

ELE00011H Digital Engineering
ELE00121M Digital Engineering for MSc

Laboratories: Session 3

Design for performance – Part 2

UG Students: - ALL LABS SHOULD BE DONE IN GROUPS OF TWO STUDENTS
- A mark penalty will apply to single-student submissions unless agreed in advance
- Any issues related to groups (conflicts) should be communicated as soon as possible

MSc Students: - ALL LABS SHOULD BE INDIVIDUAL SUBMISSIONS

Report formatting:

There is no formal report structure – you will be marked on the items listed within each lab script. Most reports will include code printout and simulation screenshots – see Lab 1 appendices for guidelines.

The reports for labs 1 - 4 must be handed in, together in a single zip archive, via the VLE by the deadline indicated on the front page of the script. A single submission should be handed in for each group (by any member of the group).

Each lab report, containing all the material required in the order specified, should be submitted as a **separate PDF file**. The exam numbers of all members of the group should be printed on the front page of each PDF.

The PDF files must be named **Yxxxxxxx-Yxxxxxxx_DE_Lab#.pdf**, where the Yxxxxxxx are the exam numbers of the group members (normally two for UG students, one for MSc students) and # is the lab number.

In all cases, read carefully the instructions on the VLE submission page. Failure to follow the instructions could lead to your assignment not being marked and in any case to a mark penalty.

Submission weight on module mark: 12.5% (25 marks)

Task 1: Pipelining [15 marks]

The circuit you implemented at the end of the previous lab should be the following:

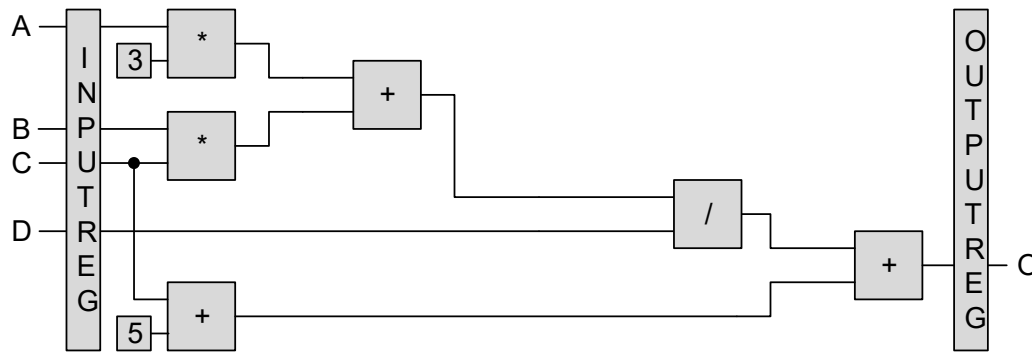


Figure 1.1

Create a new project, importing the files from the previous, and make sure to change the Synthesis setting “-max_dsp” to 0. Set the clock period of your VHDL test bench to 80ns and run the behavioural simulation.

2.1.1: Print out a screenshot of the simulation window, zoomed in to display, in readable format, all inputs, internal data busses, and the output in unsigned decimal format for the same testbench used in lab 1. Include the console output (as a separate screenshot).

Set a **80ns** constraint on the clock in the XDC file (remember to change the fall time so that you maintain a 50% duty cycle). Implement the design.

At this stage, the timing constraint *might* have been met or maybe not. Your objective is to lower or raise the target clock period (in steps of 1ns) until you find the largest value for which the constraint is not met (you can use the slack value to “skip” steps – i.e. if you see a slack of more than 3ns, you can “jump” a couple of ns). This value illustrates the very best that the tools are able to do, given the current design.

IMPORTANT: in this laboratory, you will be asked to do repeated implementation runs in several occasions. Particularly in the later stages, this is likely to require considerable time (especially if you run the project from the M drive). To avoid this, and massively reduce the time required for each run, follow this procedure:

- Click on “Settings” under “Project Manager” in the Flow Navigator pane.
- Select “Implementation” on the left-hand side.
- In the options, scroll down to “Place Design” and select “Quick” in the “-directive” pull-down menu.
- Scroll further to “Route Design” and select “Quick” in the “-directive” pull-down menu.
- Answer “No” if prompted to preserve previous runs.

Remember that this procedure will generate circuits of lesser quality with respect to what you would obtain otherwise. It is simply a “trick” to shorten the time required for you to complete the lab and should be avoided in any sort of “real world” setting!

Note also that this option will introduce a much greater variability in the timing results. Throughout this script, this implies that some of the suggested timings might not match your actual results. Do not worry overmuch (if something looks really odd, ask a demonstrator!) and simply report the values you obtain.

2.1.2: Write the best period where the constraints are met (i.e. the one just before it starts to fail) and print out a screenshot of the “Design Runs” tab, showing all columns up to “DSP” (you might need to click on “Open implemented design” to see the “Design runs” tab).

The objective of this task is to further accelerate the algorithm by introducing pipeline stages. The stages will be implemented as banks of D-type registers with synchronous reset. **The most effective way to implement the registers in VHDL is to introduce additional processes similar to the input and output registers already present in the *algorithm.vhd* entity** (an alternative would be to use a parameterizable register component, but this implies a lot more typing). Remember that the registers needed to pipeline the D vector (the divisor in the division operator) cannot be set to 0 without causing a simulation error.

It is highly recommended to create a separate project for each of the following steps, importing the files from the previous step. Remember to change the Synthesis and Implementation settings!

To start the creation of a pipeline for the circuit, insert a pipeline stage as shown below:

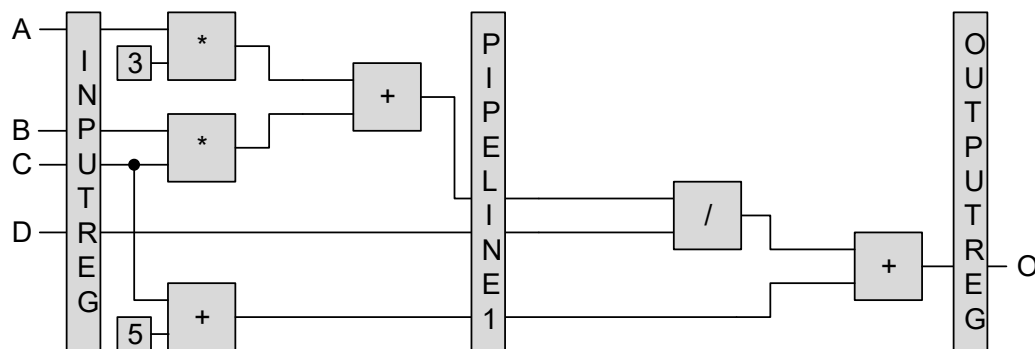


Figure 1.2

This is the most obvious place for a pipeline stage, as it separates the multipliers and the divider, the most complex operators in the design. Make sure that your pipeline includes all signals that traverse the cutset.

Of course, you need to verify that you implemented the pipeline stage correctly. Re-run the behavioural simulation using the same test-bench as above and confirm that the outputs correspond to the expected values. Note that there will be one additional clock cycle delay between the moment your inputs arrive and the corresponding outputs are produced. If you used (as suggested) a constant for the delay between inputs and outputs, you should only need to modify a single value.

2.1.3: Print out a screenshot of the simulation window, zoomed in to display, in readable format, all inputs, internal data busses, and the output in unsigned decimal format for the same sets of input (and output) values used in the preceding simulations. Include the console output (as a separate screenshot).

Set the clock period to the largest value that failed in the attempts above and run the implementation again. Go again through the process of finding the largest integer value for the target clock period for which the constraint is not met (note that again you can use the Timing Report output to speed up the process by jumping to known valid results).

2.1.4: Write the best period where the constraints are met (i.e. the one just before it starts to fail) and print out a screenshot of the “Design Runs” tab, showing all columns up to “DSP”. Based on the lecture material, what are you expecting to happen to the maximum frequency, with respect to the circuit without this pipeline stage? Why? Does the practice match the theory? Comment and explain.

By introducing the first pipeline stage, you should have seen an improvement in the performance, but maybe not as much as you might have hoped. Let us try to improve this.

Insert a second pipeline stage as shown below.

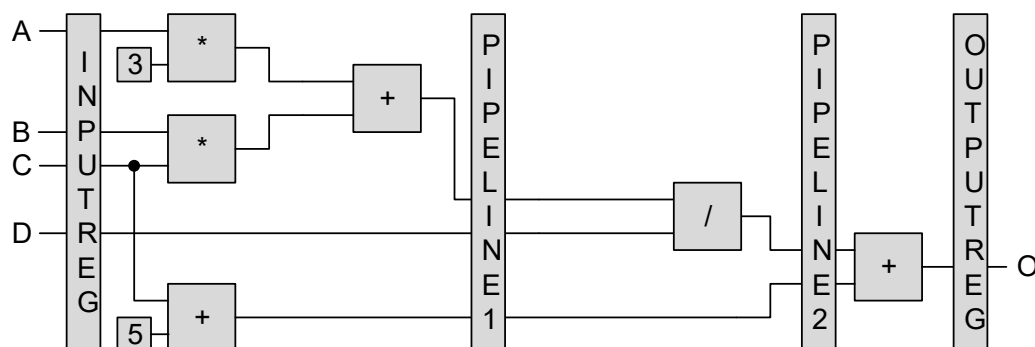


Figure 1.3

Again, considering that the division is obviously the slowest operation, it makes sense to “bracket” it inside a single pipeline stage. Repeat the procedure above for the new circuit (verify its operation and determine its performance).

2.1.5: Print out a screenshot of the simulation window, zoomed in to display, in readable format, all inputs, internal data busses, and the output in unsigned decimal format for the same sets of input (and output) values used in the preceding simulations. Include the console output (as a separate screenshot).

2.1.6: Write the best period where the constraints are met (i.e. the one just before it starts to fail) and print out a screenshot of the “Design Runs” tab, showing all columns up to “DSP”. Based on the lecture material, what are you expecting to happen to the maximum frequency, with respect to the circuit without this pipeline stage? Why? Does the practice match the theory? Comment and explain.

To experiment further with pipelines, try adding a third pipeline stage as below:

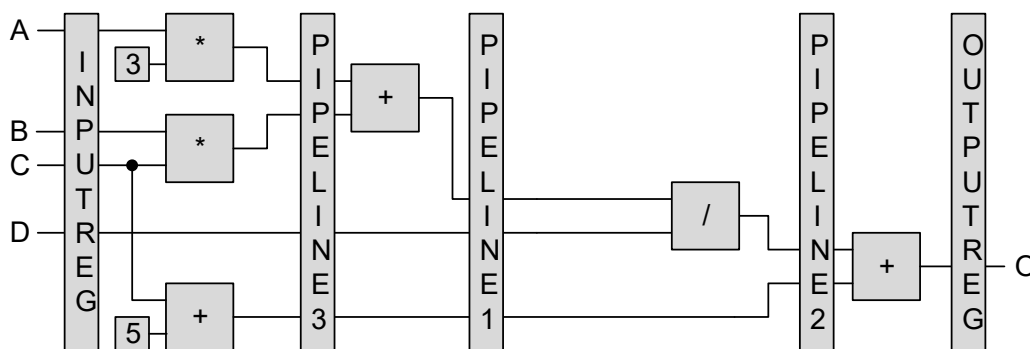


Figure 1.4

Repeat the procedure above for the new circuit (verify its operation and determine its performance). Note the warning messages during synthesis. Can you figure out what they mean?

2.1.7: Print out a screenshot of the simulation window, zoomed in to display, in readable format, all inputs, internal data busses, and the output in unsigned decimal format for the same sets of input (and output) values used in the preceding simulations. Include the console output (as a separate screenshot).

2.1.8: Write the best period where the constraints are met (i.e. the one just before it starts to fail) and print out a screenshot of the “Design Runs” tab, showing all columns up to “DSP”. Based on the lecture material, what are you expecting to happen to the maximum frequency, with respect to the circuit without this pipeline stage? Why? Does the practice match the theory? Comment and explain.

2.1.9: Print out the commented VHDL code at this step. If you have used a parameterizable register for your pipeline stages, make sure to include the code. Note that if any other components have been created (there should be no need for any), they should also be included.

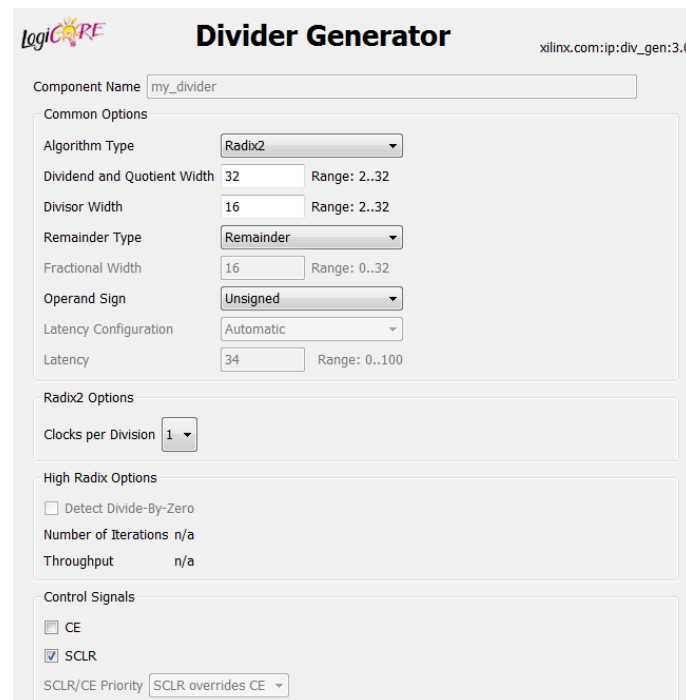
Task 2: IP Components [10 marks]

If we want to substantially improve the performance of our circuit, we need to look at ways to improve the speed of the single operations. The problem here is that the slowest component in our design is the division (we have seen with the introduction of the third pipeline stage that it is useless to improve other parts of the design until the divider is accelerated), and Xilinx does not currently provide a divider IP that you just plug into your circuit.

However, a “locked” version of a divider IP is available on the website. Download the zip file and save the directory somewhere on your computer. Next, go to “Add Sources”, select “Add Design Sources”, then “Add Directory”. Select the directory you have just imported and add it to your project.

Remove the last pipeline stage from your design (in other words, go back to the design illustrated by figure 1.3).

For your information, the IP interface used to generate the divider was the following:



The screenshot shows the 'Divider Generator' IP configuration window. The 'Component Name' is 'my_divider'. Under 'Common Options', 'Algorithm Type' is 'Radix2', 'Dividend and Quotient Width' is 32, 'Divisor Width' is 16, 'Remainder Type' is 'Remainder', 'Fractional Width' is 16, 'Operand Sign' is 'Unsigned', 'Latency Configuration' is 'Automatic', and 'Latency' is 34. Under 'Radix2 Options', 'Clocks per Division' is 1. Under 'High Radix Options', 'Detect Divide-By-Zero' is unchecked, 'Number of Iterations' is n/a, and 'Throughput' is n/a. Under 'Control Signals', 'CE' is unchecked and 'SCLR' is checked. 'SCLR/CE Priority' is set to 'SCLR overrides CE'.

Note in particular the **latency** parameter: this indicates how many internal pipeline stages are present in the implementation of the divider (in this case, 34). This is not a major issue with the operation of the circuit (a bit inconvenient for simulation, but not too much so). On the other hand, the internal pipeline stages will have to be matched by as many external stages to ensure that the data arrives at the adders in the correct order.

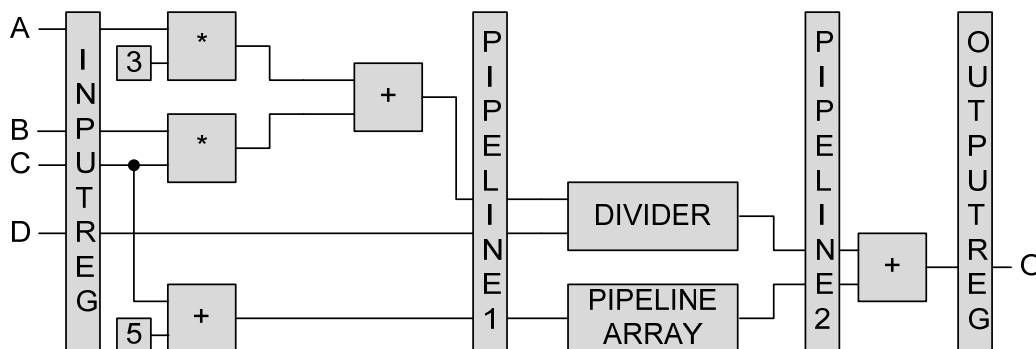


Figure 2.1

This is a large number of registers! Luckily, VHDL provides for-generate loops. Define a for-generate loop of *data_size* shift registers as wide as the pipeline is deep (in this case, 34 bits) to create a deep pipeline to synchronize the data. The depth of this *pipeline array* depends on the latency of the divider, and should be defined as a *constant* in your code.

Add the divider as a component in your VHDL design. To do so, you will need to add two pieces of code. First, a *component declaration*, which goes together with the internal signal declarations (between “architecture” and “begin”):

```
component divider
  port (
    clk: in std_logic;
    sclr: in std_logic;
    rfd: out std_logic;
    dividend: in std_logic_vector(31 downto 0);
    divisor: in std_logic_vector(15 downto 0);
    quotient: out std_logic_vector(31 downto 0);
    fractional: out std_logic_vector(15 downto 0)
  );
end component;
```

Then, a *component instantiation* (where you will have to replace “pipe1_3”, resp. “pipe1_D”, with your own labels for the 32-bit dividend, resp. 16-bit divisor).

```
my_divider : divider
  port map (
    clk => clk,
    sclr => rst,
    dividend => std_logic_vector(pipe1_3),
    divisor => std_logic_vector(pipe1_D),
    quotient => quotient
  );
```

Once the complete circuit has been designed and verified with behavioural simulation (it should give the same outputs as the previous versions, only *much* later). Note that the divider will probably output one or more illegal values before the correct latency has been achieved. This is normal.

Note that, at some point, your timing reports might start mentioning failures in WHS (Worst Hold Slack) and/or THS (Total Hold Slack) somewhere in the divider. These seem to be caused by the use of the “quick” options in the Implementation settings and can be safely ignored for the purposes of this lab.

Run the implementation and have a look at the WNS! Repeat the procedure above to find the first integer period where the implementation fails. You might want to start around the 10ns mark...

2.2.1: Print out a screenshot of the simulation window, zoomed in to display, in readable format, all inputs and the output in unsigned decimal format for the same sets of input (and output) values used in the preceding simulations. You will need two separate screenshots because of the depth of the pipeline (do not try to fit inputs and outputs in one screenshot!) Include the console output (as a separate screenshot).

2.2.2: Write the best period where the constraints are met (i.e. the one just before it starts to fail) and print out a screenshot of the “Design Runs” tab, showing all columns up to “DSP” (note the massive increase in FFs!)

Can the performance of the design be further improved? It depends on where the critical path lies. It would be quite difficult to optimize the divider further. On the other hand, if the critical path should lie elsewhere, it might be possible to further optimize the design.

Select the “Reports” tab in the console and open the “impl_1_route_report_timing_summary_0” for the implementation run that failed to meet the timing constraint. The first part contains some information about your design (including the WNS), but eventually (it *is* a long document) you will see a section called “Timing Details”.

Note that, at this point, there will be differences between each implementation, so a precise guide is not possible. However, generally the structure of the file should be consistent.

First, you should see a report that looks something like this:

```
-----
From Clock:  clk
To Clock:    clk

Setup :          2 Failing Endpoints, Worst Slack      -0.064ns, Total Violation      -0.069ns
Hold  :          0 Failing Endpoints, Worst Slack       0.004ns, Total Violation       0.000ns
PW   :          0 Failing Endpoints, Worst Slack       4.020ns, Total Violation       0.000ns
-----
```

This states that there are two paths (you might well have more) where the clock setup time is violated.

2.2.3: In your own words, explain what a setup violation is and how it relates to the critical path.

Next, there will be a list of the detailed paths that violate the timing constrain. The description will be quite long, as it will list all the gates (LUTs) and routing logic that are in the path, showing the time required for each element – Incr(ns) – and the cumulative time – Path(ns) – for the entire path. In some cases, it might be useful to examine this in detail, but often it is sufficient to look at the initial summary, which should look something like this:

```
-----
Slack (VIOLATED) :      -0.064ns  (required time - arrival time)
Source:              INTC_reg[5]__0/C
                      (rising edge-triggered cell FDRE clocked by clk {rise@0.000
Destination:         pipel_3_reg[31]/D
                      (rising edge-triggered cell FDRE clocked by clk {rise@0.000
Path Group:           clk
Path Type:            Setup (Max at Slow Process Corner)
Requirement:          10.000ns  (clk rise@10.000ns - clk rise@0.000ns)
Data Path Delay:      10.027ns  (logic 5.105ns (50.913%)  route 4.922ns (49.087%))
Logic Levels:         15  (CARRY4=9 LUT3=1 LUT4=2 LUT5=1 LUT6=2)
-----
```

What this tells you is that the path from INT_reg[5] to pipe1_3_reg[31] violates the timing constraint by 0.064ns. The requirement was 10ns but the delay of the data path, which goes through 15 levels of logic, is 10.027ns (to which a bit of clock-related delay must be added to reach 10.064ns).

The most important information is the source/destination pair, which tells me (when I relate it to the signal names in my VHDL) that the critical path is now in the multiply-add sequence, as is the second critical path that violates the constraint: the divider is now no longer the limiting factor in the performance of the circuit!

This also means that maybe there is scope for improvement, as we have not looked at the multiplier yet. Or have we? Remember the third pipeline stage that was useless earlier on?

Improve the design by re-introducing the third pipeline stage. Repeat the procedure used above to find the new period where the implementation fails. You might, or you might not, find an improvement: the adder is a very fast component and removing it from the path might not have a huge effect (in the same way as the stage 2 pipeline did not greatly improve performance by removing the adder from the divider).

2.2.4: Print out a screenshot of the simulation window, zoomed in to display, in readable format, all inputs and the output in unsigned decimal format for the same sets of input (and output) values used in the preceding simulations. Again, you will need two separate screenshots because of the pipeline. Include the console output (as a separate screenshot).

2.2.5: Write the best period where the constraints are met (i.e. the one just before it starts to fail) and print out a screenshot of the “Design Runs” tab, showing all columns up to “DSP”

What does the post route timing report say about the critical path now (you might have to reload the design – see the yellow line at the top of the window – check that the slack time matches your new value)? Probably, the timing will be violated by the multiplier. Can we improve it yet more? One way to do it would be to pipeline the multiplier, in the same way as we pipelined the divider, but remember that the Xilinx devices contain very fast embedded multipliers, so there might be an even better (and simpler) way!

Open again the settings of the Project Manager. Under “Synthesis”, find again the “-max_dsp” entry, and change it back to “-1”. Now, the tools will decide whether to use the fast multipliers or standard LUTs for the multiplier (and it’s very likely that they will choose the first option).

Repeat the “standard” procedure to find the first failing period. Looking at the WNS, you might be able to “skip” a few values. If all has gone according to plan, you should now discover that the tools are able to find an implementation that meets a constraint of approximately 4 or 5ns (at some point, you might find that WNS becomes N/A... more on that below – just stop at that point)! In other words, you have accelerated performance by a factor of almost 20 from the first implementation!

2.2.6: Write the best period where the constraints are met (i.e. the one just before it starts to fail) and print out a screenshot of the “Design Runs” tab, showing all columns up to “DSP”. What is the highest frequency at which your design should be able to run according to the WNS results?

Is that actually true? Have you designed a circuit that runs at more than 200MHz? Well, yes and no. The *core* of the design will indeed run at that frequency (you are welcome to try the post-implementation timing simulation). However, we have obviously cheated, as we have not really considered how data will arrive and leave from the circuit. In high performance computing, getting the data in and out of the computational core is often more challenging than the core itself. In fact, Xilinx knows that its pins (and some of its other internal resources) are not able to run at more than @200MHz and therefore will stop mentioning that the timing constraints are met (hence the N/A in the WNS value). But still, if you find a way (and there are indeed methods to improve that, as we will see in the lectures), then your core will run at over 200MHz and, incidentally, probably outperform any software-based implementation on multi-GHz processors!

2.2.7: Print out the commented VHDL code at this step and the “RTL Component Statistics” and “RTL Hierarchical Component Statistics” part of the synthesis report. Note that there is no need to re-print the parameterizable register, but if any other components have been created (there should be no need for any), they should be included.