Faculty of Engineering and Technology
Electrical and Computer Engineering Department Computer Architecture ENCS4370

Project #2

design and verify a simple RISC processor in Verilog

Prepared by:

Ibaa Taleeb	1203073
Sewar AbuEid	1200043
Rahaf Naser	1201319

Instructor: Dr.Ayman Hroub

Section:1

Date: 18-01-2024

Table of Contents

List of Figures

1.Designs and Testing for Stages

1.1 Instruction Memory

The Instruction Memory, also known as the Instruction Cache, is an essential element within the data path of a computer system. Its primary function is to store and supply instructions for execution, facilitating swift access and efficient processing. Instruction Memory plays a pivotal role in the execution of programs.

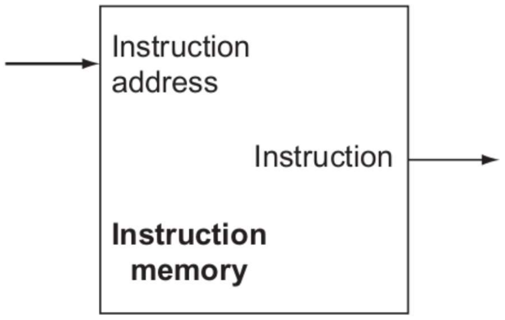


Figure 1:Instruction Memory

1.2 Program Counter

A PC (program counter) processor has a register that holds the address of the subsequent instruction that needs to be run from memory. It is also known as the instruction address register (IAR), instruction counter, and instruction pointer. It is a 16-bit register. A PC, also known as a program counter, is a digital counter used to track the current execution point and complete operations rapidly. . It retains the memory address of the subsequent instruction slated for fetching and execution. As a fundamental aspect of control flow, the PC Register dictates the sequence of instructions and undergoes updates during the fetch stage. This functionality enables the execution of branching and jumping instructions, facilitating alterations in the control flow within the program. In essence, the PC Register is instrumental in overseeing the orderly execution flow of the program.

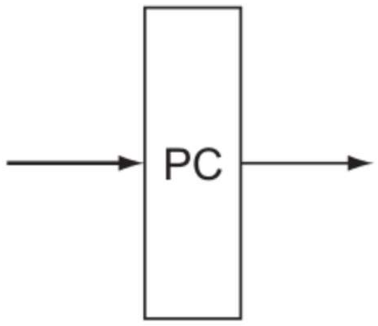


Figure 2: Program counter

1.3 MUX 2X1

The 2x1 MUX, or a 2-to-1 multiplexer, is a frequently utilized component in the data path of a computer system. function involves choosing one input signal from a pair based on a select signal. Within the data path, it is frequently employed for operations such as data selection and routing. By enabling the selection of diverse data sources through control signals, it enhances the flexibility and functionality of the data path.

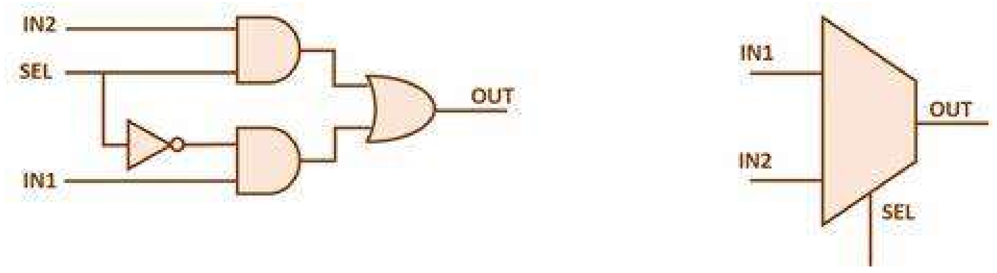


Figure 3:Mux 2x1

1.3 MUX 4X1

The 4x1 MUX, also known as a 4-to-1 multiplexer, is a frequently utilized component in the data path of a computer system. Its primary role involves the selection of one input signal from a set of four, determined by a select signal. Within the data path, it finds application in activities like instruction decoding and accessing the register file. This multiplexer facilitates signal selection and routing, thereby enhancing the flexibility and functionality of the data path.

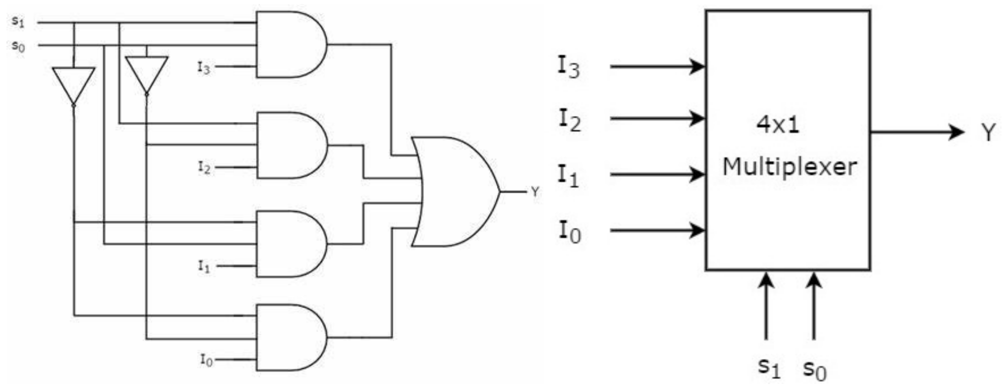


Figure 4:mux 4x1

1.4 Register File

The register file comprises a collection of registers employed to temporarily store data between memory and the functional units. It also includes the circuits responsible for writing to and reading from these registers. Additionally, the inputs and outputs of the register file, encompassing destination and source registers, have already been configured.

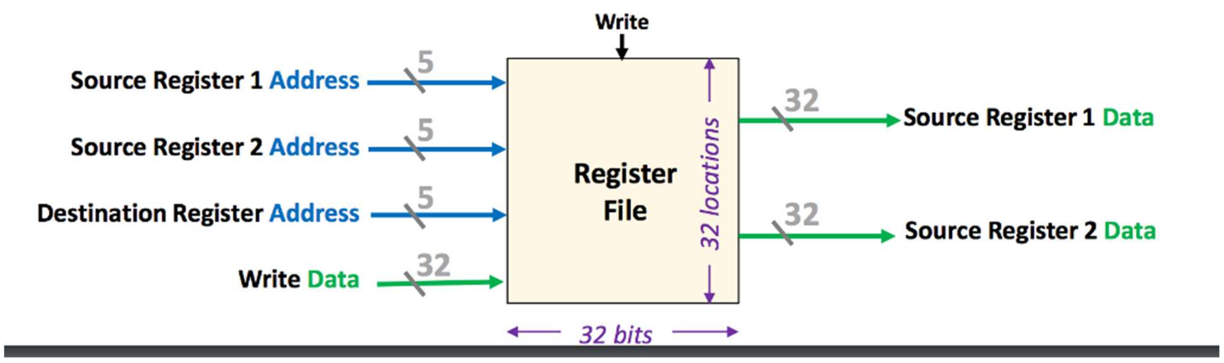


Figure 5:Register File

1.6 Data Memory

The Data Memory stage is a crucial component in the data path of a computer system. It facilitates the reading and writing of data to and from the main memory. During this stage, data is transferred between the processor and memory subsystem. Operations such as fetching data, storing data, and performing necessary conversions are performed. The Data Memory stage plays a vital role in executing instructions that involve memory operations, ensuring correct data access and manipulation. In this program the memory is organized into three segments:

- Static data segment
- Code segment
- Stack segment

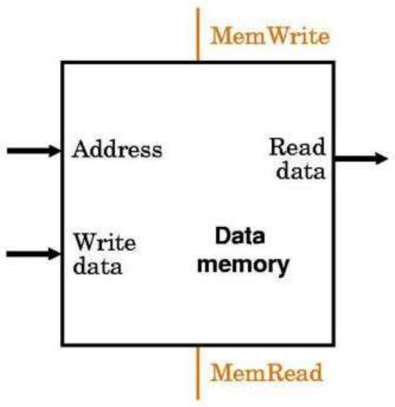


Figure 6: Data Memory

1.7 Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) is like the brain center of a computer's main processor. Its main job is to handle all the math and logic tasks with binary numbers. So, it can add, subtract, multiply, divide, and compare numbers. This is super important because it allows the computer to do all kinds of calculations and decision-making. When a computer is following instructions, the ALU kicks into action during a stage where it does the necessary math or logic stuff for each task.The Zero Flag

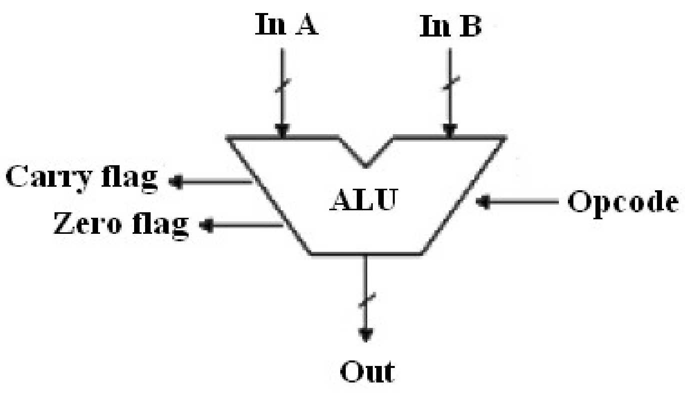


Figure 7: ALU

1.8 Extender 16

Extender 16 plays a vital role in the data path when there is a requirement to expand the size of a data signal by 16 bits. It guarantees compatibility in data size, performs sign extension, aligns with broader data paths, and facilitates expanded range or precision in operations. The inclusion of the Extender 14 ensures that the data path can effortlessly manage larger data sizes, preserve sign integrity, and support intricate calculations, thereby improving overall efficiency and precision in data processing.

1.9 control unit

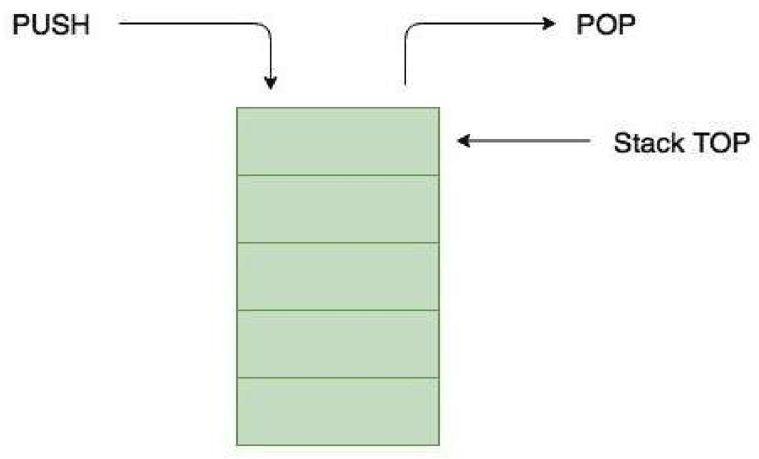
The control unit is a pivotal component within the data path of a computer system. It makes sure all the parts work together smoothly by telling them what to do. It manages things like processing data, using memory, and dealing with input/output. It reads and understands instructions from the program counter, then guides the different parts like the ALU, registers, and memory to do their jobs correctly. Basically, the control unit is the leader, making sure the computer follows the plan and works well.

1.10 Registers

The Register, serving as a buffer between stages, holds significant importance in a multi-cycle data path for several crucial reasons. Firstly, it facilitates the synchronization of operations. In a multi-cycle setup, different stages carry out operations in distinct clock cycles. The Register ensures a smooth flow of data between stages by allowing each stage to complete its operation before passing the data to the next one. This synchronization guarantees that each stage receives accurate data at the appropriate time. Additionally, the Register plays a vital role in timing control, helping regulate the timing of data propagation between stages. This control ensures the proper sequencing of operations, preventing issues like data hazards or corruption. By providing a controlled and synchronized transfer of data, it maintains the integrity of the overall process. Moreover, the Register serves as a storage element for intermediate results generated at various stages in a multi-cycle data path. This preservation of intermediate results allows for effective data dependency management and ensures the correct flow of data through the entire data path.

1.11 Stack And Stack Pointer (SP)

In this computer system, there's a 32-bit special purpose register known as the Stack Pointer (SP), and it's crucial for managing the stack. The stack is a Last In First Out (LIFO) data structure, meaning the last item added is the first one to be removed. The SP keeps track of the topmost empty element in the stack. As a programmer, you can see and manipulate this register.. The Stack segment is particularly important as it holds information like return addresses and registers' values during function calls. The system provides explicit instructions for the programmer to push or pop elements onto or from the stack, allowing for efficient management of data and control flow in the program.



2.Control Signals Table

num	inst	Op1	PcCont	Ext	Alusrc	AluOP	M.rd	M.Wr	MemAdd	DataIn	W.port1	W.port2	Wb.data	SPorAlu2	Alup
1	and	000000	00	x	00	01	0	0	x	x	1	0	0	x	X
2	add	000001	00	x	00	00	0	0	x	x	1	0	0	x	x
3	sub	000010	00	x	00	10	0	0	x	x	1	0	0	x	X
4	andi	000011	00	0	01	01	0	0	x	x	1	0	0	x	X
5	addi	000100	00	0	01	00	0	0	x	x	1	0	0	x	X
6	Lw	000101	00	1	01	00	1	0	0	x	1	0	1	x	X
7	Lw.poi	000110	00	1	01	00	1	0	0	x	1	1	1	x	X
8	Sw	000111	00	1	01	00	0	1	0	0	0	0	x	x	X
9	Bgt	001000	01	1	10	10	0	0	x	x	0	0	x	x	X
10	Blt	001001	01	1	10	10	0	0	x	x	0	0	x	x	X
11	Beq	001010	01	1	10	10	0	0	x	x	0	0	x	x	X
12	Bne	001011	01	1	10	10	0	0	X	x	0	0	x	x	X
13	Jmp	001100	10	x	x	x	0	0	x	x	0	0	x	x	X
14	Call	001101	10	x	x	x	0	0	1	1	0	0	x	1	0
15	Ret	001110	11	x	x	x	0	0	x	x	0	0	x	0	1
16	Push	001111	00	x	x	x	0	1	1	0	0	0	x	1	x
17	pop	010000	00	x	x	x	1	0	1	1	0	0	1	0	1

- PcCount :
  00 =➔ pc=pc+1
  01 ➔ pc=pc+sign_ext(imm16)
  10➔pc=contacx
  11➔pc=top of the stack

- ALU src :
  00➔ Rs2
  01➔extendedImm(16bit)
  10➔ Rd

- ALUop:
  00➔add
  01 ➔ and
  10➔ sub

- AluOp2
  0➔ add 1
  1➔ sub

- SPorALU
  0➔ SP
  1 ➔ SP added or subset by 1

- Ext
  0➔ unsigned
  1➔ signed

- Wport 1 ➔ Rd (To enable writing the value of Rd on the register file)

- Wport 2 ➔ Rs1 (To enable writing the value of Rs1 on the register file)

- Wb data
  0➔ result from Alu
  1➔ from memory

- Data in
  0➔ rd
  1➔ pc+1

- MemAddress
  0➔ alu result
  1➔ result of mux SporAlu

3. Expressions for Control Bits

- pcCount[0] : (instruction==Jmp)&(instruction==Call)&(instruction==Ret)
- pcCount[1]:(instruction==Bgt)&(instruction==Blt)&(instruction==Beq)&(instruction==B ne) &(instruction==Ret)
- Ext: (instruction==lw)&(instruction==lw.poi)&(instruction==Sw)&(instruction==Bgt) &(instruction==Bne)&(instruction==Blt)&(instruction==Beq)
- Alusrc[0]: (instruction==Bgt)&(instruction==Blt)&(instruction==Beq)&(instruction==Bne)
- Alusrc[1]:(instruction==andi)&(instruction==addi)&(instruction==lw)&(instruction==lw. poi)&(instruction==Sw)
- AluOP[0]:(instruction==sub)&(instruction==Bgt)&(instruction==Blt)&(instruction==Beq)&(instruction==Bne)
- AluOP[1]: (instruction==and)& (instruction==andi)
- M.rd: (instruction==lw)&(instruction==lw.poi)& (instruction==pop)
- M.Wr: (instruction==Sw)& (instruction==push)
- MemAdd: (instruction==Call)&(instruction==push) & (instruction==pop)
- DataIn: (instruction==Call)& (instruction==pop)
- W.port1:(instruction==R-type) &(instruction==andi) &(instruction==addi) &(instruction==lw) &(instruction==lw.poi)
- W.port2: (instruction==lw.poi)
- Wb.data: (instruction==lw)&(instruction==lw.poi)& (instruction==pop)
- SPorAlu2: (instruction==Call)&(instruction==push)
- AluSp: (instruction==Ret)&(instruction==pop)

4. The Finite State Machine

Here are the stages are the states in which:

- **Instruction Fetch:** The processor fetches the instruction from memory and loads it into the instruction register.
- **Instruction Decode:** The processor decodes the instruction, determines the type of instruction and the operands needed, and stores the relevant information in internal registers.
- **Execution:** The processor performs the necessary operations, which may involve multiple clock cycles, to complete the instruction.
- **Memory Access:** If the instruction requires data to be read from or written to memory, the processor accesses the memory to perform the operation.
- **Write Back:** The processor writes the result of the operation back to the appropriate internal register or memory location.

The finite state machine below shows the transition between stages from fetch to write back and the behavior of each instruction respectively.

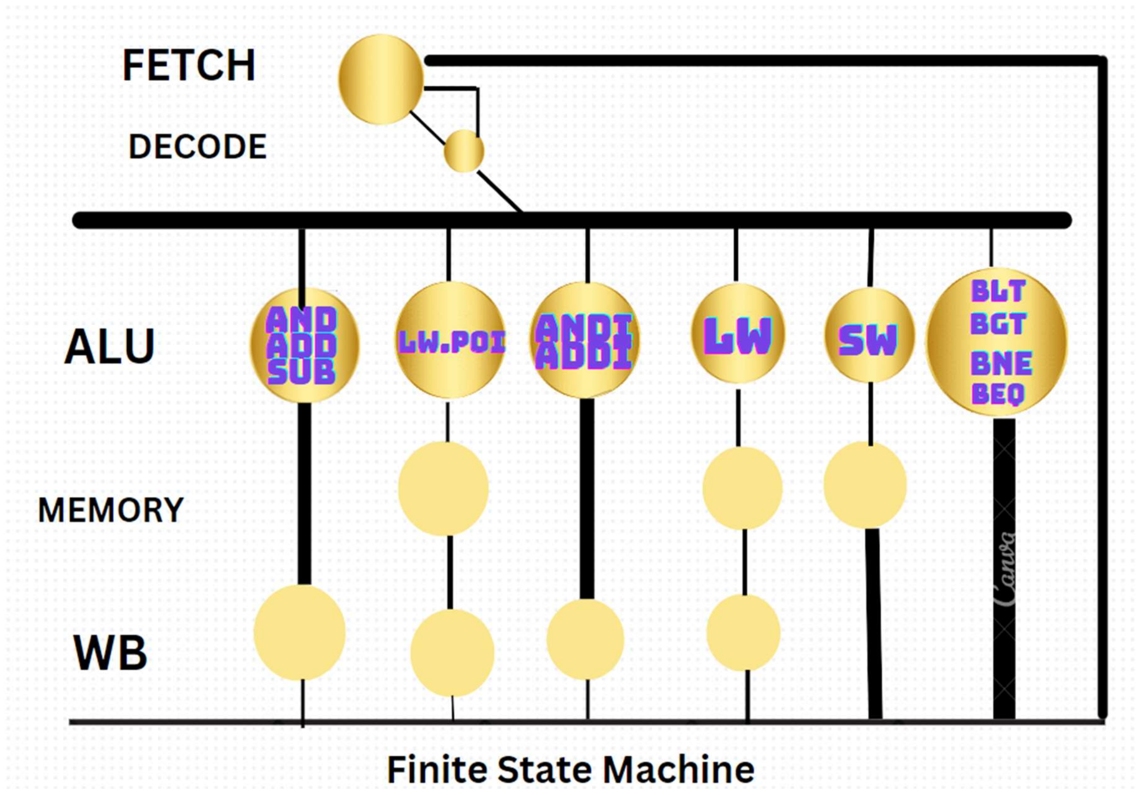
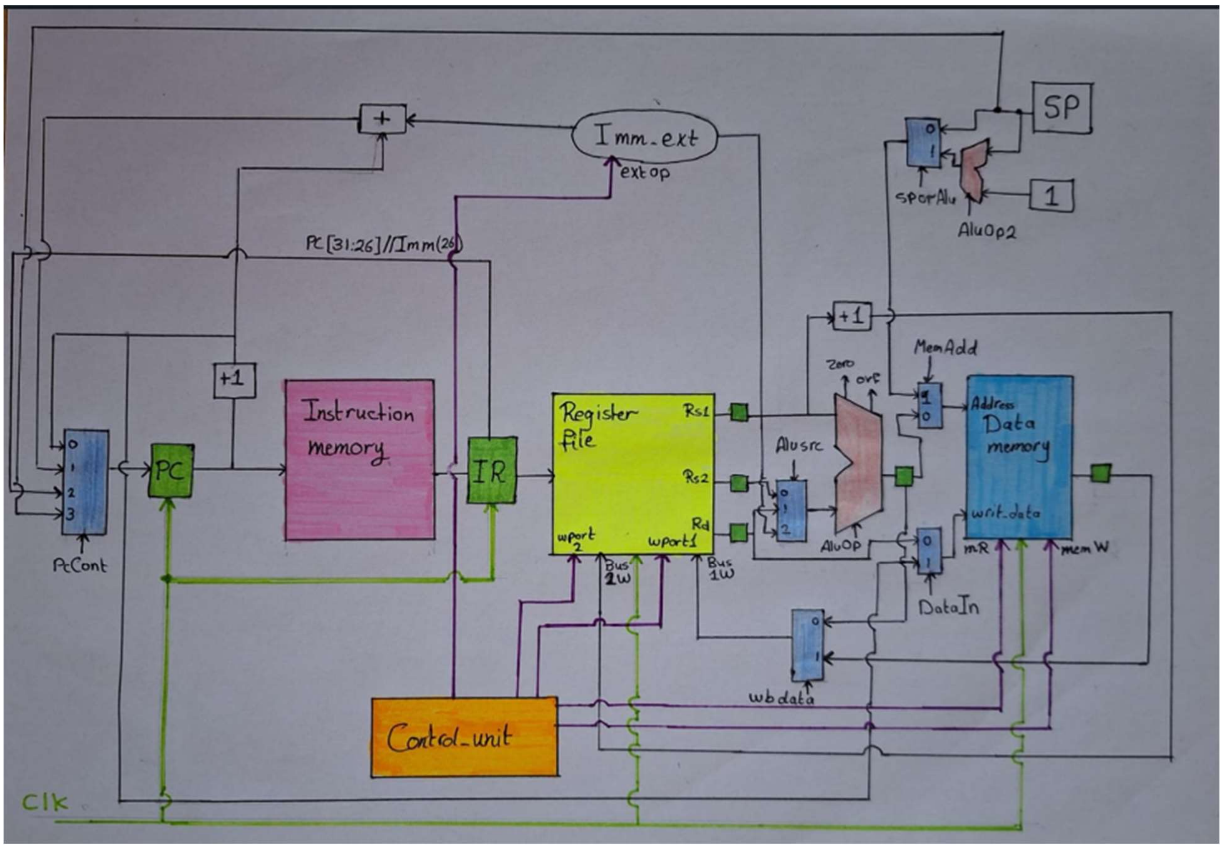


Figure1:Finite State Machine

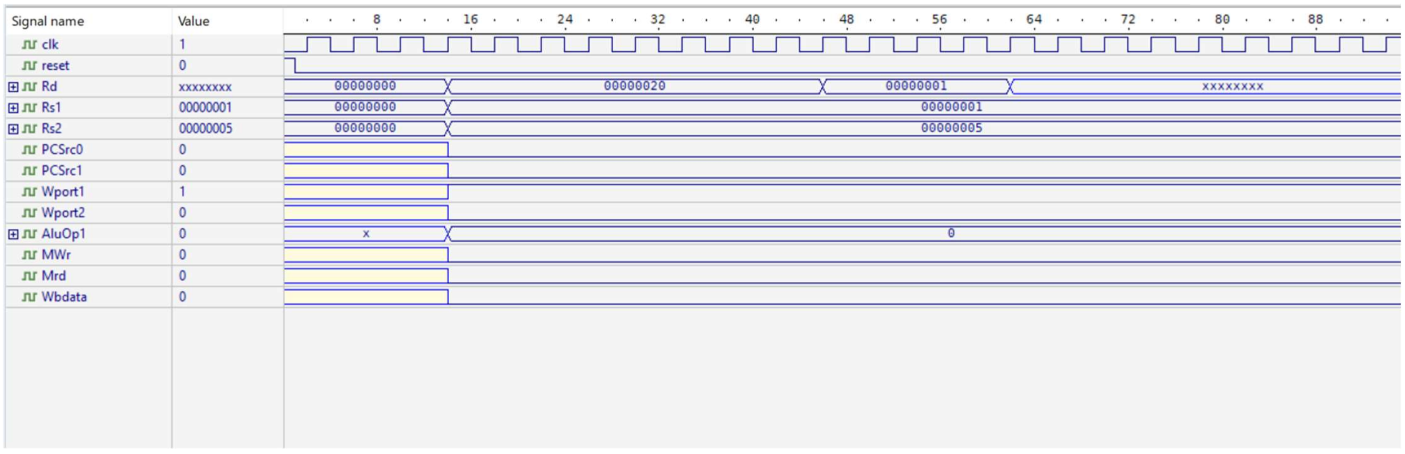
The table of control unit shows the corresponding value to control signal and instruction implemented. While the figure above shows the transition per instruction, for example 'and, add, sub' instructions being fetch decoded executed and write back to register file then return to fetch state for the next instruction, while 'j' is being fetched decoded and then return to fetch again.

5.Final Data path



Test Each Instruction

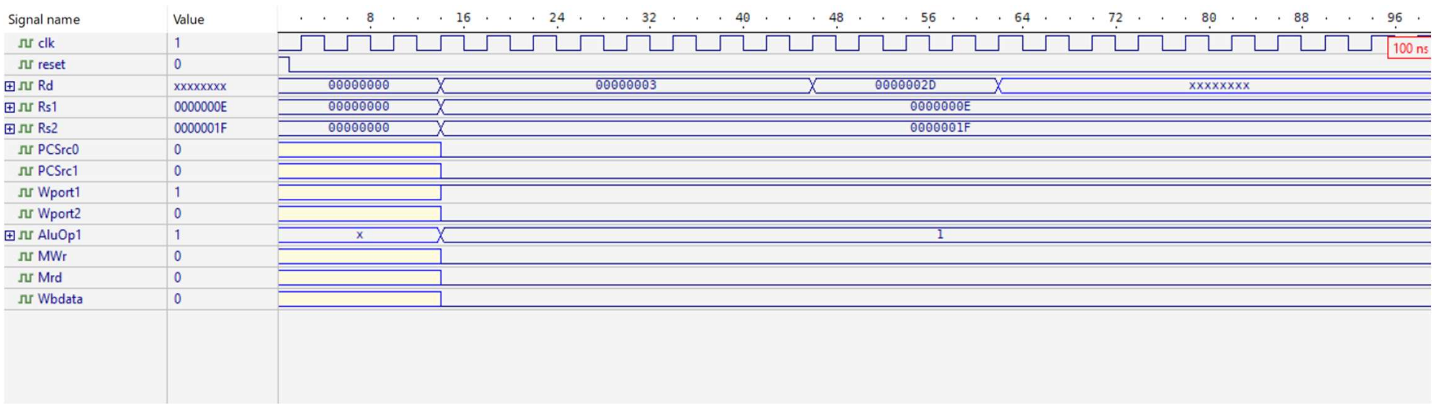
➔ AND instruction Simulation



Signal name	Value
clk	1
reset	0
Rd	xxxxxxxx
Rs1	00000001
Rs2	00000005
PCSrc0	0
PCSrc1	0
Wport1	1
Wport2	0
AluOp1	0
MWr	0
Mrd	0
Wbdata	0

The result in Rd is (00000001) which is the and result of (00000001) & (0000005).

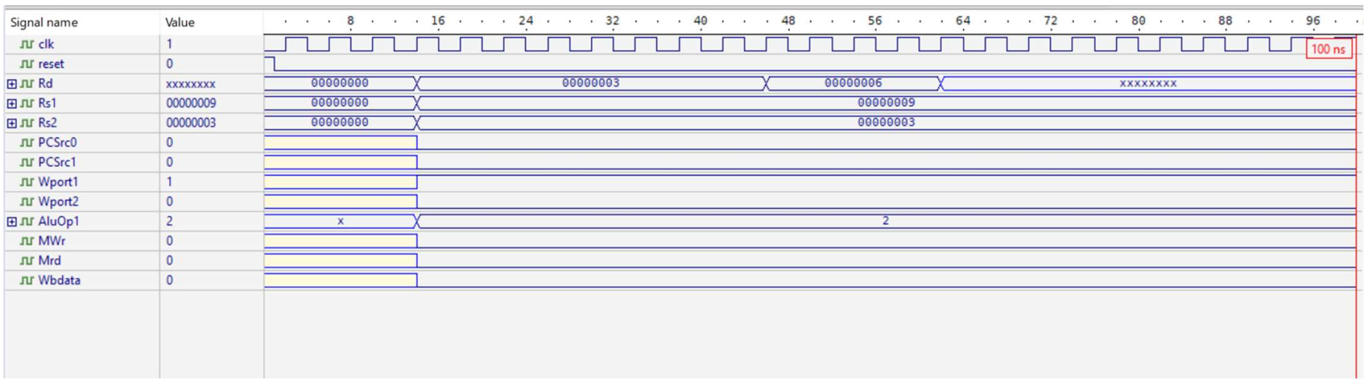
➔ADD instruction Simulation



Signal name	Value
clk	1
reset	0
Rd	xxxxxxxx
Rs1	0000000E
Rs2	0000001F
PCSrc0	0
PCSrc1	0
Wport1	1
Wport2	0
AluOp1	1
MWr	0
Mrd	0
Wbdata	0

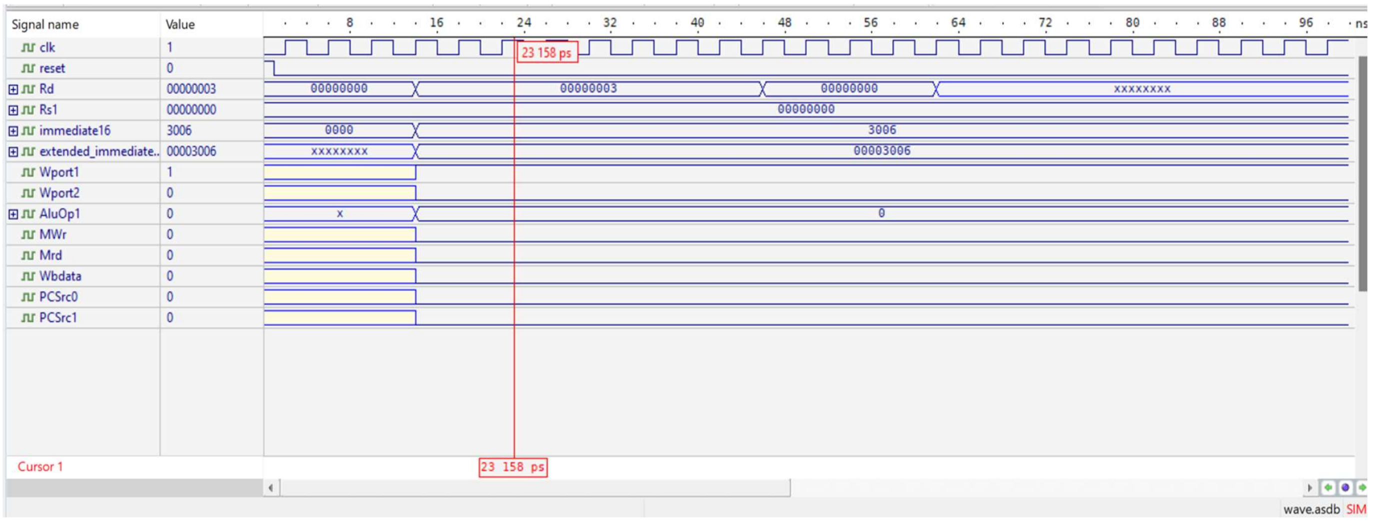
Note that the content of Rd as a result is 0000002D which equals 45 which is the summation of 31 (0000001F) and 14 (0000000E).

➔SUB instruction Simulation



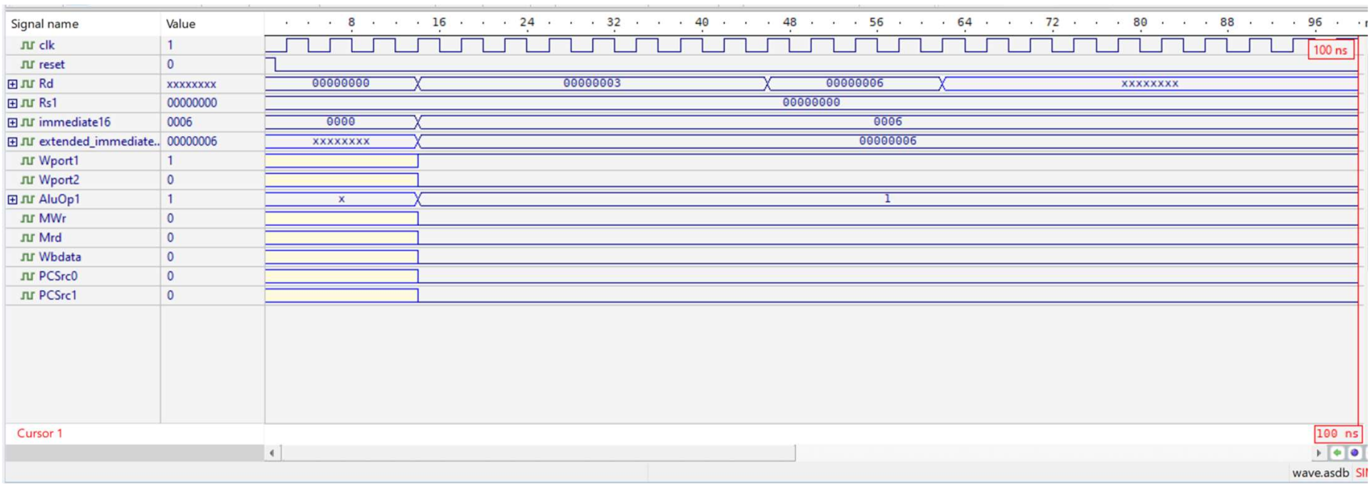
Signal name	Value
clk	1
reset	0
Rd	xxxxxxxx
Rs1	00000009
Rs2	00000003
PCSrc0	0
PCSrc1	0
Wport1	1
Wport2	0
AluOp1	2
MWr	0
Mrd	0
Wbdata	0

➔ANDI instruction Simulation



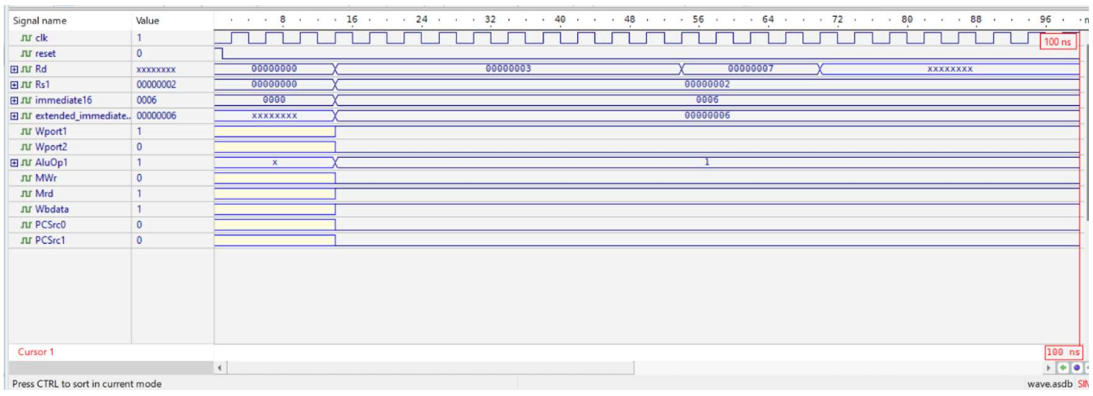
Signal name	Value
clk	1
reset	0
Rd	00000003
Rs1	00000000
immediate16	3006
extended_immediate..	00003006
Wport1	1
Wport2	0
AluOp1	0
MWr	0
Mrd	0
Wbdata	0
PCSrc0	0
PCSrc1	0

➔ADDI instruction Simulation



Signal name	Value
clk	1
reset	0
Rd	xxxxxxxx
Rs1	00000000
immediate16	0006
extended_immediate..	00000006
Wport1	1
Wport2	0
AluOp1	1
MWr	0
Mrd	0
Wbdata	0
PCSrc0	0
PCSrc1	0

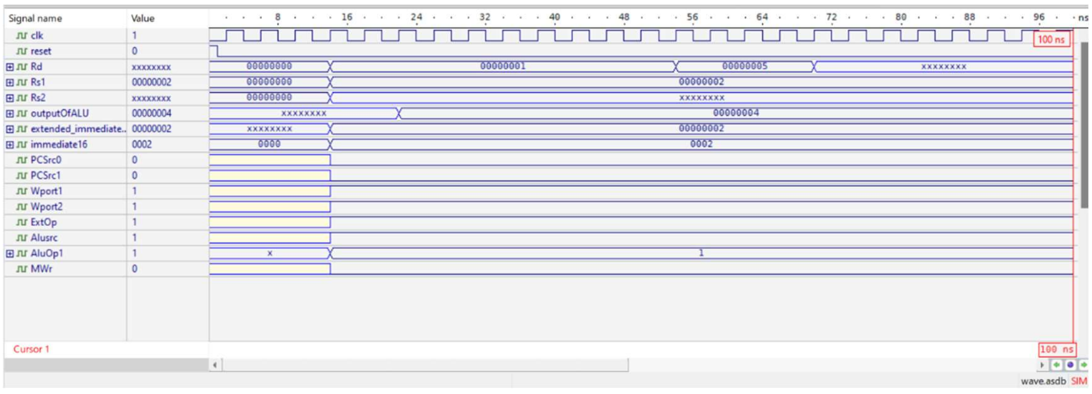
➔LW instruction Simulation



Signal name	Value
clk	1
reset	0
Rd	xxxxxxxx
Rs1	00000002
immediate16	0006
extended_immediate..	00000006
Wport1	1
Wport2	0
AluOp1	1
MWr	0
Mrd	1
Wbdata	1
PCSrc0	0
PCSrc1	0

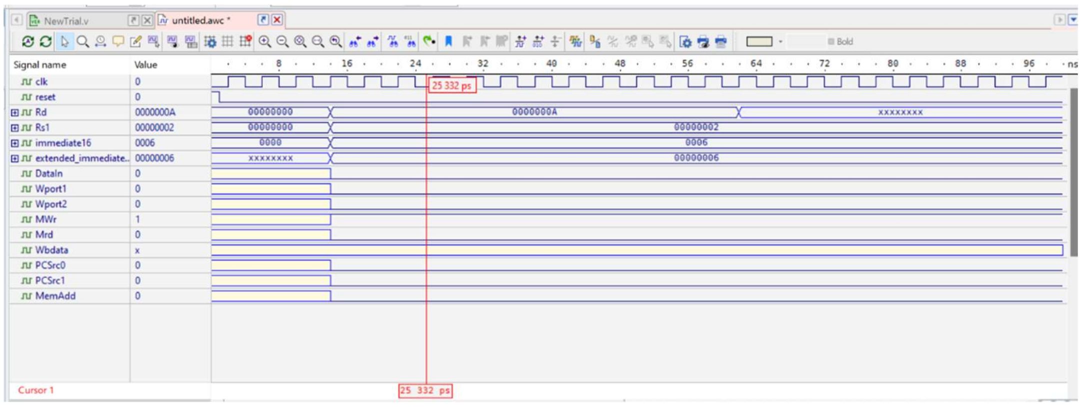
Press CTRL to sort in current mode

➔LW.POI



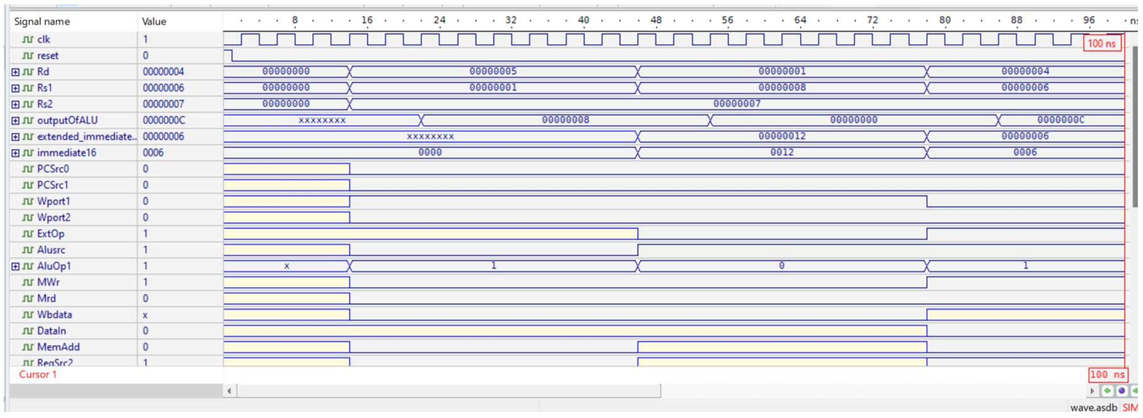
Signal name	Value
clk	1
reset	0
Rd	xxxxxxxx
Rs1	00000002
Rs2	xxxxxxxx
outputOfALU	00000004
extended_immediate..	00000002
immediate16	0002
PCSrc0	0
PCSrc1	0
Wport1	1
Wport2	1
ExtOp	1
Alusrc	1
AluOp1	1
MWr	0

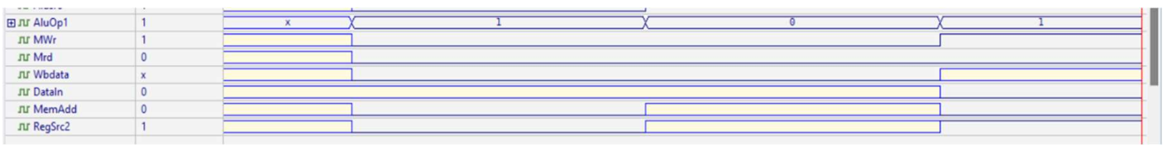
➔SW instruction simulation



Signal name	Value
clk	0
reset	0
Rd	0000000A
Rs1	00000002
immediate16	0006
extended_immediate..	00000006
DataIn	0
Wport1	0
Wport2	0
MWr	1
Mrd	0
Wbdata	x
PCSrc0	0
PCSrc1	0
MemAdd	0

Final Data Path testing





6. Conclusion

The project involved demonstrating the complete construction of the data path based on specified instructions. Each stage was developed with the necessary control signals to regulate their operations. A simulation was performed for every step, along with the design work for creating the required data path and the final state machine, all of which are detailed in the report.

7.Appendices

7.1. The Code for the data Path

```
// the first step is to define the global variables outside the module

reg [2:0]current_state;  // 0--> Instruction Fetch 1--> Instruction Decode 2-->Execution 3--> Memory
access 4--> Write Back Data

reg [2:0]next_state; // takes the same range of values of the current_state

reg [31:0]IR;  // ISA of 32-bit


module multiCycleProcessor(      // determine the control signals

input clk,input reset,

output reg [1:0] PcCont,output reg ExtOp,

output reg         Wport1,

output reg  Wport2,output reg Alusrc,output reg [1:0] AluOp1,

output reg MemAdd,output reg DataIn,output reg  Mrd,

output reg  MWr, output reg Wbdata , output reg SPorAlu2,

output reg AluSpOp,


///Instruction Decode Part

output reg [31:0] Rs1, // (Bus A )Output data from register 1

output reg [31:0] Rs2, // (Bus B) Output data from register 2

output reg [31:0] Rd, // (Bus W) Output data from register 3


 ///ALU Part

output reg [31:0] outputOfALU, // Just to verify that the ALU works properly

///immediate16 Part

output reg [31:0]extended_immediate16,

///immediate26 Part
);


 // variables will be used in the instruction fetch stage
reg [31:0] InstMem [0:31];          // bit width of each element in the array (InstMem) is 32-bit
//and The range [31:0] indicates that the least significant bit (LSB) is 0, and the most significant bit (MSB)
is 31.
reg [31:0] PC;
reg [31:0] SP;
reg [5:0] OpCode1;


// Decode Stage
reg [31:0] mux_2Output;
reg [31:0] toRegFile; // the data will be stored in the register file
reg [3:0] RegisterFile[0:31];  // 16 Registers 32-bit each
reg [3:0] RdAddress; // Address for Rd
reg [3:0] Rs1Address; // Address for RS1
reg [3:0] Rs2Address; // Address for RS2
reg [31:0] PCmuxOut;  // the decided value of the PC
reg [31:0] sequentialPC ;
reg [31:0] jumpTargetAddress;
reg [31:0] branchTargetAddress;
// Alu Part
reg [31:0] firstOperand;
reg [31:0] secondOperand;
reg zero;
reg overflow;


// Memory access part
```

```
reg [31:0] address,data_in;          // data_in represent the data itself which will enter the memory not the
selection             the same for address

reg [31:0] secondAddress,SPfinalResult; // this represent the output value of the SP mux

reg [31:0] data_out;

reg [31:0] data_memory [31:0];


//Extenders 16 and 26

reg [15:0] immediate16;

reg [25:0] immediate26;


// Stack Pointer Part


reg [31:0] SP_AluResult; // increment or decrement the SP

reg [31:0] out_SP_mux; // to select either sp or alu result


// variable i for the loop

reg [7:0] i; // Example declaration for an 8-bit register "i"


initial

        begin

                InstMem[0]= 32'h14854000;//LW

                InstMem[1]= 32'h04890000;//ADD

                InstMem[2]= 32'h01980000;//AND

                InstMem[3]= 32'h2998C000;//BEQ

                InstMem[4]= 32'h3D800000;//PUSH

                InstMem[5]= 32'h18048001;//LW.POI

                InstMem[6]= 32'h40800000;//POP

                InstMem[7]= 32'h09288000;//SUB

                InstMem[8]= 32'hC4480000;//BGT
```

```
                    InstMem[9]= 32'h0C000000;//ANDI
                    InstMem[10]= 32'h10440000;//ADDI
                    InstMem[11]= 32'h1C518000;//SW
                    InstMem[12]= 32'h0489C000;//BLT
                    InstMem[13]= 32'h34000028;//JMP
                    InstMem[14]= 32'h2D88000C;//CALL
                    InstMem[15]= 32'h2E98C000;//BNE
                    InstMem[16]= 32'h38000000;//RET

          end
          // addapt the clk setting to move between states
          always @(posedge clk)
case (current_state)
0:
next_state = 1;
1:
next_state = 2;
2:
next_state = 3;
3:
next_state = 4;
4:
next_state = 0;
endcase

 // change the current state to it's next at each clk       positive edge
always @(posedge clk)
current_state = next_state;
```

```
always @(posedge clk, posedge reset)

if (reset)

            begin

   current_state = 4;

    PC = 32'h00000000;

    Rs1 = 32'd0;

    Rs2 = 32'd0;

    Rd = 32'd0;

    mux_2Output = 32'h00000000;

    for (i = 0; i < 31; i = i + 1)

    RegisterFile[i] <= 32'h00000000;

end


///////////// Start Instruction tour Besmellah ////////////////////////

            else if (current_state == 0) begin

//instruction fetch

IR = InstMem[PC]; // the current instruction is the value of PC

OpCode1 = IR[31:26];     // the 6 MSB from IR always for the opcode



if (OpCode1 == 6'b000000)  begin ///AND


                        PcCont = 2'b00;

                        Alusrc = 2'b00;

                        AluOp1 = 2'b01;

                        Mrd = 1'b0;

      MWr = 1'b0;

                        Wport1 = 1'b1;
```

```
                    Wport2 = 1'b0;
      Wbdata = 1'b0;


end


else if (OpCode1 == 6'b000001)  begin//ADD
                    PcCont = 2'b00;
          Alusrc = 2'b00;
                    AluOp1 = 2'b00;
                    Mrd = 1'b0;
      MWr = 1'b0;
                    Wport1 = 1'b1;
                    Wport2 = 1'b0;
      Wbdata = 1'b0;


end


else if (OpCode1 == 6'b000010) begin      //SUB


             PcCont = 2'b00;
                    Alusrc = 2'b00;
                    AluOp1 = 2'b10;
                    Mrd = 1'b0;
      MWr = 1'b0;
                    Wport1 = 1'b1;
                    Wport2 = 1'b0;
      Wbdata = 1'b0;


end
```

```
else if (OpCode1== 6'b000011) begin       //ANDI

                PcCont = 2'b00;
                         ExtOp = 1'b0;
                         Alusrc = 2'b01;
                         AluOp1 = 2'b01;
                         Mrd = 1'b0;
       MWr = 1'b0;
                         Wport1 = 1'b1;
                         Wport2 = 1'b0;
       Wbdata = 1'b0;

end

else if (OpCode1 == 6'b000100) begin     //ADDI

                PcCont = 2'b00;
                         ExtOp = 1'b0;
                         Alusrc = 2'b01;
                         AluOp1 = 2'b00;
                         Mrd = 1'b0;
       MWr = 1'b0;
                         Wport1 = 1'b1;
                         Wport2 = 1'b0;
       Wbdata = 1'b0;

end
```

```
else if (OpCode1 == 6'b000101) begin   //LW

            PcCont = 2'b00;
                   ExtOp = 1'b1;
                   Alusrc = 2'b01;
                   AluOp1 = 2'b00;
                   Mrd = 1'b1;
     MWr = 1'b0;
                   MemAdd= 1'b0;
                   Wport1 = 1'b1;
                   Wport2 = 1'b0;
     Wbdata = 1'b1;

end

else if (OpCode1 == 6'b000110) begin   //LW.POI

            PcCont = 2'b00;
                   ExtOp = 1'b1;
                   Alusrc = 2'b01;
                   AluOp1 = 2'b00;
                   Mrd = 1'b1;
     MWr = 1'b0;
                   MemAdd= 1'b0;
                   Wport1 = 1'b1;
                   Wport2 = 1'b1;
     Wbdata = 1'b1;

end
```

```
else if (OpCode1 == 6'b000111) begin   //SW

                    PcCont = 2'b00;

                    ExtOp = 1'b1;

                    Alusrc = 2'b01;

                    AluOp1 = 2'b00;

                    Mrd = 1'b0;

     MWr = 1'b1;

                    MemAdd= 1'b0;

                    DataIn = 1'b0;

                    Wport1 = 1'b0;

                    Wport2 = 1'b0;


end


else if (OpCode1 == 6'b001000) begin    //BGT

                    PcCont = 2'b01;

                    ExtOp = 1'b1;

                    Alusrc = 2'b10;

                    AluOp1 = 2'b10;

                    Mrd = 1'b0;

     MWr = 1'b0;

                    Wport1 = 1'b0;

                    Wport2 = 1'b0;

end



                    else if (OpCode1 == 6'b001001) begin    //BLT

                    PcCont = 2'b01;
```

```
                        ExtOp = 1'b1;

                        Alusrc = 2'b10;

                        AluOp1 = 2'b10;

                        Mrd = 1'b0;

       MWr = 1'b0;

                        Wport1 = 1'b0;

                        Wport2 = 1'b0;


end


else if (OpCode1 == 6'b001010)begin     //BEQ

                        PcCont = 2'b01;

                        ExtOp = 1'b1;

                        Alusrc = 2'b10;

                        AluOp1 = 2'b10;

                        Mrd = 1'b0;

       MWr = 1'b0;

                        Wport1 = 1'b0;

                        Wport2 = 1'b0;


end


else if (OpCode1 == 6'b001011)  begin  //BNE

                        PcCont = 2'b01;

                        ExtOp = 1'b1;

                        Alusrc = 2'b10;

                        AluOp1 = 2'b10;

                        Mrd = 1'b0;

       MWr = 1'b0;
```

```
                        Wport1 = 1'b0;

                        Wport2 = 1'b0;


end


else if (OpCode1 == 6'b001100) begin   //JMP

                        PcCont = 2'b01;

                        Mrd = 1'b0;

       MWr = 1'b0;

                        Wport1 = 1'b0;

                        Wport2 = 1'b0;

end


else if (OpCode1 == 6'b001101) begin   //CALL

                        PcCont = 2'b01;

                        Mrd = 1'b0;

       MWr = 1'b0;

                        MemAdd= 1'b1;

                        DataIn = 1'b1;

                        Wport1 = 1'b0;

                        Wport2 = 1'b0;

                        SPorAlu2= 1'b1;

                        AluSpOp = 1'b0;

end


else if (OpCode1 == 6'b001110)begin    //RET

                        PcCont = 2'b11;

                        Mrd = 1'b0;

       MWr = 1'b0;
```

```
                    Wport1 = 1'b0;
                    Wport2 = 1'b0;
                    SPorAlu2= 1'b0;
                    AluSpOp = 1'b1;
end

else if (OpCode1 == 6'b001111) begin   //PUSH
                    PcCont = 2'b00;
                    Mrd = 1'b0;
      MWr = 1'b1;
                    MemAdd= 1'b1;
                    DataIn =1'b0;
                    Wport1 = 1'b0;
                    Wport2 = 1'b0;
                    SPorAlu2= 1'b1;

end

else if (OpCode1 == 6'b010000) begin   //POP
                    PcCont = 2'b00;
                    Mrd = 1'b1;
      MWr = 1'b0;
                    MemAdd= 1'b1;
                    DataIn =1'b1;
                    Wport1 = 1'b0;
                    Wport2 = 1'b0;
      Wbdata = 1'b1;
                    SPorAlu2= 1'b0;
                    AluSpOp = 1'b1;
```

```
end



end         // the end of Instruction Fetch stage


else if (current_state == 1) begin


            // Instruction Decode stage
            // get the values of the extended and unextended immediates
immediate16 = IR[29:14];
// initialy extend the value of immediate16 to 32-bit with zero's temporarly
extended_immediate16 = {16'b0000000000000000, immediate16}; // temporarly zero extension


immediate26 = IR[31:6];// This value will not be extended


// here, we will fill the registers as needed


RegisterFile[1]=32'h00000003;  // R1=3
RegisterFile[3]=32'h00000004;  // R3=4
RegisterFile[4]=32'h0000000E;  // R4=10
RegisterFile[5]=32'h00000007; // R5=7
RegisterFile[8]=32'h00000000; // R8=0
RegisterFile[9]=32'h00000001; // R9=1


///Getting the addresses for the RS1,RS2,Rd
RdAddress  = IR[9:6]; // is the address of(Rd) Register (write operand)
Rs1Address = IR[13:10]; //is the address of(Rs1) Register ==> first address for read operand
Rs2Address = IR[17:14]; // is the address of(Rs2) Register ==> second address for read operand
```

```
// get the source and the distenation registers from the register file via their addresses

Rs1 = RegisterFile[Rs1Address];

Rs2 = RegisterFile[Rs2Address];

Rd  = RegisterFile[RdAddress];


// determine the control signal which will determine the second AluSrc

case (Alusrc)

 2'b00: mux_2Output = Rs2;

 2'b01: mux_2Output = extended_immediate16; //Output of the Extender 16

 2'b10: mux_2Output = Rd;

endcase


// determine the control signal which will determine the PC Content

case (PcCont)

 2'b00: PCmuxOut = sequentialPC;  // pc=pc+1

 2'b01: PCmuxOut = branchTargetAddress;          // pc=pc+sign_ExtImm16            // this part will be done
in PC adder

 2'b10: PCmuxOut = jumpTargetAddress;              // pc=concatenation result

 2'b11: PCmuxOut = data_memory[SP];                            // pc= stack pointer


endcase


// determine the control signal which will determine the data_in to access the memory


case (DataIn)

            1'b0: data_in=   Rd;         // ALU RESULT

            1'b1: data_in=   PC+1; // To push up the address of the next instrucion


endcase
```

```
case (MemAdd)

             1'b0: address= outputOfALU;

             1'b1: address=   secondAddress;

endcase


case (SPorAlu2)

             1'b0: secondAddress = SP;

             1'b1: secondAddress = SPfinalResult; // either SP+1 or SP-1

endcase


case (AluSpOp)



            1'b0: SPfinalResult= SP+1;

            1'b1: SPfinalResult= SP-1;

endcase


case (Wbdata)  // to determine the output of the mux in the bottom

            1'b0: toRegFile= outputOfALU;

            1'b1: toRegFile= data_out;

endcase


// Now we will check if the instruction is JMP,

            if(OpCode1==6'b001100)begin


                        current_state = 4;

                        jumpTargetAddress={PC[31:26],immediate26};

                        PC =jumpTargetAddress;
```

```
        end        // the end of JMP Instruction Decode



// Now we will check if the instruction is CALL,
        if(OpCode1==6'b001101)begin


                jumpTargetAddress={PC[31:26],immediate26};
                PC =jumpTargetAddress;


        end        // the end of CALL Instruction Decode



end   // the end of Instruction Decode stage


else if (current_state == 2)  begin
        // ALU STAGE === Execution


        firstOperand=Rs1;
        secondOperand= mux_2Output;


        case(AluOp1)
          2'b00: outputOfALU = firstOperand + secondOperand;     // Addition
          2'b01: outputOfALU = firstOperand & secondOperand;     // Bitwise AND
          2'b10: outputOfALU = firstOperand - secondOperand;     // SUBTRACTION
          default: outputOfALU = 32'b0;                 // Default case: result is 0


         endcase


         if (outputOfALU==0)begin
```

```
                    zero=1'b1; // set the zero flag if the output of the alu is zero

                    end

             if (outputOfALU < 0) begin

                    overflow=1'b1; // set the overflow flag if the output of alu is negative, that's mean
Rs1>Rd

                    end

            // Now we will check if the instruction is BEQ
                    if (OpCode1==6'b001010) begin

                           if(zero == 1'b1 ) begin

                                  // the branch is taken and the two numbers are equal
                                  branchTargetAddress = PC + extended_immediate16;
                                  current_state = 4;
                                  PC=      branchTargetAddress;

                           end

                    else if (zero == 1'b0 ) begin

                           // Branch doesnt taken
                           PC=PC+1;
                           current_state = 4;
                    end

                end // the end of BEQ


// Now we will check if the instruction is BNE
                if (OpCode1==6'b001011) begin

                    if(zero == 1'b0 ) begin

                        // the branch is taken and the two numbers are NOT equal
                        branchTargetAddress = PC + extended_immediate16;
                        current_state = 4;
                        PC=     branchTargetAddress;

                    end

                else if (zero == 1'b1 ) begin

                    // Branch doesnt taken
                    PC=PC+1;
                    current_state = 4;

                    end

                end // the end of BNE
```

```
// Now we will check if the instruction is BGT // if Rd> Rs1 then the result of the alu is negative then
overflow is 1
        if (OpCode1==6'b001000)         begin

                if(overflow == 1'b1 ) begin

                                // the branch is taken and the two numbers are equal
                                branchTargetAddress = PC + extended_immediate16;
                                current_state = 4;
                                PC=     branchTargetAddress;

                        end

                else if (overflow == 1'b0 ) begin

                        // Branch doesnt taken
                        PC=PC+1;
                        current_state = 4;

                        end

        end /// the end of BGT

// Now we will check if the instruction is BLT // if Rd< Rs1 then the result of the alu is positive then
overflow is 0
        if (OpCode1==6'b001001)         begin

                if(overflow == 1'b0 ) begin
```

```
                    // the branch is taken and the two numbers are equal
                    branchTargetAddress = PC + extended_immediate16;
                    current_state = 4;
                    PC=      branchTargetAddress;

                end

            else if (overflow == 1'b1 ) begin

                // Branch doesnt taken
                PC=PC+1;
                current_state = 4;

                end

            end /// the end of BLT


end // the end of ALU stage

else if (current_state == 3) begin

    // start with load instruction and LW.POI
    if(OpCode1== 6'b00101 || OpCode1== 6'b000110 )begin

    if (Mrd && !MWr) begin

        data_memory[8]=32'h00000002;
        data_out= data_memory[outputOfALU];
```

```
end
end        // the end of load instruction

// check if the instruction is store
if (OpCode1== 6'b000111) begin
            if (!Mrd && MWr) begin

                    data_memory[outputOfALU]= Rd;
                    end

end // the end of Store instruction

// here the RET instruction will be a little similar to load instruction == address from the memory
if (OpCode1== 6'b001110) begin

          if (Mrd && !MWr)  begin
          PC=data_memory[SP]; // next PC is the top of the stack
          SP=SP-1; // decrement the value of SP

          end
end // the end of RET instruction

// here we will check for the instruction CALL

           if (OpCode1== 6'b001101) begin
            if (!Mrd && MWr) begin

                    SP=SP+1;
                    data_memory[SP]= PC+1;
                    end

    end // the end of CALL instruction



    // Check if the instruction is PUSH

        if (OpCode1 == 6'b001111) begin

            if (!Mrd && MWr) begin

             SP=SP+1;
             data_memory[SP]= Rd;

                end

        end // the end of PUSH

        // Check if the instruction is POP
            if (OpCode1==6'b010000) begin

                Rd=data_memory[SP];
                SP=SP-1;

            end      // THE END OF POP INSTRUCTION


end // the end of MEMORY STAGE
```

```
else if (current_state == 4) begin


            if (Wbdata == 1'b0) begin

                        toRegFile= outputOfALU;


            end

else if (Wbdata == 1'b1 ) begin

                         toRegFile= data_out;

end


if (Wport1 ==1'b1) begin

            // we will write back the result on the register file

            RegisterFile [RdAddress]=  toRegFile;

            end

else if (Wport2 == 1'b1 ) begin  // so we are in the LW.POI instruction


              RegisterFile [Rs1Address]= Rs1+1;


end


PC=PC+1;  // take the next instruction


end // the end of the write backStage


            endmodule
```

7.2. The Test Bench For The System

```
module TestBench ();


reg clk;

reg reset;


          ///Instruction Decode Part

          wire [31:0] Rs1; // (Bus A )Output data from register 1

   wire [31:0] Rs2; // (Bus B) Output data from register 2

   wire [31:0] Rd; // Rd


          // control signals


          wire [2:0]PcCont;

          wire [2:0]Alusrc ;

          wire [2:0]AluOp1;

          wire Mrd ;

   wire MWr;

          wire Wport1;

          wire Wport2;

   wire Wbdata ;

          wire ExtOp;

          wire SPorAlu2;

          wire AluSpOp;

          wire DataIn;

          wire MemAdd;


          wire [31:0] outputOfALU;
```

```
        wire [31:0]extended_immediate16;

        wire [31:0] mux_2Output;

  wire [31:0] toRegFile; // the data will be stored in the register file

        wire [31:0] PCmuxOut;

        wire [31:0] data_out;


        multiCycleProcessor  MCP (clk, reset, PcCont, ExtOp, Wport1, Wport2, Alusrc,AluOp1,MemAdd,
DataIn, Mrd, MWr, Wbdata, SPorAlu2,

        AluSpOp, Rs1,Rs2,Rd,outputOfALU, extended_immediate16);


        initial begin

        current_state = 0;

        clk = 0;

        reset = 1;

        #1ns reset = 0;

        end


        always #2ns clk = ~clk;


  initial #200ns $finish;



endmodule
```