Computer Architecture

Lab 4 Report

Name:	Asudy Wang 王浚哲	**Student ID:**	3180103011	**Major:**	Computer Science & Technology
Course:	Computer Architecture		**Place:**	Room 301, Cao Guangbiao Building West Wing, Yuquan Campus	
Due Date:	2020-12-28	**Groupmate:**	Flaze He	**Instructor:**	Kai Bu

Table of Contents

Lab 4. Pipelined CPU with Forwarding

§1 Purposes & Requirements

1.1 Experiment Purpose

- Understand the principles of **Pipelined CPU Bypass Unit**.
- Master the method of Pipelined Pipeline **Forwarding Detection** and Pipeline **Forwards**.
- Master the Condition In which Pipeline Forwards.
- Master the Condition In which Bypass Unit doesn't Work and the Pipeline **stalls**.
- Master methods of program verification of Pipelined CPU with forwarding.

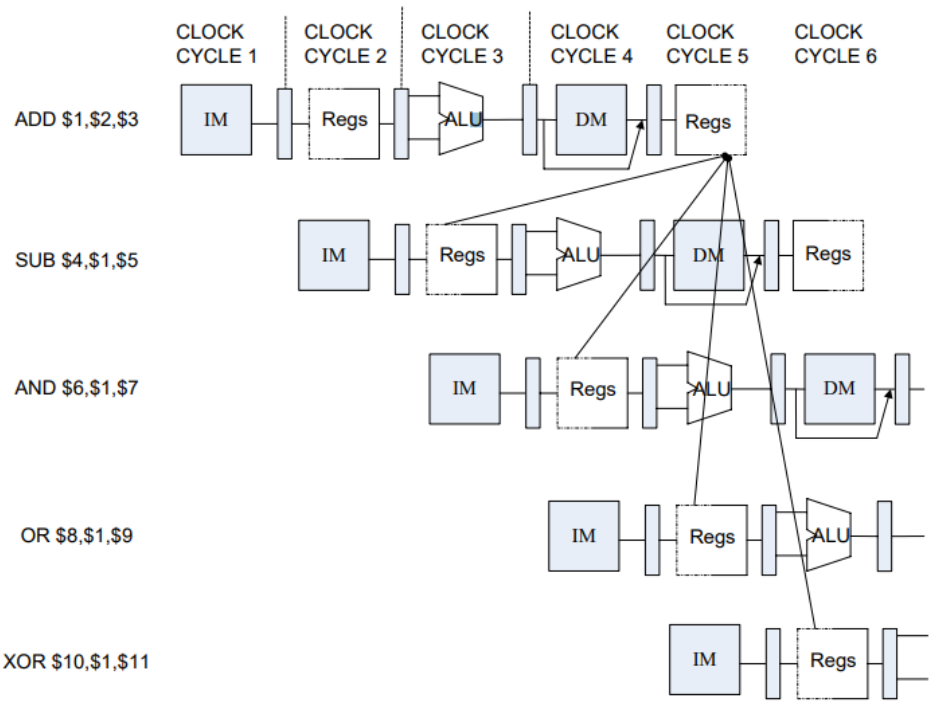
1.2 Experiment Tasks

- Design the **Bypass Unit** in datapath of the 5-staged Pipelined CPU.
- **Modify the CPU Controller**
  - Conditions in which the pipeline *forwards.*
  - Conditions in which the pipeline *Stalls.*
- **Verify** the Pipelined CPU with program and observe the execution of the program.

§2 Contents & Principles

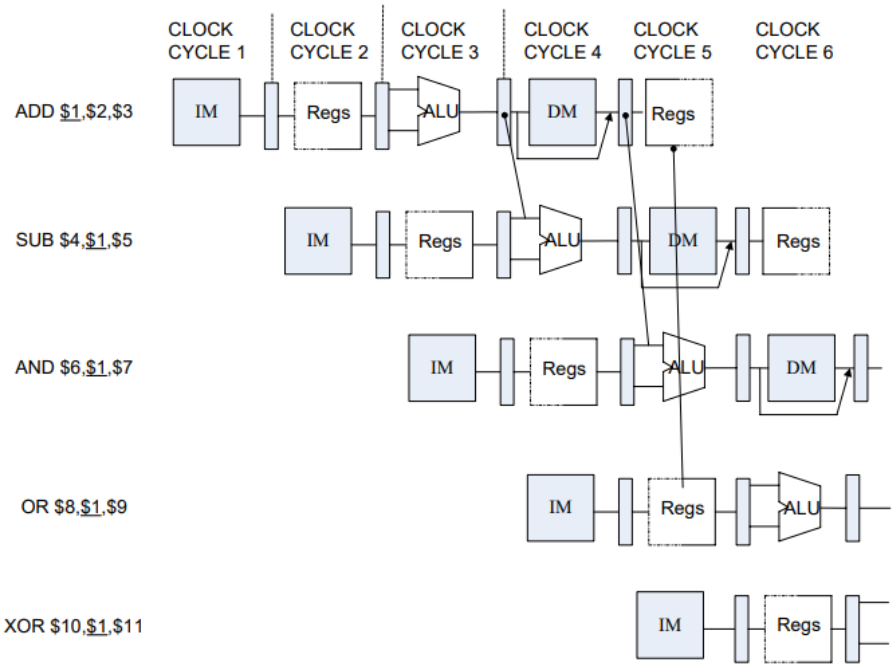
2.1 Data Hazard Stalls

- Data hazards happen mostly because that one instruction is reading a register which is NOT written back by the previous instruction yet. For example, in the following figure, the value of register `$1` is required by all instructions except the first one.



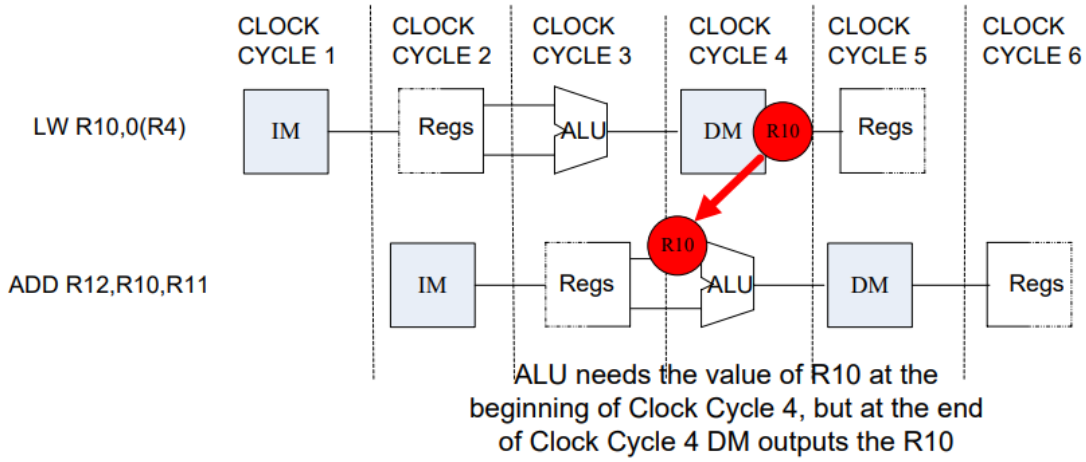
  In this case, our pipeline will have to **stall** until the result of the first instruction is written back to register `$1` (if we don't have a design for forwarding).

- We can minimize *Data Hazard Stalls* by using ***Forwarding***: transfer the needed (correct) data to other stages in the datapath even if the current instruction isn't finished. In most cases, the data hazard can be resolved by Forwarding (also called *bypassing*, or *short-circuiting*).



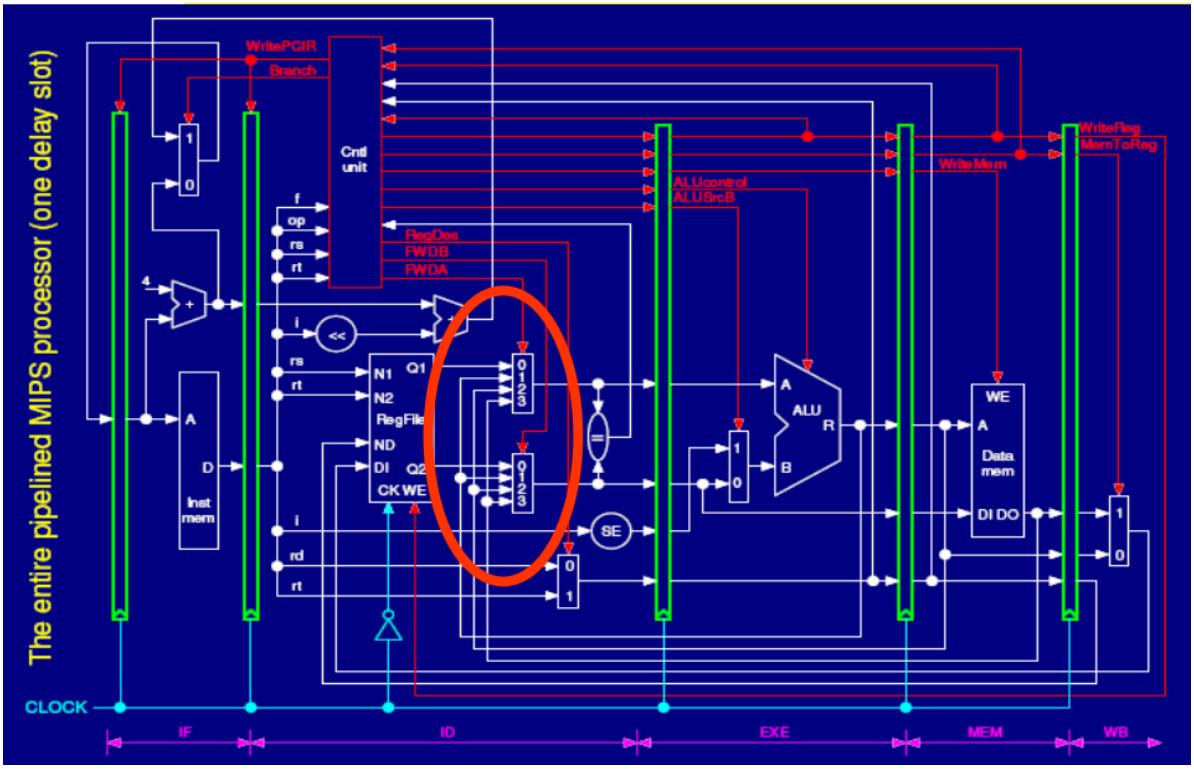
We can find that this time the pipeline isn't stalling.

- However, in some cases, data hazards can NOT be resolved by Forwarding and require pipeline stalls.



2.2 Datapath

The schematic of the modified datapath is shown in the following figure:



There're 3 parts to be added in order to implement pipeline forwarding:

1. rs data read from the register file (forwarded to ID stage).
2. rt data read from the register file (forwarded to ID stage).
3. Memory in/out forwarding.

2.3 Controller

To implement forwarding and make the additional components of the new datapath work as expected, some control signals need to be added to select the sources.

The relationship between control signals and MUXs is shown in the above figure as well.

§3 Main Instruments & Materials

3.1 Experiment Instruments

1. A Computer with ISE 14.7 Installed
2. SWORD Board

3.2 Experiment Materials

None.

§4 Experiment Procedure & Operations

4.1 Modify *datapath.v*

Add the 3 forwarding wires & MUXs to the datapath.

1. In **ID stage**, add rs and rt source selection:

```
reg [31:0] data_rs_fwd, data_rt_fwd;

always @(*) begin
    data_rs_fwd = data_rs;
    data_rt_fwd = data_rt;
    case (fwd_a_ctrl)
        0: data_rs_fwd = data_rs;
        1: data_rs_fwd = alu_out_exe;
        2: data_rs_fwd = alu_out_mem;
        3: data_rs_fwd = mem_din;
    endcase
    case (fwd_b_ctrl)
        0: data_rt_fwd = data_rt;
        1: data_rt_fwd = alu_out_exe;
        2: data_rt_fwd = alu_out_mem;
        3: data_rt_fwd = mem_din;
    endcase
    rs_rt_equal = (data_rs_fwd == data_rt_fwd);
end
```

2. In **MEM stage**, add memory source selection:

```
// MEM stage
always @(posedge clk) begin
    if (mem_rst) begin
        mem_valid <= 0;
        inst_addr_mem <= 0;
        inst_data_mem <= 0;
        regw_addr_mem <= 0;
        data_rt_mem <= 0;
        alu_out_mem <= 0;
        mem_ren_mem <= 0;
        mem_wen_mem <= 0;
        wb_data_src_mem <= 0;
        wb_wen_mem <= 0;
        fwd_m_mem <= 0;
        is_load_mem <= 0;
    end
    else if (mem_en) begin
        mem_valid <= exe_valid;
        inst_addr_mem <= inst_addr_exe;
        inst_data_mem <= inst_data_exe;
        regw_addr_mem <= regw_addr_exe;
        data_rt_mem <= data_rt_exe;
        alu_out_mem <= alu_out_exe;
        mem_ren_mem <= mem_ren_exe;
        mem_wen_mem <= mem_wen_exe;
        wb_data_src_mem <= wb_data_src_exe;
        wb_wen_mem <= wb_wen_exe;
        fwd_m_mem <= fwd_m_exe;
        is_load_mem <= is_load_exe;
```

```
    end
end

assign
    mem_ren = mem_ren_mem,
    mem_wen = mem_wen_mem,
    mem_addr = alu_out_mem,
    mem_dout = fwd_m_mem ? regw_data_wb : data_rt_mem;  // Memory source
selection
```

4.2 Modify *controller.v*

Add **Forwarding Control** to the pipeline control of the controller.

```
always @(*) begin
    reg_stall = 0;
    fwd_a = 0;
    fwd_b = 0;
    fwd_m = 0;
    if (rs_used && addr_rs != 0) begin
        if (regw_addr_exe == addr_rs && wb_wen_exe) begin
            if (is_load_exe)
                reg_stall = 1;
            else
                fwd_a = 1;
        end
        else if (regw_addr_mem == addr_rs && wb_wen_mem) begin
            if (is_load_mem)
                fwd_a = 3;
            else
                fwd_a = 2;
        end
    end
    if (rt_used && addr_rt != 0) begin
        if (regw_addr_exe == addr_rt && wb_wen_exe) begin
            if (is_load_exe) begin
                if (is_store)
                    fwd_m = 1;
                else
                    reg_stall = 1;
            end
            else
                fwd_b = 1;
        end
        else if (regw_addr_mem == addr_rt && wb_wen_mem) begin
            if (is_load_mem)
                fwd_b = 3;
            else
                fwd_b = 2;
        end
    end
end
```

4.3 Verify the Forwarding Design

1. Use the program provided by the template (`inst_mem.hex`) to verify the implementation of our *Pipelined CPU*. The code is provided in hexadecimal, which is very difficult for humans to understand. Converting it to something human-readable using a disassembler is a great idea. The translated program (in MIPS) is as the following:

code > inst_mem.asm

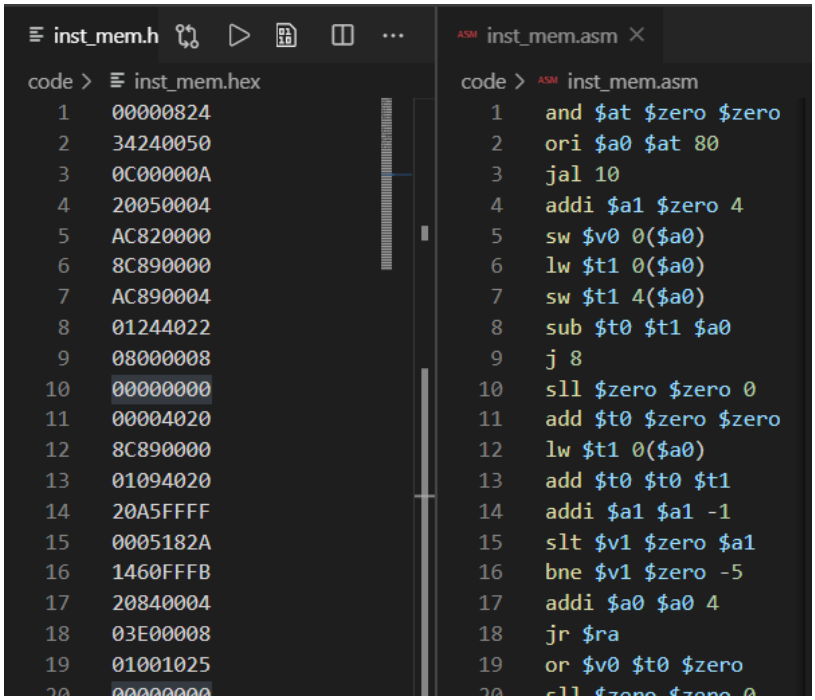
```
and $at $zero $zero
ori $a0 $at 80
addi $a1 $zero 4
jal 10
sw $v0 0($a0)
lw $t1 0($a0)
sw $t1 4($a0)
sub $t0 $t1 $a0
j 8
sll $zero $zero 0
add $t0 $zero $zero
lw $t1 0($a0)
add $t0 $t0 $t1
addi $a1 $a1 -1
addi $a0 $a0 4
slt $v1 $zero $a1
bne $v1 $zero -6
or $v0 $t0 $zero
jr $ra
```

2. Open the *ISE Project* and *Generate Programming File* of the top module, then upload the *.bit file* to the SWORD board to see whether our pipelined CPU works as desired.

§5 Results & Analysis

5.1 Function Verification

1. The hexadecimal file used to verify the design and its MIPS assembly comparison is show in the following figure:



2. The *Programming File* of the top module was **successfully generated and uploaded** to the SWORD board.
3. Turning on `SW[0]` on board makes the CPU enter *single-step debug* mode, during which time the *bottom-left* `BTN` makes it step forward. All debug information is shown on the *VGA display* connected to the board.

   You can refer to the *appended video clip* for the full execution progress of the above program.

   - When executing instruction 2 (`ori $a0 $at 80`) **without the forwarding components**, the pipeline would *stall* since it reads `$at`, which is the destination register of the previous instruction.
   - **After we added the forwarding support** to the pipeline, however, the pipeline *didn't stall* while executing instruction `ori $a0 $at 80`. That means that our forwarding design worked.
4. According to our observation, the program was executed **as desired** on the SWORD board. **We concluded that our *Pipelined CPU* was working as expected.**

§6 Discussion & Experience

In this lab course, I understood the principles of *pipeline forwarding* and took it into practice. I successfully implemented the Pipeline Forwarding Detection & Control components under the help of online tutorials and the help of my friends.

Appendix A. *controller.v*

```
1  `include "define.vh"
2
3  module controller (/*AUTOARG*/
4      input wire clk,  // main clock
5      input wire rst,  // synchronous reset
6      // debug
7      `ifdef DEBUG
8      input wire debug_en,  // debug enable
9      input wire debug_step,  // debug step clock
10     `endif
11     // instruction decode
12     input wire [31:0] inst,  // instruction
13     input wire rs_rt_equal,  // whether data from RS and RT are equal
14     input wire is_load_exe,  // whether instruction in EXE stage is load
   instruction
15     input wire [4:0] regw_addr_exe,  // register write address from EXE
   stage
16     input wire wb_wen_exe,  // register write enable signal feedback from
   EXE stage
17     input wire is_load_mem,  // whether instruction in MEM stage is load
   instruction
18     input wire [4:0] regw_addr_mem,  // register write address from MEM
   stage
19     input wire wb_wen_mem,  // register write enable signal feedback from
   MEM stage
20     output reg [1:0] pc_src,  // how would PC change to next
21     output reg imm_ext,  // whether using sign extended to immediate data
22     output reg [1:0] exe_a_src,  // data source of operand A for ALU
23     output reg [1:0] exe_b_src,  // data source of operand B for ALU
24     output reg [3:0] exe_alu_oper,  // ALU operation type
25     output reg mem_ren,  // memory read enable signal
26     output reg mem_wen,  // memory write enable signal
27     output reg [1:0] wb_addr_src,  // address source to write data back to
   registers
28     output reg wb_data_src,  // data source of data being written back to
   registers
29     output reg wb_wen,  // register write enable signal
30     output reg [1:0] fwd_a,  // forwarding selection for channel A
31     output reg [1:0] fwd_b,  // forwarding selection for channel B
32     output reg fwd_m,  // forwarding selection for memory
33     output reg is_load,  // whether current instruction is load instruction
34     output reg unrecognized,  // whether current instruction can not be
   recognized
35     // pipeline control
36     output reg if_rst,  // stage reset signal
37     output reg if_en,  // stage enable signal
38     input wire if_valid,  // stage valid flag
39     output reg id_rst,
40     output reg id_en,
41     input wire id_valid,
42     output reg exe_rst,
43     output reg exe_en,
44     input wire exe_valid,
45     output reg mem_rst,
```

```
46      output reg mem_en,
47      input wire mem_valid,
48      output reg wb_rst,
49      output reg wb_en,
50      input wire wb_valid
51      );
52
53      `include "mips_define.vh"
54
55      // instruction decode
56      reg rs_used, rt_used;
57      reg is_store;
58
59      always @(*) begin
60          pc_src = PC_NEXT;
61          imm_ext = 0;
62          exe_a_src = EXE_A_RS;
63          exe_b_src = EXE_B_RT;
64          exe_alu_oper = EXE_ALU_ADD;
65          mem_ren = 0;
66          mem_wen = 0;
67          wb_addr_src = WB_ADDR_RD;
68          wb_data_src = WB_DATA_ALU;
69          wb_wen = 0;
70          rs_used = 0;
71          rt_used = 0;
72          is_load = 0;
73          is_store = 0;
74          unrecognized = 0;
75          case (inst[31:26])
76              INST_R: begin
77                  case (inst[5:0])
78                      R_FUNC_JR: begin
79                          pc_src = PC_JR;
80                          rs_used = 1;
81                      end
82                      R_FUNC_ADD: begin
83                          exe_alu_oper = EXE_ALU_ADD;
84                          wb_addr_src = WB_ADDR_RD;
85                          wb_data_src = WB_DATA_ALU;
86                          wb_wen = 1;
87                          rs_used = 1;
88                          rt_used = 1;
89                      end
90                      R_FUNC_SUB: begin
91                          exe_alu_oper = EXE_ALU_SUB;
92                          wb_addr_src = WB_ADDR_RD;
93                          wb_data_src = WB_DATA_ALU;
94                          wb_wen = 1;
95                          rs_used = 1;
96                          rt_used = 1;
97                      end
98                      R_FUNC_AND: begin
99                          exe_alu_oper = EXE_ALU_AND;
100                         wb_addr_src = WB_ADDR_RD;
101                         wb_data_src = WB_DATA_ALU;
102                         wb_wen = 1;
103                         rs_used = 1;
```

```
                        rt_used = 1;
                    end
                    R_FUNC_OR: begin
                        exe_alu_oper = EXE_ALU_OR;
                        wb_addr_src = WB_ADDR_RD;
                        wb_data_src = WB_DATA_ALU;
                        wb_wen = 1;
                        rs_used = 1;
                        rt_used = 1;
                    end
                    R_FUNC_SLT: begin
                        exe_alu_oper = EXE_ALU_SLT;
                        wb_addr_src = WB_ADDR_RD;
                        wb_data_src = WB_DATA_ALU;
                        wb_wen = 1;
                        rs_used = 1;
                        rt_used = 1;
                    end
                    default: begin
                        unrecognized = 1;
                    end
                endcase
            end
            INST_J: begin
                pc_src = PC_JUMP;
            end
            INST_JAL: begin
                pc_src = PC_JUMP;
                exe_a_src = EXE_A_LINK;
                exe_b_src = EXE_B_LINK;
                exe_alu_oper = EXE_ALU_ADD;
                wb_addr_src = WB_ADDR_LINK;
                wb_data_src = WB_DATA_ALU;
                wb_wen = 1;
            end
            INST_BEQ: begin
                if (rs_rt_equal) begin
                    pc_src = PC_BRANCH;
                end
                imm_ext = 1;
                rs_used = 1;
                rt_used = 1;
            end
            INST_BNE: begin
                if (~rs_rt_equal) begin
                    pc_src = PC_BRANCH;
                end
                imm_ext = 1;
                rs_used = 1;
                rt_used = 1;
            end
            INST_ADDI: begin
                imm_ext = 1;
                exe_b_src = EXE_B_IMM;
                exe_alu_oper = EXE_ALU_ADD;
                wb_addr_src = WB_ADDR_RT;
                wb_data_src = WB_DATA_ALU;
                wb_wen = 1;
```

```
                rs_used = 1;
            end
            INST_ANDI: begin
                imm_ext = 0;
                exe_b_src = EXE_B_IMM;
                exe_alu_oper = EXE_ALU_AND;
                wb_addr_src = WB_ADDR_RT;
                wb_data_src = WB_DATA_ALU;
                wb_wen = 1;
                rs_used = 1;
            end
            INST_ORI: begin
                imm_ext = 0;
                exe_b_src = EXE_B_IMM;
                exe_alu_oper = EXE_ALU_OR;
                wb_addr_src = WB_ADDR_RT;
                wb_data_src = WB_DATA_ALU;
                wb_wen = 1;
                rs_used = 1;
            end
            INST_LW: begin
                imm_ext = 1;
                exe_b_src = EXE_B_IMM;
                exe_alu_oper = EXE_ALU_ADD;
                mem_ren = 1;
                wb_addr_src = WB_ADDR_RT;
                wb_data_src = WB_DATA_MEM;
                wb_wen = 1;
                is_load = 1;
                rs_used = 1;
            end
            INST_SW: begin
                imm_ext = 1;
                exe_b_src = EXE_B_IMM;
                exe_alu_oper = EXE_ALU_ADD;
                mem_wen = 1;
                is_store = 1;
                rs_used = 1;
                rt_used = 1;
            end
            default: begin
                unrecognized = 1;
            end
        endcase
    end

    // pipeline control
    reg reg_stall = 0;
    wire [4:0] addr_rs, addr_rt;

    assign
        addr_rs = inst[25:21],
        addr_rt = inst[20:16];

    always @(*) begin
        reg_stall = 0;
        fwd_a = 0;
        fwd_b = 0;
```

```
        fwd_m = 0;
        if (rs_used && addr_rs != 0) begin
            if (regw_addr_exe == addr_rs && wb_wen_exe) begin
                if (is_load_exe)
                    reg_stall = 1;
                else
                    fwd_a = 1;
            end
            else if (regw_addr_mem == addr_rs && wb_wen_mem) begin
                if (is_load_mem)
                    fwd_a = 3;
                else
                    fwd_a = 2;
            end
        end
        if (rt_used && addr_rt != 0) begin
            if (regw_addr_exe == addr_rt && wb_wen_exe) begin
                if (is_load_exe) begin
                    if (is_store)
                        fwd_m = 1;
                    else
                        reg_stall = 1;
                end
                else
                    fwd_b = 1;
            end
            else if (regw_addr_mem == addr_rt && wb_wen_mem) begin
                if (is_load_mem)
                    fwd_b = 3;
                else
                    fwd_b = 2;
            end
        end
    end

    `ifdef DEBUG
    reg debug_step_prev;

    always @(posedge clk) begin
        debug_step_prev <= debug_step;
    end
    `endif

    always @(*) begin
        if_rst = 0;
        if_en = 1;
        id_rst = 0;
        id_en = 1;
        exe_rst = 0;
        exe_en = 1;
        mem_rst = 0;
        mem_en = 1;
        wb_rst = 0;
        wb_en = 1;
        if (rst) begin
            if_rst = 1;
            id_rst = 1;
            exe_rst = 1;
```

```
            mem_rst = 1;
            wb_rst = 1;
        end
        `ifdef DEBUG
        // suspend and step execution
        else if ((debug_en) && ~(~debug_step_prev && debug_step)) begin
            if_en = 0;
            id_en = 0;
            exe_en = 0;
            mem_en = 0;
            wb_en = 0;
        end
        `endif
        // this stall indicate that ID is waiting for previous LW
instruction, should insert one NOP between ID and EXE.
        else if (reg_stall) begin
            if_en = 0;
            id_en = 0;
            exe_rst = 1;
        end
    end

endmodule
```

Appendix B. *datapath.v*

```
1  `include "define.vh"
2
3  module datapath (
4      input wire clk,  // main clock
5      // debug
6      `ifdef DEBUG
7      input wire [5:0] debug_addr,  // debug address
8      output wire [31:0] debug_data,  // debug data
9      `endif
10     // control signals
11     output reg [31:0] inst_data_id,  // instruction
12     output reg rs_rt_equal,  // whether data from RS and RT are equal
13     output reg is_load_exe,  // whether instruction in EXE stage is load
   instruction
14     output reg [4:0] regw_addr_exe,  // register write address from EXE
   stage
15     output reg wb_wen_exe,  // register write enable signal feedback from
   EXE stage
16     output reg is_load_mem,  // whether instruction in MEM stage is load
   instruction
17     output reg [4:0] regw_addr_mem,  // register write address from MEM
   stage
18     output reg wb_wen_mem,  // register write enable signal feedback from
   MEM stage
19     input wire [1:0] pc_src_ctrl,  // how would PC change to next
20     input wire imm_ext_ctrl,  // whether using sign extended to immediate
   data
21     input wire [1:0] exe_a_src_ctrl,  // data source of operand A for ALU
22     input wire [1:0] exe_b_src_ctrl,  // data source of operand B for ALU
23     input wire [3:0] exe_alu_oper_ctrl,  // ALU operation type
24     input wire mem_ren_ctrl,  // memory read enable signal
25     input wire mem_wen_ctrl,  // memory write enable signal
26     input wire [1:0] wb_addr_src_ctrl,  // address source to write data
   back to registers
27     input wire wb_data_src_ctrl,  // data source of data being written back
   to registers
28     input wire wb_wen_ctrl,  // register write enable signal
29     input wire [1:0] fwd_a_ctrl,  // forwarding selection for channel A
30     input wire [1:0] fwd_b_ctrl,  // forwarding selection for channel B
31     input wire fwd_m_ctrl,  // forwarding selection for memory
32     input wire is_load_ctrl,  // whether current instruction is load
   instruction
33     // IF signals
34     input wire if_rst,  // stage reset signal
35     input wire if_en,  // stage enable signal
36     output reg if_valid,  // working flag
37     output reg inst_ren,  // instruction read enable signal
38     output reg [31:0] inst_addr,  // address of instruction needed
39     input wire [31:0] inst_data,  // instruction fetched
40     // ID signals
41     input wire id_rst,
42     input wire id_en,
43     output reg id_valid,
44     // EXE signals
```

```
45      input wire exe_rst,
46      input wire exe_en,
47      output reg exe_valid,
48      // MEM signals
49      input wire mem_rst,
50      input wire mem_en,
51      output reg mem_valid,
52      output wire mem_ren,  // memory read enable signal
53      output wire mem_wen,  // memory write enable signal
54      output wire [31:0] mem_addr,  // address of memory
55      output wire [31:0] mem_dout,  // data writing to memory
56      input wire [31:0] mem_din,  // data read from memory
57      // WB signals
58      input wire wb_rst,
59      input wire wb_en,
60      output reg wb_valid
61      );
62
63      `include "mips_define.vh"
64
65      // control signals
66      reg [1:0] exe_a_src_exe, exe_b_src_exe;
67      reg [3:0] exe_alu_oper_exe;
68      reg mem_ren_exe, mem_ren_mem;
69      reg mem_wen_exe, mem_wen_mem;
70      reg wb_data_src_exe, wb_data_src_mem, wb_data_src_wb;
71      reg fwd_m_exe, fwd_m_mem;
72
73      // IF signals
74      wire [31:0] inst_addr_next;
75
76      // ID signals
77      reg [31:0] inst_addr_id;
78      reg [31:0] inst_addr_next_id;
79      reg [4:0] regw_addr_id;
80      wire [4:0] addr_rs, addr_rt, addr_rd;
81      reg [31:0] data_rs_fwd, data_rt_fwd;
82      wire [31:0] data_rs, data_rt, data_imm;
83
84      // EXE signals
85      reg [31:0] inst_addr_exe;
86      reg [31:0] inst_addr_next_exe;
87      reg [31:0] inst_data_exe;
88      reg [31:0] data_rs_exe, data_rt_exe, data_imm_exe;
89      reg [31:0] opa_exe, opb_exe;
90      wire [31:0] alu_out_exe;
91
92      // MEM signals
93      reg [31:0] inst_addr_mem;
94      reg [31:0] inst_data_mem;
95      reg [31:0] data_rt_mem;
96      reg [31:0] alu_out_mem;
97
98      // WB signals
99      reg wb_wen_wb;
100     reg [31:0] alu_out_wb;
101     reg [31:0] mem_din_wb;
102     reg [4:0] regw_addr_wb;
```

```
103     reg [31:0] regw_data_wb;
104 
105     // debug
106     `ifdef DEBUG
107     wire [31:0] debug_data_reg;
108     reg [31:0] debug_data_signal;
109 
110     always @(posedge clk) begin
111         case (debug_addr[4:0])
112             0: debug_data_signal <= inst_addr;
113             1: debug_data_signal <= inst_data;
114             2: debug_data_signal <= inst_addr_id;
115             3: debug_data_signal <= inst_data_id;
116             4: debug_data_signal <= inst_addr_exe;
117             5: debug_data_signal <= inst_data_exe;
118             6: debug_data_signal <= inst_addr_mem;
119             7: debug_data_signal <= inst_data_mem;
120             8: debug_data_signal <= {27'b0, addr_rs};
121             9: debug_data_signal <= data_rs;
122             10: debug_data_signal <= {27'b0, addr_rt};
123             11: debug_data_signal <= data_rt;
124             12: debug_data_signal <= data_imm;
125             13: debug_data_signal <= opa_exe;
126             14: debug_data_signal <= opb_exe;
127             15: debug_data_signal <= alu_out_exe;
128             16: debug_data_signal <= 0;
129             17: debug_data_signal <= {16'b0, 2'b0, fwd_a_ctrl, 2'b0,
    fwd_b_ctrl, 7'b0, fwd_m_ctrl};
130             18: debug_data_signal <= {19'b0, inst_ren, 7'b0, mem_ren, 3'b0,
    mem_wen};
131             19: debug_data_signal <= mem_addr;
132             20: debug_data_signal <= mem_din;
133             21: debug_data_signal <= mem_dout;
134             22: debug_data_signal <= {27'b0, regw_addr_wb};
135             23: debug_data_signal <= regw_data_wb;
136             default: debug_data_signal <= 32'hFFFF_FFFF;
137         endcase
138     end
139 
140     assign
141         debug_data = debug_addr[5] ? debug_data_signal : debug_data_reg;
142     `endif
143 
144     // IF stage
145     assign
146         inst_addr_next = inst_addr + 4;
147 
148     always @(*) begin
149         if_valid = ~if_rst & if_en;
150         inst_ren = ~if_rst;
151     end
152 
153     always @(posedge clk) begin
154         if (if_rst) begin
155             inst_addr <= 0;
156         end
157         else if (if_en) begin
158             case (pc_src_ctrl)
```

```
                PC_NEXT: inst_addr <= inst_addr_next;
                PC_JUMP: inst_addr <= {inst_addr_id[31:28],
inst_data_id[25:0], 2'b0};
                PC_JR: inst_addr <= data_rs_fwd;
                PC_BRANCH: inst_addr <= inst_addr_next_id +
{data_imm[29:0], 2'b0};
            endcase
        end
    end

    // ID stage
    always @(posedge clk) begin
        if (id_rst) begin
            id_valid <= 0;
            inst_addr_id <= 0;
            inst_data_id <= 0;
            inst_addr_next_id <= 0;
        end
        else if (id_en) begin
            id_valid <= if_valid;
            inst_addr_id <= inst_addr;
            inst_data_id <= inst_data;
            inst_addr_next_id <= inst_addr_next;
        end
    end

    assign
        addr_rs = inst_data_id[25:21],
        addr_rt = inst_data_id[20:16],
        addr_rd = inst_data_id[15:11],
        data_imm = imm_ext_ctrl ? {{16{inst_data_id[15]}},
inst_data_id[15:0]} : {16'b0, inst_data_id[15:0]};

    always @(*) begin
        regw_addr_id = inst_data_id[15:11];
        case (wb_addr_src_ctrl)
            WB_ADDR_RD: regw_addr_id = addr_rd;
            WB_ADDR_RT: regw_addr_id = addr_rt;
            WB_ADDR_LINK: regw_addr_id = GPR_RA;
        endcase
    end

    regfile REGFILE (
        .clk(clk),
        `ifdef DEBUG
        .debug_addr(debug_addr[4:0]),
        .debug_data(debug_data_reg),
        `endif
        .addr_a(addr_rs),
        .data_a(data_rs),
        .addr_b(addr_rt),
        .data_b(data_rt),
        .en_w(wb_wen_wb),
        .addr_w(regw_addr_wb),
        .data_w(regw_data_wb)
        );

    always @(*) begin
```

```
214         data_rs_fwd = data_rs;
215         data_rt_fwd = data_rt;
216         case (fwd_a_ctrl)
217             0: data_rs_fwd = data_rs;
218             1: data_rs_fwd = alu_out_exe;
219             2: data_rs_fwd = alu_out_mem;
220             3: data_rs_fwd = mem_din;
221         endcase
222         case (fwd_b_ctrl)
223             0: data_rt_fwd = data_rt;
224             1: data_rt_fwd = alu_out_exe;
225             2: data_rt_fwd = alu_out_mem;
226             3: data_rt_fwd = mem_din;
227         endcase
228         rs_rt_equal = (data_rs_fwd == data_rt_fwd);
229     end
230 
231     // EXE stage
232     always @(posedge clk) begin
233         if (exe_rst) begin
234             exe_valid <= 0;
235             inst_addr_exe <= 0;
236             inst_data_exe <= 0;
237             inst_addr_next_exe <= 0;
238             regw_addr_exe <= 0;
239             exe_a_src_exe <= 0;
240             exe_b_src_exe <= 0;
241             data_rs_exe <= 0;
242             data_rt_exe <= 0;
243             data_imm_exe <= 0;
244             exe_alu_oper_exe <= 0;
245             mem_ren_exe <= 0;
246             mem_wen_exe <= 0;
247             wb_data_src_exe <= 0;
248             wb_wen_exe <= 0;
249             fwd_m_exe <= 0;
250             is_load_exe <= 0;
251         end
252         else if (exe_en) begin
253             exe_valid <= id_valid;
254             inst_addr_exe <= inst_addr_id;
255             inst_data_exe <= inst_data_id;
256             inst_addr_next_exe <= inst_addr_next_id;
257             regw_addr_exe <= regw_addr_id;
258             exe_a_src_exe <= exe_a_src_ctrl;
259             exe_b_src_exe <= exe_b_src_ctrl;
260             data_rs_exe <= data_rs_fwd;
261             data_rt_exe <= data_rt_fwd;
262             data_imm_exe <= data_imm;
263             exe_alu_oper_exe <= exe_alu_oper_ctrl;
264             mem_ren_exe <= mem_ren_ctrl;
265             mem_wen_exe <= mem_wen_ctrl;
266             wb_data_src_exe <= wb_data_src_ctrl;
267             wb_wen_exe <= wb_wen_ctrl;
268             fwd_m_exe <= fwd_m_ctrl;
269             is_load_exe <= is_load_ctrl;
270         end
271     end
```

```
272
273     always @(*) begin
274         opa_exe = data_rs_exe;
275         opb_exe = data_rt_exe;
276         case (exe_a_src_exe)
277             EXE_A_RS: opa_exe = data_rs_exe;
278             EXE_A_LINK: opa_exe = inst_addr_next_exe;
279         endcase
280         case (exe_b_src_exe)
281             EXE_B_RT: opb_exe = data_rt_exe;
282             EXE_B_IMM: opb_exe = data_imm_exe;
283             EXE_B_LINK: opb_exe = 32'h4;  // linked address is the next one
    of the delay slot
284         endcase
285     end
286
287     alu ALU (
288         .a(opa_exe),
289         .b(opb_exe),
290         .oper(exe_alu_oper_exe),
291         .result(alu_out_exe)
292         );
293
294     // MEM stage
295     always @(posedge clk) begin
296         if (mem_rst) begin
297             mem_valid <= 0;
298             inst_addr_mem <= 0;
299             inst_data_mem <= 0;
300             regw_addr_mem <= 0;
301             data_rt_mem <= 0;
302             alu_out_mem <= 0;
303             mem_ren_mem <= 0;
304             mem_wen_mem <= 0;
305             wb_data_src_mem <= 0;
306             wb_wen_mem <= 0;
307             fwd_m_mem <= 0;
308             is_load_mem <= 0;
309         end
310         else if (mem_en) begin
311             mem_valid <= exe_valid;
312             inst_addr_mem <= inst_addr_exe;
313             inst_data_mem <= inst_data_exe;
314             regw_addr_mem <= regw_addr_exe;
315             data_rt_mem <= data_rt_exe;
316             alu_out_mem <= alu_out_exe;
317             mem_ren_mem <= mem_ren_exe;
318             mem_wen_mem <= mem_wen_exe;
319             wb_data_src_mem <= wb_data_src_exe;
320             wb_wen_mem <= wb_wen_exe;
321             fwd_m_mem <= fwd_m_exe;
322             is_load_mem <= is_load_exe;
323         end
324     end
325
326     assign
327         mem_ren = mem_ren_mem,
328         mem_wen = mem_wen_mem,
```

```
329         mem_addr = alu_out_mem,
330         mem_dout = fwd_m_mem ? regw_data_wb : data_rt_mem;
331 
332     // WB stage
333     always @(posedge clk) begin
334         if (wb_rst) begin
335             wb_valid <= 0;
336             wb_wen_wb <= 0;
337             wb_data_src_wb <= 0;
338             regw_addr_wb <= 0;
339             alu_out_wb <= 0;
340             mem_din_wb <= 0;
341         end
342         else if (wb_en) begin
343             wb_valid <= mem_valid;
344             wb_wen_wb <= wb_wen_mem;
345             wb_data_src_wb <= wb_data_src_mem;
346             regw_addr_wb <= regw_addr_mem;
347             alu_out_wb <= alu_out_mem;
348             mem_din_wb <= mem_din;
349         end
350     end
351 
352     always @(*) begin
353         regw_data_wb = alu_out_wb;
354         case (wb_data_src_wb)
355             WB_DATA_ALU: regw_data_wb = alu_out_wb;
356             WB_DATA_MEM: regw_data_wb = mem_din_wb;
357         endcase
358     end
359 
360 endmodule
```