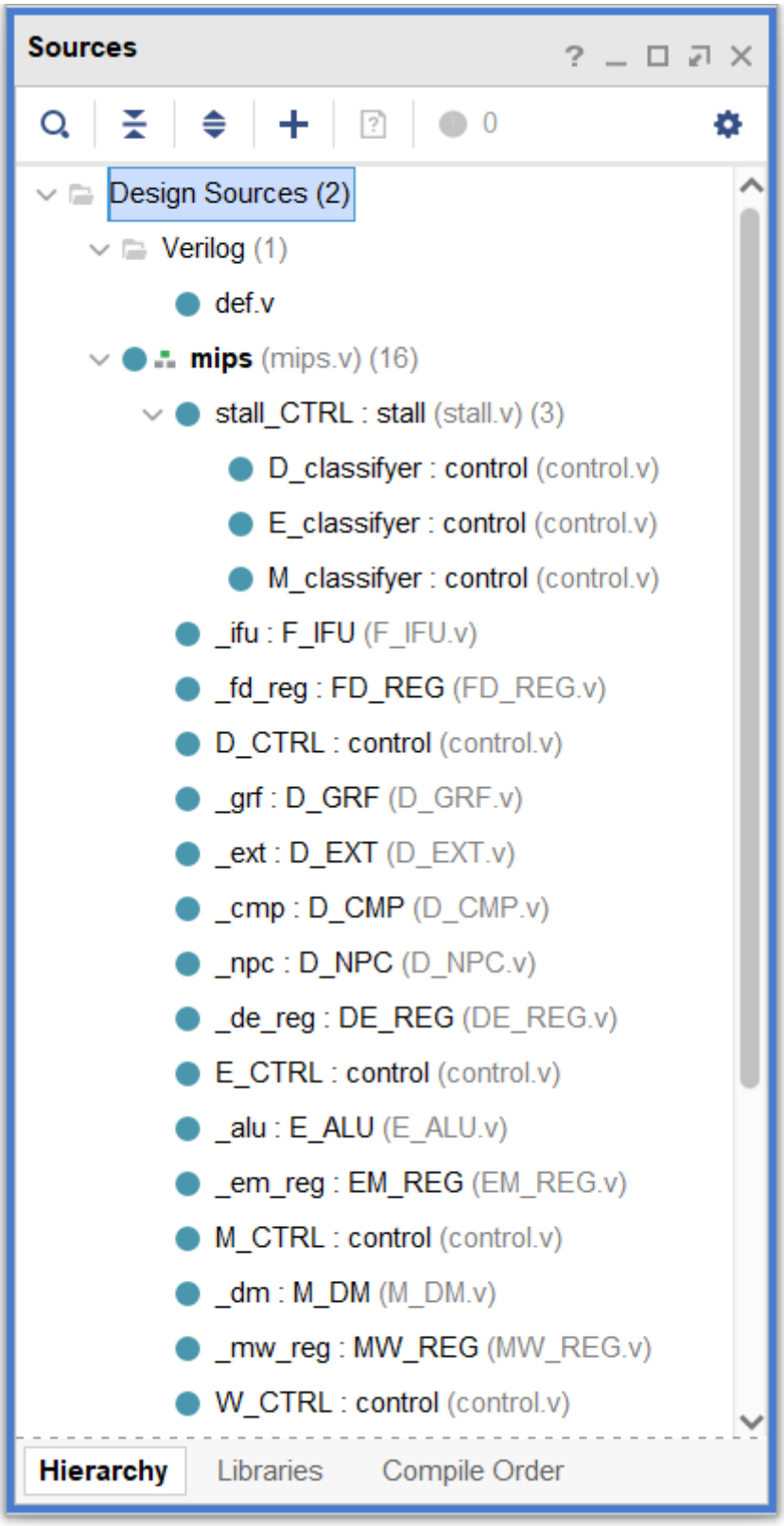
计算机组成原理实验报告—流水线CPU设计

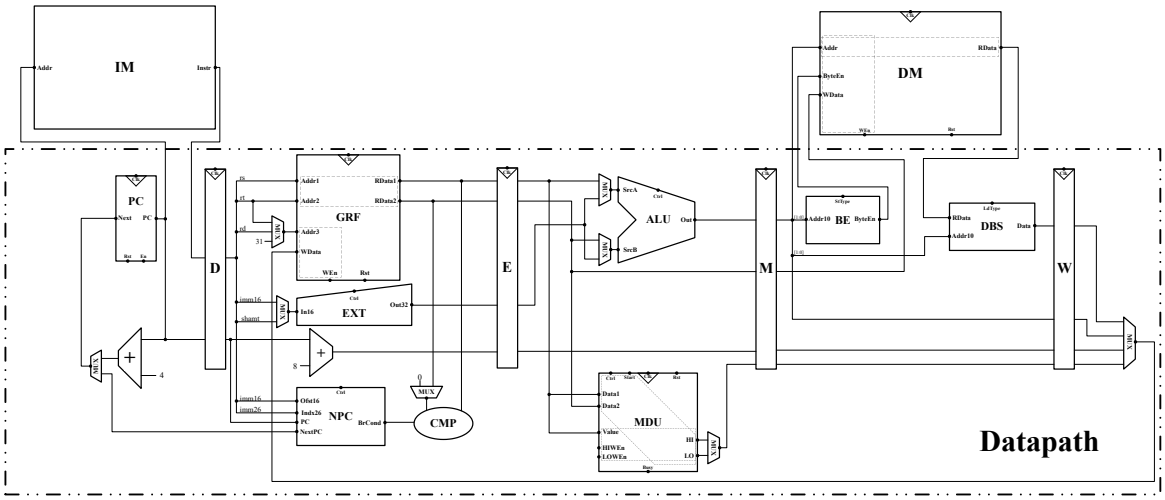
总体设计概述

要求实现的指令集为`MIPS-lite2`，即**addu，subu，ori，lw，sw，beq， jal，jr，lui，nop**

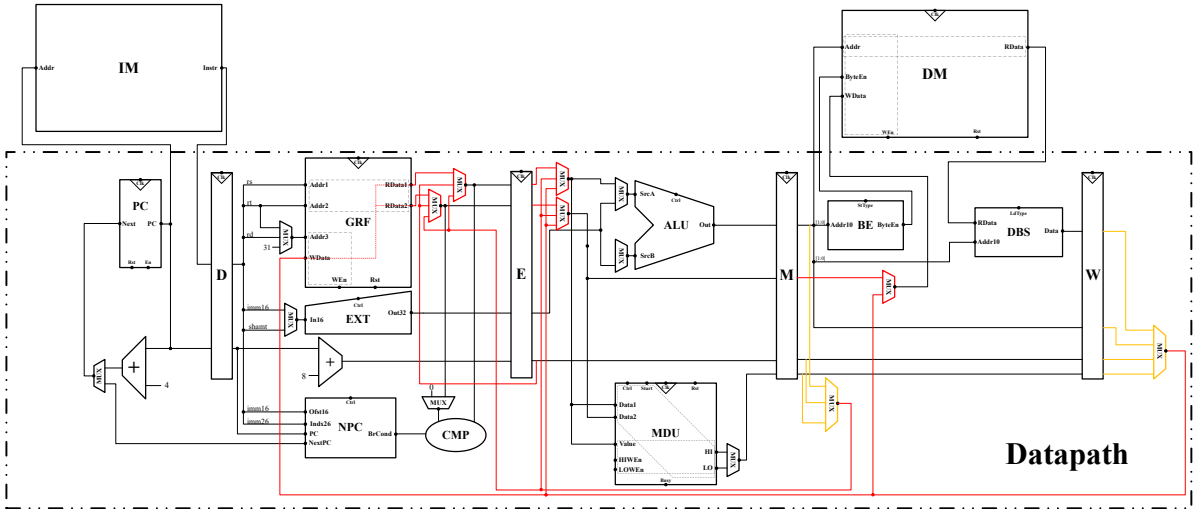
整体结构如下图所示



具体的数据通路设计参照下图（没有添加乘除块`MDU`单元）



添加完转发单元之后的图如下



命名规范

- 对于元件的文件命名，均为`流水线层级_元件英文简称`，例如`D_GRF.v`，`E_ALU.v`等，实例化时命名为`_小写英文名`，例如`_alu`，`_grf`等
- 对于流水线寄存器文件命名为`两边的流水线层级_REG`，例如`FD_REG.v`，`DE_REG.v`，实例化时命名为`_小写英文名`，例如`_fd_reg`
- 每一级的控制信号和临时的`wire`均以本级的名称开头，如`E_ALUOp`，`M_DMOp`等
- 在流水线中参与流水的信息遵从以下约定（以D级为例）
  - `PC`和`Instr`命名以流水线层级开头，如`D_PC`，`D_Instr`
  - 寄存器地址分别为`D_rs_addr`，`D_rt_addr`，读出数据为`D_rs_data`，`D_rt_data`
  - 转发得到的寄存器数据（直接读取也视为一种转发）记作`D_FWD_rs_data`，`D_FWD_rt_data`
  - 即将写入的寄存器地址为`E_GRFA3`，即将写入的数据记作`E_GRFWD`，选择信号为`E_GRFWDSel`

下面将按照流水线层级逐一分析各个单元

F级(Fetch/取指令)

- 本级没有转发，阻塞时需要取消`PC`写使能
- 本级的输入有来自D级的`NPC`，本级的输出是`F_PC`和`F_Instr`，两者需要参与流水线流水

IFU（取指单元）

信号名称	方向	功能描述
NPC[31:0]	输入	待写入PC的指令地址
clk	输入	时钟信号
reset	输入	同步复位信号
PC_WrEn	输入	PC的写使能
PC	输出	当前指令地址
Instr[31:0]	输出	32位的指令值

D级(Decode/译码)

- 本级需要处理来自E, M, W级的转发，其中W级为寄存器内部转发，另外两个分别是`D_FWD_rs`,`D_FWD_rt`，在`CMP`和`NPC`中需要用
- 本级的输入是来自F级的`PC`和`Instr`，输出是`D_rs_data`，`D_rt_data`，`D_ext32`，`D_PC`和`D_Instr`，还有输出到F级的`NPC`
- 本级元件较多，比较复杂

FD_REG（F/D级流水线寄存器）

信号名称	方向	功能描述
clk	输入	时钟信号
reset	输入	同步复位信号
flush	输入	寄存器刷新信号（阻塞时使用）
F_PC	输入	F级PC的指令地址
F_Instr[31:0]	输入	时钟信号
D_PC	输出	D级PC的指令地址
D_Instr[31:0]	输出	32位的指令值

D_GRF（寄存器堆）

端口说明

信号名称	方向	功能描述
A1[4:0]	输入	5位地址输入信号，将其储存的数据读出到RD1
A2[4:0]	输入	5位地址输入信号，将其储存的数据读出到RD2
A3[4:0]	输入	5位地址输入信号，将其作为写入数据的目标寄存器
RD1[31:0]	输出	输出A1指定的寄存器中的32位数据
RD2[31:0]	输出	输出A2指定的寄存器中的32位数据
WD[31:0]	输入	32位数据输入信号
clk	输入	时钟信号
reset	输入	异步复位信号，将32个寄存器中的数据清零；1：复位；0：无效

- 这次删去了`GRFWrEn`写使能信号，因为如果我们不写寄存器，可以把`GRFA3`设为0，就相当于不写寄存器了

控制信号说明

1. `D_GRFA3`

直接给出待写入寄存器的地址，弃用了在P4中利用`GRFA3Sel`进行选择的设计，这是因为P5采用分布式译码，每一级都需要`GRFA3`的信息，因此在`CTRL`里面直接集成了

2. `D_GRFWDSel`

控制信号值	功能
`WDSel_dmrd`	选择写入寄存器的数据来自DM
`WDSel_aluans`	选择写入寄存器的数据来自ALU运算结果
`WDSel_pc8`	选择写入寄存器的数据为**当前流水线层级中的PC+8**

D_EXT（位扩展）

将16位二进制数进行零扩展或符号扩展到32位

控制信号说明

控制信号值	功能
`EXT_unsign`	零扩展
`EXT_sign`	符号扩展

D_CMP（比较器）

把原来ALU中比较值是否相等的运算移到了CMP里面，去指导`beq`这一类型的指令是否跳转

控制信号目前只有`CMP_beq`，未来可以扩展

端口说明

信号名称	方向	功能描述
rs[31:0]	输入	**处理完转发后** `$rs` 寄存器的值
rt[31:0]	输入	**处理完转发后** `$rt` 寄存器的值
CMPOp[2:0]	输入	控制信号
b_jump	输出	指示是否跳转，输入 `NPC`，未来可以添加 `bltzal` 指令

D_NPC（次地址计算单元）

把 `beq` 是否执行的判断交给了 `CMP`，直接根据输入信号 `jump` 判断是否跳转

其实 `NPC` 横跨了F级和D级两级，因为同时会输入 `F_PC` 和 `D_PC`，前者正常跳转 `F_PC+4` 用，后者则用于流水 `PC` 值，后面转发 `PC+8` 的时候用

这次我们弃用了P4中直接输出 `PC+4` 的设计，转而让 `PC` 信号参与流水，在需要转发时计算 `PC+8`

端口说明

信号名称	方向	功能描述
F_PC[31:0]	输入	32位输入当前F级地址
D_PC[31:0]	输入	32位输入当前D级地址
b_jump	输入	指示b类型指令是否跳转
NPCOp[1:0]	输入	控制信号
RSS[31:0]	输入	**处理完转发后** `$rs` 寄存器保存的32位地址
NPC[31:0]	输出	32位输出次地址

控制信号说明

控制信号值	功能
`NPC_pc4`	`NPC=PC+4`
`NPC_b`	执行 `beq` 等b类指令
`NPC_j_jal`	执行 `j`，`jal` 指令
`NPC_jalr_jr`	执行 `jalr`，`jr` 指令

E级(Execute/执行)

DE_REG（D/E级流水线寄存器）

- 输入 `D_PC,D_Instr,D_ext32`，此外上一级的 `$rs` 和 `$rt` 的值也要参与流水，即 `D_FWD_rs,D_FWD_rt` 需要参与流水，**这是由于指令序列 `sw, nop, addu` 的存在，`sw` 在M级需要使用 `$rt` 的数据，但是在E级不会再进行转发（因为在D级已经转发过了），因此需要让正确的 `$rt` 值参与流水**
- 输出 `E_PC,E_Instr,E_ext32,E_rs_data,E_rt_data`，`ALU` 需要这些信息

信号名称	方向	功能描述
clk	输入	时钟信号
reset	输入	同步复位信号
flush	输入	寄存器刷新信号（阻塞时使用）
D_PC[31:0]	输入	D级PC的指令地址
D_Instr[31:0]	输入	32位的指令值
D_ext32[31:0]	输出	16位立即数经`EXT`扩展的结果
D_rs_data[31:0]	输出	32位的寄存器数据
D_rt_data[31:0]	输出	32位的寄存器数据
E_PC[31:0]	输入	E级PC的指令地址
E_Instr[31:0]	输入	32位的指令值
E_ext32[31:0]	输出	16位立即数经`EXT`扩展的结果
E_rs_data[31:0]	输出	32位的寄存器数据
E_rt_data[31:0]	输出	32位的寄存器数据

E_ALU（算术逻辑单元）

- 相比于P4，ALU做了很大的变动，添加了`ALUASel`信号选择A运算数的来源，这是为了便于扩展`sll`和`sllv`类指令的原因取消了`shamt`信号，`shamt`信号从`ALUBSel`中选择进入`ALU`中

端口说明

信号名称	方向	功能描述
A[31:0]	输入	32位输入运算数A
B[31:0]	输入	32位输入运算数B
ALUOp[4:0]	输入	控制信号
C[31:0]	输出	32位输出运算结果

控制信号说明

1. ALUOp

控制信号值	功能
`ALU_add`	执行加法运算
`ALU_sub`	执行减法运算
`ALU_or`	执行逻辑或运算
`ALU_lui`	执行`lui`指令

2. ALUASel

控制信号值	功能
`ASel_rt`	对于`sll`和`sllv`等移位指令，选择`$rt`的值
`ASel_rs`	对于其他大部分运算指令，采用`$rs`的值

3. ALUBSel

控制信号值	功能
`BSel_rt`	选择**处理完转发后**`$rt`寄存器中的值进行运算
`BSel_imm`	选择立即数进行运算
`BSel_shamt`	使用`{27'b0, E_ALUshamt}`得到32为扩展移位数
`BSel_rs`	考虑到`sllv`指令要求可变的位移数，这里可以选择`{27'b0, E_FWD_rs_data[4:0]}`，即`$rs`寄存器中的数据作为移位数

M级(Memory/储存)

- 输入`E_PC,E_Instr`，此外上一级的ALUAns参与流水，即`E_ALUAns,E_ext32`需要参与流水，**这是因为`ALUAns`是待写入或读取的内存地址，另外，上一级的rt值需要参与流水**，因此还需要输入`E_FWD_rt`，**这是因为`sw`指令会向内存中写入`$rt`的数据**
- 输出`M_PC,M_Instr,M_ALUAns,M_DMRD`

EM_REG（E/M级流水线寄存器）

信号名称	方向	功能描述
clk	输入	时钟信号
reset	输入	同步复位信号
flush	输入	寄存器刷新信号（阻塞时使用）
E_PC[31:0]	输入	E级PC的指令地址
E_Instr[31:0]	输入	32位的指令值
E_ext32[31:0]	输入	16位立即数经`EXT`扩展的结果
E_rt_data[31:0]	输入	32位的寄存器数据
E_ALUAns[31:0]	输入	32位的ALU运算结果
M_PC[31:0]	输出	M级PC的指令地址
M_Instr[31:0]	输出	32位的指令值
M_ext32[31:0]	输出	16位立即数经`EXT`扩展的结果
M_ALUAns[31:0]	输出	32位的ALU运算结果
M_rt_data[31:0]	输出	32位的寄存器数据

M_DM（数据储存器）

- `DM`与P4基本相同

端口说明

信号名称	方向	功能描述
Addr[31:0]	输入	待操作的内存地址
WD[31:0]	输入	待写入内存的值
clk	输入	时钟信号
reset	输入	异步复位信号
DMWrEn	输入	写使能信号；1：写入有效；0：写入无效
DMOp[2:0]	输入	控制信号
RD[31:0]	输出	输入地址指向的内存中储存的值

控制信号说明

控制信号值	功能
`DM_w`	对应`lw`和`sw`指令，写入或读取整个字
`DM_h`	（保留）对应`lh`和`sh`指令，写入或读取半字
`DM_b`	（保留）对应`lb`和`sb`指令，写入或读取整个字
`DM_hu`	（保留）对应`lhu`指令
`DM_bu`	（保留）对应`lbu`指令

W级(Write/回写)

- W级事实上与D级重合了，但是仍然需要处理向E,M级的转发

MW_REG（M/W级流水线寄存器）

信号名称	方向	功能描述
clk	输入	时钟信号
reset	输入	同步复位信号
flush	输入	寄存器刷新信号（阻塞时使用）
M_PC[31:0]	输入	M级PC的指令地址
M_Instr[31:0]	输入	32位的指令值
M_DMRD[31:0]	输入	从内存中读取的值
M_ALUAns[31:0]	输入	32位的ALU运算结果
W_PC[31:0]	输出	W级PC的指令地址
W_Instr[31:0]	输出	32位的指令值
W_DMRD[31:0]	输出	从内存中读取的值
W_ALUAns[31:0]	输出	32位的ALU运算结果

数据通路分析

指令	opcode	funct	NPCOp	GRFA3	GRFWDSel	EXTOp	ALUBSel	ALUOp	DMWrEn	DMOp
addu	000000	100001	`NPC_pc4`	`rd`	`WDSel_aluans`	X	`BSel_rt`	`ALU_add`	0	X
subu	000000	100011	`NPC_pc4`	`rd`	`WDSel_aluans`	X	`BSel_rt`	`ALU_sub`	0	X
ori	001101	X	`NPC_pc4`	`rt`	`WDSel_aluans`	`EXT_unsign`	`BSel_imm`	`ALU_or`	0	X
lw	100011	X	`NPC_pc4`	`rt`	`WDSel_dmrd`	`EXT_sign`	`BSel_imm`	`ALU_add`	0	`DM_w`
sw	101011	X	`NPC_pc4`	X	`WDSel_dmrd`	`EXT_sign`	`BSel_imm`	`ALU_add`	1	`DM_w`
beq	000100	X	`NPC_b`	X	X	X	X	X	0	X
lui	001111	X	`NPC_pc4`	`rt`	`WDSel_aluans`	X	`BSel_imm`	`ALU_lui`	0	X
j	000010	X	`NPC_j_jal`	X	X	X	X	X	X	X
jr	000000	001001	`NPC_jr_jalr`	X	X	X	X	X	X	X
jal	000011	X	`NPC_jr_jalr`	`rs`	`WDSel_pc8`	X	X	X	X	X

CTRL在每一级都会进行译码，即采用分布式译码

D级(Decode/译码)

```
// D级Decode
control D_CTRL(
    .Instr(D_Instr),

    .rs(D_rs_addr),
    .rt(D_rt_addr),
    .imm16(D_imm16),
    .imm26(D_imm26),
    .CMPOp(D_CMPOp),
    .NPCOp(D_NPCOp),
    .EXTOp(D_EXTOp)
);
```

E级(Execute/执行)

```
// E级Execute
control E_CTRL(
    .Instr(E_Instr),

    .rs(E_rs_addr),
    .rt(E_rt_addr),
    .shamt(E_ALUshamt),

    .ALUOp(E_ALUOp),
    .ALUASel(E_ALUASel),
    .ALUBSel(E_ALUBSel),
    .GRFA3(E_GRFA3),
    .GRFWDSel(E_GRFWDSel)
);
```

M级(Memory/储存)

```
control M_CTRL(
    .Instr(M_Instr),

    .rt(M_rt_addr),
    .DMOp(M_DMOp),
    .DMWrEn(M_DMWrEn),
    .GRFA3(M_GRFA3),
    .GRFWDSel(M_GRFWDSel)
);
```

W级(Write/回写)

```
control W_CTRL(
    .Instr(W_Instr),

    .GRFA3(W_GRFA3),
    .GRFWDSel(W_GRFWDSel)
);
```

冲突处理方法

转发

对于转发，我采用的是暴力转发的方法，如果不阻塞就意味着一定能够在使用该寄存器的值之前获得最新的且正确的值，那么之前转发的错误的值不用去管它，最后总能得到一个正确的值去覆盖原先错误的值

W-D级转发在寄存器内部实现

```
assign D_FWD_rs_data = (D_rs_addr == 0) ? 0 :
                       (D_rs_addr == E_GRFA3) ? E_GRFWD :
                       (D_rs_addr == M_GRFA3) ? M_GRFWD :
                       D_rs_data;

assign D_FWD_rt_data = (D_rt_addr == 0) ? 0 :
                       (D_rt_addr == E_GRFA3) ? E_GRFWD :
                       (D_rt_addr == M_GRFA3) ? M_GRFWD :
                       D_rt_data;
```

```
assign E_FWD_rs_data = (E_rs_addr == 0) ? 0 :
                       (E_rs_addr == M_GRFA3) ? M_GRFWD :
                       (E_rs_addr == W_GRFA3) ? W_GRFWD :
                       E_rs_data;

assign E_FWD_rt_data = (E_rt_addr == 0) ? 0 :
                       (E_rt_addr == M_GRFA3) ? M_GRFWD :
                       (E_rt_addr == W_GRFA3) ? W_GRFWD :
                       E_rt_data;
```

```
assign M_FWD_rt_data = (M_rt_addr == 0) ? 0 :
                       (M_rt_addr == W_GRFA3) ? W_GRFWD :
                       M_rt_data;
```

阻塞

对于阻塞的处理，我直接采用教程中$Tuse$和$Tnew$进行判断的方法，设计了`stall_CTRL`模块，专门负责处理阻塞时流水线寄存器的`flush`和`WrEn`信号

我的设计只在D级进行阻塞，阻塞控制器接受D，E，M级的指令输入，处理分析指令类别，并给他们赋上不同的$Tuse$和$Tnew$的值，然后用组合逻辑判断，如果$Tuse < Tnew$就直接阻塞D级，知道$Tuse = Tnew$时再继续执行

```
wire E_stall_rs = (E_GRFA3 == D_rs_addr && D_rs_addr != 0) && (E_Tnew > D_Tuse_rs);
wire E_stall_rt = (E_GRFA3 == D_rt_addr && D_rt_addr != 0) && (E_Tnew > D_Tuse_rt);

wire M_stall_rs = (M_GRFA3 == D_rs_addr && D_rs_addr != 0) && (M_Tnew > D_Tuse_rs);
wire M_stall_rt = (M_GRFA3 == D_rt_addr && D_rt_addr != 0) && (M_Tnew > D_Tuse_rt);

assign Stall = E_stall_rs | E_stall_rt | M_stall_rs | M_stall_rt;

assign FD_REG_WrEn = !Stall;
assign DE_REG_WrEn = 1'b1;
assign EM_REG_WrEn = 1'b1;
assign MW_REG_WrEn = 1'b1;

assign PC_WrEn = !Stall;

assign FD_REG_Flush = 1'b0;
assign DE_REG_Flush = Stall;
assign EM_REG_Flush = 1'b0;
assign MW_REG_Flush = 1'b0;
```

调试方法与辅助工具

> 我的测试方法是利用随机生成数据进行大范围测试，对于随机数据无法覆盖的点，通过手动构造特殊样例进行测试

自动对拍程序的文件目录树如下

```
+---P5Judge
|   |   code.txt
|   |   log.txt
|   |   mips_code.asm
|   |   new_code_generate.cpp
|   |   new_code_generate.exe
|   |   Process.cpp
|   |   Process.exe
|   |   Running.cpp
|   |   Running.exe
|   |   src_tb.out
|   |   src_v.out
|   |   std_tb.out
|   |   std_v.out
|   |
|   +---src
|   |       mips_tb.v
|   |       # 这里是待测试的代码目录
|   |
|   +---std
|   |       mips_tb.v
|   |       # 这里是标准代码目录
|   |
```

运行环境Windows，将`zip`解压到任意文件夹，运行`Running.exe`即可进行对拍测试

数据生成器`new_code_generate.exe`采用的是公共跳转块的逻辑+随机生成+小范围寄存器增多冲突行为，详见思考题中的回答，在调大数据组数后，覆盖性测试转发可得65分，阻塞可得70分，对于无法处理的情况采用手工测试

由于生成器生成的代码中有`la`伪指令，因此需要CPU支持`addi`指令

`Process.exe`对读入的数据进行处理，主要是直接对结果排序，避免出现写内存和寄存器的先后问题

生成器、比较器的代码均已上传

思考题

流水线冒险

1. 在采用本节所述的控制冒险处理方式下，PC 的值应当如何被更新？请从数据通路和控制信号两方面进行说明。

```
`timescale 1ns / 1ps
`include "def.v"

module D_NPC(
    input [2:0] NPCOp,
    input [31:0] D_PC,
    input [31:0] F_PC,
    input b_jump,
    input [25:0] imm26,
    input [15:0] imm16,
    input [31:0] rs,
    output [31:0] NPC
    );

    // 关于延迟槽
    // j，b型指令均有延迟槽，因为其为有条件跳转，需要在D级才能得知跳转结果
    // 具体实现代码时，一般来说F_PC + 4 = D_PC
    // 因此我们对于b型指令，写D_PC + 4 + {{14{imm16[15]}}, imm16, 2'b00}已经考虑了延迟槽
    // 对于j型指令，我们写F_PC[31:28]和D_PC[31:28]效果相同
    // 对于default情况，我们必然写F_PC + 4，即顺序执行下一条指令
    // F_PC为F级当前PC值，D_PC为D级当前PC值，也就是说，我们的NPC模块同时从F级和D级接收信息

    assign NPC = (NPCOp == `NPC_jr_jalr) ? rs :
                 (NPCOp == `NPC_b && b_jump) ? D_PC + 4 + {{14{imm16[15]}}, imm16, 2'b00} :
                 (NPCOp == `NPC_j_jal) ? {D_PC[31:28], imm26, 2'b00} :
                 F_PC + 4;

endmodule

```

如图所示，直接利用`NPCOp`控制信号维护下一周期的PC值，其中有四种可能性即`branch`型，`jr/jalr`型，`j/jal`型和`PC+4`型

2. 对于 jal 等需要将指令地址写入寄存器的指令，为什么需要回写 PC+8 ？

需要考虑延迟槽，在跳转指令后面后面有nop或者一条数据无关的指令。

数据冒险的分析

为什么所有的供给者都是存储了上一级传来的各种数据的**流水级寄存器**，而不是由 ALU 或者 DM 等部件来提供数据?

如果从非流水线寄存器部件转发，那么某一级的总延迟就会增加，从而根据木桶效应，时钟周期就会增加，总效率反而降低，得不偿失。

AT 法处理流水线数据冒险

1. “转发（旁路）机制的构造”中的 Thinking 1-4；
    1. 如果不采用已经转发过的数据，而采用上一级中的原始数据，会出现怎样的问题？试列举指令序列说明这个问题。

        计算过程或存储过程中会用到还未更改过的寄存器值，从而出错。例如：

        ```
        ori $1, $0, 1
        nop
        nop
        nop
        nop
        lw $1, 0($0)
        nop
        sw $1, 4($0)
        ```

        这时，当指令`sw`达到M级时，`lw`已经执行完毕，不会转发，但是我们有没有保留E级已经转发过的数据，这样 `sw` 指令就会把 1 存到 DM 中。

    2. 我们为什么要对GPR采用内部转发机制？如果不采用内部转发机制，我们要怎样才能解决这种情况下的转发需求呢？

        GPR采用内部转发机制相当于MW流水线寄存器的值直接实时反馈到 GPR 的输出端，从而当前处于 D 级的指令可以直接用到对应寄存器的值，即W级到D级的转发。

        如果不采用内部转发机制，需要额外建立从MW流水线寄存器转发到 D 级的数据通路。

    3. 为什么0号寄存器需要特殊处理？

        因为指令可以对 0 号寄存器赋值，只是不会造成实际作用，但是转发过程中如果不特判就默认 0 号寄存器的值被更改了，从而造成错误。

    4. 什么是“最新产生的数据”？

        根据指令的执行顺序，越后执行的指令更改的寄存器的值越新，按照 DE、EM、MW 的顺序，越靠前所转发出的信息越新，因此优先级更高。

2. 在 AT 方法讨论转发条件的时候，只提到了“供给者需求者的A相同，且不为 0”，但在 CPU 写入 GRF 的时候，是有一个 WE 信号来控制是否要写入的。为何在 AT 方法中不需要特判 we 呢？为了**用且仅用** A 和 T 完成转发，在翻译出 A 的时候，要结合 WE 做什么操作呢？
    - AT 法要求：只要**当前位点的读取寄存器地址和某转发输入来源的写入寄存器地址相等且不为0**

        那么既然是要写入的，WE 必然为 1，因此不用特判，如果不需要写入，我们零待写入地址`GRFA3`为0，向0号寄存器里写入数据相当于不写

    - 如果WE是0，我们把`GRFA3`设为`5'b00000`即可

在线测试相关说明

在本实验中你遇到了哪些不同指令类型组合产生的冲突？你又是如何解决的？相应的测试样例是什么样的？

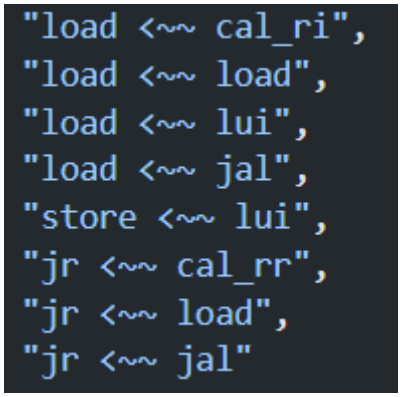
如果你是手动构造的样例，请说明构造策略，说明你的测试程序如何保证**覆盖**了所有需要测试的情况；如果你是**完全随机**生成的测试样例，请思考完全随机的测试程序有何不足之处；如果你在生成测试样例时采用了**特殊的策略**，比如构造连续数据冒险序列，请你描述一下你使用的策略如何**结合了随机性**达到强测的效果。

此思考题请同学们结合自己测试 CPU 使用的具体手段，按照自己的实际情况进行回答。

- 我的测试方法是利用随机生成数据进行大范围测试，对于随机数据无法覆盖的点，通过手动构造特殊样例进行测试
- 对于随机数据生成，我采用了对于指令进行分类，进行测试，生成的指令仅包括 `$1`, `$2`, `$3`, `$31`, `$4`, `$5` 这些寄存器，以增加相邻指令寄存器复用的概率，提高冲突发生的概率
- 生成器采用教程中所述的公共跳转区的做法，即

```
# 刚开始用ori和lui对寄存器赋初始值

jal subtest1
back1:

jal subtest2
back2:

endtest:
beq $0, $0, endtest

subtest1:
# 这里生成长度约为10~15条的随机指令
# 为了避免死循环，beq指令均跳转到endsubtest处
endsubtest1:
la $ra, back1
jr $ra

subtest2:
# ...
endsubtest2:
la $ra, back2
jr $ra
```

- 根据最终覆盖性测试的结果，生成的指令有以下部分未被覆盖，大部分是load指令，这是因为load指令的地址处理难以解决的原因



对于这些指令，我们采用手动构造测试样例进行测试即可

- 总结来看，随机数据生成加上一定的策略可以进行处理大量的冒险类型，但是对于无法覆盖的，就需要手动构造样例
- 手动构造样例时可以考虑转发的来源和接受源，以全面测试
  - D级需求: E->D(如序列 `jal-addu`), M->D(如序列 `jal-nop-addu`)(W->D隐藏于GRF的内部转发中)
  - E级需求: M->E(如序列 `addu-addu`), W->E(如序列 `addu-nop-addu`)
  - M级需求: W->M(如序列 `addu-sw`)