XMOS

AN03002: XCORE Port Buffering

Publication Date: 2025/3/11
Document Number: XM-015260-AN v1.0.0

IN THIS DOCUMENT

1 Overview

The XMOS architecture provides buffers that can improve the performance of programs that perform input or output on clocked ports. This document describes how to use these buffers and is intended to be read in conjunction with application notes

- AN03000: XCORE Input and Output, and
- AN03001: XCORE Clocked Input and Output.

In addition, the reader should be familiar with the concept of ports as described in

- AN03007: XCORE Ports, and
- AN02039: Ports, Pins, and the XN file.

A port buffer can hold data output by the processor until the next falling edge of the port's clock, allowing the processor to execute other instructions during this time. It can also store data sampled by a port until the processor is ready to input it. Using these buffers, a single processing thread can perform I/O operations on multiple ports in parallel.

2 Using a Buffered Port

Fig. 1 shows a block diagram of an example comprising 8-bit data input on port 8A and an 8-bit output on port 8B, with a clock on port 1A. Both input and output are buffered, and share the same clock.

2.1 Programming a Buffered Port

The following programme uses a buffered port function to decouple the sampling and driving of data on ports from a computation.

```
#include <xcore/port.h>
#include <xcore/clock.h>
#include <xs1.h>

port_t inP     = XS1_PORT_8A;
port_t outP    = XS1_PORT_8B;
port_t inClock = XS1_PORT_1A;
xclock_t clk   = XS1_CLKBLK_1;

int main(void) {
    port_start_buffered(inP, 8);
    port_start_buffered(outP, 8);
    port_enable(inClock);
    clock_enable(clk);

    clock_set_source_port(clk, inClock);
```

(continues on next page)

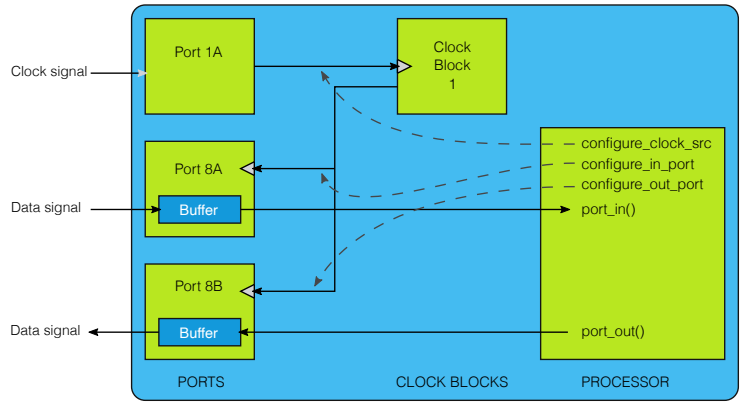

Fig. 1: Port configuration diagram

(continued from previous page)

```
    port_set_clock(inP, clk);
    port_out(outP, 0);
    port_set_clock(outP, clk);

    clock_start(clk);

    for (int i = 0; i < 10; i++) {
        int x;
        x = port_in(inP);
        port_out(outP, x+1);
    }
}
```

The programme configures the ports **inP**, **outP** and **inClock** as illustrated in Fig. 1.

The declarations:

```
port_t inP     = XS1_PORT_8A;
port_t outP    = XS1_PORT_8B;
port_t inClock = XS1_PORT_1A;
xclock_t clk   = XS1_CLKBLK_1;
```

declare four resources, three ports (**inP**, **outP**, and **inClock**) and a clock-block (**clk**). The ports are set to refer to port **8A**, **8B**, and **1A**, and the clock block is set to **clock-block_1**. These variables are constants that refer to four hardware blocks.

The first four lines of the the main program:

```
port_start_buffered(inP, 8);
port_start_buffered(outP, 8);
port_enable(inClock);
clock_enable(clk);
```

enable the four resources. Each resource must be enabled exactly once before it can be used; when the chip boots all resources are dormant. The **inClock** and **clk** resources are just enabled, whereas the **inP** and **outP** resources are set to be **buffered** with a transfer width of eight bits. In this case, the port has eights pins, which means that exactly one set of eight pin values can be buffered.

The following statements connect the ports and clock-block together:

```
clock_set_source_port(clk, inClock);
```

configures the 1-bit input port **inClock** to provide edges for the clock-block **clk**.

```
port_set_clock(inP, clk);
```

configures the input port **inP** to be clocked by the clock **clk**.

```
port_out(outP, 0);
port_set_clock(outP, clk);
```

configures the output port **outP** to be clocked by the clock **clk**, with an initial value of 0 driven on its pins. The clock block is then started with the statement:

```
clock_start(clk);
```

The remainder of the program is a simple for-loop that reads a value from the input port, increments it, and writes the result to the output port.

Fig. 2 shows example input stimuli and expected output for this program. It also shows the relative waveform of the statements executed in the **for** loop by the processor.

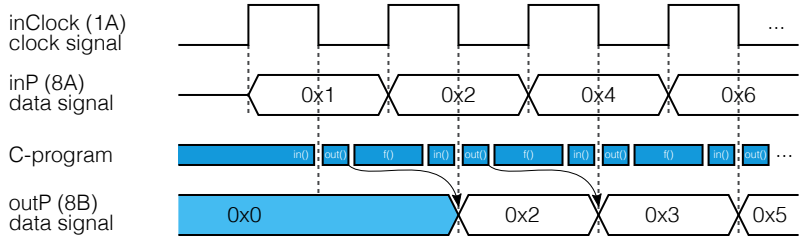

Fig. 2: Waveform diagram relative to processor execution

The first three values input are 0x1, 0x2 and 0x4, and in response the values output are 0x2, 0x3 and 0x5.

2.2 Buffered Port Hardware Logic

Fig. 3 illustrates the buffering operation in the hardware. It shows the processor executing the **for** loop that outputs data to the port. The port buffers this data so that the processor can continue executing subsequent instructions while the port drives the data previously output for a complete period. On each falling edge of the clock, the port takes the next byte of data from its buffer and drives it on its pins. As long as the instructions in the loop execute in less time than the port's clock period, a new value is driven on the pins on every clock period.

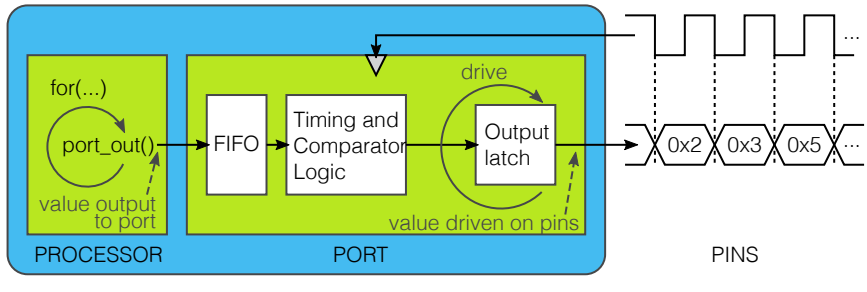

Fig. 3: Port hardware logic

The fact that the first input statement is executed before a rising edge means that the input buffer is not used. The processor is always ready to input the next data before it is sampled, which causes the processor to block, effectively slowing itself down to the rate of the port. If the first input occurs after the first value is sampled, however, the input buffer holds the data until the processor is ready to accept it and each output blocks until the previously output value is driven.

Caution: Timed operations represent time in the future. The waveform and comparator logic allows timed outputs to be buffered, but for timed and conditional inputs the buffer is emptied before the input is performed.

3 Synchronising Clocked I/O on Multiple Ports

By configuring more than one buffered port to be clocked from the same source, a single thread can cause data to be sampled and driven in parallel on these ports. The programme below first synchronises itself to the start of a clock period, ensuring the maximum amount of time before the next falling edge, and then outputs a sequence of 8-bit character values to two 4-bit ports that are driven in parallel.

```
#include <xcore/port.h>
#include <xcore/clock.h>
#include <xs1.h>

port_t   p       = XS1_PORT_4E;
port_t   q       = XS1_PORT_4F;
port_t   inClock = XS1_PORT_1A;
xclock_t clk     = XS1_CLKBLK_1;

int main(void) {
    int count;
    port_start_buffered(p, 4);
    port_start_buffered(q, 4);
    port_enable(inClock);
    clock_enable(clk);

    clock_set_source_port(clk, inClock);
    port_out(p, 0);
    port_out(q, 0);
    port_set_clock(p, clk);
    port_set_clock(q, clk);

    clock_start(clk);

    port_out(p, 0);
    count = port_get_trigger_time(p) + 3;
    port_set_trigger_time(p, count);
    port_set_trigger_time(q, count);

    for (char c='A'; c<='Z'; c++) {
        port_out(p, (c & 0xF0) >> 4);
        port_out(q, (c & 0x0F)     );
    }
}
```

The statements:

```
port_out(p, 0);
count = port_get_trigger_time(p) + 3;
port_set_trigger_time(p, count);
port_set_trigger_time(q, count);
```

cause the processor to set both ports to start outputting on a “safe” edge in the future. As the input clock is running asynchronous to our program, the first clock edge may arrive at any time, but by picking an edge ahead (in this case three clocks), we now have time to start the for-loop and ensure that both `port_out` calls in the for-loop are executed before clock edge number three arrives. From there on, the calls to `port_out` will work on the same clock edge.

Fig. 4 shows the data output by the processor and driven by the two ports.

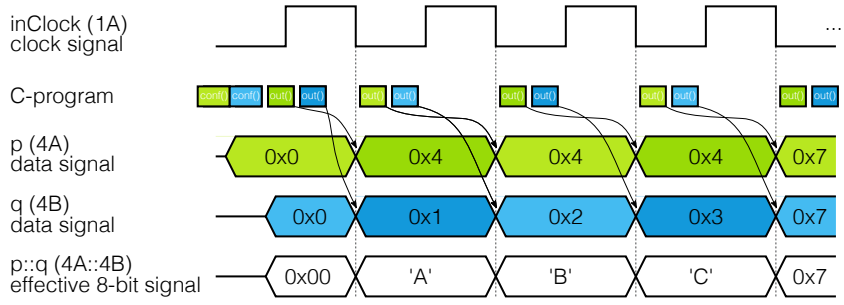

Fig. 4: Processor synchronizing data on two output ports

4 Summary of Buffering Behavior

The semantics for I/O on clocked buffered ports are summarised as follows.

Output Statements

- An output inserts data into the port's FIFO. If there is no room for the data (the FIFO is full) the processor waits until there is room and then inserts data.
- At most one data value is removed from the FIFO and driven by the port per period of its clock.
- A timed output inserts data into the port's FIFO for driving when the port counter equals the specified time. The processor waits if the FIFO is full.
- A *timestamped* output causes the processor to wait until the output is driven (required to determine the timestamp value).
- If the FIFO is empty, then the data driven on one edge continues to be driven on subsequent edges.

Input Statements

- At most one value is sampled by the port and inserted into its FIFO per period of its clock. If the FIFO is full, its oldest value is dropped to make room for the most recently sampled value.
- An input removes data from a port's FIFO. If the FIFO is empty, the processor waits for data.
- Timed and conditional inputs cause any data in the FIFO to be discarded and then behave as in the unbuffered case.

Copyright © 2025, All Rights Reserved.

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

XMOS, XCORE, VocalFusion and the XMOS logo are registered trademarks of XMOS Ltd. in the United Kingdom and other countries and may not be used without written permission. Company and product names mentioned in this document are the trademarks or registered trademarks of their respective owners.