King Abdul-Aziz University

Faculty of Engineering

# Final project:
# Designing A Full Data Path

Summer 2021

EE361

Section: EA

**Team Members**

| # | Name: | ID |
|---|---|---|
| 1 | Abdulaziz Hamid Ebrahim | 1946282 |
| 2 | Osamah Badughaish | 1855398 |
| 3 | Tamam Ahmed Alahdal | 1851778 |
| 4 | Moustafa Ahmed | 1846760 |

**Instructor Name:**

Dr. Saud Al-wasly

# Table of Contents

## Introduction

A computer consists of many parts and the CPU is one of its essential parts, in this course we studied the CPU in a single cycle and in a pipeline, therefore now is the time to apply our understanding to real work and build single cycle CPU data path. Data path is the components that build up the CPU, therefore we need build each component alone and then combine them based on the ISA we chose to impalements. in this report we will show the implementation of each component alone and after that we will show the full data path. Data path components are:

1. A Program Counter register (PC)
2. A proper decoder to decode raw instructions into relevant operands and control signals.
3. A Register File made of all 32 general-purpose registers from X0 to X31.
4. An Arithmetic and Logic Unit (ALU) that can execute the arithmetic and logic operations in RVM32IM ISA.
5. One or more control units to coordinate the operation of the different components.
6. Instruction and Data memories to store the instructions (program) and data.
7. Other required logic to deal with branches and jumps.
8. A top-level block to instantiate, assemble, and connect all components together as a full design.

By the end of doing the 9 requirements we will learn making these components using myHDL and then converting them to Verilog, moreover, is to learn about the data path implementation, indeed this will teach us how a CPU is operating and how we can choose or CPU for our work in real life.

## i. PC

When we turn on the CPU it starts generating clocks to the PC, at each rising edge the pc passES the input to the output, it’s a block that has 32 flipflops sets in parallel, therefore PC’s size is 32-bit, PC has 32 bit input therefore 32 bit output, these inputs are coming from adder or branch adder to give the instruction memory the next address in the next cycle and also goes to the adder and the branch adder to set the address of the next instruction.

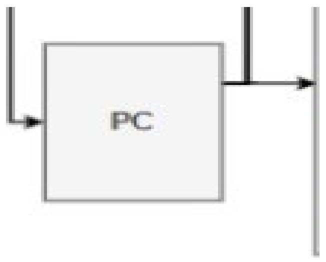


*Figure 1 PC*

## Test Bench

After writing the main block for the Program Counter, we needed to test our main Block, so we inserted random values for the input. Define the block then initialize the input and the output as signals since they will enter to the module and set # of bit to each signal. After 10 units invert the clock signal than change the state of the input signal to monitor the changes.

```
PCInput                    PCOutput

 0x77a575c2                0x77a575c2

 0x186232a4                0x186232a4

 0xf223c768                0xf223c768

 0xe048886a                0xe048886a
```

*Figure 2 output of PC*

## ii. Decoder

The instruction decoder is one of the most essential components in the CPU. This instruction decoder is designed only for RISC-V instructions. Its purpose is to decode or translate the instructions that comes from the instruction memory to other CPU components that can understand. It gives the control unit its essential information to behave properly, and it also act as an immediate generation. The instruction decoder can decode all the 6 important instruction formats. The instruction decoder, I.D., gives 7 outputs: immediate, rs1, rs2, rd, funct3, funct7, opcode. It takes one main parameter that is 32-bit instructions.

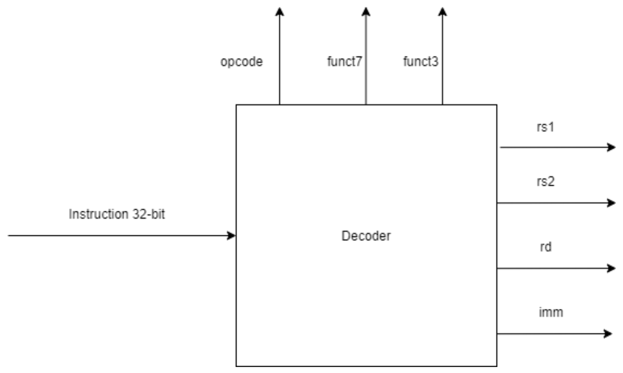
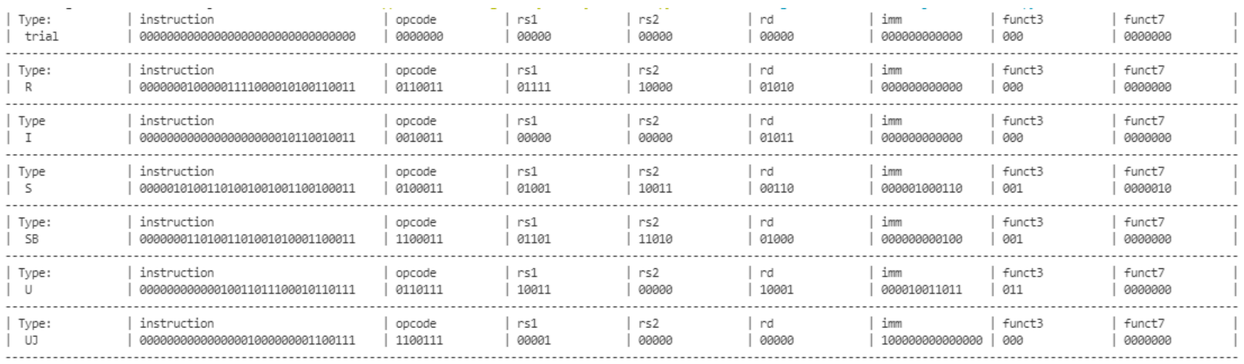


*Figure 3 Decoder block*

## Test Bench

The instruction decoder gets its instruction from the instruction memory and then it will split the instruction into the fields in respect to the instruction format. As showing in the figure below.



```
| Type:          | instruction                       | opcode         | rs1            | rs2            | rd             | imm            | funct3         | funct7         |
|  trial         | 00000000000000000000000000000000  | 0000000        | 00000          | 00000          | 00000          | 000000000000   | 000            | 0000000        |
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
| Type:          | instruction                       | opcode         | rs1            | rs2            | rd             | imm            | funct3         | funct7         |
|  R             | 00000001000001111000010100110011  | 0110011        | 01111          | 10000          | 01010          | 000000000000   | 000            | 0000000        |
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
| Type           | instruction                       | opcode         | rs1            | rs2            | rd             | imm            | funct3         | funct7         |
|  I             | 00000000000000000000010110010011  | 0010011        | 00000          | 00000          | 01011          | 000000000000   | 000            | 0000000        |
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
| Type           | instruction                       | opcode         | rs1            | rs2            | rd             | imm            | funct3         | funct7         |
|  S             | 00000101001101001001001100100011  | 0100011        | 01001          | 10011          | 00110          | 000001000110   | 001            | 0000010        |
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
| Type:          | instruction                       | opcode         | rs1            | rs2            | rd             | imm            | funct3         | funct7         |
|  SB            | 00000001101001101001010001100011  | 1100011        | 01101          | 11010          | 01000          | 000000000100   | 001            | 0000000        |
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
| Type:          | instruction                       | opcode         | rs1            | rs2            | rd             | imm            | funct3         | funct7         |
|  U             | 00000000000010011011100010110111  | 0110111        | 10011          | 00000          | 10001          | 000010011011   | 011            | 0000000        |
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
| Type:          | instruction                       | opcode         | rs1            | rs2            | rd             | imm            | funct3         | funct7         |
|  UJ            | 00000000000000001000000001100111  | 1100111        | 00001          | 00000          | 00000          | 10000000000000 | 000            | 0000000        |
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

*Figure 4 Decoder Testbench's output*

## iii. Register File

After decoding the instruction in the decoder block, the decoder will insert a certain instruction to the next block which is a Register file. It is a circuit that contains 32 register addresses from x0 - x31 and each register is 32-bits. Register file has 5 inputs three of them coming from the decoder and they are, "**rs1**, **rs2**, **rd**", and the fourth one coming from the control unit which is "**regWrite**", it will allow the register file to write in the registers or not. The fifth input is coming from the ALU result or from **dataMemory** which will explain furtherly later. Register file will output rs1 and rs2 to the next blocks. The figure below represents the Register File Block.

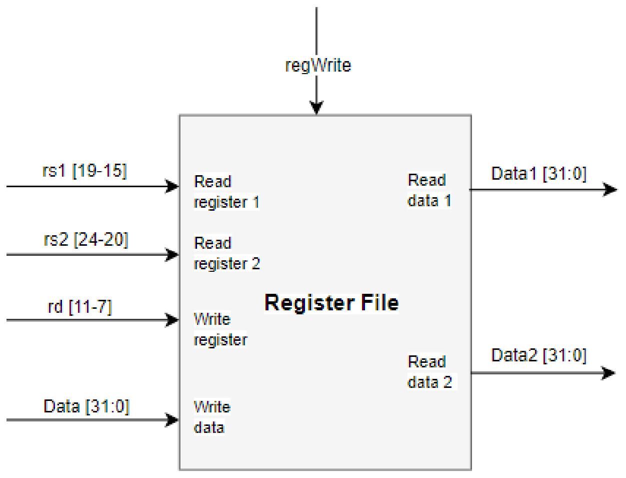
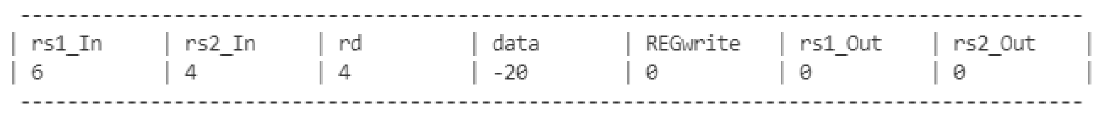


*Figure 5 Register File Block*

## Test Bench

After writing the main block for the register file, we needed to test our main Block, so we inserted random values of the inputs rs1_in, rs2_in, rd, and data. We made the register print what inside the registers two times one when the regWrite was off and the one when it's in on mode to test if the register will write on the register or not and if regWrite option is working fine or not. In the following figures, you will find two outputs the first one is before we enabled the regWrite option and the second one is after.

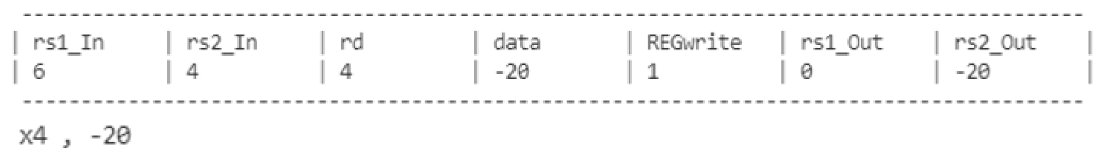
Register write-in is **disabled**:



```
 -------------------------------------------------------------------------------------
| rs1_In      | rs2_In      | rd          | data        | REGwrite    | rs1_Out     | rs2_Out     |
| 6           | 4           | 4           | -20         | 0           | 0           | 0           |
 -------------------------------------------------------------------------------------
```

*Figure 6 Register File Testbench's output*

Register write-in is **enabled**:



```
 -------------------------------------------------------------------------------------
| rs1_In      | rs2_In      | rd          | data        | REGwrite    | rs1_Out     | rs2_Out     |
| 6           | 4           | 4           | -20         | 1           | 0           | -20         |
 -------------------------------------------------------------------------------------
 x4 , -20
```

*Figure 7 Register File Testbench's output*

## iv. Arithmetic and Logic Unit (ALU)

Any CPU has its own ALU circuit performing a set of arithmetical operations such as addition, subtraction, multiplication, and logical operations such as AND, OR, and XOR. The ALU is an essential component of the CPU. In real-world conditions, the more complicated the procedure is, the more expensive the ALU will be and the space in the CPU will increase. It is intended for calculating many operations. CPU can be costlier or consume more power, because of ALU'S. ALU loads data and the control-unit specifies the ALU to operate on the data based on control unit's specification and the result is placed in the data memory or feedback in the relevant register.

In our CPU we have an ALU that comes after the "register file". ALU receive three inputs two of them are coming from multiplexers for each, and the third input is a flag input "ALUop" coming from "Control unit" to specify which operation should be implemented on the given data. It has two outputs one is the ALU operated result and the other one is just a flag Zero for branch instructions mentioned early and will be explained furtherly more in the full data path section. The following figure represent the ALU Block.

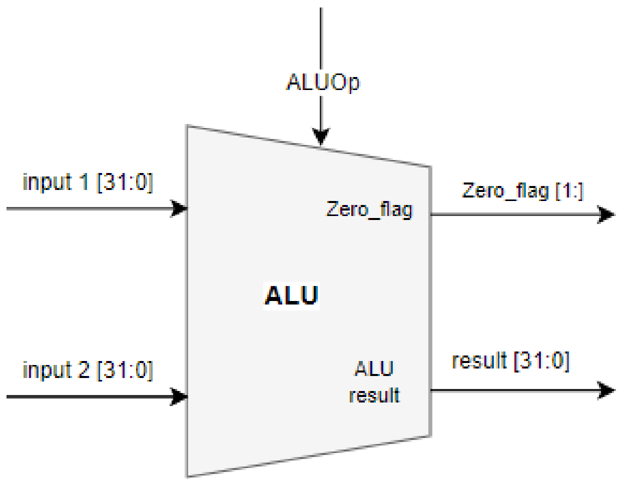


*Figure 8 Arithmetic and Logic Unit Block*

The following Algorithm explains how and when Zero_flag is reacted, first of all, two inputs are entering the ALU and the "ALUop" is also specified by the control unit then if the control unit sends a "jalr operation" command the Zero_flag will raise and normal output for "jalr" will result if not, then it might be a branch command hence, Zero_flag will raise and no result for the branch is taken, if not the Zero_flag will remain 0 for all other cases and the normal result will operate based on control unit command's as showing in the following figure.

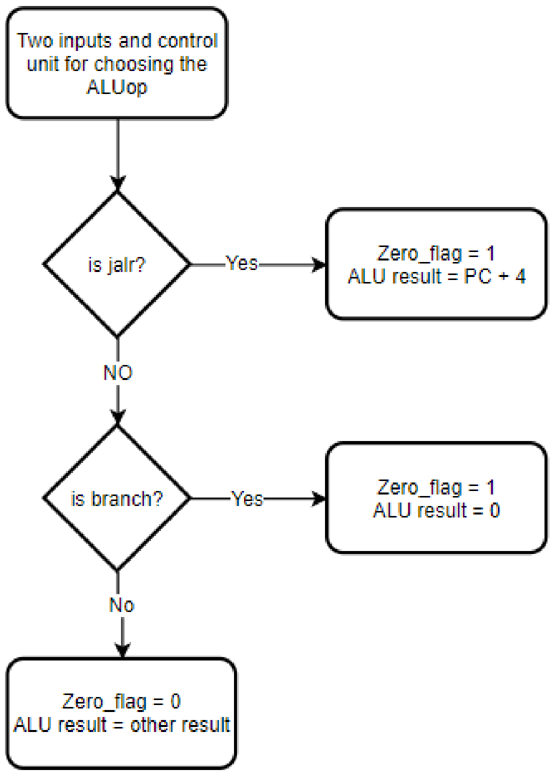
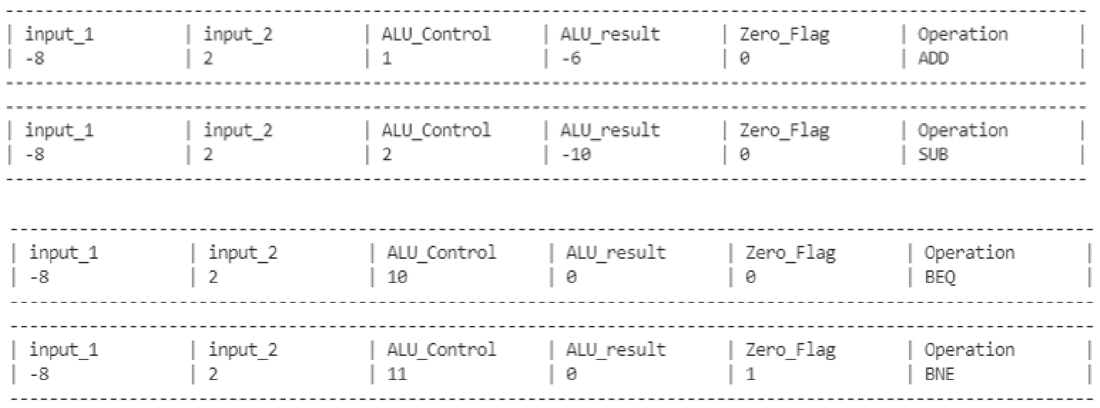


*Figure 9 ALU algorithm*

## Test Bench

In order to test operations in the ALU, we needed to insert random numbers into the ALU and test if the operation and the Zero_flag are working or not. As showing in the following figure, the output is working fine in these sample operations.



```
-----------------------------------------------------------------------------------------------------------
| input_1         | input_2         | ALU_Control     | ALU_result      | Zero_Flag       | Operation       |
| -8              | 2               | 1               | -6              | 0               | ADD             |
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
| input_1         | input_2         | ALU_Control     | ALU_result      | Zero_Flag       | Operation       |
| -8              | 2               | 2               | -10             | 0               | SUB             |
-----------------------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------------------
| input_1         | input_2         | ALU_Control     | ALU_result      | Zero_Flag       | Operation       |
| -8              | 2               | 10              | 0               | 0               | BEQ             |
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
| input_1         | input_2         | ALU_Control     | ALU_result      | Zero_Flag       | Operation       |
| -8              | 2               | 11              | 0               | 1               | BNE             |
-----------------------------------------------------------------------------------------------------------
```

*Figure 10 ALU Testbench's output*

## v. Control units

The control unit is one of the most important part in the data path. It is responsible for data flow throughout all parts. It also determines the operations that ALU and data memory that will doing. The control unit will receive three inputs which are OPcode, function 3 and function 7. Also, it is devised into three main parts. First part is responsible for data flow throughout whole data path. It receives OPcode signals as an input, then it will be sent 7 flags as an output, those flags will be sent either to multiplexer as selector signal or to other units as an enable signal. Second part is responsible for select right arithmetic and logical operator. This part will take all three inputs then, the operator will be determined by sending the appropriate signal based on the data. Last part of control unit is responsible for determined size of data in and data out from data memory. It will take OPcode and function 3 as an input then the selector signals will be sent to data memory.

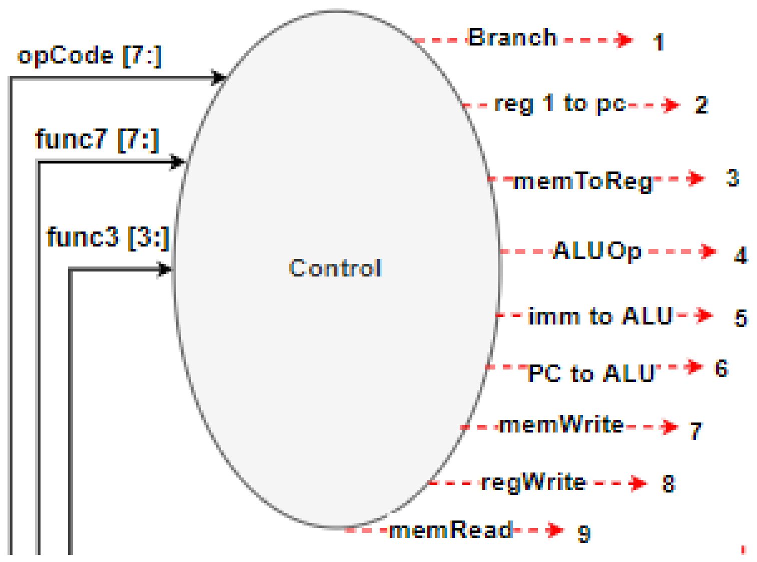
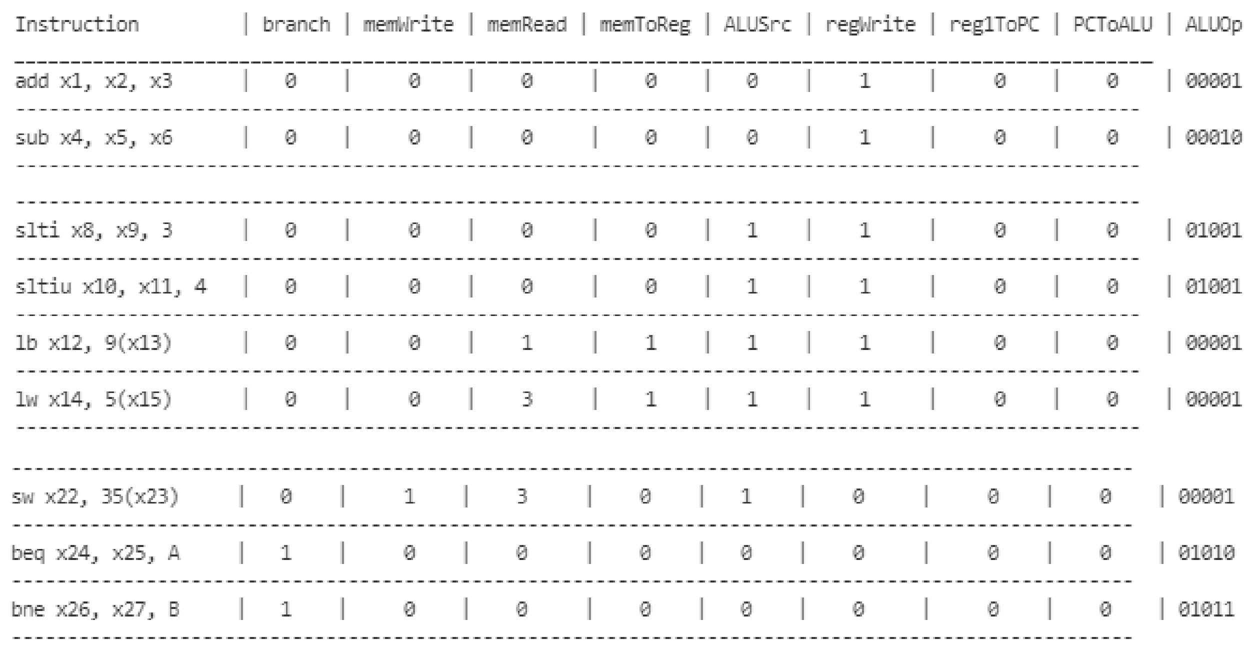


*Figure 11 Control unit*

## Test Bench

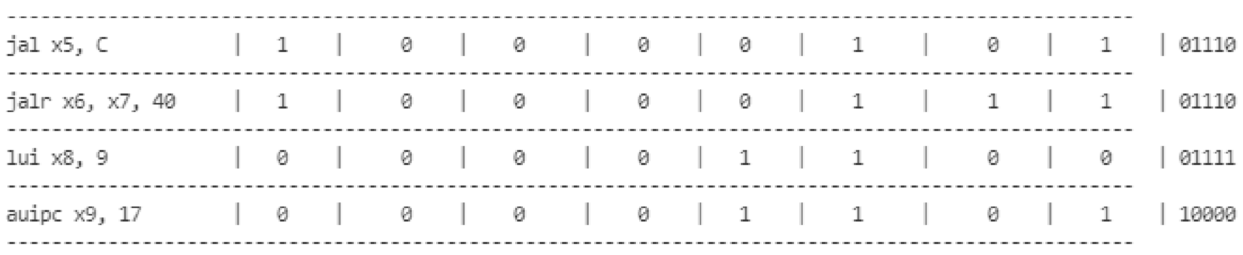
In the testbench, first of all I used the Rars assembler to write all instructions which we will used in our design. Then, the assembler will convert it to binary text code. After that, all codes passed into mine block to test it and show the result and the figures below will show some result.



```
Instruction          | branch | memWrite | memRead | memToReg | ALUSrc | regWrite | reg1ToPC | PCToALU | ALUOp
--------------------------------------------------------------------------------------------------------------
add x1, x2, x3       |   0    |    0     |    0    |    0     |   0    |    1     |    0     |    0    | 00001
--------------------------------------------------------------------------------------------------------------
sub x4, x5, x6       |   0    |    0     |    0    |    0     |   0    |    1     |    0     |    0    | 00010
--------------------------------------------------------------------------------------------------------------

--------------------------------------------------------------------------------------------------------------
slti x8, x9, 3       |   0    |    0     |    0    |    0     |   1    |    1     |    0     |    0    | 01001
--------------------------------------------------------------------------------------------------------------
sltiu x10, x11, 4    |   0    |    0     |    0    |    0     |   1    |    1     |    0     |    0    | 01001
--------------------------------------------------------------------------------------------------------------
lb x12, 9(x13)       |   0    |    0     |    1    |    1     |   1    |    1     |    0     |    0    | 00001
--------------------------------------------------------------------------------------------------------------
lw x14, 5(x15)       |   0    |    0     |    3    |    1     |   1    |    1     |    0     |    0    | 00001
--------------------------------------------------------------------------------------------------------------

--------------------------------------------------------------------------------------------------------------
sw x22, 35(x23)      |   0    |    1     |    3    |    0     |   1    |    0     |    0     |    0    | 00001
--------------------------------------------------------------------------------------------------------------
beq x24, x25, A      |   1    |    0     |    0    |    0     |   0    |    0     |    0     |    0    | 01010
--------------------------------------------------------------------------------------------------------------
bne x26, x27, B      |   1    |    0     |    0    |    0     |   0    |    0     |    0     |    0    | 01011
--------------------------------------------------------------------------------------------------------------
```



```
------------------------------------------------------------------------------------------
jal x5, C        |  1  |   0   |   0   |   0   |  0  |   1   |   0   |   1   | 01110
------------------------------------------------------------------------------------------
jalr x6, x7, 40  |  1  |   0   |   0   |   0   |  0  |   1   |   1   |   1   | 01110
------------------------------------------------------------------------------------------
lui x8, 9        |  0  |   0   |   0   |   0   |  1  |   1   |   0   |   0   | 01111
------------------------------------------------------------------------------------------
auipc x9, 17     |  0  |   0   |   0   |   0   |  1  |   1   |   0   |   1   | 10000
------------------------------------------------------------------------------------------
```

*Figure 12 output of control unit test bench*

## vi. Instruction memory

The instruction memory, I.M, have two parameter first, address that comes from the PC, program counter. The I.M will output 32 bits instruction to the decoder. Second The enable flag and it is “write enable”.

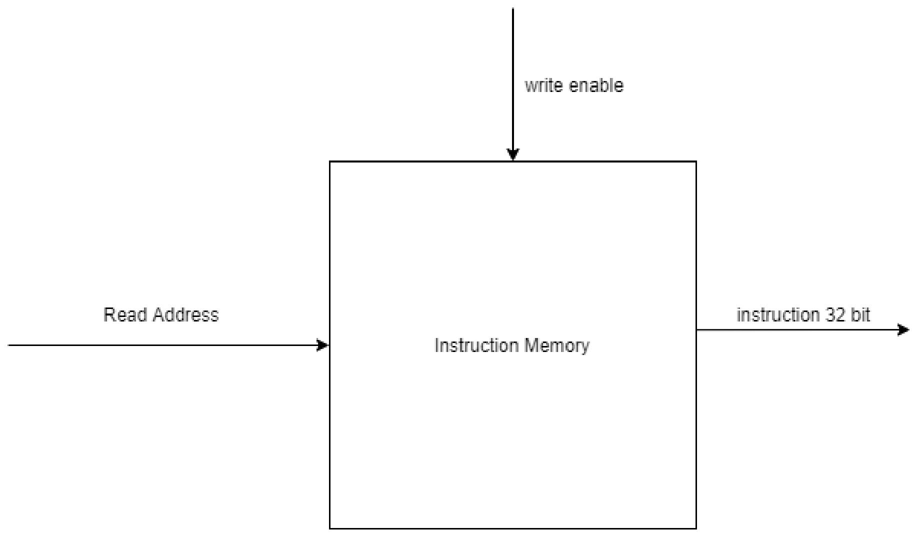
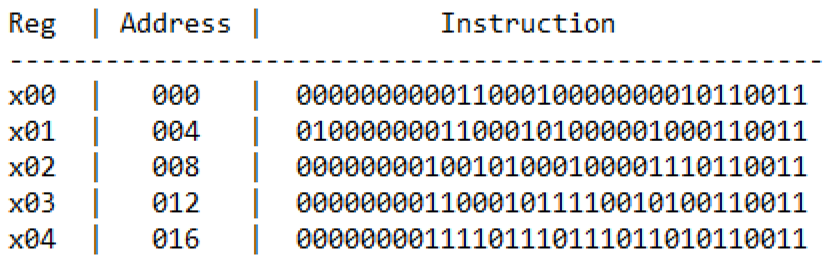


*Figure 13 Instruction memory*

### Test Bench

The instruction memory will take addresses from the PC and will fetch the 32-bit instruction and it will send it to decoder. The I.M will take the next address or next 4 bytes after it sends the first one and after write enable is ON.



```
Reg  | Address |            Instruction
------------------------------------------------------
x00  |   000   |  00000000001100010000000010110011
x01  |   004   |  01000000011000101000001000110011
x02  |   008   |  00000000100101000100001110110011
x03  |   012   |  00000000110001011110010100110011
x04  |   016   |  00000000111101110111011010110011
```

*Figure 14 output of instruction memory*

## vii. Data memory

Data memory is a block that takes an address and data and a flag for "data_write" the data will be taken form ALU as well as the address, but the flag will be given from control unit these all consider as an input furthermore data size will be given from the control unit considered as "memory_read" signal. We have storage of 4k-bytes indexes storage with 8-bit per index. we can enter data for it if we need to store. When load instruction executed the "data_write" flag will be zero but "memory_read" signal will be based on the type of load, if "lh" then it will be 2bytes if "lw" will 4 bytes and so on.

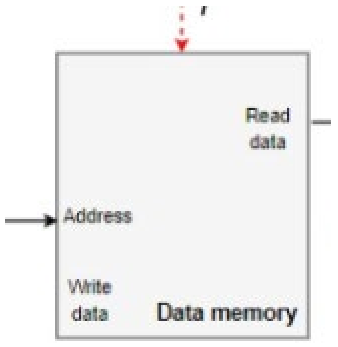


*Figure 15 Data memory*

### Test Bench

After writing the main block for the data memory, we needed to test our main Block to check the results and see the outputs of different cases depending on the inputs. First, initialize the input signals: "memory_read" (data size), "memory_write", data_write, address, dataIn, dataOut. Second, give different values for these inputs and monitor the reaction of the data memory either it'll read, or store based on the signals.

## viii. Other required logic

In order to complete our CPU and to avoid any errors and bugs, we needed to design some modules that will help us in branches and jump instructions and in fetching new instructions, etc. these modules are as follows:

### Adder & Branch Adder

Adder is a simple circuit that performs addition operation, in our data path we have an adder block which simply adds program counter (instruction address) plus 4 bytes because we have a 32 register each one has 32-bits wide range instruction and 4 byte is alike as if we moving to the next instruction address. While in Branch Adder is the same except that we are shifting input 2 to the left by one (input 2 = immediate coming from decoder) and this is for branch instructions for jumping and we will explain the concept of jumping later on at data path section. The following figure shows adder and branch adder blocks.

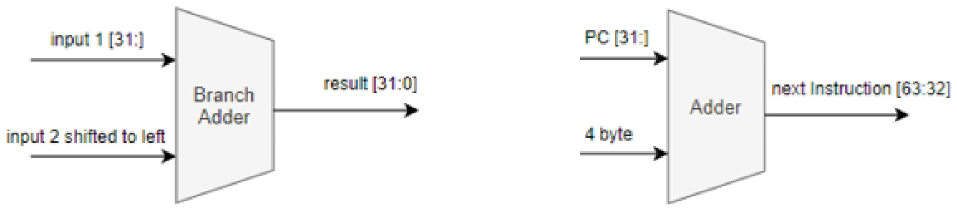
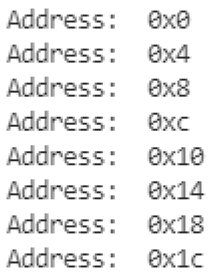


*Figure 16 Adder and Branch Adder Blocks*

## Test Bench

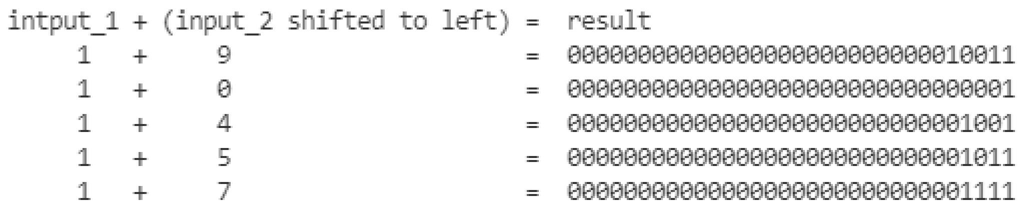
In the Adder test bench, simply we entered a for loop address plus 4 bytes. A sample of the output is shown in the next figure.



```
Address:  0x0
Address:  0x4
Address:  0x8
Address:  0xc
Address:  0x10
Address:  0x14
Address:  0x18
Address:  0x1c
```

*Figure 17 Adder Testbench's output*

While in Branch Adder test bench we inserted a random number for input 2 shifting it and the output comes in 32-bit format because it is an instruction going to the PC, the big picture is coming from the data path section. A sample of the output is shown in the next figure.



```
intput_1 + (input_2 shifted to left) =  result
     1   +     9                     =  00000000000000000000000000010011
     1   +     0                     =  00000000000000000000000000000001
     1   +     4                     =  00000000000000000000000000001001
     1   +     5                     =  00000000000000000000000000001011
     1   +     7                     =  00000000000000000000000000001111
```

*Figure 18 Branch Adder Testbench's output*

## Multiplexers

In our data path, we have five different multiplexers all of them are 2 to 1 case, but each one has a certain role which we will explain later on. The following figure shows the mux block.

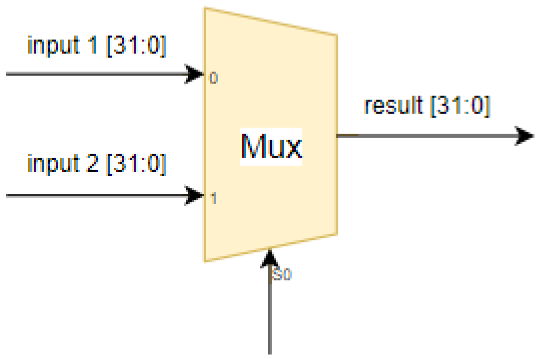
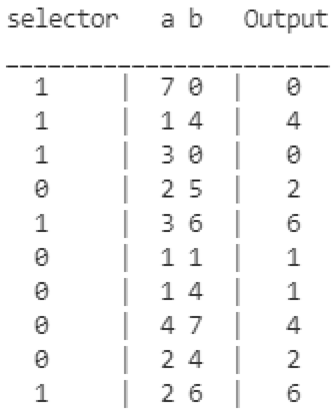


*Figure 19 Multiplexer Blocks*

### Test bench

In the Mux test bench, we simply entered random numbers for input a and b and random selectors between a or b to test the mux and it is working fine as following figure of the output mux test bench.



```
selector   a b   Output
-----------------------
  1     |  7 0  |   0
  1     |  1 4  |   4
  1     |  3 0  |   0
  0     |  2 5  |   2
  1     |  3 6  |   6
  0     |  1 1  |   1
  0     |  1 4  |   1
  0     |  4 7  |   4
  0     |  2 4  |   2
  1     |  2 6  |   6
```

*Figure 20 Multiplexer Testbench's output*

## And gate

In our data path, we needed a simple and gate to and "Zero_flag" and branch command coming from the control unit and the output will be the satisfied condition either high for branch or low for not. The output will act as a selector for multiplexer between jump instruction or normal fetching the next instruction. The reason for this gate will get clearer in the coming full data path section.

## ix. Top-level block

Now coming to the most important part in the report and after we explained every block in the data path and how they will work we are now looking at the big picture showing in the following figure.

### PC

Our data path starts from PC fetching the address of the instruction moving to the instruction memory.

### Instruction memory

Will fetch the data inside the entered address coming from the PC.

### Decoder

Then the instruction will be decoded in the decoder block output to the control unit opcode, funct3 and funct7 and outputting to the register file rs1, rs2 and rd, and outputting immediate to the branch adder and to the multiplexer2.

### Register File

Now register file will take rs1, rs2, rd, and regWrite if it is enabled to write in the register.

### Control Unit

Will control all the data path component based on the instruction opcode. So, we have 9 flags they are:

**Branch:** which is responsible on branch instructions.
**reg 1 to pc :** which is responsible on choosing between PC or rs1 that are going to sum with shifted immediate if it was rs1 + imm then it's jalr instruction if it was pc + imm then it's a branch or jal instruction and the output will go to the PC .
**memToReg:** which is responsible on which data are going to store in the register file it might be data from ALU or from data memory.
**ALUOp:**
**Imm to ALU:** which is responsible on which operation should be implemented.
**PC to ALU:** which is responsible on which input is pass to the ALU PC or rs1.
**memWrite:** which is responsible on enabling/disabling the memory from writing data in it.

**regWrite:** which is responsible on enabling/disabling to write on registers
**memRead:** which is responsible on enabling/disabling to read from data memory

### ALU

ALU block will operate based on what control unit's output.

### Data memory

Here the ALU result will pass to the address input and based on control unit the read/write will determined.

### Mux

We have five multiplexers, all of them are selected by Control unit except mux 5 is by and gate between control unit and zero flag from ALU

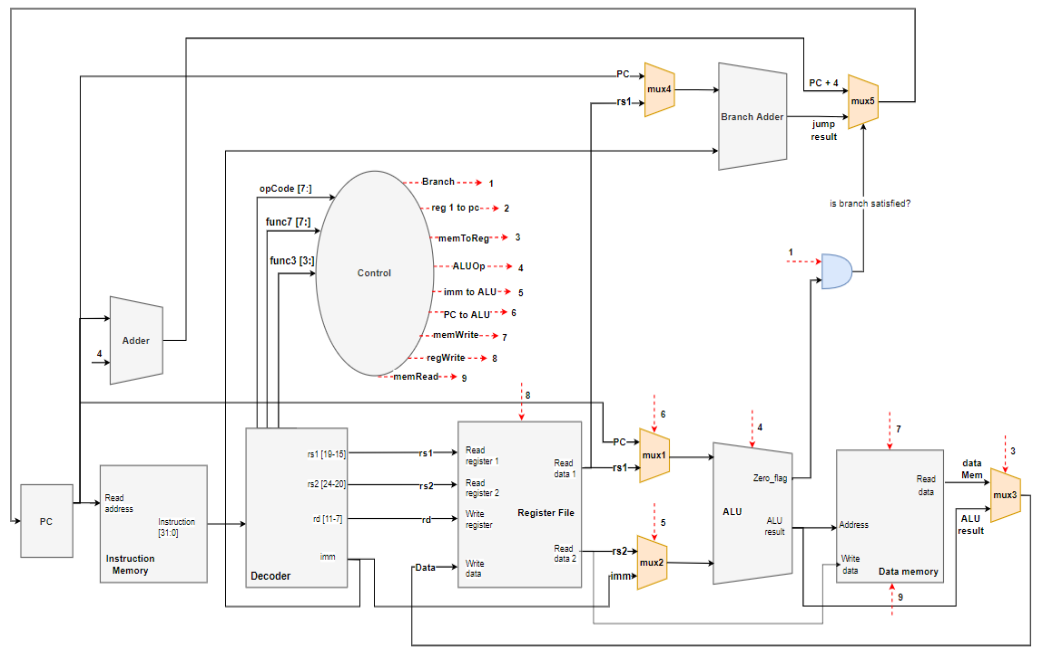


*Figure 21 Data path*

## x. Conclusion

At the end of this project, we would like to summarize all the important concepts and information we had been through in our journey. We learned how the CPU is working and what is the main component of the CPU, and how they interact with each other, we also learned many things such as Control unit determine the  flow of data in the data-path, and how the data is stored in the memory and loaded. However, when we began to execute this project, we encountered certain issues that took some time to resolve, such as data memory was storing unneeded data and branch instructions has some issues, which made the algorithm of bubble sorting unworkable. Although we solve everything and the bubble sort is working now but in general, our codes need a lot of improvement, it has lots of dump algorithms. Finally , we got experience in building and constructing a CPU, which enhanced our concept and provided us with knowledge in a highly helpful Python library known as MyHDL. Due to a lack of references, learning and training on this library required a lot of time and effort, but it was well worth it. We haven't gotten acclimated to this library yet, but with additional training and projects, we will, and this will provide us with a solid tool to utilize later.

## xi. Appendix

### Appendix A: Self-Assessment Tables

| | | Our weigh | octor weig | Osama | Azeez | Tamam | Mostafa |
|---|---|---|---|---|---|---|---|
| Instruction Memory | code | 9 | 7 | 80 | 20 | 0 | 0 |
| | Report | 2 | 3 | 0 | 0 | 70 | 30 |
| Decoder | code | 7 | 5 | 0 | 0 | 0 | 100 |
| | Report | 2 | 3 | 0 | 0 | 0 | 100 |
| Control Unit | code | 8 | 9 | 80 | 20 | 0 | 0 |
| | Report | 2 | 3 | 100 | 0 | 0 | 0 |
| Register File | code | 7 | 1 | 10 | 90 | 0 | 0 |
| | Report | 2 | 1 | 0 | 100 | 0 | 0 |
| ALU | code | 6 | 10 | 30 | 70 | 0 | 0 |
| | Report | 2 | 4 | 0 | 100 | 0 | 0 |
| Data memory | code | 9 | 6 | 20 | 0 | 80 | 0 |
| | Report | 2 | 3 | 0 | 0 | 100 | 0 |
| Adder & Branch Adder | code | 3 | 5 | 0 | 100 | 0 | 0 |
| | Report | 2 | 4 | 0 | 100 | 0 | 0 |
| CPU_TOP | code | 11 | 9 | 70 | 30 | 0 | 0 |
| | Report | 2 | 4 | 20 | 80 | 0 | 0 |
| Data Path drawing | | 5 | 6 | 0 | 100 | 0 | 0 |
| Power point Report | | 7 | 6 | 0 | 0 | 0 | 100 |
| Introduction Report | | 3 | 3 | 0 | 0 | 100 | 0 |
| Conclusion Report | | 3 | 3 | 0 | 0 | 80 | 20 |
| Formatting Report | | 5 | 5 | 0 | 0 | 100 | 0 |
| Our over all | | 99 | | 28.28% | 33.13% | 21.21% | 17.37% |
| Doctor over all | | | 100 | 27% | 37% | 20% | 16% |

| | | Mark out of 10 |
|---|---|---|
| Report | Well Formatted according to PTW checklist | 8 |
| | Spelling and grammar checked | 8 |
| | Clear Writing Style | 10 |
| | Includes detailed problem statement, project objectives, and impact | 7 |
| | Effective use of illustrative figures and tables | 9 |
| | The report covers all aspects of the project/assignment, including pictures and diagrams of the implementation | 8 |
| | Includes detailed justifications for all design choices | 9 |

| | | |
|---|---|---|
| | Reflects on learned lessons and failed experiments | 6 |
| | Includes a table of work distribution between the team members | 10 |
| | Use of References | NA |
| **Presentation** | Creative slide design and illustrations (Graphics) | 9 |
| | Only few readable text maybe with Graphics in each slide to deliver one clear idea | 8 |
| | Slides are numbered | 10 |
| | Language and fluency | 8 |
| | Voice and Confidence | 8 |
| | Clarity and Pace | 9 |
| | Enthusiasm | 9 |
| **Implementation** | Full Simulation is working | 10 |
| | Code is Working as Expected | 5 |
| | Meets all the requirements | 6 |
| | Extra Features | 4 |
| | The code is fully commented in a readable way | 8 |
| | The code is readable and meet good coding standards | 7 |
| **Discussion** | Skills Improvements | **By Instructor** |
| | Participation % | **By Instructor** |
| | Understanding | **By Instructor** |

## Appendix B: (code & synthesis)

### A. Pc

```
@block
def PC_reg(PCOutput,PCInput,clock, enable):
    @always(clock.posedge)
    def passIt():
        if enable:
            PCOutput.next = PCInput
        else:
            PCOutput.next = 0
    return passIt
```

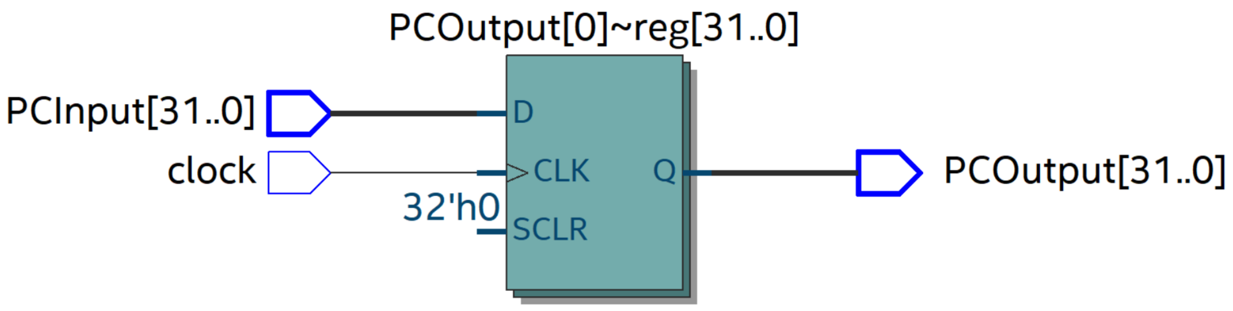


*Figure 22 Program Counter RTL*

## B. Adder

```
from myhdl import *

# ----------------------------------------------------------------------------- #
#                                main Block                                     #
# ----------------------------------------------------------------------------- #
@block
def adder(PC,nextInstruction):

    @always(PC)
    def add():
        increament=intbv(4)[32:]
        nextInstruction.next = PC + increament

    return add
# ----------------------------------------------------------------------------- #
#                                Test-Bench                                     #
# ----------------------------------------------------------------------------- #
@block
def test():

    pc, nextInstruction = [Signal(intbv(0)[32:]) for i in range(2)]

    ProgCounter = adder(pc,nextInstruction)

    @instance
    def test():
        print("Address: ",hex(nextInstruction))

        for i in range(0,40,4):
            pc.next = i

            yield delay(5)
            print("Address: ",hex(nextInstruction))


    return instances()

tb = test()
tb.run_sim(100)

# # ----------------------------------------------------------------------------- #
# #                             Verilog Conversion                                #
# # ----------------------------------------------------------------------------- #

def convert(hdl):
    pc, nextInstruction = [Signal(intbv(0)[32:]) for i in range(2)]

    inst = adder(pc, nextInstruction)
    inst.convert(hdl=hdl)
```

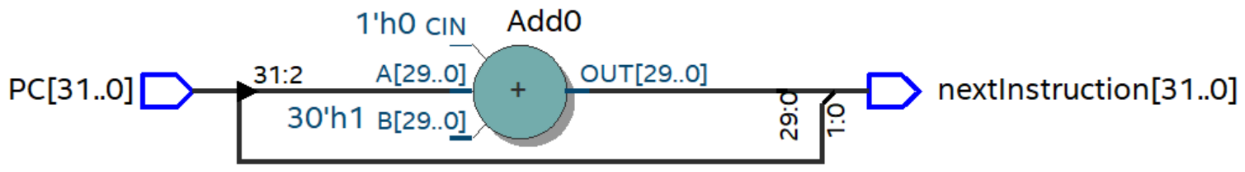


*Figure 23 Adder RTL*

## C. Branch adder

```
from myhdl import *
import random
randomrange = random.randrange

# ---------------------------------------------------------------------------- #
#                                  main Block                                  #
# ---------------------------------------------------------------------------- #

@block
def BranchAdder(int_1,int_2,out):

    @always(int_1,int_2)
    def adder():

        out.next = int_1 + (int_2 << 1)

    return adder

# ---------------------------------------------------------------------------- #
#                                  Test-Bench                                  #
# ---------------------------------------------------------------------------- #
@block
def test():

    int_1, int_2, out = [Signal(intbv(0)[32:]) for i in range(3)]
    Branch_Adder = BranchAdder(int_1,int_2,out)

    @instance
    def test():
        print()
        print("intput_1 + (input_2 shifted to left) =  result")
        for i in range(5):
            int_1.next = 1
            int_2.next = randomrange(10)
            yield delay(5)
            print("     %-3d +     %-21d =  %-
10s" % (int(int_1), int(int_2), bin(out, 32)))


    return instances()

tb = test()
tb.run_sim(100)

# ----------------------------------------------------------------------- #
# #                          conversion                                    #
# # ----------------------------------------------------------------------- #

def conversion():
```

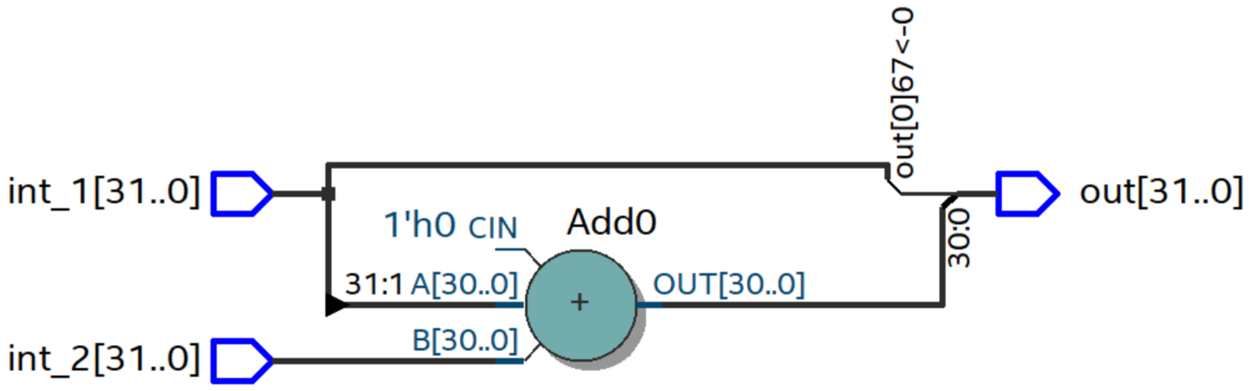


*Figure 24 Branch Adder RTL*

## D. ALU

```
from myhdl import *

# ---------------------------------------------------------------------------- #
#                                 Main Block                                   #
# ---------------------------------------------------------------------------- #

@block
def ALU(input_1,input_2,ALU_Control,ALU_result,Zero_Flag):
    @always(input_1,input_2,ALU_Control)
    def Operand():

        if ALU_Control == 1:                  #! ADD operation
            ALU_result.next = input_1 + input_2
            Zero_Flag.next = 0

        elif ALU_Control == 2:                #! SUB operation
            ALU_result.next = input_1 - input_2
            Zero_Flag.next = 0

        elif ALU_Control == 3:                #! XOR operation
            ALU_result.next = input_1 ^ input_2
            Zero_Flag.next = 0

        elif ALU_Control == 4:                #! OR operation
            ALU_result.next = input_1 | input_2
            Zero_Flag.next = 0

        elif ALU_Control == 5:                #! AND operation
            ALU_result.next = input_1 & input_2
            Zero_Flag.next = 0

        elif ALU_Control == 6:                #! SLL operation
            ALU_result.next = input_1 << input_2
            Zero_Flag.next = 0

        elif ALU_Control == 7:                #! SRL operation
            ALU_result.next = input_1 >> input_2
            Zero_Flag.next = 0

        elif ALU_Control == 8:                #! SRA operation
            ALU_result.next = modbv((input_1 >> input_2) | (input_1 >> (len(in-
put_1) - input_2))).signed()
            Zero_Flag.next = 0

        elif ALU_Control == 9:                #! SLT & SLTU operation
            if(input_1 < input_2):
                ALU_result.next = 1
                Zero_Flag.next = 0
```

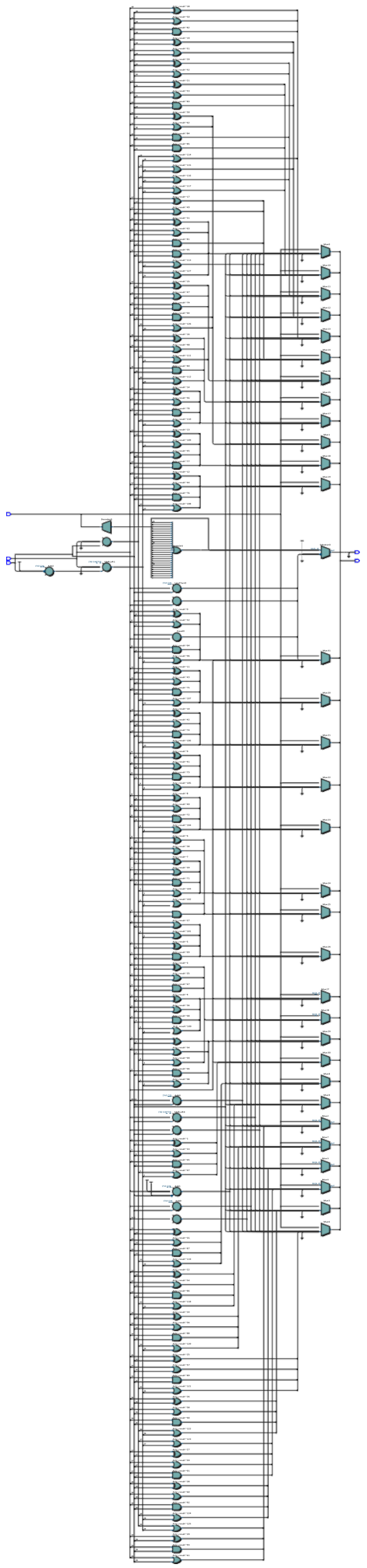


*Figure 25 Arithmetical and Logical Unit (ALU) RTL*

**Too long , can be seen clearly in the uploaded PDF.**

## E. Control unit

```
from myhdl import * #block, always_comb, instance, Signal, intbv, delay

@block
def control(branch, memWrite, memToReg, ALUOp, immToALU, regWrite, reg1ToPC, PCToALU, OP-
code, func3, func7):

    @always_comb
    def controller():
        if OPcode == 0b0110011:  # for R-type
            branch.next = 0
            memWrite.next = 0
            memToReg.next = 0
            immToALU.next = 0
            regWrite.next = 1
            reg1ToPC.next = 0
            PCToALU.next = 0

        elif OPcode == 0b0010011 : # for I-type without load
            branch.next = 0
            memWrite.next = 0
            memToReg.next = 0
            immToALU.next = 1
            regWrite.next = 1
            reg1ToPC.next = 0
            PCToALU.next = 0

        elif OPcode == 0b0000011 : # for I-type with load
            branch.next = 0
            memWrite.next = 0
            memToReg.next = 1
            immToALU.next = 1
            regWrite.next = 1
            reg1ToPC.next = 0
            PCToALU.next = 0

        elif OPcode == 0b0100011 : # for S-type
            branch.next = 0
            memWrite.next = 1
            memToReg.next = 0
            immToALU.next = 1
            regWrite.next = 0
            reg1ToPC.next = 0
            PCToALU.next = 0

        elif OPcode == 0b1100011 : # for B-type
            branch.next = 1
            memWrite.next = 0
            memToReg.next = 0
```

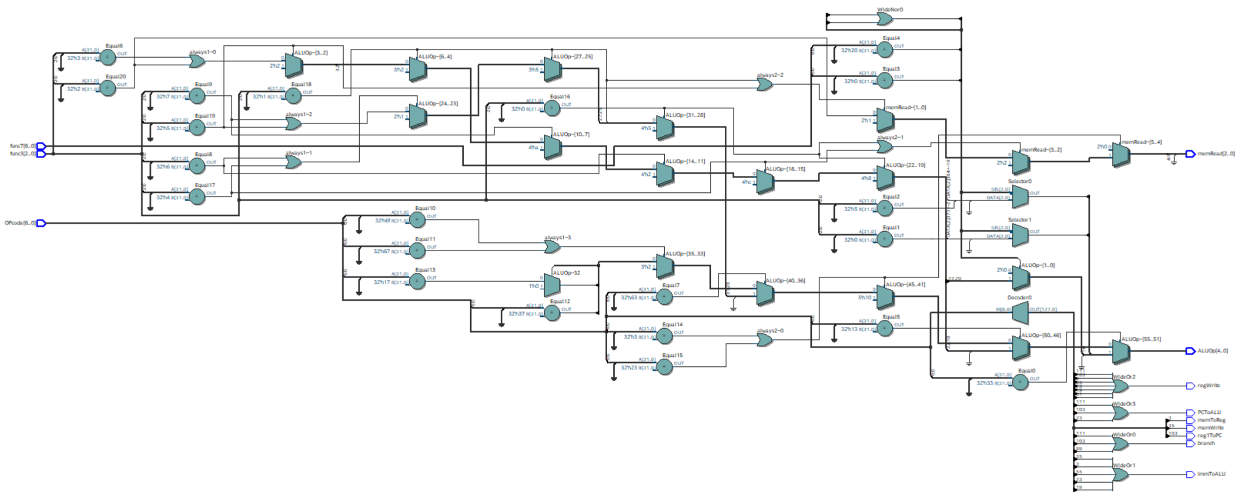


*Figure 26 Control Unit RTL*

## F. Data memory

```
'''
@auther: tamam alahdal
'''

from myhdl import block,bin,Signal,intbv,delay,instance,always,instances
import random
#------------------------------------------------------------------------
#                        main Block                                     -

#------------------------------------------------------------------------
@block
def DataMemory (address,dataIn,memoryRred,memoryWrite,dataOut):
    memory = [Signal(intbv(0, min=-
2**8, max=2**8)) for i in range(10240)]     #memory size is 10k byte
    # index = 0
    # with open('Sort_dataCode.txt', 'r') as f:
    #     for line in f:
    #         data_in = intbv(line)
    #         memory[index+3].next = data_in[32:24]
    #         memory[index+3]._update()
    #         memory[index+2].next = data_in[24:16]
    #         memory[index+2]._update()
    #         memory[index+1].next = data_in[16:8]
    #         memory[index+1]._update()
    #         memory[index].next = data_in[8:0]
    #         memory[index]._update()
    #         index += 4
    @always(address,dataIn)
    def Access():
        if memoryRred == 1:
            if(memoryWrite):
                memory[address].next = dataIn[8:0]
                dataOut.next = 0
            else:
                dataOut.next[8:0] = memory[address]

        if memoryRred == 2:
            if(memoryWrite):
                memory[address+1].next = dataIn[16:8]
                memory[address].next = dataIn[8:0]
                dataOut.next = 0
            else:
                dataOut.next[16:8] = memory[address+1]
                dataOut.next[8:0] = memory[address]
        if memoryRred == 3:
```

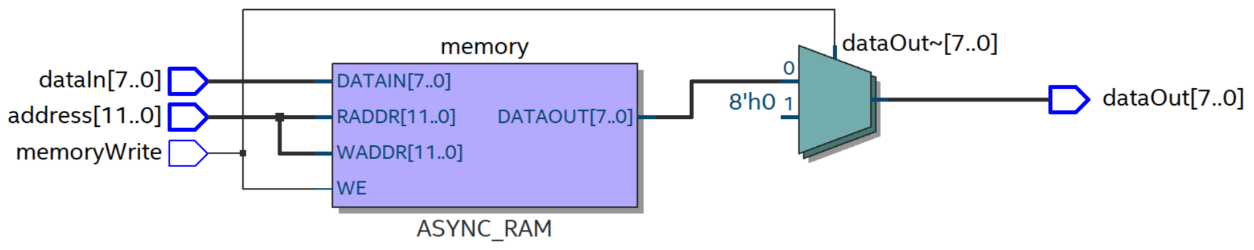


*Figure 27 Data memory RTL*

## G. Decoder

```
from myhdl import block, always_comb, instances, instance, Signal, intbv, delay,bin,concat , always
#-------------------------------------------------------------------------------------------------#
#                                         Main Block                                              #
#-------------------------------------------------------------------------------------------------#

@block
def decoder32(instruction ,opcode ,rs1, rs2, rd , imm, funct3, funct7):

    @always(instruction)
    def decode32():
        opcode.next = instruction[7:]
        rs1.next=instruction[20:15]
        rs2.next=instruction[25:20]
        rd.next=instruction[12:7]
        funct3.next =instruction[15:12]
        funct7.next = instruction[32:25]

    @always(instruction)
    def immediate():
        opcode_in = instruction[7:]

        if opcode_in == 0b0110011:
            imm.next = 0
           # R Type instructions
        elif opcode_in == 0b0101111:
            imm.next = 0
         #   R Type - Atomic Extension - instructions

        elif opcode_in == 0b0000011:
            imm.next = intbv(instruction[32:20]).signed()
        elif opcode_in == 0b0010011:
            function3 = instruction[15:12]
            if function3==0b001 or function3==0b101:
                imm.next=intbv(instruction[25:20],min=-2**31, max=2**31)
            else:
                imm.next=intbv(instruction[32:20],min=-2**31, max=2**31)
           # I type - ALU - instructions

        elif opcode_in == 0b0100011:
            imm.next = intbv(concat(instruction[32:25],instruction[12:7]),min=-2**31, max=2**31)
           # S Type instructions



        elif opcode_in == 0b1100011:
            imm.next=intbv(concat(instruction[32:31],instruction[8:7],instruction[31:25],instruc-
tion[12:8]),min=-2**31, max=2**31)
           #SB type instructions
        elif opcode_in == 0b0010111 or opcode_in== 0b0110111:
            imm.next=intbv(instruction[32:12],min=-2**31, max=2**31)
```

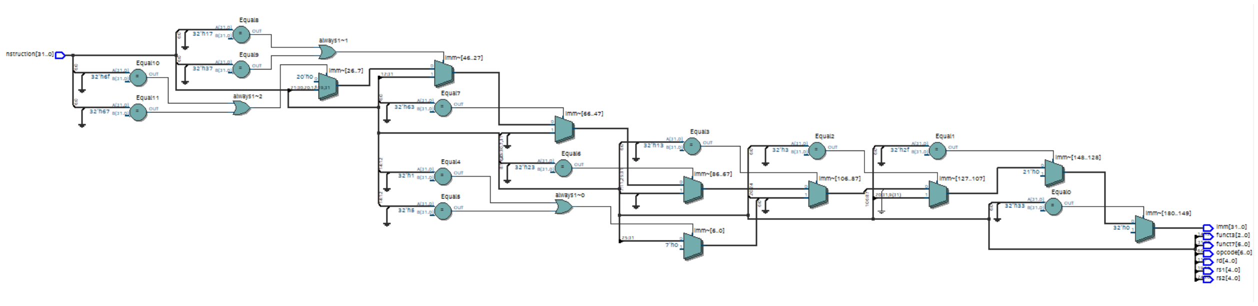


*Figure 28 Decoder RTL*

## H. Instruction memory

```
from myhdl import *

# -------------------------------------------------------------------------- #
#                                main block                                  #
# -------------------------------------------------------------------------- #

@block
def instMem(inst, address, data_in, write_enable):
    memory = [Signal(intbv(0)[8:]) for i in range(32*4)]
    @always(address, data_in)
    def fetch():
        if write_enable:
            memory[address+3].next = data_in[32:24]
            inst.next[32:24] = 0
            memory[address+2].next = data_in[24:16]
            inst.next[24:16] = 0
            memory[address+1].next = data_in[16:8]
            inst.next[16:8] = 0
            memory[address].next = data_in[8:0]
            inst.next[8:0] = 0
        else:
            inst.next[32:24] = memory[address+3]
            inst.next[24:16] = memory[address+2]
            inst.next[16:8] = memory[address+1]
            inst.next[8:0] = memory[address]
    return instances()


inst = Signal(intbv(0)[32:])
address = Signal(intbv(0)[32:])
data_in = Signal(intbv(0)[32:])
write_enable = Signal(intbv(0)[1:])
# -------------------------------------------------------------------------- #
#                                Test-Bench                                  #
# -------------------------------------------------------------------------- #
@block
def test_instMem():

    tstInstMem = instMem(inst, address, data_in, write_enable)

    @instance
    def test():
        write_enable.next = 1
        i = 0
        with open('InstructionCode.txt', 'r') as f:
            for line in f:
                data_in.next = Signal(intbv(line))
                address.next = i*4
                i += 1
```

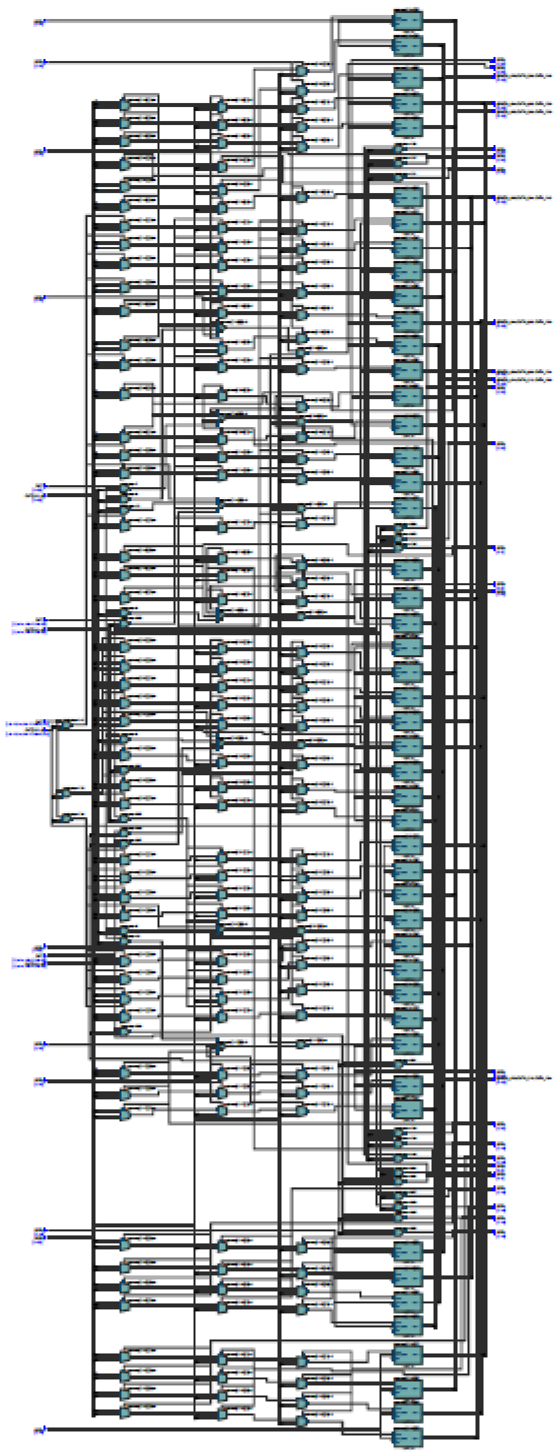


*Figure 29 Instruction memory RTL*

**Too long , can be seen clearly in the uploaded PDF.**

## I. Register File

```
from myhdl import block, instances, instance, Signal, intbv, delay, always, Re-
setSignal

# ---------------------------------------------------------------------------- #
#!                                  Main Block                                  #
# ---------------------------------------------------------------------------- #
@block
def RegisterFile(REGwrite, rs1_Out, rs2_Out, rs1_In, rs2_In, rd, data):

    registers = [Signal(intbv(0,min=-2**31,max=2**31)) for i in range(32)]
    @always(rs1_In, rs2_In, rd)
    def registersFile():

        # First output value from register number [rs1_In]
        rs1_Out.next = registers[rs1_In]
        # Second output value from register number [rs2_In]
        rs2_Out.next = registers[rs2_In]
        print("data1 ",int(data))
        print("rs1_In ",int(rs1_In))
        print("rs2_In ",int(rs2_In))
        print("rs1_Out ",int(rs1_Out))
        print("rs2_Out ",int(rs2_Out))
    @always(data, REGwrite)
    def writeData():
        if REGwrite:
            registers[rd].next = data
            print("data2 ",int(data))



    return instances()

# ---------------------------------------------------------------------------- #
#                                  Test-Bench                                  #
# ---------------------------------------------------------------------------- #


@block
def TestBench():

    rs1_In, rs2_In, rd = [Signal(intbv(0)[5:]) for i in range(3)]
    rs1_Out, rs2_Out, data = [Signal(intbv(0)) for i in range(3)]
    REGwrite = Signal(intbv(0)[2:])
    reg1 = RegisterFile(REGwrite, rs1_Out, rs2_Out, rs1_In, rs2_In, rd, data )


    @instance
    def test():
        yield delay(5)
```

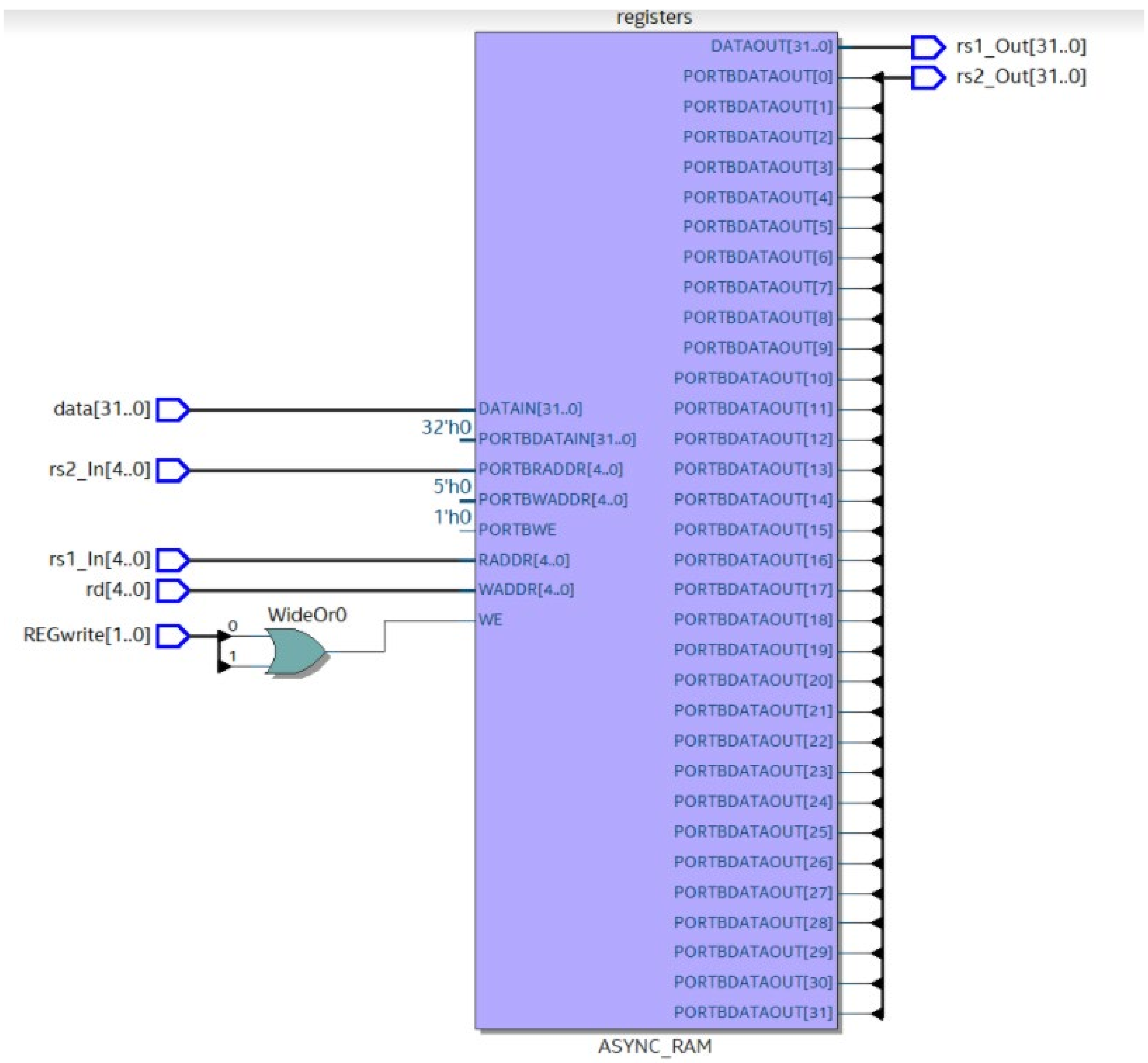


*Figure 30 Register File RTL*

## J. CPU_TOP_LEVEL

```
from myhdl import *
from PC import pc
from instMem_v2 import instMem
from Decoder import decoder32
from RegisterFile import RegisterFile
from control import control
from ALU import ALU
from mux import mux
from mux_2 import mux2
from mux_3 import mux3
from mux_4 import mux4
from mux_5 import mux5
from Adder import adder
from BranchAdder import BranchAdder
from DataMemory import DataMemory

# ---------------------------------------------------------------------------- #
#                                  main Block                                  #
# ---------------------------------------------------------------------------- #


@block
def PC_reg(PCOutput,PCInput,clock, enable):
    @always(clock.posedge)
    def passIt():
        if enable:
            PCOutput.next = PCInput
        else:
            PCOutput.next = 0
    return passIt



@block
def CPU(data, data_in, write_enable, clock, enable):


    dataMem,  ALUinput1, ALUinput2, ALU_result, rs1_Out, rs2_Out, jumpInstruc-
tion, imm = [Signal(intbv(0, min=-2**32, max=2**32)) for i in range(8)]
    instruction, nextInstruction, currentAddress = [Sig-
nal(intbv(0)[32:]) for i in range(3)]
    pcInput, address = [Signal(modbv(1, min=0, max=33)) for i in range(2)]
    rs1_In,rs2_In,rd=[Signal(intbv(0)[5:])for i in range(3)]
    funct3,memRead = [Signal(intbv(0)[3:]) for i in range(2)]
    funct7 = Signal(intbv(0)[7:])
    opCode = Signal(intbv(0)[7:])
    ALUOp = Signal(intbv(0)[5:])
    branch, memWrite, memToReg, immToALU, regWrite, reg1ToPC, pcToALU = [Sig-
nal(bool(0)) for i in range(7)]
```

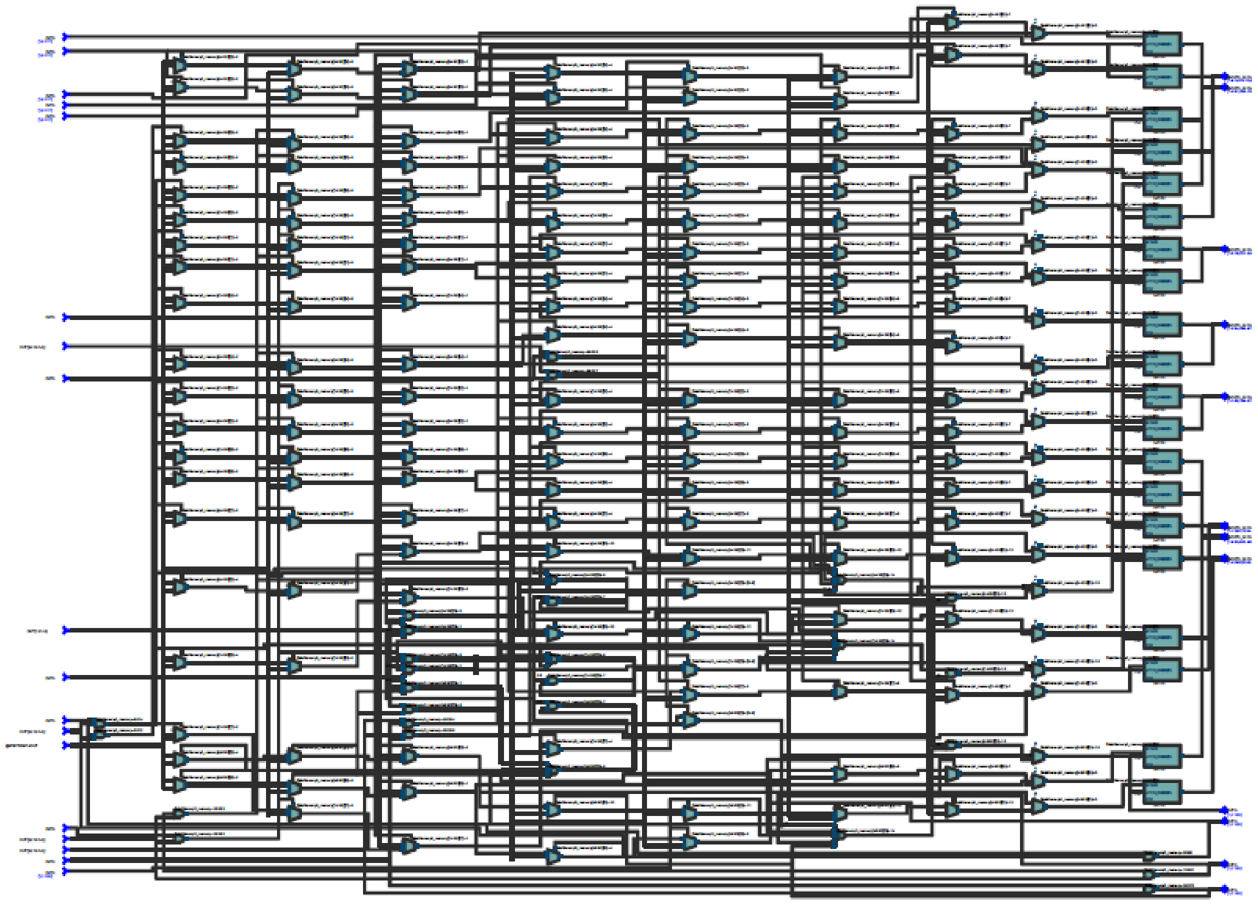


*Figure 31 CPU_TOP RTL*

**Too long , can be seen clearly in the uploaded PDF.**