Verilog Simulated 5 Stage CPU Report

By Jiayi Gu

Table of Contents

Section 1 Introduction

In this report, I present a comprehensive analysis of the design and implementation of a simple processor written in the Hardware Descriptive Language “Verilog.

In particular, this report focuses on the development and analysis of a processor's various components, functionality, design choices, and the integration of them to construct a fully functional processor.

This report is structured to guide the reader through the design process step by step. It commences with a detailed description of each individual component, outlining their operations and design choices. Subsequently, this report offers a table of resource utilization and timing analysis to offer clear insights into the component’s design and efficiency and potential bottlenecks in performance. Later sections illustrate the processor's capabilities by presenting the integration of the simple and extended processor and the respective data paths, highlighting the details of their operation.

Furthermore, the report walks through the design principles in developing the testbenches that encompasses comprehensive test cases for both individual components and the processor. This approach ensures the robustness and reliability of the processor under various conditions.

All relevant codes, testbenches and screen captures are included in the Appendix.

1.1 Warnings in compilation

It is assumed that “components.v” will not be compiled as a top level entity, and is therefore not discussed.

It is assumed that “Quartus Prime 18.1” and DE-10 will be used for compilation.

1.1.1 Processors

Warnings related to compiled code can be seen at Appendix 7.4.1. Timing related warnings are suppressed as timing analysis is not required on the processors.

Warning Code	Justification
292013	These warnings are unused software license warnings, which can be ignored.
15705	Certain pins such as HEX0~4 is not used in processor but is kept as it is used in extended. This can be safely ignored.
15714	It is suspected that certain I/O rules are not complete for some pins. This warning remains unresolved, but should not affect usage.
1771167/12241	R0 ~ R7 values are left dangling in top level when compiled. This causes this warning. However, these output values are used in testbenches. This can be safely ignored.
169177	This is a warning regarding voltage connection for interfacing 3 and 3.3V as they have different fanouts. It should not affect use in most applications.

1.1.3 Testbenches

Warnings related to testbenches can be seen at Appendix 7.4.2.

Warning Code	Justification
10036	These warnings originate from the declared registers being never read. This is expected and safe as they will only be used in $display system tasks, which is not counted as reading.
10175	During compilation unsupported system tasks such as $display or $stop are ignored but only used in ModelSim. This is expected and can be ignored.
12011/12110	The clock is not driven by any signals/source. It is set to oscillate at 10ns per cycle. This warning is expected and can be ignored.

Section 2 Task 1: Components

2.1 Section Overview

This section details the all the components used in the design of the processor. This report provides a short summary of each component's operation, functionality in processor and justifies the choices made in the design. At the end, details and analysis of the timing and resource usage of each component are collected in a table.

2.2 Sign Extender

2.2.1 Overview

This component performs a signed extension of input number to a given number of bits. This is achieved by prepending the input with the first bit up to the set number of bits.

2.2.2 Functionality

In the processor, this is used to sign extend 9-bit input immediate values to 16-bit values to be used in operations such as ***addi*** or ***movi***.

2.2.3 Design Choices

A parameter is set such that the sign extender can extend input to any arbitrary number of bits. Although not necessary in the functionality of this 16-bit fixed size processor, this allows for easy extension to a 32-bit processor in the future by changing the parameter.

2.3 Tick Finite State Machine

2.3.1 Overview

This component cycles between 4 predefined states, switching at each positive clock edge.

2.3.2 Functionality

In the processor, this is used to maintain the synchronous operation. Since each instruction may be too complex to complete within a tick, each tick correspond to a part of the instruction to perform, thereby allowing complex instructions and keep track of the state of the processor.

2.3.3 Design Choices

The states are encoded with one-hot encoding. Only 4 states are needed, such that at most 4 bits are required to represent the state compared to 2 bits using binary encoding. In exchange for the additional 2 bits for states, one-hot encoding benefits by simplifying the logic for decoding and operate faster.

2.4 Multiplexer

2.4.1 Overview

This component is a 16:1 multiplexer and selects input to place on output based on a selection signal.

2.4.2 Functionality

In the processor, this selects which register's value or the data input's value should be placed on the bus based on the control signal. It is a 16:1 multiplexer.

2.4.3 Design Choices

The value of the bus is defaulted to 0 is no selection input is given. This is to avoid unnecessary value on the bus line that may cause issues like accidental overrides of register values.

2.5 Arithmetic Logic Unit (ALU)

2.5.1 Overview

This component performs arithmetic operations including addition, subtraction, multiplication, left bit shift and right bit shift.

2.5.2 Functionality

In the processor, this actualizes the most instructions including ***add***, ***sub***. Appropriate inputs are given by the control unit and all operations are completed by this component.

2.5.3 Design Choices

Multiplication is done via Verilog built-in multiply (*) instead of IP blocks such as LPM_MULTIPLY. Via experimentation, the circuit synthesized in terms of size or speed is identical whichever option used. This is discussed further section 2.8 in timing analysis.

Additionally, overflow bits from addition or multiplication are ignored, and result is directly truncated. This is done since the processor only stores and operates with 16 bits, and additional bits are not expected to be used in any further calculations.

2.6 N bit register

2.6.1 Overview

This component serves as a synchronous sizable register for storing values.

2.6.2 Functionality

The processor contains 8 16-bit wide general purpose registers for storing results. A 9-bit register is used for storing instructions and, depending on the task, 2 to 3 16-bit registers are used for storing intermediate values for calculations in ALU.

2.6.3 Design Choices

Reset signal for the register is made synchronous, to prevent unexpected or undefined behaviour if reset of the registers are done during operations.

2.7 Display

2.7.1 Single digit BCD display

2.7.1.1 Overview

This component converts a 4 bit unsigned input to a 7 bit output of the corresponding integer on a 7 segment display.

2.7.1.2 Functionality

In the processor when run on DE-10, this is used to drive HEX5 display for displaying the tick and are used in the five-digit component to display the value in the ***disp*** instruction. Due to the tick being one-hot encoded. Tick 1 corresponds to 1. Tick 2 corresponds to 2. Tick 3 corresponds to 4. Tick 4 corresponds to 8.

2.7.2 Five-digit BCD display

2.7.2.1 Overview

This component converts a 16 bit unsigned input to five 7 bit outputs of the corresponding integer on five 7 segment displays. The input is assumed to be unsigned.

2.7.2.2 Functionality

In the processor when run on DE-10, this is used to drive HEX0 to HEX4 displays to display the value in the ***disp*** instruction.

2.7.2.3 Design Choices

4 LPM_DIVIDE IP blocks are used in the component. This is done because the block can output both the quotient and the remainder directly. If Verilog built-in arithmetic operations are used, a divide and a modulus operation will be needed, causing delay in the instruction and a bigger circuit to be synthesized.

2.8 Resource Usage and Timing Analysis

COMPONENT\METRIC	LOGIC ELEMENTS	REGISTERS	MAX FREQUENCY (MHZ)	RESTRICTED FREQUENCY (MHZ)
SIGN EXTENDER	26	25	1118.57	250
TICK FSM	9	5	776.40	250
MULTIPLEXER	9	8	833.33	250
ALU WITH *	84	51	179.60	179.60
ALU WITH IP BLOCK	84	51	179.57	179.57
16-BIT REGISTER	49	48	1230.01	250
9-BIT REGISTER	28	27	1228.50	250
SINGLE BCD OUTPUT	12	11	758.73	250
FIVE DIGIT BCD OUTPUT	554	51	12.24	12.24

Table 1 -Slow 1200mV 85C Model Timing & Resource Usage

2.8.1 Discussion

Overall, the component that caps the performance of the processor is the 5-digit BCD output used in the ***disp*** instruction due to the multiple division operations. It is worth noting that this instruction allows for 2 idle ticks, in which no operations happen, thereby mitigating this effect. Additionally, ALU is, as expected, the second most costly component. Here, it was found that using the LPM_MULTIPLY IP Block or using the * multiply operator in Verilog did not affect the resource or the frequency of the ALU. It is suspected that compiler has optimized the * operator to use the same logic as the IP block. For clarity in code, * operator is used.

Section 3 Task 2 Processor Datapath

3.1 Overview

This section will describe the process for the execution of ***addi*** and ***sub*** consecutively on the processor. The intent is to clarify the workings of the processor by the integration of the components and the control unit. Here, “din” refers to the external input to the processor. Register IR is a special 9-bit register for storing instructions given in Tick 1. Register A is the intermediate register for storing values for ALU calculations. Register G is the register for storing ALU outputs. The control signals are turned on based on saved instructions in IR.

Note that in each tick, operations are completed at the positive edge of the next tick. As an example, when control signal demands bus value to be written to R2 in tick 3, the writing operation will happen as the processor goes to tick 4. This is expected as this allows synchronous writes to happen and prevents timing issue that can arise from asynchronous changes. This also applied to the extended processor.

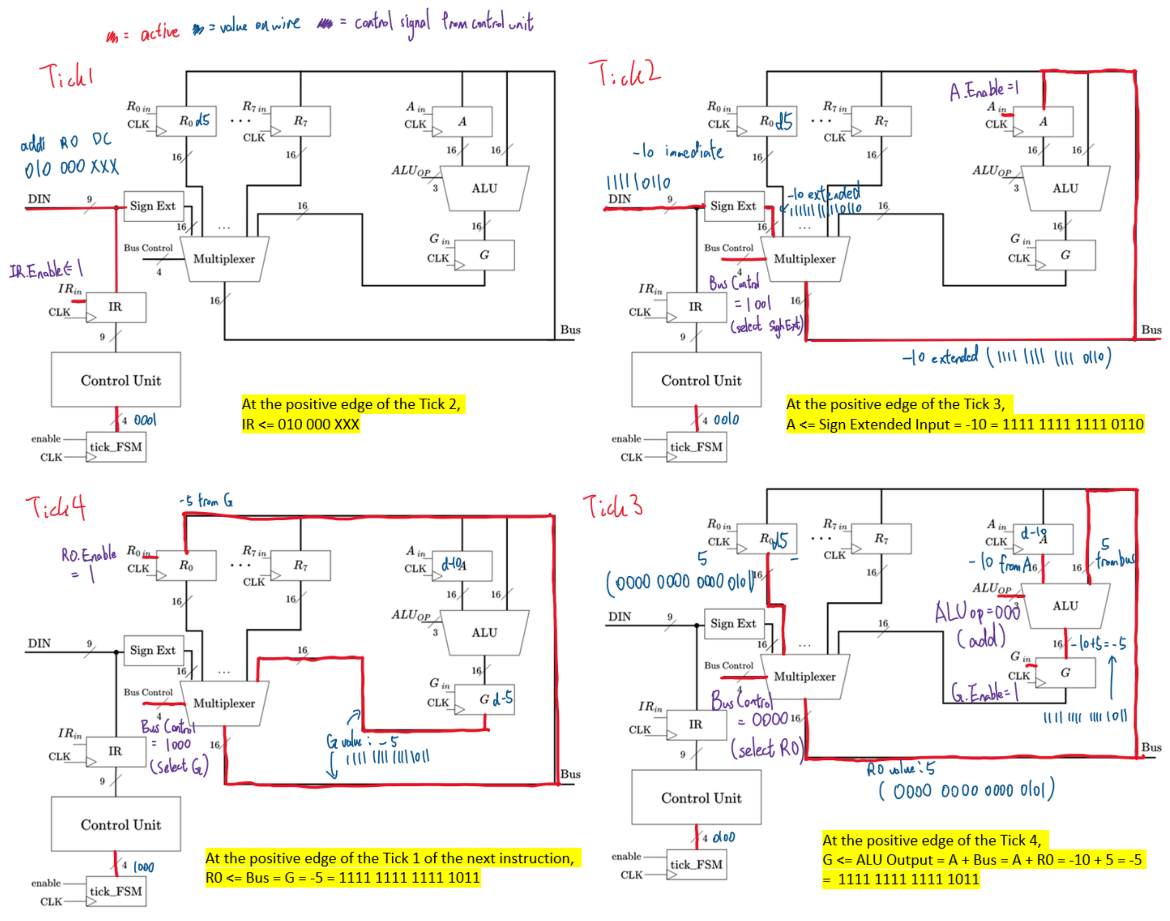
3.2 Instruction

The selected instructions are ***addi, R0, -10*** and ***sub, R0, R1***, performing R0 = (5+(-10))-110 = -115.

3.2.1 Before instruction

It is assumed that the processor register R0 contains the value d5 and R1 contains the value d110 before the 2 instructions are executed, to demonstrate the capability.

3.2.2 ***addi, 0, -10*** (Add-10 to R0)



At the positive edge of the Tick 2,
IR <= 010 000 XXX

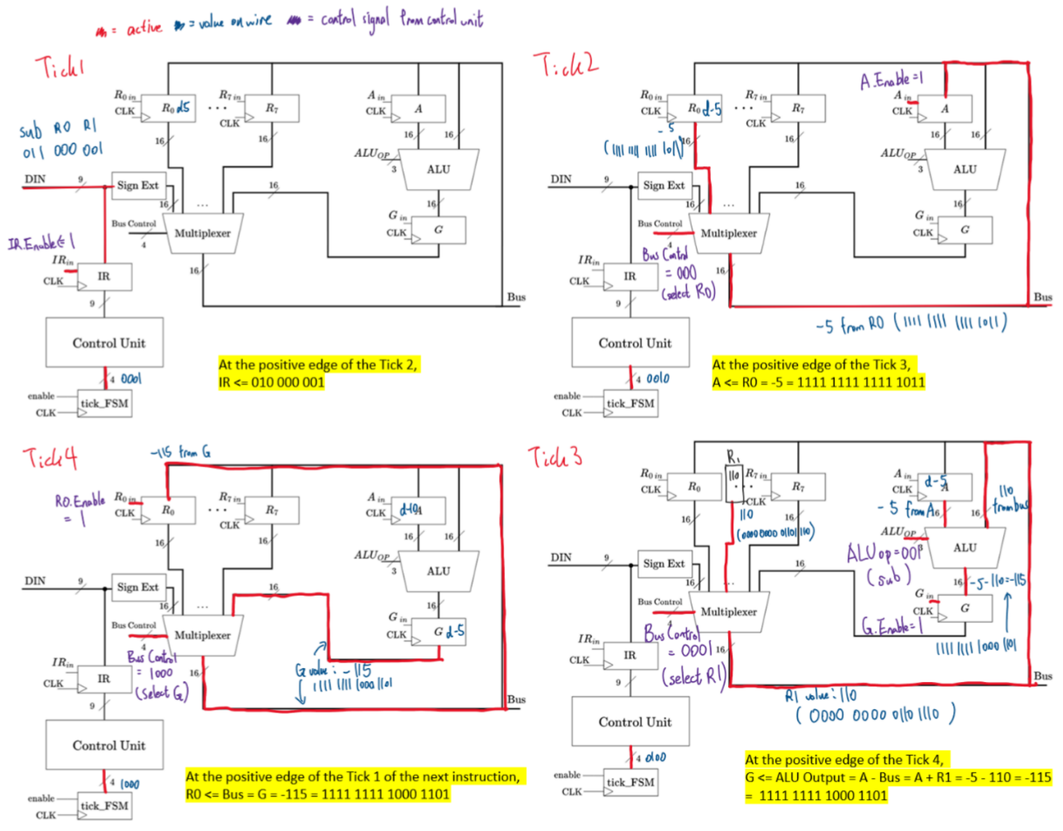
At the positive edge of the Tick 3,
A <= Sign Extended Input = -10 = 1111 1111 1111 0110

At the positive edge of the Tick 1 of the next instruction,
R0 <= Bus = G = -5 = 1111 1111 1111 1011

At the positive edge of the Tick 4,
G <= ALU Output = A + Bus = A + R0 = -10 + 5 = -5
= 1111 1111 1111 1011

Figure 1 - Datapath visualization for addi, 0, -10

3.2.3 ***sub, 0, 1*** (subtract r1 from r0 and save in r0)



At the positive edge of the Tick 2,
IR <= 010 000 001

At the positive edge of the Tick 3,
A <= R0 = -5 = 1111 1111 1111 1011

At the positive edge of the Tick 1 of the next instruction,
R0 <= Bus = G = -115 = 1111 1111 1000 1101

At the positive edge of the Tick 4,
G <= ALU Output = A - Bus = A + R1 = -5 - 110 = -115
= 1111 1111 1000 1101

Figure 2 - Datapath visualization for sub, R0, R1

Section 3 Task 3 Extended Processor Datapath

4.1 Overview

Same definition described in section 3.1 continue to be applied here. In addition, register H is used to indicate a 16-bit register reserved for displaying on 7 segment displays. Note that the register H is only shown when it is overwritten. However, it is always present, and its value is constantly fed to the BCD decoder for display on the corresponding hardware.

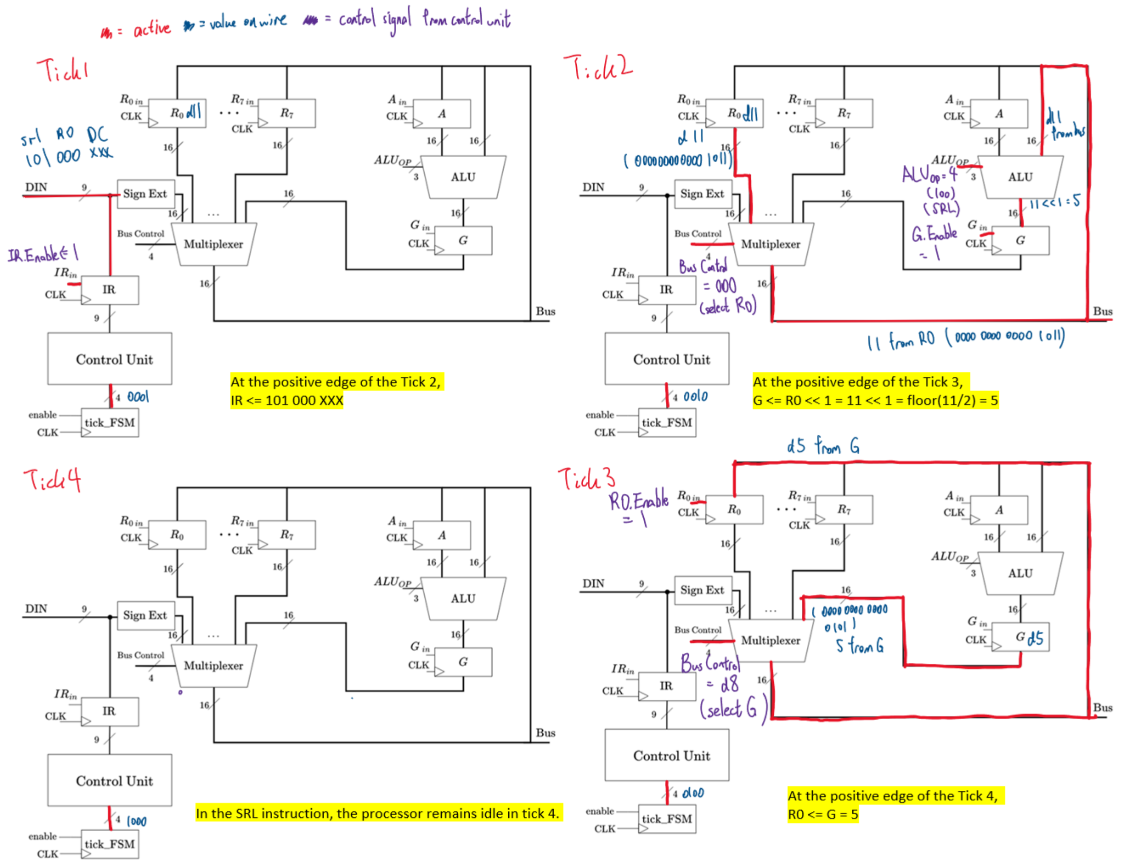
4.2 Instruction

The selected instructions are ***srl, 0*** and ***disp, 0***, performing R0 = 11 << 1 = 5. Display 5.

4.2.1 Before instruction

It is assumed that R0 contains 11. Note that idle ticks in ***disp, 0*** are not shown for simplicity.

4.3.2 *srl, 0* (Shift number in R0 rightwards by 1 bit)



= active = value on wire = control signal from control unit
Tick1
srl R0 DC
101 000 XXX
IR.Enable ← 1
0001
Tick2
d 11
(0000000000001011)
Bus Control = 000 (select R0)
ALU_OP = 4 (100) (SRL)
G.Enable = 1
d11 from bus
11<<1 = 5
11 from R0 (0000 0000 0000 1011)
0010
Tick3
R0.Enable = 1
d5 from G
(0000 0000 0000 0101)
5 from G
Bus Control = d8 (select G)
0100
Tick4
1000
R0, R7, A, G, IR, ALU, Multiplexer, Sign Ext, Control Unit, tick_FSM, Bus, DIN, CLK, enable

At the positive edge of the Tick 2,
IR <= 101 000 XXX

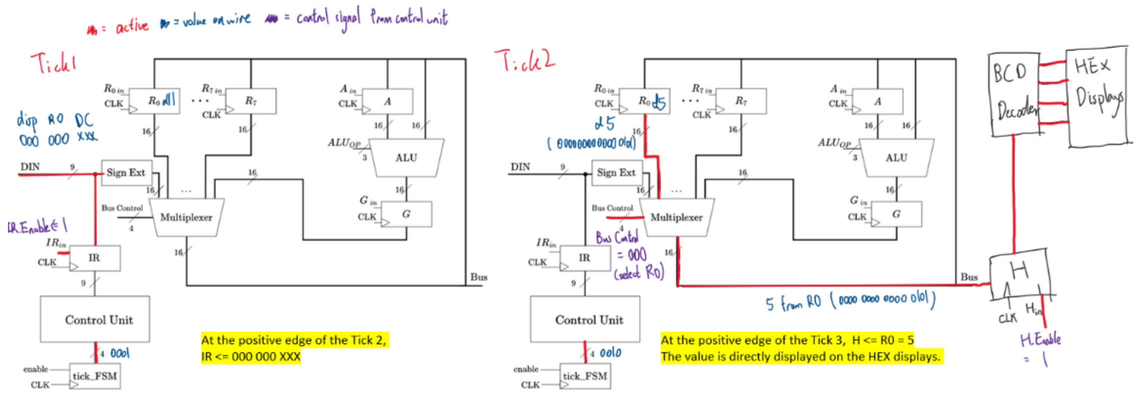
At the positive edge of the Tick 3,
G <= R0 << 1 = 11 << 1 = floor(11/2) = 5

In the SRL instruction, the processor remains idle in tick 4.

At the positive edge of the Tick 4,
R0 <= G = 5

Figure 3 - Datapath visualization for srl, R0

4.3.3 *disp, 0* (Display number in R0)



= active = value on wire = control signal from control unit
Tick1
disp R0 DC
000 000 XXX
IR.Enable ← 1
0001
Tick2
d5
(0000000000000101)
Bus Control = 000 (select R0)
5 from R0 (0000 0000 0000 0101)
0010
BCD Decoder
HEX Displays
H
CLK
H_in
H.Enable = 1

At the positive edge of the Tick 2,
IR <= 000 000 XXX

At the positive edge of the Tick 3, H <= R0 = 5
The value is directly displayed on the HEX displays.

Figure 4 - Datapath visualization for disp, 0

Section 5 Testing

5.1 Components

As seen in Appendix 7.3.1, tests cover the different types of inputs that can occur, such as negative, positive, 0 or some special input to the specific module. If failures were to occur, a message will be printed in the console with the test type, input failed for, expected output and the actual output. Otherwise, a message indicating test success is passed, and a summary of how many errors occurred is printed as test finishes.

The sign extender is tested against 4 different inputs, covering a positive number, a negative number, 0 and all 1s in binary. This covers all types of inputs that can occur and should be sufficient for this simple module.

The tick finite state machine is tested against all modes of operation. It is tested for when enable is not on where the tick should not be transitioning, when enable is on and a full 4 tick transition is observed, and when reset is on where the tick should reset back to initial state at every clock. This would cover all possible operations for this component.

The tests for the multiplexer cycles through all possible 9 inputs for selection, and the tests check for if the multiplexer has given correct output based on selection. It covers most input combination and should be sufficient to verify the operation.

The ALU is tested for each instruction (add, subtract, multiply, shift left and shift right) using a for loop for 5 pairs of input combinations, including 2 negative inputs, 2 positive inputs, 1 negative and 1 positive, 1 positive and 1 negative and both inputs being 0. This ensures that the ALU should operate normally under any type of pairs of input value under all instructions.

The two types of registers used – 9-bit and 16-bit – are tested for whether they write at clock edges. I assumed that if it writes any one type of input, it can handle all types of input, as it is simply taking the input and storing in register. They are also tested on whether reset occurs if the reset signal is asserted.

Note that BCD modules are not tested, as the validity of output is device dependent. For example, DE-10 displays are active low, but other devices may require active high outputs, which renders testing not feasible.

5.2 Processor

Each instruction is tested for various types of input combinations. At the end, complex tests where all 4 instructions are used is done to ensure that processor works with continuous operation. If errors occur, bus values at each tick in the instruction and register values before and after are printed to console. See Appendix 7.3.2 for an example of such prints. A descriptive name is given to each test, whereby it becomes easy to recognize which test failed. A summary message of how many tests failed is printed as test finishes.

Note that arguments for tasks are not passed by name, as this feature is not available in native Verilog.

Each instruction is tested with the same principle, where positive and negative inputs are tested. See Appendix 7.3.2 for the list of tests run. Passing values to din and running the instructions through the clock cycles are abstracted as tasks, making the testbench simplistic and easy to read. All tests are result based, where instructions are executed, and I verify whether the intended change in register value did occur.

Movi correctly moves positive and negative numbers. ***Add*** correctly adds positive and negative registers and ***addi*** correctly adds positive numbers and negative numbers to registers. ***Sub*** correctly subtracts positive and negative registers. It is also verified that all 4 can be used together to compute results such as ((r0+immi) +(r2-r3)). This is verified using random numbers generated by a hash function 20 times as seen in the code in Appendix 7.2.2 and prints in Appendix 7.3.2 and is sufficient for ensuring proper functioning of the simple processor.

5.3 Extended Processor

The testing principles remains identical to that of the simple processor testbench. In this extended processor, ***disp***, ***sll***, ***srl*** and ***mul*** are added to the instruction set. Since last testbench ensures proper functioning of the previous instructions, this testbench only involves testing the newly added instructions.

As seen in Appendix 7.3.3, the ***disp*** instruction is tested only once, as it is a simple case where a register value is moved to register H. The actual display of the value on BCD is considered to be separate from the processor itself, as it would be handled by external BCD decoders and displays. Thus, this testbench does not test that functionality.

The ***mul*** instruction is tested for all pairs of types of inputs, with positive multiplications, negative multiplications, opposite sign multiplications and 0 multiplications are all tested as seen in the prints in Appendix 7.3.3. This covers all situations of possible multiplications and can ensure the successful working of this instruction.

The ***sll*** instruction, in addition to the regular positive, negative and 0s, is also tested for overflow. A single bit number is left shifted 15 times in the test, and the processor correctly returns it back to 0 as bits over 16 are truncated. All tests in this section are passed, which verifies the processor ***sll*** operation with all input types.

The ***srl*** instruction is equivalent to floor dividing by 2. As a result, odd and even number behave differently, and are tested separately. The test case also covers where the input is negative and the logical shift, as opposed to an arithmetic shift, does not preserve the sign of the input. Regular tests with 0 input is also done. This ensures the successful operation of the ***srl*** instruction.

At the end, like the simple processor, a complex test involving all 4 instructions is conducted. This validates the processor operation with combinations of commands.

Section 6 Conclusion

In closing, this report encapsulates the intricacies in the design of a simple processor, showcasing the meticulous exploration of component functionality, resource utilization, timing analysis, integration, and the design of an effective testing regime. The processor combines and coordinates a range of individual components, such as an Arithmetic Logic Unit and a 16:1 Multiplexer, using a combinational control unit to perform calculations, reading, and writing to registers synchronously based on a singular system-wide clock. A comprehensive, intelligent testing is developed to ensure the correctness of the processor in execution of instructions sequentially. In the future, experiments can be conducted to measure and validate the performance and efficiency of this processor in real-world applications.

Section 7 Appendix

7.1 Verilog Modules

7.1.1 Components

```
/*
Monash University ECE2072: Assignment
This file contains Verilog code to implement individual components to be used in
    the CPU.

Please enter your name and student ID: 33114404

*/
//module components (in1, out0, out1, out2, out3, out4, clk); // for timing analysis
//  parameter IN_BIT = 16;
//  parameter OUT_BIT = 8;
//
//  input [IN_BIT-1:0] in1;
//  // input [IN_BIT-1:0] in2;
//  // input [3:0] in3;
//  output reg [OUT_BIT-1:0] out0, out1, out2, out3, out4;
//  input clk;
//
//  reg [IN_BIT-1:0] clocked_in1;
//  // reg [IN_BIT-1:0] clocked_in2;
//  // reg [3:0] clocked_in3;
//  wire [OUT_BIT-1:0] clocked_out [4:0];
//
//  always @(posedge clk) begin
//      clocked_in1 <= in1;
//      // clocked_in2 <= in2;
//      // clocked_in3 <= in3;
//      out0 <= clocked_out[0];
//      out1 <= clocked_out[1];
//      out2 <= clocked_out[2];
//      out3 <= clocked_out[3];
//      out4 <= clocked_out[4];
//  end
//  // sign_extend test (clocked_in1, clocked_out);
//  // tick_FSM tick_fsm(.rst(1'b0), .clk(clk), .tick(clocked_out), .enable(1'b1));
//  // multiplexer mux (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, clocked_in1, clocked_out);
//  // ALU alu(clocked_in1, clocked_in2, clocked_in3, clocked_out);
//  // register_n #(16) reg1 (clocked_in1, 1'b1, clk, clocked_out, 1'b0);
//  // register_n #(.N(9)) reg1 (clocked_in1, 1'b1, clk, clocked_out, 1'b0);
//  // bcd_out bcd_inst (.in(clocked_in1[3:0]), .hex(clocked_out));
//  bcd_chain_5 bcd_chain_inst (clocked_in1, clocked_out[0], clocked_out[1], clocked_out[2],
clocked_out[3], clocked_out[4]);
//endmodule




module sign_extend(in, ext);
    /*
     * This module sign extends the 9-bit Din to a 16-bit output.
     */
    // TODO: Declare inputs and outputs
    parameter IN = 9, OUT = 16;
    input [IN-1:0] in;
    output [OUT-1:0] ext;
    // TODO: implement logic
    assign ext = {{(OUT-IN){in[IN-1]}}, in};
endmodule



module tick_FSM(rst, clk, enable, tick);
    /*
     * This module implements a tick FSM that will be used to
     * control the actions of the control unit
     */

    // TODO: Declare inputs and outputs
    input clk, rst, enable;
    output reg [3:0] tick;
    parameter TICK1 = 4'b0001, TICK2 = 4'b0010, TICK3 = 4'b0100, TICK4 = 4'b1000;
    // TODO: implement FSM
    initial begin
        tick <= TICK1;
    end

    always @(posedge clk) begin
        if (rst) begin
            tick <= TICK1;
        end else if (enable) begin
            case (tick)
                TICK1: tick <= TICK2;
                TICK2: tick <= TICK3;
                TICK3: tick <= TICK4;
                TICK4: tick <= TICK1;
                default: tick <= TICK1;
            endcase
        end
    end
endmodule

module multiplexer(SignExtDin, R0, R1, R2, R3, R4, R5, R6, R7, G, sel, Bus);
    /*
     * This module takes 10 inputs and places the correct input onto the bus.
     */
    // TODO: Declare inputs and outputs
    input [15:0] SignExtDin, R0, R1, R2, R3, R4, R5, R6, R7, G;
    input [3:0] sel;
    output reg [15:0] Bus;
    parameter SEL_G = 4'd8, SEL_SIGNEXTDIN = 4'd9;
    // TODO: implement logic
    always @(*) begin
        case (sel)
            4'd0: Bus <= R0;
            4'd1: Bus <= R1;
            4'd2: Bus <= R2;
            4'd3: Bus <= R3;
            4'd4: Bus <= R4;
            4'd5: Bus <= R5;
            4'd6: Bus <= R6;
            4'd7: Bus <= R7;
            SEL_G: Bus <= G;
            SEL_SIGNEXTDIN: Bus <= SignExtDin;
            // default to give all 0
            default: Bus <= 16'd0;
        endcase
    end
```

```
endmodule

module ALU (input_a, input_b, alu_op, result);
    /*
     * This module implements the arithmetic logic unit of the processor.
     */
    // TODO: declare inputs and outputs
    input wire signed [15:0] input_a, input_b;
    input [2:0] alu_op;
    output reg signed [15:0] result;
    parameter ALU_ADD = 3'd0, ALU_SUB = 3'd1, ALU_MUL = 3'd2, ALU_SLL = 3'd3, ALU_SRL = 3'd4;
    // TODO: Implement ALU Logic:
    always @(*) begin
        case (alu_op)
            default: result <= 16'b0; // rest are don't cares
            ALU_ADD: result <= input_a + input_b;
            ALU_SUB: result <= input_a - input_b;
            ALU_MUL: result <= input_a * input_b;
            ALU_SLL: result <= input_b << 1;
            ALU_SRL: result <= input_b >> 1;
        endcase
    end
endmodule



module register_n(r_in, enable, clk, Q, rst);
    // To set parameter N during instantiation, you can use:
    // register_n #(.N(num_bits)) reg_IR(.....),
    // where num_bits is how many bits you want to set N to
    // and "..." is your usual input/output signals

    parameter N = 16;

    /*
     * This module implements registers that will be used in the processor.
     */
    // TODO: Declare inputs, outputs, and parameter:
    input [N-1:0] r_in;
    input enable, clk, rst;
    output reg [N-1:0] Q;

    initial begin
        Q = {N{1'b0}};
    end

    // TODO: Implement register logic:
    always @(posedge clk) begin
        Q <= Q; // default
        if (rst) begin
            Q <= {N{1'b0}};
        end else if (enable) begin
            Q <= r_in;
        end
    end
endmodule


module bcd_out(in, hex);
  input [3:0] in;
  output reg [6:0] hex;
  always @(*) begin
    case (in)
      4'd0: hex = 7'b1000000;
      4'd1: hex = 7'b1111001;
      4'd2: hex = 7'b0100100;
      4'd3: hex = 7'b0110000;
      4'd4: hex = 7'b0011001;
      4'd5: hex = 7'b0010010;
      4'd6: hex = 7'b0000010;
      4'd7: hex = 7'b1111000;
      4'd8: hex = 7'b0000000;
      4'd9: hex = 7'b0010000;
      default: hex = 7'b1111111;
    endcase
  end
endmodule

module bcd_chain_5(bit_input, hex_out0, hex_out1, hex_out2, hex_out3, hex_out4);
    input [15:0] bit_input;
    output [6:0] hex_out0, hex_out1, hex_out2, hex_out3, hex_out4;

    wire [15:0] quotient [3:0];
    wire [13:0] remainder [3:0];

    div div_inst4 (.denom(14'd10000), .numer(bit_input), .quotient(quotient[3]), .remain(remainder[3]));
    div div_inst3
(.denom(14'd1000), .numer(remainder[3]), .quotient(quotient[2]), .remain(remainder[2]));
    div div_inst2
(.denom(14'd100), .numer(remainder[2]), .quotient(quotient[1]), .remain(remainder[1]));
    div div_inst1 (.denom(14'd10), .numer(remainder[1]), .quotient(quotient[0]), .remain(remainder[0]));

    bcd_out bcd_inst4 (.in(quotient[3][3:0]), .hex(hex_out4));
    bcd_out bcd_inst3 (.in(quotient[2][3:0]), .hex(hex_out3));
    bcd_out bcd_inst2 (.in(quotient[1][3:0]), .hex(hex_out2));
    bcd_out bcd_inst1 (.in(quotient[0][3:0]), .hex(hex_out1));
    bcd_out bcd_inst0 (.in(remainder[0][3:0]), .hex(hex_out0));

endmodule
```

7.1.2 Processor

```
/*
Monash University ECE2072: Assignment
This file contains Verilog code to implement individual the CPU.

Please enter your student ID: 33114404

*/

module proc(SW, KEY, LEDR, HEX5);
    input [9:0] SW;
    input [0:0] KEY;
    output [9:0] LEDR;
    output [6:0] HEX5;

    wire [8:0] din;
    assign din = SW[8:0];
    wire [15:0] bus;
    wire rst = SW[9];
    wire clk = ~KEY[0];
    wire [15:0] R0, R1, R2, R3, R4, R5, R6, R7;
    wire [3:0] tick_FSM;
    assign LEDR[9:0] = bus[9:0];

    bcd_out bcd(tick_FSM, HEX5);

    simple_proc
proccessor(.clk(clk), .rst(rst), .din(din), .bus(bus), .R0(R0), .R1(R1), .R2(R2), .R3(R3), .R4(R4), .R5(
R5), .R6(R6), .R7(R7), .tick_FSM(tick_FSM));

endmodule

module simple_proc(clk, rst, din, bus, R0, R1, R2, R3, R4, R5, R6, R7, tick_FSM);
    // TODO: Declare inputs and outputs:
    input clk, rst;
    input [8:0] din;
    output [15:0] R0, R1, R2, R3, R4, R5, R6, R7; // for debugging purposes
    output [15:0] bus;
    output [3:0] tick_FSM;

    // instruction parameters
    parameter ADD = 3'd1, ADDI = 3'd2, SUB = 3'd3, MOVI = 3'd7;
    // multiplexer selection parameters
    parameter SEL_G = 4'd8, SEL_SIGNEXTDIN = 4'd9;
    // signal width parameters
    parameter DIN_WIDTH = 9, REG_WIDTH = 16;
    // ALU paramaters
    parameter ALU_ADD = 3'd0, ALU_SUB = 3'd1;
    // register parameters
    parameter IR_ID = 4'd0, A_ID = 4'd1, G_ID = 4'd2, H_ID = 4'd3;
    parameter NUM_GP_REG = 8; // 8 GP registers
    parameter NUM_SP_REG = 3; // 4 special registers (IR, A, G)


    // TODO: declare wires:
    wire [REG_WIDTH-1:0] r_out [NUM_GP_REG-1:0];
    wire [REG_WIDTH-1:0] A_in, G_in, A_out, G_out;
    wire [DIN_WIDTH-1:0] IR_out;
    wire [REG_WIDTH-1:0] sign_ext_din;
    wire [3:0] tick;
    assign tick_FSM = tick;
    wire [REG_WIDTH-1:0] alu_result, alu_input_a, alu_input_b;

    // parse the sections of the instruction given
    wire [2:0] op_code = IR_out[8:6];
    wire [2:0] rx = IR_out[5:3];
    wire [2:0] ry = IR_out[2:0];

    // declare control signals
    reg [NUM_GP_REG-1:0] gp_reg_write; // R0~7
    reg [NUM_SP_REG-1:0] sp_reg_write; // IR, A, G, H
    reg [3:0] bus_control; // Control what is being put on the bus from MUX
    reg [2:0] alu_op; // Control what operation the ALU is performing
    reg tick_enable = 1'b1; // Control whether the tick is enabled

    // instantiate sign extender:
    sign_extend sign_ext(.in(din), .ext(sign_ext_din));

    // TODO: instantiate General Purpose registers:
    genvar i;
    generate
        for (i = 0; i < NUM_GP_REG; i = i + 1) begin: reg_generate
```

```
            register_n #(.N(REG_WIDTH))
reg_inst(.r_in(bus), .enable(gp_reg_write[i]), .clk(clk), .Q(r_out[i]), .rst(rst));
        end
    endgenerate
    assign R0 = r_out[0];
    assign R1 = r_out[1];
    assign R2 = r_out[2];
    assign R3 = r_out[3];
    assign R4 = r_out[4];
    assign R5 = r_out[5];
    assign R6 = r_out[6];
    assign R7 = r_out[7];

    // instantiate special purpose IR
    register_n #(.N(DIN_WIDTH))
IR(.r_in(din), .enable(sp_reg_write[IR_ID]), .clk(clk), .Q(IR_out), .rst(rst));

    // TODO: instantiate Multiplexer:
    multiplexer
multiplexer(.sel(bus_control), .R0(r_out[0]), .R1(r_out[1]), .R2(r_out[2]), .R3(r_out[3]), .R4(r_out[4])
, .R5(r_out[5]), .R6(r_out[6]), .R7(r_out[7]), .G(G_out), .SignExtDin(sign_ext_din), .Bus(bus));

    // TODO: instantiate ALU:
    ALU alu(.input_a(alu_input_a), .input_b(alu_input_b), .alu_op(alu_op), .result(alu_result));
    // this allows us to not have to store inputs in registers
    assign alu_input_a = A_out;
    assign alu_input_b = bus;

    // instantiate ALU registers
    register_n A(.r_in(bus), .enable(sp_reg_write[A_ID]), .clk(clk), .Q(A_out), .rst(rst)); // for
storing intermediate results
    register_n G(.r_in(alu_result), .enable(sp_reg_write[G_ID]), .clk(clk), .Q(G_out), .rst(rst)); //
for storing the result of the ALU

    // TODO: instantiate tick counter:
    tick_FSM tick_fsm(.rst(rst), .clk(clk), .enable(tick_enable), .tick(tick));

    // TODO: define control unit:
    always @(*) begin
        // TODO: Turn off all control signals:
        // the separation of sp and gp reg writes prevent illegral overwrites of content in non gp reg
        sp_reg_write <= 0;
        gp_reg_write <= 0;
        alu_op <= 7; // does not correspond to any operation
        bus_control <= 15; // does not correspond to any selects
        // TODO: Turn on specific control signals based on current tick:
        case (tick)
            // tick 1
            4'b0001:
                begin
                    sp_reg_write[IR_ID] <= 1'b1; // enable IR
                end
            // tick 2
            4'b0010:
                begin
                    case (op_code)
                        ADD:
                            begin
                                bus_control <= rx; // select the 1st register to bus
                                sp_reg_write[A_ID] <= 1'b1; // write into A
                            end
                        ADDI:
```

```
                begin
                    bus_control <= SEL_SIGNEXTDIN; // select the immediate to bus
                    sp_reg_write[A_ID] <= 1'b1; // write into A
                end
            SUB:
                begin
                    bus_control <= rx; // select the immediate to bus
                    sp_reg_write[A_ID] <= 1'b1; // write into A
                end
            MOVI:
                begin
                    bus_control <= SEL_SIGNEXTDIN; // select the immediate to bus
                    gp_reg_write[rx] <= 1'b1; // write into rx
                end
                    default: begin
                                end
        endcase
    end
// tick 3
4'b0100:
    begin
        case (op_code)
            ADD:
                begin
                    bus_control <= ry; // select the 2nd register to bus
                    alu_op <= ALU_ADD; // add
                    sp_reg_write[G_ID] <= 1'b1; // write into G
                end
            ADDI:
                begin
                    bus_control <= rx; // select the rx to bus
                    alu_op <= ALU_ADD; // add
                    sp_reg_write[G_ID] <= 1'b1; // write into G
                end
            SUB:
                begin
                    bus_control <= ry; // select the ry to bus
                    alu_op <= ALU_SUB; // subtract
                    sp_reg_write[G_ID] <= 1'b1; // write into G
                end
            default: begin // no operation for any other instruction
                                end
        endcase
    end
// tick 4
4'b1000:
    begin
        case (op_code)
            ADD:
                begin
                    bus_control <= SEL_G; // select G to bus
                    gp_reg_write[rx] <= 1'b1; // write into rx
                end
            ADDI:
                begin
                    bus_control <= SEL_G; // select G to bus
                    gp_reg_write[rx] <= 1'b1; // write into rx
                end
            SUB:
                begin
                    bus_control <= SEL_G; // select G to bus
                    gp_reg_write[rx] <= 1'b1; // write into rx
```

```
                              end
                          default: begin // no operation for any other instruction
                                          end
                      endcase
                  end
                  default: begin
                              end
          endcase

      end

endmodule
```

7.1.3 Processor Extended

```
/*
Monash University ECE2072: Assignment
This file contains Verilog code to implement the extended version of CPU.

Please enter your student ID: 33114404


*/
 // TODO: Copy and paste your task 2 into here
/*
Monash University ECE2072: Assignment
This file contains Verilog code to implement individual the CPU.

Please enter your student ID: 33114404

*/

module proc_extension(SW, KEY, LEDR, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5);
    input [9:0] SW;
    input [0:0] KEY;
    output [9:0] LEDR;
    output [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;

    wire [8:0] din;
    assign din = SW[8:0];
    wire [15:0] bus;
    wire rst = SW[9];
    wire clk = ~KEY[0];
    wire [15:0] R0, R1, R2, R3, R4, R5, R6, R7, display;
    wire [3:0] tick_FSM;
    assign LEDR[9:0] = bus[9:0];

    bcd_out bcd(tick_FSM, HEX5);

    bcd_chain_5
chain(.bit_input(display), .hex_out4(HEX4), .hex_out3(HEX3), .hex_out2(HEX2), .hex_out1(HEX1), .hex_out0
(HEX0));

    simple_proc_ext
proc(.clk(clk), .rst(rst), .din(din), .bus(bus), .R0(R0), .R1(R1), .R2(R2), .R3(R3), .R4(R4), .R5(R5), .
R6(R6), .R7(R7), .tick_FSM(tick_FSM), .display(display));

endmodule

module simple_proc_ext(clk, rst, din, bus, R0, R1, R2, R3, R4, R5, R6, R7, tick_FSM, display);
    // TODO: Declare inputs and outputs:
```

```
    input clk, rst;
    input [8:0] din;
    output [15:0] R0, R1, R2, R3, R4, R5, R6, R7; // for debugging purposes
    output [15:0] bus;
    output [3:0] tick_FSM;
    output [15:0] display;

    // instruction parameters
    parameter DISP = 3'd0, ADD = 3'd1, ADDI = 3'd2, SUB = 3'd3, MUL = 3'd4, SRL = 3'd5, SLL = 3'd6, MOVI
= 3'd7;
    // multiplexer selection parameters
    parameter SEL_G = 4'd8, SEL_SIGNEXTDIN = 4'd9;
    // signal width parameters
    parameter DIN_WIDTH = 9, REG_WIDTH = 16;
    // ALU paramaters
    parameter ALU_ADD = 3'd0, ALU_SUB = 3'd1, ALU_MUL = 3'd2, ALU_SLL = 3'd3, ALU_SRL = 3'd4;
    // register parameters
    parameter IR_ID = 4'd0, A_ID = 4'd1, G_ID = 4'd2, H_ID = 4'd3;
    parameter NUM_GP_REG = 8; // 8 GP registers
    parameter NUM_SP_REG = 4; // 4 special registers (IR, A, G, H)


    // TODO: declare wires:
    wire [REG_WIDTH-1:0] r_out [NUM_GP_REG-1:0];
    wire [REG_WIDTH-1:0] A_in, G_in, A_out, G_out;
    wire [DIN_WIDTH-1:0] IR_out;
    wire [REG_WIDTH-1:0] sign_ext_din;
    wire [3:0] tick;
    assign tick_FSM = tick;
    wire [REG_WIDTH-1:0] alu_result, alu_input_a, alu_input_b;

    // parse the sections of the instruction given
    wire [2:0] op_code = IR_out[8:6];
    wire [2:0] rx = IR_out[5:3];
    wire [2:0] ry = IR_out[2:0];

    // declare control signals
    reg [NUM_GP_REG-1:0] gp_reg_write; // R0~7
    reg [NUM_SP_REG-1:0] sp_reg_write; // IR, A, G, H
    reg [3:0] bus_control; // Control what is being put on the bus from MUX
    reg [2:0] alu_op; // Control what operation the ALU is performing
    reg tick_enable = 1'b1; // Control whether the tick is enabled

    // instantiate sign extender:
    sign_extend sign_ext(.in(din), .ext(sign_ext_din));

    // TODO: instantiate General Purpose registers:
    genvar i;
    generate
        for (i = 0; i < NUM_GP_REG; i = i + 1) begin: reg_generate
            register_n #(.N(REG_WIDTH))
reg_inst(.r_in(bus), .enable(gp_reg_write[i]), .clk(clk), .Q(r_out[i]), .rst(rst));
        end
    endgenerate
    assign R0 = r_out[0];
    assign R1 = r_out[1];
    assign R2 = r_out[2];
    assign R3 = r_out[3];
    assign R4 = r_out[4];
    assign R5 = r_out[5];
    assign R6 = r_out[6];
    assign R7 = r_out[7];

    // instantiate special purpose IR
    register_n #(.N(DIN_WIDTH))
IR(.r_in(din), .enable(sp_reg_write[IR_ID]), .clk(clk), .Q(IR_out), .rst(rst));

    // TODO: instantiate Multiplexer:
    multiplexer
multiplexer(.sel(bus_control), .R0(r_out[0]), .R1(r_out[1]), .R2(r_out[2]), .R3(r_out[3]), .R4(r_out[4])
, .R5(r_out[5]), .R6(r_out[6]), .R7(r_out[7]), .G(G_out), .SignExtDin(sign_ext_din), .Bus(bus));

    // TODO: instantiate ALU:
    ALU alu(.input_a(alu_input_a), .input_b(alu_input_b), .alu_op(alu_op), .result(alu_result));
    // this allows us to not have to store inputs in registers
    assign alu_input_a = A_out;
    assign alu_input_b = bus;

    // instantiate ALU registers
    register_n A(.r_in(bus), .enable(sp_reg_write[A_ID]), .clk(clk), .Q(A_out), .rst(rst)); // for
storing intermediate results
    register_n G(.r_in(alu_result), .enable(sp_reg_write[G_ID]), .clk(clk), .Q(G_out), .rst(rst)); //
for storing the result of the ALU

    // TODO: instantiate tick counter:
    tick_FSM tick_fsm(.rst(rst), .clk(clk), .enable(tick_enable), .tick(tick));

    // instantiate HEX register (not used in this task)
    register_n H(.r_in(bus), .enable(sp_reg_write[H_ID]), .clk(clk), .Q(display), .rst(rst));


    // TODO: define control unit:
    always @(*) begin
        // TODO: Turn off all control signals:
        // the separation of sp and gp reg writes prevent illegral overwrites of content in non gp reg
        sp_reg_write <= 0;
        gp_reg_write <= 0;
        alu_op <= 7; // does not correspond to any operation
        bus_control <= 15; // does not correspond to any selects
        // TODO: Turn on specific control signals based on current tick:
        case (tick)
            // tick 1
            4'b0001:
                begin
                    sp_reg_write[IR_ID] <= 1'b1; // enable IR
                end
            // tick 2
            4'b0010:
                begin
                    case (op_code)
                        DISP:
                            begin
                                bus_control <= rx; // select the 1st register to bus
                                sp_reg_write[H_ID] <= 1'b1; // write into H
                            end
                        ADD:
                            begin
                                bus_control <= rx; // select the 1st register to bus
                                sp_reg_write[A_ID] <= 1'b1; // write into A
                            end
                        ADDI:
                            begin
                                bus_control <= SEL_SIGNEXTDIN; // select the immediate to bus
                                sp_reg_write[A_ID] <= 1'b1; // write into A
                            end
                        SUB:
                            begin
                                bus_control <= rx; // select the immediate to bus
                                sp_reg_write[A_ID] <= 1'b1; // write into A
                            end
                        MUL:
                            begin
                                bus_control <= rx; // select the 1st register to bus
                                sp_reg_write[A_ID] <= 1'b1; // write into A
                            end
                        SRL:
                            begin
                                bus_control <= rx; // select the 1st register to bus
                                alu_op <= ALU_SRL; // shift right
                                sp_reg_write[G_ID] <= 1'b1; // write into G
                            end
                        SLL:
                            begin
                                bus_control <= rx; // select the 1st register to bus
                                alu_op <= ALU_SLL; // shift right
                                sp_reg_write[G_ID] <= 1'b1; // write into G
                            end
                        MOVI:
                            begin
                                bus_control <= SEL_SIGNEXTDIN; // select the immediate to bus
                                gp_reg_write[rx] <= 1'b1; // write into rx
                            end
                    endcase
                end
            // tick 3
            4'b0100:
                begin
                    case (op_code)
                        ADD:
                            begin
                                bus_control <= ry; // select the 2nd register to bus
                                alu_op <= ALU_ADD; // add
                                sp_reg_write[G_ID] <= 1'b1; // write into G
                            end
                        ADDI:
                            begin
                                bus_control <= rx; // select the rx to bus
                                alu_op <= ALU_ADD; // add
                                sp_reg_write[G_ID] <= 1'b1; // write into G
                            end
                        SUB:
                            begin
                                bus_control <= ry; // select the ry to bus
                                alu_op <= ALU_SUB; // subtract
                                sp_reg_write[G_ID] <= 1'b1; // write into G
                            end
                        MUL:
                            begin
                                bus_control <= ry; // select the 2nd register to bus
                                alu_op <= ALU_MUL; // multiply
                                sp_reg_write[G_ID] <= 1'b1; // write into G
                            end
                        SRL:
                            begin
```

```
                                bus_control <= SEL_G; // select G to bus
                                gp_reg_write[rx] <= 1'b1; // write into rx
                            end
                        SLL:
                            begin
                                bus_control <= SEL_G; // select G to bus
                                gp_reg_write[rx] <= 1'b1; // write into rx
                            end
                        default: begin // no operation for any other instruction
                                    end
                    endcase
                end
            // tick 4
            4'b1000:
                begin
                    case (op_code)
                        ADD:
                            begin
                                bus_control <= SEL_G; // select G to bus
                                gp_reg_write[rx] <= 1'b1; // write into rx
                            end
                        ADDI:
                            begin
                                bus_control <= SEL_G; // select G to bus
                                gp_reg_write[rx] <= 1'b1; // write into rx
                            end
                        SUB:
                            begin
                                bus_control <= SEL_G; // select G to bus
                                gp_reg_write[rx] <= 1'b1; // write into rx
                            end
                        MUL:
                            begin
                                bus_control <= SEL_G; // select G to bus
                                gp_reg_write[rx] <= 1'b1; // write into rx
                            end
                        default: begin // no operation for any other instruction
                                    end
                    endcase
                end
            default: begin // no operation for any other instruction
                        end
        endcase

    end

endmodule
```

7.2 Verilog Testbenches

7.2.1 Components

```
`timescale 1ns/1ns
/*
Monash University ECE2072: Assignment
This file contains a Verilog test bench to test the correctness of the individual
    components used in the processor.

Please enter your student ID: 33114404
```

```
*/

// #1000 between each test
module components_tb;
    //sign extender
    integer sign_counter; // 4 cases are tested
    integer sign_errors;
    reg signed [15:0] true_sign_out;
    reg signed [8:0] sign_ext_in;
    wire signed [15:0] sign_ext_out;
    sign_extend sign_ext(.in(sign_ext_in), .ext(sign_ext_out));

    // for conclusion
    integer total_error;

    // tick FSM (share clock with register)
    integer tick_counter;
    integer tick_errors;
    reg clk, rst, enable;
    reg [3:0] right_tick;
    wire [3:0] tick;
    tick_FSM tick_fsm(.rst(rst), .clk(clk), .enable(enable), .tick(tick));

    // multiplexer
    reg [3:0] mux_counter;
    integer mux_errors;
    reg [3:0] mux_sel;
    wire [15:0] mux_out;
    multiplexer
mux(.SignExtDin(16'd9), .R0(16'd0), .R1(16'd1), .R2(16'd2), .R3(16'd3), .R4(16'd4), .R5(16'd5), .R6(16'd
6), .R7(16'd7), .G(16'd8), .sel(mux_sel), .Bus(mux_out));

    // ALU
    integer alu_counter;
    integer alu_errors;
    reg signed [15:0] alu_a, alu_b;
    reg [2:0] alu_op, op_iter;
    reg signed [15:0] true_alu_out;
    wire signed [15:0] alu_result;
    ALU alu(.input_a(alu_a), .input_b(alu_b), .alu_op(alu_op), .result(alu_result));

    // register
    integer reg16_counter;
    integer reg9_counter;
    integer reg_error;
    reg [0:0] reg_clk;
    reg [15:0] reg16_in;
    wire [15:0] reg16_out;
    reg [0:0]reg16_enable;
    reg [0:0] reg16_rst;
    register_n #(.N(16))
reg16(.r_in(reg16_in), .Q(reg16_out), .enable(reg16_enable), .rst(reg16_rst), .clk(reg_clk));
    reg [8:0] reg9_in;
    wire [8:0] reg9_out;
    reg [0:0] reg9_enable;
    reg [0:0] reg9_rst;
    register_n #(.N(9))
reg9(.r_in(reg9_in), .Q(reg9_out), .enable(reg9_enable), .rst(reg9_rst), .clk(reg_clk));

    task sign_ext_test;
        begin
            true_sign_out = sign_ext_in;
            if (true_sign_out != sign_ext_out) begin
                sign_errors = sign_errors + 1;
                $display("Sign extender test failed for input %b, expected %b, got %b", sign_ext_in,
true_sign_out, sign_ext_out);
            end
        end
    endtask

    task alu_op_test;
        begin
            for (op_iter=3'b000; op_iter<=3'b100; op_iter=op_iter+1) begin
                alu_op = op_iter[2:0];
                #10;
                case(alu_op)
                    3'b000: true_alu_out = alu_a + alu_b;
                    3'b001: true_alu_out = alu_a - alu_b;
                    3'b010: true_alu_out = alu_a * alu_b;
                    3'b011: true_alu_out = alu_b << 1;
                    3'b100: true_alu_out = alu_b >> 1;
                    default: true_alu_out = 16'd0;
                endcase
                if (true_alu_out != alu_result) begin
                    alu_errors = alu_errors + 1;
                    $display("ALU test failed for input pair (%d, %d), op_code %b, expected %d, got %d",
alu_a, alu_b, alu_op, true_alu_out, alu_result);
                end
            end
        end
    endtask

    // testing sign extender
    always begin
        total_error = 0;

        $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
        $display("Testing sign extender");
        sign_errors = 0;
    // test sign extender, 4 cases, 1 positive, 1 negative, all 0 & all 1
        for (sign_counter=0; sign_counter<4; sign_counter=sign_counter+1) begin
            case(sign_counter)
                2'd0:
                    sign_ext_in = 9'sd0;
                2'd1:
                    sign_ext_in = 9'sb111111111;
                2'd2:
                    sign_ext_in = -9'sd59;
                2'd3:
                    sign_ext_in = 9'sd38;
            endcase
            sign_ext_test();
        end
        if (sign_errors == 0) begin
            $display("Sign extender test passed");
        end
        else begin
            $display("Sign extender test failed with %d errors", sign_errors);
        end
        $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

        total_error = total_error + sign_errors;

        // test tick_FSM
```

```
$display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
$display("Testing tick_FSM");
tick_errors = 0;
// test tick FSM 10 ticks including some resets and enable
clk = 0;
enable = 0;
rst = 0;
right_tick = 4'b0001;
// 2 ticks with no enable, expect 0001
for(tick_counter=0; tick_counter<2; tick_counter=tick_counter+1) begin
    #10
    clk = 1; // posedge
    #10
    if (tick != right_tick) begin
        $display("Tick FSM enable test failed, expected %b, got %b", right_tick, tick);
        tick_errors = tick_errors + 1;
    end
    clk = 0;
end
#10
clk = 0;
enable = 1;
rst = 0;
right_tick = 4'b0001;
// 2 ticks with no enable, expect 0001
for(tick_counter=0; tick_counter<6; tick_counter=tick_counter+1) begin
    #10
    clk = 1; // posedge
    // update tick by logic
    case (right_tick)
            4'b0001: right_tick = 4'b0010;
            4'b0010: right_tick = 4'b0100;
            4'b0100: right_tick = 4'b1000;
            4'b1000: right_tick = 4'b0001;
        endcase
    #10
    if (tick != right_tick) begin
        $display("Tick FSM tick test failed, expected %b, got %b", right_tick, tick);
        tick_errors = tick_errors + 1;
    end
    clk = 0;
end
#10
clk = 0;
enable = 1;
rst = 1;
right_tick = 4'b0001;
// 2 ticks with rst, expect 0001
for(tick_counter=0; tick_counter<2; tick_counter=tick_counter+1) begin
    #10
    clk = 1; // posedge
    #10
    if (tick != right_tick) begin
        $display("Tick FSM reset test failed, expected %b, got %b", right_tick, tick);
        tick_errors = tick_errors + 1;
    end
    clk = 0;
end
if (tick_errors == 0) begin
    $display("Tick FSM test passed");
end
else begin
```

```
            $display("Tick FSM test failed with %d errors", tick_errors);
        end
        $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

        total_error = total_error + tick_errors;

    // test multiplexer
        $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
        $display("Testing multiplexer");
        mux_errors = 0;
        for(mux_counter=4'd0; mux_counter<=4'd9; mux_counter=mux_counter+1) begin
            mux_sel = mux_counter[3:0];
            #10;
            if (mux_out != mux_counter) begin
                $display("Multiplexer test failed, expected val in R%d, got %d (R8 is G, R9 is din)",
mux_counter, mux_out);
                mux_errors = mux_errors + 1;
            end
        end
        if (mux_errors == 0) begin
            $display("Multiplexer test passed");
        end
        else begin
            $display("Multiplexer test failed with %d errors", mux_errors);
        end
        $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

        total_error = total_error + mux_errors;

    // test ALU
        $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
        $display("Testing ALU");
        alu_errors = 0;
        // test ALU add
        alu_op = 3'b000;
        alu_a = 16'd0;
        alu_b = 16'd0;
        alu_counter = 0;
        for (alu_counter=0; alu_counter<5; alu_counter=alu_counter+1) begin
            // test 5 cases, ++, --, +-, -+, 0
            case(alu_counter)
                4'd0:
                    begin
                        alu_a = 16'sd11;
                        alu_b = 16'sd227;
                    end
                4'd1:
                    begin
                        alu_a = -16'sd29;
                        alu_b = -16'sd431;
                    end
                4'd2:
                    begin
                        alu_a = 16'sd123;
                        alu_b = -16'sd321;
                    end
                4'd3:
                    begin
                        alu_a = -16'sd213;
                        alu_b = 16'sd445;
                    end
                4'd4:
```

```
                begin
                    alu_a = 16'sd0;
                    alu_b = 16'sd0;
                end

            endcase
            #10
            alu_op_test();
        end
        if (alu_errors == 0) begin
            $display("ALU test passed");
        end
        else begin
            $display("ALU failed with %d errors", alu_errors);
        end
        $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

        total_error = total_error + alu_errors;

    // test register
        $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
        $display("Testing register");
        // test 16 bit register
        reg_error = 0;
        reg16_rst = 1'b0;
        reg16_enable = 1'b1;
        reg16_in = 16'd0;
        reg_clk = 0;
        for (reg16_counter=0; reg16_counter<5; reg16_counter=reg16_counter+1) begin
            case(reg16_counter)
                0:
                    reg16_in = 16'd0;
                1:
                    reg16_in = 16'd1;
                2:
                    reg16_in = 16'd2;
                3:
                    reg16_in = 16'd3;
                4:
                    reg16_in = 16'd4;
            endcase
            reg16_enable = 1'b1;
            #10;
            reg_clk = 1;
            #10;
            if (reg16_out != reg16_in) begin
                reg_error = reg_error + 1;
                $display("16 bit register write test failed, expected %d, got %d", reg16_in, reg16_out);
            end
            #10
            reg_clk = 0;
            reg16_rst = 1'b1;
            reg16_enable = 1'b0;
            #10
            reg_clk = 1;
            #10;
            if (reg16_out != 16'd0) begin
                reg_error = reg_error + 1;
                $display("16 bit register reset test failed, expected 0, got %d", reg16_out);
            end
            reg_clk = 0;
            reg16_rst = 1'b0;
```

```verilog
end
// test 9 bit register
reg9_rst = 1'b0;
reg9_enable = 1'b0;
reg9_in = 9'd0;
reg_clk = 0;
for (reg9_counter=0; reg9_counter<5; reg9_counter=reg9_counter+1) begin
    case(reg9_counter)
        0:
            reg9_in = 9'd0;
        1:
            reg9_in = 9'd1;
        2:
            reg9_in = 9'd2;
        3:
            reg9_in = 9'd3;
        4:
            reg9_in = 9'd4;

    endcase
    reg9_enable = 1'b1;
    #10
    reg_clk = 1;
    #10
    if (reg9_out != reg9_in) begin
        reg_error = reg_error + 1;
        $display("9 bit register write test failed, expected %d, got %d", reg9_in, reg9_out);
    end
    #10
    reg_clk = 0;
    reg9_rst = 1'b1;
    reg9_enable = 1'b0;

    #10
    reg_clk = 1;
    #10;
    if (reg9_out != 9'd0) begin
        reg_error = reg_error + 1;
        $display("9 bit register reset test failed, expected 0, got %d", reg9_out);
    end
    reg_clk = 0;
    reg9_rst = 1'b0;
end
if (reg_error == 0) begin
    $display("Register test passed");
end
else begin
    $display("Register test failed with %d errors", reg_error);
end
$display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

total_error = total_error + reg_error;
$display("The tests have ended");
if (total_error == 0) begin
    $display("All tests passed");
end
else begin
    $display("Some tests failed with %d errors", total_error);
end
```

```
        $stop;
    end

endmodule
```

7.2.2 Processor

```
`timescale 1ns/1ns
/*
Monash University ECE2072: Assignment
This file contains a Verilog test bench to test the correctness of the processor.

Please enter your student ID: 33114404
Note that I cannot pass arguments to tasks by name, so I have to use pass by position.
*/
module proc_tb;
    // parameter definitions
    parameter ADD = 3'd1, ADDI = 3'd2, SUB = 3'd3, MOVI = 3'd7;
    // constants for clarity
    parameter DONT_CARE_REG = 3'bxxx, DONT_CARE_IMMI = 9'bxxxxxxxxx, TRUE = 1'b1, FALSE = 1'b0;

    // error counting
    integer total_error;

    // mostly fixed inputs
    reg rst;
    reg [0:0] clk;
    // input
    reg [8:0] din;
    // outputs
    wire [15:0] reg_values [7:0]; // R0~7
    wire [15:0] bus;
    wire [3:0] tick_FSM;
    // processor
    simple_proc proc(.clk(clk), .rst(rst), .din(din), .bus(bus),
                    .R0(reg_values[0]), .R1(reg_values[1]), .R2(reg_values[2]),
                    .R3(reg_values[3]), .R4(reg_values[4]), .R5(reg_values[5]),
                    .R6(reg_values[6]), .R7(reg_values[7]), .tick_FSM(tick_FSM));

    reg signed [8:0] rand_immi [3:0];
    integer complicated_test_count;


    // init
    initial begin
        clk <= 0;
        rst <= 1;
        din <= 0;
        total_error <= 0;
    end

    // clock at 10ns, note the initial #5 delay to start (5, 10, 15 posedges)
    always begin
        #5
        if (clk == TRUE) begin
            clk = FALSE;
        end else begin
            clk = TRUE;
        end
```

```
end

// testbench params
reg [15:0] reg_values_before [7:0];
reg [15:0] bus_values_by_tick [3:0];
integer i;
initial begin
    for (i = 0; i < 4; i = i + 1) begin
        bus_values_by_tick[i] = 16'd0;
    end
    for (i = 0; i <= 7; i = i + 1) begin
        reg_values_before[i] = 16'd0;
    end
end

parameter A = 1664525;
parameter C = 1013904223;
integer rand_num = 108;
task gen_rand_immi;
    begin
        for (i = 0; i <= 3; i = i + 1) begin
            rand_num = A * rand_num + C; // mod 2^9
            rand_immi[i] = rand_num[8:0];
        end
    end
endtask

// print bus values during the last instruction execution
task print_bus_values;
    begin
        $display("tick    bus_value");
        // replce eith for loop
        for (i = 0; i < 4; i = i + 1) begin
            $display("%01d   %05d", i, bus_values_by_tick[i]);
        end
    end
endtask

// print last saved register value and current register value
task print_register_values;
    begin
        $display("before instruction:");
        $display("register   value");
        for (i = 0; i <= 7; i = i + 1) begin
            $display("r%01d     %05d", i, reg_values_before[i]);
        end
        $display("after instruction:");
        $display("register   value");
        for (i = 0; i <= 7; i = i + 1) begin
            $display("r%01d     %05d", i, reg_values[i]);
        end
    end
endtask

// basic instruction template, #100 delay
task instruction;
    input [2:0] op_code;
    input [2:0] rx;
    input [2:0] ry;
    input [8:0] immediate;
    input [0:0] has_immediate;
    begin
```

```
            for (i = 0; i <= 7; i = i + 1) begin
                reg_values_before[i] = reg_values[i];
            end

            // note that tick 1 only has #5 delay, to match the additional delays at the end of each
instruction
            // tick 1 starts
            din = {op_code, rx, ry};
            bus_values_by_tick[0] = bus;
            #5;
            // tick 2
            #5;
            if (has_immediate == TRUE) din = immediate;
            bus_values_by_tick[1] = bus;
            #5;

            // no input change ever occurs in tick 3 and 4
            // tick 3
            #5;
            bus_values_by_tick[2] = bus;
            #5;
            // tick 4
            #5;
            bus_values_by_tick[3] = bus;
            #5;

            // next round tick 1 to make sure update goes through
            din = 0; // clear input so that last instruction does not affect next instruction
            #5;
        end
    endtask

    task add;
        input [2:0] rx, ry;
        begin
            instruction(ADD, rx, ry, DONT_CARE_IMMI, FALSE);
        end
    endtask

    task addi;
        input [2:0] rx;
        input [8:0] immediate;
        begin
            instruction(ADDI, rx, DONT_CARE_REG, immediate, TRUE);
        end
    endtask

    task sub;
        input [2:0] rx, ry;
        begin
            instruction(SUB, rx, ry, DONT_CARE_IMMI, FALSE);
        end
    endtask

    task movi;
        input [2:0] rx;
        input [8:0] immediate;
        instruction(MOVI, rx, DONT_CARE_REG, immediate, TRUE);
    endtask

    reg [0:0] success;
    integer error_count;
    task test_error;
        input [2:0] reg_modified;
        input [15:0] expected_value;
        begin
            if (reg_values[reg_modified] != expected_value) begin
                error_count = error_count + 1;
                print_debug_values();
                success = FALSE;
                total_error = total_error + 1;
            end else begin
                success = TRUE;
            end
        end
    endtask

    task print_debug_values;
        begin
            print_bus_values();
            print_register_values();
        end
    endtask

    /**********************
        ACTUAL TESTING
    **********************/

    always begin
        #2;
          total_error = 0;
        $display("Bus values during execution and register values before and after will be printed if
error occurs. \n Exmaple below: ");
        print_bus_values();
        print_register_values();
        #3; // match the initial delay

        #2; // dummy delay for instruction to start
        rst = FALSE; // turn reset off to start
        #3

        // test movi
        $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
        $display("Tests for %s", "movi");
        error_count = 0;
        /*
        Test movi instruction with positive numbers
        r0 = 10
        */
        movi(3'd0, 9'd10);
        test_error(4'd0, 16'd10);
        if (success!=TRUE) begin
            $display("Test \<%s\> failed. %s", "movi-pos", "r0 should be 10");
        end else begin
            $display("Test \<%s\> succeeded. ", "movi-pos");
        end

        /*
        Test movi instruction with negative numbers
        r1 = -10
        */
        movi(3'd1, -9'sd10);
        test_error(4'd1, -16'sd10);
        if (success!=TRUE) begin
```

```
    $display("Test \<%s\> failed. %s", "movi-neg", "r1 should be -10");
end else begin
    $display("Test \<%s\> succeeded. ", "movi-neg");
end
$display("Test movi completed with %d errors", error_count);
$display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

// test add
$display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
$display("Tests for %s", "add");
error_count = 0;
/*
Test add instruction with positive numbers
r2 = 10
r3 = 20
r2 = r2+r3
*/
movi(3'd2, 9'd10);
movi(3'd3, 9'd20);
add(3'd2, 3'd3);
test_error(4'd2, 16'd30);
if (success!=TRUE) begin
    $display("Test \<%s\> failed. %s", "add-pos", "r2 should be 30");
end else begin
    $display("Test \<%s\> succeeded. ", "add-pos");
end

/*
Test add instruction with negative numbers
r4 = -108
r5 = -20
r4 = r4+r5
*/
movi(3'd4, -9'sd108);
movi(3'd5, -9'sd20);
add(3'd4, 3'd5);
test_error(4'd4, -16'sd128);
if (success!=TRUE) begin
    $display("Test \<%s\> failed. %s", "add-neg", "r2 should be -88");
end else begin
    $display("Test \<%s\> succeeded. ", "add-neg");
end
$display("Test add completed with %d errors", error_count);
$display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

// test addi
$display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
$display("Tests for %s", "addi");
error_count = 0;
/*
Test addi instruction with positive numbers
r6 = 10
r6 = r6+20
*/
movi(3'd6, 9'd10);
addi(3'd6, 9'd20);
test_error(4'd6, 16'd30);
if (success!=TRUE) begin
    $display("Test \<%s\> failed. %s", "addi-pos", "r6 should be 30");
end else begin
    $display("Test \<%s\> succeeded. ", "addi-pos");
end
```

```
/*
Test addi instruction with negative numbers
r7 = 10
r7 = r7-20
*/
movi(3'd7, 9'd10);
addi(3'd7, -9'sd20);
test_error(4'd7, -16'sd10);
if (success!=TRUE) begin
    $display("Test \<%s\> failed. %s", "addi-neg", "r7 should be -10");
end else begin
    $display("Test \<%s\> succeeded. ", "addi-neg");
end
$display("Test addi completed with %d errors", error_count);
$display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

// test sub
$display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
$display("Tests for %s", "sub");
error_count = 0;
/*
Test sub instruction with positive numbers
r0 = 10
r1 = 20
r0 = r0-r1
*/
movi(3'd0, 9'd10);
movi(3'd1, 9'd20);
sub(3'd0, 3'd1);
test_error(4'd0, -16'sd10);
if (success!=TRUE) begin
    $display("Test \<%s\> failed. %s", "sub-pos", "r0 should be -10");
end else begin
    $display("Test \<%s\> succeeded. ", "sub-pos");
end

/*
Test sub instruction with negative numbers
r2 = -10
r3 = -20
r2 = r2-r3
*/
movi(3'd2, -9'sd10);
movi(3'd3, -9'sd20);
sub(3'd2, 3'd3);
test_error(4'd2, 16'sd10);
if (success!=TRUE) begin
    $display("Test \<%s\> failed. %s", "sub-neg", "r2 should be 10");
end else begin
    $display("Test \<%s\> succeeded. ", "sub-neg");
end
$display("Test sub completed with %d errors", error_count);
$display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

// test complicated
$display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
$display("Tests for %s", "complicated");
error_count = 0;
complicated_test_count = 0;
/*
Test complicated instruction (use all 4 available) 20 times
```

```
        Do         ((r0-immi)<-addi+(r2-r3)<-sub)<-add
        */
        for (complicated_test_count=1; complicated_test_count<20;
complicated_test_count=complicated_test_count+1) begin
            gen_rand_immi();
            movi(3'd0, rand_immi[0]);
            movi(3'd2, rand_immi[1]);
            movi(3'd3, rand_immi[2]);
            // see if a movi is done, if not, then errors after may not be useful
            test_error(4'd0, rand_immi[0]);
            if (success!=TRUE) begin
                $display("Test \<%s\> failed. %s%05d", "complicated-1-interim_movi", "r0 should be ",
rand_immi[0]);
            end else begin
                $display("Test \<%s\> succeeded. ", "complicated-1-interim_movi");
            end
            // r0 = r0-rand
            addi(3'd0, rand_immi[3]);
            // r2 = r2-r3
            sub(3'd2, 3'd3);
            // r0 = r0+r2
            add(3'd0, 3'd2);
            test_error(4'd0, (rand_immi[0]+rand_immi[3])+(rand_immi[1]-rand_immi[2]));
            if (success!=TRUE) begin
                $display("Test \<%s%02d\> failed. %s %05d", "complicated-", complicated_test_count, "r0
should be ", (rand_immi[0]+rand_immi[3])+(rand_immi[1]-rand_immi[2]));
            end else begin
                $display("Test \<%s%02d\> succeeded. ", "complicated-", complicated_test_count);
            end
        end
        $display("Test complicated completed with %d errors", error_count);
        $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

        $display("Tests have all finished. ");
        if (total_error == 0) begin
            $display("All tests passed!");
        end else begin
            $display("There are %d errors in total.", total_error);
        end

        $stop;
    end


endmodule
```

7.2.3 Processor Extended

```
`timescale 1ns/1ns
/*
Monash University ECE2072: Assignment
This file contains a Verilog test bench to test the correctness of the processor.

Please enter your student ID: 33114404
Note that I cannot pass arguments to tasks by name, so I have to use pass by position.
*/
module proc_tb;
    // parameter definitions
    parameter ADD = 3'd1, ADDI = 3'd2, SUB = 3'd3, MOVI = 3'd7;
    // constants for clarity
    parameter DONT_CARE_REG = 3'bxxx, DONT_CARE_IMMI = 9'bxxxxxxxxx, TRUE = 1'b1, FALSE = 1'b0;
```

```
// error counting
integer total_error;

// mostly fixed inputs
reg rst;
reg [0:0] clk;
// input
reg [8:0] din;
// outputs
wire [15:0] reg_values [7:0]; // R0~7
wire [15:0] bus;
wire [3:0] tick_FSM;
// processor
simple_proc proc(.clk(clk), .rst(rst), .din(din), .bus(bus),
                .R0(reg_values[0]), .R1(reg_values[1]), .R2(reg_values[2]),
                .R3(reg_values[3]), .R4(reg_values[4]), .R5(reg_values[5]),
                .R6(reg_values[6]), .R7(reg_values[7]), .tick_FSM(tick_FSM));

reg signed [8:0] rand_immi [3:0];
integer complicated_test_count;


// init
initial begin
    clk <= 0;
    rst <= 1;
    din <= 0;
    total_error <= 0;
end

// clock at 10ns, note the initial #5 delay to start (5, 10, 15 posedges)
always begin
    #5
    if (clk == TRUE) begin
        clk = FALSE;
    end else begin
        clk = TRUE;
    end
end

// testbench params
reg [15:0] reg_values_before [7:0];
reg [15:0] bus_values_by_tick [3:0];
integer i;
initial begin
    for (i = 0; i < 4; i = i + 1) begin
        bus_values_by_tick[i] = 16'd0;
    end
    for (i = 0; i <= 7; i = i + 1) begin
        reg_values_before[i] = 16'd0;
    end
end

parameter A = 1664525;
parameter C = 1013904223;
integer rand_num = 108;
task gen_rand_immi;
    begin
        for (i = 0; i <= 3; i = i + 1) begin
            rand_num = A * rand_num + C; // mod 2^9
```

```
                rand_immi[i] = rand_num[8:0];
            end
        end
    endtask

    // print bus values during the last instruction execution
    task print_bus_values;
        begin
            $display("tick    bus_value");
            // replce eith for loop
            for (i = 0; i < 4; i = i + 1) begin
                $display("%01d   %05d", i, bus_values_by_tick[i]);
            end
        end
    endtask

    // print last saved register value and current register value
    task print_register_values;
        begin
            $display("before instruction:");
            $display("register   value");
            for (i = 0; i <= 7; i = i + 1) begin
                $display("r%01d     %05d", i, reg_values_before[i]);
            end
            $display("after instruction:");
            $display("register   value");
            for (i = 0; i <= 7; i = i + 1) begin
                $display("r%01d     %05d", i, reg_values[i]);
            end
        end
    endtask

    // basic instruction template, #100 delay
    task instruction;
        input [2:0] op_code;
        input [2:0] rx;
        input [2:0] ry;
        input [8:0] immediate;
        input [0:0] has_immediate;
        begin
            for (i = 0; i <= 7; i = i + 1) begin
                reg_values_before[i] = reg_values[i];
            end

            // note that tick 1 only has #5 delay, to match the additional delays at the end of each
instruction
            // tick 1 starts
            din = {op_code, rx, ry};
            bus_values_by_tick[0] = bus;
            #5;
            // tick 2
            #5;
            if (has_immediate == TRUE) din = immediate;
            bus_values_by_tick[1] = bus;
            #5;

            // no input change ever occurs in tick 3 and 4
            // tick 3
            #5;
            bus_values_by_tick[2] = bus;
            #5;
            // tick 4
            #5;
            bus_values_by_tick[3] = bus;
            #5;

            // next round tick 1 to make sure update goes through
            din = 0; // clear input so that last instruction does not affect next instruction
            #5;
        end
    endtask

    task add;
        input [2:0] rx, ry;
        begin
            instruction(ADD, rx, ry, DONT_CARE_IMMI, FALSE);
        end
    endtask

    task addi;
        input [2:0] rx;
        input [8:0] immediate;
        begin
            instruction(ADDI, rx, DONT_CARE_REG, immediate, TRUE);
        end
    endtask

    task sub;
        input [2:0] rx, ry;
        begin
            instruction(SUB, rx, ry, DONT_CARE_IMMI, FALSE);
        end
    endtask

    task movi;
        input [2:0] rx;
        input [8:0] immediate;
        instruction(MOVI, rx, DONT_CARE_REG, immediate, TRUE);
    endtask

    reg [0:0] success;
    integer error_count;
    task test_error;
        input [2:0] reg_modified;
        input [15:0] expected_value;
        begin
            if (reg_values[reg_modified] != expected_value) begin
                error_count = error_count + 1;
                print_debug_values();
                success = FALSE;
                total_error = total_error + 1;
            end else begin
                success = TRUE;
            end
        end
    endtask

    task print_debug_values;
        begin
            print_bus_values();
            print_register_values();
        end
    endtask
```

```
    /**********************
        ACTUAL TESTING
    **********************/

    always begin
        #2;
          total_error = 0;
        $display("Bus values during execution and register values before and after will be printed if
error occurs. \n Exmaple below: ");
        print_bus_values();
        print_register_values();
        #3; // match the initial delay

        #2; // dummy delay for instruction to start
        rst = FALSE; // turn reset off to start
        #3

        // test movi
        $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
        $display("Tests for %s", "movi");
        error_count = 0;
        /*
        Test movi instruction with positive numbers
        r0 = 10
        */
        movi(3'd0, 9'd10);
        test_error(4'd0, 16'd10);
        if (success!=TRUE) begin
            $display("Test \<%s\> failed. %s", "movi-pos", "r0 should be 10");
        end else begin
            $display("Test \<%s\> succeeded. ", "movi-pos");
        end

        /*
        Test movi instruction with negative numbers
        r1 = -10
        */
        movi(3'd1, -9'sd10);
        test_error(4'd1, -16'sd10);
        if (success!=TRUE) begin
            $display("Test \<%s\> failed. %s", "movi-neg", "r1 should be -10");
        end else begin
            $display("Test \<%s\> succeeded. ", "movi-neg");
        end
        $display("Test movi completed with %d errors", error_count);
        $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

        // test add
        $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
        $display("Tests for %s", "add");
        error_count = 0;
        /*
        Test add instruction with positive numbers
        r2 = 10
        r3 = 20
        r2 = r2+r3
        */
        movi(3'd2, 9'd10);
        movi(3'd3, 9'd20);
        add(3'd2, 3'd3);
        test_error(4'd2, 16'd30);
        if (success!=TRUE) begin
```

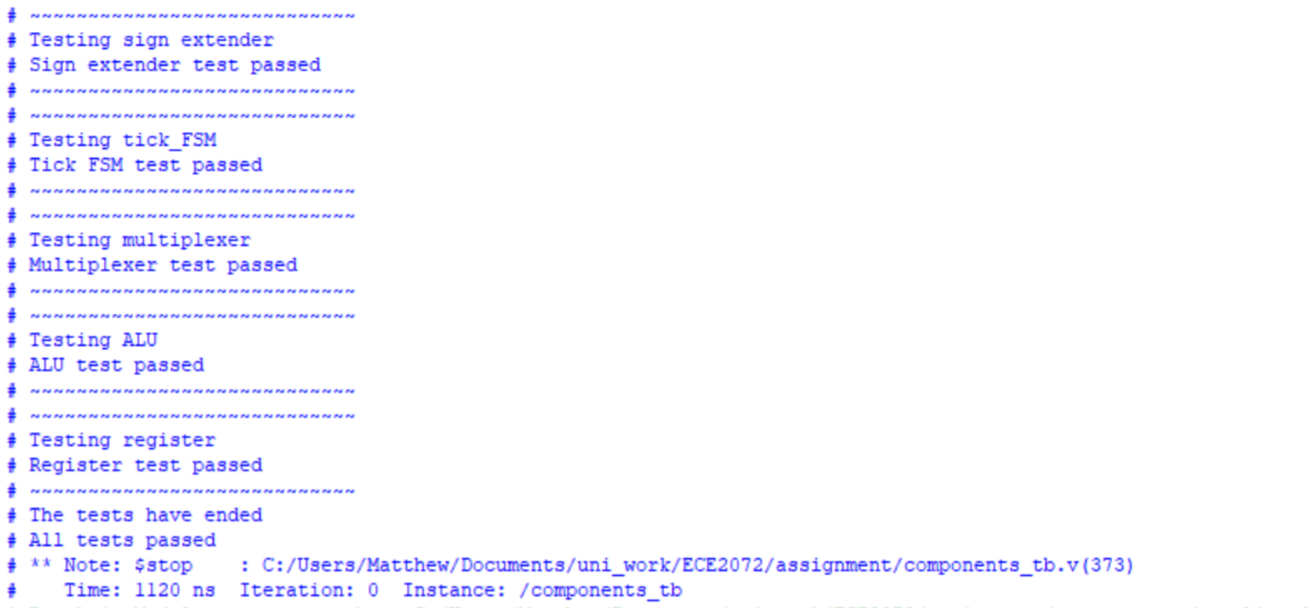
```
    $display("Test \<%s\> failed. %s", "add-pos", "r2 should be 30");
end else begin
    $display("Test \<%s\> succeeded. ", "add-pos");
end

/*
Test add instruction with negative numbers
r4 = -108
r5 = -20
r4 = r4+r5
*/
movi(3'd4, -9'sd108);
movi(3'd5, -9'sd20);
add(3'd4, 3'd5);
test_error(4'd4, -16'sd128);
if (success!=TRUE) begin
    $display("Test \<%s\> failed. %s", "add-neg", "r2 should be -88");
end else begin
    $display("Test \<%s\> succeeded. ", "add-neg");
end
$display("Test add completed with %d errors", error_count);
$display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

// test addi
$display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
$display("Tests for %s", "addi");
error_count = 0;
/*
Test addi instruction with positive numbers
r6 = 10
r6 = r6+20
*/
movi(3'd6, 9'd10);
addi(3'd6, 9'd20);
test_error(4'd6, 16'd30);
if (success!=TRUE) begin
    $display("Test \<%s\> failed. %s", "addi-pos", "r6 should be 30");
end else begin
    $display("Test \<%s\> succeeded. ", "addi-pos");
end

/*
Test addi instruction with negative numbers
r7 = 10
r7 = r7-20
*/
movi(3'd7, 9'd10);
addi(3'd7, -9'sd20);
test_error(4'd7, -16'sd10);
if (success!=TRUE) begin
    $display("Test \<%s\> failed. %s", "addi-neg", "r7 should be -10");
end else begin
    $display("Test \<%s\> succeeded. ", "addi-neg");
end
$display("Test addi completed with %d errors", error_count);
$display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

// test sub
$display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
$display("Tests for %s", "sub");
error_count = 0;
/*
```

```
        Test sub instruction with positive numbers
        r0 = 10
        r1 = 20
        r0 = r0-r1
        */
        movi(3'd0, 9'd10);
        movi(3'd1, 9'd20);
        sub(3'd0, 3'd1);
        test_error(4'd0, -16'sd10);
        if (success!=TRUE) begin
            $display("Test \<%s\> failed. %s", "sub-pos", "r0 should be -10");
        end else begin
            $display("Test \<%s\> succeeded. ", "sub-pos");
        end

        /*
        Test sub instruction with negative numbers
        r2 = -10
        r3 = -20
        r2 = r2-r3
        */
        movi(3'd2, -9'sd10);
        movi(3'd3, -9'sd20);
        sub(3'd2, 3'd3);
        test_error(4'd2, 16'sd10);
        if (success!=TRUE) begin
            $display("Test \<%s\> failed. %s", "sub-neg", "r2 should be 10");
        end else begin
            $display("Test \<%s\> succeeded. ", "sub-neg");
        end
        $display("Test sub completed with %d errors", error_count);
        $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

        // test complicated
        $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
        $display("Tests for %s", "complicated");
        error_count = 0;
        complicated_test_count = 0;
        /*
        Test complicated instruction (use all 4 available) 20 times
        Do          ((r0-immi)<-addi+(r2-r3)<-sub)<-add
        */
        for (complicated_test_count=1; complicated_test_count<20;
complicated_test_count=complicated_test_count+1) begin
            gen_rand_immi();
            movi(3'd0, rand_immi[0]);
            movi(3'd2, rand_immi[1]);
            movi(3'd3, rand_immi[2]);
            // see if a movi is done, if not, then errors after may not be useful
            test_error(4'd0, rand_immi[0]);
            if (success!=TRUE) begin
                $display("Test \<%s\> failed. %s%05d", "complicated-1-interim_movi", "r0 should be ",
rand_immi[0]);
            end else begin
                $display("Test \<%s\> succeeded. ", "complicated-1-interim_movi");
            end
            // r0 = r0-rand
            addi(3'd0, rand_immi[3]);
            // r2 = r2-r3
            sub(3'd2, 3'd3);
            // r0 = r0+r2
            add(3'd0, 3'd2);
```

```
            test_error(4'd0, (rand_immi[0]+rand_immi[3])+(rand_immi[1]-rand_immi[2]));
            if (success!=TRUE) begin
                $display("Test \<%s%02d\> failed. %s %05d", "complicated-", complicated_test_count, "r0 
should be ", (rand_immi[0]+rand_immi[3])+(rand_immi[1]-rand_immi[2]));
            end else begin
                $display("Test \<%s%02d\> succeeded. ", "complicated-", complicated_test_count);
            end
        end
        $display("Test complicated completed with %d errors", error_count);
        $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");

        $display("Tests have all finished. ");
        if (total_error == 0) begin
            $display("All tests passed!");
        end else begin
            $display("There are %d errors in total.", total_error);
        end

        $stop;
    end


endmodule
```

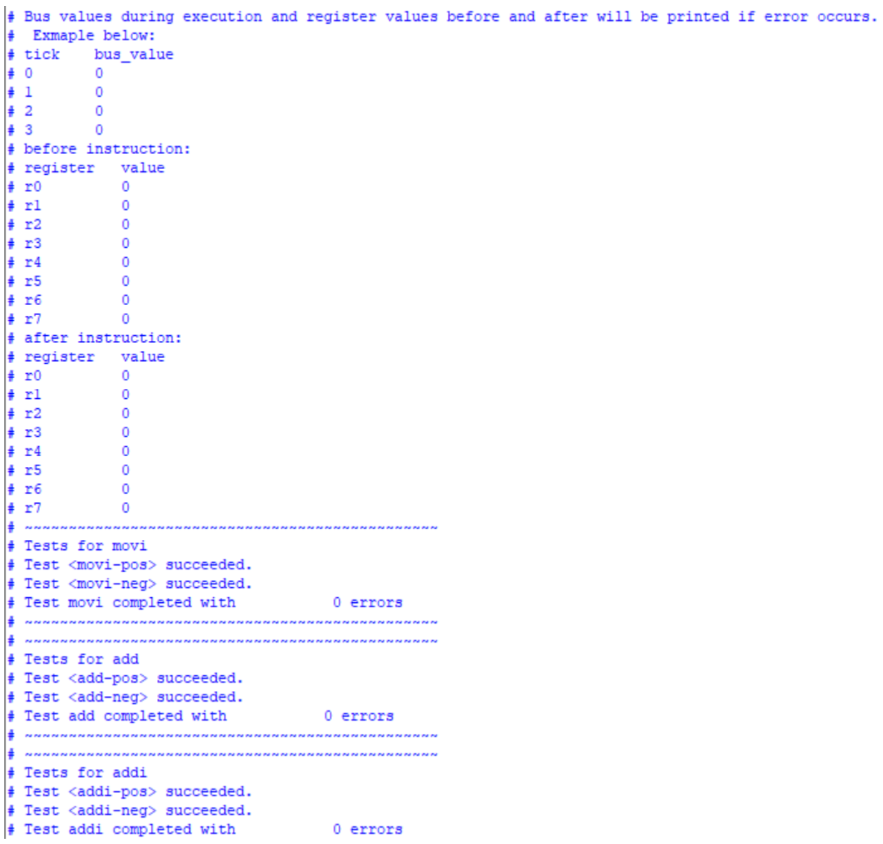
7.3 Waveforms and Console Messages

7.3.1 Component Testbench

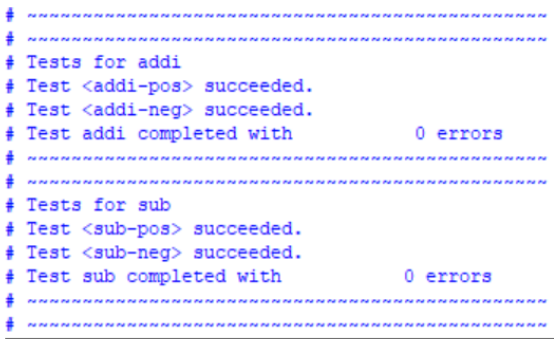


```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Testing sign extender
# Sign extender test passed
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Testing tick_FSM
# Tick FSM test passed
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Testing multiplexer
# Multiplexer test passed
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Testing ALU
# ALU test passed
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Testing register
# Register test passed
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# The tests have ended
# All tests passed
# ** Note: $stop    : C:/Users/Matthew/Documents/uni_work/ECE2072/assignment/components_tb.v(373)
#    Time: 1120 ns  Iteration: 0  Instance: /components_tb
```

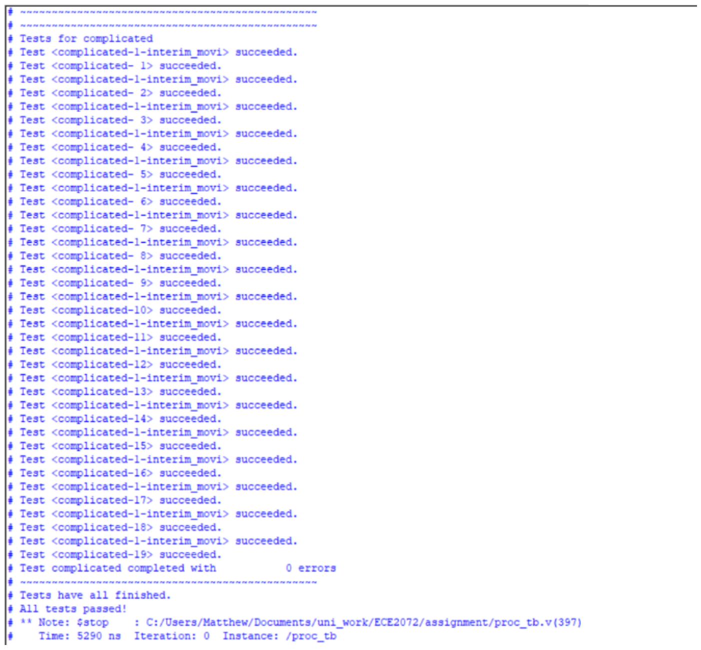
7.3.2 Processor Testbench



```
# Bus values during execution and register values before and after will be printed if error occurs.
#  Exmaple below:
# tick    bus_value
# 0       0
# 1       0
# 2       0
# 3       0
# before instruction:
# register   value
# r0         0
# r1         0
# r2         0
# r3         0
# r4         0
# r5         0
# r6         0
# r7         0
# after instruction:
# register   value
# r0         0
# r1         0
# r2         0
# r3         0
# r4         0
# r5         0
# r6         0
# r7         0
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Tests for movi
# Test <movi-pos> succeeded.
# Test <movi-neg> succeeded.
# Test movi completed with           0 errors
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Tests for add
# Test <add-pos> succeeded.
# Test <add-neg> succeeded.
# Test add completed with           0 errors
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Tests for addi
# Test <addi-pos> succeeded.
# Test <addi-neg> succeeded.
# Test addi completed with           0 errors
```

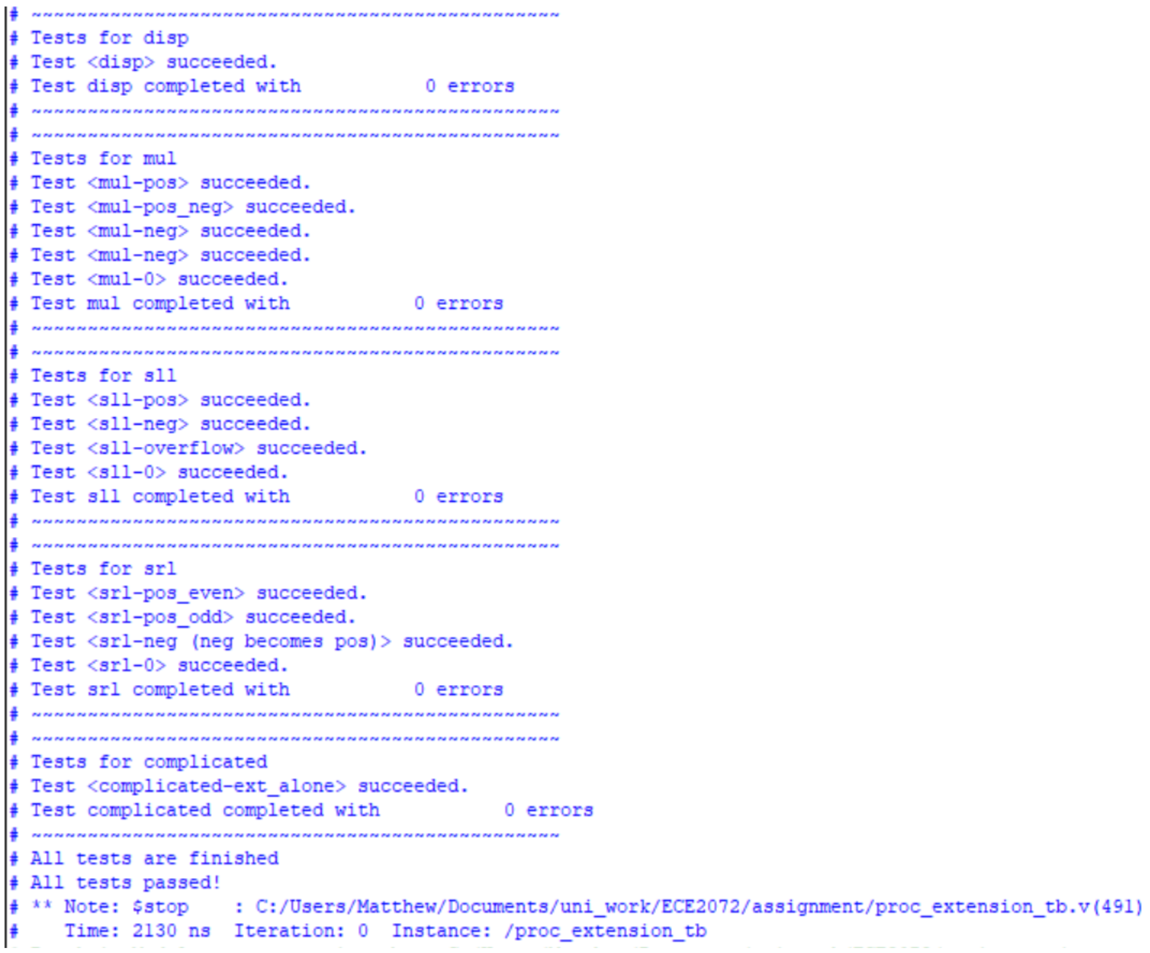


```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Tests for addi
# Test <addi-pos> succeeded.
# Test <addi-neg> succeeded.
# Test addi completed with           0 errors
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Tests for sub
# Test <sub-pos> succeeded.
# Test <sub-neg> succeeded.
# Test sub completed with           0 errors
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```



```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Tests for complicated
# Test <complicated-1-interim_movi> succeeded.
# Test <complicated- 1> succeeded.
# Test <complicated-1-interim_movi> succeeded.
# Test <complicated- 2> succeeded.
# Test <complicated-1-interim_movi> succeeded.
# Test <complicated- 3> succeeded.
# Test <complicated-1-interim_movi> succeeded.
# Test <complicated- 4> succeeded.
# Test <complicated-1-interim_movi> succeeded.
# Test <complicated- 5> succeeded.
# Test <complicated-1-interim_movi> succeeded.
# Test <complicated- 6> succeeded.
# Test <complicated-1-interim_movi> succeeded.
# Test <complicated- 7> succeeded.
# Test <complicated-1-interim_movi> succeeded.
# Test <complicated- 8> succeeded.
# Test <complicated-1-interim_movi> succeeded.
# Test <complicated- 9> succeeded.
# Test <complicated-1-interim_movi> succeeded.
# Test <complicated-10> succeeded.
# Test <complicated-1-interim_movi> succeeded.
# Test <complicated-11> succeeded.
# Test <complicated-1-interim_movi> succeeded.
# Test <complicated-12> succeeded.
# Test <complicated-1-interim_movi> succeeded.
# Test <complicated-13> succeeded.
# Test <complicated-1-interim_movi> succeeded.
# Test <complicated-14> succeeded.
# Test <complicated-1-interim_movi> succeeded.
# Test <complicated-15> succeeded.
# Test <complicated-1-interim_movi> succeeded.
# Test <complicated-16> succeeded.
# Test <complicated-1-interim_movi> succeeded.
# Test <complicated-17> succeeded.
# Test <complicated-1-interim_movi> succeeded.
# Test <complicated-18> succeeded.
# Test <complicated-1-interim_movi> succeeded.
# Test <complicated-19> succeeded.
# Test complicated completed with           0 errors
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Tests have all finished.
# All tests passed!
# ** Note: $stop    : C:/Users/Matthew/Documents/uni_work/ECE2072/assignment/proc_tb.v(397)
#    Time: 5290 ns  Iteration: 0  Instance: /proc_tb
```

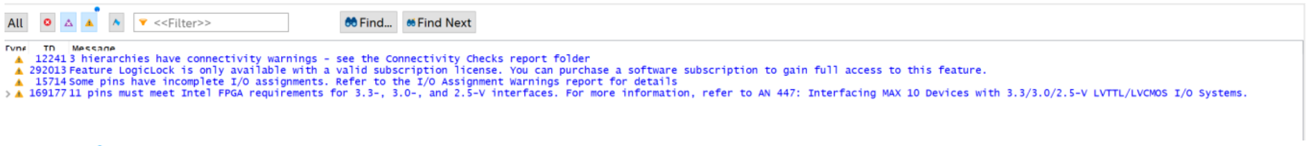
7.3.3 Processor Extended Testbench



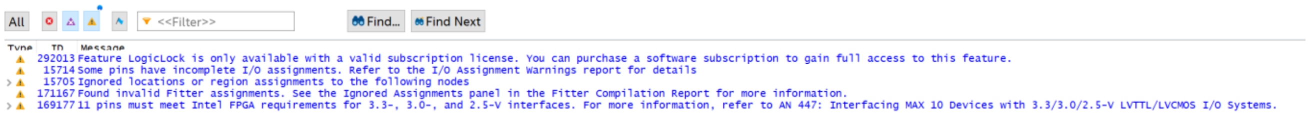
```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Tests for disp
# Test <disp> succeeded.
# Test disp completed with           0 errors
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Tests for mul
# Test <mul-pos> succeeded.
# Test <mul-pos_neg> succeeded.
# Test <mul-neg> succeeded.
# Test <mul-neg> succeeded.
# Test <mul-0> succeeded.
# Test mul completed with           0 errors
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Tests for sll
# Test <sll-pos> succeeded.
# Test <sll-neg> succeeded.
# Test <sll-overflow> succeeded.
# Test <sll-0> succeeded.
# Test sll completed with           0 errors
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Tests for srl
# Test <srl-pos_even> succeeded.
# Test <srl-pos_odd> succeeded.
# Test <srl-neg (neg becomes pos)> succeeded.
# Test <srl-0> succeeded.
# Test srl completed with           0 errors
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Tests for complicated
# Test <complicated-ext_alone> succeeded.
# Test complicated completed with           0 errors
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# All tests are finished
# All tests passed!
# ** Note: $stop    : C:/Users/Matthew/Documents/uni_work/ECE2072/assignment/proc_extension_tb.v(491)
#    Time: 2130 ns  Iteration: 0  Instance: /proc_extension_tb
```

7.4 Warnings

7.4.1 Compiler Warnings

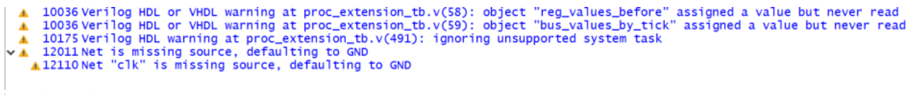


```
Type  ID     Message
  ▲  12241  3 hierarchies have connectivity warnings - see the Connectivity Checks report folder
  ▲  292013 Feature LogicLock is only available with a valid subscription license. You can purchase a software subscription to gain full access to this feature.
  ▲  15714  Some pins have incomplete I/O assignments. Refer to the I/O Assignment Warnings report for details
> ▲  169177 11 pins must meet Intel FPGA requirements for 3.3-, 3.0-, and 2.5-V interfaces. For more information, refer to AN 447: Interfacing MAX 10 Devices with 3.3/3.0/2.5-V LVTTL/LVCMOS I/O Systems.
```

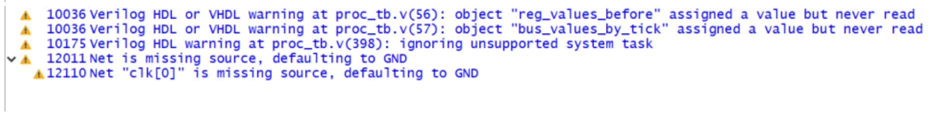


```
Type  ID     Message
  ▲  292013 Feature LogicLock is only available with a valid subscription license. You can purchase a software subscription to gain full access to this feature.
  ▲  15714  Some pins have incomplete I/O assignments. Refer to the I/O Assignment Warnings report for details
> ▲  15705  Ignored locations or region assignments to the following nodes
  ▲  171167 Found invalid Fitter assignments. See the Ignored Assignments panel in the Fitter Compilation Report for more information.
> ▲  169177 11 pins must meet Intel FPGA requirements for 3.3-, 3.0-, and 2.5-V interfaces. For more information, refer to AN 447: Interfacing MAX 10 Devices with 3.3/3.0/2.5-V LVTTL/LVCMOS I/O Systems.
```

7.4.2 Testbench Warnings



```
  ▲  10036 Verilog HDL or VHDL warning at proc_extension_tb.v(58): object "reg_values_before" assigned a value but never read
  ▲  10036 Verilog HDL or VHDL warning at proc_extension_tb.v(59): object "bus_values_by_tick" assigned a value but never read
  ▲  10175 Verilog HDL warning at proc_extension_tb.v(491): ignoring unsupported system task
v ▲  12011 Net is missing source, defaulting to GND
    ▲12110 Net "clk" is missing source, defaulting to GND
```



```
  ▲  10036 Verilog HDL or VHDL warning at proc_tb.v(56): object "reg_values_before" assigned a value but never read
  ▲  10036 Verilog HDL or VHDL warning at proc_tb.v(57): object "bus_values_by_tick" assigned a value but never read
  ▲  10175 Verilog HDL warning at proc_tb.v(398): ignoring unsupported system task
v ▲  12011 Net is missing source, defaulting to GND
    ▲12110 Net "clk[0]" is missing source, defaulting to GND
```