

Computer Assignment 2

Multi Cycle Approach

RISC-V Processor

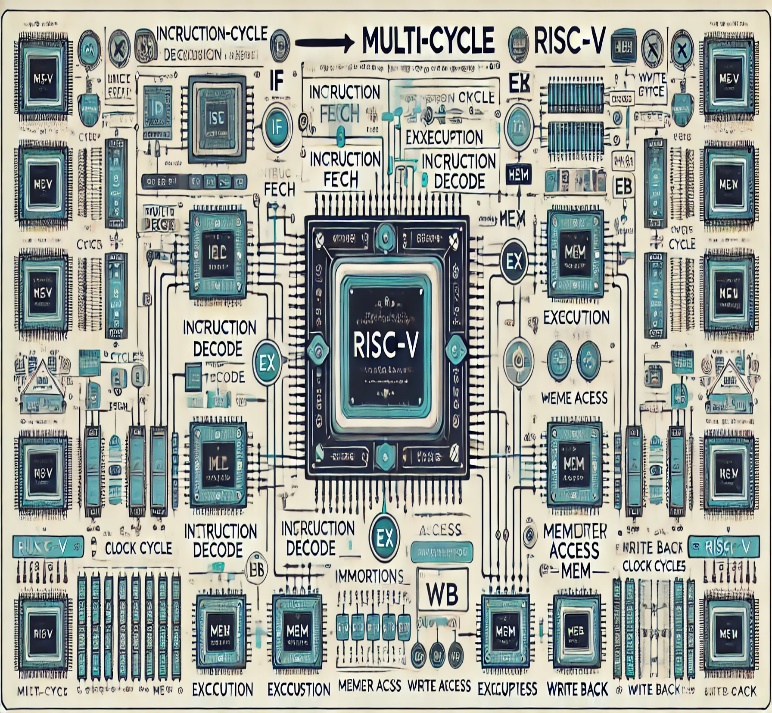


Bardia Amirian | 810101605

Amirreza Nadi Chaghadari | 810101538

Digital Systems 2

Dr. Safari



December 12, 2024

**Table of Contents**

[Problem Description 1](#_Toc184925389)

[Instruction Formats 2](#_Toc184925390)

[R-Type Instructions 2](#_Toc184925391)

[Instruction format 2](#_Toc184925392)

[Instruction Execution Table 2](#_Toc184925393)

[I-Type Instructions 3](#_Toc184925394)

[Instruction format 3](#_Toc184925395)

[Instruction Execution Table 3](#_Toc184925396)

[S-Type Instructions 4](#_Toc184925397)

[Instruction format 4](#_Toc184925398)

[Instruction Execution Table 4](#_Toc184925399)

[J-Type Instructions 5](#_Toc184925400)

[Instruction format 5](#_Toc184925401)

[Instruction Execution Table 5](#_Toc184925402)

[B-Type Instructions 6](#_Toc184925403)

[Instruction format 6](#_Toc184925404)

[Instruction Execution Table 6](#_Toc184925405)

[U-Type Instructions 7](#_Toc184925406)

[Instruction format 7](#_Toc184925407)

[Instruction Execution Table 7](#_Toc184925408)

[Datapath Design 8](#_Toc184925409)

[Abstract Block Diagram 8](#_Toc184925410)

[Designing Components 9](#_Toc184925411)

[Comparing with Previous Assignment 9](#_Toc184925412)

[ALU 9](#_Toc184925413)

[Control Unit 10](#_Toc184925414)

[ALU Controller 11](#_Toc184925415)

[PC Controller 13](#_Toc184925416)

[Main Control Unit 14](#_Toc184925417)

[Overall Description 14](#_Toc184925418)

[State Diagram 16](#_Toc184925419)

[Testing and Verification 17](#_Toc184925420)

[Writing Assembly Code 17](#_Toc184925421)

[Generating RISC-V Machine Code 19](#_Toc184925422)

[Testbench and Results 21](#_Toc184925423)

**Table of Figures**

[Figure 1. Datapath Abstract Block Diagram 8](#_Toc184925605)

[Figure 2. ALU Verilog Description 9](#_Toc184925606)

[Figure 3. ALU Controller Verilog Description 12](#_Toc184925607)

[Figure 4. PC Controller Verilog Description 13](#_Toc184925608)

[Figure 5. Main Control Unit State Diagram 16](#_Toc184925609)

[Figure 6. Minimum Finder Code in C language 17](#_Toc184925610)

[Figure 7. Assembly-Friendly Minimum Finder Code 17](#_Toc184925611)

[Figure 8. Assembly Code for Minimum Finder 18](#_Toc184925612)

[Figure 9. Assembly and Machine Code 19](#_Toc184925613)

[Figure 10. Top Module Testbench 21](#_Toc184925614)

[Figure 11. Register File Waveforms 22](#_Toc184925615)

[Figure 12. Memory Waveforms 22](#_Toc184925616)

**Table of Tables**

[Table 1. T-Type Instruction Format 2](#_Toc184925482)

[Table 2. R-Type Instruction Execution Table 2](#_Toc184925483)

[Table 3. I-Type Instruction Format 3](#_Toc184925484)

[Table 4. I-Type Instruction Execution Table 3](#_Toc184925485)

[Table 5. S-Type Instruction Format 4](#_Toc184925486)

[Table 6. S-Type Instruction Execution Table 4](#_Toc184925487)

[Table 7. J-Type Instruction Format 5](#_Toc184925488)

[Table 8. J-Type Instruction Execution Table 5](#_Toc184925489)

[Table 9. B-Type Instruction Format 6](#_Toc184925490)

[Table 10. B-Type Instruction Execution Table 6](#_Toc184925491)

[Table 11. U-Type Instruction format 7](#_Toc184925492)

[Table 12. U-Type Instruction Execution Table 7](#_Toc184925493)

[Table 13. ALU Operation Table 9](#_Toc184925494)

[Table 14. ALU Controller Functionality Table 11](#_Toc184925495)

[Table 15. Main Control Unit Tabular Explanation 14](#_Toc184925496)

# Problem Description

We are to design a simplified version of the RISC-V processor, using   
multi-cycle approach. It means that different instructions will take different numbers of clock cycles to execute, but the clock itself can become faster. Here is the list of instructions we are expected to implement:

**R-Type[[1]](#footnote-1):** add, sub, and, or, slt

**I-Type[[2]](#footnote-2):** lw, addi, xori, ori, slti, jalr

**S-Type[[3]](#footnote-3):** sw

**J-Type[[4]](#footnote-4):** jal

**B-Type[[5]](#footnote-5):** beq, bne

**U-Type[[6]](#footnote-6):** lui

Each of these instructions have their own format, thus requiring the datapath to include appropriate paths for the data to flow accordingly.

Once we designed the processor, we are to test it by running a code on it which searches for the minimum of an array with 10 elements.

# Instruction Formats

## R-Type Instructions

### Instruction format

Luckily, all R-Type instructions we are to implement are well-defined and are almost the same. The only thing that differs between them is what the ALU[[7]](#footnote-7) does with its inputs. the overall form of these instructions looks like this:

Table 1. T-Type Instruction Format

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 25 | 24 | 20 | 19 | 15 | 14 | 12 | 11 | 7 | 6 | 0 |
| f7 | | rs2 | | rs1 | | f3 | | rd | | OPC | |

### Instruction Execution Table

Table 2. R-Type Instruction Execution Table

|  |  |
| --- | --- |
| add |  |
| sub |  |
| and | (bitwise) |
| or | (bitwise) |
| slt |  |

## I-Type Instructions

### Instruction format

Immediate type instructions of this computer assignments have 3 types:

1. The arithmetic type which includes addi, ori, xori, and slti
2. Loading type which includes lw
3. Jumping type which includes jalr

Table 3. I-Type Instruction Format

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 |  | 20 | 19 | 15 | 14 | 12 | 11 | 7 | 6 | 0 |
| Imm[[8]](#footnote-8)[11:0] | | | rs1 | | f3 | | rd | | OPC | |

### Instruction Execution Table

Table 4. I-Type Instruction Execution Table

|  |  |
| --- | --- |
| addi | (Sign-Extended) |
| ori | (Sign-Extended) |
| xori | (Sign-Extended) |
| slti | (Sign-Extended) |
| lw | (Signed-Extended) |
| jalr |  |
| (Signed-Extended) |

## S-Type Instructions

### Instruction format

This assignment includes only one S-Type instruction, sw.

Table 5. S-Type Instruction Format

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 25 | 24 | 20 | 19 | 15 | 14 | 12 | 11 | 7 | 6 | 0 |
| Imm[11:5] | | rs2 | | rs1 | | f3 | | Imm[4:0] | | OPC | |

### Instruction Execution Table

Table 6. S-Type Instruction Execution Table

|  |  |
| --- | --- |
| sw | (Sign-Extended) |

## J-Type Instructions

### Instruction format

This assignment includes only one J-Type instruction: jal

Table 7. J-Type Instruction Format

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 25 | 24 | 20 | 19 | 15 | 14 | 12 | 11 | 7 | 6 | 0 |
| Imm[20, 10:1, 11, 19:12] | | | | | | | | rd | | OPC | |

The first thing to be noticed here, is how the immediate part contains bits 20:1, and not 19:0. This is because the value must be an integer coefficient of 4, meaning that its least significant 2 bits must be 0. Due to the weird choice made by the developers of RISC-V processor, the first one is ignored, but the second one must be included to avoid address exception! Therefore, the immediate could be interpreted like this:

There is also the weird format of giving the immediate, but it was another one of RISC-V processor’s developers’ choices we need to follow.

There is one more thing to be discussed here. If we do not specify rd, the default value will be ra. But, this is not the concern of this assignment, since we are not implementing an assembler.

### Instruction Execution Table

Table 8. J-Type Instruction Execution Table

|  |  |
| --- | --- |
| jal |  |
| (Signed-Extended) |

## B-Type Instructions

### Instruction format

There are 2 branch instructions we are expected to implement: bne and beq. They both have this structure

Table 9. B-Type Instruction Format

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 25 | 24 | 20 | 19 | 15 | 14 | 12 | 11 | 7 | 6 | 0 |
| Imm[12, 10:5] | | rs2 | | rs1 | | f3 | | Imm[4:1, 11] | | OPC | |

What is to be noticed here (again!), is the weird format of the immediate. It is not saved consecutively, and does not end with 0. It has a formula like this:

### Instruction Execution Table

Table 10. B-Type Instruction Execution Table

|  |  |
| --- | --- |
| beq |  |
| bne |  |

## U-Type Instructions

### Instruction format

This assignment includes only one U-Type instruction: lui

Table 11. U-Type Instruction format

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 25 | 24 | 20 | 19 | 15 | 14 | 12 | 11 | 7 | 6 | 0 |
| Imm[19:0] | | | | | | | | rd | | OPC | |

### Instruction Execution Table

Table 12. U-Type Instruction Execution Table

|  |  |
| --- | --- |
| lui | (Signed-Extended) |

With these tables at hand, we can design the datapath.

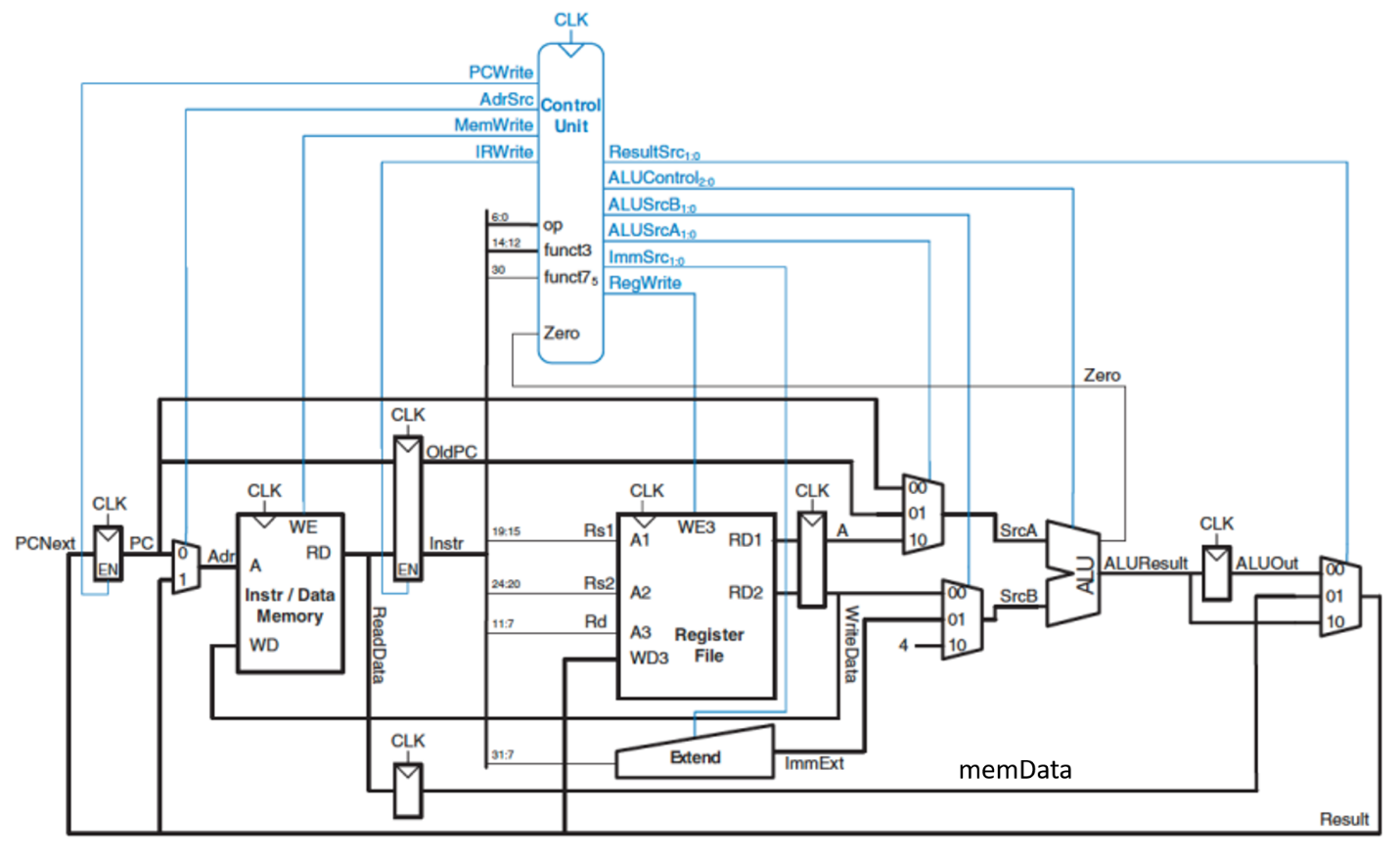
# Datapath Design

## Abstract Block Diagram

The idea here is to reuse the same hardware to handle different things in the cycle of executing each instruction. For example, when “LW” is being executed, the ALU can be used to calculate the next value of program counter, aka. PC, and calculate the address where the data is. In order to achieve this, we need to save anything that may be of use, before altering it.

Here’s how the datapath would look like:

Figure 1. Datapath Abstract Block Diagram



The “clocked” rectangles are the registers mentioned earlier[[9]](#footnote-10). Some of them have direct impact on logic of multi-cycle approach, meaning that removing them would result in design failure. Others are just there to shorten the critical path delay, allowing higher clock frequencies.

# Designing Components

## Comparing with Previous Assignment

The components used in this computer assignments are the same as the previous one. The only thing we altered was the ALU.

## ALU

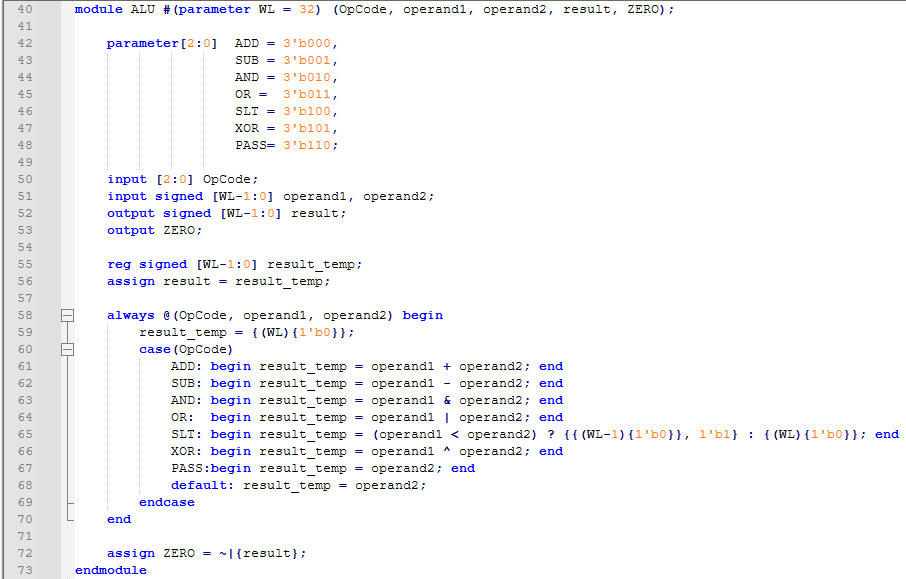
The new ALU operation table looks like this:

Table 13. ALU Operation Table

|  |  |  |
| --- | --- | --- |
| Operation Code Parameter | Operation Code | Operation |
| ADD | 000 |  |
| SUB | 001 |  |
| AND | 010 |  |
| OR | 011 |  |
| SLT | 100 |  |
| XOR | 101 |  |
| PASS | 110 |  |

Adding “PASS” operation makes the “LUI” instruction easily executable. As for the implementation, the Verilog description of this new ALU looks like this:

Figure 2. ALU Verilog Description



# Control Unit

We decided to change the control unit entirely, since we had the chance to redesign the whole thing. We partitioned the controller into 3 major parts:

* ALU Controller: Decides what the ALU does, based on what the Main Controller tells it.
* PC Controller: Decides when to update the program counter, based on what the Main Controller tells it.
* Main Controller: Handles the overall course of actions, including what the other two parts do

The detailed explanation of each partition is given in the next pages.

## ALU Controller

We use the ALU to handle every arithmetic and logic operation we do. Sometimes, the operation is defined by “f3” and “f7”. But, sometimes it has no dependency on them. For example, we use the ALU to calculate the next value of PC. This happens in the flow of each and every instruction’s execution, and has no dependency on anything. So, the ALU’s function is predetermined in some situations, but depends on the “f3” and “f7” in other ones.

The Main Control unit handles these situations. It determines when the ALU must do an operation without checking “f3” and “f7”, and when the operation depends on them. But, it does not handle the dependency of the operation on “f3” and “f7”. For example, if an R-Type instruction is being executed, the ALU must act based of “f3” and “f7”, but the Main Controller only tells that it must check these signals and does not specify which operation to perform. That’s where the ALU Controller comes in. The ALU Controller checks the Main Controller’s output. If it determines the operation, the ALU controller follows. Otherwise, it decides the operation based of “f3” and “f7”.

The table of ALU Controller’s functionality looks like this:

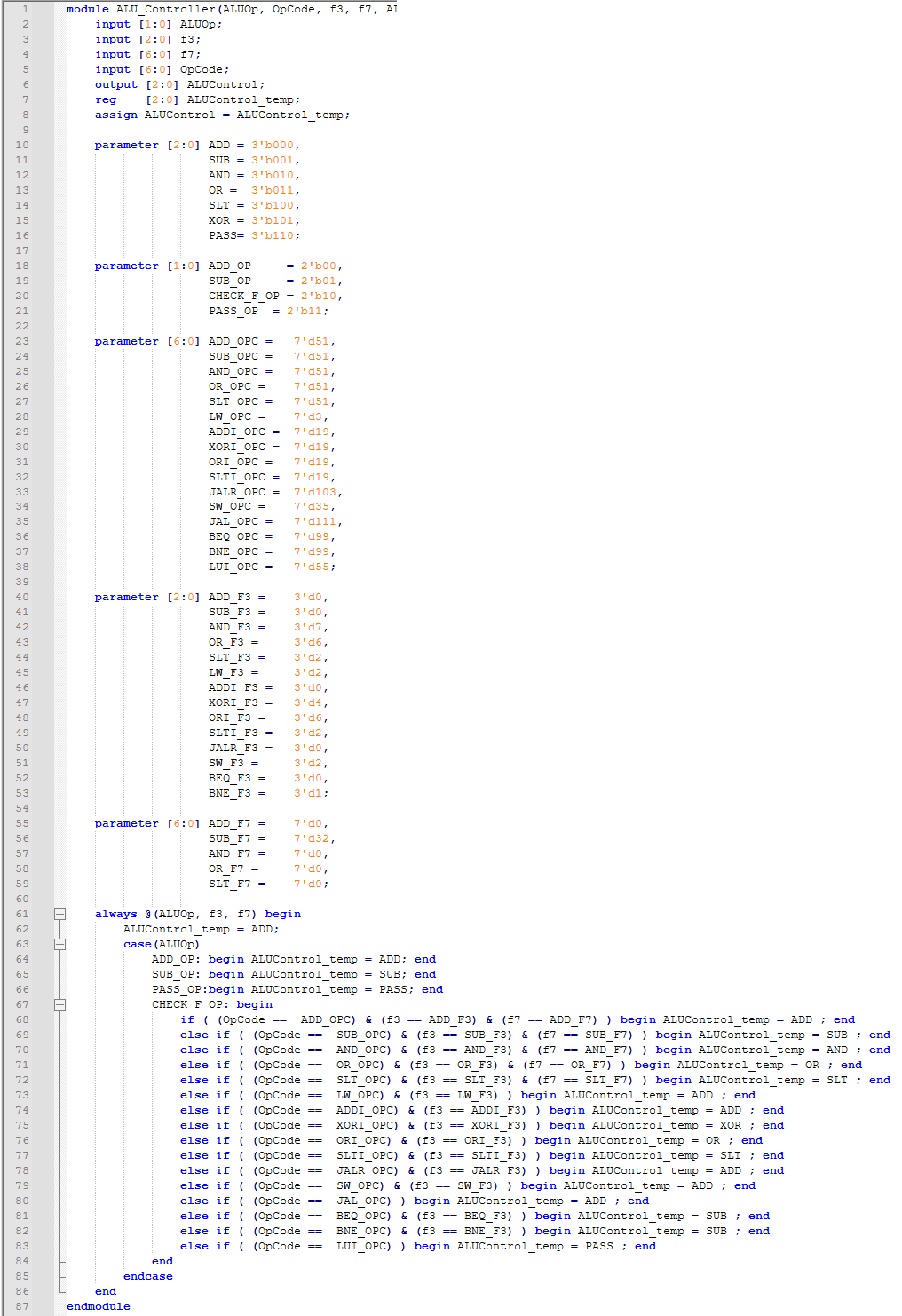
Table 14. ALU Controller Functionality Table

|  |  |  |
| --- | --- | --- |
| ALUOp | f3, f7 | ALU Operation |
| 00 |  | ADD |
| 01 |  | SUB |
| 10 | ADD | ADD |
| SUB | SUB |
| AND | AND |
| OR | OR |
| SLT | SLT |
| XOR | XOR |
| 11 | - | PASS |

Here’ ALUOp is the output of the Main Controller, it’s how it controls ALU Controller’s actions. ALU Operation is what the ALU does, and must be controlled by the ALU Controller.

The Verilog Description of the ALU Controller is shown in the next page.

Figure 3. ALU Controller Verilog Description



## PC Controller

There are three scenarios where the program counter needs to be updated:

* Going to the next instruction, located at the next memory word
* Jumping to a given address
* Branching from an instruction to somewhere else

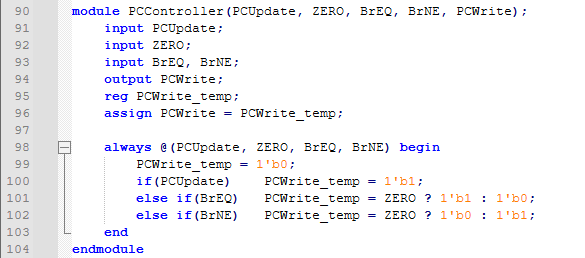
This begs the need for a hardware that governs the PC, but responds to the Main Control unit itself. Here’s a functional description for the PC controller:

Program counter gets updated if:

1. The address “PC+4” is calculated
2. The jumping address (for JAL, JAKR, etc.) is calculated
3. The current instruction is branch type, and the branch condition is met

Here’s the Verilog description of this hardware:

Figure 4. PC Controller Verilog Description



Signals “BrEQ”, “BrNE”, and “PCUpdate” come from the Main Control unit. They tell this hardware when to check the branch condition, when to jump directly, and when not to do anything. “PCWrite” is the module’s output, and gets connected to PC’s load-enable input.

## Main Control Unit

## Overall Description

So far, all we have done has been combinational. But, things change from this point on. We need to design a **Finite State Machine**, aka, **FSM**, to handle the execution of different instructions. This state machine must handle the flow of data through the datapath, by governing its control signals, it also must control the other two parts of the controller.

In order to design the controller, we need to have a step-by-step algorithm, fully describing what needs to happen in order for an instruction to execute. Here’s that algorithm.

Table 15. Main Control Unit Tabular Explanation

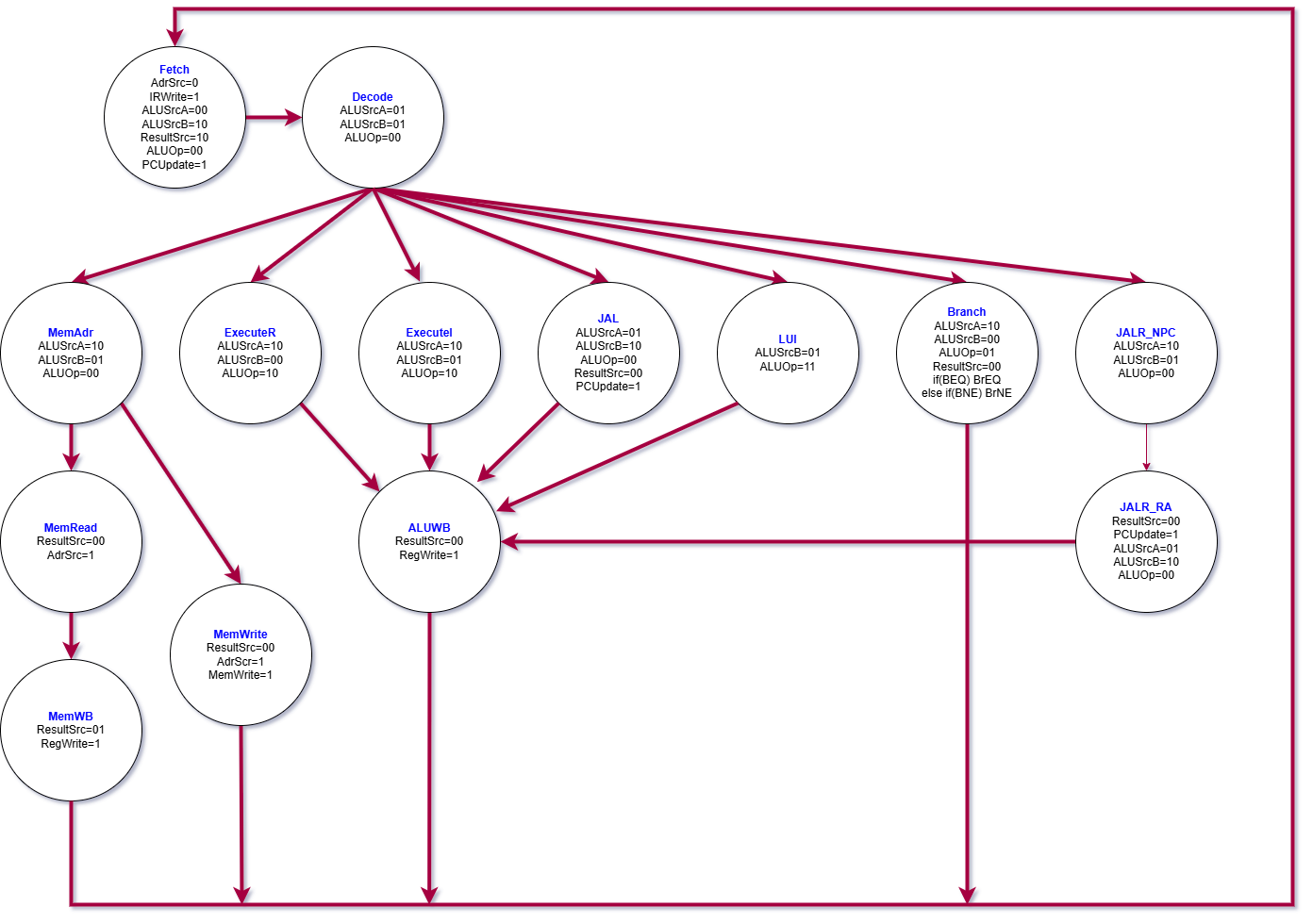
|  |  |
| --- | --- |
| R-Type | Fetch the instruction |
| Decode the instruction and get operands |
| Calculate the results |
| Write back the results |
| I-Type (LW and JALR excluded) | Fetch the instruction |
| Decode the instruction and get operands and the immediate value |
| Calculate the results |
| Write back the results |
| S-Type | Fetch the instruction |
| Decode the instruction and get operands |
| Calculate memory address |
| Store the operand in the memory |
| J-Type | Fetch the instruction |
| Decode the instruction and get operands |
| update PC and calculate return address |
| Write return address back to register file |
| B-Type | Fetch the instruction |
| Decode the instruction and get operands |
| update PC if branch condition is met |
| U-Type | Fetch the instruction |
| Decode the instruction and get operands |
| Calculate the immediate and “PASS” it through the ALU |
| Write the results back at register file |
| LW | Fetch the instruction |
| Decode the instruction and get operands |
| Calculate memory address |
| Read from the memory |
| Write the results back to the register file, from the memory path |
| JALR | Fetch the instruction |
| Decode the instruction and get operands |
| Calculate next PC |
| Update the PC and calculate return address |
| Write the return address back to register file |

Now it is easy to implement the FSM. Also, a quick glance at the table shows that some of these steps are common between different instructions, making it possible to share the states between different instructions!

## State Diagram

The state diagram for this controller looks like this:

Figure 5. Main Control Unit State Diagram



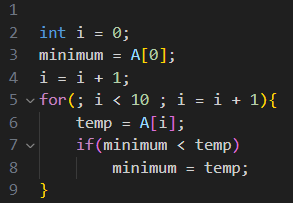
All there is left at this point, is to describe this in Verilog. Since the description was way too big (240 lines of code, give or take), we did not include it in the report.

# Testing and Verification

## Writing Assembly Code

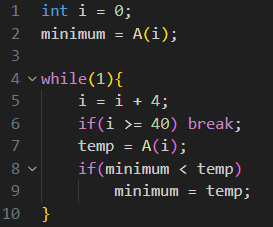
We were asked to test the processor we implemented with a piece of code that searches an array of 10 integers for its minimum value. Here’s how it’s done in C programming language:

Figure 6. Minimum Finder Code in C language



Since we are to implement this in assembly language afterwards, we could change a thing or two to make the conversion easier. Here’s the new code, written in a format between assembly and C:

Figure 7. Assembly-Friendly Minimum Finder Code

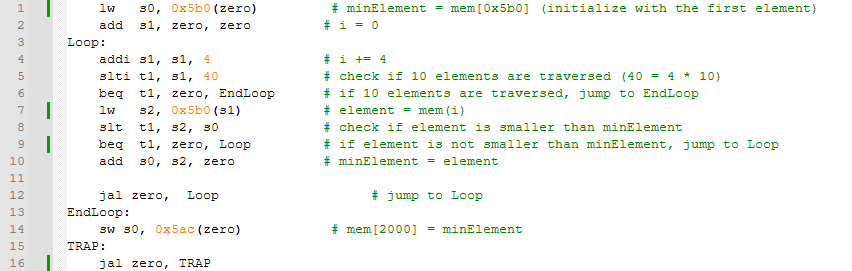


Now this can be easily turned into assembly code. We assumed that the array’s first element is located at mem[0x5b0], since that is what we did for this course’s second homework to make it compatible with the introduced website. Also, we chose our registers according to the table at the next page.

|  |  |  |
| --- | --- | --- |
| Register | Purpose | Register Number () |
| S2 | Reading data from memory (temp) | X18 |
| S1 | Saving i | X9 |
| S0 | Saving minimum | X8 |
| t1 | Saving SLT comparisons’ results | X6 |

Now we can easily write the assembly code for this.

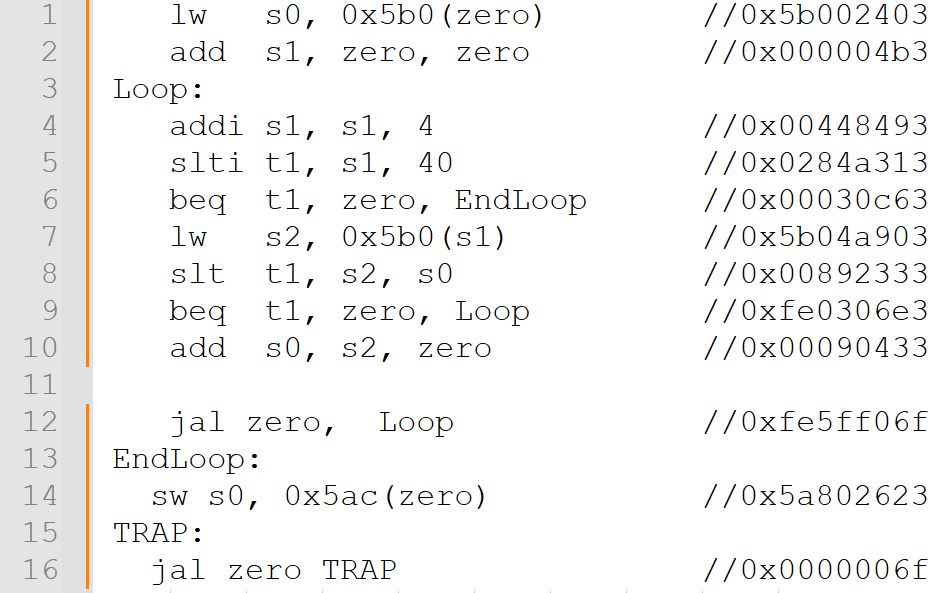
Figure 8. Assembly Code for Minimum Finder



All there is left to do is generate machine code from this and fill the memory with. Note that since the multi-cycle approach is built on hardware reuse and small combinational paths, we have to merge the data memory and the instructions memory, thus having a single memory. This leads to having one memory file, and we need to make sure the data is written in a place that has no overlap with the instructions.

## Generating RISC-V Machine Code

Figure 9. Assembly and Machine Code



The assembly codes are translated to RISC-V machine language in Hex (the related hex instruction of each line of the assembly code is shown in Figure 10).

To be written in the instruction memory, each hex instruction is first separated into four 2-digit hex numbers, and then written upside down (the 2 least significant hex digits first) in the instruction memory.

For example, the hexadecimal instruction 0x5ba40903 is written in the memory file like this:

03

09

A4

5B

This reversing is needed for the instructions to be properly read from the memory (2 most significant hex digits first).

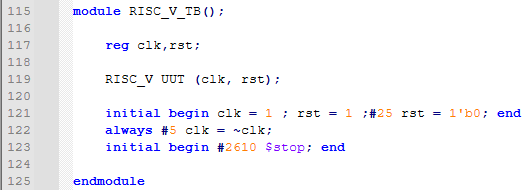
We also did this reversing for data to be stored in the data memory. The data we used is this array:

33, 114, 51, 62, -145, -127, 61, 84, 41, 15

## Testbench and Results

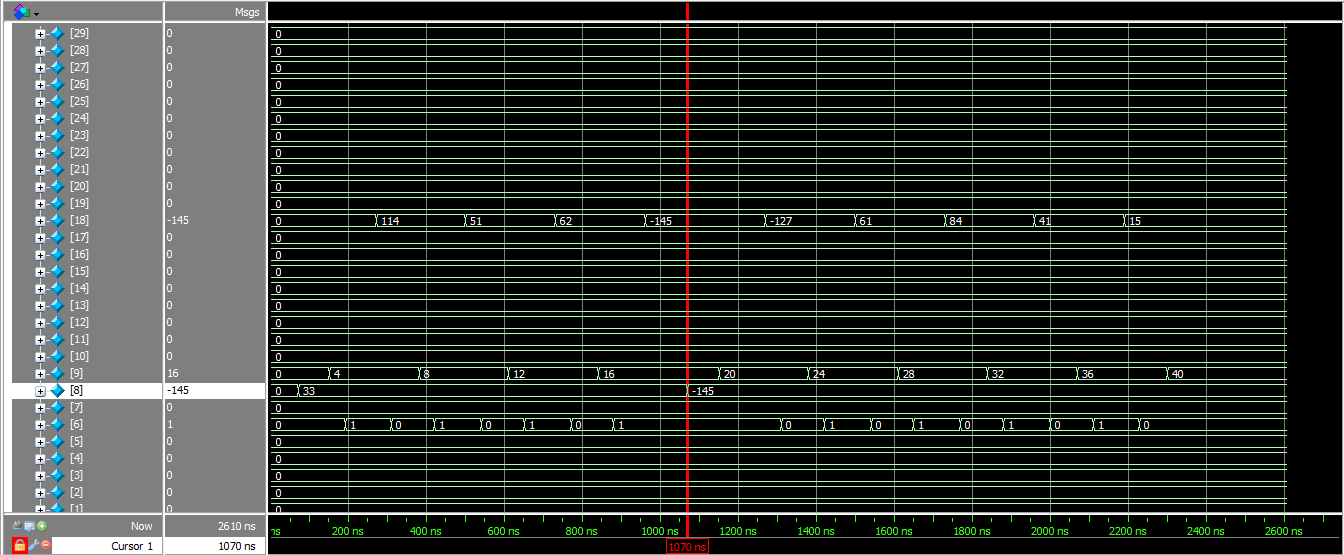
We used this testbench to test the code:

Figure 10. Top Module Testbench



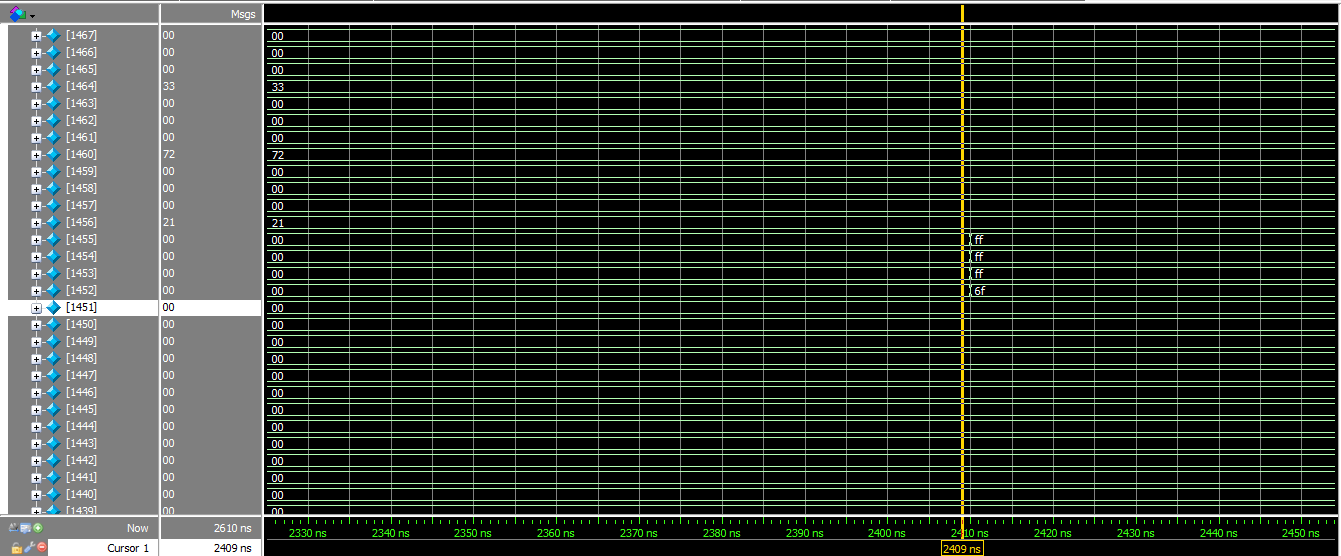
What is directly to be objected here, is the fact that we have waited a lot longer than before for the results to get ready. The answer is, we did not design these testbenches[[10]](#footnote-11) to be compared to each other. For the comparison to be possible, we need to synthesize both modules in the same technology boundaries, extract the delay values, and clock them both to their maximum allowed frequency and see which is faster. Anyway, the results of this testbench are represented in the next page.

Figure 11. Register File Waveforms



The value of the 8th register becomes -145, and stays the same for the rest of the testbench, since this is the minimum value.

Figure 12. Memory Waveforms



The results were also written on the memory, as can be seen in the assembly code. This waveform confirms that aspect as well.

THE END

1. Register Type [↑](#footnote-ref-1)
2. Immediate Type [↑](#footnote-ref-2)
3. Store Type [↑](#footnote-ref-3)
4. Jump Type [↑](#footnote-ref-4)
5. Branch Type [↑](#footnote-ref-5)
6. Upper Type [↑](#footnote-ref-6)
7. Arithmetic-Logic Unit (ALU): The part that handles arithmetic and logical calculations. [↑](#footnote-ref-7)
8. Immediate [↑](#footnote-ref-8)
9. The program counter is also a simple register in theory, but we decided to give it a separate identity due to its importance. [↑](#footnote-ref-10)
10. This testbench, and the one for single-cycle processor of the previous computer assignment [↑](#footnote-ref-11)