一个典型程序的转换处理过程

经典的“ hello.c ”C-源程序

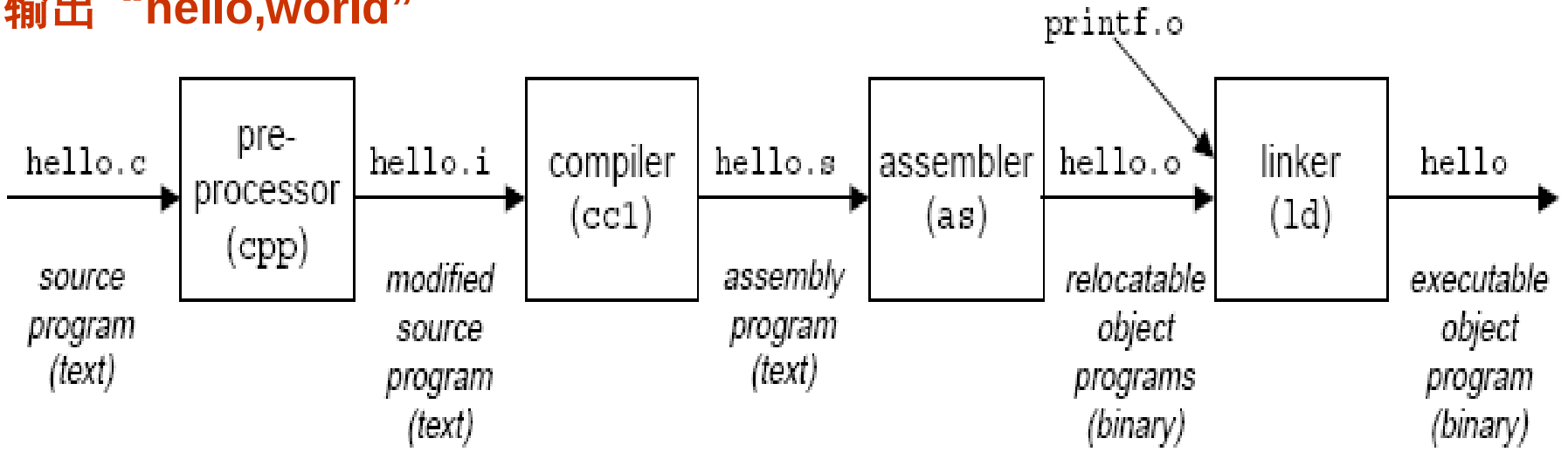
```
1 #include <stdio.h>
2
3 int main()
4 {
5 printf("hello, world\n");
6 }
```

hello.c的ASCII文本表示

```
# i n c l u d e <sp> < s t d i o .
35 105 110 99 108 117 100 101 32 60 115 116 100 105 111 46
h > \n \n i n t <sp> m a i n ( ) \n {
104 62 10 10 105 110 116 32 109 97 105 110 40 41 10 123
\n <sp> <sp> <sp> <sp> p r i n t f ( " h e l
10 32 32 32 32 112 114 105 110 116 102 40 34 104 101 108
l o , <sp> w o r l d \ n " ) ; \n }
108 111 44 32 119 111 114 108 100 92 110 34 41 59 10 125
```

程序的功能是：

输出“hello,world”

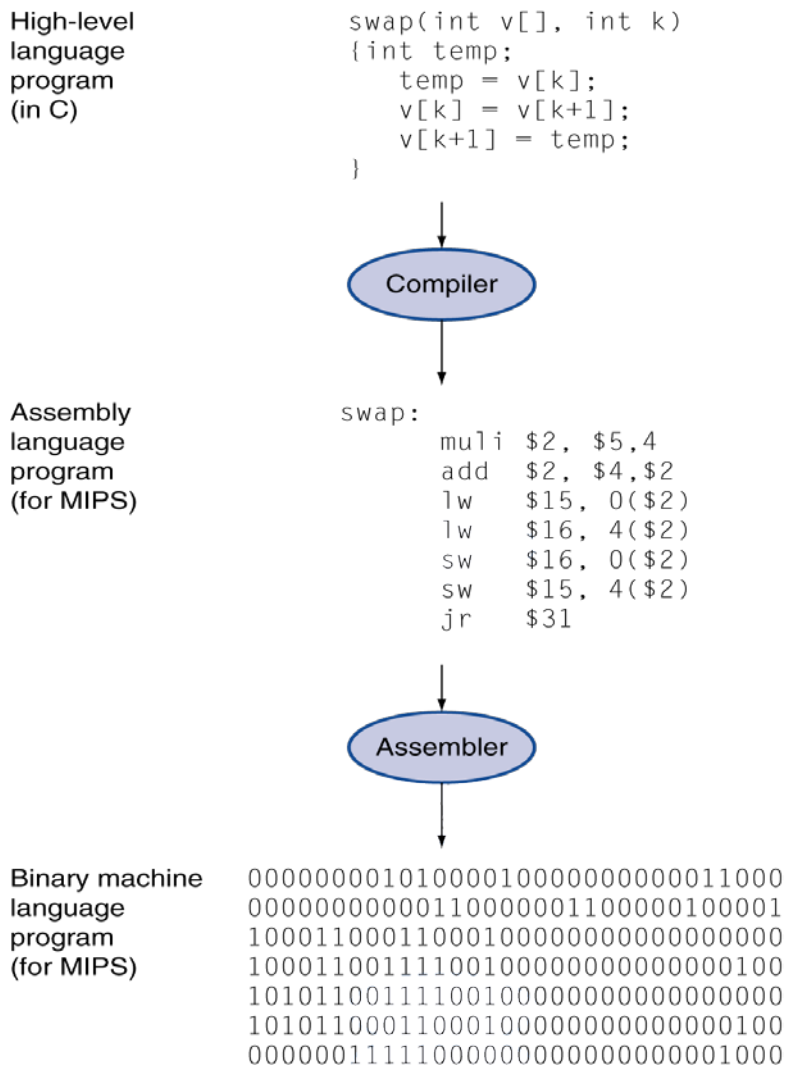


第三章 指令

计算机的机器语言

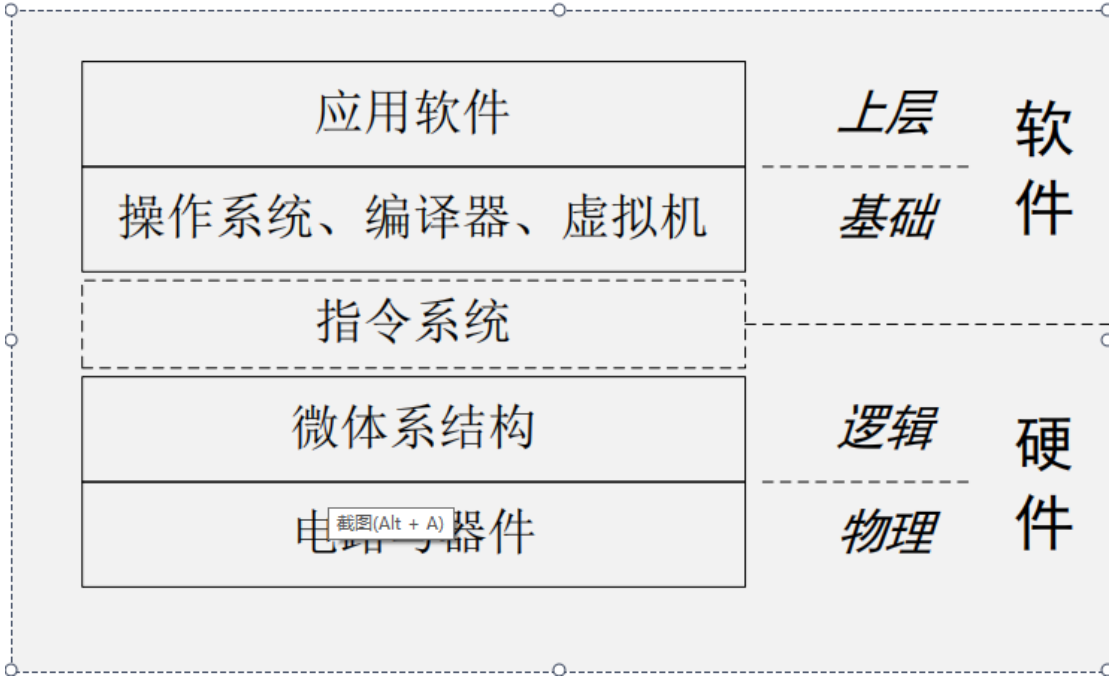
一个例子：从高级语言到机器语言

- 高级语言
  - 更接近问题领域的抽象级描述
  - 更高效且可移植性好
- 汇编语言
  - 机器指令的文本/助记符表示
- 机器语言
  - 一串二进制代码
  - 是计算机可以直接理解和执行的



指令&指令集

- 指令Instruction
  - CPU理解的“单词”
- 指令集Instruction Set
  - CPU理解的全部“单词”集合
- **指令集体系结构ISA**
  - **是软硬件的接口**



CISC/RISC

- *Complex Instruction Set Computer* (CISC)
  - 指令集设计早期阶段
  - 应用有什么操作模式，就增加对应的指令
  - 指令功能差异更大 不易于采用流水线
  - 微程序设计(80年代中后期，已日渐式微)
  - 典型代表 X86
- *Reduced Instruction Set Computer* (RISC)
  - 基于2-8定律 加速大概率事件
  - 硬布线的方式 更易设计 电路频率更高
  - 典型代表MIPS RISC V

MIPS

- **What**
  - **Microprocessor without Interlocked Pipeline Stage：无内锁流水线微处理器 1981年起源于Stanford大学 典型的学院派**
  - **主要用于嵌入式市场如消费电子、打印机、机顶盒等**
- **Why**
  - 真实：工业界实际使用的CPU
  - 简明：指令规整 层次清晰 结构简单 便于理解
  - 生态：软件开发环境丰富，易于学习和实践
    - 多种模拟器、C编译等
    - MIPS 32/64/16

龙芯中科

- 在MIPS指令集基础上设计了自主可控的指令集
- 产品包括龙芯1号/2号/3号三大系列
  - 安全领域如北斗导航
  - 通用领域如PC/Server/云计算等
  - 嵌入式领域
  - 工控产品等

MIPS32指令类型

- 运算类Computation
  - 算术运算类Arithmatic：加减乘除等
  - 逻辑运算类logic：and/or/not
  - 移位运算类 shift logic&arithmatic
- 存储类Load/Store
  - 从内存到寄存器的数据传送
- 分支跳转类 Branch & Jump
  - 分支类
  - 跳转类
- ...

MIPS运算类之指令格式

- 指令的一般格式
  - 操作码　操作数
- MIPS运算类采用了3操作数指令格式

  opcode　dst, src1, src2

  - opcode：表明指令功能，如加减乘除
  - dst：目的操作数（“destination”），来自于寄存器
  - src1：源操作数1（“source 1”），来自于寄存器
  - src2：源操作数2（“source 2”），可来自于寄存器(RR型）或立即数（RI型）

寄存器

- 32个32位寄存器
- 寄存器表示
  - $x（x为0~31），即$0~$31
- 寄存器名字
  - 程序员变量寄存器
    - **$s0-**$s7⟷$16**-$23**
  - 临时变量寄存器
    - **$t0-**$t7⟷$8**-$15**
    - **$t8-**$t9⟷$24**-$25**

注意
使用寄存器名字会让代码可读性更好

编号	名称	用途
0	$zero	常量0
1	$at	汇编器保留
2-3	$v0~$v1	返回值
4-7	$a0~$a4	参数
8-15	$t0-$t7	临时变量
16-23	$s0-$s7	程序变量
24-25	$t8-$t9	临时变量
26-27	$k0-$k1	操作系统 临时变量
28	$gp	全局指针
29	$sp	栈指针
30	$fp	帧框架指针
31	$ra	返回地址

本课程要学习的寄存器

0号寄存器

- 由于0在程序中的频度极高，为此MIPS设置了0号寄存器
  - 表示方法：$0或$zero
  - 值恒为0：读出的值恒为0；写入的值被丢弃
    - 指令的dst为$0：指令使用本身无错，但执行时没有实际意义
- 示例

```
1   add $s3, $0,  $0      # $S3=0
2   add $s1, $s2, $0      # $S1=$S2 实现寄存器值的复制
```

算数运算之加减法

指令zh	功能	格式	描述	示例
add	有符号加法 (检测溢出)	add rd, rs, rt	R[rd] = R[rs] + R[rt]	`add $s1, $s2, $s3`
addu	无符号加法 (不检测溢出)	add rd, rs, rt	R[rd] =R[rs] + R[rt]	`addu $s1, $s2, $s3`
addi	立即数加 (检测溢出)	addi rt, rs, imm16	R[rt] =R[rs] + sign_ext(imm16)	`addi $s1, $s2, -3`
addiu	立即数加 (不检测溢出)	addiu rt, rs, imm16	R[rt] = R[rs] + sign_ext(imm16)	`addiu $s1, $s2, 3`
sub	减法 (检测溢出)	sub rd, rs, rt	R[rd] =R[rs] - R[rt]	`sub $s1, $s2, $s3`
subu	减法 (不检测溢出)	sub rd, rs, rt	R[rd] =R[rs] - R[rt]	`subu $s1, $s2, $s3`

- 加法有4条指令：add，addu，addi，addiu
- 减法有2条指令：sub，subu

为什么没有subi和subiu？

算术运算类示例

- C 代码

```
f = (g + h) - (i -40);
```

  - 假定变量f, g,h ,i分别分配了寄存器 $s0, $s1, $s2, $s3

- 对应编译后的MIPS 代码

```
add $t0, $s1,$s2      # $t0 = g + h
addi $t1, $s3,-40     # $t1 = i-40
sub $s0, $t0, $t1     # f = $t0 - $t1
```

逻辑运算类指令

- 假设： a→$s1, b→$s2, c→$s3

指令	C	MIPS
与	`a = b & c;`	`and  $s1,$s2,$s3`
与立即数	`a = b & 0x1;`	`andi $s1,$s2,0x1`
或	`a = b | c;`	`or  $s1,$s2,$s3`
或立即数	`a = b | 0x5;`	`ori  $s1,$s2,0x5`
或非	`a = ~(b | c);`	`nor  $s1,$s2,$s3`
异或	`a = b ^ c;`	`xor  $s1,$s2,$s3`
异或立即数	`a = b ^ 0xF;`	`xori $s1,$s2,0xF`

- 为什么没有非运算？它可以怎么实现？

移位指令需考虑的问题

- 3个维度
  - 方向：左移or右移
  - 性质：逻辑移位or算术移位
    - 左移时，低位永远补0
    - 只有向右移位，才存在高位是补0还是符号位的问题
    - 若补0，即逻辑移位，若补符号位，则为算术移位
  - 移位量：对于32位机器字长的MIPS 32，移动位数的合理最大取值为31，即0x1F
    - 如何在指令中表示这个移位量呢？

移位指令

- 共6条移位指令
  - 若使用立即数表示移位次数：只有0~31有效
  - 若采用寄存器：寄存器的低5位有效（按5位无符号数对待）

指令	功能	示例
sll	逻辑左移	sll $t0, $s0, 1
srl	逻辑右移	srl $t0, $s0, 2
sra	算术右移	sra $t0, $s0, 3
sllv	逻辑可变左移	sllv $t0, $s0, $s1
srlv	逻辑可变右移	srlv $t0, $s0, $s1
srav	算术可变右移	srav $t0, $s0, $s1

空操作NOP(No Operation Performed)

- 空操作NOP是一条特殊指令
- 出于需要，编译器有时需在程序中插入NOP，如解决流水线冲突
- 在MIPS中，NOP指令并不存在
  - 当汇编器发现NOP时，会将其转换为

```
sll $0,$0,0
```

  - CPU执行这条指令时，只会解析指令语义，但产生任何实质操作
    - 1）$0值恒为0：故任何移位均无意义
    - 2）移位数为0：故不会产生移位操作
    - 3）$0不可写：故写入操作不会发生

- Thanks ！