Lecture 4 指令系统体系结构2

0. 大纲

指令介绍

- 算术指令：add, sub, ...
- 数据传输指令：lw, sw, lh, sh，...
- 逻辑指令：and, or，...
- 条件分支指令：beq, bne，...

过程调用/返回

1. 算数指令

指令	示例	含义	注释
加法	add $s1, $s2, $s3	s1 = s2 + s3	三个寄存器操作数
减法	sub $s1, $s2, $s3	s1 = s2 - s3	三个寄存器操作数
立即数加法	addi $s1, $s2, 20	s1 = s2 + 20	用于加常数数据

指令很简单：固定的操作数（不像C）

一行C代码将会被转换成多行汇编代码

addi

将一个寄存器里的数和一个立即数相加

```
addi $s0, $zero, 1000   #程序有一个基址1000，将它存储在$s0里
                        #$zero是一个一直等于0的寄存器
addi $s1, $s0,   0      #$s1这是变量a的地址
addi $s2, $s0,   4      #$s2这是变量b的地址
addi $s3, $s0,   8      #$s3这是变量c的地址
addi $s4, $s0,   12     #$s4这是变量d[0]的地址
```

例题1 加法的使用

将C语言中的赋值语句编译成MIPS

```
a = b + c;
d = a - e;
```

MIPS代码：

```
add a, b, c # a = b + c
sub d, a, e # d = a - e
```

例题2 加减运算

将C语言中的赋值语句编译成MIPS

```
f = (g + h) - (i + j)
```

MIPS代码：

为了保存中间结果，需要将其临时变量存储在某一地方

```
add t0, g, h # t0 = g + h
add t1, i, j # t1 = i - j
sub f, t0, t1 # f = t0 - t1
```

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

2. 数据传输指令

指令	示例	含义	注释
取字	lw $s1, 20($s2)	s1 = Memory[s2 + 20]	字从内存中放入寄存器
取半字	lh $s1, 20($s2)	s1 = Memory[s2 + 20]	半字从内存中放入寄存器
取无符号半字	lhu $s1, 20($s2)	s1 = Memory[s2 + 20]	半字从内存中放入寄存器
取字节	lb $s1, 20($s2)	s1 = Memory[s2 + 20]	字节从内存中放入寄存器

指令	示例	含义	注释
取无符号字节	lbu $s1, 20($s2)	s1 = Memory[s2 + 20]	字节从内存中放入寄存器
存字	sw $s1, 20($s2)	Memory[s2 + 20] = s1	字从寄存器存入内存
存半字	sh $s1, 20($s2)	Memory[s2 + 20] = s1	半字从寄存器存入内存
存字节	sb $s1, 20($s2)	Memory[s2 + 20] = s1	字节从寄存器存入内存

将数据从存储器复制到寄存器的数据传输指令通常叫做取数 load 指令，

load指令

将数据从存储器复制到寄存器的数据传送指令

```
lw $t0, <内存地址/标签> # 载入字
lh $t0, <内存地址/标签> # 载入半字
lb $t0, <内存地址/标签> # 载入字节
```

```
lw <寄存器名字> <在方括号中添加到寄存器的常数>(<内存地址>)
```

```
lw $t0 8($s0)
```

`$s0`里面存的**地址 + 8**加载进寄存器`$t0`里面

起始地址成为**基址**，存放基址的寄存器成为**基址寄存器 base register**

数据传送指令中的常量称为**偏移量offset**

$$address = base\ \ address + offset$$

store指令

```
sw $t0, <内存地址> #存储字
```

例题3 数据传输指令

将C语言中的赋值语句编译成MIPS

```
d[3] = d[2] + a;
```

假设 a 存储在 $s3 里面， $d[\,]$ 存储在 $4 里面

MIPS代码：

```
# 像以前一样使用addi指令
lw $t0, 8($s4) # d[2] 载入 $t0
lw $t1, 0($s1) # a 载入 $t1
add $t0, $t0, $t1 # 加和，并把和存在$t0
sw $t0, 12($s4) # $t0 存入 d[3]
```

例题4 数据传输指令

假设 A 是一个含有100字的数组，编译器将 $s1， $s2 依次分配给变量 g，h

又设数组 A 的起始地址存放在寄存器 $s3 中

C代码：

```
g = h + A[8];
```

MIPS代码：

```
lw $t0, 32($s3) # 载入A[8]
add $s1, $s2, $t0
```

3. 逻辑指令

指令	示例	含义	注释
与	and $s1, $s2, $s3	s1 = s2 & s3	寄存器操作数按位与
或	or $s1, $s2, $s3	s1 = s2 | s3	寄存器操作数按位或
或非	nor $s1, $s2, $s3	s1 = ~(s2 | s3)	寄存器操作数按位或非
立即数与	andi $s1, $s2, 20	s1 = s2 & 20	和常数按位与

指令	示例	含义	注释
立即数或	ori $s1, $s2, 20	s1 = s2 | 20	和常数按位或
逻辑左移	sll $s1, $s2, 10	s1 = s2 << 10	根据常数逻辑左移
逻辑右移	srl $s1, $s2, 10	s1 = s2 >> 10	根据常数逻辑右移

逻辑运算	C	JAVA	MIPS
逻辑左移	<<	<<	sll
逻辑右移	>>	>>>	srl
按位与	&	&	and, andi
按位或	|	|	or, ori
按位非	~	~	nor

- 逻辑左移 i bit 就相当于乘以 2^i
- 逻辑右移 i bit 就相当于除以 2^i

MIPS指令集没有设计支持NOR立即数的版本

4. 决策指令

指令	示例	含义	注释
相等时跳转	beq $s1, $s2 <标签>	if(s1 == s2) go to <标签>	相等检测
不相等时跳转	bnq $s1, $s2 <标签>	if(s1 != s2) go to <标签>	不相等检测
小于时置位	slt $s1, $s2 $3	if(s2 < s3) s1 = 1;else s1 = 0	比较是否小于
无符号数比较小于时置位	sltu $s1, $s2 $3	if(s2 < s3) s1 = 1;else s1 = 0	比较是否小于无符号数
无符号数比较小于立即数时置位	slti $s1, $s2, 20	if(s2 < 20) s1 = 1;else s1 = 0	比较是否小于常数
无符号数比较小于无符号立即数置为	sltiu $s1, $s2, 20	if(s2 < 20) s1 = 1;else s1 = 0	比较是否小于无符号常数

例题5 实现 if-else

将C语言编译成MIPS

f、g、h、i、j 都是变量，依次对应于 $s0-$s4

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

MIPS代码：

```
bne $s3, $s4, Else # 如果 s3 != s4 去 Else
    add $s0, $s1, $s2
    j Exit # 跳出 if(...)
Else:
    sub $s0, $s1, $s2
Exit:
    ... # 继续后面的代码
```

例题6 实现while循环

将C语言编译成MIPS

假设 i 和 k 存在寄存器 $s3 和 $s5中，数组 $save$ 存在寄存器 $s6中

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

MIPS代码：

```
loop:
    sll $t1, $s3, 2 # Temp reg $t1 = i * 4
    add $t1, $t1, $s6 # $t1 = address of save[i]
    lw $t0, 0($t1) # Temp reg $t0 = save[i]
    bne $t0, $s5, Exit # goto Exit if save[i] ≠ k
        addi $s3, $s3, 1 # i = i + 1
        j loop # go to loop
Exit:
    ......
```

例题7 实现for语句

将C语言编译成MIPS

假设 i 和 k 存在寄存器 $s3 和 $s5 中，10放在寄存器 $s4 中

```
for(int i = 0; i < 10; i++){
    k += i;
}
```

MIPS代码：

```
loop:
    beq $s3, $s4, Exit # if s3 == 10 Exit
        add $s5, $s5, $s3 # else k += i
        addi $s3, $s3, 1 # i += 1
        j loop # continue loop
Exit:
    ......
```

例题8 实现case/switch语句

将多个指令序列分支的地址编码成为一行表，即**转移地址表**

转移地址表是由代码中标签所对应地址构成的数组

5. 过程/函数

过程 process：根据提供的参数执行一定任务的存储的子程序

调用的程序或称为**调用者caller**，然后用 jal X 跳转到过程X，称为**被调用者callee**

指令	示例	含义	注释
跳转	j 2500	go to 10000	跳转到目标地址
跳转到寄存器所指位置	jr $ra	go to $ra	用于switch语句以及过程调用
跳转并链接	jal 2500	$ra = PC + 4 go to 10000	指令中的链接部分指向调用点的地址或连接，允许过程返回到合适的地址，存储到 $ra 里

过程遵循步骤

在过程执行中，程序必须遵循以下几个步骤

1. 将参数放在过程可以访问的位置
2. 将控制交给过程
3. 获取过程需要的存储资源
4. 执行需要的任务
5. 将结果的值放在调用程序可以访问的位置
6. 将控制返回初始点，因为一个过程可能由一个程序中的多个点调用

程序计数器

在存储程序的概念中，会使用一个寄存器来保存**当前运行的指令地址**

这个寄存器通常称为**程序计数器 program counter**，在MIPS体系结构中缩写为PC

寄存器分配

MIPS软件在为过程调用分配32个寄存器时遵循以下约定

- $a0 ~ $a3：用于传递参数的4个参数寄存器
- $v0 ~ $v1：用于返回值的2个值寄存器
- $t0 ~ $t9：10个临时寄存器，在过程调用中**不必被调用者保存**

- $s0 ~ $s7：8个保留寄存器，在调用的过程中必须被保存，**一旦被调用，由被调用者保存和恢复**
- $ra：用于返回起始点的返回地址寄存器
  jal 指令实际上将 PC + 4 保存在寄存器 $ra 里

MIPS中需要被调用者需要维护如下寄存器：

保留	不保留
保存寄存器：$s0 ~ $s7	临时寄存器：$t0 ~ $t9
栈指针寄存器：$sp	参数寄存器：$a0 ~ $a3
返回地址寄存器：$ra	返回值寄存器：$v0 ~ $v1
栈指针以上的栈	栈指针以下的栈

全局指针

静态变量在进入和退出过程中始终存在，在 C 语言中，声明 static 的变量都被视为静态的，其余的变量都被视为动态的，MIPS为了简化静态数据访问，保留了另一个寄存器，称为全局指针 global pointer，即 $gp

栈指针

由于在任务完成后必须消除踪迹，因此调用者使用的任何寄存器都必须恢复到过程调用前所存储的值，换出寄存器最理想的数据结构是**栈 stack**

栈需要一个指针指向栈中最新分配的地址，以指示下一个过程放置换出寄存器的位置，或是寄存器旧值放置的位置

在每次寄存器进行保存或恢复时，栈指针以**字为单位**进行调整

MIPS为栈指针准备了寄存器$sp

- **压栈 push**：向栈中增加元素，栈指针值减小
- **出栈 pop**：从栈中移除元素，栈指针值增大

帧指针

帧指针指向该**帧的第一个字**，而**栈指针指向栈顶**，由于程序在运行中栈指针可能会有改变，所以固定帧指针引用变量会更加简单，我们**可以使用帧指针 $fp 对栈指针进行恢复**

如果调用参数超过4个，**MIPS规定将额外的参数放在栈中 $fp 的上方**，通过$fp在内存中寻址获得其它参数

例题9 实现过程调用

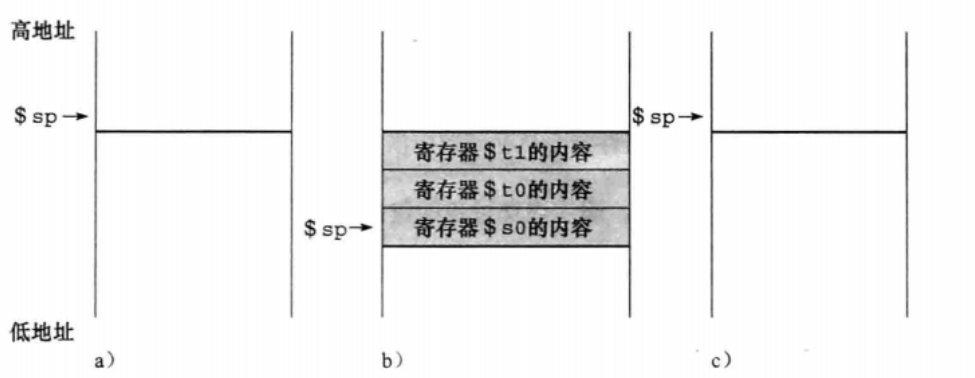
C代码：

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

MIPS代码：

参数变量 g, h, i, j 对应参数寄存器 $a0, $a1, $a2和$a3， f 对应 $s0

```
leaf_example:
    addi $sp, $sp, -12 # 调整栈空间，申请3个项
    sw $t1, 8($sp) # 把寄存器t1里的值先保存下来
    sw $t0, 4($sp) # 把寄存器t2里的值先保存下来
    sw $s0, 0($sp) # 把寄存器s0里的值先保存下来

    add $t0, $a0, $a1 # t0 = a0 + a1
    add $t1, $a2, $a3 # t1 = a2 + a3
    sub $s0, $t0, $t1 # s0 = t0 - t1

    # 为了返回f的值，我们将它复制到一个返回值寄存器中
    add $v0, $s0, $zero # return f (v0 = s0 + 0)

    # 在返回前，恢复使用的寄存器t1,t2,s0的值
    lw $s0, 0($sp) # 还原s0的值
    lw $t0, 4($sp) # 还原t0的值
    lw $t1, 8($sp) # 还原t1的值
    addi $sp, $sp, 12 # 栈指针还原

    jr $ra # 返回调用者
```



在过程调用之前、之中和之后栈指针以及栈的值

- 帧指针$fp指向该帧的第一个字，而栈指针$sp则指向栈顶
- 调用中使用$sp的地址初始化，用$fp来恢复

6. 嵌套过程

不调用其它过程的过程称为**叶过程leaf procedure**

由于递归调用时，当从过程 A 进入过程 B的时候，A 尚未结束任务，所以在寄存器 **$ra** 和**其它参数寄存器的使用上存在冲突**，一个解决方法是将其它所有必须保留的寄存器压栈，到返回时，寄存器会从存储器中恢复，栈指针也会随之调

例题10 实现递归函数

C代码：

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

汇编代码：

参变量n 对应参数寄存器 $a0

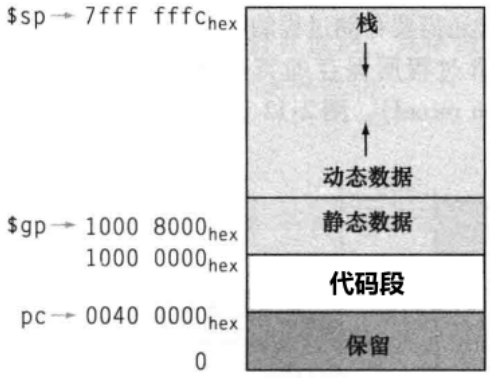
```
fact:
    addi $sp, $sp, -8 # 申请2个栈空间
    sw $ra, 4($sp) # 保存$ra
```

```
    sw $a0, 0($sp) # 保存$a0

    slti $t0, $a0, 1 # if (a0 < 1), t0 = 1
    # 如果n大于等于1，递归
    beq $t0, $zero, L1 # if (t0 == 0) goto L1
    # 如果n小于1的情况
    addi $v0, $zero, $1 # return 1
    addi $sp, $sp, 8 # 释放栈空间
    jr $ra # 跳转回调用者

    L1:
        addi $a0, $a0, -1 # a0 = a0 - 1，作为递归参数
            jal fact
        lw $a0, 0($sp) # 获得参数n
        lw $ra, 4($sp) # 重新存储返回的值
        addi $sp, $sp, 8 # 释放栈空间

        mul $v0, $a0, $v0 # v0 = a0 * v0 = n * fact(n-1)
        jar $ra
```

7. 内存空间

内存的构造（自顶向下）

- **栈空间 Stack**：方法调用传参和返回以及寄存器保留等，**向下生长**
- **堆空间 Dynamic data-Heap**：动态分配数据，在生命周期内增长或减少，**向上生长**
- **静态数据段 Static data segment**：存储常量和静态变量的空间
- **代码段 Text segment**：MIPS机器码的第一部分，包含源文件例程对应的机器码
- **保留段 Reserved**



程序和数据的 MIPS 内存分配。这些地址只是一种软件规定，并非 MIPS 体系结构的一部分。栈指针初始化为 $7fff\ fffc_{16}$，并朝数据段的方向向下增长。在另一端，程序代码（代码段）从地址 $0040\ 0000_{16}$ 开始。静态数据从 $1000\ 0000_{16}$ 开始。然后是动态数据，在 C 中使用 malloc 命令分配，在 Java 中使用 new 命令来分配。动态数据在某一区域中朝着栈的方向向上生长，该区域称为堆。全局指针 $gp 应设置为适当地址以便于访问数据。它初始化为 $1000\ 8000_{16}$，这样通过相对 $gp 的正负 16 位的偏移量就可以访问从 $1000\ 0000_{16}$ 到 $1000\ ffff_{16}$ 之间的内存空间。关于这点可参见 MIPS 参考数据卡的第 4 列

不同语言对于内存空间的处理

C显式调用函数在堆上分配和释放空间

- malloc()：堆上分配空间并返回
- free()：释放指针指向堆空间

Java使用内存分配和无用单元回收机制来防止错误发生