
Faculty of Engineering and Technology

Electrical and Computer Engineering Department

ENCS4370

Computer Architecture

Project #2


| Name | Number |
|---|---|
| Osaid Hamza | 1200875 |
| Mohammad Owda | 1200089 |
| Mahmoud Hamdan | 1201134 |


Instructor: Dr. Ayman Hroub

Section: 3

Date: 12-7-2023

# 1. Design and Implementation

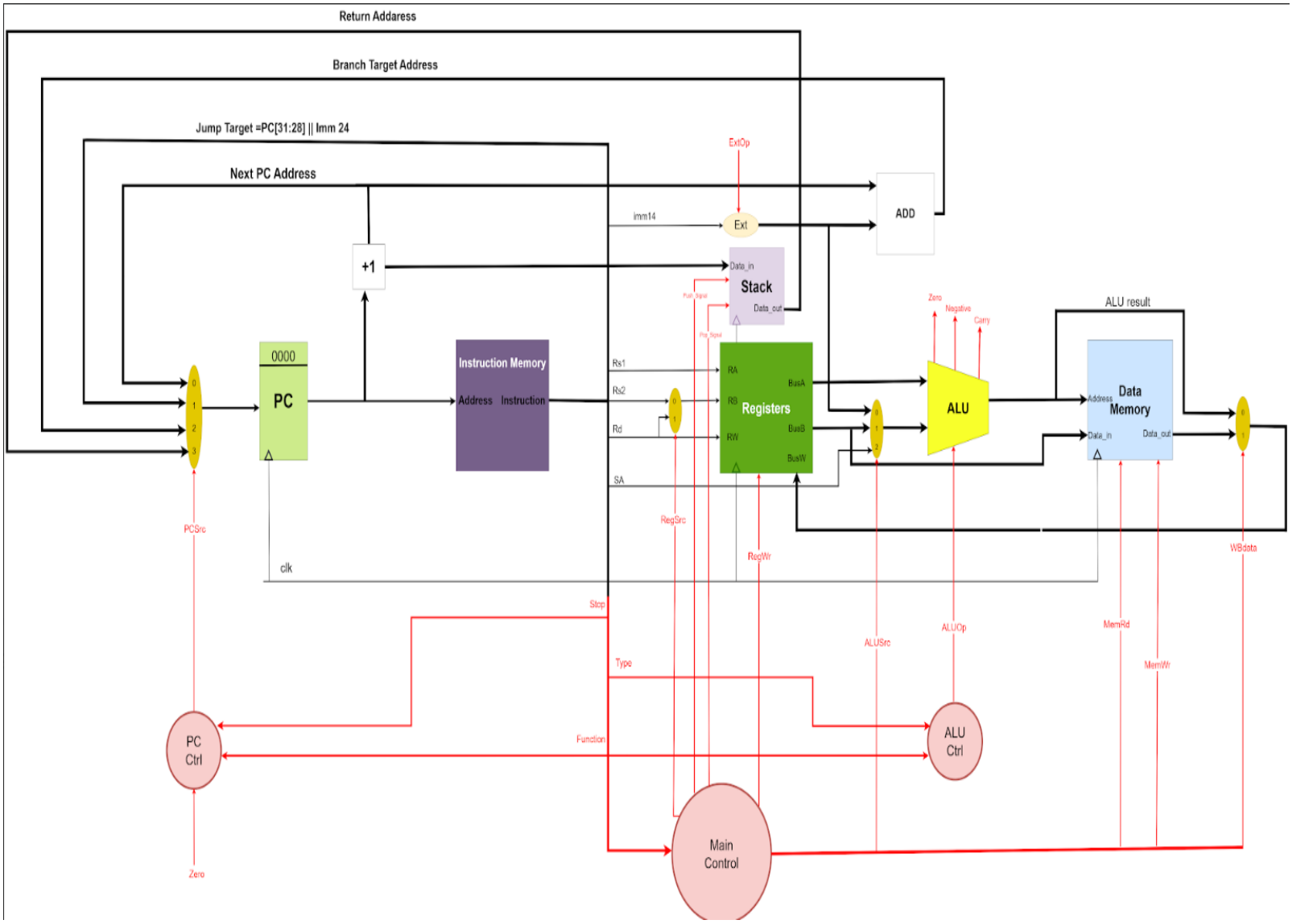



*Figure 1:Data Path*

### 1.1 PC (Program Counter)

The value of the Program Counter (PC), which indicates the memory address of the next instruction to be executed, is determined by this unit. The unit takes into account certain generated operations, such as BEQ, J, JAL.

- The next address for normal operations is PC + 4.
- The next address for BEQ (Branch if equal) operation is PC + 4 * $imm^{14}$.
- The next address for J (jump) operation PC + $imm^{24}$.
- The next address for JAL (Jump and link) operation is PC + $imm^{24}$, and the return address is stored in the stack, which is PC + 4.

➔A multiplexer was used with it's control signal (PCSrc) to determine the next address of the PC.

PC Control Truth Table

| Op | Type | Zero Signal | PCSrc |
|---|---|---|---|
| All-ops | R-Type | X | 0 = Increment PC |
| All-ops | S-Type | X | 0 = Increment PC |
| All-ops | J-Type | X | 1 = Jump Target Address |
| BEQ | I-Type | 0 | 0 = Increment PC |
| BEQ | I-Type | 1 | 2 = Branch Target Address |
| ANDI | I-Type | X | 0 = Increment PC |
| ADDI | I-Type | X | 0 = Increment PC |
| LW | I-Type | X | 0 = Increment PC |
| SW | I-Type | X | 0 = Increment PC |

### 1.2. Register File

The register file is a key component of a computer system that stores data in registers and allows for fast access to this data during execution. The register file takes in three 3-bit inputs, which are used to select the registers to read from, and the BusW input bus for writing a register, also the clock, and the RegWr signals.

The first input (Rs1)  and the second input can be either (Rs2 or Rd) depending on the instruction being executed, select the two source registers to read data from, while the third input (Rd) is used to select the destination register to write data back to .

The BusW input is used to write data back to the destination register. The RegWr signal is used to enable writing to the register file. When the RegWr signal is 1, the register file is enabled to write data back to the destination register.

The register file has two outputs, BusA and BusB, which provide the values read from the selected source registers. These values can be used by other components in the computer system to perform calculations or other operations.

### 1.3 Data Memory

The data memory is a component of a computer system that stores data in memory locations. It takes as inputs a read enable signal (MemRd), a write enable signal (MemWr), a 32-bit memory address, and a 32-bit data input. It produces a 32-bit output, which is the data stored at the specified memory address.

When the MemRd signal is 1, the data memory outputs the contents of the memory location specified by the address input. When the MemWr  signal is 1, the data memory writes the data input to the memory location specified by the address input. The write operation is performed on the rising edge of the clock signal.

### 1.4 ALU

The ALU is the unit that performs the arithmetic operations in the computer system, like addition, subtraction and logical operators like and operator. ALU takes two operands as an input sand the ALUOp to specify the arithmetic operation to be performed between these two operands. This unit has 4 outputs which are the result, zero flag, negative flag, and carry flag.

When the ALUOp is 000, then the operation is the logical (and) operator, and when ALUOp is 001, then the operation is addition. However, 010 represents subtraction, 011 is the comparison, 100 is for the shift logical right and 101 is the shift logical right.

### 1.5 Extender

Extender unit is used to extend the immediate to 32 bits, because the ALU takes only 32 bits operand. This unit extends the immediate according to the sign bit which is the most left bit.

### 1.6 Instruction memory

The instruction memory is a basic functional unit in the computer system data path. This unit contains the list of instructions to be executed through the data path, by taking the address of the instruction as an input, and know the necessary information needed to be used in the next stages of the data path, like address of the source and destination registers, the immediate value, the type of the instruction and the stop bit.

### 1.7 Control Unit

The control unit is used to determine the values of the control signals that are in the data path, each instruction has a specific values to these signals, which are performed and generated by the control unit, according to the function code and instruction type. The following table shows the values of the control signals that are generated by the control unit, these signals are for the all instructions that are handled by the data path.

### 1.8 Stack

The stack is a data structure which is used to store the current instruction address, and used when an instruction like JAL (Jump and link) is going to be executed. Any instruction that has the stop bit assigned to 1, it then call the return address from stack to return to this address after exiting from the function.

## Main Control Truth Table

| OP | RegSrc | RegWr | ExtOp | ALUSrc | MemRd | MemWr | WBdata |
|---|---|---|---|---|---|---|---|
| | | | | **R-Type** | | | |
| **AND** | 0:Rs2 | 1 | X | 1: Busb | 0 | 0 | 0:ALU result |
| **ADD** | 0:Rs2 | 1 | X | 1: Busb | 0 | 0 | 0:ALU result |
| **SUB** | 0:Rs2 | 1 | X | 1: Busb | 0 | 0 | 0:ALU result |
| **CMP** | 0:Rs2 | 0 | X | 1: Busb | 0 | 0 | X |
| | | | | **I-Type** | | | |
| **ANDI** | X | 1 | 1:sign | 0: Imm | 0 | 0 | 0:ALU result |
| **ADDI** | X | 1 | 1:sign | 0: Imm | 0 | 0 | 0:ALU result |
| **LW** | X | 1 | 1:sign | 0: Imm | 1 | 0 | 1:Mem |
| **SW** | X | 0 | 1:sign | 0: Imm | 0 | 1 | X |
| **BEQ** | 1:Rd | 0 | 1:sign | 0:Busb | 0 | 0 | X |
| | | | | **J-Type** | | | |
| **J** | X | 0 | X | X | 0 | 0 | X |
| **JAL** | X | 0 | X | X | 0 | 0 | X |
| | | | | **S-Type** | | | |
| **SLL** | X | 1 | X | 2:SA | 0 | 0 | 0:ALU result |
| **SLR** | X | 1 | X | 2:SA | 0 | 0 | 0:ALU result |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **SLLV** | 0:Rs2 | 1 | X | 1: Busb | 0 | 0 | 0:ALU result |
| **SLRV** | 0:Rs2 | 1 | X | 1: Busb | 0 | 0 | 0:ALU result |

X is a don't care (can be 0 or 1), used to minimize logic.

The following table shows the generation of the control signals that are used by the ALU.

## ALU Control Truth Table

| Op | Type | ALUOp | 3-bit Coding |
|---|---|---|---|
| **AND** | 00:R-Type | AND | 000 |
| **ADD** | 00:R-Type | ADD | 001 |
| **SUB** | 00:R-Type | SUB | 010 |
| **CMP** | 00:R-Type | CMP | 011 |
| **ANDI** | 10:I-Type | AND | 000 |
| **ADDI** | 10:I-Type | ADD | 001 |
| **LW** | 10:I-Type | ADD | 001 |
| **SW** | 10:I-Type | ADD | 001 |
| **BEQ** | 10:I-Type | SUB | 010 |
| **J** | 01:J-Type | X | X |
| **JAL** | 01:J-Type | X | X |
| **SLL** | 11:S-Type | SLL | 100 |
| **SLR** | 11:S-Type | SLR | 101 |
| **SLLV** | 11:S-Type | SLL | 100 |
| **SLRV** | 11:S-Type | SLR | 101 |

The following table shows the generation of the control signals that are used by the stack.

## Stack Truth Table

| Op | Type | Stop bit | Pop Signal | Push Signal |
|---|---|---|---|---|
| All-ops | All-Types | 1 | 1 | 0 |
| All-ops(Except JAL) | All-Types | 0 | 0 | 0 |
| JAL | J-Type | 0 | 0 | 1 |

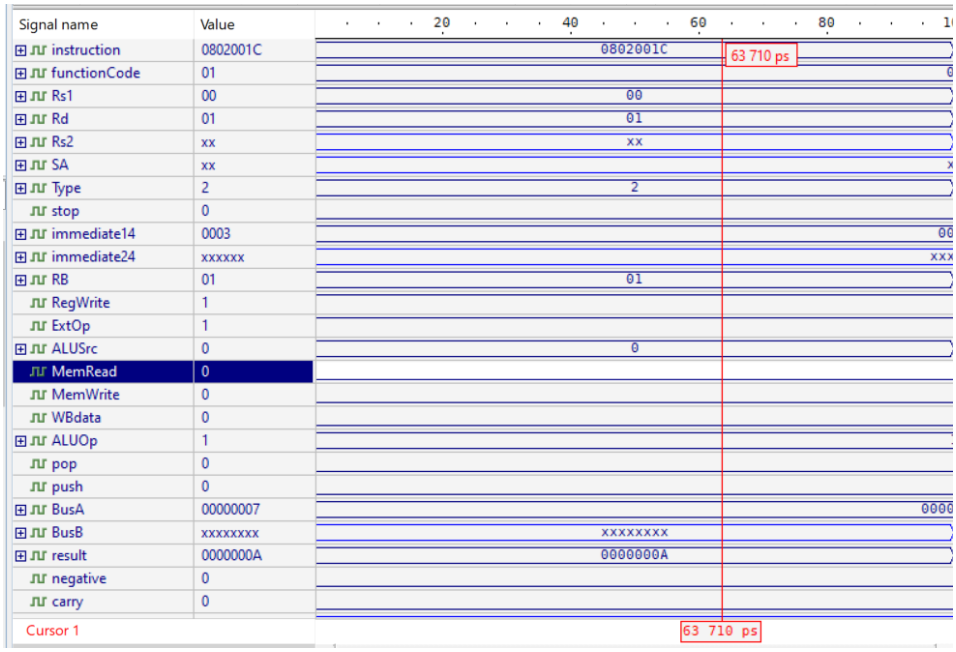
The following table shows the effect of the main control signals on the data path when apply them by the values 0 or 1 or 2.

## Main Control Signals

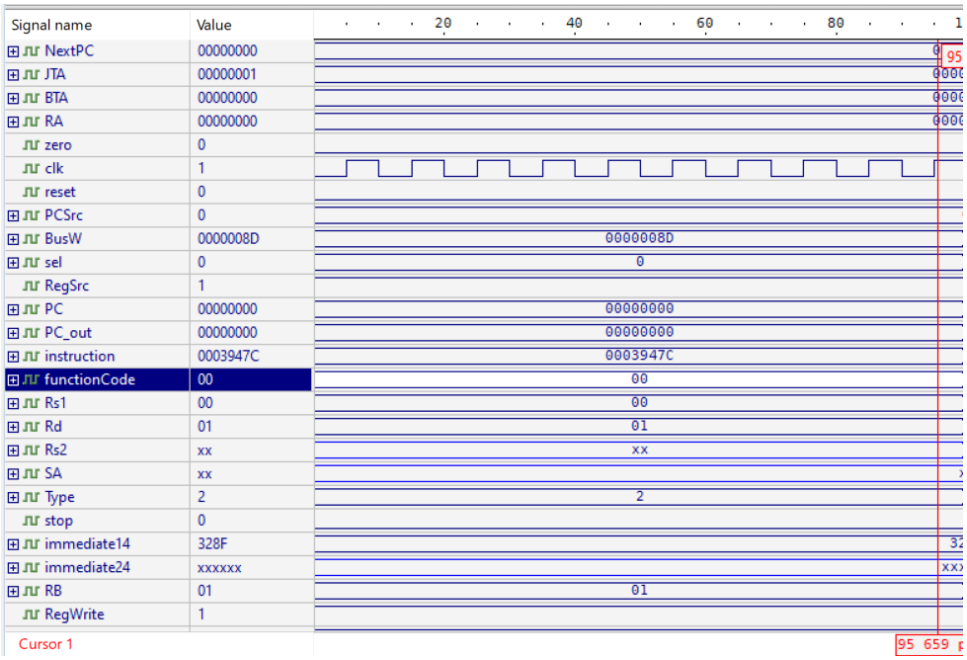
| Signal | Effect when '0' | Effect when '1' | Effect when '2' |
|---|---|---|---|
| **RegWr** | No register is written. | Destination register (Rd) is written with the data on BusW. | |
| **ExtOp** | 14-bit immediate is zero-extended | 14-bit immediate is sign-extended. | |
| **ALUSrc** | Second ALU operand is the value of the extended 14-bit immediate. | Second ALU operand is the value of register (Rs2 or Rd) that appears on BusB. | Second ALU operand is the value SA that appears on BusB. |
| **MemRd** | Data memory is NOT read. | Data memory is read<br>Data_out ← Memory[address]. | |
| **MemWr** | Data Memory is NOT written. | Data memory is written<br>Memory[address] ← Data_in. | |
| **WBdata** | BusW = ALU result | BusW = Data_out from Memory | |
| **PushSignal** | Stack is NOT pushed | Stack is pushed | |
| **PopSignal** | Stack is NOT poped | Stack is poped | |

## Simulation and Testing

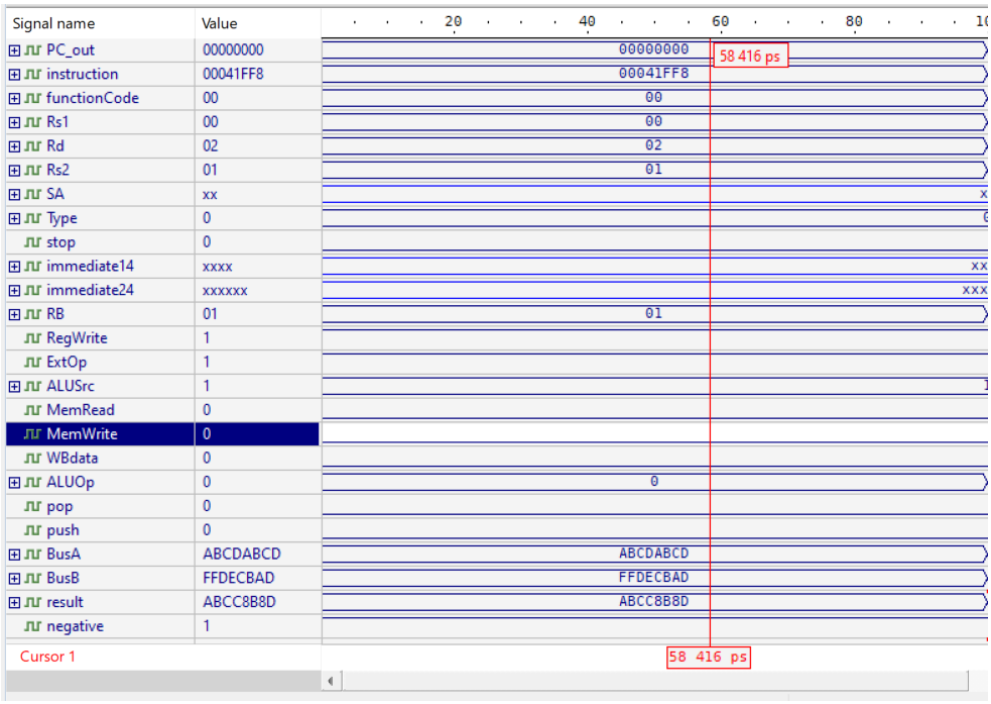
The aim of testing is to ensure correctness of the data path. Multiple instructions were applied to our data path and the results were observed, as the following figures show.



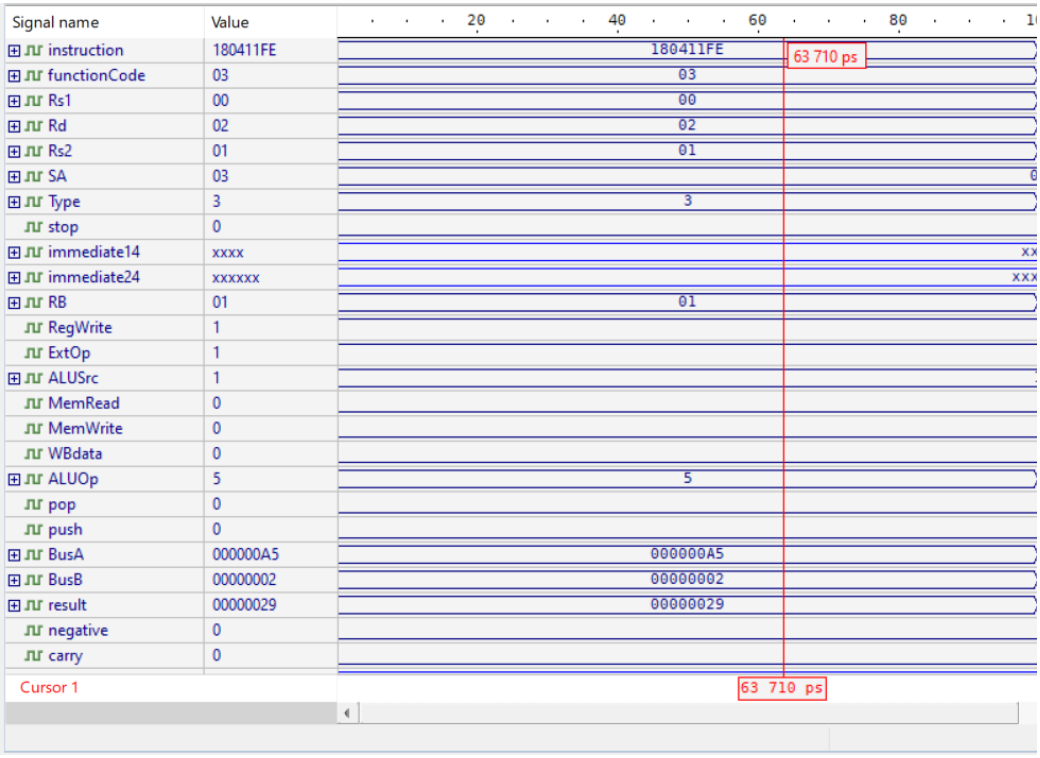

Testing of addition instruction (ADDI), Rd = Rs1 + Imm14 = 7 + 3 = 10 (32'hA)




Testing of and instruction (ANDI), Rd = Rs1 && Imm14.

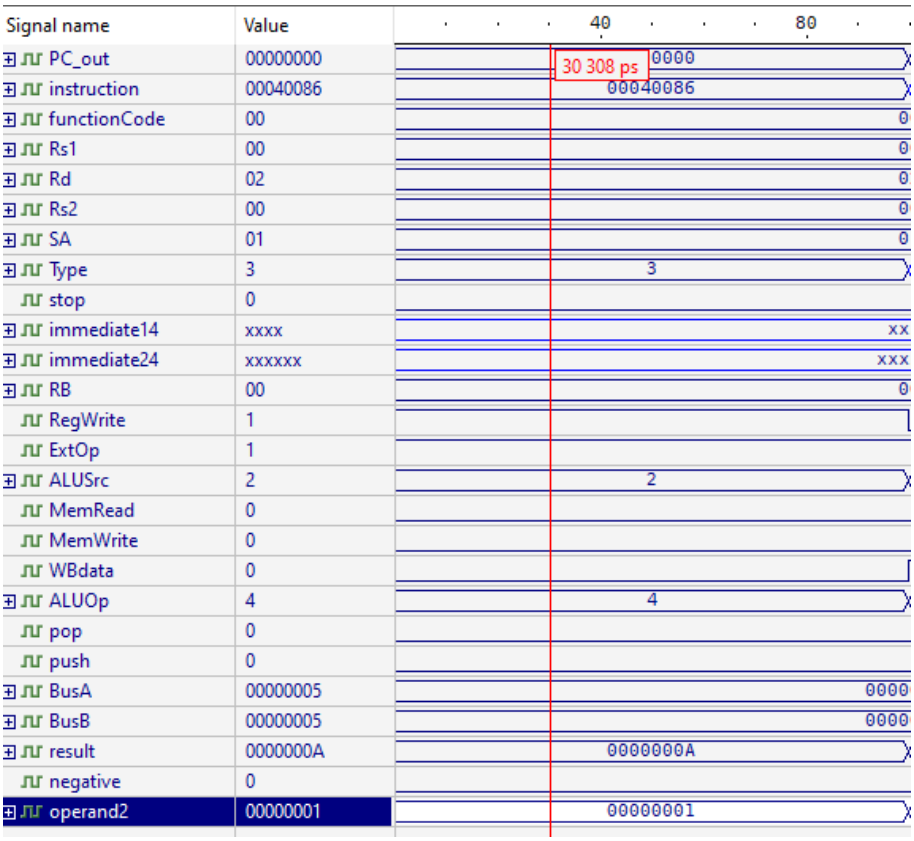



Testing of and instruction (AND), Rd = Rs1 && Rs2.

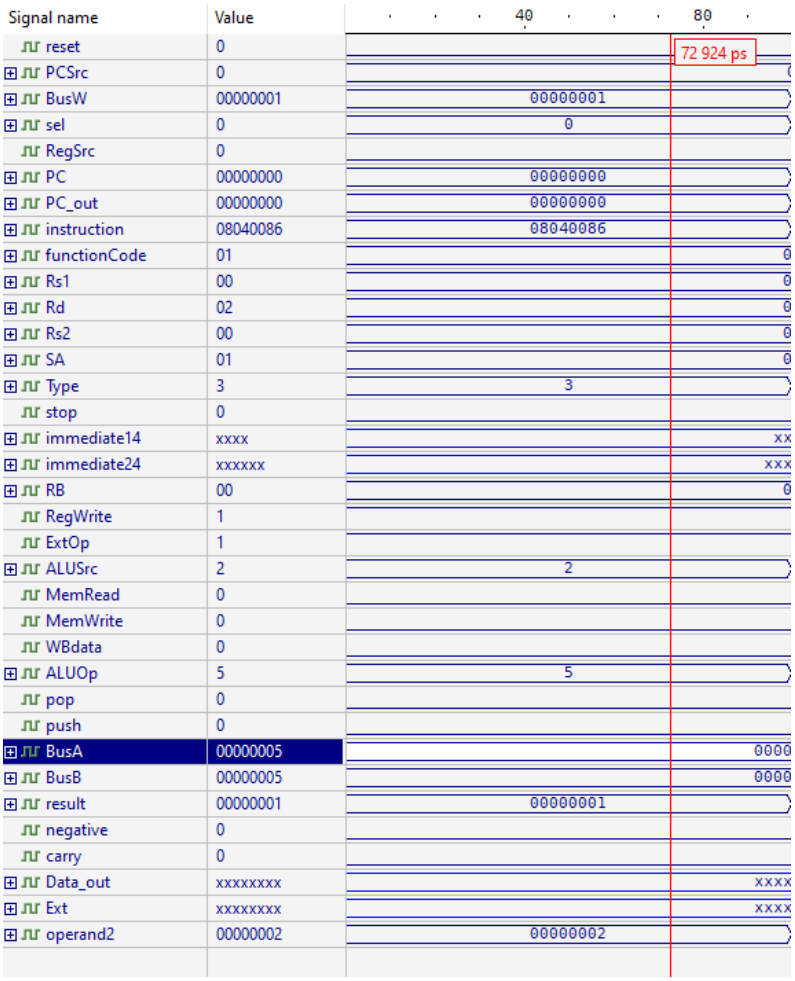



Testing of shift right instruction (SLRV) Rd = Rs1 >> Rs2.

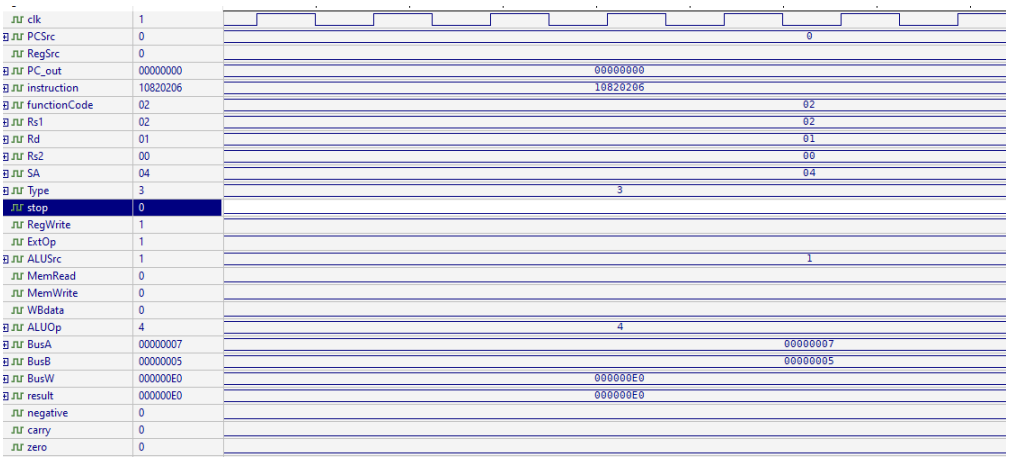
In the figure above, Rs1 = A5, Rs2 = 2.




Testing of shift left logical instruction (SLL), Rd = Rs1 << SA.




Testing of shift logical right (SLR), Rd = Rs1 >> SA.




Testing of shift right instruction (SLLV) Rd = Rs1 << Rs2.

## Multi-cycle implementation

The following figures show the implementation of multi-cycle by splitting the single-cycle's stages into IF (instruction fetch) which contains PC and instruction memory, ID (instruction decode) which contains the register file and control unit, EX (execution) which is handled in the ALU, M (Memory) which is handled in the data memory and W (Write back) which is the output data which came from ALU or data memory, to write it on the register file. Each stage of those five stages must done in one clock cycle.

```
////////////////////////////////////////////////////////////////////////////////////////////
//IF

Mux4to1 pcreg(.d0(NextPC),.d1(JTA),.d2(BTA),.d3(RA),.sel(PCSrc),.y(PC_out));//mux before PC

always @(posedge clk) begin
    if(enIF == 1)    begin
        PC = PC_out;
    end

end

//PC_selector PC1(.PC_in(PC_out),.output1(PC));


InstructionMemory  im(.address(PC), .instruction(instruction), .functionCode(functionCode),
                      .Rs1(Rs1), .Rs2(Rs2), .Rd(Rd), .Type(Type), .stop(stop),
                      .immediate14(immediate14), .immediate24(immediate24), .SA(SA));


////////////////////////////////////////////////////////////////////////////////////////////
```

The figure above shows the Instruction fetch.

```
////////////////////////////////////////////////////////////////////////////////////////////
// ID

// increment the pc after fetch the instruction
    always @(posedge clk) begin
        if (enID == 1)begin
            NextPC = PC + 1;
        end
    end

Mux2to1  src(.d0(Rs2),.d1(Rd),.sel(RegSrc),.y(RB)); //mux before REgister File


// stack with stack pointer inside it
Stack st(.en(enID), .clk(clk), .push(push), .pop(pop), .data_in(NextPC), .data_out(RA));

// control unit
ControlUnit  control(
.enEX(enEX),
.en(enID),
.instr_type(Type),
.funct(functionCode),
.stop(stop),
.zero(zero),
.RegSrc(RegSrc),
.RegWrite(RegWrite),
.ExtOp(ExtOp),
.ALUSrc(ALUSrc),
.MemRead(MemRead),
.MemWrite(MemWrite),
.WBdata(WBdata),
.PCSrc(PCSrc),
.ALUOp(ALUOp),
```

```
    .PCSrc(PCSrc),
    .ALUOp(ALUOp),
    .pop(pop),
    .push(push)

    );


    // register file
    Register_File  RF (
        .enWB(enWB),
        .en(enID),
        .clk(clk),
        .reset(reset),
        .RegWr(RegWrite),
        .RA(Rs1),
        .RB(RB),
        .RW(Rd),
        .BusW(BusW),
        .BusA(BusA),
        .BusB(BusB)
    );

    Extender  ex(immediate14,ExtOp,Ext);//extender

    Extender24  ex2(immediate24,ExtOp,Ext24);//extender

    Extender5  ex3(SA,ExtSA);//extender5

    adder_32bit add(.a(NextPC),.b(Ext),.sum(BTA));

    adder_32bit add2(.a(NextPC),.b(Ext24),.sum(JTA));
```

The figures above show the Instruction decode.

```
////////////////////////////////////////////////////////////////////////////////////////////////////

// EX

Mux3to1  mu(Ext,BusB,ExtSA,ALUSrc,operand2); // mux before ALU


ALU  uutt (
    .en(enEX),
    .operand1(BusA),
    .operand2(operand2),
    .ALUOp(ALUOp),
    .result(result),
    .zero(zero),
    .negative(negative),
    .carry(carry)
);

////////////////////////////////////////////////////////////////////////////////////////////////////
```

The figure above shows the Execute stage.

```
    ////////////////////////////////////////////////////////////////////////////////////////////////

    // MEM
    data_memory DM (
        .en(enMEM),
        .clk(clk),
        .reset(reset),
        .Address(result),
        .Data_in(BusB),
        .MemRd(MemRead),
        .MemWr(MemWrite),
        .Data_out(Data_out)
    );


    Mux2to1 n(.d0(result),.d1(Data_out),.sel(WBdata),.y(BusW)); //last mux




endmodule

////////////////////////////////////////////////////////////////////////////////////////////////////
```

The figure above shows the Memory stage.

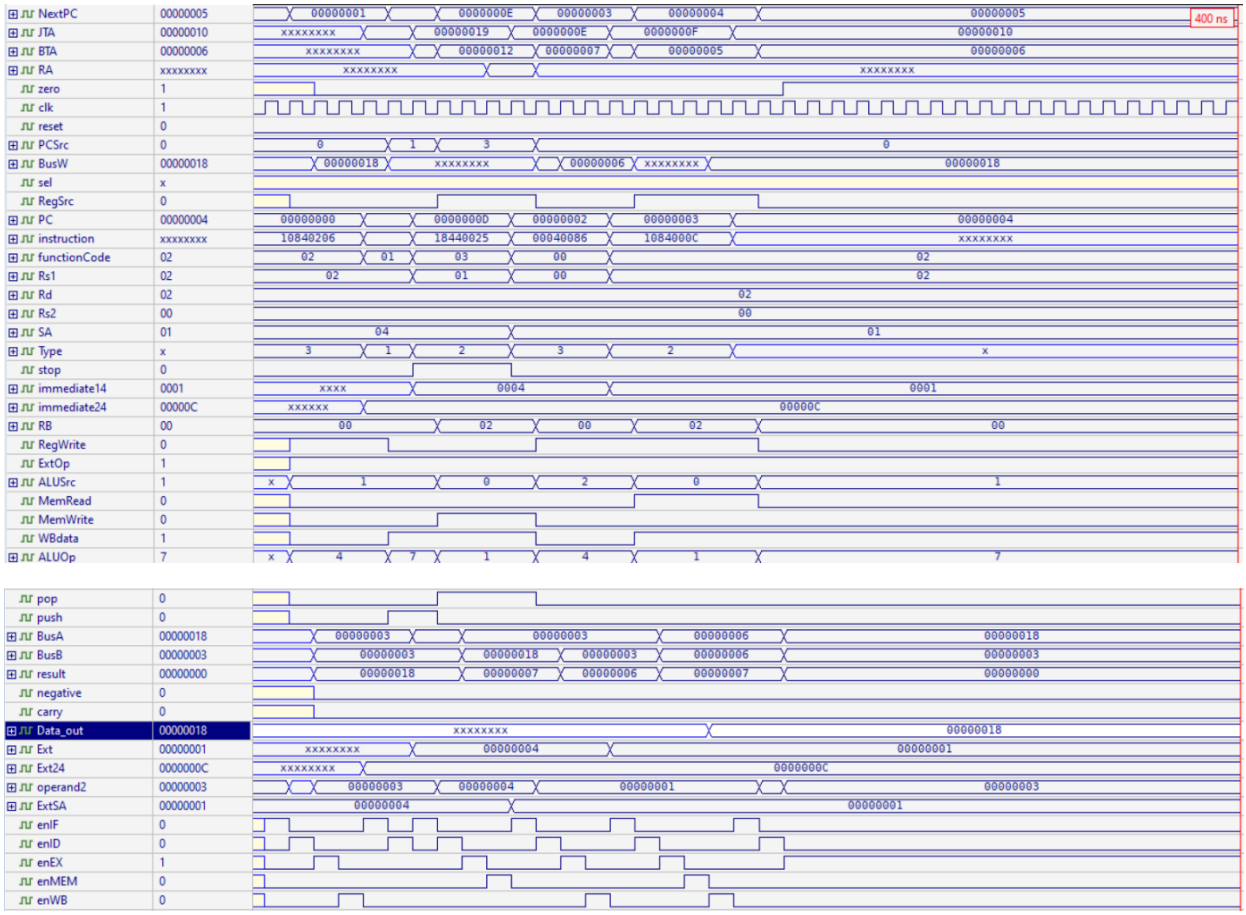
```
instruction_memory[0] = 32'h10840206; // SLLV R2, R1, R0,
instruction_memory[1] = 32'h08000062; //JAL #12
instruction_memory[2] = 32'h00040086;//SLL R1, R0, #1
instruction_memory[3] = 32'h1084000C;//lw R2, 1(R2)

instruction_memory[9] = 32'h0802001C; //ADDI
instruction_memory[10] = 32'h10820206;//SLLV
instruction_memory[11] = 32'h08040086;//SLR
instruction_memory[12] = 32'h00040086;//SLL R1, R0, #1

instruction_memory[13] = 32'h18440025;//sw R1, 4(R2)     stop=1
```

The figure above shows the instructions to be entered to the data path, which are stored in the instruction memory.

The figures below show the execution for the instructions above in the multi-cycle data path.




It has been noticed that each stage took only one clock cycle, and the instructions executed as expected.

A testbench was used to test the whole path, as the following figures show.

```
module stage_1_tb;

  wire [31:0] NextPC;
  reg [31:0] JTA;
  wire [31:0] BTA;
  reg [31:0] RA;

  wire zero;
  reg clk;
  reg reset;
  wire  [1:0] PCSrc;
  reg [31:0]BusW;
  reg sel;
  reg RegSrc;
  wire [31:0] PC,PC_out,instruction;
  wire [4:0] functionCode;
  wire [4:0] Rs1;
  wire [4:0] Rd;
  wire [4:0] Rs2;
  wire [4:0] SA;
  wire [1:0] Type;
  reg stop;
  wire [13:0] immediate14;
  wire [23:0] immediate24;


  wire [4:0] RB;
  wire RegWrite;
  wire ExtOp;
  wire [1:0] ALUSrc;
  wire MemRead;
  wire MemWrite;
  wire WBdata;
```

```
  wire [2:0] ALUOp;
  wire pop;
  wire push;

  wire [31:0]BusA;
  wire [31:0]BusB;

  wire [31:0]result;

  reg negative;
  reg carry;
  wire [31:0] Data_out;
  wire [31:0] Ext, Ext24;
  wire [31:0] operand2;
  wire [31:0] ExtSA;
  // Instantiate the module under test

  CPU ut (
    .sel(sel),
    .PCSrc(PCSrc),
    .clk(clk),
    .reset(reset),
    .instruction(instruction),
    .PC(PC),
    .PC_out(PC_out),
    .functionCode(functionCode),
    .Rs1(Rs1),
    .Rs2(Rs2),
    .Rd(Rd),
    .Type(Type),
    .immediate14(immediate14),
    .immediate24(immediate24),
    .SA(SA) ,
    .ExtSA(ExtSA),
```

```
    .RB(RB),
    .stop(stop),
    .zero(zero),
    .RegSrc(RegSrc),
    .RegWrite(RegWrite),
    .ExtOp(ExtOp),
    .ALUSrc(ALUSrc),
    .MemRead(MemRead),
    .MemWrite(MemWrite),
    .WBdata(WBdata),
    .ALUOp(ALUOp),
    .pop(pop),
    .push(push),
    .BusA(BusA),
    .BusB(BusB),
    .BusW(BusW),


    .operand1(BusA),
    .operand2(operand2),
    .result(result),
    .negative(negative),
    .carry(carry),
    .Data_out(Data_out) ,
    .Ext(Ext),
    .Ext24(Ext24),
    .NextPC(NextPC),
    .JTA(JTA),
    .BTA(BTA),
    .RA(RA)
  );
```

```
  initial begin
    // Initialize inputs
    clk = 1'b0;
    reset = 1'b0;
//  NextPC=32'h0;
    //BTA = 32'h0;
    //JTA=32'h1;
    // Generate clock
    forever begin
      #5 clk = ~clk;
    end

  end

  // Test Scenario
  initial begin

  //  #10 NextPC = NextPC + 32'h1; // Reset signal

//  #20 NextPC = NextPC + 32'h1; // Reset signal
  end

    // Finish testing



  // Monitor outputs
  initial begin
    $monitor($time, "  instruction=%h PC_out=%h functionCode=%h Rs1=%h Rs2=%h Rd=%h Type=%h stop=%b immediate14=%h immediate24=%h SA=%h\n",
             instruction, PC_out, functionCode, Rs1, Rs2, Rd, Type, stop, immediate14, immediate24, SA);
  $monitor($time, "funct=$b Type=%b  RegSrc=%b RegWrite=%b ExtOp=%b ALUSrc=%b MemRead=%b MemWrite=%b WBdata=%b stop=%b PCSrc=%b ALUOp=%b pop=%b push=%b\n",
            functionCode,Type,RegSrc, RegWrite, ExtOp, ALUSrc, MemRead, MemWrite, WBdata, stop, PCSrc, ALUOp, pop,push);

    #400 $finish;
  end
endmodule
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

## Team Work

Three of us implemented the data path, and then each one of us implement a specific functional unit. Osaid implemented the data memory and the register file, mohammad implemented the ALU and stack and mahmoud implemented instruction memory and the PC. For the simulation part, all of us took part in it, such that each member took number of instructions to test it on the data path.