第十一章

指令流水线

合肥工业大学
系统结构研究所
陈 田

一、如何提高机器速度

1. 提高访存速度

   高速芯片　　Cache　　多体并行

2. 提高 I/O 和主机之间的传送速度

   中断　　DMA　　通道　　I/O 处理机　　多总线

3. 提高运算器速度

   高速芯片　　改进算法　　快速进位链

- 提高整机处理能力

  高速器件　　改进系统结构，开发系统的并行性

二、系统的并行性

1. 并行的概念

并行 { 并发：两个或两个以上事件在 同一时间段 发生

同时：两个或两个以上事件在 同一时刻 发生

时间上互相重叠

2. 并行性的等级

过程级（程序、进程）	粗粒度	软件实现
指令级（指令之间） （指令内部）	细粒度	硬件实现

三、指令流水原理

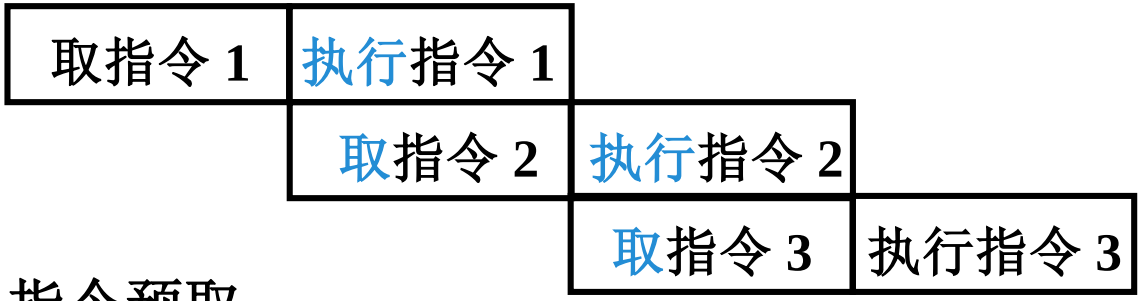
1. 指令的串行执行

取指令 1	执行指令 1	取指令 2	执行指令 2	取指令 3	执行指令 3	…

取指令　　取指令部件　　完成　　总有一个部件 空闲

执行指令　　执行指令部件　　完成

2. 指令的二级流水



指令预取

若 取指 和 执行 阶段时间上 完全重叠

指令周期 减半　速度提高 1 倍

3. 影响指令流水效率加倍的因素

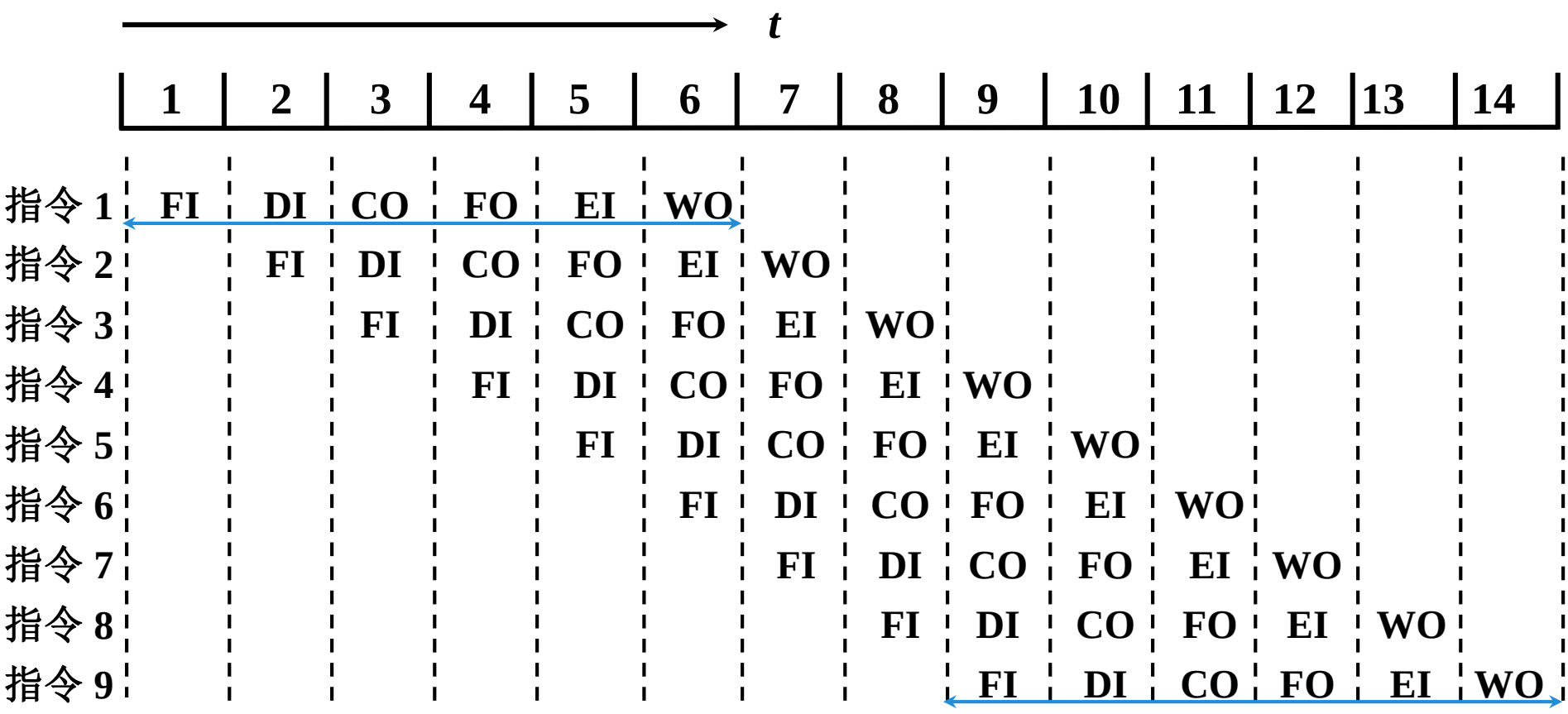
(1) 执行时间 > 取指时间

(2) 条件转移指令 对指令流水的影响

必须等 上条 指令执行结束，才能确定 下条 指令的地址，

造成时间损失　　猜测法

4. 指令的六级流水



完成一条指令　　6 个时间单位

串行执行　　6 × 9 = 54 个时间单位

六级流水　　14 个时间单位

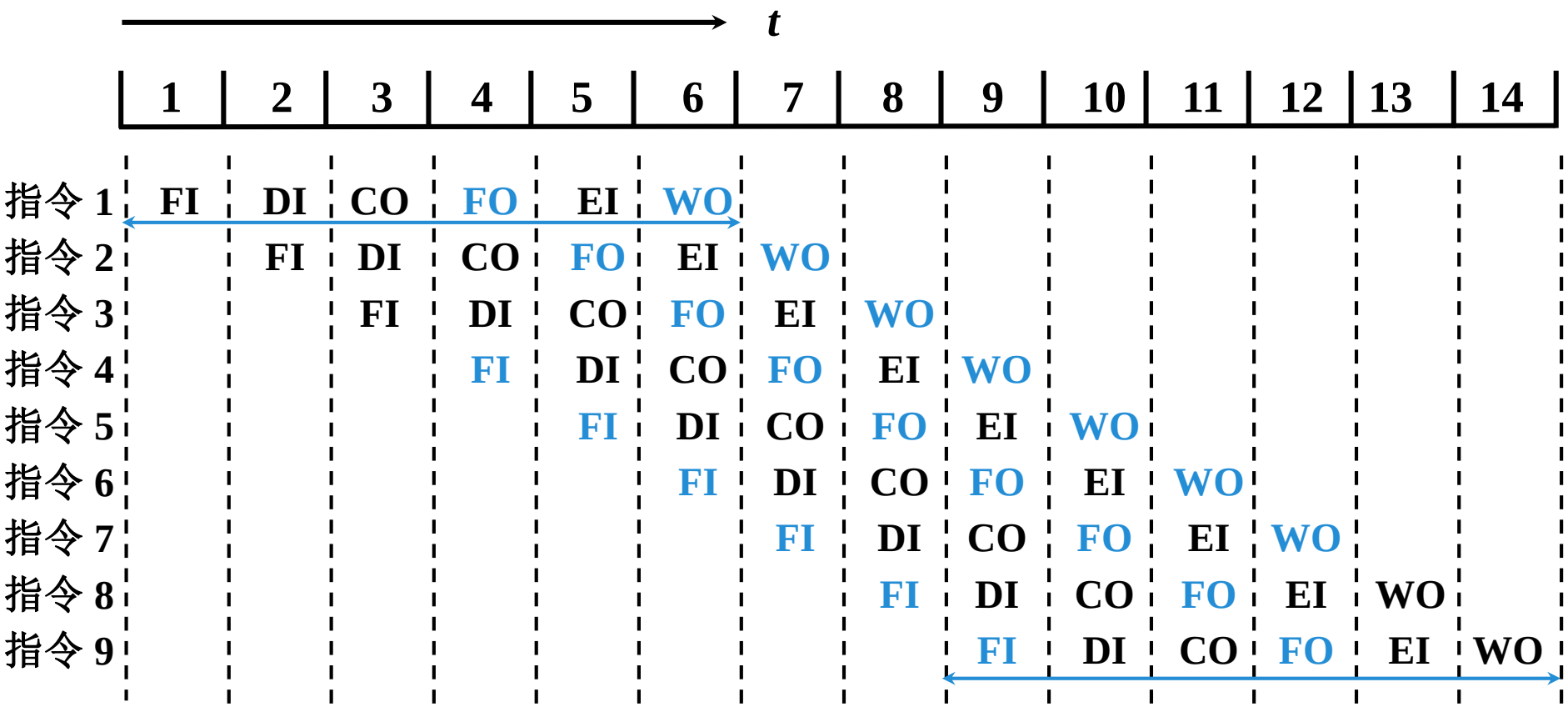
三、影响指令流水线性能的因素

程序的相近指令之间出现某种关联

使指令流水出现停顿，影响流水线效率

三、影响指令流水线性能的因素

1. 结构相关（冒险）不同指令争用同一功能部件产生资源冲突



解决办法

指令 1 与指令 4 冲突

指令 2 与指令 5 冲突

指令1、指令3、指令 6 冲突

- 停顿
- 指令存储器和数据存储器分开
- …
- 指令预取技术（适用于访存周期短的情况）

2. 数据相关

不同指令因重叠操作，可能改变操作数的 读/写 访问顺序

- 写后读相关（RAW）：指令j在指令i写入寄存器前就读此寄存器

  SUB R_1，R_2，R_3 ；（R_2）－（R_3）⟶ R_1

  ADD R_4，R_5，R_1 ；（R_5）+（R_1）⟶ R_4

- 读后写相关（WAR）：指令j在指令i读寄存器前就写入此寄存器

  MUL R_1，R_2 ；(R_1)×（R_2）⟶(R1)

  MOV R_2，0 ；0⟶R_2

- 写后写相关（WAW）：指令j在指令i写入寄存器前就写入此寄存器

  MUL R_3，R_2，R_1 ；（R_2）×（R_1）⟶ R_3

  SUB R_3，R_4，R_5 ；（R_4）－（R_5）⟶ R_3

解决办法 • 后推法 • 采用 旁路技术

3. 控制相关

由转移指令引起

```
      LDA    # 0
      LDX    # 0
M     ADD    X, D
      INX
      CPX    # N
      BNE    M
      DIV    # N
      STA    ANS
```

BNE 指令必须等

CPX 指令的结果

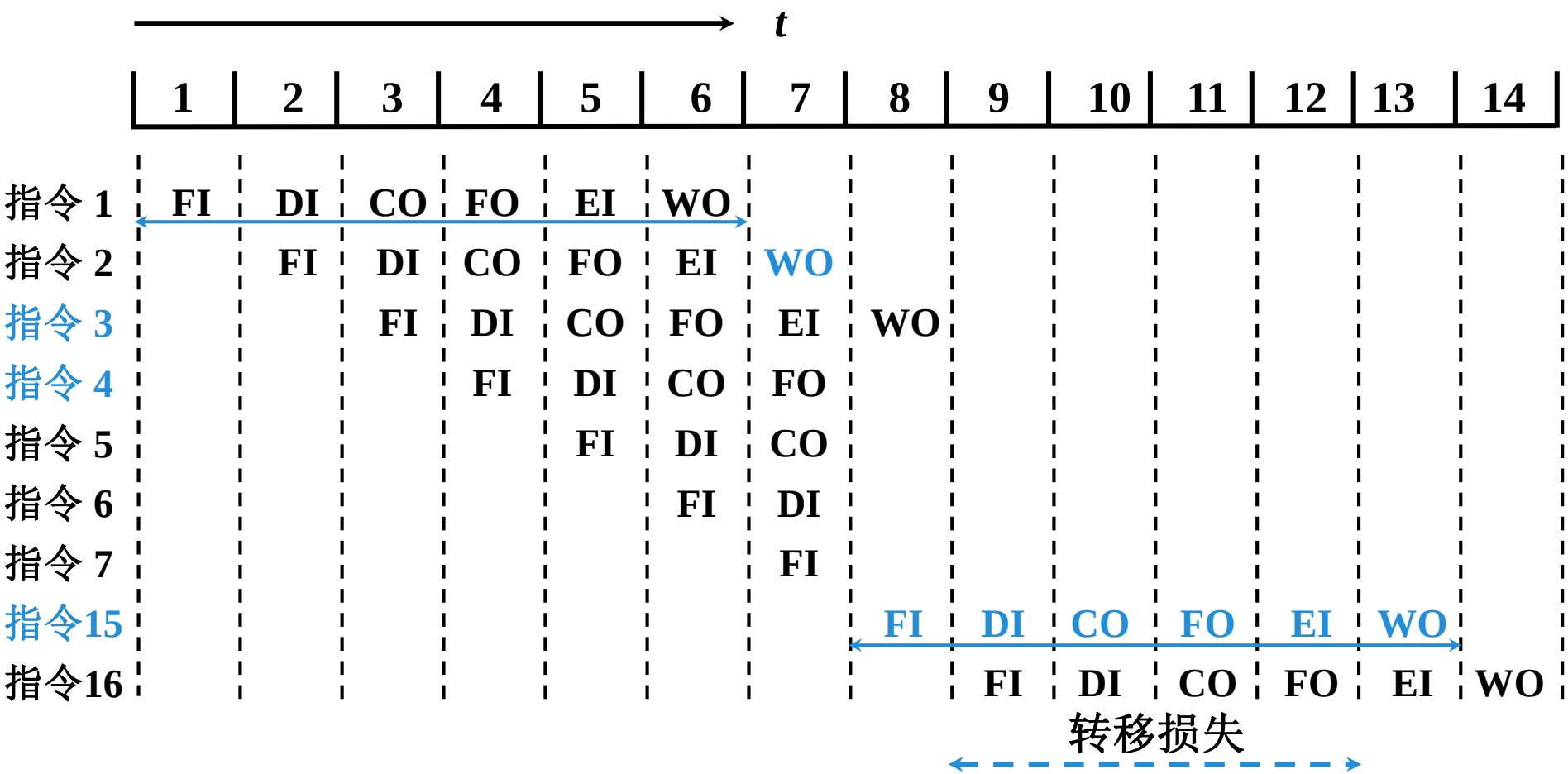
才能判断出

是转移

还是顺序执行

3. 控制相关

设 指令3 是转移指令



四、流水线性能

1. 吞吐率

单位时间内 流水线所完成指令 或 输出结果 的 数量

设 m 段的流水线各段时间为Δt

- 最大吞吐率:流水线在连续流动达到稳定状态后得到的吞吐率

$$T_{pmax} = \frac{1}{\Delta t}$$

- 实际吞吐率：流水线在运行有限个任务时实际能够达到的吞吐率

连续处理 n 条指令的吞吐率为

$$T_p = \frac{n}{m \cdot \Delta t + (n-1) \cdot \Delta t}$$

2. 加速比 S_p

m 段的 流水线的速度 与等功能的 非流水线的速度 之比

设流水线各段时间为 Δt

完成 n 条指令在 m 段流水线上共需

$$T = m \cdot \Delta t + (n-1) \cdot \Delta t$$

完成 n 条指令在等效的非流水线上共需

$$T' = nm \cdot \Delta t$$

则

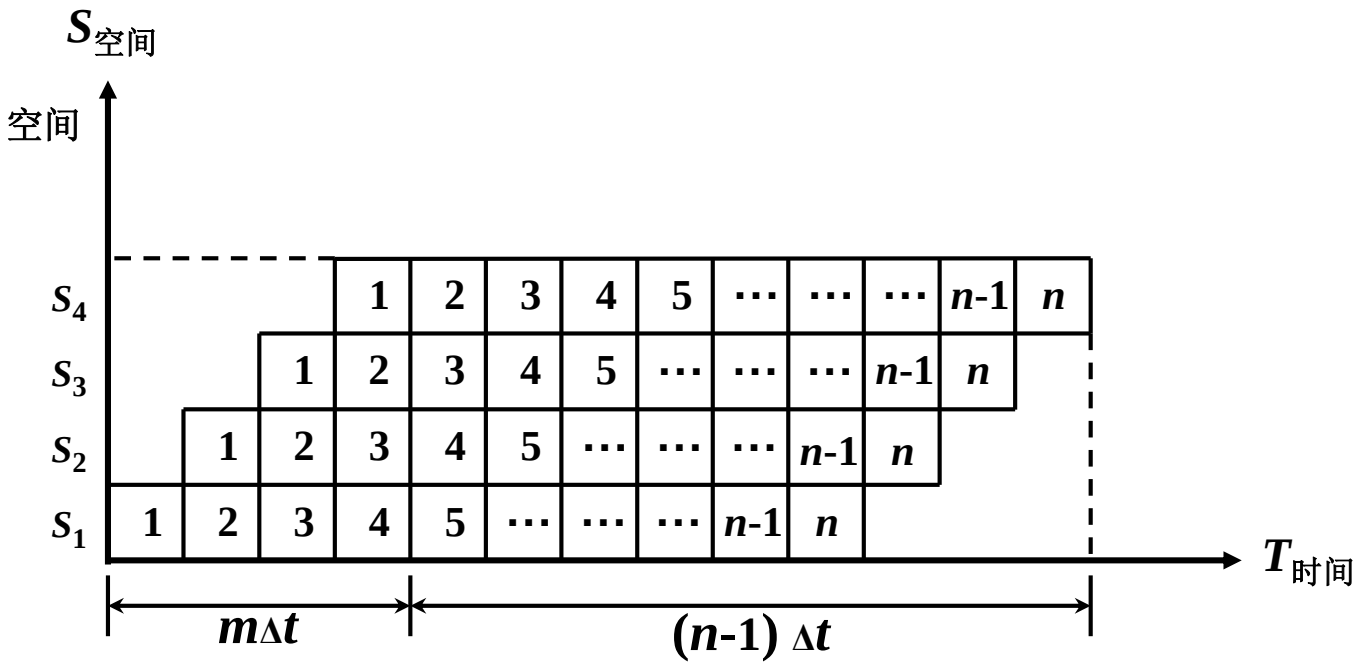
$$S_p = \frac{nm \cdot \Delta t}{m \cdot \Delta t + (n-1) \cdot \Delta t} = \frac{nm}{m + n - 1}$$

3. 效率

流水线中各功能段的 利用率

由于流水线有 建立时间 和 排空时间

因此各功能段的 设备不可能 一直 处于 工作 状态

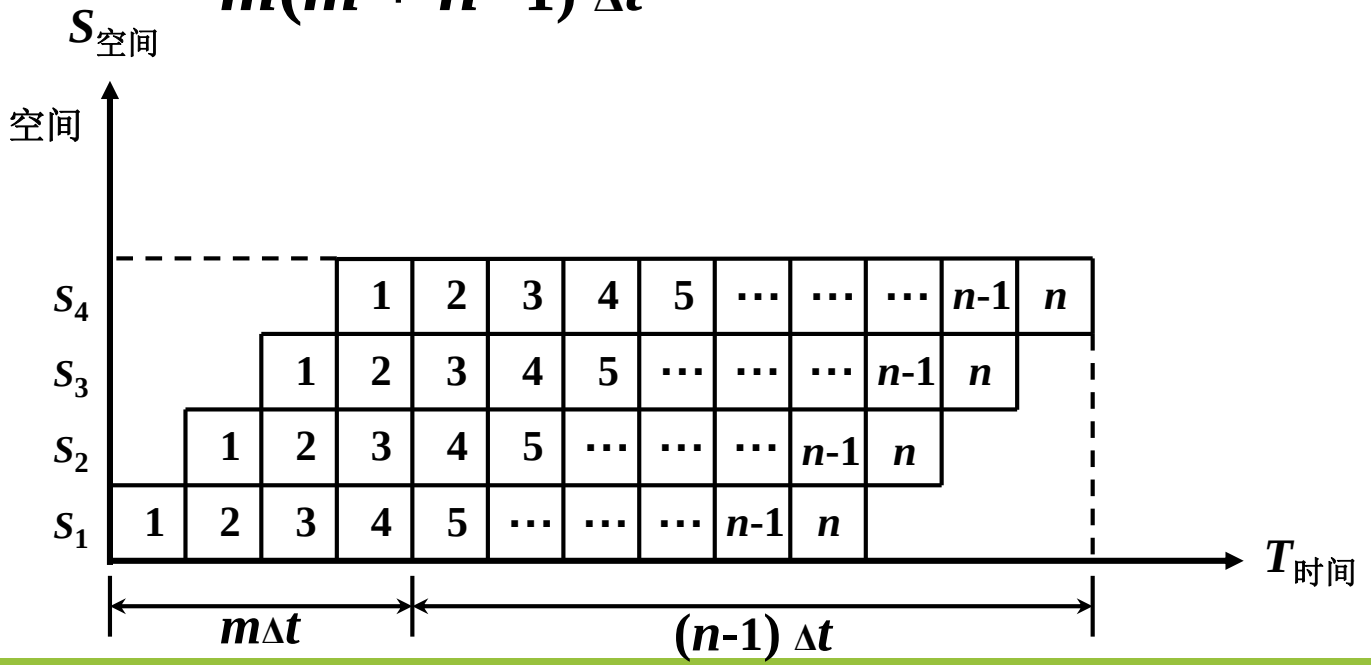


3. 效率

流水线中各功能段的 利用率

$$\text{效率} = \frac{\text{流水线各段处于工作时间的时空区}}{\text{流水线中各段总的时空区}}$$

$$= \frac{mn\Delta t}{m(m+n-1)\,\Delta t}$$



时空图

- 流水线中各功能部件的重叠工作情况可以用时空图表示。
- 时空图的两种形式：
  - 一种以流水段（流水部件）为纵坐标，以时间段为横坐标，用来表示每个部件在每个时间段的工作情况。
  - 另一种以指令序列为纵坐标，以时间段为横坐标，纵坐标的方向是从上到下的（指令序列通常从上到下表示）

流水线指令集的设计

- **具有什么特征的指令集有利于流水线执行呢？**
  - **长度尽量一致，有利于简化取指令和指令译码操作**
    - MIPS指令32位，下址计算方便: PC+4
    - X86指令从1字节到17字节不等，使取指部件及其复杂
  - **格式少，且源寄存器位置相同，有利于在指令未知时就可取操作数**
    - MIPS指令的Rs和Rt位置一定，在指令译码时就可读Rs和Rt的值

      （若位置随指令不同而不同，则需先确定指令后才能取寄存器编号）
  - **load / Store指令才能访问存储器，有利于减少操作步骤，规整流水线**
    - lw/sw指令的地址计算和运算指令的执行步骤规整在同一个周期
    - X86运算类指令操作数可为内存数据，需计算地址、访存、执行
  - **内存中”对齐”存放，有利于减少访存次数和流水线的规整**

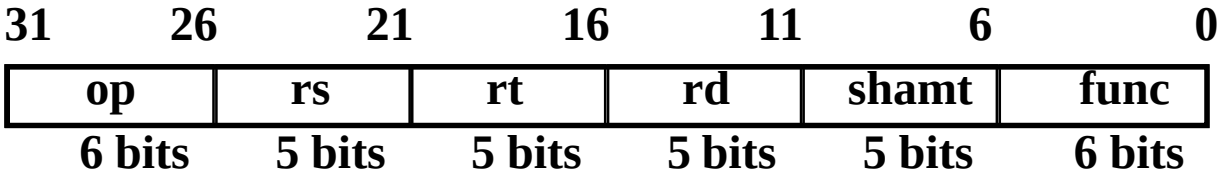
总之，规整、简单和一致等特性有利于指令的流水线执行

流水线执行方式能大大提高指令吞吐率，现代计算机都采用流水线方式！

MIPS指令格式

- 所有指令都是32位宽，须按字地址对齐

R-Type指令



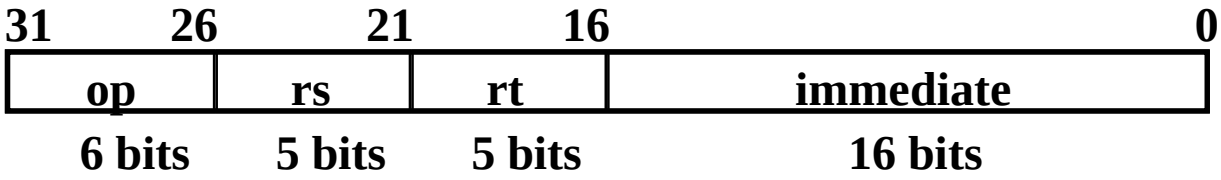
❖ 有三种指令格式

- R-Type

  两个操作数和结果都在**寄存器**的**运算**指令。如：sub rd, rs, rt

- I-Type
  - **运算指令**：一个寄存器、一个立即数。如：ori rt, rs, imm16
  - LOAD和STORE指令。如：lw rt, rs, imm16
  - 条件分支指令。如：beq rs, rt, imm16

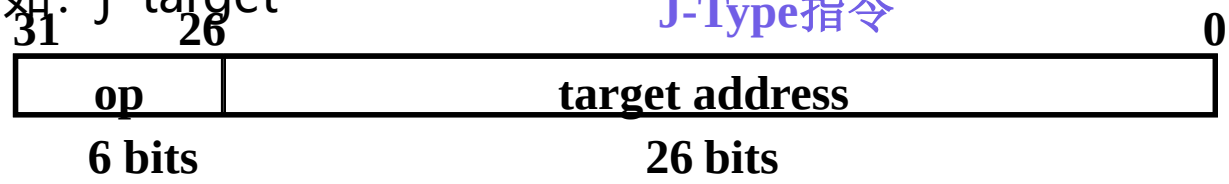
I-Type指令



- J-Type

  无条件跳转指令。如：j target

J-Type指令

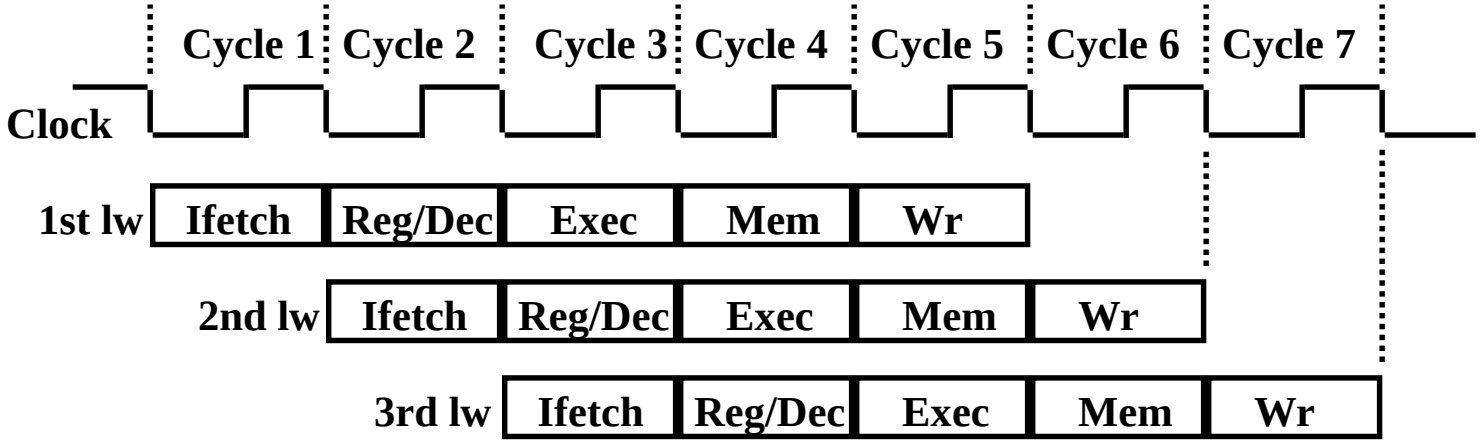


MIPS arithmetic and logic instructions

Instruction	汇编举例	含义	备注
add	add $1,$2,$3	$1 = $2 + $3	3个寄存器操作数
subtract	sub $1,$2,$3	$1 = $2 – $3	3个寄存器操作数
load word 寄存器	lw $ s1,100 ($ S2)	$ s1 =Memory[$ s2+100]	从内存取一个字到
store word 寄存器	sw $ s1,100 ($ S2)	Memory[$ s2+100] = $ s1	从内存取一个字到
multiply	mult $2,$3	Hi, Lo = $2×$3	64-bit signed product
divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
branch on equal	beq $ s1,$ s2, L	if ($ s1==$ s2) go to L	相等则转移
branch on not equal	bne $ s1,$ s2, L	if ($ s1！=$ s2) go to L	不相等则转移

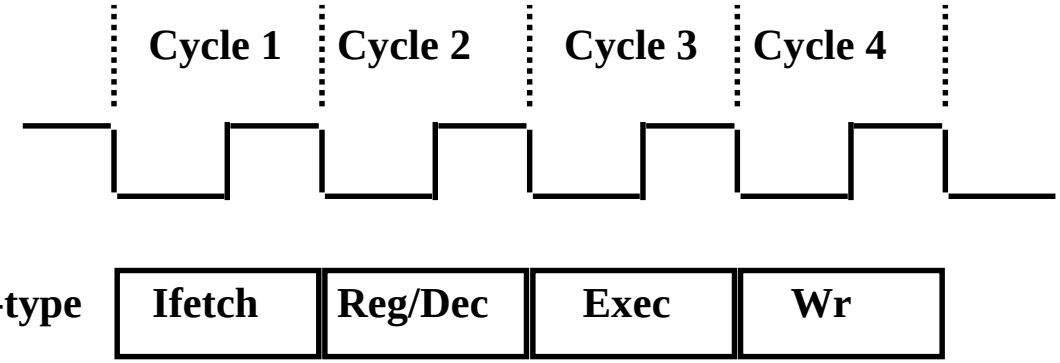
Instruction	*Example*	*Meaning*	*Comment*
and	and $1,$2,$3	$1 = $2 & $3	Logical AND
or	or $1,$2,$3	$1 = $2 | $3	Logical OR
xor	xor $1,$2,$3	$1 = $2 ⊕ $3	Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	Logical NOR

Load指令的流水线



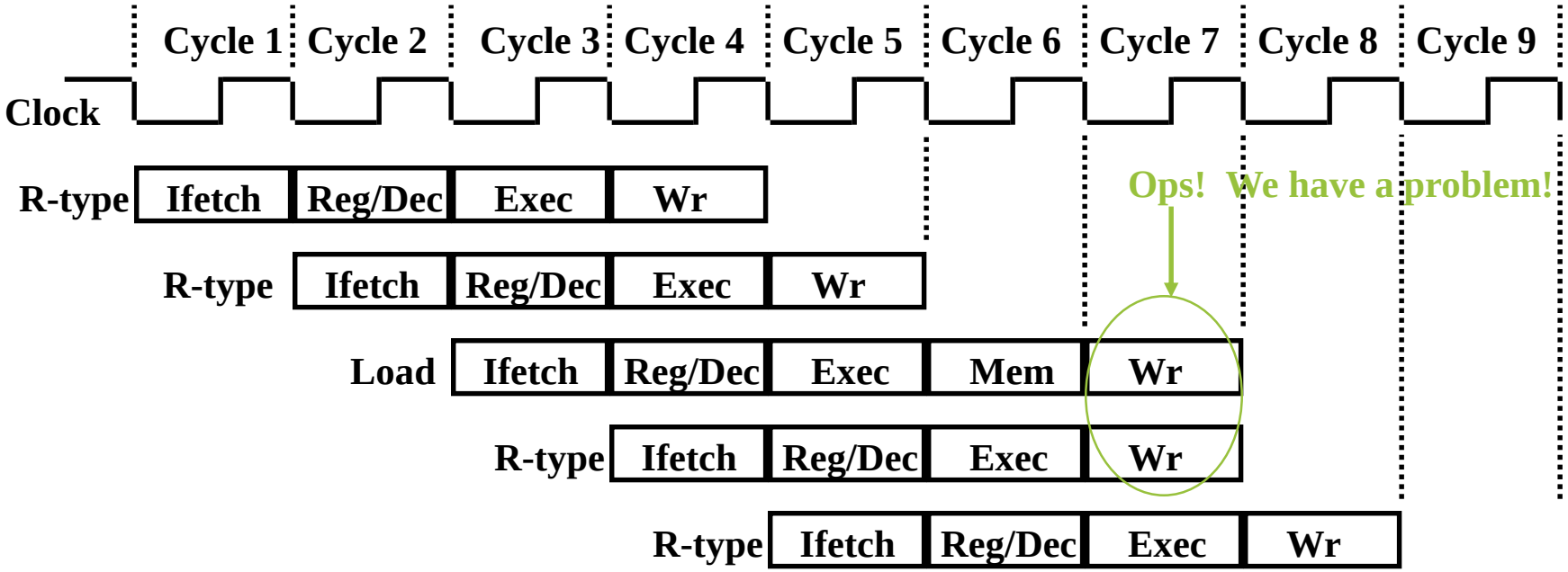
- 每个周期有五个功能部件同时在工作
- 后面指令在前面完成取指后马上开始
- 每个**load**指令仍然需要五个周期完成
- 但是吞吐率**(throughput)**提高许多，理想情况下，有：
  - 每个周期有一条指令进入流水线
  - 每个周期都有一条指令完成
  - 每条指令的有效周期(CPI)为1

R-type指令的4个阶段



- **Ifetch:** 取指令并计算PC+4
- **Reg/Dec:** 从寄存器取数，同时指令在译码器进行译码
- **Exec:** 在ALU中对操作数进行计算
- **Wr:** ALU计算的结果写到寄存器

含R-type和 Load 指令的流水线



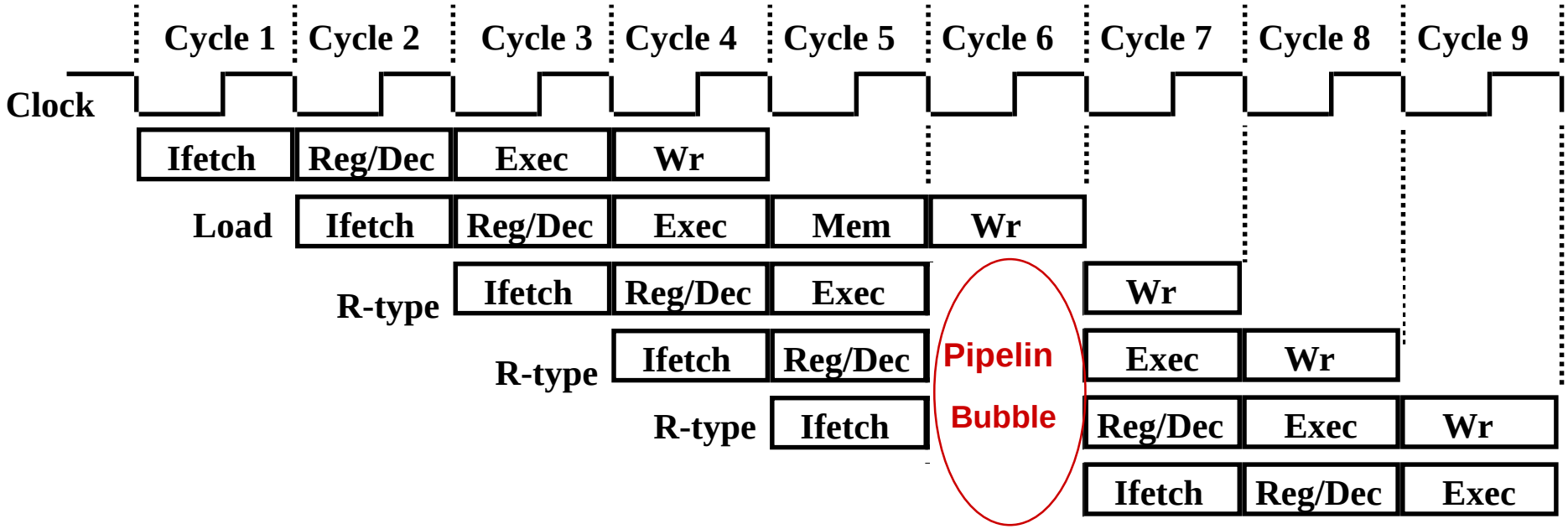
- 上述流水线有个问题：两条指令试图同时写寄存器，因为
  - Load在第5阶段用寄存器写口
  - R-type在第4 阶段用寄存器写口

或称为资源冲突！

- 把一个功能部件同时被多条指令使用的现象称为结构冒险(Struture Hazard)
- 为了流水线能顺利工作，规定：
  - 每个功能部件每条指令只能用一次（如：写口不能用两次或以上）
  - 每个功能部件必须在相同的阶段被使用（如：写口总是在第五阶段被使用）

可以用以下两种方法解决上述结构冒险问题！

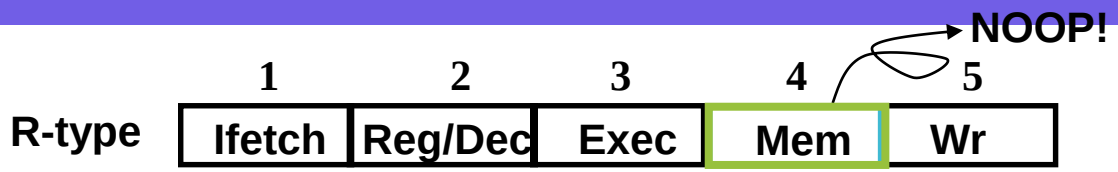
解决方案1: 在流水线中插入“Bubble”（气泡）

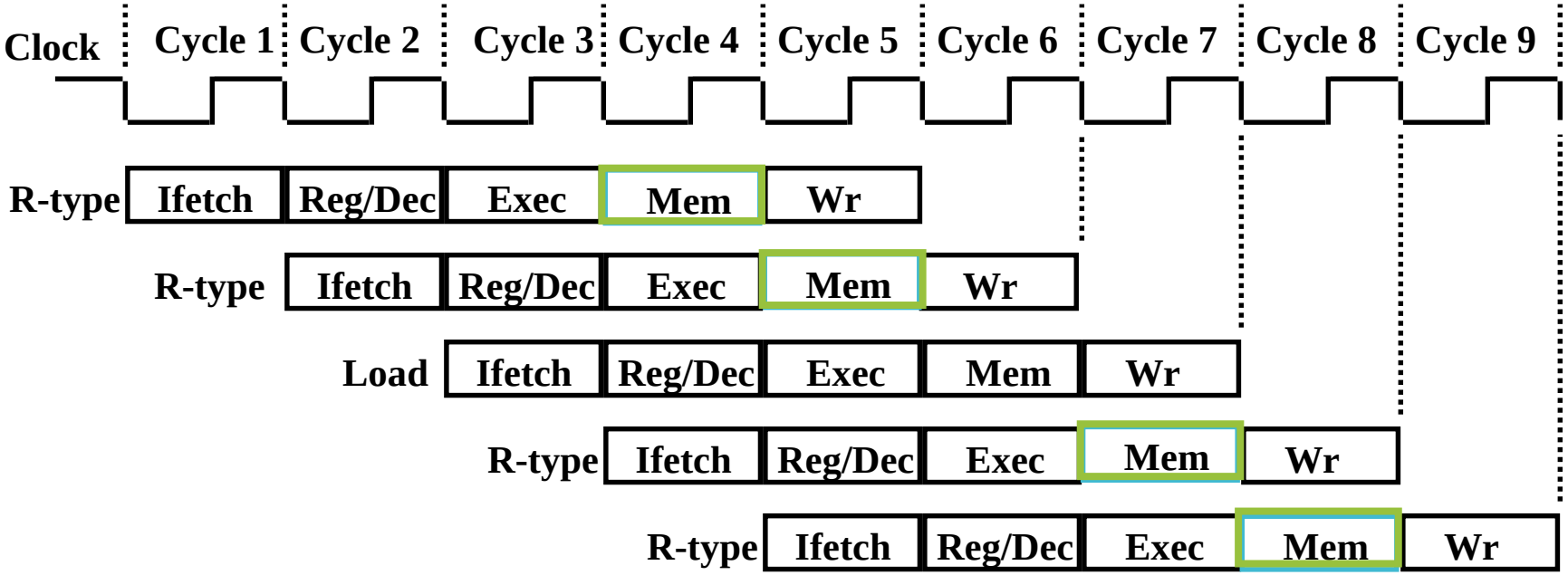


- 插入“**Bubble**”到流水线中，以禁止同一周期有两次写寄存器。缺点：
  - 控制逻辑复杂
  - 第5周期没有指令被完成（CPI不是1，而实际上是2）

方案不可行！

解决方案2: R-type的Wr操作延后一个周期执行





- 加一个NOP阶段以延迟“写”操作:
  - 把“写”操作安排在第5阶段, 这样使R-Type的Mem阶段为空NOP

这样使流水线中的每条指令都有相同多个阶段!

流水线冒险的处理

- 流水线冒险的几种类型
- 数据冒险的现象和对策
  - 数据冒险的种类
    - 相关的数据是ALU结果：可以通过转发解决
    - 相关的数据是DM读出的内容：随后的指令需被阻塞一个时钟
  - 数据冒险和转发
    - 转发检测 / 转发控制
  - 数据冒险和阻塞
    - 阻塞检测 / 阻塞控制
- 控制冒险的现象和对策
  - 静态分支预测技术
  - 动态分支预测技术
  - 缩短分支延迟技术
- 流水线中对异常和中断的处理
- 访问缺失对流水线的影响

流水线的三种冲突/冒险（Hazard）情况

- **Hazards：指流水线遇到无法正确执行后续指令或执行了不该执行的指令**
  - Structural hazards (hardware resource conflicts):

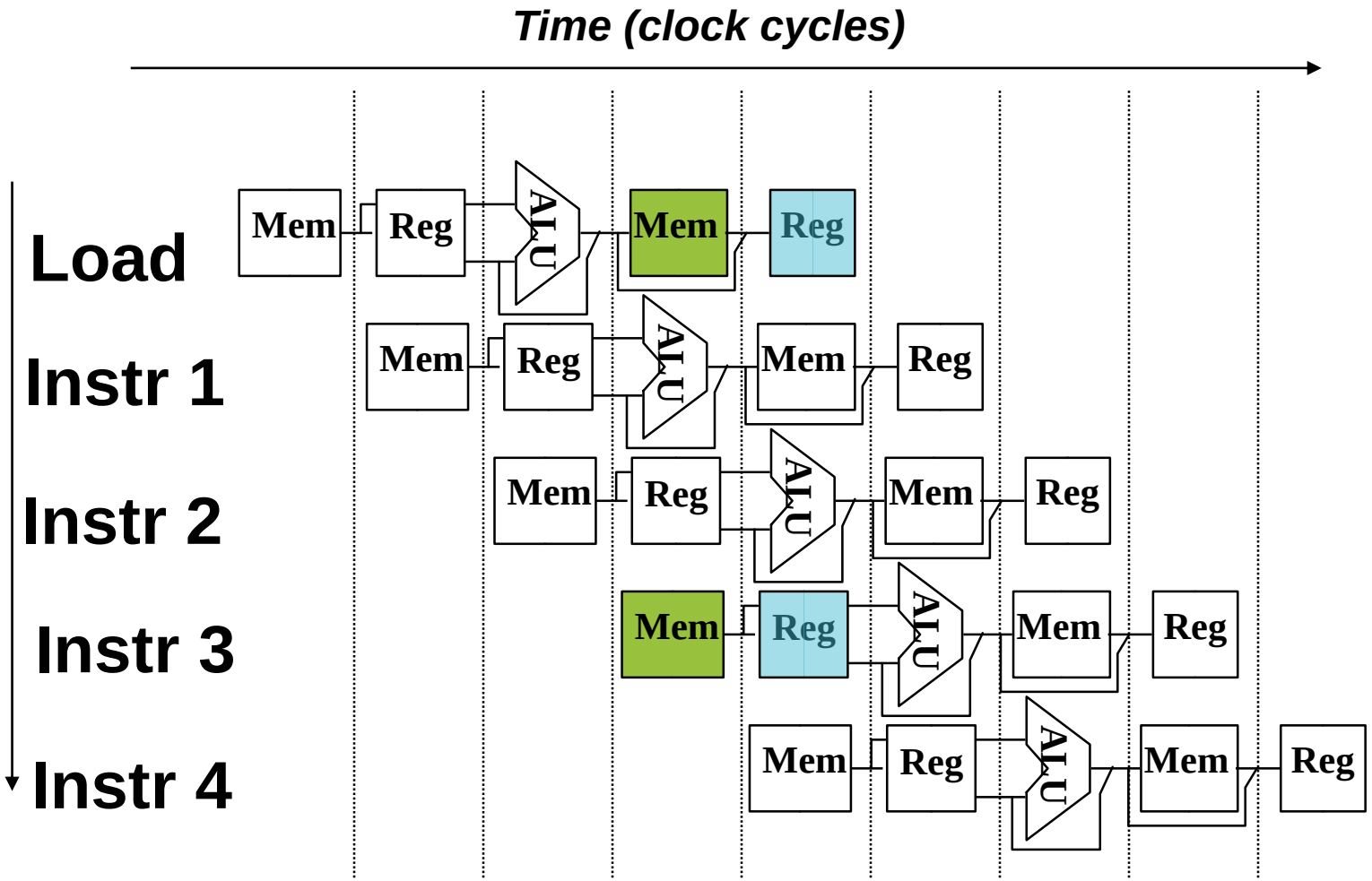
    现象：同一个部件同时被不同指令所使用
    - 一个部件每条指令只能使用1次，且只能在特定周期使用
    - 设置多个部件，以避免冲突。如指令存储器IM 和数据存储器DM分开
  - Data hazards (data dependencies):

    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
    - 采用转发(Forwarding/Bypassing)技术
    - Load-use冒险需要一次阻塞(stall)
    - 编译程序优化指令顺序
  - Control (Branch) hazards (changes in program flow):

    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
    - 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)

SKIP

Structural Hazard（结构冒险）现象



如果只有一个存储器，则在Load指令取数据同时又取指令的话，则发生冲突！

如果不对寄存器堆的写口和读口独立设置的话，则发生冲突！

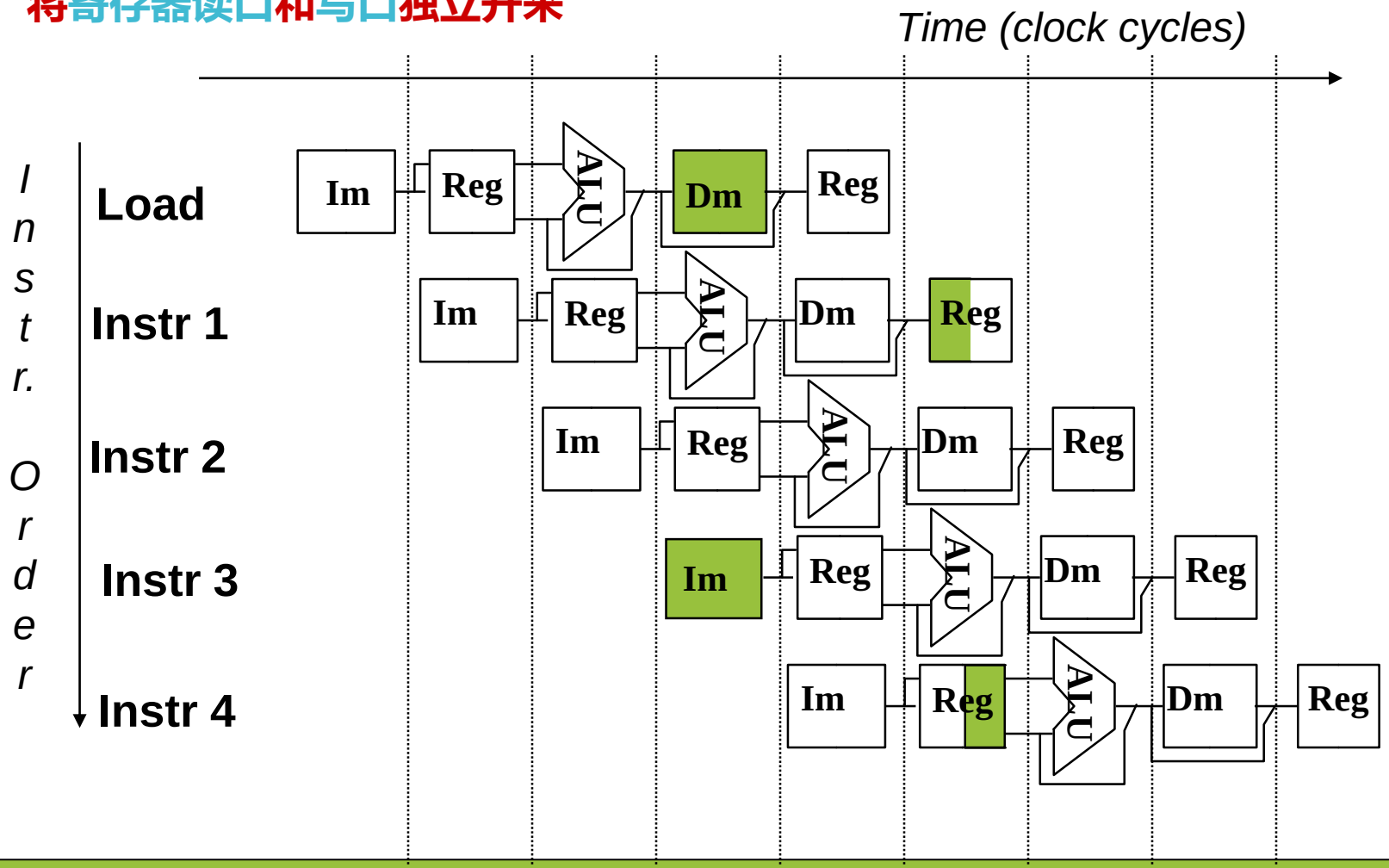
结构冒险也称为硬件资源冲突：同一个执行部件被多条指令使用。

Structural Hazard的解决方法

为了避免结构冒险，规定流水线数据通路中功能部件的设置原则为：
每个部件在特定的阶段被用！（如：ALU总在第三阶段被用！）

将Instruction Memory (Im) 和 Data Memory (Dm)分开

将寄存器读口和写口独立开来



Data Hazard现象

举例说明：以下指令序列中，寄存器r1会发生数据冒险

想一下，哪条指令的r1是老的值？
哪条是新的值？

```
add r1 ,r2,r3
sub r4, r1 ,r3
and r6, r1 ,r7
or  r8, r1 ,r9
xor r10, r1 ,r11
```

读r1时，add指令正在执行加法(EXE)，老值!

读r1时，add指令正在传递加法结果(MEM), 老值!

读r1时，add指令正在写加法结果到r1(WB), 老值!

读r1时，add指令已经把加法结果写到r1, 新值

画出流水线图能很清楚理解！

补充：三类数据冒险现象

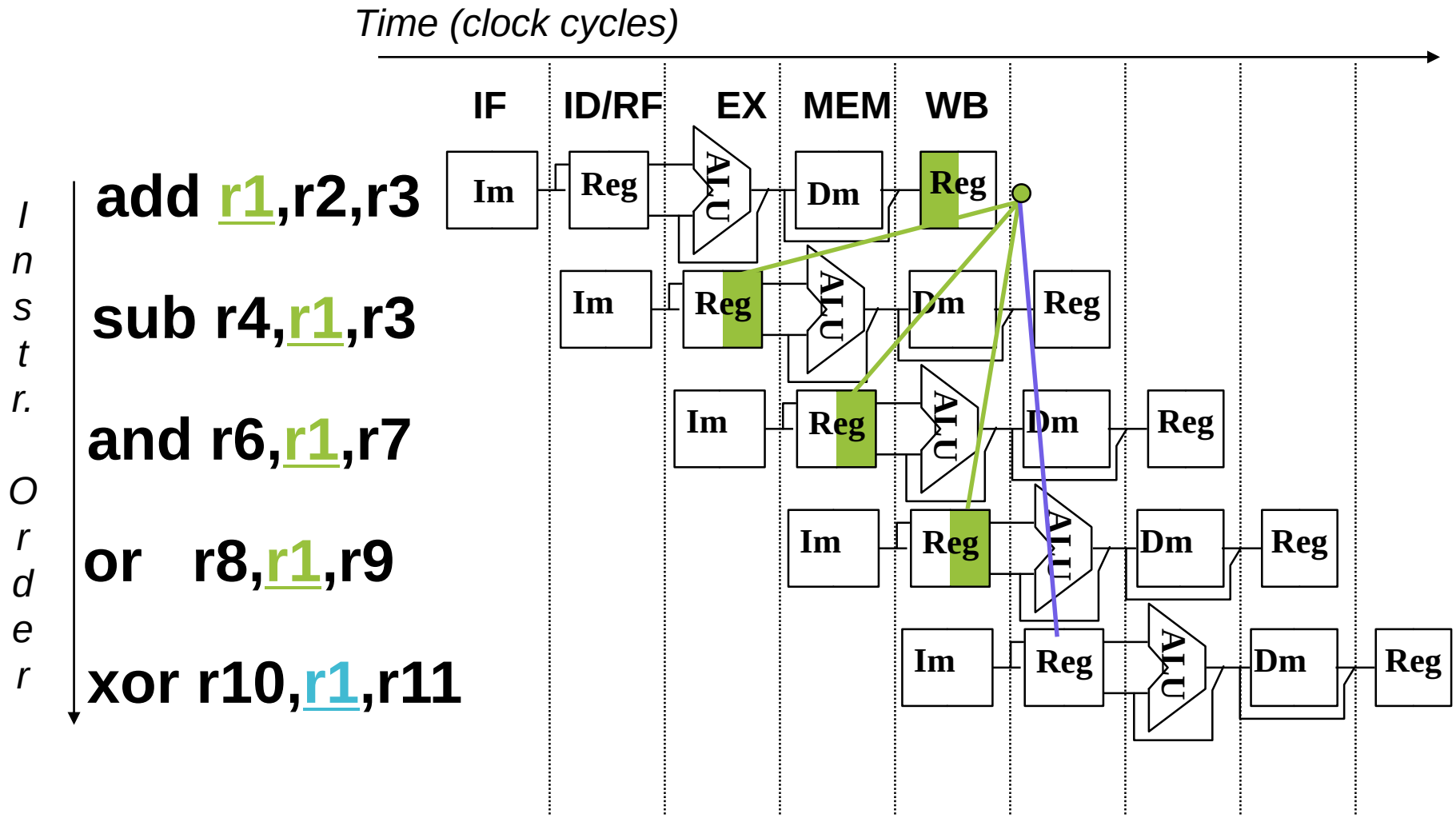
RAW：写后读（基本流水线中经常发生，如上例）

WAR：读后写（基本流水线中不会发生，多个功能部件时会发生）

WAW：写后写（基本流水线中不会发生，多个功能部件时会发生）

本讲介绍基本流水线，所以仅考虑RAW冒险

Data Hazard on r1



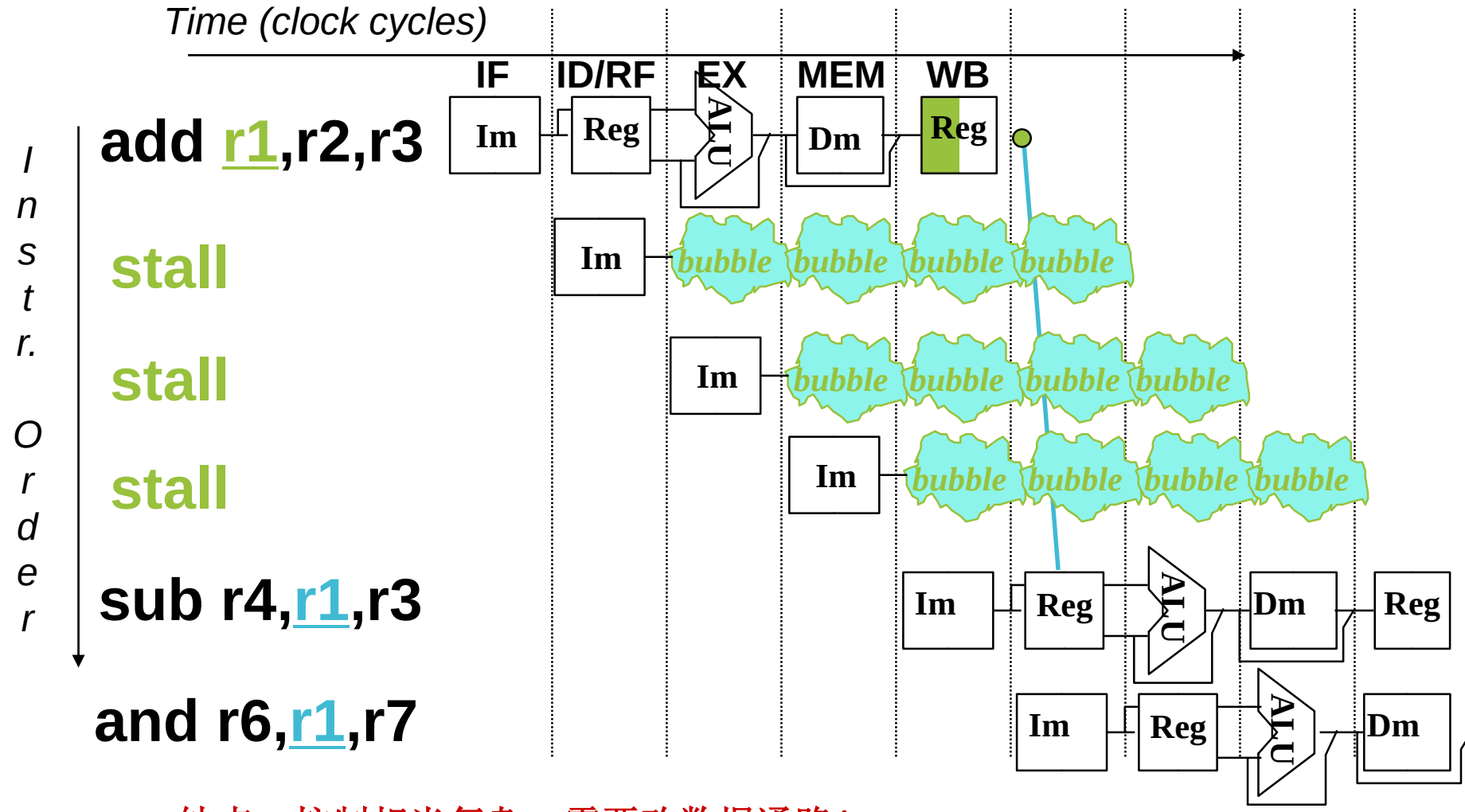
最后一条指令的r1才是新的值！

如何解决这个问题？

方案1: 在硬件上采取措施，使相关指令延迟执行

- 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！

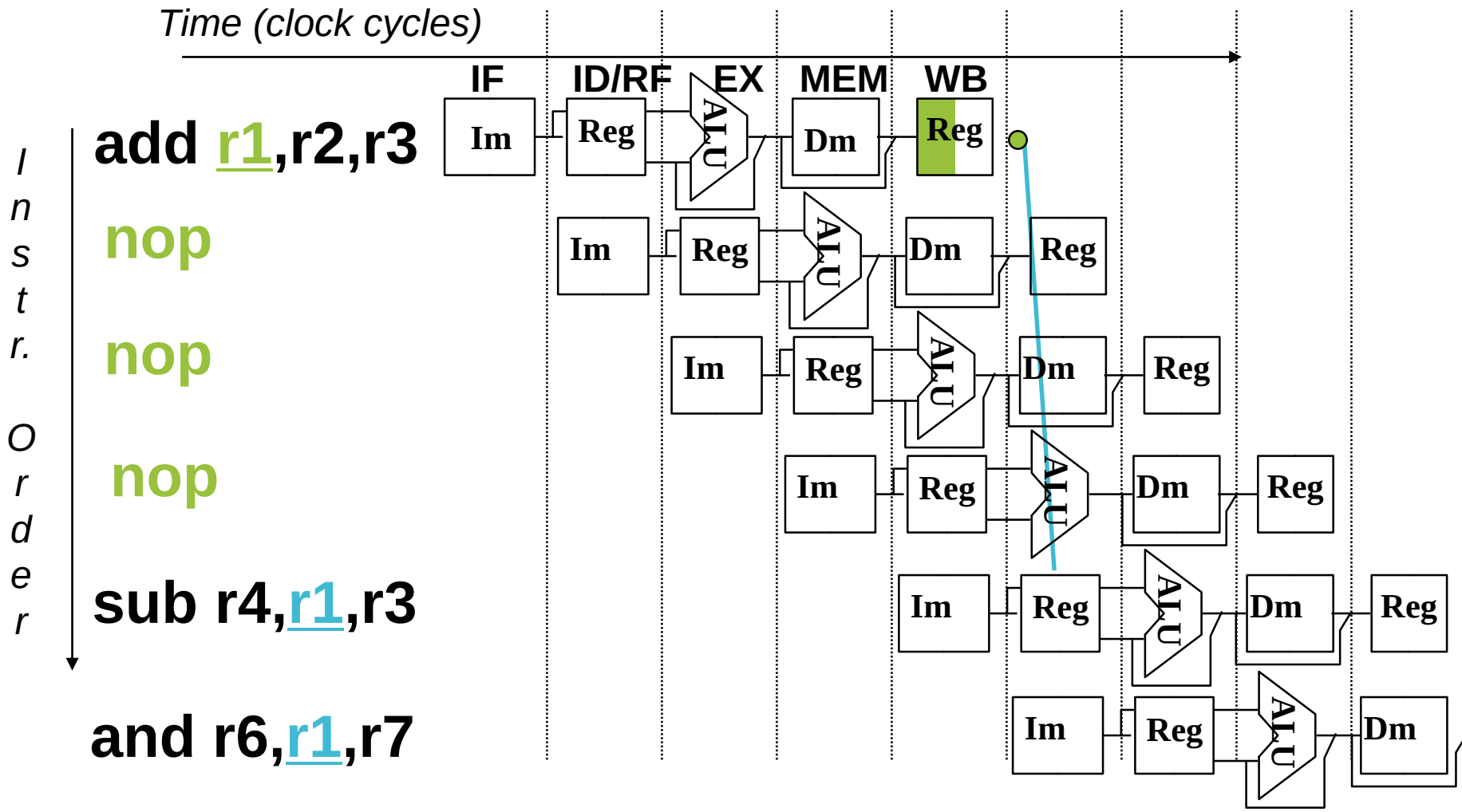
这种做法称为流水线阻塞，也称为“气泡Bubble”



- 缺点：控制相当复杂，需要改数据通路！

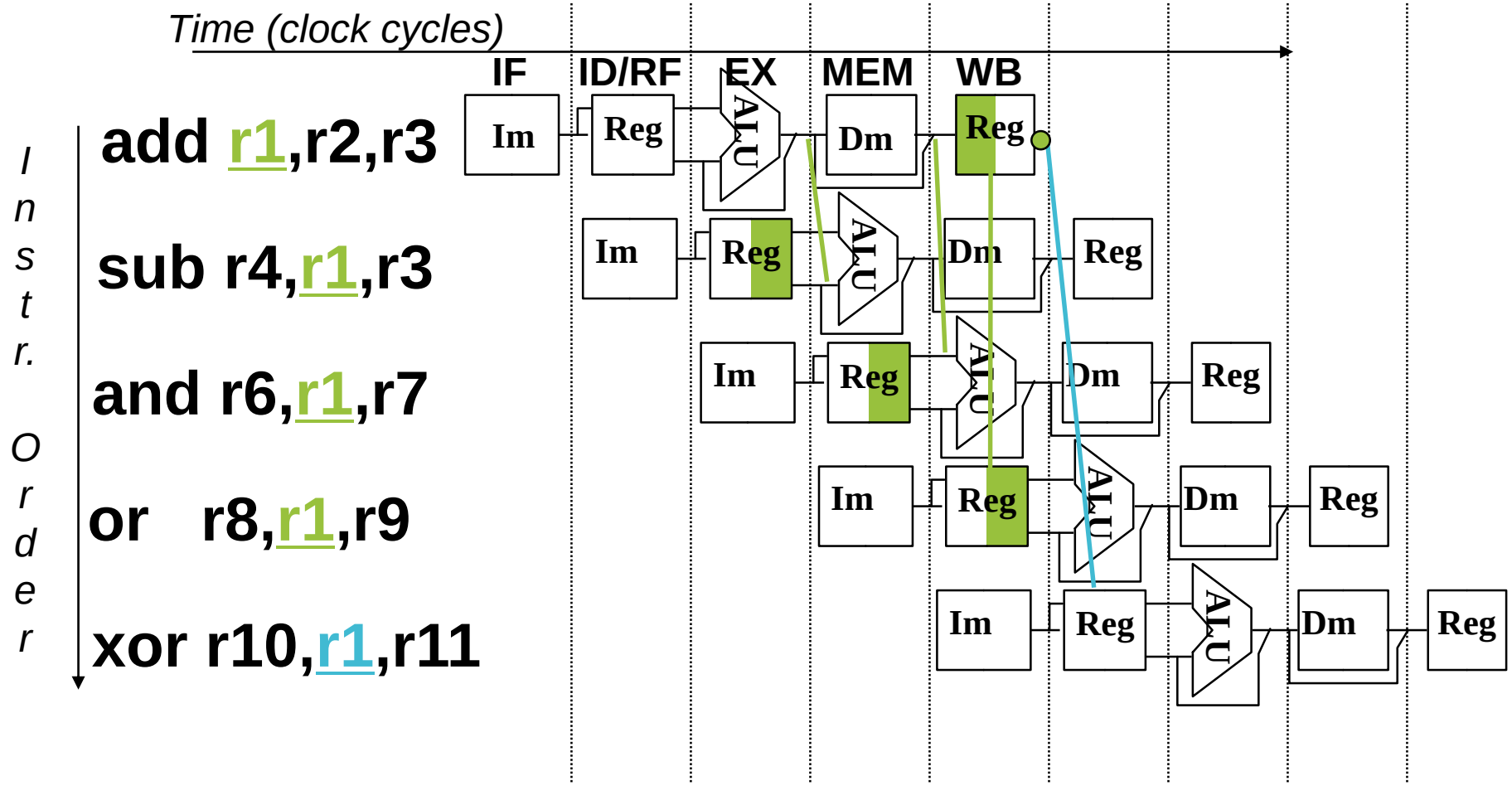
方案 2: 软件上插入无关指令

- 最差的做法：由编译器插入三条NOP指令，浪费三条指令的空间和时间



方案3：利用DataPath中的中间数据

- 仔细观察后发现：流水段寄存器中已有需要的值r1！ 在哪个流水段R中？



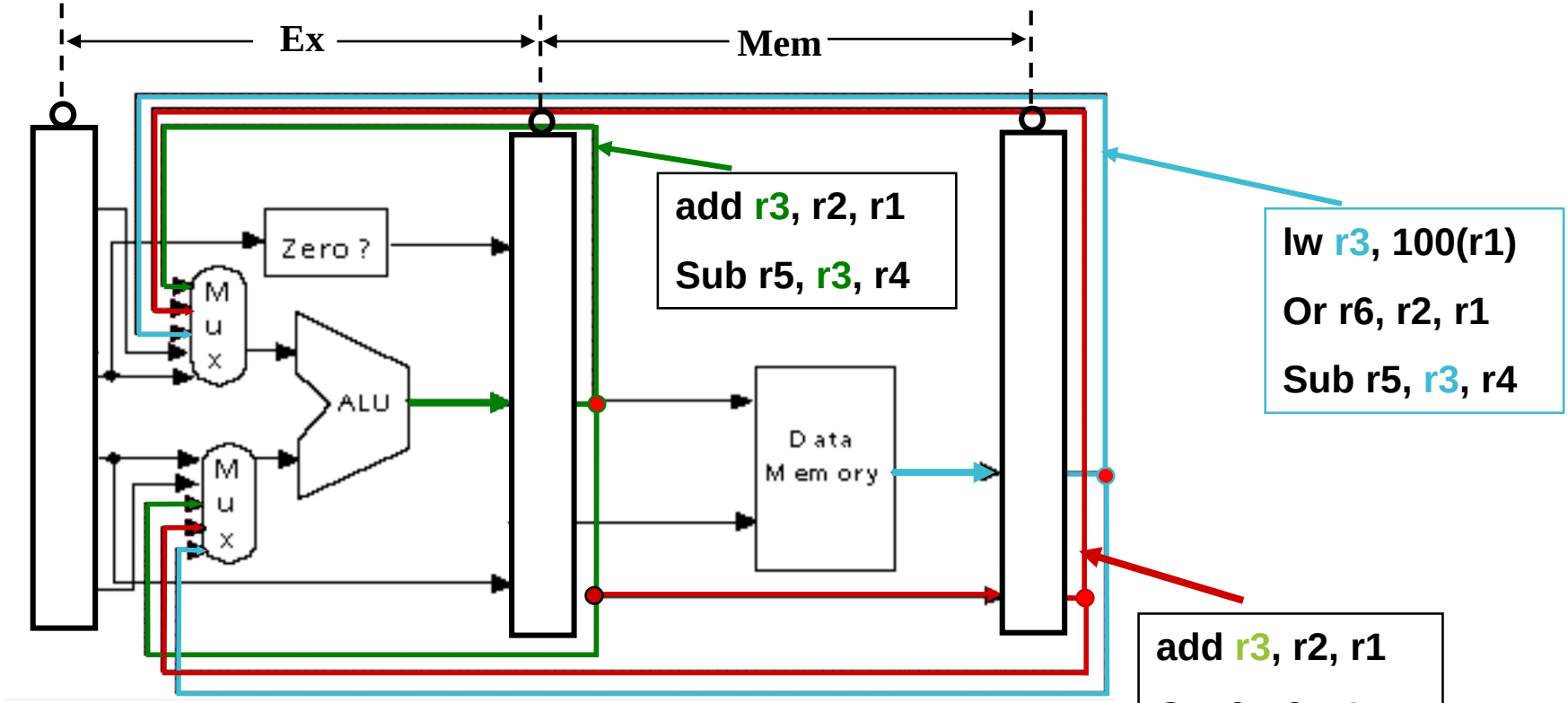
1. 把数据从流水段寄存器中直接取到ALU的输入端
2. 寄存器写/读口分别在前/后半周期，使写入被直接读出

称为转发（Forwading）或旁路（Bypassing）

BACK

硬件上的改动以支持“转发”技术

- 加MUX，使流水段寄存器值返送ALU输入端
- 假定流水段寄存器能读出新写入的值（否则，需要更多的转发数据)



如果指令序列为：

lw r3, 100(r1)

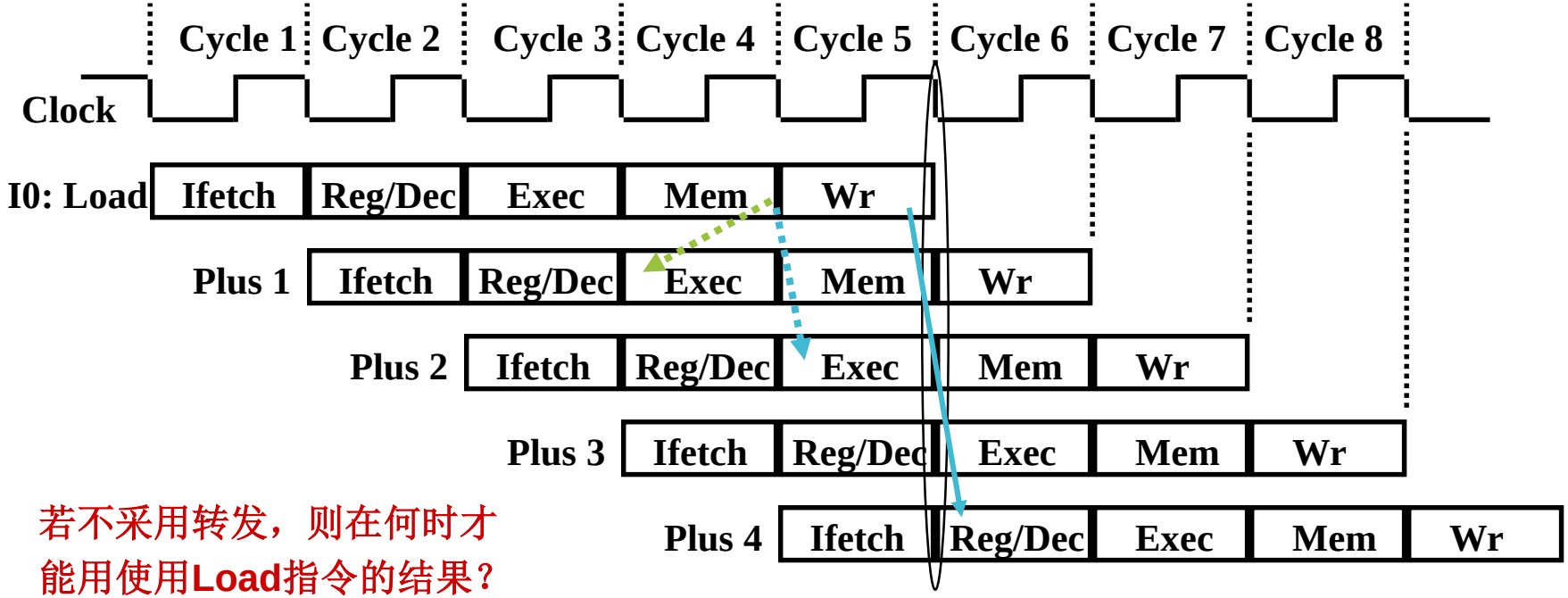
Or r6, r3, r1

Sub r5, r3, r4

能用“转发”技术解决第1、2两条指令间的数据冒险吗？

请看后面的幻灯片！

复习: Load指令引起的延迟



若不采用转发，则在何时才能用使用Load指令的结果？

- **Load指令最早在哪个流水线寄存器中开始有后续指令需要的值？**

实际上，在第四周期结束时，数据在流水段寄存器中已经有值。

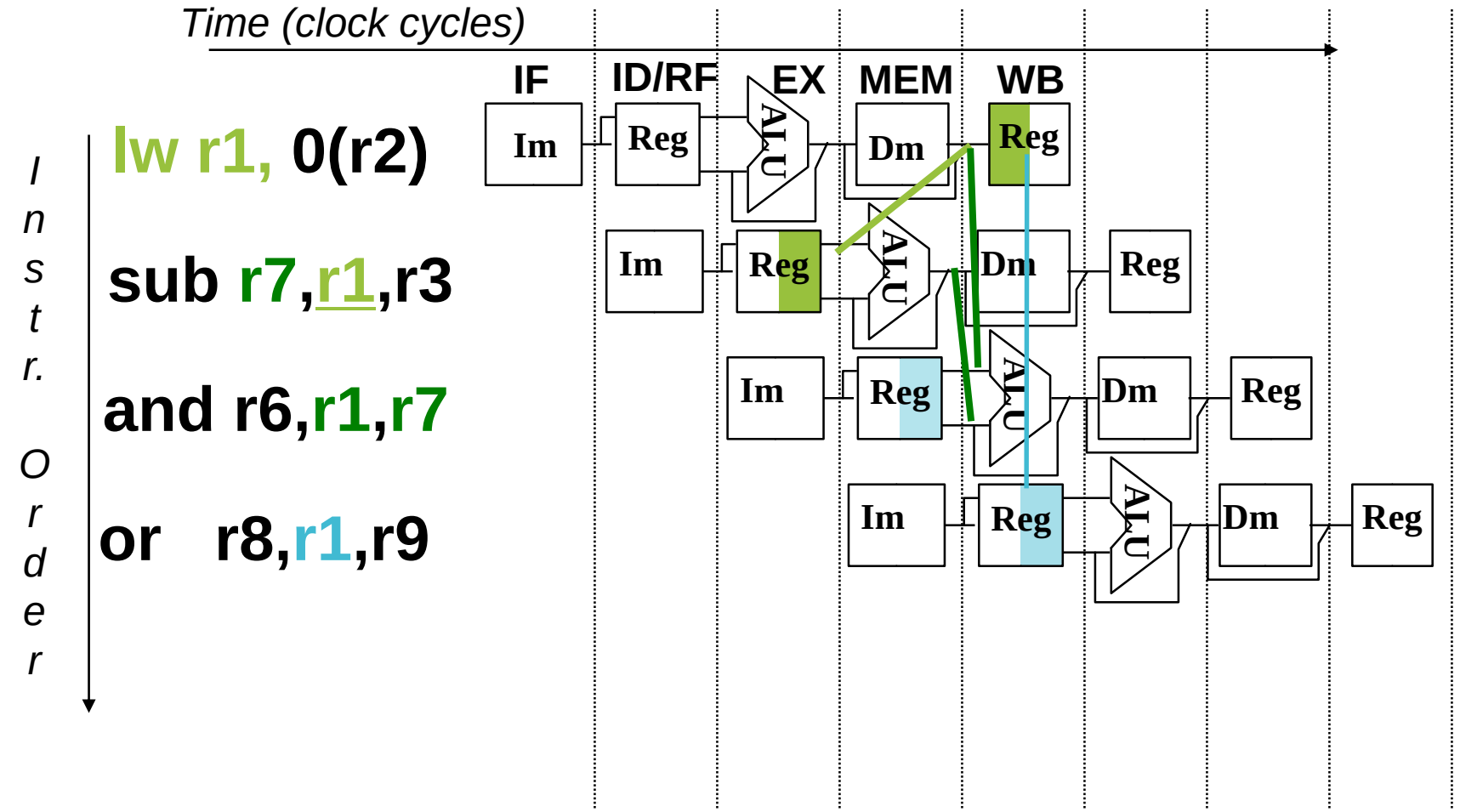
采用数据转发技术可以使load指令后面第二条指令得到所需的值

但不能解决load指令和随后的第一条指令间的数据冒险，要延迟执行一条指令！

这种load指令和随后指令间的数据冒险，称为“装入- 使用数据冒险(load- use Data Hazard)”

BACK

“Forwarding”技术使Load-use冒险只需延迟一个周期

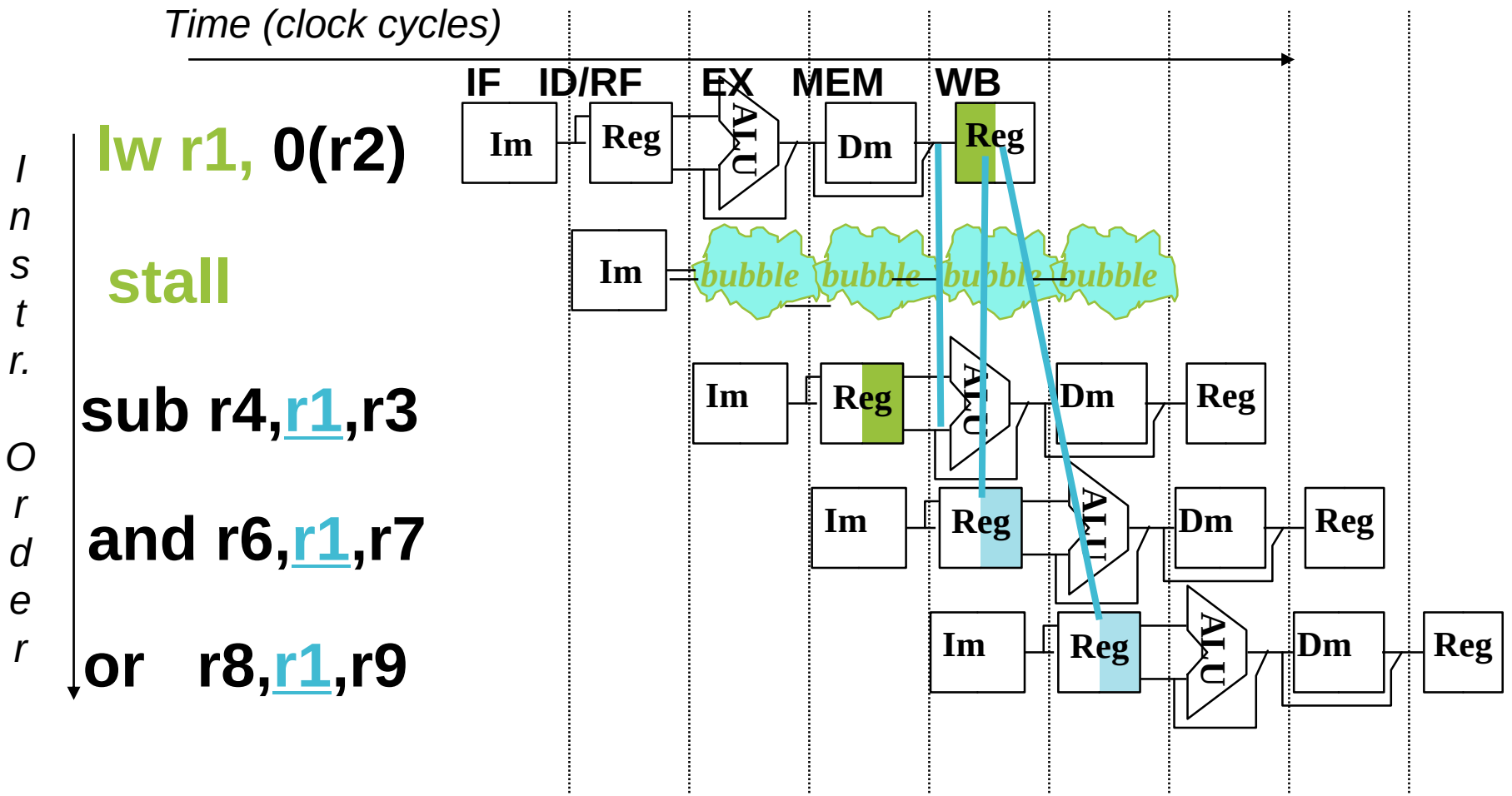


采用“转发”后仅第二条指令 **SUB r7,r1,r3** 不能按时执行！需要阻塞一个周期。

发生“装入- 使用数据冒险”时，需要对**load**后的指令阻塞一个时钟周期！

方案1: 硬件阻止指令执行来解决load-use

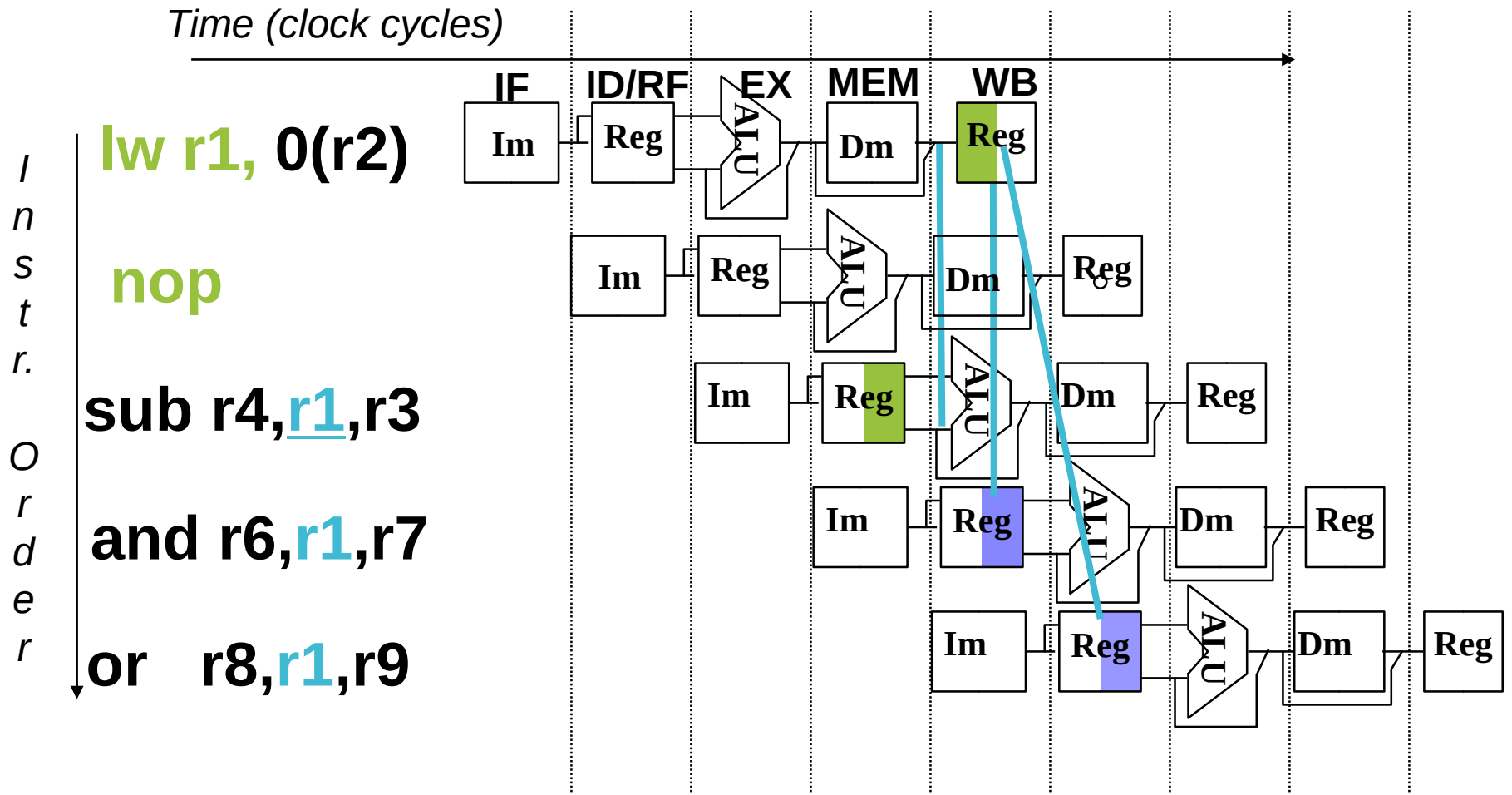
用硬件阻塞一个周期（指令被重复执行一次）



方案2: 软件上插入NOP指令来解决load-use

- 用软件插入一条NOP指令！（有些处理器不支持硬件阻塞处理）

例如：MIPS 1 处理器没有硬件阻塞处理，而由编译器（或汇编程序员） 来处理。



方案3：编译器进行指令顺序调整来解决load-use

以下源程序可生成两种不同的代码，优化的代码可避免Load阻塞

a = b + c;

d = e – f;

假定 **a, b, c, d ,e, f** 在内存

Slow code:

```
lw      $2, [b]
lw      $3, [ c ]
add     $1, $2, $3
sw      [a, $1
lw      $5, [e]
lw      $6, [f]
sub     $4, $5, $6
sw      [d], $4
```

调整后

Fast code:

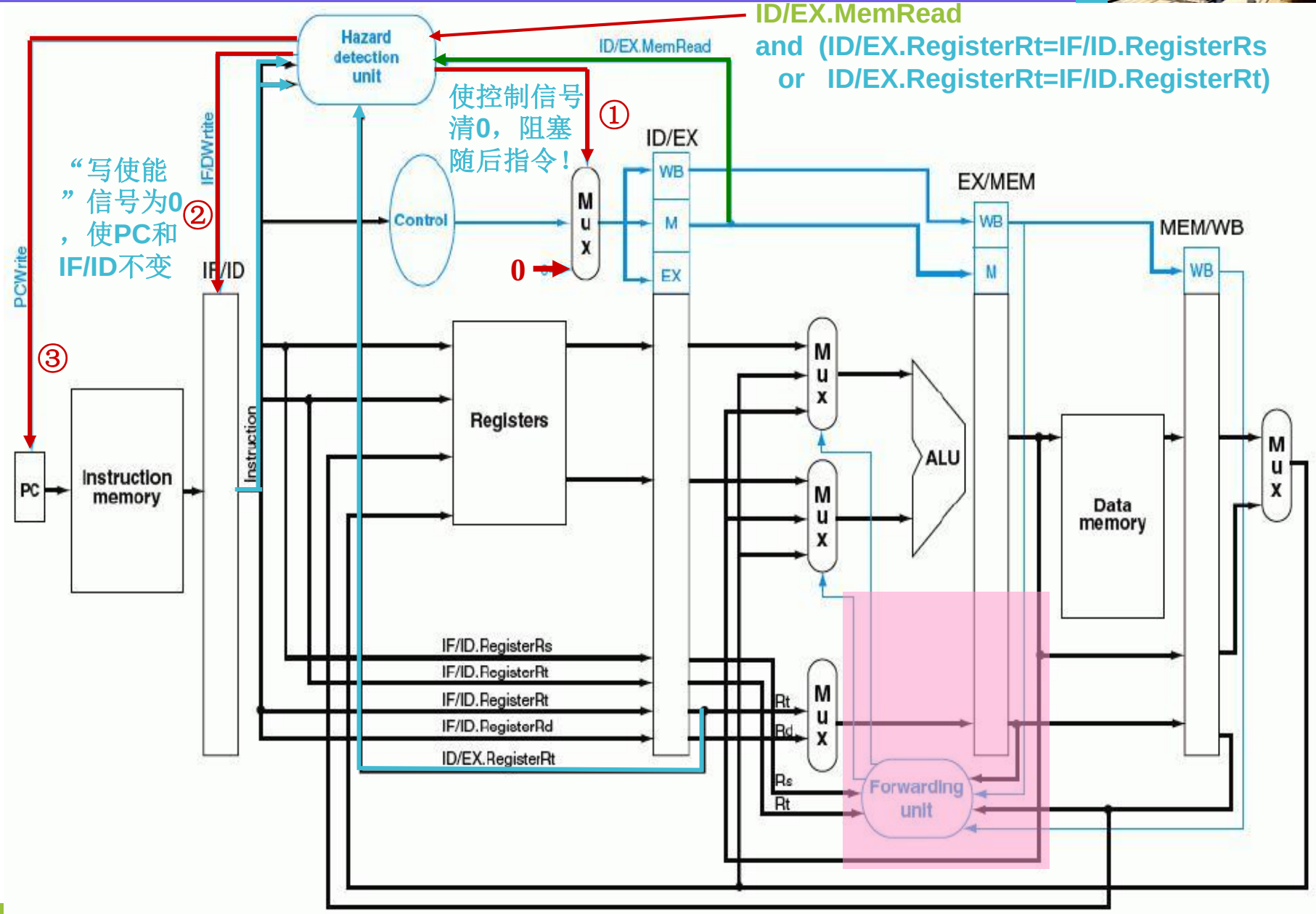
```
lw      $2, [b]
lw      $3, [c]
lw      $5, [e]
add     $1, $2, $3
lw      $6, [f]
sw      [a], $1
sub     $4, $5, $6
sw      [d], $4
```

编译器的优化很重要！

数据冒险的解决方法

- **方法1：硬件阻塞（stall）**
- **方法2：软件插入“NOP”指令**
- **方法3：编译优化：调整指令顺序，能解决所有数据冒险吗？**
- **方法4：合理实现寄存器堆的读/写操作，能解决所有数据冒险吗？**
  - 前半时钟周期写，后半时钟周期读
  - 若同一个时钟内前面指令写入的数据正好是后面指令所读数据，则不会发生数据冒险
- **方法5：转发（Forwarding或Bypassing 旁路）技术，能解决所有数据冒险吗？**
  - 若相关数据是ALU结果，则如何？
    可通过转发解决
  - 若相关数据是上条指令DM读出内容，则如何？
    不能通过转发解决，随后指令需被阻塞一个时钟 或 加NOP指令

带“转发”和“阻塞”检测的流水线数据通路



Control Hazard的解决方法

- **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

另一种控制冒险：异常或中断控制冒险的处理

动态预测基本方法

- **采用一位预测位：总是按上次实际发生的情况来预测下次**
  - 1表示最近一次发生过转移（taken），0表示未发生（not taken）
  - 预测时，若为1，则预测下次taken，若为0，则预测下次not taken
  - 实际执行时，若预测错，则该位取反，否则，该位不变
  - 可用一个简单的预测状态图表示
  - 缺点：当连续两次的分支情况发生改变时，预测错误
    - 例如，循环迭代分支时，第一次和最后一次会发生预测错误，因为循环的第一次和最后一次都会改变分支情况，而在循环中间的各次总是会发生分支，按上次的实际情况预测时，都不会错。
- **采用二位预测位**
  - 用2位组合四种情况来表示预测和实际转移情况
  - 按照预测状态图进行预测和调整
  - 在连续两次分支发生不同时，只会有一次预测错误

采用比较多的是二位预测位，也有采用二位以上预测位。
如：Pentium 4 的BTB2采用4位预测位

BACK

小结

- **流水线冒险的几种类型：**
  - 资源冲突、数据相关、控制相关（改变指令流的执行方向）
- **数据冒险的现象和对策**
  - 数据冒险的种类
    - 相关的数据是ALU结果，可以通过转发解决
    - 相关的数据是DM读出的内容，随后的指令需被阻塞一个时钟
  - 数据冒险和转发
    - 转发检测 / 转发控制
  - 数据冒险和阻塞
    - 阻塞检测 / 阻塞控制
- **控制冒险的现象和对策**
  - 静态分支预测技术
  - 缩短分支延迟技术
  - 动态分支预测技术
- **异常和中断是一种特殊的控制冒险**
- **访存缺失（Cache缺失、TLB缺失、缺页）会引起流水线阻塞**

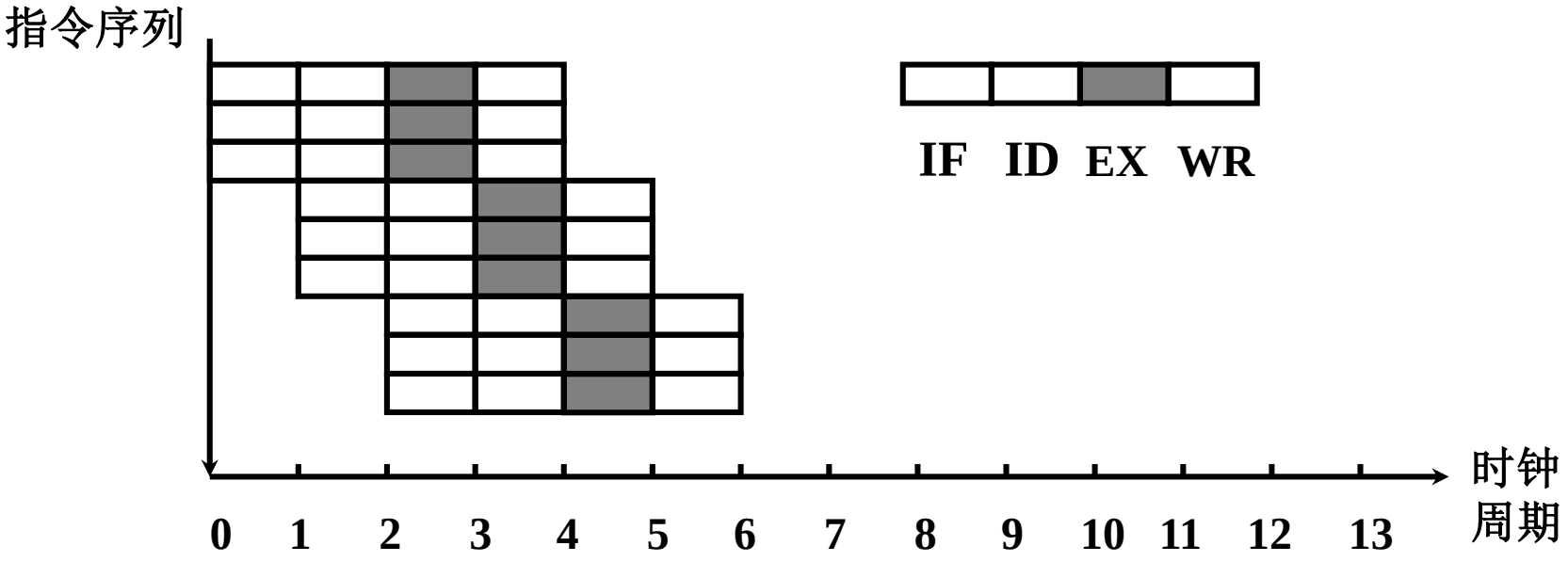
五、流水线的多发技术

- 流水线的多发技术：为了提高流水线性能，设法在一个时钟周期内产生更多条指令。
  - **超标量技术**
  - **超流水线技术**
  - **超长指令字**

1. 超标量技术(Super Scalar)

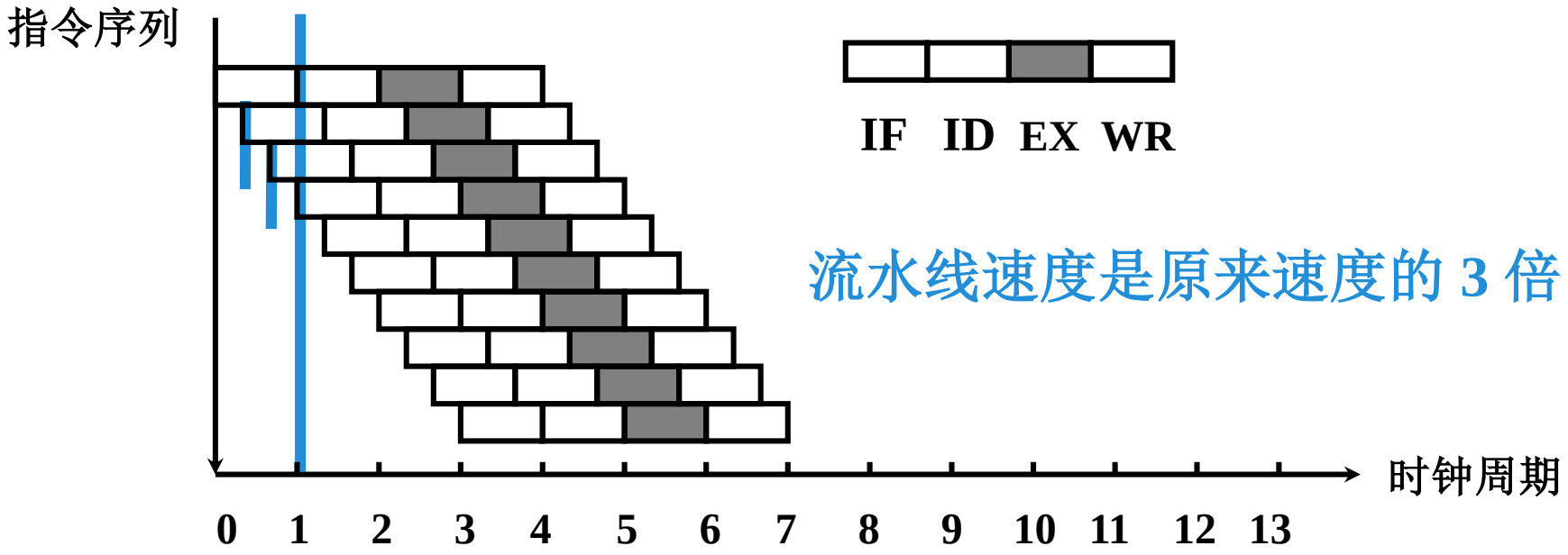
- 每个时钟周期内可 并发多条独立指令
- 配置多个功能部件，一个时钟周期内，有多个功能部件同时执行多条指令 。
- 不能调整 指令的 执行顺序

可以通过编译优化技术，把可并行执行的指令搭配起来



2. 超流水线技术（Super Pipe Lining）

- 将流水寄存器插入流水线段，好比将流水线再次分段
- 在 一个时钟周期 内 再分段 （3 段）
  在一个时钟周期内 一个功能部件使用多次（3 次）
- 不能调整 指令的 执行顺序　靠编译程序解决优化问题



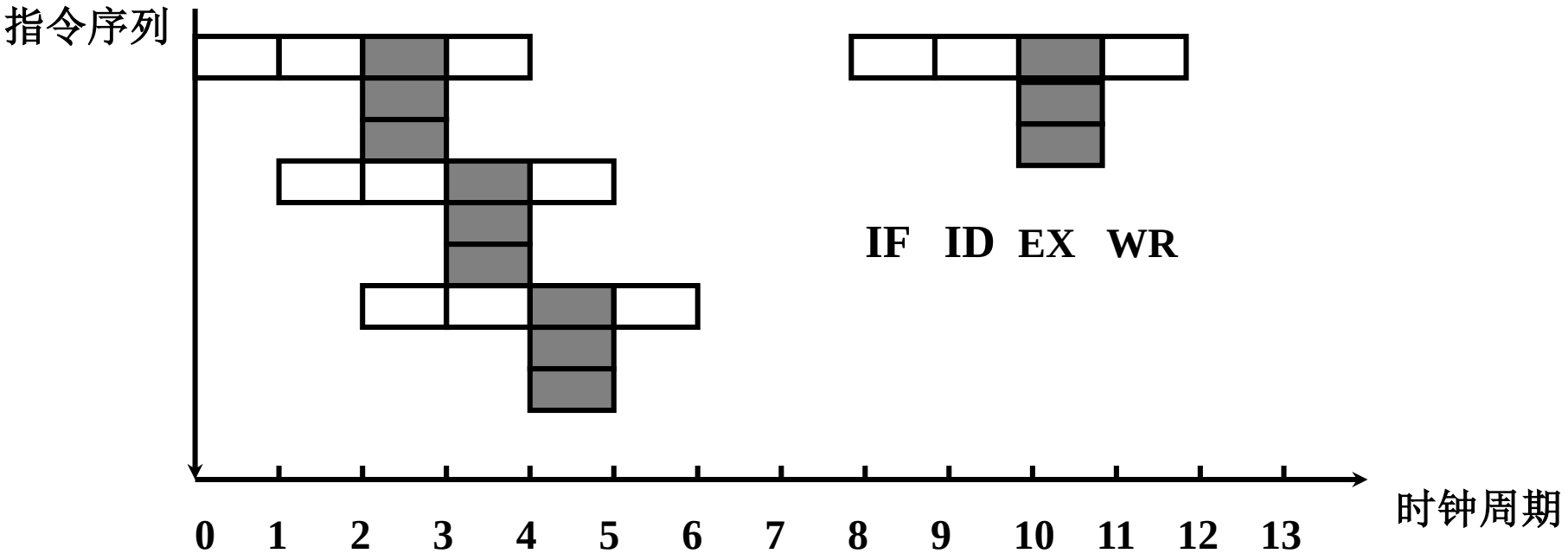
3. 超长指令字技术（VLIW）

- 由编译程序 挖掘 出指令间 潜在 的 并行性

  将 多条 能 并行操作 的指令组合成 一条

  具有 多个操作码字段 的 超长指令字（可达几百位）

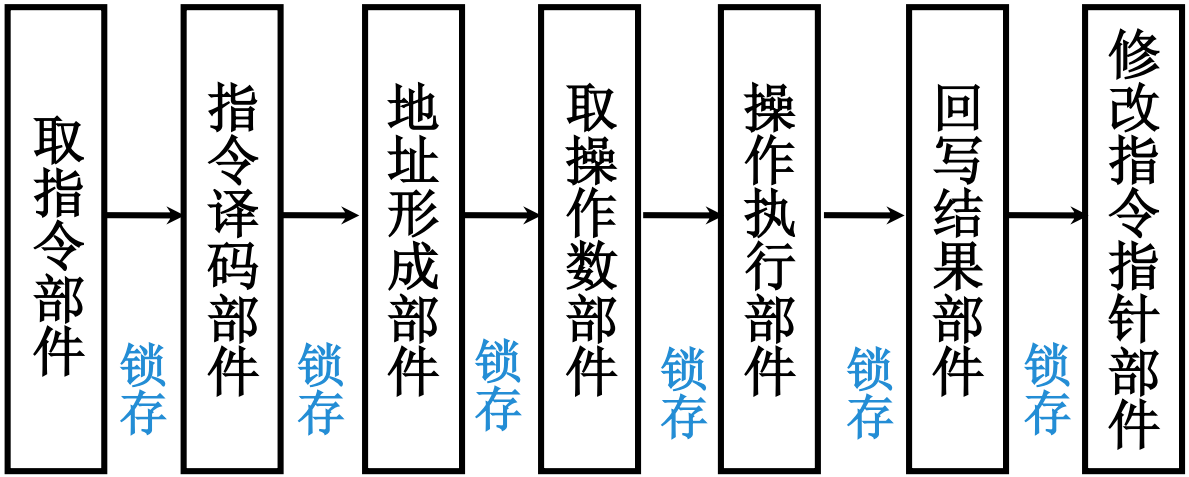
- 形成的超长指令字控制VLIW机中独立工作的多个处理部件



六、流水线结构

1. 指令流水线结构

完成一条指令分 7 段， 每段需一个时钟周期



若 流水线不出现断流　　1 个时钟周期出 1 结果

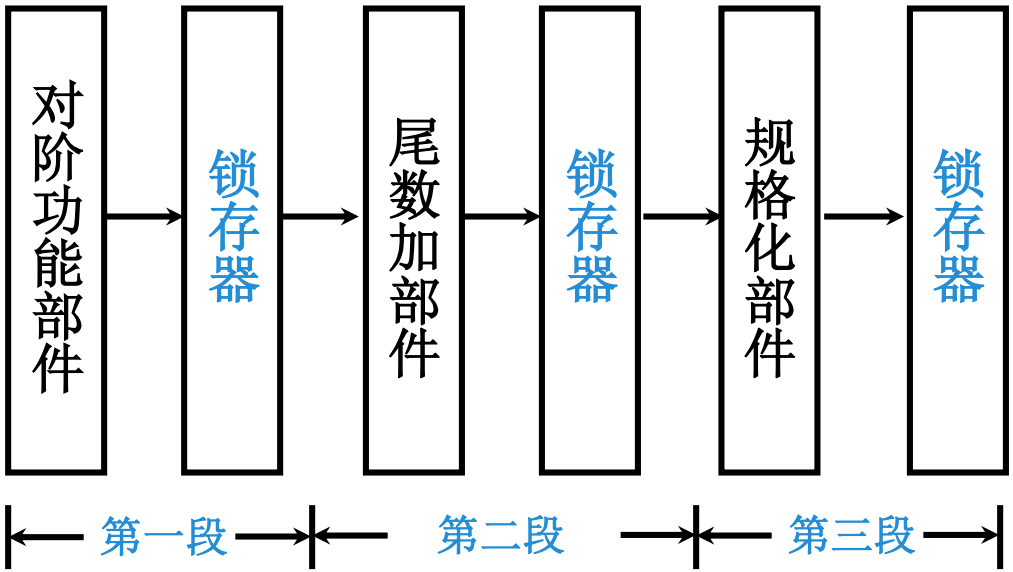
不采用流水技术　　7 个时钟周期出 1 结果

理想情况下，7 级流水 的速度是不采用流水技术的 7 倍

2. 运算流水线

完成 浮点加减 运算 可分

对阶、尾数求和、规格化 三段



分段原则　每段 操作时间 尽量 一致

Computer Organization

Thank You !

Institute of Computer Architecture

合肥工业大学
系统结构研究所
陈 田

程序的机器级表示

- **MIPS指令格式**
  - R-类型 / I-类型 / J-类型
- **MIPS寄存器**
  - 长度 / 个数 / 功能分配
- **MIPS操作数**
  - 寄存器操作数 / 存储器操作数 / 立即数 / 文本 / 位
- **MIPS指令寻址方式**
  - 立即数寻址 / 寄存器寻址 / 相对寻址 / 伪直接寻址 / 偏移寻址
- **MIPS指令类型**
  - 算术 / 逻辑 / 数据传送 / 条件分支 / 无条件转移
- **MIPS汇编语言形式**
  - 操作码的表示 / 寄存器的表示 / 存储器数据表示
- **机器语言的解码（反汇编）**
- **高级语言、汇编语言、机器语言之间的转换**
- **过程调用与堆栈**

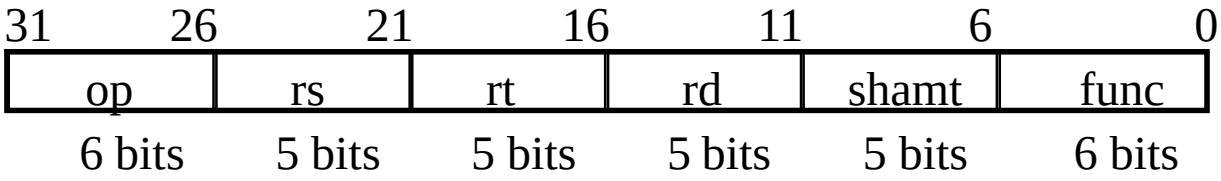
MIPS指令格式

- 所有指令都是32位宽，须按字地址对齐

◆ 有三种指令格式

– R-Type

R-Type指令

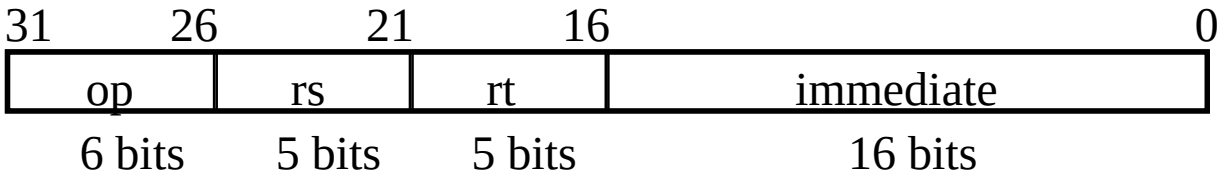


两个操作数和结果都在寄存器的运算指令。如：sub rd, rs, rt

– I-Type

- 运算指令：一个寄存器、一个立即数。如：ori rt, rs, imm16
- LOAD和STORE指令。如：lw rt, rs, imm16
- 条件分支指令。如：beq rs, rt, imm16

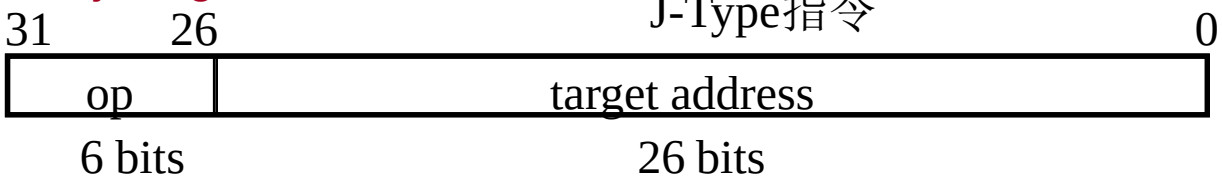
I-Type指令



– J-Type

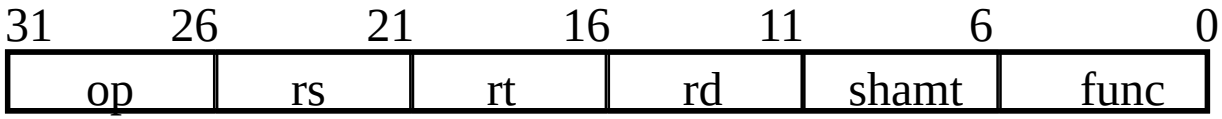
无条件跳转指令。如：j target

J-Type指令

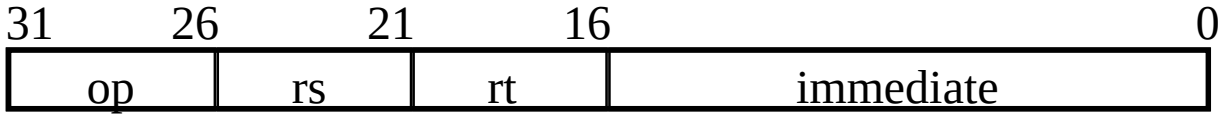


MIPS指令字段含义

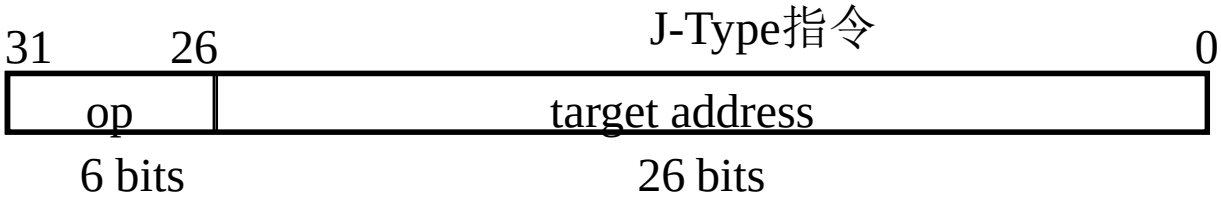
R-Type指令



I-Type指令



J-Type指令



OP：操作码

rs：第一个源操作数寄存器

rt：第二个源操作数寄存器

rd：结果寄存器

shamt：移位指令的位移量

func：R-Type指令的OP字段是特定的“000000”，具体操作由func字段给定。例如：func=“100000”时，表示“加法”运算。

操作码的不同编码定义不同的含义，操作码相同时，再由功能码定义不同的含义!

immediate：立即数或load/store指令和分支指令的偏移地址

target address：无条件转移地址的低26位。将PC高4位拼上26位直接地址，最后添2个“0”就是32位目标地址。为何最后两位要添“0”？

指令按字地址对齐，所以每条指令的地址都是4的倍数（最后两位为0）。

OP字段的含义（MIPS指令的操作码编码/解码表）

op(31:26)　　op=0: R型； op=2/3：J型； 其余：I型

28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	sltiu	andi	ori	xori	load upper imm
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	lbu	lhu	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

R-Type指令的解码（op=0时，func字段的编码/解码表）

op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump reg.	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	sltu				
6(110)								
7(111)								

BACK to Assemble

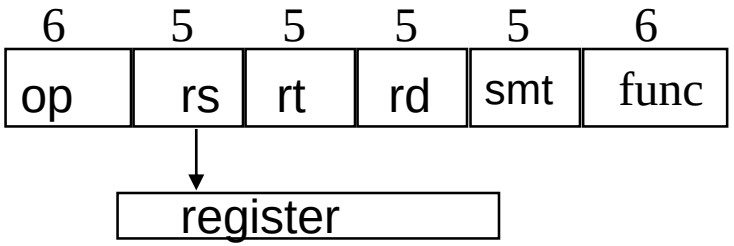
MIPS Addressing Modes（寻址方式）

有专门的寻址方式字段（Mod）吗？

没有！由指令格式来确定，而指令格式由op来确定！

R-format:

Register

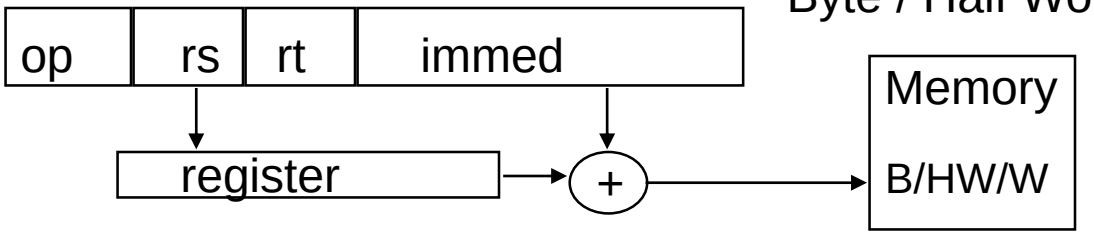


I-format:

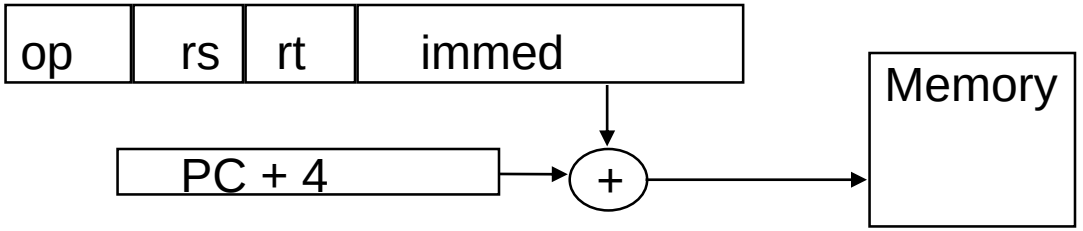
Immediate

Base或index

Byte / Half Word / Word

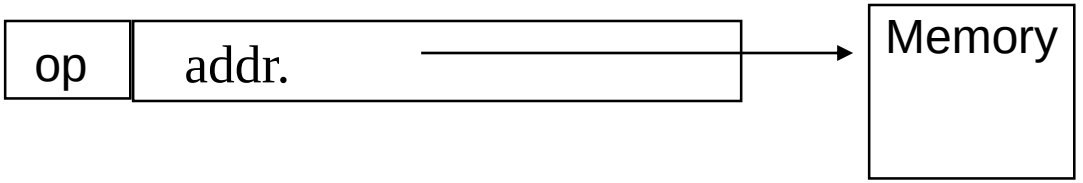


PC-relative



J-format:

Pseudodirect

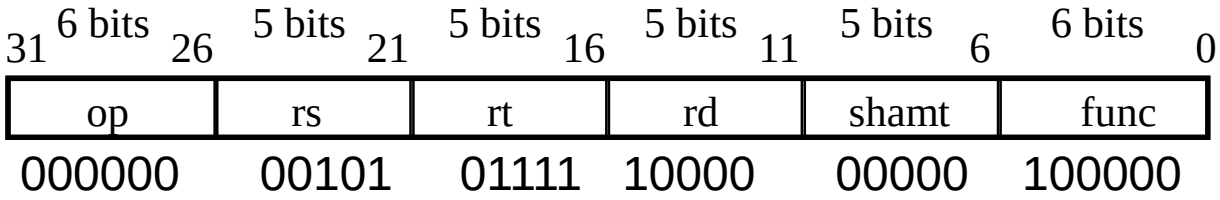


Example：汇编形式与指令的对应

- **若从存储器取来一条指令为00AF8020H，则对应的汇编形式是什么？**

32位指令代码：0000 0000 1010 1111 1000 0000 0010 0000

指令的前6位为000000，根据指令解码表知，是一条R-Type指令，按照R-Type指令的格式



得到： rs=00101, rt=01111, rd=10000, shamt=00000, funct=100000

1. 根据R-Type指令解码表，知是 “add”操作（非移位操作)

2. rs、rt、rd的十进制值分别为5、15、16，从MIPS寄存器功能表知:

rs、rt、rd分别为：$a1、$t7、$s0

故对应的汇编形式为：

add $s0，$a1，$t7

功能：$a1 + $t7 → $s0

这个过程称为“反汇编”，可用来破解他人的二进制代码（可执行程序）.

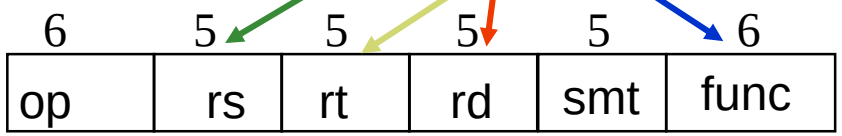
Example：汇编形式与指令的对应

❖若MIPS Assembly Instruction:

则对应的指令机器代码是什么？

Add $t0,$s1,$s2 → 汇编器 → ？

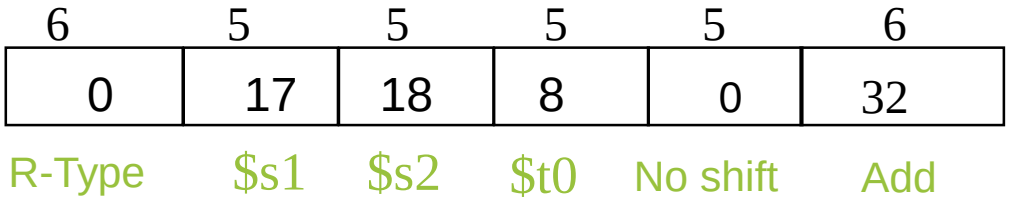
从助记符表中查到Add是R型指令，即：



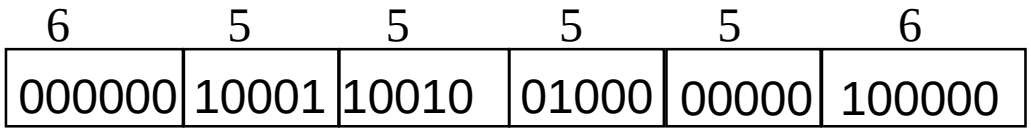
问题：如何知道是R型指令？

根据汇编指令中的操作码助记符查表能知道是什么格式！

Decimal representaton:

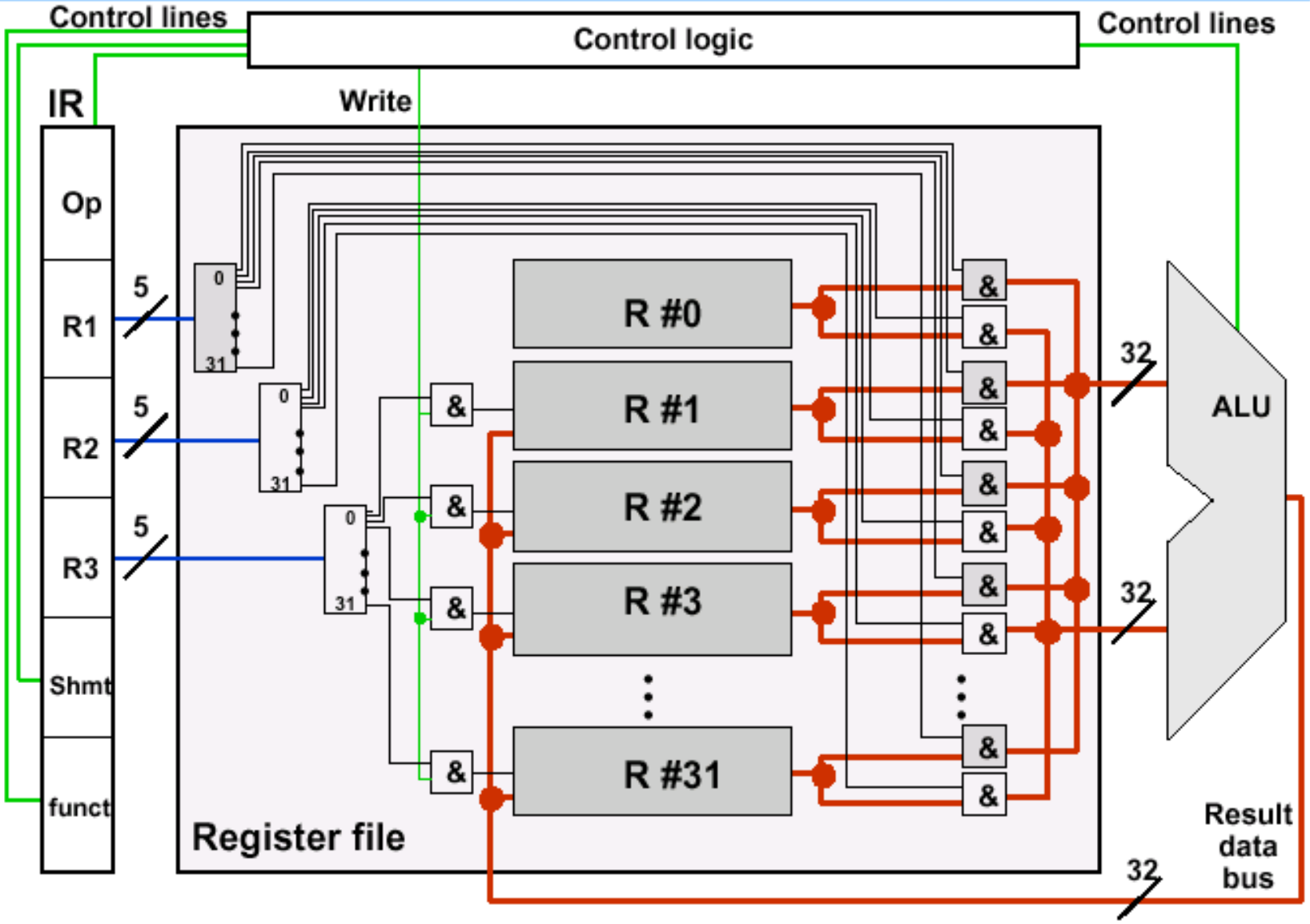


Binary representaton:



这个过程称为“汇编”，所有汇编源程序都必须汇编成二进制机器代码才能让机器直接执行！

MIPS Circuits for R-Type Instructions



问题：指令大致执行过程？

MIPS R-type指令实现电路的执行过程

Phase1: Preparation (1：准备阶段)

- 装入指令寄存器IR
- 以下相应字段送控制逻辑
  - op field (OP字段)
  - func field (func字段)
  - shmt field (shmt字段)
- 以下相应字段送寄存器堆
  - 第一操作数寄存器编号
  - 第二操作数寄存器编号
  - 存放结果的目标寄存器编号

这个过程描述仅是示意性的，实际上整个过程需要时钟信号的控制，并还有其他部件参与。

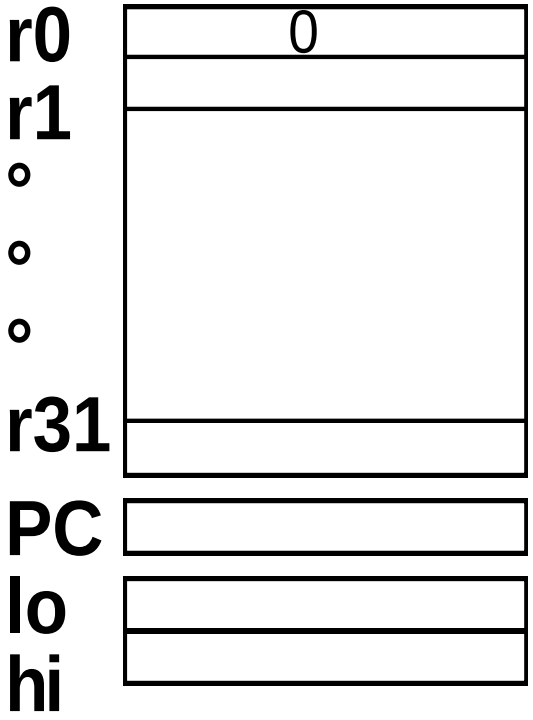
Phase2: Execution(2：执行阶段)

- 寄存器号被送选择器
- 对应选择器输出被激活
- 被选寄存器的输出送到数据线
- 控制逻辑提供：
  - ALU操作码
  - 写信号 等
- 结果被写回目标寄存器

MIPS指令中寄存器数据和存储器数据的指定

❖寄存器数据指定：

- 31 x 32-bit GPRs （r0 = 0）
- 寄存器编号占5 bit
- 32 x 32-bit FP regs （f0 ～ f31, paired DP）
- HI, LO, PC: 特殊寄存器
- 寄存器功能和2种汇编表示方式



❖存储器数据指定

- 32- bit machine --> 可访问空间: 2^{32}bytes
- Big Endian(大端方式)
- 只能通过Load/Store指令访问存储器数据
- 数据地址通过一个32位寄存器内容加16位偏移量得到
- 16位偏移量是带符号整数，符号扩展
- 数据要求按边界对齐

SKIP

MIPS寄存器的功能定义和两种汇编表示

Name	number	Usage	Reserved on call?
zero	0	constant value =0(恒为0)	n.a.
at	1	reserved for assembler(为汇编程序保留)	n.a.
v0 ～ v1	2 ～ 3	values for results(过程调用返回值)	no
a0 ～ a3	4 ～ 7	Arguments(过程调用参数)	yes
t0 ～ t7	8 ～ 15	Temporaries(临时变量)	no
s0 ～ s7	16 ～ 23	Saved(保存)	yes
t8 ～ t9	24 ～ 25	more temporaries(其他临时变量)	no
k0 ～ k1	26 ～ 27	reserved for kernel(为OS保留)	n.a.
gp	28	global pointer(全局指针)	yes
sp	29	stack pointer (栈指针)	yes
fp	30	frame pointer (帧指针)	yes
ra	31	return address (过程调用返回地址)	yes

Registers are referenced either by number—$0, … $31, or by name —$t0, $s1… $ra.

zero	at	v0-v1	a0 - a3	t0 - t7	s0 - s7	t8 - t9	k0 - k1	gp	sp	fp	ra
0	1	2 - 3	4 - 7	8 --- 15	16 --- 23	24 - 25	26 - 27	28	29	30	31

BACK to Assemble BACK to Procedure BACK to last

MIPS arithmetic and logic instructions

Instruction	*Example*	*Meaning*	*Comments*
add	**add $1,$2,$3**	**$1 = $2 + $3**	**3 operands; exception possible**
subtract	**sub $1,$2,$3**	**$1 = $2 – $3**	**3 operands; exception possible**
add immediate	**addi $1,$2,100**	**$1 = $2 + 100**	**+ constant; exception possible**
multiply	**mult $2,$3**	**Hi, Lo = $2×$3**	**64-bit signed product**
divide	**div $2,$3**	**Lo = $2 ÷ $3,** **Hi = $2 mod $3**	**Lo = quotient, Hi = remainder**
Move from Hi	**mfhi $1**	**$1=Hi**	**get a copy of Hi**
Move from Lo	**mflo $1**	**$1=lo**	

Instruction	*Example*	*Meaning*	*Comment*
and	and $1,$2,$3	$1 = $2 & $3	Logical AND
or	or $1,$2,$3	$1 = $2 | $3	Logical OR
xor	xor $1,$2,$3	$1 = $2 ⊕ $3	Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	Logical NOR

这里没有全部列出，还有其他指令，如addu（不带溢出处理）， addui 等

问题：x86没有分add还是addu，会不会有问题？

不会。x86只产生各种标志，由软件根据标志信息来判断是否溢出。

Example：算术运算

E.g. f= (g+h) - (i+j),
assuming f, g, h, i, j be assigned to $1, $2, $3, $4, $5

```
add $7, $2, $3
add $8, $4, $5
sub $1, $7, $8
```

寄存器资源由编译器分配！
简单变量尽量被分配在寄存器中，为什么？
程序中的常数如何处理呢？

E.g. f= (g+100) - (i+50) ⟶

```
addi $7, $2, 100
addi  $8, $4, 50
sub $1, $7, $8
```

```
addi $7, $2, 65000
addi  $8, $4, 50
sub $1, $7, $8
```

⟵ E.g. f= (g+65000) - (i+50)

问题：以下程序如何处理呢？

指令设计时必须考虑这种情况！MIPS有一条专门指令，后面介绍。

MIPS data transfer instructions

Instruction	*Comment*	*Meaning*
SW $3, 500($4)	Store word	$3 →($4+ 500)
SH $3, 502($2)	Store half	Low Half of $3 →($2+ 502)
SB $2, 41($3)	Store byte	LQ of $2 →($3+ 41)
LW $1, -30($2)	Load word	($2 -30) → $1
LH $1, 40($3)	Load half	($3+40) → LH of $1
LB $1, 40($3)	Load byte	($3+40) → LQ of $1

操作数长度的不同由不同的操作码指定。

问题：为什么指令必须支持不同长度的操作数？

高级语言中的数据类型有char，short，int，long,……等，故需要存取不同长度的操作数；操作数长度和指令长度没有关系

Example (Base register)

Assume A is an array of 100 words, and compiler has associated the variables g and h with the register $1 and $2. Assume the base address of the array is in $3. Translate

g = h + A[8]

有没有问题？

~~lw $4, 8($3)~~ ;$4 <-- A[8]

~~add $1, $2, $4~~

offset or displacement
（偏移量）

lw $4, 32($3)

add $1, $2, $4

sw $1, 48($3)

base register
（基址寄存器）

A[12] = h+A[8]

问题：如果在一个循环体内执行：g = h + A[i]，则能否用基址寻址方式？

不行，因为循环体内指令不能变，故首地址A不变，只能把下标 i 放在变址寄存器中，每循环一次下标加1，所以，不能用基址方式而应该用变址方式。

Example (Index Register)

Assume A is an array of 100 words, and compiler has associated the variables g and i with the register $1, $5. Assume the base address of the array is in $3. Translate

g = g + A[i]

```
addi  $6, $0, 4          ; $6 = 4
mult  $5,$6              ; Hi,Lo = i*4
mflo  $7                 ; $6 = i*4, assuming i is small

add $4, $3, $7           ; $4 <-- address of  A[i]
lw  $4, 0($4)
add $1, $1, $4
addi $5, $5, 1
```

Index Register
（变址寄存器）

Why should index i multiply 4 ?

How do speedup i multiply 4 ?

Index mode suitable for Array!

问题：若循环执行 g=g+A[i]，怎样使上述循环体内的指令条数减少？

用$5做变址器，每次$5加4 或 用移位指令，而不用乘法指令

若增设专门的“变址自增（即自动变址）”指令则可使循环更短

MIPS的call/return/ jump/branch和compare指令

Instruction	Example	Meaning
jump register	jr $31	go to $31
	For switch, procedure return	
jump and link	jal 10000	$31 = PC + 4; go to 10000
	For procedure call	
~~jump~~	~~j 10000~~	~~go to 10000~~
	Jump to target address	

call / return

伪指令：硬件不能直接执行

Pseudoinstruction blt, ble, bgt, bge

not implemented by hardware, but synthesized by assembler

set on less than	slt $1,$2,$3	if ($2 < $3) $1=1; else $1=0
set less than imm.	slti $1,$2,100	if ($2 < 100) $1=1; else $1=0
branch on equal	beq $1,$2,100	if ($1 == $2) go to PC+4+100
branch on not eq.	bne $1,$2,100	if ($1!= $2) go to PC+4+100

按补码比较大小

问题：指令中立即数是多少？ 100=0064H

问题：指令中立即数是多少？ 25=0019H

汇编中给出的是立即数符号扩展后乘4得到的值

而分支指令中给出的是相对于当前指令的指令条数！

BACK to Procedure

Example：if-then-else语句和“=”判断

```
if (i = = j)
        f = g+h ;
else
        f = g-h ;
```

Assuming variables i, j, f, g, h, ~ $1, $2, $3, $4, $5

```
            bne $1, $2, else        ; i!=j, jump to else
            add $3, $4, $5
            j   exit                ; jump to exit
else:       sub $3, $4, $5
exit:
```

Example：“less than”判断

if (a < b) f = g+h ; else f = g-h ;
Assuming variables a, b, f, g, h, ~ $1, $2, $3, $4, $5

```
        slt   $6, $1, $2          ; if a<b, $6=1, else $6=0
        bne $6, $zero, else       ; $6!=0, jump to else
        add $3, $4, $5
        j   exit                  ; jump to exit
else:   sub $3, $4, $5
exit:
```

```
        slt   $6, $1, $2          ; if a<b, $6=1, else $6=0
        beq $6, $zero, else       ; $6=0, jump to else
        add $3, $4, $5
        j   exit                  ; jump to exit
else:   sub $3, $4, $5
exit:
```

Example：Loop循环

Loop: g = g +A[i];
i = i+ j;
if (i != h) go to Loop:

Assuming variables g, h, i, j ~ $1, $2, $3, $4 and base address of array is in $5

```
Loop: add $7, $3, $3      ; i*2
      add $7, $7, $7      ; i*4
      add $7, $7, $5
      lw $6, 0($7)        ; $6=A[i]
      add $1, $1, $6      ; g= g+A[i]
      add $3, $3, $4
      bne $3, $2, Loop
```

加法比乘法快！

也可用移位来实现乘法！

$3中是i，$7中是i*4

编译器和汇编语言程序员不必计算分支指令的地址，而只要用标号即可！汇编器完成地址计算

Example：过程调用

```
int i;
void set_array(int num)
{
        int array[10];
        for (i = 0; i  < 10; i ++) {
          arrar[i] = compare (num, i);
      }
}

int compare (int a, int b)
{
       if ( sub (a, b) >= 0)
            return 1;
        else
            return 0;
}

int sub (int a, int b)
{
    return a-b;
}
```

i是全局静态变量

array数组是局部变量

set_array是调用过程
compare是被调用过程

compare是调用过程
sub是被调用过程

问题：过程调用对应的机器代码如何表示？

1. 如何从调用程序把参数传递到被调用程序？
2. 如何从调用程序执行转移到被调用程序执行？
3. 如何从被调用程序返回到调用程序执行？
4. 如何保证调用程序中寄存器内容不被破坏？

Procedure Call and Stack(过程调用和栈)

- **过程调用的执行步骤（假定过程P调用过程Q）：**

compare (num, i);

  - 将参数放到Q能访问到的地方
  - 将P中的返回地址存到特定的地方，将控制转移到过程Q

在调用过程P中完成

  - 为Q的局部变量分配空间（局部变量临时保存在栈中）
  - 执行过程Q
  - 将Q执行的返回结果放到P能访问到的地方
  - 取出返回地址，将控制转移到P，即返回到P中执行

在被调用过程Q中完成

- **MIPS中用于过程调用的指令（见MIPS过程调用指令）**
- **MIPS规定少量过程调用信息用寄存器传递（见MIPS寄存器功能定义）**
- **如果过程中用到的参数超过4个，返回值超过2个，怎么办？**
  - 更多的参数和返回值要保存到存储器的特殊区域中
  - 这个特殊区域为：栈(Stack)

一般用“栈”来传递参数、保存返回地址、临时存放过程的局部变量等。为什么？

便于递归调用！

栈(Stack)的概念

- **栈的基本概念**
  - 是一个“先进后出”队列
  - 需一个栈指针指向栈顶元素
  - 每个元素长度一致
  - 用“入栈”（push）和“出栈”（pop）操作访问栈元素
- **MIPS中栈的实现**
  - 用栈指针寄存器$sp来指示栈顶元素
  - 每个元素的长度为32位，即：一个字(4个字节)
  - “入栈” 和“出栈” 操作用 sw / lw 指令来实现，需用add / sub指令调整$sp的值，不能像x86那样自动进行栈指针的调整

（有些处理器有专门的push/pop指令，能自动调整栈指针。如x86）

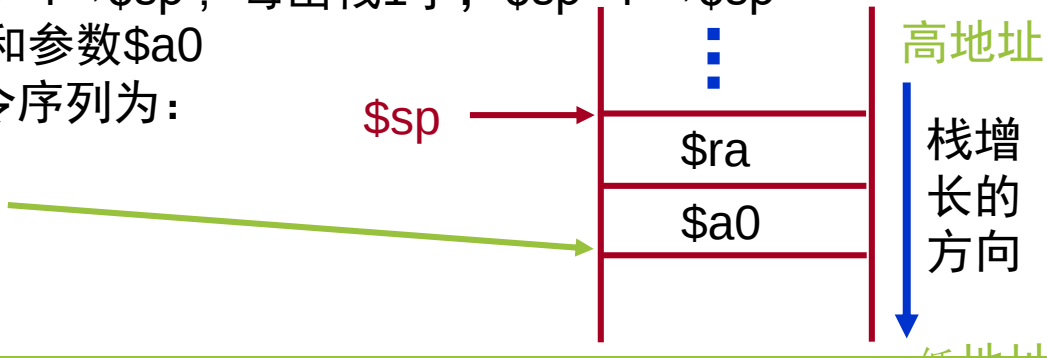
  - 栈生长方向

    从高→低地址“增长”，而取数/存数的方向是低→高地址（大端方式）

    - 每入栈1字，$sp- 4→$sp ；每出栈1字，$sp+4 →$sp

例：若将返回地址$ra和参数$a0保存到栈，则指令序列为：

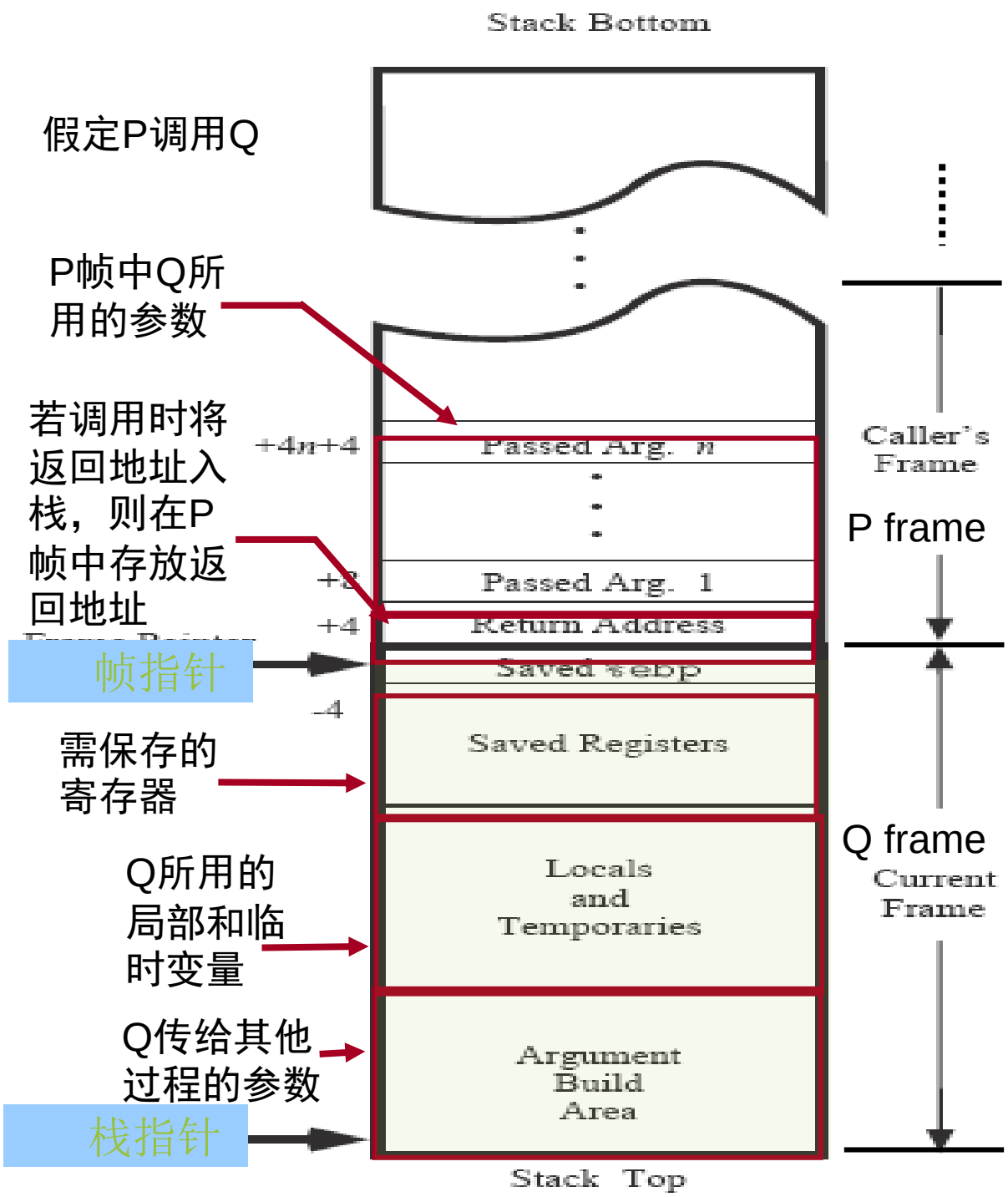
```
sub  $sp, $sp,  8
sw   $ra, 4($sp)
sw   $a0, 0($sp)
```



栈帧的概念

- 各过程有自己的栈区，称为栈帧（Stack frame），即过程的帧（procedure frame）
- 栈由若干栈帧组成
- 用专门的帧指针寄存器指定起始位置
- 当前栈帧范围在帧指针和栈指针之间
- 程序执行时，栈指针可移动，帧指针不变。所以过程内对栈信息的访问大多通过帧指针进行

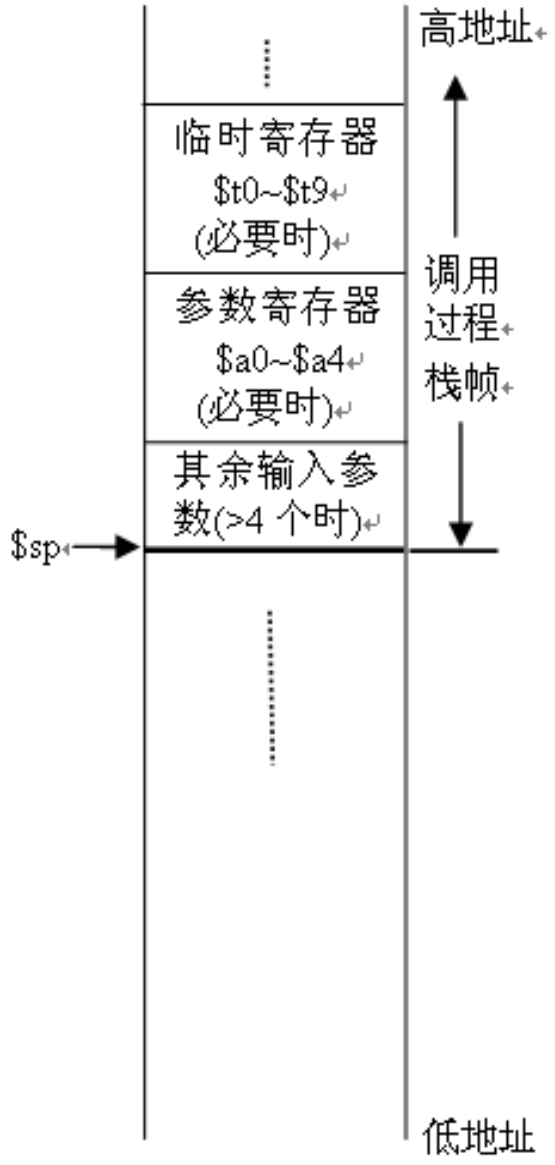
X86的栈帧情况如右图所示



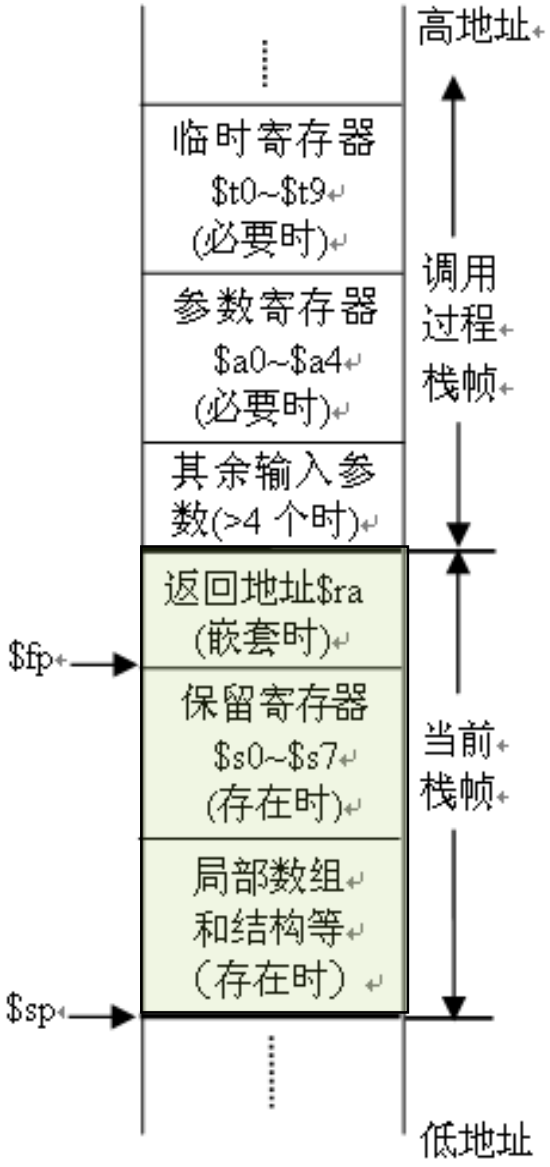
MIPS中的过程调用（假定P调用Q）

- 程序可访问的寄存器组是所有过程共享的资源，给定时刻只能被一个过程使用，因此过程中使用的寄存器的值不能被另一个过程覆盖！
- MIPS的寄存器使用约定：
  - 保存寄存器$s0 ~$s7 的值在从被调用过程返回后还要被用，被调用者需要保留
  - 临时寄存器$t0 ~$t9的值在从被调用过程返回后不需要被用（需要的话，由调用者保存），被调用者可以随意使用
  - 参数寄存器$a0~$a3在从被调用过程返回后不需要被用（需要的话，由调用者保存在栈帧或其他寄存器中），被调用者可以随意使用
  - 全局指针寄存器$gp的值不变
  - 在过程调用时帧指针寄存器$fp用栈指针寄存器$sp- 4来初始化
- 需在被调用过程Q中入栈保存的寄存器（称为被调用者保存）
  - 返回地址$ra（如果Q又调用R，则$ra内容会被破坏，故需保存)
  - 保存寄存器$s0 ~$s7 (Q返后P可能还会用到，Q中用的话就被破坏，故需保存)
- 除了上述寄存器以外，所有局部数组和结构类型变量也要入栈保存
- 如果局部变量和临时变量发生寄存器溢出（寄存器不够分配），则也要入栈
- 每个处理器对栈帧规定的“调用者保存”和“被调用者保存”的寄存器可能不同。例：
  - x86中返回地址保存在调用过程栈帧中；而MIPS则在被调用过程栈帧中保存
  - x86中调用参数保存在调用过程栈帧中；而MIPS则在被调用过程栈帧中保存
  - X86中调用过程的帧指针保存在被调用过程栈帧中；MIPS也一样。

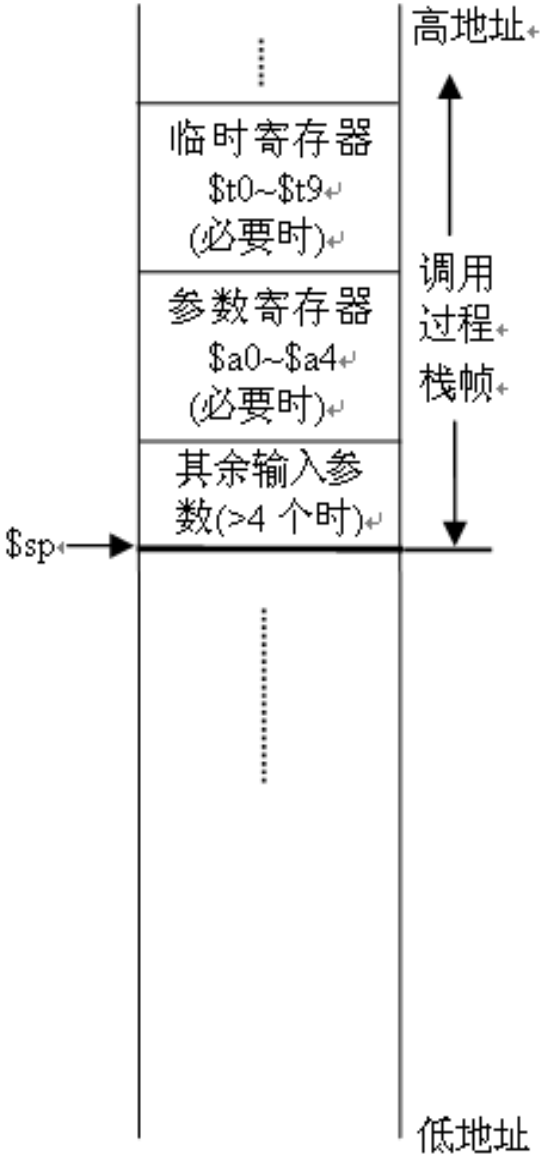
过程调用时MIPS中的栈和栈帧的变化



(a) 过程调用前



(b) 过程调用中



(c) 过程调用后

Example in C: swap

假定swap作为一个过程被调用，temp对应$t0, 变量v 和 k分别对应$s0和$s1

写出对应的MIPS汇编代码。 问题：上述假设有何问题？ 参数v和k应该在$a0和$a1

```
swap(int v[ ], int k)
{
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

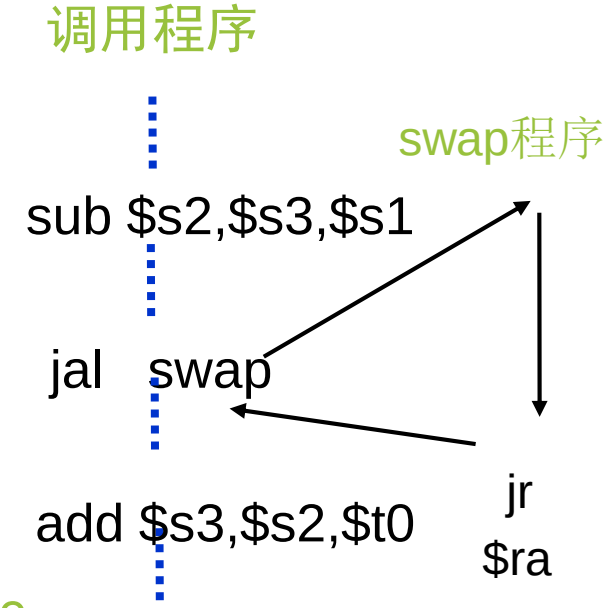
```
sll     $s2, $a1, 2      ; mulitply k by 4
addu    $s2 $s2, $a0     ; address of v[k]
lw      $t0, 0($s2)      ; load v[k]
lw      $s3, 4($s2)      ; load v[k+1]
sw      $s3, 0($s2)      ; store v[k+1] into v[k]
sw      $t0, 4($s2)      ; store old v[k] into v[k+1]
```

在调用过程中用指令 “jal swap” 进行swap调用

jal --- jump and link (跳转并链接)

$31 = PC+4 ; $31=$ra

goto swap



问题1：若swap中不保存$s2，则会发生什么情况？

caller中$s2的值被破坏！须在swap中保存$s2

问题2：若swap中不保存$t0，则会发生什么情况？

$t0约定由caller保存，故无须在swap栈帧中保存$t0

swap: MIPS中的一个过程示例

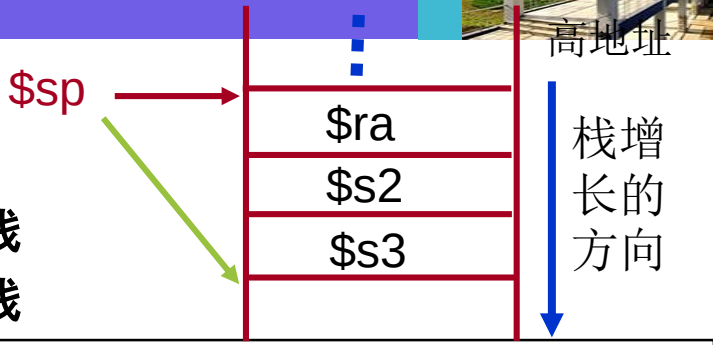
```
swap:
    addi   $sp,$sp, –12      ; 栈增长3个
    sw     $31, 8($sp)       ; 返回地址入栈
    sw     $s2, 4($sp)       ; 保留寄存器$s2入栈
    sw     $s3, 0($sp)       ; 保留寄存器$s3入栈

    ....

    lw     $s3, 0($sp)       ; 恢复$s3
    lw     $s2, 4($sp)       ; 恢复$s2
    lw     $31, 8($sp)       ; 恢复$31（$ra）
    addi   $sp,$sp, 12       ; 退栈

    jr     $31               ; 从swap返回到调用过程
```

```
sll     $s2, $a1, 2      ; mulitply k by 4
addu    $s2, $s2, $a0    ; address of v[k]
lw      $t0, 0($s2)      ; load v[k]
lw      $s3, 4($s2)      ; load v[k+1]
sw      $s3, 0($s2)      ; store v[k+1] into v[k]
sw      $t0, 4($s2)      ; store old v[k] into v[k+1]
```



问题：是否一定要将返回地址（$31）保存到栈帧中？

如果swap是叶子过程，则无需保存返回地址到栈中，为什么？

如果将所有内部寄存器都用临时寄存器 (如$t1等)，则叶子过程swap的栈帧为空，且上述黑色指令都可去掉

$ra的内容不会被破坏！

Example：过程调用

```
int i;
void set_array(int num)
{
        int array[10];
        for (i = 0; i  < 10; i ++) {
           arrar[i] = compare (num, i);
        }
}

int compare (int a, int b)
{
        if ( sub (a, b) >= 0)
            return 1;
        else
            return 0;
}

int sub (int a, int b)
{
    return a-b;
}
```

i是全局静态变量

array数组是局部变量

set_array是调用过程
compare是被调用过程

compare是调用过程
sub是被调用过程

问题1：编译器如何为全局变量和局部变量分配空间？

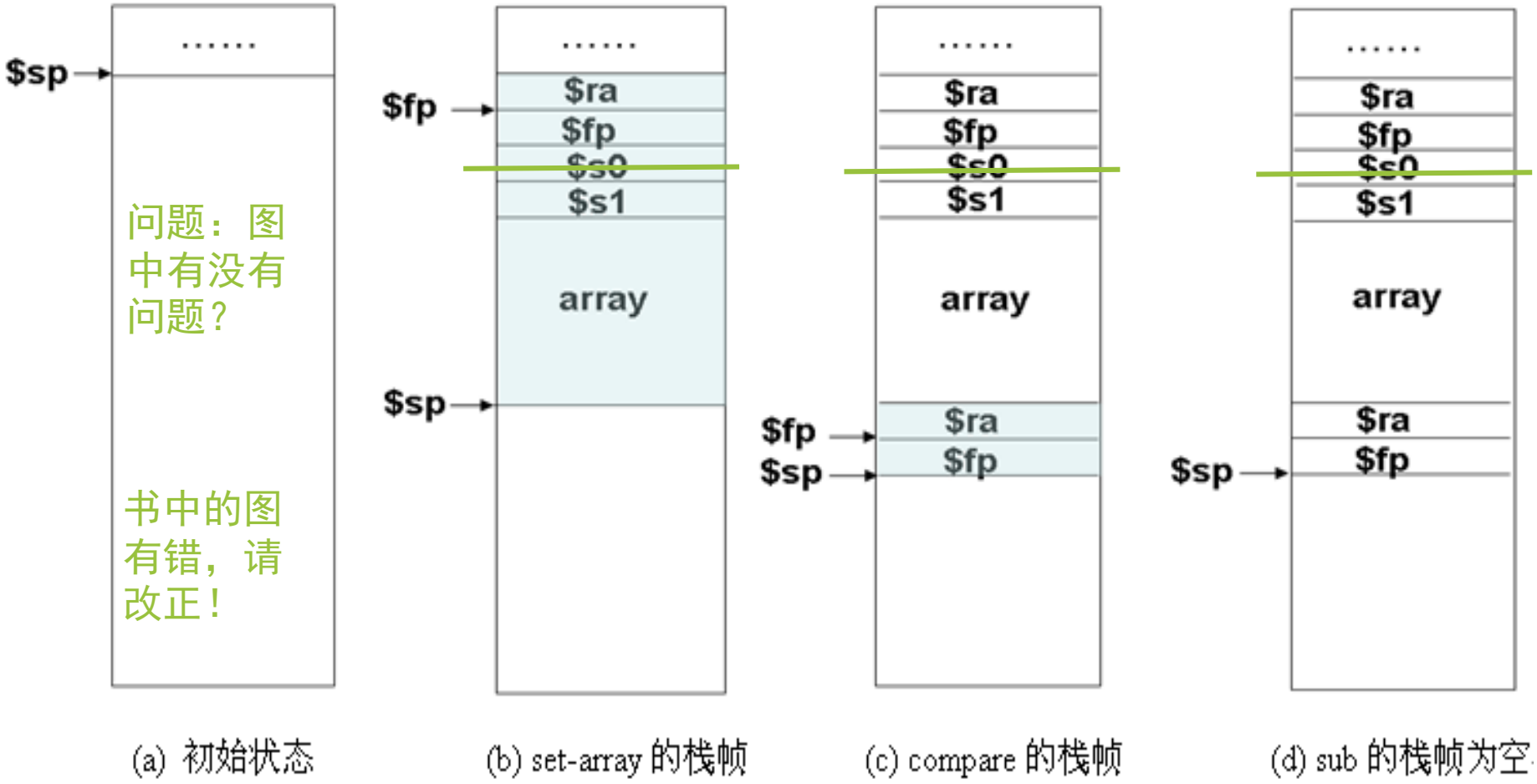
问题2：执行set_array的结果是什么？

过程调用时的变量分配

- 全局静态变量一般分配到寄存器或在R/W存储区（数组或结构等）。

  该例中只有一个简单变量i，假定分配给$s0。无需保存和恢复！
- 为减少指令条数，并减少访问内存次数。在每个过程的过程体中总是先使用临时寄存器$t0~$t9，临时寄存器不够或者某个值在调用过程返回后还需要用，就使用保存寄存器$s0~$s7。
- 过程set_array的入口参数为num，没有返回参数，有一个局部数组，被调用过程为compare，因此，其栈帧中除了保留所用的保存寄存器外，必须保留返回地址（是否保存$fp要看具体情况，如果确保后面都不用到$fp，则可以不保存，但为了保证$fp的值不被后面的过程覆盖，通常情况下，应该保存$fp的值），并给局部数组预留4×10=40个字节的空间。
- 从过程体来看，从compare返回后还需要用到数组基地址，故将其分配给$s1。因此要用到的保存寄存器有两个：$s0和$s1，但只需要保存$s1。另外加上返回地址$ra、帧指针$fp、局部数组，其栈帧空间最少为3×4+40=52B。

过程调用举例：栈指针、帧指针和栈帧中信息的变化



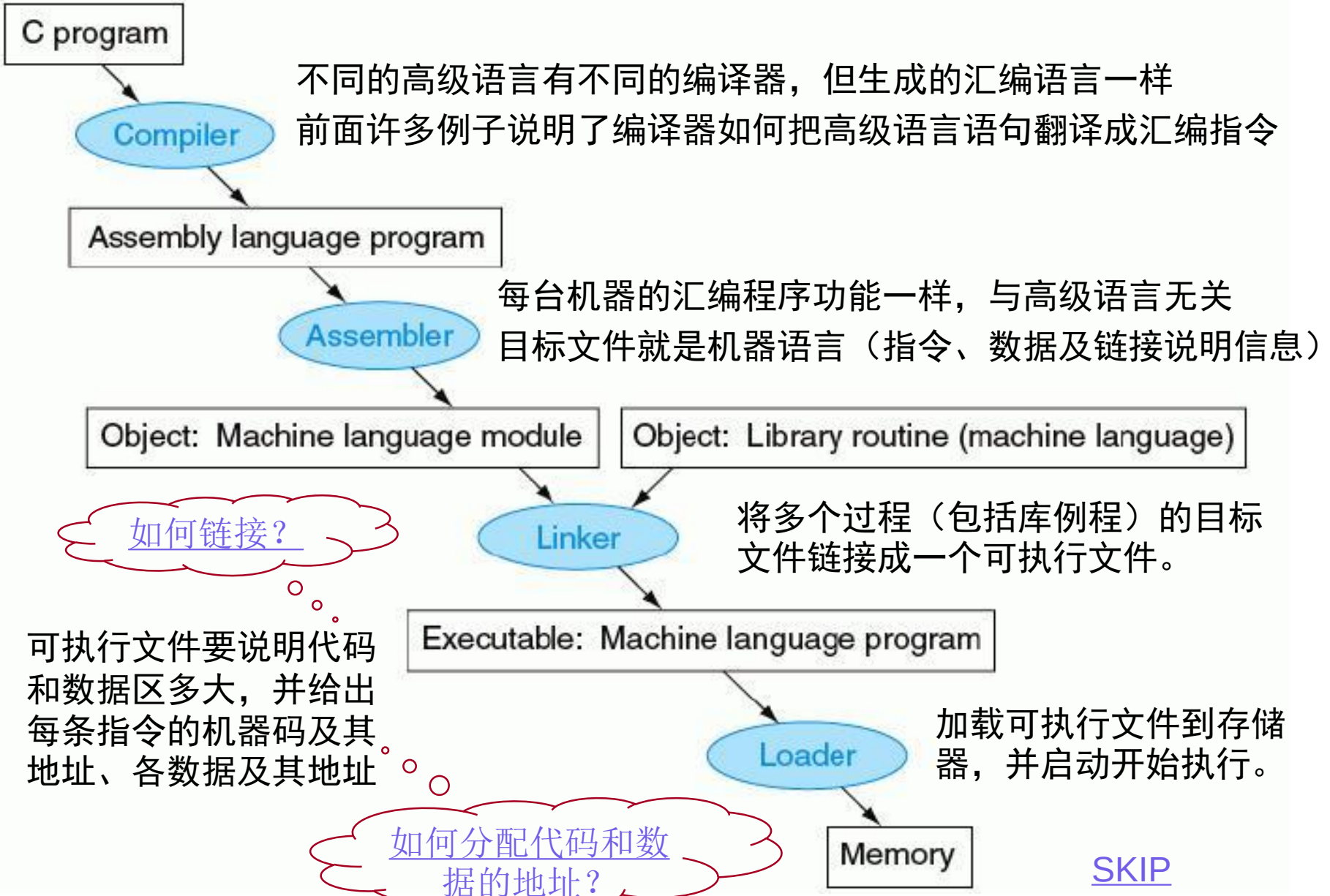
过程set-array没有多于4个额外入口参数，无需保留临时寄存器，故栈帧占52字节。

过程compare入口参数为a和b，仅一个返回参数，没有局部变量，被调用过程为sub。

过程体中没用到保存寄存器，所以，其栈帧中只需保留返回地址$ra和$fp的值。

过程sub是叶子过程，其栈帧为空。

程序的翻译、链接和加载（自学）



不同的高级语言有不同的编译器，但生成的汇编语言一样

前面许多例子说明了编译器如何把高级语言语句翻译成汇编指令

每台机器的汇编程序功能一样，与高级语言无关

目标文件就是机器语言（指令、数据及链接说明信息）

如何链接？

将多个过程（包括库例程）的目标文件链接成一个可执行文件。

可执行文件要说明代码和数据区多大，并给出每条指令的机器码及其地址、各数据及其地址

加载可执行文件到存储器，并启动开始执行。

如何分配代码和数据的地址？

SKIP

复习：MIPS程序和数据的存储器分配

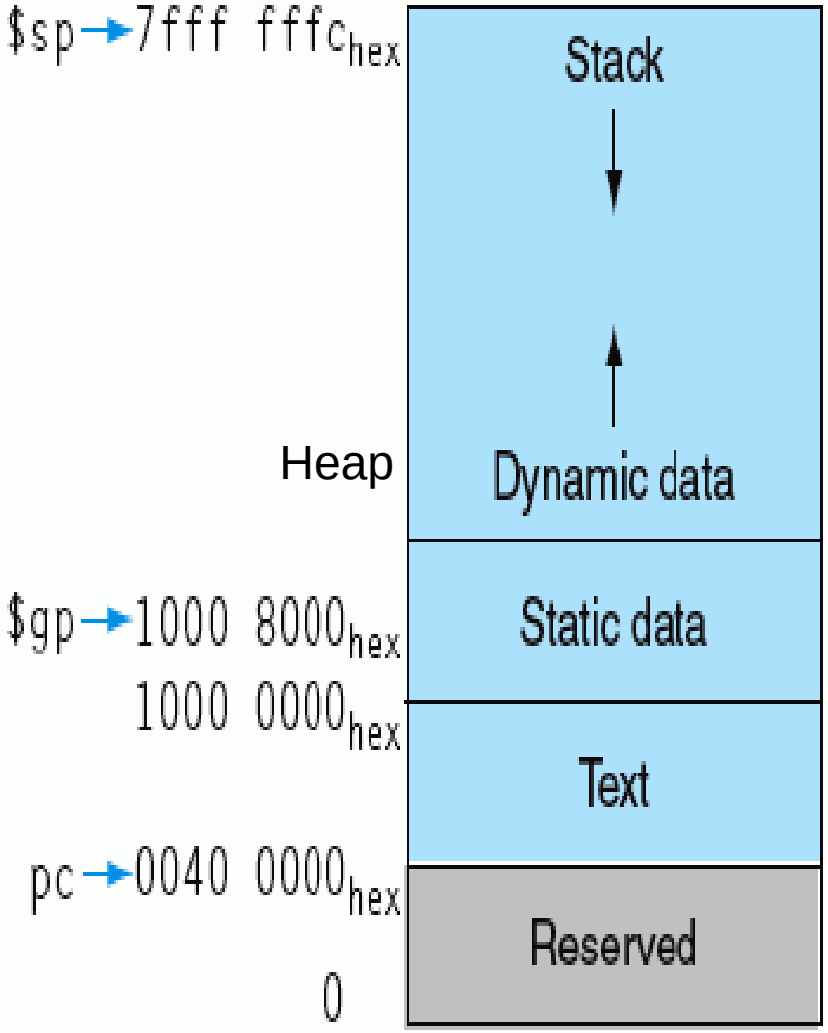
- 每个MIPS程序都按如下规定进行存储器分配
- 每个可执行文件都按如下规定给出代码和数据的地址

栈区位于堆栈高端，堆区位于堆栈低端

静态数据区存放全局变量（也称静态变量），指所有过程之外声明的变量和用Static声明的变量；从固定的0x1000 0000处开始向高地址存放

全局指针$gp总是0x1000 8000，其16位偏移量的访问范围为0x1000 0000～0x1000 ffff，可遍及整个静态数据区的访问

程序代码从固定的0x0040 0000处开始存放
故PC的初始值为0x0040 0000



BACK

目标文件的链接（自学）

过程A和过程B分别编译、汇编成目标文件，链接后

Object file header	过程A的目标文件		
	Name	Procedure A	
	Text size	100_{hex}	代码的长度为0x100
	Data size	20_{hex}	数据的长度为0x20
Text segment	Address	Instruction	
链接前地址总是从0开始 实际是指令机器码	0	lw $a0, 0($gp)	0是由x待定的地址
	4	jal 0	0是由B待定的地址
	…	…	
Data segment 链接前地址总是从0开始	0	(X)	
	…	…	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	—	X的地址待定
	B	—	B的地址待定

目标文件的链接（自学）

过程A和过程B分别编译、汇编成目标文件，链接后生成一个可执行文件

Object file header	过程B的目标文件		
	Name	Procedure B	
	Text size	200_{hex}	代码的长度为0x200
	Data size	30_{hex}	数据的长度为0x30
Text segment	Address	Instruction	
	0	sw $a1, 0($gp)	0是由Y待定的地址
	4	jal 0	0是由A待定的地址
	…	…	
Data segment	0	(Y)	
	…	…	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	—	
	A	—	

目标文件的链接（自学）

过程A和过程B分别编译、汇编成目标文件，链接后生成一个可执行文件

Executable file header	生成的可执行文件	
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
代码地址总是从0040 0000开始	$0040\ 0000_{hex}$	lw $a0, 8000_{hex}($gp)
	$0040\ 0004_{hex}$	jal $40\ 0100_{hex}$
	…	…
过程B从A后的0x100开始	$0040\ 0100_{hex}$	sw $a1, 8020_{hex}($gp)
	$0040\ 0104_{hex}$	jal $40\ 0000_{hex}$
	…	…
Data segment	Address	1000 0000=?+ 1000 8000
静态区地址从0x1000 0000开始	$1000\ 0000_{hex}$	(X)
	…	…
过程B从A后的0x20开始	$1000\ 0020_{hex}$	(Y)
	…	…

1000 8000+ FFFF 8000=1000 0000H → ?= 8000 (符号扩展后为FFFF 8000)

MIPS指令中位的指定和逻辑运算（自学）

逻辑数据表示

- 用一位表示 真：1 -True / 假：0-False
- N位二进制数可表示N个逻辑数据

逻辑运算

- 按位进行， 如: And / Or / Shift Left / Shift Right等

位的指定

- 设置某位的值：
  - 清0: 与掩码（1…101…1）相“与”
  - 置1：与位串（0…010…0）相“或”
- 判断某位的值：
  - 是否为0: 与位串（0…010…0）相“与”后，是否为0
  - 是否为1：与位串（ 0…010…0 ）相“与”后，是否不为0

MIPS中的移位指令（sll / srl）

例 : srl $t2,$s0,8

$s0右移8位后送$t2

op	rs	rt	rd	shamt	func
000000	00000	10000	01010	01000	000010

逻辑数据和数值数据在形式上并无差别，也是一串0/1序列，机器本身不能识别，需靠指令的类型来识别。包括后面所讲的字符数据等都一样。

MIPS指令中常数的指定（自学）

- **程序中经常需要使用常数，例如：**
  - C编译器gcc中52%的算术指令用到常数
  - 电路模拟程序Spice中69%的算术指令用到常数
- **指令中如何取得常数**
  - 若程序装入时，常数已在内存中，则需用load指令读入寄存器
  - 在指令中用一个“立即数”来使用常数

例1：i=i+4;　Assuming variable i ~ $1

则：　addi $1, $1, 4

例2：if (i<20) …. ;　Assuming variable i ~ $1

则：　slti $3, $1, 20　　; if (i<20) $3=1 else $3=0

如果常数的值用16位无法表示，怎么办？

用lui指令把高16位送到寄存器的高16位，再把低16位加到该寄存器中。

例3：将“0000 0000 0011 1101 0000 0000 0000 1000”送$3中

则：　lui $3, 61
　　addi $3, $3, 8

MIPS指令中如何表示文本字符串（自学）

❖ 有些情况下，程序需要处理文本。例如：

- 西文文本由ASCII码字符构成字符串
- Java等语言使用Unicode编码构成字符串
- 汉字文本使用的汉字编码字符构成字符串

❖ 字符串的表示

- 由一个个字符组成，长度不定。有三种表示方式：
  - 字符串的首字节记录长度
  - 用其他变量来记录长度（即：用“struc”类型来描述）
  - 字符串末尾用一个特殊字符表示。
    如：C语言用字符（NULL）来标记字符串结束

❖ 如何在指令中表示字符

- ASCII字符串，每个字符由8位组成，用 “lb/sb”指令存/取一个字节
- Unicode及汉字字符都有16位，用 “lh/sh”指令存/取两个字节

例：x[i] = y[j]; variables i,j ~ $1,$2, base address x,y ~ $3,$4

```
则：    add  $5, $3, $1          ; $5=the address of x[i]
        add  $6, $4, $2          ; $6=the address of y[j]
        lb   $7, 0($6)           ; $7=y[j]
        sb   $7, 0($5)           ; x[i]=$7
```

SKIP

数据类型和MIPS指令的对应（自学）

C type	Java type	Data transfers	Operations
int	int	lw, sw, lui	addu, addiu, subu, mult, div, and, andi, or, ori, nor, slt, slti
unsigned int	—	lw, sw, lui	addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu
char	—	lb, sb, lui	addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu
—	char	lh, sh, lui	addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu
float	float	lwc1, swc1	add.s, sub.s, mult.s, div.s, c.eq.s, c.lt.s, c.le.s
double	double	l.d, s.d	add.d, sub.d, mult.d, div.d, c.eq.d, c.lt.d, c.le.d

C语言中的“char”为8位， Java语言中的“char”为16位(Unicode)