流水线究竟是个啥？

V1.0@2014.12.2

目录

一、关于流水线理论的通俗解释

1.流水线是如何实现的

Q：何为流水线？我把流水线的理论都理解了，但是该转向实际的代码书写与建模？

（如果你对流水线的设计还没有初步思路的话，请看这里，如果你有了初步的思路与想法，那么可以直接跳过这一步了）

A：要设计流水线，首先要解决的问题是：何为流水线

（1）那么何为流水线？我就不赘述课本上的定义了，课本上的洗衣机的例子也很浅显生动，但是我知道你想要知道的一定不是这个，你想要知道的是：流水线如何在 Verilog 跑起来，对吧？

（2）那么我们就来讲讲用代码是如何跑动流水线的吧。那么首先我们要明白一点，我们的四个流水线寄存器，PC，GPR 都是有 CLK 信号的。或许老师上课并没有强调过这一点，但是这个是我们流水的基础：当时钟信号上升沿时，会触发我们的各级寄存器和 PC，GPR 进行寄存器内部值的更替。这一点是我们实现流水的关键步骤。那么 为什么我们的各级寄存器和 PC，GPR 会有新值输入呢？

（3）那么我们从开始一步步分析一下。所有步骤的第一步开始都应该是以 PC 初始化为开始的，PC 初始化为 0x00003000，开始从 IM 里取出第一条指令，并让它跑入我们的 IF/ID 寄存器的输入口前，就等着被时钟信号触发而被输入。那 IF 级组合逻辑还做了什么呢？它计算出了 PC+4 并且让 PC+4 在 PC 寄存器输入口前等着，等时钟信号一触发就迫不及待地取代原先 PC 的位置成为新的 PC 啦。

要注意上述过程跟时钟信号是无关的，是组合逻辑 ，我们所有的流水级寄存器之间的电路全部都是组合逻辑。

（4）当第一个时钟上升沿到来之时，在 IF 级里，我们在 IF/ID 寄存器输入端口前准备好的 Instr 指令码和等待在 PC 寄存器输入端口前的 PC+4 成功进入了寄存器，取代之前的值成为了新的值。这就是寄存器内部更新值的方式，我们可以把输入理解为需要时钟触发输入，但是输出直接和组合逻辑相连，也算是组合逻辑的一部分。

（5）我是这样理解流水线的：指令类似于小人，流水级寄存器类似于门，而时钟信号上升

沿类似于开启门的动作，我开门前每个门前都当且仅当有一个小人守着准备一开门就冲过去，那我一开门，小人冲过去了，但是由于开启的时间特别短，所以每个人小人只能冲过一扇门，小人们冲过门后继续前冲，但是无奈地发现他自己又被后面的门挡住了。这样其实完成了一个阶段的流水过程。

（6）思考：在读完这些之后是不是觉得自己对流水线的认识有一了一个初步的认知呢？那不妨继续试着思考一下如下的问题：

时钟信号上升沿触发时会影响流水级寄存器和寄存器堆，PC 寄存器当前的输出吗？

如果这些问题你独立思考过了并且有了自己心中的答案，那么你的流水线的初步认知就已经建立起来了，你就已经学会了流水线的基本构造，那么请阅读下面的内容。

2.关于流水线的进阶理解

（1）流水线 CPU 的工作原理有点类似状态机，CPU 的功能部件的运算即组合逻辑可以认为是在瞬间完成的，随后相应数据就到下一级流水线寄存器的输入端，但还没存入寄存器，当下一个时钟上升沿到来的时候这数据会被写入流水线寄存器，同时，只要流水线寄存器的值被更新，那么它之后的功能部件的输出会立即改变。

（2）那么这些话是什么意思呢，首先我们在这里要强调的一点是 **阻塞赋值** 与 **非阻塞赋值** 的区别。从之前的经验我们可以知道，在时序逻辑的 always 块中我们要采用非阻塞赋值的方式（<=）;而在组合逻辑中如果要使用 always，则必须采用阻塞赋值的方式（=）（一般不推荐在组合逻辑中使用 always 块语句）

（3）先谈一下我对两种赋值方式的理解：

阻塞赋值的字面含义是当前的赋值语句阻断了其后的语句，也就是说后面的语句必须等到当前的赋值语句执行完毕才能执行。而且阻塞赋值可以看成是一步完成的，即：计算等号右边的值并同时赋给左边变量。

而非阻塞赋值的字面含义就很清楚了，可以分为两个步骤进行：

①计算等号右边的表达式的值，（我的理解是：在进入进程后，所有的非阻塞语句的右端表达式同时计算，赋值动作只发生在顺序执行到当前非阻塞语句那一刻）。

②在本条赋值语句结束时，将等号右边的值赋给等号左边的变量。

（4）这两种赋值方式有何区别呢？

我理解的阻塞赋值中的输出和输入之间就是一根导线，他俩始终是相等的；而非阻塞赋值中的输出和输入之间的关系就是：当时钟信号上升沿触发**时**，在这个时刻当前寄存器的输出等于上一次本寄存器的输入！（不等于本次本寄存器的输入）这是指令可以流水即在流水级寄存器间传递的根本原因！

（5）这么说的原因在于：在计算非阻塞赋值的右侧表达式值和更新左侧变量值期间，其他的 Verilog 语句，包括其他的 Verilog 非阻塞赋值语句都能**同步**计算右侧表达式和更新左侧变量。那么我们可以这样理解：所有的流水级寄存器先按照上一个状态算出下一个状态的值，全部计算完后我们再赋值给寄存器。这样或许你就能对上面所说的那句话有所理解了，当然也就对这句话有了足够的理解：

寄存器输入值时为时钟触发输入，但是输出直接和组合逻辑相连，也算是组合逻辑的一部分。只要流水线寄存器的值被更新，那么它之后的功能部件的输出会立即改变，那么下一级的流水线寄存器的输入值就是上一级流水线寄存器的输出值经过组合逻辑后的结果！

（6）所以有以下几个需要注意的地方（如果你的时序逻辑电路出现混乱有可能存在着下列原因）

1、带有 posedge 或 negedge 关键字的事件表达式表示沿触发的时序逻辑，没有 posedge 或 negedge 关键字的表示组合逻辑（或电平敏感的锁存器）

2、敏感事件列表中可以包含多个敏感事件，但不可以同时包括电平敏感事件和边沿敏感事件，否则会造成时序逻辑的混乱(因为把组合逻辑和时序逻辑写在一个 always 块里了)，也不可以同时包括同一个信号的上升沿和下降沿。

简而言之，以下的情况不可以发生：

```
always@(posedge CLK or stall or clear)
```

3、组合逻辑如果不考虑门的延时的话当然可以理解为瞬时执行的，因此没有并行和顺序之分，并行和顺序是针对时序逻辑来说的。值得注意的是所有的时序块都是并行执行的。initial 块只在信号进入模块后执行 1 次而 always 块是由敏感事件作为中断来触发执行的。

3.关于延迟槽的理解

何为延迟槽？为什么要有延迟槽的存在？延迟槽对我们有什么代码上的影响？

（1）关于这个问题高老师在大班公告里已经写过了，但是我谈谈我对于这个问题的理解吧。延迟槽（branch delay slots）并不能避免处理器的控制相关的冲突，也就是说并不能使分支预测的准确率更高，但是这是一种降低分支损失的方法。

（2）支持延迟槽设计的意思是：每次必然执行跳转语句或者分支语句后的那条指令！这样的设计可以让我们不再清空我们的延迟槽指令，而是让它执行完毕，所以为了不重复执行延迟槽指令，我们的跳转失败或者跳回时要跳往的地址是 PC+8（因为 PC+4 已经执行过了），但这样产生了的问题也有：如果我延迟槽指令会改变我分支后跳转的语句块内的某些寄存器的值，难道这样也要执行？

（3）我们现在遵循的原则是：延迟槽指令不会与跳转后的语句块内的指令产生任何冲突。**但这并不意味着延迟槽指令不会与你的 jal 指令有数据依赖关系。**当然现在我们遵循的原则需要编译器的配合，目前来说对延迟槽指令的优化处理并不是我们所要解决的事。但是我们现在仍然需要考虑 jal 语句改变$ra 寄存器的值后，延迟槽指令对于$ra 寄存器的读取，该暂停和转发的时候依旧需要暂停与转发，这一点很重要！

（4）可能还有人不太明白 为什么 jal 存入 31 号寄存器的值就需要改成 PC+8，但是为什么其他分支语句不需要动呢？首先要强调的一点是：B 族指令跳转失败时跳往的地址也应该是 PC+8！PC+8 的根本原因不是因为你目的指令的地址变了，而是为了不重复执行延迟槽指令！（如果在自我小测试的时候可以尝试一下写一条有意义的延迟槽语句，比如让某个无关寄存器的值自增 1 等。示例如下：

```
ori $t0,$0,1 #$t0 赋值为 1
ori $t1,$0,5 #$t1 的值为 5
beq $t1,$t2,label #相等时跳转，不相等时跳转失败
addu $t6,$t6,$t0 #延迟槽指令，$t0 值始终为 1
label: j label #跳转成功的话进入死循环
nop
```

设置 B 族指令跳转失败地址分别为 PC+4 和+8，我们可以发现在 PC+4 时我们的$t6 最终值为 2，而在 PC+8 时 我们的$t6 最终值为 1，这表明 PC+4 时延迟槽指令执行了两次。

（5）总而言之，对我们目前的影响只有：

① 修改了 Jal 指令存入$ra 的地址

② 修改了 B 族指令跳转失败时跳往的地址

③ 不再设计清除 B 族指令和跳转指令后的无效指令（不需要在 IF/ID 寄存器上加清除信号了）

4.关于转发机制的理解

（1）一阶数据相关与转发

```
sub $2,$1,$3
addu $12,$2,$5
```

sub 指令在第五周期写回寄存器$2，而 and 指令在第四周期就对 sub 指令的结果$2 提出申请，显然将得到错误的未更新的数据。像这类第 I 条指令的源操作寄存器与第 I-1 条指令（即上一条指令）的目标寄存器相重，导致的数据相关称为一阶数据相关。

sub 指令的结果其实在 EX 级结尾，即第三周期末就产生了；而 and 指令在第四时钟周期向 sub 指令结果发出请求，请求时间晚于结果产生时间，所以只需要 sub 指令结果产生之后直接将其转发给 and 指令就可以避免一阶数据相关。

转发条件作为数据选择器的地址信号，转发条件不成立时，ALU 操作数从 ID/EX 流水线寄存器中读取；转发条件成立时，ALU 操作数取自数据旁路。

转发条件：

④ MEM 级指令是写操作

⑤ MEM 级指令写回的目标寄存器不是$0

⑥ MEM 级指令写回的目标寄存器与在 EX 级指令的源寄存器是同一寄存器

（2）二阶数据相关与转发

```
sub $2,$1,$3
nop
or $13,$6,$2
```

sub 指令在第 5 时钟周期写回寄存器，而 or 指令也在第 5 时钟周期对 sub 指令的结果提

出了请求，很显然 or 指令读取的数据是未被更新的错误内容。这类第 I 条指令的源操作寄存器与第 I-2 条指令（即之上第二条指令）的目标寄存器相重，导致的数据相关称为二阶数据相关。

如前所述，or 指令在第五时钟周期向 sub 指令结果发出请求时，sub 指令的结果已经产生。所以，我们同样采用“转发”，即通过 MEM/WB 流水线寄存器，将 sub 指令结果转发给 or 指令，而不需要先写回寄存器堆。

转发条件：

① WB 级指令是写操作 **或** ME 级指令是写操作

② WB 级指令写回的目标寄存器不是$0 **或** ME 级指令不是$0

③ WB 级指令写回的目标寄存器与在 EX 级指令的源寄存器是同一寄存器

或 ME 级指令写回的目标寄存器与在 ID 级指令的原寄存器是同一寄存器

(3)三阶数据相关与转发

```
sub $2,$1,$3
nop
nop
add $14,$2,$5
```

假设寄存器的写操作发生在时钟周期的上升沿，而读操作发生在时钟周期的下降沿，那么读操作将读取到最新写入的内容。在这种假设条件下将不会发生数据冒险。这就要求流水线中的寄存器具有“先写后读（Read After Write）”的特性。

这类“写操作发生在时钟周期的上升沿，读操作发生在时钟周期的下降沿”的寄存器虽然在理论上是可实现的，但是不适合应用于同步系统，因为它不但影响系统的运行速度，而且影响系统的稳定性，是不可取的。

在这个阶段我们采用寄存器内部转发来实现该功能：

转发条件为：

① RF 接收到的写使能信号为 1

② RF 接收到的目标寄存器不是$0

③ RF 接收到的目标寄存器与接收到的源操作寄存器是同一寄存器

了解到这些之后，我们在调试修正我们的转发通路的时候，只需要单独如此测试就行了，就是预先分类，分类后测试一，二，三阶数据转发是否正确，当然三阶数据转发由于在 RF 寄存器堆内部转发，所以不需要测试太多。调试技巧呆会会提到更多。

（4）关于转发的几点说明：

1、 我已经有了 ME→ID 的转发，为何我在后面还要设置一个 WB→EX 级的转发？这样不是没必要吗？如果我 ME 已经转发到 ID 了，那我之后就顺序传值就好了呀？

A： WB→EX 级的转发是非常有必要的！因为什么？因为我在 WB 级还有一个比 ME 级多出来的值，那就是 DM 的 ReadData，这个数据也会作为寄存器的写入数据！仔细思考一下，如果是这样两条指令：

```
lw $1,0($2)
nop
addu $4,$3,$1
```

如果我光有 ME→ID 级的转发是满足不了这两条指令的需求的！

而 WB 到 ID 级已经有内部转发的支持了，所以 WB→EX 级显得很有必要了！(二阶数据相关与转发)，那么下一个问题来了，为何我不再支持 WB→ME 级的转发了呢？

2、 为何不支持 WB→ME 级的转发？

A： WB→ME 的转发实际上不仅是没有必要的，而且有时候会作为干扰。因为已经有了 ME→EX 的转发，其实只要再过一个周期转发后的 EX 值就会到达 ME 级里，而前序指令也巧好从 ME 走到了 WB 级，这两者是一样的！为什么说它会作为干扰呢？因为我们如果设置了 sw 的 rt 字段寄存器是在 ME 级才收到来自 WB 级的转发信息的，那么以下指令就无法实现了：

```
ori $1,$0,4
Nop
sw $2,0($1)
```

那么新的问题又产生了，我们如果在 EX 级接收转发信息，如果有一条三阶转发没能信息传递到 EX 级的时候数据已经没了，怎么办？

3、 为什么能保证转发到 EX 级时接收信息的完备性？（包括三阶转发的数据都接收）

A： 这是由我们的 GPR 内部转发来保证的！我们 GPR 内部转发不看指令类型，只要你此时寄存器写使能信号为 1，且读寄存器的地址和写寄存器的地址是一致的我就给你新的要写入的值！那我们在 ID 级的时候 sw 指令就完全可以接收到来自 WB 级的转发数据！等到 sw 流水到 EX 级时 sw 所拥有的信息恰恰是转发后的信息！

5.关于暂停机制的理解

关于高老师的工程化建模的方式不太理解？

A： 采用暂停与转发分离的机制。当新指令进入到 IF/ID 级流水线寄存器时，就与 ID/EX 级、EX/MEM 级流水线寄存器的指令检测是否存在冒险，即 IF/ID 级指令所需的寄存器最新数据由 EX 级、MEM 级指令产生，若存在则先暂停，然后再启动，两条冲突指令一起向后运行，当后续指令需要冲突寄存器数据进行运算的同时，供给指令的输出正好产生，此时进行转发。（即每读一条新指令进入 IF/ID 寄存器就要进行一次是否暂停的判断）

关于暂停：

先对指令进行分类，这个分类很关键，分类的根据有：需求级别，供给级别，需求寄存器字段，供给寄存器字段。（高老师在表格里对需求级别是以数字编号的，0 为当前需求，1 为下一级需求，2 为后一级需求）

这么分类之后就可以是否需要暂停和转发进行快捷的判断了。

那么问题来了，我有的需要暂停 1 个周期，有的需要暂停 2 个周期，我怎么实现暂停 2 个周期呢？

（1）如何实现暂停两个周期？

其实我们对于暂停两个周期这种问题考虑的方案就是：暂停一个周期，再暂停一个周期看起来挺傻的，但是确实是这样，我们在判断了 ID 级的指令和 EX 级的指令有冲突时让 ID 级的指令暂停一个周期，然后 EX 级的指令就到了 ME 级，那这时候我们只需要再暂停一个周期就可以达到我们最初的暂停两个周期的效果啦！

（2）暂停需要多少个信号？（在 P5 中）

在 P5 中，我们控制暂停的信号只需要三个，那就是对 PC 寄存器的阻塞信号（也就是使能信号），对 IF/ID 寄存器的阻塞信号（也就是使能信号），对 ID/EX 寄存器的清除信号（也就是控制信号变零）为什么在暂停两个周期时不需要对 EX/ME 寄存器进行清除呢？我们可以考虑如下三种情况：

1、D、M 冲突时，我们这时候阻塞 PC 寄存器和 IF/ID 寄存器，清除 ID/EX 寄存器，可以得到什么样的结果呢？比如现在在 IF/ID ,ID/EX,EX/ME 寄存器里的指令分别为 A，B，C 清除 ID/EX

相当于现在成为了 A，BUBBLE，B，C（后两个流水级寄存器并没有对他们进行操作，正常运转）；

2、D、E 冲突时，我们仍然阻塞 PC 寄存器和 IF/ID 寄存器，现在 IF/ID,ID/EX 寄存器里的指令

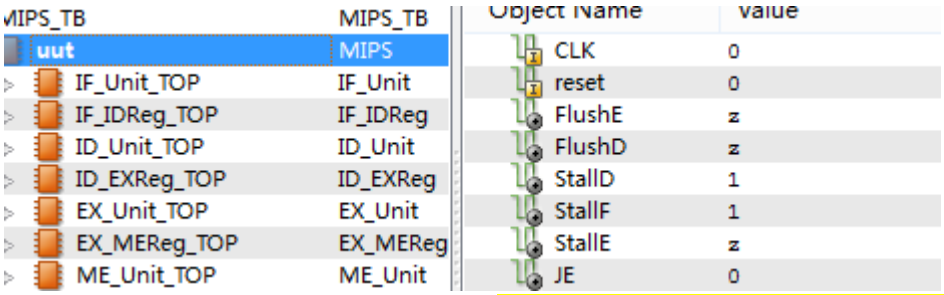
分别为 A，B，清除 ID/EX 寄存器相当于现在成为了 A，BUBBLE，B

3、D、E 冲突后又 D、M 冲突（暂停两个周期），我们现在的结果就是 A，BUBBLE，BUBBLE，B 这三种情况下跟我们的预料结果全部都是一致的，所以我们现在只需要对 StallD 进行选择赋值，再 assign　StallF = StallD ，可以省去很多的麻烦。

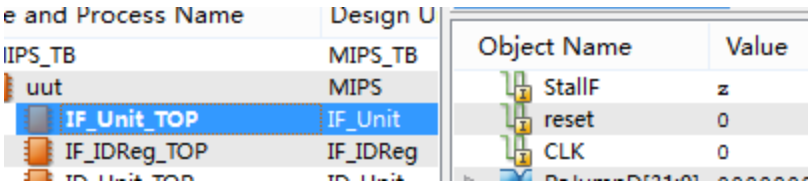
二、调试的一些小技巧：

1、在点击进入仿真后，请观察你的各个模块内部是否出现了高阻态 z 的变量。

比如如下图：



可以看到在顶层文件中我们的 StallF=1，这表明我们在顶层模块中输出 StallF 的模块有效输出了，但是看一下在内部的截图我们就可以知道出现了什么问题



发现在 IF_Unit 模块里 StallF 是 z，这说明我们需要 StallF 的模块并没有有效接收到 StallF 信号，导致这种情况发生的可能有如下几种情况：

1、你在顶层模块里没有为 IF_Unit 指定一个变量对应 StallF

2、你在顶层模块里为 IF_Unit 指定的对应 StallF 的变量名是无效的

3、你在顶层模块里为 IF_Unit 指定的对应 StallF 的变量位宽不对应

综合以上情况可以进行挑错处理，避免大家犯低级错误的第一步骤就是：

进入仿真后首先观察模块内是否有变量包含 z？如果有 z 就要小心，仔细观察在顶层模块中

该变量名是否被正确定义了？

为了有个良好的代码书写风格，我建议大家在模块实例化的时候这样写，比如 Test 内部有变量名为 A，B，在外部要把 C，D 作为该模块的外部变量，按以下风格可以有诸多好处：

```
Test Test_Top (.A(C),.B(D)
                );
```

这样写最大的好处就是不会因为某一变量的未输入导致全部变量的对应混乱，并且位宽不匹配时会报错！

2、关于错误定位与检测

首先观察 DM，GPR 里的数值与你所预想的是否有差别，如果有差别，差别在地址还是数据段？锁定这个之后可以寻找到在 Mars 里与写该数有关的指令，分析该指令的前三条是否与其有数据依赖关系？然后自己手动编写一个只包含该种转发的 Mars 程序去测试是否为该原因。如果不是该原因，请注意检查是否是你的数据通路的问题，这样逐级溯源可以错误定位。

如果是在很多很复杂的 MIPS 指令（比如 P5 测试程序）里，找到错误的数据却无法找到出错的指令，可以尝试以下策略：

1、 对于 store 类指令的错误：store 类指令与前序指令的数据依赖出现的错误无非在于在错误的 DM 地址中存入了正确的数和在正确的 DM 地址中存入了错误的数，只要找到指令和前序的可能产生冲突的指令即可。

2、 对于 Load,R 型指令的错误：这个需要在寄存器中观测即可。

3、 对于 B 族指令和跳转指令的错误：首先在波形图中加入 PC，在 DM 里每次一个周期地跑，一直到你错误数据前的最后一个正确数据，记录此时的 PC 值和运行的时间，将 PC 加入波形图，之后从波形图的 PC 和 MARS 相对应的 PC 地址开始同步运行，观测你 PC 的下一次的值是否和你 MARS 里的 PC 下一步得到的值相同，若不相同，请锁定该地址并检查该指令与其前三条指令的数据依赖性从而缩小范围，继续利用小程序跑。（当然我们现在可以使用我们系大神们改版后的 Mars，不得不说改版后的 Mars 对于各位的调试有很大的帮助！在这里感谢**何书翰，王鹿鸣**，以及**其他为该软件修改贡献力量的大**

神们~该软件在大班群可以下载）

三、常出现的错误原因

（欢迎各位致信补充）

1、 在组合逻辑里出现了 xx 等信号

A： 有可能是未对所有的寄存器进行初始化，（包括流水级寄存器）

有可能是在组合逻辑中使用了 reg 类型变量定义并且用连续赋值 reg。

2、 在所有指令执行完后依旧进行读写操作等

A： 这种情况大多数是因为把寄存器变成了锁存器导致的，而导致这一情况往往是因为：

在流水级寄存器中 if 语句并没有最终的 else 语句（我们最终的 else 应该是将寄存器清零的）Case 语句没有相应的 default 或者 default 并不是置零语句

更多关于避免引入锁存器的原理和方法请参见 ISE 帮助文档

3、有关延迟槽指令的寄存器堆内的值均不正确

A：这种情况往往是因为 B 族指令失败跳回地址写错了而导致的，如果对此有疑问请点击回到延迟槽的部分进行查阅

4、旁路与冒险控制单元都写对了，但是暂停和清除很混乱

A： 这种情况有可能是因为在流水线寄存器中书写了如下格式的语句：

```
always@(posedge CLK or StallF or FlushD)
```

请注意 always 语句敏感列表里或者是 电平高低敏感 或者是 边沿敏感 事件，一般是不允许出现两者混用的情况的！

5、在使用助教老师的测试程序时会出现 DM 被同一个数写满的情况

A：这种情况是因为没有考虑 JAL 作为前序指令的暂停与转发的情况而导致的，请注意在旁路中要考虑 JAL 作为前序指令的转发与暂停机制！

6、在控制器那里输出的控制信号是对的，但在顶层模块看发现它变成了 1x0 类似的东西？

A： 请检查你的顶层模块中，看是否有定义成 reg 的变量？如果有，且它是一个模块的输出，那么请将 reg 改成 wire，具体原理在 ISE 帮助文档里有讲，这里再重复解释一下：输入端口可以由 wire/reg 驱动，但输入端口只能是 wire；输出端口可以是 wire/reg 类型，输出端口只能由 wire 驱动。

意思是：驱动的定义时：在顶层模块定义的变量对于子模块来说叫做驱动变量（我是这么理解的），顶层模块定义的 reg 是无法作为子模块的输出变量的（就像刚才会导致 1x0 这种情况产生），所以注意 顶层模块的内部变量最好全部定义为 wire 类型

7、B 族指令和 J 指令跳转到的 PC 不正确？（J，B 族指令都没错）

A：请检查一下你的 PC 初始值是否为 **32** 位的 0x0000_3000
如果你在 PC 模块内使用的是 30 位的 PC，请在二进制数去掉两个零后作为 PC 的初始值！

四、一点小建议

1、关于数据通路的构建

由于模块数量众多，在搭建数据通的时候如果直接由底层模块拼接会相当麻烦，而且层次不好看。所以最好是先一级一级将 IF、ID 等模块搭建完成，再在顶层搭建最终的数据通路。

2、关于控制器译码方案

强烈建议采取原先类似与或阵列的方式进行译码，这种方案在调试和增加指令的时候相当便捷。

3、采用分布式译码方案

高老师在 PPT 里也说过了，分布式译码的复用性很强，而且可维护性很好，所以为了调试时能少一些改动，建议采用分布式译码方案

五、鸣谢

本文档由 刘乾 与 仇栋民 共同撰写，如果你发现了问题或者有更好的建议，欢迎致信 qianlxc@126.com （写信时请注明你的名字与学号），如果你想成为一名帮助别人的公益人士，同样欢迎你的加入~，若你平时不喜欢使用邮箱，欢迎骚扰我的 QQ：372899855

在此处鸣谢以下几位同学提出的解决过或未解决的问题：

王晨玥、杜正远