

要考虑当 CPU stall 的时候不能清空延迟槽。

还有一种方法是监测到 `D_b_jump` 时, F 级的指令插入 `nop` (这个在 P7 很常用)。

```
F_instr = (!D_b_jump && D_bgezall) ? 32'd0 : im[pc[13:2] - 12'hc00];
```

- 第三题: `lwso`

```
mem_data ← mem[GPR[rs] + offset]
temp ← (GPR[31]31 || GPR[31]) + (mem_data31 || mem_data)
if temp32 == temp31 then
    GPR[31] ← GPR[31] + mem_data
```

实际上就是: 如果第二步相加没有溢出就赋值, 否则不赋值。

## 上机总结 2

爬了, 又没过。

- 第一题: `cmco`

```
temp ← 0
count ← 0
for i in 0...31
    if GPR[rs]i == 1 then
        count ← count + 1
        if count > temp then
            temp ← count
    else
        count ← 0
GPR[rd] ← temp
```

同样改一下 ALU 就好了。就是统计二进制最长有几个连续的 1。

- 第二题: `blezalc`

```
target_offset ← signed_ext(offset || 02)
condition ← GPR[rs] ≤ 0
if condition then
    PC ← PC + 4 + target_offset
    GPR[31] ← PC + 8
```

这个题目课上的 RTL 表述其实有问题.....更正后应该是这样的，即如果跳转则链接寄存器。

- 第三题：lrm

$$vAddr \leftarrow GPR[base] + \text{sign\_extend}(offset)$$
$$mem\_reg \leftarrow Memory[vAddr]_{4..0}$$
$$GPR[mem\_reg] \leftarrow GPR[rt]$$

存储地址为 DM 输出的低位。

## 上机总结 3 & 分析

终于过了.....这次的指令是前两次的混合体，所以就不展开说了。

这里就简单讲讲 P5 的套路吧。

首先课上的三条指令：计算 + 跳转 + 存储。

1. 计算绝对是最简单的，重点在于你的写法。如果你用 `always@(*)` 那么问题不大。但是如果你用的是 `assign` 那么最好使用 `function` 来实现计算指令。
2. 跳转一般也不难，一般是条件跳转 + 条件写。跳转指令一个好处在于它是在 D 级决定是否跳转的，也就是说在 D 级你可以获得全部的正确信息（相反如果是类似于 `lwso` 这种，你必须读出 DM 的值才能决定怎么做）。所以我们的方案是 D 级生成一个 `D_check` 信号然后流水它。然后每一级根据这个信号判断写入地址/写入值之类的。

```
// 检测信号
D_check = D_bgezalc & D_b_jump;
// CU
assign RFDst = // ...
                bgezalc ? (check ? 5'd31 : 5'd0) :
                5'd0;
```

3. 条件存储一般是最难的。但是掌握了套路之后也还好。条件存储的特点是必须要到 M 级才知道要写啥，这就给转发之类的造成了困难，所以我们的策略是如果 D 级要读寄存器，而且新指令**可能**要写这个寄存器，那么就 `stall`。具体来说是这样的：

```
//lwso
wire stall_rs_e = (TuseRS < TnewE) && D_rs_addr && (D_lwso ? D_rs_addr == 5'd31 : D

// lrm
wire stall_rs_e = (TuseRS < TnewE) && D_rs_addr && (D_lrm ? D_rs_addr : D_rs_addr =
```

verilog

在 CU 中的写法则与条件跳转类似。

```
// lws0
M_check = D_lws0 && condition;
// CU
assign RFDst = // ...
               lws0 ? (check===1'd1 ? 5'd31 : 5'd0) : // 注意不是直接一个 check
               5'd0;
```

这里要注意的一点是，如果你用的是统一 CU 的写法，那么会出现一个问题：check 信号只有在 M 级传入。这个时候在 E 级的 CU 里面这个信号是不定态 x，这样会导致 RFDst 信号出锅。所以我们这里的写法是 check===1'd1，这样可以排除 x 的情况。

## 课下总结

P5 做得真是艰难，一方面是流水线这个东西本身比较复杂，包括转发/暂停等一系列机制；另一方面是数据通路由于流水线寄存器的加入，代码量大大上升，个人经验大概比 P4 增加了 1/2 到 1 倍左右。而且 P5 时流水起迭代开发的起点，所以一个好的架构非常重要，所以做的时候会非常想要不停地重构，这又浪费了很多时间。所以建议在做之前先做好设计工作，然后做的过程中不要临时起意去修改。

总的来说流水线的难点在于三方面：

1. 数据通路和控制器架构
2. 转发
3. 暂停

下面我会分别介绍这三方面的内容（主要是教程内容，我会做一些个人理解和细节补充）。

注意：做 P5 前请确保自己熟悉 Verilog 和 P4，并且看完了教程内容和高小鹏的课件。

## 流水线要点

开发时你需要注意以下几个方面：

1. 命名风格

2. 代码排布：建议同一级放在一起，同样的功能放在一起，独立的部分可以适当抽出独立成模块，即遵循“低耦合，高内聚”的原则
3. 多使用宏和常量来简化代码，如推荐使用 `ALU_add` 代替 `3'b001`

## 数据通路及控制架构

### 架构方案选择

数据通路架构即 FDEMW 五个部分的数据通路，以及数据通路中的流水线寄存器。

P5 是 CPU 迭代开发的起点，因此一个好的架构是很好的起点。总的来说架构包括这几部分：

1. 译码方式：分为集中式译码和分布式译码。

集中式译码即在 F 或 D 级进行译码，然后将控制信号流水传递，即 P3/P4 采用的译码方式；分布式译码则只流水传递指令，控制信号在每一级单独译码。

集中式译码的好处在于速度更快，关键路径更短；分布式译码关键路径更长，速度较慢（差不了很多）但是译码信息模块化，不需要流水传递控制信号。为了方便这里采用分布式译码。

实际上更推荐大家用集中式的写法。

2. 译码语句：

可以用 `always @(*)` + `case` 或者 `assign` 进行译码。这里使用后者（因为后者代码更短）。

3. 控制器设计：

为每一级单独写一个 CU，或者写一个大的 CU 每一级实例化一次，取不同的信号使用。

前者可以节省晶体管，但是实际操作会遇到很多问题：比如指令一多（P6）那样译码信息写起来很麻烦。如果你像我一样，写了一个 `interpreter.v` 来复用解码代码，那样会发现还不如直接单写一个 CU，而且一些信号会重复使用，如寄存器写使能信号在阻塞模块中会重复使用。出于面向对象的工程思想，我们倾向于把信号集中在一起。

后者写起来简单，但是要使用更多的晶体管。

这里使用后者。（我们要对编译器抱有信心，相信他会帮我们优化代码的 (。·ω·。))

总而言之，我们采用的是更工程化，更符合“低耦合高内聚”原则的方案，效率和成本不是我们考虑的首要因素。

## 数据通路架构

数据通路部分的元件基本上和 P4 一样，我们将它们分在不同的层级。

- 1. F 级：PC, IM
- 2. D 级：GRF, EXT, CMP, NPC
- 3. E 级：ALU
- 4. M 级：DM
- 5. W 级：无

连线参考教程和 P4。需要注意的有两点：

- 1. 新元件 CMP 用于 b 类指令
- 2. NPC 位于 D 级，他的输入中既需要  $D_{pc}$  (b 类, j 类) , 又需要  $F_{pc}$  ( $PC+4$ ) 。

## 控制器架构

控制器我分为三部分。第一部分分割指令（取出 rs, imm 等），第二部分解析指令（`wire add = (opcode = `OP_add) && (func = `FU_add)`），第三部分分类指令（load、calc\_r、calc\_i 之类），第四部分建模控制信号。

采用统一控制器的好处是，在数据通路、转发、阻塞中你都可以复用这些代码。

## 转发

只有 RS 和 RT 会被转发。有四个位点是转发的接受端：

- 1. NPC 的 RS 输入端
- 2. CMP 的两个输入端
- 3. ALU 的输入端
- 4. DM 的输入端

可以的话最好先画一个图。

转发接受
NPC

转发接受
D_CMP_A/B
ALU_A/B
DM_in

转发输出	(就近优先)
E_reg	j_l / lui
M_reg	calc_r/calc_i
W_reg	load
寄存器内部转发	

注意这里我把 lui 特殊处理了，并没有归入到 calc\_i 里面去。

然后是接受端选择数据，选择的顺序按照**就近原则**，优先选择下一级的数据，不行就下两级，如果都不行就采用本级寄存器的数据。接受端可以接受数据的条件为：

1. 供给端的寄存器地址与当前的相同
2. 当前需要的地址不为 0
3. 供给端可以转发（我们使用无脑转发，所以不需要考虑）
4. 供给端的指令会写数据（如果不写数据我们令写入地址为 0，所以不需要考虑）

即：

```

wire [31:0] FWD_E_RS = (E_rs_addr == 5'b0) ? 0 :
                      (E_rs_addr == M_RFDst) ? M_RFWD :
                      // ...
                      E_rs;

```

这里的 A-T 法有一个很妙的地方。注意到我们的判断条件仅仅是  $E\_rs\_addr = M\_RFDst$  而不用加其他信号，相当于暴力转发。假如 D 级可以用 E/M/W 级的数据，但是 E 级还没算好，为什么不需要判断 E 的指令能否被转发呢？我们考虑两种情况：

1. 需要暂停 比如 beq(D)-add(E)-add(M)-add(W)，那么直接 stall，下一个周期 E 变成 nop 了，那么  $E\_RFDst$  不符合  $E\_rs\_addr = M\_RFDst$ ，自然就使用 M 了。

2. 不需要暂停 比如 `add(D)-add(E)-add(M)-add(W)` , 那么这个时候 D 的 `add` 会从 E 得到一个错误的数, 但是没关系, 现在还不用。下一个周期要用到这个数据的时候, 我们进行了第二次转发, 即 M (原先的 E) 转发过来的数据, 此时这个数据经过 ALU 已经算完了, 所以我们可以直接利用。

需要注意的有以下几点:

1. 本级的寄存器接受了转发后, 一律使用转发后的值。我们可以把使用原值也看成一种转发, 即我们必须使用最新的值, 包括给下一级寄存器传递信息时也要用转发后的数据。

如 ``FWD_E_RS`` 是转发后的值, 那么在 ``M_reg`` 的 ``rs_in`` 信号就要用 ``FWD_E_RS`` 而不是 ``E_r`

## 2. 寄存器内部转发

其实就是 D 级接受 W 级的转发电路放在了寄存器内部

```
assign RD1 = (A3 == A1 && A3 && WE) ? WD : grf[A1]; // 内部转发
```

3. 为什么转发不需要判断 `M_RFW` ? 因为在 `M_RFDst` 中我们令不写寄存器的部件的写入地址为 `$0` , 所以不会对结果造成影响。

## 阻塞

建议通读教程, 看一遍高小鹏的 PPT, 尤其要搞清楚 `Tuse` 和 `Tnew` 两个东西。

- `Tuse`: 指令进入 D 级后, 其后的某个功能部件再经过多少时钟周期就必须使用寄存器值。对于有两个操作数的指令, 其每个操作数的 `Tuse` 值可能不等 (如 `store` 型指令 `rs`、`rt` 的 `Tuse` 分别为 1 和 2) 。
- `Tnew`: 位于 E 级及其后各级的指令, 再经过多少周期就能够产生要写入寄存器的结果。在我们目前的 CPU 中, W 级的指令 `Tnew` 恒为 0; 对于同一条指令,  $Tnew@M = \max(Tnew@E - 1, 0)$  。

然后直接遵循  $Tuse < Tnew$  则阻塞的规则就可以了。建议先画一个表格分类 `Tuse` 和 `Tnew`。可以按照高小鹏的 PPT, 用“产生输出的部件”来分类 `Tnew`。如 ALU 产生输出的有 R 型、I 型、`sw` 的地址, 把他们归为一类。

阻塞时只需要判断 RS 和 RT。



Tuse			
calc_r(rs)\shiftS	1	calc_r(rt)	1
calc_i(rs)	1	shiftS(rt)	1
load(rs)	1	store(rt)	2
store(rs)	1	branch(rt)	0
branch(rs)	0		
jr/jalr(rs)	0		

判断的时候要注意条件：

1.  $Tuse < Tnew$
2. 当前需要的地址不为 0
3. 写入地址和后面级写入地址相同
4. 供给端可以转发 且 后面级可写寄存器（不考虑）

```

wire [2:0] TuseRS = (D_branch | D_j_r) ? 3'd0 :
                    // ...
                    3'd3; // 如果用不到就令其为无穷大，防止多余的 stall

wire [2:0] TnewE = E_calc_r | E_calc_i ? 3'd1 :
                  // ...
                  3'd3;

wire stall_rs_e = (TuseRS < TnewE) && (D_rs_addr && D_rs_addr == E_RFDst);
// ...
wire stall_rs = stall_rs_e | stall_rs_m;

// ...

assign stall = stall_rs | stall_rt;

```

为什么这里写 stall 不用判断 wr 信号，即不考虑最后一条？理由同转发。

#### PREVIOUS

[BUAA-CO-LAB] P4 单周期 CPU - 2  
(/POSTS/BUAA-CO-LAB-P4/)

#### NEXT

[BUAA-CO-LAB] P6 流水线 CPU-LITE2  
(/POSTS/BUAA-CO-LAB-P6/)



Copyright © ROIFE BLOG 2023 | Powered by Hugo (<https://gohugo.io>) | Made with Emacs  
(<https://www.gnu.org/software/emacs/>)