

CH2 指令：计算机的语言

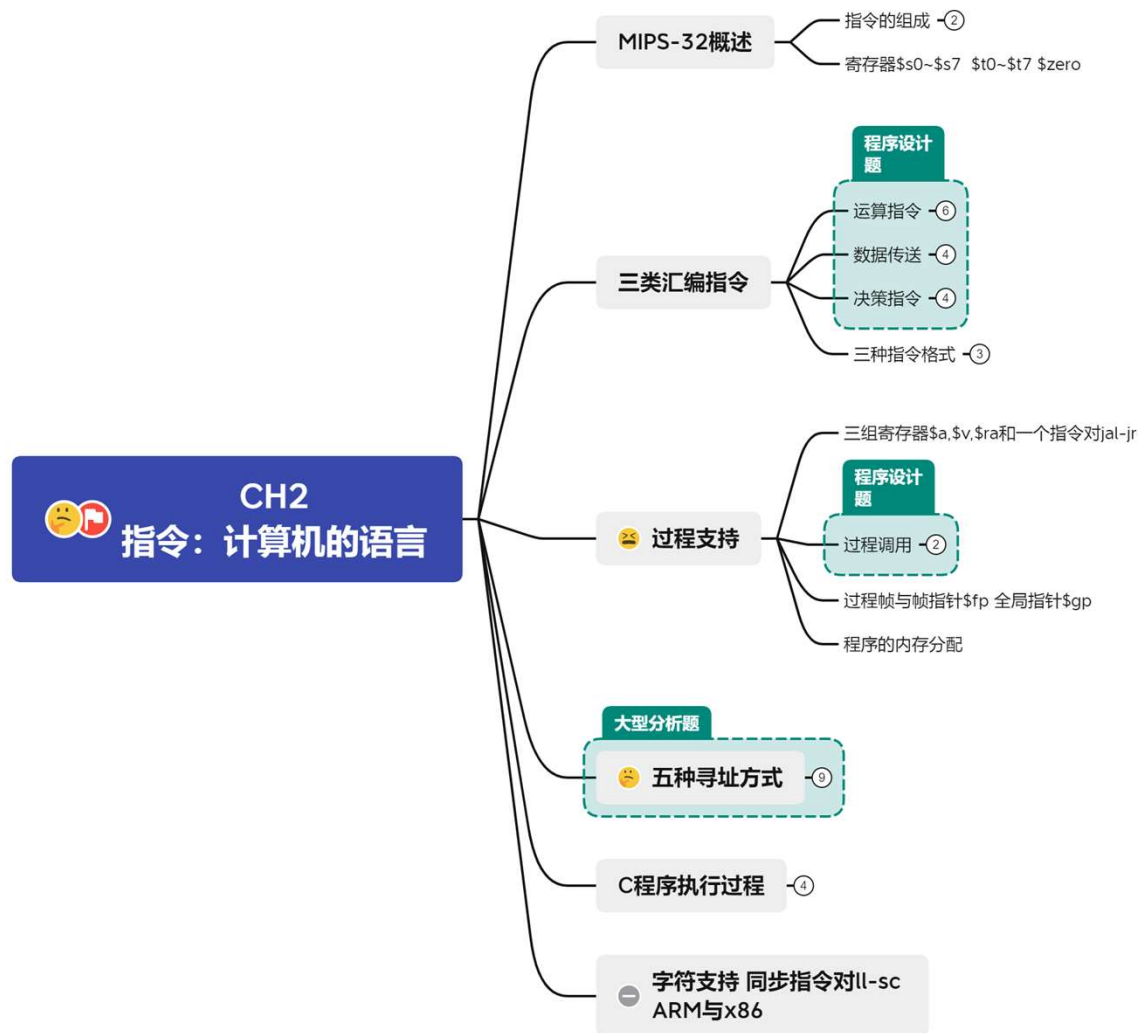
课程基于

《计算机组成与设计：硬件/软件接口》5e

Patterson & Hennesy 著

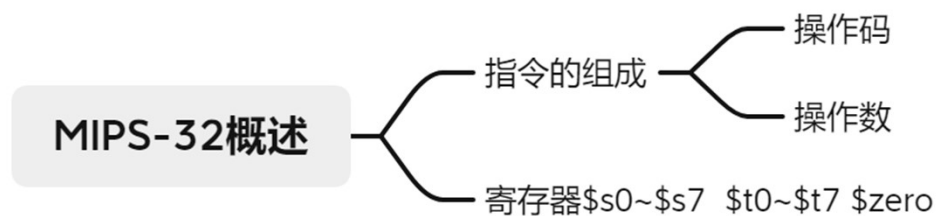
B站 翼云图灵

章节导图



第一部分

MIPS-32概述



B站 翼云图灵

指令的组成 MIPS的设计思想

计算机执行任何程序，本质上都是在执行机器语言指令 (instruction)

每条指令都是一条0-1串

指令首先要指明执行什么操作，通常用0-1串中的前几位来表示，称为操作码

指令还要指出需要操作的数据来自哪里、操作后的结果数据放回哪里

通常用0-1串中的剩余位来表示，称为操作数或地址码

大部分操作数都是一个地址编号，告诉CPU从哪里取得数据、向哪里放回数据

所以操作数通常也叫做地址码



MIPS作为一种RISC指令集，设计力求保证硬件设备的简单性

在我们讲解的32位MIPS汇编语言 (MIPS-32) 中，所有指令都是32位长

B站 翼云图灵

MIPS-32中的通用寄存器

MIPS中运算操作的操作数必须来自寄存器 (register) 或者指令本身
一种位于CPU、比cache更小更快的存储器
用来暂时存放运算的源数据和结果

一些寄存器是专用的，如存放执行中指令的地址的程序计数器 (PC)
与此相对应，用于暂时存放运算数据的寄存器称为通用寄存器

MIPS中一共有32个32位的寄存器，共128B（大部分架构都采用16或32个寄存器）
我们约定：

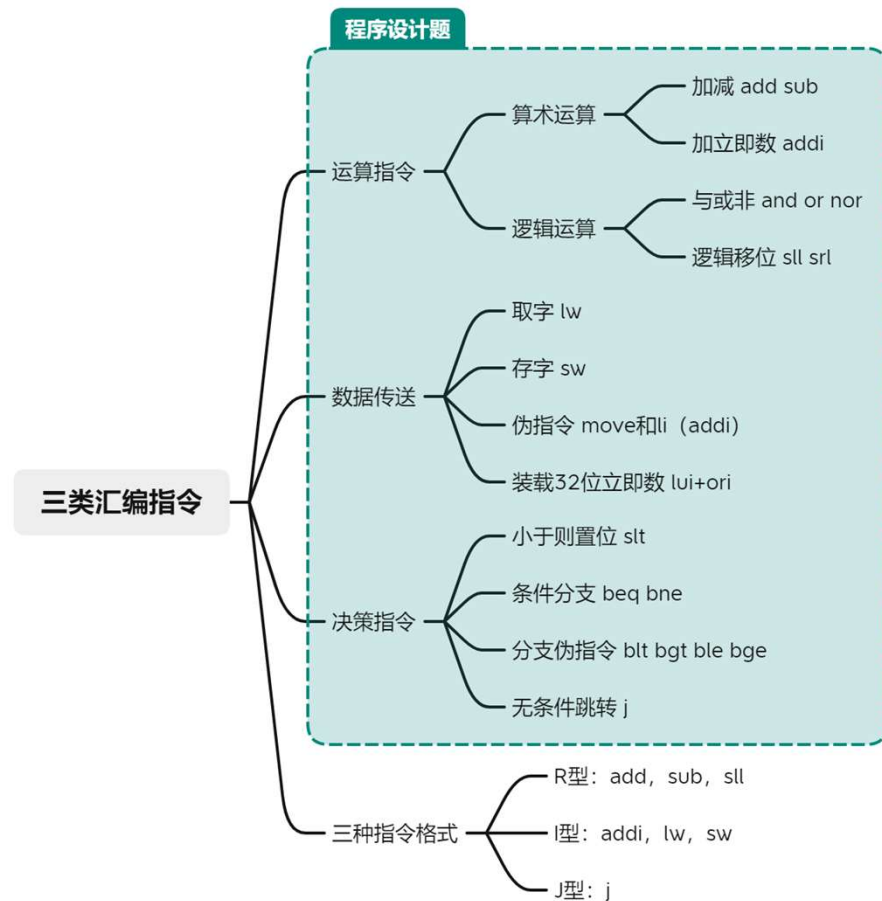
程序中的变量存放在保存寄存器 (store reg) 中：\$s0~\$s7共8个

运算的临时变量、中间变量存放在临时寄存器 (temp reg) 中：\$t0~\$t7共8个

还有一个零寄存器，永远存放32位的0，写作\$zero

第二部分

三类汇编指令



算术运算：加add、减sub

C赋值语句：c = a + b;

加法指令 **add c, a, b**：将a和b中的数据相加，并将结果存放在c中

再次强调：MIPS中运算的操作数必须来自寄存器或者指令本身！

假设变量a, b, c分别存放在寄存器\$s0, \$s1, \$s2中，这条指令就应当写为

add \$s2, \$s0, \$s1

加法中两个加数可以对换，但减法不行，故c = a - b; 必须写作

sub \$s2, \$s0, \$s1

运算的“原材料” a和b对应的寄存器\$s0, \$s1

分别称为源操作数1 (src1) 和源操作数2 (src2)

运算的结果c对应的寄存器\$s2称为目的操作数 (des)

加减指令的通式： **add/sub des, src1, src2**

算术运算：加立即数addi

在 $i++$ 即 $i = i + 1$; 这条赋值语句中，有个确定的常数1

与其采取额外的步骤将1装入某个寄存器，不如让指令本身包含这个1

假设变量i位于寄存器\$*s0*，我们把加法指令的第二个源操作数改为常数1

```
addi $s0, $s0, 1
```

就成了加立即数 (add immediate) 指令

因为addi指令中的立即数可以取负数（对立即数取负后相加）

因此，MIPS中没有subi指令

逻辑按位运算：and、or、nor 指令

当两个源寄存器中，对应的位上同时为1时，与and操作结果为1

当两个源寄存器中，对应的位上至少有一个为1时，或or操作结果为1

因此，假设

\$t0 = 0000 0000 0000 0000 0000 0000 0000 1001

\$t1 = 0000 0000 0000 0000 0000 0000 0000 1100

执行下列两条指令后，\$t2中的数据分别变为多少？

and \$t2, \$t0, \$t1

or \$t2, \$t0, \$t1

任何数据与0进行或非nor操作，都会0/1反转

执行下列指令后，\$t2中的数据会变为多少？

nor \$t2, \$t0, \$zero

逻辑移位运算：sll和srl指令

比较12和120两个十进制数，通过在最低位的右边添加一个0，变为了10倍

比较11和110两个二进制数，通过在最低位的右边添加一个0，变为了多少倍？1100呢？

逻辑左移 (shift left logic) 指令让寄存器中的数据整体往左移动指定的位数
并在右边空出来的位上补0

假设 \$s2 = 29个0+101

逻辑左移两位后，放到寄存器\$s0中：

sll \$s0, \$s2, 2

这里的2不是addi指令中的立即数，而是告诉计算机移动几位的**移位量 (shift amount)**

通过这样一句指令，我们实际上完成了x4的运算！

x2、x8、x128时，移位量分别是多少？

srl指令当然可以实现/2运算，使用场景不多，不额外讨论

综合练习1：变量运算与赋值

翻译以下C语句：

```
result = a - 10 + (b + c * 5);
```

寄存器-存储器数据传送：lw指令

运算指令的操作数必须来自于寄存器/指令本身

但是，通用寄存器一共只有128B

数组元素却可以占据成千上万个字节，只能存放在内存中

这时，我们把数组第一个元素（a[0]）的32位地址，称为数组的**基址**，放在寄存器中
基址加上要找的元素的**下标**，就组成了这个元素的地址

如果源操作数在内存中，是数组a的5号元素（第六个元素），数组a的基址存放在\$s1中
那么，a[5]的地址就表示为5(\$s1)

计算机会自动计算\$s1中的基址和**偏移量**5的和，找到a[5]的地址

将a[5]从内存传送到寄存器\$s0，使用**取字指令**（load word）：

lw \$s0, 5(\$s1)

寄存器-存储器数据传送：字与sw指令

MIPS的通用寄存器都是32位长

这个长度就是MIPS体系结构的字长，通常代表了参与运算的数据的长度

因此我们约定：整门课程中，1字=32b=4B

a[5]相对于a[0]，在内存中的距离是5个字，而不是5个字节

又因为内存按字节编址，即，内存每个字节都有一个特定的编号

所以偏移量应该是 $5 \times 4 = 20$ 个字节

a[5]的地址应表示成20(\$s1)

于是取数指令变为

lw \$s0, 20(\$s1)

如果我们要把\$t0中的运算结果送回内存中的a[2]，需要用到存字指令 (store word)：

sw \$t0, 8(\$s1)

B站 翼云图灵

寄存器间数据传送 装载立即数到寄存器

如果我们需要把数从\$t0保存到存放某变量的\$s1中，怎么实现？

MIPS没有专门的寄存器间移动数据的指令

但是，通过把源寄存器中的数据加上0再保存到目标寄存器中，可以实现相同的功能

`addi $s1, $t0, 0` 或 `add $s1, $t0, $zero`

这个功能可以用`move`伪指令来代替

`move $s1, $t0`

假如我们要把一个常数10装入寄存器\$s2，同样可以采用`addi`指令

`addi $s2, $zero, 10`

或使用`取立即数 (load immediate)` 伪指令

`li $s2, 10`

程序设计题中能否使用伪指令，请咨询老师！

装载32位立即数到寄存器

我们说可以用addi指令向寄存器装载立即数: `addi $s2, $zero, 10`

但是, `addi`指令中的立即数10只能占用32位指令中的一部分(16位, 稍后介绍指令格式)
16位只能表示 2^{16} 即六万多个数, 寄存器却能容纳 2^{32} 即40多亿个数

二进制与十六进制的转化在此不作介绍

假设我们要向寄存器\$s2装载一个32位的立即数: `10A2 7FFF` ₍₁₆₎

我们必须先用取高位立即数 (`load upper immediate`) 指令, 把10A2放入\$s2的高16位

`lui $s2, 4258` #十六进制的10A2等于十进制的4258

再让\$s2与低16位的立即数7FFF进行或运算

`ori $s2, $s2, 32767` # $7FFF(16) = 32767(10)$

这样, 就分两步把32位立即数装载到了32的寄存器中

不能使用addi替代ori指令, 如果低16位的最高位是1, addi会把它理解为负数

综合练习2：数组元素运算与赋值

$a[i] = a[0] + 100000;$

假设数组a的基址位于\$*s0*，变量*i*位于\$*s1*

$100000(10)=186A0(16)$ ， $1(16)=1(10)$ ， $86A0(16)=34464$

决策：条件分支beq和bne

计算机和一般计算器的区别在于何处？

在于决策能力！

即，根据一定的条件选择执行何种运算的能力

最基础的判断条件是相等关系

假设 $\$s0 = 0$, $\$s1 = 0$, $\$s2 = 1$

相等则分支 (branch if equal) 指令在两个源操作数寄存器中的值相同时分支

分支以分支标签表示

beq $\$s0$, $\$s1$, Label

与此相对应，不等则分支 (branch if not equal) 指令在值不同时分支到标签

bne $\$s0$, $\$s2$, Label

如果不发生分支，则继续执行内存中相邻的下一条指令

综合练习3: if-else语句（无条件跳转j和条件分支）

if (i == j) f = g + h;

else f = g - h;

假设f、g、h、i、j分别存放在\$s0~\$s4中

结论：判定相等 == 使用bne，判断不等 != 使用beq

B站 翼云图灵

决策：小于则置位slt

除了相等、不等关系，我们还经常比较两个数的大小

MIPS有一条小于则置位 (set on less than) 指令slt

置位：将一位设置为1；复位：将一位设置为0

还是假设 $\$s0 = 0$, $\$s1 = 0$, $\$s2 = 1$

slt \$t0, \$s0, \$s2

源操作数1 < 源操作数2 吗？Yes!

此时把目的操作数寄存器\$t0置位为1

slt \$t0, \$s0, \$s1

源操作数1 < 源操作数2 吗？No!

此时把目的操作数寄存器\$t0复位为0

6种条件判定及其伪指令

通过slt、beq、bne（严格来说还有小于立即数则置位slti指令，不作讨论）指令的各种组合，我们就能够实现全部六种比较条件，即六种值为真或假的布尔表达式

if (i < j) f = g + h;	slt \$t0, i, j	#当i<j时，把\$t0置为1，否则为0
else f = g - h;	beq \$t0, \$zero, Else	#当\$t0为0时，执行else后的语句
	add f, g, h	#否则顺着执行if后的语句
	j Exit	#加法完成后退出if-else语句
Else:	sub f, g, h	#else
Exit:		

结论：判定大于 > 或小于 < 使用slt和beq，判定大于等于 ≥ 或小于等于 ≤ 使用slt和bne

对于比大小的四种比较条件，可以使用伪指令：

小于则分支blt	大于则分支bgt
小于等于则分支ble	大于等于则分支bge

综合练习4：while循环

```
while (a[i] == k) i++;
```

假设i, k分别存放在\$s3和\$s5中, a的基址存放在\$s6中

MIPS汇编指令小结

类别		指令名称	指令格式
一、运算指令	算数运算	加减法指令	add/sub des, src1, src2
		加立即数指令	addi des, src1, i
	逻辑运算	与/或/或非指令	and/or/nor des, src1, src2
		或立即数指令	ori des, src1, i
		逻辑左移/右移指令	sll/srl des, src1, shamt
二、数据传送指令		取字/存字指令	lw/sw reg, num(reg)
		取高位立即数指令	lui reg, i
三、决策指令	条件分支	相等/不等则分支指令	beq/bne src1, src2, Label
		小于则置位指令	slt des, src1, src2
	无条件跳转	跳转指令	j Label
伪指令		条件分支伪指令	blt/bgt/ble/bge src1, src2, Label
		寄存器数据传送伪指令	move des, src

指令格式：R型

指令中含三个寄存器的运算指令都属于R型 (register type) 指令

add/sub des, src1, src2

and/or/nor des, src1, src2

slt des, src1, src2

32位的MIPS指令一共分为6个字段：



op: operation code, 操作码

rs: register source, 源操作数寄存器 → **rt**: s后面是t, 表示第二个源操作数寄存器

rd: register destination, 目的寄存器

shamt: shift amount, 移位量

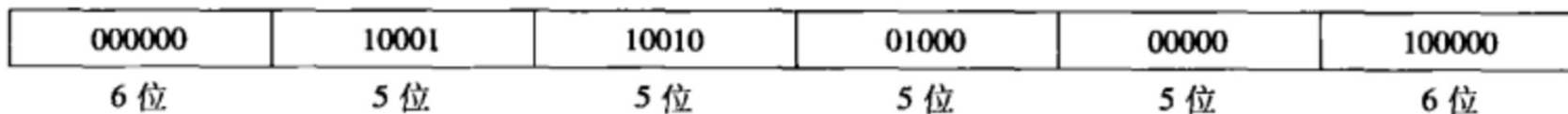
funct: function code, 功能码

指令格式：R型

R型指令的操作码op都是6个0，由6位功能码funct进一步指定执行什么操作
以add指令为例

\$t0~\$t7分别为8~15号寄存器

\$s0~\$s7分别为16~23号寄存器



sub指令仅仅是功能码funct字段从32变为了34，sub \$s1, \$s1, \$s0的32位机器码是多少？
需要记忆add、sub指令的操作码（都是0）和功能码（分别为32、34）

此外，使用移位量的两条逻辑移位指令

sll/srl des, src1, shamt

也属于R型指令，因为没有第二个源操作数寄存器，rt被置为0

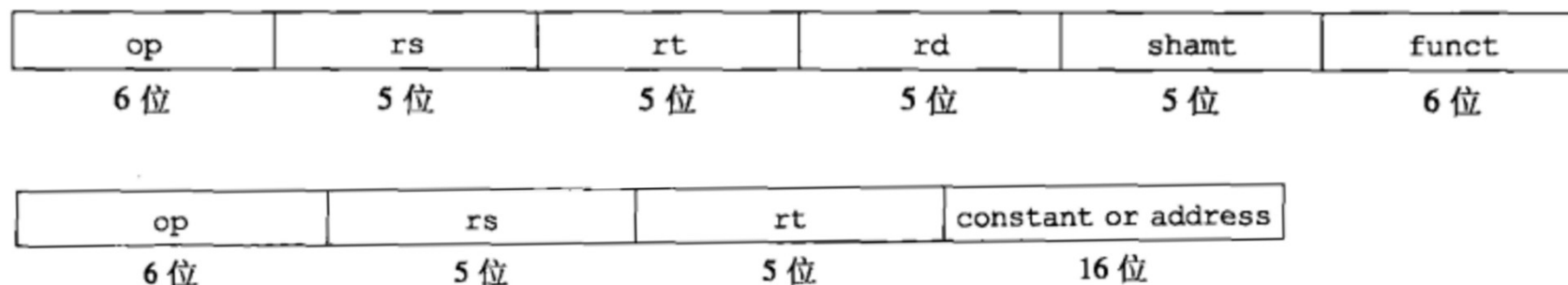
指令格式：I型（立即数）

有两条“目的reg+源reg+立即数”格式的指令

addi des, src1, i

ori des, src1, i

通过把R型指令中的后三个字段拼接成一个16位的立即数字段，让指令本身包含常数
这样的指令属于I型（immediate type）指令



以addi指令为例

其操作码为8，由于rd字段被合并了，现在rt就成了目的寄存器

指令格式：I型（偏移量）

lw/sw reg, num(reg)

两条数据传送指令也包含两个寄存器和一个常数

同样属于I型指令

此时，16位立即数字段的含义发生了改变，表示数组元素相对于数组基址的**地址偏移量**



无论是lw还是sw指令

都是由rs字段表示的寄存器值与address字段相加，得到存储器单元地址

rt字段表示与存储器单元交换数据的寄存器

lw、sw指令操作码分别为35和43

lw \$t0, 8(\$s1)

指令格式：I型（标签）

beq/bne src1, src2, Label

在这两条条件分支指令中，同样使用了两个寄存器

还有一个分支标签的地址，用16位立即数字段表示（也就变成了Address字段）
也属于I型指令

例如，当i (\$s0) 和j (\$s1) 相等时分支到地址为10000的标签Else

beq \$s0, \$s1, Else

翻译为机器语言为



这里的10000实际上并不是Else标签指向指令的地址，讲寻址方式时再具体说明

机器语言指令格式小结

MIPS 机器语言								
名字	格式	举例						注释
add	R	0	18	19	17	0	32	add \$s1, \$s2, \$s3
sub	R	0	18	19	17	0	34	sub \$s1, \$s2, \$s3
addi	I	8	18	17	100			addi \$s1, \$s2, 100
lw	I	35	18	17	100			lw \$s1, 100 (\$s2)
sw	I	43	18	17	100			sw \$s1, 100 (\$s2)
字段宽度		6 位	5 位	5 位	5 位	5 位	6 位	所有 MIPS 指令均为 32 位
R 型	R	op	rs	rt	rd	shamt	funct	算术指令格式
I 型	I	op	rs	rt	address			数据传送指令格式

lui指令的指令格式不作讨论

五条伪指令本身不是真正的指令，程序运行时会被替换为真正指令，不讨论指令格式

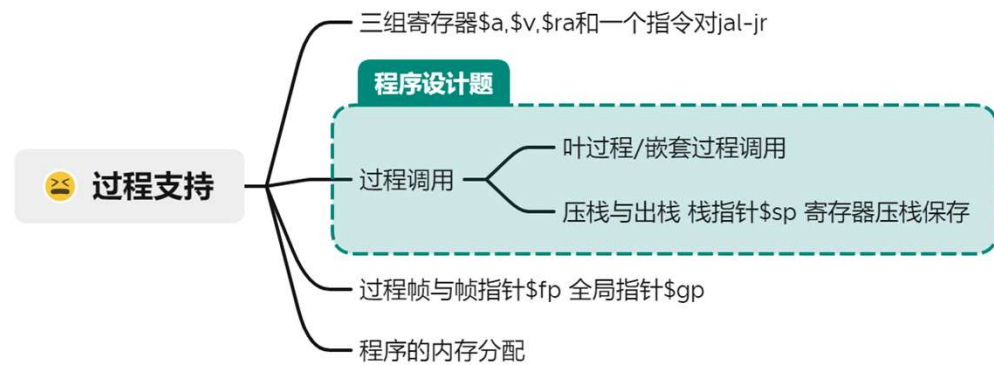
j指令的指令格式稍后讲解

复习题

- 1、指令通常由哪两个部分组成？MIPS-32指令长度均为多少？
- 2、8个临时寄存器、8个保存寄存器分别是什么编号？零寄存器存储什么？
- 3、回顾综合练习1~4，掌握运算、数据传送、决策三类汇编指令，注意字和字节的区别
- 4、练习上一页PPT中五条指令（add、sub、addi、lw、sw）汇编语言和机器语言的转化

第三部分

过程支持



B站 翼云图灵

过程（函数）的执行过程

C语言中的函数（一种典型的过程）是结构化编程的强大工具
函数获取参数、执行运算、返回结果，就好比
侦探拿着一份计划书去执行任务，再带来想要的结果

- 1、主程序（调用者）将参数放在过程（被调用者）可以取用的特定位置 什么位置？
- 2、主程序将控制权交给过程
- 3、过程申请并获得存储资源
- 4、过程执行
- 5、过程将结果的值放在主程序可以取用的特定位置 什么位置？
- 6、过程把控制权返还给主程序，执行调用过程指令的下一条指令 怎么找到这个位置？

支持过程的三大寄存器

1、主程序（调用者）将参数放在过程（被调用者）可以取用的特定位置 什么位置？

4个参数寄存器（argument reg） \$a0~\$a3

5、过程将结果的值放在主程序可以取用的特定位置 什么位置？

2个值寄存器（value reg） \$v0~\$v1

6、过程把控制权返还给主程序，执行调用过程指令的下一条指令 怎么找到这个位置？

主程序把下一条指令的32位地址存入

1个返回地址寄存器（return address reg） \$ra

截至目前，学习了保存寄存器、临时寄存器各8个，零寄存器1个，以及这一页的7个共24个寄存器，占MIPS-32寄存器总数的四分之三

主程序通过什么指令，可以跳转到过程指令，并把下一条指令的地址存入\$ra？

j+addi吗？

B站 翼云图灵

jal-jr指令对 程序计数器

跳转并链接 (jump and link) 指令可以同时实现两个功能:

- ①无条件跳转到一个标签
- ②将下一条指令的地址放入返回地址寄存器\$ra

jal Label

jal指令由调用者主程序使用，还是由被调用者过程使用？

寄存器跳转 (jump reg) 指令可以跳转到某一寄存器存储的32位地址

基本上只和返回地址寄存器搭配

jr \$ra

jr指令由调用者主程序使用，还是由被调用者过程使用？

SCUer遇到MIPS翻译C函数的题，这么写，好歹有一分：

函数名: jr \$ra

综合练习5：数组清零函数（叶过程）

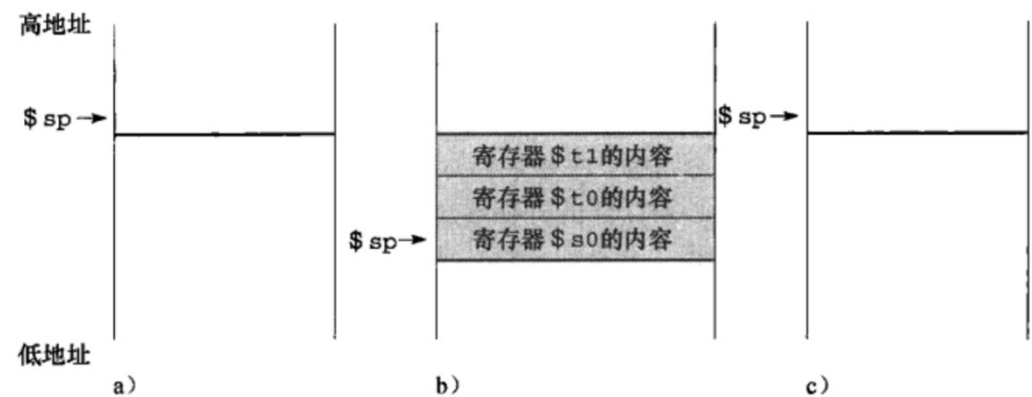
```
void clear(int a[ ], int size){  
    for (i = 0; i < size; i++) a[i] = 0;  
}
```

保存寄存器的压栈和出栈 栈指针\$sp

在过程调用前，主程序往往已经将自己要用的变量放在了保存寄存器中
如果过程要使用保存寄存器，要把主程序已经使用的保存寄存器入栈

栈在内存中以高地址为栈底，低地址为栈顶
即，栈从高地址向低地址“生长”

栈指针 (stack pointer) 永远指向栈顶



入栈时，先把 $\$sp$ 减去待保存的

保存寄存器个数的4倍 为什么是4倍？

再用sw将保存寄存器存入栈中（方向从栈底到栈顶）

过程结束时把这些数据出栈、放回保存寄存器，供主程序继续使用
步骤正好相反

综合练习6：运算函数（叶过程）

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

假设f存储在\$s0中

综合练习6改进：减少指令条数

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

嵌套过程调用 综合练习7：数组求平方和（嵌套过程）

侦探搞外包、接着雇其他侦探来完成任务，就是嵌套过程调用

```
int sum_of_squares(int a[ ], int size){
    int i = 0;
    int sum = 0;
    for(i = 0, i < size, i++)
        sum = sum + square(a[i]);
    return sum;
}

int square(int a){
    int square;
    square = a * a;
    return square;
}
```

需要压栈保存的寄存器

【综合练习6】我们默认保存寄存器\$*s0*~\$*s7*存放了主程序的变量
需要由过程开始时压栈保存，结束时出栈恢复

【综合练习7】如果一个过程（外层函数）嵌套了其他过程（内层函数）
外层函数通过jal修改了返回地址寄存器\$*ra*
\$*ra*指向外层函数jal的下一条指令，不再是外层函数的返回地址

栈指针寄存器\$*sp*、栈中的内容（即栈指针以上的栈）也需要由过程保留
在addi栈指针、sw入栈、lw出栈的过程中即可保存

结论：任何过程须显式地压栈保存即将使用的保存寄存器\$*s0*~\$*s7*（用哪几个存哪几个）
外层嵌套过程须显式地压栈保存返回地址寄存器\$*ra*

进阶内容：复杂MIPS程序示例

2.8节（68页）提供了一个递归嵌套调用过程计算阶乘的MIPS程序

2.13节（90页）提供了一个冒泡排序过程嵌套交换过程的MIPS程序

习题2.27（114页要素察觉）考察双层for循环的翻译

习题2.34（115页）考察自嵌套调用的多参数过程的翻译

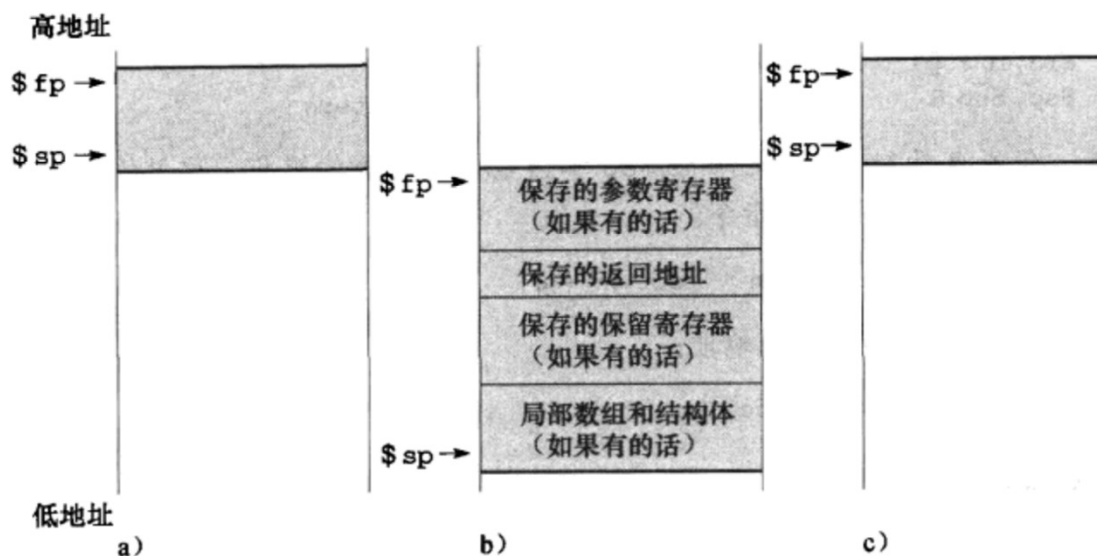
```
2.27      addi $t0, $0, 0
           beq  $0,  $0, TEST1
LOOP1:     addi $t1, $0, 0
           beq  $0,  $0, TEST2
LOOP2:     add  $t3, $t0, $t1
           sll  $t2, $t1, 4
           add  $t2, $t2, $s2
           sw   $t3, ($t2)
           addi $t1, $t1, 1
TEST2:     slt  $t2, $t1, $s1
           bne  $t2, $0, LOOP2
           addi $t0, $t0, 1
TEST1:     slt  $t2, $t0, $s0
           bne  $t2, $0, LOOP1
```

```
2.34 f:  addi $sp, $sp, -12
           sw   $ra, 8($sp)
           sw   $s1, 4($sp)
           sw   $s0, 0($sp)
           move $s1, $a2
           move $s0, $a3
           jal  func
           move $a0, $v0
           add  $a1, $s0, $s1
           jal  func
           lw   $ra, 8($sp)
           lw   $s1, 4($sp)
           lw   $s0, 0($sp)
           addi $sp, $sp, 12
           jr   $ra
```


过程帧与帧指针\$fp

为了标记运行中过程建立的栈，除了栈顶的栈指针\$sp
还可以加一个帧指针 (frame pointer) \$fp指向栈底
即过程帧的第一个字

\$fp和\$sp之间的空间由正在运行的过程使用
称为过程帧，也叫活动记录



全局指针\$gp 程序的内存分配

为了便于寻找位置固定的数据

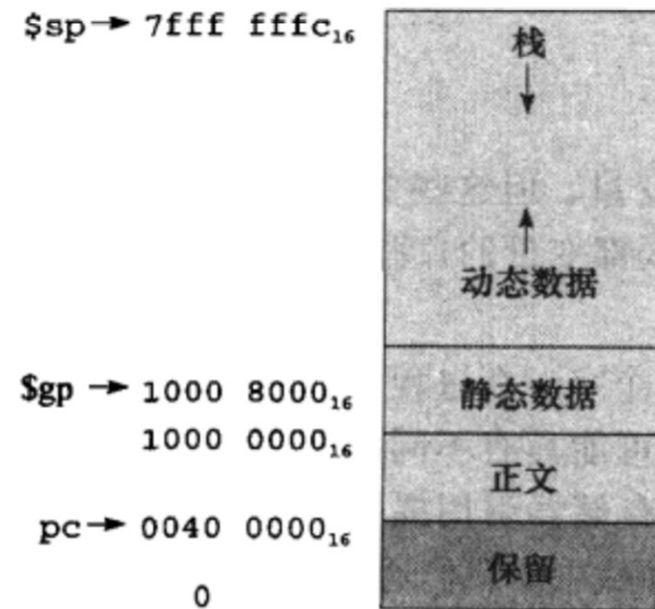
(主程序使用的变量, 以及声明为static的变量, 统称静态变量)

使用一个固定指向静态数据区某一位置的全局指针 (global pointer) \$gp

程序在内存中包含五段, 地址从低到高分别为

- 1) 保留段
- 2) 正文段 (代码段), 保存指令
- 3) 静态数据段, 保存静态数据
- 4) 动态数据段 (堆), 从低往高 “生长”
- 5) 栈, 从高往低 “生长”

栈和堆此消彼长, 实现了内存空间的高效利用



第四部分

五种寻址方式

大型分析题



五种寻址方式

32个通用寄存器小结

R型 — 1、寄存器寻址

2、立即数寻址

I型 — 3、基址偏移寻址

4、PC相对寻址

J型 — 5、伪直接寻址

B站 翼云图灵

32个通用寄存器及其编号

寄存器	名称	编号
\$zero	零寄存器	0
\$v0~\$v1	返回值寄存器	2~3
\$a0~\$a3	参数寄存器	4~7
\$t0~\$t7	临时寄存器	8~15
\$s0~\$s7	保存寄存器	16~23
\$t8~\$t9	额外的临时寄存器	24~25
\$gp	全局指针	28
\$sp	栈指针	29
\$fp	帧指针	30
\$ra	返回地址寄存器	31

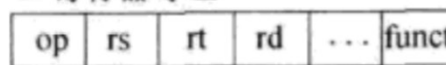
R型：寄存器寻址

所有操作数都是寄存器的指令采用寄存器寻址 (register addressing)

操作数个数从一个到三个不等

R型指令 ⇔ 寄存器寻址

2. 寄存器寻址



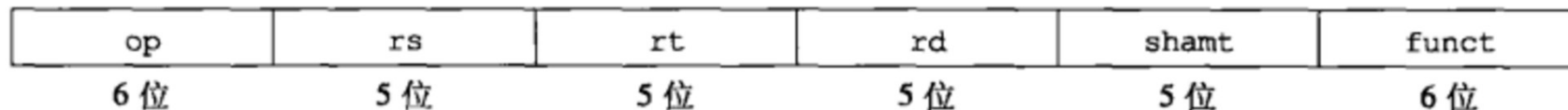
寄存器
寄存器

第二部分已经提到过的R型指令有：

- 1、运算指令：add, sub, and, or, nor 5条三寄存器操作数指令
- 2、运算指令：sll、srl 2条双寄存器操作数指令（rs不使用置为0，使用shamt）
- 3、决策指令：slt 1条三寄存器操作数指令

第三部分新增R型指令：

- 4、决策指令：jr 1条单寄存器操作数指令

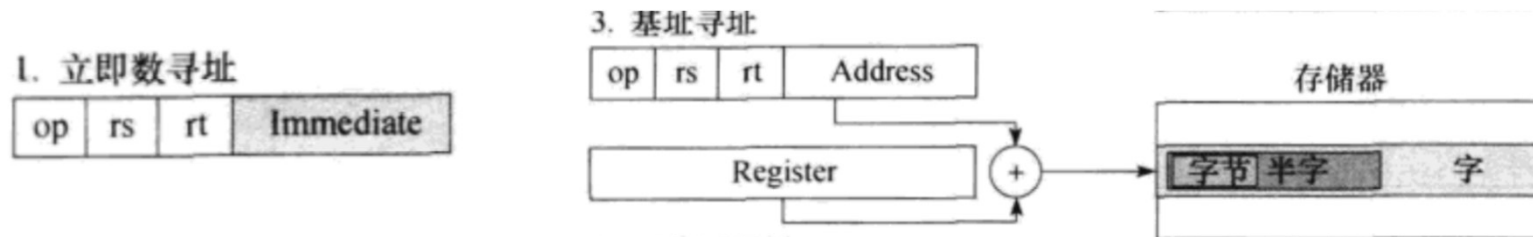


I型：立即数寻址和基址偏移寻址

第三个操作数（第二个源操作数）是常数的指令采用立即数寻址（immediate addressing）

具体包括addi, ori两条指令

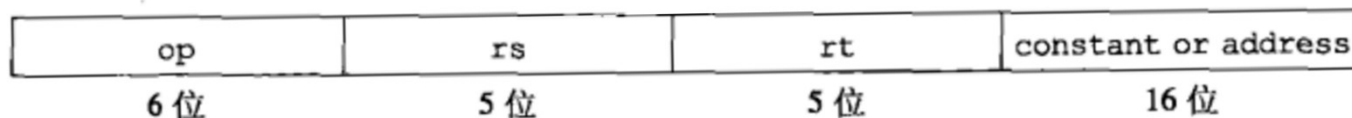
其实还包括lui指令，课本没有着重强调其指令格式



两条数据传送指令lw, sw

将基址寄存器和偏移量相加的内存寻址方式称为基址偏移寻址

可单独称为基址寻址（base addressing）和偏移寻址（displacement addressing）



I型：PC相对寻址 字地址和字节地址

两条条件分支指令beq, bne

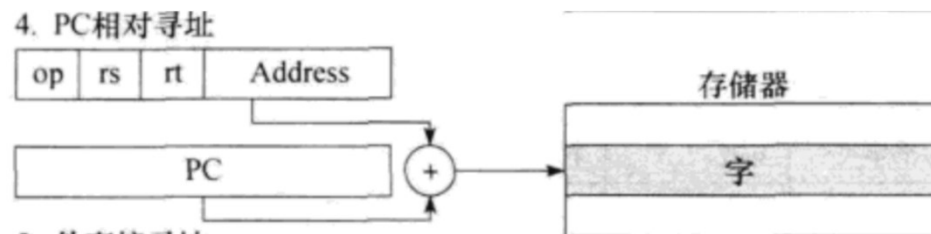
在汇编语言中使用标签来表示分支的目标地址，标签翻译成机器语言其实是个整数
告诉计算机从当前指令的地址出发，到达分支目标地址的距离是多少

程序计数器（program counter, PC）中保存了执行中指令的地址

分支指令中的16位分支地址是一个二进制补码，可正可负

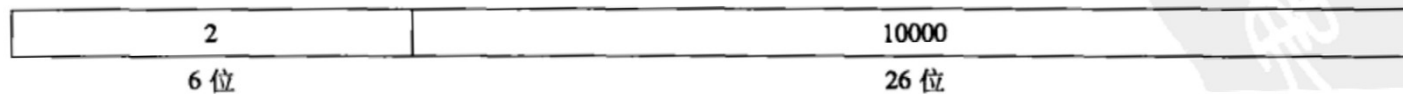
表示以PC+4为基准相加的字地址数目，叫做PC相对寻址（PC-relative addressing）

分支32位地址 = PC + 4 + 字地址偏移量



J型：伪直接寻址

J型指令只需要操作码和目标地址两个字段，形式上最为简单



J型指令 ⇔ 伪直接寻址，包含j, jal两条

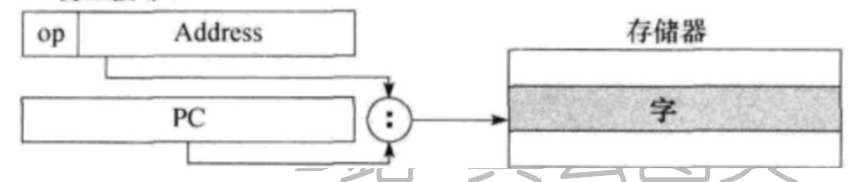
*寄存器跳转jr指令是R型指令

直接寻址指的是指令中直接给出32位内存地址，但J型指令地址字段只有26位
因此，执行J型指令时，先将26位字地址左移两位（右侧补0）形成28位字节地址
再和PC的高四位拼接成32位地址

这就是伪直接寻址（pseudodirect addressing）

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

5. 伪直接寻址



扩大分支与跳转的范围

PC相对寻址以PC+4为基准，**加上**一个可正可负的16位补码字地址，寻址范围为

$(PC + 4) - 2^{17} \sim (PC + 4) + 2^{17} - 4$ 大约是分支前后各128KB

伪直接寻址用PC中当前指令地址的高四位**拼接**指令中的26位字地址，寻址范围为

和PC高四位相同的一切地址 一个256MB的地址块

在相近的内存地址中寻址利用了加速大概率事件这一设计思想

要分支到更远距离，可以将beq/bne取反，下接一条可能绕过的j指令

要跳转到更远距离，可以先将32位地址装载到某临时寄存器，再用jr指令

例题 远距离的分支转移

假设在寄存器 \$s0 与寄存器 \$s1 值相等时需要跳转，可以使用如下指令：

```
beq    $s0, $s1, L1
```

用两条指令替换上面的指令，以获得更远的转移距离。

寻址方式小结

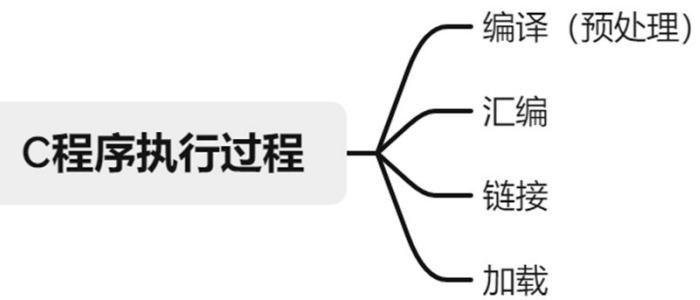
- ①R型的寄存器寻址：操作数为1个/2个/3个寄存器的数据
- ②I型的立即数寻址：addi、lui、ori三条立即数指令，其中一个操作数是指令字段中的常数
- ③I型的基址偏移寻址：lw、sw两条访存指令
将rs中的基址和偏移量直接相加，得到偏移地址 以lw \$t0, 12(\$s0)为例
- ④I型的PC相对寻址：beq、bne两条条件分支指令
分支指令中的PC相对地址（可正可负的字偏移量）
左移两位（x4）形成字节偏移量
再和PC+4中的字节地址相加，形成分支目标地址
以地址为1000（十进制）的beq reg1, reg2, 4为例
- ⑤J型的伪直接寻址：将26位字地址左移两位（x4）形成28位字节地址
再和PC（实际上也是PC+4）的高四位拼接成32位跳转目标地址
以PC高四位为1010的j 0000 0000 0000 0000 0000 0000 01为例

MIPS汇编指令小结

类别		指令名称	汇编指令	指令格式与寻址方式
一、运算指令	算数运算	加减法指令	add/sub des, src1, src2	R
		加立即数指令	addi des, src1, i	I: 立即数寻址
	逻辑运算	与/或/或非指令	and/or/nor des, src1, src2	R
		或立即数指令	ori des, src1, i	I: 立即数寻址
		逻辑左移/右移指令	sll/srl des, src1, shamt	R
二、数据传送指令		取字/存字指令	lw/sw reg, num(reg)	I: 基址偏移寻址
		取高位立即数指令	lui reg, i	/
三、决策指令	条件分支	相等/不等则分支指令	beq/bne src1, src2, Label	I: PC相对寻址
		小于则置位指令	slt des, src1, src2	R
	无条件跳转	跳转指令	j Label	J
四、过程支持指令		跳转并链接	jal Label	J
		寄存器跳转	jr \$ra	R

第五部分

C程序执行过程



B站 翼云图灵

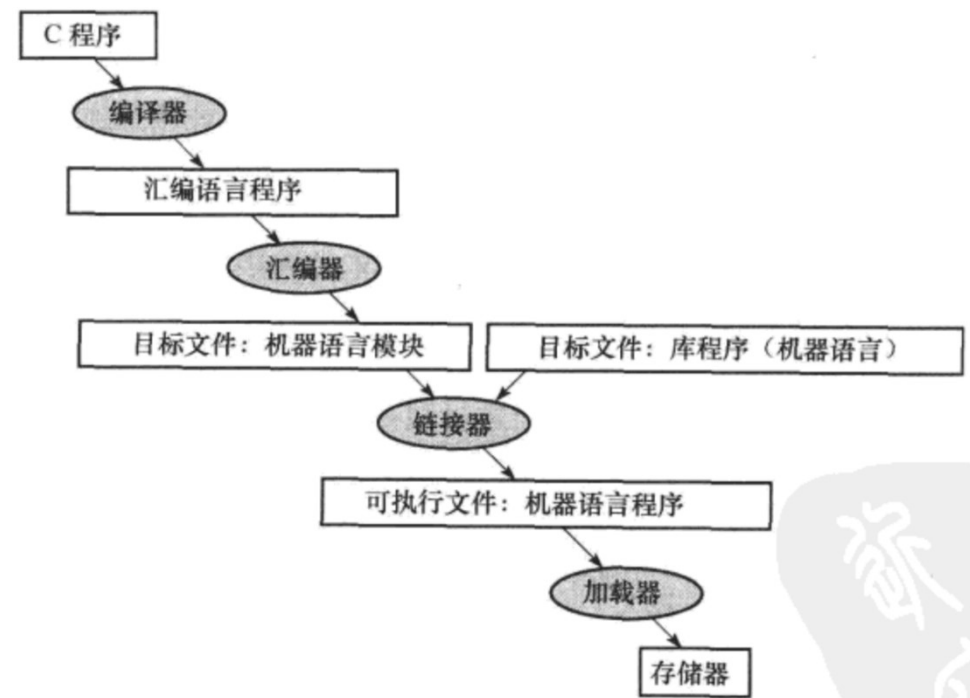
C语言的4个翻译层次

编译器将高级语言文件 (.c)
翻译成汇编语言文件 (.asm)

汇编器首先把伪指令替换为等价的真正指令
再将汇编语言翻译成机器语言目标文件 (.obj)

链接器把目标文件和
静态链接库 (.lib)、动态链接库 (.dll)
拼接成可执行文件 (.exe)

加载器将可执行文件放入内存，装载执行



字符支持 同步指令对ll-sc ARM和x86简介

为了让计算机能够处理C的8位的ASCII字符，MIPS提供**字节传送指令**lb, sb

字符通常理解为无符号数，故取字节常使用取无符号字节lbu指令

同理，为了支持Java的16位Unicode字符，MIPS提供**半字传送指令**lh, sh, lhu

当两个程序访问同一个内存单元，且其中存在写操作时，两程序操作的顺序就尤为重要

MIPS提供**链接取数ll指令**和**条件存数sc指令**，让程序员能够指定程序操作数据的顺序

ARM和MIPS同为RISC架构，具有优秀的能耗表现，广泛应用于移动端和嵌入式平台

同MIPS相比，ARM的主要区别是通用寄存器更少（16个）、寻址方式更多（9种）

Intel和AMD主导的**x86**是一种CISC架构，指令集十分庞大（2018年约1400条）

x86是一种本质非常糟糕的架构，最典型的表现是，指令长度从1B到15B不等

由于问世时间恰逢IBM进军PC领域，x86取得了巨大的商业成功，至今占据很大的份额

Intel将x86移动化的尝试屡屡碰壁，苹果将ARM电脑化的实践却高歌猛进

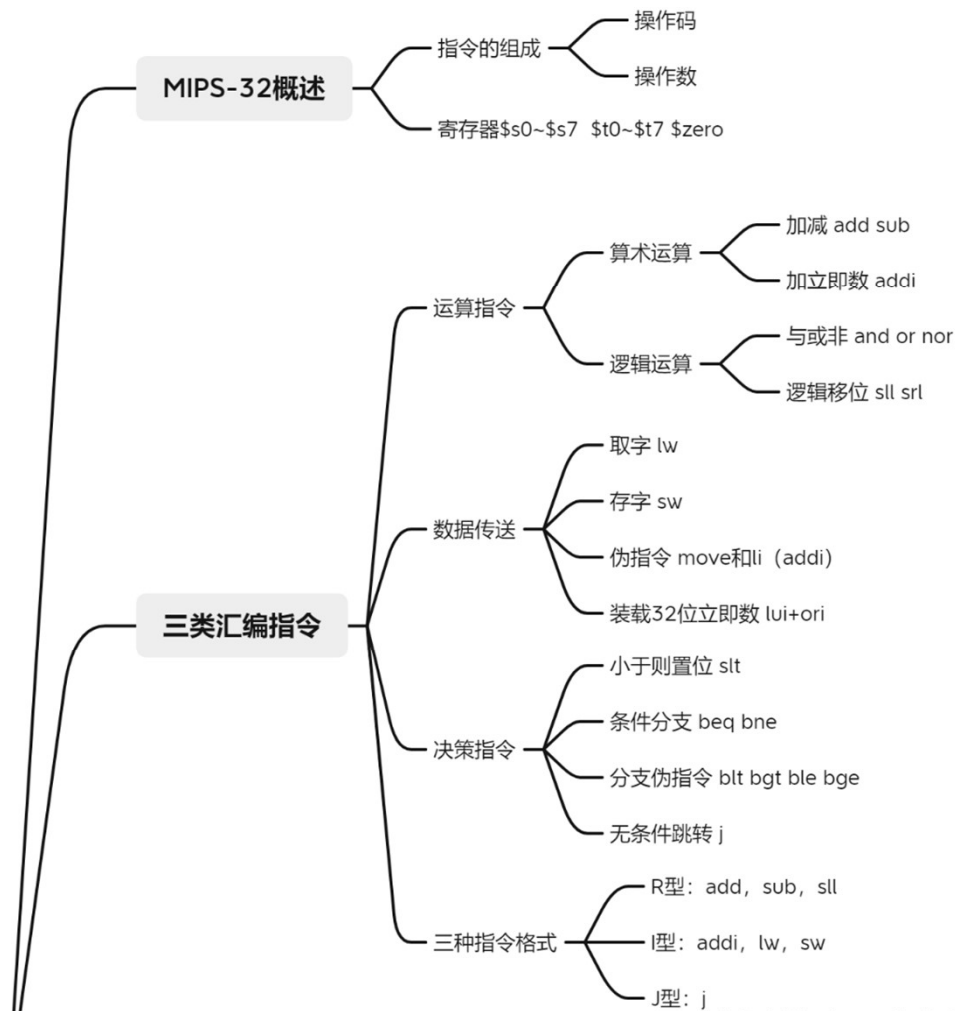
复习题

- 1、为了实现过程调用，我们引入了哪三类寄存器和哪个指令对？
- 2、三类寄存器分别存放什么？指令对中的两条指令分别由谁使用，完成什么功能？
- 3、当过程要使用保存寄存器时，要进行什么操作？
- 4、为什么过程内部的变量优先使用临时寄存器？
- 5、复习综合练习5~7，熟悉for循环、清零、函数调用等常见的C语句翻译
- 6、R型指令和J型指令分别采用什么寻址方式？I型有哪三种寻址方式？
- 7、PC相对寻址和伪直接寻址为了扩大寻址范围，其地址代表什么单位？
- 8、PC相对寻址和伪直接寻址分别怎样获得标签的真实地址？
- 9、复习MIPS汇编指令，对应上每条指令的指令格式和寻址方式
- 10、运行C程序要经过哪四个步骤？

全章复习

CH2 指令：计算机的语言

B站 翼云图灵





CH2 指令：计算机的语言

😞 过程支持

- 参数寄存器\$a0~\$a3 与值寄存器 \$v0~\$v1
- jal-jr指令对
- 过程调用
 - 叶过程/嵌套过程调用
 - 压栈与出栈 栈指针\$sp 需要保存的寄存器
 - 过程帧与帧指针\$fp 全局指针\$gp
- 程序的内存分配

C程序执行过程

- 编译
- 汇编
- 链接
- 加载

😞 五种寻址方式

- 32个通用寄存器小结
- R型 — 1、寄存器寻址
- I型
 - 2、立即数寻址
 - 3、基址偏移寻址
 - 4、PC相对寻址
- J型 — 5、伪直接寻址

— 字符支持 同步指令对ll-sc ARM与x86