RISC-SICS Core

This file describes the 32-bit RISC-V processor core. The RISC-SICS processor core comprises an instruction fetch stage, an instruction decoder stage, an execution stage, a data memory stage, a write back stage and hazard detection unit.

The RISC-SICS feature set is summarized in Table 1.

Features	Description
Specifications	RISC-V
ISA	RV32I base instruction set, ver. 2.0
Privileged Mode	Not supported
Memory	32-bit byte accessible
Pipeline	5-Stage
Language	Verilog
Max. Operation Frequency	54.35 MHz
Utilization	2029 LUT and 1520 FF
Power Consumption	152 mW

Table 1

1. Pipeline Stages

1.1 Instruction Fetch Stage

Instruction memory and program counter (PC) modules form the instruction fetch (IF) stage. The IF stage is followed by ID stage and instruction fetched from the instruction memory at the PC address is passed to the ID stage. The PC inherits both present and next instruction address. The PC module is designed specifically to handle the hazards. The memory is designed as parametric to configure it when it is necessary. However, for the sake of the datasheet and the simulation and testing of the processor, the instruction memory arranged as 16KB in size and the width of an instruction is 32-bits. Figure 1.1.1 shows RTL schematic of IF stage.

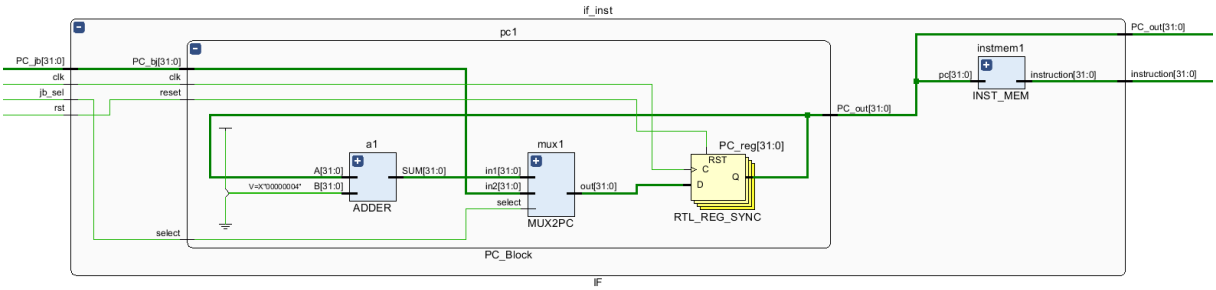

Figure 1.1.1

1.2 Instruction Decoder Unit

Instruction Decoder stage is the stage where instruction coming from IF stage is decoded and control signals are created. It consists of instruction decoder, immediate generator, register file, a comparator, two multiplexers, two adders and one branch/jump selection unit. RTL

schematic of this illustrated in Figure 2.1. Instruction decoder is capable of decoding RV32I Base Integer Instruction Set, Version 2.0.

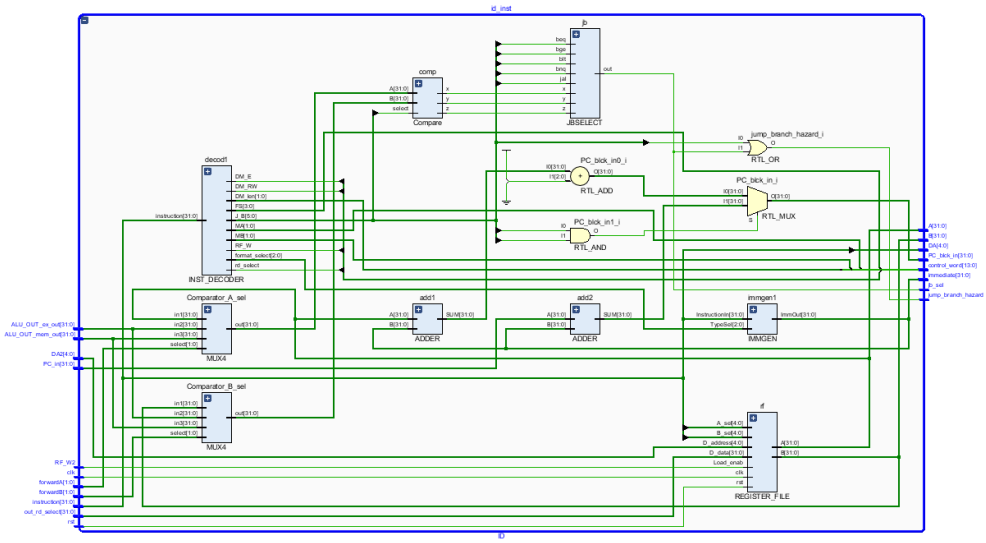

Figure 1.2.1

Instruction decoder module is responsible for reading instruction and its outputs control signals. Control signals are explained in the Table 2.1. It is designed to decode all RV32I Base Instruction Set except FENCE, ECALL, EBREAK instructions.

Signal Name	Function
RF_W	Controls write/read mode of Register File
MA, MB	Select which signal is directed to ALU
FS	Select which function is operated to ALU
MemCont	Controls Data Memory by determining write/read mode and length of signal
J_B	Outputs which jump/branch instruction is operated
rd_sel	Selects which data is written to register
format_sel	Selects one of the six different immediate types

Table 1.2.1

Immediate generator module is working with *format_sel* signal coming from instruction decoder. It can generate 6 types of immediates: R-type, I-type, S-type, B-type, U-type and J-type according to RISC-V base instruction formats.

Register file is responsible for holding register values while operation of processor. It can write to and read from registers at positive edge of clock signal. *RF_W and DA* signals control where to write data coming from write-back stage.

Multiplexer modules are responsible for detecting RAW hazards by comparing previous and current value of ALU output and generates forwarding signals. The remaining modules are

responsible for detecting type of jumping/branching and sending PC signal for new address of instruction.

1.3 Execution Stage

Execution Stage is the stage where all arithmetic and logic operations are done. It consists of four multiplexers and one arithmetic logic unit. Its RTL schematic is presented in the Figure 3.1.

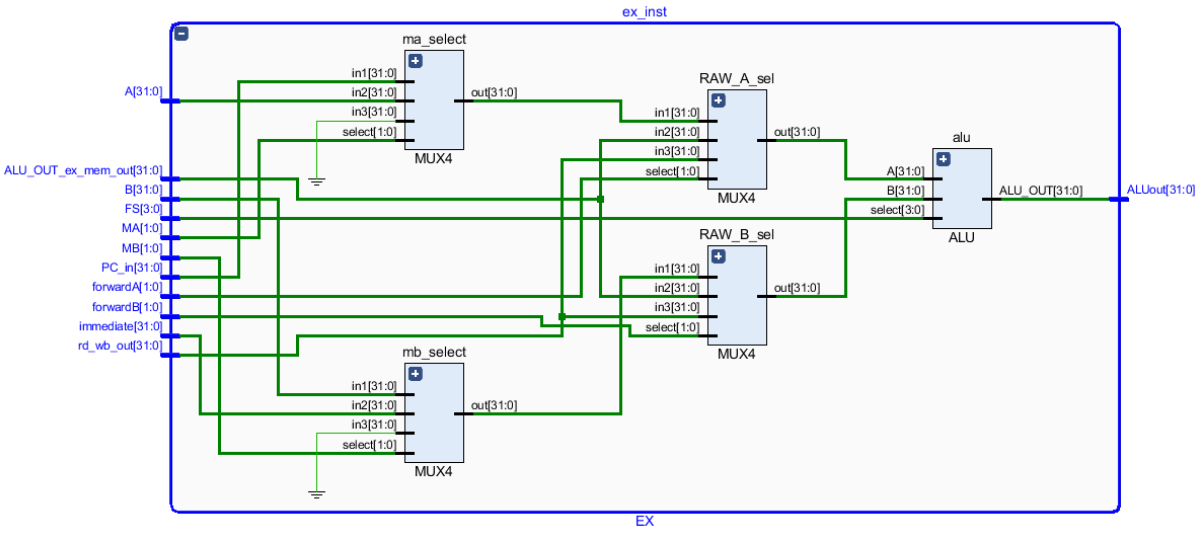

Figure 1.3.1

The ALU unit is responsible for operations: addition/subtraction, comparison, logic operations, and shifting operations. They are designed according to RISC-V rules.

There are two types of multiplexer in this stage. First one is responsible for input selection which selects either output of register or program counter, to calculate new instruction, address according to jumping/branching instructions. Second type is used for read-write data hazards which selects data according to RAW detection unit.

1.4 Data Memory Stage

The data memory is also reconfigurable and is reachable byte-wise, half-word-wise and word-wise. However, the memory sizes used in tests one byte wide. The 8-bit line size arrangement is made to write and read byte-wise and half-word-wise. The data memory and instruction memory are separated from each other, so a structural hazard is prevented. RTL schematic of data memory is presented in Figure 1.4.1.

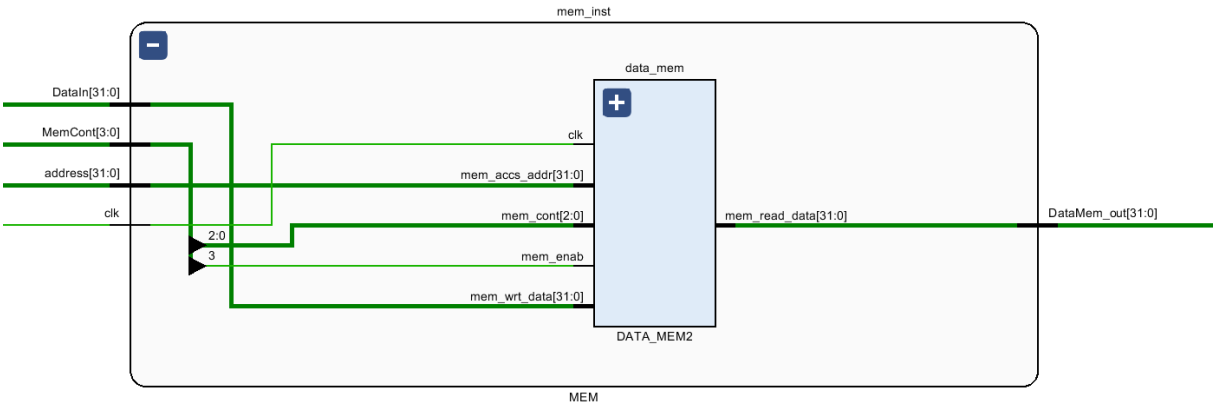

Figure 1.4.1

1.5 Write Back Stage

The write back stage is made of a multiplexer which selects the proper input that needs to be saved at the register file. Also, at this stage there is another input which lets us know if there should be a writing process on the register file. The RTL schematic is shown in Figure 3.4.1.

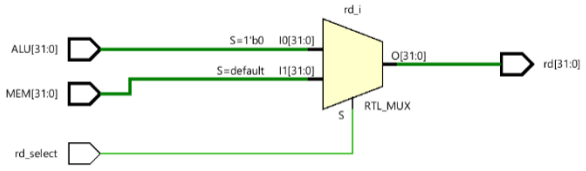

Figure 1.5.1

2. Hazard Detection

There are 4 types of Hazards which are solved in this architecture. Those hazards are: structural hazards, data hazards, load use hazards and branch-data hazards. Hazard detection unit is able to read through the destination addresses present in the processor and is able to find out if there is a hazard in the circuit. For instance, structural hazards happen when the same resource is utilized at the same time for two different purposes. This can only be solved by stalling the second instruction by one clock cycle. In this architecture, the structural hazards are solved by means of stalling. Stalling repeats the same instruction twice at the IF stage but only the second executed instruction will be able to have an effect on the CPU. The RTL schematic is shown in Figure 2.1.

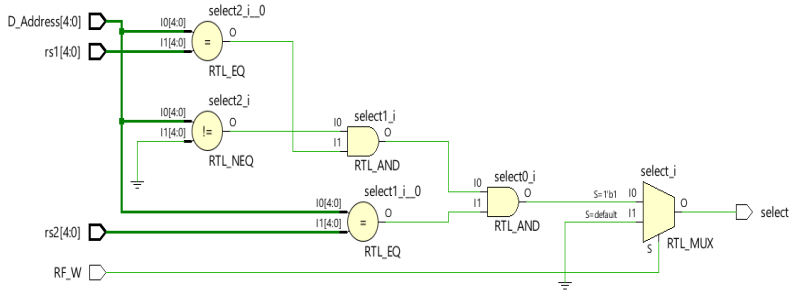

Figure 2.1

Another type of hazard which is encountered in our architecture is the Data Hazards. Since the circuits need a couple of cycles to complete, there could be instruction combinations where the calculated value from ALU is not stored yet in the register file so it cannot be ready to be used by the next instruction. This sort of hazard is also detected by having the detection circuit looking at the destination addresses, RF_W, rs1 and rs2 addresses of the processor. If this detection circuit would be activated, some forwarding circuit would forward the data from the output of the Execute and Memory stage to the input of the Execute stage. By means of some multiplexers at the inputs of the ALU, this problem is easily solved. The RTL schematic is shown in Figure 2.2.

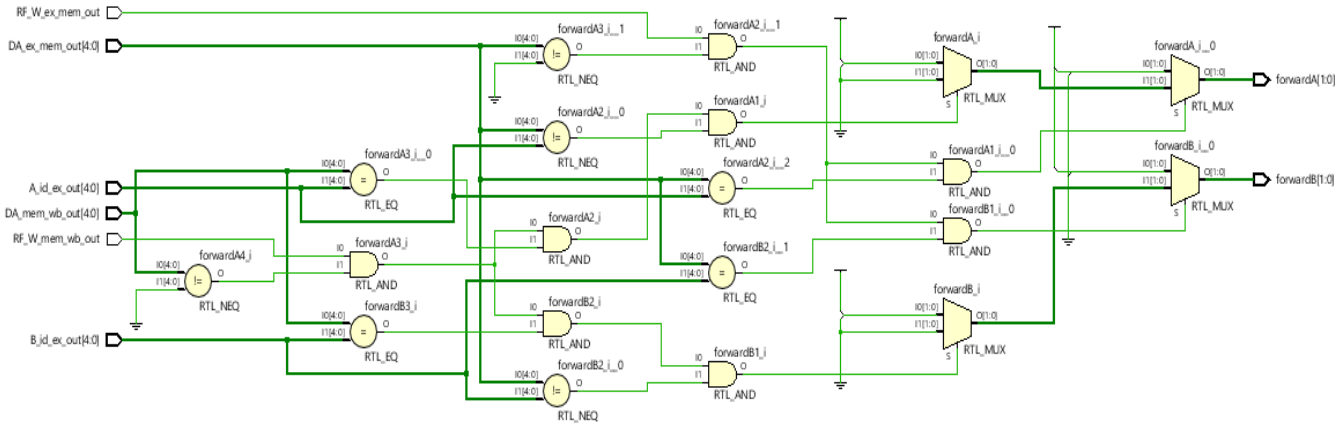

Figure 2.2

Load Use hazards happen also because of the time the data needs to be stored in the register file. Load Use hazards can be easily solved by means of stalling. The RTL schematic of the detection circuit is shown in Figure 2.3.

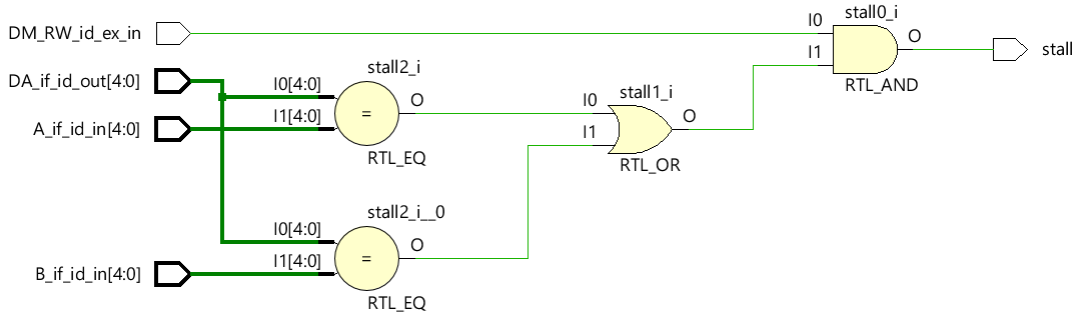

Figure 2.3

Since the branching is decided in the ID stage, there will also be forwarding problems for our architecture since some of the register file registers might not be updated and the wrong decision would have been done. To solve this a circuit was designed to detect the branching instruction and to verify if any forwarding should be done in the architecture. If forwarding is necessary, the circuit will forward values from the output of the Execute and Memory stage to the comparator of the ID stage. If the branching is done in the ID stage and the architecture should take the branch, the next instruction should be flushed away, otherwise it would create problems

in certain parts of the processor. The RTL schematic of the Branch and Forward detection is shown in Figure 2.4.

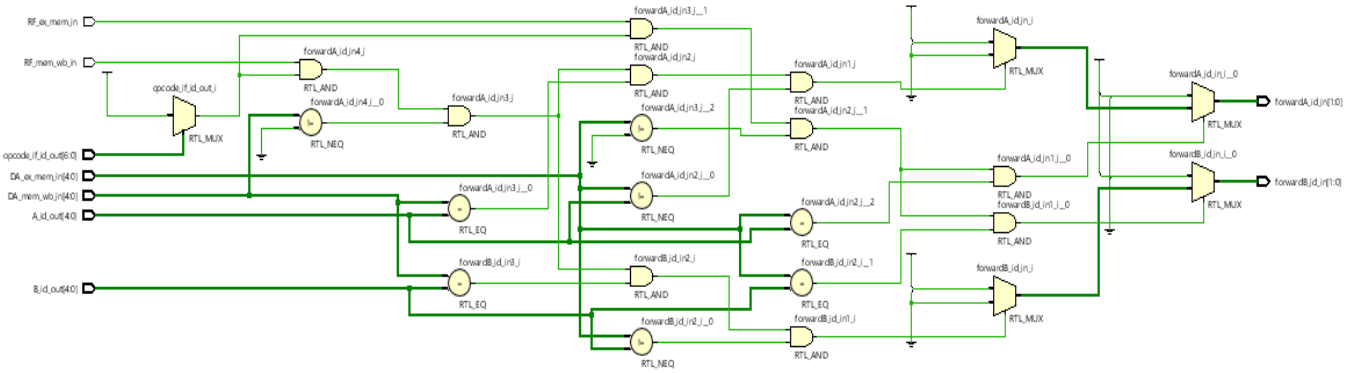

Figure 2.4

3. Performance Metrics

The performance metrics are measured by synthesizing and implementing the processor. The processor dissipates 152mW power. The power summary is shown in Figure 3.1 below.

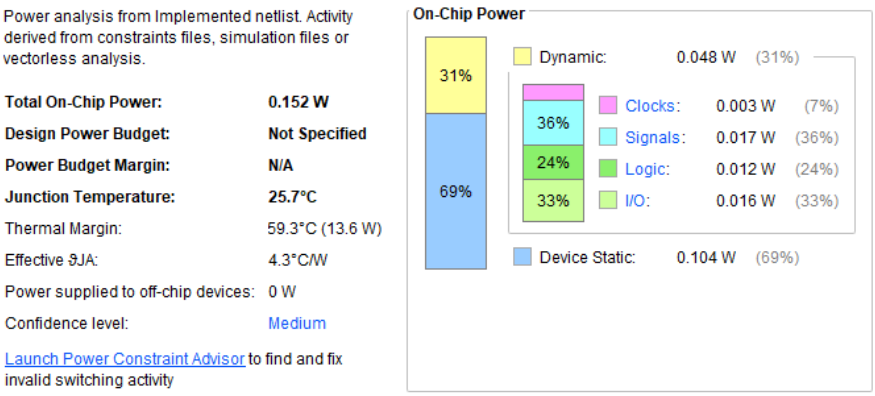

Figure 3.1

The utilization report is presented in Figure 3.2 below. The utilization percentage is low. The processor uses 2029 LUT and 1520 FF.

Resource	Utilization	Available	Utilization %
LUT	2029	63400	3.20
FF	1520	126800	1.20
IO	34	210	16.19

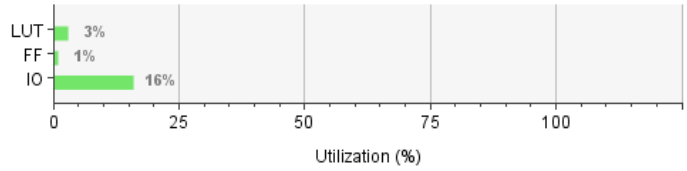

Figure 3.2

The maximum frequency is measured by adding a constraint to the design. The constraint controls the period of the clock. The implementation is repeated for different periods. If the picked period invokes a negative slack (WNS, TNS, etc.), the circuit is not suitable for that period; therefore, the new chosen period is lower. The process is repeated until all the slacks are positive. The Figure 3.3 below represents the slacks of designed processor for 18.4ns period. The chosen period corresponds to 54.35MHz which is the maximum frequency.

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power
✓ synth_1	constrs_1	synth_design Complete!						
✓ impl_1	constrs_1	route_design Complete!	0.021	0.000	0.136	0.000	0.000	0.152

Figure 3.3

The timing summary for the processor is also provided below in Figure 3.4. All of the slacks are positive which means the timing constraints are met.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.021 ns	Worst Hold Slack (WHS):	0.136 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	2685	Total Number of Endpoints:	2685	Total Number of Endpoints:	1521

All user specified timing constraints are met.

Figure 3.4

RISC-SICS Core

Risc_Sics

```
`timescale 1ns / 1ps
module Risc_Sics(input clk, rst, output [31:0] rd_wb_out);
//IF wires
wire [31:0] pc_jb_if_in, pc_if_out, instr_if_out;
wire jb_sel_if_in;
wire sh_sel_if_out;

//IF-ID wires
wire [31:0] pc_if_id_in, instr_if_id_in, pc_if_id_out, instr_if_id_out;
wire stall_bit_if_id_in, stall_bit_if_id_out;
wire instr_flush;

//ID wires
wire [31:0] instr_id_in, pc_id_in, rd_id_in;
wire  RF_W_id_in;
wire [4:0] DA_id_in;
wire [31:0] A_id_out, B_id_out, imm_id_out;
wire [13:0] control_id_out;
wire [4:0] DA_id_out;
wire jb_sel_id_out;
wire [31:0] pc_id_out;
wire jump_branch_hazard_id_out;

//ID-EX wires
wire [13:0] control_id_ex_in;
wire [31:0] A_id_ex_in, B_id_ex_in, pc_id_ex_in, imm_id_ex_in;
wire [4:0] DA_id_ex_in;
wire [13:0] control_id_ex_out;
wire [31:0] A_id_ex_out, B_id_ex_out, pc_id_ex_out, imm_id_ex_out;
wire [4:0] DA_id_ex_out;

//EX wires
wire [31:0] A_ex_in, B_ex_in, imm_ex_in, pc_ex_in, ALU_OUT_ex_out;
wire [1:0] MA_ex_in, MB_ex_in;
wire [3:0] FS_ex_in;

//EX-MEM wires
wire [5:0] control_ex_mem_in, control_ex_mem_out;
wire [4:0] DA_ex_mem_in, DA_ex_mem_out;
wire [31:0] ALU_OUT_ex_mem_in, B_ex_mem_in, ALU_OUT_ex_mem_out, B_ex_mem_out;

//MEM wires
wire [31:0] address_mem_in, data_in_mem_in;
wire[3:0] memory_control_mem_in;
wire [31:0] DM_OUT_mem_out;

//MEM-WB wires
wire [1:0]  control_mem_wb_in;
wire [4:0]  DA_mem_wb_in;
wire [31:0] ALU_OUT_mem_wb_in, DM_OUT_mem_wb_in;
wire [1:0]  control_mem_wb_out;
wire [4:0]  DA_mem_wb_out;
wire [31:0] ALU_OUT_mem_wb_out, DM_OUT_mem_wb_out;

// WB wire
wire rd_sel_wb_in;
wire [31:0] DM_OUT_wb_in, ALU_OUT_wb_in;

// stall wire
wire stall_bit, stall_lu, stall_str;
wire [31:0] pc_sjb;

//RAW Hazards
wire [4:0] rs1_id_ex_in, rs2_id_ex_in;
wire [4:0] rs1_id_ex_out, rs2_id_ex_out;
wire [1:0] forwardA, forwardB;

//Branch_Data wires
wire [1:0] forwardA_id_in, forwardB_id_in;
wire [6:0] opcode_if_id_out;
//IF stage
assign jb_sel_if_in = (jb_sel_id_out && !stall_bit_if_id_out) || stall_bit;
IF if_inst(clk, rst, pc_sjb, jb_sel_if_in, instr_if_out, pc_if_out);


// IF-ID stage
assign pc_if_id_in = pc_if_out;
assign instr_flush = jump_branch_hazard_id_out && !stall_bit_if_id_out;
```

```
MUX mux_inst(instr_flush, instr_if_out, instr_if_id_in);
assign stall_bit_if_id_in = stall_bit;
IF_ID if_id_inst(pc_if_id_in, instr_if_id_in, clk, stall_bit_if_id_in, pc_if_id_out, instr_if_id_out, stall_bit_if_id_out);

//ID stage
assign instr_id_in = instr_if_id_out;
assign pc_id_in = pc_if_id_out;
assign rd_id_in = rd_wb_out;
assign RF_W_id_in = control_mem_wb_out[0];
assign DA_id_in = DA_mem_wb_out;
ID id_inst(clk, rst, forwardA_id_in, forwardB_id_in, ALU_OUT_ex_out, ALU_OUT_mem_wb_in, instr_id_in, pc_id_in, rd_id_in, DA_id_

//ID-EX stage
assign control_id_ex_in = (stall_bit_if_id_out) ? (14'b0) : (control_id_out) ;
assign A_id_ex_in = A_id_out;
assign B_id_ex_in = B_id_out;
assign pc_id_ex_in = pc_if_id_out;
assign imm_id_ex_in = imm_id_out;
assign DA_id_ex_in = DA_id_out;
assign rs1_id_ex_in = (stall_bit_if_id_out) ? (5'b0) : instr_if_id_out [19:15];
assign rs2_id_ex_in = (stall_bit_if_id_out) ? (5'b0) : instr_if_id_out [24:20];
ID_EX id_ex_inst(clk, control_id_ex_in, A_id_ex_in, B_id_ex_in, pc_id_ex_in, imm_id_ex_in, DA_id_ex_in, rs1_id_ex_in, rs2_id_ex
control_id_ex_out, A_id_ex_out, B_id_ex_out, pc_id_ex_out, imm_id_ex_out, DA_id_ex_out, rs1_id_ex_out, rs2_id_ex_out);

//EX stage
assign A_ex_in = A_id_ex_out;
assign B_ex_in = B_id_ex_out;
assign imm_ex_in = imm_id_ex_out;
assign pc_ex_in = pc_id_ex_out;
assign MA_ex_in = control_id_ex_out[13:12];
assign MB_ex_in = control_id_ex_out[11:10];
assign FS_ex_in = control_id_ex_out[9:6];
EX ex_inst(A_ex_in,B_ex_in, imm_ex_in, pc_ex_in, MA_ex_in, MB_ex_in, FS_ex_in, forwardA, forwardB, ALU_OUT_ex_mem_out, rd_wb_ou

//EX-MEM stage
assign control_ex_mem_in = control_id_ex_out[5:0];
assign DA_ex_mem_in = DA_id_ex_out;
assign ALU_OUT_ex_mem_in = ALU_OUT_ex_out;
assign B_ex_mem_in = B_id_ex_out;
EX_MEM ex_mem_inst(clk,control_ex_mem_in, DA_ex_mem_in, ALU_OUT_ex_mem_in, B_ex_mem_in, DA_ex_mem_out, control_ex_mem_out, ALU_

//MEM stage
assign address_mem_in = ALU_OUT_ex_mem_out;
assign memory_control_mem_in = control_ex_mem_out[5:2];
assign data_in_mem_in = B_ex_mem_out;
MEM mem_inst(clk, address_mem_in, data_in_mem_in, memory_control_mem_in, DM_OUT_mem_out);

//MEM-WB stage
assign control_mem_wb_in = control_ex_mem_out[1:0];
assign DA_mem_wb_in = DA_ex_mem_out;
assign ALU_OUT_mem_wb_in = ALU_OUT_ex_mem_out;
assign DM_OUT_mem_wb_in = DM_OUT_mem_out;
MEM_WB mem_wb_inst(clk, control_mem_wb_in, DA_mem_wb_in, ALU_OUT_mem_wb_in, DM_OUT_mem_wb_in, control_mem_wb_out[1],control_mem

//WB stage
assign rd_sel_wb_in = control_mem_wb_out[1];
assign ALU_OUT_wb_in = ALU_OUT_mem_wb_out;
assign DM_OUT_wb_in = DM_OUT_mem_wb_out;
WB wb_in(rd_sel_wb_in, DM_OUT_wb_in, ALU_OUT_wb_in, rd_wb_out);


// Stall Section
Stall stall_inst(stall_bit, pc_if_out, pc_id_out, pc_sjb);
assign stall_bit = stall_lu || stall_str;

// Structural Hazard
Structural_Hazard sh_inst(stall_str, DA_ex_mem_out, instr_if_id_in[24:20], instr_if_id_in[19:15],control_mem_wb_in[0],instr_if_

//RAW Hazard
RAW_Hazard raw_haz_inst(DA_mem_wb_out, DA_ex_mem_out, rs1_id_ex_out, rs2_id_ex_out, control_ex_mem_out[0], control_mem_wb_out[0

//Load-Use Hazard
LU_Hazards lu_haz_inst(control_id_ex_in[4], DA_id_ex_in, instr_if_id_in[24:20], instr_if_id_in[19:15], stall_lu);

//Branch_Data_Hazard
assign opcode_if_id_out = instr_if_id_out[6:0];
branch_data_hazard bdh_inst(forwardA_id_in, forwardB_id_in, opcode_if_id_out, instr_if_id_out[19:15], instr_if_id_out[24:20], D
endmodule
```

IF

```
`timescale 1ns / 1ps
module IF(
input clk, rst,
```

```
input [31:0] PC_jb,
input jb_sel,
output [31:0] instruction, PC_out
);

PC_Block pc1(rst, clk, jb_sel, PC_jb, PC_out );

wire [31:0] inst_out;
(* dont_touch="true" *) INST_MEM #(32, 6) instmem1(PC_out,instruction);

endmodule
```

PC_BLOCK

```
`timescale 1ns / 1ps
module PC_Block(reset, clk, select, PC_bj, PC_out);
input   reset, clk; // reset needed to start the counting
input   [31:0] PC_bj; // calculated PC at the execute stage if branches happens
input   select; // signal comming from a block at ALU letting the PC know if the branch should happen
output  [31:0] PC_out;
reg     [31:0]PC; // creating the 32 bit long PC register
wire    [31:0] Add_out, MUX_out;
ADDER   a1(PC, 4, Add_out); // if branching doesnt happen, the PC should increment by 1 word
                            // or 4 bytes. Since our design uses 32 byte rows as words and
                            // instruciton register holds one instruction per row
MUX2PC    mux1(select,Add_out,PC_bj,MUX_out); // selecting PC + 1 or PC + offset
always  @(posedge clk)
begin
        if(reset) PC <= 0; // used to reset the PC
        else PC <= MUX_out; // updating the PC with the output value of the MUX2
end
assign PC_out = PC;

endmodule
```

INST_MEM

```
module INST_MEM #(parameter integer width = 32, address_bits = 32 ) (

input [31 : 0] pc,
output reg [width - 1:0] instruction);

reg [width - 1 : 0] memory [2**address_bits-1:0]; //RAM-NAME

initial
begin
    $readmemb("instruction_mem.mem", memory);
end

wire [address_bits - 1 : 0] address_bus;

assign address_bus = {2'b00 ,pc[31 : 2]};

always @(*)
begin
    instruction <= memory[address_bus];
end
endmodulemodule INST_MEM #(parameter integer width = 32, address_bits = 32 ) (

input [31 : 0] pc,
output reg [width - 1:0] instruction);

reg [width - 1 : 0] memory [2**address_bits-1:0]; //RAM-NAME

initial
begin
    $readmemb("instruction_mem.mem", memory);
end

wire [address_bits - 1 : 0] address_bus;

assign address_bus = {2'b00 ,pc[31 : 2]};

always @(*)
begin
    instruction <= memory[address_bus];
end
endmodule
```

IF_ID

```
`timescale 1ns / 1ps
module IF_ID(
input [31:0] PC_in, InstMem_in,
input clk, input stall_bit_in,
output reg [31:0] PC_out,InstMem_out,
output reg stall_bit_out
);

//Here, we are assigning input to output in the posedge
//of the clk signal
always@(posedge clk) begin
    PC_out <= PC_in;
    stall_bit_out <= stall_bit_in;
    InstMem_out <= InstMem_in;
end

endmodule
```

ID

```
`timescale 1ns / 1ps
module ID(
input clk, rst,
input [1:0] forwardA, forwardB,
input [31:0] ALU_OUT_ex_out, ALU_OUT_mem_out,
input [31:0] instruction, PC_in,
input [31:0] out_rd_select,
input [4:0] DA2,
input RF_W2,
output [31:0] A, B,
output [31:0] immediate,
output [13:0] control_word,
output [4:0] DA,
output jb_sel,
output [31:0] PC_blck_in,
output jump_branch_hazard
);


assign DA = instruction[11:7]; //Direct connection for data address
wire RF_W;
wire [1:0] MA, MB;
wire [3:0] FS;
wire [3:0] MemCont;
wire rd_sel;
wire [5:0] J_B;
wire [2:0] format_sel;
wire DM_E, DM_RW;
wire [1:0] DM_len;
assign jump_branch_hazard = J_B [5] | jb_sel;
(* dont_touch="true" *) INST_DECODER decod1(instruction, RF_W, MB, MA, FS, DM_E, DM_RW, DM_len, J_B, rd_sel, format_sel);
assign MemCont = {DM_E,DM_RW,DM_len};

assign control_word = {MA, MB, FS, MemCont, rd_sel, RF_W};

wire w_x, w_y, w_z;

wire [31:0] Comparator_A_in, Comparator_B_in;
MUX4 Comparator_A_sel(forwardA, A, ALU_OUT_ex_out, ALU_OUT_mem_out, ,Comparator_A_in);
MUX4 Comparator_B_sel(forwardB, B, ALU_OUT_ex_out, ALU_OUT_mem_out, ,Comparator_B_in);
Compare comp(w_x, w_y, w_z, Comparator_A_in, Comparator_B_in, J_B[0]);

JBSELECT jb(w_x,w_y,w_z,J_B[5],J_B[4],J_B[3],J_B[2],J_B[1],jb_sel);

//Register File
REGISTER_FILE rf(instruction[19:15],instruction[24:20],out_rd_select,DA2,RF_W2, clk, rst, A, B);

(* dont_touch="true" *) IMMGEN immgen1(instruction,format_sel,immediate);

wire [31:0] reg_imm_output, adder_in;
ADDER add1(A, immediate, reg_imm_output);
//ADD_SUB add1(reg_imm_output,,0, A, immediate);
assign PC_blck_in = (J_B[5]&&J_B[0]) ? (reg_imm_output+4) : (adder_in) ;
ADDER add2(PC_in, immediate, adder_in);
//ADD_SUB s3(adder_in,,0, PC_in, immediate); // sending this out signal to the Program Counter block
endmodule
```

INST_DECODER

```
`timescale 1ns / 1ps
module INST_DECODER(
input [31:0] instruction,
output reg RF_W,
output reg [1:0] MB, MA,
output reg [3:0] FS,
output reg DM_E, DM_RW,
output reg [1:0] DM_len,
output reg [5:0] J_B,
output reg rd_select,
output reg [2:0] format_select
);

always@(*)begin
    casex({instruction[30],instruction[14:12],instruction[6:0]})
        //LUI
        11'bx_xxx_0110111:  begin J_B <= 6'b00000; format_select <= 3'b000; MA <= 2'b10; MB <= 2'b01;
                                FS <= 4'b1000; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end
        //AUIPC
        11'bx_xxx_0010111:  begin J_B <= 6'b00000; format_select <= 3'b000; MA <= 2'b00; MB <= 2'b01;
                                FS <= 4'b1000; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end
        //JAL, JALR,
        11'bx_xxx_1101111:  begin J_B <= 6'b100000; format_select <= 3'b001; MA <= 2'b00; MB <= 2'b10;
                                FS <= 4'b1000; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end

        11'bx_000_1100111:  begin J_B <= 6'b100001; format_select <= 3'b011; MA <= 2'b00; MB <= 2'b10;
                                FS <= 4'b1000; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end

        //BEQ, BNE, BLT, BGE, BLTU, BGEU
        11'bx_000_1100011:  begin J_B <= 6'b010001; format_select <= 3'b010; MA <= 2'bxx; MB <= 2'bxx;
                                FS  <= 4'bxxxx; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'bx; RF_W <= 1'b0; end
        11'bx_001_1100011:  begin J_B <= 6'b001001; format_select <= 3'b010; MA <= 2'bxx; MB <= 2'bxx;
                                FS <= 4'bxxxx; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'bx; RF_W <= 1'b0; end
        11'bx_100_1100011:  begin J_B <= 6'b000101; format_select <= 3'b010; MA <= 2'bxx; MB <= 2'bxx;
                                FS <= 4'bxxxx; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'bx; RF_W <= 1'b0; end
        11'bx_101_1100011:  begin J_B <= 6'b000011; format_select <= 3'b010; MA <= 2'bxx; MB <= 2'bxx;
                                FS <= 4'bxxxx; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'bx; RF_W <= 1'b0; end
        11'bx_110_1100011:  begin J_B <= 6'b000100; format_select <= 3'b010; MA <= 2'bxx; MB <= 2'bxx;
                                FS <= 4'bxxxx; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'bx; RF_W <= 1'b0; end
        11'bx_111_1100011:  begin J_B <= 6'b000010; format_select <= 3'b010; MA <= 2'bxx; MB <= 2'bxx;
                                FS <= 4'bxxxx; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'bx; RF_W <= 1'b0; end
        //LB, LH, LW, LBU, LHU
        11'bx_000_0000011: begin J_B <= 6'b0; format_select <= 3'b011; MA <= 2'b01; MB <= 2'b01;
                               FS <= 4'b1000; DM_E <= 1'b1; DM_RW <= 1'b1; DM_len <= 2'b01;
                               rd_select <= 1'b1; RF_W <= 1'b1; end
        11'bx_001_0000011: begin J_B <= 6'b0; format_select <= 3'b011; MA <= 2'b01; MB <= 2'b01;
                               FS <= 4'b1000; DM_E <= 1'b1; DM_RW <= 1'b1; DM_len <= 2'b10;
                               rd_select <= 1'b1; RF_W <= 1'b1; end
        11'bx_010_0000011: begin J_B <= 6'b0; format_select <= 3'b011; MA <= 2'b01; MB <= 2'b01;
                               FS <= 4'b1000; DM_E <= 1'b1; DM_RW <= 1'b1; DM_len <= 2'b11;
                               rd_select <= 1'b1; RF_W <= 1'b1; end
        11'bx_100_0000011: begin J_B <= 6'b0; format_select <= 3'b011; MA <= 2'b01; MB <= 2'b01;
                               FS <= 4'b1000; DM_E <= 1'b1; DM_RW <= 1'b1; DM_len <= 2'b01;
                               rd_select <= 1'b1; RF_W <= 1'b1; end
        11'bx_101_0000011: begin J_B <= 6'b0; format_select <= 3'b011; MA <= 2'b01; MB <= 2'b01;
                               FS <= 4'b1000; DM_E <= 1'b1; DM_RW <= 1'b1; DM_len <= 2'b10;
                               rd_select <= 1'b1; RF_W <= 1'b1; end
        //SB, SH, SW
        11'bx_000_0100011: begin J_B <= 6'b0; format_select <= 3'b100; MA <= 2'b01; MB <= 2'b01;
                               FS <= 4'b1000; DM_E <= 1'b1; DM_RW <= 1'b0; DM_len <= 2'b01;
                               rd_select <= 1'bx; RF_W <= 1'b0; end
        11'bx_001_0100011: begin J_B <= 6'b0; format_select <= 3'b100; MA <= 2'b01; MB <= 2'b01;
                               FS <= 4'b1000; DM_E <= 1'b1; DM_RW <= 1'b0; DM_len <= 2'b10;
                               rd_select <= 1'bx; RF_W <= 1'b0; end
        11'bx_010_0100011: begin J_B <= 6'b0; format_select <= 3'b100; MA <= 2'b01; MB <= 2'b01;
                               FS <= 4'b1000; DM_E <= 1'b1; DM_RW <= 1'b0; DM_len <= 2'b11;
                               rd_select <= 1'bx; RF_W <= 1'b0; end
        //ADDI, SLT, SLTI
        11'bx_000_0010011: begin J_B <= 6'b0; format_select <= 3'b011; MA <= 2'b01; MB <= 2'b01;
                               FS <= 4'b1000; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                               rd_select <= 1'b0; RF_W <= 1'b1; end
        11'bx_010_0010011: begin J_B <= 6'b0; format_select <= 3'b011; MA <= 2'b01; MB <= 2'b01;
                               FS <= 4'b1101; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                               rd_select <= 1'b0; RF_W <= 1'b1; end
        11'bx_011_0010011: begin J_B <= 6'b0; format_select <= 3'b011; MA <= 2'b01; MB <= 2'b01;
```

```
                                FS <= 4'b1100; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end
        //XORI, ORI, ANDI
        11'bx_100_0010011: begin J_B <= 6'b0; format_select <= 3'b011; MA <= 2'b01; MB <= 2'b01;
                                FS <= 4'b0111; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end
        11'bx_110_0010011: begin J_B <= 6'b0; format_select <= 3'b011; MA <= 2'b01; MB <= 2'b01;
                                FS <= 4'b0110; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end
        11'bx_111_0010011: begin J_B <= 6'b0; format_select <= 3'b011; MA <= 2'b01; MB <= 2'b01;
                                FS <= 4'b0101; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end

        //SLLI, SRLI, SRAI
        11'b0_001_0010011: begin J_B <= 6'b0; format_select <= 3'b101; MA <= 2'b01; MB <= 2'b01;
                                FS <= 4'b0000; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end
        11'b0_101_0010011: begin J_B <= 6'b0; format_select <= 3'b101; MA <= 2'b01; MB <= 2'b01;
                                FS <= 4'b0001; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end
        11'b1_101_0010011: begin J_B <= 6'b0; format_select <= 3'b101; MA <= 2'b01; MB <= 2'b01;
                                FS <= 4'b0010; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end

        //ADD, SUB
        11'b0_000_0110011: begin J_B <= 6'b0; format_select <= 3'bxxx; MA <= 2'b01; MB <= 2'b00;
                                FS <= 4'b1000; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end
        11'b1_000_0110011: begin J_B <= 6'b0; format_select <= 3'bxxx; MA <= 2'b01; MB <= 2'b00;
                                FS <= 4'b1001; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end

        //SLL, SLT, SLTU
        11'b0_001_0110011: begin J_B <= 6'b0; format_select <= 3'bxxx; MA <= 2'b01; MB <= 2'b00;
                                FS <= 4'b0000; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end
        11'b0_010_0110011: begin J_B <= 6'b0; format_select <= 3'bxxx; MA <= 2'b01; MB <= 2'b00;
                                FS <= 4'b1101; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end
        11'b0_011_0110011: begin J_B <= 6'b0; format_select <= 3'bxxx; MA <= 2'b01; MB <= 2'b00;
                                FS <= 4'b1100; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end

        //XOR
        11'b0_100_0110011: begin J_B <= 6'b0; format_select <= 3'bxxx; MA <= 2'b01; MB <= 2'b00;
                                FS <= 4'b0111; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end

        //SRL, SRA
        11'b0_101_0110011: begin J_B <= 6'b0; format_select <= 3'bxxx; MA <= 2'b01; MB <= 2'b00;
                                FS <= 4'b0001; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end
        11'b1_101_0110011: begin J_B <= 6'b0; format_select <= 3'bxxx; MA <= 2'b01; MB <= 2'b00;
                                FS <= 4'b0010; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end

        // OR, AND
        11'b0_110_0110011: begin J_B <= 6'b0; format_select <= 3'bxxx; MA <= 2'b01; MB <= 2'b00;
                                FS <= 4'b0110; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end
        11'b0_111_0110011: begin J_B <= 6'b0; format_select <= 3'bxxx; MA <= 2'b01; MB <= 2'b00;
                                FS <= 4'b0101; DM_E <= 1'b0; DM_RW <= 1'bx; DM_len <= 2'bxx;
                                rd_select <= 1'b0; RF_W <= 1'b1; end
        default :          begin J_B <= 6'b0; format_select <= 3'b000; MA <= 2'b00; MB <= 2'b00;
                                FS <= 4'b0000; DM_E <= 1'b0; DM_RW <= 1'b0; DM_len <= 2'b00;
                                rd_select <= 1'b0; RF_W <= 1'b0; end

    endcase
end

endmodule
```

JBSELECT

```
`timescale 1ns/1ps
module JBSELECT(
input x,y,z,
input jal, beq, bnq, blt, bge,
output out);

//Here, we are selecting proper path for Jump/Branch instructions.
assign out = (beq&y)|(bnq&(!y))|(blt&z)|(bge&!z)|(jal);
endmodule
```

REGISTER_FILE

```
`timescale 1ns / 1ps
module REGISTER_FILE(
input  [4:0] A_sel, B_sel,
input [31:0] D_data,
input [4:0] D_address,
input Load_enab, clk, rst,
output [31:0] A, B
);
//SELECT REGISTER PART
wire [31:0] registers [31:0];
MUX32 muxA(clk, A_sel,32'b0,
registers[1],registers[2],registers[3],registers[4],registers[5],registers[6],registers[7],
registers[8],registers[9],registers[10],registers[11],registers[12],registers[13],registers[14],registers[15],
registers[16],registers[17],registers[18],registers[19],registers[20],registers[21],registers[22],registers[23],
registers[24],registers[25],registers[26],registers[27],registers[28],registers[29],
registers[30],registers[31], A);
MUX32 muxB(clk, B_sel,32'b0,
registers[1],registers[2],registers[3],registers[4],registers[5],registers[6],registers[7],
registers[8],registers[9],registers[10],registers[11],registers[12],registers[13],registers[14],registers[15],
registers[16],registers[17],registers[18],registers[19],registers[20],registers[21],registers[22],registers[23],
registers[24],registers[25],registers[26],registers[27],registers[28],registers[29],
registers[30], registers[31], B);

//DECODER PART
wire [31:0] D_select;
DECODER_5_32 dec1(D_address,D_select);

assign registers[0] = 32'b0;

//GENERATION OF REGISTERS PART
genvar i;
generate
    for (i=1;i<32;i=i+1)
    begin
        REGISTER reg_inst(D_data,Load_enab & D_select[i], clk, rst, registers[i]);
    end
endgenerate
endmodule
```

REGISTER

```
`timescale 1ns / 1ps
module REGISTER(
input [31:0] data_in,
input load, clk, rst,
output reg [31:0] out
);

always@(posedge clk) begin
    if(load) begin
        out <= data_in; // only store if the decoded signal is 1
    end
end

endmodule
```

DECODER_5_32

```
`timescale 1ns / 1ps
module DECODER_5_32(
input [4:0] select,
output reg [31:0] out
);

always @(*) begin
    out <= 1<<select;
end

endmodule
```

IMMGEN

```
`timescale 1ns / 1ps
module IMMGEN(
input [31:0] InstructionIn,
```

```
input [2:0] TypeSel,
output reg [31:0] ImmOut);

   always@(*) begin
        case(TypeSel)
            3'b011: ImmOut = {{21{InstructionIn[31]}},InstructionIn[30:20]}; //I-type
            3'b101: ImmOut = {{27{InstructionIn[31]}},InstructionIn[24:20]}; //Special I-type, detects shift amount (shamt)
            3'b100: ImmOut = {{21{InstructionIn[31]}},InstructionIn[30:25],InstructionIn[11:7]}; //S-type
            3'b010: ImmOut = {{20{InstructionIn[31]}},InstructionIn[7],InstructionIn[30:25],InstructionIn[11:8],1'b0}; //B-type
            3'b000: ImmOut = {InstructionIn[31:12],12'b0}; //U-type
            3'b001: ImmOut = {{12{InstructionIn[31]}},InstructionIn[19:12],InstructionIn[20],InstructionIn[30:21],1'b0}; //J-ty
            default: ImmOut = 32'd0;
         endcase
    end
endmodule
```

ID_EX

```
`timescale 1ns / 1ps
module ID_EX(
input clk,
input [13:0] Ctrl_in,
input [31:0] REG_A_in, REG_B_in, PC_in, Imm_Gen_in,
input [4:0] InstMem_in, rs1_in, rs2_in,
output reg [13:0] Ctrl_out,
output reg [31:0] REG_A_out, REG_B_out, PC_out, Imm_Gen_out,
output reg [4:0] InstMem_out, rs1_out, rs2_out
);

 always @(posedge clk) begin
    Ctrl_out <= Ctrl_in;
    InstMem_out <= InstMem_in;
    PC_out <= PC_in;
    REG_A_out <= REG_A_in;
    REG_B_out <= REG_B_in;
    Imm_Gen_out <= Imm_Gen_in;
    rs1_out <= rs1_in;
    rs2_out <= rs2_in;
 end

endmodule
```

EX

```
`timescale 1ns / 1ps
module EX(
input [31:0] A, B,
input [31:0] immediate,
input [31:0] PC_in,
input [1:0] MA, MB,
input [3:0] FS,
input [1:0] forwardA, forwardB,
input [31:0] ALU_OUT_ex_mem_out, rd_wb_out,
output [31:0] ALUout
);

wire [31:0] ALU_A, ALU_B;
MUX4 ma_select(MA, PC_in ,A, 32'd0, , ALU_A);
MUX4 mb_select(MB, B, immediate,32'd0 , , ALU_B);

wire [31:0] ALU_A_in, ALU_B_in;
MUX4 RAW_A_sel(forwardA, ALU_A, ALU_OUT_ex_mem_out, rd_wb_out, ,ALU_A_in );
MUX4 RAW_B_sel(forwardB, ALU_B, ALU_OUT_ex_mem_out, rd_wb_out, ,ALU_B_in );

(* dont_touch="true" *) ALU alu(ALUout, , , , , FS, ALU_A_in, ALU_B_in);


endmodule
```

MUX4

```
`timescale 1ns/1ps
module MUX4(
input [1:0] select,
input [31:0] in1, in2, in3, in4,
output reg [31:0] out);

always@(*) begin
```

```
    case(select)
        0: out <= in1;
        1: out <= in2;
        2: out <= in3;
        3: out <= in4;
    endcase
end

endmodule
```

ALU

```
module ALU(ALU_OUT,Zero,x, y, z, select, A, B);
output reg [31:0] ALU_OUT;
output Zero;
output x,y,z;
input  [3:0] select;
input  [31:0] A, B;
wire   [31:0] w_out [2:0];
wire w_x, w_y, w_z, C_out;
SHIFTER s1(A, B, select[1:0], w_out[0]); //
LOGIC   s2(A, B, select[1:0], w_out[1]);
ADD_SUB s3(w_out[2],C_out,select[0], A, B);
Compare s4(w_x,w_y,w_z, A, B, select[0]);
assign x = w_x;
assign y = w_y;
assign z = w_z;
always @(*)
     begin
      case(select[3:2])
       2'd0: ALU_OUT = w_out[0];
       2'd1: ALU_OUT = w_out[1];
       2'd2: ALU_OUT = w_out[2];
       2'd3: ALU_OUT = {31'b0,w_z};
    endcase
end
assign Zero = ALU_OUT | 0 ? 0 : 1;

endmodule
```

SHIFTER

```
`timescale 1ns / 1ps
module SHIFTER #(parameter width=32)(
input [width-1:0] in1, in2,
input [1:0] sel,
output reg [width-1:0] out
);

wire [width-1:0] out1,out2,out3;
SLL #(width)SLL1(in1,in2[4:0],out1);
SRL #(width)SRL1(in1,in2[4:0],out2);
SRA #(width)SRA1(in1,in2[4:0],out3);

always @(*) begin
    case(sel)
        2'b00: out <= out1;
        2'b01: out <= out2;
        2'b10: out <= out3;
        2'b11: out <= 0;
    endcase
end

endmodule
```

LOGIC

```
module LOGIC #(parameter width = 32)(
input [width - 1:0] in1, in2,
input [1:0] sel,
output reg [width-1:0] out
    );

    wire [width-1:0] out1,out2,out3,out4;
    NOT #(width) NOT1(in1, out1);
    AND #(width) AND1(in1, in2, out2);
    OR  #(width) OR1(in1, in2, out3);
    XOR #(width) XOR1(in1, in2, out4);
```

```
    always @(*)
     begin
      case(sel)
       2'b00: out <= out1; //NOT
       2'b01: out <= out2; //AND
       2'b10: out <= out3; //OR
       2'b11: out <= out4; //XOR
      endcase
     end

endmodule
```

ADD_SUB

```
module ADD_SUB #(parameter Width = 32)(Sum, Cout, Select, A, B);
output [Width - 1 : 0] Sum;
output Cout;
input [Width - 1 : 0] A, B;
input Select;
wire [Width - 1 : 0]  B_i;
CLA CLA1(Sum, Cout, A, B_i, Select);
assign B_i = (Select == 1) ? (~B):(B); //if select == 0 --> addition, else subtraction
endmodule
```

COMPARE

```
module Compare(x,y,z, A, B, select);
output reg x, y, z;
input [31:0] A,B;
input select;
always @(*)
begin
{x,y,z} <= 3'b000;
   if(select == 0) //unsigned
         begin
            if(A > B) x <= 1;
            else if(A == B) y <= 1;
            else z <= 0;
         end
   else     //signed
         begin
            if($signed(A) > $signed(B)) x <= 1;
            else if ($signed(A) == $signed(B)) y <= 1;
            else z <= 1;
         end
end
endmodule
```

EX_MEM

```
`timescale 1ns / 1ps
module EX_MEM(
input clk,
input [5:0] control,
input [4:0] DA,
input [31:0] ALU_out, B,
output reg [4:0] DA2,
output reg [5:0] control_out,
output reg [31:0] ALU_out2, B2
);

always@(posedge clk) begin

    DA2 <= DA;
    ALU_out2 <= ALU_out;
    B2 <= B;
    control_out <= control;
end

endmodule
```

MEM

```
`timescale 1ns / 1ps
module MEM(
```

```
input clk,
input [31:0] address, DataIn,
input [3:0] MemCont,
output [31:0] DataMem_out//address_out,
);

//Data memory implementation
(* dont_touch="true" *) DATA_MEM2 data_mem(clk, MemCont[3],  MemCont[2:0], address, DataIn, DataMem_out);

endmodule
```

DATA_MEM

```
module DATA_MEM2 #(parameter integer RAM_WIDTH = 8, BUS_WIDTH = 4 )(
input clk, mem_enab,
input [2:0] mem_cont,
input [31:0] mem_accs_addr,
input [31:0] mem_wrt_data,
output reg [31:0] mem_read_data
);

reg [RAM_WIDTH-1:0] memory [(2**BUS_WIDTH)-1:0];
wire [BUS_WIDTH-1:0] ram_adrr = mem_accs_addr [BUS_WIDTH-1:0];

initial begin
    $readmemb("data_memory.mem",memory,0,2**BUS_WIDTH-1);
end

reg [(4*RAM_WIDTH)-1:0] mem_out;
always @(posedge clk) begin
if(mem_enab)
    begin

        if(mem_cont == 3'b001)
        begin
            memory[ram_adrr] <= mem_wrt_data[7:0];
        end
        else if(mem_cont == 3'b010)
        begin
            memory[ram_adrr] <= mem_wrt_data[7:0];
            memory[ram_adrr+1] <= mem_wrt_data[15:8];
        end
        else if(mem_cont == 3'b011)
        begin
            memory[ram_adrr] <= mem_wrt_data[7:0];
            memory[ram_adrr+1] <= mem_wrt_data[15:8];
            memory[ram_adrr+2] <= mem_wrt_data[23:16];
            memory[ram_adrr+3] <= mem_wrt_data[31:24];
        end
    end
end

always @(*) begin
    if(mem_cont == 3'b101)
    begin
        mem_read_data = {24'b0,memory[ram_adrr]};
    end
    else if(mem_cont == 3'b110)
    begin
        mem_read_data = {16'b0,memory[ram_adrr+1],memory[ram_adrr]};
    end
    else if(mem_cont == 3'b111)
    begin
        mem_read_data = {memory[ram_adrr+3],memory[ram_adrr+2],memory[ram_adrr+1],memory[ram_adrr]};
    end
    else
    begin
        mem_read_data = {24'b0,memory[ram_adrr]};
    end
end
endmodule
```

MEM_WB

```
`timescale 1ns / 1ps
module MEM_WB(
input clk,
input [1:0] Ctrl_in,
input [4:0] InstMem_in,
input [31:0] ALU_in, DataMem_in,
output reg rd_sel,
```

```
output reg rf_w,
output reg [4:0] InstMem_out,
output reg [31:0] ALU_out, DataMem_out
);

 always @(posedge clk) begin
    rd_sel <= Ctrl_in[1];
    InstMem_out <= InstMem_in;
    ALU_out <= ALU_in;
    DataMem_out <= DataMem_in;
    rf_w <= Ctrl_in[0];
 end

endmodule
```

WB

```
`timescale 1ns / 1ps
module WB(rd_select, MEM, ALU, rd);
input   rd_select;
input   [31:0] MEM;
input   [31:0] ALU;
output  [31:0] rd;

assign  rd = (rd_select == 0) ? (ALU) : (MEM);

endmodule
```

STALL

```
`timescale 1ns / 1ps

module Stall(stall_bit, pc_if_out, pc_id_out, pc_if_in);
input stall_bit;
input [31:0] pc_if_out, pc_id_out;
output[31:0] pc_if_in;

assign pc_if_in = (stall_bit) ? (pc_if_out):(pc_id_out);

endmodule
```

STRUCTURAL_HAZARD

```
module Structural_Hazard(output reg select, input [4:0] D_Address, rs2, rs1, input RF_W, input [6:0] opcode_if_id_in);

always @(*)
begin
    case (RF_W)
    1:  begin

        if(D_Address !=0 && D_Address == rs1 && D_Address == rs2) select = 1;
        else if (opcode_if_id_in == 0) select = 0;
        else select = 0;
        end
    default: select = 0;
    endcase
end

endmodule
```

RAW_HAZARD

```
module RAW_Hazard(DA_mem_wb_out, DA_ex_mem_out,  A_id_ex_out, B_id_ex_out, RF_W_ex_mem_out, RF_W_mem_wb_out, forwardA, forwardB
input RF_W_ex_mem_out, RF_W_mem_wb_out;
input [4:0] DA_mem_wb_out, DA_ex_mem_out, A_id_ex_out, B_id_ex_out;
output reg [1:0] forwardA, forwardB;

always @(*)
begin

    if(RF_W_ex_mem_out == 1 && DA_ex_mem_out != 0 && DA_ex_mem_out == A_id_ex_out) forwardA <= 2'b01;
    else if(RF_W_mem_wb_out == 1 && DA_mem_wb_out != 0 && DA_mem_wb_out == A_id_ex_out && DA_ex_mem_out != A_id_ex_out) forward
    else forwardA <= 2'b00;

    if(RF_W_ex_mem_out == 1 && DA_ex_mem_out != 0 && DA_ex_mem_out == B_id_ex_out) forwardB <= 2'b01;
    else if(RF_W_mem_wb_out == 1 && DA_mem_wb_out != 0 && DA_mem_wb_out == B_id_ex_out && DA_ex_mem_out != B_id_ex_out) forward
```

```
    else forwardB <= 2'b00;

end
endmodule
```

LU_HAZARD

```
module LU_Hazards(
input DM_RW_id_ex_in,
input [4:0] DA_if_id_out, B_if_id_in, A_if_id_in,
output reg stall);

always @(*)
begin
    if(DM_RW_id_ex_in == 1 && ((DA_if_id_out == A_if_id_in) || (DA_if_id_out == B_if_id_in))) stall = 1'b1;
    else stall = 1'b0;
end

endmodule
```

BRANCH_DATA_HAZARD

```
module branch_data_hazard(forwardA_id_in, forwardB_id_in,
                              opcode_if_id_out, A_id_out, B_id_out,
                              DA_ex_mem_in, DA_mem_wb_in, RF_ex_mem_in, RF_mem_wb_in);
input   [6:0] opcode_if_id_out;
input   RF_ex_mem_in, RF_mem_wb_in;
input   [4:0] A_id_out, B_id_out, DA_ex_mem_in, DA_mem_wb_in;
output reg  [1:0] forwardA_id_in, forwardB_id_in;

always @(*)
begin

    if(RF_ex_mem_in == 1 && opcode_if_id_out == 7'b1100011 && DA_ex_mem_in != 0 && DA_ex_mem_in == A_id_out) forwardA_id_in <=
    else if(RF_mem_wb_in == 1 && opcode_if_id_out == 7'b1100011 && DA_mem_wb_in != 0 && DA_mem_wb_in == A_id_out && DA_ex_mem_i
    else forwardA_id_in <= 2'b00;

    if(RF_ex_mem_in == 1 && opcode_if_id_out == 7'b1100011 && DA_ex_mem_in != 0 && DA_ex_mem_in == B_id_out) forwardB_id_in <=
    else if(RF_mem_wb_in == 1 && opcode_if_id_out == 7'b1100011 && DA_mem_wb_in != 0 && DA_mem_wb_in == B_id_out && DA_ex_mem_i
    else forwardB_id_in <= 2'b00;

end
endmodule
```