ECSE425 Final Project - Pipelined Processor

Winter 2021

due date: April 9, 2021

You will implement a standard five-stage pipelined 32-bit MIPS processor in VHDL. A pipelined processor stores control signals and intermediate results associated with each executing instruction in pipeline registers between stages. Refer to the text for further information. For full credit, implement:

- Hazard Detection
- Forwarding

1 Background

1.1 Instruction Set Architecture

You will implement a subset of the MIPS instruction set architecture.Refer to the MIPS Reference Data card: MIPS_Green_Sheet

To ensure compatibility with the provided assembly language programs, your instruction formats must conform to those in the MIPS Reference Data card. You can also refer to the MIPS Assembler Specifications document in the project files for reference. You may assume that all instructions—including *mult*—require a single cycle in the EX stage.

Your processor must be able to execute each of the required assembly language instructions. Assume that execution begins at address 0x0 in memory. You have to implement the instructions shown in the table below:

Class	Instruction	Mnemonic
Arithmetic	Add	add
	Subtract	sub
	Add Immediate	addi
	Multiply	mult
	Divide	div
	Set Less Than	slt
	Set Less Than Immediate	slti
Logical	And	and
	Or	or
	Nor	nor
	Xor	xor
	And Immediate	andi
	Or Immediate	ori
	Xor Immediate	xori
Transfer	Move From HI	mfhi
	Move From LO	mflo
	Load Upper Immediate	lui
Shift	Shift Left Logical	sll
	Shift Right Logical	srl
	Shift Right Arithmetic	sra
Memory	Load Word	lw
	Store Word	sw
Control-flow	Branch On Equal	beq
	Branch On Not Equal	bne
	Jump	j
	Jump Register	jr
	Jump and Link	jal

1.2 Processor Datapath

The figure below illustrates the datapath of the pipelined MIPS processor:

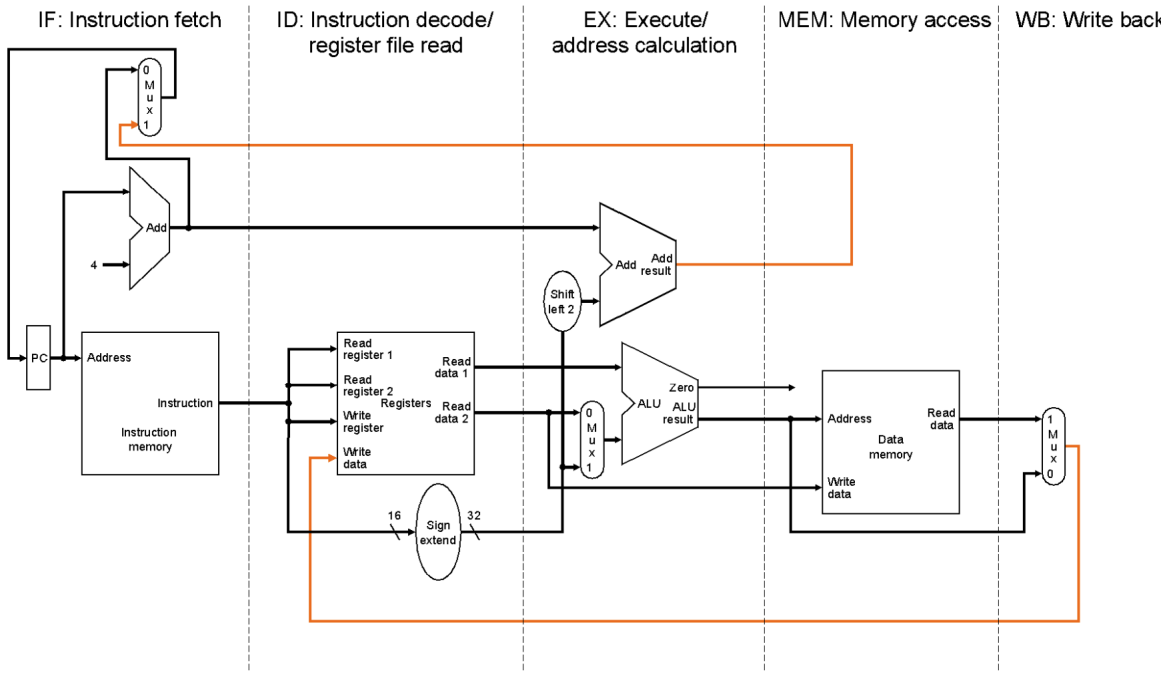

Figure 1: Pipelined Procesor Datapath

Source: https://cseweb.ucsd.edu/classes/su06/cse141/slides/s09-pipeline-1up.pdf

Note that, although this diagram has an explicit block indicating sign extension, some I-type instructions do not sign-extend but rather zero-extend. Make sure that your processor performs the appropriate extension by checking the MIPS Reference Data card. Instructions which sign-extend have "SignExtImm" in the operation description, while instructions which zero-extend instead have "ZeroExtImm".

Beyond the required instructions, your processor **must implement** the following functionality:

- Register $0 must be wired to 0x0000, and
- The PC must be initialized to 0x0.

Your processor **need not** implement the following functionality:

- Floating point arithmetic, and
- Interrupts and Exceptions.

Also, you may assume that any program we test on your processor will end in an infinite loop, as the two example programs we have given you do.

1.3 Assembler

To allow you to try different MIPS programs on your processor, we are providing you with an assembler which can convert MIPS assembly into machine code. For our purposes, MIPS assembly is restricted to the subset of functionality supported by the MiSaSiM instruction set simulator.

The provided assembler supports the following functionality:

- Comments, indicated with "#",
- Empty lines,
- Arbitrary whitespace (spaces, tabs) within lines,
- Labels, if present at the beginning of a line and ending with ":",
- Instruction arguments using numbered registers, such as "$0" and "$31" and
- Branches and jumps to labels.

The output of the assembler is an ASCII text file, with one 32-bit word on each line, encoded in binary (i.e., 32 '1's or '0's), in ascending order; this is the decoded instructions and the expected input format for your processor instruction memory. Your processor's output files (**register_file.txt** and **memory.txt**) should also have this format (lines of 32-bit binary code). Refer to the readme file of the assembler for more instructions on how to use it.

2 Implementation Guide

2.1 Project Overview

We are providing you with two test programs and their corresponding register_file and memory_file for verification of your implementation. You can also use the assembler to write your own program for testing. Your processor will take the **program.txt** file generated by the assembler as input instructions, execute these instructions, and output the register file and data memory file after execution as txt files.

2.2 Hazard Detection

In order to function properly, your pipelined processor must implement hazard detection. Hazard detection stalls instructions in ID when a required operand is not ready yet (e.g., due to a pending ALU operation or load instruction). A stall (or bubble) can be inserted in the pipeline by inserting an *add $r0, $r0, $r0* instruction into the EX stage rather than the waiting instruction.

Implement hazard detection logic and the corresponding control logic such that instructions dependent operands that are not yet available stall in ID. It is recommended that you implement and test hazard detection first.

2.3 Forwarding

Forwarding takes results from the EX/ME and ME/WB pipeline registers and makes them available as ALU inputs in order to eliminate stalls. Without forwarding, an instruction stalled in ID cannot proceed until the cycle during which the required operand is written back to the register file.

Implement forwarding from the EX/ME and ME/WB pipeline registers and the corresponding control logic such that stalls are eliminated when possible. It is recommended that you implement forwarding second, after hazard detection has been tested, and that you update hazard detection accordingly.

2.4 Memory

To implement the memory for instructions and data, use the memory model provided for the Cache project. You would need separate memories for instructions and data. You may alter the memory model as you see fit (e.g., set the memory delay to 1 clock cycle, if it makes your life easier). However, you must keep the data memory sized at 32768 bytes, and you must initialize the data memory to all zeros. You must also ensure that your processor can run a program of at most 1024 instructions.

Since the data memory has 32768 bytes, "memory.txt" should have 32768/4 = 8192 lines, one for each 32-bit word. Likewise, since there are 32 registers, "register_file.txt" should have 32 lines.

2.5 Suggested Components

Below are some suggestions on how you might want to breakdown the project into different components. Note that these are just suggestions, you are free to implement the processor in any way that you see fit.

2.5.1 Instruction_Memory.vhd

- Adapted from 'memory.vhd' of the Cache project
- Read from 'program.txt' outputted by the Assembler
- Address of memory is defined by byte (integer range from 0 to RAM_SIZE-1), where RAM_SIZE is 32768 bytes
- MIPS processor instructions are always in words, so you need to concatenate 4 bytes together in the end to form the 32-bit instructions that you can pass on to later pipeline stages

You need to define the entity instruction_memory in a similar way as 'memory.vhd' from the Cache project, except the writedata and readdata signals are 32-bits instead of 8-bits, as we are reading from 'program.txt' line-by-line.

2.5.2 Fetch.vhd

- Fetch instructions from instruction_memory
- Update PC accordingly (+4 for basic instructions, specified address for branches and jumps)

2.5.3 Decode.vhd

- Define register file here (32 registers, R0 is always zero)
- Decode instructions according to the MIPS Assembler specifications
- Sign-extension can be implemented here
- Handle hazard detection and forwarding

2.5.4 Execute.vhd

- Define execution of the instructions (ADD is Rd = Rs + Rt, etc.)
- Continue to deal with hazard detection and forwarding

2.5.5 Memory.vhd

- Handle read/write to data memory (lw, sw, etc.)
- Prepare for write-back stage
- Continue to deal with hazard detection and forwarding

2.5.6 Write_Back.vhd

- Write back to register file (the number written back to the registers will be used in the Decode stage)

2.5.7 Pipelined_Processor.vhd

- Connect all components together (initialize them first and then connect them to appropriate signals)
- Write register values and data memory to 'register_file.txt' and 'memory.txt', respectively

2.5.8 Testbench.tcl

Compile and run your code. Your testbench should run for 10,000 clock cycles, after which point the simulation should stop and the output should be written to the proper files. You may assume that we will not test your processor on a program with a duration of longer than 10,000 clock cycles. Give your clock a frequency of 1 GHz.

3 Grading

To test and grade your processor, the instructional staff will run your testbench.tcl and check whether the contents of the output files (register_file.txt and memory.txt) are correct. We are providing you with a subset of the suite of assembly programs which we will use to test your processor. In the event that your results do not match the correct results, your processor will be inspected to the extent possible and partial credit will be awarded accordingly.

Note1: we will rely heavily on comments provided in your code to evaluate incomplete implementation, therefore obtuse, uncommented code will receive substantially less partial credit.
Note2: if you fail to submit a readme file on how to run your code or if your code does not run properly (e.g. compile or run-time error), you will RECEIVE A GRADE OF 0. Even if your implementation is incomplete, your intermediate-stage code still needs to run without errors and output intermediate results.

3.1 Grade Breakdown

- **Completeness (20%)**
  - Did you submit all the materials required?
- **Correctness (40%)**
  - Is the processor implemented correctly? Do the register_file and memory_file match with the benchmarks?
- **Report (40%)**
  - Is your report clear and free of grammatical errors and typos?
  - Did you effectively explain your thought process?
  - Does your report contain all the required discussions?
- **Peer Evaluation**
  - Each member grades other members based on their contribution (submitted separately and confidentially)
- **BONUS TASK: Processor Optimization (+30% of Project Grade)**

  Optimize and evaluate the optimization of your processor. Below are some potential optimization ideas you might consider implementing:

  Caching: Instead of using separate memories for instructions and data, use a single main memory with an instruction cache and data cache. You will need to create a simple arbiter to coordinate access to the main memory by the two caches. It would make sense to set the delay of your memory model to a suitably high number of clock

cycles, to simulate the effect of a miss penalty. You could also use the CACTI tool to optimize the size and associativity of your cache.

Branch prediction: Implement a branch target buffer for branch prediction in IF, 1-bit, 2-bit, tournament predictors, or go even deeper and use a neural network to predict branches!

Multiple-issue: For this optimization, you could implement either static scheduling or dynamic scheduling. If you go with dynamic scheduling, you could implement the Tomasulo architecture covered in class or a more modern architecture.

SIMD: You could implement arithmetic instructions to operate on multiple words in parallel (or four 8-bit wide operands, to save memory bandwidth).

If you decide to do the bonus task, please send an email to the course instructors to confirm your choice. Include sufficient detail to make it possible for us to judge the design complexity of your choice (e.g., which branch predictors, or cache organizations, you will compare).

3.2 Report Guideline

You would need to submit a report as a team. Your report must be in IEEE format and a maximum of 6-pages (an extra page is allowed for references, if you have any). If you decide to implement the bonus optimization task, your report can be maximum of 8-pages. We strongly recommend students to write their report in LaTeX (Overleaf is a good tool to use). Your report should include the following sections:

- **Introduction:** Define the goal, summarize the project task, explain general approach taken
- **Methodology:** Explain your implementation of the major components in details, especially how you handled data detection and forwarding
- **Results + Discussion:** Discuss your results, do they match with the benchmark files? If your results do not match or if you are unable to complete the project, this section is especially important. You might want to show some intermediate results (after various pipeline stages such as Decode or Execute, for example) to show that your processor is partially implemented.
- **Optimization Results:** You only need to write this section if you implemented the bonus optimization task. Compare and discuss the results before and after optimization.
- **Limitations:** If your processor does not work correctly or if you are unable to complete the project, clearly state which parts you are unable to complete/implement correctly. If you think some components do not work correctly, you have to explain what might went wrong.
- **Conclusion:** Summarize the key takeaways of the project.

4 Submission Instructions

Submit a zip-folder on MyCourses containing the testbench.tcl file, all relevant VHDL files, report, and readme file on how to run your code. Late submission will be deducted 20% every 24 hours from the deadline, for up to 5 days.