Dylan Thornburg and Viet Ha

SANTA CLARA UNIVERSITY	ECEN 122 Winter 2024	Hoeseok Yang

Laboratory #3: Define and implement new operations

For lab section on January 30/February 2, 2024

I. OBJECTIVES

Given a simple CPU datapath and the definition of just a few operations, define and implement new operations by making modifications to the datapath and enhancements to the control state machine.

PROBLEM STATEMENT

We will build off of the simple CPU that you worked on in the last lab. That CPU only supported four operations: *load*, *move*, *add*, and *subtract*. In this lab we are going to add a few new instructions. Specifically, we are going to implement variants of the add and subtract instructions that will use an immediate value for the second operand rather than a value from a register, and a "display" operation that will cause the value in a specified register to be captured in a special memory that we can think of as a value that might be displayed on a screen, if we were implementing something like a calculator.

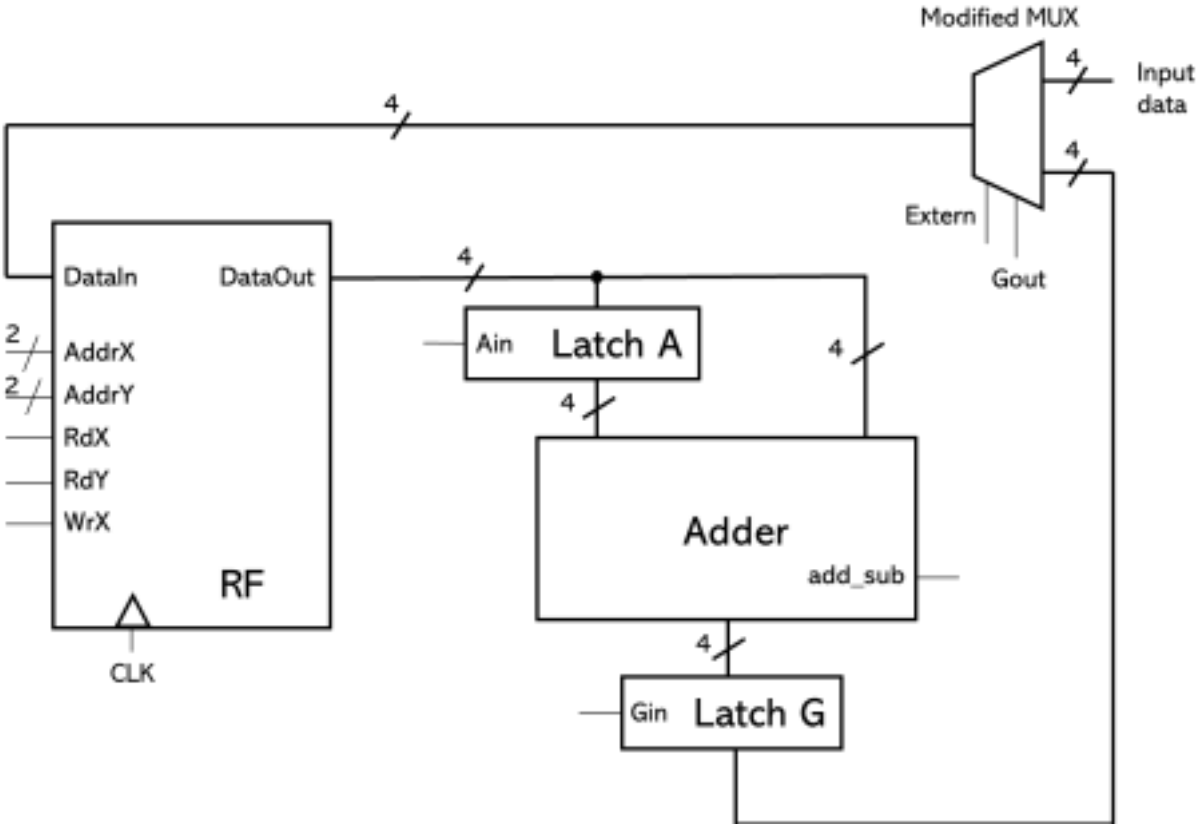


Figure 1. A simple datapath implemented in Lab 2

Added Operations

Figure 1 shows the datapath that you had to work with in the last lab. Let us discuss the new operations and what changes need to happen to support them:

- Add/subtract immediate (*addi*/*subi*) – The datapath changes are the same for these two new operations, so we will discuss them together. The "normal" *add*/*sub* operations (the ones you implemented in the last lab) read a value from the RF and capture it into staging latch A, and then read a second operand from the RF that is used by the secondary input of the adder. In the case of this new operation, the first operand is still read from the RF and captured in staging latch A. But the data for the secondary input to the adder needs to be the data coming from the immediate (*Imm*) field of the instruction (i.e., *addi* causes X = X + *Imm* while *subi* causes X = X – *Imm*.) So we will need a pathway for that data to reach the secondary input to the adder, without breaking the support for the register-based *add* or the support for the *load* operation.

- Display –You need to implement a holding latch (not unlike latch A and latch G in the diagram above) that gets loaded by the display operation with the contents of whatever register is specified by *AddrX* as part of the operation. Again, the idea is that the value in this memory would be displayed on a screen (not really, but conceptually). If you want to display a different value, you will need to execute another display operation. Relative to the diagram above, you
  will need to define where this new latch will be added and how data from the RF can be loaded into it.

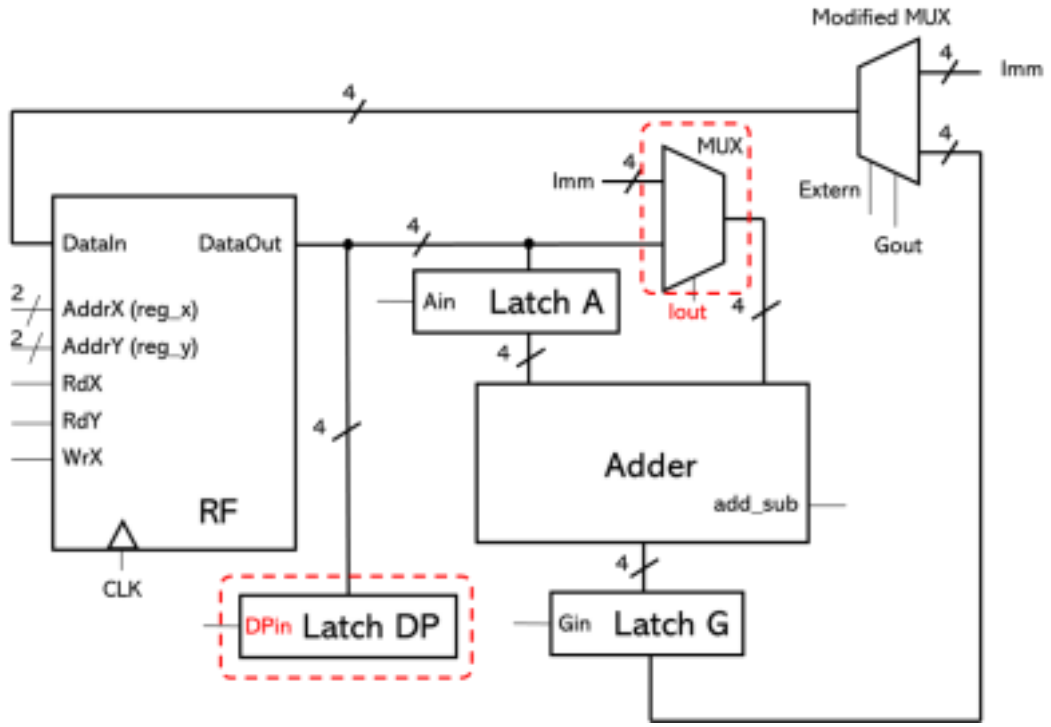


Figure 2. The modified datapath for Lab 3

Changes to the datapath

Given the new operations that we are looking to implement, it needs to change. To support the newly added instructions, we need to add two more Verilog modules as depicted in Figure 2. That is,

- MUX – Previously, the second 4-bit operand of the adder always came from the RF. On the other hand, we now must selectively feed the data from the *Imm* field of the instruction (for *addi* and *subi*) or from the RF's *DataOut* (for the other instructions). A 4-bit 2-to-1 MUX (mux_2_to_1.v) in added in the datapath for this purpose. While the source code of this MUX is given in a completed form, take the time to look at the source code; it is worth having a look as it is written in a generic way to support any bit lengths. I.e., the same implementation can be reused without any modification for different big lengths such as 16-bit or 32-bit.[1] Through this MUX, *DataOut* from the RF is connected to the second operand of Adder if *Iout* is 0. On the other hand, when *Iout* is 1, *Imm* is the second operand that Adder gets.

- Latch DP – For the display latch, you can reuse the latch code (A.v) that was used for latch A and latch G in Lab 2. The enable input (*DPin*) of this latch should be properly controlled by the control logic.

- Control logic – The control state machine needs to be modified properly with two additional control output signals (*Iout* and *DPin*).

[1] See page 13 and page 31 of the Verilog Tutorial for the usage of parameter and parameterized instantiation.

Extending Opcode

In Lab 2, we used a 2-bit opcode to specify which operation to perform. Now that we are adding three new operations, we need to add another bit to our opcode. This will give us a 3-bit value for the purposes of specifying the operation to perform, which in turn means that you will need to extend the specification of your encodings for all the possible operations, so that each is now a unique 3-bit value. And your state machine, which is dependent on properly decoding the specified operation, will need to account for the new encoding specification. We assume the following extended encoding for opcode (operation in l3_SM.v):

- 3'b000: load
- 3'b001: move
- 3'b010: subtract
- 3'b011: add
- 3'b100: display
- 3'b101: (reserved, i.e., not used for now)
- 3'b110: subi
- 3'b111: addi

With this extended opcode, we have an 11-bit instruction format for this datapath as illustrated below:

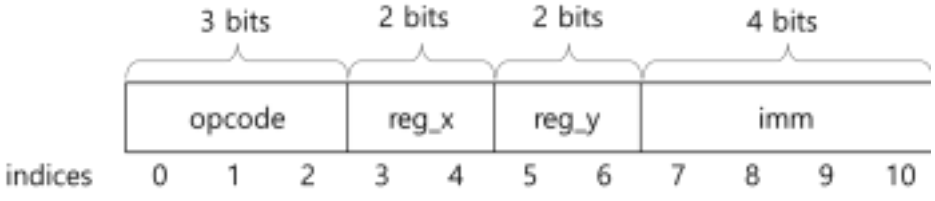


Figure 3. Instruction Format

II. PRE-LAB

(i) Based on the updated block diagram (Figure 2) and instruction format (Figure 3) which show the datapath support for the new operations, properly modify the module interface of l3_SM.v (TODO 1).

(ii) Draw out an updated state diagram that accounts for the newly added operations. You may need to define additional states in addition to the existing state machine you implemented in Lab 2 (See below for your reference). As per the modified state diagram, you would need to define more states in the control logic Verilog file (l3_SM.v, TODO 2).

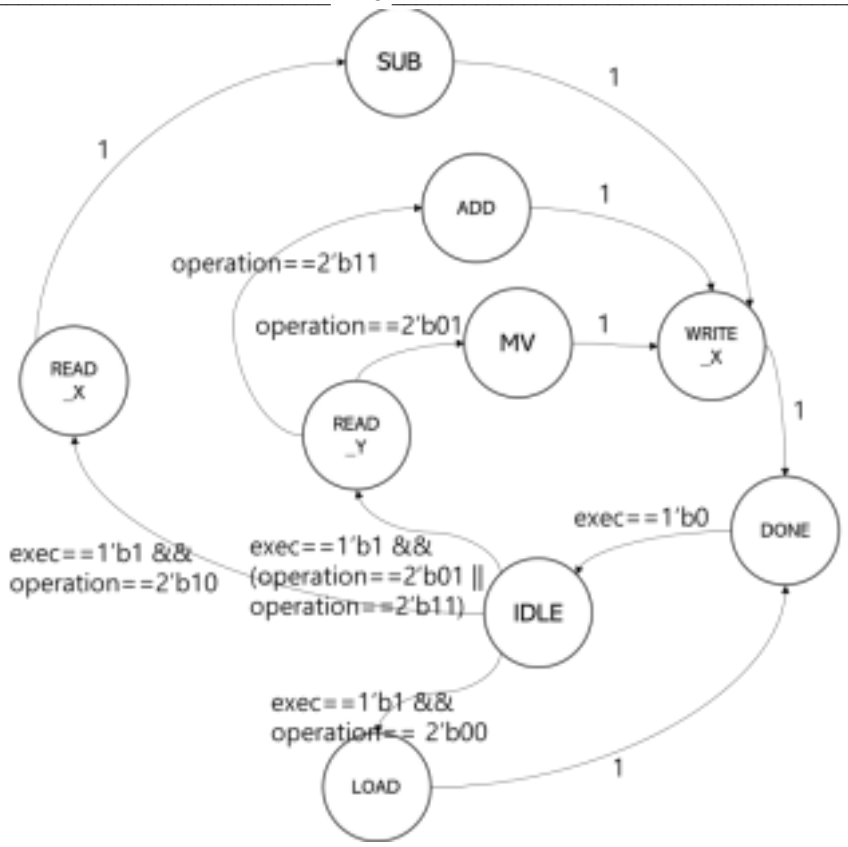


Figure 4. A Moore machine that was used as control logic in Lab 2

(iii) Write the output table of your new Moore machine that has the additional control output signals (l3_SM.v, TODO 3).

State	*RdX*	*RdY*	*WrX*	*Ain*	*Gin*	*Extern*	*Gout*	*add_sub*	*DPin*	*Iout*
IDLE	0	0	0	0	0	0	0	0	0	0
LOAD	0	0	1	0	0	1	0	0	0	0
READ_Y	0	1	0	1	0	0	0	0	0	0
READ_X	1	0	0	1	0	0	0	0	0	0

ADD	1	0	0	0	1	0	0	0	0	0
SUB	0	1	0	0	1	0	0	1	0	0
MV	0	0	0	0	1	0	0	0	0	0
WRITE_X	0	0	1	0	0	0	1	0	0	0
DONE	0	0	0	0	0	0	0	0	0	0
ADDI	0	0	0	0	1	0	0	0	0	1
SUBI	0	0	0	0	1	0	0	1	0	1
DISP	1	0	0	0	0	0	0	0	1	0
Reserved (not used yet)	0	0	0	0	0	0	0	0	0	0

Again, while completing the Verilog for pre-lab submission is not required, it is advisable that you implement as much of the updated state machine as you can before coming to lab, so that you will have more time for testing and debugging.

III. LAB PROCEDURE

(i) Once again, work with your lab partner to reconcile your pre-labs. Compare your state diagrams and output tables and make a decision on what you're going to implement.

(ii) One of you should work on the state machine code that you have decided to pursue. (Start from the given skeleton code – TODO 4 - or the code you used in Lab 2.)

(iii)In parallel, the other person can bring up Vivado and start making the necessary changes to the datapath (in l3_tb.v). Complete TODO 5, 6, and 7 to instantiate datapath elements such as MUXes and Latches.

(iv)Once the updated state machine code is ready, import this into Vivado and finish making the connections.

(v) The testbench you used in Lab 2 is in l3_tb.v. Add your own test sequences to test if the newly added operations work as intended (TODO 8). You must see all addi, subi, and disp work well in your test sequence and make sure that the final value stored in the display latch is 10 (in decimal).

(vi)Once you have determined that your design appears to be working properly, demonstrate to the TA.

IV. REPORT

For your lab report, include the source code for your working state machine. In addition, include answers to the following questions.

• What problems did you encounter while implementing and testing your system? • Did any problems arise when demonstrating for the TA? In other words, did the TA ask you to demonstrate something that you did not think of yourself? What was the scenario that you were asked to demonstrate? Provide some thoughts about why you didn't think of this yourself.

Our main problems were syntax related. We also had one stand alone issue where we switched around the inputs for the multiplexor. We didn't have any problems during the demo, but Rikesh wanted us to explain what exactly "instr" was and how it worked, which we did.

• In this design, we moved to a 3-bit operation encoding yet only implemented 7 operations in total. That leaves one encoding unused. Think of another operation you could envision implementing with this unused operation encoding, and describe/explain the datapath changes that would be necessary to support execution of this new operation.

We think one of the new operations could Display the number in the y register/address (call it DisplayY). The display function currently only displays the the value in register X, so instead of moving Y into X and then displaying, it could be convenient to just display Y. I think we could do this by adding in a 2 to 1 mux that takes the dataout input and rdy input (rdy for y, dataout for x). We would need a new selection signal too.

CODE SM File:

```
module l3_SM(input clk,
                input execute,
                input [2:0] operation, // opcode (part of instruction)
                output reg _Extern,
                output reg Gout,
                output reg Iout,
                output reg Ain,
                output reg Gin,
                output reg DPin,
                //output reg [1:0] AddrX,
                //output reg [1:0] AddrY,
                output reg RdX,
                output reg RdY,
```

```
                    output reg WrX,
                    output reg add_sub,
                    output [3:0] cur_state);

    //defining all my states - 8 total
    parameter IDLE          = 4'b0000;
    parameter LOAD          = 4'b0001;
    parameter READ_Y  = 4'b0010;
    parameter READ_X  = 4'b0011;
    parameter ADD           = 4'b0100;
    parameter SUB           = 4'b0101;
    parameter MV            = 4'b0110;
    parameter WRITE_X = 4'b0111;
    parameter DONE          = 4'b1000;
    parameter ADDI          = 4'b1001;
    parameter SUBI          = 4'b1010;
    parameter DISP          = 4'b1011;

    reg [3:0] state = IDLE; // initial state being IDLE

    assign cur_state = state;

    initial begin //instead of reset
state <= IDLE;
    end

    always@(*)
    begin
      case(state)
        IDLE:
          begin
            _Extern = 1'b0;
            Gout = 1'b0;
            Iout = 1'b0;
            Ain = 1'b0;
            Gin = 1'b0;
            DPin = 1'b0;
            RdX = 1'b0;
            RdY = 1'b0;
            WrX = 1'b0;
            add_sub = 1'b0;
          end
            LOAD:
          begin
                    _Extern = 1'b1;
            Gout = 1'b0;
                        Iout = 1'b0;
```

```
        Ain = 1'b0;
        Gin = 1'b0;
                        DPin = 1'b0;
        RdX = 1'b0;
        RdY = 1'b0;
        WrX = 1'b1;
        add_sub = 1'b0;
    end
READ_Y:
    begin
                    _Extern = 1'b0;
        Gout = 1'b0;
                        Iout = 1'b0;
        Ain = 1'b1;
        Gin = 1'b0;
                        DPin = 1'b0;
        RdX = 1'b0;
        RdY = 1'b1;
        WrX = 1'b0;
        add_sub = 1'b0;
    end
READ_X:
    begin
                    _Extern = 1'b0;
        Gout = 1'b0;
                        Iout = 1'b0;
        Ain = 1'b1;
        Gin = 1'b0;
                        DPin = 1'b0;
        RdX = 1'b1;
        RdY = 1'b0;
        WrX = 1'b0;
        add_sub = 1'b0;
    end
ADD:
    begin
                    _Extern = 1'b0;
        Gout = 1'b0;
                        Iout = 1'b0;
        Ain = 1'b0;
        Gin = 1'b1;
DPin = 1'b0;
        RdX = 1'b1;
        RdY = 1'b0;
        WrX = 1'b0;
        add_sub = 1'b0;
    end
```

```
SUB:
   begin
                    _Extern = 1'b0;
      Gout = 1'b0;
                          Iout = 1'b0;
      Ain = 1'b0;
      Gin = 1'b1;
                          DPin = 1'b0;
      RdX = 1'b0;
      RdY = 1'b1;
      WrX = 1'b0;
      add_sub = 1'b1;
   end
MV:
   begin
                    _Extern = 1'b0;
      Gout = 1'b0;
                          Iout = 1'b0;
      Ain = 1'b0;
      Gin = 1'b1;
                          DPin = 1'b0;
      RdX = 1'b0;
      RdY = 1'b0;
      WrX = 1'b0;
      add_sub = 1'b0;
   end
WRITE_X:
   begin
                    _Extern = 1'b0;
      Gout = 1'b1;
                          Iout = 1'b0;
      Ain = 1'b0;
      Gin = 1'b0;
                          DPin = 1'b0;
      RdX = 1'b0;
      RdY = 1'b0;
      WrX = 1'b1;
      add_sub = 1'b0;
   end
DONE:
   begin
                    _Extern = 1'b0;
      Gout = 1'b0;
                          Iout = 1'b0;
      Ain = 1'b0;
      Gin = 1'b0;
                          DPin = 1'b0;
```

```
            RdX = 1'b0;
            RdY = 1'b0;
            WrX = 1'b0;
            add_sub = 1'b0;
        end
            DISP:
        begin
                        _Extern = 1'b0;
            Gout = 1'b0;
                            Iout = 1'b0;
            Ain = 1'b0;
            Gin = 1'b0;
                            DPin = 1'b1;
            RdX = 1'b1;
            RdY = 1'b0;
            WrX = 1'b0;
            add_sub = 1'b0;
        end
            ADDI:
        begin
                        _Extern = 1'b0;
            Gout = 1'b0;
                            Iout = 1'b1;
            Ain = 1'b0;
            Gin = 1'b1;
                            DPin = 1'b0;
            RdX = 1'b0;
            RdY = 1'b0;
            WrX = 1'b0;
            add_sub = 1'b0;
        end
            SUBI:
        begin
                        _Extern = 1'b0;
            Gout = 1'b0;
                            Iout = 1'b1;
            Ain = 1'b0;
            Gin = 1'b1;
                            DPin = 1'b0;
            RdX = 1'b0;
            RdY = 1'b0;
            WrX = 1'b0;
            add_sub = 1'b1;
        end
         /* TODO 3: complete output logic */
   endcase
end
```

```
/*
opcode encodings
000 - load
001 - move
010 - subtract
011 - add
100 - disp
101 - reserved
110 - subi
111 - addi
*/

        always@(posedge clk)
        begin

                case(state)
                        /* TODO 4: complete the next state logic */
                        IDLE: begin
                                        if(execute == 1 && operation == 3'b000) state <= LOAD;
                                        if(execute == 1 && operation == 3'b001) state <= READ_Y;
                                        if(execute == 1 && operation == 3'b010) state <= READ_X;
                                        if(execute == 1 && operation == 3'b110) state <= READ_X;
                                        if(execute == 1 && operation == 3'b111) state <= READ_X;
                                        if(execute == 1 && operation == 3'b011) state <= READ_Y;
                                        if(execute == 1 && operation == 3'b100) state <= DISP;
                           end
                        LOAD: begin
                                        state <= DONE; // always go to the DONE state at the next
clock tick
                                   end

                        READ_Y: begin
                                        if(operation == 3'b001) state <= MV;
                if(operation == 3'b011) state <= ADD;
                                   end

                        READ_X: begin
                                        if(operation == 3'b010) state <= SUB;
                if(operation == 3'b110) state <= SUBI;
                                        if(operation == 3'b111) state <= ADDI;
                                   end

                        ADD: begin
                                        state <= WRITE_X;

                         end
```

```
                SUB: begin
                                state <= WRITE_X;


                        end

                ADDI: begin
                                state <= WRITE_X;

                 end

                SUBI: begin
                                state <= WRITE_X;


                        end

                MV: begin
                                state <= WRITE_X;

                        end

                WRITE_X: begin
                                state <= DONE;

                        end

                DISP: begin
                                state <= DONE;

                        end

                DONE: begin

                                //back to idle if execute back to low
                                if(execute == 0) state <= IDLE;

                         end
                default: state <= IDLE;

        endcase

end //end always
```

```
endmodule
```

CODE TB FILE:

```
`timescale 1ns / 1ps

module l3_tb();

    /* clock and instruction control */
    reg clk, exec;

    /* instruction: to be generated in the testbench part */
    reg [0:10] instr;
    /*  4 different fields in an instruction */
    wire [0:2] opcode;  /* instr [0:2] */
    wire [1:0] reg_x;   /* instr [3:4] */
    wire [1:0] reg_y;   /* instr [5:6] */
    wire [3:0] imm;     /* instr [7:10] */

    /* control state machine outputs */
    wire extern, gout, iout, ain, gin, dpin, rdx, rdy, wrx, add_sub;

    /* state machine states */
    wire [3:0] smstate;

    /* latch a output */
    wire [3:0] a_out_data;
    /* latch g output */
    wire [3:0] g_out_data;
    /* latch dp output */
    wire [3:0] dp_out_data;

    /* mux 2 output */
    wire [3:0] mux2_output;

    /* adder output */
    wire [3:0] adder_out;



    l3_SM sm(.clk(clk),
            .execute(exec),
```

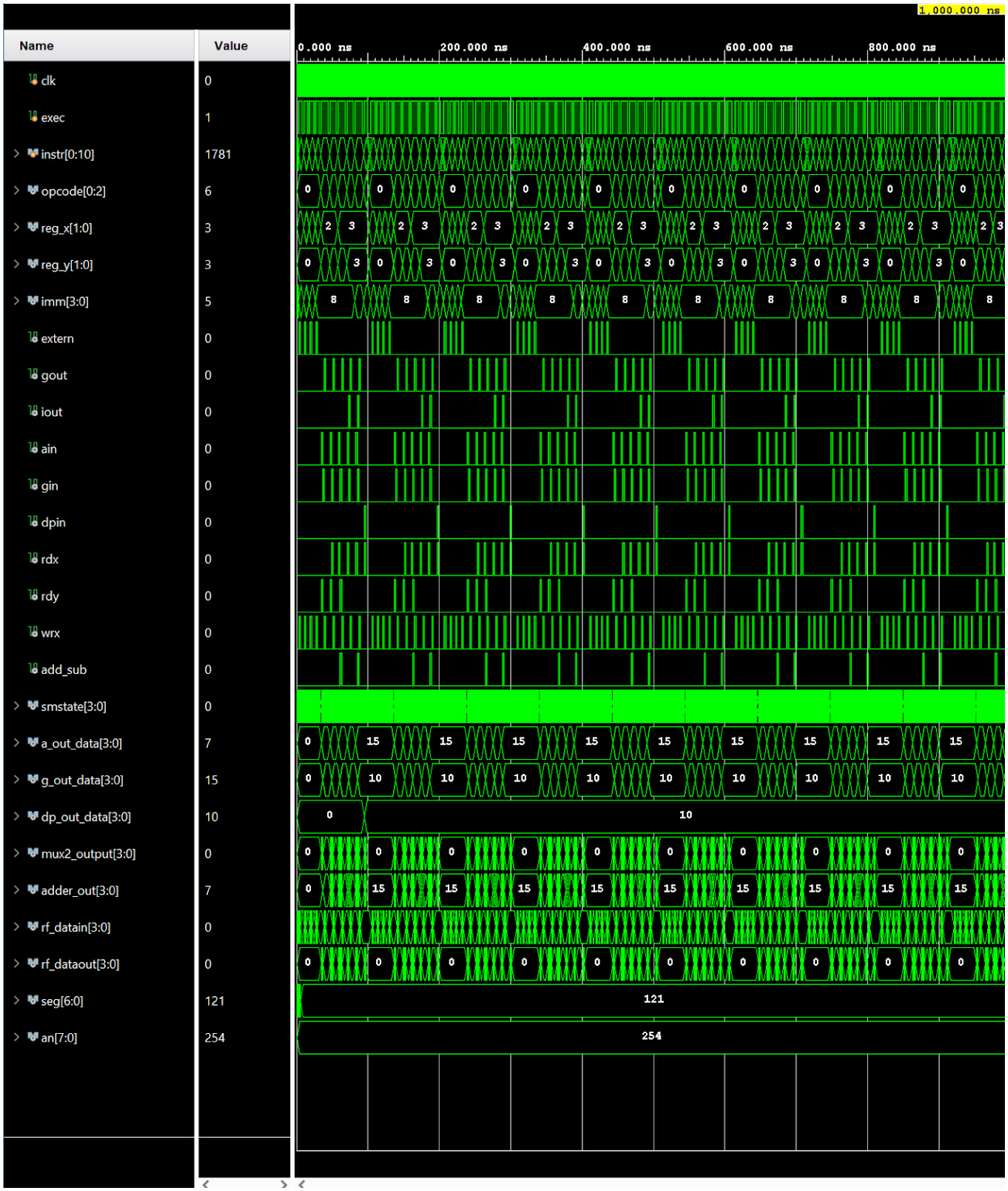
```
        .operation(opcode),
        ._Extern(extern),
        .Gout(gout),
        .Iout(iout),
        .Ain(ain),
        .Gin(gin),
        .DPin(dpin),
        .RdX(rdx),
        .RdY(rdy),
        .WrX(wrx),
        .add_sub(add_sub),
        .cur_state(smstate));

wire [3:0] rf_datain, rf_dataout;
wire [6:0] seg;
wire [7:0] an;

RF rf(.fpga_clk(clk),
      .DataIn(rf_datain),
      .AddrX(reg_x),
      .AddrY(reg_y),
      .RdX(rdx),
      .RdY(rdy),
      .WrX(wrx),
      .sm_state(smstate),
      .Dataout(rf_dataout),
      .seg(seg),
      .an(an));

 A a(.Ain(rf_dataout),
    .load_en(ain),
    .Aout(a_out_data));

 A g(.Ain(adder_out),
    .load_en(gin),
    .Aout(g_out_data));

       /* TODO 5: add DP latch */
        A dp(.Ain(rf_dataout),
    .load_en(dpin),
    .Aout(dp_out_data));

       /* TODO 6: add a 2-to-1 MUX, use # for parameterizing */
       mux_2_to_1 #(4) m2(.in0(rf_dataout), .in1(imm), .sel(iout), .mux_output(mux2_output));

 l2_adder adder(.in_A(a_out_data),
              .in_B(mux2_output), /* TODO 7: modify the connection of the 2nd operand in Adder */
```

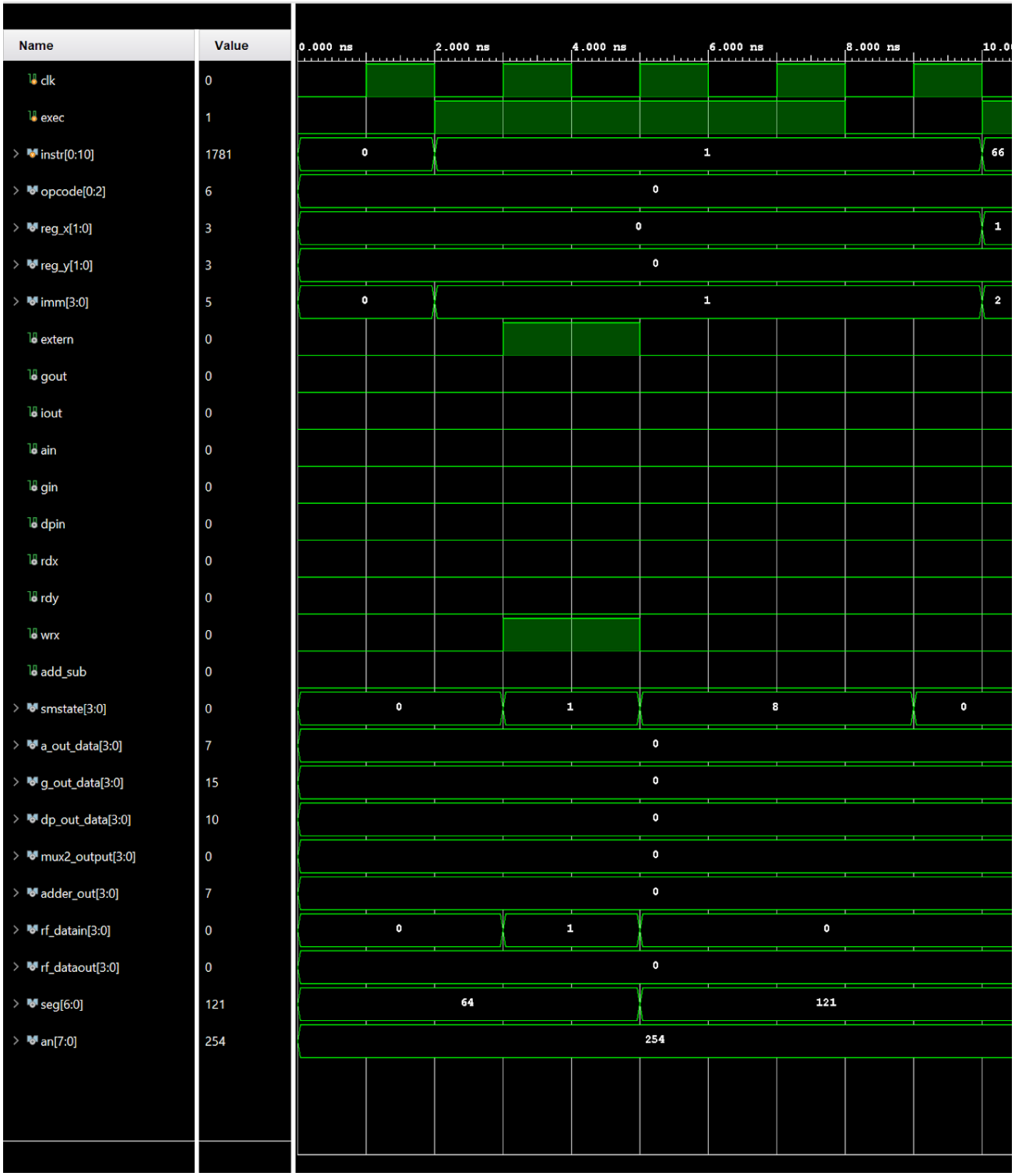
```
            .add_sub(add_sub),
            .adder_out(adder_out));

 modified_mux m1(.input_data(imm),
          .G_data(g_out_data),
          .Gout(gout),
          ._Extern(extern),
          .mux_output(rf_datain));


// clock generation
initial begin
    clk = 0;
end
always #1 clk = !clk;


assign opcode = instr[0:2];
assign reg_x = instr[3:4];
assign reg_y = instr[5:6];
assign imm = instr[7:10];

// insturction test bench
always
begin
    // initialization
    exec = 0;
    //  op = 3'b000, reg_x = 2'b00, reg_y = 2'b00, imm = 4'b0000
    instr = 11'b00000000000;
    #2;

    // load 1 to reg 0
    exec = 1;
    // op = 3'b000, reg_x = 2'b00, reg_y = 2'b00, imm = 4'b0001
    instr = 11'b00000000001;
    #6;
    // end of load 1 to reg 0
    // now, r0 = 1
    exec = 0;
    #2;

    // load 2 to reg 1
    exec = 1;
    // op = 3'b000, reg_x = 2'b01, reg_y = 2'b00, imm = 4'b0010
    instr = 11'b00001000010;
    #6;
    // end of load 2 to reg 1
```

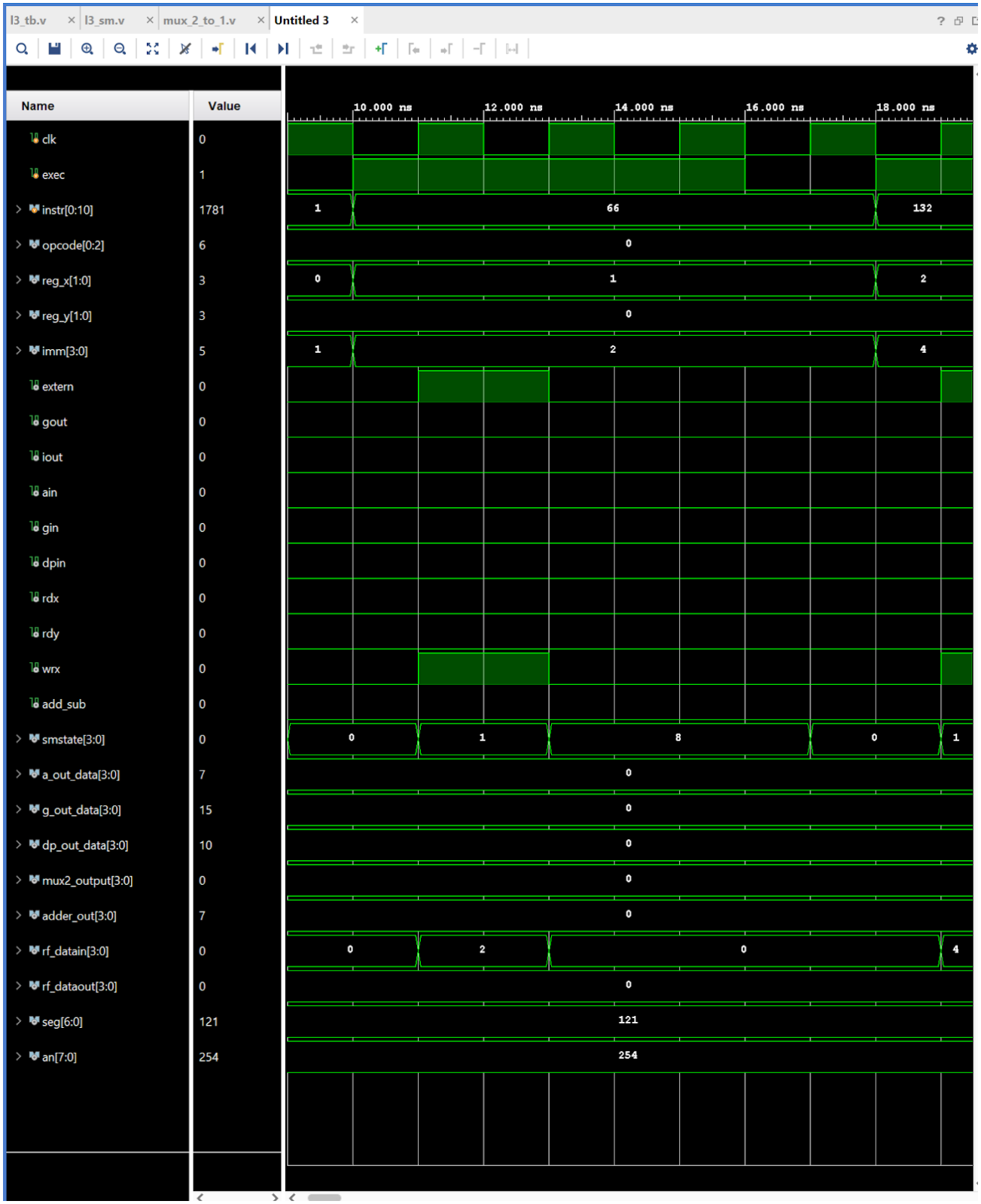
```
// now r1 = 2
exec = 0;
#2;

// load 4 to reg 2
exec = 1;
// op = 3'b000, reg_x = 2'b10, reg_y = 2'b00, imm = 4'b0100
instr = 11'b00010000100;
#6;
// end of load 4 to reg 2
// now r2 = 4
exec = 0;
#2;

// load 8 to reg 3
exec = 1;
// op = 3'b000, reg_x = 2'b11, reg_y = 2'b00, imm = 4'b1000
instr = 11'b00011001000;
#6;
// end of load 8 to reg 3
// now r3 = 8
exec = 0;
#2;

// move r3 to r2
exec = 1;
// op = 3'b001, reg_x = 2'b10, reg_y = 2'b11, imm = 4'b1000;
instr = 11'b00110111000;
#10;
// end of move r3 to r2
// now r2 = 8
exec = 0;
#2;

//  add: r2 = r2 + r1
exec = 1;
// op = 3'b011, reg_x = 2'b10, reg_y = 2'b01, imm = 4'b1000;
instr = 11'b01110011000;
#10;
// end of r2 = r2 + r1
// now r2 = 10
exec = 0;
#2;

//  sub: r3 = r3 - r0
exec = 1;
// op = 3'b010, reg_x = 2'b11, reg_y = 2'b00, imm = 4'b1000
```

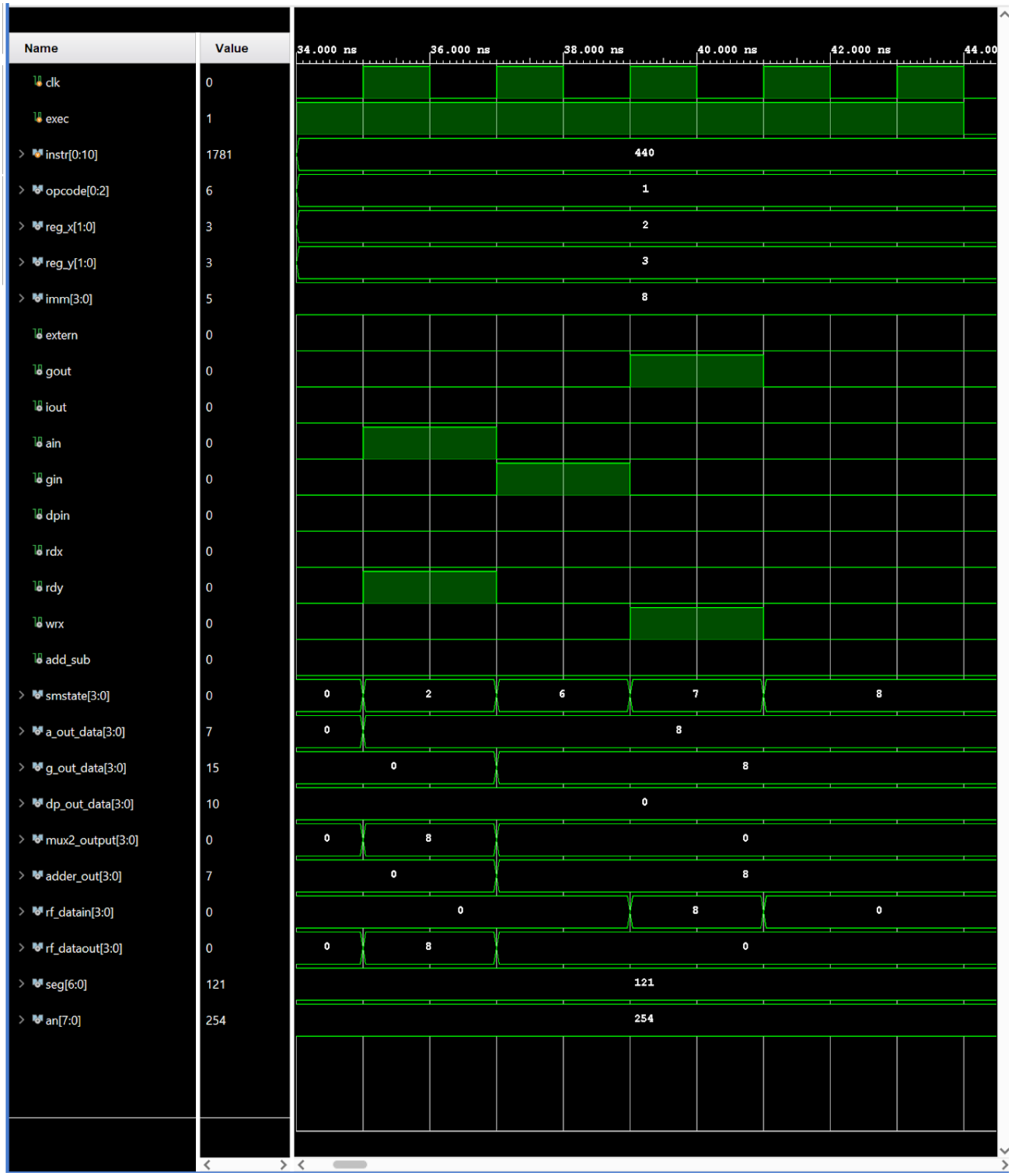
```
        instr = 11'b01011001000;
        #10;
        // end of r3 = r3 - r0
        // now r3 = 7
        exec = 0;
        #2;

            /* TODO 8: add more test sequences for addi, subi, and disp */

        // addi: r3 = r3 + 8
        exec=1;
        // op = 3'b111, reg_x = 2'b11, reg_y = 2'b11, imm = 4'b1000
        instr =  11'b11111111000;
        #10;
        // end of r3 = r3 + 8
        // r3 = 15
        exec = 0;
        #2;

        // subi: r3 = r3 - 5
        exec=1;
        // op = 3'b110, reg_x = 2'b11, reg_y = 2'b11, imm = 4'b0101
        instr =  11'b11011110101;
        #10;
        // end of r3 = r3 - 5
        // r3 = 10
        exec = 0;
        #2;

        // disp: r3
        exec=1;
        // op = 3'b100, reg_x = 2'b11, reg_y = 2'b11, imm = 4'b0000
        instr =  11'b10011110000;
        #6;
        // end of display r3
        // r3 = 10
        exec = 0;
        #2;
    end

endmodule
```

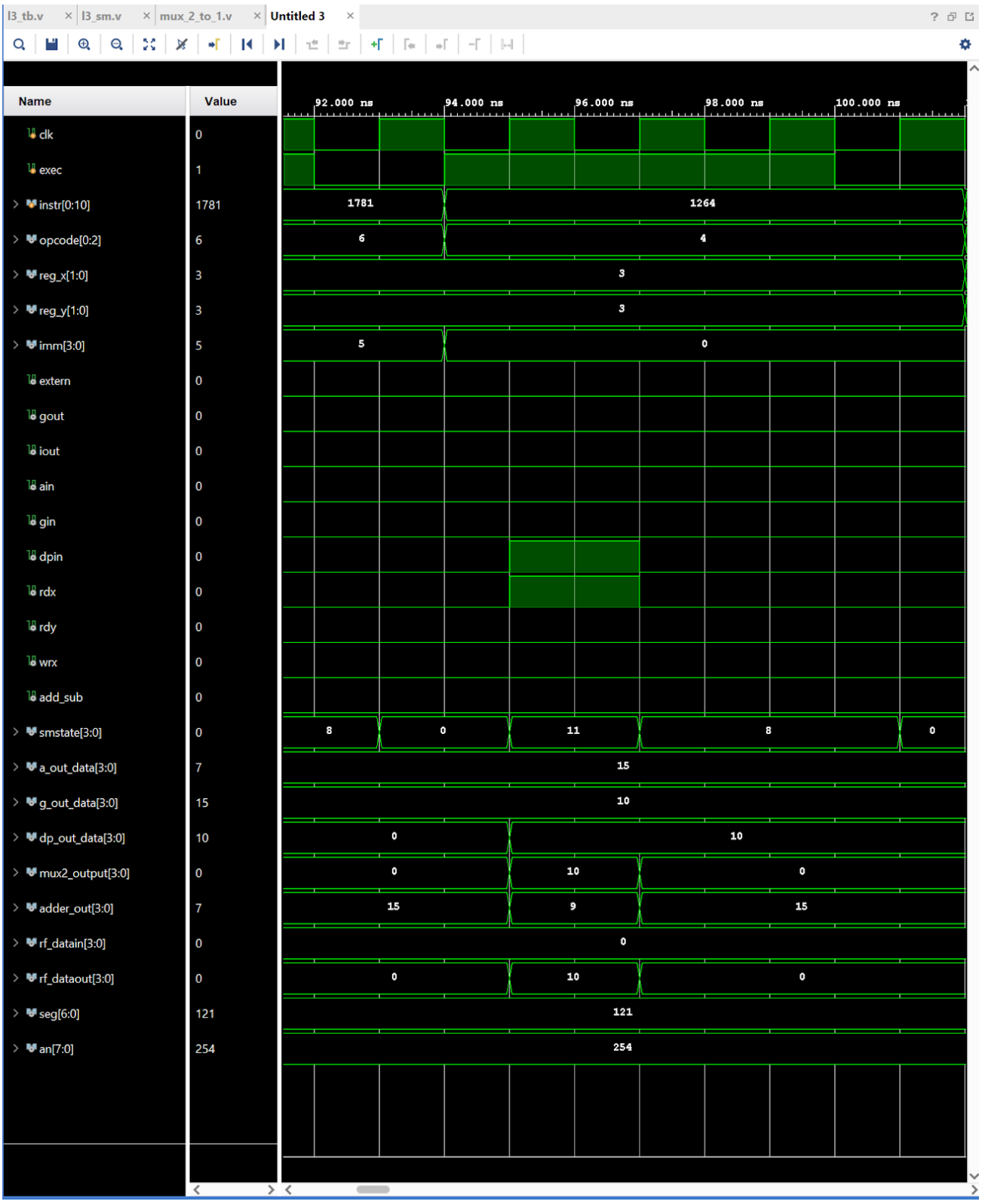
SCREENSHOTS:


Name
Value
1,000.000 ns
0.000 ns
200.000 ns
400.000 ns
600.000 ns
800.000 ns
clk 0
exec 1
instr[0:10] 1781
opcode[0:2] 6
reg_x[1:0] 3
reg_y[1:0] 3
imm[3:0] 5
extern 0
gout 0
iout 0
ain 0
gin 0
dpin 0
rdx 0
rdy 0
wrx 0
add_sub 0
smstate[3:0] 0
a_out_data[3:0] 7
g_out_data[3:0] 15
dp_out_data[3:0] 10
mux2_output[3:0] 0
adder_out[3:0] 7
rf_datain[3:0] 0
rf_dataout[3:0] 0
seg[6:0] 121
an[7:0] 254


Name
Value
0.000 ns
2.000 ns
4.000 ns
6.000 ns
8.000 ns
clk 0
exec 1
instr[0:10] 1781
opcode[0:2] 6
reg_x[1:0] 3
reg_y[1:0] 3
imm[3:0] 5
extern 0
gout 0
iout 0
ain 0
gin 0
dpin 0
rdx 0
rdy 0
wrx 0
add_sub 0
smstate[3:0] 0
a_out_data[3:0] 7
g_out_data[3:0] 15
dp_out_data[3:0] 10
mux2_output[3:0] 0
adder_out[3:0] 7
rf_datain[3:0] 0
rf_dataout[3:0] 0
seg[6:0] 121
an[7:0] 254
0
1
66
0
0
1
0
1
2
0
1
8
0
0
0
0
0
0
0
1
0
0
64
121
254


l3_tb.v
l3_sm.v
mux_2_to_1.v
Untitled 3
Name
Value
clk 0
exec 1
instr[0:10] 1781
opcode[0:2] 6
reg_x[1:0] 3
reg_y[1:0] 3
imm[3:0] 5
extern 0
gout 0
iout 0
ain 0
gin 0
dpin 0
rdx 0
rdy 0
wrx 0
add_sub 0
smstate[3:0] 0
a_out_data[3:0] 7
g_out_data[3:0] 15
dp_out_data[3:0] 10
mux2_output[3:0] 0
adder_out[3:0] 7
rf_datain[3:0] 0
rf_dataout[3:0] 0
seg[6:0] 121
an[7:0] 254
10.000 ns
12.000 ns
14.000 ns
16.000 ns
18.000 ns


Name
Value
34.000 ns
36.000 ns
38.000 ns
40.000 ns
42.000 ns
clk 0
exec 1
instr[0:10] 1781
opcode[0:2] 6
reg_x[1:0] 3
reg_y[1:0] 3
imm[3:0] 5
extern 0
gout 0
iout 0
ain 0
gin 0
dpin 0
rdx 0
rdy 0
wrx 0
add_sub 0
smstate[3:0] 0
a_out_data[3:0] 7
g_out_data[3:0] 15
dp_out_data[3:0] 10
mux2_output[3:0] 0
adder_out[3:0] 7
rf_datain[3:0] 0
rf_dataout[3:0] 0
seg[6:0] 121
an[7:0] 254
440
1
2
3
8
0 2 6 7 8
0 8
0 8
0
0 8 0
0 8
0 8 0
0 8 0
121
254


l3_tb.v
l3_sm.v
mux_2_to_1.v
Untitled 3
Name
Value
92.000 ns
94.000 ns
96.000 ns
98.000 ns
100.000 ns
clk 0
exec 1
instr[0:10] 1781
opcode[0:2] 6
reg_x[1:0] 3
reg_y[1:0] 3
imm[3:0] 5
extern 0
gout 0
iout 0
ain 0
gin 0
dpin 0
rdx 0
rdy 0
wrx 0
add_sub 0
smstate[3:0] 0
a_out_data[3:0] 7
g_out_data[3:0] 15
dp_out_data[3:0] 10
mux2_output[3:0] 0
adder_out[3:0] 7
rf_datain[3:0] 0
rf_dataout[3:0] 0
seg[6:0] 121
an[7:0] 254
1781
1264
6
4
3
3
5
0
8
0
11
8
0
15
10
0
10
0
10
0
15
9
15
0
0
10
0
121
254