P5 流水线CPU设计文档

设计概述

- 设计的处理器为32位五级流水线处理器
- 处理器支持的指令集为

```
add, sub, and, or, slt, sltu, lui
addi, andi, ori
lb, lh, lw, sb, sh, sw
mult, multu, div, divu, mfhi, mflo, mthi, mtlo
beq, bne, jal, jr
```

- 所有运算类指令均暂不考虑因溢出而产生的异常

整体架构参考了《数字设计与计算机体系结构》图7-58。

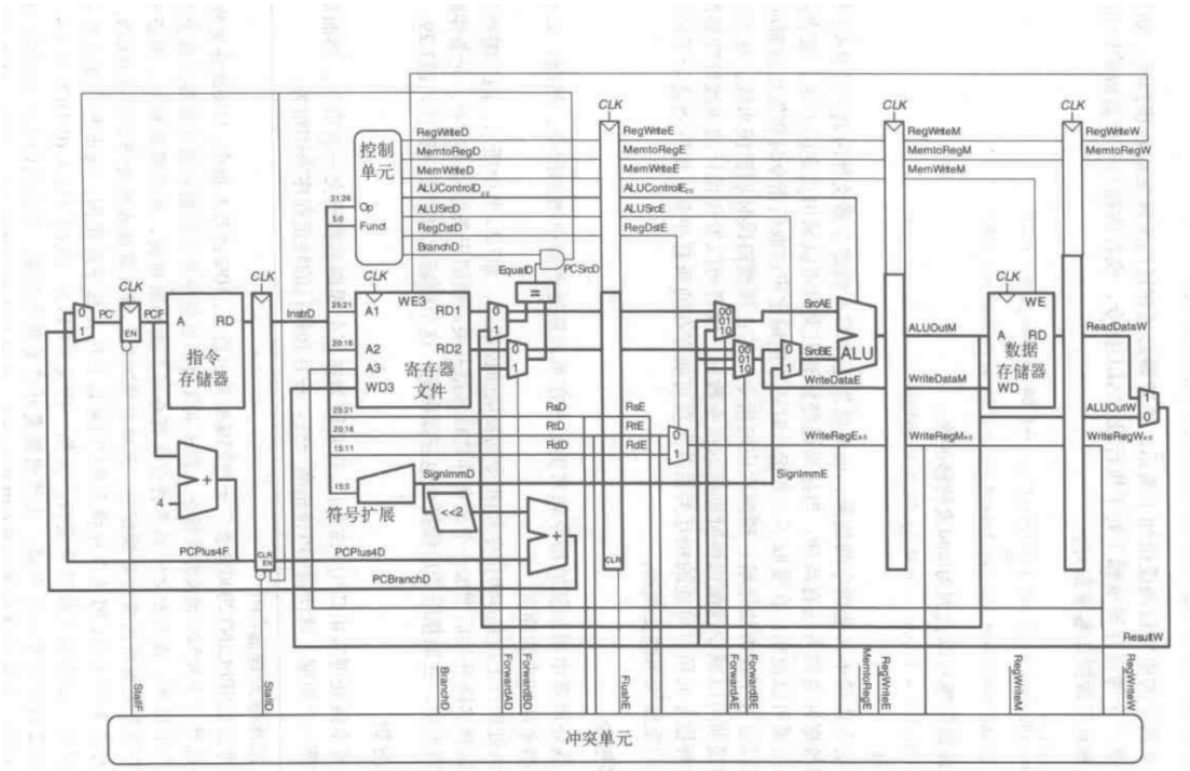

图7-58 处理所有冲突的流水线处理器

F级：取指令（Fetch）

本级的输入为来自D级的`next_pc`，用于更新下一个PC的值。

本级的输出为`F_PC`和`F_Instr`，分别对应从F级指令的PC和F级指令的内容，均需要参与流水。

F_IFU

只负责PC的存储与更新，F_instr来自`mips_txt.v`的交互。

信号名	方向	描述
clk	I	时钟信号
reset	I	**同步**复位信号，高电平有效
enable	I	PC写使能信号，高电平有效
next_pc[31:0]	I	待更新的指令地址
pc[31:0]	O	当前指令地址

与`mips_txt.v`交互

```
ifu F_IFU(
    .clk(clk),
    .reset(reset),
    .enable(IFU_WE),
    .next_pc(next_PC),
    .pc(F_PC)
);

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

F/D级流水线寄存器

信号名	方向	功能描述
clk	I	时钟信号
reset	I	**同步**复位信号，高电平有效
flush	I	寄存器刷新信号，高电平有效，发生阻塞时使用
enable	I	写使能信号，高电平有效
F_pc[31:0]	I	F级PC
F_instr[31:0]	I	时钟信号
D_pc[31:0]	O	D级PC
D_instr[31:0]	O	32位的指令值

D级：译码（Decode）

本级的输入为来自F级的`F_PC`和`F_Instr`。

本级的输出为`D_gpr_rs`，`D_gpr_rt`，`D_extres`，`D_PC`，`D_Instr`和`next_pc`。

本级涉及到来自E级、M级、W级的转发，其中来自W级的转发通过**GRF内部转发**的方式实现。

`$rs`和`$rt`的值在本级转发成`D_fwd_gprrs`和`D_fwd_gprrt`，和`D_extres`， `D_PC`， `D_Instr`参与流水。

本级需要对**此级指令**的$Tuse$和此时**E级**指令与**M级指令**的$Tnew$进行比较，从而确定是否执行阻塞。

> $Tuse$和$Tnew$：
>
> - $Tuse$表示这条指令位于D级的时候，再经过多少个时钟周期就必须要使用相应的数据。
>   - 每个指令的$Tuse$是固定不变的
>   - 一个指令可以有两个$Tuse$值
> - $Tnew$表示位于**某个流水级**的**某个指令**，它经过多少个时钟周期可以算出结果并且存储到流水级寄存器里。
>   - $Tnew$是一个动态值，每个指令处于流水线不同阶段有不同的$Tnew$值
>   - 一个指令在一个时刻至多有一个$Tnew$值（一个指令至多写一个寄存器）
> - 当$Tuse \geqslant Tnew$，说明需要的数据可以及时算出，可以通过**转发**来解决
>
>   当$Tuse < Tnew$，说明需要的数据不能及时算出，必须**阻塞**流水线解决

D_GRF

信号名	方向	描述
clk	I	时钟信号
reset	I	**同步**复位信号，高电平有效
A1[4:0]	I	5位地址输入信号，指定32个寄存器中的一个，将其中存储的数据读出到RD1
A2[4:0]	I	5位地址输入信号，指定32个寄存器中的一个，将其中存储的数据读出到RD2
A3[4:0]	I	5位地址输入信号，指定32个寄存器中的一个将WD中的数据写入
WD[31:0]	I	32位数据输入信号
WPC[31:0]	I	写入寄存器时对应的指令PC值
RD1[31:0]	O	输出A1指定的寄存器中的32位数据
RD2[31:0]	O	输出A2指定的寄存器中的32位数据

> - 如果不需要写寄存器，只需要把A3Sel设为0即可。
> - 此处**WPC**和**WD**均来自**W级**。

控制信号

WDSel	操作
WDSel_aluans	WD来自ALU的运算结果
WDSel_dmrd	WD来自DM的输出RD
WDSel_PCa8	WD为**当前流水线层级的PC** + 8

D_NPC

信号名	方向	描述
F_pc[31:0]	I	当前F级PC的值
D_pc[31:0]	I	当前D级PC的值
PCSel[1:0]	I	指定更新PC的方式
branch	I	branch类型指令**是否达到跳转条件**，高电平有效
imm[25:0]	I	j指令和jal指令中的立即数，即D_Instr[25:0]
offset[15:0]	I	branch类型指令的偏移量，即D_Instr[15:0]
ra[31:0]	I	**完成转发后** $rs 寄存器保存的地址值
next_pc[31:0]	O	下一指令的PC

控制信号

PCSel	操作
PCSel_PCa4	PC ← PC + 4
PCSel_branch	PC ← PC + 4 + sign_extend(offset||0^2>)
PCSel_j	PC ← PC[31:28] || imm || 0^2
PCSel_jr	PC ← GPR[rs]

D_CMP

信号名	方向	描述
gpr_rs[31:0]	I	**完成转发后** $rs 寄存器中的值
gpr_rt[31:0]	I	**完成转发后** $rt 寄存器中的值
CMPOp[1:0]	I	指定比较数据的方式
flag	O	是否满足所设条件，高电平有效

控制信号

CMPOp	操作
CMP_beq	beq指令：若GPR[rs] == GPR[rt]，则flag置1，否则置0

D_EXT

信号名	方向	描述
num[15:0]	I	16位需要扩展的立即数
EXTOp	I	指定进行扩展的方式
result[31:0]	O	32位完成扩展的立即数

控制信号

EXTOp	操作
EXT_zero	result = zero_extend(num)
EXT_signed	result = sign_extend(num)

D/E级流水线寄存器

信号名	方向	功能描述
clk	I	时钟信号
reset	I	**同步**复位信号，高电平有效
flush	I	寄存器刷新信号，高电平有效，发生阻塞时使用
enable	I	写使能信号，高电平有效
D_pc[31:0]	I	D级PC
D_instr[31:0]	I	32位指令值
D_extres[31:0]	I	16位立即数扩展的结果
D_gpr_rs[31:0]	I	GPR[rs]
D_gpr_rt[31:0]	I	GPR[rt]
E_pc[31:0]	O	E级PC
E_instr[31:0]	O	32位指令值
E_gpr_rs[31:0]	O	GPR[rs]
E_gpr_rt[31:0]	O	GPR[rt]

E级：执行（Execute）

本级的输入为`D_PC`，`D_Instr`，`D_extres`，`D_gpr_rs`，`D_gpr_rt`。

本级的输出为`E_PC`，`E_Instr`，`E_aluans`，`E_gpr_rt`，`E_grfWD`。

本级涉及到来自M级、W级的转发，`$rt`的值在本级转发得`D_fwd_gprrt`，和`E_PC`，`E_Instr`，`E_aluans`参与流水。

E_ALU

信号名	方向	描述
A[31:0]	I	32位运算数
B[31:0]	I	32位运算数
ALUOp[2:0]	I	指定ALU进行的计算
result[31:0]	O	32位运算结果

控制信号

详见 `def.v` 文件中的定义。

E_MDU（乘除槽）

信号名	方向	功能描述
clk	I	时钟信号
reset	I	**同步**复位信号，高电平有效
MDUOp[2:0]	I	指定乘除槽进行的操作
gpr_rs[31:0]	I	GPR[rs]
gpr_rt[31:0]	I	GPR[rt]
start	I	指定乘除槽是否开始计算，高电平有效
busy	O	乘除槽是否处于运算过程中
HI[31:0]	O	32位HI寄存器值结果
LO[31:0]	O	32位LO寄存器值结果

控制信号

MDUOp	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 `gpr_rs` 的值赋给HI寄存器中
`MDU_mtlo`	`mtlo` 指令，把 `gpr_rs` 的值赋给LO寄存器中

E/M级流水线寄存器

信号名	方向	功能描述
clk	I	时钟信号
reset	I	**同步**复位信号，高电平有效
flush	I	寄存器刷新信号，高电平有效，发生阻塞时使用
enable	I	写使能信号，高电平有效
E_pc[31:0]	I	E级PC
E_instr[31:0]	I	32位指令值
E_aluans[31:0]	I	ALU的运算结果
E_gpr_rt[31:0]	I	GPR[rt]
M_pc[31:0]	O	M级PC
M_instr[31:0]	O	32位指令值
M_aluans[31:0]	O	ALU的运算结果
M_gpr_rt[31:0]	O	GPR[rt]

M级：存储器（Memory）

本级的输入为`E_PC`，`E_Instr`，`E_aluans`，`E_gpr_rt`。

本级的输出为`M_PC`，`M_Instr`，`M_aluans`，`M_dmrd`，`M_grfWD`。

本级涉及到来自M级、W级的转发，`$rt`的值在本级转发得`M_fwd_gprrt`。

本机参与流水的有`M_PC`，`M_Instr`，`M_aluans`，`M_dmrd`。

M_DM

本次实验只需要调用调用`mips_txt.v`中的接口即可，无需自行实现DM。

使用`fromRAM`模块处理DM返回的数据，使其符合写入寄存器的要求。

使用`toRAM`模块处理写入DM的数据，支持按照字、半字、字节的模式储存进DM。

M_fromRAM

信号名	方向	功能描述
A[31:0]	I	进行写操作的地址
DMOp[1:0]	I	指定模块进行的操作
m_data_rdata[31:0]	I	从`mips_txt.v`中的DM读出的数据
RD	O	处理后的正确的读取数据

M_toRAM

信号名	方向	功能描述
A[31:0]	I	进行读操作的地址
gpr_rt[31:0]	I	读取的待处理的寄存器数据
DMOp[1:0]	I	指定模块进行的操作
MemWrite	I	写使能信号，高电平有效
m_data_byteen[3:0]	O	控制写入数据在DM中的位置
m_data_wdata[31:0]	O	处理后的正确的待写入数据

控制信号

DMOp	操作
DM_word	读/写整个字，对应lw和sw指令
DM_halfword	读/写半个字，对应lh和sh指令
DM_byte	读/写一个字节，对应lb和sb指令

与mips_txt.v交互

```
wire [31:0] M_dmrd;
assign m_inst_addr = M_PC;
assign m_data_addr = M_aluans;

toRAM M_Store(
    .A(M_aluans),
    .gpr_rt(M_fwd_gprrt),
    .DMOp(M_DMOp),
    .MemWrite(M_MemWrite),
    .m_data_byteen(m_data_byteen),
    .m_data_wdata(m_data_wdata)
);

fromRAM M_Load(
    .A(M_aluans),
    .DMOp(M_DMOp),
    .m_data_rdata(m_data_rdata),
    .RD(M_dmrd)
);

assign w_grf_wdata = W_grfWD;
assign w_inst_addr = W_PC;
assign w_grf_addr = W_A3Sel;
```

M/W级流水线寄存器

信号名	方向	功能描述
clk	I	时钟信号
reset	I	**同步**复位信号，高电平有效
flush	I	寄存器刷新信号，高电平有效，发生阻塞时使用
enable	I	写使能信号，高电平有效
M_pc[31:0]	I	M级PC
M_instr[31:0]	I	32位指令值
M_aluans[31:0]	I	ALU的运算结果
M_dmrd[31:0]	I	从DM中读取的值
W_pc[31:0]	O	W级PC
W_instr[31:0]	O	32位指令值
W_aluans[31:0]	O	ALU的运算结果
W_dmrd[31:0]	O	从DM中读取的值

W级：写回（Writeback）

本级与D级是重合的，需要处理向E级和M级的转发。

转发

采用**暴力转发**的方式。由AT法的分析，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。因此采用暴力转发总能得到一个正确的值去覆盖原先错误的值。

```
// D级转发
assign D_fwd_gprrs = (D_rs == 5'd0) ? (32'd0) :
                     (D_rs == E_A3Sel) ? E_grfWD :
                     (D_rs == M_A3Sel) ? M_grfWD : D_gpr_rs;

assign D_fwd_gprrt = (D_rt == 5'd0) ? (32'd0) :
                     (D_rt == E_A3Sel) ? E_grfWD :
                     (D_rt == M_A3Sel) ? M_grfWD : D_gpr_rt;


// E级转发
assign E_grfWD = (E_WDSel == `WDSel_PCa8) ? (E_PC + 8) : 32'd0; // 不能对功能部件输出进行转发
assign E_fwd_gprrs = (E_rs == 5'd0) ? 32'd0 :
                     (E_rs == M_A3Sel) ? M_grfWD :
                     (E_rs == W_A3Sel) ? W_grfWD :
                     E_gpr_rs;

assign E_fwd_gprrt = (E_rt == 5'd0) ? 32'd0 :
                     (E_rt == M_A3Sel) ? M_grfWD :
```

```
                    (E_rt == W_A3Sel) ? W_grfWD :
                    E_gpr_rt;


// M级转发
assign M_grfWD = (M_WDSel == `WDSel_aluans) ? M_aluans :
                 (M_WDSel == `WDSel_mduans) ? M_mduans :
                 (M_WDSel == `WDSel_PCa8) ? (M_PC + 8) : 32'd0;
assign M_fwd_gprrt = (M_rt == 5'd0) ? (5'd0) :
                    (M_rt == W_A3Sel) ? W_grfWD :
                    M_gpr_rt;


// W级转发
assign W_grfWD = (W_WDSel == `WDSel_aluans) ? W_aluans :
                 (W_WDSel == `WDSel_dmrd) ? W_dmrd :
                 (W_WDSel == `WDSel_mduans) ? W_mduans :
                 (W_WDSel == `WDSel_PCa8) ? (W_PC + 8) : 32'd0;
```

阻塞

使用组合逻辑，判断每一级中指令的$Tuse$和$Tnew$。

如果有$Tuse < Tnew$，就执行阻塞。**只可能在D级进行阻塞。**

```
// stall_handle.v
assign D_Tuse_rs = (D_branch | D_j2r) ? 3'd0 :
                 (D_ic | D_rc | D_load | D_store | D_mt | D_md) ? 3'd1 :
3'd3;

assign D_Tuse_rt = (D_branch) ? 3'd0 :
                    (D_rc | D_md) ? 3'd1 :
                    (D_store) ? 3'd2 : 3'd3;

assign E_Tnew = (E_rc | E_ic | E_mf) ? 3'd1 :
                (E_load) ? 3'd2 :
                3'd0;

assign E_stall_rs = (E_A3Sel == D_rs && D_rs != 5'd0) && (E_Tnew >
D_Tuse_rs);
assign E_stall_rt = (E_A3Sel == D_rt && D_rt != 5'd0) && (E_Tnew >
D_Tuse_rt);

assign M_Tnew = (M_load) ? 3'd1 : 3'd0;

assign M_stall_rs = (M_A3Sel == D_rs && D_rs != 5'd0) && (M_Tnew >
D_Tuse_rs);
assign M_stall_rt = (M_A3Sel == D_rt && D_rt != 5'd0) && (M_Tnew >
D_Tuse_rt);

assign stall = E_stall_rs | E_stall_rt | M_stall_rs | M_stall_rt |
E_stall_mdu;


```

```
// mips.v
wire stall;
assign FD_WE = !stall; //冻结FD寄存器
assign IFU_WE = !stall; //冻结PC
assign DE_flush = stall; //清空DE寄存器

assign DE_WE = 1'b1;
assign EM_WE = 1'b1;
assign MW_WE = 1'b1;

assign FD_flush = 1'b0;
assign EM_flush = 1'b0;
assign MW_flush = 1'b0;
```

测试方案

改进了P5的随机数据生成器，添加了延迟槽。

一个测试数据如下

```
add $a0,$0,$0
ori $t1,$0,1
ori $t2,$0,2
ori $t3,$0,3
add $t4,$t1,$t2
sub $t5,$t1,$t2
jal out
add $t3,$t4,$0

out:
ori $a0,$0,11
beq $a0,$a1,end
lui $a2,111
jr $ra
ori $a1,$0,11
end:
nop
```

思考题

1. 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？

   乘除法的延迟远大于ALU，若整合进ALU，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。

2. 真实的流水线CPU是如何使用实现乘除法的？请查阅相关资料进行简单说明。

   乘法通常有若干个较小的组合逻辑的乘法单元组成，然后每个周期计算特定的几位，依次累加起来，于是会在几个周期后得到正确的最终结果；

   除法通常使用试商法，通常也是使用组合逻辑在一个周期内计算4位左右的商，经过8个周期正好可以计算结束。

3. 请结合自己的实现分析，你是如何处理 Busy 信号带来的周期阻塞的？

   当Busy信号或Start信号为 1 时，`mult, multu, div, divu, mfhi, mflo, mthi, mtlo` 等乘除法相关的指令均被阻塞在 D 流水级。

4. 请问采用字节使能信号的方式处理写指令有什么好处？（提示：从清晰性、统一性等角度考虑）

   使用字节使能大幅度提高M级的效率，控制器提供控制信号，存储器决定最终存储地址，使得每个模块的功能更加清晰。

5. 请思考，我们在按字节读和按字节写时，实际从DM获得的数据和向DM写入的数据是否是一字节？在什么情况下我们按字节读和按字节写的效率会高于按字读和按字写呢？

   不是，读取到的是DM所在字的数据。只实现按字使能，就不得不先读取值再进行更改，这样数据通路就延长了，导致效率下降。

   访问类型只占一个字节时，比如C语言中的 `char` 类型，按字节访问内存相对于按字访问内存性能上更有优势。

6. 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？

   将指令集按照一定的规则进行分类。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。相同的类别在译码上有共同的结果，可以避免偶然错误。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。

7. 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？

   P6比P5增加了乘除槽的冲突，在进行乘除指令后紧接着立即执行`mfhi`或`mflo`会造成数据冲突。此时应进行阻塞，直到乘除指令执行完成为止。

   构造样例如下：

```
lui $t0,0x0123
lui $t2,0xffff
ori $t0,$t0,0x9576
ori $t1,$t1,0xa312
mult $t0,$t1
mfhi $s0
mflo $s1
mult $t0,$t2
mfhi $s0
mflo $s1
multu $t0,$t2
mfhi $s0
mflo $s1
```

8. 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。

   新增冲突只有一条，所以构造上述测试样例足够。