Hannah Bajakian and Dylan Thornburg

SANTA CLARA UNIVERSITY	**ECEN 122 Winter 2024**	**Hoeseok Yang**
Laboratory #5: Formatted instructions and support for stores		
For lab sections on Thursday/Friday February 20/23, 2024		

I. OBJECTIVES

Add structure to our instruction formatting, and add support for a store instruction.

PROBLEM STATEMENT

Up to now, the data on which our instructions have executed has been held solely in the register file. But register files tend to be fairly small, in terms of the number of registers, which means that we can only manage a small number of values if we do not have more locations to hold onto data. That is where memory comes in.

In the last lab, we added a memory module for holding instructions. In this lab we are going to add the support to use this memory to also hold onto data values as well. We will need to add a *store* instruction to write data to memory. And we will need to modify our existing load instruction to read data from memory rather than taking data from the instruction itself.

Because our memory module stores 16-bit values, we are also going to widen our datapath (the width of the registers and the width of the ALU) to 16 bits.

And, now that we are using 16-bit instructions, we are going to expand our register file to contain 16 registers. Since that will shake things up a bit relative to your previous formatting, we will take this opportunity to standardize on a new format that will be similar to what we have seen in class.

ECEN 122 Lab 5

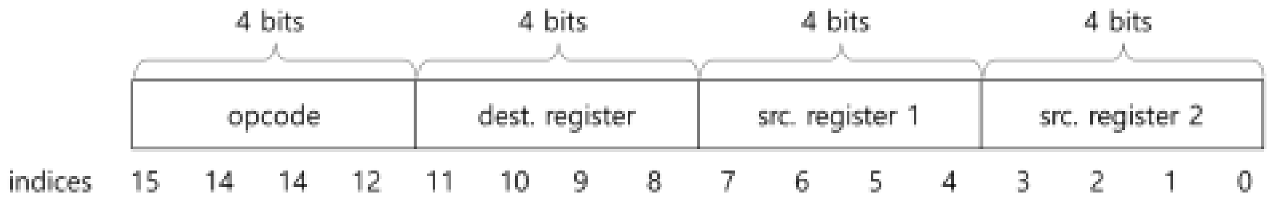
Specifying three registers, or two registers and an immediate value

With the elbow room that we now have with 16-bit instructions, we are going to do two things. First, we are going to expand our register file to contain 16 registers. That, in turn, means that we will need a 4-bit index to specify/address any given register. Also, we are going to move away from our previous instruction format, which only allowed us to specify two different registers and thus forced us to “overload” one of the addresses as both a source and a destination register, and now use three register identifiers: two source registers and a separate identifier for the destination of the instruction.

With three 4-bit fields as register identifiers, that consumes 12 bits out of our 16-bit instruction. The remaining 4 bits we will dedicate to specifying the opcode, the encoding that indicates what the instruction should actually be doing.

But some of our instructions still want to have immediate values (like *addi*/*subi*). Rather than implementing a different instruction format (R-format vs. I-format), we are going to limit ourselves to a 4-bit immediate value, and just treat one of the fields as an immediate value for the instructions that use an immediate value.

That leads us to the following instruction format:



Note that this all implies some important changes to our datapath:

- The interface to the register file is going to change, by adding a third address port and a second read data port.
- Since we will have two data read ports out of the register file, we will need two separate buses connecting to the ALU.
- Since we are now dealing with 16-bit data values, the 4-bit "immediate" data from our instruction will need to be sign extended into a 16-bit value before it is fed into the mux that feeds the second input port of the ALU.

Because we are introducing a register file with the ability to read our two source operands concurrently, out of two data ports, we no longer need the "read" control signals we had to apply to the register file. This new register file will just always read out whatever values are stored in the registers specified by the two read addresses.

Note also that we could get away with removing the staging latches, *a* and *g*. But that would then mean making substantial changes to the control state machine. We will remove these in future labs, but we will keep them in place for now to minimize the amount of work that needs to be done to get this lab working.

Also, we want to keep the display instruction that was previously implemented, and have it work the same as before. So, the bus coming out of the RF's *DataOut1* port should be fed to the display staging latch, in addition to staging latch A.

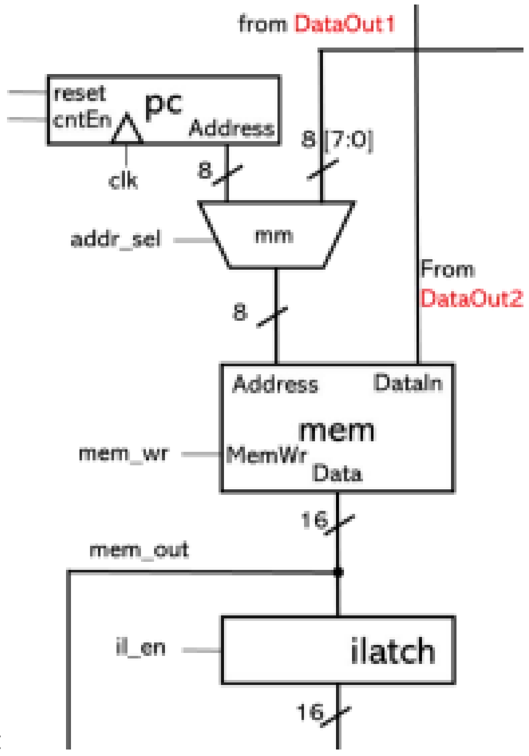
Given the changes described above, our new datapath is now going to look like the datapath depicted in a separate page at the end of this material.

<u>Adding a store instruction, and modifying the load instruction</u>

Now let us turn our attention to what we need to support the use of memory for storing and loading data operands. For a *store* operation, we need to provide an address to identify the location in memory that we want to update, and we need to provide the data that we want to store at that location. Both of these values (the address and the data to store) will come from the register file. Specifically, the address will be supplied by source register 1 (i.e., the value we read out of port *DataOut1*) and the data to store will be supplied by source register 2 (i.e., the value we read out of port *DataOut2*) .

The *load* operation is going to have some symmetry with store: the address of the location from which we want to load will be supplied by source register 1. But now data will be going to the register file. So, the data coming from source register 2 will be unused. The destination register specified by the load operation will be the register that is written with the data that is supplied by memory.

Note that we are using the same memory module that is holding our instructions. And for instruction fetches, the address supplied to the memory module is coming from the PC. So, we have two different addresses (the PC for instructions, source register 1 for load/store operations), but only one address port into the memory module. So, we will need a mux to select between these two addresses, and we will need to make sure we are selecting the right value to pass thru the mux depending on what we
are trying to do. All of the above description leads to a picture that looks like this:



ECEN 122 Lab 5
Note the two control signals: *addr_sel* and *mem_wr*. For the address mux (*mm*), if you connect the PC to port 0, and source register 1 to port 1, then *addr_sel* should be asserted when performing either a *load* or a *store* operation. The *mem_wr* signal should be asserted when performing a store operation.

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

It should be possible to correctly control this updated datapath with just a few changes to your state machine:

- Your opcode will be coming from slightly different bits from the Instruction latch (*ilatch*), but if you use the same instruction encodings then it should be fine.
- The *RdX* and *RdY* control signals have gone away, but you do not really have to prune them from your state machine, you can just leave them unconnected.
- The above change would impact the implementation of the move operation. But we no longer need the *move* operation. Given that our instructions can now specify separate source and destination registers, the move operation can be accomplished by using the *addi* (or *subi*) instruction and just setting the immediate field to 0. So, you can completely remove the *move* operation.
- You will need to define a new opcode encoding for the *store* operation, and then implement the correct transitions thru your state machine. Note that the *store* operation needs to assert both *addr_sel* and *mem_wr*. Also note that you could reuse the operation encoding for move since it is going away, or you can allocate a new encoding.
- The load instruction will need to assert the *addr_sel* signal to make sure you load from the correct address.

Change to the writing of a program

For this lab, we will be using a text file (mem.txt) to specify and load a program into memory. But rather than specifying the instructions in binary, we will now specify them in hex. This works out reasonably well since the four fields in our instruction format are all 4-bit quantities, so they all align exactly with a hex digit.

PRE-LAB

(i) Identify the opcode you will be using for the store instruction; whether you are reusing the opcode previously used for move, or whether you are allocating a new encoding.

(ii) Draw out an updated state diagram that accounts for the store instruction, and any other relevant changes.

(iii) Write a program, to load and execute in lab, that accomplishes the following: a. Load the value at addresses 48, 49, and 50. We will call these *Val1*, *Val2*, and *Val3*. b. Add *Val1* and *Val2* and store the result at the address indicated by *Val3*.
c. Display the sum of *Val1*, *Val2*, and *Val3*.
Make sure to end your program with a halt instruction.

The TA will be initializing memory with known values at address locations 48, 49, and 50, so that the results of the program can be checked in lab.

II. LAB PROCEDURE

(i) Once again, work with your lab partner to reconcile your pre-labs. In particular, decide which program you are going to execute, which will in turn dictate which instruction encodings you will need to work with and which state machine you will use.

(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 from the previous lab and modify it to account for the agreed-upon changes to the state machine.

(iv)The Integrator will bring up Vivado and make all the necessary changes. The TA will provide the modules you need, as well as a partially connected testbench file. You will need to make sure all the control signals are properly connected, and the values coming out of the instruction latch are connected appropriately.

(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.

(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 an issue initially where our waveform was all red X's and we went through each "to do" several times without being able to figure out what the issue was until Dr. Yang walked us through each signal and we realized that the l5_mem.v file was not reading the correct mem.txt file and that it needed to be reading the instr.mem file instead.

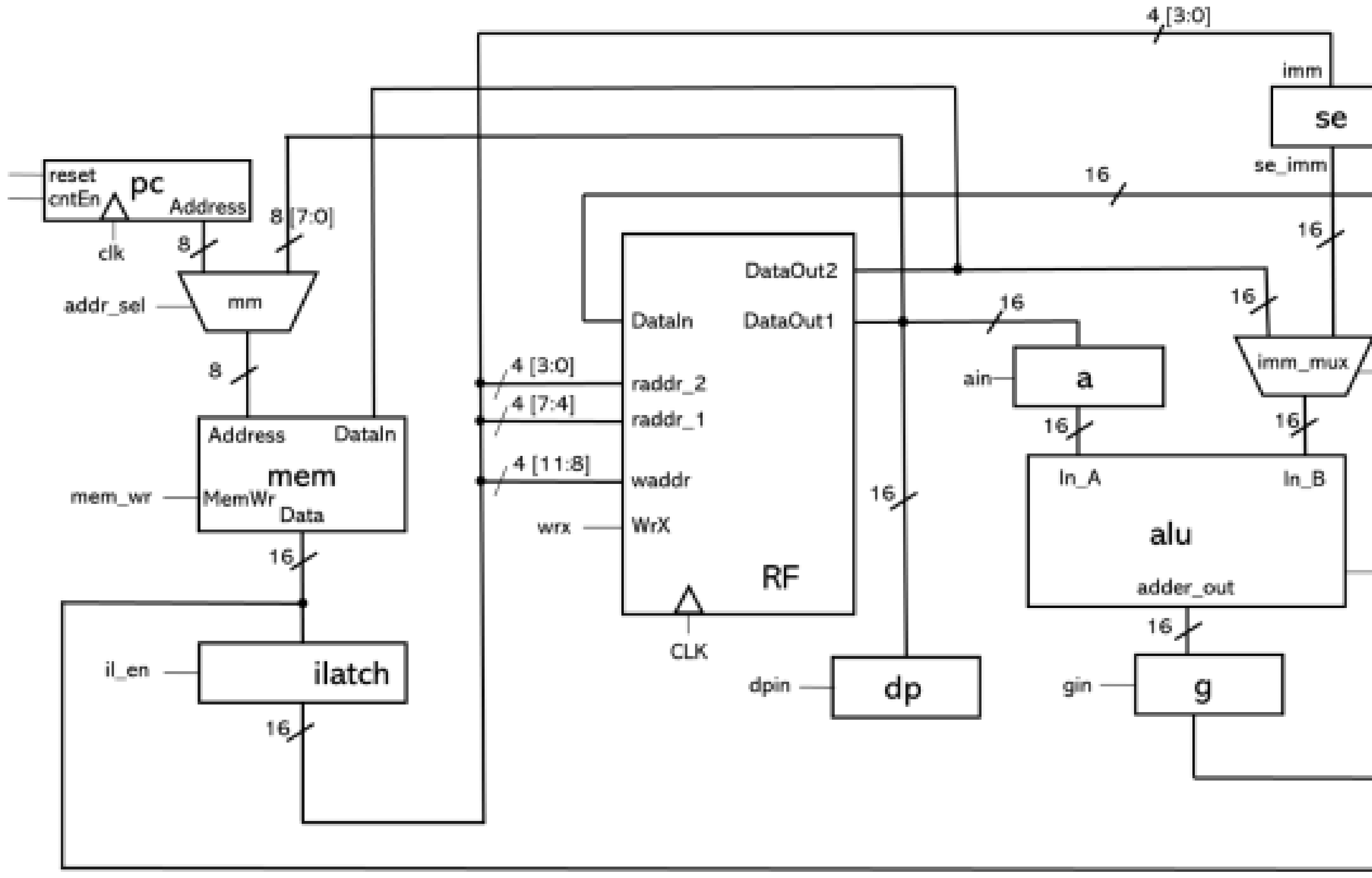
- **Did any problems arise when demonstrating for the TA? In other words, did the TA ask you to demonstrate something that you did not think of yourself? What was the scenario that you were asked to demonstrate? Provide some thoughts about why you did not think of this yourself.**

The only issue that we ran into during our first demo was a misunderstand that we would be able to "show" the store operation happening at memory address 50. In reality, as Dr. Yang explained, we cannot see that happening. Instead, we altered our mem.txt file to load the value stored at memory location 50 into register 0 so at the end of the instruction set, the same value is stored in r0 as in Val2.

● **This design uses memory addresses directly from registers, rather than allowing for an offset to be added to a register value. How would you modify the datapath to allow for an offset, particularly for a store operation? How would this impact your state machine design?**

To allow for an offset to be added to a register value to access memory addresses, we would need to implement an immediate in the address selection mux. We could use a mux to connect the immSel signal as the input for addr_sel signal so that when it is asserted it provides the immediate offset value.

ECEN 122 Lab 5



[Revised datapath for Lab 5]

REPORT

State Machine Code:

```
module l5_SM(input clk,
                          input reset,
                          input [3:0] operation,
                          output reg _Extern,
                          output reg Gout,
                          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,
                          output reg ILin,
                          output reg rf_sel,
                          output reg sw_sel,
                          output reg MemWr,
                          output reg AddrSel,
                          output [3:0] cur_state);

    // state definitions
        parameter FETCH        =  4'b0000;
        parameter DECODE       =  4'b0001;
        parameter LOAD         =  4'b0010;
        parameter READ_Y       =  4'b0011;
        parameter READ_X       =  4'b0100;
        parameter ADD          =  4'b0101;
        parameter SUB          =  4'b0110;
        parameter MV           =  4'b0111;
        parameter WRITE_X      =  4'b1000;
        parameter ADDI         =  4'b1001;
        parameter SUBI         =  4'b1010;
        parameter DISP         =  4'b1011;
        parameter HALT         =  4'b1100;
        parameter STORE        =  4'b1101;

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


        always@(*)
        begin
           case(state)
               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;
                       rf_sel = 1'b0;     /* TODO #4: complete the control logic output */
                  sw_sel = 1'b0;  /* TODO #4: complete the control logic output */
                  MemWr = 1'b0;   /* TODO #4: complete the control logic output */
                  AddrSel = 1'b0; /* TODO #4: complete the control logic output */
                   end
               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;
                       rf_sel = 1'b0; /* TODO #4 */
                       sw_sel = 1'b0; /* TODO #4 */
                       MemWr = 1'b0; /* TODO #4 */
                       AddrSel = 1'b0; /* TODO #4 */
                   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;
                       pc_en = 1'b0;
                       ILin = 1'b0;
                       rf_sel = 1'b0; /* TODO #4 */
                       sw_sel = 1'b0; /* TODO #4 */
                       MemWr = 1'b0; /* TODO #4 */
                       AddrSel = 1'b1; /* TODO #4 */
                   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;
                       pc_en = 1'b0;
                       ILin = 1'b0;
                       rf_sel = 1'b0; /* TODO #4 */
                       sw_sel = 1'b0; /* TODO #4 */
                       MemWr = 1'b0; /* TODO #4 */
                       AddrSel = 1'b0; /* TODO #4 */
                   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;
                       pc_en = 1'b0;
                       ILin = 1'b0;
                       rf_sel = 1'b0; /* TODO #4 */
                       sw_sel = 1'b0; /* TODO #4 */
                       MemWr = 1'b0; /* TODO #4 */
                       AddrSel = 1'b0; /* TODO #4 */
                   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;
                       pc_en = 1'b0;
                       ILin = 1'b0;
                       rf_sel = 1'b1; /* TODO #4 */
                       sw_sel = 1'b0; /* TODO #4 */
                       MemWr = 1'b0; /* TODO #4 */
                       AddrSel = 1'b0; /* TODO #4 */
                   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;
        pc_en = 1'b0;
        ILin = 1'b0;
        rf_sel = 1'b1; /* TODO #4 */
        sw_sel = 1'b0; /* TODO #4 */
        MemWr = 1'b0; /* TODO #4 */
        AddrSel = 1'b0; /* TODO #4 */
    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;
        pc_en = 1'b0;
        ILin = 1'b0;
        rf_sel = 1'b1; /* TODO #4 */
        sw_sel = 1'b0; /* TODO #4 */
        MemWr = 1'b0; /* TODO #4 */
        AddrSel = 1'b0; /* TODO #4 */
    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;
        pc_en = 1'b0;
        ILin = 1'b0;
        rf_sel = 1'b0; /* TODO #4 */
        sw_sel = 1'b0; /* TODO #4 */
        MemWr = 1'b0; /* TODO #4 */
        AddrSel = 1'b0; /* TODO #4 */
    end
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;
        rf_sel = 1'b0; /* TODO #4 */
        sw_sel = 1'b0; /* TODO #4 */
        MemWr = 1'b0; /* TODO #4 */
        AddrSel = 1'b0; /* TODO #4 */
    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;
                        pc_en = 1'b0;
                        ILin = 1'b0;
                        rf_sel = 1'b0; /* TODO #4 */
                       sw_sel = 1'b0; /* TODO #4 */
                       MemWr = 1'b0; /* TODO #4 */
                       AddrSel = 1'b0; /* TODO #4 */
                    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;
                        pc_en = 1'b0;
                        ILin = 1'b0;
                        rf_sel = 1'b0; /* TODO #4 */
                       sw_sel = 1'b1; /* TODO #4 */
                       MemWr = 1'b0; /* TODO #4 */
                       AddrSel = 1'b0; /* TODO #4 */
                    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;
                        pc_en = 1'b0;
                        ILin = 1'b0;
                        rf_sel = 1'b0; /* TODO #4 */
                       sw_sel = 1'b1; /* TODO #4 */
                       MemWr = 1'b0; /* TODO #4 */
                       AddrSel = 1'b0; /* TODO #4 */
                    end
                STORE:
                    begin
                        _Extern = 1'b0;
                        Gout = 1'b0;
                        //Iout = 1'b1;
                        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;
                        rf_sel = 1'b0; /* TODO #4 */
                       sw_sel = 1'b0; /* TODO #4 */
                       MemWr = 1'b1; /* TODO #4 */
                       AddrSel = 1'b1; /* TODO #4 */
                    end
            endcase
        end



/*
opcode encodings
0000 - load
0001 - move
0010 - subtract
0011 - add
0100 - disp
0101 - HALT
0110 - subi
```

```
0111 - addi
1000 - store
*/

        always@(posedge clk or posedge reset)
        begin

                if (reset==1) state <= FETCH;
                else
                case(state)
                        FETCH:
                                    state <= DECODE;
                        DECODE:
                                        if(operation == 4'b0000) state <= LOAD;
                                        else if(operation == 4'b0001) state <= READ_Y;
                                        else if(operation == 4'b0010) state <= READ_X;
                                        else if(operation == 4'b0011) state <= READ_Y;
                                        else if(operation == 4'b0100) state <= DISP;
                                        else if(operation == 4'b0101) state <= HALT;
                                        else if(operation == 4'b0110) state <= READ_X;
                                        else if(operation == 4'b0111) state <= READ_X;
                                        else if(operation == 4'b1000) state <= STORE;
                                        else state <= FETCH;
                        LOAD:
                                        state <= FETCH;
                        READ_Y:
                                                if(operation == 4'b0001) state <= MV;
                                                else if(operation == 4'b0011) state <= ADD;
                                                else state <= READ_Y;
                        READ_X:
                                                if(operation == 4'b0010) state <= SUB;
                                                else if(operation == 4'b0110) state <= SUBI;
                                                else if(operation == 4'b0111) state <= ADDI;
                                                else state <= READ_X;
                        ADD:
                                        state <= WRITE_X;
                        SUB:
                                        state <= WRITE_X;
                        MV:
                                        state <= WRITE_X;
                        WRITE_X:
                                        state <= FETCH;
                        DISP:
                                        state <= FETCH;
                        ADDI:
                                        state <= WRITE_X;
                        SUBI:
                                state <= WRITE_X;
                        STORE:
                                state <= FETCH;
                        HALT:
                                state <= HALT;

                        default: state <= FETCH;
```

mem.txt Code:

```
        2000 // sub r0, r0, r0
        2111 // sub r1, r1, r1
        2222 // sub r2, r2, r2
        2333 // sub r3, r3, r3
        // Do this to make sure r0-r3 all have nothing in them
        7006 // addi r0, r0, 6
        3000 // add r0, r0, r0 (12)
        3000 // add r0, r0, r0 (24)
        3000 // add r0, r0, r0 (48)
        0100 // load r1, r0 (48)
        7001 // addi r0, r0, 1
        0200 // load r2, r0 (49)
        7001 // addi r0, r0, 1
        0300 // load r3, r0 (50)
```

```
3221 // add val1 and val2
8032 // store r2 at r3 value
0030 // load r3 address value in r0
4000 // display r0 to prove the store works
3323 // add r3 and r2
4030 // display r3 (now the sum of r3,r2,r1)
5000 // HALT!
```

TB Code (modified only):

```
/* TODO #2-2: instantiate the data MUX (data_mux.v) */
        data_mux d_mux(
                .switch_data(mem_out),
                                  .G_data(g_out_data),
                                  .Gout(gout),
                                  ._Extern(extern),
                                  .mux_output(mout)
       );
    /* end of TODO #2-2 */

    /* TODO #3-2: instantiate the imm mux (imm_mux.v) */
        imm_mux imm_mux(.sw_data(se_imm),
                    .rf_data(rf_2),
                .sw_select(sw_sel),
                .rf_select(rf_sel),
                .adder_b(IM_out)
    );
    /* end of TODO #3-2 */

/* TODO #1: Instatiate Memory Mux */
    assign rfaddr = rf_1[7:0];
    mux_2_to_1 #(.bit_width(8))
                mm(
                 .in0(pc_out),
                 .in1(rfaddr),
                 .sel(addr_sel),
                 .mux_output(mm_out));
    /* end of TODO #1 */
```

16-bit Sign Extension Code:

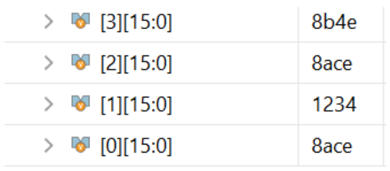
```
module SE_16(
        input [3:0] imm,
        output [15:0] extended
);

    /* TODO #3-1: prefix the MSB 12 times */
    assign extended = {{12{imm[3]}}, imm};


    /*{imm[3],imm[3],imm[3],imm[3],
                   imm[3],imm[3],imm[3],imm[3],
                   imm[3],imm[3],imm[3],imm[3],imm}; */
    /* end of TODO #3-1*/

endmodule
```

Data_Mux Instantiation:

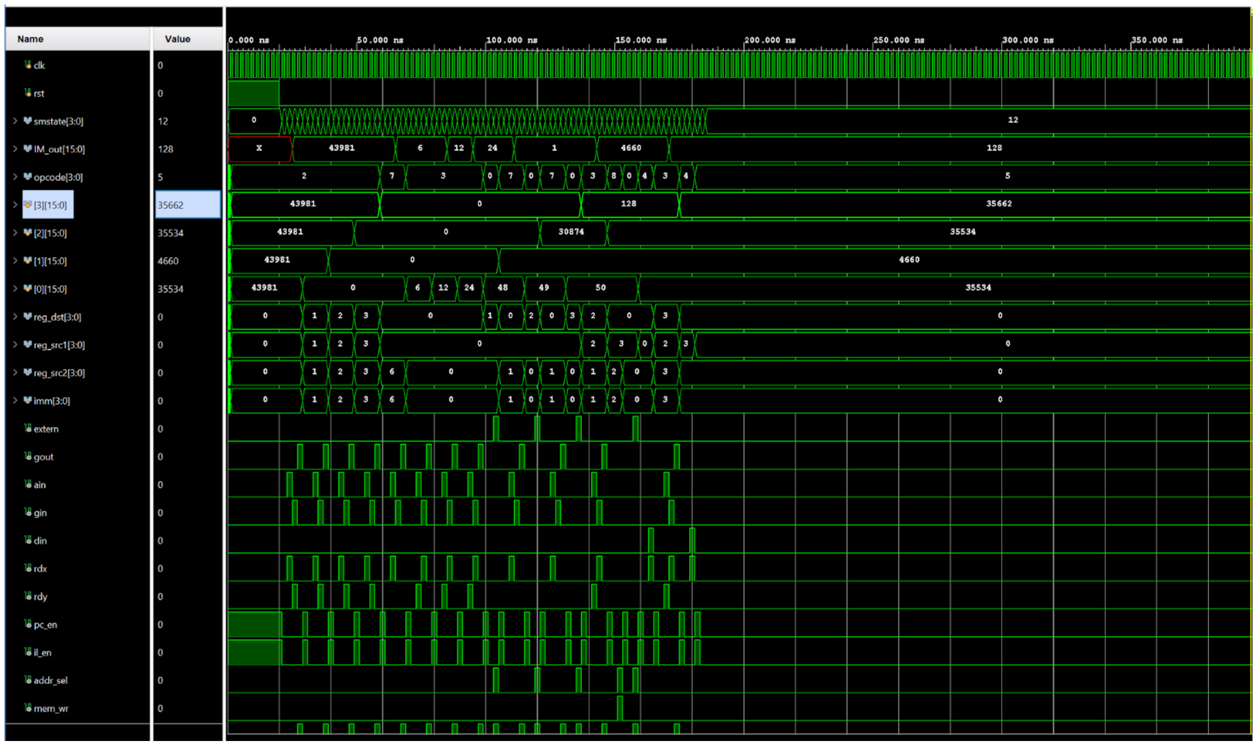
```
module data_mux(input [15:0] switch_data, // data from mem
                input [15:0] G_data,        // data from latch G (ALU)
                input Gout,
                input _Extern,
                output reg [15:0] mux_output);

    always@(*)
    begin
        /* TODO #2-1: complete the design of Data MUX */
        /* if Gout == 1 and switch_data == 0 G_data is chosen for output */
        if(Gout) mux_output <= G_data;
        if(_Extern) mux_output <= switch_data;

        /* if Gout == 0 and switch_data == 1 switch_data is chosen for output */
        /* if both are 0, output is just deasserted */
        /* end of TODO #2-1 */
    end
```

```
endmodule
```

Waveform Results:



[3][15:0]	8b4e
[2][15:0]	8ace
[1][15:0]	1234
[0][15:0]	8ace



Name	Value
clk	0
rst	0
smstate[3:0]	12
IM_out[15:0]	128
opcode[3:0]	5
[3][15:0]	35662
[2][15:0]	35534
[1][15:0]	4660
[0][15:0]	35534
reg_dst[3:0]	0
reg_src1[3:0]	0
reg_src2[3:0]	0
imm[3:0]	0
extern	0
gout	0
ain	0
gin	0
din	0
rdx	0
rdy	0
pc_en	0
il_en	0
addr_sel	0
mem_wr	0