Name: Abdullah Khan

Roll No. : 111801001

Course Name: CS2160 - Computer Organization Lab

**POST LAB REPORT - 3**horizontal line

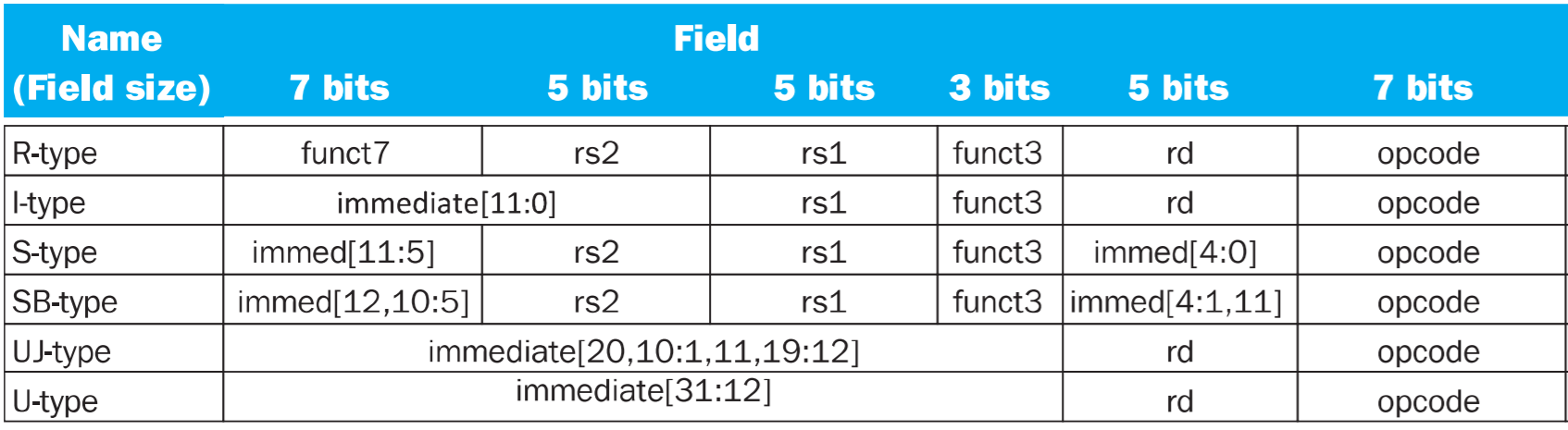
# **Exp : Datapath for RISC-V ISA** Completed on: 25 Feb ’20

**Problem Statement**

Design and implement the data path for various instructions in the RISC-V ISA including R-type ( **ADD**, **SUB**, **AND**, **OR** ), I-type ( **ADD\_I**, **XOR\_I**, **SLL\_I**, **SRL\_I**, **SRA\_I**, **LD**, **JALR** ), S-type ( **SD** ), SB-type ( **BEQ**, **BNE**, **BLT**, **BGE** ) and UJ-type ( **JAL** ).

**Theory**

**Instruction Formats :** Different types of instructions in RISC-V have different fields and formats. They are classified on the basis of **OPCODE** which is unique for a particular type of instruction. All these things are summarized in the following table:



**Control Unit :** the control unit is a very important part of our implementation as it decides which module will function at a given time. We may not need to write back to our register file after every instruction, similarly we may not need to access the memory in every instruction etc. Thus the role of the control unit is to dictate over all modules for proper function of the datapath. The flags used by the control unit are as follows:

**AluFunc** : decides what function the ALU module will perform.

**BSEL** : chooses between the inputs of R-type and I-type to be sent in the ALU.

**WDSEL** : decide to take the output of MEM or ALU.

**MWR** : memory write enable.

**WERF** : Register file write enable.

**PCSEL** : decides how to update program-counter value.

**Type** : “0” -- 32 bit output(sign extended to 64 bits) , “1” -- 64 bit output.

The following table gives an overview of all the flags used in an instruction.

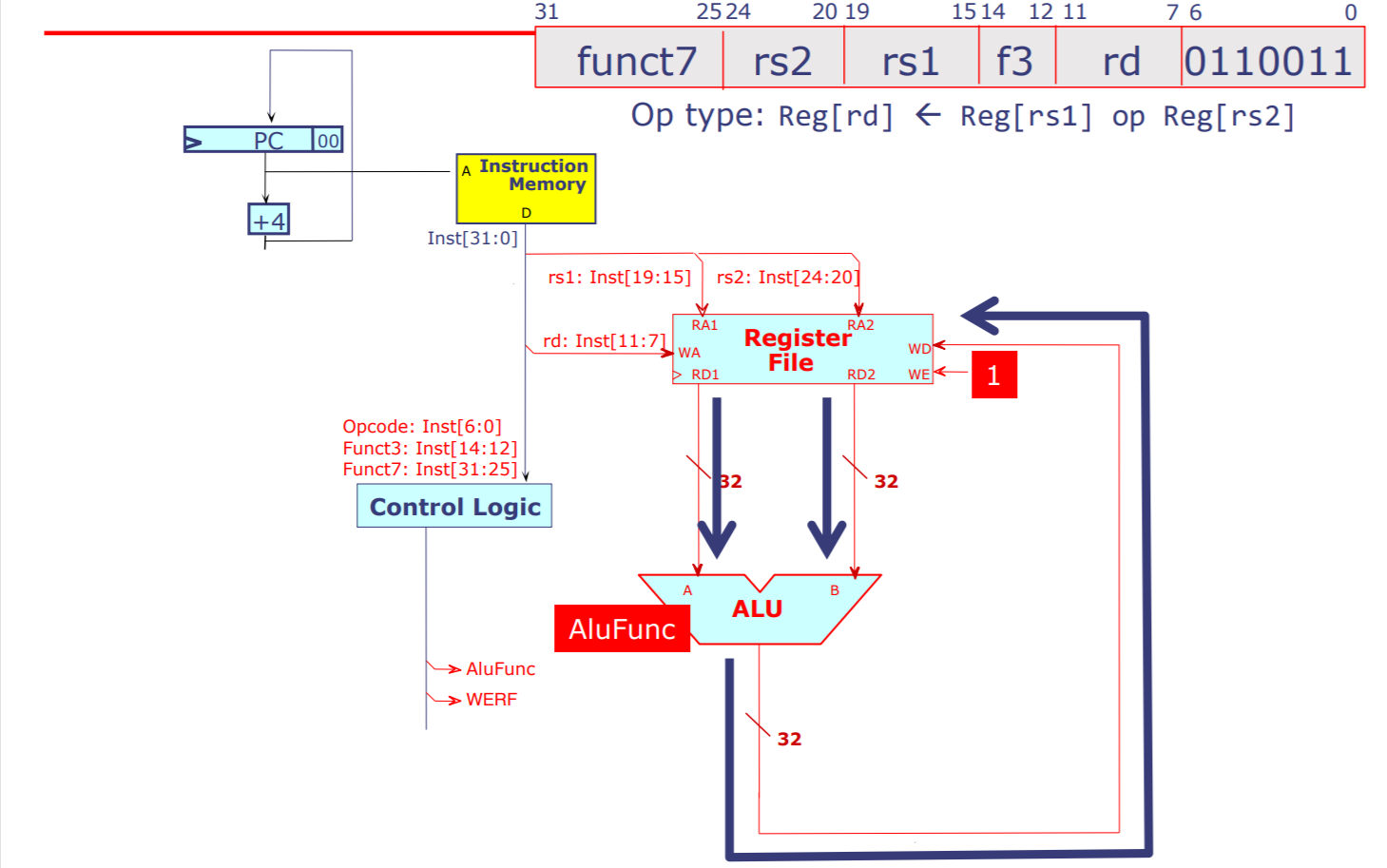
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **R-type** | **I-type** | **LOAD** | **S-type** | **SB-type** | **JALR** | **UJ-type** |
| **AluFunc** | ⨏(op) | ⨏(op) | “+” | “+” | “bool” | “+” | “+” |
| **BSEL** | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| **WDSEL** | 0 | 0 | 1 | 0 | 0 | 2 | 2 |
| **MWR** | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| **WERF** | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| **PCSEL** | 0 | 0 | 0 | 0 | 1 | 2 | 3 |

**R-type Instructions:** takes in 3 register addresses in the register file to take input (rs1 and rs2) and store output ( rd ); along with func7,func3 and opcode fields as an identifier for different types of instructions. The R-type instructions that we are going to implement in our design include ADD, SUB, AND, OR. The following table gives a quick view of all the identifiers of these instructions.

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction** | **Opcode** | **Func3** | **Func7** |
| add | 0110011 | 000 | 0000000 |
| sub | 0110011 | 000 | 0100000 |
| or | 0110011 | 110 | 0000000 |
| and | 0110011 | 111 | 0000000 |

**Datapath for and/or operation:**

In the second cycle after instruction-Fetch, the **DECODER** decodes what type of instruction it is and assign appropriate value to the AluFunc which decides which ALU function to perform. In our implementation AND is assigned an AluCode : 0010 and OR : 0011. Next the rs1 and rs2 values are passed onto the **REG\_FILE** module which returns the values stored in the registers at addresses rs1, rs2 in the same cycle. The value of these registers is passed onto the **ALU** along with the AluFunc where the result is calculated ( i.e bitwise AND, bitwise OR etc ). This result is then passed through a MUX whose use will come later. In the next cycle the result is then written back to the register file.



The following figure shows a basic datapath for R-type instructions. **Note** this figure doesn’t show many components of the datapath that are not being used currently.

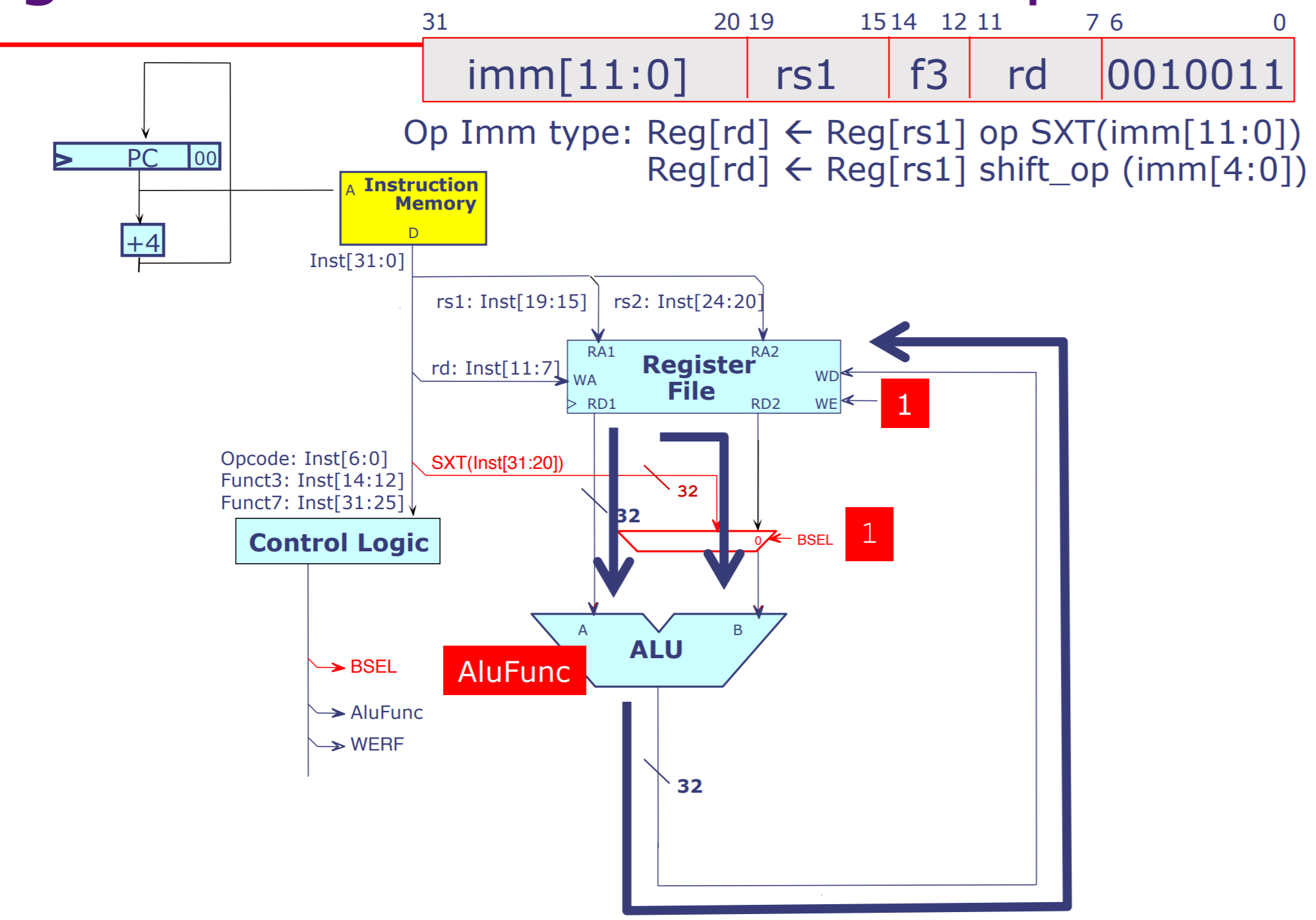
**I-type Instructions:** takes in 2 register addresses in the register file to take input ( rs1 ) and store output ( rd ); and an immediate value of 12 bits in the front of the instruction as an absolute value which is given directly and not from any register. These instructions don’t have a func7 field, rather only func3 and opcode are present. The I-type instructions that we are going to discuss are XOR\_I ( Logical ) , SLL\_I, SRL\_I, SRA\_I ( Shift ), load, . There are other I-type instructions which are discussed later due to their different datapaths.

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction** | **Opcode** | **Func3** | **func6** |
| add\_i | 0010011 | 000 | n.a. |
| xor\_i | 0010011 | 100 | n.a. |
| sll\_i | 0010011 | 001 | 000000 |
| srl\_i | 0010011 | 101 | 000000 |
| sra\_i | 0010011 | 101 | 010000 |
| ld | 0000011 | 011 | n.a. |
| jalr | 1100111 | 000 | n.a. |

**Datapath for xor\_i/shift operation:**

After Instruction-Fetch in the first cycle the DECODER decodes the various fields and identifies the instruction as an I-type. Thus the ALU takes as its second input the SXT(imm) provided by the decoder instead of the value in rs2 register. In the third cycle the ALU executes its operation which may be bitwise XOR or bit-shift (facilitated by the shift operator in verilog). During the fifth cycle the result is written back onto the rd register in the register file.

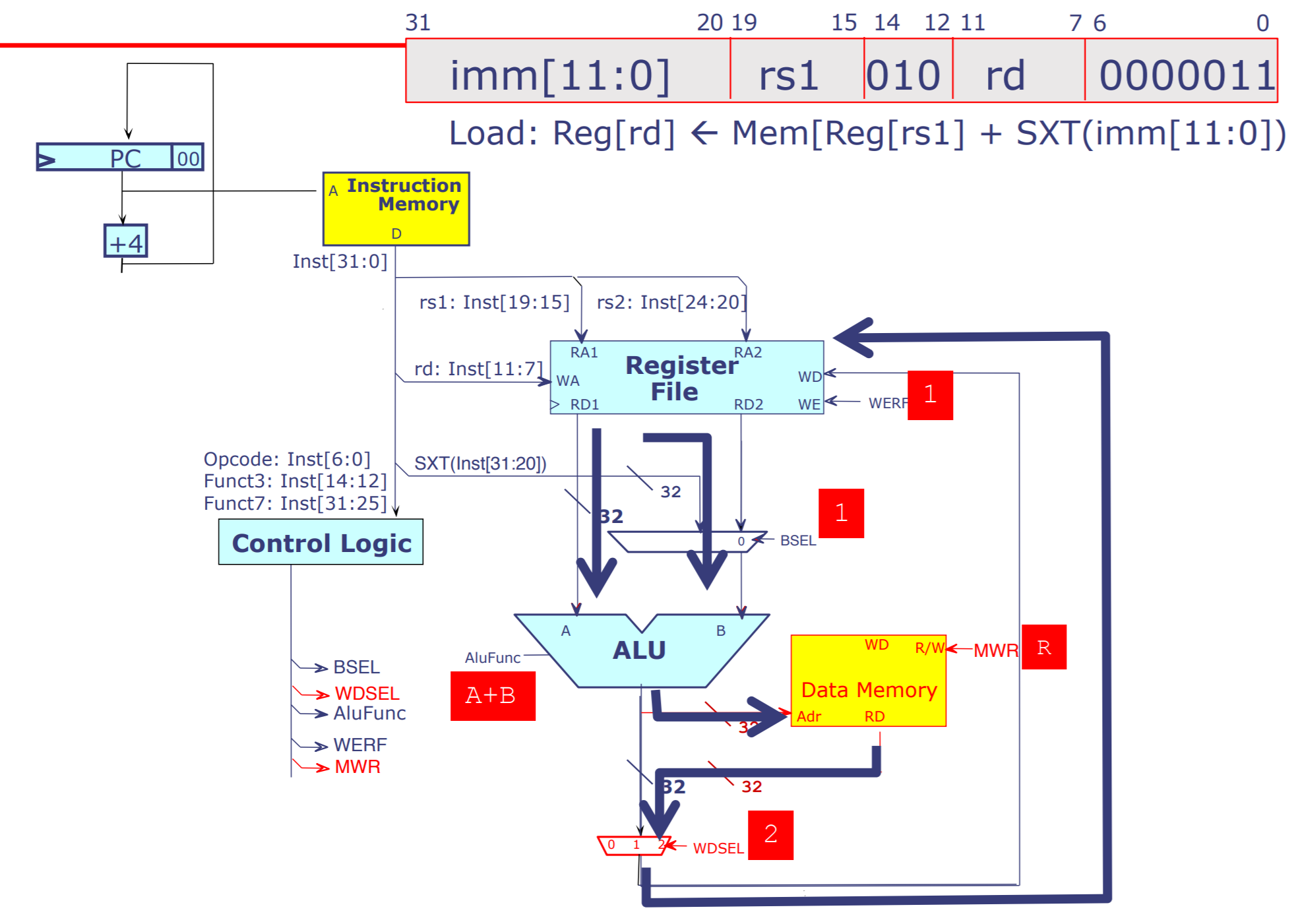
The figure shown below represents the datapath drawn over the block diagram of our implementation; showing only those components that are working presently. The BSEL line becomes “1” which chooses SXT(imm) instead of B(coming from reg-file). WERF also becomes “1” in the fifth cycle allowing us to write back into the register file.



Register-immediate ALU datapath

**Datapath for load operation:**

Load operation needs to access memory for execution and thus, is required to compute an effective memory address. Thus the only difference from the previous datapath is that our ALU result now goes to data memory for accessing the data value.



Load instruction datapath

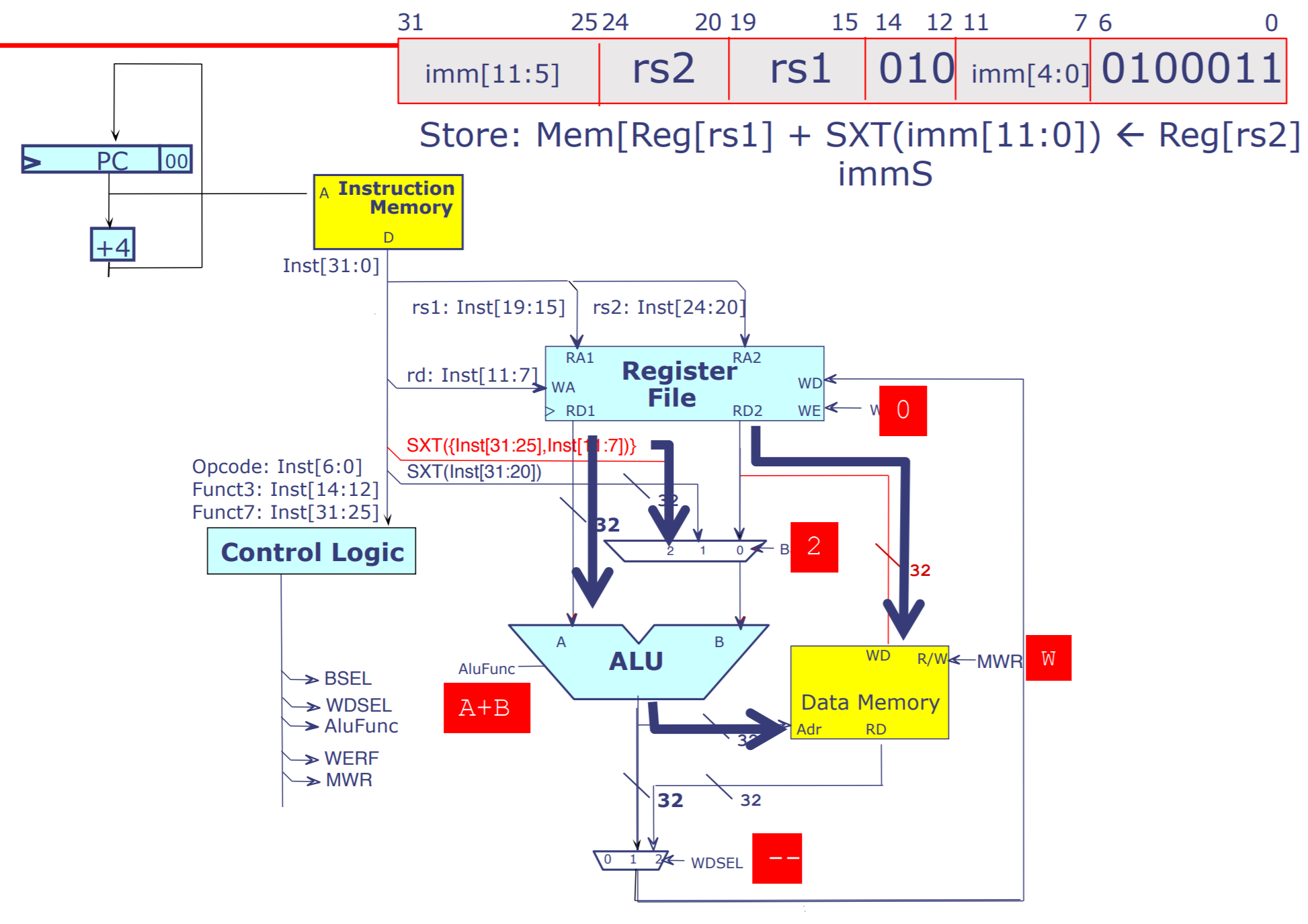
## 

## 

**S-Type instructions :** are a special class of instructions assigned only for store instructions from register file to memory. They include

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Opcode** | **Func3** |
| sw | 0100011 | 010 |
| sd | 0100011 | 111 |

**Datapath for store operation:**

Just like load operations, store op also needs to access the memory to write register value to it. We introduce a new flag Memory write enable to allow us make changes in the memory and write the value from a register to the mem. The ALU calculates the address in memory where change is needed. WREF is put to “0” to stop any unintentional change made to the reg-file. 

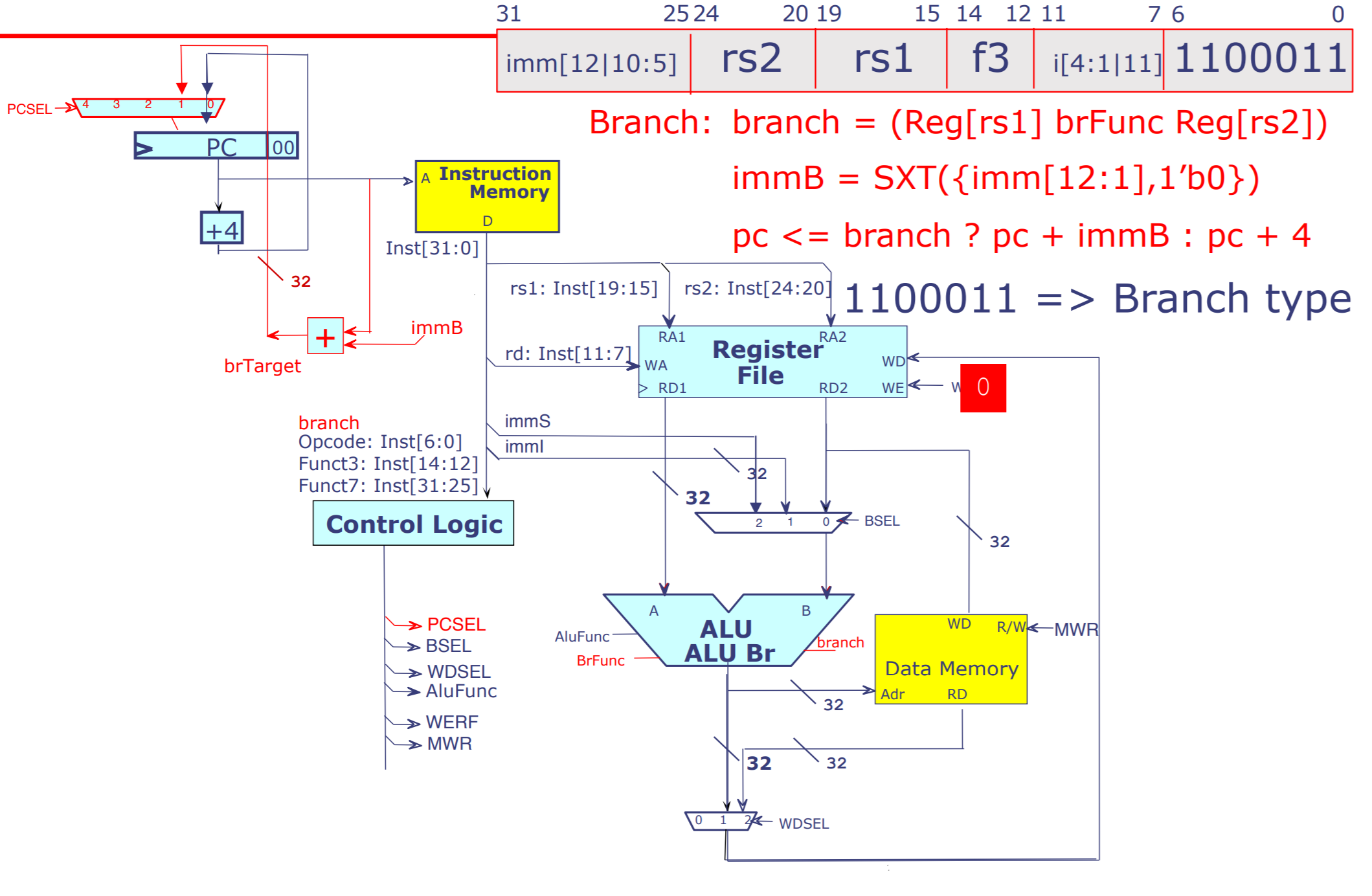
Store instruction datapath

**SB-Type instructions :** are used to implement branch instructions that arise due to the use of loops, conditions etc. they primarily change the value of program counter so that the next instruction in sequence can be stored in a different location in instruction memory reducing redundancy in our code. These include BEQ, BNE, BLT, BGE.

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Opcode** | **Func3** |
| beq | 1100011 | 000 |
| bne | 1100011 | 001 |
| blt | 1100011 | 100 |
| bge | 1100011 | 110 |

**Datapath for conditional branch operation:**

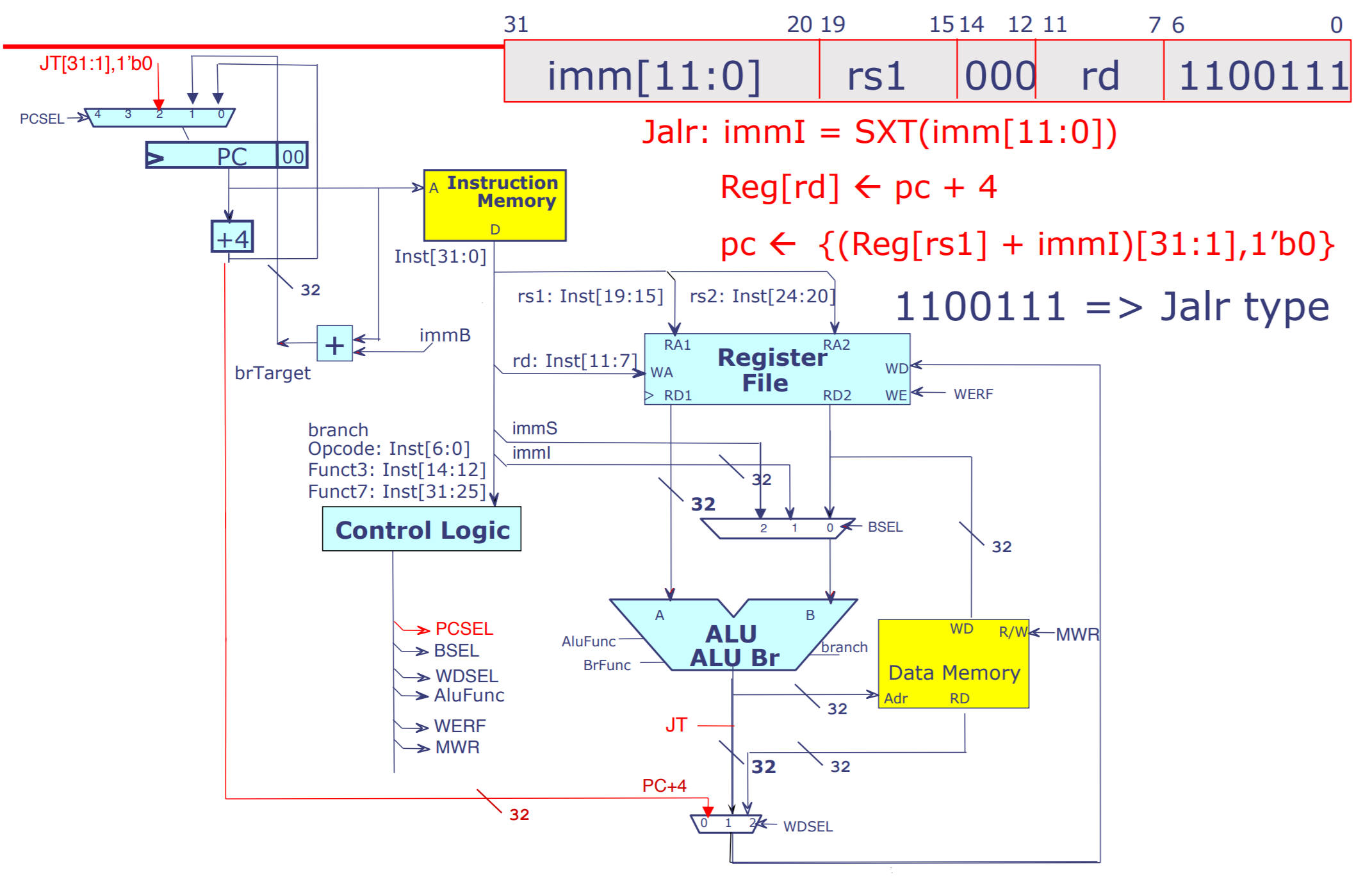
ALU returns a bool value indicating whether the condition is met or not and depending on that we change the value of the program counter. Other components remain unused in this step.



Conditional branch instruction datapath

**Jump-Instruction:** they include JAL and JALR instructions and are known as unconditional jump. The difference between the two is in the source of data that determines jump size. In JALR jump size is determined by the value stored in a register whereas in JAL that value comes from an immediate field.

**Datapath for jump/unconditional branch operation:**

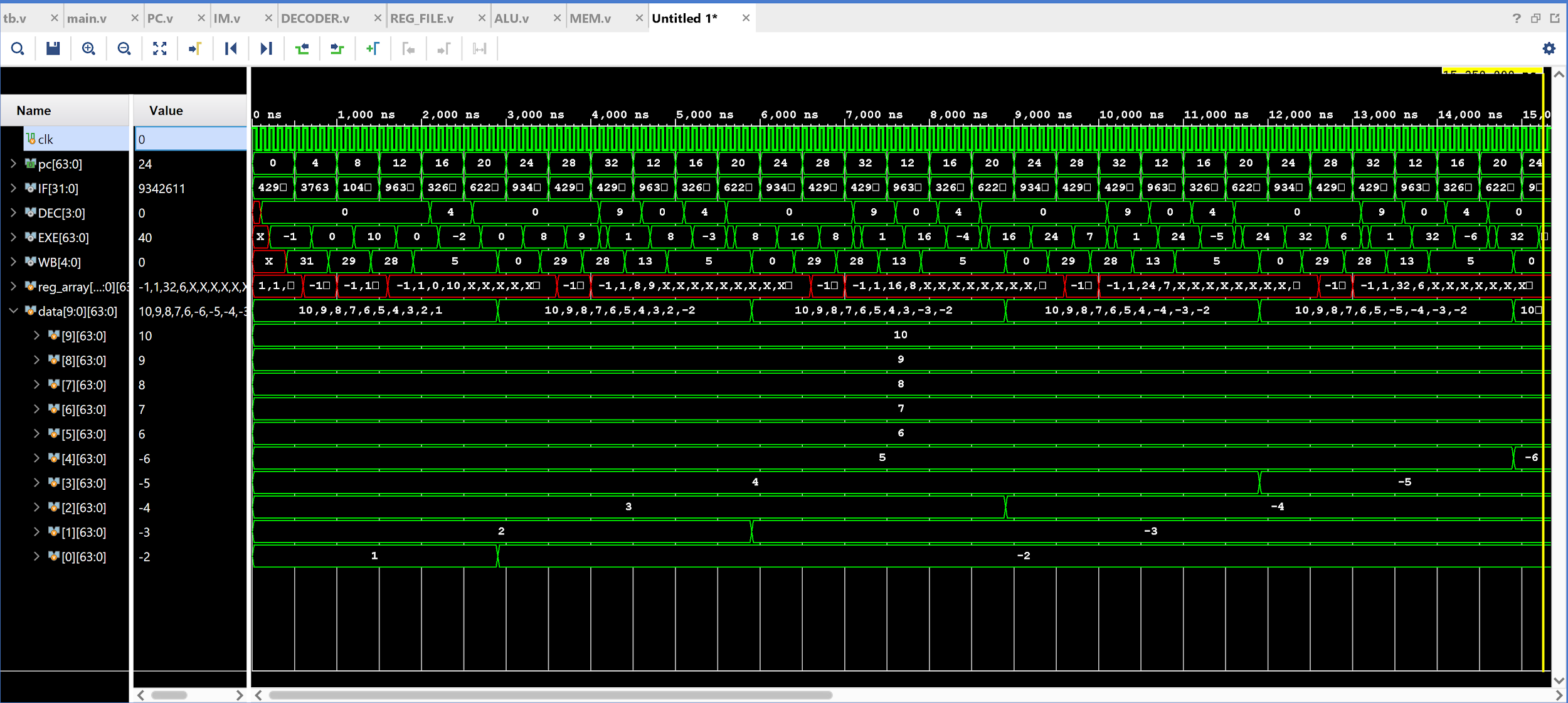


The value of PC+4 is stored in “rd” and then the PC value is updated. So the next instruction jumps to our new PC value calculated in this step.

## **Observations**

It was observed that the program was working correctly for all the input instructions given through the test-bench and was verified manually through the simulation tab as well as manually in the REGISTER\_FILE.

Below is a figure that shows the program run for the inputs given in the test-bench.



**Simulation**

The output shown in the figure is in signed decimal form and tested against these instructions.

Instructions given to the processor as a test were as shown above.

After complete Execution of the above code the following changes were made in the adat memory.

Memory before execution of the code



Memory before execution of the code

## 

## **Result**

The modules for performing all the types of instructions are working for every valid instruction given through the testbench. The desired datapath for the basic functioning of the processor was also successfully created.

**CODE:**

Testbench

`timescale 1ns / 1ps

module tb();

reg clk = 0;

wire [31:0] IF;

wire [3:0] DEC;

wire [63:0] EXE;

wire [4:0] WB;

initial begin

forever #50 clk = ~clk;

end

main DUT( .clk(clk),.inst(IF),.func(DEC),.res(EXE),.wa(WB) );

endmodule

Instruction Memory (IM)

`timescale 1ns / 1ps

module **IM**( input [63:0] pc, output [31:0] inst );

reg [31:0] data [31:0];

initial begin

$readmemb("C:\\Users\\Abdullah Khan\\Desktop\\project\_1\\instr.txt",data);

end

assign inst = data[pc/4];

endmodule

Program Counter (PC)

`timescale 1ns / 1ps

module **PC**( input clk, en, input [1:0] pcsel, input [63:0] branch,immed, output reg [63:0] pc, output reg [63:0] pc\_4 );

reg [63:0] pc2;

initial begin

pc = -4;

end

always@(posedge clk ) begin

case(pcsel)

2'b01:begin

if(branch[0] == 1) pc2 = pc + immed;

else pc2 = pc + 4;

end

2'b10: pc2 = branch; //JALR

2'b11: pc2 = pc + branch; //JAL

default: pc2 = pc + 4;

endcase

if (en)

pc = pc2;

pc\_4 = pc + 4;

end

endmodule

DECODER

`timescale 1ns / 1ps

module **DECODER**(input [31:0] inst,output [4:0] rs2,rd,output reg [4:0] rs1,output reg [63:0] immediate,output reg [3:0] AluFunc,output reg bsel,mwr,werf,type,reg [1:0] wdsel, pcsel);

wire [6:0] opcode,func7;

wire [5:0] func6;

wire [2:0] func3;

assign rs2 = inst[24:20];

assign rd = inst[11:7];

assign opcode = inst[6:0];

assign func3 = inst[14:12];

assign func7 = inst[31:25];

assign func6 = inst[31:26];

always@(\*) begin

case(opcode)

7'b0110011 : begin // **R - type**

if(func3 == 3'b000)begin

if(func7==7'b0000000) AluFunc = 4'b0000; //ADD - add

else if(func7==7'b0100000) AluFunc = 4'b0001; //SUBTRACT - sub

end

else if(func3 == 3'b111) AluFunc = 4'b0010; //AND - and

else if(func3 == 3'b110) AluFunc = 4'b0011; //OR - or

bsel=0; wdsel=2'b0; mwr = 0; werf=1; pcsel = 2'b00; type = 1;

rs1 = inst[19:15];

end

7'b0010011 : begin //**I - type arithmetic**

if(func3 == 3'b000)begin

AluFunc = 4'b0000; //ADD\_I - addi

immediate = {{52{inst[31]}},inst[31:20]};

end

else if(func3 == 3'b100)begin //XOR\_I - xori

AluFunc = 4'b0100;

immediate = {{52{inst[31]}},inst[31:20]};

end

else if(func3 == 3'b001)begin

if (func6 == 6'b000000) AluFunc = 4'b0101; //SLL\_I - slli

immediate = {{58{inst[31]}},inst[25:20]};

end

else if(func3 == 3'b101)begin

if(func6 == 6'b000000) AluFunc = 4'b0110; //SRL\_I - srli

else if(func6 == 6'b010000) AluFunc = 4'b0111; //SRA\_I - srai

immediate = {{58{inst[31]}},inst[25:20]};

end

bsel = 1; wdsel=0; mwr = 0; werf=1; pcsel = 2'b0; type = 1;

rs1 = inst[19:15];

end

7'b0000011 : begin //**I - type load**

if(func3 == 3'b011)begin //LOAD\_DOUBLE ld

AluFunc = 0000;

type = 1;

end

if(func3 == 3'b010)begin //LOAD\_WORD lw

AluFunc = 0000;

type = 0;

end

immediate = {{52{inst[31]}},inst[31:20]};

bsel = 1; wdsel = 1; mwr = 0; werf=1; pcsel = 2'b0;

rs1 = inst[19:15];

end

7'b0100011 : begin //**S - type**

if(func3 == 3'b111) AluFunc = 0000;

immediate = {{52{inst[31]}},inst[31:26],inst[11:7]};

bsel=1; wdsel = 0; mwr = 1; werf=0; pcsel = 2'b0; type = 1;

rs1 = inst[19:15];

end

7'b1100011 : begin //**SB - type**

if(func3 == 3'b000) AluFunc = 4'b1000; // BEQ

else if (func3 == 3'b001) AluFunc = 4'b1001; //BNE

else if (func3 == 3'b100) AluFunc = 4'b1010; //BLT

else if (func3 == 3'b101) AluFunc = 4'b1011; //BGE

immediate = {{51{inst[31]}},inst[7],inst[30:25],inst[11:8],1'b0};

bsel=0; wdsel = 0; mwr = 0; werf=0; pcsel = 2'b01; type = 1;

rs1 = inst[19:15];

end

7'b1100111 : begin //**I-type**

if(func3 == 3'b000) AluFunc = 4'b0000; //JALR

immediate = {{58{inst[31]}},inst[25:20]};

bsel=1; wdsel = 2'b10; mwr = 0; werf=1; pcsel = 2'b10; type = 1;

rs1 = inst[19:15];

end

7'b1101111 : begin //**UJ - type**

AluFunc = 4'b0000; //JAL

immediate = {{44{inst[31]}},inst[19:12],inst[20],inst[30:21],1'b0};

bsel=1; wdsel = 2'b10; mwr = 0; werf=1; pcsel = 2'b11; type = 1;

rs1 = 5'b0;

end

endcase

end

endmodule

REG\_FILE

`timescale 1ns / 1ps

module **REG\_FILE**(input clk,input [4:0] RA1, RA2, WA,input [63:0] WD,input W\_EN,output [63:0] RD1,RD2);

reg [63:0] reg\_array [31:0];

integer file,i;reg en = 1;

initial begin

file = $fopen("C:\\Users\\Abdullah Khan\\Desktop\\project\_1\\RF.txt","r");

for(i=0;i<32;i=i+1)

$fscanf(file,"%b\n",reg\_array[i]);

$fclose(file);

en = 1;

end

assign RD1 = reg\_array[RA1];

assign RD2 = reg\_array[RA2];

always@(posedge clk) begin

if(W\_EN==1 && en==1) begin

if(WA == 5'b0) reg\_array[WA] = 64'b0;

else reg\_array[WA] = WD;

file = $fopen("C:\\Users\\Abdullah Khan\\Desktop\\project\_1\\RF.txt","w");

for(i=0;i<32;i=i+1)

$fwrite(file,"%b\n",reg\_array[i]);

$fclose(file);

En = 0;

end

else en = 1;

end

endmodule

ALU

`timescale 1ns / 1ps

module **ALU**(input [63:0] A, B,input [3:0] op,output reg [63:0] result);

// ADD : 0000 // SUB : 0001 // AND : 0010 // OR : 0011 // XOR : 0100 // SLL : 0101

// SRL : 0110 // SRA : 0111 // BEQ : 1000 // BNE : 1001 // BLT : 1010 // BGE : 1011

always@(\*) begin

case(op)

4'b0000 : result <= A + B;

4'b0001 : result <= A - B;

4'b0010 : result <= A & B;

4'b0011 : result <= A | B;

4'b0100 : result <= A ^ B;

4'b0101 : result <= A << B;

4'b0110 : result <= A >> B;

4'b0111 : result <= $signed(A) >>> B;

4'b1000 : result <= (A==B) ? 1 : 0;

4'b1001 : result <= (A!=B) ? 1 : 0;

4'b1010 : result <= (A<B) ? 1 : 0;

4'b1011 : result <= (A>=B) ? 1 : 0;

endcase

end

endmodule

MEM

`timescale 1ns / 1ps

module **MEM**(input clk,mwr,type,input [63:0] addr,input [63:0]W,output [63:0] out );

reg [63:0] data [9:0];

integer file,i;

initial begin

file = $fopen("C:\\Users\\Abdullah Khan\\Desktop\\project\_1\\memory.txt","r");

$readmemb("C:\\Users\\Abdullah Khan\\Desktop\\project\_1\\memory.txt",data);

$fclose(file);

end

assign out = (type==1'b1)?data[addr/4] : {{34{data[addr/4][31]}},data[addr/4][30:0]};

always@(posedge clk) begin

if(mwr==1) begin

data[addr] = W;

file = $fopen("C:\\Users\\Abdullah Khan\\Desktop\\project\_1\\memory.txt","w");

for(i=0;i<10;i=i+1) $fwrite(file,"%b\n",data[i]);

end

end

endmodule

Main module

`timescale 1ns / 1ps

module **main**(input clk,output [31:0] inst,output [3:0] func,output [63:0] res,output reg [4:0] wa);

wire [63:0] A,B,immed,mem\_out,pc\_out,pc4;

wire [4:0] rs1,rs2,rd;

wire en,imm,mwr,out,ty;

wire [1:0] wdsel,pcsel;

reg [1:0] w1,w2;

reg [31:0] fetch;

reg [63:0] val1,val2,out\_val,to\_mem,b1,b2,pc1,pc,pcx;

reg [4:0] w\_addr,wa2;

reg [3:0] op;

reg w\_en,w\_en2,w\_en1,m1,m2,typ1,typ2;

reg enable\_IM=0;

integer i=-1;

always@(posedge clk) begin

i=i+1;

if(i%5==0) enable\_IM = 1;

if(i%5==1) enable\_IM = 0;

end

always@(posedge clk) begin

fetch <= inst;

op <= func;

w\_addr <= rd;

wa2 <= w\_addr;

wa<=wa2;

val1 <= A;

val2 <= (imm==0)?B:immed;

b1<=B;

b2<=b1;

w1<=wdsel;

w2<=w1;

to\_mem <= res;

pc1 <= pc4;

pcx<=pc1;

pc <= pcx;

out\_val <= (w2==2'b01)? mem\_out : ( (w2==2'b0)?to\_mem : ( (w2==2'b10)?pc:-1 ) );

w\_en2 <= en;

w\_en1 <= w\_en2;

w\_en <=w\_en1;

m1<=mwr;

m2<=m1;

typ1<=ty;

typ2<=typ1;

end

**PC** PC(.clk(clk),.en(enable\_IM),.pcsel(pcsel),.branch(res),.immed(immed),.pc(pc\_out)

,.pc\_4(pc4));

**IM** mem(.pc(pc\_out),.inst(inst));

**DECODER** control(.inst(fetch),.rs1(rs1),.rs2(rs2),.rd(rd),.AluFunc(func),.bsel(imm)

,.immediate(immed),.wdsel(wdsel),.mwr(mwr),.werf(en),.pcsel(pcsel),.type(ty) );

**REG\_FILE** rf(.clk(clk),.RA1(rs1),.RA2(rs2),.WA(wa),.WD(out\_val),.W\_EN(w\_en)

,.RD1(A),.RD2(B));

**ALU** alu32(.A(val1),.B(val2),.op(op),.result(res));

**MEM** data\_memory(.clk(clk),.addr(to\_mem),.W(b2),.mwr(m2),.out(mem\_out),.type(typ2));

endmodule