Dylan Thornburg and Ayden Dauenhauer

SANTA CLARA UNIVERSITY	ECEN 122 Winter 2024	Hoeseok Yang

Laboratory #7: Pipelining

For lab section on Tuesday/Friday March 5/8, 2024

I. OBJECTIVES

Convert our CPU to a pipelined implementation.

PROBLEM STATEMENT

Up to now we have been controlling our CPU with a state machine, and executing one instruction at a time. Now it is time to throw our state machine out and move to a pipelined implementation.

You will be given a datapath that is a slight permutation of what we have been using already. It will be structured for a 4-stage pipeline: 1) instruction fetch, 2) instruction decode and register read, 3) execute (which will encompass both ALU computation and data memory access), and 4) register file update.

We have learned that pipelining introduces hazards. The datapath you will be given will solve the structural hazard of supporting both instruction fetch and data memory accesses by splitting the memory module into two: one for instruction fetch and the other for loads/stores. But the support for data and control hazards will not be present in this datapath. Nor will we be adding it in this lab.

For this lab, we are going to avoid control hazards by just executing the program from lab 5, which had not control transfers in it. And to avoid data hazards, you will need to add a NOP instruction and figure if and where to use them in your lab 5 program.

In summary, you will need to do the following:

- Create a decode block to generate the proper control signals based on the opcode of the instruction in the IR.
- Apply the correct staging of these control signals.
- Add a no-op instruction, to be used in your program as a means for dealing with the lack of data hazard support in the hardware.

Staging registers

We need to make sure that we have a means for isolating each of the pipeline stages. The basic parameterizable register implementation is given in A.v. In the given top-level Verilog file (l7_TB.v), staging registers are properly instantiated between pipeline stages.

PC support

In the previous lab, we added some muxes in front of the PC, which allowed us to implement

transfers of control (i.e., branches and jumps). In this lab, we are going to avoid the complexities associated with transferring control in a pipeline, and just run a "straight line" program, where the PC just always get incremented. So, the PC module you will be given will just be a counter.

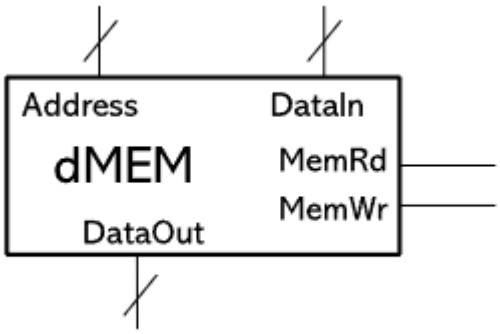
The pipeline

Our four pipeline stages are bracketed by five sequential circuit elements:

PC - <instr. fetch> - IR - <decode/op read> - Op A/B - <execute> - G/MDR - <writeback> - RF

[Note that the Op A/B and the G/MDR staging registers should also conceptually include the staging of control signals derived from the instruction decode as shown on the last page of the instructions (staging registers are shaded).]

The data memory module that you will receive from the TA will look like:



You will need to provide the control inputs for *MemRd* and *MemWr*.

Definition and use of a NOP instruction

Control and data hazards have the characteristic of incorrect instruction execution if the hazard is not handled properly. Hardware handling of hazards involves instruction flushing (in the case of control hazards) and data forwarding (in the case of data hazards). But for simplicity, we are not going to implement this support in this lab.

Control hazards will be avoided in this lab by essentially not supporting control transfer instructions.

The register file you will be using will not have support for bypass. This means that a value cannot be read out of the register file until the cycle after it is written to the register file. Mapping to our 4-stage pipeline, that means a register can't be used until two instructions after an instruction that writes to that register. Any use sooner than that presents a data hazard; allowing the instruction to read the register file will yield the wrong result.

Dealing with data hazards, without any hardware support, requires ensuring that no data dependent instructions are too close together in the instruction stream. If it is not possible to move instructions

around to keep data dependent instructions separated, the program may need to include instructions that do nothing. This type of instruction is called a no-op (NOP). The decode of a NOP needs to ensure that none of the control signals that would affect the state of the system are asserted, i.e., it needs to make sure that no value is written to the RF and that no value is stored to memory.

Controlling the IR via the HALT instruction

One additional behavior we need to account for is how to implement the HALT instruction. Ultimately, what we need to happen when a HALT instruction is decoded is prevent any further instructions from being loaded into the IR, which means affecting how the load enable for the IR is implemented. In this lab, you can think of the load enable for the IR as being an indication that any instruction other than a HALT instruction is being executed.

Additionally, since the decode of the current contents of the IR will affect the load enable of the IR, we need to make sure the IR is initialized (after reset is applied) to a value that ensures the load enable isn't adversely affected. The approach we will take is to have the IR initialized to the opcode for NOP. Since you have been free to define a different opcode for NOP than other teams, you will need to modify the IR module provided to you to ensure that it is initialized with your version of NOP.

PRE-LAB

(i) Review the pipelined datapath shown on the last page and briefly explain the role of each pipeline stage.

(ii) The following is a list all of the control signals that you need to derive during the decode stage of the pipeline. Based on your understanding of the pipelined datapath, specify which pipeline stage (i.e., DEC, EXE, or WB) each signal is used in. For example, the control signal used to indicate writing to the register file (*reg_rw*) needs to be used in the WB stage. (See the output ports of id.v)

Control signals	Pipeline Stage	Control signals	Pipeline Stage
reg_src1	DEC	mem_wr	WB
reg_src2	DEC	reg_wr	WB
reg_dst	DEC	imm_sel	DEC
imm	DEC	data_sel	EX
add_sub	EX	dp_en	EX
mem_rd	EX	halt	WB

(iii)Specify an opcode that you will use for a NOP instruction.

1001

(iv)For each instruction type, the ID block (id.v) should generate proper control signals. Fill out the following table.

	load	move	sub	add	disp	halt	subi	addi	store	nop
add_sub	0	0	1	0	0	0	1	0	0	0
mem_rd	1	0	0	0	0	0	0	0	0	0
mem_wr	0	0	0	0	0	0	0	0	1	0
reg_wr	1	1	1	1	0	0	1	1	0	0
imm_sel	0	0	0	0	0	0	1	1	0	0
data_sel	1	0	0	0	0	0	0	0	1	0
dp_en	0	0	0	0	1	0	0	0	0	0
halt	0	0	0	0	0	1	0	0	0	0

(v) Review the program you used for lab 5 and determine if it would present any data hazards. If it does, then re-write your program so that it no longer has any data hazards. You need to include at least one NOP instruction, but feel free to use more if you need to.

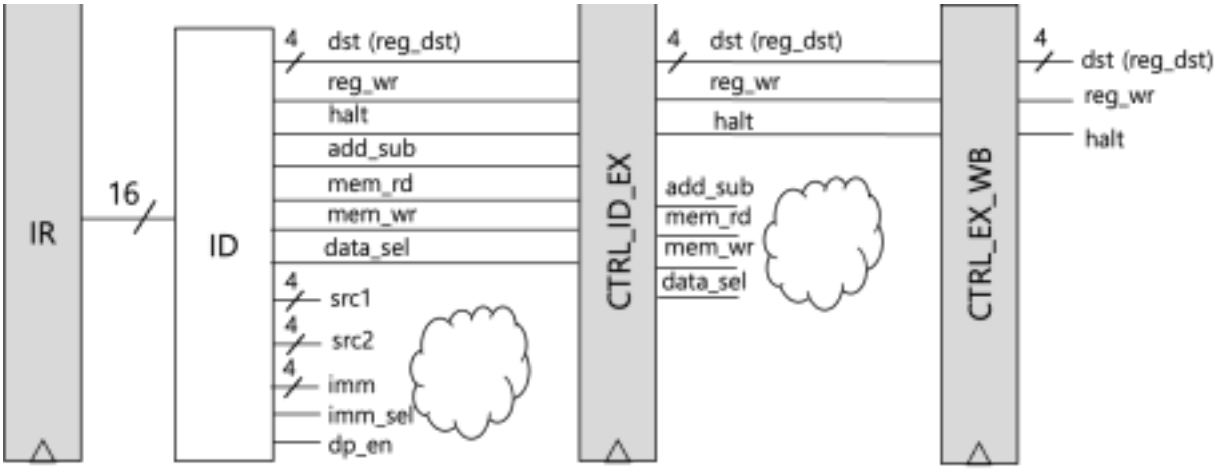
Note that, in previous labs, we put the data at address location 48. This was done to separate the data from the instructions of your program, given that they were stored in the same memory. For this lab, since we're actually instantiating a separate memory for data, we could avoid this step and have our data start from data address 0. But to keep things consistent and minimize change from the previous labs, we'll stick with the data starting at address 48.

II. LAB PROCEDURE

(i) Load the datapath provided by the TA into Vivado.

(ii) After reconciling your pre-labs, split the work between you and partner to get all the necessary changes implemented. Note that you will need

a. A decode module that decodes all of your instructions (TODO #1 – in id.v). b. Proper connection of control signals to all the datapath elements, including capturing some of them in staging registers so that they get delayed appropriately. Remember that the destination register ID will need to be staged so that it is correctly applied in the WB stage; don't use reg_dst from the ID block directly. Refer to the following diagram for the signal connection (TODO #2 – in l7_TB.v).



(iii)Make sure to insert NOPs properly to the given program to avoid data hazards (TODO #3 – in imem.txt). When you believe you have everything ready to run, simulate your design, and start working to demonstrate that your program can be correctly executed.

(iv)Try the following program (TODO #4). You may rearrange the instructions or add NOPs (as long as it doesn't change the result) to avoid data hazards

```
sub r0, r1, r1 // r0 = 0
addi r0, r0, 1 // r0 = 1
addi r1, r0, 1 // r1 = 2
addi r2, r0, 2 // r2 = 3
disp. r2 // display 3
store. r1, r2 // Mem[2] = 3
```

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

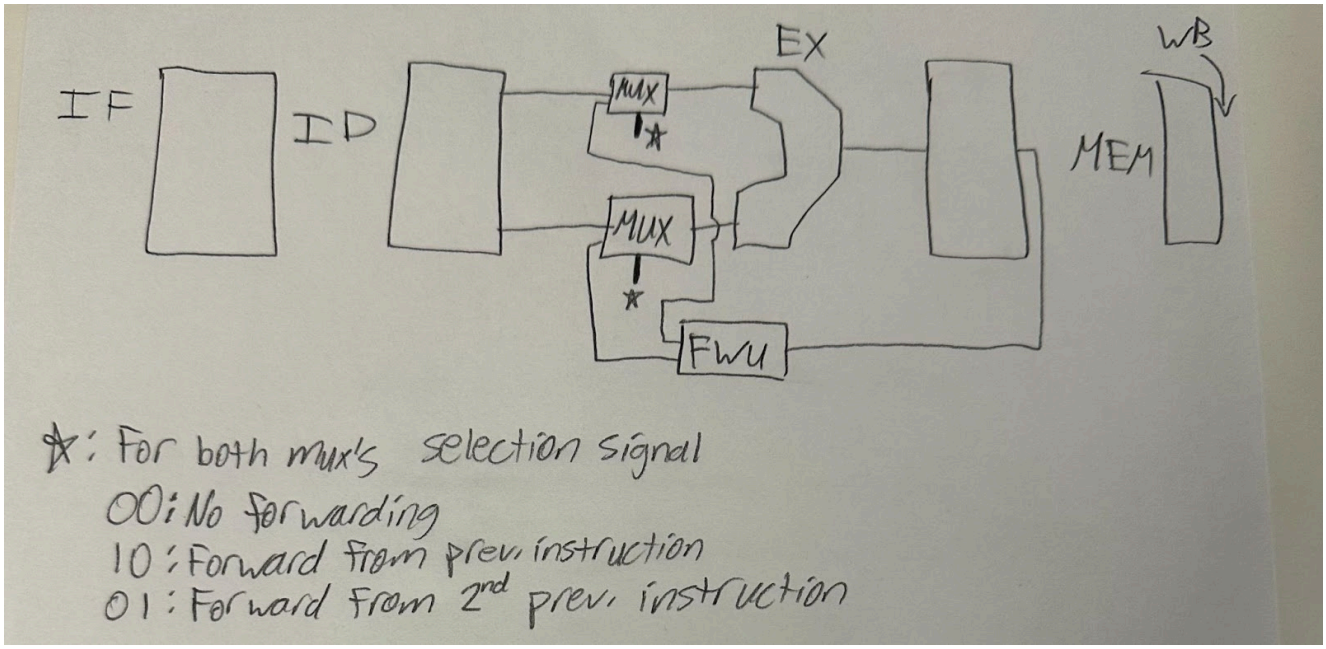
III. REPORT

For your lab report, include the source code for your decode logic, and the final version of your program that you demonstrated working. 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.**

  We had an error in our testbench because we flipped the 'out' and 'in' of one of the connections which caused an error with our load function. As for the demo, we were able to explain what was displayed and why. We also explained why state machines are preferable in some cases compared to pipelining – if we wanted to create something which has a clear input and output rather than making it more complex through pipelining. eg: fighter planes

- **Draw enough of a block diagram to demonstrate how you would implement data forwarding if needed, and write the Verilog that would be needed to generate the proper control signals for whatever datapath elements you would need to add to the base datapath.**



You would instantiate the muxes as needed (mux code is given we'd just need to declare them)

```
module forwarding_unit(
input [4:0] EX_MEM_Rd, /* Rd of the previous instr */
input [4:0] MEM_WB_Rd,/* Rd of the 2nd previous instr */
```

```
input EX_MEM_RegWrite,  /* RegWrite for the previous instr */
input MEM_B_RegWrite, /* RegWrite for the 2nd previous instr */
input [4:0] ID_EX_Rs1, /* Rs1 of the current instr */
input [4:0 ID_EX_Rs2, /* Rs2 of the current instr */
output [1:0] ForwardA /* MUX sel. for the Ist ALU operand */
output [1:0] ForwardB /* MUX sel. for the 2nd ALU operand */
);
always@(*)
begin
if (EX_MEM_RegWrite == 1'b1) /* data hazard from the previous instr. */
        if (EX_MEM_Rd != 5'b00000)
                if (EX_MEM_Rd == ID_EX_Rs1)
                        ForwardA <= 2b'10;
                else ForwardA <= 2'b00;
        else ForwardA <= 2'b00;
else ForwardA <= 2'b00;

if (MEM_W_RegWrite == 1'b1) /* data hazard from the 2nd previous instr. */
        if (MEM_WB_Rd != 5'b00000)
                if (MEM_WB_Rd == ID_EX_Rs2)
                        ForwardB <= 2b'01;
                else ForwardB <= 2'b00;
        else ForwardB <= 2'b00;
else ForwardB <= 2'b00;
end
endmodule
```

- **If you were to implement the data forwarding changes you identified, your program wouldn't need all the NOPs you inevitably had to put in. In fact, you may not need any of them anymore. Explain which NOPs from your working program could be removed, and why.**

  All the NOPs between the adds and subs would be removed. Just the NOP for load-use hazards would remain.

CODE

id.v

```
module id( /* instruction decode */
   input [15:0] instr,
   output [3:0] reg_src1,
   output [3:0] reg_src2,
   output [3:0] reg_dst,
   output [3:0] imm,
   output reg add_sub,
   output reg mem_rd,
   output reg mem_wr,
   output reg reg_wr,
   output reg imm_sel,
   output reg data_sel,
   output reg dp_en,
   output reg halt
   );

   wire [3:0] opcode;

   assign opcode = instr[15:12];

   assign reg_dst = instr[11:8];
   assign reg_src1 = instr[7:4];
   assign reg_src2 = instr[3:0];
   assign imm = instr[3:0];

   /* for other control signals */
   always@(*)
   begin
      if(opcode==4'b0000) /* load */
        begin
          add_sub = 1'b0;
          mem_rd = 1'b1;
          mem_wr = 1'b0;
          reg_wr = 1'b1;
          imm_sel = 1'b0;
          data_sel = 1'b1;
          dp_en = 1'b0;
          halt = 1'b0;
        end

      /* TODO #1: Complete the implmentation of the ID block */
       else if (opcode==4'b0001) /* move */
        begin
          add_sub = 1'b0;
          mem_rd = 1'b0;
          mem_wr = 1'b0;
          reg_wr = 1'b1;
          imm_sel = 1'b0;
          data_sel = 1'b0;
```

```
    dp_en = 1'b0;
    halt = 1'b0;
  end

else if (opcode==4'b0010) /* sub */
  begin
    add_sub = 1'b1;
    mem_rd = 1'b0;
    mem_wr = 1'b0;
    reg_wr = 1'b1;
    imm_sel = 1'b0;
    data_sel = 1'b0;
    dp_en = 1'b0;
    halt = 1'b0;
  end

else if (opcode==4'b0011) /* add */
  begin
    add_sub = 1'b0;
    mem_rd = 1'b0;
    mem_wr = 1'b0;
    reg_wr = 1'b1;
    imm_sel = 1'b0;
    data_sel = 1'b0;
    dp_en = 1'b0;
    halt = 1'b0;
  end

else if (opcode==4'b0100) /* disp */
  begin
    add_sub = 1'b0;
    mem_rd = 1'b0;
    mem_wr = 1'b0;
    reg_wr = 1'b0;
    imm_sel = 1'b0;
    data_sel = 1'b0;
    dp_en = 1'b1;
    halt = 1'b0;
  end

else if (opcode==4'b0101) /* halt */
  begin
    add_sub = 1'b0;
    mem_rd = 1'b0;
    mem_wr = 1'b0;
    reg_wr = 1'b0;
    imm_sel = 1'b0;
    data_sel = 1'b0;
    dp_en = 1'b0;
    halt = 1'b1;
```

```
  end

else if (opcode==4'b0110) /* subi */
  begin
    add_sub = 1'b1;
    mem_rd = 1'b0;
    mem_wr = 1'b0;
    reg_wr = 1'b1;
    imm_sel = 1'b1;
    data_sel = 1'b0;
    dp_en = 1'b0;
    halt = 1'b0;
  end

else if (opcode==4'b0111) /* addi */
  begin
    add_sub = 1'b0;
    mem_rd = 1'b0;
    mem_wr = 1'b0;
    reg_wr = 1'b1;
    imm_sel = 1'b1;
    data_sel = 1'b0;
    dp_en = 1'b0;
    halt = 1'b0;
  end

else if (opcode==4'b1000) /* store */
  begin
    add_sub = 1'b0;
    mem_rd = 1'b0;
    mem_wr = 1'b1;
    reg_wr = 1'b0;
    imm_sel = 1'b0;
    data_sel = 1'b1;
    dp_en = 1'b0;
    halt = 1'b0;
  end

else if (opcode==4'b1001) /* nop */
  begin
    add_sub = 1'b0;
    mem_rd = 1'b0;
    mem_wr = 1'b0;
    reg_wr = 1'b0;
    imm_sel = 1'b0;
    data_sel = 1'b0;
    dp_en = 1'b0;
    halt = 1'b0;
  end
```

```
        else
          begin
            add_sub = 1'b0;
            mem_rd = 1'b0;
            mem_wr = 1'b0;
            reg_wr = 1'b0;
            imm_sel = 1'b0;
            data_sel = 1'b0;
            dp_en = 1'b0;
            halt = 1'b0;
          end

    end

endmodule
```

TB.v:

```
module l7_TB();
    /* top-level design file for lab 7 */

    reg clk, rst;   // clock and reset

    reg reg_en;     /* register enable signals */

      wire [3:0] reg_src1;
      wire [3:0] reg_src2;
      wire [3:0] imm;

      /* control signals */

      /* control signals used in the 2nd stage (DEC) */
      wire imm_sel;      /* imm mux selection signal */
      wire dp_en;        /* disp register enable */

      /* control signals used in the 3rd stage (EXE) */
      wire add_sub;        /* alu control */
      wire mem_rd;
      wire mem_wr;
      wire data_sel;     /* data mux selection signal */

      /* control signals used in the 4th stage (WB) */
      wire [3:0] reg_dst;    /* destination register index */
      wire wrx;           /* RF write enable */
      wire halt;

      /* staging register in/out */
      wire [9:0] ctrl_id_ex_in;
      wire [9:0] ctrl_id_ex_out;
```

```
  wire [5:0] ctrl_ex_wb_in;
  wire [5:0] ctrl_ex_wb_out;


  /* datapath signals */
  wire [7:0] pc_out;           /* PC output: PC -> iMEM */
  wire [15:0] instr;           /* instruction: IR output */
  wire [15:0] imem_out;        /* iMEM output: iMEM -> IR */
wire [15:0] rf_2, rf_1;       /* two rf output */
wire [15:0] se_imm;           /* sign-extended imm signal: SE -> imm_mux */
wire [15:0] imm_mux_out;      /* imm mux out */
wire [15:0] alu_out;          /* ALU out */
wire [15:0] dmem_out;         /* dMEM out */
wire [15:0] data_mux_out;     /* data mux out */


wire [15:0] a_out_data;       /* reg a output */
wire [15:0] b_out_data;       /* reg b output */
wire [15:0] g_out_data;       /* reg g output */
wire [15:0] dp_out_data;      /* reg dp output */


/* Staging Register 0: Program Counter */
  PC pc(.clk(clk),
        .countEn(reg_en),
        .reset(rst),
        .Address(pc_out));


/* Instruction Memory (iMEM), Read-Only */
  l7_iMEM mem(.clk(clk),
        .address(pc_out),
        .DataOut(imem_out));


  /* Staging Register 1: Instruction Register (IR) - 16-bits */
A #(.bit_width(16)) IR(.Ain(imem_out), .load_en(reg_en), .clk(clk), .Aout(instr));


/* instruction decode */
id id(.instr(instr),
   .reg_src1(reg_src1),
   .reg_src2(reg_src2),
   .reg_dst(reg_dst),
   .imm(imm),
   .add_sub(add_sub),
   .mem_rd(mem_rd),
   .mem_wr(mem_wr),
   .reg_wr(wrx),
   .imm_sel(imm_sel),
```

```
    .data_sel(data_sel),
    .dp_en(dp_en),
    .halt(halt)
);

/* 16-bit sign-extension */
SE_16 se(.imm(imm), .extended(se_imm));


/* Register File */
l7_RF RF(.clk(clk),
      .rst(rst),
      .DataIn(g_out_data),
      .raddr_2(reg_src2),
      .raddr_1(reg_src1),
      .waddr(ctrl_ex_wb_out[5:2]), /* TODO #2-1 reg_dst*/
      .WrX(ctrl_ex_wb_out[1]), /* TODO #2-2 wrx*/
      .out_data_2(rf_2),
      .out_data_1(rf_1)
              );

   /* imm_mux */
   mux_2_to_1 #(.bit_width(16)) imm_mux(
             .in0(rf_2),
          .in1(se_imm),
          .sel(imm_sel),
          .mux_output(imm_mux_out));


/* diplay register */
 A #(.bit_width(16)) DP(
    .clk(clk), .rst(rst),
    .Ain(rf_1),
    .load_en(dp_en),
    .Aout(dp_out_data));

   /* Staging Register 2: OP_A - 16-bits, OP_B - 16-bits, CTRL - n bits? */
 A #(.bit_width(16)) OP_A(
    .clk(clk), .rst(rst),
    .Ain(rf_1),
    .load_en(reg_en),
    .Aout(a_out_data));

 A #(.bit_width(16)) OP_B(
    .clk(clk), .rst(rst),
    .Ain(imm_mux_out),
    .load_en(reg_en),
    .Aout(b_out_data));
```

```
/* TODO #2-3: ID -> CTRL_ID_EX */
assign ctrl_id_ex_in[9:6] = reg_dst;
assign ctrl_id_ex_in[5] = wrx;
assign ctrl_id_ex_in[4] = halt;
assign ctrl_id_ex_in[3] = add_sub;
assign ctrl_id_ex_in[2] = data_sel;
assign ctrl_id_ex_in[1] = mem_rd;
assign ctrl_id_ex_in[0] = mem_wr;

 A #(.bit_width(10)) CTRL_ID_EX(
    .clk(clk), .rst(rst),
    .Ain(ctrl_id_ex_in), /* TODO #2-4 */
    .load_en(reg_en),
    .Aout(ctrl_id_ex_out)); /* TODO #2-5 */

   /* Execution (ALU) and Memory Accesses */

ALU alu (.in_A(a_out_data),
           .in_B(b_out_data),
           .add_sub(ctrl_id_ex_out[3]),
           .adder_out(alu_out));

l7_dMEM dmem (
  .clk(clk),
  .address(a_out_data[7:0]),
  .DataIn(b_out_data),
  .MemRd(ctrl_id_ex_out[1]), /* TODO #2-6 mem_rd*/
  .MemWr(ctrl_id_ex_out[0]),/* TODO #2-7 mem_wr*/
  .DataOut(dmem_out)
);

/* data mux */
mux_2_to_1 #(.bit_width(16)) data_mux(
          .in0(alu_out),
          .in1(dmem_out),
          .sel(ctrl_id_ex_out[2]),
          .mux_output(data_mux_out));


   /* Staging Register 3: G/MDR*/
 A #(.bit_width(16)) g(
    .clk(clk), .rst(rst),
    .Ain(data_mux_out),
    .load_en(reg_en),
    .Aout(g_out_data));

 /* TODO #2-8 */
 assign ctrl_ex_wb_in[5:2] = ctrl_id_ex_out[9:6]; /* reg_dst */
 assign ctrl_ex_wb_in[1] = ctrl_id_ex_out[5];       /* wrx */
 assign ctrl_ex_wb_in[0] = ctrl_id_ex_out[4];       /* halt */
```

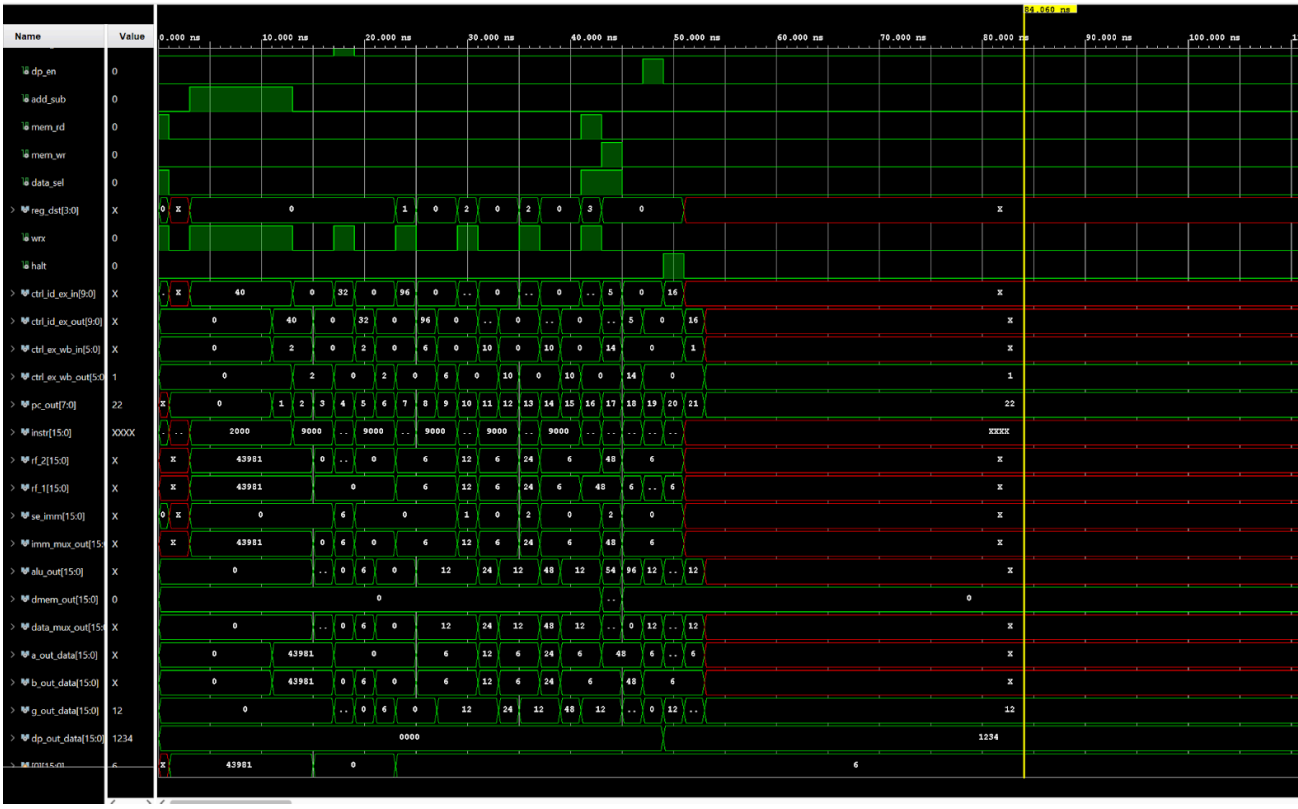
```
 A #(.bit_width(6)) CTRL_EX_WB(
    .clk(clk), .rst(rst),
    .Ain(ctrl_ex_wb_in), /* TODO #2-9 */
    .load_en(reg_en),
    .Aout(ctrl_ex_wb_out)); /* TODO #2-10 */


  /* halt signal generator */
  always@(*)
  begin
   if (rst) reg_en <= 1'b1;
   else
    if (ctrl_ex_wb_out[0]) /* halt signal */
     reg_en <= 1'b0;

  end


  initial begin
     clk = 0;
            //this testbench is a bit different from the others
            //the instructinos come entirely from the program
            //to verify correctness, you will need to see that the
            //results match what you expect from the program
            rst = 1;
            #10;
            rst = 0;
            #1000;
            $finish;
   end

   always #1 clk = !clk;

endmodule
```

imem.txt:

```
2000 // sub r0, r0, r0
9000
9000
7006 // addi r0, r0, 6
9000
9000
3100 // add r1, r0, r0
9000
9000
3211 // add r2, r1, r1
9000
9000
```

3222 // add r2, r2, r2
9000
9000
0320 // load r3, r2(=48)
8022 // store r2, r2
9000 // needed for proper display
4030 // disp r3
5000 // halt

Screenshots:




Name
Value
0.000 ns
10.000 ns
20.000 ns
30.000 ns
40.000 ns
50.000 ns
60.000 ns
70.000 ns
80.000 ns
90.000 ns
100.000 ns
84.960 ns
wrx
halt
ctrl_id_ex_in[9:0]
ctrl_id_ex_out[9:0]
ctrl_ex_wb_in[5:0]
ctrl_ex_wb_out[5:0]
pc_out[7:0]
instr[15:0]
rf_2[15:0]
rf_1[15:0]
se_imm[15:0]
imm_mux_out[15:0]
alu_out[15:0]
dmem_out[15:0]
data_mux_out[15:0]
a_out_data[15:0]
b_out_data[15:0]
g_out_data[15:0]
dp_out_data[15:0]
[0][15:0]
[1][15:0]
[2][15:0]
[3][15:0]
imem_out[15:0]
[4][15:0]
1234
abcd
43981
XXXX