CS F342
Computer Architecture

Project Report

BITS Pilani
Pilani | Dubai | Goa | Hyderabad | Mumbai

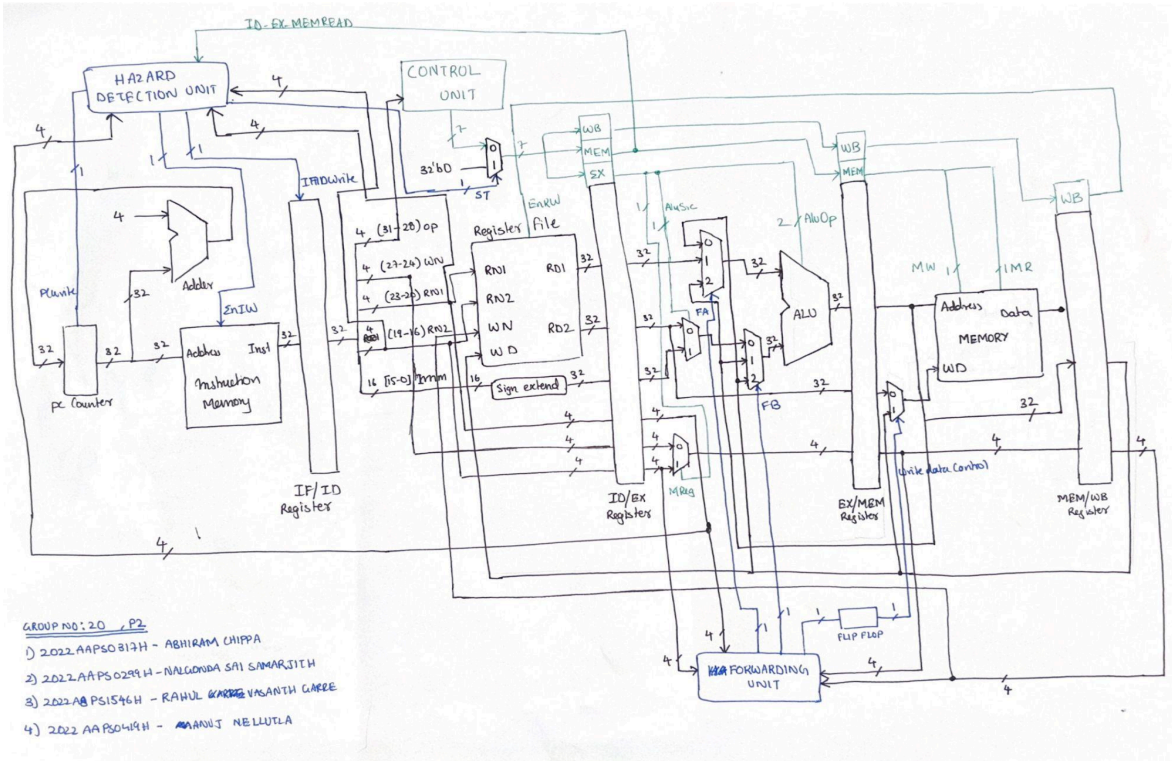
Group No: 20

Section: P2

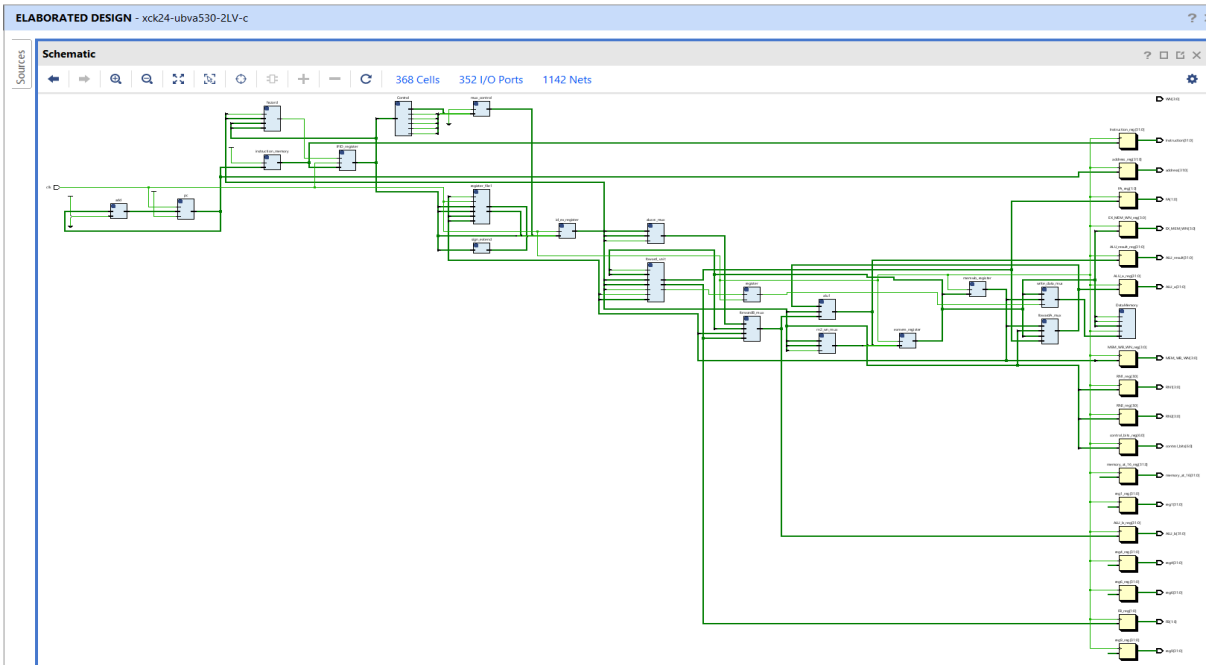
ID Number	Name
ABHIRAM CHIPPA	2022AAPS0317H
NALGONDA SAI SAMARJITH	2022AAPS0299H
RAHUL VASANTH GARRE	2022A8PS1546H
ANUJ NELLUTLA	2022AAPS0419H

Block Diagram of the RISC Processor

Assignment Question: Link (hyperlinked)



fig(1): Architecture of MIPS 32(Modified)



fig(2): Schematic obtained from Vivado

Description Of The Components

Program Counter: Program counter is a register controlled by a clock(100Mhz) and incremented by 4(since 32 bits as 4 bytes) on every clock cycle until or unless a hazard is detected.

Instruction Memory: Instruction memory takes address from program counter and required fields like opcode,rn1,rn2,wn and offset will be available in the memory location of that instruction.

IF/ID Register: This register will store output of the instruction memory contents(like opcode,rn1..etc as mentioned above) in the registers to make them available for the ID(Instruction Decode)step in the next clock cycle.Also,we need not forward pc+4 value as the set has no branch instructions.

Register File: From IF/ID register we get register location information.Using that as an input to register file we will read data from registers.If EnRW signal is triggered we will write into register (WN is location and write data is from write back(WB) step).Here,write back occurs at negative edge of clock cycle and read at positive edge of clock cycle.

Sign Extend: Sign extension is for load/sw instructions ,to add base address with offset(16 bits) .MSB of offset is extended for 16 bits and appended with original offset.

Control Unit: Location from where all the control signals(excluding pc write and IFIDWrite because they are before instruction decode step) are generated using the opcode as input.

ID/EX Register: This register stores information of read1,read2,control signals,rn1,rn2,wn and sign extended data, as this information is required in next clock cycle for EX(execute) step.Contents of ID/EX are shown in fig(3).

ALU: ALU does operation based on the control signal ALUop from control unit.Inputs for alu comes from a mux controlled by forwarding unit as in fig(1).ALU inputs are 32 bits wide and control signal is 2 bits wide.

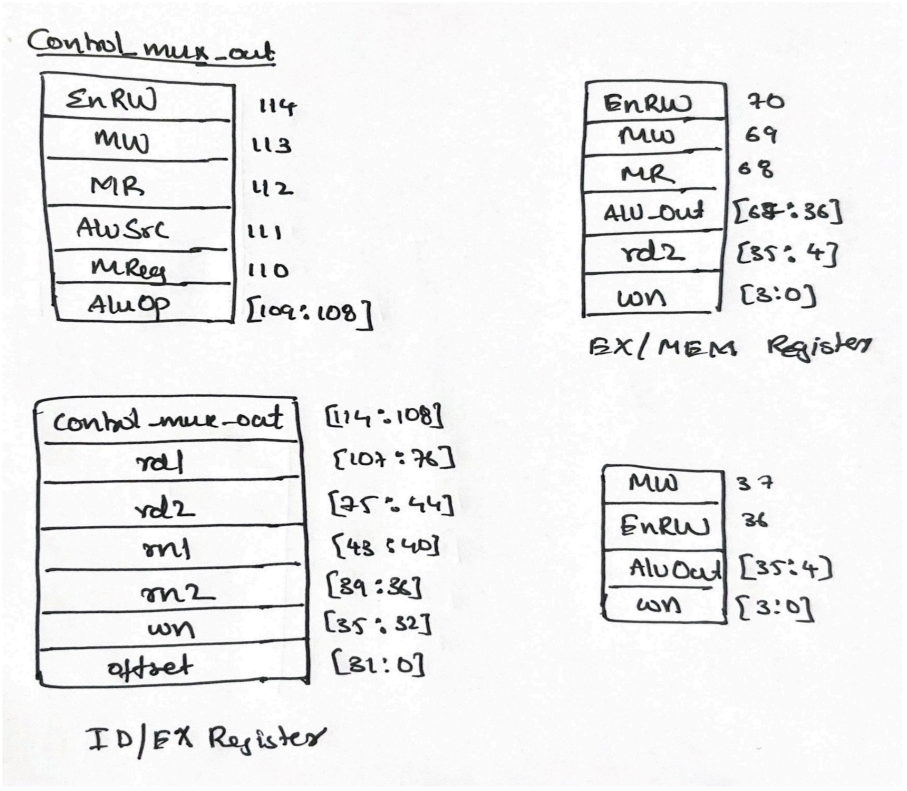
EX/MEM Register(71 bits): This register stores information of alu output, write back control signals,memory access control signals , wn and read2.This information is required for subsequent MEM and WB steps.Contents of EX/MEM are shown in fig(3).

Data Memory: This unit is helpful for lw/sw type instructions as other instructions dont need any memory access.For all other types of instructions,data is forwarded to MEM/WB register.And the output of data memory is not required because we are not working on load instructions.Also read or write is controlled by two control signals MR and MW respectively.

MEM/WB Register(38 bits): This register stores information of ALU output forwarded from EX/MEM register, write back control signals and wn as mentioned in fig(3).

Forwarding Unit: It plays a key role in forwarding the register data/memory data to next instructions before completing the final step if there exists any dependencies and protects from garbage outputs and saves clock cycles.

Hazard Detection Unit: Mainly used for lw/sw type instruction where instruction decode step has to wait for a clock cycle even after using data forwarding (due to memory access).So,we should not update pc counter and IF/ID register should be flushed as it contains wrong information(This is called as a stall in a pipeline).



fig(3): Register Contents and size

Verilog Code

Link to access Verilog module files: Verilog_Modules (hyperlinked)

Program Counter:

```
module pc_counter(
input clk,
input [31:0] pc_in,
input pc_write,
output reg [31:0] pc_out
);

initial pc_out = 32'b00;

always @(posedge clk) begin
    if (pc_write)
        pc_out <= pc_in;
end
endmodule
```

Instruction Memory:

```
module inst_memory(
input [31:0] address,
input EnIM,
output reg [31:0] inst
);

reg [7:0] memory [0:31];

initial begin
memory[0] = 8'h01; memory[1] = 8'h23; memory[2] = 8'h00; memory[3] = 8'h0;
memory[4] = 8'h14; memory[5] = 8'h15; memory[6] = 8'h0; memory[7] = 8'h0;
memory[8] = 8'h36; memory[9] = 8'h14; memory[10] = 8'h0; memory[11] = 8'h0;
memory[12] = 8'h70; memory[13] = 8'h76; memory[14] = 8'h00; memory[15] = 8'h05;
memory[16] = 8'hF0; memory[17] = 8'h89; memory[18] = 8'hAB; memory[19] = 8'h1E;
memory[20] = 8'h0; memory[21] = 8'h0; memory[22] = 8'h0; memory[23] = 8'h0;

end

always @(*) begin
    if (EnIM)
        inst <= {memory[address], memory[address + 1], memory[address + 2], memory[address +
3]};
end
endmodule
```

32-bit adder:

```
module adder(
input [31:0] a, b,
output  [31:0] out
);

assign out = a + b;


endmodule
```

IF/ID Register:

```
module IF_ID_reg(
input clk,
input IFIDWrite,
input [31:0] in,
output reg [31:0] out
);

always @(posedge clk) begin
    if(IFIDWrite)
        out <= in;
end
endmodule
```

Register file:

```
module register_file(
input clk,
input [3:0] rn1, rn2, wn,
input [31:0] wd,
input EnRW,
output [31:0] rd1, rd2
);

reg [32:0] register [0:15];

initial begin
    register[0] = 32'd0;
    register[1] = 32'h0;
    register[2] = 32'h64511;
    register[3] = 32'h343EE;
    register[4] = 32'h0;
    register[5] = 32'h8E319;
    register[6] = 32'h0;
    register[7] = 32'hB;
```

```
    register[8] = 32'hF3390;
    register[9] = 32'h0;
    register[10] = 32'h0;
    register[11] = 32'h0;
    register[12] = 32'h0;
    register[13] = 32'h0;
    register[14] = 32'h0;
    register[15] = 32'h0;
end

assign rd1 = register[rn1];
assign rd2 = register[rn2];


always @(negedge clk) begin
    if(EnRW)
        register[wn] <= wd;
end
endmodule
```

ID/EX Register:

```
module ID_EX_reg(
input clk,
input [114:0] in,
output reg [114:0] out
);
//initial out = 115'b0;
always @(posedge clk) begin
    out <= in;
end
endmodule
```

Forwarding Mux:

```
module mux_3_1(
input [31:0] a,b,c,
input [1:0] s,
output reg [31:0] out
);

always @(*) begin
    case(s)
        2'b00: out<= a;
        2'b01: out<= b;
        2'b10: out<= c;
    endcase
end
endmodule
```

ALU:

```
module alu(
input [31:0] a,b,
input [1:0] alu_op,
output reg [31:0] result
);

always @(*) begin
   case(alu_op)
      2'b00: result = a + b;
      2'b01: result = a - b;
      2'b10: result = a | b;
      2'b11: result = ~(a & b);
   endcase
end
endmodule
```

EX/MEM Register:

```
module EX_MEM_reg(
input clk,
input [70:0] in,
output reg [70:0] out
);

initial out = 71'b00;

always @(posedge clk) begin
   out <= in;
end
endmodule
```

Data Memory:

```
module data_memory(
input clk,
input MR, MW,
input [31:0] add, wd,
output reg [31:0] data
);

reg [7:0] mem [0:40];

initial begin
   mem[0] = 8'b0;
   mem[16] = 8'd45;
end
```

```
always @(posedge clk) begin
   if (MR == 1 && MW == 0)
      data <= {mem[add],mem[add+1], mem[add+2], mem[add+3]};
   else if (MR == 0 && MW == 1)
      {mem[add],mem[add+1], mem[add+2], mem[add+3]} <= wd;
end
endmodule
```

2 to 1 mux for writing data into memory:

```
module mux_2_1(
input [31:0] a,
input [31:0] b,
input s,
output reg [31:0] out
   );

always @(*) begin
   if (s==0)
      out <= a;
   else
      out <= b;
end
endmodule
```

MEM/WB Register:

```
module MEM_WB_reg(
input clk,
input [37:0] in,
output reg [37:0] out
);

initial out <=  38'b0;
always @(posedge clk) begin
   out <= in;
end
endmodule
```

Hazard Detection:

```
module hazard_detection(
input [3:0]if_id_rn1, if_id_rn2,
input [3:0]id_ex_rn2,
input id_ex_memread,
output reg IFIDWrite, ST, EnIW,
output reg pc_write
    );

initial begin
    pc_write = 1;
    IFIDWrite = 1;
    ST = 0;
    EnIW = 1;
end
always @(*) begin
    //default no stall

    if (id_ex_memread && ((id_ex_rn2 == if_id_rn1) || (id_ex_rn2 == if_id_rn2))) begin
        pc_write <= 0;
        IFIDWrite <= 0;
        ST <= 1;
    end
end
endmodule
```

Control Mux(used for flushing control signals):

```
module control_mux(
input [6:0] a,
input ST,
output reg [6:0] control
);

always @(*) begin
    if (ST == 0)
        control <= a;
    else
        control <= 7'b0;
end
endmodule
```

Forwarding Unit:

```
module forwarding_unit(
input  [3:0] ID_EX_rn1, ID_EX_rn2, EX_MEM_wn, MEM_WB_wn,
input  EX_MEM_RegWrite, MEM_WB_RegWrite ,MEM_WB_MW, //id_ex mw
output reg write_data_control,
output reg [1:0] fa, fb
);

always @(*) begin
   write_data_control <= 0;
   fa <= 2'b01;
   if (EX_MEM_RegWrite && (EX_MEM_wn != 4'b0) && (EX_MEM_wn == ID_EX_rn1))
      fa <= 2'b10;
   else if (MEM_WB_RegWrite && (MEM_WB_wn != 4'b0) && (MEM_WB_wn == ID_EX_rn1))
      fa <= 2'b00;
      fb <= 2'b00;
   if (EX_MEM_RegWrite && (EX_MEM_wn != 4'b0) && (EX_MEM_wn == ID_EX_rn2))
      fb <= 2'b01;
      if (MEM_WB_MW) begin
         fb <= 2'b00;
         write_data_control <= 1;
      end
   else if (MEM_WB_RegWrite && (MEM_WB_wn != 4'b0) && (MEM_WB_wn == ID_EX_rn2))
      fb <= 2'b00;
end

endmodule
```

Sign extension:

```
module sign_ex(
input [15:0] imm,
output [31:0] out
);
assign out = {{16{imm[15]}},imm};
endmodule
```

4 bit input 2 to 1 mux:

```
module mux_4bit(
input [3:0] a, b,
input s,
output [3:0] out
);

assign out = (a & {4{~s}}) | (b & {4{s}});
endmodule
```

Control Unit:

```
module control_unit(
input [3:0] op,
output reg ALUSrc, MR, MW, MReg, EnRW,
output reg [1:0] ALUOp
);

always @(*) begin
   case(op)
      4'b0000: begin //0000 ADD reg1, reg2, reg3
         ALUOp <= 2'b00;
         ALUSrc <= 0;
         MReg <= 1;
         EnRW <= 1;
      end

      4'b0001: begin //0004 SUB reg4, reg1, reg5
         ALUOp <= 2'b01;
         ALUSrc <= 0;
         MReg <= 1;
         EnRW <= 1;
      end

      4'b0011: begin //0008 OR reg6, reg1, reg4
         ALUOp <= 2'b10;
         ALUSrc <= 0;
         MReg <= 1;
         EnRW <= 1;
      end

      4'b0111: begin//0012 SW reg6, 5(reg7)
         ALUOp <= 2'b00;
         MR <= 0;
         MW <= 1;
         ALUSrc <= 1;
         MReg <= 0;
         EnRW <= 0;
      end
      4'b1111: begin //0016 NANDI reg9, reg8, AB1E
         ALUOp <= 2'b11;
         ALUSrc <= 1;
         MReg <= 0;
         EnRW <= 1;
      end
   endcase
end
endmodule
```

DATAPATH:

```
module datapath(
input clk,
output reg [31:0] address, Instruction, ALU_a, ALU_b, ALU_result,
output reg [31:0] reg1, reg4, reg6, reg9, memory_at_16,
output reg [6:0] control_bits,
output reg [1:0] FA, FB,
output reg [3:0] RN1, RN2, WN, EX_MEM_WN, MEM_WB_WN
);

wire [31:0] pc_in, pc_out;
wire pc_write, EnIM, EnIW;

pc_counter pc(clk, pc_in, 1'b1, pc_out);

adder add(pc_out, 32'd4, pc_in);

wire [31:0] inst, ifid_out;
wire IFID_Write, ST; //from hazard;
wire [3:0] op, wn, rn1, rn2;
inst_memory instruction_memory(pc_out, 1, inst);

IF_ID_reg IFID_register(clk, EnIW, inst, ifid_out); //IFID write change

////////////////////////////////
hazard_detection hazard(rn1, rn2, id_ex_out[39:36], id_ex_out[112], IFID_Write, ST, EnIW,
pc_write);
//////////////////////////

wire [15:0] imm;

assign op = ifid_out[31:28];
assign wn = ifid_out[27:24];
assign rn1 = ifid_out[23:20];
assign rn2 = ifid_out[19:16];
assign imm = ifid_out[15:0];

wire ALUSrc, MR, MW, MReg, EnRW;
wire [1:0] ALUOp;

control_unit Control(op, ALUSrc, MR, MW, MReg, EnRW, ALUOp);

wire [6:0] control_mux_in, control_mux_out;
assign control_mux_in = {EnRW, MW, MR, ALUSrc, MReg, ALUOp}; // wb, m, ex

control_mux mux_control(control_mux_in, 0, control_mux_out);// change ST

wire [37:0] mem_wb_out, mem_wb_in;
```

```
//wire [3:0] mem_wb_wn;
//wire [31:0] mem_wb_wd;
//wire mem_wb_EnRW;

wire [31:0] rd1, rd2;
register_file register_file1(clk, rn1, rn2, mem_wb_out[3:0], mem_wb_out[35:4], mem_wb_out[36],
rd1, rd2);
// assign the value from mem_wb register

wire [31:0] offset;
sign_ex sign_extend(imm, offset);

wire [114:0] id_ex_in, id_ex_out;

assign id_ex_in = {control_mux_out, rd1, rd2, rn1, rn2, wn, offset};

ID_EX_reg id_ex_register(clk, id_ex_in, id_ex_out);

wire [3:0] wn1;
mux_4bit rn2_wn_mux(id_ex_out[39:36], id_ex_out[35:32], id_ex_out[110], wn1);
                // rn2, wn, MR

wire [31:0] fb_mux_in;
mux_2_1 alusrc_mux(id_ex_out[75:44], id_ex_out[31:0], id_ex_out[111], fb_mux_in);
               //rd2, offset, alusrc

wire [70:0] ex_mem_out, ex_mem_in;

// Forwarding UNIt
wire write_data_control;
wire [1:0] fa, fb;
forwarding_unit foward_unit(id_ex_out[43:40], id_ex_out[39:36], ex_mem_out[3:0],
mem_wb_out[3:0], ex_mem_out[70], mem_wb_out[36], id_ex_out[113],write_data_control, fa, fb);
//change this
////////////////////////////////////

wire [31:0] alu_a, alu_b;
mux_3_1 forwardB_mux(fb_mux_in, ex_mem_out[67:36], mem_wb_out[35:4], fb, alu_b); //change
fb
                //fb_mux_in, alu_out, alu_out
mux_3_1 fowardA_mux(mem_wb_out[35:4], id_ex_out[107:76], ex_mem_out[67:36], fa, alu_a);
//change fa
               //alu_out, rd1, alu_out

wire [31:0] alu_out;
alu alu1(alu_a, alu_b, id_ex_out[109:108], alu_out);

assign ex_mem_in = {id_ex_out[114], id_ex_out[113], id_ex_out[112], alu_out, id_ex_out[75:44],
wn1};
```

```
                //regwrite, MW, MR, alu_out, rd2, wn

EX_MEM_reg exmem_register(clk, ex_mem_in, ex_mem_out);

wire write_data_control1;
regis register(clk, write_data_control, write_data_control1);

wire [31:0] mux_wd, data_out;//data_out is not required because we are not using load
mux_2_1 write_data_mux (ex_mem_out[35:4], mem_wb_out[35:4], write_data_control1, mux_wd);

data_memory DataMemory(clk, ex_mem_out[68], ex_mem_out[69], ex_mem_out[67:36], mux_wd,
data_out);

assign mem_wb_in = {ex_mem_out[69], ex_mem_out[70], ex_mem_out[67:36],
ex_mem_out[3:0]};
                //memwrite, regwrite, alu_out, wn
MEM_WB_reg memwb_register(clk, mem_wb_in, mem_wb_out);


always @(posedge clk) begin
   address <= pc_out;
   Instruction <= inst;
   ALU_a <= alu_a;
   ALU_b <= alu_b;
   ALU_result <= alu_out;
   memory_at_16 <= {DataMemory.mem[16], DataMemory.mem[17], DataMemory.mem[18],
DataMemory.mem[19]};
   control_bits <= id_ex_out[114:108];
   FA <= fa;
   FB <= fb;
   RN1 <= id_ex_out[43:40]; //rn1
   RN2 <= id_ex_out[39:36];// rn2
   EX_MEM_WN <= ex_mem_out[3:0]; // wnex_mem_out[3:0]
   MEM_WB_WN <= mem_wb_out[3:0]; // wn mem_wb_out[3:0]

end
always @(negedge clk) begin
   reg1 <= register_file1.register[1];
   reg4 <= register_file1.register[4];
   reg6 <= register_file1.register[6];
   reg9 <= register_file1.register[9];
end

endmodule
```

Testbench

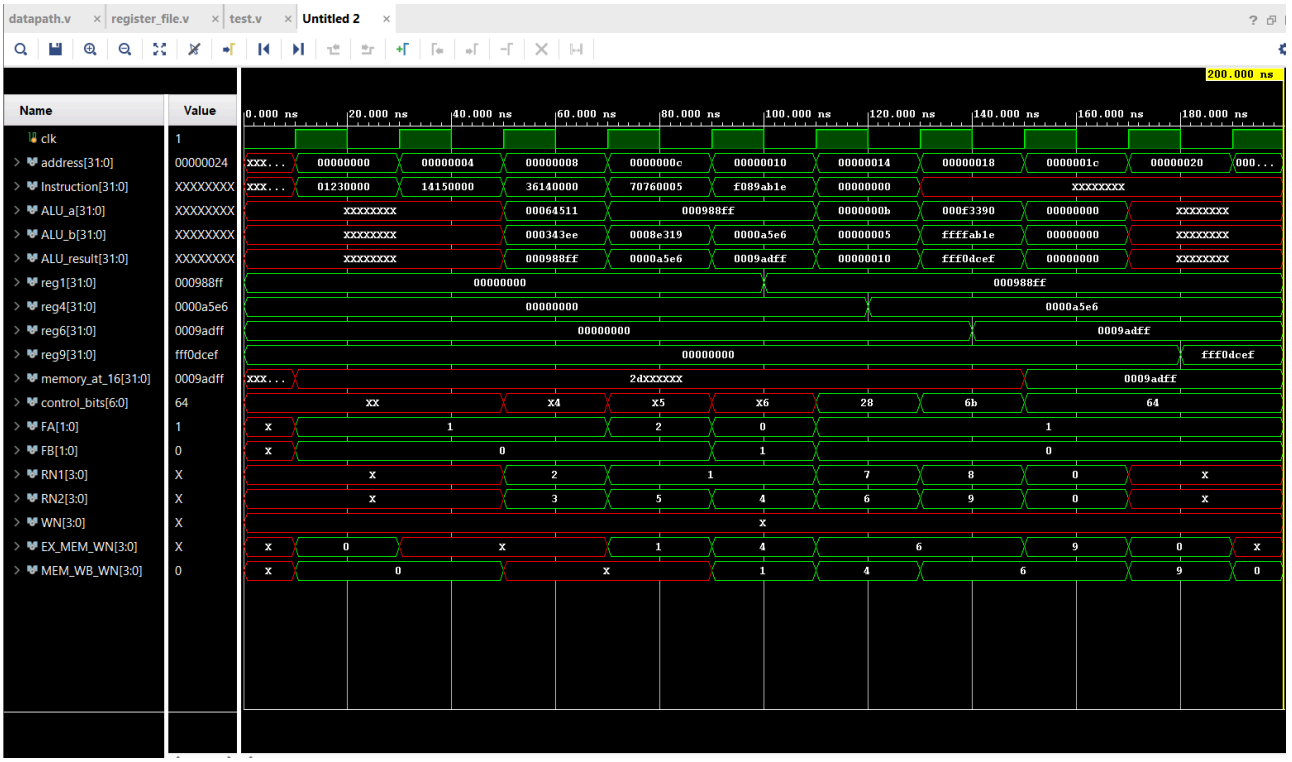
```
module test();
reg clk;
wire [31:0] address, Instruction, ALU_a, ALU_b, ALU_result;
wire [31:0] reg1, reg4, reg6, reg9, memory_at_16;
wire [6:0] control_bits;
wire [1:0] FA, FB;
wire[3:0] RN1, RN2, WN, EX_MEM_WN, MEM_WB_WN;

datapath dut(clk, address, Instruction, ALU_a, ALU_b, ALU_result,
            reg1, reg4, reg6, reg9, memory_at_16, control_bits,
            FA, FB, RN1, RN2, WN, EX_MEM_WN, MEM_WB_WN);

always #10 clk = ~clk;

initial begin
   clk = 0;
   #200;
   $finish;
end
endmodule
```

Output Waveform



Work Contribution

- Datapath and forwarding unit: Done by Abhiram Chippa and Sai samarjith Nalgonda
- Schematic block design & report: Done by Rahul Garre and Anuj