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 { 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) 细粒度 硬件实现 (指令内部)

## 三、指令流水原理



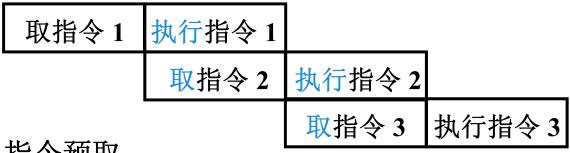
### 1. 指令的串行执行

 取指令1
 取指令2
 执行指令2
 取指令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

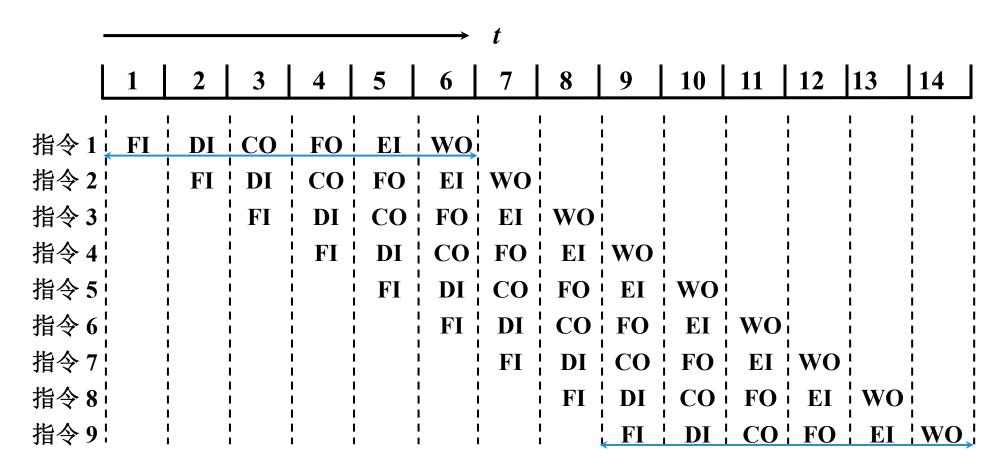
必须等上条 指令执行结束,才能确定下条 指令的地址,

造成时间损失

猜测法

# 4. 指令的六级流水





完成一条指令

串行执行

六级流水

6个时间单位

 $6 \times 9 = 54$  个时间单位

14个时间单位

## 三、影响指令流水线性能的因素

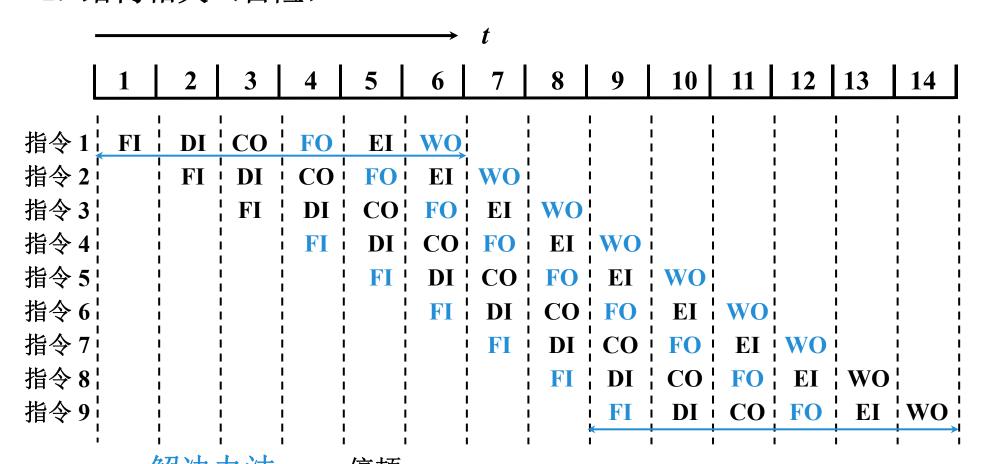


程序的相近指令之间出现某种关联 使指令流水出现停顿,影响流水线效率

### 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



冲突 指令1、指令3、指令6冲突 指令存储器和数据存储器分开

指令2与指令5万

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 
$$R_1$$
,  $R_2$ ,  $R_3$  ;  $(R_2) - (R_3) \rightarrow R_1$ 

ADD 
$$R_4$$
,  $R_5$ ,  $R_1$  ;  $(R_5) + (R_1) \rightarrow R_4$ 

• 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 
$$R_1$$
,  $R_2$  ;  $(R_1) \times (R_2) \longrightarrow (R1)$ 

$$MOV R_2, 0 ; 0 \longrightarrow R_2$$

• 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 
$$R_3$$
,  $R_2$ ,  $R_1$  ;  $(R_2) \times (R_1) \rightarrow R_3$ 

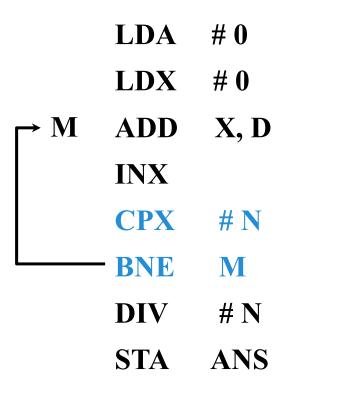
SUB 
$$R_3$$
,  $R_4$ ,  $R_5$  ;  $(R_4) - (R_5) \rightarrow R_3$ 

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



BNE 指令必须等

CPX 指令的结果

才能判断出

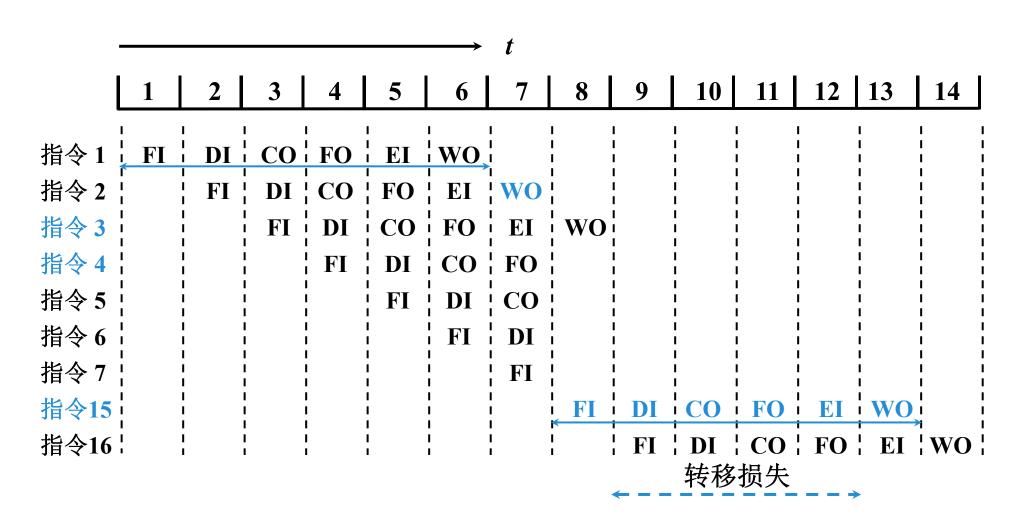
是转移

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 $\Delta t$ 

• 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△t

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

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

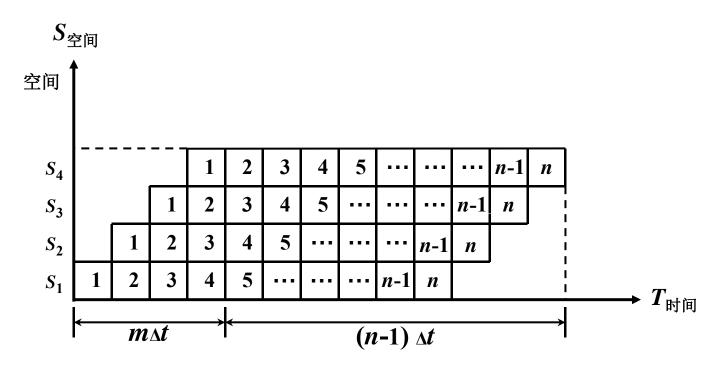
则 
$$S_p = \frac{nm \cdot \Delta t}{m \cdot \Delta t + (n-1) \cdot \Delta t} = \frac{nm}{m+n-1}$$

## 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

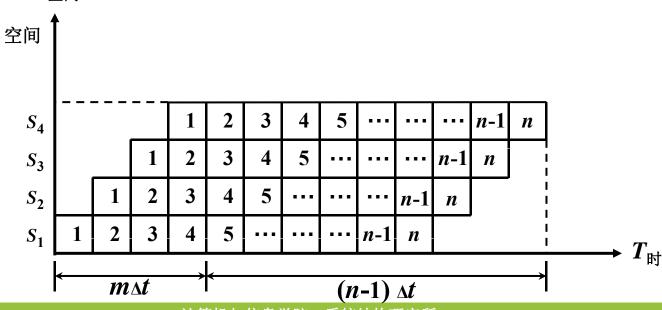


### 3. 效率



#### 流水线中各功能段的 利用率

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



15

### 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)

## 例



- 1. 设某任务需要8个加工部件加工才能完成,每个加工部件加工需要时间为T,现在采用流水线加工方式,要完成100个任务,共需多少时间。
- 2. 分别用时空图法证明流水CPU比非流水CPU具有 更大的吞吐率。
- 解:假设指令周期包含4个过程:取指令(IF)、译码(ID)、取操作数(EX)、运算(WB),每个子过程称为过程段(Si),这样,一个流水线由一系列串连的过程段组成。在统一时钟信号控制下,数据从一个过程段流向相邻的过程段。

课本图8.19 a) b) c)

## 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



\* 有三种指令格式

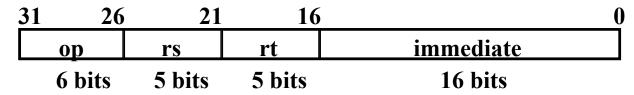
31	<b>26</b>	21	16	11	6	0
op		rs	rt	rd	shamt	func
6 bit	S	5 bits	5 bits	5 bits	5 bits	6 bits

R-Type

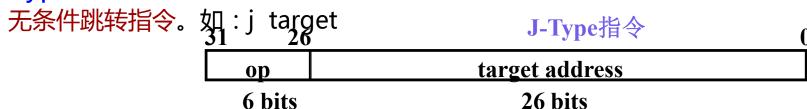
两个操作数和结果都在寄存器的运算指令。如:sub rd, rs, rt

- 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



# MIPS指令字段含义



R-Type指令

OP:操作码

rs:第一个源操

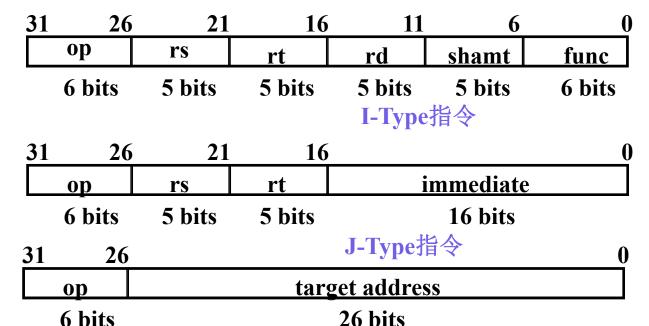
作数寄存器

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)。

# MIPS arithmetic and logic instructions

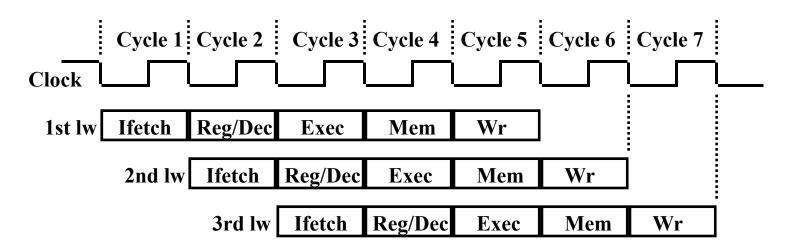


Instruction	汇编举例	含义	<u> </u>					
add	add \$1,\$2,\$3	<b>\$1 = \$2 + \$3</b>	3个寄存器	操作数				
subtract	sub \$1,\$2,\$3	<b>\$1 = \$2 - \$3</b>	3个寄存器	操作数				
load word 寄存器	lw \$ s1,100 ( \$	5 S2)	nory[\$ s2+100]	从内存取一个字到				
store word 寄存器	sw \$ s1,100 ( s	\$ S2) Memory[\$	s2+100] = \$ s1	从内存取一个字到				
multiply	mult \$2,\$3	Hi, Lo = $2\times$ 3	64-bit signed pr	oduct				
divide	div \$2,\$3	$Lo = $2 \div $3,$	Lo = quotient, F	li = remainder				
Hi = \$2 mod \$3								
branch on equal	beq \$ s1,	\$ s2, L if (\$ s1=	=\$ s2) go to L	相等则转移				
branch on not ed	qual bne \$ s1,	\$ s2, L if (\$ s1!	=\$ s2) go to L	不相等则转移				

<b>Instruction</b>	Example	Meaning	Comment
and	and \$1,\$2,\$3	<b>\$1 = \$2 &amp; \$3</b>	<b>Logical AND</b>
or	or \$1,\$2,\$3	<b>\$1 = \$2   \$3</b>	Logical OR
xor	xor \$1,\$2,\$3	$1 = 2 \oplus 3$	<b>Logical XOR</b>
nor	nor \$1,\$2,\$3	\$1 = ~(\$2  \$3)	<b>Logical NOR</b>

# Load指令的流水线

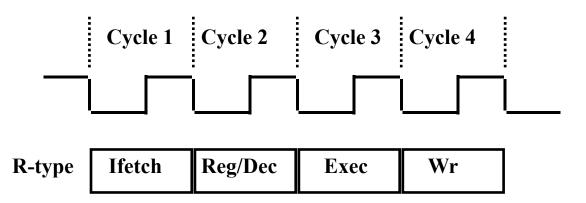




- 每个周期有五个功能部件同时在工作
- 后面指令在前面完成取指后马上开始
- 每个load指令仍然需要五个周期完成
- 但是吞吐率(throughput)提高许多,理想情况下,有:
  - 每个周期有一条指令进入流水线
  - 每个周期都有一条指令完成
  - 每条指令的有效周期(CPI)为1

# R-type指令的4个阶段

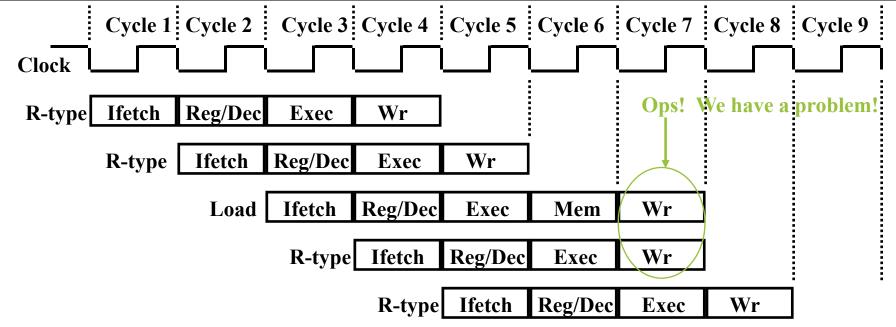




- ❖Ifetch: 取指令并计算PC+4
- ❖Reg/Dec: 从寄存器取数,同时指令在 译码器进行译码
- **❖Exec:** 在ALU中对操作数进行计算
- ❖Wr: ALU计算的结果写到寄存器

## 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

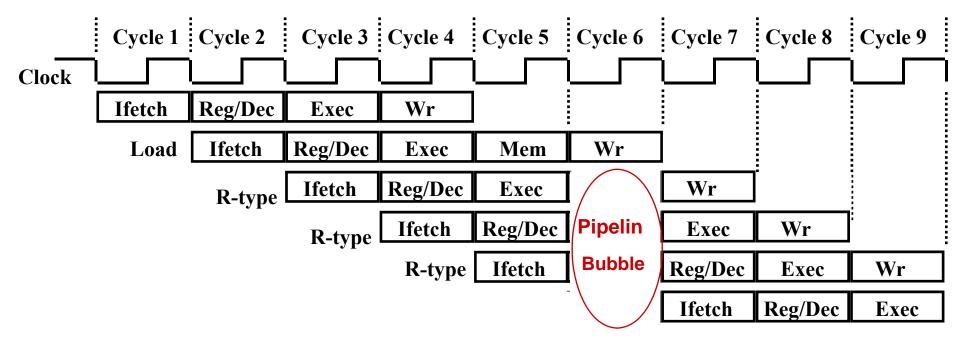
#### 或称为资源冲突!

- ❖ 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!

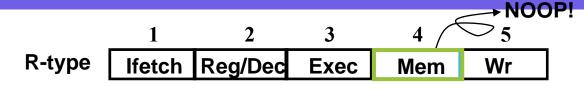
#### 解决方案1:在流水线中插入"Bubble"(气泡)

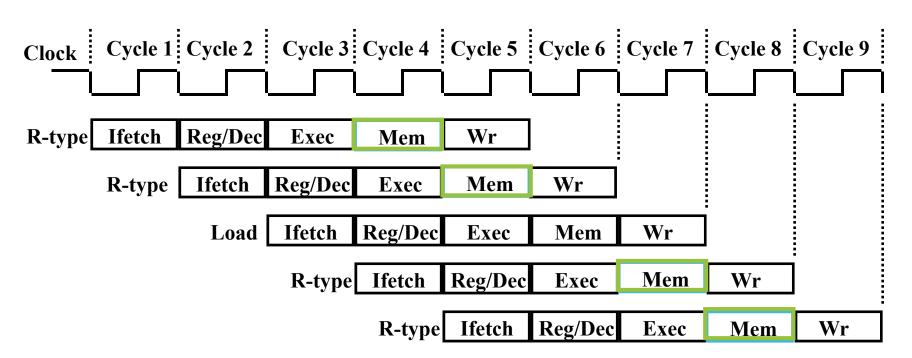




- ❖插入"Bubble"到流水线中,以禁止同一周期有两次写寄存器。缺点:
  - 控制逻辑复杂
  - 第5周期没有指令被完成(CPI不是1,而实际上是2) 方案不可行!

## 解决方案2: R-type的Wr操作延后一个周期执行



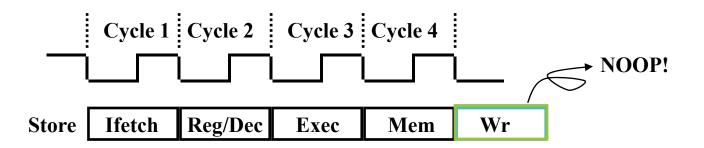


- ❖ 加一个NOP阶段以延迟"写"操作:
  - 把 "写"操作安排在第5阶段,这样使R-Type的Mem阶段为空NOP

这样使流水线中的每条指令都有相同多个阶段!

## Store指令的四个阶段

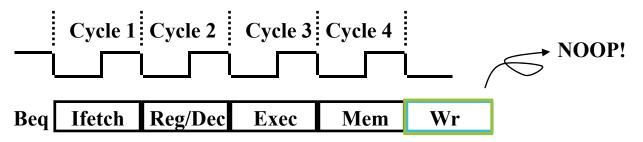




- ❖ Ifetch: 取指令并计算PC+4
- ❖ Reg/Dec: 从寄存器取数,同时指令在译码器进行译码
- ❖ Exec: 16位立即数符号扩展后与寄存器值相加,计算主 存地址
- ❖ Mem:将寄存器读出的数据写到主存
  - 。Wr: 加一个空的写阶段, 使流水线更规整!

## Beq的四个阶段





❖ Ifetch: 取指令并计算PC+4

与多周期通路有什么不同?

- ❖ Reg/Dec:从寄存器取数,同时指令在译码器进行译码
- ❖ Exec: 执行阶段
  - ALU中比较两个寄存器的大小(做减法)
  - Adder中计算转移地址
- ❖ Mem: 如果比较相等,则:
  - 转移目标地址写到PC

Reg/Dec阶段投机进行了 转移地址的计算!可以减 少Branch指令的时钟数

多周期通路中,在

为什么流水线中不进行"投机"计算?

。Wr:加一个空写阶段,使流水线更规整!

按照上述方式,把所有指令都按照最复杂的"load"指令所需的五个阶段来划分,不需要的阶段加一个"NOP"操作

因为,流水线中所有指令的 执行阶段一样多,Branch 指令无需节省时钟,因为有 比它更复杂的指令。

# 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
  - <u>Structural hazards</u> (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
- <u>Data hazards</u> (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):

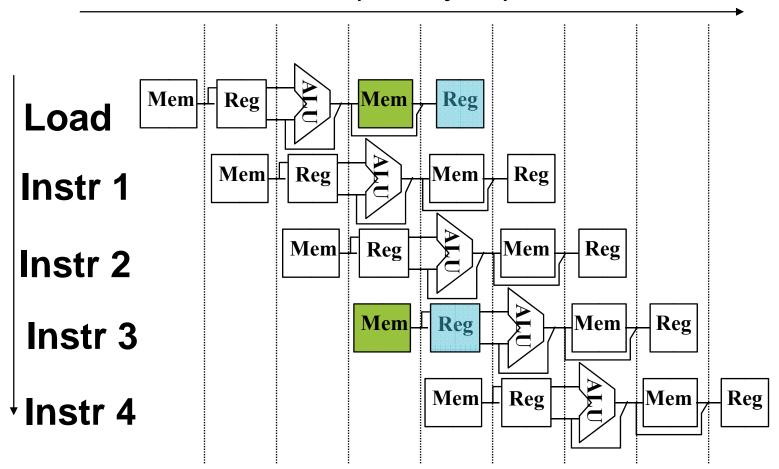
现象:转移或异常改变执行流程,顺序执行指令在目标地址产生前已被取出

- 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



#### Time (clock cycles)



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 Time (clock cycles) Reg Reg Load n Reg Dm Reg Im Instr 1 Reg Reg Dm Im Instr 2 Reg Reg Dm Instr 3 e Dm Reg Reg Instr 4 计算机与信息学院 系统结构研究所

# 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 <u>r1</u> ,r2,r3

sub r4, r1, r3 读r1时, add指令正在执行加法(EXE), 老值!

and r6, r1, r7 读r1时, add指令正在传递加法结果(MEM), 老值!

or r8, <u>r1</u>, r9 读r1时, add指令正在写加法结果到r1(WB), 老值!

xor r10, <u>r1</u> ,r11 读r1时, add指令已经把加法结果写到r1,新值

补充: 三类数据冒险现象

画出流水线图能很清楚理解!

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)

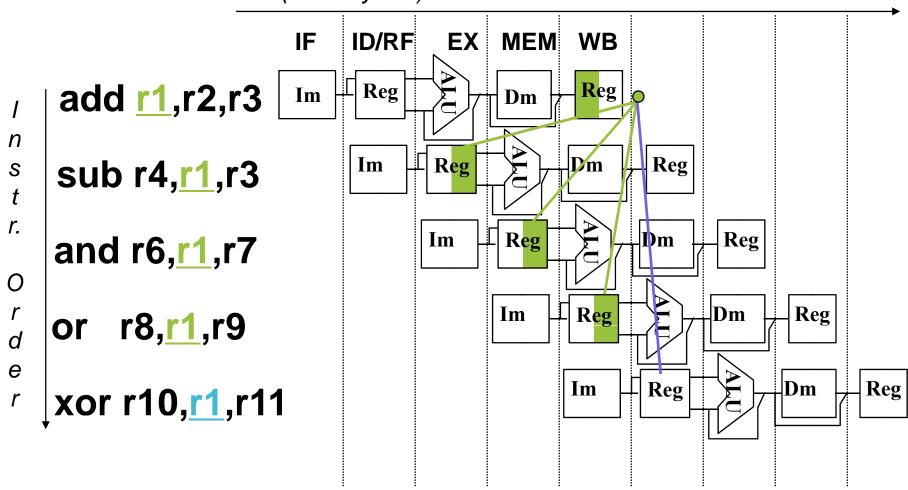
WAW: 写后写(基本流水线中不会发生,多个功能部件时会发生)

本讲介绍基本流水线,所以仅考虑RAW冒险

## Data Hazard on r1



Time (clock cycles)



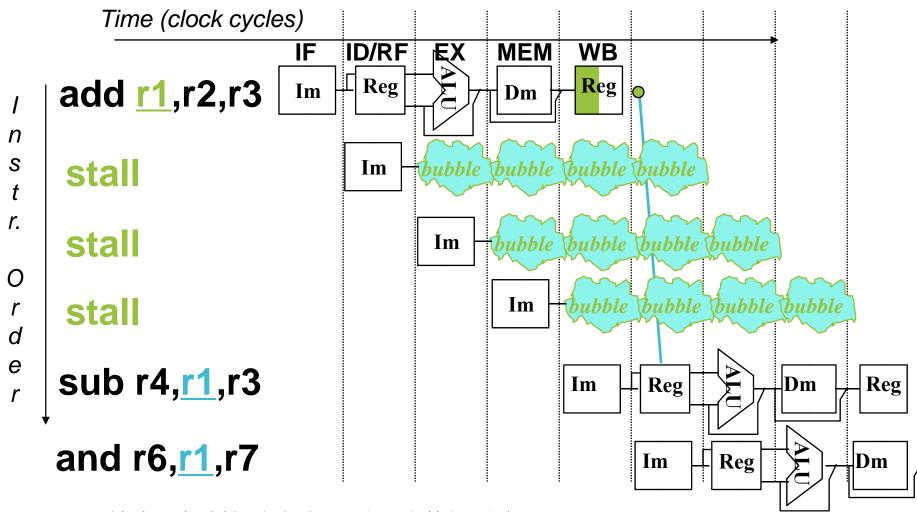
最后一条指令的r1才是新的值!

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"

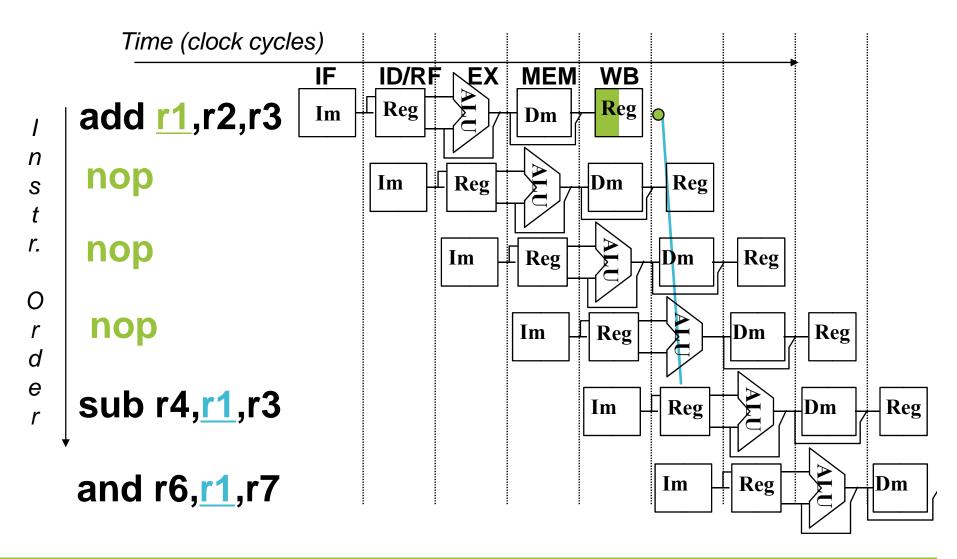


•缺点:控制相当复杂,需要改数据通路!





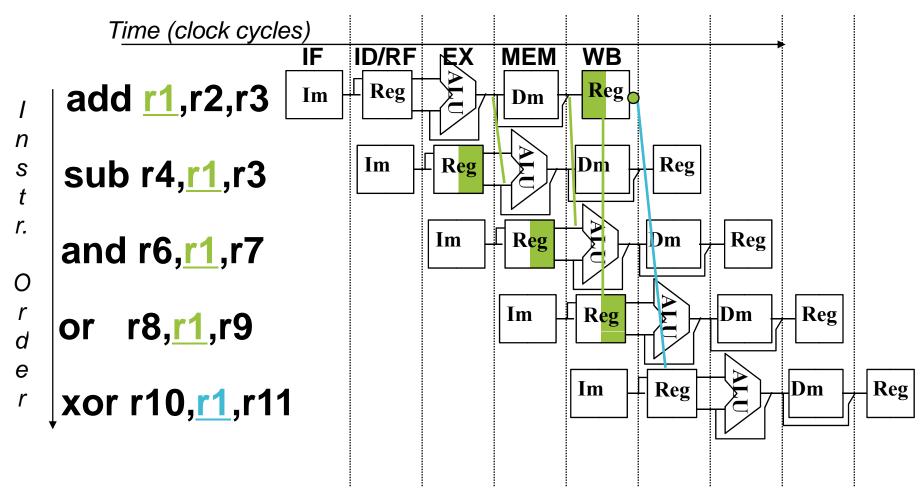
• 最差的做法:由编译器插入三条NOP指令,浪费三条指令的空间和时间



### 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 ◆———

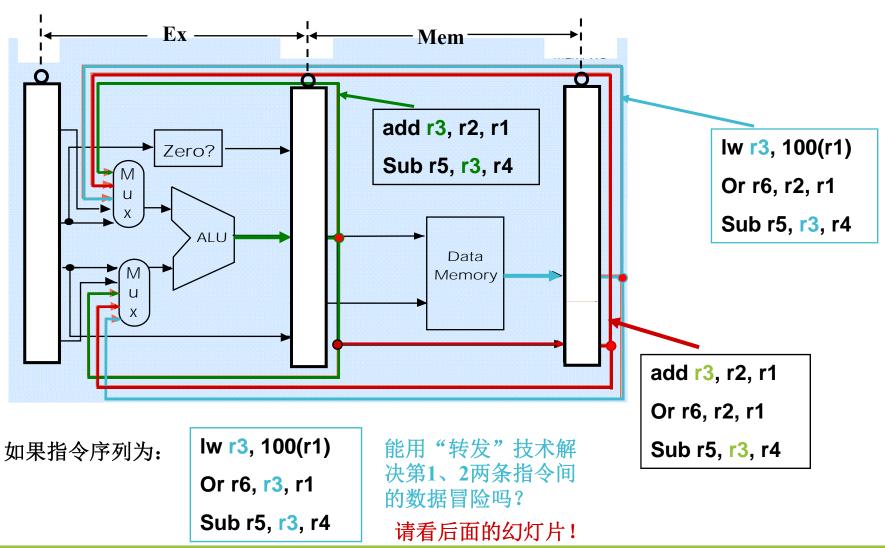
称为转发(Forwading) 或旁路(Bypassing)

2. 寄存器写/读口分别在前/后半周期, 使写入被直接读出

# 硬件上的改动以支持"转发"技术

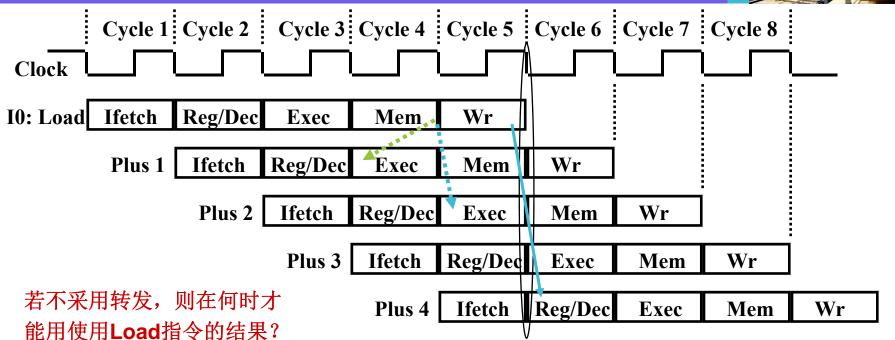


- 加MUX, 使流水段寄存器值返送ALU输入端
- 假定流水段寄存器能读出新写入的值 (否则,需要更多的转发数据)



## 复习: Load指令引起的延迟





❖ Load指令最早在哪个流水线寄存器中开始有后续指令需要 寄作?在第四周期结束时,数据在流水段寄存器中已经有值。

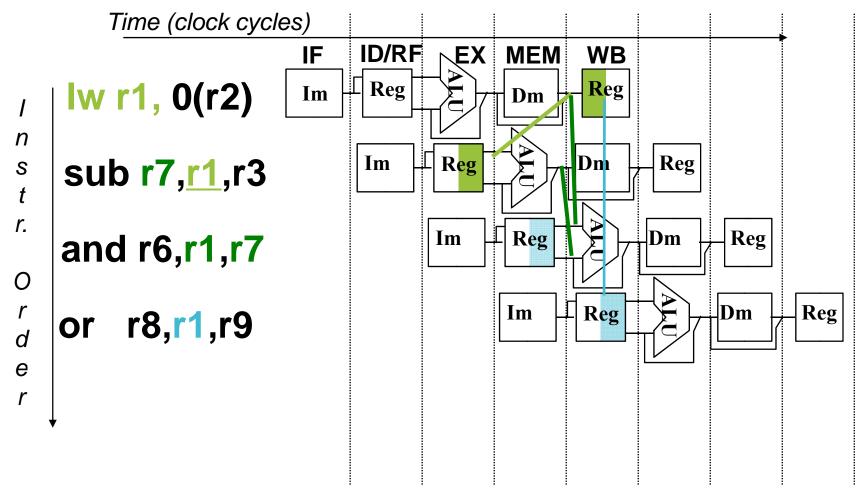
采用数据转发技术可以使load指令后面第二条指令得到所需的值

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

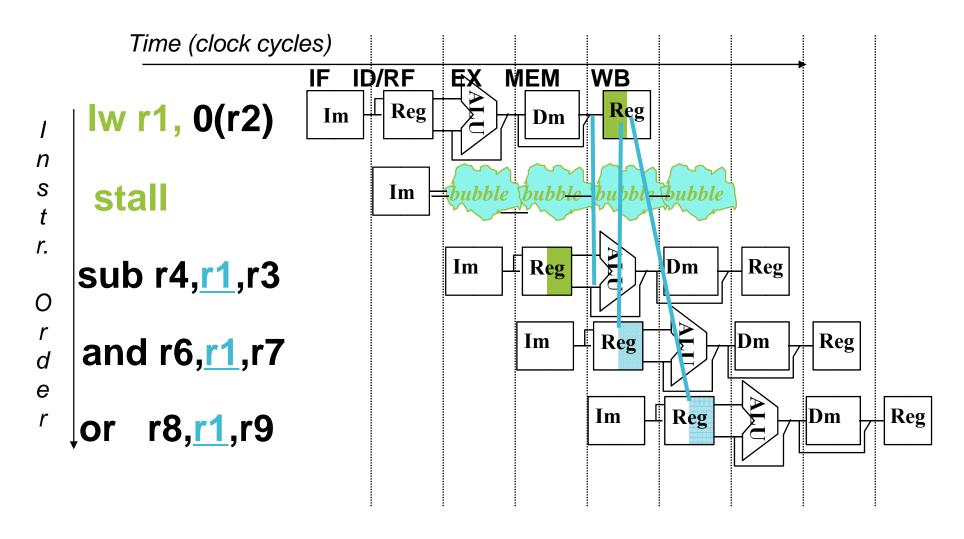


采用"转发"后仅第二条指令 SUB r7,r1,r3 不能按时执行!需要阻塞一个周期。 发生"装入-使用数据冒险"时,需要对load后的指令阻塞一个时钟周期!





#### 用硬件阻塞一个周期(指令被重复执行一次)

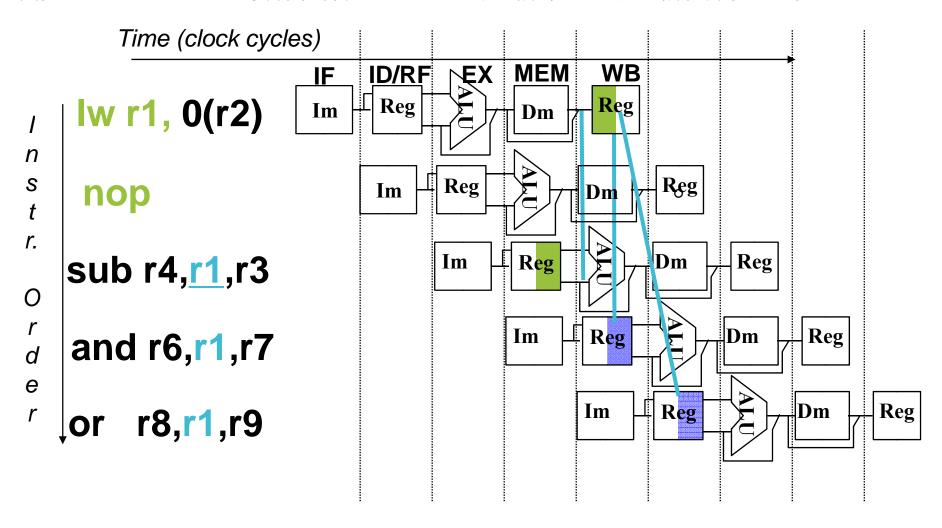


### 方案2: 软件上插入NOP指令来解决load-use



#### •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:

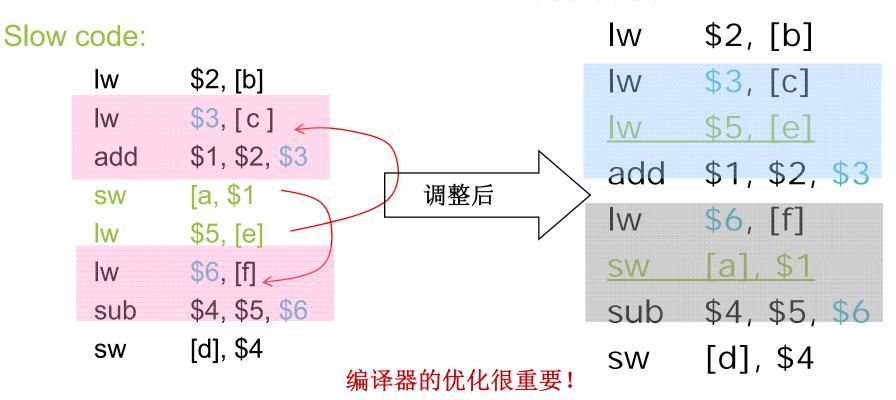
以下源程序可生成两种不同的代码,优化的代码可避免Load阻塞

$$a = b + c$$
;

$$d = e - f$$
;

假定 a, b, c, d ,e, f 在内存

#### Fast code:



# 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

# 数据冒险的解决方法



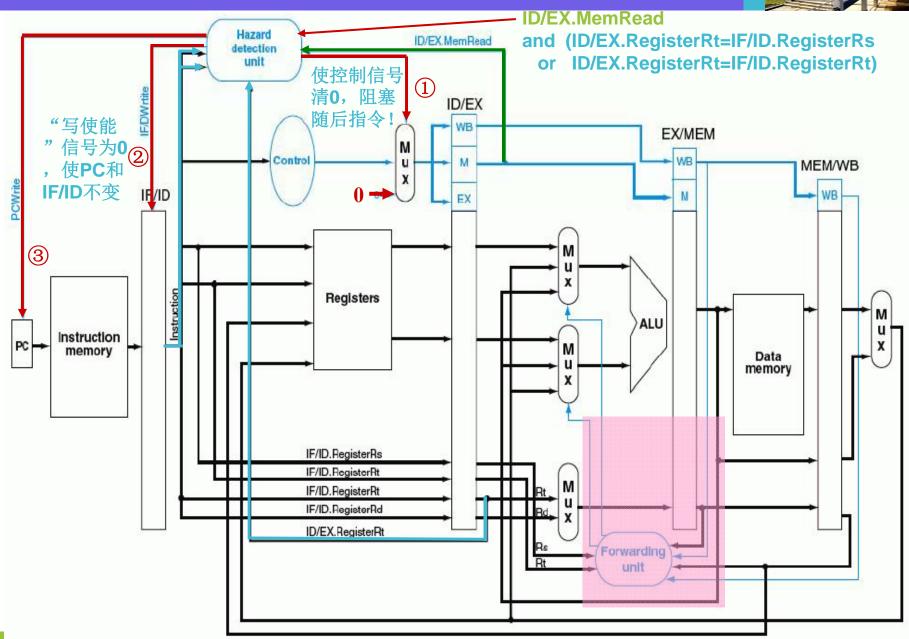
- ❖ 方法5: 转发(Forwarding或Bypassing 旁路)技术, 能解决所有数据冒险吗?
  - 若相关数据是ALU结果,则如何? 可通过转发解决
  - 若相关数据是上条指令DM读出内容,则如何? 不能通过转发解决,随后指令需被阻塞一个时钟 或 加NOP指令

称为Load-use数据冒险!

#### 实现"转发"和"阻塞"要修改数据通路:

- (1) 检测何时需要"转发",并控制实现"转发"
- (2) 检测何时需要"阻塞",并控制实现"阻塞"

## 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:<u>延迟分支</u>(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

## 简单(静态)分支预测方法



### ❖ 基本做法

■ 总预测条件不满足(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

- 预测失败时,需把流水线中<u>三条错误预测指令</u>丢弃掉
  - 将三条丢弃指令的控制信号值设置为0,使其后续过程中执行nop操作(注:涉及到当时在IF、ID和EX三个阶段的指令)

#### ❖ 性能

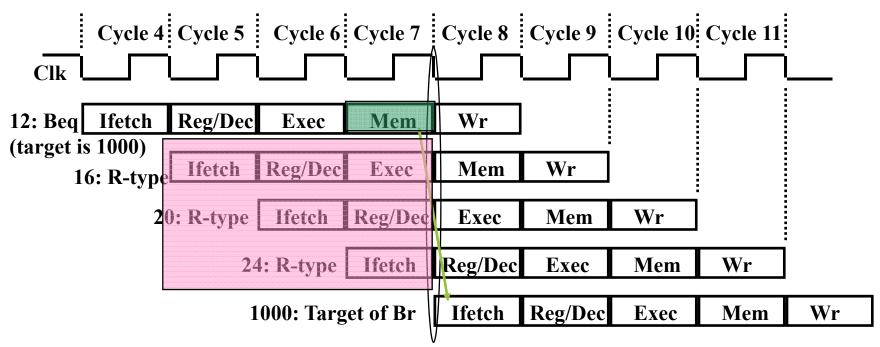
■ 如果转移概率是50%,则预测正确率仅有50%

### ❖ 预测错误的代价 那最早可以提前到哪个阶段呢?

- 预测错误的代价与何时能确定是否转移有关。越早确定代价越少
- 可以把"是否转移"的确定工作提前,而不要等到MEM阶段才确定

## Control Hazard现象





- ❖ 虽然Beq指令在第四周期取出,但:
  - "是否转移"在Mem阶段确定,目标地址在第七周期才被送到PC输入端
  - 第八周期才取出目标地址处的指令执行

结果: 在取目标指令之前,已有三条指令被取出,取错了三条指令!

- ❖ 发生转移时,要在流水线中清除Beq后面的三条指令,分别在EXE、ID、IF段中
- ❖ 延迟损失时间片C: 发生转移时,给流水线带来的延迟损失

## 简单(静态)分支预测方法

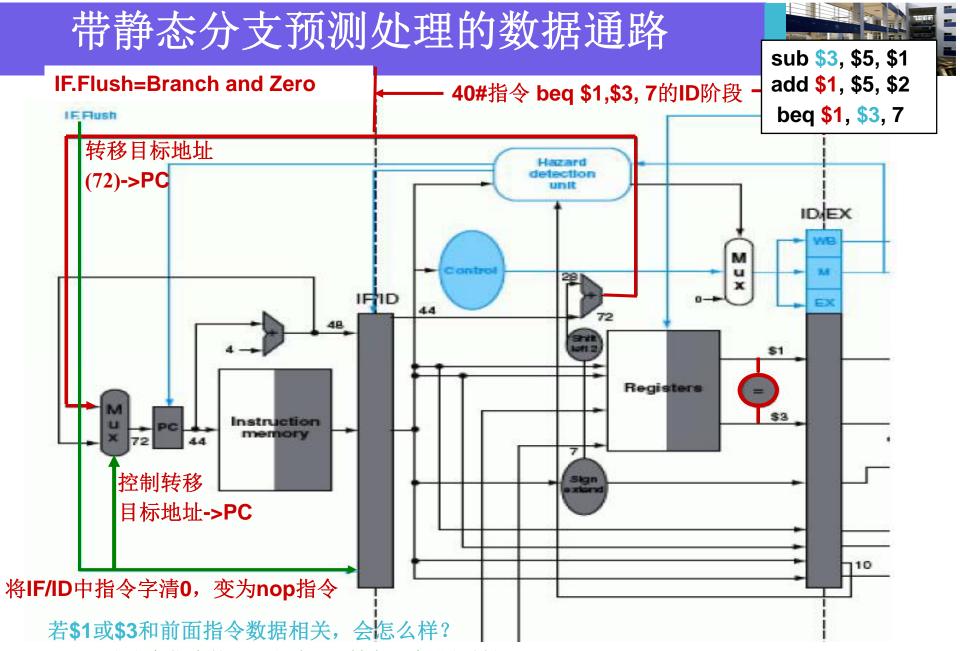


- ❖ 缩短分支延迟,减少错误预测代价
  - 可以通过调整"转移地址计算"和"分支条件判断"操作到ID阶段来缩短延迟
    - 将转移地址生成从MEM阶段移到ID阶段,可以吗?为什么?
    - (是可能的:IF/ID流水段寄存器中已经有PC的值和立即数)
    - 将"判0"操作从EX阶段移到ID阶段,可以吗?为什么?
    - (用逻辑运算(如,先按位异或,再结果各位相或)来直接比较Rs和Rt的值)
    - (简单判断用逻辑运算,复杂判断可以用专门指令生成条件码)
    - (许多条件判断都很简单)

#### ❖ 预测错误的检测和处理(称为"冲刷、冲洗" -- Flush)

- 当Branch=1并且Zero=1时,发生转移(taken)
- 增加控制信号:IF.Flush=Branch and Zero,取值为1时,说明预测失败
- 预测失败(条件满足)时,完成以下两件事(延迟损失时间片C=1时):
  - ① 将转移目标地址->PC
  - ② 清除IF段中取出的指令,即:将IF/ID中的指令字清0,转变为nop指令

原来要清除三条指令,调整后只需要清除一条指令,因而只延迟一个时钟周期,每次预测错误减少了两个周期的代价!



- 上上条指令的EXE段结果可转发回来进行判断
- ·上条指令的EXE段结果来不及转发回来,引起1次阻塞!

## 动态分支预测方法

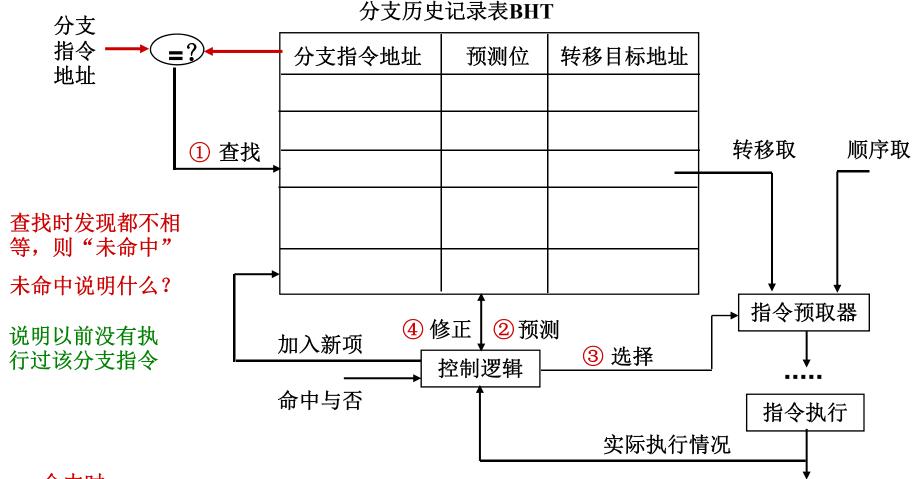


- ❖ 简单的静态分支预测方法的预测成功率不高,应考虑动态预测
- ❖ 动态预测基本思想:
  - 利用最近转移发生的情况,来预测下一次可能发生的转移
  - 预测后,在实际发生时验证并调整预测
  - 转移发生的历史情况记录在BHT中(有多个不同的名称)
    - 分支历史记录表BHT (Branch History Table)
    - 分支预测缓冲BPB (Branch Prediction Buffer)
    - 分支目标缓冲BTB (Branch Target Buffer)
  - 每个表项由分支指令地址的低位索引,故在IF阶段就可以取到预测位
    - 低位地址相同的分支指令共享一个表项,所以,可能取的是其他分支指令的预测位。会不会有问题?
    - 由于仅用于预测,所以不影响执行结果

现在几乎所有的处理器都采用动态预测(dynamic predictor)

## 分支历史记录表BHT(或BTB、BPB)





#### 命中时:

根据预测位,选择"转移取"还是"顺序取"

#### 未命中时:

加入新项,并填入指令地址和转移目标地址、初始化预测位

# 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<u>预测状态图</u>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<u>预测状态图</u>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** 

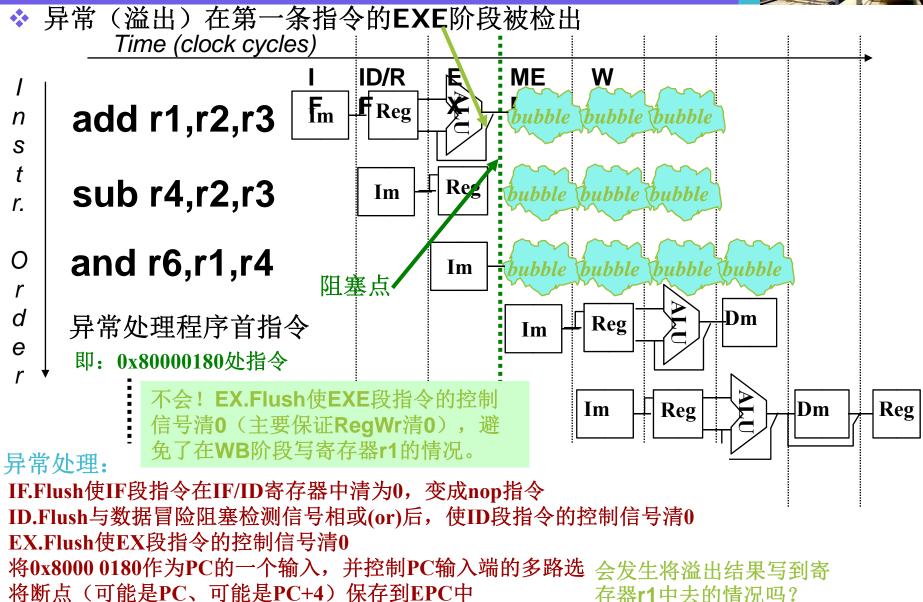
# 另一种控制冒险: 异常和中断



- \* 异常和中断会改变程序的执行流程
- ❖ 某条指令发现异常时,后面多条指令已被取到流水线中正在执行
  - 例如ALU指令发现"溢出"时,已经到EX阶段结束了,此时, 它后面已有两条指令进入流水线了
- ❖ 流水线数据通路如何处理异常? (举例说明)
  - 假设指令add r1,r2,r3产生了溢出
     (记住: MIPS异常处理程序的首地址为0x8000 0180)
  - 处理思路:
    - ☀ 清除add指令以及后面的所有已在流水线中的指令
    - ☀ 保存PC或PC+4 到 EPC
    - \* 从0x8000 0180处开始取指令

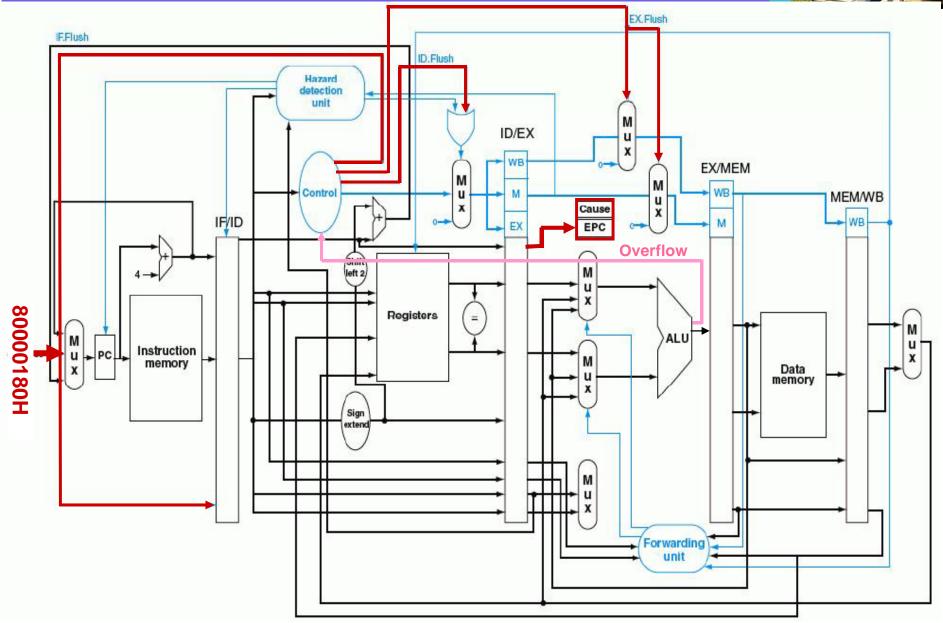
## 异常的处理





# 带异常处理的流水线数据通路





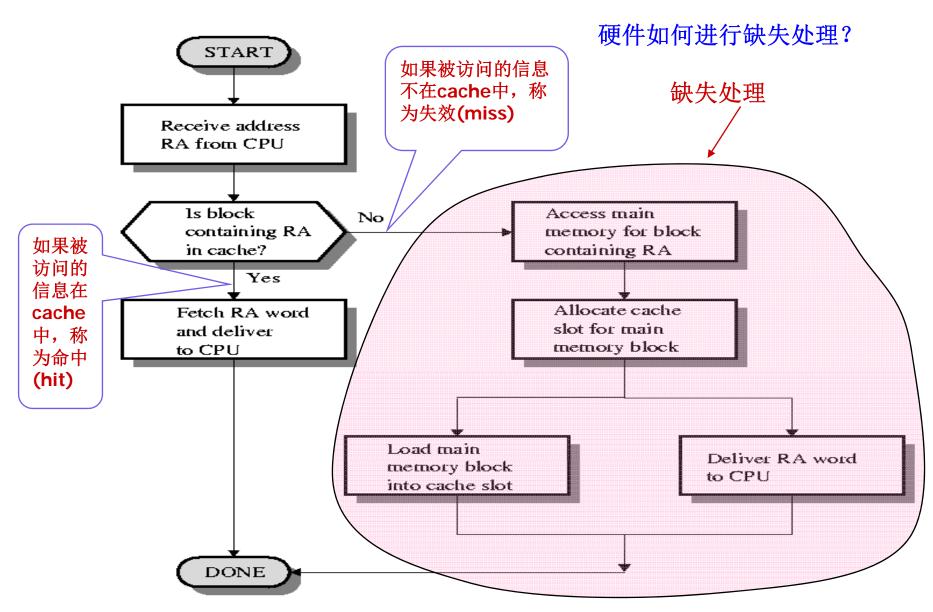
### 流水线方式下的异常处理的难点问题



- ※ 流水线中有5条指令,到底是哪一条发生异常?
  - 根据异常发生的流水段可确定是哪条指令,因为各类异常发生的流水段不同。
    - ☀ "溢出"在EXE段检出
    - ★ "无效指令"在ID段检出
    - ☀ "除数为0"在ID段段检出
    - ☀ "无效指令地址"在IF段检出
    - ☀ "无效数据地址"在Load/Store指令的EXE段检出
- ❖ 外部中断与特定指令无关,如何确定处理点?
  - 可在IF段或WB段中进行中断查询,需要保证当前WB段的指令能正确完成,并在有中断 发生时,确保下个时钟开始执行中断服务程序
- ❖ 检测到异常时,指令已经取出多条,当前PC的值已不是断点,怎么办?
  - 指令地址(哪条?)存放在流水段R,可把这个地址送到EPC保存,以实现精确中断 (非精确中断不能提供准确的断点,而由操作系统来确定哪条指令发生了异常)
- ❖ 一个时钟周期内可能有多个异常,该先处理哪个?
  - 异常:检出异常存到专门寄存器,
  - 中断:在中断查询程序或中断优先级排队电路中按顺序查询
- ❖ 系统中只有一个EPC,多个中断发生时,一个EPC不够放多个断点,怎么办?
  - 总是把优先级最高的送到EPC中
- 在异常处理过程中,又发生了新的异常或中断,怎么办?
  - 利用中断屏蔽和中断嵌套机制来处理

### 复习: Cache 的操作过程





### Cache缺失处理会引起流水线阻塞(停顿)



- ❖ 在使用Cache的系统中,数据通路中的IM和DM分别是Code Cache和Data Cache
- ❖ CPU执行指令过程中,取指令或取数据时,如果发生缺失,则指令执行被阻塞
- ❖ Cache缺失的检测(如何进行的?记得吗?)
  - Cache中有相应的检测线路(地址高位与Cache标志比较)
- \* 阻塞处理过程
  - 冻结所有临时寄存器和程序员可见寄存器的内容(即: 使整个计算机阻塞)
  - 由一个单独的控制器处理Cache缺失,其过程 (假定是指令缺失) 还记得吗?:
    - 把发生缺失的指令地址(PC-4)所在的主存块首址送到主存
    - 启动一次"主存块读"操作,并等待主存完成一个主存块(Cache行)的读操作
    - 把读出的一个主存块写到Cache对应表项的数据区 (若对应表项全满的话,还要考虑淘汰掉一个已在Cache中的主存块)
    - 把地址高位部分(标记)写到Cache的"tag"字段,并置"有效位"
    - 重新执行指令的第一步: "取指令"
  - 若是读数据缺失,其处理过程和指令缺失类似
    - 从主存读出数据后,从"取数"那一步开始重新执行就可以了
  - 若是写数据缺失,则要考虑用哪种"写策略"解决"一致性"问题
- ❖ 比数据相关或分支指令引起的流水线阻塞简单:只要保持所有寄存器不变
- ❖ 与中断引起的阻塞处理不同:不需要程序切换

### TLB缺失和缺页也会引起流水线阻塞



#### **❖ TLB**缺失处理

- 当TLB中没有一项的虚页号与要找的虚页号相等时,发生TLB miss
- TLB miss说明可能发生以下两种情况之一:
  - 页在内存中: 只要把主存中的页表项装载到TLB中
  - 页不在内存中(缺页): OS从磁盘调入一页,并更新主存页表和TLB

### ❖ 缺页(page fault)处理

- 当主存页表的页表项中 "valid"位为 "0"时,发生page fault
- Page fault是一种"故障"异常,按以下过程处理(MIPS异常处理)
  - 在Cause寄存器置相应位为"1"
  - 发生缺页的指令地址(PC-4)送EPC
  - 0x8000 0180(异常查询程序入口)送PC
    - 执行OS的异常查询程序,取出Cause寄存器中相应的位分析,得知发生了"缺页",转到"缺页处理程序"执行
- page fault一定要在发生缺失的存储器操作时钟周期内捕获到,并在下个时钟 转到异常处理,否则,会发生错误。
  - 例: lw \$1, 0(\$1), 若没有及时捕获"异常"而使\$1改变,则再重新执行该指令时, 所读的内存单元地址被改变,发生严重错误!

处理Cache缺失和缺页的不同之处在哪里? 哪种要进行程序切换?

## (缺页) 异常处理时要考虑的一些细节



### ❖ 缺页异常结束后,回到哪里继续执行?

- 指令缺页: 重新执行发生缺页的指令
- 数据缺页:
  - 简单指令(仅一次访存): 强迫指令结束, 重新执行缺页指令
  - 复杂指令(多次访存):可能会发生多次缺页,指令被中止在中间某个阶段,缺页处理后,从中间阶段继续执行;因而,需要能够保存和恢复中间机器状态

### ❖ 异常发生后,又发生新的异常,怎么办?

- 在发现异常、转到异常处理程序中,若在保存正在运行进程的状态时又发生新的异常,则因为要处理新的异常,会把原来进程的状态和保存的返回断点破坏掉,所以,应该有一种机制来"禁止"响应新的异常处理(?)
- 通过"中断/异常允许"状态位(或"中断/异常允许"触发器)来实现?
- "中断/异常允许"状态位置1,则"开中断"(允许异常),清0则"关中断"(禁止异常)?
- OS通过管态指令来设置该位的状态?

# 三种处理器实现方式的比较



❖ 单周期、多周期、流水线三种方式比较

假设各主要功能单元的操作时间为:

• 存储单元: 200ps

• ALU和加法器: 100ps

• 寄存器堆(读/写): 50ps

假设MUX、控制单元、PC、扩展器和传输线路都没有延迟,指令组成为:

25%取数、10%存数、52%ALU、11%分支、2%跳转

则下面实现方式中,哪个更快?快多少?

- (1) 单周期方式: 每条指令在一个固定长度的时钟周期内完成
- (2) 多周期方式:每类指令时钟数:取数-5,存数-4,ALU-4,分支-3,跳转-3
- (3)流水线方式:每条指令分取指令、取数/译码、执行、存储器存取、写回五阶段(假定没有结构冒险,数据冒险采用转发处理,分支延迟槽为1,预测准确率

为75%;不考虑异常、中断和访问缺失引起的流水线冒险)

# 三种处理器实现方式的比较



解: CPU执行时间=指令条数 x CPI x 时钟周期长度

三种方式的指令条数都一样,所以只要比较CPI和时钟周期长度即可。

Instruction class	Functional units used by the instruction class								
R-type	Instruction fetch	Register access	ALU	Register access					
Load word	Instruction fetch	Register access	ALU	Memory access	Register access				
Store word	Instruction fetch	Register access	ALU	Memory access					
Branch	Instruction fetch	Register access	ALU						
Jump	Instruction fetch								

#### 各指令类型要求的时间长度为:

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

# 三种处理器实现方式的比较



#### 对于单周期方式:

时钟周期将由最长指令来决定,应该是load指令,为600ps 所以,N条指令的执行时间为600N(ps)

#### 对于多周期方式:

时钟周期将取功能部件最长所需时间,应该是存取操作,为**200ps** 根据各类指令的频度,计算平均时钟周期数为:

CPU时钟周期=5x25%+4x10%+4x52%+3x11%+3x2%=4.12

所以, N条指令的执行时间为4.12x200xN=824N(ps)

#### 对于流水线方式:

Load指令: 当发生Load-use依赖时, 执行时间为2个时钟, 否则1个时钟, 故平均执行时间为1.5个时钟;

Store、ALU指令: 1个时钟;

Branch指令: 预测成功时, 1个时钟, 预测错误时, 2个时钟,

所以: 平均约为: .75x1+.25x2=1.25个;

Jump指令: 2个时钟(总要等到译码阶段结束才能得到转移地址)

平均CPI为: 1.5x25%+1x10%+1x52%+1.25x11%+2x2%=1.17

所以, N条指令的执行时间为1.17x200xN=234N(ps)

## 流水线冒险对程序性能的影响



- ❖ 结构冒险对浮点运算的性能影响较大,因为浮点运算单元不能有 效被流水化,可能造成运算单元的资源冲突
- ❖ 控制冒险更多出现在整数运算程序中,因为分支指令对应于循环 或选择结构,大多由整数运算结果决定分支
- ❖ 数据冒险在整数运算程序和浮点运算程序中都一样
  - 浮点程序中的数据冒险容易通过编译器优化调度来解决
    - 分支指令少
    - 数据访问模式较规则
  - 整数程序的数据冒险不容易通过编译优化调度解决
    - 分支指令多
    - 数据访问模式不规则
    - 过多使用指针

## 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



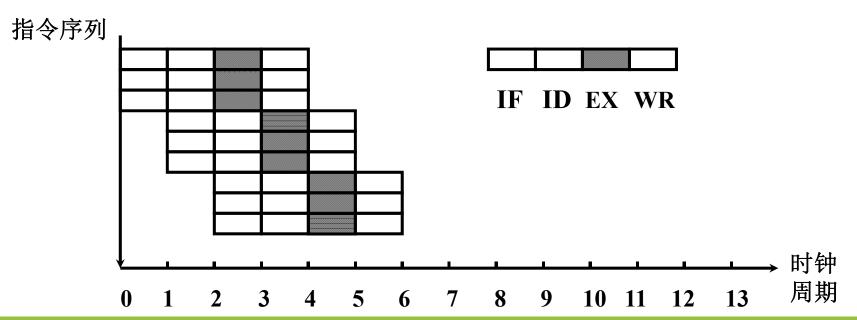
- ❖流水线的多发技术:为了提高流水线性能, 设法在一个时钟周期内产生更多条指令。
  - 超标量技术
  - 超流水线技术
  - 超长指令字

# 1. 超标量技术(Super Scalar)



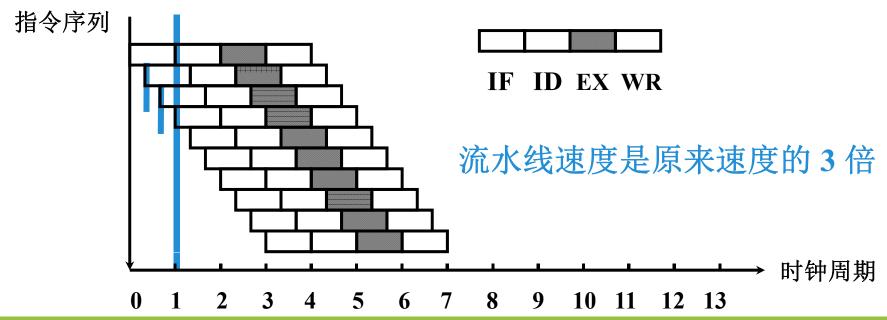
- >每个时钟周期内可 并发多条独立指令
- ▶配置多个功能部件,一个时钟周期内,有多个功能部件 同时执行多条指令。
- > 不能调整 指令的 执行顺序

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)

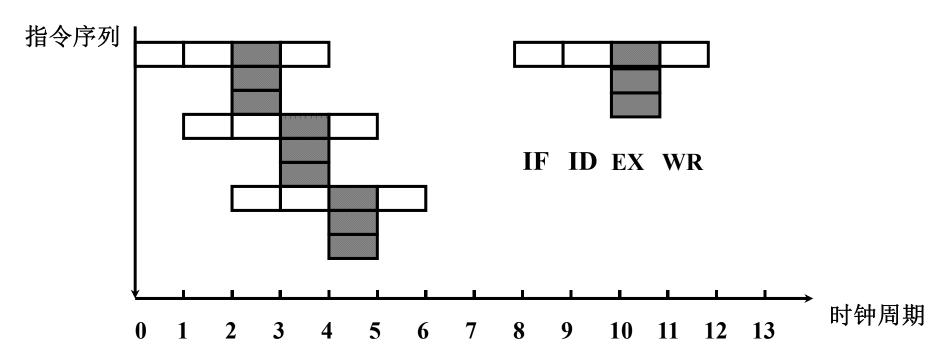
- > 将流水寄存器插入流水线段,好比将流水线再次分段
- > 不能调整 指令的 执行顺序 靠编译程序解决优化问题



## 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

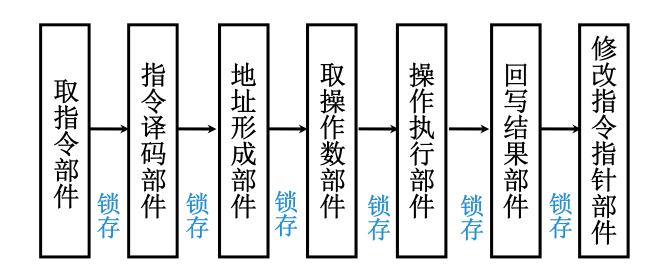


## 六、流水线结构



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

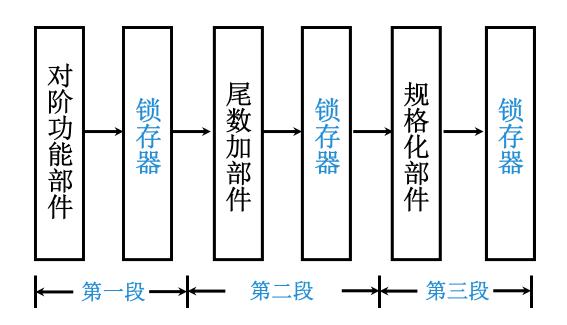
7 个时钟周期出 1 结果

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

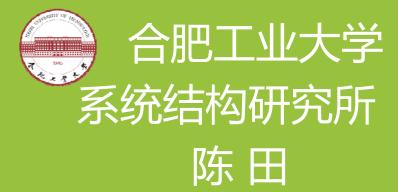
Computer Organization

# Thank You



Institute of Computer Architecture





### 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

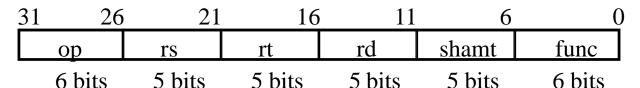


• 所有指令都是32位宽,须按字地址对齐

R-Type指令

◆ 有三种指令格式

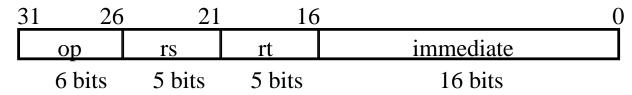
R-Type



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
31 26 J-Type指令
op target address
6 bits 26 bits

### MIPS指令字段含义



R-Type指令

OP:操作码

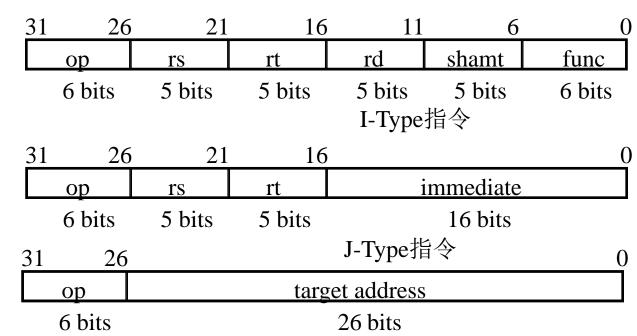
rs:第一个源操 作数寄存器

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)	ор=0: R <sup>#</sup>	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	B1tz/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	F1Pt						
3(011)	Ö		(3)					Č.
4(100)	load byte	load half	lwl	load word	1bu	1hu	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	1wc0	lwc1	ĺ.					v v
7(111)	swc0	swc1						

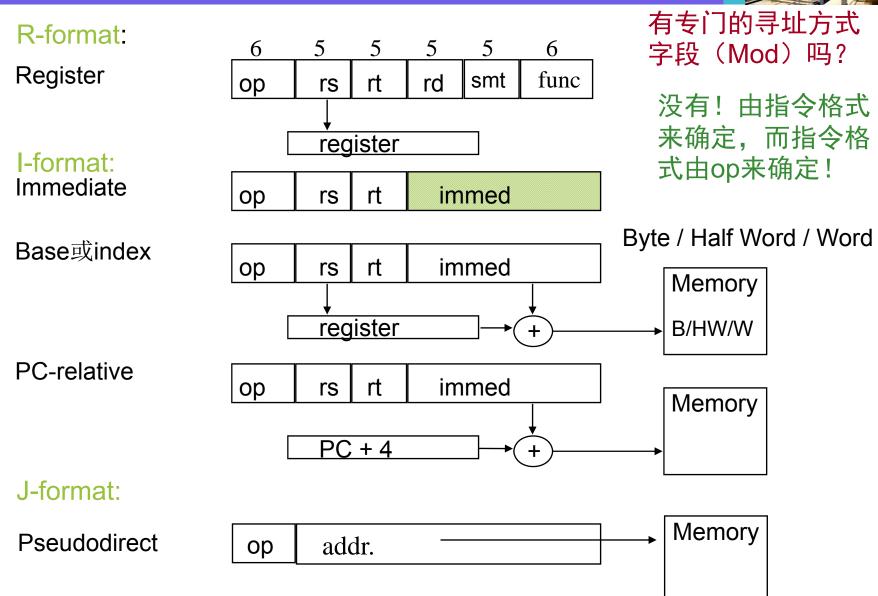
### R-Type指令的解码(op=0时,func字段的编码解码表

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

7(111)

#### MIPS Addressing Modes(寻址方式)





### 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<u>指令解码表</u>知,是一条R-Type指令,按照 R-Type指令的格式

31 <sup>6 bits</sup> 26	5 bits 21	5 bits 16	5 bits 11	5 bits 6	6 bits	0
op	rs	rt	rd	shamt	func	
000000	00101	01111	10000	00000	100000	_

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).

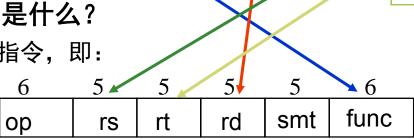
### Example: 汇编形式与指令的对应



**❖若MIPS Assembly Instruction:** 

则对应的指令机器代码是什么?

从助记符表中查到Add是R型指令,即:



Add \$t0,\$\$1,\$\$2→

Decimal representation:

_ 6	5	5	5	5	6
0	17	18	8	0	32
R-Type	<b>\$</b> s1	\$s2	\$t0	No shift	Add

问题:如何知道是R型指令?

汇编器

根据汇编指令中的操作码助记符 查表能知道是什么格式!

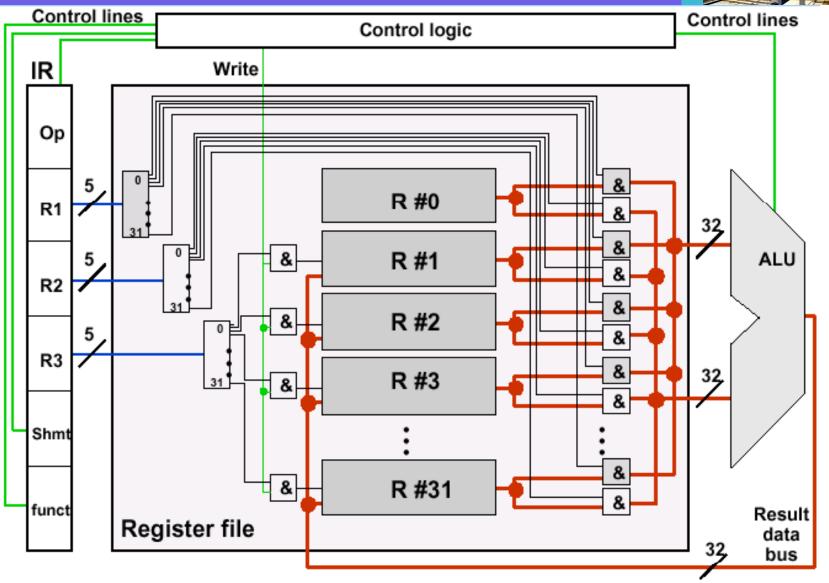
Binary representation:

6	5	5	5	5	6
000000	10001	10010	01000	00000	100000

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)

- 少 装入指令寄存器IR
- 少 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)
- 少 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

#### Phase2: Execution(2: 执行阶段)

- 少 寄存器号被送选择器
- 少 对应选择器输出被激活
- 少 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
- 少 结果被写回目标寄存器

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。

#### MIPS指令中寄存器数据和存储器数据的指定

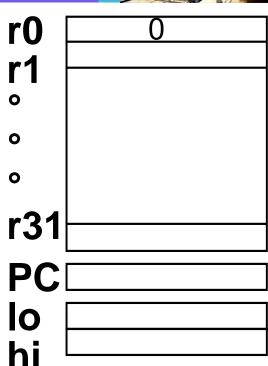


#### ❖寄存器数据指定:

- 31 x 32-bit GPRs (r0 = 0)
- 寄存器编号占5 bit
- 32 x 32-bit FP regs (f0  $\sim$  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<sup>32</sup>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 $\sim$ v1	<b>2</b> ~ <b>3</b>	values for results(过程调用返回值)	no
a0 $\sim$ a3	4~7	Arguments(过程调用参数)	yes
t0 ∼ t7	8 ~ 15	Temporaries(临时变量)	no
s0 ~ s7	16 ~ <b>23</b>	Saved(保存)	yes
t8 $\sim$ t9	$24\sim25$	more temporaries(其他临时变量)	no
k0 $\sim$ k1	<b>26</b> ~ <b>27</b>	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.

Z	zero	at	v0-v1	a0 - a3	t0 - 1	t <b>7</b>	s0 - s	7	t8 - t9	k0 - k1	gp	sp	fp	ra
(	)	1	2 - 3	4 - 7	8	15	16	23	24 - 25	26 - 27	28	29	30	31

#### 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	$Hi, Lo = \$2 \times \$3$	64-bit signed product
divide	div \$2,\$3	$L_0 = \$2 \div \$3,$	Lo = quotient, Hi = remainder
		$Hi = \$2 \mod \$3$	
Move from Hi	mfhi \$1	\$1=Hi	get a copy of Hi
Move from Hi Move from Lo	mfhi \$1 mflo \$1	\$1=Hi \$1=lo	get a copy of Hi
	•		get a copy of Hi <u>Comment</u>
Move from Lo	mflo \$1	\$1=lo	
Move from Lo <i>Instruction</i>	mflo \$1  Example	\$1=lo  Meaning	<u>Comment</u>
Move from Lo <u>Instruction</u> and	mflo \$1  Example  and \$1,\$2,\$3	\$1=lo  Meaning  \$1 = \$2 & \$3	Comment Logical AND
Move from Lo  Instruction and or	mflo \$1  Example  and \$1,\$2,\$3  or \$1,\$2,\$3	\$1=lo  Meaning  \$1 = \$2 & \$3  \$1 = \$2   \$3	Comment Logical AND Logical OR

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

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. 
$$f=(g+100) - (i+50)$$
  $\longrightarrow$  addi \$7, \$2, 100 addi \$8, \$4, 50 sub \$1, \$7, \$8

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	<u>Meaning</u>
SW \$3,500(\$4)	Store word	$$3 \rightarrow ($4+500)$
SH \$3, 502(\$2)	Store half	Low Half of $\$3 \rightarrow (\$2+502)$
SB \$2, 41(\$3)	Store byte	LQ of $$2 \rightarrow ($3+41)$
LW \$1, -30(\$2)	Load word	$(\$2 - 30) \rightarrow \$1$
LH \$1, 40(\$3)	Load half	$(\$3+40) \rightarrow LH \text{ of } \$1$
LB \$1, 40(\$3)	Load byte	$(\$3+40) \rightarrow LQ \text{ of }\$1$

操作数长度的不同由不同的操作码指定。

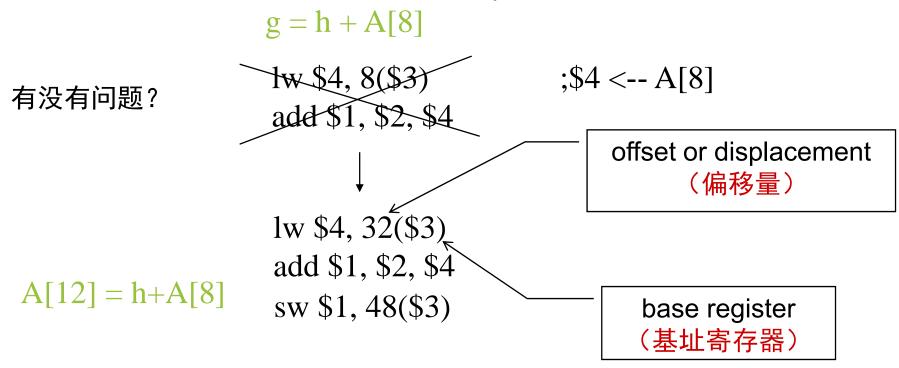
问题: 为什么指令必须支持不同长度的操作数?

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



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

mult \$5,5\$6

mflo \$7

 $\frac{1}{3}$ \$6 - 4

; Hi,Lo = i\*4

Index Register (变址寄存器)

; \$6 = i\*4, assuming i is small

add \$4, \$3, \$7 lw \$4, 0(\$4) add \$1, \$1, \$4

addi \$5, \$5, 1

; \$4 <-- address of A[i]
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



call / return

<u>Instruction</u>	<u>Example</u>	<u>Meaning</u>
jump register	jr \$31	go to \$31
	For switch, pro	ocedure return
jump and link	jal 10000 <i>For <mark>procedure</mark></i>	\$31 = PC + 4; go to 10000 call
jump	i 10000	go to 10000
•	Jump to target	address

Pseudoinstruction blt, ble, bgt, bge not implemented by hardware, but synthesized by assembler 伪指令: 硬件不能直接执行

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

问题:指令中立即数是多少? 100=0064H

branch on not eq. bne \$1,\$2,100 if (\$1!= \$2) go to PC+4+100

问题:指令中立即数是多少? 25=0019H

按补码比 较大小

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
                                       ; jump to exit
          j 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
bne $6, $zero, else
add $3, $4, $5
i exit
```

else: sub \$3, \$4, \$5

exit:

slt \$6, \$1, \$2 beq \$6, \$zero, else

add \$3, \$4, \$5

i exit

else: sub \$3, \$4, \$5

exit:

; if a<b, \$6=1, else \$6=0

; \$6!=0, jump to else

; jump to exit

; if a<b, \$6=1, else \$6=0

; \$6=0, jump to else

; jump to 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  $\sim$  \$1, \$2, \$3, \$4 and base address of array is in \$5

```
Loop: add $7, $3, $3 ; i*2 加法比乘法快! add $7, $7, $7 ; i*4 也可用移位来实现乘法! add $7, $7, $5 $3中是i, $7中是i*4 lw $6, 0($7) ; $6=A[i] add $1, $1, $6 ; g= g+A[i] add $3, $3, $4 bne $3, $2, Loop
```

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; ◄
                    —— i是全局静态变量
void set_array(int num)
                          _array数组是局部变量
    int array[10];
    for (i = 0; i < 10; i ++) {
                                set array是调用过程
     arrar[i] = compare (num, i);
                                compare是被调用过程
int compare (int a, int b)
                            compare是调用过程
                            sub是被调用过程
   if (sub(a, b) >= 0)
      return 1;
    else
      return 0;
                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?
int sub (int a, int b)
                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?
  return a-b;
                   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 J
  - 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

中完成

在调用过程P

- **❖ MIPS中用于过程调用的指令(见<u>MIPS过程调用指令</u>)**
- ❖ 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

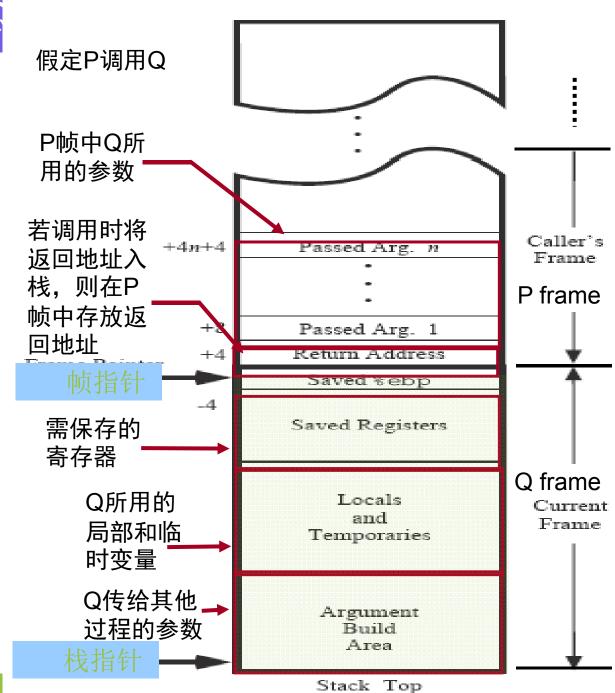
从高→低地址"增长",而取数/存数的方向是低→高地址(大端方式)



### 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

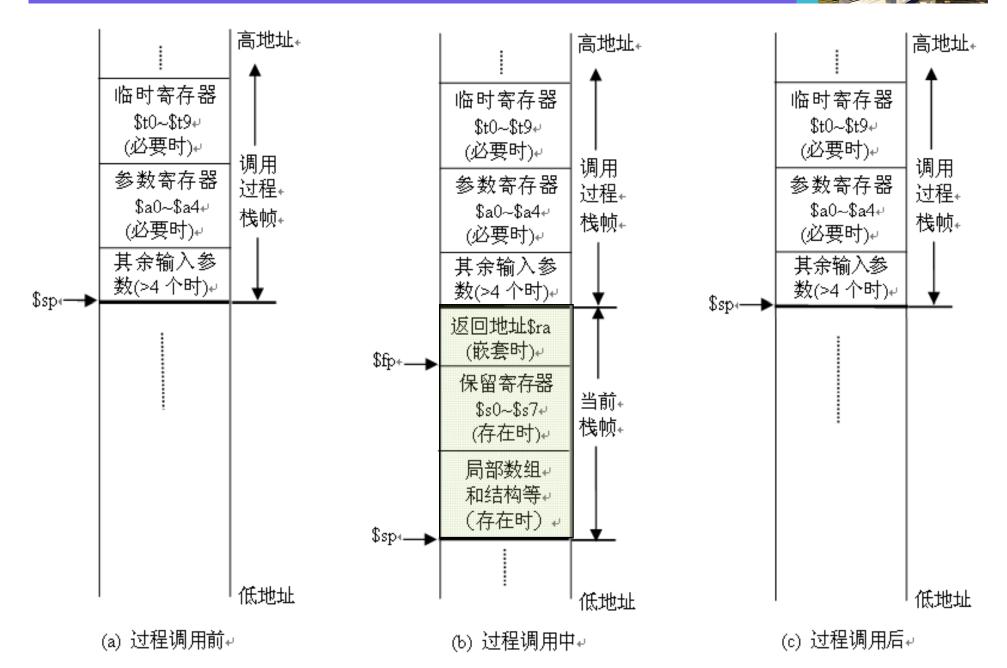


Stack Bottom

### 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。



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

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
```

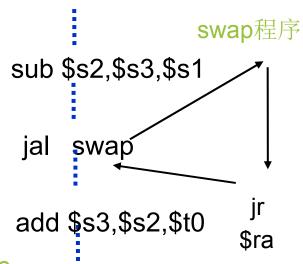
问题1: 若swap中不保存\$s2,则会发生什么情况?

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

#### 调用程序



### swap: MIPS中的一个过程示例



sll

lw

lw

SW

SW

addu



; store old v[k] into v[k+1]

```
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

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?

\$ra的内容不 会被破坏!

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, 且上述黑色指今都可去掉

\$s2, \$a1, 2

\$s2, \$s2, \$a0

\$t0, 0(\$s2)

\$s3, 0(\$s2)

\$t0, 4(\$s2)

\$s3, 4(\$s2)

### Example: 过程调用



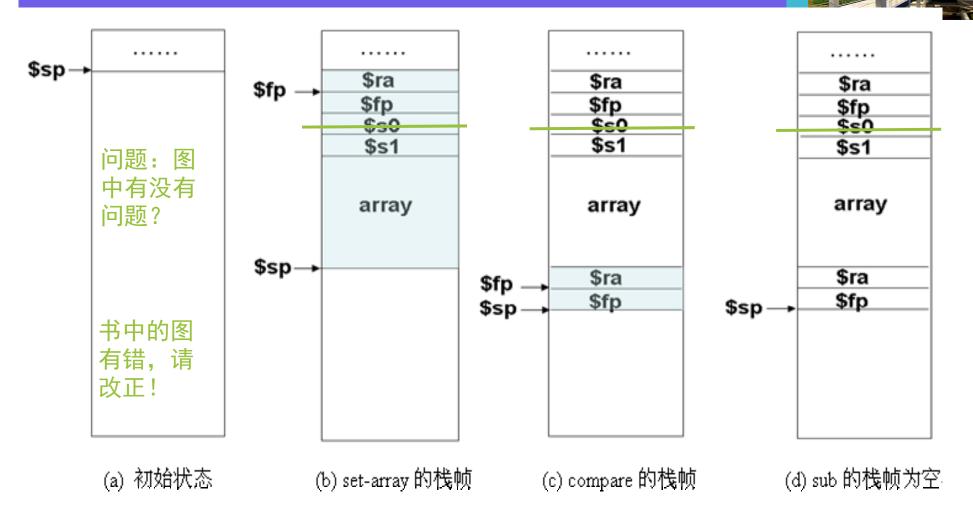
```
int i; 🗸
                        —— i是全局静态变量
void set array(int num)
                              array数组是局部变量
     int array[10];
     for (i = 0; i < 10; i ++) {
                                    set_array是调用过程
      arrar[i] = compare (num, i); 
                                    compare是被调用过程
int compare (int a, int b)
                                compare是调用过程
                                sub是被调用过程
    if ( sub (a, b) >= 0 )
       return 1;
     else
       return 0;
                       问题1:编译器如何为全局变量和局部变量
                              分配空间?
int sub (int a, int b)
                       问题2: 执行set array的结果是什么?
  return a-b;
```

### 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。

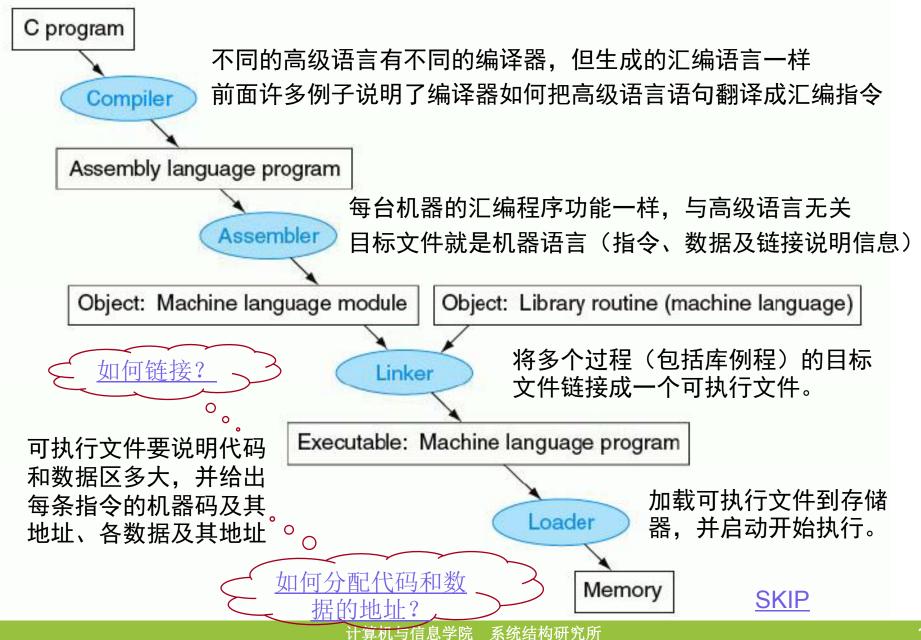
#### 过程调用举例:栈指针、帧指针和栈帧中信息的变化



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。 计算机与信息学院 系统结构研究所

#### 程序的翻译、链接和加载(自学)





#### 复习: MIPS程序和数据的存储器分配



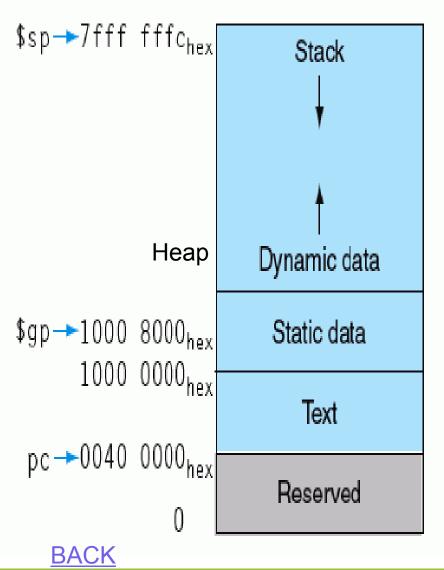
- ※ 每个MIPS程序都按如下规定进行存储器分配
- ❖ 每个可执行文件都按如下规定给出代码和数据的地址

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



### 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 <sub>hex</sub>	代码的长度为0x100		
	Data size	20 <sub>hex</sub>	数据的长度为0x20		
Text segment	Address	Instruction			
链接前地址总是从0开始	0	lw \$a0, 0(\$gp)	0是由x待定的地址		
	4	→ jal0	0是由B待定的地址		
实际是指令机器码	***	***			
Data segment	0	(X)			
链接前地址总是从0开始	• • • •				
Relocation information	Address	Instruction type	Dependency		
	0	1 w	X		
	4	jal	В		
Symbol table	Label	Address			
	Χ	<del>y -</del> :	X的地址待定		
	В	<del>y=</del> :	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 <sub>hex</sub>	代码的长度为0x200		
88	Data size	30 <sub>hex</sub>	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	Υ		
8	4	jal	A		
Symbol table	Label	Address			
1	γ	-			
Š.	Α	<u> </u>			

### 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 <sub>hex</sub>
	Data size	50 <sub>hex</sub>
Text segment	Address Instruction	
代码地址总是从0040 0000开始	0040 0000 <sub>hex</sub>	lw \$a0, 8000 <sub>hex</sub> (\$gp)
	0040 0004 <sub>hex</sub>	jal 40 0100 <sub>hex</sub>
		***
过程B从A后的0x100开始	0040 0100 <sub>hex</sub>	sw \$a1, 8020 <sub>hex</sub> (\$gp)
	0040 0104 <sub>hex</sub>	jal 40 0000 <sub>hex</sub>
		***
Data segment	Address	1000 0000=?+ 1000 8000
静态区地址从0x1000 0000开始	1000 0000 <sub>hex</sub>	( X )
		***
过程B从A后的0x20开始	1000 0020 <sub>hex</sub>	(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 op rs rt rd shamt func \$s0右移8位后送\$t2 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	1.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)