Austin Petersen & Dylan Thornburg

SANTA CLARA UNIVERSITY	ECEN 122 Winter 2024	Hoeseok Yang

Laboratory #4: Fetch and execute instructions from memory

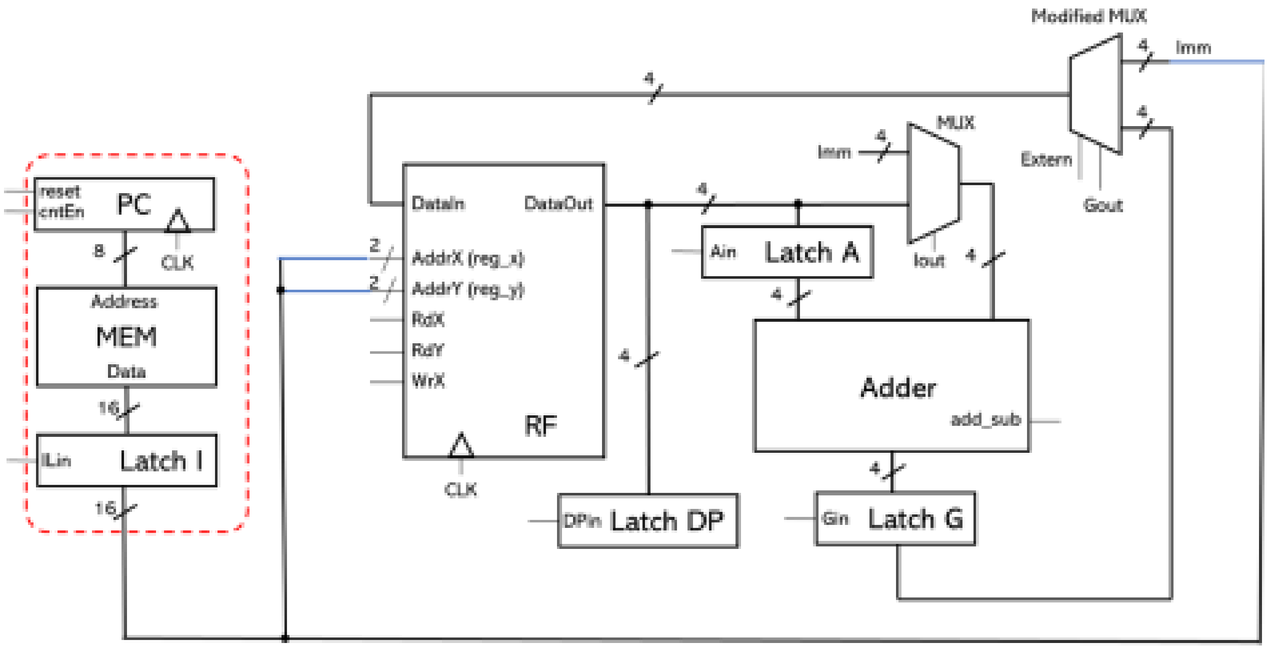
For lab section on Tuesday/Friday February 13/16, 2024

I. OBJECTIVES

Extend our CPU with a memory module that contains a sequence of instructions, and demonstrate the execution of those instructions.

PROBLEM STATEMENT

Continuing our evolution of the CPU we have implemented over the previous labs, we are going to create a system that no longer uses the testbench to provide instruction inputs but rather fetches instructions from a memory module. You will be given the memory module but you will need to connect it to your CPU design in such a way that a) it fetches instructions at the appropriate time, and b) each fetched instruction correctly controls the remaining CPU hardware. You will also create the "program" that will be loaded into memory.



An update to our diagram

The diagram above represents what will take the place of the "instruction input" in the design we have been working with so far. The TA will provide the PC (Program Counter), MEM (Memory), and IL (Instruction Latch) modules for you to instantiate; the IL is just another latch with a load enable (in this case, *ILin*). But in this case, it is going to be a 16-bit latch rather than the 4-bit ones that are instantiated in other parts of the design.

The MEM module for this lab will not have any control signals. Whatever data is present at the location indexed by the *Address* input will show up on the *Data* output. As soon as *Address* changes, *Data* will change to the value at this new location.

Writing and executing a program

To write a "program", or a sequence of instructions to execute, you will create a simple text file where each line is an "instruction". An instruction, for our purposes, is just a binary representation of how you would be setting the testbench inputs if you were still controlling your CPU as you did in the previous lab.

Your program is only going to be a handful of instructions, which will be fetched and executed one after another. Since the hardware support for fetching and executing instructions is going to be designed to continually fetch and execute, we will need a way to tell it to stop. From the last lab, you have a 3-bit code that specifies what instruction to execute, but we only have 7 of the 8 encodings specified. To support the idea of causing a program to end, you will define your currently unused opcode to be a HALT instruction. So, the encoding for opcode (operation in l4_SM.v) is now as follows:

- 3'b000: load
- 3'b001: move
- 3'b010: subtract
- 3'b011: add
- 3'b100: display
- 3'b101: HALT
- 3'b110: subi
- 3'b111: addi

When a HALT instruction is fetched, the "execution" of this instruction will cause the CPU to stop fetching further instructions and just sit until the system is reset.

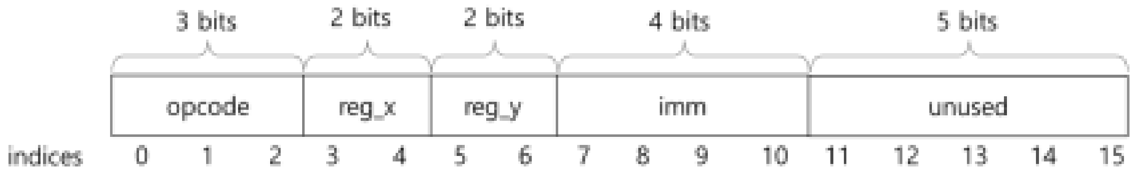
You will put the sequence of instructions (your program) into a file called "instructions.mem". The MEM module provided by your TA will know how to load the instructions into the memory so that your specified list of instructions will be fetched as the PC starts counting up from 0.

A few points about the format of the instructions.mem file:

- Since each memory location is going to be 16 bits of information, yet our "instructions" only need 11 bits, we will need to fill in the missing 5 bits with 0's. So, every line will have sixteen 1's and 0's.
- You can also put comments after the bit pattern, preceded by "//". So, you might have a line that looks like:

  1111100011000000 // op = 3'b111 (addi), reg_x = 2'b11, reg_y = 2'b00, imm = 4'b0110, and the last 5-bits are reserved (unused)

The 16-bit instruction formatting is as follows:



<u>Updates to the state machine</u>

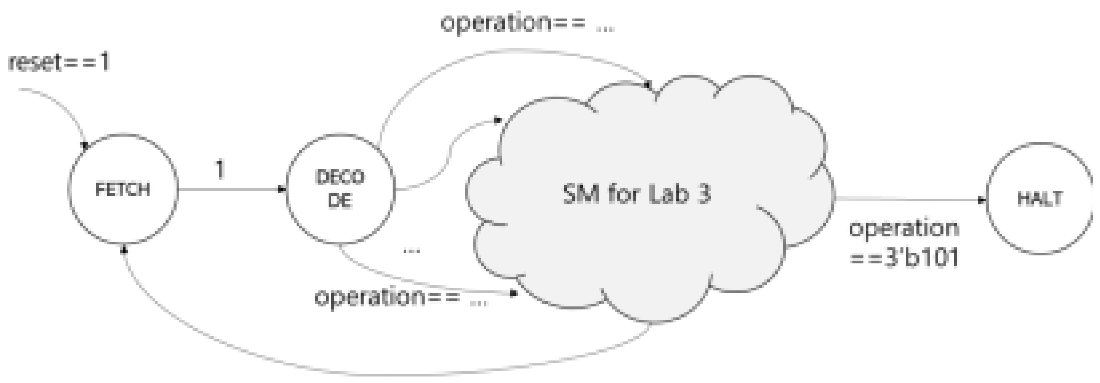
Up to now, you have a state machine that is sitting in the *IDLE* state until it sees the *execute* signal, and it has the *DONE* state that it will stay in unless it sees that *execute* is no longer asserted. The states in between represent the execution of the current instruction. What we need to do here is amend this flow so that we cause an instruction to be loaded into Latch I before we enter the states representing the execution of this instruction. We can repurpose our *IDLE* state to do this. And we'll rename it the *FETCH* state.

Keep in mind that the contents of the instruction latch (*Latch I*) are now going to be taking the place of the information that was coming from the testbench. So, depending on how you have implemented your state machine up to now, you may actually need to insert another state, after the *FETCH* state, that allows for responding to the contents of *Latch I* to determine how you traverse through the execution portion of your state machine. If you need to add such another state, think of it as a DECODE state.

When the execution of the current instruction is complete, we need to cause a new instruction to be loaded into *Latch I.* In other words, we need to return to the FETCH state. But somewhere along the line between "fetching" one instruction and fetching the next instruction, we need to cause the PC to increment by having the *countEn* signal assert. And we need to make sure the count enable only asserts for one cycle, so we do not skip any instructions.

Note that we do not really still need the *DONE* state at the end of each instruction execution. We just need to return to the *FETCH* state so we can get a new instruction. We also need to account for the "execution" of the HALT instruction, which should put the state machine into a state that does nothing, and can only be exited by asserting a reset signal.

In summary, from the state machine you used in the previous lab, the *IDLE* state will be replaced with *FETCH* and two more states (*DECODE* and *HALT*) are to be added. The DONE state should be removed from the state machine. You may reuse the most part of the previous state machine and the overall structure of the new state machine will look like the following diagram:



Resetting

In addition to providing a clock, your testbench will need to provide a *reset* signal. This signal should be connected to the reset port of the PC module (causing it to go to 0 when reset is asserted), and it will need to cause your state machine to go to its initial state. When the reset signal de-asserts, your state machine will then be able to proceed with instruction execution.

II. LAB PROCEDURE

(i) Once again, work with your lab partner to reconcile your pre-labs. In particular, decide which program you're going to execute, which will in turn dictate which instruction encodings you will need to work with. This will dictate how your instructions are decoded inside of your state machine implementation.

(ii) The person in your partnership who wrote the program that you chose in step (i) will be the Integrator for this lab and move to step (iv). The other person will be the Developer and perform step (iii).

(iii)The Developer will take the Integrator's state machine code they turned in for pre-lab and modify it to account for the agreed-upon changes to the state machine.

(iv)The Integrator will bring up Vivado and start working with the new elements provided by the TA. Namely, the module for the PC, the MEM module, and a latch to be used as the Instruction Latch. The TA will also provide a partially updated testbench file. You will need make sure the state machine instantiation matches the interface definition of the state machine that the Developer is providing, and that the correctly connections are made to the newly provided modules in the testbench.

(v) Once the updated state machine code is ready, import it into Vivado and start working to demonstrate that your program can be correctly executed. At this point you should be working together to debug issues that you come across.

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

III. 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?

We had issues regarding default register values and to solve it we loaded all registers with 0 at the start. We laos had problems instantiating new modules, but we solved this by looking the code we were given and eventually figuring it out. After this, we were able to get the testbench and state machine working without issue.

• 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.

We had some issues with the registers (mentioned previously) that we fixed the first time we demoed. The second time we didn't have any errors, but Rikesh did ask us some stuff regarding states that we had to think about. We eventually responded correctly to his questions.

• What would happen if you didn't implement the HALT instruction? What would your system likely have done?

If we did not have a halt instruction then the state machine would keep repeating the previous instructions.

Source Code for instructions:

```
//Load reg11 2 //reg11 = 2 loads immediate
//Addi reg00 reg11 2 //reg00 =4
//Subi reg01 reg00 2 //reg01 = 2
//Mv reg10 reg00 //reg10 = 4
//Add reg00 reg00 //reg00=8
//Sub reg01 reg01 //reg01 = 0
//Disp reg00 //display 8
//Halt
0000000000000000
0000100000000000
0001000000000000
0001100000000000
//loading zero to all reg

//actual instruction code:
1110001010000000
```

```
1100100001000000
0001100001000000
0011000000000000
0110000000000000
0100101000000000
1000000000000000
1010000000000000
```

SM Source Code:

```
module l4_SM(input clk,
                                input reset,
                                input [0:2] 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 RdX,
                                output reg RdY,
                                output reg WrX,
                                output reg add_sub,
                                output reg pc_en,   // PC enable
                                output reg ILin,    // Latch I enable
                                output [3:0] cur_state);

        /* in total, 13 states are defined */
        parameter FETCH      =  4'b0000;
        parameter LOAD                 =  4'b0001;
        parameter READ_Y     =  5'b0010;
        parameter READ_X     =  4'b0011;
        parameter ADD                  =  4'b0100;
        parameter SUB                  =  4'b0101;
        parameter MV                   =  4'b0110;
        parameter WRITE_X =  4'b0111;
        parameter ADDI       =  4'b1001;
        parameter SUBI       =  4'b1010;
        parameter DISP       =  4'b1011;
        parameter DECODE     =  4'b1100;
        parameter HALT                 =  4'b1110;

        reg [3:0] state = FETCH; // initial state being FETCH
        assign cur_state = state;



        /* TODO #3: complete the following always statement */
```

```
always@(*)
begin
  case(state)
     FETCH:
                    /* TODO #3: control signal output for FETCH */
        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;
           pc_en = 1'b1;
           ILin = 1'b1;
        end
     DECODE:
                    /* TODO #3: control signal output for DECODE */
        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;
           pc_en = 1'b0;
           ILin = 1'b0;
        end
         LOAD:
                    /* TODO #3: control signal output for 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;
        pc_en = 1'b0;
        ILin = 1'b0;
    end
READ_Y:
                /* TODO #3: control signal output for 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;
        pc_en = 1'b0;
        ILin = 1'b0;
    end
READ_X:
                /* TODO #3: control signal output for 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;
        pc_en = 1'b0;
        ILin = 1'b0;
    end
ADD:
                /* TODO #3: control signal output for 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;
        pc_en = 1'b0;
        ILin = 1'b0;
    end
SUB:
                /* TODO #3: control signal output for 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;
        pc_en = 1'b0;
        ILin = 1'b0;
    end
                /* TODO #3: control signal output for MV */
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;
        pc_en = 1'b0;
        ILin = 1'b0;
    end
                /* TODO #3: control signal output for WRITE_X */
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;
        pc_en = 1'b0;
        ILin = 1'b0;
    end
                /* TODO #3: control signal output for HALT */
HALT:
    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;
        pc_en = 1'b0;
        ILin = 1'b0;
    end
DISP:
                /* TODO #3: control signal output for 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;
        pc_en = 1'b0;
        ILin = 1'b0;
    end
ADDI:
                /* TODO #3: control signal output for 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;
                pc_en = 1'b0;
                ILin = 1'b0;
            end
        SUBI:
                    /* TODO #3: control signal output for 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;
                pc_en = 1'b0;
                ILin = 1'b0;
            end
      endcase
    end

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

    /* TODO #2
      based on the state diagram you drew in the pre-lab
      complete the following always statement in which all state transitions are specified */

    always@(posedge clk or posedge reset)
    begin
            if (reset==1) state <= FETCH;
            else
            case(state)

                    FETCH:  begin
                            state <= DECODE;
                      end
                DECODE: begin
                                /* TODO #2: in DECODE, if opcode is 000, switch to LOAD
                                                complete all other transitions ... */
                                if(operation == 3'b000) state <= LOAD;
                                if(operation == 3'b001) state <= READ_Y;
                                if(operation == 3'b010) state <= READ_X;
                                if(operation == 3'b110) state <= READ_X;
                                if(operation == 3'b111) state <= READ_X;
                                if(operation == 3'b011) state <= READ_Y;
                                if(operation == 3'b100) state <= DISP;
                                if(operation == 3'b101) state <= HALT;

                        end
                LOAD: begin
                                state <= FETCH;
                        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 <= FETCH;
                              end

                        DISP: begin
                                    state <= FETCH;
                              end
                        HALT: begin
                              state <= HALT;
                        end

                        default: state <= FETCH;

                  endcase

            end //end always



endmodule
```

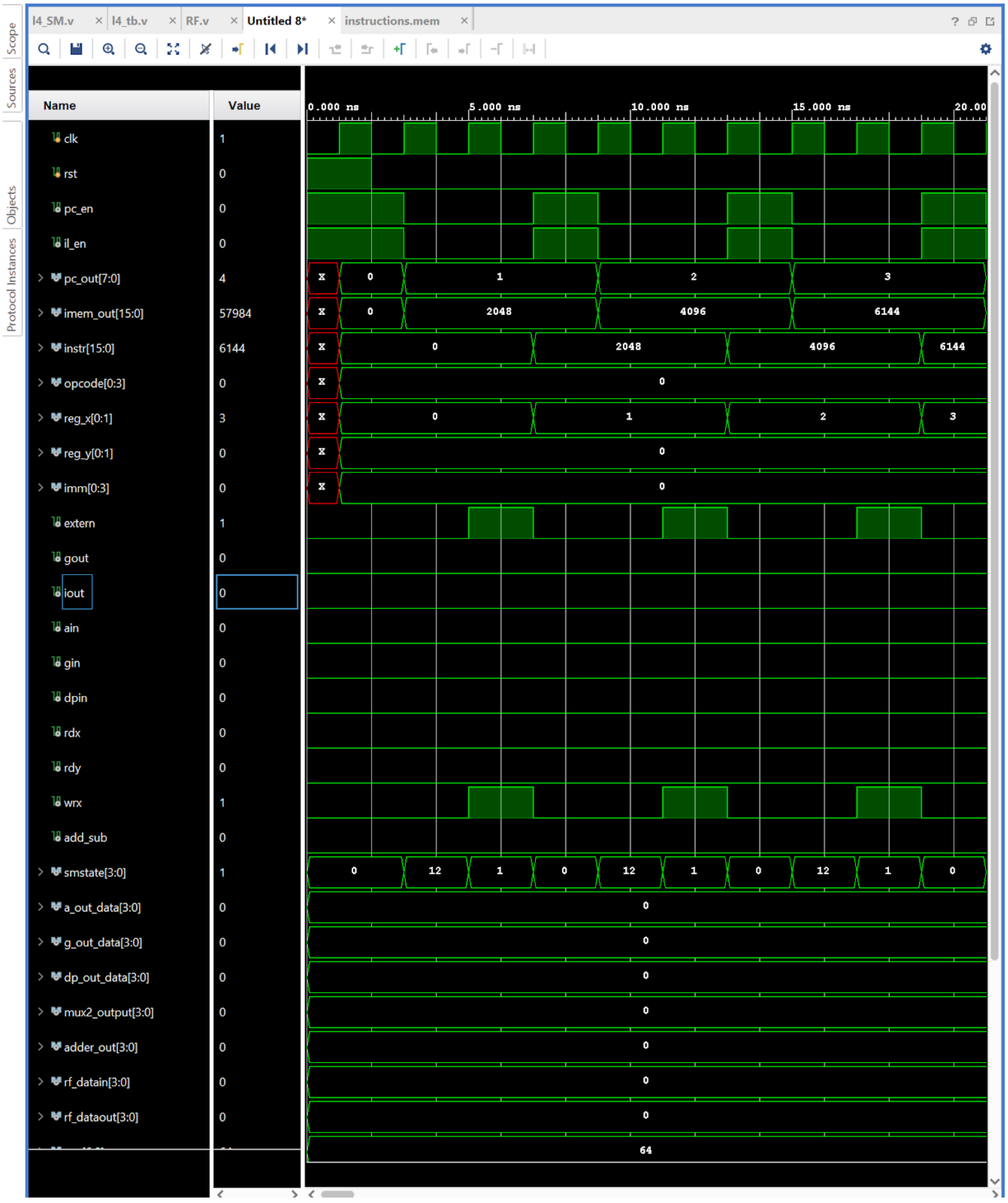
TB Source Code:

```
`timescale 1ns / 1ps

module l4_tb();

   /* clock and instruction control */
   reg clk, rst;
   /* PC enable */
   wire pc_en;  // control logic -> PC
   /* Instruction latch enable */
   wire il_en;  // control logic -> Latch I
   /* PC out: current instruction address */
   wire [7:0] pc_out;  // PC -> MEM
   /* Instruction memoery out */
   wire [15:0] imem_out; // MEM -> instr
   /* instruction: to be generated in the testbench part */
   wire [15:0] instr;

   /*  4 different fields in an instruction */
   wire [0:3] opcode;  /* instr [15:13] */
   wire [0:1] reg_x;   /* instr [12:11] */
   wire [0:1] reg_y;   /* instr [10:9] */
   wire [0:3] imm;     /* instr [8:10] */

   /* 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;



l4_SM sm(.clk(clk),
        .reset(rst),
        .operation(opcode),
        ._Extern(extern),
        .Gout(gout),
        .Iout(iout),
        .Ain(ain),
        .Gin(gin),
        .DPin(dpin),
        .RdX(rdx),
        .RdY(rdy),
        .WrX(wrx),
        .add_sub(add_sub),
        .pc_en(pc_en),
        .ILin(il_en),
        .cur_state(smstate));

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

/* TODO #1
   1) instantiate the three modules: l4_PC, l4_MEM, and A (16-bit latch)
   2) connect them each other properly using the declared signals (pc_out, imem_out, and isntr)
   3) connect the control logic singals (rst, pc_en, and il_en) properly to PC and Latch I.
*/
/* program counter */
l4_PC pc (
        .clk(clk),
        .countEn(pc_en),
        .reset(rst),
        .Address(pc_out));

/* instruction memory */
l4_MEM imem(
        .count(pc_out),
        .Data(imem_out));

/* instruction latch */
A #(.bit_width(16)) ilatch(.Ain(imem_out),
      .load_en(il_en),
      .Aout(instr));

/* end of TODO #1 */

/* instruction decode */
assign opcode = instr[15:13];
assign reg_x = instr[12:11];
assign reg_y = instr[10:9];
assign imm = instr[8:5];

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 #(.bit_width(4)) a(.Ain(rf_dataout),
     .load_en(ain),
     .Aout(a_out_data));

 A #(.bit_width(4)) g(.Ain(adder_out),
     .load_en(gin),
     .Aout(g_out_data));

 A #(.bit_width(4)) dp(.Ain(rf_dataout),
      .load_en(dpin),
      .Aout(dp_out_data));
```

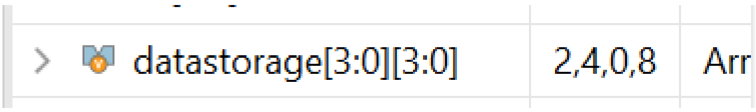
```
    l2_adder adder(.in_A(a_out_data),
                   .in_B(mux2_output),
                   .add_sub(add_sub),
                   .adder_out(adder_out));

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

    mux_2_to_1     #(.bit_width(4))
                   m2( .in0(rf_dataout),
                   .in1(imm),
                   .sel(iout),
                   .mux_output(mux2_output));

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

   // reset generation
   always begin
      rst = 1; #2;
      rst = 0; #200;
   end


endmodule
```

SCREENSHOTS:

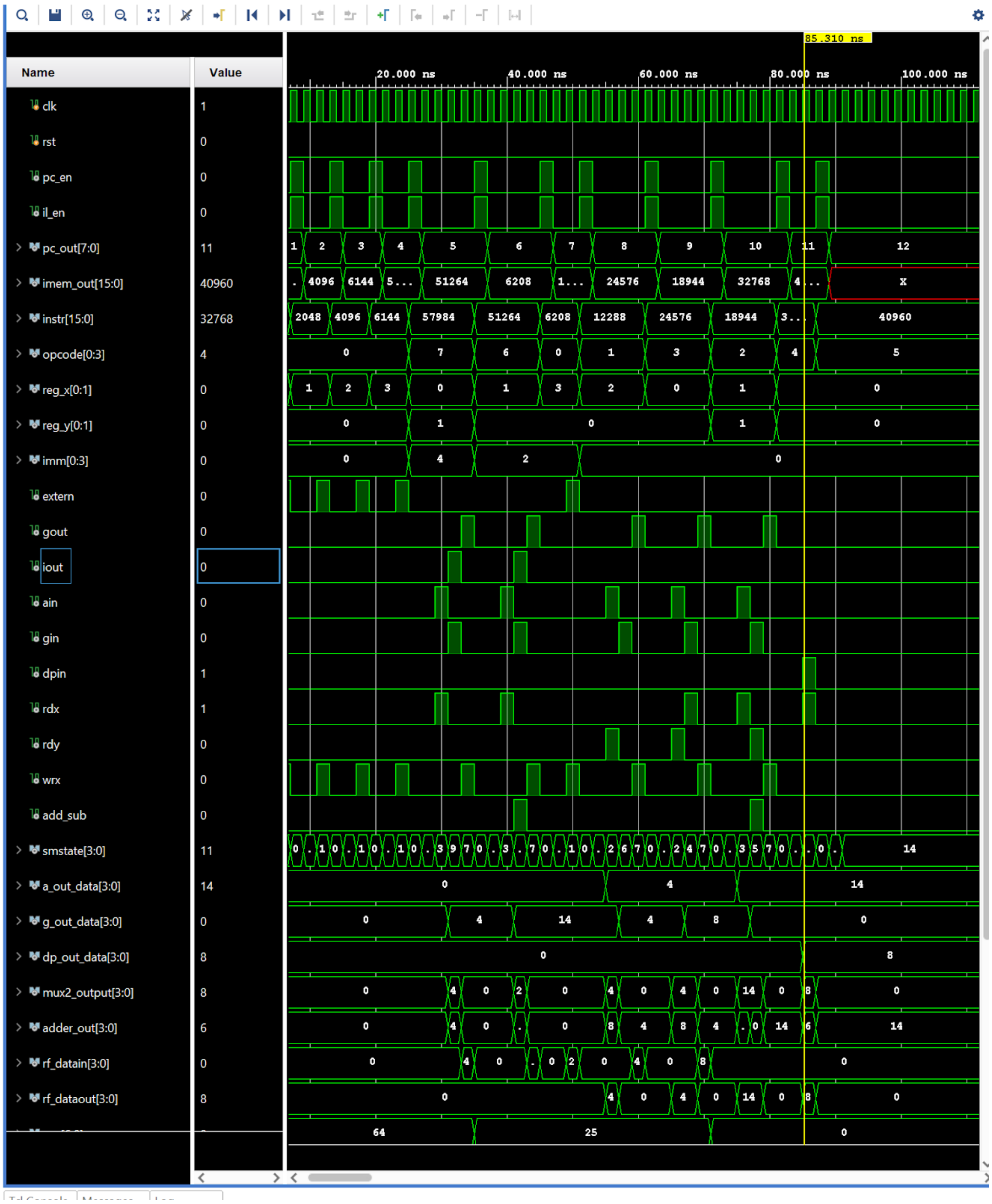


Name	Value
clk	1
rst	0
pc_en	0
il_en	0
pc_out[7:0]	4
imem_out[15:0]	57984
instr[15:0]	6144
opcode[0:3]	0
reg_x[0:1]	3
reg_y[0:1]	0
imm[0:3]	0
extern	1
gout	0
iout	0
ain	0
gin	0
dpin	0
rdx	0
rdy	0
wrx	1
add_sub	0
smstate[3:0]	1
a_out_data[3:0]	0
g_out_data[3:0]	0
dp_out_data[3:0]	0
mux2_output[3:0]	0
adder_out[3:0]	0
rf_datain[3:0]	0
rf_dataout[3:0]	0

```
pc_out[7:0]:     X | 0 | 1 | 2 | 3
imem_out[15:0]:  X | 0 | 2048 | 4096 | 6144
instr[15:0]:     X | 0 | 2048 | 4096 | 6144
opcode[0:3]:     X | 0
reg_x[0:1]:      X | 0 | 1 | 2 | 3
reg_y[0:1]:      X | 0
imm[0:3]:        X | 0
smstate[3:0]:    0 | 12 | 1 | 0 | 12 | 1 | 0 | 12 | 1 | 0
a_out_data[3:0]:   0
g_out_data[3:0]:   0
dp_out_data[3:0]:  0
mux2_output[3:0]:  0
adder_out[3:0]:    0
rf_datain[3:0]:    0
rf_dataout[3:0]:   0
                   64
```



datastorage[3:0][3:0]	2,4,0,8	Arr



Name	Value
clk	1
rst	0
pc_en	0
il_en	0
pc_out[7:0]	11
imem_out[15:0]	40960
instr[15:0]	32768
opcode[0:3]	4
reg_x[0:1]	0
reg_y[0:1]	0
imm[0:3]	0
extern	0
gout	0
iout	0
ain	0
gin	0
dpin	1
rdx	1
rdy	0
wrx	0
add_sub	0
smstate[3:0]	11
a_out_data[3:0]	14
g_out_data[3:0]	0
dp_out_data[3:0]	8
mux2_output[3:0]	8
adder_out[3:0]	6
rf_datain[3:0]	0
rf_dataout[3:0]	8