
**Faculty of Engineering & Technology**

**Electrical & Computer Engineering Department**

# COMPUTER ARCHITECTURE - ENCS4370

## Project # 2: RISC processor

---

**Prepared by:**

Jehad Hamayel 1200348

Musab Masalmah 1200078

Mohammad Salem 1200651

**Instructors:** Dr.Aziz Qaroush & Dr. Ayman Hroub

**Sections:** 1 + 3

**Date:** 1/7/2023

**Place:** Masri 302&404

# Table of Contents

# Table of Figures

# Table of Tables

## Design Description:

The design is a multi-cycle processor with a five-stage pipeline: fetch, decode, execute, memory access, and write back. The processor follows the provided architecture and incorporates the following components:

## Data Path:

### 1. Program Counter (PC):

PC is a special-purpose register that holds the address of the next instruction to be fetched in the instruction memory. The PC gets updated at the end of each instruction execution to point to the next instruction. It plays a crucial role in controlling the program flow and ensuring the sequential execution of instructions.

### 2. Instruction Memory:

Instruction Memory is a component that stores the program instructions in the processor. It is accessed by the PC to fetch the next instruction for execution, the Instruction Memory holds the machine code instructions, each represented by a fixed number of bits (e.g., 32 bits).

### 3. Register File:

Register File is a component in the processor that stores a set of general-purpose registers. It provides fast access to the registers, allowing data to be read from and written to them during instruction execution, the Register File is typically organized as an array of registers, each capable of holding a fixed-size data value (e.g., 32 bits).

### 4. Stack Pointer (SP):

The Stack Pointer (SP) is a special-purpose register that points to the top of the stack in memory, It is used to manage the stack operations, such as pushing and popping data.

### 5. Control Stack:

Control Stack: is a specialized memory structure used for storing return addresses during function calls and handling program flow. The Control Stack File is a storage component that holds the values of the return addresses. It allows for efficient retrieval and modification of return addresses during subroutine calls and return operations.

### 6. ALU (Arithmetic Logic Unit):

ALU is a fundamental component in a processor responsible for performing arithmetic and logical operations. It operates on binary data, executing operations such as addition, subtraction, AND, OR, and more. The ALU takes input operands, performs the specified operation, and produces a result along with status flags such as zero and carry.

### 7. Data Memory

Data Memory is a component in the processor that stores data values during program execution. It is used to read from and write data during memory access instructions, such as load and store operations. The Data Memory is typically organized as a separate memory module, distinct from the instruction memory, and provides storage for variables, arrays, and other data used by the program.

### 8. Some Multiplexers:

Some Multiplexers are a component we used to control the information that must be enter to the inputs for each of the components in the stages.

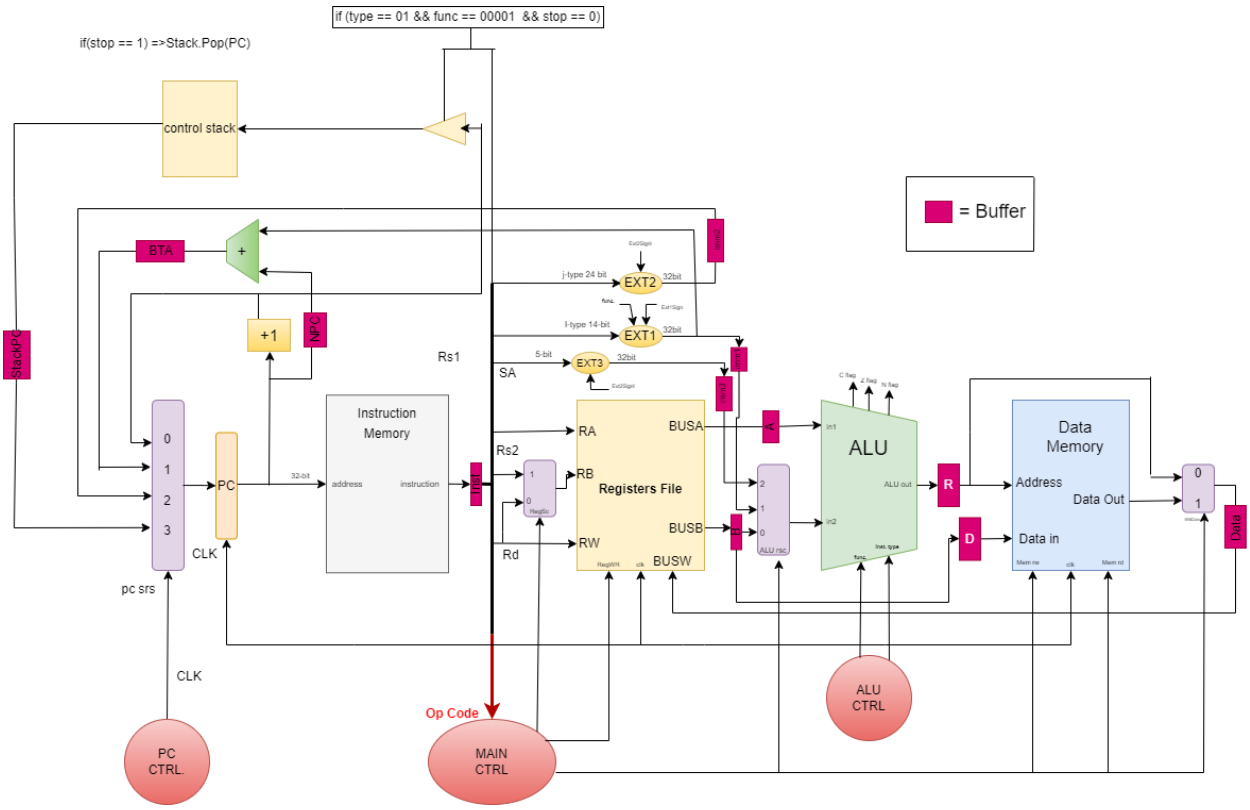


*Figure 1: Data Path Of Multi-cycle RISC processor*

## Control Units:

### 1. The Main Control Unit:

The Main Control Unit is responsible for generating control signals that coordinate the operation of various components in the processor. It manages the execution of instructions by providing control signals to multiplexers, ALU, register file, and other components. The Main Control Unit interprets the opcode of the current instruction and generates the appropriate control signals to ensure proper instruction execution, and the table below discover all his work.

### 2. The PC Control Unit:

The PC Control Unit is a subunit of the Control Unit that specifically handles the Program Counter (PC) operations. It generates control signals to increment the PC, update it with branch or jump targets, and handle exceptions or interrupts. The PC Control Unit ensures that the PC is properly updated to fetch the next instruction and maintain the correct program flow.

### 3. The ALU Control Unit:

The ALU Control Unit is responsible for generating control signals that determine the operation to be performed by the Arithmetic Logic Unit (ALU). Based on the instruction function code, the ALU Control Unit determines the specific arithmetic or logical operation to be executed by the ALU. It generates control signals to select the appropriate ALU function and handles any necessary data manipulation or flag updates based on the operation performed.

## How data path works:

| Instruction No | instruction type | # of stages | instruction type | pc srs | RedWR | memWR | memRd | WBData |
|---|---|---|---|---|---|---|---|---|
| 1 | AND | 4 | R-type | 0 | 1 | 0 | 0 | 0 |
| 2 | ADD | 4 | R-type | 0 | 1 | 0 | 0 | 0 |
| 3 | SUB | 4 | R-type | 0 | 1 | 0 | 0 | 0 |
| 4 | CMP | 3 | R-type | 0 | 1 | 0 | 0 | x |
| 5 | ANDI | 4 | I-type | 0 | 1 | 0 | 0 | 0 |
| 6 | ADDI | 4 | I-type | 0 | 1 | 0 | 0 | 0 |
| 7 | LW | 5 | I-type | 0 | 1 | 0 | 1 | 1 |
| 8 | SW | 4 | I-type | 0 | 0 | 1 | 0 | x |
| 9 | BEQ(not taken) | 3 | I-type | 0 | 0 | 0 | 0 | x |
| 9 | BEQ(taken) | 3 | I-type | 1 | 0 | 0 | 0 | x |
| 10 | J | 2 | J-type | 2 | 0 | 0 | 0 | x |
| 11 | JAL | 2 | J-type | 3 | 0 | 0 | 0 | x |
| 12 | SLL | 4 | S-type | 0 | 1 | 0 | 0 | 0 |
| 13 | SLR | 4 | S-type | 0 | 1 | 0 | 0 | 0 |
| 14 | SLLV | 4 | S-type | 0 | 1 | 0 | 0 | 0 |
| 15 | SLRV | 4 | S-type | 0 | 1 | 0 | 0 | 0 |

*Table 1:Truth Table of Data Path Part1*

| Instruction No | instruction type | RegSc | ALU src | Inst Type | func | Ext1Signl | Ext2Signl | Ext3Signl |
|---|---|---|---|---|---|---|---|---|
| 1 | AND | 1 | 0 | 2'b00 | 5'b00000 | 0 | 0 | 0 |
| 2 | ADD | 1 | 0 | 2'b00 | 5'b00001 | 0 | 0 | 0 |
| 3 | SUB | 1 | 0 | 2'b00 | 5'b00010 | 0 | 0 | 0 |
| 4 | CMP | 1 | 0 | 2'b00 | 5'b00011 | 0 | 0 | 0 |
| 5 | ANDI | x | 1 | 2'b10 | 5'b00000 | 1 | 0 | 0 |
| 6 | ADDI | x | 1 | 2'b10 | 5'b00001 | 1 | 0 | 0 |
| 7 | LW | x | 1 | 2'b10 | 5'b00010 | 1 | 0 | 0 |
| 8 | SW | x | 1 | 2'b10 | 5'b00011 | 1 | 0 | 0 |
| 9 | BEQ(not taken) | 0 | 1 | 2'b10 | 5'b00100 | 1 | 0 | 0 |
| 9 | BEQ(taken) | 0 | 1 | 2'b10 | 5'b00100 | 1 | 0 | 0 |
| 10 | J | x | x | 2'b01 | 5'b00000 | 0 | 1 | 0 |
| 11 | JAL | x | x | 2'b01 | 5'b00001 | 0 | 1 | 0 |
| 12 | SLL | x | 2 | 2'b11 | 5'b00000 | 0 | 0 | 1 |
| 13 | SLR | x | 2 | 2'b11 | 5'b00001 | 0 | 0 | 1 |
| 14 | SLLV | 1 | 0 | 2'b11 | 5'b00010 | 0 | 0 | 0 |
| 15 | SLRV | 1 | 0 | 2'b11 | 5'b00011 | 0 | 0 | 0 |

*Table 2:Truth Table of Data Path Part2*

## Boolean Expretion:

| Instruction No | instruction type | Boolean Expretion with sum of minterms |
|---|---|---|
| 1 | AND | (pc srs1)` + (pc srs2)` + (RedWR) + (memWR)` + (memRd)` + (WBData)` + (RegSc) + (ALU src1)` + (ALU src2)` + (Ext1Signl)` + (Ext2Signl)` + (Ext3Signl)` |
| 2 | ADD | (pc srs1)` + (pc srs2)` + (RedWR) + (memWR)` + (memRd)` + (WBData)` + (RegSc) + (ALU src1)` + (ALU src2)` + (Ext1Signl)` + (Ext2Signl)` + (Ext3Signl)` |
| 3 | SUB | (pc srs1)` + (pc srs2)` + (RedWR) + (memWR)` + (memRd)` + (WBData)` + (RegSc) + (ALU src1)` + (ALU src2)` + (Ext1Signl)` + (Ext2Signl)` + (Ext3Signl)` |
| 4 | CMP | (pc srs1)` + (pc srs2)` + (RedWR) + (memWR)` + (memRd)` +(RegSc) + (ALU src1)` + (ALU src2)` + (Ext1Signl)` + (Ext2Signl)` + (Ext3Signl)` |
| 5 | ANDI | (pc srs1)` + (pc srs2)` + (RedWR) + (memWR)` + (memRd)` + (WBData)` + (ALU src1)` + (ALU src2) + (Ext1Signl) + (Ext2Signl)` + (Ext3Signl)` |
| 6 | ADDI | (pc srs1)` + (pc srs2)` + (RedWR) + (memWR)` + (memRd)` + (WBData)` + (ALU src1)` + (ALU src2) + (Ext1Signl) + (Ext2Signl)` + (Ext3Signl)` |
| 7 | LW | (pc srs1)` + (pc srs2)` + (RedWR) + (memWR)` + (memRd) + (WBData) + (ALU src1)` + (ALU src2) + (Ext1Signl) + (Ext2Signl)` + (Ext3Signl)` |
| 8 | SW | (pc srs1)` + (pc srs2)` + (RedWR)` + (memWR) + (memRd)` + (ALU src1) + (ALU src2)` + (Ext1Signl) + (Ext2Signl)` + (Ext3Signl)` |
| 9 | BEQ(not taken) | (pc srs1)` + (pc srs2)` + (RedWR)` + (memWR)` + (memRd)` + (RegSc)` + (ALU src1) + (ALU src2)` + (Ext1Signl) + (Ext2Signl)` + (Ext3Signl)` |
| 9 | BEQ(taken) | (pc srs1) + (pc srs2)` + (RedWR)` + (memWR)` + (memRd)` + (RegSc)` + (ALU src1) + (ALU src2)` + (Ext1Signl) + (Ext2Signl)` + (Ext3Signl)` |
| 10 | J | (pc srs1) + (pc srs2)` + (RedWR)` + (memWR)` + (memRd)` + (Ext1Signl)` + (Ext2Signl) + (Ext3Signl)` |
| 11 | JAL | (pc srs1) + (pc srs2) + (RedWR)` + (memWR)` + (memRd)` + (Ext1Signl)` + (Ext2Signl) + (Ext3Signl)` |
| 12 | SLL | (pc srs1)` + (pc srs2)` + (RedWR) + (memWR)` + (memRd)` + (WBData)` + (ALU src1) + (ALU src2)` + (Ext1Signl)` + (Ext2Signl)` + (Ext3Signl) |
| 13 | SLR | (pc srs1)` + (pc srs2)` + (RedWR) + (memWR)` + (memRd)` + (WBData)` + (ALU src1) + (ALU src2)` + (Ext1Signl)` + (Ext2Signl)` + (Ext3Signl) |
| 14 | SLLV | (pc srs1)` + (pc srs2)` + (RedWR) + (memWR)` + (memRd)` + (WBData)` + (RegSc) + (ALU src1)` + (ALU src2)` + (Ext1Signl)` + (Ext2Signl)` + (Ext3Signl)` |
| 15 | SLRV | (pc srs1)` + (pc srs2)` + (RedWR) + (memWR)` + (memRd)` + (WBData)` + (RegSc) + (ALU src1)` + (ALU src2)` + (Ext1Signl)` + (Ext2Signl)` + (Ext3Signl)` |

*Table 3:Boolean Functions*

## Implementation Details and Design Choices:

1. **Instruction Size:** Each instruction is 32 bits in size, accommodating the opcode, register addresses, immediate values, and other necessary fields.
2. **Register File:** The processor includes a 32 x 32 register file for efficient data storage and retrieval. Each register can hold a 32-bit value.
3. **Program Counter (PC):** The PC keeps track of the address of the current instruction being executed. It gets updated at the end of each instruction to fetch the next instruction.
4. **Control Stack:** A separate on-chip memory is used for the control stack, which stores return addresses. The stack pointer (SP) holds the address of the empty element at the top of the stack.
5. **ALU and Zero, Carry, Negative Signals**: The ALU in the processor is designed to perform arithmetic and logical operations. In addition to the "zero" signal, which indicates if the result of the last operation is zero, the ALU also generates the "carry" and "negative" signals. The "carry" signal indicates if a carry or borrow occurred during addition or subtraction operations, while the "negative" signal indicates if the result is negative. These signals are important for subsequent instructions or conditional branching that rely on the ALU operation results. Including these signals provides a more comprehensive and versatile ALU functionality.
6. **Separate Data and Instruction Memories:** The processor has separate memories for data and instructions. This separation allows independent access to data and instructions, improving overall performance.
7. **Five-Stage:** The processor follows a five-stage to enable concurrent instruction execution and maximize throughput.
8. **Multi-Cycle Processor:** The processor is designed as a multi-cycle processor, where each instruction takes multiple cycles to complete. This design allows for simpler control logic and more flexibility in optimizing the performance.

## Notable Features:

1. **Stack Support:** The processor incorporates a control stack for managing function calls and returns, allowing for structured program execution.

2. **Instruction Types:** The processor supports four instruction types (R-Type, I-Type, J-Type, and S-Type) to provide a wide range of operations, enabling flexible programming.

3. **ALU Zero Signal:** Enables efficient branching and conditional execution based on ALU results.

4. **Negative Signal:** Indicates if the result of an operation is negative, supporting signed arithmetic.

5. **Carry Signal:** Handles carry and borrow operations during addition and subtraction, ensuring accurate arithmetic calculations.

## Multi-cycle CPU Finite State Machine (FSM) - State Transition Diagram:

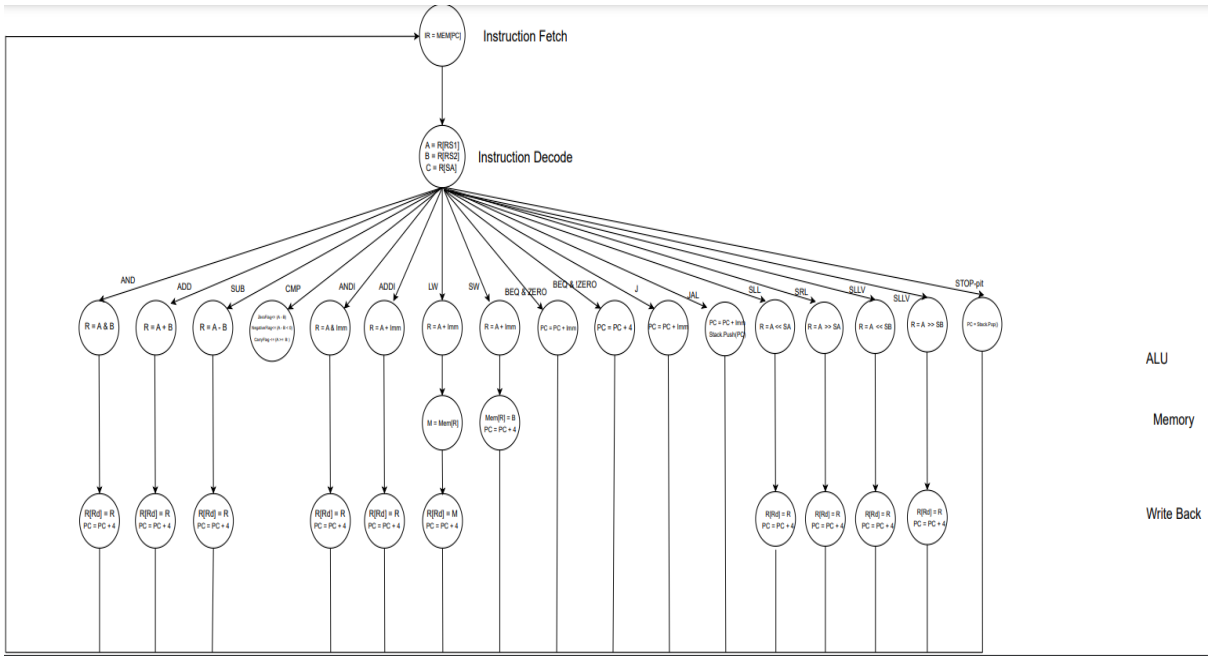


*Figure 2:Finite State Machine(FSM)*

## Correctness of the individual components

We implemented these instructions to illustrate the work of the component.

instmem[0]=32'b00010100000100000000000000000100;//lw $R8, 0($R16)

instmem[1]=32'b00010100010100100000000000000100;//lw $R9, 0($R17);

instmem[2]=32'b00010100100101000000000000000100;//lw $R10, 0($R18);

instmem[3]=32'b00001000000000000000000000011010;//JALLABEL: JAL FUNCJAL ;

instmem[4]=32'b00100010010101000000000000100100;//BEQ $R9 , $R10 , EXIT ;

instmem[5]=32'b00000111111111111111111111110010;//J JALLABEL;

instmem[6]=32'b00001000010000101000000000000000;//FUNCJAL:add $R1, $R1, $R8;

instmem[7]=32'b00001010010100111111111111111101;//ADDI $R9, $R9, -1;

instmem[8]=32'b00011100000000100000000000011100;//SW  $R1, 3($R16)

## 1. Program Counter (PC):

As shown in the following Figure 3, there is a block for the Program Counter (PC) register, which has one 32 bit input and one 32 bit output, so that the address is stored for the next instruction to execute, and it is running at the positive edge of the clk.

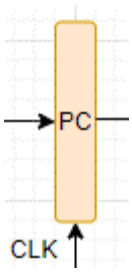


*Figure 3:Program Counter (PC) Block*

```
reg [31:0]PC;
initial
    begin
        PC=32'b00000000000000000000000000000000;
        inPC=PC;
        programCounter(inPC,outPC);
    end
task programCounter(input [31:0] inPC,output reg [31:0]outPC);
PC = inPC;
$display("\nPC = %b",PC);
outPC=PC;
endtask
```

*Figure 4:Program Counter (PC) Code*

Program Counter (PC) output if we enter the instruction address in the figure below:

```
PC = 00000000000000000000000000000000

Fetch:


Fetch From Instruction Memory :
00010100000100000000000000000100
```

*Figure 5:Program Counter (PC) Output*

## 2. Instruction Memory:

As shown in the following Figure, there is a block for Instruction Memory, which has one 32 bit input and one 32-bit output, so that the instruction address enters it and the instruction exits from it.

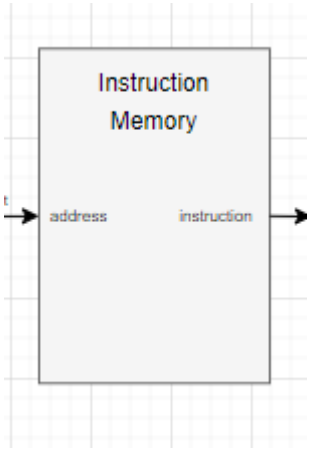


*Figure 6:Instruction Memory Block*

```
task instMem(input [31:0] instaddress,output reg [31:0] instdata);
reg [31:0] instmem [0:1023];
instmem[0]=32'b00010100000100000000000000000100;//lw $R8, 0($R16)
instmem[1]=32'b00010100010100100000000000000100;//lw $R9, 0($R17);
instmem[2]=32'b00010100100101000000000000000100;//lw $R10, 0($R18);
instmem[3]=32'b00001000000000000000000000011010;//JALLABEL: JAL FUNCJAL ;
instmem[4]=32'b00100010010101000000000000100100;//BEQ $R9 , $R10 , EXIT ;
instmem[5]=32'b00000011111111111111111111110010;//J JALLABEL;
instmem[6]=32'b00001000010000101000000000000000;//FUNCJAL:add $R1, $R1, $R8;
instmem[7]=32'b00001010010100111111111111111101;//ADDI $R9, $R9, -1;
instmem[8]=32'b00011100000000100000000000011100;//SW  $R1, 3($R16)
instdata = instmem[instaddress];
 $display("\nFetch From Instruction Memory :\n%b",instdata);
endtask
```

*Figure 7:Instruction Memory Code*

The output of the Instruction Memory if we enter address of load instruction in the figure below:

```
Fitch:

Fitch From Instruction Memory :
00010100000100000000000000000100
------------------
```

*Figure 8:Instruction Memory Output*

## 3. Registers File:

As shown in the following figure, there is a block for the Registers File, which contains three 5-bit entries, an entry to control writing to it, a 32-bit entry to store it in, and two 32-bit outputs, so that it has 32 32-bit registers in which the data is stored, and it is running at the positive edge of the clk.

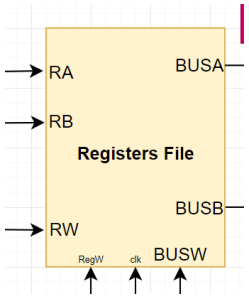


*Figure 9:Registers File Block*

```
reg [31:0] regmem [31:0];
assign regmem[16]=0;
assign regmem[17]=1;
assign regmem[18]=2;
assign regmem[1]=0;

task regsitersRead(input [4:0] rs1,rs2,output [31:0] bus1,bus2);
bus1 = regmem [rs1];
bus2 = regmem [rs2];
endtask

task regsitersWrite(input [4:0] rd,input [31:0] busw,input regW);

if(regW)
regmem[rd]=busw;

endtask
```

*Figure 10:Registers File Code*

As shown, the result in the following form is the implementation of a set of instructions so that the Registers File is read and written:

```
------------------
Decode:

rs1 =  1 ,  busA=          0

rs2 =  8 ,  busB =         11

rd =  1
------------------
```

*Figure 11:Registers File Output for read*

```
Write Back:

Write Back on rd =  8    ,  Data in to Register File =          11
```

*Figure 12:Registers File Storing for write*

## 4. ALU:

As shown in the figure, there is the ALU Block, whose function is to carry out arithmetic operations and other tasks, so that two inputs of 32 bits size are entered into it, and also the input for selecting the operation to be executed is called ALU op, and these operations result in what results in the effect on the flags, and the other on the ALU out result.

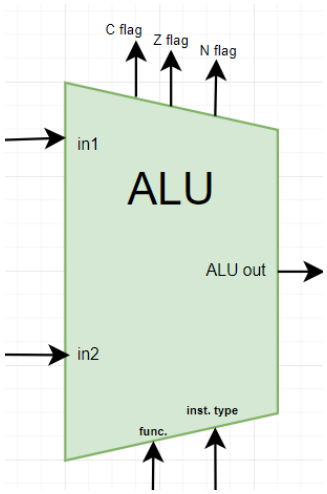


*Figure 13:ALU Block*

As shown in the set of instructions that were implemented previously, the content of R1 and R8 was collected and placed in R1 so that the addition process was 0 + 11 equal to 11 and stored in R1.

```
Decode:

rs1 =  1 ,  busA=          0

rs2 =  8 ,  busB =         11

rd =  1
------------------
Excute:

ALU Output =          11
------------------
```

*Figure 14:ALU output*

```
task ALU(input [31:0] busA,busB,input [1:0] InstType,input [4:0] aluOP,output reg NegativeFlag,output reg Carry,output reg ZeroFlag,output reg [31:0] re

  case(InstType)
    //*********************************R-type*********************************************
    2'b00:  // Type R
    begin
      case(aluOP)
      5'b00000:  //AND
        result = busA & busB;

      5'b00001:  // ADD
        result = busA + busB;

      5'b00010:  // SUB
        result = busA - busB;

      5'b00011:  // CMP
      begin
        result=(busA - busB);
        NegativeFlag=(result<0);
        ZeroFlag=(result==0);
        Carry=(busA<busB);
      end
      default:
        result = 32'b0;

      endcase
    end
    //*********************************I-type*********************************************
    2'b10:  // Type I
    begin
      case(aluOP)
      5'b00000:  //ANDI
        result = busA & busB;

      5'b00001:  // ADDI
        result = busA + busB;
      5'b00010:  // ADD for Address load LW
        result = busA + busB;
      5'b00011:  // ADD for Address store SW
        result = busA + busB;
      5'b00100:  // BEQ
      begin
        result = busA - busB;
        ZeroFlag=(result==0);
      end
      default:
        result = 32'b0;

      endcase
    end
    //*****************************************J-type*************************************************
    2'b11:  // Type J
    begin
      case(aluOP)
      5'b00000:  //SLL
      begin
          while(busB!=0)
          begin
            busA= {busA[30:0],1'b0};
            busB=busB-1;

          end
          result=busA;
      end
      5'b00001:  // SLR
      begin
          while(busB!=0)
          begin
            busA= {1'b0,busA[31:1]};
            busB=busB-1;

          end
          result=busA;
      end
      5'b00010:  // SLLV
       begin
          while(busB!=0)
          begin
            busA= {busA[30:0],1'b0};
            busB=busB-1;

          end
          result=busA;
      end

      5'b00011:  // SLRV
        begin
          while(busB!=0)
          begin
            busA= {1'b0,busA[31:1]};
            busB=busB-1;
```

*Figure 15::ALU Code*

## 5. Data Memory:

As shown in the figure, there is the Data Memory Block, whose function is to store the data, so that two inputs of 32 bits size enter it, one is an address inside the memory and the other is data for storage, and also two inputs for choosing the operation to be executed in writing or execution called Mem rw, Mem rd so that Data is stored or read from memory, and it is running at the positive edge of the clk.

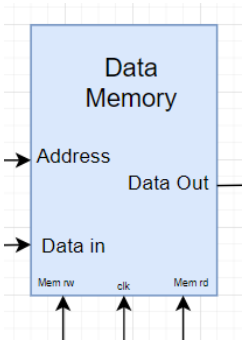


*Figure 16:Data Memory Block*

```
task dataMemory(input memRd,memWr,input [31:0] address,Datain,output reg [31:0]Dataout);
reg [31:0]memey[0:1023];
memey[0]=11;
memey[1]=5;
memey[2]=0;
        if(memRd==2'b1 && memWr==2'b0)
            begin
                Dataout=memey[address];
            end
        else if(memRd==2'b0 && memWr==2'b1)
               begin
                memey[address]=Datain;
               end

endtask
```

*Figure 17:Data Memory Code*

As shown in the following figure, there is a value that came out of memory when executing the load instruction.

```
------------------
Memory:


Memory Output for Load:00000000000000000000000000001011
------------------
```

*Figure 18:Data Memory Output*

## 6. Extenders:

As shown in the following figure, there are Extenders Blocks, which extend the bits until they are in the 32-bit format in order to work on them correctly.

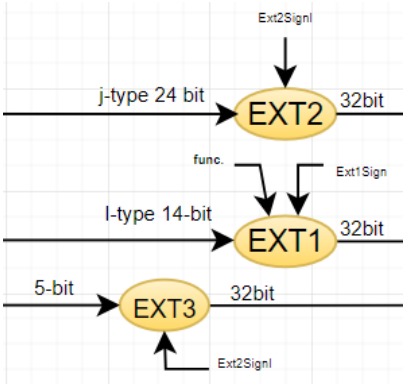


*Figure 19:Extenders Blocks*

```
task Ext1(input [4:0] aluOP,input [13:0]imm,output [31:0]outExt);
if(aluOP==5'b00000)
    outExt={18'b000000000000000000,imm};
else
    begin
    if(imm[13]==1'b1)
        outExt={18'b111111111111111111,imm};
    else
        outExt={18'b000000000000000000,imm};
    end
endtask

task Ext2(input [23:0]imm,output [31:0]outExt);
if(imm[23]==1'b1)
    outExt={8'b11111111,imm};
else
    outExt={8'b00000000,imm};
endtask

task Ext3(input [4:0]SA,output [31:0]outExt);
outExt={27'b000000000000000000000000000,SA};
endtask
```

*Figure 20:Extenders Blocks Code*

The Ext1 is special in the branch instruction, the Ext2 is special for the jump instructions, and the Ext3 is special for the value to be shifting.

## 7. Control Stack:

As shown in the figure, the Control Stack is shown, whose function is to store the private address of the function that is called using the JAL.

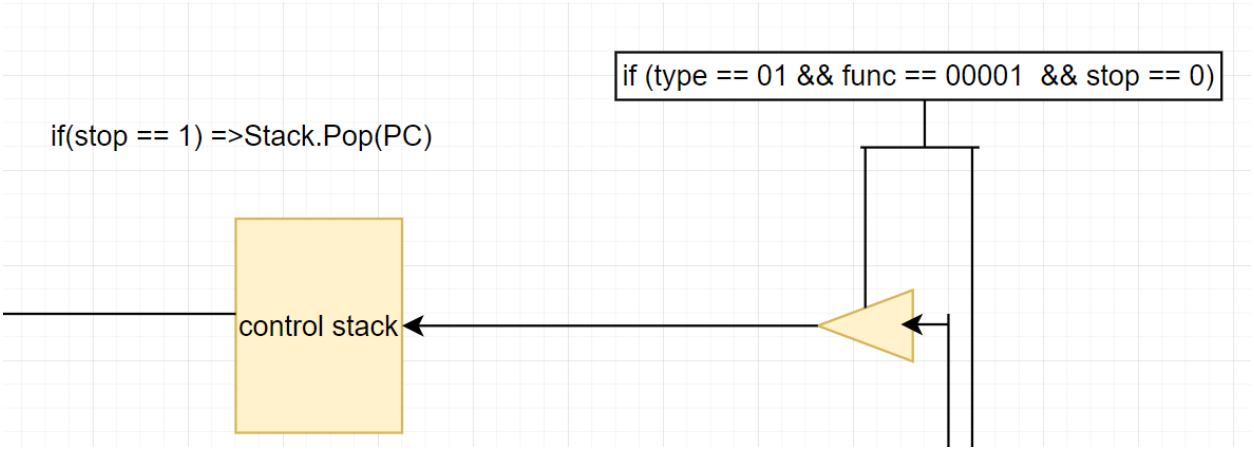


*Figure 21:Control Stack Block*

The following figure shows the code for building the stack.

```
reg [9:0]sp;
initial sp=10'b0000000000;
reg [31:0]stackMem[1023:0];

task pushStack(input [31:0] DataIn);
stackMem[sp]=DataIn;
$display("\nPush in the Stack = %b",stackMem[sp]);
sp =sp + 1;
endtask

task popStack(output [31:0] DataOut);
sp =sp - 1;
DataOut=stackMem[sp];
stackMem[sp]=32'b00000000000000000000000000000000;
$display("\nPop From the Stack = %b",DataOut);
endtask
```

*Figure 22:Stack Code*

As shown, the result in the following figure is when we process the JAL it calls a function and stores PC +1 at the top of the stack.

```
Decode:

Ext2:00000000000000000000000000000011
Push in the Stack = 00000000000000000000000000000100
```

*Figure 23:push to the stack*

As shown, the result in the following figures is that when we reached the last instruction in the function, the value of the PC that was stored when the function was called was restored.

```
Pop From the Stack = 00000000000000000000000000000100
PC = 00000000000000000000000000000100
Fetch:

Fetch From Instruction Memory :
00100010010101000000000000100100
```

*Figure 24:pop for the stack*

## Simulation and Testing

We have implemented a set of instructions that we wrote ourselves, which are the following:

LW $R8, 0($R16)

LW $R9, 0($R17)

LW $R10, 0($R18)

JALLABEL: JAL FUNCJAL

BEQ $R9, $R10, EXIT

J JALLABEL

FUNCJAL: ADD $R1, $R1, $R8

ADDI $R9, $R9, -1

EXIT: SW $R1, 3($R16)

And we needed some values that were stored in the data memory and instruction memory:

Mem[reg($R16)] =11

Mem[reg($R17)] =5

Mem[reg($R18)] =0

$R16=0

$R17=1

$R18=2

$R1=0

The goal of the program is to find the product of multiplying two numbers. As in the example here, we multiplied 5 by 11 and the result is 55 by using a function. We call it 5 times to add the 11 with 0 5 times and the result comes out with us.

## Test Bench:

We have created the following Test Bench, which only controls clk, where the entire Test Bench will be discussed in print and wave forms.

```
module DataPath_test;
 //Declarations of test inputs and outputs

reg clk;
wire [31:0]ALUOut,MemOut;
wire zeroFlag,carryFlag,negativeFlag;

//Call GeneralMod module and give it inputs and outputs
GeneralMod DP(clk, zeroFlag, carryFlag, negativeFlag, ALUOut, MemOut);

//Give clk values and change it every 0.5 units of time
initial clk = 0;
always #0.5 clk = ~clk;
initial #220 $finish;
endmodule
```

*Figure 25:Test Bench*

At first, we implemented the following:

LW $R8, 0($R16)

As the figure below shows the stages that the instruction LW went through:

```
PC = 00000000000000000000000000000000

Fetch:


Fetch From Instruction Memory :
00010100000100000000000000000100
------------------
Decode:

rs1:16


rd: 8


Ext1:00000000000000000000000000000000
------------------
Excute:


Address:00000000000000000000000000000000
------------------
Memory:


Memory Output for Load:00000000000000000000000000001011
------------------
Write Back:

Write Back on rd =  8   ,  Data in to Register File =         11

PC = 00000000000000000000000000000001
```

*Figure 26:LW instruction*

The stages were implemented as follows, the instruction was taken from the instruction file, and then the decoding process took place, where the register was prepared to be read from and stored on, where the address was taken from R16 and an account was made with offset 0 through the ALU, then the data was extracted from the data memory and stored on the register file through the write back stage and the rest of the LW is processed in the program like this instruction.

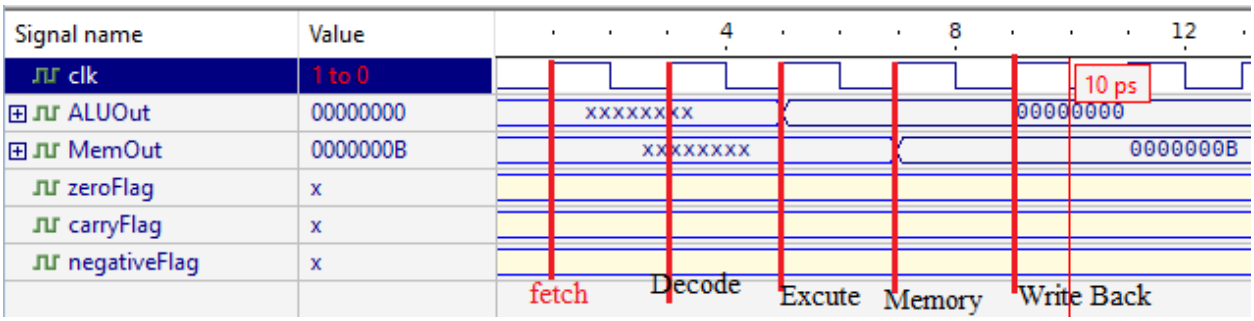


*Figure 27: wave form of load instruction*

When we execute the JAL function and call the function, the result is as shown in the following figure:

JAL FUNCJAL

FUNCJAL:

ADD $R1, $R1, $R8

ADDI $R9, $R9, -1

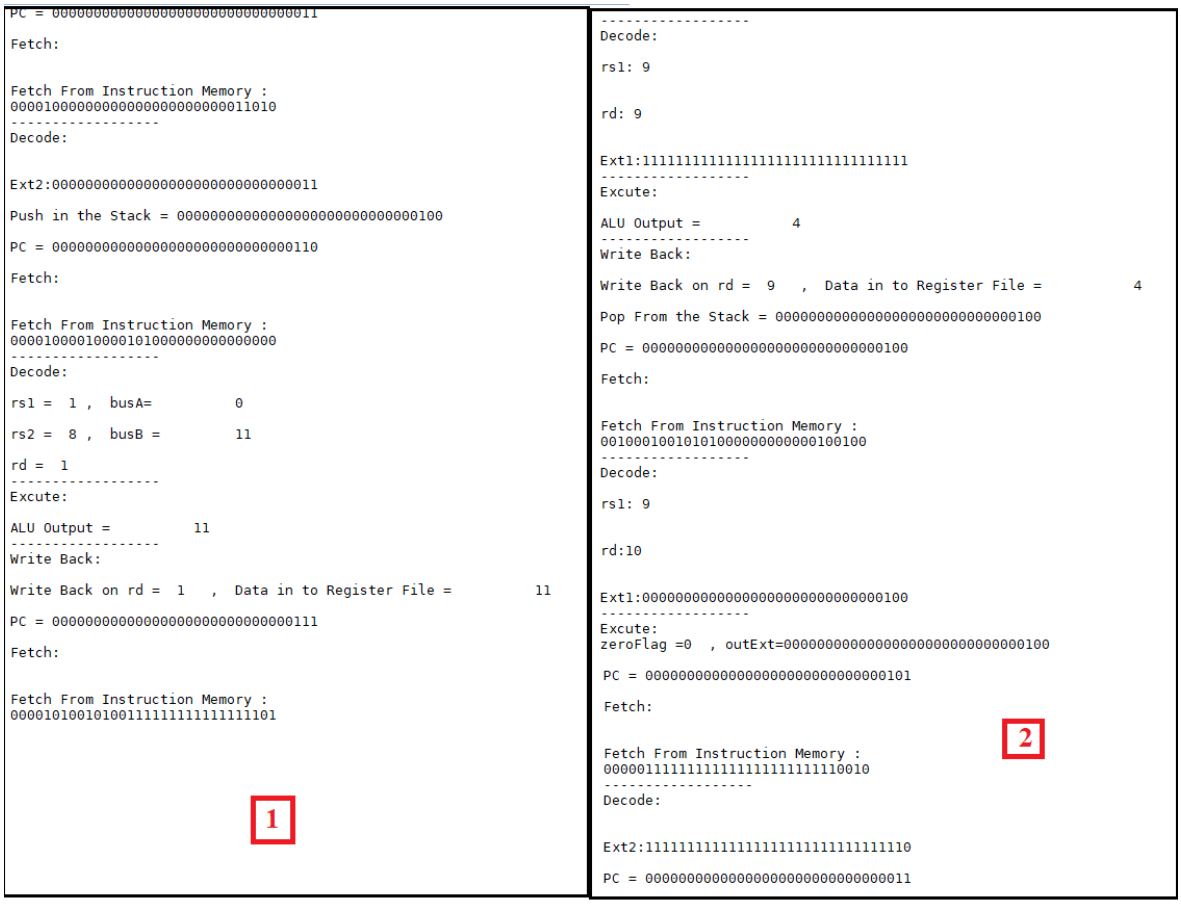


*Figure 28:JAL instruction and the begin of the program*

As shown in Figure 28, the address of the instruction that follows the JAL is saved in the stack, then it is jumped to execute the function and upon completion at the end of the function the address of the instruction that was previously saved in the stack is restored and executed since the counter was not equal to zero and the branch was not executed, so it executed The next instruction where he jumped to JAL and repeated the steps until he carried out the desired operation.

Also, in the following figure, the waveform shows how the instructions went in the data path, and also shows the output for the ALU and also for the memory.

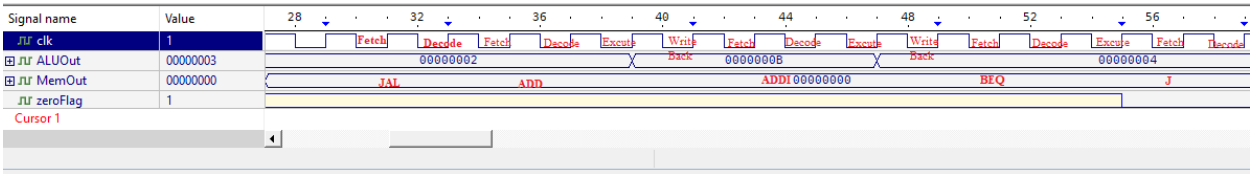


*Figure 29:wave form for begin program*

```
: PC = 00000000000000000000000000000100
:
: Fetch:
:
:
: Fetch From Instruction Memory :
: 00100010010101000000000000100100
: -------------------
: Decode:
:
: rs1: 9
:
:
: rd:10
:
:
: Ext1:00000000000000000000000000000100
: -------------------
: Excute:
: zeroFlag =1  , outExt=00000000000000000000000000000100

PC = 00000000000000000000000000001000

Fetch:


Fetch From Instruction Memory :
00011100000100000000000000011100
------------------
Decode:

rs1:16


rd: 8


Ext1:00000000000000000000000000000011
------------------
Excute:


Address:00000000000000000000000000000011
------------------
Memory:

Data In to Momery for Store= 00000000000000000000000000001011

PC = 00000000000000000000000000001001
```

*Figure 30:End of program*

As shown in the figure 30, when the program is completed, the BEQ checks the end of the counter and then jumps to the end of the program to store the value and exit, and so the program finishes executing as the program needs 92 cycles, and as shown in the wave form, the end of the programs, as in the figure31.

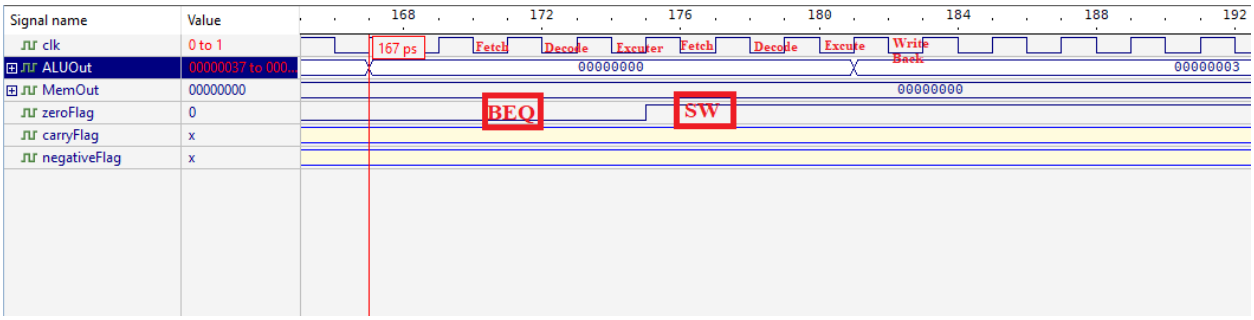


*Figure 31:wave form for end of the program*

As shown in Figure 32 This value has been stored

```
Write Back:

Write Back on rd =  1    ,  Data in to Register File =          55
```

*Figure 32:Data that we store*

```
PC = 00000000000000000000000000001000

Fetch:


Fetch From Instruction Memory :
00011100000000100000000000011100
------------------
Decode:

rs1:16


rd: 1


Ext1:00000000000000000000000000000011
------------------
Excute:


Address:00000000000000000000000000000011
------------------
Memory:

Data In to Momery for Store= 00000000000000000000000000110111
```

*Figure 33:SW result*

We have implemented a set of other instructions that was different than the above code instructions, which are the following:

LW $R8, 0($R16)

LW $R9, 0($R17)

AND $R10, $R8, $R9

SUB $R10, $R8, $R9

CMP $R8, $R9

ANDI $R0, $R8, 5

SLL $R1, $R8,2

SLR $R1, $R8,2

SLLV $R1, $R8,$R17

SLRV $R1, $R8,$R17

And we needed some values that were stored in the data memory and instruction memory:

Mem[reg($R16)]=5

Mem[reg($R17)]=11

$R16=0

$R17=1

The goal of the program is to make a test for all instruction types and format, and in this code we have all instruction format and types that doesn't exist in the previous code.

At first, we implemented the following:

LW $R8, 0($R16)

LW $R9, 0($R17)

And the result and simulation was been as the previous code above.

Then we implemented this instruction, the result is as shown in the following figure:

AND $R10, $R8, $R9

```
PC = 00000000000000000000000000000010

Fetch:


Fetch From Instruction Memory :
00000010000101001001000000000000
------------------
Decode:

rs1 =  8 ,  busA=          11

rs2 =  9 ,  busB =           5

rd = 10
------------------
Excute:

ALU Output =           1
------------------
Write Back:

Write Back on rd = 10   ,  Data in to Register File =           1
```

*Figure 34:AND result*

As shown in the figure above, in the first the instruction fetch values from register file and put them in the registers, the execute them in ALU, and write back the result on Rd.

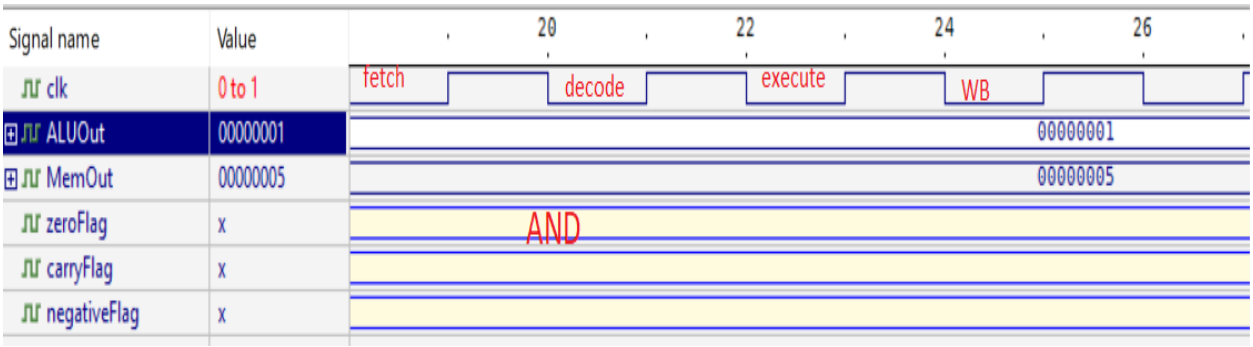


*Figure 35:AND waveform result*

Then we implemented this instruction, the result is as shown in the following figure:

SUB $R10, $R8, $R9

```
PC = 00000000000000000000000000000011

Fetch:


Fetch From Instruction Memory :
00010010000101001001000000000000
------------------
Decode:

rs1 =  8 ,  busA=          11

rs2 =  9 ,  busB =           5

rd = 10
------------------
Excute:

ALU Output =           6
------------------
Write Back:

Write Back on rd = 10   ,  Data in to Register File =           6
```

*Figure 36:SUB result*

As shown in the figure above, in the first the instruction fetch values from register file and put them in the registers, the execute them in ALU (SUB), the output value from subtract 11 – 5 = 6, and write back the (result = 6) on Rd.

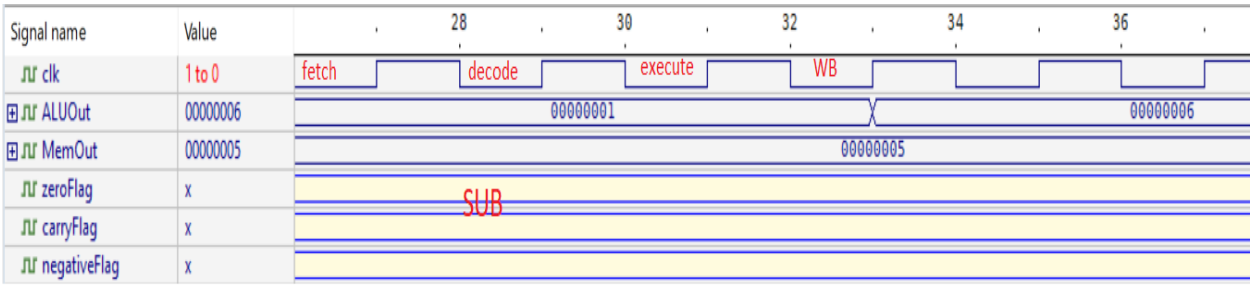


*Figure 37:SUB waveform result*

Then we implemented this instruction, the result is as shown in the following figure:

CMP $R8, $R9

```
PC = 00000000000000000000000000000100

Fetch:


Fetch From Instruction Memory :
00011010000000001001000000000000
------------------
Decode:

rs1 =  8 ,  busA=         11

rs2 =  9 ,  busB =          5

rd =  0
------------------
Excute:

Zero Flag = 0 ,  Negative Flag = 0, carry Flag = 1
```

*Figure 38:CMP result*

As shown in the figure above, in the first the instruction fetch values from register file and put them in the registers, the execute them in ALU (CMP), this instruction just have 3 stages, in this instruction the value of flags will be change.

(Z-flag = 0) because (11 – 6) != 0

(N-flag=0) because (11 – 6) !< 0

(C=flag = 1) because (11-6) make carry flag = 1

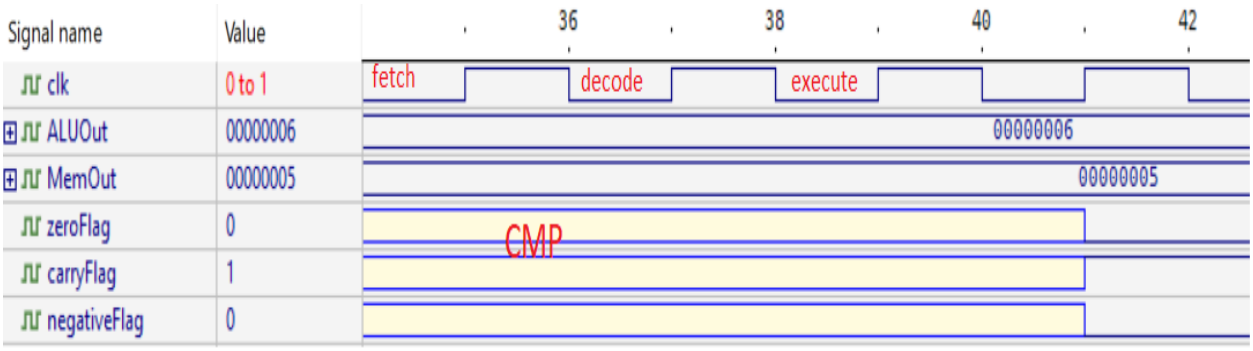


*Figure 39:CMP waveform result*

Then we implemented this instruction, the result is as shown in the following figure:

ANDI $R0, $R8, 5

```
PC = 00000000000000000000000000000101

Fetch:


Fetch From Instruction Memory :
00000010000000000000000000101100
------------------
Decode:

rs1: 8


rd: 0


Ext1:00000000000000000000000000000101
------------------
Excute:

ALU Output =           1
------------------
Write Back:

Write Back on rd =  0    ,  Data in to Register File =           1
```

*Figure 40:ANDI result*

As shown in the figure above, in the first the instruction fetch values from register file and put them in the registers and extend the immediate value and put in in the ALU to execute, then execute them in ALU (ANDI), the output value from subtract 11 & 5 = 1, and write back the (result = 1) on Rd.

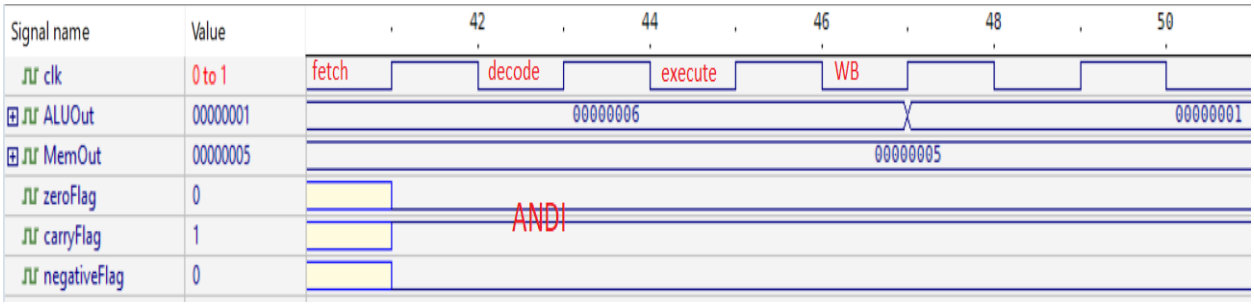


*Figure 41:ANDI waveform result*

Then we implemented the shift instructions, the result is as shown in the following figure:

SLL $R1, $R8,2

SLR $R1, $R8,2

SLLV $R1, $R8, $R17

SLRV $R1, $R8, $R17

```
PC = 00000000000000000000000000000110

Fetch:


Fetch From Instruction Memory :
00000010000000100000000100000110
------------------
Decode:
rs1 =  8 ,  busA=         11

rs2 =  0 ,  busB =          1

SA:00010

Ext3:00000000000000000000000000000010
------------------
Excute:

Before Shifting =         11

Shifting Amount=          2

After Shifting =         44
------------------
Write Back:

Write Back on rd =  1   ,  Data in to Register File =         44
```

*Figure 42:SLL result*

```
PC = 00000000000000000000000000000111

Fetch:


Fetch From Instruction Memory :
00001010000000100000000100000110
------------------
Decode:
rs1 =  8 ,  busA=         11

rs2 =  0 ,  busB =          1

SA:00010

Ext3:00000000000000000000000000000010
------------------
Excute:

Before Shifting =         11

Shifting Amount=          2

After Shifting =          2
------------------
Write Back:

Write Back on rd =  1   ,  Data in to Register File =          2
```

*Figure 43:SLR result*

```
PC = 00000000000000000000000000001000

Fetch:


Fetch From Instruction Memory :
00010010000000110001000000000110
------------------
Decode:
rs1 =  8 ,  busA=         11

rs2 = 17 ,  busB =          1

rd =  1
------------------
Excute:

Before Shifting =         11

Shifting Amount=          1

After Shifting =         22
------------------
Write Back:

Write Back on rd =  1   ,  Data in to Register File =          22
```

*Figure 44:SLLV result*

```
PC = 00000000000000000000000000001001

Fetch:


Fetch From Instruction Memory :
00011010000000110001000000000110
------------------
Decode:
rs1 =  8 ,  busA=         11

rs2 = 17 ,  busB =          1

rd =  1
------------------
Excute:

Before Shifting =         11

Shifting Amount=          1

After Shifting =          5
------------------
Write Back:

Write Back on rd =  1   ,  Data in to Register File =           5
```

*Figure 45:SLRV result*

SLL $R1, $R9,2: the value in R9 = 11 shifted left by the immediate = 2, and then R1 = 11 << 2 = 44

SLR $R1, $R9,2: the value in R9 = 11 shifted right by the immediate = 2, and then R1 = 11 >> 2 = 2

SLLV $R1, $R9, $R17: the value in R9 = 11 shifted left by the (R17 = 1), and then R1 = 11 << 1 = 22

SLRV $R1, $R9, $R17: the value in R9 = 11 shifted right by the (R17 = 1), and then R1 = 11 >> 1 = 5

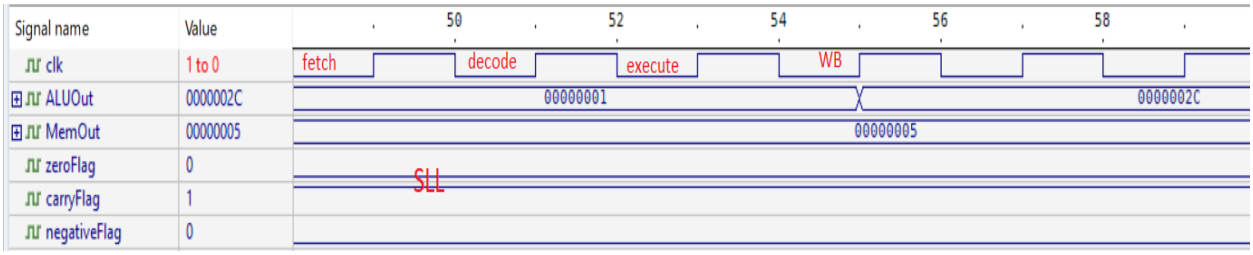


*Figure 46:SLL waveform result*

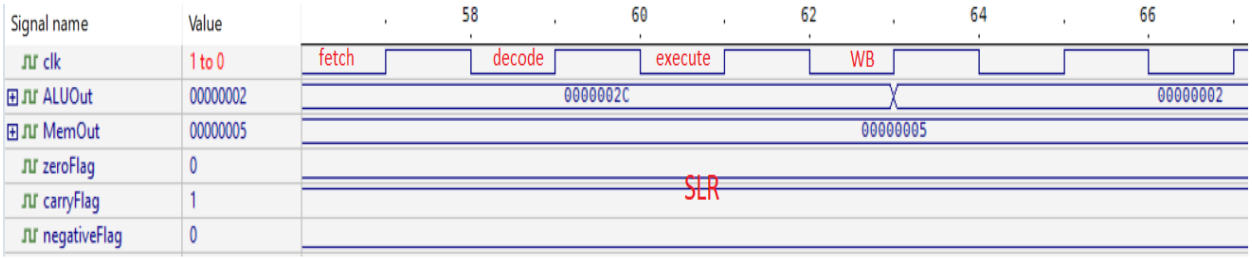


*Figure 47:SLR waveform result*

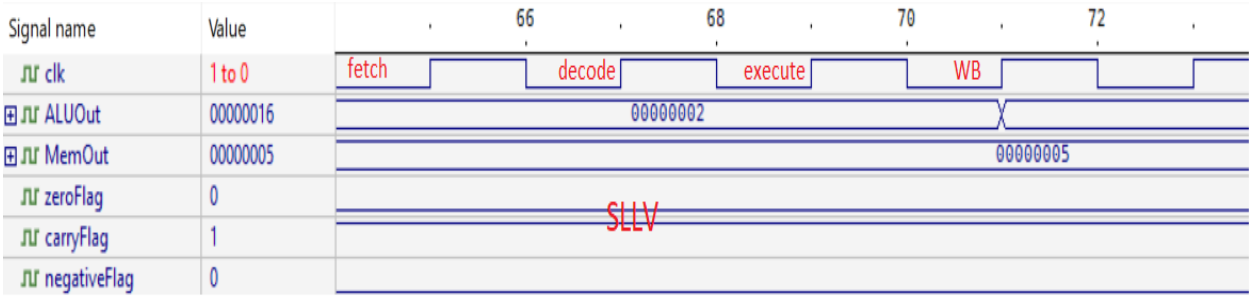


*Figure 48:SLLV waveform result*

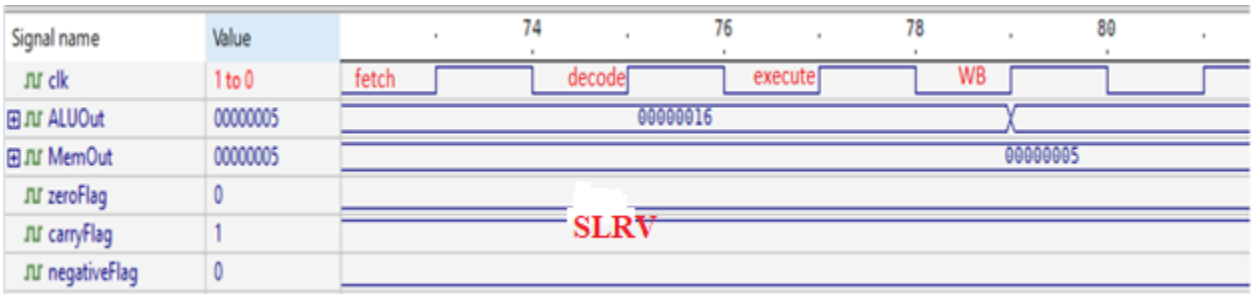


*Figure 49:SLRV waveform result*