Faculty of Engineering & Technology – Electrical & Computer Engineering Department

Second Semester 2022 – 2023

Computer Architecture - ENCS4370

Course Project – Multicycle RISC Processor

Prepared By:
Ahmaide Awawda – Mohammed Deek – Diyar Barham
1190823 – 1190556 – 1191314

Instructors:
Dr. Aziz Qaroush
Dr. Ayman Hroub

Date: June 2023

Introduction

The aim of this project is to design a multicycle RISC processor, that does different kinds of instructions, and testing its outcomes too using Verilog HDL language and the Quartos software.

In this project the instructions will be made in 32-bits, where the will be a program counter register that will give the address of each instruction, a register file that consists of 32 general purpose register each one of them is 32-bits, an ALU that does the processors needed operations, a 64GB memory, a control unit that generates the needed signals for each instruction type, a stack that saves the return addresses with a stack pointer register to point on top of the stack, and finally a clock organizer that decides the stage of each clock cycle as the multicycle approach spreads the cycles where each stage in the instruction takes one cycle.

This processor supports four different types of instructions (R-Type, I-Type, J-Type, and S-Type) with five stages that the instruction will have to go to some or all of them which are (Instruction Fetch, Instruction Decode, Execute, Memory Access, and Register Write) so can the system meet it's needed specifications.

Overall, the project comes in helping to understand and get a better knowledge of the architecture of the processor and the path that each different instruction can take to do it's needed operations.

❖ Table of Content

❖ Table of Figures

❖ Table of Tables

1. Design Requirements

1.1. Initial Specifications

The initial required specifications are listed below:

1. The instruction size is 32 bits.

2. 32 32-bit general-purpose registers: from R0 to R31.

3. A special purpose register for the program counter (PC).

4. It has a stack called control stack which saves the return addresses.

5. Stack pointer (SP), another special purpose register to point to the top of the control stack. SP holds the address of the empty element on the top of the stack. For simplicity, you can assume a separate on-chip memory for the stack, and the initial value of SP is zero.

6. Four instruction types (R-type, I-type, J-type, and S-type).

7. The processor's ALU has an output signal called "zero" signal, which is asserted when the result of the last ALU operation is zero.

8. Separate data and instructions memories

1.2. Instruction Types and Format

The given four instruction types are R-Type, I-Type, J-Type, and S-Type all the four instructions have the following fields:

a. 2-bit instruction type (00: R-Type, 01: J-Type, 10: I-type, 11: S-type).

b. 5-bit function, to determine the specific operation of the instruction.

c. Stop bit, which is the least significant bit of each instruction binary format, and it is used to mark the end of a function code block. In other words, if the value of this stop bit is "1", this means that this instruction is the last instruction of the function, and hence the execution control should return to the return address which is stored on the top of the control stack.

In addition, for each instruction type the following fields:

R-Type (Register Type) Format

• 5-bit Rs1: first source register

• 5-bit Rd: destination register

• 5-bit Rs2: second source register

• 9-bit unused

Function^5	Rs1^5	Rd^5	Rs2^5	Unused^9	Type^2	Stop^1

I-Type (Immediate Type) Format

• 5-bit Rs1: first source register

• 5-bit Rd: destination register

• 14-bit immediate: unsigned for logic instructions and signed otherwise.

Function^5	Rs1^5	Rd^5	Immediate^{14}	Type^2	Stop^1

J-Type (Jump Type) Format

• 24-bit signed immediate: jump offset

Function^5	Signed Immediate^{24}	Type^2	Stop^1

S-Type (Shift Type) Format

• 5-bit Rs1: first source register.

• 5-bit Rd: destination register.

• 5-bit Rs2: second source register. This register stores the shift amount in case the shift amount is variable, and it is calculated at runtime.

• 5-bit SA: the constant shift amount.

• 4-bit unused.

Function^5	Rs1^5	Rd^5	Rs2^5	SA^5	Unused^4	Type^2	Stop^1

1.3. Instructions Encoding

The instructions that the processor in this project can perform are all shown in table 1 below, with the function for each instruction in its instruction type with it's meaning too. (Note that the branch would do the same instruction of the jump if its requirement met) the table also includes the number of cycles per instruction.

Table 1: Instructions Encoding

No.	Instr	Meaning	Num of Cycles	Function Value
R-Type Instructions				
1	AND	Reg(Rd) = Reg(Rs1) & Reg(Rs2)	4	00000
2	ADD	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	4	00001
3	SUB	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	4	00010
4	CMP	zero-signal = Reg(Rs) < Reg(Rs2)	3	00011
I-Type Instructions				
5	ANDI	Reg(Rd) = Reg(Rs1) & Immediate^{14}	4	00000
6	ADDI	Reg(Rd) = Reg(Rs1) + Immediate^{14}	4	00001
7	LW	Reg(Rd) = Mem(Reg(Rs1) + Imm^{14})	5	00010
8	SW	Mem(Reg(Rs1) + Imm^{14}) = Reg(Rd)	4	00011
9	BEQ	Branch if (Reg(Rs1) == Reg(Rd))	3	00100
J-Type Instructions				
10	J	PC = PC + Immediate^{24}	2	00000
11	JAL	PC = PC + Immediate^{24} Stack.Push (PC + 4)	3	00001
S-Type Instructions				
12	SLL	Reg(Rd) = Reg(Rs1) << SA^{5}	4	00000
13	SLR	Reg(Rd) = Reg(Rs1) >> SA^{5}	4	00001
14	SLLV	Reg(Rd) = Reg(Rs1) << Reg(Rs2)	4	00010
15	SLRV	Reg(Rd) = Reg(Rs1) >> Reg(Rs2)	4	00011

1.4. RTL Design

Since this project will have the multicycle approach, so each instruction stage will take a cycle, as each instruction will take 2-5 cycles depending on its requirements and features as the system will be built from five stages (instruction fetch, instruction decode, execution, memory, and register write), the number of cycles is shown in table 1 above.

2. System Design and Components

2.1. The Full Data Path

The data path of the ISA consists of different elements, each element is built behaviorally, and it does its job for stage that it is settled in, as it can be shown in the figure below.

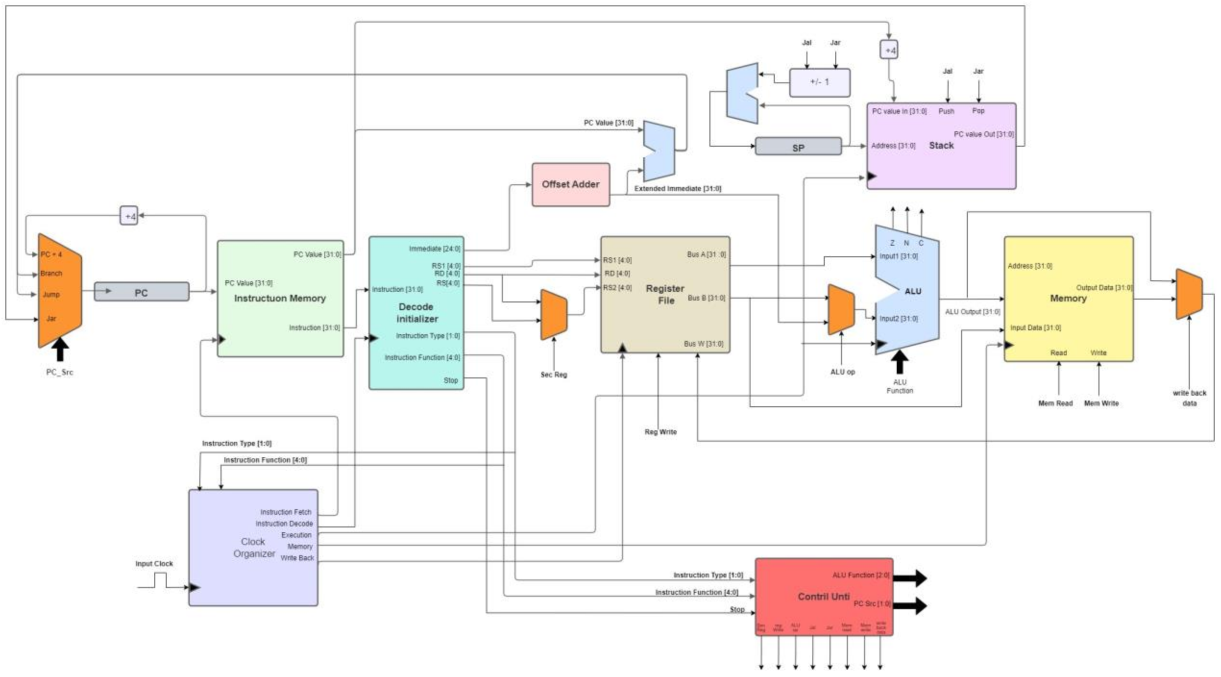

Figure 1: The Full Data Path

As shown in figure 1, each stage takes an input clock in order to spread the clock cycle on every stage depending on the state of the ISA which looks at the previous clock and the instruction type and the instruction function.

The system consists of 4 Muxes, yet only one of them will have its own module in the code which is the last Mux as it won't have its own stage or module to relate to, while one mux will be with the instruction memory module, and the second one will be with the register file module, and the third one will be with the ALU module.

2.2. The clock organizer

The clock organizer is the moule that is responsible for making this processor a multicycle processor (finite state machine), as it gives each clock cycle to the right stage for it so that each instruction takes as many stages time as it needs unlike the single cycle approach, the clock organizer takes the instruction type and the function too to decide the next stage.

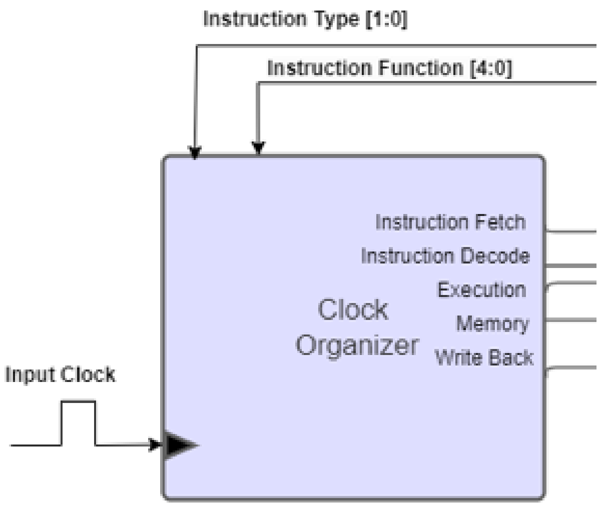

Figure 2: Clock Organizer Diagram

The clock organizer will have a state variable that indicates what stage the system is in, as the state diagram is shown in the figure below, to tell the system's next stage.

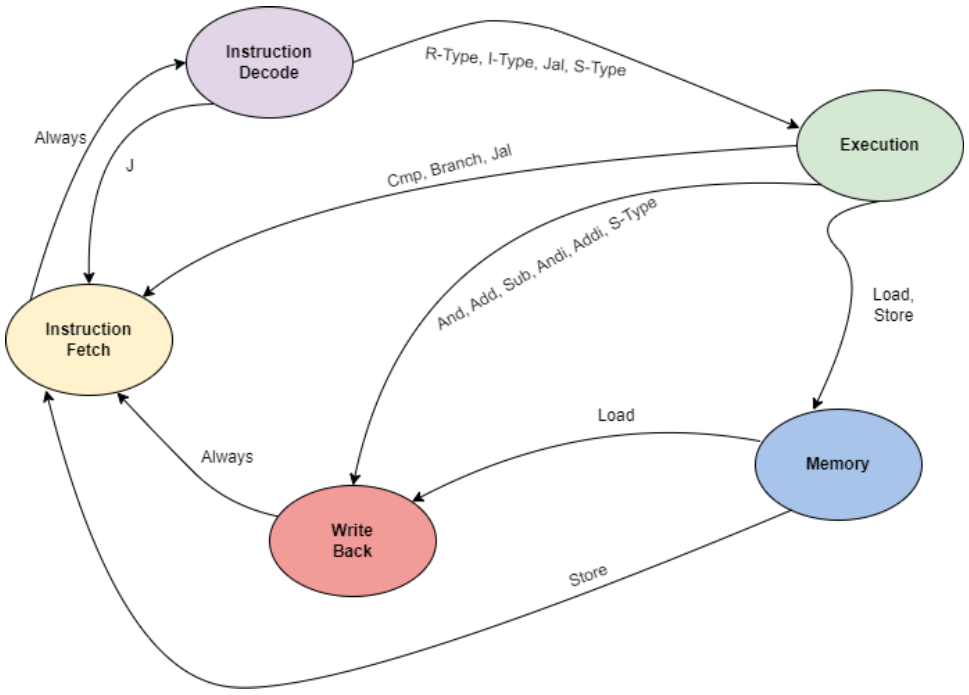

Figure 3: Stages State Diagram

The code for the clock organizer, can be found in **Appendix 1**, and the table below shows the state for each state and the possible next stages for the code.

Table 2: Stages States

Current Stage	State Code	Next Stage	Next Stage State Code	Instructions Cases
Instruction Fetch	00000	Instruction Decode	00001	Always
Instruction Decode	00001	Execute	00010	R-Type, I-Type, Jal, S-Type
		Instruction Fetch	00001	J
Execute	00010	Memory	00011	Load, Store
		Register Write	00100	And, Add, Sub, Andi, Addi, S-Type
		Instruction Fetch	00000	CMP, BEQ, Jal
Memory	00011	Register Write	00100	Load
		Instruction Fetch	00000	Store
Register Write	00100	Instruction Fetch	00000	Always

The clock organization testing waveform is shown in the figure below, (for this case the first instruction type is R-Type, and the function is 00010 which is Sub followed by a store operation, so the following stages should go in order IF, ID, EX, WB, IF, ID, EX, MEM).

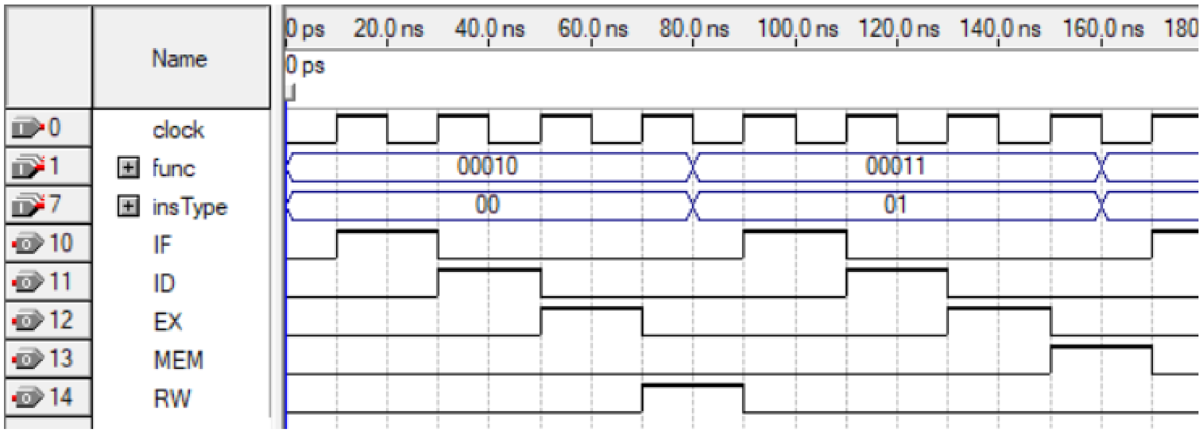

Figure 4: Clock Organizer Testing

2.3. Instruction Fetch Stage Components

The first stage of every instruction first determines the next PC address where it can be the previous PC + 4, or the jump address, or the return address (Jar), then it takes that PC address and returns the instruction at that address, as it is a memory of instructions with the PC pointing at the address of the current instruction, as shown in figure 5 below.

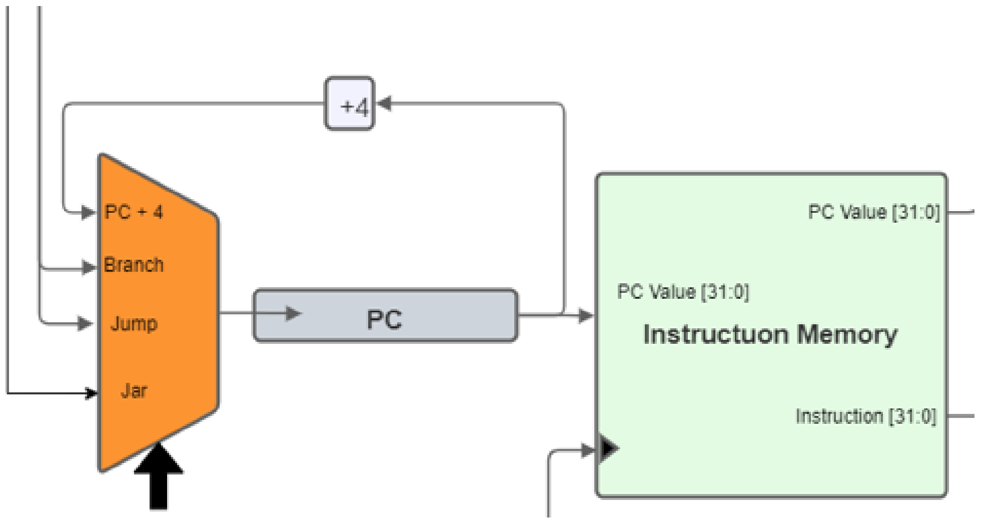

Figure 5: Instruction Fetch Diagram

The PC address consists of 32-bits which means that this machine can support 4Gb instructions, The next PC address is decided by a 2-bit signal called Pc Src which gives in the 32-bit next PC address. The instruction memory gives an instruction that also consists of 32-bits.

All those components (4x1 Mux, PC, and the instruction memory are all in one module which is called instructionFetch and its code can be found in **Appendix 2**, and the testing waveform for the module is as shown below in figure 6 as all cases were taken and the outputs were displayed in hexadecimal.

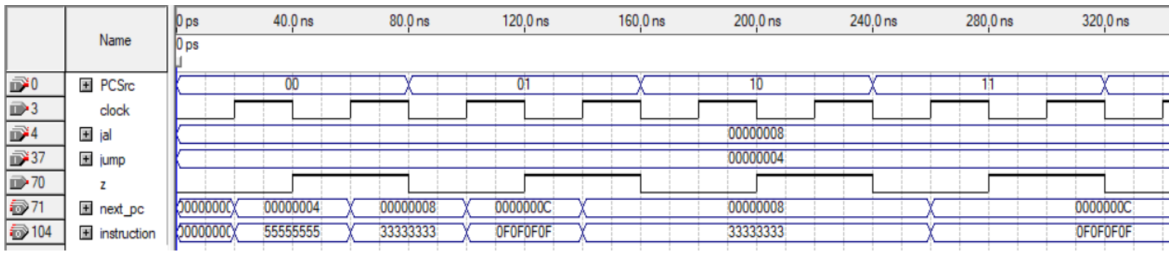

Figure 6: Instruction Fetch Testing

As shown in figure 6, pc takes its values depending on the PCSrc, and then decides what is the next PC, the difference between the jump and branch cases is that the jump will always take the jump address, yet the branch will only go if the zero flag was on as the two inputs of the ALU were the same.

2.3. Instruction Decode Stage Components

In this Stage the Instruction will be taken down to its elements (Type, Function, stop bit, registers, and immediate) in this project two modules were made, the diagram of the stage is shown below in figure 7.

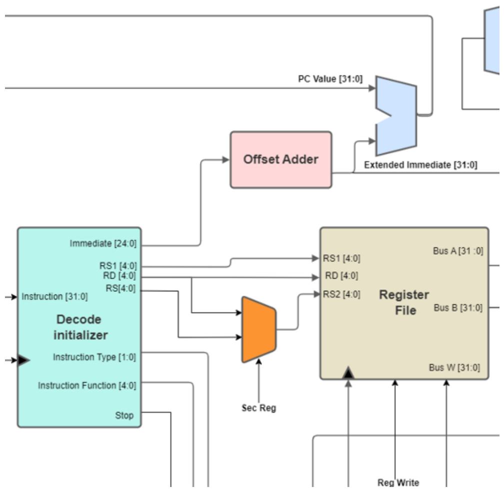

Figure 7: Instruction Decode Diagram

The first one called decode in code which takes the instruction and gives out it’s elements and adds the offset to the immediate (which can be 14-bits in I-Type or 24-bits in J-type, or 5-bits in S-Type) in order to extend it to 32-bits (note that in the S-Type the shift amount will also be used as a 5-bit immediate), the module also calculates the jump and branch address with the PC value and the immediate, this module will do its work on its clock’s positive edge, and its code can be found in **<u>Appendix 3</u>**.

The second module is the register file which contains all the 32 registers values (all initially zeros). It takes three 5-bit registers addresses checks which one is the second source register (the mux is part of the module) and then output’s the data on both source registers on bus A and bus B both are 32-bit, note that the register file has an input clock and another input called bus W which will both work in the register write stage, and will be detailed in that stage’s section, the code can be found in **<u>Appendix 4</u>**.

The testing for the Decode module is shown below in figure 8, yet the testing for the register file is in the Write back stage section, so that all the register file data can be shown read and written.

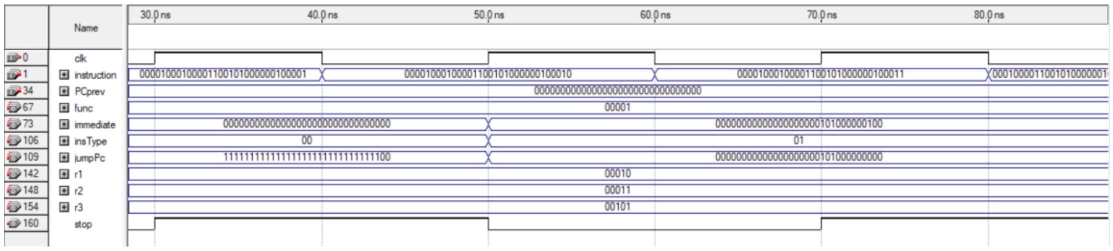
Name
30.0 ns
40.0 ns
50.0 ns
60.0 ns
70.0 ns
80.0 ns
0 clk
1 instruction 00001000100001100101000000100001 00001000100001100101000000100010 00001000100001100101000000100011 0001000011001010000001
34 PCprev 00000000000000000000000000000000
67 func 00001
73 immediate 00000000000000000000000000000000 00000000000000000000101000000100
106 insType 00 01
109 jumpPc 11111111111111111111111111111100 00000000000000000000101000000000
142 r1 00010
148 r2 00011
154 r3 00101
160 stop

Figure 8: Instruction Decode Testing

2.4. Execution Stage Components

The third stage of this ISA consists of two modules the ALU and the stack which both start working when the EX-clock triggers, as shown in figure 9 below.

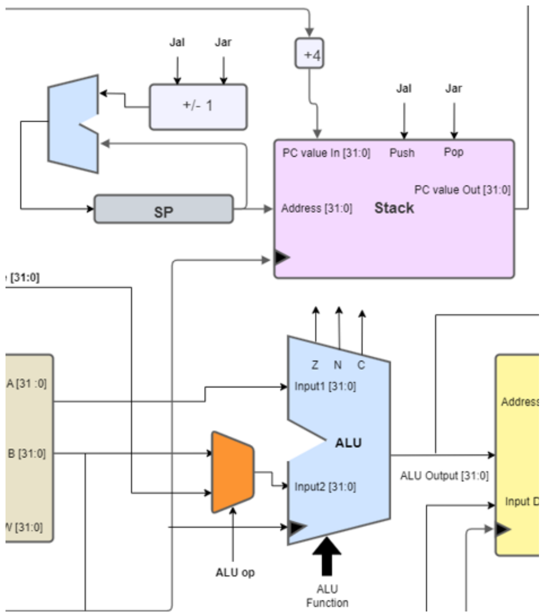

Figure 9: Execution Stage Diagram

The first module is the ALU, the code for the ALU module can be found in **Appendix 5**. The first input is the ALU op signal which indicates whether the second input (the first is always BusA) is the BusB or the extended immediate. If the value of the ALU op is equal to 1 then the second input will be the extended immediate. Otherwise, the second input will be the BusB. The BusA,BusB and the extended immediate are a 32-bit registers that the different arithmetic operations will be performed using them. In order to determine which arithmetic operation to be performed the FUNTION input will be used. The FUNCTION field is a 3-bit register and it indicates which arithmetic operation to be performed as follows:

- When the binary Value of it is 000 and ALU op value is 0 then it will perform bitwise and operation between BusA and BusB
- When the binary Value of it is 000 and ALU op value is 1 then it will perform bitwise and operation between BusA and the extended immediate
- When the binary Value of it is 001 and ALU op value is 0 then it will perform addition operation between BusA and BusB
- When the binary Value of it is 001 and ALU op value is 1 then it will perform addition operation between BusA and the extended immediate
- When the binary Value of it is 010 then it will perform subtraction operation between BusA and BusB

- When the binary Value of it is 011 then it will perform bitwise shift-left on BusA by the value of extended immediate
- When the binary Value of it is 100 then it will perform bitwise shift-right on BusA by the value of extended immediate
- When the binary Value of it is 101 then it will perform bitwise shift-left on BusA by the value of BusB
- When the binary Value of it is 110 then it will perform bitwise shift-right on BusA by the value of BusB

Lastly, the first output is the result of the arithmetic operation, after that there is three output flags which are:

- Zero_flag which indicates whether the inputs are equal or not
- Negative flag which indicates if BusA value is less than the second input
- Carry flag which indicates if there is a carry bit in the result of the ALU.

The second module is the stack which contains the stack memory element along side the stack pointer register, the stack is word addressable so each index points at a 32-bit element, this module stores the PC + 4 of the Jal instruction, and returns it as the current next instruction when the stop-bit = 1, the SP register point at the address of the address of the last entered return address in the stack, as the value of the SP increments in the Jal instructions and decrements when the stop-bit is equal to one, also the stack pushes the PC + 4 address in cases of Jal, and pops it in the stop-bit cases, the code for the stack module can be found in **Appendix 6**.

The testcase waveforms for the two modules are shown in the two figures below, figure 10 for the ALU and figure 11 for the stack module.

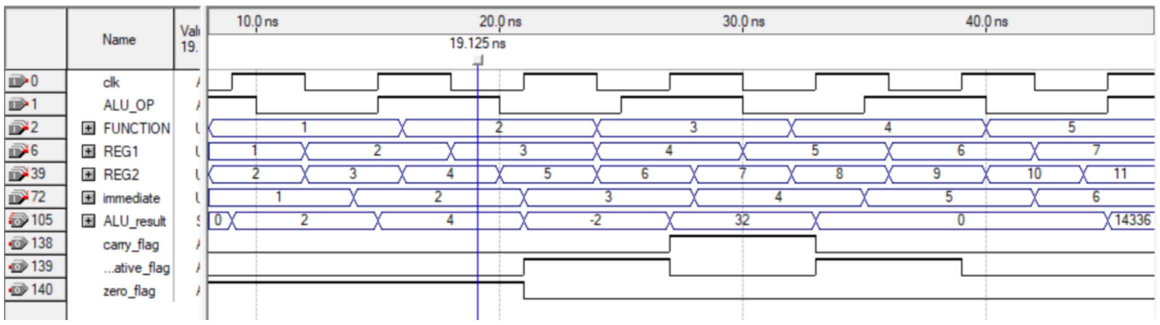

Figure 10: ALU Testing

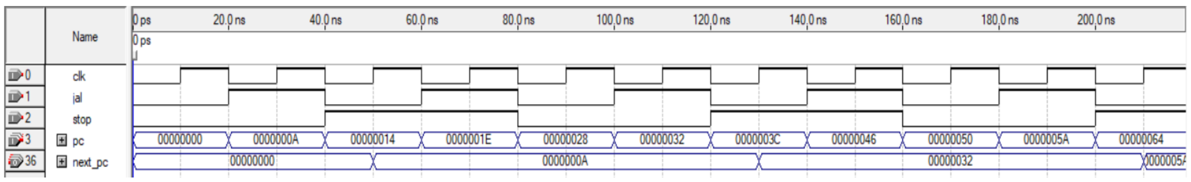

Figure 11: Stack Testing

2.5. Memory Stage Components

Initially, the value of the memory is zero because nothing is saved inside of it. Then memory has 4 inputs which are

- 32-bit Address that gives the memory address for the value the operation will be performed on
- 32-bit input data if the operation is writing on the memory
- Mem Read which reads the value of the memory in the specified address and put it in the output.
- Mem Write which overwrite the value inside the memory in the specified address by the input data value.

The memory has only one output that works only if Mem Read = 1, which gives out the data that is stored in the address (ALU output).

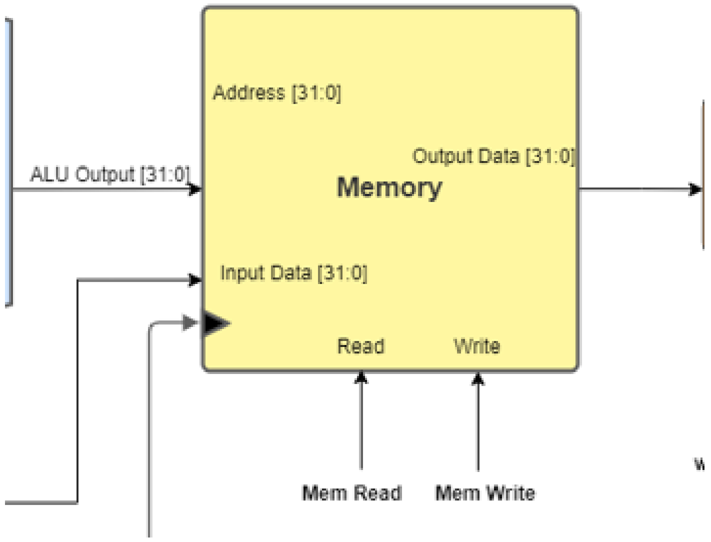

Figure 12: Memory Stage Diagram

The code for the memory module can be found in **Appendix 7**.

The testing for the memory element is in figure 13 below, as a value will be stored on an address, and then the same value will be loaded from the same address.

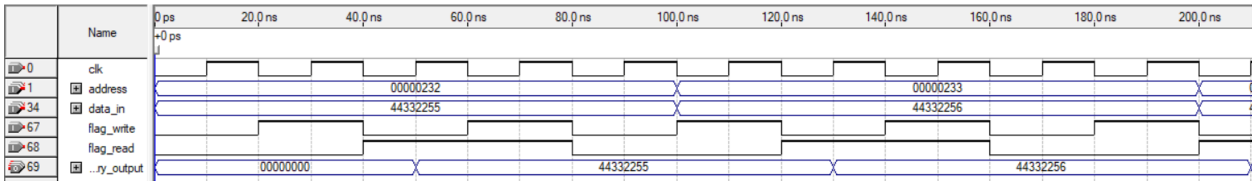

Figure 13: Memoery Testing

2.6. Register Write Stage Components

If this stage comes in an instruction, it will for sure be the final stage of the instruction as the next stage will always be instruction fitch for the next instruction, in this stage either the memory or the ALU output will be written on the destination register that was set in the decode stage, as shown in figure 14.

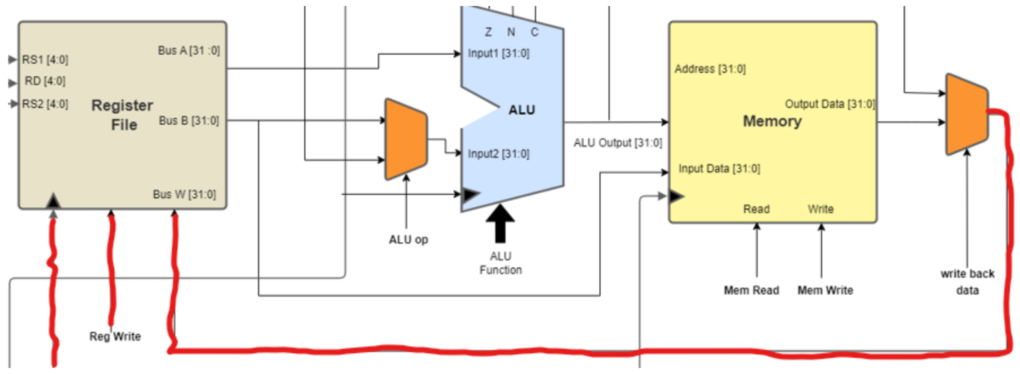

Figure 14: Write Back Stage Diagram

In the decode stage the destination register was set, yet nothing will be written on it not unless the WB clock triggers which will happen in this stage as the data to be written on the destination register is ready on the Bus W input and the reg Write flag is set in order to allow the system to write the data on the register.

The Mux decides if the write back data comes from the ALU result or the data stored on the memory address, as it takes 32-bit inputs and outputs 32-bit output, and the 2x1 mux code can be found in **Appendix 8**.

Figure 15 below shows the waveform that tests the register file as the data will be written on it, then it will go as output on the buses on the next instruction, for it to be validated, and the testing for the 2x1 mux is displayed below it in figure 16.

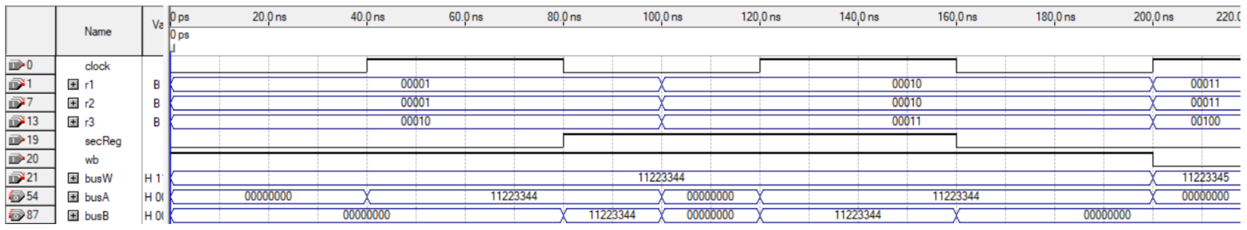

Figure 15: Register File Testing

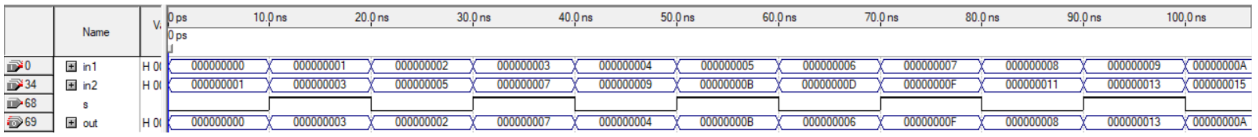

Figure 16: 2x1 Mux Testing

2.7. Control Unit

The Control unit is the element that set that flags for the signals that control and choose the needed operations or elements to work in each different instruction, so it's basically the decider for the instruction, as it takes the instruction type and the instruction function in the decode stage and set up the flags for the register file, immediate extender and the elements of the upcoming stages, alongside the instruction fetch for the next stage, and the diagram of the control unit is displayed below in figure 17.

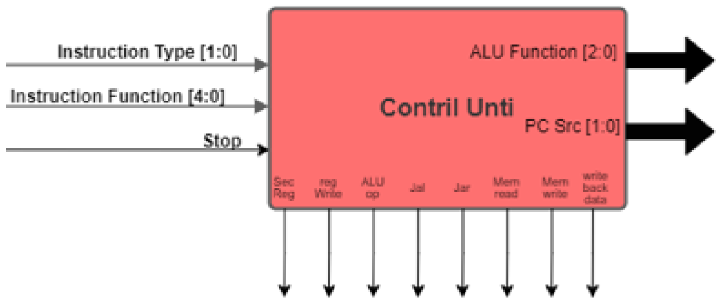

Figure 17: Control Unit Diagram

As shown in figure 17, the control unit sets up the flags which are (the second source register, the flag that allows to write on the destination register, the ALU second operant, the Jal instruction flag, the Jar to return the address flag, the memory read flag, the memory write flag, and the write back data flag too) the code for the control unit unit's code can be found in **Appendix 9**.

Figure 18 below shows the testing waveform for the system's control unit.

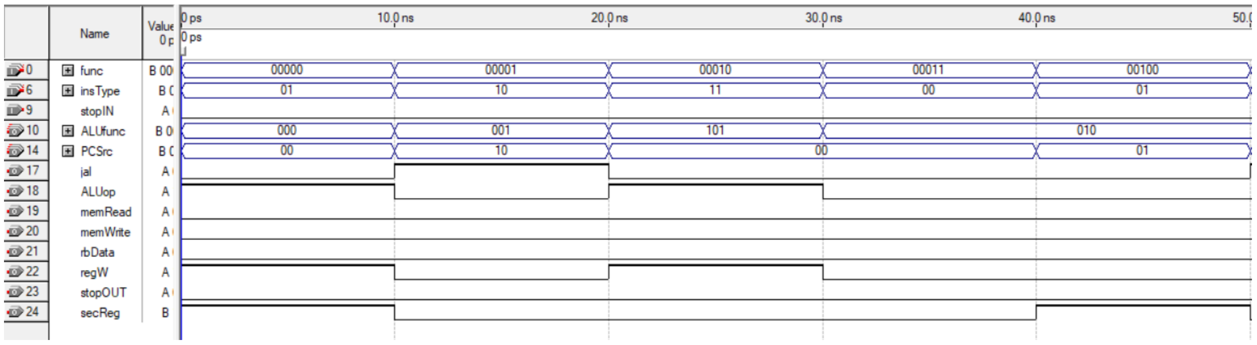

Figure 18: Control Unit Testing

The calculations and control equations with the Boolean expressions can be found in the system signals chapter.

3. The System’s Control Signals

3.1. Signals Meanings

As said before the system contains ten control signals, each one has its value depending on the instruction type, instruction function, and the stop bit too the following are the control signals with their meanings:

1. PC Src: This is a two-bit signal that determines if weather the next PC source should be (in order), PC +4 or the jump address in case of the equality of the registers, or the jump address without the equality of register, or the return address as the stop bit was equal to 1.

2. Sec Reg: this flag determines if the second or third register address from the instruction is the second source register in the register file.

3. Jal: triggers only on Jal operations in order to save the PC + 4 address on the stack.

4. Jar: triggers when the stop bit is equal to one, as to pop the return PC from the stack and put it as the next instruction.

5. ALU Op: This flag decides whether the second input to the ALU is the second source register data or the extended immediate.

6. ALU Func: This is a 3-bit flag that decides the ALU function which can be in order (AND, ADD, SUB, Shift-left, Shift-right)

7. Mem Read: This flag is triggered when there is a need to read data from the memory.

8. Mem Write: This flag is triggered to allow data to be written on the memory.

9. WB Data: This flag decides if the data that will be written on the destination register will be the ALU output or the memory output.

10. Reg Write: This flag triggers to allow the bus W data to be written on the destination register.

3.2. Signals Values by Instructions

The table below shows all the control signals values in each instruction, where some of them will have the value (x) as the value doesn't really matter.

Table 3: Flags Values for Each Instruction

Instruction	PC Src	Sec Reg	Jal	ALU Op	ALU Func	Mem Read	Mem Write	WB Data	Reg Write	Jar
R-Type										
AND	00	0	0	0	000	0	0	0	1	0
ADD	00	0	0	0	001	0	0	0	1	0
SUB	00	0	0	0	010	0	0	0	1	0
CMP	00	0	0	0	010	0	0	X	0	0
I-Type										
ANDI	00	X	0	1	000	0	0	0	1	0
ADDI	00	X	0	1	001	0	0	0	1	0
LW	00	X	0	1	001	1	0	1	1	0
SW	00	1	0	1	001	0	1	X	0	0
BEQ	01	1	0	0	010	0	0	X	0	0
J-Type										
J	10	X	0	X	X	0	0	X	0	0
JAL	10	X	1	X	X	0	0	X	0	0
S-Type										
SLL	00	0	0	1	011	0	0	0	1	0
SLR	00	0	0	1	100	0	0	0	1	0
SLLV	00	0	0	0	011	0	0	0	1	0
SLRV	00	0	0	0	100	0	0	0	1	0
Stop Bit										
Stop = 1	11	Depends on The Instruction								1

3.3. Signals Boolean Expressions

Each signal has its Boolean value depending on as said before (instruction type, function, and stop bit), each signal has its Boolean expression as shown below:

PC Src[0] = (BEQ && Z) || Stop-bit

PC Src[1] = J-Type || Stop-bit

Sec Reg = SW || BEQ

Jal = JAL

ALU Op = (I-Type && ~BEQ) || SLL || SLR

ALU Fun[0] = ADD || ADDI || LW || SW || SLL || SLRV

ALU Fun[1] = SUB || CMP || BEQ || SLL || SLRV

ALU Fun[2] = SLR || SLRV

Mem Read = LW

Mem Write = SW

WB Data = LW

Reg Wr = (R-Type && ~ CMP) || (I-Type && ~SW && ~BEQ) || S-Type

Jar = Stop-bit

4. Final Design Simulation and Testing

4.1. Combining The Components

The final design is as shown in figure 19 below after adding up and connecting all the components together, in the conde all the connections were initialized as wires, except for the ones that will be used in their stages as input before as output, in this case they needed to have initial values so there initialized as registers in order to have zero values, the code for the Processor that adds up all the components together can be found in **Appendix 10**.

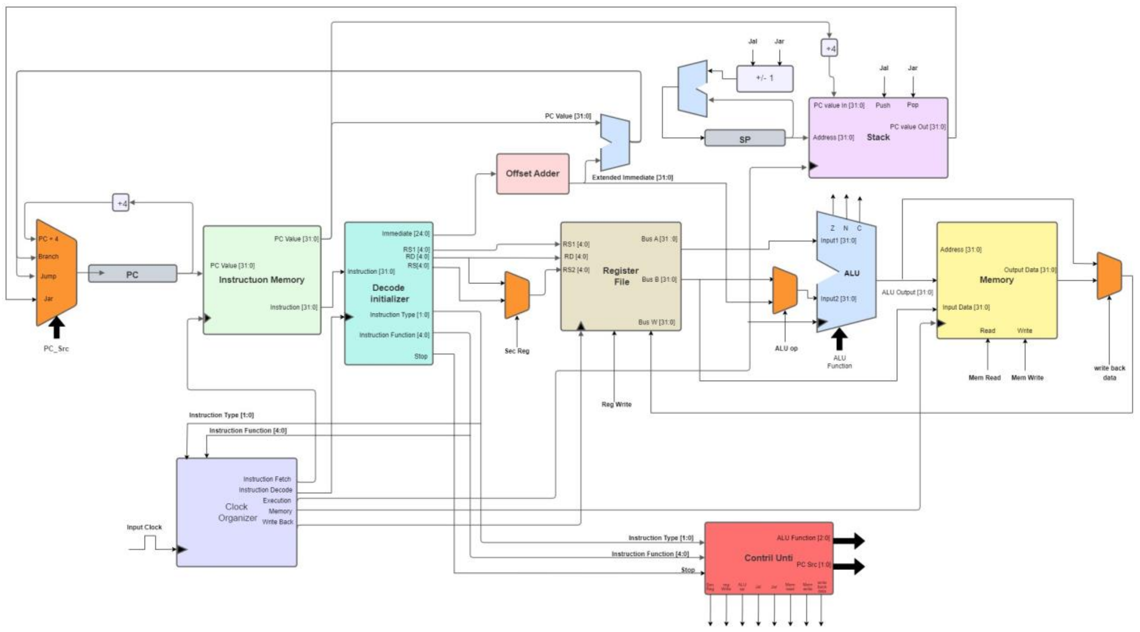

Figure 19: Final Data Path

4.2. Writing a Test Bench for The System

To test the design a test bench needs to be written and then check the validation and correctness of its data, so the following test bench was written and given in the instruction fetch, in order to be tested.

```
Addi R1, R1, 5

Andi R3, R1, 9

Sub R6, R1, R3

Sw R6, $(R1), 3

Lw R10, $(R3), 7
```

```
SLL R11, R10, 2

SLRV R11, R11, R3

Jal 3

And R7, R2, R11

BEQ R6, R10, 8

Add R2, R1, R6

Jar
```

As it was written in code in binary format:

```
memory[0] <= 32'b00001000010000100000000000101010;

memory[1] <= 32'b00000000010001100000000001001010;

memory[2] <= 32'b00010000010011000011000000000000;

memory[3] <= 32'b00011000010011000000000000011010;

memory[4] <= 32'b00010000110101000000000000111010;

memory[5] <= 32'b00000010100101100000000100000110;

memory[6] <= 32'b00011010110101100011000000000110;

memory[7] <= 32'b00001000000000000000000001100100;

memory[8] <= 32'b00000000010001110101100000000000;

memory[9] <= 32'b00100001100101000000000001000010;

memory[10]<= 32'b00001000010001000110000000000001;

memory[11] <= 32'b00000000000000000000000000000000;
```

4.3. Testing The System

The testbench waveform screenshots can be found in the following link:

https://drive.google.com/drive/folders/1ei3_YfGWaeYLf2_r6SuwRuow7qnjyNin?usp=sharing

5. Conclusion and Future Work

In conclusion, an efficient MIPS CPU was constructed using the multi-cycle approach. Throughout the project, a more in-depth understanding about how exactly the MIPS CPU works when using the multi-cycle approach was obtained while designing the different components. The multicycle approach was used due to its efficiency benefits compared to the single-cycle approach because it divides each stage, and the stages works separately unlike the single-cycle approach. In addition to that, the best way to connect the modules using code and testing them was learnt. Finally, in the future work the final module can be optimized using the pipeline approach in which it reduces the cycles per instruction (CPI) in the CPU.

6. Appendix

Appendix 1

```
module clkOrganizer(
    input clock,
    input wire [4:0] func,
    input wire [1:0] insType,
    output reg IF,
    output reg ID,
    output reg EX,
    output reg MEM,
    output reg RW
);

reg [2:0] state;

always @(posedge clock) begin

    case(state)

            3'b000: begin
                    state<=001;
                    IF<=0;
                    ID<=1;
                    EX<=0;
                    MEM<=0;
                    RW<=0;
            end

            3'b001: begin
                    if(insType == 2'b10 && func == 5'b00000)begin
                            state<=000;
                            IF<=1;
                            ID<=0;
                            EX<=0;
                            MEM<=0;
                            RW<=0;
                    end
                    else begin
                            state<=010;
                            IF<=0;
```

```
                        ID<=0;
                        EX<=1;
                        MEM<=0;
                        RW<=0;
                    end
                end

                3'b010: begin
                    if(insType == 2'b10 || (insType == 2'b01 && func == 5'b00100) ||
        (insType == 2'b00 && func == 5'b00011)) begin
                        state<=000;
                        IF<=1;
                        ID<=0;
                        EX<=0;
                        MEM<=0;
                        RW<=0;
                    end
                    else if(insType== 2'b01 && (func == 5'b00010 || func == 5'b00011))begin
                        state<=011;
                        IF<=0;
                        ID<=0;
                        EX<=0;
                        MEM<=1;
                        RW<=0;
                    end
                    else begin
                        state<=100;
                        IF<=0;
                        ID<=0;
                        EX<=0;
                        MEM<=0;
                        RW<=1;
                    end
                end

                3'b011: begin
                    if(insType==2'b01 && func==5'b00010) begin
                        state<=100;
                        IF<=0;
                        ID<=0;
                        EX<=0;
                        MEM<=0;
                        RW<=1;
                        end
                        else begin
                                state<=000;
                                IF<=1;
                                ID<=0;
                                EX<=0;
                                MEM<=0;
                                RW<=0;
                        end
                end

                default: begin
                        state<=000;
                        IF<=1;
                        ID<=0;
                        EX<=0;
                        MEM<=0;
                        RW<=0;
                end


        endcase

    end

    initial begin

        state <= 3'b111;

    end
    endmodule
```

Appendix 2

```
module instructionFetch (
  input wire [31:0] jump,
  input wire [31:0] jar,
  input wire [1:0] PCSrc,
  input wire z,
  input wire clock,
  output reg [31:0] next_pc,
  output reg [31:0] instruction
);
```

```
reg [31:0] pc;
reg [31:0] memory [0:20];

always @(posedge clock) begin
  case (PCSrc)
    2'b00: begin
              next_pc <= pc + 32'b100;
              pc <= pc + 32'b100;
              instruction <= memory[pc/4];
       end

    2'b01: begin
      if (z == 1'b0) begin
                next_pc <= pc + 32'b100;
        pc <= pc + 32'b100;
        instruction <= memory[pc/4];
      end
      else begin
        next_pc <= jump + 32'b100;
        pc <= jump + 32'b100;
        instruction <= memory[jump/4];
      end
    end

    2'b10: begin
         next_pc <= jump + 32'b100;
         pc <= jump + 32'b100;
         instruction <= memory[jump/4];
       end

    2'b11:begin
      next_pc <= jar + 32'b100;
      pc <= jar + 32'b100;
      instruction <= memory[jar/4];
    end
  endcase


end

initial begin
```

```
  memory[0] <= 32'b00001000010000100000000000101010;
  memory[1] <= 32'b00000000010001100000000001001010;
  memory[2] <= 32'b00010000010011000011000000000000;
  memory[3] <= 32'b00011000010011000000000000011010;
  memory[4] <= 32'b00010000110101000000000000111010;
  memory[5] <= 32'b00000010100101100000000100000110;
  memory[6] <= 32'b00011010110101100011000000000110;
  memory[7] <= 32'b00001000000000000000000001100100;
  memory[8] <= 32'b00000000010001110101100000000000;
  memory[9] <= 32'b00100001100101000000000001000010;
  memory[10]<= 32'b00001000010001000110000000000001;
  memory[11] <= 32'b00000000000000000000000000000000;

  end


  initial begin
    pc <= 32'b0;
  end
endmodule
```

Appendix 3

```
module Decode (
        input clk,
        input [31:0] instruction,
        input signed [31:0] PCprev,
        output reg [4:0] r1,
        output reg [4:0] r2,
        output reg [4:0] r3,
        output reg [1:0] insType,
        output reg [4:0] func,
        output wire [31:0] immediate,
        output reg stop,
        output wire [31:0] jumpPc
);

reg [31:0] offset0;
reg [31:0] offset1;

reg [1:0] insTypeFl;
reg [4:0] funFl;
reg [31:0] immediateFl;
```

```
always @(posedge clk)
begin
    func <= instruction[31:27];
    funFl <= instruction[31:27];

    r1 <= instruction[26:22];
    r2 <= instruction[21:17];
    r3 <= instruction[16:12];

    insType <= instruction[2:1];
    insTypeFl <= instruction[2:1];

    stop <= instruction[0];

    if (instruction[2:1] == 2'b01 && instruction[31:27] == 5'b0)
    begin
            immediateFl <= {offset0[31:14], instruction[16:3]};
    end

    else if(instruction[2:1] == 2'b01 && instruction[31:27] >= 5'b00001)
    begin
            if(instruction[16]==0) begin
                    immediateFl <= {offset0[31:14], instruction[16:3]};
            end
            else begin
                    immediateFl <= {offset1[31:14], instruction[16:3]};
            end
    end

    else if (instruction[2:1] == 2'b10)
    begin
            if(instruction[26]==1) begin
                    immediateFl <= {offset1[31:24], instruction[26:3]};
            end
            else begin
                    immediateFl <= {offset0[31:24], instruction[26:3]};
            end

    end
    else begin
            immediateFl <= {offset0[31:5], instruction[11:7]};
    end
```

```
end

        assign jumpPc = immediateFl + PCprev - 4;
        assign immediate = immediateFl;

initial begin

        offset0 <= 32'b0;
        offset1 <= 32'b11111111111111111111111111111111;

end
endmodule
```

Appendix 4

```
module registerFile (
  input wire [4:0] r1,
  input wire [4:0] r2,
  input wire [4:0] r3,
  input wire secReg,
  input wire wb,
  input wire clock,
  input [31:0] busW,
  output reg [31:0] busA,
  output reg [31:0] busB
);

  reg [31:0] registers [0:31];

  always @(posedge clock) begin
    if (wb)
      registers[r2] <= busW;
  end



  always @(*) begin
        busA <= registers[r1];
    if (secReg == 0)
      busB <= registers[r3];
    else
      busB <= registers[r2];
  end
  initial begin
    registers[0]  = 32'h0; registers[1]  = 32'h0; registers[2]  = 32'h0; registers[3]  = 32'h0;
    registers[4]  = 32'h0; registers[5]  = 32'h0; registers[6]  = 32'h0; registers[7]  = 32'h0;
    registers[8]  = 32'h0; registers[9]  = 32'h0; registers[10] = 32'h0; registers[11] = 32'h0;
    registers[12] = 32'h0; registers[13] = 32'h0; registers[14] = 32'h0; registers[15] = 32'h0;
    registers[16] = 32'h0; registers[17] = 32'h0; registers[18] = 32'h0; registers[19] = 32'h0;
    registers[20] = 32'h0; registers[21] = 32'h0; registers[22] = 32'h0; registers[23] = 32'h0;
    registers[24] = 32'h0; registers[25] = 32'h0; registers[26] = 32'h0; registers[27] = 32'h0;
    registers[28] = 32'h0; registers[29] = 32'h0; registers[30] = 32'h0; registers[31] = 32'h0;
  end

endmodule
```

Appendix 5

```
module ALU(
  input wire [31:0] REG1,
  input wire [31:0] REG2,
  input wire [31:0] immediate,
  input wire ALU_OP,
  input wire [2:0] FUNCTION,
  input wire clk,
  output reg negative_flag,
  output reg carry_flag,
  output reg zero_flag,
  output reg [31:0] ALU_result
);


  always @(posedge clk) begin

    reg [31:0] x;
    if (ALU_OP == 1'b1) begin
      x <= immediate;
    end

    else begin
      x <= REG2;
    end

    if(ALU_OP == 1'b1)begin
       zero_flag <= (REG1 == immediate);
```

```
     negative_flag <= (REG1 < immediate);
end
else begin
zero_flag <= (REG1 == REG2);
negative_flag <= (REG1 < REG2);
end


     if(FUNCTION == 3'b000 && ALU_OP == 1'b1)begin
      ALU_result <= (immediate & REG1);
end
if(FUNCTION == 3'b000 && ALU_OP == 1'b0)begin
     ALU_result <= (REG2 & REG1);
end

if(FUNCTION == 3'b001 && ALU_OP == 1'b1)begin
      ALU_result <= (immediate + REG1);
end

if(FUNCTION == 3'b001 && ALU_OP == 1'b0)begin
     ALU_result <= (REG2 + REG1);
end


   case (FUNCTION)

    3'b010: ALU_result <= (REG1 - REG2);
    3'b011: ALU_result <= (REG1 << immediate);
    3'b100: ALU_result <= (REG1 >> immediate);
    3'b101: ALU_result <= (REG1 << REG2);
    3'b110: ALU_result <= (REG1 >> REG2);

  endcase

  carry_flag <= (ALU_result[31] == 1'b1);

end
endmodule
```

Appendix 6

```
module stack(
    input wire clk,
    input wire jal,
    input wire stop,
    input wire [31:0] pc,
    output reg [31:0] next_pc
);

reg [4:0] sp;
reg [31:0] stack_mem [0:31];

        always @(posedge clk) begin
        if (jal) begin
            stack_mem[sp] <= pc;
            sp <= sp + 1;
        end else if (stop) begin
            next_pc <= stack_mem[sp-1];
            sp <= sp - 1;
        end
    end
endmodule
```

Appendix 7

```
module memory (
  input [31:0] address,
  input [31:0] data_in,
  input flag_read,
  input flag_write,
  input clk,
  output reg [31:0] memory_output
);

  reg [31:0] Memory [0:31];

  integer i;

  initial begin
    for (i = 0; i < 32; i = i + 1) begin
      Memory[i] = 0;
    end
  end
  always @(posedge clk) begin
    if (flag_read) begin
      memory_output <= Memory[address];
    end

    if (flag_write) begin
      Memory[address] <= data_in;
    end
  end
endmodule
```

Appendix 8

```
module mux2x1(
	input wire [32:0] in1,
	input wire [32:0] in2,
	input wire s,
	output reg [32:0] out
);

always @(*) begin
	if(s==0)
		out <= in1;
	else
		out <= in2;
end
endmodule
```

Appendix 9

```
module controlUnit (
  input wire [4:0] func,
  input wire [1:0] insType,
  input wire stopIN,
  output reg [1:0] PCSrc,
  output reg secReg,
  output reg regW,
  output reg ALUop,
  output reg [2:0] ALUfunc,
  output reg jal,
  output reg stopOUT,
  output reg memRead,
  output reg memWrite,
  output reg rbData
);

always @(*) begin
	stopOUT <= stopIN;

	if(insType == 2'b00) begin
		if(stopIN==1)
			PCSrc <= 2'b11;
		else
			PCSrc <= 2'b00;
		if(func==5'b00011)
			regW <= 0;
		else
			regW <= 1;
		secReg <=0;
		ALUop <= 0;
		jal <= 0;
		memRead <= 0;
		memWrite <= 0;
		rbData <= 0;
	end

	else if(insType == 2'b01)begin
		if(stopIN==1) begin
			PCSrc <= 2'b11;
		end
		else begin
			if(func == 5'b00100)
				PCSrc <= 2'b01;
			else
				PCSrc <= 2'b00;
		end
		if(func < 5'b00011)
			regW <= 1;
		else
			regW <= 0;
		if(func == 5'b00100)
			ALUop <= 0;
		else
			ALUop <= 1;
```

```
            secReg <=1;
            jal <= 0;
            if(func == 5'b00010)begin
                        memRead <= 1;
               memWrite <= 0;
               rbData <= 1;
            end
            else if(func == 5'b00011)begin
                        memRead <= 0;
               memWrite <= 1;
               rbData <= 0;
            end
            else begin
                        memRead <= 0;
               memWrite <= 0;
               rbData <= 0;
            end
end

else if(insType == 2'b10)begin
            if(stopIN==1)
                        PCSrc <= 2'b11;
            else
                        PCSrc <= 2'b10;
            secReg <=0;
            regW <= 0;
            if(func == 5'b0)
                        jal <= 0;
            else
                        jal <= 1;
            ALUop <= 0;
            memRead <= 0;
            memWrite <= 0;
            rbData <= 0;
end

else begin
            if(stopIN==1)
                        PCSrc <= 2'b11;
            else
                        PCSrc <= 2'b00;
            secReg <=0;
            regW <= 1;
```

```
                if(func[1] == 0)begin
                        ALUop <= 0;
                end
                else begin
                        ALUop <= 1;
                end
                jal <= 0;
                memRead <= 0;
                memWrite <= 0;
                rbData <= 0;
        end


        if(insType == 2'b11)begin
                ALUfunc <= func[2:0] + 3'b011;
        end
        else begin
                if(func == 5'b0)begin
                        ALUfunc <= 3'b000;
                end
                else if(func == 5'b00001)begin
                        ALUfunc <= 3'b001;
                end
                else if(func == 5'b00010 || func == 5'b00011)begin
                        if(insType == 2'b00)
                                ALUfunc <= 3'b010;
                        else
                                ALUfunc <= 3'b001;
                end
                else begin
                        ALUfunc <= 3'b010;
                end
        end

    end

    endmodule
```

Appendix 10

```
    module PROCESSOR(
      input wire clock,
  output reg IF_clk,
  output reg ID_clk,
  output reg EX_clk,
  output reg MEM_clk,
  output reg WB_clk,
  output reg [31:0] jump_address,
  output reg [31:0] jar_address,
  output reg [31:0] next_pc,
  output reg [31:0] instruction,
  output reg Z,
  output reg N,
  output reg C,
  output reg [4:0] R1,
  output reg [4:0] R2,
  output reg [4:0] R3,
  output reg [1:0] ins_type,
  output reg [4:0] func ,
 output reg [31:0] immediate,
  output reg [31:0] bus_a,
  output reg [31:0] bus_b,
  output reg [31:0] bus_w,
  output reg [1:0] PC_src,
  output reg sec_reg,
  output reg reg_w,
  output reg ALU_op,
  output reg [2:0] ALU_func,
  output reg jal_flag,
  output reg stop_stop,
  output reg mem_read,
  output reg mem_write,
  output reg rb_data,
  output reg [31:0] ALU_result
  //output reg [31:0] memory
);

  reg [31:0] jump_addressR;
  reg [31:0] jar_addressR;
  reg ZR;
  reg [1:0] ins_typeR;
  reg [4:0] funcR;
  reg stopR;
  reg [31:0] bus_wR;
  reg [1:0] PC_srcR;
  reg reg_wR;

  wire IF_clkS;
  wire ID_clkS;
  wire EX_clkS;
  wire MEM_clkS;
  wire WB_clkS;
  wire [31:0] jump_addressS;
  wire [31:0] jar_addressS;
  wire [31:0] next_pcS;
  wire [31:0] instructionS;
  wire ZS;
  wire NS;
  wire CS;
  wire [4:0] R1S;
  wire [4:0] R2S;
  wire [4:0] R3S;
  wire [1:0] ins_typeS;
  wire [4:0] funcS;
  wire [31:0] immediateS;
  wire stopS;
  wire [31:0] bus_aS;
  wire [31:0] bus_bS;
  wire [31:0] bus_wS;
  wire [1:0] PC_srcS;
  wire sec_regS;
  wire reg_wS;
  wire ALU_opS;
  wire [2:0] ALU_funcS;
  wire jal_flagS;
  wire stop_flagS;
  wire mem_readS;
  wire mem_writeS;
  wire rb_dataS;
  wire [31:0] ALU_resultS;
  wire [31:0] memory_outputS;


  clkOrganizer M1( clock, funcR, ins_type, IF_clkS, ID_clkS, EX_clkS, MEM_clkS,
WB_clkS);

  instructionFetch M2(jump_addressR, jar_addressR, PC_srcR, ZR, IF_clkS, next_pcS,
instructionS);
```

```
  Decode M3(ID_clkS, instructionS, next_pcS, R1S, R2S, R3S, ins_typeS, funcS,
immediateS, stopS, jump_addressS);

  controlUnit M4(funcS, ins_typeS, stopS, PC_srcS, sec_regS, reg_wS, ALU_opS,
ALU_funcS, jal_flagS, stop_flagS, mem_readS, mem_writeS, rb_dataS);

  registerFile M5(R1S, R2S, R3S, sec_regS, reg_wR, WB_clkS, bus_wR, bus_aS, bus_bS);

  ALU M6(bus_aS, bus_bS, immediateS, ALU_opS, ALU_funcS, EX_clkS, NS, CS, ZS,
ALU_resultS);

  stack M7(EX_clkS, jal_flagS, stop_flagS, next_pcS, jar_addressS);

  memory M8(ALU_resultS, bus_bS, mem_readS, mem_writeS, MEM_clkS,
memory_outputS);

  mux2x1 M9(ALU_resultS, memory_outputS, rb_dataS, bus_wS);

  always @* begin
      jump_addressR <= jump_addressS;
    jar_addressR <= jar_addressS;
    ZR <= ZS;
    ins_typeR <= ins_typeS;
    funcR <= funcS;
    stopR <= stopS;
    bus_wR <= bus_wS;
    PC_srcR <= PC_srcS;
    reg_wR <= reg_wS;
    IF_clk <= IF_clkS;
    ID_clk <= ID_clkS;
    EX_clk <= EX_clkS;
    MEM_clk <= MEM_clkS;
    WB_clk <= WB_clkS;
    jump_address <= jump_addressS;
    jar_address <= jar_addressS;
    next_pc <= next_pcS;
    instruction <= instructionS;
    Z <= ZS;
    N <= NS;
    C <= CS;
    R1 <= R1S;
    R2 <= R2S;
    R3 <= R3S;
    ins_type <= ins_typeS;
    func <= funcS;
    immediate <= immediateS;
    bus_a <= bus_aS;
    bus_b <= bus_bS;
    bus_w <= bus_wS;
    PC_src <= PC_srcS;
    sec_reg <= sec_regS;
    reg_w <= reg_wS;
    ALU_op <= ALU_opS;
    ALU_func <= ALU_funcS;
    jal_flag <= jal_flagS;
    mem_read <= mem_readS;
    mem_write <= mem_writeS;
    rb_data <= rb_dataS;
    ALU_result <= ALU_resultS;
    //memory <= memory_outputS;
  end


initial begin
    jump_addressR = 32'b0;
    jar_addressR = 32'b0;
    ZR = 1'b0;
    ins_typeR = 2'b0;
    funcR = 5'b0;
    bus_wR = 32'b0;
    PC_srcR = 2'b0;
    reg_wR = 1'b0;
end


endmodule
```