计算机组成课程设计

P5 课下测试 – Verilog 流水线

周美廷-76066002

*由于本人是留学生，随最终文档依然使用中文写但为了本人无需翻来覆去看英文版和中文版，于是将文档写成两种语言

1. 作业概括 (Homework Summary)

在上次实验的基础上，使用 Verilog 语言设计一个流水线处理器。以下是流水线 CPU 的设计图。

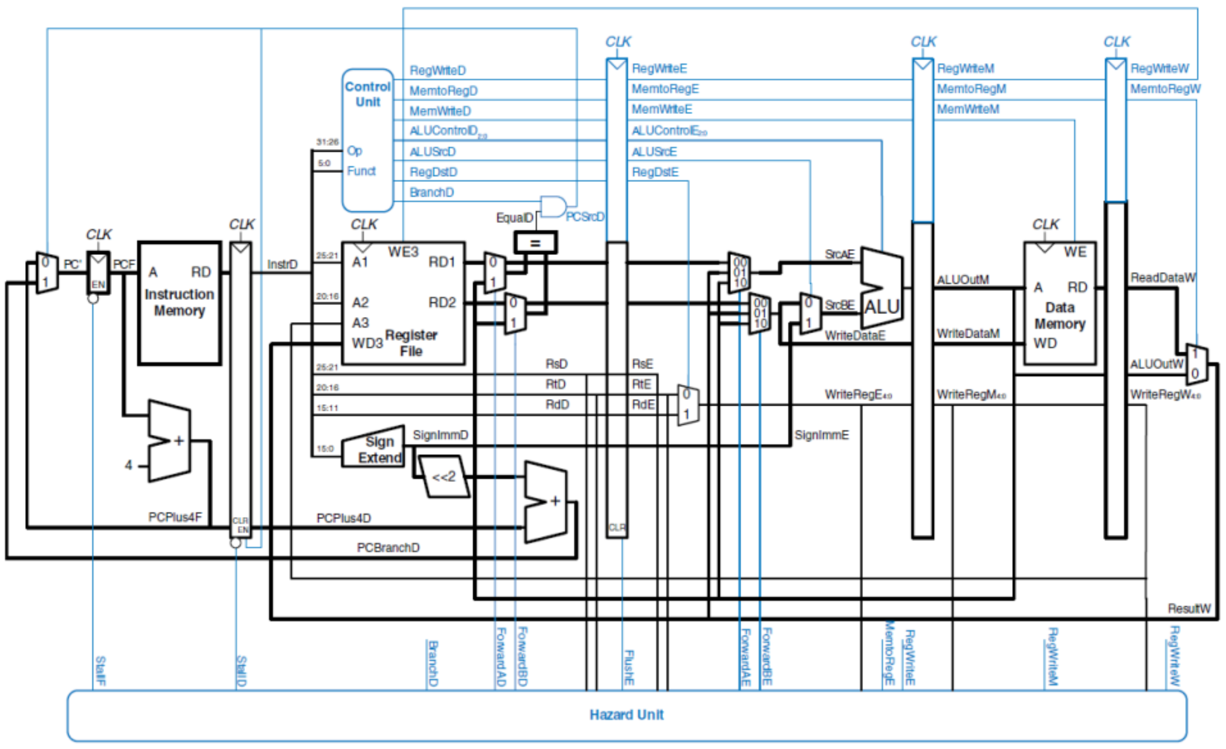
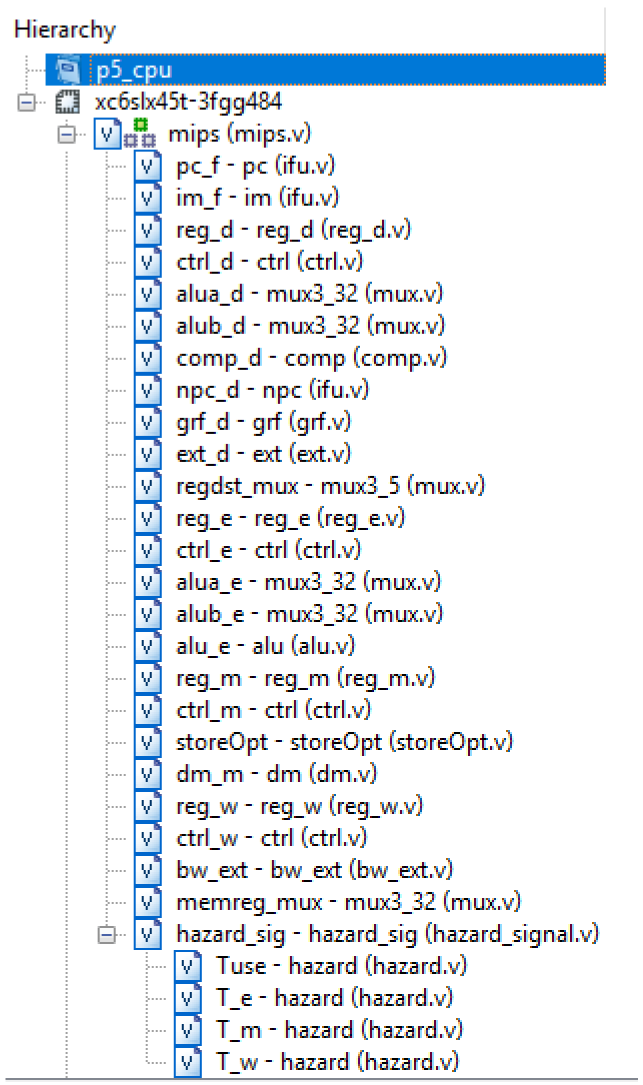


Figure 7.58 Pipelined processor with full hazard handling

图 1 流水线 CPU 设计图

以下是本人在 Verilog ISE 中实现出来的顶层代码结构。



```
Hierarchy
  p5_cpu
  xc6slx45t-3fgg484
    mips (mips.v)
      pc_f - pc (ifu.v)
      im_f - im (ifu.v)
      reg_d - reg_d (reg_d.v)
      ctrl_d - ctrl (ctrl.v)
      alua_d - mux3_32 (mux.v)
      alub_d - mux3_32 (mux.v)
      comp_d - comp (comp.v)
      npc_d - npc (ifu.v)
      grf_d - grf (grf.v)
      ext_d - ext (ext.v)
      regdst_mux - mux3_5 (mux.v)
      reg_e - reg_e (reg_e.v)
      ctrl_e - ctrl (ctrl.v)
      alua_e - mux3_32 (mux.v)
      alub_e - mux3_32 (mux.v)
      alu_e - alu (alu.v)
      reg_m - reg_m (reg_m.v)
      ctrl_m - ctrl (ctrl.v)
      storeOpt - storeOpt (storeOpt.v)
      dm_m - dm (dm.v)
      reg_w - reg_w (reg_w.v)
      ctrl_w - ctrl (ctrl.v)
      bw_ext - bw_ext (bw_ext.v)
      memreg_mux - mux3_32 (mux.v)
      hazard_sig - hazard_sig (hazard_signal.v)
        Tuse - hazard (hazard.v)
        T_e - hazard (hazard.v)
        T_m - hazard (hazard.v)
        T_w - hazard (hazard.v)
```

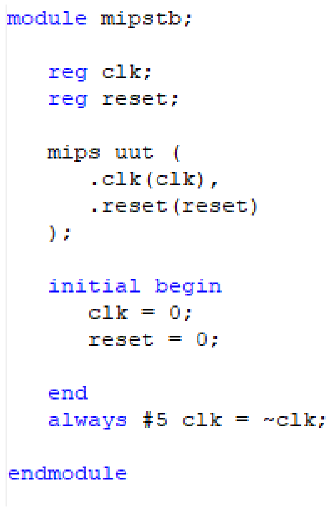
图 2 Verilog 中的代码结构

从**图 1** 可看到 mips.v 是顶层代码，其负责连线任务，把相关模块连起来，自己只有两个输入。

表 1 mips.v 规格

功能名称	方向	功能描述
clk	I	时钟信号
reset	I	复位信号

于是，testbench 中只需要驱动 clk 和 reset：



```
module mipstb;

   reg clk;
   reg reset;

   mips uut (
      .clk(clk),
      .reset(reset)
   );

   initial begin
      clk = 0;
      reset = 0;

   end
   always #5 clk = ~clk;

endmodule
```

图 3 CPU 的 testbench 代码

2. 模块规格 (Module Specifications)

这种单周期 CPU 应采用模块化设计，因此它包含多个具有不同用途的模块。

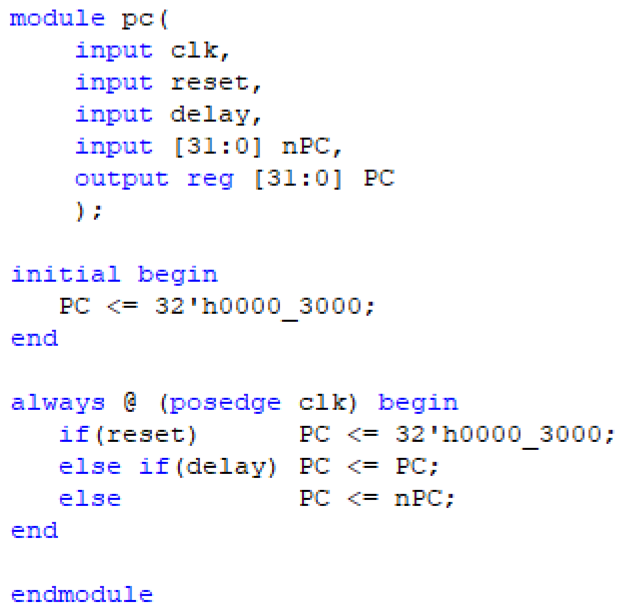
This single-cycle CPU is meant to be modularly designed and therefore it consists of several modules with different purposes.

I. IFU（取指令单元 / Instruction Fetch Unit）

IFU 模块分程 3 个主要部分，以下是详细细节：

The IFU module is divided into 3 main modules: PC (Program Counter), IM (Instruction Memory), and nPC (PC Destination).

A. PC (Program Counter)



```
module pc(
    input clk,
    input reset,
    input delay,
    input [31:0] nPC,
    output reg [31:0] PC
    );

initial begin
   PC <= 32'h0000_3000;
end

always @ (posedge clk) begin
   if(reset)       PC <= 32'h0000_3000;
   else if(delay) PC <= PC;
   else            PC <= nPC;
end

endmodule
```

图 4 ifu.v 中的 PC 模块

表 2 ifu.v - PC 规格

功能名称	方向	功能描述
clk	I	时钟信号
reset	I	复位信号
delay	I	CPU 暂停信号
nPC [31:0]	I	目标地址
PC [31:0]	O	当前程序地址

设计说明/Design Details:

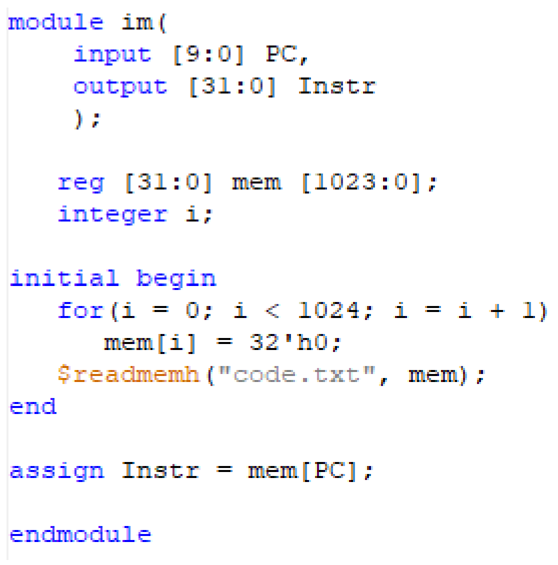
- 复位后，PC 指向 0x0000_3000，此处为第一条指令的地址。（After reset, PC will point to 0x0000_3000 which is also the first instruction address）

模块流程/Design Workflow:

- PC’s starting address is 0x0000_3000, which means if reset is valid, PC will point to this address

- If the CPU has to be paused (delay signal is TRUE), PC will point to itself, meaning that it won’t point to the next command address which will cause the CPU to not execute the next command until the CPU isn’t paused
- Otherwise PC will just go to the next destination (nPC).

B. IM



```
module im(
     input [9:0] PC,
     output [31:0] Instr
     );

   reg [31:0] mem [1023:0];
   integer i;

initial begin
   for(i = 0; i < 1024; i = i + 1)
      mem[i] = 32'h0;
   $readmemh("code.txt", mem);
end

assign Instr = mem[PC];

endmodule
```

图 5 ifu.v 中的 IM 模块

表 3 ifu.v - IM 规格

功能名称	方向	功能描述
PC [9:0]	I	PC 的第 [11:2] 地址
Instr [31:0]	O	当前指令

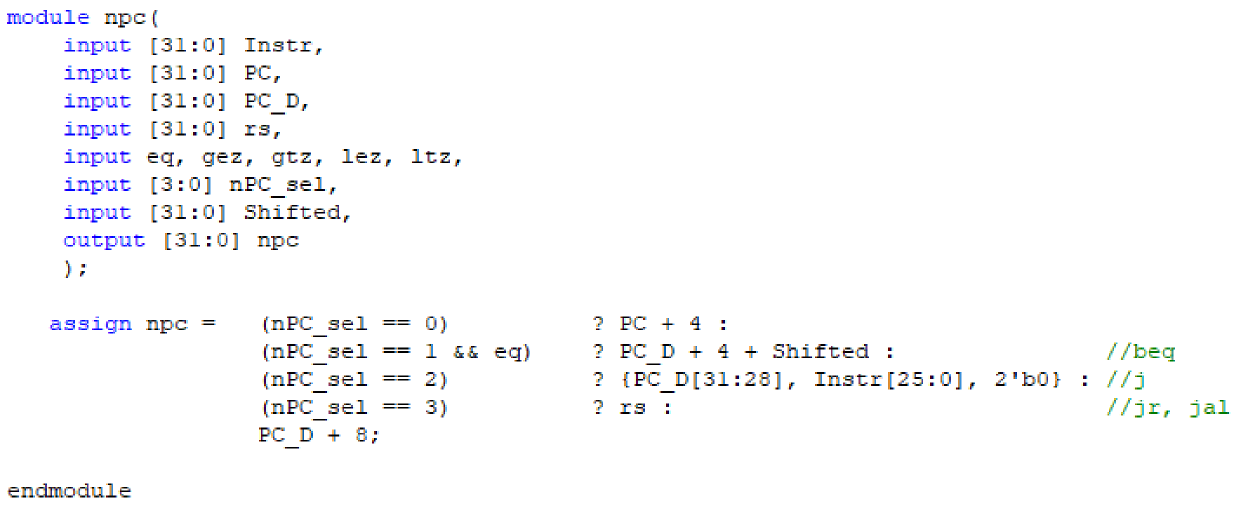
设计说明/Design Details:

- IM 容量为 4KB（32bit×1024 字）（The capacity of IM is 4 KB or 32bit x 1024 word）
- 采用$readmemh 指令来完成相应的功能 （This module is using $readmemh function to do required actions）

模块流程/Design Workflow:

- Set an initial value for the entire Instruction Memory as 0, meaning there are no instructions at start.
- Read the instructions from **code.txt** using **$readmemh** and save it to the IM

C. nPC



```
module npc(
    input [31:0] Instr,
    input [31:0] PC,
    input [31:0] PC_D,
    input [31:0] rs,
    input eq, gez, gtz, lez, ltz,
    input [3:0] nPC_sel,
    input [31:0] Shifted,
    output [31:0] npc
    );

   assign npc =    (nPC_sel == 0)          ? PC + 4 :
                   (nPC_sel == 1 && eq)    ? PC_D + 4 + Shifted :               //beq
                   (nPC_sel == 2)          ? {PC_D[31:28], Instr[25:0], 2'b0} : //j
                   (nPC_sel == 3)          ? rs :                               //jr, jal
                   PC_D + 8;

endmodule
```

图 6 ifu.v 中的 nPC 模块

表 4 ifu.v - nPC 规格

功能名称	方向	功能描述
Instr [31:0]	I	当前指令
PC [31:0]	I	F Level 的 PC 地址
PC_D [31:0]	I	D Level 的 PC 地址
rs [31:0]	I	jr 指令用来当跳转地址
eq	I	两个比较数据是否相等的信号
nPC_sel	I	跳转指令的信号
Shifted [31:0]	I	beq 用的有符号移
nPC [31:0]	O	下一个目标地址

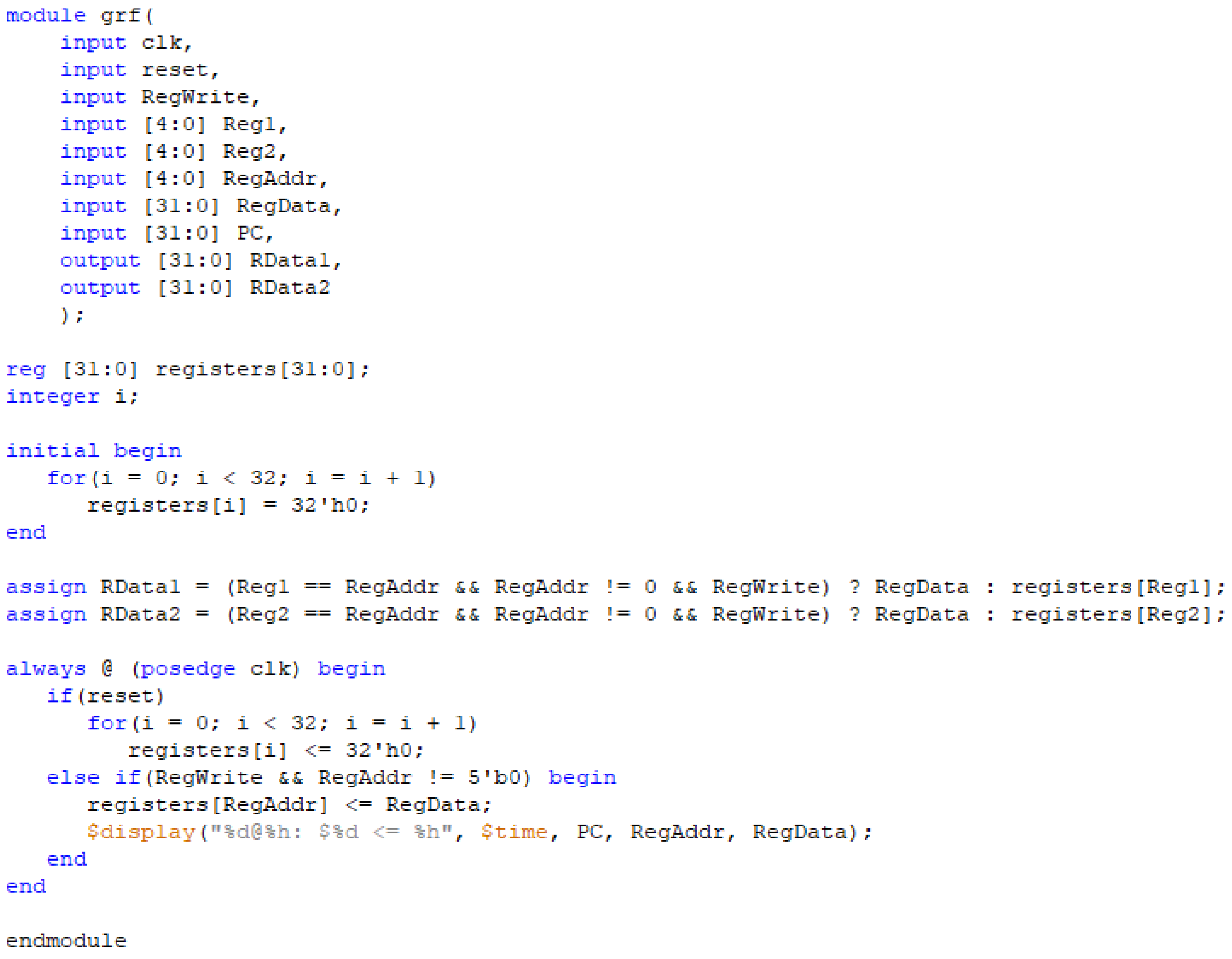
设计说明/Design Details:

- 这设计中有四个跳转指令会影响到 PC 的目标地址，**beq, jal, jr**, **j**（There are 4 jump instructions that can affect the PC address destination, **beq, jal, jr, j**）

模块流程/Design Workflow:

- Define the actions of different jump conditions, by default it’s the normal PC+4
- 2’b01 = beq | 2’b10 = jal/j | 2’b11 = jr

II. GRF（通用寄存器组/General Register File）



```
module grf(
    input clk,
    input reset,
    input RegWrite,
    input [4:0] Reg1,
    input [4:0] Reg2,
    input [4:0] RegAddr,
    input [31:0] RegData,
    input [31:0] PC,
    output [31:0] RData1,
    output [31:0] RData2
    );

reg [31:0] registers[31:0];
integer i;

initial begin
   for(i = 0; i < 32; i = i + 1)
      registers[i] = 32'h0;
end

assign RData1 = (Reg1 == RegAddr && RegAddr != 0 && RegWrite) ? RegData : registers[Reg1];
assign RData2 = (Reg2 == RegAddr && RegAddr != 0 && RegWrite) ? RegData : registers[Reg2];

always @ (posedge clk) begin
   if(reset)
      for(i = 0; i < 32; i = i + 1)
         registers[i] <= 32'h0;
   else if(RegWrite && RegAddr != 5'b0) begin
      registers[RegAddr] <= RegData;
      $display("%d@%h: $%d <= %h", $time, PC, RegAddr, RegData);
   end
end

endmodule
```

图 7 grf.v 模块

表 5 grf.v 规格

功能名称	方向	功能描述

clk	I	时钟信号
reset	I	复位信号
RegWrite	I	写控制信号
Reg1 [4:0]	I	读寄存器地址 1
Reg2 [4:0]	I	读寄存器地址 2
RegAddr [4:0]	I	目标写寄存器地址
RegData [31:0]	I	写入数据
PC [31:0]	I	当前 PC 地址
RData1 [31:0]	O	32 位数据 1
RData2 [31:0]	O	32 位数据 2

设计说明/Design Details:

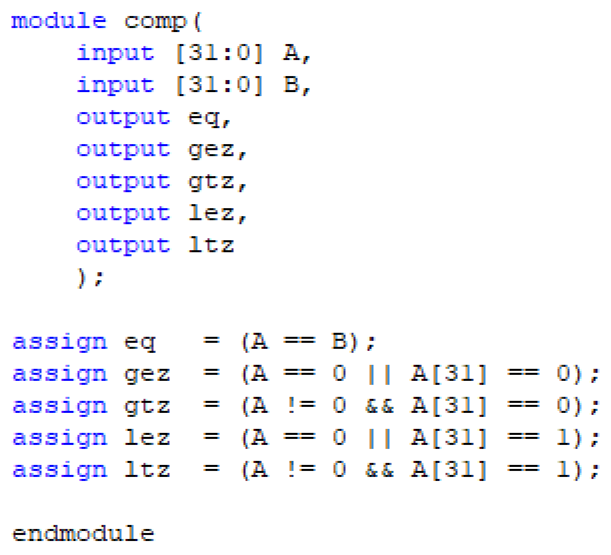
- 容量为 **4KB** (32bit/word×1024word)。（The capacity of this module is 4KB which means 32bit/word×1024word）
- 0 号寄存器的值始终保持为 0。其他寄存器初始值均为 0，无需专门设置（The value of Register No. 0 is always 0. The other registers have an initial value of 0 so they don’t require special settings）
- 每个时钟上升沿到来时若要写入数据(即写使能信号为 1 且非 reset 时)则输出写入的位置及写入的值，格式为 : (At clock posedge, always display some data information using the following command)
  $display("%d@%h: $%d <= %h", $time, PC, RegAddr, RegData);

模块流程/Design Workflow:

- In total, there are 5 different inputs in this GRF module:
  - The first two inputs are taken from the instruction code generated by IFU (Address no. 25-21 (rs/base/offset) and 20-16 (rt))
  - The third one is either Address no 20-16 (rt) in case of **ori, lw, sw, beq, lui** or 15-11 (rd) in case of **addu** and **subu**
  - The fourth one is the RegWrite signal (this will be TRUE if the current instruction isn’t **sw, jr, j** nor **beq**)
  - The fifth one is Register Data (RegData)

- At clock posedge, if RegWrite signal is TRUE and RegAddr isn’t 0, then write the RegData input to the corresponding register, after that display the information wanted.
- For RData1 and RData2, if it’s a write command (RegWrite == True), then both will be assigned the value of the write data (RegData), otherwise just output the data that is to be read.

III. Comp（数据比较单元/Data Comparator）



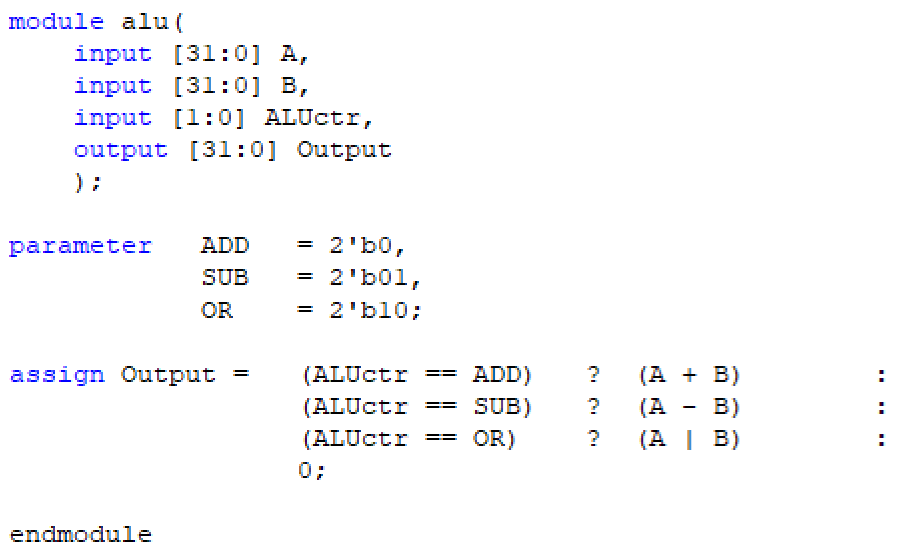
```
module comp(
    input [31:0] A,
    input [31:0] B,
    output eq,
    output gez,
    output gtz,
    output lez,
    output ltz
    );

assign eq   = (A == B);
assign gez  = (A == 0 || A[31] == 0);
assign gtz  = (A != 0 && A[31] == 0);
assign lez  = (A == 0 || A[31] == 1);
assign ltz  = (A != 0 && A[31] == 1);

endmodule
```

图 8 comp.v 模块

表 6 comp.v 规格

功能名称	方向	功能描述
A[31:0]	I	输入数据 A
B[31:0]	I	输入数据 B
eq	O	A == B
gez	O	A >= 0
gtz	O	A > 0
lez	O	A <= 0
ltz	O	A < 0

IV. ALU（算术逻辑单元/Arithmetic Logic Unit）



```
module alu(
    input [31:0] A,
    input [31:0] B,
    input [1:0] ALUctr,
    output [31:0] Output
    );

parameter   ADD   = 2'b0,
            SUB   = 2'b01,
            OR    = 2'b10;

assign Output =    (ALUctr == ADD)   ?  (A + B)          :
                   (ALUctr == SUB)   ?  (A - B)          :
                   (ALUctr == OR)    ?  (A | B)          :
                   0;

endmodule
```

图 9 alu.v 模块

表 7 alu.v 规格

功能名称	方向	功能描述
A[31:0]	I	输入数据 A
B[31:0]	I	输入数据 B
ALUctr[1:0]	I	ALU 控制信号 00：加法运算 01：减法运算 10：或运算 11：比较
Output[31:0]	O	输出结果 32 位

ALU is the most common module, this module is meant to perform arithmetic operations such as addition, subtraction, comparation, etc.

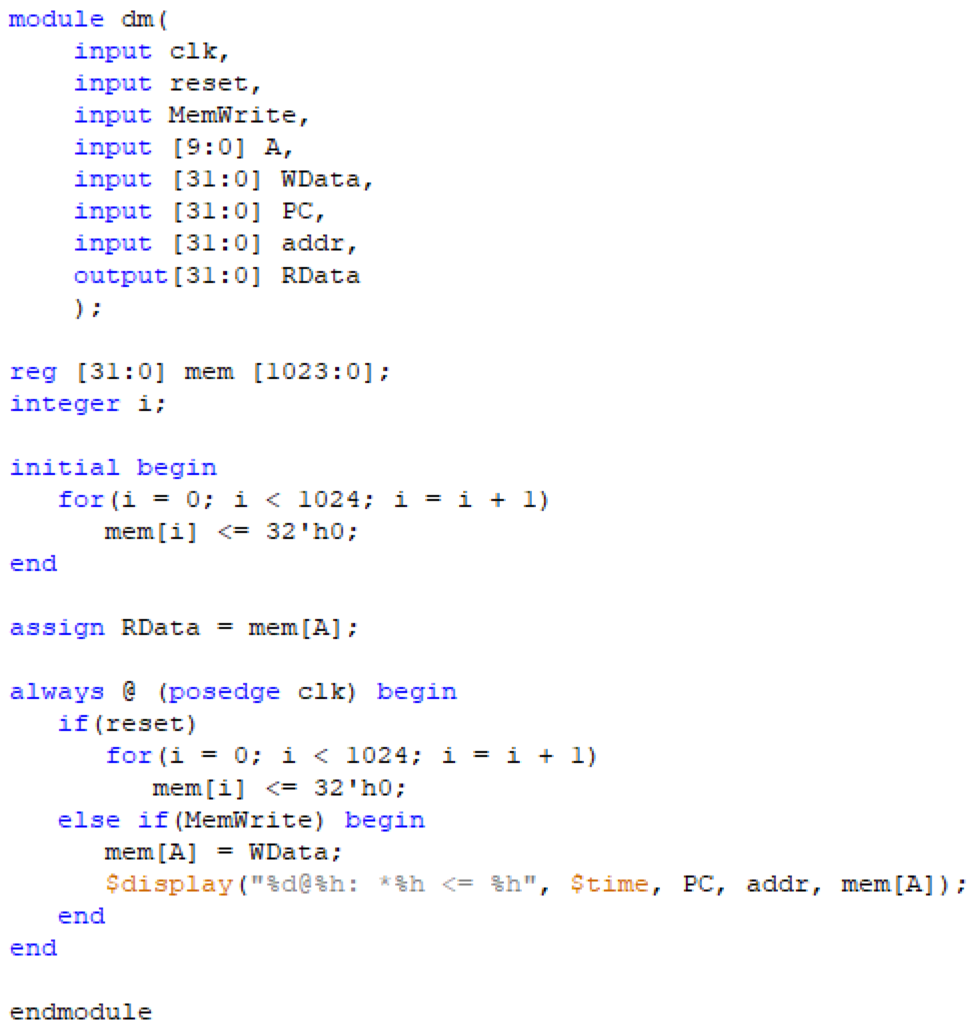
设计说明/Design Details:

- 提供 32 位加、减、或运算及大小比较功能（Provides 32-bit addition, subtraction, OR operation and size comparison）
- 可以不支持溢出（不检测溢出）（Overflow may not be supported (no overflow detected)）
- ALU 的输出不允许直接作为转发输入，只能是 ALUOutM (The Output of ALU module shouldn't be a direct input, it should be brought to the M level register first and come out as ALUOutM (or in my case, it's called AO_M))

模块流程/Design Workflow:

- According to the instruction that's through ALUctr input (this instruction is generated by the Controller module), the module will perform corresponding operation. The **ori** instruction will cause ALU to perform OR operation (10), the **subu** will cause ALU to perform subtraction operation (01), and the other instructions will simply cause ALU to perform addition operation.
- At the same time, both data inputs A and B will be checked, if the values are equal, then the Zero signal will be TRUE. The value of A and B will be decided based on what level it is currently on, and the signal of the forward module.

V. DM（数据存储器/Data Memory）



```
module dm(
    input clk,
    input reset,
    input MemWrite,
    input [9:0] A,
    input [31:0] WData,
    input [31:0] PC,
    input [31:0] addr,
    output[31:0] RData
    );

reg [31:0] mem [1023:0];
integer i;

initial begin
   for(i = 0; i < 1024; i = i + 1)
      mem[i] <= 32'h0;
end

assign RData = mem[A];

always @ (posedge clk) begin
   if(reset)
      for(i = 0; i < 1024; i = i + 1)
         mem[i] <= 32'h0;
   else if(MemWrite) begin
      mem[A] = WData;
      $display("%d@%h: *%h <= %h", $time, PC, addr, mem[A]);
   end
end

endmodule
```

图 10 dm.v 模块

表 8 dm.v 规格

功能名称	方向	功能描述
clk	I	时钟信号
reset	I	复位信号
MemWrite	I	读写控制信号
A [9:0]	I	MemAddr 地址，为 ALU 结果的[11:2]
WData [31:0]	I	输入数据 32 位
PC [31:0]	I	当前 PC 地址

Addr [31:0]	I	当前 DM 指定的地址
RData [31:0]	O	输出结果 32 位

The DM module is meant to save data generated from previous operations.

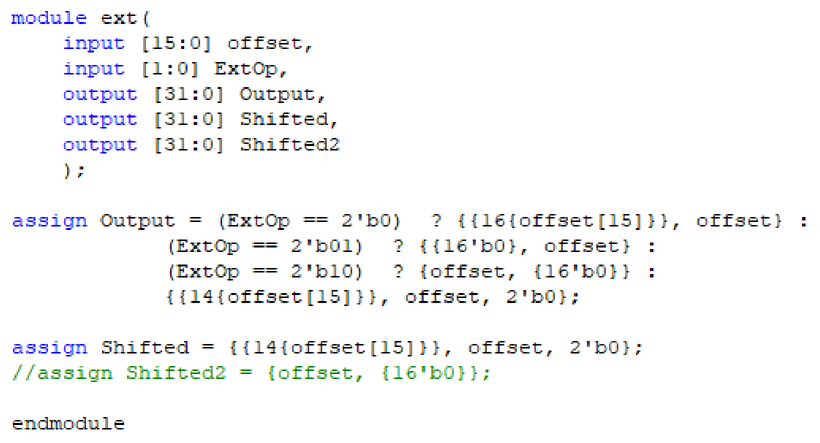
设计说明/Design Details:

- DM 容量为 4KB（32bit×1024 字）（The capacity of DM is 4 KB or 32bit x 1024 word）
- 每个时钟上升沿到来时若要写入数据(即写使能信号为 1 且非 reset 时)则输出写入的位置及写入的值，格式为 : (At clock posedge, always display some data information using the following command)
  $display("%d@%h: *%h <= %h", $time, PC, addr, mem[A]);

模块流程/Design Workflow:

- This is just a simple module to store data to the Data Memory, whether the CPU will write the data is decided by the MIPS instruction sent to MemWrite. If the instruction is **sw**, MemWrite will be TRUE, therefore storing the data to the DM.
- If MemWrite is FALSE, the module would simply read a data from the current pointed address.

VI. EXT（Extender）



```
module ext(
    input [15:0] offset,
    input [1:0] ExtOp,
    output [31:0] Output,
    output [31:0] Shifted,
    output [31:0] Shifted2
    );

assign Output = (ExtOp == 2'b0)  ? {{16{offset[15]}}, offset} :
            (ExtOp == 2'b01)  ? {{16'b0}, offset} :
            (ExtOp == 2'b10)  ? {offset, {16'b0}} :
            {{14{offset[15]}}, offset, 2'b0};

assign Shifted = {{14{offset[15]}}, offset, 2'b0};
//assign Shifted2 = {offset, {16'b0}};

endmodule
```

图 11 ext.v 模块

表 8 ext.v 规格

功能名称	方向	功能描述
offset [15:0]	I	将被 extended 的地址
extop	I	Extender 功能信号
Output [31:0]	O	基本 extender 输出
Shifted [31:0]	O	已被移的地址输出

设计说明/Design Details:

模块流程/Design Workflow:

- If extop is 0, thus FALSE, then the module will perform **signed-extend**, if it's 1 it will perform **zero-extend**, if it's 2 it will perform a simple **zero-shift**, meaning that the first 16bit data will be shifted to the upper half while the lower half will then be filled with 0s
- Shifted is used for beq destination jump address (sign_extend (offset||0^2))

VII. Controller（控制器）

```verilog
module ctrl(
    input [5:0] funct,
    input [5:0] opcode,
    output [1:0] RegDst,
    output ALUSrc,
    output [1:0]MemtoReg,
    output RegWrite,
    output MemWrite,
    output [1:0] nPC_sel,
    output [1:0] ExtOp,
    output [1:0] ALUctr
    );

reg [1:0] regdst;
reg alusrc;
reg [1:0] memtoreg;
reg regwrite;
reg memwrite;
reg [1:0] extop;
reg [1:0] npc_sel;
reg [1:0] aluctr;

parameter   addu_f   = 6'b100001,
         subu_f   = 6'b100011,
         ori      = 6'b001101,
         lw       = 6'b100011,
         sw       = 6'b101011,
         beq      = 6'b000100,
         lui      = 6'b001111,
         jal      = 6'b000011,
         j        = 6'b000010,
         jr_f     = 6'b001000;
         //nop    = 6'b0;
```

```verilog
always @ (opcode or funct) begin
   if (opcode == 6'b0) begin
      case (funct)
         addu_f: begin
            regdst = 2'b01;
            alusrc = 0;
            memtoreg = 2'b0;
            regwrite = 1;
            memwrite = 0;
            npc_sel = 2'b0;
            extop = 0;
            aluctr = 2'b0;
         end
         subu_f: begin
            regdst = 2'b01;
            alusrc = 0;
            memtoreg = 2'b0;
            regwrite = 1;
            memwrite = 0;
            npc_sel = 2'b0;
            extop = 0;
            aluctr = 2'b01;
         end
         jr_f: begin
            regdst = 2'b0;
            alusrc = 0;
            memtoreg = 2'b0;
            regwrite = 0;
            memwrite = 0;
            npc_sel = 2'b11;
            extop = 0;
            aluctr = 2'b0;
         end
```

```verilog
         default: begin
            regdst = 2'b0;
            alusrc = 0;
            memtoreg = 2'b0;
            regwrite = 0;
            memwrite = 0;
            npc_sel = 2'b0;
            extop = 0;
            aluctr = 2'b0;
         end
      endcase
   end
   else begin
      case (opcode)
         ori: begin
            regdst = 2'b0;
            alusrc = 1;
            memtoreg = 2'b0;
            regwrite = 1;
            memwrite = 0;
            npc_sel = 2'b0;
            extop = 1;
            aluctr = 2'b10;
         end
         lw: begin
            regdst = 2'b0;
            alusrc = 1;
            memtoreg = 2'b01;
            regwrite = 1;
            memwrite = 0;
            npc_sel = 2'b0;
            extop = 0;
            aluctr = 2'b0;
         end
         sw: begin
            regdst = 2'b0;
            alusrc = 1;
            memtoreg = 2'b0;
            regwrite = 0;
            memwrite = 1;
            npc_sel = 2'b0;
            extop = 0;
            aluctr = 2'b0;
         end
```

```verilog
beq:    begin
   regdst = 2'b0;
   alusrc = 0;
   memtoreg = 2'b0;
   regwrite = 0;
   memwrite = 0;
   npc_sel = 2'b01;
   extop = 2'b0;
   aluctr = 2'b0;
end
jal:    begin
   regdst = 2'b10;
   alusrc = 0;
   memtoreg = 2'b10;
   regwrite = 1;
   memwrite = 0;
   npc_sel = 2'b10;
   extop = 2'b11;
   aluctr = 2'b0;
end
j  :    begin
   regdst = 2'b0;
   alusrc = 0;
   memtoreg = 2'b0;
   regwrite = 0;
   memwrite = 0;
   npc_sel = 2'b10;
   extop = 2'b0;
   aluctr = 2'b0;
end
```

```verilog
      lui:    begin
         regdst = 2'b0;
         alusrc = 1;
         memtoreg = 2'b0;
         regwrite = 1;
         memwrite = 0;
         npc_sel = 2'b0;
         extop = 2'b10;
         aluctr = 2'b0;
      end
      default: begin
         regdst = 2'b0;
         alusrc = 0;
         memtoreg = 2'b0;
         regwrite = 0;
         memwrite = 0;
         npc_sel = 2'b0;
         extop = 2'b0;
         aluctr = 2'b0;
      end
   endcase
  end
end

assign RegDst     = regdst;
assign ALUSrc     = alusrc;
assign MemtoReg   = memtoreg;
assign RegWrite   = regwrite;
assign MemWrite   = memwrite;
assign nPC_sel    = npc_sel;
assign ExtOp     = extop;
assign ALUctr     = aluctr;

endmodule
```

图 12 控制器模块

表 9 控制器规格

功能名称	方向	功能描述
funct [5:0]	I	Function 6 位
opcode [5:0]	I	Opcode 6 位
RegDst	O	写地址控制
ALUSrc	O	控制 ALU
MemtoReg [1:0]	O	DM 读控制
RegWrite	O	GRF 读写控制
MemWrite	O	DM 写控制
nPC_sel [1:0]	O	跳转指令标志
ExtOp [1:0]	O	Ext 扩展控制
ALUctr[1:0]	O	ALU 运算操作控制

模块流程/Design Workflow:

- First, define all the instruction codes set as a static constant, in Verilog this kind of code has a type called **parameter**
- The input of this module is the 5-0 and 31-26 address bit of the instruction code generated from IFU. While most MIPS instruction opcode are located in the 31-26 bit of the instruction code, it works differently for **addu** and **subu**. The opcode for both **addu** and **subu** are 000000 but they got function code in the 5-0 bit of the instruction code. The complete code is listed in the table below.
- RegDst means that the register that will be used is the rd register, this is used by **addu** and **subu** while the other instructions will just use the rt register.
- MemtoReg literally means loading data from the memory to the register. Of course, this signal will be true if the instruction code is a loading instruction such as **lw** or **lui**
- RegWrite signal is to let the CPU write the data into a register. This will be used by instructions that cause changes to data so that it needs to be written (saved) in the register such as **lui, ori, lw, addu, subu, jal**
- MemWrite is a signal to let the CPU write the data into memory. This is only used by instructions that will store data such as **sw**

- The nPC_sel signal basically is just a bool to see if it's a branch instruction. In this project there are three branch instructions **beq, jal, jr, j**, therefore it's a 2 bit signal. This signal will then be used in the IFU module to decide whether to use PC+4 or PC+X or X for the address movements

表 10 控制器指令真值表

funct	100001	100011	n/a							001000	000000
opcode	000000	000000	001101	100011	101011	000100	001111	000011	000010	000000	000000
	addu	subu	ori	lw	sw	beq	lui	jal	j	jr	nop
RegDst	01	01	00	00	00	00	00	10	00	00	x
ALUSrc	0	0	1	1	1	0	0	0	0	0	x
MemtoReg[1:0]	00	00	00	01	xx	xx	10	11	00	00	xx
RegWrite	1	1	1	1	0	0	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0	0	0	0	0
nPC_sel	00	00	00	00	00	01	00	10	10	11	0
ExtOp	x	x	1	0	0	x	0	x	00	x	x
ALUctr[1:0]	Add	Subtract	Or	Add	Add	xx	xx	xx	xx	xx	xx

3. 数据通路（Data Path）

Since this is a pipeline-CPU, the process of the CPU will be divided into 5 big parts: F, D, E, M, W. These represents different operations:

3.1 F Level

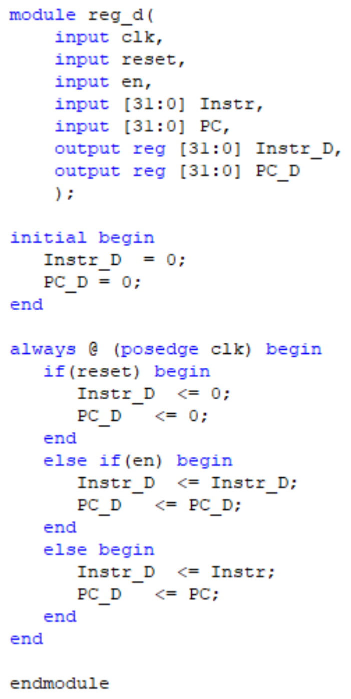
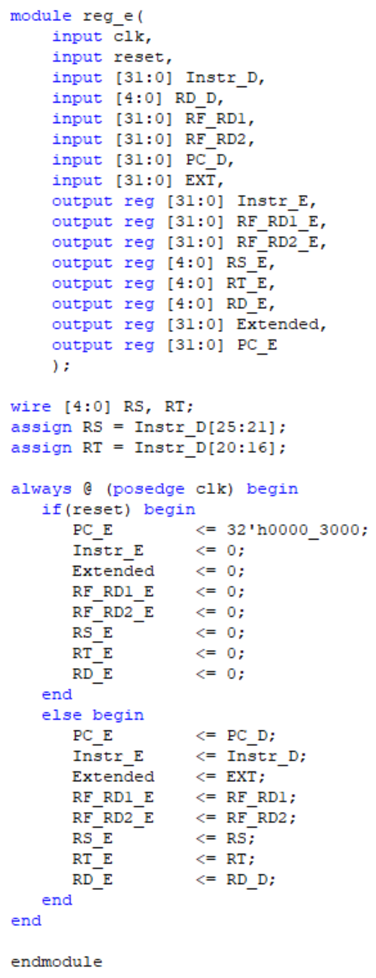

```
module reg_d(
    input clk,
    input reset,
    input en,
    input [31:0] Instr,
    input [31:0] PC,
    output reg [31:0] Instr_D,
    output reg [31:0] PC_D
    );

initial begin
   Instr_D  = 0;
   PC_D = 0;
end

always @ (posedge clk) begin
   if(reset) begin
      Instr_D  <= 0;
      PC_D   <= 0;
   end
   else if(en) begin
      Instr_D  <= Instr_D;
      PC_D   <= PC_D;
   end
   else begin
      Instr_D  <= Instr;
      PC_D   <= PC;
   end
end

endmodule
```

图 13 reg_d 代码

I. Included Modules

- PC : Decide whether the CPU should read the next instruction or not, if yes, which
- IM : Read and save command instructions
- Reg_D : Prepare register values for the D Level

3.2 D Level



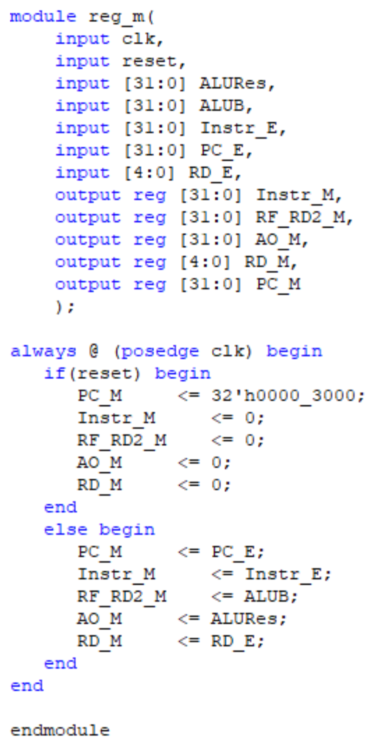
```
module reg_e(
    input clk,
    input reset,
    input [31:0] Instr_D,
    input [4:0] RD_D,
    input [31:0] RF_RD1,
    input [31:0] RF_RD2,
    input [31:0] PC_D,
    input [31:0] EXT,
    output reg [31:0] Instr_E,
    output reg [31:0] RF_RD1_E,
    output reg [31:0] RF_RD2_E,
    output reg [4:0] RS_E,
    output reg [4:0] RT_E,
    output reg [4:0] RD_E,
    output reg [31:0] Extended,
    output reg [31:0] PC_E
    );

wire [4:0] RS, RT;
assign RS = Instr_D[25:21];
assign RT = Instr_D[20:16];

always @ (posedge clk) begin
   if(reset) begin
      PC_E        <= 32'h0000_3000;
      Instr_E     <= 0;
      Extended    <= 0;
      RF_RD1_E    <= 0;
      RF_RD2_E    <= 0;
      RS_E        <= 0;
      RT_E        <= 0;
      RD_E        <= 0;
   end
   else begin
      PC_E        <= PC_D;
      Instr_E     <= Instr_D;
      Extended    <= EXT;
      RF_RD1_E    <= RF_RD1;
      RF_RD2_E    <= RF_RD2;
      RS_E        <= RS;
      RT_E        <= RT;
      RD_E        <= RD_D;
   end
end

endmodule
```

图 14 reg_d 代码

I. Included Modules

- CTRL : Get the controller signals to use in the D level
- NPC : Decide if it’s a jump instruction and decide the next instruction address
- GRF : Write data to register if needed so
- EXT : Extend data
- Mux : Multiplexer to decide which data will be compared for the **zero** signal
- ALU : Compare data if they equal and then set the **zero** signal
- Reg_E : Prepare register values for the E Level

3.3 E Level



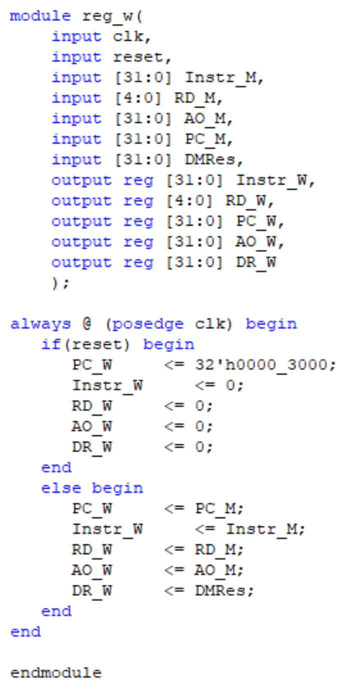
```
module reg_m(
    input clk,
    input reset,
    input [31:0] ALURes,
    input [31:0] ALUB,
    input [31:0] Instr_E,
    input [31:0] PC_E,
    input [4:0] RD_E,
    output reg [31:0] Instr_M,
    output reg [31:0] RF_RD2_M,
    output reg [31:0] AO_M,
    output reg [4:0] RD_M,
    output reg [31:0] PC_M
    );

always @ (posedge clk) begin
   if(reset) begin
      PC_M     <= 32'h0000_3000;
      Instr_M     <= 0;
      RF_RD2_M    <= 0;
      AO_M     <= 0;
      RD_M     <= 0;
   end
   else begin
      PC_M     <= PC_E;
      Instr_M     <= Instr_E;
      RF_RD2_M    <= ALUB;
      AO_M     <= ALURes;
      RD_M     <= RD_E;
   end
end

endmodule
```

图 15 reg_m 代码

I. Included Modules

- CTRL : Get the controller signals to use in the E level
- Mux : Multiplexer to decide which data will be used for arithmetic operations
- Mux2 : Multiplexer to decide which data will be used for the RegAddr for use when it reaches the GRF module in the D Level
- ALU : Execute arithmetic operations
- Reg_M : Prepare register values for the M Level

3.4 M Level



```
module reg_w(
    input clk,
    input reset,
    input [31:0] Instr_M,
    input [4:0] RD_M,
    input [31:0] AO_M,
    input [31:0] PC_M,
    input [31:0] DMRes,
    output reg [31:0] Instr_W,
    output reg [4:0] RD_W,
    output reg [31:0] PC_W,
    output reg [31:0] AO_W,
    output reg [31:0] DR_W
    );

always @ (posedge clk) begin
   if(reset) begin
      PC_W      <= 32'h0000_3000;
      Instr_W      <= 0;
      RD_W      <= 0;
      AO_W      <= 0;
      DR_W      <= 0;
   end
   else begin
      PC_W      <= PC_M;
      Instr_W      <= Instr_M;
      RD_W      <= RD_M;
      AO_W      <= AO_M;
      DR_W      <= DMRes;
   end
end

endmodule
```

图 16 reg_w 代码

I. Included Modules

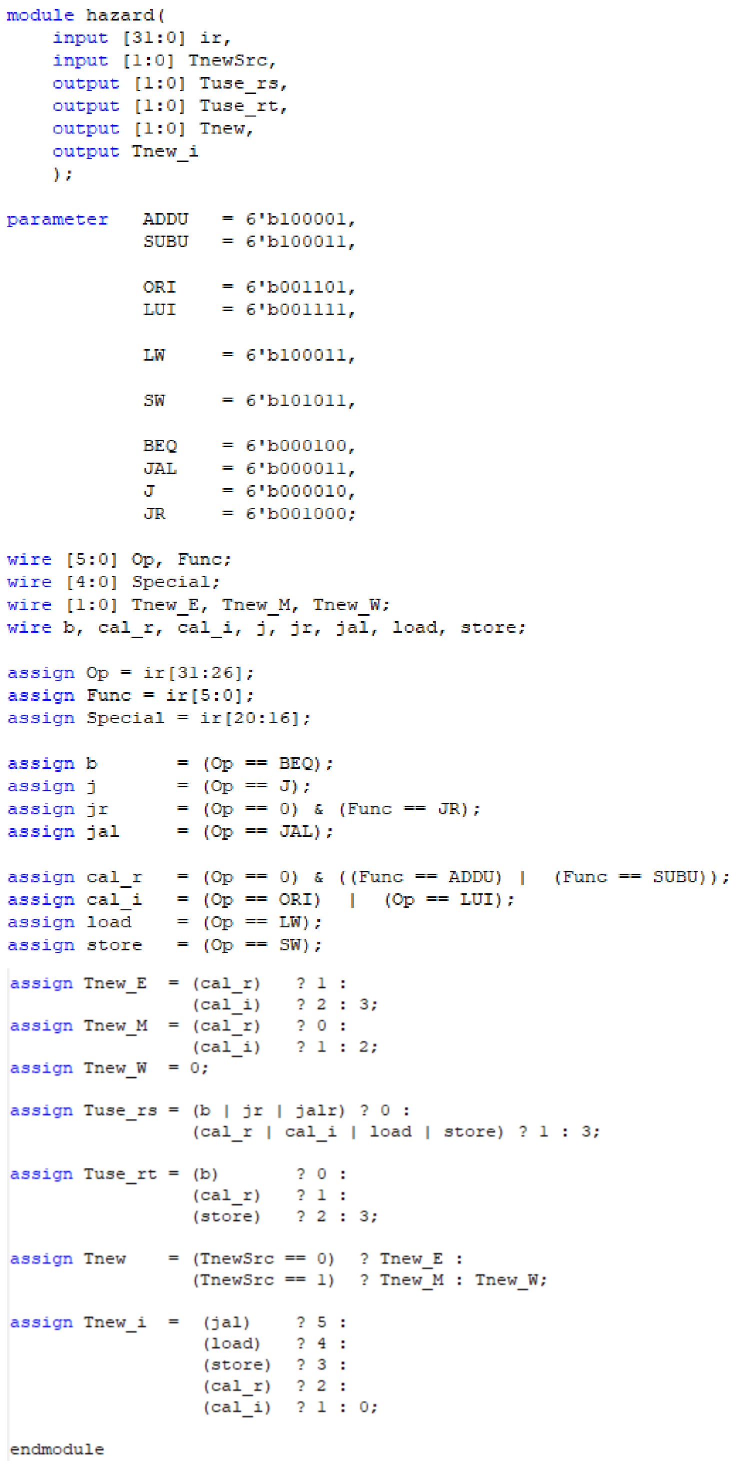
- CTRL : Get the controller signals to use in the M level
- DM : Writes data to memory if needed so
- Reg_W : Prepare register values for the W Level

3.5 W Level

I. Included Modules

- CTRL : Get the controller signals to use in the W level
- Mux : Multiplexer to decide which data will be used to be written in the register in the GRF module of the D Level

3.6 Hazard Control



```
module hazard(
    input [31:0] ir,
    input [1:0] TnewSrc,
    output [1:0] Tuse_rs,
    output [1:0] Tuse_rt,
    output [1:0] Tnew,
    output Tnew_i
    );

parameter   ADDU   = 6'b100001,
            SUBU   = 6'b100011,

            ORI    = 6'b001101,
            LUI    = 6'b001111,

            LW     = 6'b100011,

            SW     = 6'b101011,

            BEQ    = 6'b000100,
            JAL    = 6'b000011,
            J      = 6'b000010,
            JR     = 6'b001000;

wire [5:0] Op, Func;
wire [4:0] Special;
wire [1:0] Tnew_E, Tnew_M, Tnew_W;
wire b, cal_r, cal_i, j, jr, jal, load, store;

assign Op = ir[31:26];
assign Func = ir[5:0];
assign Special = ir[20:16];

assign b        = (Op == BEQ);
assign j        = (Op == J);
assign jr       = (Op == 0) & (Func == JR);
assign jal      = (Op == JAL);

assign cal_r    = (Op == 0) & ((Func == ADDU) |  (Func == SUBU));
assign cal_i    = (Op == ORI)   |   (Op == LUI);
assign load     = (Op == LW);
assign store    = (Op == SW);

assign Tnew_E  = (cal_r)    ? 1 :
                 (cal_i)    ? 2 : 3;
assign Tnew_M  = (cal_r)    ? 0 :
                 (cal_i)    ? 1 : 2;
assign Tnew_W  = 0;

assign Tuse_rs = (b | jr | jalr) ? 0 :
                 (cal_r | cal_i | load | store) ? 1 : 3;

assign Tuse_rt = (b)        ? 0 :
                 (cal_r)    ? 1 :
                 (store)    ? 2 : 3;

assign Tnew    = (TnewSrc == 0)  ? Tnew_E :
                 (TnewSrc == 1)  ? Tnew_M : Tnew_W;

assign Tnew_i  =  (jal)     ? 5 :
                  (load)    ? 4 :
                  (store)   ? 3 :
                  (cal_r)   ? 2 :
                  (cal_i)   ? 1 : 0;

endmodule
```

图 17 hazard.v 模块

首先，我们把各种指令根据它们对寄存器堆的读取行为进行分类，同一类指令的工作流程是相似的。

- Cal_r 类（R-R，寄存器与寄存器进行计算）：add, addu, subu 等。
- Cal_i 类（R_I，寄存器与常数进行计算）：addi, lui, ori 等。
- Beq 类（通过 CMP 比较器判断跳转）：beq, bne, bgez 等。
- Load 类（读取内存的值）：lw, lb, lbu, lh 等。
- Save 类（保存值到内存）：sw, sb, sh 等。
- J 类的四条指令各自都有微妙差别，推荐四条指令分开处理。

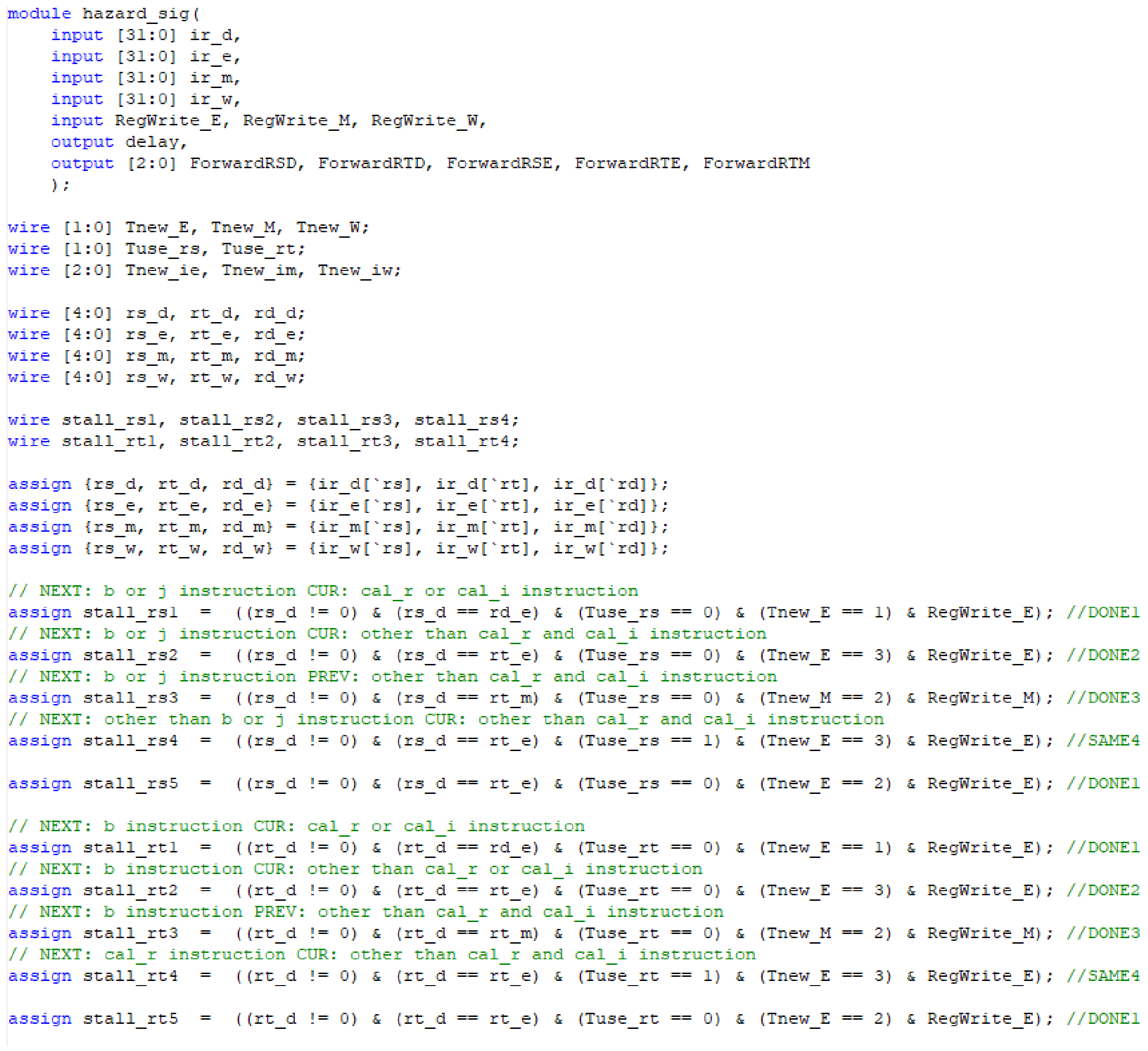
其中，只有 cal_r, cal_i, load 三类指令和 jal, jalr 会产生结果。

简单来说，就是遇到需要读取寄存器堆的指令时，暂停信号会变 TRUE，让 CPU 暂停以下，先不要执行下一个指令而先让整个 CPU 走到最后的 W 级，读取数据然后在继续执行。

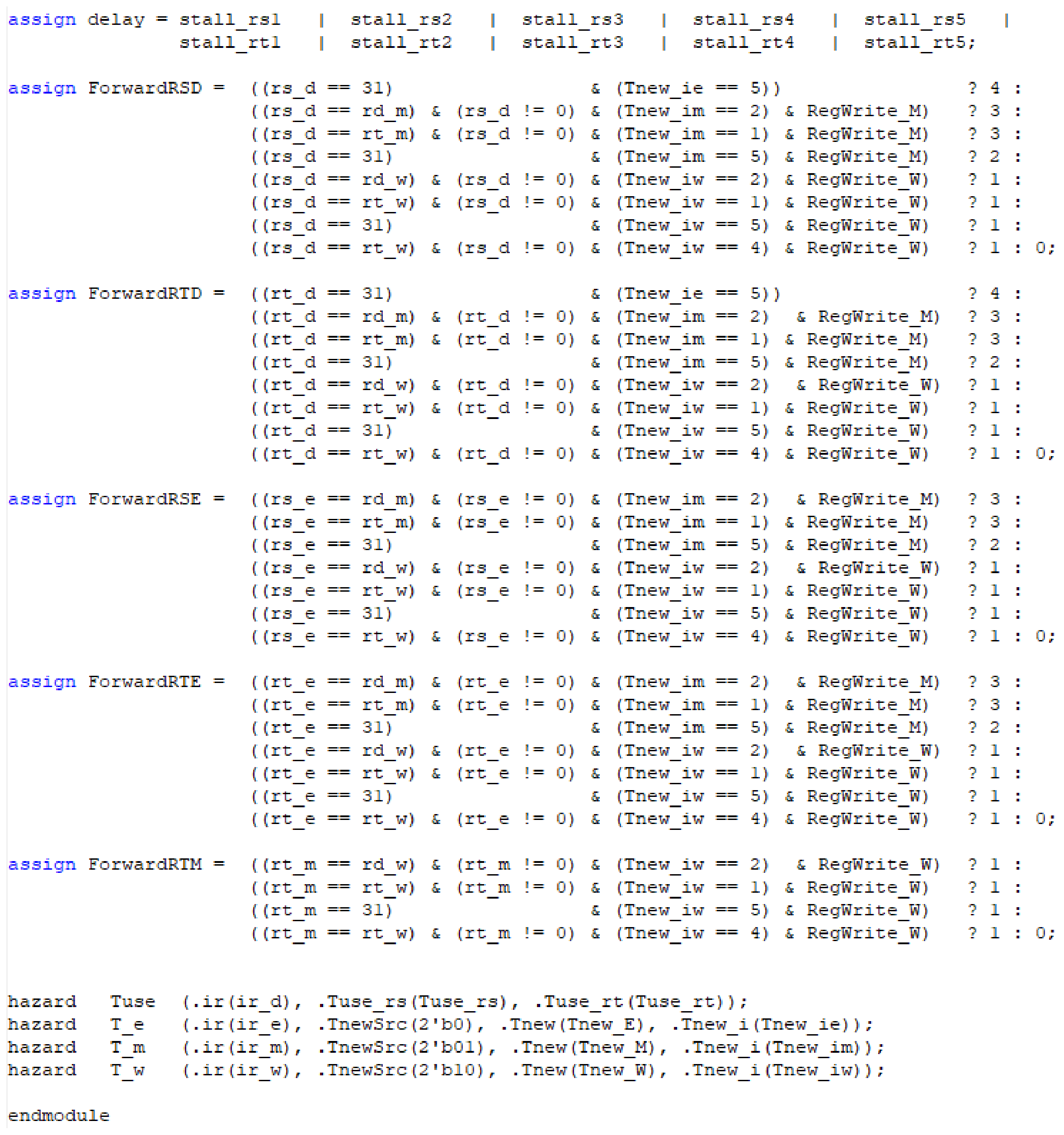
当判定需要暂停的时候，我们需要执行三个操作：

- 冻结 PC，不让 PC 的值改变
- 让 ID/EX 流水级寄存器清零，等于插入了一个 NOP 指令
- 冻结 IF/ID 流水级寄存器，不让它的值改变。

转发的原理，是后面的指令需要用到前面指令的运算结果，而后面指令在执行时，前面指令的运算结果还没有来得及写入寄存器堆。则此时，我们增加若干条数据通路，把前面指令的运算结果直接传递回来，达到不用暂停也能规避风险的目的。



```
module hazard_sig(
    input [31:0] ir_d,
    input [31:0] ir_e,
    input [31:0] ir_m,
    input [31:0] ir_w,
    input RegWrite_E, RegWrite_M, RegWrite_W,
    output delay,
    output [2:0] ForwardRSD, ForwardRTD, ForwardRSE, ForwardRTE, ForwardRTM
    );

wire [1:0] Tnew_E, Tnew_M, Tnew_W;
wire [1:0] Tuse_rs, Tuse_rt;
wire [2:0] Tnew_ie, Tnew_im, Tnew_iw;

wire [4:0] rs_d, rt_d, rd_d;
wire [4:0] rs_e, rt_e, rd_e;
wire [4:0] rs_m, rt_m, rd_m;
wire [4:0] rs_w, rt_w, rd_w;

wire stall_rs1, stall_rs2, stall_rs3, stall_rs4;
wire stall_rt1, stall_rt2, stall_rt3, stall_rt4;

assign {rs_d, rt_d, rd_d} = {ir_d[`rs], ir_d[`rt], ir_d[`rd]};
assign {rs_e, rt_e, rd_e} = {ir_e[`rs], ir_e[`rt], ir_e[`rd]};
assign {rs_m, rt_m, rd_m} = {ir_m[`rs], ir_m[`rt], ir_m[`rd]};
assign {rs_w, rt_w, rd_w} = {ir_w[`rs], ir_w[`rt], ir_w[`rd]};

// NEXT: b or j instruction CUR: cal_r or cal_i instruction
assign stall_rs1  =  ((rs_d != 0) & (rs_d == rd_e) & (Tuse_rs == 0) & (Tnew_E == 1) & RegWrite_E); //DONE1
// NEXT: b or j instruction CUR: other than cal_r and cal_i instruction
assign stall_rs2  =  ((rs_d != 0) & (rs_d == rt_e) & (Tuse_rs == 0) & (Tnew_E == 3) & RegWrite_E); //DONE2
// NEXT: b or j instruction PREV: other than cal_r and cal_i instruction
assign stall_rs3  =  ((rs_d != 0) & (rs_d == rt_m) & (Tuse_rs == 0) & (Tnew_M == 2) & RegWrite_M); //DONE3
// NEXT: other than b or j instruction CUR: other than cal_r and cal_i instruction
assign stall_rs4  =  ((rs_d != 0) & (rs_d == rt_e) & (Tuse_rs == 1) & (Tnew_E == 3) & RegWrite_E); //SAME4

assign stall_rs5  =  ((rs_d != 0) & (rs_d == rt_e) & (Tuse_rs == 0) & (Tnew_E == 2) & RegWrite_E); //DONE1

// NEXT: b instruction CUR: cal_r or cal_i instruction
assign stall_rt1  =  ((rt_d != 0) & (rt_d == rd_e) & (Tuse_rt == 0) & (Tnew_E == 1) & RegWrite_E); //DONE1
// NEXT: b instruction CUR: other than cal_r or cal_i instruction
assign stall_rt2  =  ((rt_d != 0) & (rt_d == rt_e) & (Tuse_rt == 0) & (Tnew_E == 3) & RegWrite_E); //DONE2
// NEXT: b instruction PREV: other than cal_r and cal_i instruction
assign stall_rt3  =  ((rt_d != 0) & (rt_d == rt_m) & (Tuse_rt == 0) & (Tnew_M == 2) & RegWrite_M); //DONE3
// NEXT: cal_r instruction CUR: other than cal_r and cal_i instruction
assign stall_rt4  =  ((rt_d != 0) & (rt_d == rt_e) & (Tuse_rt == 1) & (Tnew_E == 3) & RegWrite_E); //SAME4

assign stall_rt5  =  ((rt_d != 0) & (rt_d == rt_e) & (Tuse_rt == 0) & (Tnew_E == 2) & RegWrite_E); //DONE1
```



```
assign delay = stall_rs1   |  stall_rs2   |  stall_rs3   |  stall_rs4   |  stall_rs5   |
               stall_rt1   |  stall_rt2   |  stall_rt3   |  stall_rt4   |  stall_rt5;

assign ForwardRSD =  ((rs_d == 31)                    & (Tnew_ie == 5))                ? 4 :
                     ((rs_d == rd_m)  &  (rs_d != 0)  & (Tnew_im == 2)  & RegWrite_M)   ? 3 :
                     ((rs_d == rt_m)  &  (rs_d != 0)  & (Tnew_im == 1)  & RegWrite_M)   ? 3 :
                     ((rs_d == 31)                    & (Tnew_im == 5)  & RegWrite_M)   ? 2 :
                     ((rs_d == rd_w)  &  (rs_d != 0)  & (Tnew_iw == 2)  & RegWrite_W)   ? 1 :
                     ((rs_d == rt_w)  &  (rs_d != 0)  & (Tnew_iw == 1)  & RegWrite_W)   ? 1 :
                     ((rs_d == 31)                    & (Tnew_iw == 5)  & RegWrite_W)   ? 1 :
                     ((rs_d == rt_w)  &  (rs_d != 0)  & (Tnew_iw == 4)  & RegWrite_W)   ? 1 : 0;

assign ForwardRTD =  ((rt_d == 31)                    & (Tnew_ie == 5))                ? 4 :
                     ((rt_d == rd_m)  &  (rt_d != 0)  & (Tnew_im == 2)   & RegWrite_M)  ? 3 :
                     ((rt_d == rt_m)  &  (rt_d != 0)  & (Tnew_im == 1)  & RegWrite_M)   ? 3 :
                     ((rt_d == 31)                    & (Tnew_im == 5)  & RegWrite_M)   ? 2 :
                     ((rt_d == rd_w)  &  (rt_d != 0)  & (Tnew_iw == 2)   & RegWrite_W)  ? 1 :
                     ((rt_d == rt_w)  &  (rt_d != 0)  & (Tnew_iw == 1)  & RegWrite_W)   ? 1 :
                     ((rt_d == 31)                    & (Tnew_iw == 5)  & RegWrite_W)   ? 1 :
                     ((rt_d == rt_w)  &  (rt_d != 0)  & (Tnew_iw == 4)  & RegWrite_W)   ? 1 : 0;

assign ForwardRSE =  ((rs_e == rd_m)  &  (rs_e != 0)  & (Tnew_im == 2)   & RegWrite_M)  ? 3 :
                     ((rs_e == rt_m)  &  (rs_e != 0)  & (Tnew_im == 1)  & RegWrite_M)   ? 3 :
                     ((rs_e == 31)                    & (Tnew_im == 5)  & RegWrite_M)   ? 2 :
                     ((rs_e == rd_w)  &  (rs_e != 0)  & (Tnew_iw == 2)   & RegWrite_W)  ? 1 :
                     ((rs_e == rt_w)  &  (rs_e != 0)  & (Tnew_iw == 1)  & RegWrite_W)   ? 1 :
                     ((rs_e == 31)                    & (Tnew_iw == 5)  & RegWrite_W)   ? 1 :
                     ((rs_e == rt_w)  &  (rs_e != 0)  & (Tnew_iw == 4)  & RegWrite_W)   ? 1 : 0;

assign ForwardRTE =  ((rt_e == rd_m)  &  (rt_e != 0)  & (Tnew_im == 2)   & RegWrite_M)  ? 3 :
                     ((rt_e == rt_m)  &  (rt_e != 0)  & (Tnew_im == 1)  & RegWrite_M)   ? 3 :
                     ((rt_e == 31)                    & (Tnew_im == 5)  & RegWrite_M)   ? 2 :
                     ((rt_e == rd_w)  &  (rt_e != 0)  & (Tnew_iw == 2)   & RegWrite_W)  ? 1 :
                     ((rt_e == rt_w)  &  (rt_e != 0)  & (Tnew_iw == 1)  & RegWrite_W)   ? 1 :
                     ((rt_e == 31)                    & (Tnew_iw == 5)  & RegWrite_W)   ? 1 :
                     ((rt_e == rt_w)  &  (rt_e != 0)  & (Tnew_iw == 4)  & RegWrite_W)   ? 1 : 0;

assign ForwardRTM =  ((rt_m == rd_w)  &  (rt_m != 0)  & (Tnew_iw == 2)   & RegWrite_W)  ? 1 :
                     ((rt_m == rt_w)  &  (rt_m != 0)  & (Tnew_iw == 1)  & RegWrite_W)   ? 1 :
                     ((rt_m == 31)                    & (Tnew_iw == 5)  & RegWrite_W)   ? 1 :
                     ((rt_m == rt_w)  &  (rt_m != 0)  & (Tnew_iw == 4)  & RegWrite_W)   ? 1 : 0;


hazard   Tuse   (.ir(ir_d), .Tuse_rs(Tuse_rs), .Tuse_rt(Tuse_rt));
hazard   T_e    (.ir(ir_e), .TnewSrc(2'b0), .Tnew(Tnew_E), .Tnew_i(Tnew_ie));
hazard   T_m    (.ir(ir_m), .TnewSrc(2'b01), .Tnew(Tnew_M), .Tnew_i(Tnew_im));
hazard   T_w    (.ir(ir_w), .TnewSrc(2'b10), .Tnew(Tnew_W), .Tnew_i(Tnew_iw));

endmodule
```

图 18 hazard_sig.v 模块

具体实现过程：

- 把指令分成几个分类：cal_r, cal_i, b, j, jr, jal, jalr, load, store
- cal_r 指令都是用 rd 而 cal_i 和 load 使用 rt，判断冲突时，需要针对正确的指令和寄存器，不能大致判断（并且寄存器当时并非为 0，说明确实发生有效冲突）
- 定义 Tuse_rs 和 Tuse_rt，两个分别是代表 rs 和 rt 寄存器是否要被新的指令使用。对于 Tuse_rs，若新的指令是 cal_r | cal_i | load | store 那就是会使用 rs

对于 Tuse_rt，若新的指令是 cal_r 信号为 1，store 信号为 2

- 然后定义几个暂停的信号，本人的程序中有 8 种：

  NEXT: D Level, CUR: E Level, PREV: M Level

  对于 rs 寄存器的情况

  - Stall_rs1: NEXT: b | j   CUR: cal_r
  - Stall_rs2: NEXT: b | j   CUR: ~(cal_r | cal_i)
  - Stall_rs3: NEXT: b | j   PREV: ~(cal_r | cal_i)
  - Stall_rs4: NEXT: ~(b | j)   CUR: ~(cal_r | cal_i)
  - Stall_rs5: NEXT: b | j   CUR: cal_i

  对于 rt 寄存器的情况

  - Stall_rt1: NEXT: b   CUR: cal_r
  - Stall_rt2: NEXT: b   CUR: ~(cal_r | cal_i)
  - Stall_rt3: NEXT: b   PREV: ~(cal_r | cal_i)
  - Stall_rt4: NEXT: cal_r   CUR: ~(cal_r | cal_i)
  - Stall_rt5: NEXT: b   CUR: cal_i

- 之所以把 Stall_rs1 和 Stall_rs5 以及 Stall_rt1 和 Stall_rt5 分开，是因为 cal_r 和 cal_i 使用不同寄存器的缘故
- 发现以上 10 种情况中的任何一个都要发出暂停的信号，意味着需要先让那些需要把数据写到寄存器上的指令把数据写完再继续执行下一个指令，以免发生冲突。

转发系统使用 Mux 部件来进行的。

3.7 The Data Path Details

部件	输入	lw	sw	addu	subu	ori	beq	j	jal	jr	lui
PC											
IM		PC	PC	PC	PC	PC	PC	PC	PC	PC	PC
IR_D		IM	IM	IM	IM	IM	IM	IM	IM	IM	IM
PC_D							PC4	PC4	PC4	PC4	PC4
NPC	PC						PC_D	PC_D	PC_D	PC_D	PC_D
	Instr						IR_D	IR_D	IR_D	IR_D	IR_D

GRF	Reg1	IR_D [rs]	IR_D [rs]	IR_D [rs]	IR_D [rs]	IR_D [rs]	IR_D [rs]			IR_D [rs]	IR_D [rs]
	Reg2			IR_D [rt]	IR_D [rt]		IR_D [rt]				
EXT		IR_D [16]	IR_D [16]			IR_D [16]	IR_D [16]				IR_D [16]
CMP	D1						RD1				
	D2						RD2				
Instr_E		IR_D	IR_D	IR_D	IR_D	IR_D	IR_D	IR_D	IR_D	IR_D	IR_D
PC_E							PC_D	PC_D	PC_D	PC_D	PC_D
RS_E		RD1	RD1	RD1	RD1	RD1	RD1				
RT_E			RD2	RD2	RD2	RD2	RD2				
EXTE		EXT	EXT			EXT	EXT				EXT
ALU	A	RS_E	RS_E	RS_E	RS_E	RS_E	RS_E				
	B	EXTE	EXTE	RT_E	RT_E	EXTE	RT_E				
Instr_M		IR_E	IR_E	IR_E	IR_E	IR_E	IR_E	IR_E	IR_E	IR_E	IR_E
PC_M						PC_E	PC_E	PC_E	PC_E	PC_E	PC_E
AO_M		ALU	ALU	ALU	ALU	ALU					
RT_M			RT_E								
DM	A	AO_M	AO_M								
	WD		RD_M								
Instr_W		IR_M		IR_M	IR_M	IR_M					IR_M
PC_W							PC_W	PC_W	PC_W	PC_W	PC_W
AO_W				AO_M	AO_M	AO_M					
DR_W		DM									
GRF	RegA	IR_W [rt]		IR_W [rd]	IR_W [rd]	IR_W [rt]					IR_W [rt]
	RegD	DR_W		AO_W	AO_W	AO_W					

4. 测试程序说明 (Testing Program)

测试指令集：

```
.data
    testarr: .space 8
    # addu, subu, ori, lui, lw, sw, beq, jal, jr, j, nop
.text
ori $gp, $zero, 0x0000
ori $sp, $zero, 0x0000
ori $at, $zero, 0x3456
addu $at, $at, $at
lw $at, 0x0004
sw $at, 0x0004
lui $v0, 0x7878
subu $v1, $v0, $at
lui $a1, 0x1234
ori $a0, $zero, 0x0005
nop
lui $v0, 0x8723
nop
ori $a3, $v1, 0x0404
nop
lui $t0, 0x7777
addu $at, $at, $v0
lw $at, 0x0000
ori $t0, $t0, 0xffff
addu $t1, $at, $v1
subu $zero, $zero, $t0
ori $zero, $zero, 0x1100
lw $t2, 0x0000
addu $t2, $a3, $t2
ori $at, $a3, 0x1010
sw $at, 0x0000
ori $t0, $zero, 0x0000
ori $t1, $zero, 0x0001
ori $t2, $zero, 0x0001
addu $t0, $t0, $t2
subu $zero, $0, $t0

nop
lui $t0, 0x1234
ori $t1, $t0, 0x0001
lui $t0, 0x0000
ori $t1, $t0, 0x0000
nop
nop

addu $t2, $t1, 3
subu $t3, $t1, $t2
nop

sw $t3, testarr($zero)
beq $t3, $t2, any
lw $t3, testarr($zero)
nop

any: ori $t0, $t0, 0x1010

jal procedure
lui $t0, 0x5678
jal end

procedure:
ori $t1, $t0, 0x0009
jr $ra

end:
```

机器码：

```
341c0000
341d0000
34013456
00210821
8c010004
ac010004
3c027878
00411823
3c051234
34040005
00000000
3c028723
00000000
34670404
00000000
3c087777
00220821
8c010000
3508ffff
00234821
00080023
34001100
8c0a0000
00ea5021
34e11010
ac010000
34080000
34090001
340a0001
010a4021
00080023
00000000
3c081234
```

```
35090001
3c080000
35090000
00000000
00000000
3c010000
34210003
01215021
012a5823
00000000
ac0b0000
116a0002
8c0b0000
00000000
35081010
0c000c33
3c085678
0c000c35
35090009
03e00008
```

5. 思考题

1. 在本实验中你遇到了哪些不同指令组合产生的冲突？你又是如何解决的？相应的测试样例是什么样的？请有条理的罗列出来。(非常重要)

   以下是测试过程中考虑到有可能发生冲突的指令组合。

No.	测试类型	前序指令	冲突位置	冲突寄存器	测试用例
1	R-M-RS	addu	MEM	rs	addu $t0, $t0, $t2 subu $zero, $0, $t0
2	R-M-RT	addu	MEM	rt	addu $t2, $t1, 3 subu $t3, $t1, $t2
3	R-W-RS	addu	MEM	rs	addu $at, $at, $v0

					~~ori $t0, $t0, 0xffff~~ addu $t1, $at, $v1
4	R-W-RT	addu	MEM	rt	addu $at, $at, $at ~~lw $at, 0x0004~~ ~~sw $at, 0x0004~~ ~~lui $v0, 0x7878~~ subu $v1, $v0, $at
5	I-M-RS	ori	MEM	rs	ori $at, $zero, 0x3456 addu $at, $at, $at
6	I-M-RT	lui	MEM	rt	lui $t0, 0x7777 ori $t0, $t0, 0xffff
7	I-W-RS	ori	MEM	rs	ori $zero, $zero, 0x1100 ~~addu $t2, $a3, $a2~~ ori $t0, $zero, 0x0000
8	I-W-RT	lui	MEM	rt	lui $t0, 0x7777 ~~ori $t0, $t0, 0xffff~~ subu $zero, $zero, $t0
9	~~LD-M-RS~~	~~lw~~	~~MEM~~	~~rs~~	
10	LD-M-RT	lw	MEM	rt	lw $t2, 0x0000 addu $t2, $a3, $t2
11	LD-W-RS	lw	MEM	rs	lw $at, $v0 ~~ori $t0, $t0, 0xffff~~ addu $t1, $at, $v1
12	LD-W-RT	lw	MEM	rt	lw $at, 0x0004 ~~sw $at, 0x0004~~ ~~lui $v0, 0x7878~~ subu $v1, $v0, $at

解决方案：

- 把指令分成几个分类：cal_r, cal_i, b, j, jr, jal, jalr, load, store
- 定义 Tuse_rs 和 Tuse_rt，两个分别是代表 rs 和 rt 寄存器是否要被新的指令使用。

  对于 Tuse_rs，若新的指令是 cal_r | cal_i | load | store 那就是会使用 rs

  对于 Tuse_rt，若新的指令是 cal_r 信号为 1，store 信号为 2
- 然后定义几个暂停的信号，本人的程序中有 8 种：

NEXT: D Level, CUR: E Level, PREV: M Level

对于 rs 寄存器的情况

- Stall_rs1: NEXT: b | j   CUR: cal_r
- Stall_rs2: NEXT: b | j   CUR: ~(cal_r | cal_i)
- Stall_rs3: NEXT: b | j   PREV: ~(cal_r | cal_i)
- Stall_rs4: NEXT: ~(b | j)   CUR: ~(cal_r | cal_i)
- Stall_rs5: NEXT: b | j   CUR: cal_i

对于 rt 寄存器的情况

- Stall_rt1: NEXT: b   CUR: cal_r
- Stall_rt2: NEXT: b   CUR: ~(cal_r | cal_i)
- Stall_rt3: NEXT: b   PREV: ~(cal_r | cal_i)
- Stall_rt4: NEXT: cal_r   CUR: ~(cal_r | cal_i)
- Stall_rt5: NEXT: b   CUR: cal_i

- 之所以把 Stall_rs1 和 Stall_rs5 以及 Stall_rt1 和 Stall_rt5 分开，是因为 cal_r 和 cal_i 使用不同寄存器的缘故
- 发现以上 10 种情况中的任何一个都要发出暂停的信号，意味着需要先让那些需要把数据写到寄存器上的指令把数据写完再继续执行下一个指令，以免发生冲突。