clock block can be used to clock input and output ports.

An example use of a generated clock is the code to drive an LCD screen. Without going in much detail as to how an LCD screen is driven, an LCD screen requires a clock signal (DCLK), a horizontal sync (HSYNC) a data ready (DTMG) and red/green/blue data (RGB). The code shown in Figure 3 has a clock block that drives DCLK, and is used to output values at the right time on the HSYNC, DTMG and RGB ports. The clock runs at 2.5 MHz. The output is shown in the same Figure. The port driving the clock is initialised by calling `configure_port_clock_output(DCLK_port,clk)`, which dedicates the given

```
out buffered port:1 toggle = XS1_PORT_1B;
in           port  inClock = XS1_PORT_1C;
clock                  clk = XS1_CLKBLK_1;

int main(void) {
  int count;

  configure_clock_src(clk, inClock);
  configure_out_port_no_ready(toggle, clk, 0);
  start_clock(clk);

  toggle <: 0 @ count;     // read port counter
  while (1) {
    count += 3;
    toggle @ count <: 1;  // timed output
    count += 2;
    toggle @ count <: 0;  // timed output
  }
}
```

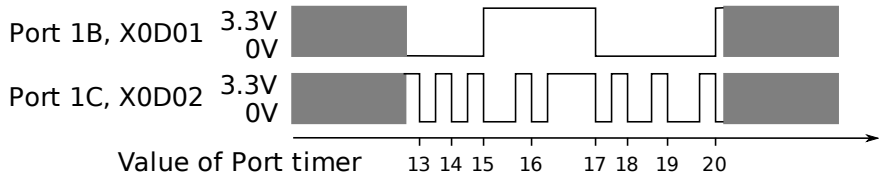

Figure 2 *Use of a port counter to count input clocks.*

port to output the given clock signal.

2.3 Buffering, Serialising and Deserialising data

Ports have the capability to serialise and deserialise data. In order to declare a port that serialises or deseralises the data, the port is specified to be wider than the physical number of pins. The number after the port declaration in that case specifies the *transfer-width* of the port.

An example program is shown in Figure 4. The first line declares `oneBit` to be attached to a single physical pin (X0D12, port 1E) which is driven from a 32-bit shift register (the `:32` after the **`port`** keyword). The least significant bits of the shift register drive the pin, and on each rising edge of the clock, the shift register is shifted one position, until it has been shifted 32 places, after which the next 32-bit word could be written into the shift register. Using a shift register reduces the number of instructions required to output data, for one output operation prepares 32 output transitions.

An example where this is useful is situations where you drive systems that have a high data rate, such as USB or Ethernet over a port that is only a few bits wide. With a buffered port the program only has to interact with the port once every 32 bits.

Note that the port *declaration* specifies the number of bits that are transferred in each input and output operation; the *transfer-width*. The port *initialisation* specifies the number of physical pins attached to the port, the *port width*.

Bits are shifted in and out starting with the *least significant bit*. When you need to start with the most significant bit you can use a single cycle bit-reverse operation (BITREV) that can be called by invoking the `bitrev()` function. Figure 4 shows the timing output related to the example program above. Note that in reality the signal will be not as sharp as shown in Figure 4, and because of capacitance on the output pin the signal will be slowly approaching one and zero. This can be used as a feature if an analogue signal is to be generated: a 1-bit DA converter can be built using a serialising 1-bit port and a resistor/capacitor low-pass filter on the output pin.

Below is an example of serial to parallel conversion on an input port, driven by a 25 MHz clock. The input port declaration declares a port referring to four physical pins (X0D04 - X0D07), with an 8-bit input shift register. This allows

```
out buffered port:4  HSYNC_port = XS1_PORT_4F;
out buffered port:1  DTMG_port  = XS1_PORT_1B;
out          port    DCLK_port  = XS1_PORT_1A;
out buffered port:32 RGB_port   = XS1_PORT_32A;
clock clk = XS1_CLKBLK_1;

void lcd_init() {
  unsigned rows, lines, x, time = 0;
  set_clock_div(clk, 20);
  configure_out_port_no_ready(HSYNC_port, clk, 0);
  configure_out_port_no_ready(DTMG_port, clk, 0);
  configure_out_port_no_ready(RGB_port, clk, 0);
  configure_port_clock_output(DCLK_port, clk);
  start_clock(clk);
  x = nextRGBSample();
  while(1) {
    time += 500;
    for(int lines = 0; lines < 320; lines++) {
      time += 8;  HSYNC_port @ time <: 1;
      time += 31; DTMG_port @ time <: 1;
      RGB_port @ time <: x;
      x = nextRGBSample();
      for(int rows = 1; rows < 240; rows++) {
        RGB_port <: x;
        x = nextRGBSample();
      }
      time += 240; DTMG_port @ time <: 0;
      time += 13;  HSYNC_port @ time <:0;
    }
  }
}
```

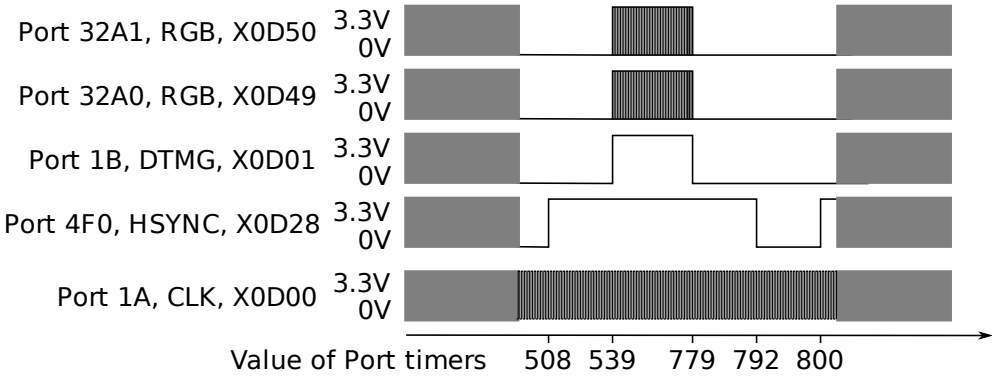

Figure 3 *Program for an LCD driver and signals generated. Only the lowest two bits of the 32-bit port are shown; the signals are data dependent.*

```
out buffered port:32 oneBit = XS1_PORT_1E;
clock                ref = XS1_CLKBLK_REF;

int main(void) {
  configure_out_port_no_ready(oneBit, ref, 0);
  oneBit <: 0xFFFF00AA;
}
```

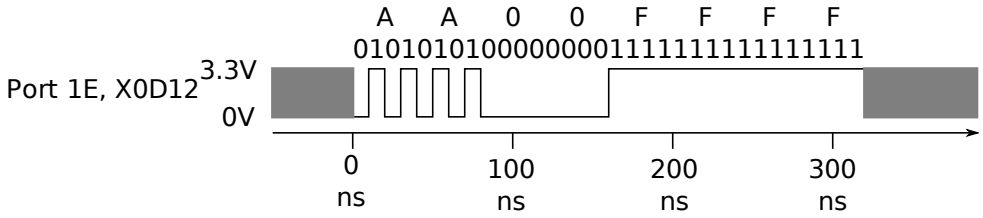

Figure 4 *Program showing serialising port, and the signal generated.*

two 4-bit values to be stored before they must be processed by software. As with the output, using a deserialising port reduces the number of instructions required to perform an input. The clock is declared in two parts (as discussed in Section 2.2): an output port that will be used to make the clock signal visible on a pin, and a clock block declaration. An example program and signals are shown in Figure 5; it will read the value 0x28 into the variable x.

The data is sampled on the rising edges of the clock. The clock does not need to be output on a port, but it has been output on port 1A (X0D00) in order to clarify the timing diagram. The lowest bit of port 4B is connected to pin X0D04, the highest to pin X0D07. When shifting the LSB is read first, resulting in 0x28 being read. Note that this code relies on when the clock starts - the next section on strobing discusses how to read (and deserialise signals) at specific times.

The sampling can be performed on the rising edge of the clock if desired, by setting the *sdelay* mode. The clock can also be inverted, meaning that the output will happen on the rising edge, and input on the falling edge.

2.4 Using strobe signals

In many cases input and output signals are accompanied by *strobing* signals. The ports on an XS1 can input and interpret strobe signals generated by out-

```
in  buffered port:8 inP   = XS1_PORT_4B;
out           port  clockOut = XS1_PORT_1A;
clock               clk25    = XS1_CLKBLK_1;

int main(void) {
  set_clock_ref(clk25);
  set_clock_div(clk25,2);
  configure_in_port_no_ready(inP, clk25);
  configure_port_clock_output(clockOut,clk25);
  start_clock(clk25);

  inP :> int x;
}
```

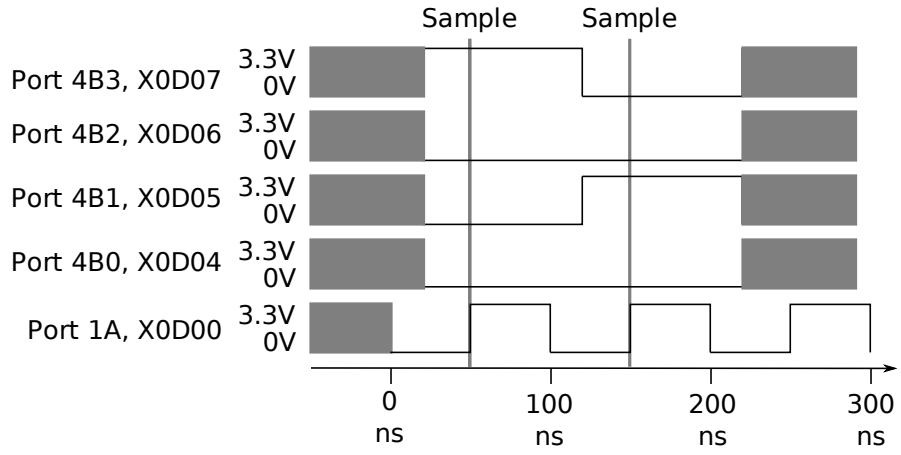

Figure 5 *Example showing the use of a 4-bit deserialising port.*

side sources, and ports can generate strobe signals to accompany outputted data. These signals are known as *ready-in* and *ready-out*, and the port can be configured in one of four modes:

- No ready signals: this is the normal mode that was used in all preceding sections.
- Strobed master, in which case ready signals are only output. The configuration function requires a one bit output port that is to be used for ready-output.
- Strobed slave, in which case ready signals are only input. The configuration function requires a one bit input port that is to be used for ready-input.
- Bidirectional strobe, in which case ready out signals are both input and output. The configuration function requires a one bit input port and a one bit output port that are to be used for ready-input and ready-output.

As is the case with clocks, the ports that are used for ready input and output cannot be buffered. When a ready input signal is used on a port, the port only operates when the ready-input signal is high. When a ready output signal is used, the port will raise the ready-output when it operates. If active low signals are required, you can simply invert the port used for the ready signal(s).

Below we give two example sections of code, one that uses ready-output signals, and one that uses ready-input signals. The LCD driver discussed earlier requires a pin (DTMG) to be pulled high when data is output. After declaring the RGB port to be a strobed master that uses DTMG as the ready output signal we can simply remove the outputs to DTMG. This is shown in Figure 6.

An example section of slave code is shown in Figure 7. This code uses a ready input signal, and samples data on a 4-bit port when the ready-input strobe is high. When running this code on the stimulus shown at the bottom of Figure 7, the input statement will input the value 0x82.

2.5 Conditional input

It is often useful to wait for some condition on an input line. For example, a program may wait for a wire to go high or for a specific bit pattern to be sampled on the line. The XS1 ports allow a program to wait for one of two conditions:

```
out buffered port:4  HSYNC_port = XS1_PORT_4F;
out          port:1  DTMG_port  = XS1_PORT_1B;
out          port    DCLK_port  = XS1_PORT_1A;
out buffered port:32 RGB_port   = XS1_PORT_32A;
clock                clk         = XS1_CLKBLK_1;

void lcd_init(chanend toLCD) {
  unsigned rows, lines, x, time = 0;
  set_clock_ref(clk);
  set_clock_div(clk, 20);
  configure_port_clock_output(DCLK_port, clk);
  configure_out_port_no_ready(HSYNC_port, clk, 0);
  configure_out_port_strobed_master(RGB_port, DTMG_port, clk, 0);
  start_clock(clk);

  x = inuint(toLCD);
  while(1) {
    time += 500;
    for(int lines = 0; lines < 320; lines++) {
      time += 8;
      HSYNC_port @ time <: 1;
      time += 31;
      RGB_port @ time <: x;
      x = inuint(toLCD);
      for(int rows = 1; rows < 240; rows++) {
        RGB_port <: x;
        x = inuint(toLCD);
      }
      time += 240 + 13;
      HSYNC_port @ time <:0;
    }
  }
}
```

Figure 6 *Driving an LCD display using ready output; the output is the same as shown in Figure 3.*

```
in  buffered port:8 inP     = XS1_PORT_4B;
in           port   clockIn = XS1_PORT_1A;
in           port   ready   = XS1_PORT_1B;
clock               clk     = XS1_CLKBLK_2;

int main(void) {
  set_clock_src(clk, clockIn);
  configure_in_port_strobed_slave(inP, ready, clk);
  start_clock(clk);

  inP :> int x;
}
```

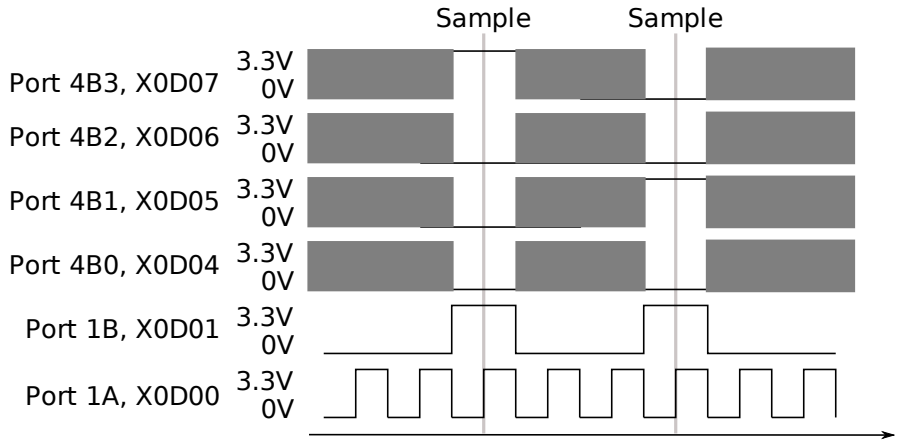

Figure 7 *Program and wave-forms showing use of ready-input signal.*

```
in  buffered port:1 oneBit  = XS1_PORT_1B;
out buffered port:4 counter = XS1_PORT_4A;
clock               ref     = XS1_CLKBLK_REF;

int main(void) {
  int oldValue = 0, i = 0;
  configure_in_port_no_ready(oneBit, ref);
  configure_out_port_no_ready(oneBit, ref, 0);
  oneBit :> oldValue;
  while (1) {
    oneBit when pinsneq(oldValue) :> oldValue;
    counter <: ++i;
  }
}
```

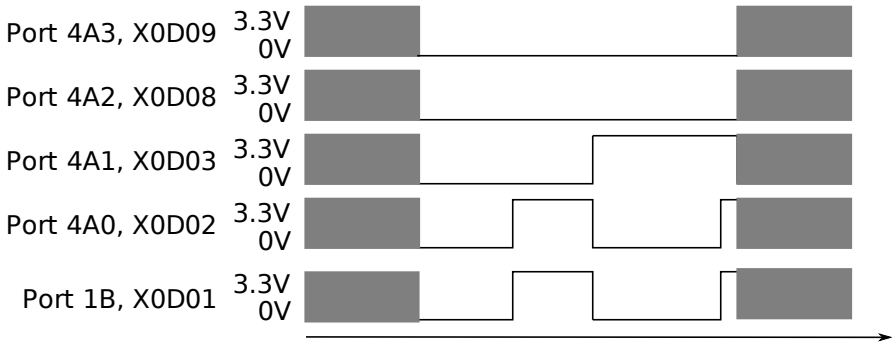

Figure 8 *Program and wave forms showing conditional input.*

equal or not equal to some value. As an example, the code in Figure 8 counts the number of transitions on its input wire. Example input stimuli and expected output are shown below the code.

Patterns can be more complex on multi-bit ports. For example, in order to wait for a preamble on an Ethernet cable the only operation required is:

```
in  buffered port:32 ethData  = XS1_PORT_4A;

int main(void) {
  ethData when pinseq(0xD) :> int _;
  //... packet has started...
}
```

2.6 Summary of port configuration

- Ports must be configured in order to specify how many (if any) ready wires are used, or whether this port should be driving a clock. This is done by calling a `configure_*_port_*` function. After this function completes, the port is ready to be used for input and output. The functions used to set the mode are
  `configure_*_port_no_ready`,
  `configure_*_port_handshake`,
  `configure_*_port_strobed_master`,
  `configure_*_port_strobed_slave`, and
  `configure_port_clock_output`.
- Ports may have some modes set, such as inverting, pull-mode, and sdelay. These modes are set using the appropriate `set_port_*` function, and must be called before configuring the port. `set_port_inv()`, `set_port_sample_delay()`, and `set_port_pull_up()` can be used to set the appropriate mode.
- Each port is clocked, either by the default reference clock, or by a specific clock. The port source is specified in the call to configure.
- You can configure clock blocks to either generate a clock based on a divided reference clock, or use an input pin as a clock. When you are done configuring a clock you must call `start_clock()`. Call this function *after* all ports are configured, and you are guaranteed that all ports will have the same port counter values.

3 Port specification

Ports define the interface between hardware attached to an XCore and software running on an XCore. The port logic can take care of timing, serialisation/deserialisation, strobing and pattern matching.

Ports can be be configured to be in one of 10 modes. Most of those modes are strobed, where data is input and output depending on the presence of strobing signals. Two special modes implement input sampling and bidirectional I/O. The ports are listed below, and are discussed in detail in subsequent sections.

Direction	Strobing	Principal operations
Input	Bidirectional	Input, conditional input, both optionally timed
Input	Master	Input, conditional input, both optionally timed
Input	Slave	Input, conditional input, both optionally timed
Input	Implicit	Input, conditional input, both optionally timed
Output	Bidirectional	Output, optionally timed
Output	Master	Output, optionally timed
Output	Slave	Output, optionally timed
Output	Implicit	Output, optionally timed
Input	-	Timed sampling input
Bidirectional	-	Sampling Input, output

Buffered ports, discussed in Section 3.1, should be used unless bidirectional ports (Section 3.3) or a precise timed sample (Section 3.2) are required.

All input and output operations always record the value of the port counter. This value records when the last I/O operation happened, as measured according to the port's clock. All ports have a clock associated with them; the clock can be set to come from an internal source (optionally divided) or an external source. Clocks are discussed in Section 3.4

3.1 Buffered modes

Buffered modes allow:

- Optional use of a FIFO to serialise and deserialise data.
- Use of up to two pins for strobing.
- Perform conditional input on data.

3.1.1 FIFO

A double-buffering FIFO can be enabled on for 1, 4, 8, 16, and 32-bit ports. The FIFO serialises (output) or deserialises (input) data. The double buffering allows the entire contents of the FIFO to be copied for transfer to (input) or from (output) the program. This enables data to be serialised using a high-frequency clock decoupled from the program.

The FIFO holds at most *bpw* bits; this enables a single word-wide operation to transfer the entire contents of the FIFO to or from a register. Hence, for a *w*-bit wide port, the size of the FIFO is limited to $\frac{bpw}{w}$ elements. The topmost *w* bits of the FIFO are the value that is most recently clocked in. The lengths of the FIFO that are supported are:

- 1-bit ports: 4, 8, 32.
- 4-bit ports: 8 (2 elements), 32 (8 elements).
- 8-bit ports: 32 (4 elements).
- 16-bit ports: 32 (2 elements).
- 32-bit port: double buffering only.

On input, when the FIFO has been filled up, all bits are transferred to the transfer register, and the FIFO is cleared. On output, when the FIFO is empty, data is transferred from the transfer register into the FIFO, and the transfer register is cleared. The value of the port counter is recorded when the FIFO is transferred to or from the double buffer, and made available as a “timestamp”.

The FIFO can serialise (output) and deserialise (input) data. On output, data is shifted out starting with the least significant bit (nibble, byte, or 16-bit entity). On input, the data is read in starting with the least-significant bit.

3.1.2 Strobing

Up to two pins can be used for strobing of the data. This corresponds to four strobing modes:

- Bidirectional strobing, where both an input strobe and an output strobe are used. When both strobes are high, the data is ready and can be sampled.
- Master strobing, where only an output strobe is generated by the XCore. This strobe is pulled high to signal that data is available (OUT) or to signal that there is space to input data (IN). This is equivalent to bidirectional strobing with a high input strobe.

- Slave strobing, where only an input strobe is read by the XCore. Data is only input or output when the input strobe is high. This is equivalent to bidirectional strobing with a high output strobe.
- Implicit strobing, where data is transferred every clock cycle. This is the equivalent of bidirectional strobing with both strobes high.

If the port is an input port, then data is clocked in from the pins into the FIFO, and then subsequently made available to the ISA (when the FIFO is full). If the port is an output port, then data is clocked out from the FIFO onto the output pins. Data has to be supplied to the FIFO fast enough to keep up with the clocked strobing.

In the case of bidirectional or master strobing on an input port, the output strobe is high if there is room in the FIFO, and low otherwise. On an output port, the output strobe is high if there is data in the FIFO, and low otherwise.

3.1.3 Timed and conditional I/O

Input operations can be delayed by making the input *timed* and/or *conditional*. If an input is timed then the port shall wait until the port counter has reached the specified value. If an input is conditional, then the port shall wait until the specified input condition is met. If both timing and conditions are used, then the input will first wait for the specified value on the port counter, and then for the specified condition. While waiting, the FIFO clocks data in according to the strobing mode. Any data that overflows is discarded.

Output operations can be delayed by making the output *timed*. If an output is timed, then the port shall wait until the port counter has reached the specified value. While waiting, the FIFO clocks data out according to the strobing mode. If there is insufficient data in the FIFO, the last data values held are held on the pins.

The port counter is 16 bits wide and counts edges on its associated clock.

When performing a conditional input on a buffered port, the strobe output is kept high while the condition does not match; any non matching input is discarded. Then the strobe is kept high for as long as there is space in the FIFO and double buffer. When performing a timed input on a buffered port, the strobe output is kept low until the port counter reaches the specified value; at that stage, the

strobe is pulled high, requesting data, until the FIFO and secondary buffer are filled up. When performing a timed output on a buffered port, the strobe output is kept low until the port counter reaches the specified value; at that stage, the strobe is pulled high, outputting data, until the FIFO and secondary buffer are empty.

3.2 Sampling input mode

When a port is configured in sampling mode, the Xcore I/O instructions in the ISA communicate directly with the hardware pins. When an IN is performed in sampling mode, the data latched at the previous clock edge is returned. If an IN is performed twice within a clock-cycle, then the second IN will block until the next clock edge. Input can be timed, in which case the input operation is performed immediately after the port counter reaches the specified value. The data that is input is the data actually clocked in at that specific moment.

Input can be delayed until a condition is met, for example, until the port does not equal the value '0xA'. This condition stays on the port, hence only matching data will be read until the condition is reset.

When using a WAIT instruction to wait for conditional input, the data must be input when the WAIT completes - this will return the data that matched the condition. If this data is not read, a later read will return the matched data.

3.3 Bidirectional mode

When a port is configured in bidirectional mode, it will tri-state when an IN operation is performed, and will start to drive when an OUT operation is performed. In the case of a timed IN or OUT, a change in tri-state or drive mode is delayed until the port counter reaches the specified value. When a conditional IN is performed, the input pins are tristated on executing the IN instruction (not on a SETD or SETC in the case of a conditional in). No buffering is available in Bidirectional mode. IN operations behave as in the “Sampling input mode” output operations behave as buffered without ready wires.

3.4 Clocks

All ports are synchronised to a clock provided by a *clock block*. By default, the ports are clocked from a clock block that runs at the reference frequency of 100 MHz. A port can be clocked from a different clock block, which can be driven using either a divided reference clock, or using a clock from an external source. Note that any external clock is itself clocked-in using the 400 MHz core clock. When using ports with a ready-input signal, this ready input is dealt with by the clock block.

There are five clock blocks available for general use. These are not allocated by the compiler or architecture, and like ports have to be assigned by the programmer. They are referred to by using one of CLKBLK_1 to CLKBLK_5. The reference clock block is denoted XS1_CLKBLK_REF.

clock blocks govern port delays (timed input, timed output), port counters, and clocking of the signal. clock blocks can also generate a clock on an external pin.

3.5 Use from XC

All input and output operations can be timestamped by adding "`@ variable`" to the right of an input or output statement.

3.5.1 Buffered ports

A buffered port is declared in XC using the keyword **`buffered`**. If a FIFO is required, this is denoted by inserting `:`*length* immediately after the keyword **`port`**. For example:

```
in buffered port:Y x;
out buffered port:Y z;
```

The port must subsequently be configured with the appropriate clock input and ready signals; unconfigured ports have no defined behaviour. The configuration can be performed using a single function call, or can be written as a sequence of low-level operations:

Bidirectional Strobe assigning ready pins for bidirectional strobe to a port

```
void configure_in_port_handshake(
          void port p, in port readyin,
          out port readyout, clock clk);
void configure_out_port_handshake(
          void port p, in port readyin,
          out port readyout, clock clk,
          int initial);
```

Master Strobe assigning ready pin for master strobe to a port

```
void configure_in_port_strobed_master(
          void port p, out port readyout,
          clock clk);
void configure_out_port_strobed_master(
          void port p, out port readyout,
          clock clk, int initial);
```

Slave Strobe assigning ready pin for slave strobe to a port

```
void configure_in_port_strobed_slave(
          void port p, in port readyin,
          clock clk);
void configure_out_port_strobed_slave(
          void port p, in port readyin,
          clock clk, int initial);
```

No Strobe Configure a port that needs no strobing

```
void configure_in_port_no_ready(
          void port p, clock clk);
void configure_out_port_no_ready(
          void port p, clock clk, int initial);
```

Syntax for input:

- `x :> y;`

- `x @ t :> y;`
- `x when pinsneq(1) :> y;`
- `x @ t when pinsneq(1) :> y;`

Syntax for output:

- `x <: y;`
- `x @ t <: y;`

3.5.2 Sampling Input

A port is declared as sampling input by using `in port` as the type. Subsequently, a timed input can be performed:

```
in port x;

... x @ t :> y ...
```

3.5.3 Bidirectional I/O

A port is declared as a bidirectional port by using `port` as the type. Subsequently, output and input operations can be interleaved with only few restrictions:

```
port x;

... x :> y ...
... x @ t :> y ...
... x when pinsneq(1) :> y ...
... x <: y ...
... x @ t <: y ...
```

It is illegal to perform a `select` on the port immediately after an output.

3.5.4 Clock output

A clock can be output onto a pin by configuring the clock block accordingly:

```
void configure_port_clock_output
          (void port p, clock c)
```

3.6 Use from Assembly

3.6.1 Buffered modes

Before a buffered port can be used, it must be switched on. The code below switches the port on and configures it to be buffered. After that, one of the four following code segments is required to set it to the appropriate strobing mode.

```
SETC     R0,SETC_INUSE_ON
LDC      R5,XS1_CLKBLK_REF; not required
SETCLK   R5,R0         ; after reset
SETC     R0,SETC_BUF_BUFFERS
LDC      R1,width
SETTW    R0,R1
```

In order to configure the bidirectional strobe, the ready pins have to be assigned to a port. The ready input is assigned via the clock. The code below assumes that R0 is the port to be configured, R1 the ready input wire, R2 the ready output pin, and R3 the clock.

```
SETRDY    R1,R3
SETCLK    R3,R0
SETC      R0,CTRL_RDY_HANDSHAKE
SETC      R0,CTRL_RUN_CLRBUF
SETRDY    R0,R2
SETC      R2,CTRL_PORT_READYPORT
```

An output port needs to be initialised with its initial value (R4) prior to configuration. In addition, the first two lines will cause the port to be reset to using the reference clock.

```
OUT       R4,R0
SYNCR     R0
SETRDY    R1,R3
SETCLK    R3,R0
SETC      R0,CTRL_RDY_HANDSHAKE
SETC      R0,CTRL_RUN_CLRBUF
SETRDY    R0,R2
SETC      R2,CTRL_PORT_READYPORT
```

In order to configure the master strobe, the ready output pin has to be assigned to a port. The code below assumes that R0 is the port to be configured, R2 the ready output pin, and R3 the clock.

```
SETCLK    R3,R0         ; not required if REF
SETC      R0,CTRL_RDY_MASTER
SETC      R0,CTRL_RUN_CLRBUF
SETRDY    R0,R2
SETC      R2,CTRL_PORT_READYPORT
```

An output port needs to be initialised with its initial value (R4) prior to configuration. In addition, the first two lines will cause the port to be reset to using the reference clock.

```
OUT       R4,R0
SYNCR     R0
SETCLK    R3,R0         ; not required if REF
SETC      R0,CTRL_RDY_MASTER
SETC      R0,CTRL_RUN_CLRBUF
SETRDY    R0,R2
SETC      R2,CTRL_PORT_READYPORT
```

In order to configure the slave strobe, the ready input pin has to be assigned to a port. The ready input is assigned via the clock. The code below assumes that R0 is the port to be configured, R1 the ready input wire, and R3 the clock.

```
SETRDY    R1,R3
```

```
SETCLK    R3,R0
SETC      R0,CTRL_RDY_STROBED
SETC      R0,CTRL_RDY_SLAVE
SETC      R0,CTRL_RUN_CLRBUF
```

An output port needs to be initialised with its initial value (R4) prior to configuration. In addition, the first two lines will cause the port to be reset to using the reference clock.

```
OUT       R4,R0
SYNCR     R0
SETRDY    R1,R3
SETCLK    R3,R0
SETC      R0,CTRL_RDY_STROBED
SETC      R0,CTRL_RDY_SLAVE
SETC      R0,CTRL_RUN_CLRBUF
```

In order to configure implicit strobing, the clock has to be assigned to a port. The code below assumes that R0 is the port to be configured and R3 the clock.

```
SETCLK    R3,R0
SETC      R0,CTRL_RDY_NOREADY
SETC      R0,CTRL_RUN_CLRBUF
```

An output port needs to be initialised with its initial value (R4) prior to configuration. In addition, the first two lines will cause the port to be reset to using the reference clock.

```
OUT       R4,R0
SYNCR     R0
SETC      R0,CTRL_RDY_NOREADY
SETC      R0,CTRL_RUN_CLRBUF
```

In all cases the FIFO can be set by executing the following code fragment:

```
XXXX
```

Input from a buffered port is performed by using one of the following four code fragments:

```
; Input
    IN   y, x

; Timed input; the IN can be preceeded by a WAIT.
    SETPT  x, t
    ...
    IN   y, x

; Conditional input; the IN can be preceeded by a
; WAIT.
    SETC  x, COND
    SETD  x, DATA
    ...
    IN   y, x

; Timed conditional input; the IN can be preceeded
; by a WAIT.
    SETPT  x, t
    SETC  x, COND
    SETD  x, DATA
    ...
    IN   y, x
```

Conditional input (SETC) is automatically reset to unconditional on a match. This allows subsequent data to be read in without delay into the FIFO. A timed conditional input will not switch the condition on until the counter has reached the specified value.

Output to a buffered port is performed by using one of the following two code fragments:

```
; Output
    OUT  y, x
```

```
; Timed output; the OUT must be in time, otherwise
; data will be INPUT at that time. A timed OUT
; will not block, but a subsequent SETPT or OUT
; will block if the previous OUT has yet to happen
    SETPT  x, t
    ...
    OUT  y, x
```

3.6.2 Sampling input

A sampling input port must be switched on prior to use. After that, the clock must be set. On reset, all clocks are set to XS1_CLKBLK_REF.

```
     SETC    R0,SETC_INUSE_ON
     LDC     R5,XS1_CLKBLK_REF; warm reset only
     SETCLK  R5,R0
```

There are then three methods for inputting data:

```
; Timed input; the IN can be preceeded by a WAIT.
    SETPT  x, t
    ...
    IN   y, x
```

Behaviour of the SETC instruction is subtly different from a buffered SETC; the condition is *not* reset, hence only matching data is read until the port is made unconditional.

3.6.3 Bidirectional I/O

A bidirectional port must be switched on prior to use. After that, the clock must be set. On reset, all clocks are set to XS1_CLKBLK_REF.

```
     SETC    R0,SETC_INUSE_ON
```

```
    LDC     R5,XS1_CLKBLK_REF; warm reset only
    SETCLK  R5,R0
```

Bidirectional input/output uses the same sequence of instructions for input, but no WAIT can be issued in the case of a conditional IN that follows an OUT. In order to perform an output, one of the following two sequences is used.

```
; Input
    IN   y, x

; Output
    OUT  y, x

; Timed input; the IN is guaranteed to return the
; data read at the specified time.
    SETPT  x, t
    ...
    IN     y, x

; Timed output; the OUT must be in time, otherwise
; data will be INPUT at that time. The OUT will not
; block, but a subsequent SETPT or OUT will block
; if the OUT has yet to happen.
    SETPT  x, t
    ...
    OUT    y, x

; Conditional input; the IN can be preceeded by
; a WAIT instruction
    SETC  x, COND
    SETD  x, DATA
    ...
    IN   y, x
```

3.6.4 Clock output

In order to output the clock of clock block R5 to port R0 use the following assembly sequence:

```
SETCLK    R5,R0
SETC      R0,CTRL_PORT_CLOCKPORT
```

3.7 Drive modes

By default a pin is driven both high and low. It can be set to just drive high on a 1 (and become high impedance on 0). This is done by calling the following C-primitive:

```
set_port_pull_up(void port p);
```

Or the following assembly instruction:

```
SETC      R0,CTRL_DRIVE_PULL_UP
```

A 1-bit port can be set to invert all data on both input and output

```
set_port_inv(void port p);
```

Or the following assembly instruction:

```
SETC      R0,CTRL_INV_INVERT
```

A PEEK instruction is available to inspect the value of the port pins bypassing all mechanisms described before. This can be used to, for example, check whether a pull-up port is high or low.

```
PEEK      R0,x
```

Gets the current value on port x into R0.

3.8 Hardware Port pin-out

The XS1 architecture has 16 logical ports, and the total number of pins by far exceeds the number of pins on the package. Ports, and Xlinks are therefore *multiplexed*, and there is a defined *precedence* when overlapping ports and links are used. The

3.8.1 Precedence

The mapping of ports to pins is shown in Table 1. This table lists for each pin which ports and links *can be connected* to it. Links and ports on the left hand side of the table have precedence over ports on the right hand side of the table. Each port is identified by its width (the first number 1, 4, 8, 16, or 32) and a letter that distinguishes multiple ports of the same width (A-P). The bits of the port are identified with a superscripted digit 0-31. Links are identified by means of a single letter identifier A-D. The wires of a link are identified by means of a superscripted digit 0-4.

The port or link that is actually connected to a pin is determined by the program running on the XS1. For each core, software can enable ports and links as required:

- If a link is enabled, then this link will have access to the pins; the pins of the underlying ports will be disabled.
- If a port is enabled then it will overrule any ports with higher widths that it shares its pins with.

For example, suppose that on the software on core 2 link enable link C, and ports 32A, 4C, and 8B. In that case:

- X2D01 - X2D10 will be connected to link C.
- X2D14, X2D15, X2D20, X2D21 will be connected to port 4C.
- X2D16 - X2D19 will be connected to bits 2 to 5 of port 8B.
- X2D49 - X2D70 will be connected to bits 0 to 19 of port 32A.

	⇐ highest		Precedence			lowest ⇒
Pin	link	1-bit ports	4-bit ports	8-bit ports	16-bit ports	32-bit port
XnD00		1A				
XnD01	A^4 in/out	1B				
XnD02	A^3 in/out		4A^0	8A^0	16A^0	32A^{20}
XnD03	A^2 in/out		4A^1	8A^1	16A^1	32A^{21}
XnD04	A^1 in/out		4B^0	8A^2	16A^2	32A^{22}
XnD05	A^0 in/out		4B^1	8A^3	16A^3	32A^{23}
XnD06	A^0 out/in		4B^2	8A^4	16A^4	32A^{24}
XnD07	A^1 out/in		4B^3	8A^5	16A^5	32A^{25}
XnD08	A^2 out/in		4A^2	8A^6	16A^6	32A^{26}
XnD09	A^3 out/in		4A^3	8A^7	16A^7	32A^{27}
XnD10	A^4 out/in	1C				
XnD11		1D				
XnD12		1E				
XnD13	B^4 in/out	1F				
XnD14	B^3 in/out		4C^0	8B^0	16A^8	32A^{28}
XnD15	B^2 in/out		4C^1	8B^1	16A^9	32A^{29}
XnD16	B^1 in/out		4D^0	8B^2	16A^{10}	
XnD17	B^0 in/out		4D^1	8B^3	16A^{11}	
XnD18	B^0 out/in		4D^2	8B^4	16A^{12}	
XnD19	B^1 out/in		4D^3	8B^5	16A^{13}	
XnD20	B^2 out/in		4C^2	8B^6	16A^{14}	32A^{30}
XnD21	B^3 out/in		4C^3	8B^7	16A^{15}	32A^{31}
XnD22	B^4 out/in	1G				
XnD23		1H				
XnD24		1I				
XnD25		1J				
XnD26			4E^0	8C^0	16B^0	
XnD27			4E^1	8C^1	16B^1	
XnD28			4F^0	8C^2	16B^2	
XnD29			4F^1	8C^3	16B^3	
XnD30			4F^2	8C^4	16B^4	
XnD31			4F^3	8C^5	16B^5	
XnD32			4E^2	8C^6	16B^6	
XnD33			4E^3	8C^7	16B^7	
XnD34		1K				
XnD35		1L				
XnD36		1M		8D^0	16B^8	
XnD37		1N		8D^1	16B^9	
XnD38		1O		8D^2	16B^{10}	
XnD39		1P		8D^3	16B^{11}	
XnD40				8D^4	16B^{12}	
XnD41				8D^5	16B^{13}	
XnD42				8D^6	16B^{14}	
XnD43				8D^7	16B^{15}	
XnD49	C^4 in/out					32A^0
XnD50	C^3 in/out					32A^1
XnD51	C^2 in/out					32A^2
XnD52	C^1 in/out					32A^3
XnD53	C^0 in/out					32A^4
XnD54	C^0 out/in					32A^5
XnD55	C^1 out/in					32A^6
XnD56	C^2 out/in					32A^7
XnD57	C^3 out/in					32A^8
XnD58	C^4 out/in					32A^9
XnD61	D^4 in/out					32A^{10}
XnD62	D^3 in/out					32A^{11}
XnD63	D^2 in/out					32A^{12}
XnD64	D^1 in/out					32A^{13}
XnD65	D^0 in/out					32A^{14}
XnD66	D^0 out/in					32A^{15}
XnD67	D^1 out/in					32A^{16}
XnD68	D^2 out/in					32A^{17}
XnD69	D^3 out/in					32A^{18}
XnD70	D^4 out/in					32A^{19}

Table 1 *Available links and ports for each pin in order of decreasing precedence.*

Normally, the system designer will be ensure that there is no overlap, but the precedence has been designed so that, if required, portions of the wider ports can still be used when overlapping narrower ports are used. Note that the relative place of input and output links depends on the core number in order to simplify routing tracks on a PCB.

3.8.2 Banks

Table 1 is divided in six parts which are six *banks*. Different packaging options export different numbers of banks; all banks of all cores may be exported (512 BGA), or only four banks of each of two cores (144 BGA). The first few banks have a selection of 1, 4, and 8 bit ports, and a link each. Banks further down incorporate port 32. On small packages the 32-bit port will not be available.

3.8.3 Port identifiers

Each port is, architecturally, represented with a *bpw*-bit identifier, called a resource identifier. The least significant byte of a port-resource-identifier is 0 (identifying this as a port as opposed to for example a channel or timer), the next bytes identifies the port and the width of the port. A full list of ports is given in Table 2. Note that in almost all cases one can use `XS1_PORT_1E` rather than `0x10600` since the include file `xs1.h` contains all mappings.

Acknowledgements

Many thanks to Robert Jarnot (JPL) and Koen Buskes for their feedback on earlier versions of this document.

References

[1] XC library documetation. Website, 2008. https://www.xmos.com/support/documentation.

Port name	Resource identifier
XS1_PORT_32A	0x200000
XS1_PORT_16A	0x100000
XS1_PORT_16B	0x100100
XS1_PORT_8A	0x80000
XS1_PORT_8B	0x80100
XS1_PORT_8C	0x80200
XS1_PORT_8D	0x80300
XS1_PORT_4A	0x40000
XS1_PORT_4B	0x40100
XS1_PORT_4C	0x40200
XS1_PORT_4D	0x40300
XS1_PORT_4E	0x40400
XS1_PORT_4F	0x40500
XS1_PORT_1A	0x10200
XS1_PORT_1B	0x10000
XS1_PORT_1C	0x10100
XS1_PORT_1D	0x10300
XS1_PORT_1E	0x10600
XS1_PORT_1F	0x10400
XS1_PORT_1G	0x10500
XS1_PORT_1H	0x10700
XS1_PORT_1I	0x10a00
XS1_PORT_1J	0x10800
XS1_PORT_1K	0x10900
XS1_PORT_1L	0x10b00
XS1_PORT_1M	0x10c00
XS1_PORT_1N	0x10d00
XS1_PORT_1O	0x10e00
XS1_PORT_1P	0x10f00

Table 2 *Resource identifiers for all XS1 ports*

[2] Douglas Watt and Richard Osborne and David May. XC Reference Manual (8.7). Website, 2008. http://www.xmos.com/published/xc87.

[3] XC tutorial. Website, 2008. https://www.xmos.com/support/documentation.