Faculty Of Engineering and Technology

Electrical And Computer Engineering Department

Computer Architecture (ENCS4370)

Project 2

Multicycle MIPS RISC Processor

Prepared by:

Nicola Abu Shaibeh   ID:1190843

Contents

1. DESIGN AND IMPLEMENTATION

Our approach in this project was a multi-cycle processor. Multi-cycle processors divided the data path into 5 cycles, namely FETCH, DECODE, EXECUTE, MEMORY, and WRITEBACK. Each processor cycle is executed in a single clock cycle. The project's processor design is based on the MIPS Reduced Instruction Set Computer (RISC) architecture and includes a subset of the MIPS Instruction set. MIPS Instructions are always 32-bits wide and use one of the four following instruction types R-Type, J-Type, I-type, and S-type.

MULTI-CYCLE STAGES

1. *Instruction Fetch Stage:* During the instruction fetch stage, the processor retrieves the next instruction from memory. It involves fetching the instruction from the memory location pointed to by the program counter (PC). The PC is incremented to point to the next instruction to be fetched. The fetched instruction is stored in a dedicated register

2. *Instruction Decode Stage:* In the instruction decode stage, the fetched instruction is decoded to determine the type of instruction and the operands involved. The control signals necessary for executing the instruction are generated based on the instruction type and function. The source registers (Rs1 and Rs2) and the destination register (Rd) are identified, and the necessary data paths are set up to facilitate the subsequent execution of the instruction.

3. *Execution Stage:* In the execution stage, the actual computation or operation specified by the instruction takes place. The operands required for the operation are fetched from the register file or memory, and the appropriate arithmetic or logical operation is performed. This stage involves ALU operations, such as addition, subtraction, AND, OR, etc., based on the instruction type and function.

4. *Memory Access Stage:* During the memory access stage, memory operations are performed if the instruction involves accessing or modifying data in memory. This stage is primarily used for load (LW) and store (SW) instructions. In the case of a load instruction, data is read from memory and stored in a register. For a store instruction, data from a register is written into memory at a specified memory address.

5. *Writeback Stage:* The writeback stage is the final stage of the instruction execution process. It involves writing the results of the computation back to the destination register or memory, depending on the instruction type. If the instruction is a register-to-register operation, the result is written back to the destination register. In the case of a memory operation, the writeback stage ensures that the data is correctly stored in memory.

RTL DESCRIPTION

R-TYPE (REGISTER TYPE) :

Function[31:27]	R_{s1}[26:22]	Rd[21:17]	R_{s2}[16:12]	Unused[11:3]	Type[2:1]	Stop[0]
5 bits	5 bits	5 bits	5 bits	9 bits	2 bits	1 bit

- 5-bit function, to determine the specific operation of the instruction
- 5-bit Rs1: first source register
- 5-bit Rd: destination register
- 5-bit Rs2: second source register
- 9-bit unused
- 2-bit instruction type
- Stop bit

Function	RTL				
	FETCH	**DECODE**	**EXECUTE**	**MEMORY**	**WRITEBACK**
AND(00000)	PC ← PC + 1	A = Rs1[26:22] B = Rs2[16:12]	Rd ← Rs1 AND Rs2	-	Rd = ALU Out
ADD(0001)	PC ← PC + 1	A = Rs1[26:22] B = Rs2[16:12]	Rd ← Rs1 + Rs2	-	Rd = ALU Out
SUB(00010)	PC ← PC + 1	A = Rs1[26:22] B = Rs2[16:12]	Rd ← Rs1 - Rs2	-	Rd = ALU Out
CMP(00011)	PC ← PC + 1	A = Rs1[26:22] B = Rs2[16:12]	Rd ← Rs1 - Rs2 Set zero-signal result	-	-

Table 1 R-Type RTL

I-TYPE (IMMEDIATE TYPE):

Function[31:27]	R_{s1}[26:22]	Rd[21:17]	Immediate14[16:3]	Type[2:1]	Stop[0]
5 bits	5 bits	5 bits	14 bits	2 bits	1 bit

- 5-bit function, to determine the specific operation of the instruction
- 5-bit Rs1: first source register
- 5-bit Rd: destination register
- 14-bit immediate: unsigned for logic instructions and signed otherwise.
- 2-bit instruction type
- Stop bit

Function	RTL				
	FETCH	**DECODE**	**EXECUTE**	**MEMORY**	**WRITEBACK**
ANDI(00000)	PC ← PC + 1	A = Rs1[26:22] B = Imm[16:3]	Rd ← Rs1 AND Rs2	-	Rd = ALU Out
ADDI(0001)	PC ← PC + 1	A = Rs1[26:22] B = Imm[16:3]	Rd ← Rs1 + Imm	-	Rd = ALU Out
LW(00010)	PC ← PC + 1	A = Rs1[26:22] B = Imm[16:3]	Rd ← Rs1 + Imm	Rd ← M[Rs1 + Imm]	Rd = ALU Out
SW(00011)	PC ← PC + 1	A = Rs1[26:22] B = Imm[16:3]	Rd ← Rs1 + Imm	M[Rs1 + Imm] → Rd	-
BEQ(00100)	PC ← PC + Imm[14:0]	A = Rs1[26:22] B = Rd[21:17]	Compare Rs1 and Rd for	-	-

Table 2 I-Type RTL

J-TYPE (JUMP TYPE) :

Function[31:27]	Immediate24[26:3]	Type[2:1]	Stop[0]
5 bits	24 bits	2 bits	1 bit

- 5-bit function, to determine the specific operation of the instruction
- 24-bit signed immediate: jump offset
- 2-bit instruction type
- Stop bit

Function	RTL				
	FETCH	DECODE	EXECUTE	MEMORY	WRITEBACK
J(00000)	PC ← PC + Immediate24[26:3]	PC + Imm24[26:3]	-	-	-
JAL(0001)	PC ← PC + Immediate24[26:3]	PC + Imm24[26:3] Stack.Push (PC + 4)	-	-	-

Table 3 J-Type RTL

S-TYPE (SHIFT TYPE) :

Function[31:27]	R_{s1}[26:22]	Rd[21:17]	R_{s2}[16:12]	SA[11:7]	Unused[6:3]	Type[2:1]	Stop[0]
5 bits	5 bits	5 bits	5 bits	5 bits	4 bits	2 bits	1 bit

- 5-bit function, to determine the specific operation of the instruction
- 5-bit Rs1: first source register
- 5-bit Rd: destination register
- 5-bit Rs2: second source register
- 5-bit SA: the constant shift amount.
- 4-bit unused
- 2-bit instruction type
- Stop bit

Function	RTL				
	FETCH	DECODE	EXECUTE	MEMORY	WRITEBACK
SLL (00000)	PC ← PC + 1	A = Rs1[26:22] B = SA[6:2]	Rd = Rs1 << SA	-	Rd = ALU Out
SLR (0001)	PC ← PC + 1	A = Rs1[26:22] B = SA[6:2]	Rd = Rs1 >> SA	-	Rd = ALU Out
SLLV(00010)	PC ← PC + 1	A = Rs1[26:22] B = Rs2[16:12]	Rd = Rs1 << Rs2	-	Rd = ALU Out
SLRV(00011)	PC ← PC + 1	A = Rs1[26:22] B = Rs2[16:12]	Rd = Rs1 >> Rs2	-	Rd = ALU Out

Table 4 S-Type RTL

CIRCUIT DESIGN AND THE OVERALL DATA PATH.

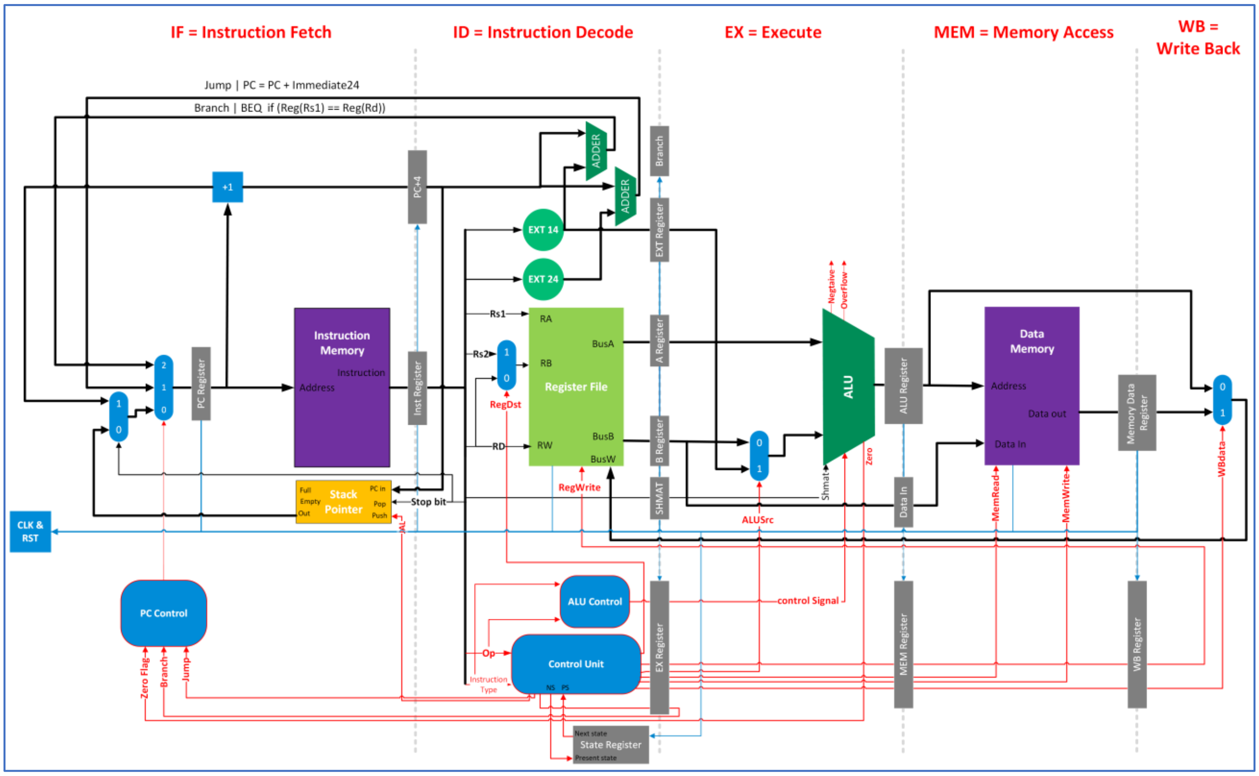


Figure 1 MIPS processor design

As shown in figure 1 the MIPS multicycle processor design was adapted to match the RTL (Register Transfer Level), instruction types, and functions involve several key considerations that are being addressed. Firstly, the multicycle processor design is modified to incorporate the specific instruction types mentioned in the RTL description, namely R-Type, I-Type, J-Type, and S-Type instructions. Each instruction type requires a different control path and data path to handle the specific operations and data transfers associated with it.

The control unit of the multicycle processor was expanded to include control signals for each instruction type and function. This entails adding control signals to enable the proper sequencing and coordination of the different stages of instruction execution. This includes adding the necessary multiplexers, arithmetic units, and memory access units to handle the specific data operations specified in the RTL.

COMPONENTS USED TO BUILD THE PROCESSOR:

- Adder: 1bit adder, 32-bit adder
- Arithmetic logical unit: 32-bit ALU
- Data Memory
- Register File

- Instruction Memory
- Mux: 2-bit mux, 5-bit mux 2x1, 32-bit mux 2x1, 32-bit mux 3x1
- Stack Pointer
- Sign Extend: 14 bits sign Extend and 24 bits
- Registers: 1bit register, 4bit register, 5bit register, 32-bit register
- State Control Unit
- ALU control Unit
- PC Control
- Decoder
- State Register

CONTROL UNITS AND CONTROL SIGNALS

MAIN CONTROL UNIT

The control unit generates various control signals based on the input opcode and instruction type. These control signals determine the behavior of the processor during different stages of instruction execution.

The control signals produced by the control unit include:

- **RegDst:** This signal determines whether the destination register should be selected from Rs2 or Rd based on the instruction type.
- ALUSrc: This signal selects the second operand for the ALU, either from Rs2 or an immediate value based on the instruction type.
- **WBdata:** This signal indicates whether the ALU output or memory data should be written back to the register file.
- **RegWrite:** This signal enables the write operation to the register file.
- **MemRead:** This signal enables the read operation from memory.
- **MemWrite:** This signal enables the write operation to memory.
- **Branch:** This signal indicates whether a branch should be taken based on the comparison result.
- **Jump:** This signal indicates whether a jump instruction is being executed.
- **JumpJAL:** This signal is specific to the JAL instruction and indicates whether a jump with link operation is being performed.

The control signals are dynamically determined based on the opcode and instruction type. If an unsupported opcode or instruction type is encountered, the control signals are set to "don't care" values denoted by 'x'.

Opcode (Binary)	Opcode	Instruction Type	RegDst	ALUSrc	WBdata	Reg Write	Mem Read	Mem Write	Branch	Jump	JAL
00000	AND	R-Type	1	0	0	1	0	0	0	0	0
00001	ADD	R-Type	1	0	0	1	0	0	0	0	0
00010	SUB	R-Type	1	0	0	1	0	0	0	0	0
00011	CMP	R-Type	1	0	X	1	0	0	0	0	0
00000	ANDI	I-Type	X	1	0	1	0	0	0	0	0
00001	ADDI	I-Type	X	1	0	1	0	0	0	0	0
00010	LW	I-Type	X	1	1	1	1	0	0	0	0
00011	SW	I-Type	0	1	X	0	0	1	0	0	0
00100	BEQ	I-Type	0	0	X	0	0	0	1	0	0
00000	J	J-Type	X	X	X	0	0	0	0	1	0
00001	JAL	J-Type	X	X	X	0	0	0	0	1	1
00000	SLL	S-Type	1	0	0	1	0	0	0	0	0
00001	SLR	S-Type	1	0	0	1	0	0	0	0	0
00010	SLLV	S-Type	1	0	0	1	0	0	0	0	0
00011	SLRV	S-Type	1	0	0	1	0	0	0	0	0
Default			X	X	X	X	X	X	X	X	X

Table 5 Control Unit Signals

logic equations for each control signal

- RegDst = (instructionType == 2'b00) ? 1'b1 : 1'b0;
- ALUSrc = ((instructionType == 2'b00) || (instructionType == 2'b01)) ? 1'b0 : 1'b1
- WBdata = (instructionType == 2'b01) ? 1'b0 : 1'b1;
- RegWrite = ((instructionType == 2'b00) || (instructionType==2'b01) || (instructionType== 2'b11)) ? 1'b1 : 1'b0;
- MemRead = (instructionType == 2'b01) ? 1'b0 : 1'b0
- MemWrite = (opcode == 5'b00011) ? 1'b1 : 1'b0;
- Branch = (opcode == 5'b00100) ? 1'b1 : 1'b0;
- Jump = ((instructionType == 2'b10) && (opcode == 5'b00000)) ? 1'b1 : 1'b0;
- JumpJAL = ((instructionType == 2'b10) && (opcode == 5'b00001)) ? 1'b1 : 1'b0;

ALU CONTROL UNIT

The ALU Control module generates the control signal ALU Control based on the input opcode and instruction type. The ALU Control signal determines the specific operation to be performed by the Arithmetic Logic Unit (ALU) during the execution of an instruction.

Opcode	Instruction Type	ALU Control
00000	00 (R-Type)	AND(0001)
00001	00 (R-Type)	ADD(0010)
00010	00 (R-Type)	SUB(0011)
00011	00 (R-Type)	CMP(0100)
00000	01 (I-Type)	AND(0001)
00001	01 (I-Type)	ADD(0010)
00010	01 (I-Type)	ADD(0010)
00011	01 (I-Type)	ADD(0010)
00100	01 (I-Type)	BEQ(0101)
00000	11 (S-Type)	SLL(1100)
00001	11 (S-Type)	SLR(1101)
00010	11 (S-Type)	SLLV(1110)
00011	11 (S-Type)	SLRV(1111)

Table 6 ALU Control Signals

STATE MACHINE

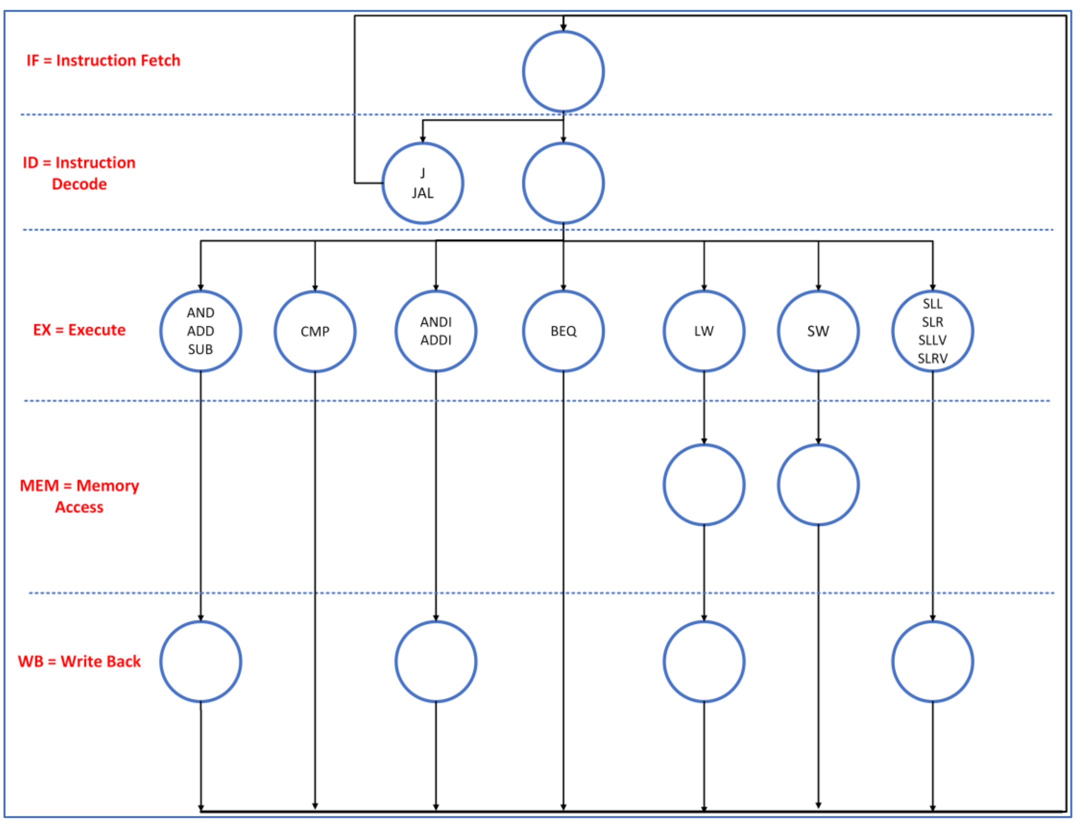


FIGURE 2 STATE DIAGRAM

Multi-cycle processors divide the data path into five stages: *Instruction Fetch Stage*, *Instruction Decode Stage*, *Execution Stage*, *Memory Access Stage*, and *Writeback Stage*. Each stage is executed in a single clock cycle. However, not every instruction requires all five stages as declared in Table (6). In some cases, only the load operation utilizes all five stages. Inefficient multi-cycle processors execute each instruction in the worst-case number of cycles, resulting in wasted time and decreased throughput. To address this, this multi-cycle processor uses the State Controller and State Register to execute instructions in the required number of cycles. The State Controller, which is a state machine, determines the next state to be executed. The output of the State Controller, Next State, is fed into the State Register. At the rising edge of the system clock, the State Register latches this value and outputs it back to the State Controller, causing a jump to the specified state. This synchronization with the system clock ensures proper timing of processor cycles and allows the State Controller to jump to any cycle, thereby increasing throughput.

Instruction	Stages	Number of Stages(# of cycles)
AND	IF→ ID→ EX→ WB	4
ADD	IF→ ID→ EX→ WB	4
SUB	IF→ ID→ EX→ WB	4
CMP	IF→ ID→ EX	3
ANDI	IF→ ID→ EX→ WB	4
ADDI	IF→ ID→ EX→ WB	4
LW	IF→ ID→ EX→ MEM→WB	5
SW	IF→ ID→ EX→MEM	4
BEQ	IF→ ID→ EX	3
J	IF→ ID	2
JAL	IF→ ID	2
SLL	IF→ ID→ EX→ WB	4
SLR	IF→ ID→ EX→ WB	4
SLLV	IF→ ID→ EX→ WB	4
SLRV	IF→ ID→ EX→ WB	4

Table 7 Stages of each Instruction

Here is an explanation of each State Controller FSM state corresponding with Figure 2:

• **State 0 (Fetch Instruction):** This state is responsible for fetching the instruction from the instruction memory. It sets up the control signals for fetching the next instruction and transitions to State 1.

• **State 1 (Decode):** In this state, the fetched instruction is decoded. The opcode and other relevant fields are examined to determine the instruction type and set up the appropriate control signals for the subsequent stages. The state transitions to the corresponding state based on the instruction type.

• **State 2 (Execute R-type):** This state is specific to R-type instructions. It performs the ALU operation specified by the instruction and computes the result. The necessary control signals for the execution of R-type instructions are set up in this state, and the state transitions to State 3 for the write back.

• **State 3 (Write Back of R-type):** After the execution of an R-type instruction, the result is written back to the register file in this state. The control signals related to register writing are activated, and the state transitions back to State 0 to fetch the next instruction.

• **State 4 (Execution of various I-type):** This state covers the execution of various I-type instructions, such as ANDI, ADDI, and others. The specific operation dictated by the instruction is performed in this state, and the control signals are set accordingly. The state transitions to State 5 for the write back.

• **State 5 (Write Back of various I-type):** After the execution of an I-type instruction, the result is written back to the register file in this state. The control signals for register writing are activated, and the state transitions back to State 0 for the next instruction.

• **State 6 (Execute of BEQ):** This state is dedicated to the execution of the BEQ (branch equal) instruction. It compares the values of two registers and determines whether a branch should be taken based on the result. The necessary control signals for the branch operation are set up, and the state transitions accordingly.

• **State 7 (Execute of CMP):** This state handles the execution of the CMP (compare) instruction. It compares the values of two registers and updates the necessary flags or status bits based on the result. Control signals for the comparison operation are activated, and the state transitions back to State 0.

• **State 8 (Memory write for SW):** This state is responsible for writing data from a register to the memory location specified by the SW (store word) instruction. The necessary control signals for memory writing are activated, and the state transitions back to State 0.

• **State 9 (Memory access for LW):** In this state, the LW (load word) instruction is executed by accessing the memory and retrieving the data from the specified memory location. The control signals for memory reading are activated, and the state transitions to State 10

• **State 10 (Write Back of LW):** After the LW instruction's memory access, the retrieved data is written back to the register file in this state. The control signals for register writing are activated, and the state transitions back to State 0.

• **State 11 (Write Back of SW):** This state is responsible for the write back process after the SW instruction. Since the SW instruction does not involve writing data to the register file, this state mainly focuses on transitioning back to State 0.

• **State 12 (Execute S-type):** This state is specific to S-type instructions, such as SLL, SLR, SLLV, and SLRV. It performs the shift or logical operations dictated by the instruction and computes the result. The necessary control signals for the execution of S-type instructions are set up in this state, and the state transitions to State 13 for the write back.

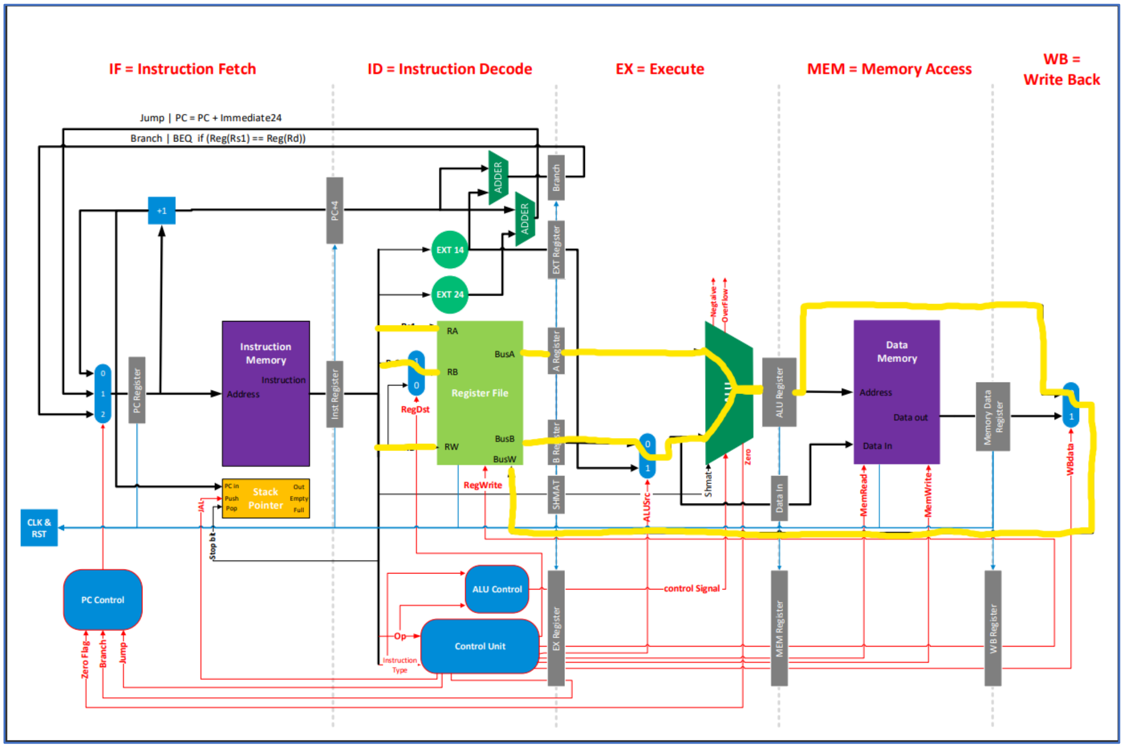
• **State 13 (Write Back of S-type):** After the execution of an S-type instruction, the result is written back to the register file in this state. The control signals related to register writing are activated, and the state transitions back to State 0.

• **State 14 (Decode of J-type):** This state covers the decoding of J-type instructions, such as J and JAL. The target address is extracted from the instruction, and the necessary control signals are set up for the subsequent stages. The state transitions to the execution state, i.e., State 0.

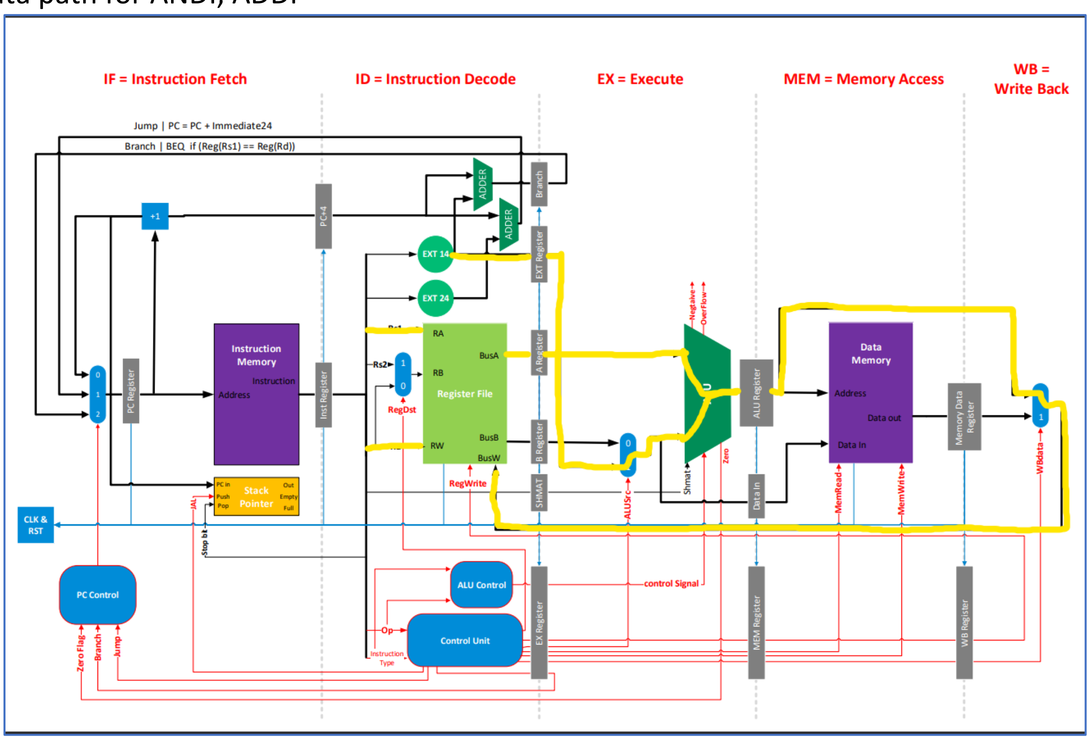
• **State 15 (Execute of LW):** This state is specific to the LW instruction. It involves the execution of the LW instruction, which includes accessing memory to retrieve the data from the specified memory location. The control signals for memory reading are activated, and the state transitions to State 10 for the write back.

DATA PATH

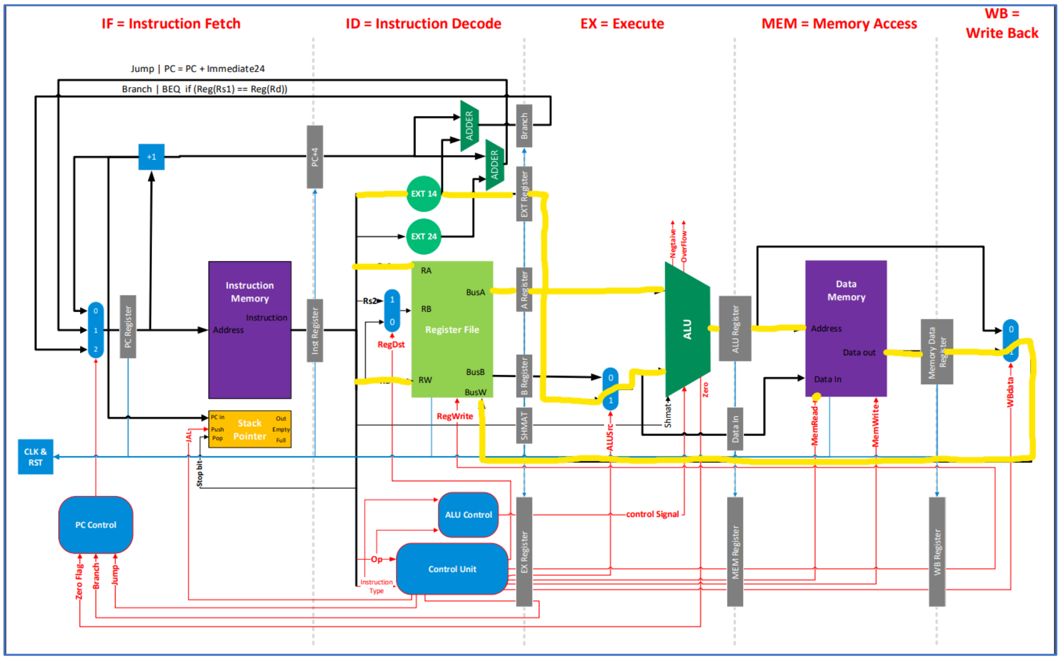
- Data path for AND, ADD, SUB, SLL, SLR, SLLV, SLRV



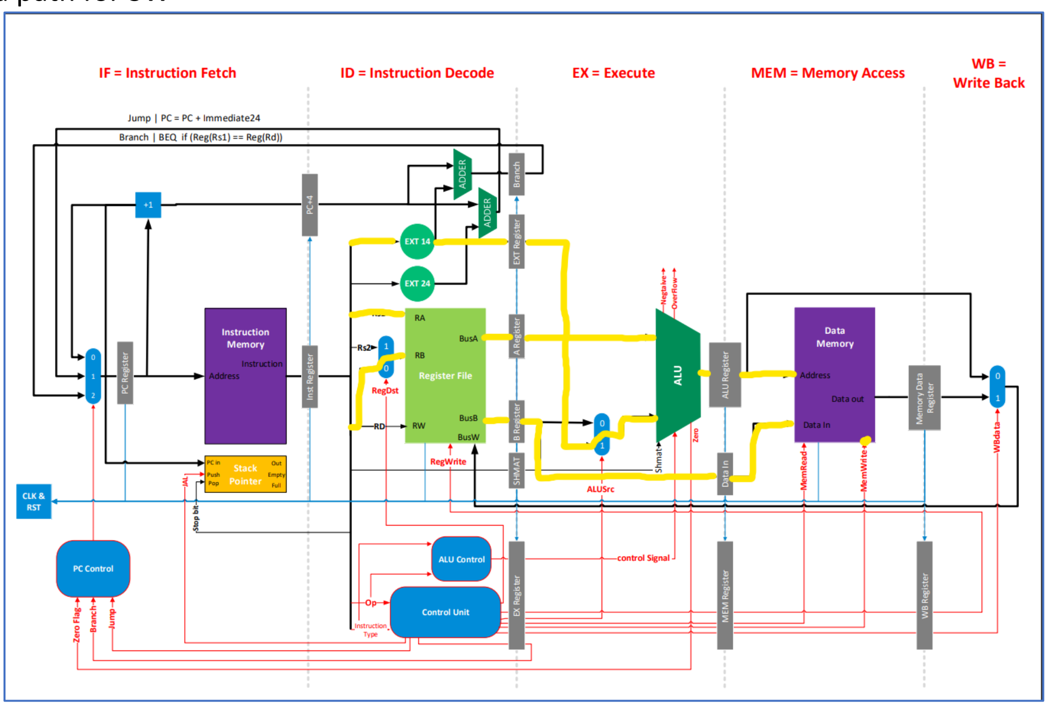
- Data path for ANDI, ADDI



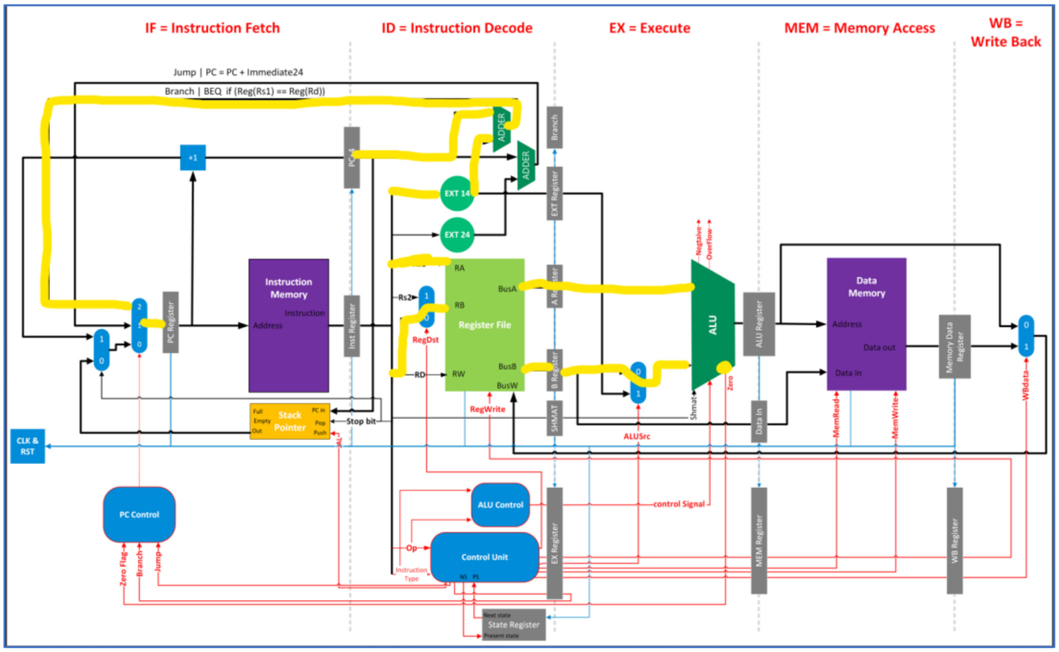
- Data path for LW



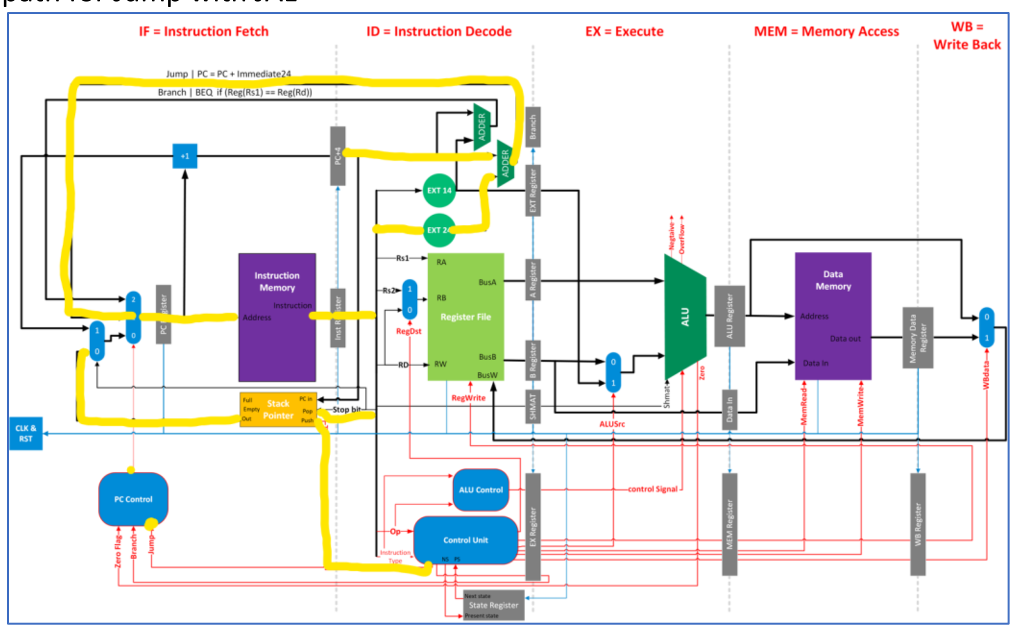
- Data path for SW



- Data path for Branch



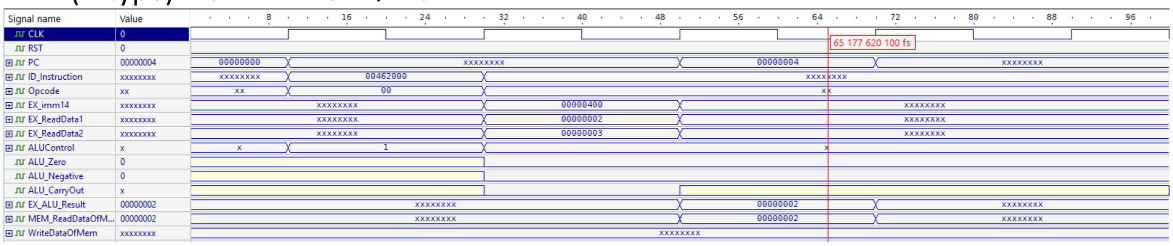
- Data path for Jump with JAL



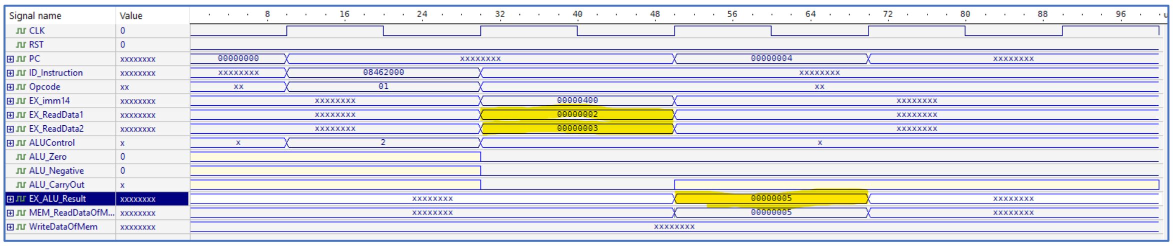
2. SIMULATION AND TESTING

INSTRUCTION TEST RESULTS AND VALIDATION

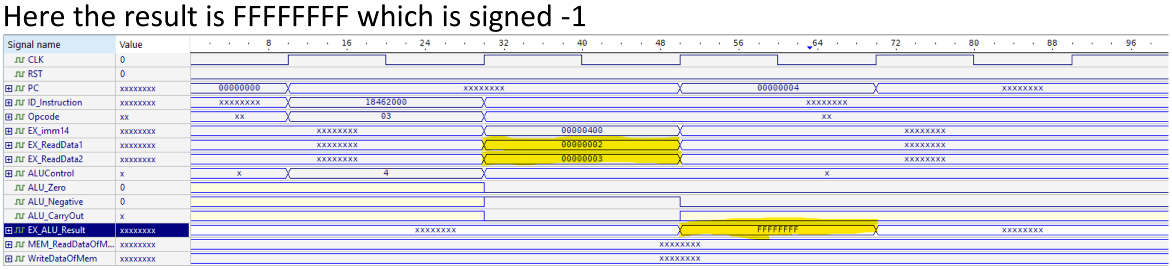
- AND (R-type): Rs1 AND Rs2 → Rd



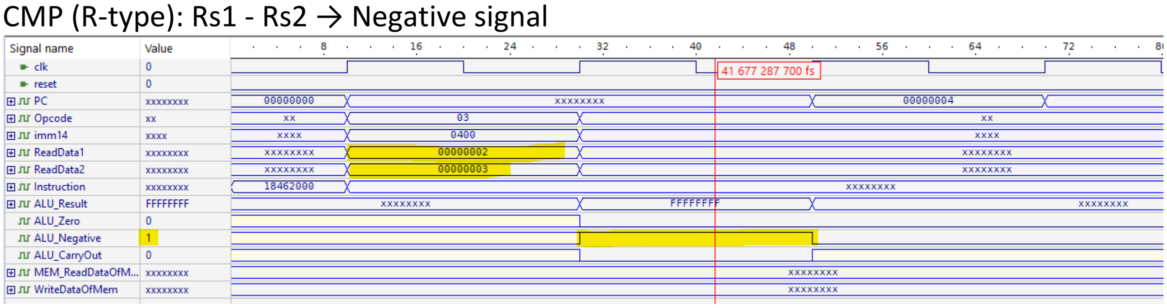
- ADD (R-type): Rs1 + Rs2 → Rd.



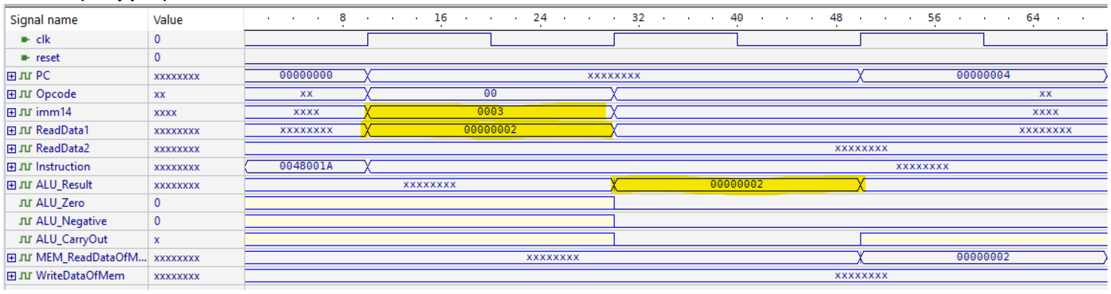
- SUB (R-type): Rs1 - Rs2 → Rd
  2 – 3 → -1
  Here the result is FFFFFFFF which is signed -1



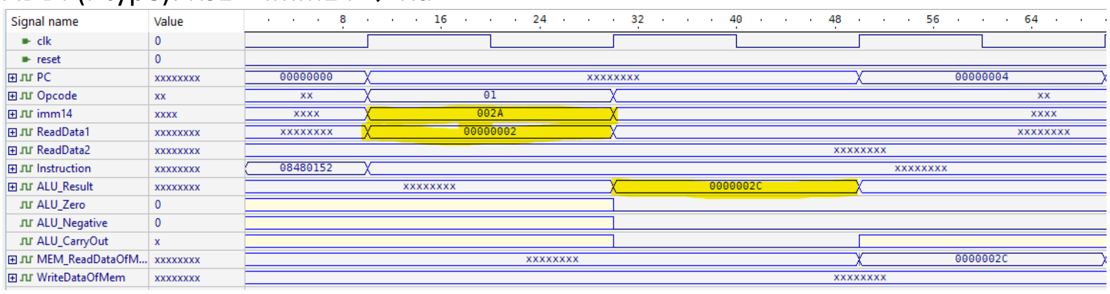
- CMP (R-type): Rs1 - Rs2 → Negative signal



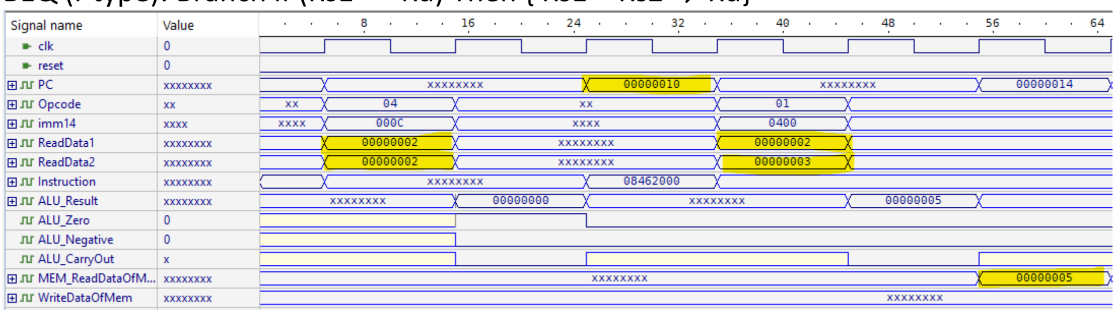
- ADDI (I-type): Rs1 & Imm14 → Rd



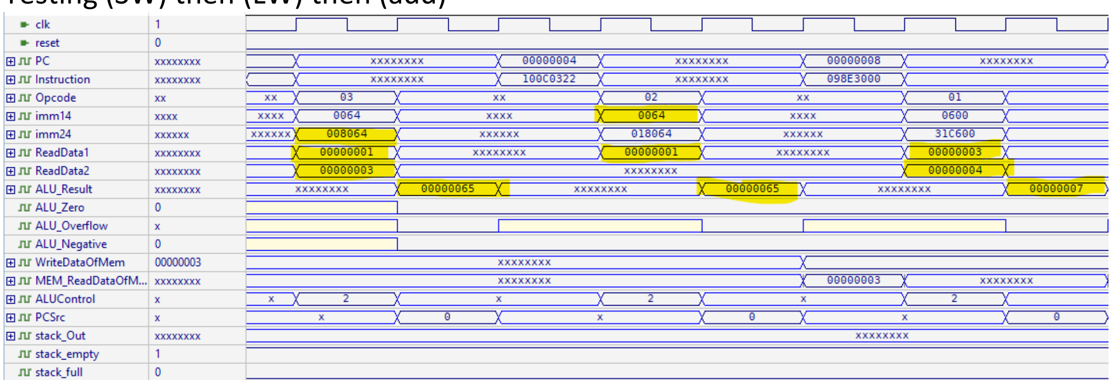
- ADDI (I-type): Rs1 + Imm14 → Rd



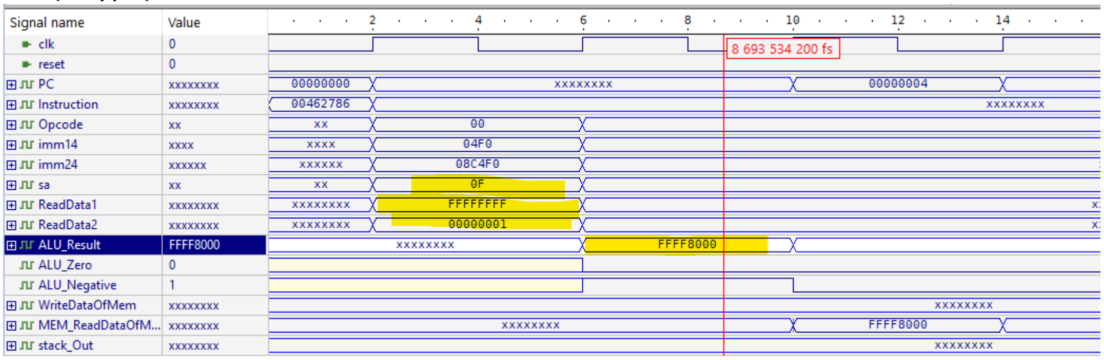
- BEQ (I-type): Branch if (Rs1 == Rd) Then { Rs1 + Rs2 → Rd}



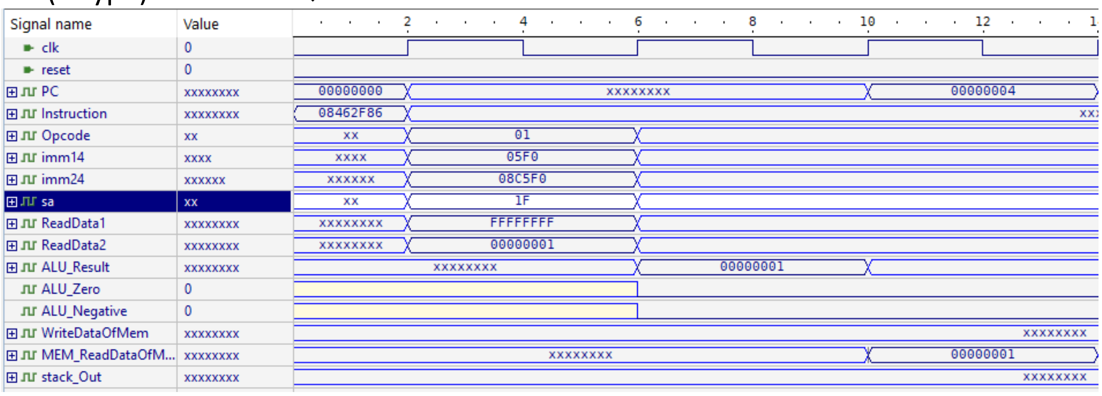
- Testing (SW) then (LW) then (add)



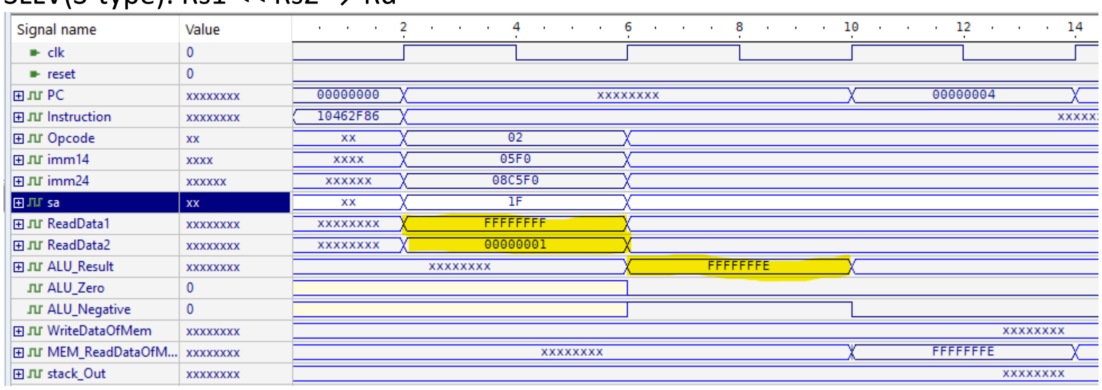
- SLL (S-type): Rs1 << SA → Rd



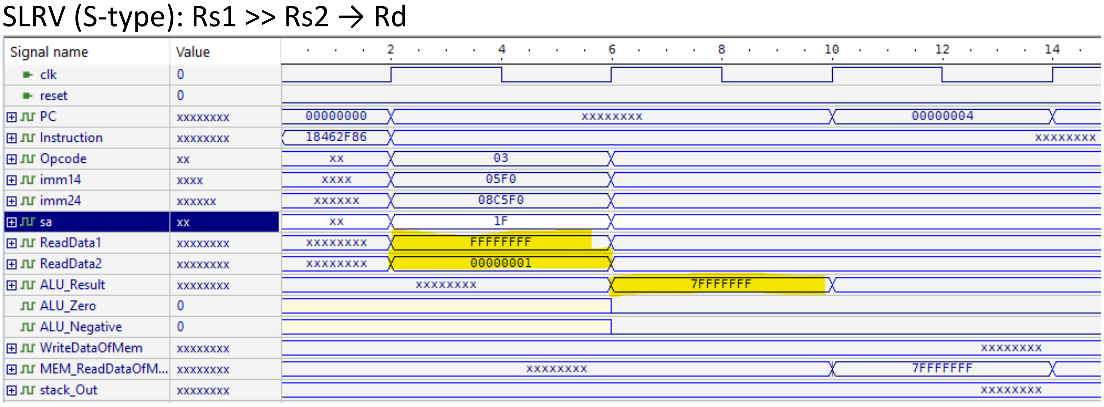
- SLR(S-type): Rs1 >> SA → Rd



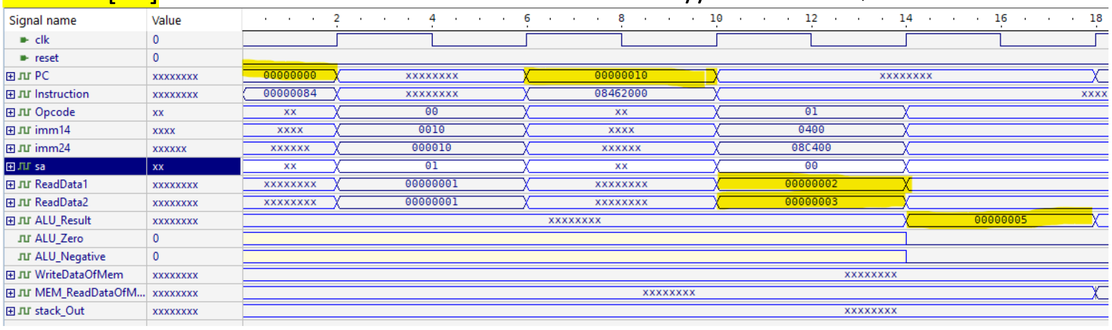
- SLLV(S-type): Rs1 << Rs2 → Rd



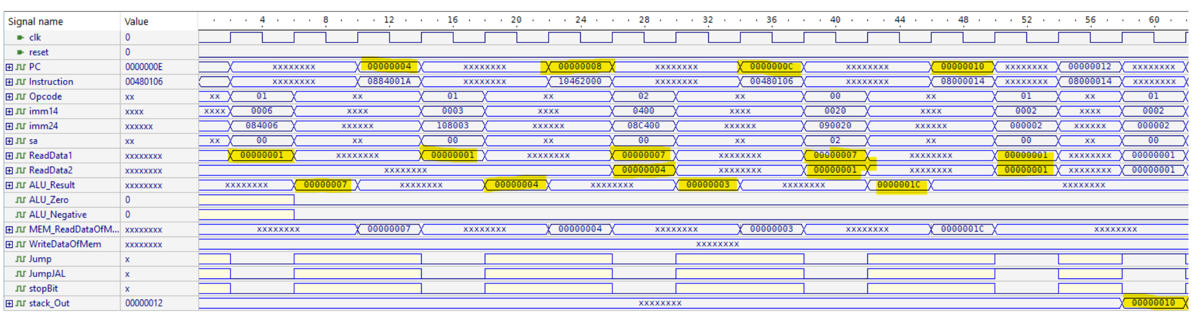
- SLRV (S-type): Rs1 >> Rs2 → Rd



- J (J-type): PC ← PC + Immediate24[26:3]
  Address[0] 00000000000000000000000010000100 // Jump to address 16
  Address[4] 00011000000001000000001100100010 // SW
  Address[8] 00010000000011000000001100100010 //LW
  Address[12] 00001001100011100011000000000000 // ADD
  Address[16] 00001000010001100010000000000000// Rs1 + Rs2 → Rd.



- JAL (J-type): PC ← PC + Immediate24[26:3] Stack. Push (PC + 4)
  - Address[0] = 00001000010000100000000000110010;//R1=R1+7
  - Address[0] = 00001000100001000000000000011010;//R2=R2+3
  - Address[0] = 00010000010001100010000000000000;//R3=R2+R1
  - Address[0] = 00000000010010000000000100000110;//R4=R1<<2
  - Address[0] = 00001000000000000000000000010100;// JAL +2
  - Address[0] = 00000111111111111111111111010100;//Jump -6
  - Address[0] = 00011000000011000000000000000010;// sw r3, [r4+0]
  - Address[0] = 00010001010000000000000000000011;//lw r5, [r0 + 0]



3. APPINDIX

CONTROL STATE

```
module controlState(
    output reg RegDst,
    output reg ALUSrc,
    output reg WBdata,
    output reg RegWrite,
    output reg MemRead,
    output reg MemWrite,
    output reg Branch,
    output reg Jump,
    output reg JumpJAL,
    output reg [2:0] Next_State,
    input [2:0] Present_State,
    input [6:0] Opcode       // Opcode = {instructionType, Opcode}
);

    // OpCodes
    localparam AND   = 7'b0000000,
               ADD   = 7'b0000001,
               SUB   = 7'b0000010,
               CMP   = 7'b0000011,
               ANDI  = 7'b0100000,
               ADDI  = 7'b0100001,
               LW    = 7'b0100010,
               SW    = 7'b0100011,
               BEQ   = 7'b0100100,
               J     = 7'b1000000,
               JAL   = 7'b1000001,
               SLL   = 7'b1100000,
               SLR   = 7'b1100001,
               SLLV  = 7'b1100010,
               SLRV  = 7'b1100011;

    // Stages
    localparam IF = 3'b000,
               ID = 3'b001,
               EX = 3'b010,
               MEM = 3'b011,
               WB = 3'b100;
```

```
always @(Opcode, Present_State)
begin
    case (Present_State)
        IF: begin // Fetch
            RegDst = 1'bx;
            ALUSrc = 1'bx;
            WBdata = 1'bx;
            RegWrite = 1'bx;
            MemRead = 1'bx;
            MemWrite = 1'bx;
            Branch = 1'bx;
            Jump = 1'bx;
            JumpJAL = 1'bx;
            Next_State = ID;
        end

        ID: begin // Decode
            case (Opcode)
                AND: begin
                    RegDst = 1'b1;
                    Jump = 1'b0;
                    JumpJAL = 1'b0;
                    Next_State = EX;
                end
                ADD: begin
                    RegDst = 1'b1;
                    Jump = 1'b0;
                    JumpJAL = 1'b0;
                    Next_State = EX;
                end
                SUB: begin
                    RegDst = 1'b1;
                    Jump = 1'b0;
                    JumpJAL = 1'b0;
                    Next_State = EX;
                end
                CMP: begin
                    RegDst = 1'b1;
                    Jump = 1'b0;
                    JumpJAL = 1'b0;
                    Next_State = EX;
                end
                ANDI: begin
                    RegDst = 1'bx;
```

```
        Jump = 1'b0;
        JumpJAL = 1'b0;
        Next_State = EX;
    end
    ADDI: begin
        RegDst = 1'bx;
        Jump = 1'b0;
        JumpJAL = 1'b0;
        Next_State = EX;
    end
    LW: begin
        RegDst = 1'bx;
        Jump = 1'b0;
        JumpJAL = 1'b0;
        Next_State = EX;
    end
    SW: begin
        RegDst = 1'b0;
        Jump = 1'b0;
        JumpJAL = 1'b0;
        Next_State = EX;
    end
    BEQ: begin
        RegDst = 1'b0;
        Jump = 1'b0;
        JumpJAL = 1'b0;
        Next_State = EX;
    end
    J: begin
        RegDst = 1'bx;
        Jump = 1'b1;
        JumpJAL = 1'b0;
        Next_State = IF;
    end
    JAL: begin
        RegDst = 1'bx;
        Jump = 1'b1;
        JumpJAL = 1'b1;
        Next_State = IF;
    end
    SLL: begin
        RegDst = 1'b1;
        Jump = 1'b0;
        JumpJAL = 1'b0;
        Next_State = EX;
```

```
            end
            SLR: begin
                RegDst = 1'b1;
                Jump = 1'b0;
                JumpJAL = 1'b0;
                Next_State = EX;
            end
            SLLV: begin
                RegDst = 1'b1;
                Jump = 1'b0;
                JumpJAL = 1'b0;
                Next_State = EX;
            end
            SLRV: begin
                RegDst = 1'b1;
                Jump = 1'b0;
                JumpJAL = 1'b0;
                Next_State = EX;
            end
            default: begin
                // Handle unrecognized opcode
                RegDst = 1'bx;
                Jump = 1'bx;
                JumpJAL = 1'bx;
                Next_State = IF;
            end
        endcase
    end
    EX: begin // Execute
        case (Opcode)
            AND: begin
                ALUSrc = 1'b0;
                Branch = 1'b0;
                Next_State = WB;
            end
            ADD: begin
                ALUSrc = 1'b0;
                Branch = 1'b0;
                Next_State = WB;
            end
            SUB: begin
                ALUSrc = 1'b0;
                Branch = 1'b0;
                Next_State = WB;
            end
```

```
CMP: begin
    ALUSrc = 1'b0;
    Branch = 1'b0;
    Next_State = IF;
end
ANDI: begin
    ALUSrc = 1'b1;
    Branch = 1'b0;
    Next_State = WB;
end
ADDI: begin
    ALUSrc = 1'b1;
    Branch = 1'b0;
    Next_State = WB;
end
LW: begin
    ALUSrc = 1'b1;
    Branch = 1'b0;
    Next_State = MEM;
end
SW: begin
    ALUSrc = 1'b1;
    Branch = 1'b0;
    Next_State = MEM;
end
BEQ: begin
    ALUSrc = 1'b0;
    Branch = 1'b1;
    Next_State = IF;
end
SLL: begin
    ALUSrc = 1'b0;
    Branch = 1'b0;
    Next_State = WB;
end
SLR: begin
    ALUSrc = 1'b0;
    Branch = 1'b0;
    Next_State = WB;
end
SLLV: begin
    ALUSrc = 1'b0;
    Branch = 1'b0;
    Next_State = WB;
end
```

```
            SLRV: begin
                ALUSrc = 1'b0;
                Branch = 1'b0;
                Next_State = WB;
            end
            default: begin
                ALUSrc = 1'bx;
                Branch = 1'bx;
                Next_State = IF;
            end
        endcase
    end
    MEM: begin // Memory
        case (Opcode)
            LW: begin
                MemRead = 1'b1;
                MemWrite = 1'b0;
                Next_State = WB;
            end
            SW: begin
                MemRead = 1'b0;
                MemWrite = 1'b1;
                Next_State = IF;
            end
            default: begin
                // Handle unrecognized opcode
                MemRead = 1'bx;
                MemWrite = 1'bx;
                Next_State = IF;
            end
        endcase
    end

    WB: begin // Writeback
        case (Opcode)
            AND: begin
                WBdata = 1'b0;
                RegWrite = 1'b1;
                Next_State = IF;
            end
            ADD: begin
                WBdata = 1'b0;
                RegWrite = 1'b1;
                Next_State = IF;
            end
```

```
SUB: begin
    WBdata = 1'b0;
    RegWrite = 1'b1;
    Next_State = IF;
end
ANDI: begin
    WBdata = 1'b0;
    RegWrite = 1'b1;
    Next_State = IF;
end
ADDI: begin
    WBdata = 1'b0;
    RegWrite = 1'b1;
    Next_State = IF;
end
LW: begin
    WBdata = 1'b1;
    RegWrite = 1'b1;
    Next_State = IF;
end
SLL: begin
    WBdata = 1'b0;
    RegWrite = 1'b1;
    Next_State = IF;
end
SLR: begin
    WBdata = 1'b0;
    RegWrite = 1'b1;
    Next_State = IF;
end
SLLV: begin
    WBdata = 1'b0;
    RegWrite = 1'b1;
    Next_State = IF;
end
SLRV: begin
    WBdata = 1'b0;
    RegWrite = 1'b1;
    Next_State = IF;
end
default: begin
    // Handle unrecognized opcode
    WBdata = 1'bx;
    RegWrite = 1'bx;
    Next_State = IF;
```

```
                    end
                endcase
            end
        endcase
    end
endmodule
```

ALU

```
/////////////////////////////////////////////////// ALU 32bit
module alu(Output, carryOut, zero, overflow, negative, BussA, BussB, Shamt, controlSignal);
    // alu outputs
    output overflow, negative, zero, carryOut;
    output signed [31:0] Output;
    // alu inputs
    input signed [31:0] BussA;
    input signed [31:0] BussB;
    input [4:0] Shamt;
    input [3:0] controlSignal;

    // registers declarations
    reg signed [31:0] BussBComp;
    reg signed [31:0] Output;
    reg overflow, negative, zero, carryOut;

    // control signals
    parameter   AND  = 4'b0001,
                ADD  = 4'b0010,
                SUB  = 4'b0011,
                CMP = 4'b0100,
                BEQ = 4'b0101,
                SLL = 4'b1100,
                SLR = 4'b1101,
                SLLV = 4'b1110,
                SLRV = 4'b1111;



    always @(BussA, BussB, controlSignal, Shamt) begin
        case (controlSignal)
            AND: begin
                Output = BussA & BussB;
                overflow = 1'b0;
                carryOut = 1'b0;
            end
```

```
ADD: begin
    Output = BussA + BussB;
    carryOut = (BussA[31] && BussB[31]) || (!Output[31] && (BussA[31] || BussB[31]));
    if ((BussA[31] && BussB[31]) && !Output[31]) overflow = 1'b1;
    else if ((!BussA[31] && !BussB[31]) && Output[31]) overflow = 1'b1;
    else overflow = 1'b0;
end
SUB: begin
    BussBComp = ~BussB + 1;
    Output = BussA + BussBComp;
    carryOut = (BussA[31] && BussBComp[31]) || (!Output[31] && (BussA[31] || BussBComp[31]));
    if ((BussA[31] && BussBComp[31]) && !Output[31]) overflow = 1'b1;
    else if ((!BussA[31] && !BussBComp[31]) && Output[31]) overflow = 1'b1;
    else overflow = 1'b0;
end
CMP: begin
    BussBComp = ~BussB + 1;
    Output = BussA + BussBComp;
    carryOut = (BussA[31] && BussBComp[31]) || (!Output[31] && (BussA[31] || BussBComp[31]));
end
BEQ: begin
    if (BussA == BussB) Output = 32'b0;
    else Output = 32'b1;
    overflow = 1'b0;
    carryOut = 1'b0;
end
SLL: begin
    Output = BussA << Shamt;
    overflow = 1'b0;
    carryOut = 1'b0;
end
SLR: begin
    Output = BussA >> Shamt;
    overflow = 1'b0;
    carryOut = 1'b0;
end
SLLV: begin
    Output = BussA << BussB;
    overflow = 1'b0;
    carryOut = 1'b0;
end
SLRV: begin
    Output = BussA >> BussB;
    overflow = 1'b0;
    carryOut = 1'b0;
```

```
            end
            default: begin
                Output = 32'bx;
                overflow = 1'bx;
                carryOut = 1'bx;
            end
        endcase
    end
    // zero and negative for all cases
    always @(Output) begin
        if (!Output) zero = 1'b1;
        else zero = 1'b0;

        if (Output[31]) negative = 1'b1;
        else negative = 1'b0;
    end
endmodule
```