RISC-32 5-Stage Pipeline Implementation

Arnav Kumar Behera, Vedanta Mohapatra

October 2022

1 Introduction

The pipeline that is implemented follows the 5-stage RISC Pipeline, which executes each instruction in a fixed time of 5 clock cycles. The Implementation has been done in Logisim-Evolution. For the sake of clarity, the whole circuit has been implemented in terms of smaller sub-circuits which made the integration of different parts of the circuit smooth and error-free. As a bonus, abstracting away the nitty-gritty details of the implementation of sub circuits like the register file, memory, ALU, etc., helps bring about the essential backbone of the pipeline and highlights the essential parts and how they interact with each other.

2 Supported Instruction Set

Here, we discuss the instructions supported by the pipeline and their respective OP-Codes and their encoding scheme.

Encoding Scheme			
Instruction Syntax	RTN	OP-Code	Encoding Scheme
MOV R_i, R_j	$R_i \leftarrow [R_j]$	000000	$[R_j]_{31-27}[R_i]_{26-22}[*]_{21-6}[OP]_{5-0}$
MVI R_i, X	$R_i \leftarrow X$	000001	$[*]_{31-27}[R_i]_{26-22}[X]_{21-6}[OP]_{5-0}$
LOAD $R_i, X(R_j)$	$R_i \leftarrow [X + [R_j]]$	000010	$[R_j]_{31-27}[R_i]_{26-22}[X]_{21-6}[OP]_{5-0}$
STORE $R_i, X(R_j)$	$R_i \rightarrow [X + [R_j]]$	000011	$[R_j]_{31-27}[R_i]_{26-22}[X]_{21-6}[OP]_{5-0}$
ADD R_i, R_j, R_k	$R_i \leftarrow [R_j] + [R_k]$	000100	$[R_j]_{31-27}[R_k]_{26-22}[R_i]_{21-17}[*]_{16-6}[OP]_{5-0}$
ADI R_i, R_j, X	$R_i \leftarrow [R_j] + X$	000101	$[R_j]_{31-27}[R_i]_{26-22}[X]_{21-6}[OP]_{5-0}$
SUB R_i, R_j, R_k	$R_i \leftarrow [R_j] - [R_k]$	000110	$[R_j]_{31-27}[R_k]_{26-22}[R_i]_{21-17}[*]_{16-6}[OP]_{5-0}$
SUI R_i, R_j, X	$R_i \leftarrow [R_j] - X$	000111	$[R_j]_{31-27}[R_i]_{26-22}[X]_{21-6}[OP]_{5-0}$
AND R_i, R_j, R_k	$R_i \leftarrow [R_j]\&[R_k]$	001000	$[R_j]_{31-27}[R_k]_{26-22}[R_i]_{21-17}[*]_{16-6}[OP]_{5-0}$
ANI R_i, R_j, X	$R_i \leftarrow [R_j]\&X$	001001	$[R_j]_{31-27}[R_i]_{26-22}[X]_{21-6}[OP]_{5-0}$
OR R_i, R_j, R_k	$R_i \leftarrow [R_j] \| [R_k]$	001010	$[R_j]_{31-27}[R_k]_{26-22}[R_i]_{21-17}[*]_{16-6}[OP]_{5-0}$
ORI R_i, R_j, X	$R_i \leftarrow [R_j] \| X$	001010	$[R_j]_{31-27}[R_i]_{26-22}[X]_{21-6}[OP]_{5-0}$
HLT		001100	$[*]_{31-6}[OP]_{5-0}$

The general Encoding Scheme followed is $[Src1][Src2(opt)][Dst(opt)][Imm][Op]$ where
$Scr1$= Address of the First Source register
$Scr2$= Address of the Second Source register(optional)
Dst= Address of the Destination register
Imm= 16-Bit Immediate Value
Op= Opcode of the Instruction

The position of the respective fields in the instruction is fixed, which allows the values(and addresses) to be loaded even before the instruction is decoded This allows us to complete the decoding and fetching of values from the register file in one clock cycle.

Let us now see how instructions can be encoded using our encoding scheme:
Consider the Instruction ADD R_1, R_2, R_3 , the opcode corresponding to ADD is 000100 (can be seen from the table), the first Source register's Address is 00010 (Since it is R_2), the second Source register's Address is 00011 (Since it is R_3), and the destination register's address is 00001 (Since it is R_1). The rest of the bits in the instruction have no significance, so they can be zero-padded.
Following the encoding scheme given in the table:

$$[R_j]_{31-27}[R_k]_{26-22}[R_i]_{21-17}[*]_{16-6}[OP]_{5-0}$$

we arrive at our instructions:

$$\overbrace{00010}^{\text{Src1}}\ \overbrace{00011}^{\text{Src2}}\ \overbrace{00001}^{\text{Dst}}\ \overbrace{00000000000}^{\text{0padding}}\ \overbrace{000100}^{\text{Op}}$$

The 32-Bit Decimal Code for the instruction is therefore:

$$00010000110000100000000000000100$$

Grouping 4-Bits at a time we get the equivalent hexadecimal code:

$$\underbrace{0001}_{1}\ \underbrace{0000}_{0}\ \underbrace{1100}_{C}\ \underbrace{0010}_{2}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0100}_{4}$$

Therefore the Equivalent Hexadecimal Instruction for ADD R_1, R_2, R_3 is 10C20004. A few other Encoding examples are given below:
ADI $R_1, R_2, 0BA0$: 1042E805
LOAD $R_1, 000A(R_2)$: 10400282
STORE $R1, 000B(R_2)$: 104002C3

Now we shall discuss the implementation of different Sub Circuits:

1. Register File
   - Register File of Size - 8
   - Integration of The 4-Files
2. ALU
3. PC, IR, Memory Circuitry
4. Control Unit
5. Assembling The Pipeline

3 Register File

The Register File has 32 General purpose Registers, which can be used to store and use intermediate values during the execution of a program.

According to the Pipeline Schematic, the register file needs to load two inter-stage Registers R_A and R_B as well as to write back to a register whose address is given to it as Address C (on receiving a RF_write signal) from another inter stage register R_Y. Since a 32-Register File would be too large to build and connect. We have implemented a smaller 8-register file and used 4 such files to combine into the final register file.

3.1 Register File of Size-8

Let us refer to the following implementation of the 8 register file:

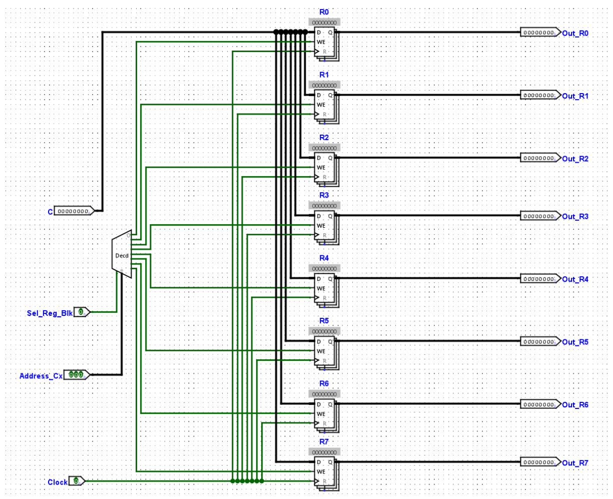


Figure 1: 8 Register File

There are 8 32-Bit registers in this register file, each of which is connected to the same clock, and the current values of each register are shown with output pins with appropriate names. There are 4 input pins: C (This is used to denote the value that will be written back to the register file on receiving the Rf_write signal) , $Clock$, $Address\_Cx$ which is used to enable which register to write back to (with the help of a decoder) upon receiving the RF_write signal, The enable bit of the decoder is named as Sel_Reg_Blk . In essence, this bit is vital to determine which of the 4 files we are writing back to (recall that our final register file would be made up of 4 of these sub circuits).
The block Diagram of our circuit, therefore, looks like

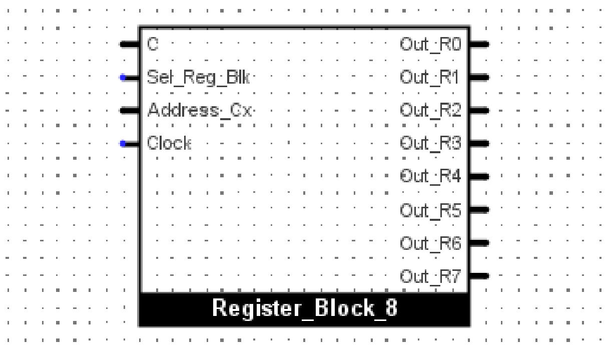


Figure 2: 8 Register File Block

Now let's discuss how we can combine the smaller sub circuits to form the final register file.

3.2 Integration of the 4 Files

The Output of 2 registers can be taken from the register files (to be fed to R_A and R_B) quite easily by using 2 multiplexers with their respective select bits as Address of A and Address of B. This has been shown below:

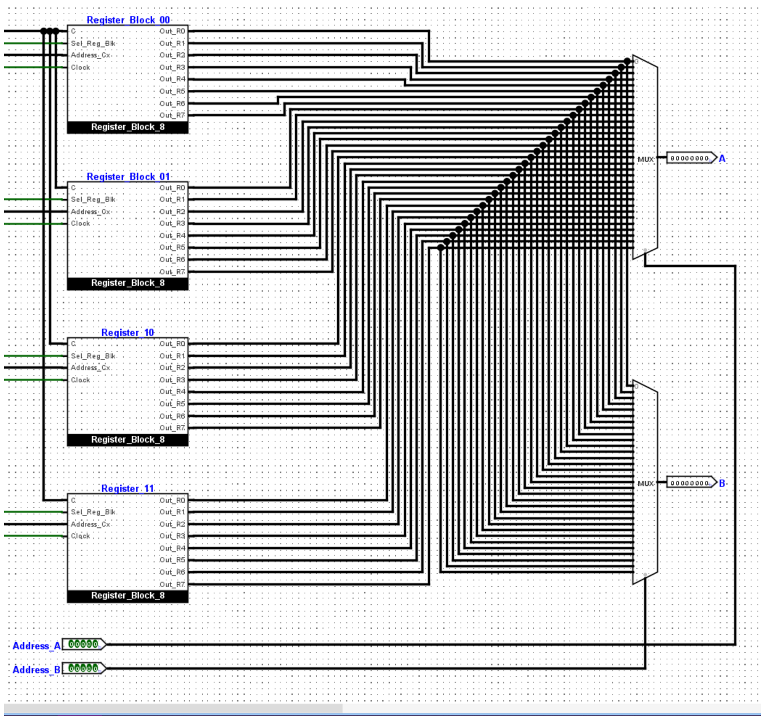


Figure 3: Multiplexing the Output

The Pressing issue is how to decide which register file to write back to. We address this issue by asserting that the first 2 bits from left (MSB) of the address of the destination register decide which register file we write back to, and all other register files' write_enable signal should be turned off. This has been done with the help of a decoder. The Enable Bit of that decoder is taken as the RF_write signal (if it is off, then so are the outputs of the decoder, and consequently, so are the write enable signals of the registers in the register files). The remaining 3 Bits in the address of the destination register decide which register inside the sub-circuit we write back to. This has been implemented as shown below:

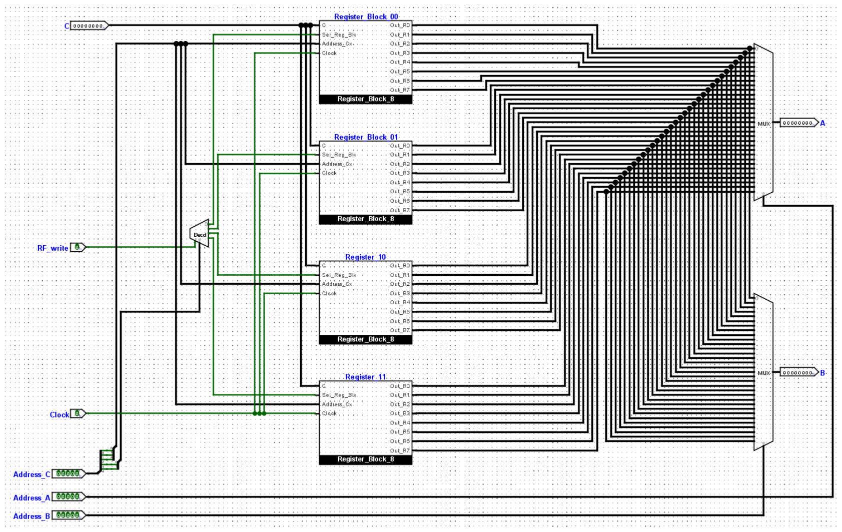


Figure 4: Final Register File

Abstracting away the complex implementation of the register file, we use the block diagram (as shown below) in all of our further discussions about the pipeline and shall only be concerned with how it interacts with other components.

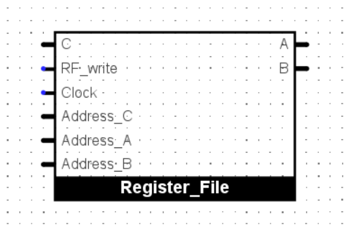


Figure 5: Final Register File Block

4 ALU

The Implementation of ALU that we use is a simple one. The ALU takes 2 inputs InA and InB; these inputs are 32 bits in size, then the ALU computes the sum, difference, bitwise AND, and bitwise OR of the two operands and finally, a multiplexer selects the output we need for the particular Instruction that it is executing as shown below:

Since the adder, subtractor, Multiplexer, etc, are passive components in Logisim-Evolution, we do not need to power them up in order to use them. Therefore they can be assumed to be left functioning for the entirety of the program execution. As we shall soon see, this makes implementing the control signals for the ALU much easier. The Block Diagram for the ALU is given below:

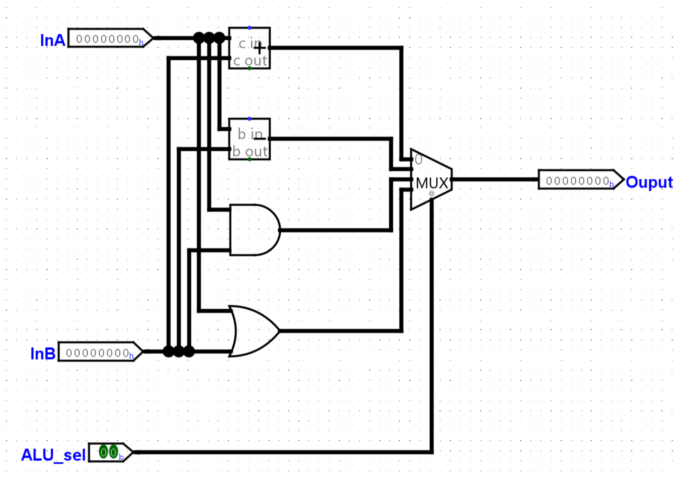


Figure 6: ALU Circuit

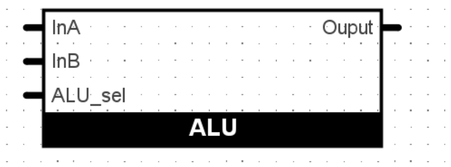


Figure 7: ALU Block

5 PC, IR, Memory Circuitry

This part of the circuitry deals with three important circuits which form the basis for reading/writing to/from memory and storing and decoding instructions, as well as keeping track of where the instructions are located. As such, they are interdependent to a high degree, to the point that we are forced to put them into a single sub-circuit. But, we shall, nevertheless, take up each circuitry one by one and understand their implementation:

5.1 Memory Interface

Instead of implementing the Memory Interface from scratch, the implementation of memory that we use here is the Logisim-Evolution RAM (Figure given below), which suffices for the needs of our pipeline.

The functioning of the Memory circuitry is as follows: the *Address* pin is used to point to the address from which we read/write. Reading/Writing to memory is done within one clock cycle. Furthermore, for reading/writing to take place, the respective read_enable or write_enable bits must be set. This has been taken care of by the control unit.

5.2 PC Circuitry

Refer to the figure given below:

The job of the PC register is to point to the memory address where the next instruction is located. The Circuitry for the PC register (Program Counter) is relatively simple. The adder is provided with its output looped to the input of the PC register, which allows us to increment the PC register by 1 during every instruction. Recall that each instruction is executed in 5 clock cycles. Does that mean the PC will be incremented 5 times during each instruction? The role of *PC_enable* is to ensure that the PC is incremented only once during the first clock cycle, for which it shall remain ON (this will also be managed by the Control Unit). The Current contents of the PC register are fed to the address of the memory unit. We do not implement registers like PC temp and a variable increment of the PC register since we do not support instructions like CALL and BRANCH.

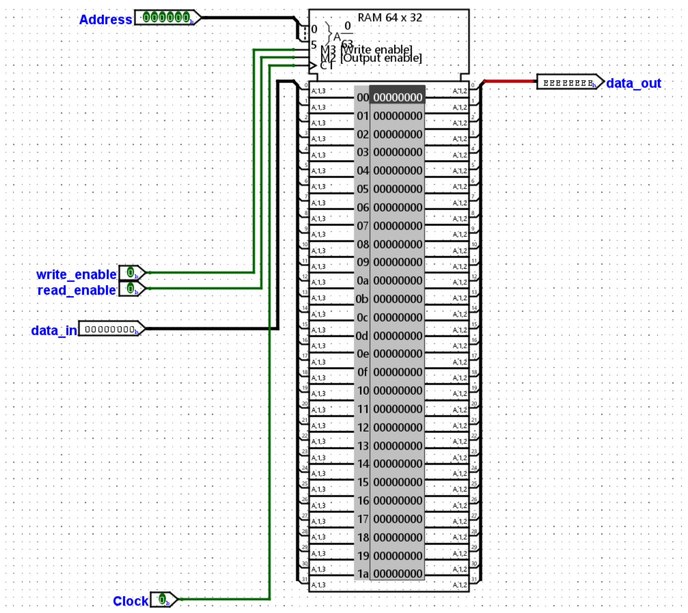


Figure 8: Memory

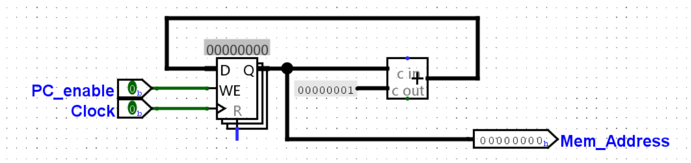


Figure 9: PC Circuitry

5.3 IR Circuitry

Refer to the figure given below:

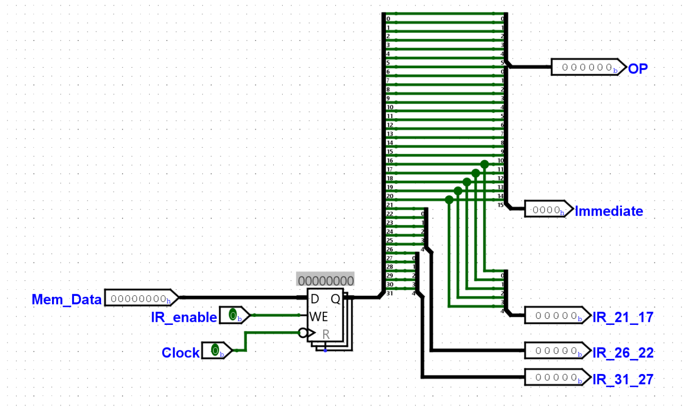


Figure 10: IR Circuitry

The Instruction Register (IR) is responsible for storing the instruction fetched from memory pointed to by the PC register. The *IR_enable* signal is responsible for writing to the IR only during the first clock cycle of the Instruction. This signal is handled by the Control unit as well. The contents of the IR are segmented out to be fed to other parts of the circuit like address of A, Address of B, Immediate Values, OP-code is fed to the CU, etc.

5.4 Combining the Three Circuits

There are 3 ways in which we might access the data from memory: Reading from memory during the loading of the IR register (Instruction register) from the address mentioned in the PC register(program counter), Reading From Memory during a LOAD instruction from the effective address stored in the R_Z register, of writing to the memory at the effective address stored in R_Z with the data stored in R_M. From our discussion, we note that the address that is fed to the memory can be one of 2 values: contents of the PC register or the Contents of the R_Z register. We can therefore choose a single address using a multiplexer and the select bit of that MUX can be controlled by the Control Unit.

Now let's focus on what happens to the data that is read from the memory, it is used in 2 ways: the first is to load the IR register, and the other way is when we perform a LOAD instruction where the data is fed to the R_Y register.

Another issue is that we require the PC register to be incremented, Read the memory at the address stored in the PC, and load the IR all in one clock cycle. So we may increment the PC and read from the memory in one rising edge one of the clock since they can be synchronized. But the issue of loading the IR still remains since reading from memory requires one clock cycle, and only after reading, we can load the IR register. One way to deal with the issue is to load the IR on the negative edge of the same clock cycle. This is the approach that we have taken here. A similar issue arises when we focus on what happens during the 4^{th} Clock Cycle of the LOAD instruction, the memory read function happens during the rising edge of the clock cycle, so there is no time to load the R_Y register. We have taken a similar approach to resolve this issue as well, i.e. to load the R_Y register during the negative edge of the clock cycle. This assures that the writing of the read memory data to R_Y is complete within one clock cycle.

The Operation of HLT Instruction is to reset the contents of registers PC, IR to 0 (execution begins from 0) and stop the program execution by somehow disabling the clock (this has been discussed later). We have implemented a *hlt* signal that is decoded from the IR register as soon as it is loaded. This signal is used to reset the PC and IR registers and also disable the clock(the mechanism for which will be discussed later) Keeping these notes in mind; we arrive at the following implementation:

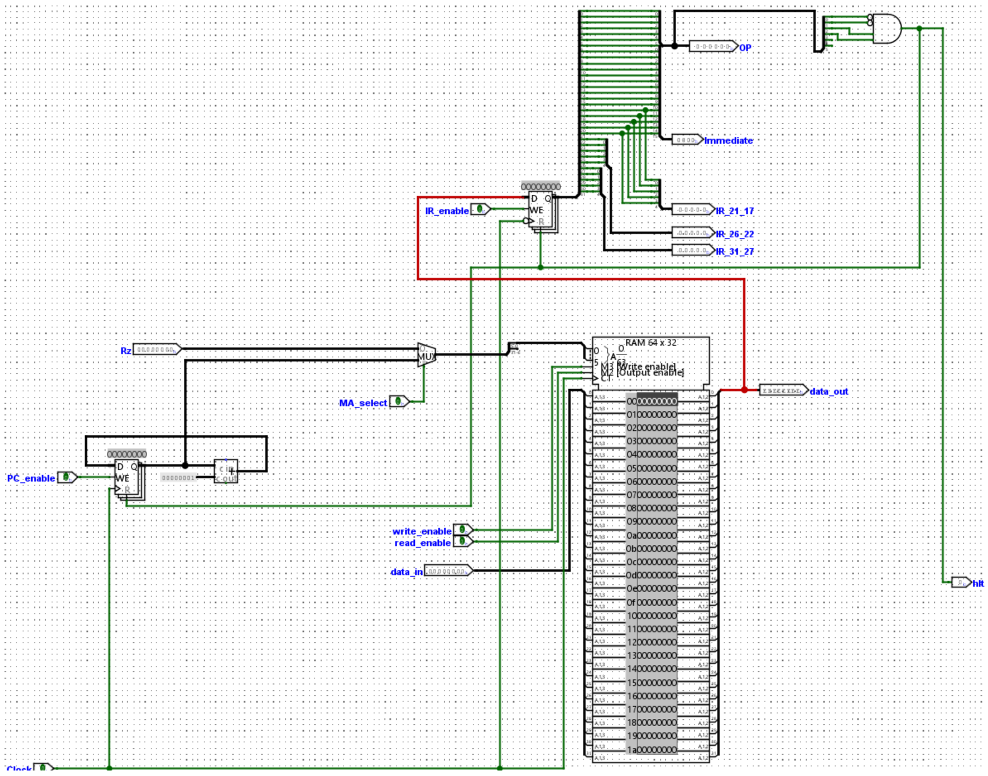


Figure 11: IR,PC,Memory Circuitry

The Block Diagram of the Combined Circuit is given below:

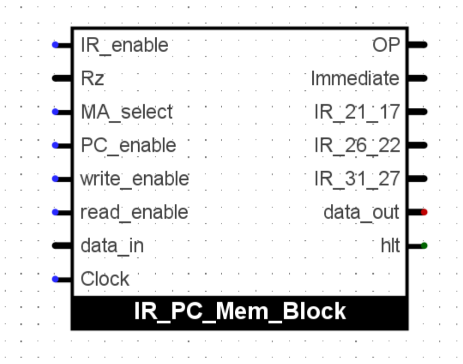


Figure 12: IR,PC,Memory Circuitry

6 Control Unit

Now, we have arrived at the most important part of our pipeline: The Control Unit. This circuit is responsible for decoding instructions and generating appropriate control signals during appropriate clock cycles. We can do this in the following way:
The CU takes 2 inputs: OP-Code of the instruction in the IR and the clock. The clock is used to operate a counter which counts the clock cycles and resets at the 6^{th} Cycle. It can be seen that only these inputs are sufficient to generate the signals for the rest of the pipeline. So, we first try to figure out which signals are generated during which clock cycle for the various instructions in our instruction set.

6.1 Output Signals

The Output signals that we wish to generate are

1. *RF_write*: When high, Register file writes back to a register whose address is given in *Address_C* with the data stored in R_Y
2. *C_select*: This is the select bit of the multiplexer, which chooses which address to pick for writing back to RF. The possible addresses where we can write back to are IR_{21-17} and IR_{26-22}, which are the positions where the address of the destination is given (Recall the Encoding Scheme)
3. *B_select*: This is the select bit of the multiplexer that chooses whether to send the value of R_B or the sign extended IR_{21-6} value (Immediate value) to the ALU.
4. *ALU_select*: Selects which of the outputs of different operations in ALU are selected and sent to R_Z
5. *move_select*: This controls the select bit of the multiplexer that is used to bypass the ALU. This multiplexer chooses between 2 values: R_A and the forwarded value of MUXB .
6. *Z_select*: This is the Select bit of MUXZ; this multiplexer selects between the output of the ALU or the output of MUX that bypasses the ALU.
7. *Y_select* : This is the select bit of MUXY which selects between 2 values: R_Z and Mem_data and forwards the output to R_Y.
8. *IR_enable*: This signal is responsible for writing to the IR register.
9. *PC_enable*: This signal is responsible for writing to the PC register.
10. *MA_select* : This is the select bit for the MUXMA which selects which address to forward to the *MEM_address* : R_Z or PC .

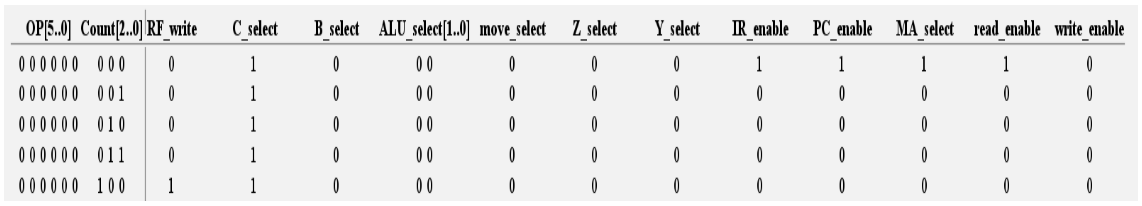
11. *read_enable*: This controls the read_enable signal of the memory interface.
12. *write_enable*: This controls the write_enable signal of the memory interface.

6.2 Signals for Different Instructions

In this section, we see which signals are generated during which clock cycles for different instructions of our instruction set: The Clock cycles are numbered 0 through 4 and 3-bit numbers are used to denote them.

6.2.1 MOV

Refer to the truth table given Below:



OP[5..0]	Count[2..0]	RF_write	C_select	B_select	ALU_select[1..0]	move_select	Z_select	Y_select	IR_enable	PC_enable	MA_select	read_enable	write_enable
0 0 0 0 0 0	0 0 0	0	1	0	0 0	0	0	0	1	1	1	1	0
0 0 0 0 0 0	0 0 1	0	1	0	0 0	0	0	0	0	0	0	0	0
0 0 0 0 0 0	0 1 0	0	1	0	0 0	0	0	0	0	0	0	0	0
0 0 0 0 0 0	0 1 1	0	1	0	0 0	0	0	0	0	0	0	0	0
0 0 0 0 0 0	1 0 0	1	1	0	0 0	0	0	0	0	0	0	0	0

Figure 13: Truth Table for MOV

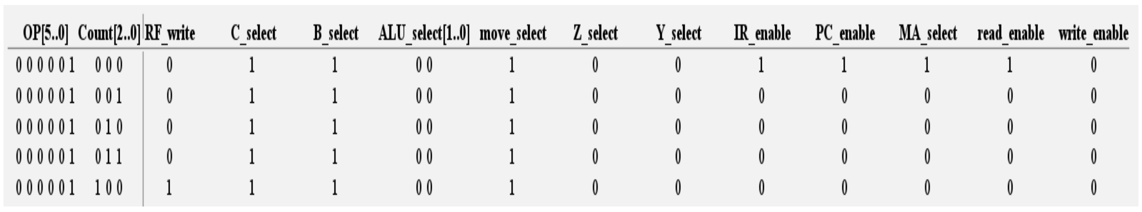
First, we note that the OP-code for MOV is 000000. From the table we can infer:

1. *RF_write* is high only during the 5th clock cycle (during write back).
2. *C_select* is set to 1 which selects the IR_{26-22} address.
3. *B_select* is a don't care bit as our final value comes from R_A only
4. *ALU_select* is also a don't care bit.
5. *move_select* is set to 0 which selects the R_A value.
6. *Z_select* is set to 0, indicating we want to choose the output of the bypassing MUX.
7. *Y_select* is set to 0, indicating we want to forward the value of R_Z to R_Y
8. *IR_enable* is set to 1 only during the first clock cycle when we are fetching the contents of IR.
9. *PC_enable* : is set to 1 only during the first clock cycle when we are incrementing the contents of PC.
10. *MA_select* is set to 1 only during the first clock cycle to select the *PC* contents.
11. *read_enable* is set to 1 only during the first clock cycle(for reading the IR contents from memory). This happens for all instructions.
12. *write_enable* is set to 0 always as we are never writing to memory.

Since a lot of the signals are repeated for all instructions, such as *IR_enable* and *PC_enable*, we omit their discussion for the remaining instructions. We also omit the discussion of the don't care bits to keep it brief.

6.2.2 MVI

Refer to the truth table given Below:



OP[5..0]	Count[2..0]	RF_write	C_select	B_select	ALU_select[1..0]	move_select	Z_select	Y_select	IR_enable	PC_enable	MA_select	read_enable	write_enable
0 0 0 0 0 1	0 0 0	0	1	1	0 0	1	0	0	1	1	1	1	0
0 0 0 0 0 1	0 0 1	0	1	1	0 0	1	0	0	0	0	0	0	0
0 0 0 0 0 1	0 1 0	0	1	1	0 0	1	0	0	0	0	0	0	0
0 0 0 0 0 1	0 1 1	0	1	1	0 0	1	0	0	0	0	0	0	0
0 0 0 0 0 1	1 0 0	1	1	1	0 0	1	0	0	0	0	0	0	0

Figure 14: Truth Table for MVI

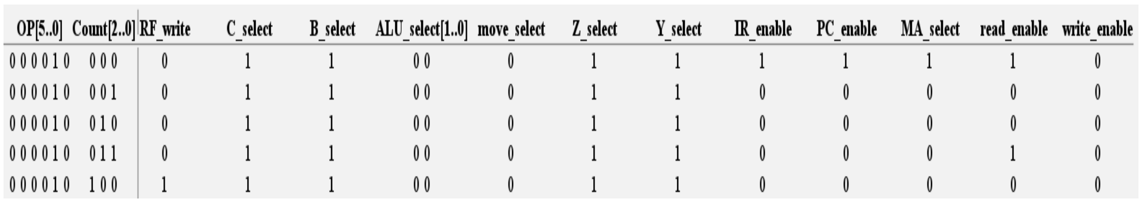
We note that the OP-code for MVI is 000001. From the table, we can infer:

1. *RF_write* is high only during the 5th clock cycle (during write back).
2. *C_select* is set to 1 which selects the IR_{26-22} address.
3. *B_select* is set to 1 since we select the Immediate value.
4. *move_select* is set to 1 which selects the *Immediate* value that comes from MUXB.
5. *Z_select* is set to 0, indicating we want to choose the output of the bypassing MUX.
6. *Y_select* is set to 0, indicating we want to forward the value of R_Z to R_Y

Rest of the control signals are similar to MOV.

6.2.3 LOAD

Refer to the truth table given Below:



OP[5..0]	Count[2..0]	RF_write	C_select	B_select	ALU_select[1..0]	move_select	Z_select	Y_select	IR_enable	PC_enable	MA_select	read_enable	write_enable
0 0 0 0 1 0	0 0 0	0	1	1	0 0	0	1	1	1	1	1	1	0
0 0 0 0 1 0	0 0 1	0	1	1	0 0	0	1	1	0	0	0	0	0
0 0 0 0 1 0	0 1 0	0	1	1	0 0	0	1	1	0	0	0	0	0
0 0 0 0 1 0	0 1 1	0	1	1	0 0	0	1	1	0	0	0	1	0
0 0 0 0 1 0	1 0 0	1	1	1	0 0	0	1	1	0	0	0	0	0

Figure 15: Truth Table for LOAD

We note that the OP-code for LOAD is 000010. From the table, we can infer:

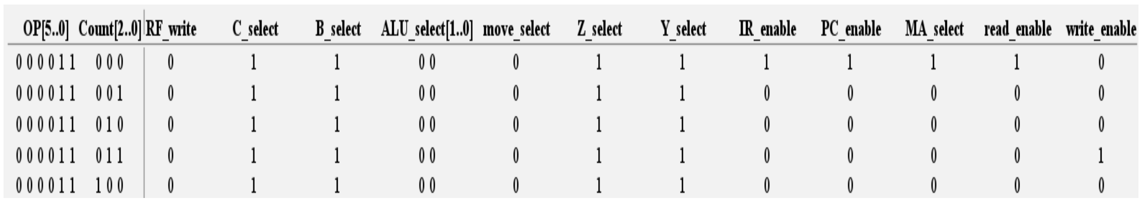
1. *RF_write* is high only during the 5th clock cycle (during write back since we are loading a register).
2. *C_select* is set to 1 which selects the IR_{26-22} address.
3. *B_select* is set to 1 since we select the Immediate value.
4. *ALU_select* is set to 00, indicating we want to select the result of the addition of the base register and offset.
5. *Z_select* is set to 1, indicating we want to choose the output of the ALU.
6. *Y_select* is set to 1, indicating we want to forward the value of *MEM_data* to R_Y

7. *MA_select* is set to 1 only during the first clock cycle, and for the rest 4 clock cycles, is set to 0, which selects the value of R_Z. This is sufficient for all instructions as every instruction reads the memory at PC address during the first clock cycle, and we are free to do whatever we want after that. Therefore the description of this signal is the same for every instruction, and we shall omit this from now onward.

8. *read_enable* is set to 1 only during the 1st and 4th clock cycle(during reading from memory).

6.2.4 STORE

Refer to the truth table given Below:



OP[5..0]	Count[2..0]	RF_write	C_select	B_select	ALU_select[1..0]	move_select	Z_select	Y_select	IR_enable	PC_enable	MA_select	read_enable	write_enable
000011	000	0	1	1	00	0	1	1	1	1	1	1	0
000011	001	0	1	1	00	0	1	1	0	0	0	0	0
000011	010	0	1	1	00	0	1	1	0	0	0	0	0
000011	011	0	1	1	00	0	1	1	0	0	0	0	1
000011	100	0	1	1	00	0	1	1	0	0	0	0	0

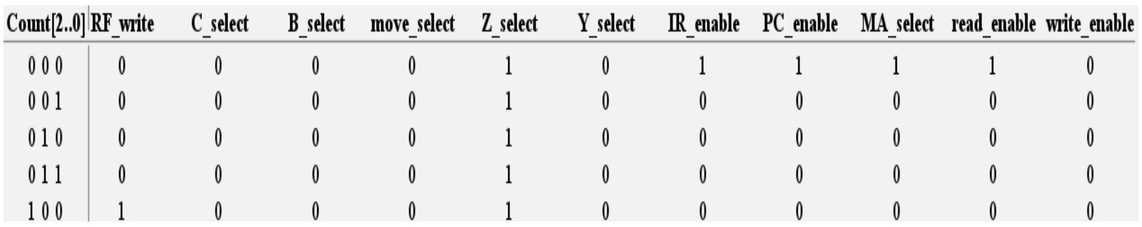
Figure 16: Truth Table for STORE

We note that the OP-code for STORE is 000011. From the table, we can infer:

1. *RF_write* is 0 for all clock cycles since we do not write back to the Register file at any stage.
2. *B_select* is set to 1 since we select the Immediate value to forward to ALU.
3. *ALU_select* is set to 00, indicating we want to select the result of the addition of the base register and offset.
4. *Z_select* is set to 1, indicating we want to choose the output of the ALU.
5. *MA_select* is set to 0 for all except the first clock cycle, which selects the R_Z value as MEM_address.
6. *write_enable* is set to 1 only during the 4th clock cycle when we write from the R_M register to the address given by R_Z.

6.2.5 ADD,SUB,AND,OR

These instructions are almost the same with respect to the control signals, except for the *ALU_select* signal. Therefore we discuss them collectively in this section. Refer to the truth Table given below:



Count[2..0]	RF_write	C_select	B_select	move_select	Z_select	Y_select	IR_enable	PC_enable	MA_select	read_enable	write_enable
000	0	0	0	0	1	0	1	1	1	1	0
001	0	0	0	0	1	0	0	0	0	0	0
010	0	0	0	0	1	0	0	0	0	0	0
011	0	0	0	0	1	0	0	0	0	0	0
100	1	0	0	0	1	0	0	0	0	0	0

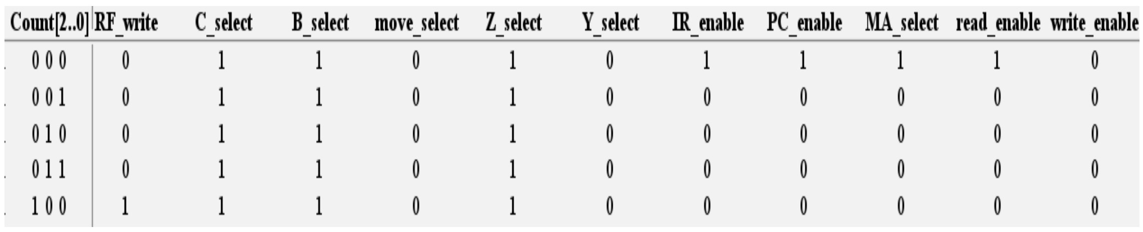
Figure 17: Truth Table for ADD-Type instructions

From the Truth Table, we can infer that:

1. RF_write is 1 only for the 5^{th} clock cycle when we write back to the register file.
2. B_select is set to 0 since we select the R_B value to forward to ALU.
3. ALU_select is set to 00 for ADD, 01 for SUB, 10 for AND, and 11 for OR.
4. Z_select is set to 1, indicating we want to choose the output of the ALU.
5. Y_select is set to 0, indicating we want to forward the value of R_Z to R_Y

6.2.6 ADI,SUI,ANI,ORI

These instructions are almost the same with respect to the control signals, except the ALU_select signal. Therefore we discuss them collectively in this section. Refer to the truth Table given below:



Count[2..0]	RF_write	C_select	B_select	move_select	Z_select	Y_select	IR_enable	PC_enable	MA_select	read_enable	write_enable
0 0 0	0	1	1	0	1	0	1	1	1	1	0
0 0 1	0	1	1	0	1	0	0	0	0	0	0
0 1 0	0	1	1	0	1	0	0	0	0	0	0
0 1 1	0	1	1	0	1	0	0	0	0	0	0
1 0 0	1	1	1	0	1	0	0	0	0	0	0

Figure 18: Truth Table for ADI-Type instructions

From the Truth Table, we can infer that:

1. RF_write is 1 only for the 5^{th} clock cycle when we write back to the register file.
2. B_select is set to 1 since we select the sign extended IR_{21-6} value (Immediate value) to forward to ALU.
3. ALU_select is set to 00 for ADI, 01 for SUI, 10 for ANI, and 11 for ORI.
4. Z_select is set to 1 indicating we want to choose the output of the ALU.
5. Y_select is set to 0 indicating we want to forward the value of R_Z to R_Y

6.2.7 HLT

The circuitry for HLT is already explained in the PC, IR, and Memory Circuitry. Let's discuss how the *hlt* signal can be used to stop the execution of the program. As there is no provision to stop the clock signal, we can artificially tie the clock to 0 voltage by using a AND gate. This has been shown below:

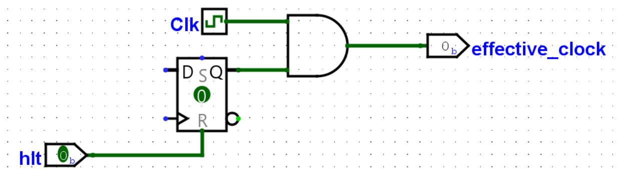


Figure 19: Stopping the Clock

The clock runs without interruption and we use a Register to function as a switch that when pushed, executes the program up to HLT. Upon receiving the *hlt* signal, the register resets to 0 and makes the effective clock signal to the whole pipeline 0.

6.3 Implementing the CU

The Implementation of the CU according to the truth tables discussed in the preceding section is shown below (the implementation uses a minimal number of logic gates):

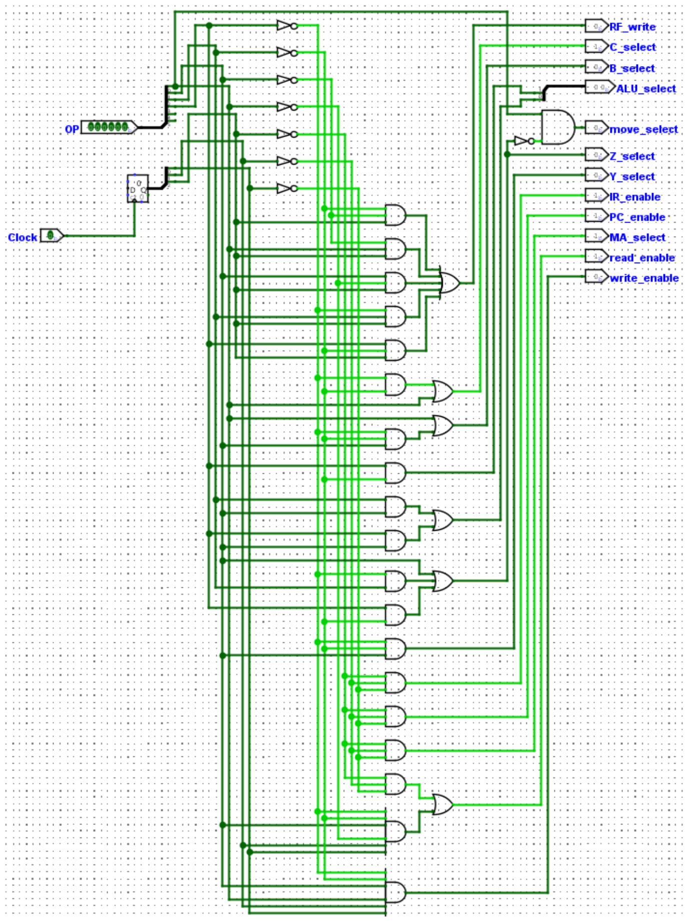


Figure 20: Control Unit

The Block Diagram for the Control Unit is given below:

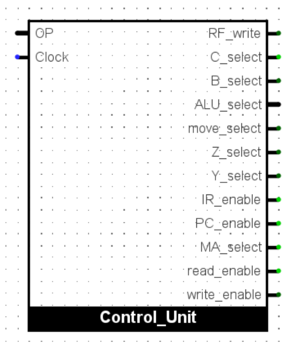


Figure 21: Control Unit Block

7 Assembling The Pipeline

We connect the Sub-Circuits that we have built is steps as shown:

7.1 Register File, ALU and Inter stage registers

Refer to the Circuit given below:

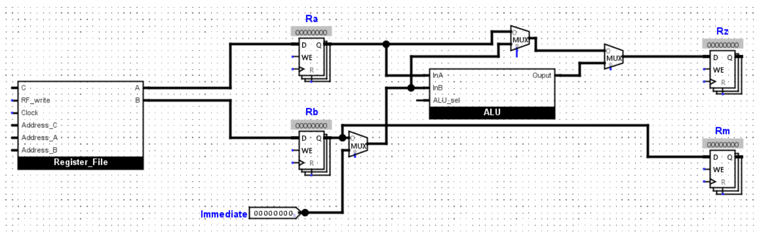


Figure 22: First Step

We have connected the outputs of the register file to the inter-stage registers R_A and R_B, and the output of R_B is fed to MUXB whose other input is the Immediate value. The outputs from R_A and MUXB are fed to the inputs of the ALU. Another MUX has the same 2 inputs as the ALU. This is used to select which value will bypass the ALU(either R_A in case of MOV or output of MUXB in case of MVI). Another multiplexer MUXZ chooses between the output of the bypassing MUX or the output of the ALU to forward to R_Z. In the same stage R_M is loaded with the value of R_B. All the registers are always write-enabled.

7.2 Final stage,PC,IR,Mem,CU

Refer to the Circuit given below:

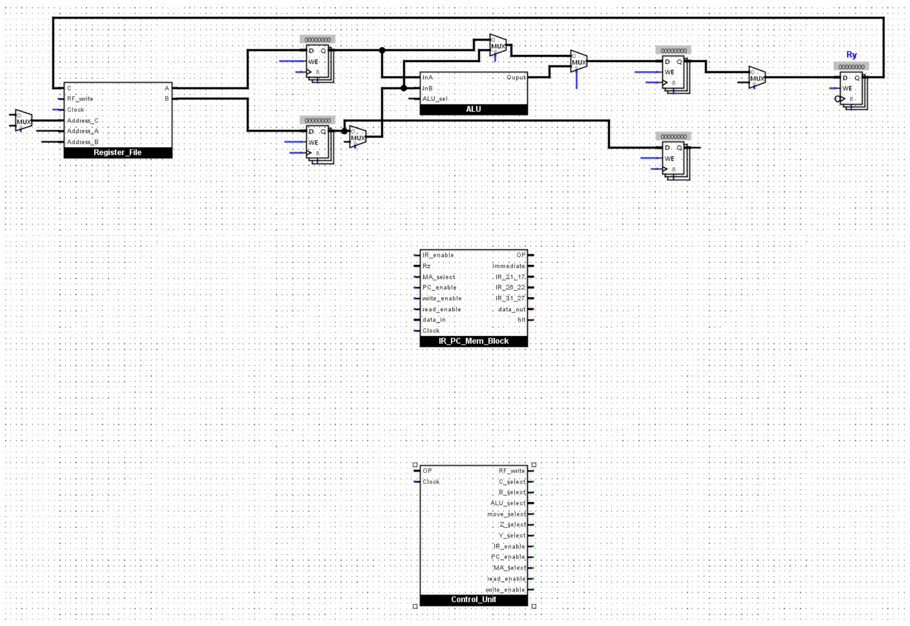


Figure 23: Second Step

We have added the final stage register R_Y through a multiplexer MUXY and looped its output to the input of the register file. We have also added a multiplexer MUXC which selects the address to write back to in the register file. We have also added the sub Circuits of the Control Unit and IR, PC, and Memory Circuit. In the next stage, we add the clock and do the wiring part.

7.3 Adding Clock, Wiring of PC Block

Refer to the Circuit given below:

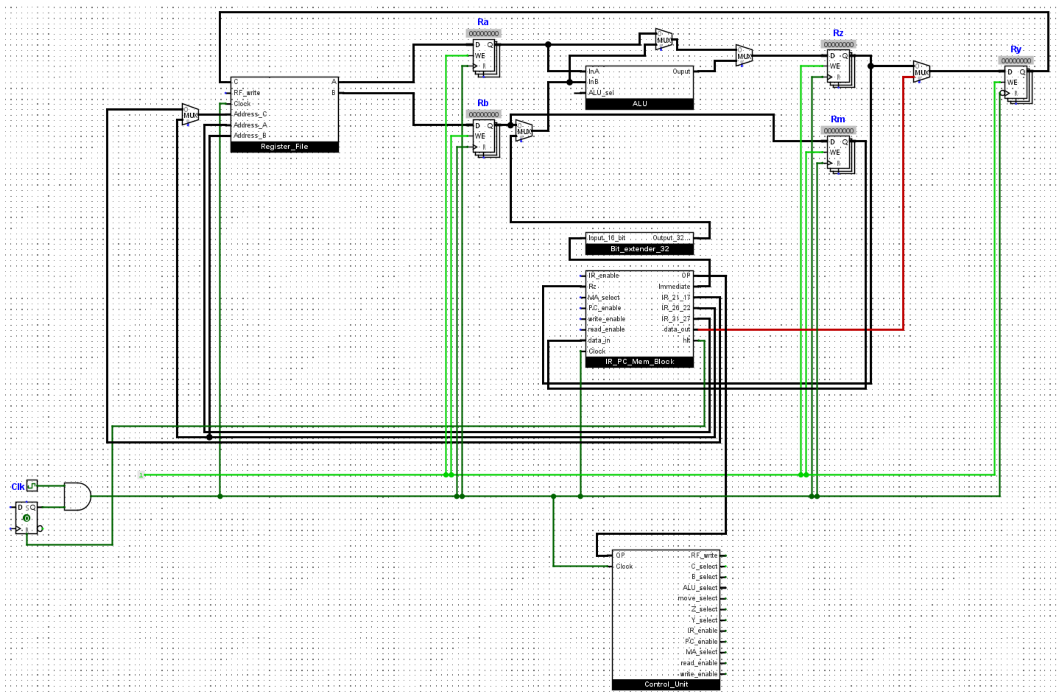


Figure 24: Third Step

Instead of adding a simple clock to our pipeline, we have added a modified clock as discussed in section 6.2.7. This will help us in implementing the halting of our execution. The output of this clock is given to all of the clock signals required in our circuit. A constant 1(high) signal is given to all of the write-enable bits of the inter-stage registers. Now, Let's discuss the wiring of the PC, IR, and Memory Block:
The value of R_Z is given to the PC block as input (this goes into the MUXMA multiplexer) and the value of R_M is given to the *data_in* input of the PC block (this value is written to the memory). The Outputs of the PC block are connected as follows:

- OP Code output of the PC Block is given to the input of Control Unit
- Immediate Value (IR_{21-6}) is sign-extended (by zero padding) and fed to the other input of MUXB.
- IR_{21-17} and IR_{26-22} outputs are connected to the input of MUXC which chooses the address of the RF to where we write back to.
- IR_{31-27} and IR_{26-22} outputs are connected to $Address_A$ and $Address_B$ inputs of the register file. These correspond to the address of Scr1 and Src2 given in the encoding scheme.
- *data_out* is connected to MUXY's other input.
- *hlt* signal is connected to the reset pin of the register controlling the clock.

7.4 Wiring Control Unit

We connect the outputs of the control unit as follows:

- *RF_write* is connected to the input pin of the same name of the register file.
- *C_select* is connected to the select bit of MUXC
- *B_select* is connected to the select bit of MUXB
- *ALU_select* is connected to the input pin of the same name of the ALU.
- *move_select* is connected to the select bit of the bypassing MUX.
- *Z_select* is connected to the select bit of MUXZ
- *Y_select* is connected to the select bit of MUXY
- *IR_enable* is connected to the input pin of the same name in the PC Block
- *PC_enable* is connected to the input pin of the same name in the PC Block
- *MA_select* is connected to the input pin of the same name in the PC Block
- *read_enable* is connected to the input pin of the same name in the PC Block
- *write_enable* is connected to the input pin of the same name in the PC Block

Given below is the final circuit after doing all the necessary connections:

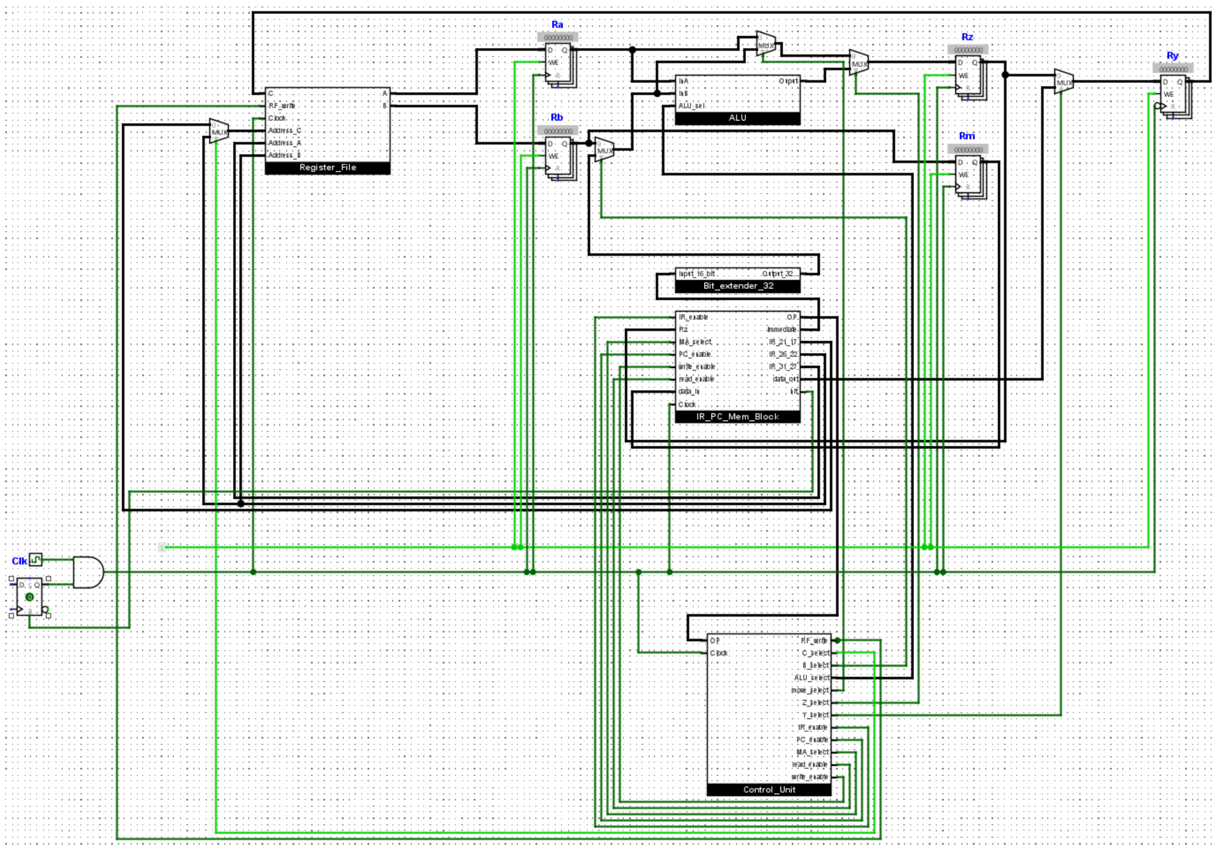


Figure 25: Final Circuit