Lab 7: Processor Optimization

Introduction

In this lab, you will improve the performance of the processor you designed in Lab 5. Performance is defined by the number of instructions per second that can be executed, and is the product of:

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

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 0.3 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_mem_addr = PC` `PC = PC + 2`
2	`IR = i_mem_rddata`
3	*(execute instruction A)*
4	`o_mem_addr = PC` `PC = PC + 2`
5	`IR = i_mem_rddata`
6	*(execute instruction B)*

#	Operations			
1	`o_mem_addr=PC`	`PC=PC+2`		
2	`o_mem_addr=PC`	`PC=PC+2`	`IR=i_mem_rddata`	
3	`o_mem_addr=PC`	`PC=PC+2`	`IR=i_mem_rddata`	*(execute A)*
4	`o_mem_addr=PC`	`PC=PC+2`	`IR=i_mem_rddata`	*(execute B)*
5	`o_mem_addr=PC`	`PC=PC+2`	`IR=i_mem_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 provided testbench, running a suite of provided *micro-benchmark* programs that measure the processor's IPC in different situations using carefully-selected sequences of instructions. You will also measure the resource utilization and clock frequency of your processor using Quartus Prime. No hardware will be actually tested on the DE1-SoC board, however.

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.

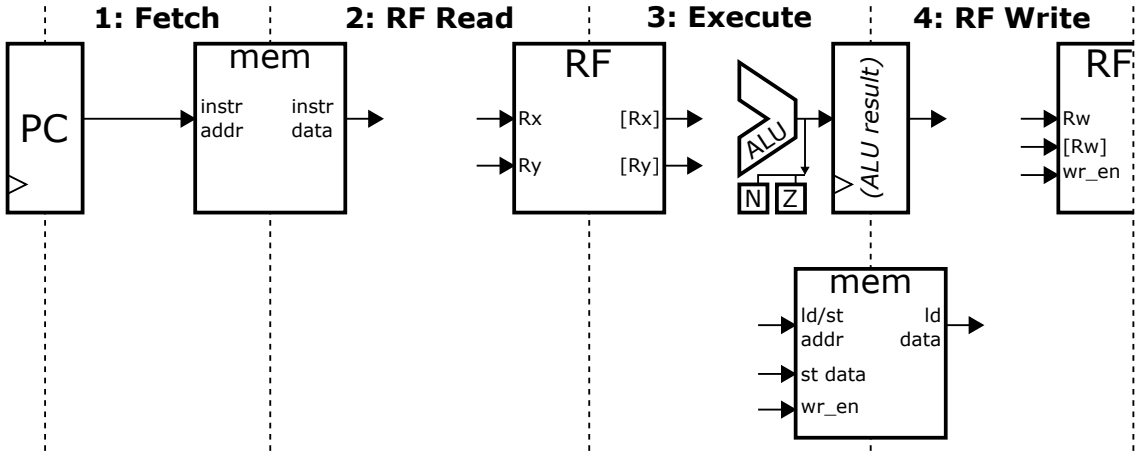

Figure 2: Pipelined processor datapath

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 2 (or overridden with a branch target address).

2. **Regfile Read**: The 16-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 `Rx` and `Ry`.

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 (and update N/Z). Branch instructions 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**: If the instruction needs to write to the register file, it does so here. Note that the register being written to (called `Rw`) can be a *completely different* one than the `Rx` or `Ry` 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. 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. This isn't possible with the memory interface from Lab 5 (it only had a single set of address/rddata/wrdata/write signals). 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. The signals are already declared in the starter kit. Memory Port 1 is read-only and is used for fetching instructions, and Port 2 is used for loads and stores and can be read and written. The timing for each port's signals is identical to Lab 5. There are no `waitrequest` or `readdatavalid` signals from Lab 6. This means that 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 and read enable are provided.

Signal	Direction	Size	Description
`clk`	input	1	Clock
`reset`	input	1	Active-high reset
`o_pc_addr`	output	16	Port 1 address (in bytes)
`o_pc_rd`	output	1	Port 1 read enable
`i_pc_rddata`	input	16	Port 1 read data
`o_ldst_addr`	output	16	Port 2 address (in bytes)
`o_ldst_rd`	output	1	Port 2 read enable
`i_ldst_rddata`	input	16	Port 2 read data
`o_ldst_wr`	output	1	Port 2 write enable
`o_ldst_wrdata`	output	16	Port 2 write data
`o_tb_regs`	output	[7:0][15:0]	Regfile contents

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, so the work will be divided into multiple parts that build on each other. In this first part, you will create an initial version of the pipelined processor that can correctly execute very simple programs. These programs don't have branches of any kind, and don't have instructions that need to read a register written to recently by an earlier instruction. You can skip implementation of the branch instructions (instructions that start with a `j__` or `call`) until Part III.

Project Setup

Start by unpacking the starter kit for this lab. You will find an empty skeleton `cpu.sv` with all the signals from Table 1 declared for you. There is also a ModelSim testbench (`tb.sv`) and six programs that test your pipelined processor in specific scenarios. Finally, there is a Quartus project called `harness` which is used to measure the clock frequency and FPGA resource utilization of your processor. It also makes sure that the Verilog/SystemVerilog you write is synthesizable on an FPGA.

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

The `o_tb_regs` output of your top-level processor module should contain the current contents of your register file. It's a two-dimensional signal, with outer index 0 containing the 16 bit value of `R0`, and so forth. The testbench uses this to verify the results of the processor.

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. 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 five 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.

Part II: Dependent Instructions

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

```
mvi r0, 2
add r1, r0
add r2, r0
```

The instructions are fetched sequentially and start moving forward in the pipeline. When the `mvi` reaches Stage 4 (RF Write), the first `add` is in Stage 3 (Execute) and the second `add` is in Stage 2 (RF Read). If proper measures are not taken, this code will not execute correctly, because both `add` instructions will use the **old** value of `r0`, which the `mvi` 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 (`r0` 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* 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 `[Rx]` and/or `[Ry]` with Stage 4's `[Rw]`.

After you add the forwarding logic, you should be able to pass the `1_arithdep` test.

Part III: Branches

Branches here refer to any of the eight instructions that can change the default control flow of the processor: `j jz jn call jr jzr jnr callr`. 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 lcoated at PC+2 and PC+4. This leap of faith is required if you have any hope of reaching an IPC of 1.

For this lab, your Fetch stage can assume that the next PC is PC+2 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 an unconditional branch (`j jr call callr`), or if it's a conditional branch (`jz jn jzr jnr`) **and** the condition is true. This changes the PC to something else than PC+2[1]. 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.
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 all six benchmark programs. Taken branches will degrade the IPC of the processor, and this is expected. The `3_branch_taken` test has a minimum expected IPC of 0.5 to reflect this. The final `5_capital` test has a minimum IPC of 0.4, and its correctness requires it to print out the proper capitalized sentence, in addition to a Pass output from the testbench.

Part IV: Competition (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 fastest designs in the class earn bonus marks.

Your score in the competition which will be calculated using the equation shown below:

$$Score = Fmax_{normalized} \times IPC_{5\_capital}$$

The IPC will be calculated for the `5_capital` test program. To obtain your Fmax, from the compilation report select *TimeQuest Timing Analyzer* → *Slow 1100mV 85C Model* → *Fmax Summary* To get your normalized Fmax, divide this number by 100Mhz. (The goal of normalizing the Fmax is to balance the contributions of your design's clock speed and its IPC.)

You are free to modify the processor as you wish, including adding or removing pipeline stages. It must still correctly execute all the benchmark programs and be able to be compiled in the Quartus harness project. You can not change the processor's signal interface. Here are 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.

[1] Don't worry about the case where the target PC specified by the branch *is* actually PC+2. Consider this a taken branch, and make an optimization in Part IV if you wish.

- Find the critical path of your circuit as reported by TimeQuest.

Modifications that are too closely-tailored to the `5_capital` benchmark will be deemed clever, but unacceptable, at your TA's discretion. For example, this includes creating a lookup table that knows in advance the outcome of every branch for the benchmark.

Marking Scheme

This is a two week lab. Have the following done for the first week:

- Preparation: Verilog/SystemVerilog code for basic Part I pipelined processor without support for branches or dependent instructions (2 marks)
- In-Lab: Demonstrate successfully passing the `0_basic` testcase in ModelSim (2 marks)

Have the remaining items done for the second week:

- Preparation: Verilog/SystemVerilog code for Part II processor that includes support for dependent instructions through forwarding (2 marks)
- Preparation: Verilog/SystemVerilog code for Part III processor that includes support for branches (2 marks)
- In-Lab: Demonstrate successfully passing the `1_arithdep` testcase in ModelSim (2 marks)
- In-Lab: Demonstrate successfully passing the `2_branch_nottaken` testcase in ModelSim (2 marks)
- In-Lab: Demonstrate successfully passing the `3_branch_taken` testcase in ModelSim (2 marks)
- In-Lab: Demonstrate successfully passing the `4_memdep` testcase in ModelSim (2 marks)
- In-Lab: Demonstrate successfully passing the `5_capital` testcase in ModelSim (2 marks)

Processor designs must successfully compile in the Quartus harness project to receive full marks. For the bonus, the 10 groups 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. Performance results will be collected during the second lab week, and students notified afterwards.