VIETNAM NATIONLA UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF INFORMATION TECHNOLOGY
FACULTY OF COMPUTER ENGINEERING
---oOo---

DIGITAL SYSTEM DESIGN WITH HDL
CE213.O12.MTCL.2

LAB REPORT 5

VIETNAMESE NAME : THIẾT KẾ DATAPATH ĐƠN GIẢN

ENGLISH NAME: Designing a Simple DATAPATH

STUDENT NAME: Trương Duy Đức
STUDENT ID: 21521970
Lecturer: Tạ Trí Đức

Contents

I. Content of Lab 3
1.1. Objectives 3
1.2. Practice content 3
II.THEORETICAL BASIS 3
1. Datapath 3
III.PRACTICAL 5
1. CODE 7
1.1 CODE Datapath 7
1.2. CODE SRAM (DATA MEMORY) 9
1.3. CODE REGFILE 9
1.4. CODE ALU 11
1.4. SIGN_EXTEND 13
1.5. CODE MUX 13
1.6 CODE TESTBENCH 14
2. Simulation 16
2.1.Waveform 16
2.2 Transcipt 16
3. Schematic 16
IV. EVALUATION 17

Pictures

Figure 1: Data path in accordance with the MIPS architecture 3
Figure 2: "Datapath" image of a processor with 8 MIPS instructions: add, sub, AND, OR, slt, lw, sw, and beq. 4
Figure 3: Datapath's Output 5
Figure 4:The bold (red) lines represent the active paths during the execution of the ADD instruction. 6
Figure 5:The bold (red) lines represent the active paths during the execution of the lw instruction. 6
Figure 6:The bold (red) lines represent the active paths during the execution of the sw instruction. 7
Figure 7: Waveform 16
Figure 8: Trancript 16
Figure 9: Schematic 17

I. Content of Lab

1.1. Objectives

Using the Verilog HDL language, design a simple DATAPATH following the MIPS architecture.

1.2. Practice content

Based on the theory of the MIPS architecture's DATAPATH that students have studied in the Computer Architecture course, as shown in Figure 1, and utilizing the modules designed in previous labs, students design a DATAPATH to execute the following instructions using the Verilog HDL:

- add $1, $2, $3
- lw $1, 0($2)
- sw $1, 0($2)"

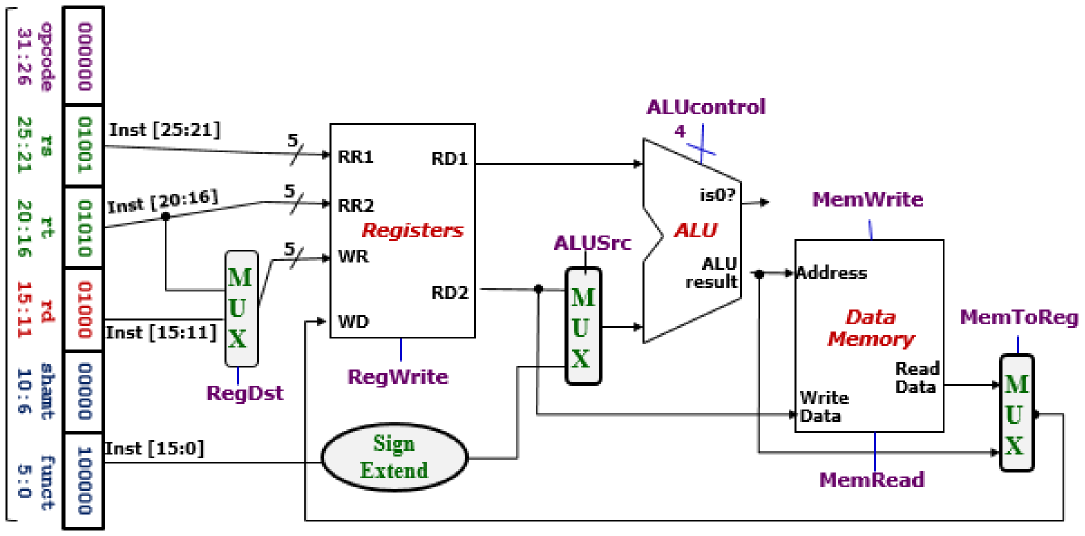


Figure 1: Data path in accordance with the MIPS architecture

II.THEORETICAL BASIS

1. Datapath

"Datapath" is a concept in computer architecture that describes the path along which data travels within a system. It typically includes components and modules responsible for executing operations and processing data.

In a computer system, the datapath is responsible for performing operations, retrieving data from memory, and transferring data between different components of the processor. This often involves registers, memory, and units that perform logical and arithmetic operations.

In the MIPS computer architecture, the datapath includes components such as registers, the Arithmetic Logic Unit (ALU), memory, and the interconnections between them to execute machine instructions. The datapath is usually designed to support a range of specific machine instructions defined in that computer architecture.

When a machine instruction is executed, the ALU is often where the computation takes place. It takes input from registers or memory, performs the required operation, and then produces the result. The ALU is designed to work with binary data and carry out operations quickly and efficiently.

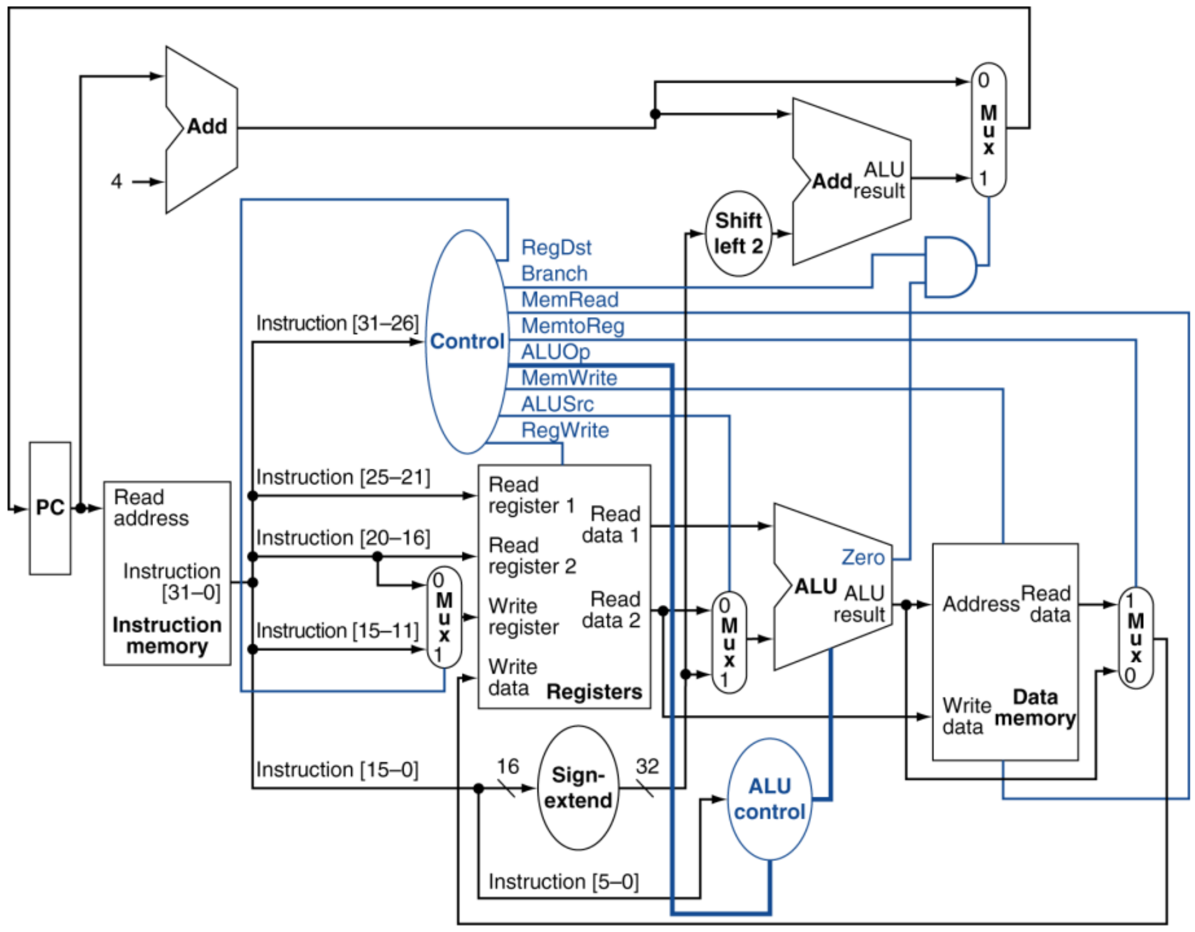


Figure 2: "Datapath" image of a processor with 8 MIPS instructions: add, sub, AND, OR, slt, lw, sw, and beq.

III.PRACTICAL

We will connect the modules from previous labs to form a datapath block and use this block to execute the following instructions

- add $1, $2, $3
- lw $1, 0($2)
- sw $1, 0($2)"

The datapath output will depend on the instruction it belongs to. If it is an ADD instruction, the output will be the result of the ALU. For lw or sw instructions, the output will take the value of the Read Data from the Data Memory block. Based on this, we will place the output after the mux block as shown in the figure below.

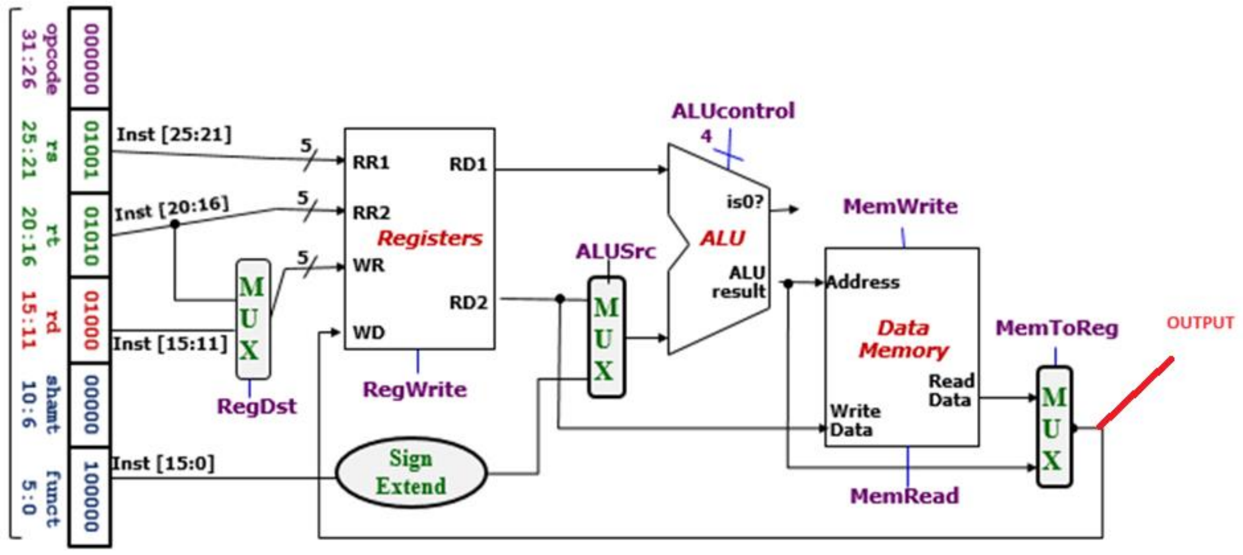


Figure 3: Datapath's Output

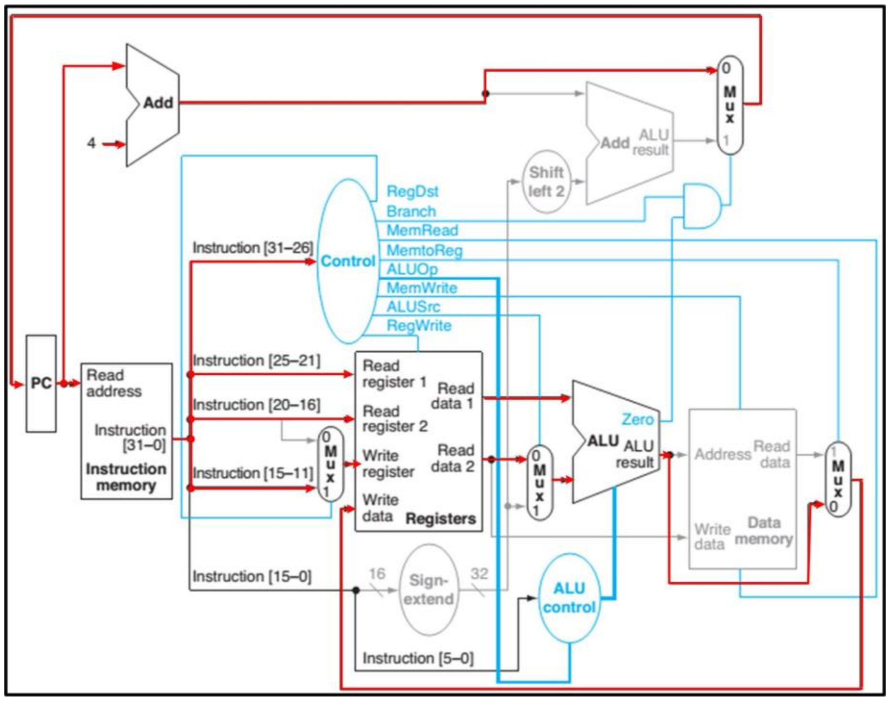


Figure 4:The bold (red) lines represent the active paths during the execution of the ADD instruction.

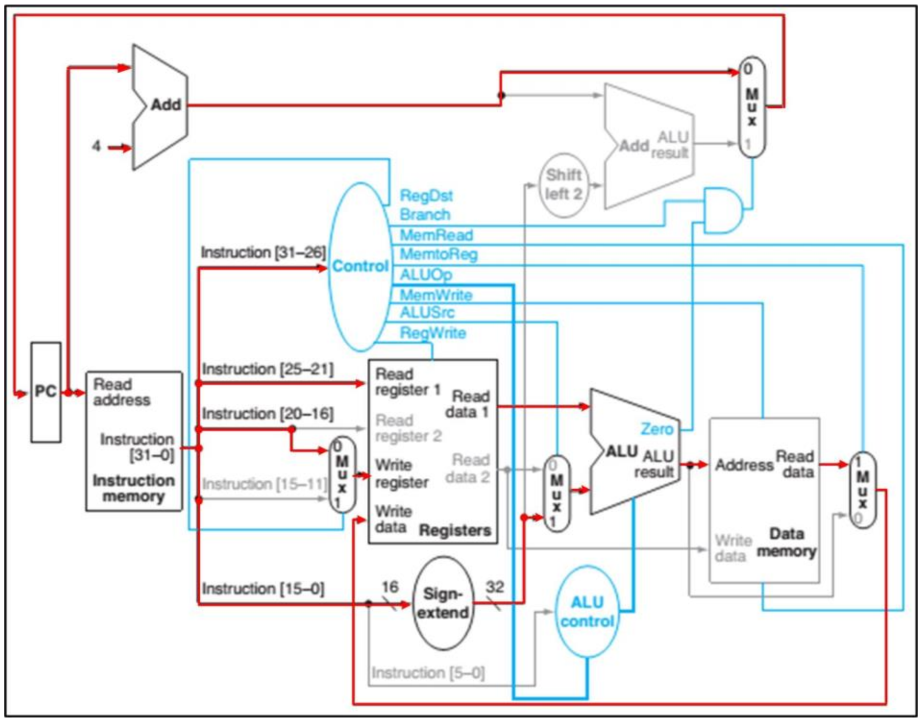


Figure 5:The bold (red) lines represent the active paths during the execution of the lw instruction.

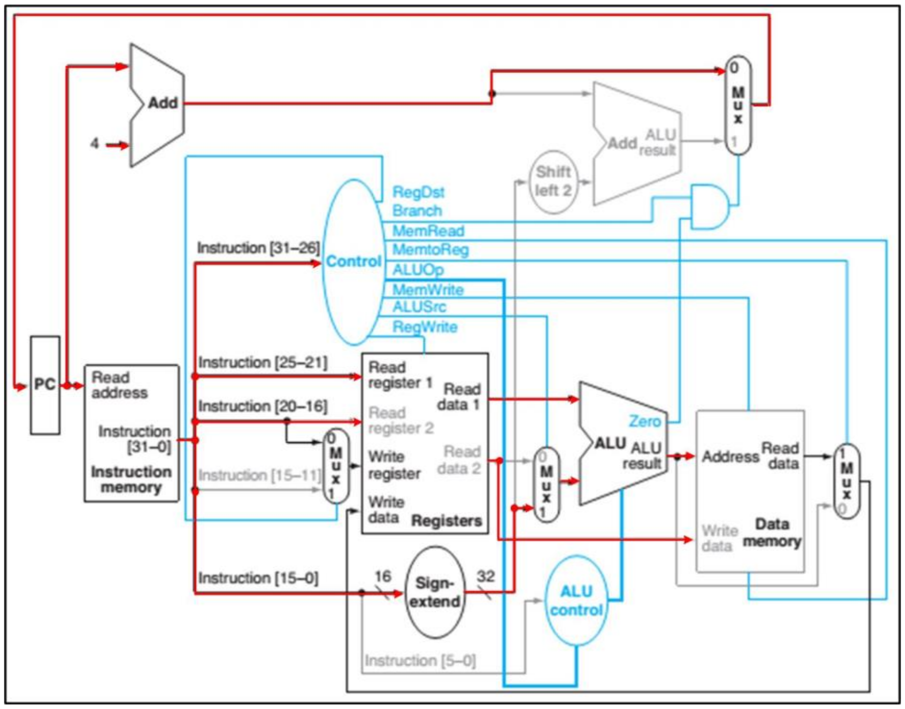


Figure 6:The bold (red) lines represent the active paths during the execution of the sw instruction.

Operation	RegDST	RegWrite	ALUScr	ALUOp	MemWrite	MemRead	MemToReg	JToPc
ADD	1	1	0	101	0	0	0	0
LW	0	1	1	101	0	1	1	0
SW	X	0	1	101	1	0	X	0

Control Signal Table

1. CODE

1.1 CODE Datapath

```
module
DATAPATH(ALURESULT,RD1,RD2,CLK,INST,REGDST,ALUSRC,ALUCONTRO
L,MEMWRITE,MEMREAD,MEMTOREG,OUT,REGWRITE);
input [25:0] INST;
output [31:0] OUT,RD1,RD2,ALURESULT;
input CLK;
input REGDST,ALUSRC,MEMWRITE,MEMREAD,MEMTOREG,REGWRITE;
input [2:0] ALUCONTROL;
wire [31:0] RD1,RD2,ALURESULT,O_SIGNEXTEND,READDATA,WD;
wire [4:0] WR;
```

```
MUX_32bit #(.WIDTH(5))
mux1(.OP(REGDST),.A(INST[15:11]),.B(INST[20:16]),.O(WR));

REGFILE
REG1(.RWE(REGWRITE),.WA(WR),.WD(WD),.RA1(INST[25:21]),.RA2(INST[20:
16]),.CLK(CLK),.RD1(RD1),.RD2(RD2));

wire [31:0] OM2;
MUX_32bit #(.WIDTH(32))
mux2(.OP(ALUSRC),.A(O_SIGNEXTEND),.B(RD2),.O(OM2));

ALU ALU1(.A(OM2),.B(RD1),.O(ALURESULT),.OP(ALUCONTROL),.OF());

SRAM
SRAM1(.CLK(CLK),.WE(MEMWRITE),.RE(MEMREAD),.WD(RD2),.RA(READD
ATA),.ADDRESS(ALURESULT));

SIGN_EXTEND SignEXTEND(.I(INST[15:0]),.O(O_SIGNEXTEND));

MUX_32bit #(.WIDTH(32))
mux3(.OP(MEMTOREG),.A(READDATA),.B(ALURESULT),.O(WD));

assign OUT=WD;
endmodule
```

1.2. CODE SRAM (DATA MEMORY)

- **Purpose:** Data Memory serves as a storage location for data during the execution of machine instructions. It is commonly used to store data from and to the processor.
- **Operation:** When instructions like **lw** (load word) or **sw** (store word) are encountered, Data Memory acts as the space for accessing and storing data. In the case of **lw**, it provides a value from memory to a register. In the case of **sw**, it stores a value from a register into memory.

```
module SRAM(CLK,WE,RE,WD,RA,ADDRESS);
input [31:0] ADDRESS;
//reg [31:0] SRAM [2**32-1:0];
reg [31:0] SRAM [128:0];

input [31:0] WD;//WA:WriteData
input WE,RE,CLK;
output reg [31:0] RA;//RA:ReadData

initial begin
SRAM[2]=32'd10;
end
always @(posedge CLK) begin
if (RE && ! WE) begin
        RA<=SRAM[ADDRESS];
        end
else if (WE && !RE) begin
        SRAM[ADDRESS]<= WD;
        end
else RA<=32'bz;
end
endmodule
```

1.3. CODE REGFILE

- **Purpose:** Registers are temporary storage locations for values and results of operations. Registers are often used to store temporary data during the execution of instructions.
- **Operation:** When instructions perform operations, the values from registers are used as inputs for these operations. For example, the **add $1, $2, $3** instruction adds the values in registers **$2** and **$3** and stores the result in register **$1**.

```
module REGFILE(RWE,WA,WD,RA1,RA2,CLK,RD1,RD2);
input RWE,CLK;
input [31:0] WD;
input [4:0] WA,RA1,RA2;
//RWE: read write enable
//WA: write address
//WD: write data
//RA1,RA2: read address


output reg [31:0] RD1,RD2;
reg [31:0] REG32X32 [31:0]; // reg [size] name [size]

always @(posedge CLK) begin

if(RWE)begin
REG32X32[WA] = WD;//wd write data
RD1 = REG32X32[RA1];
RD2 = REG32X32[RA2];
end
else begin
 RD1 = 32'bz;
 RD2 = 32'Bz;
 end

end
endmodule
```

1.4. CODE ALU

In the datapath of a computer, the Arithmetic Logic Unit (ALU) primarily functions to perform arithmetic and logic operations. Here is a description of the roles of the ALU:

1. **Arithmetic Operations:**
   - The ALU executes basic arithmetic operations such as addition, subtraction, multiplication, and division.
   - Example: In the instruction **add $1, $2, $3**, the ALU performs the addition operation between the values in registers **$2** and **$3**, and the result is stored in register **$1**.
2. **Logic Operations:**
   - The ALU performs logic operations such as AND, OR, NOT, and XOR.
   - Example: In the instruction **and $1, $2, $3**, the ALU performs the AND operation between the values in registers **$2** and **$3**, and the result is stored in register **$1**.
3. **Comparison:**
   - The ALU can execute comparison operations such as equality (**beq**), greater than (**bgt**), less than (**blt**), etc.
   - Example: In the instruction **beq $1, $2, label**, the ALU compares the values in registers **$1** and **$2**, and if they are equal, the computer will branch to the specified label.

```
module ALU #(parameter WIDTH = 32) (A,B,O,OP,OF);
input [WIDTH-1:0] A,B;
input [2:0] OP;
output reg [WIDTH-1:0] O;
output reg OF;
always @(*)
    case(OP)
    3'b000: begin
                    O=~A;
                    OF=1'b0;
            end
    3'b001: begin
                    O=A&B;
                    OF=1'b0;
            end
```

```
        3'b010: begin
                        O=A^B;//xor
                        OF=1'b0;
                end
        3'b011:  begin
                        O=A|B;//or
                        OF=1'b0;
                end
        3'b100: begin
                        O=A-1;
                        if (A[31]==1 && O[31]==0) OF=1'b1;
                        else  OF=1'b0;
                end
        3'b101: begin
                        O=A+B;
                        if (A[31]==0 && B[31]==0 && O[31]==1)OF=1'b1;
                        else if (A[31]==1&& B[31]==1 && O[31]==0)OF=1'b1;
                        else OF=1'b0;
                end
        3'b110: begin
                        O=A-B;
                        if (A[31]==0 && B[31]==1 && O[31]==1)OF=1'b1;
                        else if(A[31]==1&& O[31]==0)OF=1'b1;
                        else OF=1'b0;
                end
        3'b111: begin
                        O=A+1;
                        if (A[31]==0&& O[31]==1)OF=1'b1;
                        else OF=1'b0;
                end
        endcase

    endmodule
```

1.4. SIGN_EXTEND

The SIGN_EXTEND block functions based on the most significant bit (MSB) to add bits. For example, if the MSB is 1, it will extend with additional n bits, all set to 1. In the following code, it extends the input of 16 bits to 32 bits based on the MSB (this can be adjusted by changing the parameter)."

```
module SIGN_EXTEND(I,O);
parameter WIDTH = 16;
input [WIDTH-1:0] I;
output [WIDTH*2-1:0] O;
//assign O=(I[WIDTH-1]==0)?{16'b0000000000000000,I}:{16'b1111111111111111,I};
assign O = {{16{I[WIDTH-1]}}, I};
endmodule
```

1.5. CODE MUX

A 2-to-1 Multiplexer (MUX) is a digital circuit device that functions to select between two inputs and outputs a single output based on a select signal. The Mux has additional parameters to accommodate various sizes.

```
module MUX_32bit #(parameter WIDTH = 32) (OP,A,B,O);

input [WIDTH-1:0] A,B;
input OP;
output [WIDTH-1:0] O;

assign O=(OP==1)?A:B;

endmodule
```

1.6 CODE TESTBENCH

```
`timescale 1ns/1ps

module DATAPATH_tb;

  reg CLK;
  reg [25:0] INST;
  reg REGDST, ALUSRC, MEMWRITE, MEMREAD, MEMTOREG, REGWRITE;
  wire [31:0] OUT;
  integer COUNT;
  DATAPATH DUT(
    .CLK(CLK),
    .INST(INST),
    .REGDST(REGDST),
    .ALUSRC(ALUSRC),
    .MEMWRITE(MEMWRITE),
    .MEMREAD(MEMREAD),
    .ALUCONTROL(4'b0101),
    .MEMTOREG(MEMTOREG),
    .REGWRITE(REGWRITE),
    .OUT(OUT)
  );

  initial begin
    CLK = 0; REGDST = 1; ALUSRC = 0; MEMWRITE = 0; MEMREAD = 0;
MEMTOREG = 0; REGWRITE = 1;

    // Test case 1: R-type instruction (e.g., ADD)
        COUNT=0;
    INST = 32'b00010_00011_00001_00000000000;  // ADD $1, $2, $3
    #20;

    // Test case 2: lw instruction
        COUNT=1;
        CLK = 0; REGDST = 1; ALUSRC = 1; MEMWRITE = 0; MEMREAD = 1;
MEMTOREG = 1; REGWRITE = 1;
    INST = 32'b00010_00001_00000_00000000000;  // lw $1, 0($2)
    #20;
```

```
   // Test case 3: sw instruction
         COUNT=2;
         CLK = 0; REGDST = 1'bx; ALUSRC = 1; MEMWRITE = 1; MEMREAD = 0;
MEMTOREG = 1/*X*/; REGWRITE = 0;
   INST = 32'b00010_00001_00000_00000000000;  // sw $1, 0($2)
   #20;

   $stop;
  end

  always #10 begin
   CLK = ~CLK;
         if (COUNT==0)
         #1 $display("ADD $1,$2,$3 <===> OUTPUT: %d",OUT);
         else if (COUNT==1)
         #1 $display("LW $1,0($2) <===> OUTPUT: %d",OUT);
         else if (COUNT==1)
         #1 $display("SW $1,0($2) <===> OUTPUT: %d",OUT);
  end

endmodule
```

In the 2 modules REGFILE and SRAM, add 2 initialization blocks for easier testing

REGFILE:

```
initial begin
REG32X32[5'b00010]= 32'd2;
REG32X32[5'b00011]= 32'd3;
end
```

SRAM:

```
initial begin
SRAM[32'd2]=32'd10;
end
```

2. Simulation

2.1.Waveform

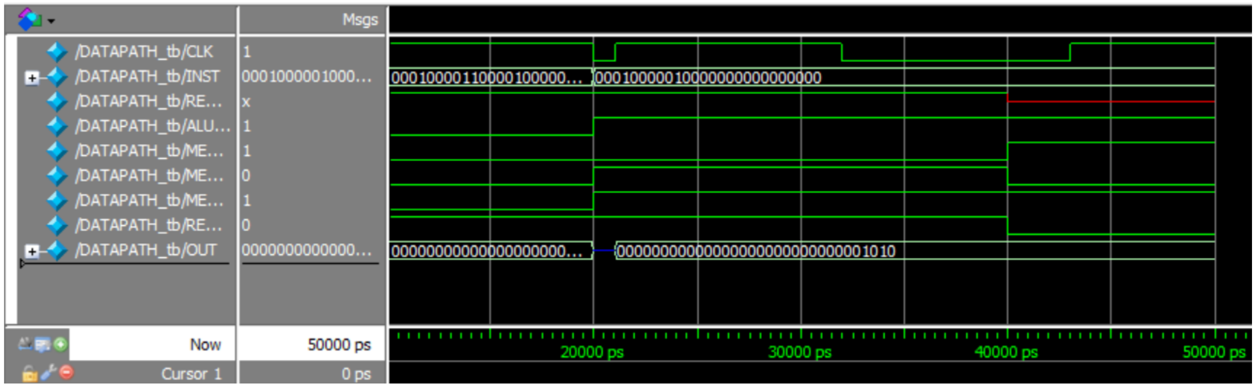
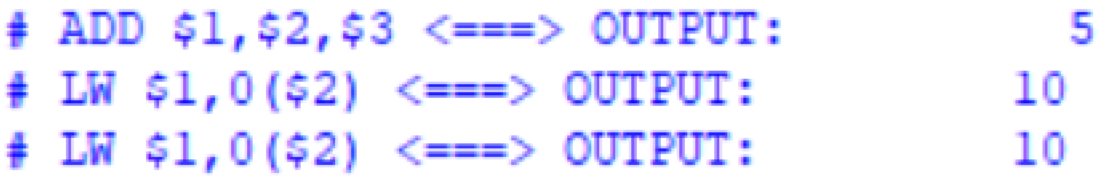


Figure 7: Waveform

2.2 Transcipt



```
# ADD $1,$2,$3 <===> OUTPUT:          5
# LW $1,0($2) <===> OUTPUT:          10
# LW $1,0($2) <===> OUTPUT:          10
```

Figure 8: Trancript

Explanation of code and output:

In test **case 1**, the instruction **add $1, $2, $3** is executed, which means the values in registers $2 and $3 are added, and the result is stored in the REGISTER file of register $1. Since the values in registers $2 and $3 are 2 and 3 respectively, $1 will hold the value 5 (REG[1] will have a value of 5).

In test **case 2**, the instruction **lw $1, 0($2)** is executed, which means the value is loaded from the memory address obtained by adding 0 to the value in register $2, and this value is stored in register $1. Since SRAM[2] is 32'd10, the value in register $1 will be 10.

In test **case 3,** the instruction **sw $1, 0($2)** is executed, which means the value in register $1 is stored in the memory address obtained by adding 0 to the value in register $2. Similarly, after performing the operation, the value of $1 (specifically, 10, as modified in test case 2) will be stored in SRAM[2].

3. Schematic

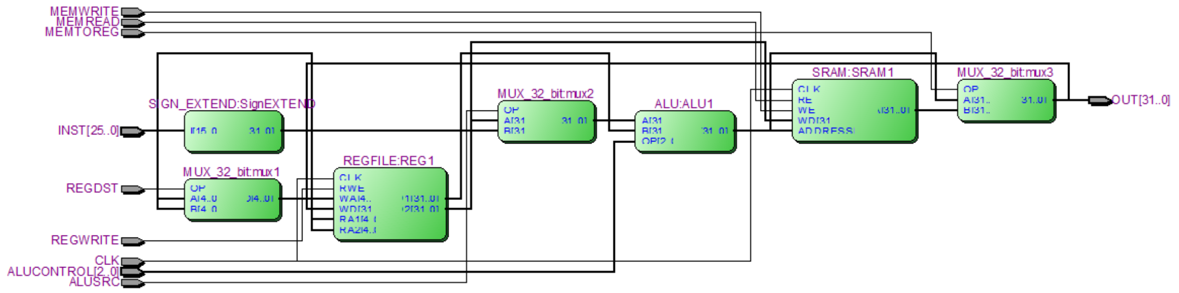


Figure 9: Schematic

IV. EVALUATION

Correct completion of lab requirements, circuits and results as predicted

Error:

Warning (13046): Tri-state node(s) do not directly drive top-level pin(s)

Reason: The MUX module in Quartus does not support logical operations like AND, OR, etc., when the data is Z (high impedance state), so it will automatically interpret it as X (undefined value).