南京大学
NANJING UNIVERSITY

指令系统

殷亚凤
智能软件与工程学院
苏州校区南雍楼东区225
yafeng@nju.edu.cn，https://yafengnju.github.io/

指令系统

- 指令格式设计
- 指令系统设计
- **程序的机器级表示**
- 指令系统实例

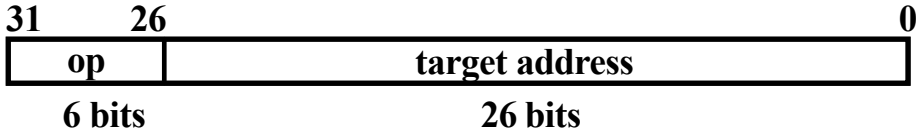
程序的机器级表示——MIPS汇编语言和机器语言

- MIPS中，所有指令都是32位宽，须按字地址对齐，字地址为4的倍数！
- **MIPS有三种指令格式**
  - **R-Type**

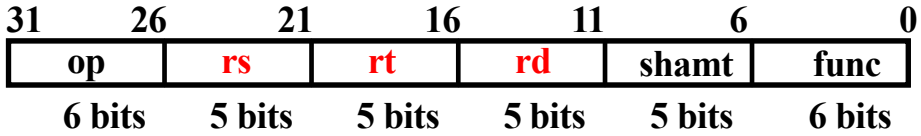
    两个操作数和结果都在寄存器的运算指令。如：sub rd, rs, rt
  - **I-Type**
    - 运算指令：一个寄存器、一个立即数。如：ori rt, rs, imm16
    - LOAD和STORE指令。如：lw rt, rs, imm16
    - 条件分支指令。如：beq rs, rt, imm16
  - **J-Type**

    无条件跳转指令。如：j target

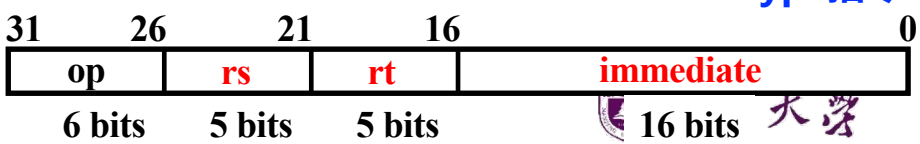
J-Type指令



R-Type指令



I-Type指令



程序的机器级表示——MIPS汇编语言和机器语言

表 4.2　MIPS 通用寄存器

名　　称	编　　号	功　　能
zero	0	恒为 0
at	1	为汇编程序保留
v0～v1	2～3	过程调用返回值
a0～a3	4～7	过程调用参数
t0～t7	8～15	临时变量，在被调用过程无须保存
s0～s7	16～23	在被调用过程需保存
t8～t9	24～25	临时变量，在被调用过程无须保存
k0～k1	26～27	为 OS 保留
gp	28	全局指针
sp	29	栈指针
fp	30	帧指针
ra	31	过程调用返回地址

寄存器的汇编表示以$符号表示，可以使用名称（**$ a0**），也可以用使用编号 (**$ 4**)。

程序的机器级表示——MIPS汇编语言和机器语言

表 4.3　MIPS 汇编语言示例列表

类别	指令名称	汇编举例	含义	备注
算术运算	add	add $s1, $s2, $s3	$s1 = $s2 + $s3	三个寄存器操作数
	subtract	sub $s1, $s2, $s3	$s1 = $s2 − $s3	三个寄存器操作数
存储访问	load word	lw $s1,100($s2)	$s1 = Memory[$s2+100]	从内存取一个字到寄存器
	store word	sw $s1,100($s2)	Memory[$s2+100] = $s1	从寄存器存一个字到内存
逻辑运算	and	and $s1, $s2, $s3	$s1 = $s2 & $s3	三个寄存器操作数,按位与
	or	or $s1, $s2, $s3	$s1 = $s2 | $s3	三个寄存器操作数,按位或
	nor	nor $s1, $s2, $s3	$s1 = ~($s2 | $s3)	三个寄存器操作数,按位或非
	and immediate	andi $s1, $s2,100	$s1 = $s2 & 100	寄存器和常数,按位与
	or immediate	ori $s1, $s2,100	$s1 = $s2 | 100	寄存器和常数,按位或
	shift left logical	sll $s1, $s2,10	$s1 = $s2≪10	按常数对寄存器逻辑左移
	shift right logical	srl $s1, $s2,10	$s1 = $s2≫10	按常数对寄存器逻辑右移
条件分支	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
	set on less than	slt $s1, $s2, $s3	if($s2< $s3) $s1=1; else $s1=0	小于则置寄存器为 1,否则为 0,用于后续指令判 0
	set on less than immediate	slt $s1, $s2,100	if($s2<100) $s1=1; else $s1=0	小于常数则置寄存器为 1,否则为 0,用于后续指令判 0
无条件跳转	jump	j L	go to L	直接跳转至目标地址
	jump register	jr $ra	go to $ra	过程返回
	jump and link	jal L	$ra=PC+4; go to L	过程调用

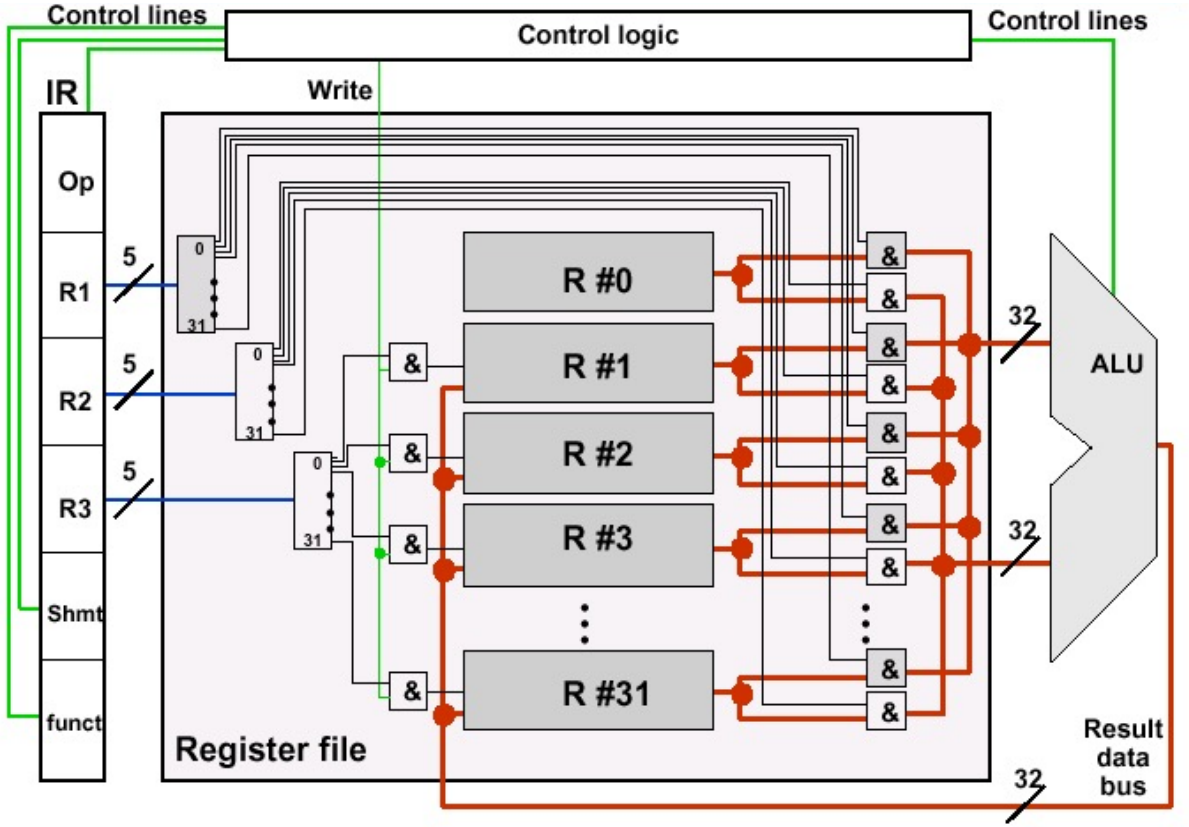
程序的机器级表示——MIPS汇编语言和机器语言

表 4.4　MIPS 机器代码示例列表

指　令	格式	指令举例						备　注
add	R	0	18	19	17	0	32	add　$s1，$s2，$s3
sub	R	0	18	19	17	0	34	sub　$s1，$s2，$s3
lw	I	35	18	17	100			lw　$s1，100($s2)
sw	I	43	18	17	100			sw　$s1，100($s2)
and	R	0	18	19	17	0	36	and　$s1，$s2，$s3
or	R	0	18	19	17	0	37	or　$s1，$s2，$s3

指　令	格式	指令举例						备　注
nor	R	0	18	19	17	0	39	nor　$s1，$s2，$s3
andi	I	12	18	17	100			andi　$s1，$s2，100
ori	I	13	18	17	100			ori　$s1，$s2，100
sll	R	0	0	18	17	10	0	sll　$s1，$s2，10
srl	R	0	0	18	17	10	2	srl　$s1，$s2，10
beq	I	4	17	18	25			beq　$s1，$s2，100
bne	I	5	17	18	25			bne　$s1，$s2，100
slt	R	0	18	19	17	0	42	slt　$s1，$s2，$s3
j	J	2	2500					j　10000
jr	R	0	31	0	0	0	8	jr　$ra
jal	J	3	2500					jal　10000
字段大小		6 位	5 位	5 位	5 位	5 位	6 位	
R-型指令	R	OP	rs	rt	rd	shamt	func	
I-型指令	I	OP	rs	rt	address			

程序的机器级表示——MIPS R-型指令的电路



问题：

你能给出R-型指令在上述通路中的大致执行过程吗？

程序的机器级表示——MIPS R-型指令的电路

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
  - 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

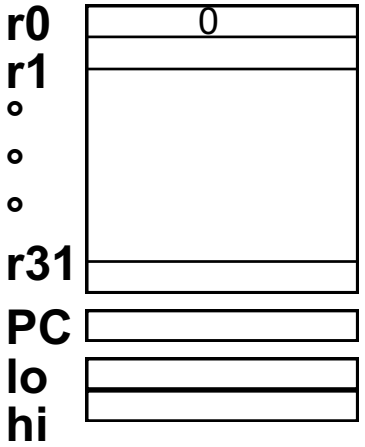
2：执行阶段

- 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

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。

程序的机器级表示——MIPS指令中寄存器和存储器

- **寄存器数据指定：**
  - 32 x 32-bit 通用寄存器（**r0 = 0**）
  - 寄存器编号占5 bit
  - **32 x 32-bit 单精度浮点寄存器（f0 ~ f31, 可配对为16个双精度）**
  - HI, LO, PC: 特殊寄存器
  - **寄存器功能（书本103页）**
- **存储器数据指定：**
  - 32- bit 有效地址--> 可访问空间: 2^{32} 字节
  - 采用大端方式访问数据
  - 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



程序的机器级表示——MIPS算术和逻辑指令

指令	举例	含义	备注
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	

需要判溢出，溢出时发生“异常”

指令	举例	含义	备注
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。而MIPS是由硬件直接判溢出否。

程序的机器级表示——MIPS数据传输指令

指令	备注	含义
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

- **问题：为什么指令必须支持不同长度的操作数？**

因为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；

- **指令中操作数长度由什么决定？**

操作数长度由不同的操作码指定。

程序的机器级表示——MIPS调用/返回等指令

指令	举例	含义
jump register	jr $31	go to $31
	For switch, procedure return（对应过程返回）	
jump and link	jal 10000	$31 = PC + 4; go to 10000
	For procedure call（对应过程或函数调用）	
jump	j 10000	go to 10000
	Jump to target address	

call / return

伪指令：blt, ble, bgt, bge，有若干指令序列实现

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

} 按补码比较大小

问题：机器指令中立即数是多少？ **100=0064H**

branch on equal	beq $1,$2,100	if ($1 == $2) go to PC+4+25*4
branch on not eq.	bne $1,$2,100	if ($1!= $2) go to PC+4+25*4

} 汇编指令中给出的是相对单元数！

问题：机器指令中立即数是多少？ **25=0019H**

分支指令的机器代码中给出的是相对于当前指令的指令条数！

南京大学 NANJING UNIVERSITY

程序的机器级表示——算术运算

例子：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？

举例：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
```

←

问题：以下程序如何处理呢？

E.g. f= (g+65000) - (i+50)

16位立即数：-32768~32767

指令设计时必须考虑这种情况！MIPS有一条专门的处理指令！

程序的机器级表示——选择结构的表示

```
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:
```

程序的机器级表示——选择结构的表示

比较判断

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:
```

程序的机器级表示——循环结构的表示

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

数组元素为int类型，sizeof(int)=4

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

程序的机器级表示——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是全局静态区变量

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？并把返回结果传递给调用程序？
4. 如何保证调用程序中寄存器内容不被破坏？

程序的机器级表示——过程调用

- **过程调用的执行步骤（假定过程P调用过程Q）：**
  - 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)

```
set_array
{
  compare (num, i);
}
```

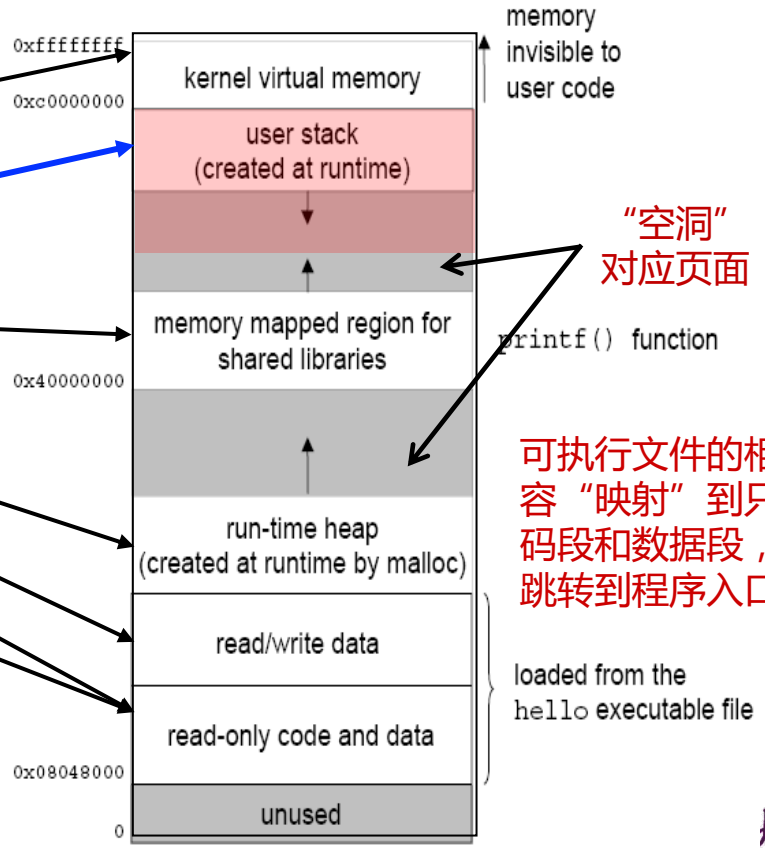
还记得栈在虚拟空间中哪个区域吗？

一般用“栈”来传递参数、保存返回地址，并用来临时存放过程中的局部变量等。为什么？

→ 便于嵌套或递归调用！

程序的机器级表示——虚拟地址空间

- Linux在X86上的虚拟地址空间
  （其他Unix系统的设计类此）
  - 内核（Kernel）
  - **用户栈（User Stack）**
  - 共享库（Shared Libraries）
  - 堆（heap）
  - 可读写数据（Read/Write Data）
  - 只读数据（Read-only Data）
  - 代码（Code）



问题：加载时是否真正从磁盘调入信息到主存？

实际上不会从磁盘调入，只是将代码和数据与虚拟空间建立对应关系，称为“映射”。

程序的机器级表示——栈的概念

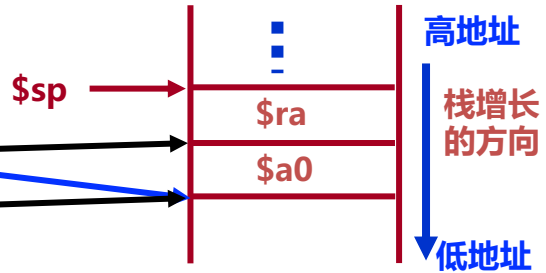
- **栈的基本概念**
  - 是一个“先进后出”队列
  - 需一个栈指针指向栈顶元素
  - 每个元素长度一致
  - 用“入栈”（push）和“出栈”（pop）操作访问栈元素
- **MIPS中栈的实现**
  - 用栈指针寄存器$sp来指示栈顶元素
  - 每个元素的长度为32位，即：一个字(4个字节)
  - “入栈” 和“出栈” 操作用 sw / lw 指令来实现，需用add / sub指令调整$sp的值，不能像x86那样自动进行栈指针的调整（有些处理器有专门的push/pop指令，能自动调整栈指针。如x86）
  - 栈生长方向

    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：

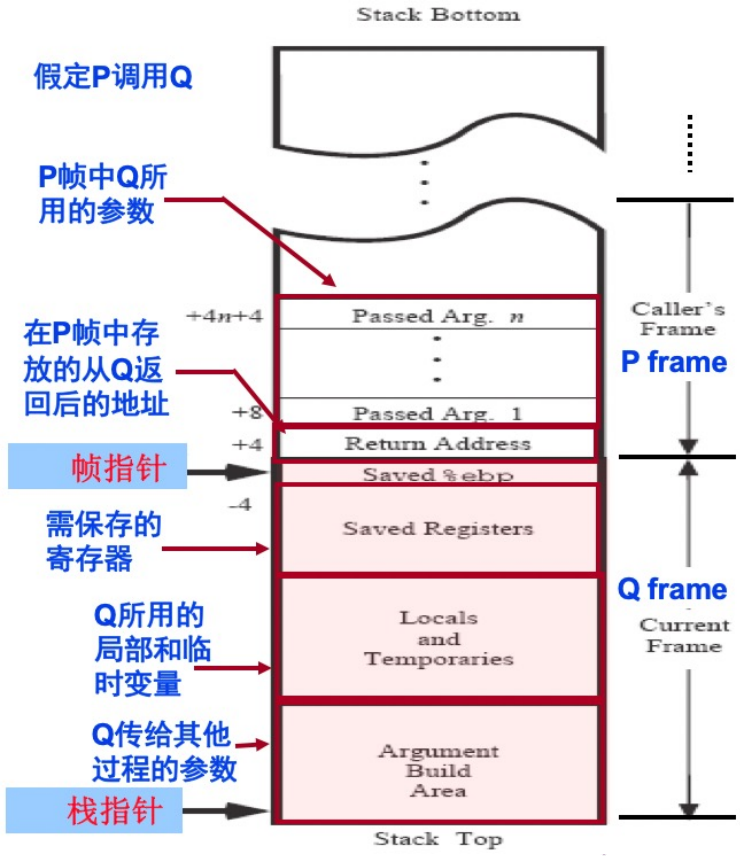
```
sub  $sp, $sp, 8
sw   $ra, 4($sp)
sw   $a0, 0($sp)
```



程序的机器级表示——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可通过帧指针进行
- 复杂局部变量一定分配在栈帧中

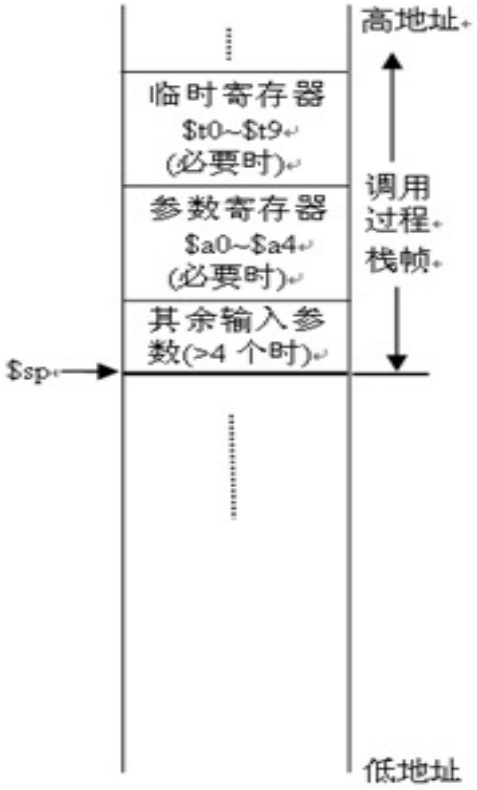
X86栈帧如右图所示



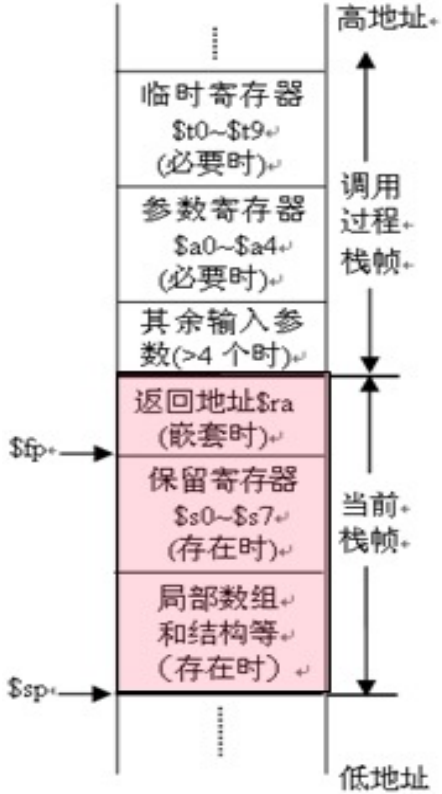
程序的机器级表示——MIPS中的过程调用（P调用Q）

- **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$gp的值不变
  - 帧指针寄存器$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等复杂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
- **不同架构对（被）调用过程栈帧规定保存的信息可能不同**。如：
  - 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中入口参数保存在调用过程栈帧中；MIPS也在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。

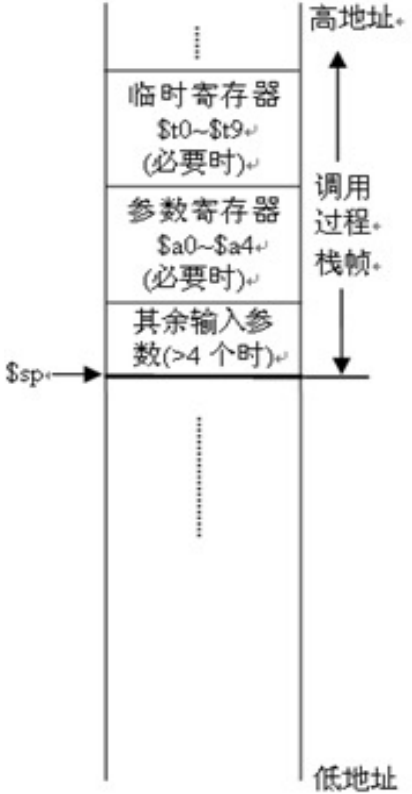
程序的机器级表示——过程调用时栈和栈帧的变化



(a) 过程调用前



(b) 过程调用中



(c) 过程调用后

程序的机器级表示——过程调用举例

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。

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

问题：上述假设有何问题？

```
sll     $s2, $a1, 2
addu    $s2 $s2, $a0
lw      $t0, 0($s2)
lw      $s3, 4($s2)
sw      $s3, 0($s2)
sw      $t0, 4($s2)
```

参数v和k应该在$a0和$a1

```
; mulitply k by 4
; address of v[k]
 ; load v[k]
 ; load v[k+1]
; store v[k+1] into v[k]
; 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
```

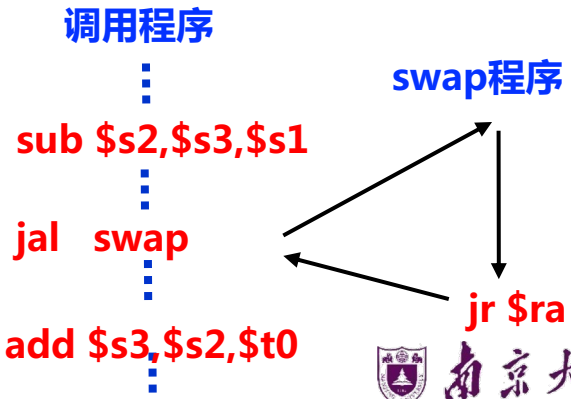
问题1：若swap中不保存$s2，则会发生什么情况？

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

问题3：程序的执行轨迹是什么？



程序的机器级表示——过程调用举例

```
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]
```

如果过程体中又调用了其他函数，则 $ra是否需保存？

保存！

问题：是否一定要将返回地址（$31）保存到栈中？

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

程序的机器级表示——过程调用举例

```
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是全局静态区变量

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

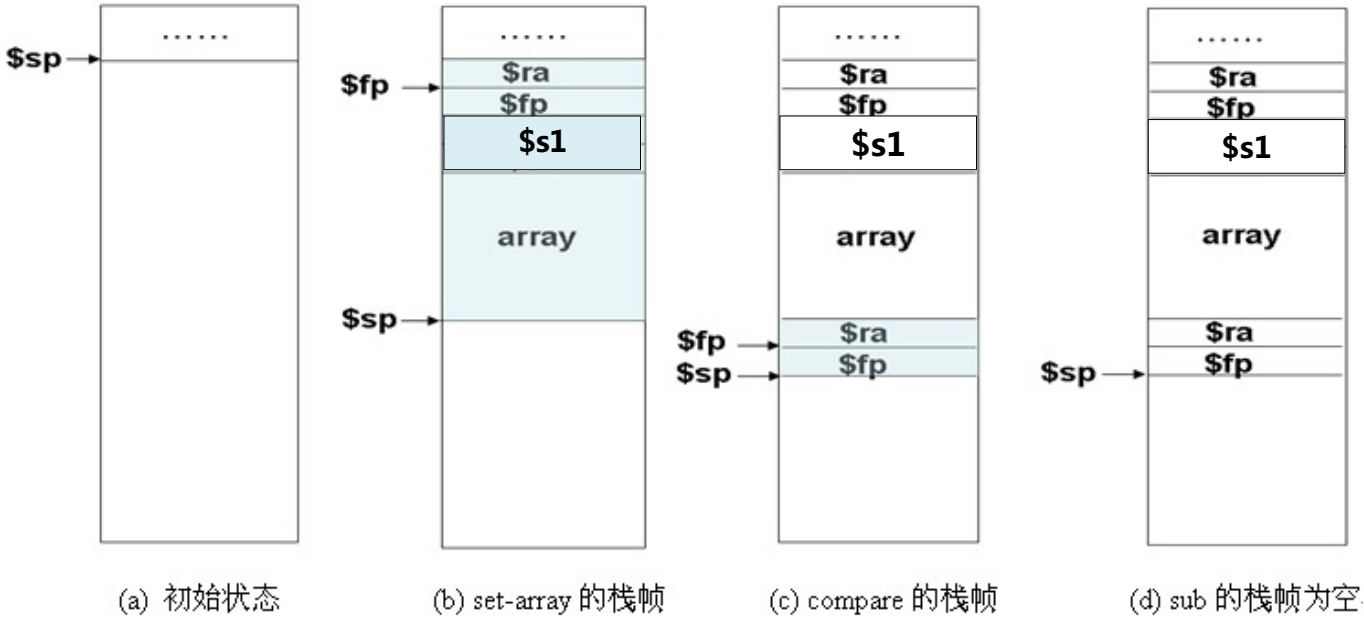
问题1：编译器如何为全局变量和局部变量分配空间？

问题2：set_array执行结果是什么？为什么？

程序的机器级表示——过程调用举例

- **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~$s7。
- 过程**set_array**的入口参数为num，没有返回参数，有一个局部数组，被调用过程为compare。因此，栈帧中除了保留所用的保存寄存器外，必须**保留返回地址**（是否保存$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，**但只需将$s1保存在栈中**，另外加上返回地址$ra、帧指针$fp、局部数组，其栈帧空间最少为4×3+40=52B。

程序的机器级表示——过程调用举例



- **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。

指令系统

- 指令格式设计
- 指令系统设计
- 程序的机器级表示
- **指令系统实例**

指令系统实例——RISC-V指令系统概述

- **设计目标**
  - 广泛的适应性：从最袖珍的嵌入式微控制器，到最快的高性能计算机
  - 支持各种异构处理架构，成为定制加速器的基础
  - 稳定的基础指令集架构，并能灵活扩展，且扩展时不影响基础部分
- **开源理念和设计原则**
  - 本着“指令集应自由（Instruction Set Want to be Free）”的理念，指令集完全公开，且无需为指令集付费
  - 由一个非盈利性质的基金会管理，以保持指令集的稳定性，并加快生态建设
  - 与以前的增量ISA不同，遵循“大道至简”的设计哲学，采用模块化设计，既保持基础指令集的稳定，也保证扩展指令集的灵活配置
  - 特点：具有模块化结构、稳定性和可扩展性好，在简洁性、实现成本、功耗、性能和程序代码量等各方面具有显著优势
- **RISC-V的模块化结构**
  - **核心：RV32I** + **标准扩展集：RV32M、RV32F、RV32D、RV32A** = RV32G
  - 32位架构RV32G = RV32IMAFD，其压缩指令集RV32C（指令长度16位）
  - 64位架构RV64G = RV64IMAFD，其压缩指令集RV64C（指令长度16位）
  - 向量计算RV32V和RV64V；嵌入式RV32E（RV32I的子集，16个通用寄存器）

指令系统实例——RISC-V指令参考卡

- **核心指令集**：基础整数指令集 RV32I 和 RV64I
- **特权指令**：陷阱指令对应的返回指令、wfi等待中断指令、sfence.vma虚拟存储器的同步操作
- **伪指令举例**
- **压缩指令集**：RV32C和RV64C

Base Integer Instructions: RV32I and RV64I

Category	Name	Fmt	RV32I Base		+RV64I	
Shifts	Shift Left Logical	R	SLL	rd,rs1,rs2	SLLW	rd,rs1,rs2
	Shift Left Log. Imm.	I	SLLI	rd,rs1,shamt	SLLIW	rd,rs1,shamt
	Shift Right Logical	R	SRL	rd,rs1,rs2	SRLW	rd,rs1,rs2
	Shift Right Log. Imm.	I	SRLI	rd,rs1,shamt	SRLIW	rd,rs1,shamt
	Shift Right Arithmetic	R	SRA	rd,rs1,rs2	SRAW	rd,rs1,rs2
	Shift Right Arith. Imm.	I	SRAI	rd,rs1,shamt	SRAIW	rd,rs1,shamt
Arithmetic	ADD	R	ADD	rd,rs1,rs2	ADDW	rd,rs1,rs2
	ADD Immediate	I	ADDI	rd,rs1,imm	ADDIW	rd,rs1,imm
	SUBtract	R	SUB	rd,rs1,rs2	SUBW	rd,rs1,rs2
	Load Upper Imm	U	LUI	rd,imm		
	Add Upper Imm to PC	U	AUIPC	rd,imm		
Logical	XOR	R	XOR	rd,rs1,rs2		
	XOR Immediate	I	XORI	rd,rs1,imm		
	OR	R	OR	rd,rs1,rs2		
	OR Immediate	I	ORI	rd,rs1,imm		
	AND	R	AND	rd,rs1,rs2		
	AND Immediate	I	ANDI	rd,rs1,imm		
Compare	Set <	R	SLT	rd,rs1,rs2		
	Set < Immediate	I	SLTI	rd,rs1,imm		
	Set < Unsigned	R	SLTU	rd,rs1,rs2		
	Set < Imm Unsigned	I	SLTIU	rd,rs1,imm		
Branches	Branch =	B	BEQ	rs1,rs2,imm		
	Branch ≠	B	BNE	rs1,rs2,imm		
	Branch <	B	BLT	rs1,rs2,imm		
	Branch ≥	B	BGE	rs1,rs2,imm		
	Branch < Unsigned	B	BLTU	rs1,rs2,imm		
	Branch ≥ Unsigned	B	BGEU	rs1,rs2,imm		
Jump & Link	J&L	J	JAL	rd,imm		
	Jump & Link Register	I	JALR	rd,rs1,imm		
Synch	Synch thread	I	FENCE			
	Synch Instr & Data	I	FENCE.I			
Environment	CALL	I	ECALL			
	BREAK	I	EBREAK			

Control Status Register (CSR)				
	Read/Write	I	CSRRW	rd,csr,rs1
	Read & Set Bit	I	CSRRS	rd,csr,rs1
	Read & Clear Bit	I	CSRRC	rd,csr,rs1
	Read/Write Imm	I	CSRRWI	rd,csr,imm
	Read & Set Bit Imm	I	CSRRSI	rd,csr,imm
	Read & Clear Bit Imm	I	CSRRCI	rd,csr,imm

					+RV64I	
Loads	Load Byte	I	LB	rd,rs1,imm		
	Load Halfword	I	LH	rd,rs1,imm		
	Load Byte Unsigned	I	LBU	rd,rs1,imm		
	Load Half Unsigned	I	LHU	rd,rs1,imm		
	Load Word	I	LW	rd,rs1,imm	LWU	rd,rs1,imm
					LD	rd,rs1,imm
Stores	Store Byte	S	SB	rs1,rs2,imm		
	Store Halfword	S	SH	rs1,rs2,imm		
	Store Word	S	SW	rs1,rs2,imm	SD	rs1,rs2,imm

RV Privileged Instructions

Category	Name	Fmt	RV mnemonic
Trap	Mach-mode trap return	R	MRET
	Supervisor-mode trap return	R	SRET
Interrupt	Wait for Interrupt	R	WFI
MMU	Virtual Memory FENCE	R	SFENCE.VMA rs1,rs2

Examples of the 60 RV Pseudoinstructions

Branch = 0 (BEQ rs,x0,imm)	B	BEQZ rs,imm
Jump (uses JAL x0,imm)	J	J imm
MoVe (uses ADDI rd,rs,0)	R	MV rd,rs
RETurn (uses JALR x0,0,ra)	I	RET

Optional Compressed (16-bit) Instruction Extension: RV32C

Category	Name	Fmt	RVC		RISC-V equivalent	
Loads	Load Word	CL	C.LW	rd',rs1',imm	LW	rd',rs1',imm*4
	Load Word SP	CI	C.LWSP	rd,imm	LW	rd,sp,imm*4
	Float Load Word SP	CL	C.FLW	rd',rs1',imm	FLW	rd',rs1',imm*8
	Float Load Word	CI	C.FLWSP	rd,imm	FLW	rd,sp,imm*8
	Float Load Double	CL	C.FLD	rd',rs1',imm	FLD	rd',rs1',imm*16
	Float Load Double SP	CI	C.FLDSP	rd,imm	FLD	rd,sp,imm*16
Stores	Store Word	CS	C.SW	rs1',rs2',imm	SW	rs1',rs2',imm*4
	Store Word SP	CSS	C.SWSP	rs2,imm	SW	rs2,sp,imm*4
	Float Store Word	CS	C.FSW	rs1',rs2',imm	FSW	rs1',rs2',imm*8
	Float Store Word SP	CSS	C.FSWSP	rs2,imm	FSW	rs2,sp,imm*8
	Float Store Double	CS	C.FSD	rs1',rs2',imm	FSD	rs1',rs2',imm*16
	Float Store Double SP	CSS	C.FSDSP	rs2,imm	FSD	rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD	rd,rs1	ADD	rd,rd,rs1
	ADD Immediate	CI	C.ADDI	rd,imm	ADDI	rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP	x0,imm	ADDI	sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN	rd',imm	ADDI	rd',sp,imm*4
	SUB	CR	C.SUB	rd,rs1	SUB	rd,rd,rs1
	AND	CR	C.AND	rd,rs1	AND	rd,rd,rs1
	AND Immediate	CI	C.ANDI	rd,imm	ANDI	rd,rd,imm
	OR	CR	C.OR	rd,rs1	OR	rd,rd,rs1
	eXclusive OR	CR	C.XOR	rd,rs1	AND	rd,rd,rs1
	MoVe	CR	C.MV	rd,rs1	ADD	rd,rs1,x0
	Load Immediate	CI	C.LI	rd,imm	ADDI	rd,x0,imm
	Load Upper Imm	CI	C.LUI	rd,imm	LUI	rd,imm
Shifts	Shift Left Imm	CI	C.SLLI	rd,imm	SLLI	rd,rd,imm
	Shift Right Ari. Imm.	CI	C.SRAI	rd,imm	SRAI	rd,rd,imm
	Shift Right Log. Imm.	CI	C.SRLI	rd,imm	SRLI	rd,rd,imm
Branches	Branch=0	CB	C.BEQZ	rs1',imm	BEQ	rs1',x0,imm
	Branch≠0	CB	C.BNEZ	rs1',imm	BNE	rs1',x0,imm
Jump	Jump	CJ	C.J	imm	JAL	x0,imm
	Jump Register	CR	C.JR	rd,rs1	JALR	x0,rs1,0
Jump & Link	J&L	CJ	C.JAL	imm	JAL	ra,imm
	Jump & Link Register	CR	C.JALR	rs1	JALR	ra,rs1,0
System	Env. BREAK	CI	C.EBREAK		EBREAK	

Optional Compressed Extention: RV64C

All RV32C (except C.JAL, 4 word loads, 4 word strores) plus:

ADD Word (C.ADDW)	Load Doubleword (C.LD)
ADD Imm. Word (C.ADDIW)	Load Doubleword SP (C.LDSP)
SUBtract Word (C.SUBW)	Store Doubleword (C.SD)
	Store Doubleword SP (C.SDSP)

指令系统实例——RISC-V指令参考卡

- **扩展指令集**

乘除运算指令集RVM、原子操作指令集RVA、浮点运算指令集RVF和RVD、向量操作指令集RVV

- **通用寄存器的调用约定**

32个定点通用寄存器x0~x31；32个浮点寄存器f0~f31；通用寄存器x0中恒0；x1中返回地址；x2、x3和x4分别为栈指针、全局指针和线程指针

Optional Multiply-Divide Instruction Extension: RVM

Category	Name	Fmt	RV32M (Multiply-Divide)	+RV64M
Multiply	MULtiply	R	MUL rd,rs1,rs2	MULW rd,rs1,rs2
	MULtiply High	R	MULH rd,rs1,rs2	
	MULtiply High Sign/Uns	R	MULHSU rd,rs1,rs2	
	MULtiply High Uns	R	MULHU rd,rs1,rs2	
Divide	DIVide	R	DIV rd,rs1,rs2	DIVW rd,rs1,rs2
	DIVide Unsigned	R	DIVU rd,rs1,rs2	
Remainder	REMainder	R	REM rd,rs1,rs2	REMW rd,rs1,rs2
	REMainder Unsigned	R	REMU rd,rs1,rs2	REMUW rd,rs1,rs2

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV32A (Atomic)	+RV64A
Load	Load Reserved	R	LR.W rd,rs1	LR.D rd,rs1
Store	Store Conditional	R	SC.W rd,rs1,rs2	SC.D rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.W rd,rs1,rs2	AMOSWAP.D rd,rs1,rs2
Add	ADD	R	AMOADD.W rd,rs1,rs2	AMOADD.D rd,rs1,rs2
Logical	XOR	R	AMOXOR.W rd,rs1,rs2	AMOXOR.D rd,rs1,rs2
	AND	R	AMOAND.W rd,rs1,rs2	AMOAND.D rd,rs1,rs2
	OR	R	AMOOR.W rd,rs1,rs2	AMOOR.D rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.W rd,rs1,rs2	AMOMIN.D rd,rs1,rs2
	MAXimum	R	AMOMAX.W rd,rs1,rs2	AMOMAX.D rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W rd,rs1,rs2	AMOMINU.D rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W rd,rs1,rs2	AMOMAXU.D rd,rs1,rs2

Two Optional Floating-Point Instruction Extensions: RVF & RVD

Category	Name	Fmt	RV32{F|D} (SP,DP Fl. Pt.)	+RV64{F|D}
Move	Move from Integer	R	FMV.W.X rd,rs1	FMV.D.X rd,rs1
	Move to Integer	R	FMV.X.W rd,rs1	FMV.X.D rd,rs1
Convert	ConVerT from Int	R	FCVT.{S|D}.W rd,rs1	FCVT.{S|D}.L rd,rs1
	ConVerT from Int Unsigned	R	FCVT.{S|D}.WU rd,rs1	FCVT.{S|D}.LU rd,rs1
	ConVerT to Int	R	FCVT.W.{S|D} rd,rs1	FCVT.L.{S|D} rd,rs1
	ConVerT to Int Unsigned	R	FCVT.WU.{S|D} rd,rs1	FCVT.LU.{S|D} rd,rs1
Load	Load	I	FL{W,D} rd,rs1,imm	
Store	Store	S	FS{W,D} rs1,rs2,imm	
Arithmetic	ADD	R	FADD.{S|D} rd,rs1,rs2	
	SUBtract	R	FSUB.{S|D} rd,rs1,rs2	
	MULtiply	R	FMUL.{S|D} rd,rs1,rs2	
	DIVide	R	FDIV.{S|D} rd,rs1,rs2	
	SQuare RooT	R	FSQRT.{S|D} rd,rs1	
Mul-Add	Multiply-ADD	R	FMADD.{S|D} rd,rs1,rs2,rs3	
	Multiply-SUBtract	R	FMSUB.{S|D} rd,rs1,rs2,rs3	
	Negative Multiply-SUBtract	R	FNMSUB.{S|D} rd,rs1,rs2,rs3	
	Negative Multiply-ADD	R	FNMADD.{S|D} rd,rs1,rs2,rs3	
Sign Inject	SiGN source	R	FSGNJ.{S|D} rd,rs1,rs2	
	Negative SiGN source	R	FSGNJN.{S|D} rd,rs1,rs2	
	Xor SiGN source	R	FSGNJX.{S|D} rd,rs1,rs2	
Min/Max	MINimum	R	FMIN.{S|D} rd,rs1,rs2	
	MAXimum	R	FMAX.{S|D} rd,rs1,rs2	
Compare	compare Float =	R	FEQ.{S|D} rd,rs1,rs2	
	compare Float <	R	FLT.{S|D} rd,rs1,rs2	
	compare Float ≤	R	FLE.{S|D} rd,rs1,rs2	
Categorize	CLASSify type	R	FCLASS.{S|D} rd,rs1	
Configure	Read Status	R	FRCSR rd	
	Read Rounding Mode	R	FRRM rd	
	Read Flags	R	FRFLAGS rd	
	Swap Status Reg	R	FSCSR rd,rs1	
	Swap Rounding Mode	R	FSRM rd,rs1	
	Swap Flags	R	FSFLAGS rd,rs1	
	Swap Rounding Mode Imm	I	FSRMI rd,imm	
	Swap Flags Imm	I	FSFLAGSI rd,imm	

Calling Convention

Register	ABI Name	Saver
x0	zero	---
x1	ra	Caller
x2	sp	Callee
x3	gp	---
x4	tp	---
x5-7	t0-2	Caller
x8	s0/fp	Callee
x9	s1	Callee
x10-11	a0-1	Caller
x12-17	a2-7	Caller
x18-27	s2-11	Callee
x28-31	t3-t6	Caller
f0-7	ft0-7	Caller
f8-9	fs0-1	Callee
f10-11	fa0-1	Caller
f12-17	fa2-7	Caller
f18-27	fs2-11	Callee
f28-31	ft8-11	Caller
zero	Hardwired zero	
ra	Return address	
sp	Stack pointer	
gp	Global pointer	
tp	Thread pointer	
t0-0,ft0-7	Temporaries	
s0-11,fs0-11	Saved registers	
a0-7,fa0-7	Function args	

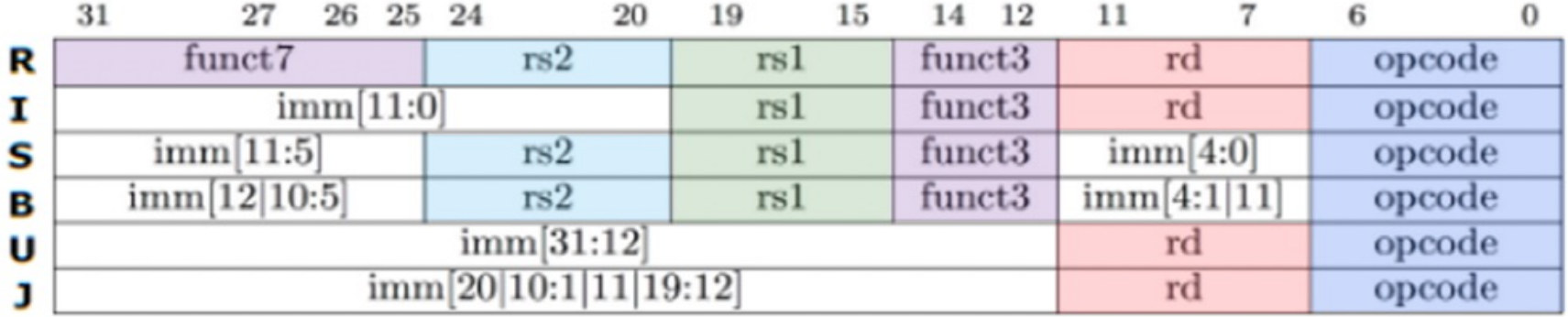
Optional Vector Extension: RVV

Name	Fmt	RV32V/R64V
SET Vector Len.	R	SETVL rd,rs1
MULtiply High	R	VMULH rd,rs1,rs2
REMainder	R	VREM rd,rs1,rs2
Shift Left Log.	R	VSLL rd,rs1,rs2
Shift Right Log.	R	VSRL rd,rs1,rs2
Shift R. Arith.	R	VSRA rd,rs1,rs2
LoaD	I	VLD rd,rs1,imm
LoaD Strided	R	VLDS rd,rs1,rs2
LoaD indeXed	R	VLDX rd,rs1,rs2
STore	S	VST rd,rs1,imm
STore Strided	R	VSTS rd,rs1,rs2
STore indeXed	R	VSTX rd,rs1,rs2
AMO SWAP	R	AMOSWAP rd,rs1,rs2
AMO ADD	R	AMOADD rd,rs1,rs2
AMO XOR	R	AMOXOR rd,rs1,rs2
AMO AND	R	AMOAND rd,rs1,rs2
AMO OR	R	AMOOR rd,rs1,rs2
AMO MINimum	R	AMOMIN rd,rs1,rs2
AMO MAXimum	R	AMOMAX rd,rs1,rs2
Predicate =	R	VPEQ rd,rs1,rs2
Predicate ≠	R	VPNE rd,rs1,rs2
Predicate <	R	VPLT rd,rs1,rs2
Predicate ≥	R	VPGE rd,rs1,rs2
Predicate AND	R	VPAND rd,rs1,rs2
Pred. AND NOT	R	VPANDN rd,rs1,rs2
Predicate OR	R	VPOR rd,rs1,rs2
Predicate XOR	R	VPXOR rd,rs1,rs2
Predicate NOT	R	VPNOT rd,rs1
Pred. SWAP	R	VPSWAP rd,rs1
MOVe	R	VMOV rd,rs1
ConVerT	R	VCVT rd,rs1
ADD	R	VADD rd,rs1,rs2
SUBtract	R	VSUB rd,rs1,rs2
MULtiply	R	VMUL rd,rs1,rs2
DIVide	R	VDIV rd,rs1,rs2
SQuare RooT	R	VSQRT rd,rs1,rs2
Multiply-ADD	R	VFMADD rd,rs1,rs2,rs3
Multiply-SUB	R	VFMSUB rd,rs1,rs2,rs3
Neg. Mul.-SUB	R	VFNMSUB rd,rs1,rs2,rs3
Neg. Mul.-ADD	R	VFNMADD rd,rs1,rs2,rs3
SiGN inJect	R	VSGNJ rd,rs1,rs2
Neg SiGN inJect	R	VSGNJN rd,rs1,rs2
Xor SiGN inJect	R	VSGNJX rd,rs1,rs2
MINimum	R	VMIN rd,rs1,rs2
MAXimum	R	VMAX rd,rs1,rs2
XOR	R	VXOR rd,rs1,rs2
OR	R	VOR rd,rs1,rs2
AND	R	VAND rd,rs1,rs2
CLASS	R	VCLASS rd,rs1
SET Data Conf.	R	VSETDCFG rd,rs1
EXTRACT	R	VEXTRACT rd,rs1,rs2
MERGE	R	VMERGE rd,rs1,rs2
SELECT	R	VSELECT rd,rs1,rs2

指令系统实例——32位RISC-V指令格式

- **共有6种指令格式**
  - R-型为寄存器操作数指令
  - I-型为短立即数或装入（Load）指令
  - S-型为存储（Store）指令
  - B-型为条件跳转指令
  - U-型为长立即数操作指令
  - J-型为无条件跳转指令

- opcode：操作码字段
- rd、rs1和rs2：通用寄存器编号
- imm：立即数，其位数在括号[ ]中表示
- funct3和funct7：分别表示3位功能码和7位功能码，和opcode字段一起定义指令的操作功能



指令系统实例——16位RISC-V压缩指令格式

- 共有8种指令格式。与32位指令相比，16位指令中的一部分寄存器编号还是占5位。指令变短了，但还是32位架构，处理的还是32位数据，还是有32个通用寄存器。
- 为了缩短指令长度，操作码op、功能码funct、立即数imm和另一部分寄存器编号的位数都减少了。
- 每条16位指令都有功能完全相同的32位指令，在执行时由硬件先转换为32位指令再执行。**目的是：缩短程序代码量，用少量时间换空间！**

	15 14 13	12	11 10 9 8 7	6 5 4 3 2	1 0
CR	funct4		rd/rs1	rs2	op
CI	funct3	imm	rd/rs1	imm	op
CSS	funct3	imm		rs2	op
CIW	funct3	imm		rd′	op
CL	funct3	imm	rs1′	imm rd′	op
CS	funct3	imm	rs1′	imm rs2′	op
CB	funct3	offset	rs1′	offset	op
CJ	funct3	jump target			op

指令系统实例——基础整数指令集

- 包含：
  - 移位（Shifts）
  - 算术运算（Arithmetic）
  - 逻辑运算（Logical）
  - 比较（Compare）

  } 整数运算类指令

  - 分支（Branch）
  - 跳转链接（Jump & Link）

  } 控制转移类指令

  - 同步（Synch）
  - 环境（Environment）
  - 控制状态寄存器（Control Status Register）

  } 系统控制类指令

  - 取数（Load）
  - 存数（Store）

  } 存储访问类指令

RTL规定：

R[r]：通用寄存器r的内容

M[addr]：存储单元addr的内容

M[R[r]]：寄存器r的内容所指存储单元的内容

PC：PC的内容

M[PC]：PC所指存储单元的内容

SEXT[imm]：对imm进行符号扩展

ZEXT[imm]：对imm进行零扩展

传送方向用←表示，即传送源在右，传送目的在左

指令系统实例——RISC-V基础整数指令集（RV32I）

整数运算类指令

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	R sltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and

指令系统实例——RISC-V基础整数指令集（RV32I）

U型指令共2条

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc

lui rd, imm20：将立即数imm20存到rd寄存器高20位，低12位为0。该指令和“addi rd, rs1, imm12”结合，可以实现对一个32位变量赋初值。

举例：请给出C语句“int x=-8191;”对应的RISC-V机器级代码。
解：C语句“int x=-8191;”对应的RISC-V机器指令和汇编指令为：

1111 1111 1111 1111 1110 00101 0110111 lui x5, 1048574 #R[x5]←FFFF E000H
0000 0000 0001 00101 000 00101 0010011 addi x5, x5, 1 #R[x5]←R[x5]+SEXT[001H]

SEXT表示符号扩展

-8191的机器数为：1111 1111 1111 1111 1110 0000 0000 0001

auipc rd, imm20：将立即数imm20加到PC的高20位上，结果存rd。可用指令“auipc x10, 0”获取当前PC的内容，存入寄存器x10中。

指令系统实例——RISC-V基础整数指令集（RV32I）

I 型指令共9条，其中三条为用立即数指定所移位数的移位指令

31 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai

操作码opcode：都是0010011，其功能由funct3指定，而当funct3=101时，再有高7位区分是算术右移（srai）还是逻辑右移（srli）。

imm[11:0]：12位立即数，符号扩展为32位，作为第2个源操作数，和R[rs1]（寄存器rs1中的内容）进行运算，结果存rd。

shamt：指出移位位数，因为最多移31位，故用5位即可。

指令系统实例——RISC-V基础整数指令集（RV32I）

举例：请给出C语句“int x=8191;”对应的RISC-V机器级代码。

解：8191的机器数为：0000 0000 0000 0000 0001 1111 1111 1111

lui rd, imm20：将立即数imm20存到rd寄存器高20位，低12位为0。该指令和“addi rd, rs1, imm12”结合，可以实现对一个32位变量赋初值。

“int x=8191;”对应的RISC-V机器指令和汇编指令如下，对不对？

0000 0000 0000 0000 0001 00101 0110111 lui x5, 1 #R[x5]← 0000 1000H

1111 1111 1111 00101 000 00101 0010011 addi x5, x5,-1#R[x5]←R[x5]+SEXT[FFFH]

不对！因为低12位中第一位为1，addi按符号扩展相加！结果为4095。

可利用addi符号扩展特性进行调整！因为 imm12范围为-2048~2048，**故可用lui先装入一个距离目标常数小于2048的数，再通过 addi 进行 加 或 减（imm12为负时）来调整！**

这里 8191=8192-1，故可先装入8192，再用 addi 减1（加全1）！

0000 0000 0000 0000 0010 00101 0110111 lui x5, 2 #R[x5]← 0000 2000H

1111 1111 1111 00101 000 00101 0010011 addi x5, x5,-1#R[x5]←R[x5]+SEXT[FFFH]

指令系统实例——RISC-V基础整数指令集（RV32I）

R型指令共10条

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	R sltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and

操作码opcode：都是0110011，其功能由funct3指定，而当funct3=000、101时，再由funct7区分是加（add）还是减（sub）、逻辑右移（srl）还是算术右移（sra）。

rs1、rs2、rd：5位通用寄存编号，共32个；两个源操作数分别在rs1和rs2寄存器中，结果存rd。
sll：逻辑左移指令，无算术左移指令。因逻辑左移和算术左移结果完全相同！

指令系统实例——RISC-V基础整数指令集（RV32I）

4条比较指令：带符号小于（slt、slti）、无符号小于（sltu、sltiu）

例如，“**sltiu rd, rs1, imm12**”功能为：将rs1内容与imm12符号扩展结果按无符号整数比较，若小于，则1存入rd中；否则，0存入rd中。

举例：假定变量x、y和z都是long long型，占64位，x的高、低32位分别存放在寄存器x13、x12中；y的高、低32位分别存放在寄存器x15、x14中；z的高、低32位分别存放在寄存器x11、x10中，请写出C语句“z=x+y;”对应的32位字长RISC-V机器级代码。

解：可通过sltu指令将低32位的进位加入到高32位中。

```
0000000 01110 01100 000 01010 0110011 add x10,x12,x14 #R[x10]←R[x12]+R[x14]

0000000 01100 01010 011 01011 0110011 sltu x11,x10,x12 #若R[x10]<R[x12]，则
                                                        #  R[x11]←1（若和比加数小，则一定有进位）

0000000 01111 01101 000 10000 0110011 add x16,x13,x15 #R[x16]←R[x13]+R[x15]

0000000 10000 01011 000 01011 0110011 add x11,x11,x16 #R[x11]←R[x11]+R[x16]
```

指令系统实例——RISC-V基础整数指令集（RV32I）

控制转移类指令

31 25	24 20	19 15	14 12	11 7	6 0	
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu

J 型：jal 功能为: PC←PC+SEXT[imm[20:1]<<1]；R[rd]←PC+4
rd指定为x1时可实现过程调用；指定为x0时，可实现无条件跳转。

I 型：jalr功能为: PC←R[rs1]+SEXT[imm[12]；R[rd]←PC+4
指令“jalr x0,x1,0”可实现过程调用的返回。

B型：皆为分支指令，其中，bltu、bgeu分别为无符号数比较小于、大于等于转移。
转移目标地址=PC+SEXT[imm[12:1]<<1]

<<1: 指令地址总是2的倍数（RV32G、RV32C指令分别为4、2字节长）

指令系统实例——RISC-V基础整数指令集（RV32I）

举例：若int型变量x、y、z分别存放在寄存器x5、x6、x7中，写出C语句“z=x+y;”对应的RISC-V机器级代码，要求检测是否溢出。

解：当x、y为int类型时，若“y<0且x+y≥x”或者 “y≥0且x+y<x” ，则x+y溢出。可通过slti指令对y与0进行比较。

```
0000000 00110 00101 000 00111 0110011  add x7,x5,x6 #R[x7]←R[x5]+R[x6]
0000 0000 0000 00110 010 11100 0010011 slti x28,x6,0  #若R[x6]<0，则R[x28]←1
0000000 00101 00111 010 11101 0110011  slt x29,x7,x5 #若R[x7]<R[x5] 则R[x29]←1
0000010 11101 11100 001 10000 0110011  bne x28,x29, overflew #若R[x28]≠R[x29]
                                                              #则转溢出处理
  ……
overflew:
```

imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011

假定标号为overflew的指令与“bne x28, x29, overflew”之间相距20条指令，每条指令4字节，则“bne x28, x29, overflew”**指令中的偏移量应为80，因此，指令中的立即数为40=0000 0010 1000B**，按照B-型格式，该指令的机器码为“0000001 11101 11100 001 01000 1100011”。

指令系统实例——RISC-V基础整数指令集（RV32I）

存储访问指令

31 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw

I 型：5条取数（Load）指令。功能: R[rd]←M[R[rs1]+SEXT[imm[12]]。

lbu、lhu：分别为无符号字节、半字取，取出数据按0扩展为32位，装入rd

S型：3条存数（Store）指令。功能: M[R[rs1]+SEXT[imm[12]]←R[rs2].

sb、sh: 分别将rs2寄存器中低8、16位写入存储单元中。

汇编形式：存储地址可写成imm12(rs1)。

指令系统实例——RISC-V基础整数指令集（RV32I）

系统控制类指令

31 25	24 20	19 15	14 12	11 7	6 0	
0000	pred	succ	00000	000	00000	0001111	I fence
0000	0000	0000	00000	001	00000	0001111	I fence.i
000000000000			00000	00	00000	1110011	I ecall
000000000000			00000	000	00000	1110011	I ebreak
csr			rs1	001	rd	1110011	I csrrw
csr			rs1	010	rd	1110011	I csrrs
csr			rs1	011	rd	1110011	I csrrc
csr			zimm	101	rd	1110011	I csrrwi
csr			zimm	110	rd	1110011	I csrrsi
csr			zimm	111	rd	1110011	I csrrci

fence：RISC-V架构在不同硬件线程之间使用宽松一致性模型，fence和fence.i 两条屏障指令，用于保证一定的存储访问顺序。

ecall和ebreak：陷阱（trap）指令，也称自陷指令，主要用于从用户程序陷入到操作系统内核（ecall）或调试环境（ebreak）执行，因此也称为环境（Environment）类指令。

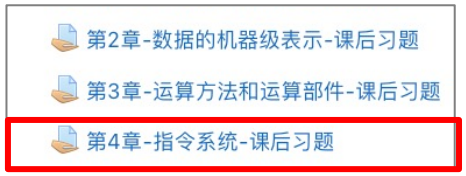
csrxxx: 6条csr指令用于设置和读取相应的控制状态寄存器（CSR）。

指令系统实例——RISC-V可选的扩展指令集

- **标准扩展指令集**
  - RV32I基础指令集之上，可标准扩展RV32M、RV32F/D、RV32A，以形成32位架构合集RV32IMAFD，也称为RV32G
  - RV32G基础上，对每个指令集进行调整和添加，可形成64位架构RV64G，原先在RV32G中处理的数据将调整为64位。但为了支持32位数据操作，每个64位架构指令集中都会添加少量32位数据处理指令。
- **RISC-V扩展集包括**
  - 针对64位架构需要，在47条RV32I指令基础上，增加12条整数指令（+RV64I），包括6条32位移位指令、3条32位加减运算指令、两条64位装入（Load）指令和1条64位存储（Store）指令，故RV64I共59条指令。
  - 针对乘除运算需要，提供了32位架构乘除运算指令集RV32M中的8条指令，并在此基础上增加了4条RV64M专用指令（+RV64M）
  - 针对浮点数运算的需要，提供了32位架构的单精度浮点处理指令集RV32F和双精度浮点处理指令集RV32D，并在此基础上分别增加了RV64F和RV64D专用指令集（+RV64F）和（+RV64D）。
  - 针对事务处理和操作原子性的需要，提供了32位架构原子操作指令集RV32A以及RV64A专用指令集（+RV64A）。关于事务处理和原子性操作问题的说明可参考第8章。
- 向量处理指令集RVV、未来可选扩展指令集RVB、RVE、RVH、……

课程习题（作业）——截止日期：10月21日晚23:59

- **课本124-128页**：第3、4、6、8、9、10、11、12、13、15、17题
- 提交方式：https://selearning.nju.edu.cn/（教学支持系统）



第4章-指令系统-课后习题

课本124-128页：第3、4、6、8、9、10、11、12、13、15、17题

- 命名：学号+姓名+第*章。
- 若提交遇到问题请及时发邮件或在下一次上课时反馈。

课程习题（作业）——截止日期：10月21日晚23:59

3. 假定某计算机中有一条转移指令，采用相对寻址方式，共占两字节，第一字节是操作码，第二字节是相对位移量(用补码表示)，CPU 每次从内存只能取一字节。假设执行到某转移指令时 PC 的内容为 200，执行该转移指令后要求转移到 100 开始的一段程序执行，则该转移指令第二字节的内容应该是多少？

4. 假设地址为 1200H 的内存单元中的内容为 12FCH，地址为 12FCH 的内存单元的内容为 38B8H，而 38B8H 单元的内容为 88F9H。说明以下各情况下操作数的有效地址是多少？

(1) 操作数采用变址寻址，变址寄存器的内容为 252，指令中给出的形式地址为 1200H。

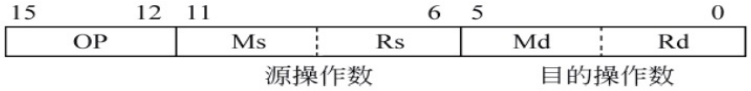
(2) 操作数采用一次间接寻址，指令中给出的地址码为 1200H。

(3) 操作数采用寄存器间接寻址，指令中给出的寄存器编号为 8，8 号寄存器的内容为 1200H。

6. 某计算机指令系统采用定长指令字格式，指令字长 16 位，每个操作数的地址码长 6 位。指令分二地址、单地址和零地址 3 类。若二地址指令有 k2 条，零地址指令有 k0 条，则单地址指令最多有多少条？

课程习题（作业）——截止日期：10月21日晚23:59

8. 某计算机字长为 16 位，主存地址空间大小为 128KB，按字编址。采用单字长定长指令格式，指令各字段定义如下：



转移指令采用相对寻址方式，相对位移量用补码表示，寻址方式定义如下表所示。

Ms/Md	寻址方式	助记符	含义
000B	寄存器直接	Rn	操作数＝R[Rn]
001B	寄存器间接	(Rn)	操作数＝M[R[Rn]]
010B	寄存器间接、自增	(Rn)＋	操作数＝M[R[Rn]]，R[Rn]←R[Rn]＋1
011B	相对	D(Rn)	转移目标地址＝PC＋R[Rn]

注：M[x]表示存储器地址 x 中的内容，R[x]表示寄存器 x 中的内容。

请回答下列问题：

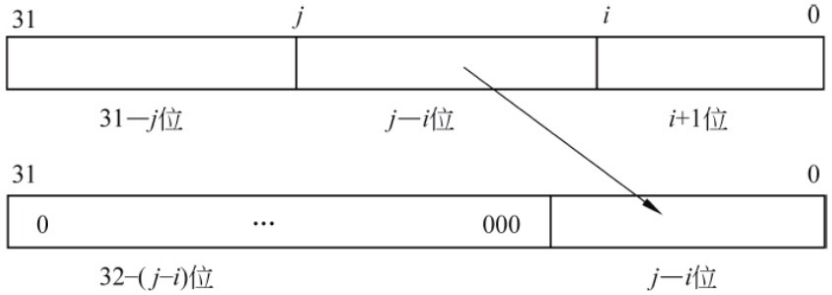
(1) 该指令系统最多可有多少条指令？最多有多少个通用寄存器？存储器地址寄存器(MAR)和存储器数据寄存器(MDR)至少各需要多少位？

(2) 转移指令的目标地址范围是多少？

(3) 若操作码 0010B 表示加法操作(助记符为 add)，寄存器 R4 和 R5 的编号分别为 100B 和 101B，R4 的内容为 1234H，R5 的内容为 5678H，地址 1234H 中的内容为 5678H，地址 5678H 中的内容为 1234H，则汇编语句“add(R4)，(R5)＋”(逗号前为第一源操作数，逗号后为第二源操作数和目的操作数)对应的机器码是什么(用十六进制表示)？该指令执行后，哪些寄存器和存储单元的内容会改变？改变后的内容是什么？

课程习题（作业）——截止日期：10月21日晚23:59

9. 有些计算机提供了专门的指令，能从一个 32 位寄存器中抽取其中任意一个位串置于另一个寄存器的低位有效位上，并在高位补 0，如下图所示。MIPS 指令系统中没有这样的指令，请写出最短的一个 MIPS 指令序列来实现这个功能，要求 $i=5, j=22$，操作前后的寄存器分别为 $s0 和 $s2。



10. 以下程序段是某个过程对应的指令序列。入口参数 int a 和 int b 分别置于 $a0 和 $a1 中，返回参数是该过程的结果，置于 $v0 中。要求为以下 MIPS 指令序列加注释，并简单说明该过程的功能。

```
        add     $t0, $zero, $zero
loop:   beq     $a1, $zero, finish
        add     $t0, $t0, $a0
        sub     $a1, $a1, 1
        j       loop
finish: addi    $t0, $t0, 100
        add     $v0, $t0, $zero
```

课程习题（作业）——截止日期：10月21日晚23:59

11. 下列指令序列用来对两个数组进行处理，并产生结果存放在 $v0 中，两个数组的基地址分别存放在 $a0 和 $a1 中，数组长度分别存放在 $a2 和 $a3 中。要求为以下 MIPS 指令序列加注释，并简单说明该过程的功能。假定每个数组有 2500 个字，其数组下标为 0～2499，该指令序列运行在一个时钟频率为 2GHz 的处理器上，add、addi 和 sll 指令的 CPI 为 1；lw 和 bne 指令的 CPI 为 2，则最坏情况下运行所需时间是多少秒？

```
        sll    $a2, $a2, 2
        sll    $a3, $a3, 2
        add    $v0, $zero, $zero
        add    $t0, $zero, $zero
outer:  add    $t4, $a0, $t0
        lw     $t4, 0($t4)
        add    $t1, $zero, $zero
inner:  add    $t3, $a1, $t1
        lw     $t3, 0($t3)
        bne    $t3, $t4, skip
        addi   $v0, $v0, 1
skip:   addi   $t1, $t1, 4
        bne    $t1, $a3, inner
        addi   $t0, $t0, 4
        bne    $t0, $a2, outer
```

课程习题（作业）——截止日期：10月21日晚23:59

12. 用一条 MIPS 指令或最短的 MIPS 指令序列实现以下 C 语言语句：b=25|a。假定编译器将 a 和 b 分别分配到 $t0 和 $t1 中。如果把 25 换成 65536，即 b=65536|a，则用 MIPS 指令或指令序列如何实现？

13. 以下程序段是某个过程对应的 MIPS 指令序列，其功能为复制一个存储块数据到另一个存储块中，存储块中每个数据的类型为 float，源数据块和目的数据块的首地址分别存放在 $a0 和 $a1 中，复制的数据个数存放在 $v0 中，作为返回参数返回给调用过程。假定在复制过程中遇到 0 就停止复制，最后一个 0 也需要复制，但不被计数。已知程序段中有多个错误，请找出它们并修改之。

```
        addi    $v0, $zero, 0
loop:   lw      $v1, 0($a0)
        sw      $v1, 0($a1)
        addi    $a0, $a0, 4
        addi    $a1, $a1, 4
        beq     $v1, $zero, loop
```

课程习题（作业）——截止日期：10月21日晚23:59

15. 以下 C 语言程序段中有两个函数 sum_array 和 compare，假定 sum_array 函数先被调用，全局变量 sum 分配在寄存器 $s0 中。要求按照 MIPS 过程调用协议写出每个函数对应的 MIPS 汇编语言程序，并画出每个函数调用前、后栈中的状态、帧指针和栈指针的位置。

```
int sum=0;

int sum_array(int array[], int num)
{
    int i;
    for(i=0; i<num; i++)
        if compare(num, i+1) sum+=array[i];
    return sum;
}

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

课程习题（作业）——截止日期：10月21日晚23:59

17. 假定编译器将 a 和 b 分别分配到 t0 和 t1 中，用一条 RV32I 指令或最短的 RV32I 指令序列实现以下 C 语言语句：b=31&a。如果把 31 换成 65535，即 b=65535&a，则用 RV32I 指令或指令序列如何实现？对比第 12 题 RV32I 与 MIPS 之间在立即数处理上有何不同？

提问

Q & A

殷亚凤
智能软件与工程学院
苏州校区南雍楼东区225
yafeng@nju.edu.cn，https://yafengnju.github.io/

南京大学 NANJING UNIVERSITY