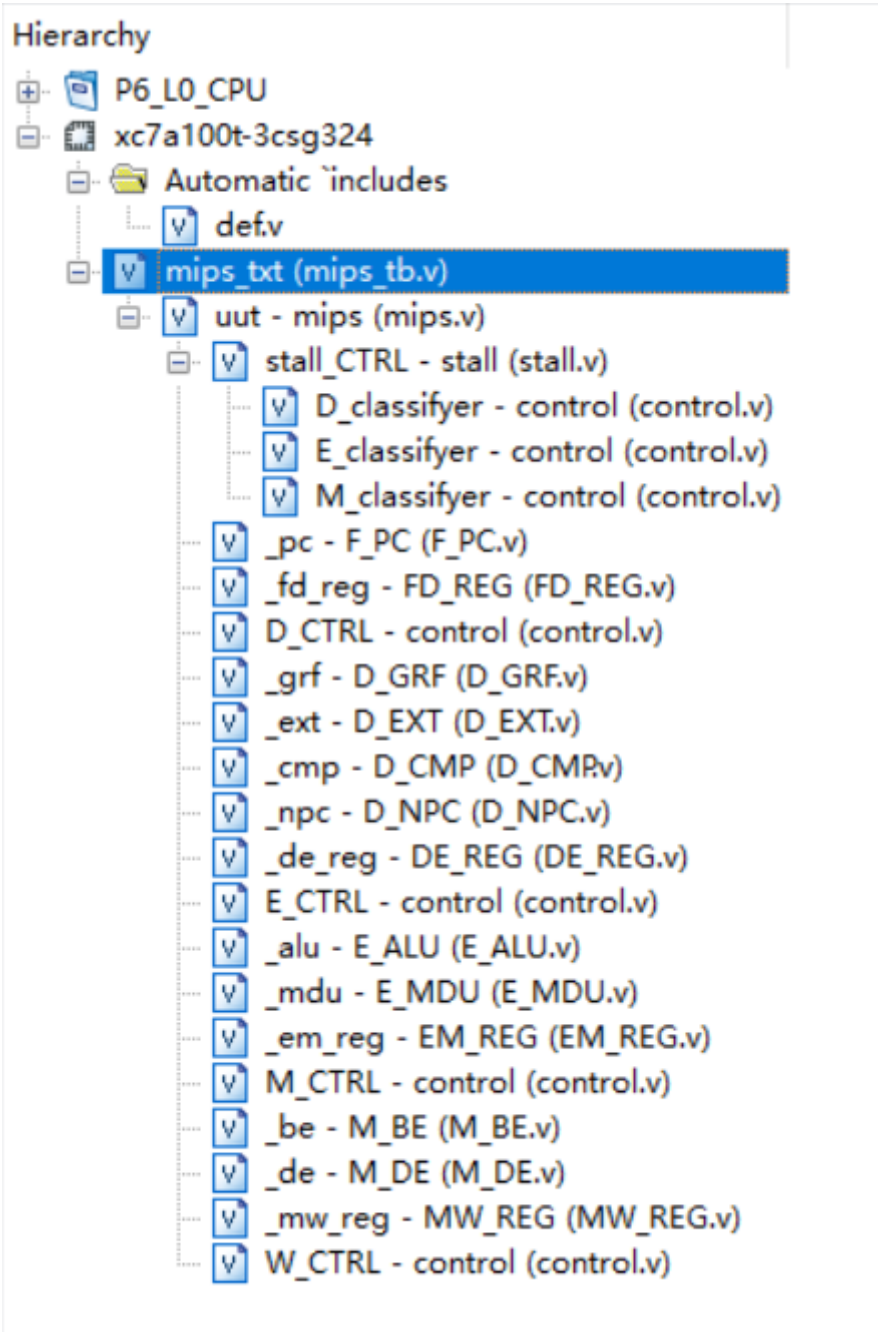
计算机组成原理实验报告—流水线CPU设计

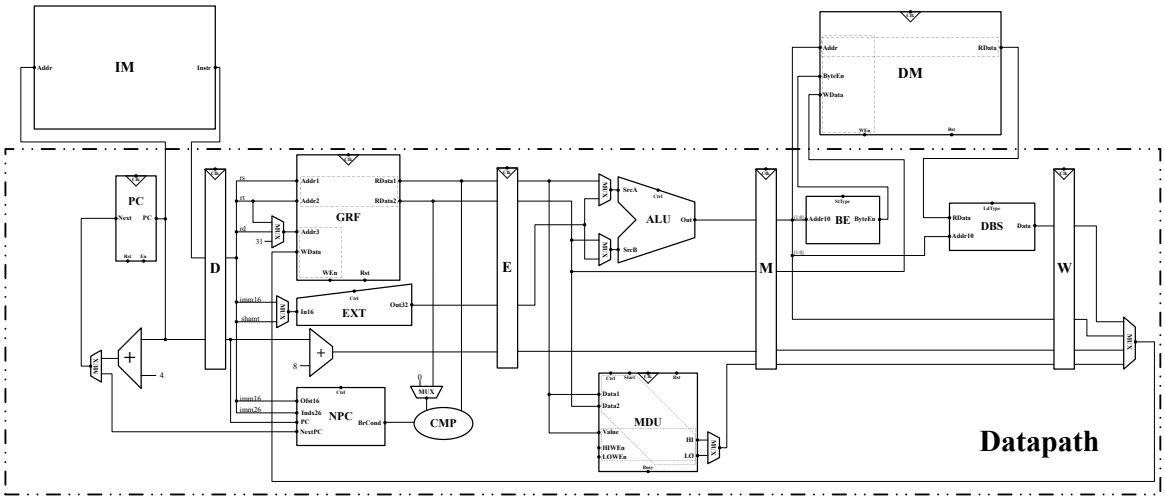
总体设计概述

要求实现的指令集为`MIPS-C3`，即**LB、LBU、LH、LHU、LW、SB、SH、SW、ADD、ADDU、 SUB、SUBU、MULT、MULTU、DIV、DIVU、SLL、SRL、SRA、SLLV、 SRLV、SRAV、AND、OR、XOR、NOR、ADDI、ADDIU、ANDI、ORI、 XORI、LUI、SLT、SLTI、SLTIU、SLTU、BEQ、BNE、BLEZ、BGTZ、 BLTZ、BGEZ、J、JAL、JALR、JR、MFHI、MFLO、MTHI、MTLO**

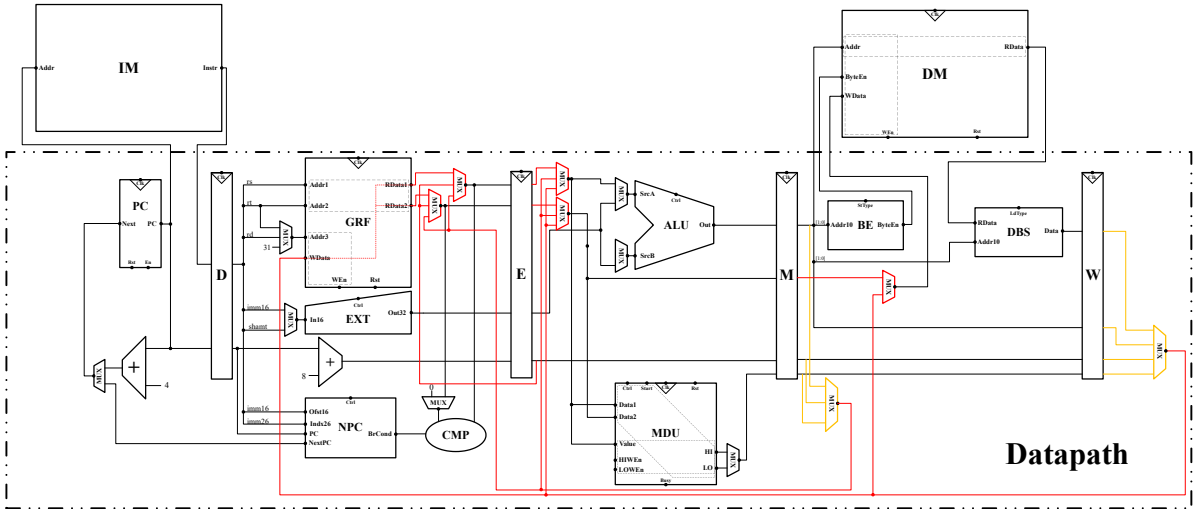
整体结构如下图所示



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

PC（程序计数器）

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

然后与`mips_txt.v`交互获得当前指令

```
F_PC _pc(
    .clk(clk),
    .reset(reset),

    .PC_WrEn(PC_WrEn),
    .NPC(NPC),

    .PC(F_PC)
);

assign i_inst_addr = F_PC;
assign F_Instr = i_inst_rdata;
```

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

现在支持了所有运算指令功能，详见`def.v`文件

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

E_MDU（乘除槽）

端口说明

信号名称	方向	功能描述
clk	输入	时钟信号
reset	输入	复位信号
MDUOp[2:0]	输入	控制信号
D1[31:0]	输入	32位输入运算数A
D1[31:0]	输入	32位输入运算数B
Start	输入	开始运算的指示信号
Busy	输出	是否处于运算过程中
HI[31:0]	输出	32位HI寄存器值结果
LO[31:0]	输出	32位LO寄存器值结果

控制信号说明

1. MDUOp

控制信号值	功能
`MDU_mult`	乘法运算
`MDU_div`	除法运算
`MDU_multu`	无符号乘法运算
`MDU_divu`	无符号除法运算
`MDU_mfhi`	`mfhi`指令
`MDU_mflo`	`mflo`指令
`MDU_mthi`	`mthi`指令，把D1的值赋给HI寄存器中
`MDU_mtlo`	`mtlo`指令，把D1的值赋给LO寄存器中

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`已经不需要自行实现，调用`mips_txt.v`中的接口即可
- 利用BE模块处理待写入数据，使其支持按半字、字节、字储存
- 利用DE模块处理DM返回的数据，使其可以按照不同要求存入寄存器

M_BE

信号名称	方向	功能描述
BEOp[1:0]	输入	控制信号
Addr[31:0]	输入	地址信息，用于处理半字、字节
rt_data[31:0]	输入	读取的寄存器数据，待处理
DMWrEn	输入	写使能
m_data_byteen[3:0]	输出	控制写入半字、字节的位置位置
m_data_wdata[31:0]	输出	待写入数据

M_DE

信号名称	方向	功能描述
DEOp[1:0]	输入	控制信号
Addr[31:0]	输入	地址信息，用于处理半字、字节
m_data_rdata[31:0]	输入	`mips_txt.v`返回的DM中的数据
DMRD[31:0]	输出	处理之后的正确的读取数据

与接口进行交互

```
// 与DM交互
M_BE _be(
    .BEOp(M_BEOp),
    .Addr(M_ALUAns),
    .rt_data(M_FWD_rt_data),
    .m_data_byteen(m_data_byteen),
    .m_data_wdata(m_data_wdata)
);

assign m_inst_addr = M_PC;
assign m_data_addr = M_ALUAns;

wire [31:0] M_DMRD;
M_DE _de(
    .DEOp(M_DEOp),
    .Addr(M_ALUAns),
    .m_data_rdata(m_data_rdata),
    .DMRD(M_DMRD)
);

// 正确输出GRF读写信息
assign w_grf_addr = W_GRFA3;
```

```
assign w_grf_wdata = W_GRFWD;
assign w_grf_we = W_GRFWrEn;
assign w_inst_addr = W_PC;
```

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

由于指令很多，采用分类的方法，同一类型的指令共享相似的数据通路

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

分为两部分：

功能性测试

采用课下提供的`P6_L0_weak.asm`测试每条指令的功能，对于未进行测试的指令，进行手动测试

覆盖性测试

首先在P5的测试工具基础上进行简单的修改，由于指令类型相似，因此仅在P5的基础上添加了`mfhi`,`mflo`,`mthi`,`mtlo`,`div`,`divu`,`mult`,`multu`指令然后进行大范围的随机生成测试

> 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

```
+---P6Judge
|   |   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

`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

对于更多情况，手动构造数据处理

思考题

- 为什么需要有单独的乘除法部件而不是整合进ALU？为何需要有独立的HI、LO寄存器？
  - 乘除法延迟远大于 ALU，整合进 ALU 那么根据木桶原理 CPU 整体周期将大幅增加。增加 HI 和 LO 寄存器可以让乘除法指令和其它指令并行执行，需要结果时再取出即可。
- 参照你对延迟槽的理解，试解释“乘除槽”。
  - 当乘除法进行或即将开始时，乘除有关指令会被阻塞在 FD 流水线寄存器，相当于处于“乘除槽”。
- 举例说明并分析何时按字节访问内存相对于按字访问内存性能上更有优势。（Hint： 考虑C语言中字符串的情况）
  - 当访问类型只占一个字节时，比如`char`。
- 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。
  - 主要是数据冒险和控制冒险，分别通过暂停转发以及比较前移+延迟槽解决。
  - 数据生成器采用了特殊策略：单组数据中除了 0 和 31 号寄存器外，至多涉及 3 个寄存器。一方面，这样产生的代码中，邻近的指令几乎全部都存在数据冒险，可以充分测试转发和暂停；另一方面，当测试数据的组数一定多，几乎涉及了每个寄存器，避免了只测试部分寄存器。此外，所有跳转指令都是特殊构造的，不会进入死循环的同时如果跳转出错可以输出中体现。
  - 对于一些会产生异常的指令，为防止 MARS 报错，进行了一定的规避，比如除法不会去生成有关0号寄存器的除法
- 为了对抗复杂性你采取了哪些抽象和规范手段？这些手段在译码和处理数据冲突的时候有什么样的特点与帮助？
  - 主要采用了指令分类的方法，P6 完全沿用了 P5 的分类方法，新增的指令对应的特点都没有脱离这些分类，因此对于每条指令而言，只需译码后将其加入对应的分类，数据通路部分和 P5 完全类似，转发部分完全不用改，暂停部分只需添加一个因乘除块而导致的暂停。