Lab #5: Processor

Joshua Hubert
Student ID # 38223876
Introduction to Digital Logic Design Lab
EECS 31L
March 13, 2021

1. Objective

In this final lab I created controller and ALU controller modules to control my Datapath module, and then created a processor super-module to house those three submodules using the structural design pattern. I then took the processor testbench code provided and used it to test the processor, running the instructions stored in the Instruction Memory and checking for correct output.

2. Procedure

As shown in the block diagram below, the processor consists of a controller, ALU controller, and datapath. I did not change my datapath's functionality from the previous lab so I will not comment on that.

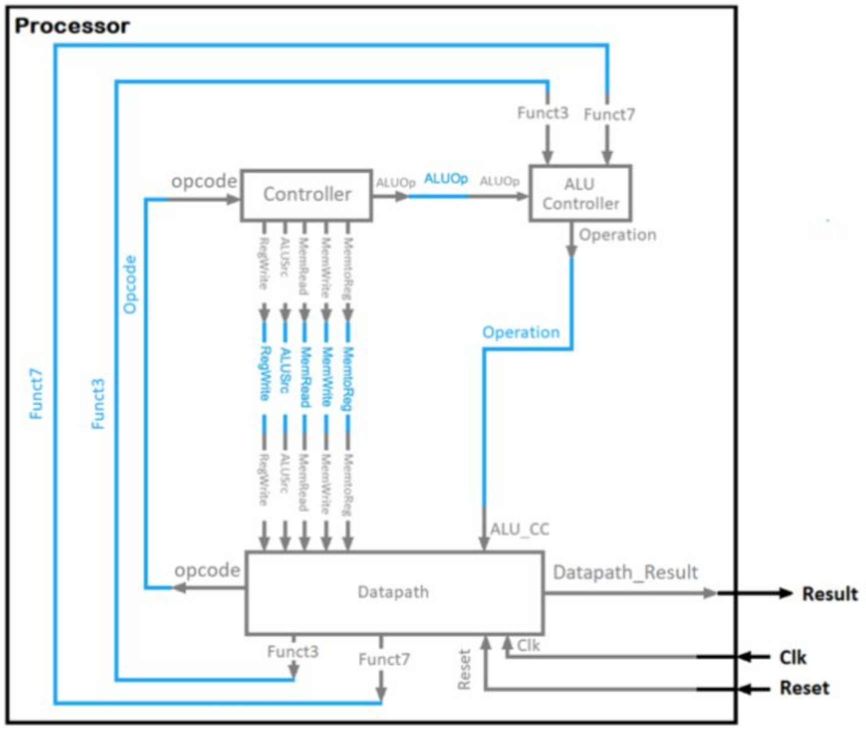
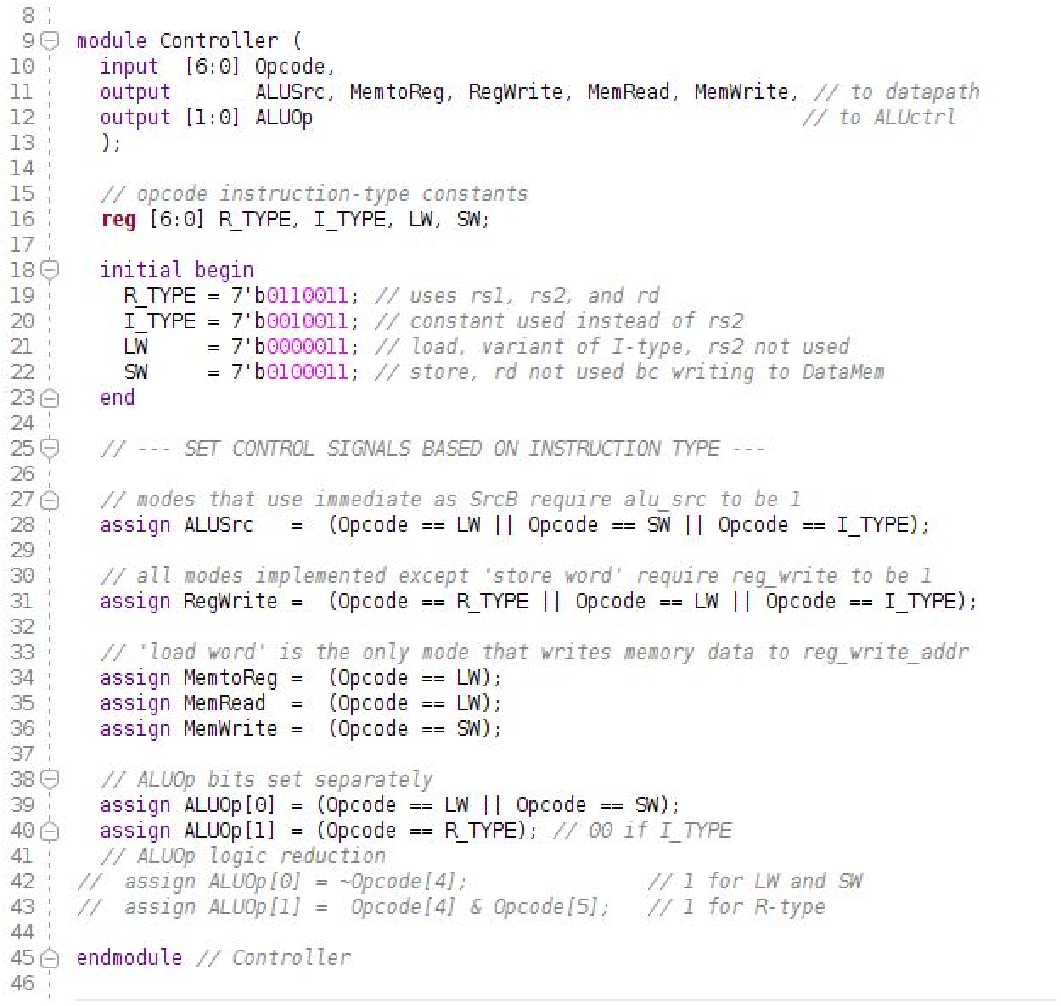


Figure 1 : Processor.

Controller:

The controller takes in the opcode for the current instruction from the datapath, and determines which single-bit datapath control flags to set for that instruction. I used four constant reg signals to give the four instruction types human-readable names (R-type for 2 source registers used, I-type for the second source being an immediate, LW or load word for loading from memory, SW or store word for storing to memory). Then the logic for the concurrent assignments was pretty straightforward based on the table given in the lab manual (see code comments for further analysis/reiteration of the table). The other output is the ALUOp. The logic for the ALUOp was broken down for each of its two bits, to produce the proper boolean as simply possible. Originally I thought it would be slick to check only the bits that differentiated the different opcodes for this (see commented-out code at the bottom) but then decided it was safer to be more explicit.

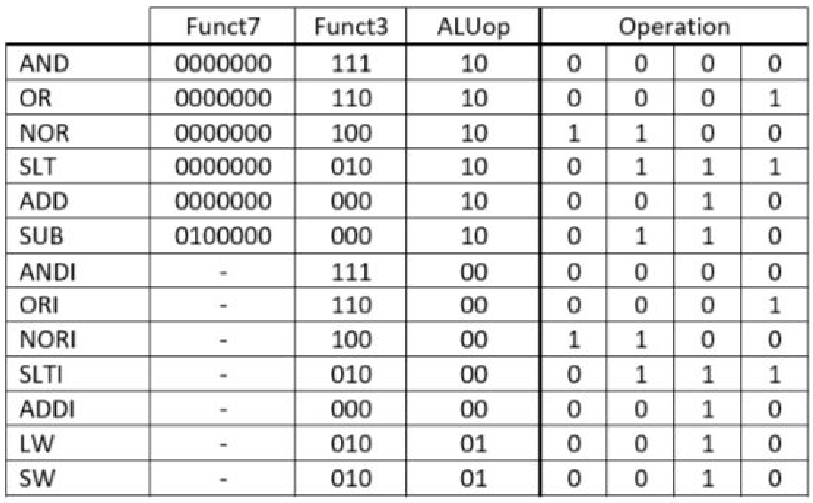


```

module Controller (
  input  [6:0] Opcode,
  output       ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, // to datapath
  output [1:0] ALUOp                                          // to ALUctrl
  );

  // opcode instruction-type constants
  reg [6:0] R_TYPE, I_TYPE, LW, SW;

  initial begin
    R_TYPE = 7'b0110011; // uses rs1, rs2, and rd
    I_TYPE = 7'b0010011; // constant used instead of rs2
    LW     = 7'b0000011; // load, variant of I-type, rs2 not used
    SW     = 7'b0100011; // store, rd not used bc writing to DataMem
  end

  // --- SET CONTROL SIGNALS BASED ON INSTRUCTION TYPE ---

  // modes that use immediate as SrcB require alu_src to be 1
  assign ALUSrc   =  (Opcode == LW || Opcode == SW || Opcode == I_TYPE);

  // all modes implemented except 'store word' require reg_write to be 1
  assign RegWrite =  (Opcode == R_TYPE || Opcode == LW || Opcode == I_TYPE);

  // 'load word' is the only mode that writes memory data to reg_write_addr
  assign MemtoReg =  (Opcode == LW);
  assign MemRead  =  (Opcode == LW);
  assign MemWrite =  (Opcode == SW);

  // ALUOp bits set separately
  assign ALUOp[0] = (Opcode == LW || Opcode == SW);
  assign ALUOp[1] = (Opcode == R_TYPE); // 00 if I_TYPE
  // ALUOp logic reduction
//  assign ALUOp[0] = ~Opcode[4];              // 1 for LW and SW
//  assign ALUOp[1] =  Opcode[4] & Opcode[5];  // 1 for R-type

endmodule // Controller

```

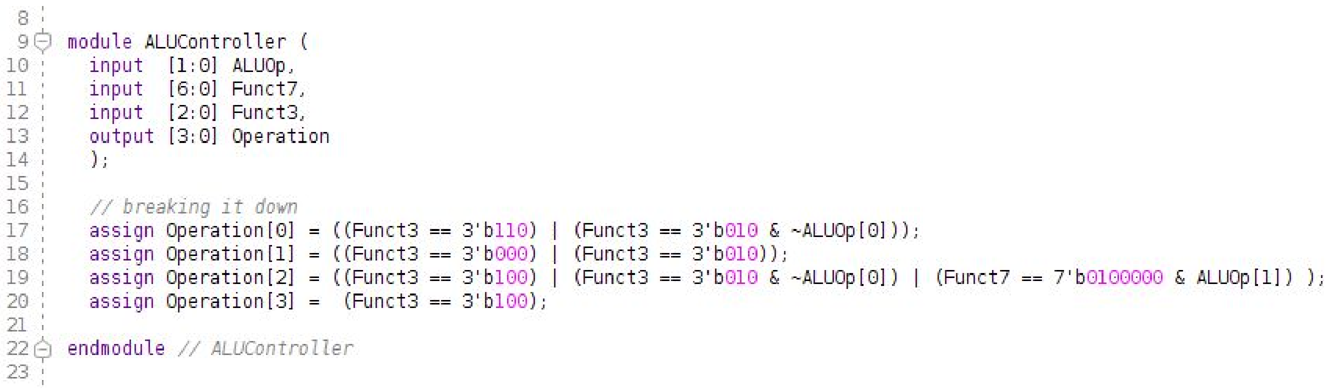
ALU Controller:

The ALU controller takes the 2-bit ALUOp from the controller and the Funct3 and Funct7 codes from the datapath as input, and determines the proper ALU control signal to send to the ALU in the datapath based on the operation that needs to be performed for the current instruction. Once again it was simplest and cleanest to just assign each bit of the 4-bit output Operation code separately, using the parts of input codes that differentiated them, based on the truth table in the lab manual, shown below.

Table 4 : The truth table for the 4 operation code.



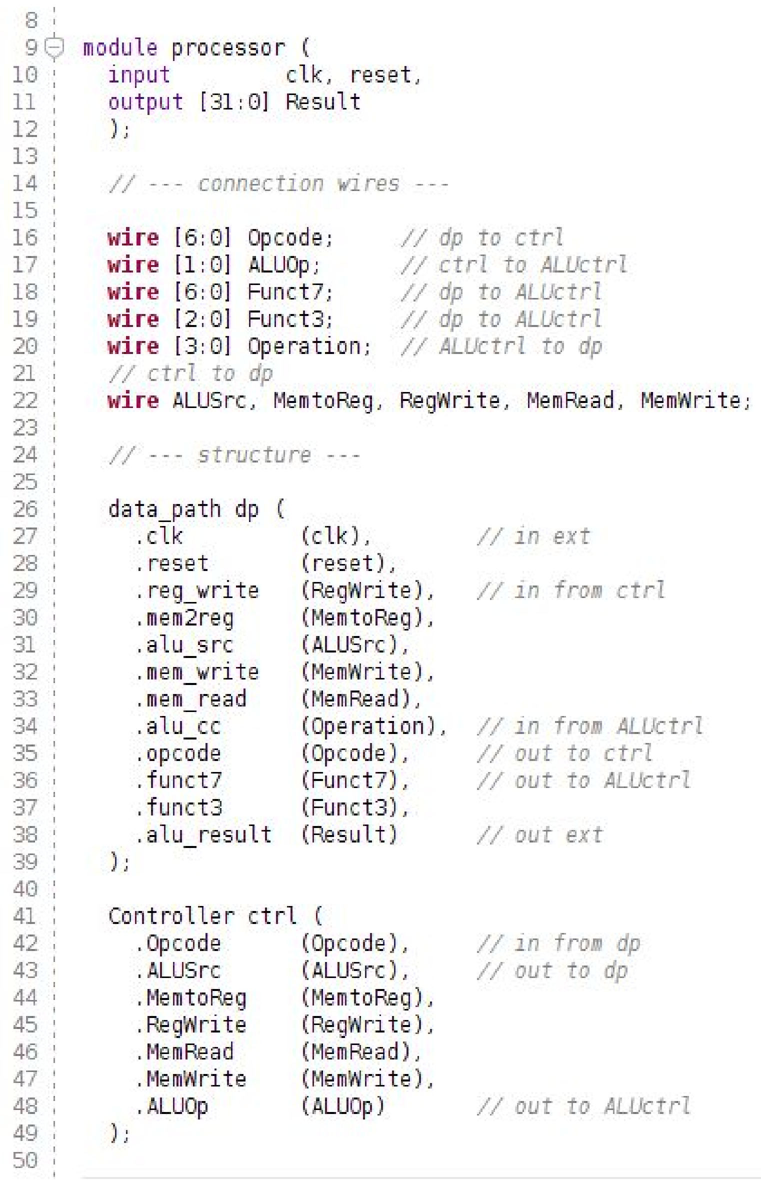
	Funct7	Funct3	ALUop	Operation			
AND	0000000	111	10	0	0	0	0
OR	0000000	110	10	0	0	0	1
NOR	0000000	100	10	1	1	0	0
SLT	0000000	010	10	0	1	1	1
ADD	0000000	000	10	0	0	1	0
SUB	0100000	000	10	0	1	1	0
ANDI	-	111	00	0	0	0	0
ORI	-	110	00	0	0	0	1
NORI	-	100	00	1	1	0	0
SLTI	-	010	00	0	1	1	1
ADDI	-	000	00	0	0	1	0
LW	-	010	01	0	0	1	0
SW	-	010	01	0	0	1	0



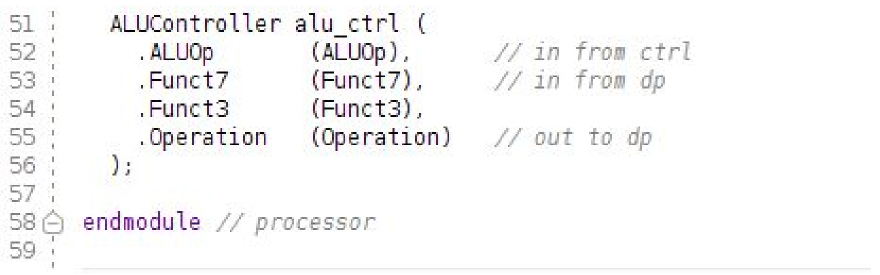
```
module ALUController (
  input  [1:0] ALUOp,
  input  [6:0] Funct7,
  input  [2:0] Funct3,
  output [3:0] Operation
  );

  // breaking it down
  assign Operation[0] = ((Funct3 == 3'b110) | (Funct3 == 3'b010 & ~ALUOp[0]));
  assign Operation[1] = ((Funct3 == 3'b000) | (Funct3 == 3'b010));
  assign Operation[2] = ((Funct3 == 3'b100) | (Funct3 == 3'b010 & ~ALUOp[0]) | (Funct7 == 7'b0100000 & ALUOp[1]) );
  assign Operation[3] =  (Funct3 == 3'b100);

endmodule // ALUController
```

Processor:

The processor module is just a container for the three previously mentioned modules, to connect them with wires just like in the block diagram on the first page. The only inputs are the clock and reset signals, and the only output is the 32 bit ALU result “Result”. The wires were named just like in the diagram, and mostly very similar to the inputs and outputs they are connected to. Since so many different inputs and outputs were being connected, I took care to lay the instantiations out cleanly to make it easy to see everything.



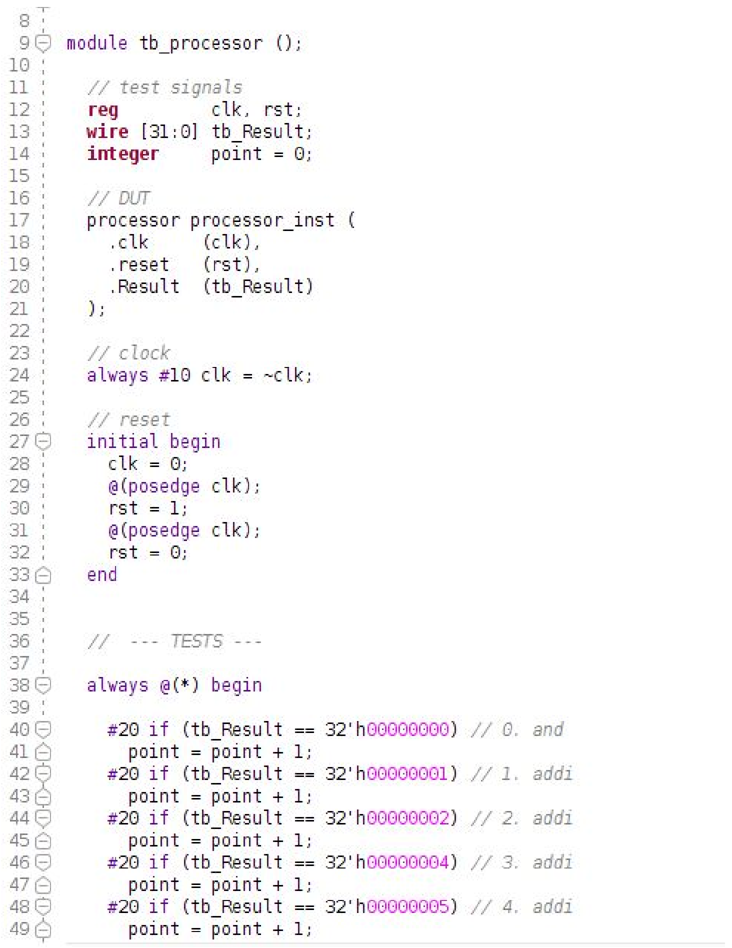
```
module processor (
  input         clk, reset,
  output [31:0] Result
  );

  // --- connection wires ---

  wire [6:0] Opcode;     // dp to ctrl
  wire [1:0] ALUOp;      // ctrl to ALUctrl
  wire [6:0] Funct7;     // dp to ALUctrl
  wire [2:0] Funct3;     // dp to ALUctrl
  wire [3:0] Operation;  // ALUctrl to dp
  // ctrl to dp
  wire ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite;

  // --- structure ---

  data_path dp (
    .clk         (clk),        // in ext
    .reset       (reset),
    .reg_write   (RegWrite),   // in from ctrl
    .mem2reg     (MemtoReg),
    .alu_src     (ALUSrc),
    .mem_write   (MemWrite),
    .mem_read    (MemRead),
    .alu_cc      (Operation),  // in from ALUctrl
    .opcode      (Opcode),     // out to ctrl
    .funct7      (Funct7),     // out to ALUctrl
    .funct3      (Funct3),
    .alu_result  (Result)      // out ext
  );

  Controller ctrl (
    .Opcode      (Opcode),     // in from dp
    .ALUSrc      (ALUSrc),     // out to dp
    .MemtoReg    (MemtoReg),
    .RegWrite    (RegWrite),
    .MemRead     (MemRead),
    .MemWrite    (MemWrite),
    .ALUOp       (ALUOp)       // out to ALUctrl
  );

```



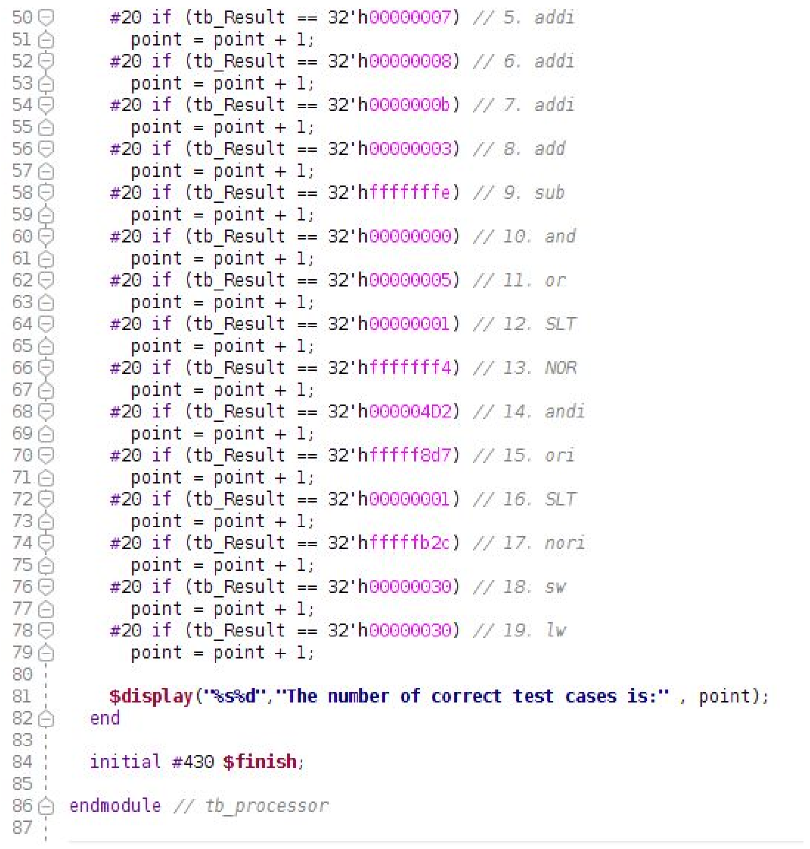
```
  ALUController alu_ctrl (
    .ALUOp       (ALUOp),      // in from ctrl
    .Funct7      (Funct7),     // in from dp
    .Funct3      (Funct3),
    .Operation   (Operation)   // out to dp
  );

endmodule // processor
```

Processor Testbench:

All that the testbench has to do is supply a clock and reset signal, and run for 430 nanoseconds, the 20 clock cycle duration it takes to read the 20 instructions, checking in the middle of each cycle whether the ALU output is giving the correct value for its instruction. The point variable is incremented for each correct output, and the total points are printed to the TCL console when the tests finish.



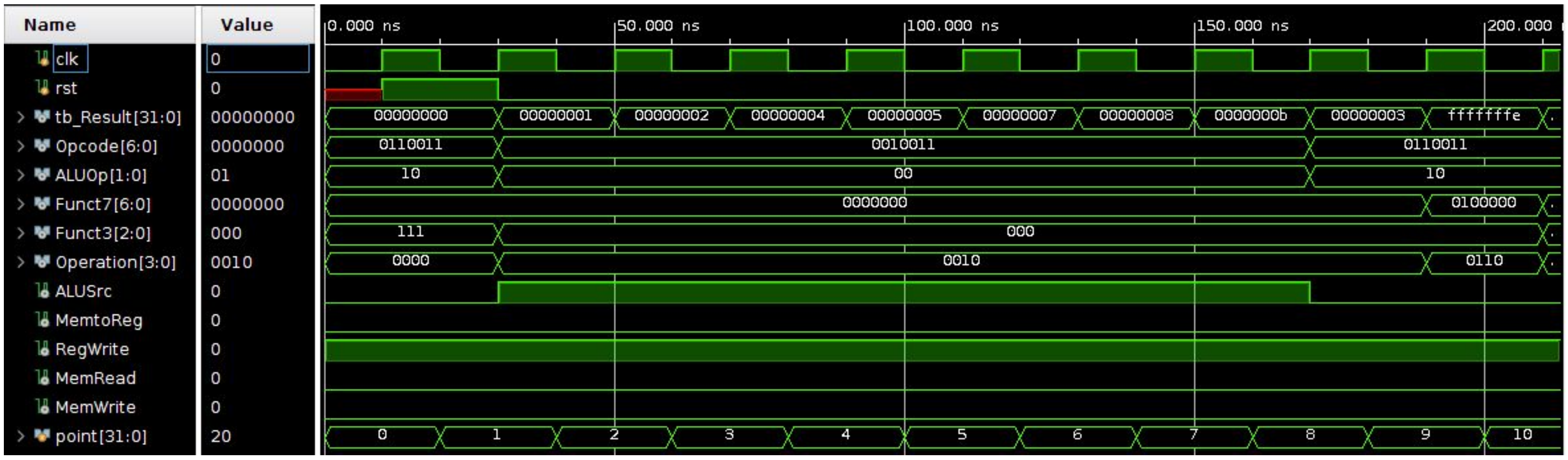
```verilog

module tb_processor ();

  // test signals
  reg          clk, rst;
  wire [31:0]  tb_Result;
  integer      point = 0;

  // DUT
  processor processor_inst (
    .clk     (clk),
    .reset   (rst),
    .Result  (tb_Result)
  );

  // clock
  always #10 clk = ~clk;

  // reset
  initial begin
    clk = 0;
    @(posedge clk);
    rst = 1;
    @(posedge clk);
    rst = 0;
  end


  //  --- TESTS ---

  always @(*) begin

    #20 if (tb_Result == 32'h00000000) // 0. and
      point = point + 1;
    #20 if (tb_Result == 32'h00000001) // 1. addi
      point = point + 1;
    #20 if (tb_Result == 32'h00000002) // 2. addi
      point = point + 1;
    #20 if (tb_Result == 32'h00000004) // 3. addi
      point = point + 1;
    #20 if (tb_Result == 32'h00000005) // 4. addi
      point = point + 1;
```



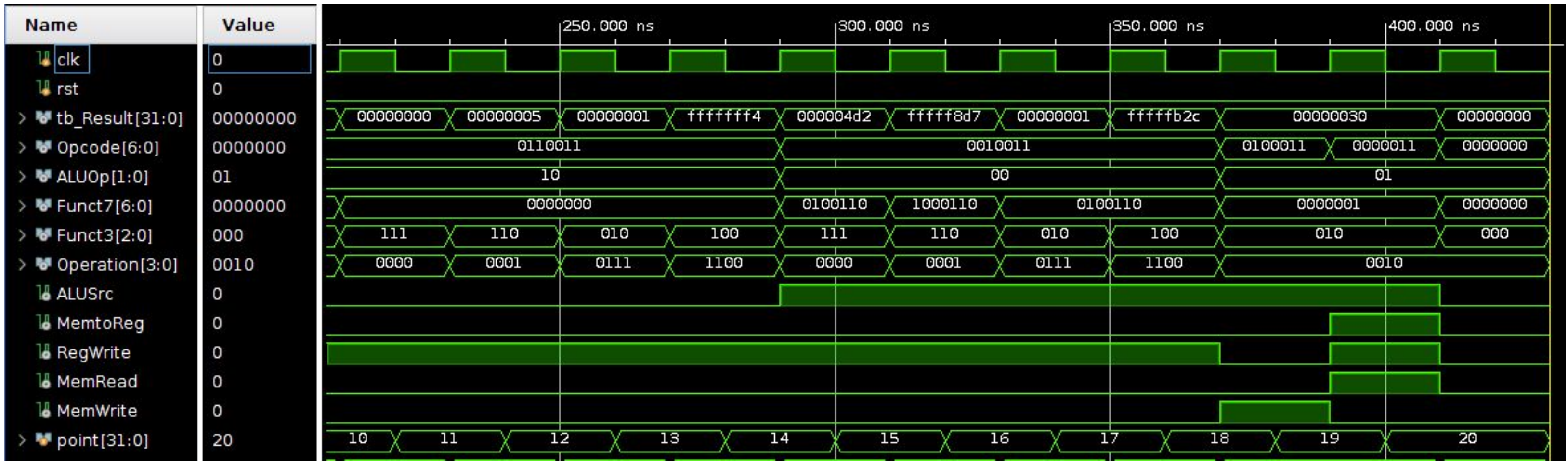
```verilog
    #20 if (tb_Result == 32'h00000007) // 5. addi
      point = point + 1;
    #20 if (tb_Result == 32'h00000008) // 6. addi
      point = point + 1;
    #20 if (tb_Result == 32'h0000000b) // 7. addi
      point = point + 1;
    #20 if (tb_Result == 32'h00000003) // 8. add
      point = point + 1;
    #20 if (tb_Result == 32'hfffffffe) // 9. sub
      point = point + 1;
    #20 if (tb_Result == 32'h00000000) // 10. and
      point = point + 1;
    #20 if (tb_Result == 32'h00000005) // 11. or
      point = point + 1;
    #20 if (tb_Result == 32'h00000001) // 12. SLT
      point = point + 1;
    #20 if (tb_Result == 32'hfffffff4) // 13. NOR
      point = point + 1;
    #20 if (tb_Result == 32'h000004D2) // 14. andi
      point = point + 1;
    #20 if (tb_Result == 32'hfffff8d7) // 15. ori
      point = point + 1;
    #20 if (tb_Result == 32'h00000001) // 16. SLT
      point = point + 1;
    #20 if (tb_Result == 32'hfffffb2c) // 17. nori
      point = point + 1;
    #20 if (tb_Result == 32'h00000030) // 18. sw
      point = point + 1;
    #20 if (tb_Result == 32'h00000030) // 19. lw
      point = point + 1;

    $display("%s%d","The number of correct test cases is:" , point);
  end

  initial #430 $finish;

endmodule // tb_processor

```

3. Simulation Results

I received all 20 points for the output checks for all 20 instructions. This was not surprising, as my ALU outputs had previously been correct when I tested them last lab with the datapath testbench, and that testbench functionally did the same job as the controller and ALU controller combined. See my results below, which include the processor internal signals as well, split into two screenshots.



Name
Value
clk 0
rst 0
tb_Result[31:0] 00000000
Opcode[6:0] 0000000
ALUOp[1:0] 01
Funct7[6:0] 0000000
Funct3[2:0] 000
Operation[3:0] 0010
ALUSrc 0
MemtoReg 0
RegWrite 0
MemRead 0
MemWrite 0
point[31:0] 20
0.000 ns
50.000 ns
100.000 ns
150.000 ns
200.000
00000000 00000001 00000002 00000004 00000005 00000007 00000008 0000000b 00000003 fffffffe
0110011 0010011 0110011
10 00 10
0000000 0100000
111 000
0000 0010 0110
0 1 2 3 4 5 6 7 8 9 10



Name
Value
clk 0
rst 0
tb_Result[31:0] 00000000
Opcode[6:0] 0000000
ALUOp[1:0] 01
Funct7[6:0] 0000000
Funct3[2:0] 000
Operation[3:0] 0010
ALUSrc 0
MemtoReg 0
RegWrite 0
MemRead 0
MemWrite 0
point[31:0] 20
250.000 ns
300.000 ns
350.000 ns
400.000 ns
00000000 00000005 00000001 fffffff4 000004d2 fffff8d7 00000001 fffffb2c 00000030 00000000
0110011 0010011 0100011 0000011 0000000
10 00 01
0000000 0100110 1000110 0100110 0000001 0000000
111 110 010 100 111 110 010 100 010 000
0000 0001 0111 1100 0000 0001 0111 1100 0010
10 11 12 13 14 15 16 17 18 19 20