Lab 7: Processor Optimization

Introduction

In this lab, you will improve the performance of the processor you designed in Lab 5. You will implement the same instruction set and use the same memory interface.

The processor from Lab 5 required many cycles to execute each instruction, so its IPC was less than 1. Its datapath was underutilized – while an instruction was being executed (for example, two registers are being added by your ALU), the other parts of the pipeline were laying dormant and not doing anything that cycle. However, it is possible to utilize more of your datapath every cycle. For example, while an instruction is being executed, you could be simultaneously fetching the next instruction from memory. This is called *pipelining* and it can increase the IPC of your processor.

As an example, Figure 1 shows the cycle-by-cycle execution of instructions in an unpipelined 3-cycle processor on the left (not necessarily the one from Lab 5), versus a pipelined one on the right, with operations belonging to the same instruction highlighted using the same colour. The unpipelined processor achieves an IPC of 0.33 whereas the pipelined one can achieve a steady-state IPC of 1 after an initial start-up period – a three-fold improvement. This is only the ideal case, and in practice the IPC will be lower for certain types and sequences of instructions.

#	Operations	
1	`o_pc_addr = PC`	`PC = PC + 4`
2	`IR = i_pc_rddata`	
3	(execute instruction A)	
4	`o_pc_addr = PC`	`PC = PC + 4`
5	`IR = i_pc_rddata`	
6	(execute instruction B)	

#	Operations			
1	`o_pc_addr=PC`	`PC=PC+4`		
2	`o_pc_addr=PC`	`PC=PC+4`	`IR=i_pc_rddata`	
3	`o_pc_addr=PC`	`PC=PC+4`	`IR=i_pc_rddata`	(execute A)
4	`o_pc_addr=PC`	`PC=PC+4`	`IR=i_pc_rddata`	(execute B)
5	`o_pc_addr=PC`	`PC=PC+4`	`IR=i_pc_rddata`	(execute C)

Figure 1: Example Unpipelined vs. Pipelined Execution

Your goal in this lab will be to pipeline your existing processor from Lab 5. Your modified design will be simulated in ModelSim using a testbench, running a suite of provided *microbenchmark* programs that measure the processor's IPC in different situations using carefully-selected sequences of instructions. **You must ensure that your design is hardware-synthesizable** (see instructions on *Using the `harness` Project with Quartus* below). No hardware will be actually tested on the DE1-SoC board.

In the last, optional part of the lab, you will have a chance to improve your processor's performance in a free-form way and compete with your classmates for bonus marks.

Description

For this part of the lab, your CPU's pipeline will have four stages. It takes one clock cycle for an instruction to move from one stage to the next, and unlike your CPU from Lab 5, multiple stages can be occupied with instructions simultaneously. Figure 2 gives an overview of the CPU's pipeline, showing the names of the stages and the hardware that is accessed (and operations that are performed) during each stage.

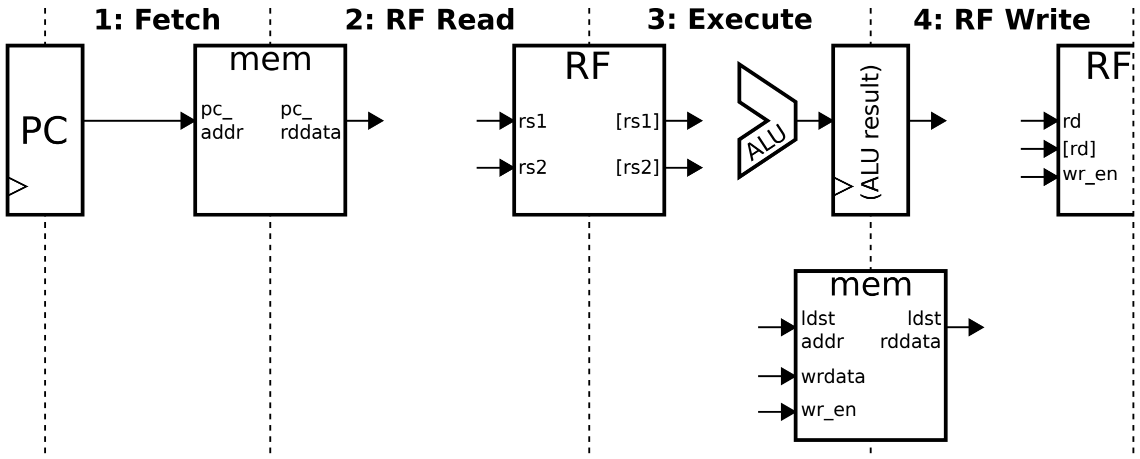


Figure 2: Pipelined processor datapath

Here's a detailed look at what each stage does:

1. **Fetch**: The PC is used to generate a read request to memory and is incremented by 4 (or overridden with a branch target address).
2. **Regfile Read (Decode)**: The 32-bit instruction word returning from memory is decoded into its parts and is used to initiate up to two reads to the register file for register operands `rs1` and `rs2`.
3. **Execute**: The operations specific to each instruction are performed here (everything except writing a result to an RF register). Arithmetic instructions calculate their result using the ALU. Branch and jump instructions may modify the PC. Load and store instructions generate a read or write to memory (simultaneously and independent of the instruction fetch from memory in Stage 1).
4. **Regfile Write (Writeback)**: If the instruction needs to write to the register file, it does so here. Note that the register being written to (called `rd`) can be a *completely different* one than the `rs1` or `rs2` being read at this time by a different instruction. This stage exists separately from Execute because for load instructions, an extra cycle is required to retrieve the data value to write into the RF.

Note that in Figure 2 some components show up twice (the memory and the register file), but are in fact just the same blocks redrawn twice for clarity, to better show their relationships to their pipeline stages. Other pipeline registers that you will need (ex. to retain `rd` until it is needed in stage 4) are not shown. The Fetch and Execute stages need to be able to operate simultaneously – if a load or store instruction is being executed in Stage 3, it must still be possible to fetch an instruction (at a different address) in the *same* clock cycle. The memory interface to your processor in this lab will be **dual-ported**. It's the same memory block, but allows two simultaneous accesses during the same clock cycle.

Table 1 shows the processor's signal interface for this lab, which are the same those you used in Lab 5. Memory Port 1 ('pc') is read-only and is used for fetching instructions, and Port 2 ('ldst') is used for loads and stores and can be read and written. The timing for each port's signals is identical to Lab 5. You can assume that reads and writes will always be accepted by the memory without stalling, and that read data always arrives one cycle after the address, read enable and byte enable are provided.

Signal	Direction	Size	Description
`clk`	input	1	Clock
`reset`	input	1	Active-high reset
`o_pc_addr`	output	32	Address (in bytes)
`o_pc_rd`	output	1	PC Read enable
`i_pc_rddata`	input	32	PC Read data
`o_pc_byte_en`	output	4	PC Byte enable
`o_ldst_addr`	output	32	Address (in bytes)
`o_ldst_rd`	output	1	Load/Store Read enable
`i_ldst_rddata`	input	32	Load/Store Read data
`o_ldst_wr`	output	1	Load/Store Write enable
`o_ldst_wrdata`	output	32	Load/Store Write data
`o_ldst_byte_en`	output	4	Load/Store Byte enable
`o_tb_regs`	output	32 x 32	Register values

Table 1: Processor signal interface

Part I: Basic Pipelined Processor

The implementation of the processor in this lab is more complicated than in Lab 5. It is similarly divided into multiple parts that build on each other. Note: you will not submit a version of your processor for each stage. You will sumbit one version that satisfies all the requirements and optionally a second version for the bonus (see Part V). In this first part, you will create an initial version of the pipelined processor that can correctly execute very simple programs consisting of only arithmetic instructions, `lui` and `auipc`. You can first implement this assuming independent instructions that do not read registers recently written to by an earlier instruction. In Part II you will add bypassing.

Project Setup

Start with your submission for Lab 5. In addition, there is a Quartus project called `harness` which is used to make sure that the Verilog/SystemVerilog you write is synthesizable on an FPGA. We discuss using the `harness` project further below.

Implementation

Write a control and datapath module for this new processor, and instantiate/connect them in the top-level `cpu` module. Use your Lab 5 code as a reference and starting point, especially the datapath. It might be helpful to reorganize your datapath by pipeline stage.

You will also need to add registers into your pipeline to 'remember' certain information from stage to stage. For example, Stage 4 still needs to know the register number being written to, and possibly the type of instruction as well. It must get these from Stage 3, which itself gets a copy from Stage 2 where the instruction word was actually read from memory.

A 'valid' register per-stage[1] will allow you to know if a particular stage is actually occupied or not, enabling/preventing that stage from making changes to registers, flags, or memory. For example, when the CPU is first reset, no instructions have been fetched yet, and Stages 2/3/4 are empty, and a zeroed valid flag for, say, Stage 4, will prevent a register file write during that cycle.

The control module for your CPU will require the most extensive re-write. Previously, you used a state machine, which can only be in one state at a time. This worked fine as a single thread of control for your multi-cycle CPU. For a pipelined CPU, multiple instructions exist in different phases of their execution simultaneously in your datapath. You can use multiple state machines, or ideally, abolish the use of state

[1]You are **not** required to use a valid register if you choose to implement this some other way.

machines altogether and control each pipeline stage independently based solely on the signals produced by the previous stage.

Testing

The testbench does two things: it ensures your processor's output is correct, and it measures its IPC to make sure its performance is high enough. When you run the testbench, it will give a Pass or Fail for both the correctness and performance categories. The goal of Part I is for your processor to be able to pass the first test only (`0_basic`) with an IPC of 1.0. This means your processor should complete one instruction every cycle in the steady state. If functional correctness fails, the testbench will print out the expected vs. observed values of each register.

At the bottom of `tb.sv` in the main `initial` block, you can comment out the other test cases to just test `0_basic` at first. After it passes, compile the `harness` Quartus project to verify that the Verilog/SystemVerilog code is hardware-synthesizable and causes no compilation errors or warnings about latches or combinational loops (see *Using the `harness` Project with Quartus*).

Part II: Dependent Instructions

In this part, you will enhance your processor to be able to handle sequences of dependent instructions. Consider this code:

```
ori  s0, zero, 1
addi s1, s0, 2
sub  s2, s0, s1
add  s0, s1, s1
```

The instructions are fetched sequentially and start moving forward in the pipeline. When the `ori` reaches Stage 4 (RF Write), the `addi` is in Stage 3 (Execute) and the `sub` is in Stage 2 (RF Read). If proper measures are not taken, this code will not execute correctly, because the `addi` and `sub` instructions will use the **old** value of `s0`, which the `ori` in Stage 4 hasn't had a chance to commit to the register file yet.

Stages 2 and 3 must be able to use the value of a register (`s0` in this case) from Stage 4 *before* it has been written to the register file. This value, which is connected to the Register File's write port, must be *forwarded* [2] to Stages 2 and 3. To do this, you need to create hardware that:

1. Recognizes when a register that's being read as an input in Stage 2, or Stage 3, is simultaneously being written to in Stage 4
2. Overrides Stage 2 and/or Stage 3's contents of `[rs1]` and/or `[rs2]` with Stage 4's `[rd]`.

After you add the forwarding logic, you should be able to pass the `1_arithdep` test. Again, verify that the Verilog/SystemVerilog code is hardware-synthesizable.

Part III: Branches

Branches here refer to any of the eight jump or branch instructions that can change the default control flow of the processor: `jal jalr beq bne blt bge bltu bgeu`. They pose a challenge for pipelining because it takes until the Execute stage to determine that an instruction *is* a branch instruction, whether or not that branch is taken, and what value the PC should be set to. Until that information is available, the Fetch and RF Read stages can only *guess* that the next two instructions are located at PC+4 and PC+8 [3]. This leap of faith is required if you have any hope of reaching an IPC of 1.

[2] This is sometimes referred to as bypassing. The two terms are equivalant.

[3] Modern processors use branch prediction to make better guesses. You can assume that instructions are not branches and fetch from PC+4.

For this lab, your Fetch stage can assume that the next PC is PC+4 unless the Execute stage knows for sure that it needed to be something else. This way, your processor can continue fetching 1 instruction per cycle and hoping that this assumption is correct. However, if there is a branch instruction in the Execute stage, and this branch ends up being taken, then that means that the two instructions behind the branch in the pipeline are *not actually the next 2 instructions.* In this scenario, you must make sure these two instructions never make it to the Execute stage. This will create an empty bubble of 2 cycles in your processor until the correct program flow is re-established.

A branch is considered 'taken' if it's either a jump (`jal jalr`), or if it's a conditional branch (**`beq bne blt bge bltu bgeu`**) **and** the condition is true. This changes the PC to something else than PC+4[4]. If the Execute stage contains a taken branch instruction, then at the end of that cycle:

1. The PC will be set to its correct value specified by the branch instruction and will generate a correct fetch *next* cycle. That means that the instruction being fetched (address being read from) next cycle will be the correct instruction to follow the branch.
2. Stage 2 will not be considered occupied/valid.
3. Stage 3 will not be considered occupied/valid.

If you used a 'valid' register for each pipeline stage, as suggested in Part I, then steps 2 and 3 should be straightforward. After modifying your processor to support branches, then you should be able to run `2_branch_nottaken` and `3_branch_taken`. Taken branches will degrade the IPC of the processor, and this is expected. The `3_branch_taken` test has a minimum expected IPC of 0.33 to reflect this.

Again, verify that the Verilog/SystemVerilog code is hardware-synthesizable.

Part IV: Loads/Stores

Finally, you will add support for load and store instructions. Make sure that you include forwarding logic for load and stores (and for jumps and branches).

You should now be able to run `4_memdep`. Again, verify that the Verilog/SystemVerilog code is hardware-synthesizable.

Part V: Competition (BONUS)

The bonus is **optional**. We recommend successfully completing parts I to IV before trying the bonus.

For the bonus, modify the architecture of your pipelined processor to improve its performance even further, and compete against your classmates. The top 10 performing designs in the class earn bonus marks. We will evaluate performance using two metrics:

- The average number of instructions per clock cycle (IPC) that the processor can execute
- The processor's clock frequency (f_{max})

To encourage designs that use architectural innovation to increase IPC we will determine your processor's performance as

$$score = f_{max} \cdot IPC^3$$

The IPC will be calculated as the geometric average over a set of test programs. The set of test programs will be different from the microbenchmarks in the tester. We will release some, but **not all**, of these programs for you to test with. Instructions on determining f_{max} for your processor using Quartus are further below.

[4]Don't worry about the case where the target PC specified by the branch *is* actually PC+4. Consider this a taken branch, and make an optimization in Part V if you wish.

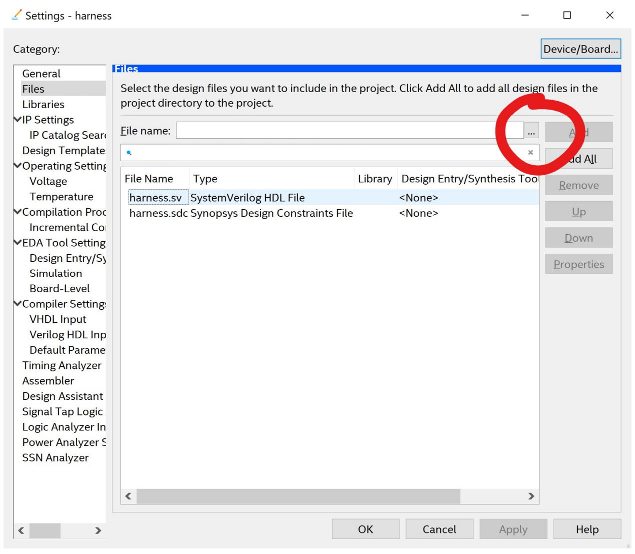
You are free to modify the processor as you wish, including adding or removing pipeline stages. It must still correctly execute all the instructions and be able to be compiled in the Quartus `harness` project. You can not change the processor's signal interface. **Your bonus design is not required to pass the IPC performance requirements of benchmarks 0 to 4.** For example, a branch predictor might improve performance on average but degrade performance for one of the benchmarks – this is OK. Here are some possible things you can do to improve performance:

- Add more pipeline stages.
- Have fewer pipeline stages.
- Look for signals that can be calculated a pipeline stage earlier than they currently are.
- Try to predict the outcome of branches in a smarter way than "assume always not-taken".
- Find a way to execute more than one instruction simultaneously, for an IPC greater than 1.
- You know what the instruction is by Stage 2 – see if you can do any work there.
- Find the critical path of your circuit as reported by TimeQuest.

Using the `harness` Project with Quartus

We have provided a Quartus project for you which will allow you to check that your processor is hardware-synthesizable. Your processor **must** compile using the `harness` project. To check if your code is synthesizable:

1. Extract the Quartus project files from cpu_harness.zip and open the project [5].
2. Go to *Project* → *Add/Remove Files in Project...* and click the '...' icon to browse for files (see figure 3). Select the file(s) containing your processor implementation. The files should then be listed with the `harness` files. Click OK.



Select the design files you want to include in the project. Click Add All to add all design files in the project directory to the project.

File Name	Type	Library	Design Entry/Synthesis Tool
harness.sv	SystemVerilog HDL File		<None>
harness.sdc	Synopsys Design Constraints File		<None>

Figure 3: Adding files to Quartus project.

[5] On the U of T Windows machines, opening the project from the W drive works (in my experience). Quartus may have trouble with finding files in networked storage for some other locations.

3. Run *Analysis & Synthesis* then *Fitter (Place & Route)*.
4. Verify that the compilation was successful, that the code is hardware-synthesizable and causes no compilation errors or warnings about latches or combinational loops.

Determining Maximum Frequency (f_{max})

This section is only required for students trying the bonus.

To obtain your f_{max}, first compile your processor with the above steps. Then run *Timing Analysis.* From the compilation report select *Timing Analyzer → Slow 1100mV 85C Model → Fmax Summary.*

Improving Maximum Frequency

This section is only required for students trying the bonus.

To improve f_{max} you must decrease the critical path. The critical path is the path that contains the longest delay between a pair of registers. We provide here a brief overview of determining what the critical path is. It is up to you to determine how to decrease it. Usually by reducing the total amount of dependent logic on the critical path or by breaking up long stages into shorter parts.

1. From the compilation report select *Timing Analyzer → Slow 1100mV 85C Model → Timing Closure Recommendations.* This will display pairs source and sink registers which have the longest delay between them. Note: the names of these registers will be different from, but have some resemblance to, the names you used in your Verilog/SystemVerilog.
2. Click *Report recommendations for this path* beside the top entry in the table. Click OK on the pop-up titled *Report Timing Closure Recommendations.*
3. Here Quartus is telling you what the longest paths are and the cause. Also check out the other pages in the *Reports* section on the left side of Quartus.
4. Click *Reports → Timing Closure Recommendations → Detailed Per-Path Results.* Click *report timing* in the top row of the table and click *Report Timing* in the pop-up.
5. Under the *datapath* tab (*Data Arrival Path*) you can see a list of all the intermediate wires between the registers. These can help you determine what logic the signal is passing through between the registers.
6. Right click in the *Data Arrival Path* table and select *Locate Path → Locate in Technology Map Viewer.* This will show a pictoral representation of the path.

Netlist Viewer

This section is not required.

You may be interested in how your processor is translated into hardware. To view a schematic of your design click *Tools → Netlist Viewers → RTL Viewer* or *Technology Map Viewers (Post-Fitting).* The first is more readable but has not been optimized, but the second is a better representation of the actual hardware that would be implemented.

Testing Your Processor

You must ensure that your design is hardware-synthesizable. Specifically, **it must be synthesizable using the Quartus `harness` project**. See instructions above on *Using the `harness` Project with Quartus.*

You can test your processor almost exactly as you tested your processor for lab 5. The testbench (tb.sv) is almost exactly the same and the programs are the same. **Note: the programs for this lab have**

different performance requirements than lab 5 which turns up in the hex files not being quite identical (they were compiled using the `-L7` flag).

For students attempting the bonus we have released some of the programs that we will use to test your IPC (`7_gemm`, `9_dfs`, `10_sssp`). You can use these to test your processor's IPC, but for more accurate results we recommend you also create some more test programs of your own.

Submission

To help you move forward in stages this lab is split into 4 parts plus a bonus. These stages are just a suggestion so feel free to implement in whatever order you prefer.

For the automarker you will submit two files: part1.sv and part2.sv. The first is your pipelined processor implementing parts I to IV. The second is optional and is your processor for the bonus submission. If you choose not to try the bonus, you are not required to submit a `part2.sv` file. Submit using the following command: `submitece342s 7 part1.sv part2.sv`

Processor designs must successfully compile in the Quartus harness project to receive full marks. For the bonus, the 10 students with the highest performance figures will be awarded up to an additional 5% on their final course grades, depending on the performance of their processor.