ECEN 651 Lab Exercise 6 Report

Pipelined MIPS Processor

Name: Pranav Anantharam

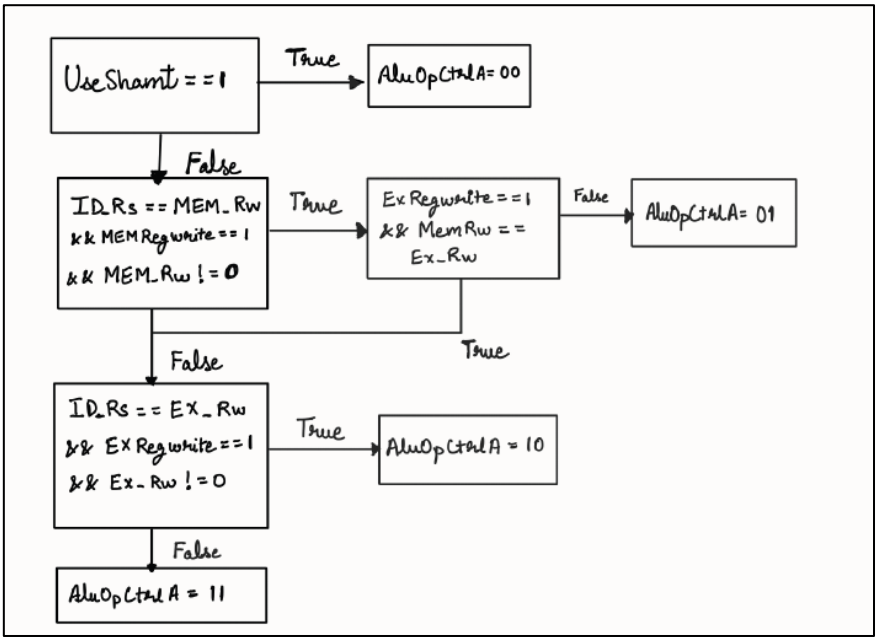
UIN: 734003886

Date: 12/3/2023

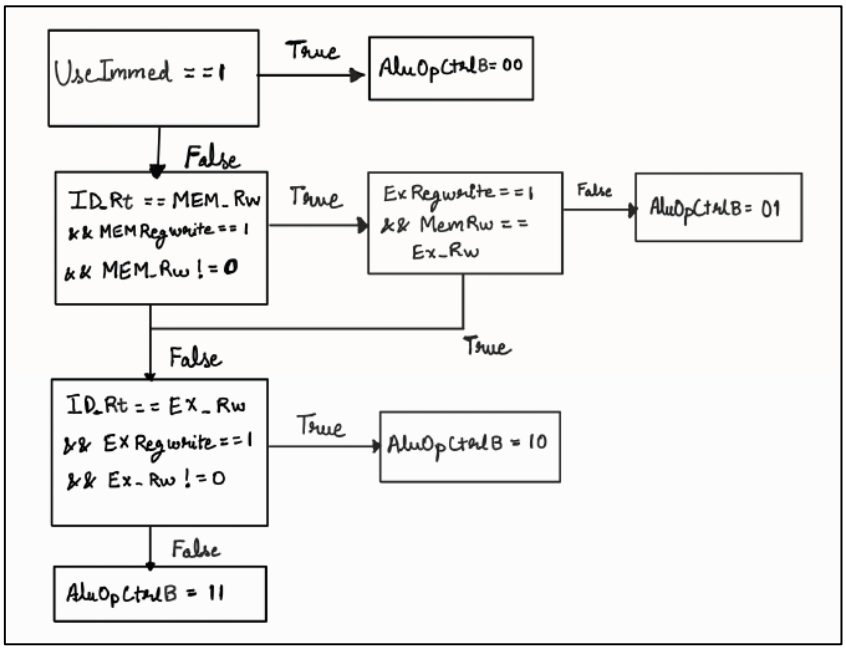
1. Design the forwarding unit.

(a) Draw a block diagram which reflects the forwarding unit logic

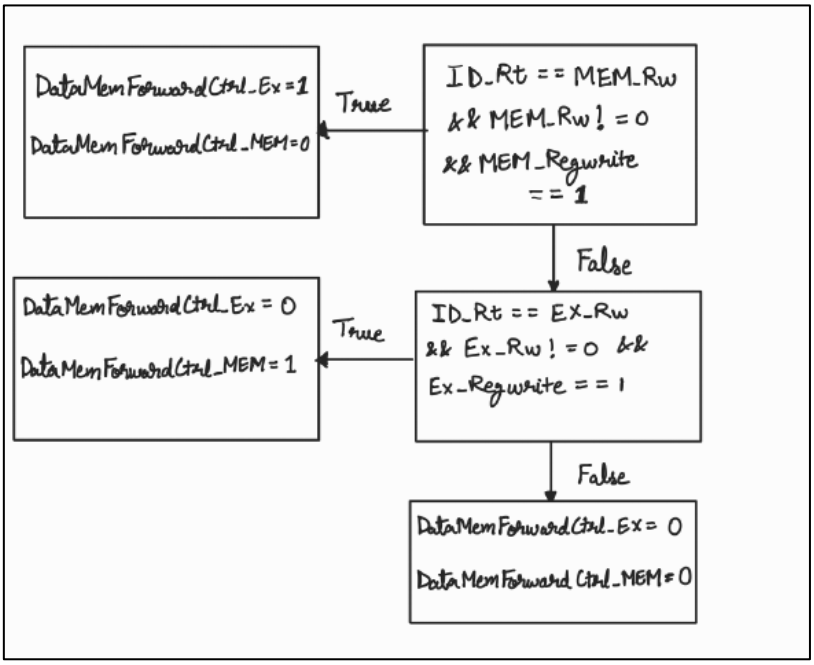
For R-type instructions: AluOCtrlA



For Immediate instructions: AluOCtrlB



Select lines for Muxes for Memory Write Data in EX and MEM stages:



(b) Write a Verilog module to implement the Forwarding Unit.

Design Source code filename: ForwardingUnit.v

```
`timescale 1ns / 1ps

module ForwardingUnit( UseShamt, UseImmed, ID_Rs, ID_Rt, EX_Rw, MEM_Rw, EX_RegWrite,
MEM_RegWrite, AluOpCtrlA, AluOpCtrlB, DataMemForwardCtrl_EX, DataMemForwardCtrl_MEM);

	// Inputs
	input UseShamt, UseImmed;
	input [4:0] ID_Rs, ID_Rt, EX_Rw, MEM_Rw;
	input EX_RegWrite, MEM_RegWrite;

	// Outputs
	output reg [1:0] AluOpCtrlA, AluOpCtrlB;
	output reg DataMemForwardCtrl_EX, DataMemForwardCtrl_MEM;

	always @(*) begin

		if( UseShamt == 1 )			// Shift values used in current instruction
			AluOpCtrlA <= 2'b00;	// Shift value data -> BusA

		else if( UseShamt == 0 )begin // Shift values not used in current instruction

			// Check priority of pipeline stage for previous instructions
			if( (ID_Rs == MEM_Rw) && ~(MEM_Rw==EX_Rw && EX_RegWrite) &&
(MEM_RegWrite == 1) && (MEM_Rw != 0) )
				AluOpCtrlA <= 2'b01;	// Memory stage data -> BusA
			else if ( (ID_Rs == EX_Rw) && (EX_RegWrite == 1) && (EX_Rw != 0) )
				AluOpCtrlA <= 2'b10;	// Execute stage data -> BusA
			else
				AluOpCtrlA <= 2'b11;	// Register read data -> BusA
			end

		else
			AluOpCtrlA <= 2'b11;		// Register read data -> BusA
```

```
            if( UseImmed == 1 )       // Immediate values used in current instruction
                  AluOpCtrlB <= 2'b00;            // Immediate value data -> BusB

            else if( UseImmed == 0 ) begin // Immediate values not used in current
instruction

                  // Check priority of pipeline stage for previous instructions
                  if( (ID_Rt == MEM_Rw) && ~(MEM_Rw==EX_Rw && EX_RegWrite) &&
(MEM_RegWrite == 1) && (MEM_Rw != 0) )
                        AluOpCtrlB <= 2'b01;      // Memory stage data -> BusB
                  else if( (ID_Rt == EX_Rw) && (EX_RegWrite == 1) && (EX_Rw != 0) )
                        AluOpCtrlB <= 2'b10;      // Execute stage data -> BusB
                  else
                        AluOpCtrlB <= 2'b11;      // Register read data -> BusB
                  end

            else
                  AluOpCtrlB <= 2'b11;            // Register read data -> BusB


            // Select Lines for Muxes for Memory Write data in EX and MEM stages
            if( (MEM_RegWrite == 1) && (ID_Rt == MEM_Rw) && (MEM_Rw != 0) ) begin
                  DataMemForwardCtrl_EX = 1'b1;
                  DataMemForwardCtrl_MEM = 1'b0;

            end else if( (EX_RegWrite == 1) && (ID_Rt == EX_Rw) && (EX_Rw != 0) ) begin
                  DataMemForwardCtrl_EX = 1'b0;
                  DataMemForwardCtrl_MEM = 1'b1;

            end else begin
                  DataMemForwardCtrl_EX = 1'b0;
                  DataMemForwardCtrl_MEM = 1'b0;
            end

      end

endmodule
```

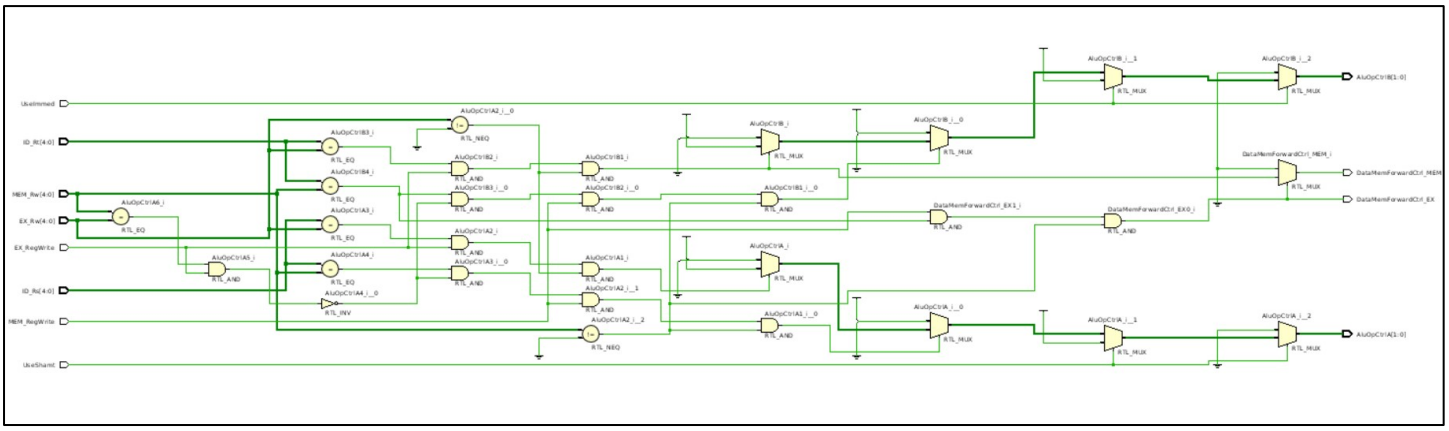
(c) Synthesize the hardware and ensure the code generates no warnings or errors and that it creates no latches. Provide the summary results of the synthesis process.

Synthesis Report: (ForwardingUnit module)

```
---------------------------------------------------------------------------------
Start Writing Synthesis Report
---------------------------------------------------------------------------------
Report BlackBoxes:
+-+--------------+----------+
| |BlackBox name |Instances |
+-+--------------+----------+
+-+--------------+----------+
Report Cell Usage:
+------+-----+------+
|      |Cell |Count |
+------+-----+------+
|1     |LUT2 |     2|
|2     |LUT3 |     2|
|3     |LUT4 |     2|
|4     |LUT6 |    11|
|5     |IBUF |    24|
|6     |OBUF |     6|
```

```
+------+-----+------+
Report Instance Areas:
+------+---------+-------+------+
|      |Instance |Module |Cells |
+------+---------+-------+------+
|1     |top      |       |    47|
+------+---------+-------+------+
-------------------------------------------------------------------------------
Finished Writing Synthesis Report : Time (s): cpu = 00:00:13 ; elapsed = 00:00:20 . Memory
(MB): peak = 1770.918 ; gain = 306.605 ; free physical = 2643 ; free virtual = 39972
-------------------------------------------------------------------------------
```
Synthesis finished with 0 errors, 0 critical warnings and 0 warnings.
```
Synthesis Optimization Runtime : Time (s): cpu = 00:00:13 ; elapsed = 00:00:20 . Memory (MB):
peak = 1770.918 ; gain = 306.605 ; free physical = 2655 ; free virtual = 39984
Synthesis Optimization Complete : Time (s): cpu = 00:00:13 ; elapsed = 00:00:20 . Memory (MB):
peak = 1770.918 ; gain = 306.605 ; free physical = 2716 ; free virtual = 40044
INFO: [Project 1-571] Translating synthesized netlist
INFO: [Project 1-570] Preparing netlist for logic optimization
INFO: [Opt 31-138] Pushed 0 inverter(s) to 0 load pin(s).
Netlist sorting complete. Time (s): cpu = 00:00:00 ; elapsed = 00:00:00 . Memory (MB): peak =
1909.930 ; gain = 0.000 ; free physical = 2641 ; free virtual = 39973
INFO: [Project 1-111] Unisim Transformation Summary:
No Unisim elements were transformed.

INFO: [Common 17-83] Releasing license: Synthesis
```
9 Infos, 0 Warnings, 0 Critical Warnings and 0 Errors encountered.
```
synth_design completed successfully
synth_design: Time (s): cpu = 00:00:17 ; elapsed = 00:00:24 . Memory (MB): peak = 1909.930 ;
gain = 474.547 ; free physical = 2714 ; free virtual = 40047
Netlist sorting complete. Time (s): cpu = 00:00:00 ; elapsed = 00:00:00 . Memory (MB): peak =
1909.930 ; gain = 0.000 ; free physical = 2710 ; free virtual = 40043
WARNING: [Constraints 18-5210] No constraints selected for write.
Resolution: This message can indicate that there are no constraints for the design, or it can
indicate that the used_in flags are set such that the constraints are ignored. This later case
is used when running synth_design to not write synthesis constraints to the resulting
checkpoint. Instead, project constraints are read when the synthesized design is opened.
INFO: [Common 17-1381] The checkpoint
'/home/grads/p/pranav_anantharam/lab6/project_1/project_1.runs/synth_1/ForwardingUnit.dcp' has
been generated.
INFO: [runtcl-4] Executing : report_utilization -file ForwardingUnit_utilization_synth.rpt -pb
ForwardingUnit_utilization_synth.pb
INFO: [Common 17-206] Exiting Vivado at Sun Dec  3 18:12:59 2023...
```

Compiled Hardware Schematic: (FowardingUnit module)

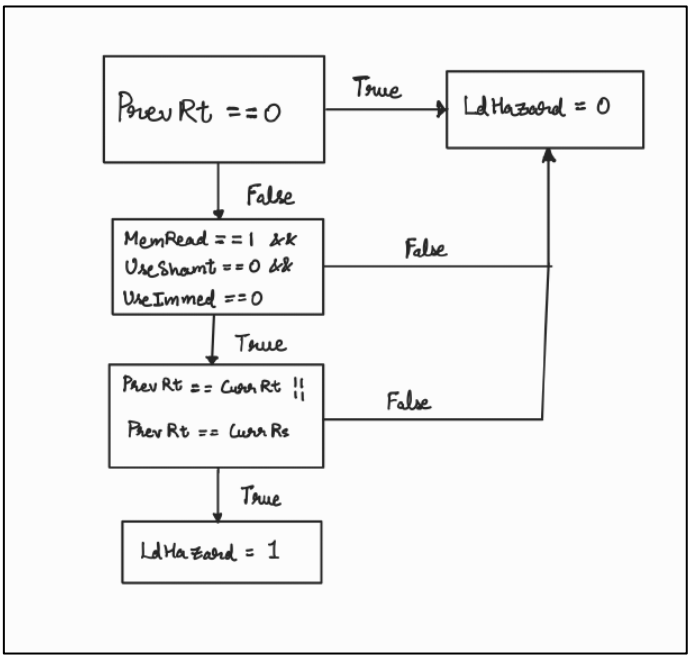


No latches or flip flops present in the forwarding unit.

2. Design the Hazard Detection Unit

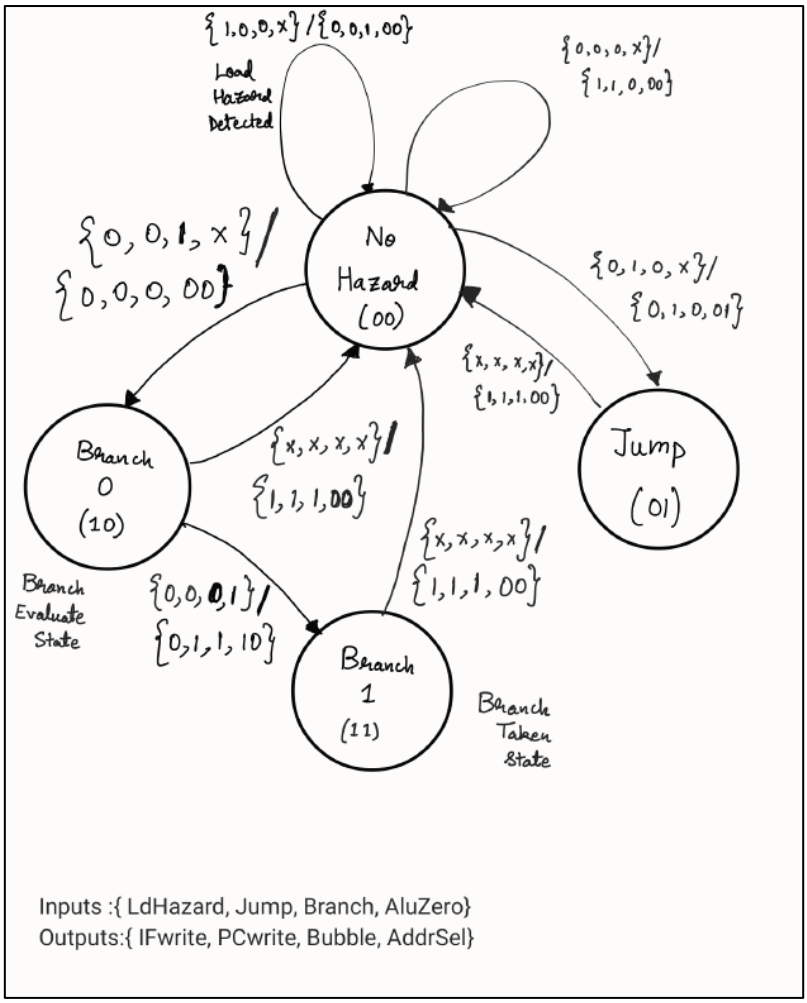
(a) Create a block diagram for the logic required to detect a data hazard that requires stalling.

Load Hazard:



(b) Create the state diagram that describes the workings of the Hazard unit. Assume that the inputs to this FSM are Jump, LdHazard, Branch, and ALUZero and the outputs are PC write, IF write, bubble, and addrSel.

Hazard Unit FSM:



(c) Describe the Hazard Detection Unit in Verilog.

Design Source code filename: HazardUnit.v

```
`timescale 1ns / 1ps


module HazardUnit( IF_write, PC_write, bubble, addrSel, Jump, Branch, ALUZero, memReadEX,
currRs, currRt, prevRt, UseShamt, UseImmed, Clk, Rst);

	// Outputs
	output reg IF_write, PC_write, bubble;
	output reg [1:0] addrSel;

	// Inputs
	input Jump, Branch, ALUZero, memReadEX, Clk, Rst;
	input UseShamt, UseImmed;
	input [4:0] currRs;
	input [4:0] currRt;
	input [4:0] prevRt;

	// State definition for FSM
	parameter NoHazard_state = 3'b000;    // No hazard - Normal state
	parameter Jump_state = 3'b001;        // Jump
	parameter Branch_0_state = 3'b010;    // Branch Encountered
	parameter Branch_1_state = 3'b011;    // Branch Taken

	// Internal signal - LdHazard - HIGH when hazard exists
	reg LdHazard;

	// Internal states - FSM_state, FSM_nxt_state
	reg [2:0] FSM_state;
	reg [2:0] FSM_nxt_state;


	// FSM state register
	always @(negedge Clk) begin
		if( Rst == 0 )
			FSM_state <= 0;
		else
			FSM_state <= FSM_nxt_state;
	end

	// Load Hazard Flowchart logic
    always @(*)begin
      if(prevRt==0)
        LdHazard <= 0;  //no hazard when prev RT is 0
      else if(memReadEX==1)
        case({UseShamt,UseImmed})
            2'b00:
                if((prevRt==currRs) || (prevRt==currRt))
                    LdHazard <= 1;   //current Rt or Rs is same as prev Rt
                else
                    LdHazard <= 0;

            2'b10:
                if(prevRt==currRs)
                    LdHazard <= 1;   //current Rs for shamt is same as prev Rt
                else
                    LdHazard <= 0;

            2'b01:
                if(prevRt==currRs)
                    LdHazard <= 1;   //current Rs for Immediate is same as prev Rt
```

```
                else
                    LdHazard <= 0;

            default:
                LdHazard <= 0;
        endcase
    end

    // FSM next state and output logic
    always @(*) begin  // Combinatory logic
          case( FSM_state )

                NoHazard_state: begin    // Prioritize jump

                      if( Jump == 1'b1 ) begin
                            // Unconditional return to no hazard state
                            IF_write = 1'b0;
                            PC_write = 1'b1;
                            bubble = 1'b0;
                            addrSel = 2'b01;
                            FSM_nxt_state = Jump_state;
                      end

                      else if( LdHazard == 1'b1 ) begin
          // If hazard is detected, stop fetching new IF and stop incrementing PC and
writing into PC
                            IF_write = 1'b0;
                            PC_write = 1'b0;
                            bubble = 1'b1;
                            addrSel = 2'b00;
                            FSM_nxt_state = NoHazard_state;
                      end

                      else if (Branch == 1'b1)begin
                            // Go to branch zero and check ALU zero flag
                            IF_write = 1'b0;
                            PC_write = 1'b0;
                            bubble = 1'b0;
                            addrSel = 2'b00;
                            FSM_nxt_state = Branch_0_state;
                      end
                      else begin
                            // Normal state
                            IF_write = 1'b1;
                            PC_write = 1'b1;
                            bubble = 1'b0;
                            addrSel = 2'b00;
                            FSM_nxt_state = NoHazard_state;
                      end
                end
                Jump_state: begin
                      // Unconditional return to no hazard state
                      // Stop execution until jump is resolved completely
                      IF_write = 1'b1;
                      PC_write = 1'b1;
                      bubble = 1'b1;
                      addrSel = 2'b00;
                      FSM_nxt_state = NoHazard_state;
                end

                Branch_0_state: begin
                      if( ALUZero == 1'b0 ) begin
                            IF_write = 1'b1;
                            PC_write = 1'b1;
```

```
                        bubble = 1'b1;
                        addrSel = 2'b00;
                        FSM_nxt_state = NoHazard_state;
                    end
                    else if( ALUZero == 1'b1 ) begin
                        IF_write = 1'b0;
                        PC_write = 1'b1;
                        bubble = 1'b1;
                        addrSel = 2'b10;
                        FSM_nxt_state = Branch_1_state;
                    end
                end
                Branch_1_state: begin
                    // Unconditional return to no hazard state
                    IF_write = 1'b1;
                    PC_write = 1'b1;
                    bubble = 1'b1;
                    addrSel = 2'b00;
                    FSM_nxt_state = NoHazard_state;
                end
                default: begin
                    FSM_nxt_state = NoHazard_state;
                    PC_write = 1'bx;
                    IF_write = 1'bx;
                    bubble  = 1'bx;
                    addrSel = 2'bxx;
                end
            endcase
        end
endmodule
```

(d) Synthesize the hardware and ensure the code generates no warnings or errors. Provide the summary results of the synthesis process.

Synthesis Report: (HazardUnit module)

```
---------------------------------------------------------------------------------
Start Writing Synthesis Report
---------------------------------------------------------------------------------
Report BlackBoxes:
+-+--------------+----------+
| |BlackBox name |Instances |
+-+--------------+----------+
+-+--------------+----------+
Report Cell Usage:
+------+-----+------+
|      |Cell |Count |
+------+-----+------+
|1     |BUFG |     2|
|2     |LUT2 |     1|
|3     |LUT3 |     2|
|4     |LUT4 |     3|
|5     |LUT5 |     3|
|6     |LUT6 |     5|
|7     |FDRE |     2|
|8     |LDC  |     1|
|9     |IBUF |    23|
|10    |OBUF |     5|
+------+-----+------+
Report Instance Areas:
+------+---------+-------+------+
|      |Instance |Module |Cells |
```

```
+------+---------+-------+------+
|1     |top      |       |    47|
+------+---------+-------+------+
```

```
-----------------------------------------------------------------------------------------
Finished Writing Synthesis Report : Time (s): cpu = 00:00:12 ; elapsed = 00:00:19 . Memory
(MB): peak = 1770.926 ; gain = 279.355 ; free physical = 4161 ; free virtual = 42064
-----------------------------------------------------------------------------------------
```

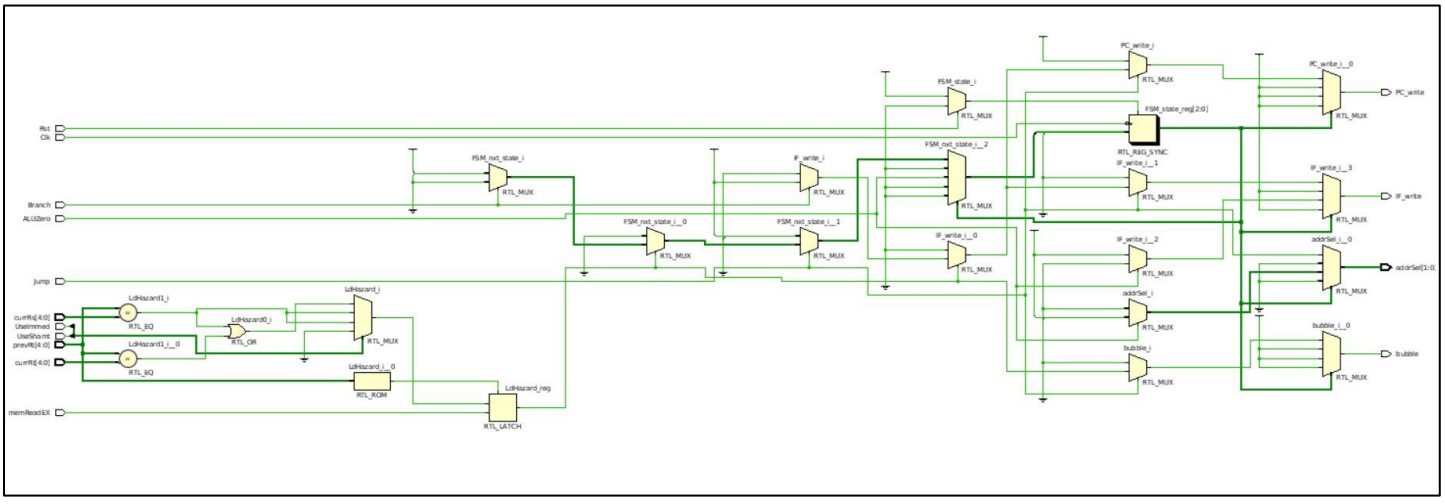
Synthesis finished with 0 errors, 0 critical warnings and 1 warnings.

```
Synthesis Optimization Runtime : Time (s): cpu = 00:00:12 ; elapsed = 00:00:19 . Memory (MB):
peak = 1770.926 ; gain = 279.355 ; free physical = 4163 ; free virtual = 42066
Synthesis Optimization Complete : Time (s): cpu = 00:00:12 ; elapsed = 00:00:19 . Memory (MB):
peak = 1770.926 ; gain = 279.355 ; free physical = 4173 ; free virtual = 42076
INFO: [Project 1-571] Translating synthesized netlist
INFO: [Netlist 29-17] Analyzing 1 Unisim elements for replacement
INFO: [Netlist 29-28] Unisim Transformation completed in 0 CPU seconds
INFO: [Project 1-570] Preparing netlist for logic optimization
INFO: [Opt 31-138] Pushed 0 inverter(s) to 0 load pin(s).
Netlist sorting complete. Time (s): cpu = 00:00:00 ; elapsed = 00:00:00 . Memory (MB): peak =
1909.938 ; gain = 0.000 ; free physical = 4026 ; free virtual = 41972
INFO: [Project 1-111] Unisim Transformation Summary:
  A total of 1 instances were transformed.
  LDC => LDCE: 1 instances
INFO: [Common 17-83] Releasing license: Synthesis
```

16 Infos, 1 Warnings, 0 Critical Warnings and 0 Errors encountered.

```
synth_design completed successfully
synth_design: Time (s): cpu = 00:00:16 ; elapsed = 00:00:24 . Memory (MB): peak = 1909.938 ;
gain = 447.297 ; free physical = 4078 ; free virtual = 42025
Netlist sorting complete. Time (s): cpu = 00:00:00 ; elapsed = 00:00:00 . Memory (MB): peak =
1909.938 ; gain = 0.000 ; free physical = 4077 ; free virtual = 42024
WARNING: [Constraints 18-5210] No constraints selected for write.
Resolution: This message can indicate that there are no constraints for the design, or it can
indicate that the used_in flags are set such that the constraints are ignored. This later case
is used when running synth_design to not write synthesis constraints to the resulting
checkpoint. Instead, project constraints are read when the synthesized design is opened.
INFO: [Common 17-1381] The checkpoint
'/home/grads/p/pranav_anantharam/lab6/project_1/project_1.runs/synth_1/HazardUnit.dcp' has been
generated.
INFO: [runtcl-4] Executing : report_utilization -file HazardUnit_utilization_synth.rpt -pb
HazardUnit_utilization_synth.pb
INFO: [Common 17-206] Exiting Vivado at Sun Dec  3 18:21:01 2023...
```

Compiled Hardware Schematic: (HazardUnit module)



3. Make slight modifications to the Register file, Control Unit, and ALU Control Module.

(a) The Register file must be modified to latch data on the positive edge of the clock rather than the negative edge.

Design Source code filename: RegisterFile.v

```
`timescale 1ns / 1ps
module RegisterFile( input [4:0] RA, input [4:0] RB, input [4:0] RW, input [31:0] BusW,
input RegWr, input Clk, output reg [31:0] BusA, output reg [31:0] BusB );
// RA - Bus A Address
// RB - Bus B Address
// RW - Write Port Address
// BusW - Write Port Data Input
// RegWr - Write enable signal
// Clk - Clock signal
// BusA - Bus A output data register
// BusB - Bus B output data register
reg [31:0] register_file [31:0]; // 32 elements x 32 bit wide element
// 0th register is wired to value 0
initial begin
	register_file[0] = 32'b0;
end
// Latch on positive edge of clock
always @(posedge Clk) begin
	// If Write Port address is not zero and Write enable signal is high
	if ( (RW != 5'b0) && (RegWr == 1'b1) ) begin
		// Write data into register
		register_file[RW] <= BusW;
	end
end
// Read values from registers
always @* begin
	BusA <= register_file[RA];
	BusB <= register_file[RB];
end
endmodule
```

(b) Modify the Control unit to include the UseShamt signal.

Design Source code filename: PipelinedControl.v

```
`timescale 1ns / 1ps

// MACROs
`define RTYPEOPCODE      6'b000000
`define LWOPCODE        6'b100011
`define SWOPCODE        6'b101011
`define BEQOPCODE       6'b000100
`define JOPCODE          6'b000010
`define ORIOPCODE       6'b001101
`define ADDIOPCODE       6'b001000
`define ADDIUOPCODE      6'b001001
`define ANDIOPCODE       6'b001100
`define LUIOPCODE       6'b001111
`define SLTIOPCODE       6'b001010
`define SLTIUOPCODE      6'b001011
`define XORIOPCODE       6'b001110
`define SLLFUNCCODE      6'b000000
`define SRLFUNCCODE      6'b000010
`define SRAFUNCCODE      6'b000011

`define AND     4'b0000
`define OR      4'b0001
`define ADD     4'b0010
`define SLL     4'b0011
`define SRL     4'b0100
`define SUB     4'b0110
`define SLT     4'b0111
`define ADDU    4'b1000
`define SUBU    4'b1001
`define XOR     4'b1010
`define SLTU    4'b1011
`define NOR     4'b1100
`define SRA     4'b1101
`define LUI     4'b1110
`define FUNC    4'b1111


module PipelinedControl(RegDst, MemToReg, RegWrite, MemRead, MemWrite, Branch, Jump,
SignExtend, ALUOp, Opcode, Func_code, Bubble, UseShamt, UseImmed);
   // Inputs
   input [5:0] Opcode;
   input [5:0] Func_code;
   input Bubble;

   // Outputs
   output RegDst;              // For dest register : 0-> 20-16 or  1-> 15-11
   output MemToReg;            // 0 -> ALU output written back to register, 1-> data
memory written to register eg. Load
   output RegWrite;            // Write enable for register
   output MemRead;             // Read enable for data memory
   output MemWrite;            // Write enable for data memory
   output Branch;              // Branch instruction or not
   output Jump;                // Jump instruction or not
   output SignExtend;          // Sign extend enable flag
   output [3:0] ALUOp;         // ALUopcode into ALU control to decide ALU operation
   output UseShamt;            // Signal for instructions with shift
   output UseImmed;            // Signal for instructions with immediate values
```

```
	reg   RegDst, MemToReg, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend,
UseShamt, UseImmed;
	reg  [3:0] ALUOp;
	always @ (Opcode or Bubble) begin
		if(Bubble) begin
			// Put your code here!
			RegDst      <= #2 1'b0;
			MemToReg    <= #2 1'b0;
			RegWrite    <= #2 1'b0;
			MemRead     <= #2 1'b0;
			MemWrite    <= #2 1'b0;
			Branch      <= #2 1'b0;
			Jump        <= #2 1'b0;
			SignExtend  <= #2 1'b0;
			ALUOp       <= #2 4'b0000;
			UseShamt    <= #2 1'b0;
			UseImmed    <= #2 1'b0;
		end
		else begin
			case(Opcode)
			   `RTYPEOPCODE: begin // R-type instructions: RegDst and Regwrite is 1
				RegDst      <= #2 1'b1;       // dest register
address = Instruction[15:11]
				MemToReg    <= #2 1'b0;
				RegWrite    <= #2 1'b1;       // Write to register
				MemRead     <= #2 1'b0;
				MemWrite    <= #2 1'b0;
				Branch      <= #2 1'b0;
				Jump        <= #2 1'b0;
				SignExtend  <= #2 1'bx;
				ALUOp       <= #2 `FUNC;
				// Set UseShamt to 1 for only SLL, SRA and SRL
operations
				if( (Func_code == `SLLFUNCCODE) || (Func_code ==
`SRLFUNCCODE) || (Func_code == `SRAFUNCCODE) )
					UseShamt <= #2 1'b1;
				else
					UseShamt <= #2 1'b0;    // Shamt not required
for other operations

				UseImmed    <= #2 1'b0;       // No Immediate value
used
			end

			/*add your code here. Reuse your code from lab 10 from the
file Lab10_SingleCycleControl.v */
			`LWOPCODE: begin
				RegDst      <= #2 1'b0;       // dest register
address = Instruction[20:16]
				MemToReg    <= #2 1'b1;       // load data from
memory to register
				RegWrite    <= #2 1'b1; // write data to register
				MemRead     <= #2 1'b1;       // read from memory
				MemWrite    <= #2 1'b0;
				Branch      <= #2 1'b0;
				Jump        <= #2 1'b0;
				SignExtend  <= #2 1'b1;       // sign extended
offset
				ALUOp       <= #2 `ADD;       // effective address
calculation using ADD
				UseShamt    <= #2 1'b0;
				UseImmed    <= #2 1'b1;       // Immediate value
for effective address calculation
			end
```

```
                        `SWOPCODE: begin
                              RegDst      <= #2 1'bx;       // No register writes
                              MemToReg    <= #2 1'bx;       // No loading of data
from memory to register
                              RegWrite    <= #2 1'b0;
                              MemRead     <= #2 1'b0;
                              MemWrite    <= #2 1'b1;  // Write data into memory
                              Branch      <= #2 1'b0;
                              Jump        <= #2 1'b0;
                              SignExtend  <= #2 1'b1;       // sign extended
offset
                              ALUOp       <= #2 `ADD;       // effective address
calculation using ADD
                              UseShamt    <= #2 1'b0;
                              UseImmed    <= #2 1'b1;       // Immediate value
for effective address calculation
                        end

                        `BEQOPCODE: begin
                              RegDst      <= #2 1'bx;       // No register writes
                              MemToReg    <= #2 1'bx;       // No loading of data
from memory to register
                              RegWrite    <= #2 1'b0;
                              MemRead     <= #2 1'b0;
                              MemWrite    <= #2 1'b0;
                              Branch      <= #2 1'b1;       // Branch instruction
                              Jump        <= #2 1'b0;
                              SignExtend  <= #2 1'b1;
                              ALUOp       <= #2 `SUB;       // BEQ subtraction
result used to decide branch taken / not taken
                              UseShamt    <= #2 1'b0;       // No shift
operations required
                              UseImmed    <= #2 1'b0;       // No Immediate
values used
                        end

                        `JOPCODE: begin
                              RegDst      <= #2 1'bx;       // No register writes
                              MemToReg    <= #2 1'bx;       // No loading of data
from memory to register
                              RegWrite    <= #2 1'b0;
                              MemRead     <= #2 1'b0;
                              MemWrite    <= #2 1'b0;
                              Branch      <= #2 1'b0;
                              Jump        <= #2 1'b1;       // Jump instruction
                              SignExtend  <= #2 1'bx;       // Sign extension not
used for JUMP
                              ALUOp       <= #2 4'bxxxx;
                              UseShamt    <= #2 1'b0;       // No shift
operations required
                              UseImmed    <= #2 1'b0;       // No Immediate
values used
                        end

                        `ORIOPCODE: begin
                              RegDst      <= #2 1'b0;       // dest register
address = Instruction[20:16]
                              MemToReg    <= #2 1'b0;
                              RegWrite    <= #2 1'b1;  // write data to register
                              MemRead     <= #2 1'b0;
                              MemWrite    <= #2 1'b0;
                              Branch      <= #2 1'b0;
                              Jump        <= #2 1'b0;
```

```
                        SignExtend  <= #2 1'b0;       // Not required for
logical operations
                        ALUOp       <= #2 `OR;        // Logical OR
operation
                        UseShamt    <= #2 1'b0;
                        UseImmed    <= #2 1'b1; // Immediate values used
                    end

                    `ADDIOPCODE: begin
                        RegDst      <= #2 1'b0;       // dest register
address = Instruction[20:16]
                        MemToReg    <= #2 1'b0;
                        RegWrite    <= #2 1'b1; // write data to register
                        MemRead     <= #2 1'b0;
                        MemWrite    <= #2 1'b0;
                        Branch      <= #2 1'b0;
                        Jump        <= #2 1'b0;
                        SignExtend  <= #2 1'b1;       // Sign extended for
immediate values
                        ALUOp       <= #2 `ADD;       // ADD operation
                        UseShamt    <= #2 1'b0;
                        UseImmed    <= #2 1'b1; // Immediate values used
                    end

                    `ADDIUOPCODE: begin
                        RegDst      <= #2 1'b0;       // dest register
address = Instruction[20:16]
                        MemToReg    <= #2 1'b0;
                        RegWrite    <= #2 1'b1; // write data to register
                        MemRead     <= #2 1'b0;
                        MemWrite    <= #2 1'b0;
                        Branch      <= #2 1'b0;
                        Jump        <= #2 1'b0;
                        SignExtend  <= #2 1'b0;
                        ALUOp       <= #2 `ADDU;      // ADDU operation
                        UseShamt    <= #2 1'b0;
                        UseImmed    <= #2 1'b1; // Immediate values used
                    end

                    `ANDIOPCODE: begin
                        RegDst      <= #2 1'b0;       // dest register
address = Instruction[20:16]
                        MemToReg    <= #2 1'b0;
                        RegWrite    <= #2 1'b1; // write data to register
                        MemRead     <= #2 1'b0;
                        MemWrite    <= #2 1'b0;
                        Branch      <= #2 1'b0;
                        Jump        <= #2 1'b0;
                        SignExtend  <= #2 1'b0;
                        ALUOp       <= #2 `AND;       // Logical AND
operation
                        UseShamt    <= #2 1'b0;
                        UseImmed    <= #2 1'b1; // Immediate values used
                    end

                    `LUIOPCODE: begin
                        RegDst      <= #2 1'b0;       // dest register
address = Instruction[20:16]
                        MemToReg    <= #2 1'b0;
                        RegWrite    <= #2 1'b1; // write data to register
                        MemRead     <= #2 1'b0;
                        MemWrite    <= #2 1'b0;
                        Branch      <= #2 1'b0;
                        Jump        <= #2 1'b0;
```

```
                SignExtend  <= #2 1'b1;
                ALUOp       <= #2 `LUI;        // LUI operation
                UseShamt    <= #2 1'b0;
                UseImmed    <= #2 1'b1; // Immediate values used
            end

            `SLTIOPCODE: begin
                RegDst      <= #2 1'b0;        // dest register
address = Instruction[20:16]
                MemToReg    <= #2 1'b0;
                RegWrite    <= #2 1'b1; // write data to register
                MemRead     <= #2 1'b0;
                MemWrite    <= #2 1'b0;
                Branch      <= #2 1'b0;
                Jump        <= #2 1'b0;
                SignExtend  <= #2 1'b1;        // Sign extended for
immediate values
                ALUOp       <= #2 `SLT;        // SLT operation
                UseShamt    <= #2 1'b0;
                UseImmed    <= #2 1'b1; // Immediate values used
            end

            `SLTIUOPCODE: begin
                RegDst      <= #2 1'b0;        // dest register
address = Instruction[20:16]
                MemToReg    <= #2 1'b0;
                RegWrite    <= #2 1'b1; // write data to register
                MemRead     <= #2 1'b0;
                MemWrite    <= #2 1'b0;
                Branch      <= #2 1'b0;
                Jump        <= #2 1'b0;
                SignExtend  <= #2 1'b0;
                ALUOp       <= #2 `SLTU;       // SLTU operation
                UseShamt    <= #2 1'b0;
                UseImmed    <= #2 1'b1; // Immediate values used
            end

            `XORIOPCODE: begin
                RegDst      <= #2 1'b0;        // dest register
address = Instruction[20:16]
                MemToReg    <= #2 1'b0;
                RegWrite    <= #2 1'b1; // write data to register
                MemRead     <= #2 1'b0;
                MemWrite    <= #2 1'b0;
                Branch      <= #2 1'b0;
                Jump        <= #2 1'b0;
                SignExtend  <= #2 1'b0;
                ALUOp       <= #2 `XOR; // Logical XOR operation
                UseShamt    <= #2 1'b0;
                UseImmed    <= #2 1'b1; // Immediate values used
            end

            default: begin
                RegDst      <= #2 1'bx;
                MemToReg    <= #2 1'bx;
                RegWrite    <= #2 1'bx;
                MemRead     <= #2 1'bx;
                MemWrite    <= #2 1'bx;
                Branch      <= #2 1'bx;
                Jump        <= #2 1'bx;
                SignExtend  <= #2 1'bx;
                ALUOp       <= #2 4'bxxxx;
                UseShamt    <= #2 1'bx;
                UseImmed    <= #2 1'bx;
```

```
                        end
                endcase
        end
    end
endmodule
```

(c) For the ALU Control Module, you must remove the UseShamt logic as it is now generated in the main control unit. (No changes were needed to be made in this file since Single Cycle Processor implementation)

Design Source code filename: ALUControl.v

```
`timescale 1ns / 1ps

// MACROs
`define AND 4'b0000
`define OR 4'b0001
`define ADD 4'b0010
`define SLL 4'b0011
`define SRL 4'b0100
`define SUB 4'b0110
`define SLT 4'b0111
`define ADDU 4'b1000
`define SUBU 4'b1001
`define XOR 4'b1010
`define SLTU 4'b1011
`define NOR 4'b1100
`define SRA 4'b1101
`define LUI 4'b1110

`define SLLFunc 6'b000000
`define SRLFunc 6'b000010
`define SRAFunc 6'b000011
`define ADDFunc 6'b100000
`define ADDUFunc 6'b100001
`define SUBFunc 6'b100010
`define SUBUFunc 6'b100011
`define ANDFunc 6'b100100
`define ORFunc 6'b100101
`define XORFunc 6'b100110
`define NORFunc 6'b100111
`define SLTFunc 6'b101010
`define SLTUFunc 6'b101011
`define MULAFunc 6'b111000

module ALUControl(ALUCtrl, ALUop, FuncCode);
// Inputs
input [3:0] ALUop;
input [5:0] FuncCode;

// Outputs
output reg [3:0] ALUCtrl;

always@(*) begin
// If ALUop is not equal to 4'b1111, ALUop is passed directly to the ALU
if( ALUop != 4'b1111 )
        ALUCtrl <= ALUop;
else
        // Function code is used to determine the ALU control code
```

```
        case(FuncCode)
                `SLLFunc: ALUCtrl <= SLL;
                `SRLFunc: ALUCtrl <= SRL;
                `SRAFunc: ALUCtrl <= SRA;
                `ADDFunc: ALUCtrl <= ADD;
                `ADDUFunc: ALUCtrl <= ADDU;
                `SUBFunc: ALUCtrl <= SUB;
                `SUBUFunc: ALUCtrl <= SUBU;
                `ANDFunc: ALUCtrl <= AND;
                `ORFunc: ALUCtrl <= OR;
                `XORFunc: ALUCtrl <= XOR;
                `NORFunc: ALUCtrl <= NOR;
                `SLTFunc: ALUCtrl <= SLT;
                `SLTUFunc: ALUCtrl <= SLTU;
               default: ALUCtrl <= 4'bx;
        endcase
end
endmodule
```

---

4. Design the remainder of the MIPS pipelined datapath.

(a) Describe the pipelined MIPS datapath in Verilog.

Design Source code filename: PipelinedProc.v

```
`timescale 1ns / 1ps


module PipelinedProc(CLK, Reset_L, startPC, dMemOut);

// Inputs
input CLK;
input Reset_L;
input [31:0] startPC;

// Output
output [31:0] dMemOut;

wire [31:0] Data;
wire [31:0] Address;
reg [31:0] updated_PC_Address;

// Output of Hazard Unit
wire [1:0] addrSel; // Mux select for PC
wire IF_write;
wire PC_write;
wire bubble;

// Pipelined Control wires
wire RegDst; // Selecting the actual bits of register file. If ==1 register file = 15 to
11 else  20 to 16.
wire MemToReg; // Data of memory is written if =1 or data of alu is written if =0
wire RegWrite; // To write to the registers in the register file
wire MemRead; // To read from the memory
wire MemWrite; // To write into the memory unit. Data to be written comes from bus B
wire Branch; // To activate branch control
wire Jump; // Activate jump control
```

```
wire SignExtend; // Extend the sign of the immediate value
wire UseShamt; // To change the source registers in case of shift operations
wire UseImmed;// Indicate a immedieate type of instruction.
wire [3:0]ALUOp_IF_ID;// Type of operation to be performed.


// Forwarding Unit
wire [1:0] AluOpCtrlA_ID;
wire [1:0] AluOpCtrlB_ID;
wire DataMemForwardCtrl_EX_IF_ID;
wire DataMemForwardCtrl_MEM_IF_ID;


// Sign extended data
wire[31:0] sign_extension_data;

// Register wires Fetch-Decode stage
wire [31:0] Register_file_A_IF_ID, Register_file_B_IF_ID;
wire [31:0] BusW_WB;
wire [4:0] RA_IF_ID, RB_IF_ID, RW_WB;
wire[4:0] write_register_select_IF_ID;
wire [31:0] write_register_data_IF_ID;
wire [5:0] Opcode_IF_ID;
wire [4:0] RS_IF_ID;
wire [4:0] RT_IF_ID;
wire [4:0] RD_IF_ID;
wire [5:0] FUNC_IF_ID;
reg [31:0] IM_IF_ID;
wire [25:0] Jump_IF_ID;
wire [15:0] Immedi_IF_ID;
reg  [31:0] Normal_PC_IF_ID;
wire RegWr_WB;
wire Clk;


// Stage 3 ID -> Execute
reg RegDst_ID_EX;
reg MemToReg_ID_EX;
reg RegWrite_ID_EX;
reg MemRead_ID_EX;
reg MemWrite_ID_EX;
reg [3:0]ALUOp_ID_EX;
reg [1:0] AluOpCtrlA_ID_EX;
reg [1:0] AluOpCtrlB_ID_EX;
reg DataMemForwardCtrl_EX_ID_EX;
reg DataMemForwardCtrl_MEM_ID_EX;
reg [20:0] IM_20_0_ID_EX;
(* keep = "true"*)reg [31:0] Sign_Extended_ID_EX;
reg [31:0] Registers_A_ID_EX;
reg [31:0] Registers_B_ID_EX;

wire [5:0] Funccode_ID_EX;
wire [4:0] RT_ID_EX;
wire [4:0] RD_ID_EX;
wire [4:0] Shamt_ID_EX;
wire [4:0] RW_ID_EX;
wire [31:0] Data_Memory_Input_ID_EX;


// ALU Control wires
wire [31:0] ALU_OUT;
reg [31:0] ALU_IN1;
reg [31:0] ALU_IN2;
wire ALU_Zero;
```

```
wire [3:0] ALU_control;


// Stage 4
// Data Memory
wire [31:0] Data_memory_out;
reg [31:0] Data_Memory_Input_EX_MEM;
reg [31:0] ALU_OUT_EX_MEM;
reg [4:0] RW_EX_MEM;
wire [31:0] Data_Memory_actual_in;

reg MemToReg_EX_MEM;
reg RegWrite_EX_MEM;
reg MemRead_EX_MEM;
reg MemWrite_EX_MEM;

reg DataMemForwardCtrl_MEM_EX_MEM;

// Stage 5
// Writeback stage
reg MemToReg_MEM_WB;
reg RegWrite_MEM_WB;
reg [4:0] RW_MEM_WB;
reg [31:0] DataOut_MEM_WB;
reg [31:0] ALU_OUT_MEM_WB;
wire [31:0] Register_W_MEM_WB;

// Instruction Memory
InstructionMemory IM1(.Data(Data), .Address(updated_PC_Address));

// Pipelined Control
PipelinedControl PCC1 (.RegDst(RegDst), .MemToReg(MemToReg), .RegWrite(RegWrite),
.MemRead(MemRead), .MemWrite(MemWrite), .Branch(Branch), .Jump(Jump),
.SignExtend(SignExtend), .ALUOp(ALUOp_IF_ID), .Opcode(Opcode_IF_ID),
.Func_code(FUNC_IF_ID), .Bubble(bubble), .UseShamt(UseShamt), .UseImmed(UseImmed) );


// Hazard Unit
HazardUnit Hazard (.IF_write(IF_write), .PC_write(PC_write), .bubble(bubble),
.addrSel(addrSel), .Jump(Jump), .Branch(Branch), .ALUZero(ALU_Zero),
.memReadEX(MemRead_ID_EX), .currRs(RS_IF_ID), .currRt(RT_IF_ID), .prevRt(RT_ID_EX),
.UseShamt(UseShamt), .UseImmed(UseImmed), .Clk(CLK), .Rst(Reset_L));

// Register File
RegisterFile RF1 ( .RA(RS_IF_ID), .RB(RT_IF_ID), .RW( RW_MEM_WB ),
.BusW(Register_W_MEM_WB), .RegWr(RegWrite_MEM_WB), .Clk(CLK),
.BusA(Register_file_A_IF_ID), .BusB(Register_file_B_IF_ID) );

// Forwarding Unit
ForwardingUnit Forward ( .UseShamt(UseShamt), .UseImmed(UseImmed),
.ID_Rs(RS_IF_ID),.ID_Rt(RT_IF_ID), .EX_Rw(RW_ID_EX),.MEM_Rw(RW_EX_MEM),
.EX_RegWrite(RegWrite_ID_EX), .MEM_RegWrite(RegWrite_EX_MEM), .AluOpCtrlA(AluOpCtrlA_ID),
.AluOpCtrlB(AluOpCtrlB_ID), .DataMemForwardCtrl_EX(DataMemForwardCtrl_EX_IF_ID),
.DataMemForwardCtrl_MEM(DataMemForwardCtrl_MEM_IF_ID) );

// ALU
ALU ALU1 (.BusW(ALU_OUT), .Zero(ALU_Zero), .BusA(ALU_IN1), .BusB(ALU_IN2),
.ALUCtrl(ALU_control));

// ALU Control Unit
ALUControl AC1 (.ALUCtrl(ALU_control), .ALUop(ALUOp_ID_EX), .FuncCode(Funccode_ID_EX));

// Data Memory
```

```
DataMemory DM1 (.ReadData(Data_memory_out), .Address(ALU_OUT_EX_MEM[7:2]),
.WriteData(Data_Memory_actual_in), .MemoryRead(MemRead_EX_MEM),
.MemoryWrite(MemWrite_EX_MEM), .Clock(CLK));


// Program Counter Logic
always @ (negedge CLK or negedge Reset_L) begin
	if (~Reset_L)
		updated_PC_Address <= startPC;
	else if (PC_write)
		updated_PC_Address <= Address;
end


// Stage 2
// Register File to ID stage
always @ (negedge CLK or negedge Reset_L) begin
	if(~Reset_L) begin
		IM_IF_ID <= 32'b0;
		Normal_PC_IF_ID <= 32'b0;
	end
	else if(IF_write) begin
		IM_IF_ID <= Data;
		Normal_PC_IF_ID <= updated_PC_Address + 4;
	end
end


// STAGE 2
// Instruction Decode -> Execute Stage
always @ (negedge CLK or negedge Reset_L) begin
	if(~Reset_L) begin
		RegDst_ID_EX<=1'b0;
		MemToReg_ID_EX<=1'b0;
		RegWrite_ID_EX<=1'b0;
		MemRead_ID_EX<=1'b0;
		MemWrite_ID_EX<=1'b0;
		ALUOp_ID_EX<=4'b0;
		AluOpCtrlA_ID_EX<=2'b0;
		AluOpCtrlB_ID_EX<=2'b0;
		DataMemForwardCtrl_EX_ID_EX<=1'b0;
		DataMemForwardCtrl_MEM_ID_EX<=1'b0;
		IM_20_0_ID_EX<=21'b0;
		Sign_Extended_ID_EX<=32'b0;
		Registers_A_ID_EX<=32'b0;
		Registers_B_ID_EX<=32'b0;
	end

	else if(bubble) begin		// If bubble, stop execution
		RegDst_ID_EX<=1'b0;
		MemToReg_ID_EX<=1'b0;
		RegWrite_ID_EX<=1'b0;
		MemRead_ID_EX<=1'b0;
		MemWrite_ID_EX<=1'b0;
		ALUOp_ID_EX<=4'b0;
		AluOpCtrlA_ID_EX<=2'b0;
		AluOpCtrlB_ID_EX<=2'b0;
		DataMemForwardCtrl_EX_ID_EX<=1'b0;
		DataMemForwardCtrl_MEM_ID_EX<=1'b0;
		IM_20_0_ID_EX<=21'b0;
		Sign_Extended_ID_EX<=32'b0;
		Registers_A_ID_EX<=32'b0;
		Registers_B_ID_EX<=32'b0;
	end
```

```
	else begin
		RegDst_ID_EX<=RegDst;
		MemToReg_ID_EX<=MemToReg;
		RegWrite_ID_EX<=RegWrite;
		MemRead_ID_EX<=MemRead;
		MemWrite_ID_EX<=MemWrite;
		ALUOp_ID_EX<=ALUOp_IF_ID;
		AluOpCtrlA_ID_EX<=AluOpCtrlA_ID;
		AluOpCtrlB_ID_EX<=AluOpCtrlB_ID;
		DataMemForwardCtrl_EX_ID_EX<=DataMemForwardCtrl_EX_IF_ID;
		DataMemForwardCtrl_MEM_ID_EX<=DataMemForwardCtrl_MEM_IF_ID;
		IM_20_0_ID_EX<=IM_IF_ID[20:0];
		Sign_Extended_ID_EX<=sign_extension_data;
		Registers_A_ID_EX<=Register_file_A_IF_ID;
		Registers_B_ID_EX<=Register_file_B_IF_ID;
	end
end


// STAGE 3
// Execute Stage
// Input 1
always @(*)begin
	case (AluOpCtrlA_ID_EX)
		2'b00: ALU_IN1 = {27'b0, Shamt_ID_EX};
		2'b01: ALU_IN1 = Register_W_MEM_WB;
		2'b10: ALU_IN1 = ALU_OUT_EX_MEM;
		2'b11: ALU_IN1 = Registers_A_ID_EX;
	endcase
end

// Input 2
always @(*)begin
	case (AluOpCtrlB_ID_EX)
		2'b00: ALU_IN2 = Sign_Extended_ID_EX;
		2'b01: ALU_IN2 = Register_W_MEM_WB;
		2'b10: ALU_IN2 = ALU_OUT_EX_MEM;
		2'b11: ALU_IN2 = Registers_B_ID_EX;
	endcase
end


// Execute -> Memory Stage
always @ (negedge CLK or negedge Reset_L) begin
	if (~Reset_L)begin
		Data_Memory_Input_EX_MEM <= 32'b0;
		ALU_OUT_EX_MEM	<=32'b0;
		RW_EX_MEM	<=5'b0;
		MemToReg_EX_MEM	<=1'b0;
		RegWrite_EX_MEM	<=1'b0;
		MemRead_EX_MEM	<=1'b0;
		MemWrite_EX_MEM	<=1'b0;
		DataMemForwardCtrl_MEM_EX_MEM<=1'b0;
	end

	else begin
		Data_Memory_Input_EX_MEM<=	Data_Memory_Input_ID_EX;
		ALU_OUT_EX_MEM	<=	ALU_OUT;
		RW_EX_MEM	<=	RW_ID_EX;
		MemToReg_EX_MEM	<=	MemToReg_ID_EX;
		RegWrite_EX_MEM	<=	RegWrite_ID_EX;
		MemRead_EX_MEM	<=	MemRead_ID_EX;
		MemWrite_EX_MEM	<=	MemWrite_ID_EX;
```

```
            DataMemForwardCtrl_MEM_EX_MEM <=DataMemForwardCtrl_MEM_ID_EX;
      end
end


// Memory -> Writeback
always @(negedge CLK or negedge Reset_L)begin
      if (~Reset_L)begin
            MemToReg_MEM_WB<=1'b0;
            RegWrite_MEM_WB<=1'b0;
            RW_MEM_WB<=5'b0;
            DataOut_MEM_WB<=32'b0;
            ALU_OUT_MEM_WB<=32'b0;
      end
      else begin
            MemToReg_MEM_WB<=MemToReg_EX_MEM;
            RegWrite_MEM_WB<=RegWrite_EX_MEM;
            RW_MEM_WB<=RW_EX_MEM;
            DataOut_MEM_WB<=Data_memory_out;
            ALU_OUT_MEM_WB<=ALU_OUT_EX_MEM;
      end
end


assign Opcode_IF_ID = IM_IF_ID[31:26];
assign FUNC_IF_ID = IM_IF_ID [5:0];
assign RS_IF_ID = IM_IF_ID[25:21];
assignRT_IF_ID = IM_IF_ID[20:16];
assign Jump_IF_ID = IM_IF_ID[25:0];
assign Immedi_IF_ID = IM_IF_ID[15:0];

// Sign Extension
assign sign_extension_data = SignExtend ? {{16{IM_IF_ID[15]}},IM_IF_ID[15:0]} :
{{16{1'b0}},IM_IF_ID[15:0]}  ;

// Address Select Logic
assign Address = (addrSel==2'b00) ? updated_PC_Address+4 :(addrSel == 2'b01) ?
{Normal_PC_IF_ID [31:28], Jump_IF_ID, 2'b0} : Normal_PC_IF_ID + (Sign_Extended_ID_EX[30:0]
<< 2);

assign RT_ID_EX = IM_20_0_ID_EX[20:16];
assign RD_ID_EX = IM_20_0_ID_EX[15:11];
assign Shamt_ID_EX = IM_20_0_ID_EX[10:6];
assign Funccode_ID_EX = IM_20_0_ID_EX [5:0];

assign RW_ID_EX = RegDst_ID_EX ? RD_ID_EX : RT_ID_EX;
assign Data_Memory_Input_ID_EX = DataMemForwardCtrl_EX_ID_EX ?  Register_W_MEM_WB :
Registers_B_ID_EX;

// Stage 4
// Memory
assign Data_Memory_actual_in = DataMemForwardCtrl_MEM_EX_MEM ? Register_W_MEM_WB
:Data_Memory_Input_EX_MEM;

// Stage 5
// Write back
assign Register_W_MEM_WB = MemToReg_MEM_WB ? DataOut_MEM_WB : ALU_OUT_MEM_WB;

// Final Output
assign dMemOut = DataOut_MEM_WB;

endmodule
```

Design Source code filename: InstructionMemory.v

```
`timescale 1ns / 1ps
/*
 * Module: InstructionMemory
 *
 * Implements read-only instruction memory
 * Memory contents are initialized from the file "ImemInit.v"
 */
module InstructionMemory(Data, Address);
	parameter T_rd = 20;
	parameter MemSize = 40;

	output [31:0] Data;
	input [31:0] Address;
	reg [31:0] Data;

	/*
	 * ECEN 651 Processor Test Functions
	 * Texas A&M University
	 */

	always @ (Address) begin
		case(Address)
		/*
		 * Test Program 1:
		 * Sums $a0 words starting at $a1.  Stores the sum at the end of the array
		 * Tests add, addi, lw, sw, beq
		 */

		/*
		main:
				li $t0, 50				# Initialize the array to
(50, 40, 30)
				sw $t0, 0($0)				# Store first value
				li $t0, 40
				sw $t0, 4($0)				# Store Second Value
				li $t0, 30
				sw $t0, 8($0)				# Store Third Value
				li $a0, 0				# address of array
				li $a1, 3				# 3 values to sum
		TestProg1:
				add $t0, $0, $0			# This is the sum
				add $t1, $0, $a0			# This is our array pointer
				add $t2, $0, $0			# This is our index counter
		P1Loop:		beq $t2, $a1, P1Done	 # Our loop
				lw	$t3, 0($t1)				# Load Array[i]
				add $t0, $t0, $t3			# Add it into the sum
				add $t1, $t1, 4			# Next address
				add $t2, $t2, 1			# Next index
				j P1Loop					# Jump to loop
		P1Done:		sw $t0, 0($t1)			# Store the sum at end of array
				lw $t0, 12($0)			# Load Final Value
				nop					# Complete
				add $0, $s0, $s0		# do nothing
		*/
			32'h00: Data = 32'h34080032;
			32'h04: Data = 32'hac080000;
			32'h08: Data = 32'h34080028;
			32'h0C: Data = 32'hac080004;
			32'h10: Data = 32'h3408001e;
			32'h14: Data = 32'hac080008;
			32'h18: Data = 32'h34040000;
```

```
            32'h1C: Data = 32'h34050003;
            32'h20: Data = 32'h00004020;
            32'h24: Data = 32'h00044820;
            32'h28: Data = 32'h00005020;
            32'h2C: Data = 32'h11450005;
            32'h30: Data = 32'h8d2b0000;
            32'h34: Data = 32'h010b4020;
            32'h38: Data = 32'h21290004;
            32'h3C: Data = 32'h214a0001;
            32'h40: Data = 32'h0800000b;
            32'h44: Data = 32'had280000;
            32'h48: Data = 32'h8c08000c;
            32'h4C: Data = 32'h00000000;
            32'h50: Data = 32'h02100020;

        /*
         * Test Program 2:
         * Does some arithmetic computations and stores result in memory
         */

        /*
        main2:
                    li    $a0, 32                           # Address of memory
to store result
        TestProg2:
                    addi $2, $0, 1                      # $2 = 1
                    sub   $3, $0, $2              # $3 = -1
                    slt   $5, $3, $0              # $5 = 1
                    add   $6, $2, $5        # $6 = 2
                    or    $7, $5, $6                    # $7 = 3
                    sub   $8, $5, $7              # $8 = -2
                    and   $9, $8, $7              # $9 = 2
                    sw    $9, 0($a0)                    # Store $9 in DMem[8]
                    lw  $9, 32($0)                # Load Final Value
                    nop                                 # Complete
        */
            32'h60: Data = 32'h34040020;
            32'h64: Data = 32'h20020001;
            32'h68: Data = 32'h00021822;
            32'h6C: Data = 32'h0060282a;
            32'h70: Data = 32'h00453020;
            32'h74: Data = 32'h00a63825;
            32'h78: Data = 32'h00a74022;
            32'h7C: Data = 32'h01074824;
            32'h80: Data = 32'hac890000;
            32'h84: Data = 32'h8c090020;
            32'h88: Data = 32'h00000000;

        /*
         * Test Program 3
         * Test Immediate Function
         */

        /*
                    TestProg3:
                    li $a0, 0xfeedbeef      # $a0 = 0xfeedbeef
                    sw $a0, 36($0)                  # Store $a0 in DMem[9]
                    addi $a1, $a0, -2656          # $a1 = 0xfeedb48f
                    sw $a1, 40($0)                  # Store $a1 in DMem[10]
                    addiu $a1, $a0, -2656   # $a1 = 0xfeeeb48f
                    sw $a1, 44($0)                  # Store $a1 in DMem[11]
                    andi $a1, $a0, 0xf5a0   # $a1 = 0xb4a0
                    sw $a1, 48($0)                  # Store $a1 in DMem[12]
                    sll $a1, $a0, 5               # $a1 = 0xddb7dde0
```

```
                sw $a1, 52($0)                      # Store $a1 in DMem[13]
                srl $a1, $a0, 5                 # $a1 = 0x07f76df7
                sw $a1, 56($0)                      # Store $a1 in DMem[14]
                sra $a1, $a0, 5                 # $a1 = 0xfff76df7
                sw $a1, 60($0)                      # Store $a1 in DMem[15]
                slti $a1, $a0, 1                # $a1 = 1
                sw $a1, 64($0)                      # Store $a1 in DMem[16]
                slti $a1, $a1, -1               # $a1 = 0
                sw $a1, 68($0)                      # Store $a1 in DMem[17]
                sltiu $a1, $a0, 1               # $a1 = 0
                sw $a1, 72($0)                      # Store $a1 in DMem[18]
                sltiu $a1, $a1, -1      # $a1 = 1
                sw $a1, 76($0)                      # Store $a1 in DMem[19]
                xori $a1, $a0, 0xf5a0   # $a1 = 0xfeed4b4f
                sw $a1, 80($0)                      # Store $a1 in DMem[20]
                lw $a0, 36($0)                      # Load Value to test
                lw $a1, 40($0)                      # Load Value to test
                lw $a1, 44($0)                      # Load Value to test
                lw $a1, 48($0)                      # Load Value to test
                lw $a1, 52($0)                      # Load Value to test
                lw $a1, 56($0)                      # Load Value to test
                lw $a1, 60($0)                      # Load Value to test
                lw $a1, 64($0)                      # Load Value to test
                lw $a1, 68($0)                      # Load Value to test
                lw $a1, 72($0)                      # Load Value to test
                lw $a1, 76($0)                      # Load Value to test
                lw $a1, 80($0)                      # Load Value to test
                nop                                         # Complete
*/
        32'hA0: Data = 32'h3c01feed;
        32'hA4: Data = 32'h3424beef;
        32'hA8: Data = 32'hac040024;
        32'hAC: Data = 32'h2085f5a0;
        32'hB0: Data = 32'hac050028;
        32'hB4: Data = 32'h2485f5a0;
        32'hB8: Data = 32'hac05002c;
        32'hBC: Data = 32'h3085f5a0;
        32'hC0: Data = 32'hac050030;
        32'hC4: Data = 32'h00042940;
        32'hC8: Data = 32'hac050034;
        32'hCC: Data = 32'h00042942;
        32'hD0: Data = 32'hac050038;
        32'hD4: Data = 32'h00042943;
        32'hD8: Data = 32'hac05003c;
        32'hDC: Data = 32'h28850001;
        32'hE0: Data = 32'hac050040;
        32'hE4: Data = 32'h28a5ffff;
        32'hE8: Data = 32'hac050044;
        32'hEC: Data = 32'h2c850001;
        32'hF0: Data = 32'hac050048;
        32'hF4: Data = 32'h2ca5ffff;
        32'hF8: Data = 32'hac05004c;
        32'hFC: Data = 32'h3885f5a0;
        32'h100: Data = 32'hac050050;
        32'h104: Data = 32'h8c040024;
        32'h108: Data = 32'h8c050028;
        32'h10C: Data = 32'h8c05002c;
        32'h110: Data = 32'h8c050030;
        32'h114: Data = 32'h8c050034;
        32'h118: Data = 32'h8c050038;
        32'h11C: Data = 32'h8c05003c;
        32'h120: Data = 32'h8c050040;
        32'h124: Data = 32'h8c050044;
        32'h128: Data = 32'h8c050048;
```

```
            32'h12C: Data = 32'h8c05004c;
            32'h130: Data = 32'h8c050050;
            32'h134: Data = 32'h00000000;

        /*
         * Test Program 4
         * Test jal and jr
         */
        /*
        TestProg4:
                    li $t1, 0xfeed                  # $t1 = 0xfeed
                    li $t0, 0x190                   # Load address of P4jr
                    jr $t0                          # Jump to P4jr
                    li $t1, 0                       # Check for failure to jump
        P4jr:  sw $t1, 84($0)                   # $t1 should be 0xfeed if
successful
                    li $t0, 0xcafe                  # $t0 = 0xcafe
                    jal P4Jal                       # Jump to P4Jal
                    li $t0, 0xbabe                  # Check for failure to jump
        P4Jal: sw $t0, 88($0)                # $t0 should be 0xcafe if successful
                    li $t2, 0xface                  # $t2 = 0xface
                    j P4Skip                              # Jump to P4Skip
                    li $t2, 0
        P4Skip:     sw $t2, 92($0)                  # $t2 should be 0xface if
successful
                    sw $ra, 96($0)                  # Store $ra
                    lw $t0, 84($0)                  # Load value for check
                    lw $t1, 88($0)                  # Load value for check
                    lw $t2, 92($0)                  # Load value for check
                    lw $ra, 96($0)                  # Load value for check

        */
            32'h180: Data = 32'h3409feed;
            32'h184: Data = 32'h34080190;
            32'h188: Data = 32'h01000008;
            32'h18C: Data = 32'h34090000;
            32'h190: Data = 32'hac090054;
            32'h194: Data = 32'h3408cafe;
            32'h198: Data = 32'h0c000068;
            32'h19C: Data = 32'h3408babe;
            32'h1A0: Data = 32'hac080058;
            32'h1A4: Data = 32'h340aface;
            32'h1A8: Data = 32'h0800006c;
            32'h1AC: Data = 32'h340a0000;
            32'h1B0: Data = 32'hac0a005c;
            32'h1B4: Data = 32'hac1f0060;
            32'h1B8: Data = 32'h8c080054;
            32'h1BC: Data = 32'h8c090058;
            32'h1C0: Data = 32'h8c0a005c;
            32'h1C4: Data = 32'h8c1f0060;
            32'h1C8: Data = 32'h00000000;


        /*
         * Test Program 5
         * Tests Overflow Exceptions
         */

        /*
        Test5-1:
                    li $t0, -2147450880
                    add $t0, $t0, $t0
                    lw $t0, 4($0)        #incorrect if this instruction completes
```

```
Test5-2:
            li $t0, 2147450879
            add $t0, $t0, $t0
            lw $t0, 4($0)        #incorrect if this instruction completes

Test 5-3:
            lw $t0, 4($0)
            li $t0, -2147483648
            li $t1, 1
            sub $t0, $t0, $t1
            lw $t0, 4($0)

Test 5-4:
            li $t0, 2147483647
            mula $t0, $t0, $t0
            lw $t0, 4($0)
*/
      32'h300: Data = 32'h3c018000;
      32'h304: Data = 32'h34288000;
      32'h308: Data = 32'h01084020;
      32'h30C: Data = 32'h8c080004;

      32'h310: Data = 32'h3c017fff;
      32'h314: Data = 32'h34287fff;
      32'h318: Data = 32'h01084020;
      32'h31C: Data = 32'h8c080004;

      32'h320: Data = 32'h8c080004;
      32'h324: Data = 32'h3c088000;
      32'h328: Data = 32'h34090001;
      32'h32C: Data = 32'h01094022;
      32'h330: Data = 32'h8c080004;

      32'h334: Data = 32'h3c017FFF;
      32'h338: Data = 32'h3428FFFF;
      32'h33C: Data = 32'h01084038;
      32'h340: Data = 32'h8c080004;

/*
 * Overflow Exception
 */
/*
            lw $t0, 0($0)
*/
      32'hF0000000: Data = 32'h8c080000;

/*
 * Test Program 6
 * Test Branch Prediction performance
 */
                  /*
            li $t5, 0      # initialize data to 0
            li $t0, 100    # initialize exit value
            li $t1, 0      # initialize outer loop index to 0
      outer_loop:
            addi $t1, $t1, 1 #increment outer loop index
            li $t2, 0         #initialize inner loop index to 0
      inner_loop:
            addi $t2, $t2, 1 #increment inner loop index
            addi $t5, $t5, 1 #increment data
            bne $t2, $t0, inner_loop #go back to top of inner loop
            bne $t1, $t0, outer_loop #go back to top of outer loop
            sw $t5, 12($0) #store data into memory
            lw $t5, 12($0) #load data back out of memory
```

```
   */
32'h500: Data = 32'h240d0000;
32'h504: Data = 32'h24080064;
32'h508: Data = 32'h24090000;
32'h50C: Data = 32'h21290001;
32'h510: Data = 32'h240a0000;
32'h514: Data = 32'h214a0001;
32'h518: Data = 32'h21ad0001;
32'h51C: Data = 32'h1548fffd;
32'h520: Data = 32'h1528fffa;
32'h524: Data = 32'hac0d000c;
32'h528: Data = 32'h8c0d000c;


   /*
          * Test Program 7
          * Test Branch Prediction performance again
          */
          /*
                li $t5, 0       # initialize data to 0
                li $t0, 100     # initialize exit value
                li $t1, 0       # initialize outer loop index to 0
        outer_loop:
                addi $t1, $t1, 1 #increment outer loop index
                li $t2, 0         #initialize inner loop index to 0
        inner_loop:
                addi $t2, $t2, 1 #increment inner loop index
                andi $t3, $t2, 2 #mask inner loop index
                li $t4, 1         #set $t4 to 1
                beq $t3, $0, skip1
                li $t4, 0        #set $t4 to 0
        skip1:
                beq $t4, $0, skip2
                addi $t5, $t5, 1 #increment data
        skip2:
                beq $t2, $t1, exit_inner
                j inner_loop #go back to top of loop
        exit_inner:
                beq $t1, $t0, exit_outer
                j outer_loop
        exit_outer:
                sw $t5, 12($0) #store data into memory
                lw $t5, 12($0) #load data back out of memory
           */

32'h400: Data = 32'h240d0000;
32'h404: Data = 32'h24080064;
32'h408: Data = 32'h24090000;
32'h40C: Data = 32'h21290001;
32'h410: Data = 32'h240a0000;
32'h414: Data = 32'h214a0001;
32'h418: Data = 32'h314b0002;
32'h41C: Data = 32'h240c0001;
32'h420: Data = 32'h11600001;
32'h424: Data = 32'h240c0000;
32'h428: Data = 32'h11800001;
32'h42C: Data = 32'h21ad0001;
32'h430: Data = 32'h11490001;
32'h434: Data = 32'h08000105;
32'h438: Data = 32'h11280001;
32'h43C: Data = 32'h08000103;
32'h440: Data = 32'hac0d000c;
32'h444: Data = 32'h8c0d000c;
```

```
                default: Data = 32'hXXXXXXXX;
            endcase
        end
endmodule
```

Design Source code filename: DataMemory.v

```
`timescale 1ns / 1ps
module DataMemory( output reg [31:0] ReadData, input [5:0] Address, input [31:0]
WriteData, input MemoryRead, input MemoryWrite, input Clock );
// ReadData - Data output
// Addresss - Address bus for read/write
// WriteData - Data input
// MemoryRead - Read signal
// MemoryWrite - Write signal
// Clock - Clock signal
// Each word is 32 bits
// Hence, we need 64 words for 256 bytes storage
// 64 x 32 bits = 2048 bits = 256 bytes
reg [31:0] data_memory[63:0]; // 64 elements x 32 bit wide element
// Writes are synchronous on negative edge of clock
always @(negedge Clock) begin
        // Check if write signal is high
        if ( MemoryWrite == 1'b1 ) begin
                data_memory[Address] <= WriteData;
        end
end
// Reads are synchronous on positive edge of clock
always @(posedge Clock) begin
        // Check if read signal is high
        if ( MemoryRead == 1'b1 ) begin
                ReadData <= data_memory[Address];
        end
end
endmodule
```

Design Source code filename: ALU.v

```
`timescale 1ns / 1ps
// MACROs
`define AND 4'b0000
`define OR 4'b0001
`define ADD 4'b0010
`define SLL 4'b0011
`define SRL 4'b0100
`define SUB 4'b0110
`define SLT 4'b0111
`define ADDU 4'b1000
`define SUBU 4'b1001
`define XOR 4'b1010
`define SLTU 4'b1011
`define NOR 4'b1100
`define SRA 4'b1101
`define LUI 4'b1110
module ALU(BusW, Zero, BusA, BusB, ALUCtrl);
// Inputs
input wire [31:0] BusA, BusB;
input wire [3:0] ALUCtrl;
```

```
// Outputs
output reg [31:0] BusW;
output wire Zero;
wire less;
wire [63:0] Bus64;
// Zero signal is HIGH when BusW is zero
assign Zero = ( BusW == 32'b0 ? 32'b1 : 32'b0);

// less is HIGH when BusA < BusB (unsigned comparison)
assign less = ({1'b0,BusA} < {1'b0,BusB} ? 1'b1 : 1'b0);
assign Bus64 = 0;

// Switch case - performing arithmetic operations based on ALU Control Line
always@(*)begin
	case (ALUCtrl)
	`AND: BusW <= BusA & BusB;
	`OR: BusW <= BusA | BusB;
	`ADD: BusW <= BusA + BusB;
	`ADDU: BusW <= BusA + BusB; // Unsigned Addition
	`SLL: BusW <= BusB << BusA; // Shift Left Logical
	`SRL: BusW <= BusB >> BusA; // Shift Right Logical
	`SUB: BusW <= BusA - BusB;
	`SUBU: BusW <= BusA - BusB; // Unsigned Subtraction
	`XOR: BusW <= BusA ^ BusB;
	`NOR: BusW <= ~(BusA|BusB);
	// Set if less than (SLT)
	`SLT: BusW <= $signed(BusA) < $signed(BusB) ? 32'b1 : 32'b0;
	`SLTU: BusW <= less; // SLT unsigned
	`SRA: BusW <= $signed(BusB) >>> BusA; // Shift Right Arithmetic
	`LUI: BusW <= BusB << 16; // Load Upper Immediate
	default:BusW <= 32'bx;
	endcase
end
endmodule
```

(b) Use the test bench provided to test the functionality of your overall design.

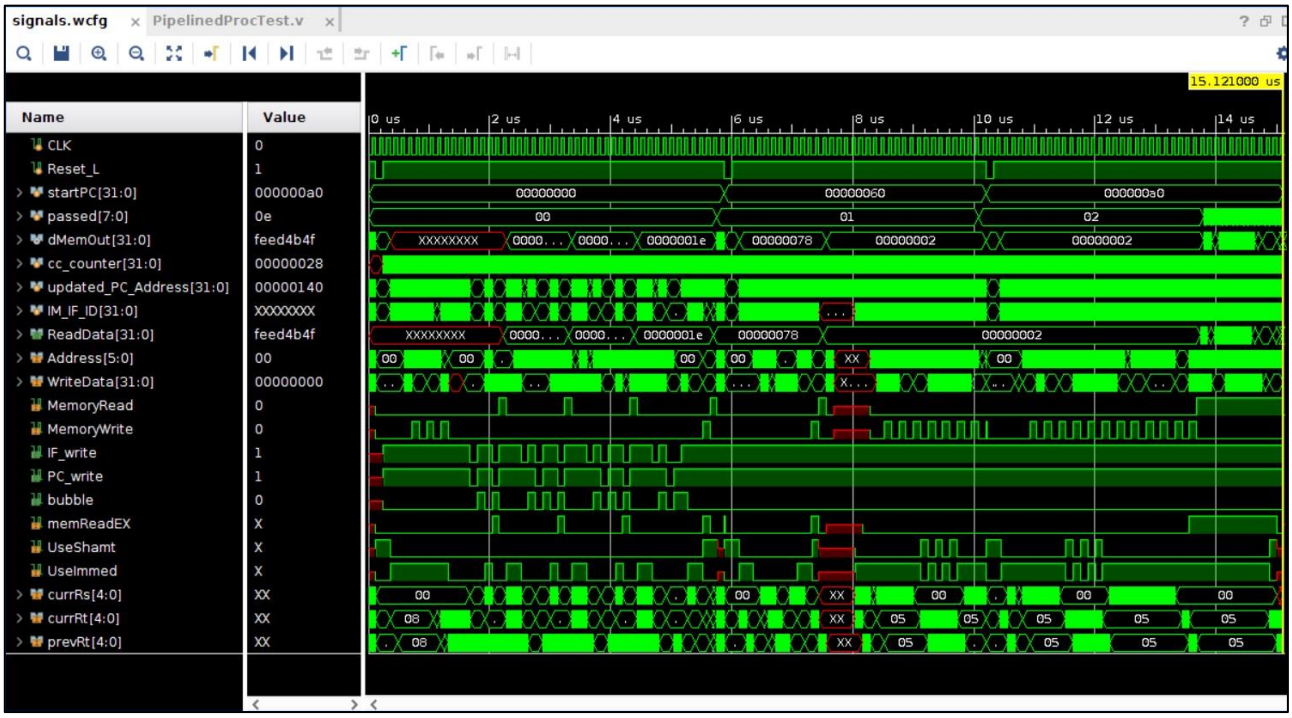
Testbench code filename: PipelinedProcTest.v

```
`timescale 1ns / 1ps

`define STRLEN 32
`define HalfClockPeriod 60
`define ClockPeriod `HalfClockPeriod * 2
module PipelinedProcTest_v;

	task passTest;
		input [31:0] actualOut, expectedOut;
		input [`STRLEN*8:0] testType;
		inout [7:0] passed;

		if(actualOut == expectedOut) begin $display ("%s passed", testType); passed =
passed + 1; end
		else $display ("%s failed: 0x%x should be 0x%x", testType, actualOut,
expectedOut);
	endtask

	task allPassed;
		input [7:0] passed;
		input [7:0] numTests;

		if(passed == numTests) $display ("All tests passed");
```

```
		else $display("Some tests failed: %d of %d passed", passed, numTests);
	endtask

	// Inputs
	reg CLK;
	reg Reset_L;
	reg [31:0] startPC;
	reg [7:0] passed;

	// Outputs
	wire [31:0] dMemOut;

	//book keeping
	reg [31:0] cc_counter;

	always@(negedge CLK)
		if(~Reset_L)
			cc_counter <= 0;
		else
			cc_counter <= cc_counter+1;
	initial begin
		CLK= 1'b0;
	end

	 /*generate clock signal*/
always begin
	#`HalfClockPeriod CLK = ~CLK;
	#`HalfClockPeriod CLK = ~CLK;
end

	// Instantiate the Unit Under Test (UUT)
	PipelinedProc uut (
		.CLK(CLK),
		.Reset_L(Reset_L),
		.startPC(startPC),
		.dMemOut(dMemOut)
	);

	initial begin
		// Initialize Inputs
		Reset_L = 1;
		startPC = 0;
		passed = 0;

		// Wait for global reset
		#(1 * `ClockPeriod);

		// Program 1
		#1;
		Reset_L = 0; startPC = 32'h0;
		#(1 * `ClockPeriod);
		Reset_L = 1;
		#(46 * `ClockPeriod);
		passTest(dMemOut, 120, "Results of Program 1", passed);

		// Program 2
		#(1 * `ClockPeriod);
		Reset_L = 0; startPC = 32'h60;
		#(1 * `ClockPeriod);
		Reset_L = 1;
		#(34 * `ClockPeriod);
		passTest(dMemOut, 2, "Results of Program 2", passed);

		// Program 3
```

```
            #(1 * `ClockPeriod);
            Reset_L = 0; startPC = 32'hA0;
            #(1 * `ClockPeriod);
            Reset_L = 1;
            #(29 * `ClockPeriod);
            passTest(dMemOut, 32'hfeedbeef, "Result 1 of Program 3", passed);
            #(1 * `ClockPeriod);
            passTest(dMemOut, 32'hfeedb48f, "Result 2 of Program 3", passed);
            #(1 * `ClockPeriod);
            passTest(dMemOut, 32'hfeeeb48f, "Result 3 of Program 3", passed);
            #(1 * `ClockPeriod);
            passTest(dMemOut, 32'h0000b4a0, "Result 4 of Program 3", passed);
            #(1 * `ClockPeriod);
            passTest(dMemOut, 32'hddb7dde0, "Result 5 of Program 3", passed);
            #(1 * `ClockPeriod);
            passTest(dMemOut, 32'h07f76df7, "Result 6 of Program 3", passed);
            #(1 * `ClockPeriod);
            passTest(dMemOut, 32'hfff76df7, "Result 7 of Program 3", passed);
            #(1 * `ClockPeriod);
            passTest(dMemOut, 1, "Result 8 of Program 3", passed);
            #(1 * `ClockPeriod);
            passTest(dMemOut, 0, "Result 9 of Program 3", passed);
            #(1 * `ClockPeriod);
            passTest(dMemOut, 0, "Result 10 of Program 3", passed);
            #(1 * `ClockPeriod);
            passTest(dMemOut, 1, "Result 11 of Program 3", passed);
            #(1 * `ClockPeriod);
            passTest(dMemOut, 32'hfeed4b4f, "Result 12 of Program 3", passed);

            // Done
            allPassed(passed, 14);
            $finish;
        end
endmodule
```

Simulation Output Waveform: (Pipelined MIPS Processor)



Simulation Output Logs: (Pipelined MIPS Processor)

```
INFO: [Simtcl 6-17] Simulation restarted
run all
             Results of Program 1 passed
             Results of Program 2 passed
            Result 1 of Program 3 passed
            Result 2 of Program 3 passed
            Result 3 of Program 3 passed
            Result 4 of Program 3 passed
            Result 5 of Program 3 passed
            Result 6 of Program 3 passed
            Result 7 of Program 3 passed
            Result 8 of Program 3 passed
            Result 9 of Program 3 passed
           Result 10 of Program 3 passed
           Result 11 of Program 3 passed
           Result 12 of Program 3 passed
All tests passed
$finish called at time : 15121 ns : File "/home/grads/p/pranav_anantharam/lab6/PipelinedProcTest.v" Line 119
```

(c) Synthesize the hardware and ensure the code generates no warnings or errors. Provide the summary results of the synthesis process.

Synthesis Report: (Pipelined MIPS Processor)

```
---------------------------------------------------------------------------------
Start Writing Synthesis Report
---------------------------------------------------------------------------------

Report BlackBoxes:
+-+--------------+----------+
| |BlackBox name |Instances |
+-+--------------+----------+
+-+--------------+----------+
Report Cell Usage:
+------+---------+------+
|      |Cell     |Count |
+------+---------+------+
|1     |BUFG     |     1|
|2     |CARRY4   |    38|
|3     |LUT1     |     1|
|4     |LUT2     |   169|
|5     |LUT3     |   132|
|6     |LUT4     |   207|
|7     |LUT5     |   328|
|8     |LUT6     |   450|
|9     |MUXF7    |    38|
|10    |MUXF8    |     4|
|11    |RAM32M   |    12|
|12    |RAMB18E1 |     1|
|13    |FDCE     |   331|
|14    |FDC_1    |    32|
|15    |FDPE     |    28|
|16    |FDRE     |     2|
|17    |LDC      |    29|
|18    |IBUF     |    30|
|19    |OBUF     |    32|
+------+---------+------+
Report Instance Areas:
+------+---------+------------+------+
|      |Instance |Module      |Cells |
+------+---------+------------+------+
|1     |top      |            |  1865|
```

```
|2     |  RF1    |RegisterFile |    13|
|3     |  ALU1   |ALU          |   205|
|4     |  DM1    |DataMemory   |    33|
|5     |  Hazard |HazardUnit   |   717|
+------+---------+-------------+------+
```

```
---------------------------------------------------------------------------------
Finished Writing Synthesis Report : Time (s): cpu = 00:00:26 ; elapsed = 00:00:33 . Memory
(MB): peak = 1879.168 ; gain = 387.602 ; free physical = 9253 ; free virtual = 44685
---------------------------------------------------------------------------------
```

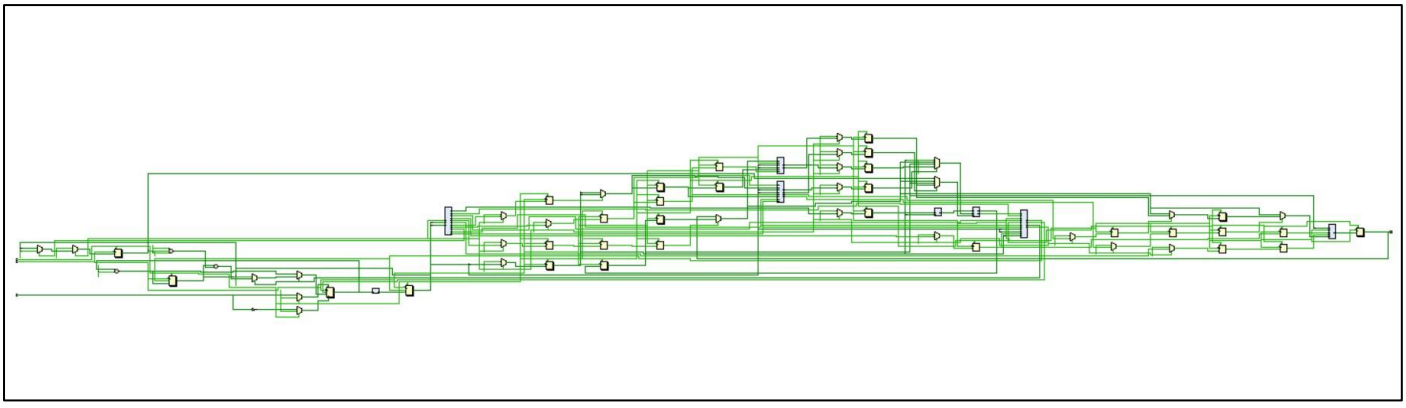
Synthesis finished with 0 errors, 0 critical warnings and 3 warnings.

```
Synthesis Optimization Runtime : Time (s): cpu = 00:00:26 ; elapsed = 00:00:33 . Memory (MB):
peak = 1879.168 ; gain = 387.602 ; free physical = 9255 ; free virtual = 44688
Synthesis Optimization Complete : Time (s): cpu = 00:00:26 ; elapsed = 00:00:33 . Memory (MB):
peak = 1879.172 ; gain = 387.602 ; free physical = 9264 ; free virtual = 44697
INFO: [Project 1-571] Translating synthesized netlist
INFO: [Netlist 29-17] Analyzing 154 Unisim elements for replacement
INFO: [Netlist 29-28] Unisim Transformation completed in 0 CPU seconds
INFO: [Project 1-570] Preparing netlist for logic optimization
INFO: [Opt 31-138] Pushed 0 inverter(s) to 0 load pin(s).
Netlist sorting complete. Time (s): cpu = 00:00:00.01 ; elapsed = 00:00:00 . Memory (MB): peak
= 1935.195 ; gain = 0.000 ; free physical = 9075 ; free virtual = 44519
INFO: [Project 1-111] Unisim Transformation Summary:
  A total of 73 instances were transformed.
  FDC_1 => FDCE (inverted pins: C): 32 instances
  LDC => LDCE: 29 instances
RAM32M => RAM32M (RAMD32, RAMD32, RAMD32, RAMD32, RAMD32, RAMD32, RAMS32, RAMS32): 12 instances

INFO: [Common 17-83] Releasing license: Synthesis
```

37 Infos, 3 Warnings, 0 Critical Warnings and 0 Errors encountered.

```
synth_design completed successfully
synth_design: Time (s): cpu = 00:00:31 ; elapsed = 00:00:38 . Memory (MB): peak = 1935.195 ;
gain = 472.559 ; free physical = 9132 ; free virtual = 44575
Netlist sorting complete. Time (s): cpu = 00:00:00 ; elapsed = 00:00:00 . Memory (MB): peak =
1935.195 ; gain = 0.000 ; free physical = 9132 ; free virtual = 44575
WARNING: [Constraints 18-5210] No constraints selected for write.
Resolution: This message can indicate that there are no constraints for the design, or it can
indicate that the used_in flags are set such that the constraints are ignored. This later case
is used when running synth_design to not write synthesis constraints to the resulting
checkpoint. Instead, project constraints are read when the synthesized design is opened.
INFO: [Common 17-1381] The checkpoint
'/home/grads/p/pranav_anantharam/lab6/project_1/project_1.runs/synth_1/PipelinedProc.dcp' has
been generated.
INFO: [runtcl-4] Executing : report_utilization -file PipelinedProc_utilization_synth.rpt -pb
PipelinedProc_utilization_synth.pb
INFO: [Common 17-206] Exiting Vivado at Sun Dec  3 17:38:45 2023...
```

Pipelined MIPS Processor – Elaborated Design:



5. Answer the following review questions:

a) Instead of stalling the pipeline while waiting for a branch to be evaluated, we could simply continue to fetch from PC+4. What name is commonly given to this technique? What changes would have to be made to our existing design to accommodate this improvement? Be sure to include modifications to the Hazard Detection Unit and pipeline registers.

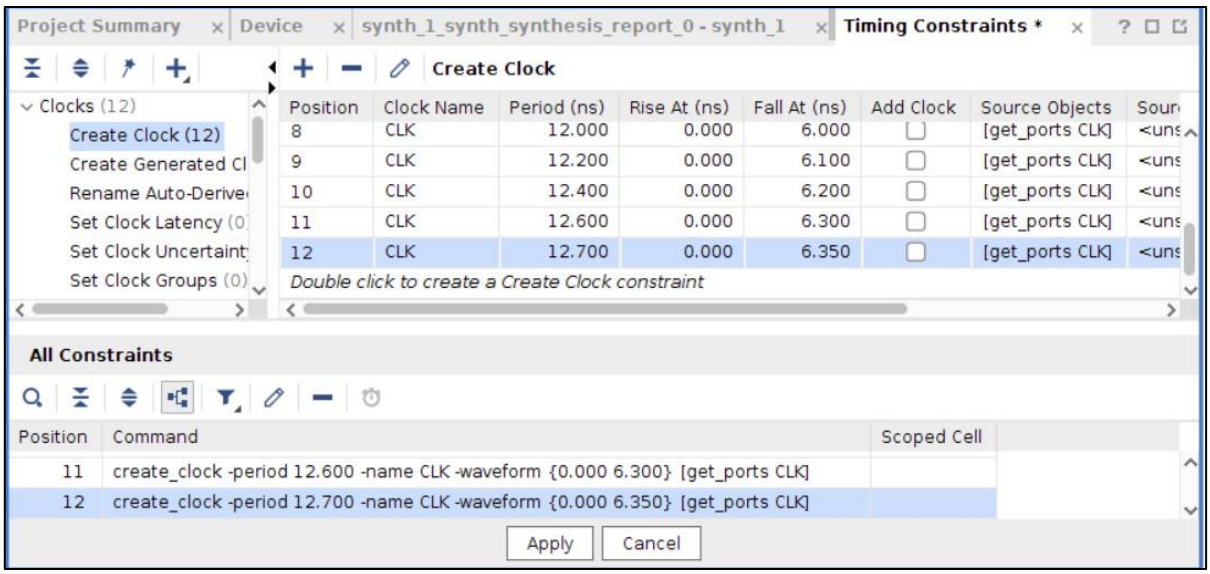
Answer:

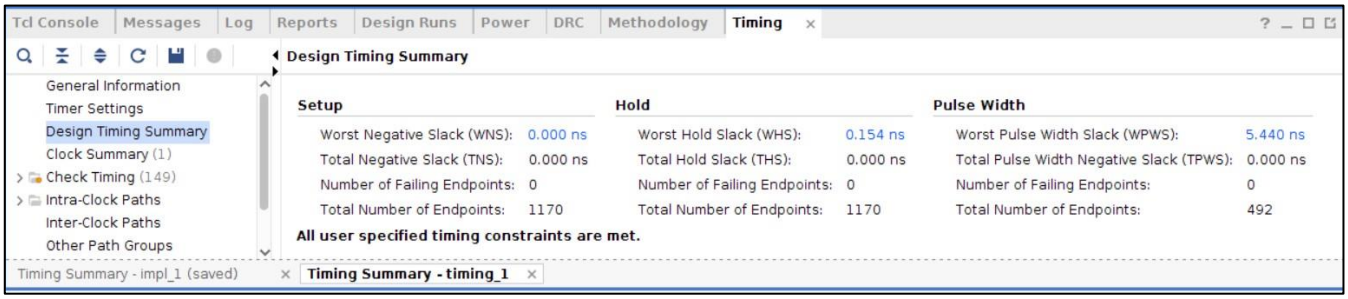
The name given to this technique is static branch prediction, in particular, it is the 'branch always not taken' prediction. To accommodate this improvement in our existing design, we need to make the following changes:

(i) When a branch is encountered, do not add a bubble in the pipeline.
(ii) Keep IFwrite and PCwrite signals HIGH to ensure that instruction at PC+4 address is fetched.
(iii) After resolving the branch, if the branch is not taken, then no operation needs to be performed.
(iv) However, if the branch is taken, the pipeline must be flushed and the correct instruction at the branch target address must be fetched at the start of the next cycle and resume normal execution.

b) What clock rate did the synthesis process estimate your overall design would run at? Look through the synthesis report and locate the section where the design timing is estimated. From that, determine which components are in the critical path. What changes could you make to the design to improve the clock rate without adding additional pipeline stages?

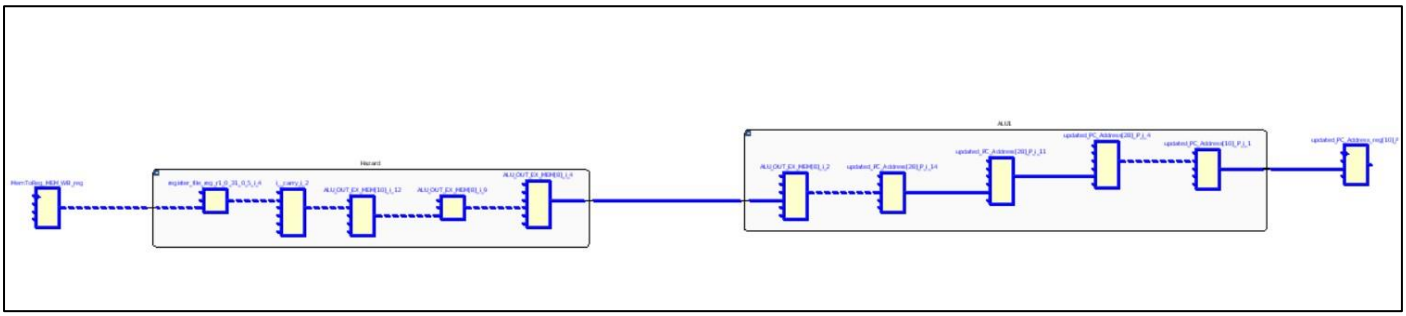
Answer:





According to the design timing report, the clock period is 12.7 ns, where the worst slack is minimum. The frequency is calculated to be **78.740 MHz.**

Critical path of design:



The above path is used by instructions like Load/Store instructions which requires and executes all stages of the MIPS pipeline.

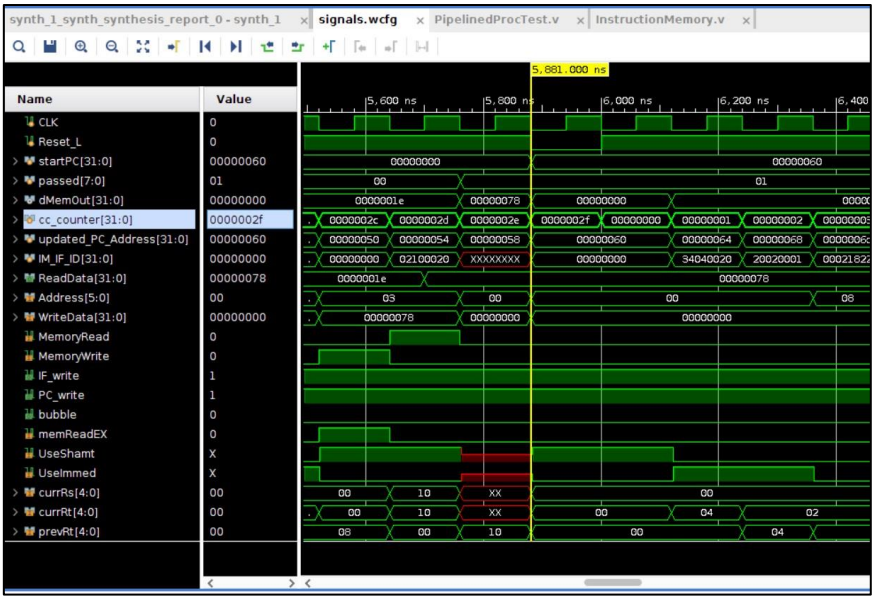
To improve the clock rate without adding additional pipeline stages, we can enhance and speed up the hardware and decrease the delay in the critical path. Furthermore, instructions can be executed out-of-order. And so, efficiency can be increased by making use of the stall cycles in the pipeline. We can use techniques like loop unrolling and scheduling to further speed up the processor.

c) Determine the CPI for each of the three programs executing on your processor. Compare those results to the CPI of the single-cycle processor. Also, provide the average CPI of all three programs.

Answer:

Program 1 Instruction Memory:

```
        add $t1, $t1, 4          # Next address
        add $t2, $t2, 1          # Next index
        j P1Loop                        # Jump to loop
P1Done: sw $t0, 0($t1)           # Store the sum at end of array
        lw $t0, 12($0)              # Load Final Value
        nop                         # Complete
        add $0, $s0, $s0         # do nothing
*/
    32'h00: Data = 32'h34080032;
    32'h04: Data = 32'hac080000;
    32'h08: Data = 32'h34080028;
    32'h0C: Data = 32'hac080004;
    32'h10: Data = 32'h3408001e;
    32'h14: Data = 32'hac080008;
    32'h18: Data = 32'h34040000;
    32'h1C: Data = 32'h34050003;
    32'h20: Data = 32'h00004020;
    32'h24: Data = 32'h00044820;
    32'h28: Data = 32'h00005020;
    32'h2C: Data = 32'h11450005;
    32'h30: Data = 32'h8d2b0000;
    32'h34: Data = 32'h010b4020;
    32'h38: Data = 32'h21290004;
    32'h3C: Data = 32'h214a0001;
    32'h40: Data = 32'h0800000b;
    32'h44: Data = 32'had280000;
    32'h48: Data = 32'h8c08000c;
    32'h4C: Data = 32'h00000000;
    32'h50: Data = 32'h02100020;
```

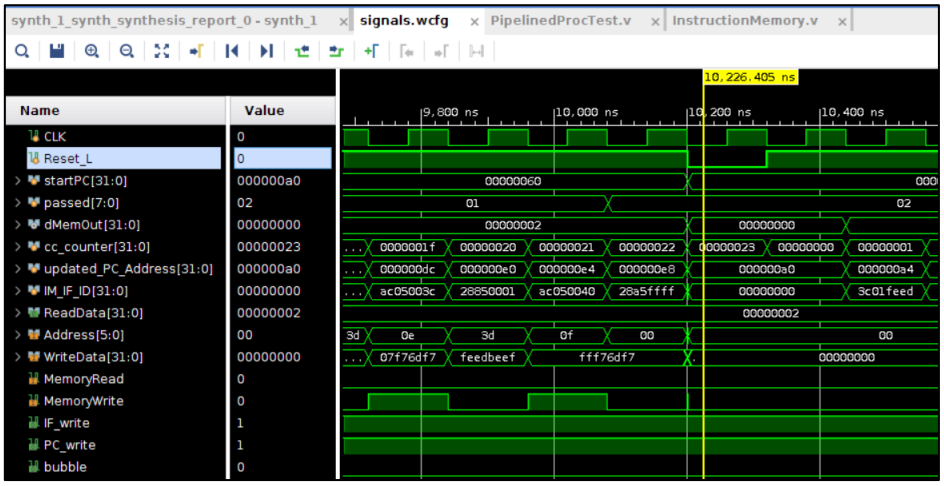


Program 1 contains 21 instructions. It requires 46 (0x2E) cycles to execute as shown in the above waveform.

CPI = 46 / 21 = 2.1904

Program 2 Instruction Memory:

```
32'h60: Data = 32'h34040020;
32'h64: Data = 32'h20020001;
32'h68: Data = 32'h00021822;
32'h6C: Data = 32'h0060282a;
32'h70: Data = 32'h00453020;
32'h74: Data = 32'h00a63825;
32'h78: Data = 32'h00a74022;
32'h7C: Data = 32'h01074824;
32'h80: Data = 32'hac890000;
32'h84: Data = 32'h8c090020;
32'h88: Data = 32'h00000000;
```

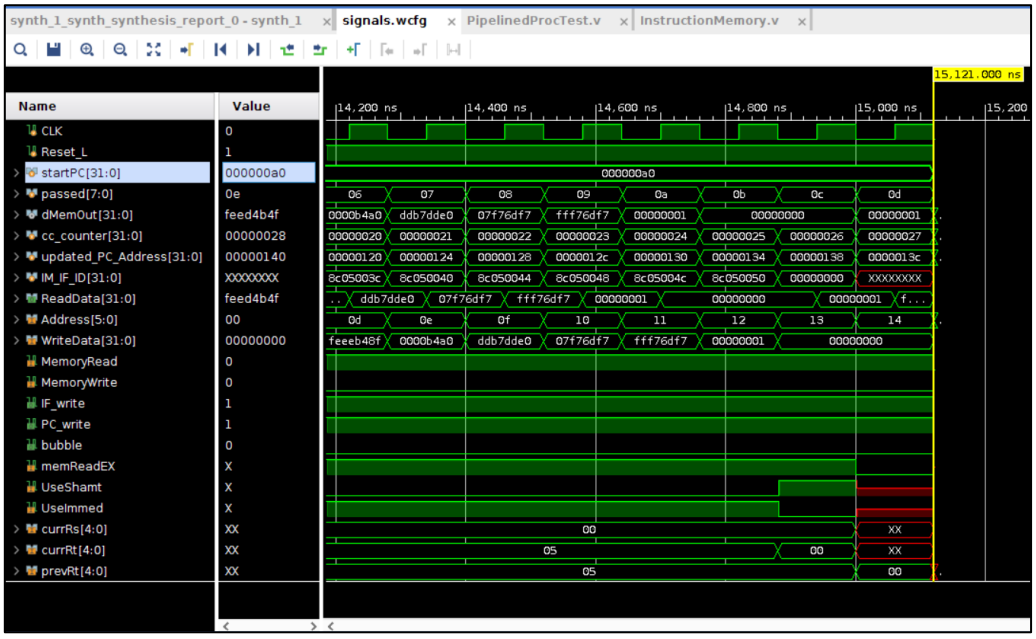


Program 2 contains 11 instructions. It requires 34 (0x22) cycles to execute as shown in the above waveform.

CPI = 34 / 11 = 3.090

Program 3 Instruction Memory:

```
32'hA0: Data = 32'h3c01feed;
32'hA4: Data = 32'h3424beef;
32'hA8: Data = 32'hac040024;
32'hAC: Data = 32'h2085f5a0;
32'hB0: Data = 32'hac050028;
32'hB4: Data = 32'h2485f5a0;
32'hB8: Data = 32'hac05002c;
32'hBC: Data = 32'h3085f5a0;
32'hC0: Data = 32'hac050030;
32'hC4: Data = 32'h00042940;
32'hC8: Data = 32'hac050034;
32'hCC: Data = 32'h00042942;
32'hD0: Data = 32'hac050038;
32'hD4: Data = 32'h00042943;
32'hD8: Data = 32'hac05003c;
32'hDC: Data = 32'h28850001;
32'hE0: Data = 32'hac050040;
32'hE4: Data = 32'h28a5ffff;
32'hE8: Data = 32'hac050044;
32'hEC: Data = 32'h2c850001;
32'hF0: Data = 32'hac050048;
32'hF4: Data = 32'h2ca5ffff;
32'hF8: Data = 32'hac05004c;
32'hFC: Data = 32'h3885f5a0;
32'h100: Data = 32'hac050050;
32'h104: Data = 32'h8c040024;
32'h108: Data = 32'h8c050028;
32'h10C: Data = 32'h8c05002c;
32'h110: Data = 32'h8c050030;
32'h114: Data = 32'h8c050034;
32'h118: Data = 32'h8c050038;
32'h11C: Data = 32'h8c05003c;
32'h120: Data = 32'h8c050040;
32'h124: Data = 32'h8c050044;
32'h128: Data = 32'h8c050048;
32'h12C: Data = 32'h8c05004c;
```



Program 3 contains 38 instructions. It requires 40 (0x28) cycles to execute as shown in the above waveform.

CPI = 40 / 38 = 1.05263

Average CPI of the 3 programs in pipelined MIPS = (1.05263 + 3.090 + 2.1904) / 3 = 2.11082

For a single cycle processor, pipelining is not performed and so the CPI will be equivalent to the pipelined depth, in this case, it is 5. Hence, MIPS single cycled CPI = 5.

Speed up of pipelined MIPS = single cycle MIPS CPI / pipelined MIPS CPI = 5 / 2.11082 = 2.36874

Speed up of pipelined MIPS = 2.36874

d) Given the clock rate and average CPI, compute the MIPS (Million Instructions Per Second) rating for both versions of processor. Which one is faster and why?

Answer:

Single Cycle MIPS
Clock rate = 36.735MHz
CPI = 5 cycles/instruction
MIPS = (Clock rate/CPI)/10^6
MIPS = (36.735/5)
MIPS = 7.347

Pipelined MIPS
Clock rate = 78.740 MHz
CPI = 2.11082 cycles/instruction
MIPS = (Clock rate/CPI)/10^6
MIPS = (78.740/2.11082)
MIPS = 37.3030

Hence, we can conclude that Pipelined MIPS can process more number of instructions per second than single cycle MIPS.