RISC-V Dependency Checking and Data Forwarding

Computer Architecture and Design – Practical 2

October 2019

1 Objectives

The aim of this practical is to give you an opportunity to work on some of the most performance-critical code within a 5-stage RISC-V processor pipeline. During this practical you will learn how the dependency logic of an in-order scalar processor can be constructed, and you will gain an appreciation for the benefit of forwarding speculative results within the pipeline. You will get an opportunity to work on timing optimizations, and gain an appreciation for the tradeoffs between instruction latencies and core operating frequencies. You have the option to begin by incorporating your ALU from Practical 1 into this processor, to build on your earlier work.

The first part of this practical involves implementing the logic necessary to detect dependencies between instructions that are currently executing within the pipeline. Initially you will use this logic to *stall* any dependent instructions so they will obtain the correct operand values. You will test and debug your dependency logic by simulating the execution of a test program that contains all possible inter-instruction dependencies, and only when this program runs without error can you declare success at this stage. To help you with this, a test-bench is provided that contains a reference model of a RISC-V processor, against which the results of each instruction on your processor will be compared by the test-bench, cycle by cycle. If your processor commits an incorrect result to its destination register, the test-bench will immediately report an error. The test-bench also computes the dynamic execution rate, measured in cycles-per-instruction (CPI), and prints this to the TCL Console along with the result of each instruction that completes at the Writeback stage.

Having implemented the dependency checking and stall logic, your second task will be to implement the logic to *forward speculative results* from the EXE, MEM and WRB stages to the DEC stage. When each forwarding pathway is implemented, the associated stall condition can be removed, and the CPI of the pipeline will improve (and the processor will become faster and more cycle-efficient). Your goal will be to minimize the CPI while ensuring that the processor remains functionally correct.

When you have completed the forwarding logic, and removed as many stalls as possible, you will replace the test program with a more realistic benchmark application. You will synthesize the processor to run on the FPGA, run the benchmark, and measure the execution time of the benchmark. Initially you will target a default clock frequency of 60 MHz. After the initial benchmark run you will have the opportunity to try synthesizing at higher frequencies, adjusting the Verilog where required (and where possible) to aim for higher frequencies.

This practical requires that you **write a report of your activities and hand it in with your lab sheet, along with listings of your Verilog code at each stage**. For more details on how this practical is assessed, please refer to the marking scheme document that is available on the LEARN page where you obtained this document.

2 Introducing the RV32IM Core

For this practical, a packaged Vivado project containing an almost-complete RISC-V processor has been created for you to use. This contains a top-level testbench that instantiates the processor and runs a selected test program. The RISC-V processor implements an RV32IM configuration, which means it has a 32-bit RISC-V baseline instruction set and also supports the 32-bit integer multiply and divide instructions. The processor essentially implements the micro-architecture explained in lecture notes 10 and 11, on *Pipelined Instruction Execution* and *Pipeline Hazards* respectively. Slides 4–11 in lecture note 11 are of particular relevance to this assignment – study them carefully before attempting this assignment.

3 Getting Started

A zipped project file `prac2.zip` can be found on the course website, on the Practicals page, just below the information about Practical 2. The zipped project contains all of the Verilog source files, libraries, and build scripts needed to create a Vivado project that is capable of simulating and synthesising your RISC-V processor. It also contains some test programs that you can choose to run. Instead of going through the tedious process of creating the project manually, there is a TCL script called `create_project.tcl` that you can run once after unzipping the project. This is done as follows:

1. As always, start by sourcing the Vivado settings script
   `source /disk/scratch/Xilinx/Vivado/2018.3/settings64.sh`
2. Take a copy of `prac2.zip` and unzip it in a suitable location.
3. You will find the unzipped project has a root directory called `prac2`, within which various subdirectories exist.
4. **Important:** copy your `alu.v` file from Practical 1 into the `src/hdl` directory. If your ALU comprises multiple Verilog files copy them all into this location.
5. `cd` to the `prac2/proj` directory.
6. Launch Vivado from the command line by typing `vivado`.
7. At the bottom of the Vivado window select the **TCL Console** tab, and in that console type:
   `source create_project.tcl`

This will take a minute or two and will create a project called `RV32IM_PYNQ_Z2`, after which you will see the Project Manager view of the project. If you go to the Project Manager's Sources window you will see a collection of Verilog files, arranged in two hierarchies; one for the Design Sources, and a separate hierarchy for the Simulation Sources. Select the Verilog Header folder and open the `params.v` file. This contains macros and `localparam` declarations that are used throughout the processor. In particular, note the `HEX_FILE` declaration which is a string that names a hex-coded binary program file located in the Memory Initialization Files folder of the Sources window. This macro specifies the program that will be automatically loaded into the processor's instruction memory on reset, and then run when the processor begins operating. Initially this is set to `deps_test.hex`, which is a directed test program designed for Part (a) of this practical.

If your ALU module is working and ready for use in this processor you should switch out the reference ALU and replace it with your ALU. This is easily done by editing the `ALU_MODULE` macro definition in `params.v`, changing it from `rv32im_alu_0` to the name of your ALU module. If your

ALU module did not work perfectly, then you can leave the reference ALU in place, as this is known to work correctly.

Once you've investigated the overall structure, and optinally selected your own ALU, navigate to the file called `bypass_or_stall.v`, open it up, and move on to Initial Testing.

4 Initial Testing

As a preliminary test, you are recommended to attempt to run a simulation on the processor, after inserting your ALU but before you make any further changes or additions to the Verilog. This will compile the Verilog and simulate the execution of the `deps_test.hex` test program. Compilation should succeed, but the simulation is expected to immediately report a mismatch between the DUT and the reference model. This is because there is initially no dependency checking logic within the DUT and therefore any dependencies will not be honoured. Thus, some stale register operands will be used, and in turn some instructions will prodice incorrect results. The test-bench will detect these errors as soon as they are encountered, it will stop the simulation, and it will output a suitably informative message in the TCL Console. A failure report from the test-bench is a good result at this stage! Now move on to Part (a).

5 Part (a) – Dependency Checking and Stalling

Each instruction typically requires one or two source register operands, **rs1** and **rs2**. These are read from the register file at the Decode (DEC) stage before the instruction moves on to the Execute (EXE) stage. If the destination register of a *downstream* instruction, *i.e.*, at EXE, MEM or WRB, matches one of the source register operands of the DEC-stage instruction, then a dependency exists between that downstream instruction and the DEC-stage instruction. In this case, the value read from the register file is *stale* and cannot be used. However, the register file continuously outputs the current value of each DEC-stage source register, so by *stalling* the DEC-stage instruction until the downstream instruction producing that register has exited the WRB stage, we can be sure that the DEC stage gets the correct operand value. This is the strategy you will initially use to make the processor function correctly in the presence of dependencies.

5.1 Dependency Module Functional Specification

The dependency checking module is implemented as a block of purely combinational logic, *i.e.* without any internal flip-flops. It therefore does not require a clock or reset input. The module header, including the input and output ports, is shown in Figure reffig:header1. You will see that each signal name in the port list comprises a prefix that identifies the pipeline stage producing that signal. The rest of the name is indicative of the purpose of the signal, and in some cases they appear in several pipeline stages. For example, the signal with an `rd` suffix always represents the address of the destination register. Therefore, `exe_rd` is the desination register address for the instruction currently in the EXE stage, and likewise `wrb_rd` is the destination register address for the instruction at the WRB stage. The comments alongside each signal in the port list shown in Figure reffig:header1 document the meanings of each of the pipeline signals you will need to use in this exercise. You should not need to reference any other signals from elsewhere in the processor.

The `bypass_or_stall` module has five interfaces, each preceded by a one-line comment in the module header. The *DEC-stage source operands* interface contains signals that identify each of the two source operands of the DEC-stage instruction. These are the operands we are trying to supply correctly. Each operand has 5-bit register address, a 1-bit read-enable signal, and a 32-bit data value obtained for that register from the register file (note, this value may be stale).

The next three interfaces contain signals that identify the destination operands at the EXE, MEM and WRB stages respectively. As you might expect, these are defined in terms of a 5-bit destination register address, a 1-bit write-enable signal, and a 32-bit result value for that destination register. At the EXE stage we also have signals that indicate whether the EXE-stage instruction is a **load** instruction or some type of CSR read/write operation. Any instruction that is *not* one of these two types will have a valid result by the time they get to the EXE stage. However, a load or CSR operation will not have a valid result at the EXE stage – they obtain a valid result at the MEM stage. Once an instruction has obtained a valid result, that result remains available on the corresponding interface for each subsequent downstream stage.

The final interface contains five outputs from this module. The `dec_stall` signal should be asserted by this module if the DEC-stage instruction should stall in the current cycle, *i.e.*, remain at the DEC stage for another cycle. The two outputs `dec_load_use` and `dec_csr_use` are used for performance monitoring and should be asserted if a stall is due to a dependency on a downstream load instruction or a downstream CSR instruction respectively. The `dec_rs1_data` and `dec_rs2_data` output ports should respectively contain the first and second source register data values. The default implementation of this module drives these outputs from the `dec_rdata1` and `dec_rdata2` inputs, which is correct for part (a) of the practical.

```
module bypass_or_stall(

  //==== DEC-stage source operands  ============================================
  //
  input       [4:0]  dec_rs1,          // first source operand register address
  input              dec_rs1_renb,     // 1 => rs1 is used, 0 => rs1 is unused
  input       [31:0] dec_rdata1,       // X[rs1] from register file
  //
  input       [4:0]  dec_rs2,          // second source operand register address
  input              dec_rs2_renb,     // 1 => rs2 is used, 0 => rs2 is unused
  input       [31:0] dec_rdata2,       // X[rs2] from register file

  //==== EXE-stage instruction information  ====================================
  //
  input       [4:0]  exe_rd,           // EXE destination operand register address
  input              exe_rd_wenb,      // 1 => rd is written, 0 => rd not written
  input       [31:0] exe_result,       // result at EXE stage, destined for X[rd]
  input              exe_load,         // EXE instruction is a Load operation
  input              exe_csr,          // EXE instruction is a CSRRx operation

  //==== MEM-stage instruction information  ====================================
  //
  input       [4:0]  mem_rd,           // MEM destination operand register address
  input              mem_rd_wenb,      // 1 => rd is written, 0 => rd not written
  input       [31:0] mem_result,       // result at MEM stage, destined for X[rd]

  //==== WRB-stage instruction information  ====================================
  //
  input       [4:0]  wrb_rd,           // WRB destination operand register address
  input              wrb_rd_wenb,      // 1 => rd is written, 0 => rd not written
  input       [31:0] wrb_result,       // result at WRB stage, destined for X[rd]

  //==== Outputs to stall DEC stage and provide forwarded results ==============
  //
  output reg         dec_stall,        // 1 => stall DEC, 0 => no stall at DEC
  output reg         dec_load_use,     // DEC stall is due to Load use with MEM
  output reg         dec_csr_use,      // DEC stall is due to CSR use with MEM
  output reg  [31:0] dec_rs1_data,     // R[rs1] value to pass on to EXE stage
  output reg  [31:0] dec_rs2_data      // R[rs2] value to pass on to EXE stage
);
```

Figure 1: Verilog header for the `bypass_or_stall` module

5.2 Implementation and Testing

The task for Part (a) is to implement the `dec_stall`, `dec_load_use` and `dec_csr_use` signals properly within the `bypass_or_stall` module. The `dec_stall` signal should be asserted whenever there is a read-after-write dependency between the DEC-stage instruction and any downstream instruction. Such dependencies exist if there is an enabled downstream destination operand that will write to the same register as an enabled source register operand at the DEC stage.

The `dec_load_use` signal should be asserted if there exists a dependency from the destination of a *load* instruction at the EXE stage and a source operand at the DEC stage. An EXE-stage instruction is a load if the `exe_load` signal is asserted. Likewise, the `dec_csr_use` signal should be aserted if there exists a dependency from the destination of a CSR read operation at the EXE stage and a source operand at the DEC stage. An EXE-stage instruction returns a CSR if the `exe_csr` signal is asserted.

Within the `bypass_or_stall` module you will find assignments to the `dec_stall`, `dec_load_use` and `dec_csr_use` signals; you will need to completely replace these with your own logic for those signals. You can test your implementation of Part (a) by simply running a simulation. You may find it helpful to know that the test program executes various sequences of instructions that systematically cover all possible dependencies that can occur. A listing of the `deps_test` program can be found in `prac2/src/others/deps_test.dump`.

As the simulation runs you will see one line of output per instruction in the TCL console window. From left-to-right, each line shows the cycle at which the instruction completed, the CPI up to that point, and any state update from that instruction. State updates may be writes to destination registers or writes to memory. If your processor writes incorrect values, or writes to incorrect registers, or writes when not expected, the test-bench will report the error as precisely as it can.

The tests in the `deps_test` program start by testing dependencies from EXE to DEC, then move on to MEM to DEC dependencies, and finally dependencies from WRB to DEC. If you wish, you can therefore implement the checks one stage at a time, and then the failure point of successive tests will occur later and later in the run. Eventually your implementation will be complete and the test will reach the end without failure.

5.3 Evaluation and Reporting Baseline CPI Results

When you have implemented Part (a), and the test runs to completion without error, take a copy of your `bypass_or_stall.v` file so that you can include it in your report (you will further modify this file in Part (b)). When the test program runs, some instructions will be forced to stall at the DEC stage due to the stall logic that you created. While this ensures correct operation, it also reduces the throughput of the processor pipeline. If the pipeline operated without any stalls, and never had to be flushes, the average throughput would equal the peak throughput of one instruction per cycle, *i.e.* CPI = 1.0. However, the presence of stalls in your implementation will reduce throughput and therefore increase CPI to a value greater than 1.0. An obvious question to ask at this point is "are there any unecessary stalls?" This can be evaluated by measuring the CPI and reporting it. Therefore, for this part, you are asked to record and report the **cycle count**, the **instruction count**, and the **CPI** that is reported in the TCL console from your final successful test run. You should include these measurements in your write-up for this part of the exercise.

At this point call over a lab demonstrator and get your CPI measurement checked and signed off on your laboratory sheet.

5.4 Baseline Benchmark of a Real Program on the FPGA

You now have a working processor, so the next step is to test it with a more significant benchmark and that requires you to run it on the FPGA. First you must switch from using the test program to using the benchmark program. This is done by editing the `params.v` file and setting the `HEX_FILE` macro to `mbrot.hex`. Once you've saved it, run Synthesis and Implementation on your design, and then generate a bit-stream, in the same way you have done for Practical 1.

5.4.1 Critical Path for Part (a)

Open the Implementation and, under the Implementation menu, select the Timing Summary Report. This will run a timing analysis and create timing reports. The longest paths are listed in a table, from which you can select the first – this will be the **critical path**. **Describe the critical path in your report, and include details of the timing slack**.

5.4.2 Setting up the PYNQ board and running the test

You will need to set up your PYNQ board with additional cables, as illustrated in Figure 2. Under the keyboard of your DICE machine you will find a spare ethernet cable and a spare VGA cable. These should be connected to the PYNQ-Z2 board, for which you will need a HDMI-to-VGA adapter (available from the top drawer of the filing cabinet). Carefully insert the pins of the seven-segment display (SSD) board to the **lower row of pins** on the PMOD port. Turn on the board and open up the Hardware Manager in Vivado, then connect the device and program it, as you've done before. You should see that the SSD now displays a single zero digit, and LEDS 0 and 1 should be illuminated. These indicate that the clock generators for the processor and the frame buffer are both up and running.

Switch the display of your DICE machine to select the VGA input instead of the DICE machine, and you should see a black screen with a green rectangle in the centre. **Call a demonstrator for assistance if you have problems at this stage.**

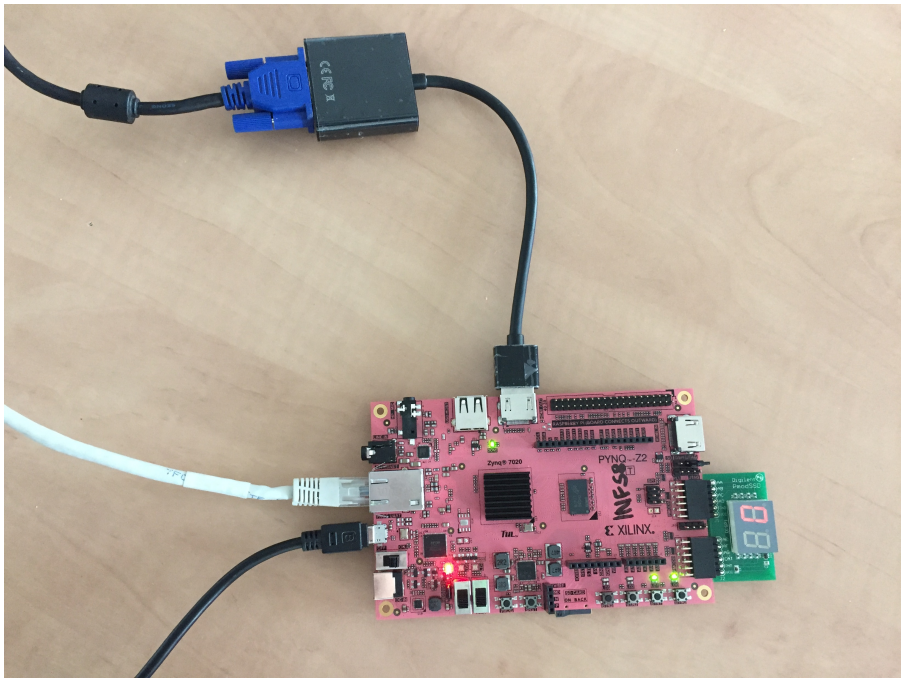

Figure 2: Setting up the PYNQ-Z2 board to run the benchmark test

Once you have a green rectangle on screen you're ready to run the benchmark. You will need to use the buttons on the PYNQ-Z2 board to control the execution of the program. The function of each button and switch is given in Table 1.

Button	Function
0	System reset
1	CPU run (press to start the CPU running)
2	CPU stop (press to suspend the CPU)
3	Benchmark re-run
Switch	**Function**
0	Select the benchmark (OFF = low complexity, ON = high complexity)
1	Timer resolution (OFF = 0.1 second, ON = 0.2 second)

Table 1: Functionality of push buttons and switches on the PYNQ-Z2 board

Ensure both switches are off, and then press button 1 to start the program running. You should see a Mandlebrot set appear in the display area, and this should take a few seconds to complete. When the program is running, the SSD counts the execution time. The units of the timer are 0.1 seconds when switch 1 is OFF, or 0.2 seconds when switch 1 is ON. As the SSD has only 2 digits, if your benchmark takes more than 9.9 seconds to run it will be unable to display execution time when switch 1 is OFF. For this unoptimized processor, the SSD is very likely to overflow when counting at a resolution of 0.1 seconds. If the SSD display overflows it will display two dashes. If you set switch 1 to ON, and then re-run by pressing button 3, the SSD will count fifths of a second, and then the Mandlebrot for Part (a) should not overflow the timer display. If you've reached this point, congratulations you have a working processor that is capable of executing complex programs!

At this point call over a lab demonstrator, show them the running benchmark, and get the demo signed off on your laboratory sheet.

6 Part (b) – Forwarding Speculative Results

There are two objectives in Part (b). Firstly, to implement the forwarding of speculative results from EXE, MEM and WRB to the DEC stage, as required by the DEC-stage instruction. Secondly, to remove all stalls that are not needed when a speculative result can be forwarded to the DEC stage. An error in the design of this part can lead to two types of impact: (i), if a stall is removed in circumstances that cannot be handled by forwarding a downstream result, then the program will fail; (ii), if a stall is not removed in all possible circumstances, then the CPI improvement will be less than it could be. Your aim is therefore to minimize the stalls as much as possible while ensuring correct functionality under all possible types of downstream dependency.

Figure 3 shows a diagram of the RV32IM pipeline in which the pathways needed to forward values from downstream, and hence bypass the register file when obtaining operands, are highlighted in red. These red pathways carry the speculative destination register result from EXE, MEM and WRB to the DEC stage. These values are inputs to the `bypass_or_stall` module and hence you can use them to bypass (*i.e.*, override) the source operand values read from the register file at the DEC stage.

6.1 Implementation and Testing

The first task is to change the `HEX_FILE` macro back to the original `deps_test.hex` test program. This will allow you to run simulations of Part (b) and use the test-bench reference model to detect any bugs in your design. To implement Part (b) you should edit the `bypass_or_stall` module, redefining

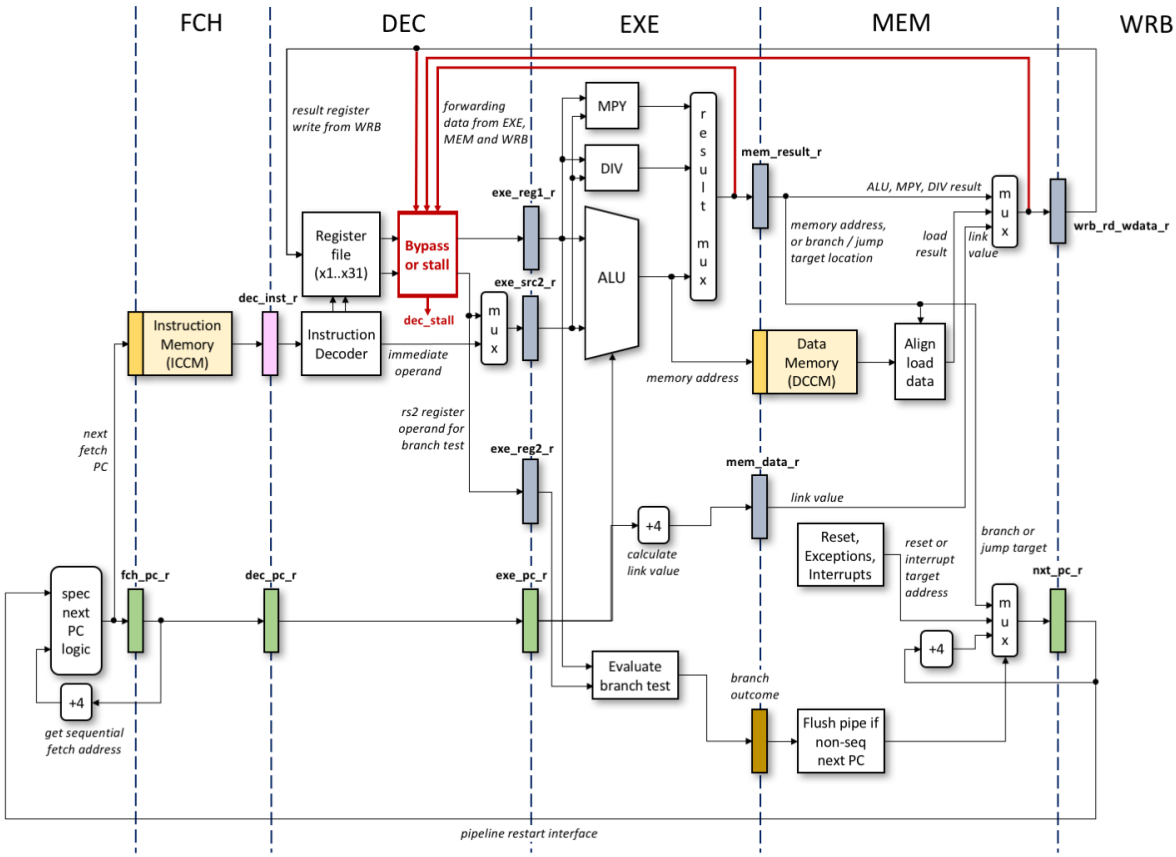

Figure 3: RV32IM pipeline diagram highlighting the forwarding paths

the `dec_rdata1` and `dec_rdata1` outputs so they deliver the most recent downstream speculative value for each operand, or the value read from the register file if there is no downstream dependency. The `dec_stall` signal will need to be modified to prevent a stall when a speculative downstream value can be used.

Run the simulated test program, as in Part (a), until you are satified that all opportunities for forwarding and stall removal have been found, and the test runs to completion without error.

6.2 Evaluation and Reporting Optimized CPI Results

When you have implemented Part (b), and the test runs to completion without error, record and report the **cycle count**, the **instruction count**, and the **CPI** that is reported in the TCL console from your final successful test run. You should include these measurements in your write-up for this part of the exercise. Your report should also address the following questions:

1. Why has the CPI not reduced all the way to 1.0 in this optimized run?
2. Tabulate any remaining stalls that occur in this test
3. If there are pipeline flushes, explain why they are there and estimate their performance impact for this test

At this point call over a lab demonstrator and get your CPI measurement checked and signed off on your laboratory sheet.

6.3 Optimized Benchmark of a Real Program on the FPGA

You now have an *optimized* processor, so the next step is to test it with a more significant benchmark and that requires you to run it again on the FPGA. First you must switch back to using the Mandlebrot program. Recall, this is done by editing the `params.v` file and setting the `HEX_FILE` macro to `mbrot.hex`. Once you've save it, run Synthesis and Implementation on your design, and then generate a bit-stream.

After synthesis you should carefully check the TCL Console output for any sign of synthesis errors, and fix any errors before proceeding. In particular, look out for combinational loops and inferred latches – these are common design bugs that show up during synthesis.

6.3.1 Critical Path for Part (b)

Open the Implementation and, under the Implementation menu, select the Timing Summary Report. This will run a timing analysis and create timing reports. The longest paths are listed in a table, from which you can select the first – this will be the **critical path**. **Describe the critical path in your report, and include details of the timing slack. Compare and contrast the critical path of the optimized design with the critical path of the unoptimized design obtained in Part (a)**.

6.3.2 Running the benchmark on an optimized design in the FPGA

Run the Mandlebrot test on the processor in the FPGA, as you did in Part (a). This time you should find that the benchmark runs noticeably faster, and the final execution time displayed on the SSD should now be within range of a 2-digit display (*i.e.*, it should take less than 10 seconds). **Note the execution time, for inclusion in your report**.

At this point call over a lab demonstrator, show them the running benchmark, and get the demo signed off on your laboratory sheet.

7 Part (c) – Optimizing the Critical Path

The aim of this part of the assignment is to increase the operating frequency of the processor, and thereby reduce the execution time of the benchmark. However, the amount of positive slack on the critical path will limit the extent to which you can reduce the period of the processor clock without violating the setup time on the capture flip-flop of the critical path. Optimizing timing is an iterative process, so you can stop whenever you are happy with the result, or convinced you cannot make any further improvement, or you have run out of time. The steps at each iteration are:

1. Work out from the timing report where the critical path lies. Look at the start and end points, and any intermediate points that you can identify in the detailed timing report for the critical path. **Describe this path in your written report, and explain how it might be reduced**.

2. Adjust the RTL design of the processor in order to implement the timing optimization you identified in the previous step.

3. Using the `deps_test.hex` test program, verify that the processor still functions correctly after your optimization.

4. Re-synthesize the RTL and re-run the Implementation phase.
5. Using the new timing report determing whether your optimization has reduced the critical path.
6. Keep this optimization if it reduces critical path length
7. (optionally) iterate again

When you have optimized the critical path as much as you feel is appropriate, calculate the new maximum possible operating frequency, F_{max}. You can at this point adjust the CPU clock generator to deliver this revised clock frequency. This is done by double-clicking on the `cpu_clock_gen` component in the Sources window. This opens up the *Clocking Wizard* as shown in Figure 4.

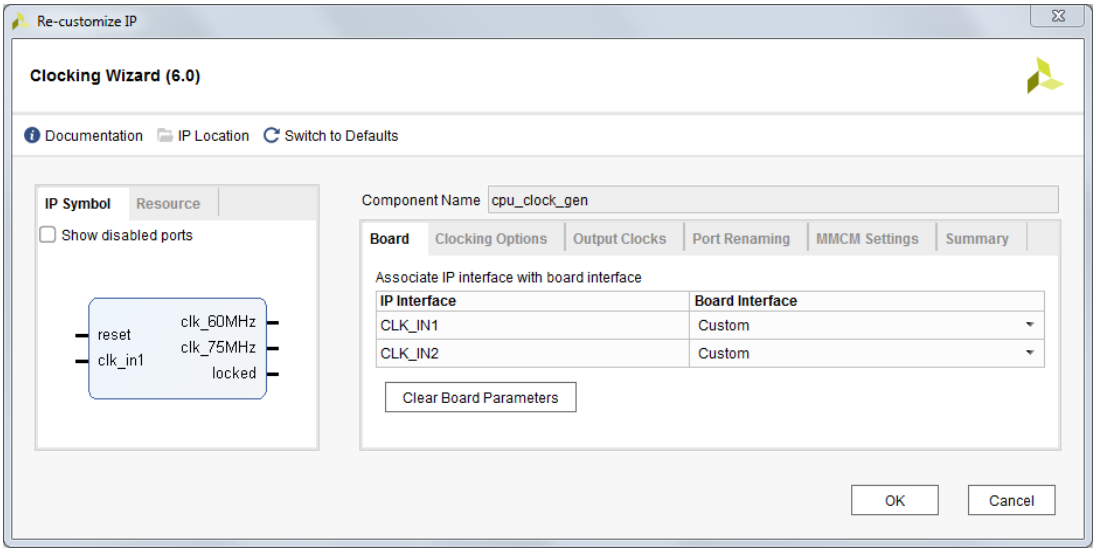

Figure 4: Vivado clocking wizard

Select the *Output Clocks* tab and, under the *Output Freq (MHz)* heading as shown in Figure 5, change the *Requested* value on the `clk_out1` row from 60.000 to your desired frequency (in MHz). Do not alter the 75 MHz clock, as this is used to generate the video clocks for the graphics display.

When you click OK, Vivado will pop up a *Generate Output Products* dialog to offer options for generating the new clock module, as shown in Figure 6. Leave all options at their defaults, and click **Generate**.

Vivado will automatically derive the timing constraints relating to the CPU clock whenever the clock generator is re-configured in this way. This is convenient and means you do not have to manually edit any timing constraints in the `timing.xdc` file.

If you are curious to find out what happens if the critical path is violated, (just for fun) try setting the clock frequency to a value higher than F_{max}. For example, you might set the clock period so there is a timing violation of just a few nanoseconds. This may not necessarily mean a particular program will fail catastrophically, but it may produce incorrect results and this may be visible on your system as errors in the Mandlebrot image.

At this point call over a lab demonstrator and get Part (c) checked and signed off on your laboratory sheet.

Document your timing analysis, RTL optimizations, final F_{max} results, and any relevant observations from Part (c) in your report.

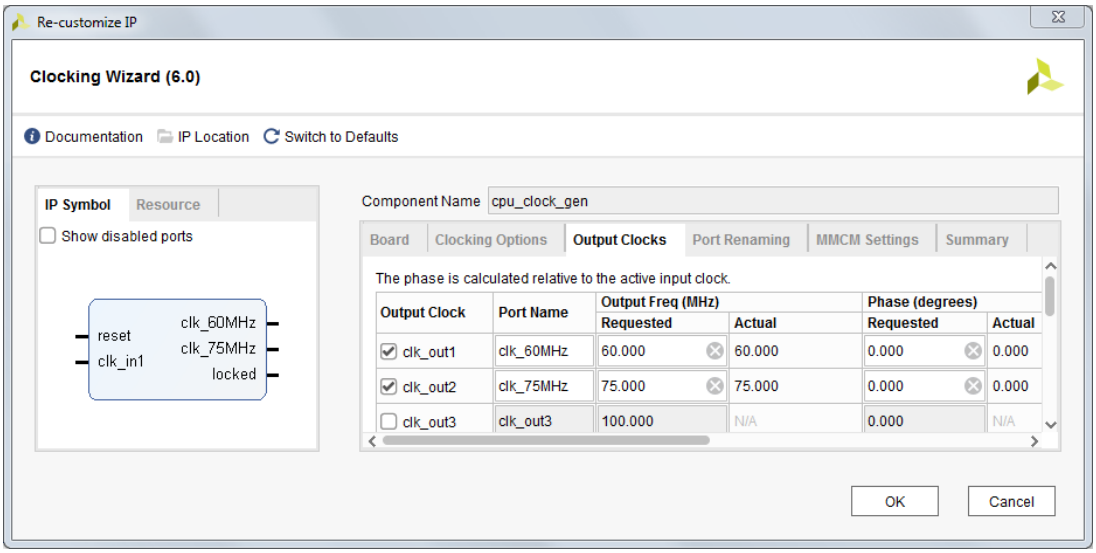

Figure 5: Vivado: output clocks tab

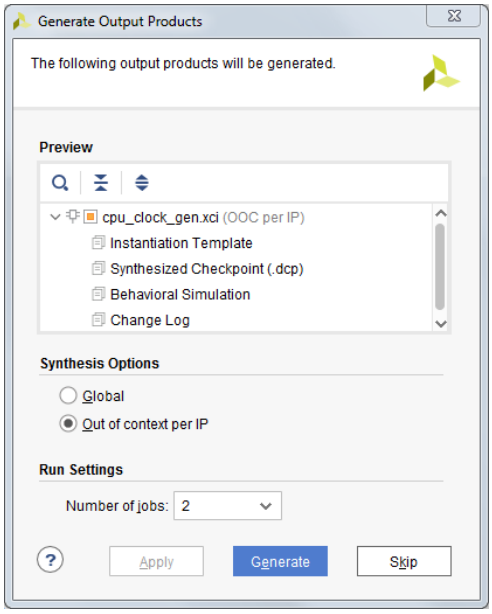

Figure 6: Vivado: generate outputs for configured clock IP block