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，并没有多于的设计，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。

```
module pc(

    input                          clk,
    input                      rst,
    input  [5:0]               stall,
    input                      branch,
    input  [31:0]                  branch_addr,
    output reg [31:0]              pc = 32'd0

);
```

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。

```
module if_id(
input clk,
input rst,
input [5:0] stall,
input [31:0] if_pc,
input [31:0] if_inst,
input branch,
output reg [31:0] id_pc,
output reg [31:0] id_inst,
output reg last_branch
    );
```

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。

```
module id(

 input                        rst,
 input [31:0]                 pc_i,
 input [31:0]                    inst,

 input                        ex_wreg_i,
 input [31:0]                 ex_wdata_i,
 input [31:0]                    ex_wd_i,

    input                        mem_wreg_i,
    input [31:0]                 mem_wdata_i,
    input [31:0]                   mem_wd_i,

    input [31:0]                   reg1_data_i,
    input [31:0]                      reg2_data_i,

    input                          last_branch,
    input [7:0]                    ex_aluop_i,

    output reg                        reg1_read_o,
    output reg                        reg2_read_o,
    output reg[31:0]                  reg1_addr_o,
    output reg[31:0]                  reg2_addr_o,

    output reg[7:0]                aluop_o,
    output reg[2:0]                alusel_o,
    output reg[31:0]                  reg1_o,
    output reg[31:0]                  reg2_o,
    output reg[31:0]                  wd_o,
    output reg                        wreg_o,
    output reg                     branch,
    output reg [31:0]              branch_addr,
    output [31:0]                  this_inst,
    output                         stall_id
);
```

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。

```
module ex(

 input              rst,

 input [7:0]           aluop_i,
 input [2:0]           alusel_i,
 input [31:0]             reg1_i,
 input [31:0]             reg2_i,
 input [31:0]          wd_i,
    input                   wreg_i,
    input  [31:0]         ex_inst,


    output reg[31:0]          wd_o,
    output reg              wreg_o,
    output reg[31:0]   wdata_o,
    output  [7:0]         aluop_o,
    output  [31:0]        mem_addr_o,
    output  [31:0]        reg2_o
 );
```

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。

```
module mem(

 input                    rst,

 input   [31:0]               wd_i,
 input                        wreg_i,
 input   [31:0]               wdata_i,
 input   [7:0]                alu_op,
 input   [31:0]               mem_data_i,
    input   [31:0]               mem_addr_i,
    input   [31:0]               mem_reg2,
    output reg [31:0]               wd_o,
    output reg                      wreg_o,
    output reg [31:0]            wdata_o,
    output reg [31:0]               mem_addr_o,
    output reg                   mem_we,
    output reg [31:0]               mem_data_o
);
```

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。

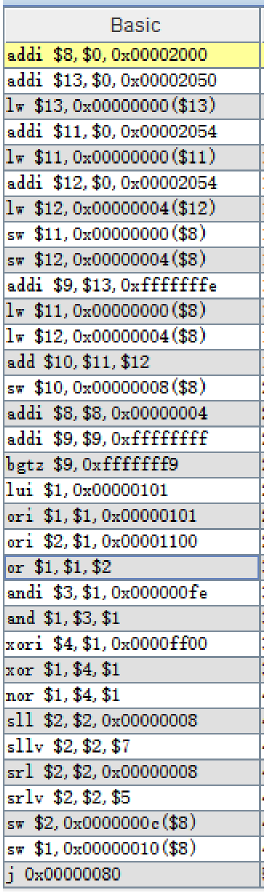
```
RAM your_instance_name (
  .a(a), // input [7 : 0] a
  .d(d), // input [31 : 0] d
  .dpra(dpra), // input [7 : 0] dpra
  .clk(clk), // input clk
  .we(we), // input we
  .spo(spo), // output [31 : 0] spo
  .dpo(dpo) // output [31 : 0] dpo
);
```

实验结果

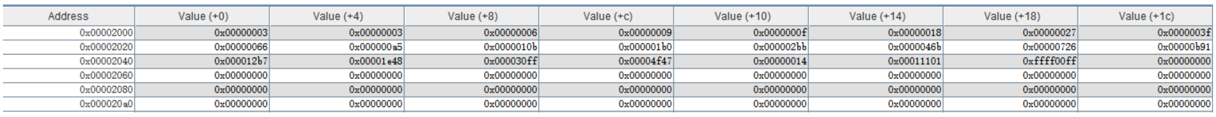
Simulation 结果

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。

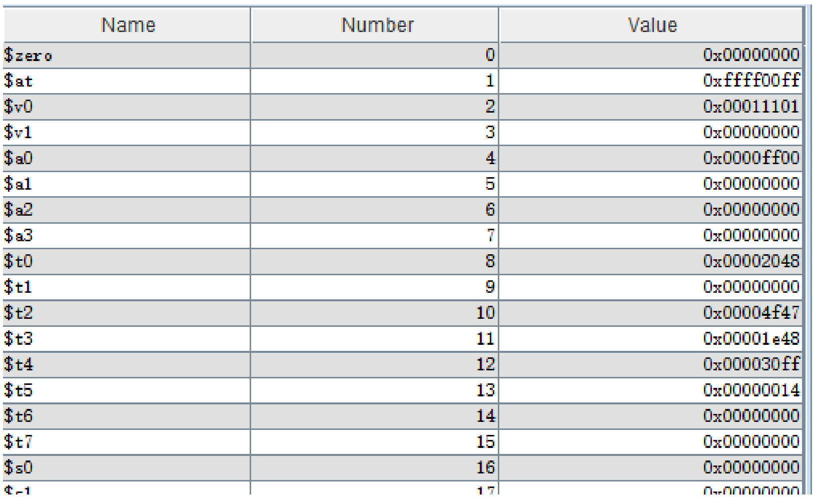


```
Basic
addi $8, $0, 0x00002000
addi $13, $0, 0x00002050
lw $13, 0x00000000($13)
addi $11, $0, 0x00002054
lw $11, 0x00000000($11)
addi $12, $0, 0x00002054
lw $12, 0x00000004($12)
sw $11, 0x00000000($8)
sw $12, 0x00000004($8)
addi $9, $13, 0xfffffffe
lw $11, 0x00000000($8)
lw $12, 0x00000004($8)
add $10, $11, $12
sw $10, 0x00000008($8)
addi $8, $8, 0x00000004
addi $9, $9, 0xffffffff
bgtz $9, 0xfffffff9
lui $1, 0x00000101
ori $1, $1, 0x00000101
ori $2, $1, 0x00001100
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	0x000001b0	0x000002bb	0x0000046b	0x00000726	0x00000b91
0x00002040	0x000012b7	0x00001e48	0x000030ff	0x00004f47	0x00000014	0x00011101	0xffff00ff	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

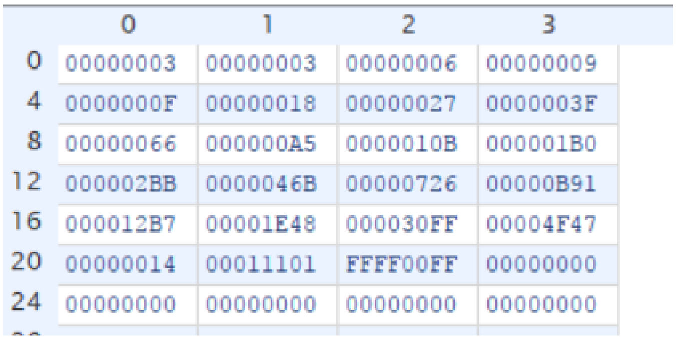


Name	Number	Value
$zero	0	0x00000000
$at	1	0xffff00ff
$v0	2	0x00011101
$v1	3	0x00000000
$a0	4	0x0000ff00
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00002048
$t1	9	0x00000000
$t2	10	0x00004f47
$t3	11	0x00001e48
$t4	12	0x000030ff
$t5	13	0x00000014
$t6	14	0x00000000
$t7	15	0x00000000
$s0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

Contents of Registers by MARS

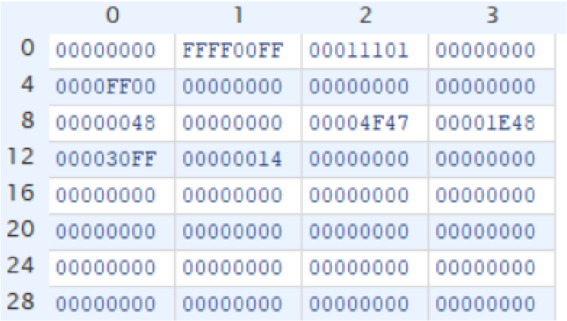
实际仿真结果

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



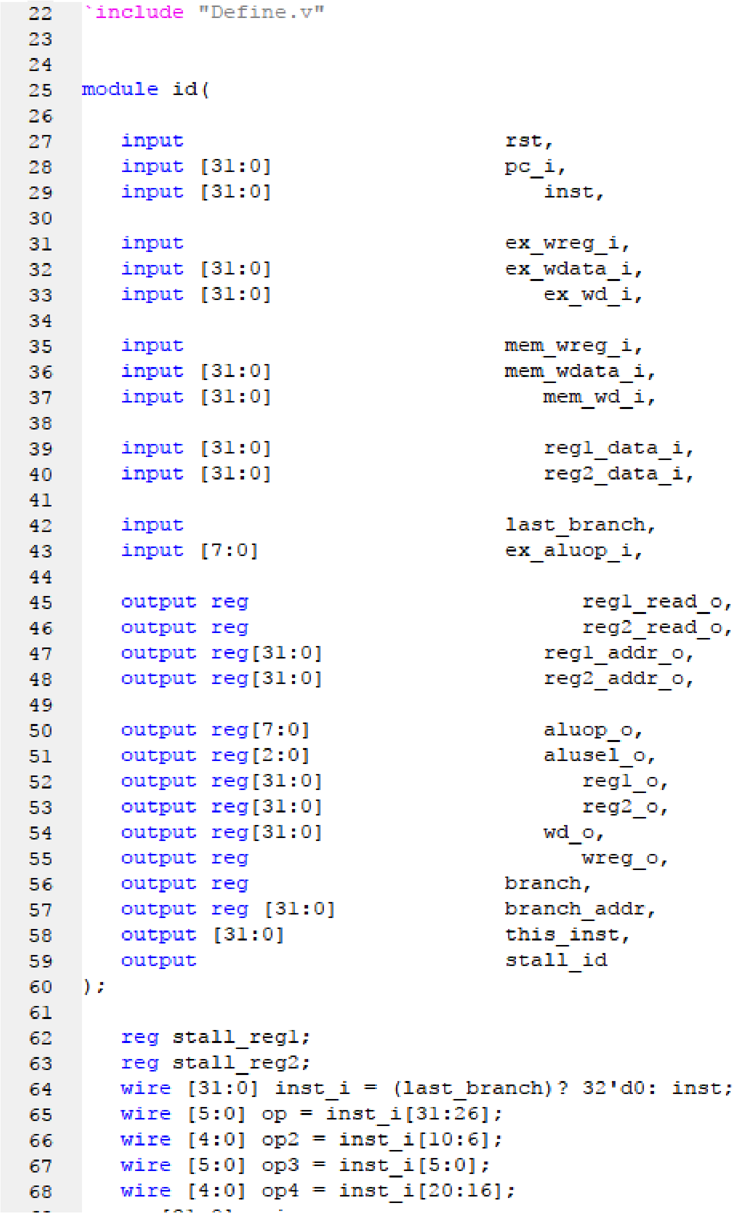
	0	1	2	3
0	00000000	FFFF00FF	00011101	00000000
4	0000FF00	00000000	00000000	00000000
8	00000048	00000000	00004F47	00001E48
12	000030FF	00000014	00000000	00000000
16	00000000	00000000	00000000	00000000
20	00000000	00000000	00000000	00000000
24	00000000	00000000	00000000	00000000
28	00000000	00000000	00000000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

Contents of Registers by CPU

下载电路板结果

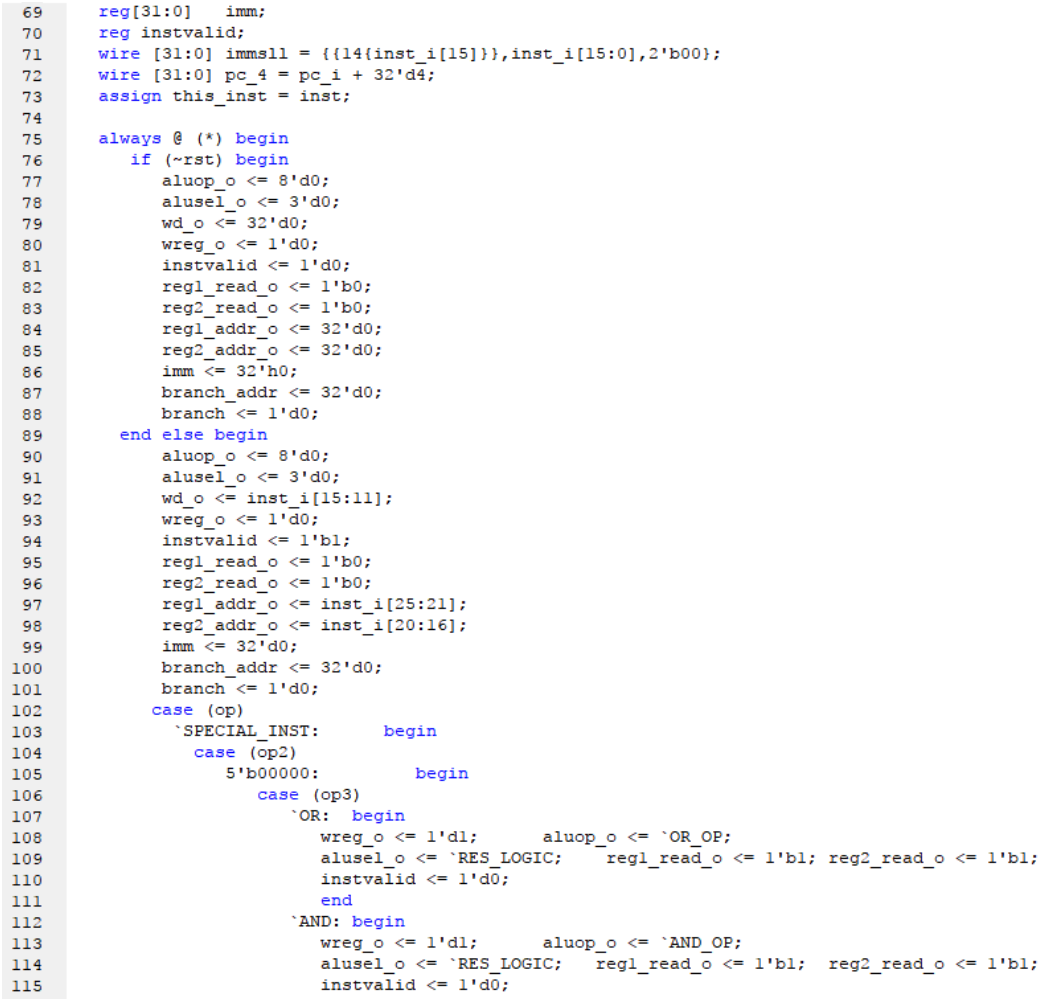
配置好管脚文件后，在实验检查现场给助教检查结果无误。使用的是开关查找内存，内存使用的是双端口内存，这样一个端口用于 CPU，一个端口用于开关查找结果，就非常方便。下载结果也正确。

附录（代码量大，只给出核心代码）



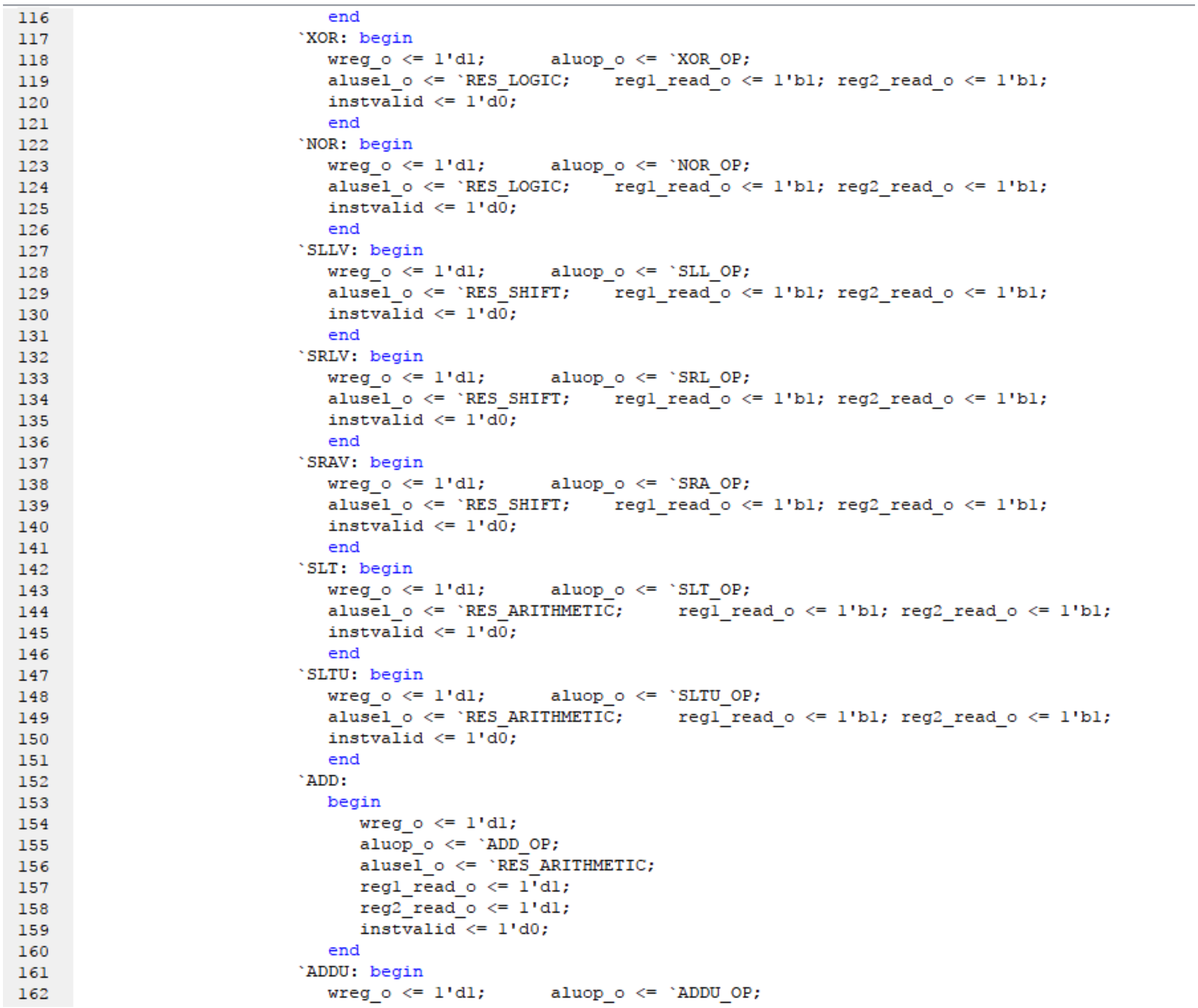
```
`include "Define.v"


module id(

    input                        rst,
    input [31:0]                 pc_i,
    input [31:0]                    inst,

    input                        ex_wreg_i,
    input [31:0]                 ex_wdata_i,
    input [31:0]                    ex_wd_i,

    input                        mem_wreg_i,
    input [31:0]                 mem_wdata_i,
    input [31:0]                    mem_wd_i,

    input [31:0]                    reg1_data_i,
    input [31:0]                    reg2_data_i,

    input                        last_branch,
    input [7:0]                  ex_aluop_i,

    output reg                         reg1_read_o,
    output reg                         reg2_read_o,
    output reg[31:0]                reg1_addr_o,
    output reg[31:0]                reg2_addr_o,

    output reg[7:0]                 aluop_o,
    output reg[2:0]                 alusel_o,
    output reg[31:0]                   reg1_o,
    output reg[31:0]                   reg2_o,
    output reg[31:0]                wd_o,
    output reg                         wreg_o,
    output reg                   branch,
    output reg [31:0]            branch_addr,
    output [31:0]                this_inst,
    output                       stall_id
);

    reg stall_reg1;
    reg stall_reg2;
    wire [31:0] inst_i = (last_branch)? 32'd0: inst;
    wire [5:0] op = inst_i[31:26];
    wire [4:0] op2 = inst_i[10:6];
    wire [5:0] op3 = inst_i[5:0];
    wire [4:0] op4 = inst_i[20:16];
```

Id.v (part 1)



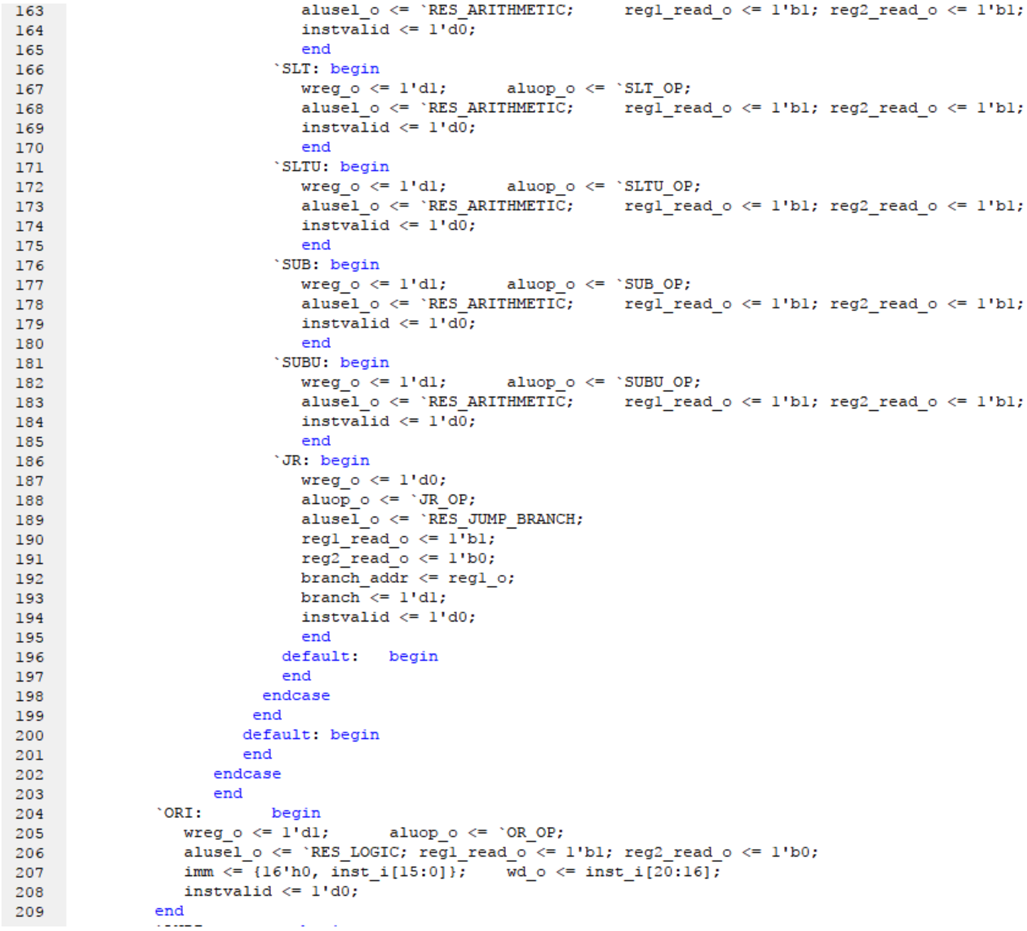
```
reg[31:0]   imm;
reg instvalid;
wire [31:0] immsll = {{14{inst_i[15]}},inst_i[15:0],2'b00};
wire [31:0] pc_4 = pc_i + 32'd4;
assign this_inst = inst;

always @ (*) begin
   if (~rst) begin
      aluop_o <= 8'd0;
      alusel_o <= 3'd0;
      wd_o <= 32'd0;
      wreg_o <= 1'd0;
      instvalid <= 1'd0;
      reg1_read_o <= 1'b0;
      reg2_read_o <= 1'b0;
      reg1_addr_o <= 32'd0;
      reg2_addr_o <= 32'd0;
      imm <= 32'h0;
      branch_addr <= 32'd0;
      branch <= 1'd0;
  end else begin
      aluop_o <= 8'd0;
      alusel_o <= 3'd0;
      wd_o <= inst_i[15:11];
      wreg_o <= 1'd0;
      instvalid <= 1'b1;
      reg1_read_o <= 1'b0;
      reg2_read_o <= 1'b0;
      reg1_addr_o <= inst_i[25:21];
      reg2_addr_o <= inst_i[20:16];
      imm <= 32'd0;
      branch_addr <= 32'd0;
      branch <= 1'd0;
     case (op)
       `SPECIAL_INST:      begin
         case (op2)
            5'b00000:        begin
               case (op3)
                  `OR:  begin
                     wreg_o <= 1'd1;      aluop_o <= `OR_OP;
                     alusel_o <= `RES_LOGIC;    reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
                     instvalid <= 1'd0;
                     end
                  `AND: begin
                     wreg_o <= 1'd1;      aluop_o <= `AND_OP;
                     alusel_o <= `RES_LOGIC;   reg1_read_o <= 1'b1;  reg2_read_o <= 1'b1;
                     instvalid <= 1'd0;
```

Id.v (part 2)



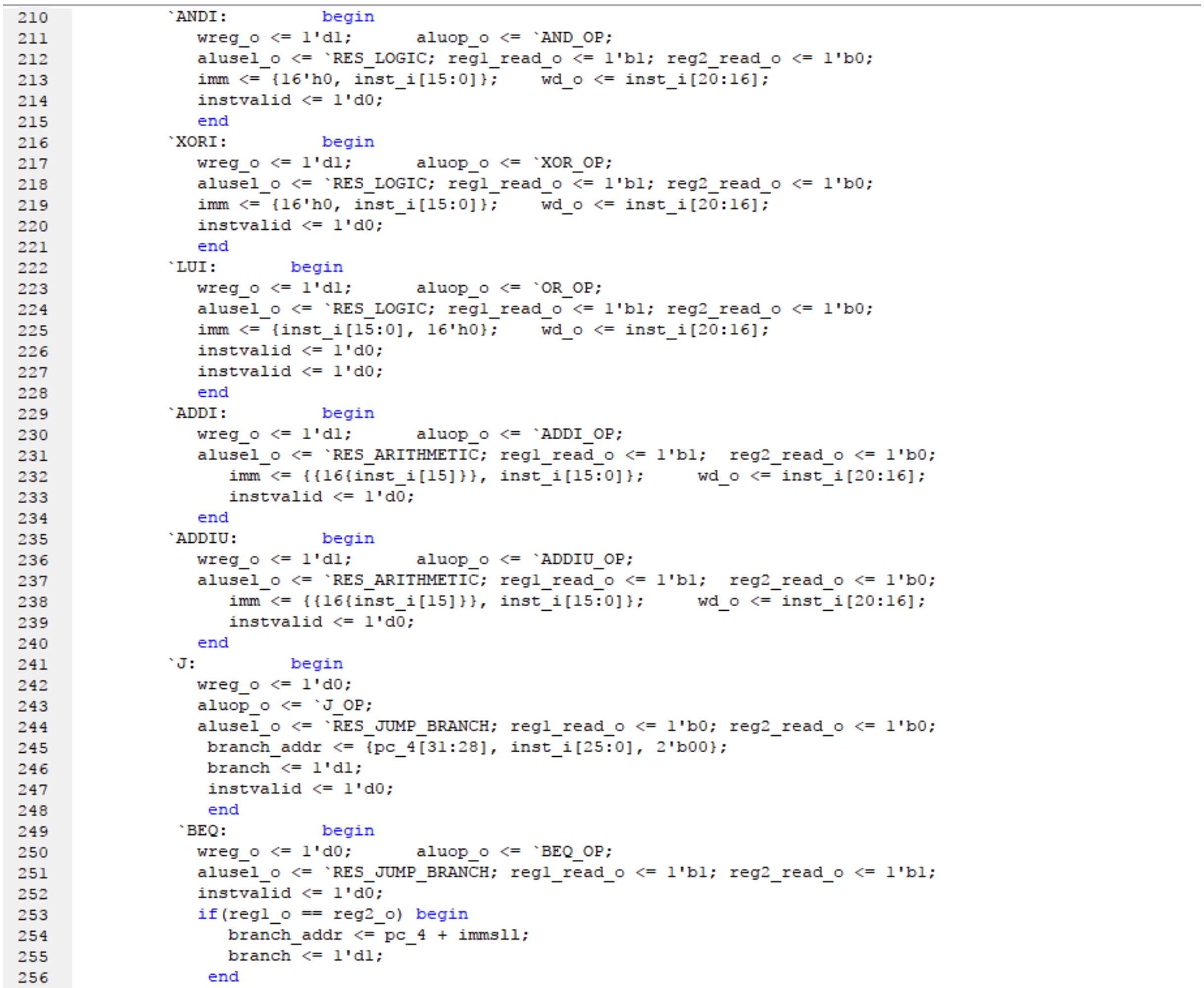
```
               end
            `XOR: begin
               wreg_o <= 1'd1;      aluop_o <= `XOR_OP;
               alusel_o <= `RES_LOGIC;    reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
               instvalid <= 1'd0;
               end
            `NOR: begin
               wreg_o <= 1'd1;      aluop_o <= `NOR_OP;
               alusel_o <= `RES_LOGIC;    reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
               instvalid <= 1'd0;
               end
            `SLLV: begin
               wreg_o <= 1'd1;      aluop_o <= `SLL_OP;
               alusel_o <= `RES_SHIFT;    reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
               instvalid <= 1'd0;
               end
            `SRLV: begin
               wreg_o <= 1'd1;      aluop_o <= `SRL_OP;
               alusel_o <= `RES_SHIFT;    reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
               instvalid <= 1'd0;
               end
            `SRAV: begin
               wreg_o <= 1'd1;      aluop_o <= `SRA_OP;
               alusel_o <= `RES_SHIFT;    reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
               instvalid <= 1'd0;
               end
            `SLT: begin
               wreg_o <= 1'd1;      aluop_o <= `SLT_OP;
               alusel_o <= `RES_ARITHMETIC;     reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
               instvalid <= 1'd0;
               end
            `SLTU: begin
               wreg_o <= 1'd1;      aluop_o <= `SLTU_OP;
               alusel_o <= `RES_ARITHMETIC;     reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
               instvalid <= 1'd0;
               end
            `ADD:
               begin
                  wreg_o <= 1'd1;
                  aluop_o <= `ADD_OP;
                  alusel_o <= `RES_ARITHMETIC;
                  reg1_read_o <= 1'd1;
                  reg2_read_o <= 1'd1;
                  instvalid <= 1'd0;
               end
            `ADDU: begin
               wreg_o <= 1'd1;      aluop_o <= `ADDU_OP;
```

Id.v (part 3)



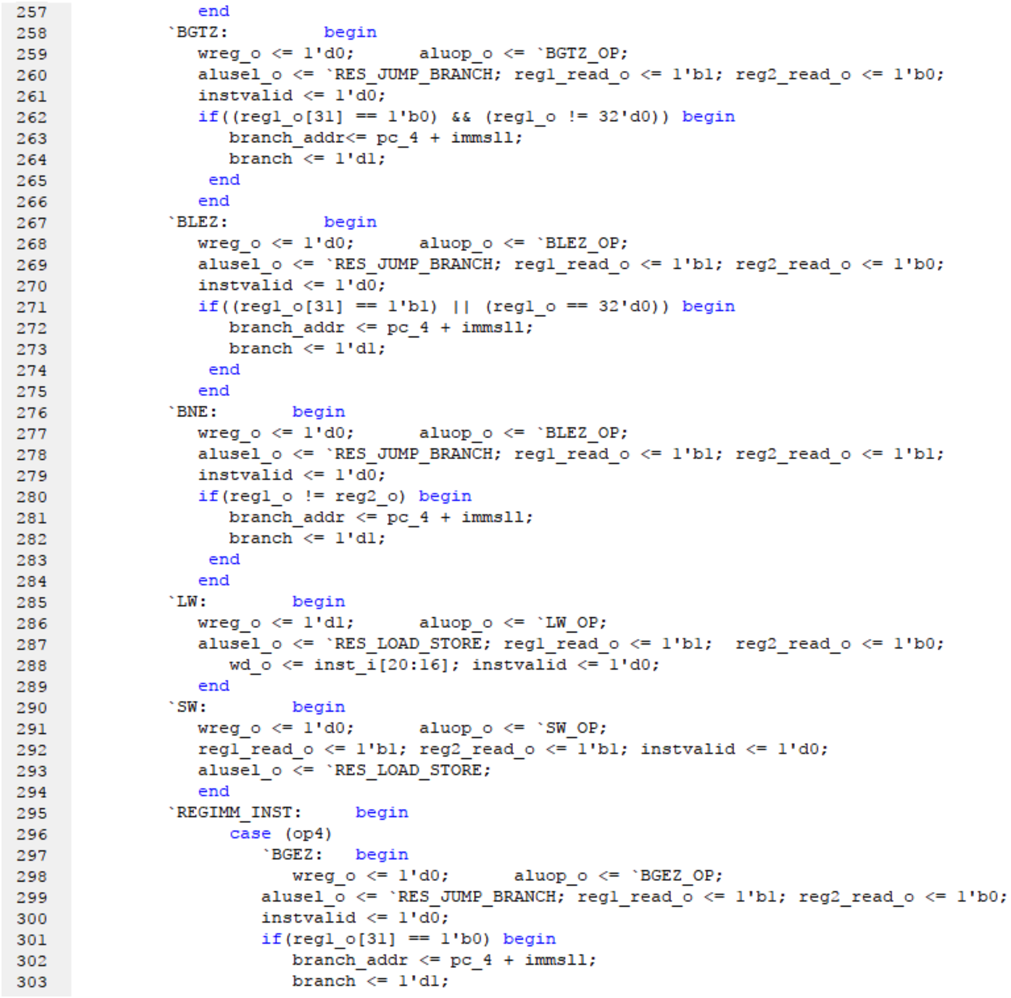
```
               alusel_o <= `RES_ARITHMETIC;     reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
               instvalid <= 1'd0;
               end
            `SLT: begin
               wreg_o <= 1'd1;      aluop_o <= `SLT_OP;
               alusel_o <= `RES_ARITHMETIC;     reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
               instvalid <= 1'd0;
               end
            `SLTU: begin
               wreg_o <= 1'd1;      aluop_o <= `SLTU_OP;
               alusel_o <= `RES_ARITHMETIC;     reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
               instvalid <= 1'd0;
               end
            `SUB: begin
               wreg_o <= 1'd1;      aluop_o <= `SUB_OP;
               alusel_o <= `RES_ARITHMETIC;     reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
               instvalid <= 1'd0;
               end
            `SUBU: begin
               wreg_o <= 1'd1;      aluop_o <= `SUBU_OP;
               alusel_o <= `RES_ARITHMETIC;     reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
               instvalid <= 1'd0;
               end
            `JR: begin
               wreg_o <= 1'd0;
               aluop_o <= `JR_OP;
               alusel_o <= `RES_JUMP_BRANCH;
               reg1_read_o <= 1'b1;
               reg2_read_o <= 1'b0;
               branch_addr <= reg1_o;
               branch <= 1'd1;
               instvalid <= 1'd0;
               end
             default:   begin
             end
           endcase
          end
         default: begin
         end
      endcase
      end
`ORI:       begin
   wreg_o <= 1'd1;      aluop_o <= `OR_OP;
   alusel_o <= `RES_LOGIC; reg1_read_o <= 1'b1; reg2_read_o <= 1'b0;
   imm <= {16'h0, inst_i[15:0]};    wd_o <= inst_i[20:16];
   instvalid <= 1'd0;
end
```

Id.v(part 4)



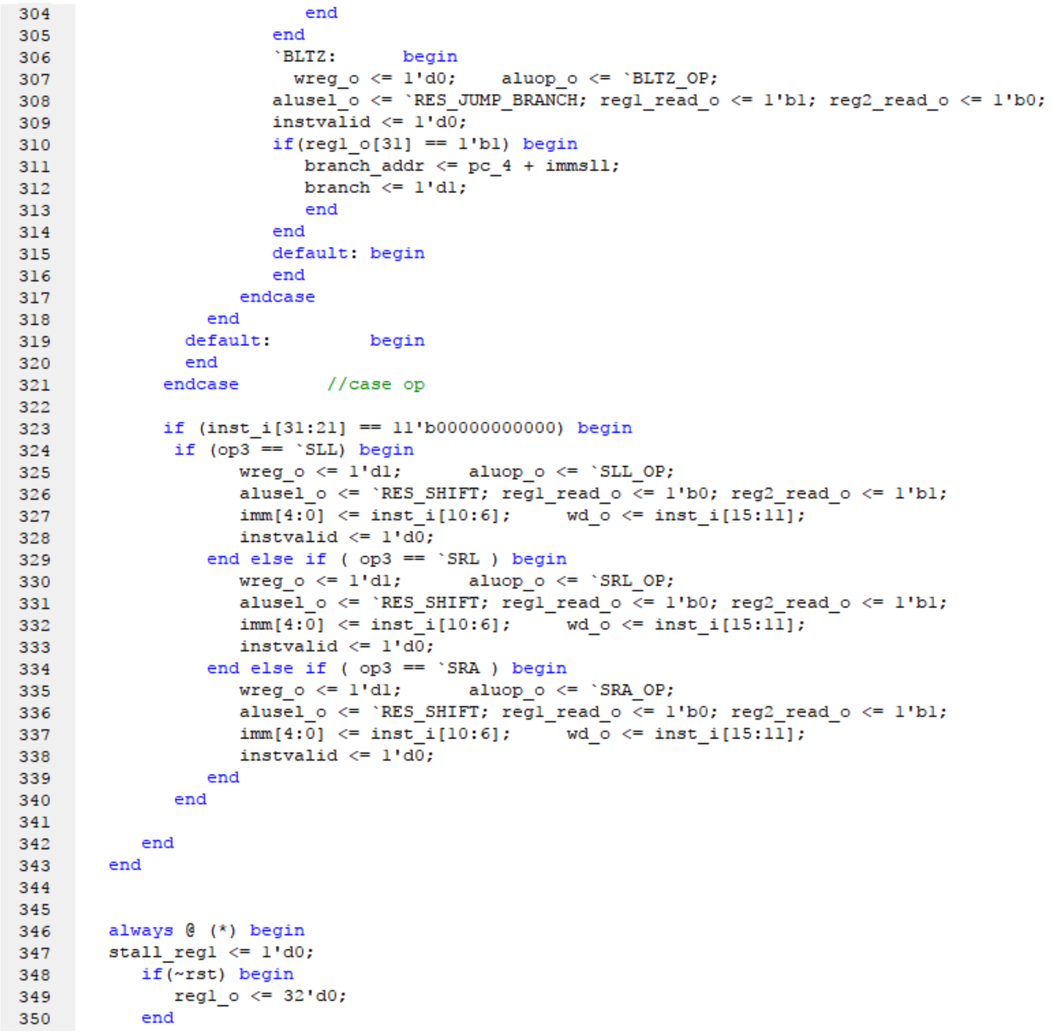
```
        `ANDI:         begin
           wreg_o <= 1'd1;      aluop_o <= `AND_OP;
           alusel_o <= `RES_LOGIC; reg1_read_o <= 1'b1; reg2_read_o <= 1'b0;
           imm <= {16'h0, inst_i[15:0]};    wd_o <= inst_i[20:16];
           instvalid <= 1'd0;
           end
        `XORI:         begin
           wreg_o <= 1'd1;      aluop_o <= `XOR_OP;
           alusel_o <= `RES_LOGIC; reg1_read_o <= 1'b1; reg2_read_o <= 1'b0;
           imm <= {16'h0, inst_i[15:0]};    wd_o <= inst_i[20:16];
           instvalid <= 1'd0;
           end
        `LUI:       begin
           wreg_o <= 1'd1;      aluop_o <= `OR_OP;
           alusel_o <= `RES_LOGIC; reg1_read_o <= 1'b1; reg2_read_o <= 1'b0;
           imm <= {inst_i[15:0], 16'h0};    wd_o <= inst_i[20:16];
           instvalid <= 1'd0;
           instvalid <= 1'd0;
           end
        `ADDI:         begin
           wreg_o <= 1'd1;      aluop_o <= `ADDI_OP;
           alusel_o <= `RES_ARITHMETIC; reg1_read_o <= 1'b1;  reg2_read_o <= 1'b0;
              imm <= {{16{inst_i[15]}}, inst_i[15:0]};     wd_o <= inst_i[20:16];
              instvalid <= 1'd0;
           end
        `ADDIU:        begin
           wreg_o <= 1'd1;      aluop_o <= `ADDIU_OP;
           alusel_o <= `RES_ARITHMETIC; reg1_read_o <= 1'b1;  reg2_read_o <= 1'b0;
              imm <= {{16{inst_i[15]}}, inst_i[15:0]};     wd_o <= inst_i[20:16];
              instvalid <= 1'd0;
           end
        `J:         begin
           wreg_o <= 1'd0;
           aluop_o <= `J_OP;
           alusel_o <= `RES_JUMP_BRANCH; reg1_read_o <= 1'b0; reg2_read_o <= 1'b0;
            branch_addr <= {pc_4[31:28], inst_i[25:0], 2'b00};
            branch <= 1'd1;
            instvalid <= 1'd0;
            end
         `BEQ:         begin
           wreg_o <= 1'd0;      aluop_o <= `BEQ_OP;
           alusel_o <= `RES_JUMP_BRANCH; reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
           instvalid <= 1'd0;
           if(reg1_o == reg2_o) begin
              branch_addr <= pc_4 + immsll;
              branch <= 1'd1;
            end
```

Id.v (part 5)



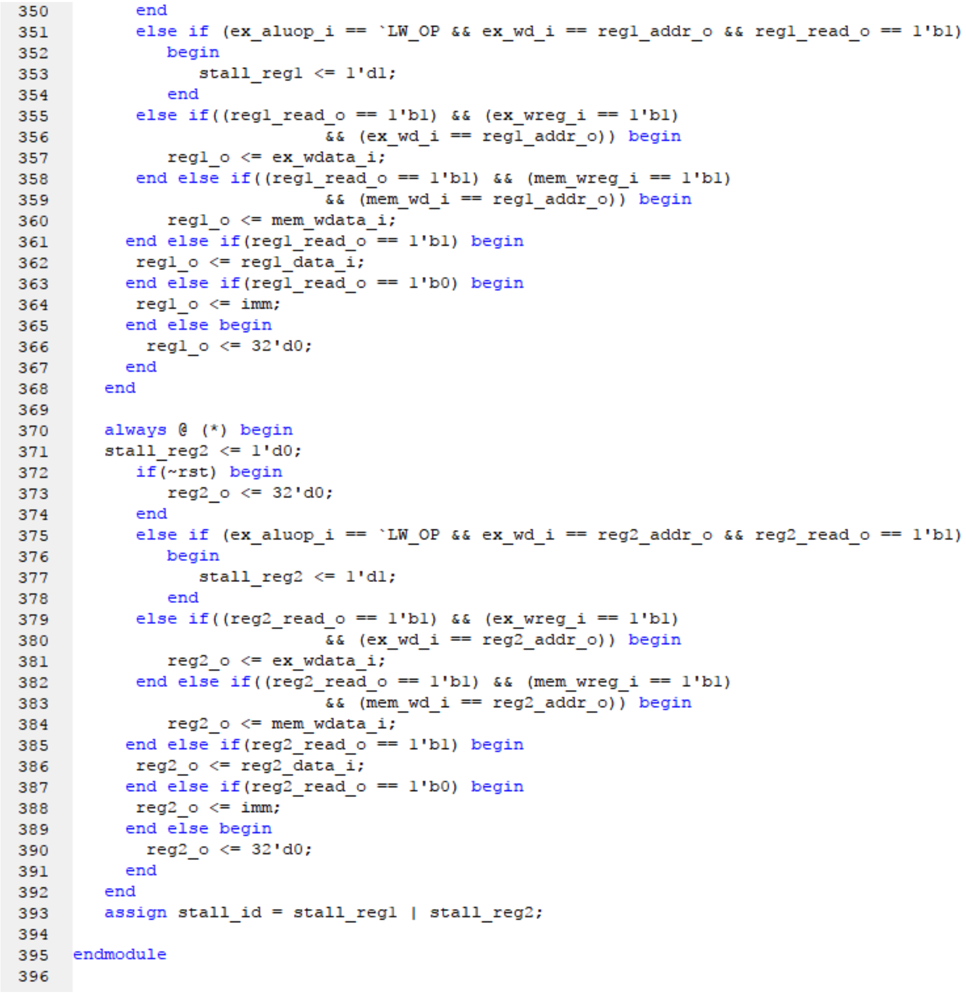
```
           end
        `BGTZ:          begin
           wreg_o <= 1'd0;      aluop_o <= `BGTZ_OP;
           alusel_o <= `RES_JUMP_BRANCH; reg1_read_o <= 1'b1; reg2_read_o <= 1'b0;
           instvalid <= 1'd0;
           if((reg1_o[31] == 1'b0) && (reg1_o != 32'd0)) begin
              branch_addr<= pc_4 + immsll;
              branch <= 1'd1;
            end
           end
        `BLEZ:          begin
           wreg_o <= 1'd0;      aluop_o <= `BLEZ_OP;
           alusel_o <= `RES_JUMP_BRANCH; reg1_read_o <= 1'b1; reg2_read_o <= 1'b0;
           instvalid <= 1'd0;
           if((reg1_o[31] == 1'b1) || (reg1_o == 32'd0)) begin
              branch_addr <= pc_4 + immsll;
              branch <= 1'd1;
            end
           end
        `BNE:        begin
           wreg_o <= 1'd0;      aluop_o <= `BLEZ_OP;
           alusel_o <= `RES_JUMP_BRANCH; reg1_read_o <= 1'b1; reg2_read_o <= 1'b1;
           instvalid <= 1'd0;
           if(reg1_o != reg2_o) begin
              branch_addr <= pc_4 + immsll;
              branch <= 1'd1;
            end
           end
        `LW:         begin
           wreg_o <= 1'd1;      aluop_o <= `LW_OP;
           alusel_o <= `RES_LOAD_STORE; reg1_read_o <= 1'b1;  reg2_read_o <= 1'b0;
              wd_o <= inst_i[20:16]; instvalid <= 1'd0;
           end
        `SW:         begin
           wreg_o <= 1'd0;      aluop_o <= `SW_OP;
           reg1_read_o <= 1'b1; reg2_read_o <= 1'b1; instvalid <= 1'd0;
           alusel_o <= `RES_LOAD_STORE;
           end
        `REGIMM_INST:     begin
              case (op4)
                 `BGEZ:    begin
                    wreg_o <= 1'd0;      aluop_o <= `BGEZ_OP;
                 alusel_o <= `RES_JUMP_BRANCH; reg1_read_o <= 1'b1; reg2_read_o <= 1'b0;
                 instvalid <= 1'd0;
                 if(reg1_o[31] == 1'b0) begin
                    branch_addr <= pc_4 + immsll;
                    branch <= 1'd1;
```

Id.v (part 6)



```
                  end
               end
               `BLTZ:      begin
                 wreg_o <= 1'd0;    aluop_o <= `BLTZ_OP;
               alusel_o <= `RES_JUMP_BRANCH; reg1_read_o <= 1'b1; reg2_read_o <= 1'b0;
               instvalid <= 1'd0;
               if(reg1_o[31] == 1'b1) begin
                  branch_addr <= pc_4 + immsll;
                  branch <= 1'd1;
                  end
               end
               default: begin
               end
            endcase
         end
       default:         begin
       end
     endcase        //case op

     if (inst_i[31:21] == 11'b00000000000) begin
      if (op3 == `SLL) begin
            wreg_o <= 1'd1;      aluop_o <= `SLL_OP;
            alusel_o <= `RES_SHIFT; reg1_read_o <= 1'b0; reg2_read_o <= 1'b1;
            imm[4:0] <= inst_i[10:6];     wd_o <= inst_i[15:11];
            instvalid <= 1'd0;
         end else if ( op3 == `SRL ) begin
            wreg_o <= 1'd1;      aluop_o <= `SRL_OP;
            alusel_o <= `RES_SHIFT; reg1_read_o <= 1'b0; reg2_read_o <= 1'b1;
            imm[4:0] <= inst_i[10:6];     wd_o <= inst_i[15:11];
            instvalid <= 1'd0;
         end else if ( op3 == `SRA ) begin
            wreg_o <= 1'd1;      aluop_o <= `SRA_OP;
            alusel_o <= `RES_SHIFT; reg1_read_o <= 1'b0; reg2_read_o <= 1'b1;
            imm[4:0] <= inst_i[10:6];     wd_o <= inst_i[15:11];
            instvalid <= 1'd0;
         end
      end

   end
end


always @ (*) begin
stall_reg1 <= 1'd0;
   if(~rst) begin
      reg1_o <= 32'd0;
   end
```

Id.v(part 7)



```
      end
      else if (ex_aluop_i == `LW_OP && ex_wd_i == reg1_addr_o && reg1_read_o == 1'b1)
         begin
            stall_reg1 <= 1'd1;
         end
      else if((reg1_read_o == 1'b1) && (ex_wreg_i == 1'b1)
                        && (ex_wd_i == reg1_addr_o)) begin
         reg1_o <= ex_wdata_i;
      end else if((reg1_read_o == 1'b1) && (mem_wreg_i == 1'b1)
                        && (mem_wd_i == reg1_addr_o)) begin
         reg1_o <= mem_wdata_i;
     end else if(reg1_read_o == 1'b1) begin
      reg1_o <= reg1_data_i;
     end else if(reg1_read_o == 1'b0) begin
      reg1_o <= imm;
     end else begin
       reg1_o <= 32'd0;
     end
   end

   always @ (*) begin
   stall_reg2 <= 1'd0;
      if(~rst) begin
         reg2_o <= 32'd0;
      end
      else if (ex_aluop_i == `LW_OP && ex_wd_i == reg2_addr_o && reg2_read_o == 1'b1)
         begin
            stall_reg2 <= 1'd1;
         end
      else if((reg2_read_o == 1'b1) && (ex_wreg_i == 1'b1)
                        && (ex_wd_i == reg2_addr_o)) begin
         reg2_o <= ex_wdata_i;
      end else if((reg2_read_o == 1'b1) && (mem_wreg_i == 1'b1)
                        && (mem_wd_i == reg2_addr_o)) begin
         reg2_o <= mem_wdata_i;
     end else if(reg2_read_o == 1'b1) begin
      reg2_o <= reg2_data_i;
     end else if(reg2_read_o == 1'b0) begin
      reg2_o <= imm;
     end else begin
       reg2_o <= 32'd0;
     end
   end
   assign stall_id = stall_reg1 | stall_reg2;

endmodule

```

Id.v (part 8)