ECEN 651

LAB NO. 5

Non-pipelined MIPS Processor

TA: Mr. Ye Wang

DATE: 10/09/2019

Student Name: Dhiraj Dinesh Kudva

UIN: 829009538

Objective: The objectives of this lab are

1. To complete the design of a single cycle (non-pipelined) processor.: In this, the modules designed in the earlier lab such as register file, data memory, etc. will be connected along with the new modules like ALU Control Unit (part of this lab) to form a non pipelined single cycle MIPS processor.

Design:

The below figure is the MIPS Block diagram. In this case, only a single cycle without pipeline MIPS architecture is considered. The instruction memory has all the instructions to be carried which is decoded by the control unit and based on the instruction the result is stored in the register file or memory.

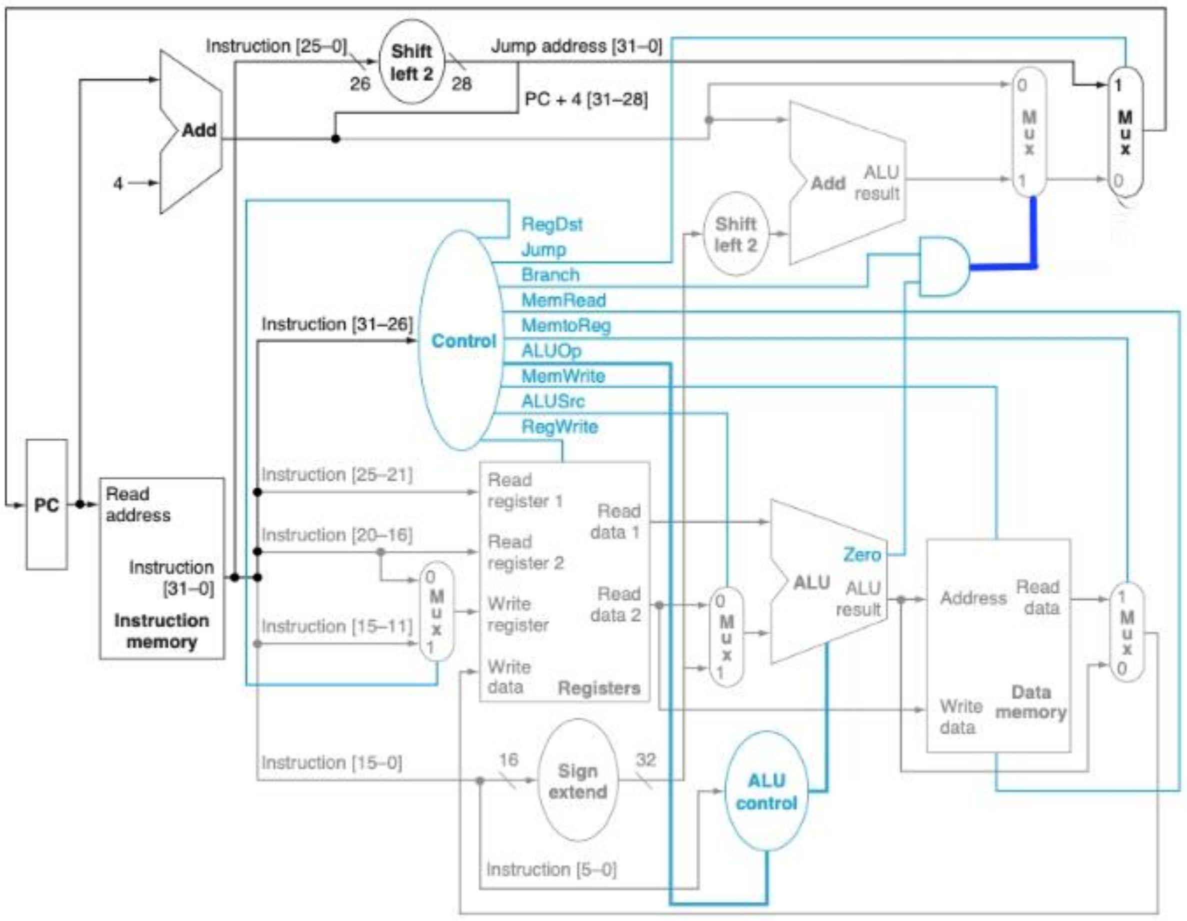


Figure 1: MIPS Block Diagram

Part 1: Designing the ALU

The below figure is representation of the ALU. BusA and BusB are the input operands. Based on the ALUCtrl signal, which basically acts as a select signal, the result is calculated and output is generated at BusW.

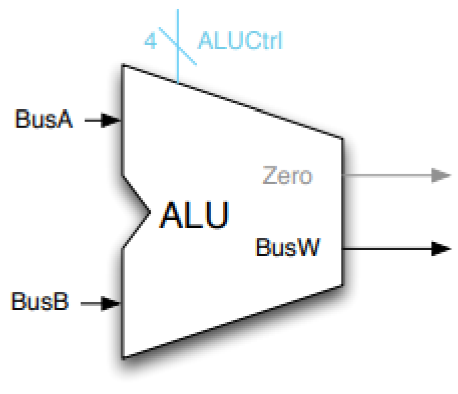


Figure 2: ALU port interface

Opperation	ALU Control Line
AND	0000
OR	0001
ADD	0010
SLL	0011
SRL	0100
SUB	0110
SLT	0111
ADDU	1000
SUBU	1001
XOR	1010
SLTU	1011
NOR	1100
SRA	1101
LUI	1110

The ALU can be thought of as combination of arithmetic and logic blocks, each sharing the same inputs, such that the outputs of each of the blocks are multiplexed to create the result.

Verilog Code for ALU:

```
`timescale 1ns / 1ps

//case definition
`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 mips1(BusW, Zero, BusA, BusB, ALUCtrl);/*module
initialization*/
input wire [31:0] BusA, BusB;//inputs
output reg [31:0] BusW;//output
```

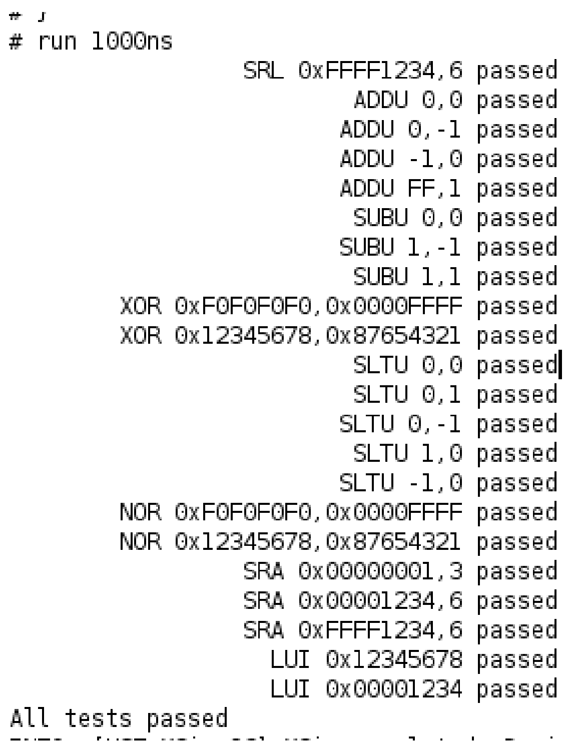
```
input wire [3:0] ALUCtrl ; //input
output wire Zero ; //output
wire less;
assign Zero = {{BusW==0} ? 1'b1 :1'b0}; /*only when the output is
zero, this will be high*/
assign less = ({1'b0,BusA} < {1'b0,BusB}  ? 1'b1 : 1'b0);
always@(*)begin

      case (ALUCtrl)
      `AND:  BusW <= BusA & BusB; //logical AND
      `OR:    BusW <= BusA | BusB; //logical OR
      `ADD:   BusW <= BusA +BusB; //addition
      `ADDU:  BusW <= BusA +BusB; //addition without overflow
      `SLL:   BusW <= BusA<<BusB; //shift logical left
      `SRL:   BusW <= BusA>>BusB; //shift logical right
      `SUB:begin

      BusW<=BusA-BusB; //subtraction
       end
      `SUBU:begin
                  BusW<=BusA-BusB;//subtraction with overflow
                  end
      `XOR:   BusW <= BusA ^BusB; //logical XOR
      `NOR:   BusW <= ~(BusA | BusB); //logical NOR
      `SLT:begin  //set less than
      if(BusA[31]!=BusB[31])
      begin
            if (BusA[31]>BusB[31])
                 begin
                 BusW<=1;
                 end
            else
                begin
                 BusW<=0;
                 end
      end
      else if(BusA > BusB && BusA[31]==BusB[31])
      begin
             BusW<=0;
             end
             else
             begin
             BusW<=1;
             end
        end
      `SLTU:begin //set less than
               if(BusA>BusB || BusA==BusB)
                      BusW<=0;
                      else
                      BusW<=1;
      end
      `SRA: //shift arithmetic right
      BusW <= $signed(BusA)>>>(BusB);
      `LUI:   //BusW <= BusB << 16;
      BusW <={BusB[15:0],16'd0}; //load upper immediate
      default:BusW <= 0; //default case
      endcase
end
endmodule
```

Testbench:

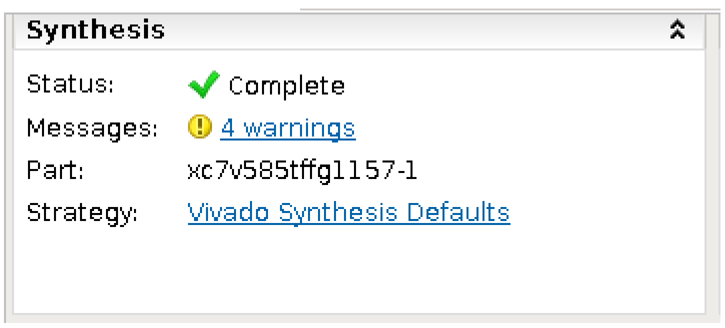
```
`timescale 1ns / 1ps
`define STRLEN 32
module ALUControlTest_v;
task passTest;
            input [32:0] actualOut, expectedOut;
            input [`STRLEN*8:0] testType;
            inout [7:0] passed;

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

      // Inputs
      reg [31:0] BusA;
      reg [31:0] BusB;
      reg [3:0] ALUCtrl;
      reg [7:0] passed;

      // Outputs
      wire [31:0] BusW;
      wire Zero;

      // Instantiate the Unit Under Test (UUT)
      mips1 uut (
            .BusW(BusW),
            .Zero(Zero),
            .BusA(BusA),
            .BusB(BusB),
            .ALUCtrl(ALUCtrl)
      );
      initial begin
            // Initialize Inputs
            BusA = 0;
            BusB = 0;
            ALUCtrl = 0;
            passed = 0;
            // Add stimulus here
            //ADD YOUR TEST VECTORS FROM THE PRELAB HERE
{BusA, BusB, ALUCtrl} = {32'd6,32'hFFFF1234 , 4'd4}; #40;
passTest({Zero, BusW}, 33'h003FFFC48, "SRL 0xFFFF1234,6", passed);
{BusA, BusB, ALUCtrl} = {32'h00000000, 32'h00000000, 4'd8}; #40;
passTest({Zero, BusW}, 33'h100000000, "ADDU 0,0", passed);
{BusA, BusB, ALUCtrl} = {32'h00000000, 32'hFFFFFFFF, 4'd8}; #40;
passTest({Zero, BusW}, 33'h0FFFFFFFF, "ADDU 0,-1", passed);
{BusA, BusB, ALUCtrl} = {32'hFFFFFFFF, 32'h00000000, 4'd8}; #40;
passTest({Zero, BusW}, 33'h0FFFFFFFF, "ADDU -1,0", passed);
{BusA, BusB, ALUCtrl} = {32'h000000FF, 32'h00000001, 4'd8}; #40;
passTest({Zero, BusW}, 33'h000000100, "ADDU FF,1", passed);
{BusA, BusB, ALUCtrl} = {32'h00000000, 32'h00000000, 4'd8}; #40;
passTest({Zero, BusW}, 33'h100000000, "SUBU 0,0", passed);
{BusA, BusB, ALUCtrl} = {32'h00000001, 32'hFFFFFFFF, 4'd9}; #40;
passTest({Zero, BusW}, 33'h000000002, "SUBU 1,-1", passed);
{BusA, BusB, ALUCtrl} = {32'h00000001, 32'h00000001, 4'd9}; #40;
passTest({Zero, BusW}, 33'h100000000, "SUBU 1,1", passed);
{BusA, BusB, ALUCtrl} = {32'hF0F0F0F0, 32'h0000FFFF, 4'd10}; #40;
passTest({Zero, BusW}, 33'h0F0F00F0F, "XOR 0xF0F0F0F0,0x0000FFFF",
passed);
{BusA, BusB, ALUCtrl} = {32'h12345678, 32'h87654321, 4'd10}; #40;
passTest({Zero, BusW}, 33'h095511559, "XOR 0x12345678,0x87654321",
passed);
{BusA, BusB, ALUCtrl} = {32'h00000000, 32'h00000000, 4'd11}; #40;
passTest({Zero, BusW}, 33'h100000000, "SLTU 0,0", passed);
{BusA, BusB, ALUCtrl} = {32'h00000000, 32'h00000001, 4'd11}; #40;
passTest({Zero, BusW}, 33'h000000001, "SLTU 0,1", passed);
{BusA, BusB, ALUCtrl} = {32'h00000000, 32'hFFFFFFFF, 4'd11}; #40;
passTest({Zero, BusW}, 33'h000000001, "SLTU 0,-1", passed);
{BusA, BusB, ALUCtrl} = {32'h00000001, 32'h00000000, 4'd11}; #40;
passTest({Zero, BusW}, 33'h100000000, "SLTU 1,0", passed);
{BusA, BusB, ALUCtrl} = {32'hFFFFFFFF, 32'h00000000, 4'd11}; #40;
passTest({Zero, BusW}, 33'h100000000, "SLTU -1,0", passed);
{BusA, BusB, ALUCtrl} = {32'hF0F0F0F0, 32'h0000FFFF, 4'd12}; #40;
passTest({Zero, BusW}, 33'h00F0F0000, "NOR 0xF0F0F0F0,0x0000FFFF",
passed);
{BusA, BusB, ALUCtrl} = {32'h12345678, 32'h87654321, 4'd12}; #40;
passTest({Zero, BusW}, 33'h0688aa886, "NOR 0x12345678,0x87654321",
passed);
{BusA, BusB, ALUCtrl} = {32'd3, 32'h00000001, 4'd13}; #40;
passTest({Zero, BusW}, 33'h100000000, "SRA 0x00000001,3", passed);
{BusA, BusB, ALUCtrl} = {32'd6, 32'h00001234, 4'd13}; #40;
passTest({Zero, BusW}, 33'h000000048, "SRA 0x00001234,6", passed);
```

```
{BusA, BusB, ALUCtrl} = {32'd6,32'hFFFF1234 , 4'd13}; #40;
passTest({Zero, BusW}, 33'h0FFFFFC48, "SRA 0xFFFF1234,6", passed);
{BusA, BusB, ALUCtrl} = {32'h0, 32'h12345678, 4'd14}; #40;
passTest({Zero, BusW}, 33'h056780000, "LUI 0x12345678", passed);
{BusA, BusB, ALUCtrl} = {32'h0, 32'h00001234, 4'd14}; #40;
passTest({Zero, BusW}, 33'h012340000, "LUI 0x00001234", passed);
allPassed(passed, 22);
                end;

endmodule
```

Output:



```
# run 1000ns
                   SRL 0xFFFF1234,6 passed
                           ADDU 0,0 passed
                          ADDU 0,-1 passed
                          ADDU -1,0 passed
                          ADDU FF,1 passed
                           SUBU 0,0 passed
                          SUBU 1,-1 passed
                           SUBU 1,1 passed
          XOR 0xF0F0F0F0,0x0000FFFF passed
          XOR 0x12345678,0x87654321 passed
                           SLTU 0,0 passed
                           SLTU 0,1 passed
                          SLTU 0,-1 passed
                           SLTU 1,0 passed
                          SLTU -1,0 passed
          NOR 0xF0F0F0F0,0x0000FFFF passed
          NOR 0x12345678,0x87654321 passed
                   SRA 0x00000001,3 passed
                   SRA 0x00001234,6 passed
                   SRA 0xFFFF1234,6 passed
                     LUI 0x12345678 passed
                     LUI 0x00001234 passed
All tests passed
```



Synthesis

Status: ✓ Complete

Messages: 4 warnings

Part: xc7v585tffg1157-1

Strategy: Vivado Synthesis Defaults

Part 2: ALU Control Logic

The process of decoding an instruction into a control code that the ALU can understand (Table 1) is split up into two blocks. The first block is contained within the control unit and the second block is the ALU control block.

The ALU control logic asynchronously functions as follows:

• When the ALU op bus is not equal to 4'b1111, the ALU op code should be passed directly to the ALU.

• When the ALU op bus is equal to 4'b1111, the function code should be used to determine the ALU control code.

Verilog Code:

```
`timescale 1ns / 1ps
//function definition
`define AND           4'b0000
`define OR       4'b0001
`define ADD      4'b0010
`define SLL      4'b0011
`define SRL      4'b0100
`define MULA     4'b0101
`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); // module
initialization
input [3:0]ALUop; //inputs
input [5:0]FuncCode;
output[3:0] ALUCtrl;//output
reg [3:0]ALUCtrl;
always@(FuncCode or ALUop)
begin
if(ALUop==4'b1111) //only when ALUop is 1111, the ALUCtrl is as per
the FuncCode
begin
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;
`MULAFunc:ALUCtrl= `MULA;
default: ALUCtrl=4'b0000;
endcase
end
else
ALUCtrl<=ALUop; //else ALUCtrl is same as the ALUop
end
endmodule
```

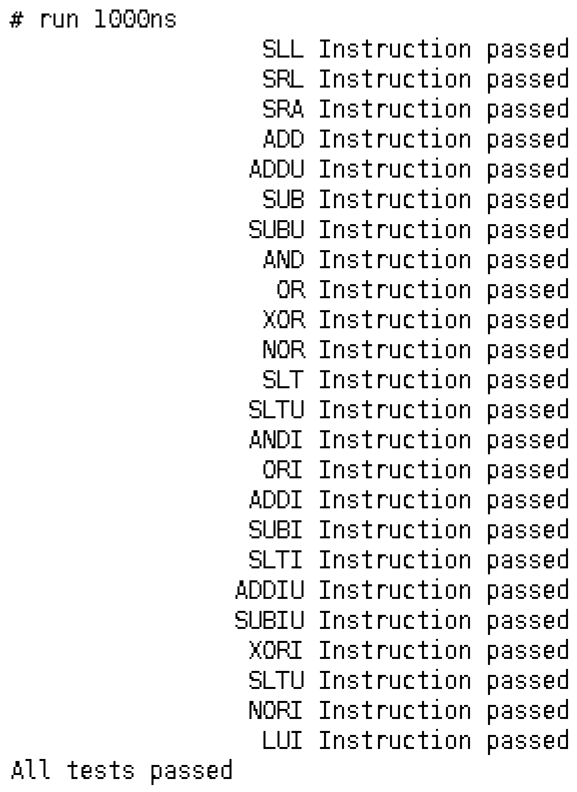
Testbench:

```
`timescale 1ns / 1ps
`define AND           4'b0000
`define OR       4'b0001
`define ADD      4'b0010
`define SLL      4'b0011
`define SRL      4'b0100
`define MULA     4'b0101
`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

`define STRLEN 32
```

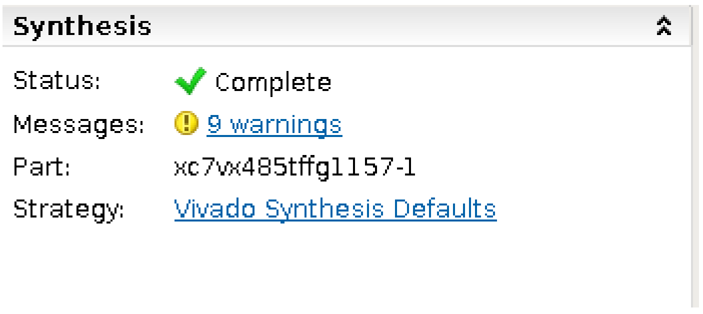
```
module ALUControlTest_v;


      task passTest;
            input [5:0] 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: %d should be %d", 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");
      endtask

      // Inputs
      reg [3:0] ALUop;
      reg [5:0] FuncCode;
      reg [7:0] passed;

      // Outputs
      wire [3:0] ALUCtrl;

      // Instantiate the Unit Under Test (UUT)
      ALUControl uut (
            .ALUCtrl(ALUCtrl),
            .ALUop(ALUop),
            .FuncCode(FuncCode)
      );

      initial begin
            // Initialize Inputs
            passed = 0;

            {ALUop, FuncCode} = {4'b1111, `SLLFunc};
            #10
            passTest(ALUCtrl, `SLL, "SLL Instruction", passed);

            {ALUop, FuncCode} = {4'b1111, `SRLFunc};
            #10
            passTest(ALUCtrl, `SRL, "SRL Instruction", passed);

            {ALUop, FuncCode} = {4'b1111, `SRAFunc};
            #10
            passTest(ALUCtrl, `SRA, "SRA Instruction", passed);

            {ALUop, FuncCode} = {4'b1111, `ADDFunc};
            #10
            passTest(ALUCtrl, `ADD, "ADD Instruction", passed);

            {ALUop, FuncCode} = {4'b1111, `ADDUFunc};
            #10
      passTest(ALUCtrl, `ADDU, "ADDU Instruction", passed);

            {ALUop, FuncCode} = {4'b1111, `SUBFunc};
            #10
            passTest(ALUCtrl, `SUB, "SUB Instruction", passed);

            {ALUop, FuncCode} = {4'b1111, `SUBUFunc};
            #10
      passTest(ALUCtrl, `SUBU, "SUBU Instruction", passed);

            {ALUop, FuncCode} = {4'b1111, `ANDFunc};
            #10
            passTest(ALUCtrl, `AND, "AND Instruction", passed);

            {ALUop, FuncCode} = {4'b1111, `ORFunc};
            #10
            passTest(ALUCtrl, `OR, "OR Instruction", passed);

            {ALUop, FuncCode} = {4'b1111, `XORFunc};
            #10
            passTest(ALUCtrl, `XOR, "XOR Instruction", passed);

            {ALUop, FuncCode} = {4'b1111, `NORFunc};
            #10
            passTest(ALUCtrl, `NOR, "NOR Instruction", passed);

            {ALUop, FuncCode} = {4'b1111, `SLTFunc};
            #10
            passTest(ALUCtrl, `SLT, "SLT Instruction", passed);

            {ALUop, FuncCode} = {4'b1111, `SLTUFunc};
            #10
      passTest(ALUCtrl, `SLTU, "SLTU Instruction", passed);

            {ALUop, FuncCode} = {`AND, 6'bXXXXXX};
            #10
            passTest(ALUCtrl, `AND, "ANDI Instruction", passed);

            {ALUop, FuncCode} = {`OR, 6'bXXXXXX};
            #10
            passTest(ALUCtrl, `OR, "ORI Instruction", passed);

            {ALUop, FuncCode} = {`ADD, 6'bXXXXXX};
```

```
        #10
        passTest(ALUCtrl, `ADD, "ADDI Instruction", passed);

        {ALUop, FuncCode} = {`SUB, 6'bXXXXXX};
        #10
        passTest(ALUCtrl, `SUB, "SUBI Instruction", passed);

        {ALUop, FuncCode} = {`SLT, 6'bXXXXXX};
        #10
        passTest(ALUCtrl, `SLT, "SLTI Instruction", passed);

        {ALUop, FuncCode} = {`ADDU, 6'bXXXXXX};
        #10
    passTest(ALUCtrl, `ADDU, "ADDIU Instruction", passed);

        {ALUop, FuncCode} = {`SUBU, 6'bXXXXXX};
        #10
    passTest(ALUCtrl, `SUBU, "SUBIU Instruction", passed);

        {ALUop, FuncCode} = {`XOR, 6'bXXXXXX};
        #10
        passTest(ALUCtrl, `XOR, "XORI Instruction", passed);

        {ALUop, FuncCode} = {`SLTU, 6'bXXXXXX};
        #10
    passTest(ALUCtrl, `SLTU, "SLTU Instruction", passed);

        {ALUop, FuncCode} = {`NOR, 6'bXXXXXX};
        #10
        passTest(ALUCtrl, `NOR, "NORI Instruction", passed);

        {ALUop, FuncCode} = {`LUI, 6'bXXXXXX};
        #10
        passTest(ALUCtrl, `LUI, "LUI Instruction", passed);

        allPassed(passed, 24);
    end

endmodule
```

Output:



```
# run 1000ns
                          SLL Instruction passed
                          SRL Instruction passed
                          SRA Instruction passed
                          ADD Instruction passed
                         ADDU Instruction passed
                          SUB Instruction passed
                         SUBU Instruction passed
                          AND Instruction passed
                           OR Instruction passed
                          XOR Instruction passed
                          NOR Instruction passed
                          SLT Instruction passed
                         SLTU Instruction passed
                         ANDI Instruction passed
                          ORI Instruction passed
                         ADDI Instruction passed
                         SUBI Instruction passed
                         SLTI Instruction passed
                        ADDIU Instruction passed
                        SUBIU Instruction passed
                         XORI Instruction passed
                         SLTU Instruction passed
                         NORI Instruction passed
                          LUI Instruction passed
All tests passed
```



Synthesis

Status: ✓ Complete

Messages: 9 warnings

Part: xc7vx485tffg1157-1

Strategy: Vivado Synthesis Defaults

Part 3: Control Unit Logic

The control unit orchestrates the flow of data through the microprocessor by decoding the 6-bit op code within the current instruction.

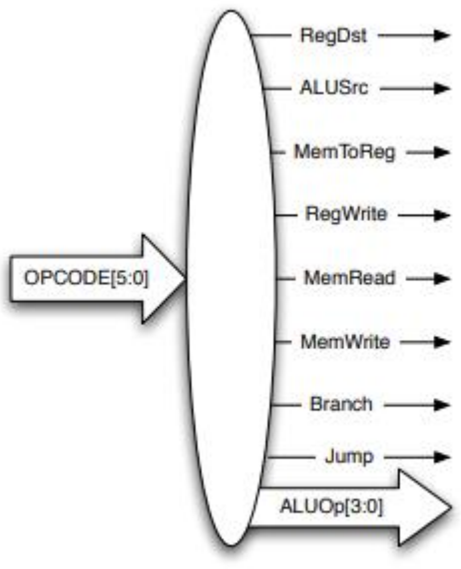


Figure 3: Control Unit Decoder

Verilog Code:

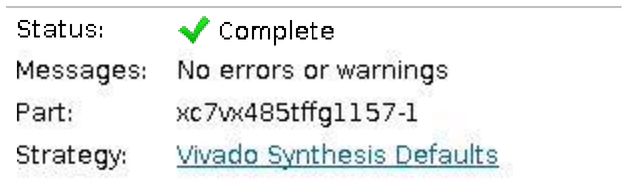
```
`timescale 1ns / 1ps
`define RTYPEOPCODE 6'b000000
`define LWOPCODE        6'b100011
`define SWOPCODE        6'b101011
`define BEQOPCODE       6'b000100
`define JOPCODE     6'b000010
`define ORIOPCODE       6'b001101
`define ADDIOPCODE  6'b001000
`define ADDIUOPCODE 6'b001001
`define ANDIOPCODE  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 initialization
module SingleCycleControl(
RegDst,ALUSrc,MemToReg,RegWrite,MemRead,MemWrite,Branch,Jump,SignExt
end,ALUOp,Opcode);
input[5:0] Opcode; //inputs
      output reg RegDst; //outputs
      output reg ALUSrc;
      output reg MemToReg;
      output reg RegWrite;
      output reg MemRead;
      output reg MemWrite;
      output reg Branch;
      output reg Jump;
      output reg SignExtend;
      output reg[3:0] ALUOp;
 always @(Opcode)begin //whenever the opcode changes, the following
case is executed
              case(Opcode)
                  //for Rtype instructions
```

```
`RTYPEOPCODE: begin
    RegDst <= 1'b1;
    ALUSrc <=  1'b0;
    MemToReg <= 1'b0;
    RegWrite <=  1'b1;
    MemRead <=  1'b0;
    MemWrite <=  1'b0;
    Branch <=  1'b0;

    Jump <=  1'b0;
    SignExtend <=  1'b0;
    ALUOp <=  `FUNC;
end
//for Load word instructions
`LWOPCODE: begin
                RegDst <= 1'b0;
                ALUSrc <= 1'b1;
                MemToReg <=  1'b1;
                RegWrite <= 1'b1;
                MemRead <=  1'b1;
                MemWrite <=  1'b0;
                Branch <=  1'b0;
                Jump <= 1'b0;
                ALUOp <= `ADD;

                SignExtend <= 1'b1;

 end
 //for store word instructions
 `SWOPCODE: begin
                RegDst <= 1'b0;
                ALUSrc <= 1'b1;
                MemToReg <= 1'b1;
                RegWrite <= 1'b0;
                MemRead <= 1'b0;
                MemWrite <= 1'b1;
                Branch <= 1'b0;
                Jump <=  1'b0;
                ALUOp <=  `ADD;

                SignExtend <=  1'b1;

   end
//for branch if equal instructions
 `BEQOPCODE: begin
                  RegDst <=  1'b0;
                  ALUSrc <=  1'b0;
                  MemToReg <=  1'b0;
                  RegWrite <=  1'b0;
                  MemRead <=  1'b0;
                  MemWrite <=  1'b0;
```

```
                    Branch <=  1'b1;
                    Jump <=  1'b0;
                    ALUOp <=  `SUB;
                    SignExtend <=  1'b1;

    end
  //for jump instructions
 `JOPCODE: begin
                        RegDst <=  1'b0;
                        ALUSrc <=  1'b0;
                        MemToReg <=  1'b0;
                        RegWrite <=  1'b0;
                        MemRead <=  1'b0;
                        MemWrite <=  1'b0;
                        Branch <=  1'b0;
                        Jump <=  1'b1;
                        SignExtend <=  1'b1;
                        ALUOp <=  `AND;
        end
 //for logical OR instructions
 `ORIOPCODE: begin
                        RegDst <=  1'b0;
                        ALUSrc <=  1'b1;
                        MemToReg <=  1'b0;
                        RegWrite <=  1'b1;
                        MemRead <=  1'b0;
                        MemWrite <=  1'b0;
                        Branch <=  1'b0;
                        Jump <=  1'b0;
                        SignExtend <=  1'b0;
                        ALUOp <= `OR;
            end
  //for addition
 `ADDIOPCODE: begin
                        RegDst <=  1'b0;
                        ALUSrc <=  1'b1;
                        MemToReg <=  1'b0;
                        RegWrite <=  1'b1;
                        MemRead <=  1'b0;
                        MemWrite <=  1'b0;
                        Branch <=  1'b0;
                        Jump <=  1'b0;
                        SignExtend <=  1'b1;
                        ALUOp <=  `ADD;
            end
 //for addition without overflow
 `ADDIUOPCODE: begin
                        RegDst <=  1'b0;
                        ALUSrc <=  1'b1;
                        MemToReg <=  1'b0;
                        RegWrite <=  1'b1;
                    MemRead <=  1'b0;
                    MemWrite <=  1'b0;
                    Branch <=  1'b0;
                    Jump <=  1'b0;
                    SignExtend <=  1'b0;
                    ALUOp <= `ADDU;
            end
 //for logical AND
`ANDIOPCODE: begin
                      RegDst <=  1'b0;
                      ALUSrc <=  1'b1;
                      MemToReg <=  1'b0;
                      RegWrite <=  1'b1;
                      MemRead <=  1'b0;
                      MemWrite <=  1'b0;
                      Branch <=  1'b0;
                      Jump <=  1'b0;
                      SignExtend <=  1'b0;
                      ALUOp <=  `AND;
              end
 //for load upper immediate
`LUIOPCODE: begin
                    RegDst <=  1'b0;//changed
                    ALUSrc <=  1'b1;
                    MemToReg <=  1'b0;
                    RegWrite <=  1'b1;
                    MemRead <=  1'b0;
                    MemWrite <=  1'b0;
                    Branch <=  1'b0;
                    Jump <=  1'b0;
                    SignExtend <= 1'b0;
                    ALUOp <=  `LUI;
            end
 //for shift less than
`SLTIOPCODE: begin
                  RegDst <=  1'b0;
                  ALUSrc <=  1'b1;
                  MemToReg <=  1'b0;
                  RegWrite <=  1'b1;
                  MemRead <=  1'b0;
                  MemWrite <=  1'b0;
                  Branch <=  1'b0;
                  Jump <=  1'b0;
                  SignExtend <= 1'b1;
                  ALUOp <= `SLT;
          end
`SLTIUOPCODE: begin
                    RegDst <= 1'b0;
                    ALUSrc <=  1'b1;
                    MemToReg <=  1'b0;
                    RegWrite <=  1'b1;
```

```
                                                MemRead <=  1'b0;
                                                MemWrite <=  1'b0;
                                                Branch <=  1'b0;
                                                Jump <=  1'b0;
                                                SignExtend <=  1'b1;
                                                ALUOp <=  `SLTU;
                                        end
                              //for logical XOR
                          `XORIOPCODE: begin
                                                  RegDst <=  1'b0;
                                                  ALUSrc <=  1'b1;
                                                  MemToReg <=  1'b0;
                                                  RegWrite <=  1'b1;
                                                  MemRead <=  1'b0;
                                                  MemWrite <=  1'b0;
                                                  Branch <=  1'b0;
                                                  Jump <=  1'b0;
                                                  SignExtend <=  1'b0;
                                                  ALUOp <= `XOR;
                                          end
                              //default case
                          default: begin
                              RegDst <=  1'bx;
                              ALUSrc <=  1'bx;
                              MemToReg <=  1'bx;
                              RegWrite <=  1'bx;
                              MemRead <=  1'bx;
                              MemWrite <=  1'bx;
                              Branch <=  1'bx;
                              Jump <=  1'bx;
                              SignExtend <= 1'bx;
                              ALUOp <=  4'bxxxx;
                          end
                      endcase
                  end
              endmodule
```

Output:



Status: ✓ Complete
Messages: No errors or warnings
Part: xc7vx485tffg1157-1
Strategy: Vivado Synthesis Defaults

Part 4: MIPS single cycle datapath:

In this path, interconnection of all the previous modules is done to obtain a single cycle MIPS datapath.

Verilog Code:

```
`timescale 1ns / 1ps
module SingleCycleProc( Clk,
           Reset_L,
           startPC,
           dMemOut
    ); // module initialization
//input and output initialization
input Clk,Reset_L;
        input [31:0] startPC;
        wire [31:0]startPC;
        wire [31:0] nextPC;
        output [31:0] dMemOut;
        wire [31:0]dMemOut;
        wire [27:0]updInstruction;
        wire [27:0]unupdInstruction;
        wire [31:0] jumpinstruction;
        wire[31:0] nextupdPC;
        wire [31:0] finalupdPC;
        wire [31:0] Data_memory_read_data_out;
 wire [31:0] instruction;
 wire [5:0] FuncCode;
wire [3:0]ALUop;
 wire [3:0]ALUCtrl;
  wire[5:0] Opcode;
   wire RegDst;
       wire ALUSrc;
     wire MemToReg;
      wire RegWrite;
        wire MemRead;
        wire MemWrite;
        wire Branch;
        wire Jump;
        wire SignExtend;
        wire [32:0] Signextended;
        wire [32:0] modSignextended;
        wire[3:0] ALUOp;
        wire [31:0] BusA, BusB,srca,srcb;
         wire [4:0] RA, RB, RW;
         wire RegWr;
        wire [31:0] muxoutputaddr;
wire [31:0] ALU_result, ALU_BusB;
wire [31:0] addr;
wire [31:0] write_data;
wire memwrite, memread;
wire [31:0] read_data ;
```

```
reg [31:0] address1;
wire shift;


//instruction memory instance creation
InstructionMemory imem (instruction, address1);
//Program counter: triggered at negative edge of clock and reset
always @(negedge Clk or negedge Reset_L)
 begin
 if(!Reset_L)
    address1<=startPC ;
    else
    address1<=muxoutputaddr;
 end
//single cycle control instance creation
SingleCycleControl scc
(RegDst,ALUSrc,MemToReg,RegWr,MemRead,MemWrite,Branch,Jump,SignExten
d,ALUOp,Opcode);
//register file instance creation
RegisterFile rf(BusA, BusB, Data_memory_read_data_out, instruction
[25:21], instruction [20:16], (RegDst? instruction [15:11]:
instruction[20:16]),RegWr,Clk);
//alu control unit instance creation
ALUControl acl (ALUCtrl,ALUOp,FuncCode[5:0]);
//ALU control instance creation
mips1 alu_final (ALU_result, Zero,srca,srcb, ALUCtrl);
//Data memory instance creation
data_memory dm(ALU_result, BusB, MemWrite, MemRead, Clk, read_data);
//sign extension
assign Signextended = SignExtend ?{{16{instruction[15]}},
instruction[15:0]} : {{16{1'b0}},instruction[15:0]};
//selection of output
assign ALU_BusB = ALUSrc ? Signextended:BusB;
//for shift logic
assign shift= (ALUCtrl==4'b0011 )?
1:(ALUCtrl==4'b0100)?1:(ALUCtrl==4'b1101)? 1:0;
assign srcb=shift?{27'b0,instruction[10:6]}:ALU_BusB;
assign srca=shift?ALU_BusB:BusA;
//Opcode
assign Opcode=instruction [31:26];
//FuncCode
assign FuncCode=instruction[5:0];
assign Data_memory_read_data_out = MemToReg?read_data:ALU_result;
assign nextPC =address1 +32'd4; //PC address updation
assign updInstruction ={instruction[25:0]}<<2;
assign jumpinstruction ={nextPC[31:28],updInstruction};/jump
instruction address*/
assign modSignextended =Signextended<<2;
 assign nextupdPC =nextPC + modSignextended;
 assign finalupdPC =(Branch & Zero)?nextupdPC:nextPC;/*checking
address for branch instruction*/
```

```
 assign muxoutputaddr =Jump?jumpinstruction:finalupdPC; /*checking
address condition for jump instruction*/
assign dMemOut=read_data; //output
endmodule
```

Verilog Code for instruction memory:

```
`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, 5 # 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
```

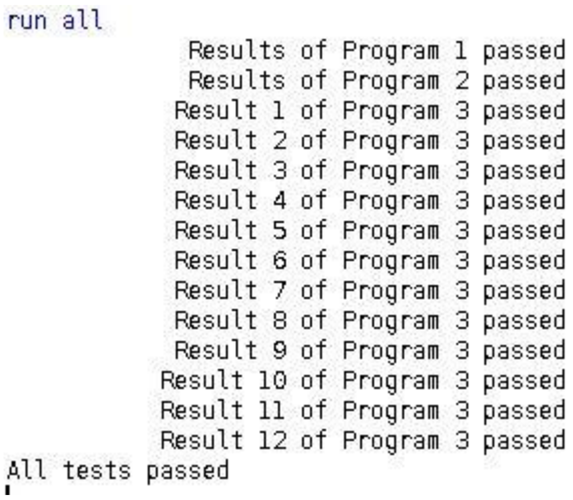
```
             */

            /*
            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
```

Testbench:

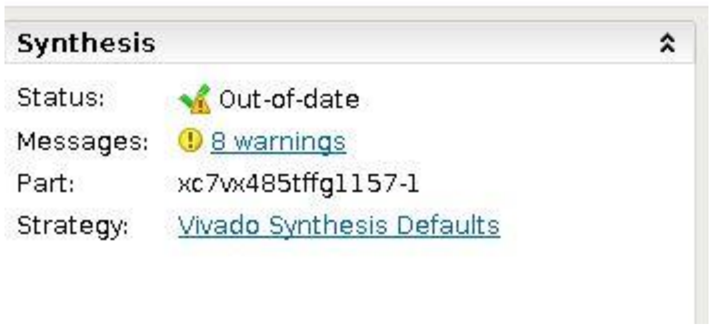
```
`timescale 1ns / 1ps
`define STRLEN 32
`define HalfClockPeriod 60
`define ClockPeriod `HalfClockPeriod * 2
module SingleCycleProcTest_v;

      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;
             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'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);
              $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
```

Output:



```
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
```



Synthesis

Status: Out-of-date

Messages: 8 warnings

Part: xc7vx485tffg1157-1

Strategy: Vivado Synthesis Defaults

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.

Ans. The data is read at the positive edge and written at negative edge to ensure that the read and write operation do not access the same memory location at any time. If a data is read at a particular memory location before write operation is completed, we will get a garbage value or an incorrect value. Also, if both read and write operates on the same memory location at the same clock edge, one of them would get stalled (like structural hazard) and this will cause an unwanted output and increase in latency.

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

Ans: Test Program 1:

It initializes an array starting at location 0 and with values 50, 40 and 30. Then it sets up a counter equal to the number of elements in the array. In this case, it is 3. Then the addition is carried out using loop and the final result is stored at the end of the array.

Instructions tested: load, store, add, sub, branch and jump.

Test Program 2:

It does a series of arithmetic computation. First value 32 is loaded in register a0. It initially stores value of 1 in register 2. Then makes it negative by subtracting with register 0 and stores at register 3. Compares this value with zero and sets register 5 if the value is less than 0. As the number stored in register 3 is negative, it sets the register 5. Then addition of register 5 and 2 is stored in register 6. Then a logical OR operation is carried out on register 5 and register 6 and the result is stored register 7. After this, a subtraction operation is carried out on register 5 and register 7 and the resulting negative number is stored in register 8. Then logical and is carried between register 8 and 7 and the result is stored in register 9. This value is then stored in a0 and loaded back in register 9.

Instructions tested: add, sub, and, or, load, store, set less than(slt).

Test Program 3:

In this a series of load and store instructions are carried out. Initially a value is stored at DMem[9] (address=36) which is 0xfeedbeef. This value is then loaded in register a0 and added (addition with overflow flag) with an immediate number -2656. As this is a negative number, the actual operation is subtraction. Then the result is stored at the next memory location. Similarly addition without overflow, and, shift left, shift right and shift right arithmetic operations are carried out on a0 value and the result is stored at the next memory locations. Then a set less than instruction is carried out on this. Finally a xor operation is carried out and all the result values are loaded from the memory.

Instructions tested: load immediate, load, store, and, add, addiu, sub, slti, shift left (sll),shift right logical(srl) , shift right arithmetic (sra) and logical xor (xori) .

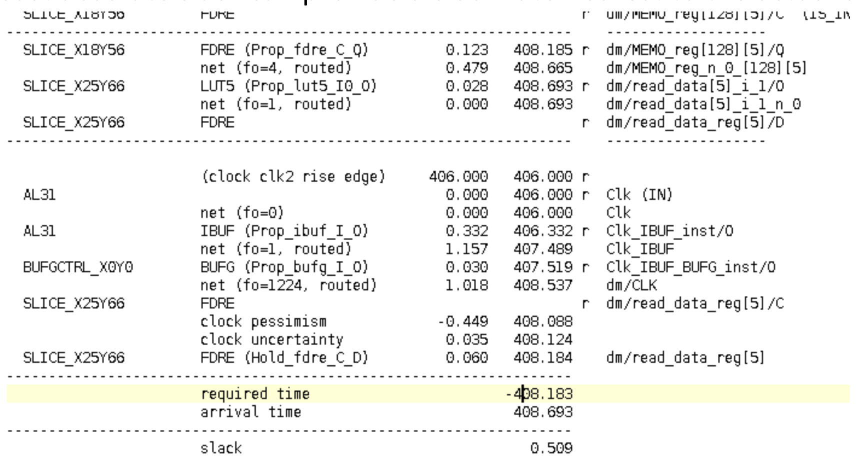
c. If you wanted to support the bne instruction, what modifications would you have to do to your design? Include changes to all affected subcomponents as well.

Ans: As per the MIPS diagram, considered for this lab, we have used an AND gate. The Branch and Zero signal is inputs of and gate. This satisfies for beqz (Branch if equal to zero instruction). In order to satisfy bne, we can add a NOT gate at the output of the

AND gate (or replace AND gate by NAND gate). Then the output would be branch if not equal to. The rest of the subcomponents will remain the same.

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.

Ans: The Vivado does not provide any clock rate after synthesis. However clock rate can be calculated by manually entering the timing constraint and checking the slack. The one with the least absolute slack can provide a clock rate nearest to the actual clock rate.



```
  SLICE_X18Y56       FDRE                                         r  dm/MEMO_reg[128][5]/C  (IS_IN
  -------------------------------------------------------------------    -------------------
  SLICE_X18Y56       FDRE (Prop_fdre_C_Q)         0.123   408.185 r  dm/MEMO_reg[128][5]/Q
                     net (fo=4, routed)           0.479   408.665    dm/MEMO_reg_n_0_[128][5]
  SLICE_X25Y66       LUT5 (Prop_lut5_I0_O)        0.028   408.693 r  dm/read_data[5]_i_1/O
                     net (fo=1, routed)           0.000   408.693    dm/read_data[5]_i_1_n_0
  SLICE_X25Y66       FDRE                                         r  dm/read_data_reg[5]/D
  -------------------------------------------------------------------    -------------------

                     (clock clk2 rise edge)     406.000   406.000 r
  AL31                                            0.000   406.000 r  Clk (IN)
                     net (fo=0)                   0.000   406.000    Clk
  AL31               IBUF (Prop_ibuf_I_O)         0.332   406.332 r  Clk_IBUF_inst/O
                     net (fo=1, routed)           1.157   407.489    Clk_IBUF
  BUFGCTRL_X0Y0      BUFG (Prop_bufg_I_O)         0.030   407.519 r  Clk_IBUF_BUFG_inst/O
                     net (fo=1224, routed)        1.018   408.537    dm/CLK
  SLICE_X25Y66       FDRE                                         r  dm/read_data_reg[5]/C
                     clock pessimism             -0.449   408.088
                     clock uncertainty            0.035   408.124
  SLICE_X25Y66       FDRE (Hold_fdre_C_D)         0.060   408.184    dm/read_data_reg[5]
  -------------------------------------------------------------------
                     required time                       -408.183
                     arrival time                         408.693
  -------------------------------------------------------------------
                     slack                                  0.509
```

As seen above, the slack closest to zero (0.509) occurs at time of 29 ns, which results in clock rate of 34.48 MHz. This is less because of the stalls and other latencies present in this design. This can be overcome by using pipelining where multiple instructions can be executed simultaneously.