Lecture 20: MIPS Assembly Language II

Example：过程调用

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

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

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

i是全局静态变量

array数组是局部变量

set_array是调用过程
compare是被调用过程

compare是调用过程
sub是被调用过程

问题：过程调用对应的机器代码如何表示？

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

Procedure Call and Stack(过程调用和栈)

- 过程调用的执行步骤（假定过程P调用过程Q）：

compare (num, i);

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

在调用过程P中完成

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

在被调用过程Q中完成

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

一般用“栈”来传递参数、保存返回地址、临时存放过程的局部变量等。为什么？

便于递归调用！

栈(Stack)的概念

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

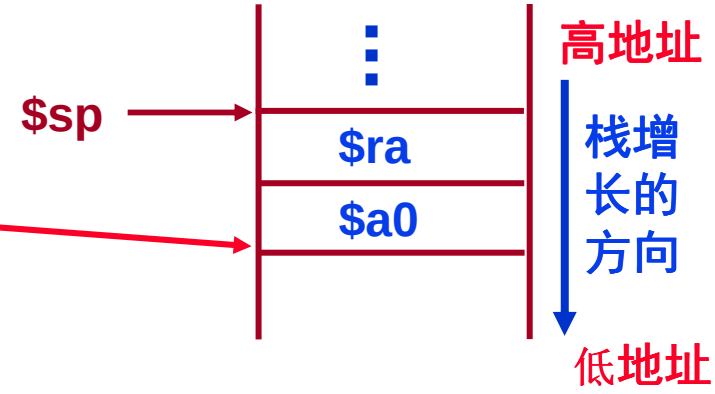
    （有些处理器有专门的push/pop指令，能自动调整栈指针。如x86）
  - 栈生长方向

    从高➡低地址“增长”，而取数/存数的方向是低➡高地址（大端方式）

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

例：若将返回地址$ra和参数$a0保存到栈，则指令序列为：

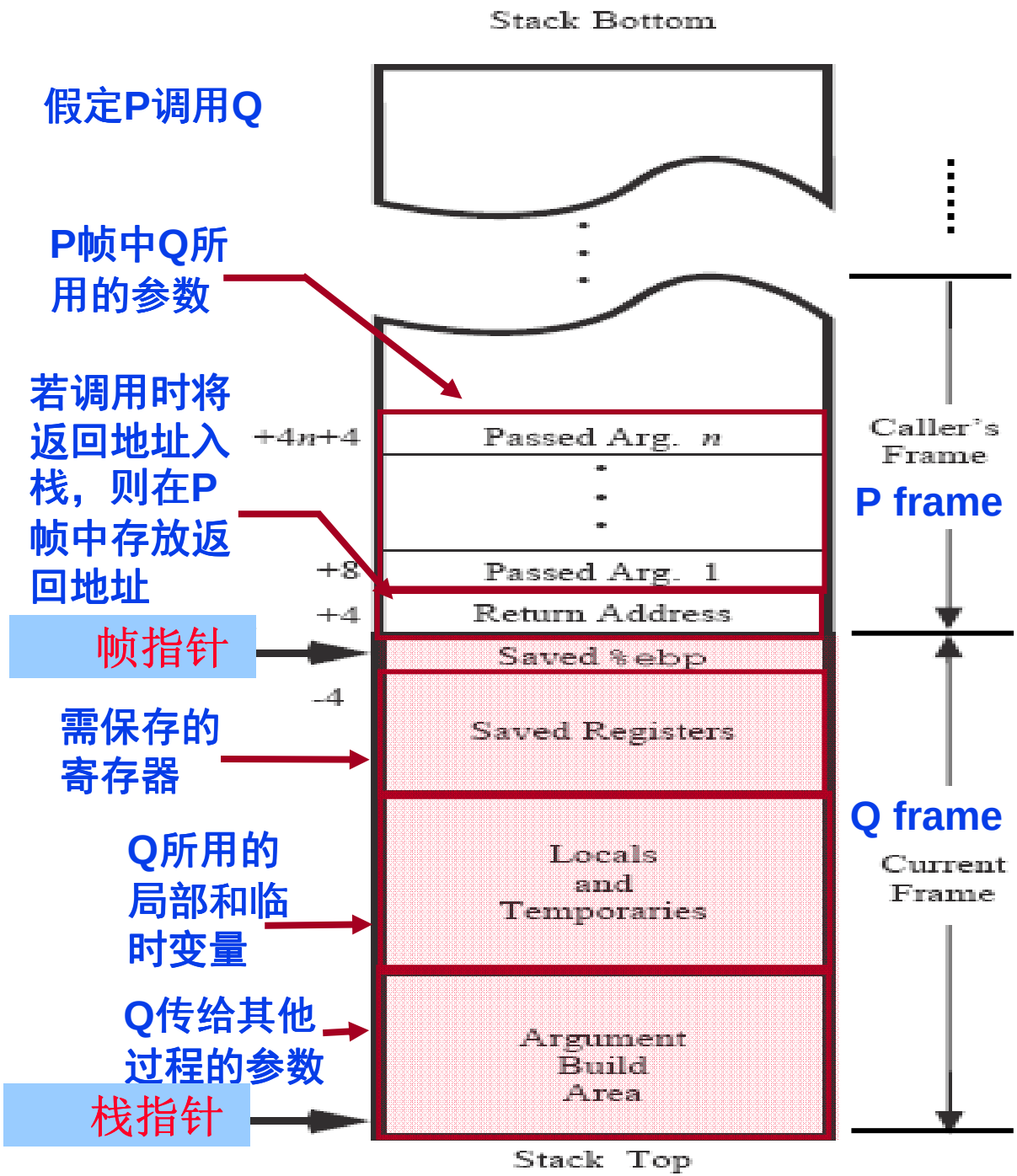
```
sub  $sp, $sp, 8
sw   $ra, 4($sp)
sw   $a0, 0($sp)
```



栈帧的概念

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

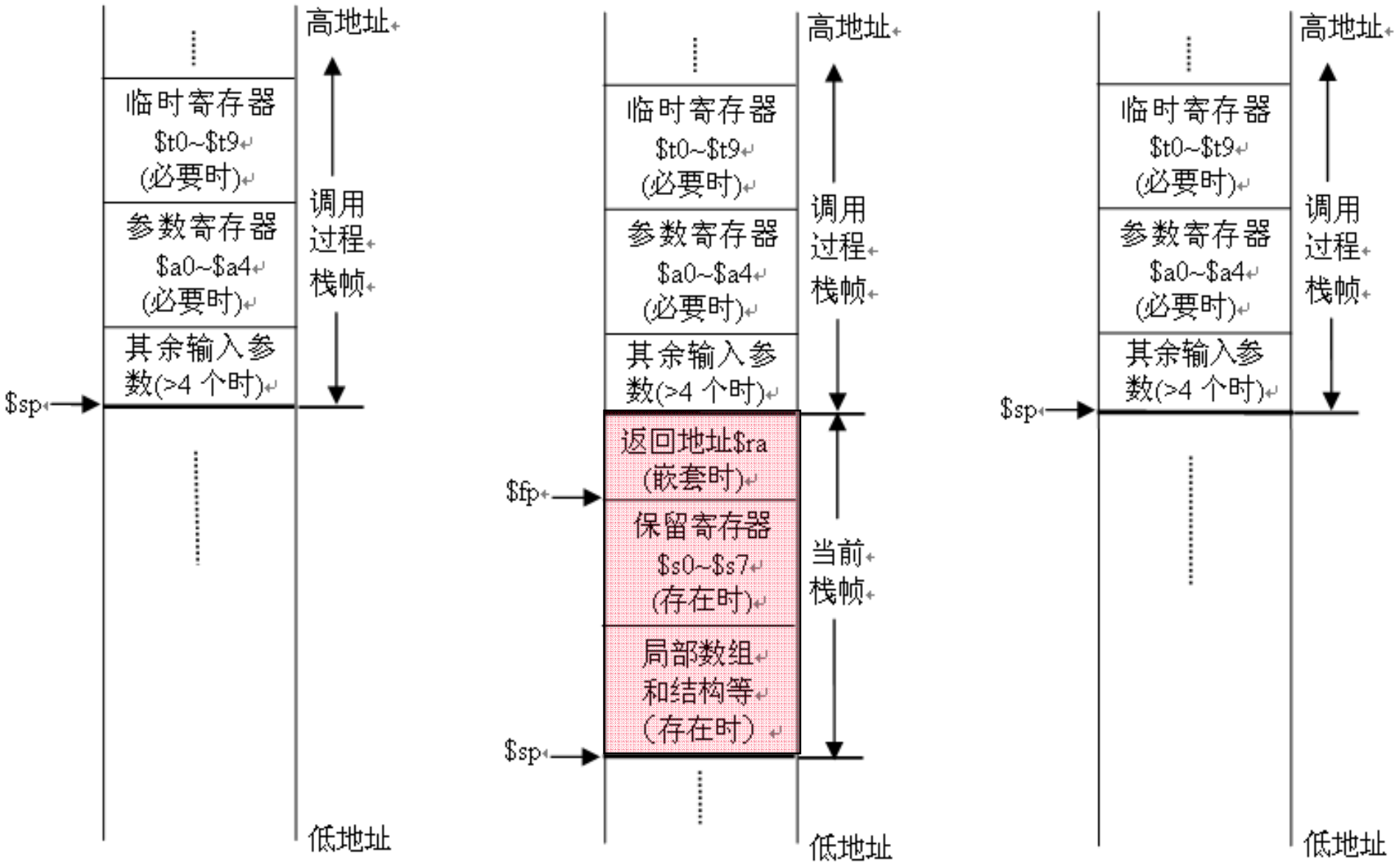
X86的栈帧情况如右图所示



MIPS中的过程调用（假定P调用Q）

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

过程调用时MIPS中的栈和栈帧的变化



Example in C: swap

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

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

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

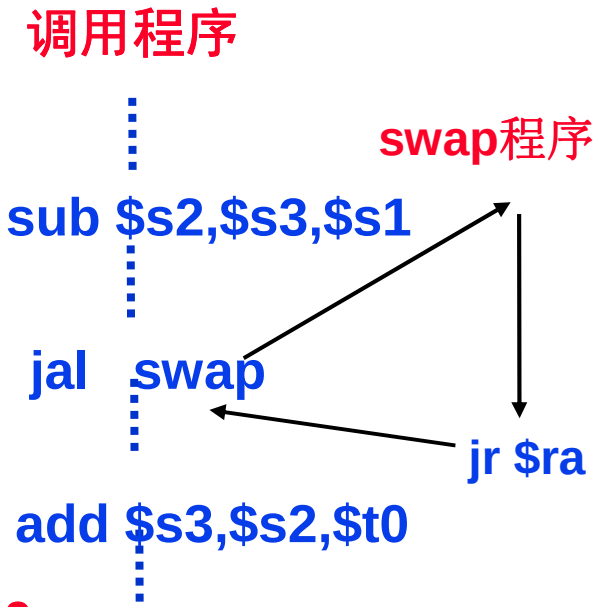
```
sll     $s2, $a1, 2      ; mulitply k by 4
addu    $s2 $s2, $a0     ; address of v[k]
lw      $t0, 0($s2)      ; load v[k]
lw      $s3, 4($s2)      ; load v[k+1]
sw      $s3, 0($s2)      ; store v[k+1] into v[k]
sw      $t0, 4($s2)      ; store old v[k] into v[k+1]
```

在调用过程中用指令“jal swap” 进行swap调用

jal --- jump and link (跳转并链接)

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

goto swap



问题1：若swap中不保存$s2，则会发生什么情况？

caller中$s2的值被破坏！须在swap中保存$s2

问题2：若swap中不保存$t0，则会发生什么情况？

$t0约定由caller保存，故无须在swap栈帧中保存$t0

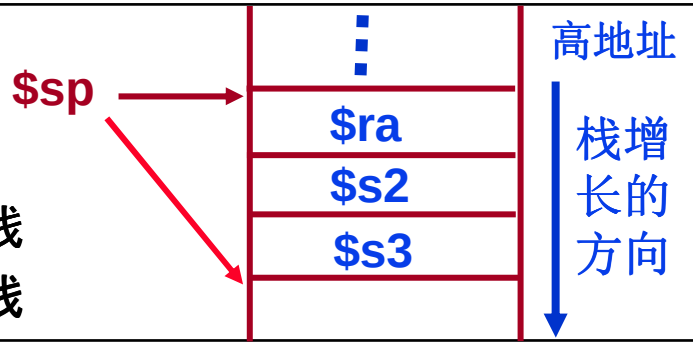
swap: MIPS中的一个过程示例

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

    ....

    lw     $s3, 0($sp)       ; 恢复$s3
    lw     $s2, 4($sp)       ; 恢复$s2
    lw     $31, 8($sp)       ; 恢复$31（$ra）
    addi   $sp,$sp, 12       ; 退栈
    jr     $31               ; 从swap返回到调用过程
```

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



问题：是否一定要将返回地址（$31）保存到栈帧中？

如果swap是叶子过程，则无需保存返回地址到栈中，为什么？

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

$ra的内容不会被破坏！

Example：过程调用

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

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

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

i是全局静态变量

array数组是局部变量

set_array是调用过程
compare是被调用过程

compare是调用过程
sub是被调用过程

问题1：编译器如何为全局变量和局部变量分配空间？

问题2：执行set_array的结果是什么？

过程调用时的变量分配

- 全局静态变量一般分配到寄存器或在R/W存储区（数组或结构等）。

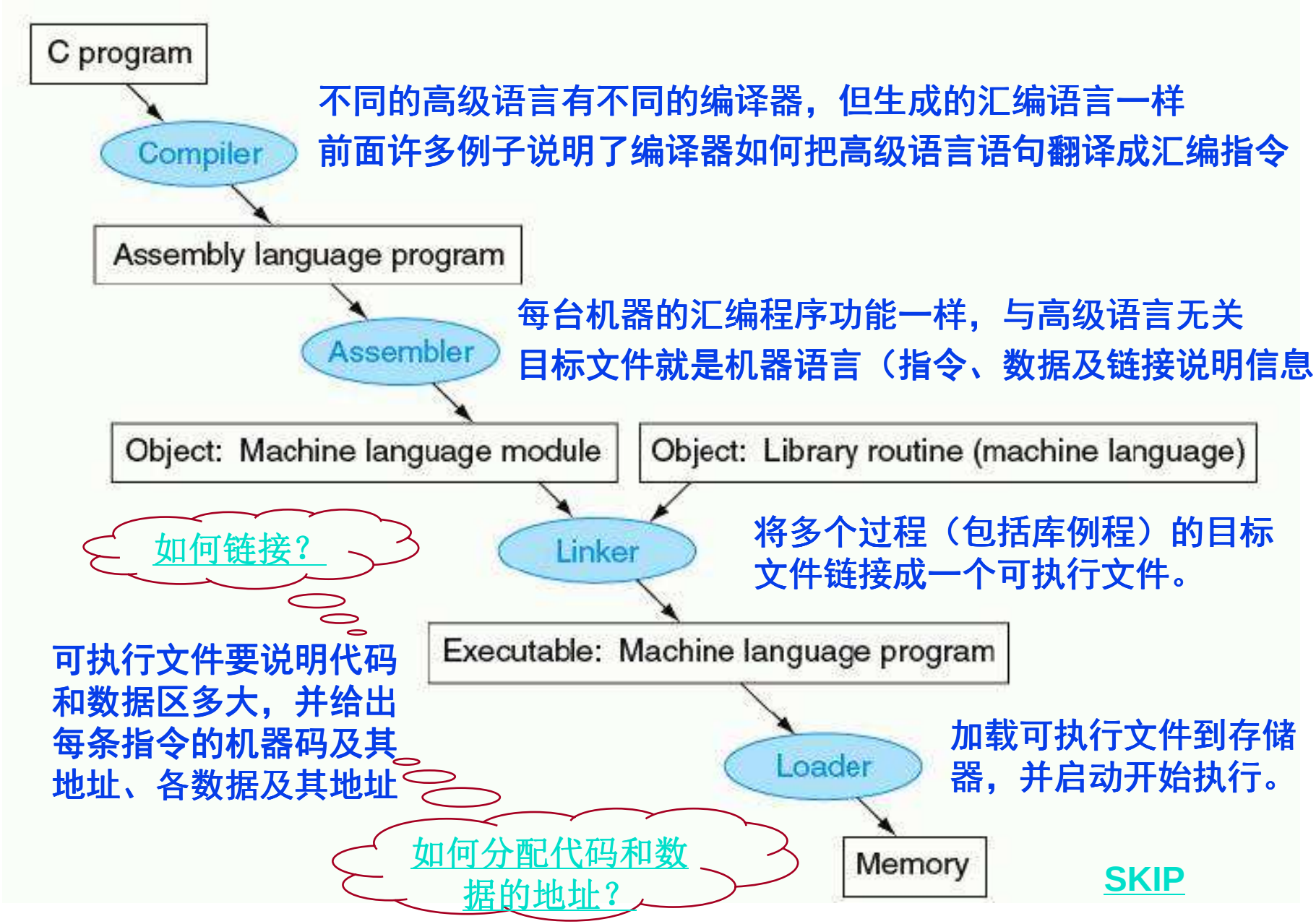
  该例中只有一个简单变量i，假定分配给$s0。无需保存和恢复！

- 为减少指令条数，并减少访问内存次数。在每个过程的过程体中总是先使用临时寄存器$t0~$t9，临时寄存器不够或者某个值在调用过程返回后还需要用，就使用保存寄存器$s0~$s7。

- 过程set_array的入口参数为num，没有返回参数，有一个局部数组，被调用过程为compare，因此，其栈帧中除了保留所用的保存寄存器外，必须保留返回地址（是否保存$fp要看具体情况，如果确保后面都不用到$fp，则可以不保存，但为了保证$fp的值不被后面的过程覆盖，通常情况下，应该保存$fp的值），并给局部数组预留4×10=40个字节的空间。

- 从过程体来看，从compare返回后还需要用到数组基地址，故将其分配给$s1。因此要用到的保存寄存器有两个：$s0和$s1，但只需要保存$s1。另外加上返回地址$ra、帧指针$fp、局部数组，其栈帧空间最少为3×4+40=52B。

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

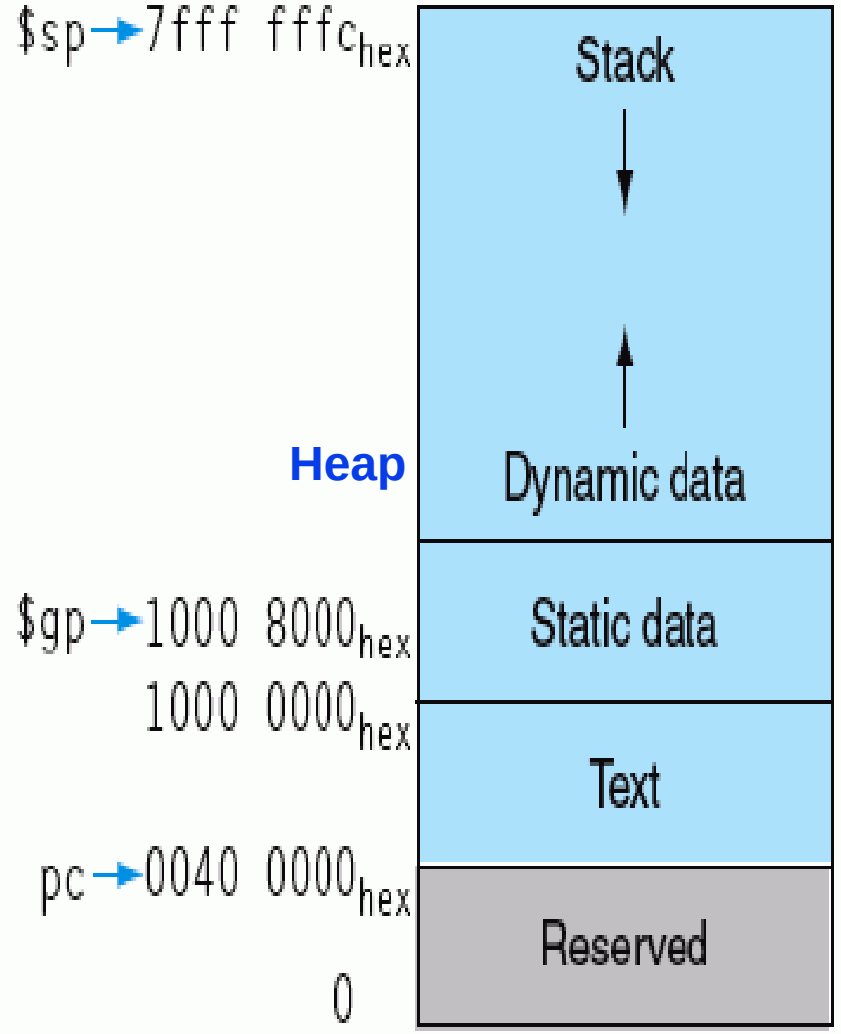
- 每个MIPS程序都按如下规定进行存储器分配
- 每个可执行文件都按如下规定给出代码和数据的地址

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



BACK

目标文件的链接（自学）

过程A和过程B分别编译、汇编成目标文件，链接后生成一个可执行文件

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

目标文件的链接（自学）

过程A和过程B分别编译、汇编成目标文件，链接后生成一个可执行文件

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

目标文件的链接（自学）

过程A和过程B分别编译、汇编成目标文件，链接后生成一个可执行文件

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

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

BACK

MIPS指令中位的指定和逻辑运算（自学）

逻辑数据表示

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

逻辑运算

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

位的指定

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

MIPS中的移位指令（sll / srl）

例：srl $t2,$s0,8

$s0右移8位后送$t2

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

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

MIPS指令中常数的指定（自学）

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

例1：i=i+4;　Assuming variable i ~ $1

则：　addi $1, $1, 4

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

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

如果常数的值用16位无法表示，怎么办？

用lui指令把高16位送到寄存器的高16位，再把低16位加到该寄存器中。

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

则：　lui $3, 61

　　addi $3, $3, 8

MIPS指令中如何表示文本字符串（自学）

- 有些情况下，程序需要处理文本。例如：
  - 西文文本由ASCII码字符构成字符串
  - Java等语言使用Unicode编码构成字符串
  - 汉字文本使用的汉字编码字符构成字符串
- 字符串的表示
  - 由一个个字符组成，长度不定。有三种表示方式：
    - » 字符串的首字节记录长度
    - » 用其他变量来记录长度（即：用“struc”类型来描述）
    - » 字符串末尾用一个特殊字符表示。
      如：C语言用字符（NULL）来标记字符串结束
- 如何在指令中表示字符
  - ASCII字符串，每个字符由8位组成，用 “lb/sb”指令存/取一个字节
  - Unicode及汉字字符都有16位，用 “lh/sh”指令存/取两个字节

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

则：

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

SKIP

数据类型和MIPS指令的对应（自学）

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

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

本讲小结

- 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
- 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
  - 运算表达式 / If语句 / 循环 / 数组访问 / 过程 / 堆栈 / 栈帧
- 其他指令系统：PowerPC、80x86
- CISC vs. RISC