南京大学计算机科学与技术系

计算机组成原理实验

流水线 CPU 设计

15

指导老师：张泽生 蔡晓燕

2010/1/3

组长：071221148 仲琛

组员：071221079 马静雯

071221139 张莲舟

071221150 周洁

（注：按学号次序排序）

本实验设计并且完成了流水线 CPU，加入移位，乘除法的功能，解决了结构，数据，loaduse 冒险。全面测试，详尽对比单周期，多周期 CPU 的设计。本实验的成功完成离不开老师和助教团队的教导，在此特别感谢。

目录

一、实验目的

（1） 进一步掌握流水线 CPU 的基本思想

（2） 深入了解流水线 CPU 的设计方法

（3） 综合运用和总结 Verilog 编程方法和经验

（4） 深入了解数据冒险，结构冒险和控制冒险的产生原因和解决方案

（5） 通过对流水线的设计，进一步思考单周期 CPU，多周期 CPU 和流水线 CPU 的异同之处

二、实验仪器和平台

（1） 装有 Quartus II 软件的计算机。

（2） Altera DE2-70 开发平台

三、实验任务

本实验中完成的指令：

add rd,rs,rt
addu rd,rs,rt
addi rt,rs,imm
addiu rt,rs,imm
sub rd,rs,rt
subu rd,rs,rt
nor rd,rs,rt
xori rt,rs,imm
clo
clz
slt rd,rs,rt
sltu rd,rs,rt
slti rt,rs,imm
sltiu rt,rs imm
sllv rd,rt,rs
sra rd,rt,shamt
blez rs,imm
j target
lwl rt,offset(base)
lwr rt,offset(base)
lw rt, imm(rs)
sw rt,imm(rs)
div rs,rt
divu rs,rt
mult rs,rt
multu rs,rt

四、实验原理和步骤

4.1 MIPS 指令集说明

下面是我们是实现的 MIPS 指令集介绍：

指令	功能	说明
add,rd,rs,rt	M[PC] PC←PC+4 R[rd]←R[rs]+R[rt]	从 PC 所指的内存单元中取指令；PC 加 4，使 PC 指向下条指令；从 rs,rt 中取数后相加，结果送 rd
addu rd,rs,rt	R[rd]←R[rs]+[rt] temp←R[rs]+R[rt] R[rd]←temp	将寄存器 rs 与 rt 的和存入寄存器 rd 中，不产生溢出
addi rt,rs,imm	R[rt]←R[rs]+immediate	如果没有溢出就将寄存器 rs 与有符号立即数的和存入寄存器 rt 中，有溢出则寄存器 rs 与有符号立即数的和不存入寄存器 rt 中
sub rd,rs,st	M[PC] PC←PC+4 R[rd]←R[rs]-R[rt]	从 PC 所指的内存单元中取指令；PC 加 4，使 PC 指向下条指令；从 rs,rt 中取数后相减，结果送 rd
subu rd,rs,st	R[rd]←R[rs]-R[rt] Temp←R[rs]+R[rt] R[rd]←temp	将寄存器 rs 和 rt 的差(无符号数的减法) 存入寄存器 rd 中，不产生溢出
nor rd,rs,rt	R[rd]←R[rs] nor R[rt]	将寄存器 rs 与 rt 安慰逻辑或非的结果存入寄存器 rd 中
Xori rt,rs,imm	R[rt]←R[rs]xor zero_extend(immediate)	将寄存器 rs 与 0 扩展立即数的诸位逻辑异或结果存入寄存器 rt 中
Clo	R[rd]←count_leading_ones R[rs]	将寄存器 rs 中数据其实为一的个数存入 rd 中，如果字中都是 1，结果为 32
clz	R[rd]←count_leading_zeros R[rs]	将寄存器 rs 中的数据起始为 0 的个数存入 rd 中，如果字中都是 0，结果为 32
Slt rd,rs,rt	R[rd]←(R[rs]<R[rt])	若寄存器 rs 比 rt 小，寄存器

		rd 置 1；否则，rd 置 0

Sltu rd,rs,rt	PR[rd]←(GPR[rs]<GPR[rt])
	If(0||GPR[rs])<(0||GPR[rt])then
	GPR[rd]←0GPRLEN-1||1
	Else
	GPR[rd]←0GPRLRN
	endif
		若寄存器 rs 比 rt 小（有符号数），寄存器 rd 置 1；否则，rd 置 0，不产生溢出

Slti rt,rs,imm	GPR[rt]←(GPR[rs]<immediate)	寄存器 rs 与有符号数扩展立即数进行有符号比较若小，寄存器 rt 置 1，否则，置 0
Sltiu rt,rs imm	GPR[rt]←(GPR[rs]<immediate)	寄存器 rs 与有符号数扩展立即数进行无符号数比较若小，寄存器 rt 置 1；否则，rt 置 0
Sllv rd,rt,rs	GPR[rd]←GPR[rt]<<rs	由 rs 指定寄存器 rt 的左移数，并将结果存入寄存器 rd
Sra rd,rt,rs	)	
Sra rd,rt,shamt	GPR[rd]<-GPR[rt]>>shamt(arithmetic	由立即数 shamt 指定寄存器 rt 的算术右移位数，并将结果存入寄存器 rd
Blez rs,imm	If GPR[rs]<=0 then branch I:taget_offset←sign_extend(offset||02) Condition←GPR[rd]<=0GPRLEN I+1： if condition then PC←PC + target_offset endif	若寄存器 rs 小于等于 0，转移指令数由有符号偏移量左移 2 位来决定
J target	M[PC] PC<31:2>←PC<31:28>||target<25:0>	从 PC 所指的内存单元中取指令； 计算目标地址，符号||表示“拼接”。 跳转目标地址的范围是：当指令后的 0000000H~FFFFFFCH
Mul rd,rs,rt	GPR[rd]←GPR[rs]*GPR[rt]	将 rs 和 rt 乘积的低 32 位存入寄存器 rd 中
Mult rs,rt	（HI,LO）←GPR[rs]*GPR[rt]	将寄存器 rs 和 rt 的数据相乘，乘积的低位和高位数分别存入寄存器 HI 和 LO
Multu rs,rt	(HI,LO)←GPR[rs]*GPR[rt]	寄存器 rs 和 rt 的数据相乘 ,乘积的低位和高位数分别存入寄存器 HI 和 LO
Div rs,rt	(HI,LO)←GPR[rs]/GPR[rt]	寄存器 rs 被寄存器 rt 除 ,将商存入寄存器 LO，余数存入 HI

Divu rs,rt	(HI,LO)←(HI,LO)+(GPR[rs]*GPR[rt])	将 rs 和 rt 的乘积所得的 64 位结果与链接寄存器 LO 和 HI 中的 64 位值相加

4.2 三种 CPU 设计

4.2.1 单周期 CPU

计算机的性能由三个关键因素决定：指令数目，时钟周期，CPI。其中，指令数目由编译器和指令集决定；时钟周期和 CPI 由处理器的设计和实现决定。单周期处理器每条指令在一个时钟周期内完成，所以 CPI 为 1，而时钟周期往往很长，通常取最长的指令周期。

单周期 CPU 的特点是每条指令的执行只需要一个时钟周期，一条指令执行完再执行下一条指令。再这一个周期中，完成更新地址，取指，解码，执行，内存操作以及寄存器操作。由于每个时钟上升沿时更新地址，因此要在上升沿到来之前完成所有运算，而这所有的运算除可以利用一个下降沿外，只能通过组合逻辑解决。这给寄存器和存储器 RAM 的制作带来了些许难度。且因为每个时钟周期的时间长短必须统一，因此在确定时钟周期的时间长度时，要依照最长延迟的指令时间来定，这也限制了它的执行效率。

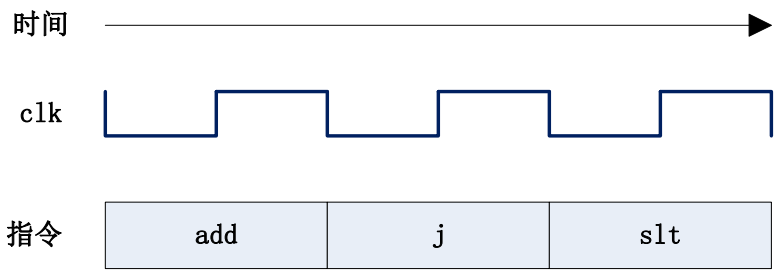


图 1 CPU 指令执行时序图

4.2.2 多周期 CPU

多周期处理器的基本思想是：把每条指令的执行分为多个大致相等的阶段，每个阶段在一个时钟周期内完成；各阶段内最多完成一次访存或一次寄存器读写或一次 ALU 操作，个阶段的执行结果在下个时钟到来时保存到相应的存储单元或稳定的保持在组合电路中，时钟周期的宽度以最复杂阶段所花的时间为准，通常取一次存储器读或写的时间。

（1） 取指令阶段

（2）译码/读寄存器堆阶段

（3）地址生成阶段（ALU运算）

（4）读存储器阶段

（5）写结果到寄存器

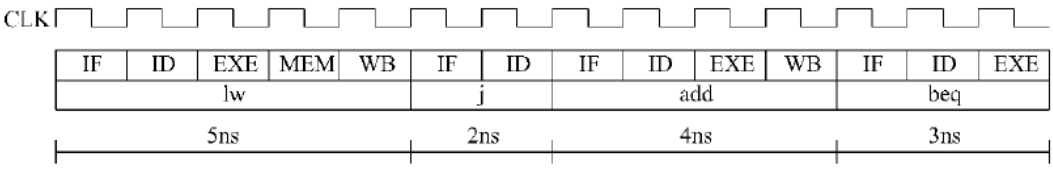


图1多周期时序示意图

多周期中，一个阶段称为一个周期，lw用5个周期，add用4个周期，beq用3个周期，j用2个周期。

4.2.3 流水线 CPU

和工业流水线相似，流水线的核心思想是把多条指令的执行重叠起来。在任何时候，CPU 同时处理多条指令，这些指令分处于不同的运行周期，使用不同的物理器件。

MIPS 指令集是一种典型的 RISC 体系结构的指令集。在 RISC 体系结构中，使用流水

线来提高运行速度是其本质的特征之一。在 MIPS 指令集中，有多个方面体现了对流水线结构的适应和支持。

（１）　所有的 MIPS 指令长度都相同。这种设计将简化指令逻辑和指令译码逻辑，使得取指令ＩＦ周期和指令译码ＩＤ周期能够以更快的速度执行。

（２）　MIPS 指令集中只有 R 类型、 Ｉ类型和 J 类型几种很少的指令类型。这样，指令字中各个域的位置相对固定。源操作数的寄存器号和目的寄存器号都很容易从指令中分离出来。这样，就可以在指令译码的同时从寄存器堆中读出指令的源操作数，从而简化了指令周期。

（３）　MIPS 指令集中绝大多数指令都是寄存器操作指令，只有 load 指令和 store 指令涉及存储器的操作。这样 ,存储器访问的操作就非常简单 ,从而可以减少存储器访问的时间。

（４）　MIPS 指令在存储器中按 32 位对齐。这样，就可以仅仅使用一次存储器操作来读入一条指令字，从而减少指令的读取时间。

流水线的基本原理是把一个重复的过程分解为若干个子过程 ,前一个子过程为下一个子过程创造执行条件，每一个过程可以与其它子过程同时进行。流水线各段执行时间最长的那段为整个流水线的瓶颈，一般地，将其执行时间称为流水线地周期。

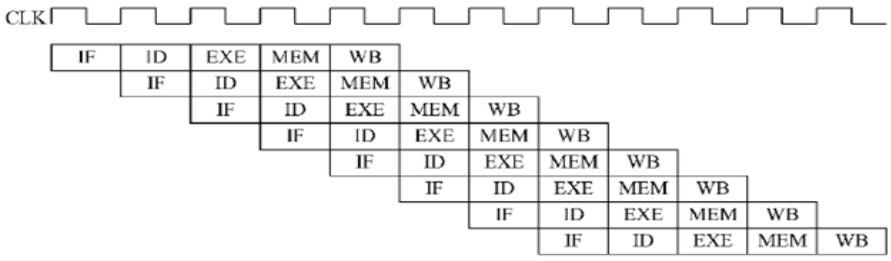


图 流水线流水指令示例

4.2.4 更高性能 CPU

效率较高的 MIPS CPU 一般具有如下结构：

（1）高速缓存 cache，提高存储器访问指令的执行速度。

（2）存储器处理单元，支持虚拟地址，具有 TLB（Translation Lookaside Buffer）即快速地址转换表，能够处理虚拟地址到物理地址的转换。

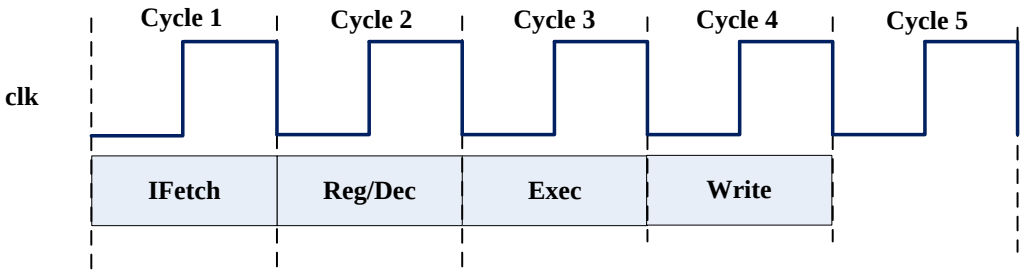
（3）多发射结构，在同一个时钟周期上可以流出 2-3 条指令并行运算，即提供了多条并行的流水线。

（4）支持向量运算，比如 SIMD（Single Instruction Multiple Data）结构，一次性处理一批具有相同运算方式的数据。用以对多媒体和图像处理提供更好的支持。

（5）转移预测，虽然 MIPS 提供了分支延迟槽技术，使得跳转只需延迟一个时钟周期，且得到兼容的支持。但转移预测可以做到这个延迟的避免，进一步提高 CPU 的转移效率。

4.3 指令在流水线 CPU 中的执行

4.3.1　R 型指令

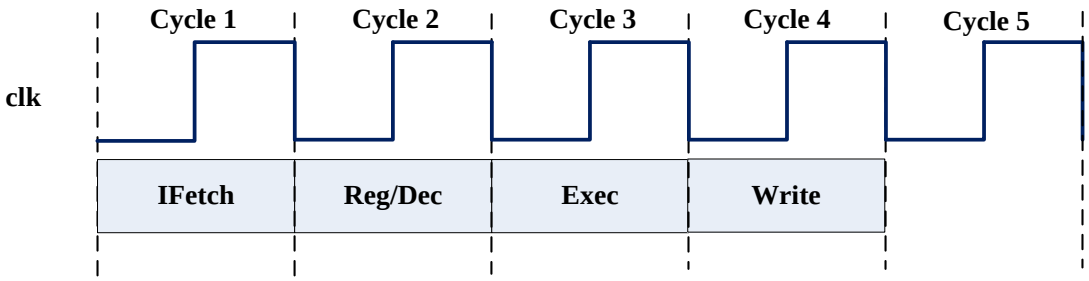


R 型指令执行流程

Ifetch 阶段	从指令存储器中取指令并计算 PC+4
Reg/Dec 阶段	寄存器取数，同时对指令进行译码

Exec 阶段	在 ALU/移位器/乘除法器内进行操作数运算（乘除法器是使能信号而非时钟信号触发，避免如果指令不是乘除法操作而依然进行乘除运算造成的效率降低），并将这三个结果和 32’ b0 送入四路选择器进行选择，得到执行结果。
Mem 阶段	空
Wr 阶段	将 Exec 阶段的执行结果写到寄存器。

4.3.2 I 型指令

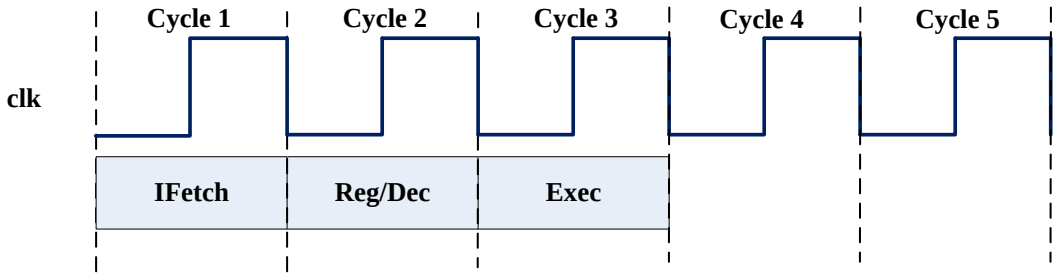


I 型指令执行流程

Ifetch 阶段	从指令存储器中取指令并计算 PC+4
Reg/Dec 阶段	寄存器取数，同时对指令进行译码
Exec 阶段	立即数在位扩展器内进行符号扩展或零扩展，在 ALU/移位器/乘除法器内进行操作数运算（乘除法器是使能信号而非时钟信号触发，避免如果指令不是乘除法操作而依然进行乘除运算造成的效率降低），并将这三个结果和 32’ b0 送入四路选择器进行选择，得到执行结果。

Mem 阶段	空
Wr 阶段	将 Exec 阶段的执行结果写到寄存器。

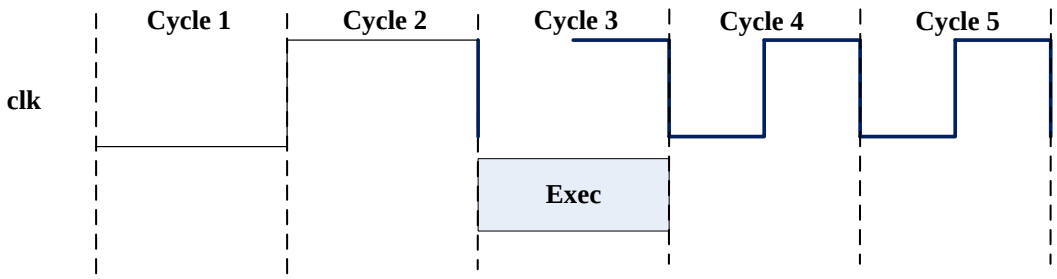
4.3.3 Branch 指令



Branch 指令执行流程

Ifetch 阶段	从指令存储器中取指令并计算 PC+4
Reg/Dec 阶段	寄存器取数，同时对指令进行译码
Exec 阶段	在 ALU 中做减法判断条件是否满足，同时用一个加法器计算转移地址。 如果条件满足，则把转移目标地址写到 PC 中
Mem 阶段	空 注： 教材上说 Branch 指令地址转移的时间是在 Exe 和 Mem 阶段之间，故将 Branch 跳转归到第四阶段，而在本次实验中，放在了第三阶段
Wr 阶段	空

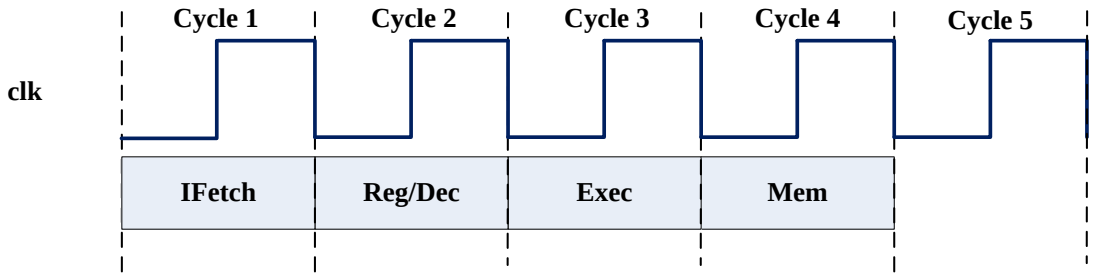
4.3.4 Jump 指令



Jump 指令执行流程

Ifetch 阶段	从指令存储器中取指令并计算 PC+4
Reg/Dec 阶段	寄存器取数，同时对指令进行译码
Exec 阶段	目标地址计算，如果跳转条件满足，则把转移目标地址写到 PC 中
Mem 阶段	空
Wr 阶段	空

4.3.5 Load 指令

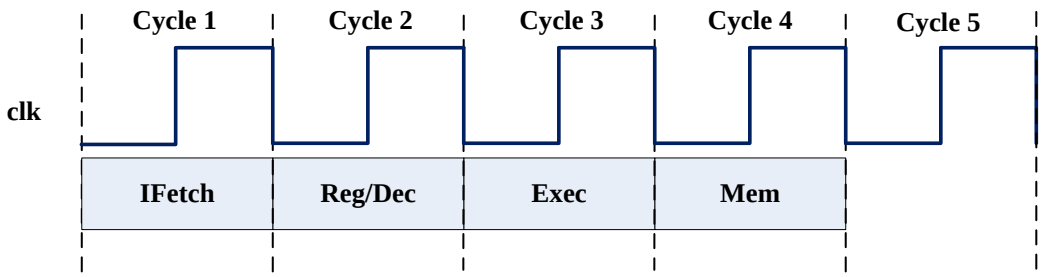


Load 指令执行流程

Ifetch 阶段	从指令存储器中取指令并计算 PC+4

Reg/Dec 阶段	寄存器取数，同时对指令进行译码
Exec 阶段	计算内存单元地址
Mem 阶段	从数据存储器中读数据
Wr 阶段	将数据写到寄存器目标单元地址中

4.3.6 Sw 指令



sw 指令执行流程

Ifetch 阶段	从指令存储器中取指令并计算 PC+4
Reg/Dec 阶段	寄存器取数，同时对指令进行译码
Exec 阶段	16 位立即数符号扩展后与寄存器值相加，计算主存地址
Mem 阶段	将寄存器读出的数据写到主存
Wr 阶段	空

4.4 流水线 CPU 的设计思想

这里我们从横向和纵向两个方面来考虑。

首先从横向考虑，流水线是把处理的过程分成若干阶段，每一阶段中通过组合逻辑进行某些操作。根据 MIPS 处理器指令的特点 ,将整体的处理过程分为 5 个阶段 ,也就是 Ifetch(取指)，Reg/Dec(取数和译码)，Exec(执行)，Mem(存储器操作)，Wr(写寄存器)。也就是说，一个指令的执行需要 5 个时钟周期，每个时钟周期的下降沿来临时(我们的实现存储器写是上升沿触发的，其他都是下降沿触发)，此指令所代表的一系列数据和控制信号将转移到下一个周期的组合逻辑输入上，在经过组合逻辑延时后，处理过的数据在组合逻辑的输出端产生，并等待下一个时钟沿到来时被转移到下一个周期去。

从纵向看，CPU 的每一级电路都在不同的时期上，IF 总要比 ID 要早一个时钟周期，比 EXEC 早两个周期，比 Mem 早三个周期，比 WB 早四个周期。也就是说，在某一个时刻，有五条指令处在五个不同的阶段。每条指令的控制信号都是在 ID 段产生，存储在流水段寄存器内，EXEC 段上如果要采用 ID 段产生的某个控制信号，则要在 1 个时钟周期后使用。

4.5 在流水线 CPU 指令数据通路的设计

4.4.1 Ifetch（IF）段

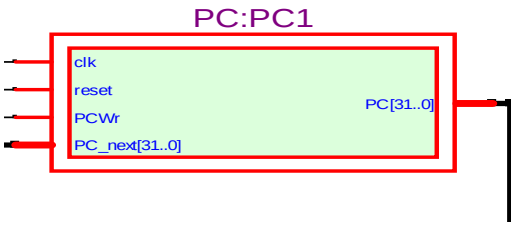
功能：

将 PC 的值作为地址送入指令存储器 IM 取得指令，并计算 PC+4,送 PC 输入端，IF 段的执行结果将送到 IF_ID 寄存器输入端，以便在下个时钟到来时，在 IF_ID 寄存器输入端的信息将被送到 ID 段继续执行。显然，在 IF_ID 寄存器中应当存放下列信息：从 IM 中取出的指令；PC+4 的结果。其中，存放 PC+4 的结果的目的是如果当前指令时分支指令，则它在 Ex 阶段要用来计算分支目标地址。

主要部件（这里仅介绍部件接口，内部结构将在下面详细介绍）：

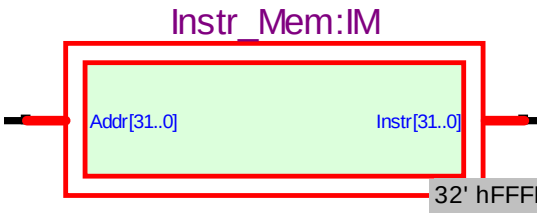
（1）PC

PC 寄存器，存放下一指令地址。



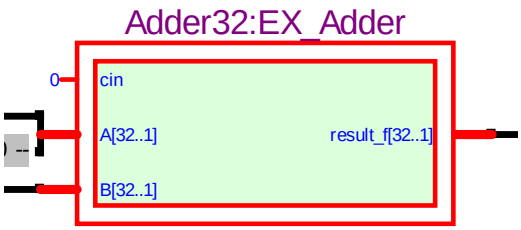
（2）IM

IM 指令存储器，存放指令。



（3）32 位全并行加法器

执行两位 32 位操作数的加法运算，IF 阶段无需返回 Zero，Less，Overflow 等结果。



4.4.2 IReg/Dec(ID)段

功能：

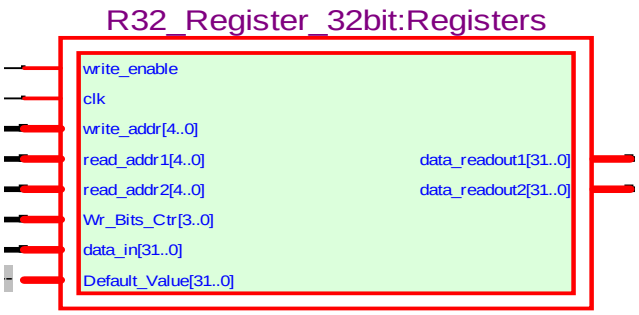
根据指令中 Rs 和 Rt 的值到寄存器中取得相应的值，同时根据指令中的操作码 Op 和功

能码 func 字段进行译码，生成相应的控制信号。ID 段的功能由寄存器读口和控制部件完成。ID 段执行的结果被送到 ID_EX 寄存器输入端，下个时钟到来时，ID_EX 寄存器输入端的信息被送到 Ex 段继续被处理。

主要部件（部件将在下面详细介绍**）**

（1）32 位寄存器组

32 位寄存器组，存储将要执行运算的操作数，以及中间结果。



4.4.3 Exec（Ex）段

功能：

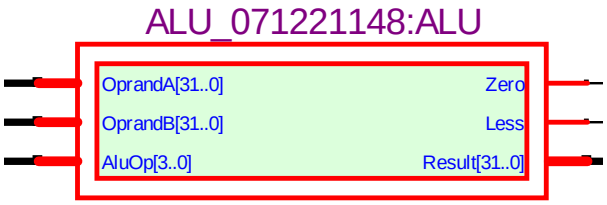
Ex 段是 CPU 的核心阶段，Ex 段的功能有具体指令而定，不同的指令译码后得到不同的控制信号，用来控制执行部件进行不同的操作，由于在我们的设计中没有在 **ALU** 中完成移位和乘除功能，而是采用了**桶形移位器**和除法乘法器，其中乘法和除法，我们有**带符号乘法器，无符号乘法器，带符号除法器，无符号除法器**，都封装在一个乘除模块中。

若是 blez 或者 jump 指令，则将 Ex 段生成的转移目标地址更新到 PC 中。

主要部件（部件将在下面详细介绍**）：**

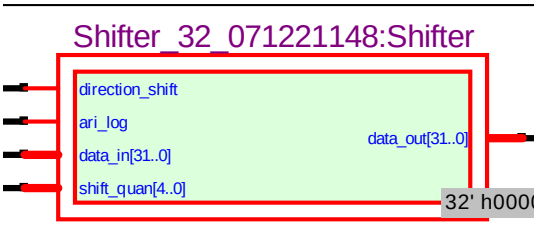
（1）ALU 部件

完成加、减、逻辑基本运算。



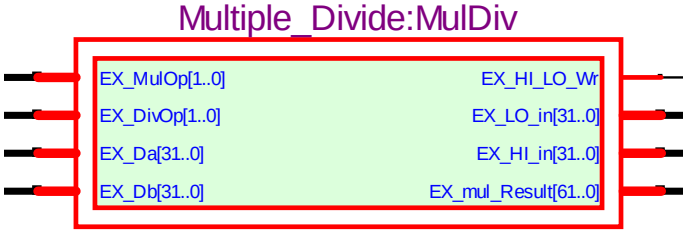
（2）32 位桶形移位

完成移位运算。



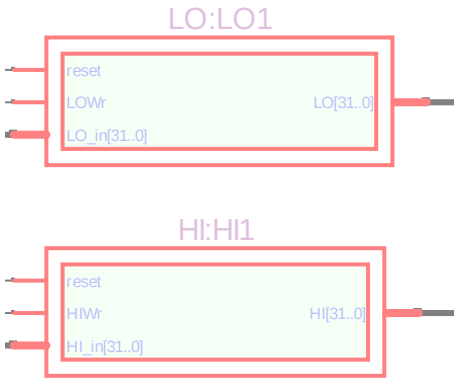
（3）乘法，除法部件

完成带符号乘法，无符号乘法，带符号除法，无符号除法运算。



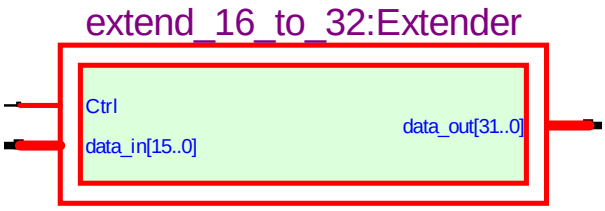
（4）LO,HI 寄存器

存储乘除法结果的寄存器。乘法：HI 存储高 32 位，LO 存储低 32 位。除法：LO 存储商，HI 存储余数。



（5）扩展器

将 16 位数扩展为 32 位，有符号扩展和逻辑扩展。



4.4.4 Mem 段

功能：

若为 R 型指令或者 I 型指令，Mem 段无操作 ,因此只需把相应的信息继续传下去即可。

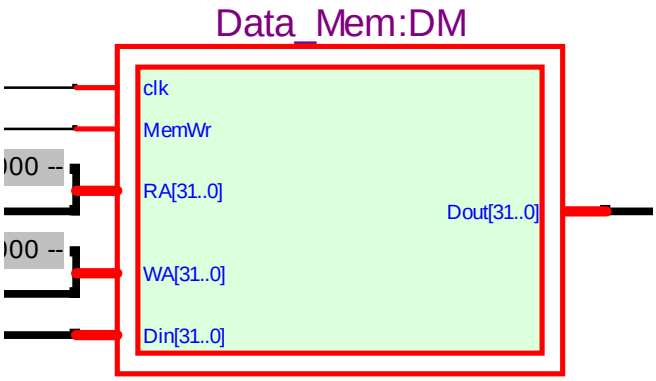
若是 lw，lwl，lwr 指令，则进行取数操作。在 Ex 段得到的地址被送到数据存储器 DM 的读地址端 经过一段存取时间 数据从 DM 的输出端 Do 送到 Mem_Wr 寄存器的输入端。

若是 sw 指令 则进行取数操作。在 Ex 段得到的地址被送到数据存储器 DM 的写地址端，同时把 Ex_Mem 寄存器送来的要存的数据送 DM 的数据输入端 Di，经过一段存取时间后，数据被存入 DM 中。

主要部件（部件将在下面详细介绍）：

（1）数据存储器

以字节编址，存储数据。



4.4.5 Wr 段

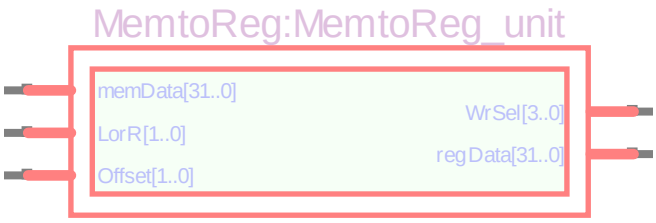
功能：

若为 R 型指令或者 I 型指令，则将 ALU 的输出结果或者桶形移位器，乘除器的计算结果送入寄存器堆的输入端 Di，目的寄存器送写地址端 Rw。若是 lw 指令，则选择 DM 读出结果送寄存器堆的输入端 Di ,目的寄存器送写地址端 Rw。其他指令任何寄存器的值都不改变，即不能写寄存器。

主要部件（部件将在下面详细介绍）：

（1）MemtoReg

针对 lw,lwl,lwr 指令，根据 Offset，将从 DM 中取出的值做一定的移位，并输出寄存器八位组的写使能信号。

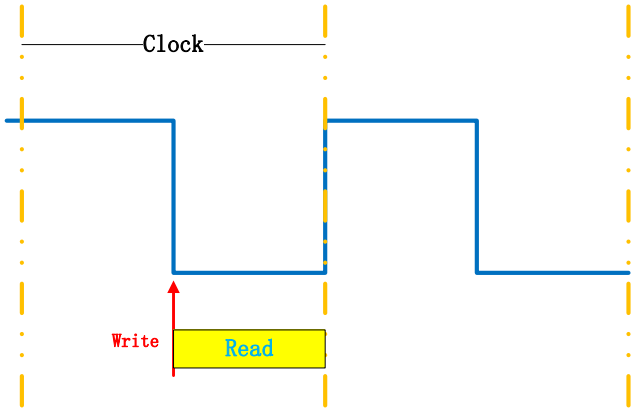


4.5 流水线 CPU 各个功能组件的设计

4.5.1 寄存器组的设计和实现

功能冒险和竞争冒险的解决：

我们是这样解决这两个问题的：首先，因为我们的实现中都是时钟的下降沿触发，而在寄存器和存储器写，为了解决写地址和写数据，写使能的竞争冒险，我们定义的是时钟上升沿触发写，然后为了上半周期写，下半周期读，我们定义只能在时钟为 1 的时候才可以读出，具体如下图：



存储器的读写时机

4.5.1.1 通用寄存器组

1、　功能

寄存器堆位于 CPU 内部，用于暂存要使用的数据，以供 ALU 的使用。若 DM 中的数据要参与运算，都必须先 load 进寄存器堆中。因此寄存器堆是 CPU 中一个十分重要的部件。

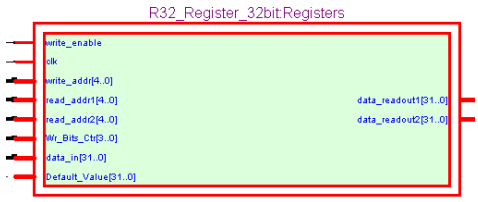
2、　原理

它由 32 个 32 位的寄存器构成，其中 0 号寄存器是一个很特殊的寄存器，它只能用来读，而且里面的值总是 0。它有两个读口，分别对应按指令读出的 Rs 和 Rt 两个寄存器中的值，送到 busA 和 busB 上；有一个写口，在时钟的控制下，如果当前写寄存器信号 write_enable 为低电平，表示可写，则根据 Wr_Bits_Ctr 指定的地址写入输入的数据 data_in。读出数据不需要时钟控制。

3、　流水线的特殊点

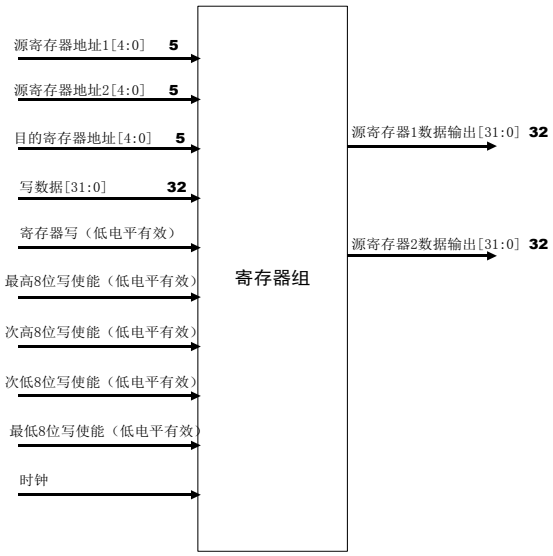
由于要实现 lwl 和 lwr 的功能，因此用 Wr_Bits_Ctr[3]、Wr_Bits_Ctr[2]、Wr_Bits_Ctr[1]、Wr_Bits_Ctr[0]的四位分别控制该单个寄存器的高八位、次高八位、次低八位和低八位是否写入。我们在写入前先执行一定的操作将输入数据和要输入的位置对准，然后时钟到来时通过要写入的某几位的信号是否为低电平决定是否写入。这一点和多周期是相同的，为了避免结构冒险，我们采用上升沿写入，下降沿读出的方式。

4、　RTL 图

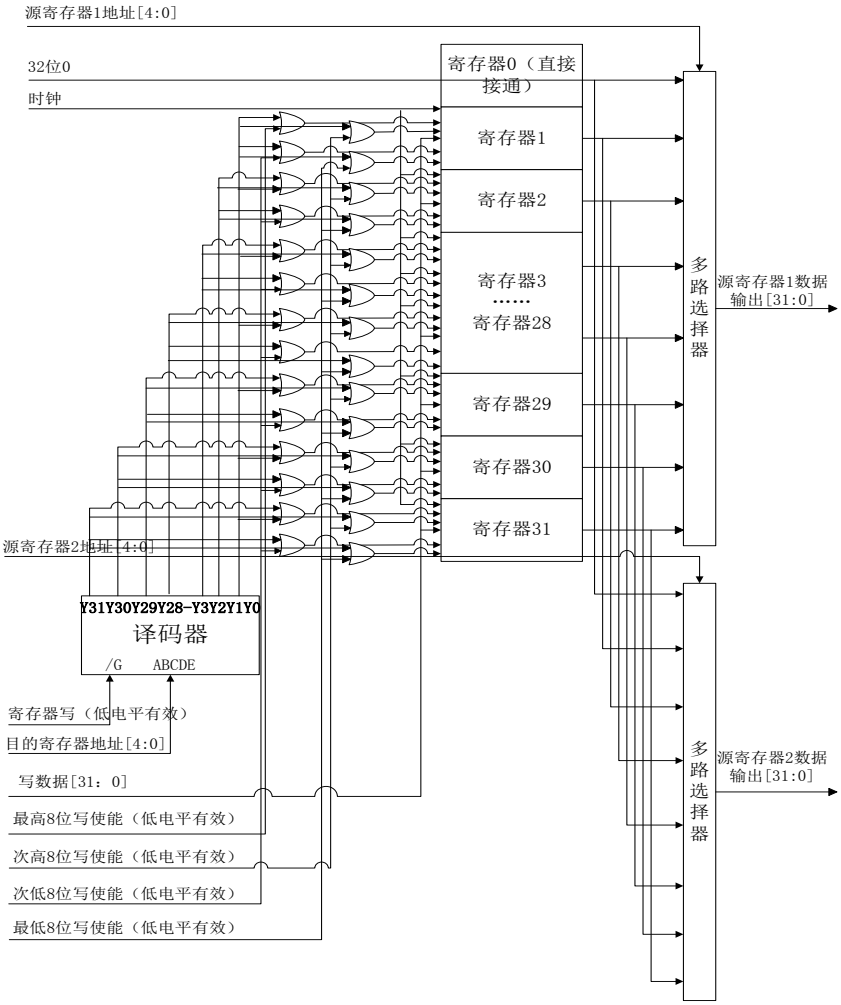


5、　电路图

原理图：


源寄存器地址1[4:0] 5
源寄存器地址2[4:0] 5
目的寄存器地址[4:0] 5
写数据[31:0] 32
寄存器写（低电平有效）
最高8位写使能（低电平有效）
次高8位写使能（低电平有效）
次低8位写使能（低电平有效）
最低8位写使能（低电平有效）
时钟
寄存器组
源寄存器1数据输出[31:0] 32
源寄存器2数据输出[31:0] 32

实现细节图：


源寄存器1地址[4:0]
32位0
时钟
寄存器0（直接接通）
寄存器1
寄存器2
寄存器3
……
寄存器28
寄存器29
寄存器30
寄存器31
多路选择器
源寄存器1数据输出[31:0]
源寄存器2地址[4:0]
Y31Y30Y29Y28-Y3Y2Y1Y0
译码器
/G ABCDE
寄存器写（低电平有效）
目的寄存器地址[4:0]
写数据[31：0]
最高8位写使能（低电平有效）
次高8位写使能（低电平有效）
次低8位写使能（低电平有效）
最低8位写使能（低电平有效）
多路选择器
源寄存器2数据输出[31:0]

4.5.1.2 流水线寄存器组

（1）IF_ID 寄存器

1、　功能：

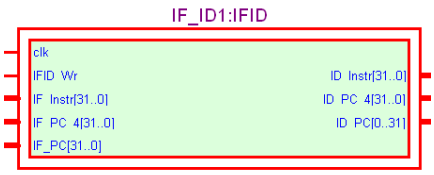
该寄存器位于取指令逻辑和指令译码逻辑之间。用于保存指令和 PC 地址、PC+4 的值供之后的阶段使用。

2、　原理：

首先我们看一下在这两个阶段之间有哪些数据是要保存的。最直观的就是指令，因为指令取出以后要把它传到下个周期供指令译码使用，因此指令本身一定要保存起来。然后，如果这条取出的指令是分支指令，那么 PC+4 的结果一直要保存到执行阶段（Ex）参与转移地址的计算，因此 PC+4 也要保存起来。由于之后要计算 J 型指令的地址（PC[32:28]||Target[25:0]||00），因此 PC 本身也要保存起来。

小结：IF_ID 寄存器中保存的是 指令 和 PC+4 和 PC。

3、　RTL 图展示：



（2）ID_EX 寄存器

1、　功能：

该寄存器位于取指令逻辑和指令译码逻辑之间。除了前面保存的内容之外，还需要保存那些在指令译码阶段产生的数据和控制信号供之后的阶段使用。

2、　原理：

首先我们看一下在上一次写 IF_ID 寄存器到这一次写 ID_EX 寄存器之间发生了什么事。

1、指令读出，对指令译码，产生了立即数 ID_imm16;寄存器读口地址 ID_Rs,ID_Rt, 寄存器写口地址 ID_Rd,还有信号 ID_shamt，ID_Target；

2、从寄存器堆中读出两个操作数 ID_Da,ID_Db；

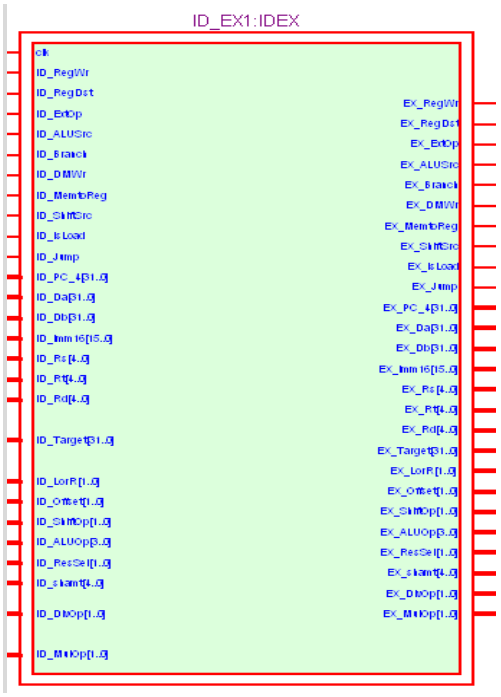
3、将指令送到控制信号生成部件得到一系列控制信号：

信号	功能
ID_ResSel	选择移位的结果
ID_LorR	判断是 lwl 还是 lwr
ID_Offset	判断 lwl 和 lwr 时的偏移量
ID_ShiftOp	判断是左移、算数右移还是逻辑右移
ID_ALUOp	判断 ALU 的操作符
ID_Jump	判断是否为 J 型跳转指令
ID_IsLoad	判断是否为 Load 指令
ID_ShiftSrc	判断要移位的位数是由 Ra 还是 shamt 决定的
ID_RegWr	寄存器写使能
ID_RegDst	判断是写到 Rt 还是 Rd 所指定的寄存器中
ID_ExtOp	判断是有符号扩展还是无符号扩展
ID_ALUSrc	判断是 busB 还是立即数参加 ALU 的运算
ID_Branch	判断是否为分支指令
ID_DMWr	数据寄存器写使能
ID_MemtoReg	判断是否能将数据从数据存储器传输到寄存器写端口
ID_DivOp	判断是有符号除法还是无符号除法
ID_MulOp	判断是有符号乘法还是无符号乘法

4、由于 PC 在计算完 j 型指令地址之后已经用不到了，指令本身也已经完成了译码工作，因此前一个流水线寄存器中留下来的数据中只需要保存 PC+4 的值。（ID_PC_4）

这些信号也是下降沿写入，下半周期读出。

3、　RTL 图展示：



（3）EX_MEM 寄存器

1、　　功能：

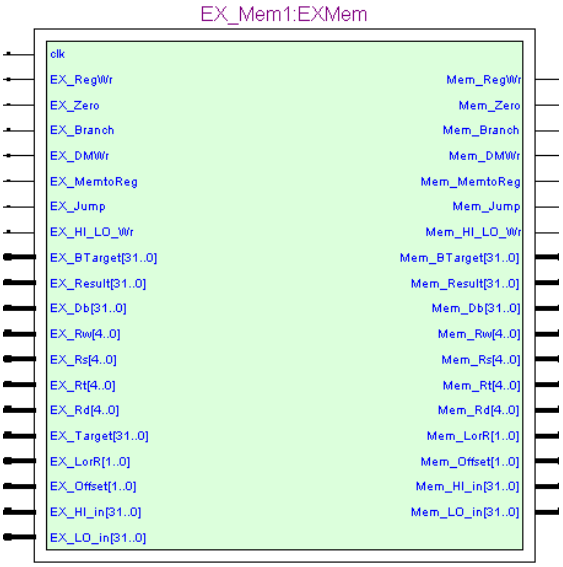
这个寄存器位于 EX 执行阶段和 MEM 阶段之间，主要保存执行阶段得到的计算结果和之后还要用到的控制信号等。

2、　　原理：

首先我们看一下在这之前的那个阶段做了些什么。1、执行阶段 ALU 计算出的结果或桶形移位器的结果 EX_Result 要存起来,同时输出结果是否为 0 的信号 EX_Zero。2、因为下一阶段如果是 sw 指令，那么需要把寄存器 Rb 内的数据写入数据寄存器，因此需要记录 EX_Db。3、如果本条指令是跳转指令，那么下一个阶段就要把新的 PC 送到 PC 寄存器的写口，因此要记录该跳转信号 EX_Jump ,EX_Branch 以及跳转地址 EX_Target 和 EX_BTarget。4、写阶段写数据寄存器要用到的数据寄存器写使能 EX_DMWr。5、还有后续阶段写寄存器时要用到的信号，这些信号将在下一个流水线寄存器的介绍中详细说明，这些信号有：EX_Rw,EX_Rs,EX_Rt,EX_Rd,EX_Offset,EX_LorR,EX_RegWr,EX_MemtoReg。

6、乘除法要用到的数据 EX_HI_LO_Wr 表示乘除法寄存器的写使能，EX_HI_in 表示乘除法计算得到的高 32 位数据，EX_LO_in 表示乘除法计算得到的低 32 位数据;

3、　RTL 图展示



（4）MEM_WB 寄存器

1、　功能：

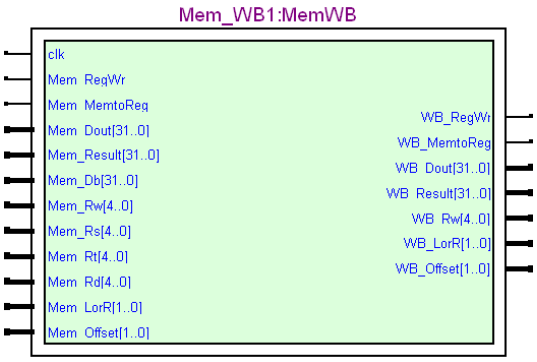
这个寄存器位于 MEM 阶段和写寄存器阶段之间，用于保存要写入寄存器的数据、写入的控制信号、寄存器写口地址等信息。

2、　原理：

我们来看一下这个寄存器具体要存的数据或信号，首先因为可能是 load 指令，因此要保存从数据寄存器读出来的数 Mem_Dout ,若不是那么要保存 ALU 或桶形移位器的计算结果 Mem_Result，要写入的寄存器的地址 Mem_Rw 以及寄存器写入的控制信号：

```
Mem_Offset      //控制写入的偏移量
Mem_LorR            //控制是 lwl 还是 lwr
Mem_RegWr       //写使能信号
Mem_MemtoReg    //判断是否允许把数据从数据寄存器读到寄存器写口
```

3、 RTL 图展示：



可见，随着流水线的推移，流水线寄存器中需要存储的数据越来越少。

4.5.2 ALU 部件的设计和实现

1 功能：

MIPS 中，ALU 可执行的功能与操作见下面的表 1，需三位控制信号。

“ALUop”（外部输入的 ALUctr）	ALUctr	操作
0000 **0010**	110	加法 无符号加
0001 **0011**	110	有符号减 无符号减
0111	011	或非
0101 **0100**	000	前导一 前导零
1101 **1111**	101	slt sltu
1000	001	异或

表 1 ALU 操作表

2 原理：

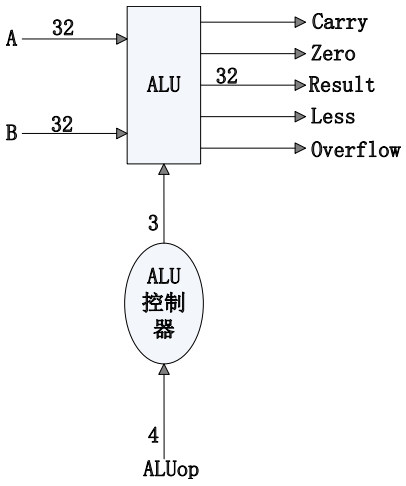
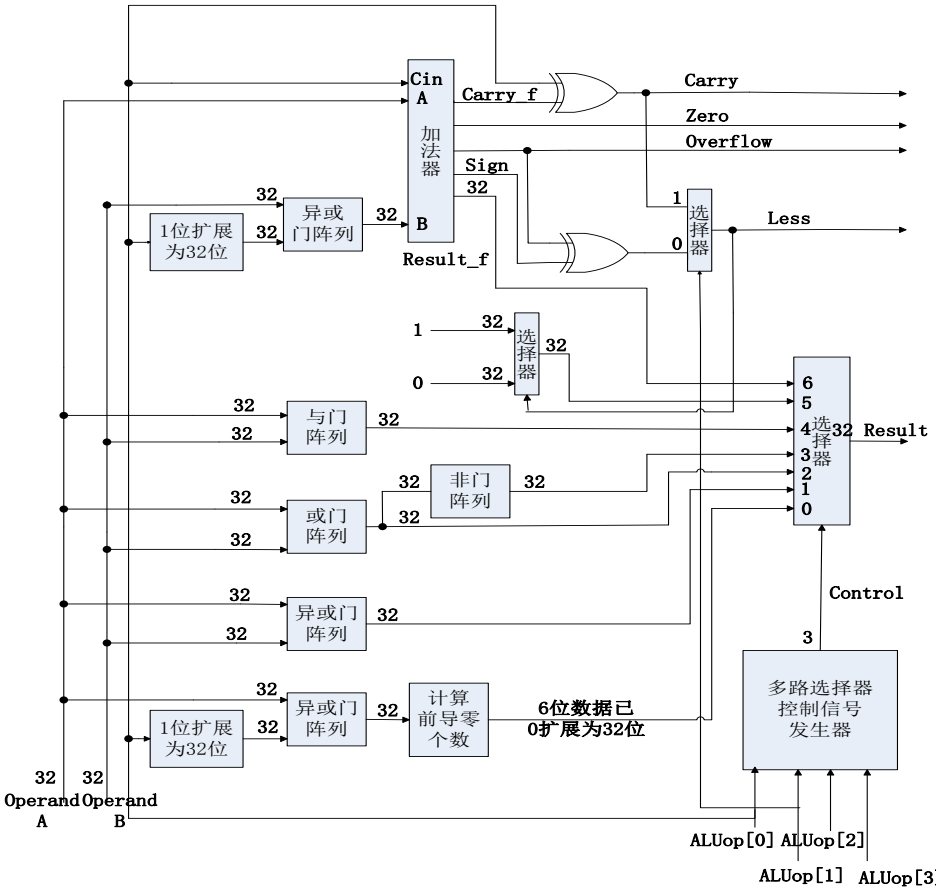


图 1 ALU 原理示意图

为提高 ALU 的控制效率，ALU 采用两级控制，即通过 ALU 控制器实现对 ALU 的控制，而不是直接控制 ALU。图 1 和图 2 分别为 ALU 原理示意图。



除输出运算结果 Result，ALU 还输出信号 zero, Less, Overflow，分别表示运算结果是

否为 0，两数比较是大还是小，是否有溢出，以用于某些判断指令。

说明：ALUop4 位最低位控制加减法以及前导零还是前导一，优点是，无需额外译码倒数第二位控制作有无符号判定，有无符号数判定大小逻辑不同(less 标志)。两个有符号数比较，V 异或 S 的结果为 less，两个无符号数比较，C 的结果为 less。

核心部件

（1）32 位超前进位加法器

在一些对计算速度要求较高的地方，我们需要用复杂的电路设计来达到提升运算速度：

$$C_0 = A_0B_0 + (A_0 \oplus B_0)C_{0-1}$$

$$C_1 = A_1B_1 + (A_1 \oplus B_1)C_0 = A_1B_1 + (A_1 \oplus B_1)[A_0B_0 + (A_0 \oplus B_0)C_{0-1}]$$

…

$$C_i = A_iB_i + (A_i \oplus B_i)\,C_{i-1}$$

定义两个辅助函数：Gi=XiYi…进位生成 Pi=Xi+Yi…进位传递（或 Pi=Xi⊕Yi ）

全加逻辑方程：Si=Pi⊕Ci

Ci+1=Gi+PiCi (i=0,1,…n)

C1=G0+P0C0

C2=G1+P1C1=G1+P1G0+P1P0C0

C3=G2+P2C2=G2+P2G1+P2P1G0+P2P1P0C0

C4=G3+P3C3=G3+P3G2+P3P2G1+P3P2P1G0+P3P2P1P0C0

…

（2）前导零/前导一模块

对于一个 32 位的数据（实验中的 OperandA），前导零为计算该数据从高位到低位第一个 1 之前的 0 的个数，前导一为计算该数据从高位到低位第一个 0 之前的 1 的个数。前导零和前导一虽然结果不同，但原理是一样的，用同一个模块实现。

（3）ALU 控制器

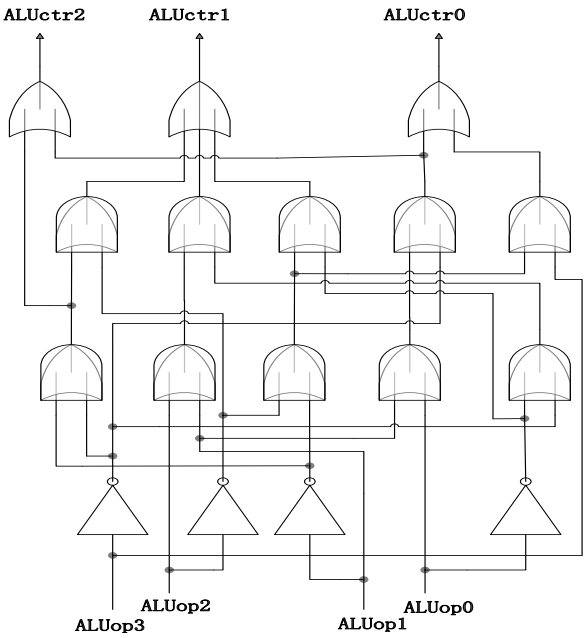


图 3 ALU 控制器逻辑图

说明：

ALUctr2 = $\overline{ALUop3}\ \overline{ALUop1}$ + $\overline{ALUop3}$ ALUop2 ALUop0

ALUctr1 = $\overline{ALUop3}\ \overline{ALUop2}\ \overline{ALUop1}$ + $\overline{ALUop3}$ ALUop2 ALUop1 $\overline{ALUop0}$ + $\overline{ALUop2}\ \overline{ALUop1}\ \overline{ALUop0}$

ALUctr0 = ALUop3 $\overline{ALUop2}\ \overline{ALUop1}$ + $\overline{ALUop3}$ ALUop2 ALUop0

（4）多路选择器 mux_7by1

多路选择器模块，利用 3 位控制信号，从 7 个 32 位的输入数据中，选择出其中一个作为输出。

（5）位扩展，extend1_32

位扩展模块，将一位输入数据扩展为 32 位输出数据，并用 ari_log 控制逻辑扩展还是算数扩展。

（6）其他模块

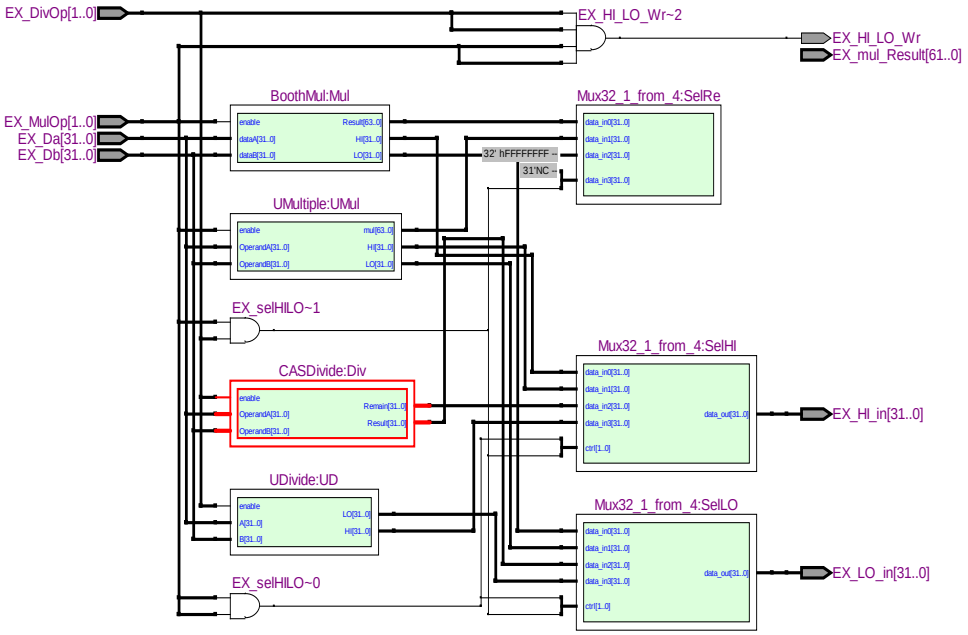
如 and32，or32，not32，xor32 这类逻辑操作，直接用逻辑语句实现，而 extend6_32，extend1_32_alg 位扩展模块，与 extend1_32 原理相同，由于较为简单，这里不再详述。

3 在流水线 CPU 数据通路中对 ALU 控制信号的编码

ALU 实现与单周期、多周期还是流水线 CPU 没有关系，故一直沿用，未作修改。然而，在整个流水线 CPU 中的控制逻辑部件中，需要一级编码，根据指令的 op 等信号得到 ALUop，进而根据 ALUop 和 func 字段获得 ALUctr，也就是 ALU 部件内部的 ALUop 控制信号。具体编码将在下面的流水线 CPU 控制逻辑中详细说明。

4.5.3 乘法器和除法器的设计和实现

乘法除法器部件中封装了带符号乘法器：BoothMul；不带符号的乘法器：UMultiple；带符号的除法器：CASDivide；不带符号的除法器：UDivide



乘法器

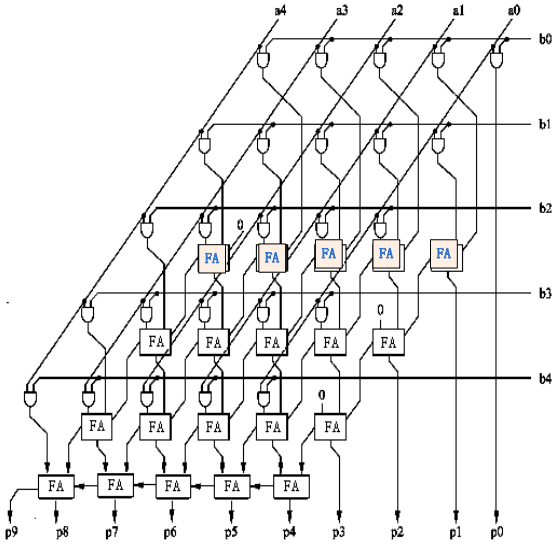
在 ALU 实现的功能中，还有无符号乘法和有符号乘法两项。无符号乘法采用了并行乘法器树实现。有符号乘法可采用 booth 乘法器实现，booth 乘法器的实现我们是参照网上的示例，下面仅就无符号乘法的实现予以探讨。

虽然可以采用迭代加法的方法计算无符号乘法，但耗时相当严重，为了将运算并行化，只需要把每位乘出来的结果错位加起来。如下图所示的一个五位乘法运算式。

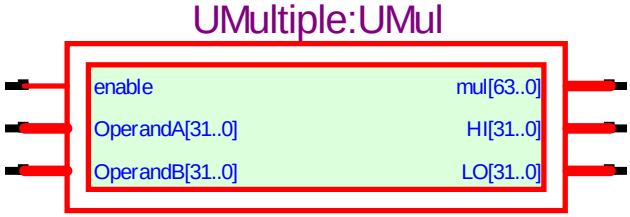
					a4	a3	a2	a1	a0
					× b4	b3	b2	b1	b0
					a4b0	a3b0	a2b0	a1b0	a0b0
				a4b1	a3b1	a2b1	a1b1	a0b1	
			a4b2	a3b2	a2b2	a1b2	a0b2		
		a4b3	a3b3	a2b3	a1b3	a0b3			
+	a4b4	a3b4	a2b4	a1b4	a0b4				
p9	p8	p7	p6	p5	p4	p3	p2	p1	p0

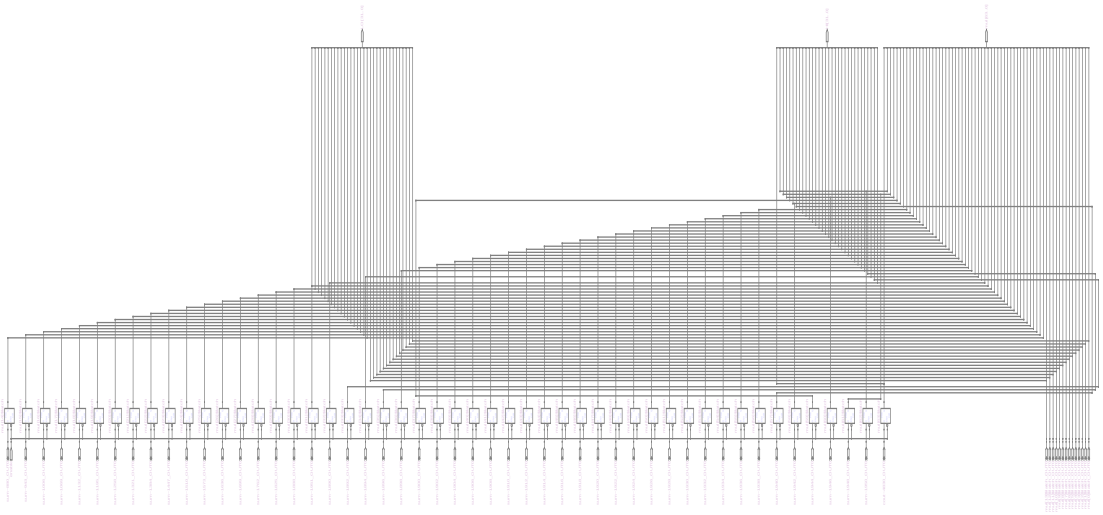
乘法运算图

根据乘法的运算形式，可以设计一种加法阵列来实现乘法运算，由于是二进制运算，相乘环节可以理解为“与”运算，相加环节可以采用全加器网络实现。



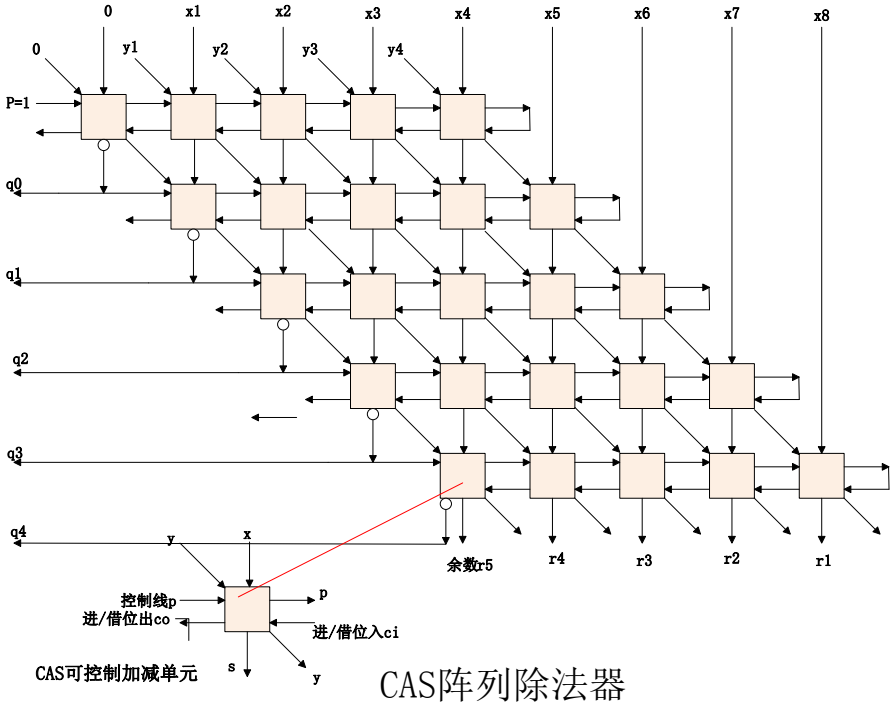
并行乘法器树





除法器：

除法器，我们的实现有无符号的 32 位除法器和带符号的除法器，其中带符号的除法器，我们参照了刘文慧设计的 CAS 阵列除法器，CAS 阵列除法器的效率较高，不恢复余数算法实现的阵列除法器。



CAS阵列除法器

CAS 逻辑表达式：

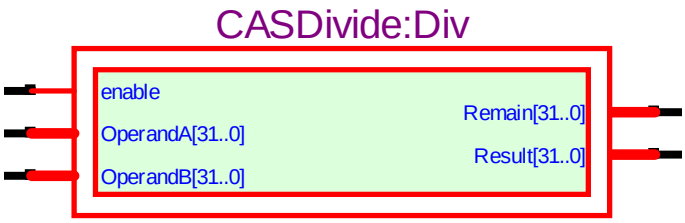
s=x^(p^y)^ci

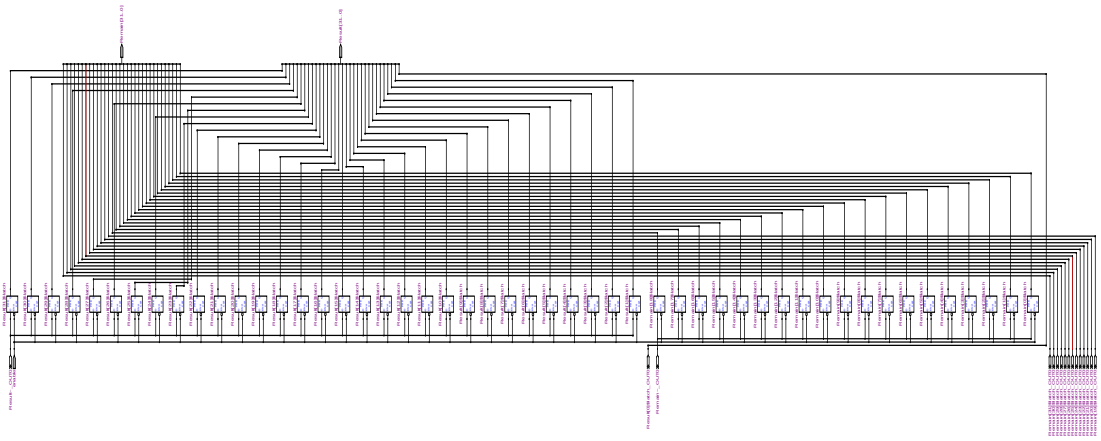
co=(x+ci)(p^y)+xci

当 p=0 时，s=x^y^ci ，ci=xy+yci+xci

当 p=1 时，s=x^(!y)^ci ，ci=x(!y)+(!yci)+xci

此外，所有数据的计算都是通过将数据取绝对值之后进行除法计算，最后根据数据的实际符号对商和余数进行修正，使商的符号与被除数的符号一致，余数的符号由被除数和除数的符号共同决定。



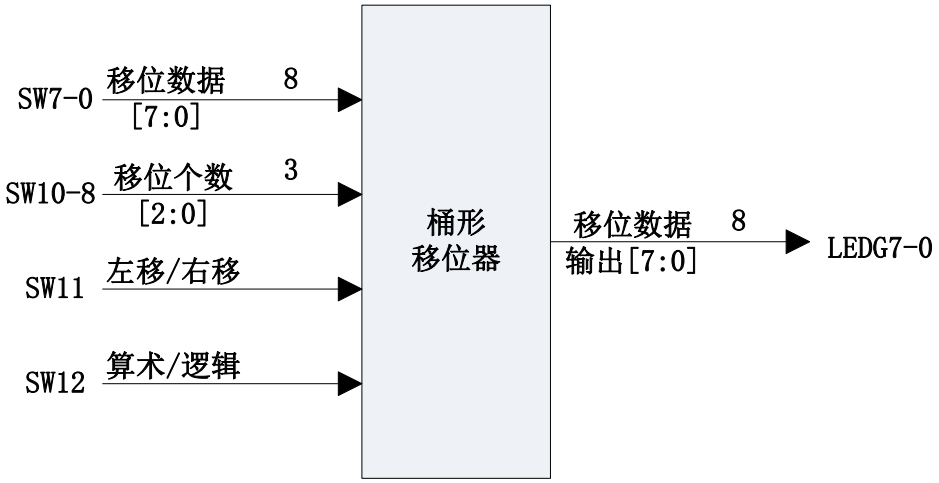


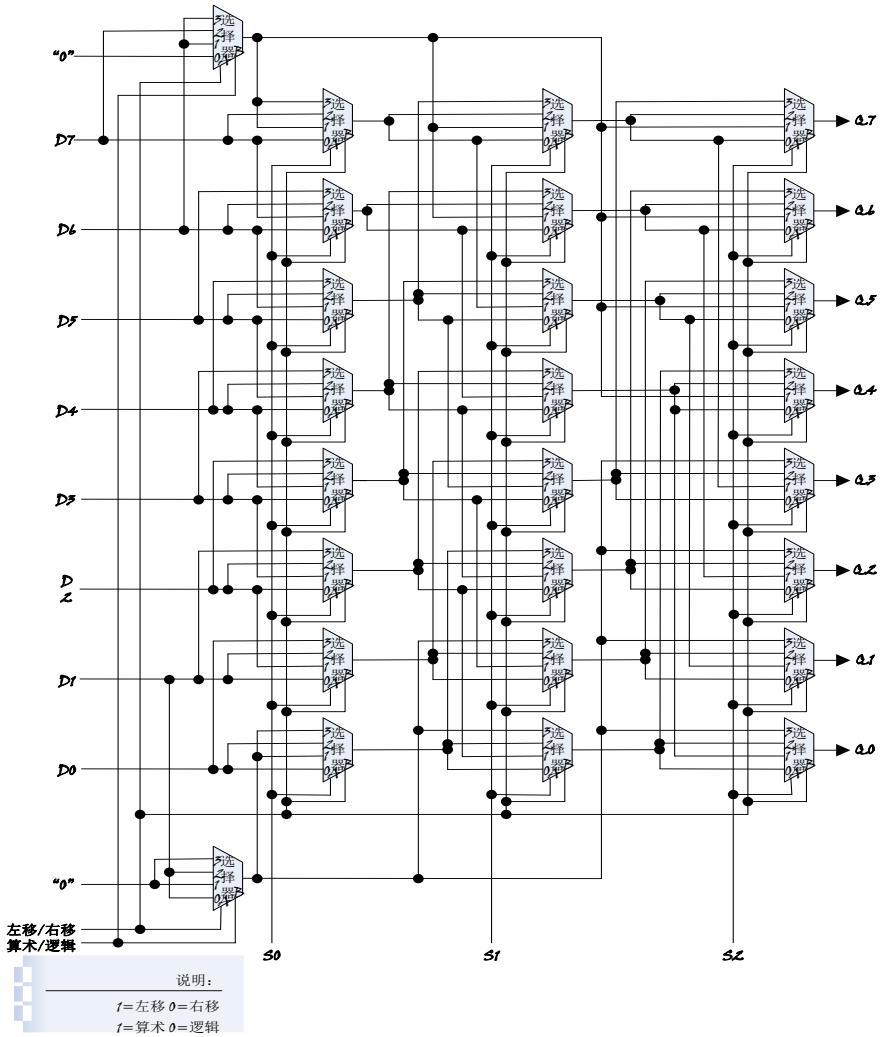
4.5.4 桶形移位器的设计和实现

1 功能

桶形移位器在一次操作中可以移动任意位数。输入为要移位的数据，移位个数，左移/

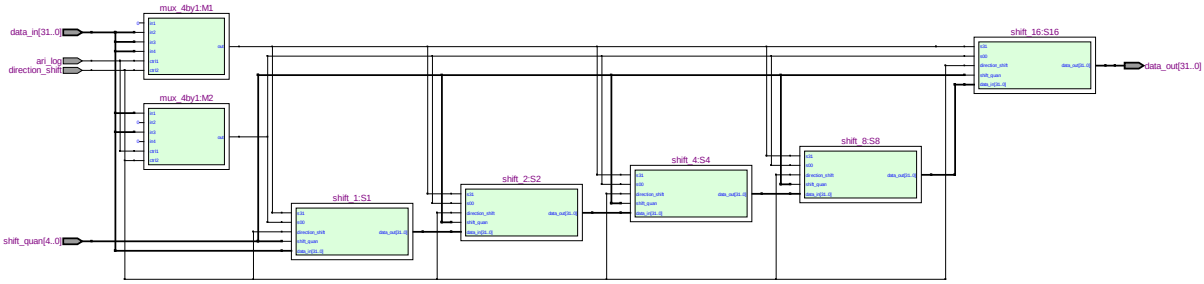
右移控制，算术/逻辑移位控制。输出为得到的结果。我们用多个多路选择器即可搭建起一个桶形移位器，下图就是 8 位桶型移位器的逻辑电路图。





我们实现的桶形移位由以下几个模块组成：

Shift_1bit :移 1 位模块 ,**Shift_2bit** :移 2 位模块 ,**Shift_4bit** :移 4 位模块 ,**Shift_8bit** : 移 8 位模块，**Shift_16bit**：移 16 位模块



4.5.5 指令和数据存储器的设计和实现

指令寄存器 IM（Instruction Memory）

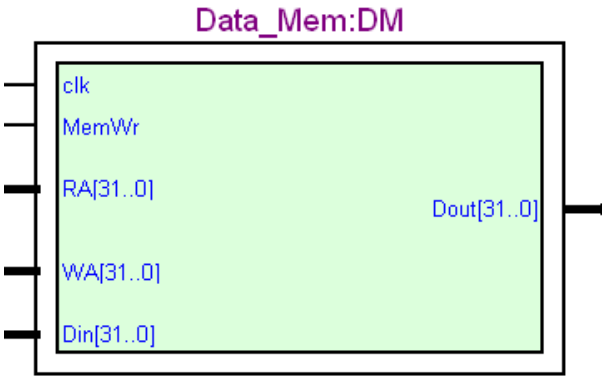
1、　功能：

指令寄存器是由 256 个 8 位的寄存器构成的寄存器组。根据 PC 的值读出指令。它是一个只读型寄存器。在程序内部事先定义好了所有要用来测试的指令，在初始化时予以定义。具体测试指令相关，参看测试指令部分。

2、　流水线的特殊点：

由于在 IFetch 阶段时钟到来后拿出 PC 值，然后直接读出指令，将指令和 PC+4 的值送入 IF/ID 寄存器。这整个过程是由一开始取 PC 的那个时钟控制的，而指令寄存器是立即读出的，因此不需要时钟的控制。

3、　RTL 图展示



数据寄存器 DM（Data Memory）

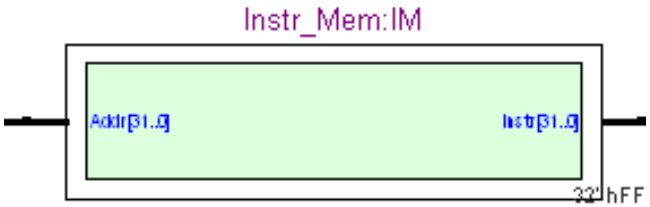
1、 功能：

这是一个由 256 个 8 位的寄存器构成的寄存器组。在时钟的控制下，当上升沿到来时，如果当前 MemWr 为低电平，表明写使能有效，数据被写入由 WA 所指定地址的寄存器中，在后半周期可以进行数据的读出。

2、 流水线的特殊点：

这里要注意的是，因为是流水线 CPU，因此如果前一条指令是 Load 指令，正在进行取数据的操作，而这个时候后面的某条指令到来，正要进行取指令操作，那么则发生冲突，这也就是为什么我们这里要把数据寄存器和指令寄存器分开来的原因。对于单周期和多周期来说，两者不分开来也是可行的，对于 lwl 及 lwr 指令得到的指令地址可能不是 4 的整数倍，所以需要先对指令地址处理一下，然后再进行读写操作，这里我们先取出地址的前 31 到 2 位，然后再拼上后两位 0。

3、 RTL 图展示

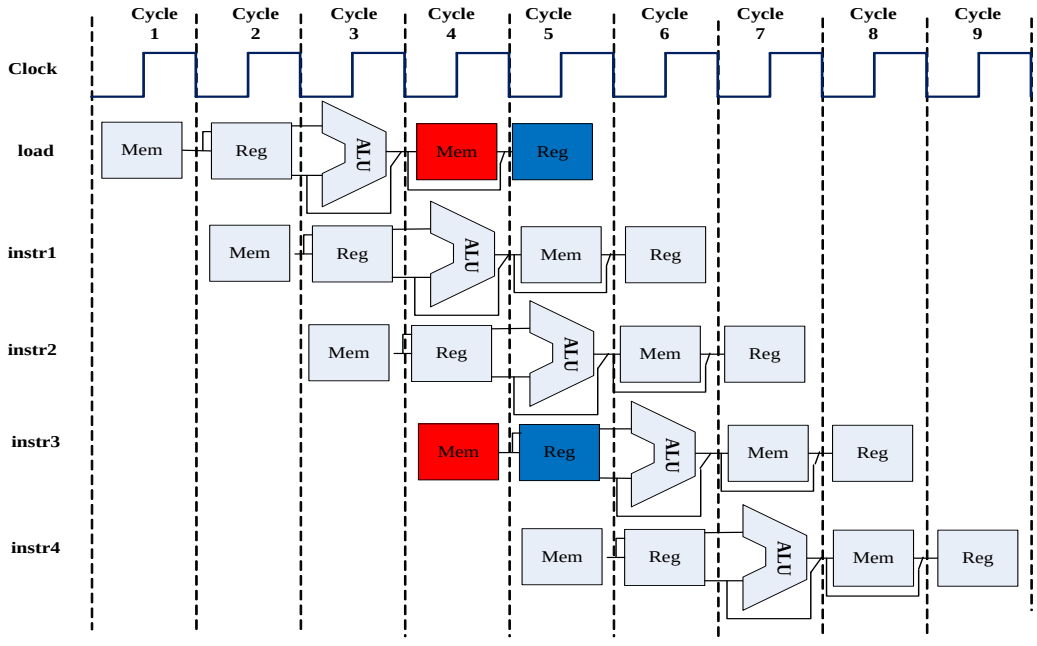


4.6 流水线 CPU 的冲突冒险问题及解决方案

流水线的冒险主要分为三种：结构冒险、数据冒险和控制冒险。对于控制冒险，由于比较复杂，需要引入静态或者动态预测，我们没有在数据通路中实现。

4.6.1 结构冒险

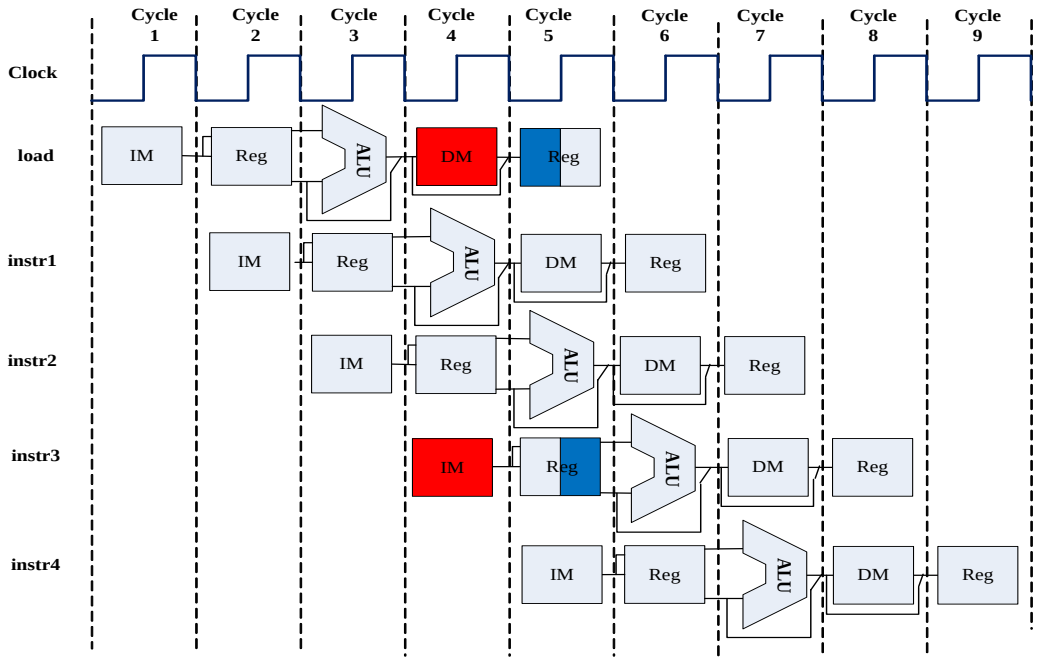
结构冒险也称为硬件资源冲突，引起结构冒险的原因在于同一个部件同时被不同指令所用。



结构冒险解决方案

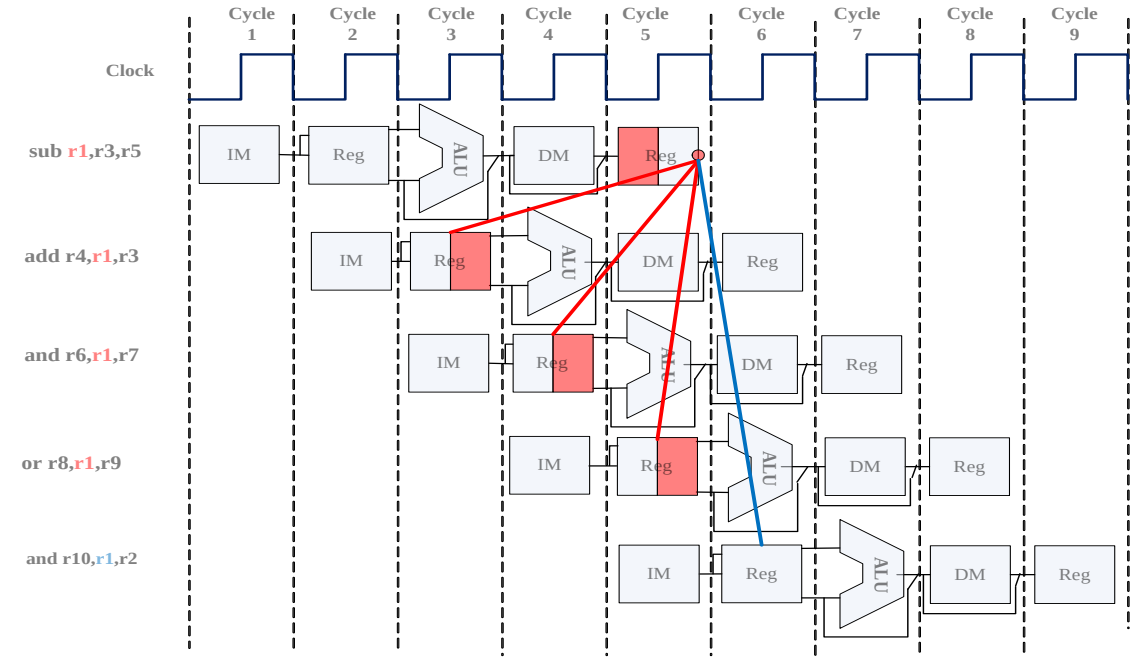
在实现中，我们将寄存器读口和写口独立开来，分别利用时钟下降沿和上升沿两次触发；

对于存储器访存冲突，我们采用把指令存储器 IM 和数据存储器 DM 分开达到消除结构冲突的目的。



4.6.2 数据冒险

数据冒险也称为数据相关。引起数据冒险的原因在于后面指令用到前面指令结果时前面指令结果还没产生。

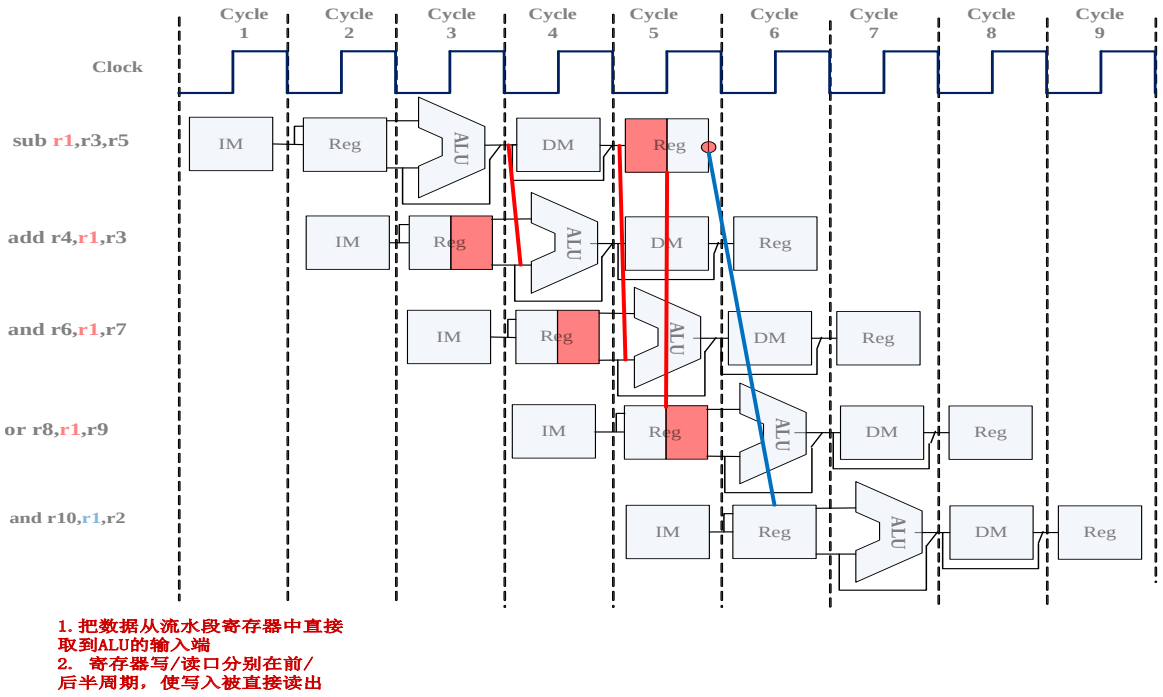


只有最后一条的r1是新值

数据冒险解决方案

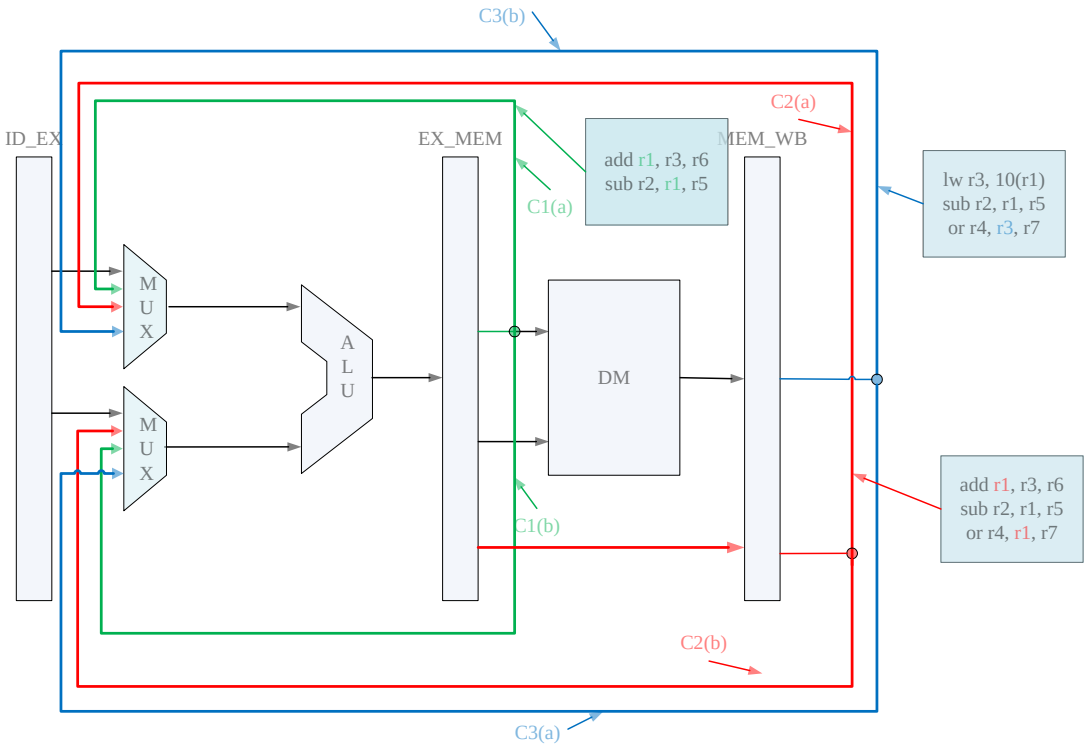
关于数据冒险的解决方法有很多种，比如说可以在硬件上通过阻塞(stall)方式阻止后续指令执行，延迟到有新值以后，这种做法称为流水线阻塞，也称为“插入气泡（Bubble）”，但是这种做法控制相当复杂，需要改数据通路；此外，还可以用软件的方法来解决，即由编译器插入三条空指令，这是最差的解决方法，因为这意味着浪费三条指令的空间和时间。

转发



我们观察发现，在流水段寄存器中有需要的值。以上图为例，第一条指令实际上在 Exec 阶段结束时就已经得到 r1 的值了，那么，我们可以直接从 Ex_Mem 段寄存器中取到 r1 的新值，然后送到第二条指令的 ALU 输入端；同样，我们可以在 Mem_WB 段寄存器中取得 r1 的新值，送入第三条指令的 ALU 输入端。对于第四条指令，上面所说的解决结构冒险的方法实际上已经解决这里的问题。通过将寄存器写口和读口分别控制在前后两个半周期内，使得前半周期刚被写入的值在后半周期读出来。这种技术成为转发或旁路技术。

那么，如何实现转发呢？我们需要对流水线 CPU 的结构做一些调整，在 ALU 输入端增加多路选择器。如下图所示：



这里 C1 反映的是本条指令和随后指令间的数据相关，C2 反映的是本条指令和随后第二条指令间的数据相关，而 C3 则是反映 lw 指令与随后第二条指令之间的数据相关。在实现中，我们把 C1(a)和 C1(b)合并成一个条件 C1，并把转发线合在一起同时送入 ALU 输入端 A 口和 B 口(ForwardA = {C1_a, C2_a})。同样 ,把 C2(a)和 C2(b)合起来(ForwardB = {C1_b, C2_b})。值得注意的是，这里与书上说的合并并不是一个意思。书上 C1=C1(a)|C1(b)，而我们则是拼接到一起。具体理由在心得里有叙述。另外，我们设计了一个二路选择器，将 C2 和 C3 合并在一起输出到寄存器堆。

那么转发的条件是什么呢？我们最容易想到的是：

C1(a)=Mem_Rw == EX_Rs
C1(b)=Mem_Rw == EX_Rt
C2(a)=WB_Rw == EX_Rs
C2(b)=WB_Rw == EX_Rt

但是以下两种情况下，根据前面的转发条件转发会发生错误：

（1） 指令的结果不写入目的寄存器 Rd 时

即：EX_MEM 或 MEM_WB 流水段寄存器的 RegWrite 信号为 0。

例如：Beq 指令只对 rs 和 rt 相减，但不写结果到目的寄存器

（2） Rd 等于$0 时

例如：指令 sllv &0, $1, 2 的转发结果可能为非 0，但实际上应该是 0

于是，修改条件为

```
C1(a) = ( (!Mem_RegWr) && (Mem_Rw != 5'b0) && (Mem_Rw == EX_Rs ));
C1(b) = ( (!Mem_RegWr) && (Mem_Rw != 5'b0) && (Mem_Rw == EX_Rt ));
C2(a) = ( (!WB_RegWr) && (WB_Rw != 5'b0) && (WB_Rw == EX_Rs));
C2(b) = ( (!WB_RegWr) && (WB_Rw != 5'b0) && (WB_Rw == EX_Rt) );
```

考虑以下指令序列，采用前面转发条件会怎么样？

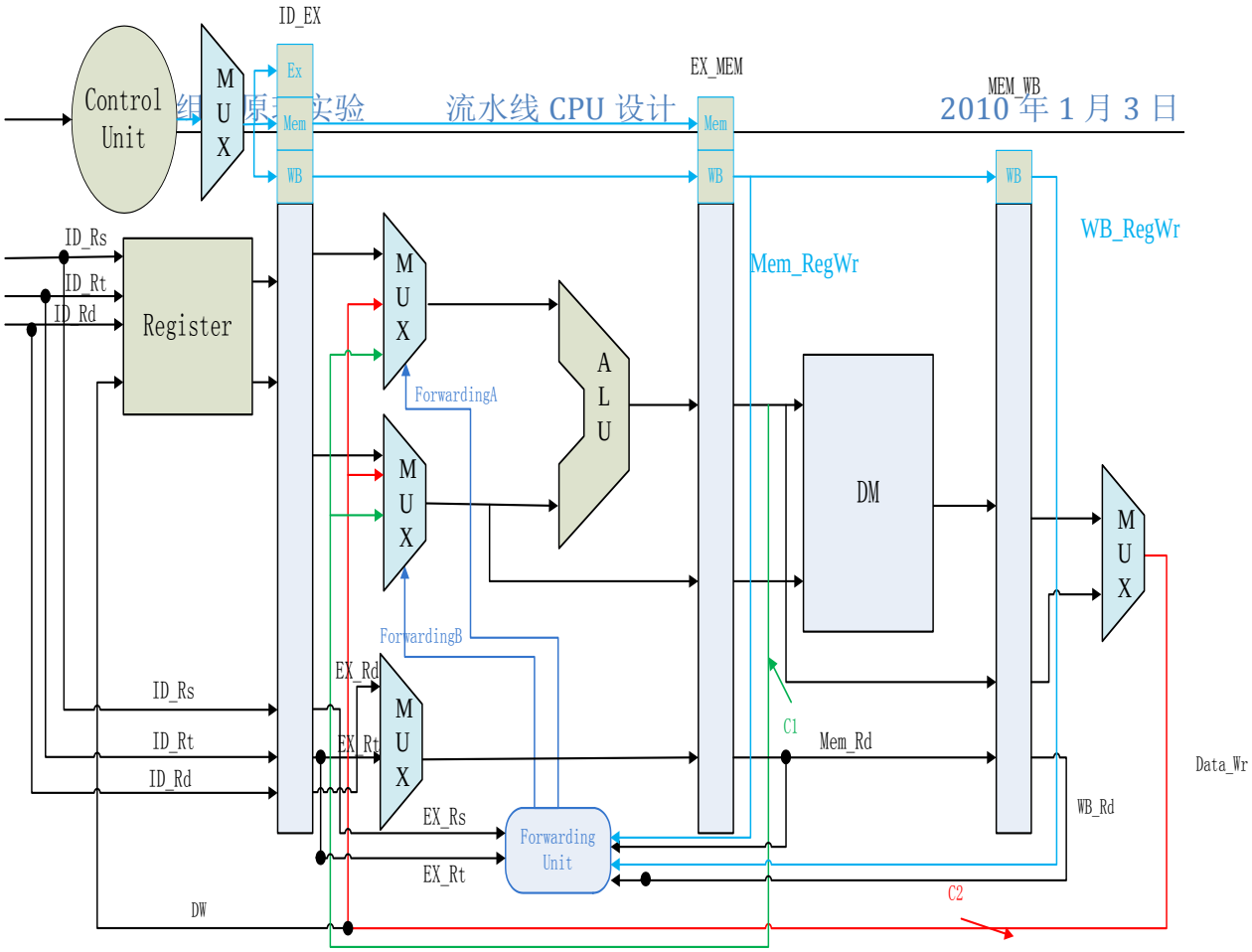
```
Add $1, $1, $2
Add $1, $1, $3
Add $1, $1, $4
```

对于第三条指令，由于 Forward 信号取值不确定，可能使转发到第三条指令的操作数是第一条指令的结果，而实际上应该是第二条指令的结果。再次修改转发条件：

```
C1(a) = ( (!Mem_RegWr) && (Mem_Rw != 5'b0) && (Mem_Rw == EX_Rs ));
C1(b) = ( (!Mem_RegWr) && (Mem_Rw != 5'b0) && (Mem_Rw == EX_Rt ) &&(!MemtoReg));
C2(a) = ( (!WB_RegWr) && (WB_Rw != 5'b0) && ((Mem_Rw != EX_Rs)) && (WB_Rw == EX_Rs));
C2(b) = ( (!WB_RegWr) && (WB_Rw != 5'b0) && ((Mem_Rw != EX_Rt)) && (WB_Rw == EX_Rt)
&&(!MemtoReg));
```

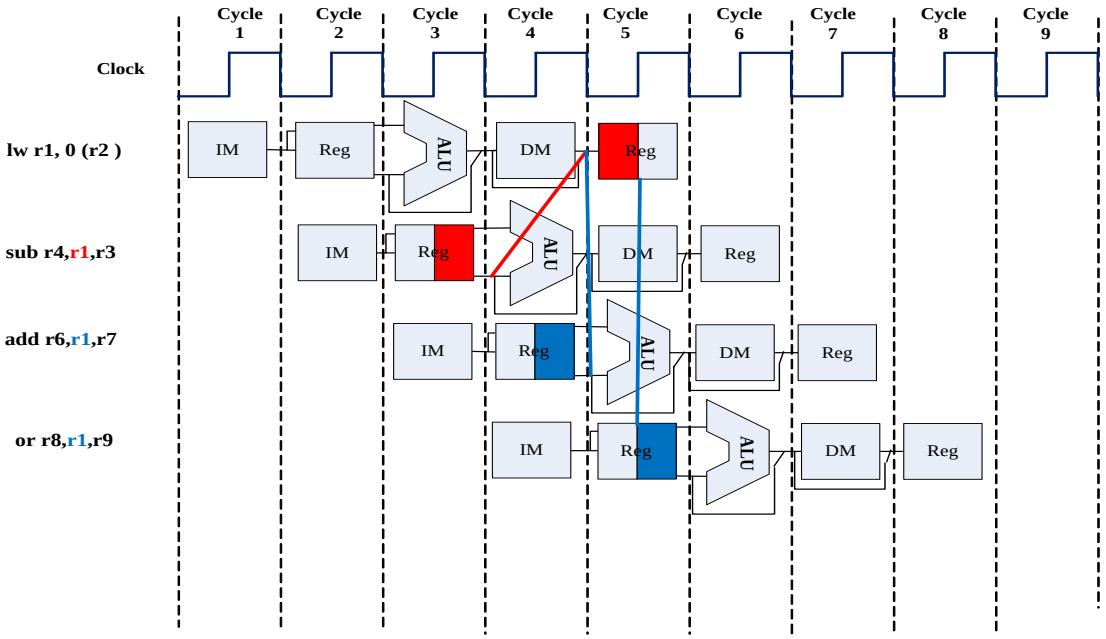
对于 C2 修改相当于加了一条条件限制：如果本条指令源操作数和上条指令的目的寄存器一样，则不转发上上条指令的结果，转发上条指令的结果。

对于转发的实现，参见 CPU 带转发数据通路图。



load-use

然而如果如下图所示的情况，转发并不能解决第一条指令和第二条指令之间的数据相关。



lw 指令只有在 Mem 段结束后才能读到 DM 中的数据，也就是要写到寄存器中的数据，

然后送入 Mem_WB 寄存器。在 Wr 段结束后，r1 中才能存入新值，因此随后的 sub 指令在 Ex 段无法取到 r1 的新值。而根据前述的数据转发线路，ALU 的输入端要么来自上条指令在 Ex 段生成的、存放于 Ex_Mem 寄存器中的值，要么来自上上条指令的执行结果。所以上述转发线路无法解决上图中 lw 指令和随后的 sub 指令间的数据相关问题。通常把这种情况称为“load-use”数据冒险。这种情况，必须要阻塞一个周期。

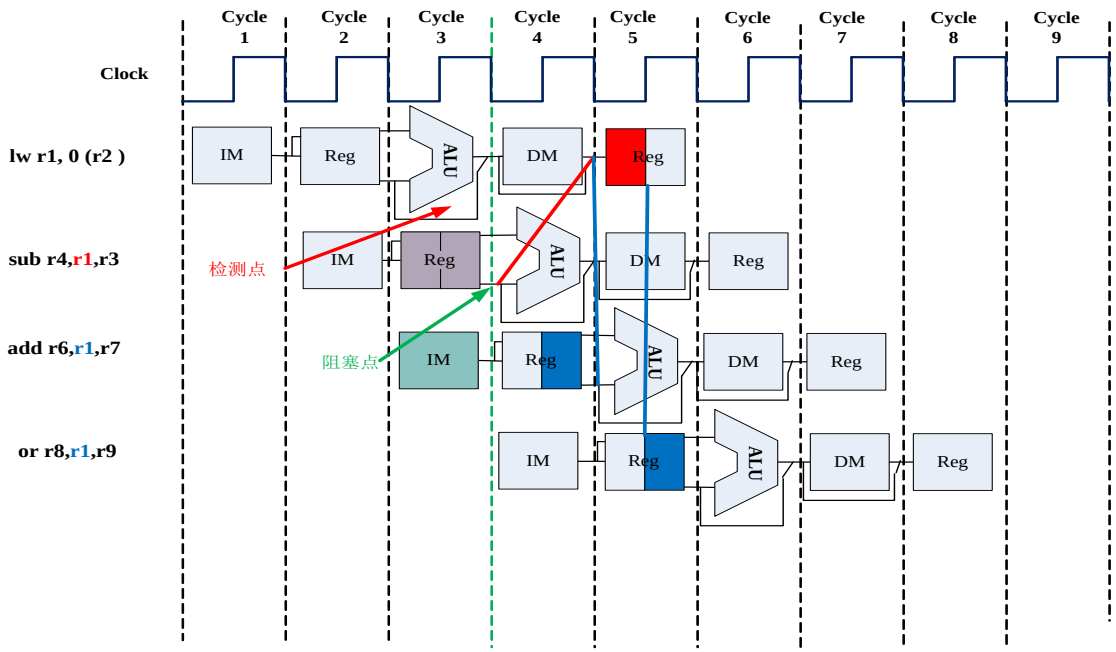
通常我们有三种方法来解决 load-use。第一种是用硬件阻塞一个周期，指令被重复执行一次，也就是我们在实现中使用的方法；第二种是用软件插入一条空指令；第三种方法是利用编译器对指令顺序进行调整来解决 load-use。

首先什么条件下需要阻塞呢？(EX_IsLoad)&& (EX_Rt == ID_Rs || EX_Rt == ID_Rt)，也就是前面指令为 Load 并且前面指令的目的寄存器等于当前刚取出指令的源寄存器。

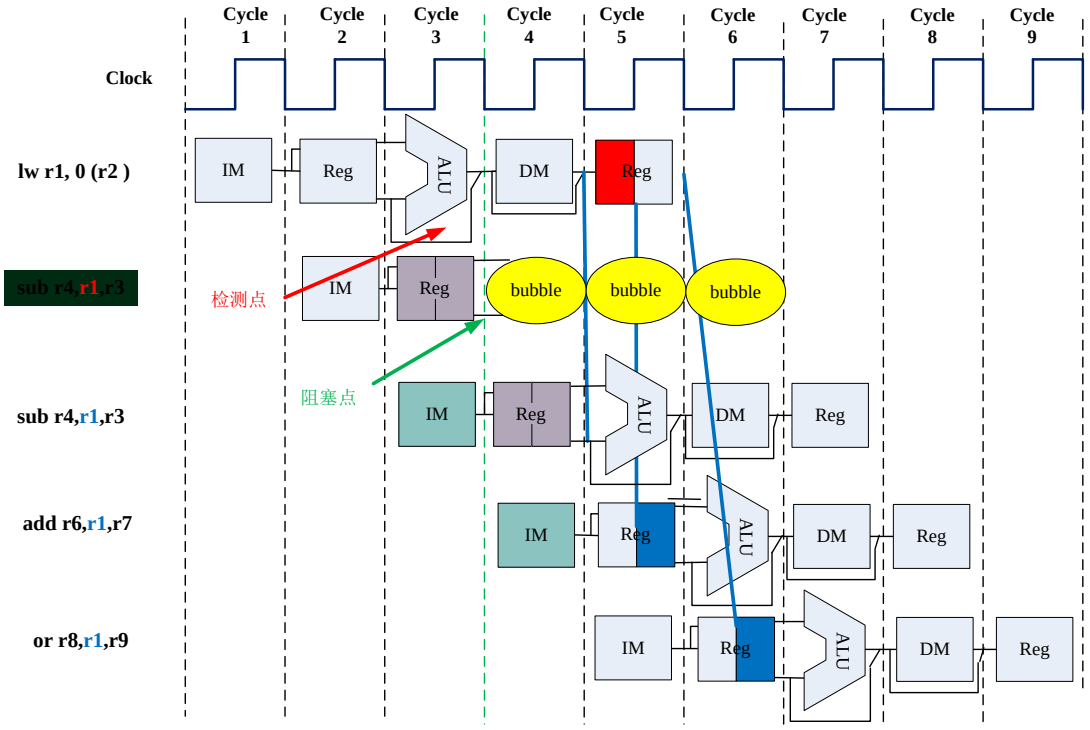
检测“阻塞”过程中：

1 sub 指令在IF_ID 段寄存器中，并正被译码/取数，控制信号和 Rs/Rt 的值将被写到ID_EX 段寄存器

2 and 指令地址在 PC 中，正被取出，取出的指令将被写到 IF_ID 段寄存器中



在阻塞点，必须将上述两条指令的执行结果清除，并延迟一个周期执行这两条指令。阻塞一个时钟周期在执行相应指令的情况如下图所示：



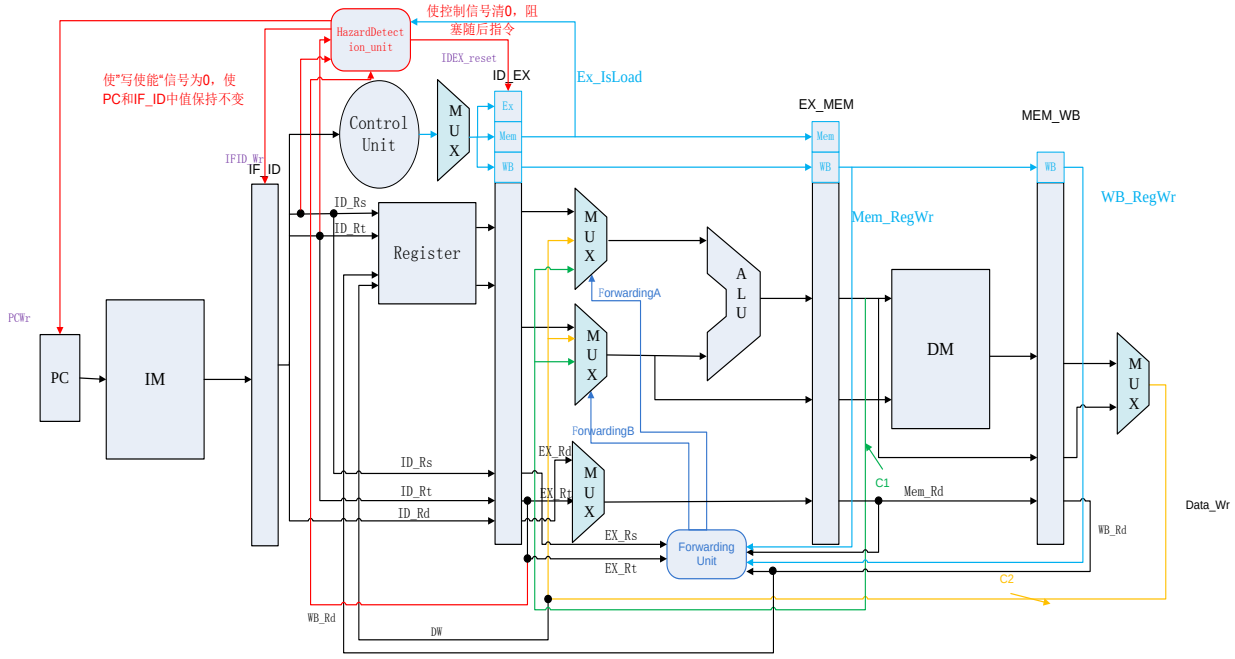
在阻塞点上要做的操作有：

1 将 ID_EX 段寄存器中所有控制信号清 0

2 IF_ID 段寄存器中的信息不变，sub 指令重新译码执行

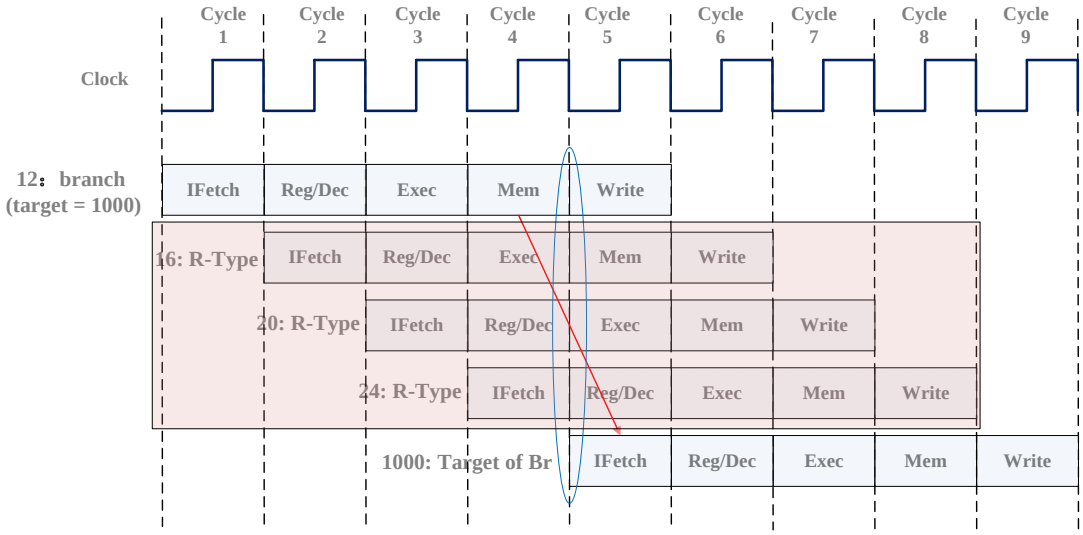
3 PC 中的值不变，and 指令重新被取出执行

关于 load-use 数据冒险阻塞的实现，参见带转发和阻塞的数据通路图。



4.6.3 控制冒险

引起控制冒险的原因是由于分支（条件转移）指令或异常而改变程序执行流程，使得流水线被阻塞。假设 Branch 指令在第一周期被取出，但是目标地址要到第四个周期才被送到 PC 输入端，第五个周期才能取出目标地址处的指令执行。而在取目标指令之前，已经有三条指令被取出，也就是说，取错了三条指令。



控制冒险的解决方法主要有四个：

方法 1：硬件上阻塞（stall）分支指令后三条指令的执行

使后面三条指令清 0 或其操作信号清 0，以插入三条 NOP 指令

方法 2：软件上插入三条“NOP”指令

（以上两种方法的效率太低，需结合分支预测进行）

方法 3：分支预测（Predict）

简单（静态）预测：

- 总是预测条件不满足(not taken)，即：继续执行分支指令的后续指令可加启发式规则：在特定情况下总是预测满足(taken)，其他情况总是预测不满足。如：循环顶（底）部分支总是预测为不满足（满足）。能达 65%-85%的预测准确率

动态预测：

- 根据程序执行的历史情况，进行动态预测调整，能达 90%的预测准确率

注：采用分支预测方式时，流水线控制必须确保错误预测指令的执行结果不能生效，而且要能从正确的分支地址处重新启动流水线工作

方法 4：延迟分支（Delayed branch）（通过编译程序优化指令顺序！）

把分支指令前面与分支指令无关的指令调到分支指令后面执行，也称延迟转移。

本实验中，因为时间等原因，没有实现通过预测的分支指令的冒险。

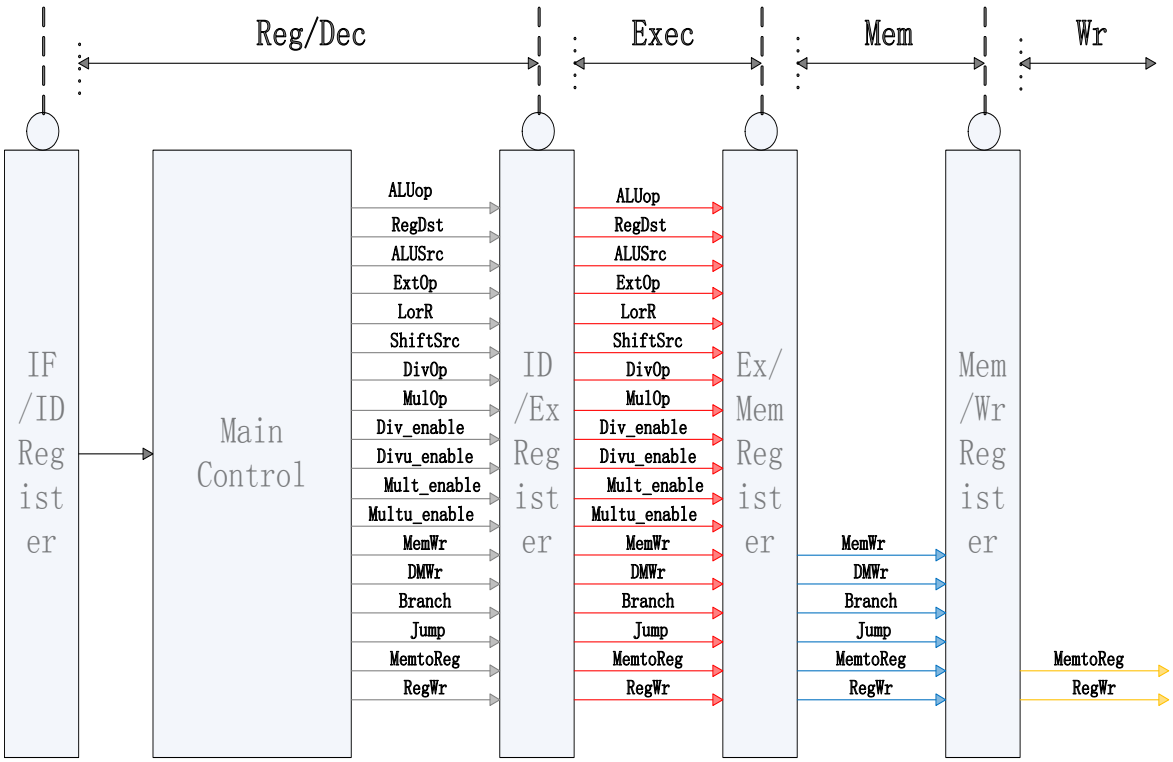
4.7 流水线 CPU 的控制逻辑的实现

4.7.1 基本的流水线控制

如前面说过，流水线 CPU 中对于一条指令来说，在它的生命周期中对应的控制信号是不变的。这一点类似于单周期 CPU 的控制逻辑。

Ifecth 阶段和 Dec/Reg 阶段不需要控制信号，因为每条指令所执行的功能都一样，是确定的操作，无需根据指令的不同来控制执行不同的操作。

流水线 CPU 是在 Reg/Dec 阶段由控制器产生指令各流水段的所有控制信号，并存放 ID/EX 流水段寄存器中，在分别在随后的各个时钟周期内被使用。Exec 信号(ALUop, RegDst, ALUSrc, ExtOp, LorR, ShiftSrc, ShiftOp, DivOp, MulOp, div_enable, divu_enable, mult_enable, multu_enable)在 1 个周期后使用，Mem 信号(MemWr, DMWr, Branch, Jump)在 2 个周期后使用，Wr 信号(MemtoReg, RegWr)在 3 个周期后使用。



某一时刻每个流水线执行的是不同指令的某个阶段，因而某一时刻每个流水段中的控制信号应该是正在执行指令的对应功能段的控制信号。所以我们在实现中每一个控制信号前都加上对应阶段的缩写 ,如在取址译码阶段的 ALU 控制信号写作 ID_ALUop ,执行阶段的 ALU 控制信号 ,就是 Ex_ALUop。各个流水段部件在一个时钟内完成某条指令的某个阶段的工作，然后，在下个时钟到达时，把执行的结果以及前面传递过来的后面各阶段要用到的所有数据和控制信号保留到流水线寄存器中。

在设计中，控制逻辑分为两个部分：

——主控制逻辑，输入为指令的 op 和 func 字段，输出为包括 ALUop 的各种控制信号。

——局部 ALU 控制逻辑，输入为 ALUop，输出为 ALUctr（这里的 ALUctr 就是 ALU 部件内部的输入控制信号 ALUop）

下面详细谈一下对主控制逻辑的输出 ALUOp 的编码：

（1）ALU 控制信号的一级编码：

在本实验中ALUop取值的情况，可以用4位数。但是考虑到以后可能会对指令集进行拓展，又因为指令中op字段为6位，所以我在实现时将ALUop编码为6位，这样排除了以后位数不够的后顾之忧。

R型指令除去前导零/一的计算，ALUop编码相同，前导零/一计算编码相同，其他指令对应不同ALUop编码。为了区分R型指令和非R型指令，以便以后确定ALUctr的编码，我将R型指令的ALUop的最高位都设为1，非R型的都为0。其他位在原则上只要赋予不同的编码即可，但是在实际操作构成中还涉及一些编码技巧，将在下面ALUctr编码中一起提到。

（2）ALU 控制信号的二级编码:

在之前的实验中，已经实现了ALU部件，所以在本实验中可以直接利用，当这要求我们对控制信号进行适当的修改。这里的ALUctr输入ALU部件，在之前实现的ALU中，我定义了一个ALU内部的控制信号模块，这里输入的ALUctr相当于当时模块中的”ALUop”。所以确定ALUctr为4位，并且在编码中，对ALUctr有些特殊的要求。考虑到在ALU实现中，输入的ALUctr[0]决定了位数扩展的类型，加法减法运算的选择，ALUctr[1]决定less的输出。所以，在加法，前导零中ALUctr[0]要为0，在减法和前导一，slt中ALUctr[0]要为1。在涉及无符号操作时的ALUctr[1]要为1，有符号为ALUctr[1]=0。

在确定了ALUctr的编码基础上，对于非R型指令，为了简便，可以让ALUctr等于ALUop的第四位的值，而对于R型指令，这要通过func和ALUop的真值表得到ALUctr。

具体编码表见下表：

MIPS 指令	op 字段	Func 字段	ALU 操作	ALUop	ALUctr
add	000000	100000	add	100000	0000

addu	000000	100001	add	100000	0010
sub	000000	100010	sub	100000	0001
subu	000000	100011	sub	100000	0011
nor	000000	100111	nor	100000	0111
clo	011100	100001	clo	100001	0101
clz	011100	100000	clz	100001	0100
slt	000000	101010	slt	100000	1101
sltu	000000	101011	sltu	100000	1111
addi	001000	*	add	000000	0000
addiu	001001	*	add	000010	0010
xori	001110	*	xor	001000	1000
slti	001010	*	slt	001101	1101
sltiu	001011	*	sltu	001111	1111
blez	000110	*	sub	000001	0001
j	000010	*	*	000111	*

所有控制信号的编码：

注：LA :Low Active

指令	**控制信号**								
	ALUop	**div_enable (LA)**	**divu_enable (LA)**	**mult_enable (LA)**	**multu_enable(LA)**	**ALUsrc(LA)**	**ExtOp (LA)**	**RegDst (LA)**	**ResSel (LA)**
add	0000	1	1	1	1	0	1	1	00
addi	0000	1	1	1	1	1	1	0	00
addiu	0000	1	1	1	1	1	1	0	00
addu	0000	1	1	1	1	0	1	1	00
sub	0001	1	1	1	1	0	1	1	00
subu	0011	1	1	1	1	0	1	1	00
nor	0101	1	1	1	1	0	1	1	00
xori	0100	1	1	1	1	1	0	0	00
clo	0101	1	1	1	1	0	1	1	00
clz	0100	1	1	1	1	0	1	1	00
slt	1101	1	1	1	1	0	1	1	00
slti	1101	1	1	1	1	1	1	0	00
sltiu	1111	1	1	1	1	1	1	0	00
sltu	1111	1	1	1	1	0	1	1	00
blez	0001	1	1	1	1	0	1	0	00
j	*	1	1	1	1	0	1	0	00
sllv	*	1	1	1	1	0	1	1	11

sra	*	1	1	1	1	0	1	1	01
lw	0000	1	1	1	1	0	1	0	00
lwl	0000	1	1	1	1	0	1	0	00
lwr	0000	1	1	1	1	0	1	0	00
sw	0000	1	1	1	1	0	1	0	00
div	*	0	1	1	1	0	1	1	00
divu	*	1	0	1	1	0	1	1	00
mul	*	1	1	0	1	0	1	1	11
mult	*	1	1	0	1	0	1	1	00
multu	*	1	1	1	0	0	1	1	00
指令	**控制信号**								
	Branch (LA)	**Jump (LA)**	**RegWr (LA)**	**DMWr (LA)**	**MemtoReg**	**ShiftSrc**	**Offset**	**ShiftOp**	**LorR**
add	1	1	0	1	0	0	00	00	01
addi	1	1	0	1	0	0	00	00	01
addiu	1	1	0	1	0	0	01	00	01
addu	1	1	0	1	0	0	01	00	01
sub	1	1	0	1	0	0	10	00	01
subu	1	1	0	1	0	0	11	00	01
nor	1	1	0	1	0	0	11	00	01
xori	1	1	0	1	0	0	*	00	01
clo	1	1	0	1	0	0	01	00	01
clz	1	1	0	1	0	0	00	00	01
slt	1	1	0	1	0	0	10	00	01
slti	1	1	0	1	0	0	*	00	01
sltiu	1	1	0	1	0	0	*	00	01
sltu	1	1	0	1	0	0	11	00	01
blez	0	1	1	1	0	0	*	00	01
j	1	0	1	1	0	0	*	00	01
sllv	1	1	0	1	0	1	00	11	01
sra	1	1	0	1	0	0	11	10	01
lw	1	1	0	1	1	0	*	00	01
lwl	1	1	0	1	1	0	*	00	11
lwr	1	1	0	1	1	0	*	00	00
sw	1	1	1	0	0	0	*	00	01
div	1	1	0	1	0	0	10	00	01
divu	1	1	0	1	0	0	11	00	01
mul	1	1	0	1	0	0	10	00	01
mult	1	1	0	1	0	0	00	00	01
multu	1	1	0	1	0	0	01	00	01

4.7.2 带转发的流水线控制

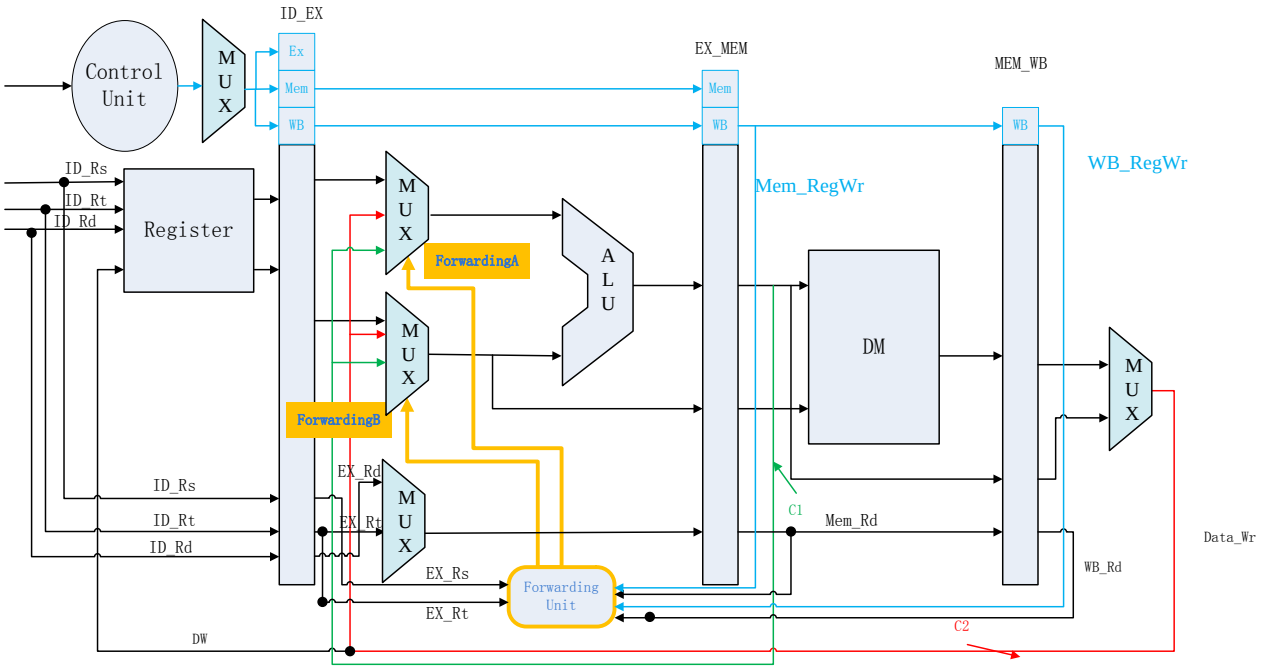
要解决数据冒险，必须考虑带转发逻辑的流水线控制。

在实现中，我们在数据通路中加入了转发单元，用来判断是否会产生数据冒险，进而触发旁路转发功能。

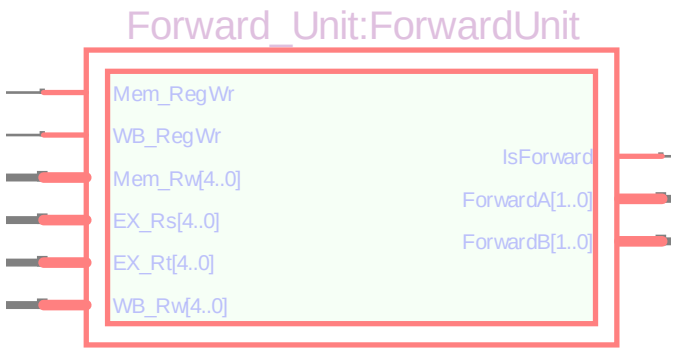
转发的判定逻辑上面已经提到：

```
C1(a) = ( (!Mem_RegWr) && (Mem_Rw != 5'b0) && (Mem_Rw == EX_Rs ));
C1(b) = ( (!Mem_RegWr) && (Mem_Rw != 5'b0) && (Mem_Rw == EX_Rt ));
C2(a) = ( (!WB_RegWr) && (WB_Rw != 5'b0) && ((Mem_Rw != EX_Rs)) && (WB_Rw == EX_Rs) &&(!MemtoReg));
C2(b) = ( (!WB_RegWr) && (WB_Rw != 5'b0) && ((Mem_Rw != EX_Rt)) && (WB_Rw == EX_Rt) &&(!MemtoReg));
```

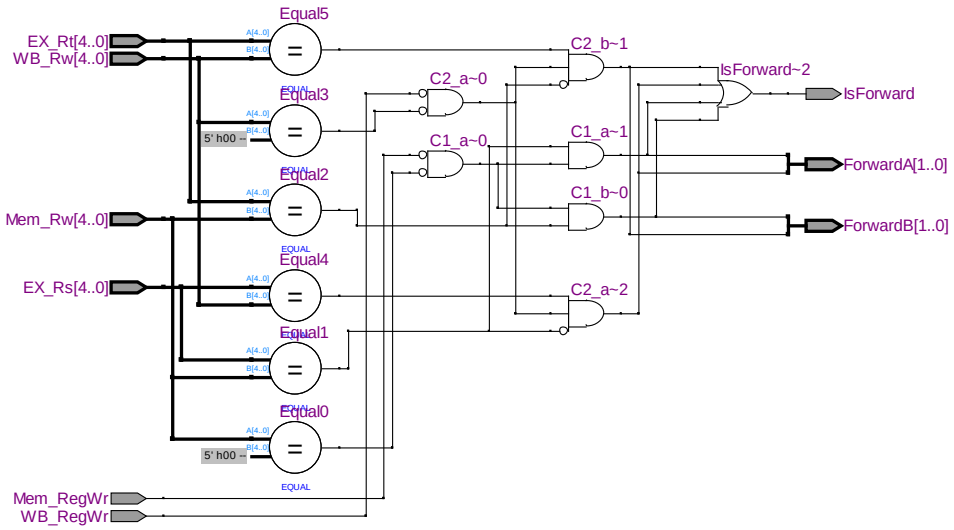
从而产生 ForwardA，forwardB 作为两个多路选择器的控制信号，选择 ALU 的操作数。



转发单元的 RTL 图：

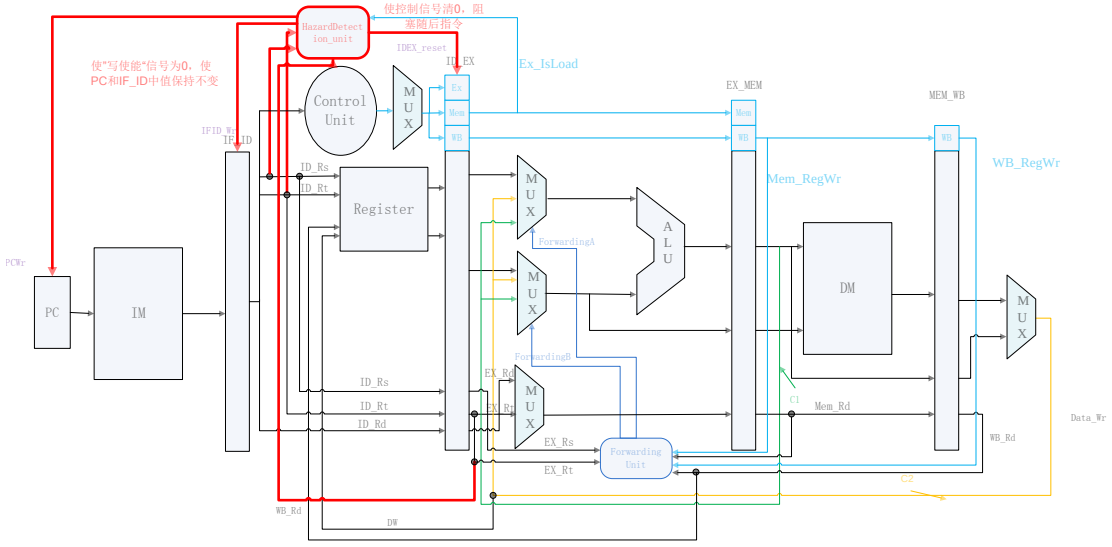


下图是 ForwardingUnit 的具体内部结构：



4.7.3 带冒险检测的流水线控制

为了解决 LoadUse 冒险，加入冒险检测单元。



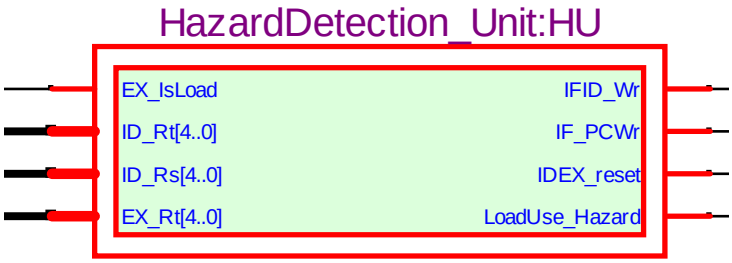
检测是否 LoadUse 冒险的逻辑：

```
assign Hazard =( (EX_IsLoad) && (EX_Rt == ID_Rs || EX_Rt == ID_Rt) );
```

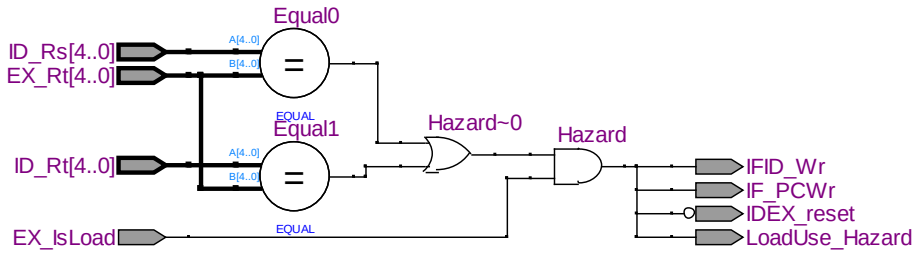
产生 LoadUse 冒险后的相关控制：

```
assign IFID_Wr = (Hazard == 1)? 1'b1:1'b0;
assign IF_PCWr = (Hazard == 1)? 1'b1:1'b0;
assign IDEX_reset = (Hazard == 1)? 1'b0:1'b1;
```

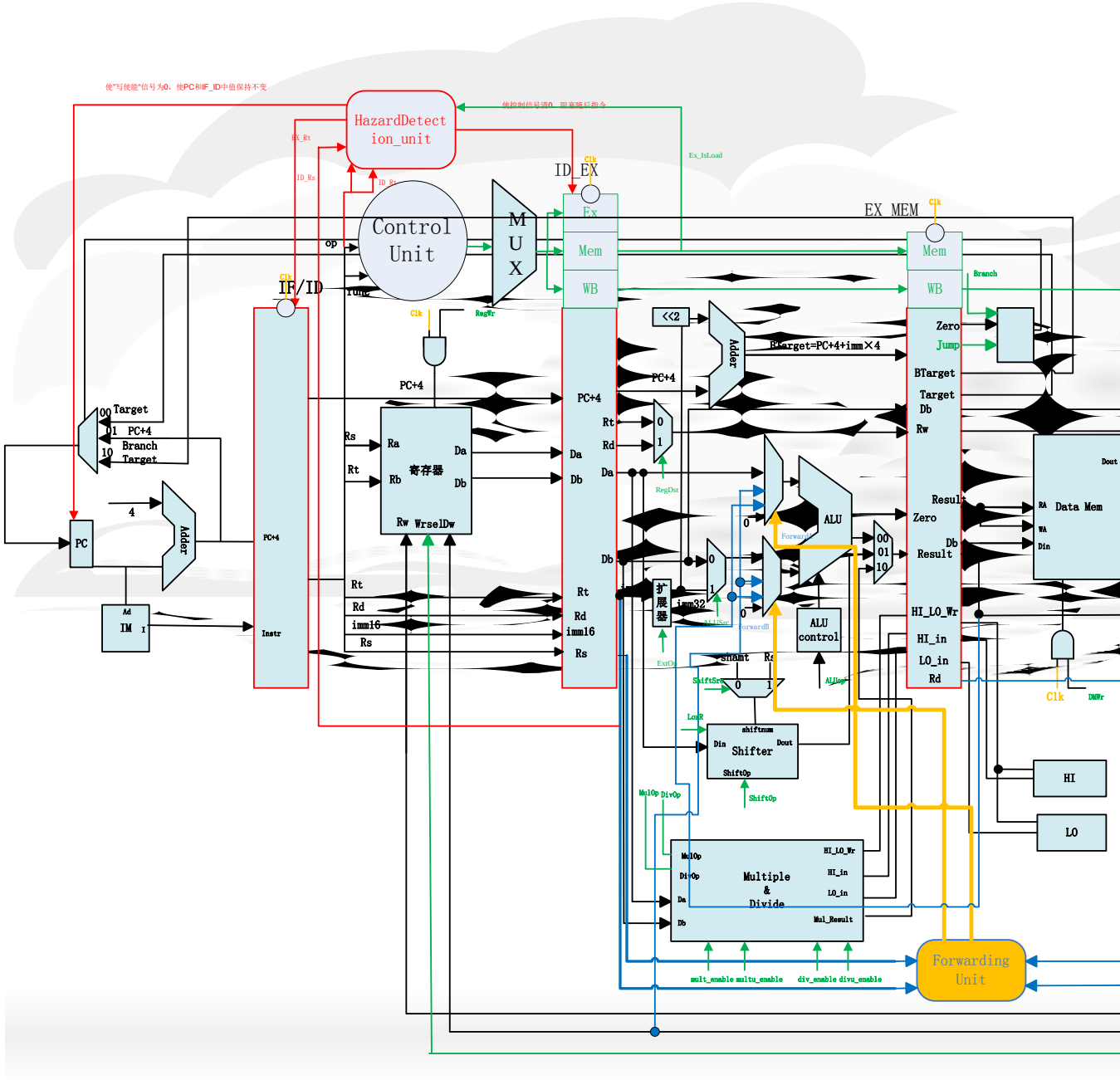
LoadUse 竞争检测单元的 RTL 图：



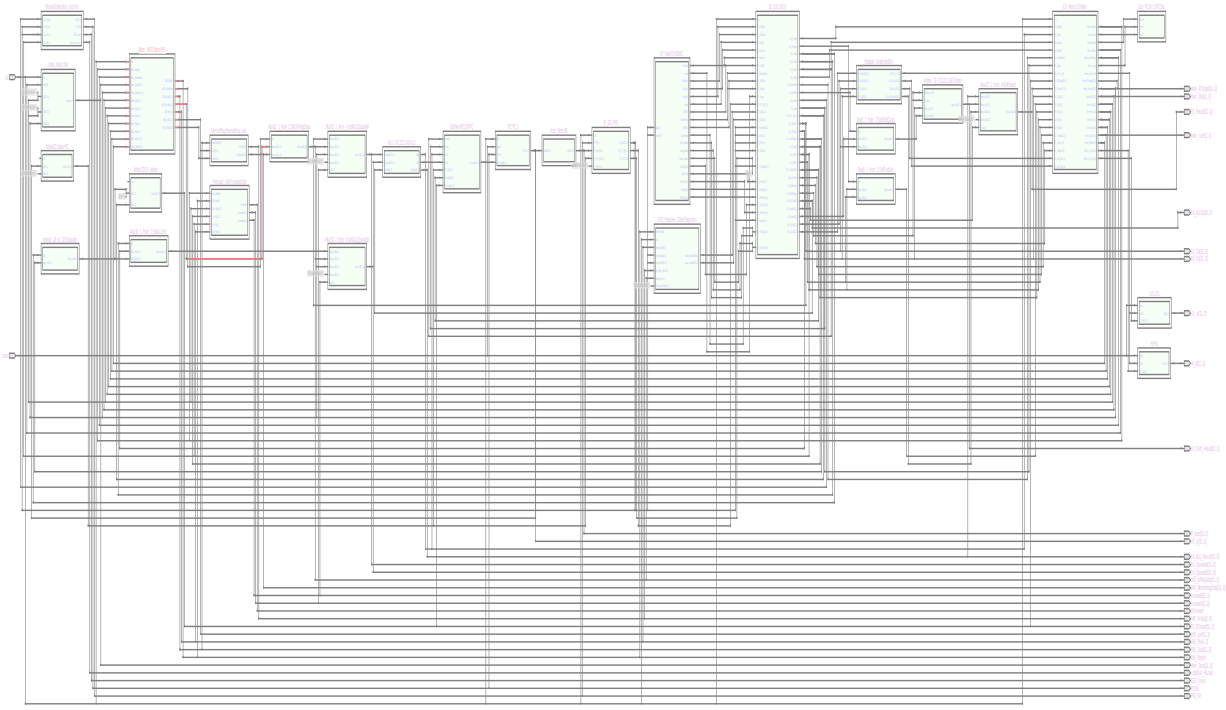
下图是具体内部结构：



至此，我们完成了所有流水线 CPU 数据通路的设计：

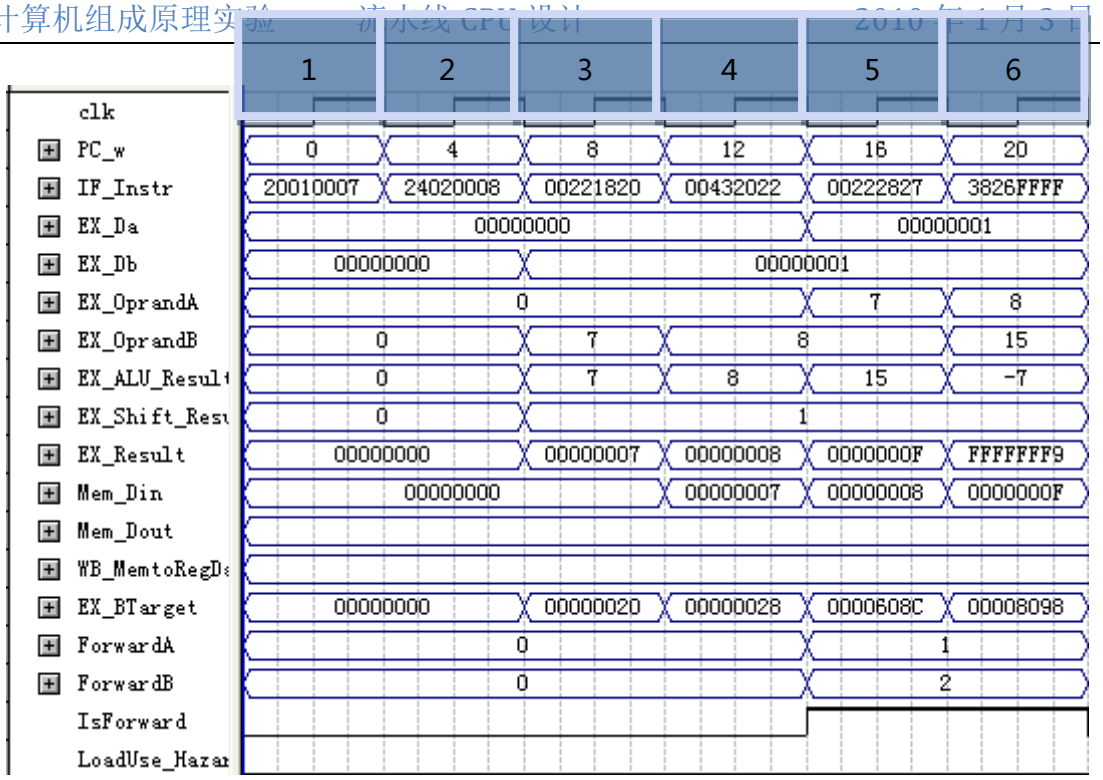


如下是整个的 RTL 图：



五、实验测试及结果分析

注：所有的寄存器（除了 0 号寄存器）都被初始化为 1



【1】 指令 20010007 addi $1(rt), $0(rs), 0x07

操作数 OprandA=Reg[Rs]=Reg[0]=0,OprandB=Imm16=7

计算结果=ALUResult=0+7=7。

【2】 指令 24020008 addiu $2(rt), $0(rs), 0x08

操作数 OprandA=Reg[Rs]=Reg[0]=0,OprandB=Imm16=8

计算结果= ALUResult=0+8=8。

【3】 指令 00221820 add $3(rd), $1(rs), $2(rt)

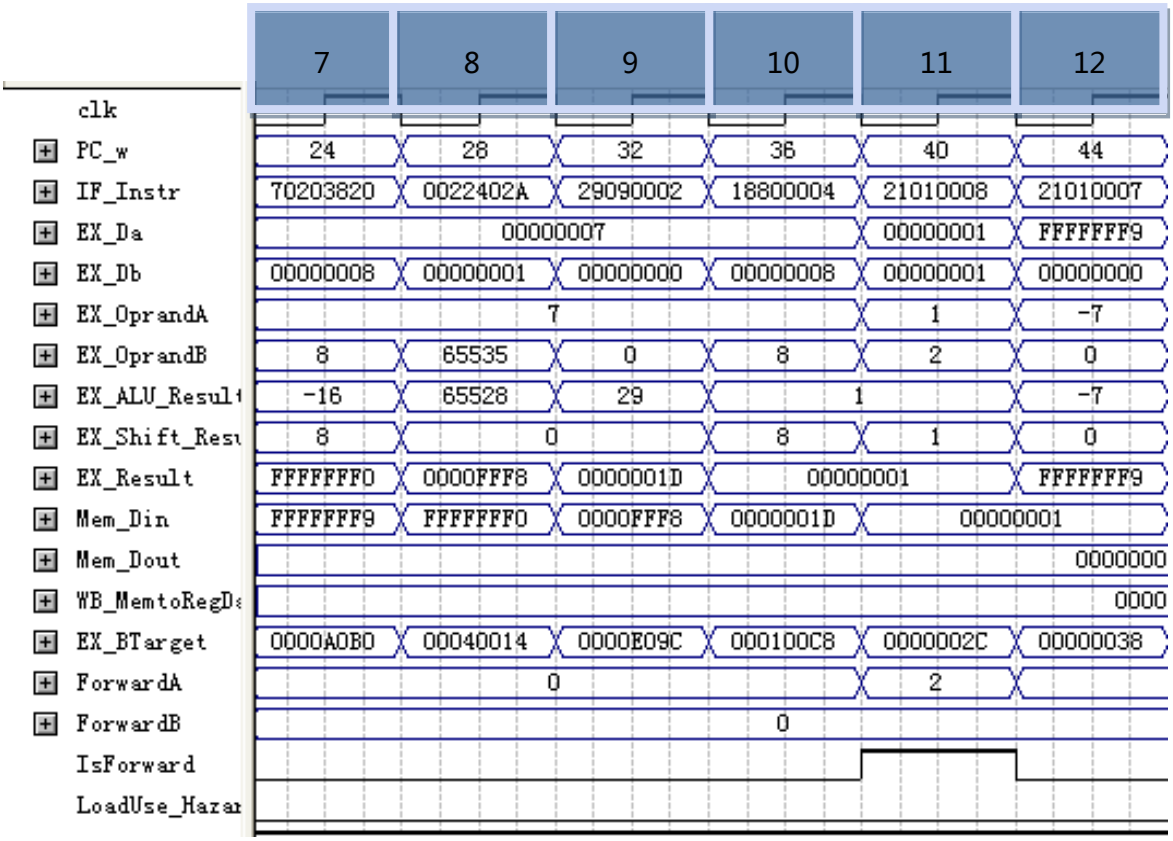
这条指令和前两条都发生了数据相关。

首先，在指令 1Ex 阶段刚计算完还没有写入$1 的时候，指令 3 就已经要读取$1 中的值，由于设计了转发，因此操作数 OprandA=7 而真实的寄存器 Reg[$1]=1，说明我们设计的转发正确。同理$2 中的情况也是如此，因此操作数 OprandB=8 而真实的寄存器 Reg[$2]=1。

计算结果= ALUResult=7+8=15。

【4】指令 00432022 Sub $4 (rd) $2 (rs) $3 (rt)

这条指令$3 的情况和前面的情况一样，在上一条指令在 Ex 阶段刚计算出结果，就要用到这个值，所以 OprandB=15。$2 的情况稍有不同，与再前面的指令 2 发生数据相关，这个时候那条指令正处在 MEM 阶段，直接把数据给了指令 4。所以 OprandA=8，计算结果= ALUResult=8-15=-7。



【5】指令 00222827 nor $5(rd), $1(rs), $2(rt)

OprandA=Reg[$1]=7，OprandB=Reg[$2]=15

计算结果=0x00000007 nor 0x0000000F=FFFFFFF0=-16

【6】指令 3826FFFF xori $6(rt), $1(rs), 0xffff

OprandA=Reg[$1]=7,OprandB=0x0000FFFF=65535

计算结果=0x00000007 xor 0xFFFFFFFF=0x0000FFF8=65528

【7】指令 70203820 clz　$1 前导零

操作数为 Reg[$1]=7=0x00000007 前导零个数为 29

计算结果=29，正确。

【8】指令 0022402A　　slt $8(rd), $1(rs), $2(rt)

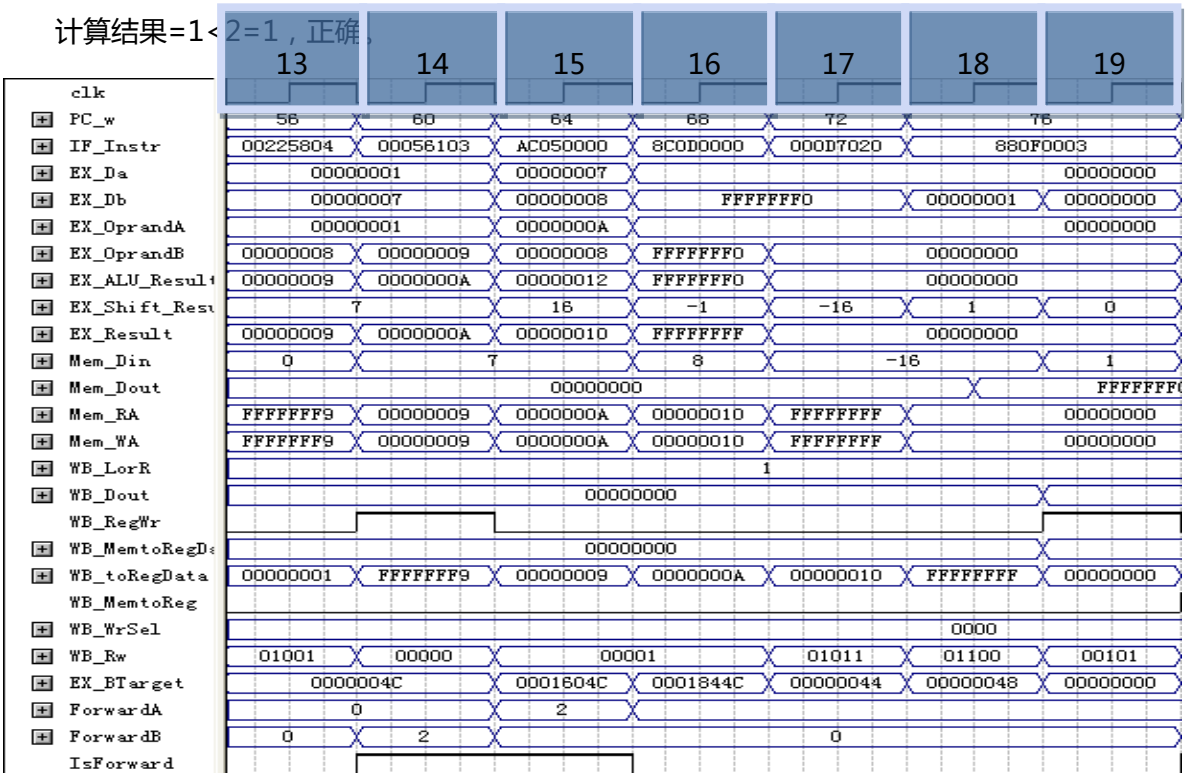
OprandA=Reg[$1]=7,OprandB[$2]=8

计算结果=7<8=1，正确。

【9】指令 29090002 slti $9(rt), $8(rs), 2

OprandA=Reg[$8]=1，OprandB=2，转发正确。

计算结果=1<2=1，正确。



【10】指令 18800004　　blez $4, 4

OprandA=Reg[$4]=-7<0，NextPC=PC+4+4<<2=PC+20=36+20=56 即往下跳转 4 条。但是计算得下一条地址时 PC 已经等于 44，也就是浪费了两个周期。因此当中要有 4 个空指令，如下所示：

nope: to test blez　21010008

nope: to test blez　21010007

nope: to test blez　21010008

nope: to test blez　21010007

【11】指令 00225804　　sllv $11(rd), $2(rt), $1(rs)

OprandA=1 ,OprandB=Reg[$2]=8

将 8 左移 1 位结果为 16

计算结果=移位器结果=16

【12】指令 00056103　　sra $12(rd), $5(rt), 4(shamt)

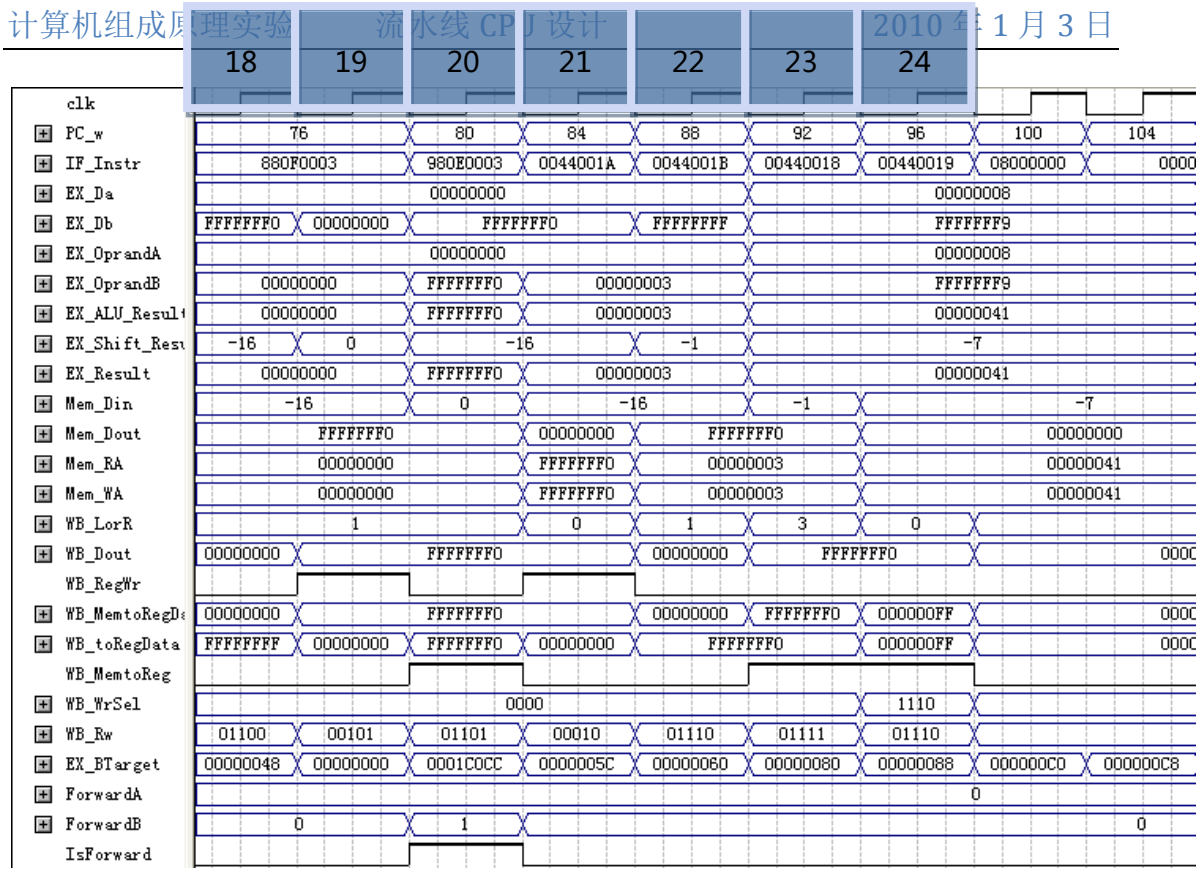
该条指令把 Reg[$5]内的数值算数右移 4 位

Reg[$5]=0xFFFFFFF0，算数右移之后变成 0xFFFFFFFF，正确。

【13】指令 AC050000　　sw $5(rt), 0(0)

该条指令把 Reg[$5]中的数值完整的写入地址为 0 的数据寄存器中。

Reg[$5]=0xFFFFFFF0，Mem_in=-16=0xFFFFFFF0（该结果在图中标注的 18 段可以看到），正确。



【14】指令 8C0D0000　　lw $13(rt), 0(0)

该条指令把数据寄存器地址为 0 的数据写入到 Reg[$13]内。

这时候看到下面的指令 15 要用到 Reg[$13]内的内容，但是里面的数值要到最后一个阶段才能得到，发生了 load-use 数据冒险，于是我们看到流水线阻塞一个时钟周期，在发现 load-use 的时候后面两条指令已经取出来了，所以波形上反映的是 PC=76 阻塞了一个周期。但本条指令继续正常执行，在第 20 个周期我们可以看到 WB_toRegData=0xFFFFFFFF。正确。

【15】指令 000D7020　　add $14(rd), $0(rs), $13(rt)

本条指令由于阻塞了一个周期，因此实际上从第 18 个周期才开始执行。到 20 个周期进行 ALU 计算，我们可以看到取出的 Reg[$13]的值就是刚才 lw 的值 0xFFFFFFFF。

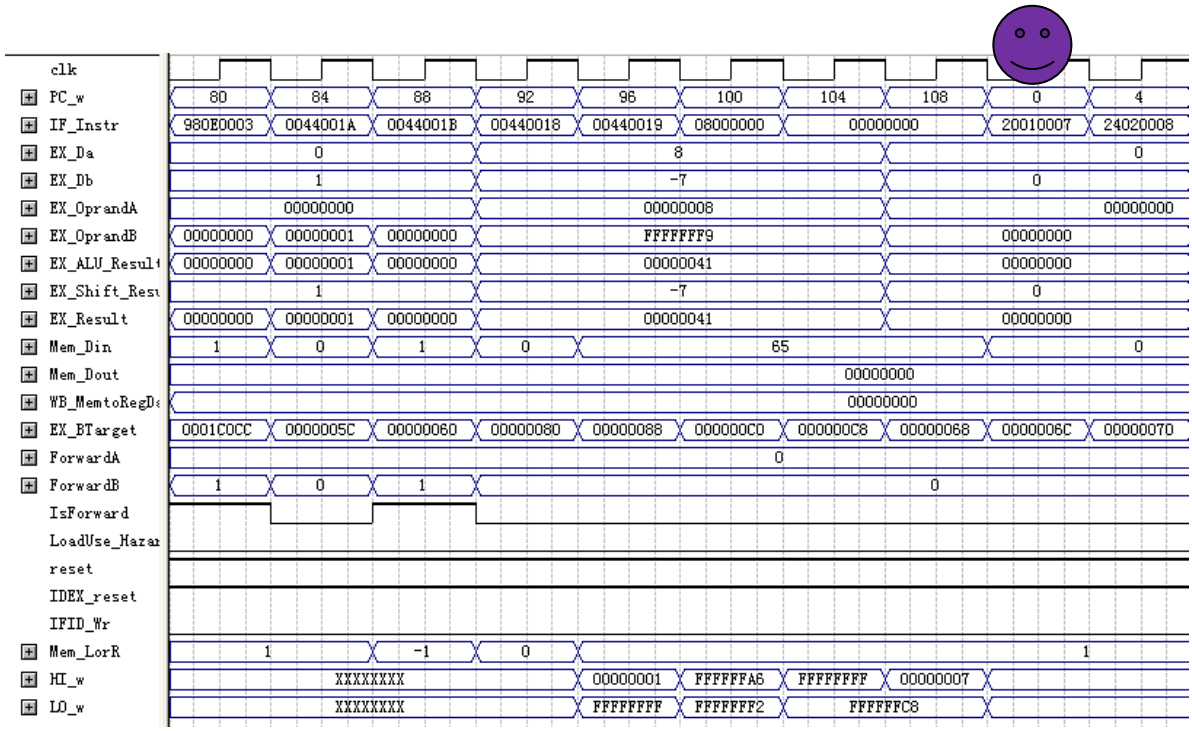
【16】指令 880F0003　　lwl $14, 3($0 = 0)

该条指令把 0 号地址的数据全部写入 Reg[$14]，因为 Offset=3，我们看到第 23 个周

期写入，数据为 0xFFFFFFF0，正确。

【17】指令 980E0003　　lwr $15, 3($0 =0)

该条指令把 0 号地址的数据的最高位八位组写入 Reg[$15]的最低位八位组，因为 offse=3。我们看到第 24 个周期写入，数据为 0x000000FF，正确。



【18】指令 0044001A　　div $2(rs),$4(rt)

该条指令把 Reg[$2]和 Reg[$4]相除，商放入 LO，余数放入 HI。

OprandA=8，OprandB=-7

HI=1，LO=-1=0xFFFFFFFF

【19】指令 0044001B　　divu $2(rs),$4(rt)

无符号除法。

OprandA=8，OprandB=0xFFFFFFF9

HI=0xFFFFFFA6，LO=0xFFFFFFF2

【20】指令 00440018　　mult $2(rs), $4(rt)

该条指令把 Reg[$2]和 Reg[$4]相乘，乘积低位放入 LO，高位放入 HI

OprandA=8，OprandB=-7

HI=0xFFFFFFFF，LO=0xFFFFFFC8

【21】指令 00440019　　multu $2(rs), $4(rt)

无符号乘法。

OprandA=8，OprandB=0xFFFFFFF9

HI=7，LO=0xFFFFFFC8

【22】指令 08000000　　j 0

我们看到在之后的第三个时钟(笑脸的地方)，PC 跳转到 0，跳转成功！

六、思考题

流水线 CPU 的特点？设计流水线 CPU 与多周期 CPU 有什么不同？应注意哪些问题？三种 CPU 的区别比较

MIPS 指令可大致分为以下五个阶段：

- IF 阶段：外部时钟信号下降沿触发 PC 寄存器，使得地址值更新；指令存储器根据 PC 值寻址找到指令代码，送交控制单元与寄存器地址输入端
- ID 阶段：控制单元解析指令，更新各个控制信号线的状态。同时寄存器将地址所对应的值输出。
- EXE 阶段：根据输入的数据和指令码进行运算，并将结果输出。

- MEM 阶段：在 lw 指令下，数据存储器根据地址找到数据并输出。在 sw 指令下，数据存储器根据地址和输入数据更新存储器的值。
- WB 阶段：将计算结果或数据存储器的数据写入寄存器。

从指令的执行来看

单周期 CPU 每条指令在一个时钟周期内完成，一条指令执行完再执行下一条指令。在一个时钟周期内完成上述五个阶段的操作，每个时钟下降沿更新地址。因此，要依照最长延迟的指令（lw）的时间来确定时钟周期的时间长度，无论指令的类型和它实际需要的执行时间如何，每条指令都要执行一个时钟周期。

多周期 CPU 一条指令采用多个周期，分别执行指令的若干阶段。由于 R 型指令和 I 型指令都不需要访问数据存储器，只需要 4 个周期；sw 指令不需要写回寄存器，只需要 4 个周期；blez 指令在译码/取数阶段可以投机计算分支目标地址，因此只需要 3 个周期；j 指令直接把指令中的 target 与 PC+4 的高 4 位拼接，低位在拼接两个 0 就得到了转移目标地址，也只需要 3 个周期。也就是说，只有 lw、lwr、lwl 指令需要 5 个周期。我们使用有限状态机来设计多周期 CPU 的控制部件，使不同的指令使用不同数量的时钟周期。多周期 CPU 与单周期 CPU 其实非常类似，只是控制单元从一个组合解码逻辑变成了一个状态机。

流水线与单周期有点相似，都是一个时钟周期完成一条指令，但是从指令本身来看，是根据 MIPS 指令的五个阶段来划分，需要五个周期执行。在指令译码阶段，由控制单元生成所有的控制信号，分别在随后的各个时钟周期内被使用。每个流水段寄存器中保存的信息包括：后面阶段需要用到的所有数据信息（也就是前面阶段在数据通路中执行的结果）和前面传递过来的后面各阶段要用到的所有控制信号。每个时钟周期的下降沿来临时，此指令所代表的一系列数据和控制信号将转移到下一个周期的组合逻辑输入上，在经过组合逻辑延时后，

处理过的数据在组合逻辑的输出端产生，并等待下一个时钟沿到来时被转移到下一个周期去。每个周期都有一条指令开始执行，也都有一条指令执行完毕。

从执行效率来看

单周期 CPU 时钟周期远远大于许多指令实际所需执行时间，R 型指令和 I 型指令都不需要访问数据存储器，sw 指令不需要写回寄存器，blez 指令不需要访问内存和写寄存器，j 指令不需要 ALU 运算，不需要读内存，也不需要写寄存器。受时钟周期宽度的影响，单周期 CPU 的效率低下，性能较差。

多周期 CPU 使得指令不需要做没有必要的操作，从而使每条指令执行的平均时间下降，但是依然是串行执行，总是在执行完一条指令后才取出下一条指令执行。显然这种方式没有充分利用执行部件的并行，因而执行效率低。

流水线 CPU 每条指令的执行时间其实并没有缩短，反而可能会比单周期还要长。但是执行 N 条指令时，如果每个功能段划分均匀，使得执行时间大致相等的话，流水线 CPU 的执行效率将是单周期　CPU 的 5 倍。

从信号竞争来看

首先实际的寄存器堆和存储器在单周期通路里不可能可靠工作，因为不能保证地址和数据能在“写使能”信号有效前稳定，也就是说，地址、数据和“写使能”之间存在竞争问题。

多周期 CPU 通过如下方式解决竞争问题：首先确认地址和数据在第 N 周期结束时已经稳定，然后，使“写使能”信号在第 N+1 个周期时有效，并使地址和数据在“写使能”信号无效前不改变其值。

流水线 CPU 则不能采用上述方法，原因是每个周期都必须能够写寄存器和存储器。流水线 CPU 采用将写使能信号和时钟信号与的方式来处理。

从控制单元来看

单周期 CPU 每条指令的控制信号由控制单元产生，在指令执行期间是不变的。

多周期 CPU 每条指令分多个周期执行，控制单元的功能采用有限状态机来描述，根据状态来确定控制信号。

而流水线 CPU 控制信号是在 ID 段由控制单元生成的，一旦生成就不会改变，并按部就班地一次传递到后面的流水段中，与单周期 CPU 类似。与单周期不同的是，流水线 CPU 在每个时钟周期都同时存在不同指令的控制信号。

流水线的冒险及其处理

流水线的冒险主要分为三种：结构冒险、数据冒险和控制冒险。

结构冒险：

现象：同一个部件同时被不同指令所使用

解决：将指令存储器 IM 和数据存储器 DM 分开，以避免冲突

数据冒险：

现象：后面指令用到前面指令结果时，前面指令结果还没产生

解决：采用转发(Forwarding/Bypassing)技术

Load-use 冒险需要一次阻塞(stall)

控制冒险：

现象：转移或异常改变执行流程，顺序执行指令在目标地址产生前已被取出

解决：采用静态或动态分支预测

编译程序优化指令顺序(实行分支延迟)

七、实验注意点和心得体会

首先，关于流水线设计思路的体会。

有了单周期和多周期 CPU 的实验基础，我们对各种指令的执行流程和数据通路都有了较为深入的认识，这也给流水线 CPU 的设计打下了基础。但是在开始设计流水线时，必须先了解流水线的思想，流水线 CPU 虽然在数据通路上看上去和单周期，多周期差不多，都是由那几个功能部件组成，但是当理解了流水线的思想后我们就发现其实流水线的实现与前面那两个是有本质不同的。

和单周期类似，流水线 CPU 是一个时钟完成一条指令，和多周期一样，将一条指令分为几个不同的阶段完成，如果你不了解它的机制，就会感到困惑。所以在初期，我们复习了流水线的基本内容，查阅了组原教材和相关资料，深入认识了流水线 CPU。

在流水线 CPU 中，将各条指令都统一划分为五个阶段，我们需要弄清楚各个指令在每个周期都完成什么样的操作，它们的数据通路分别是什么，对应什么控制信号。我们发现在流水线 CPU 中，对于一条指令来说，在它的生命周期里它所对应的所有控制信号是不变的，这跟单周期比较类似，与多周期不同，所以我们在译码阶段就可以确定下来所有的控制信号，而不需要想多周期那样通过有限状态机的转换来产生控制信号。进而，我们发现尽管如此，但是对于一个部件或者部件单元来说，它在不同的时钟周期内可能执行不同的操作，因而对应的是不同的控制信号，究其原因，是因为它在不同的时钟周期完成的是不同的指令。

2、基于 1 中的讨论，我们认识到了流水线寄存器的必要性。这是由流水线特有的性质决定的。在流水线 CPU 的数据通路中有必要定义四个流水线寄存器。因为在流水线 CPU 中，我们将一条指令的执行分为五个阶段，也将流水线的数据通路分为五个部分，在同一个

时钟周期内，五个部分分别执行五种不同的操作，操作数和控制信号的值都是不同的，所以每个时钟周期都需要更新，而对于一条指令，它从一个阶段进入下一个阶段，它所对应的控制信号，也就是对它进行译码是产生的控制信号的值是始终不变的，这就需要我们用寄存器将这些值和控制信号都保存起来，等时钟到来时，输出给下一个部分，同时，存入前面（后来到）的控制信号和前面部件产生的中间结果，等待在下一个时钟到来输出给下一个部分从而传递下去直至指令执行结束。

在本次实验中，我们将两个阶段间的寄存器组定义为一个模块，其中存储前一阶段计算的中间结果和下一阶段所需的控制信号。在这个模块的定义中，将新的结果以及新的控制信号定义为输入，原来寄存器组中存储的值为输出。这个看似很简单，但是在实现中我们遇到了一个不大不小的问题：我需要在这个模块中另外再定义 reg 变量，用这些变量来存储值，但是在有很多输入的时候，这样的话赋值语句就会很多，带来编码的麻烦。如果模块中不定义 reg（后来没有采用），可以将每个输出都定义为 output reg 效果是差不多的。总而言之，在这些寄存器组中总需要 reg 来存储旧值。

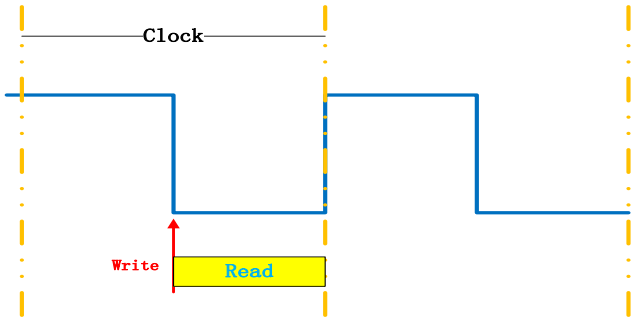
认识到 1，2 的问题就可以着手设计是实现出一个一般的不带竞争冒险检测的流水线 CPU 了。这里面也有一些需要讨论的问题，在组成原理的教材中是将 Branch 和 Jump 指令放在第四周期跳转，在本实验的实现中，是在第三个周期计算出 ALU 的结果就判断，进而获得下一个 PC 的值，等到下一个时钟周期到来时，进行跳转。可能由于我们认识还不是很到位，但是现阶段我们还是认为等到第四周期计算下一个 PC 的值没有必要，而且在我们这样的实现中没有遇到什么问题。其次，对于前取指令部件，指令译码和寄存器取数部件，不需要控制信号，因为对应所有指令来说前两个周期的功能是一样的，而且 PC 每个时钟周期都会变，所以不需要写使能的控制信号。

在设计好最基本的流水线 CPU 后，我们开始着手流水线 CPU 最棘手的问题：冒险的解决。我们知道，因为流水线的机制，给流水线 CPU 带来了三种冒险：功能冒险，数据冒险，控制冒险。

功能冒险：由于我们实现的时候将指令存储器和数据存储器分开，所以功能冒险的问题解决了一大半。查阅了相关资料，我们针对这些冒险，提出并实现了一些解决办法。首先，我修改了寄存器和存储器的定义，使得写总发生在读之后。本实验中都是时钟下降沿触发，为了避免寄存器或存储器的数据地址和写使能信号的竞争冒险，当时钟上升沿才触发写事件，然后再进行读访问。在组原书上也给出了一种解决方案，是将写使能信号和时钟信号与，相当于一个新的写使能，这样可以防止竞争冒险。考虑到上半周期写下半周期读的要求以及时钟下降沿触发的事实，我们最后没有采取这种方案。

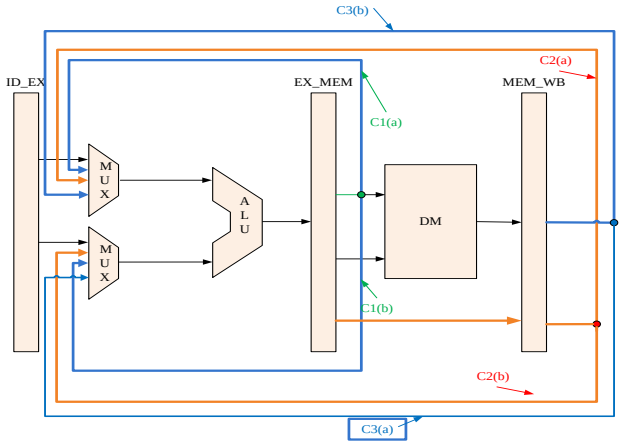
寄存器的功能冒险和竞争冒险的解决和存储器 DM 的竞争冒险的解决：

我们是这样解决这两个问题的：首先，因为我们的实现中都是时钟的下降沿触发，而在寄存器和存储器写，为了解决写地址和写数据，写使能的竞争冒险，我们定义的是时钟上升沿触发写，然后为了上半周期写，下半周期读，我们定义只能在时钟为 1 的时候才可以读出，具体如下图：



数据冒险：组成原理的课件给了我们很好的启发，所以最后也采用了类似的方案。但是

实现后发现，书上的方案并不是完全正确的。在检测需不需要旁路转发时，书上提到可以将图中的 C1(a) C1(b)C2(a)C2(b)合并为两根线 C1,C2.如图：



但是我们很快发现这样是行不通的，因为要判断转发的条件是看下一条指令 rs 或者 rt 是否和上一条指令 rd 或者 rs 相同，如果将两条线合起来表达的是并且的意思而非或者的意思了。当然做到这一点还有缺陷，应该排除 load 指令的情况。具体在后面有详细说明。

C)　Loaduse 冲突的检测，检测同数据转发的检测差不多，只是控制逻辑稍微复杂。在阻塞点，我们需要延迟一个周期执行后面的指令，这也就相当于巴阻塞点前面一个周期的状态再保持一个周期。也就是说发生了 load use 冲突，冲突点以前的指令照常执行，冲突点以后的指令冻结一个时钟周期。我们要做的就是使 PC 中的值不改变，下面 use 指令重新被取出，IF/ID 寄存器中的信息不变，重新译码，ID/EX 段寄存器中的所有控制信号清零。

在本 CPU 的数据通路中，我们还加入了除法器和乘法器。这样使得这个流水线 CPU 可以处理更多的指令。以前学过的除法器有保留余数法和不保留余数法，由于不保留余数法效率比较高，我们采用了后面的方案。在了解除法器和乘法器的原理的基础上，我们在网上查阅了很多资料，各实现了带符号和不带符号的除法器和乘法器，其中 CAS 除法器借鉴了刘文慧的设计，Booth 乘法器借鉴了网上的实现。

其次，在实现中遇到的具体错误和问题：

本实验的实现中，有很多 wire 型的变量，也就是说我们的电路图相对于前两次的比较复杂，布线复杂了，所以在调试过程中的很多问题都出在接线上面。有些线没有接上却以为接上了，后面又用到这根线，作为输入，这样导致输出错误。

开始的时候，我是基于前几次的经验，在编译成功后排查 warning，发现了不少错误，但是还有很多的漏网之鱼。后来同学介绍了一种方法，帮助我发现了很多找了很久都没有找到的错误，就是通过 RTL 图来排查布线的错误。之前只是用 RTL 图来看一下整个逻辑电路的框架，和基本模块的组织，这次在排查错误方面帮了我大忙。首先，我通过时序仿真，通过波形分析错误主要出在哪里，是哪个阶段，哪个部件的错误，然后放大 RTL 图很容易就找到那些没有接好的线。

本实验控制信号中的写使能信号都是低电平有效，在冲突检测部件和旁路转发部件中，一开始是照着书上来做的，而书上都是采用高电平有效，所以导致实现后，冲突检测部件一直无法发挥作用，经过长时间的分析和排查，终于意识到这样的问题，最后得以解决。

在实现旁路转发模块时，我也因为一味参照书本犯了错误。旁路转发模块需要根据 EX/Mem，ID/EX 中存储的 Rt,Rd,Rs 的值以及 RegWr 控制信号来判断是否需要转发。总的来说就是当判断到下一条指令的目的寄存器与上一条指令的 Rt 或者 Rs 相同并且寄存器写使能有效，就需要转发。书上对原始逻辑电路图的化简是错误的，在上面已经提到。

注意变量命名的方法。在开始实现的时候，我吃了很多不规范命名的亏，这并不是因为我的变量名毫无意义，而是因为在流水线 CPU 中我们需要区分在不同阶段的控制信号和中间值，所以后来我们一致使用规范命名。流水线中一共有五个阶段：取指令（IF），指令译

码，取数阶段（ID/DF）,执行阶段（EX）,存储器读数阶段（Mem）,写寄存器阶段（WB），每个变量处与那个阶段都以阶段的英文缩写加下划线开头，比如说执行阶段的 Ra 的值，就表示为 EX_Ra。同样各个功能部件在哪个阶段发挥做用也基本是这样的命名规则，放在不同的路径下面，比如 ALU 是在执行阶段，就放在 EX 文件夹下。

最后，是很严重的逻辑失误。

（1）在控制逻辑，我对 ALUSrc 的分析不周，只考虑了当指令为 I 型指令时 ALUSrc 应该为 1，然而 ALUSrc 取 1 还有 sw 和 lwl，lwr，lw 的情况，我起初忽视了这些错误导致逻辑出错。

（2）对于 DM（数据存储器）的读地址和写地址，对于 load 指令，地址应该是 base（寄存器取出来的值）加上 offset（立即数）的值，我误认为就是 Rt 和 Rs 中的一个（这个地址是写寄存器的写地址），所以产生了错误。我把它改成 ALU 做加法后的值就对了。

（3）转发单元（ForwardUnit）中的错误，我发现我输入这样的指令：

add $14, $0, $13　000D7020

lwl $14, 3 ($0 = 0)　880F0003

经过我写的转发单元竟然发生了转发，当时我写的转发逻辑是：

```
assign C1_a = ( (!Mem_RegWr) && (Mem_Rw != 5'b0) && (Mem_Rw == EX_Rs ));
assign C1_b = ( (!Mem_RegWr) && (Mem_Rw != 5'b0) && (Mem_Rw == EX_Rt ));
assign C2_a = ( (!WB_RegWr) && (WB_Rw != 5'b0) && ((Mem_Rw != EX_Rs)) && (WB_Rw == EX_Rs));
assign C2_b = ( (!WB_RegWr) && (WB_Rw != 5'b0) && ((Mem_Rw != EX_Rt)) && (WB_Rw == EX_Rt) );
assign ForwardA = {C1_a, C2_a};
assign ForwardB = {C1_b, C2_b};
```

原因是 C2_b 信号变为了 1，我发现当第一条 add 指令要写的目的地址是 14，而 lwl 虽然 14 也是要写的目的寄存器的地址，但是在 MIPS 指令中 14 是 lwl 的 rt 值，根据我的逻辑就会产生转发，所以我需要排除 lwl,lwr,lw 指令中 rt 与它上一条指令的写地址一样的情况。

修改 forwardUnit 逻辑如下：(MemtoReg=1 表示指令为 lw，lwl，lwr)

```
assign C1_a = ( (!Mem_RegWr) && (Mem_Rw != 5'b0) && (Mem_Rw == EX_Rs ));
assign C1_b = ( (!Mem_RegWr) && (Mem_Rw != 5'b0) && (Mem_Rw == EX_Rt )&&(!MemtoReg));
assign C2_a = ( (!WB_RegWr) && (WB_Rw != 5'b0) && ((Mem_Rw != EX_Rs)) && (WB_Rw == EX_Rs));
assign C2_b = ( (!WB_RegWr) && (WB_Rw != 5'b0) && ((Mem_Rw != EX_Rt)) && (WB_Rw == EX_Rt)
&&(!MemtoReg) );
assign ForwardA = {C1_a, C2_a};
assign ForwardB = {C1_b, C2_b};
```

总结

完成了流水线 CPU 的设计，我们的组成原理实验的课程也告一段落。回顾一个学期的学习，感觉学到了很多东西。老师课程安排的非常合理，让我们循序渐进，从几乎不懂 Verilog，一点一点熟悉起来，最后用它完成了 CPU 的设计，让我们感觉非常有成就感，所以在此感谢老师的安排和付出，以及平时不厌其烦帮我们排查错误的助教团队。

附录说明

程序模块功能说明
组员分工和评价
见“文档”文件夹内
Viso 图见“viso 图”文件夹内
工程实现见“代码”文件夹内