Computer Architecture

Lab 2 Report

Name:	Asudy Wang 王浚哲	**ID:**	3180103011	**Major:**	Computer Science & Technology
Course:	Computer Architecture		**Place:**	Room 301, Cao Guangbiao Building West Wing, Yuquan Campus	
Due Date:	2020-12-07	**Groupmate:**	Flaze He	**Instructor:**	Kai Bu

Table of Contents

Lab 2. 5-Stage Pipelined CPU

§1 Purposes & Requirements

1.1 Experiment Purpose

- Understand the **principles** of *Pipelined CPU*;
- Understand the **basic units** of *Pipelined CPU*;
- Understand the **working flow** of a *5-stage pipeline*;
- Master the method of implementing a simple *Pipelined CPU*;
- Master methods of program verification of simple *Pipelined CPU*.

1.2 Experiment Tasks

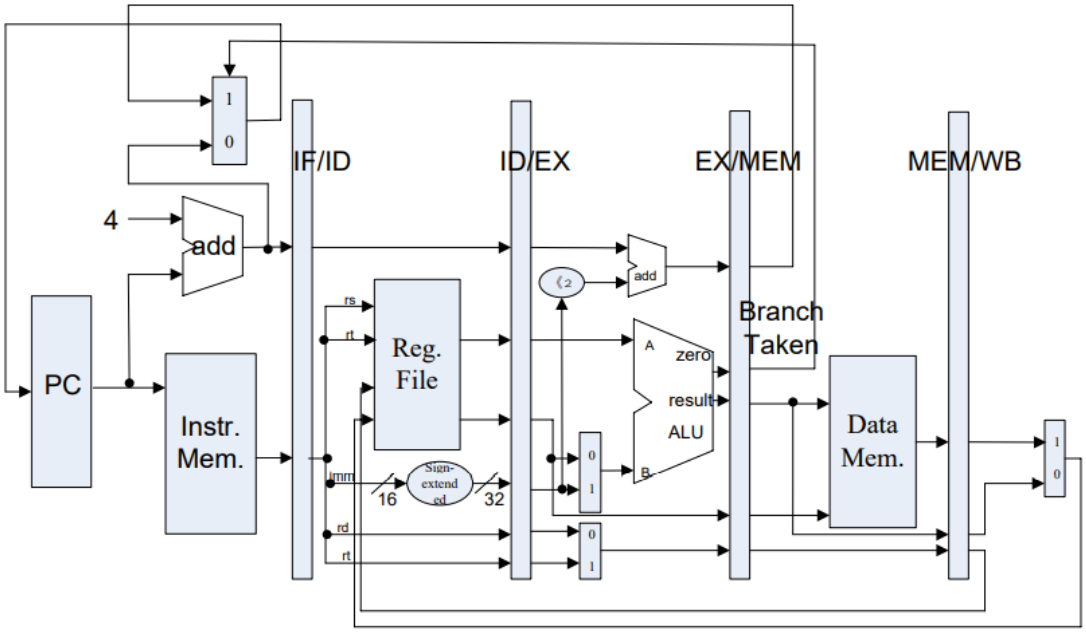
- Design the CPU **controller** & **datapath** of the *5-stage Pipelined CPU*.
    - 5-stage implementation
    - Register File
    - Memory (instruction & data memories)
    - Other basic & auxiliary units
- Verify the *Pipelined CPU* with program and observe the execution of the program.

§2 Contents & Principles

2.1 Datapath

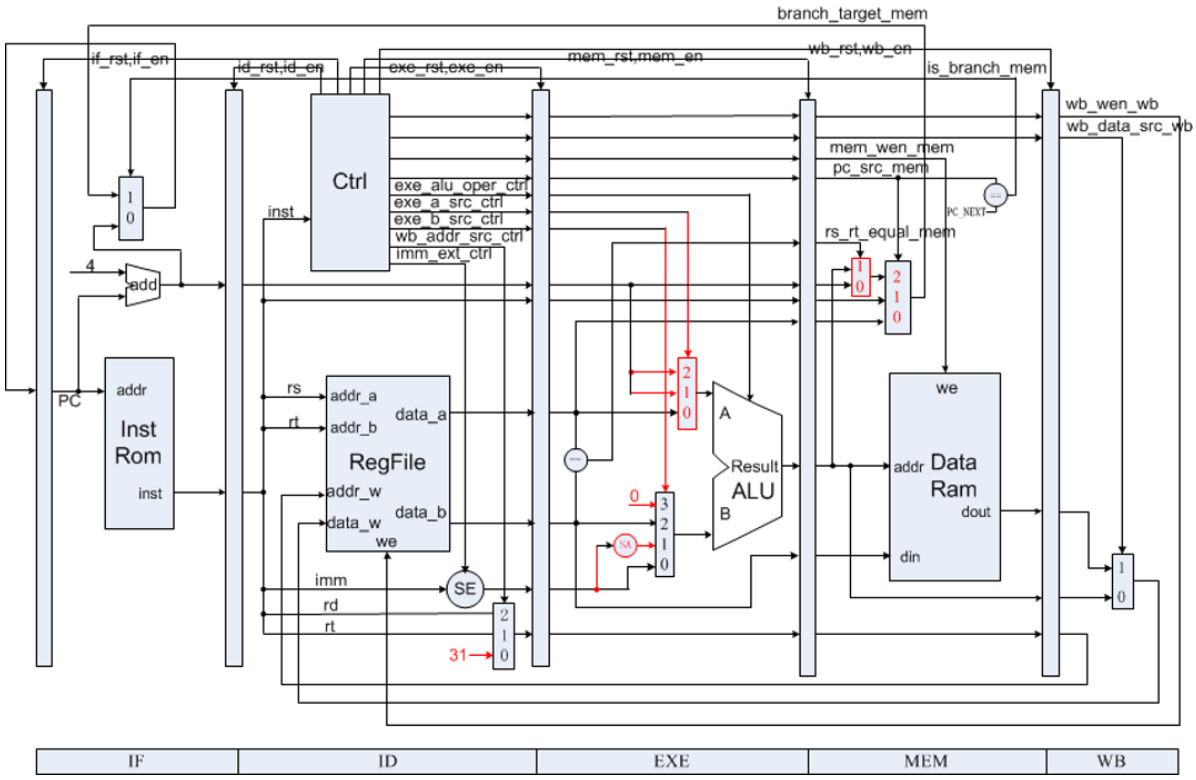
The **5-staged** pipelined datapath divides the workflow of a CPU into 5 specific stages: *IF* (Instruction Fetch), *ID* (Instruction Decode & register fetch), *EXE* (EXEcution), *MEM* (MEMory access) and *WB* (Write Back). The key idea of the *Pipelined CPU* is, whenever an instruction is using units in a certain stage, other instructons are able to use other stages **simultaneously**. Thus in a 5-staged pipeline, 5 instructions can be executed parallelly.

A simplified vesion of the pipelined datapath we're implementing in this lab is as the following.



2.2 Controller

The controller in the pipelined CPU translates the input **instruction** into several **control signals** asynchronously. Different control signals select inputs for datapath units, control write enable for registers and memories, and decide whether a certain stage in the datapath is enabled/cleared. A **datapath with controller** in the pipelined CPU is shown as the following figure:



2.3 Basic Units of a Pipelined CPU

1. CPU Controller
2. **Stage Registers**
3. Register File
    - Positive edge: transfer data for stages
    - Negative edge: write
4. Memories
    - Instruction Memory
        - Read-only, with width 32
        - Falling-edge triggered
    - Data Memory
        - Read and Write, with width 32
        - Falling-edge triggered
5. ALU
6. Other registers, Signed/Unsigned-extension unit, MUXs

§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

Thanks to the SWORD Website who provided us with very helpful Verilog implementations of all units we need except for the *controller* and the *datapath* (can be downloaded here), all we need to do in this lab is to fill in the blanks in the *controller & datapath* modules, and test if our code makes the CPU run expectedly.

4.1 Finish `controller.v`

1. Open file `controller.v` (in the files downloaded just now) in a text editor. We can find that some codes are masked by `???`.
2. Fill in the blanks in the **instruction decode part** according to the functions of each given instruction. All *parameter* definitions can be found in file `mips_define.vh`. Always use symbols provided in that file if possible.
    - For example, the decoding of `ADD` instruction is:

```
case ...
    R_FUNC_ADD: begin
        exe_alu_oper = EXE_ALU_ADD; // ALU operation type: +
        wb_addr_src = WB_ADDR_RD;   // Write back addr.: rd
        wb_data_src = WB_DATA_ALU;  // Write back data: ALUout
        wb_wen = 1;     // Write back REG enable: true
        rs_used = 1;    // rs register used: true
        rt_used = 1;    // rt register used: true
    end
...
```

    - To reduce the length of the report, the full code is provided in *Appendix A*.
3. Other `???`s lie in the **pipeline control part** which is implemented for possible *stalls*. Since *stall implementation is not required* in this lab (Lab 2), the detail will be discussed in the next lab report. However you may find the finished code in *Appendix A* as well.

4.2 Finish `datapath.v`

1. Open file `datapath.v` (in the files downloaded just now) in a text editor. We can find that some codes are masked by `???`.
2. Fill in the blanks about **write back addr selection**, **write back data selection**, **ALU input selection** and **target PC selection** according to meanings of the given control signals. All *parameter* definitions can be found in file `mips_define.vh`. Always use symbols provided in that file if possible.
    1. ***Write Back Addr Selection***

```
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
```

2. ***Write Back Data Selection***

```
always @(*) begin
    regw_data_wb = alu_out_wb;
    case (wb_data_src_wb)
        WB_DATA_ALU: regw_data_wb = alu_out_wb;
        WB_DATA_MEM: regw_data_wb = mem_din_wb;
    endcase
end
```

3. ***ALU Input Selection***

```
always @(*) begin
    opa_exe = data_rs_exe;
    opb_exe = data_rt_exe;
    case (exe_a_src_exe)    // ALU input A source selection
        EXE_A_RS: opa_exe = data_rs_exe;
        EXE_A_LINK: opa_exe = inst_addr_next_exe;
        EXE_A_BRANCH: opa_exe = inst_addr_next_exe;
    endcase
    case (exe_b_src_exe)    // ALU input B source selection
        EXE_B_RT: opb_exe = data_rt_exe;
        EXE_B_IMM: opb_exe = data_imm_exe;
        EXE_B_LINK: opb_exe = 32'h0;
        EXE_B_BRANCH: opb_exe = {data_imm_exe[29:0], 2'b0};
    endcase
end
```

4. ***Target PC Selection***

```
always @(*) begin
    case (pc_src_mem)
        PC_JUMP: branch_target_mem <= {
            inst_addr_mem[31:28], inst_data_mem[25:0], 2'b0
        };
        PC_JR: branch_target_mem <= data_rs_mem;
        PC_BEQ: branch_target_mem <= rs_rt_equal_mem ?
                alu_out_mem : inst_addr_next_mem ;
        PC_BNE: branch_target_mem <= rs_rt_equal_mem ?
                inst_addr_next_mem : alu_out_mem ;
        default: branch_target_mem <= inst_addr_next_mem;  // never
used
    endcase
end
```

3. The full code of *datapath* is provided in *Appendix B* of this report.

4.3 Verify the Pipelined CPU 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 > ASM inst_mem.asm
1   and $at $zero $zero
2   ori $a0 $at 80
3   addi $a1 $zero 4
4   jal 10
5   sw $v0 0($a0)
6   lw $t1 0($a0)
7   sw $t1 4($a0)
8   sub $t0 $t1 $a0
9   j 8
10  sll $zero $zero 0
11  add $t0 $zero $zero
12  lw $t1 0($a0)
13  add $t0 $t0 $t1
14  addi $a1 $a1 -1
15  addi $a0 $a0 4
16  slt $v1 $zero $a1
17  bne $v1 $zero -6
18  or $v0 $t0 $zero
19  jr $ra
```

2. Open the *ISE Project* and *Generate Programming File* of the top module, then upload the *.bit file* to the SWORD borad 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:

```
code > inst_mem.hex
1   00000824
2   34240050
3   20050004
4   0C00000A
5   AC820000
6   8C890000
7   AC890004
8   01244022
9   08000008
10  00000000
11  00004020
12  8C890000
13  01094020
14  20A5FFFF
15  20840004
16  0005182A
17  1460FFFA
18  01001025
19  03E00008
```

```
code > ASM inst_mem.asm
1   and $at $zero $zero
2   ori $a0 $at 80
3   addi $a1 $zero 4
4   jal 10
5   sw $v0 0($a0)
6   lw $t1 0($a0)
7   sw $t1 4($a0)
8   sub $t0 $t1 $a0
9   j 8
10  sll $zero $zero 0
11  add $t0 $zero $zero
12  lw $t1 0($a0)
13  add $t0 $t0 $t1
14  addi $a1 $a1 -1
15  addi $a0 $a0 4
16  slt $v1 $zero $a1
17  bne $v1 $zero -6
18  or $v0 $t0 $zero
19  jr $ra
```

2. The *Programming File* of the top module was **successfully generated and uploaded** to the SWORD board.

3. Turning on `SW[0]` on borad 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 zip file* for the photos of the full execution progress of the above program.

4. Accoding to our observation, the program was exected **as desired** on the SWORD borad. **We concluded that our *Pipelined CPU* was working as expected.**

§6 Discussion & Experience

In this experiment, I got to know the detailed implementation of a pipelined CPU rather than the theorems.

Before I started this lab, I had no idea how to implement such a pipelined CPU in practice. The template provided by SWORD really helped me to construct an outline of the design.

When we tried to fill in the blanks in those files, we suffered from figuring out what all the control signals actually controls since there isn't a visualized schematic provided. With the help of the comments and our teammates, we managed to figure that out and our understanding about the pipeline was further deepend.

Appendix A. *controller.v*

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

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

    /* pipeline control (stall) */
    reg reg_stall;
    reg branch_stall;
    wire [4:0] addr_rs, addr_rt;

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

    always @(*) begin
        reg_stall = 0;
        if (rs_used && addr_rs != 0) begin
            if (regw_addr_exe == addr_rs && wb_wen_exe) begin
                reg_stall = 1;
            end
            else if (regw_addr_mem == addr_rs && wb_wen_mem) begin
```

```
220                 reg_stall = 1;
221             end
222         end
223         if (rt_used && addr_rt != 0) begin
224             if (regw_addr_exe == addr_rt && wb_wen_exe) begin
225                 reg_stall = 1;
226             end
227             else if (regw_addr_mem == addr_rt && wb_wen_mem) begin
228                 reg_stall = 1;
229             end
230         end
231     end
232 
233     always @(*) begin
234         branch_stall = 0;
235         if (pc_src != PC_NEXT || is_branch_exe || is_branch_mem)
236             branch_stall = 1;
237     end
238 
239     `ifdef DEBUG
240     reg debug_step_prev;
241 
242     always @(posedge clk) begin
243         debug_step_prev <= debug_step;
244     end
245     `endif
246 
247     always @(*) begin
248         if_rst = 0;
249         if_en = 1;
250         id_rst = 0;
251         id_en = 1;
252         exe_rst = 0;
253         exe_en = 1;
254         mem_rst = 0;
255         mem_en = 1;
256         wb_rst = 0;
257         wb_en = 1;
258         if (rst) begin
259             if_rst = 1;
260             id_rst = 1;
261             exe_rst = 1;
262             mem_rst = 1;
263             wb_rst = 1;
264         end
265         `ifdef DEBUG
266         // suspend and step execution
267         else if ((debug_en) && ~(~debug_step_prev && debug_step)) begin
268             if_en = 0;
269             id_en = 0;
270             exe_en = 0;
271             mem_en = 0;
272             wb_en = 0;
273         end
274         `endif
275         // this stall indicate that ID is waiting for previous instruction,
    should insert NOPs between ID and EXE.
276         else if (reg_stall) begin
```

```
277             if_en = 0;
278             id_en = 0;
279             exe_rst = 1;
280         end
281         // this stall indicate that a jump/branch instruction is running,
    so that 3 NOP should be inserted between IF and ID
282         else if (branch_stall) begin
283             id_rst = 1;
284         end
285     end
286 
287 endmodule
```

Appendix B. *datapath.v*

```
1  `include "define.vh"
2
3  /**
4   * Data Path for MIPS 5-stage pipelined CPU.
5   * Author: Zhao, Hongyu  <power_zhy@foxmail.com>
6   */
7  module datapath (
8      input wire clk,  // main clock
9      // debug
10     `ifdef DEBUG
11     input wire [5:0] debug_addr,  // debug address
12     output wire [31:0] debug_data,  // debug data
13     `endif
14     // control signals
15     output reg [31:0] inst_data_id,  // instruction
16     output reg is_branch_exe,  // whether instruction in EXE stage is
   jump/branch instruction
17     output reg [4:0] regw_addr_exe,  // register write address from EXE
   stage
18     output reg wb_wen_exe,  // register write enable signal feedback from
   EXE stage
19     output reg is_branch_mem,  // whether instruction in MEM stage is
   jump/branch instruction
20     output reg [4:0] regw_addr_mem,  // register write address from MEM
   stage
21     output reg wb_wen_mem,  // register write enable signal feedback from
   MEM stage
22     input wire [2:0] pc_src_ctrl,  // how would PC change to next
23     input wire imm_ext_ctrl,  // whether using sign extended to immediate
   data
24     input wire [1:0] exe_a_src_ctrl,  // data source of operand A for ALU
25     input wire [1:0] exe_b_src_ctrl,  // data source of operand B for ALU
26     input wire [3:0] exe_alu_oper_ctrl,  // ALU operation type
27     input wire mem_ren_ctrl,  // memory read enable signal
28     input wire mem_wen_ctrl,  // memory write enable signal
29     input wire [1:0] wb_addr_src_ctrl,  // address source to write data
   back to registers
30     input wire wb_data_src_ctrl,  // data source of data being written back
   to registers
31     input wire wb_wen_ctrl,  // register write enable signal
32     // IF signals
33     input wire if_rst,  // stage reset signal
34     input wire if_en,  // stage enable signal
35     output reg if_valid,  // working flag
36     output reg inst_ren,  // instruction read enable signal
37     output reg [31:0] inst_addr,  // address of instruction needed
38     input wire [31:0] inst_data,  // instruction fetched
39     // ID signals
40     input wire id_rst,
41     input wire id_en,
42     output reg id_valid,
43     // EXE signals
44     input wire exe_rst,
45     input wire exe_en,
```

```
46      output reg exe_valid,
47      // MEM signals
48      input wire mem_rst,
49      input wire mem_en,
50      output reg mem_valid,
51      output wire mem_ren,  // memory read enable signal
52      output wire mem_wen,  // memory write enable signal
53      output wire [31:0] mem_addr,  // address of memory
54      output wire [31:0] mem_dout,  // data writing to memory
55      input wire [31:0] mem_din,  // data read from memory
56      // WB signals
57      input wire wb_rst,
58      input wire wb_en,
59      output reg wb_valid
60      );
61
62      `include "mips_define.vh"
63
64      // control signals
65      reg [2:0] pc_src_exe, pc_src_mem;
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
72      // IF signals
73      wire [31:0] inst_addr_next;
74
75      // ID signals
76      reg [31:0] inst_addr_id;
77      reg [31:0] inst_addr_next_id;
78      reg [4:0] regw_addr_id;
79      wire [4:0] addr_rs, addr_rt, addr_rd;
80      wire [31:0] data_rs, data_rt, data_imm;
81
82      // EXE signals
83      reg [31:0] inst_addr_exe;
84      reg [31:0] inst_addr_next_exe;
85      reg [31:0] inst_data_exe;
86      reg [31:0] data_rs_exe, data_rt_exe, data_imm_exe;
87      reg [31:0] opa_exe, opb_exe;
88      wire [31:0] alu_out_exe;
89      wire rs_rt_equal_exe;
90
91      // MEM signals
92      reg [31:0] inst_addr_mem;
93      reg [31:0] inst_addr_next_mem;
94      reg [31:0] inst_data_mem;
95      reg [4:0] data_rs_mem;
96      reg [31:0] data_rt_mem;
97      reg [31:0] alu_out_mem;
98      reg [31:0] branch_target_mem;
99      reg rs_rt_equal_mem;
100
101     // WB signals
102     reg wb_wen_wb;
103     reg [31:0] alu_out_wb;
```

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

```
            if (is_branch_mem)
                inst_addr <= branch_target_mem;
            else
                inst_addr <= inst_addr_next;
        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)
        );

    // EXE stage
    always @(posedge clk) begin
        if (exe_rst) begin
            exe_valid <= 0;
```

```
            inst_addr_exe <= 0;
            inst_data_exe <= 0;
            inst_addr_next_exe <= 0;
            regw_addr_exe <= 0;
            pc_src_exe <= 0;
            exe_a_src_exe <= 0;
            exe_b_src_exe <= 0;
            data_rs_exe <= 0;
            data_rt_exe <= 0;
            data_imm_exe <= 0;
            exe_alu_oper_exe <= 0;
            mem_ren_exe <= 0;
            mem_wen_exe <= 0;
            wb_data_src_exe <= 0;
            wb_wen_exe <= 0;
        end
        else if (exe_en) begin
            exe_valid <= id_valid;
            inst_addr_exe <= inst_addr_id;
            inst_data_exe <= inst_data_id;
            inst_addr_next_exe <= inst_addr_next_id;
            regw_addr_exe <= regw_addr_id;
            pc_src_exe <= pc_src_ctrl;
            exe_a_src_exe <= exe_a_src_ctrl;
            exe_b_src_exe <= exe_b_src_ctrl;
            data_rs_exe <= data_rs;
            data_rt_exe <= data_rt;
            data_imm_exe <= data_imm;
            exe_alu_oper_exe <= exe_alu_oper_ctrl;
            mem_ren_exe <= mem_ren_ctrl;
            mem_wen_exe <= mem_wen_ctrl;
            wb_data_src_exe <= wb_data_src_ctrl;
            wb_wen_exe <= wb_wen_ctrl;
        end
    end

    always @(*) begin
        is_branch_exe <= (pc_src_exe != PC_NEXT);
    end

    assign
        rs_rt_equal_exe = (data_rs_exe == data_rt_exe);

    always @(*) begin
        opa_exe = data_rs_exe;
        opb_exe = data_rt_exe;
        case (exe_a_src_exe)
            EXE_A_RS: opa_exe = data_rs_exe;
            EXE_A_LINK: opa_exe = inst_addr_next_exe;
            EXE_A_BRANCH: opa_exe = inst_addr_next_exe;
        endcase
        case (exe_b_src_exe)
            EXE_B_RT: opb_exe = data_rt_exe;
            EXE_B_IMM: opb_exe = data_imm_exe;
            EXE_B_LINK: opb_exe = 32'h0;  // linked address is the next one
of current instruction
            EXE_B_BRANCH: opb_exe = {
                data_imm_exe[29:0], 2'b0
```

```
                };
        endcase
    end

    alu ALU (
        .a(opa_exe),
        .b(opb_exe),
        .oper(exe_alu_oper_exe),
        .result(alu_out_exe)
        );

    // MEM stage
    always @(posedge clk) begin
        if (mem_rst) begin
            mem_valid <= 0;
            pc_src_mem <= 0;
            inst_addr_mem <= 0;
            inst_data_mem <= 0;
            inst_addr_next_mem <= 0;
            regw_addr_mem <= 0;
            data_rs_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;
            rs_rt_equal_mem <= 0;
        end
        else if (mem_en) begin
            mem_valid <= exe_valid;
            pc_src_mem <= pc_src_exe;
            inst_addr_mem <= inst_addr_exe;
            inst_data_mem <= inst_data_exe;
            inst_addr_next_mem <= inst_addr_next_exe;
            regw_addr_mem <= regw_addr_exe;
            data_rs_mem <= data_rs_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;
            rs_rt_equal_mem <= rs_rt_equal_exe;
        end
    end

    always @(*) begin
        is_branch_mem <= (pc_src_mem != PC_NEXT);
    end

    always @(*) begin
        case (pc_src_mem)
            PC_JUMP: branch_target_mem <= {
                inst_addr_mem[31:28],
                inst_data_mem[25:0],
                2'b0
                };
```

```
333             PC_JR: branch_target_mem <= data_rs_mem;
334             PC_BEQ: branch_target_mem <= rs_rt_equal_mem ?
335                     alu_out_mem :
336                     inst_addr_next_mem ;
337             PC_BNE: branch_target_mem <= rs_rt_equal_mem ?
338                     inst_addr_next_mem :
339                     alu_out_mem ;
340             default: branch_target_mem <= inst_addr_next_mem;  // will
    never used
341         endcase
342     end
343 
344     assign
345         mem_ren = mem_ren_mem,
346         mem_wen = mem_wen_mem,
347         mem_addr = alu_out_mem,
348         mem_dout = data_rt_mem;
349 
350     // WB stage
351     always @(posedge clk) begin
352         if (wb_rst) begin
353             wb_valid <= 0;
354             wb_wen_wb <= 0;
355             wb_data_src_wb <= 0;
356             regw_addr_wb <= 0;
357             alu_out_wb <= 0;
358             mem_din_wb <= 0;
359         end
360         else if (wb_en) begin
361             wb_valid <= mem_valid;
362             wb_wen_wb <= wb_wen_mem;
363             wb_data_src_wb <= wb_data_src_mem;
364             regw_addr_wb <= regw_addr_mem;
365             alu_out_wb <= alu_out_mem;
366             mem_din_wb <= mem_din;
367         end
368     end
369 
370     always @(*) begin
371         regw_data_wb = alu_out_wb;
372         case (wb_data_src_wb)
373             WB_DATA_ALU: regw_data_wb = alu_out_wb;
374             WB_DATA_MEM: regw_data_wb = mem_din_wb;
375         endcase
376     end
377 
378 endmodule
```