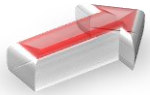
数字逻辑与处理器基础

第五部分

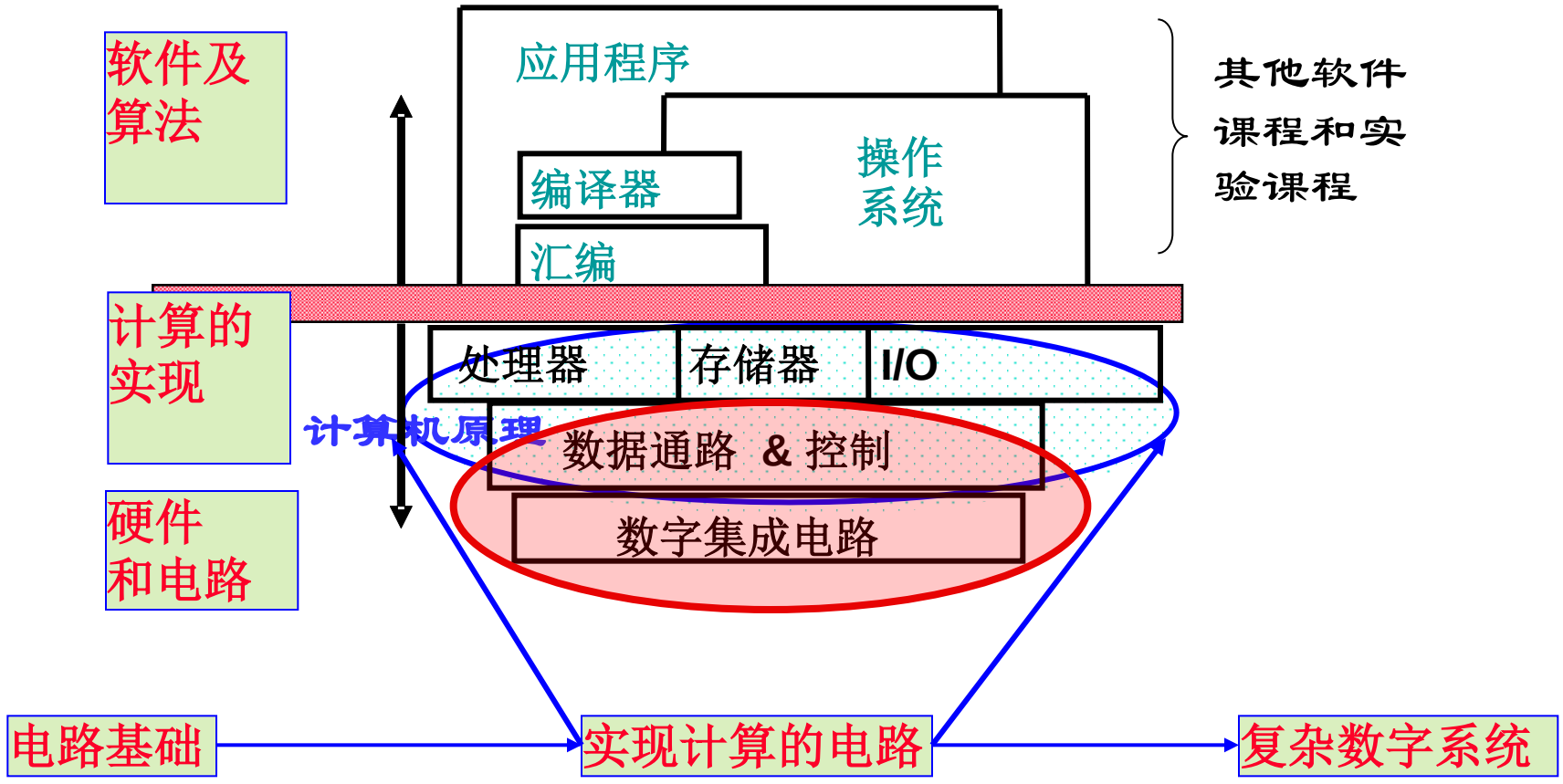
程序与指令系统

程序与指令系统



1. 基本概念
2. MIPS体系结构
3. 计算机性能评价
4. MIPS模拟器
5. MIPS汇编语言程序设计
6. 一个排序算法的实例

数字逻辑与处理器基础


软件及算法
应用程序
编译器
操作系统
汇编
其他软件
课程和实验课程
计算的实现
处理器
存储器
I/O
计算机原理
数据通路 & 控制
数字集成电路
硬件和电路
电路基础
实现计算的电路
复杂数字系统

关于计算机：A big challenge

Personal Computer:
Hardware & Software

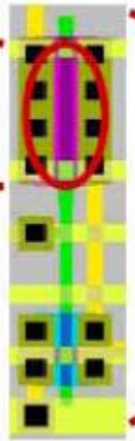
Circuit Board:
≈8 / system
1-2G devices

Integrated Circuit:
≈8-16 / PCB
.25M-16M devices

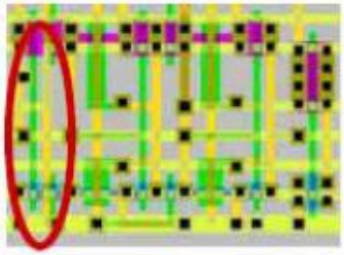
Module:
≈8-16 / IC
100K devices

MOSFET

Scheme for
representing
information

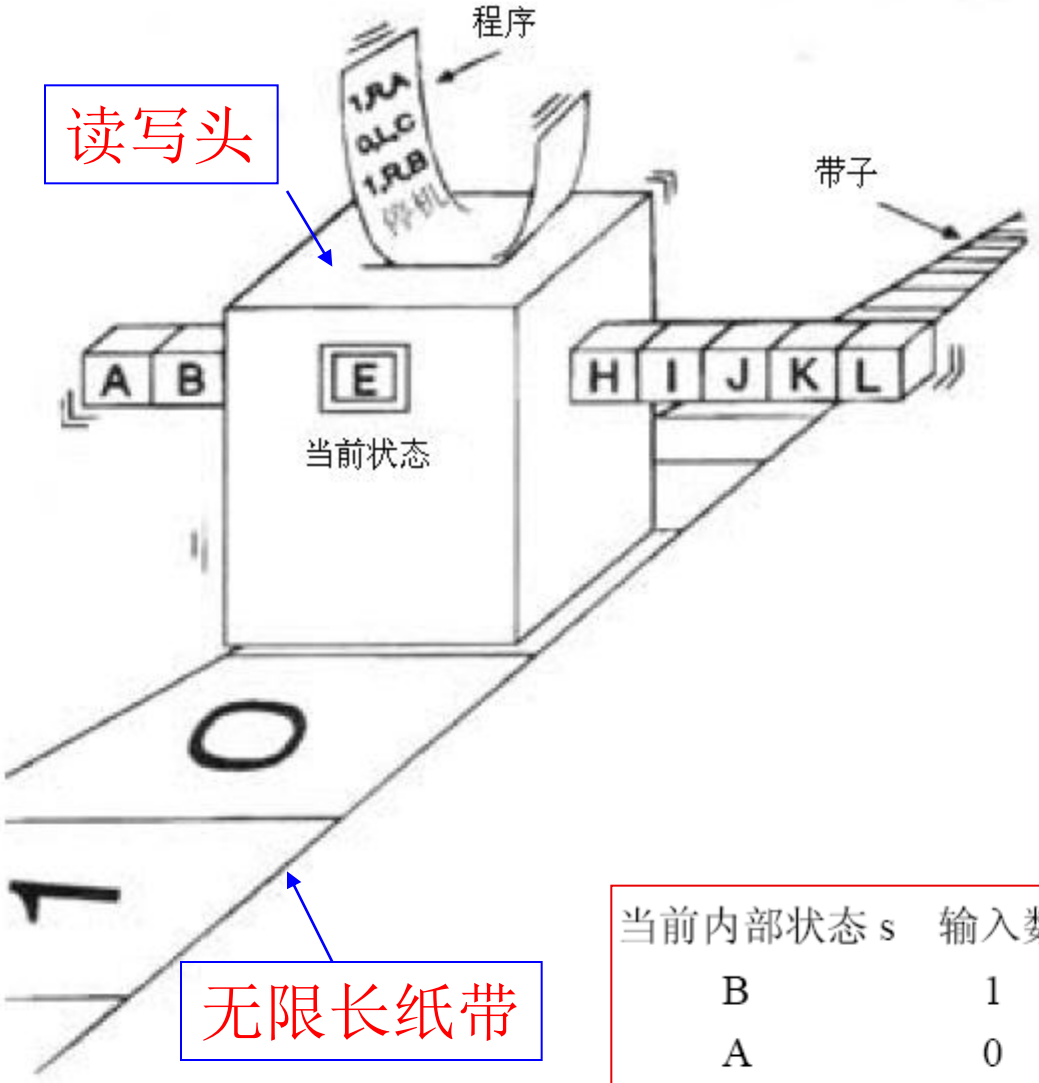


Gate:
≈2-16 / Cell
8 devices



Cell:
≈1K-10K / Module
16-64 devices

怎么去设计建造这样一个有 $1G=10^9$ 个部件组成的复杂系统呢?



图灵机模型

- 图灵机只要根据每一时刻读写头读到的一个方格的信息和当前的内部状态对程序进行查表，就可以确定下一时刻的内部状态和输出动作
- 具体的程序是一个列表，也叫做规则表

当前内部状态 s	输入数值 i	输出动作 o	下一时刻的内部状态 s'
B	1	前移	C
A	0	往纸带上写 1	B
C	0	后移	A
…	…	…	…

从控制器来理解计算机的通用性

- **UTM，Universal Turing Machine**
- 通用图灵模型

如何用一个通用电路实现多种计算任务？

处理器指令集的任务！

数据通路和控制的实现！

什么是处理器指令集？

- 处理器只是在块硅晶片上所集成的超大规模的集成电路，然而如此一颗精密的芯片为什么能够控制一个庞大而复杂的电脑系统呢？
  - 这就是处理器所集成的指令集
  - 所谓指令集，就是用来计算和控制计算机系统的一套命令的集合，而每一种新型的处理器在设计时就规定了一系列与硬件电路相配合的指令系统
  - 指令集的先进与否，也关系到处理器的性能发挥，它也是处理器性能体现的一个重要标志

计算机系统

硬件/软件接口

- 任何完整的计算机系统都是由硬件和软件两部分组成
- **ISA(Instruction Set Architecture，指令集体系结构)是计算机硬件与底层软件之间的接口，它是程序员为使一个二进制机器语言程序正确运行所需要了解的属性**

计算机软件

ISA

计算机硬件

机器语言

- 计算机硬件的基本功能就是执行指令，指令在冯·诺伊曼计算机中由二进制数字进行编码
- 描述完成一个确定任务的指令序列称为程序
- 计算机的全部二进制机器指令组成了一种可供人与计算机进行交流的语言，称为机器语言

```
00100000000010000000000000000001
00100001000010000000000000000010
10101100000010000001111101000000
```

机器语言程序

汇编语言

- 使用机器语言编写程序十分困难，于是人们发明了用助记符表示指令的方法。助记符形式的指令的集合组成了汇编语言
- 汇编语言的助记符形式的指令必须翻译成机器语言二进制指令才能在计算机上执行，实现这种翻译的程序称为汇编器**(assembler)**

```
addi $s1, $s0, 1
addi $s1, $s1, 2
sw $s1, 8000($s0)
```

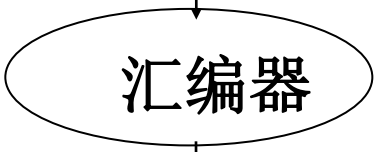
汇编语言源程序

汇编语言

C 程序：
s1=1+2

```
addi $s1, $s0, 1
addi $s1, $s1, 2
sw $s1, 8000($s0)
```

汇编语言源程序



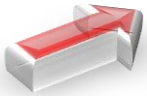

```
00100000000010000000000000000001
00100001000010000000000000000010
10101100000010000001111101000000
```

机器语言程序

高级语言

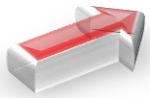
- 汇编语言与机器语言是一一对应的，所以开发效率仍然十分低下，于是人们发明了高级程序设计语言，如FORTRAN、C等等
- 使高级语言程序在只能运行二进制机器指令的计算机上运行，有两种途径：
  - 编译——将高级语言编写的程序翻译成等价的二进制指令序列来代替，计算机执行等价的机器语言程序
  - 解释——以高级语言程序作为输入数据，顺序地检查它的每一条语句，并直接执行等价的机器语言指令序列，这种方法称为解释，例如MATLAB

程序与指令系统



1. 基本概念
2. MIPS体系结构
3. 计算机性能评价
4. MIPS模拟器
5. MIPS汇编语言程序设计
6. 一个排序算法的实例

MIPS体系结构



1. 概述
2. MIPS指令格式
3. MIPS寻址方式
4. MIPS指令系统
5. MIPS过程调用

MIPS体系结构

- **RISC是相对于CISC的**
  - **RISC的杰出代表有ARM、POWERPC、MIPS**
  - **CISC的处理器大家天天用，intel＆amd的x86**

- MIPS:无内部互锁流水级的微处理器
  - ✓ref.2/Ch2.1~2.9
- Microprocessor without interlocked piped stages

对于一条汇编语言指令来说，有两个问题要解决

- 要指出进行什么操作
- 要指出大多数指令涉及的**操作数**和**操作结果**放在何处

MIPS：操作数

名称	实例	注释
32个寄存器	$s0, …$s7 $t0,…$t7	数据的快速定位，算术运算指令的操作数必须在寄存器中
2^{30}个存储字	存储器[1] … 存储器[2^{30}]	MIPS只能使用数据传送指令访问 MIPS中使用字寻址，相邻数据字的字地址相差**4**

常数操作数怎么办？

通用寄存器的习惯用法和命名

寄存器编号	助记符	用法
0	zero	永远为0
1	at	用做汇编器的临时变量
2-3	v0, v1	用于过程调用时返回结果
4-7	a0-a3	用于过程调用时传递参数
8-15	t0-t7	临时寄存器 在过程调用中被调用者不需要保存与恢复
24-25	t8-t9	
16-23	s0-s7	保存寄存器。在过程调用中被调用者一旦使用这些寄存器时，必须负责保存和恢复这些寄存器的原值
26,27	k0,k1	通常被中断或异常处理程序使用，用来保存一些系统参数
28	gp	全局指针global pointer。一些运行系统维护这个指针来更方便的存取静态（static）和外部（extern）变量
29	sp	堆栈指针stack pointer
30	fp	帧指针frame pointer
31	ra	返回地址

MIPS：操作数

- 寄存器为什么只有32个？
  - 因为电信号传输距离越远，传输时间就越长，寄存器太多将会延长时钟周期
- 存储器对齐限制：
  - MIPS中字的地址必须是4的倍数

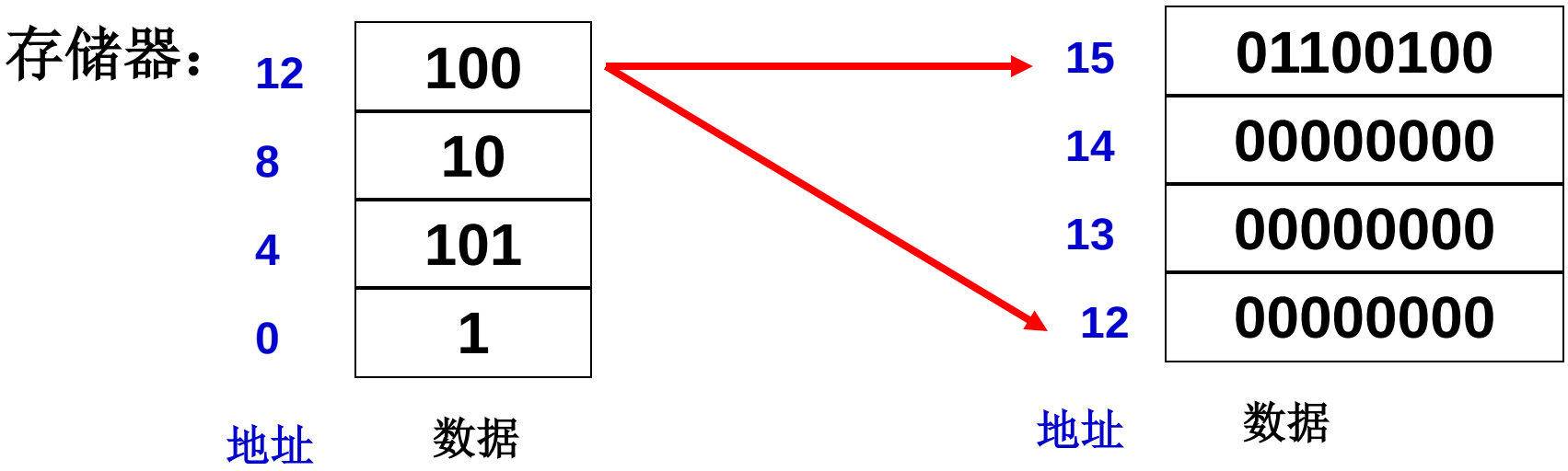
存储器：

地址	数据
12	100
8	10
4	101
0	1

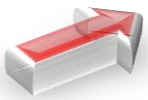
Q: MIPS是大端(Big-endian)的，还是小端的(Little-endian)？

A: 这是可以配置的

一般来说使用大端模式，和网络序相同



MIPS体系结构



1. 概述
2. MIPS指令格式
3. MIPS寻址方式
4. MIPS指令系统
5. MIPS过程调用

MIPS指令格式

所有的MIPS指令都是32位长。

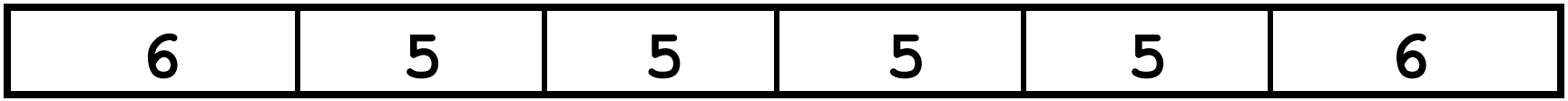
简单性来自规则性

- ✓R-format: 所有其他
- ✓I-format: 用于有立即数的指令,

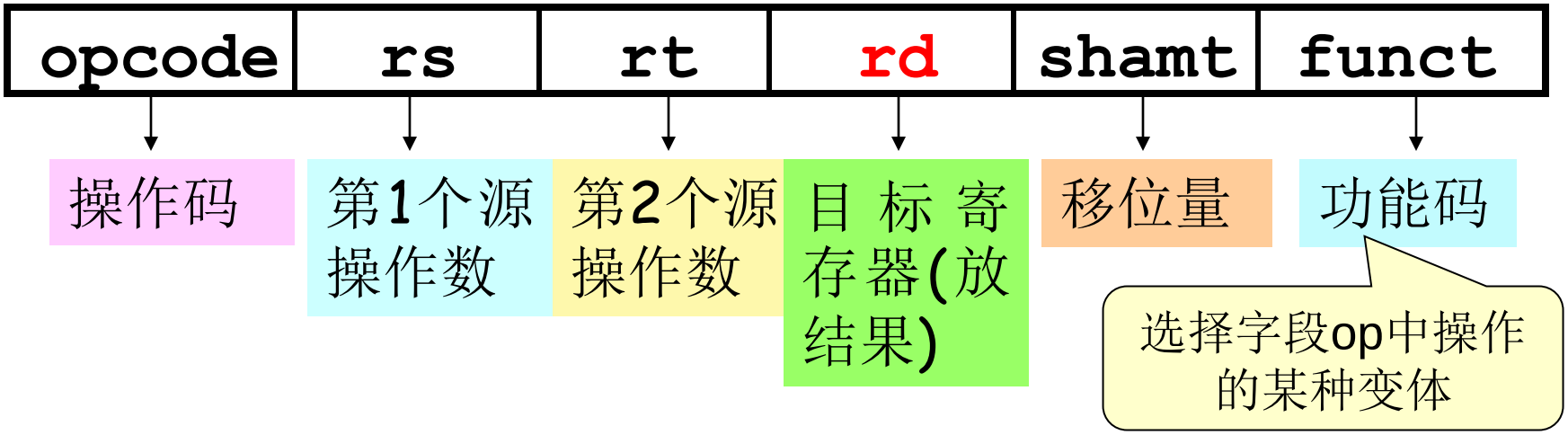
  lw, sw, beq, bne
- ✓J-format: 无条件跳转 j, 并连接jal

（1）R-型 指令

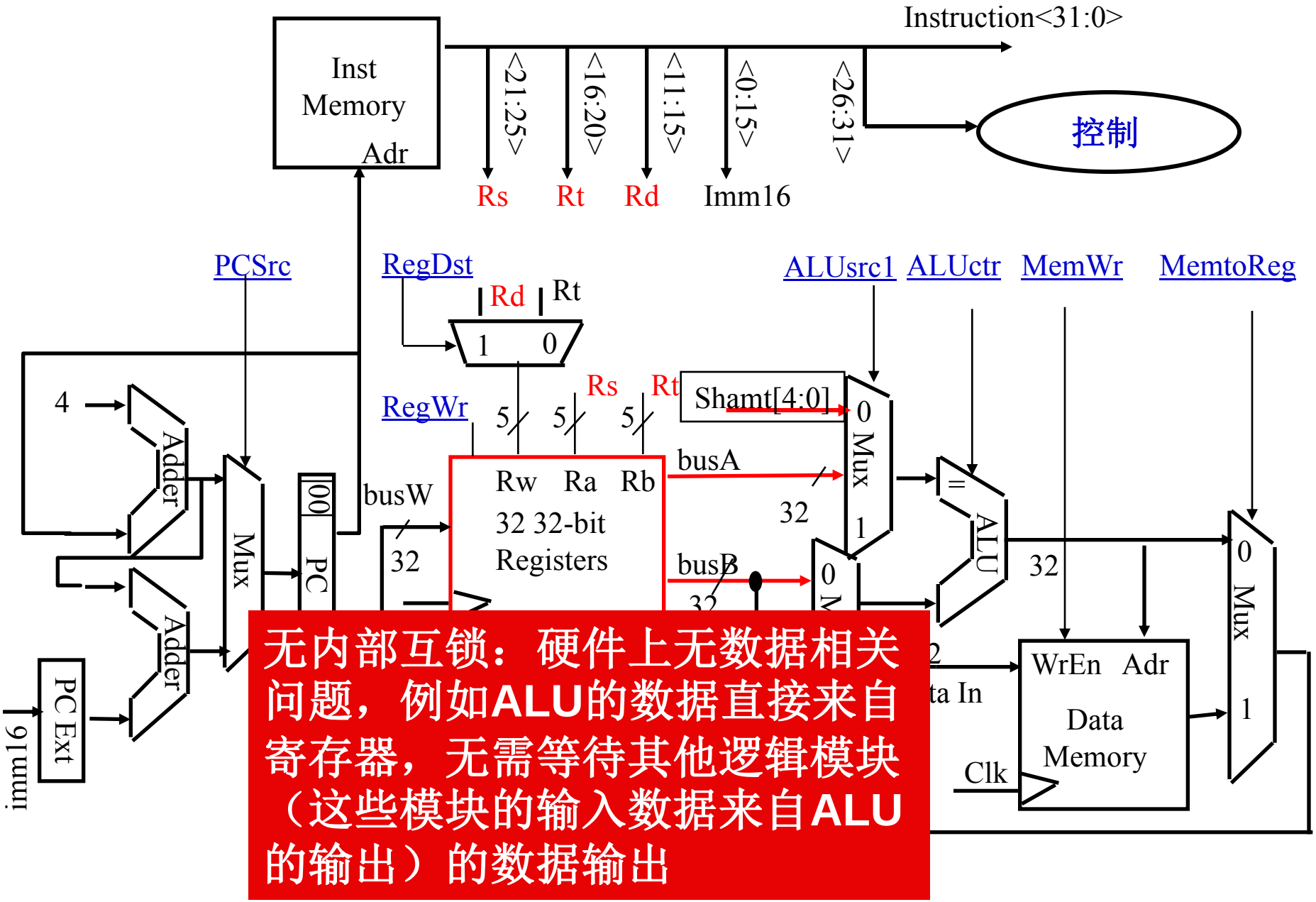
- 一条32位的MIPS R型指令分为6个字段：
  - 6 + 5 + 5 + 5 + 5 + 6 = 32bit



- 各段含义如下：



数据通路


Instruction<31:0>
Inst Memory
Adr
<21:25>
<16:20>
<11:15>
<0:15>
<26:31>
Rs
Rt
Rd
Imm16
控制
PCSrc
RegDst
Rd
Rt
1
0
RegWr
5
5
5
Rs
Rt
Shamt[4:0]
busA
busB
32
Rw
Ra
Rb
32 32-bit Registers
busW
32
4
Adder
Adder
Mux
00
PC
imm16
PC Ext
ALUsrc1
ALUctr
MemWr
MemtoReg
0
Mux
1
ALU
32
WrEn
Adr
Data Memory
Clk
ta In
2
无内部互锁：硬件上无数据相关问题，例如ALU的数据直接来自寄存器，无需等待其他逻辑模块（这些模块的输入数据来自ALU的输出）的数据输出

（1）R-型 指令举例

```
add    $8,$9,$10
```

十进制指令格式:

0	9	10	8	0	32

二进制指令格式:

000000	01001	01010	01000	00000	100000

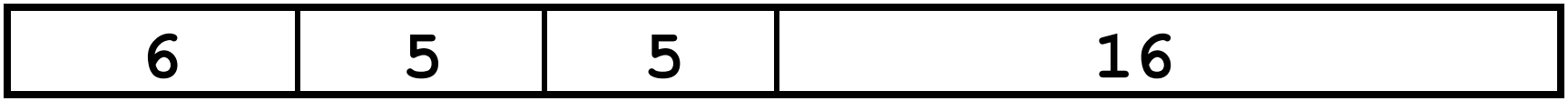
hex

16进制: $012A\ 4020_{hex}$

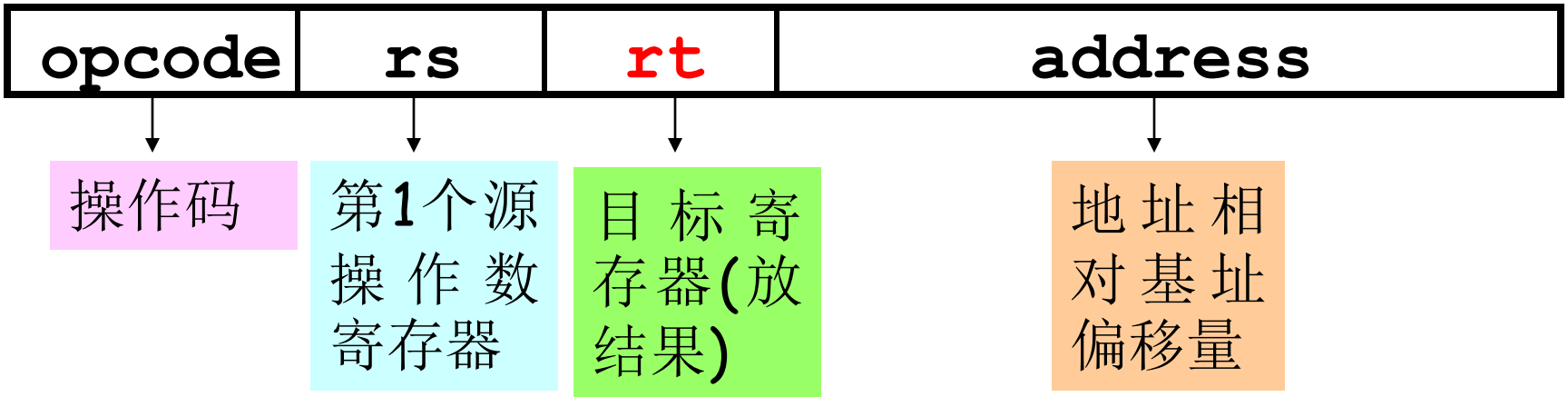
10进制: $19,546,144_{ten}$

（2）I-型 指令

- 一条32位的MIPS I型指令划分为4个字段：
  6 + 5 + 5 + 1 6 = 32bit



- 各段含义如下：



装入指令lw、存储指令sw、分支指令和立即数运算指令

（2）I-型 指令

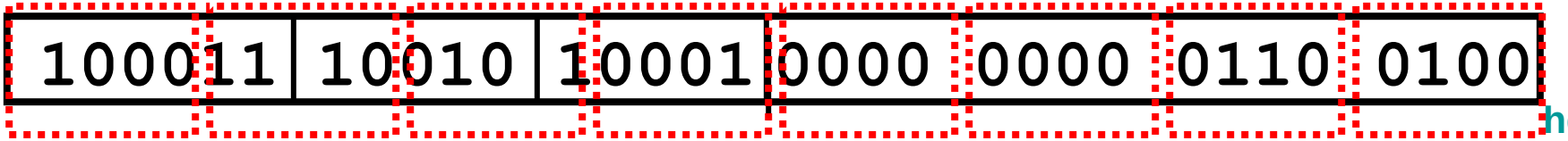
- 装入指令lw、存储指令sw的格式
  - 数据装入lw：Rt = Mem[Rs + Address]
  - 数据存储sw：Mem[Rs + Address] = Rt
- 只有装入lw（从内存中读出数据）和存储sw （写入内存中）指令实现对内存的访问
- 存储器寻址方式： c($rx) 寄存器rx的地址值+立即数c

符号扩展

例1：装入/存储指令

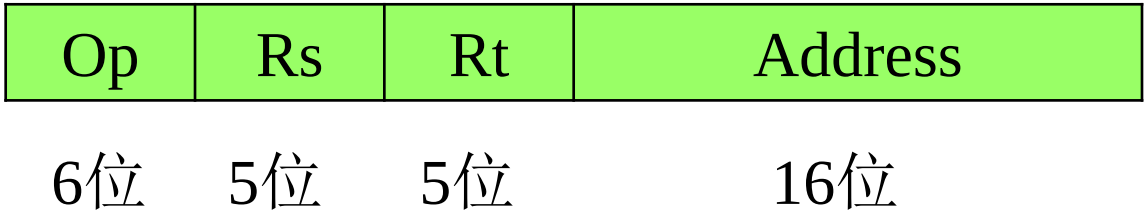
lw $s1, 100($s2)

$8E510064_{hex}$



（2）I-型 指令

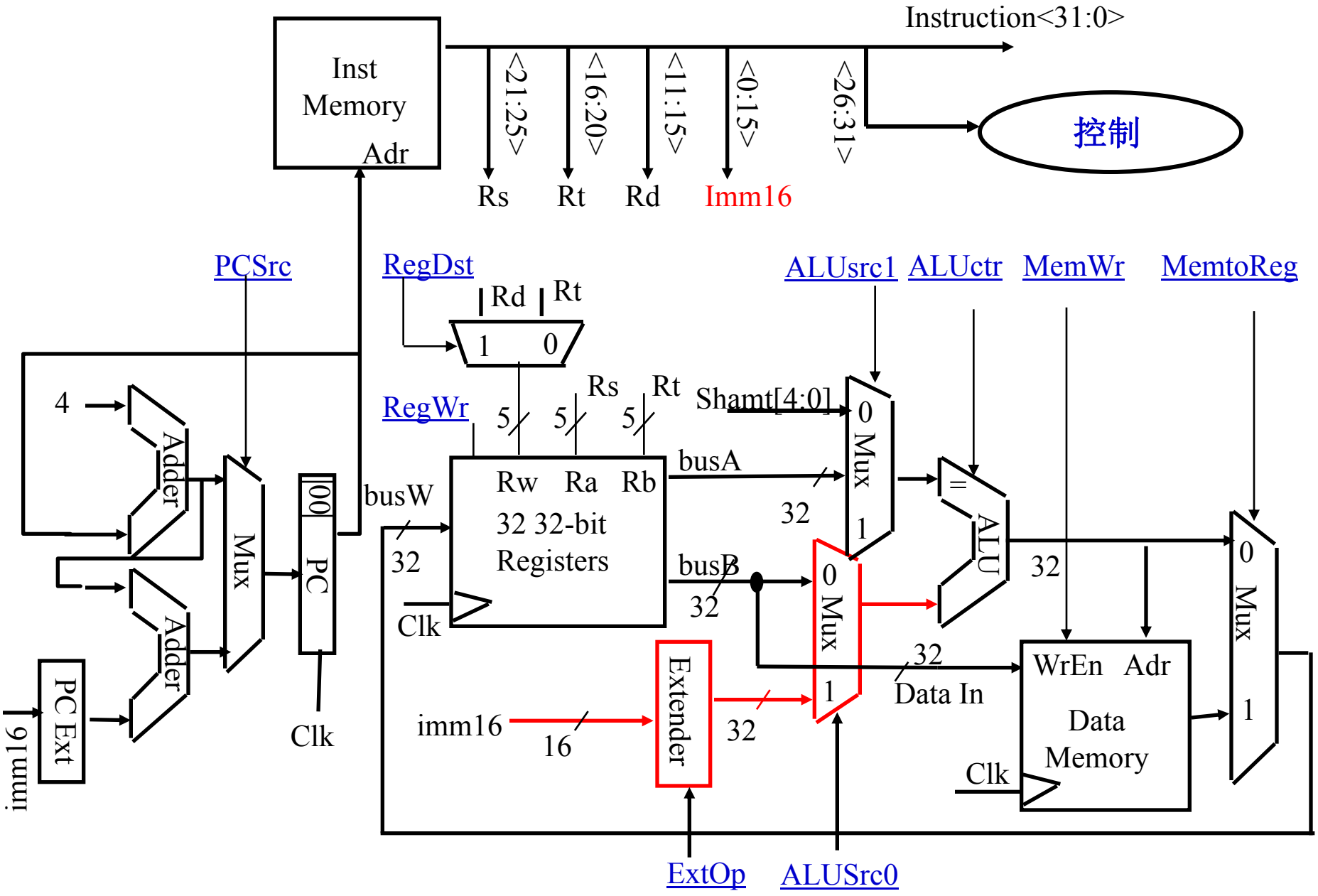
- 例2：分支指令
  - if(Rs <relation> Rt) goto (PC+4)+Address



beq $t0, $t1, Target

- 分支指令采用的寻址方式为**PC相对寻址**——分支目标的地址是**PC+4**与指令中的偏移量之和

数据通路


Instruction<31:0>
Inst
Memory
Adr
<21:25>
<16:20>
<11:15>
<0:15>
<26:31>
控制
Rs
Rt
Rd
Imm16
PCSrc
RegDst
ALUsrc1
ALUctr
MemWr
MemtoReg
Rd
Rt
1
0
4
Adder
Adder
Mux
PC
00
Clk
RegWr
5
5
5
Rs
Rt
busW
32
Rw
Ra
Rb
32 32-bit
Registers
Clk
busA
32
Shamt[4:0]
0
Mux
1
busB
32
0
Mux
1
=
ALU
32
0
Mux
1
32
Data In
WrEn
Adr
Data
Memory
Clk
imm16
PC Ext
imm16
16
Extender
32
ExtOp
ALUSrc0

（2）I-型 指令举例

例3： addi $21,$22,-50

opcode = 8 (具体是什么操作)

rs = 22 (操作数寄存器)

rt = 21 (目的寄存器)

immediate = -50 (默认是十进制数)

请将指令翻译成机器码

十进制指令格式： 十进制表示： $584,449,998_{ten}$

8	22	21	-50

二进制指令格式： 十六进制表示： 22D5 FFCE$_{hex}$

001000	10110	10101	1111111111001110

（3）J-型 指令

- J型指令格式:

6 bits	26 bits

- 每部分的名字是:

opcode	target address

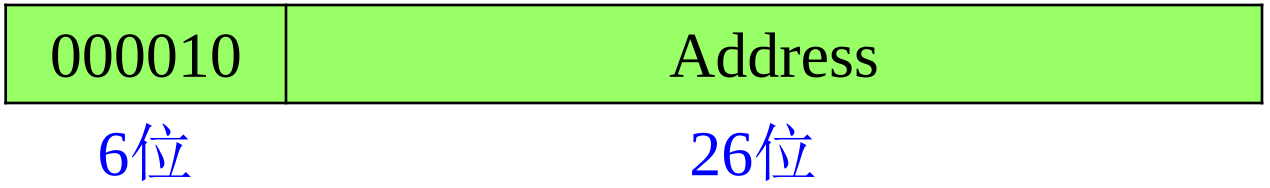
- 关键点:

✓ 必须让 opcode 部分与R和I型指令一致，以便于电路理解执行

✓ 其他字段都节省出来给跳转的目的地址用以表示很大的跳转范围

（3）J-型 指令

- 例：跳转指令的格式



– 跳转指令采用伪直接寻址——跳转地址为指令中的26位常数与PC（program counter）中的高位拼接得到

例： **j 10000** **08002710_{hex}**

0000 1000 0000 0000 0010 0111 0001 0000

（3）J-型 指令

- 总之:
  - 新的PC = { PC[31..28], 目标地址, 00 }
- 一定要明白上面的每一项是从哪里来的!
- Note: { , , } 表示合并
  { 4 bits , 26 bits , 2 bits } = 32 bit address
  - { 1010, 11111111111111111111111111, 00 }
    = 10101111111111111111111111111100
  - Note: 书上用 ||, Verilog里用{ , , }

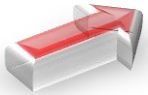
？关于MIPS指令格式的思考

- 指令格式为什么要分三类呢？
  - 所有指令长度相同，都是32位
  - 要让每一条指令刚好合适，充分发挥作用
  - ……

  提示：和ALU及指令译码电路设计有关

MIPS体系结构

1. 概述
2. MIPS指令格式
3. MIPS寻址方式
4. MIPS指令系统
5. MIPS过程调用



MIPS寻址方式

- ✓ 寄存器寻址
- ✓ 立即数寻址
- ✓ 基址或偏移寻址
- ✓ PC相对寻址
- ✓ 伪直接寻址

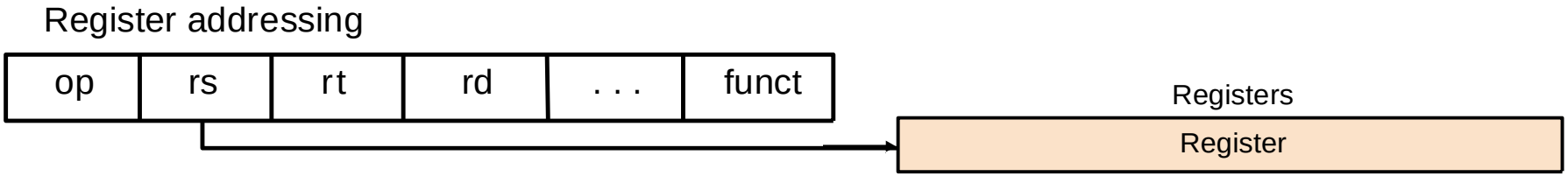
指令

MIPS 寻址方式

（1）寄存器寻址

例：MIPS算术运算指令的操作数必须从32个32位寄存器中选取

```
add $ t0, $s1, $s2   # 寄存器$ t0的内容为 g+h
add $ t1, $s3, $s4   # 寄存器$ t1的内容为 i+j
sub $ s0, $t0, $t1   # 寄存器$ s0的内容为 (g+h)-(i+j)
```



MIPS 寻址方式

（2）立即数寻址

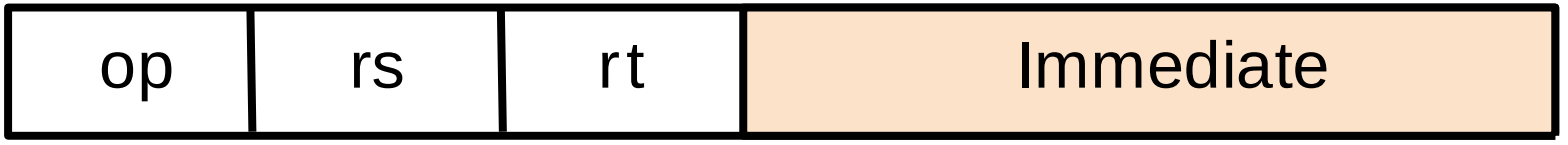
加快对常见情况的处理

- 以常数作为操作数，无须访问存储器就可以使用常数
- 因为常数操作数频繁出现，所以在算术指令中加入常数字段，比从存储器中读取常数快得多

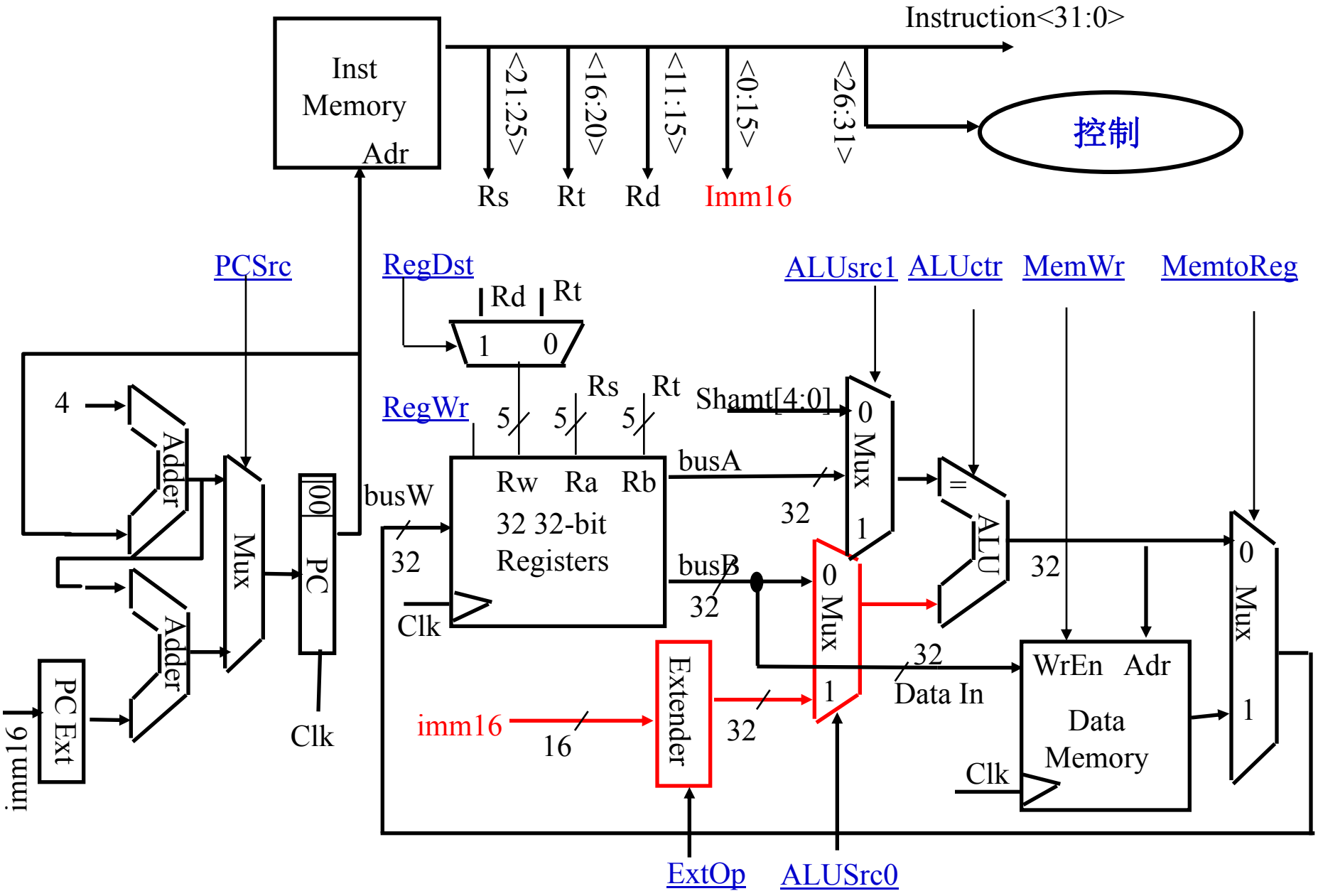
```
Lw $ t0, AddrConstant4($ zero)  # $ t0 = 常数4
add $ sp, $sp, $t0              # $ sp = $ sp + $t0 ( $ t0 == 4)
```

```
addi $ sp, $sp, 4               # $ sp =$  sp + 4
```

Immediate addressing



数据通路


Instruction<31:0>
Inst Memory
Adr
<21:25>
<16:20>
<11:15>
<0:15>
<26:31>
Rs
Rt
Rd
Imm16
控制
PCSrc
RegDst
ALUsrc1
ALUctr
MemWr
MemtoReg
Rd
Rt
1
0
RegWr
5
5
5
Rs
Rt
Shamt[4:0]
busA
busW
32
Rw
Ra
Rb
32 32-bit Registers
busB
32
Clk
4
Adder
Adder
Mux
PC
00
Clk
PC Ext
imm16
0
Mux
1
ALU
=
32
0
Mux
1
32
Data In
WrEn
Adr
Data Memory
Clk
0
Mux
1
Extender
imm16
16
32
ExtOp
ALUSrc0

MIPS 寻址方式

（2）立即数寻址

- 怎样把一个32位长的常数装入寄存器$ s0中？

0000 0000 0011 1101	0000 1001 0000 0000
61	2304

Load Upper Immediate

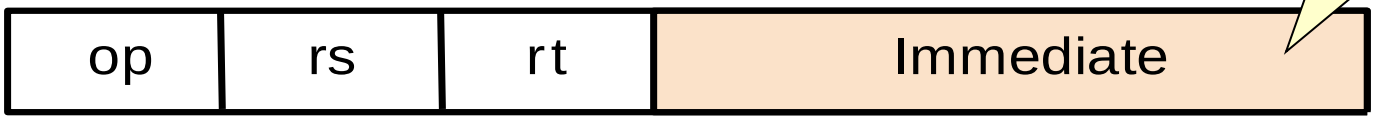
```
Lui  $ s0, 61      # $ s0= 0000 0000 0011 1101 0000 0000 0000 0000
addi $ s0, $s0, 2340  # $ s0= 0000 0000 0011 1101 0000 1001 0000 0000
```

拆散和重装大常数还可以这样完成：

```
ori $s5,$0,0x1000
sll $s5,$s5,16      #$s5=0x10000000
```

有长度限制，addi为16位

Immediate addressing

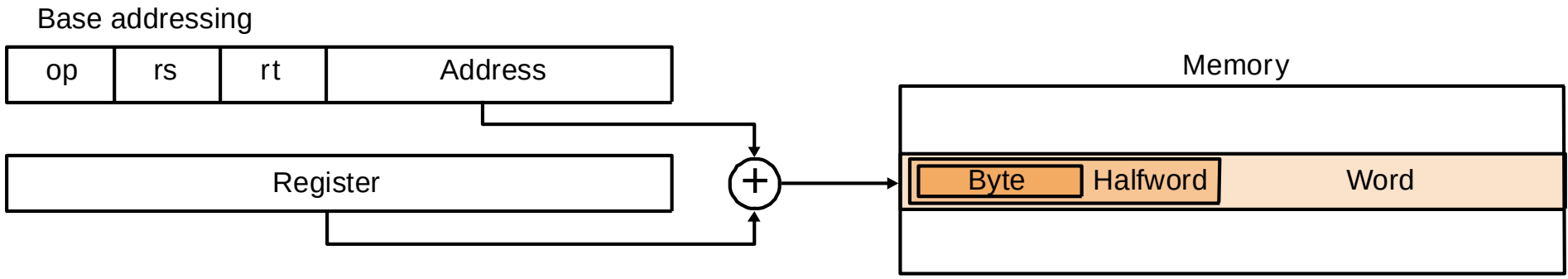


MIPS 寻址方式

（3）基址或偏移寻址

- 操作数在存储器中，且存储器地址是某寄存器与指令中某常量的和

Lw $ t0, 8 ($ s0) **#$ s0中装的是存储器中的地址**



- 只有装入lw和存储sw指令实现对内存的访问
- 存储器寻址方式： c(rx) 立即数c + 寄存器rx的值

MIPS 寻址方式

（4）PC相对寻址

例：条件分支指令

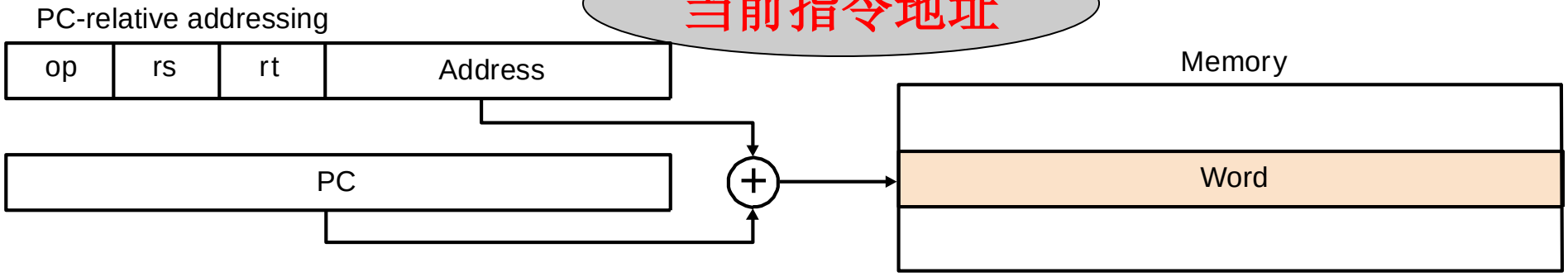
bne $ s0, $ s1, Exit　#如果$ s0不等于$s1，则跳转到Exit

5	16	17	Exit
6位	5位	5位	16位

±2^{15}个字范围内的地址

程序计数器=PC寄存器+分支地址

当前指令地址



MIPS 寻址方式

（4）PC相对寻址

问题1：为什么选PC寄存器？

因为几乎所有的条件分支指令都是跳转到附近的地址

问题2：如何处理16位无法表达的远距离分支？

插入一个无条件跳转到分支目标地址的指令，把分支指令中的条件变反以决定是否跳过该指令

```
Bne $s0, $s1, L2
```

```
      beq   $s0, $s1, L1
      j     L2
L1:
      ……
L2:
      ……
```

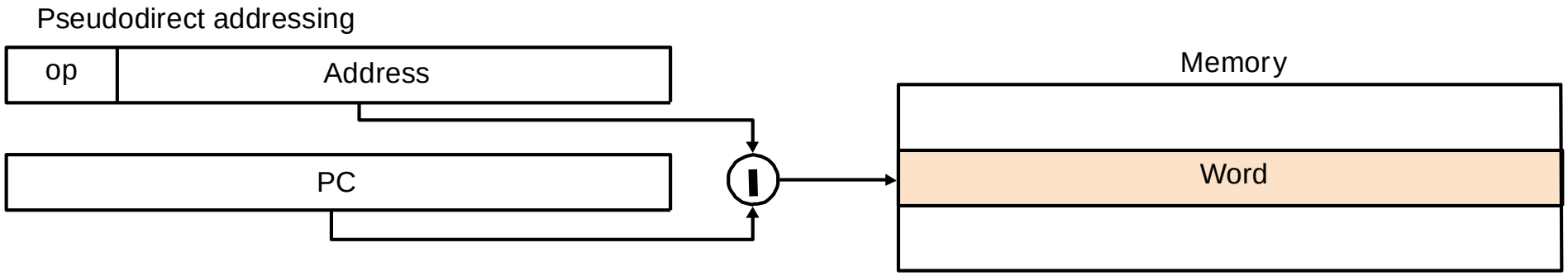
$2^{15} < |L2-PC| < 2^{27}$

MIPS 寻址方式

（5）伪直接寻址

跳转地址= PC中原高4位 | 指令中的26位 | 00

j 10000　　#跳转到PC | 10000

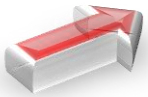


问题1： PC高位限制了跳转的范围，如何跳出高4位限制的范围？

问题2： 目的地址比当前的PC地址小怎么办？

MIPS体系结构

1. 概述
2. MIPS指令格式
3. MIPS寻址方式
4. MIPS指令系统
5. MIPS过程调用



MIPS 指令系统

（1）数据处理类（算术/逻辑运算指令）

（2）数据传送类（装入/存储指令）

（3）控制类（分支与跳转指令）

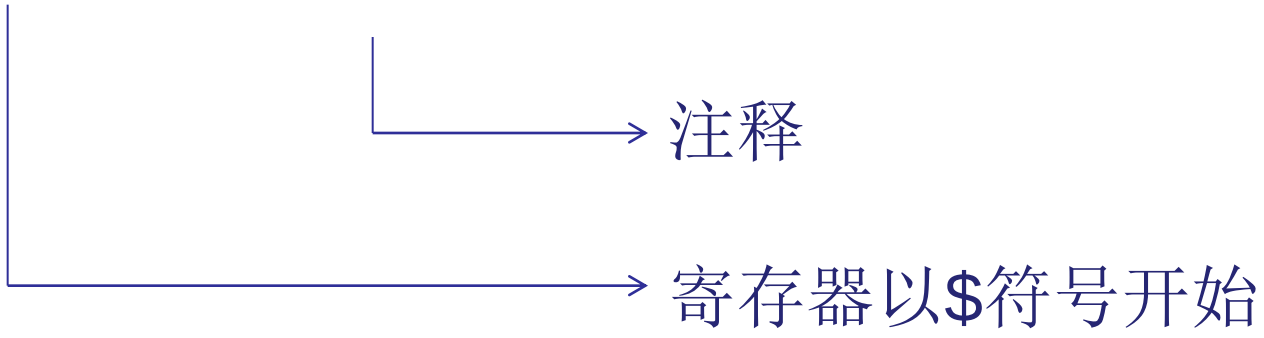
算术运算指令

- 加法

  add $t0,$t1,$t2　　# $t0 = $t1 + $t2

- 减法

  sub $t2,$t3,$t4　　# $t2 = $t3 - $t4



算术运算指令

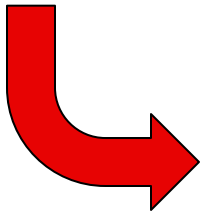
- 编译C语言的赋值语句

  f = (g + h) – (i + j)

```
add t1, g, h
add t2, i, j
sub f, t1, t2
```

假设变量f、g、h、i、j分别分配给寄存器$s0~$s4

临时变量t1、t2分别分配给寄存器$t1、$t2



```
add $t1, $s1, $s2
add $t2, $s3, $s4
sub $s0, $t1, $t2
```

算术运算指令

- 加16位有符号立即数

```
addi $t2,$t3, 5      # $t2 = $t3 + 5
```

- MIPS没有减立即数的指令，如何实现减立即数操作？

逻辑运算指令

```
and $t0,$t1,$t2          # $t0 = $t1 & $t2
or    $t0,$t1,$t2        # $t0 = $t1 | $t2
xor  $t0,$t1,$t2         # $t0 = $t1 ⊕ $t2
nor  $t0,$t1,$t2         # $t0 = ~($t1 | $t2)
```

– 如何实现NOT运算？

- 立即数逻辑运算

```
andi $t0,$t1,10
ori $t0,$t1,10
xori $t0,$t1,10
```

逻辑运算指令

- 移位运算，移位量为立即数

  逻辑左移——

  sll $t0, $t1, 10　　# $t0 = $t1 << 10，　shift left logical

  逻辑右移——

  srl $t0, $t1, 10　　# $t0 = $t1 >> 10，　shift right logical

  算术右移——

  sra $t0, $t1, 10　　# $t0 = $t1 >> 10，　shift right arithm

- 移位运算，移位量在某个寄存器中

  sllv $t0, $t1, $t3　　# $t0 = $t1 << ($t3%32)

  srlv $t0, $t1, $t3　　# $t0 = $t1 >> ($t3%32)

  srav $t0, $t1, $t3　　# $t0 = $t1 >> ($t3%32)

比较指令

- 比较两个寄存器的内容，并根据比较的结果设置第三个寄存器

```
slt $t1,$t2,$t3          # if ($t2 < $t3) $t1=1;
                         # else $t1=0
sltu $t1,$t2,$t3         # 无符号比较
```

- 寄存器与立即数比较

```
slti $t1,$t2,10          # 与立即数比较
sltui $t1,$t2,10         # 与无符号立即数比较
```

MIPS 指令系统

（1）数据处理类（算术/逻辑运算指令）

（2）数据传送类（装入/存储指令）

（3）控制类（分支与跳转指令）

装入/存储指令

- 实现内存与寄存器之间的数据传送

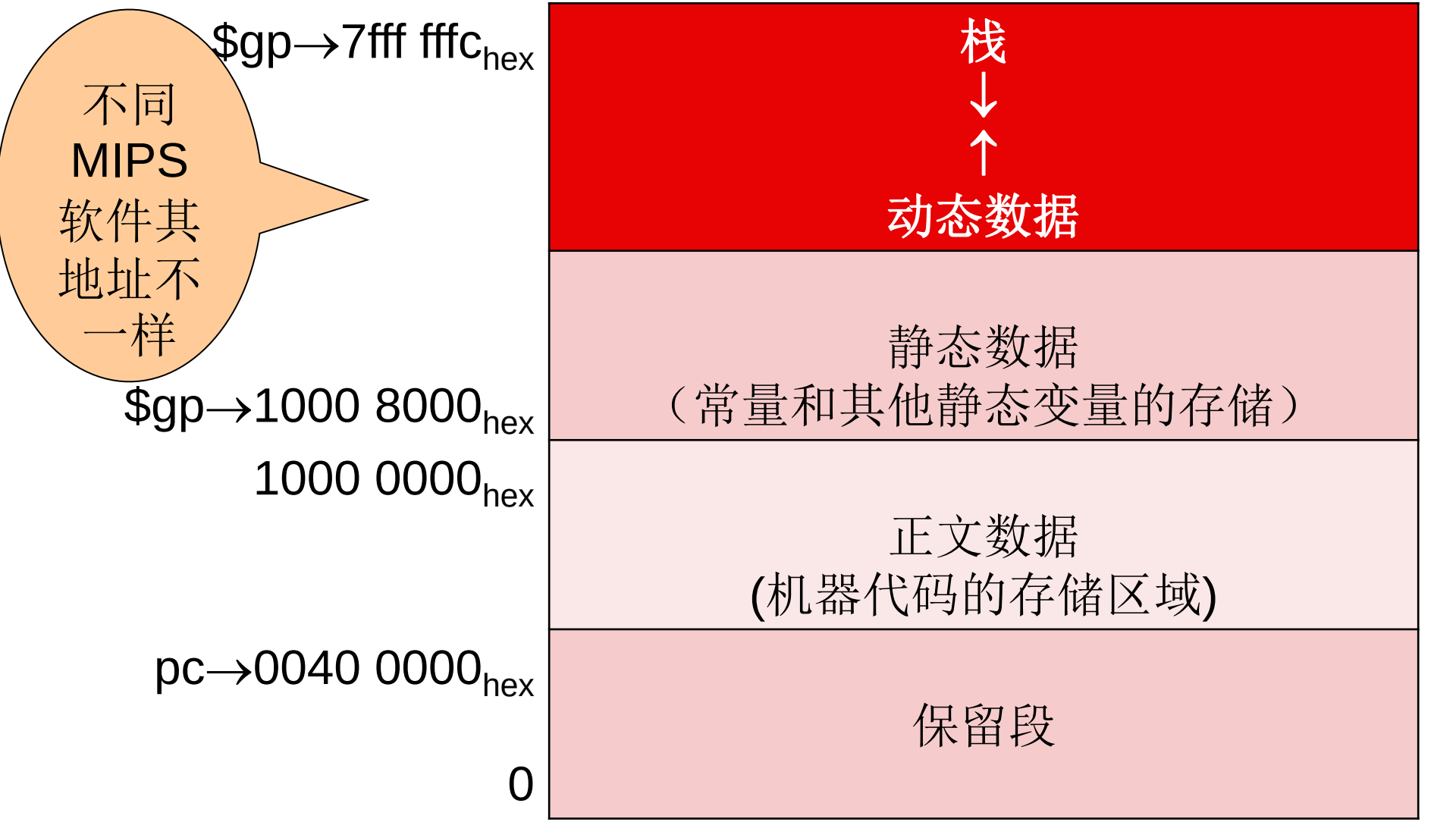
```
lw $t1, 32($t2)          #Load word
sw  $t3,  500($t4)       #Store word
```

- **MIPS只支持基址(变址)+位移量的内存寻址方式**
- 存放基址的寄存器称为基址寄存器，指令中的常量称为位移量

栈操作

- 虽然MIPS有32个通用寄存器，但是在某些情况下(例如子程序调用)仍然需要将寄存器的内容换出到内存中，在这种情形下，栈是保存寄存器内容的理想场所
- MIPS有一个$sp寄存器可以用做栈指针
- 将数据放入栈中称为压栈(PUSH)，从栈中取出数据称为出栈(POP)

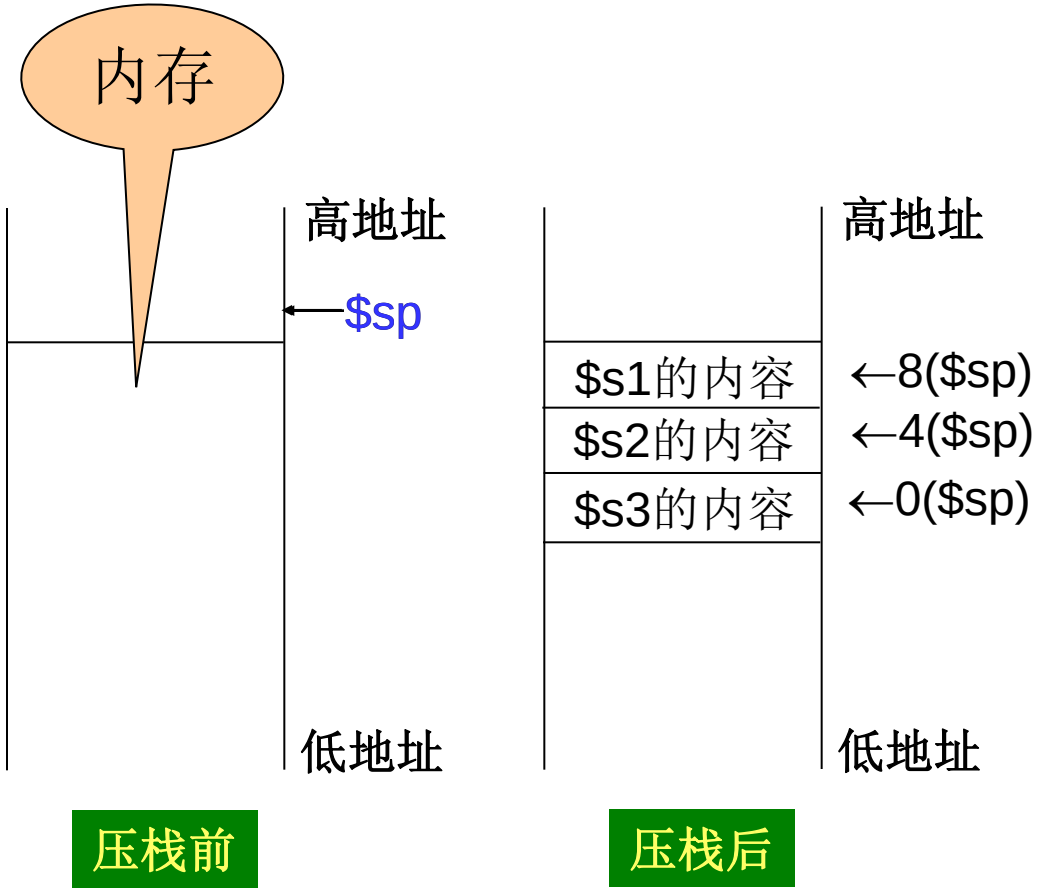
MIPS内存分配约定


不同MIPS软件其地址不一样
$gp→7fff fffc$_{hex}$
栈
↓
↑
动态数据
静态数据
（常量和其他静态变量的存储）
$gp→1000 8000$_{hex}
1000 0000$_{hex}$
正文数据
(机器代码的存储区域)
pc→0040 0000$_{hex}$
保留段
0

栈操作

- 例，将$s1、 $s2、$s3寄存器的内容压入栈

```
addi $sp, $sp, -12
sw   $s1, 8($sp)
sw   $s2, 4($sp)
sw   $s3, 0($sp)
```

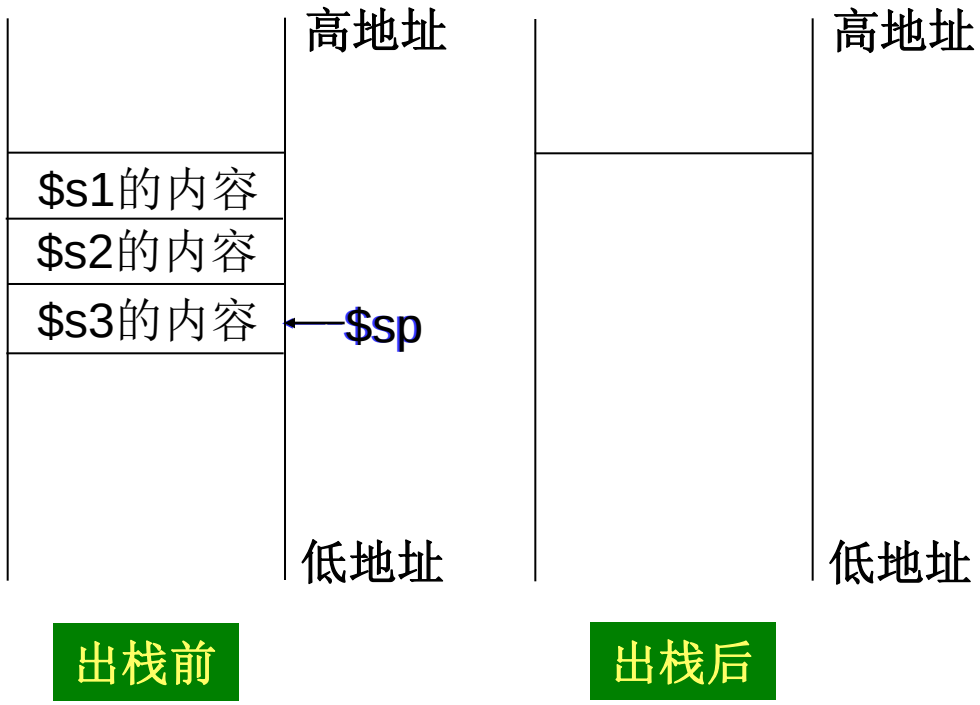


习惯上，栈按照从高到到低的地址顺序增长

栈操作

- 出栈操作

```
lw    $s1, 8($sp)
lw    $s2, 4($sp)
lw    $s3, 0($sp)
addi $sp, $sp, 12
```



装入高位立即数

- 装入高位立即数**(load upper immediate)**

  lui $t1, 30

- 例，将**32**位立即数**0x1234abcd**装入**$t1**寄存器

  lui $t1, 0x1234

  ori $t1, $t1, 0xabcd

MIPS 指令系统

（1）数据处理类（算术/逻辑运算指令）

（2）数据传送类（装入/存储指令）

（3）控制类（分支与跳转指令）

分支(branch)指令

```
beq $t0, $t1, Target     #如果$t0=$t1，则分支执
                         #行标号为Target的指令
bne $t0, $t1, Target     #如果$t0!=$t1，则分支执
                         #行标号为Target的指令
```

- 分支指令与比较指令相结合，可以实现各种条件分支：相等、不等、小于、小于或等于、大于、大于或等于

跳转(jump)指令

- 无条件分支

  j Lable　#无条件跳转到标号Lable处

分支指令的应用

- 例

```
unsigned int i, j;
…
If(i <= j)
      f = g + h;
else
      f = g - h;
```

假设变量f、g、h 、i、j分配给寄存器$s0~$s4

- 汇编代码

```
          sltu $t2, $s4, $s3
          beq $t2, $zero, Else
          sub $s0, $s1, $s2
          j      Exit
Else:  add $s0, $s1, $s2
Exit:  …
```

分支指令的应用

- 例

```
while(save[i] == k)
    i += 1;
```

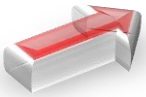
假设变量i、k对应寄存器$s3和$s5，数组save的基址存放在$s6中

- 汇编代码

```
Loop：  sll   $t1, $s3, 2
        add   $t1, $t1, $s6
        lw    $t0, 0($t1)
        bne   $t0, $s5, Exit
        addi  $s3, $s3, 1
        j     Loop
Exit:   …
```

MIPS体系结构

1. 概述
2. MIPS指令格式
3. MIPS寻址方式
4. MIPS指令系统
5. MIPS过程调用



过程调用

- **MIPS的过程调用遵循如下约定：**
  - 通过**$a0～$a3**四个参数寄存器传递参数
  - 通过**$v0～$v1**两个返回值寄存器传递返回值
  - 通过**$ra**寄存器保存返回地址
- 子程序调用通过跳转与链接指令**jal**进行

```
jal Procedure# 将返回地址保存在$ra寄存器中
              # 程序跳转到过程Procedure处执行
```

- 子程序返回通过寄存器跳转指令**jr**进行

```
jr $ra          # 跳转到寄存器指定的地址
```

叶过程

不调用其他过程的过程

```
int  leaf_example (int g, int h, int i, int j)
{
    int f ;
    f = (g + h) – (i + j);
    return f;
}
```

```
leaf_example:
    addi    $sp,  $sp,  -12
    sw      $t1, 8($sp)
    sw      $t0, 4($sp)
    sw      $s0, 0($sp)
    add     $t0, $a0, $a1
    add     $t1, $a2, $a3
    sub     $s0, $t0, $t1
    add     $v0, $s0, $zero
    lw      $s0, 0($sp)
    lw      $t0, 4($sp)
    lw      $t1, 8($sp)
    addi    $sp, $sp, 12
    jr      $ra
```

跳转到ra寄存器保存的返回地址

叶过程

```
int  leaf_example (int g, int h, int i, int j)
{
    int f ;
    f = (g + h) – (i + j);
    return f;
}
```

临时寄存器不必保存
保存寄存器必须保存

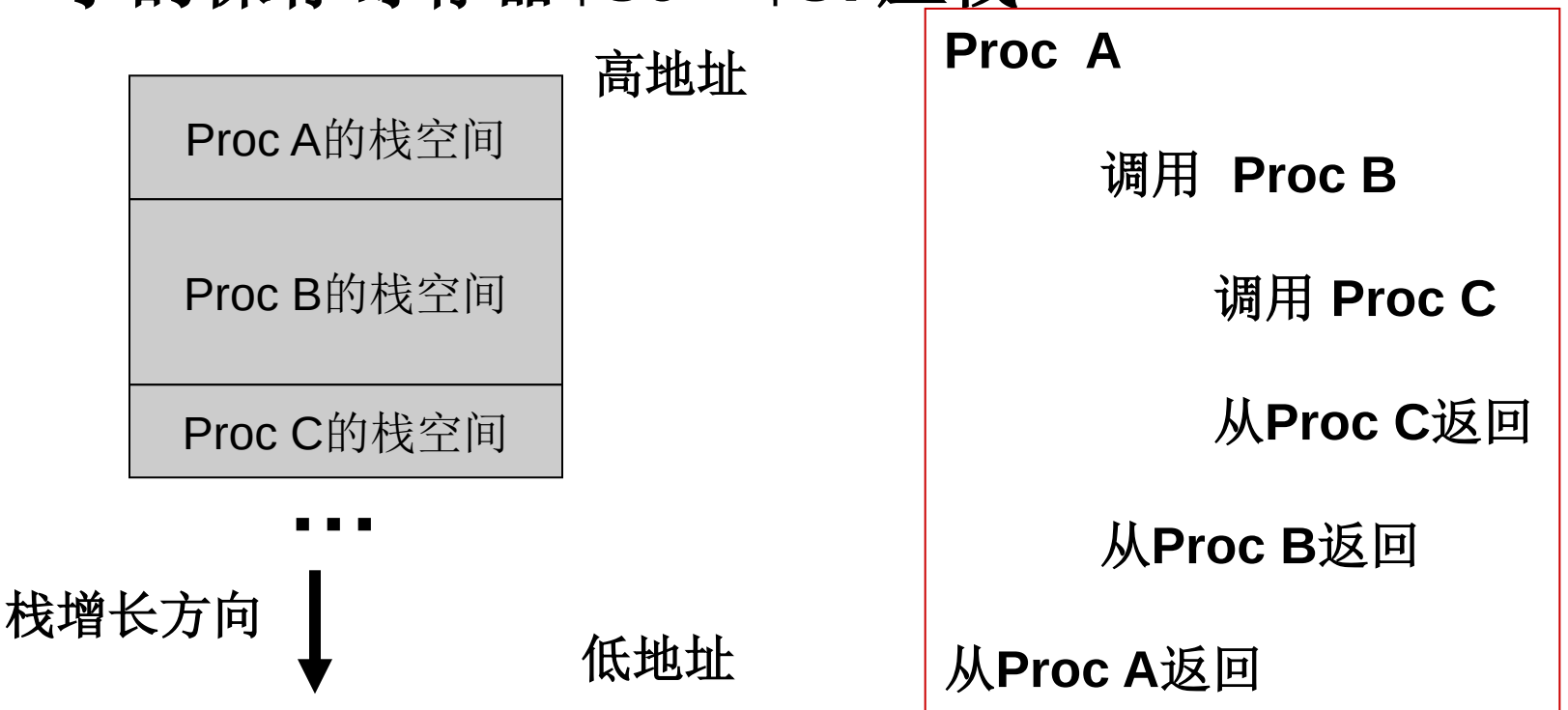
```
leaf_example:
  addi    $sp,  $sp,  -4
  sw      $t1, 8($sp)
  sw      $t0, 4($sp)
  sw      $s0, 0($sp)
  add     $t0, $a0, $a1
  add     $t1, $a2, $a3
  sub     $s0, $t0, $t1
  add     $v0, $s0, $zero
  lw      $s0, 0($sp)
  lw      $t0, 4($sp)
  lw      $t1, 8($sp)
  addi    $sp, $sp, 4
  jr      $ra
```

嵌套过程

- 如果只有叶过程，情况非常简单，但现实是残酷的
- 假设主程序调用过程A，参数为3时，将3存入$a0中，jal A
- 再假设过程A调用过程B，参数为7时，同样存入$a0，导致$a0使用上的冲突

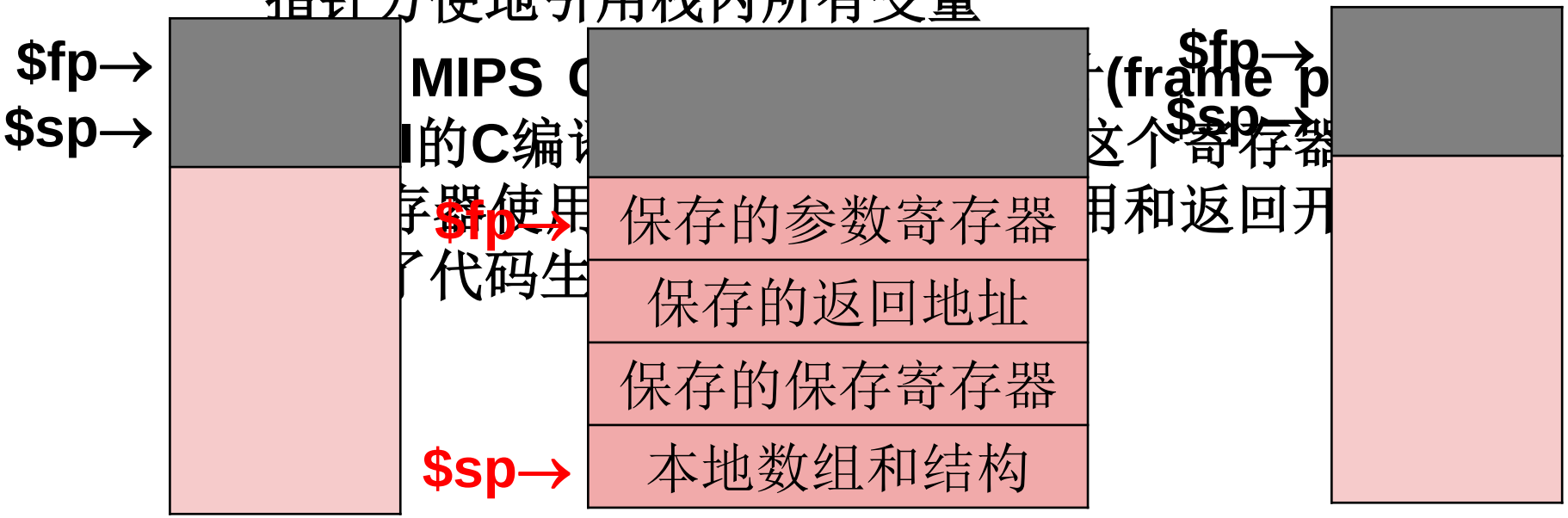
嵌套过程调用

- 主调过程将调用后还需要使用的参数寄存器**$a0～$a3**和临时寄存器**$t0～$t9**压栈
- 被调过程将返回地址寄存器和在被调过程中修改了的保存寄存器**$s0～$s7**压栈



帧指针fp

- 过程帧：栈中包含过程保存的寄存器和局部变量的段
- 帧指针：指向过程帧的第一个字
  - 在过程中提供用于局部存储器引用的稳定的基寄存器
  - 过程中栈指针可以修改，但帧指针不变，可通过帧指针方便地引用栈内所有变量

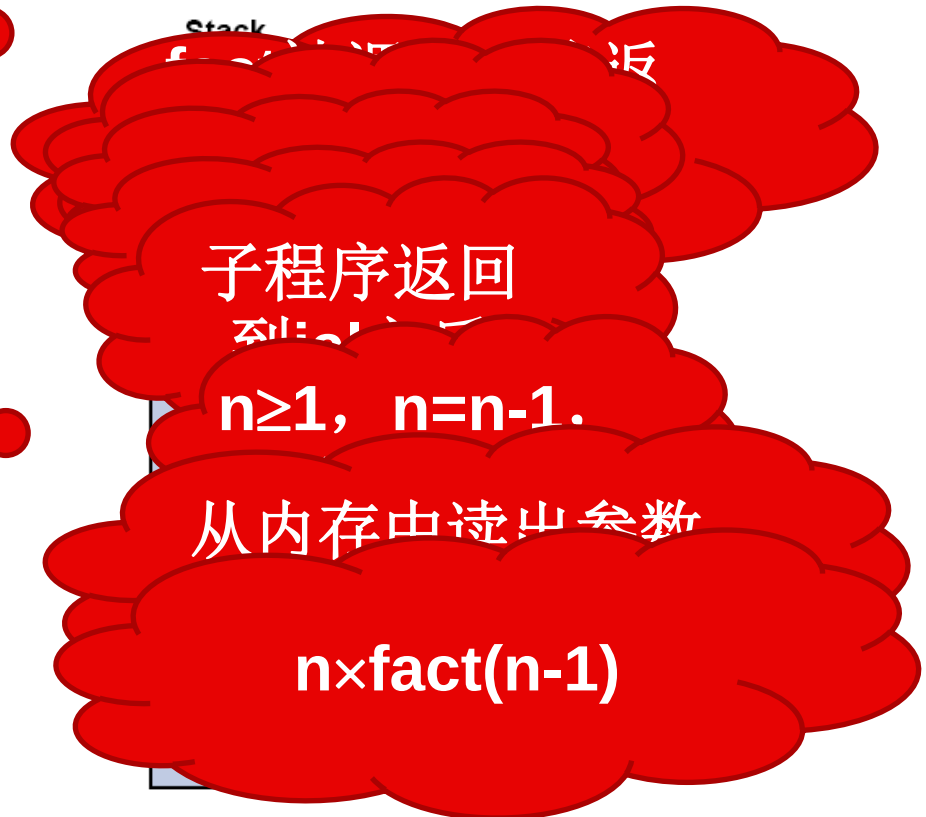


嵌套过程调用

- 计算n!

```
fact:
        addi    $sp, $sp, -8
        sw      $ra, 4($sp)
        sw      $a0, 0($sp)
        slti    $t0, $a0, 1
        beq     $t0, $zero, L1
        addi    $v0, $zero, 1
        addi    $sp, $sp, 8
        jr      $ra
L1:
        addi    $a0, $a0, -1
        jal     fact
        lw      $a0, 0($sp)
        lw      $ra, 4($sp)
        addi    $sp, $sp, 8
        mul     $v0, $a0, $v0
        jr      $ra
```

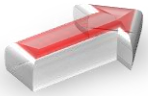
```
int fact(int n)
{
    if (n < 1)  return 1;
    else return (n * fact(n-1));
}
```



- 框架指针$fp指向框架的第一个字，通常是保存的参数寄存器
- 栈指针$sp指向栈顶，在程序执行的过程中栈指针有可能改变
- 因此通过固定的框架指针来访问变量要比用栈指针更简便
- 如果一个过程的栈中没有局部变量，编译器将不设置和恢复框架指针，以节省时间
- 当需要框架指针时，以调用时的$sp值作为框架指针的初值，调用返回时，根据$fp恢复$sp值

程序与指令系统

1. 基本概念
2. MIPS体系结构
3. 计算机性能评价
4. MIPS模拟器
5. MIPS汇编语言程序设计
6. 一个排序算法的实例



计算机性能评价

计算机性能定义

- 对计算机性能进行比较时，最基本的标准就是时间标准——执行同样的程序所需时间最短的计算机就是最好的计算机

$$\text{性能} = \frac{1}{\text{执行时间}}$$

计算机性能评价

CPU执行时间

- 对于多任务系统，应该从响应时间中去除因为等待I/O操作而花去的时间和CPU执行其他程序所花费的时间，为此引入CPU执行时间的概念

CPU执行时间＝CPU时钟周期数×时钟周期

$$= \frac{\text{CPU时钟周期数}}{\text{时钟频率}}$$

CPU时钟周期数＝程序指令数×每条指令平均时钟周期数

每条指令平均时钟周期数

——CPI(clock cycle per instruction)

CPU执行时间 ＝ 指令数 × CPI × 时钟周期

计算机性能评价

CPI

- 执行不同的指令所需的时钟周期是不同的，简单的指令需要较少的时钟周期，复杂的指令需要较多的时钟周期

$$CPI = \sum_{i=1}^{n} CPI_i \times P_i$$

  - P_i —— 第i类指令出现的频度
  - CPI_i —— 执行第i类指令所需的时钟周期

计算机性能评价

影响计算机性能的因素

CPU执行时间 = 指令数 × CPI × 时钟周期

影响因素	影响
算法	指令数、CPI
程序设计语言	指令数、CPI
编译器	指令数、CPI
ISA	指令数、CPI、时钟周期
硬件实现	CPI、时钟周期

找一找： find the best mismatch!

性能有关的参数:

I. 指令数

II. CPI

III. 时钟频率

测试时的选择:

A. 测试方法

B. 编译器技术

C. 硬件技术

测试时的选择对于每一项参数来说，哪个影响最小？

	I.	II.	III.
1.	A	B	C
2.	A	C	B
3.	B	A	C
4.	B	C	A
5.	C	A	B
6.	C	B	A

影响计算机性能的因素

$$\text{CPU执行时间} = \text{指令数} \times \text{CPI} \times \text{时钟周期}$$

- 指令数
  - 取决于指令集体系结构(ISA)，与指令集的具体实现无关
  - 编译器也对指令数有很大影响
- CPI
  - 机器的实现细节（存储系统结构、处理器结构）
  - 测试程序包含的各类指令的组成
  - 编译器：充分利用硬件资源、优化分配寄存器（统计意义下影响相对较小）
- 时钟周期
  - 与机器的实现细节密切相关

计算机性能评价

提高计算机性能的途径

CPU执行时间 = 指令数 × CPI × 时钟周期

- 通过减少公式中任意一项来提高处理器的性能
- 但是，公式中的三项并不是相互独立的，它们之间有着复杂的联系，减少三项中的任意一项都有可能增加其他两项

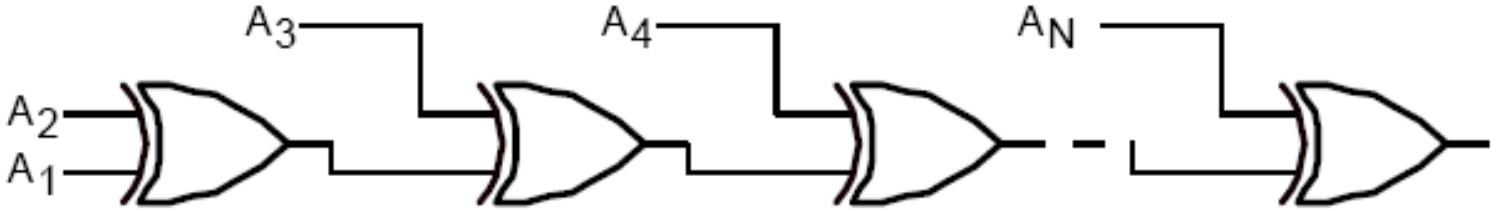
计算机性能评价

提高计算机性能的途径

- 一些技术可以在保持其他两项不变的前提下减少其中的一项：
  - ✓采用优化编译技术，在目标代码中消除冗余代码，可以减少指令数，但是并不提高CPI和时钟周期
  - ✓采用快速电路技术或更为先进的结构减少信号传输延迟，可以减少时钟周期，但是并不提高CPI和指令数

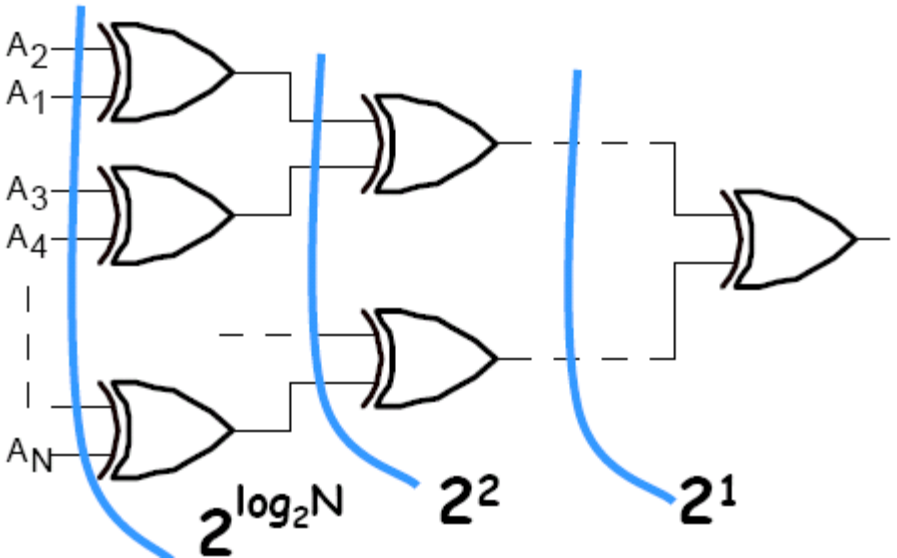
快速电路技术：速度和规模提高了

设计一个具有N输入的Xor门 .....



看起来很简单吧?

此电路的从输入A_1到输出的延时是一个异或门延时的N倍

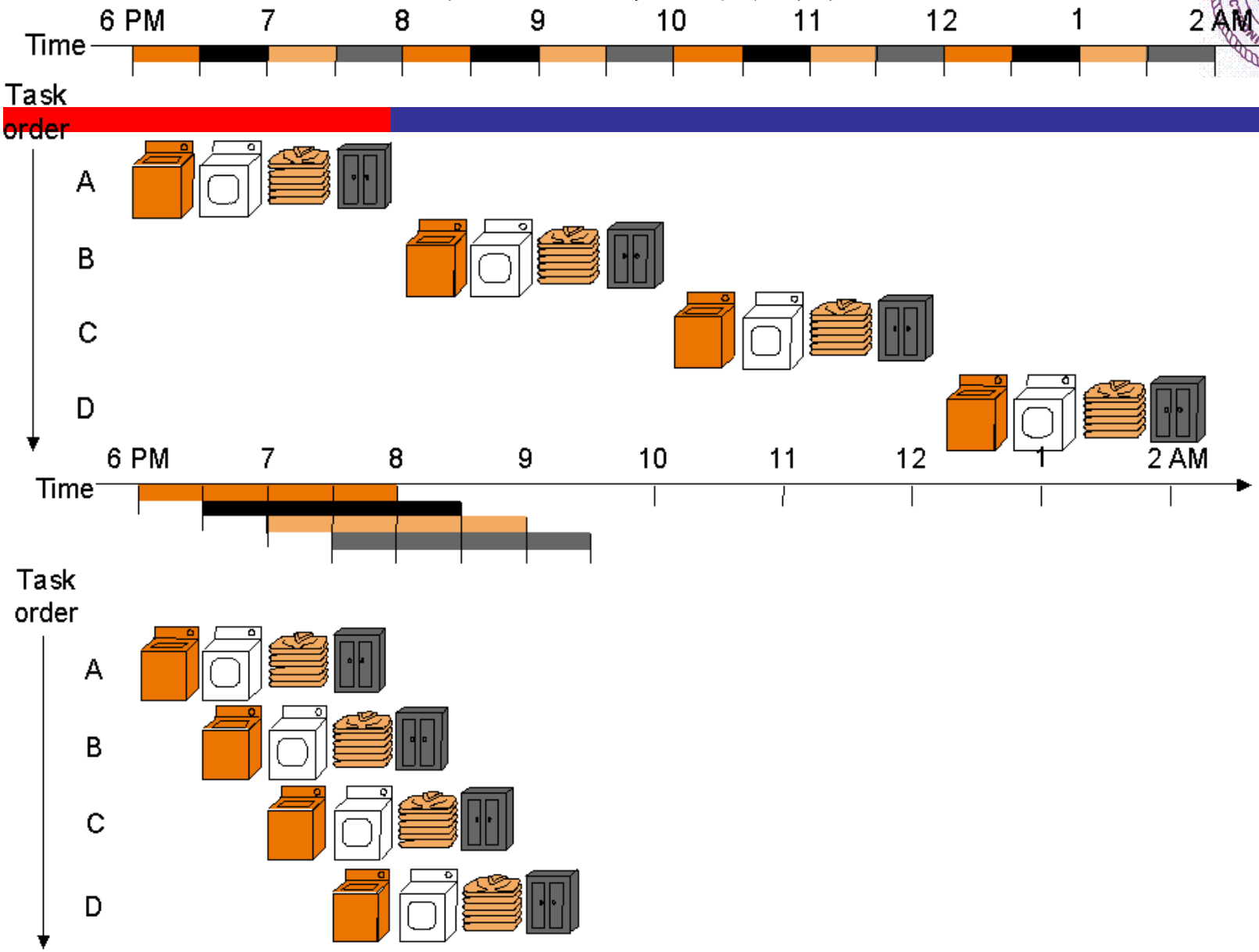


新的设计速度更快

- ✓ 此电路的从输入到输出的延时是一个异或门延时的$\log_2 N$倍

面积不变

直观的流水线结构


6 PM
7
8
9
10
11
12
1
2 AM
Time
Task order
A
B
C
D
6 PM
7
8
9
10
11
12
1
2 AM
Time
Task order
A
B
C
D

计算机性能评价

提高计算机性能的途径

- 一些技术可能在减少某一项的同时，增加另一项或两项的值
  - 例如，指令集可以包含更多复杂的指令，使每条指令执行更多的动作，可以减少指令数 ⟶ CISC
  - 虽然指令数减少，但是执行部件的复杂性增加，从而导致CPI和时钟周期的增加
- 只有在减少项的作用大于增加项的情况下，才能获得性能的提高

计算机性能评价

提高计算机性能的途径

- 减少CPI的愿望激发了许多体系结构和微体系结构(即体系结构的逻辑实现)技术
  - 体系结构方面：采用精简的指令集，减少每条指令的复杂性，从而减少CPI ⟶ RISC
  - 微体系结构方面：同时重叠执行多条指令

    ⟶ 流水线技术、超标量技术

课后作业7

Ref.2(第三版)

- **2.6、2.29、2.31、2.32**
- **4.14、4.15**
- **提交时间：4月23日**