处理器体系结构

第二章 指令集体系结构D

—执行程序

(Micro-processor Architecture)

内容概述

1. 执行程序

1.1 人机交互

1.2 翻译并执行程序

1.3 并行与同步

2. 汇编实战练习

2.1 几个设计实例

2.2 常见问题

内容概述

1. 执行程序

1.1 人机交互

1.2 翻译并执行程序

1.3 并行与同步

2. 汇编实战练习

2.1 几个设计实例

2.2 常见问题

1.1 人机交互—ASCII码

ASCII：American Standard Code for Information Interchange

ASCII表

（ American Standard Code for Information Interchange 美国标准信息交换代码 ）

高四位 / 低四位		ASCII控制字符 0000							ASCII控制字符 0001							ASCII打印字符 0010				ASCII打印字符 0011				ASCII打印字符 0100				ASCII打印字符 0101				ASCII打印字符 0110				ASCII打印字符 0111		
		0							1							2				3				4				5				6				7		
		十进制	字符	Ctrl	代码	转义字符	字符解释	十进制	字符	Ctrl	代码	转义字符	字符解释	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	Ctrl
0000	0	0		^@	NUL	\0	空字符	16	▶	^P	DLE		数据链路转义	32		48	0	64	@	80	P	96	`	112	p													
0001	1	1	☺	^A	SOH		标题开始	17	◀	^Q	DC1		设备控制 1	33	!	49	1	65	A	81	Q	97	a	113	q													
0010	2	2	☻	^B	STX		正文开始	18	↕	^R	DC2		设备控制 2	34	"	50	2	66	B	82	R	98	b	114	r													
0011	3	3	♥	^C	ETX		正文结束	19	‼	^S	DC3		设备控制 3	35	#	51	3	67	C	83	S	99	c	115	s													
0100	4	4	♦	^D	EOT		传输结束	20	¶	^T	DC4		设备控制 4	36	$	52	4	68	D	84	T	100	d	116	t													
0101	5	5	♣	^E	ENQ		查询	21	§	^U	NAK		否定应答	37	%	53	5	69	E	85	U	101	e	117	u													
0110	6	6	♠	^F	ACK		肯定应答	22	▬	^V	SYN		同步空闲	38	&	54	6	70	F	86	V	102	f	118	v													
0111	7	7	•	^G	BEL	\a	响铃	23	↨	^W	ETB		传输块结束	39	'	55	7	71	G	87	W	103	g	119	w													
1000	8	8	◘	^H	BS	\b	退格	24	↑	^X	CAN		取消	40	(	56	8	72	H	88	X	104	h	120	x													
1001	9	9	○	^I	HT	\t	横向制表	25	↓	^Y	EM		介质结束	41	)	57	9	73	I	89	Y	105	i	121	y													
1010	A	10	◙	^J	LF	\n	换行	26	→	^Z	SUB		替代	42	*	58	:	74	J	90	Z	106	j	122	z													
1011	B	11	♂	^K	VT	\v	纵向制表	27	←	^[	ESC	\e	溢出	43	+	59	;	75	K	91	[	107	k	123	{													
1100	C	12	♀	^L	FF	\f	换页	28	∟	^\	FS		文件分隔符	44	,	60	<	76	L	92	\	108	l	124	|												
1101	D	13	♪	^M	CR	\r	回车	29	↔	^]	GS		组分隔符	45	-	61	=	77	M	93	]	109	m	125	}													
1110	E	14	♫	^N	SO		移出	30	▲	^^	RS		记录分隔符	46	.	62	>	78	N	94	^	110	n	126	~													
1111	F	15	☼	^O	SI		移入	31	▼	^-	US		单元分隔符	47	/	63	?	79	O	95	_	111	o	127	⌂	^Backspace 代码：DEL												

注：表中的ASCII字符可以用“Alt + 小键盘上的数字键”方法输入。 2013/08/08

128个码：95个图形码、33个控制字符

Latin-1：

ASCII+96个图形码

如何表示中文、日文、韩文…？

Unicode：（万国码）

32bits

世界上大多数字符

1.1 人机交互—Unicode

思考：Unicode有没有什么问题？（提示：信息密度）

不同的解决方案：

***为每个字符分配固定长度内存?**

***为每个字符分配尽量少的内存?**

1. **UTF32: 使用4字节存储**
2. **UTF8: 使用1~4字节存储**
3. **UTF16: 使用2或4字节存储**

1.1人机交互—字节操作码

lb：load byte，读出一个字节 ，并将其放入寄存器的低8位

sb：store byte，读取寄存器低8位，写入内存

指令格式：

```
lb rt, offset(rs)        #符号位扩展0或1
lbu rt, offset(rs)       #符号位扩展0
sb rt, offset(rs)
```

由于高级语言几乎都是用byte来表示字符，而非数据，所以读取字节几乎都采用lbu!

->MIPS提供lb，但其不属于31条核心指令集

1.1人机交互—字节操作码

Char类数据为8位

例:

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

x和y的基址存于$a0和$a1

i存于$s0

子函数内如果使用$s0~s7，则必须对其进行保存/恢复

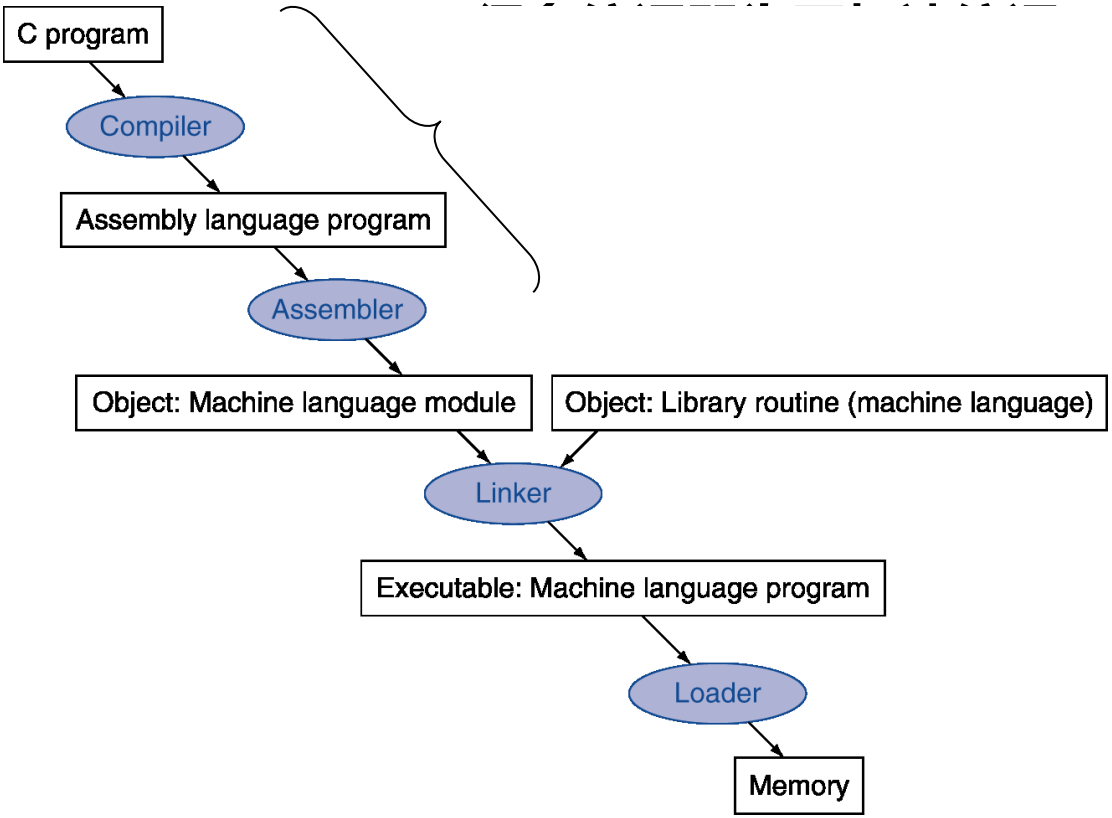
MIPS code:

```
strcpy: addi $sp, $sp, -4      #将$s0压栈
        sw   $s0, 0($sp)
        add  $s0, $zero, $zero # i = 0
L1:     add  $t1, $s0, $a1     # y[i]的地址
        lbu  $t2, 0($t1)       # $t2 = y[i]
        add  $t3, $s0, $a0     #x[i]的地址
        sb   $t2, 0($t3)       # x[i] = y[i]
        beq  $t2, $zero, L2    # 判定y[i] == 0
        addi $s0, $s0, 1       # i = i + 1
        j    L1                # 跳转至L1
L2:     lw   $s0, 0($sp)       # $s0出栈
        addi $sp, $sp, 4
        jr   $ra               # 跳转到返回地址
```

内容概述

1. 执行程序
    - 1.1 人机交互
    - 1.2 翻译并执行程序
    - 1.3 并行与同步
2. 汇编实战练习
    - 2.1 几个设计实例
    - 2.2 常见问题

1.2 翻译并执行程序



1975年前，很多操作系统和汇编器都是用汇编语言写的，因为那时候的内存容量很小，编译器的效率也不高。如今内存容量大大提高了，从而减少了对程序大小的限制；编译器经优化后产生的汇编语言程序，可以比得上汇编语言专家所写的程序，对于一些大程序，甚至比专家写的更好。这使得高级语言和编译器得到了广泛的应用

1.2 翻译并执行程序—汇编器与伪指令

汇编器：将汇编语言（指令、伪指令）转换为机器码

合法的汇编语言，汇编器可识别，但硬件不需要实现

$1	$at

Assembler temporary，留给汇编器的临时变量

可简化程序转换和编程，如：

move $t0, $t1 ->add $t0, $zero, $t1

blt $t0, $t1, L ->slt $?, $t0, $t1

bne $?, $zero, L

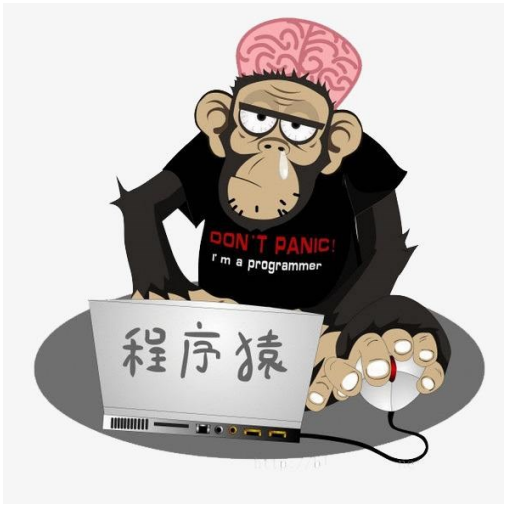
伪指令是合法的汇编语言，汇编器可识别，但硬件不需要实现

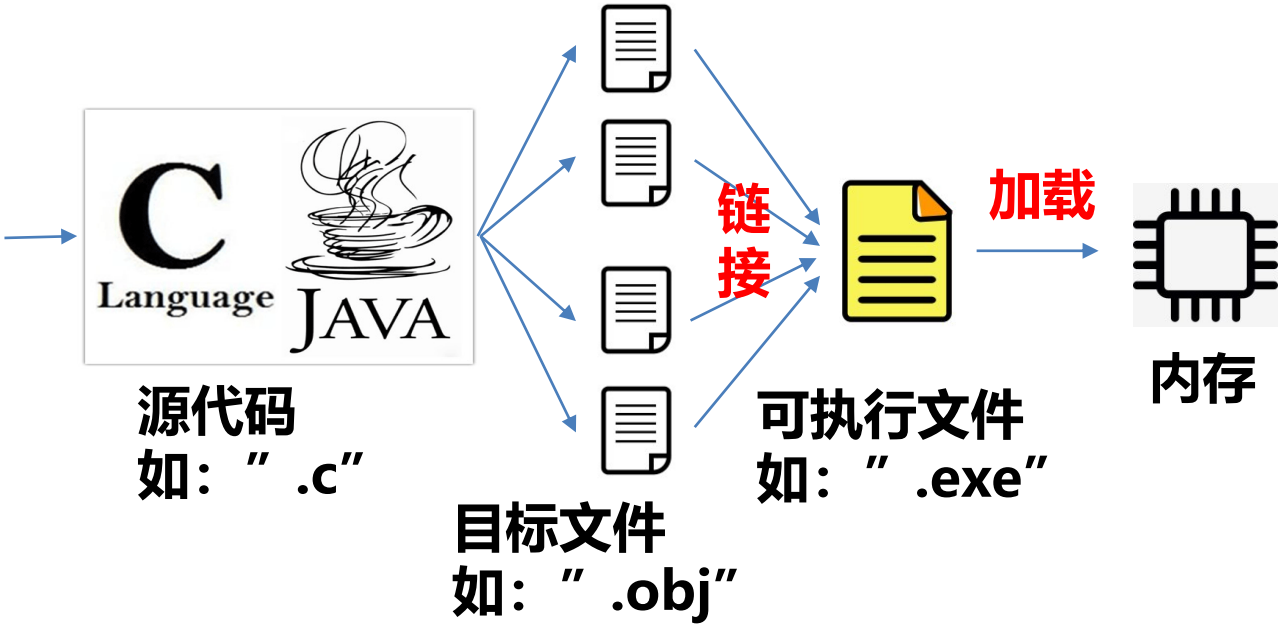
->丰富了汇编语言指令集，唯一的代价就是保留了一个寄存器$at

1.2 翻译并执行程序—链接与加载

问题：修改任意一行源代码都需要重新编译和汇编 ->严重浪费计算资源

解决方案：单独编译和汇编每个过程，生成独立的目标文件(object file)，然后把不同过程“缝合”在一起，形成可执行文件(executable file)





链接器将各个独立汇编的机器语言程序组合起来，生成一个可执行文件

加载器读取程序文件，把目标程序加载到内存中，以准备运行

1.2 翻译并执行程序—链接与加载

目标文件通常包括：

1、文件头：描述文件其它部分的大小和位置；

2、代码段：机器语言代码；

3、静态数据段：程序内分配的静态数据；

4、重定位信息：对指令字和数据字进行标识，以便程序装入内存时进行重定位(相对地址->绝对地址)；

5、符号表：未定义的符号；

6、调试信息：说明目标模块如何编译的简明描述（关联到C源文件，增加数据结构可读性）

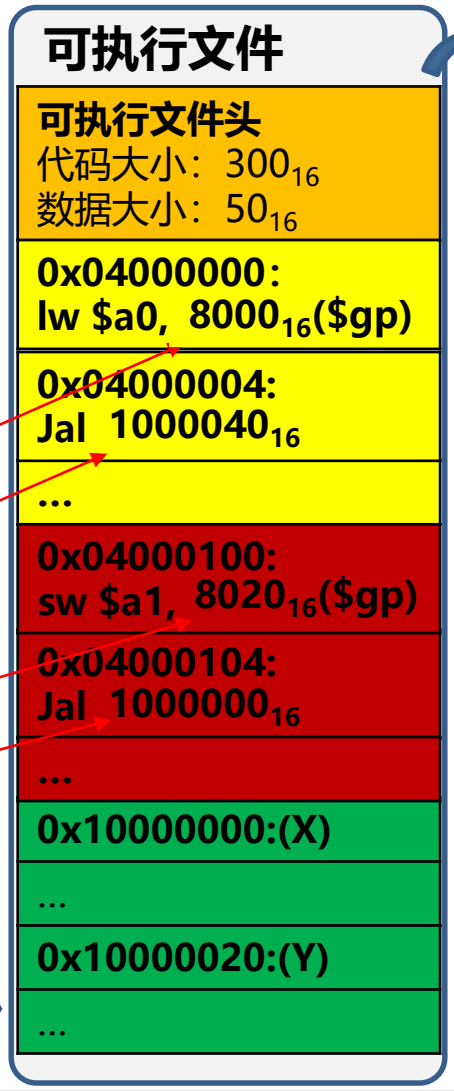
文件头	名称：过程A
	代码大小：100_{16}
	数据大小：20_{16}
代码段	地址0：Lw $a0, 0($gp)
	地址4：Jal 0
	…
静态数据段	地址0：(X)
	…
重定位信息	地址0：Lw ->X
	地址4：Jal->B
符号表	X：地址待定
	B：地址待定

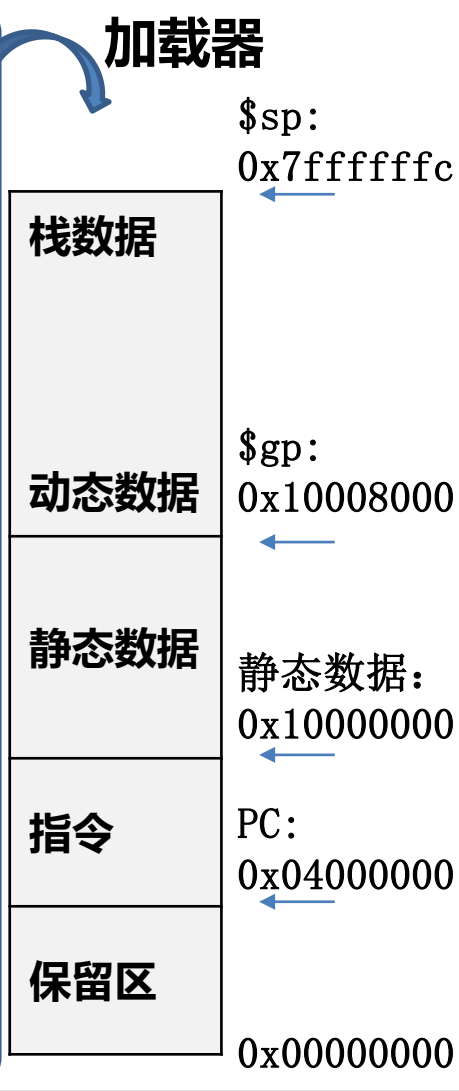
->链接器如何“缝合”目标文件?

1.2 翻译并执行程序—链接与加载

例：完成目标文件A->目标文件B的链接

文件头	名称：过程A	名称：过程B
	代码大小：100_{16}	代码大小：200_{16}
	数据大小：20_{16}	数据大小：30_{16}
代码段	地址0：Lw $a0, 0($gp)	地址0：sw $a1, 0($gp)
	地址4：Jal 0	地址4：Jal 0
	…	…
数据段	地址0：(X)	地址0：(Y)
	…	…
重定位信息	地址0：Lw ->X	地址0：sw ->Y
	地址4：Jal->B	地址4：Jal->A
符号表	X：地址待定	Y：地址待定
	B：地址待定	A：地址待定





内容概述

1. 执行程序

1.1 人机交互

1.2 翻译并执行程序

1.3 并行与同步

2. 汇编实战练习

2.1 几个设计实例

2.2 常见问题

1.3 并行与同步--多线程与多核

并行：什么是多线程？什么是多核？

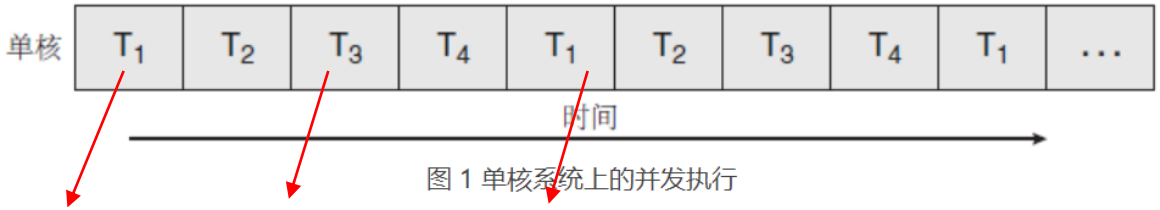


图 1 单核系统上的并发执行

a写入内存1

b写入内存1

读内存1

错误：读出的是b！

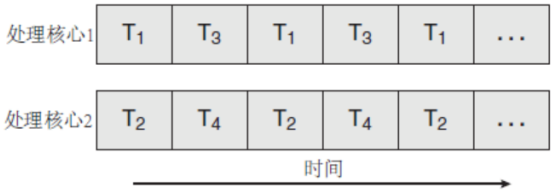
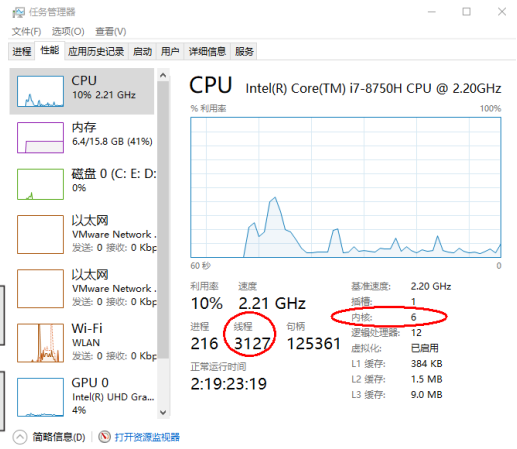


图 2 多核系统上的并行执行



多核系统：每个“核”都是一个独立CPU，都有自己的一套寄存器和ALU

内存是共享的！

多线程同时访问内存怎么办？

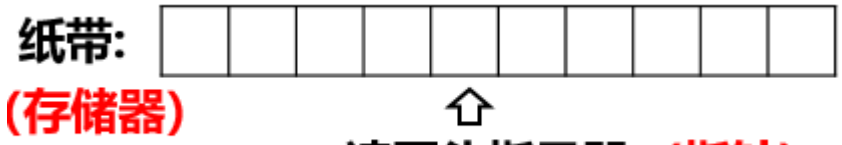
多个核同时访问内存怎么办？

->产生数据竞争

同步：任务之间需要协作，防止数据竞争 ->需要对内存加锁、解锁

1.3 并行与同步--原子操作

回顾现代计算机的基本原理： (图灵.论可计算数及其在判定问题中的应用,1936.)



每一步运算都是一个RMW过程：Read-Modify-Write

原子性：RMW过程如果对存储数据进行了保护，则是**原子操作**，具有原子性

可以(could)设计独立的原子操作指令 ->但会极大增加指令和硬件设计难度

更可行的解决方案：使用**原子指令对**，即硬件提供“原子读”“原子写”操作

“原子读”和“原子写”之间不允许其它线程或核访问被锁定的内存

可以利用原子读和原子写构造一系列原子操作（同步原语）

构造难度较大->通常系统程序员会建立**同步库**

需要用一个**标签**来标定内存是否被锁定

1.3 并行与同步—LL/SC指令对

MIPS提供原子读指令ll(Load Linked)和原子写指令sc(Store Conditionally)：

指令格式：ll rt, offset(rs)

sc rt, offset(rs)

标签：LLbit　LLaddr<-offset(rs)

LL指令：从内存取出数据，同时将LLbit至1(表示sc条件成立)

SC指令：if(LLbit==1){offset(rs)=rt;} rt=LLbit; LLbit=0;

例：原子操作—交换$s4和0($s1)中的数据

原子操作：(需要循环判定)

```
try: add $t0,$zero,$s4
     ll  $t1,0($s1)    #取数，并将LLbit置1，表示sc条件成立
     sc  $t0,0($s1)    #if(LLbit==1){0($s1)=$t0;} $t0=LLbit; LLbit=0;
     beq $t0,$zero,try
     add $s4,$zero,$t1
```

非原子操作：
```
lw $t1, 0($s1)
sw $s4, 0($s1)
add $s4, $zero, $t1
```

多线程，此时不一定为1！

内容概述

1. 执行程序

1.1 人机交互

1.2 翻译并执行程序

1.3 并行与同步

2. 汇编实战练习

2.1 几个设计实例

2.2 常见问题

2.1 设计实例1--swap

定义C函数swap，用于交换数组中的第k和第k+1个元素：

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

```
swap: sll $t1, $a1, 2   # $t1 = k * 4
      add $t1, $a0, $t1     #v[k]的地址
      lw $t0, 0($t1)        #取出v[k]
      lw $t2, 4($t1)        #取出v[k+1]
      sw $t2, 0($t1)        #存v[k+1]
      sw $t0, 4($t1)        #存v[k]
      jr $ra
```

v的首地址在$a0，k存于$a1，

参数temp存于$t0

--叶过程

2.1 设计实例2--sort

数组元素排序C函数sort：

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i – 1;
        j >= 0 && v[j] > v[j + 1];
        j -= 1) {
            swap(v,j);
            }
    }
}
```

分别传递到$s2和$s3 (比压栈快)

v的首地址存于$a0，n存于$a1，
i存于$s0，j存于$s1

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

v的首地址在$a0，k存于$a1

Caller保存调用后仍需使用(且callee需要写入)的$a和$t系列寄存器

Callee保存返回地址寄存器(叶过程除外)和需要使用的$s系列寄存器

```
sort:   addi $sp,$sp, –20   #压栈
        sw $ra, 16($sp)
        sw $s3,12($sp)
        sw $s2, 8($sp)
        sw $s1, 4($sp)
        sw $s0, 0($sp)
        move $s2, $a0       #传递参数
        move $s3, $a1
            …(for循环主体)
exit1:lw $s0, 0($sp)  #过程结束后，出栈
        lw $s1, 4($sp)
        lw $s2, 8($sp)
        lw $s3,12($sp)
        lw $ra,16($sp)
        addi $sp,$sp, 20
        jr $ra
```

2.1 设计实例2--sort

数组元素排序C函数sort：

```
void sort (int v[], int n)
{
   int i, j;
   for (i = 0; i < n; i += 1) {
      for (j = i – 1;
      j >= 0 && v[j] > v[j + 1];
      j -= 1) {
         swap(v,j);
         }
   }
}
```

分别传递到$s2和$s3 (比压栈快)

v的首地址存于$a0，n存于$a1，
i存于$s0，j存于$s1

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

v的首地址在$a0，k存于$a1

for循环主体

```
        move $s0, $zero   #i=0
for1:   slt  $t0, $s0, $s3   #判别i<n是否成立
        beq  $t0, $zero, exit1  #如果i<n不成立，则跳转到exit1
        addi $s1, $s0, –1  #j=i-1
for2:   slti $t0, $s1, 0  #判别j<0是否成立
        bne  $t0, $zero, exit2 #如果j<0成立，则跳转至exit2
        sll  $t1, $s1, 2
        add  $t2, $s2, $t1  #v[j]的地址
        lw   $t3, 0($t2)   #取出v[j]
        lw   $t4, 4($t2)   #取出v[j+1]
        slt  $t0, $t4, $t3   #判断v[j+1]<v[j]是否成立
        beq  $t0, $zero, exit2 #如果v[j+1]≥v[j]，则跳至exit2
        move $a0, $s2  #传递参数给swap函数
        move $a1, $s1
        jal  swap
        addi $s1, $s1, –1  #j=j-1
        j    for2
exit2:  addi $s0, $s0, 1
        j    for1
exit1:...
```

标记索引，每次for循环都要进行新的左移、相加计算，效率较低

->是否有更好解决方案?

*数组与指针

以数组A[k]为例，理解标记索引和指针索引的不同：

标记索引--以k为标记，索引数组，需要先计算4*k，再加上基址

指针索引--用一个指针直接指向数组(指针值为A[k]的地址)，则可以直接索引到A[k]

*数组与指针

分别用标记和指针的方式实现数组array清零：（假设array的地址和size存于$a0和$a1）

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

简单、直观

```
        move $t0,$zero
Loop:   slt $t3,$t0,$a1  #判别i<size
        beq $t3,$zero,exit
        sll $t1,$t0,2
        add $t2,$a0,$t1 #array[i]的地址
        sw $zero, 0($t2)
        addi $t0,$t0,1
        J Loop
exit:   …
```

效率较低：7*size+1

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];p = p + 1)
    *p = 0;
}
```

不易读

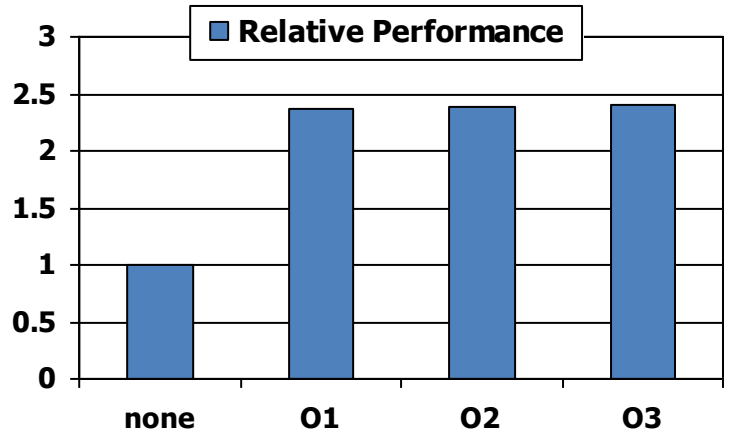
```
        move $t0,$a0  #p=&array[0]
        sll $t1,$a1,2   #$t1=size*4
        add $t2,$a0,$t1   #&array[size]
Loop:   slt $t3,$t0,$t2   #判定p<&array[size]
        beq $t3,$zero,exit
        sw $zero,0($t0)   #p指向的内存清理
        addi $t0,$t0,4   #p=p+4
        J Loop
exit:   …
```

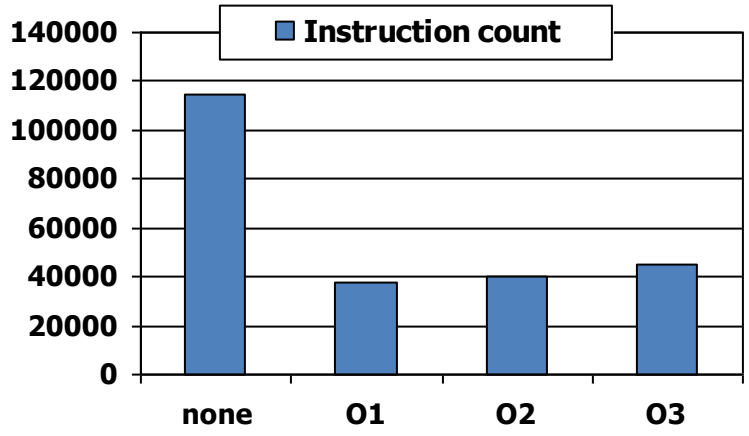
效率更高：5*size+3

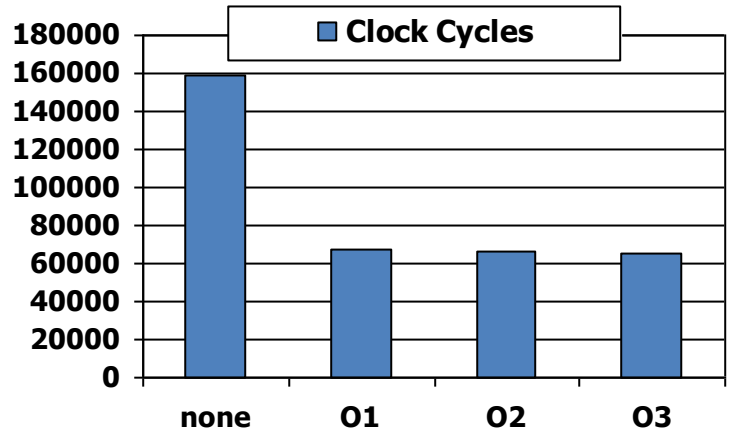
->当今阶段，先进的编译器可以解决效率问题，程序员的自由度更大

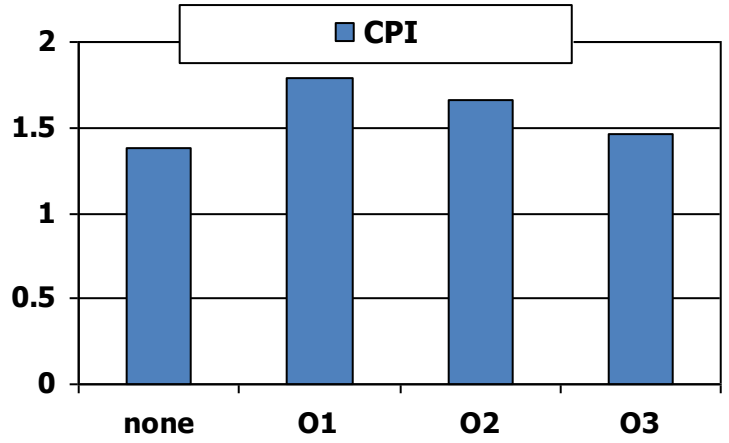
*编译器优化对性能的影响

对10000个字的数组排序：









内容概述

1. 执行程序

1.1 人机交互

1.2 翻译并执行程序

1.3 并行与同步

2. 汇编实战练习

2.1 几个设计实例

2.2 常见问题

2.2 常见问题

问题1：指令越强大，性能越好?

指令强大，则可以用较少的指令完成编程，但复杂的指令难于硬件实现，从而降低单个指令的执行速度

问题2：相比高级语言，使用汇编语言总是可以优化性能?

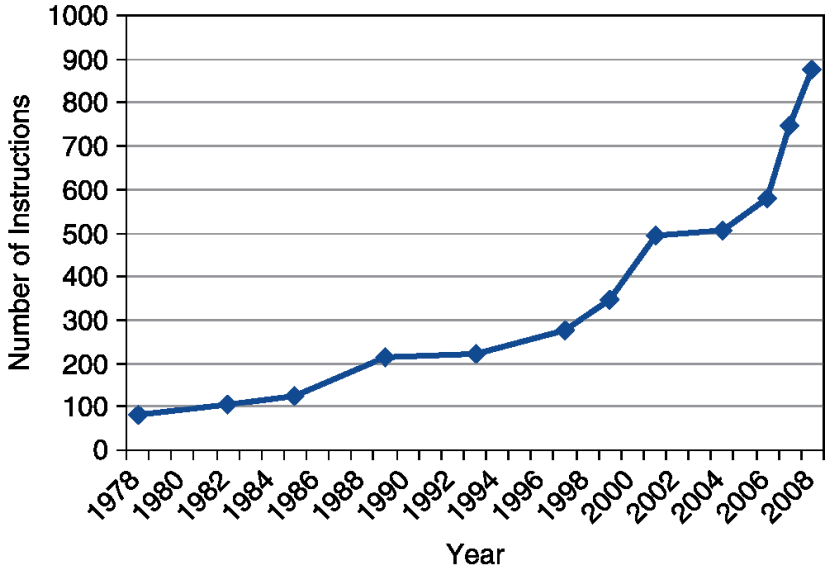
汇编程序员　　vs.　　编译器+C/java程序员

程序较大时，直接使用汇编语言容易产生更多错误

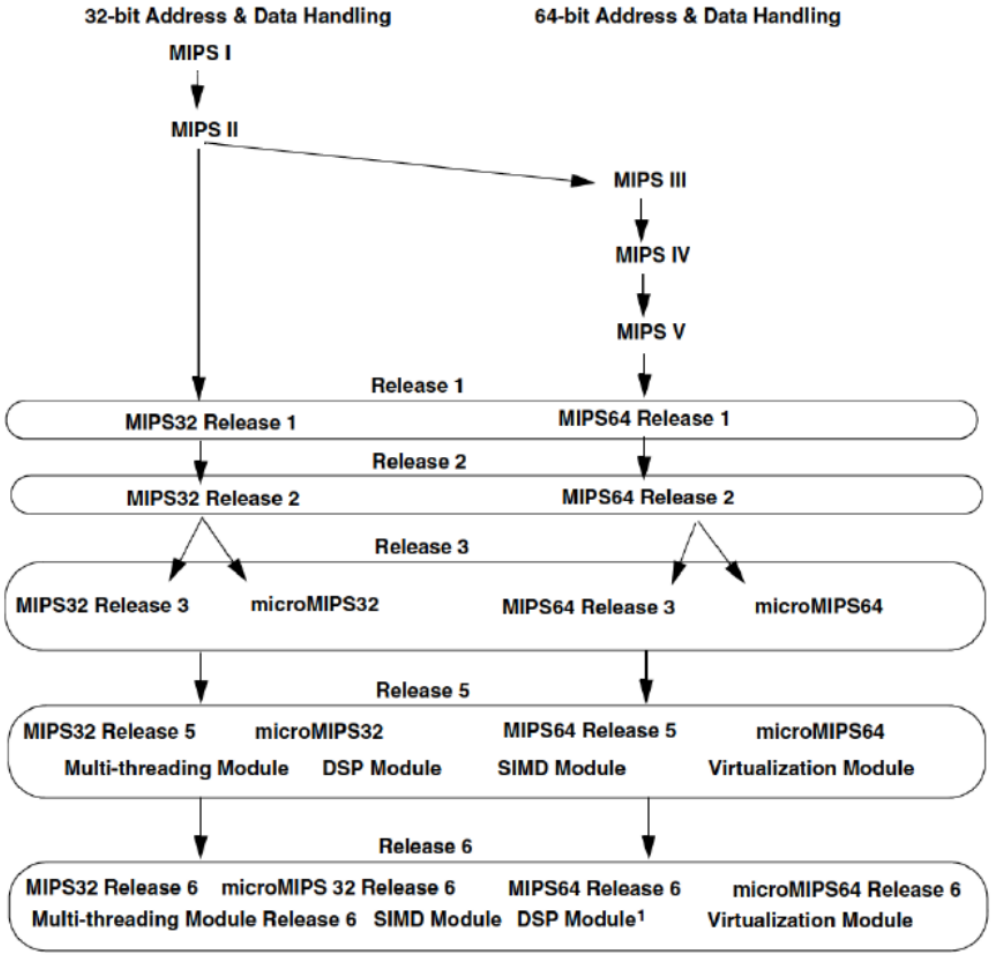
->降低整体性能、效率、可移植性、可维护性

2.2常见问题

问题3：因为商业计算机要求向下二进制兼容，所以成功的指令集不需要改变?



x86 指令集的发展



MIPS指令集的发展

本章小结

主流ISA

主流ISA(X86、ARM、MIPS、POWER、C6000)的发展、应用领域…

MIPS指令集体系结构与汇编语言

MIPS处理器中的操作数、寄存器、指令集、指令的二进制格式、程序存储…

硬件对过程的支持

支持过程的相关指令和寄存器、叶过程、嵌套过程、栈与帧、内存空间布局…

执行程序

关于人机交互、编译与汇编、链接与加载、并行与同步…

ISA设计原则：

1. 越规整、越简单、越容易实现、越快；
2. 越小(少)越快；
3. 加速常用操作的效果更好；
4. 好的设计需要好的折中