简化的 MIPS32 处理器设计报告

成立

1110379003

第一章 绪论

1.1 背景

MIPS 的全称是 Microprocessor without Interlocked Piped Stages，即“无内部互锁流水级微处理器”，是一款非常流行的 RISC 处理器。其机制是利用各种软件方法来避免流水线中的数据相关（Data Hazard）问题。它最早是在上世纪 80 年代初期由斯坦福大学 John Hennessy 与 David Patterson 教授领导设计研发出来的，尽管如此，它仍然活跃在当今世界的各种电子设备上，如手机、路由器、超级计算机等。

说到 MIPS 处理器，不得不先提到 RISC 的概念。IBM 研究中心的 John Cocke 证明，计算机中约 20%的指令承担了 80%的工作，他于 1974 年提出了 RISC 的概念。RISC 的精髓在于精简，它指令种类少（但是用的很频繁），指令格式规整，寻址方式少。这大大简化了 CPU 控制器的设计，减少了晶体管和电路元件的数量，从而大大提高了运算速度。

正是在“精简”这样一个前提下，才诞生了 MIPS 这样巧妙的设计。它天生就透着易于流水，易于扩展的巨大优势，这使得 MIPS 处理器在短短几十年间，得到了高速发展。

MIPS 目前已发展到 64 位 MIPS 体系结构，中国龙芯 2 即采用该架构，并且已与 MIPS 公司达成了合作，相信 MIPS 的前景会更加广阔。

本文实现的是一个简化的 32 位 MIPS 处理器。

1.2 目标

实现一个五级流水线 MIPS32 处理器，它能够：

a) 支持标准 MIPS 指令集的一个子集，涵盖数据搬移指令（Load 与 Store）、运算指令（Add 等）、分支指令（Beq 等）。
b) 能够检测并解决三种相关。
c) 支持 Cache。

1.3 工具

本处理器采用 C#语言编写，开发环境为 Windows 7 与 Visual Studio 2008。选择用 C#而不是*HDL 语言来设计 CPU 的初衷，是整个“CPU”完全由自编代码从零开始堆积，自由度较大，并且易于调试与仿真。

但到最后，事实证明：普通的程序设计语言并不适合硬件开发。因为它们不能很方便地处理时序问题，比如“在下跳沿写入寄存器”是无法实现的，只能利用别的方法来代替，其实反而将问题复杂化了。

另外，汇编器采用 C 语言编写，由 GCC 编译。

1.4 指令集

本处理器支持标准 MIPS 指令集的一个子集，共 17 条指令，如下表所示：

```
op     rs    rt    rd    sa    func    ; R format instructions
000000 rs    rt    rd    00000 100000  ; add  rd, rs, rt
000000 rs    rt    rd    00000 100010  ; sub  rd, rs, rt
000000 rs    rt    rd    00000 100100  ; and  rd, rs, rt
000000 rs    rt    rd    00000 100101  ; or   rd, rs, rt
000000 rs    rt    rd    00000 100110  ; xor  rd, rs, rt
000000 00000 rt    rd    sa    000000  ; sll  rd, rt, sa
000000 00000 rt    rd    sa    000010  ; srl  rd, rt, sa
000000 00000 rt    rd    sa    000011  ; sra  rd, st, sa
op     rs    rt    imm                 ; I format instructions
001000 rs    rt    imm                 ; addi rt, rs, imm
001100 rs    rt    imm                 ; andi rt, rs, imm
001101 rs    rt    imm                 ; ori  rt, rs, imm
001110 rs    rt    imm                 ; xori rt, rs, imm
100011 rs    rt    imm                 ; lw   rt, imm(rs)
101011 rs    rt    imm                 ; sw   rt, imm(rs)
000100 rs    rt    imm                 ; beq  rs, rt, imm
000101 rs    rt    imm                 ; bne  rs, rt, imm
op     addr                           ; J format instructions
000010 addr                           ; j    addr
```

第二章 MIPS32 汇编器

2.1 概述

为了调试 CPU，通常需要用大量不同的汇编代码去测试。在这个过程中，首先要写出文本形式的汇编代码，其次根据指令类型的不同，逐个翻译成机器指令，输入指令内存去调试。这个工作如果由人工来完成，是十分繁琐的，何况若涉及到分支跳转指令，地址的计算不仅麻烦，还容易出错。

汇编器就是为这种情况而诞生的。针对本处理器支持的指令集，我用 C 语言编写了一个简单的 MIPS32 汇编器，能够将输入的汇编语言文件转换成 C#数组风格的机器码，极大加快了调试速度。

2.2 词法分析

作为汇编器的第一步，就是要从源文件中分割出一个个记号（Token），形成记号流（Token Stream）。

在本汇编器中，为了简化设计，将 17 条指令的名称均视为关键字（Keyword）。除此之外，还有七种记号：标签（Label）、冒号、逗号、数字常量、美元符号$、左括号、右括号。

词法分析过程采用手写有限自动机模型，共有如下 5 种状态：

a) START，起始状态
b) IN_STRING，正在读取关键字或标签的字符串
c) IN_NUMBER，正在读取数字常量
d) IN_COMMENT，正在读取注释
e) END，终止状态

每个记号的识别都从 START 出发，至 END 结束。若期间遇到错误，则直接输出错误信息后退出程序。

公开给语法分析器的 API 主要包括两个函数：

a) next_token()，获得下一个记号
b) eat(token)，匹配当前记号并“吃掉它”，同时获得下一个记号

2.3 语法分析与代码生成

因为 MIPS32 汇编语言的语法非常简单，所以语法分析器就直接采用“递归下降法”手工编写了。在分析过程中，无需建立语法树，而是直接边分析边生成代码。主要实现思路是：根据 MIPS 汇编语言 3 类指令的不同语法形式分别处理，生成一个 instruction_t 结构的变量，该结构体内包含了该条指令完整的代码生成信息。故而在分析完每条指令的语法后，是直接按该结构体的内容生成对应的机器码并填充到指令缓冲区中。

2.4 链式回填

处理标签时，要特别注意：有时，为分支或跳转指令生成代码时，遇到的标签还未定义，

此时是不能生成正确的机器码的，必须等到该标签的地址确定下来以后才可以。

因此，通常代码生成遍历一遍代码是不够的，需要再遍历一次，按照已知的标签地址填充用到该标签的指令地址域，这种技术被称为回填（Backpatch）。

回填的实现方式多种多样，效率与应用范围都不尽相同。在本汇编器中，我采用的是“链式回填”方法。即为每个标签维护一张链表，将需要用到该标签的每条指令都逐一添加至该链表中，最后遍历所有标签及其对应的链表，一次性填充完毕。

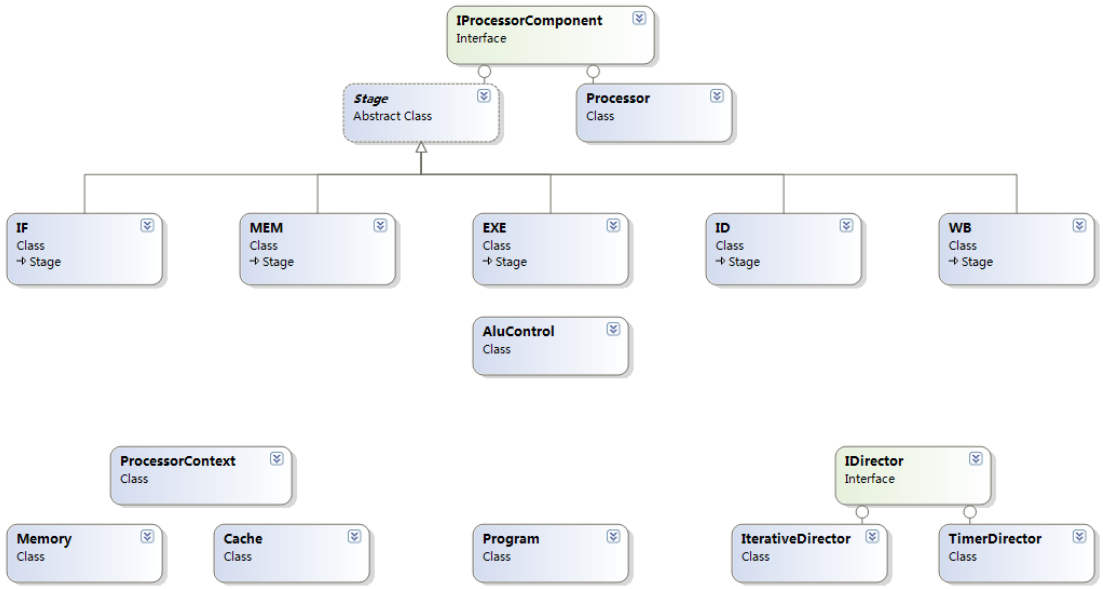
不过在本例中，链表的实现方式不太一样，为了节省空间，我将跳转或分支指令的地址域自身当作 next 指针，让它指向下一个要回填的指令，这样省去了另行编写链表的麻烦。

第三章 五级流水线处理器

3.1 设计描述

3.1.1 程序框架

下图，是我利用 Visual Studio 2008 的“生成类关系图”功能生成的整个程序的框架图：



五级流水线，意味着指令的执行分为五个阶段，在MIPS的设计中，它们是指令提取（IF）、指令译码（ID）、指令执行（EXE）、存储器访问（MEM）与寄存器写回（WB）。在 MIPS 处理器中，这五个阶段的功能通常是在处理器的五个空间区域中被分开实现的，在这里我们不妨将每个阶段对应的区域称为一个处理器部件（Processor Component），即一个 MIPS 处理器是由五个部件构成的，同时该处理器自身也认为是一个部件。

基于这样的组合关系，我们得到了上图中上半部分的树型结构图，描述如下：

a) IProcessorComponent 接口代表一个最抽象的处理器部件。
b) Processor 实现了 IProcessorComponent 接口，代表处理器。
c) Stage 是一个实现了 IProcessorComponent 接口的抽象类，代表抽象的“阶段”。
d) IF、MEM、EXE、ID 与 WB 分别是继承自 Stage 的类，即为处理器的五大主要组成部件。

现在有了处理器的五大部件，但是它们如何关联呢？即真实处理器电路中的各种连线、节点、寄存器在 C#中又该用何种抽象来表示？我的做法是，将它们放在一个被称为处理器上下文（Processor Context）的类中。该类除了包含这些信息的“抽象”外，还持有对一个 Cache（高速缓存）和一个 Memory（内存）的引用，因为我认为，至少在这个程序中，数据高速缓存与指令内存也可以算作“处理器上下文”的一部分。Memory 类其实只是对一个数组，即假象中的内存的一种简单的封装。Cache 类具有与 Memory 类同样的接口，因为对于数据通路（Datapath）来说，通过高速缓存来访问内存的过程对其应当是透明的。具体 Cache 类的实现逻辑比较复杂，我将放到第四章详细说明。这一块内容对应上图的左下角那三个方块。

我们有了处理器的五大部件、寄存器、内存及其连线，这些都是“静态的”东西。如何让它们“动”起来呢？答案是必须有时钟，定时地供给这些部件时钟脉冲。这在我的设计中，被抽象为一个导演者（Director），可以把这些静态的东西理解为道具，导演用它们加上时间轴拍成了一部“运动”的电影。说白了，导演者的作用就是定时为处理器提供时钟脉冲，让处理器维持正常运作。

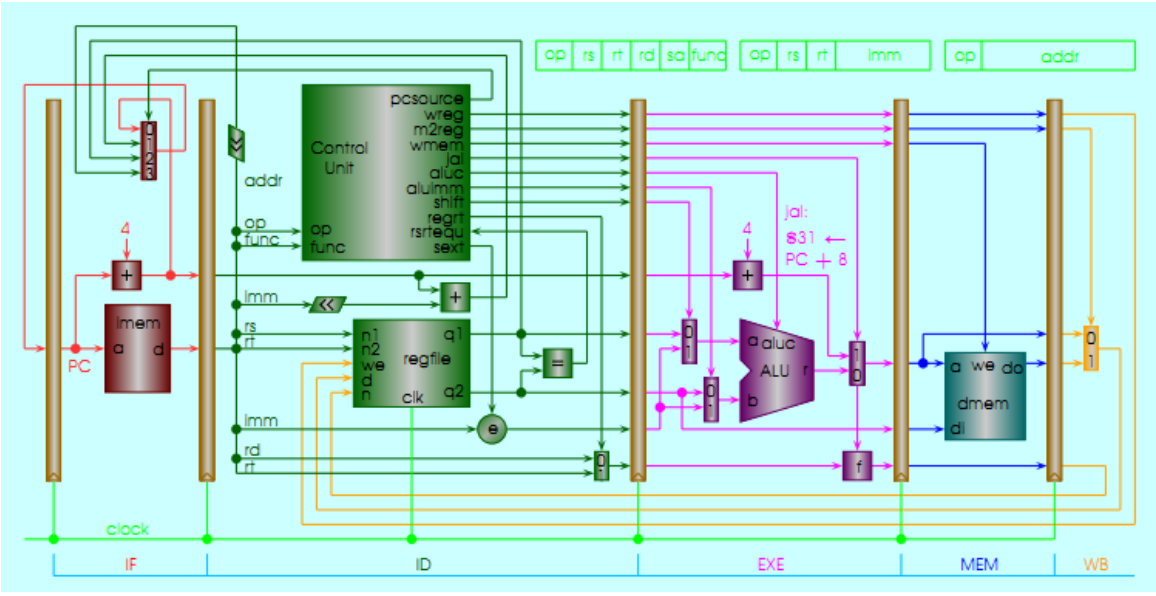
这部分对应的是上图的右下角三个方块。IDirector 表示一个抽象的导演者的接口，我实现了两种导演者，即两类不同的时钟脉冲提供方法。TimerDirector 类是我最初使用的，它关联了一个.NET Timer 对象，隔固定时间触发定时器事件，应该说这是最接近真实处理器的方式。但是经实践发现，由于处理器是我们用高级语言模拟的，有些操作的时间会花费较长时间，以致这一次触发的事件还未处理完毕，下一次的事件已经到来了，从而产生了错误。若统一制定一个较大的时钟间隔，又会导致整个处理器运行的时间过长。

再三思量之下，我设计了 IterativeDirector 类，它用循环的方式为处理器输入脉冲。尽管这样会导致节拍不整齐，但在我的模拟处理器中，这不是个问题，因为普通的程序设计语言，若不采用特殊复杂的方式，根本无法模拟精确时序。事实证明，这种方法工作得很好，既不会导致错误，也不至于降低运行效率。

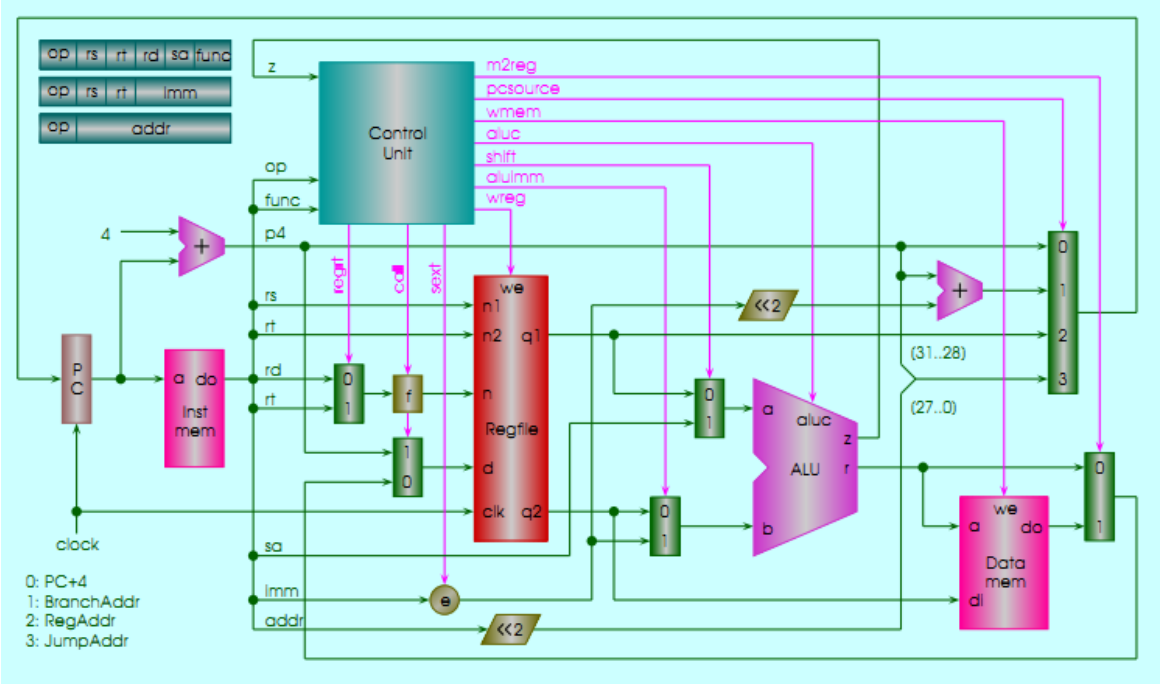
另外还有两个单独的类，一个是 AluControl，它其实只包含一系列整数常量，用来定义 ALU 部件要执行的操作的序号；另一个是 Program，包含了程序的入口点 Main 函数，其中创建了一个 Processor 对象与一个 ProcessorContext 对象，还创建了一个 IDirector 对象将两者关联起来，并启动处理器。

3.1.2 电路模拟

尽管在本质上，与真实处理器的运行方式有很大的不同，但就五级流水线来说，模拟处理器所依据的原理与真实处理器并无二致。所以，如下的处理器设计图完全可以用来描述模拟处理器，事实上，我也是依据下图完成的整个开发过程。



在该图中，有一些信号的设置有些小错误，因此开发过程中，我还参照了单周期 CPU 的设计图。



下面就 C#如何模拟数字电路，作简要的说明。我所依据的核心原则有四条：

a) 多路选择器用 switch 语句来描述。

b) 直连的线用赋值语句描述。

c) 同一阶段内的信号用 if 语句来描述。

d) 跨阶段的信号用流水线寄存器来传递。

比如，在 ID 阶段，sext 就被抽象为如下伪代码：

if sext=1 then

imm=sign_extend(imm)

既然讲到流水线寄存器，它的实现方式是这样的：考虑到，一个流水线寄存器在一个时钟周期内部，它将被位于它右边的阶段读取，而在时钟周期的末尾，它将被位于它左边的阶段写入。这在指令按序执行的 C#中是不可能实现的，因为当这一周期内，某一阶段的代码执行完毕后，若不保存，那么相关结果都将丢失，而不会像现实中的电路那样具有持续性。我的方案是：为流水线寄存器设置两组状态，一组状态供寄存器左方的阶段修改，另一组状态供寄存器右方的阶段读取，当发生时钟脉冲时，将两组状态交换即可。这种方式，和图形学中的双缓冲是一个思路，而且效率较高，不必在周期末再发生额外的赋值操作。

那么流水线寄存器中如何存储各信号呢？我的办法是为每组状态设置一个哈希表，键为信号名，值即为信号的内容。信号的命名需要遵循一定的规则，否则会导致重名错误（比如 wreg 信号就要经过 EXE 和 MEM 两个阶段的传递）。命名的规则是：<阶段名>:<信号名>，如 EXE 阶段的 wreg 信号被称为 exe:wreg，MEM 阶段的 m2reg 信号被称为 mem:m2reg。当发生状态交换时，只需将两个哈希表的引用交换即可。

3.1.3 五个阶段

本节对 MIPS32 处理器的五个阶段的功能及其实现方式作一概述。

根据观察，所有阶段在一个时钟周期内要做的事情，可以抽象为 5 个步骤：

a) 启动（Startup）：阶段开始运作时，要处理诸如空转（Stall）等事件。

b) 读取（Read）：从它左边的流水线寄存器中读取相关的信号值。
c) 传递（Pass）：对于在本阶段无需做任何处理的信号，应当将其传递到它右边的流水线寄存器，以便后一阶段使用。
d) 模拟（Simulate）：这步的名字我取的不是很好，想不出能准确描述它的词语。总之就是在该步中完成这个阶段的逻辑。
e) 写入（Write）：有一些信号，必须经过模拟步骤后，才能确定其值，并写入它右边的流水线寄存器。

五级流水线每一级的功能如下：

a) 指令提取（IF）：由程序计数器存储的地址访问相应指令单元，将其取到指令寄存器IR，同时在 PC 中产生下一条指令的地址。
b) 指令译码（ID）：对指令寄存器中的指令进行译码，确定该指令需要完成的操作，产生相应控制信号，确定其后各阶段针对该指令的动作。同时，还会访问寄存器堆读取寄存器内容，以备后面阶段使用。
c) 指令执行（EXE）：根据左侧流水线寄存器中的控制信号，执行相应的动作，得到运算结果，然后转移到下一阶段。
d) 存储器访问（MEM）：所有的存储器访问操作都在这个阶段中执行。首先从左侧流水线寄存器中读出访问存储器的地址，接着按控制信号，或者把数据写入到存储器相应地址处，或者从存储器中得到地址所标记的数据。
e) 寄存器写回（WB）：将 MEM 阶段通过流水线寄存器传递过来的结果写回到 regdst 信号指定的寄存器中。

3.2 三种指令相关

指令相关，也称为冒险（Hazard），是 CPU 流水线设计中的重大问题。相关的发生可能会导致流水线的停顿，就好比一个很长的在流动中的队伍，如果有一个人突然停下不动，那么就会导致他后面所有的人被迫停止前进。如果经常发生指令相关，那么整个“指令队伍”的行进效率就可想而知了。同时队伍中的“空缺”还可能会导致能量的白白消耗，性能功耗比也会因此下降。

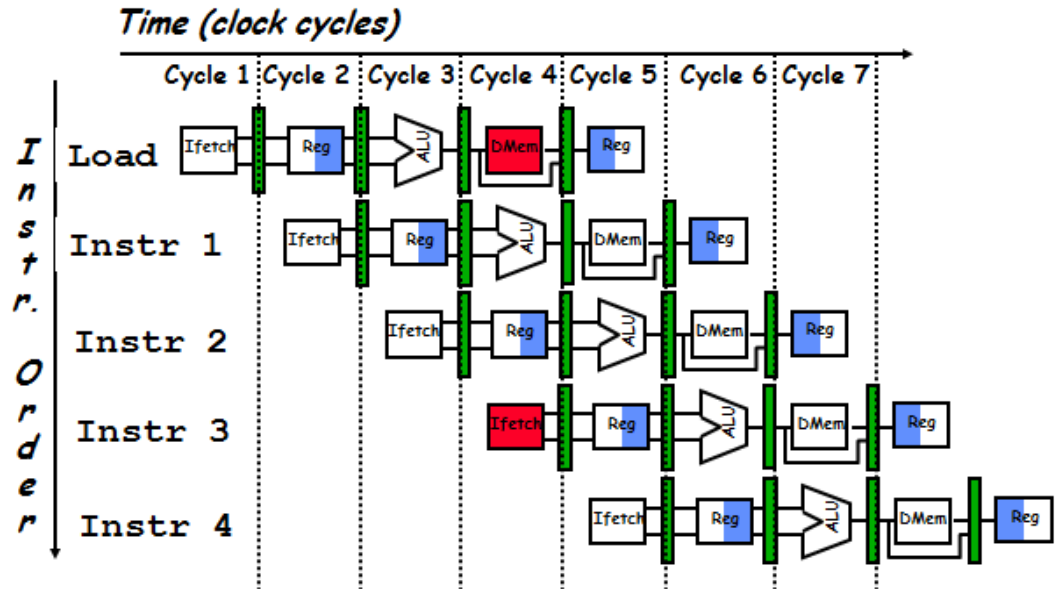
指令相关分为三种：

a) 结构相关（Structure Hazard）：由于硬件资源的不足引起。若采用指令数据合一的内存，就可能会出现取值与访存冲突。因此一般采用哈佛（Harvard）结构，即指令内存与数据内存分开，比较好。解决方案是：复制硬件资源，但通常会导致成本的上升。在 MIPS 处理器中，如果采用哈佛结构，那么不会出现结构相关问题。
b) 数据相关（Data Hazard）：流水线中的后进指令要用到先行指令的结果。在 MIPS 体系结构中，只可能存在 RAW（Read after Write）相关，也称为真相关。解决方案是：后面要提到的数据旁路技术（Forwarding）。
c) 控制相关（Control Hazard）：由分支或跳转指令引起的流水线中指令失效的问题。完美的解决方案：无。只能通过分支预测、延迟槽等技术降低控制相关发生的概率以及带来的代价。

下面的图来源于 David Patterson 的 PPT，直观地显示了三种 Hazard 可能发生的情况：

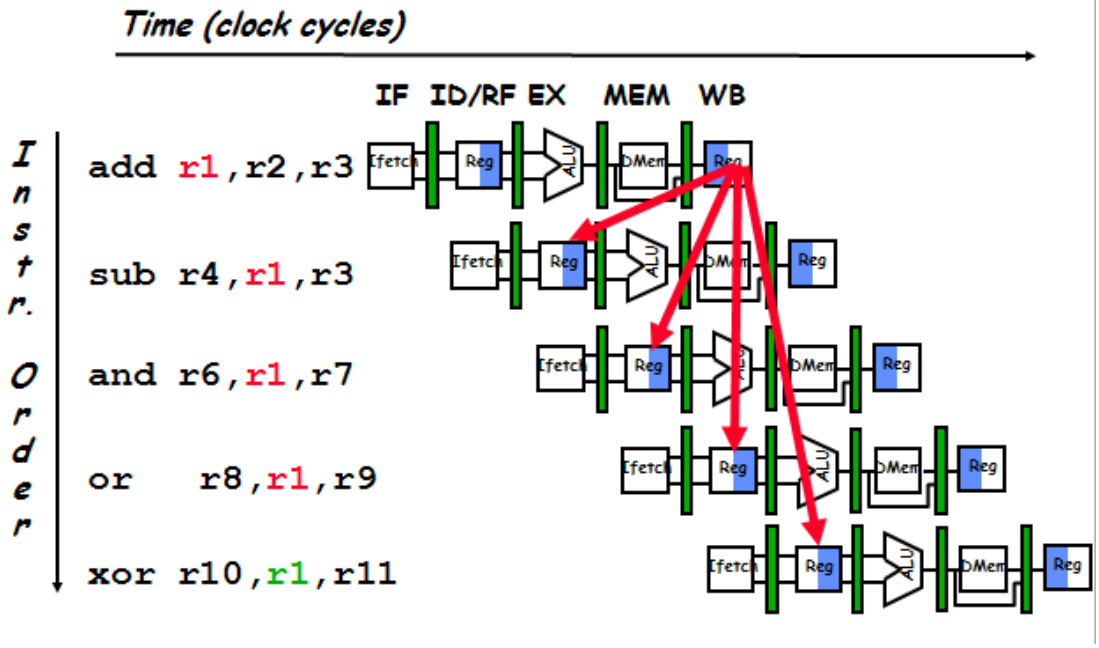
One Memory Port/Structural Hazards

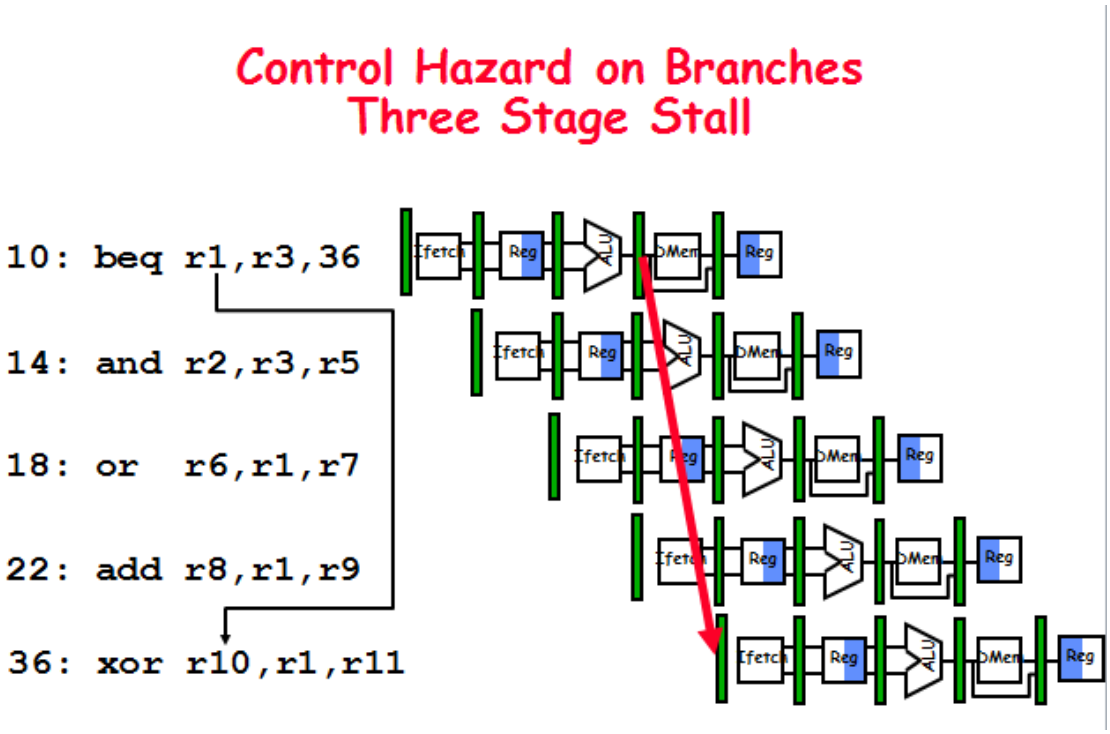
Figure 3.6, Page 142 , CA:AQA 2e



Data Hazard on R1

Figure 3.9, page 147 , CA:AQA 2e





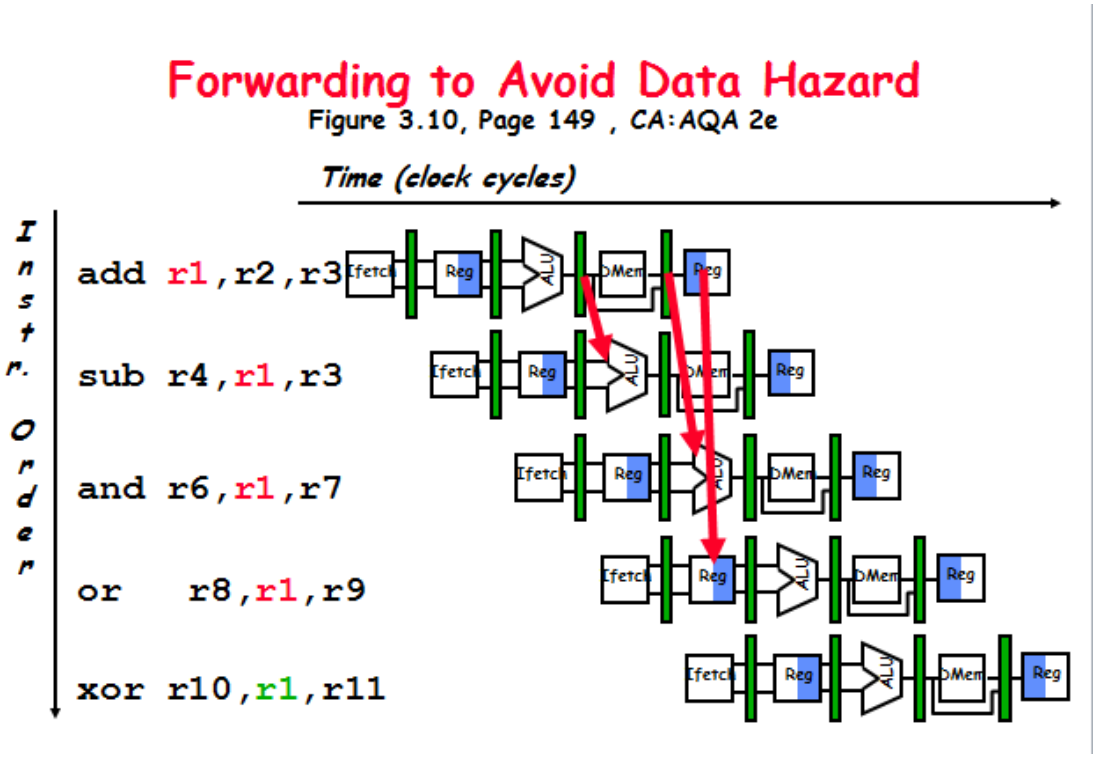
3.3 空转的实现

实现阶段空转较为容易，只需为每个阶段设置一个信号：<阶段名>:stall 即可。当一个阶段开始执行时，它发现 stall=1，那么这个阶段它就不需要做任何事。同时由于该阶段空转，会造成其后一个阶段在下一个时钟周期也空转，因此还需要设置：<后一个阶段名>:stall=1。

但当某个阶段的空转被控制器或者其前一阶段取消以后，它也必须取消后一个阶段在下一时钟周期的空转，以防流水线堵死。

3.4 数据旁路技术

数据旁路（Forwarding），是一种将寄存器结果，在被写入目标寄存器前，提前交给 ALU 进行运算的技术。如下图，演示了一种可以旁路推送的情况（注意与上面倒数第二幅图的对比）。



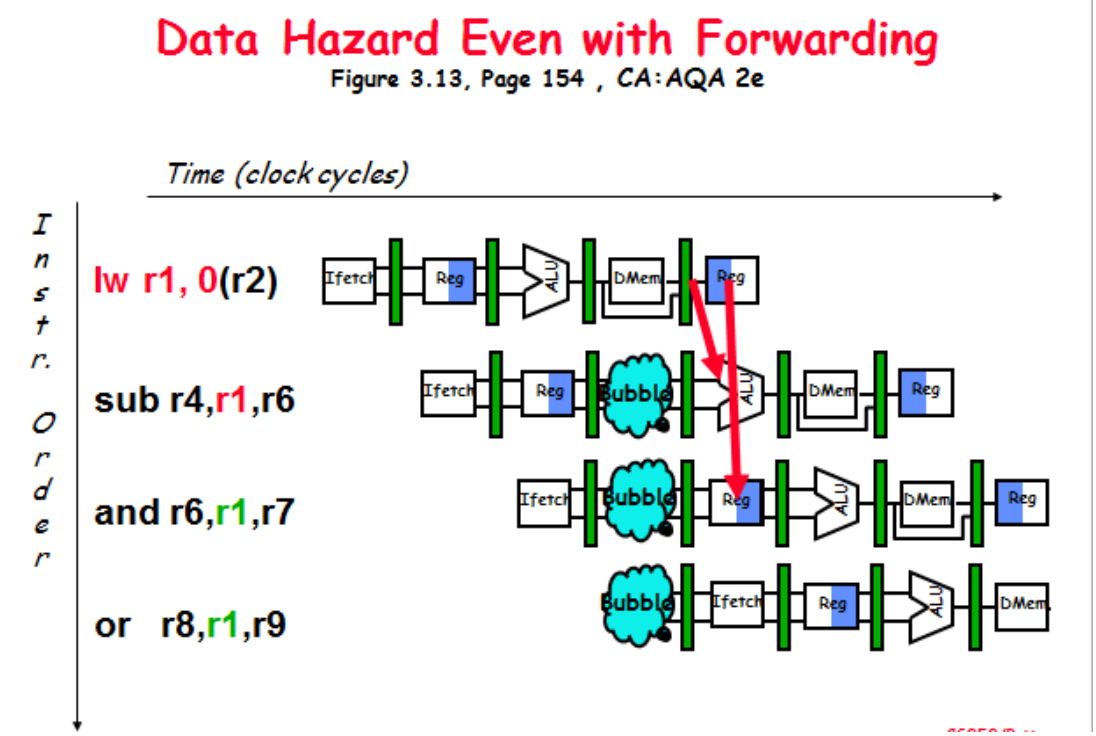
数据旁路技术一共有三种类型：

a) 从 EXE 阶段推送 ALU 结果至 ID 阶段。

b) 从 MEM 阶段推送 ALU 结果至 ID 阶段。

c) 从 MEM 阶段推送 MEM 结果至 ID 阶段。

利用数据旁路技术可以很有效地缩短平均 CPI。但同时，我们必须理解的是，数据旁路技术并不能解决所有的问题，比如下图中，即便使用了该技术，仍将导致一次空转。



数据相关的检测表现在控制器，即 ID 阶段，就是一系列对信号的判断语句，较为冗长，这里不再赘述。需要注意的是，同一条指令可能会与多条指令发生数据相关，这可能需要同时做多个推送。

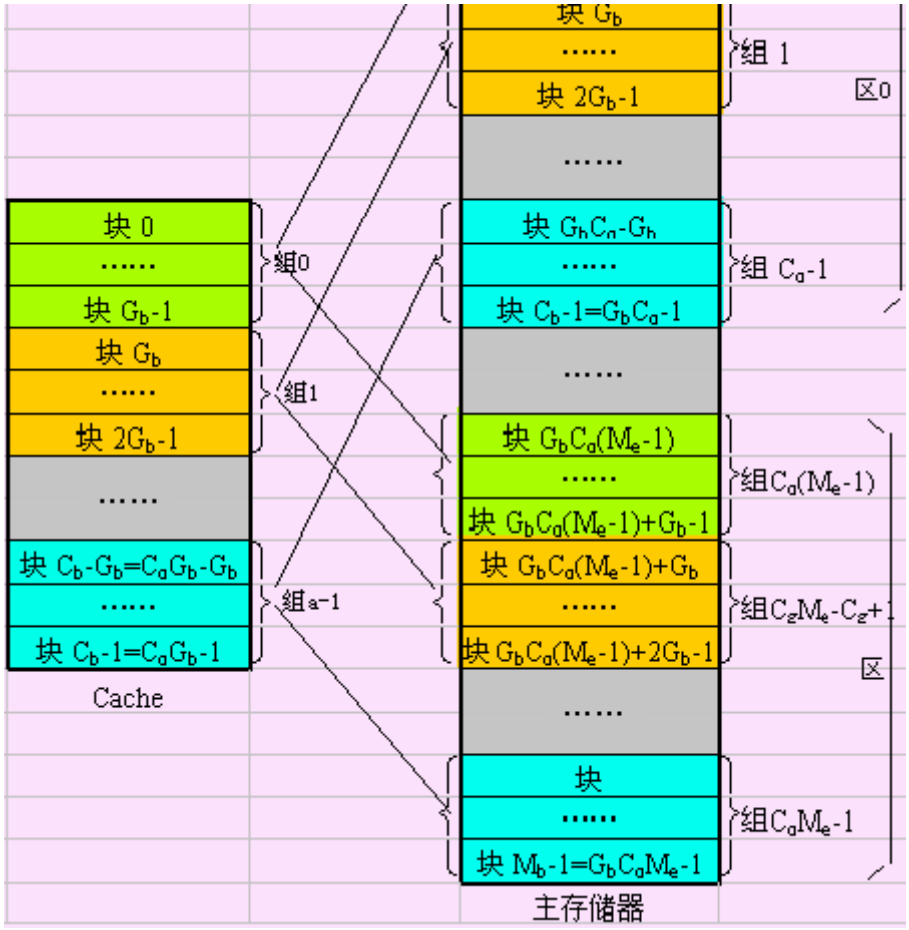
在我的设计中，数据相关的检测是在 ID 阶段的写步骤中完成的，因为它需要将一系列推送信号写入右侧流水线寄存器中，让后一阶段实现数据旁路。

第四章 高速缓存

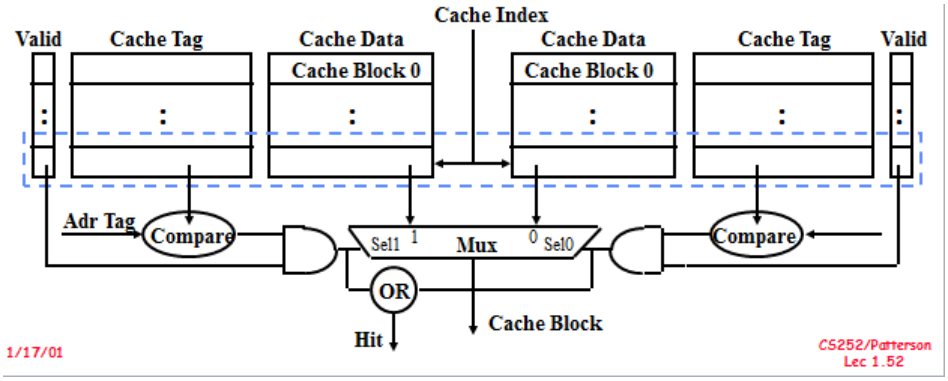
4.1 概述

高速缓存（Cache）作为处理器寄存器与内存之间的缓冲器，能够卓越地影响处理器的性能。本处理器中实现的高速缓存为 8 路组相联映象，每个行大小为 16 个字，一共 16 组，总大小为 8KB。

采用的组织结构，如下图，非常传统，看上去也非常直观：



但实际上，经过体系结构课的考试，我发现这样的组织方式，对硬件的排布来说非常的不方便。更好的组织方式如下图（2 路组相联）：



即横向看，所有位于同一水平高度的 Cache Block 组成一个组。

关于块淘汰算法，本程序中使用的是类 LRU 算法，即每次访问都为该块的计数加一，淘汰时选择计数最低的块。淘汰时才将脏块写入内存，因此此高速缓存为写回式（Write Back）。

当发生写缺失时，我设计为，必须先将对应的内存单元读入高速缓存后，才能进行写入，即写分配式（Write Allocation）。

4.2 读操作

读操作的步骤如下：

a) 根据内存地址，分离出标记（Tag）、组号、块内偏移（Block Offset）。
b) 在组号对应组内，8 路并行查找与给定标记相匹配的缓存块。查找的同时记下第一个空闲块（如果有的话），以及访问计数最低的块（必须是有效块）。
c) 若找到，转到 f)；若未找到，继续下列步骤。
d) 若有空闲块，则选择空闲块，将相应内存单元读入该块中，转到 f)；若没有，继续下列步骤。
e) 判断访问计数最低的块是否脏（Dirty）了，若是，则应将块内容写入内存对应单元；若不是，则直接读入相应内存单元覆盖该块。
f) 为选定块的访问计数加一（初始为 0），用块内偏移进行索引，得到所需要的字后返回给处理器。

4.3 写操作

写操作的步骤如下：

a) 根据内存地址，分离出标记（Tag）、组号、块内偏移（Block Offset）。
b) 在组号对应组内，8 路并行查找与给定标记相匹配的缓存块。查找的同时记下第一个空闲块（如果有的话），以及访问计数最低的块（必须是有效块）。
c) 若找到，转到 f)；若未找到，继续下列步骤。
d) 若有空闲块，则选择空闲块，将相应内存单元读入该块中，转到 f)；若没有，继续下列步骤。
e) 判断访问计数最低的块是否脏（Dirty）了，若是，则应将块内容写入内存对应单元；若不是，则直接读入相应内存单元覆盖该块。
f) 修改脏位（Dirty Bit）为 1，为选定块的访问计数加一,（初始为 0），用块内偏移进行索引，写入对应的字。

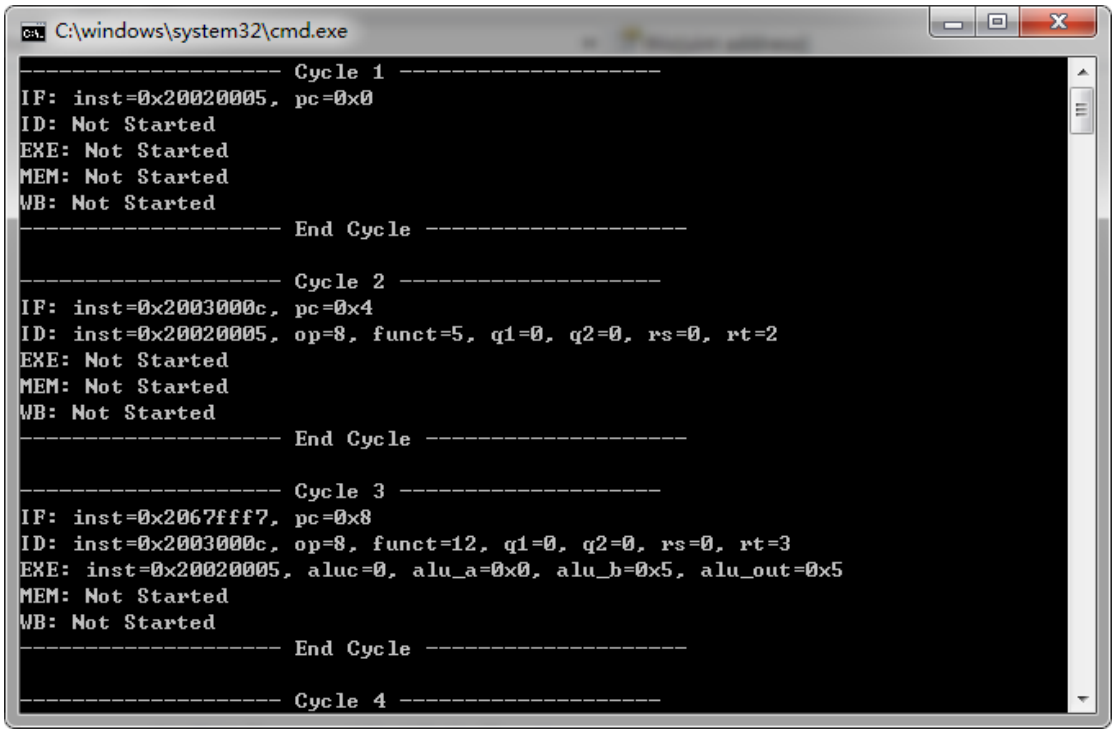
第五章 测试

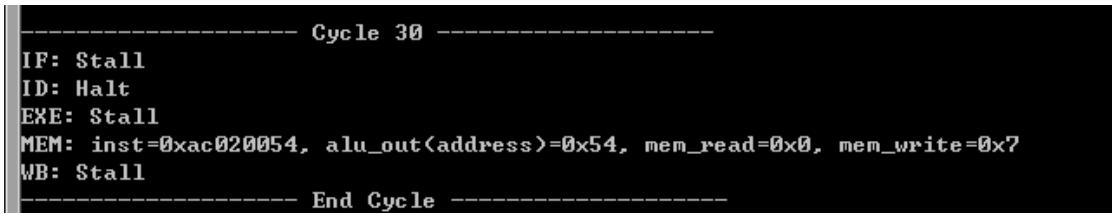
5.1 用例

```
1  #        Assembly                Description
2  main:    addi $2, $0, 5          # $2 = 5
3           addi $3, $0, 12         # $3 = 12
4           addi $7, $3, -9         # $7 = 3
5           or   $4, $7, $2         # $4 = 3 | 5 = 7
6           and  $5, $3, $4         # $5 = 12 & 7 = 4
7           add  $5, $5, $4         # $5 = 4 + 7 = 11
8           beq  $5, $7, end        # shouldn't be taken
9           slt  $4, $3, $4         # $4 = 12 < 7 = 0
10          beq  $4, $0, around     # should be taken
11          addi $5, $0, 0          # shouldn't happen
12          addi $5, $0, 0          # shouldn't happen
13          addi $5, $0, 0          # shouldn't happen
14          addi $5, $0, 0          # shouldn't happen
15 around:  slt  $4, $7, $2         # $4 = 3 < 5 = 1
16          add  $7, $4, $5         # $7 = 1 + 11 = 12
17          sub  $7, $7, $2         # $7 = 12 - 5 = 7
18          sw   $7, 68($3)         # [80] = 7
19          lw   $2, 80($0)         # $2 = [80] = 7
20          j end                   # should be taken
21          addi $2, $0, 1          # shouldn't happen
22          addi $2, $0, 1          # shouldn't happen
23          addi $2, $0, 1          # shouldn't happen
24          addi $2, $0, 1          # shouldn't happen
25          addi $2, $0, 1          # shouldn't happen
26          addi $2, $0, 1          # shouldn't happen
27 end:     sw   $2, 84($0)         # [84] = 7
```

5.2 结果与分析

若运行正确，最后应当向内存地址 84 写入数值 7。模拟处理器的运行截图如下：





可以看到结果是正确的，一共花了 30 个时钟周期。

第六章 结束语

6.1 心得与体会

以前一直觉得 MIPS 处理器运行机制就那么回事，上课看 PPT 也看得懂，可以说有了感性认识。但是当真正着手开始做的时候，才知道：个中复杂超乎了我的想象。现在懂得了，什么叫实践出真知，光靠课本上、课堂上看到的、听到的是不足以真正掌握一样复杂的东西的。

同时，做这个大作业，对我的益处，不光是加深了对高级计算机体系结构这门课的理解。更重要的是教会了我一种思维方法，那就是计算机的大脑到底是怎么处理问题的？只有了解了计算机“思维”的机制，才能够更好地运用它。

这次的大作业还是对人意志的一种磨练。由于我采用的工具比较另类，不是很符合硬件设计的需要，因此在电路模拟这块，我动足了脑筋。好不容易处理完电路方面的事情，还要纠结各个信号在各阶段到底该如何传递和使用。在开发过程中，遇到了无数的悬疑 bug，我只能一遍又一遍跟踪程序的执行来发现错误的根源。这绝对是对耐心的一种考验。

其实当你对一样东西一知半解的时候，你总要走很多弯路。但是从弯路走出来，你对这样东西的理解就会加深很多。任何知识的学习都是这样一个曲折反复的过程。

最重要的，在这个过程中，不能气馁，不能怀有恐惧感，要相信一切总会有光明的时候。冷静看待问题，问题才会迎刃而解！

6.2 不足与改进

尽管测试用例运行通过，但是这个处理器一定还存在很多的 bug，而且我感觉执行的时钟周期数似乎也略微超过了预期。我分析下来，是我在碰到 hazard 要 stall 时过于保守，不管三七二十一将 IF 阶段一起 stall 了，这样虽然能保证逻辑上一定正确，但是效率却不能保证。我尝试过将 IF 阶段 stall 适时去掉，但是总会引发各种各样其他的问题，时间有限，我不得不暂时放弃了。

归根结底，C#之类的高级语言是偏向于人类大脑的，根本不适合做硬件设计与仿真这样底层的工作，这是我最初没有想到的。即便要电路仿真，也应当从连续时序的角度去做，按序执行或者简单的并发执行是没有办法匹敌自然界真正的并行和连续性的。

就纯设计方面，其实也存在很多可以改进的地方。比如用哈希表方式存储节点、寄存器状态信息，不如用阶段间的责任链模式传递起来方便，而且如果采用了这种设计模式，连 stall 的实现都可以简化了。只是这样做，和真实 CPU 的样子就越离越远了。

总之，如果今后还有机会做 CPU 的话，那么一定要用 Quartus 或者 MaxPlus 这样的专业软件进行设计。选错工具，到头来没什么成就倒也罢了，还可能把自己搞神伤了。