

## CS F342 COMPUTER ARCHITECTURE

### ASSIGNMENT 1

Implement 5-stage pipelined processor in Verilog. This processor supports **load (lw), store (sw), or immediate (ori), load upper immediate (lui), multiply (mul) and jump to register (jr) instructions only**. The processor should implement forwarding to resolve data hazards. The processor has Reset, CLK as inputs and no outputs. The processor has instruction fetch unit, decode, **reg read (with 32 32-bit registers)**, execution, memory and writeback units. The processor also contains four pipelined registers IF/ID, ID/EX, EX/MEM and MEM/WB. When reset is activated the PC, IF/ID, ID/EX, EX/MEM and MEM/WB registers are initialized to 0, the instruction memory and register file get loaded by predefined values.

When the instruction unit starts fetching the first instruction the pipelined registers contain unknown values. When the second instruction is being fetched in the IF unit, the IF/ID register will hold the instruction code for the first instruction. When the third instruction is being fetched by the IF unit, the IF/ID register contains the instruction code of the second instruction, the ID/EX register contains information related to the first instruction and so on. (Assume a 32-bit PC. Also Assume Address and Data size as 32-bit).

The instruction and its 32-bit instruction format are shown below:

**lw destinationReg, offset [sourceReg]** (Sign extends data specified in instruction field (15:0) to 32-bits, add it with register specified by register number in rs field and store the data corresponding to the memory location defined by the calculated address in the rt register. Opcode for lw is 100011).

lw R1, R11, #12

op	rs	rt	offset
6 bits (31-26)	5-bits (25-21)	5-bits (20-16)	16-bits (15-0)

Address-10001101 01100001 00000000 00001100

**sw sourceReg, offset [destinationReg]** (Sign extends data specified in instruction field (15:0) to 32-bits, add it with register specified by register number in rs field. Opcode for sw is 101011).

sw R4, 4[R5]

op	rs	rt	offset
6 bits (31-26)	5-bits (25-21)	5-bits (20-16)	16-bits (15-0)

10101101 00001010 00000000 00000100

10101100 100 00101 00000000 00000100

**lui destinationReg, sourceReg, immediate** (loads the highest 16 bits of the register rt with a constant (immediate value), and clears the lowest 16 bits to zeros. Opcode for lui is 001111.)

lui R2, R8, #8

op	rs	rt	immediate
6 bits (31-26)	5-bits (25-21)	5-bits (20-16)	16-bits (15-0)

Address-00111101 00000010 00000000 00001000

**ori destinationReg, sourceReg, immediate** (Sign extends data specified in instruction field (15:0) to 32-bits, or it with register specified by register number in rs field. And store the result in rt. Opcode for ori is 001110).

op	rs	rt	immediate
6 bits (31-26)	5-bits (25-21)	5-bits (20-16)	16-bits (15-0)

**mul destinationReg, sourceReg, targetReg** (Performs signed multiplication between targetReg and sourceReg. After this operation, result should be stored in destinationReg. Opcode for mul is 011010).

mul R2, R1, R8

mul R6, R6, R6

op	rs	rt	rd	shamt	funct
6 bits (31-26)	5-bits (25-21)	5-bits (20-16)	5-bits (15-11)	5-bits (10-6)	6-bits (5-0)

01101000 00101000 00010000 00000000

01101000 11000110 00110000 00000000

**jr sourceReg** (Jumps to an address stored in register rs. Opcode for j is 000010, take funct to bits as 001000).

jr R12

op	rs	rt	rd	shamt	funct
6 bits (31-26)	5-bits (25-21)	5-bits (20-16)	5-bits (15-11)	5-bits (10-6)	6-bits (5-0)

00001001 10000000 00000000 00001000

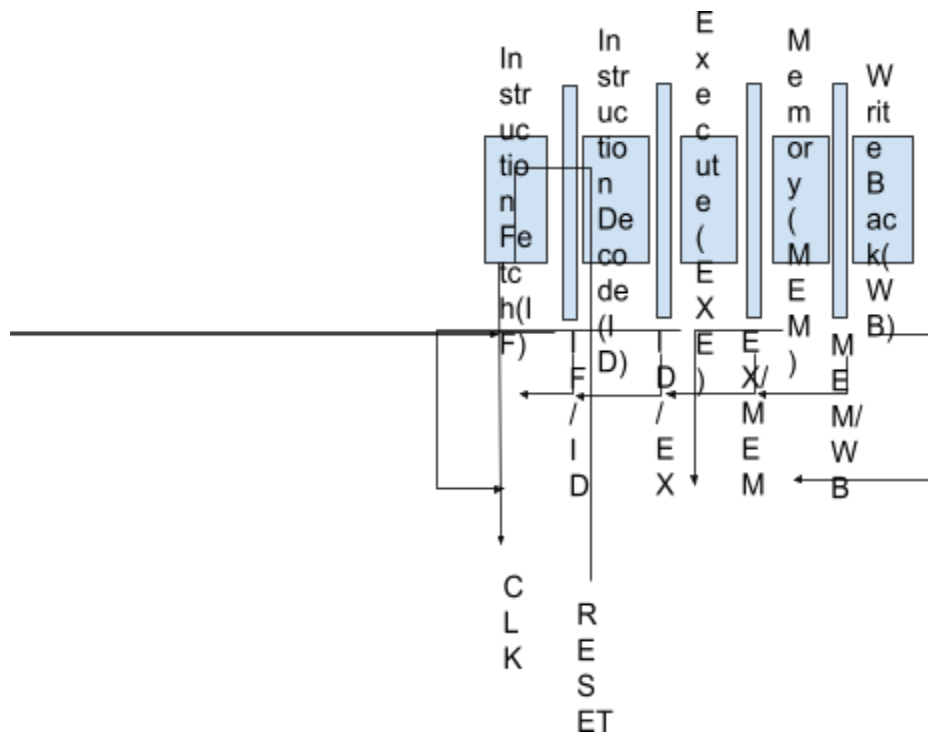
Assume the register file contains 32 registers (R0-R31) each register can hold 32-bit data. On reset, PC and all register file registers should get initialized to 0. Ensure r0 is always zero. Each

location in DMEM has 8-bit data. So, to store a 32-bit value, you need 4 locations in the DMEM, stored in big-endian format. Also ensure that on reset, the instruction memory gets initialized with the following instructions, starting at address 0:

```
lw R1, R11, #12
lui R2, R8, #8
mul R2, R1, R8
jr R12
mul R6, R6, R6
L1: sw R4, 4[R5]
```

The above code should run correctly on the processor implementation. Ensure that you handle the data hazards present, if any.

A partial block level representation of the 5-stage pipelined processor is shown below. **Please note that for registerfile implementation, write should be on the positive edge and read should be on the negative edge of the clock.** Write operation depends on the control signal.



As part of the assignment three files should be submitted in a zipped folder.

1. PDF version of this Document with all the Questions below answered with file name as **IDNO\_NAME.pdf.**
2. Design Verilog Files for all the Sub-modules (instruction fetch, Register file, forwarding unit).
3. Design Verilog file for the main processor.

The name of the zipped folder should be in the format **IDNO\_NAME.zip**

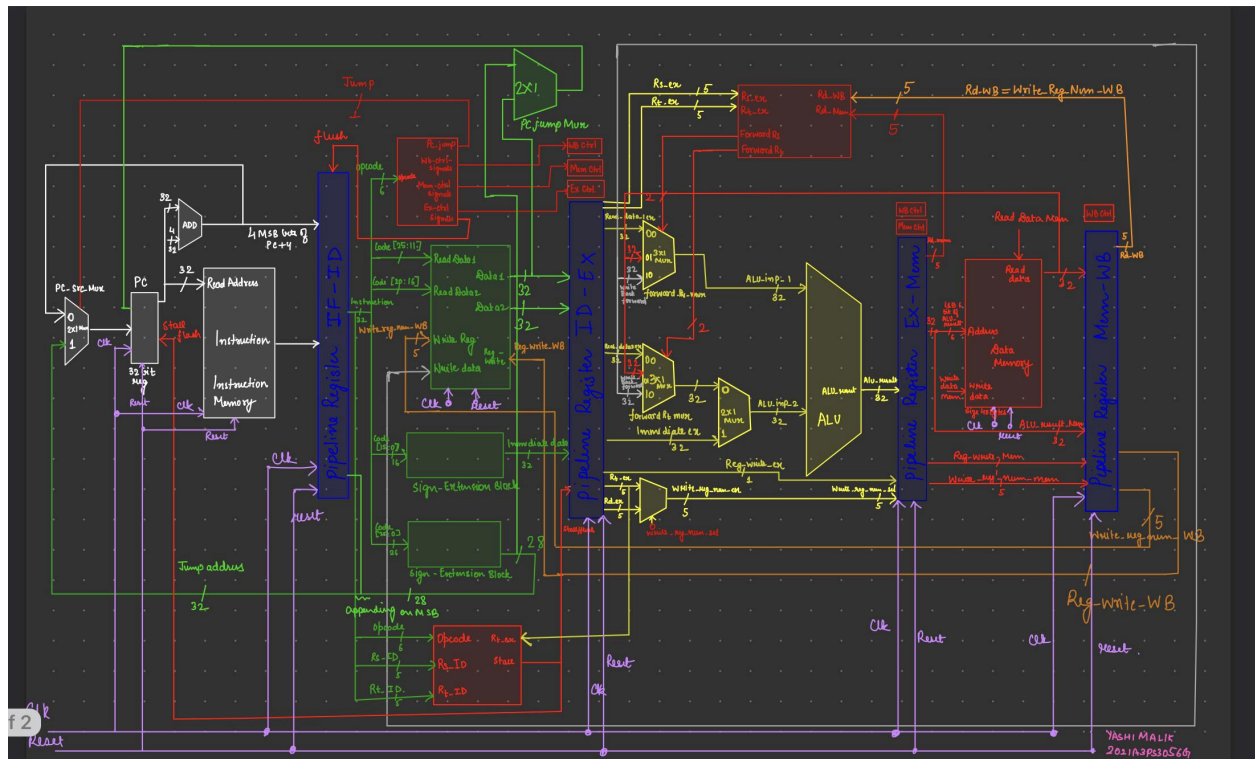
NAME: Yashi Malik

ID No: 2021A3PS3056G

### Questions Related to Assignment

1. Draw the complete Datapath and show control signals of the 5-stage pipelined processor. A sample Datapath for 5-stage pipelined MIPS processor has been discussed in class. A ppt named Assignmenthelp.ppt contains this 5-stage processor and is uploaded in CMS. You can modify this according to your specification.

Answer:



2. List the control signals used and also the values of control signals for different instructions in a tabular format as follows:

Answer:

Instructions	Control Signals								
	flush	PC_Src	reg_write_ID	Write_regnum_Src_sel_line_ID	ALU_Src_sel_line_ID	ALU_ctrl_ID	data_write_ID	write_Data_Src_mux_ID	data_read_ID
lw	0	0	1	0	1	4'b0001	0	1	1
sw	0	0	0	1'bx	1	4'b0001	1	1'bx	0
lui	0	0	1	0	1	4'b0010	0	0	0
ori	0	0	1	0	1	4'b0010	0	0	0
jr	1	1	0	0	0	4'b0000	0	0	0
mul	0	0	1	1	0	4'b0011	0	0	0

3. In a program, there are 25% load instructions, 1/x of which are immediately followed by an instruction that uses a result, requiring a stall. 10% are stores. 50% are R-type. 10% are branch, 1/y of which are taken. 5% are jumps. What is the average CPI of this program? If the number of instructions are  $10^9$ , and the clock cycle is 100 ps, how much time does a MIPS single cycle pipelined processor take to execute all instructions? Assume the processor always predicts branch not-taken.

Where x, y, z are related to last 3 digits of your ID No.

If ID number: 20XXXXXXABCG, then  $x = (A \% 8) + 1$ ,  $y = ((B + 2) \% 8) + 1$ , and  $z = ((C + 3) \% 8) + 1$ .

Answer: last 3 digits-056

$x=1$ ,  $y=8$ ,  $z=2$

Assuming the correct target for Jump and Branch instructions are determined in the execute stage.

$$CPI = 1 + (0.25 \times 1 \times 1) + (0.1 \times 0.125 \times 2) + (0.05 \times 2)$$

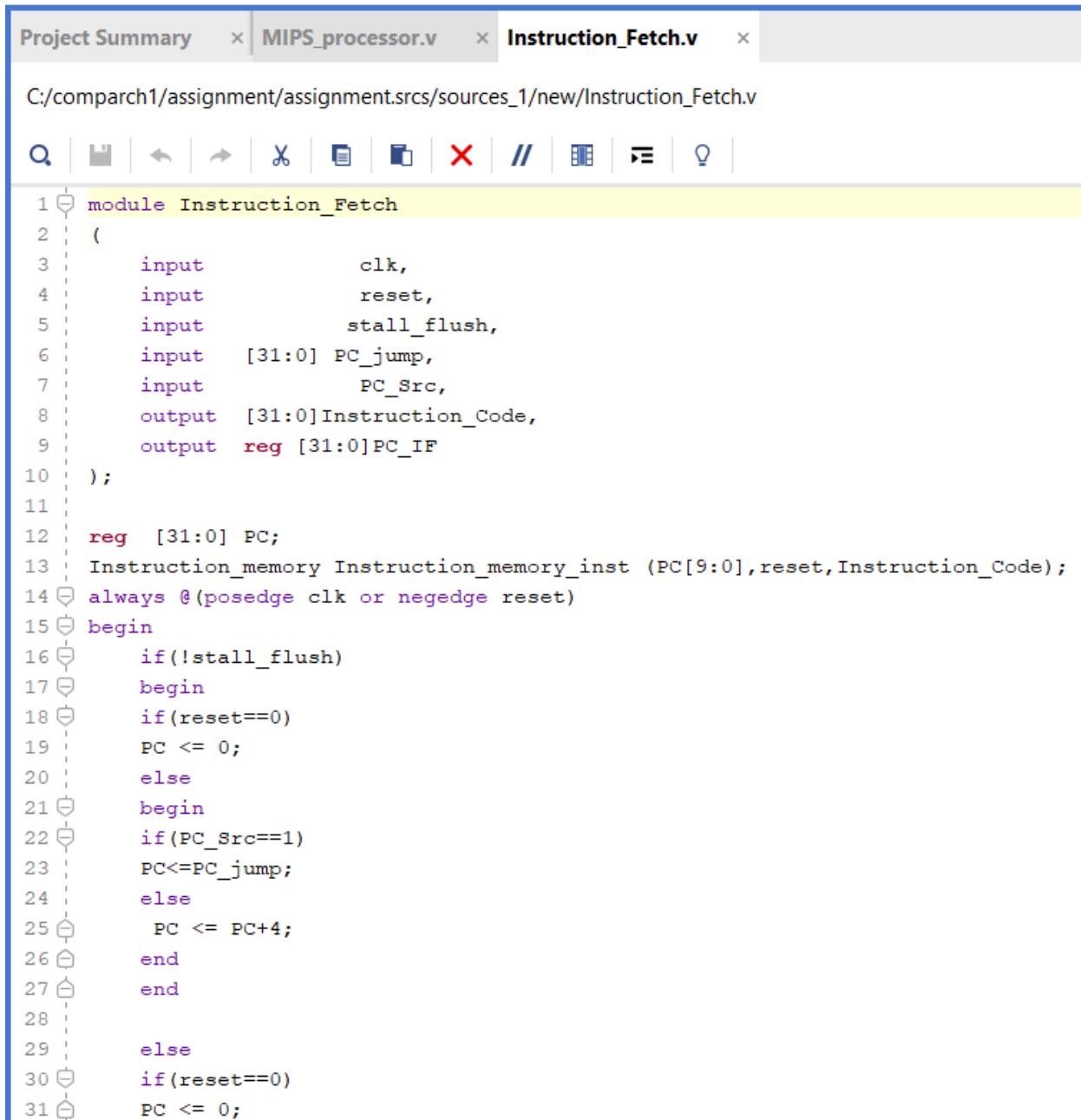
$$CPI = 1.375$$

$$\text{Execution time} = CPI \times \text{Cycle time} \times \text{No. of Instructions}$$

$$= 1.375 \times 100\text{ps} \times 10^9$$

$$\text{Execution time} = 0.1375 \text{ seconds}$$

4. Implement the Instruction Fetch block. Copy the image of Verilog code of the Instruction fetch block here



```
1 module Instruction_Fetch
2 (
3     input        clk,
4     input        reset,
5     input        stall_flush,
6     input  [31:0] PC_jump,
7     input        PC_Src,
8     output  [31:0] Instruction_Code,
9     output reg  [31:0] PC_IF
10 );
11
12 reg  [31:0] PC;
13 Instruction_memory Instruction_memory_inst (PC[9:0], reset, Instruction_Code);
14 always @(posedge clk or negedge reset)
15 begin
16     if(!stall_flush)
17     begin
18         if(reset==0)
19             PC <= 0;
20         else
21         begin
22             if(PC_Src==1)
23                 PC<=PC_jump;
24             else
25                 PC <= PC+4;
26         end
27     end
28
29     else
30         if(reset==0)
31             PC <= 0;
```

Answer:

```
32 end
33
34 always @(*) begin
35     PC_IF <= PC;
36 end
37
38
39 endmodule
40
```

5. Implement the Instruction Decode block. Copy the image of Verilog code of the Instruction decode block here

Answer:

C:/comparch1/assignment/assignment.srscs/sources\_1/new/MIPS\_processor.v



```
61 // ID Unit
62
63 //Select-lines All#####
64
65 //sel_lines_IF.....
66 //PC_Src not going further but comes from control unit
67
68 //sel_lines_ID.....
69 //reg_write.
70 wire reg_write_ID; //Comes directly form CU
71 wire reg_write_EX;
72 wire reg_write_MEM;
73 wire reg_write_WB; //USED
74
75 //sel_lines_EX.....
76 //Write_regnum_Src_sel_line
77 wire Write_regnum_Src_sel_line_ID; //Comes directly form CU
78 wire Write_regnum_Src_sel_line_EX; //USED
79
80 //ALU_Src_sel_line_EX
81 wire ALU_Src_sel_line_ID; //Comes directly form CU
82 wire ALU_Src_sel_line_EX; //USED
83
84 //ALU_ctrl_EX
85 wire [3:0]ALU_ctrl_ID; //Comes directly form CU
86 wire [3:0]ALU_ctrl_EX; //USED
87
88
89 //sel_lines_MEM.....
90 //data_write_MEM
91 wire data_write_ID; //Comes directly form CU
```



```

92 wire data_write_EX;
93 wire data_write_MEM; //USED
94
95 //sel_lines_WB.....
96 wire write_Data_Src_mux_ID; //Comes directly form CU
97 wire write_Data_Src_mux_EX;
98 wire write_Data_Src_mux_MEM;
99 wire write_Data_Src_mux_WB; //USED
100
101 //#####
102 //Wires ip/op ID
103 wire [25:0]ip_Sign_ext_offset_ID_25;
104 wire [27:0]op_Sign_ext_offset_ID_25;
105 //Wires op ID
106 wire [31:0]Write_Data_WB;
107
108 wire [31:0]op_Sign_ext_offset_ID_15;
109 wire [31:0]Read_Data_1_ID;
110 wire [31:0]Read_Data_2_ID;
111
112 assign ip_Sign_ext_offset_ID_25={Rs_ID,Rt_ID,ip_Sign_ext_offset_ID_15[15:0]};
113 // sign extention blk 15->31
114 Sign_ext_blk_15 Sign_ext_blk_15 (ip_Sign_ext_offset_ID_15,op_Sign_ext_offset_ID_15);
115 // Register file instantiation
116 wire[4:0] Reg_write_num_WB;
117 wire [31:0]Final_Write_Data; //Coming form Set_less_than_Writing_mitigation_mux.
118 Register_file_block Register_file_inst(clk,reset,Rs_ID,Rt_ID,Reg_write_num_WB,
119 Final_Write_Data,reg_write_WB,Read_Data_1_ID,Read_Data_2_ID);
120 //jr
121 wire [31:0] PC_jump_ID_Im, PC_jump_ID_reg;

122 wire PC_jump_ID_select;
123 assign PC_jump_ID_select=(Instruction_Code_ID[31:26]==6'b000010)?1:0;
124 assign PC_jump_ID_reg=Read_Data_1_ID;
125
126 //...instantiating Sign_ext_blk_25
127 Sign_ext_blk_25 Sign_ext_blk_25_inst (ip_Sign_ext_offset_ID_25,op_Sign_ext_offset_ID_25);
128 assign PC_jump_ID_Im =(PC_MSB_ID,op_Sign_ext_offset_ID_25);// appending to get PC if Jump present
129 //for choosing what to write if set less than instruction is there in WB
130 mux2_1 #(32) PC_jump_mux(PC_jump_ID_Im, PC_jump_ID_reg, PC_jump_ID_select, PC_jump_ID);
131 wire Set_Less_than_inst_ID;
132 wire Set_Less_than_inst_EX;
133 wire Set_Less_than_inst_MEM;
134 wire Set_Less_than_inst_WB;
135
136 wire data_read_ID;
137 wire data_read_EX;
138 wire data_read_MEM;
139
140 // Control_Unit instantiation
141 Control_Unit Control_Unit_inst (Instruction_Code_ID[31:26],reg_write_ID,Write_regnum_Src_sel_line_ID,
142 ALU_Src_sel_line_ID,ALU_ctrl_ID,data_write_ID,write_Data_Src_mux_ID,PC_Src,flush,Set_Less_than_inst_ID,data_read_ID);
143
144 //#####

```

## 6. Determine the condition that can be used to detect data hazard?

Answer:

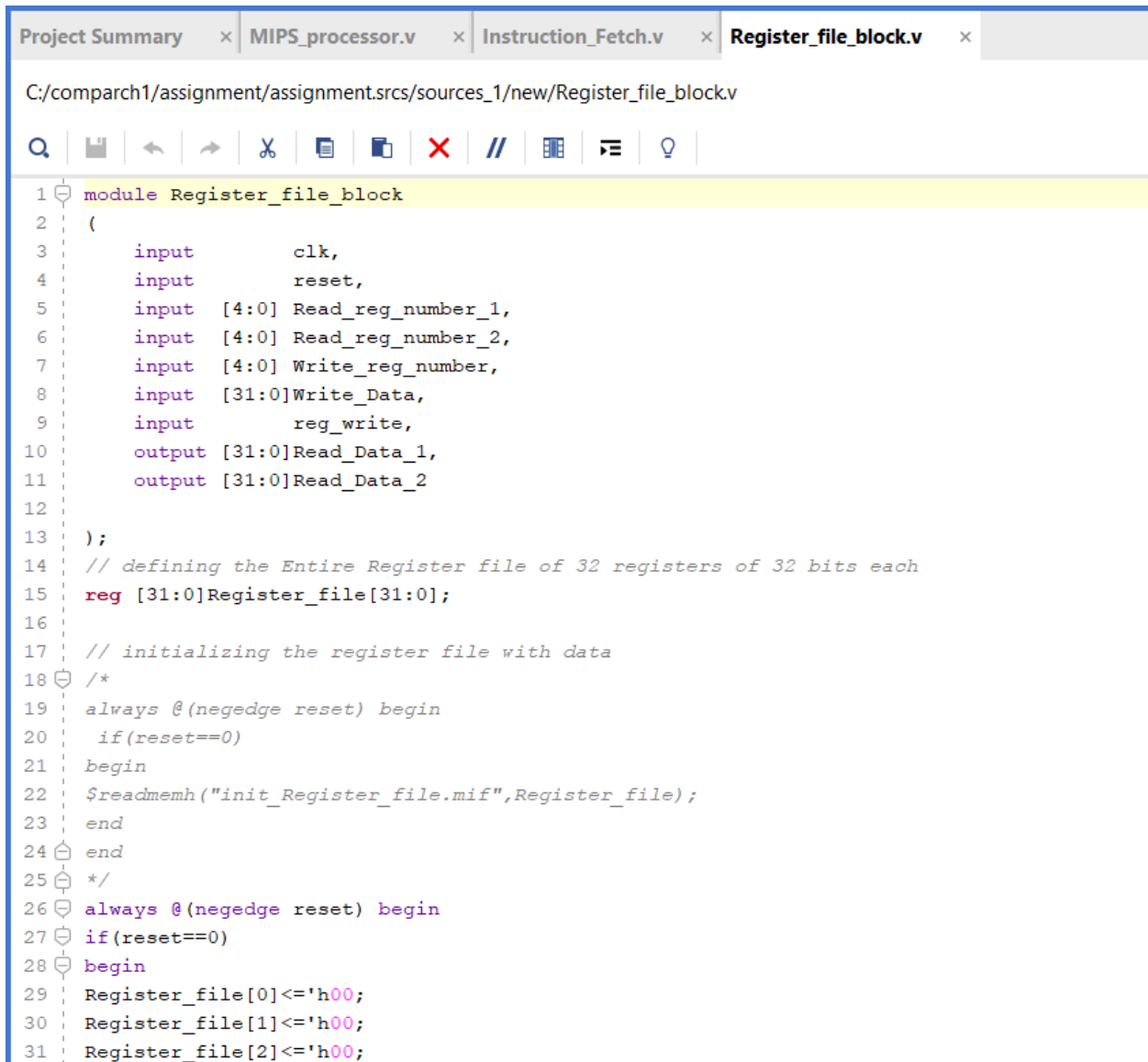
1. First is when there is a jump, we have to flush the IF-ID\_reg that will come in it in the next cycle after the Jump instruction is identified in the decode stage, this happens as currently the PC+4 is stored in the PC\_reg which is pointing towards the PC+4 location in the Instruction\_Memory and as soon as the Clock edge comes this wrong instruction gets stored in the IF-ID\_reg which is an Hazard,

hence we need to flush it so as to pass one NOP when there is a jump detected in the decoding

2. When there is a RAW dependency we need to forward if  $Write\_Reg\_Num\_MEM = Rs / Rt$  from the EX/Mem reg also known as Execution hazard similarly if we are not forwarding from the EX/Mem reg and if  $Write\_Reg\_Num\_WB = Rs / Rt$  we need to forward the data from MEM/WB reg as it was a memory hazard, here we need to take care that we forward from EX stage if both Ex or MEM hazard are present as the value of current instruction should get the latest updating value. equations shown in the code of forwarding block

7. **Implement the Register File and copy the image of Verilog code of Register file unit here.**

Answer:



```
1 module Register_file_block
2 (
3     input        clk,
4     input        reset,
5     input  [4:0] Read_reg_number_1,
6     input  [4:0] Read_reg_number_2,
7     input  [4:0] Write_reg_number,
8     input  [31:0] Write_Data,
9     input        reg_write,
10    output [31:0] Read_Data_1,
11    output [31:0] Read_Data_2
12
13 );
14 // defining the Entire Register file of 32 registers of 32 bits each
15 reg [31:0] Register_file[31:0];
16
17 // initializing the register file with data
18 /*
19 always @(negedge reset) begin
20     if(reset==0)
21     begin
22         $readmemh("init_Register_file.mif",Register_file);
23     end
24 end
25 */
26 always @(negedge reset) begin
27     if(reset==0)
28     begin
29         Register_file[0]<='h00;
30         Register_file[1]<='h00;
31         Register_file[2]<='h00;
```

```
32 Register_file[3]<='h00;  
33 Register_file[4]<='h00;  
34 Register_file[5]<='h00;  
35 Register_file[6]<='h00;  
36 Register_file[7]<='h00;  
37 Register_file[8]<='h00;  
38 Register_file[9]<='h00;  
39 Register_file[10]<='h00;  
40 Register_file[11]<='h00;  
41 Register_file[12]<='h14;  
42 Register_file[13]<='h00;  
43 Register_file[14]<='h00;  
44 Register_file[15]<='h00;  
45 Register_file[16]<='h00;  
46 Register_file[17]<='h00;  
47 Register_file[18]<='h00;  
48 Register_file[19]<='h00;  
49 Register_file[20]<='h00;  
50 Register_file[21]<='h00;  
51 Register_file[22]<='h00;  
52 Register_file[23]<='h00;  
53 Register_file[24]<='h00;  
54 Register_file[25]<='h00;  
55 Register_file[26]<='h00;  
56 Register_file[27]<='h00;  
57 Register_file[28]<='h00;  
58 Register_file[29]<='h00;  
59 Register_file[30]<='h00;  
60 Register_file[31]<='h00;  
61 end
```

```

62 end
63 //Read Logic (can be done asynchronously)
64 assign Read_Data_1 = Register_file[Read_reg_number_1];
65 assign Read_Data_2 = Register_file[Read_reg_number_2];
66
67 //Write Logic (should be synchronous)
68 always @(negedge clk) begin //writes on negative edge
69     if(reg_write)
70     begin
71         if(Write_reg_number==0)
72             Register_file[Write_reg_number]=0;
73         else
74             Register_file[Write_reg_number]=Write_Data;
75     end
76
77 end
78
79 endmodule
80

```

8. Implement the forwarding unit and copy the image of Verilog code of forwarding unit here.

Answer:

```

Project Summary x MIPS_processor.v x Instruction_Fetch.v x Register_file_block.v x Forwarding_Unit.v x
C:/comparch1/assignment/assignment.srcs/sources_1/new/Forwarding_Unit.v

1 module Forwarding_Unit (
2     input [4:0]Rs_EX,
3     input [4:0]Rt_EX,
4     input [4:0]Reg_write_num_MEM,
5     input [4:0]Reg_write_num_WB,
6     input reg_write_MEM,
7     input reg_write_WB,
8     output reg [1:0]Forward_Rs,
9     output reg [1:0]Forward_Rt
10 );
11 //Execution and Mem Hazard for Rs
12 always @(*)
13 begin
14     if((reg_write_MEM==1)&(Reg_write_num_MEM!=0)&(Reg_write_num_MEM==Rs_EX)) //Froward from EX-Mem_reg
15         Forward_Rs <= 2'b01;
16     else
17     begin
18         if((reg_write_WB==1)&(Reg_write_num_WB!=0)&(Reg_write_num_WB==Rs_EX)) //Froward from Mem-WB_reg
19             Forward_Rs <= 2'b10;
20         else
21             Forward_Rs <= 2'b00; // No forwading needed
22     end
23 end
24 end
25 //Execution and Mem Hazard for Rt
26 always @(*)
27 begin
28     if((reg_write_MEM==1)&(Reg_write_num_MEM!=0)&(Reg_write_num_MEM==Rt_EX)) //Froward from EX-Mem_reg
29         Forward_Rt <= 2'b01;
30     else
31     begin
32         if((reg_write_WB==1)&(Reg_write_num_WB!=0)&(Reg_write_num_WB==Rt_EX)) //Froward from Mem-WB_reg
33             Forward_Rt <= 2'b10;
34         else
35             Forward_Rt <= 2'b00; // No forwading needed
36     end
37 end
38 endmodule
39

```

9. Implement a complete processor in Verilog (using all the Datapath blocks). Copy the image of Verilog code of the processor here. (Use comments to describe your Verilog implementation)

Answer:

Project Summary
MIPS\_processor.v
Instruction\_Fetch.v
Register\_file\_block.v
Forwarding\_Unit.v

C:/comparch1/assignment/assignment.srscs/sources\_1/new/MIPS\_processor.v

```

1 module MIPS_processor
2 (
3     input clk,
4     input reset
5 );
6
7 // IF Unit
8 //wires_op_IF
9 wire PC_Src; ///will come from control unit->Instruction Fetch unit present in decode stage directly
10 wire [31:0] PC_jump_ID; ///will IF -> IF/ID reg -> ID -> ID/Ex reg -> Ex -> Instruction Fetch unit
11
12 wire [3:0] PC_MSB_ID;
13 wire [3:0] PC_MSB_IF;
14 wire [3:0] PC_MSB_EX;
15
16 wire [31:0] Instruction_Code_IF ;
17 wire [31:0] PC_IF ;
18
19 wire stall_flush; // comes as an output of the stalling unit
20
21
22
23 Instruction_Fetch Instruction_Fetch_inst (clk,reset,stall_flush,PC_jump_ID,PC_Src,Instruction_Code_IF,PC_IF);
24
25 assign PC_MSB_IF = PC_IF[31:28];
26
27
28 //*****
29
30 //IF/ID reg (Stores Instruction_code, PC(MSB-4 bits))
31
32
33 wire [31:0] Instruction_Code_ID ;
34 wire [4:0] Rd_ID;
35 wire [4:0] Rs_ID;
36 wire [4:0] Rt_ID;
37
38 wire [15:0] ip_sign_ext_offset_ID_15;
39
40 // Flushing unit ip wire
41 wire flush; // we'll have to flush the IF/ID_reg to pass a Nop in system instead of passing the instruction currently that will come in IF-ID_reg as PC has Stored PC+4 in the PC reg
42
43 IF_ID_reg_block IF_ID_reg_block_inst (clk,reset,stall_flush,flush,Instruction_Code_IF,PC_MSB_IF,PC_MSB_ID,Rd_ID,Rs_ID,Rt_ID,ip_sign_ext_offset_ID_15,Instruction_Code_ID);
44
45 //lui
46 wire [31:0] final_write_value_for_lui_ID;
47 reg [31:0] final_write_value_for_lui_EX, final_write_value_for_lui_MEM, final_write_value_for_lui_WB ;
48 reg [5:0] lui_opcode_EX, lui_opcode_MEM, lui_opcode_WB;
49
50 assign final_write_value_for_lui_ID=(Instruction_Code_ID [15:0], 16'b0);
51 always @(posedge clk) begin
52     final_write_value_for_lui_EX<=final_write_value_for_lui_ID;
53     final_write_value_for_lui_MEM<=final_write_value_for_lui_EX;
54     final_write_value_for_lui_WB<=final_write_value_for_lui_MEM;
55     lui_opcode_EX<=Instruction_Code_ID [31:26];
56     lui_opcode_MEM<=lui_opcode_EX;
57     lui_opcode_WB<=lui_opcode_MEM;
58 end
59 //*****
60
61 // ID Unit

```

```

62
63 //Select-lines All#####
64
65 //sel_lines_IF.....
66 //PC_Src not going further but comes from control unit
67
68 //sel_lines_ID.....
69 ⊕ //reg_write.
70 wire reg_write_ID; //Comes directly form CU
71 wire reg_write_EX;
72 wire reg_write_MEM;
73 wire reg_write_WB; //USED
74
75 ⊕ //sel_lines_EX.....
76 ⊕ //Write_regnum_Src_sel_line
77 wire Write_regnum_Src_sel_line_ID; //Comes directly form CU
78 wire Write_regnum_Src_sel_line_EX; //USED
79
80 //ALU_Src_sel_line_EX
81 wire ALU_Src_sel_line_ID; //Comes directly form CU
82 wire ALU_Src_sel_line_EX; //USED
83
84 //ALU_ctrl_EX
85 wire [3:0]ALU_ctrl_ID; //Comes directly form CU
86 wire [3:0]ALU_ctrl_EX; //USED
87
88
89 ⊕ //sel_lines_MEM.....
90 ⊕ //data_write_MEM
91 wire data_write_ID; //Comes directly form CU

92 wire data_write_EX;
93 wire data_write_MEM; //USED
94
95 //sel_lines_WB.....
96 wire write_Data_Src_mux_ID; //Comes directly form CU
97 wire write_Data_Src_mux_EX;
98 wire write_Data_Src_mux_MEM;
99 wire write_Data_Src_mux_WB; //USED
100
101 ⊕ #####
102 ⊕ //Wires ip/op ID
103 wire [25:0]ip_Sign_ext_offset_ID_25;
104 wire [27:0]op_Sign_ext_offset_ID_25;
105 //Wires op ID
106 wire [31:0]Write_Data_WB;
107
108 wire [31:0]op_Sign_ext_offset_ID_15;
109 wire [31:0]Read_Data_1_ID;
110 wire [31:0]Read_Data_2_ID;
111
112 assign ip_Sign_ext_offset_ID_25 ={Rs_ID,Rt_ID,ip_Sign_ext_offset_ID_15[15:0]};
113 // sign extention blk 15->31
114 Sign_ext_blk_15 Sign_ext_blk_15 (ip_Sign_ext_offset_ID_15,op_Sign_ext_offset_ID_15);
115 // Register file instantiation
116 wire[4:0] Reg_write_num_WB;
117 wire [31:0]Final_Write_Data; //Coming form Set_less_than_Writing_mitigation_mux.
118 ⊕ Register_file_block Register_file_inst(clk,reset,Rs_ID,Rt_ID,Reg_write_num_WB,
119 ⊕ Final_Write_Data,reg_write_WB,Read_Data_1_ID,Read_Data_2_ID);
120 //jr
121 wire [31:0] PC_jump_ID_Im, PC_jump_ID_reg;

```



```

122 wire PC_jump_ID_select;
123 assign PC_jump_ID_select=(Instruction_Code_ID[31:26]==6'b000010)?1:0;
124 assign PC_jump_ID_reg=Read_Data_1_ID;
125
126 //...instantiating Sign_ext_blk_25
127 Sign_ext_blk_25 Sign_ext_blk_25_inst (ip_Sign_ext_offset_ID_25,op_Sign_ext_offset_ID_25);
128 assign PC_jump_ID_Im =(PC_MSB_ID,op_Sign_ext_offset_ID_25); // appending to get PC if Jump present
129 //for choosing what to write if set less than instruction is there in WB
130 mux2_1 #(32) PC_jump_mux(PC_jump_ID_Im, PC_jump_ID_reg, PC_jump_ID_select, PC_jump_ID);
131 wire Set_Less_than_inst_ID;
132 wire Set_Less_than_inst_EX;
133 wire Set_Less_than_inst_MEM;
134 wire Set_Less_than_inst_WB;
135
136 wire data_read_ID;
137 wire data_read_EX;
138 wire data_read_MEM;
139
140 // Control_Unit instantiation
141 Control_Unit Control_Unit_inst(Instruction_Code_ID[31:26],reg_write_ID,Write_regnum_Src_sel_line_ID,
142 ALU_Src_sel_line_ID,ALU_ctrl_ID,data_write_ID,write_Data_Src_mux_ID,PC_Src,flush,Set_Less_than_inst_ID,data_read_ID);
143
144 //*****
145
146 //ID/EX reg (Stores Read_Data_1,Read_Data_2,op_Sign_ext_offset_ID,Rs_ID,Rt_ID,Rd_ID)
147 wire [31:0]Read_Data_1_EX;
148 wire [31:0]Read_Data_2_EX;
149 wire [31:0]op_Sign_ext_offset_EX_15;
150 wire [4:0]Rs_EX;
151 wire [4:0]Rt_EX;

152 wire [4:0]Rd_EX;
153
154
155 wire [5:0]Opcode_ex; // used for checking the Stalling condition as if it were Load inst
156 //present in the ID-EX reg and there is a data dependency then we need to Stall the Pipeline for
157 // One Cycle and also flush the IF-ID_reg of the pipeline to pass a nop
158
159 //Stalling_Unit .....instantiation
160 Stalling_Unit Stalling_Unit_inst(Opcode_ex,Rt_EX,Rs_ID,Rt_ID,stall_flush);
161
162 ID_EX_reg_block ID_EX_reg_block (clk,reset,stall_flush,Instruction_Code_ID[31:26],data_read_ID,Set_Less_than_inst_ID,reg_write_ID,Write_regnum_Src_sel_line_ID,ALU_Src_sel_line_ID,
163 ALU_ctrl_ID,data_write_ID,write_Data_Src_mux_ID,
164 Read_Data_1_ID,Read_Data_2_ID,op_Sign_ext_offset_ID_15,Rs_ID,Rt_ID,Rd_ID,PC_MSB_ID,Opcode_ex,data_read_EX,Set_Less_than_inst_EX,
165 reg_write_EX,Write_regnum_Src_sel_line_EX,ALU_Src_sel_line_EX,ALU_ctrl_EX,data_write_EX,write_Data_Src_mux_EX,
166 Read_Data_1_EX,Read_Data_2_EX,op_Sign_ext_offset_EX_15,Rs_EX,Rt_EX,Rd_EX,PC_MSB_EX);
167
168 //*****
169
170 //EX unit
171
172 //wires Ip_EX
173 wire [31:0]ALU_inp_1 ; // comes from output of Forward_Rs_data_Mux
174 wire [31:0]ALU_inp_2 ; // comes from output of ALU_Src_mux
175
176 //wires Op_EX
177 wire [31:0]ALU_result;
178 wire STL_EX;
179 wire [4:0]Reg_write_num_EX;
180
181 //assign ALU_inp_1=Read_Data_1_EX; not needed now

```

```

182
183 //wires_op from forwarding_unit
184 wire [1:0]Forward_Rs;
185 wire [1:0]Forward_Rt;
186 wire [31:0]forward_from_mem;
187 wire [31:0]ALU_result_MEM;
188
189 wire [31:0]ALU_Src_mux_Input_1; // comes from output of Forward_Rt_data_Mux
190 //....instantiating Forward_Rs_data_Mux for ALU_inp_1
191 mux3_1 #(32) Forward_Rs_data_Mux (Read_Data_1_EX,forward_from_mem,Final_Write_Data,Forward_Rs,ALU_inp_1);
192
193 //....instantiating Forward_Rt_data_Mux for ALU_inp_2
194 mux3_1 #(32) Forward_Rt_data_Mux (Read_Data_2_EX,forward_from_mem,Final_Write_Data,Forward_Rt,ALU_Src_mux_Input_1);
195
196 //...instantiating ALU_Src_mux
197 mux2_1 #(32) ALU_Src_mux (ALU_Src_mux_Input_1,op_Sign_ext_offset_EX_15,ALU_Src_sel_line_EX,ALU_inp_2);
198 //ALU_Src_sel_line_EX=0 for R type, ALU_Src_sel_line_EX=1 for load-store same for ALU_ctrl
199
200 //...Instantiating Write_regnum_Src_mux
201 mux2_1 #(5) Write_regnum_Src_mux (Rt_EX,Rd_EX,Write_regnum_Src_sel_line_EX,Reg_write_num_EX);
202
203 //...Instantiating ALU_Assignment
204 ALU_Assignment ALU_Assignment(ALU_inp_1,ALU_inp_2,ALU_ctrl_EX,STL_EX,ALU_result);
205
206 //*****
207
208 //EX-MEM_reg
209
210 wire [31:0]Datamem_Read_Data_MEM ;
211 wire [7:0]Address_Datamem;

212 wire [31:0]Write_Data_Datamem;
213 wire [4:0]Reg_write_num_MEM;
214 wire STL_MEM;
215
216
217
218 EX_MEM_reg_block EX_MEM_reg_block_inst (clk,reset,data_read_EX,Set_Less_than_inst_EX,ALU_result,Read_Data_2_EX,Reg_write_num_EX,STL_EX,reg_write_EX,
219 data_write_EX,write_Data_Src_mux_EX,data_read_MEM,Set_Less_than_inst_MEM,ALU_result_MEM,Write_Data_Datamem,Reg_write_num_MEM,STL_MEM,reg_write_MEM,data_write_MEM,
220 write_Data_Src_mux_MEM);
221 //*****
222
223 //Forwarding Unit
224
225
226
227 Forwarding_Unit Forwarding_Unit_inst (Rs_EX,Rt_EX,Reg_write_num_MEM,Reg_write_num_WB,
228 reg_write_MEM,reg_write_WB,Forward_Rs,Forward_Rt);
229
230
231 //MEM unit
232 //Wires_Ip_MEM
233 assign Address_Datamem= ALU_result_MEM [7:0] ;
234
235
236
237 mux2_1 #(32) forward_from_mem_inst (ALU_result_MEM,Datamem_Read_Data_MEM,data_read_MEM,forward_from_mem);
238
239 Data_memory Data_memory_inst (clk,reset,data_read_MEM,Address_Datamem,Address_Datamem,Write_Data_Datamem,data_write_MEM,Datamem_Read_Data_MEM);
240
241

---
242 //Wires_Op_MEM
243 wire[31:0] Write_regfile_Source_1;
244 wire[31:0] Write_regfile_Source_2;
245
246
247 //*****
248
249 //MEM_WB_reg
250 wire STL_WB;
251
252 MEM_WB_reg_block MEM_WB_reg_block_inst(clk,reset,Set_Less_than_inst_MEM,reg_write_MEM,write_Data_Src_mux_MEM,ALU_result_MEM,Datamem_Read_Data_MEM,Reg_write_num_MEM,STL_MEM,Set_Less_than_inst_WB,
253 Write_regfile_Source_1,Write_regfile_Source_2,Reg_write_num_WB,STL_WB,reg_write_WB,write_Data_Src_mux_WB);
254
255 //*****
256
257 //WB unit
258 //Wires_Ip_WB
259 //Wires_Op_WB
260
261 mux2_1 #(32) Write_Data_Src_mux (Write_regfile_Source_1,Write_regfile_Source_2,write_Data_Src_mux_WB,Write_Data_WB);
262
263
264 //Set Less than mitigation code
265 // As if Set Less Than inst_WB=0 it should write Write_Data_WB regardless of the value of STL_WB
266 wire [31:0] Final_Write_Data_dummy;
267 mux4_1 #(32) Set_less_than_Writing_mitigation_mux (Write_Data_WB,Write_Data_WB,32'd0,32'd1,(Set_Less_than_inst_WB,STL_WB),Final_Write_Data_dummy);
268
269 assign Final_Write_Data=(lui_opcode_WB==6'b001111)?final_write_value_for_lui_WB:Final_Write_Data_dummy;
270 endmodule
271
272

```

```

273 module mux2_1 #(parameter n=4)
274     (input  [n-1:0]a, [n-1:0]b,
275     input  s0,
276     output [n-1:0]out);
277
278     assign out=s0? b: a;
279 endmodule
280
281
282 module mux3_1 #(parameter n=4)
283
284     (input  [n-1:0]a, b, c,
285     input  [1:0]s,
286     output [n-1:0]out);
287
288     wire [n-1:0]a_out,b_out,c_out;
289     Gate_and_2s #(n) and3_1  (a,~s[0],~s[1],a_out);
290     Gate_and_2s #(n) and3_2  (b,s[0],~s[1],b_out);
291     Gate_and_2s #(n) and3_3  (c,~s[0],s[1],c_out);
292
293     assign out[n-1:0] =  a_out | b_out | c_out;
294
295 endmodule
296
297 module mux4_1 #(parameter n=4)
298
299     (input  [n-1:0]a, b, c, d,
300     input  [1:0]s,
301     output [n-1:0]out);
302
303     assign out = s[1]? (s[0]?d:c) : (s[0]?b:a);

```

```

304 |
305 | endmodule
306 |
307 | module Gate_and_2s #(parameter n=4)
308 | (
309 |     input [n-1:0]a,
310 |     input s0,s1,
311 |     output [n-1:0]a_out
312 | );
313 |
314 | wire sel;
315 |
316 |     assign sel=s1&s0;
317 |
318 | genvar i;
319 | generate
320 |     for(i=0;i<n;i=i+1)
321 |     begin
322 |         assign  a_out[i]=a[i]&sel;
323 |     end
324 | endgenerate
325 |
326 | endmodule
327 |

```

10. Test the processor design by generating the appropriate clock and reset. Copy the image of your testbench code here.

Answer:

```

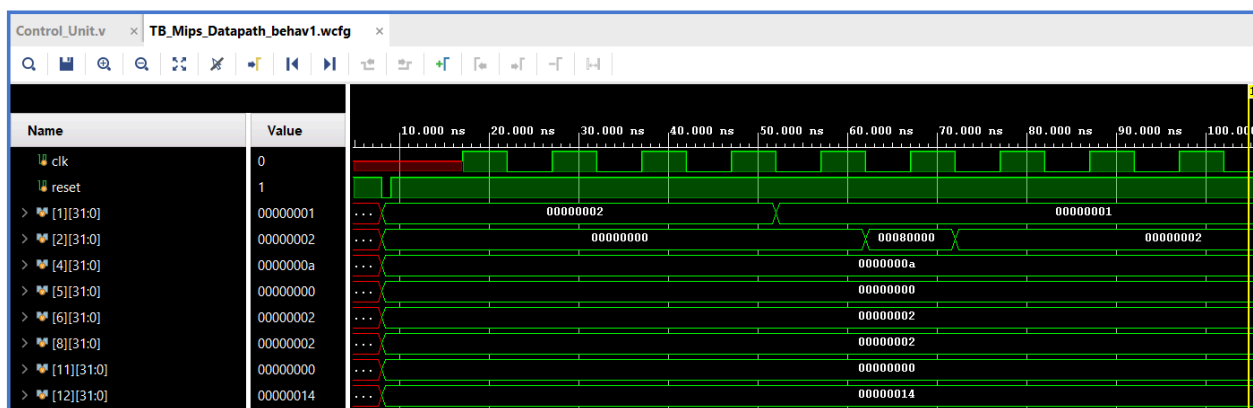
Project Summary x MIPS_processor.v * x Instruction_Fetch.v x Register_file_block.v x Forwarding_Unit.v x TB_Mips_Datapath.v x
C:/comparch1/assignment/assignment.srcs/sim_1/new/TB_Mips_Datapath.v

1 `timescale 1ns/1ps
2 module TB_Mips_Datapath();
3   reg clk;
4   reg reset;
5
6   MIPS_processor MIPS_processor_inst(clk,reset);
7
8
9   initial begin
10      reset=1;
11      #8 reset=0;
12      #1 reset=1;
13
14      #8;
15
16      //defining olook
17
18      forever
19      begin
20          clk <= 1'b1;
21          #5;
22
23          clk <= 1'b0;
24          #5;
25      end
26
27   end
28
29
30 endmodule

```

11. Verify if the register file is getting updated according to the set of instructions (mentioned earlier).

Copy verified **Register file** waveform here (show only the Registers that get updated, CLK, and RESET):



12. What is the total number of cycles needed to issue the program given above on the pipelined MIPS Processor? What is the CPI of the program?

Answer:

Neglecting the initial filling time it takes 1 cycle per instruction, without any NOP (i.e. control dependency not present). But as there is jr we need one NOP. So, we have penalty of 1.

% jr in the code =  $\frac{1}{6} \times 100 = \frac{100}{6} = 50\frac{1}{3}\%$

New CPI =  $1 + \left( \frac{50\frac{1}{3}\%}{100} \right) \times \text{penalty}$

$$= 1 + \frac{1}{6} = \frac{7}{6} = 1.\underline{\underline{1667}}$$

Total number of cycles required=10

13. Make a diagram showing the clock-by-clock execution of each instruction, indicating stalling, forwarding etc wherever necessary.

Answer:

INSTRUCTION	C <sub>1</sub>	C <sub>2</sub>	C <sub>3</sub>	C <sub>4</sub>	C <sub>5</sub>	C <sub>6</sub>	C <sub>7</sub>	C <sub>8</sub>	C <sub>9</sub>	C <sub>10</sub>
lw R <sub>1</sub> , R <sub>11</sub> , #12	IF	ID	EX	Mem	WB					
lui R <sub>2</sub> , R <sub>9</sub> , #8		IF	ID	EX	Mem	WB				
mul R <sub>2</sub> , R <sub>1</sub> , R <sub>8</sub>			IF	ID	EX	Mem	WB			
jrc R <sub>12</sub>				IF	ID	EX	Mem	WB		
mul R <sub>6</sub> , R <sub>6</sub> , R <sub>6</sub>					IF	NOP	NOP	NOP	NOP	
w R <sub>4</sub> , 4[R <sub>5</sub> ]						IF	ID	EX	Mem	WB

*Handwritten notes:*

- Red arrows from R<sub>1</sub> in the first instruction to R<sub>1</sub> in the third instruction indicate a data dependency.
- Green arrows from the WB stage of the first instruction to the EX stage of the second and third instructions indicate a write-back dependency.
- Green text "Control dependency => flush" with an arrow from the IF stage of the fifth instruction to the IF stage of the sixth instruction.
- Green text "IF = ID" and "NOP passed" with arrows pointing to the IF and NOP stages of the sixth instruction.

14. Your design synthesisable? Which target FPGA was used for synthesis?

Answer: Yes Module-By-Module; Zynq-Ultrascale+-ZCU106

15. Provide the synthesis report in tabular form (resources consumed)?

Answer:

	LUT	Registers	Carry 8	IOB Buffer	Clock Buffer	F7	F8	
Instruction Fetch Unit	58	32	4	100	1	0	0	
IF_ID_reg	20	36	0	107	1	0	0	
Register_File	72	0	0	113	1	0	0	LUT's as RAM
Sign_ext_15	0	0	0	48	0	0	0	
Sign_ext_25	0	0	0	54	0	0	0	
Control_Unit	8	0	0	19	0	0	0	
Stalling_Unit	3	0	0	17	0	0	0	
ID_EX_reg	0	264	0	267	1	0	0	
Forward_Rs	4	0	0	18	0	0	0	
FDownward_Rt	4	0	0	18	0	0	0	
ALU_Src_Mux	2	0	0	13	0	0	0	
Write_Reg_Num_Mux	2	0	0	13	0	0	0	
ALU_Assignment	81	0	6	101	0	0	0	
EX_MEM_reg	1	75	0	152	1	0	0	
Forwarding_Unit	14	0	0	26	0	0	0	
Data_Memory	656	352	0	79	1	152	48	
MEM_WB_reg	1	73	0	148	1	0	0	
Write_Data_Src_Mux	2	0	0	13	0	0	0	
Set_less_than_Mitigation_Mux	2	0	0	13	0	0	0	
Total	930	832	10	1319	7	152	48	

## Unrelated Questions

What were the problems you faced during the implementation of the processor?

Answer: Debugging was difficult

Did you implement the processor on your own? If you took help from someone whose help did you take? Which part of the design did you take help for?

Answer: Yes, I implemented the processor on my own

## Honor Code Declaration by student:

- My answers to the above questions are my own work.
- I have not shared the codes/answers written by me with any other students. (I might have helped clear doubts of other students).
- I have not copied other's code/answers to improve my results. (I might have got some doubts cleared from other students).

**Name:** Yashi Malik

**Date:** 14/4/24

**ID No.:** 2021A3PS3056G