

2017/5/26

---

---

# MIPS-CPU 流水线设计

计算机组成原理 Lab 07

---

---

PB15121735

王若冰

# MIPS CPU 流水线设计

计算机组成原理 Lab 07

## 实验目的

---

1. 实现多周期 MIPS CPU，包括要求的 16 条指令。
2. 在 16 条的基础上，增加了 16 条指令。
3. 实现了溢出中断和指令中断。
4. 在流水线方面，实现了基础暂停，也实现了两类前推。
5. 实现了下载，并可以用开关输入地址。
6. 实现了分支预测。

## 实验工具

---

1. ISE 开发平台。
2. ISE 集成在内部的模拟插件。
3. NEXYS 3 FPGA.

## 实验内容

---

### 实验具体步骤

1. 分析 CPU 所需要实现的逻辑，设计项目所需的模块。
2. 依据需要实现的逻辑编写代码实现 CPU。

3. 对于对应的内存进行例化。
4. 编写对应的仿真文件。
5. 验证仿真结果，如果不对，则应该修改检测修改代码。
6. 配置管脚文件。
7. 下载电路板上查看结果。

## 实验结果

---

### 设计结果

这次实验和之前的实验不一样，我基于所需要实现的要求，修改了大体实现的思路。之前的实验由于需要实现的功能比较简单，而且实验 PPT 上面具体的给出了实验的逻辑图，所以我所需要做的是按照逻辑图的功能去完成实验。我姑且把这种方式叫为**面向图的实现**。但是这次实验不一样，第一没有现有的逻辑图可以完美的涵盖所有需要实现的细节，第二实验所需要实现的逻辑也比较复杂，如果从逻辑图角度先去设计，然后在去实现，就会比较繁琐。所以这次实验，我没有依赖于现有的逻辑图，也没有去刻意的设计逻辑图，而是直接根据自己需要实现的逻辑去编写实验。我姑且把这种方式叫做**面向逻辑的实现**。在这种实现中，我将每一个阶段，都封装成了一个一个的模块。如，pc module, if\_id module, id module, 等等。现在详细叙述各个模块的功能。和一些需要注意的细节。（模块介绍中省略了与 if\_id 较为重复的 id\_ex , ex\_mem, mem\_wb 等逻辑简单的中间过渡模块）

#### CPU\_top 设计结果

CPU\_top 模块就是总体的设计连接，并没有多余的设计，只是运用 wire 信号连接各个模块。所以不再赘述。

#### Output\_div 设计结果

在该模块和 CPU 无关，主要负责输出的分频，为了下载在电路班上的显示信号而分频。注意的是所需要显示的是 32 位数据，我们将用 16 进制显示结果，但是电路板上的硬件只能同时显示 4 位，所以我们需要分页。每一秒改变一页去交叉显示结果。

### Output\_code 模块设计结果

该模块也是与 CPU 无关，负责输出译码，所以不再赘述。

### Pc 模块的编写

正式的 CPU 设计从这个模块开始。pc 负责控制 pc。这个模块值得一提的是 stall 信号，branch 信号。Stall 信号是延迟信号，由于在 lw 指令之后紧跟的 jr 指令，add 指令等会出现数据相关。而这种数据相关不是数据前推可以解决的。必须使用 stall 信号将流水线推迟等待。而 branch 指令负责控制跳转指令 pc 的变换。

```
1. module pc(  
2.  
3.   input                clk,  
4.   input                rst,  
5.   input  [5:0]         stall,  
6.   input                branch,  
7.   input  [31:0]        branch_addr,  
8.   output reg [31:0]    pc = 32'd0  
9.  
10. );
```

### InstRom 模块

该模块只用 Distributed Simple Port Rom，用于存储指令。使用的是异步读，不在赘述。

### If\_id 模块

该模块为 if 和 id 的中间模块，需要注意的是，该模块含有一个 last\_branch 信号。这个信号用于处理分支预测。因为，该设计为静态分支预测，预测 branch 不会被取得，那么如果取得了，需要将下一阶段的指令清空。

```
1. module if_id(  
2. input clk,  
3. input rst,  
4. input [5:0] stall,  
5. input [31:0] if_pc,  
6. input [31:0] if_inst,  
7. input branch,  
8. output reg [31:0] id_pc,  
9. output reg [31:0] id_inst,  
10. output reg last_branch  
11. );
```

#### Id 模块

该模块为译码模块。几乎是代码量最大的模块，本设计在原来 16 个指令的基础上，增加了 17 条指令，也就是需要译码 33 条指令。所以，给模块的逻辑比较复杂。而且在本模块中使用了两个前推和一个暂停设计。两个前推分别是 ex 阶段的信号和 mem 阶段信号的前推，而暂停则是在 load 指令之后如果在 id 阶段就需要使用这个 load 后的寄存器，如 load 之后的 add 或者 branch，则暂停设计必须要实现。

```
1. module id(  
2.  
3. input rst,  
4. input [31:0] pc_i,  
5. input [31:0] inst,  
6.  
7. input ex_wreg_i,  
8. input [31:0] ex_wdata_i,  
9. input [31:0] ex_wd_i,  
10.  
11. input mem_wreg_i,  
12. input [31:0] mem_wdata_i,
```

```

13.    input [31:0]                mem_wd_i,
14.
15.    input [31:0]                reg1_data_i,
16.    input [31:0]                reg2_data_i,
17.
18.    input                        last_branch,
19.    input [7:0]                  ex_aluop_i,
20.
21.    output reg                    reg1_read_o,
22.    output reg                    reg2_read_o,
23.    output reg[31:0]              reg1_addr_o,
24.    output reg[31:0]              reg2_addr_o,
25.
26.    output reg[7:0]                aluop_o,
27.    output reg[2:0]                alusel_o,
28.    output reg[31:0]              reg1_o,
29.    output reg[31:0]              reg2_o,
30.    output reg[31:0]              wd_o,
31.    output reg                    wreg_o,
32.    output reg                    branch,
33.    output reg [31:0]             branch_addr,
34.    output [31:0]                 this_inst,
35.    output                        stall_id
36. );

```

### Ex 模块

该模块是执行模块，本设计将 ALU 和其辅助线路全部抽象为逻辑包装在一个模块中，执行阶段并不难实现，不在赘述。

```

1. module ex(
2.
3.    input                rst,
4.
5.    input [7:0]          aluop_i,
6.    input [2:0]          alusel_i,
7.    input [31:0]         reg1_i,
8.    input [31:0]         reg2_i,
9.    input [31:0]         wd_i,
10.    input                wreg_i,

```

```

11.    input  [31:0]          ex_inst,
12.
13.
14.    output reg[31:0]        wd_o,
15.    output reg              wreg_o,
16.    output reg[31:0]        wdata_o,
17.    output  [7:0]           aluop_o,
18.    output  [31:0]          mem_addr_o,
19.    output  [31:0]          reg2_o
20. );

```

### mem 模块

给模块负责仿存，也负责 R 型指令的继续后移。当需要仿存的时候，alu\_op 会被译码然后判断是否要读还是要写内存。然后 mem 模块就会将相应的信号给 RAM 进行相应的操作。

```

1. module mem(
2.
3.    input                rst,
4.
5.    input  [31:0]        wd_i,
6.    input                wreg_i,
7.    input  [31:0]        wdata_i,
8.    input  [7:0]         alu_op,
9.    input  [31:0]        mem_data_i,
10.   input  [31:0]        mem_addr_i,
11.   input  [31:0]        mem_reg2,
12.   output reg [31:0]    wd_o,
13.   output reg          wreg_o,
14.   output reg [31:0]    wdata_o,
15.   output reg [31:0]    mem_addr_o,
16.   output reg          mem_we,
17.   output reg [31:0]    mem_data_o
18. );

```

### RAM 模块

RAM 模块为数据内存，采用了 32 位，256 深度的 RAM，内存大小较为合理，可以满足该实验的需求。

```
1. RAM your_instance_name (
2.   .a(a), // input [7 : 0] a
3.   .d(d), // input [31 : 0] d
4.   .dpra(dpra), // input [7 : 0] dpra
5.   .clk(clk), // input clk
6.   .we(we), // input we
7.   .spo(spo), // output [31 : 0] spo
8.   .dpo(dpo) // output [31 : 0] dpo
9.);
```

## 实验结果

### Simulation 结果

#### MARS 汇编结果

如左图，在原来的斐波那契数列的汇编程序的基础上，给出了自己加的指令测试在后面，测试的结果将反应在寄存器文件中。

所以只需对比寄存器文件中的值和 MARS 执行结果中寄存器文件值一样即可。MARS 执行的 RAM 结果和寄存器结构如下。

Basic
addi \$8, \$0, 0x00002000
addi \$13, \$0, 0x00002050
lw \$13, 0x00000000(\$13)
addi \$11, \$0, 0x00002054
lw \$11, 0x00000000(\$11)
addi \$12, \$0, 0x00002054
lw \$12, 0x00000004(\$12)
sw \$11, 0x00000000(\$8)
sw \$12, 0x00000004(\$8)
addi \$9, \$13, 0xfffffffffe
lw \$11, 0x00000000(\$8)
lw \$12, 0x00000004(\$8)
add \$10, \$11, \$12
sw \$10, 0x00000008(\$8)
addi \$8, \$8, 0x00000004
addi \$9, \$9, 0xffffffffff
bgtz \$9, 0xfffffffff9
lui \$1, 0x00000101
ori \$1, \$1, 0x00000101
ori \$2, \$1, 0x00000100
or \$1, \$1, \$2
andi \$3, \$1, 0x000000fe
and \$1, \$3, \$1
xori \$4, \$1, 0x0000ff00
xor \$1, \$4, \$1
nor \$1, \$4, \$1
sll \$2, \$2, 0x00000008
sllv \$2, \$2, \$7
srl \$2, \$2, 0x00000008
srlv \$2, \$2, \$5
sw \$2, 0x0000000c(\$8)
sw \$1, 0x00000010(\$8)
j 0x00000080

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00002000	0x00000003	0x00000003	0x00000006	0x00000009	0x0000000f	0x00000018	0x00000027	0x0000003f
0x00002020	0x00000066	0x000000a5	0x0000010b	0x00000150	0x000002bb	0x0000046b	0x00000726	0x00000b91
0x00002040	0x000012b7	0x00001e48	0x000030ff	0x00004f47	0x00000014	0x00011101	0xffffffff	0x00000000
0x00002060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00002080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000020a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Contents of RAM by MARS



Name	Number	Value
\$zero	0	0x00000000
\$at	1	0xffff00ff
\$v0	2	0x00011101
\$v1	3	0x00000000
\$a0	4	0x0000ff00
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00002048
\$t1	9	0x00000000
\$t2	10	0x00004f47
\$t3	11	0x00001e48
\$t4	12	0x000030ff
\$t5	13	0x00000014
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000

Contents of Registers by MARS

实际仿真结果

如下就是用自己实现的 CPU 运行上述指令的仿真结果。观察结果，与上面结果完全一致，则 CPU 逻辑正确。

	0	1	2	3
0	00000003	00000003	00000006	00000009
4	0000000F	00000018	00000027	0000003F
8	00000066	000000A5	0000010B	000001B0
12	000002BB	0000046B	00000726	00000B91
16	000012B7	00001E48	000030FF	00004F47
20	00000014	00011101	FFFF00FF	00000000
24	00000000	00000000	00000000	00000000

Contents of RAM by CPU

	0	1	2	3
0	00000000	FFFF00FF	00011101	00000000
4	0000FF00	00000000	00000000	00000000
8	00000048	00000000	00004F47	00001E48
12	000030FF	00000014	00000000	00000000
16	00000000	00000000	00000000	00000000
20	00000000	00000000	00000000	00000000
24	00000000	00000000	00000000	00000000
28	00000000	00000000	00000000	00000000

*Contents of Registers by CPU*

## 下载电路板结果

配置好管脚文件后，在实验检查现场给助教检查结果无误。使用的是开关查找内存，内存使用的是双端口内存，这样一个端口用于 CPU，一个端口用于开关查找结果，就非常方便。下载结果也正确。

## 附录（代码量大，只给出核心代码）

```

22 `include "Define.v"
23
24
25 module id(
26
27     input                rst,
28     input [31:0]         pc_i,
29     input [31:0]         inst,
30
31     input                ex_wreg_i,
32     input [31:0]         ex_wdata_i,
33     input [31:0]         ex_wd_i,
34
35     input                mem_wreg_i,
36     input [31:0]         mem_wdata_i,
37     input [31:0]         mem_wd_i,
38
39     input [31:0]         reg1_data_i,
40     input [31:0]         reg2_data_i,
41
42     input                last_branch,
43     input [7:0]          ex_aluop_i,
44
45     output reg           reg1_read_o,
46     output reg           reg2_read_o,
47     output reg[31:0]     reg1_addr_o,
48     output reg[31:0]     reg2_addr_o,
49
50     output reg[7:0]      aluop_o,
51     output reg[2:0]      alusel_o,
52     output reg[31:0]     reg1_o,
53     output reg[31:0]     reg2_o,
54     output reg[31:0]     wd_o,
55     output reg           wreg_o,
56     output reg           branch,
57     output reg [31:0]    branch_addr,
58     output [31:0]        this_inst,
59     output               stall_id
60 );
61
62 reg stall_reg1;
63 reg stall_reg2;
64 wire [31:0] inst_i = (last_branch)? 32'd0: inst;
65 wire [5:0] op = inst_i[31:26];
66 wire [4:0] op2 = inst_i[10:6];
67 wire [5:0] op3 = inst_i[5:0];
68 wire [4:0] op4 = inst_i[20:16];

```

*Id.v (part 1)*

```

69 reg[31:0] imm;
70 reg instvalid;
71 wire [31:0] immssl = {{14{inst_i[15]}},inst_i[15:0],2'b00};
72 wire [31:0] pc_4 = pc_i + 32'd4;
73 assign this_inst = inst;
74
75 always @ (*) begin
76   if (~rst) begin
77     aluop_o <= 8'd0;
78     alusel_o <= 3'd0;
79     wd_o <= 32'd0;
80     wreg_o <= 1'd0;
81     instvalid <= 1'd0;
82     reg1_read_o <= 1'b0;
83     reg2_read_o <= 1'b0;
84     reg1_addr_o <= 32'd0;
85     reg2_addr_o <= 32'd0;
86     imm <= 32'h0;
87     branch_addr <= 32'd0;
88     branch <= 1'd0;
89   end else begin
90     aluop_o <= 8'd0;
91     alusel_o <= 3'd0;
92     wd_o <= inst_i[15:11];
93     wreg_o <= 1'd0;
94     instvalid <= 1'b1;
95     reg1_read_o <= 1'b0;
96     reg2_read_o <= 1'b0;
97     reg1_addr_o <= inst_i[25:21];
98     reg2_addr_o <= inst_i[20:16];
99     imm <= 32'd0;
100    branch_addr <= 32'd0;
101    branch <= 1'd0;
102    case (op)
103      `SPECIAL_INST: begin
104        case (op2)
105          5'b000000: begin
106            case (op3)
107              `OR: begin
108                wreg_o <= 1'd1; aluop_o <= `OR_OP;
109                alusel_o <= `RES_LOGIC; reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
110                instvalid <= 1'd0;
111              end
112              `AND: begin
113                wreg_o <= 1'd1; aluop_o <= `AND_OP;
114                alusel_o <= `RES_LOGIC; reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
115                instvalid <= 1'd0;

```

*Id.v (part 2)*

```

116         end
117     `XOR: begin
118         wreg_o <= 1'd1;         aluop_o <= `XOR_OP;
119         alusel_o <= `RES_LOGIC;   reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
120         instvalid <= 1'd0;
121     end
122     `NOR: begin
123         wreg_o <= 1'd1;         aluop_o <= `NOR_OP;
124         alusel_o <= `RES_LOGIC;   reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
125         instvalid <= 1'd0;
126     end
127     `SLLV: begin
128         wreg_o <= 1'd1;         aluop_o <= `SLL_OP;
129         alusel_o <= `RES_SHIFT;   reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
130         instvalid <= 1'd0;
131     end
132     `SRLV: begin
133         wreg_o <= 1'd1;         aluop_o <= `SRL_OP;
134         alusel_o <= `RES_SHIFT;   reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
135         instvalid <= 1'd0;
136     end
137     `SRAV: begin
138         wreg_o <= 1'd1;         aluop_o <= `SRA_OP;
139         alusel_o <= `RES_SHIFT;   reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
140         instvalid <= 1'd0;
141     end
142     `SLT: begin
143         wreg_o <= 1'd1;         aluop_o <= `SLT_OP;
144         alusel_o <= `RES_ARITHMETIC;   reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
145         instvalid <= 1'd0;
146     end
147     `SLTU: begin
148         wreg_o <= 1'd1;         aluop_o <= `SLTU_OP;
149         alusel_o <= `RES_ARITHMETIC;   reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
150         instvalid <= 1'd0;
151     end
152     `ADD:
153     begin
154         wreg_o <= 1'd1;
155         aluop_o <= `ADD_OP;
156         alusel_o <= `RES_ARITHMETIC;
157         reg1_read_o <= 1'd1;
158         reg2_read_o <= 1'd1;
159         instvalid <= 1'd0;
160     end
161     `ADDU: begin
162         wreg_o <= 1'd1;         aluop_o <= `ADDU_OP;

```

*Id.v (part 3)*

```

163         alusel_o <= `RES_ARITHMETIC;      reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
164         instvalid <= 1'd0;
165     end
166 `SLT: begin
167     wreg_o <= 1'd1;      aluop_o <= `SLT_OP;
168     alusel_o <= `RES_ARITHMETIC;      reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
169     instvalid <= 1'd0;
170 end
171 `SLTU: begin
172     wreg_o <= 1'd1;      aluop_o <= `SLTU_OP;
173     alusel_o <= `RES_ARITHMETIC;      reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
174     instvalid <= 1'd0;
175 end
176 `SUB: begin
177     wreg_o <= 1'd1;      aluop_o <= `SUB_OP;
178     alusel_o <= `RES_ARITHMETIC;      reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
179     instvalid <= 1'd0;
180 end
181 `SUBU: begin
182     wreg_o <= 1'd1;      aluop_o <= `SUBU_OP;
183     alusel_o <= `RES_ARITHMETIC;      reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
184     instvalid <= 1'd0;
185 end
186 `JR: begin
187     wreg_o <= 1'd0;
188     aluop_o <= `JR_OP;
189     alusel_o <= `RES_JUMP_BRANCH;
190     reg1_read_o <= 1'b1;
191     reg2_read_o <= 1'b0;
192     branch_addr <= reg1_o;
193     branch <= 1'd1;
194     instvalid <= 1'd0;
195 end
196     default: begin
197     end
198 endcase
199 end
200     default: begin
201     end
202 endcase
203 end
204 `ORI: begin
205     wreg_o <= 1'd1;      aluop_o <= `OR_OP;
206     alusel_o <= `RES_LOGIC; reg1_read_o <= 1'b1; reg2_read_o <= 1'b0;
207     imm <= {16'h0, inst_i[15:0]}; wd_o <= inst_i[20:16];
208     instvalid <= 1'd0;
209 end

```

*Id.v(part 4)*

```

210 `ANDI:      begin
211     wreg_o <= 1'd1;      aluop_o <= `AND_OP;
212     alusel_o <= `RES_LOGIC; reg1_read_o <= 1'b1; reg2_read_o <= 1'b0;
213     imm <= {16'h0, inst_i[15:0]}; wd_o <= inst_i[20:16];
214     instvalid <= 1'd0;
215     end
216 `XORI:      begin
217     wreg_o <= 1'd1;      aluop_o <= `XOR_OP;
218     alusel_o <= `RES_LOGIC; reg1_read_o <= 1'b1; reg2_read_o <= 1'b0;
219     imm <= {16'h0, inst_i[15:0]}; wd_o <= inst_i[20:16];
220     instvalid <= 1'd0;
221     end
222 `LUI:       begin
223     wreg_o <= 1'd1;      aluop_o <= `OR_OP;
224     alusel_o <= `RES_LOGIC; reg1_read_o <= 1'b1; reg2_read_o <= 1'b0;
225     imm <= {inst_i[15:0], 16'h0}; wd_o <= inst_i[20:16];
226     instvalid <= 1'd0;
227     instvalid <= 1'd0;
228     end
229 `ADDI:      begin
230     wreg_o <= 1'd1;      aluop_o <= `ADDI_OP;
231     alusel_o <= `RES_ARITHMETIC; reg1_read_o <= 1'b1; reg2_read_o <= 1'b0;
232     imm <= {{16{inst_i[15]}}, inst_i[15:0]}; wd_o <= inst_i[20:16];
233     instvalid <= 1'd0;
234     end
235 `ADDIU:     begin
236     wreg_o <= 1'd1;      aluop_o <= `ADDIU_OP;
237     alusel_o <= `RES_ARITHMETIC; reg1_read_o <= 1'b1; reg2_read_o <= 1'b0;
238     imm <= {{16{inst_i[15]}}, inst_i[15:0]}; wd_o <= inst_i[20:16];
239     instvalid <= 1'd0;
240     end
241 `J:         begin
242     wreg_o <= 1'd0;
243     aluop_o <= `J_OP;
244     alusel_o <= `RES_JUMP_BRANCH; reg1_read_o <= 1'b0; reg2_read_o <= 1'b0;
245     branch_addr <= {pc_4[31:28], inst_i[25:0], 2'b00};
246     branch <= 1'd1;
247     instvalid <= 1'd0;
248     end
249 `BEQ:       begin
250     wreg_o <= 1'd0;      aluop_o <= `BEQ_OP;
251     alusel_o <= `RES_JUMP_BRANCH; reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
252     instvalid <= 1'd0;
253     if(reg1_o == reg2_o) begin
254         branch_addr <= pc_4 + imm$11;
255         branch <= 1'd1;
256     end

```

*Id.v (part 5)*

```

257     end
258 `BGTZ:    begin
259     wreg_o <= 1'd0;      aluop_o <= `BGTZ_OP;
260     alusel_o <= `RES_JUMP_BRANCH; reg1_read_o <= 1'b1; reg2_read_o <= 1'b0;
261     instvalid <= 1'd0;
262     if((reg1_o[31] == 1'b0) && (reg1_o != 32'd0)) begin
263         branch_addr <= pc_4 + imm11;
264         branch <= 1'd1;
265     end
266 end
267 `BLEZ:    begin
268     wreg_o <= 1'd0;      aluop_o <= `BLEZ_OP;
269     alusel_o <= `RES_JUMP_BRANCH; reg1_read_o <= 1'b1; reg2_read_o <= 1'b0;
270     instvalid <= 1'd0;
271     if((reg1_o[31] == 1'b1) || (reg1_o == 32'd0)) begin
272         branch_addr <= pc_4 + imm11;
273         branch <= 1'd1;
274     end
275 end
276 `BNE:     begin
277     wreg_o <= 1'd0;      aluop_o <= `BLEZ_OP;
278     alusel_o <= `RES_JUMP_BRANCH; reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
279     instvalid <= 1'd0;
280     if(reg1_o != reg2_o) begin
281         branch_addr <= pc_4 + imm11;
282         branch <= 1'd1;
283     end
284 end
285 `LW:      begin
286     wreg_o <= 1'd1;      aluop_o <= `LW_OP;
287     alusel_o <= `RES_LOAD_STORE; reg1_read_o <= 1'b1; reg2_read_o <= 1'b0;
288     wd_o <= inst_i[20:16]; instvalid <= 1'd0;
289 end
290 `SW:      begin
291     wreg_o <= 1'd0;      aluop_o <= `SW_OP;
292     reg1_read_o <= 1'b1; reg2_read_o <= 1'b1; instvalid <= 1'd0;
293     alusel_o <= `RES_LOAD_STORE;
294 end
295 `REGIMM_INST: begin
296     case (op4)
297     `BGEZ: begin
298         wreg_o <= 1'd0;      aluop_o <= `BGEZ_OP;
299         alusel_o <= `RES_JUMP_BRANCH; reg1_read_o <= 1'b1; reg2_read_o <= 1'b0;
300         instvalid <= 1'd0;
301         if(reg1_o[31] == 1'b0) begin
302             branch_addr <= pc_4 + imm11;
303             branch <= 1'd1;

```

*Id.v (part 6)*



```

304         end
305     end
306     `BLTZ:      begin
307         wreg_o <= 1'd0;      aluop_o <= `BLTZ_OP;
308         alusel_o <= `RES_JUMP_BRANCH; reg1_read_o <= 1'b1; reg2_read_o <= 1'b0;
309         instvalid <= 1'd0;
310         if(reg1_o[31] == 1'b1) begin
311             branch_addr <= pc_4 + imm511;
312             branch <= 1'd1;
313         end
314     end
315     default: begin
316     end
317 endcase
318 end
319 default:      begin
320 end
321 endcase      //case op
322
323 if (inst_i[31:21] == 11'b000000000000) begin
324     if (op3 == `SLL) begin
325         wreg_o <= 1'd1;      aluop_o <= `SLL_OP;
326         alusel_o <= `RES_SHIFT; reg1_read_o <= 1'b0; reg2_read_o <= 1'b1;
327         imm[4:0] <= inst_i[10:6];      wd_o <= inst_i[15:11];
328         instvalid <= 1'd0;
329     end else if ( op3 == `SRL ) begin
330         wreg_o <= 1'd1;      aluop_o <= `SRL_OP;
331         alusel_o <= `RES_SHIFT; reg1_read_o <= 1'b0; reg2_read_o <= 1'b1;
332         imm[4:0] <= inst_i[10:6];      wd_o <= inst_i[15:11];
333         instvalid <= 1'd0;
334     end else if ( op3 == `SRA ) begin
335         wreg_o <= 1'd1;      aluop_o <= `SRA_OP;
336         alusel_o <= `RES_SHIFT; reg1_read_o <= 1'b0; reg2_read_o <= 1'b1;
337         imm[4:0] <= inst_i[10:6];      wd_o <= inst_i[15:11];
338         instvalid <= 1'd0;
339     end
340 end
341
342 end
343 end
344
345
346 always @ (*) begin
347     stall_reg1 <= 1'd0;
348     if(~rst) begin
349         reg1_o <= 32'd0;
350     end

```

*Id.v(part 7)*

```

350     end
351     else if (ex_aluop_i == `LW_OP && ex_wd_i == reg1_addr_o && reg1_read_o == 1'b1)
352         begin
353             stall_reg1 <= 1'd1;
354         end
355     else if ((reg1_read_o == 1'b1) && (ex_wreg_i == 1'b1)
356             && (ex_wd_i == reg1_addr_o)) begin
357         reg1_o <= ex_wdata_i;
358     end else if ((reg1_read_o == 1'b1) && (mem_wreg_i == 1'b1)
359             && (mem_wd_i == reg1_addr_o)) begin
360         reg1_o <= mem_wdata_i;
361     end else if (reg1_read_o == 1'b1) begin
362         reg1_o <= reg1_data_i;
363     end else if (reg1_read_o == 1'b0) begin
364         reg1_o <= imm;
365     end else begin
366         reg1_o <= 32'd0;
367     end
368 end
369
370 always @ (*) begin
371     stall_reg2 <= 1'd0;
372     if (~rst) begin
373         reg2_o <= 32'd0;
374     end
375     else if (ex_aluop_i == `LW_OP && ex_wd_i == reg2_addr_o && reg2_read_o == 1'b1)
376         begin
377             stall_reg2 <= 1'd1;
378         end
379     else if ((reg2_read_o == 1'b1) && (ex_wreg_i == 1'b1)
380             && (ex_wd_i == reg2_addr_o)) begin
381         reg2_o <= ex_wdata_i;
382     end else if ((reg2_read_o == 1'b1) && (mem_wreg_i == 1'b1)
383             && (mem_wd_i == reg2_addr_o)) begin
384         reg2_o <= mem_wdata_i;
385     end else if (reg2_read_o == 1'b1) begin
386         reg2_o <= reg2_data_i;
387     end else if (reg2_read_o == 1'b0) begin
388         reg2_o <= imm;
389     end else begin
390         reg2_o <= 32'd0;
391     end
392 end
393 assign stall_id = stall_reg1 | stall_reg2;
394
395 endmodule
396

```

*Id.v (part 8)*