AN02030: Improving IO response times using FAST or PRIORITY modes

Publication Date: 2024/11/21
Document Number: XM-015205-AN v1.0.0

IN THIS DOCUMENT

XCORE processors can respond very quickly to external stimuli. This app note shows the effect of two modes that a thread can be set into that further improves the response time. These modes come at a cost (higher power consumption and less MIPS for other threads), but they can be helpful in tight IO loops where the timing requirement is close to the XCORE performance.

The two modes discussed are FAST mode and PRIORITY mode. The purpose of FAST mode is to specifically improve the IO response time; the purpose of PRIORITY mode is more broad, but in the context of this app note we only look at using PRIORITY mode for improving the IO response time of a single thread. Using PRIORITY modes in multiple threads is outside the scope of this app note. FAST and PRIORITY modes are defined in the XS3 architecture manual.

This app-note relies on an in-depth understanding of the XCORE pipeline. It shows the improved response times relative to an external stimulus, but the techniques discussed also work on responses to other XCORE resources, such as inputs from channel-ends.

1 FAST mode

FAST is one of the mode bits in the status register of a thread. When in FAST mode any instruction that would normally pause, will re-execute as if the resource was marked ready. This as opposed to normal execution where the thread would be taken out of the scheduling queue until such a time where the resource is ready.

The purpose of this mode is to make threads slightly more responsive than they normally are. Because the thread is already executing, it will save up to four core-clock-cycles in waking a thread up.

In the best case, it may be that the resource becomes ready just before the instruction goes into the execution stage of the pipeline, and the signal will go straight from the port into the code with as little delay as possible.

In the worst case, it may be that the port becomes ready just after the instruction has entered the execution stage of the pipeline, and it just misses it.

FAST mode enables you to remove between one and four core clock cycles (say 5-7 ns assuming a 600 MHz clock) of the reaction time of a port or other resource.

A thread is set to execute in FAST mode by executing the following line:

```
local_thread_mode_set_bits(thread_mode_fast);          // Comment out to not have fast mode
```

For this function to work, you need to include the `xcore/thread.h` file, which is automatically included through `xcore/parallel.h` in our example program. FAST mode can be cleared through a call to `local_thread_mode_clear_bits(thread_mode_fast)`. The functions `local_thread_mode_set_bits()` and `local_thread_mode_clear_bits()` dynamically change the setting of the current thread, enabling FAST mode (or PRIORITY mode below) to be switched on and off as desired.

2 PRIORITY mode

PRIORITY is another mode bit in the status register of a thread. When in PRIORITY mode the thread is queued for the pipeline in preference to the threads that are not in priority mode.

The purpose of this mode is to give threads a larger slice of the execution bandwidth of the processor to the prioritised thread. The pipeline is five stages long, so suppose that there are, for example, a total of seven threads running with thread 0 being the high priority thread the schedule of threads will be as follows:

- **0** 1 2 3 4 **0** 5 6 1 2 **0** 3 4 5 6 **0** 1 2 3 4 **0** 5 6 1 2 **0** 3 4 5 6 ...

That is, thread 0 gets 20% of the execution time, and the remaining 80% are shared between the other 6 threads giving them 13.33% each. Assuming we have a 600 MHz processor that works out as an issue rate of 120 MHz for thread 0 and (600-120)/6 = 80 Mhz for each of the other six threads. Had the thread not been running in fast mode each thread would have had 600/7 = 85.7MHz each. When running a single priority thread with

- seven other threads, these see a 8.5% reduction in speed;
- six other threads, these see a 6.7% reduction in speed;
- five other threads, these see a 4% reduction in speed.
- Up to four other threads there is no noticeable reduction in speed for the other threads.

In this app note we look specifically into how PRIORITY affects the latency when responding to an IO event. We only discuss the case where there is one single priority thread. When a resource becomes ready, a thread that has PRIORITY set will immediately enter the pipeline, therefore responding faster than when in normal mode.

PRIORITY mode enables you to remove up to nine core clock cycles (15 ns assuming a 600 MHz clock) of the reaction time of a port or other resource. This will come at the cost of a reduced issue rate of all other threads, so this must be taken into account when timing closing the other threads.

A thread is set to execute in PRIORITY mode by executing the following line:

```
local_thread_mode_set_bits(thread_mode_high_priority);  // Comment out to not have priority mode
```

For this function to work, you need to include the `xcore/thread.h` file, which is automatically included through `xcore/parallel.h` in our example program. PRIORITY mode can be cleared through a call to `local_thread_mode_clear_bits(thread_mode_high_priority)`.

3 Comparing FAST and PRIORITY modes

In PRIORITY mode, one or more threads are given priority access to the pipeline, enabling those threads to achieve a higher issue rate at the expense of all other threads. FAST mode specifically enables a shorter response time to accessing a resource.

PRIORITY mode significantly reduces the issue rate of other threads; FAST mode significantly increase power consumption of a waiting thread.

FAST and PRIORITY can both be switched on for fastest response. That will combine the disadvantages: it adds consumption and reduces the bandwidth to other threads.

4 Measurement method

We measure the effect of the different mode as follows:

- The test program runs in one of 4 x 8 = 32 combinations:
  - An IO-thread that is running either in NORMAL, FAST, PRIORITY, or FAST+PRIORITY mode (a total of four combinations).
  - Between zero and seven busy-threads; each comprising a `while(1)` loop (times eight combinations)
- For `i` in the range (0..400):
  - The IO-thread in the test program sets the input port up to wait for it to go high
  - The test-harness drives the port high at clock cycle X_i
  - The IO-thread sees the input port go high, and in response sets the output port high
  - The test-harness records the clocks cycle Y_i at which the output port goes high; We define Y_i - X_i to be the reaction time of the IO-thread.
- X_0 is a constant large enough to allow the program to start.

We want the test-harness to schmoo the stimulus arriving across all possible stages of the processor pipeline. To this end we time the next event to be either relative to the previous event (for even values of `i`) or relative to the response (for odd values of `i`). As `i` increases this creates all scenarios of the port becoming ready against the instruction's position in the pipeline.

5 The test program

The code for responding to the external stimulus is a very short function. It waits for the input port **p** to go high, and then briefly sets the output port **q** high:

```
void respond(port_t q, port_t p) {
    port_set_trigger_in_equal(p, 1);
    port_in(p);
    port_out(q, 1);
    port_out(q, 0);
}
```

The ports are initialised with a clock-block that is clocked of the core clock. That way the ports are clocked every core-clock cycle:

```
void init_port(port_t p, port_t q, xclock_t clk) {
    port_enable(p);
    port_enable(q);
    clock_enable(clk);
    clock_set_source_clk_xcore(clk);
    port_set_clock(p, clk);
    port_set_clock(q, clk);
    clock_start(clk);
}
```

The IO-thread first optionally sets the thread to fast mode, it then initialises the ports, and finally the responses in an infinite loop. The test harness will terminate the program when it has tried enough stimuli:

```
void iomain(void) {
    local_thread_mode_set_bits(thread_mode_high_priority);  // Comment out to not have priority mode
    local_thread_mode_set_bits(thread_mode_fast);           // Comment out to not have fast mode
    init_port(input, output, clk);
    port_out(output, 0);
    while(1) {
        respond(output, input);
    }
}
```

6 The test-harness

In addition to the code running on an XCORE we have a test-harness that toggles the input ports and measures the response times. This test-harness is stored in the `host` directory at the top level of this app note. After creating an instance of the simulator, the clock is toggled in the simulator until it is time to produce a pulse:

```
        while(clockcount != next_pulse_at) {
            status = xsi_clock(xsim);
            assert(status == XSI_STATUS_OK || status == XSI_STATUS_DONE);
            clockcount++;
        }
```

Now the pulse is created on the input pin for 15 clock cycles:

```
        status = xsi_drive_port_pins(xsim, "tile[0]", "XS1_PORT_1B", 1, 1);
        assert(status == XSI_STATUS_OK || status == XSI_STATUS_DONE );
        int ps = clockcount;
        for(int k = 0; k < 15; k++) {
            status = xsi_clock(xsim);
            assert(status == XSI_STATUS_OK || status == XSI_STATUS_DONE);
            clockcount++;
        }
        status = xsi_drive_port_pins(xsim, "tile[0]", "XS1_PORT_1B", 1, 0);
        assert(status == XSI_STATUS_OK || status == XSI_STATUS_DONE );
```

And finally the output pin is monitored until it has gone high, the results are printed, and the time for the next input pulse is calculated:

```
        unsigned samp;
        do {
            status = xsi_clock(xsim);
            assert(status == XSI_STATUS_OK || status == XSI_STATUS_DONE);
            clockcount++;
            status = xsi_sample_port_pins(xsim, "tile[0]", "XS1_PORT_1A", 1, &samp);
            assert(status == XSI_STATUS_OK || status == XSI_STATUS_DONE);
        } while(samp == 0);
        reactions[clockcount - next_pulse_at]++;
        if (i & 1) {
            next_pulse_at = clockcount + 1000 + i/2;
        } else {
            next_pulse_at = ps + 1000 + i/2;
        }
```

7 Results

The table below shows the results of running one-IO thread in FAST mode, normal mode, PRIORITY mode, or both FAST and PRIORITY modes; with up to seven busy threads. For each number of busy threads we measure the range of response times as measured in core clocks:

Thread count		Delay Measured			
Busy	IO	FAST	NORMAL	PRIORITY	FAST + PRIORITY
7	1	18..25	23..29	20	16..19
6	1	17..23	22..27	20	16..19
5	1	16..21	21..25	20	16..19
4	1	16..19	20..23	20	16..19
3	1	16..19	20..22	20	16..19
2	1	16..19	20..21	20	16..19
1	1	16..19	20	20	16..19
0	1	16..19	20	20	16..19

From this table we can derive the differences relative or normal mode. For brevity we just list the improvement in worst-case timings between normal mode and the other modes:

Thread		Worst case difference with normal mode		
Busy	IO	Fast	Priority	Fast + Priority
7	1	4	9	10
6	1	4	7	8
5	1	4	5	6
4	1	4	3	4
3	1	3	2	3
2	1	2	1	2
1	1	1	0	1
0	1	1	0	1

7.1 Comparing FAST mode against NORMAL mode

- The worst-case (WC) respone-time is better by up to four cycles when executed in fast mode.
- When only few threads are running, normal mode will always respond in a precise number of cycles: the thread becomes ready, and it gets going. A fast thread may just hit or miss the event and there is some variability.
- A system that has been designed for eight threads in fast mode, does not need fast mode when used with only six threads
- A system that has been designed for seven threads in fast mode, does not need fast mode when used with only five threads
- A system that has been designed for six threads in fast mode, does not need fast mode when used with only three threads
- Power consumption in FAST mode is higher, as the pipeline will now be executing instructions where it used to be idle.

7.2 Comparing PRIORITY mode and NORMAL mode

When running a single thread thread in high priority mode:

- PRIORITY responds up to nine clock cycles faster than normal mode.
- PRIORITY mode has a predictable response time.

Note: PRIORITY slows down the other threads.

7.3 Comparing PRIORITY + FAST modes and NORMAL mode

When running a single thread thread in high priority mode:

- PRIORITY responds up to ten clock cycles faster than normal mode.

Note: PRIORITY slows down the other threads.

7.4 Comparing PRIORITY mode and FAST mode

When running a single thread thread in high priority mode:

- PRIORITY threads respond faster than a thread in FAST mode when the total number of threads is greater than five.
- PRIORITY threads respond one cycles (worst case) slower than a thread in FAST mode when the total number of threads is five or less.

8 Summary

FAST and PRIORITY modes can be used independently or together to improve response times to an external signal.

- FAST mode does so at the expense of mostly power, as threads no longer wait but actively retry the instruction at the earliest opportunity
- PRIORITY mode does so at the expense of other threads as the prioritised thread will always be first in line to execute

9 Further reading

- The XS3 architecture manual.
- IO timings for xcore.ai.