University of Nevada Las Vegas. Department of Electrical and Computer Engineering Laboratories.

Class:	**Computer Architecture**	Semester:	**Spring 2022**
Points		Document author:	**Erick Serrano**
		Author's email:	**serrae4@unlv.nevada.edu**
		Document topic:	**Final Project**
Instructor's comments:			

1. Introduction / Theory of Operation

In this experiment, we design a soft microprocessor with a restriction of 8-bit instructions. We borrowed the template from the MIPS single cycle processor and simplified it to accommodate built-in instructions. We do not use the conventional SMP8 microprocessor and instead endeavor to create a custom microprocessor. We constructed the processor so that our instructions were interpreted as such:

Opcode	**Function**	**Address**	**Data**

XXX	XX	X	XX

In other words, we limit our register file and data memory to two addresses. In addition, the largest amount of data either one can store is 2-bits of information. However, we sacrifice the size of the memory/registers in exchange for the opportunity to execute multiple operations. Among these operations are:

1. **Load Memory(LW)**
2. **Store Register(SW)**
3. **Move Register(MV)**
4. **Move Immediate(MVi)**
5. **Add Immediate(Addi)**
6. **Subtract Immediate(Subi)**
7. **Branch if Equal to Zero (beq)**
8. **Jump (jmp)**

To accommodate these instructions, we also modified the capabilities of the ALU in the microprocessor. The following are instructions that the ALU is capable of executing:

Opcode Table		**Function Table**	
ALUOp (Meaning)	**Meaning**	**Function**	**ALUControl**
00	Pass Data	00	If reg=0, PC+=data
01	Add Immediate	01	If reg = 0, PC-=data
10	Subtract Immediate	10	PC+= data
11	Function	11	PC-=data

While reducing the arithmetic capabilities of the ALU, we provide more versatility to the jump functions instead. Further work can be done to reduce the number of slots in the function table to use the *jmp* and *beq* functions.

2. Control Word Table and Machine Code for testing:

1. *Control Word Table*

Opcode	Meaning	$WE_{1,0}$	$RE_{1,0}$	ALUOp	MemOrRegW	ALUOrMemOut
000	LW **(M→R)**	01	10	00	0	1

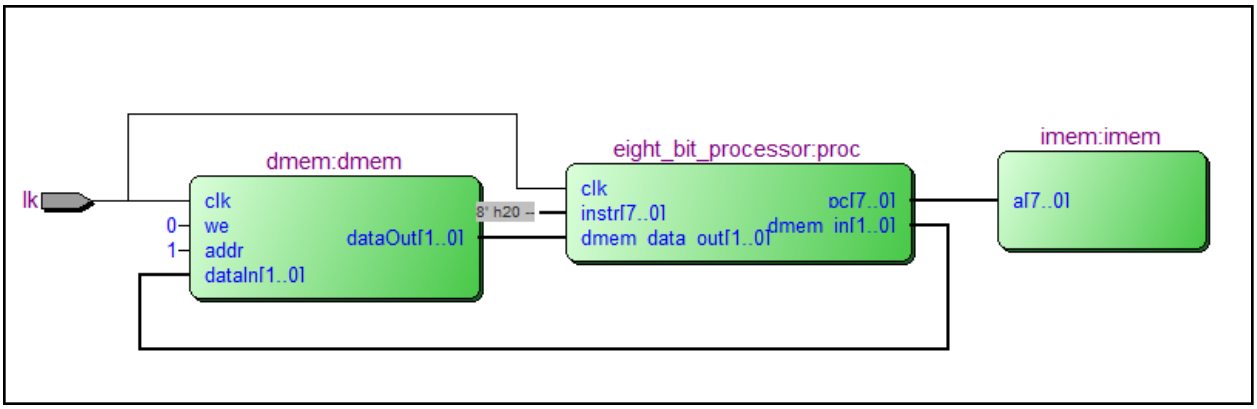
001	SW **(R→M)**	10	01	01	1	X
010	MV **(R→R)**	01	01	01	0	0
011	MVi **(C→R)**	01	00	00	0	X
100	Addi **(C→R)**	01	01	01	0	0
101	Subi **(C→R)**	01	01	10	0	0
110	Beq **(If R is zero, jmp)**	00	01	11	X	X
111	Jmp **(PC± data)**	00	00	11	X	X

2. *Machine Code for Test Code 1 and 2*
    a. *Memfile.dat (remove the comments before implementation)*

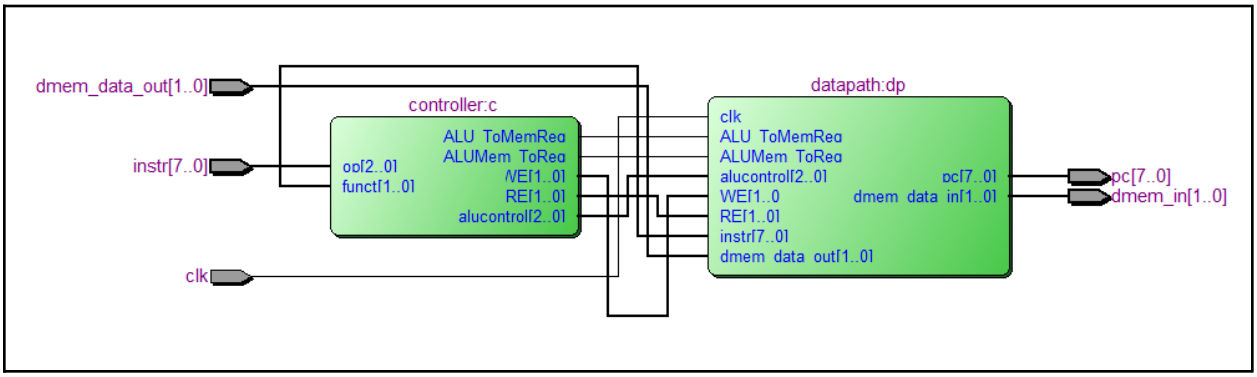
```
66 // MVi R1, 2
24 // SW M1, R1
40 // MV R0, R1
c5//subi R1, 1
ec//beq -4
b0//jmp 2
66//MVi R1, 2
04//LW R1, M1
85//Addi R1,3
```

3. Results of Experiments

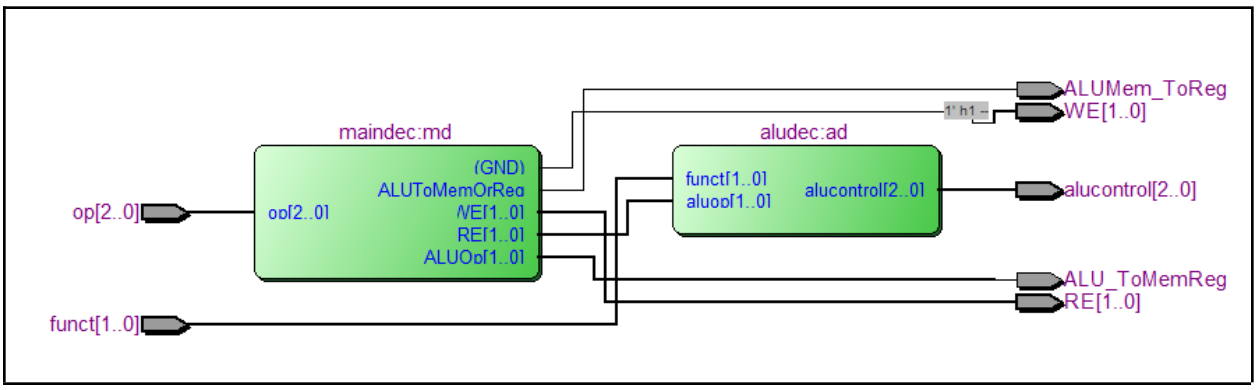
1. *Experiment 1*
    a. **Block Diagram of Microprocessor**
        i. *Top-Level RTL Design*



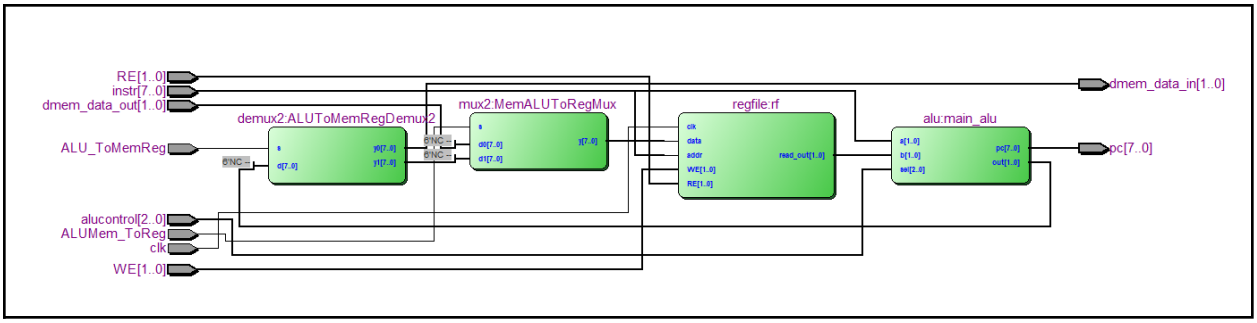
ii. Eight-Bit-Processor RTL



iii. Controller RTL



iv. Datapath RTL



2. *Experiment 2*
    a. Verilog Code of the design

```
module top (input clk);
wire [7:0] instr; // Instruction
wire [7:0]pc; //Current PC
wire[1:0] dmem_data_out,dmem_in; // Data from dmem, to dmem
// instantiate processor and memories
eight_bit_processor proc(clk, pc, instr, dmem_in,dmem_data_out);//INSIDE
imem imem (pc, nstr);//FINISHED
dmem dmem (clk, we, dmem_data_out,dmem_in, instr[5]);//FINISHED

endmodule

module dmem (input clk, we,
output [1:0]dataOut,input [1:0]dataIn,input addr);
//Clock, RE, WE, DATAIN, DATAOUT, ADDR,
reg RAM[1:0];

assign dataOut = RAM[addr];

always @ (posedge clk)begin
if (we)
RAM[addr] <= dataIn;
end
endmodule

module imem (input [7:0] a, output [2:0] rd);
reg [2:0] RAM[63:0]; // limited memory
initial
begin
$readmemh ("memfile.dat",RAM);//FINISHED
end
assign rd = RAM[a]; // word aligned
endmodule
```

```
module eight_bit_processor (input clk,
output [7:0] pc,
input [7:0] instr,
output[1:0] dmem_in,//Data to send to dmem
input [1:0]dmem_data_out);//FINISHED

wire [1:0] WE, RE;
wire ALU_ToMemReg,ALUMem_ToReg;
wire [2:0] alucontrol;

controller c(instr[2:0], instr[4:3], WE, RE,
ALU_ToMemReg, ALUMem_ToReg, alucontrol);//FINISHED

//Clock, Reset, ALUControl(wire from controller==>datapath), Input Enable
//Write Enable, Read Enable, MuxSel from ALU to MEM/Reg,
//MuxSel from ALU/MEM to reg,Program Counter, instruction,
//Output from ALU to dmem, Addr to write inside dmem, addr to read from dmem,
//Data from dmem to write inside Register!
datapath dp(clk, alucontrol,
WE, RE, ALU_ToMemReg,ALUMem_ToReg, pc,instr, dmem_in,
dmem_data_out);//FINISHED

endmodule

module controller (input [2:0] op, input [1:0] funct,
output [1:0] WE, RE,
output ALU_ToMemReg,ALUMem_ToReg,
output [2:0] alucontrol);

wire [1:0] aluop;

maindec md(op, WE, RE, aluop, ALU_ToMemReg, ALUMem_ToReg);//FINISHED

aludec ad (funct, aluop, alucontrol);//FINISHED

endmodule

module datapath (input clk,
input[2:0] alucontrol,
input [1:0]WE, RE,
input ALU_ToMemReg,ALUMem_ToReg,
output signed [7:0] pc,
input [7:0] instr,
output [1:0]dmem_data_in, input [1:0]dmem_data_out);

wire [1:0] rf_result, reg_out;
```

```
wire [1:0] mux_result = 2'b00;
wire [1:0] aluout;

//ALU Next (Input, Input, Input, output, output), connects to DECODER:
alu main_alu(instr[7:6], rf_result, alucontrol, pc,aluout);//FINISHED
//MemALUToRegMux2
mux2 MemALUToRegMux(dmem_data_out, reg_out, ALUMem_ToReg,
mux_result);//FINISHED
//Regfile first; only 2 registers:
regfile rf(clk, mux_result,WE,RE,instr[5], rf_result);//FINISHED
//demux ALUToMemRegDemux, connected to regfile
demux2 ALUToMemRegDemux2(aluout, ALU_ToMemReg, dmem_data_in,
reg_out);//FINISHED
endmodule

module aludec (input [1:0] funct,
input [1:0] aluop,
output reg [2:0] alucontrol);
always @ (*)

case (aluop)
2'b00: alucontrol <= 3'b000; // Pass data
2'b01: alucontrol <= 3'b001; // Add Immediate
2'b10: alucontrol <= 3'b010; // Sub Immediate
default: case(funct) // RTYPE
6'b00: alucontrol <= 3'b011; // If reg=0, PC+= data
6'b01: alucontrol <= 3'b100; // If reg = 0, PC -= data
6'b10: alucontrol <= 3'b101; // PC+=data
6'b11: alucontrol <= 3'b110; // PC-=data
default: alucontrol <= 3'bxxx; // ???
endcase
endcase

endmodule

module maindec (input [2:0] op, output IE,
output [1:0] WE, RE, ALUOp,
output ALUToMemOrReg, ALUOrMemToReg);

reg [7:0] controls;

assign {WE, RE, ALUOp, ALUToMemOrReg, ALUOrMemToReg}  = controls;

always @ (* )
 case(op)
3'b000: controls <= 8'b01100001;                //LW (Copies from Mem to Reg)
```

```
3'b001: controls <= 8'b1001011X;              //SW (Copies from reg to Mem)
3'b010: controls <= 8'b01010100;              //MV (Copies from reg to reg)
3'b011: controls <= 8'b0100000X;              //MVi (Copies from Constant to Reg)
3'b100: controls <= 8'b01010100;              //Addi (Add Constant to Reg)
3'b101: controls <= 8'b01011000;              //Subi (Subtract Constant from Reg)
3'b110: controls <= 8'b000111XX;              //beq (if reg ==0 move pc up to 4 instructions)
3'b111: controls <= 8'b000011XX;              //Jmp (Change PC up to 4 instructions)
default:  controls <= 8'bXXXXXXXX;                    //???
endcase
endmodule



module demux2 # (parameter WIDTH = 8)
(input [WIDTH-1:0] d, input s,
output reg [WIDTH-1:0] y0, y1);

always@(*) begin
if (s === 0)begin
        y0 <= d;
        y1 <= 0;
end
else begin
        y1 <= d;
        y0 <= 0;
        end
end
endmodule

module mux2 # (parameter WIDTH = 8)
(input [WIDTH-1:0] d0, d1, input s,
output [WIDTH-1:0] y);
assign y = s ? d1 : d0;
endmodule
module alu (a,b,sel, pc,out);
        input [1:0] a,b;
        input [2:0] sel;
        output reg [1:0] out;
        output reg signed  [7:0] pc;
        reg signed [1:0] pc_next;
        initial begin
                out <= 0;
        end
        //Whenever pc_next is changed, change pc too!
        always @(pc_next) begin
                pc = pc+pc_next;
```

```
        end
        always @ (*) begin
                case(sel)
                        3'b000:begin//Pass Data
                                out <= a;
                                pc_next <= 2'b01;
                        end
                        3'b001:begin//Add Immediate
                                out<= b+a;
                                pc_next <= 2'b01;
                        end
                        3'b010:begin//Subtract Immediate
                                out<=b-a;
                                pc_next <= 2'b01;
                        end
                        3'b011:begin
                                if(b === 0)//Conditional Jump Down
                                        pc_next <= a;
                                else
                                        pc_next <= 2'b01;
                        end
                        3'b100:begin//Conditional Jump Up
                                if(b===0)
                                        pc_next <= -a;
                                else
                                        pc_next <= 2'b01;
                        end
                        3'b101:begin//Unconditional Jump Down
                                pc_next <= a;
                        end
                        3'b110:begin//Unconditional Jump Up
                                pc_next <= -a;
                        end
                        3'b111:begin

                        end
                endcase
  end
endmodule

module regfile (input clk, data,
input [1:0]WE,RE,
input addr,
output [1:0] read_out);
reg rf[1:0];
// 2X2 bit register file
```

```
always @ (posedge clk)
if (WE) rf[addr] <= data;
assign read_out = (RE == 1) ? rf[addr] : 0;
endmodule
```

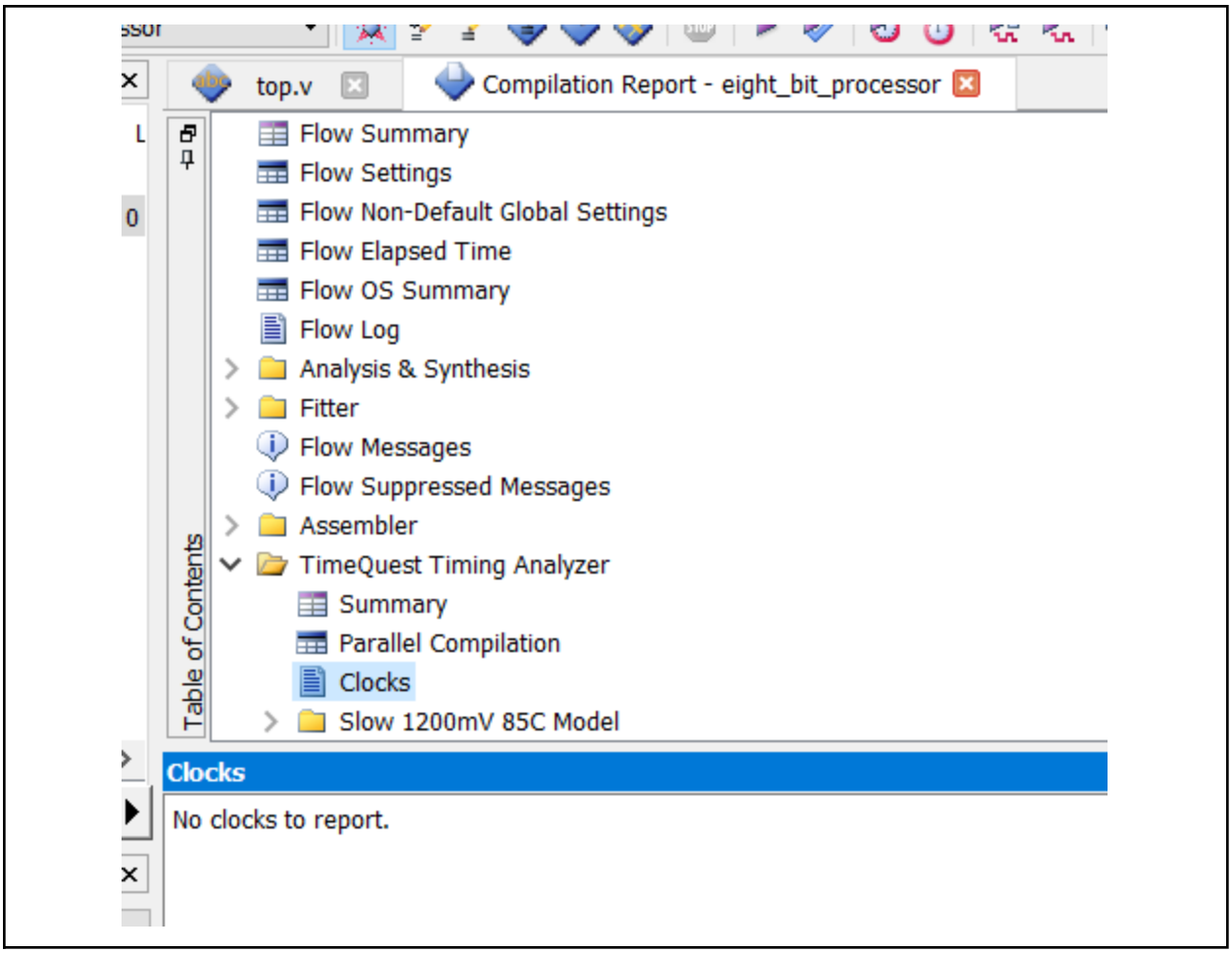
b. Testbench code for test code 1 and 2

```
module testbench();
reg clk;
// instantiate device to be tested
top dut (clk);
// generate clock to sequence tests
always
begin
clk <= 1;
 # 5;
 clk <= 0;
 # 5; // clock duration
end
endmodule
```

c. VCS Waveform
   i. *Unable to provide the VCS waveform due to the limited I/O ports of the top entity.*

d. DE2 Delivery
   i. *Unable to provide the DE2 Delivery due to the limited I/O ports of the top entity.*

3. *Experiment 3*
   a. Screenshot of the timing report before and after setting up timing constraints
      i. *We were unable to report timing since the report did not have a clock available.*

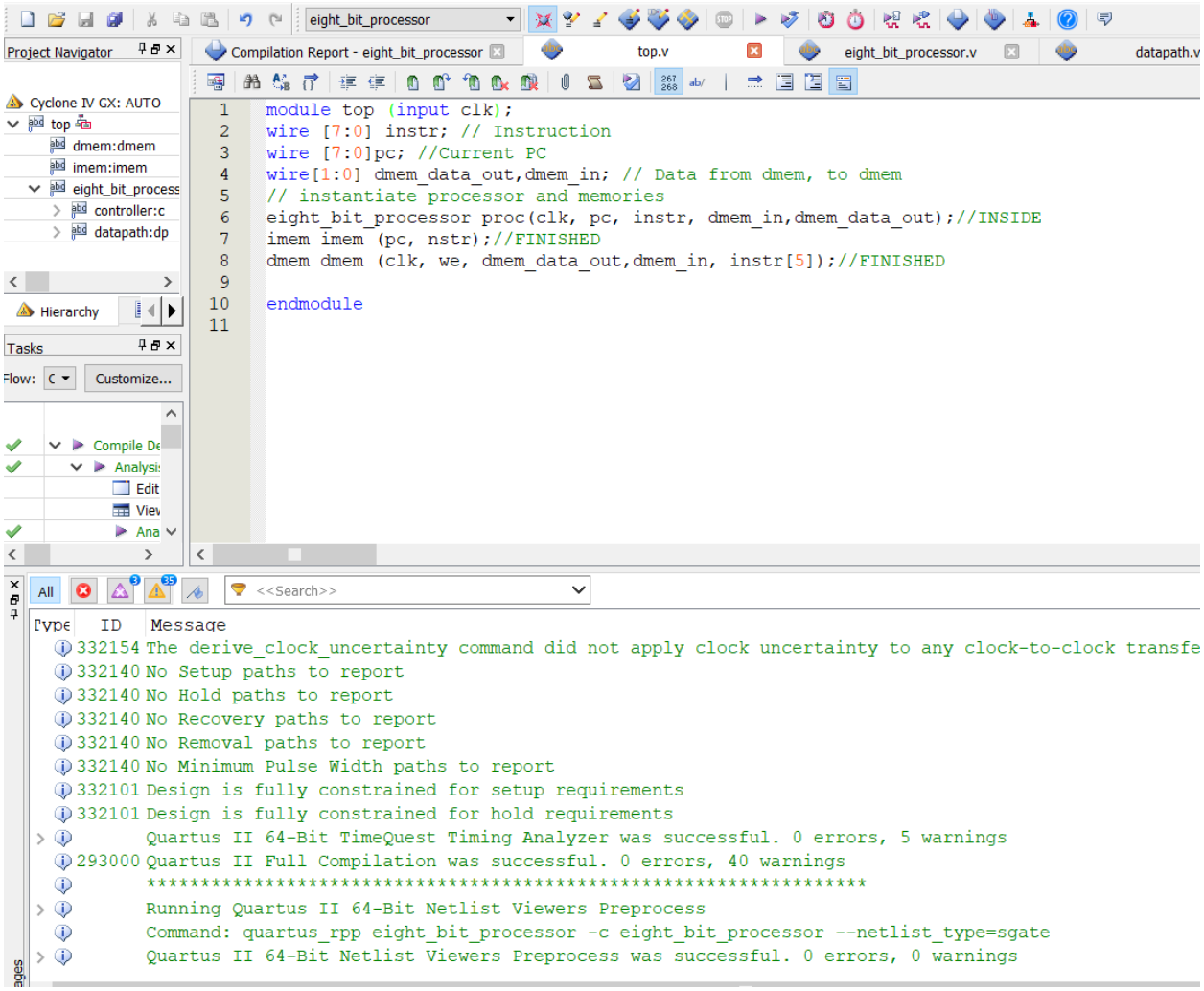


b. Calculate how long it takes to complete TestCode1 and TestCode2
    i. *The microprocessor described is of the single cycle type. Assuming each instruction takes 1 cycle, and each cycle takes n seconds, the test code would take approximately 9*n seconds to complete since it executes 9 instructions.*

5. Conclusions & Summary

Write down your conclusions, things learned, problems encountered during the lab and how they were solved, etc…

In this experiment, we created a microprocessor to interpret basic machine code instructions. We gave the microprocessor custom instructions and ALU operations, and we limited the instructions to 8-bits. The way we formatted the instruction limited the size and number of registers and data memory slots. However, this created an opportunity to implement more instructions than usual. Below is a successful compilation of the microprocessor we designed:



The biggest difficulty we had was in testing and verifying the functionality of the microprocessor. However, we paid close attention to the ports and operations to ensure that the microprocessor would work as intended given the right ports to test. A good next step would be to receive information about the memory/register file and test it at the top level entity using a testbench.