Digital System Design
2016
Laboratory Notes – Throughput

Professor Jonathan H. Manton

Week 4

Type of Lab

This lab has homework.

Aims

1. To learn, by example, about pipelining.
2. To see how a Finite State Machine can allow a single multiplier to be re-used. (This is a precursor to the simple computer you will be building for Project 1.)
3. To see how Quartus automatically uses DSP cores for multiplication.

Prior Preparation

You should read these Lab Notes thoroughly prior to attending the lab session. If you are still unsure about pipelining, after reading through this Lab, you should try to read up about it (textbook, online etc).

Lab Session

The lab is based around computing x^4 where x is an unsigned 16-bit integer.

- How many bits are required to store x^4 in the worst case?

Part 1

Create a new Quartus Project based on the **Cyclone IV EP4CE22F17C6** chip. (This is the chip used for the DE0-Nano board. Recall that timing simulations are not available for Cyclone V chips.) Type in the following module.

```
module MyMult(input [15:0] x, output [63:0] xp4);
        wire [31:0] xp2 = x*x;
        wire [47:0] xp3 = xp2*x;
        assign xp4 = xp3*x;
endmodule
```

Compile the design.

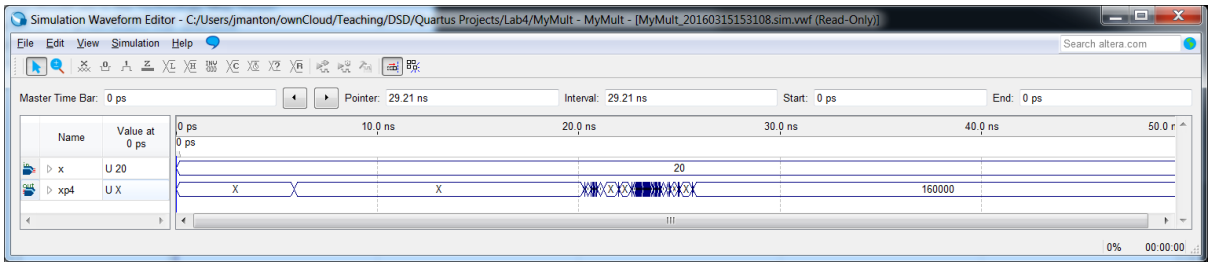
- You will receive a warning message that some of the pins are stuck at VCC or GND. Why?

Look at the RTL then the Technology Map Viewer. You should see in the Technology Map Viewer (click on the "+" symbol in the upper left-hand corners of the green boxes) that dedicated multiplication circuitry has been created for you. (Another hint is that, in the compiler messages, it says Inferred 3 megafunctions from design logic.) Moreover, in the first multiplication, if you click again on the green box to go down even deeper, you will see "DSP_MULT", suggesting that the DSP blocks on the FPGA are being used.

Select the whole DSP_MULT block. (Click away from any label but inside the DSP_MULT rectangle.) Then right-click, Locate Node > Locate in Chip Planner. Zoom in on the selected region. It should look something like the following. (If it is blank, go back and Compile your design.)

- In your lab book, make a sketch (or a copy or a print out) of the Chip Planner view.
    - Show where the Logic Elements (LEs) you are using are.
    - Show where the DSP blocks you are using are.

Test out the multiplier by performing a timing simulation. For example, recreate the following.



Unless you are an expert at binary multiplication, you should switch the radix to "unsigned decimal". The above shows that twenty to the power of four equals 160,000.

- In your lab book, draw a rough sketch of the output "xp4".
- Were there glitches?
- How long did it take to compute the correct answer?
- If you change the input, do the glitches change?
- If you change the input, does the time it takes to get the final answer change?

- Redo the simulations, but this time, keep "x" at 0 for the first 30 ns, then set "x" to 65432 from 30 ns till the end of the simulation (say 80 ns).
    - Were there any changes in terms of glitches, or timing?

Part 2

Being able to compute 65432^4 in under 25 ns is impressive. But it is not fast enough! Modify your Verilog so that you do not need xp3. Instead, assign xp4 = xp2*xp2;.

Perform a timing simulation.

- Is this design faster or slower than in Part 1?
    - By a lot or a little?
    - Does this surprise you?
    - Try to explain why you observed what you did.

Part 3

There is a stream of numbers coming in, and we must compute what each one is to the power of four. We could use our module from before to achieve this, as follows. (I have called it a "pipeline" already, because data comes in one end and data leaves the other end, but we will see some tricks for increasing the throughput in Part 4, in which case it becomes a real example of pipelining.)

```
module Pipeline(input [15:0] x, input clk, output reg [63:0] xp4);
        reg [15:0] data_in;
        wire [63:0] data_out;
        MyMult worker(.x(data_in), .xp4(data_out));

        always @(posedge clk) begin
                data_in <= x;
                xp4 <= data_out;
        end
endmodule
```

Change the top-level entity name to Pipeline. Compile the design, and look at the RTL.

- In your lab book, draw the RTL, and explain what is going on.

From Part 2, it looks like the multiplication takes more than 20 ns to complete. If the clock is going too fast, then the flip-flop after the multiplier will try to "take a photograph" of its input too early, before the multiplication has finished, and the design will not work.

Thankfully, it is largely Quartus' responsibility to work out how much time is required. (We could not do it perfectly ourselves because we do not know how long the flip-flop needs for the input to be stable before it "takes a photograph", and we also do not know exactly how long the multiplication will take. Sure, we looked at a simulation, but temperature affects speed, as does imperfections in the silicon and so forth. Altera have built into Quartus all the parameters specific to their FPGAs to ensure Quartus can compute what a safe clock frequency is.)

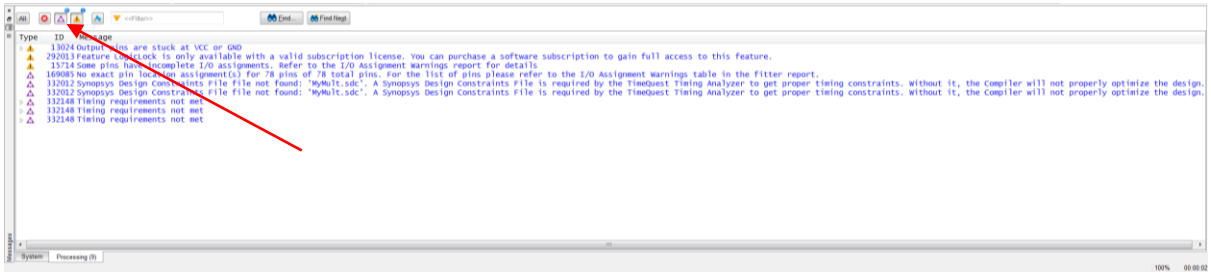
As engineers though, we need to be able to understand the origins of constraints, and perform rough calculations. Reasons for this include the following.

1. Software tools are not (yet) smart enough to re-design a large circuit to make it run faster. Only by us understanding the causes of delays can we end up with "better" circuits.

2. It is very easy to make a small typing mistake that the compiler does not detect; change one variable name into another (can you see the difference between “nO” and “n0” quickly?); miscalculate bit alignments or bit widths; and so forth. By knowing roughly what to expect, we are building in simple checks along the way.

- Assume it takes 20 ns for one “power of four” operation to be performed. What is an upper bound on the input clock frequency?

Go to the Processing console where all the error messages get printed out. (If you are like me, you will be very familiar with looking there for error messages!) Click on the small purple icon to filter the error messages.



You should see Quartus complaining that Timing requirements not met.

In the Table of Contents pane, click on TimeQuest Timing Analyzer and select Clocks. You should see a table with only one row. Your clock is there, “clk”.

- What is the current frequency[1] of “clk” set to?
- Why are timing requirements not met?

Since we didn’t tell Quartus what the frequency of the incoming clock was, it chose one for us! It is also complaining about a Synopsys Design Constraints File not being found, and without one, it claims it cannot get proper timing constraints.

We will fix this another day; for today, it is not an issue because Quartus realised we have a clock coming into the circuit, and so it is still able to give us advice on the fastest clock speed the circuit will tolerate. Look again under TimeQuest Timing Analyzer. You will see “Slow 1200mV 85C Model”. Under there, click on Fmax Summary.

- What is the maximum frequency (Fmax) your circuit can operate at?
- How is it possible that this is larger than the frequency you calculated before, assuming it takes 20 ns for each “power of four” operation? (Hint: recall Lab 3.)

Now look at the Fmax for the “Slow 1200mV 0C Model”.

- If you put your FPGA in the refrigerator (at 0 degrees Celcius), how much faster could it run than if you left it in the desert (at 85 degrees Celcius)?

[1] It is worth emphasising that there is no clock in the FPGA. Quartus will not build on the FPGA a clock of the specified frequency! What we should have done, is tell Quartus what frequency to expect on the input pin that will be connecting to the “clk” wire. It is up to us to build an oscillator on our PCB board and connect it up to a suitable pin on the FPGA to serve as the clock for our design. (Although the fastest clock on the DE1 board is 50MHz, there are PLLs on the FPGA that can be used to increase the frequency.)

Part 4

The faster the better! We want to be able to process the input stream as fast as possible. We need to make a compromise though: we will increase the throughput but accept an increase in latency. What does this mean?

Recall how an assembly line works. We want to assemble a hamburger. We can have a conveyor belt with workers spaced evenly alongside it. Every second, a bun is placed on one end of the conveyor belt. The first worker cuts open the bun. That's his job. Every second, cut open a bun. The second worker adds a slice of cheese to the bun. That's his job. Every second, add cheese. The third worker adds tomato. The fourth some lettuce. You get the picture. A hamburger cannot be made in one second, but each worker can contribute to its assembly within one second.

The throughput is the number of hamburgers that enter and exit the system every second. (There is a serious problem if the number of hamburgers entering is different from the number exiting. It could mean a hungry worker in this case, but generally, the number out must equal the number in because the hamburgers are evenly spaced on the conveyor belt and are all moving at the same speed.) In this case, the throughput is one hamburger per second.

The latency is how long it takes for a particular hamburger to be made, from start to finish. If there are 30 workers, and a hamburger is in front of each worker for 1 second, then it will take 30 seconds from when a bun goes in to when it comes out the other side as a hamburger.

You can imagine that the latency and throughput can be traded off. By reducing the number of workers to 2, it might be possible for each one to spend only 14 seconds doing his part. They have more complicated roles now: they have to add many more things each, but maybe some operations like adding cheese could have been done in less than a second and hence it is more efficient to combine smaller operations into one bigger operation. In this case, the latency becomes 28 seconds; we have saved two seconds. But the throughput is now terrible: rather than one hamburger per second, it has plummeted to one hamburger every 14 seconds, or 0.071 hamburgers per second.

Here is a simple way of pipelining the "power of four" operation.

```
module Pipeline(input [15:0] x, input clk, output reg [63:0] xp4);
        reg [15:0] data_in;
        wire [63:0] data_out;

        reg [31:0] xp2;

        always @(posedge clk) begin
                data_in <= x;
                xp2 <= data_in * data_in;
                xp4 <= xp2*xp2;
        end
endmodule
```

- If we run our FPGA in the refrigerator ("Slow 1200mV 0C Model"), what clock rate can we safely use?
    - Is this significantly faster than in Part 4?

- Have a look at the RTL view and, if necessary, simulate the design, so you can understand how it works and how the speed-up was achieved. (These concepts are important for this course; hopefully being able to play around with them in Quartus will make them more easily understood and remembered than if you were simply presented with theory.)

Part 5

We won't do this, but in principle we could go faster still. We could have several "workers", i.e., circuits that take x as input and compute x^4 at the output. The first number that arrived at the input we would send to Worker 1. The next number we would send to Worker 2, and the third to Worker 3. By this time, Worker 1 will have finished, so we send the output of Worker 1 to the output pins, and give Worker 1 the next number as input. By now, Worker 2 will have finished, so we send her output to the output pins, and give her the next number as input. And so forth.

Doing the above means we need to use more logic elements (and DSP cores) on the FPGA. And there is a limit on the number of logic elements available. And since higher density FPGAs cost more, we should not use logic elements recklessly. Indeed, we have changed our perspective on life and have decided that faster is not necessarily better. What we want now, is to be as small as possible.

To illustrate this more conveniently, we will change the function slightly. Rather than compute the power of four, we will compute the power of eight *modulo* 2^{16}. The modulo operation means that we simply discard the higher-order bits every time we do a multiplication, so each time, we are left with a 16-bit number.

Here is the "original" version.

```
module MyMult(input [15:0] x, input clk, output reg [15:0] xp8);
        reg [15:0] data_in;
        always @(posedge clk) begin
                data_in <= x;
                xp8 <= data_in ** 8;
        end
endmodule
```

- Look at the RTL to confirm it creates the design we want.
- How many resources does it use? (In the Table of Contents, go to Fitter > Summary and look at Total logic elements and Embedded Multiplier 9-bit elements.)

Here is the "small" version. Note that the input is only sampled once every three clock cycles; the throughput is one-third of what the clock rate suggests.

```
module Pipeline(input [15:0] x, input clk3, output reg [15:0] xp8);
        reg [1:0] state = 0;

        reg [15:0] data_in;
        wire [15:0] data_out;
        SingleMult worker(.x(data_in), .xsq(data_out));

        always @(posedge clk3)
                case (state)
                        default:        state <= state+1'b1;
                        2'd2:           state <= 0;
```

```
                endcase

        always @(posedge clk3)
                case (state)
                        2'd0:   begin
                                        data_in <= x;
                                        xp8 <= data_out;
                                end
                        default:        data_in <= data_out;
                endcase
endmodule

module SingleMult(input [15:0] x, output [15:0] xsq);
        assign xsq = x*x;
endmodule
```

- How many resources does it use? (Compared with the original version, we have substantially reduced the number of multiplications for a relatively small increase in the total number of logic elements. To emphasise the point, each multiplication would have taken about 73 logic elements to implement, had there not been dedicated Embedded Multipliers present[2].)
- Copy the RTL into your textbook. Try to work out how it works.
- To help you determine if your deduction is correct, perform the following **functional** simulation. (A timing simulation might add to the confusion right now.)
    - Open the Waveform Editor in Quartus
    - Add the nodes: clk3, state, x, data_in, xp4.
    - For all but clk3, change the radix to Unsigned Decimal.
    - Set clk3 to be a 5 ns clock (that is 200 MHz!!! We are flying!!!)
    - Set x to be a Count Value: start value = 1; increment by = 1; count type = binary; count every 15 ns.
    - Set the end time to be 100 ns.
    - Observe that the input data x changes on the *falling* edge of clk3. It will be sampled on the next *rising* edge of clk3, thereby ensuring that a clean version of it is stored in the flip-flop.
    - Copy the results of the functional simulation into your lab book.
    - Write a detailed description (so you can read it again when you come to revise for the exam) of how the design works.

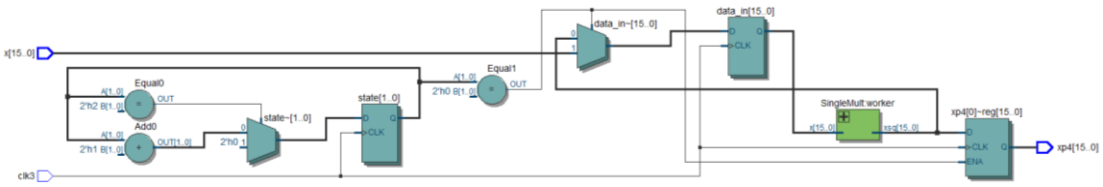
Remark

There is an invaluable lesson to be learnt here, about inferred latches. If the second “always” statement is changed from “default: data_in <= data_out” to “1,2: data_in <= data_out”, then in theory the operation should be identical because “state” can only take on the values 0, 1 or 2. What if the FPGA gets hit by ionising radiation and a bit gets corrupted though? The compiler may rightfully decide that it should implement exactly what you asked it to do: if the state is anything other than 0, 1 or 2, then data_in must not change. The only way for this to happen is to feed

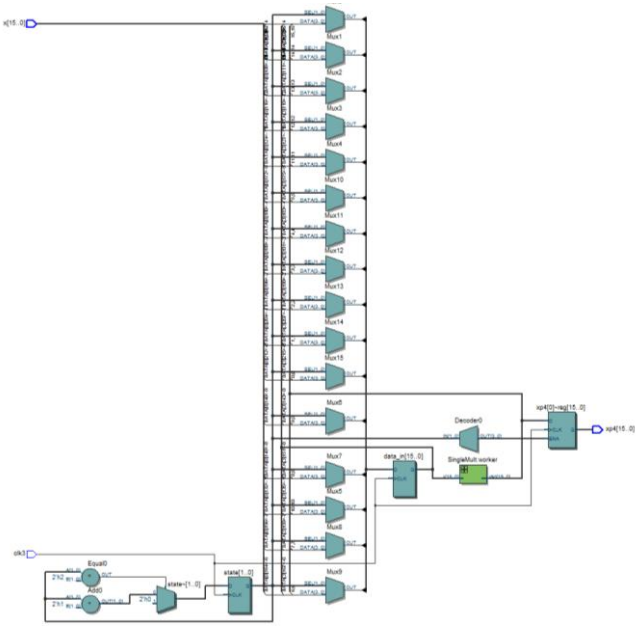
[2] If you are not convinced, you can go to Assignments > Settings > Compiler Settings > Advanced Settings (Synthesis)… and set the Maximum DSP Block Usage to 0, then recompile your design. Note though that the Fmax will be substantially lower if you do this: the DSP blocks can multiply faster than the logic elements!

data_in back through the multiplexer if anything other than 0, 1 or 2 is in “state”. Rather than use a 2-input multiplexer, it has to use a 3-input multiplexer, and this is a big deal because each input is 16 bits wide! Go ahead and make this “small” change; you will see that suddenly the RTL becomes more complicated. (Always check your designs by looking at the RTL.) In the RTL, identify the feedback loop allowing data_in to remain the same. Even if the overall behaviour will be the same, we might have used more logic elements than necessary. (In other cases, an inferred latch may mean we have done something wrong and the design will not even perform as intended. In this particular case, subsequent states of the synthesis process recognise the structure as a finite state machine, and Quartus’ default settings allow it to make optimisations; we would have gotten away with it this time, but it is not good practice to code something different from what you were expecting, even if it still works in the end.)

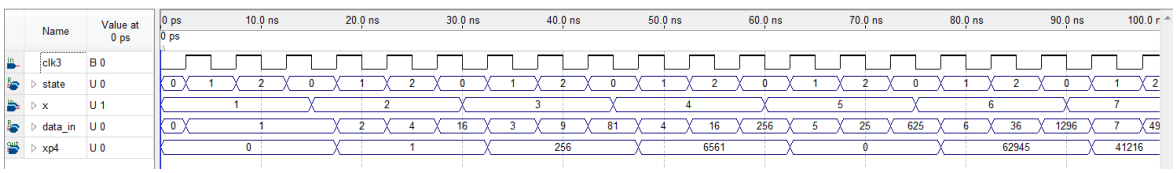
For reference after you have completed the lab, here is the correct RTL.



Here is the RTL with the inferred latch.

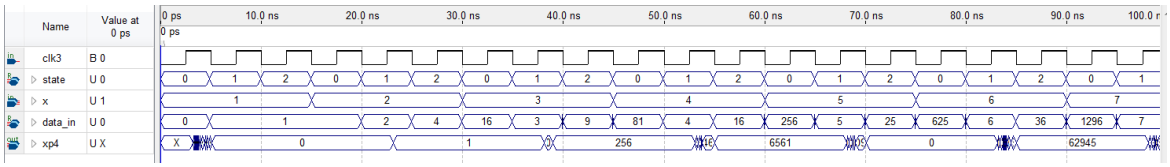


Here is what your functional simulation should look like. (One reason for running the program yourself is so you can read the functional simulation; the following is a bit small.)



The timing simulation below shows glitches on the output. These are caused primarily by the 16 wires carrying the data from the output of the flip-flop to the physical pins on the FPGA being of

different lengths. (You might guess that perhaps the clock arrives at slightly different times at the inputs of the sixteen flip-flops used to buffer the output. Differences in clock arrival times are known as clock skew, and FPGAs are designed with all sorts of tricks in them to try to keep clock skew to a minimum. Clocks can travel along special paths in the FPGA: our clock in this design will automatically be using a global clock pathway specially designed for travelling around the FPGA at high speeds.)



Marking

You will be required to show the demonstrator your lab book and answer questions.