Computer Architecture Project #1
Team 14

B03901089 Paul Wang, B03901116 Danny Wang

1 Team Work

- Paul Wang: Report.
- Danny Wang: Verilog implementation.

2 Pipeline Structure

We break each instruction into five pipeline stages: IF(Instruction fetch), ID(Intruction decode and register fetch), EX(Execution), MEM(Access data memory), and WB(Write back to register). Also, to avoid hazards, we have

- a data forwarding unit to avoid usual data hazards.
- a hazard detection unit to detect load-use hazard and stall the pipeline.
- simple branch prediction to reduce stalls on control hazards.

The implementation details are listed below.

3 Module Implementation

The top-level module is CPU, which links all the smaller modules. Its implementation is identical to the last image on the project slide. (See CPU.v for details)

As for the smaller modules, the implementations are listed below. (Some modules, including ALU, MUX, etc, are mostly the same as the last homework, so the implementation are not shown here.)

- Pipeline registers: use simple D flip-flops

```
always@(posedge clk_i or posedge rst_i) begin
  if(rst_i) begin
    read_o <= 0;
  end else begin
    read_o <= write_i;
  end
end
```

- Forwarding control unit: Detects which kind of data hazard is present and generates the corresponding forwarding control signals.

```
assign EX_Hazard  = EX_MEM_RegWrite_i & (EX_MEM_Rd_i != 0);
assign MEM_Hazard = MEM_WB_RegWrite_i & (MEM_WB_Rd_i != 0);

assign ForwardA_o =
    (EX_Hazard &
     (EX_MEM_Rd_i == ID_EX_Rs_i)) ? 2'b10 :
    (MEM_Hazard &
     (EX_MEM_Rd_i != ID_EX_Rs_i) &
     (MEM_WB_Rd_i == ID_EX_Rs_i)  ) ? 2'b01 : 2'b00;

assign ForwardB_o =
    (EX_Hazard &
     (EX_MEM_Rd_i == ID_EX_Rt_i)) ? 2'b10 :
    (MEM_Hazard &
     (EX_MEM_Rd_i != ID_EX_Rt_i) &
     (MEM_WB_Rd_i == ID_EX_Rt_i)  ) ? 2'b01 : 2'b00;
```

- Load-use hazard detection unit: Detects load-use hazards and generates signals to stall the pipeline.

```
assign stall = ID_EX_MemRead_i &
               ((ID_EX_Rt_i == IF_ID_Rs_i) |
                (ID_EX_Rt_i == IF_ID_Rt_i));
assign bubble_o = stall;
assign IF_ID_Write_o = stall;
assign PC_Write_o = stall;
```

- Main control unit: new signals to support new instructions(beq, jump, lw, sw, etc...)

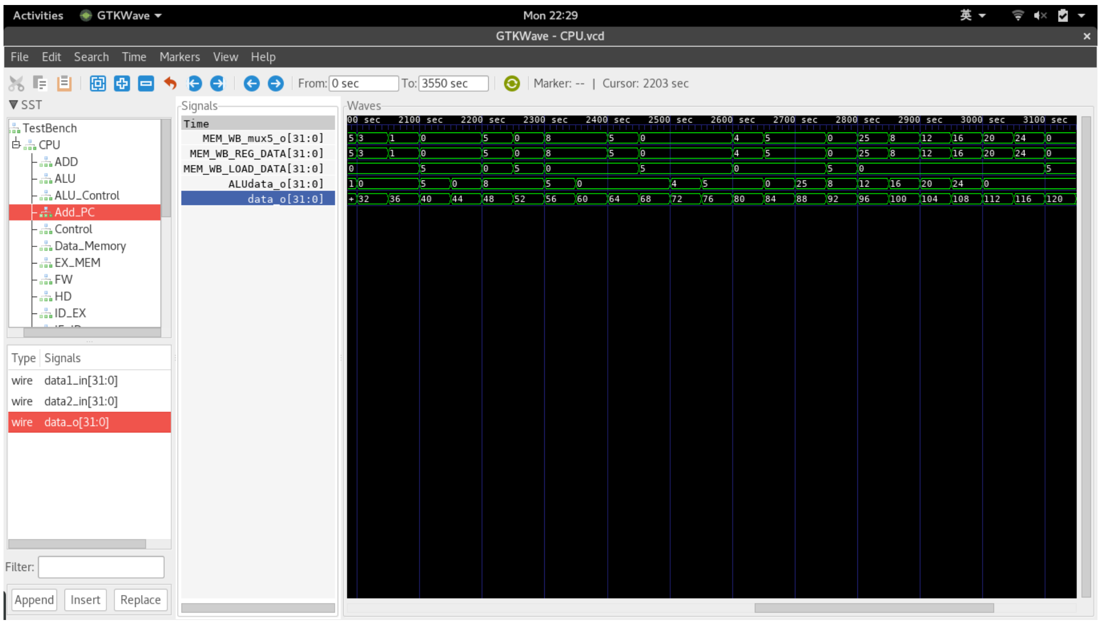
```
// According to pdf page 4 and hw4
// assign TYPE values for instructions
assign r_type = (Op_i == 6'b00_0000) ? 1 : 0;
assign addi   = (Op_i == 6'b00_1000) ? 1 : 0;
assign lw     = (Op_i == 6'b10_0011) ? 1 : 0;
assign sw     = (Op_i == 6'b10_1011) ? 1 : 0;
assign beq    = (Op_i == 6'b00_0100) ? 1 : 0;
assign jump   = (Op_i == 6'b00_0010) ? 1 : 0;

// assign SIGNAL values for use
assign RegDst_o   = r_type;
assign RegWrite_o = r_type | addi | lw;
assign MemWrite_o = sw;
assign MemtoReg_o = lw;
assign Branch_o   = beq;
assign Jump_o     = jump;
assign ExtOp_o    = lw | sw;
assign ALUSrc_o   = addi | lw | sw;
assign ALUOp_o    = r_type ? RTYPE :
                    addi | lw | sw ? ADD :
                    beq ? SUB : 2'b00;
```

- Program counter: Can now stall when receiving stall signal from the hazard detection unit.

```
always@(posedge clk_i or posedge rst_i) begin
    if(rst_i) begin
        pc_o <= 32'b0;
    end
    else begin
        if(start_i & ~pc_write_i)
            pc_o <= pc_i;
        else
            pc_o <= pc_o;
    end
end
```

4 Testing

By using *gtkwave* software on Linux, first dump waveform into a *.vcd* file.
It's easy to get any signal from any module at any time.(Figure 1.)



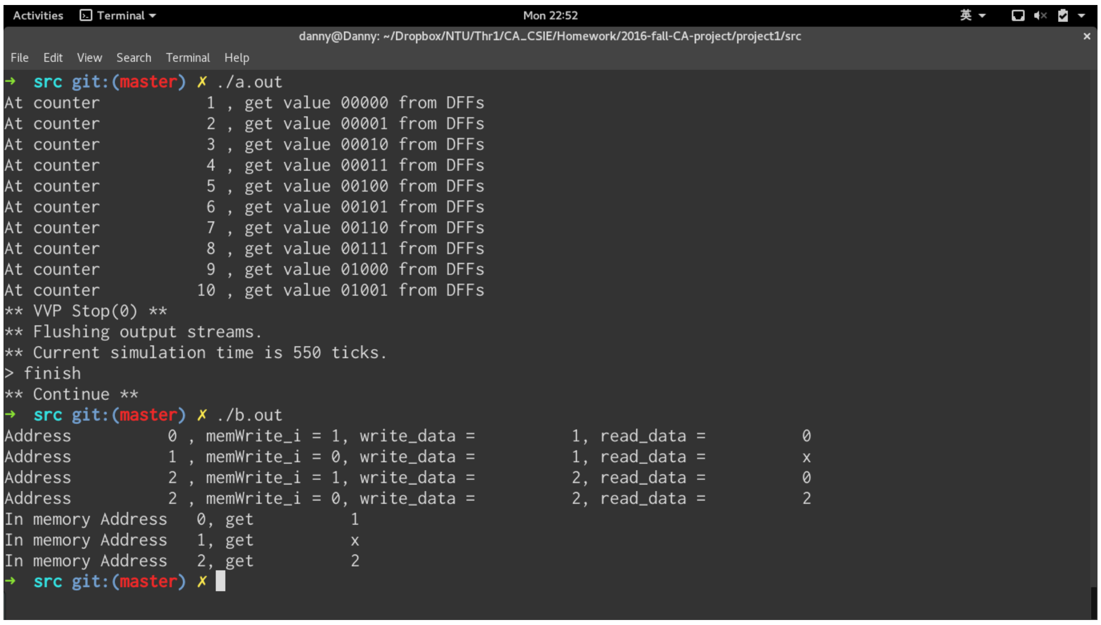
```
Time                 00 sec  2100 sec  2200 sec  2300 sec  2400 sec  2500 sec  2600 sec  2700 sec  2800 sec  2900 sec  3000 sec  3100 sec
MEM_WB_mux5_o[31:0]     5 3 1 0 5 0 8 5 0 4 5 0 25 8 12 16 20 24 0
MEM_WB_REG_DATA[31:0]   5 3 1 0 5 0 8 5 0 4 5 0 25 8 12 16 20 24 0
MEM_WB_LOAD_DATA[31:0]  0 5 0 5 0 5 0 5 0 5
ALUdata_o[31:0]         1 0 5 0 8 5 0 4 5 0 25 8 12 16 20 24 0
data_o[31:0]            + 32 36 40 44 48 52 56 60 64 68 72 76 80 84 88 92 96 100 104 108 112 116 120
```

Figure 1: Screenshot for debugging

We also wirte two testbenchs for DFF and Data_Memory to check the functionalities are right as we want.(Figure 2.)

5 Problems & Solutions

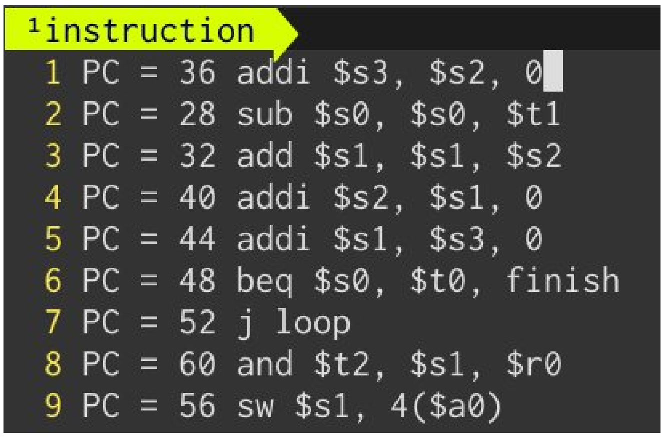
1. To avoid ID stage forwarding, our instruction order is modified as below.
   PC's are the original PC ordering. Move *addi* instruction to the beginning of the loop will avoid all ID stage forwarding inside the loop.
   For *sw* instruction at PC = 56, since *beq* at PC = 48 will stall for one cycle. It


Activities
Terminal
Mon 22:52
danny@Danny: ~/Dropbox/NTU/Thr1/CA_CSIE/Homework/2016-fall-CA-project/project1/src
File Edit View Search Terminal Help

```
➜  src git:(master) ✗ ./a.out
At counter           1 , get value 00000 from DFFs
At counter           2 , get value 00001 from DFFs
At counter           3 , get value 00010 from DFFs
At counter           4 , get value 00011 from DFFs
At counter           5 , get value 00100 from DFFs
At counter           6 , get value 00101 from DFFs
At counter           7 , get value 00110 from DFFs
At counter           8 , get value 00111 from DFFs
At counter           9 , get value 01000 from DFFs
At counter          10 , get value 01001 from DFFs
** VVP Stop(0) **
** Flushing output streams.
** Current simulation time is 550 ticks.
> finish
** Continue **
➜  src git:(master) ✗ ./b.out
Address          0 , memWrite_i = 1, write_data =           1, read_data =          0
Address          1 , memWrite_i = 0, write_data =           1, read_data =          x
Address          2 , memWrite_i = 1, write_data =           2, read_data =          0
Address          2 , memWrite_i = 0, write_data =           2, read_data =          2
In memory Address   0, get           1
In memory Address   1, get           x
In memory Address   2, get           2
➜  src git:(master) ✗
```

Figure 2: Screenshot for our testbenchs

causes ID stage forwarding. To avoid this, we need to add a dummy instruction or change the order of instructions after loop.(Figure 3.)



```
1instruction
PC = 36 addi $s3, $s2, 0
PC = 28 sub $s0, $s0, $t1
PC = 32 add $s1, $s1, $s2
PC = 40 addi $s2, $s1, 0
PC = 44 addi $s1, $s3, 0
PC = 48 beq $s0, $t0, finish
PC = 52 j loop
PC = 60 and $t2, $s1, $r0
PC = 56 sw $s1, 4($a0)
```

Figure 3: Modified instruction order

2. It's important to take care of clk's and rst's raising time, so adding #1 in testbench before *reset* signal to handle this problem well.

3. When using *$display* command in the testbench, we need to let signal to be in *reg* type. This problem let us need to write some dummy code in CPU.v.