**32位MIPS流水线处理器加分申请报告**

汤宸 无62 2016011018

李云飞 无62 2016011017

成大立 无62 2016011029

2018年8月1日

**1 32位MIPS流水线处理器概述**

本次流水线MIPS处理器设计中，我们组设计的流水线经过Vivado时序分析，在135MHz的情况下，关键路径仍有0.230ns的建立时间余量。因此，实际时钟频率可以进一步达到139.5MHz。实际使用串口收发助手进行验证时，输出结果正确，且未出现丢包、错包的情况，经老师验收通过。因此，我们组在此申请高频率流水线MIPS处理器加分。

**2 32位MIPS流水线处理器数据通路简介**

我们组采用五级流水结构线结构设计，基本架构、各模块端口如下图所示。下图可能不够清晰，请参见附件中的“流水线框图.png”。

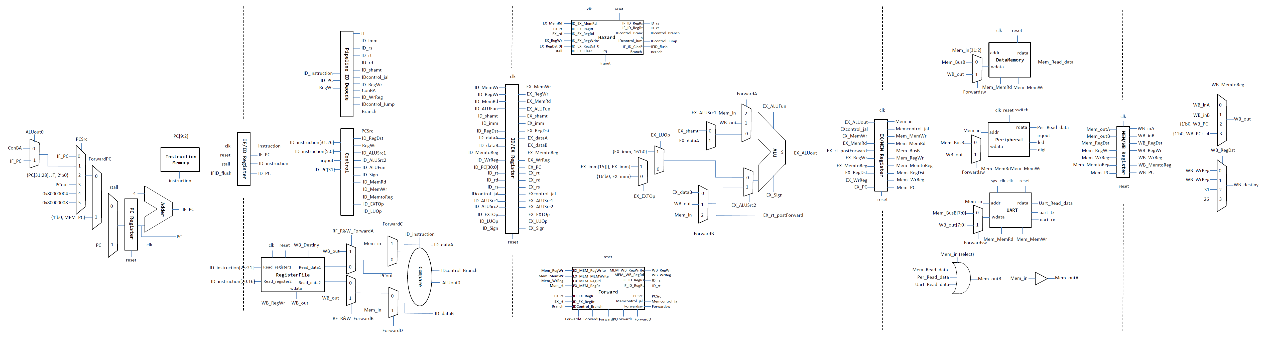


图1流水线CPU整体框图

五级流水线分别为IF、ID、EX、MEM、WB，通过PCSrc及其它特殊控制信号来决定下一个周期的PC值，并读出与PC相对应的instruction信号一同传入ID段。除了对数据通路基本的划分之外，我们设计了冒险和转发单元，转发单元以尽量减少流水线的stall操作为目的，而冒险单元则是对必须的跳转等情况，对送入ID段的数据进行flush或stall操作等。按照要求，我们还设计了外设模块，读写UART、leds、digi等内容的模块。同时，我们设计了定时器中断处理模块，增加了保存断点进入26号寄存器的功能。

首先，我们简单介绍各个模块功能如下。

（1）pipeline.v：顶层模块，定义了各个模块间传递的变量，并把各个模块相连，以实现整体的流水线功能。

（2）pipeline\_IF.v：在这里主要完成PC变化的操作，由PCSrc控制多路选择，正常情况下，PC进行加四操作；stall信号起作用时，PC不变；中断处理入口、异常处理入口、分支目标地址、跳转目标地址等不再赘述。值得一提的是，我们发现当jal指令后紧跟着jr $ra指令时，会出现一次数据冒险，即从31号寄存器里读出来的值，是正确数据存进去之前的，因此我们添加了一条转发操作，进行PC的地址转发。

（3）InstructionMemory.v：通过PC读出对应的指令，准备送入ID段。

（4）IFID\_reg.v：IF与ID的段间寄存器，从IF段及InstructionMemory取出ID段需要用的PC值及指令，在正常情况下送入ID段。当stall控制信号处于高电平时，送入ID段的信号与上一周期保持一致，不进行变化，当flush信号处于高电平时，送出去的PC值与上一周期保持一致，而instruction置零。

（5）pipeline\_ID.v：在这一段，我们对instruction进行了解析，生成了主要的控制信号、需要使用的寄存器编号、立即数、写入寄存器编号（如果需要当前指令需要写寄存器的话）；我们通过RegisterFile.v读出需要使用的寄存器内的值；我们对分支指令进行提前判断，并将相应生成的控制信号送入冒险单元及IF段，当发生reg-beq数据冒险时，通过转发来控制送入的数据；

（6）IDEX\_reg.v：ID与EX的段间寄存器，传递后续段需要用到的信号。

（7）pipeline\_EX.v：这里主要进行ALU操作，ALU部分已在实验报告中给出，这里不再赘述。由ALUSrc及Forward信号控制送入ALU的两条数据通路上的值。输出ALU的结果。值得一提的是，对于sw操作，虽然由ALUSrc2控制的多路选择选择了传递立即数，但是rt寄存器通路上的值在MEM阶段也需要用到，因此我们在这里额外添加了一个输出，以解决这个问题。

（8）EXMEM\_reg.v：EX与MEM间的段间寄存器，传递后续段需要用到的信号。

（9）pipeline\_MEM.v：接入DataMemory，Peripheral及UART三个底层模块，进行数据存储器、外设数据的读写以及中断信号的生成等操作。通过多路选择器控制输出。

外设模块完成中断信号的生成，送入ID段及冒险单元来影响流水线。同时完成计时器功能，外设存储器写功能等。UART模块负责UART的读写功能。

（10）MEMWB\_reg.v：MEM与WB的段间寄存器，传递WB段需要用的信号。

（11）pipeline\_WB.v：进行写入数据、写入寄存器的选择。

（12）Forward\_Unit.v：转发单元，涉及到四类转发。EX阶段需要使用MEM阶段或者WB阶段正在传递的，还未写入寄存器堆的数据，先判断是否要从MEM阶段转发，再判断是否从WB阶段转发，以实现从最近的一条指令进行转发；提前判断分支的reg-beq转发；IF段需要使用还未写入31号寄存器堆的PC值，这种情况带来的转发；load-use型转发。

（13）Hazard\_Unit.v：冒险单元，包括中断指令带来的冒险共有五类。跳转指令需要的对一行的flush操作；reg-beq数据冒险，需要一次stall；load-use数据冒险，需要一次stall；分支判断，需要进行一次flush置零指令；中断的到来，致使需要跳转到中断处理程序，这里需要一次flush操作。

如果不考虑stall，以及在合理进行段间划分的情况下，五级MIPS流水线处理器的稳定速度是单周期处理器的五倍，一个处理器最多可以同时处理五条指令。而为了保证流水线的顺畅执行，转发模块、冒险检测模块以及PC的跳转都十分重要。下面我们就这些模块作出进一步分析。

（1）Forward模块

如图所示，共有四种类型的Forward模块。ForwardA(B)，ForwardC(D)，ForwardPC及Forwardsw。

·在IF段需要使用的有一种，通过ForwardPC来进行控制。当运行jr $ra指令时，若31号寄存器的值还没有更新成正确值，那么就需要将PC提前转发给IF段。当ID段解析出这是一个jr指令时，ID段的周期内置Memcontrol\_jal为1并一路传到MEM段，ForwardPC在PCSrc等于3（对应从31号寄存器读PC），且MEM段的Memcontrol\_jal为高电平时，开始起作用，从MEM段取出PC传入IF段。

·ID段需要使用的有一种。在本段，由于需要提前判断分支，在发生reg-beq数据冒险时，要用到的数据值还未存入寄存器堆，这就需要从MEM阶段向ID阶段进行转发。当接到信号IDControl\_Branch提示此时为分支指令，且正在传递的写入寄存器编号等于此时判断分支需要用的读出寄存器编号时，起作用并控制对数据进行转发。

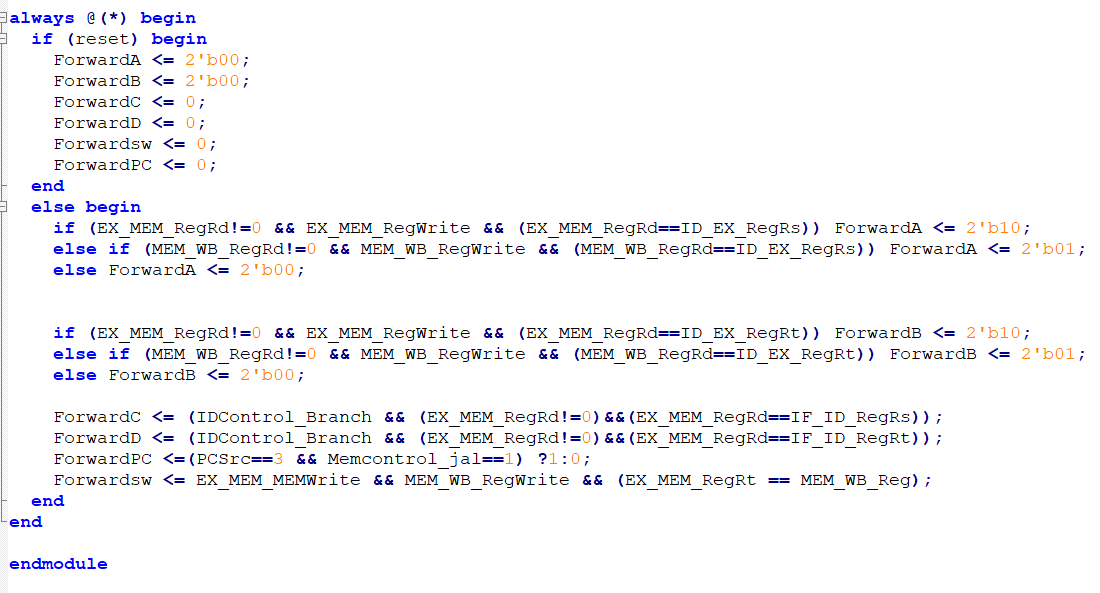


图2 转发模块的代码实现

·EX段需要使用的有一种。在本段，输入到ALU的两条数据通路上的值，都存在数据冒险，需要进行判断。我们采取先判断就近的一条指令，再判断上上条指令的方法，当写入寄存器不为零，且与本EX段两条输入数据通路的寄存器编号相同时，forward起作用，对参与EX段数据通路的多路选择，如下图所示。

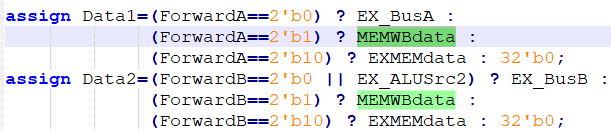


图3 EX阶段使用的转发信号

值得一提的是，当指令为sw时，即会有rt寄存器的编号，立即数也会传来一个值并送入ALU，此时对立即数不能转发，因此上图的Data2比之Data1略有不同。

·MEM阶段有一种转发情况。当MEM阶段需要写寄存器，且写入数据的寄存器编号与上一条要写入寄存器堆的寄存器编号相同时，起作用。

这四条是最初的设计，但当我们的寄存器堆部分出问题后（问题已在实验报告中描述），就多了一条从WB阶段向ID段的转发，即当寄存器堆不能先写后读时，相隔两条指令也有可能存在数据冒险。这条转发只要稍作修改即可，并没有在Forward模块进行改动。

（2）Hazard模块

在冒险检测单元中，主要存在两种冲突处理形式，一种是flush，使ID段PC保持不变，且清零instruction（instruction置零时，是对零号寄存器的值左移零格，相当于清空指令）；另一种是stall，使ID段的PC和instruction保持不变，重复执行上一条指令的内容，并将送入EX段后面的写入使能归零，让上一行的执行无效，这样就相当于阻塞了一个周期。上图的ID\_EX\_Clear代表stall信号，IF\_ID\_clear代表flush信号。

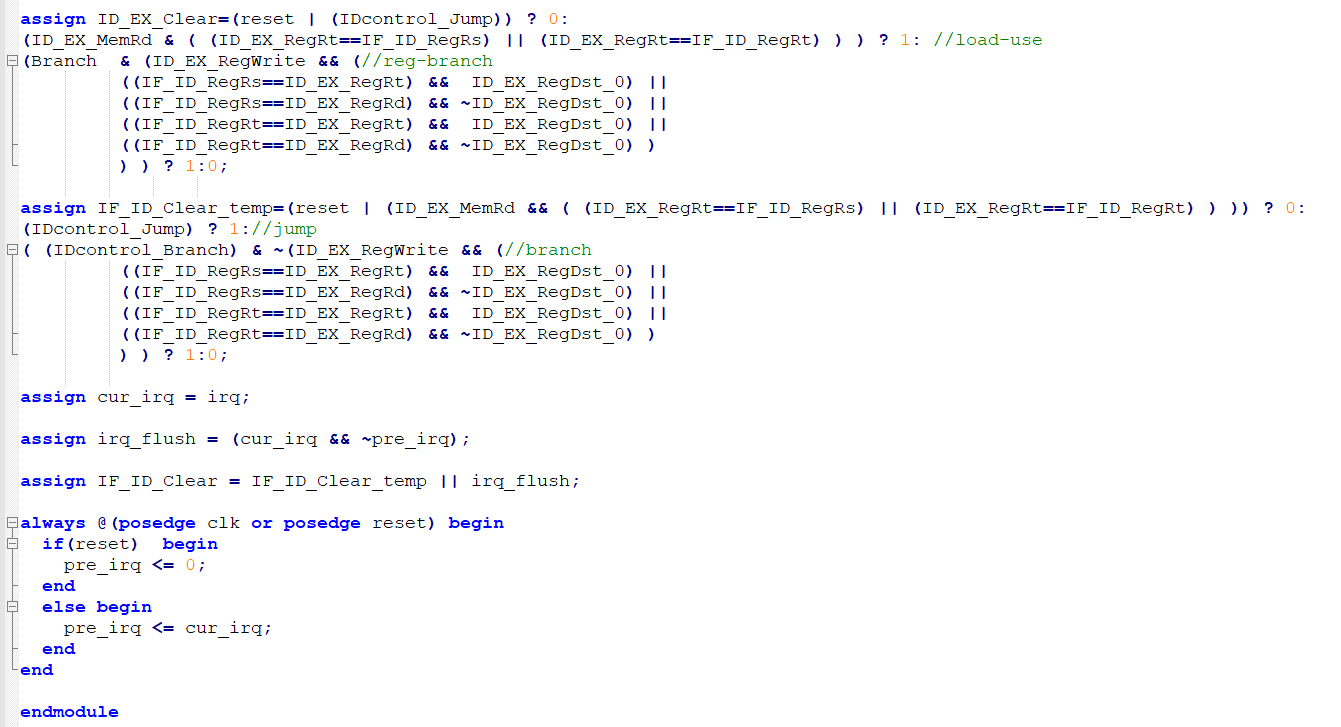


图4 冒险检测模块的代码实现

·需要stall的情况有两种，主要是数据冒险。一是load-use数据冒险，另一个是reg-beq数据冒险。都需要stall一个周期来等待转发。

·需要flush的情况有三种，首先是跳转指令，需要对下一条指令进行清空；其次是分支指令，首先判断这不是一个reg-beq数据冒险，如果是的话需要先stall一个周期等待转发重新判断分支，然后得到分支判断成功的控制指令时，清空下一条指令；最后是中断操作，虽然中断进行的操作本质也是一个jal指令，但中断信号和表明这条是跳转指令的控制信号不一样，在我们的设计中，中断信号会持续高电平一段时间，那么为了让它只flush一次，我们使用了irq的现态和次态进行预判，只有当前一个周期中断信号为0，本周期为1时，才需要进行一次flush。

（3）PC多路选择

PC的正确选择对流水线处理器的正常运转至关重要，这里需要强调两点，一是PC高位置零，这在实验报告中详细阐述，这里就不过多赘述；第二点是ForwardPC，由于ForwardPC的生成过程中就要求了PCSrc为3，所以在这里的判断就没有加上PCSrc。

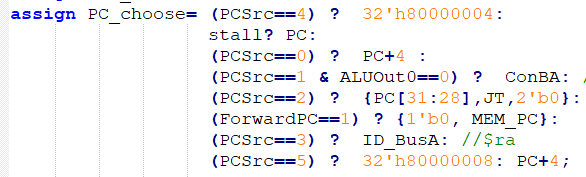


图5 PC跳转地址的代码实现

**3 时序分析与关键路径分析**

在综合过程中，我们发现，受到Vivado算法的影响，在不同的时序约束下，甚至在不同的计算机上，关键路径都有所不同。在此，我们首先选取了在优化过程中出现最频繁的一条关进路径进行分析。注意为了方便起见，这里的时钟频率仅设置为50MHz。

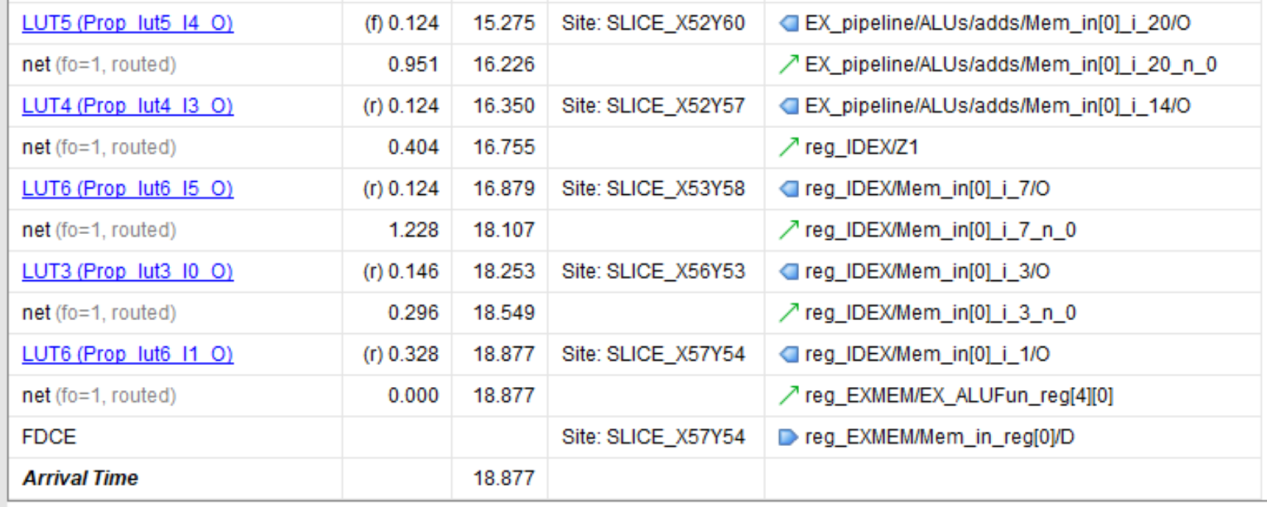
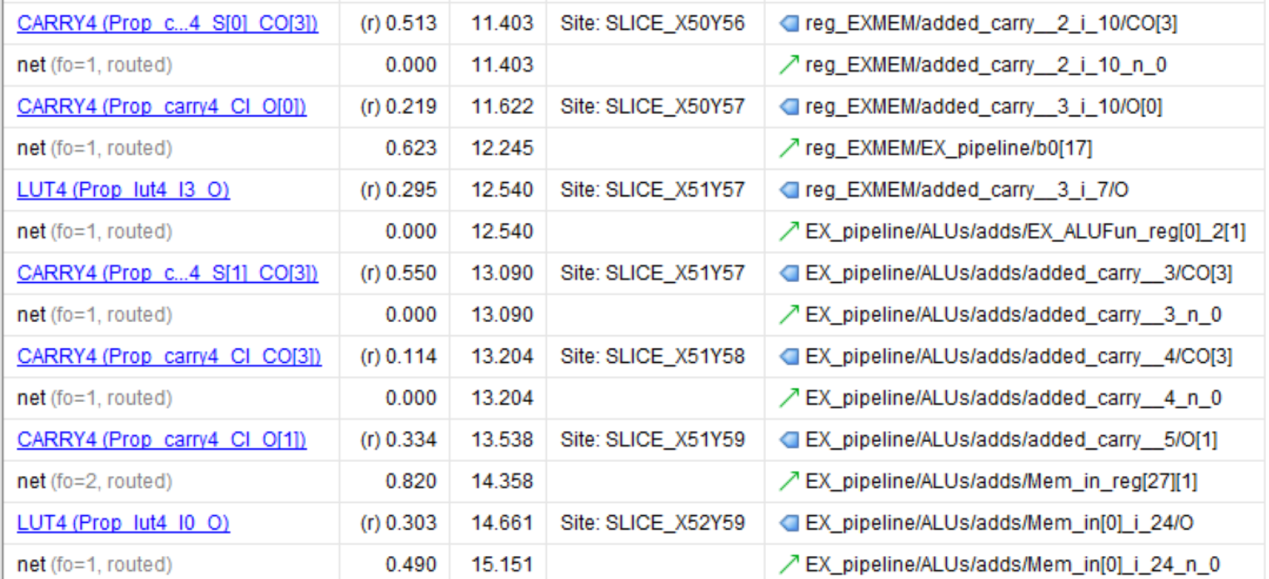
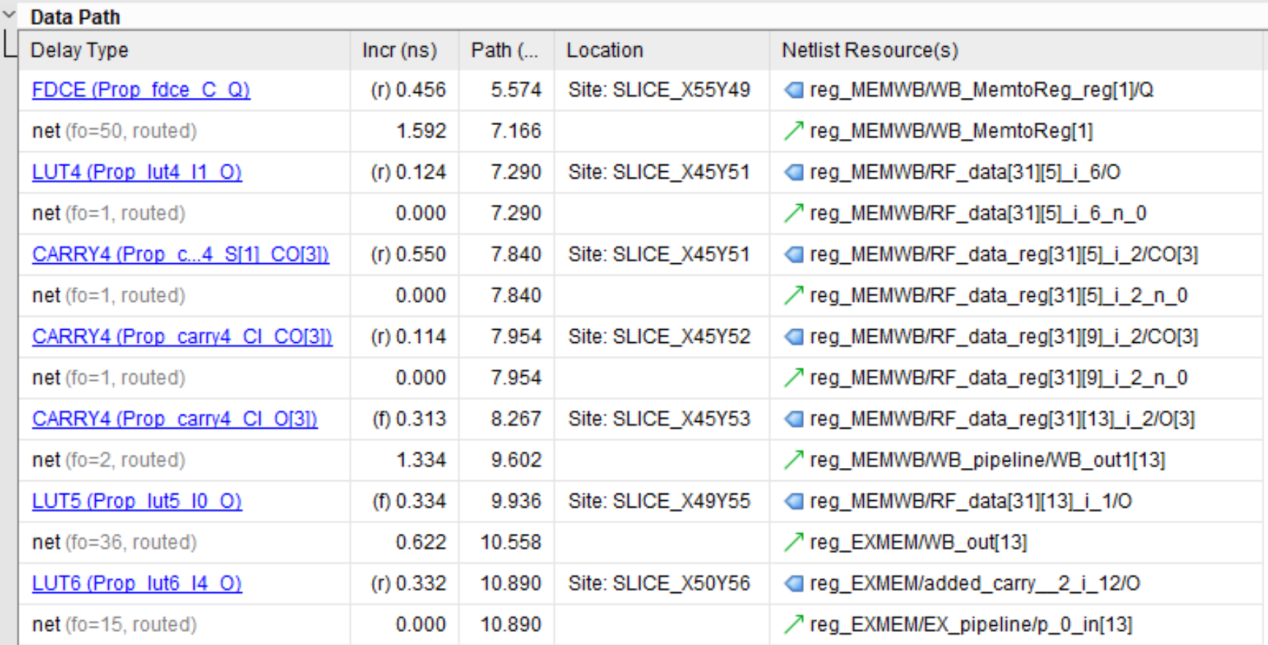
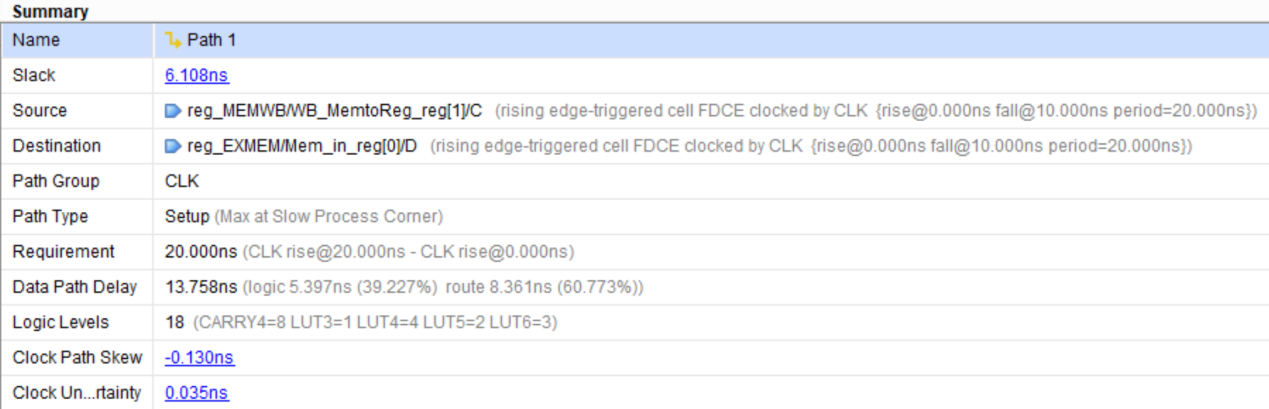
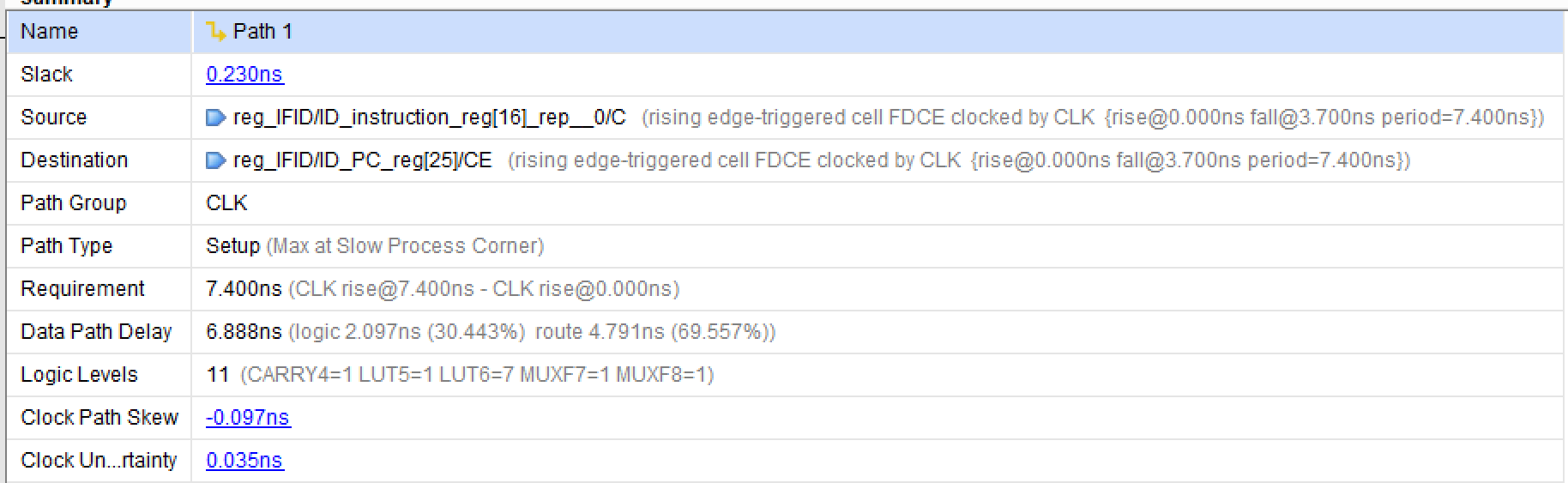


图6 一条在综合过程中曾多次出现的关键路径

简单观察上述关键路径，可以看到上述路径的起始点位于MEM/WB寄存器内部；这条路径经过了由MemtoReg决定WB\_out的操作之后，来到了EX阶段，并且进入了ALU内部；最后抵达了EX/MEM寄存器当中。虽然表面上看起来这条路径曾经经历过ID/EX寄存器，但我们发现，它经历的只是ID/EX\_reg中的LUT，而没有经过FDCE；使用Schematic->Netlist->Go to Source操作后，发现ID/EX仍然对应ALU内部的源代码。（Vivado似乎会把其他模块的功能放置在一些段间寄存器当中。）很明显，这是一条转发路径，是ForwardA和ForwardB对应的转发。在这条路径上，对延时贡献较大的包括导线延时（高扇出、路程长），以及CARRY4的串行延时。

在我们最终提交的代码版本中，关键路径如下图所示。这条路径是从IF/ID段间寄存器的instruction到PC的通路，中间经过了RegisterFile，MEM/WB段间寄存器，IF/ID段间寄存器等通路。根据前述经验，虽然显示了MEM/WB和IF/ID寄存器，这条路径应该没有进入寄存器内部。（关键路径的首尾一定是寄存器，其中间不可能有别的寄存器！）既然这条路径从取指令开始，中间访问了寄存器堆，最后回到了PC，这有可能是一条类似于jr指令的数据通路。既然是正常指令的数据通路，我们认为，对这条路径进行进一步优化的余地可能不是很大。

另外值得注意的是，这条路径中的逻辑延时只占30%，导线延时占到了70%，因此在我们的处理器设计中，影响时序的更关键的因素或许是导线延时。



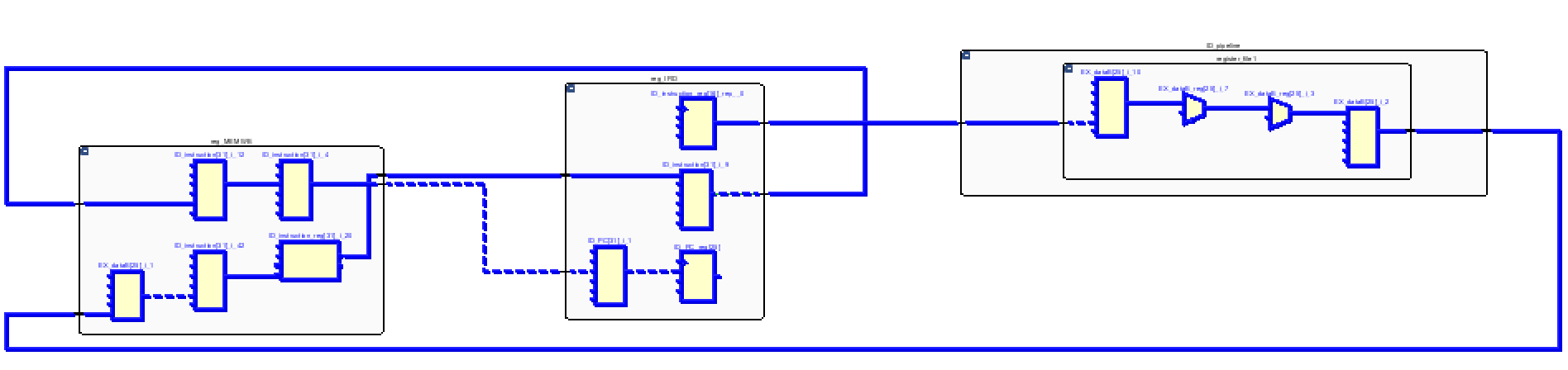


图7 135MHz流水线处理器的关键路径示意图

**4 时序性能优化策略**

为了尽可能提高流水线CPU的时钟主频，我们采取了一系列的优化措施，在此逐一介绍。

**4.1 以空间换时间**

在实验报告中我们已经看到，CPU占用的硬件资源仅占到了FPGA总资源的10%上下，因此硬件资源是非常充分的。如果我们对时序性能有更高的要求，可以考虑牺牲空间资源。这种策略有以下两个体现。

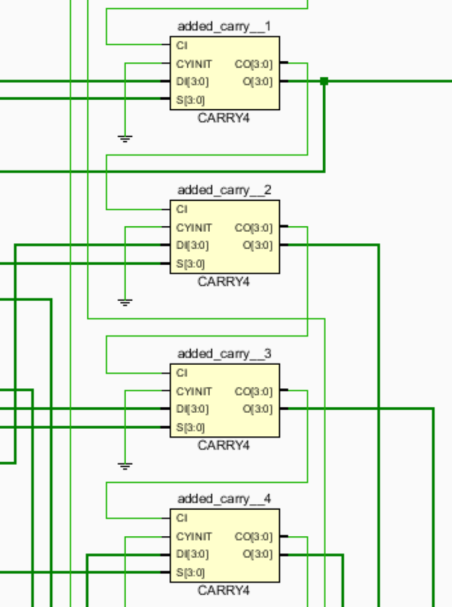
ALU中移位的串行与并行的处理。按照实验指导书的要求，ALU中的移位操作由移位1，2，4，8，16等几种操作拼合而成，因此理论上，完成一次移位至多需要5次串行操作（例如左移31位）。我们在分析关键路径的过程中，曾经发现关键路径中含有连续的5个CARRY4，每个CARRY4的逻辑延时都相对比较长，它们串联起来造成了很大的延时。这5个CARRY4都在EX\_pipeline阶段，并且使用Go to Source命令后，发现这些CARRY4都指向Shift模块，由此可以确定，ALU当中移位的串行可能带来较长的延时。我们使用查找表的方式，将每一种移位的结果都手动列写出来，希望可以解决这一问题；但在后续调试过程中发现，关键路径不再经过这几个CARRY4了，因此也就没有再做修改。但令人费解的是，CARRY4本质上应该是4位并行的超前进位加法器，为什么移位操作（本质上就是导线的合并）需要用到CARRY4呢？或许也有可能是为了将shamt操作数分解为1，2，4，8，16的组合；还有一种可能，即移位操作和普通的加法操作复用了同一批CARRY4，因此我们看到的关键路径，可能不仅仅是Shift模块导致的，也可能是普通的32位加法引起的延时。

图8 ALU内部的串行的CARRY4加法器

对于$zero零号寄存器的处理。在助教提供的RegisterFile版本中，我们看到，寄存器堆只有31个32位寄存器，0号寄存器是不存在的。RF的输出端有一个MUX，如果源寄存器地址为0，那么RF的输出由MUX选通为32’b0。同样是在关键路径分析过程中，我们发现了这个MUX的存在。这个MUX的摆放位置距离其他硬件很远，带来了较大的导线延时。我们于是删掉了这个MUX，在RF中实实在在地添加了一个0号寄存器，保证它的取值始终为0。经过这次操作，建立时间余量增加了大约0.1ns。

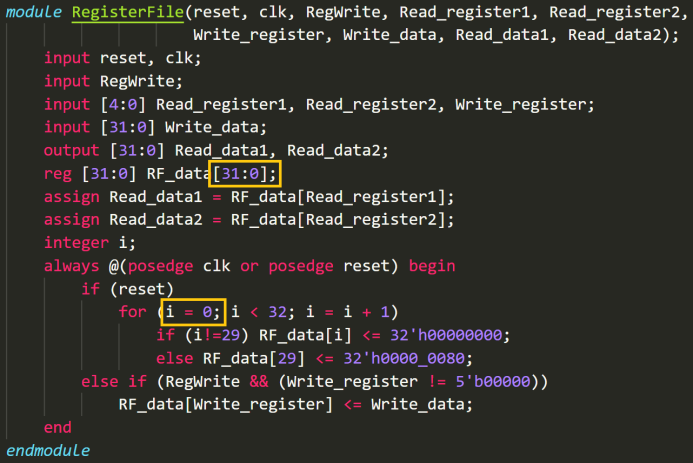


图9 加上零号寄存器后的寄存器堆（如图中橙色框所示）

**4.2 想方设法减小导线延时**

在实验报告中我们看到，关键路径的总延时中，只有大约30%来自逻辑延时，剩下70%都是导线延时（net delay）。因此，花大力气优化逻辑延时，不如想办法降低导线延时。我们认为，导线延时太大，主要是两种原因导致的：扇出太大或导线太长。

扇出系数（fanout）表征了某一元件驱动负载的多少。如果并联负载太多，下一级的输入电容就会增加，导致延时增长。整个设计中最大的扇出是全局时钟，总计2000余个触发器都需要时钟来驱动。但一方面，全局时钟使用了名叫BUFG的模块，这种缓冲器能有效提高扇出能力和时钟的稳定性；另一方面，源地点的时钟和目的地的时钟都要受到大扇出的干扰。因此，全局时钟的处理不在我们的优化范围内。我们仅尝试对某些扇出较大（大约50以上）的寄存器做了(\*MAX\_FANOUT\*)规定，但从结果看来，没有明显优化效果，因此我们放弃了这种操作。



图10 一个典型的高扇出的例子，导线延时长达1.15ns，或许是由于扇出系数高达263

导线太长也会引起导线延时过大。在Device图中，经常能看到关键路径跨越了很多个子区域。但这是Vivado的自动布线和绑定的结果，很难从源代码层级上修改。好在Vivado比较智能，能够根据时序要求调整布线策略。当我们缩短约束文件中的时钟周期时，可以明显看到，Vivado把更多的使用到的硬件资源安排到了更近的位置，使得导线延时大大缩短，使得建立时间余量往往比我们预期的更大。至于在时序要求不严格的情况下，Vivado会将硬件资源分散在FPGA芯片各处，我们认为，这样做可能更有利于芯片散热，进而提高硬件电路工作的稳定性。

**4.3 优化代码风格**

我们在后续调试过程中，发现了一些冗余代码和冗余变量。这些冗余主要包括两部分：一是有用但无连接的变量，二是无用但有连接的变量。

有用但无连接的变量主要是为了程序的完备性。例如，PC在InstructionMemory中寻址时，低两位总是00，并不需要作为索引。如果强行把PC的32位全部都接入InstructionMemory，Vivado会产生warning，提示我们PC[0]和PC[1]是unconnected port。我们认为这些有用但未连接的管脚，或多或少总会对时序产生影响，因此将取消了所有不必要的变量的连接。

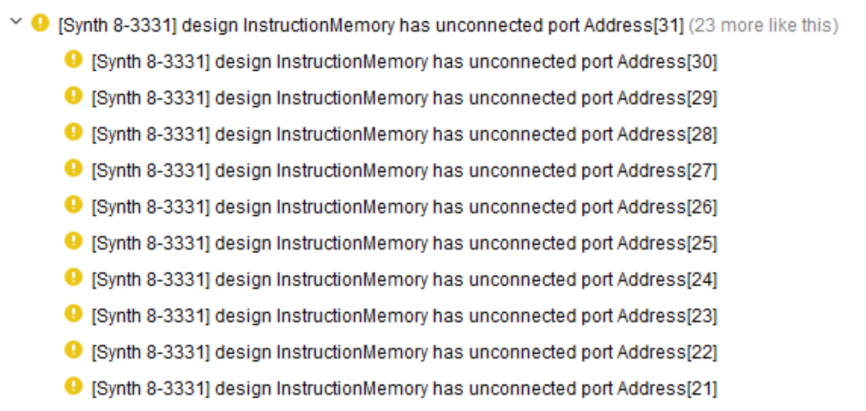


图11 未连接的端口会引发Vivado的警告

无用但有连接的变量主要是代码风格的问题，例如input A; output B; wire C; assign C = A; assign B = C;一类的代码，或者assign notA = A ? 0 : 1;一类的不够简洁的代码。在第一个例子中，中间变量C必然引入一个缓冲器；在第二个例子中，一个非门可以完成的任务（而且往往D触发器就有反相输出端）由一个MUX（甚至是LUT2）来完成。这些冗余代码不仅造成了硬件资源的浪费，还造成了延时的增加。因此这些无用的变量也是要删除的。

C:\Users\cheng\AppData\Local\Temp\WeChat Files\7271c6bd2d47382da9b694a41213776.png

图12 一行可以进一步优化风格的代码

事实上，完成CPU的编写之后，通过和其他小组的交流，我们发现，初始编写时的设计，很可能比后期优化更为重要。我们的流水线CPU没有经过优化时就可以在100MHz时钟上运行了，但其他一些小组的流水线CPU即使经过了优化，也达不到100MHz；还有一些小组，没有经过优化都可以达到170MHz或180MHz。虽然不是很理解为什么差异如此巨大，但我们认为，一般而言，后续优化只能提升30MHz左右，不能指望通过修修补补实现时钟频率的巨大飞跃。因此初期的编写设计可能更为本质。流水线的各个阶段之间应该尽量平均，避免工作量分配不均匀；开发初期应该对整体需求有明确的认识、对整体架构有详细的规划，避免后期调试时给一个漏洞百出的数据通路打补丁；可能这些前期编写中的注意事项，反倒决定了最终CPU主频的基调。