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. 掌握流水线 MIPS 微处理器的测试方法。

二、实验任务

设计一个 32 位流水线 MIPS 微处理器，具体要求如下：

1. 至少运行下列 MIPS32 指令。
   （1） 算术运算指令：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。

二、实验原理

1. 总体设计

流水线是数字系统中一种提高系统稳定性和工作速度的方法，广泛应用于高档 CPU 的架构中。根据 MIPS 处理器的特点，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）和寄存器会写（WB）五级，对应多周期的五个处理阶段。如图 3.1 所示，一个指令的执行需要 5 个时钟周期，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。

	1	2	3	4	5	6	7	8	9
I1	IF	ID	EX	MEM	WB				
I2		IF	ID	EX	MEM	WB			
I3			IF	ID	EX	MEM	WB		
I4				IF	ID	EX	MEM	WB	
I5					IF	ID	EX	MEM	WB

图 3.1 流水线流水作业示意图

由于在流水线中，数据和控制信息将在时钟周期的上升沿转移到下一级，所以规定流水线转移变量命名遵守如下格式：

名称_流水线级名称

例如：在 ID 级指令译码电路（Decode）产生的寄存器写允许信号 RegWrite 在 ID 级、

EX 级、MEM 级和 WB 级上的命名分别为 RegWrite_id、RegWrite_ex、RegWrite_mem 和 RegWrite_wb。在顶层文件中，类似的变量名称有近百个，这样的命名方式起到了很好的识别作用。

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、阻塞流水线的 PC_IFwrite 信号、清空流水线的 IF_flush 信号。

（2）ID 级：指令译码器。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。该级自身不需任何控制信号。

流水线冒险检测也在该级进行，冒险检测电路需要上一条指令的 MemRead，即在检测到冒险条件成立时，冒险检测电路产生 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。该级进行算术或逻辑操作。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、ALUScrB 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 ForwardA 和 ForwardB 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，对其他指令只起到一个周期的作用。该级只需存储器写操作允许信号 MemWrite。

（5）WB 级：写回级。该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。

2）流水线冒险

在流水线 CPU 中，多条指令通知执行，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。数据冒险分为两类：

ⅰ数据相关：流水线内部其中任何一条指令要用到任何其他指令的计算结果时，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。

ⅱ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，而此时其他指令可能还在执行中，这时会出现控制冒险，也称为分支冒险。解决此类冒险的常用方法是延迟分支。

2.1）数据相关与转发

下面通过具体例子来阐述数据相关。见图 3.2

```
sub   $2, $1,$3     # Register $2 written by sub
and   $12,$2,$5     # 1st 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
```

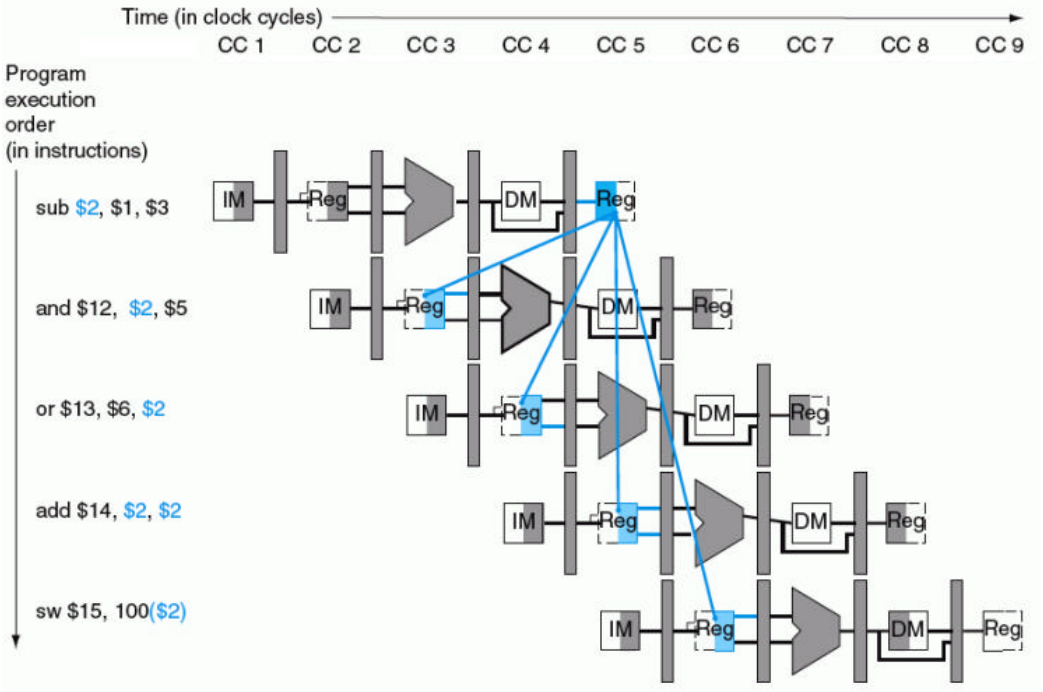


图 3.2 数据相关性问题实例图

可见，后 4 条指令都依赖于第一条指令得到寄存器$2 的结果，但 sub 指令要在第五周期才写回寄存器$2，但在第三、四、五个时钟周期$2 分别要被 and、or 和 add 三个指令用到，所以这三个指令得到的是错误的未更新的数据，会引起错误的结果；而第六个时钟周期$2 要被 sw 指令用到，此时得到的才是正确的已更新的数据。这种数据之间的互相关联引起的冒险就是数据相关。

可以看出，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）一阶数据相关与转发（EX 冒险）

首先讨论指令 sub 与 and 之间的相关问题。

sub 指令在第五周期写回寄存器$2，而 and 指令在第四周期就对 sub 指令的结果$2 提出申请，显然将得到错误的未更新的数据。像这类第 I 条指令的源操作寄存器与第 I-1 条指令（即上一条指令）的目标寄存器相重，导致的数据相关称为一阶数据相关。见图 3.3 中实线所示。

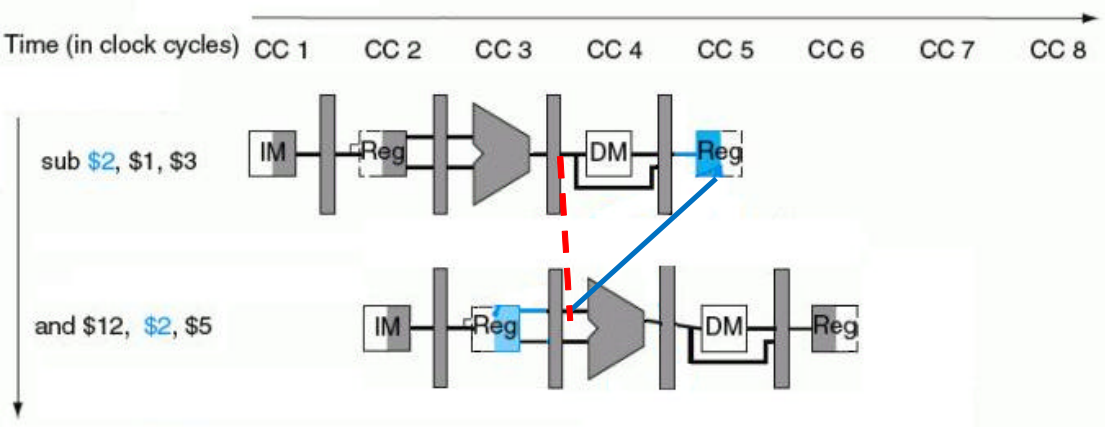


图 3.3 一阶数据相关实例图

可以发现，sub 指令的结果其实在 EX 级结尾，即第三周期末就产生了；而 and 指令在第四时钟周期向 sub 指令结果发出请求，请求时间晚于结果产生时间，所以只需要 sub 指令结果产生之后直接将其转发给 and 指令就可以避免一阶数据相关。如图 3.3 虚线所示。转发数据为 ALUResult_mem

数据转发由 Forwarding unit 单元控制，判断转发条件是否成立。转发机制硬件实现见图 3.4

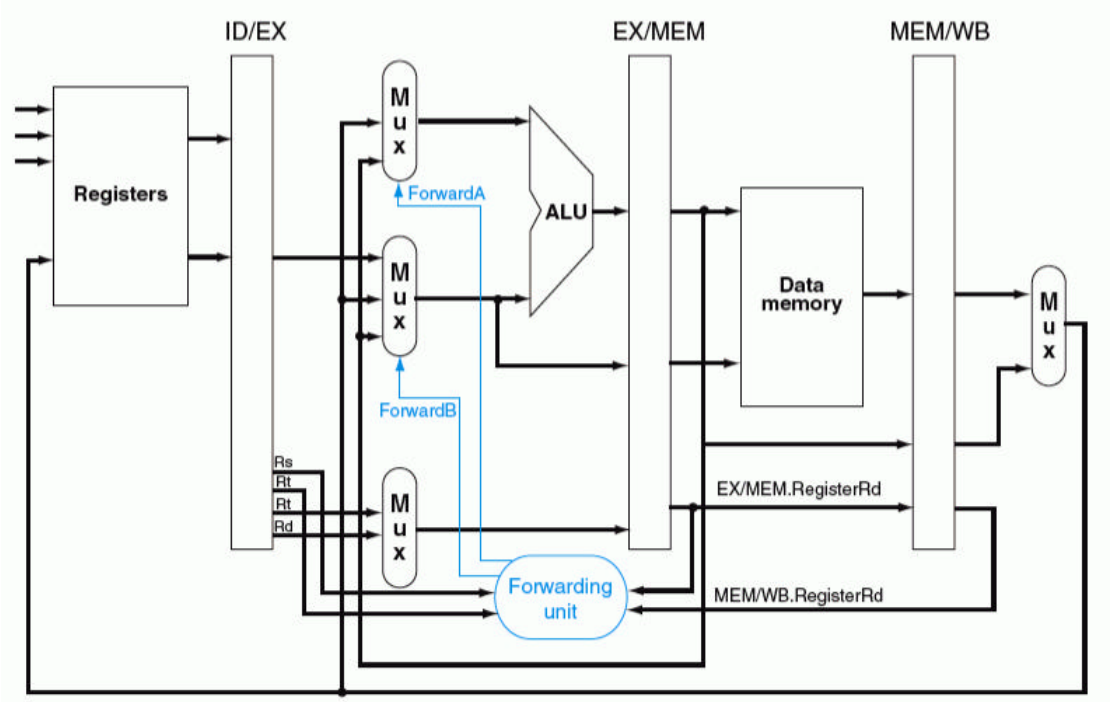


图 3.4 转发机制的硬件实现

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；转发条件成立时，ALU 操作数取自数据旁路。

转发条件：

① MEM 级指令是写操作，即 RegWrite_mem=1；

② MEM 级指令写回的目标寄存器不是$0，即 RegWriteAddr_mem≠0；

③ MEM 级指令写回的目标寄存器与在 EX 级指令的源寄存器是同一寄存器，即 RegWriteAddr_mem=RsAddr_ex 或 RegWriteAddr_mem=RtAddr_ex。

（2）二阶数据相关与转发（MEM 冒险）

接下来讨论 sub 指令与 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。这类第 I 条指令的源操作寄存器与第 I-2 条指令（即之上第二条指令）的目标寄存器相重，导致的数据相关称为二阶数据相关。见图 3.5 中实线所示。

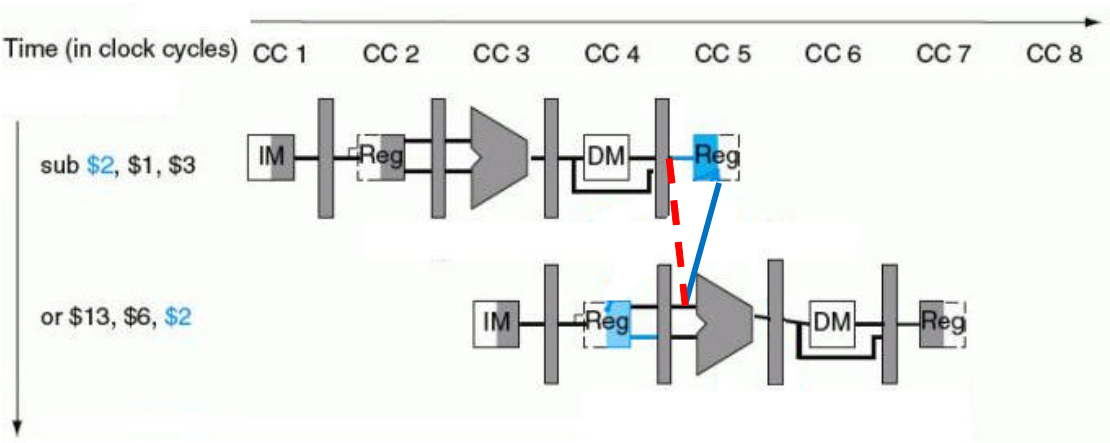


图 3.5 一阶数据相关实例图

如前所述，or 指令在第五时钟周期向 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。如图 3.5 中虚线所示。转发数据为 RegWriteData_wb

转发条件：

① WB 级指令是写操作，即 RegWrite_wb=1；

② WB 级指令写回的目标寄存器不是$0，即 RegWriteAddr_wb≠0；

③ WB 级指令写回的目标寄存器与在 EX 级指令的源寄存器是同一寄存器，即 RegWriteAddr_wb=RsAddr_ex 或 RegWriteAddr_wb=RtAddr_ex；

④ EX 冒险不成立，即 RegWriteAddr_mem≠RsAddr_ex 或 RegWriteAddr_mem=RtAddr_ex。

(3)三阶数据相关与转发

最后讨论 sub 指令与 add 指令之间的相关问题。

sub 指令与 add 指令在第五时钟周期内同时读写同一个寄存器。这类同一周期内同时读写同一个寄存器的数据相关称之为三阶数据相关。如图 3.6 中实线所示。

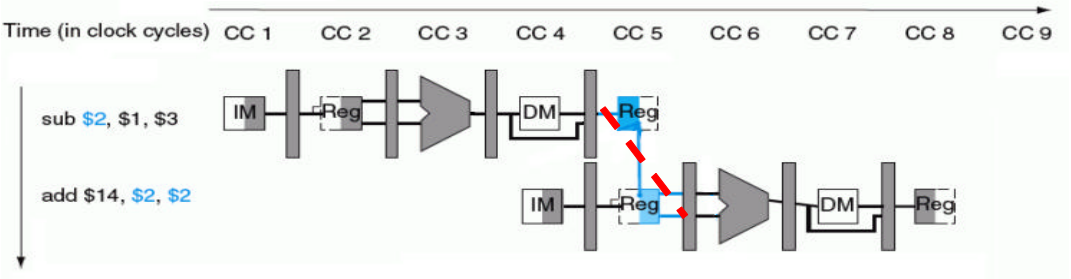


图 3.6 三阶数据相关实例图

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，我们采用“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。如图 3.6 中虚线所示。转发数据为 RegWriteData_wb。

转发条件为：

① WB 级指令是写操作，即 RegWrite_wb=1；
② WB 级指令写回的目标寄存器不是$0，即 RegWriteAddr_wb≠0；
③ WB 级指令写回的目标寄存器与在 ID 级指令的源寄存器是同一寄存器，即 RegWriteAddr_wb=RsAddr_id 或 RegWriteAddr_wb=RtAddr_id。

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。如图 3.7 所示

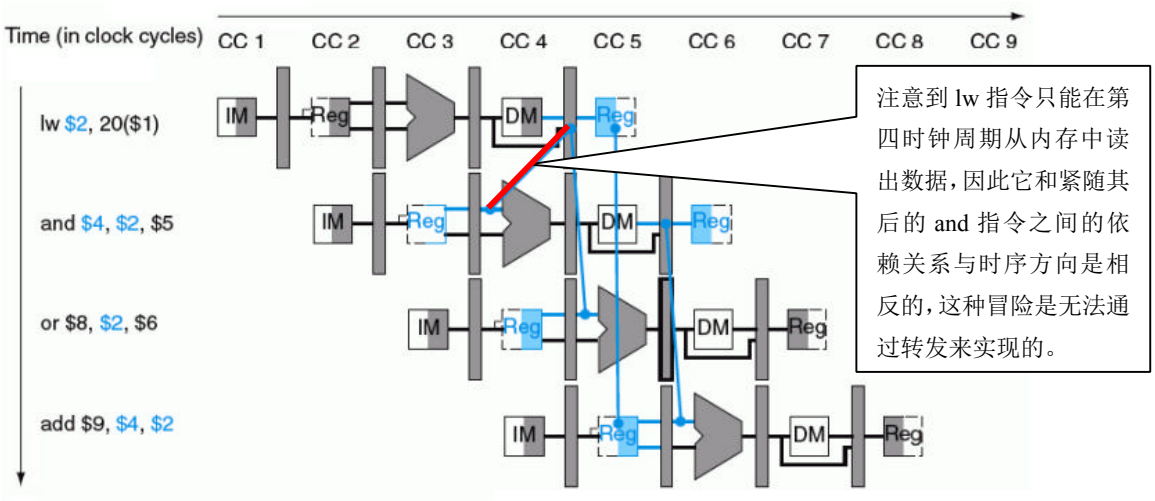


图 3.7 数据冒险与阻塞实例图

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：

为解决数据冒险，我们引入流水线阻塞。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。如图 3.8 所示。

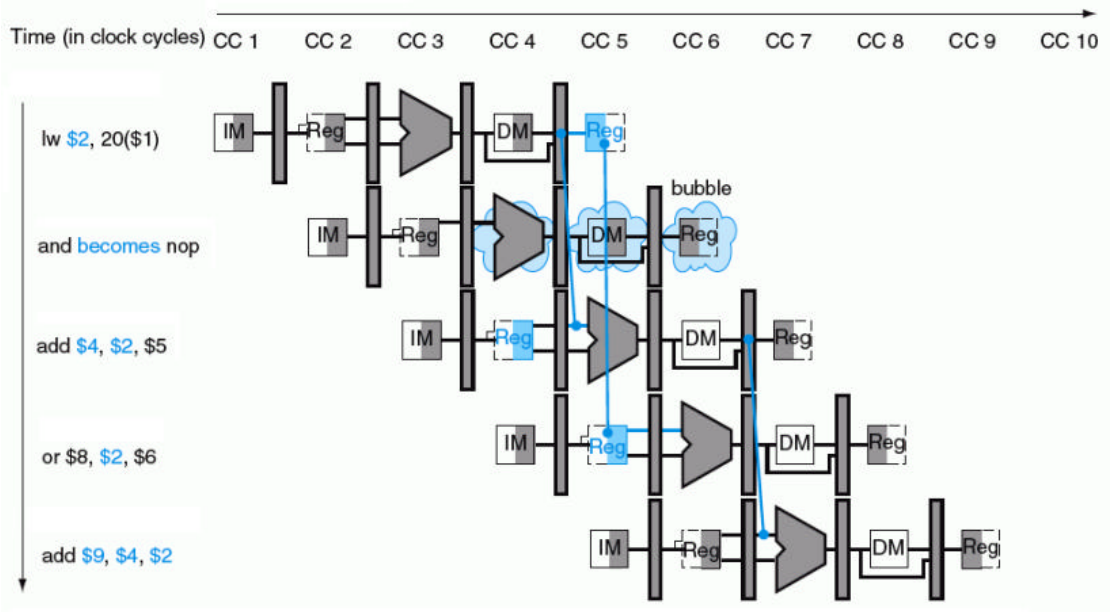


图 3.8 流水线气泡的引入

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。

具体实现方法如下：

在 ID 级检测到冒险条件时， HazardDetector 输出两个信号：Stall 与 PC_IFWrite。

Stall 信号将 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，高电平有效。

PC_IFWrite 信号禁止 PC 寄存器和 IF/ID 流水线寄存器接收新数据，低电平有效。

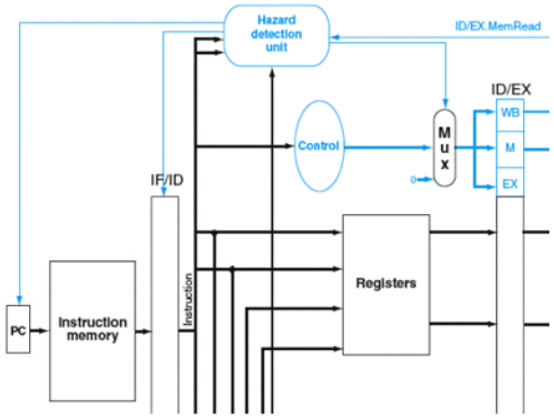


图 3.9 阻塞信号

2.3）分支冒险

还有一类冒险是包含分支的流水线冒险，下图。

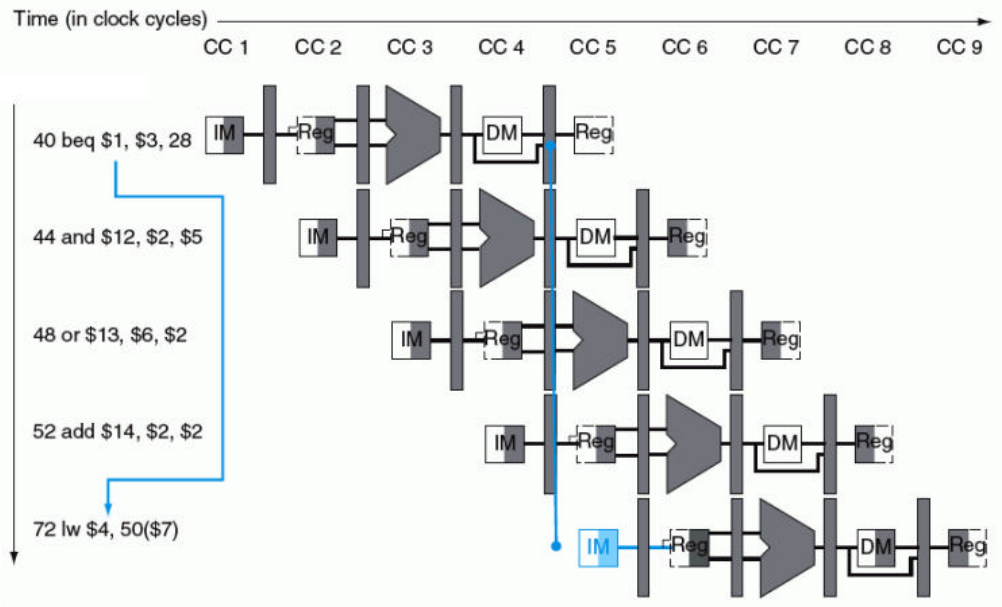


图 3.10 分支冒险实例

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。

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。

如果我们能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。

因此我们采用提前分支指令的方法解决分支冒险。

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：

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。

② 判断分支指令的跳转条件：

我们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。

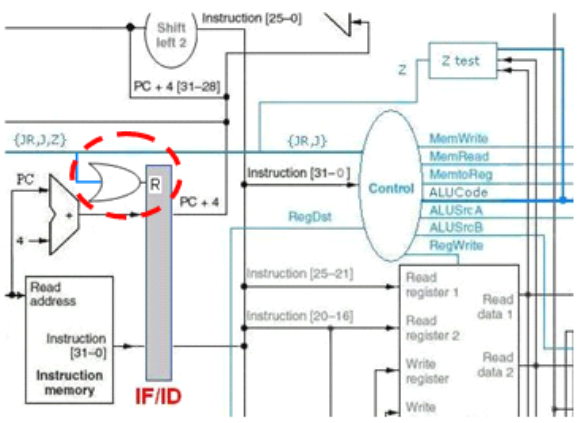
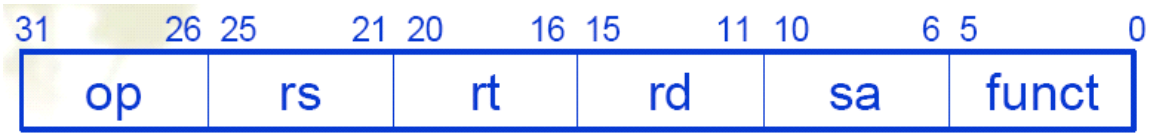


图 3.11 控制信号 IF_flush

2. MIPS 指令格式

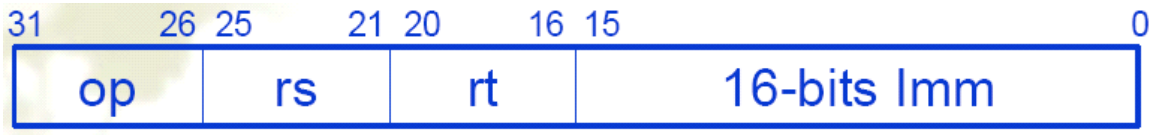
1） R 型指令格式



op、funct：共同决定指令名称，都为 6 位；
rs ： 指定第一操作数的寄存器地址，为 5 位；
rt ： 指定第二操作数的寄存器地址，为 5 位；
rd ： 指定目标寄存器地址，为 5 位；
sa ： 位移运算的移动位数，为 5 位。

本实验要实现的 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

2） I 型指令格式



op ： 决定指令名称，为 6 位；
rs ： 指定第一操作数的寄存器地址，为 5 位；
rt ： 储存结果的寄存器地址，为 5 位；
Imm：立即数，为 16 位。

本实验要实现的 I 型指令有：
① 存储器访问指令：LW、SW
② 立即数算术逻辑运算指令：ADDI、ADDIU、ANDI、ORI、XORI、SLTI、SLTIU
③ 分支指令：BEQ、BNE、BGEZ、BGTZ、BLEZ、BLTZ

注：
① 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：sign-extend(Imm)。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 对于条件分支指令：不执行分支语句时，跳转地址应为下一条地址；执行分支语句时，跳转地址应为下一条地址加上跳转指令数，即为立即数。由于跳转方向有两个：向前与向后，故立即数存在正负性，应该有符号扩展为 32 位：sign-extend(Imm)。
如果能够将立即数的位数扩充，跳转指令的范围将大大增加。
由于 CPU 中指令的起始地址都是 4 的倍数，因此它们地址的后两位都是 0，那么跳过的指令数后两位也应为 2’b00，故可将 16 为立即数左移两位扩充为 18 位，寻址地址范围也

扩大 4 倍。因此分支地址可表示为{PC+4+sign-extend(Imm)<<2}.

③ 对于取字指令 LW 操作为：rt<=Mem[rs+sign_extend(Imm)]

对于存字指令 SW 操作为：Mem[rs+sign_extend(Imm)]<=rt

由于地址的变化是在内存中进行，故立即数只需进行有符号位扩展，不能位移。

3） J 型指令格式

op ： 决定指令名称，为 6 位；

address：跳转的地址，为 26 位。

本实验要实现的 I 型指令有：

无条件跳转指令：J

注：指令都是 32 位，因此我们要把 26 位跳转地址扩展为 32 位。由于 CPU 中指令的起始地址都是 4 的倍数，因此它们地址的后两位都是 0，可以把 26 位的 address 扩展为 28 位：{26-bits address，2’b00}；剩下的最高 4 位则直接从 PC 中取，即跳转地址扩展为：{PC[31:28]，26-bits address，2’b00}。

3. 指令译码模块 ID 的设计

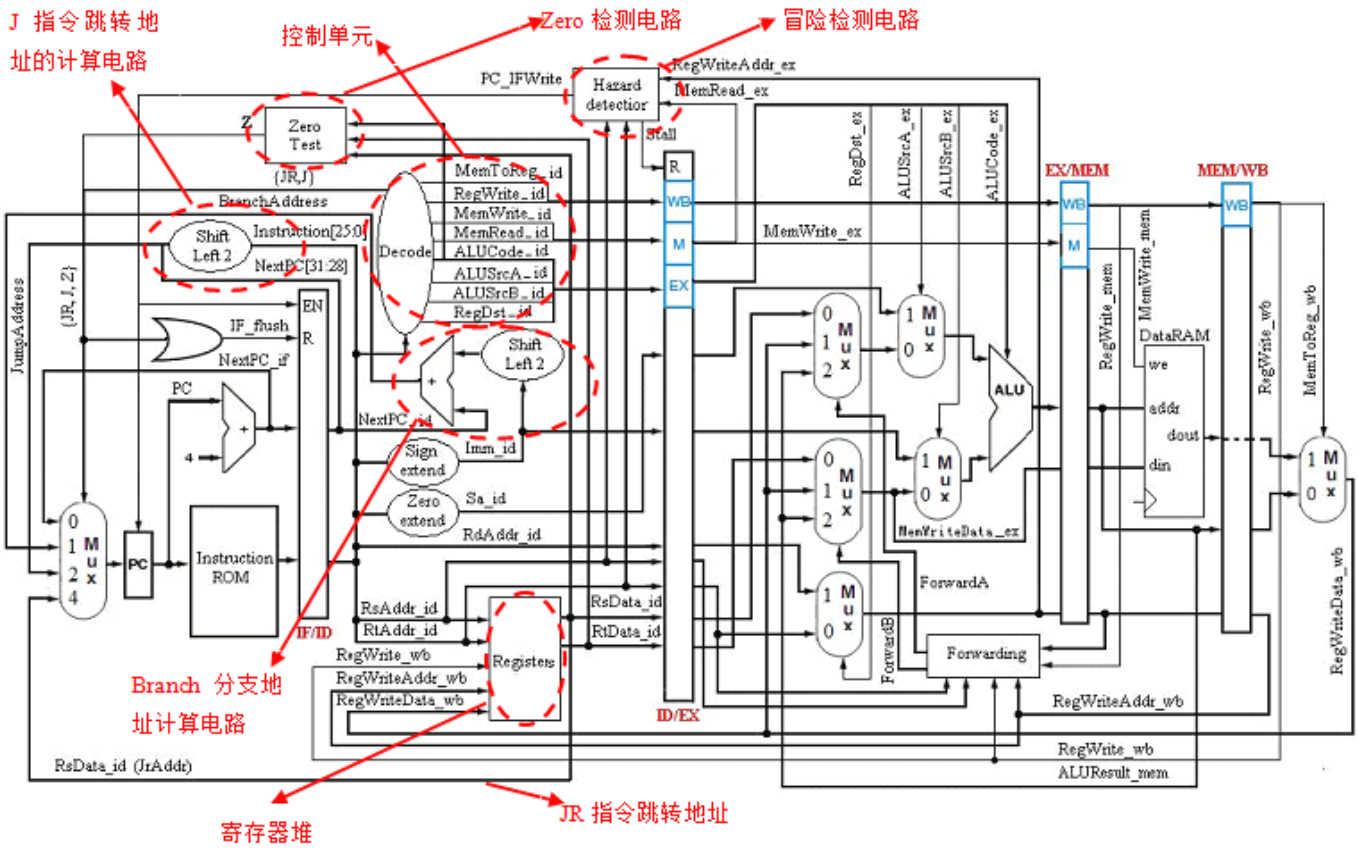


图 3.12 ID 级电路结构

指令译码模块的主要作用是从机器码中解析出指令，并根据解析结果输出各种控制信号。ID 模块主要有指令译码（Decode）、寄存器堆（Registers）、冒险监测、分支检测和加法器等组成。ID 模块的接口信息如下表所示：

引脚名称	方向	说　　明
clk	Input	系统时钟
Instruction_id[31:0]		指令机器码
NextPC_id[31:0]		指令指针
RegWrite_wb		寄存器写允许信号，高电平有效
RegWriteAddr_wb[4:0]		寄存器的写地址
RegWriteData_wb[31:0]		写入寄存器的数据
MemRead_ex		冒险检测的输入
RegWriteAddr_ex[4:0]		
MemtoReg_id	Output	决定回写的数据来源（0：ALU　1：存储器）
RegWrite_id		寄存器写允许信号，高电平有效
MemWrite_id		存储器写允许信号，高电平有效
MemRead_id		存储器读允许信号，高电平有效
ALUCode_id[4:0]		决定 ALU 采用何种运算
ALUSrcA_id		决定 ALU 的 A 操作数的来源（0：rs　1:Sa）
ALUSrcB_id		决定 ALU 的 B 操作数的来源（0：rt　1:Imm）
RegDst_id		决定 Register 回写是采用的地址（rt/rd）
Stall		ID/EX 寄存器清空信号，高电平插入一个流水线气泡
Z		分支指令的条件判断结果
J		跳转指令
JR		寄存器跳转指令
PC_IFWrite		阻塞流水线的信号，低电平有效
BranchAddr[31:0]		条件分支地址
JumpAddr[31:0]		跳转地址
Imm_id[31:0]		符号扩展成 32 位的立即数
Sa_id[31:0]		0 扩展成 32 位的移位立即数
RsData_id[31:0]		Rs 寄存器数据
RtData_id[31:0]		Rt 寄存器数据
RdAddr_id[4:0]		Rd 寄存器地址
RsAddr_id[4:0]		Rs 寄存器地址
RtAddr_id[4:0]		Rt 寄存器地址

(1) 指令译码（Decode）子模块的设计

Decode 控制器的主要作用是根据指令确定各个控制信号的值，是一个组合电路。我们将指令分成八类：

R_type1：ADD、ADDU、SUB、SUBU、AND、OR、
NOR、XOR 、SLT、SLTU、SLLV、SRLV、SRAV
R_type2：SLL、SRL、SRA
JR_type：JR
} R 型指令

J_type：J　　J 型指令

I_type：ADDI、ADDIU、ANDI、ORI、XORI、SLTI、SLTIU
Branch：BEQ、BNE、BGEZ、BGTZ、BLEZ、BLTZ
LW：LW
SW：SW
} I 型指令

Decode 输出的九组控制信号：

① RegWrite

决定是否对寄存器（Registers）进行写操作。当 RegWrite 高电平有效时，将数据写入指定的寄存器中。

需要写回寄存器的指令有：LW、R_type1、R_type2 和 I_type，则

RegWrite_id= LW ‖ R_type1 ‖ R_type2 ‖ I_type

② RegDst

决定目标寄存器是 rt 还是 rd。当 RegDst=0 时，rt 为目标寄存器；当 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。当 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。当 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。 当 MemtoReg=0 时，数据来自 ALU；当 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。当 ALUSrcA=0 时，ALU 第一操作数 A（详见转发电路设计）；当 ALUSrcA=1 时，ALU 第一操作数来源于 0 扩展的用于移位指令的 5 位 sa。

八种指令类型中 J_tpye、JR_tpye 及 Branch 类型指令没有使用 ALU；其他使用 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，所以：

ALUSrcA_id= R_type2

⑦ ALUSrcB

决定 ALU 第二操作数来源。当 ALUSrcB=0 时，ALU 第二操作数 B。当 ALUSrcB=1 时，ALU 第二操作数来源于符号扩展的 16 位 Imm。

八种指令类型中 J_tpye、JR_tpye 及 Branch 类型指令没有使用 ALU。其他使用 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

⑧ PCSource

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。

⑨ ALUCode

决定 ALU 的功能，由指令中的 op 段、rt 段和 funct 段决定。功能表如下：

op	funct	rt	运算	ALUCode
BEQ_op	××××××	××××××	Z=(A==B)	5'd10
BNE_op	××××××	××××××	Z=~(A==B)	5'd11
BGEZ_op	××××××	5'd1	Z=(A≥0)	5'd12
BGTZ_op	××××××	5'd0	Z=(A>0)	5'd13
BLEZ_op	××××××	5'd0	Z=(A≤0)	5'd14
BLTZ_op	××××××	5'd0	Z=(A<0)	5'd15
R_type_op	ADD_funct	××××××	加	5'd0
	ADDU_funct	××××××		
	AND_funct	××××××	与	5'd1
	XOR_funct	××××××	异或	5'd2
	OR_funct	××××××	或	5'd3
	NOR_funct	××××××	或非	5'd4
	SUB_funct	××××××	减	5'd5
	SUBU_funct	××××××		
	SLT_funct	××××××	A<B?1:0	5'd19
	SLTU_funct	××××××	A<B?1:0(无符号数)	5'd20
	SLL_funct	××××××	B<<A	5'd16
	SLLV_funct	××××××		
	SRL_funct	××××××	B>>A	5'd17
	SRLV_funct	××××××		
	SRA_funct	××××××	B>>>A	5'd18
	SRAV_funct	××××××		

op	funct	rt	运算	ALUCode
ADDI_op	×××××××	×××××××	加	5’d0
ADDIU_op	×××××××	×××××××		
ANDI_op	×××××××	×××××××	与	5’d6
XORI_op	×××××××	×××××××	异或	5’d7
ORI_op	×××××××	×××××××	或	5’d8
SLTI_op	×××××××	×××××××	A<B?1:0	5’d19
SLTIU_op	×××××××	×××××××	A<B?1:0(无符号数)	5’d20
SW_op	×××××××	×××××××	加（计算地址）	5’d0
LW_op	×××××××	×××××××		

(2) 分支检测（Branch Test）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：

$$
Z=\begin{cases}
\text{RsData[31]} \,\|\, \sim(|\,\text{RsData[31:0]}) \;; & \text{ALUCode=alu\_blez} \\
\text{RsData[31]} \;; & \text{ALUCode=alu\_bltz} \\
\sim\text{RsData[31]} \,\&\&\, (|\,\text{RsData[31:0]}) \;; & \text{ALUCode=alu\_bgtz} \\
\sim\text{RsData[31]}; & \text{ALUCode=alu\_bgez} \\
|\,(\,\text{RsData[31:0]} \,\hat{}\, \text{RtData[31:0]}) \;; & \text{ALUCode=alu\_bne} \\
\&\,(\,\text{RsData[31:0]} \sim\hat{}\, \text{RtData[31:0]}); & \text{ALUCode=alu\_beq} \\
0; & \text{ALUCode=OTHER}
\end{cases}
$$

(3) 寄存器堆（Registers）的设计

寄存器堆由 32 个 32 位寄存器组成，这些寄存器通过寄存器号进行读写存取。寄存器堆的原理框图如图 3.13 所示。因为读取寄存器不会更改其内容，故只需提供寄存号即可读出该寄存器内容。读取端口采用数据选择器即可实现读取功能。应注意“0”号寄存器为常数 0.

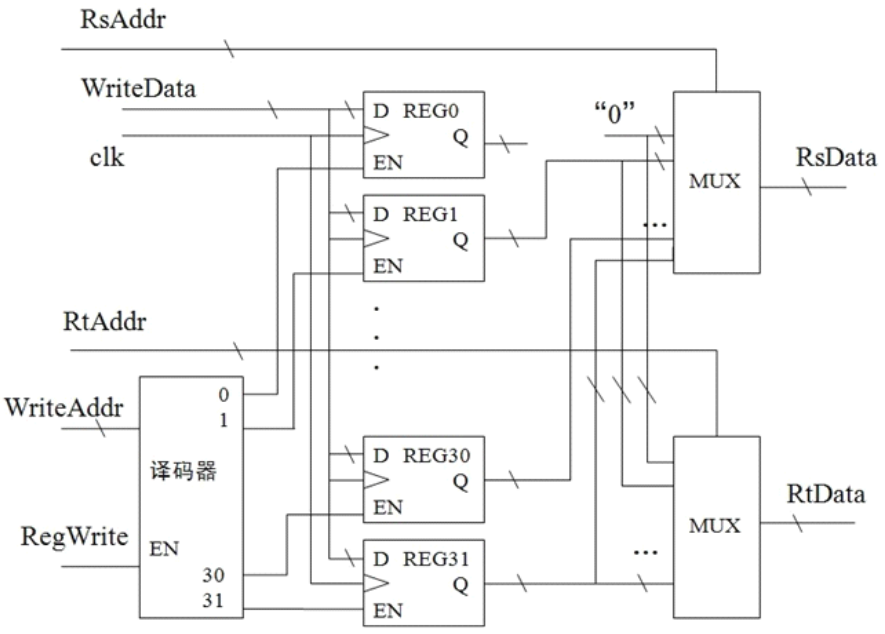


图 3.13 寄存器堆的原理框图

对于往寄存器里写数据，需要目标寄存器号(WriteRegister)、待写入数据(WriteData)、写允许信号（RegWrite）三个变量。图 3.13 中 5 位二进制译码器完成地址译码，其输出控制目标寄存器的写使能信号 EN，决定将数据 WriteData 写入哪个寄存器。

在流水线 CPU 设计中，寄存器堆设计还应解决三阶数据相关的数据转发问题。当满足三阶数据相关条件时，寄存器具有 Read after Write 的特性。为实现该功能，在寄存器堆的基础上加一转发电路。如图 3.14 所示。图中转发检测电路的输出表达式为

RsSel=RegWrite_wb&&(~(RegWriteAddr_wb==0))&&(RegWriteAddr_wb==RsAddr_id)

RtSel=RegWrite_wb&&(~(RegWriteAddr_wb==0))&&(RegWriteAddr_wb==RtAddr_id)

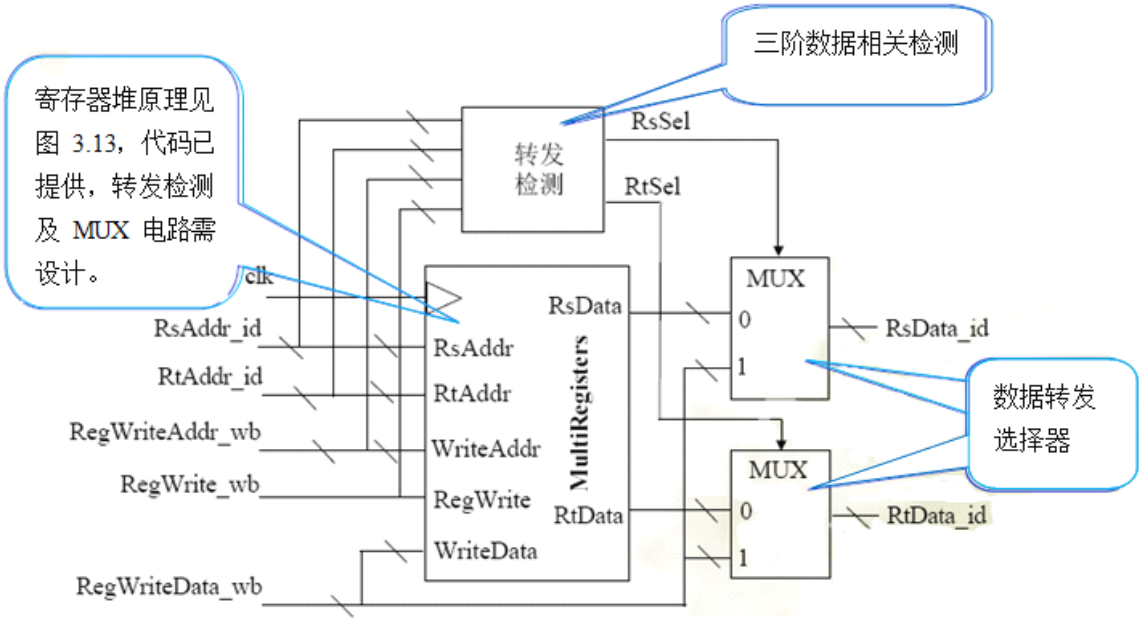


图 3.14 具有 Read after Write 特性寄存器堆的原理框图

(4) 冒险检测功能（Hazard Deterctor）的设计

由前面分析可知，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

解决冒险的方法为：

① 插入一个流水线气泡 Stall 清空 ID/EX 寄存器并且阻塞流水线 ID 级、IF 级流水线，有：

Stall=((RegWriteAddr_ex==RsAddr_id)||

(RegWriteAddr_ex==RtAddr_id))&&MemRead_ex

② 保持 PC 寄存器和 IF/ID 流水线寄存器不变，有：

PC_IFWrite=~Stall

(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；如果最高位是 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） 。

4. 执行模块 EX 的设计

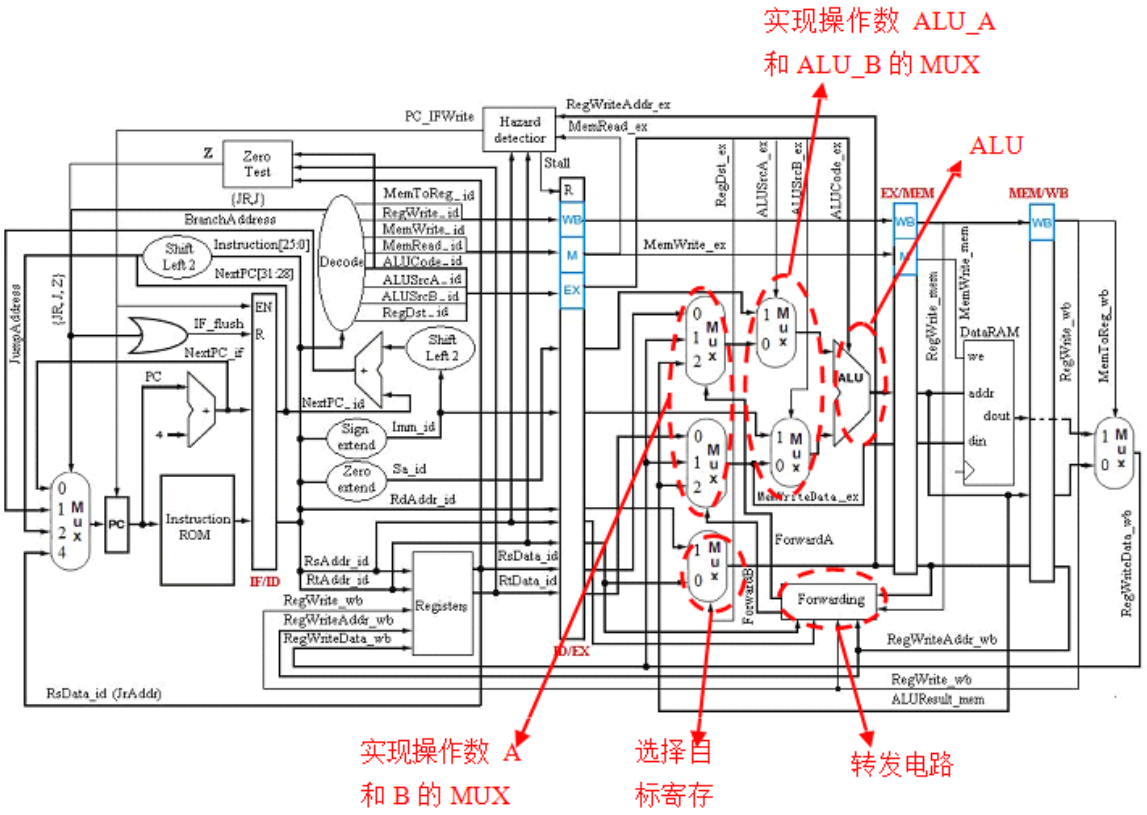


图 3.15 EX 级电路结构

执行模块主要有 ALU 子模块、转发电路 Forwarding 以及若干数据选择器组成。这行模块的接口信息如下表所示：

引脚名称	方向	说　　明
RegDst_ex	Input	决定 Register 回写时采用的地址（rt/rd）
ALUCode_ex[4:0]		决定 ALU 采用何种运算
ALUSrcA_ex		决定 ALU 的 A 操作数的来源（rs/Sa）
ALUSrcB_ex		决定 ALU 的 B 操作数的来源（rt/Imm）

引脚名称	方向	说　　明
Imm_ex[31:0]	Input	立即数
Sa_ex[31:0]		移位位数
RsAddr_ex[4:0]		Rs 寄存器地址，即 Instruction_id[25:21]
RtAddr_ex[4:0]		Rt 寄存器地址，即 Instruction_id[20:16]
RdAddr_ex[4:0]		Rd 寄存器地址，即 Instruction_id[15:11]
RsData_ex[31:0]		Rs 寄存器数据
RtData_ex[31:0]		Rt 寄存器数据
RegWriteData_wb[31:0]		写入寄存器的数据
ALUResult_mem[31:0]		ALU 输出数据
RegWriteAddr_wb[4:0]		寄存器的写地址
RegWriteAddr_mem[4:0]		
RegWrite_wb		寄存器写允许信号
RegWrite_mem		
RegWriteAddr_ex[4:0]	Output	寄存器的写地址
ALUResult_ex[31:0]		ALU 运算结果
MemWriteData_ex[31:0]		寄存器的回写数据
ALU_A[31:0]		ALU 操作数，测试时使用
ALU_B[31:0]		

(1) ALU 子模块的设计

ALU 是提供 CPU 基本运算能力的重要电路。ALU 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。ALU 功能见下表：

ALUCode	ALUResult
alu_add = 5'b00000	A+B
alu_and = 5'b00001	A&B
alu_xor = 5'b00010	A^B
alu_or = 5'b00011	A|B
alu_nor = 5'b00100	~(A|B)
alu_sub = 5'b00101	A-B
alu_andi= 5'b00110	A&{16'b0,B[15:0]}
alu_xori= 5'b00111	A^{16'b0,B[15:0]}
alu_ori = 5'b01000	A|{16'b0,B[15:0]}
alu_sll = 5'b10000	B<<A
alu_srl = 5'b10001	B>>A
alu_sra = 5'b10010	B>>>A
alu_slt = 5'b10011	A<B?1:0,其中 A、B 为有符号数
alu_sltu= 5'b10100	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。ALU 的基本结构如图 3.16 所示。

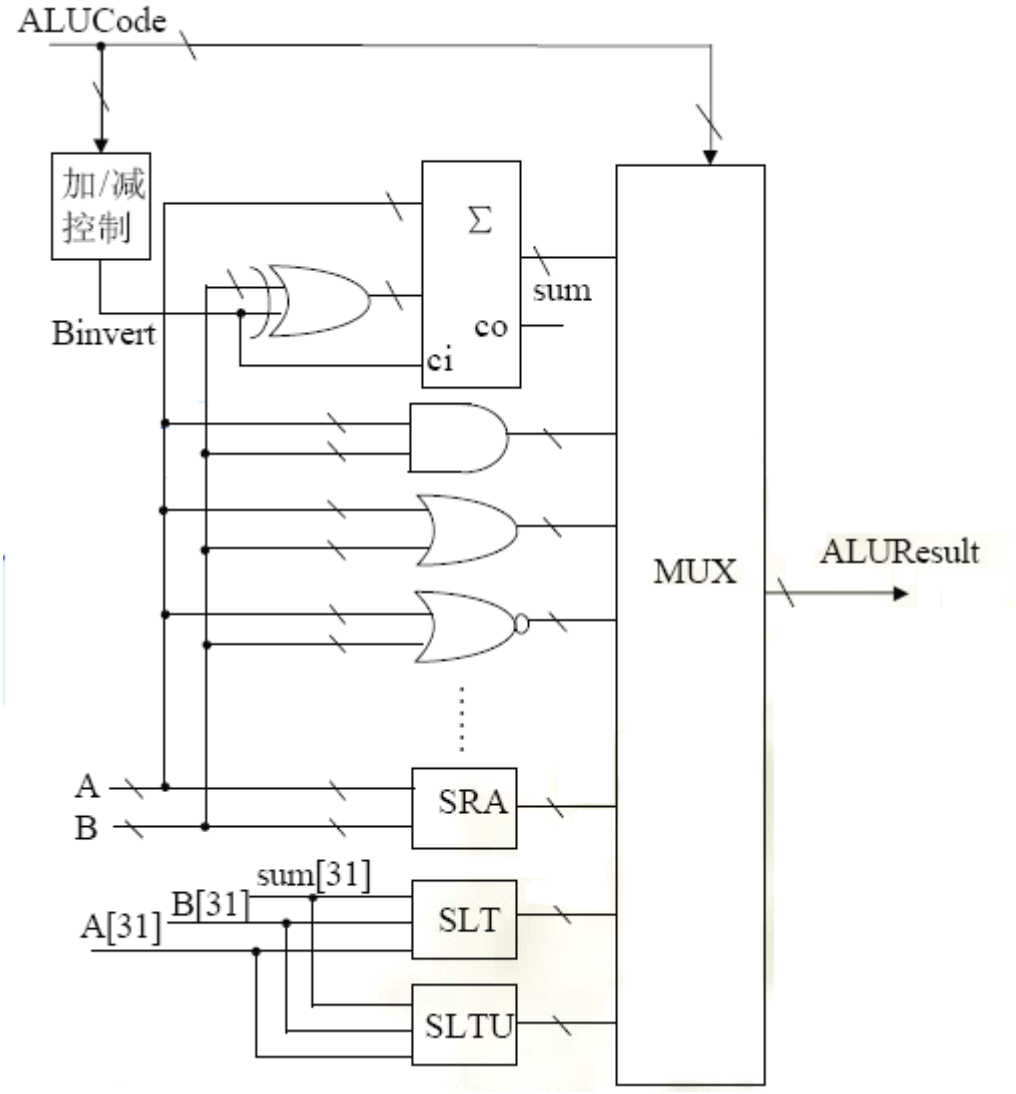


图 3.16 ALU 的基本结构

a) 加、减电路的设计考虑

减法、比较（SLT、SLTI）及部分分支指令（BEQ、BNE）均可用加法器和必要的辅助电路来实现。图 3.16 中的 Binvert 信号控制加减运算：

Binvert=~(ALUCode==alu_add)

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

加法器完成的功能为：

sum=A+B^{32{Binvert}}+Binvert

当 Binvert=0 时，sum=A+B^0+0= A+B；

当 Binvert=1 时，sum=A+B^{32’b1}+1= A-B；（B^{32’b1}即对 B 按位取反）

即可完成加减运算。

由于 32 位加法器的运算速度影响着 CPU 频率的高低，因此设计一个高速加法器尤

为重要，本实验采用 lab7 中介绍的 32 位进位选择加法器。

b) 比较电路的设计考虑

对于比较运算，如果最高为不同，即 A[31]≠B[31]，则根据 A[31]、B[31]决定比较结果，但应注意有符号数和无符号数比较运算的区别。

①在有符号数比较 SLT 运算中，判断 A<B 的方法为：

若 A 为负数、B 为 0 或正数：A[31]&&(~B[31])

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

则

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

②在无符号数比较 SLT 运算中，判断 A<B 的方法为：

若 A 最高位为 0、B 最高位为 1：(~A[31])&&B[31]

若 A、B 最高位相同，A-B 为负：(A[31]~^B[31]) && sum[31]

则

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

c) 算术右移运算电路的设计考虑

Verilog HDL 的算术右移的运算符为“<<<”。要实现算术右移应注意，被移位的对象必须定义为 reg 类型，但是在 SRA 指令，被移位的对象操作数 B 为输入信号，不能定义为 reg 类型，因此必须引入 reg 类型中间变量 B_reg，相应的 Verilog HDL 语句为：

```
reg signed [31:0] B_reg;
always @(B) begin
        B_reg = B;   end
```

引入 reg 类型的中间变量 B_reg 后，就可对 B_reg 进行算术右移操作。

d) 逻辑运算

与、或、或非、异或、逻辑移位等运算较为简单，只是要注意一点，ANDI、XORI、ORI 三条指令的立即数为 16 位无符号数，应“0 扩展”为 32 位无符号数，在运算的同时完成“0 扩展”。如：ADDI 指令的运算为 A&{16'b0，B[15:0]}。

(2) 转发电路 Forwarding 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，其含义如下表：

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

由前面介绍的一、二阶数据相关判断条件，不难得到：

```
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);
```

$$\begin{cases} \text{ForwardB[0]=RegWrite\_wb\&\&(RegWriteAddr\_wb!=0)} \\ \qquad\qquad \text{\&\&(RegWriteAddr\_mem!=RtAddr\_ex)} \\ \qquad\qquad \text{\&\&(RegWriteAddr\_wb==RtAddr\_ex);} \\ \text{ForwardB[1]=RegWrite\_mem\&\&(RegWriteAddr\_mem!=0)} \\ \qquad\qquad \text{\&\&(RegWriteAddr\_mem==RtAddr\_ex);} \end{cases}$$

5. 存储器访问 MEM 模块的设计

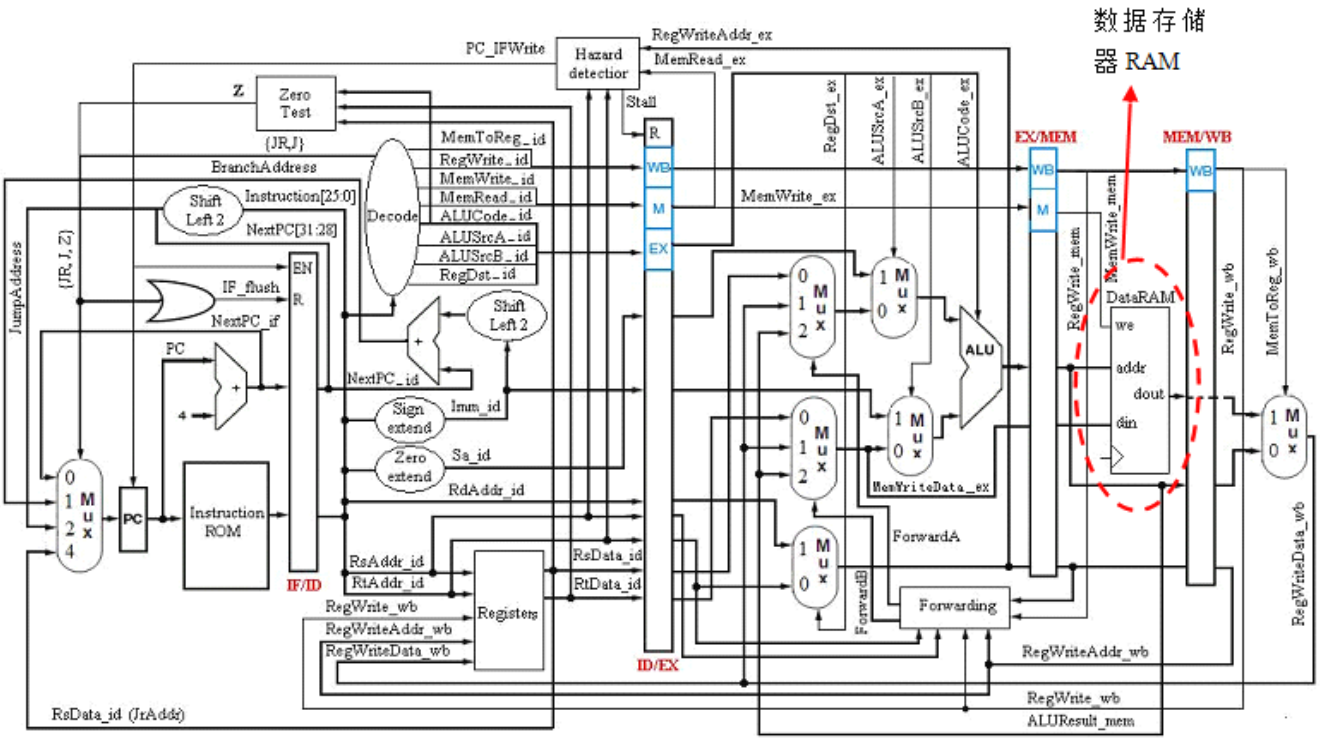


图 3.17 MEM 级电路的结构

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。故我们设计的数据 RAM 的地址应该接的信号是 ALUResult_mem[7:2]。

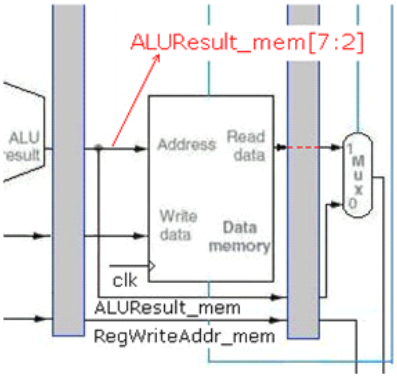


图 3.18 RAM 输入输出信号

由于 VIRTEXII PRO 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。图 3.17 与图 3.18 中的虚线就是表示这种含义.

6. 写回级 WB 模块的设计

WB 部分非常简单，只有一个数据选择器选择写回的数据来源,可在顶层模块中直接实现。

7. 取指令级 IF 模块的设计

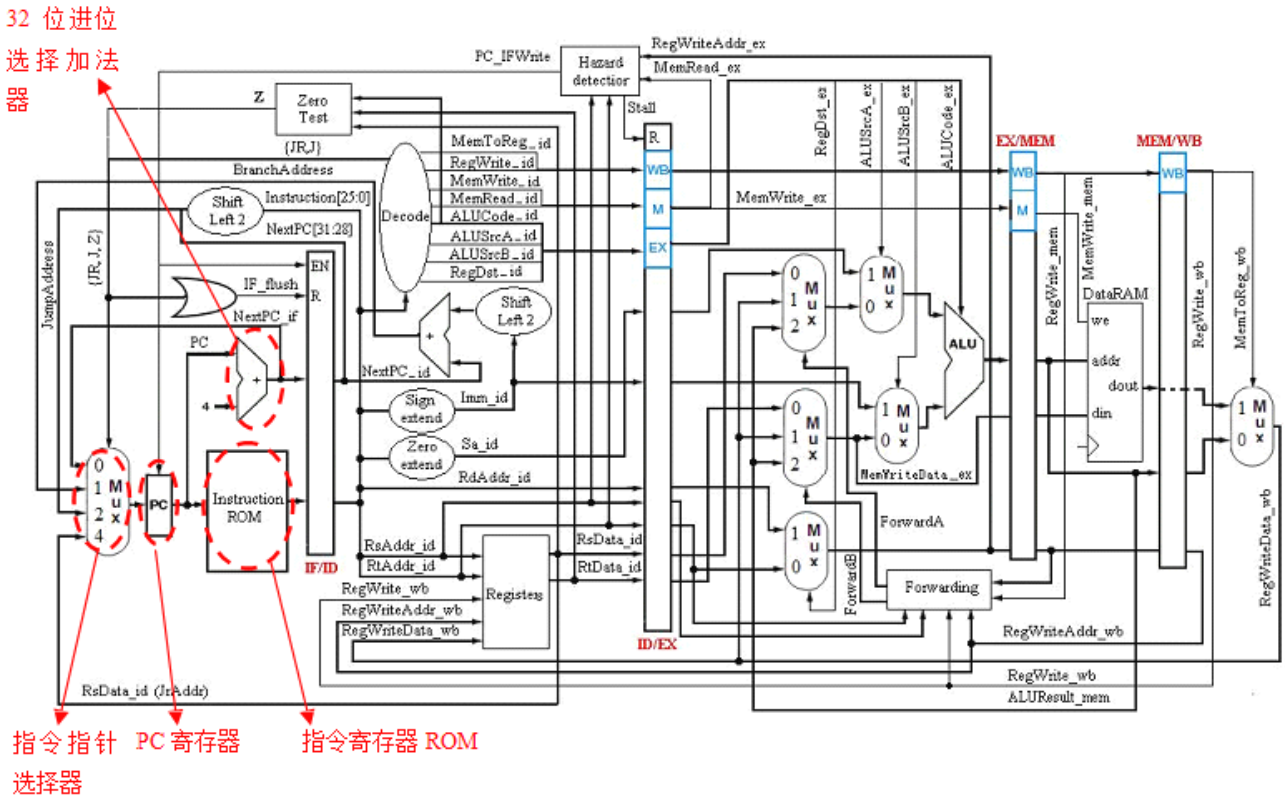


图 3.19 IF 级基本结构

IF 模块由指令指针寄存器 PC、指令存储器子模块 Instruction ROM、指令指针选择器 MUX 和一个 32 位加法器组成，IF 级模块接口信息如下表所示：

引脚名称	方向	说　　明
clk	Input	系统时钟
reset		系统复位信号，高电平有效
Z		分支指令的条件判断结果
J		跳转指令
JR		寄存器跳转指令
PC_IFWrite		阻塞流水线的信号，低电平有效
JumpAddr[31:0]		J 指令跳转地址
JrAddr[31:0]		JR 指令跳转地址
BranchAddr[31:0]		条件分支地址
Instruction_if[31:0]	Output	指令机器
NextPC_if[31:0]		下一个 PC 值

(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，课程提供一段简单测试程序的机器码，机器码存于 PipelineDemo.coe 文件中。

课程已提供该 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) 指令指针选择器

指令指针选择器为一 8 选 1 数据选择器，选择信号为 PCSource={JR，J，Z}，具体含义如下表：

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

(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。

8. 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。

9. 顶层文件的设计

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

三、实验代码

见附录二。

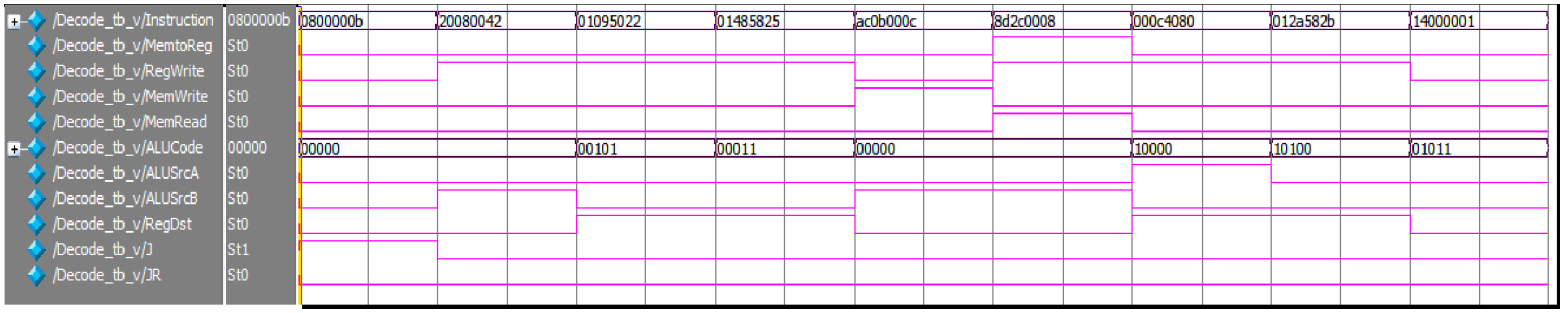
四、实验设备

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、实验仿真结果与分析

1. ID 级仿真

(1) Decode 子模块的仿真



测试指令为：

Instruction = 32'h0800000b;　　//j later(later address is 2Ch)

跳转指令，J 信号为“1”，其他控制信号为“0”。

Instruction = 32'h20080042;　　//addi $t0,$0,42

加立即数，ALUCode=00000，ALUScrB 选择立即数即为“1”；同时它需要写回，则 RegWrite 信号为“1”，其他控制信号为“0”。

Instruction = 32'h01095022;　　//sub $t2,$t0,$t1

减法，ALUCode=00101；从寄存器中取值，ALUScrA 与 ALUScrB 都为“0”；需要写回，则 RegWrite 信号为“1”；写回地址是 RdAddr，RegDst=1；其他控制信号为“0”。

Instruction = 32'h01485825;　　//or $t3,$t2,$t0

逻辑或运算，ALUCode=00011；从寄存器中取值，ALUScrA 与 ALUScrB 都为“0”；需要写回，则 RegWrite 信号为“1”；写回地址是 RdAddr，RegDst=1；其他控制信号为“0”。

Instruction = 32'hac0b000c;　　//sw $t3,0C($0)

SW 指令，MemWrite=1；它属于 I 型指令，ALUScrB 选择立即数即为“1”；其他控制信号为“0”.

Instruction = 32'h8d2c0008;　　//lw $t4,08($t1)

LW 指令，MemtoReg=1，Memread=1；需要写回，则 RegWrite 信号为“1”；它属于 I 型指令，ALUScrB 选择立即数即为“1”；其他控制信号为“0”。

Instruction = 32'h000c4080;　　//sll $t0,$t4,2

移位运算指令，ALUCode=10000；ALUScrA 选择位移数 sa，即为“1”；需要写回，则 RegWrite 信号为“1”；写回地址是 RdAddr，RegDst=1；其他控制信号为“0”。

Instruction = 32'h012a582b ;　　//sltu $t3,$t1,$t2

SLTU 指令，ALUCode=10100；需要写回，则 RegWrite 信号为“1”；写回地址是 RdAddr，RegDst=1；其他控制信号为“0”。

Instruction = 32'h14000001 ;　//bne $0,$0,end　　(end address is 34h)

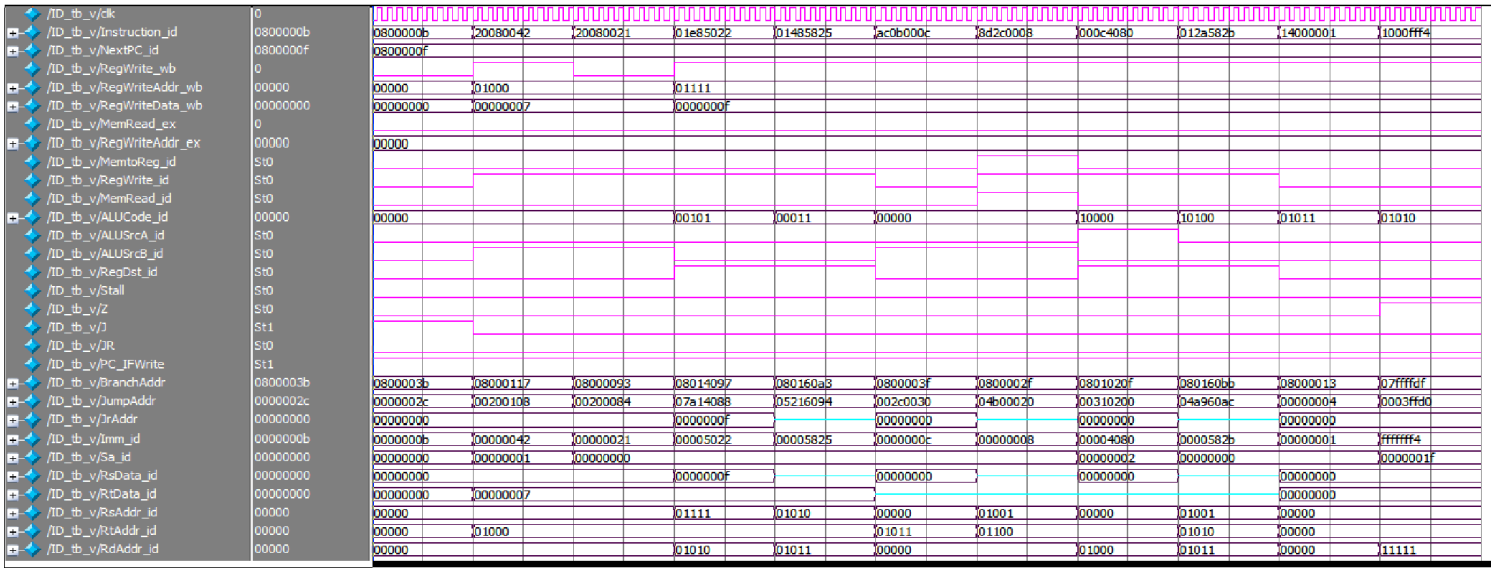
分支条件判断 BNE 指令，ALUCode=01011；其他控制信号为“0”。

Instruction = 32'h1000fff4 ;　//beq $0,$0,earlier　(earlier address is 4h)

分支条件判断 BEQ 指令，ALUCode=00010；其他控制信号为“0”.

由此判断该模块符合设计要求。

（2）ID 模块仿真

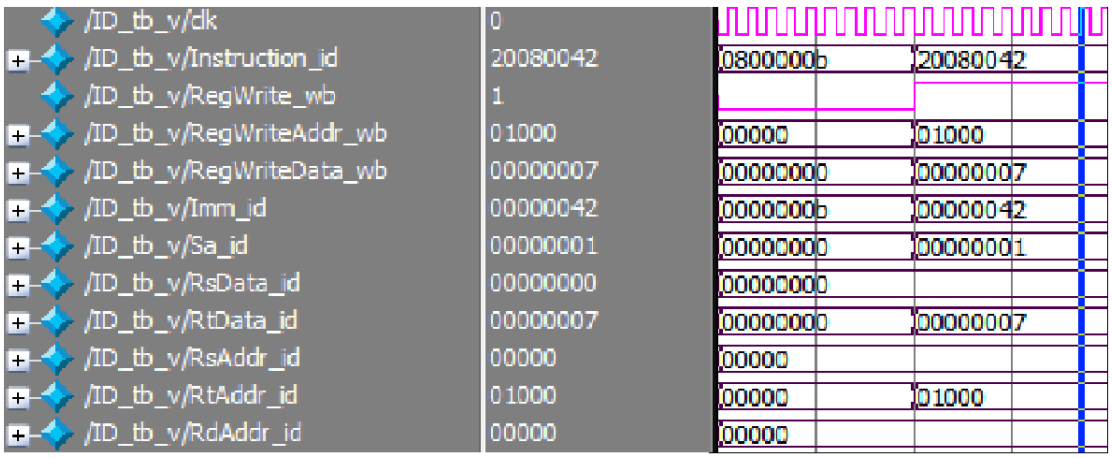


测试指令仍为 Decode 模块的测试指令，由于已确定 Decode 模块设计正确，故只观察出 Decode 输出信号外的信号。

a) Instruction = 32'h20080042;　//addi $t0,$0,42

加立即数，定义控制信号 RegWrite_wb=1；RegWriteAddr_wb=5'b01000；RegWriteData_wb=32'b111。

可看到 Imm=00000042 已经无符号扩展，写回地址 RtAddr= RegWriteAddr_wb=5'b01000，写回数据 RtData= RegWriteData_wb=32'b111

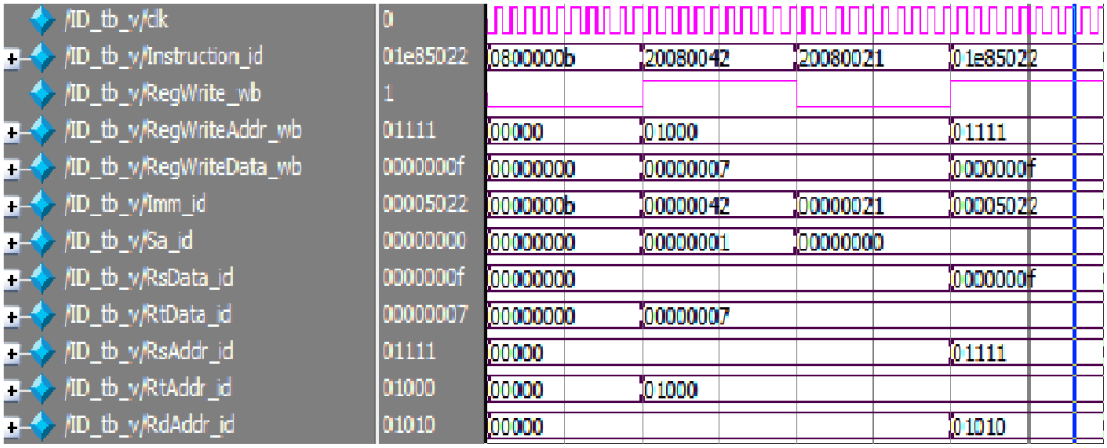


b) Instruction_id = 32'h20080021;　//addi $t1,$0,21

Instruction_id = 32'h01e85022; //sub $t2,$t1,$t0

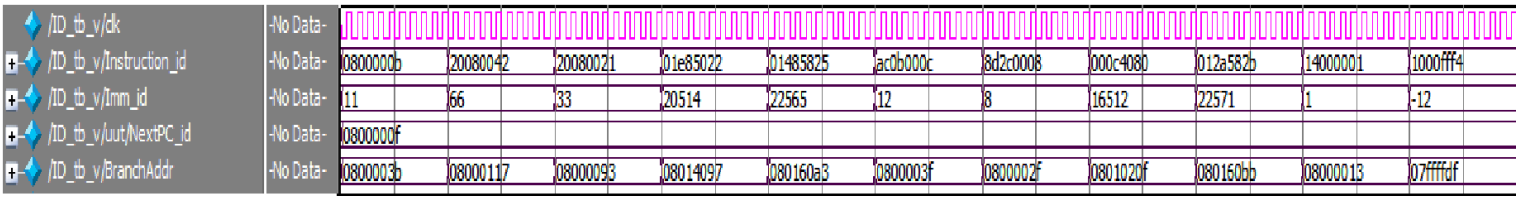
测试寄存器寄存功能。定义控制信号 RegWrite_wb=1；RegWriteAddr_wb=5'b01111；RegWriteData_wb=32'b1111。

可看到立即数 Imm，Sa 在不同指令中都进行了无符号扩展；Rs、Rt 寄存器的数据和地址都可以得到寄存，可以保持直到数据地址必须更改。由此得出寄存器堆模块 Registers 设计符合要求。



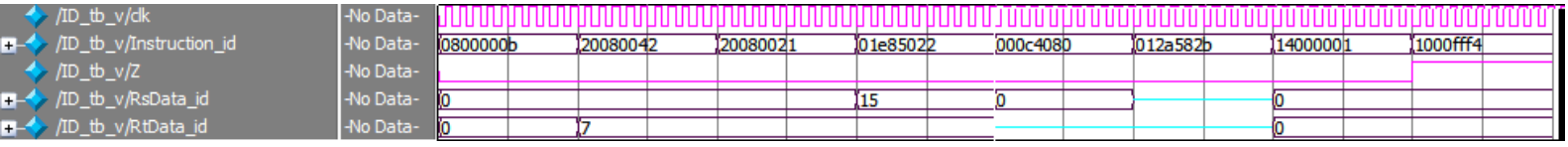
c)观察 BrandAddr 信号，可以发现其满足

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



d)观察 Z 信号可发现其满足式子

$$Z=\begin{cases} RsData[31] \,||\, \sim(|\,RsData[31:0]) \;; & ALUCode=alu\_blez \\ RsData[31] \;; & ALUCode=alu\_bltz \\ \sim RsData[31] \,\&\&\, (|\,RsData[31:0]) \;; & ALUCode=alu\_bgtz \\ \sim RsData[31]; & ALUCode=alu\_bgez \\ |\,(RsData[31:0] \,\hat{}\, RtData[31:0]) \;; & ALUCode=alu\_bne \\ \&\,(RsData[31:0] \sim\hat{}\, RtData[31:0]); & ALUCode=alu\_beq \\ 0; & ALUCode=OTHER \end{cases}$$

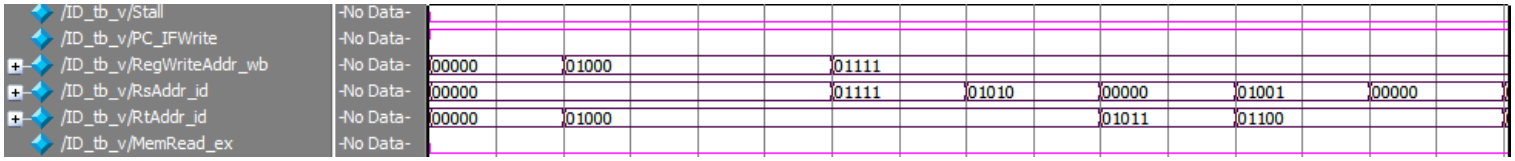


e) 观察 Stall 信号和 PC_IFWrite 信号可发现其满足关系式：

Stall=((RegWriteAddr_ex==RsAddr_id)||

(RegWriteAddr_ex==RtAddr_id))&&MemRead_ex

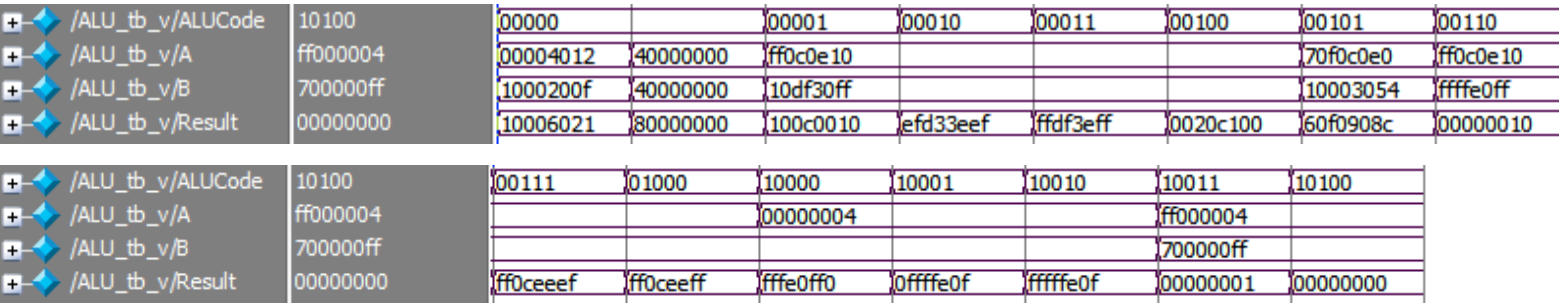
PC_IFWrite=~Stall



ID 级设计符合要求。

2. EX 级仿真

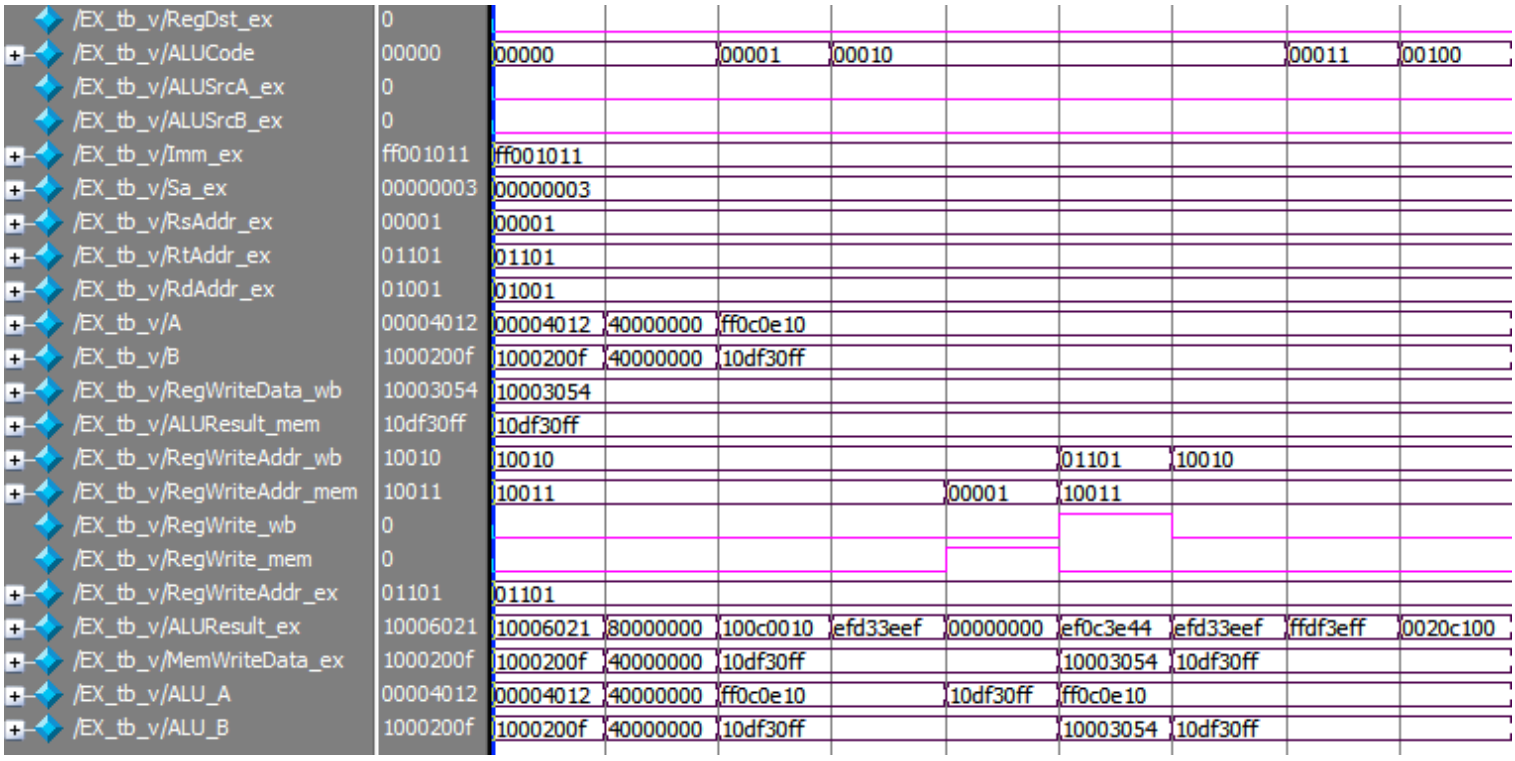
(1) ALU 子模块仿真

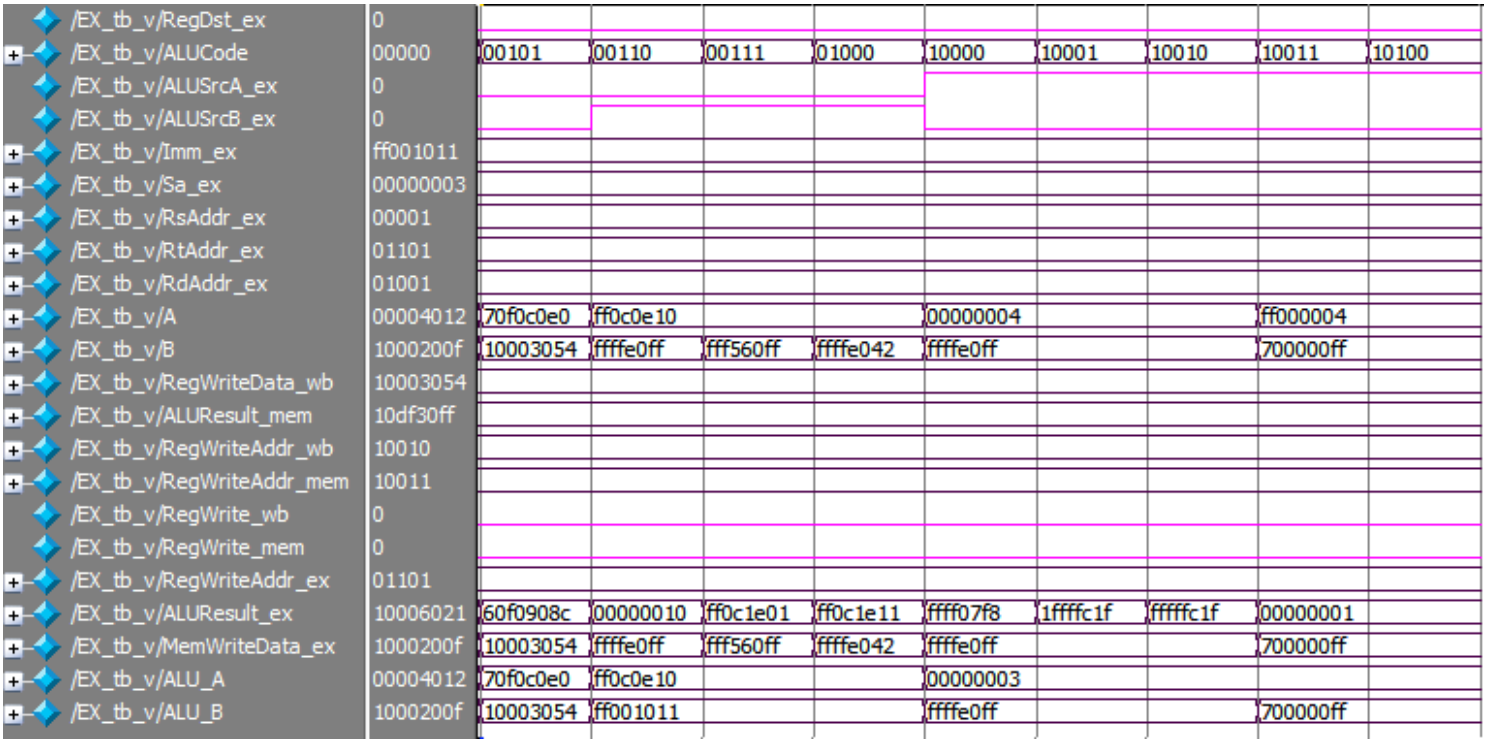


对照 ALU 功能表，仿真结果正确。说明 ALU 子模块设计符合要求。

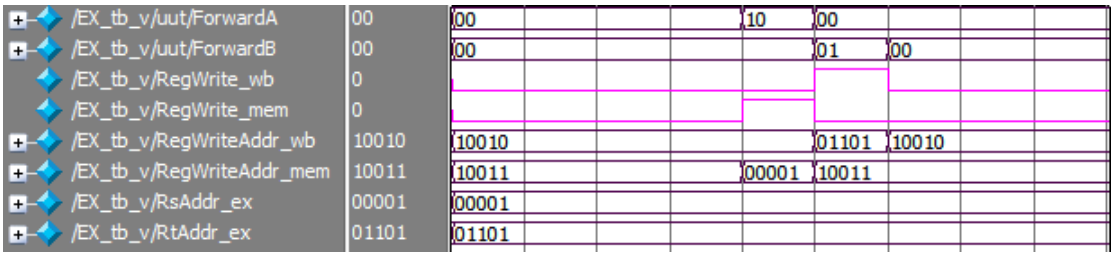
(2) EX 模块仿真

ALU 子模块功能已测试完毕，只需确认数据选择器与转发电路 Forwarding 的功能。





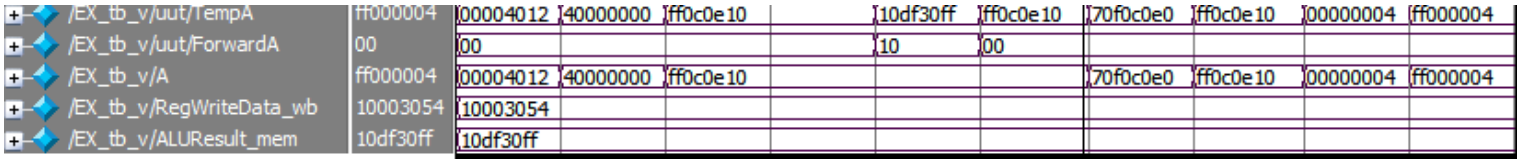
先观察 Forwarding 转发电路的信号：



信号之间的逻辑关系符合 ForwardingA 与 ForwardingB 的定义关系式。

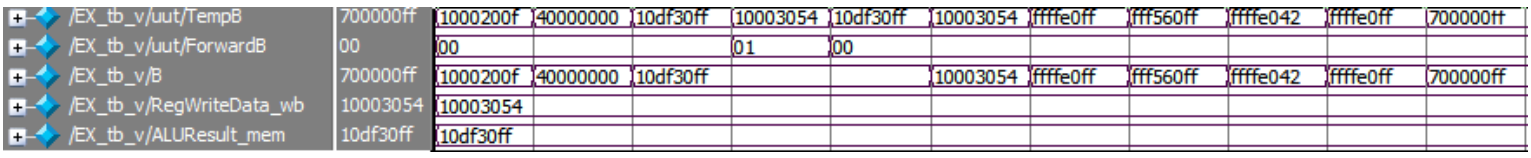
然后观察操作数 A 与 B 的数据选择器的信号：

操作数 A：



其中 TempA 为数据选择器输出信号；A 为 RsData。

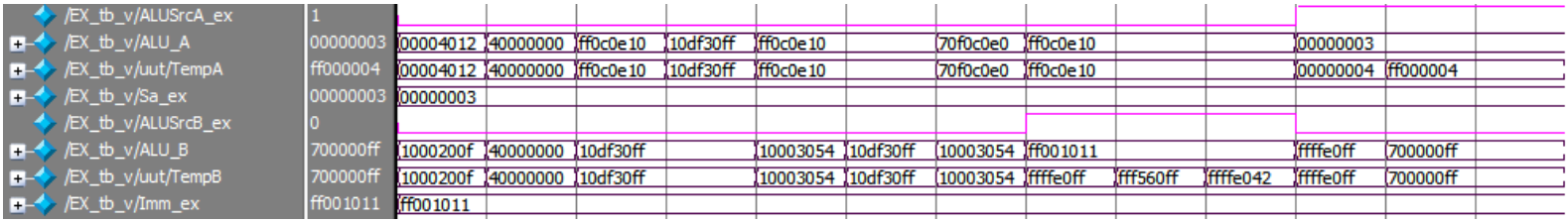
操作数 B：



其中 TempB 为数据选择器输出信号；B 为 RTData。

波形满足 Forwarding 的功能要求。

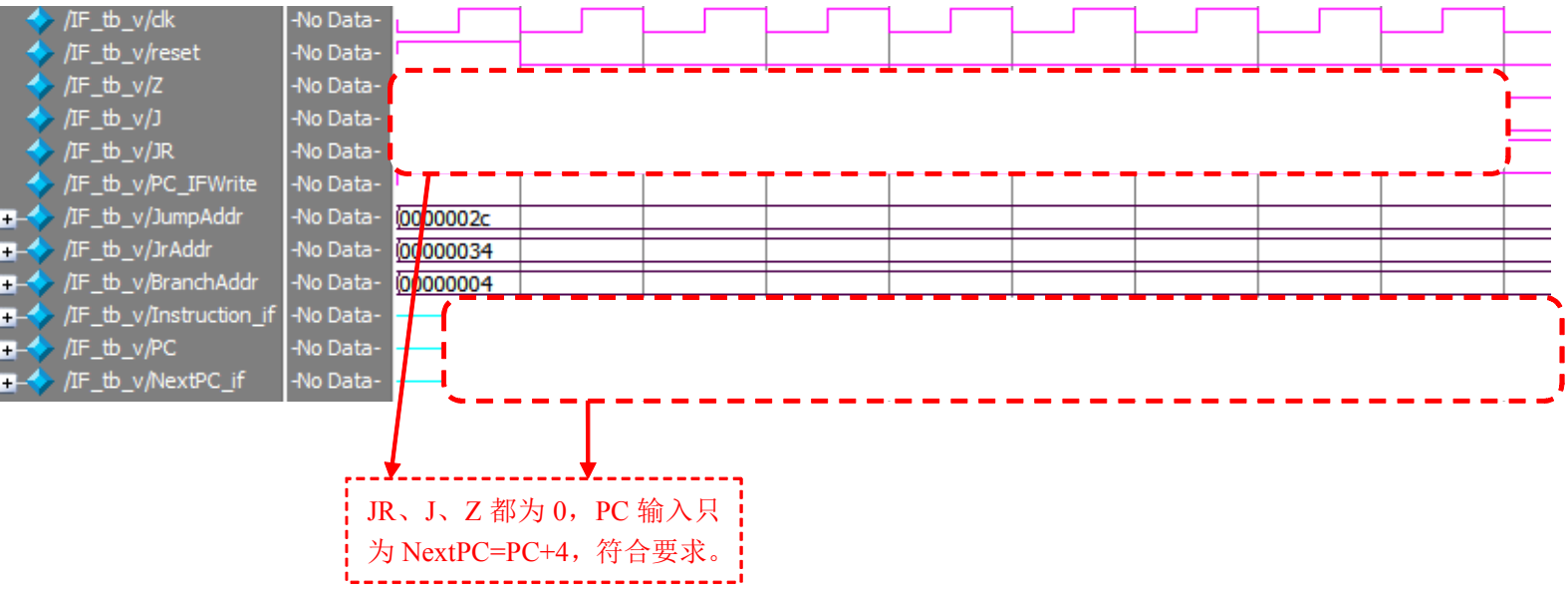
观察 ALU_A 与 ALU_B 数据选择器功能：

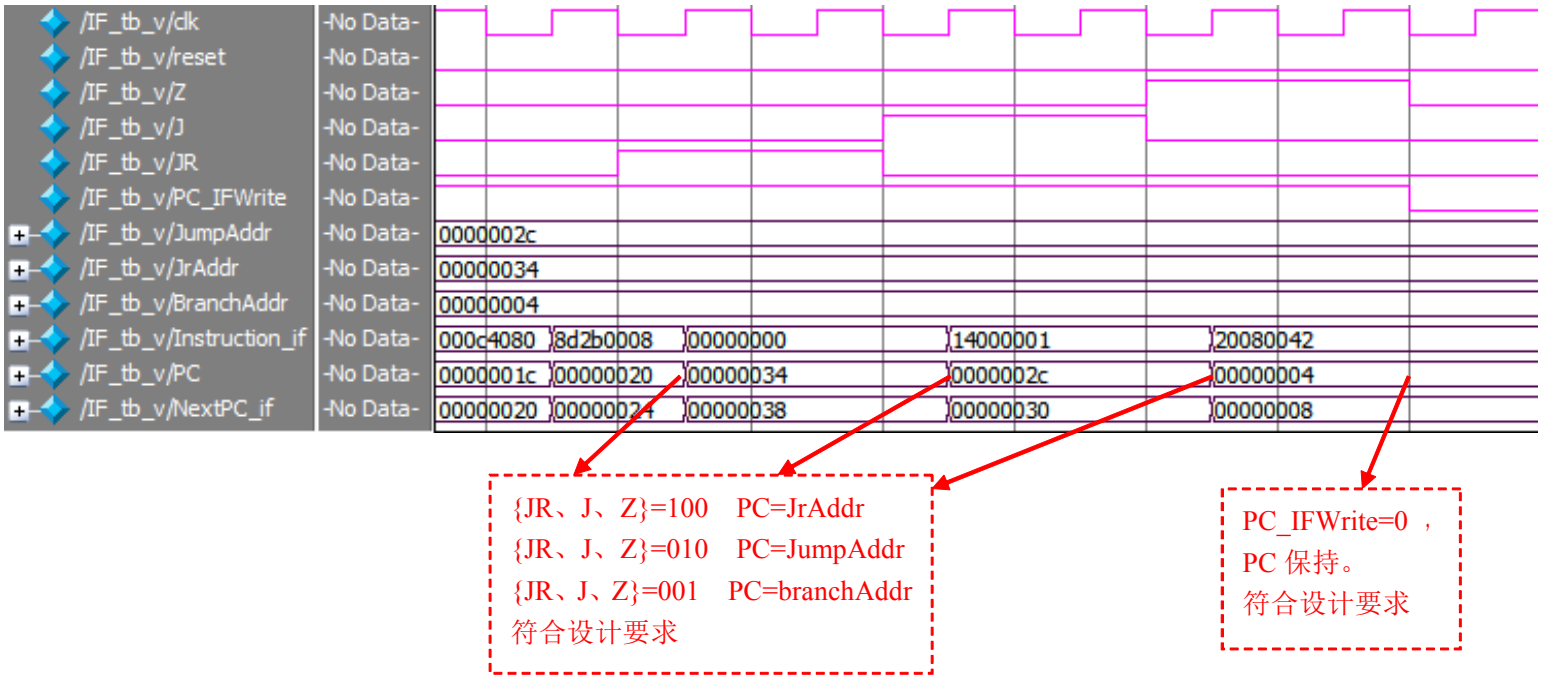


TempA 与 TempB 分别是操作数 A 与 B。从波形上可看出选择器符合设计要求。

EX 级设计符合要求。

3. IF 级仿真





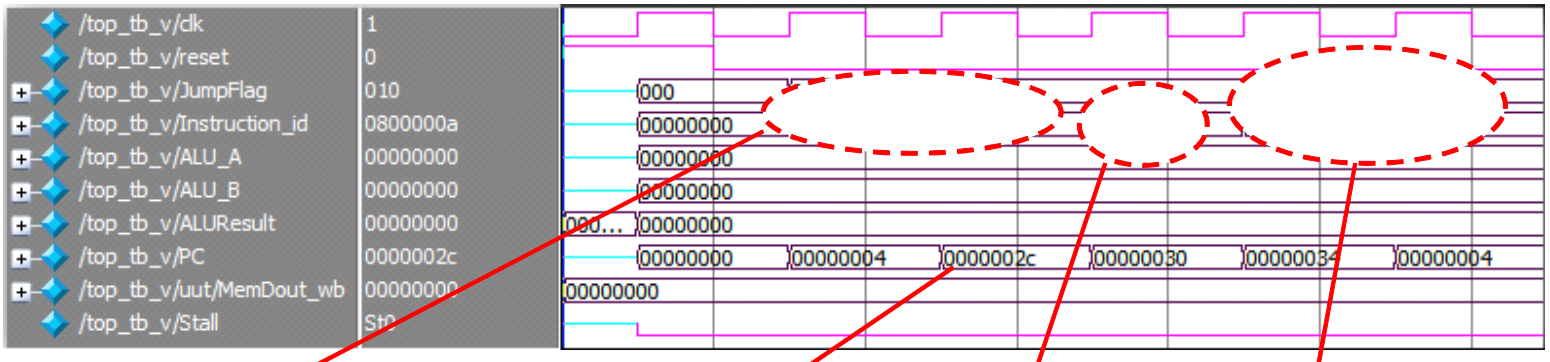
IF 级符合设计要求。

4. 顶层仿真

测试指令为：

```
dout=32'h0800000b ; //          j later
dout=32'h20080042 ; //earlier: addi $t0,$0,42
dout=32'h20090004 ; //          addi $t1,$0,4
dout=32'h01095022 ; //          sub $t2,$t0,$t1    （操作 B 一阶相关，操作 A 二阶相关）
dout=32'h01485825 ; //          or $t3,$t2,$t0     （操作 A 一阶相关，操作 B 三阶相关）
dout=32'hac0b000c ; //          sw $t3,0c($0)
dout=32'h8d2c0008 ; //          lw $t4,08($t1)
dout=32'h000c4080 ; //          sll $t0,$t4,2      （数据冒险）
dout=32'h8d2b0008 ; //          lw $t3,08($t1)
dout=32'h012a582b ; //          sltu $t3,$t1,$t2
dout=32'h0800000a ; //done:    j done
dout=32'h14000001 ; //later:    bne $0,$0,end      （分支条件不成立）
dout=32'h1000fff4 ; //          beq $0,$0,earlier  （分支条件成立）
dout=32'h00000000 ; //end:     nop
```

JumpFlag={JR、J、Z}

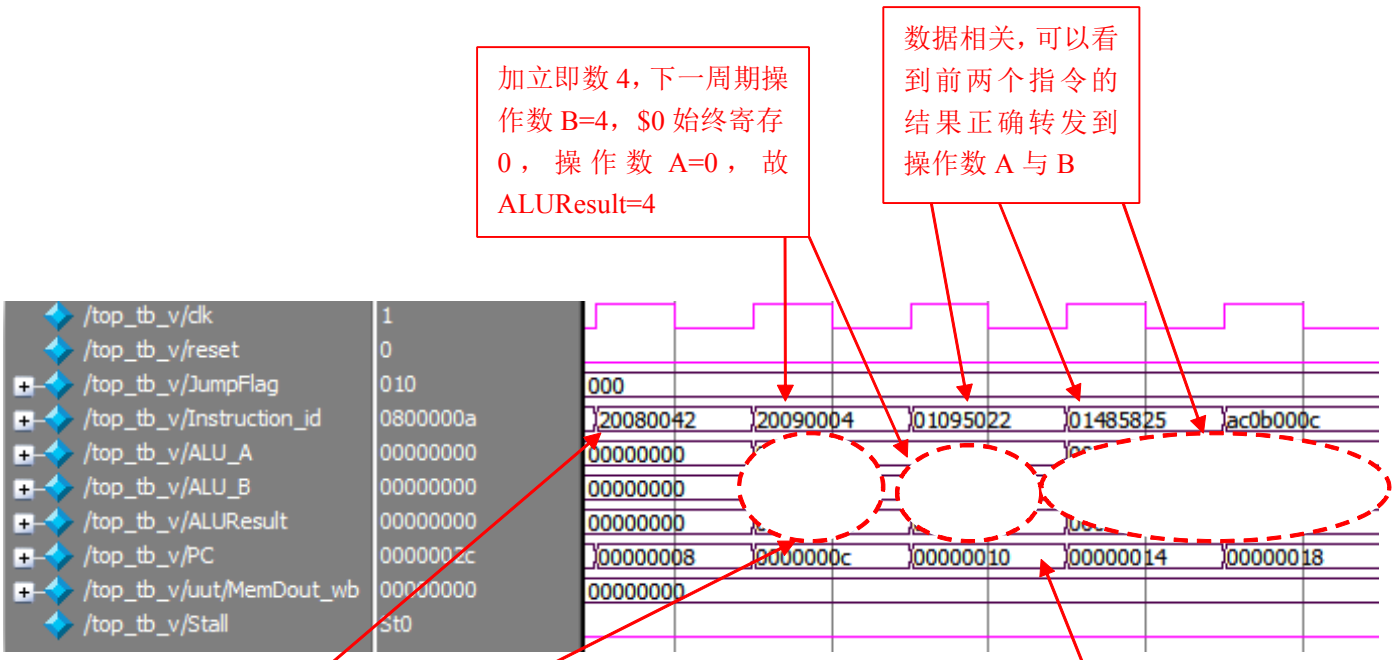


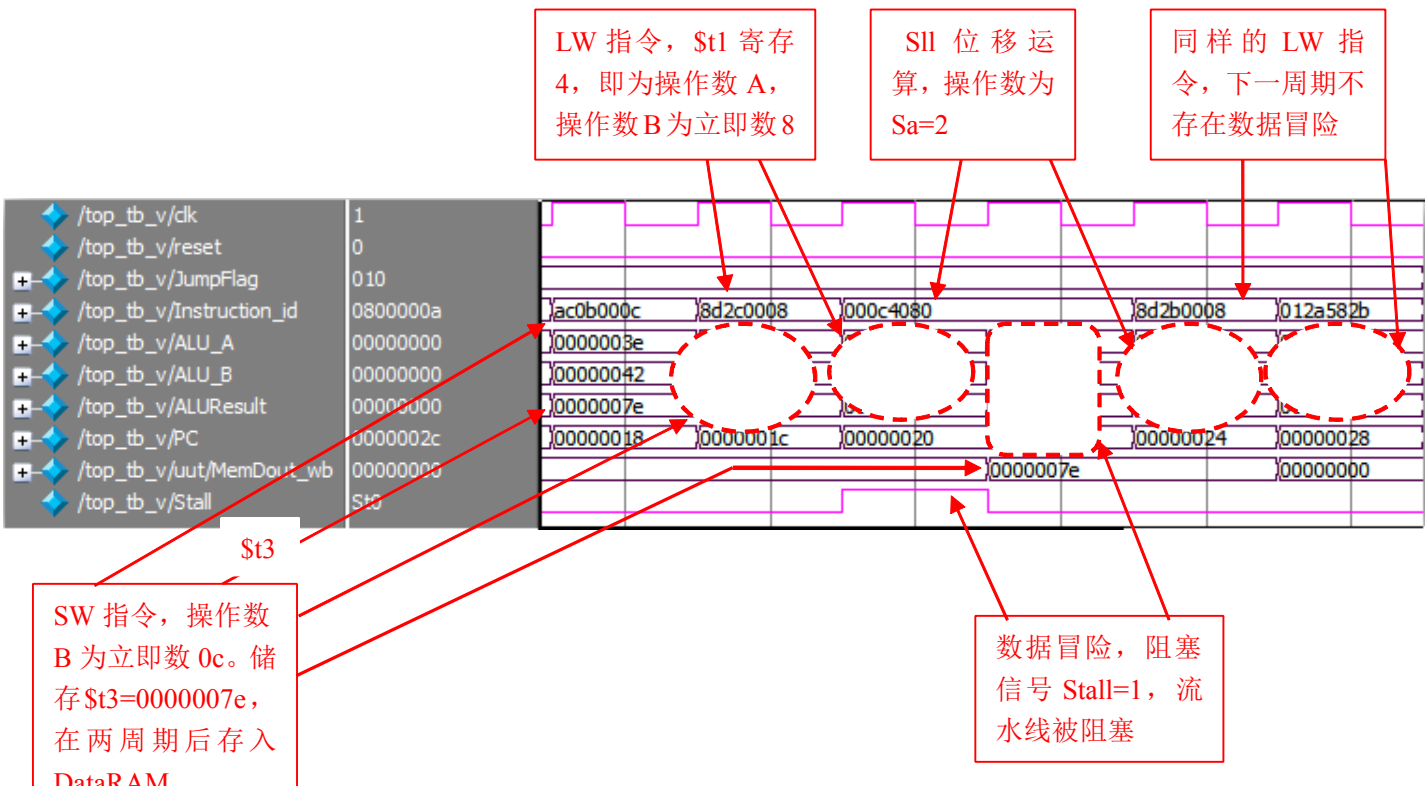
初始指令为 J 指令，J=1，所以 IF_flash=1，清空 IF/ID 级寄存器，下一周期指令=0.

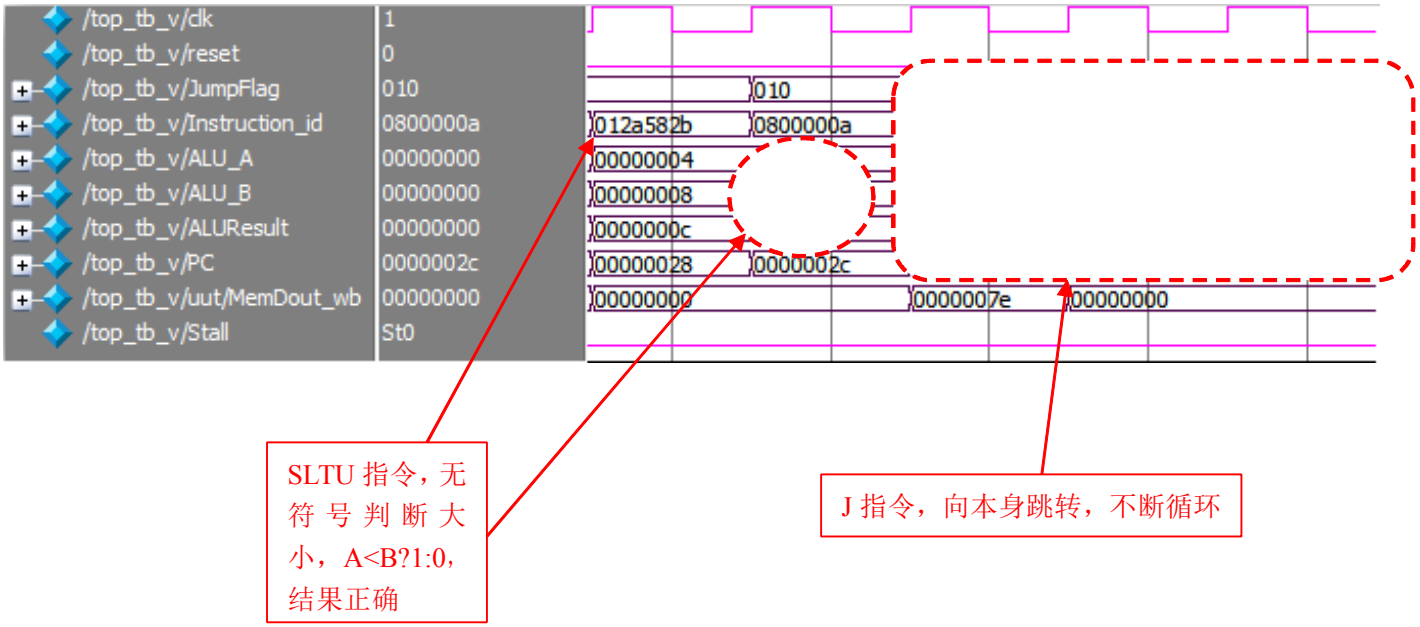
J=1，使 PC 跳转

bne 分支不成立

beq 分支成立，Z=1，IF_flash=1，清空 IF/ID 级寄存器，下一周期指令=0.


加立即数 4，下一周期操作数 B=4，$0 始终寄存 0，操作数 A=0，故 ALUResult=4
数据相关，可以看到前两个指令的结果正确转发到操作数 A 与 B
/top_tb_v/clk
/top_tb_v/reset
/top_tb_v/JumpFlag
/top_tb_v/Instruction_id
/top_tb_v/ALU_A
/top_tb_v/ALU_B
/top_tb_v/ALUResult
/top_tb_v/PC
/top_tb_v/uut/MemDout_wb
/top_tb_v/Stall
20080042
20090004
01095022
01485825
ac0b000c
00000008
0000000c
00000010
00000014
00000018
加立即数 42，下一周期操作数 B=42，$0 始终寄存 0，操作数 A=0，故 ALUResult=42
没有跳转，可以看到 PC 正常增加


LW 指令，$t1 寄存 4，即为操作数 A，操作数 B 为立即数 8
Sll 位移运算，操作数为 Sa=2
同样的 LW 指令，下一周期不存在数据冒险
/top_tb_v/clk
/top_tb_v/reset
/top_tb_v/JumpFlag
/top_tb_v/Instruction_id
/top_tb_v/ALU_A
/top_tb_v/ALU_B
/top_tb_v/ALUResult
/top_tb_v/PC
/top_tb_v/uut/MemDout_wb
/top_tb_v/Stall
ac0b000c
8d2c0008
000c4080
8d2b0008
012a582b
0000003e
00000042
0000007e
00000018
0000001c
00000020
00000024
00000028
0000007e
00000000
$t3
SW 指令，操作数 B 为立即数 0c。储存$t3=0000007e，在两周期后存入 DataRAM
数据冒险，阻塞信号 Stall=1，流水线被阻塞



通过顶层仿真分析，证明流水线寄存器与各模块之间的链接正确。

至此，流水线 MIPS 微处理器设计全部符合要求。

六、综合约束实现

打开 PipelineCPU_VGA 中的工程文件，将已完成的代码添加到工程文件中，对工程进行综合、约束、实现，下载到 XUP Virtex-Ⅱ Pro 开发实验板中。接入 SVGA 显示器，复位，每按一下 UP 键，MIPS CPU 运行一步。观察显示器显示的结果，结果正确。具体操作步骤不再赘述。

七、实验心得

本次实验代码编写部分较为简单，实验指导中已经很明确的给出了各个模块中的工作原理与信号之间的逻辑关系，但由于连接信号众多，信号名称相近，很容易由于录入粗心而产生错误，这也是本次实验最主要的错误来源。

我认为本次实验的难点在于对流水线 CPU 多种工作状态的整体把握，以及顶层仿真中的纠错工作。流水线 CPU 速度更快，多条指令同时运行，这就要求我们对不同指令的运行以及数据的传递有一定的认识。在理论课程中已经对流水线 CPU 有了一定的了解学习，通过实验可以更直观更透彻的学习其具体流程和实现方式。

由于各子模块的结构清晰、原理明确，仿真中出现的错误也很容易纠正，但在顶层模块中存在各级数据转发，同一指令各级运行时间不同，错误的排查有一定困难。实验中我出现的错误也主要在顶层仿真中。出现错误时我一般会先观察错误波形所对应的输入波形是否正确，若正确再顺着输入信号向前一级排查直至信号源。如果输入信号全部正常，则问题在于错误波形所在模块，仔细检查模块的各个输入输出信号之间的逻辑关系是否正确，再具体排查一遍代码中是否存在问题，基本就可以发现错误

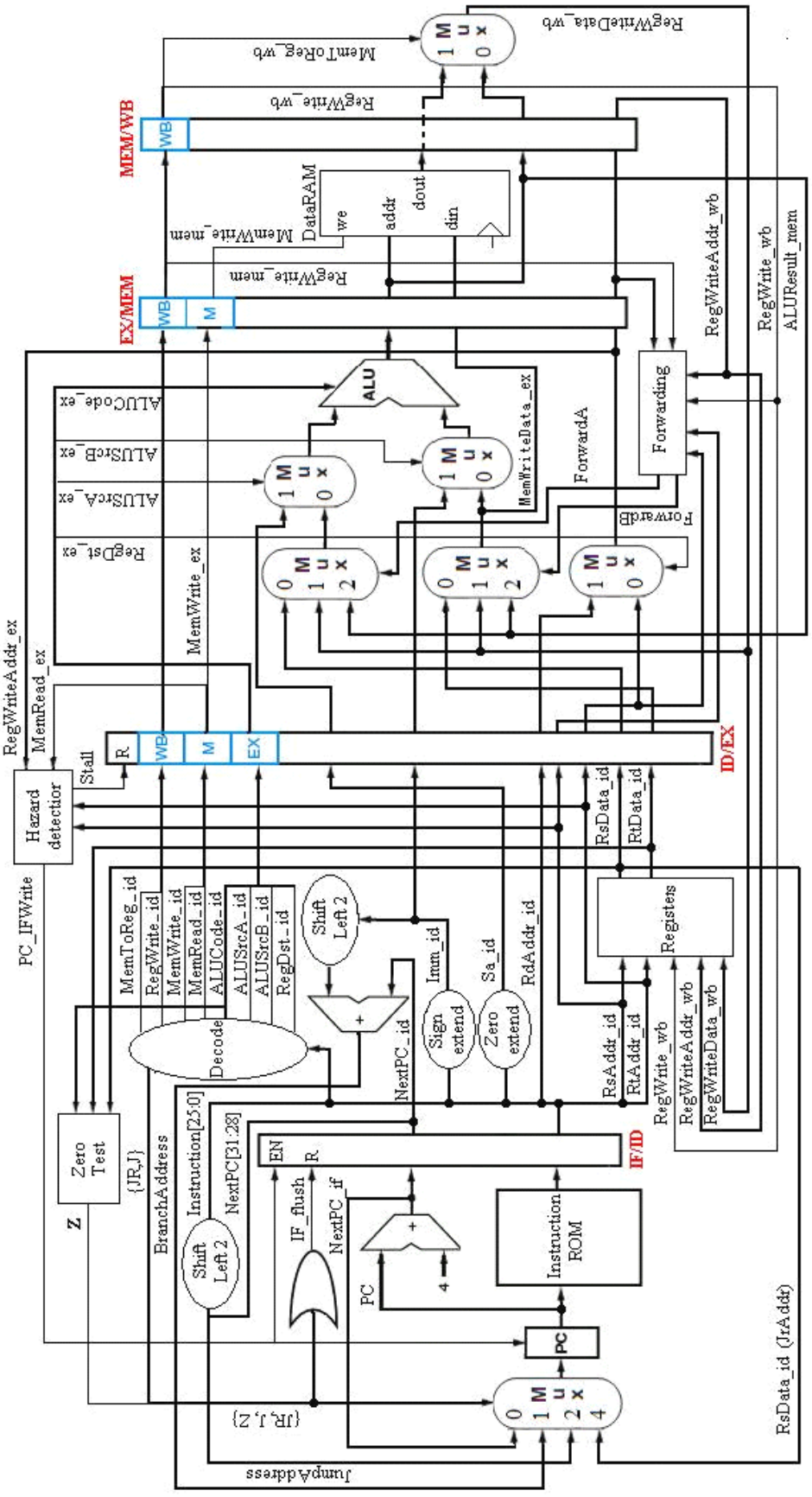
所在。我在顶层仿真时 EX 级与 MEM 级的输出信号存在红线，将其展开发现信号部分错误，检查输入信号也是同样问题。继续向前一级检查发现 ID/EX 寄存器与 EX/MEM 寄存器的输出信号位数不对，查看顶层设计代码发现编写寄存器时个别 D 触发器的输入输出信号位数没有定义，改正后顶层仿真就正确了。

在 ISE 工程文件中同样要注意信号参数的传递，工程中时钟信号有一定变化，模块连接过程中也要注意传递信号的名称是否与工程定义的一致。

回顾整个实验过程，最重要的是严谨细致的态度。笔误这种错误最容易避免，也最容易发生，排查起来却相当费力。只要再细心一点，实验就能更轻松一点。对于仿真软件的应用也更为熟练，学习到很多新的方便的小功能，对于分析波形很有帮助。

实验中的不足之处在于对 ISE 软件的应用还是很生涩，很多功能仍未学习到，一些错误提示也不理解，还需要进一步学习。

附录一：流水线 MIPS 微处理器的原理框图



附录二：部分代码

1. Decode 中 ALUCode 状态机

```
reg [4:0] ALUCode;
 always @(*)
 begin
    if(op==R_type_op)
       begin
         case(funct)
            ADD_funct: ALUCode<=alu_add;
            ADDU_funct: ALUCode<=alu_add;
            AND_funct: ALUCode<=alu_and;
            XOR_funct: ALUCode<=alu_xor;
            OR_funct: ALUCode<=alu_or;
            NOR_funct: ALUCode<=alu_nor;
           SUB_funct: ALUCode<=alu_sub;
            SUBU_funct: ALUCode<=alu_sub;
            SLT_funct : ALUCode<=alu_slt;
            SLTU_funct : ALUCode<=alu_sltu;
           SLL_funct: ALUCode<=alu_sll;
            SLLV_funct: ALUCode<=alu_sll;
            SRL_funct: ALUCode<=alu_srl;
            SRLV_funct: ALUCode<=alu_srl;
            SRA_funct: ALUCode<=alu_sra;
            default: ALUCode<=alu_sra;
          endcase
       end
     else
       begin
        case(op)
           BEQ_op: ALUCode<=alu_beq;
           BNE_op: ALUCode<=alu_bne;
          BGEZ_op: begin    if(rt==BGEZ_rt) ALUCode<=alu_bgez;      end
           BGTZ_op: begin    if(rt==BGTZ_rt) ALUCode<=alu_bgtz;      end
          BLEZ_op: begin    if(rt==BLEZ_rt) ALUCode<=alu_blez;       end
          BLTZ_op: begin    if(rt==BLTZ_rt) ALUCode<=alu_bltz;       end
          ADDI_op: ALUCode<=alu_add;
           ADDIU_op: ALUCode<=alu_add;
           ANDI_op: ALUCode<=alu_andi;
           XORI_op: ALUCode<=alu_xori;
           ORI_op: ALUCode<=alu_ori;
          SLTI_op: ALUCode<=alu_slt;
          SLTIU_op: ALUCode<=alu_sltu;
          SW_op: ALUCode<=alu_add;
```

```
            LW_op: ALUCode<=alu_add;
            default: ALUCode<=alu_add;
           endcase
      end
 end
```

2. ALU

```
module ALU (    Result,ALUCode, A, B);
      input [4:0]  ALUCode;                    // Operation select
      input [31:0]       A, B;
      output [31:0]      Result;
      reg [31:0]    Result;
//*****************************************************************************
// Shift operation: ">>>" will perform an arithmetic shift, but the operand
// must be reg signed
//*****************************************************************************
      reg signed [31:0] B_reg;
      always @(B) begin            B_reg = B; end
// Decoded ALU operation select (ALUsel) signals
          //略
//*****************************************************************************
// ALU Result datapath
//*****************************************************************************
   wire [31:0] sum,B1;
   wire Binvert;
   assign Binvert=~(ALUCode==alu_add);
   assign B1=B^{32{Binvert}};
   adder_32bits add( .a(A), .b(B1), .ci(Binvert), .s(sum), .co());

   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'd0,B[15:0]};
         alu_xori: Result<=A^{16'd0,B[15:0]};
         alu_ori: Result<=A|{16'd0,B[15:0]};
         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]);
            alu_jr: Result<=A;
            default: Result<=32'b0;
          endcase
        end
      endmodule
```

3. 顶层模块

```
module MipsPipelineCPU(clk, reset, JumpFlag, Instruction_id, ALU_A, ALU_B, ALUResult, PC,
                       RegWriteData_wb,Stall);
    input clk;
    input reset;
    output[2:0] JumpFlag;
    output [31:0] Instruction_id;
    output [31:0] ALU_A;
    output [31:0] ALU_B;
    output [31:0] ALUResult;
    output [31:0] PC;
     output [31:0] RegWriteData_wb;
    output Stall;
//IF   module
  wire[31:0] Instruction_id;
     wire PC_IFWrite,J,JR,Z,IF_flush;
  wire[31:0] JumpAddr,JrAddr,BranchAddr,NextPC_if,Instruction_if;
     assign JumpFlag={JR,J,Z};
     assign IF_flush=Z || J ||JR;

     IF IF(
    //input
      .clk(clk), .reset(reset),
      .Z(Z), .J(J), .JR(JR),
      .PC_IFWrite(PC_IFWrite),
      .JumpAddr(JumpAddr),
      .JrAddr(JrAddr),
      .BranchAddr(BranchAddr),
    //   output
      .Instruction_if(Instruction_if),
      .PC(PC),
      .NextPC_if(NextPC_if));

//    IF->ID Register
  wire [31:0] NextPC_id;
```

```
    dffre #(32) dffre1( .d(Instruction_if), .en(PC_IFWrite), .r(IF_flush|reset), .clk(clk),
                        .q(Instruction_id));
    dffre #(32) dffre2( .d(NextPC_if), .en(PC_IFWrite), .r(IF_flush|reset), .clk(clk),
                       .q(NextPC_id));
//    ID Module
     wire[4:0] RtAddr_id,RdAddr_id,RsAddr_id;
     wire    RegWrite_wb,MemRead_ex,MemtoReg_id,RegWrite_id,MemWrite_id;
        wire    MemRead_id,ALUSrcA_id,ALUSrcB_id,RegDst_id,Stall;
        wire[4:0]    RegWriteAddr_wb,RegWriteAddr_ex,ALUCode_id;
        wire[31:0] RegWriteData_wb,Imm_id,Sa_id,RsData_id,RtData_id;
       ID    ID (
         .clk(clk),
            .Instruction_id(Instruction_id),
            .NextPC_id(NextPC_id),
            .RegWrite_wb(RegWrite_wb),
            .RegWriteAddr_wb(RegWriteAddr_wb),
            .RegWriteData_wb(RegWriteData_wb),
            .MemRead_ex(MemRead_ex),
           .RegWriteAddr_ex(RegWriteAddr_ex),
            .MemtoReg_id(MemtoReg_id),
            .RegWrite_id(RegWrite_id),
            .MemWrite_id(MemWrite_id),
            .MemRead_id(MemRead_id),
            .ALUCode_id(ALUCode_id),
            .ALUSrcA_id(ALUSrcA_id),
            .ALUSrcB_id(ALUSrcB_id),
            .RegDst_id(RegDst_id),
            .Stall(Stall), .Z(Z), .J(J), .JR(JR),
            .PC_IFWrite(PC_IFWrite),
            .BranchAddr(BranchAddr),
            .JumpAddr(JumpAddr),
            .JrAddr(JrAddr),
            .Imm_id(Imm_id),       .Sa_id(Sa_id),
            .RsData_id(RsData_id),
            .RtData_id(RtData_id),
            .RtAddr_id(RtAddr_id),
            .RdAddr_id(RdAddr_id),
            .RsAddr_id(RsAddr_id));
//     ID->EX    Register
    wire MemtoReg_ex,RegWrite_ex,ALUSrcA_ex,ALUSrcB_ex,RegDst_ex;
    wire [4:0] ALUCode_ex,RdAddr_ex,RsAddr_ex,RtAddr_ex;
    wire [31:0] Sa_ex,Imm_ex,RsData_ex,RtData_ex;

    dffr #(2) WB_I_E( .d({MemtoReg_id,RegWrite_id}), .r(Stall|reset), .clk(clk),
```

```
                        .q({MemtoReg_ex,RegWrite_ex}));
    dffr #(2) M_I_E( .d({MemWrite_id,MemRead_id}), .r(Stall|reset), .clk(clk),
                        .q({MemWrite_ex,MemRead_ex}));
    dffr #(8) EX_I_E( .d({ALUCode_id,ALUSrcA_id,ALUSrcB_id,RegDst_id}), .r(Stall|reset),
                        .clk(clk), .q({ALUCode_ex,ALUSrcA_ex,ALUSrcB_ex,RegDst_ex}));
    dffr #(32) Data_Sa( .d(Sa_id), .r(Stall|reset), .clk(clk), .q(Sa_ex));
    dffr #(32) Data_Imm( .d(Imm_id), .r(Stall|reset), .clk(clk), .q(Imm_ex));
    dffr #(32) Data_Rs( .d(RsData_id), .r(Stall|reset), .clk(clk), .q(RsData_ex));
    dffr #(32) Data_Rt( .d(RtData_id), .r(Stall|reset), .clk(clk), .q(RtData_ex));
    dffr #(15) Data_I_E( .d({RdAddr_id,RsAddr_id,RtAddr_id}), .r(Stall|reset), .clk(clk),
                        .q({RdAddr_ex,RsAddr_ex,RtAddr_ex}));

// EX Module
  wire[31:0] ALUResult_mem,ALUResult_ex,MemWriteData_ex;
  wire[4:0] RegWriteAddr_mem;
  wire RegWrite_mem;
  EX    EX(
  .RegDst_ex(RegDst_ex),
  .ALUCode_ex(ALUCode_ex),
  .ALUSrcA_ex(ALUSrcA_ex),
  .ALUSrcB_ex(ALUSrcB_ex),
  .Imm_ex(Imm_ex),
  .Sa_ex(Sa_ex),
  .RsAddr_ex(RsAddr_ex),
  .RtAddr_ex(RtAddr_ex),
  .RdAddr_ex(RdAddr_ex),
  .RsData_ex(RsData_ex),
  .RtData_ex(RtData_ex),
  .RegWriteData_wb(RegWriteData_wb),
  .ALUResult_mem(ALUResult_mem),
  .RegWriteAddr_wb(RegWriteAddr_wb),
  .RegWriteAddr_mem(RegWriteAddr_mem),
  .RegWrite_wb(RegWrite_wb),
  .RegWrite_mem(RegWrite_mem),
  .RegWriteAddr_ex(RegWriteAddr_ex),
  .ALUResult_ex(ALUResult_ex),
  .MemWriteData_ex(MemWriteData_ex),
  .ALU_A(ALU_A),
  .ALU_B(ALU_B));

 assign ALUResult=ALUResult_ex;

//EX->MEM
    wire MemtoReg_mem,MemWrite_mem;
```

```
    wire [31:0] MemWriteData_mem;

    dffr #(2) WB_E_M( .d({MemtoReg_ex,RegWrite_ex}), .r(reset), .clk(clk),
                              .q({MemtoReg_mem,RegWrite_mem}));
    dffr M_E_M( .d(MemWrite_ex), .r(reset), .clk(clk), .q(MemWrite_mem));
    dffr #(32) ALUResult_E_M( .d(ALUResult_ex), .r(reset), .clk(clk), .q(ALUResult_mem));
    dffr #(32) dffr_MemWriteData( .d(MemWriteData_ex), .r(reset), .clk(clk),
                                          .q(MemWriteData_mem));
    dffr #(5) dffr_RegWrite( .d(RegWriteAddr_ex), .r(reset), .clk(clk), .q(RegWriteAddr_mem));

//MEM Module
          wire[31:0] MemDout_wb;
        DataRAM      DataRAM(
        .addr(ALUResult_mem[7:2]),
        .clk(clk),
        .din(MemWriteData_mem),
        .dout(MemDout_wb),
        .we(MemWrite_mem));

//MEM->WB
    wire MemtoReg_wb;
    wire [31:0] ALUResult_wb;

    dffr #(2) WB_M_W( .d({MemtoReg_mem,RegWrite_mem}), .r(reset), .clk(clk),
                                .q({MemtoReg_wb,RegWrite_wb}));
    dffr #(32) ALUResultM_W( .d(ALUResult_mem), .r(reset), .clk(clk), .q(ALUResult_wb));
    dffr #(5) dffr_M_W( .d(RegWriteAddr_mem), .r(reset), .clk(clk), .q(RegWriteAddr_wb));
//WB
    assign RegWriteData_wb=MemtoReg_wb?MemDout_wb:ALUResult_wb;

endmodule
```