

Computer Assignment 1

November 23rd, 2024

Single Cycle Approach

RISC-V Processor

Bardia Amirian | 810101605

Amirreza Nadi Chaghadari | 810101538

Digital Systems 2

Dr. Safari

**Table of Contents**

[Problem Description 1](#_Toc183900404)

[Instruction Formats 2](#_Toc183900405)

[R-Type Instructions 2](#_Toc183900406)

[Instruction format 2](#_Toc183900407)

[Instruction Execution Table 2](#_Toc183900408)

[I-Type Instructions 3](#_Toc183900409)

[Instruction format 3](#_Toc183900410)

[Instruction Execution Table 3](#_Toc183900411)

[S-Type Instructions 4](#_Toc183900412)

[Instruction format 4](#_Toc183900413)

[Instruction Execution Table 4](#_Toc183900414)

[J-Type Instructions 5](#_Toc183900415)

[Instruction format 5](#_Toc183900416)

[Instruction Execution Table 5](#_Toc183900417)

[B-Type Instructions 6](#_Toc183900418)

[Instruction format 6](#_Toc183900419)

[Instruction Execution Table 6](#_Toc183900420)

[U-Type Instructions 7](#_Toc183900421)

[Instruction format 7](#_Toc183900422)

[Instruction Execution Table 7](#_Toc183900423)

[Datapath Design 8](#_Toc183900424)

[Abstract Block Diagram 8](#_Toc183900425)

[Designing Components 10](#_Toc183900426)

[MUX 10](#_Toc183900427)

[Adder 11](#_Toc183900428)

[ALU 12](#_Toc183900429)

[Immediate Extension 15](#_Toc183900430)

**Table of Contents**

[PC (Program counter) 16](#_Toc183900431)

[Instruction Memory 17](#_Toc183900432)

[Data Memory 18](#_Toc183900433)

[Register File 19](#_Toc183900434)

[Assembling Datapath 20](#_Toc183900435)

[Controller 21](#_Toc183900436)

[Top Module 22](#_Toc183900437)

[Testing Phase 23](#_Toc183900438)

[Writing Assembly Code 23](#_Toc183900439)

[Generating RISC-V Machine Code 25](#_Toc183900440)

[Testbench and Results 27](#_Toc183900441)

**Table of Figures**

[Figure 1. Datapath Block Diagram 8](#_Toc183900442)

[Figure 2. MUX Verilog Description 10](#_Toc183900443)

[Figure 3. MUX Testbench Results 10](#_Toc183900444)

[Figure 4. Adder Verilog Description 11](#_Toc183900445)

[Figure 5. Adder Testbench Results 11](#_Toc183900446)

[Figure 6. ALU Verilog Description 12](#_Toc183900447)

[Figure 7. ALU Testbench 13](#_Toc183900448)

[Figure 8. ALU Testbench Results 14](#_Toc183900449)

[Figure 9. Immediate Extension Verilog 15](#_Toc183900450)

[Figure 10. Program Counter Verilog Description 16](#_Toc183900451)

[Figure 11. Instruction Memory Verilog Description 17](#_Toc183900452)

[Figure 12. Data Memory Verilog 18](#_Toc183900453)

[Figure 13. Register File 19](#_Toc183900454)

[Figure 14. Datapath Module Verilog Description 20](#_Toc183900455)

[Figure 15. Controller: Result Source Handling Always Block 21](#_Toc183900456)

[Figure 16. Top Module Verilog Description 22](#_Toc183900457)

[Figure 17. Minimum Finder Code in C language 23](#_Toc183900458)

[Figure 18. Assembly-Friendly Minimum Finder Code 23](#_Toc183900459)

[Figure 19. Assembly Code for Minimum Finder 24](#_Toc183900460)

[Figure 20. Assembly and Machine Code 25](#_Toc183900461)

[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 21). 25](#_Toc183900462)

[Figure 22. Top Module Testbench 27](#_Toc183900463)

[Figure 23. Testbench Results 27](#_Toc183900464)

**Table of Tables**

[Table 1. T-Type Instruction Format 2](#_Toc183900465)

[Table 2. R-Type Instruction Execution Table 2](#_Toc183900466)

[Table 3. I-Type Instruction Format 3](#_Toc183900467)

[Table 4. I-Type Instruction Execution Table 3](#_Toc183900468)

[Table 5. S-Type Instruction Format 4](#_Toc183900469)

[Table 6. S-Type Instruction Execution Table 4](#_Toc183900470)

[Table 7. J-Type Instruction Format 5](#_Toc183900471)

[Table 8. J-Type Instruction Execution Table 5](#_Toc183900472)

[Table 9. B-Type Instruction Format 6](#_Toc183900473)

[Table 10. B-Type Instruction Execution Table 6](#_Toc183900474)

[Table 11. U-Type Instruction format 7](#_Toc183900475)

[Table 12. U-Type Instruction Execution Table 7](#_Toc183900476)

[Table 13. ALU Op-Code Table 12](#_Toc183900477)

[Table 14. Immediate Formatting 15](#_Toc183900478)

# Problem Description

We are to design a simplified version of the RISC-V processor, using   
single-cycle approach. It means that each instruction takes exactly one clock cycle to execute. Here are 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) |

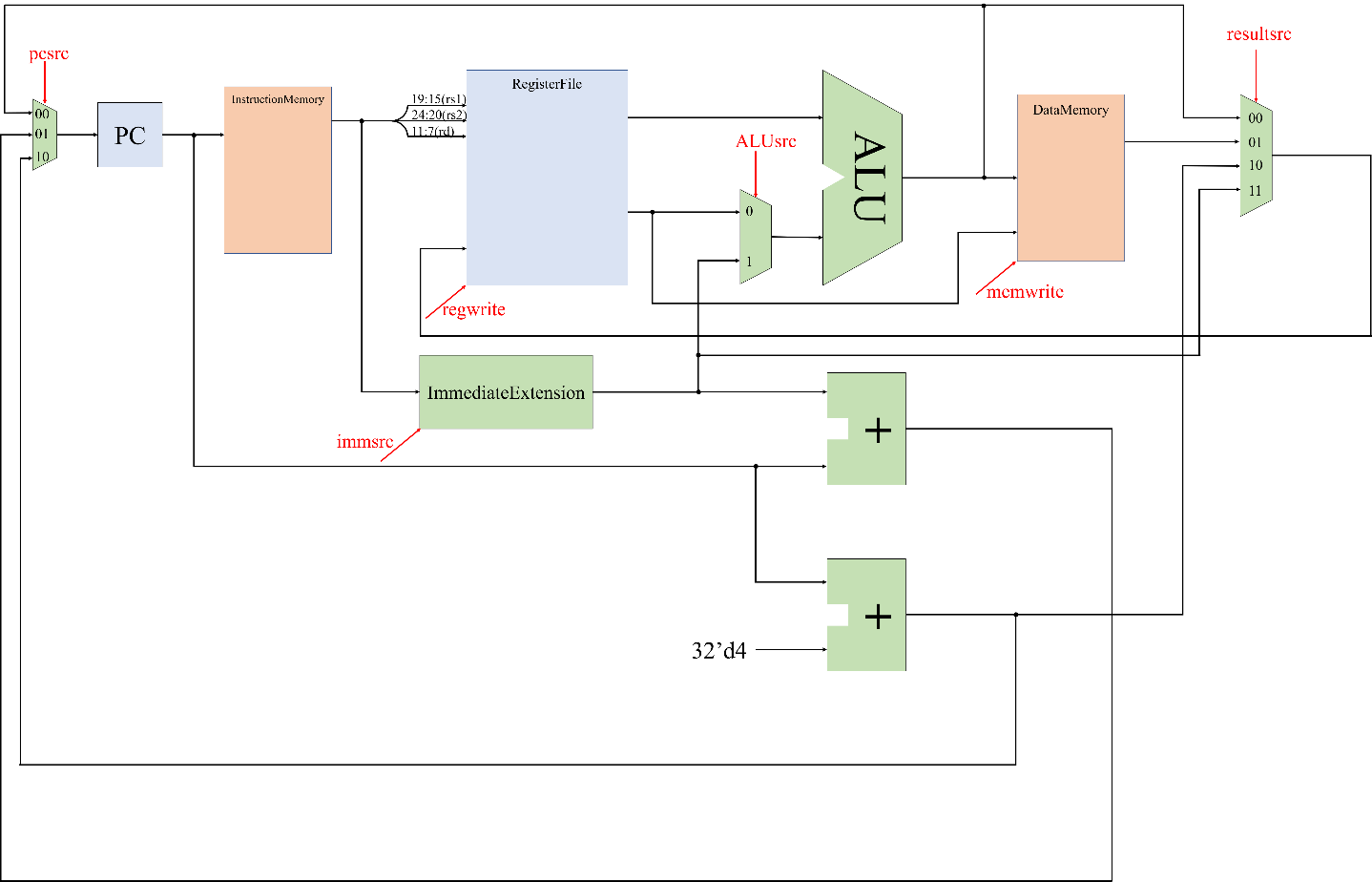
With these tables at hand, we can design the datapath.

# Datapath Design

## Abstract Block Diagram

The block diagram of the datapath looks like this:

Figure 1. Datapath Block Diagram



Where red signals come from the controller, edged connections (ones that are marked by a dot) mean the entire wire array is given to both branches, and rounded branches (at the input of the Register File) mean that the wire array is split into parts.

Here, the combination of “PC” and “Instruction Memory” handles the flow of the code being executed on the process. Instruction Memory’s output is connected to the “Register File” to address the source and destination registers. It is also given to the Immediate Extension block in order to extract immediate values out of it. The ALU has 2 inputs. One comes from the Register File itself, and the other is the output if a MUX choosing between another register or the immediate value. This allows the execution of R-Type and I-Type instructions. By connecting Register File’s second output to the input of “Data Memory”, the datapath becomes capable of executing “sw” instruction. Also, the output of the memory id connected to the input of the Register File, making “lw” instruction’s execution possible. Considering the fact that the ALU has a “ZERO” flag, B-Type instructions are easily handled by subtracting the operands and using the ZERO flag to set PC’s next value. Also, J-Type instructions are easily done because of the two extra adders. As for the “lui” instruction, we can direct Immediate Extension’s output to the Register File’s input. Therefore, all of the given instructions can be executed using this datapath.

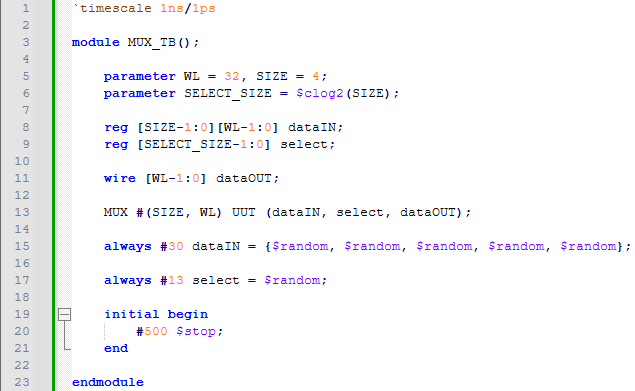
Next step is to design and implement each component used in the datapath, instantiate them, and assemble the datapath by wiring the instances together.

## Designing Components

### MUX[[9]](#footnote-10)

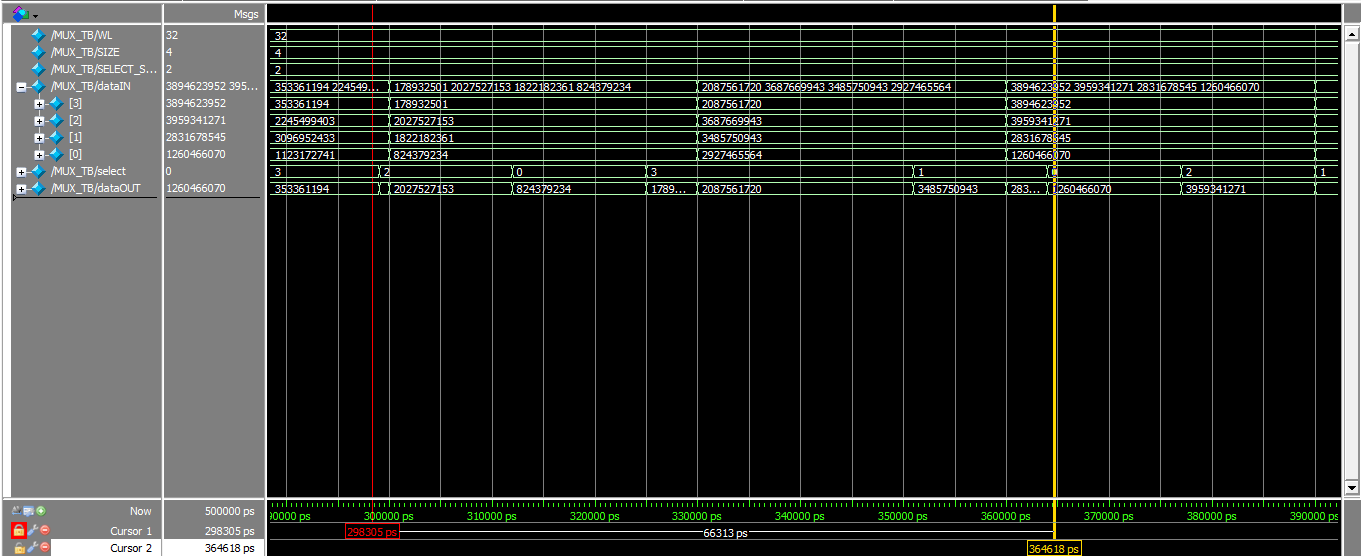
Multiplexer is one of the easiest purely-combinational components we are to implement. The Verilog description of it looks like this:

Figure 2. MUX Verilog Description



We also designed a simple testbench to test this MUX with random data, and got these results:

Figure 3. MUX Testbench Results

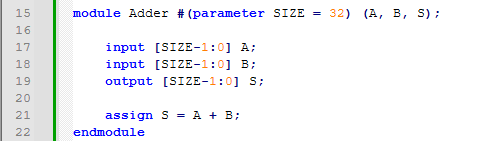


This waveform (the part between the two cursors) represents different things that could happen to the inputs, and the MUX’s response to them, verifying the functionality of the MUX.

### Adder

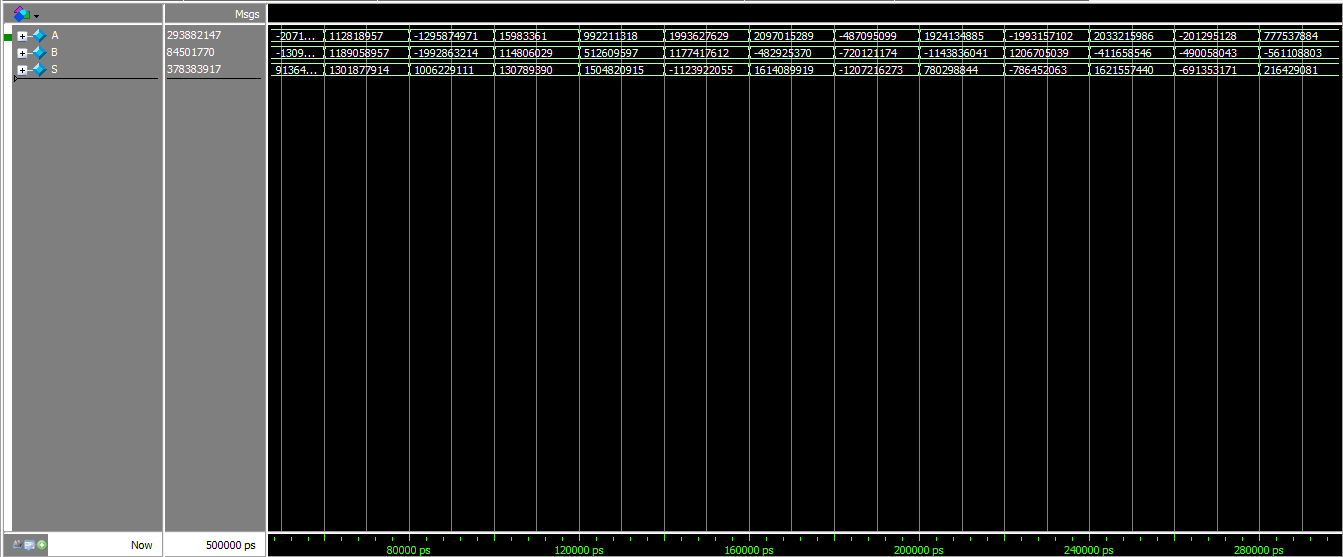
Adder is also simple to implement. Here’s the Verilog description:

Figure 4. Adder Verilog Description



As for the testing, this waveform confirms its functionality:

Figure 5. Adder Testbench Results



### ALU

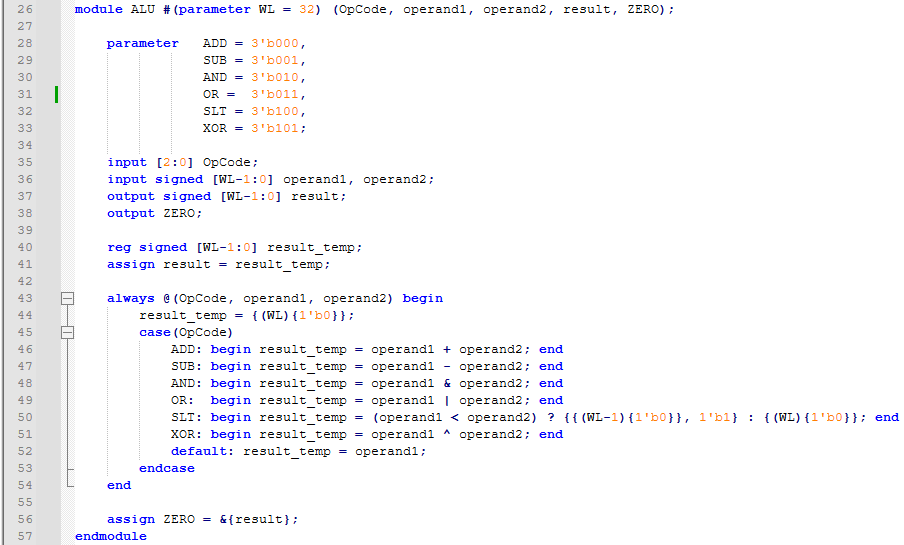
ALU can be simply implemented using an always block. But, we need to have a clear operation table first. Here’s the one we used:

Table 13. ALU Op-Code Table

|  |  |
| --- | --- |
| Op-Code[[10]](#footnote-11) | Operation |
| 000 | Add |
| 001 | Subtract |
| 010 | AND |
| 011 | OR |
| 100 | SLT |
| 101 | XOR |

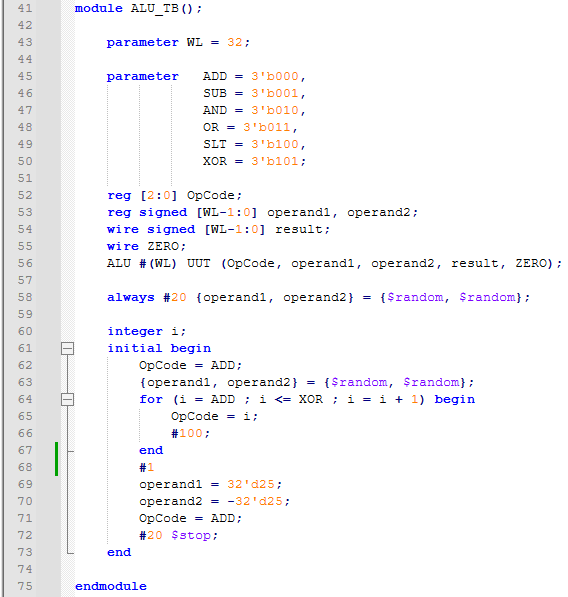
Now we can describe it in Verilog. Here’s the description:

Figure 6. ALU Verilog Description



The inputs and output are defined as signed values, so that “slt” operation becomes possible. As for functionality confirmation, we used this testbench to test its response to different op-codes:

Figure 7. ALU Testbench



The results are shown in the next page.

Figure 8. ALU Testbench Results



### Immediate Extension

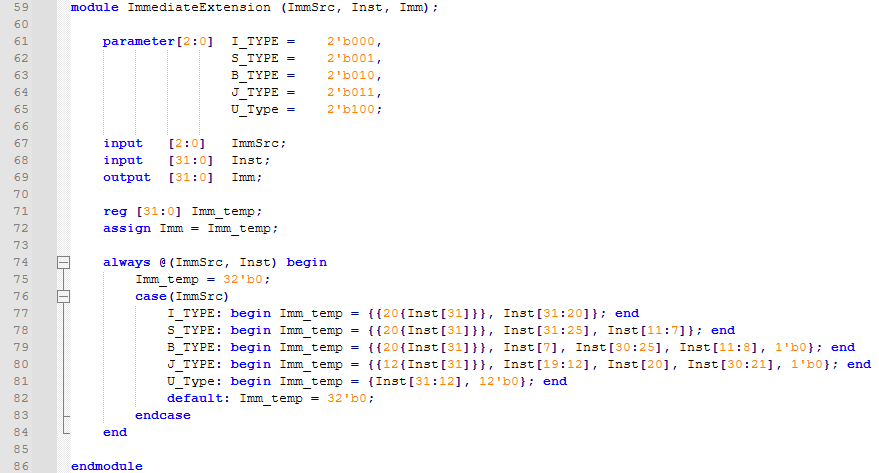
This part is nothing but wiring and multiplexing, and is done due to this table:

Table 14. Immediate Formatting

|  |  |  |  |
| --- | --- | --- | --- |
| immsrc | Type | Size | Format |
| 000 | I-Type | 12 | {{20{Instr[31]}}, Instr[31:20]} |
| 001 | S-Type | 12 | {{20{Instr[31]}}, Instr[31:25], Instr[11:7]} |
| 010 | B-Type | 13 | {{20{Instr[31]}}, Instr[7], Instr[30:25], Instr[11:8], 1’b0} |
| 011 | J-Type | 21 | {{12{Instr[31]}}, Instr[19:12], Instr[20], Instr[30:21], 1’b0} |
| 100 | U-Type | 21 | {Instr[31:12], 12’b0} |

Where “Instr” stands for instruction. The rest is just a matter of writing this in Verilog.

Figure 9. Immediate Extension Verilog

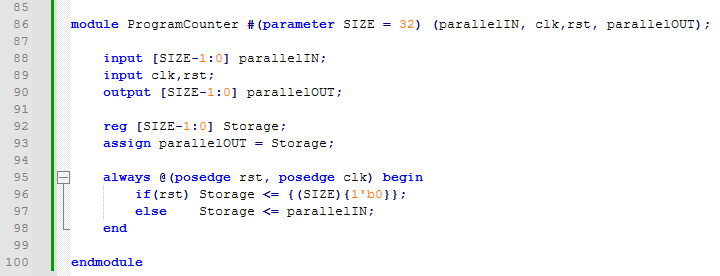


With this, all combinational components are ready. The rest of the components are sequential. But due to their simplicity[[11]](#footnote-12), we did not use “Huffman Coding Style” to implement them.

### PC (Program counter)

PC is just a simple register. Here’s how it’s described in Verilog:

Figure 10. Program Counter Verilog Description

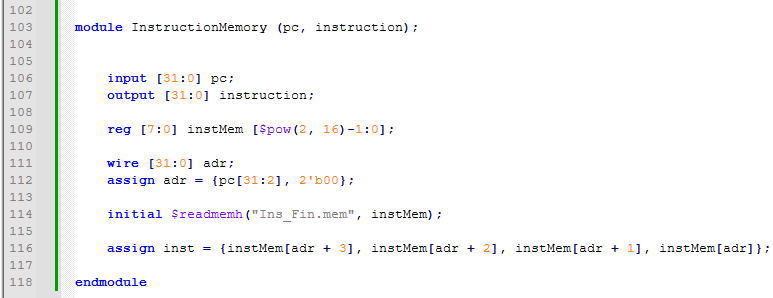


The rest of the components are memory-type ones. For better test-automation, we decided to use some extra files to store some pre-determined data in them.

### Instruction Memory

We decided to save the instructions inside a file and read the file upon simulation. Here’s how it’s done:

Figure 11. Instruction Memory Verilog Description

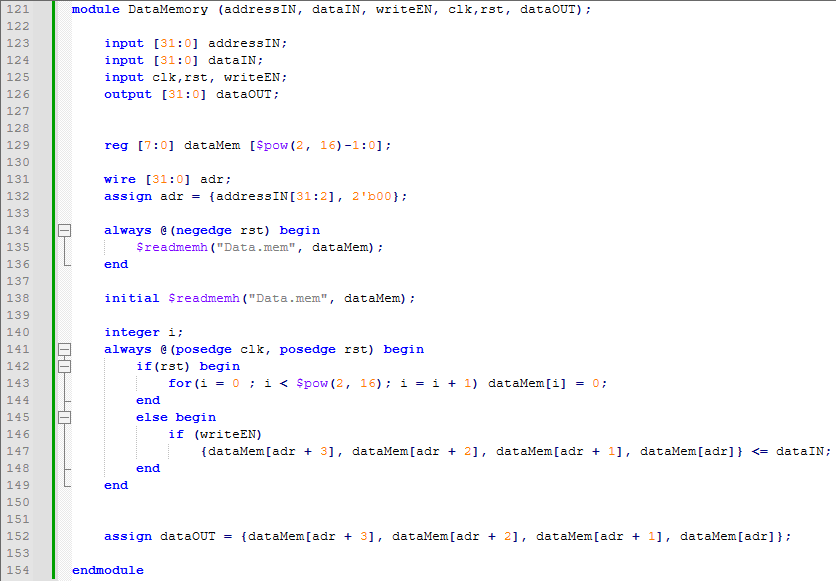


Thankfully, Verilog has the “$readmemh” function and we do not need to read the file line-by-line manually. The file must have “.mem” postfix, and each line of it must contain 2 hexadecimal digits.

### Data Memory

This memory needs to be loaded as well, but it also needs initialization to get the data we aim to sort. Therefore, we decided to do the initialization upon the negative edge of the restart signal. Here’s how the Verilog description looks like:

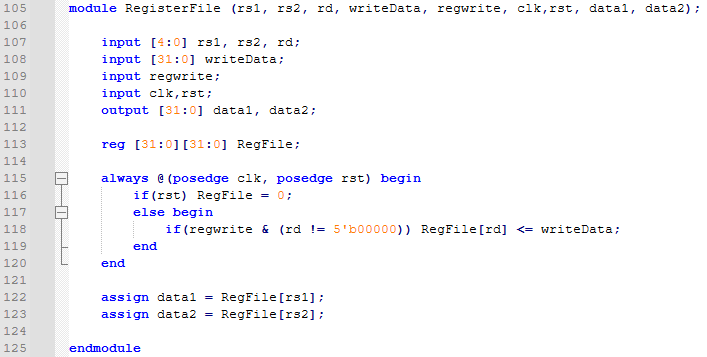
Figure 12. Data Memory Verilog



### Register File

This part is easy, since it is just a simple double-output single-input register array. Here’s the Verilog code:

Figure 13. Register File



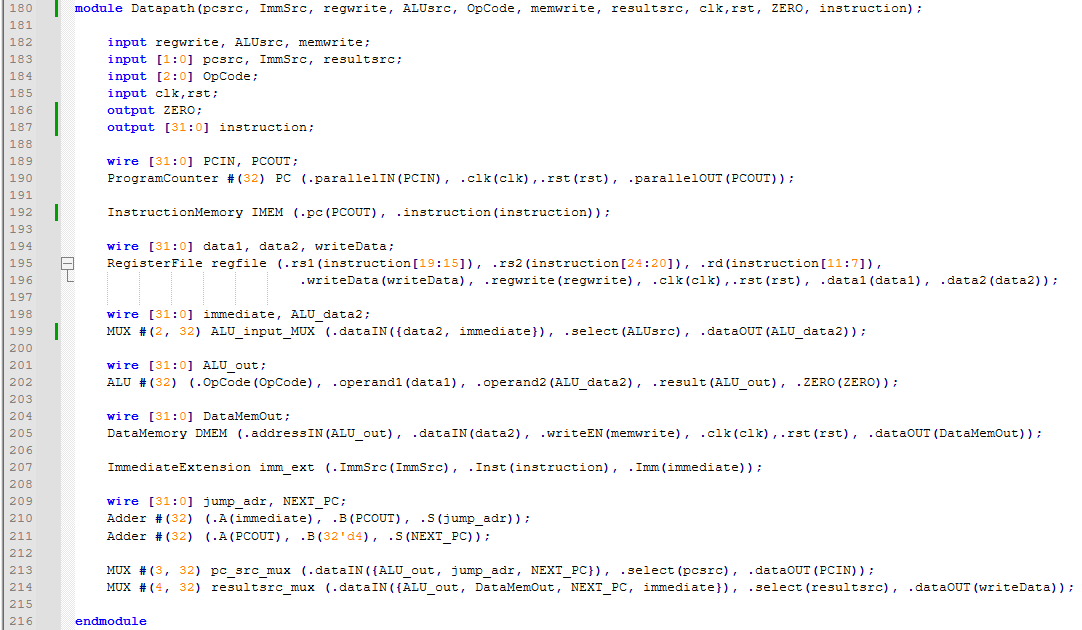
What is important here, is line 118. Writing in “ZERO” register is not allowed. It must always remain 0. The rest is just a matter of multiplexing.

With this, all the components are ready, allowing us to proceed to the next phase.

## Assembling Datapath

With all the components ready at hand, we can assemble the datapath by instantiating the components and wiring them together. Here’s how the Verilog code looks like:

Figure 14. Datapath Module Verilog Description

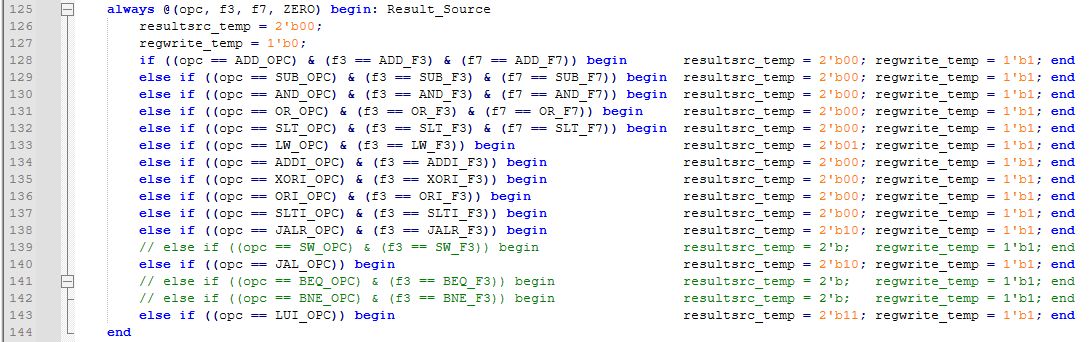


Now we need to design the controller. Since we are using single-cycle approach, there is no need for a state machine. All we need is a combinational logic.

# Controller

The controller is just a simple lookup table. The rest is just a matter of using “always” blocks and “assign” statements to model the lookup tables. The code was way too long to be included here (170 lines in total). Therefore, we only included snapshots of one of the always blocks, handling “resultsrc” and “regwrite”.

Figure 15. Controller: Result Source Handling Always Block

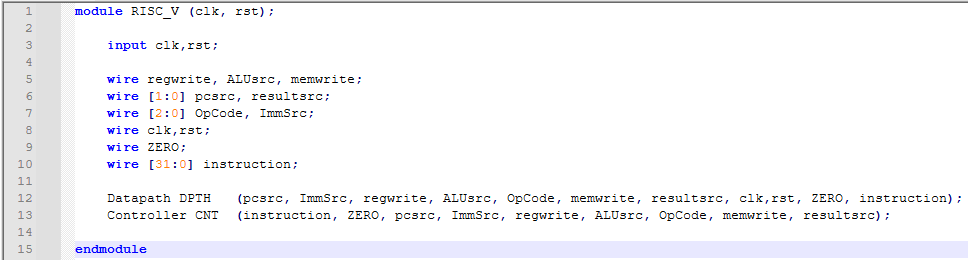


Now we can generate the top module and test it.

# Top Module

Here’s how the top module looks like:

Figure 16. Top Module Verilog Description



The only inputs of the top module are clock and reset signals. Since control unit is implemented in a purely combinational manner, it does not need clock or reset signals.

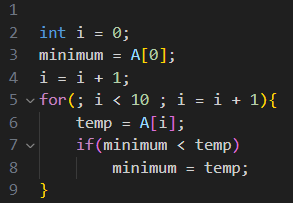
With the top module ready at hand, we could proceed to the testing phase.

# Testing Phase

## Writing Assembly Code

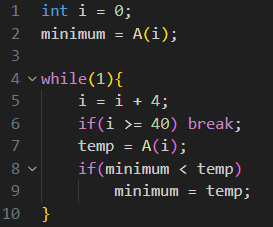
We were asked to test the processor we implemented with a piece of code that searches an array of 10 integers for its minimum value. Here’s how its done in C programming language:

Figure 17. 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 18. Assembly-Friendly Minimum Finder Code

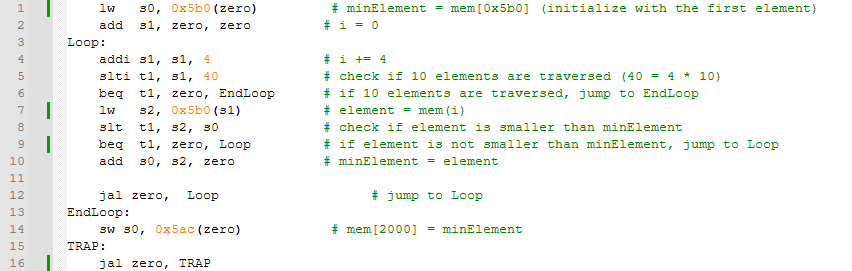


Now this can be easily turned into assembly code. We assumed that the array’s first element is located at mem[0x5b0], since that is what we did for this course’s second homework to make it compatible with the introduced website. Also, we chose our registers according to the table at the next page.

|  |  |  |
| --- | --- | --- |
| 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.

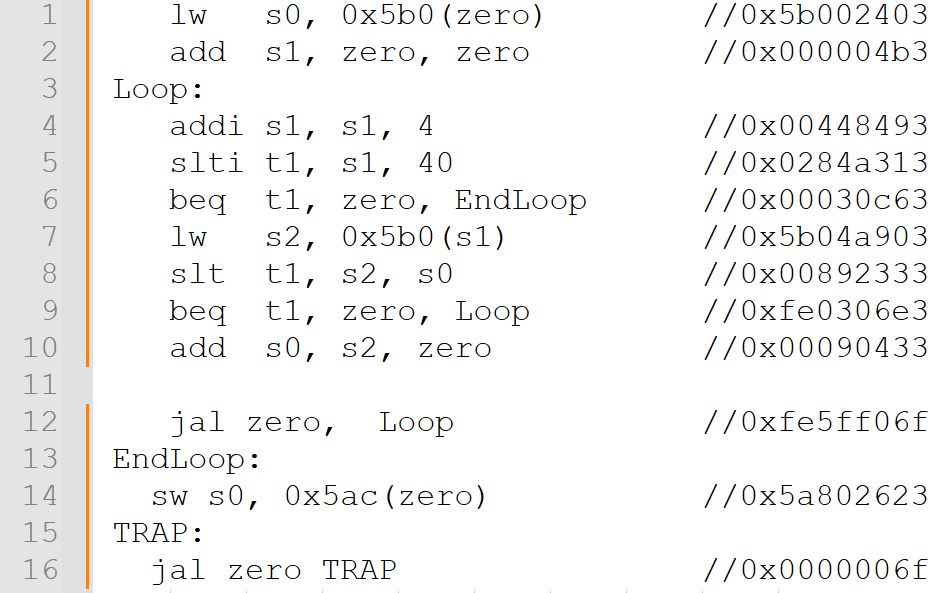
Figure 19. Assembly Code for Minimum Finder



All there is left to do is generate machine code from this and fill data and instruction memories.

## Generating RISC-V Machine Code

Figure 20. 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 21).

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 hex instruction 0x5ba40903 is written in the memory 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).

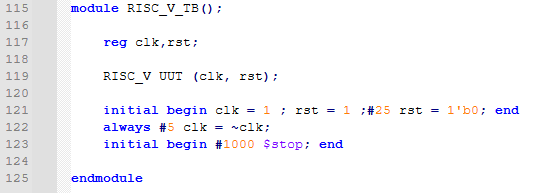
We also did this reversing for data to be stored in the data memory. The data we used is this array:

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

# Testbench and Results

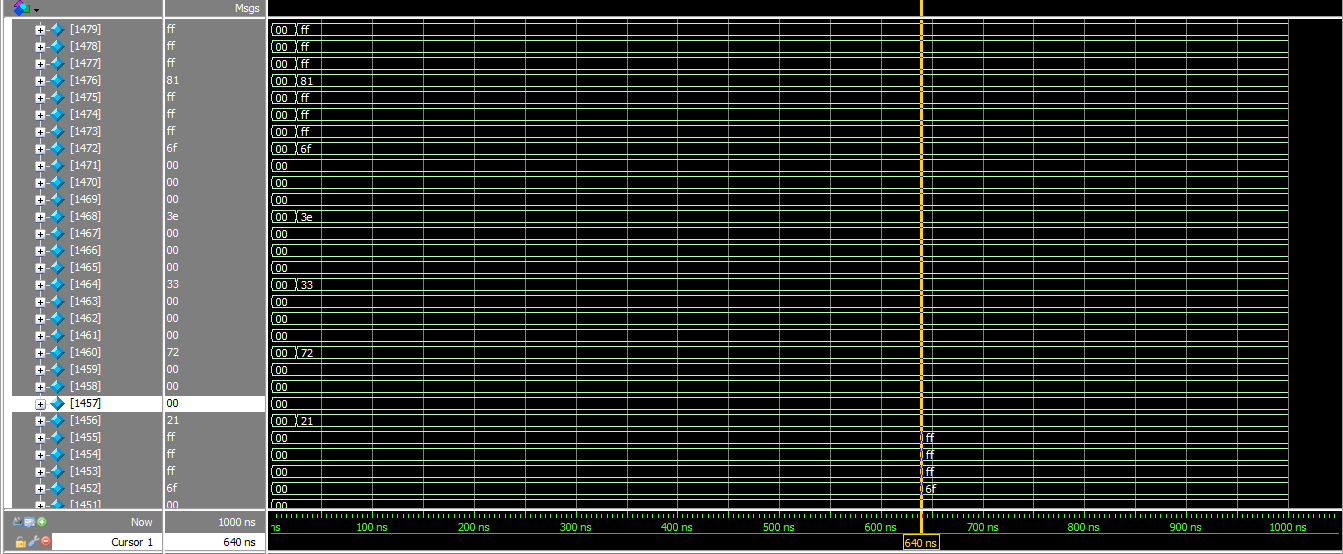
We used this testbench to test the code:

Figure 22. Top Module Testbench



And the memory was updated like this:

Figure 23. Testbench Results



This confirms the minimum finding functionality.

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. Multiplexer [↑](#footnote-ref-10)
10. Operation Code [↑](#footnote-ref-11)
11. All of them are simple storage units, making them easy to describe in a single always block. [↑](#footnote-ref-12)