P6课下-Verilog搭建流水线CPU-2

五级流水线，支持指令集：

```
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
```

不考虑溢出。

新增指令行为及分类

R 型指令

- add
- sub
- and：新增与运算行为，其余同 add、sub
- or：或运算，其余同 add、sub
- slt：如果 GPR[rs] < GPR[rt]，则 GPR[rd] 置为 1 ，否则置为 0；**拟实现**：放进 ALU 处理，目的是能将这个整合进 R 指令。
- sltu：无符号版本的 slt；**拟实现**：ALU .

I 型指令

- lui
- addi：立即数版本的 add，ori 和 add 混合版
- andi：立即数版本的 and
- ori

访存指令

- lb：用 GPR[rs]+IMM16 计算地址 addr ，取出 Memory[addr] ，addr 最低两位决定截取 Memory[addr] 的哪一部分字节，符号扩展后存入 GPR[rt]；**拟实现**：在 M 级取出 Memory[addr] 后，即刻判断并取出字节，符号扩展，流水到 W 级。在 M 和 W 都可转发。
- lh：部分同 lb ， addr 第二位决定截取 Memory[addr] 的那一部分半字，符号扩展后存入 GPR[rt]
- lw
- sb：将 GPR[rt] 的 [7:0] 存入 Memory[addr] 特定字节位，选择方式同 lb
- sh：将 GPR[rt] 的 [15:0] 存入 Memory[addr] 特定半字位
- sw

乘除指令

- mult：符号乘，GPR[rs]*GPR[rt] 结果低 32 位存入 LO，高 32 位存入 HI
- multu：无符号版本的 mult
- div：符号除，GPR[rs]/GPR[rt] 存入 LO，GPR[rs]%GPR[rt] 存入 HI
- divu：无符号版本的 div
- mfhi：将 HI 存入 GPR[rd]
- mflo：将 LO 存入 GPR[rd]
- mthi：将 GPR[rs] 写入 HI
- mtlo：将 GPR[rs] 写入 LO

J 型指令

- beq
- bne：不相等跳转，改造 beq 即可
- jal
- jr

指令	op5	op4	op3	op2	op1	op0
"special"	0	0	0	0	0	0
lui	0	0	1	1	1	1
addi	0	0	1	0	0	0
andi	0	0	1	1	0	0
ori	0	0	1	1	0	1
lb	1	0	0	0	0	0
lh	1	0	0	0	0	1
lw	1	0	0	0	1	1
sb	1	0	1	0	0	0
sh	1	0	1	0	0	1
sw	1	0	1	0	1	1
beq	0	0	0	1	0	0
bne	0	0	0	1	0	1
jal	0	0	0	0	1	1

指令	func5	func4	func3	func2	func1	func0
add	1	0	0	0	0	0
sub	1	0	0	0	1	0
and	1	0	0	1	0	0
or	1	0	0	1	0	1
slt	1	0	1	0	1	0
sltu	1	0	1	0	1	1

指令	func5	func4	func3	func2	func1	func0
mult	0	1	1	0	0	0
multu	0	1	1	0	0	1
div	0	1	1	0	1	0
divu	0	1	1	0	1	1
mfhi	0	1	0	0	0	0
mflo	0	1	0	0	1	0
mthi	0	1	0	0	0	1
mtlo	0	1	0	0	1	1
jr	0	0	1	0	0	0
nop	0	0	0	0	0	0

冒险处理的相似性抽象

1. R-计算型 R 指令：add, sub, and, or, slt, sltu
2. I-计算型 I 指令：lui, addi, andi, ori
3. Load-加载指令：lb, lh, lw
4. Store-存储指令：sb, sh, sw
5. Md-计算型乘除指令：mult, multu, div, divu
6. Mt-存数型乘除指令：mthi, mtlo
7. Mf-取数型乘除指令：mfhi, mflo
8. Branch-分支指令：beq, bne
9. Jump-跳转指令：jal
10. Jr-返回指令：jr
11. Nop-空指令：nop

指令	rs_T_{use}	rt_T_{use}	E_T_{new}	M_T_{new}	W_T_{new}
R	1	1	1	0	0
I	1	3	1	0	0
Load	1	3	2	1	0
Store	1	1	0	0	0
Md	1	1	0	0	0
Mt	1	3	0	0	0
Mf	3	3	1	0	0
Branch	0	0	0	0	0
Jump	3	3	2	1	0
Jr	0	3	0	0	0
Nop	3	3	0	0	0

控制信号

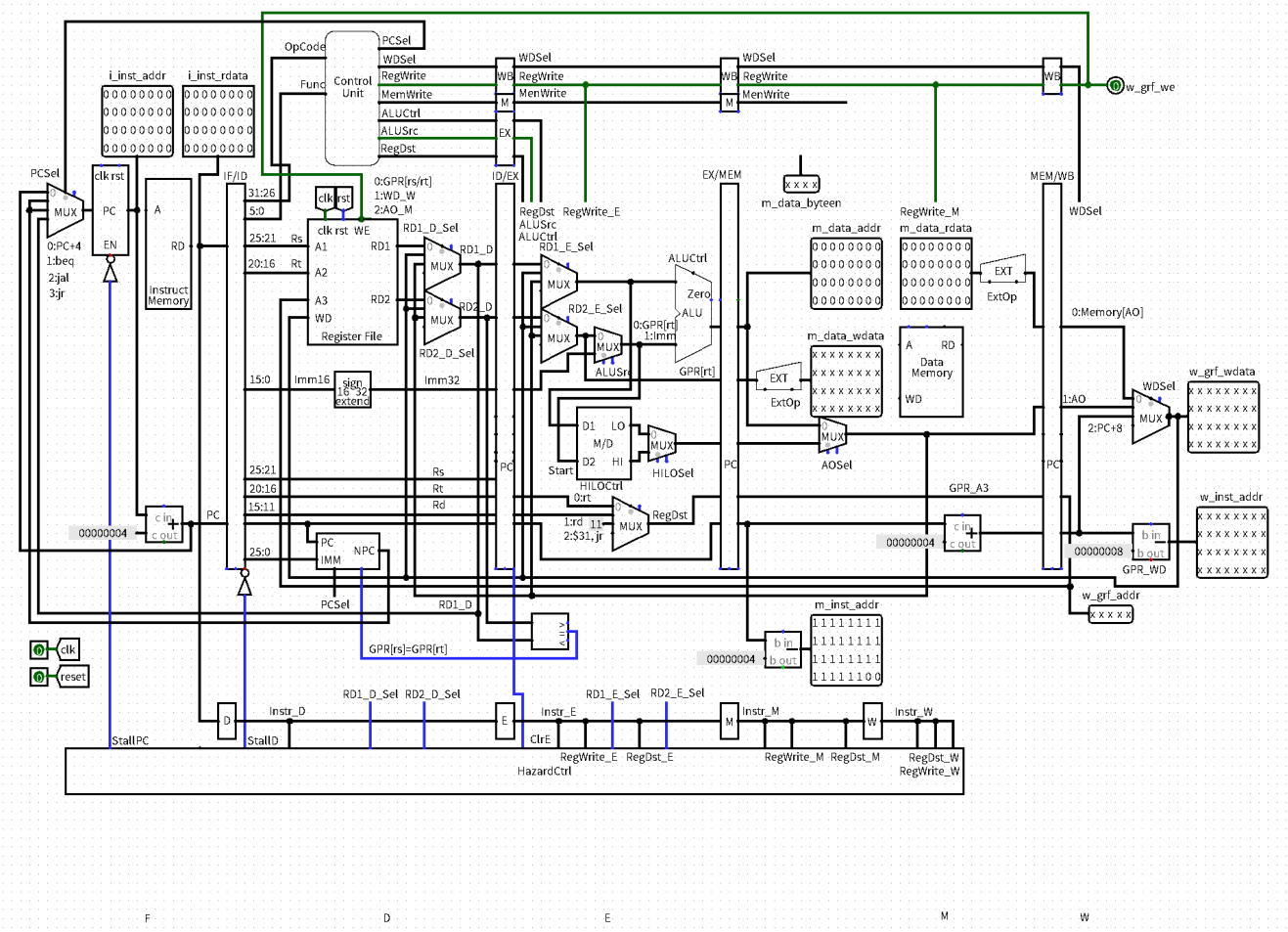
基础控制信号

指令	WDSel	RegWrite	MemWrite	PCSel	ALUCtrl	ALUSrc	RegDst	HILOCtrl	A
add	01	1		000	000	0	01		
sub	01	1		000	001	0	01		
and	01	1		000	010	0	01		
or	01	1		000	011	0	01		
slt	01	1		000	100	0	01		
sltu	01	1		000	101	0	01		
lui	01	1		000	110	1	00		
addi	01	1		000	000	1	00		
andi	01	1		000	010	1	00		
ori	01	1		000	011	1	00		
lb	00	1		000	000	1	00		
lh	00	1		000	000	1	00		
lw	00	1		000	000	1	00		
sb			1	000	000	1			
sh			1	000	000	1			
sw			1	000	000	1			
mult				000				0100	
multu				000				0101	
div				000				0110	
divu				000				0111	
mfhi	01	1		000			01	0000	
mflo	01	1		000			01	0001	
mthi				000				0010	
mtlo				000				0011	
beq				001					
bne				100					
jal	10	1		010			10		
jr				011					

最后一列有个 AOSel 信号在 pdf 中显示不出来，是在所有和乘除有关的指令置 1，所有和 ALU 有关的置 0.

字节使能信号

指令	地址[1:0]	m_data_byteen	ExtOp
sb	00	0001	0
sb	01	0010	0
sb	10	0100	0
sb	11	1000	0
sh	0X	0011	0
sh	1X	1100	0
sw	XX	1111	0
lb	00	0001	1
lb	01	0010	1
lb	10	0100	1
lb	11	1000	1
lh	0X	0011	1
lh	1X	1100	1
lw	XX	1111	1



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？

因为乘除法相比于加减法有时间延迟，会有新的控制信号，因此独立出来比较好。单独的 HI、LO 寄存器的存在是较于 Mars 行为以及 MIPS 指令集而言。

> 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。

乘法器电路：一个 32x32 位的乘法器电路来进行操作，可以是以下两种结构：

- Booth 编码：可以优化乘法操作，尤其是处理符号位；
- Wallace 树：一种快速实现多位数乘法的算法结构。
  除法器电路：包括以下步骤：
- 预处理：调整除数与被除数的符号，使之变成正数；
- 估算和调整：通过一系列估算和调整来逼近最终的商；
- 迭代：重复减去除数的倍数，直到被除数小于除数为止。

> 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?

Busy 信号的产生：Start 信号产生后的下一个时钟上升沿产生；
Busy 信号的消失：当 cnt 达到 max 的时候消失。
Busy 信号带来的阻塞：Busy 和 Start 信号都输入冒险控制模块，只要其中一个为高电平就产生阻塞信号，这个与冒险阻塞是一致的。

> 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）

直观清晰、统一且便于管理。

> 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?

从 DM 获得的数据是一整个字，而写入的时候写入的也是一整个字。
当部分数据更新，即只需要修改一个字节的时候，效率更高；或者数据大小不匹配的时候，按字节读更有效率，这样避免了不必要的内存操作。

> 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?

见上表的”冒险处理的相似性抽象“。这样避免了多余的译码，可以更具有普遍性和适配性。

> 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?

没有遇到指令之间的冒险冲突，因为这些东西在 P5 已经解决。唯一的冲突是乘除模块的阻塞。

> 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。

这是一个测 bne 的程序（发现 PCSel 段没有更新导致的 bug）：

```
ori $t1, 4
ori $t2, 8
ori $t3, 16
ori $t4, 16
bne $t1, $t2, next
nop
sw $t1, 0($t1)
next:
sw $t2, 0($t2)
add $t2, $t2, $t1
bne $t3, $t4, what
nop
sw $t2, 0($t2)
what:
sw $t3, 0($t3)
```

另外我用 C 语言写了一个机器码生成 MIPS 语言的程序，主要是课程平台给出的 testcode 有这种需求……