

Lab05-4

流水线处理器—冒险与stall

Ma De (马德)

made@zju.edu.cn

2020

College of Computer Science, Zhejiang University

Course Outline

- 一、实验目的
- 二、实验环境
- 三、实验目标及任务

实验目的

1. 理解流水线CPU的基本原理和组织结构
2. 掌握五级流水线的工作过程和设计方法
3. 理解流水线CPU停机的原理与解决办法
4. 设计流水线测试程序

实验环境

□ 实验设备

1. 计算机（Intel Core i5以上，4GB内存以上）系统
2. Sword 2.0/Sword4.0开发板
3. VIVADO 2017.4及以上开发工具

□ 材料

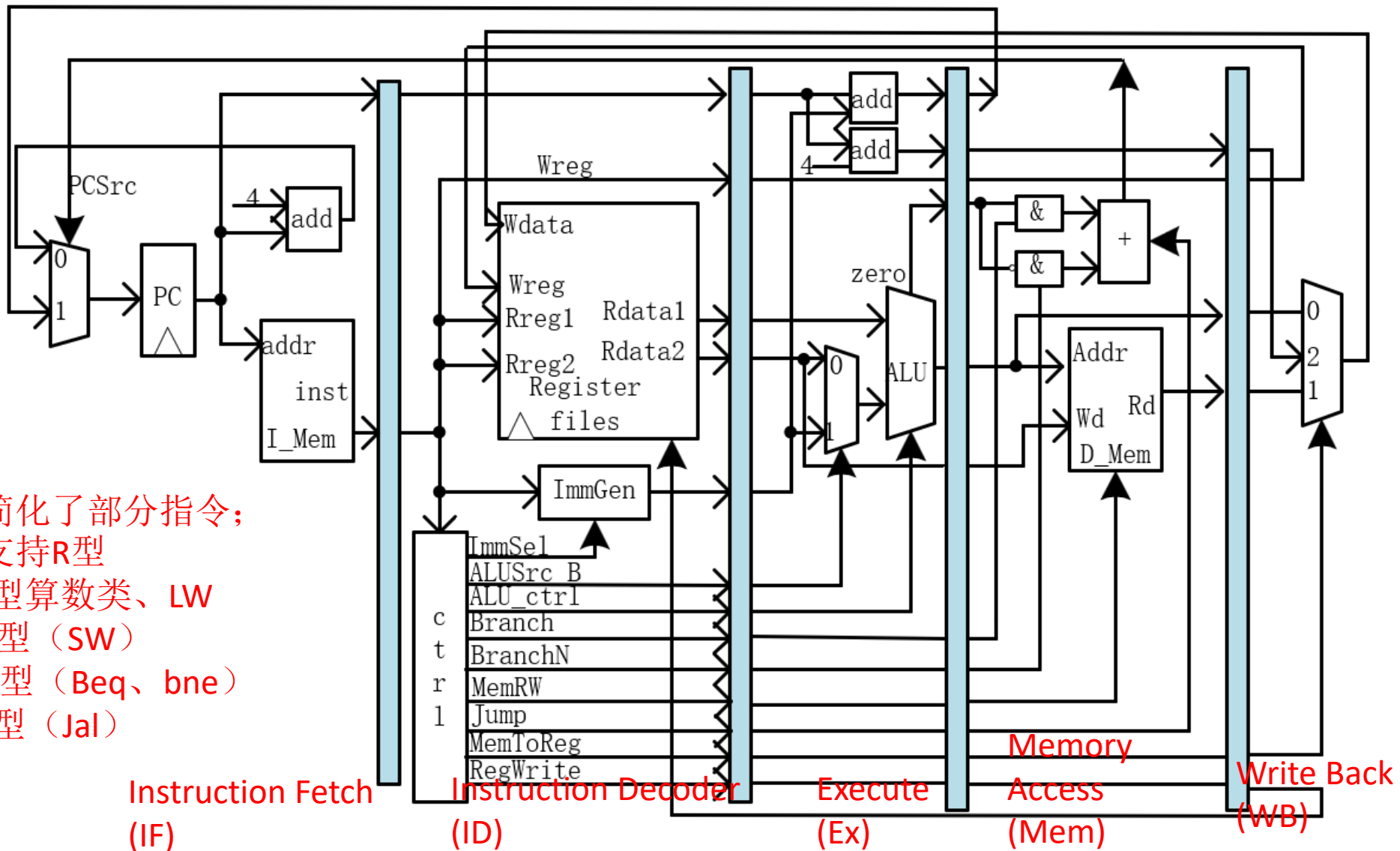
无

实验目标及任务

- **目标**：熟悉RISC-V 五级流水线的工作特点，了解流水线冒险的产生原因及解决办法，掌握IP核的使用方法，集成并测试CPU
- **任务一**：集成设计利用stall解决冒险的流水线CPU，在lab05-3的基础上完成
 - ▣ 设计冒险检测及stall消除冒险的流水线CPU
 - ▣ 替换 lab05-3的CPU为本实验集成的带stall处理的流水线CPU
- **任务二**：设计流水线测试方案并完成测试

RISC-V 流水线冒险的原理介绍

Pipelined RISC-V RV32I Datapath



简化了部分指令；
支持R型
I型算数类、LW
S型（SW）
B型（Beq、bne）
J型（Jal）

Pipelining Hazards

- **流水线冒险**：在下一个时钟周期中下一条指令无法正常执行；引起冒险的原因有多种，大致分为以下三种
 - ▣ **结构冒险**（*Structural hazard*）：硬件不支持多条指令在同一时钟周期执行
 - ▣ **数据冒险**（*Data hazard*）：当前指令的执行需要等待前一条指令的数据结果
 - ▣ **控制冒险**（*Control hazard*）：指令非顺序执行而导致下一条执行的指令不是真实期望的

Structural Hazard-- Problem

▣ 结构冒险（ *Structural hazard* ）：也称为硬件资源冲突

◎ 在流水线执行期间，两条及以上指令同一时间对同一个硬件资源发起使用的请求

◎ 因缺乏硬件支持而导致指令无法在预定的时钟周期内执行

resource
conflicts

- Memory conflicts
- Register File conflicts
- Other units conflicts

Structural Hazard-- Solution

-Memory conflicts

解决主
存资源
冲突的
方法

① 将后续的第(i+3)条指令推迟一拍进入流水线。

② 增设一个存储器。将指令和数据分别存放在两个存储器中。

③ 采用先行控制技术，或在处理器内部设置指令缓冲队列。

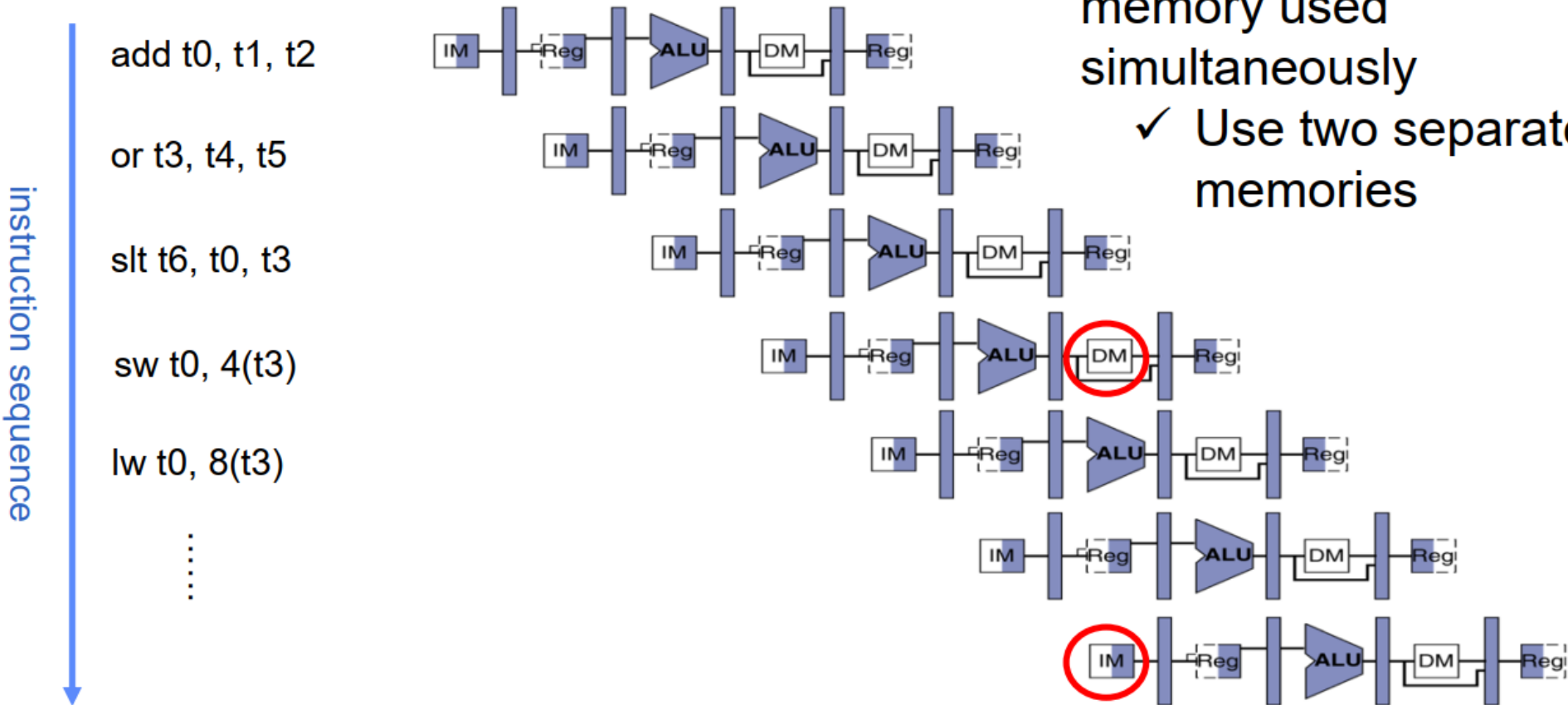
- **Solution 1:** 按序使用资源，其他指令需要暂停执行
- **Solution 2:** 增加更多的硬件支持单元

Can always solve a structural hazard by adding more hardware !

Structural Hazard-- Solution

Memory Access

- Instruction and data memory used simultaneously
✓ Use two separate memories



Structural Hazards--Solution

–Register File conflicts

解决寄存器堆资源冲突的方法

读写端口分离。

设置两个不同的读端口。

设置一个写端口。

- 每条指令：
 - 在解码阶段最多可以读取两个操作数
 - 在回写阶段可以写入一个值
- 每个周期可以同时进行三个访问

Structural Hazards--Solution

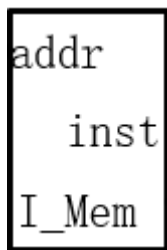
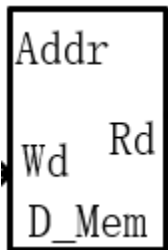
–Register File conflicts

- positive edge for write operation
- negative edge for read operation
- **Double Bump**

Structural Hazard



- 寄存器堆读写端口分开，数据存储器 and 指令存储器也分离，因此，本实验中不会存在结构冲突



Data Hazard--Problem

- ❑ 数据冒险（*Data hazard*）：也称为数据相关
 - ❑ 后一条指令的执行需要前一条指令的执行结果，而此时前一条指令的结果还未产生
 - ❑ 当指令i先于指令j执行时，以下三种情况均会发生数据冒险：
 - *RAW*（*Read After Write*）：当指令i写回结果之前，指令j就已经对此结果发起读操作
 - *WAW*（*Write After Write*）：当指令i写回结果之前，指令j就已经将结果写回
 - *WAR*（*Write After Read*）：当指令i读取数据之前，指令j就已经将结果写回

Data Hazard --Problem

□ 三种数据冒险的举例

(1) 读后写 (WAR) 相关

```
MUL R1, R2 ; (R1) × (R2) → R1,  
ADD R3, R1 ; (R1) + (R3) → R3
```

(2) 写后读 (RAW) 相关

```
MUL R1, R2 ; (R1) × (R2) → R1  
MOV R2, #00H ; 0 → R2
```

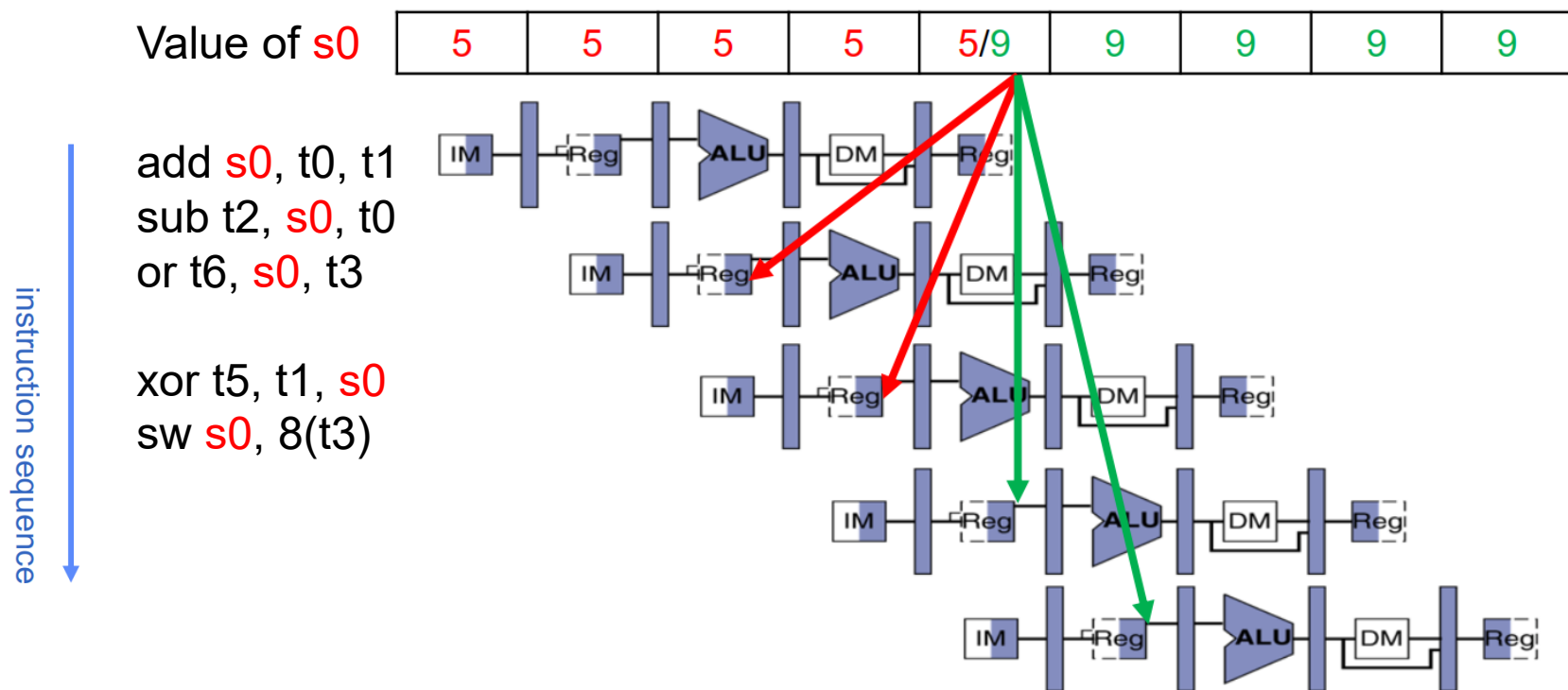
(3) 写后写 (WAW) 相关

```
MUL R1, R2 ; (R1) × (R2) → R1  
MOV R1, #00H ; 0 → R1
```

□ 本实验实现的是基本流水线，所有的数据冒险都属于**RAW数据冒险**，后续只针对此类型做讨论

Data Hazard -- Problem

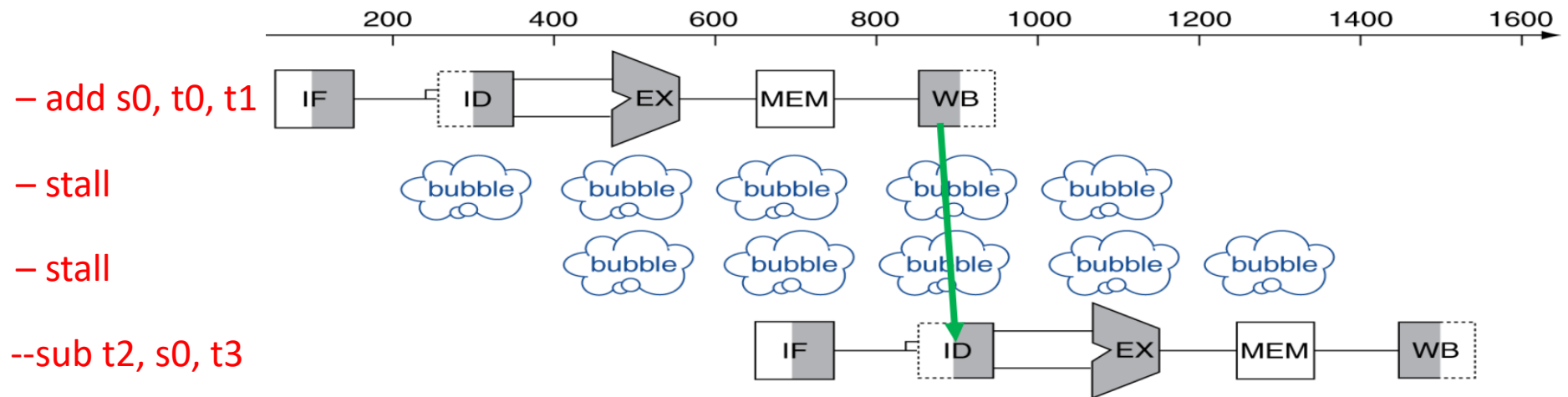
RAW数据冒险



- 指令**SUB**和**OR**的源操作数都依赖于**ADD**的计算结果，若不采取应对措施，则在**s0**结果写回之前，错误的值会被提前读走

Data Hazard----Solution 1: Stalling

- 硬件解决方法：流水线阻塞（**stall**），使数据相关的后续指令延迟执行，也称为插入气泡（**bubble**）

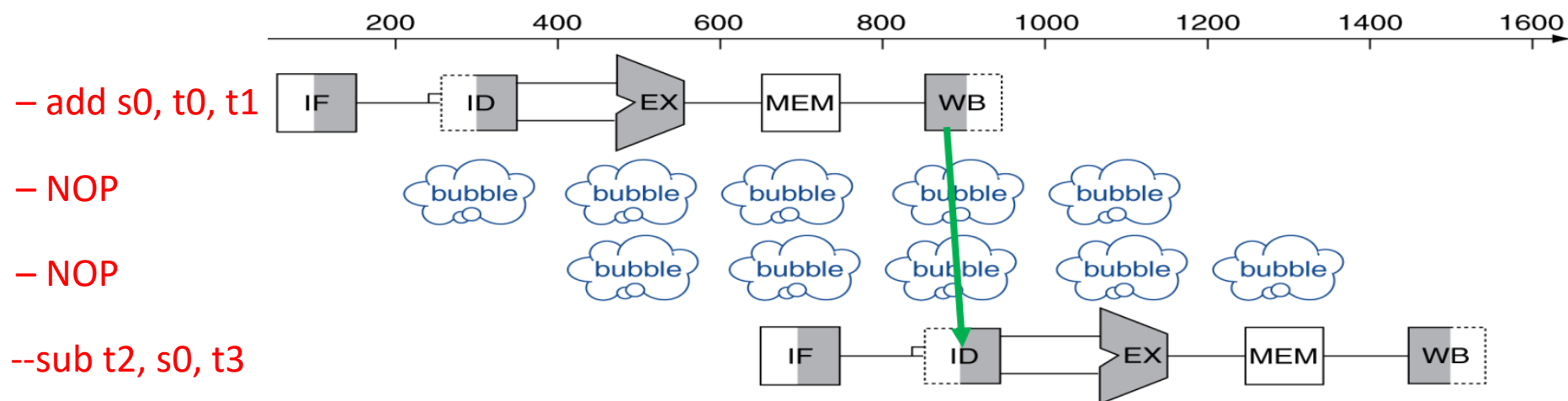


- **Bubble:**

- 其作用类似于空指令（**NOP**），通过硬件控制使流水线不执行有效的操作

Data Hazard----Solution 2: NOP

- 软件解决方法：插入空操作（**NOP**），使数据相关的后续指令延迟执行。



- **NOP:**

- 空指令（NOP），执行不影响运算结果的无关指令（`addi x0,x0,0`）

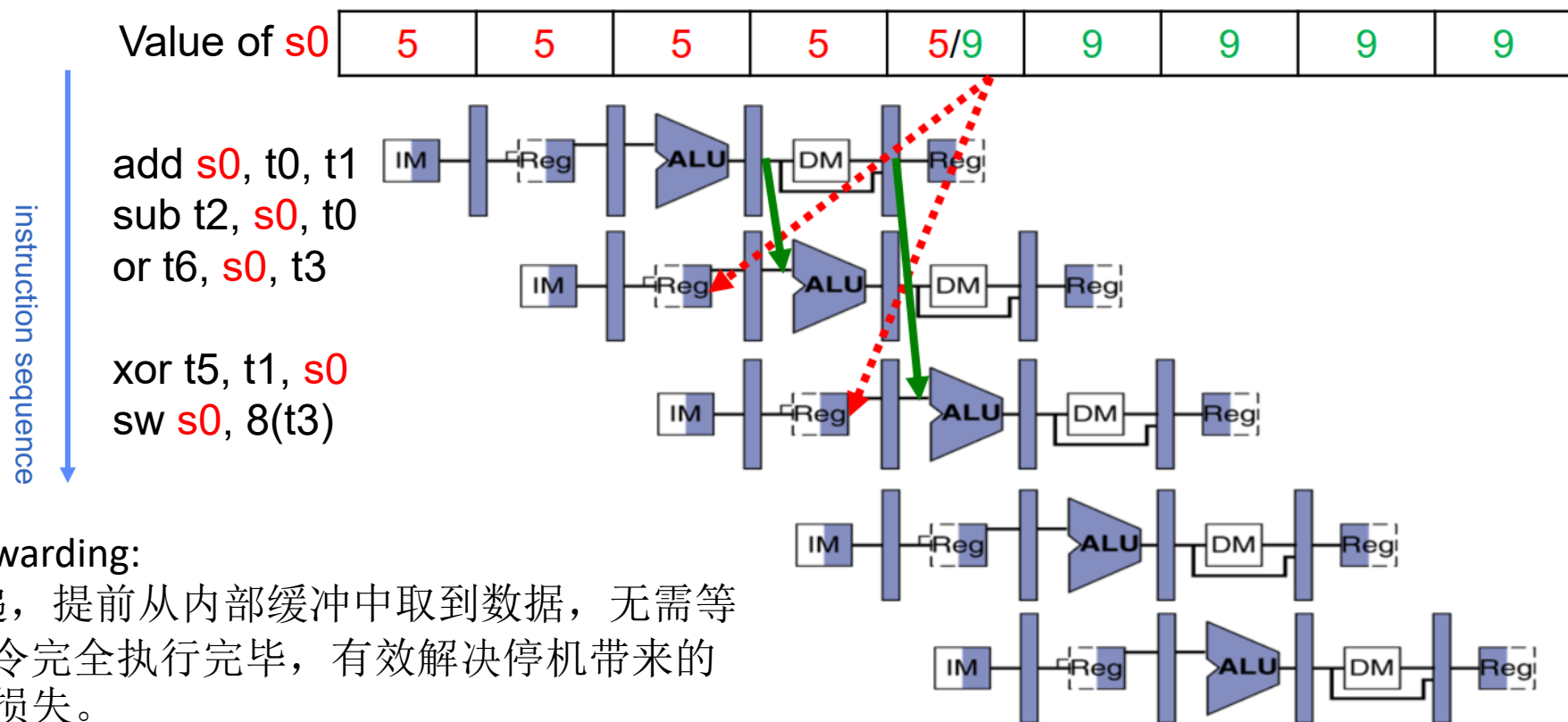
Data Hazard----Stall & nop

- ❑ **stall**: 硬件检测到数据相关问题后，通过硬件阻塞流水线的方式来防止指令的误执行。
- ❑ **nop**: 人为检查到数据相关问题后，通过编译时预先插入空操作指令**nop**的方式来延迟指令的执行。
- ❑ 两种方式虽然都使得流水线能够正常的执行操作，但延迟下一条指令执行的方式使**CPU**性能大打折扣



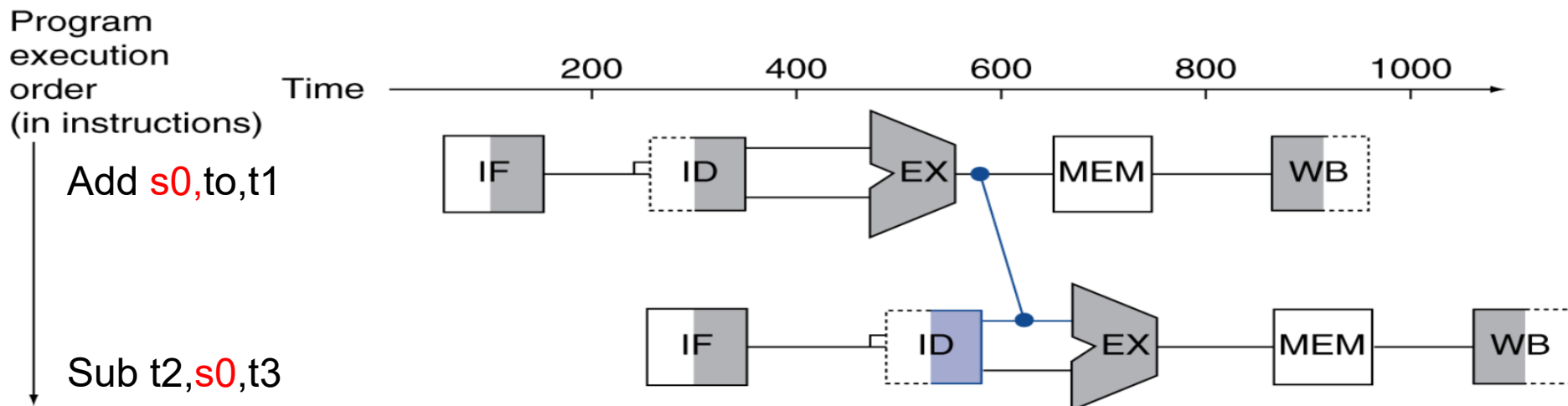
Data Hazard--- Solution 3: Forwarding

- 硬件解决方法：前递（forwarding），又称为旁路（bypassing）
将数据通路生成的中间数据直接往前传递到ALU的输入端，参与下一条指令的运算。



Data Hazard---- Solution 3: Forwarding

- 如图：将ADD指令执行阶段的输出值前递到SUB指令的执行阶段的输入，替换SUB指令在第二阶段读出的寄存器s0的值（无需等待寄存器的值被写回，避免RAW）

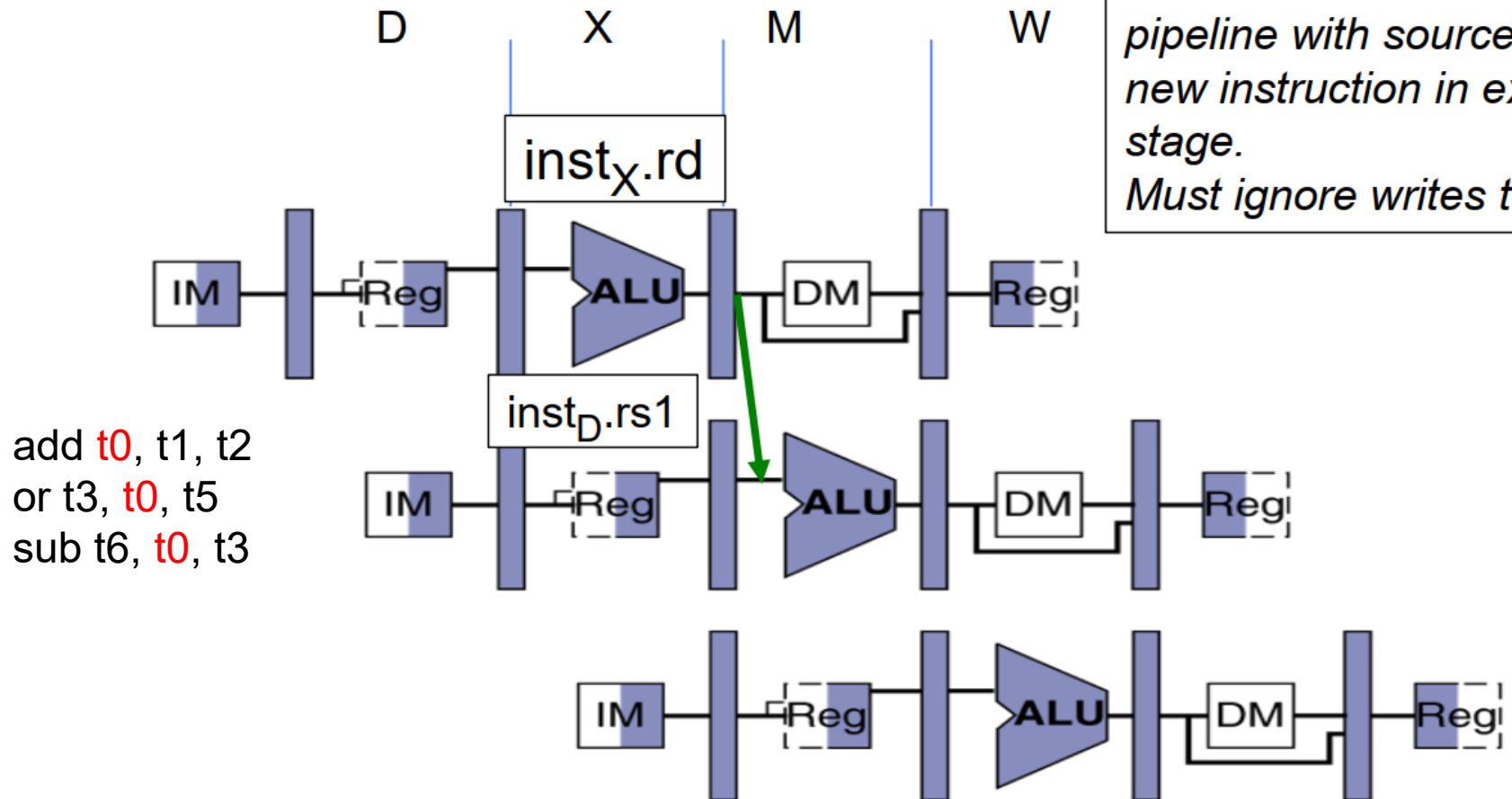


- Forwarding:

—前递，将结果提前传递到下一指令的ALU作为操作对象，所以当且仅当下一条指令的需求时间晚于当前指令结果的产生时间，前递才有效。



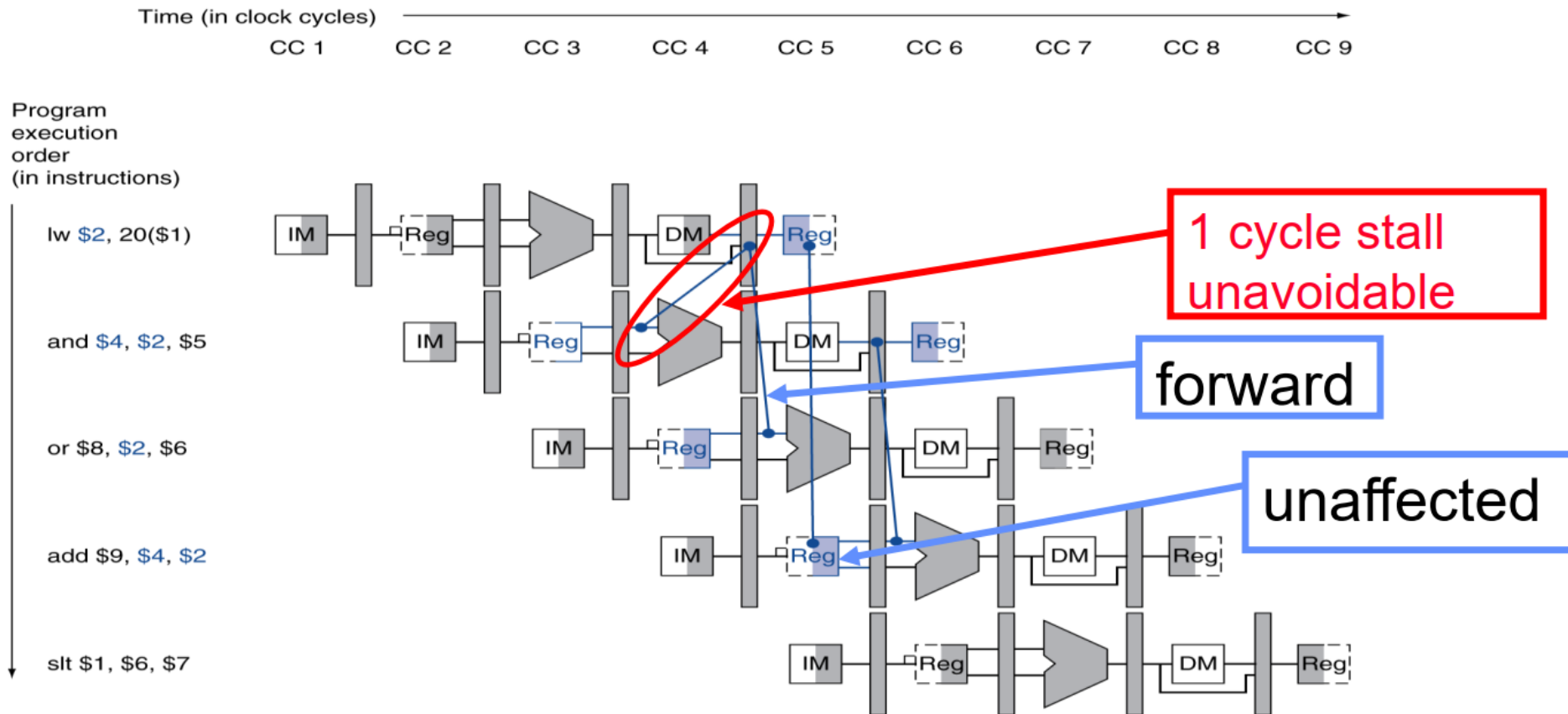
Detect Need for Forwarding



*Compare destination of older two instructions in pipeline with sources of new instruction in execute stage.
Must ignore writes to x0!*

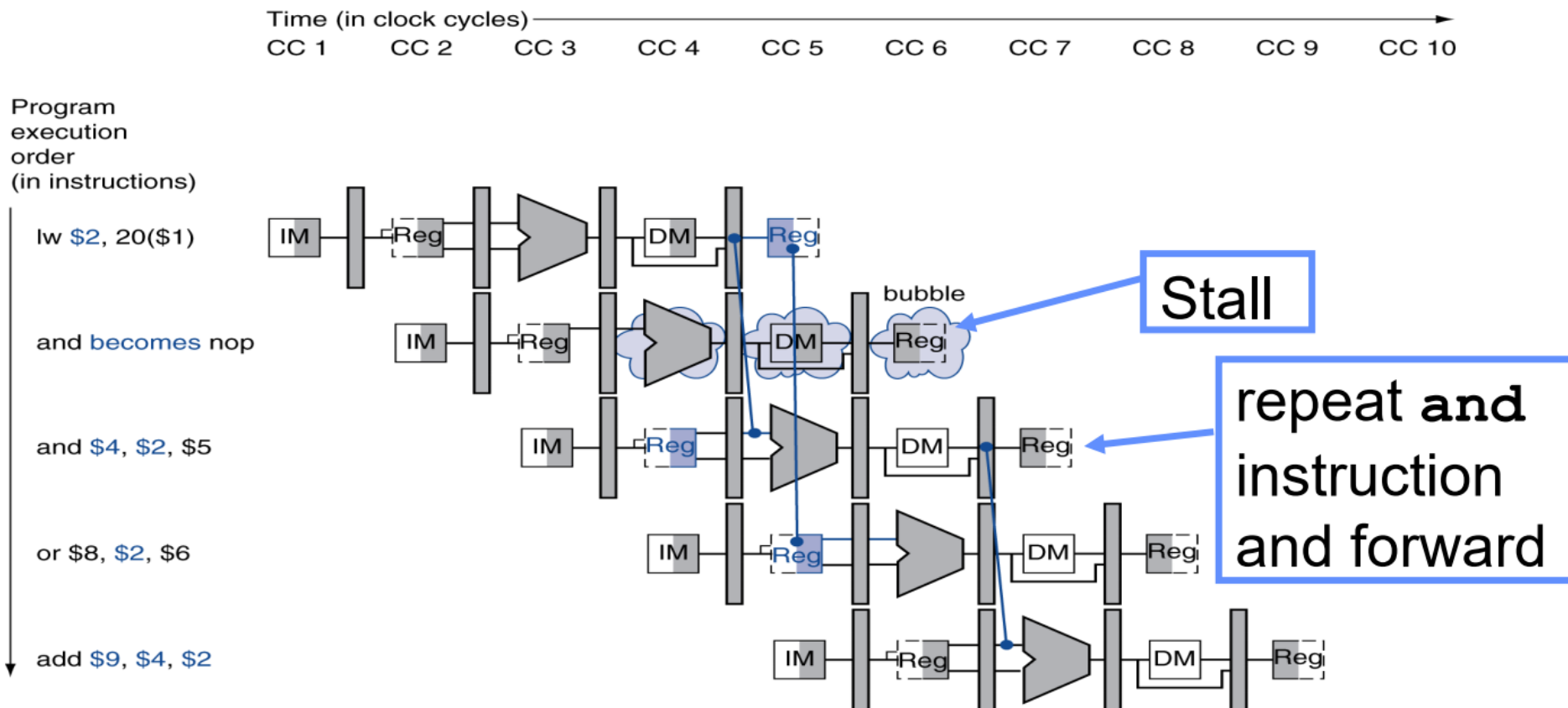
Data Hazard-- Problem: Load Data Hazard

- 载入—使用型数据冒险：如图，当一条load指令后为一条需要使用其结果的R型指令后，直接前递是无效的（此时，阻塞流水线操作是无法避免的）。



Data Hazard-- solution:Load Data Hazard

- 载入—使用型数据冒险解决办法：stall+forwarding；流水线停顿一个时钟周期之后再采用前递的方式。



Data Hazard---- Solution 4: Scheduling

□ 编译器进行指令顺序调整来解决数据冒险。

- eg: RISC-V code for $A[3]=A[0]+A[1]$;
 $A[4]=A[0]+A[2]$

Original Order:

```
lw    t1, 0(t0)
lw    t2, 4(t0)
Stall! → add t3, t1, t2
sw    t3, 12(t0)
lw    t4, 8(t0)
Stall! → add t5, t1, t4
sw    t5, 16(t0)
```

9 cycles

Alternative:

```
lw    t1, 0(t0)
lw    t2, 4(t0)
lw    t4, 8(t0)
add t3, t1, t2
sw    t3, 12(t0)
add t5, t1, t4
sw    t5, 16(t0)
```

7 cycles

Code Scheduling to Avoid Stalls

Data Hazard

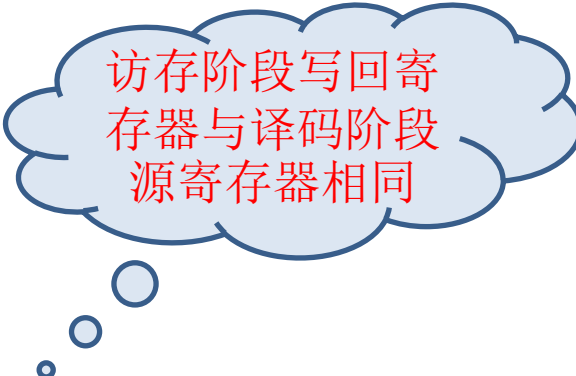
- 针对各种情况导致的数据冒险，本实验统一采用 **pipeline stall** 的方式进行解决

Data Hazard ----- 检测机制

- ❑ 针对数据关联引起的冒险，出现在Mem阶段
- ❑ 检测条件：

- ❑ **MEM hazard**

- if (EX/MEM.RegWrite and Rs1_used)
and (ID.RegisterRs1 \neq 0)
and (EX/MEM.RegisterRd = ID.RegisterRs1)) Data_stall = 1
- if (EX/MEM.RegWrite and Rs2_used)
and (ID.RegisterRs2 \neq 0)
and (EX/MEM.RegisterRd = ID.RegisterRs2)) Data_stall = 1



访存阶段写回寄存器与译码阶段源寄存器相同

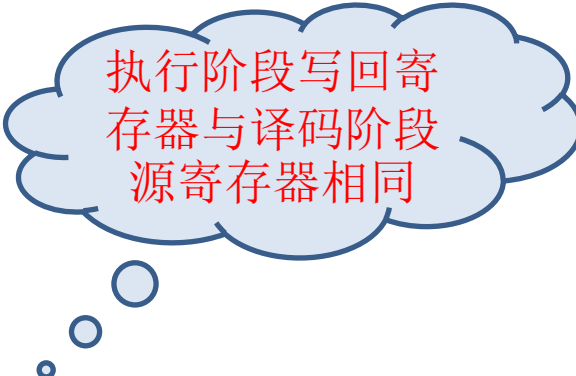
Data Hazard ----- 检测机制

❑ 针对数据关联引起的冒险，出现在EX阶段

❑ 检测条件：

❑ Ex hazard

- if (ID/EX.RegWrite and Rs1_used)
and (ID.RegisterRs1 \neq 0)
and (ID/EX.RegisterRd = ID.RegisterRs1)) Data_stall = 1
- if (ID/EX.RegWrite and Rs2_used)
and (ID.RegisterRs2 \neq 0)
and (ID/EX.RegisterRd = ID.RegisterRs2)) Data_stall = 1



执行阶段写回寄存器与译码阶段源寄存器相同

Data Hazard ----- stall解决

- 检测到Data Hazard，拉高Data_stall 标志信号，此时采取停机操作，在译码和执行之间插入NOP指令

```
if (Data_stall) begin
    en_IF= 0;
    en_IFID= 0;
    NOP_IDEX= 1;
end
else begin
    en_IF= 1;
    en_IFID= 1;

    NOP_IDEX=0;
end
```

Hold pc
and if/id
reg

Insert nop and disable
RegWriet,MemRW

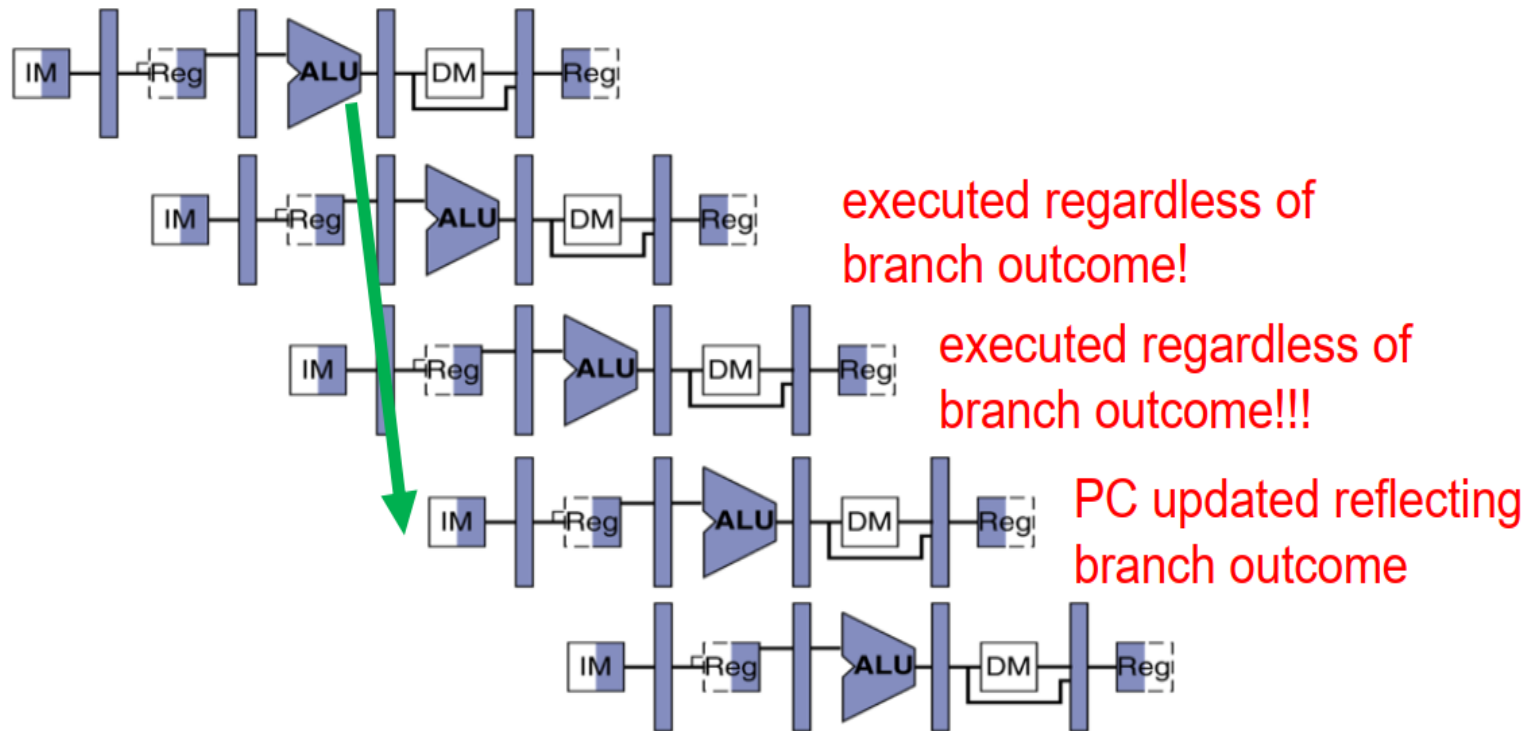
Control Hazards---problem

- ❑ 控制冒险：当PC值不按顺序执行时，流水线中指令的正常执行会被阻塞。
- ❑ 分支、跳转等引起的控制冒险（分支冒险）
- ❑ 异常、中断等引起的控制冒险

Control Hazards---problem: beq

- 如图，**beq**指令只执行之后，下一条指令应该等待其执行完成才知道其目标地址，此时若不采取措施，**pc**会顺序的开始**sub**操作，显然不是期望的结果

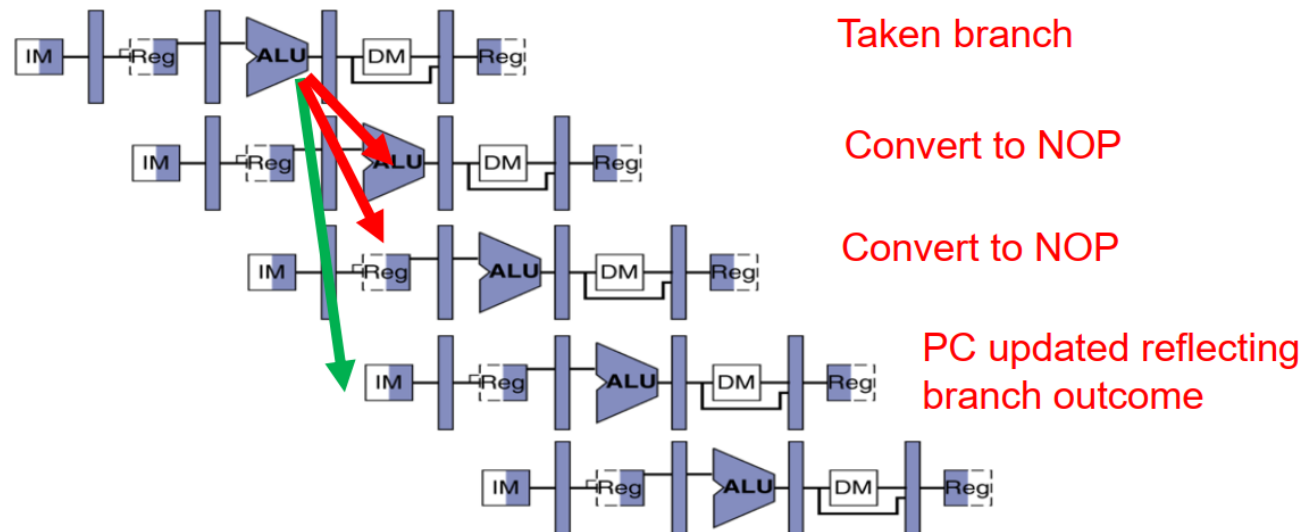
beq t0, t1, label
sub t2, s0, t5
or t6, s0, t3
xor t5, t1, s0
sw s0, 8(t3)



Control Hazards---solution1: stalling/nop

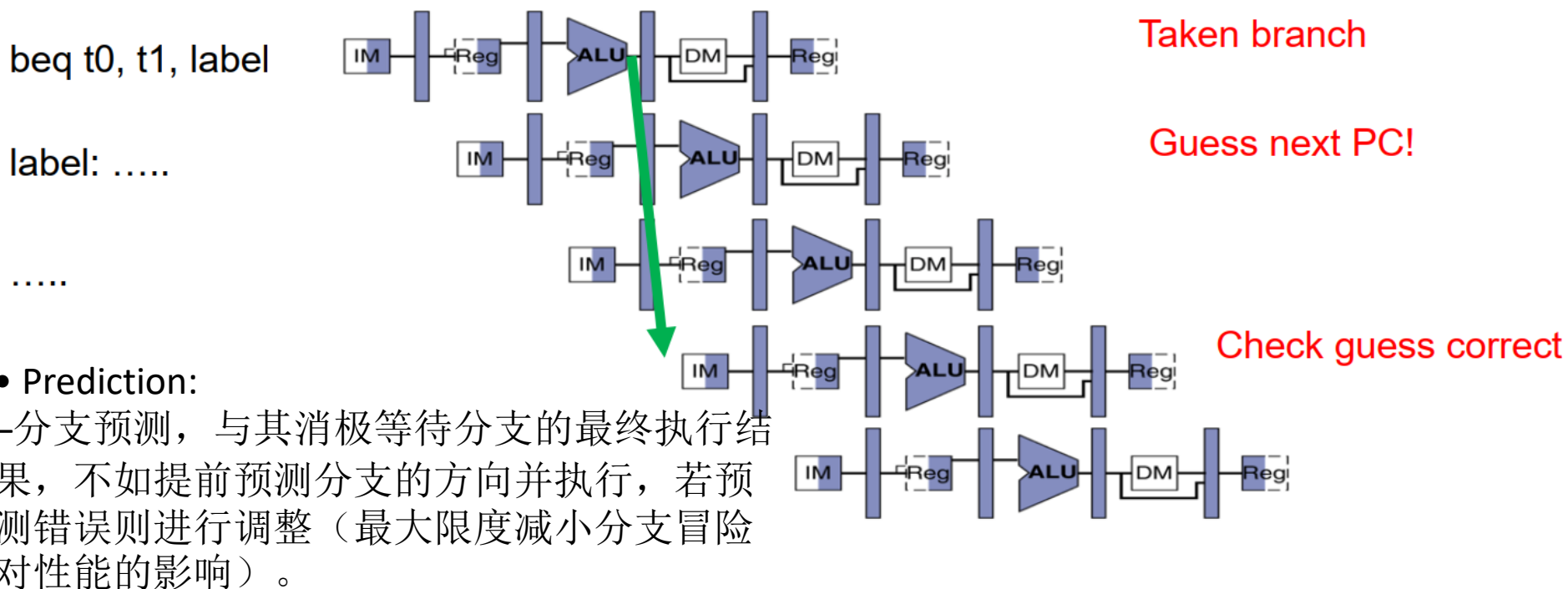
- 类似于数据冒险的解决，针对控制冒险，仍然可以采用硬件阻塞流水线或软件插入nop指令的方式延迟指令的执行。（同样的问题是，流水线的性能会受影响）

```
beq t0, t1, label  
sub t2, s0, t5  
or t6, s0, t3  
label: xxxxxx
```



Control Hazards---solution2:Branch Prediction

- 分支预测（Branch Prediction）：分为静态预测（简单预测分支指令的条件总是满足或不满足）和动态预测（根据实际执行情况动态调整预测位），预测准确则无时间损失，若预测不准确则对分支后不该执行的指令进行冲刷。



Control Hazard

- 针对各种情况导致的控制冒险，本实验统一采用 **pipeline stall** 的方式进行解决

Control Hazard ----- 检测机制

❑ 本实验会改变PC的指令为beq;bne;jal; 因此只需检测这三条指令即可

❑ 检测条件: if(Branch_ID=1 or BranchN_ID=1 or Jump_ID=1)
or (Branch_out_IDEX = 1 or BranchN_out_IDEX=1 or Jump_out_IDEX = 1)
or (Branch_out_EXMem= 1 or BranchN_out_EXMem=1 or Jump_out_EXMem = 1))

译码阶段

执行阶段

Control_stall =1

访存阶段

Control Hazard ----- stall解决

- 检测到Control Hazard，拉高Control_stall 标志信号，此时采取停机操作，在取指和译码之间插入NOP指令

```
if (Control_stall) begin
    NOP_IFID= 1;
end
else begin
    NOP_IFID=0;
end
```



Insert nop

-
- **任务一**：集成设计利用stall解决冒险的流水线CPU，在lab05-3的基础上完成
 - ▣ 设计冒险检测及stall消除冒险的流水线CPU
 - ▣ 替换lab05-3的CPU为本实验集成的带stall处理的流水线CPU

方法一 **RTL代码组织实现（推荐）**
方法二 原理图组织实现

具体方法请参照
lab04，此处以原
理图方式参考供
端口连接使用

冒险检测及stall解决的CPU设计与集成

Stall 冒险检测及处理单元

◎ stall

☞ 根据冒险相关性特点停顿流水线

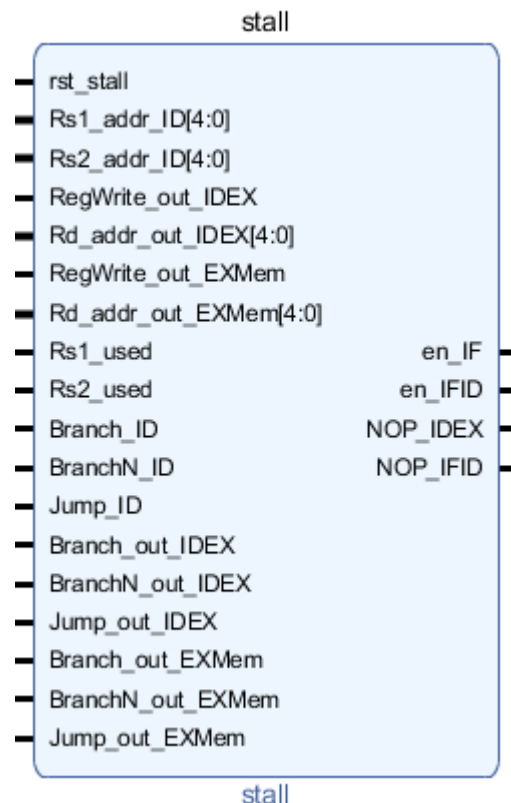
◎ 基本功能

☞ 检测数据冒险和控制冒险

☞ 根据冒险类型，停顿流水线

◎ 接口要求

☞ stall模块接口如图：



stall模块接口信号标准: stall.v

```
module stall(  
    input rst_stall,                //复位  
    input RegWrite_out_IDEX,        //执行阶段寄存器写控制  
    input [4:0]Rd_addr_out_IDEX,    //执行阶段寄存器写地址  
    input RegWrite_out_EXMem,       //访存阶段寄存器写控制  
    input [4:0]Rd_addr_out_EXMem,   //访存阶段寄存器写地址  
    input [4:0]Rs1_addr_ID,         //译码阶段寄存器读地址1  
    input [4:0]Rs2_addr_ID,         //译码阶段寄存器读地址2  
    input Rs1_used,                 //Rs1被使用  
    input Rs2_used,                 //Rs2被使用  
    input Branch_ID,                //译码阶段beq  
    input BranchN_ID,               //译码阶段bne  
    input Jump_ID,                  //译码阶段jal  
    input Branch_out_IDEX,          //执行阶段beq  
    input BranchN_out_IDEX,         //执行阶段bne  
    input Jump_out_IDEX,            //执行阶段jal  
    input Branch_out_EXMem,         //访存阶段beq  
    input BranchN_out_EXMem,        //访存阶段bne  
    input Jump_out_EXMem,           //访存阶段jal
```

stall模块接口信号标准: stall.v

```
output en_IF,           //流水线寄存器的使能及NOP信号
output en_IFID,
output NOP_IFID,
output NOP_IDEx,
);

endmodule
```

-
- ◎ 由于数据冒险的检测涉及到寄存器堆的2个读地址信号，源寄存器是否被使用，以及寄存器堆的Double Bump问题；因此译码阶段需要修改
 - ◎ 检测到冒险后，需要插入NOP指令，建议重新修改流水线寄存器将每一级的PC、指令以及表征是否有效（valid）等信号引出便于更加细致地观察流水线执行状态

译码模块、流水线寄存器 重新设计集成

◎ IF_reg_ID

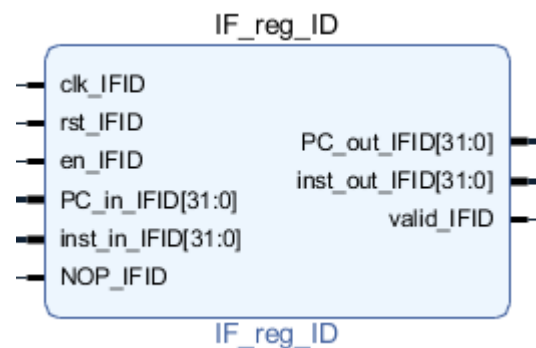
- ☞ 流水线CPU取指和译码之间的寄存器
- ☞ 存储PC值和指令

◎ 基本功能

- ☞ 寄存IF级的输出指令，分割IF级和ID级的指令或控制信号，防止相互干扰，在IF级执行结束时将指令的控制信号传递至下一级。


◎ 接口要求

- ☞ 取指译码寄存器接口如图：



取指-译码寄存器接口：IF_reg_ID.v

```
module IF_reg_ID(  
    input          clk_IFID,          //寄存器时钟  
    input          rst_IFID,          //寄存器复位  
    input          en_IFID,           //寄存器使能  
    input [31:0]   PC_in_IFID,        //PC输入  
    input [31:0]   inst_in_IFID,      //指令输入  
    input          NOP_IFID,          //插入NOP使能  
  
    output reg [31:0] PC_out_IFID,     //PC输出  
    output reg [31:0] inst_out_IFID   //指令输出  
    output reg      valid_IFID        //寄存器有效  
  
    );  
  
    .....  
    else if(NOP_IFID) begin  
        PC_out_IFID  <= 32'h00000000;  
        inst_out_IFID <= 32'h00000013;  
        //nop:addi x0,x0,0x0  
        valid_IFID   <= 1'b0;  
    end  
endmodule
```



一种参考实现

◎ Pipeline_ID

☞ 流水线CPU第二阶段

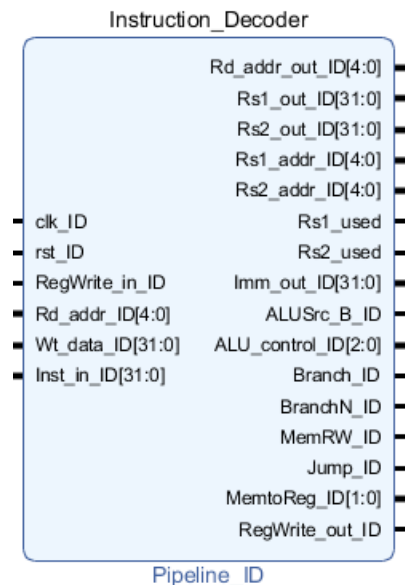
☞ 指令译码

◎ 基本功能

☞ 译码是指将从指令存储器取指的指令进行翻译的过程；译码之后产生各种控制信号，同时寄存器堆根据所需操作数寄存器的索引读出操作数，立即数生成单元输出所需立即数。

◎ 接口要求

☞ 译码模块接口如图：



译码模块接口： Pipeline_ID.v

```
module Pipeline_ID(  
    input          clk_ID,           //时钟  
    input          rst_ID,           //复位  
    input          RegWrite_in_ID,   //寄存器堆使能  
    input [4:0]    Rd_addr_ID,       //写目的地址输入  
    input [31:0]   Wt_data_ID,       //写数据输入  
    input [31:0]   Inst_in_ID,       //指令输入  
    .....  
);
```


译码模块接口： Pipeline_ID.v

```
.....
output reg [31:0] Rd_addr_out_ID, //写目的地址输出
output reg [31:0] Rs1_out_ID,      //操作数1输出
output reg [31:0] Rs2_out_ID,      //操作数2输出
output reg [4:0]  Rs1_addr_ID,      //寄存器地址1
output reg [4:0]  Rs2_addr_ID,      //寄存器地址2
output reg        Rs1_used,         //Rs1被使用
output reg        Rs2_used,         //Rs2被使用
output reg [31:0] Imm_out_ID,       //立即数输出
output reg        ALUSrc_B_ID,      //ALU B端输入选择
output reg [2:0]  ALU_control_ID,   //ALU控制
output reg        Branch_ID,        //Beq控制
output reg        BranchN_ID,       //Bne控制
output reg        MemRW_ID,         //存储器读写
output reg        Jump_ID,          //Jal控制
output reg [1:0]  MemtoReg_ID,      //寄存器写回选择
output reg        RegWrite_out_ID,  //寄存器堆读写
);
```

◎ ID_reg_Ex

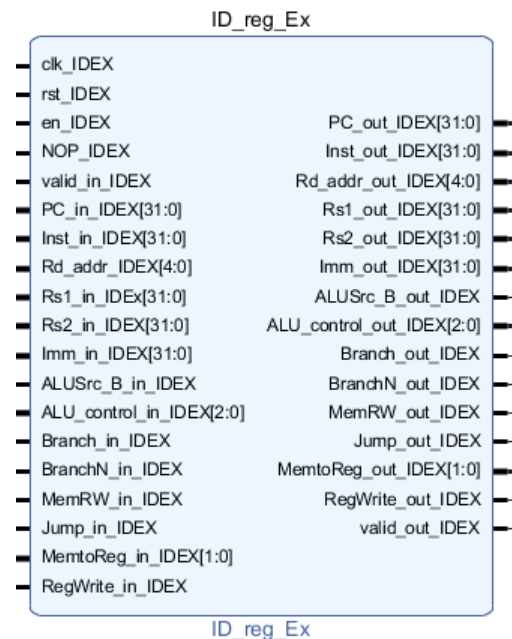
- ☞ 流水线CPU译码和执行之间的寄存器
- ☞ 存储ALU数据和控制信号

◎ 基本功能

- ☞ 寄存ID级的输出指令，分割ID级和EX级的指令或控制信号，防止相互干扰，在ID级执行结束时将指令的控制信号传递至下一级。。

◎ 接口要求

- ☞ 译码执行寄存器接口如图：



译码-执行寄存器接口：ID_reg_Ex.v

```
module ID_reg_Ex(  
    input      clk_IDEX,           //寄存器时钟  
    input      rst_IDEX,           //寄存器复位  
    input      en_IDEX,            //寄存器使能  
    input      NOP_IDEX,           //插入NOP使能  
    input      valid_in_IDEX,      //有效  
    input[31:0] PC_in_IDEX,        //PC输入  
    input[31:0] Inst_in_IDEX,      //指令输入  
    input[4:0]  Rd_addr_IDEX,      //写目的地址输入  
    input[31:0] Rs1_in_IDEX,       //操作数1输入  
    input[31:0] Rs2_in_IDEX,       //操作数2输入  
    input[31:0] Imm_in_IDEX,       //立即数输入  
    input      ALUSrc_B_in_IDEX,   //ALU B输入选择  
    input[2:0]  ALU_control_in_IDEX, //ALU选择控制  
    input      Branch_in_IDEX,     //Beq  
    input      BranchN_in_IDEX,    //Bne  
    input      MemRW_in_IDEX,      //存储器读写  
    input      Jump_in_IDEX,       //Jal
```

译码-执行寄存器接口：ID_reg_Ex.v

```
input[1:0]      MemtoReg_in_IDEX,      //写回选择
input          RegWrite_in_IDEX,      //寄存器堆读写
output reg[31:0] PC_out_IDEX,         //PC输出
output reg[31:0] Inst_out_EXMem,      //inst输出
output reg[4:0] Rd_addr_out_IDEX      //目的地址输出
output reg[31:0] Rs1_out_IDEX,        //操作数1输出
output reg[31:0] Rs2_out_IDEX,        //操作数2输出
output reg[31:0] Imm_out_IDEX,        //立即数输出
output reg     ALUSrc_B_out_IDEX,     //ALU B选择
output reg[2:0] ALU_control_out_IDEX, //ALU控制
output reg     Branch_out_IDEX,       //Beq
output reg     BranchN_out_IDEX,      //Bne
output reg     MemRW_out_IDEX,        //存储器读写
output reg     Jump_out_IDEX,         //Jal
output reg [1:0] MemtoReg_out_IDEX,   //写回
output reg     valid_out_IDEX,        //有效
output reg     RegWrite_out_IDEX      //寄存器堆读写);
```

◎ Ex_reg_Mem

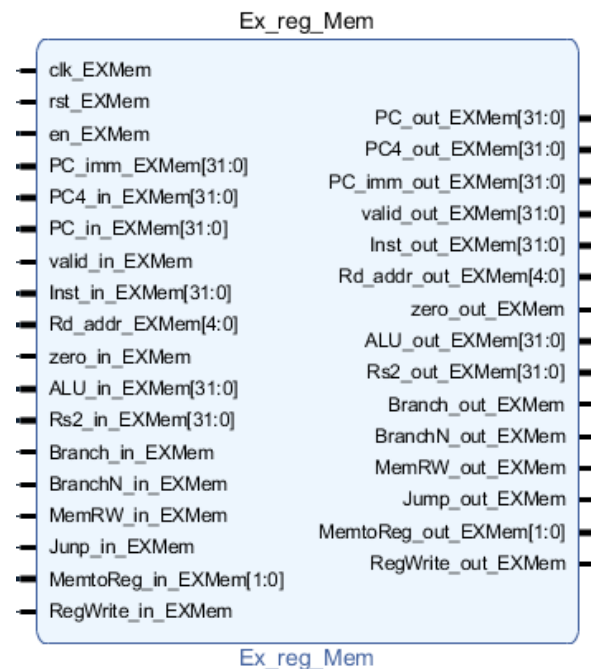
- ☞ 流水线CPU执行和访存之间的寄存器
- ☞ 存储ALU数据和控制信号

◎ 基本功能

- ☞ 寄存EX级的输出指令，分割EX级和MEM级的指令或控制信号，防止相互干扰，在EX级执行结束时将指令的控制信号传递至下一级。

◎ 接口要求

- ☞ 执行访存寄存器接口如图：



执行-访存寄存器接口： Ex_reg_Mem.v

```
module Ex_reg_Mem(  
    input      clk_EXMem,           //寄存器时钟  
    input      rst_EXMem,           //寄存器复位  
    input      en_EXMem,            //寄存器使能  
    input[31:0] PC_imm_EXMem,       //PC输入（imm+pc）  
    input[31:0] PC4_in_EXMem,       //PC+4输入  
    input[31:0] PC_in_EXMem,        //PC输入（from ID/EX）  
    input      valid_in_EXMem,      //有效  
    input[31:0] Inst_in_EXMem,      //指令输入  
    input [4:0] Rd_addr_EXMem,      //写目的寄存器地址输入  
    input      zero_in_EXMem,       //zero  
    input[31:0] ALU_in_EXMem,       //ALU输入  
    input[31:0] Rs2_in_EXMem        //操作数2输入  
    input      Branch_in_EXMem,     //Beq  
    input      BranchN_in_EXMem,    //Bne  
    input      MemRW_in_EXMem,      //存储器读写  
    input      Jump_in_EXMem,       //Jal  
    input [1:0] MemtoReg_in_EXMem,  //写回  
    input      RegWrite_in_EXMem,   //寄存器堆读写
```

执行-访存寄存器接口： Ex_reg_Mem.v

```
.....
    output reg[31:0] PC_imm_out_EXMem, //PC输出(imm+pc)
    output reg[31:0] PC4_out_EXMem, //PC+4输出
    output reg[31:0] PC_out_EXMem, //PC输出
    output reg[31:0] valid_out_EXMem, //valid
    output reg[31:0] Inst_out_EXMem, //PC输出
    output reg[4:0] Rd_addr_out_EXMem, //写目的寄存器输出
    output reg zero_out_EXMem, //zero
    output reg[31:0] ALU_out_EXMem, //ALU输出
    output reg[31:0] Rs2_out_EXMem //操作数2输出
    output reg Branch_out_EXMem, //Beq
    output reg BranchN_out_EXMem, //Bne
    output reg MemRW_out_EXMem, //存储器读写
    output reg Jump_out_EXMem, //Jal
    output reg MemtoReg_out_EXMem, //写回
    output reg RegWrite_out_EXMem, //寄存器堆读写
);
```

◎ Mem_reg_WB

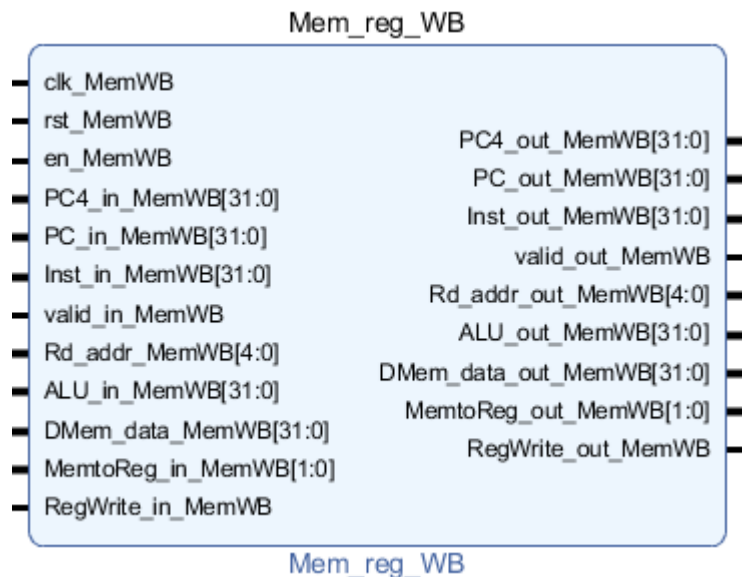
- ☞ 流水线CPU访存和写回之间的寄存器
- ☞ 存储ALU数据和存储器数据

◎ 基本功能

- ☞ 寄存Mem级的输出指令，以及输出数据，传递给写回阶段和寄存器堆。

◎ 接口要求

- ☞ 访存写回寄存器接口如图：



访存-写回寄存器接口:Mem_reg_WB.v

```
module  Mem_reg_WB(  
    input          clk_MemWB,           //寄存器时钟  
    input          rst_MemWB,           //寄存器复位  
    input          en_MemWB,            //寄存器使能  
    input[31:0]    PC4_in_MemWB,        //PC+4输入  
    input[31:0]    PC_in_MemWB,         //PC输入  
    input[31:0]    Inst_in_MemWB,       //inst输入  
    input          valid_in_MemWB,       //有效  
    input[4:0]     Rd_addr_MemWB,        //写目的地址输入  
    input[31:0]    ALU_in_MemWB,        //ALU输入  
    input[31:0]    Dmem_data_MemWB      //存储器数据输入  
    input[1:0]     MemtoReg_in_MemWB,    //写回  
    input          RegWrite_in_MemWB,    //寄存器堆读写  
    .....  
)
```

访存-写回寄存器接口:Mem_reg_WB.v

.....

```
output reg[31:0] PC4_out_MemWB,           //PC+4输出
output reg[31:0] PC_out_MemWB,           //PC输出
output reg[31:0] Inst_out_MemWB,         //指令输出
output reg      valid_out_MemWB,         //有效
output reg[4:0]  Rd_addr_out_MemWB,       //写目的地址输出
output reg[31:0] ALU_out_MemWB,          //ALU输出
output reg[31:0] DMem_data_out_MemWB     //存储器数据输出
output reg[1:0]  MemtoReg_out_MemWB,      //写回
output reg      RegWrite_out_MemWB,      //寄存器堆读写);
```

```
endmodule
```

流水线CPU集成（原理图仅供参考）

The image shows two overlapping dialog boxes from a software development environment. The background dialog is titled 'New Project' and contains the following fields and options:

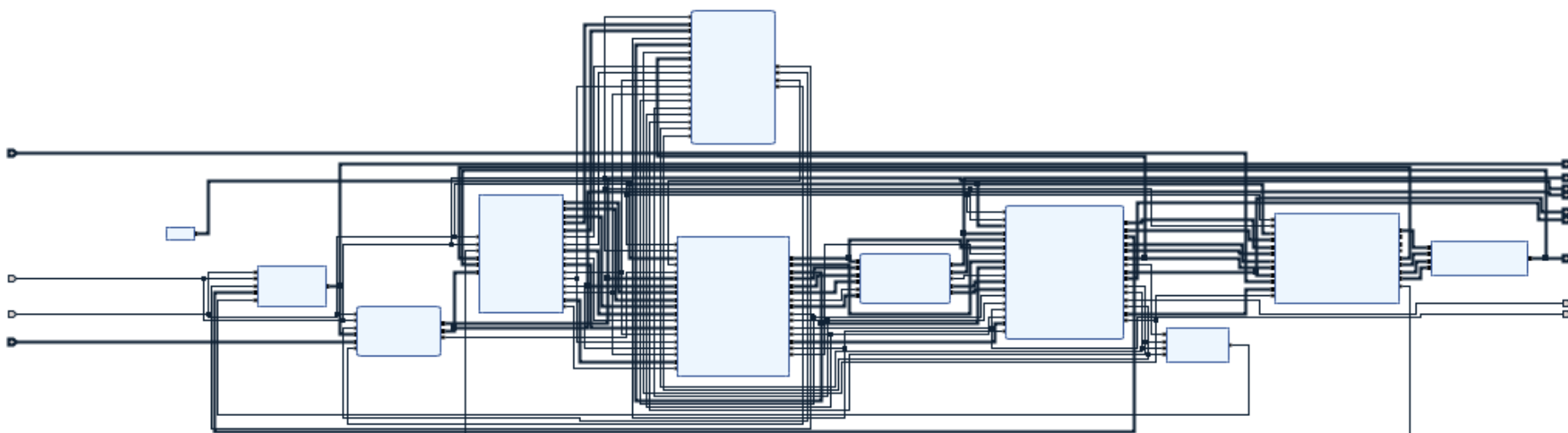
- Project Name:** A text box with the value 'OExp09-Pipeline_CPU'.
- Project location:** A text box with the value 'C:/Users/ASUS/Desktop/OExp09'.
- ☒ **Create project subdirectory**
- Project will be created at:** C:/Users/ASUS/Desktop/OExp09/OExp09-Pipeline_CPU

The foreground dialog is titled 'Create Block Design' and contains the following fields and options:

- Please specify name of block design.**
- Design name:** A text box with the value 'Pipeline_CPU'.
- Directory:** A dropdown menu showing '<Local to Project>'.
- Specify source set:** A dropdown menu showing 'Design Sources'.
- Buttons:** '?', 'OK', and 'Cancel'.

At the bottom of the 'New Project' dialog, there are navigation buttons: '?', '< Back', 'Next >', 'Finish', and 'Cancel'.

流水线CPU集成



可参照详细原理图端口连接，在RTL代码顶层例化各个模块

□ 集成Pipeline CPU 的模块层次结构

Stall（本实验
设计）

```

Pipeline_CPU_i : Pipeline_CPU (Pipeline_CPU.bd) (11)
├── Pipeline_CPU_Ex_reg_Mem_0_0 (Pipeline_CPU_Ex_reg_Mem_0_0.xci)
├── Pipeline_CPU_ID_reg_Ex_0_0 (Pipeline_CPU_ID_reg_Ex_0_0.xci)
├── Pipeline_CPU_IF_reg_ID_0_0 (Pipeline_CPU_IF_reg_ID_0_0.xci)
├── Pipeline_CPU_Mem_reg_WB_0_0 (Pipeline_CPU_Mem_reg_WB_0_0.xci)
├── Pipeline_CPU_Pipeline_Ex_0_0 (Pipeline_CPU_Pipeline_Ex_0_0.xci)
├── Pipeline_CPU_Pipeline_ID_0_0 (Pipeline_CPU_Pipeline_ID_0_0.xci)
├── Pipeline_CPU_Pipeline_IF_0_0 (Pipeline_CPU_Pipeline_IF_0_0.xci)
├── Pipeline_CPU_Pipeline_Mem_0_0 (Pipeline_CPU_Pipeline_Mem_0_0.xci)
├── Pipeline_CPU_Pipeline_WB_0_0 (Pipeline_CPU_Pipeline_WB_0_0.xci)
├── Pipeline_CPU_stall_0_0 (Pipeline_CPU_stall_0_0.xci)
└── Pipeline_CPU_xlconstant_0_0 (Pipeline_CPU_xlconstant_0_0.xci)

```



- 任务二：设计流水线测试方案并完成测试

物理验证

□ 使用**DEMO**程序目测**CPU**运行情况

■ DEMO接口功能

- SW[8]=0, SW[2]=0(全速运行)
- SW[8]=0, SW[2]=1(自动单步)
- SW[8]=1, SW[2]=x(手动单步)

□ 用汇编语言设计测试程序

- 测试ALU指令(R-格式译码\I-立即数格式译码)
- 测试LW指令(I-格式译码)
- 测试SW指令(S-格式译码)
- 测试分支指令(B-格式译码)

物理验证

- ❑ 为更好追踪流水线CPU的特点，VGA显示的接口稍有调整，分别从取指、译码、执行、访存、写回进行显示，请采用更新版本的IP
- ❑ 实验中选取了部分信号进行观测，若想观察其他信号，请参照Lab04将其他待测信号引出即可

```
===== If =====
c: 00000000 inst: 00000000

===== Id =====
c: 00000000 inst: 00000000
0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
5: 00000000 t6: 00000000

===== Ex =====
c: 00000000 inst: 00000000
d: 00 rs1: 00 rs2: 00 rs1_val: 00000000 rs2_val: 00000000 reg_wen: 0
s_inn: 0 inn: 00000000 forward_rs1: 00000000 forward_rs2: 00000000
en_wen: 0 mem_ren: 0 is_branch: 0 is_jal: 0 is_jalr: 0
s_auiipc: 0 is_lui: 0 alu_ctrl: 0 cmp_ctrl: 0

===== Ma =====
c: 00000000 inst: 00000000
d: 00 reg_wen: 0 mem_i_data: 00000000 alu_res: 00000000
en_wen: 0 mem_ren: 0 is_jal: 0 is_jalr: 0

===== Wb =====
c: 00000000 inst: 00000000
d: 00 reg_wen: 0 reg_i_data: 00000000
```

取指阶段信号

译码阶段信号

执行阶段信号

访存阶段信号

写回阶段信号

测试程序参考：

□ 方案一： 无冒险的流水线测试（p.mem）

```
#baseAddr 0000
main:  addi x1,x0,0x1      #x1 = 0x1
      addi x2,x0,0x1      #x2 = 0x1
      addi x3,x0,0x1      #x3 = 0x1
      addi x4,x0,0x1      #x4 = 0x1
      lw  x5,0x8(x0)      #x5 = 0x80000000
      add x6,x1,x1        #x6 = 0x2
      xor x7,x1,x2        #x7 = 0
      sub x8,x2,x1        #x8 = 0
      lw  x9,0x5c(x0)     #x9 = 0xFFFFFFFF
      and x10,x4,x3       #x10= 0x1
      sw  x5,0x4(x0)      #mem(1)=
                          0x80000000

      slt x11,x6,x5       #x11= 0x1
      xori x12,x7,0xAA    #x12= 0xAA
      srl x13,x5,x1       #X13=0x40000000
      andi x14,x8,0x1     #x14= 0x1
      or  x15,x9,x3       #x15=0xFFFFFFFF
      add x16,x10,x10     #x16= 0x2
      xor x17,x11,x8      #x17= 0x1
      lw  x18,0x4(x0)     #x18=0x80000000
```

```
      slt x19,x12,x4      #x19= 0
      srli x20,x13,0x1    #x20= 0x20000000
      and x21,x14,x6      #x21= 0
      sub x22,x5,x1       #x22= 0x7FFFFFFF
      addi x23,x10,0x1    #x23= 0x3
      or  x24,x16,x9      #x24= 0xFFFFFFFF
      xor x25,x19,x11     #x25= 0x1
      andi x26,x20,0xFF   #x26= 0x200000FF
      add x27,x18,x3      #x27= 0x80000001
      srl x28,x20,x2      #x28= 0x10000000
      ori x29,x19,0xAF    #x29= 0xAF
      add x30,x20,x1      #x30= 0x20000001
      lw  x31,0x8(x0)     #x31= 0x80000000
      jal x0,main
      add x0,x0,x0
      add x0,x0,x0
      add x0,x0,x0
```

执行顺序如何？

测试程序参考：

□ 方案二： 有冒险的流水线测试(h.mem)

```
#baseAddr 0000
main:  addi x0,x0,0x0
      addi x1,x0,0x1    #x1 = 0x1
      addi x2,x0,0x1    #x2 = 0x1
      addi x3,x0,0x1    #x3 = 0x1
      addi x4,x0,0x1    #x4 = 0x1
      lw x5,0x8(x0)      #x5 = 0x80000000
      add x6,x5,x1       #x6 = 0x80000001
      xor x7,x1,x2       #x7 = 0
      sub x8,x1,x7       #x8 = 0x1
      lw x9,0x5c(x0)     #x9 = 0xFFFFFFFF
      and x10,x4,x3      #x10 = 0x1
      sw x5,0x4(x0)      #mem(1)=0x80000000
      slt x11,x6,x5      #x11 = 0x0
      xori x12,x7,0xAA   #x12 = 0xAA
      beq x3,x8,loop1
      addi x0,x0,0x0
      add x0,x0,x0
loop1: srl x13,x5,x1      #x13 = 0x40000000
      andi x14,x8,0x1    #x14 = 0x1
      or x15,x9,x3       #x15 = 0xFFFFFFFF
      add x16,x10,x10    #x16 = 0x2
      xor x17,x11,x8     #x17 = 0x1
      lw x18,0x4(x0)     #x18 = 0x80000000
      slt x19,x12,x4     #x19 = 0
      srli x20,x13,0x1   #x20 = 0x20000000
      and x21,x14,x10    #x21 = 0x1
      bne x14,x12,loop2
      addi x0,x0,0x0
loop2: sub x22,x5,x1     #x22 = 0x7FFFFFFF
      addi x23,x10,0x1   #x23 = 0x2
      or x24,x16,x9      #x24 = 0xFFFFFFFF
      xor x25,x19,x11    #x25 = 0x0
      andi x26,x20,0xFF  #x26 = 0x200000FF
      add x27,x18,x3     #x27 = 0x80000001
      srl x28,x20,x2     #x28 = 0x10000000
      ori x29,x19,0xAF   #x29 = 0xAF
      add x30,x20,x1     #x30 = 0x20000001
      lw x31,0x8(x0)     #x31 = 0x80000000
      jal x0,main
      addi x21,x21,0x1
      addi x23,x23,0x1
      addi x25,x25,0x1
```

与之前相比
执行结果如
何？

设计测试记录表格

- **ALU指令测试结果记录**
 - 自行设计记录表格

 **END**