ECEN 651 Lab Exercise 4 Report

Single-Cycle MIPS Processor

Name: Pranav Anantharam

UIN: 734003886

Date: 10/29/2023

1. Design the control unit logic.

Block Diagram:

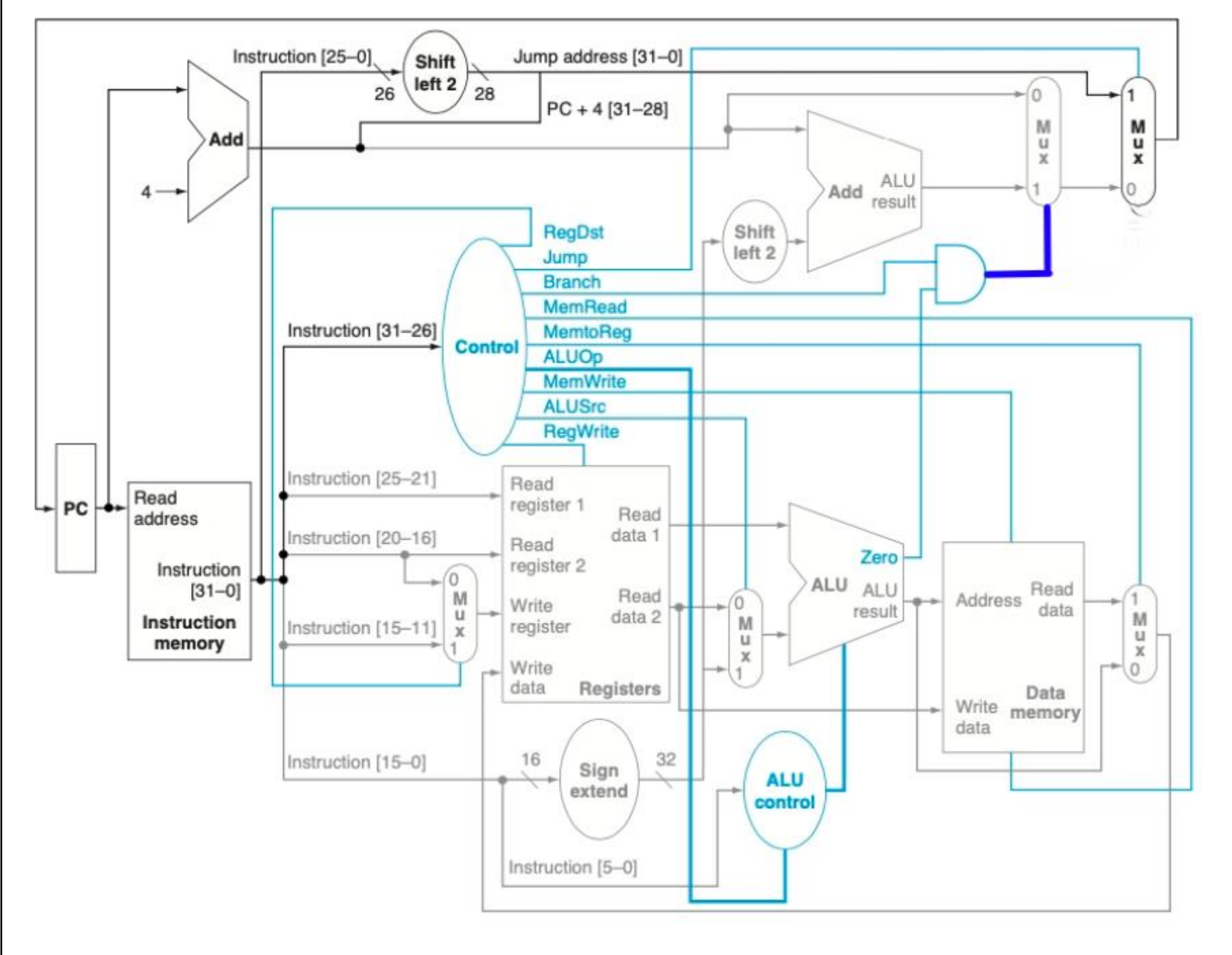


Figure 1: MIPS Block Diagram

Block Diagram:

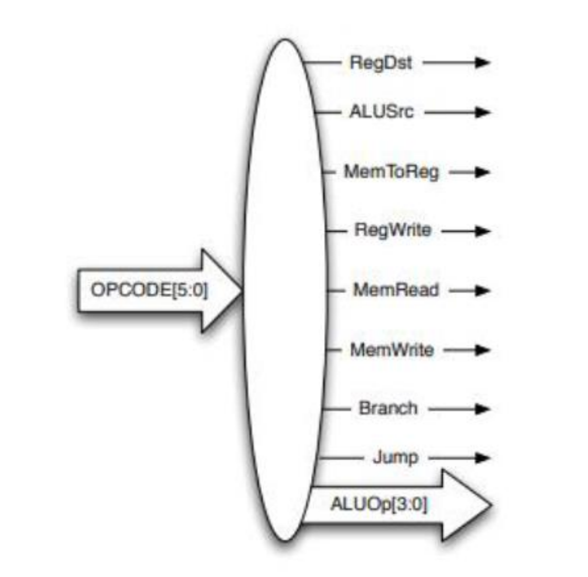
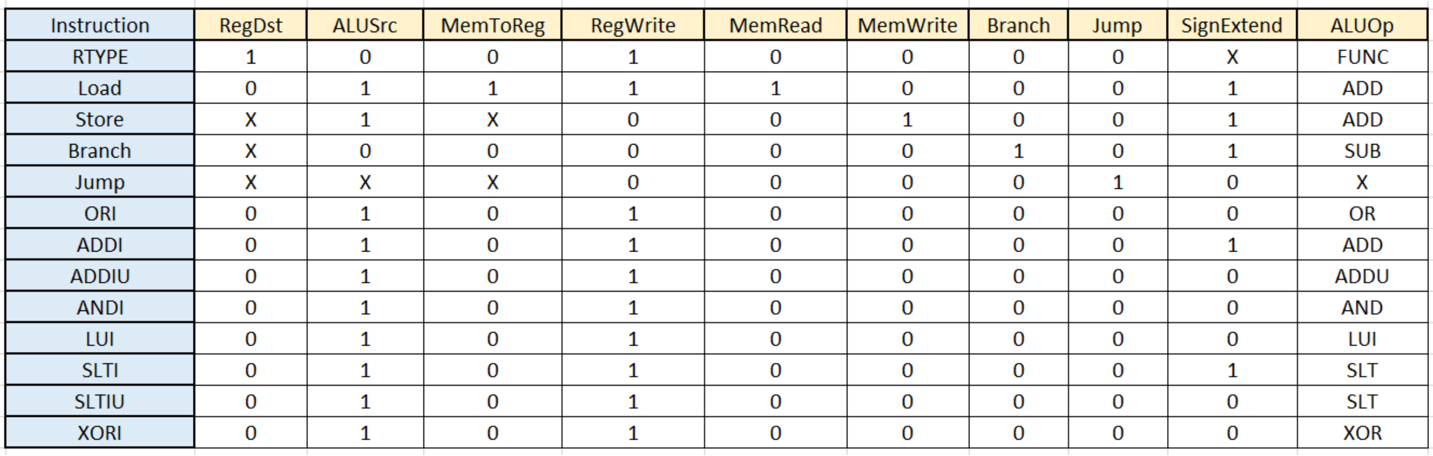


Figure 2: Control Unit Decoder

(1) Create a table which lists the values of all control signals for the following instructions: R-type, load/store word, immediate, branch and jump.



Instruction	RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	Jump	SignExtend	ALUOp
RTYPE	1	0	0	1	0	0	0	0	X	FUNC
Load	0	1	1	1	1	0	0	0	1	ADD
Store	X	1	X	0	0	1	0	0	1	ADD
Branch	X	0	0	0	0	0	1	0	1	SUB
Jump	X	X	X	0	0	0	0	1	0	X
ORI	0	1	0	1	0	0	0	0	0	OR
ADDI	0	1	0	1	0	0	0	0	1	ADD
ADDIU	0	1	0	1	0	0	0	0	0	ADDU
ANDI	0	1	0	1	0	0	0	0	0	AND
LUI	0	1	0	1	0	0	0	0	0	LUI
SLTI	0	1	0	1	0	0	0	0	1	SLT
SLTIU	0	1	0	1	0	0	0	0	0	SLT
XORI	0	1	0	1	0	0	0	0	0	XOR

(2) Write the Verilog to describe the designed control logic.

Design Source code filename: SingleCycleControl.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    12:23:34 03/10/2009
// Design Name:
// Module Name:    SingleCycleControl
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

`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 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 SingleCycleControl(RegDst, ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch,
Jump, SignExtend, ALUOp, Opcode);


      // Inputs

      input [5:0] Opcode;

      // Outputs

      output RegDst;         // For dest register : 0-> 20-16 or  1-> 15-11

      output ALUSrc;         // 0 - Logical operations - register values, 1 - Immediate /
Arithmetic values

      output MemToReg;       // 0 -> ALU output written back to register, 1-> data memory
written to register eg. Load instruction

      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


      reg RegDst, ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend;

      reg  [3:0] ALUOp;

      always @ (Opcode) begin

            case(Opcode)

                  `RTYPEOPCODE: begin // R-type instructions: RegDst and Regwrite is 1

                        RegDst <= #2 1'b1; // dest register address = Instruction[15:11]

                        ALUSrc <= #2 1'b0;

                        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;

                  end
                  `LWOPCODE: begin
                        RegDst <= #2 1'b0; // dest register address = Instruction[20:16]
                        ALUSrc <= #2 1'b1; // effective address calculation using offset
                        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
                  end

                  `SWOPCODE: begin
                        RegDst <= #2 1'bx;       // No register writes
                        ALUSrc <= #2 1'b1; // effective address calculation using offset
                        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
                  end
                  `BEQOPCODE: begin
                        RegDst <= #2 1'bx;       // No register writes
                        ALUSrc <= #2 1'b0;       // Operands are read from registers
                        MemToReg <= #2 1'b0;// 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
                  end
```

```
`JOPCODE: begin
      RegDst <= #2 1'bx;       // No register writes
      ALUSrc <= #2 1'bx;       // No ALU operations
      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'b0;   // Sign extension not used for JUMP
      ALUOp <= #2 4'bxxxx;
end

`ORIOPCODE: begin
      RegDst <= #2 1'b0; // dest register address = Instruction[20:16]
      ALUSrc <= #2 1'b1;       // Immediate value
      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
end

`ADDIOPCODE: begin
      RegDst <= #2 1'b0; // dest register address = Instruction[20:16]
      ALUSrc <= #2 1'b1;       // Immediate value
      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
end
```

```
`ADDIUOPCODE: begin
      RegDst <= #2 1'b0; // dest register address = Instruction[20:16]
      ALUSrc <= #2 1'b1;       // Immediate value
      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
end

`ANDIOPCODE: begin
      RegDst <= #2 1'b0; // dest register address = Instruction[20:16]
      ALUSrc <= #2 1'b1;       // Immediate value
      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
end

`LUIOPCODE: begin
      RegDst <= #2 1'b0; // dest register address = Instruction[20:16]
      ALUSrc <= #2 1'b1;       // Immediate value
      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 `LUI;        // LUI operation
end
```

```
`SLTIOPCODE: begin
      RegDst <= #2 1'b0; // dest register address = Instruction[20:16]
      ALUSrc <= #2 1'b1;       // Immediate value
      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
end

`SLTIUOPCODE: begin
      RegDst <= #2 1'b0; // dest register address = Instruction[20:16]
      ALUSrc <= #2 1'b1;       // Immediate value
      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
end

`XORIOPCODE: begin
      RegDst <= #2 1'b0; // dest register address = Instruction[20:16]
      ALUSrc <= #2 1'b1;       // Immediate value
      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
end
```

```
                        default: begin
                              RegDst <= #2 1'bx;
                              ALUSrc <= #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;
                        end
                  endcase
            end
endmodule
```

(3) 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: (SingleCycleControl module)

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

Report BlackBoxes:
+-+--------------+----------+
| |BlackBox name |Instances |
+-+--------------+----------+
+-+--------------+----------+

Report Cell Usage:
+------+-----+------+
|      |Cell |Count |
+------+-----+------+
|1     |LUT2 |     5|
|2     |LUT3 |     1|
|3     |LUT4 |     4|
|4     |LUT5 |     2|
|5     |IBUF |     5|

|6     |OBUF |    13|

+------+-----+------+


Report Instance Areas:

+------+---------+-------+------+

|      |Instance |Module |Cells |

+------+---------+-------+------+

|1     |top      |       |    30|

+------+---------+-------+------+

---------------------------------------------------------------------------------

Finished Writing Synthesis Report : Time (s): cpu = 00:00:13 ; elapsed = 00:00:20 . Memory
(MB): peak = 1760.898 ; gain = 269.336 ; free physical = 135645 ; free virtual = 154540

---------------------------------------------------------------------------------
```

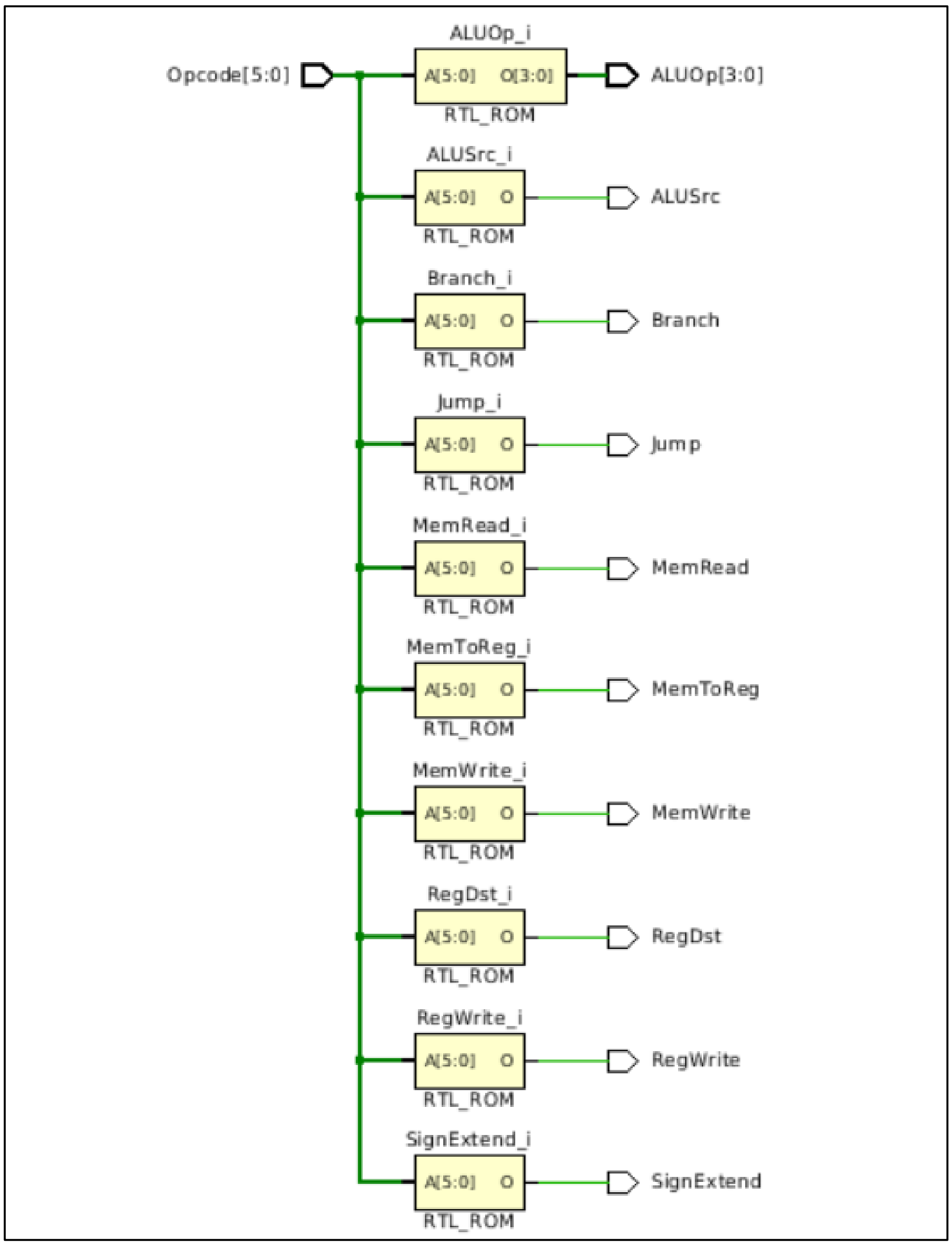
Synthesis finished with 0 errors, 0 critical warnings and 0 warnings.

```
Synthesis Optimization Runtime : Time (s): cpu = 00:00:13 ; elapsed = 00:00:20 . Memory (MB):
peak = 1760.898 ; gain = 269.336 ; free physical = 135646 ; free virtual = 154541

Synthesis Optimization Complete : Time (s): cpu = 00:00:13 ; elapsed = 00:00:20 . Memory (MB):
peak = 1760.902 ; gain = 269.336 ; free physical = 135651 ; free virtual = 154547

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 =
1908.945 ; gain = 0.000 ; free physical = 135549 ; free virtual = 154445

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 = 00:00:25 . Memory (MB): peak = 1908.945 ;
gain = 446.312 ; free physical = 135605 ; free virtual = 154501

Netlist sorting complete. Time (s): cpu = 00:00:00 ; elapsed = 00:00:00 . Memory (MB): peak =
1908.945 ; gain = 0.000 ; free physical = 135605 ; free virtual = 154501

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/lab5_final/project_1/project_1.runs/synth_1/SingleCycleControl
.dcp' has been generated.

INFO: [runtcl-4] Executing : report_utilization -file SingleCycleControl_utilization_synth.rpt
-pb SingleCycleControl_utilization_synth.pb

INFO: [Common 17-206] Exiting Vivado at Sun Oct 29 16:58:40 2023...
```

Compiled Hardware Schematic: (SingleCycleControl module)



No latches present in the single cycle control unit.

2. Design the remainder of the MIPS single cycle datapath.

(1) Describe the single-cycle MIPS datapath in Verilog.

Design Source code filename: SingleCycleProc.v

```
`timescale 1ns / 1ps
// Program Counter Module
module PC( input clk, input reset, input [31:0 ] startPC, input [31:0] updated_address,
output reg [31:0] address );

      // PC is sensitive to negative edge of clock
      // Reset (active low) should asynchronously reset the PC
      always @(negedge clk or negedge reset) begin
            if ( reset == 1'b0 )
                  address <= startPC;           // Reset PC to starting address
            else
                  address <= updated_address;   // Set PC to updated address
      end
endmodule

// Single Cycle Processor Module
module SingleCycleProc( CLK, Reset_L, startPC, dMemOut);

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

// Instruction Memory
wire [31:0] Data;
wire [31:0] Address;
wire [31:0] New_PC_addr;

// Wires - Register File
wire [31:0] BusA_wire, BusB_wire;
```

```verilog
wire[4:0] write_reg_select_wire;
wire [31:0] write_reg_data_wire;

// Wire - Data Memory output
wire [31:0] Data_mem_out_wire;

// Wires - Control Unit
wire RegDst_wire;
wire ALUSrc_wire;
wire MemToReg_wire;
wire RegWrite_wire;
wire MemRead_wire;
wire MemWrite_write;
wire Branch_wire;
wire Jump_wire;
wire SignExtend_wire;
wire [3:0] ALUOp_wire;

// Wire - ALU Control
wire [3:0] ALU_control_wire;

// Wires - MUX before ALU
wire [31:0] operand_2_ALU_wire;
wire[31:0] sign_ext_data_wire;

// Wires - ALU output
wire [31:0] ALU_Output_wire;
wire ALU_Zero_wire;

// Wire - shift left 2 bits for branch
wire [31:0] SL_2_branch_wire;

// Wire - branch address
wire [31:0] branch_addr_wire;

// Wire - PC update
wire[31:0] PC_update_wire;
```

```
// Wire - Branch AND output
wire [31:0] branch_AND_wire;


// Wire - Branch MUX output
wire [31:0] branch_MUX_wire;


// Wire - Shift left 2 bits 25 to 0
wire [27:0] SL_2_jump_wire;


// Wire - Final jump
wire [31:0] jump_addr_wire;


// Wire - Register File Final outputs
wire [31:0] Bus_A_output;
wire [31:0] Bus_B_output;


// *** Instantiating modules ***
// Instruction Memory
InstructionMemory IM_1(.Data(Data), .Address(Address));


// Program Counter
PC PC_1( .clk(CLK) , .reset(Reset_L) , .startPC(startPC), .updated_address(New_PC_addr),
.address(Address));


// Single Cycle Control
SingleCycleControl SCC_1 (.RegDst(RegDst_wire), .ALUSrc(ALUSrc_wire),
.MemToReg(MemToReg_wire), .RegWrite(RegWrite_wire), .MemRead(MemRead_wire),
.MemWrite(MemWrite_write), .Branch(Branch_wire), .Jump(Jump_wire),
.SignExtend(SignExtend_wire), .ALUOp(ALUOp_wire), .Opcode(Data[31:26]));


// Register File
RegisterFile RF_1 ( .RA(Data[25:21]), .RB(Data[20:16]), .RW(write_reg_select_wire),
.BusW(write_reg_data_wire), .RegWr(RegWrite_wire), .Clk(CLK), .BusA(BusA_wire),
.BusB(BusB_wire) );


// ALU Control Unit
ALUControl AC_1 (.ALUCtrl(ALU_control_wire), .ALUop(ALUOp_wire), .FuncCode(Data[5:0]));
```

```
// ALU

ALU ALU_1 (.BusW(ALU_Output_wire), .Zero(ALU_Zero_wire), .BusA(Bus_A_output),
.BusB(Bus_B_output), .ALUCtrl(ALU_control_wire));


// Data Memory - Word addressable - skip first 2 bits

DataMemory DM_1 (.ReadData(Data_mem_out_wire), .Address(ALU_Output_wire[7:2]),
.WriteData(BusB_wire), .MemoryRead(MemRead_wire), .MemoryWrite(MemWrite_write),
.Clock(CLK));


// Sign extension

assign sign_ext_data_wire = SignExtend_wire ? {{16{Data[15]}},Data[15:0]} :
{{16{1'b0}},Data[15:0]}  ;


// Selecting write register of RF_1

assign write_reg_select_wire = RegDst_wire ? Data[15:11] : Data[20:16];


// Selecting 2nd operand of ALU

assign operand_2_ALU_wire = ALUSrc_wire ? sign_ext_data_wire : BusB_wire;


assign Bus_A_output = BusA_wire;

assign Bus_B_output = operand_2_ALU_wire;


// Data_written in register

assign write_reg_data_wire = MemToReg_wire ? Data_mem_out_wire : ALU_Output_wire;


// Shift left for branch

assign SL_2_branch_wire = sign_ext_data_wire << 2;


// Updating PC -> PC = PC + 4

assign PC_update_wire =  Address + 32'd4 ;


// Branch address

assign branch_addr_wire = SL_2_branch_wire + PC_update_wire;


// Compute Branch AND output

assign branch_AND_wire = Branch_wire & ALU_Zero_wire;
```

```
// Compute Branch MUX output

assign branch_MUX_wire = branch_AND_wire ?  branch_addr_wire : PC_update_wire;


// Left shift 26 bits of data by 2 bits for jump instruction - no. of instructions x 4 ->
No. of bytes / address location

assign SL_2_jump_wire = ( Data[25:0] << 2 );


// Concatenate PC + 4 (4 bits) + 28 bits offset of JUMP

assign jump_addr_wire = { PC_update_wire[31:28], SL_2_jump_wire };


// Calculate New PC

assign New_PC_addr = Jump_wire ? jump_addr_wire : branch_MUX_wire;


// Data Memory output - Read data output

assign dMemOut = Data_mem_out_wire;


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: 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
always @(negedge 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
```

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: 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
```

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
```

(2) Use the testbench to test the functionality of the overall design. Display the output of the simulator.

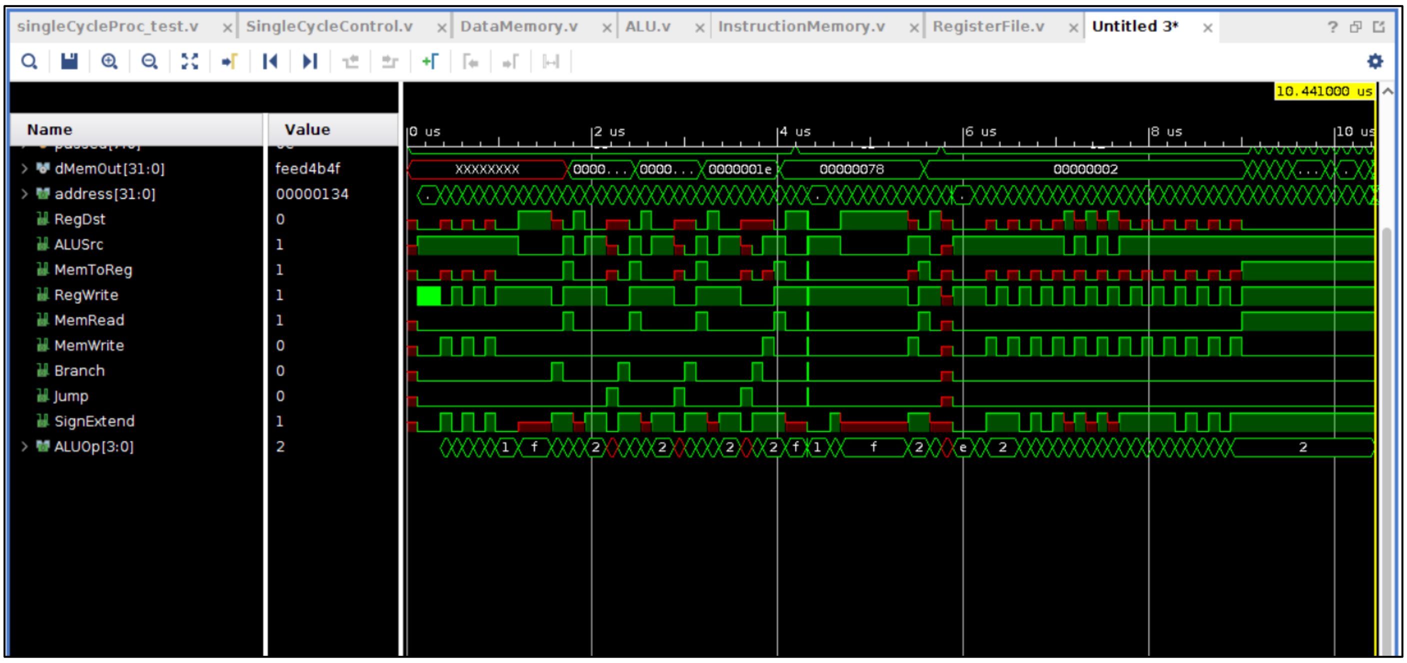
Testbench code filename: ALUControlTest.v

```
`timescale 1ns / 1ps

`define STRLEN 32

`define HalfClockPeriod 60

`define ClockPeriod `HalfClockPeriod * 2

module SingleCycleProcTest_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;

        // Instantiate the Unit Under Test (UUT)

        SingleCycleProc uut (

                .CLK(CLK),

                .Reset_L(Reset_L),

                .startPC(startPC),

                .dMemOut(dMemOut)

        );

        initial begin

                // Initialize Inputs

                Reset_L = 1;
```

```verilog
startPC = 0;
passed = 0;
// Wait for global reset
#(1 * `ClockPeriod);
// Program 1
#1 Reset_L = 0; startPC = 0;
#(1 * `ClockPeriod);
Reset_L = 1;
#(33 * `ClockPeriod);
passTest(dMemOut, 120, "Results of Program 1", passed);

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

// Program 3
#(1 * `ClockPeriod)
Reset_L = 0; startPC = 32'hA0;
#(1 * `ClockPeriod);
Reset_L = 1;
#(26 * `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'hfeedbeef, "Result 5 of Program 3", passed);
#(1 * `ClockPeriod);
passTest(dMemOut, 32'hfeedbeef, "Result 6 of Program 3", passed);
#(1 * `ClockPeriod);
passTest(dMemOut, 32'hfeedbeef, "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);
            $stop;
      end
   initial begin
      CLK = 0;
   end
   // The following is correct if clock starts at LOW level at StartTime //
   always begin
      #`HalfClockPeriod CLK = ~CLK;
      #`HalfClockPeriod CLK = ~CLK;
   end
endmodule
```

Simulation Output Waveform: (Single Cycle MIPS Processor)



Name	Value
dMemOut[31:0]	feed4b4f
address[31:0]	00000134
RegDst	0
ALUSrc	1
MemToReg	1
RegWrite	1
MemRead	1
MemWrite	0
Branch	0
Jump	0
SignExtend	1
ALUOp[3:0]	2

Simulation Output Logs: (Single Cycle MIPS Processor)

```
INFO: [USF-XSim-61] Executing 'SIMULATE' step in
'/home/grads/p/pranav_anantharam/lab5_final/project_1/project_1.sim/sim_1/behav/xsim'

INFO: [USF-XSim-98] *** Running xsim

   with args "SingleCycleProcTest_v_behav -key
{Behavioral:sim_1:Functional:SingleCycleProcTest_v} -tclbatch {SingleCycleProcTest_v.tcl} -log
{simulate.log}"

INFO: [USF-XSim-8] Loading simulator feature

Vivado Simulator 2018.3

Time resolution is 1 ps

source SingleCycleProcTest_v.tcl

# set curr_wave [current_wave_config]

# if { [string length $curr_wave] == 0 } {

#   if { [llength [get_objects]] > 0} {

#     add_wave /

#     set_property needs_save false [current_wave_config]

#   } else {

#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without
a wave window. If you want to open a wave window go to 'File->New Waveform Configuration' or
type 'create_wave_config' in the TCL console."

#   }

# }

# run 1000ns

INFO: [USF-XSim-96] XSim completed. Design snapshot 'SingleCycleProcTest_v_behav' loaded.

INFO: [USF-XSim-97] XSim simulation ran for 1000ns

launch_simulation: Time (s): cpu = 00:00:11 ; elapsed = 00:00:10 . Memory (MB): peak = 7518.594
; gain = 16.820 ; free physical = 132605 ; free virtual = 152355

restart

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
```

(3) Synthesize your design. Provide the summary results of the synthesis process.

Synthesis Report: (Single Cycle 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     |   166|
|5     |LUT3     |   149|
|6     |LUT4     |    89|
|7     |LUT5     |   218|
|8     |LUT6     |   529|
|9     |MUXF7    |    44|
|10    |MUXF8    |     1|
|11    |RAM32M   |    12|
|12    |RAMB18E1 |     1|
|13    |FDCE     |    28|
|14    |FDPE     |    28|
|15    |LDC      |    28|
|16    |IBUF     |    30|
|17    |OBUF     |    32|
+------+---------+------+
Report Instance Areas:

+------+---------+------------+------+

|      |Instance |Module      |Cells |

+------+---------+------------+------+

|1     |top      |            |  1395|

|2     |  ALU_1  |ALU         |    31|

|3     |  DM_1   |DataMemory  |     2|

|4     |  PC_1   |PC          |   956|

|5     |  RF_1   |RegisterFile |   342|

+------+---------+------------+------+

---------------------------------------------------------------------------------

Finished Writing Synthesis Report : Time (s): cpu = 00:08:25 ; elapsed = 00:08:33 . Memory
(MB): peak = 1889.656 ; gain = 426.344 ; free physical = 128695 ; free virtual = 149197

---------------------------------------------------------------------------------
```

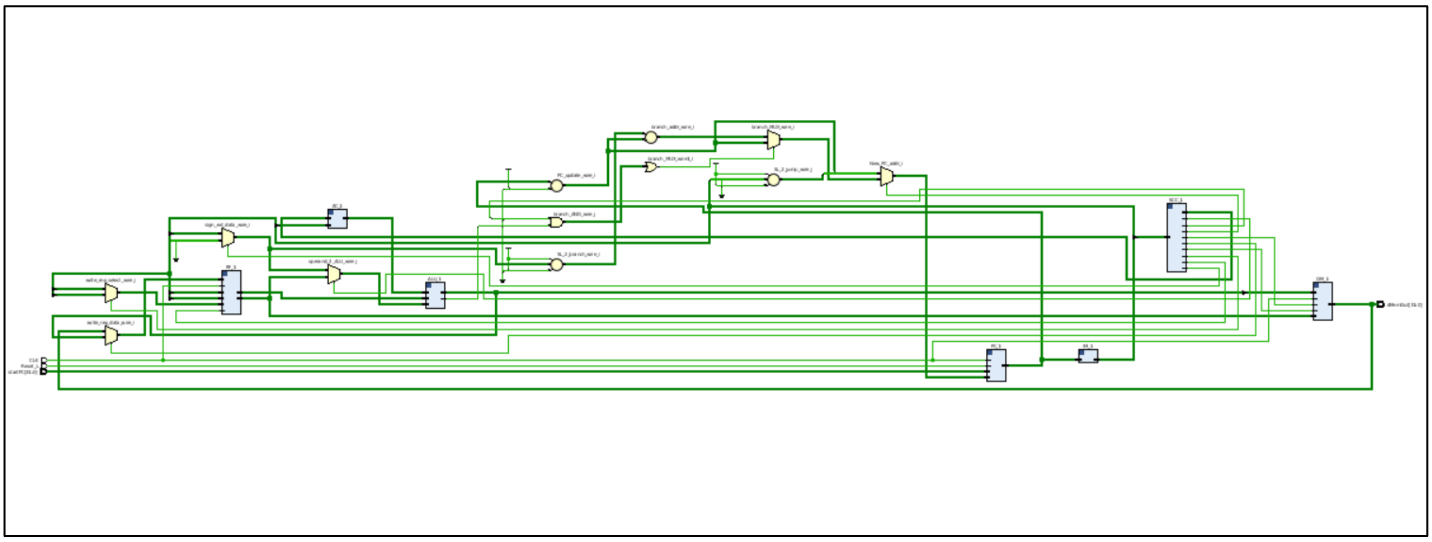
Synthesis finished with 0 errors, 0 critical warnings and 1 warnings.

```
Synthesis Optimization Runtime : Time (s): cpu = 00:08:25 ; elapsed = 00:08:33 . Memory (MB):
peak = 1889.656 ; gain = 426.344 ; free physical = 128697 ; free virtual = 149199

Synthesis Optimization Complete : Time (s): cpu = 00:08:25 ; elapsed = 00:08:33 . Memory (MB):
peak = 1889.660 ; gain = 426.344 ; free physical = 128707 ; free virtual = 149209

INFO: [Project 1-571] Translating synthesized netlist

INFO: [Netlist 29-17] Analyzing 124 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 1 inverter(s) to 12 load pin(s).

Netlist sorting complete. Time (s): cpu = 00:00:00 ; elapsed = 00:00:00.01 . Memory (MB): peak
= 1945.684 ; gain = 0.000 ; free physical = 128642 ; free virtual = 149145

INFO: [Project 1-111] Unisim Transformation Summary:

  A total of 40 instances were transformed.

  LDC => LDCE: 28 instances

  RAM32M => RAM32M (inverted pins: WCLK) (RAMD32, RAMD32, RAMD32, RAMD32, RAMD32, RAMD32,
RAMS32, RAMS32): 12 instances


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

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

```
synth_design completed successfully

synth_design: Time (s): cpu = 00:08:30 ; elapsed = 00:08:38 . Memory (MB): peak = 1945.684 ;
gain = 511.301 ; free physical = 128703 ; free virtual = 149205

Netlist sorting complete. Time (s): cpu = 00:00:00 ; elapsed = 00:00:00 . Memory (MB): peak =
1945.684 ; gain = 0.000 ; free physical = 128703 ; free virtual = 149205
```

Single Cycle MIPS Processor – Elaborated Design:



4. Answer the following review questions:

a) Explain why we designed the data memory such that reads happen on the positive edge of the clock, while writes happen on the negative edge of the clock.

Answer:

The data memory is designed such that reads happen on the positive edge of clock and writes happen on the negative edge of clock in order to keep the read/write operations mutually exclusive. It prevents read and write operations from occurring on the same clock edge. If read and write operations occur on the same clock edge, the data memory can get corrupted as the data maybe read out before it has been written completely. Moreover, it enables read and write operations to happen in a single cycle, allowing for forwarding of values from one instruction to another in a pipeline.

b) Take a look at the test programs provided in the instruction memory Verilog file (only test programs 1 - 3) and briefly explain what each of them do and what instructions they test.

Answer:

Test program 1:

An array is created { 50, 40, 30 }. The loop iterates through the array elements and adds to the sum variable. The three array elements are added and the sum of '120' is stored at the end of the array. After executing the instructions, the array elements are : { 50, 40, 30 , 120 }The expected testbench output is compared with the calculated output.

Instructions tested: LW, SW, BEQ, LW, ADD, JUMP

Test program 2:

Various arithmetic operations are executed and results are stored in memory. The final output after executing all the instructions is stored in memory and compared with the expected testbench output.

Instructions tested: LW, SW, ADD, ADDI, SUB, SLT, OR, AND

Test program 3:

ALU operations (signed and unsigned) are tested with immediate values. The result of each instruction is stored in memory and compared with the expected testbench output.

Instructions tested: LW, SW, XORI, SLTIU, SLTI, SRA, SRL, SLL, ADDIU, ANDI

c) If you wanted to support the **bne** instruction, what modifications would you have to do to your design? Include changes to all affected subcomponents as well.

Answer:

To support the **bne** instruction, an additional control signal should be added to the output of the single cycle control unit exclusively for **BNE** instructions. The ALU output Zero signal is used with the **bne** control signal to calculate the branch offset for updating the PC.

Sample implementation in SingleCycleControl.v:

```
`BNEOPCODE: begin
        RegDst <= #2 1'bx;
        ALUSrc <= #2 1'b0;
        MemToReg <= #2 1'b0;
        RegWrite <= #2 1'b0;
        MemRead <= #2 1'b0;
        MemWrite <= #2 1'b0;
        BNE <= #2 1'b1;
        Branch <= #2 1'b0;
        Jump <= #2 1'b0;
        SignExtend <= #2 1'b1;
        ALUOp <= #2 `SUB;
end
```

Hardware modifications are required in the Single Cycle Clock Unit and Interconnect logic.

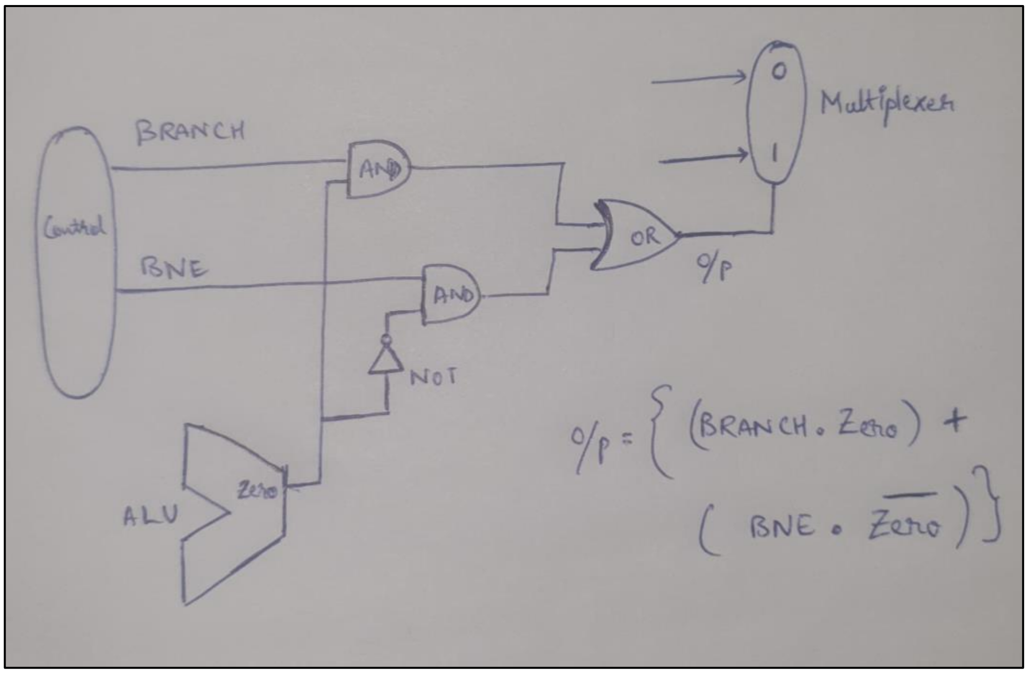
BNE control signal will be 1 for BNE instructions and 0 for BEQ instructions

Branch signal will be 0 for BNE instructions and 1 for BEQ instructions.

Required Expression:

{ (Branch AND Zero) OR (BNE AND (NOT Zero)) }

The output of the above expression is fed as the select line / control signal for the Branch Multiplexer to decide whether to take the branch offset.


Multiplexer
0
1
BRANCH
Control
BNE
AND
AND
OR
O/P
NOT
ALU
Zero
O/P = { (BRANCH . Zero) + (BNE . $\overline{\text{Zero}}$) }

d) What clock rate did the synthesis process estimate your overall design would run at? Explain why this design is inefficient and provide suggestions for improvement.

Answer:

The MIPS design implemented does not support pipelining, meaning that new instructions are fetched only on the completion of previous instructions. If there are any dependences across instructions, then in a non-pipelined design, a large number of stalls will occur, leading to lower performance.

By implementing and integrating pipelining into the MIPS design, the efficiency can be greatly improved. The execution of each instruction involves 5 stages in a 5-stage pipeline: Fetch, Decode, Execute, Memory, Write Back. When the first instruction is being decoded, the next instruction can be fetched. Hence, by pipelining instructions we can execute up to one instruction every cycle in an ideal case.