BERZIET UNIVERSITY

Faculty of Engineering & Technology – Electrical & Computer

Department of Electrical and Computer Engineering

Second Semester, 2023/2024

Project2 Report

Abdelrhman abed 1193191

Ody shbayeh 1201462

Yousef hatem 1200252

Abstract:

The objective of this project is to develop and evaluate a 16-bit pipelined RISC processor using Verilog. The processor incorporates a 16-bit program counter (PC), eight 16-bit general-purpose registers (R0-R7), and utilizes conditional execution for all instructions. R0 is hardwired to zero, and any attempt to write to it will be discarded. The processor supports four instruction types: R-type, I-type, J-type, and S-type. It features separate data and instruction memories, byte addressable memory with little-endian byte ordering, and an ALU that generates condition signals (zero, carry, overflow) to calculate the branch outcome

Contents

Table of Figures

1. Designing the Datapath and Control Signals

The data path was built as shown in the figure below:

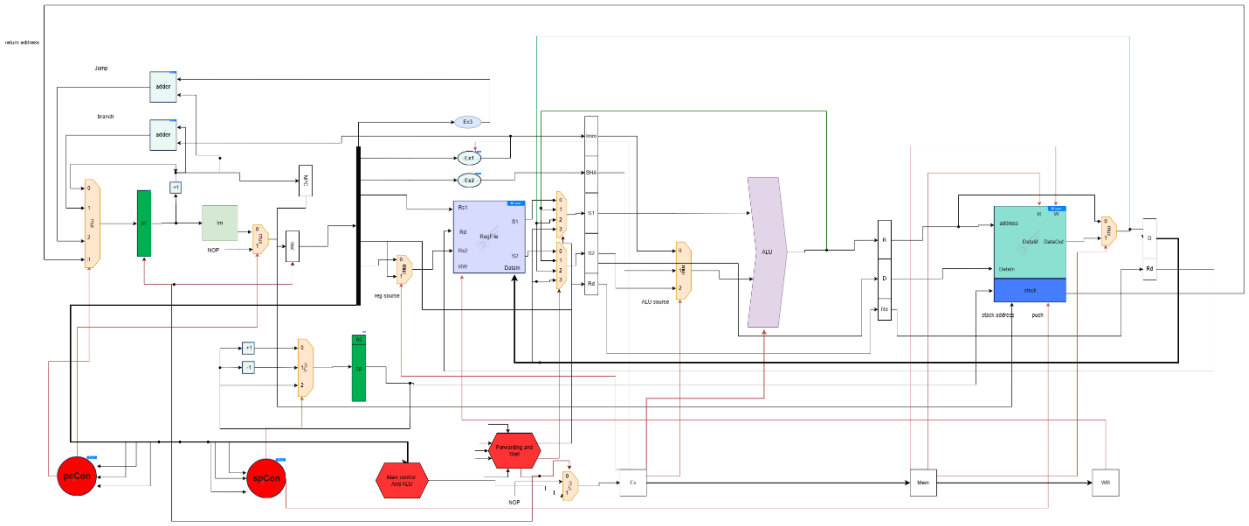

Figure 1: Datapath

Design and Implementation

Our design consists of four main blocks:

- Arithmetic and Logical Unit (ALU)
- Control Unit
- Register File
- Memory

Each one of these components will be used in the full Datapath along with the parts that are needed to enable pipelining and its hazard detection unit, forwarding unit, and the buffers between the stages. The design will include five stages: Fetching, Decoding, ALU, Memory and the Write Back stage.

1.1. Fetch

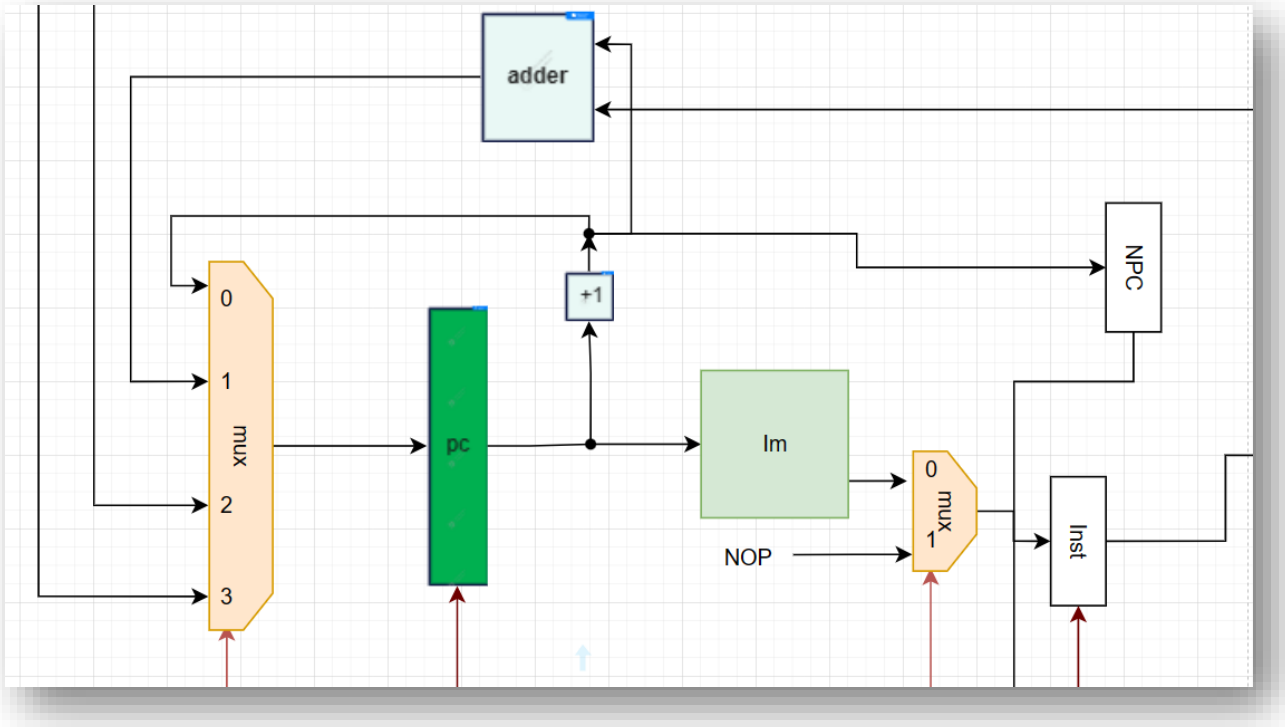

Figure 2: fetch

During the Fetch stage, the CPU relies on the PC Register to hold the memory address of the next instruction to fetch. The PC Register receives the address from the Memory code (IM) as input. A multiplexer (MUX) before the PC determines the source of the next address: 0 for the subsequent instruction, 1 for a branch instruction, 2 for a jump instruction, and 3 for a return address. Another MUX after the Instruction Memory (IM) allows the instruction to be accessed if the select value is zero, or performs a no-operation (NOP) if the value is 1. Additionally, two buffers are visible in the figure: the first buffer, Next PC (NPC), is connected to the memory stage's Stack to store the return

address, while the second buffer, Instruction (Inst), stores the fetched instruction.

Overall, the Fetch stage involves the PC Register, MUXes for determining the next address source and accessing the instruction, and buffers for storing the next PC and fetched instruction. These components work together to retrieve the instruction from memory and prepare for subsequent stages of the instruction execution cycle.

1.2. Decode stage:

We have two parts in this stage, the first one for Register file and the second for stack pointer

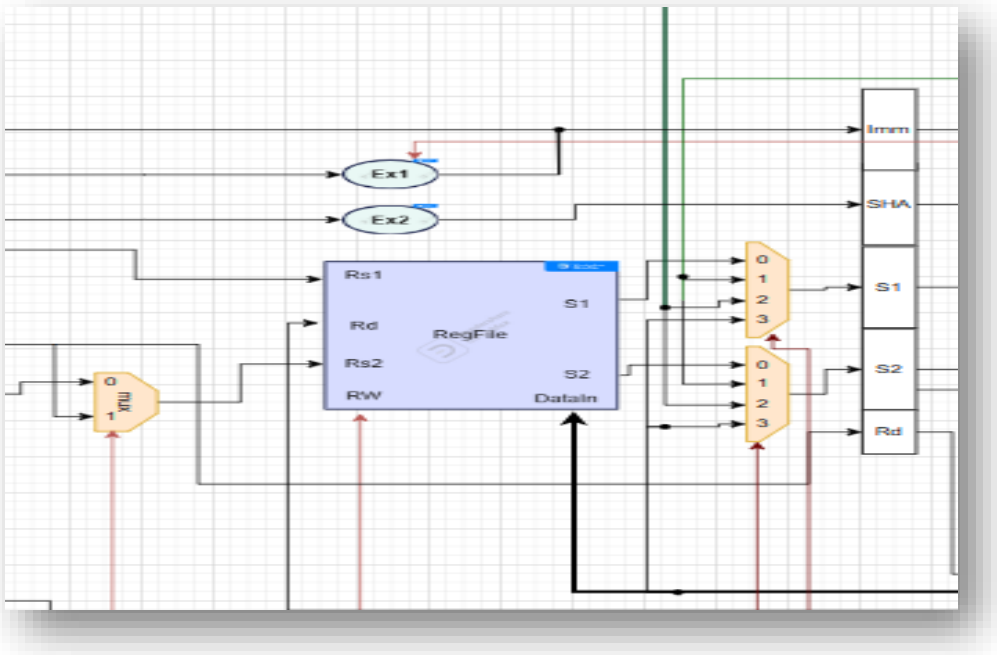

Figure 3: RegFile

The figure illustrates the Register File component, which consists of various registers. It includes Rs1 and Rs2 as the first and second source

registers, respectively. Rd serves as the destination register, while RW acts as the read-write flag. DataIn represents the input data for writing into the register. The outputs S1 and S2 are also part of the Register File.

Additionally, there are three multiplexers (MUXes) depicted in the figure. The first MUX on the left selects Rs if the select value is zero, and Rd if the value is 1. The second and third MUXes operate as follows: 0 corresponds to the output of the Register File from the decode stage, 1 corresponds to the output of the Arithmetic Logic Unit (ALU) from the execution stage, 2 corresponds to the output of the Memory block from the memory stage, and 3 corresponds to the result of the write back stage.

Overall, the Register File and MUXes play crucial roles in managing register operations, data flow, and selection of appropriate sources and outputs at different stages of the instruction execution cycle.

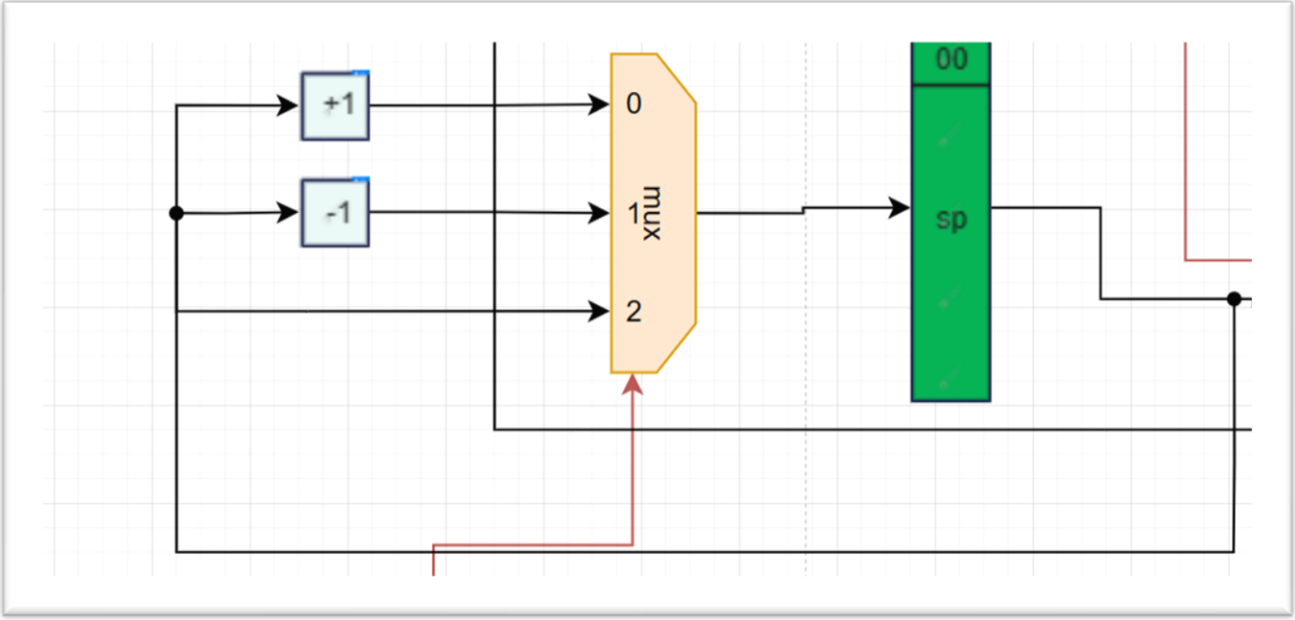

Figure 4: Stack pointer

The figure shows the presence of the Stack Pointer (SP) as the second part. The output of the SP is connected to the stack in the Memory component. A multiplexer (MUX) controls the behavior of the SP. When the select value of the MUX is 0, it triggers a "PUSH" operation, indicating that data is being added to the stack. Conversely, when the select value is 1, it triggers a "POP" operation, indicating that data is being removed from the stack. If the select value is 2, it implies that there is no change to the SP.

The SP plays a crucial role in managing the stack in memory, which is typically used for storing temporary data and managing function calls and returns. The MUX associated with the SP controls whether data is added to or removed from the stack, or if there is no change to the SP at that particular stage of the instruction execution cycle.

1.3. execute stage

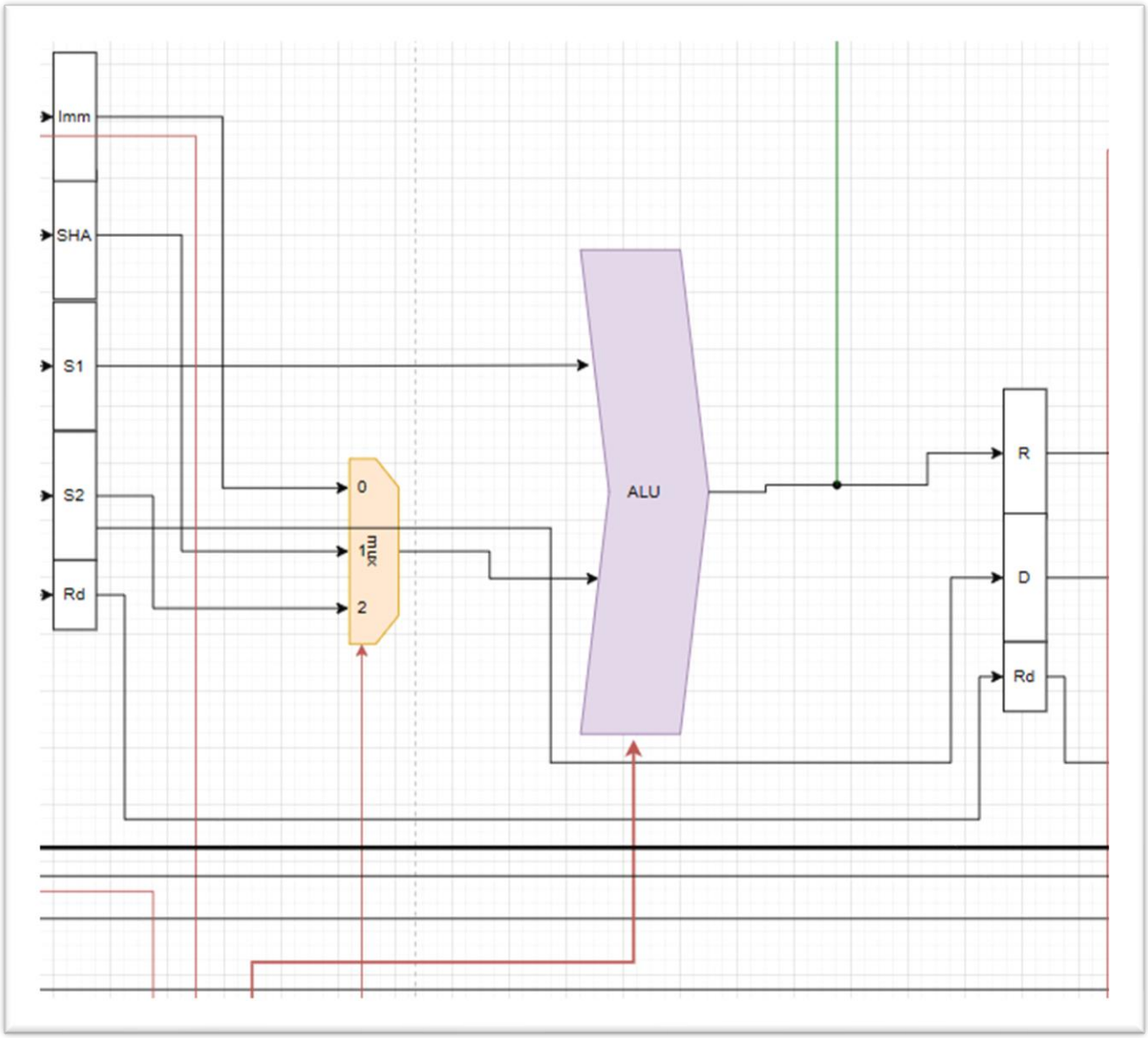

Figure 5: Execute stage

The figure highlights the primary component in this stage, which is the Arithmetic Logic Unit (ALU). The ALU has two inputs: the first input comes from the source register S1, while the second input comes from the multiplexer (MUX).

The MUX is responsible for selecting the appropriate input for the ALU. It operates as follows: when the select value is 0, the immediate value is chosen as the input. If the select value is 1, the shift immediate amount (SHA) is selected as the input. Finally, when the select value is 2, the second source (S2) is chosen as the input for the ALU.

The ALU performs arithmetic and logical operations on the inputs it receives, generating an output that is utilized in subsequent stages of the instruction execution cycle. The MUX allows for flexibility in selecting the appropriate input based on the specific operation being performed by the ALU.

1.4. Memory Stage and Write Back Stage

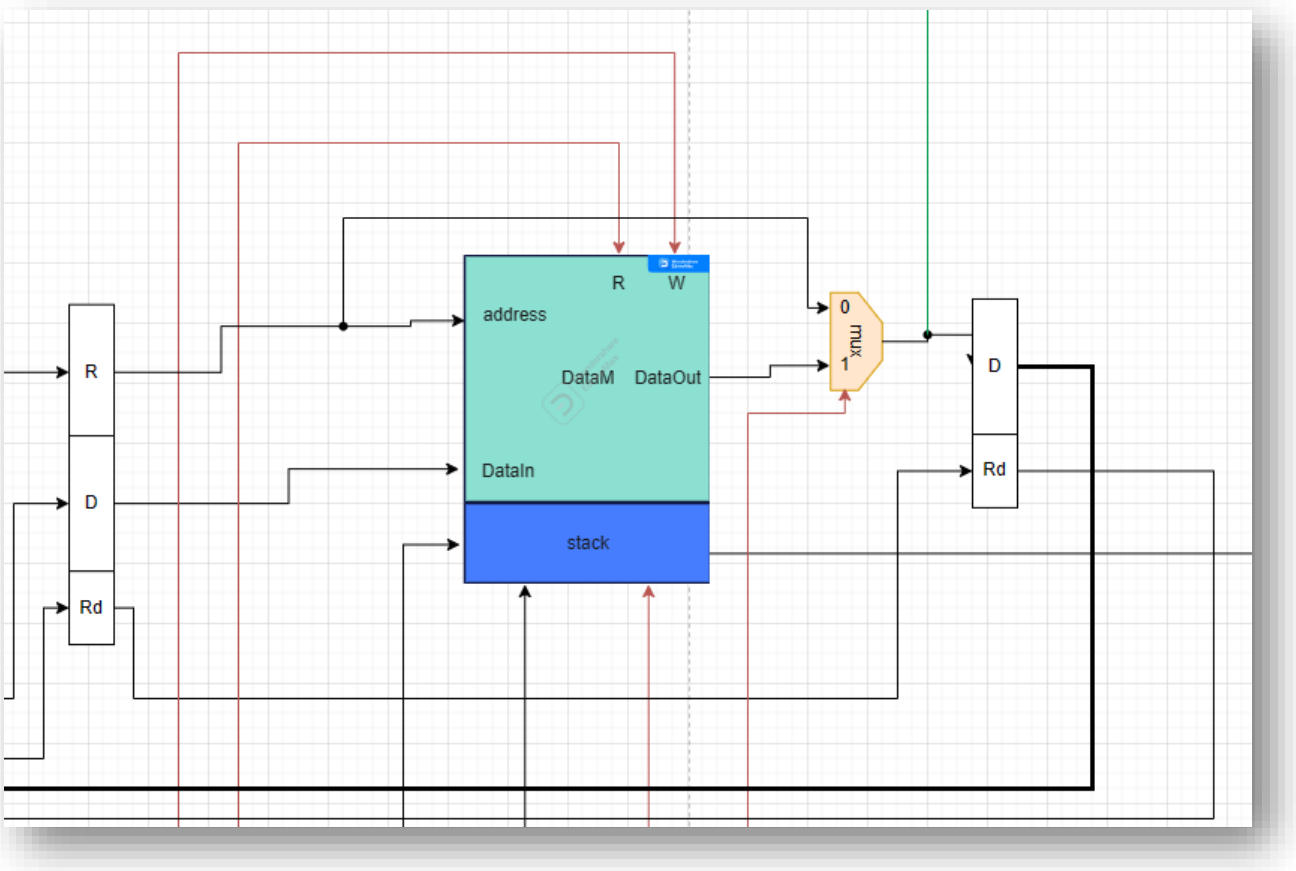

Figure 6: Memory Stage and Write Back Stage

The main component depicted is the Memory data block, which also encompasses the Stack. The Memory data block has four inputs: address and Datain, each consisting of 32 bits, and R and W which function as flags for read and write operations, respectively. It has one output. On the other hand, the Stack has three inputs: one connected to the NPC (Next PC) buffer to retrieve the return address, the second one from the Stack Pointer (SP), and the last one acts as a flag connected to the Stack Pointer condition (SpCon). The Stack also has one output.

The multiplexer (MUX) situated next to the memory block performs a selection based on the following: 0 indicates the output from the memory data, while 1 signifies the result obtained from the execution stage.

Furthermore, there are two additional buffers present. The first buffer, labeled as D, corresponds to the write-back stage and contains the data to be written onto the register. The second buffer, denoted as Rd, represents the address of the destination register where the write operation occurs.

2. Building the State Machine Diagram with Control signals generation

2.1 Control signals generation

2.1.1 main control

No.	Instr	Format	Meaning	Opcode Value	m
1	AND	R-Type	Reg(Rd) = Reg(Rs1) & Reg(Rs2)	0000	-
2	ADD	R-Type	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	0001	-
3	SUB	R-Type	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	0010	-
4	ADDI	I-Type	Reg(Rd) = Reg(Rs1) + Imm	0011	-
5	ANDI	I-Type	Reg(Rd) = Reg(Rs1) & Imm	0100	-
6	LW	I-Type	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0101	-
7	LBu	I-Type	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0110	0
8	LBs	I-Type	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0110	1
9	SW	I-Type	Mem(Reg(Rs1) + Imm) = Reg(Rd)	0111	-
10	BGT	I-Type	if (Reg(Rd) > Reg(Rs1)) PC += Imm	1000	0
11	BGTZ	I-Type	if (Reg(Rd) > Reg(R0)) PC += Imm	1000	1
12	BLT	I-Type	if (Reg(Rd) < Reg(Rs1)) PC += Imm	1001	0
13	BLTZ	I-Type	if (Reg(Rd) < Reg(R0)) PC += Imm	1001	1
14	BEQ	I-Type	if (Reg(Rd) == Reg(Rs1)) PC += Imm	1010	0
15	BEQZ	I-Type	if (Reg(Rd) == Reg(R0)) PC += Imm	1010	1
16	BNE	I-Type	if (Reg(Rd) != Reg(Rs1)) PC += Imm	1011	0
17	BNEZ	I-Type	if (Reg(Rd) != Reg(R0)) PC += Imm	1011	1
18	JMP	J-Type	Next PC = {PC[15:10], Immediate}	1100	-
19	CALL	J-Type	Next PC = {PC[15:10], Immediate}, PC + 4 saved on r15	1101	-
20	RET	J-Type	Next PC = r7	1110	-
21	Sv	S-Type	M[rs] = imm	1111	-

Truth table of Main Control

2.1.2 ALU table

ALUOp	Function Code	Operation
00	-	ADD
01	-	SUB
10	000	AND
10	001	OR
10	010	ADD
10	011	SUB
10	100	SLT
11	-	LW/SW

ALU table

2.1.3 SP control and PC control tables

Instruction	SPControl	Push
JAL	0	1
S-type	1	0
Default	2	0

SP control table

Instruction	PCSource
J	2
Branch	1
S-type	3
Default	0

PC control table

2.1.4 Forwarding table

Condition	ForwardA	ForwardB
EX/MEM.RegWrite and EX/MEM.Rd = ID/EX.Rs	10	-
MEM/WB.RegWrite and MEM/WB.Rd = ID/EX.Rs	01	-
EX/MEM.RegWrite and EX/MEM.Rd = ID/EX.Rt	-	10
MEM/WB.RegWrite and MEM/WB.Rd = ID/EX.Rt	-	01
Otherwise	00	00

2.1.5 Control hazard code

```
always @(*) begin
    if ((BEQ && Zero)) begin
        Jmp = 0;
        Br = 1;
        Stop = 0;
        Kill1 = 1;
    end else if (J) begin
        Jmp = 1;
        Br = 0;
        Stop = 0;
        Kill1 = 1;
    end else if (Stop) begin
        Jmp = 0;
        Br = 0;
        Stop = 1;
        Kill1 = 1;
    end else begin
        Jmp = 0;
        Br = 0;
        Stop = 0;
        Kill1 = 0;
    end
end

```

2.2 state diagram

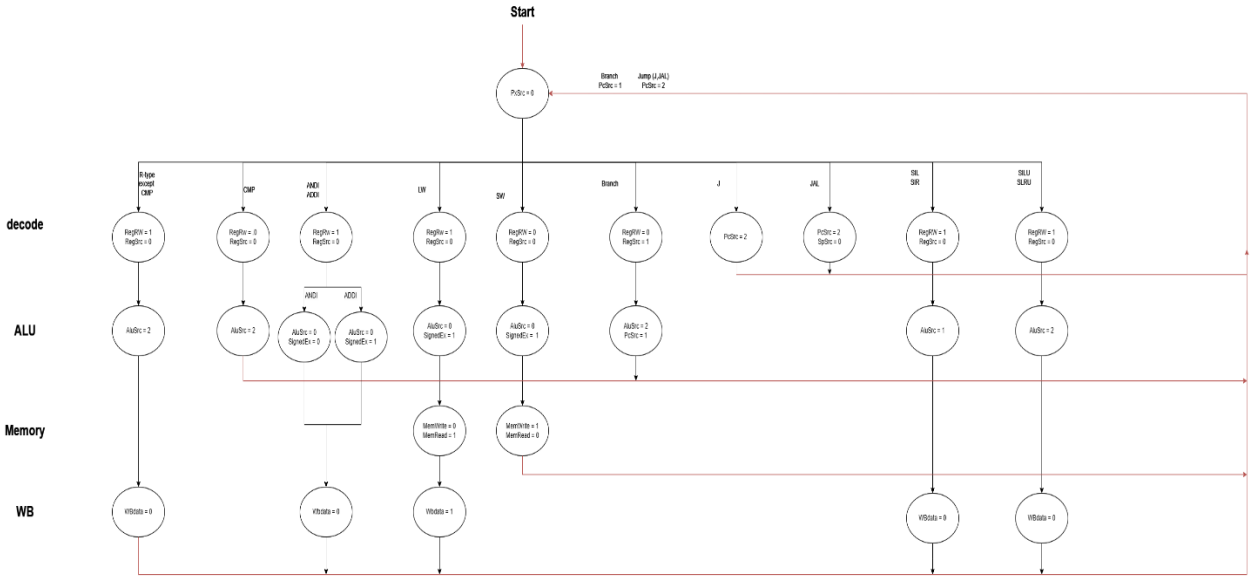

Figure 7: state machine diagram

A systematic approach was followed to construct the state machine diagram for the stages of the Pipelined processor. The process involved several steps.

Firstly, the operations to be executed in each stage and the corresponding conditions were thoroughly examined. This analysis helped identify the specific states required for the state machine diagram.

Next, the transitions between these states were defined. These transitions were triggered by control signals generated by the processor. For instance, the transition from the instruction fetch stage to the instruction decode stage occurred when the "state number" value indicated that the instruction had been fetched from memory and was ready for decoding.

Moreover, the actions to be performed during each state were defined. Examples of such actions include incrementing the program counter (PC) or writing to the register file.

Lastly, the accuracy of the state machine diagram was ensured by testing the processor's operation and comparing the results with the expected behavior. This verification step confirmed that the constructed diagram correctly represented the functioning of the Pipelined processor.

3. Testing the Modules

3.1 First Test

We have a code that we use to test our machine. The code helps us to ensure that the machine is working properly and that it is meeting our expectations.

```
reg [15:0] Mem [0:6] = '{

16'b0000000000000000, // INITIAL INSTRUCTION TO GET READY

16'b0001000000001100, // LW R5, R0(LL)

16'b0000000101001110, // SLS R6, R5

16'b0000100111001000, // ADD R7, R7, R6

16'b0001100101011110, // SLRV R8, R5, R6

16'b0001100000001100, // SW R8, R0(LL)

16'b0001000010010000  // LW R9, R0(LL)

};
```

```
Fetch   decode   ALU   Memory  WB

C1      LW

C2      SLS     LW

C3      ADD     SLS    LW

C4      ADD     SLS    ___    LW

C5      SLRV    ADD    SLS    ___    LW

C6      SW      SLRV   ADD    SLS    ___

C7      LW      SW     SLRV   ADD    SLS

C8          LW     SW     SLRV   ADD

C9              LW     SW     SLRV

C10                 LW     SW

C11                    LW
```

Cycle Descriptions for the first test

Cycle 1

- **Fetch**: LW (16'b0001000000001100) is fetched into the Fetch stage.
- **Decode**: -
- **ALU**: -
- **Memory**: -
- **Write Back**: -

Cycle 2

- **Fetch**: SLS (16'b0000000101001110) is fetched into the Fetch stage.
- **Decode**: LW is decoded.
- **ALU**: -
- **Memory**: -
- **Write Back**: -

Cycle 3

- **Fetch**: ADD (16'b0000100111001000) is fetched into the Fetch stage.
- **Decode**: SLS is decoded.
- **ALU**: LW is in the ALU stage.
- **Memory**: -
- **Write Back**: -

Cycle 4

- **Fetch**: ADD (16'b0000100111001000) is in the Decode stage.
- **Decode**: SLS is in the ALU stage.
- **ALU**: LW is in the Memory stage.
- **Memory**: -
- **Write Back**: -

Cycle 5

- **Fetch**: SLRV (16'b0001100101011110) is fetched into the Fetch stage.
- **Decode**: ADD is in the Decode stage.
- **ALU**: SLS is in the ALU stage.
- **Memory**: LW is in the Write Back stage.
- **Write Back**: -

Cycle 6

- **Fetch**: SW (16'b0001100000001100) is fetched into the Fetch stage.
- **Decode**: SLRV is in the Decode stage.
- **ALU**: ADD is in the ALU stage.

- **Memory**: SLS is in the Write Back stage.
- **Write Back**: LW is in the Write Back stage.

Cycle 7

- **Fetch**: LW (16'b0001000010010000) is fetched into the Fetch stage.
- **Decode**: SW is in the Decode stage.
- **ALU**: SLRV is in the ALU stage.
- **Memory**: ADD is in the Write Back stage.
- **Write Back**: SLS is in the Write Back stage.

Cycle 8

- **Fetch**: -
- **Decode**: LW is in the Decode stage.
- **ALU**: SW is in the ALU stage.
- **Memory**: SLRV is in the Memory stage.
- **Write Back**: ADD is in the Write Back stage.

Cycle 9

- **Fetch**: -
- **Decode**: -
- **ALU**: LW is in the ALU stage.
- **Memory**: SW is in the Memory stage.
- **Write Back**: SLRV is in the Write Back stage.

Cycle 10

- **Fetch**: -
- **Decode**: -
- **ALU**: -
- **Memory**: LW is in the Memory stage.
- **Write Back**: SW is in the Write Back stage.

Cycle 11

- **Fetch**: -
- **Decode**: -
- **ALU**: -
- **Memory**: -
- **Write Back**: LW is in the Write Back stage.

3.2 Second Test

We have a code that we use to test our machine. The code helps us to ensure that the machine is working properly and that it is meeting our expectations.

```
reg [15:0] Mem [0:9] = '{
16'b0000000000000000, // INITIAL INSTRUCTION TO GET READY
16'b0101000000001100, // LW R2, 0(LL)
16'b1101000000110010, // JAL AA
16'b0101000000011100, // LW R3, 0(LL)
16'b0010100001000000, // ADD R1, R2, R3
16'b1111111111111110, // J for ever
16'b0110000001010100, // ADDI R1, R1, 20
16'b1000000100000000, // CMP R1, R2
16'b1010000100110010, // BEQ R1, R2, 0(AA)
16'b0011000010000001  // SUB R1, R1, R2 -> STOP = 1
};
```

Fetch	Decode	ALU	Memory	WB
C1	LW			
C2	JAL	LW		
C3	LW	JAL	LW	
C4	ADD	LW	JAL	LW
C5	ADD	LW	___	JAL	LW
C6	JMP	ADD	LW	___	JAL
C7	ADDI	JMP	ADD	LW	___	JAL
C8	CMP	ADDI	JMP	ADD	LW	___
C9	BEQ	CMP	ADDI	JMP	ADD	LW
C10	SUB	BEQ	CMP	ADDI	JMP	ADD

Cycle Descriptions

Cycle 1

- **Fetch**: LW (16'b0101000000001100) is fetched into the Fetch stage.
- **Decode**: -
- **ALU**: -
- **Memory**: -
- **Write Back**: -

Cycle 2

- **Fetch**: JAL (16'b1101000000110010) is fetched into the Fetch stage.
- **Decode**: LW is decoded.
- **ALU**: -
- **Memory**: -
- **Write Back**: -

Cycle 3

- **Fetch**: LW (16'b0101000000011100) is fetched into the Fetch stage.
- **Decode**: JAL is decoded.
- **ALU**: LW is in the ALU stage.
- **Memory**: -
- **Write Back**: -

Cycle 4

- **Fetch**: ADD (16'b0010100001000000) is fetched into the Fetch stage.
- **Decode**: LW is decoded.
- **ALU**: JAL is in the ALU stage.
- **Memory**: LW is in the Memory stage.
- **Write Back**: -

Cycle 5

- **Fetch**: ADD (16'b0010100001000000) is in the Decode stage.
- **Decode**: LW is in the Memory stage.
- **ALU**: JAL is in the Write Back stage.
- **Memory**: -
- **Write Back**: LW is in the Write Back stage.

Cycle 6

- **Fetch**: JMP (16'b1111111111111110) is fetched into the Fetch stage.
- **Decode**: ADD is in the Decode stage.
- **ALU**: LW is in the Write Back stage.

- **Memory**: JAL is in the Write Back stage.
- **Write Back**: -

Cycle 7

- **Fetch**: ADDI (16'b0110000001010100) is fetched into the Fetch stage.
- **Decode**: JMP is in the Decode stage.
- **ALU**: ADD is in the ALU stage.
- **Memory**: LW is in the Write Back stage.
- **Write Back**: JAL is in the Write Back stage.

Cycle 8

- **Fetch**: CMP (16'b1000000100000000) is fetched into the Fetch stage.
- **Decode**: ADDI is in the Decode stage.
- **ALU**: JMP is in the ALU stage.
- **Memory**: ADD is in the Write Back stage.
- **Write Back**: LW is in the Write Back stage.

Cycle 9

- **Fetch**: BEQ (16'b1010000100110010) is fetched into the Fetch stage.
- **Decode**: CMP is in the Decode stage.
- **ALU**: ADDI is in the ALU stage.
- **Memory**: JMP is in the Memory stage.
- **Write Back**: ADD is in the Write Back stage.

Cycle 10

- **Fetch**: SUB (16'b0011000010000001) is fetched into the Fetch stage.
- **Decode**: BEQ is in the Decode stage.
- **ALU**: CMP is in the ALU stage.
- **Memory**: ADDI is in the Memory stage.
- **Write Back**: JMP is in the Write Back stage.

4. Conclusion

In conclusion, an efficient MIPS CPU was constructed using the pipelined approach. Throughout the project, a more in-depth understanding of how the MIPS CPU functions with the pipelined approach was obtained while designing the various components. The pipelined approach was chosen due to its efficiency benefits compared to the single-cycle approach, as it allows for concurrent execution of multiple instructions by dividing them into separate pipeline stages. Moreover, valuable insights were gained on connecting the modules using code and performing thorough testing. Looking ahead, future work could focus on optimizing the final module using the pipelined approach to reduce the cycles per instruction (CPI) in the CPU.