

# Computer Architecture

## Lab 3 Report

<b>Name:</b>	Asudy Wang 王浚哲	<b>ID:</b>	3180103011	<b>Major:</b>	Computer Science & Technology
<b>Course:</b>	Computer Architecture		<b>Place:</b>	Room 301, Cao Guangbiao Building West Wing, Yuquan Campus	
<b>Due Date:</b>	2020-12-21	<b>Groupmate:</b>	Flaze He	<b>Instructor:</b>	Kai Bu

## Table of Contents

### Table of Contents

#### Lab 2. 5-Stage Pipelined CPU with Stall

- §1 Purposes & Requirements
  - 1.1 Experiment Purpose
  - 1.2 Experiment Tasks
- §2 Contents & Principles
  - 2.1 Datapath
  - 2.2 Controller
  - 2.3 Data Hazard
- §3 Main Instruments & Materials
  - 3.1 Experiment Instruments
  - 3.2 Experiment Materials
- §4 Experiment Procedure & Operations
  - 4.1 Add Stall Detections & Controls to `controller.v`
  - 4.2 Verify the Pipelined CPU Design
- §5 Results & Analysis
  - 5.1 Function Verification
- §6 Discussion & Experience
- Appendix A. *controller.v*

## Lab 2. 5-Stage Pipelined CPU with Stall

### §1 Purposes & Requirements

## 1.1 Experiment Purpose

- Understand the **principles** of Pipelined CPU Stall;
- Understand the **principles** of Data Hazard;
- Understand the method of Pipelined CPU Stall Detection & Stall the Pipeline;
- Master methods of program verification of Pipelined CPU *with Stall*.

## 1.2 Experiment Tasks

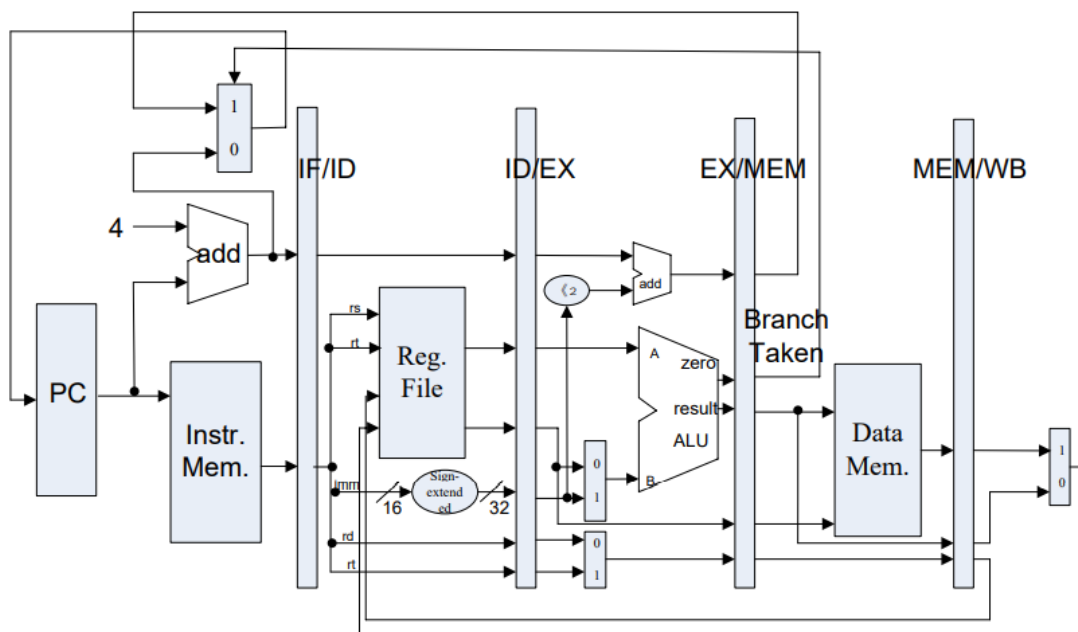
- Design the *Stall Part* of Datapath of the 5-stage pipelined CPU implemented in Lab 2.
- Modify the CPU controller by adding *Stall Condition Detection*.
- Verify the *Pipelined CPU* with program and observe the execution of the program.

## §2 Contents & Principles

### 2.1 Datapath

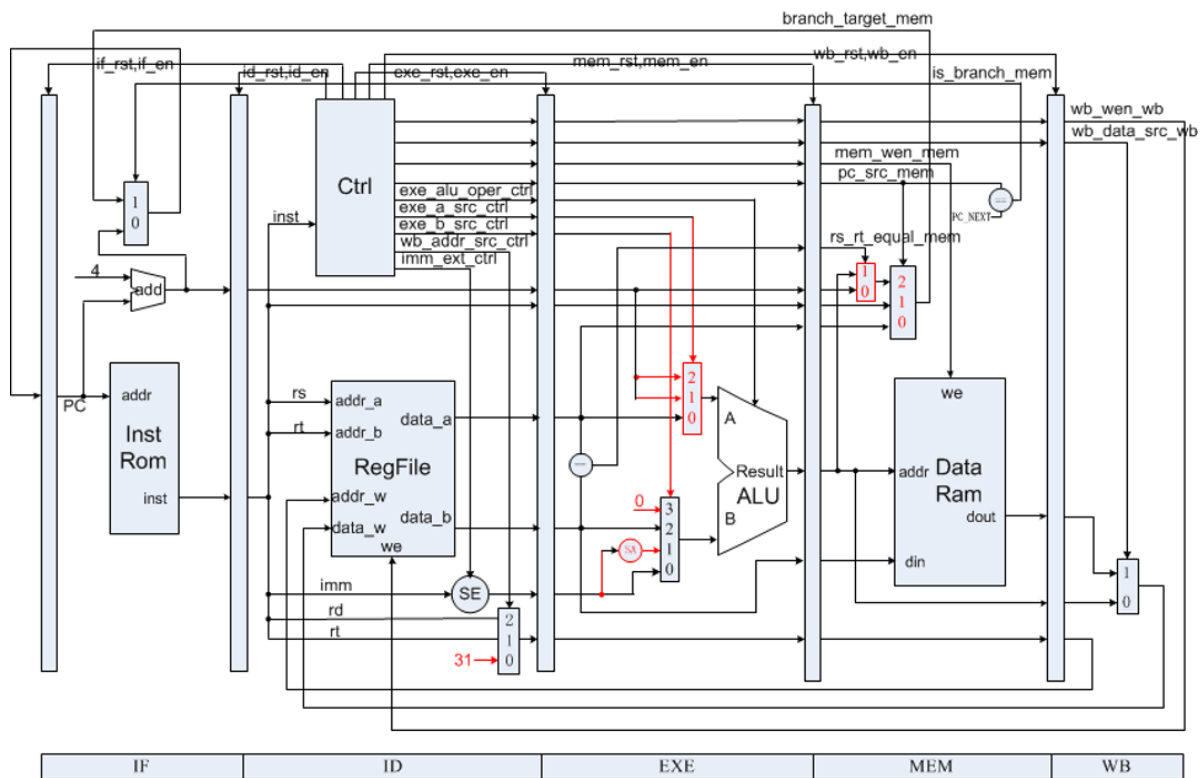
The *datapath* in the *Stall Version* of the 5-staged pipelined CPU is much like the same as the one we've implemented in the previous lab course, since the main difference for detecting & controlling the stalls lies in the *controller*.

A simplified version of the pipelined datapath we implemented in Lab 2 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:



When an instruction needs to be *stalled*, it is "stopped" at the current stage and "waiting" for the required data. This important operation is done by controller's disabling/clearing a certain stage.

## 2.3 Data Hazard

Data Hazards arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.

When  $inst_i$  executes before  $inst_j$ , there're data hazards:

- **RAW** (Read After Write), i.e.  $inst_j$  reads a source (operand) *before* the  $inst_i$  writes it back.
- **WAW** (Write After Write), i.e.  $inst_j$  writes an operand *before* the  $inst_i$  writes it back.
- **WAR** (Write After Read), i.e.  $inst_j$  writes an operand *before* the  $inst_i$  reads it.

## §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*, 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 Add Stall Detections & Controls to `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 **pipeline control 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.

The stall control is divided into 2 parts: *register stall* and *branch stall*, of which the meanings and implementations will be shown below.

### 1. Register Stall

Register Stalls happen when an instruction is requiring to read an operand at the **ID Stage** which is not yet written back by the precedent instruction. When this happens, we need to *disable IF and ID stage* and *reset the EXE stage*.

```
1  // Register Stall Detection
2  always @(*) begin
3      reg_stall = 0;
4      if (rs_used && addr_rs != 0) begin
5          if (regw_addr_exe == addr_rs && wb_wen_exe) begin
6              reg_stall = 1;
7          end
8          else if (regw_addr_mem == addr_rs && wb_wen_mem) begin
9              reg_stall = 1;
10         end
11     end
12     if (rt_used && addr_rt != 0) begin
13         if (regw_addr_exe == addr_rt && wb_wen_exe) begin
14             reg_stall = 1;
15         end
16         else if (regw_addr_mem == addr_rt && wb_wen_mem) begin
17             reg_stall = 1;
18         end
19     end
20 end
```

### 2. Branch Stall

In our 5-stage pipeline datapath, the *branch destination* can be obtained only at the end of the EXE stage, and we decided to flush the pipeline once there's a branch instruction. So once we detect a branch instruction at the ID stage, 3 NOPs should be inserted, i.e. the ID stage should be reset for 3 clock cycles.

```
1  always @(*) begin
2      branch_stall = 0;
3      if (pc_src != PC_NEXT || is_branch_exe || is_branch_mem)
4          branch_stall = 1;
5  end
```

### 3. Pipeline Stall Control

This part controls each stage according to the `reg_stall` and `branch_stall` signals as well as the `DEBUG` indicator.

```

1  always @(*) begin
2      if_rst = 0;
3      if_en = 1;
4      id_rst = 0;
5      id_en = 1;
6      exe_rst = 0;
7      exe_en = 1;
8      mem_rst = 0;
9      mem_en = 1;
10     wb_rst = 0;
11     wb_en = 1;
12     if (rst) begin
13         if_rst = 1;
14         id_rst = 1;
15         exe_rst = 1;
16         mem_rst = 1;
17         wb_rst = 1;
18     end
19     `ifdef DEBUG
20         // suspend and step execution
21         else if ((debug_en) && ~(~debug_step_prev && debug_step)) begin
22             if_en = 0;
23             id_en = 0;
24             exe_en = 0;
25             mem_en = 0;
26             wb_en = 0;
27         end
28     `endif
29     // this stall indicate that ID is waiting for previous
    instruction, should insert NOPs between ID and EXE.
30     else if (reg_stall) begin
31         if_en = 0;
32         id_en = 0;
33         exe_rst = 1;
34     end
35     // this stall indicate that a jump/branch instruction is
    running, so that 3 NOP should be inserted between IF and ID
36     else if (branch_stall) begin
37         id_rst = 1;
38     end
39 end

```

## 4.2 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 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:

code >  inst_mem.hex	code > <b>ASM</b> inst_mem.asm
1 00000824	1 and \$at \$zero \$zero
2 34240050	2 ori \$a0 \$at 80
3 20050004	3 addi \$a1 \$zero 4
4 0C00000A	4 jal 10
5 AC820000	5 sw \$v0 0(\$a0)
6 8C890000	6 lw \$t1 0(\$a0)
7 AC890004	7 sw \$t1 4(\$a0)
8 01244022	8 sub \$t0 \$t1 \$a0
9 08000008	9 j 8
10 00000000	10 sll \$zero \$zero 0
11 00004020	11 add \$t0 \$zero \$zero
12 8C890000	12 lw \$t1 0(\$a0)
13 01094020	13 add \$t0 \$t0 \$t1
14 20A5FFFF	14 addi \$a1 \$a1 -1
15 20840004	15 addi \$a0 \$a0 4
16 0005182A	16 slt \$v1 \$zero \$a1
17 1460FFFA	17 bne \$v1 \$zero -6
18 01001025	18 or \$v0 \$t0 \$zero
19 03E00008	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 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 zip file* for the photos of the full execution progress of the above program.

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, I read carefully through the [code provided by SOWRD](#) and understood how *stalls* in a pipelined CPU is detected and implemented. Many auxiliary wires and variables are added to store and pass some states or data of the CPU. The introduction of `en` and `rst` signals played a very important role in controlling the datapath and further, implementing stall controls and debug controls.

## 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;

```

```

104         wb_addr_src = WB_ADDR_RD;
105         wb_data_src = WB_DATA_ALU;
106         wb_wen = 1;
107         rs_used = 1;
108         rt_used = 1;
109     end
110     R_FUNC_SLT: begin
111         exe_alu_oper = EXE_ALU_SLT;
112         wb_addr_src = WB_ADDR_RD;
113         wb_data_src = WB_DATA_ALU;
114         wb_wen = 1;
115         rs_used = 1;
116         rt_used = 1;
117     end
118     default: begin
119         unrecognized = 1;
120     end
121 endcase
122 end
123 INST_J: begin
124     pc_src = PC_JUMP;
125 end
126 INST_JAL: begin
127     pc_src = PC_JUMP;
128     exe_a_src = EXE_A_LINK;
129     exe_b_src = EXE_B_LINK;
130     exe_alu_oper = EXE_ALU_ADD;
131     wb_addr_src = WB_ADDR_LINK;
132     wb_data_src = WB_DATA_ALU;
133     wb_wen = 1;
134 end
135 INST_BEQ: begin
136     pc_src = PC_BEQ;
137     exe_a_src = EXE_A_BRANCH;
138     exe_b_src = EXE_B_BRANCH;
139     exe_alu_oper = EXE_ALU_ADD;
140     imm_ext = 1;
141     rs_used = 1;
142     rt_used = 1;
143 end
144 INST_BNE: begin
145     pc_src = PC_BNE;
146     exe_a_src = EXE_A_BRANCH;
147     exe_b_src = EXE_B_BRANCH;
148     exe_alu_oper = EXE_ALU_ADD;
149     imm_ext = 1;
150     rs_used = 1;
151     rt_used = 1;
152 end
153 INST_ADDI: begin
154     imm_ext = 1;
155     exe_b_src = EXE_B_IMM;
156     exe_alu_oper = EXE_ALU_ADD;
157     wb_addr_src = WB_ADDR_RT;
158     wb_data_src = WB_DATA_ALU;
159     wb_wen = 1;
160     rs_used = 1;
161 end

```

```

162         INST_ANDI: begin
163             imm_ext = 0;
164             exe_b_src = EXE_B_IMM;
165             exe_alu_oper = EXE_ALU_AND;
166             wb_addr_src = WB_ADDR_RT;
167             wb_data_src = WB_DATA_ALU;
168             wb_wen = 1;
169             rs_used = 1;
170         end
171         INST_ORI: begin
172             imm_ext = 0;
173             exe_b_src = EXE_B_IMM;
174             exe_alu_oper = EXE_ALU_OR;
175             wb_addr_src = WB_ADDR_RT;
176             wb_data_src = WB_DATA_ALU;
177             wb_wen = 1;
178             rs_used = 1;
179         end
180         INST_LW: begin
181             imm_ext = 1;
182             exe_b_src = EXE_B_IMM;
183             exe_alu_oper = EXE_ALU_ADD;
184             mem_ren = 1;
185             wb_addr_src = WB_ADDR_RT;
186             wb_data_src = WB_DATA_MEM;
187             wb_wen = 1;
188             rs_used = 1;
189         end
190         INST_SW: begin
191             imm_ext = 1;
192             exe_b_src = EXE_B_IMM;
193             exe_alu_oper = EXE_ALU_ADD;
194             mem_wen = 1;
195             rs_used = 1;
196             rt_used = 1;
197         end
198         default: begin
199             unrecognized = 1;
200         end
201     endcase
202 end
203
204 /* pipeline control (stall) */
205 reg reg_stall;
206 reg branch_stall;
207 wire [4:0] addr_rs, addr_rt;
208
209 assign
210     addr_rs = inst[25:21],
211     addr_rt = inst[20:16];
212
213 always @(*) begin
214     reg_stall = 0;
215     if (rs_used && addr_rs != 0) begin
216         if (regw_addr_exe == addr_rs && wb_wen_exe) begin
217             reg_stall = 1;
218         end
219         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
```