ECE 429: Project Information

Hiren Patel

May 10, 2020

List of Tasks

Required Submissions

Overview of Project

You will build a 5-stage pipelined processor using the synthesizable subset of Verilog at the register-transfer level (RTL). The processor will implement the RISC-V instruction-set architecture (ISA). Please note that this document supplements that details of the processor implementation that will be a part of the course lecture notes and discussions.

The project will be completed individually. The project is split into six deliverables as defined next.

The deadlines for each of these project deliverables (PDs) will be posted on LEARN.

The high-level overview of the deliverables is as follows.

- All PDs until PD4 will work towards implementing a single-cycle datapath as a first version of our processor. This has the advantage that we can implement all the necessary datapath components, and minimal control logic to correctly execute all benchmarks. This ensures correct implementation of instruction semantics.
- Once this is done, PD5 requires pipelining the processor with forwarding.

Advice. My general advice is to get started immediately, and work ahead. There are quite a few PDs that you could do before we actually cover the content in the course. However, once we have covered pipelined execution, I highly encourage you to work ahead, and well before the deadline date. You have access to all the information necessary. Since there is a mandatory demo of the full processor, I encourage you to do this well in advance as well since we have to complete all demos before the last day of class.

RISC-V Benchmarks

You may use the following benchmarks to test your processor design. Please do this throughout the entire design process. I **highly encourage you to also create your own tests**. You can setup your own compiler toolchain as well https://riscv.org/software-tools/risc-v-gnu-compiler-toolchain/. The repository with the benchmarks has a Makefile that will show you how to build your own binaries. Please let me know if any issues arise.

In order to get access to the benchmarks, you will need to clone the git repository as described below. You would have to login to git.uwaterloo.ca with your Waterloo ID, and then you should be able to access the repository: https://git.uwaterloo.ca/ece429/rv32-benchmarks/.

Download benchmarks

```
git clone gitlab@git.uwaterloo.ca:ece429/rv32-benchmarks.git
```

You will notice that each benchmark has multiple files with different extensions.

- `.c`: The source code in C.
- `.s`: The assembly language code.
- `.d`: The disassembler output. This is very useful in understanding the instructions, and their locations in memory.
- `.x`: The binary file that you will load into the memory for execution.

1 PD0: Setup and tutorial

PD0 – Task 1.1 Verilog tutorial

1. Step through the Verilog tutorial provided for this course.
2. Complete the exercises for PD0.

PD0 – Submission 1.1 Tutorial

There are exercises in the tutorial. Please submit a PDF file with the answers to those exercises in your submission.

2 PD1: Behavioural model of main memory

You will create a “magic” main memory module that is byte-addressable. That is, the memory should support addressing individual bytes (8 bits). The main memory module is to be modelled at the behavioural level. This means that it does not have to be synthesizable. The main memory module must have the following properties.

- The memory will have the following ports: clock, `address` (32-bits), `data_in` (32-bits), `data_out` (32-bits), and `read_write` (1 bit).
  - `clock`: Self-explanatory what this does.
  - `address`: A 32-bit address.
  - `data_in`: The data to be written into the memory at the provided `address`.
  - `data_out`: The data response from the main memory to the `address` provided.
  - `read_write`: Whether the memory is being read from or written to.

Make the depth of the memory configurable (default 1MB) by using a `define` in verilog. We should be able to change this easily with a single change in the Verilog source; however, I don’t expect our benchmarks to be larger than 1MB. The address and data port for the memory module should be 32-bits wide. This means that whenever a 32-bit address is supplied on the address line, the memory module returns 32-bit data out on `data_out`. Memory should use little-endian ordering. The program code (instructions) starts at address `0x01000000`. This means that we would start fetching instructions from address `0x01000000`, and data would follow the instruction segment. Note that the program counter’s (PC) default starting address should be `0x01000000`. The main memory should return the 32-bit data word within one clock cycle. To do this, make the reads combinational, and the writes sequential as discussed in class. This means that on a read operation, the output on `data_out` has the value at the specified `address` in the same clock cycle. However, on a write operation, the new value is only available to be read in the next clock cycle.

In order to load the binaries, you should use an **`initial`** block to read the `.x` file. I suggest that you read it using `readmemh()` into a temporary array, and then copy it within the **`initial`** block into the main memory array. You could certainly do stores into the memory, but the above suggested approach is simpler and faster in simulation.

PD1 – Task 2.1 Testbench for main memory module

Write a testbench that will produce the following output.

- The `address` and its corresponding contents in the main memory for every example provided in the benchmark suite. You should retrieve this by performing reads from the memory. Hence, show that the data returned after placing the address. This should be done for only the addresses that contain data loaded from the binary.

PD1 – Task 2.2 VCD

Create a value-change dump (VCD) file that displays all value changes on all ports for the benchmark `SumArray`.

PD1 – Submission 2.1 Fetch stage

Please submit the following files in a tar (or zipped) file on LEARN.

- Source code for memory module, testbench, and the VCD file.
- A text (TXT) file with the output from the simulation.
- An image file showing a portion of the VCD waveform. This should show that the memory responds with the data in one clock cycle.

3 PD2: Fetch and decode stages

You will create the fetch and decode parts of the instruction execution loop.

Fetch stage. The fetch stage maintains a program counter (PC) register. The PC holds an address, which is sent to the main memory module. The output from the main memory are the instruction bits, which is often referred to as the instruction word. Once the address in PC is supplied to the address lines of the main memory, the fetch stage increments the PC by 4. Thus, in the next cycle, the next instruction will be fetched from the memory. In order to interact with the main memory module, you will have to keep in mind the following.

- `read_write`: Should be wired to always read. Instructions are typically only read-only.

PD2 – Task 3.1 Fetch stage: Fetching instructions

Write the Verilog specification for fetching instructions from the main memory. Note that you should already have this done as a consequence of the testbench from PD1. You would need to primarily introduce a PC register for holding the address of the next instruction, and the increment hardware. You **do not** have to create this as a separate module. This will avoid unnecessary ports and interfacing.

Decode stage. The decode stage splits the instruction word into its respective fields. This allows the hardware to identify the instruction, its operands, and generate any control signals that may be necessary to drive other datapath components. This decode stage must be combinational. Connect the decode stage to the fetch logic. The decode stage should support all the instructions shown on the next page.

PD2 – Task 3.2 Decode stage: Decoding instructions

Write the Verilog specification for decoding the fetched instructions. Use `$display()` to output the learned information (operands, opcodes, fields, etc.) to the terminal. You **must confirm that all instruction** you see in the disassembly are identified by your decoding logic.

PD2 – Submission 3.1 Decode stage

Please submit the following files in a tar (zip them up) file on LEARN.

- Source code for fetch logic, decode module, testbench, and the VCD file.
- A text (TXT) file with the output from the simulation. Ensure that for every instruction you `$display()` the PC, instruction word bits, opcode, and, the source and destination operands.
- An image file showing a portion of the VCD waveform. This should show that the decoding happens within the same clock cycle as when the `instruction` is activated with the instruction word bits. Using an image editor tool such as gimp, and please annotate the image to show this clearly.

Please note the following: For the individual instruction tests, a simple way to terminate the program is to identify the `ecall` instruction. You do not need to implement the actual functionality of `ecall`.

27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12 10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

RV32I Base Instruction Set

imm[31:12]					rd	0110111	LUI
imm[31:12]					rd	0010111	AUIPC
imm[20|10:1|11|19:12]					rd	1101111	JAL
imm[11:0]			rs1	000	rd	1100111	JALR
imm[12|10:5]		rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]		rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]		rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]		rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]		rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]		rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]			rs1	000	rd	0000011	LB
imm[11:0]			rs1	001	rd	0000011	LH
imm[11:0]			rs1	010	rd	0000011	LW
imm[11:0]			rs1	100	rd	0000011	LBU
imm[11:0]			rs1	101	rd	0000011	LHU
imm[11:5]		rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]		rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]		rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]			rs1	000	rd	0010011	ADDI
imm[11:0]			rs1	010	rd	0010011	SLTI
imm[11:0]			rs1	011	rd	0010011	SLTIU
imm[11:0]			rs1	100	rd	0010011	XORI
imm[11:0]			rs1	110	rd	0010011	ORI
imm[11:0]			rs1	111	rd	0010011	ANDI
0000000		shamt	rs1	001	rd	0010011	SLLI
0000000		shamt	rs1	101	rd	0010011	SRLI
0100000		shamt	rs1	101	rd	0010011	SRAI
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	010	rd	0110011	SLT
0000000		rs2	rs1	011	rd	0110011	SLTU
0000000		rs2	rs1	100	rd	0110011	XOR
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	111	rd	0110011	AND
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK
csr			rs1	001	rd	1110011	CSRRW
csr			rs1	010	rd	1110011	CSRRS
csr			rs1	011	rd	1110011	CSRRC
csr			zimm	101	rd	1110011	CSRRWI
csr			zimm	110	rd	1110011	CSRRSI

DO NOT IMPLEMENT

4 PD3: Register file and execute stage

Register file. Implement a register file as presented in class for the subset of the RISC-V ISA instructions decoded in PD2. We discussed the register file datapath design in class. That is the register file you are required to implement. The register file will be a synchronous component. Once again, reads in the register file are combinational, and the writes are sequential. The timing behaviour of the register file should be as follows: the output of the reads are available within the same clock cycle as when the address for the registers is supplied, and the writes to the registers are only available to be read in the next clock cycle. You should see a resemblance between your memory module's implementation and your register file.

PD3 – Task 4.1 Register file

Implement the register file design for the RISC-V ISA using Verilog. The register file should have the following ports.

- `clock`: 1-bit input.
- `addr_rs1`: 5-bit input to select the first source register.
- `addr_rs2`: 5-bit input to select the second source register.
- `addr_rd`: 5-bit input to select the destination register.
- `data_rd`: 32-bit input to write into the destination register.
- `data_rs1`: 32-bit output of the contents of register specified by `addr_rs1`
- `data_rs2`: 32-bit output of the contents of register specified by `addr_rs2`
- `write_enable`: 1-bit input to write the contents of destination register specified by `addr_rd`. If `write_enable` is not asserted, then the register file does not do any writes.

Execute stage. The execute stage implements the immediate generation, branch comparison, and arithmetic logic unit (ALU) components. Note that branches get resolved in the execute stage, and the effective address is also computed in the execute stage. The entire execute stage is combinational.

PD3 – Task 4.2 ALU

Implement the three components of the execute stage, and connect it to the output of the decode stage. The implementation of the execute stage was discussed in class.

PD3 – Submission 4.1 Register file and execute stage

Please submit the following files in a tar (zip them up) file on LEARN.

- Your source code for the fetch, decode, and execute stages; your testbench; and, your VCD file.
- A text (TXT) file with the output from the simulation of the register file. For every instruction you decode and supply the inputs to the register file, show the outputs of the register file on the terminal.
  - Initialize the register file with values that are the same as its index. That means register x0, x1, x2 have values initialized as 0, 1, 2, ...
- A text (TXT) file with the output from the simulation with the execute stage. Note that you could combine this with the previous text file. For each instruction that is executed, show the inputs to the execute stage and the computed result. If the instruction is an ALU operation, then the result of the operation, and if it's a control instruction then the effective address, and whether it is taken or not.
  - You may display this information on the terminal as well.
- An image file showing a portion of the VCD waveform. This should show the timing behaviour of the register file for both reads and writes.

5 PD4: Memory and write-back

Memory stage. We are going to simplify the memory stage for ourselves by creating another memory module similar to the one we created for PD1. This makes sense because both instruction and data are loaded into the memory. The difference in the memory stage will be that we will **not** be wiring the `read_write` to only read. This is because we want to perform the operation specified by the instructions: for a store, we perform a write, and for a load, we perform a read. Note that you have memory operations that do load-byte, and store-byte as well. You can retrieve the full word from the data memory, and strip out the required data in the memory stage.

There are some additions you will have to make in order for the data memory to work correctly. Recall that we will allow byte and half-word loads and stores to the data memory. My recommendation is to do the following.

Loads For a given address, retrieve the 4-byte word that contains the corresponding byte or half-word, and return it on the data output lines. The correct 32-bit value can be organized as per the specification in the processor pipeline.

Stores To perform a half-word or byte store, we would need to make minor modifications to the memory interface. I suggest adding an `access_size` port that indicates the size of the access to the memory. Consequently, a store of `access_size` of 2-bytes (half-word) would select the appropriate two bytes provided in the input, and write it to the memory. Note that you could implement support for loads using the `access_size` as well. The choice is entirely up to you.

PD4 – Task 5.1 Replicate memory

Write a module for the data main memory. Use this to connect to the necessary logic in order to perform memory operations. Note that there is a PC+4 component in the memory stage. Please implement that component as well. This is what will form the memory stage.

Write-back stage. The write-back stage takes the output from either the ALU or the memory, and writes it into the register file.

Important note. Make sure that every instruction exits the pipeline before the next one is inserted into the pipeline. Consequently, there is no need for hazard checking as yet. The first version of your processor will be a single-cycle processor. In effect, a correctly implemented single-cycle processor can serve as an ISA simulator for yourselves.

PD4 – Task 5.2 Write-back

Implement the necessary logic for the write-back stage.

With the completion of PD4, you should be able to execute all benchmarks correctly. Ensure that all benchmarks work correctly as the computed result will also be correct. I suggest you save a a snapshot of this version of the processor.

Hint. A simple way to terminate the simulation at the end of the program is by recognizing that at the end of `main`, a return is performed. This means that the stack pointer returns back to the initial value. My recommendation is that you initialize the stack pointer at the end of the memory region (1MB). Since the stack grows down towards lower address, this would be the largest addressable address that the stack will use. You can add a check within your simulation code to see if this initial stack pointer address is reached, and if it is, then you can simply use `$finish()` to complete the simulation.

For the individual instruction tests, a simple way to terminate the program is to identify the `ecall` instruction. Actual functionality of `ecall` is not required.

PD4 – Submission 5.1 Memory and write-back

Please submit the following files in a tar (zip them up) file on LEARN.

- Your source code for the fetch, decode, execute, memory, and write-back stages; your testbench; and, your VCD file.
- A text (TXT) file with the output from the simulation (for each of the benchmarks).
  - Please display the initial contents of the register file before the program executes.
  - Display the contents of the register file after execution. Note that the changes in the register file should reflect the program's execution, and values you see should be the correct results of the computation.
  - Perform this for all the benchmarks.

6 PD5: 5-stage pipelined RISC-V with bypassing

You will add support for pipelined execution, and bypassing. Recall that the basic method to avoid read-after-write (RAW) hazards is to insert nops (stalls) until the RAW does not exist. Note that a processor with RAWs will insert nops whenever there is a data dependence that cannot be resolved via forwarding.

PD5 – Task 6.1 Pipelined execution

Implement pipelined instruction execution, and ensure that you support forwarding.

PD5 – Submission 6.1 Pipelined RISC-V

Please submit the following files in a tar (zip them up) file. **You will submit this before your demo.**

- A TXT file showing the output of the simulation for every benchmark using the pipelined processor.
- Multiple image files using the provided benchmarks to illustrate that pipelined execution is working. You must annotate the images to highlight what you are attempting to show. This means you should show the following.
  - Correct pipelined execution (show that instructions are being executed in an overlapped manner)
  - Instance of stalls due to RAW.
  - Forwarding happening on each of the inputs, and every stage.
- A PDF file that combines the output from simulations, and the respective PNG image files.
- All files tarred together with the following folder structure.
  - Verilog source files in a folder called `v/`.
  - Image files in `img/`.
  - VCD files in `vcd/`.
  - TXT files in `txt/`.

You must show correct execution of all benchmarks provided. Show the initial state of the register file before the execution of the program, and the end state of the register file after executing the benchmarks (all benchmarks), and any additional information to inform me that the benchmark works correctly. Ensure to annotate the PDF. Highlight in one benchmark examples of stalls, and why they occur, and forwarding. You can pick any one of the benchmarks. The date, and time of your scheduled demo. Tar (zip them) up your Verilog files and submit it too.