**Lab 6报告**

马迪峰 2021K8009929033

饶嘉怡 2021K8009929005

范子墨 2021K8009929006

箱子号 5

一、实验任务（10%）

本次实验分为两部分任务，实践任务10为在已有单发射五级流水线的基础上添加算术逻辑类指令，包括slti、sltui、andi、ori、xori、sll、srl、sra、pcaddu12i，和乘除运算指令，包括mul.w、mulh.w、mulh.wu、div.w、mod.w、mod.wu、div.wu。实践任务11在任务10的基础上添加转移指令包括blt、bge、bltu、bgcu，及访存指令包括ld.b、ld.h、ld.bu、ld.hu、st.b、st.h。

该实验的正确性同样由基于 trace 比对的调试辅助手段进行验证，通过将 myCPU 中的 WB 阶段的 PC 值与 golden\_trace 的 PC 值进行比对，如果不一致则立即报错并且停止仿真。在仿真正确后进行上版验证。

二、实验设计（40%）

（一）总体设计思路

对于算术逻辑运算指令，要实现的指令都尽量复用了原有指令的数据通路和控制信号。这部分指令需要在译码阶段进行对应数据通路的更改，在后续流水中只需复用原alu\_result的数据通路鄂控制信号。

在乘除运算指令方面，

对于转移指令，与原有的转移指令类似，只需添加跳转条件判断。

在访存指令方面，首先需要在译码阶段增加load\_op和store\_op，并在发起访存请求前和后分别根据访存指令进行数据的修改使数据符合指令需求。

实验中为了使时序结果更好，尽量使用与或逻辑，而非三目运算符。

（二）重要模块1设计：算术逻辑运算类指令

1. 工作原理

增添了9条算术逻辑运算类指令，所有指令均可以复用原有的数据通路和控制相关，只需对操作数进行选择，因此只需在译码阶段进行更改。

1. 功能描述

具体来说，slti可以复用slt的数据通路，sltui可以复用sltu的数据通路，andi可以复用and的数据通路，ori可以复用or的数据通路，xori可以复用xor的数据通路，sll可以复用slli的数据通路，srl可以复用srli的数据通路，sra可以复用srai的数据通路，pcaddul12i可以复用add的数据通路。

而slti、sltui、andi、ori和xori的两个操作数分别为rj寄存器中值和立即数，其中andi、ori和xori新增了一种类型的操作数为ui12，即零扩展立即数。

assign need\_ui12  =  inst\_andi | inst\_ori | inst\_xori;

因此对于立即数的选择需要增添对应ui12的判断。以及增添操作数2是否来自立即数的判断和操作数1是否来自PC的判断。

assign imm =                  src2\_is\_4 ? 32'h4                                    :

                              need\_si20 ? {i20[19:0], 12'b0}                       :

  /\*need\_ui5 || need\_si12\*/   (need\_ui5 || need\_si12) ? {{20{i12[11]}}, i12[11:0]} :

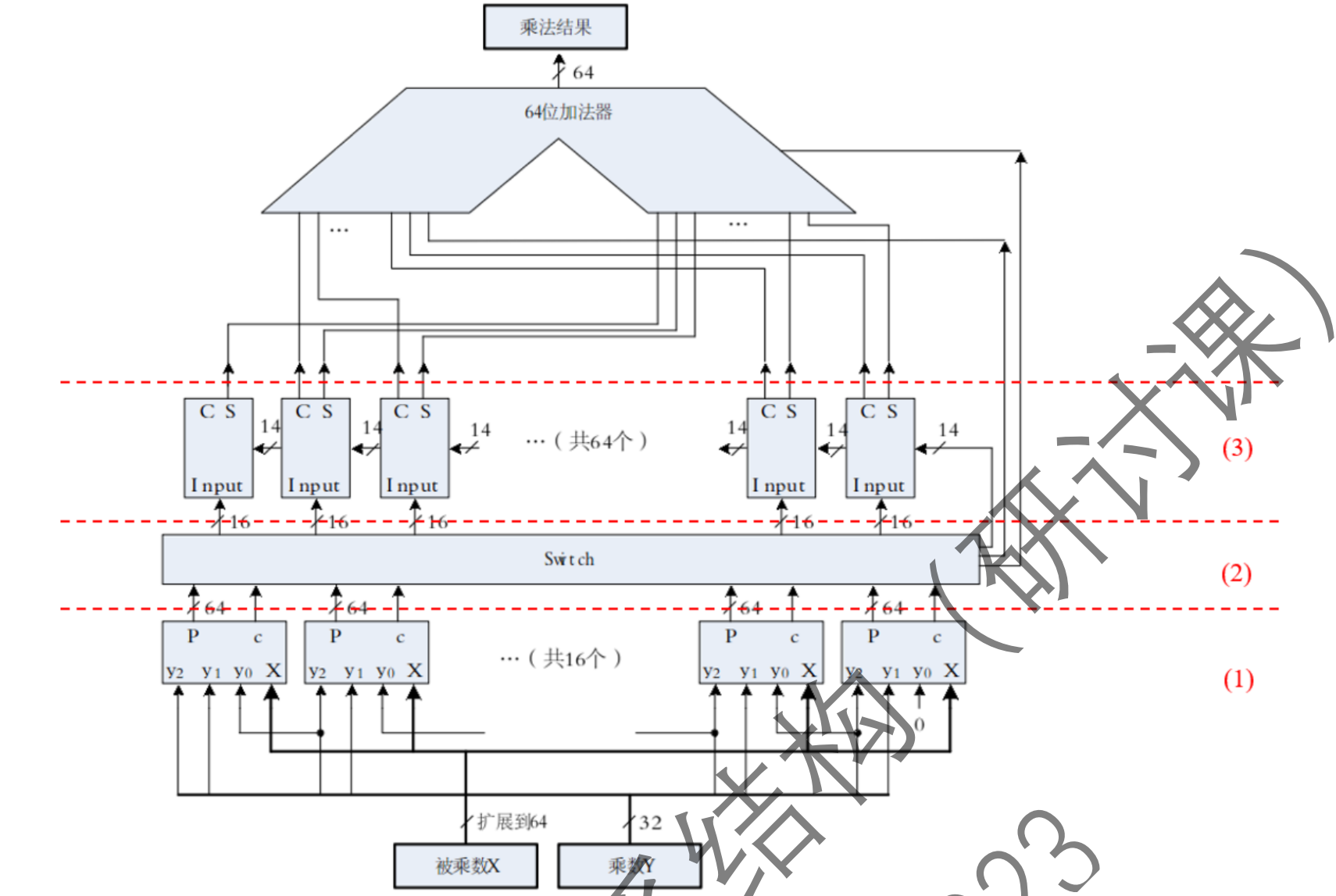
                              {20'b0, i12[11:0]};

1. 重要模块2设计：乘法器

1、工作原理

1. 乘法器

乘法器采用讲义推荐的两位Booth编码+华莱士树的设计原理，如下图所示。



本次实验需要实现的乘法指令有三条：

MUL.W：进行有符号乘法，将低32位作为结果。

MULH.W：进行有符号乘法，将高32位作为结果。

MULH.WU：进行无符号乘法，将高32位作为结果。

由此可知，该乘法器需要能够实现有符号和无符号乘法，因此可以将操作数符号拓展为34位，计算得到68位结果。这里之所以拓展成偶数位，是为了配合booth两位乘法。

该乘法器大致分为四部分：生成部分积、转置、华莱士树、64位加法器。其中转置和64位加法器在代码设计阶段都好实现，主要难点在生成部分积和华莱士树。该实验中我们将生成部分积与华莱士树分别封装成了模块。

在生成部分积模块中，用输入的三位乘数判断要对被乘数进行的操作，逐位算出该部分积。

在华莱士树模块中，对17个1bit数，使用六层华莱士树将加数减少为2个1bit数相加，得到17个1bit数相加的结果。

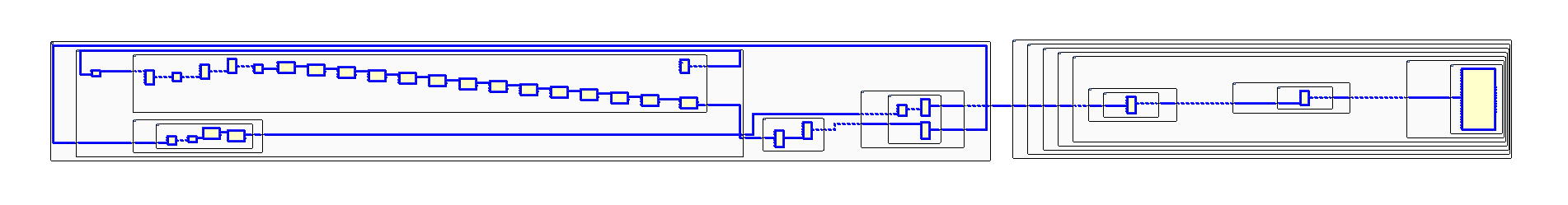
只需在booth乘法顶层模块中例化17个生成部分积模块，算出17个部分积；将17个68bits数转置成68个17bits数；再例化68个华莱士树模块，将每个17bits数送入一个华莱士树，算出17个1bit数的和；最后，将计算结果送入64位加法器即可。

1. 切分流水

该模块最难的部分在于如何切分流水使得时序更好，切成几拍和在哪里切是最主要的问题。我组最后采用的方案是：

* + - 1. 切成两拍，在MEM级拿到结果，不阻塞，只在冲突时阻塞一拍；
      2. 在64位加法器之后切分流水，即第一拍算出乘法结果，将结果存入寄存器送到第二拍。

之所以选择在64位加法器之后切分流水，是因为我们的slack违约最严重的关键路径，是“乘法器的结果前递回ID级的跳转指令，计算跳转地址之后送入指令RAM”，所以如果把切割流水级改在乘法器的结尾，时序会好很多。



尝试最佳的切分流水级的过程会写在实验过程中，在此不赘述。

1. 乘法模块外部处理

添加了两级流水乘法器之后的CPU大致如下图所示。之所以选择将乘法器直接例化在EXE模块，而不是例化在alu模块内部，就是因为乘法器是两级流水的，而alu内部是组合逻辑，如果将乘法器模块例化在alu模块里，需要给alu也切分出两级流水，而这是没有必要的。

在EXE模块里例化乘法器之后，需要在乘法器模块外部添加的操作有：将alu的源操作数位拓展至34位之后送入乘法器作为源操作数，将乘法器模块计算得到的68位结果送入MEM级，在MEM级中根据从ID级传来的alu\_op\_reg选择高32位或低32位作为乘法结果。将final\_result改为alu、乘法器、数据RAM三者结果的三选一逻辑。

由于乘法的结果在MEM级才能得到，因此乘法的前递操作与ld指令相同。（但我的乘法器在EXE其实就得到结果了，感觉是不是能不阻塞，直接在EXE拿到z\_temp前递）如果在EXE时与ID级的指令冲突，就将ID级的指令阻塞一拍，之后再将MEM级的结果前递回ID级。

2、接口定义

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| clk | IN | 1 | 时钟信号 |
| x | IN | 34 | 符号拓展的34位被乘数 |
| y | IN | 34 | 符号拓展的34位乘数 |
| z | OUT | 68 | 68位乘法结果 |

1. 功能描述

输入信号来自EXE级，在EXE和MEM两拍计算，输出信号于MEM级被获取。如果乘法指令处于EXE阶段时与当前处于ID阶段的指令冲突，将ID阶段的指令阻塞一拍，之后前递。

1. 重要模块3设计： 除法器

1、工作原理

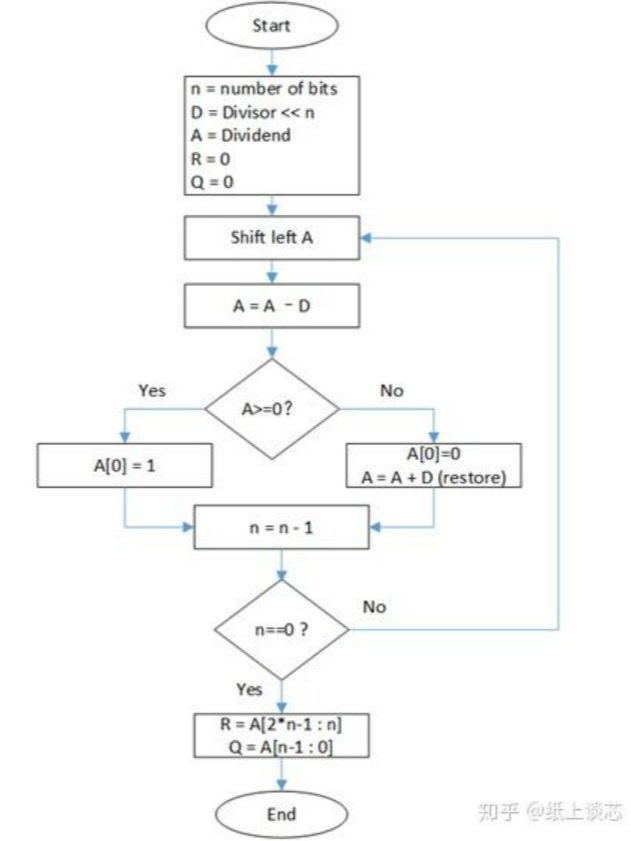
除法器模块设计采用的是恢复余数法的思路，需要完整的33拍才能计算出结果

2、接口定义

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| clk | IN | 1 | 时钟信号 |
| resetn | IN | 1 | 复位信号 |
| div\_en | IN | 1 | 除法器使能信号 |
| sign | IN | 1 | 除法器操作类型，为1 代表有符号除法否则为无符号 |
| divisor | OUT | 32 | 被除数 |
| dividend | OUT | 32 | 除数 |
| result | OUT | 64 | 运算结果，高32位为商，底32位为余数 |
| complete | OUT | 1 | 除法器完成信号 |

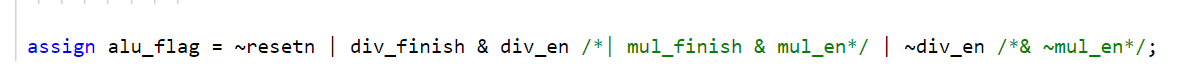
2、功能描述

恢复余数法的流程图如下：



恢复余数法，每次都需要判断结果寄存器是否小于0，再做是否恢复A寄存器的操作；最小需要33个时钟才能计算商和余数；利用cunter信号进行计数，当除法器开始工作且操作数有效时，计数开始，技术达到33时代表除法运算结束，complete信号拉高向外输出到ALU模块表明除法器已算得结果，不必在阻塞可以继续向后输出运算结果。

除法器设计中遇到的问题主要在于结果的保持和在alu中的例化，因为我们的设计中并未将乘法器例化至ALU中，所以ALU发生阻塞时只会是除法器运算导致，需要在alu模块中中引入alu\_flag信号标明其运算是否完成。



除法器的其余数据通路与其他在alu完成的运算基本一致，不需要做其他改变。

（五）重要模块4设计：转移类指令

1. 工作原理

由于本次跳转指令所用的数据通路于之前的一致，因此只需要增加针对于这次新增指令的跳转条件判断，而不需要更改寄存器数据是否有效等部分。以及本组采用直接在译码阶段增加加法器进行判断，而不是复用alu。

1. 功能描述

assign {cout, cout\_test} = {1'b0, rj\_value} + {1'b0, ~rkd\_value} + 1'b1;

assign rj\_lt\_rd = rj\_value[31] ^ ~rkd\_value[31] ^ cout;

assign rj\_ltu\_rd = ~cou.t;

首先对两个需要比较的操作数进行减法，具体来说是rj\_value加上rkd\_value的补码，而后将减法结果赋值为cout\_test，而进位赋给cout。对于有符号数的两个数，如果。对于无符号数，如果有借位cout为0，即小于，无借位cout为1，即大于等于。

而后对是否跳转的信号进行补充。

assign br\_taken = (   inst\_beq  &&  rj\_eq\_rd

                   || inst\_bne  && !rj\_eq\_rd

                   || inst\_blt  && rj\_lt\_rd

                   || inst\_bge  && !rj\_lt\_rd

                   || inst\_bltu && rj\_ltu\_rd

                   || inst\_bgeu && !rj\_ltu\_rd

                   || inst\_jirl

                   || inst\_bl

                   || inst\_b

                  ) && ds\_valid && ds\_ready\_go;

（六）重要模块5设计：访存类指令

1. 工作原理

对于load型指令，定义load\_op，由于需要存到寄存器中的指令需要在MEM阶段才能读出，因此load\_op需要从ID阶段译出后一直传递到MEM阶段后进行赋值。而对于store指令，定义store\_op，虽然在ID阶段就已经可以进行store数据的选择，但考虑到组合逻辑路径可能过长导致时序不好，因此放在EXE阶段进行赋值。

1. 功能描述

load型一共五种指令，因此load\_op位宽为5位并分别赋值。

assign load\_op[0] = inst\_ld\_b;

assign load\_op[1] = inst\_ld\_h;

assign load\_op[2] = inst\_ld\_w;

assign load\_op[3] = inst\_ld\_bu;

assign load\_op[4] = inst\_ld\_hu;

而后传递到MEM阶段后，将五种load指令分为三种，lb\_data，lh\_data和mem\_result，对于在结果中填入不同长度的mem\_result，其中前两种通过对zero\_ext的赋值来进行零扩展或者符号扩展的判断，可以节省资源。

assign zero\_ext = ~(mem\_load\_op[3] | mem\_load\_op[4]);

assign lb\_data = {32{ld\_sel[0]}} & {{24{mem\_result[7] & zero\_ext}}, mem\_result[7:0]}

            | {32{ld\_sel[1]}} & {{24{mem\_result[15] & zero\_ext}}, mem\_result[15:8]}

            | {32{ld\_sel[2]}} & {{24{mem\_result[23] & zero\_ext}}, mem\_result[23:16]}

            | {32{ld\_sel[3]}} & {{24{mem\_result[31] & zero\_ext}}, mem\_result[31:24]};

assign lh\_data = {32{ld\_sel[0]}} & {{16{mem\_result[15] & zero\_ext}}, mem\_result[15:0]}

               | {32{ld\_sel[2]}} & {{16{mem\_result[31] & zero\_ext}}, mem\_result[31:16]};

最后对写入ld\_data的数据进行选择。

assign ld\_data = {32{mem\_load\_op[0] | mem\_load\_op[3]}} & lb\_data

               | {32{mem\_load\_op[1] | mem\_load\_op[4]}} & lh\_data

               | {32{mem\_load\_op[2]}} & mem\_result;

而store指令共三种指令，因此store\_op为三位并分别赋值。

assign store\_op[0] = inst\_st\_b;

assign store\_op[1] = inst\_st\_h;

assign store\_op[2] = inst\_st\_w;

在EXE阶段，首先对alu计算出的地址后两位对要写入的字节进行多路选择，而后针对不同的store指令进行赋值。再将要写入的数据进行扩充到32位写入数据种，方便选择时可以直接选到所需要的部分。

assign st\_strb = {4{store\_op\_reg[0]}} &  st\_sel

               | {4{store\_op\_reg[1]}} & (st\_sel[0] ? 4'b0011 : 4'b1100)

               | {4{store\_op\_reg[2]}} &  4'b1111;

assign st\_data = {32{store\_op\_reg[0]}} & {4{rkd\_value\_reg[7:0]}}

               | {32{store\_op\_reg[1]}} & {2{rkd\_value\_reg[15:0]}}

               | {32{store\_op\_reg[2]}} & rkd\_value\_reg;

三、实验过程（50%）

（一）实验流水账

## exp10实验流水账

* 2023/10/13 19:00 开始exp10实验，确定分工
  + 马迪峰负责除法模块和数据通路的设计
  + 饶嘉怡负责乘法模块和数据通路的设计
  + 范子墨负责其余逻辑运算的设计
* 2023/10/14 10:00 完成除法模块的设计和测试
* 2023/10/14 15:00 完成乘法模块的设计和测试
* 2023/10/16 10:00-22:00 完成调试和debug >.<
* 2023/10/16 22:00 完成exp10实验仿真，但未通过实现。
* 2023/10/17 8:00 开始exp11。

## exp11 实验流水账

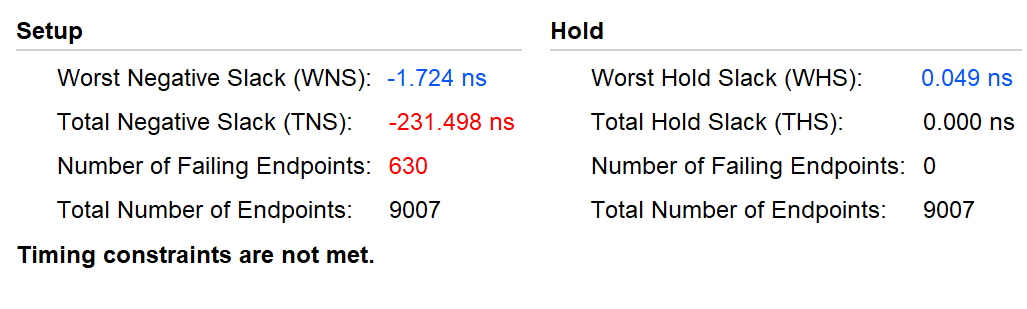
* 2023/10/17 10:00 完成exp11实验，开始debug
* 2023/10/17 19:00 exp11调试完成，开始综合与实现
* 2023/10/23 17:00 上板测试通过，开始撰写实验报告
* 2023/10/23 22:00 实验报告撰写完毕，开始exp12分工

（二）错误记录

1、错误1：乘法器时序违约过多

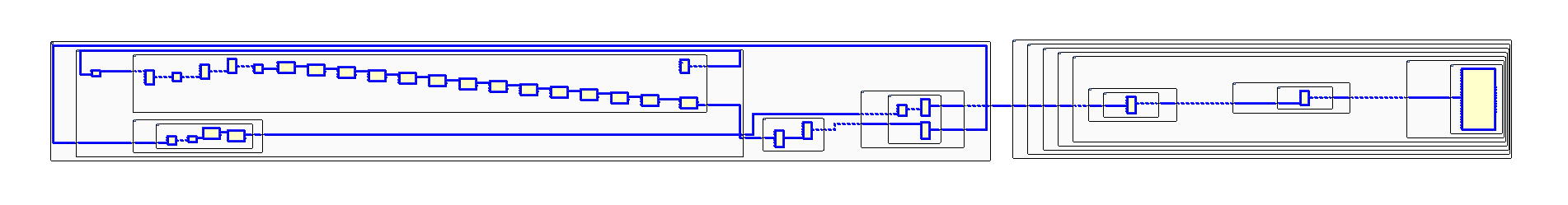
（1）错误现象

乘法器模块级仿真通过，将其嵌入整个CPU的代码后，仿真通过。但综合实现之后的时序报告中，可以看到有高达630条违约的关键路径，WNS违约值高达1.724ns。

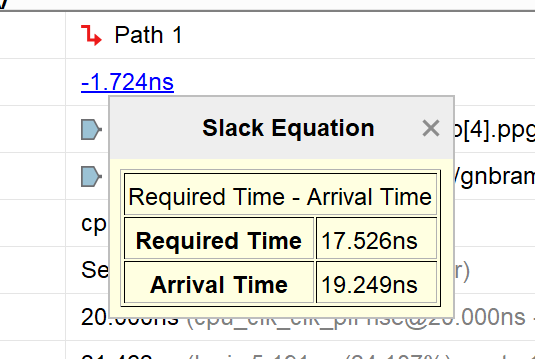


（2）分析定位过程

打开Design Timing Summary，看到Setup时间违约最高的关键路径，查看其Schematic原理图，如下。

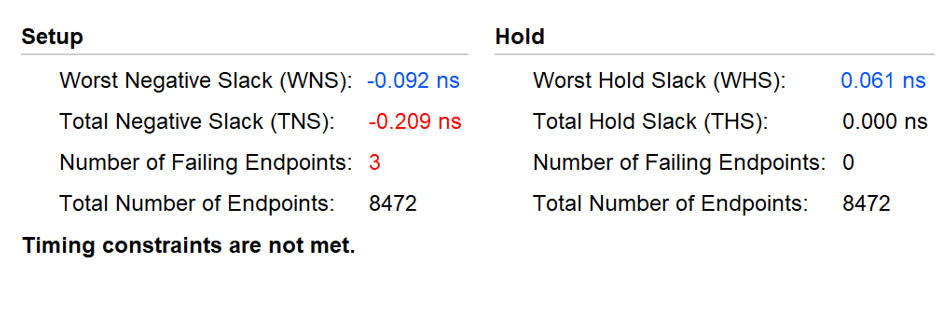


可以看到，这是“用华莱士树算出乘法结果，前递给ID级的跳转指令，去算跳转地址送入指令RAM”的路径。由于其组合逻辑路径过长，难以在建立时间开始之前计算完成（也就是一个时钟周期内难以完成）。如下图所示，Arrival Time大于Required Time，也就是说计算结果到达的时间晚于要求的时间。



鉴于这条路径中华莱士树所占的路径过长，很自然地能够想到，如果在乘法器的结尾切一个流水级，这条关键路径就能短很多，也许就能解决这个时序问题。

现在就需要考虑，是把乘法器改成三拍，还是仍旧保留两级，把切流水的地方从部分积和华莱士树之间改到乘法器结尾。鉴于三拍需要增加阻塞，修改的地方过多，因此我先尝试了后者。再次综合实现，得到时序报告，发现违约路径已经减到只剩3条，WNS减小到-0.092ns，已经不存在和乘法器有关的违约路径，说明乘法器已经修改到对时序来说的最佳状态。



（3）错误原因

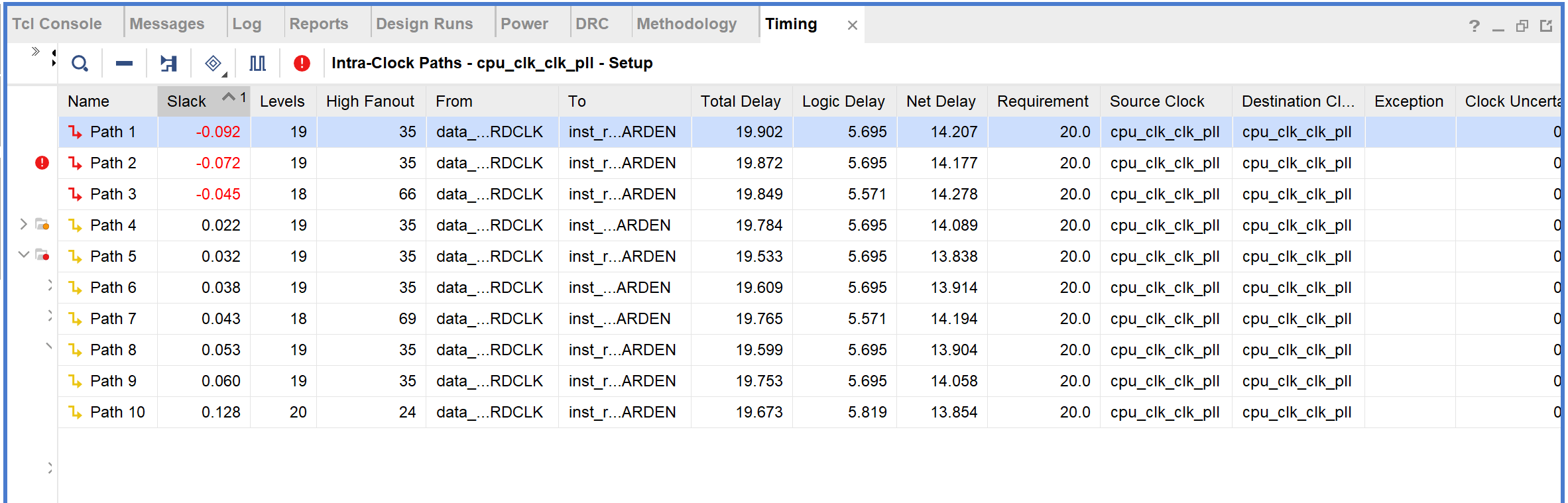
“用华莱士树算出乘法结果，前递给ID级的跳转指令，去算跳转地址送入指令RAM”这条关键路径过长。

（4）修正效果

时序报告中，违约路径只剩3条，WNS减小到-0.092ns，已经不存在和乘法器有关的违约路径。

1、错误1：仍然存在三条与前递有关的违约路径

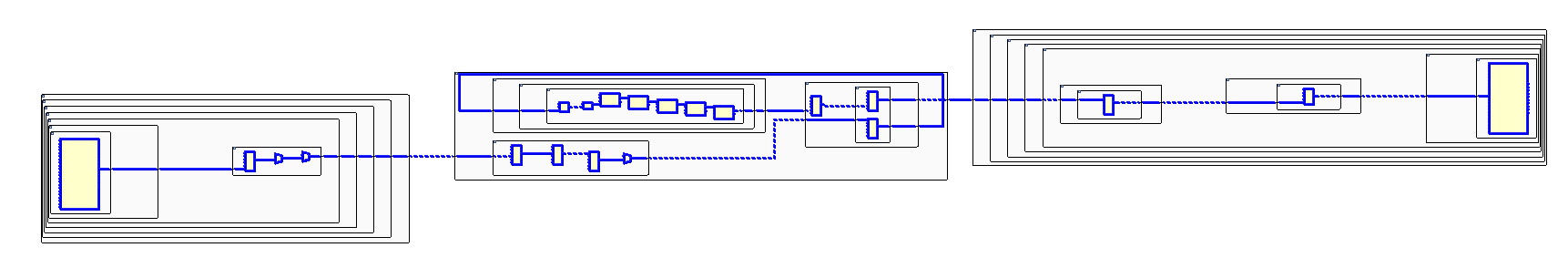
（1）错误现象



修改完上一个错误之后，前面提到仍然剩三条关键路径是违约的。

（2）分析定位过程

仍然从Design Timing Summary中找出违约最严重的关键路径的Schematic，只要在关键路径上合适的地方切流水，就能够解决这个问题。



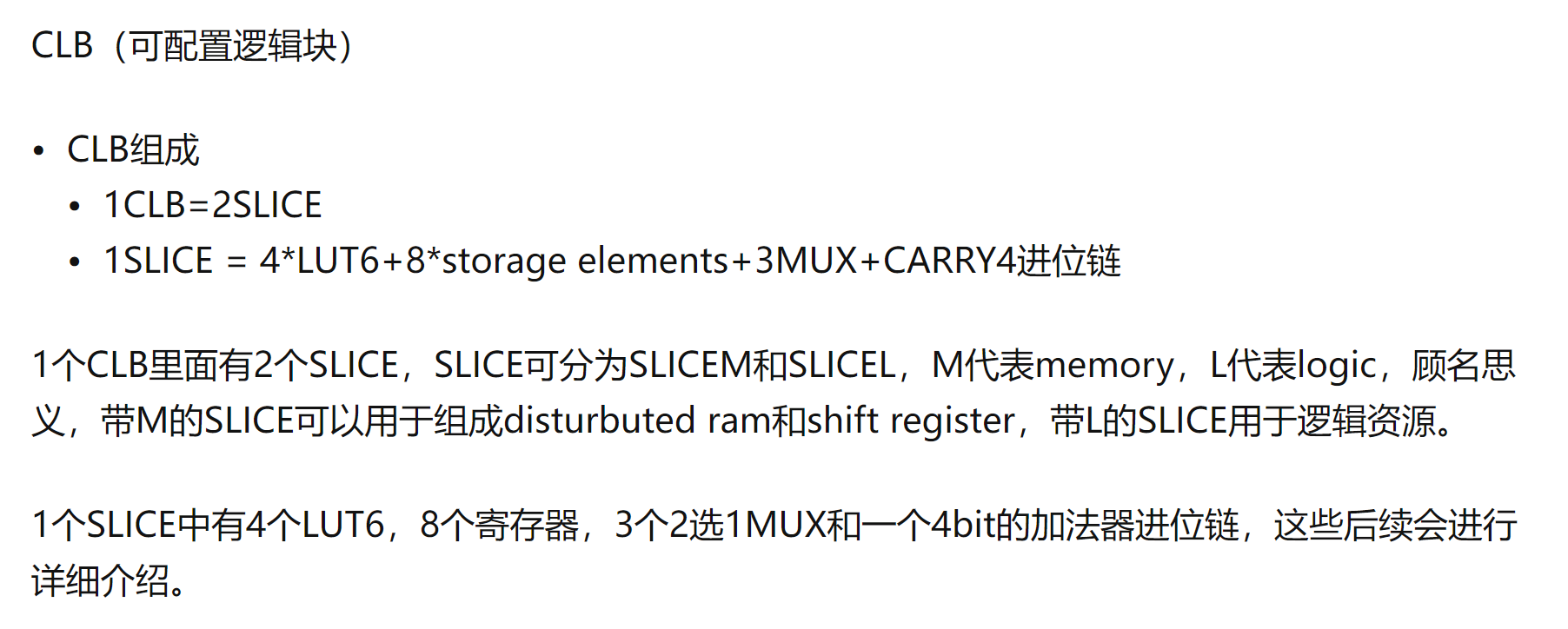
但是，从原理图上看到数据经过的组合逻辑路径是“dataRAM→MEM→ID(RF)→EXE(alu div)→ID(RF)→instRAM”。这条数据通路很像是：ld指令从数据RAM取回数据之后送给MEM，MEM将结果前递给ID的寄存器堆，某个数据从ID传递到EXE并进入alu中的乘法器进行计算，再次前递给ID的寄存器堆，ID将跳转地址送给指令RAM。但这听起来就很不合理，因为

①前递回ID不需要进寄存器堆；

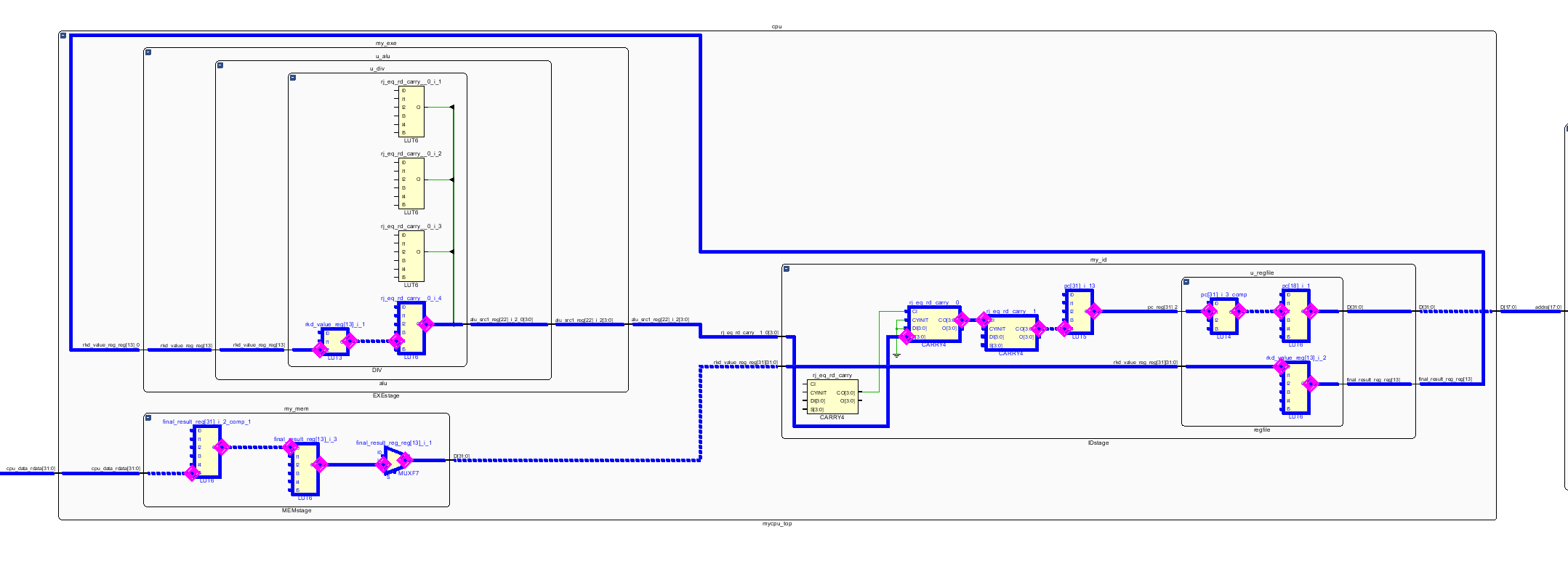
②不会有两个前递在同一个时钟周期内发生。

这说明，这条逻辑路径并不是表面看起来的这样，只能打开完整的Schematic图去探寻真正的路径。

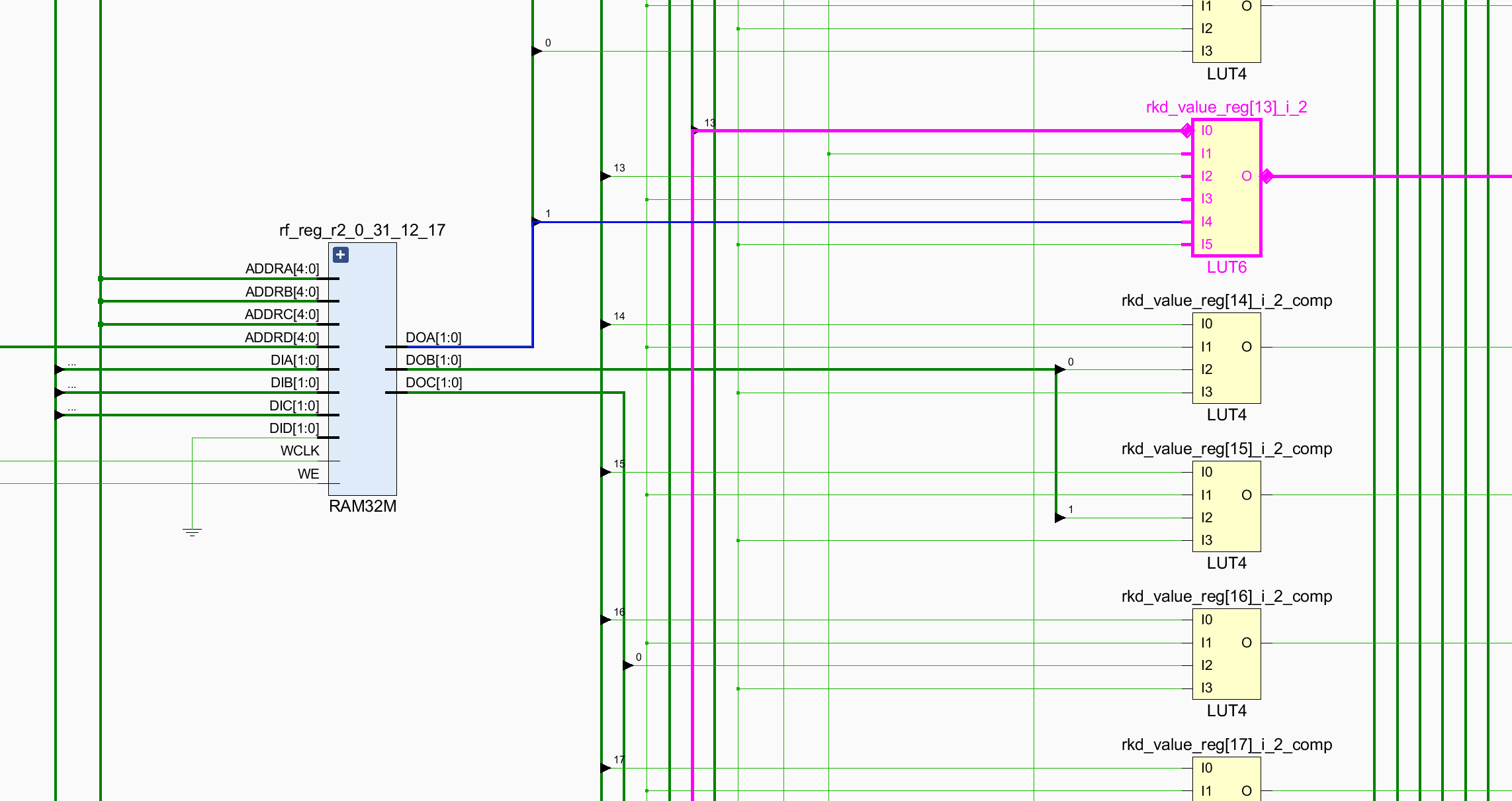
经过调研，发现FPGA的内部的结构单位是可配置逻辑块CLB，它由LUT6、寄存器、MUX、加法器进位链组成。也就是说，在综合实现的时候，不会简单地布局布线成零散的逻辑门，而是将代码中变量拆开成零散的位，去利用这些可配置逻辑块。也就是说，综合实现可能会将某些变量的某些位拉进某个模块里，但实际上代码里并未如此设计。因此，可能会造成，Schematic图虽然显示数据通路进了某个模块，但代码里的数据其实并没有进到我们写的模块里的“假象”。



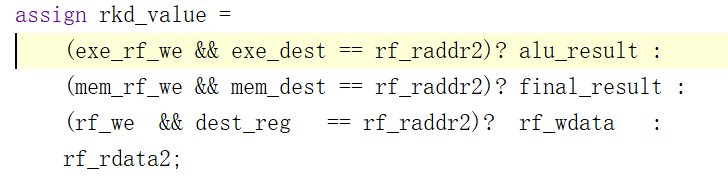
有了这个认知，打开Schematic全图，再次追踪数据通路的走向，就能解决之前的很多疑问。



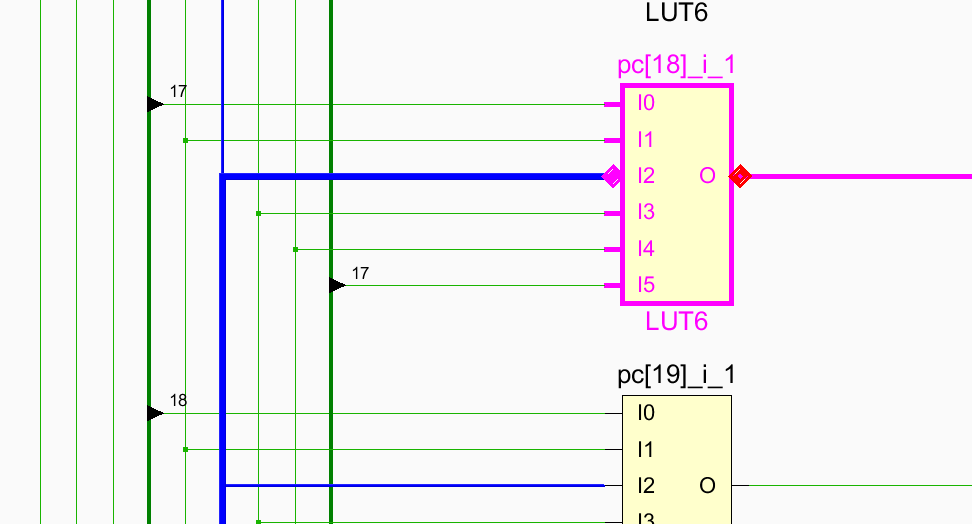
1. 路径第一次进ID的时候，其实并没有进寄存器堆。路径看似进了寄存器堆，是因为LUT6的一个输入是从RF出来的，这可能是综合实现的时候为了在FPGA上布局布线作出的安排。如下图，粉色的是关键路径，蓝色的是从寄存器堆来的输入。



1. 路径其实并没有进入EXE模块，更没有进入alu和除法器。进了除法器之后的第一个模块，除了rkd\_value之外，还有一个叫quotient\_reg的输入，我猜这个叫quotient\_reg的输入就是为什么信号莫名其妙拉进了除法器的原因。进除法器之后的第二个模块，除了rkd\_value之外，还有一个输入信号叫exe\_gr\_we，这是为了判断冲突从EXE拉回ID的信号。所以很自然地就能联想到，虽然我写代码的时候把exe\_gr\_we改成了exe\_rf\_we拉回了ID，但在FPGA上布局布线的时候处于某些考虑，直接带着rkd\_value（这应该也不是真的rkd\_value，而是rkd\_value生成之前的某个中间信号）进了EXE，甚至还进到了除法器模块里，“舍近求远”地来找exe\_gr\_we，最后才算出来真正的rkd\_value，再回到ID。在schematic的模块上右键点“go to source”，回到的也是下图这部分，根本不是EXE，说明该猜想很正确。也就是说，布局布线时把rkd\_value的生成逻辑放到了除法器里。



1. 路径第二次进ID的时候，也并没有进寄存器堆。寄存器堆里的合格LUT6模块，输入信号有br\_zip, pc, ds\_valid,seq\_pc，很像选择下一拍的pc的逻辑，go to source之后也确实来到这里了。但这个逻辑其实没有道理会在寄存器堆里。我点了它的每一个输入信号，都是直接或间接从RF的外面进来的，它的输出信号也会直接出RF。所以相当于布局布线时出于某些考虑，把选下一拍PC的逻辑从IF模块挪进了ID的寄存器堆模块里。





因此，这条关键路径实际上是dataRAM→MEM→ID→IF→instRAM。它做的事情就是，从数据RAM取回ld的结果到MEM，算出final\_result，前递回ID，得到rkd\_value，又算出br\_target传给IF，在IF里算出nextpc，送给指令RAM。仔细想想这些事情确实是在一拍里完成的。

也就是说，这个过长的关键路径是从MEM前递造成的，如果要解决，要做的事情就是换成阻塞，但这势必又会影响流水线效率。目前上板是可以通过的，因此我们决定暂时不修改。

（3）错误原因

从MEM级前递回ID的逻辑的关键路径过长。

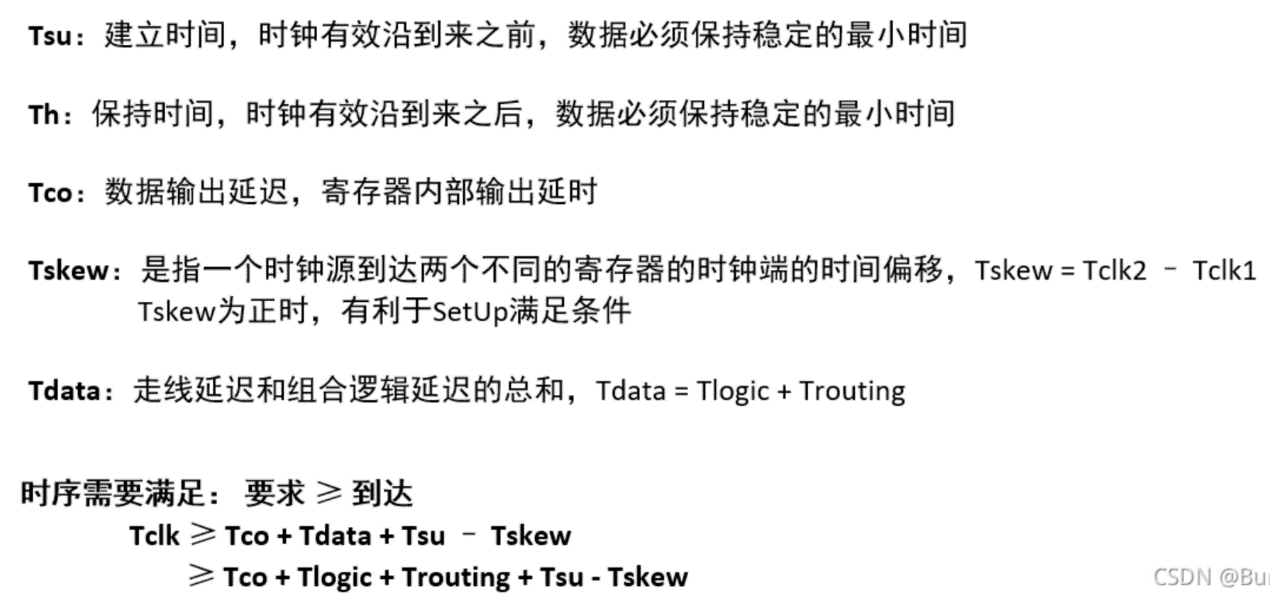
（4）修正效果

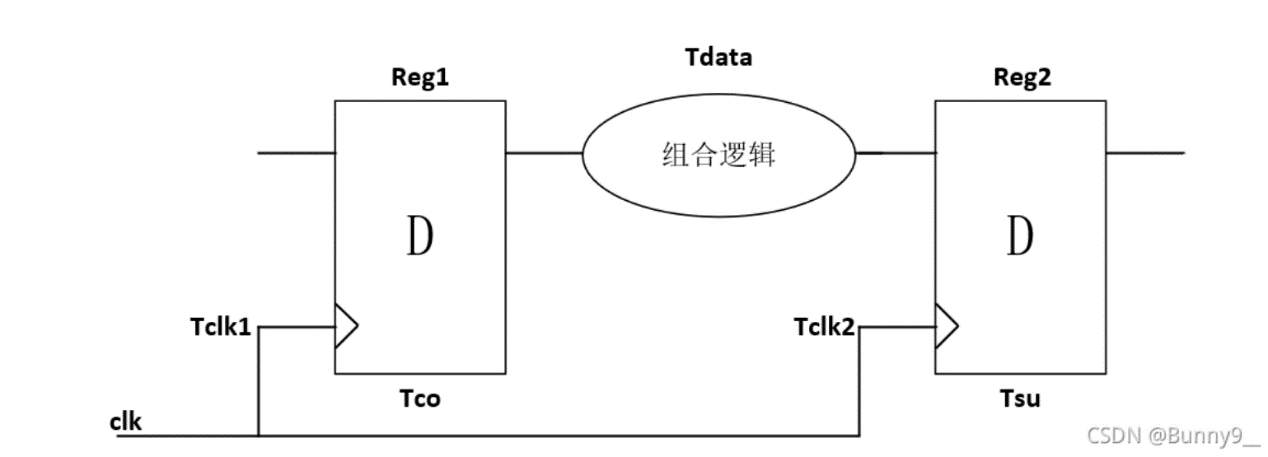
决定不予修改。

（5）归纳总结

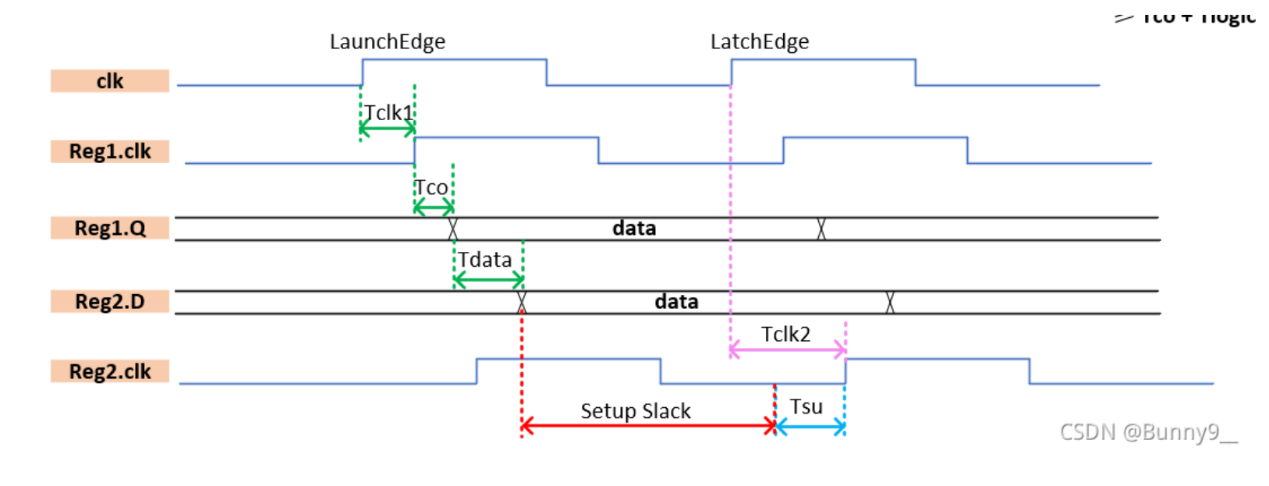
在解决时序违约的问题时，我们由于要阅读时序报告，调研了时序相关的内容。

复述一下时序的基本概念：

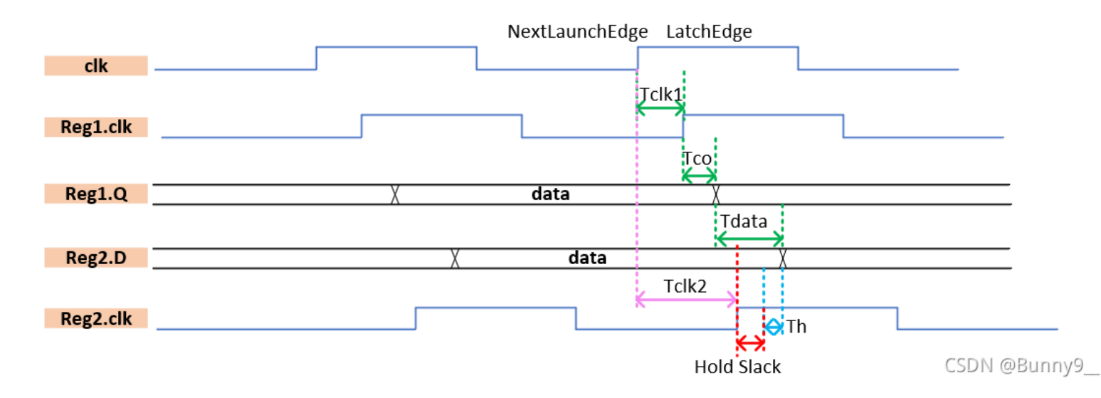


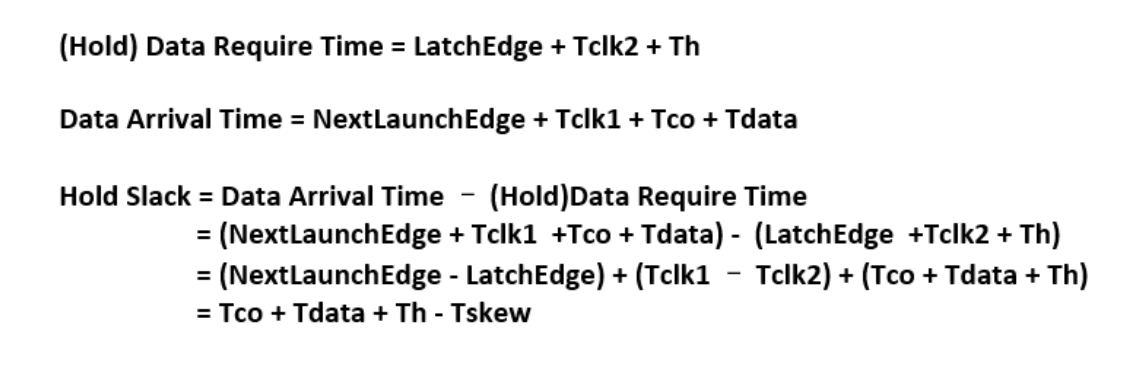


建立余量：

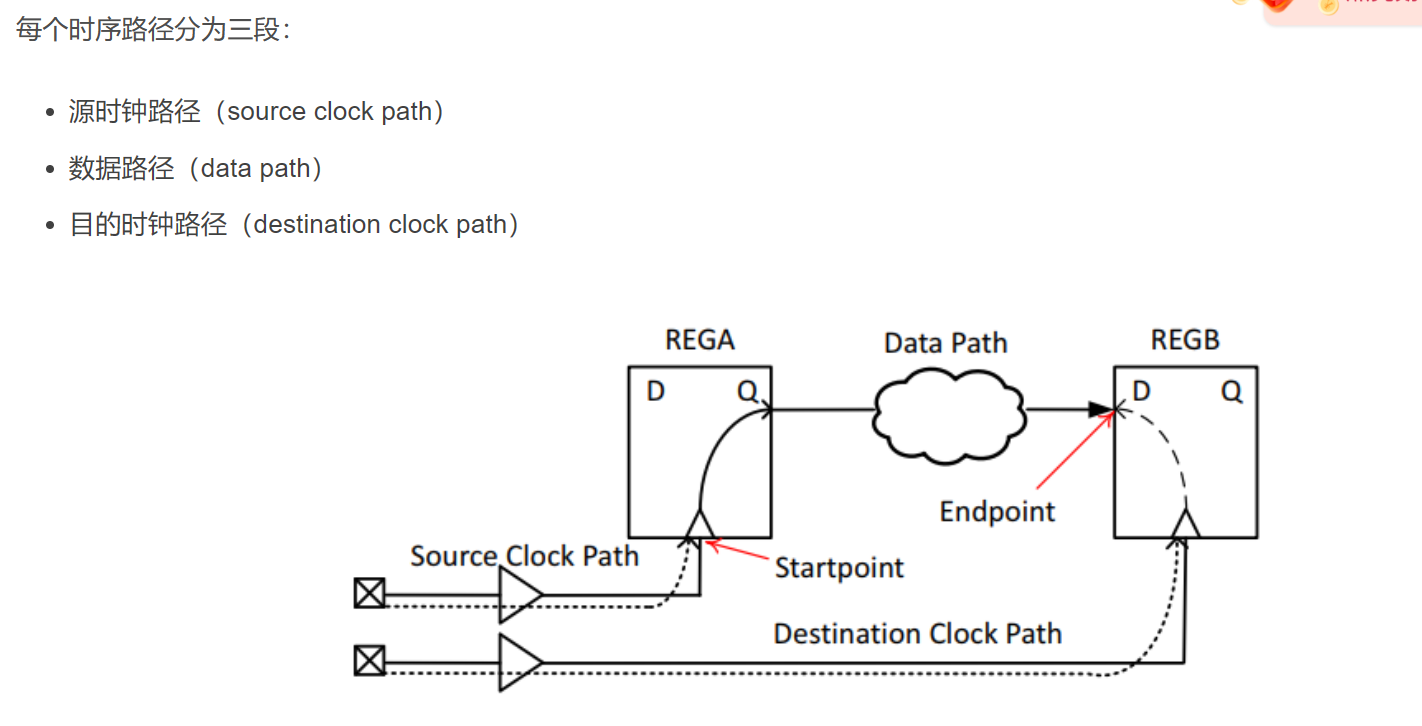


保持余量：





通过看level和fanout，可以看到路径时序违例的原因，level值过大，则表示逻辑层数太多，需要考虑将这条路径对应HDL代码分成几拍完成；如果fanout值过大，则表示该寄存器的扇出过大。理解建立余量的原理后，如果时序过大，那就需要尽可能减少两个除法器之间的组合逻辑级数，减低Tdate.



* 源时钟路径

源时钟路径：源时钟 从【源点（通常是输入端口）】到 【启动时序单元的时钟引脚】的路径。

对于一个从输入端口（port）开始的时序路径，没有源时钟路径。

* 数据路径

数据路径：数据从路径起点（path startpoint）到路径终点（path endpoint）的传递路径。

路径起点：时序单元 的【时钟引脚 或 数据输入端口（port）】。

路径终点：时序单元 的【数据输入 或 输出port】。

* 目的时钟路径

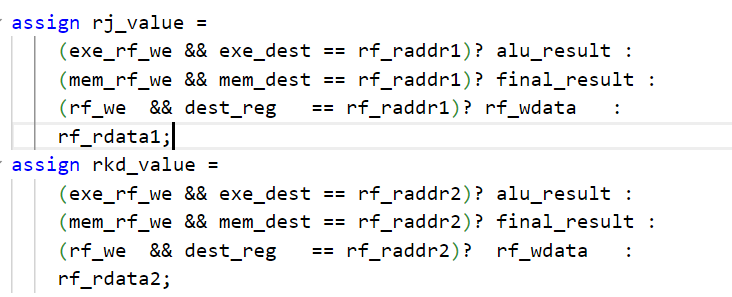
目的时钟路径：目的时钟 从 【源点（通常是输入port）】 到 【捕获时序单元的时钟引脚】的路径。

对于一个在输出port结束的时序路径，没有目的时钟路径。

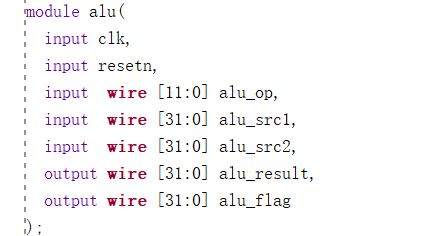
**错误1：**

ID阶段中前递信号的处理部分中，可以看到运用了许多三元运算符，考虑到其时延可能会受到较大影响，考虑到可以改成与或逻辑继续选择，但是这样的操作将会倒置结果与预期完全不符！

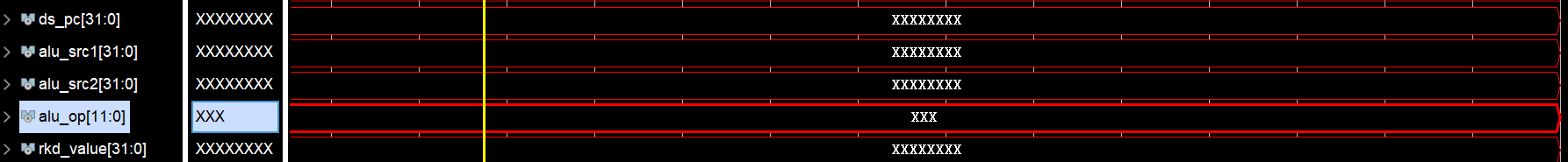
原因是如果改成选择信号，就失去了EXE MEM WB阶段之间的优先级，导致两个阶段同时冲突且需要前递时发生意想不到的错误结果，因此为保证三个阶段之间的优先级，这里的三元运算符并不允许修改。（或者引入其他与或逻辑来代表优先级）

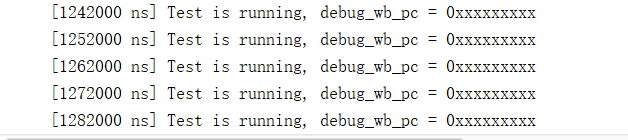


错误2： alu\_op错误位宽



因为需要扩充额外的运算，所以alu\_op位数应该有所增加，但是在进行exp10实验时，遗漏了这点，因此出现在alu中进行结果选择时，选出了含有不定态值的结果，写回值出现不定态。写使能信号未拉高，导致未被采样，但并未报错。





将所有alu\_op定义部分和将其加入传输bus信号位宽增加到需要的位宽即可。我们的设计中乘法器并不例化在ALU中，因此实际上的alu\_op仅有16位。

1、错误3：除法器功能错误

（1）错误现象

1. 除法器算出的结果出现错误，
2. 提前输出结果

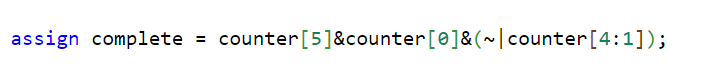
（2）分析定位过程

1. 最后一次循环内counter值为32， [31 - counter] 会导致数组下标越界，因此最后一次循环时将当前余数赋值为recover\_r。
2. 33的二进制码是6’b100001，其最后一位和最高位均为1，最开始时判断的结束信号是31拍，导致提前输出结果

（3）修正效果

将两部分代码改成如下所示即可，即可



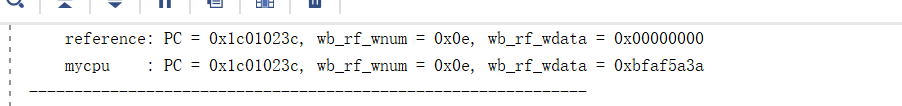


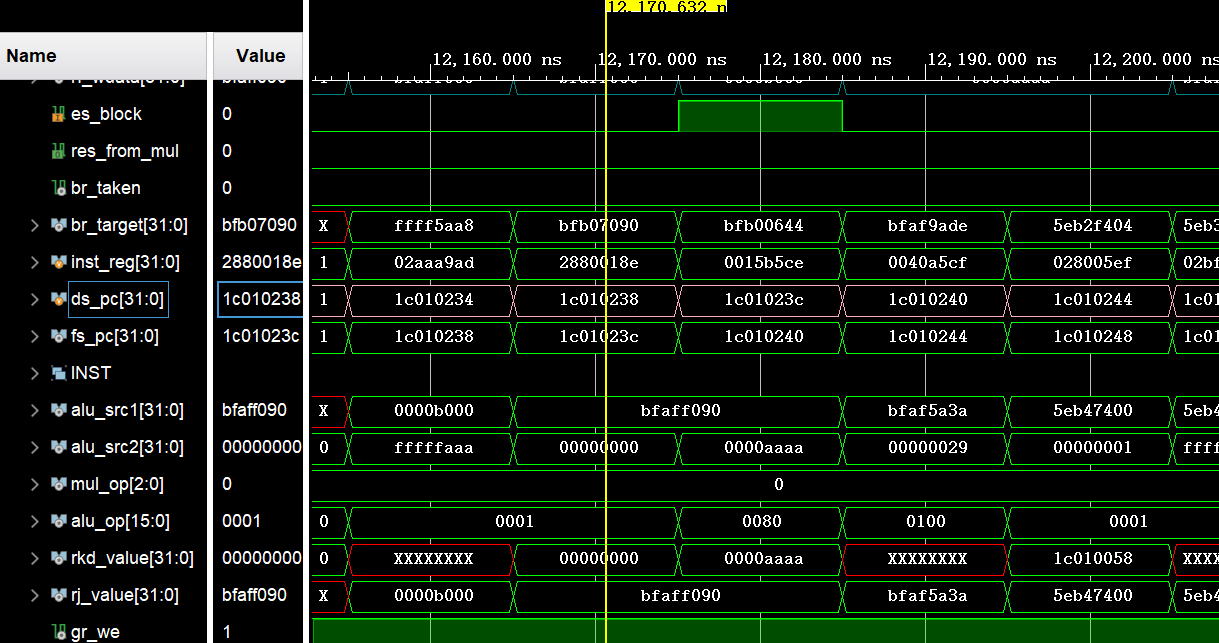
修改后结果正确，仿真验证通过。

1、错误4：前递信号判断遗漏

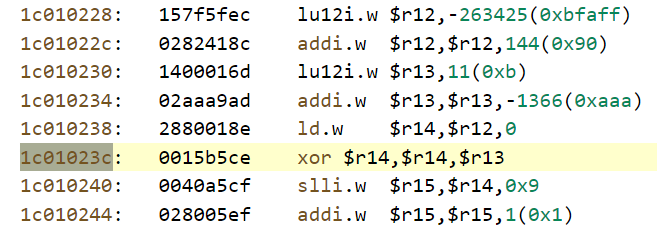
（1）错误现象

因为未考虑乘法器信号可能出现的前递，导致前递回错误的值发生错误



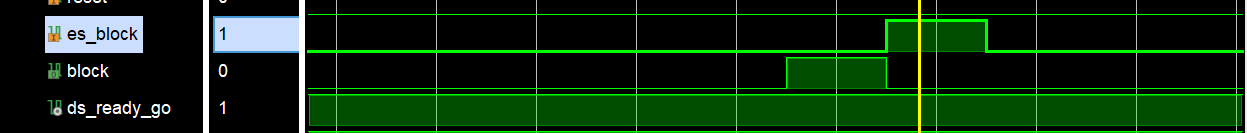


（2）分析定位过程



观察汇编文件，可以看到出现了较多的写后读访问冲突，按照我们的流水线前递设计，将会直接从所需操作数的对应阶段取回，减少流水线引起的阻塞损失。但是在这里，操作数的运算并没有发生错误，流水线也正常前递，可是结果并不对应。一开始想的是是否可能因为存数指令出错，导致ld.w指令取出错误的操作数，因为存数指令并没有写回寄存器的操作，不会进行调试比对。分析后发现，这不可能，因为前面任意一条存数指令都是从寄存器中取数，而寄存器中的值必定是某条指令写入的值，因此如果存数出错一定会在前面某条指令报错。

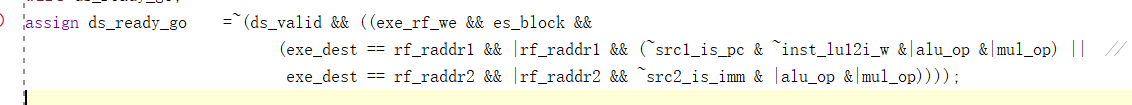
所以调试陷入了瓶颈，考虑到此处的写后读操作较多，因此再次从前递技术上着手，而后发现，ld.w发生了冲突，理应在ID阶段阻塞，但是此时的ds\_ready\_go并未拉低，于是猜测就是因为这个原因导致尽管前递了数据，但是却不是正确的数据。



俩必定不会同时满足，所以这个设计是错误的。

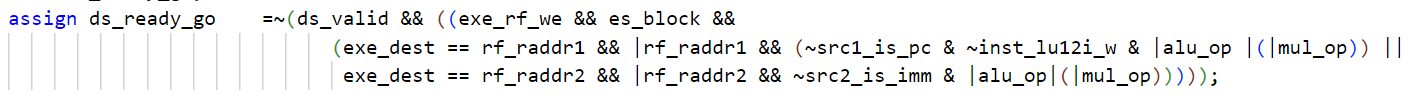
（3）错误原因

在ds\_ready\_go信号的赋值部分，仔细检查，恍然大悟，是因为考虑阻塞条件时对mul\_op信号的引入并不严谨，因为阻塞发生时当前的指令应该是需要操作数并且进行运算的指令，而因为mul\_op和alu\_op互斥，所以它们必定不会同时满足，所以ds\_ready\_go始终为高，出现错误。



（4）修正效果

考虑到mul\_op的引入后，重现修改赋值逻辑即可。

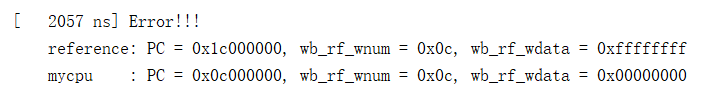


（5）归纳总结（可选）

这个bug是因为前递信号设计遗漏而导致的，所以在控制逻辑部分引入新的信号时理应考虑到兼容和向后延展。

错误5：赋值逻辑错误（实践任务11）

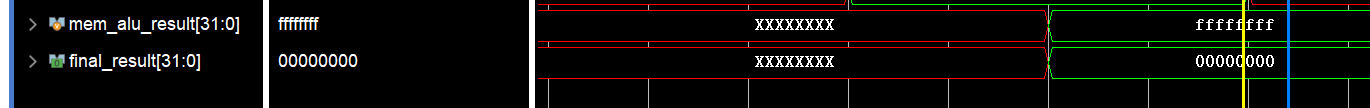
（1）错误现象



2057sPC值和写数据均发生错误

（2）分析定位过程

首先是PC值的问题，发现主要是因为在执行到访存阶段的数据的传递的位宽出现问题，更改后查找写回数据问题。写回数据应该来自于alu\_result，而后赋值给final\_result，可以看到在赋值阶段出现了错误。而后发现mem\_res\_from\_mem拉高了，但此时不应该拉高。查找mem\_res\_from\_mem找到错误。



（3）错误原因

assign mem\_res\_from\_mem = ~(|mem\_load\_op);

右侧式子意思为对首先查找load\_op是否为全0，而后进行非操作。但是当load\_op非全零时应该给mem\_res\_from\_mem赋值为1，因此不应该取反。

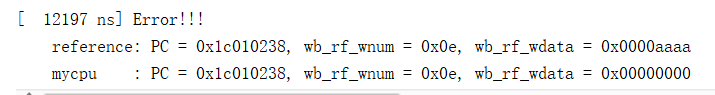
修改为assign mem\_res\_from\_mem = (|mem\_load\_op);

（4）修正效果

修改后该时间点不报错。

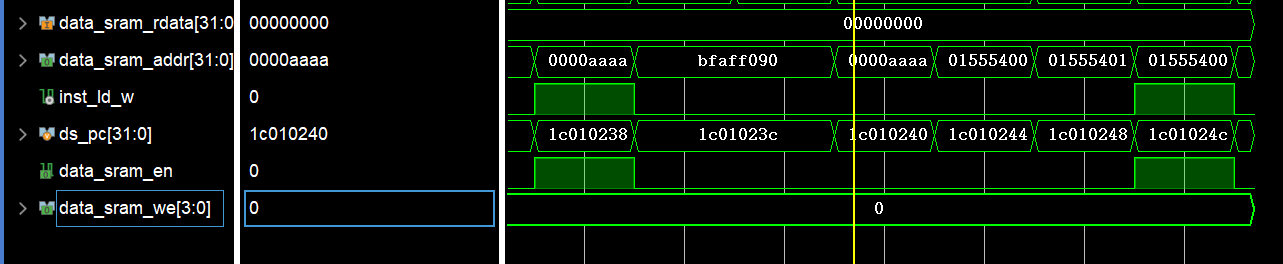
错误6：load\_op错位

（1）错误现象



写入数据错误。

（2）分析定位过程



发现使能信号为0，因此是有问题的。而后查找发现使用的是load\_op，即下一拍的load\_op，因此是错位的，应该使用load\_op\_reg为对应的load信号。

（3）错误原因

assign data\_sram\_en = (mem\_we\_reg || (|load\_op)) & es\_valid;

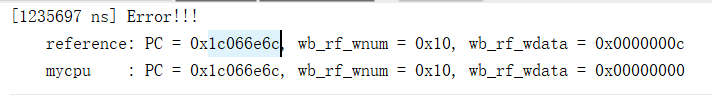
（4）修正效果

assign data\_sram\_en = (mem\_we\_reg || (|load\_op\_reg)) & es\_valid;

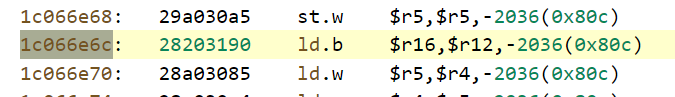
修改后顺利通过测试点

错误7： Ld.w指令出错，立即数信号给错

（1）错误现象

。

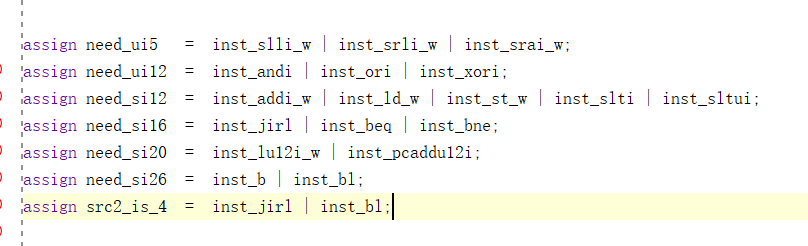
（2）分析定位过程



通过查看汇编文件，可以看到该地址是一条访存指令，取出的数据与预期金标准不符，考虑访存指令的数据通路，可能是掩码操作出现错误或者访存地址出错。

（3）错误原因

先考虑简单的情况，检查立即数选取发现need\_sil2拉高未考虑新加入的访存指令，导致地址出错。



（4）修正效果

加入新的访存指令后，仿真通过。

（5）归纳总结

除此之外，还有诸多因为未考虑引入新指令的控制信号出错，例如rd寄存器选择信号中，需要考虑新引入的跳转指令和访存指令



以及执行阶段和访存阶段，数据RAM的读写使能信号未考虑新加入的访存指令导致结果错误。



四、实验总结（可选）