King Fahd University of Petroleum and Minerals

Department of Information and Computer Science

Control Process Unit Design Project

ICS 233 – Term 142

Instructor: Mr. Kamal

Team Members:

Mossad Helali **201265040**

Mohammed Al-Sayed **201256520**

Table of Contents

Figure Table

Introduction

Objectives:

- Using the Logisim simulator
- Designing and testing a Pipelined 16-bit processor

Description:

Designing a 16-bit RISC processor with eight 16-bit general-purpose registers: R0 through R7. R0 is hardwired to zero and cannot be written. The program counter is a special-purpose 16-bit register. All instructions are 16 bits. There are four instruction formats, R-type, I-type, B-type and J-type as shown below:

R-type format:

5-bit opcode (Op), 2-bit function field (f), and 3-bit register numbers (Rd, Rs, and Rt)

I-type format:

5-bit opcode (Op), 5-bit immediate constant and 3-bit register number (Rs and Rt)

B-type format:

5-bit opcode (Op), 8-bit immediate constant and 3-bit register number (Rt)

J-type format:

5-bit opcode (Op) and 11-bit immediate constant for R-type instructions, Rs and Rt specify the source registers, and Rd specifies the destination register number. The function field can specify at most four functions with same opcode. For I-type instructions, Rs specifies a source register number. Rt can be a second source or a destination register number. The immediate constant is only 5 bits because of the short-size nature of the instruction. The 5-bit immediate constant is always signed (sign-extended). For B-type, Rt can be a source or a destination register number. The 8-bit Immediate is signed. For J-type, an 11-bit PC-relative signed immediate is used for J (jump), and JAL (jump-and-link) instructions.

Phase One: Designing the CPU Components

Designing Components:

Program Counter (PC)

To clearly understand the program counter, we know that every program is constructed of multiple lines of code. So, to determine in which line instruction is fetched there is a register that holds this task. Similarly, the pc determines which instruction to fetch in the machine code. However, the PC is constructed of a register block determining the instructions address to fetch.

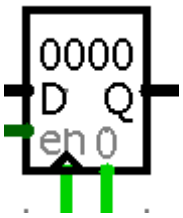


Figure 1: PC

Splitter

The splitter takes the instruction data (code) and splits it into different chunks. The size of each chunk is based on the instruction formats. Where we have 4 formats of instruction: R-Type, I-Type, B-Type, J-Type, and each type has its unique instruction format. Therefore the instruction data is split into 8 different chunks, where all the chunks are available permanently as following:

1- Operation Code (Opcode) – 5 Bits
2- Resource (Rs) – 3 Bits
3- Resource Target (Rt) – 3 Bits
4- Resource Destination (Rd) – 3 Bits
5- Function (f) – 2 Bits
6- Immediate (Imm) – 5,8,11 different sizes of Bit chunks

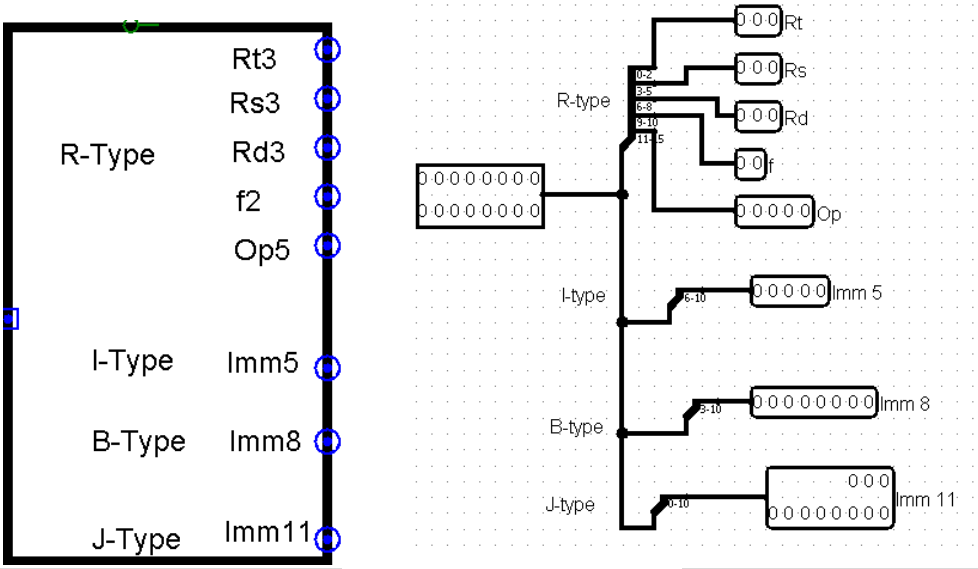


Figure 2: Instruction Splitter

Register File:

As programming, there are values that should be saved in order to process it by different methods. Therefore, there is a block named “File Register” containing multiple registers that saves various values in each register.

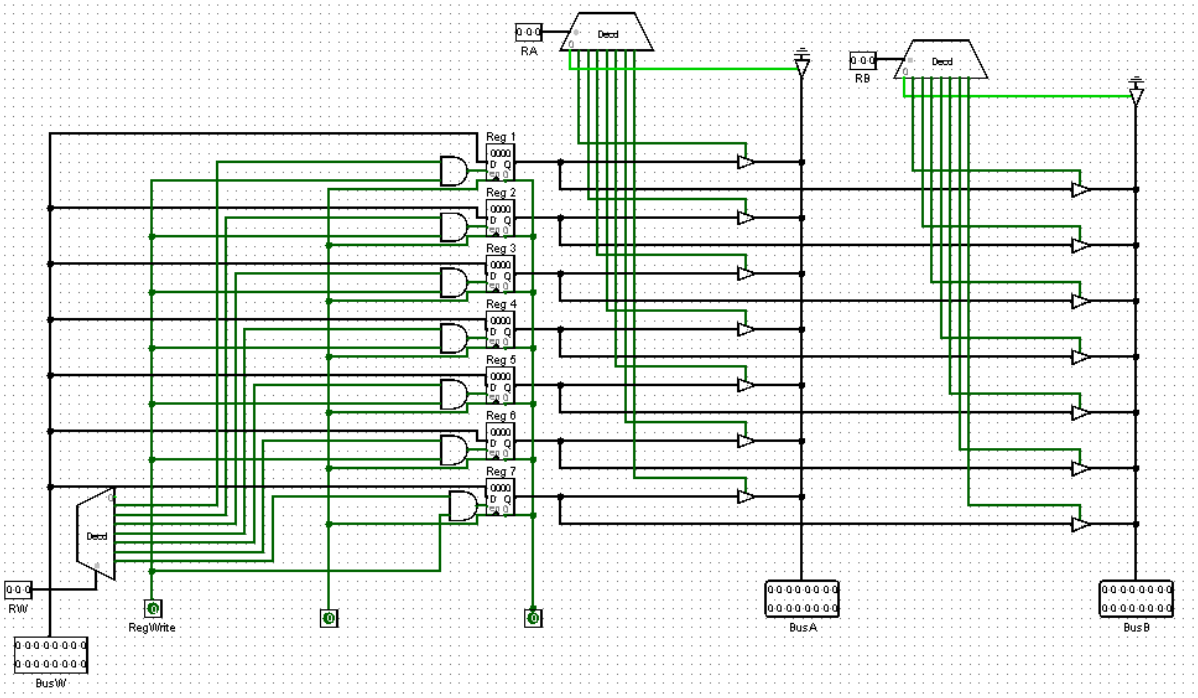


Figure 3: File Register

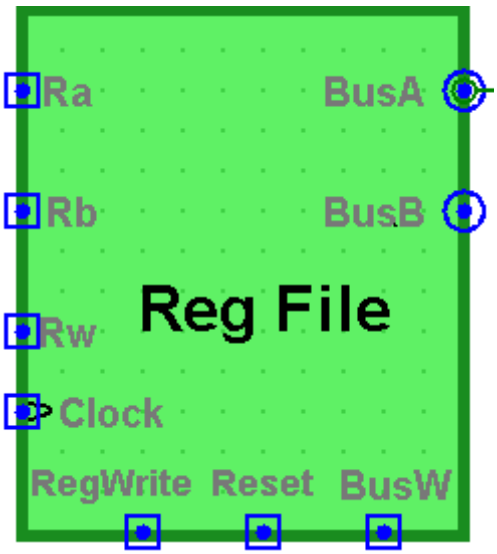


Figure 4: File Register block

1- Ra = Register address a, takes the address of Rs
2- Rb = Register address a, takes the address of Rt
3- Rt = Register address a, takes the address of Rd or Rt
4- Rt = Register address a, takes the address of Rd or Rt
5- RegWrite = Register Write, enabling the registers to accept input values.
6- BusA = The data stored in Ra which consists of 16 bits.
7- BusB = The data stored in Rb which consists of 16 bits.
8- BusW = The write back data input, it writes the data into the Rw register address after a single cycle over the processor.
9- Reset = Reset's the register values into ZERO.

Arithmetic and Logic Unit

The ALU is responsible for doing arithmetic operations on operand whether they are immediate values or registers. It consists of three blocks: shift, arithmetic and logic blocks.

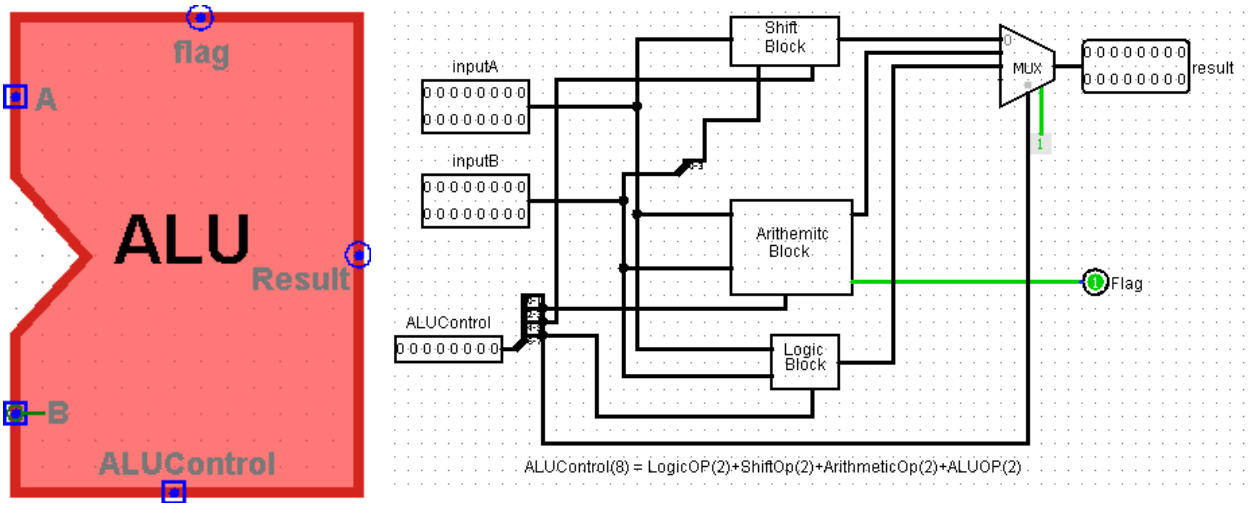


Figure 5: Arithmetic and Logic Unit (ALU)

1. *Shift Block*

Responsible for shifting and rotating as required in the project description.

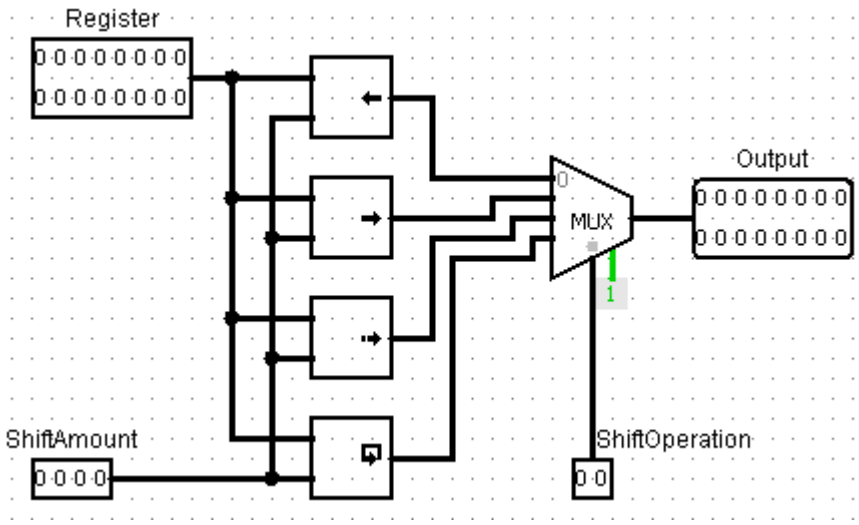


Figure 6: Shift Block

2. *Arithmetic Block*

Responsible for adding/subtracting, and slt, sltu instructions.

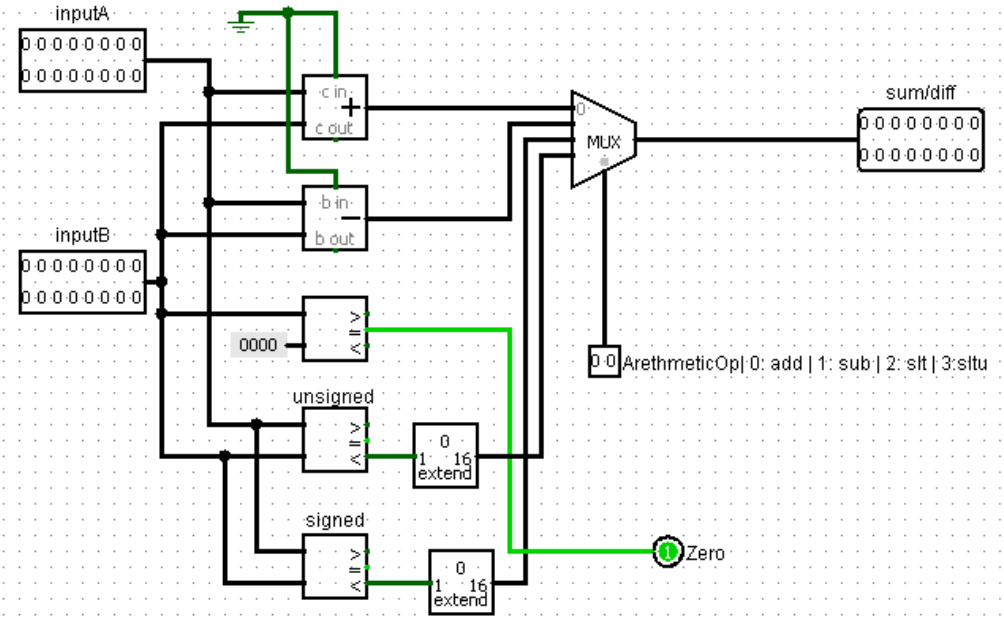


Figure 7: Arithmetic Block

3. *Logic Block*

Responsible for doing logic operations such as: and, andc, xor and or.

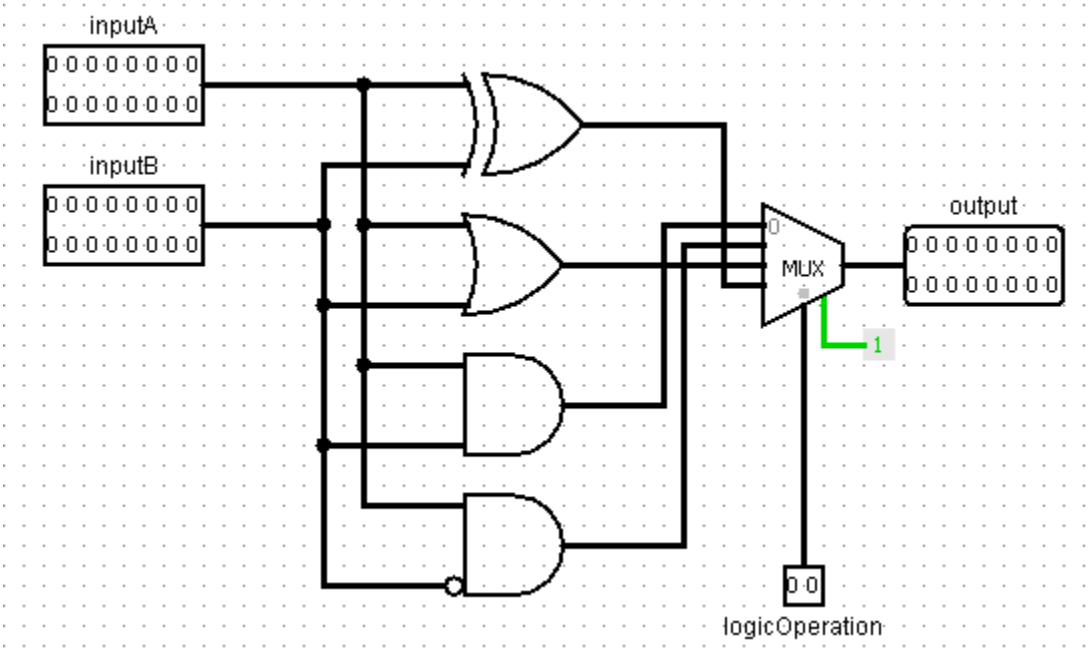


Figure 8: Logic Block

Memory Data

The memory data is a **Random-access memory** (RAM) device that allows data items to be read and written in approximately the same amount of time regardless of the order in which data items are accessed, and the block is already implemented in Logisim.

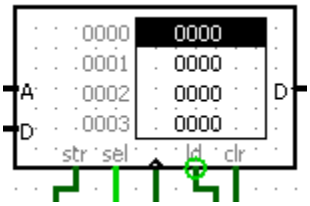


Figure 9: RAM

Next Pc Block

The PC in the default case is incremented by one, but in some cases, such as in branching and jumping, the PC jumps into a specified PC address to fetch a specific instruction. The **next Pc block** holds this task where it finds the next PC if any of the *jump* or *branch* instructions are fetched.

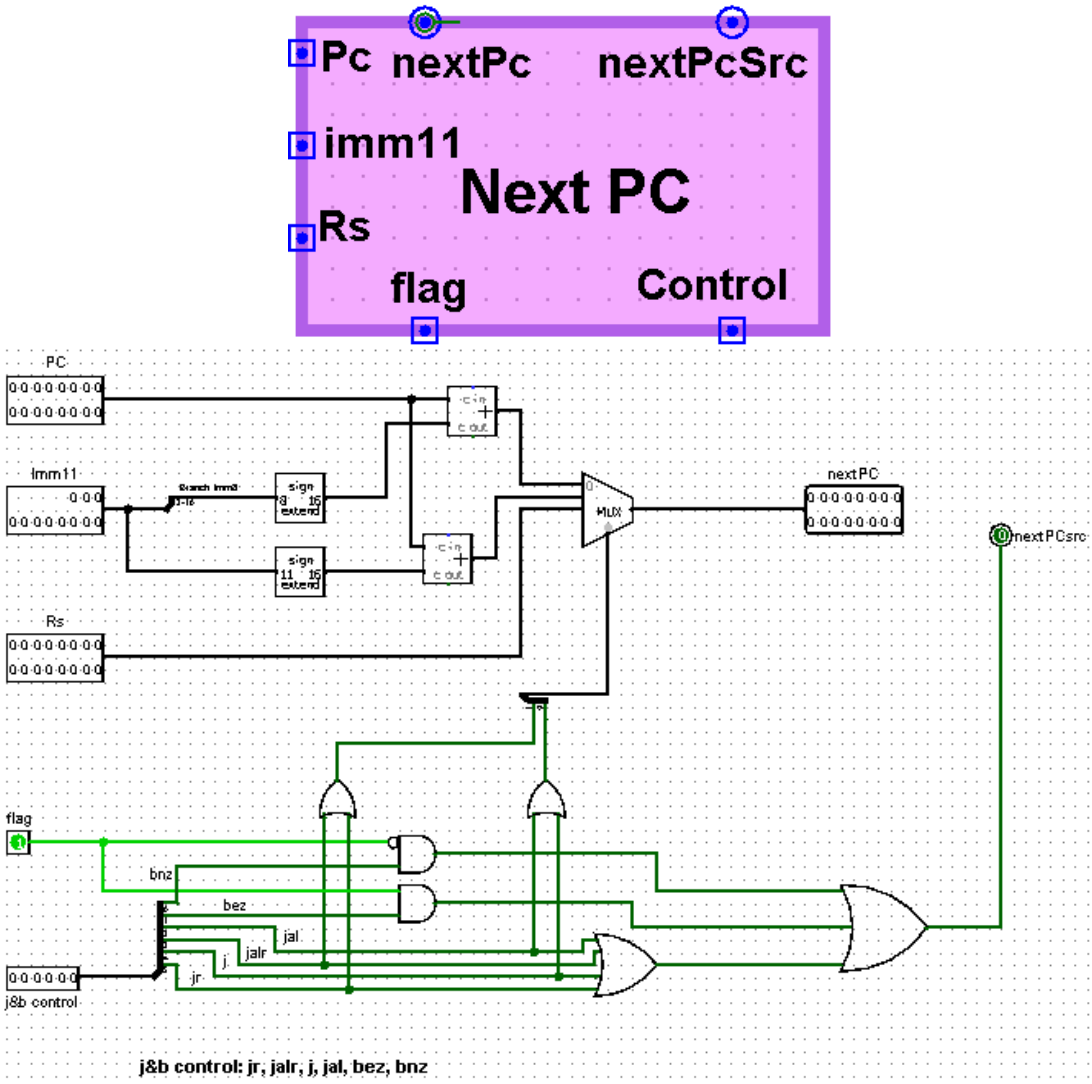


Figure 10: single cycle next PC

1- Pc = The current pc.
2- Imm11 = The immediate values (pc value) of the *jump* and *jump and link instructions* which consists of 11 bits.
3- Rs = explained above.
4- Flag = detects if the value of BusA is equal to ZERO.
5- Control =
6- nextPc = The next pc for the PC if any of the *jump* or *branch* instructions are detected.
7- nextPCSrc =

Set Block

As explained in a previous point there is block named File Register (FR) which save various values in different registers. However, to set certain value into the FR this block provides setting the 8th least significant bits (LSB) or most significant bits (MSB) into a certain register.

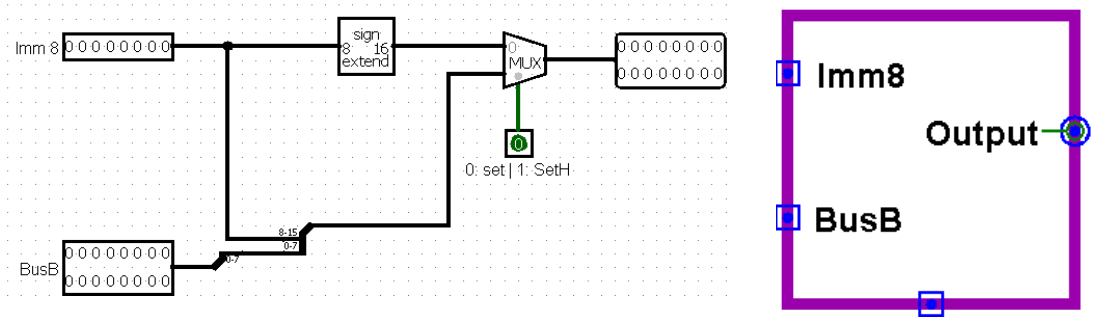


Figure 11: Set Block

1- Imm8 = The immediate value to set either from the MS 8 bits or form the LS 8 bits
2- BusB = The data of the destination Register to overwrite on it.
3- Output = The new data.

Control Unit:

The central control unit that is responsible for generating the control signals. It takes the OP code and the function and uses a decoder to generate the control signals. The ALU control signal is 8-bit and consists of: 2-bit Logic op, 2-bit shift op, 2-bit arithmetic op, 2-bit ALU op. The nextPC control signal is 6-bit and consists of: jr , jalr, j, jal, bez, bnz 1-bit each. The MemToReg signal is 2-bit and determines what will be written to the register Rw. Value 1 = ALU, value 2 = Memory, value 3 = SET block, value 4 = PC+1.

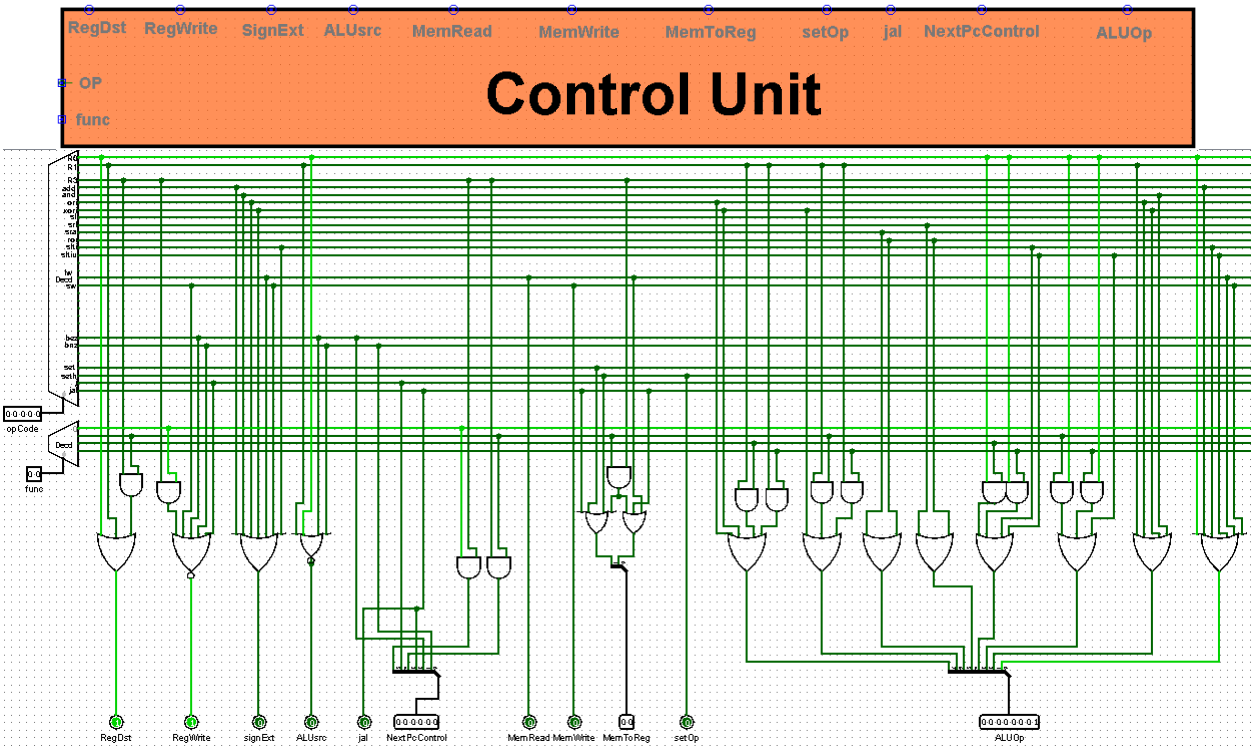


Figure 12: Control Unit

The Complete Single Cycle CPU Design

Finally after connecting all components together the single cycle CPU appears as in figure 11 below.

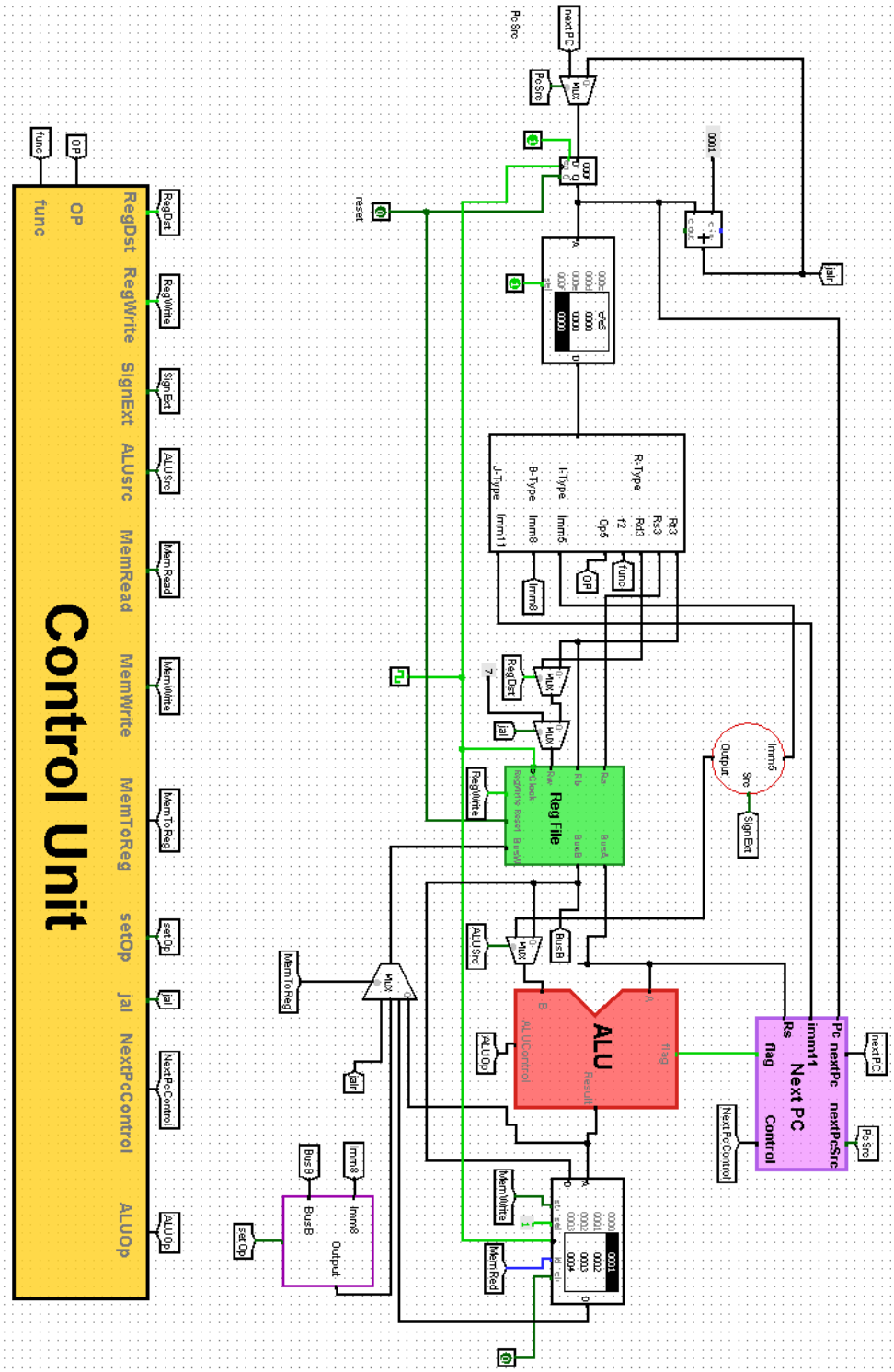


Figure 13: Single Cycle CPU

Testing Single Cycle CPU:

We conducted two tests, the first of which is basically testing all instructions together. Here we show the second test, which is creating an array of size 10 containing integer values, then summing these values and saving it in a register.

Address	0	1	2	3	4	5	6	7	8	9	Register value ($4)
Value	1	2	3	4	5	6	7	8	9	10	55

Assembly Code:

```
      addi 1 $1 $1            # $1  = 1
storing:
      sw 0 $2 $1              # mem($2 + 0) = $1
      slti 10 $2 $3           # if ($2 < 10) $3 = 1 else $3 = 0
      addi 1 $1 $1            # $1 = $1 + 1
      addi 1 $2 $2            # $2 = $2 + 1
      bnz -4 $3               # if ($3 != 0) branch to storing
      xor $3 $3 $3            # $3 = 0
      xor $2 $2 $2            # $2 = 0
loading:
      lw 0 $2 $3              # $3 = mem(0+$2)
      add $4 $4 $3            # $4 = $4 + $3
      addi 1 $2 $2            # $2 = $2 + 1
      slti 10 $2 $5           # if($2 < 10) $5 = 1 else $5 = 0
      bnz -4 $5               # if ($5 != 0) branch to loading
So $4 = 55
```

Hex code: 2049 8811 6293 2049 2052 cfe3 0edb 0e92 8013 0123 2052 6295 cfe5

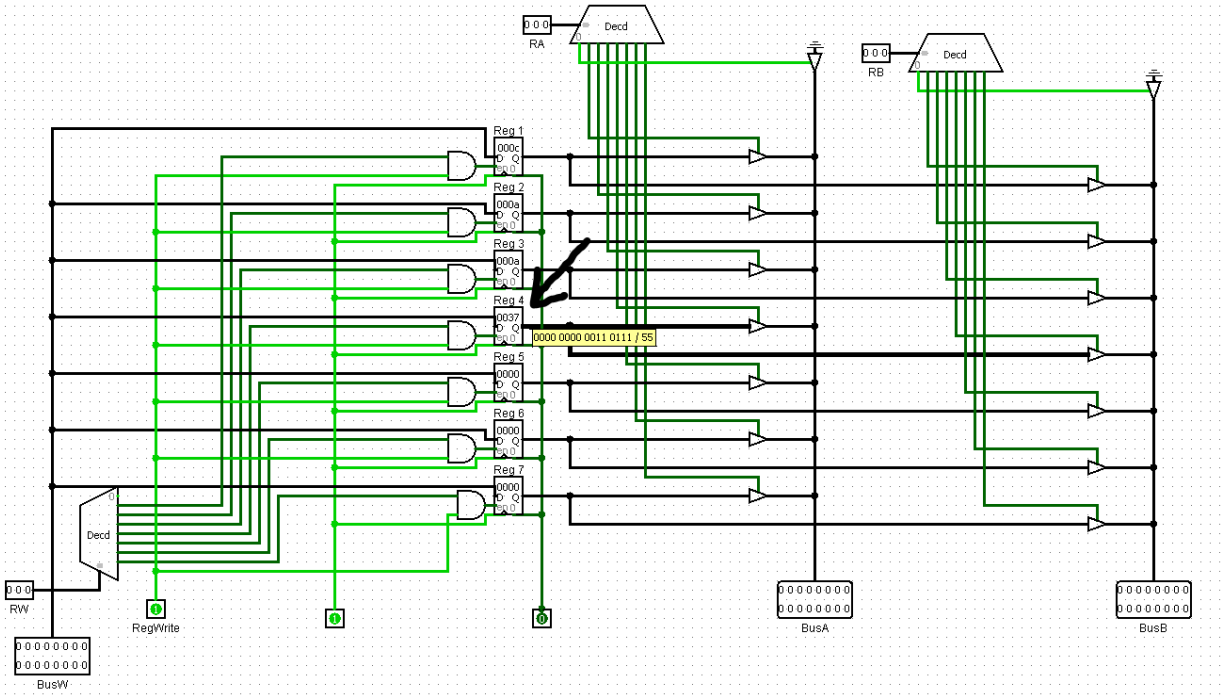


Figure 14: Test Value

Problems Detected After the Test:

1- Next PC:

Phase Two: Pipelining the CPU

The CPU is divided into four stages saving in every stage the required values and control signals, every stage has its own data and control register except the first stage it has now control signal register. However, stage has a distinct color:

1- Stage one (instruction decode): crème
2- Stage two (instruction fetch): blue
3- Stage three (execution/ALU): yellow
4- Stage four (write back): green

1) PC – IM

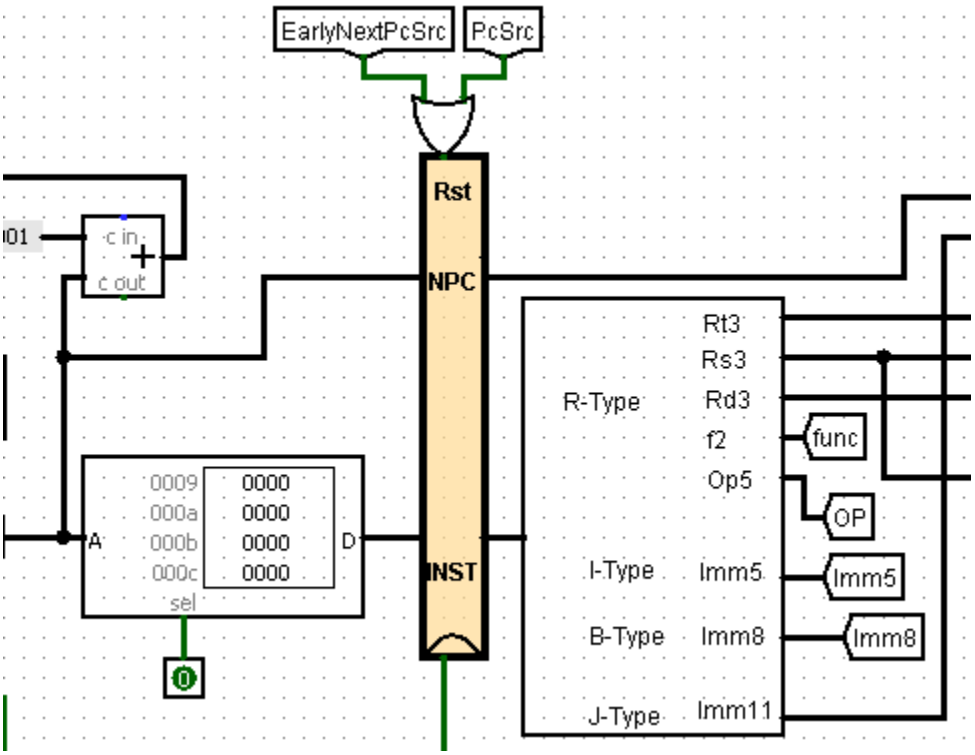


Figure 15: Pipeline Stage 1

2) IM – ALU

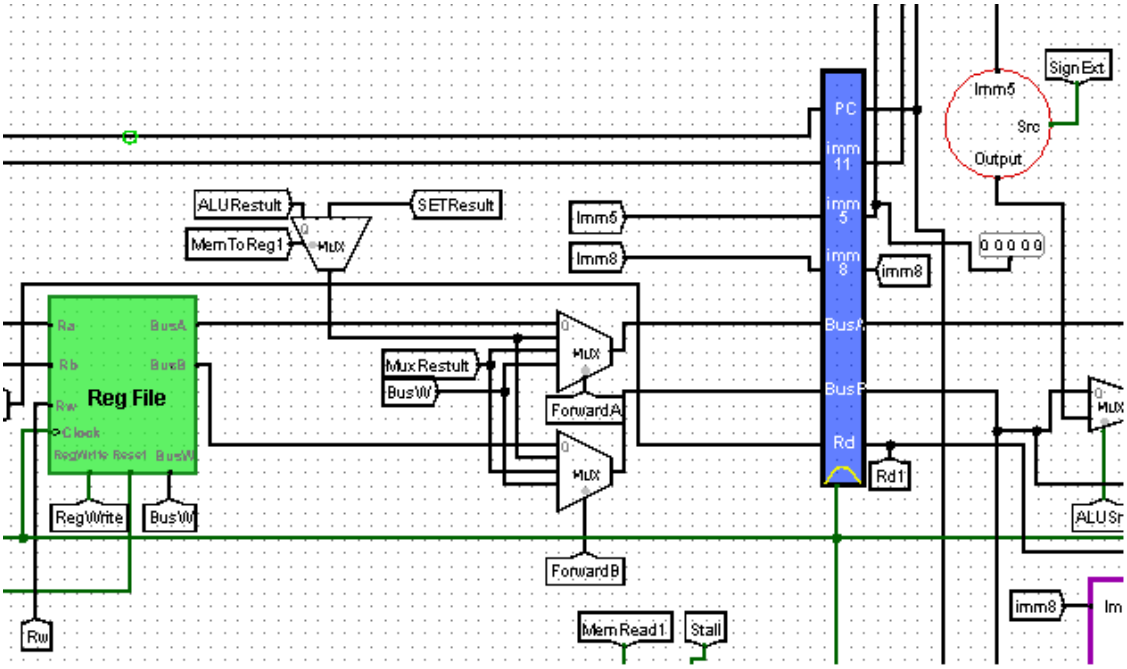


Figure 16: Pipeline Stage 2

3) ALU – DM

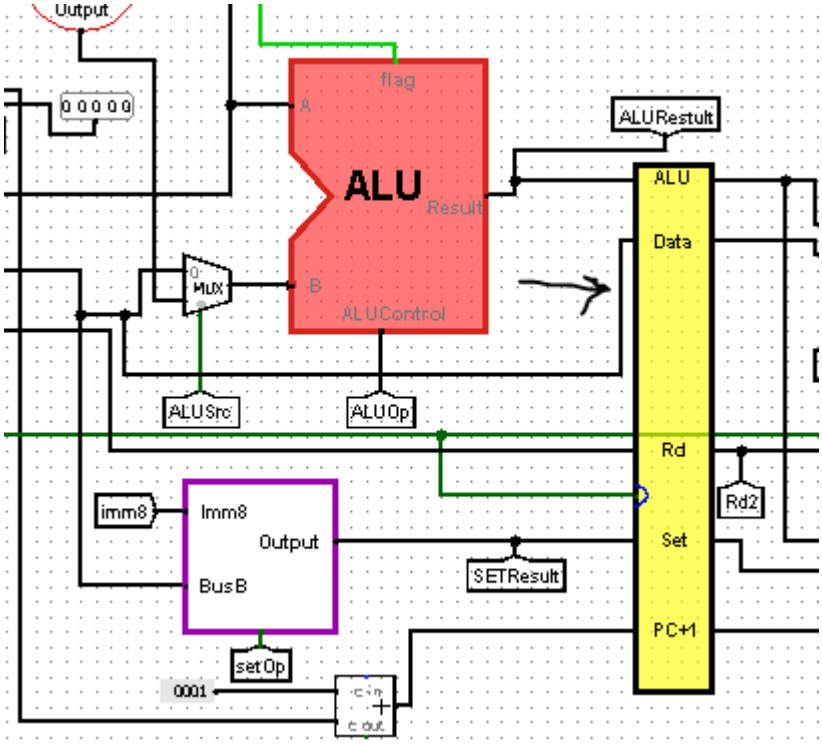


Figure 17: Pipeline Stage 3

4) DM – WB

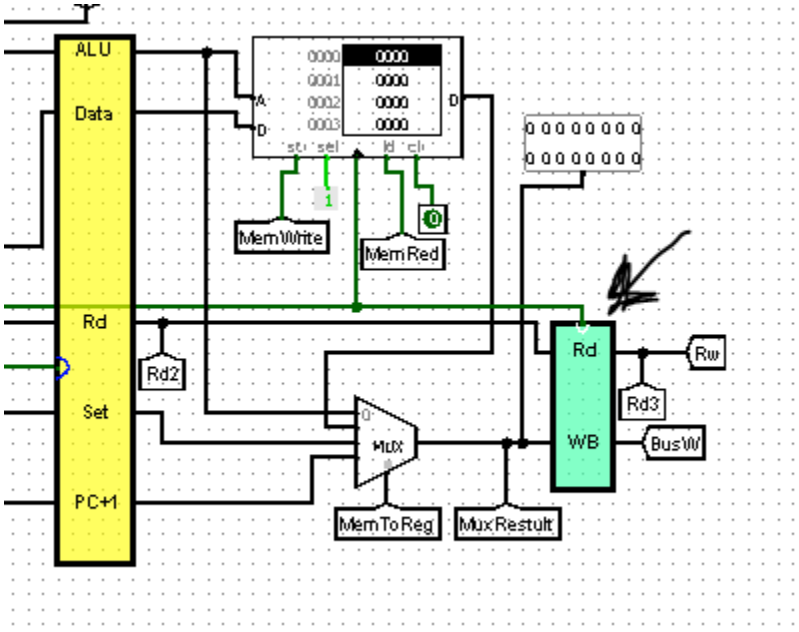


Figure 18: Pipeline Stage 4

Phase Three: Managing Data and Control Hazards

1) Data Hazards:

The data hazard means that an instruction is dependent on a previous instruction output that has still not have been written in the file register **RG**. For example, adding a value into register one, then adding two registers (register one and two) into a third register, notice that register three's result is dependent on register's one value. Anyhow, in our CPU design we have three situations where there is data dependency:

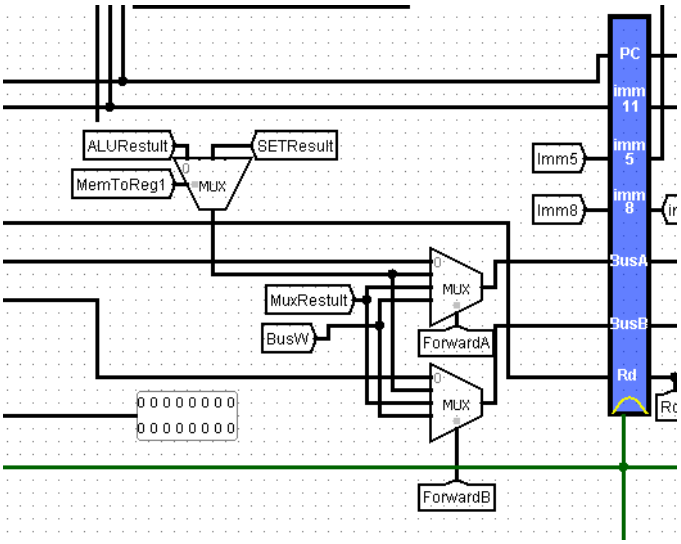
- Data is dependent on the value in the execution stage from either the **(ALU or SET blocks)** (Blue stage)
- Data is dependent on the value in the Memory Stage. (Yellow)
- Data is dependent on the value in the Write Back stage. (Green)

However, to detect a data dependency we check if the address of Rs or Rt is equal to the Rd address of next stages. However, to solve this problem we forward the data from the desired next stage into BusA or BusB.

The data is forwarded by a *mux* having four inputs:

1- Zero: for normal flow.
2- One: for execution dependency, either (ALU or SET block).
3- Two: for memory dependency.
4- Three: for write back dependency.

However, the forwarding is detected by the **Forwarding and Stall Control Unit Block (FSCU).** The FSCU selects the correct selection input for ForwardA and ForwardB mux's. Nevertheless, for the load word delay, we insert a bubble (no operation) and jam the pc for one clock.



MemToReg1: determines whether we are forwarding from the ALU or the SET block.

Figure 19: Forwarding The data to ALU stage

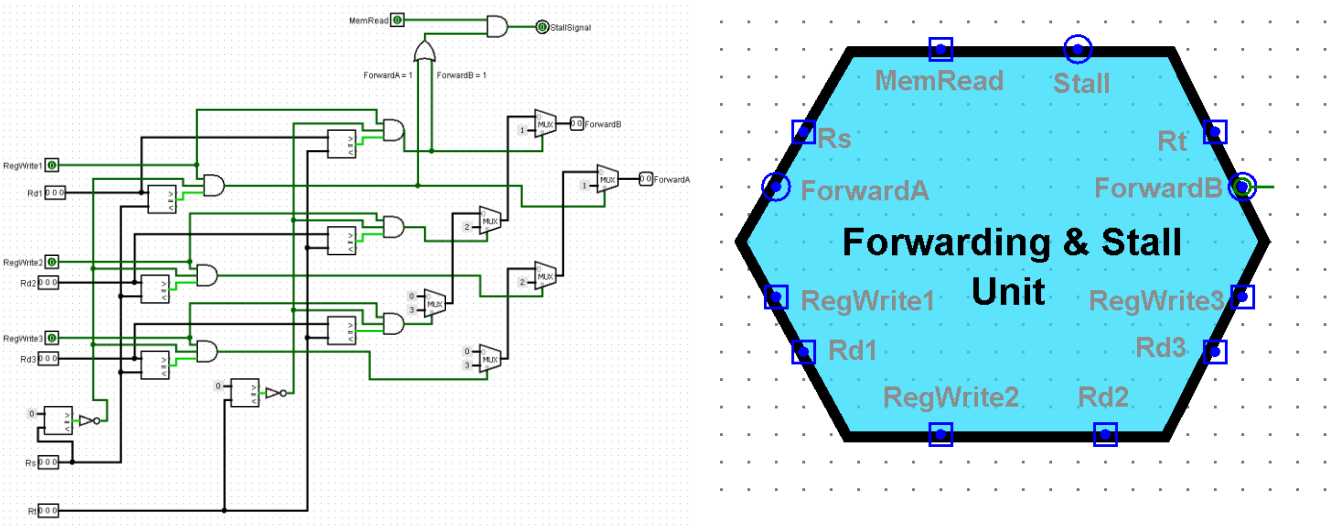


Figure 20: Forwarding and Stalling Unit

2) Control Hazards:

Jump and branch instructions (if taken) will cause some instructions to be loaded in the pipeline while they will not be executed. This will cause a penalty of two clock cycles since the next Pc block which changes the next address is at the ALU stage. As instructed, the jump penalty should be only one clock cycle. So, we separated the jump instructions from the next PC block to put them in a separate block in the instruction fetch stage and the other in the execution stage. Therefore, the next PC block has changed to only change the address for branch instructions. Therefore, the jump instructions now have a one clock cycle penalty and the branch instructions have a two clock cycle penalty as instructed. The figures below show the new next Pc block as well as the early jump detection block.

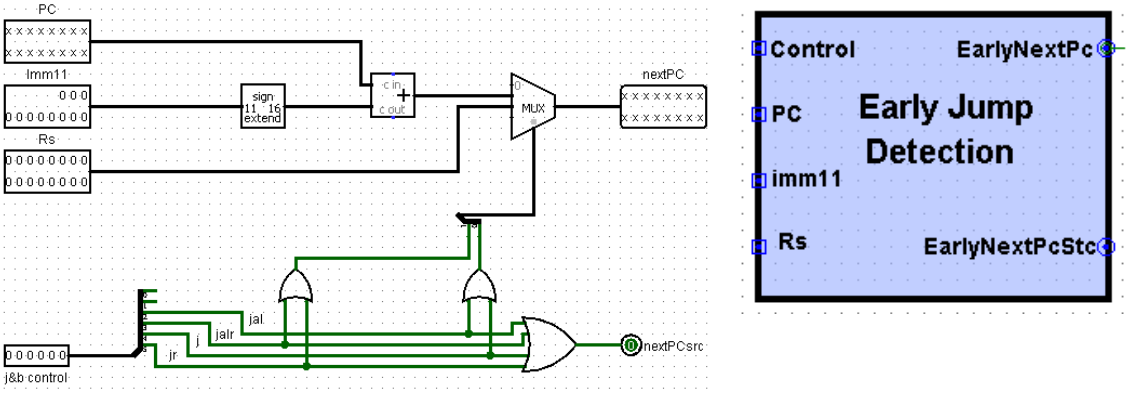


Figure 21: Early Jump Detection Block.

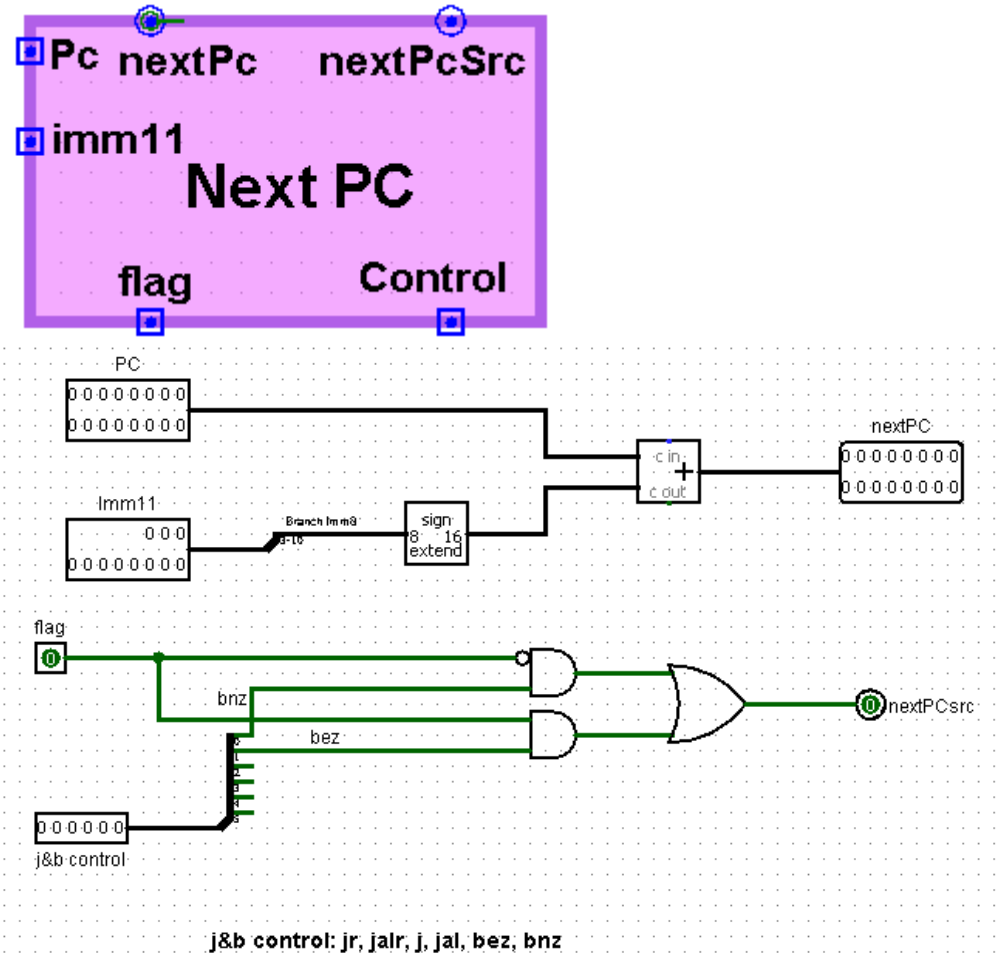


Figure 22: New next PC block.

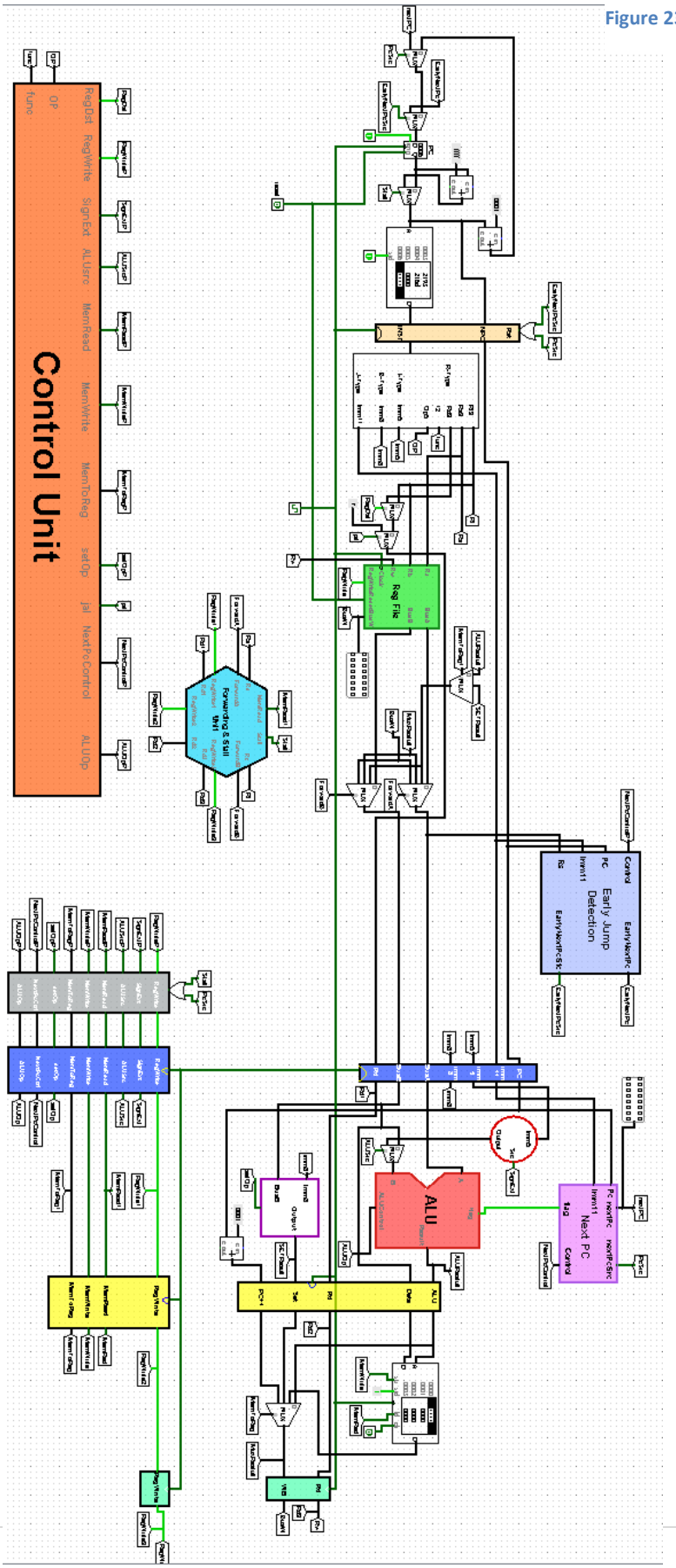


Figure 23: The final pipelined processor.

Final Testing:

Now we conclude a final test to test the pipelined processor with hazard detection. The following code snippet is for bubble sorting:

```
# This program creates an array of size 15 in the CPU memory
# starting at address 0 and sorts them using bubble sort.

# The algorithm is translated from: http://en.wikipedia.org/wiki/Bubble_sort
# Creating elements:
set 9 $1
sw 0 $0 $1

set 5 $1
sw 1 $0 $1

set -12 $1
sw 2 $0 $1

set -4 $1
sw 3 $0 $1

set 13 $1
sw 4 $0 $1

set 7 $1
sw 5 $0 $1

set 11 $1
sw 6 $0 $1

set 14 $1
sw 7 $0 $1

set 3 $1
sw 8 $0 $1

set 6 $1
sw 9 $0 $1

set 1 $1
sw 10 $0 $1
```

Final Test

```
set 15 $1
sw 11 $0 $1

set 2 $1
sw 12 $0 $1

set 8 $1
sw 13 $0 $1

set 10 $1
sw 14 $0 $1

# Bubble Sort:
set 15 $1   # $1 = n = array size
# loop1 until not swapped:
set 0 $2          # $2 = swapped = 0 = false

set 1 $3          # loop counter $3 = 1
      # loop2 for i = $3 = 1 to n - 1 inclusive:
      addi -1 $3 $4     # $4 = i - 1
      lw 0 $4 $4        # $4 = Array[i-1]
      lw 0 $3 $5        # $5 = Array[i]
      slt $6 $4 $5      # $6 = is Array[i-1] < Array[i]
      bnz 5 $6          # branch if Array[i-1] < Array[i]
# Swap:
      addi -1 $3 $6     # $6 = i - 1
      sw 0 $6 $5        # Array[i - 1] = Array[i]
      sw 0 $3 $4        # Array[i] = Array[i - 1] (old value)
      set 1 $2          # $2 = swapped = true

      addi 1 $3 $3      # i++
      slt $6 $3 $1      # $6 = is i < array size
      bnz -11 $6 # loop2
bnz -14 $2        # loop1
```

Results:

After loading the previous code in the instruction memory using Ahamd Tayeb's assembler, we ran the test. The test took about 9 seconds at clock tick rate of 512 Hz. As shown in the figure below, the inserted data are sorted in the memory.

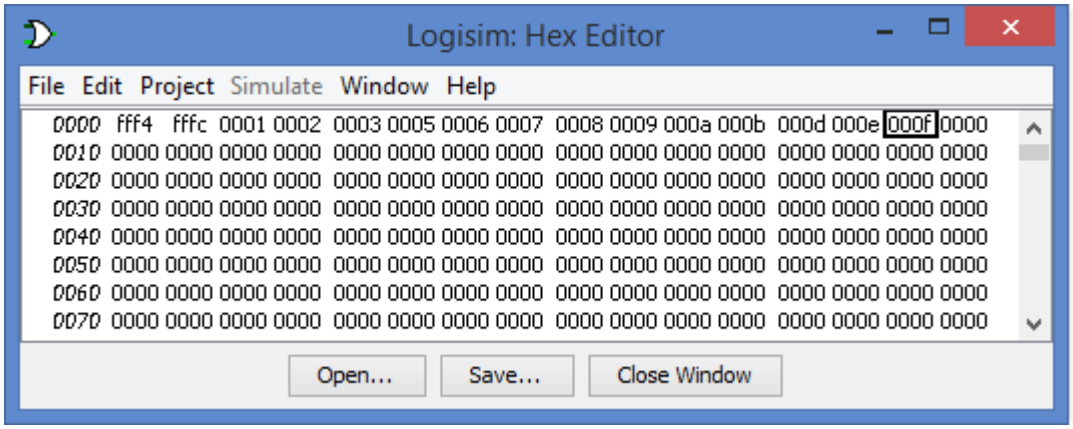


Figure 24: The final Memory Content after executing bubble sort.

Work Distribution:

Some parts were done by Mossad, some by Mohammad and the rest were done by both members. The following is a breakdown of the work:

1. Mohammad's parts:
    - Instruction fetch (PC + instruction memory + instruction splitter).
    - SET and Extend blocks.
    - Data memory.
    - Wiring the data path.
2. Mossad's parts:
    - ALU block.
    - Next Pc block.
    - Control Unit (circuit and functions).
3. Both members:
    - First stage of pipelining.
    - Data hazard detection and forwarding unit.
    - Memory stall detection.
    - Control hazard detection.
    - Final testing.

Problems Faced:

1. Next PC Control: we swapped **jalr** and **j** signals in the **CU.**
2. There was a problem in the **bnz** instruction flag detection
3. There was a problem in **load word delay**, where the stall detection was at injecting an instruction into the IM stage. To solve it we made the pc jam as the stall instruction is detected.
4. Another problem was when connecting wires into registers, the wires connects into wrong inputs without our knowledge.
5. The final part of pipelined branch and jump instruction was lost on the submission day and we had to make it again before submission.