浙江大学

数字系统设计实验Ⅱ

实验名称 流水线 MIPS 微处理器设计

姓　　名 张先喆

学　　号 3110100812

授课教师 屈民军　唐奕　马洪庆

专　　业 电子科学与技术

年　　级 2011 级

时　　间 2013 年 12 月

上课时间 单周六上午

目录

流水线 MIPS 微处理器设计

张先喆 3110100812

专业：电子科学与技术

日期：2013 年 11 月-12 月

一、实验目的

1. 了解提高 CPU 性能的方法。
2. 掌握流水线 MIPS 微处理器的工作原理。
3. 理解数据冒险、控制冒险的概念以及流水线冲突的解决方法。
4. 掌握流线 MIPS 微处理器的测试方法

二、实验内容

设计一个 32 位流水线 MIPS 微处理器，具体要求如下所述。

1. 至少运行下列 MIPS32 指令。

（1）算数运算指令：ADD、ADDU、SUB、SUBU、ADDI、ADDIU。

（2）逻辑运算指令：AND、OR、NOR、XOR、ANDI、ORI、XORI、SLT、SLTU、SLTI、SLTIU。

（3）移位指令：SLL、SLLV、SRL、SRLV、SRA。

（4）条件分支指令：BEQ、BNE、BGEZ、BGTZ、BLEZ、BLTZ。

（5）无条件跳转指令：J、JR。

（6）数据传送指令：LW、SW。

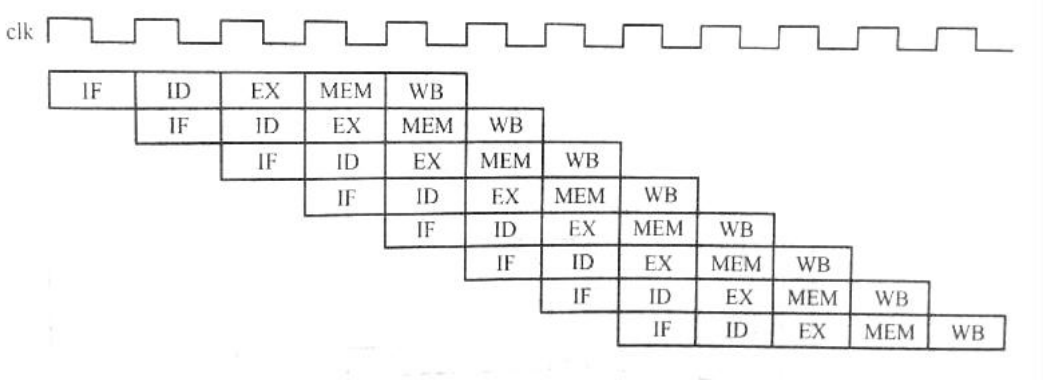
（7）空指令：NOP。

2. 采用 5 级流水线技术，对数据冒险实现转发或阻塞功能。
3. 在 XUP Virtex-Ⅱ Pro 开发系统中实现 MIPS 微处理器，要求 CPU 的运行速度大于 25MHz。

三、实验原理

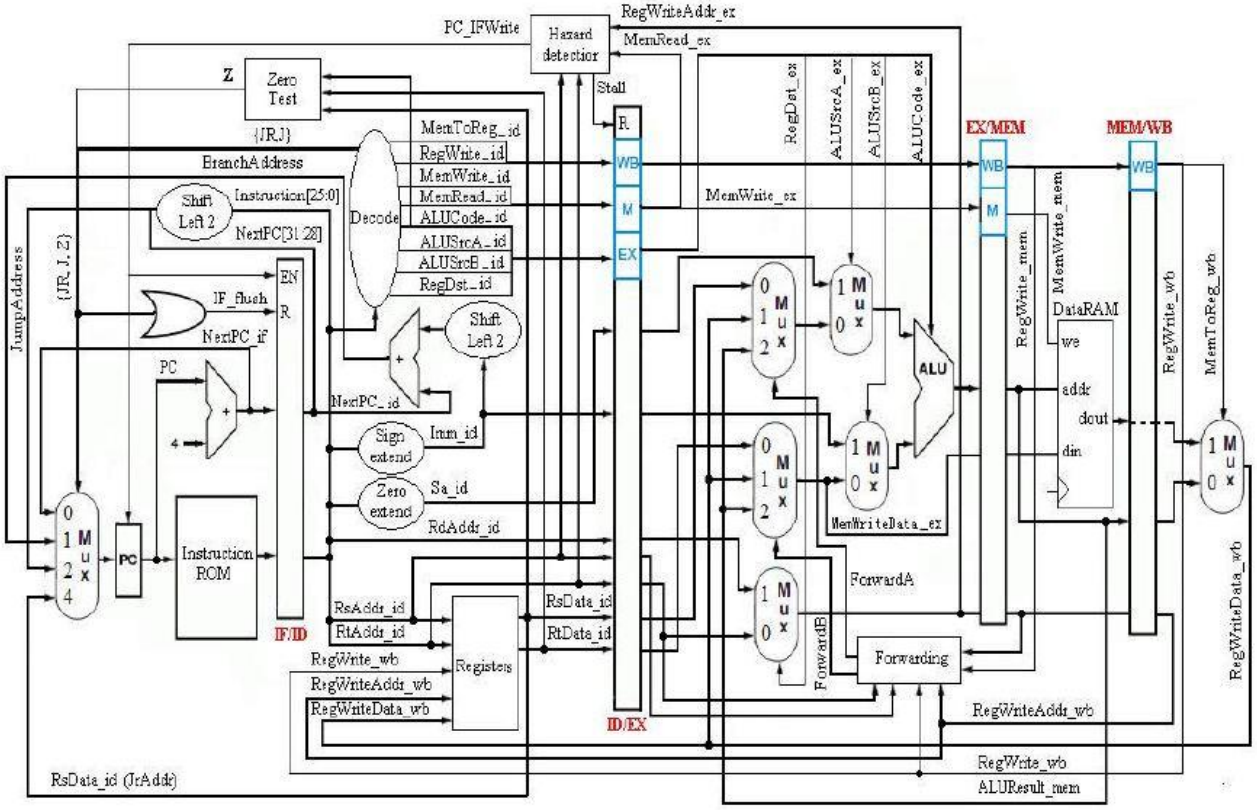
1. 总体设计

根据 MIPS 处理器指令的特点，将整体的处理过程分为取指令（IF）、指令译码（ID）、执行（EX）、存储器访问（MEM）和寄存器回写（WB）五级，对应多周期 CPU 的五个处理阶段。一个指令的执行需要 5 个始终周期，每个时钟周期的上升沿来临时，此指令所代表的一系列数据和控制信息将转移到下一级处理，主要过程如下图所示：



为设计符合要求的流水线 MIPS 微处理器，作出以下框图，采用五级流水线。为了在其它四级流水线中各条指令保持各自的值，从指令存储器中读出的指令必须保存在寄存器中。同样的方法应用到每个流水线步骤中，需要在上图中各级分割线处都加入寄存器。在流水线中转移的数据和控制信号的命名格式为：

变量名_流水线级名称



1.1 流水线中的控制信号

（1）IF 级：取指令级。从 ROM 中读取指令，并在下一个时钟沿到来时把指令送到 ID 级的指令缓冲器中。该级控制信号决定下一个指令指针的 PCSource 信号、阻塞流水线 PC_IFwrite 信号、清空流水线的 IF_flush 信号。

（2）ID 级：指令译码级。对 IF 级来的指令进行译码，并产生相应的控制信号。整个 CPU 的控制信号基本都是在这级上产生。该级自身不需要任何控制信号。

流水线冒险检测也在该级上进行，冒险检测电路需要上一条指令的 MemRead，即在检测到毛细那条件成立时，冒险检测电路会产生 stall 信号清空 ID/EX 寄存器，插入一个流水线气泡。

（3）EX 级：执行级。此级进行算数或逻辑操作。此外 LW、SW 指令所用的 RAM 访问地址也是在本级上实现。控制信号有 ALUCode、ALUSrcA、ALUSrcB 和 RegDst，根据这些信号确定 ALU 操作、选择两个 ALU 操作数 A、B，并确定目标寄存器。

另外，数据转发也在该级完成。数据转发控制电路产生 FowardA 和 FowardB 两组控制信号。

（4）MEM 级：存储器访问级。只有在执行 LW、SW 指令时才对存储器进行读写，对其他指令只起到缓冲一个周期的作用。该级只需要存储器写操作允许信号 MemWrite。

（5）WB 级：回写级。此级把指令执行的结果回写到寄存器文件中。该级设置信号 MemToReg 和寄存器写操作允许信号 RegWrite，其中 MemToReg 决定写入寄存器的数据来自于 MEM 级上的缓冲值或来自于 MEM 级上的存储器。

1.2 流水线冒险

在流水线 CPU 中，多条指令同时执行，由于各种各样的原因，在下一个时钟周期中下一条指令不能

执行，这种情况称为冒险。冒险分为三类：

①结构冒险：硬件不支持多条指令在同一个时钟周期内执行。MIPS 指令集专为流水线设计，因此在 MIPS CPU 中不存在此类冒险。

②数据冒险：在一个操作必须等待另一操作完成后才能进行时，流水线必须停顿，这种情况称为数据冒险。数据冒险又分为两类：

数据相关：流水线内部其中任何一条指令要用到任何其它指令的计算结果时，将导致数据冒险。通常可以用数据转发（数据定向）来解决此类冒险。

数据冒险：此类冒险发生在当定向的目标阶段在时序上早于定向的源阶段时，数据转发无效。通常是引入流水线阻塞，即气泡（bubble）来解决。

③控制冒险：CPU 需要根据分支指令的结果做出决策，而此时其它指令可能还在执行中，这时会出现控制冒险，也称分支冒险。解决此类冒险的常用方法为延迟分支。

1.2.1 数据相关与转发

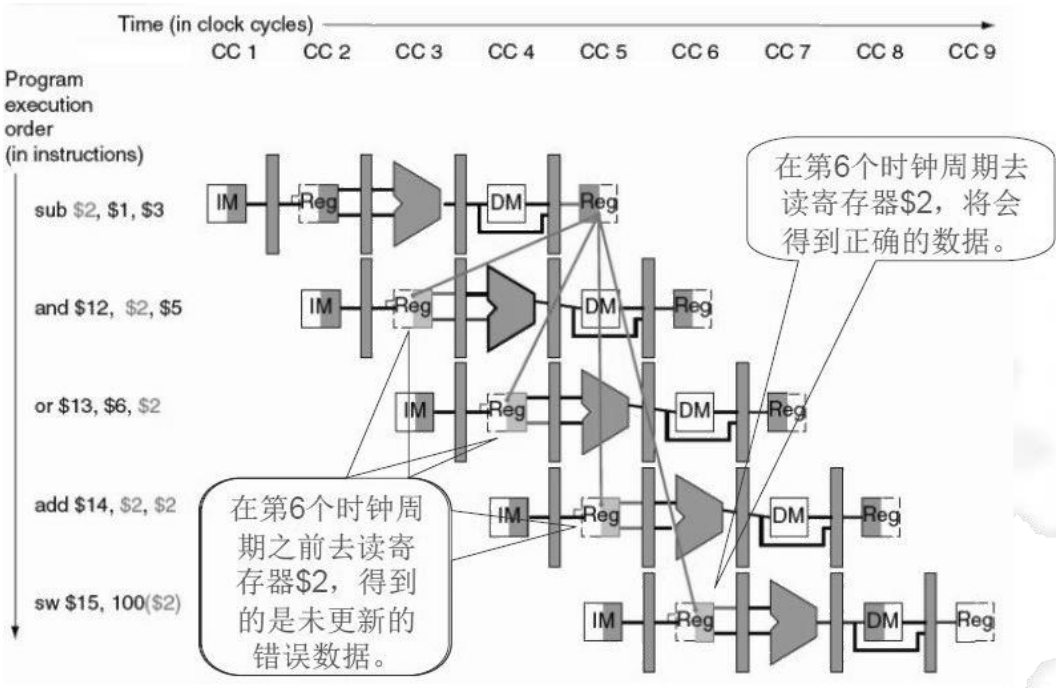
如果上一条指令的结果还没有写入到寄存器中，而下一条指令的源操作数又恰恰是此寄存器的数据，那么，它所获得的将是原来的数据，而不是更新后的数据。这样的相关问题成为数据相关。这种问题可以通过数据转发和插入流水线气泡的方法解决此类相关问题。

以具体实例来解释数据相关的概念。

```
sub     $2, $1, $3      //Register $2 written by sub
and     $12, $2, $5     //lst operand($2) depends on sub
or      $13, $6, $2     //2nd operand($2) depends on sub
add     $14, $2, $2     //1st($2) & 2nd($2) depend on sub
sw      $15,100($2)     //Base ($2) depends on sub
```

后 4 条指令都依赖于第一条指令得到的寄存器$2 的结果。但是，sub 指令只在第 5 个时钟周期时才写回寄存器$2，而 and 指令在第 3 个时钟周期、or 指令在第 4 个时钟周期、add 指令在第 5 个时钟周期就分别要用到寄存器$2 中的值。这种数据之间的相互关联引起的冒险就叫数据相关。

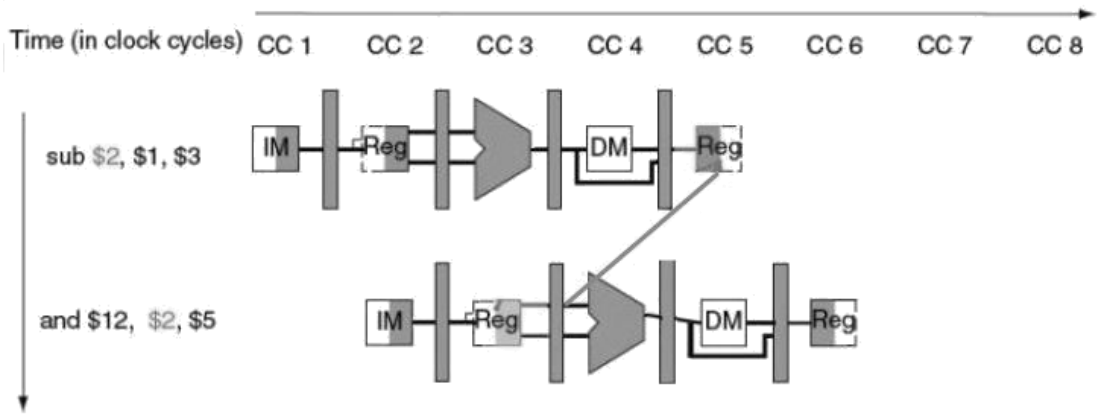
5 条指令在流水线中的执行情况：



当一条依赖关系的方向与时间轴的方向相反时，就会产生数据冒险。

（1）一阶数据相关（EX 冒险）

在这段代码中，与 sub 指令相关的指令有三条，首先是 sub 指令和 and 指令之间的相关问题。



sub 指令在第 5 时钟周期写回寄存器，而 and 指令在第 4 时钟周期就对 sub 指令的结果提出了请求，很显然 and 指令读取的数据是未被更新的错误内容。这类数据相关称之为一阶数据相关。仔细观察 sub 指令，可以发现，sub 指令的结果其实在 EX 级结尾，即第 3 时钟周期末就产生了。而 and 指令在第 4 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，只要 sub 指令结果产生之后，就可以直接将它“转发”给 and 指令，而不需要先写回寄存器堆，再由 and 指令将其读取出来。

转发机制的硬件实现:

转发条件 ForwardA、ForwardB 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；若转发条件成立，则 ALU 操作数取自数据旁路。

转发单元：判断转发条件是否成立。在 EX 级完成。

转发条件：

①在 MEM 级的指令需要写回寄存器，即 RegWrite_mem=1

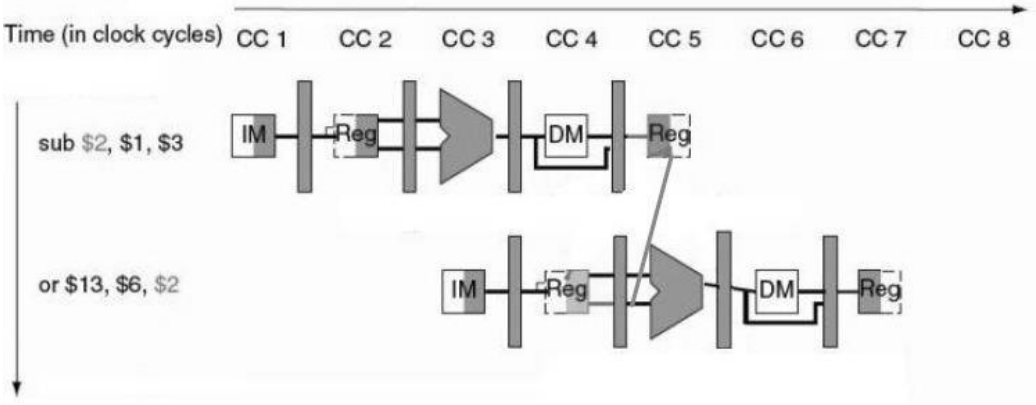
②在 MEM 级的指令在写回时,目标寄存器不能是寄存器$0，即 RegWriteAddr_mem≠0

③在 MEM 级的指令写回时的目标寄存器与在 EX 级的指令的源寄存器是同一寄存器，即:

RegWriteAddr_mem=RsAddr_ex 或 RegWriteAddr_mem=RtAddr_ex

（2）二阶数据相关（MEM 冒险）

我们现在讨论 sub 指令和 or 指令之间的相关问题。



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。这类数据相关称之为二阶数据相关。

如前所述，or 指令在第 5 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在 WB 级的指令需要写回寄存器，即 RegWrite_wb=1

②在 WB 级的指令在写回时,目标寄存器不能是寄存器$0，即 RegWriteAddr_wb≠0

③在 WB 级的指令写回时的目标寄存器与在 EX 级的指令的源寄存器是同一寄存器，即:

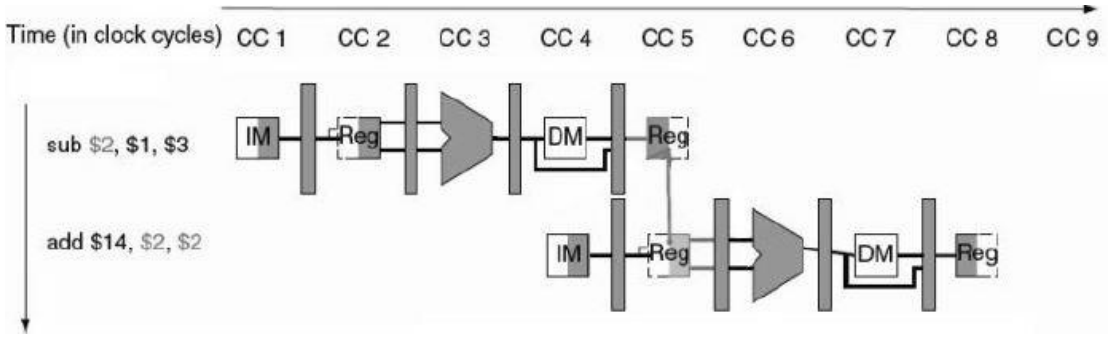
RegWriteAddr_wb=RsAddr_ex 或 RegWriteAddr_wb=RtAddr_ex

④EX 冒险不成立，即:

RegWriteAddr_mem ≠ RsAddr_ex 或 RegWriteAddr_mem ≠ RtAddr_ex

（3）三阶数据相关

在这段代码中，与 sub 指令相关的指令有三条，我们先来讨论 sub 指令和 add 指令之间的相关问题。



这两条指令在第 5 时钟周期内同时读写同一个寄存器。这类数据相关称之为三阶数据相关。

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。

因此，采用“转发”机制来解决三阶数据相关冒险。该部分转发电路我们放在寄存器堆的设计中完成。

转发条件：

①在 WB 级的指令需要写回寄存器，即 RegWrite_wb=1

②在 WB 级的指令在写回时,目标寄存器不能是寄存器$0，即 RegWriteAddr_wb≠0

③在 WB 级的指令写回时的目标寄存器与在 ID 级的指令的源寄存器是同一寄存器，即:

RegWriteAddr_wb=RsAddr_id 或 RegWriteAddr_wb=RtAddr_id

1.2.2 数据冒险与阻塞

当一条指令试图读取一个寄存器，而它前一条指令是 lw 指令，并且该 lw 指令写入的是同一个寄存器时，定向转发的方法就无法解决问题。

这类冒险不同于数据相关冒险，需要单独一个冒险检测单元（Hazard Detector）。它在 ID 级完成。

冒险成立的条件为：

①上一条指令是 lw 指令，即 MemRead_ex =1

②在 EX 级的 lw 指令与在 ID 级的指令读写的是同一个寄存器，即:

RegWriteAddr_ex = RsAddr_id 或 RegWriteAddr_ex = RtAddr_id

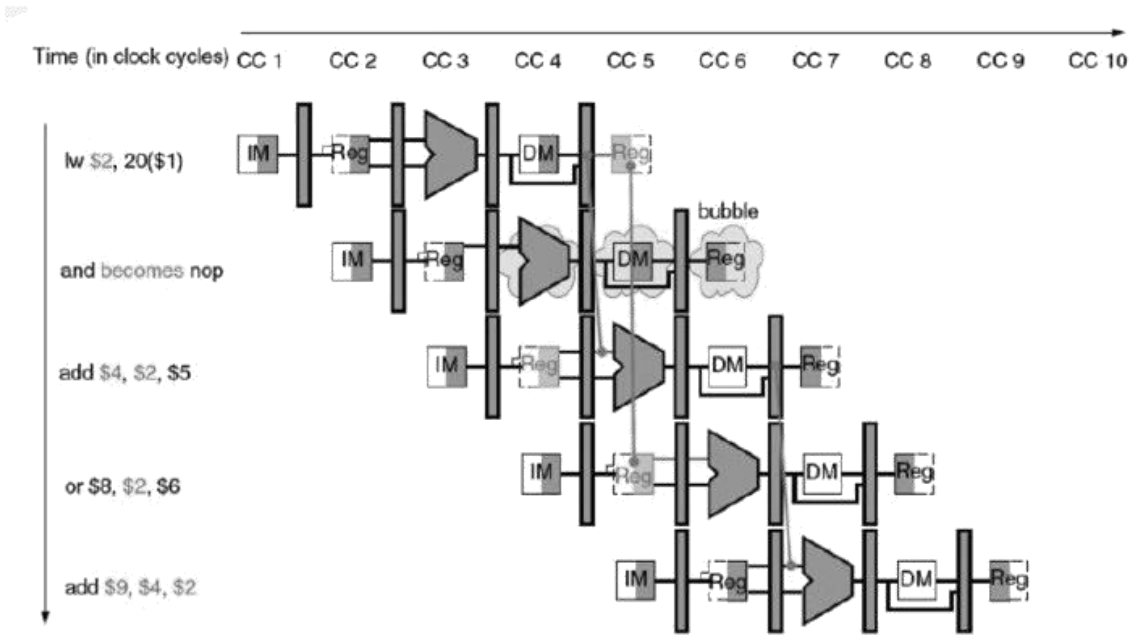
冒险的解决：引入流水线阻塞。

当 Hazard Detector 检测到冒险条件成立时，在 lw 指令和下一条指令之间插入阻塞，即流水线气泡（bubble），使后一条指令延迟一个时钟周期执行，这样就将该冒险转化为二阶数据相关，可用转发解决。

需要注意的是，如果处于 ID 级的指令被阻塞，那么处于 IF 级的指令也必须阻塞，否则，处于 ID 级

的指令就会丢失。防止这两条指令继续执行的方法是：

保持 PC 寄存器和 IF/ID 流水线寄存器不变，同时插入一个流水线气泡。

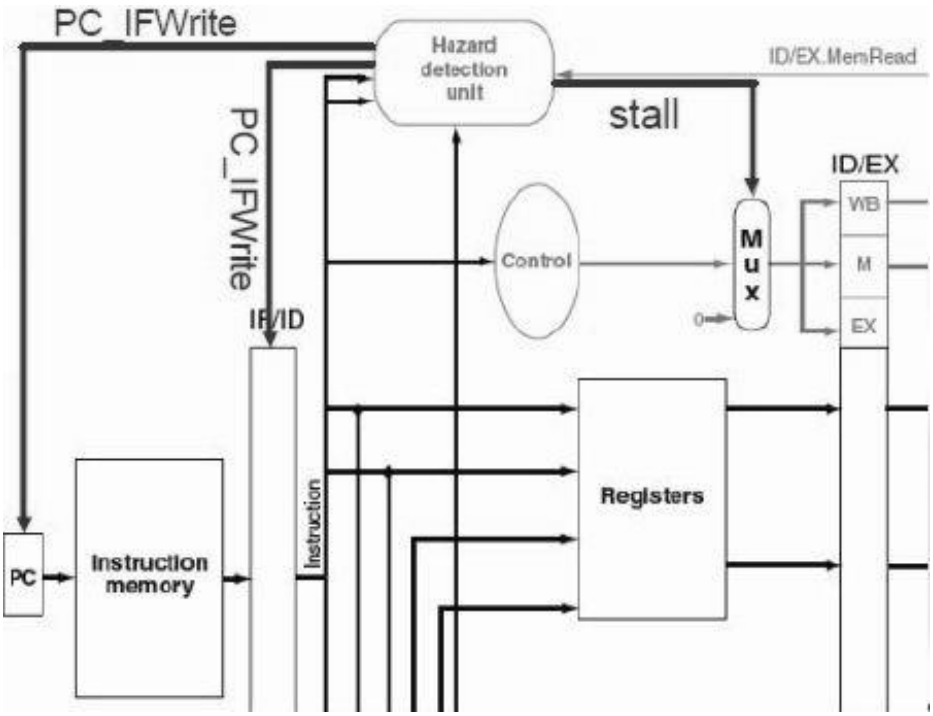


保持 PC 寄存器和 IF/ID 流水线寄存器不变：

在 ID 级检测到冒险条件时， Hazard Detector 输出一个信号：PC_IFWrite，作为使能信号同时送给 PC 寄存器和 IF/ID 流水线寄存器。冒险成立时，该信号为低电平，禁止 PC 寄存器和 IF/ID 流水线寄存器接收新数据。

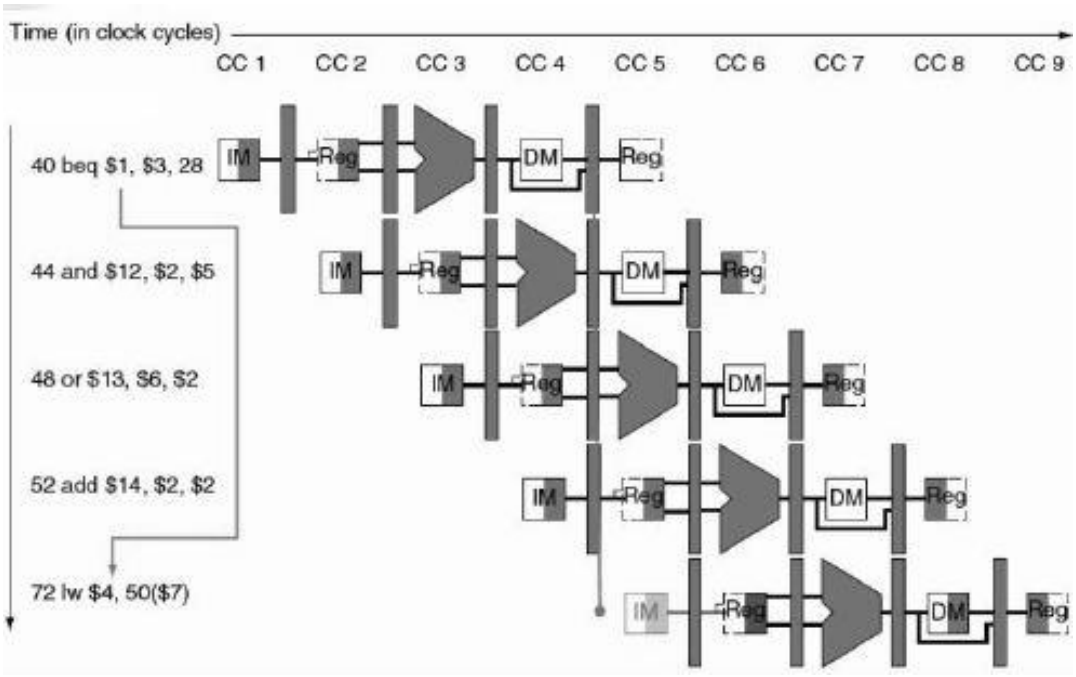
流水线气泡的插入：

在 ID 级检测到冒险条件时， HazardDetector 输出一个信号：stall，将 ID/EX 流水线寄存器中的 EX、MEM 和 WB 级控制信号全部清零。这些信号传递到流水线后面的各级，由于控制信号均为零，所以不会对任何寄存器和存储器进行写操作。



1.2.3 分支冒险

以上各类冒险局限于算术运算和数据传输中。还有一类冒险就是包含分支的流水线冒险。如下图所示：



流水线每个时钟周期都得取指令才能维持运行，但分支指令必须等到 MEM 级才能确定是否执行分支。这种为了确定预取正确的指令而导致的延迟叫做控制冒险或分支冒险。

分支冒险的解决方法：提前分支指令

一种比较普遍的提高分支阻塞速度的方法是假设分支不发生，并继续执行顺序的指令流。如果分支发生的话，就丢弃已经预取并译码的指令，指令的执行沿着分支目标继续。由于分支指令直到 MEM 级才能确定下一条指令的 PC，这就意味着为了丢弃指令必须将流水线中的 IF、ID 和 EX 级的指令都清除掉（flush）。这种优化方法的代价较大，效率较低。

假如在流水线中提前分支指令的执行过程，那么就能减少需要清除的指令数。这是一种提高分支效率的方法，降低了执行分支的代价。

提前分支指令需要提前完成两个操作：

①计算分支的目的地址

②判断分支指令的跳转条件。

提前计算分支的目的地址：由于已经有了 PC 值和 IF/ID 流水线寄存器中的指令值，所以可以很方便地将 EX 级的分支地址计算电路移到 ID 级。我们针对所有指令都执行分支地址的计算过程，但只有在需要它的时候才会用到。

提前判断分支指令的跳转条件：将用于判断分支指令成立的 Zero 信号检测电路（Z test ）从 ALU 中独立出来，并将它从 EX 级提前至 ID 级。具体的设计将在 ID 级设计中介绍。

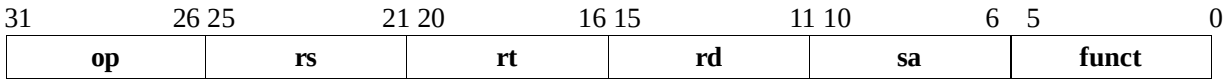
在提前完成以上两个操作之外，我们还需丢弃 IF 级的指令。具体做法是：加入一个控制信号 IF_flush，做为 IF/ID 流水线寄存器的清零信号。当分支冒险成立，即 Z=1，则 IF_flush=1，否则 IF_flush=0，故 IF_flush = Z。

考虑到本系统还要实现的无条件跳转指令：J 和 JR，在执行这两个指令时也必须要对 IF/ID 流水线寄存器进行清空，因此， IF_flush 的表达式应表示为：

$$IF\_flush = Z \,||\, J \,||\, JR$$

1.3 MIPS 指令格式

1.3.1 R 型指令格式



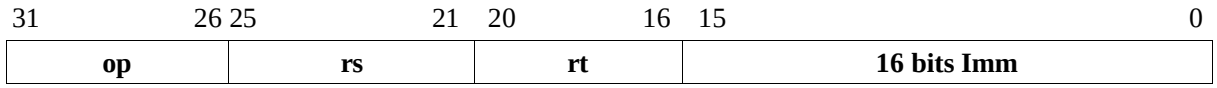
本实验需要实现的 R 型指令有：

（1）算术逻辑运算指令：ADD、ADDU、SUB、SUBU、AND、OR、NOR、XOR 、SLT、SLTU

（2）移位指令：SLLV、SRLV、SRAV、SLL、SRL、SRA

（3）寄存器跳转指令：JR

1.3.2 I 型指令格式



本实验需要实现的 I 型指令有：

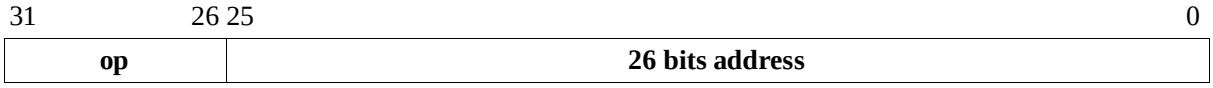
I)存储器访问指令：LW、SW

II)立即数算术逻辑运算指令：ADDI、ADDIU、ANDI、ORI、XORI、SLTI、SLTIU

III)分支指令：BEQ、BNE、BGEZ、BGTZ、BLEZ、BLTZ

分支地址为：PC+4+(sign-extend(Imm)<<2)

1.3.3 J 型指令格式



本实验需要实现的 J 型指令只有：无条件跳转指令：J

跳转地址为：{PC[31:28],IR[25:0],2’b00}

注意：

（1）寄存器跳转指令 JR 不是 J 型指令，而是 R 型指令。

跳转地址为：$ra

（2）移位指令 SLL、SRL、SRA 只有 rt 一个源操作数

（3）取字指令 LW 的操作过程：rt <= Mem[rs+sign_extend(Imm)]

存字指令 SW 的操作过程：Mem[rs+sign_extend(Imm)]<= rt

（4）I 型指令中立即数算术逻辑运算指令对立即数（Imm）的处理应分为两类情况考虑：

当指令为 ADDI、ADDIU、SLTI、SLTIU 时，指令中的 16 位立即数（Imm）应做符号扩展为 32 位。此符号扩展电路在 ID 级完成。

当指令为 ANDI、ORI、XORI 时，Imm 应做“0”扩展为 32 位。考虑到资源的限制，在执行 ANDI、ORI、XORI 指令时， “0”扩展功能放在 ALU 内部（即 EX 级）完成。

2.流水线 MIPS 微处理器的设计

2.1 IF 级的设计

2.1.1 指令存储器 ROM

用 Xilinx CORE Generator 实现产生的 ROM 无法满足流水线 CPU 的指令要求，我们需用 Verilog HDL 设计一个 ROM 阵列。考虑到 FPGA 的资源，指令存储器可设计为容量各为 26×32bit 的 ROM。设计 ROM 时需将测试的机器码写入，我们提供一段简单测试程序的机器码，机器码存于 PipelineDemo.coe 文件中。

我们提供该 ROM 的代码，已设计好上述测试程序机器码的指令存储器，文件名为 InstructionROM.v。

需要注意的是，ROM 的地址信号同 RAM 一样有“对齐限制”的要求，因此， ROM 的地址应该接的信

号是 PC[7:2]。

2.1.2 指令指针选择器

该 8 选 1 数据选择器的选择信号 PCSource={JR,J,Z}，具体的含义如下表：

地址	PC 来源
{JR,J,Z} = 000	下一条指令地址 PC+4
{JR,J,Z} = 001	Branch 指令的分支地址
{JR,J,Z} = 010	J 指令的跳转地址
{JR,J,Z} = 100	JR 指令的跳转地址

2.1.3 PC 寄存器

当发生数据冒险时，需要保持 PC 寄存器不变，因此 PC 寄存器是一个带使能端的 D 型寄存器，使能信号为 PC_IFWrite。

2.2 ID 级的设计

2.2.1 控制单元的设计

控制单元的主要任务是，根据指令存储器读出的指令，确定各个控制信号的值。它是一个组合电路。

我们将本实验需要实现的指令分成八类来讨论：

这三类指令均为 R 型指令：

R_type1：包括指令 ADD、ADDU、SUB、SUBU、AND、OR、NOR、XOR 、SLT、SLTU、SLLV、SRLV、SRAV。

R_type2：包括指令 SLL、SRL、SRA。

JR_type：包括指令 JR。

这四类指令均为 I 型指令：

I_type：包括指令 ADDI、ADDIU、ANDI、ORI、XORI、SLTI、SLTIU。

Branch：包括指令 BEQ、BNE、BGEZ、BGTZ、BLEZ、BLTZ。

LW：包括指令 LW。

SW：包括指令 SW。

另一类：

J_type：包括指令 J

9 组控制信号：

（1）RegWrite

决定是否对寄存器(registers)进行写操作。

RegWrite 有效时，将数据写入指定的寄存器中。

需要写回寄存器的指令类型有：LW、R_type1、

R_type2 和 I_type，所以：

RegWrite_id= LW || R_type1 || R_type2 || I_type

（2）RegDst

决定目标寄存器是 rt 还是 rd。

RegDst=0 时，rt 为目标寄存器。

RegDst=1 时，rd 为目标寄存器。

需要写回寄存器的指令类型有：LW、R_type1、R_type2 和 I_type。其中， R_type1 和 R_type2 的目标寄存器是 rd，而 LW 和 I_type 的目标寄存器是 rt。所以：

RegDst_id= R_type1 || R_type2

（3）MemWrite

决定是否对数据存储器进行写操作。

MemWrite 有效时，将数据写入数据存储器指定的位置。

需要对写存储器的指令只有 SW，所以：

MemWrite_id= SW

（4）MemRead

决定是否对数据存储器进行读操作。

MemRead 有效时，读取数据存储器指定位置的数据。

需要对读存储器的指令只有 LW，所以：

MemRead_id= LW

（5）MemtoReg

决定写入寄存器(registers)的数据来自 ALU 还是数据存储器。

MemtoReg=0 时，数据来自 ALU。

MemtoReg=1 时，数据来自数据存储器。

需要写回寄存器的指令类型有：LW、R_type1、R_type2 和 I_type，其中，只有 LW 写回寄存器的数据取自存储器，所以：

MemtoReg_id= LW

（6）ALUSrcA

决定 ALU 第一操作数来源。

ALUSrcA=0 时，ALU 第一操作数 A。（具体来源取决于转发电路的输出控制信号 ForwardA，详见转发电路的设计）

ALUSrcA=1 时，ALU 第一操作数来源于 0 扩展的 5 位 sa。

J_tpye、JR_tpye 及 Branch 类型指令没有使用 ALU。

使用 ALU 的指令类型中 LW、SW、R_type1 和 I_type 均采用的 rs 作为 ALU 第一操作数，只有 R_type2 的第一操作数采用的是 0 扩展的 sa 段，所以：

ALUSrcA_id= R_type2

（7）ALUSrcB

决定 ALU 第二操作数来源。

ALUSrcB=0 时， ALU 第二操作数 B。（具体来源取决于转发电路的输出控制信号 ForwardB，详见转发电路的设计）

ALUSrcB=1 时， ALU 第二操作数来源于符号扩展的 16 位 Imm。

J_tpye、JR_tpye 及 Branch 类型指令没有使用 ALU。

使用 ALU 的指令类型中 R_type1 和 R_type2 采用 rt 作为 ALU 第二操作数，而 LW、SW、I_type 的第二操作数采用的是符号扩展的立即数 Imm 段，所以：

ALUSrcB_id= LW || SW || I_type

即 IR[15:0]，用于 I 型指令

（8）ALUCode（5 位信号）

决定 ALU 的操作。

由指令中的 op 段、rt 段及 funct 段决定。

具体功能表详见教材中表 7.16(P222)。

（9）PCSource（3 位信号）

决定写入 PC 寄存器的来源。

PCSource 由 JR_tpye 、J_tpye 及 Z 决定：即：

PCSource={JR, J, Z}

PCSource=000 时， 写入值为下一条指令的地址 PC+4。

PCSource=001 时， 写入值为 Branch 指令的分支地址。

PCSource=010 时，写入值为 J 指令的跳转地址。

PCSource=100 时，写入值为 JR 指令的跳转地址。

以上前 7 组控制信号均为 1 位信号，而最后 2 组信号为多位信号。

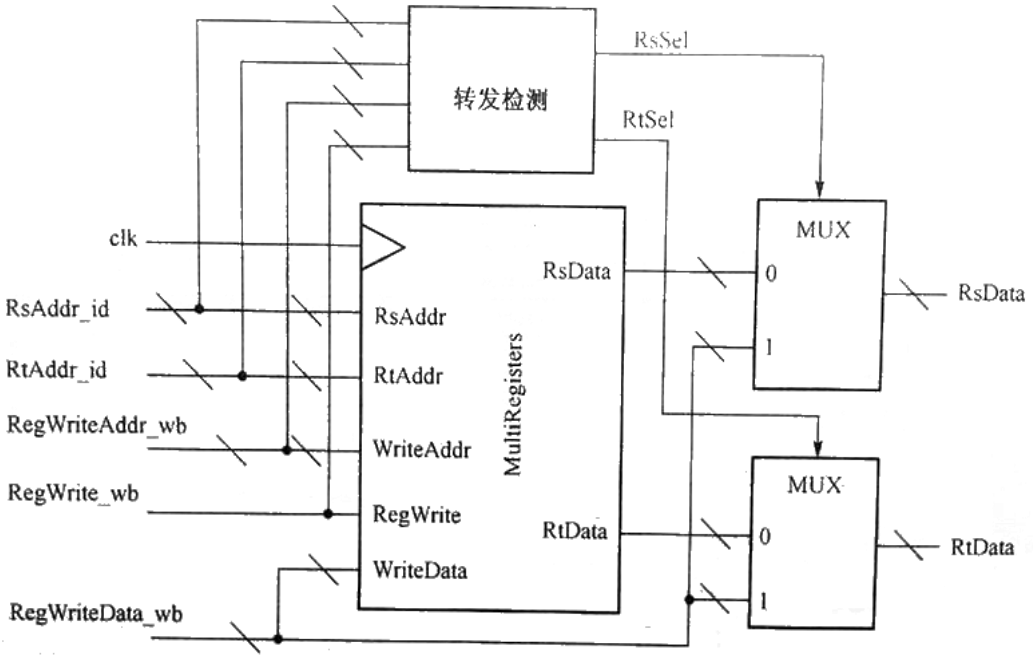
2.2.2 Zero 检测电路的设计

Zero 检测电路主要用于判断 Branch 指令的分支条件是否成立，其中 BEQ、BNE 两个操作数为 RsData 与 RtData，而 BGEZ、BGTZ、BLEZ 和 BLTZ 指令则为 RsData 与常数 0 比较，所以输出信号 Z 的表达式为：

```
alu_beq:    Z=&(RsData[31:0]~^RtData[31:0]);
alu_bne:    Z=|(RsData[31:0]^RtData[31:0]);
alu_bgez:   Z=~RsData[31];
alu_bgtz:   Z=~RsData[31]&&(|RsData[31:0]);
alu_bltz:   Z=RsData[31];
alu_blez:   Z=RsData[31]||~(|RsData[31:0]);
```

2.2.3 寄存器堆的设计

流水线型 CPU 中的寄存器堆要求具有“先写后读（Read After Write）”特性。为实现该功能，可在实验 27 中的寄存器堆的基础上增加一转发电路。



2.2.4 冒险检测电路的设计

冒险成立的条件为：

①上一条指令是 lw 指令，：

MemRead_ex =1

②在 EX 级的 lw 指令与在 ID 级的指令读写的是同一个寄存器，即：

RegWriteAddr_ex = RsAddr_id 或 RegWriteAddr_ex = RtAddr_id

解决冒险的方法为：

①插入一个流水线气泡：

stall = MemRead_ex &&((RegWriteAddr_ex == RsAddr_id)||(RegWriteAddr_ex == RtAddr_id))

②保持 PC 寄存器和 IF/ID 流水线寄存器不变：

PC_IFWrite = ~stall

2.2.5 其它单元电路的设计

1、Branch 指令分支地址的计算电路：

BranchAddr= NextPC_id+(sign-extend(Imm_id)<<2)

2、JR 指令跳转地址的计算电路：

JRAddr = RsData_id

3、J 指令跳转地址的计算电路：

JAddr = {NextPC_id[31:28],IR_id[25:0],2'b00}

4、符号扩展的方法— 针对有符号数

如果最高位（即符号位）是 0，则要扩展的高位用 0 补齐；最高位是 1 则用 1 补齐。

例： 8 位的+1，表示为二进制为 00000001，扩展成 16 位的话，符号扩展为 0000000000000001；8 位的-1，表示成二进制为 11111111，扩展成 16 位的话，符号扩展为 1111111111111111 。

5、0 扩展的方法— 针对无符号数

要扩展的高位用 0 补齐。

例：16 位二进制 0xFFFF(无符号数 65535), 0 扩展成 32 位为 0x0000FFFF (无符号数 65535) 。

2.3 EX 级的设计

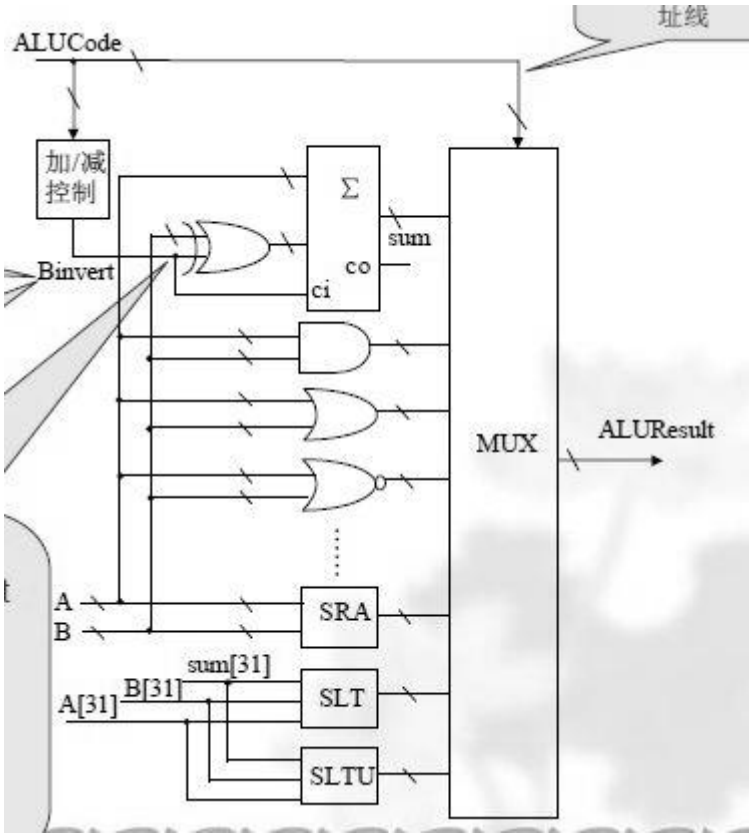
2.3.1 ALU 的设计

ALU 是提供 CPU 基本运算能力的重要电路。

ALU 具体执行何种运算，由控制单元中的 ALU 控制器输出的 ALUCode 信号决定。如下表所示：

ALUCode	ALUResult
00000（alu_add）	A + B
00001（alu_and）	A& B
00010（alu_xor）	A ^ B
00011（alu_or）	A | B
00100（alu_nor）	~(A|B)
00101（alu_sub）	A-B
00110（alu_ andi）	A& {16'd0，B[15:0]}
00111（alu_ xori）	A ^{16'd0，B[15:0]}
01000（alu_ ori）	A | {16'd0，B[15:0]}
10000（alu_sll）	B<<A
10001（alu_srl）	B>>A
10010（alu_sra）	B>>>A
10011（alu_slt）	A<B? 1:0，其中 A、B 为有符号数
10100（alu_sltu）	A<B? 1:0，其中 A、B 为无符号数

为了提高运算速度，可将各种运算同时执行，得到的运算结果由 ALUCode 信号进行挑选。



Binvert=~(ALUCode==alu_add)

Binvert 为 1 位信号，当它需要与操作数 B 按位异或时，应将 Binvert 复制为 32 位的信号，具体写法为：{32{Binvert}}

加法器完成的功能为：sum=A+B^Binvert+Binvert

当 Binvert=0 时，sum=A+B^0+0=A+B 即完成加法运算。

当 Binvert=1 时，sum=A+B^1+1=A-B 即完成减法运算。

5 位 ALUCode 作为 MUX 的地址线

关于运算电路的几点说明：

（1）减法的实现

对 ALU 来说，它的两个操作数输入时都已经是补码形式，当要完成两个操作数的减法时，即 A 补-B 补，可将减法转换为加法，利用加法器来实现：

$$A_{补}-B_{补}=A_{补}+(-B_{补})=A_{补}+(B_{补})_{补}=A_{补}+(B_{补})_{反}+1$$

（2）比较电路中要注意有符号数和无符号数比较运算的区别。在有符号数比较 SLT 运算中，判断 A<B 的方法有：

方法一，利用减法

对有符号数 A、B，满足下列两种情况之一即可判断 A<B：

A 为负数、B 为 0 或正数：A[31]&&(~B[31])

A、B 符号相同，A-B 为负：(A[31]~^B[31]) && sum[31]

因此：

SLTResult=(A[31]&&(~B[31])) ||((A[31]~^B[31]) && sum[31])

方法二，利用比较运算符

在使用比较运算符时，要注意它的两个操作数输入时都已经是补码形式，但比较运算符把所有定义为 wire 或 reg 型的数据都当做无符号数看待，因此，直接利用比较运算符对 A、B 判断大小是不妥的。要正

确使用比较运算符的话，必须要对运算后的结果再做进一步判断，具体方法如下：

若 A[31]与 B[31]不相等，即两数为一正一负时：这时比较运算得到的结果与实际结果是相反的。例如，比较运算得到的结果为 A>B，说明 A 为负数 B 为正数，则实际结果应该为 A<B 成立。反之亦然。

若 A[31]与 B[31]相等，即两数同为正数或同为负数时：这时比较运算得到的结果与实际结果是一致的。

2.3.2 实现操作数 A 和 B 的 MUX 的设计

操作数 A 和 B 由数据选择器决定，数据选择器的地址信号即为 ForwardA 和 ForwardB。

数据选择器地址信号 ForwardA、ForwardB 的含义如下表：

地址	操作数来源	说明
ForwardA= 00	RsData_ex	操作数 A 来自寄存器堆
ForwardA= 01	RegWriteData_wb	操作数 A 来自二阶数据相关的转发数据
ForwardA= 10	ALUresult_mem	操作数 A 来自一阶数据相关的转发数据
ForwardB= 00	RtData_ex	操作数 B 来自寄存器堆
ForwardB= 01	RegWriteData_wb	操作数 B 来自二阶数据相关的转发数据
ForwardB= 10	ALUresult_mem	操作数 B 来自一阶数据相关的转发数据

2.3.3 转发电路（Forwarding）的设计

转发电路主要用于检测一阶和二阶数据相关冒险。该电路产生两个信号 ForwardA 和 ForwardB。

由前面介绍的一阶和二阶数据相关的判断条件，再结合上页中的表格，可得：

```
ForwardA[0]=RegWrite_wb&&
            (RegWriteAddr_wb!=0)&&
            (RegWriteAddr_mem!=RsAddr_ex)&&
            (RegWriteAddr_wb==RsAddr_ex);
ForwardA[1]=RegWrite_mem&&
            (RegWriteAddr_mem!=0)&&
            (RegWriteAddr_mem==RsAddr_ex);

ForwardB[0]=RegWrite_wb&&
            (RegWriteAddr_wb!=0)&&
            (RegWriteAddr_mem!=RtAddr_ex)&&
            (RegWriteAddr_wb==RtAddr_ex);
ForwardB[1]=RegWrite_mem&&
            (RegWriteAddr_mem!=0)&&
            (RegWriteAddr_mem==RtAddr_ex);
```

2.4 MEM 级的设计

数据存储器利用 Xilinx Core Generator 实现。考虑到 FPGA 的资源，数据存储器可设计为容量各为 26×32bit 单端口 RAM。

由于 MIPS 系统的 32 位字地址由 4 个字节组成，根据“对齐限制”要求字地址必须是 4 的倍数，也就是说字地址的低两位必须是 0，所以字地址的低两位不接入电路。故实验中设计的数据 RAM 的地址应该接的信号是 ALUResult_mem[7:2]。

由于 VIRTEXIIPRO 只能产生带寄存器的内核 RAM，所以存储器输出绕过 MEM/WB 流水线寄存器，直接接入 WB 级的数据选择器。

2.5 WB 级的设计

WB 级的设计比较简单，只涉及到寄存器写回，所以只需要简单的电路就可以实现，给出寄存器写信号就可以采用以下方式：

```
RegWriteData_wb=MemToReg_wb?MemDout_wb:ALUResult_wb;
```

2.6 流水线寄存器的设计

流水线寄存器负责将流水线的各部分分开，共有 IF/ID、ID/EX、EX/MEM、MEM/WB 四组。

根据前面的介绍可知，四组流水线寄存器要求不完全相同，因此设计也有不同考虑。

（1）EX/MEM、MEM/WB 两组流水线寄存器只是普通 D 型寄存器。

（2）当流水线发生数据冒险时，需清空 ID/EX 流水线寄存器而插入一个气泡，因此 ID/EX 流水线寄存器是一个带同步清零功能的 D 型寄存器，清零信号为 stall。

（3）当流水线发生数据冒险时，需保持 IF/ID 流水线寄存器不变，因此 IF/ID 流水线寄存器具有使能信号输入，使能信号为 PC_IFWrite；当流水线发生分支冒险时，需清空 IF/ID 流水线寄存器，清零信号为 IF_flush。因此，IF/ID 流水线寄存器是一个带使能功能、同步清零功能的 D 型寄存器。

需要注意的是，由于仿真对初始值的要求，上述寄存器都应考虑有 reset 信号的接入，以提供仿真时各寄存器的初值。

2.7 顶层文件设计

按照 MIPS 处理器原理框图链接各模块即可。为测试方便，可将关键变量输出，关键变量有：指令指针 PC、指令码 Instruction_id、流水线插入气泡标志 stall、分支标志 JumpFlag 即{J, JR, Z}、ALU 输入输出（ALU_A, ALU_B, ALUResult）和数据存储器的输出 MemDout_wb。

3.说明

实验中，已经提供了流水线未处理的基本架构。在提供的代码中基本已经做了端口说明和参数定义等，代码的关键部分需要编写。

部分模块已经提供测试代码，可以根据测试代码进行模块的仿真测试以验证模块编写的正确性。

提供的工程文件中已经提供了引脚分配和时序约束，不需要另外进行。ISE 工程文件中还提供了一套 SVGA 显示系统，可以将最终工程文件下载到开发板中，并在显示器上显示出来以验证实验最终结果的正确性。

文件管理说明：

本实验中由于涉及到代码编写、仿真、ISE 综合实现等步骤，文件的管理非常重要，故本项目中主要文件结构如下：

<table>
<tr><td rowspan="5">lab28</td><td rowspan="3">PipelineCPU</td><td>ISE</td><td>存放 Xilinx 工程文件</td></tr>
<tr><td>SIM</td><td>存放 ModelSim SE 仿真文件</td></tr>
<tr><td>SRC</td><td>存放源代码</td></tr>
<tr><td>PipelineCPU_VGA</td><td></td><td>存放用于 SVGA 显示的工程文件</td></tr>
<tr><td>*Recycle</td><td></td><td>存放实验过程中删除的可能没用的文件，可用于错误记录比较或文件备份。但在上交的 Soulution 中已经删除</td></tr>
</table>

四、实验设备

（1）装有 ISE、ModelSim SE 和 ChipScope Pro 软件的计算机。

（2）XUP Virtex-Ⅱ Pro 开发系统一套

（3）SVGA 显示器一台

五、实验内容

1. 完成代码设计及仿真测试

实验中已经提供了 IF、ID、EX 和顶层文件以及部分子模块的部分代码，主要做了端口的说明和参数的定义。

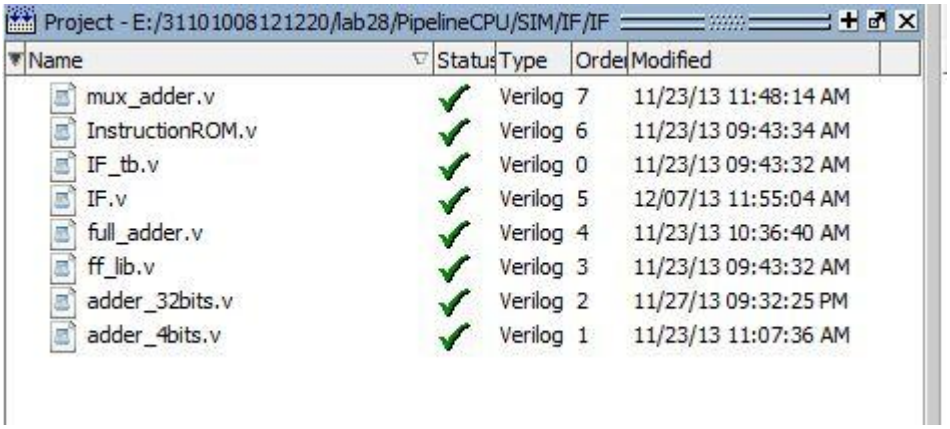
根据实验已经提供的部分代码继续进行代码的编写与设计。本次实验，按照 IF 级、ID 级、EX 级、顶层文件的顺序编写以及其子模块 Verilog HDL 代码的编写。

由于篇幅过长，在此不再一一详细展示。只作简单的说明，具体详细代码及注释参见 Solution 文件。

编写文件与 word 格式不统一，在报告中可能出现部分的格式混乱，还请谅解。

1.1 IF 级代码及仿真

根据实验原理编写 IF 级 Verilog HDL 代码，在 ModelSim 中可以看到 IF 具体涉及到的文件如下图所示，各文件的详细代码参见 Solution 中的文件。



IF 级功能比较容易实现，其顶层代码如下：

```
module IF(clk, reset, Z, J, JR, PC_IFWrite, JumpAddr,
            JrAddr, BranchAddr, Instruction_if,PC, NextPC_if);
    input clk;
    input reset;
    input Z;
    input J;
    input JR;
    input PC_IFWrite;
    input [31:0] JumpAddr;
    input [31:0] JrAddr;
    input [31:0] BranchAddr;
    output [31:0] Instruction_if;
    output [31:0] PC,NextPC_if;
// MUX for PC
    reg[31:0] PC_in;

    always @(*)
        begin
            case({JR,J,Z})
                3'b000:   PC_in=NextPC_if;                  //next instruction PC+4
                3'b001:   PC_in=BranchAddr;                 //Branch
                3'b010:   PC_in=JumpAddr;                   //Jump
                3'b100:   PC_in=JrAddr;                     //JR
                endcase
            end
```

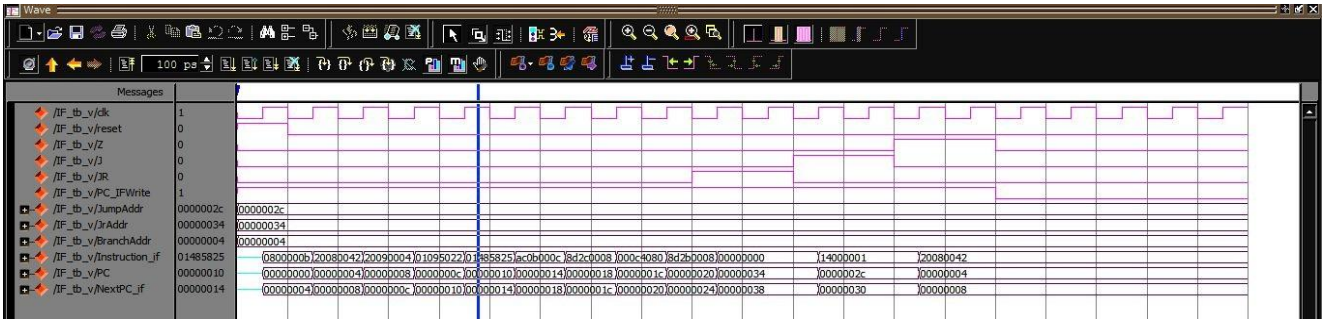
```
//PC REG
        dffre #(32) dffre_PC_REG(
        .d(PC_in),
        .en(PC_IFWrite),
        .r(reset),
        .clk(clk),
        .q(PC));

//Adder for NextPC
        adder_32bits adder_32bits_NextPC(
        .a(PC),
        .b(32'h00000004),
        .ci(0),
        .s(NextPC_if),
        .co());

//ROM
        InstructionROM    InstructionROM(
        .addr(PC[7:2]),
        .dout(Instruction_if));

endmodule
```

经调试正确，在 ModelSim 中对 IF 级进行仿真测试，得到仿真结果如下：



分析：从仿真图中可以判断，IF 级代码编写正确，并已实现预期功能，图中数值均以 16 进制表示。以 JR 信号为例，当它出现 1 时，{JR,J,Z} = 100，PC 的数据来源为 JrAddr(00000034H)，PC_IFWrite 信号为高电平，PC 的值变为 00000034H，说明数据选择正确；当{JR,J,Z} = 000 时，PC 信号与 NextPC_if 信号在时间上传递；PC_IFWrite=0 时，Instruction_if、PC、NextPC_if 信号保持不变，实现了其阻塞的功能。其它情况下采用类似的分析均可验证 IF 级功能的正确性。

1.2 ID 级代码及仿真

根据实验原理编写 ID 级 Verilog HDL 代码。

由于篇幅过长，不全部展示。具体参加 Solution 文件。

其中部分关键模块代码如下：

```
//JumpAddress

    assign JumpAddr={NextPC_id[31:28],Instruction_id[25:0],2'B00};
```

```
//BranchAddrress

        adder_32bits adder_32bits_BranchAddr(
        .a(NextPC_id),
        .b({Imm_id[29:0],2'b00}),
        .ci(0),
        .s(BranchAddr),
        .co()
        );
//JrAddress

        assign JrAddr=RsData_id;

//Zero test

        Zero_test zero_test_inst(
        .ALUCode(ALUCode_id),.RsData(RsData_id),.RtData(RtData_id),.Z(Z));

//Hazard detector
          parameter       alu_beq=    5'b01010;
      parameter    alu_bne=    5'b01011;
          parameter       alu_bgez= 5'b01100;
      parameter    alu_bgtz= 5'b01101;
      parameter    alu_blez= 5'b01110;
      parameter    alu_bltz= 5'b01111;

        Hazard_detector Hazard_detector_inst(
        .MemRead_ex(MemRead_ex),
        .RegWriteAddr_ex(RegWriteAddr_ex),
        .RsAddr_id(RsAddr_id),
        .RtAddr_id(RtAddr_id),
        .stall(Stall),
        .PC_IFWrite(PC_IFWrite)
        );



//      Decode inst
      Decode    Decode(
                // Outputs
                .MemtoReg(MemtoReg_id),
                .RegWrite(RegWrite_id),
                .MemWrite(MemWrite_id),
```

```
            .MemRead(MemRead_id),
            .ALUCode(ALUCode_id),
            .ALUSrcA(ALUSrcA_id),
            .ALUSrcB(ALUSrcB_id),
            .RegDst(RegDst_id),
            .J(J) ,
            .JR(JR),
            // Inputs
          .Instruction(Instruction_id)
        );

// Registers inst

    //MultiRegisters inst
    wire [31:0] RsData_temp,RtData_temp;

     Registers     MultiRegisters(
     // Outputs
     .RsData(RsData_temp),
     .RtData(RtData_temp),
     // Inputs
     .clk(clk),
     .WriteData(RegWriteData_wb),
     .WriteAddr(RegWriteAddr_wb),
     .RegWrite(RegWrite_wb),
     .RsAddr(RsAddr_id),
     .RtAddr(RtAddr_id)
     );


     //RsSel & RtSel
     wire    RsSel,RtSel;

     RsSel_RtSel RsSel_RtSel_inst(
     .RegWrite_wb(RegWrite_wb),
     .RegWriteAddr_wb(RegWriteAddr_wb),
     .RsAddr_id(RsAddr_id),
     .RtAddr_id(RtAddr_id),
     .RsSel(RsSel),
     .RtSel(RtSel)
     );


   //MUX for RsData_id    &    MUX for RtData_id
```

```
assign RsData_id=(RsSel==0)?(RsData_temp):(RegWriteData_wb);
assign RtData_id=(RtSel==0)?(RtData_temp):(RegWriteData_wb);
```

Decode 模块关键部分如下所示：

```
//ALUCode----------------------parameter that will be used must be declared before using.
        always @(*)
            begin
                case(op)
                    R_type_op:    begin
                                        if(ADD)
                                        ALUCode=    alu_add;
                                        else if(ADDU)
                                        ALUCode=    alu_add;
                                        else if(AND)
                                        ALUCode=    alu_and;
                                        else if(XOR)
                                        ALUCode=    alu_xor;
                                        else if(OR)
                                        ALUCode=    alu_or;
                                        else if(NOR)
                                        ALUCode=    alu_nor;
                                        else if(SUB)
                                        ALUCode=    alu_sub;
                                        else if(SUBU)
                                        ALUCode=    alu_sub;
                                        else if(SLL)
                                        ALUCode=    alu_sll;
                                        else if(SLLV)
                                        ALUCode=    alu_sll;
                                        else if(SRA)
                                        ALUCode=    alu_sra;
                                        else if(SRA)
                                        ALUCode=    alu_sra;
                                        else if(SRL)
                                        ALUCode=    alu_srl;
                                        else if(SRLV)
                                        ALUCode=    alu_srl;
                                        else if(SLT)
                                        ALUCode=    alu_slt;
                                        else if(SLTU)
                                        ALUCode=    alu_sltu;
                                        else if(JR)
                                        ALUCode=    alu_jr;
```

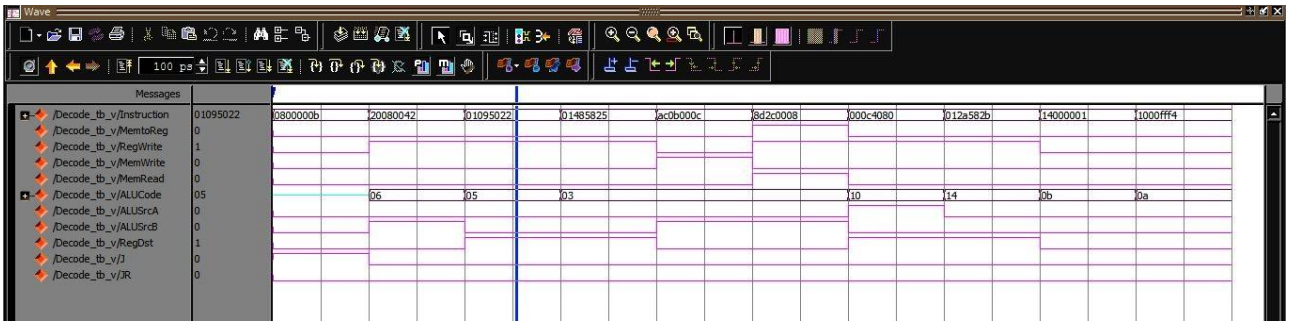
```
                else  ALUCode=0;
                end

        BEQ_op:         ALUCode=    alu_beq;        //Brunch
        BNE_op:         ALUCode=    alu_bne;
        BGEZ_op:        ALUCode=    alu_bgez;
        BGTZ_op:        ALUCode=    alu_bgtz;
        BLEZ_op:        ALUCode=    alu_blez;
        BLTZ_op:        ALUCode=    alu_bltz;

        ADDI:           ALUCode=    alu_add;
        ADDIU:          ALUCode=    alu_add;
        ANDI_op:        ALUCode=    alu_andi;
        XORI_op:        ALUCode=    alu_xori;
        ORI_op:         ALUCode=    alu_ori;
        SLTI:           ALUCode=    alu_slt;
        SLTIU:          ALUCode=    alu_sltu;
        SW:             ALUCode=    alu_add;
        LW:             ALUCode=    alu_add;

        default:    ALUCode=0;
    endcase
end
```

经调试正确，对 Decode 模块在 ModelSim 中进行仿真测试，得到仿真结果如下：



分析：根据仿真图可验证 Decode 模块的正确性，图中信号均用 16 进制表示。例如：图中蓝线位置 Instruction 为 01095022，即 sub 指令，根据指令操作，可知经 Decode 之后其它信号均得到了正确的值，同理可验证其它情况的正确性。

Decode 模块编码正确。

由于测试的关键部分在于 Decode 模块，ID 级不再单独作仿真测试。

1.3EX 级代码及仿真

根据实验原理编写 EX 级代码。EX 级涉及的主要模块包括 5 个 MUX、1 个 ALU、1 个 Fowarding 机构。

EX 级顶层代码关键部分如下：

```
//forwarding
    wire[1:0] ForwardA,ForwardB;
```

```
        assign ForwardA[0]=RegWrite_wb&&
                        (RegWriteAddr_wb!=0)&&
                        (RegWriteAddr_mem!=RsAddr_ex)&&
                        (RegWriteAddr_wb==RsAddr_ex);
        assign ForwardA[1]=RegWrite_mem&&
                        (RegWriteAddr_mem!=0)&&
                        (RegWriteAddr_mem==RsAddr_ex);

        assign ForwardB[0]=RegWrite_wb&&
                        (RegWriteAddr_wb!=0)&&
                        (RegWriteAddr_mem!=RtAddr_ex)&&
                        (RegWriteAddr_wb==RtAddr_ex);
        assign ForwardB[1]=RegWrite_mem&&
                        (RegWriteAddr_mem!=0)&&
                        (RegWriteAddr_mem==RtAddr_ex);

    //MUX for A

        wire [31:0] A,B;
        assign
A=(ForwardA[1]==0)?((ForwardA[0]==0)?(RsData_ex):(RegWriteData_wb)):(ALUResult_mem);

    //MUX for B

        assign
B=(ForwardB[1]==0)?((ForwardB[0]==0)?(RtData_ex):(RegWriteData_wb)):(ALUResult_mem);

    //MUX for ALU_A

        assign ALU_A=(ALUSrcA_ex==1)?(Sa_ex):(A);
        assign MemWriteData_ex=B;

    //MUX for ALU_B

        assign ALU_B=(ALUSrcB_ex==1)?(Imm_ex):(B);

    //ALU inst
          ALU    ALU (
         // Outputs
        .Result(ALUResult_ex),
        .overflow(),
        // Inputs
        .ALUCode(ALUCode_ex),
        .A(ALU_A),
```

```
        .B(ALU_B)
);

//MUX for RegWriteAddr_ex

        assign RegWriteAddr_ex=(RegDst_ex==1)?(RdAddr_ex):(RtAddr_ex);
```

其中 ALU 模块关键部分代码如下：

```
//*************************************************************************************
// ALU Result datapath
//*************************************************************************************

        wire [31:0] sum;
        wire co;
        reg Binvert;
                                                        //Judge add or sub
        reg [31:0] B_adder;
                                                //B in adder

//Adder//
        always @(*)
            begin
                Binvert=~(ALUCode==alu_add);
                if(Binvert)
                    B_adder=~B+1;
                else
                    B_adder=B;
            end

        adder_32bits adder_32bits_ALU(
        .a(A),.b(B_adder),.ci(0),.s(sum),.co(co));

//Result//
  always @(*)
        begin
            case(ALUCode)
                alu_add:    Result=sum;
                alu_and:    Result=A&B;
                alu_xor:    Result=A^B;
                alu_or:     Result=A|B;
                alu_nor:    Result=~(A|B);
                alu_sub:    Result=sum;
                alu_andi:   Result=A&{16'b0,B[15:0]};
                alu_xori:   Result=A^{16'b0,B[15:0]};
```

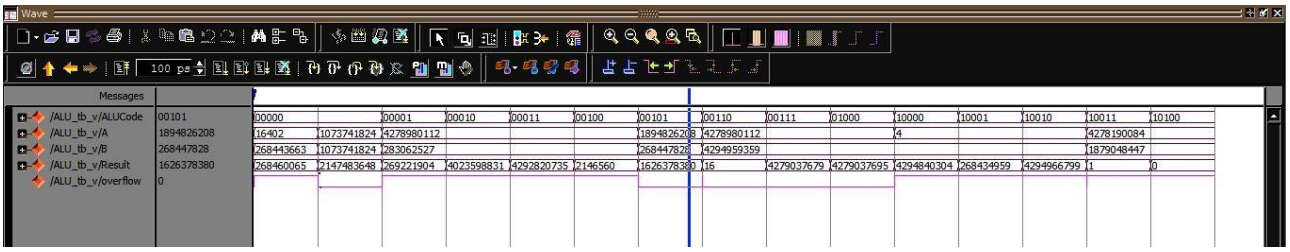
```
        alu_ori:    Result=A|{16'b0,B[15:0]};
        alu_jr:     Result=A;
        alu_sll:    Result=B<<A;
        alu_srl:    Result=B>>A;
        alu_sra:    Result=B_reg>>>A;
        alu_slt:    Result=(A[31]&&(~B[31]))||((A[31]~^B[31])&&sum[31]);
        alu_sltu:   Result=((~A[31])&&B[31])||((A[31]~^B[31])&&sum[31]);
        default:    Result=32'b0;
    endcase

    //judge overflow
    if(co==sum[31])
        overflow=1'b1;
    else
        overflow=1'b0;

end
```

其余部分代码参见 Solution 文件。

经调试正确，在 ModelSim 中对 ALU 模块进行仿真测试，得到仿真结果如下：



分析：根据实验原理，可以验证 ALU 单元的正确性，图中，ALUCode 用 2 进制表示，其余均用 10 进制表示。在图中所示位置可知，ALUCode=00101B，对比实验原理中，可知当前操作为 sub 指令，A=1894826208D，B=268447828D，Result=1626378380，overflow=0，故操作正确。其它情况也采用类似的方法分析可知，仿真测试的结果表明 ALU 单元正确。

1.4 DataRAM 内核存储器生成

数据存储器利用 Xilinx Core Generator 实现。考虑到 FPGA 的资源，数据存储器可设计为容量各为 26×32bit 单端口 RAM。

由于 MIPS 系统的 32 位字地址由 4 个字节组成，根据“对齐限制”要求字地址必须是 4 的倍数，也就是说字地址的低两位必须是 0，所以字地址的低两位不接入电路。故实验中设计的数据 RAM 的地址应该接的信号是 ALUResult_mem[7:2]。

最终利用 Xilinx 生成 DataRAM 文件，并把它拷贝到 SRC 文件夹中。

1.5 顶层文件编写及仿真

根据实验原理进行顶层文件的代码编写。顶层文件中，主要涉及的就是 IF、ID、EX 与 MIPS 处理器其它部分的链接，以及 MEM、WB 的编写，和流水线寄存器的编写。

由于源代码过长，在此只截取关键部分，详细代码参见 Solution 文件。

因为 VIRTEXIIPRO 只能产生带寄存器的内核 RAM，所以存储器输出绕过 MEM/WB 流水线寄存器，直接接入 WB 级的数据选择器。

MEM、WB 的编写如下：

```
//MEM Module
```

```
DataRAM     DataRAM(
.addr(ALUResult_mem[7:2]),
.clk(clk),
.din(MemWriteData_mem),
.dout(MemDout_wb),
.we(MemWrite_mem));

//WB
  assign RegWriteData_wb=MemToReg_wb?MemDout_wb:ALUResult_wb;
```

流水线寄存器共有 IF/ID、ID/EX、EX/MEM、MEM/WB 四组。

四组流水线寄存器要求不完全相同：

（1）EX/MEM、MEM/WB 两组流水线寄存器为普通 D 型寄存器。

（2）ID/EX 流水线寄存器是一个带同步清零功能的 D 型寄存器，清零信号为 stall。

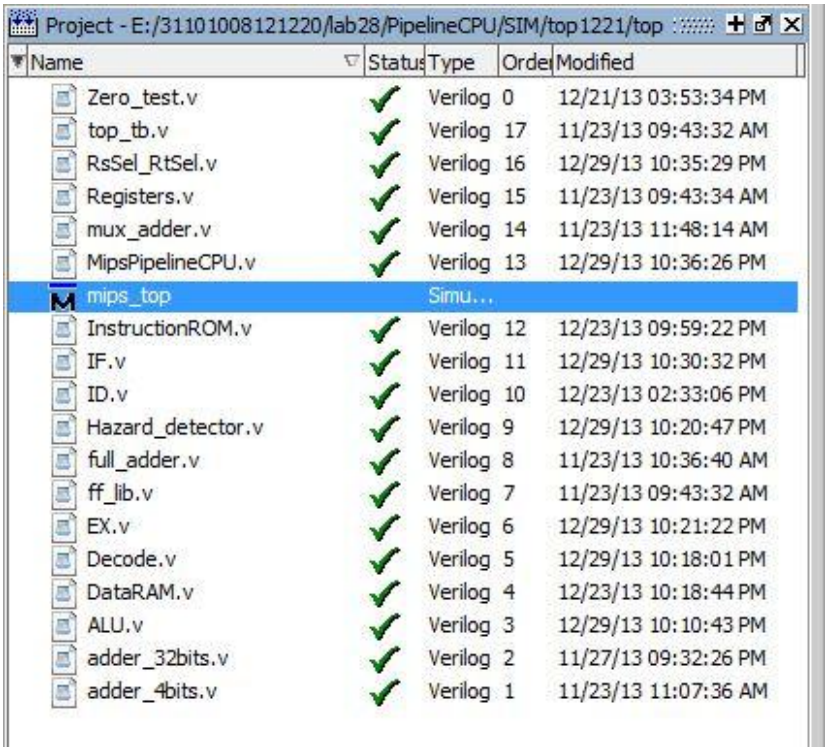
（3）IF/ID 流水线寄存器是一个带使能功能、同步清零功能的 D 型寄存器。使能信号为 PC_IFWrite，清零信号为 IF_flush。

由于对初始值的要求，上述寄存器都应考虑有 reset 信号的接入，以使整个 MIPS 微处理器在初始时刻有一个确定的状态。

流水线寄存器的代码不再列出，具体参见 Solution 文件。

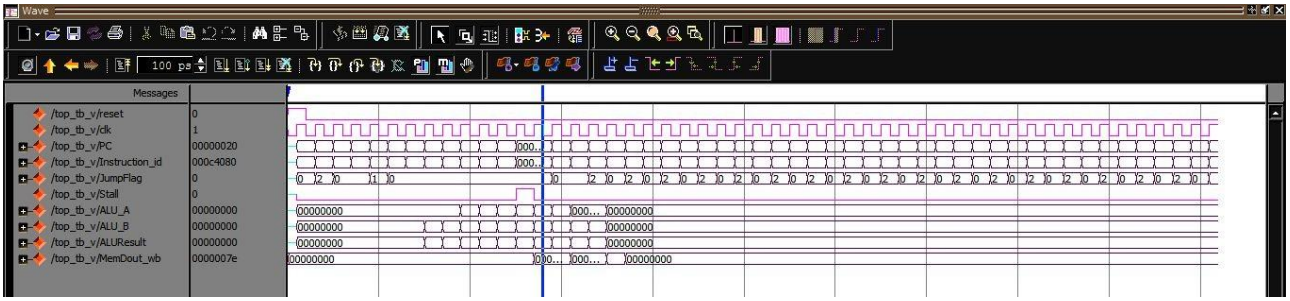
顶层文件涉及到 IP 核的仿真，所以事先做好 IP 核仿真设置，生成仿真所需的文件。IP 核仿真的具体步骤不再作过多阐述。

至此，整个工程文件基本已经生成，在 ModelSim 可以看到整个工程涉及到的源代码文件如下所示：



对流水线 MIPS 微处理器进行功能仿真，根据提供的测试程序的运行结果进行比较验证仿真结果的正确性。

在 ModelSim 中对 MIPS 流水线处理器进行功能仿真，得到仿真结果如下图所示：



测试程序对应的代码为：

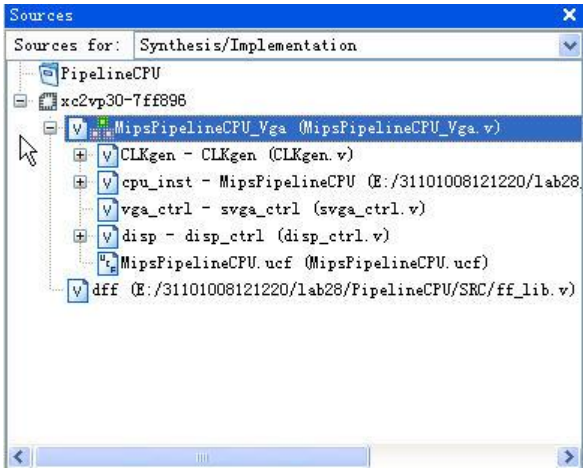
```
                j       later
earlier:    addi $t0, $0, 42
            addi $t1, $0, 4
            sub     $t2, $t0, $t1   //操作数 B 一阶数据相关，操作数 A 二阶数据相关
            or      $t3, $t2, $t0   //操作数 A 一阶数据相关，操作数 B 三阶数据相关
            sw      $t3, 0C($0)
            lw      $t4, 08($t1)
            sll     $t0, $t4, 2             //数据冒险
            lw      $t4, 08($t1)
            sltu  $t3, $t1, $t2
done:       j           done
later:      bne     $0, $0, end             //分支条件不成立
            beq     $0, $0,earlier      //分支条件成立
end:        nop
```

分析：根据提供的测试程序的运行结果与图中 reset、clk、PC、Instruction_id、JumpFlag、stall、ALU_A、ALU_B、ALUResult、MemDout_wb 逐一对比，验证其正确性，图中均为 16 进制表示。经对比可知，结果全部正确，如图中测量线所示位置 MemDout_wb 结果为 0000007eH，对应的其它信号也与其一致。对比测试程序也可以看到测试结果的正确。

整个工程文件实现了 MIPS 流水线的基本功能，达到了实验设计的要求。

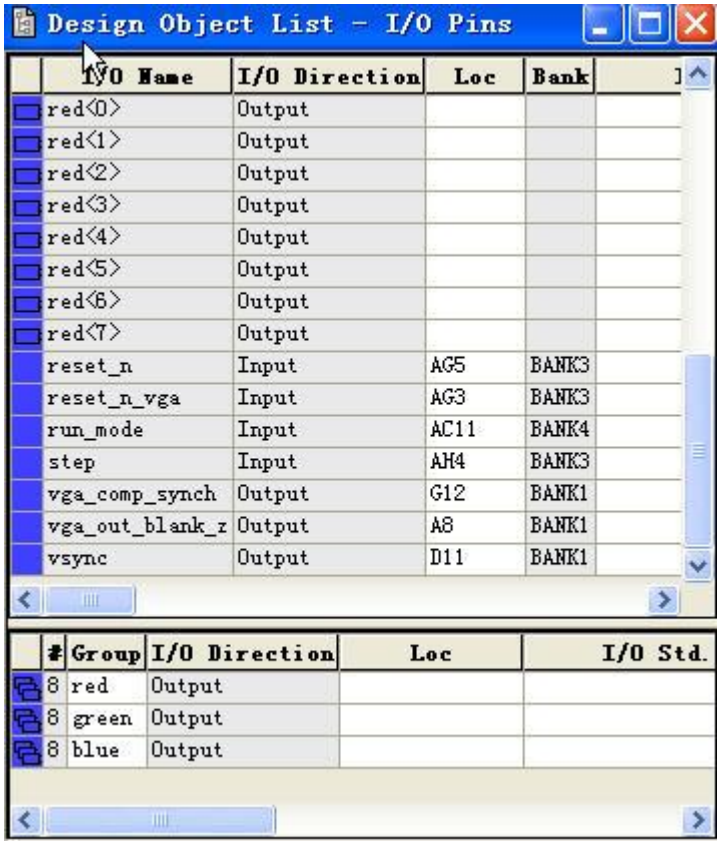
2. 在 ISE 工程中进行综合、约束、实现

打开 PipelineCPU_VGA 文件夹中的 ISE 工程文件，将各文件添加进去，尤其是 DataRAM 相关文件，由于实验已经提供.ucf 约束文件，引脚、时序约束都已经做好，只需按顺序将文件进行综合、约束、实现即可。



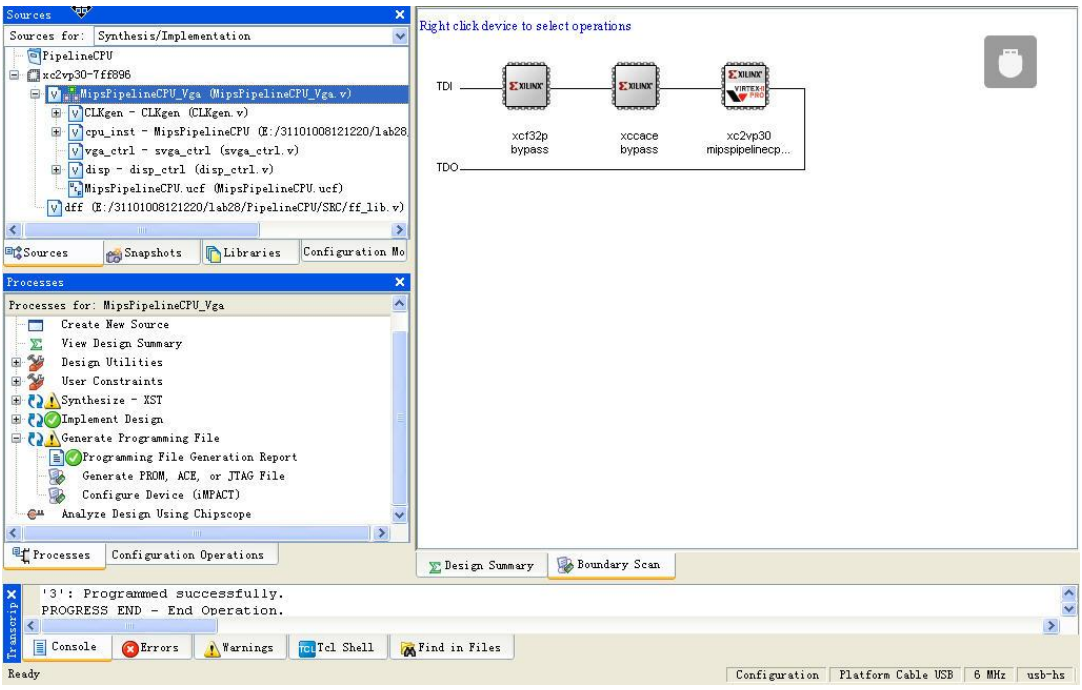
上图为 ISE 文件工程结构，可以看到 MIPS 微处理器模块、显示模块、时钟引脚约束等都已经添加进去。

其中引脚约束详细内容如下图所示：



主要输入按键有四个，MIPS 流水线微处理器 reset 按键，显示 reset 按键，单步执行按键，全速执行按键。

对工程文件进行综合实现，结果如下图所示：



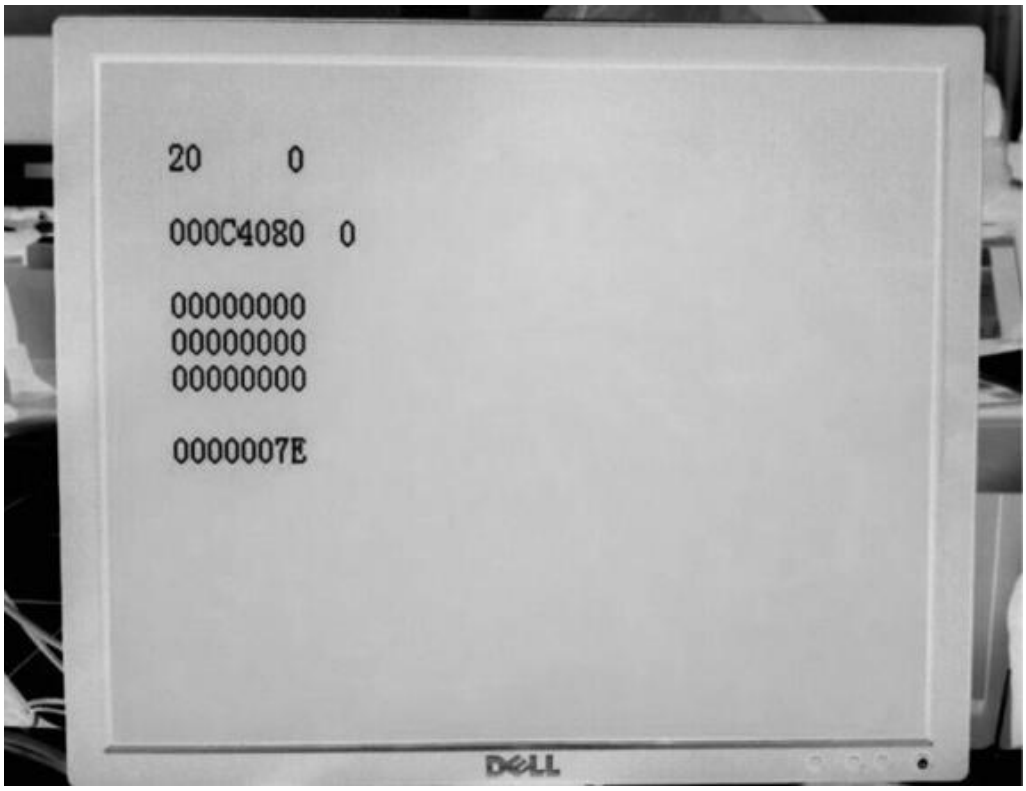
3. 工程的实际测试

（1）下载工程到 XUP Virtex-Ⅱ Pro 开发板中，接入 SVGA 显示器。

（2）将 SW0 置于 ON 位，使 MIPS CPU 工作在“单步”运行模式。按住 Enter 键，再按 Up 键，可以将 CPU 复位。

刚下载入开发板中的程序处于初始状态，每按一下 Up 按键，MIPS CPU 运行一步，记录下显示器上的结果，根据测试程序的运行结果验证设计是否正确。

现象：程序按照测试程序运行结果表中的顺序有序的执行，每按一下 Up 按键，MIPS CPU 运行一步，显示器上的结果对应于表中的数据变化。其中在第 14 个时钟下的状态如下图所示：



执行结果均正确。

（3)然后将 SW0 置于 OFF 位，使 MIPS CPU 工作在连续运行模式，测试 MIPS CPU 能否工作在 25MHz 时钟下正常工作。

实验现象：MIPS CPU 正常工作，屏幕中部分数字出现闪烁现象，因为程序执行到后面不停地在两条指令间跳转，稳定在两条指令的切换之中，这在仿真的时候也可以看得出来，上述现象说明 MIPS CPU 正常工作在 25MHz 时钟下。

（4）本次测试程序只测试了 10 条指令，编写测试其它指令的测试程序并编译为机器码，修改指令存储器 InstructionROM.v 文件。重复实验步骤，测试其他指令的运行。

至此，整个实验完成。

六、问题与解决方法

实验过程中遇到了很多的问题，经过仔细研究、相互探讨、老师的帮助，这些问题一一得到了解决。现将这些问题分类整理，并作简要的举例说明。

实验过程中遇到的问题主要有以下几类：

1. Verilog HDL 语法类。

由于对 Verilog HDL 还不够完全的熟悉，虽然经过了数字系统设计 1 的训练，但还是有些问题，在实

验中遇到了一些问题。如：

（1）接口的定义

对输出接口可以直接用 reg 或者 wire 类型直接定义接口的类型，但对于输入接口不能用 reg 语句进行定义，下列语句就是一个错误的示范：

```
module RsSel_RtSel(
    RegWrite_wb,RegWriteAddr_wb,RsAddr_id,RtAddr_id,RsSel,RtSel
    );

    input   reg RegWrite_wb;
    input   reg [4:0] RegWriteAddr_wb;
    input   reg [4:0] RsAddr_id;
    input   reg [4:0] RtAddr_id;
    output reg RsSel=0;
    output reg RtSel=0;

    always @(*)
        begin
            if((RegWrite_wb==1)&&(RegWriteAddr_wb!=0)&&
                (RegWriteAddr_wb==RsAddr_id))
                RsSel=1;
            else if((RegWrite_wb==1)&&
                    (RegWriteAddr_wb!=0)&&
                    (RegWriteAddr_wb==RtAddr_id))
                RtSel=1;
        end

endmodule
```

解决方法：将输入输出全部定义为 wire 型，用 assign 语句配合“？：”运算符直接对上述代码中的 RsSel、RtSel 进行赋值；或者输入定义为 wire 型，也可在 always 语句中简单的使用，但如果想对输入做复杂的操作，如移位等操作，则需要在模块中重新定义一个 reg 型的中间变量，将输入赋值给该变量，然后对该中间变量进行操作。

（2）赋值

在定义模块的接口或中间变量时只能赋值常量，不可以直接赋值一个运算型的语句，如下述错误定义：

```
    output   PC_IFWrite= ~stall;
```

解决方法：这种定义是错误的，需要分成两步写，如：

```
    output   PC_IFWrite;
    assign PC_IFWrite=~stall;
```

（3）进制的声明

在给某些常量赋值时或者与常量比较时，要声明常量的进制，如果没有声明常量的二进制位数和其进制，系统会自动默认为十进制，这样有些时候就会引起错误。如：

```
    always @(*)
        begin
            case({JR,J,Z})
                000: PC_in=NextPC_if;                    //next instruction PC+4
```

```
        001: PC_in=BranchAddr;                    //Branch
        010: PC_in=JumpAddr;                      //Jump
        100: PC_in=JrAddr;                        //JR
        endcase
    end
```

这样只有在{JR,J,Z}为 0 或 1 的时候是正确的，在另外两种情况下就是错误的，在仿真的时候出现的情况可能就是，有的时候正确，有的时候错误，让人误以为可能是其它部分出现了问题，不易被察觉。

解决方法：改为如下代码：

```
always @(*)
    begin
        case({JR,J,Z})
            3'b000:   PC_in=NextPC_if;            //next instruction PC+4
            3'b001:   PC_in=BranchAddr;               //Branch
            3'b010:   PC_in=JumpAddr;                 //Jump
            3'b100:   PC_in=JrAddr;                       //JR
            endcase
        end
```

另外，要养成习惯，无论是二进制、十进制、十六进制等其它任何进制，在出现常数时均要声明常量的二进制位数以及其本身使用的进制。

2. 流水线 MIPS CPU 结构中的错误

在代码语法错误检查完之后，接下来的错误就是在 MIPS CPU 结构中的错误，这些错误机器是不可能查出来的，只能经过仔细的分析，才能找到错误的根源。在解决这类问题时，可以对照仿真波形，从开始错误的那一点起，向前推，如新号来源、使能信号等，寻找错误来源。遇到的问题有：

（1）字母大小写错误

在 CPU 的设计中，由于出现的信号比较多，尤其是在这种流水线结构中，在不同的分级中，同一个信号会有不同的名称，按照“名称_流水线级名称”的格式定义。在名称的定义时，可能会出现英文字母大小写的不统一，这种不统一不一定会引起错误，但有着引起错误的潜在可能，如果对所有地方的信号都很熟悉可能会在信号连接时注意到这种问题，并得到解决，但有时候，如果没注意到这一点，可能就会引起错误，这种错误在仿真时，机器不一定会报错，但在仿真波形上看不到正确的结果。

例如：在提供的代码中有的写法是 MemtoReg，有的时候写成 MemToReg，如果没有注意到这一点细节，就有可能会引起错误。

解决方法：尽量避免这种大小写不统一，可以全部按照原理图中为标准去写，或者列出一个表格来，以表格中的信号为标准写法，全部统一。

（2）reset 信号

在流水线寄存器中均要加入 reset 的信号，以使 CPU 在初始时刻有一个稳定的状态，否则，CPU 可能从一开始就不正确导致后面一系列的错误。开始时，由于原理图中的流水线寄存器基本都没有画出 reset 信号的接入，所以流水线寄存器多数选择的是 DFF 型，但仿真时从开始时就出错，出现很多不稳定状态，后面的结果也均不正确。

解决方法：选择 dffr 或 dffre 类型的 D 触发器，接入 reset 信号。

（3）ALU 与 Decode 设计与赋值

在 ALU 与 Decode 模块中，有很多的赋值，在赋值时可能会出现漏掉某些赋值，如给 ALUCode 赋值时，可能会漏掉某些指令运算的赋值，如果这些指令在测试时没有用到，可能不会出现问题，但如果用到

了，就会引起错误。

解决方法：对照指导书中的表格，分类进行赋值，以避免遗漏。

（4）程序执行到某一条语句时无法继续执行

在本次实验中，开始时，运行到 Instruction_id=20080042H（即 addi 指令），程序会卡在这一条指令无法继续运行。

解决方法：ALU 或者 Decode 不需要操作，或者处在不确定的状态时，要给一个固定值，如 0：

```
default:        Result=32'b0;
```

经过赋值，程序得以正常运行。

3.未知原因的错误

在流水线寄存器使用时，顶层文件中的流水线信号传递有一处信号没有根据时序变化传递，而是二者始终保持一致，像是直接相连，D 触发器没有起到作用，但仔细检查没有发现任何错误。原代码如下：

```
wire [31:0] ALUResult_wb;
dffr #(32) dffr_MEMWB_ALUResult
    (
    .d(ALUResult_mem),
    .clk(clk),
    .r(reset),
    .q(ALUResult_wb));

dffr #(5) dffr_MEMWB_RegWriteAddr
    (
    .d(RegWriteAddr_mem),
    .clk(clk),
    .r(reset),
    .q(RegWriteAddr_wb));
```

此时 RegWriteAddr_mem 与 RegWriteAddr_wb 信号始终相等，并没有体现出时序的作用。

解决方法：把两个 D 出发器顺序调换，代码如下：

```
dffr #(5) dffr_MEMWB_RegWriteAddr
    (
    .d(RegWriteAddr_mem),
    .clk(clk),
    .r(reset),
    .q(RegWriteAddr_wb));

wire [31:0] ALUResult_wb;
dffr #(32) dffr_MEMWB_ALUResult
    (
    .d(ALUResult_mem),
    .clk(clk),
    .r(reset),
    .q(ALUResult_wb));
```

错误得到改正，不知道此处错误出在了哪。

七、总结、体会

经过这近半学期以来的实验，我对流水线的 MIPS CPU 有了一个细致的理解，对整个 CPU 的工作方式不仅是从整体上有了一个概况的了解，甚至很多细节也有了较为详细的了解。

我确实了解了提高 CPU 性能的方法，基本掌握了流线 MIPS 微处理器的工作原理。

比如之前我以为 Decode 的编码很简单，就是一个直接翻译的过程，但经过仔细参看 Decode 文件中提供的常量和部分语句，我才发现原来没有我想象的那么简单。ALU 之前在我的印象中就是一个加法器，但经过 ALU 的编写，我才知道，原来加法器只能算作 ALU 功能的一部分，为了实现更加快捷的操作，可以增加其它电路，虽然电路复杂了，但确实提高了 CPU 的性能。

本次实验还帮助我理解了数据冒险、控制冒险的概念，让我对流水线冲突的解决方法——转发与阻塞机制有了进一步的理解。

理论课上，对这些概念都只是一个大体概括性的理解，对数据路径等都只有一个简单的印象，如果详细去查就没有办法准确说出 CPU 的执行方式和数据的传递方式。对于转发和阻塞并不清楚真正是怎么执行的，但经过自己动手实践才知道其工作方式，和它具体处在哪些流水线级。流水线寄存器的工作方式，在这次实验中也有了完美的体现，让我对它的理解有了新的认识。

对流水线 MIPS CPU 的特点在本次实验中得到了充分体现，虽然最终完成的 CPU 的功能还比较简单，但对于理论的学习加深印象的同时，也让我学到了很多理论课学不到的东西，更加细节性的东西通过自己的动手有了较深的印象。

另外，通过本次实验，我还进一步掌握了流水线 MIPS 微处理器的测试方法，让我对 ModelSim 软件有了进一步的熟悉。我进一步学习了用 ModelSim 仿真波形对设计进行有效的分析。编写代码只是一个初步的过程。对代码进行调试才是一个相当艰巨的任务，代码编译通过后，仍然存在着很多的错误，需要自己一步步动手解决，对于这些问题的分析要采用有效的方法，如果只是盲目的去找可能会浪费非常多的时间！此外，我也对 IP 核的仿真有了更进一步的学习，对 SVGA 的显示方式有了一定的了解。

在使用教材的过程中，我发现教材里除了勘误表中的一些错误外，还存在着几点小错误，我将这些错误附在附录中说明，如果是我认识上的错误，还希望批评指正！

这次实验对我的帮助非常大！让我有了很大的提高，这不仅仅是对我这门课的提高，也极大的培养了我对这方面的兴趣，同时，实验时用到的一些思维方式也会对我在其它课程的学习有一定的帮助。

当然，这些进步、提高和最终实验的顺利完成都少不了老师们的帮助和同学的支持，在此对给我诸多帮助的唐老师、屈老师、马老师表示感谢！谢谢你们不厌其烦的解答我的困惑，谢谢你们在非上课时间仍然开放实验室并为我们解决问题，也要感谢老师一遍一遍地帮我仔细检查错误，更感谢你们传达给我的思想和思维方式，教会了我用有效的方式解决问题！谢谢！算是一年的相识，从数设 1 到数设 2，我学会了很多，希望在今后的学习中还能跟各位老师有更多的交流！

祝一切顺利！新年快乐！

2013 年 12 月 31 日

联系方式：

姓名：张先喆

专业：电子科学与技术

学号：3110100812

手机：15267052157 / 565474

邮箱：zhangxianzhe5@126.com

通信地址：浙江大学玉泉校区 32 舍 120 室

附录：

教材勘误新增

1. P.229 表格中 clk 所指的 11 行，Ac0b010c 应修改为 ac0b000c，因为该指令为：

sw $t3, 0C($0)

SW 为 I 类型指令格式，低 16 位为立即数域，所以应为 000c 才符合该指令，而且在仿真图和 InstructionROM 均表述为 ac0b000c。

2. P.229 表格中 clk 所指的 15 行，8d2c0008 应该修改为 8d2b0008，该指令为 LW 指令，为 I 类型指令格式，区别在与 rt 指向的分别是 11 和 12 号寄存器，即$t3 和$t4，当指令为 8d2c0008 时，指令为：

lw $t4, 08($t1)；

这在之前给的勘误表里 P.226 页倒数第二行已经修正为 lw $t4, 08($t1)；

但是在提供的源代码中，即 InstructionROM.v 文件里这一行指令仍为 8d2b0008，因为不影响程序接下来的运行结果，在仿真图中 Instruction_id 也是 8d2b0008，所以，P.226 中不需要修改，只要在 P.229 中 Instruction_id 改为 8d2b0008 即可。否则，在 InstructionROM.v 中也应作对应修正！