

第一章 计算机系统概述

主要内容

- 计算机基本工作原理
- 程序的开发和执行过程
- 计算机系统层次结构
- 计算机性能评价

1.1 计算机基本工作原理

- 第一台通用电子计算机的诞生
- 冯·诺依曼结构的基本思想
- 程序和指令的执行过程

第一台通用电子计算机的诞生

手动式计算工具

算筹 -> 算盘

机械式计算工具 (1642年)

法国 帕斯卡 (Pascal) 加法器

德国 莱布尼兹 (Leibnitz) 四则运算器

机电式计算机 (1886年)

美国 霍勒瑞斯 (Hollerith)

穿孔卡片存储数据 成立的公司后改名为 IBM

电子计算机 (1939年)

参考网上资料：计算工具的发展简史

第一台通用电子计算机的诞生

1935-1939年，第一台电子数字计算机**样机 (ABC)** 研制成功

Atanasoff-Berry Computer

1946年，第1台**实际使用的**电子数字计算机 **ENIAC**诞生

–美国宾夕法尼亚大学研制

–用于解决复杂弹道计算问题

–5000次加法/s

–平方、立方、sin、cos等

–用**十进制**表示信息并运算

–采用**手动编程**，通过设置开关和插拔电缆来实现

Electronic
Numerical
Integrator
And
Computer

电子数字积分计算机

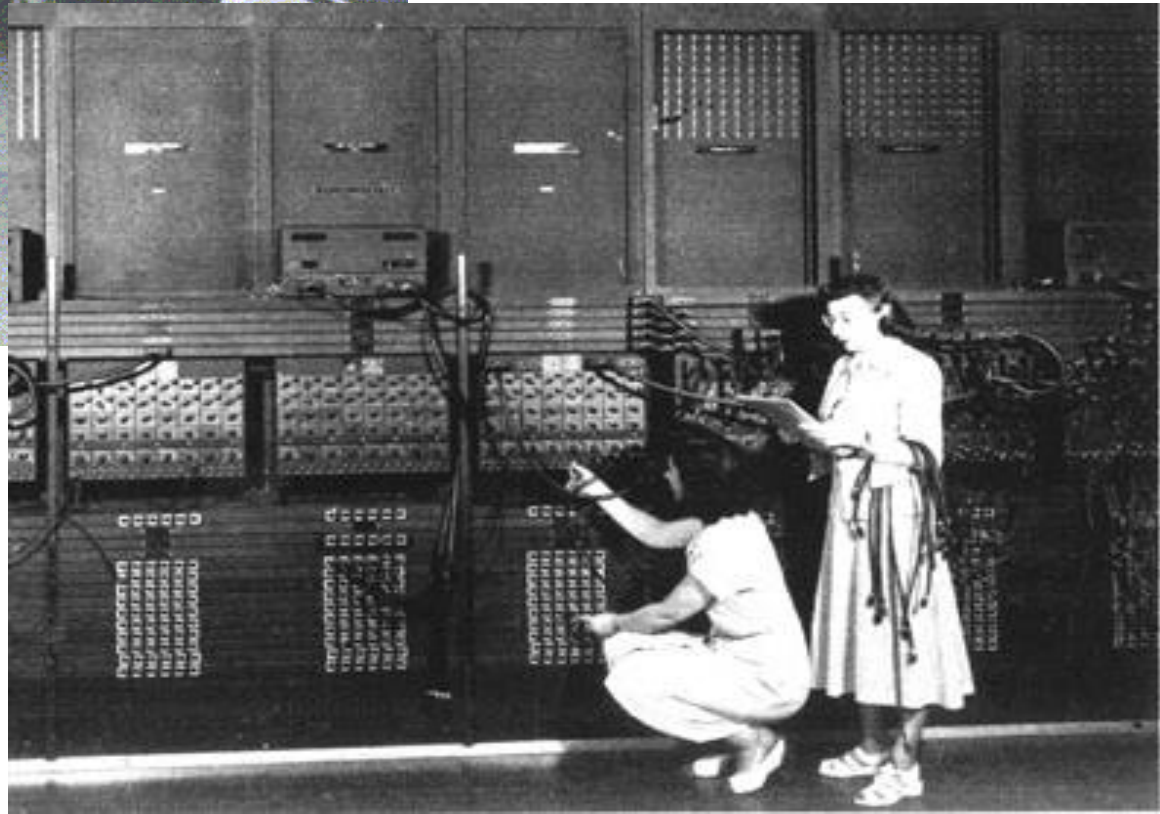
美国艾奥瓦州立大学约翰·文森特·阿塔那索夫 (John Vincent Atanasoff) 被称为 “**电子计算机之父**”

Electronic Numerical Integrator And Computer



占地170平方米，重30吨
有18000多个电子管
6000个开关
靠设置开关、连接插头
和插座来编程

每一位十进制数用
10个电子管表示，
十个电子管中哪个
亮表示几



冯·诺依曼

- 原籍匈牙利的数学家
- 计算机之父 和 博弈论之父
- 1944年，冯·诺依曼参加原子弹的研制工作，加入 *ENIAC* 研制组
- 1945年，冯·诺依曼以“关于EDVAC的报告草案”为题，发表了全新的“**存储程序通用电子计算机方案**”。
- 依据这份报告，**普林斯顿高等研究院**批准让冯·诺依曼建造计算机。

电子数字积分计算机 ENIAC

离散变量自动电子计算机 EDVAC



Electronic
Discrete
Variable
Automatic
Computer

现代计算机的原型

1946年，普林斯顿高等研究院（the Institute for Advance Study at Princeton, IAS）开始设计“**存储程序**”计算机，被称为**IAS计算机**（1951年才完成，它并不是第一台存储程序计算机，1949年由英国剑桥大学完成的EDSAC是第一台）。

- 在那个报告中提出的计算机结构被称为**冯·诺依曼结构**。

- **冯·诺依曼结构最重要的思想**

- “**存储程序(Stored-program)**” 工作方式：

任何要计算机完成的工作都要先被编写成程序，然后将程序和原始数据送入主存并启动执行。一旦程序被启动，计算机应能在不需操作人员干预下，自动完成逐条取出指令和执行指令的任务。

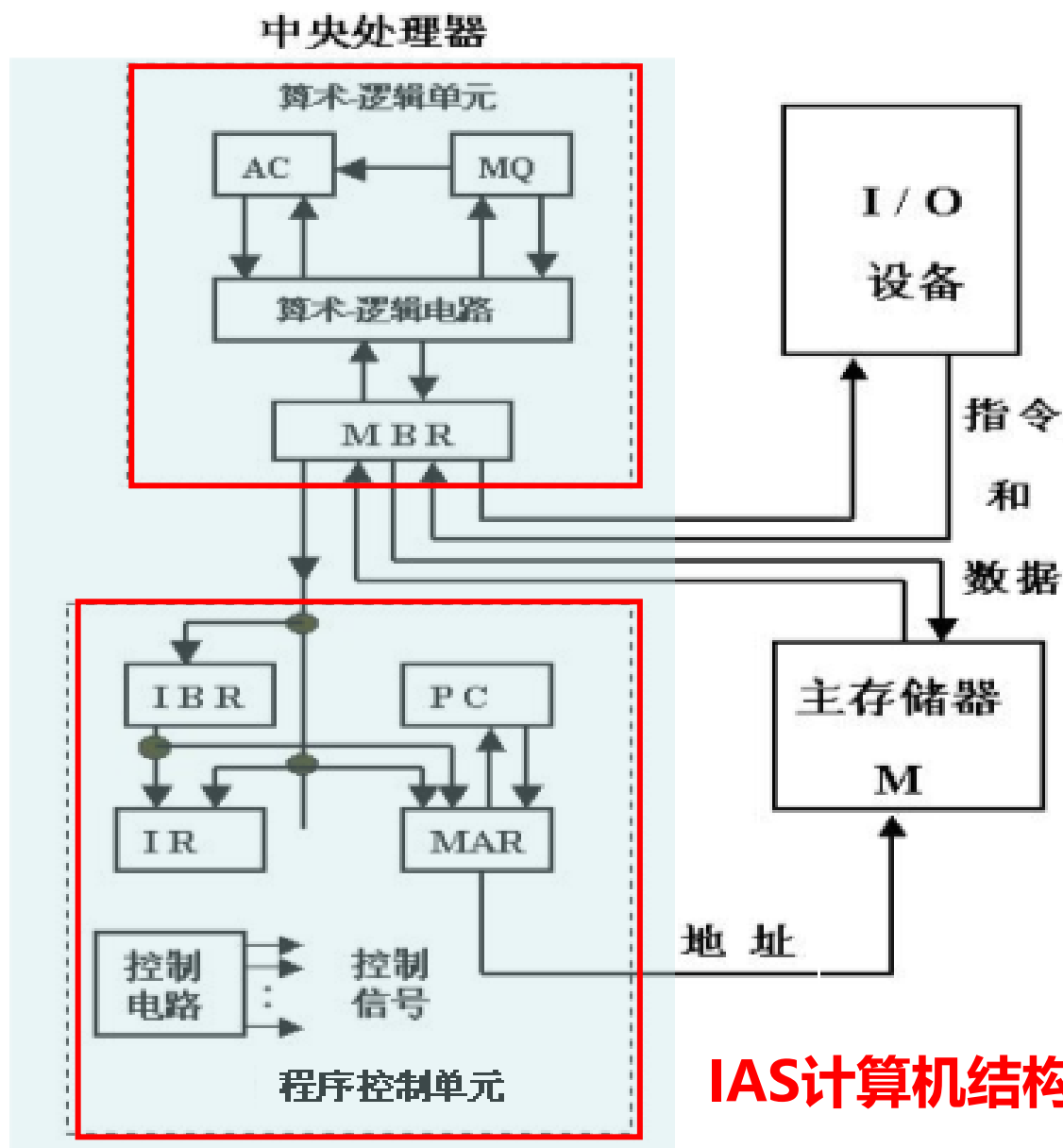
- **冯·诺依曼结构计算机也称为冯·诺依曼机器（Von Neumann Machine）。**

- **几乎现代所有的通用计算机大都采用冯·诺依曼结构，因此，IAS计算机是现代计算机的原型机。**

冯·诺依曼结构

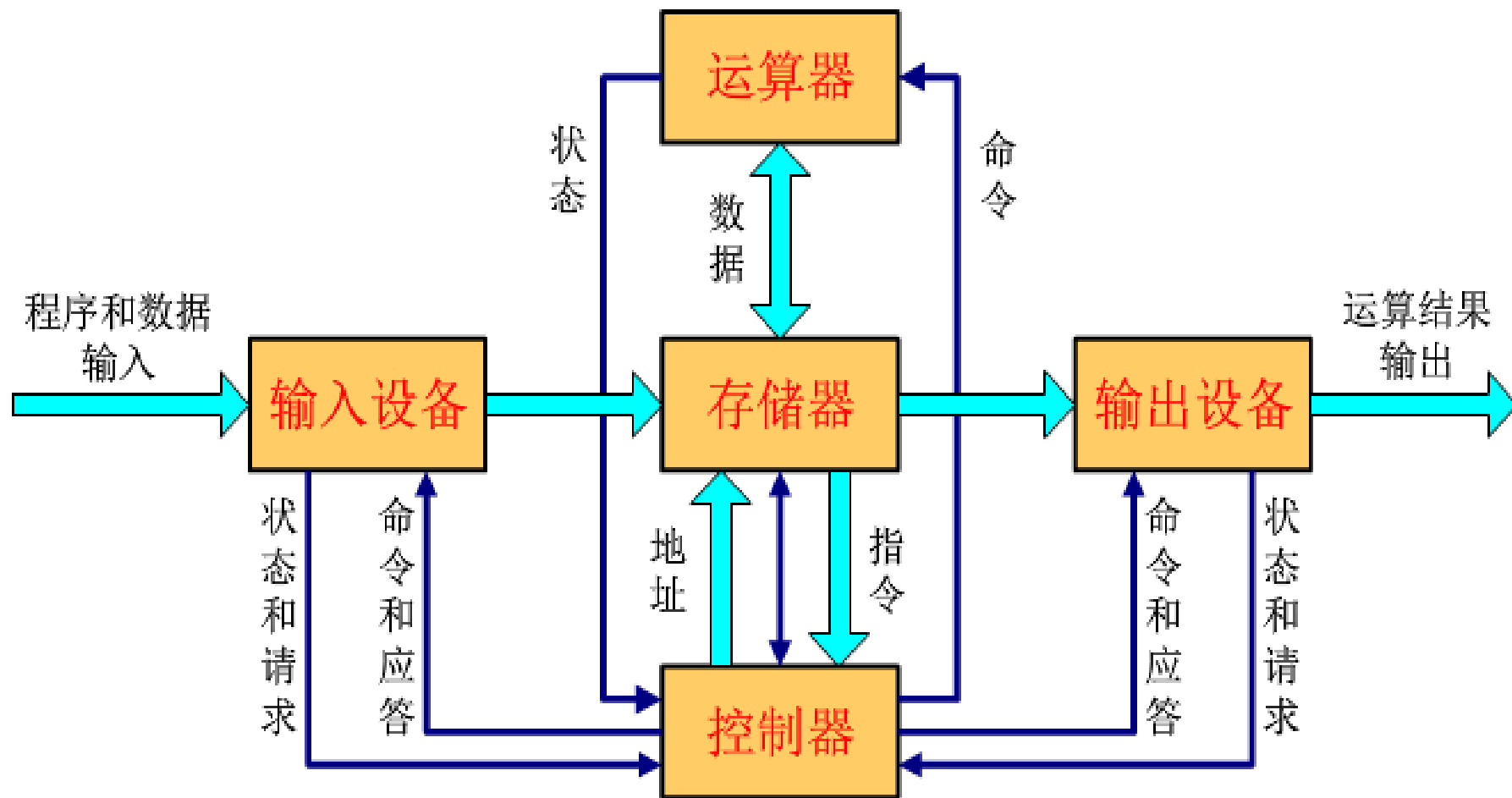
要实现存储程序的工作方式，对设备的要求

- 应该有个主存，用来存放程序和数据
- 应该有一个自动逐条取出指令的部件
- 还应该具体执行指令（即运算）的部件
- 应该有将程序和原始数据输入计算机的部件
- 应该有将运算结果输出计算机的部件



IAS计算机结构

冯.诺依曼结构计算机模型



早期，部件之间用**分散方式**相连

现在，部件之间大多用**总线方式**相连

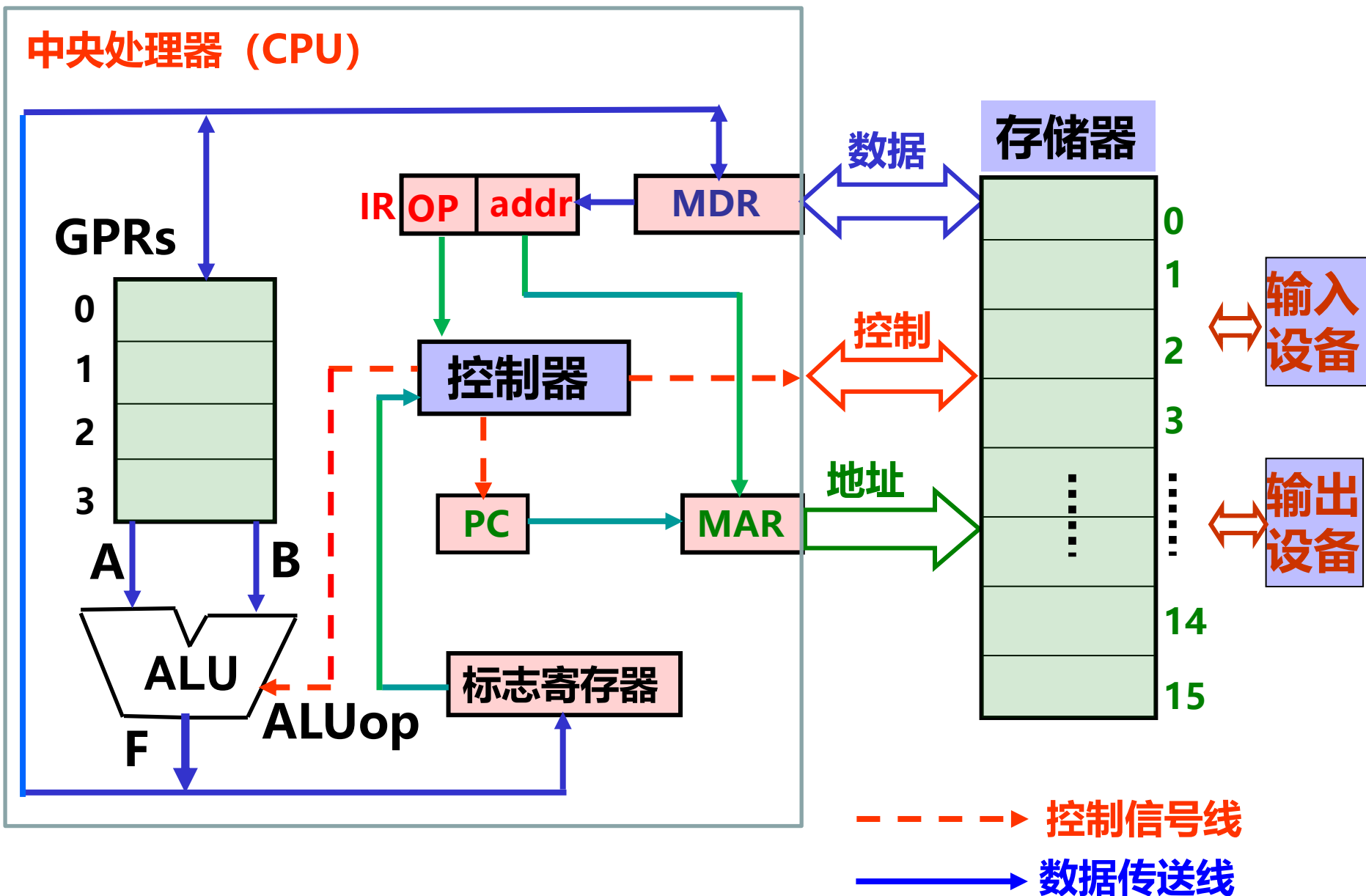
趋势，点对点（**分散方式**）高速连接

冯·诺依曼结构的主要思想

1. 计算机应由**运算器**、**控制器**、**存储器**、**输入设备**和**输出设备**五个基本部件组成。
2. 各基本部件的功能
 - **存储器**不仅能存放数据，而且也能存放指令，形式上两者没有区别，但计算机应能区分数据还是指令；
 - **控制器**应能自动取出指令来执行；
 - **运算器**应能进行加/减/乘/除四种基本算术运算，并且也能进行一些逻辑运算和附加运算；
 - 操作人员可以通过**输入设备**、**输出设备**和主机进行通信。
3. 内部以**二进制表示**指令和数据。每条指令由操作码和地址码两部分组成。操作码指出操作类型，地址码指出操作数的地址。由一串指令组成程序。
4. 采用“**存储程序**”工作方式。

现代计算机结构模型

中央处理器 (CPU)

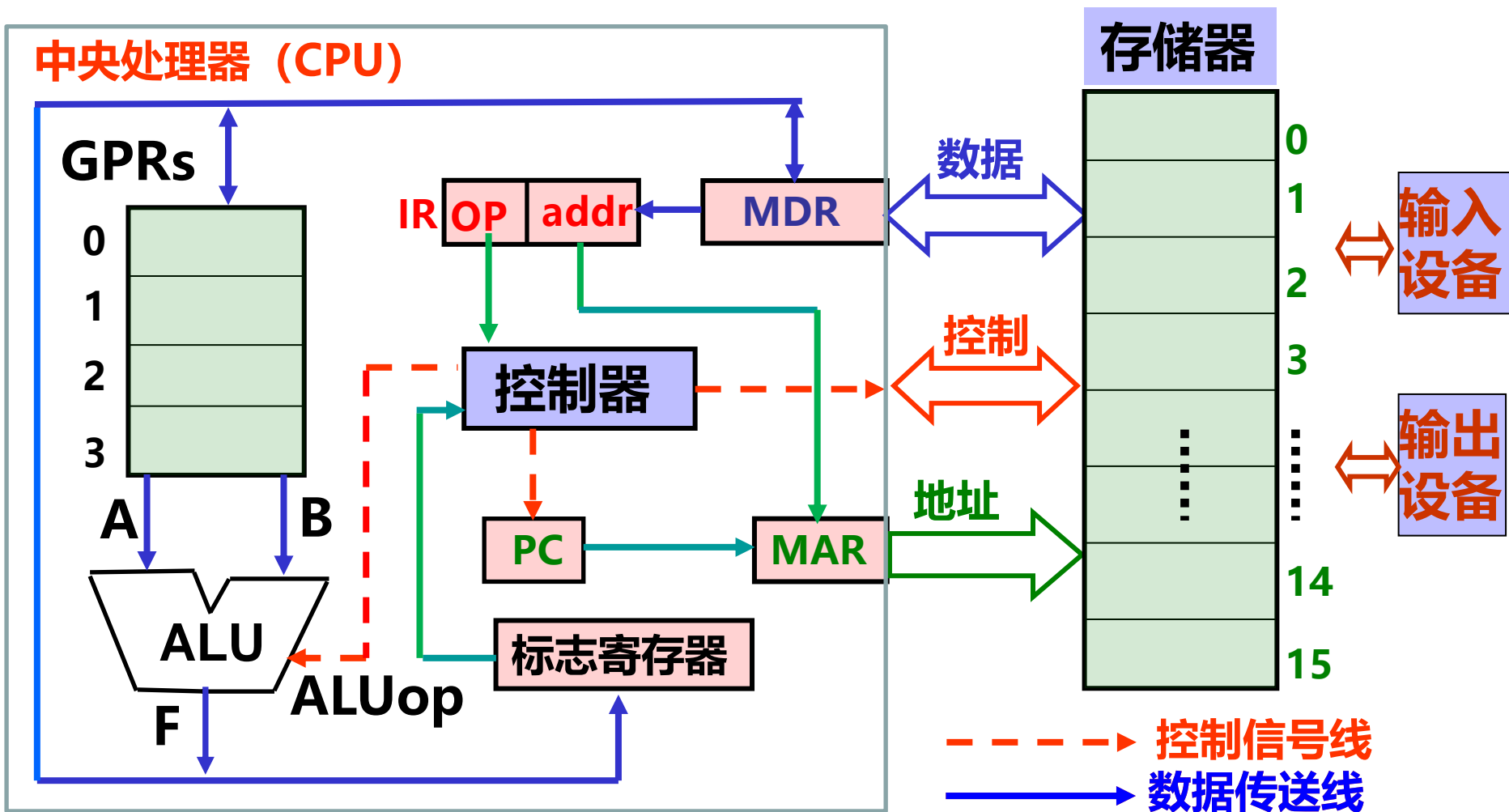


现代计算机结构模型

CPU: 中央处理器; **PC**: 程序计数器; **MAR**: 存储器地址寄存器

ALU: 算术逻辑部件; **IR**: 指令寄存器; **MDR**: 存储器数据寄存器

GPRs: 通用寄存器组 (由若干通用寄存器组成, 早期就是累加器)



计算机工作的基本过程

程序由指令组成，若所有指令执行完，则程序执行结束

- 程序在执行前

数据和指令事先存放在存储器中，每条指令和每个数据都有地址，指令按序存放，指令由OP、ADDR字段组成，程序起始地址置PC
开始执行程序

第一步：根据PC取指令

第二步：指令译码，修改PC的值

第三步：取操作数

第四步：指令执行（可能会再次修改PC的值）

第五步：回写结果

转第一步，继续执行下一条指令。

计算机工作的基本过程

- 指令和数据都存放在存储器中，形式上没有差别，都是0/1序列

指令中需给出的信息：

操作性质（操作码）

源操作数1 或/和 源操作数2 （立即数、寄存器编号、存储地址）

目的操作数地址 （寄存器编号、存储地址）

存储地址的描述与操作数的数据结构有关！

现代计算机结构 的发展

冯·诺依曼结构  哈佛体系结构

- 将数据和指令都存储在存储器中的计算机；
- 计算系统由一个中央处理单元（CPU）和一个存储器组成；
- 存储器拥有数据和指令，可以根据所给的地址对它进行读写；
- 程序指令和数据的宽度相同

Intel 8086、ARM7、MIPS处理器等

- 为数据和程序提供了各自独立的存储器；
- 程序计数器只指向程序存储器而不指向数据存储器，很难在哈佛机上编写出一个自修改的程序；
- 指令和数据可以有不同的数据宽度；

ARM9、ARM10、摩托罗拉 MC68 、Zilog Z8 等

现代计算机结构 的发展

- ARM : Advanced RISC Machines
- 1985年第一个ARM原型在英国剑桥诞生
- 公司 只设计芯片，设计了大量高性能、廉价、耗能低的RISC处理器
- 公司不生产芯片
- 它提供ARM技术知识产权（IP）核，将技术授权给世界上许多著名的半导体、软件和OEM厂商，并提供服务；
- ARM， 一个公司的名字
 - 一类微处理器的通称
 - 一种技术的名字

1.2 程序的开发与运行

最早的程序开发过程

- 用机器语言编写程序，并记录在纸带或卡片上

穿孔表示0，未穿孔表示1

输入：按钮、开关；所有信息都是0/1序列！
输出：指示灯等

假设：0010-jxx 转移指令

0: 0101 0110

1: 0010 0100

2:

3:

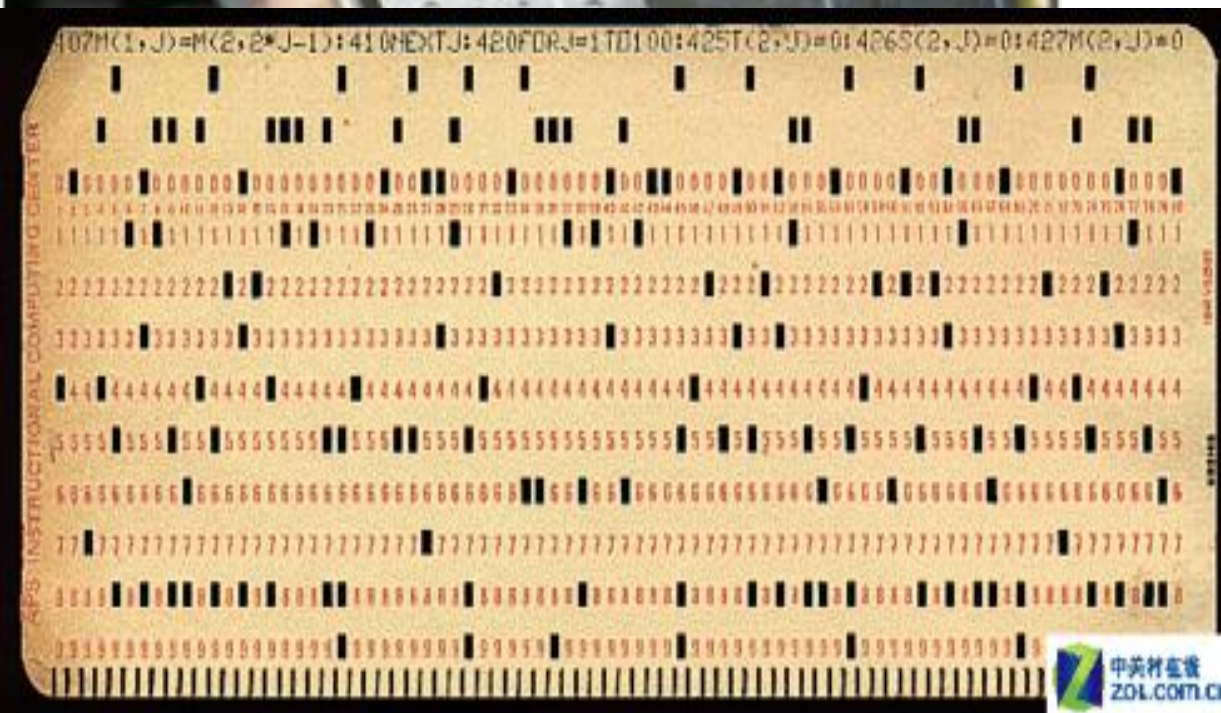
4: 0110 0111

5:

6:

若在第4条指令前加入指令，则需重新计算地址码（如jxx的目标地址），然后重新打孔。不灵活！

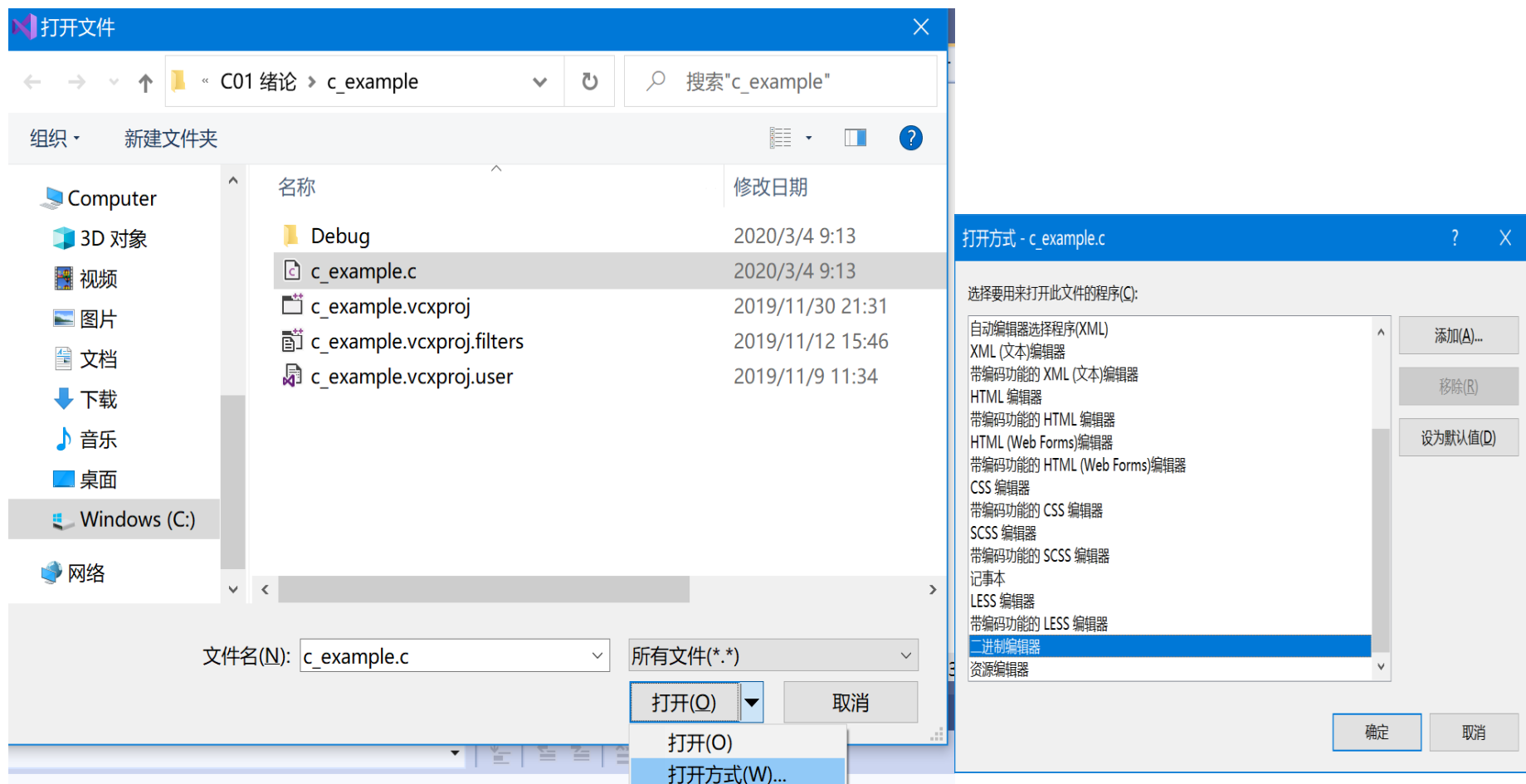
书写、阅读困难！



计算机是 0-1 的世界

```
#include <stdio.h>                                // 工程: c_example
int main(int argc, char* argv[])
{
    int  x, y, z;
    x = 10;      y = 20;
    z = 3 * x + 6 * y + 4 * 8;
    printf("3*%d+6*%d+4*8=%d\n", x, y, z);
    return 0;
}
```

计算机是 0-1 的世界



VS2019, 单击“文件”->“打开文件”, 在打开文件对话框中选择要打开的文件, 并在“打开方式”中选择“**二进制编辑器**”

计算机是 0-1 的世界

ASCII : American Standard Code for Information Interchange

c_example.c

00000000	23 69 6E 63 6C 75 64 65	20 3C 73 74 64 69 6F 2E	#include <stdio.
00000010	68 3E 0D 0A 0D 0A 69 6E	74 20 6D 61 69 6E 28 69	h>....int main(i
00000020	6E 74 20 61 72 67 63 2C	20 63 68 61 72 2A 20 61	nt argc, char* a
00000030	72 67 76 5B 5D 29 0D 0A	7B 0D 0A 09 69 6E 74 20	rgv[]).. {...int
00000040	20 78 2C 20 79 2C 20 7A	3B 0D 0A 09 78 20 3D 20	x, y, z;...x =
00000050	31 30 3B 0D 0A 09 79 20	3D 20 32 30 3B 0D 0A 09	10;...y = 20;...
00000060	7A 20 3D 20 33 20 2A 20	78 20 2B 20 36 20 2A 20	z = 3 * x + 6 *
00000070	79 2B 20 34 2A 38 3B 0D	0A 09 70 72 69 6E 74 66	y+ 4*8;...printf
00000080	28 22 33 2A 25 64 2B 36	2A 25 64 2B 34 2A 38 3D	("3*%d+6*%d+4*8=
00000090	25 64 5C 6E 22 2C 78 2C	79 2C 7A 29 3B 0D 0A 09	%d\n", x, y, z);...
000000a0	72 65 74 75 72 6E 20 30	3B 0D 0A 7D 0D 0A	return 0;...}..

用二进制编辑器打开源程序 c_example.c

计算机是 0-1 的世界



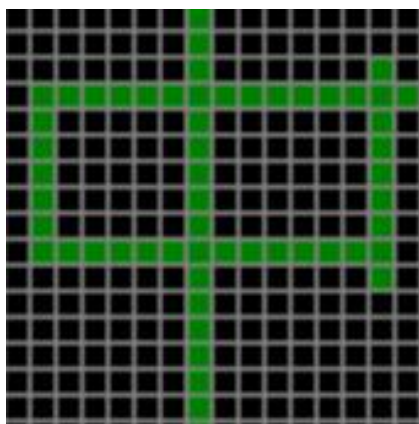
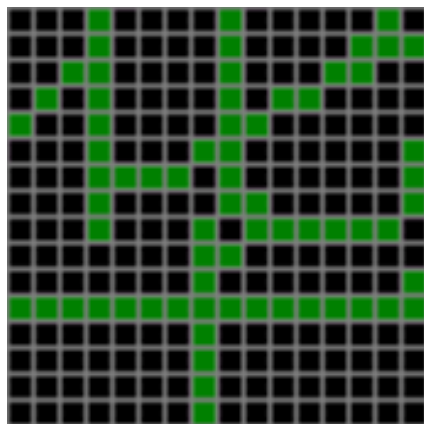
R: 227 G: 109 B: 106	R: 196 G: 213 B: 224	R: 198 G: 229 B: 247	R: 198 G: 229 B: 247	R: 198 G: 229 B: 247	R: 198 G: 229 B: 247	R: 198 G: 229 B: 247	R: 198 G: 229 B: 247	R: 198 G: 229 B: 247
R: 216 G: 147 B: 151	R: 196 G: 225 B: 244	R: 198 G: 232 B: 251	R: 198 G: 232 B: 251	R: 198 G: 232 B: 251	R: 198 G: 232 B: 251	R: 198 G: 232 B: 251	R: 198 G: 232 B: 251	R: 209 G: 184 B: 194
R: 209 G: 205 B: 221	R: 198 G: 232 B: 251	R: 198 G: 232 B: 251	R: 198 G: 232 B: 251	R: 198 G: 232 B: 251	R: 198 G: 232 B: 251	R: 198 G: 232 B: 251	R: 209 G: 184 B: 194	R: 239 G: 59 B: 45
R: 214 G: 191 B: 197	R: 198 G: 232 B: 251	R: 198 G: 232 B: 251	R: 198 G: 232 B: 251	R: 208 G: 198 B: 209	R: 213 G: 165 B: 171	R: 215 G: 169 B: 178	R: 231 G: 82 B: 73	R: 240 G: 33 B: 15
R: 222 G: 132 B: 133	R: 205 G: 215 B: 222	R: 198 G: 232 B: 251	R: 198 G: 232 B: 251	R: 216 G: 147 B: 151	R: 238 G: 49 B: 33	R: 240 G: 33 B: 15	R: 236 G: 64 B: 51	R: 224 G: 140 B: 144
R: 230 G: 115 B: 111	R: 208 G: 198 B: 209	R: 206 G: 231 B: 247	R: 206 G: 215 B: 230	R: 221 G: 124 B: 122	R: 233 G: 68 B: 57	R: 230 G: 76 B: 67	R: 216 G: 150 B: 154	R: 206 G: 231 B: 247
R: 198 G: 229 B: 247	R: 198 G: 229 B: 247	R: 198 G: 229 B: 247	R: 207 G: 227 B: 239	R: 214 G: 191 B: 197	R: 209 G: 205 B: 221	R: 206 G: 215 B: 230	R: 230 G: 115 B: 111	R: 223 G: 118 B: 116

计算机是 0-1 的世界

华 中 科 技 大 学

hua zhong ke ji da xue : 外码, 输入法编码

BB AA D6 D0 BF C6 BC BC B4 F3 D1 A7 : 内码



中 : 字形码 (字型码, 字模码, 输出码)

01 00 01 00 01 02 7F FF 41 02 41 02 41 02 41 02

41 02 7F FE 01 02 01 00 01 00 01 00 01 00 01 00

计算机是 0-1 的世界

计算机文件：0、1串 二进制编码

4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00
B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	00	E8	00	00	00
0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68
69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F
74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20
6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00

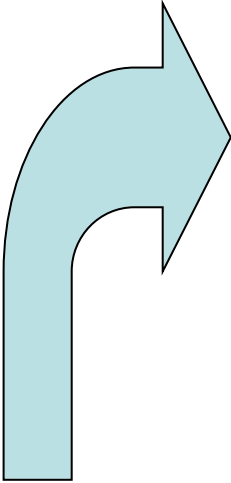
用二进制编辑器打开可执行文件 c_example.exe

计算机是 0-1 的世界

例：将 变量 **x** 中的内容 置为 10。

C7 45 F8 0A 00 00 00

思考题：计算机是**0**、**1**的世界，**0**，**1**串可代表什么含义？



1100	0111
0100	0101
1111	1000
0000	1010
0000	0000

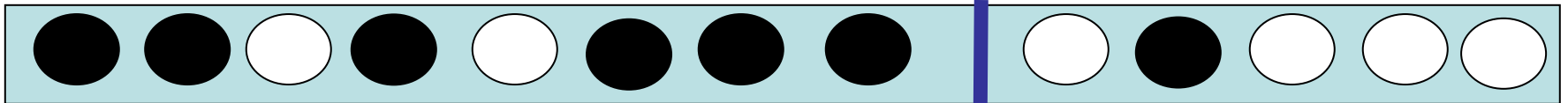
.....

0A 00 00 00 -> 10

F8 -> -8，局部变量**x**在栈中的位置；

45 -> 寻址方式

C7 -> 操作码



注意：在不同程序中，相同语句的编码不一定相同；
变量的地址不是固定的。

计算机真正懂得的语言是机器语言！

机器语言程序 太难读了！

机器语言程序 太难写了！

怎么办？

用汇编语言开发程序

机器语言 >> 汇编语言

反汇编

x = 10;

00251828 C7 45 F8 0A 00 00 00 mov dword ptr [ebp-8],0Ah

y = 20;

0025182F C7 45 EC 14 00 00 00 mov dword ptr [ebp-14h],14h

z = 3 * x + 6 * y + 4 * 8;

00251836 6B 45 F8 03	imul	eax,dword ptr [ebp-8],3
0025183A 6B 4D EC 06	imul	ecx,dword ptr [ebp-14h],6
0025183E 8D 54 08 20	lea	edx,[eax+ecx+20h]
00251842 89 55 E0	mov	dword ptr [ebp-20h],edx

用汇编语言开发程序

操作提示

The screenshot shows the Visual Studio interface during a debug session. The 'Debug' menu is open, and the 'Windows (W)' option is selected, which has opened a submenu. In this submenu, the 'Disassembly (D)' option is highlighted. The background shows the memory window with addresses 0x00B57000 and 0x00B57016, and the disassembly window showing assembly code for 'c_example'.

文件(F) 编辑(E) 视图(V) 项目(P) 生成(B) **调试(D)** 测试(S) 分析(N) 工具(T) 扩展(X) 窗口(W) 帮助(H) 搜索 (Ctrl+Q)

进程: [11932] c_example.exe

内存 1

地址: 0x00B57000

0x00B57000 ?? ?? ?? ?? ??

0x00B57016 ?? ?? ?? ?? ??

反汇编 asm_01_02.asm asm_01_01.asm

c_example

```
{
    int x;
    int y, z;
    x = 10;
    y = 20;
}
```

100 % 未找到相关问题

监视 1

搜索(Ctrl+E) 搜索深度

名称

窗口(W)

- 图形(C)
- 继续(C) F5
- 全部中断(K) Ctrl+Alt+Break
- 停止调试(E) Shift+F5
- 全部拆离(L)
- 全部终止(M)
- 重新启动(R) Ctrl+Shift+F5
- 应用代码更改(A) Alt+F10
- 性能探查器(F)... Alt+F2
- 重启性能探查器(L) Shift+Alt+F2
- 附加到进程(P)... Ctrl+Alt+P
- 其他调试目标(H)
- 逐语句(S) F11
- 逐过程(O) F10
- 跳出(T) Shift+F11
- 快速监视(Q)... Shift+F9
- 切换断点(G) F9
- 新建断点(B)
- 删除所有断点(D) Ctrl+Shift+F9
- 启用所有断点(N)

- 断点(B) Ctrl+Alt+B
- 异常设置(X) Ctrl+Alt+E
- 输出(O)
- 显示诊断工具(T) Ctrl+Alt+F2
- GPU 线程(U)
- 任务(S) Ctrl+Shift+D, K
- 并行堆栈(K) Ctrl+Shift+D, S
- 并行监视(R)
- 监视(W)
- 自动窗口(A) Ctrl+Alt+V, A
- 局部变量(L) Ctrl+Alt+V, L
- 即时(I) Ctrl+Alt+I
- 实时可视化树(V)
- 实时属性资源管理器(P)
- 调用堆栈(C) Ctrl+Alt+C
- 线程(H) Ctrl+Alt+H
- 模块(O) Ctrl+Alt+U
- 进程(P) Ctrl+Alt+Z
- 内存(M)
- 反汇编(D) Ctrl+Alt+D**

调试-» 窗口-» 反汇编

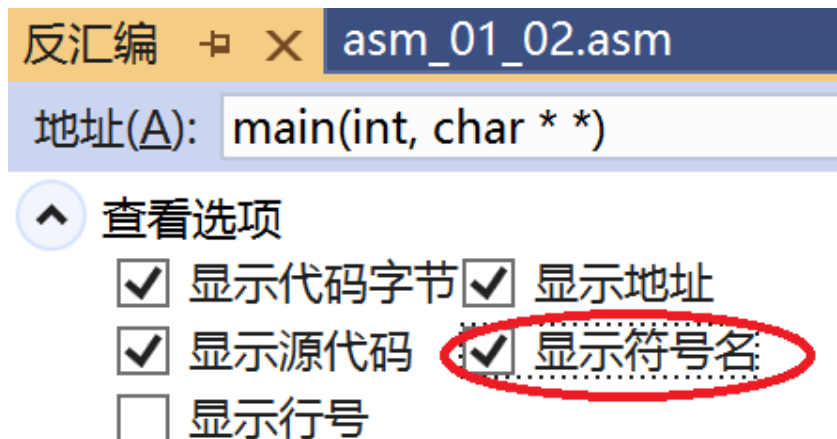
用汇编语言开发程序

操作提示

```
x = 10;  
00FB1838 C7 45 F8 0A 00 00 00 mov     dword ptr [ebp-8], 0Ah  
y = 20;  
00FB183F C7 45 EC 14 00 00 00 mov     dword ptr [ebp-14h], 14h
```

```
x = 10;  
00FB1838 C7 45 F8 0A 00 00 00 mov     dword ptr [x], 0Ah  
y = 20;  
00FB183F C7 45 EC 14 00 00 00 mov     dword ptr [y], 14h
```

反汇编窗口下，有查看选项。勾选不同的项目，看到的信息有所差异。



用汇编语言开发程序

比较不同环境下，汇编语句的差异

```
root@LAPTOP-CJLSTBTI: /home
Microsoft Windows [版本 10.0.19043.1526]
(c) Microsoft Corporation。保留所有权利。

C:\WINDOWS\system32>bash
root@LAPTOP-CJLSTBTI:/mnt/c/WINDOWS/system32# cd /
root@LAPTOP-CJLSTBTI:/# ls
bin    dev    home   lib     media  opt     root   sbin    srv    tmp    var
boot   etc    init   lib64   mnt     proc    run    snap    sys    usr
root@LAPTOP-CJLSTBTI:/# cd home
root@LAPTOP-CJLSTBTI:/home# ls
test  test.asm  test.c  test.i  test.o  test.s  test_intel.s
root@LAPTOP-CJLSTBTI:/home#
```

用汇编语言开发程序

执行: **#gcc -S -g test.c -o test.s**
#vi test.s

```
.loc 1 5 0
    movl    $10, -12(%rbp)
.loc 1 6 0
    movl    $20, -8(%rbp)
.loc 1 7 0
    movl    -12(%rbp), %edx
    movl    %edx, %eax
    addl    %eax, %eax
    leal    (%rax,%rdx), %ecx
    movl    -8(%rbp), %edx
    movl    %edx, %eax
    addl    %eax, %eax
    addl    %edx, %eax
    addl    %eax, %eax
    addl    %ecx, %eax
    addl    $32, %eax
    movl    %eax, -4(%rbp)
```

x = 10;

y = 20;

z = 3 * x + 6 * y + 4*8;

x → edx → eax

eax+eax → eax

eax+edx → ecx

y → edx → eax

edx+eax → eax

edx+eax → eax

eax+eax → eax

eax+ecx → eax

eax+32 → eax

eax → z

用汇编语言开发程序

执行: **#gcc -S -g test.c -masm=intel -o test_intel.s**
#vi test_intel.s

```
.loc 1 5 0
mov     DWORD PTR -12[rbp], 10
.loc 1 6 0
mov     DWORD PTR -8[rbp], 20
.loc 1 7 0
mov     edx, DWORD PTR -12[rbp]
mov     eax, edx
add     eax, eax
lea     ecx, [rax+rdx]
mov     edx, DWORD PTR -8[rbp]
mov     eax, edx
add     eax, eax
add     eax, edx
add     eax, eax
add     eax, ecx
add     eax, 32
mov     DWORD PTR -4[rbp], eax
```

x = 10;

y = 20;

z = 3 * x + 6 * y + 4 * 8;

x → edx → eax

eax+eax → eax

eax+edx → ecx

y → edx → eax

edx+eax → eax

edx+eax → eax

eax+eax → eax

eax+ecx → eax

eax+32 → eax

eax → z

用汇编语言开发程序

比较不同环境下，汇编语句的差异

x=10;

AT&T 格式 gcc, objdump 的默认格式
 movl \$10, -12(%rbp)

Intel 格式

 mov DWORD PTR -12[rbp], 10

- 源操作数地址 与 目的操作数地址顺序相反
- 寄存器的表示方法不同 rbp ↔ %rbp
- 寄存器间接寻址的表达方式不同 [rbp] ↔ (%rbp)
- 立即数的表达方式不同
- 操作符表示不同
 movl : move long

用汇编语言开发程序

执行: **#objdump -d test > test_dump.txt**

执行程序反汇编输出到 文件 **test_dump.txt** 中

c7 45 f4 0a 00 00 00	movl \$0xa,-0xc(%rbp)	x = 10;
c7 45 f8 14 00 00 00	movl \$0x14,-0x8(%rbp)	y = 20;
8b 55 f4	mov -0xc(%rbp),%edx	
89 d0	mov %edx,%eax	z = 3 * x + 6 * y + 4*8;
01 c0	add %eax,%eax	
8d 0c 10	lea (%rax,%rdx,1),%ecx	x → edx → eax
8b 55 f8	mov -0x8(%rbp),%edx	eax+eax → eax
89 d0	mov %edx,%eax	eax+edx → ecx
01 c0	add %eax,%eax	
01 d0	add %edx,%eax	y → edx → eax
01 c0	add %eax,%eax	edx+eax → eax
01 c8	add %ecx,%eax	edx+eax → eax
83 c0 20	add \$0x20,%eax	eax+eax → eax
89 45 fc	mov %eax,-0x4(%rbp)	
		eax+ecx → eax
		eax+32 → eax
		eax → z

用汇编语言开发程序

执行: **#objdump -S test**

执行程序反汇编, 列出了源程序。

```
int x,y,z;
x=10;
8:  c7 45 f4 0a 00 00 00    movl  $0xa,-0xc(%rbp)
y=20;
f:  c7 45 f8 14 00 00 00    movl  $0x14,-0x8(%rbp)
z=x*3+y*6+4*8;
16: 8b 55 f4                mov    -0xc(%rbp),%edx
19: 89 d0                  mov    %edx,%eax
1b: 01 c0                  add    %eax,%eax
1d: 8d 0c 10               lea    (%rax,%rdx,1),%ecx
20: 8b 55 f8                mov    -0x8(%rbp),%edx
23: 89 d0                  mov    %edx,%eax
25: 01 c0                  add    %eax,%eax
27: 01 d0                  add    %edx,%eax
29: 01 c0                  add    %eax,%eax
2b: 01 c8                  add    %ecx,%eax
2d: 83 c0 20               add    $0x20,%eax
30: 89 45 fc                mov    %eax,-0x4(%rbp)
```

用汇编语言开发程序

```
x = 10;  
00FB1838 C7 45 F8 0A 00 00 00 mov     dword ptr [ebp-8], 0Ah  
y = 20;  
00FB183F C7 45 EC 14 00 00 00 mov     dword ptr [ebp-14h], 14h
```

```
c7 45 f4 0a 00 00 00      movl    $0xa,-0xc(%rbp)  
c7 45 f8 14 00 00 00      movl    $0x14,-0x8(%rbp)
```

比较 VS2019 下看到的机器码 与
Ubuntu 下用 objdump 看到的机器码

同样的机器码，对应不同的汇编语句；
机器码由CPU决定；汇编语句由编译器决定
汇编语句格式：AT&T, intel

用汇编语言开发程序

- 用助记符表示操作码
- 用标号表示位置
- 用变量表示操作对象
- 用助记符表示寄存器
-

0:	0101 0110	sub B
1:	0010 0100	jnz L0
2:
3:
4:	0110 0111	L0: add C
5:
6:	B:
7:	C:

用汇编语言编写的优点是:

不需记忆指令码, 编写方便

可读性比机器语言强

不会因为增减指令而需要修改其他指令

在第4条指令前加指令时不用改变sub、jnz和add指令中的地址码!

需将汇编语言转换为机器语言!

用汇编程序转换

机器级语言

机器语言和汇编语言都是面向机器结构的语言，

它们统称为 **机器级语言**，都属于 **低级语言**

- 汇编语言**源**程序 由**汇编指令**构成



用汇编语言编程比机器语言好，但还是很麻烦！

用高级语言开发程序

- 它们与具体机器结构无关
- 面向算法描述，比机器级语言描述能力强得多
- 高级语言中一条语句对应几条、几十条甚至几百条指令
- 有“面向过程”和“面向对象”的语言之分
- 有两种转换方式：“编译”和“解释”
 - 编译程序(Compiler): 将高级语言源程序转换为机器级目标程序，执行时只要启动目标程序即可
 - 解释程序(Interpreter): 将高级语言语句逐条翻译成机器指令并立即执行，不生成目标文件。

编译程序的工作细分：词法分析、语法分析、语义分析、中间代码生成、代码优化、目标程序生成

一个典型程序的转换处理过程

经典的 “hello.c” C-源程序

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
}
```

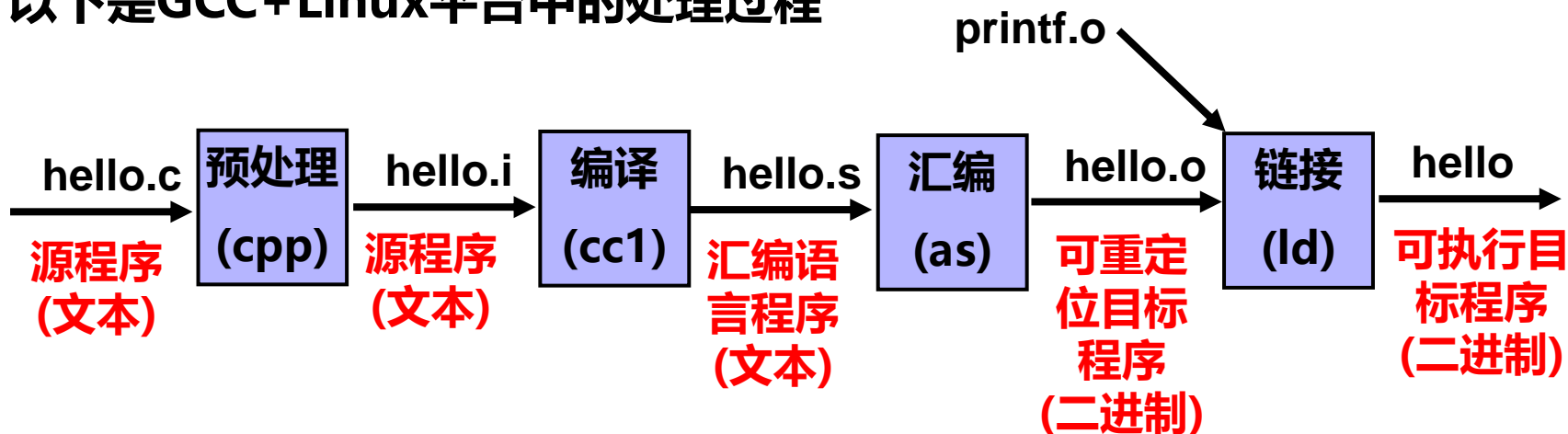
hello.c的ASCII文本表示

```
# i n c l u d e < s p > < 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 < s p > m a i n ( ) \n {
104 62 10 10 105 110 116 32 109 97 105 110 40 41 10 123
\n < s p > < s p > < s p > < s p > 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 , < s p > w o r l d \ n " ) ; \n }
108 111 44 32 119 111 114 108 100 92 110 34 41 59 10 125
```

功能：输出 “hello,world”

计算机不能直接执行hello.c!

以下是GCC+Linux平台中的处理过程



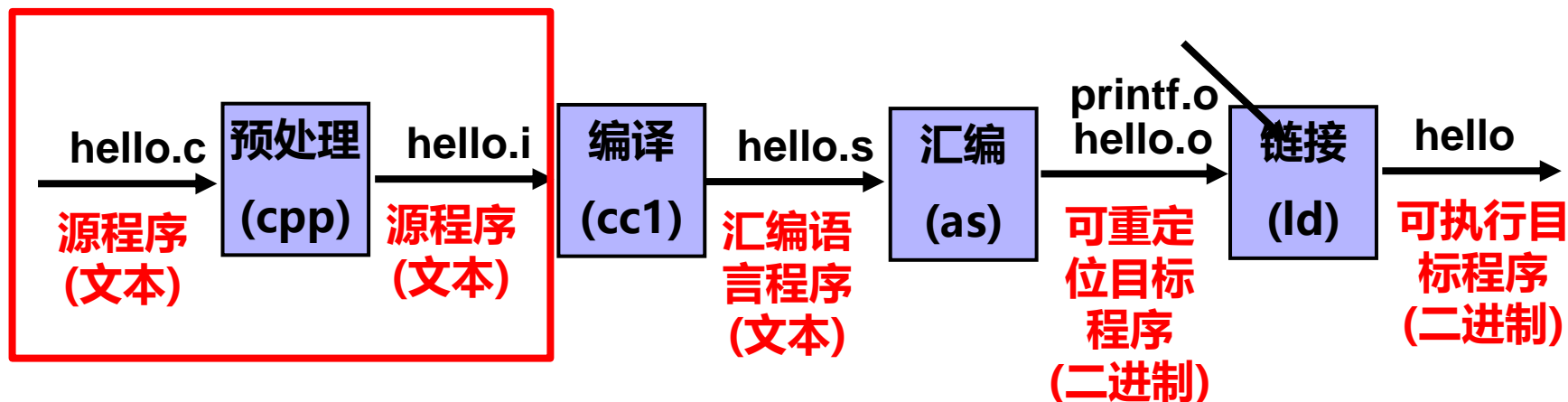
一个典型程序的转换处理过程

➤ 预处理

`#include` 头文件

`#define` 宏

条件编译



一个典型程序的转换处理过程

- 预处理指令 是以 # 号开头的代码行
- # 号必须是该行除了任何空白字符外的第一个字符
- # 后是指令关键字，在关键字和 # 号之间允许存在任意个数的空白字符，整行语句构成了一条预处理指令
- 该指令将在编译器进行编译之前对源代码做某些转换。

一个典型程序的转换处理过程

- **#** 空指令，无任何效果
- **#include** 包含一个源代码文件
- **#define** 定义宏
- **#undef** 取消已定义的宏
- **#if** 如果给定条件为真，则编译下面代码
- **#ifdef** 如果宏已经定义，则编译下面代码
- **#ifndef** 如果宏没有定义，则编译下面代码
- **#elif** 如果前面的#if给定条件不为真，当前条件为真，
则编译下面代码
- **#endif** 结束一个#if.....#else条件编译块

一个典型程序的转换处理过程

#gcc -E test.c -o test.i

预处理后的 文件为 test.i

```
#include <stdio.h>
#define myfunc(a,b) ((a)>(b)?(a):(b))
#define N 30
int main()
{
    int x=10;
    int y=20;
    int z=15;
    z=N + myfunc(x,y);
    printf("z=%d\n", z);
    return 0;
}
```

```
int main()
{
    int x=10;
    int y=20;
    int z=15;
    z=30 + ((x)>(y)?(x):(y));
    printf("z=%d\n", z);
    return 0;
}
```

提醒：定义宏函数时，注意在参数上加括号

一个典型程序的转换处理过程

提醒：定义宏函数时，注意在参数上加括号

```
#define square(x) x*x
```

```
int t;
```

```
t = square(1 + 2);
```

```
t = 1 + 2*1 + 2;
```

```
#define square(x) (x)*(x)
```

```
t = square(1 + 2);
```

```
t = (1 + 2)*(1 + 2);
```

一个典型程序的转换处理过程

条件编译

什么是条件编译? 为什么要写有条件编译的程序?

```
#include <stdio.h>
int main()
{
    #ifdef _FIRST
        printf("hello , First \n");
    #endif
    #ifdef _SECOND
        printf("good, Second\n");
    #endif
    printf("game over\n");
    return 0;
}
```

一个典型程序的转换处理过程

```
#gcc -E test.c -D _FIRST -o test.i
```

预处理后的 文件为 test.i

```
#include <stdio.h>
int main()
{
    #ifdef _FIRST
        printf("hello , First \n");
    #endif
    #ifdef _SECOND
        printf("good, Second\n");
    #endif
    printf("game over\n");
    return 0;
}
```

```
int main()
{
    printf("hello , First \n");

    printf("game over\n");
    return 0;
}
```

```
#gcc -E test.c -o test.i
```

```
int main()
{

    printf("game over\n");
    return 0;
}
```


一个典型程序的转换处理过程

```
#gcc -E -D _SECOND test.c -o test.i
```

预处理后的 文件为 test.i

```
#include <stdio.h>
int main()
{
    #ifdef _FIRST
        printf("hello , First \n");
    #endif
    #ifdef _SECOND
        printf("good, Second\n");
    #endif
    printf("game over\n");
    return 0;
}
```

```
int main()
{

    printf("good, Second\n");

    printf("game over\n");
    return 0;
}
```

一个典型程序的转换处理过程

```
#gcc -E -D _SECOND test.c -o test.i
```

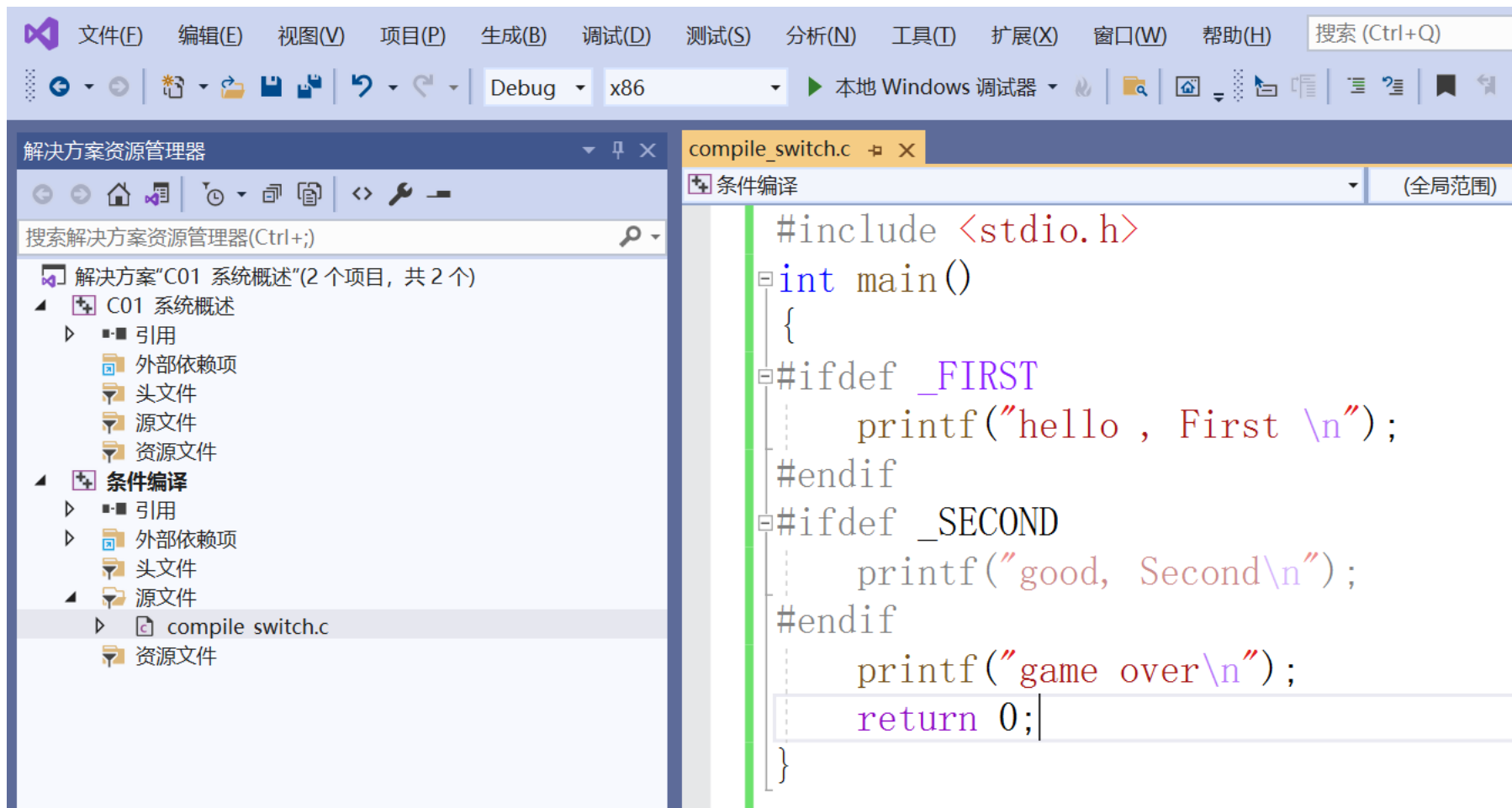
条件编译的选项也可以直接 写在程序中

```
#include <stdio.h>
#define _SECOND
int main()
{
    #ifdef _FIRST
        printf("hello , First \n");
    #endif
    #ifdef _SECOND
        printf("good, Second\n");
    #endif
    printf("game over\n");
    return 0;
}
```

```
#gcc -E test.c -o test.i
```

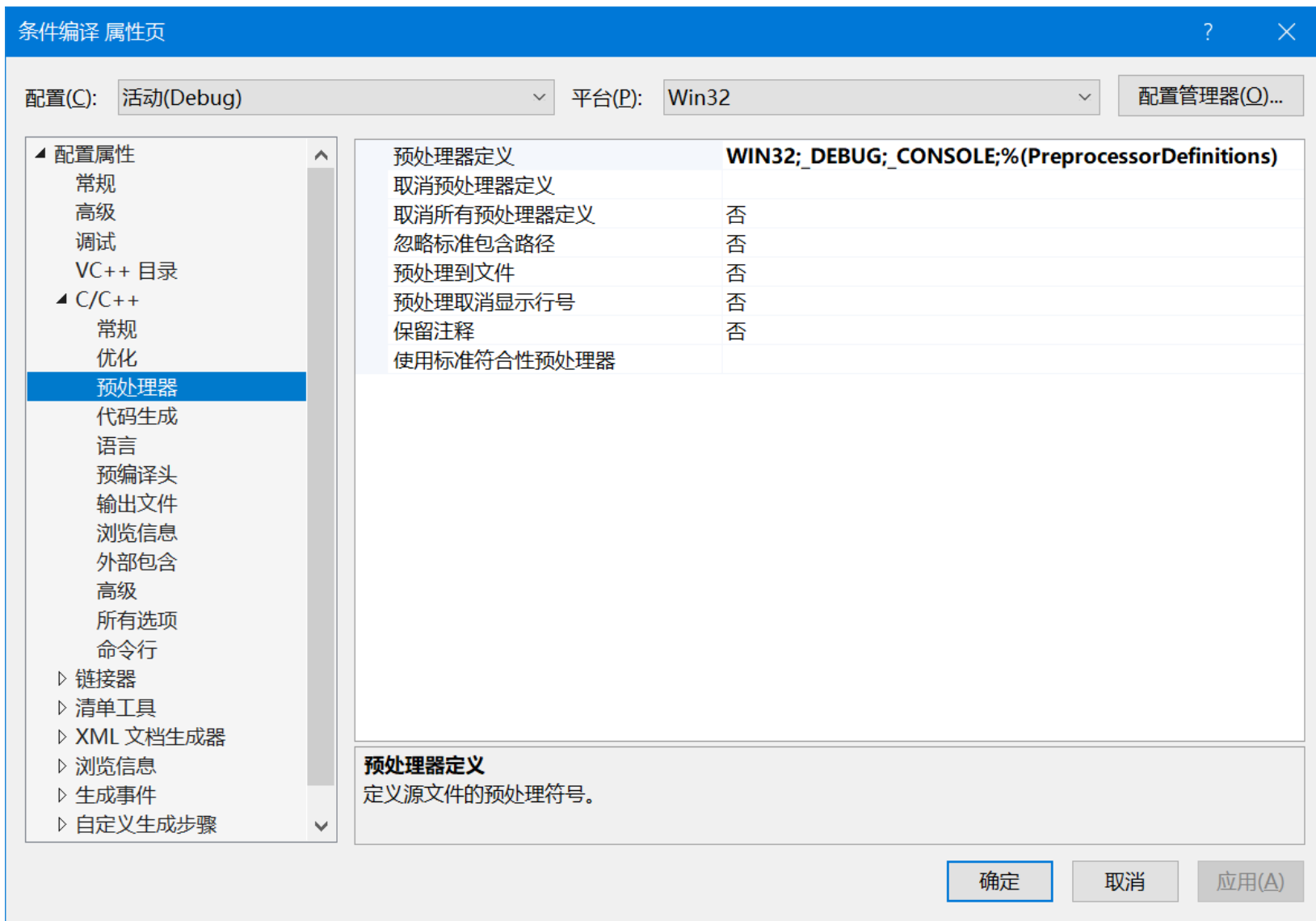
一个典型程序的转换处理过程

VS2019 下 C程序编译 预处理



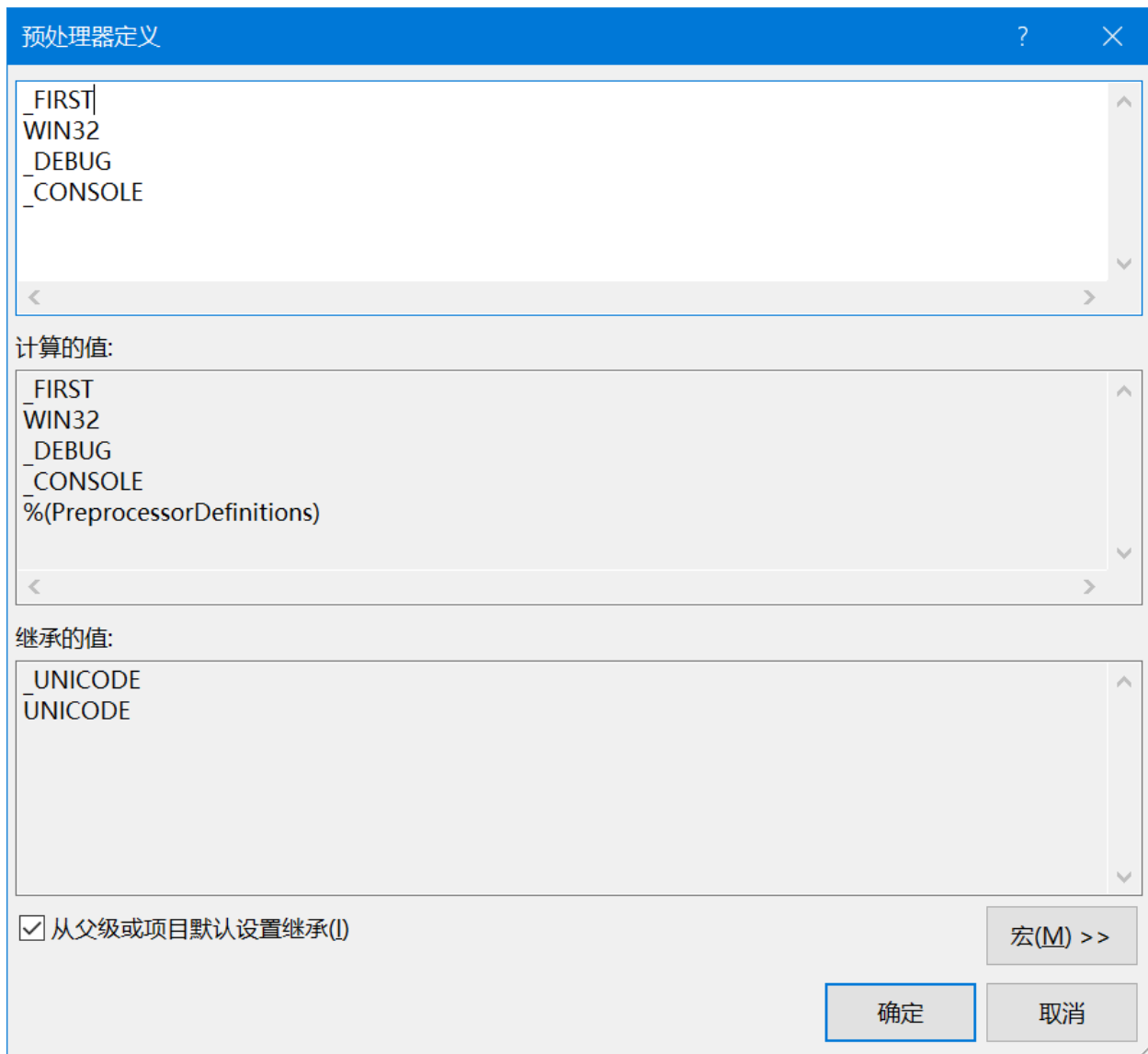
一个典型程序的转换处理过程

右键单击 项目名称，在弹出的菜单上，单击“属性”



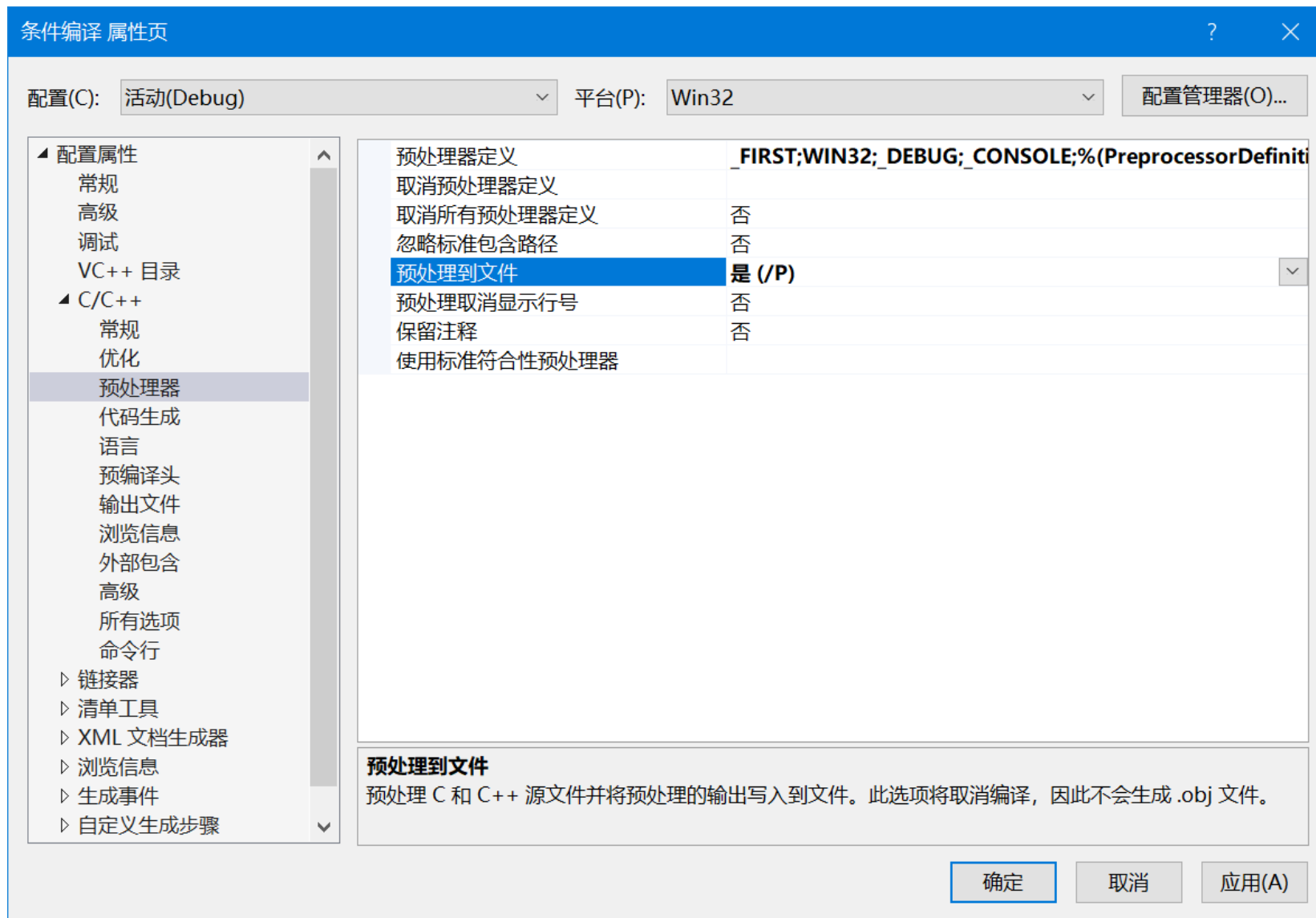
一个典型程序的转换处理过程

VS2019 下 C程序编译 预处理：在预处理器定义后，可加上宏名



一个典型程序的转换处理过程

VS2019 下 预处理到文件，生成 .i 文件



一个典型程序的转换处理过程

预处理到文件, 生成 .i 文件

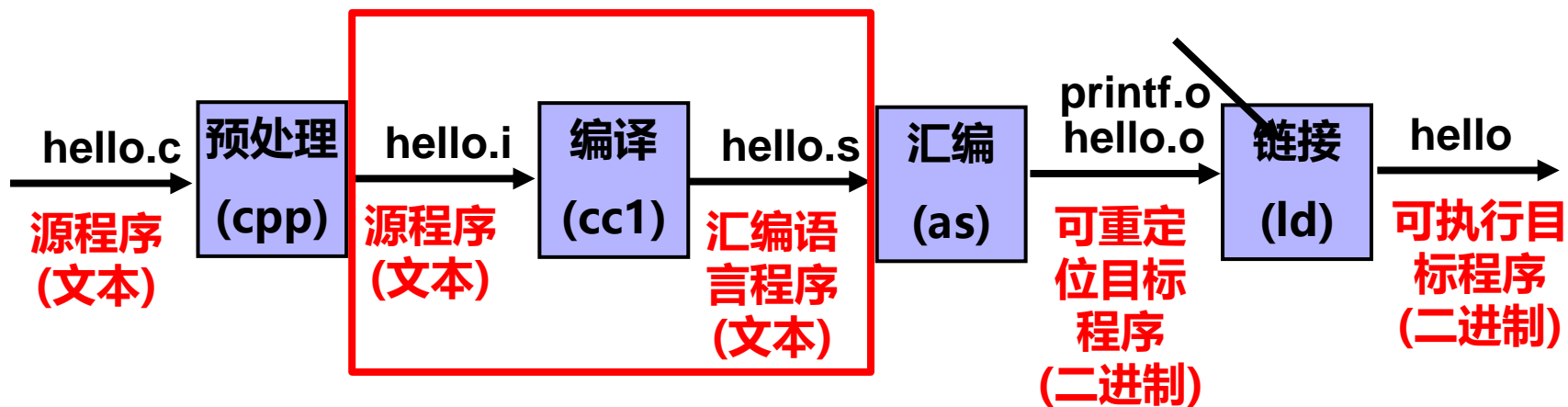
```
#line 2 "C:\\教学\\本科教学\\计算机系统基础\\计算机系统基础_程序
\\C01_系统概述\\条件编译\\compile_switch.c"
int main()
{
    printf("hello , First \n");
#line 7 "C:\\教学\\本科教学\\计算机系统基础\\计算机系统基础_程序
\\C01_系统概述\\条件编译\\compile_switch.c"

    printf("game over\n");
    return 0;
}
```

一个典型程序的转换处理过程

➤ 编译 生成汇编语言程序

```
#gcc -S -D _FIRST test.c -o test.s
```



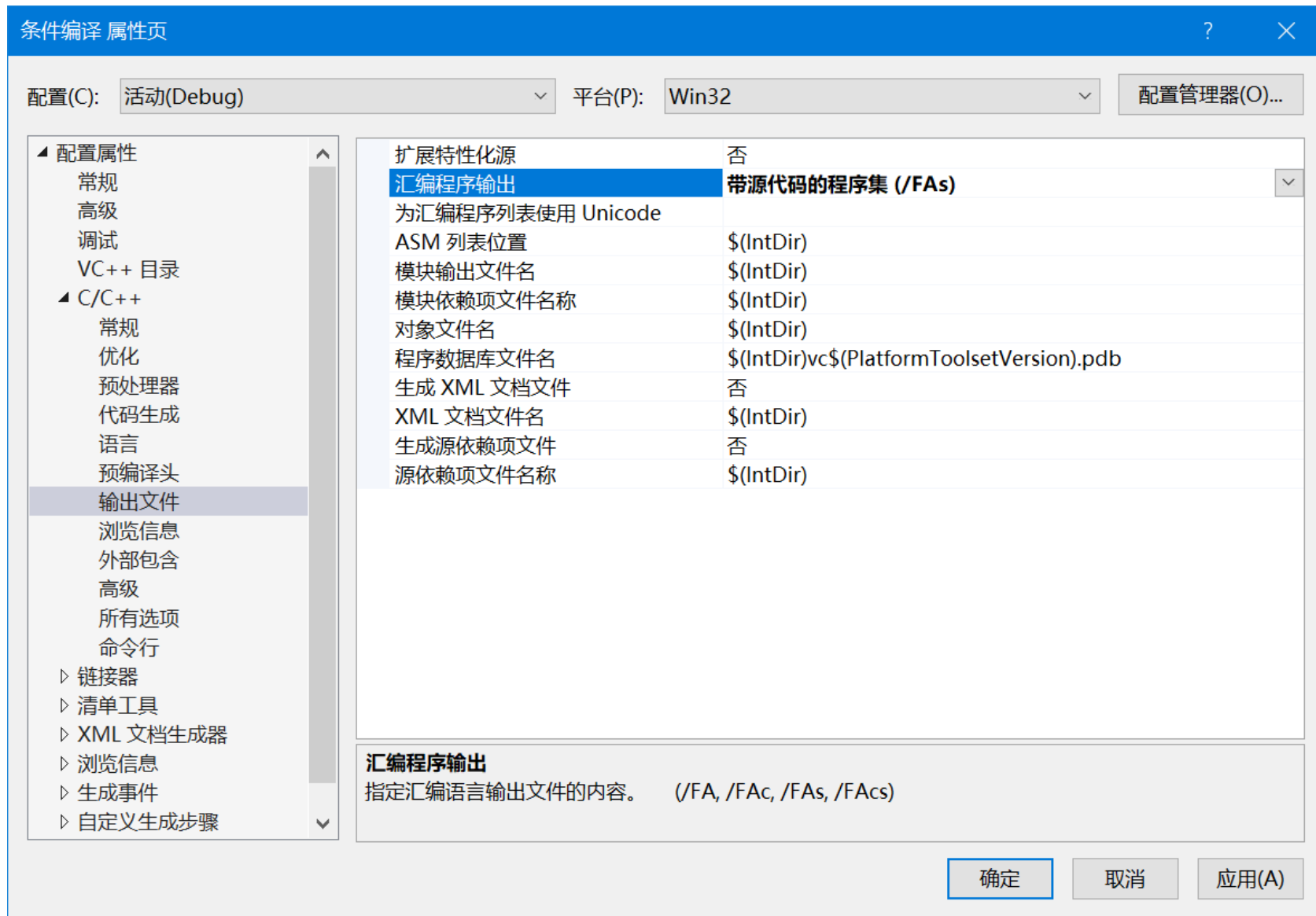
一个典型程序的转换处理过程

```
.file "test.c"
.text
.section      .rodata
.LC0:
.string "hello , First "
.LC1:
.string "game over"
.text
.globl main
.type  main, @function
```

```
main:
.LFB0:
.cfi_startproc
pushq  %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq   %rsp, %rbp
.cfi_def_cfa_register 6
leaq   .LC0(%rip), %rdi
call   puts@PLT
leaq   .LC1(%rip), %rdi
call   puts@PLT
movl   $0, %eax
popq   %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size  main, .-main
.ident "GCC: (Ubuntu ~18.04) 7.5.0"
.section      .note.GNU-
stack,"",@progbits
```

一个典型程序的转换处理过程

➤ VS2019 生成汇编语言程序 *.asm 文本文件



一个典型程序的转换处理过程

```
_main PROC ; COMDAT
; 3 : {
    push    ebp
    mov     ebp, esp
    sub     esp, 192 ; 000000c0H
    push    ebx
    push    esi
    push    edi
    mov     edi, ebp
    xor     ecx, ecx
    mov     eax, -858993460 ; ccccccccH
    rep stosd
    mov     ecx, OFFSET __BE3C319C_compile_switch@c
    call    @__CheckForDebuggerJustMyCode@4
; 4 : #ifdef _FIRST
; 5 :     printf("hello , First \n");

    push    OFFSET ??_C@_0BA@DOLPFANA@hello?5?0?5First?5?6@
    call    _printf
    add     esp, 4
; 6 : #endif
```

一个典型程序的转换处理过程

#objdump -d -M att test.o 反汇编

#objdump -d test.o 默认用 AT&T 格式显示指令

test.o: file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:

```
0: 55                push  %rbp
1: 48 89 e5          mov   %rsp,%rbp
4: 48 8d 3d 00 00 00 00 lea   0x0(%rip),%rdi    # b <main+0xb>
b: e8 00 00 00 00    callq 10 <main+0x10>
10: 48 8d 3d 00 00 00 00 lea   0x0(%rip),%rdi    # 17 <main+0x17>
17: e8 00 00 00 00    callq 1c <main+0x1c>
1c: b8 00 00 00 00    mov   $0x0,%eax
21: 5d                pop   %rbp
22: c3                retq
```

一个典型程序的转换处理过程

#objdump -d -M intel test.o 反汇编

test.o: file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:

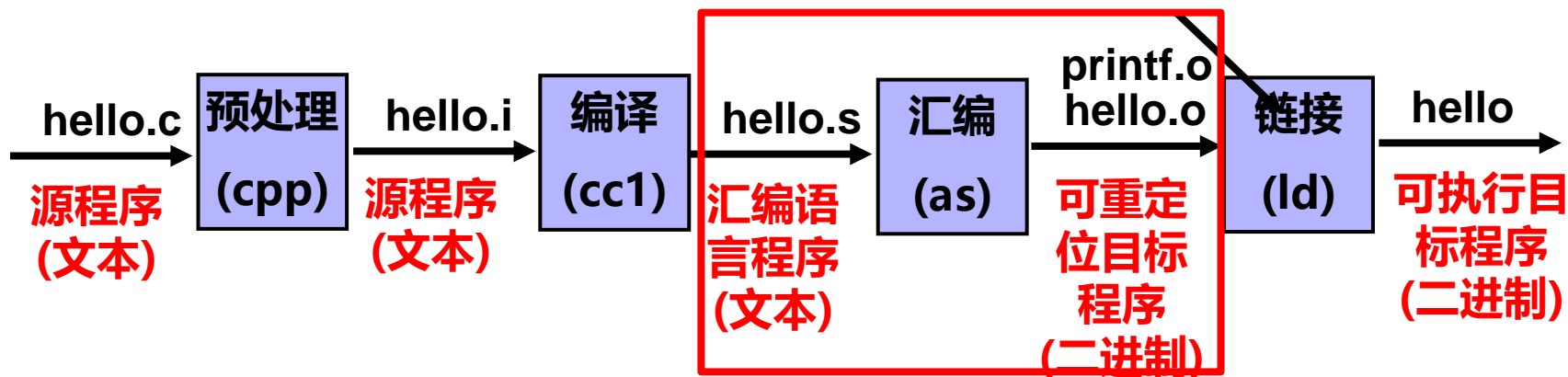
0:	55	push	rbp	
1:	48 89 e5	mov	rbp,rsi	
4:	48 8d 3d 00 00 00 00	lea	rdi,[rip+0x0]	# b <main+0xb>
b:	e8 00 00 00 00	call	10 <main+0x10>	
10:	48 8d 3d 00 00 00 00	lea	rdi,[rip+0x0]	# 17 <main+0x17>
17:	e8 00 00 00 00	call	1c <main+0x1c>	
1c:	b8 00 00 00 00	mov	eax,0x0	
21:	5d	pop	rbp	
22:	c3		ret	

一个典型程序的转换处理过程

➤ 汇编

```
#gcc -c -D _FIRST test.c -o test.o
```

```
#objdump -d test.o 反汇编
```

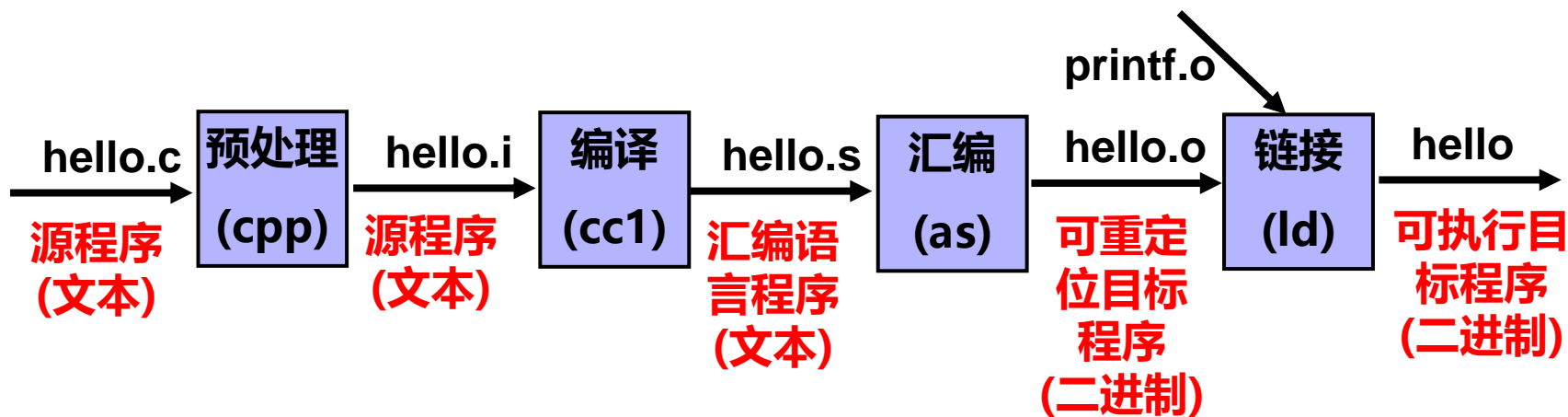


一个典型程序的转换处理过程

➤ 编译链接，生成执行程序

➤ `#gcc -D _FIRST test.c -o test`

`#objdump -d test` 反汇编执行程序



软件的层次

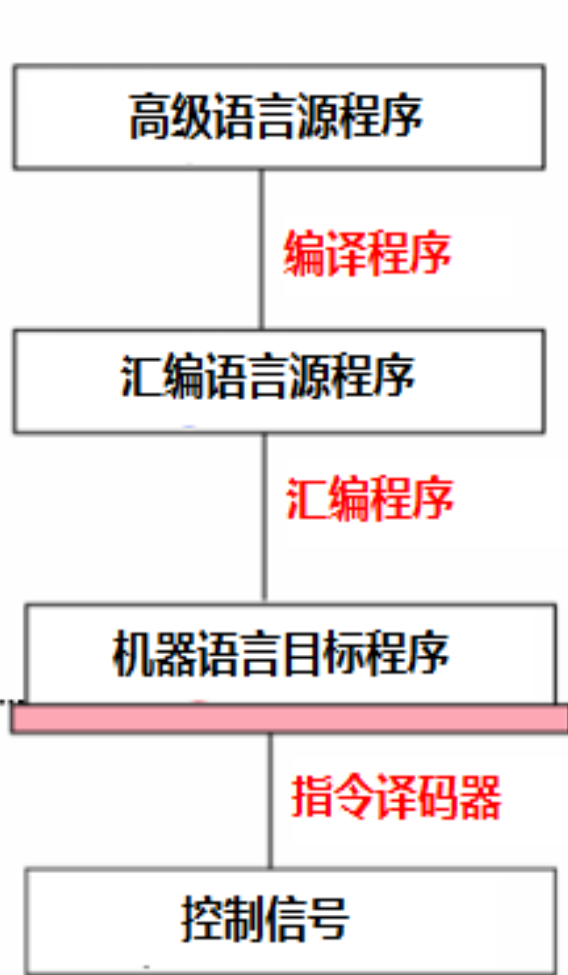
- **System software(系统软件)** - 简化编程过程, 并使系统资源被有效利用
 - 操作系统 (Operating System) : 用户接口, 资源管理, ...
 - 语言处理系统: 翻译程序+ Linker, Debug, etc ...
 - 翻译程序(Translator)有三类:
 - 汇编程序(Assembler):** 汇编语言源程序→机器语言目标程序
 - 编译程序(Compiler):** 高级语言源程序→机器级目标程序
 - 解释程序(Interpreter):** 将高级语言语句逐条翻译成机器指令并立即执行,不生成目标文件。
 - 其他实用程序: 如: 磁盘碎片整理程序、备份程序等
 - **Application software(应用软件)** - 解决具体应用问题/完成具体应用任务
 - 各类媒体处理程序: Word/ Image/ Graphics/...
 - 管理信息系统 (MIS)
 - Game, ...

1.3 计算机系统层次结构

不同层次语言之间的等价转换

计算机软件

计算机硬件



```
tmp = a[i];  
a[i] = a[i+1];  
a[i+1] = tmp;
```

```
lw $8, 0($2)  
lw $9, 4($2)  
sw $9, 0($2)  
sw $8, 4($2)
```

每条指令由操作码和若干地址码组成

100011	00010	01000	0000000000000000
100011	00010	01001	0000000000000100
101011	00010	01001	0000000000000000
101011	00010	01000	0000000000000100

... , EXTop=1,ALUSelA=1,ALUSelB=11,ALUOp=add,
IorD=1,Read,MemtoReg=1,RegWr=1, ...

任何高级语言程序最终通过执行若干条指令来完成!

开发和运行程序需要的支撑

- 最早的程序开发很简单

直接输入指令和数据，启动后把第一条指令地址送PC开始执行

- 用高级语言开发程序需要复杂的支撑环境

需要编辑器编写源程序

- 需要一套翻译转换软件处理各类源程序

- 编译方式：预处理程序、编译器、汇编器、链接器
- 解释方式：解释程序

语言
处理
程序

语言处理系统 +

- 需要一个可以执行程序的界面（环境）

- GUI方式：图形用户界面
- CUI方式：命令行用户界面

人机
接口

操作系统

语言的运行时系统

操作系统内核

指令集体系结构

计算机硬件

支撑程序开发和运行的环境由系统软件提供

最重要的系统软件是操作系统和语言处理系统

语言处理系统运行在操作系统之上，操作系统利用指令管理硬件

早期计算机系统的层次

- 最早的计算机用机器语言编程

机器语言称为第一代程序设计语言（
First generation programming language , 1GL ）

应用程序

指令集体系结构

计算机硬件

- 用汇编语言编程

汇编语言称为第二代程序设计语言（
Second generation programming language , 2GL ）

应用程序

汇编程序

操作系统

指令集体系结构

计算机硬件

现代（传统）计算机系统的层次

- 现代计算机用高级语言编程

第三代程序设计语言（3GL）为过程式语言，编码时需要描述实现过程，即“如何做”。

第四代程序设计语言（4GL）为非过程化语言，编码时只需说明“做什么”，不需要描述具体的算法实现细节。

可以看出：语言的发展是一个不断“抽象”的过程，因而，相应的计算机系统也不断有新的层次出现

应用程序

语言处理系统

操作系统

指令集体系结构

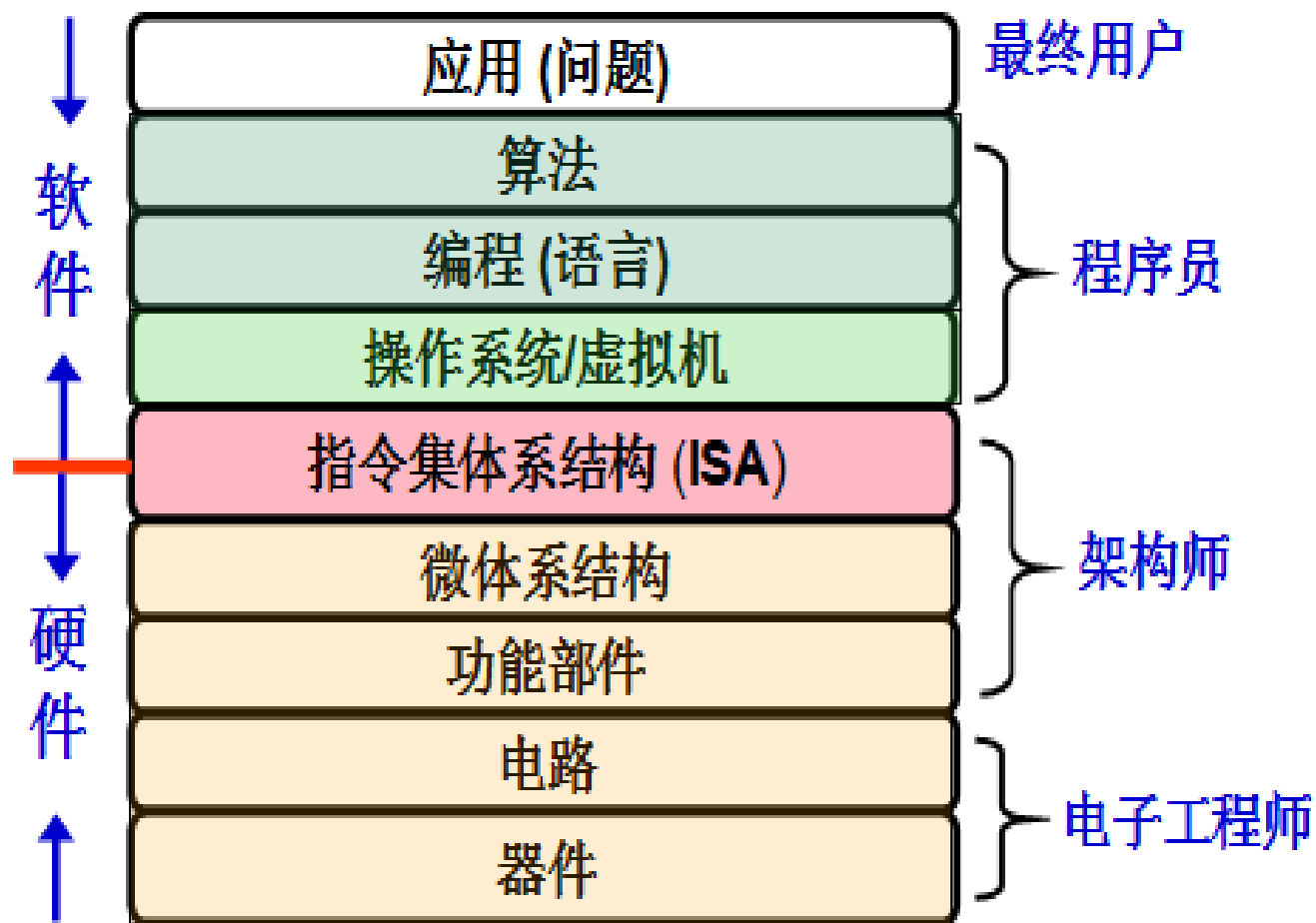
计算机硬件

语言处理系统包括：各种语言处理程序（如编译、汇编、链接）、运行时系统（如库函数、调试、优化等功能）

操作系统包括人机交互界面、提供服务功能的内核例程

计算机系统抽象层的转换

功能转换：上层是下层的**抽象**，下层是上层的**实现**
底层为上层提供支撑环境！



必须将各层次关联起来解决问题

计算机系统的不同用户

最终用户工作在由应用程序提供的最上面的抽象层

系统管理员工作在由操作系统提供的抽象层

应用程序员工作在由语言处理系统（**主要有编译器和汇编器**）的抽象层

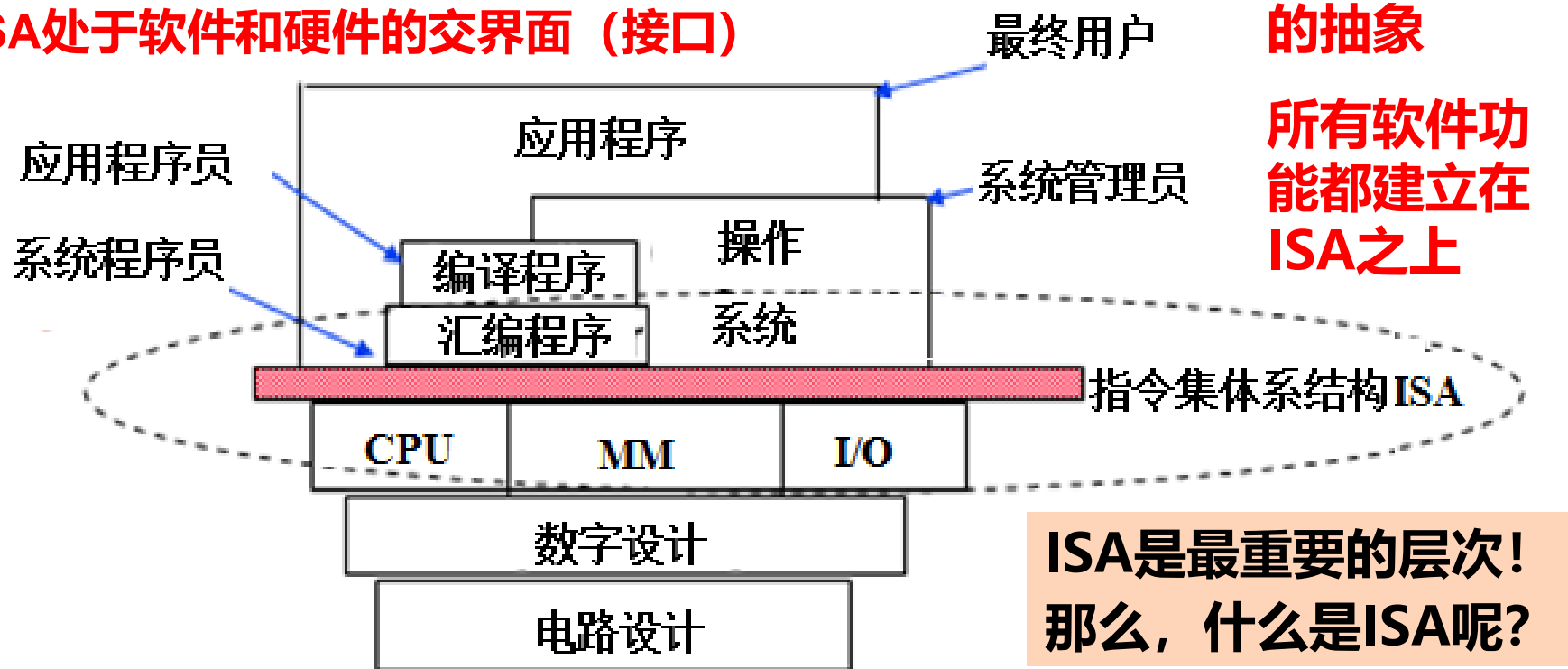
语言处理系统建立在**操作系统**之上

系统程序员（实现系统软件）工作在ISA层次，必须对ISA非常了解

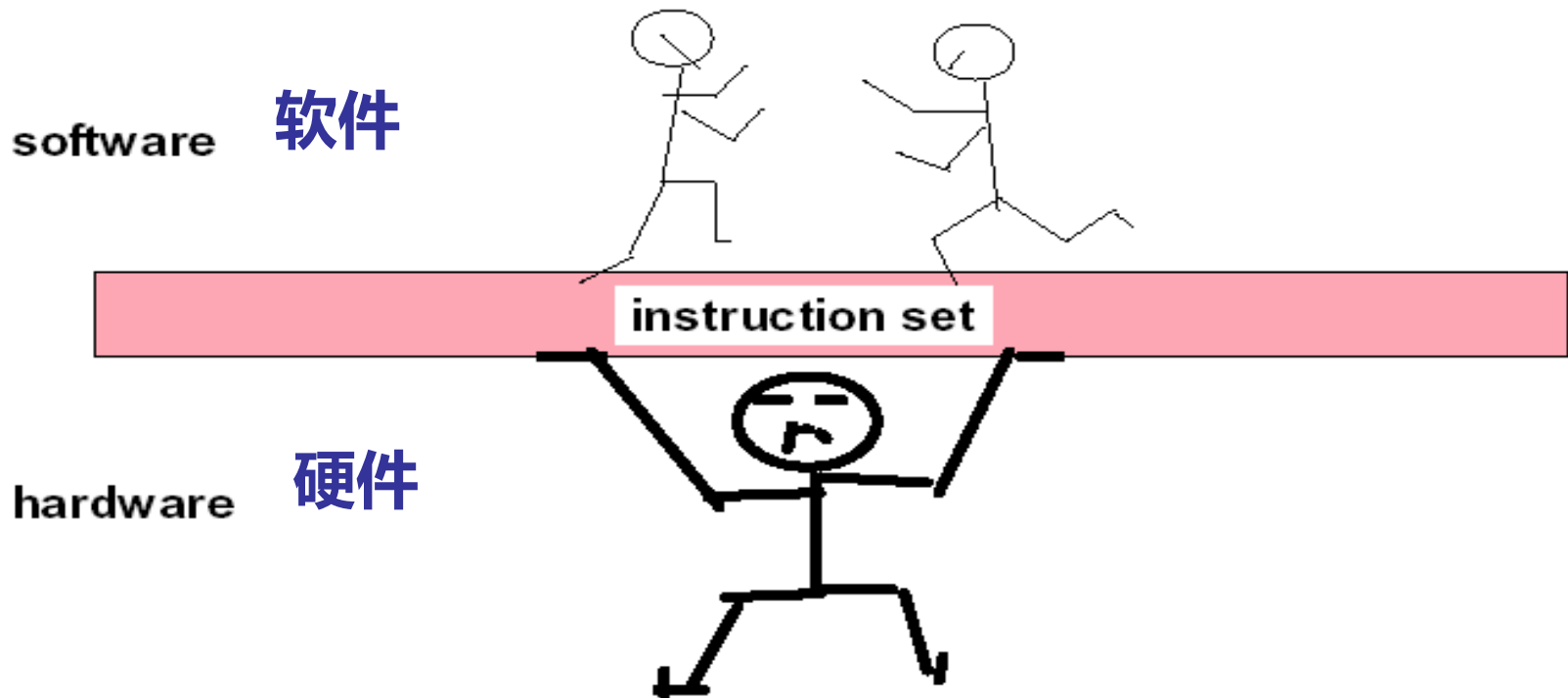
编译器和汇编器的目标程序由机器级代码组成

操作系统通过指令直接对硬件进行编程控制

ISA处于软件和硬件的交界面（接口）



Hardware/Software Interface (界面)



软件和硬件的界面：ISA (Instruction Set Architecture)
指令集体系结构

机器语言由指令代码构成，能被硬件直接执行。

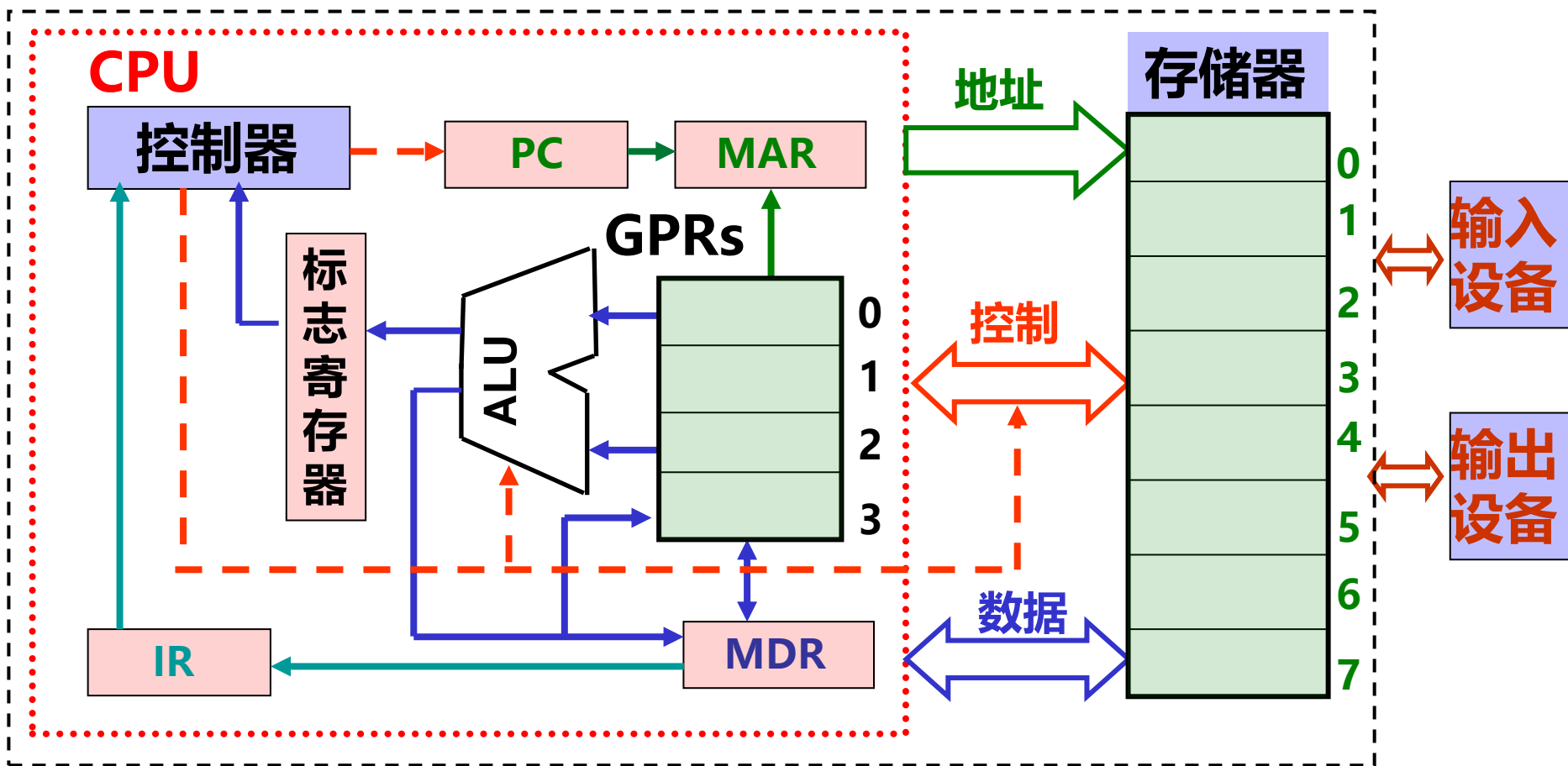
指令集体系结构（ISA）

- ISA指Instruction Set Architecture，即指令集体系结构
- ISA是一种规约（Specification），它规定了**如何使用硬件**
 - 可执行的指令的集合，包括**指令格式、操作种类以及每种操作对应的操作数的相应规定**；
 - 指令可以接受的**操作数的类型**；
 - 操作数所能存放的寄存器组的结构，包括每个**寄存器的名称、编号、长度和用途**；
 - 操作数所能存放的**存储空间的大小和编址方式**；
 - 操作数在存储空间存放时按照**大端还是小端方式存放**；
 - 指令获取操作数的方式，即**寻址方式**；
 - 指令执行过程的控制方式，包括**程序计数器、条件码定义等**。
- ISA在计算机系统中是必不可少的一个抽象层，Why？
 - 没有它，软件无法使用计算机硬件！
 - 没有它，一台计算机不能称为“通用计算机”

微体系结构

ISA和计算机组成（Organization，即MicroArchitecture）是何关系？

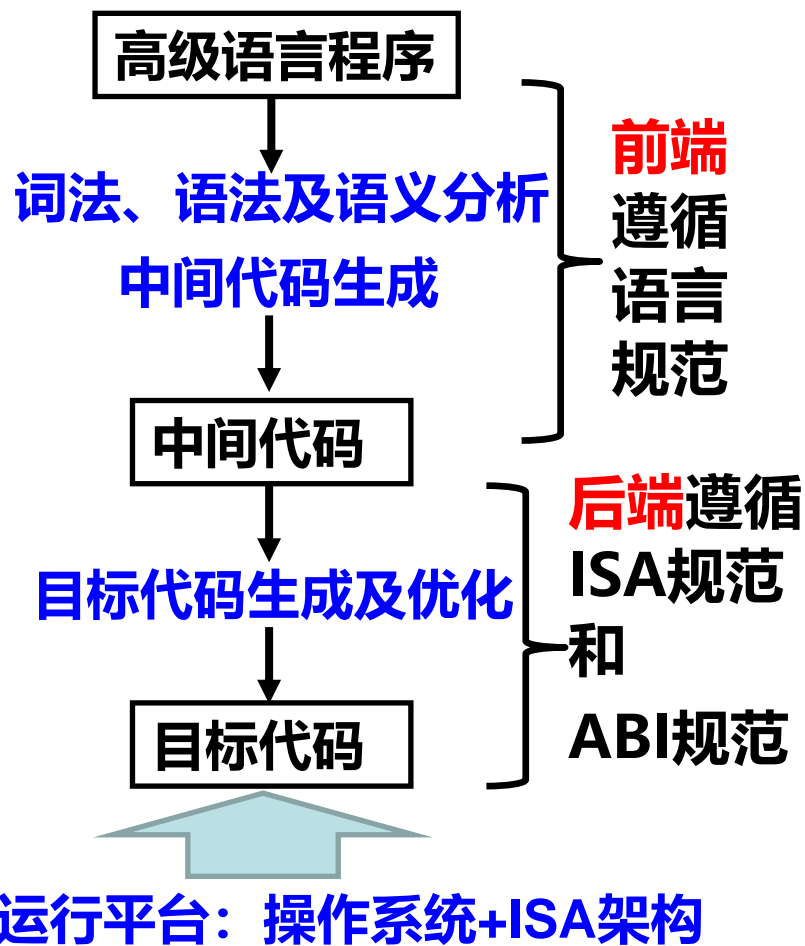
ISA和计算机组成（微结构）之间的关系



不同ISA规定的指令集不同，如，IA-32、MIPS、ARM等
计算机组成必须能够实现ISA规定的功能，如提供GPR、标志、运算电路等
同一种ISA可以有不同的计算机组成，如乘法指令可用ALU或乘法器实现

ISA是计算机
组成的抽象

计算机系统核心层之间的关联



执行结果不符合程序开发者预期举例：

C90中， $-2147483648 < 2147483647$

结果为false

```
int x=1234;  
printf( "%lf" ,x);
```

} 未定义行为

不同平台结果不同，相同平台每次结果不同

结果不符合预期的原因通常有两种：

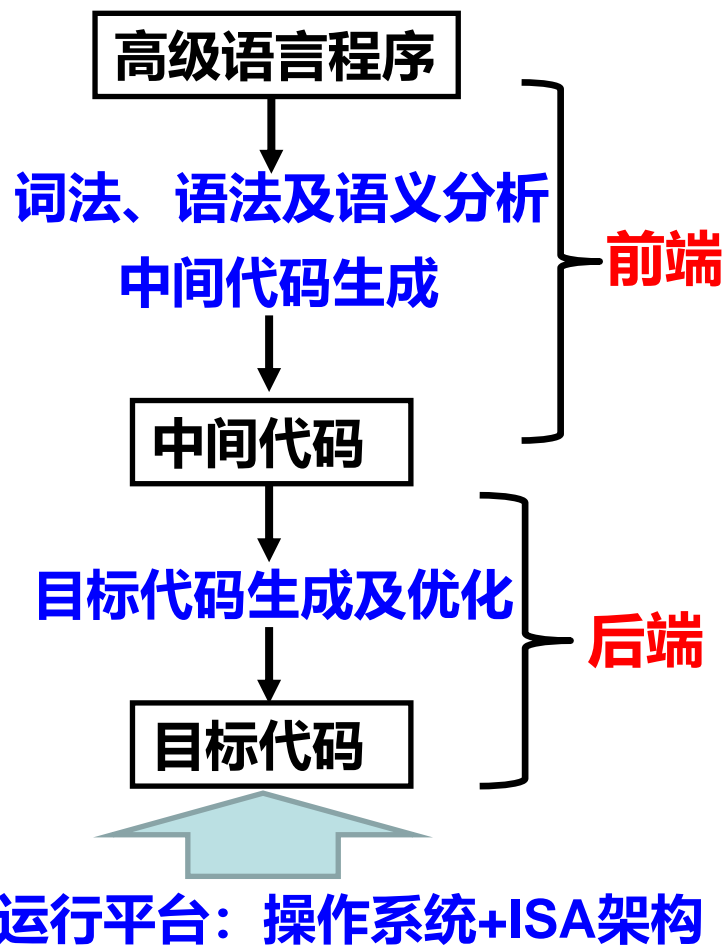
- (1) 程序员不了解语言规范；
- (2) 程序含有未定义行为（undefined behavior）或未确定行为（unspecified behavior）的语句

ABI是为运行在特定ISA及特定操作系统之上的应用程序中所遵循的一种机器级目标代码层接口

描述了应用程序和操作系统之间、应用程序和所调用的库之间、不同组成部分（如过程或函数）之间在较低层次上的机器级代码接口。

后端根据ISA规范和应用程序二进制接口（Application Binary Interface, ABI）规范进行设计实现。

计算机系统核心层之间的关联



ABI是运行在**特定ISA及特定操作系统之上**的应用程序所遵循的一种机器级目标代码层**接口规约**。例如：

过程间调用约定（参数和返回值传递等）

系统调用约定（系统调用的参数和调用号如何传递以及如何从用户态陷入操作系统内核等）

目标文件的二进制格式

函数库使用约定

寄存器使用规定

程序的虚拟地址空间划分

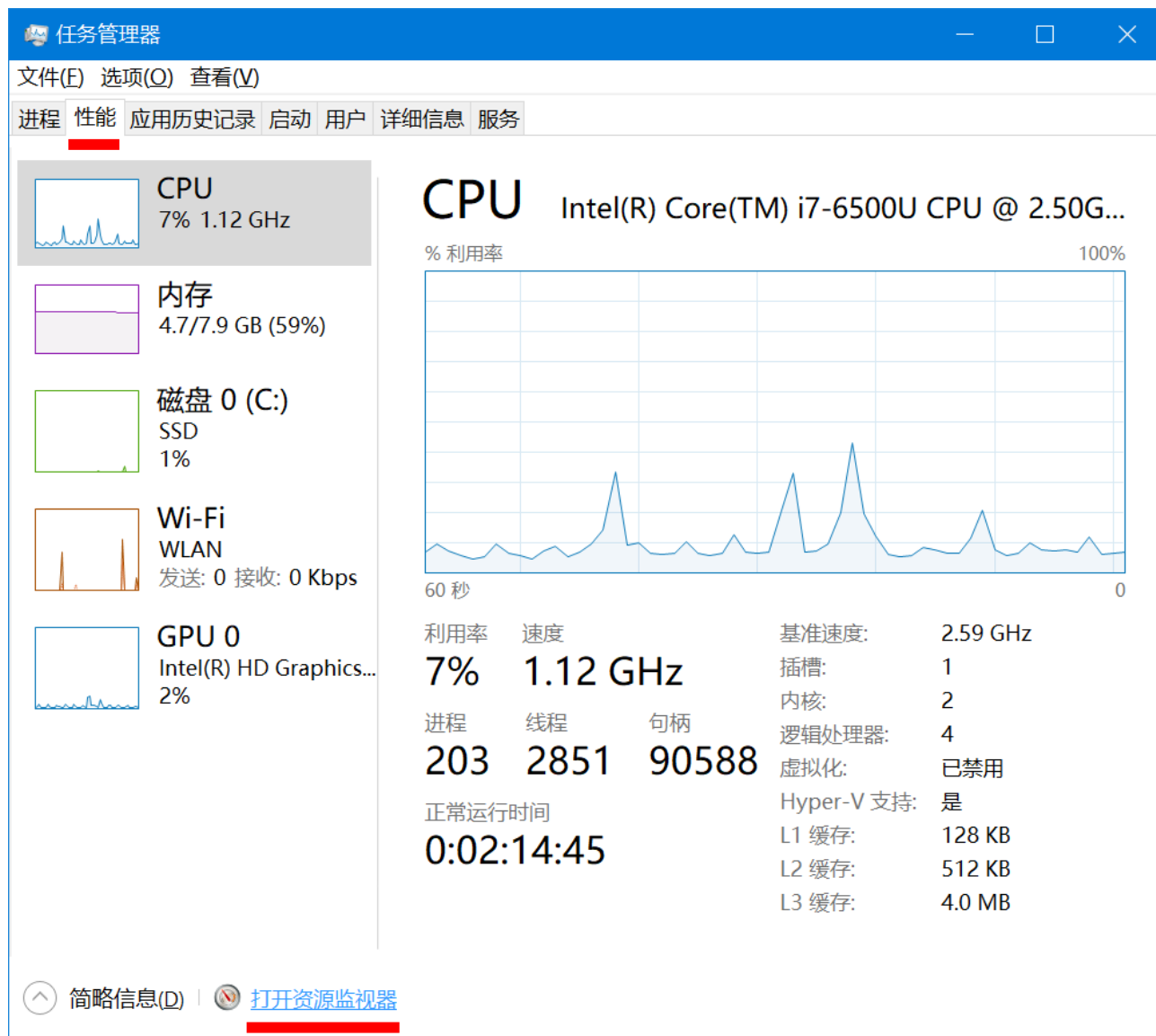
.....

平台：IA-32/x86-64
+Linux+GCC+C语言
Linux操作系统下一般使用
system V ABI

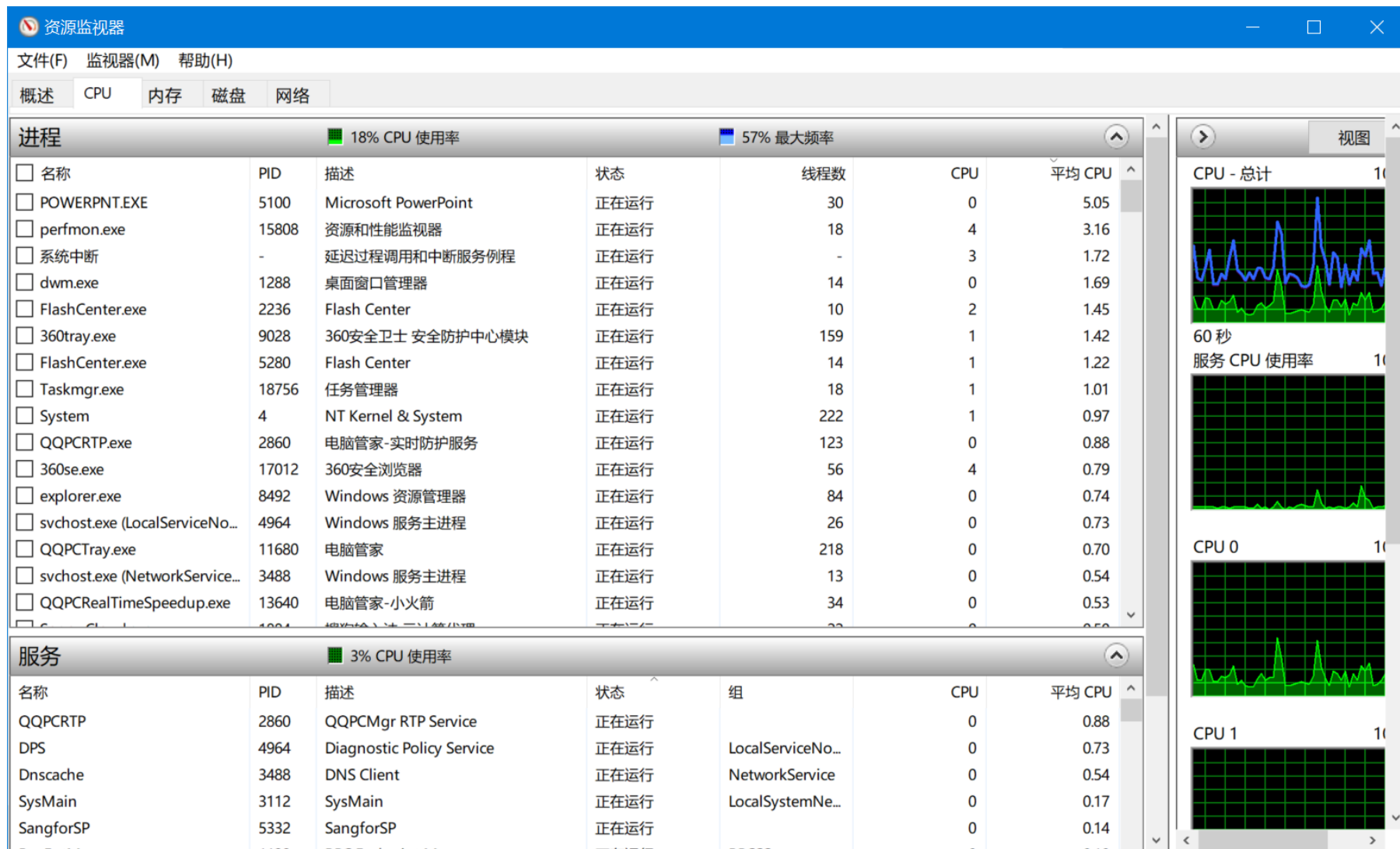
后端根据ISA规范和**应用程序二进制接口（Application Binary Interface, ABI）**规范进行设计实现。

1.4 计算机性能评价

1.4 计算机性能评价



1.4 计算机性能评价



1.4 计算机性能评价

一个程序的**响应时间**：从程序提交 到程序完成所用的时间

- 程序的**执行时间**
- 程序的**等待时间**

一个程序的**执行时间**：

- 程序包含的指令在CPU上执行所用的时间
- 磁盘访问时间
- 存储器访问时间
- I/O操作时间
- 操作系统运行这个程序所用的额外开销

1.4 计算机性能评价

- **用户程序的执行时间通常分为两部分：**
 - **CPU时间**：指CPU用于本程序执行的时间，它又包含以下两部分：
 - 用户CPU时间，指真正用于运行用户程序代码的时间；
 - 系统CPU时间，指为了执行用户程序而需要CPU运行操作系统程序的时间，例如：调用read和write内核方法时，消耗的时间就计入系统CPU时间。
 - **其他时间**：指等待I/O操作完成的时间、CPU用于执行其他用户程序的时间。
- **用执行时间评价的是CPU性能**

1.4 计算机性能评价

```
root@LAPTOP-CJLSTBTI:/home# ./array_subtract
please input : 0,1,2 :0
real :      6.25
user :      0.21
sys :      0.07
```

real : 指实际时间

user : 指用户CPU时间 (user CPU time)

sys : 系统CPU时间 (system CPU time)

在程序中，有一个scanf 语句，故意不及时输入，可看到real 时间增长

```
root@LAPTOP-CJLSTBTI:/home# ./array_subtract
please input : 0,1,2 :0
real :     13.25
user :      0.25
sys :      0.03
```

1.4 计算机性能评价

在linux下，可以使用`time`命令来查看程序执行时这三种时间值的消耗。最后三行为 `time` 的输出结果

```
root@LAPTOP-CJLSTBTI:/home# time ./array_subtract
please input : 0,1,2 :0
real :    19.31
user :    0.12
sys :    0.10

real    0m19.376s
user    0m0.125s
sys     0m0.141s
root@LAPTOP-CJLSTBTI:/home#
```

1.4 计算机性能评价

在linux下, 程序计时程序段

```
#include <stdio.h>
#include <sys/times.h>
#include <unistd.h>
int clktck=sysconf(_SC_CLK_TCK);
struct tms tmsstart, tmsfinish;
clock_t start, finish;
```

```
start = times(&tmsstart);
```

```
.....
```

```
finish = times(&tmsfinish);
```

```
printf("real : %7.2f\n", (finish-start)/(double)clktck);
```

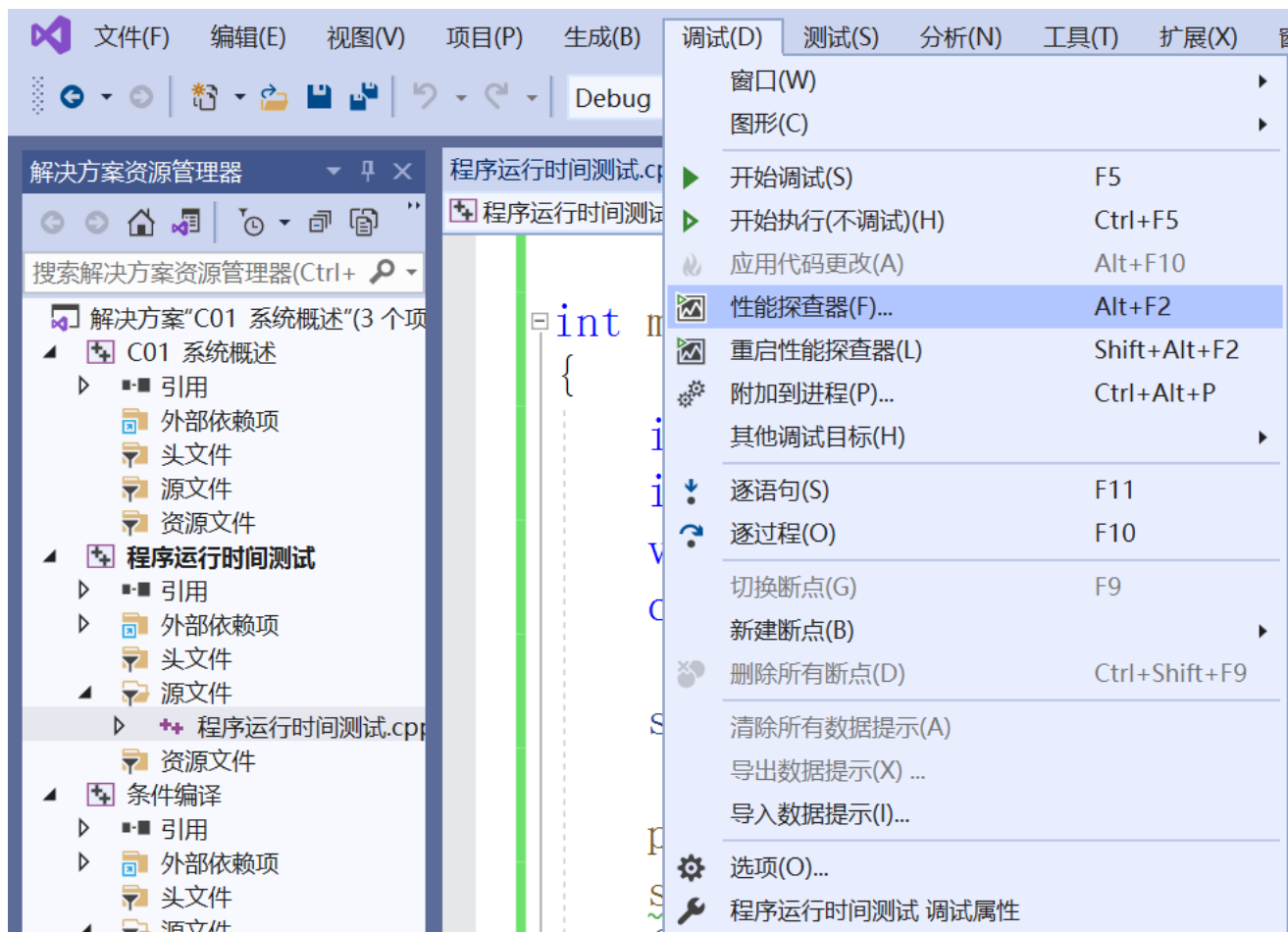
```
printf("user : %7.2f\n", (tmsfinish.tms_ftime - tmsstart.tms_ftime)/(double)clktck);
```

```
printf(" sys : %7.2f\n", (tmsfinish.tms_stime - tmsstart.tms_stime)/(double)clktck);
```

```
struct tms {
    clock_t tms_ftime;
    /* user time */
    clock_t tms_stime;
    /* system time */
    clock_t tms_cutime;
    /*user time of children*/
    clock_t tms_cstime;
    /*system time of children*/
};
```

1.4 计算机性能评价

在VS 2019 中，对于一个程序的性能进行测试



1.4 计算机性能评价

在VS 2019 中，对于一个程序的性能进行测试

报告20220316-0920.diagsession × 程序运行时间测试.cpp

>



更改目标 ▼

启动项目

程序运行时间测试

⚠ 解决方案配置设置为“调试”。请切换到“发布”配置以获取更准确的结果。

可用工具

☒ CPU 使用率 ⚙️ ?
查看 CPU 执行代码时的时间耗费情况。当 CPU 遇到性能瓶颈时很有用

☐ GPU 使用情况 ⚙️
检查 DirectX 应用程序中的 GPU 使用情况。这有助于确定性能瓶颈是 CPU 还是 GPU

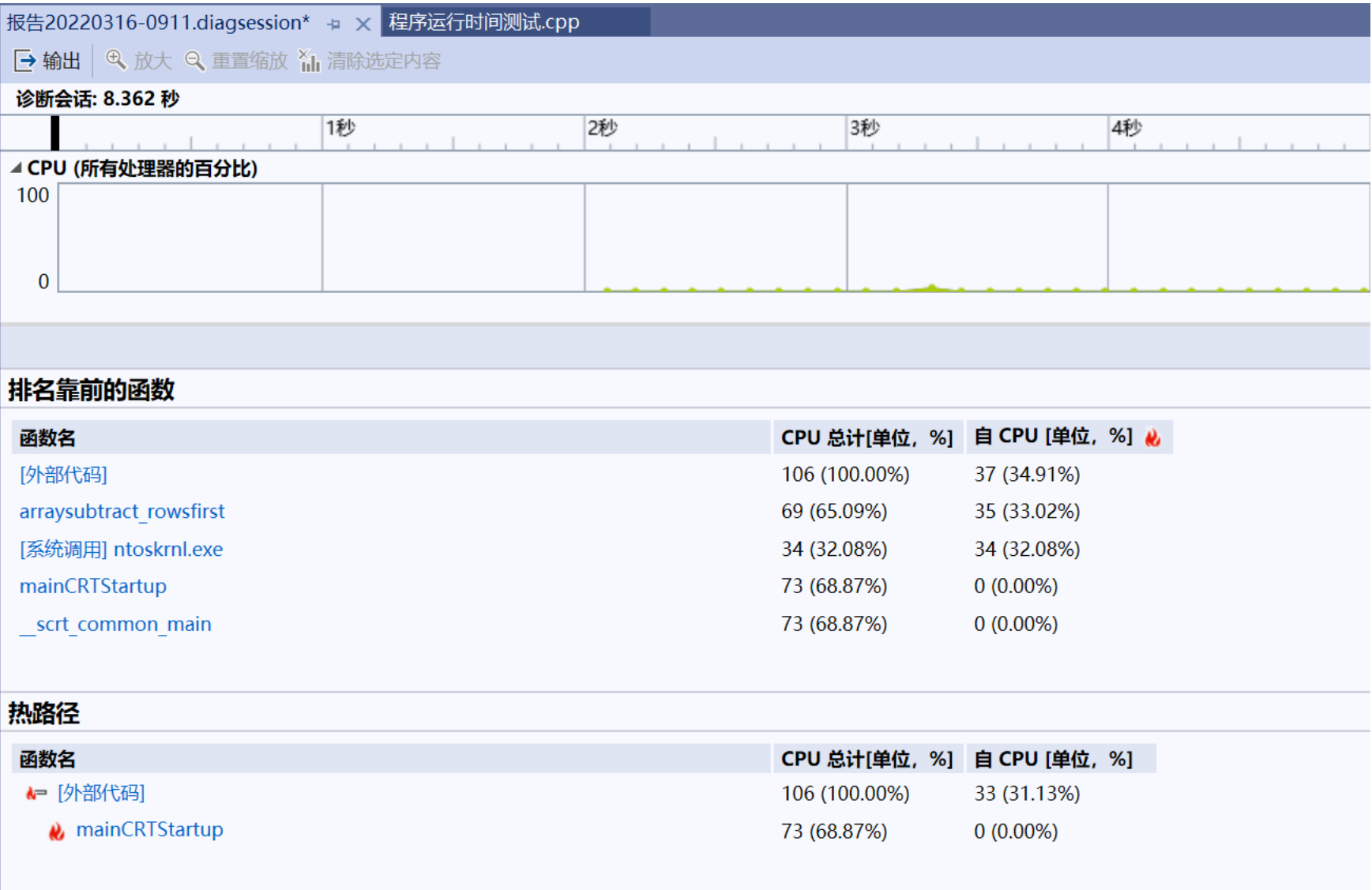
☐ 检测
检测应用程序以调查确切的调用数和调用时间

☐ 内存使用率
检查应用程序内存以查找内存泄漏等问题

☐ 事件查看器 ⚙️ ?
查看会话期间发生的事件(ETW 或 NetTrace)，例如日志消息、异常和 HTTP 请求

开始

1.4 计算机性能评价



计算机性能的基本评价指标

- 计算机有两种不同的性能

- Time to do the task

- 响应时间 (response time)
 - 执行时间 (execution time)
 - 等待时间或时延 (latency)

- Tasks per day, hour, sec, ns. ..

- 吞吐率 (throughput)
 - 带宽 (bandwidth)

- 基本的性能评价标准是：CPU的执行时间

不同应用场合用户关心的性能不同：

-要求吞吐率高的场合，例如：

多媒体应用（音/视频播放要流畅）

-要求响应时间短的场合：例如：

事务处理系统（存/取款速度要快）

-要求吞吐率高且响应时间短的场合：

ATM、文件服务器、Web服务器等

“机器X的速度（性能）是Y的n倍” 的含义：

$$\frac{\text{ExTime}(Y)}{\text{ExTime}(X)} = \frac{\text{Performance}(X)}{\text{Performance}(Y)} = n$$

相对性能用执行时间的倒数来表示！

CPU执行时间的计算

CPU时钟周期：时钟发生器发出的脉冲信号，周期性的变化
一个节拍脉冲的时间长度

CPU时钟频率：1秒钟的时钟周期数， 周期的倒数

1K Hz: 千赫兹 1000Hz

1MHz: 兆赫兹 (百万) 1000 KHz

1GHz: 吉赫兹 (十亿) 1000M Hz

设备规格

ThinkPad X1 Carbon 4th Signature Edition

设备名称

LAPTOP-CJLSTBTI

处理器

Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz
2.59 GHz

机带 RAM

8.00 GB (7.89 GB 可用)



CPU执行时间的计算

CPI: Cycles Per Instruction

执行一条指令所需的时钟周期数

$$\begin{aligned}\text{CPU 执行时间} &= \text{CPU时钟周期数} / \text{程序} \times \text{时钟周期} \\ &= \text{CPU时钟周期数} / \text{程序} \div \text{时钟频率} \\ &= \text{指令条数} / \text{程序} \times \text{CPI} \times \text{时钟周期}\end{aligned}$$

$$\text{CPU时钟周期数} / \text{程序} = \text{指令条数} / \text{程序} \times \text{CPI}$$

$$\text{CPI} = \text{CPU时钟周期数} / \text{程序} \div \text{指令条数} / \text{程序}$$

CPI 用来衡量以下各方面的综合结果

- Instruction Set Architecture (ISA)
- Implementation of that architecture
(Organization & Technology)
- Program (Compiler、Algorithm)

Aspects of CPU Performance

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	instr. Count	CPI	clock rate
Programming			
Compiler			
Instr. Set Arch.			
Organization			
Technology			

思考：三个因素与哪些方面有关？

例如，{.....
 $y=4*x;$
 }

Aspects of CPU Performance

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	instr. Count	CPI	clock rate
Programming	X	X	
Compiler	X	(X)	
Instr. Set Arch.	X	X	
Organization		X	X
Technology			X

如何计算CPI?

- 对于某一条特定的指令，其CPI是一个确定的值。
- 对于某一个程序或一台机器，其CPI是一个平均值，
表示该程序或该机器指令集中每条指令执行时平均需要多少时钟周期。

假定 CPI_i 和 C_i 分别为第 i 类指令的CPI和指令条数，则程序的总时钟数为：

$$\text{总时钟数} = \sum_{i=1}^n CPI_i \times C_i \qquad \text{CPU时间} = \text{时钟周期} \times \sum_{i=1}^n CPI_i \times C_i$$

假定 CPI_i 、 F_i 是各指令CPI和在程序中的出现频率，则程序综合CPI为：

$$CPI = \sum_{i=1}^n CPI_i \times F_i \quad \text{where } F_i = \frac{C_i}{\text{Instruction_Count}}$$

已知CPU时间、时钟频率、总时钟数、指令条数，则程序综合CPI为：

$$CPI = (\text{CPU 时间} \times \text{时钟频率}) / \text{指令条数} = \text{总时钟周期数} / \text{指令条数}$$

问题：指令的CPI、机器的CPI、程序的CPI各能反映哪方面的性能？

单靠CPI不能反映CPU性能！为什么？

例如，单周期处理器CPI=1，但性能差！

Example1

程序P在机器A上运行需10 s， 机器A的时钟频率为400MHz。
现在要设计一台机器B， 希望该程序在B上运行只需6 s.

机器B时钟频率的提高导致了其CPI的增加， 使得程序P在机器B上时钟周期数是在机器A上的1.2倍。 机器B的时钟频率达到A的多少倍才能使程序P在B上执行速度是A上的 $10/6=1.67$ 倍？

Answer:

$$\text{CPU时间A} = \text{时钟周期数A} / \text{时钟频率A}$$

$$\text{时钟周期数A} = 10 \text{ sec} \times 400\text{MHz} = 4000\text{M个}$$

$$\begin{aligned} \text{时钟频率B} &= \text{时钟周期数B} / \text{CPU时间B} \\ &= 1.2 \times 4000\text{M} / 6 \text{ sec} = 800 \text{ MHz} \end{aligned}$$

机器B的频率是A的两倍， 但机器B的速度并不是A的两倍！

Marketing Metrics （产品宣称指标）

MIPS = Instruction Count / (Time $1s \times 10^6$)
= Clock Rate / CPI $\times 10^6$

Million Instructions Per Second （定点指令执行速度）

每条指令执行时间不同，MIPS总是一个平均值。

- 不同机器的指令集不同
- 程序由不同的指令混合而成
- 指令使用的频度动态变化
- Peak MIPS: （不实用）

用MIPS数表示性能有没有局限？

MIPS数不能说明性能的好坏

MFLOPS = FP Operations / Time $\times 10^6$

Million Floating-point Operations Per Second （浮点操作速度）

- 不一定是程序中花时间的部分

用MFLOPS数表示性能也有一定局限！

问题：GFLOPS、TFLOPS、PFLOPS等的含义是什么？

Example: MIPS数不可靠!

(书中例1.3) Assume we build **an optimizing compiler** for the load/store machine. The compiler discards 50% of the ALU instructions.

- 1) What is the CPI? 仅在软件上优化，没涉及到任何硬件措施。
- 2) Assuming a 20 ns clock cycle time (50 MHz clock rate). What is the MIPS rating for optimized code versus unoptimized code? Does the MIPS rating agree with the rating of execution time?

Op	Freq	Cycle	Optimizing compiler	New Freq
ALU	43%	1	$21.5 / (21.5 + 21 + 12 + 24) = 27\%$	27%
Load	21%	2	$21 / (21.5 + 21 + 12 + 24) = 27\%$	27%
Store	12%	2	$12 / (21.5 + 21 + 12 + 24) = 15\%$	15%
Branch	24%	2	$24 / (21.5 + 21 + 12 + 24) = 31\%$	31%
1.57是如何算出来的?				
CPI	1.57		$50M / 1.57 = 31.8MIPS$	1.73
MIPS	31.8		$50M / 1.73 = 28.9MIPS$	28.9

结果：因为优化后减少了ALU指令（其他指令数没变），所以程序执行时间一定减少了，但优化后的MIPS数反而降低了。

浮点操作速度单位

问题：GFLOPS、TFLOPS、PFLOPS等的含义是什么？

浮点运算实际上包括了所有涉及小数的运算，在某类应用软件中常常出现，比整数运算更费时间。现今大部分的处理器中都有浮点运算器。因此每秒浮点运算次数所量测的实际上就是浮点运算器的执行速度。而最常用来测量每秒浮点运算次数的基准程序 (benchmark) 之一，就是Linpack。

- 一个MFLOPS (megaFLOPS) 每秒**一百万** ($=10^6$) 次的浮点运算,
- 一个GFLOPS (gigaFLOPS) 每秒**拾亿** ($=10^9$) 次的浮点运算,
- 一个TFLOPS (teraFLOPS) 每秒**万亿** ($=10^{12}$) 次的浮点运算,
- 一个PFLOPS (petaFLOPS) 每秒**千万亿** ($=10^{15}$) 次的浮点运算,
- 一个EFLOPS (exaFLOPS) 每秒**百亿亿** ($=10^{18}$) 次的浮点运算。

全球超级计算机500强

2017年6月19号公布:

- 第一名: 中国国家超级计算无锡中心研制的“神威·太湖之光” 浮点运算速度为每秒9.3亿亿次。
- 第二名: 国防科大研制的“天河二号” 超级计算机, 每秒3.386亿亿次的浮点运算速度, 之前曾获得五连冠。

速度单位是 TfloP/s 或 TFLOPS

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Supercomputing Center in Wuxi China	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRPC	10,649,600	93,014.6	125,435.9	15,371
2	National Super Computer Center in Guangzhou China	Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
3	Swiss National Supercomputing Centre (CSCS) Switzerland	Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 Cray Inc.	361,760	19,590.0	25,326.3	2,272

选择性性能评价程序（Benchmarks）

- 用基准程序来评测计算机的性能
 - 基准测试程序是专门用来进行性能评价的一组程序
 - 基准程序通过运行实际负载来反映计算机的性能
 - 最好的基准程序是用户实际使用的程序或典型的简单程序
- 基准程序的缺陷
 - 现象：基准程序的性能与某段短代码密切相关时，会被利用以得到不当的性能评测结果
 - 手段：硬件系统设计人员或编译器开发者针对这些代码片段进行特殊的优化，使得执行这段代码的速度非常快
 - 例1：Intel Pentium处理器运行SPECint时用了公司内部使用的特殊编译器，使其性能极高
 - 例2：矩阵乘法程序SPECmatrix300有99%的时间运行在一行语句上，有些厂商用特殊编译器优化该语句，使性能达VAX11/780的729.8倍！

Amdahl定律

- 阿姆达尔定律是计算机系统设计方面重要的定量原则之一
 - 基本思想：对系统中某部分（硬件或软件）进行更新所带来的系统性能改进程度，取决于该部分被使用的频率或其执行时间占总执行时间的比例。

$$\text{改进后的执行时间} = \frac{\text{改进部分执行时间}}{\text{改进部分的改进倍数}} + \text{未改进部分执行时间}$$

或

$$p = 1 / (t/n + 1 - t)$$

$$\text{整体改进倍数} = \frac{1}{\text{改进部分时间比例} / \text{改进部分的改进倍数} + \text{未改进部分时间比例}}$$

若整数乘法器改进后可加快**10倍**，整数乘法指令在程序中占40%，则整体性能可改进多少倍？若占比达60%和90%，则整体性能分别能改进多少倍？

$$40\%: 1 / (0.4/10 + 0.6) = 1.56;$$

$$60\%: 1 / (0.6/10 + 0.4) = 2.17;$$

$$90\%: 1 / (0.9/10 + 0.1) = 5.26。$$

Amdahl定律

例：某程序在某台计算机上运行所需时间是100秒，其中，80秒用来执行乘法操作。要使该程序的性能是原来的5倍，若不改进其他部件而仅改进乘法部件，则乘法部件的速度应该提高到原来的多少倍？

解：根据公式 $p=1/(t/n + 1-t)$ 知：

$$5=1/(0.8/n+0.2) , \quad 0.8/n+0.2 = 1/5 = 0.2$$

要使上述公式满足，则必须 $0.8/n=0$ ，即 $n \rightarrow \infty$

也就是说，即使乘法运算时间占80%，也不可能通过对乘法部件的改进，使整体性能提高到原来的5倍。

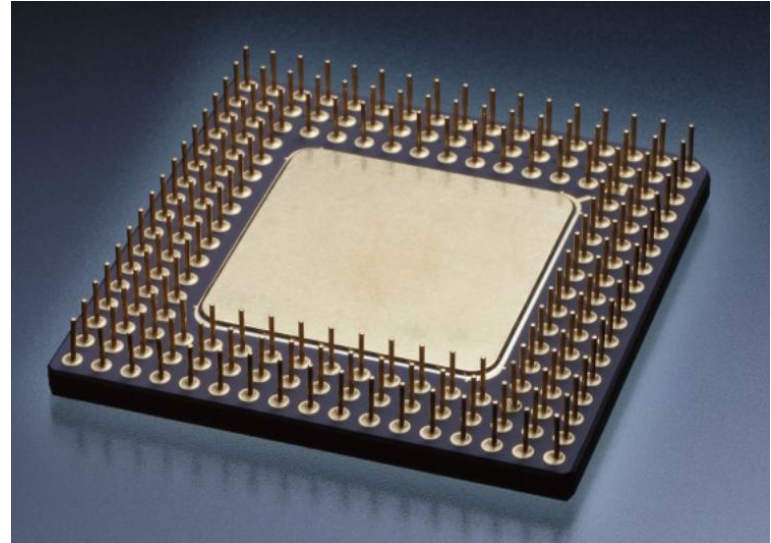
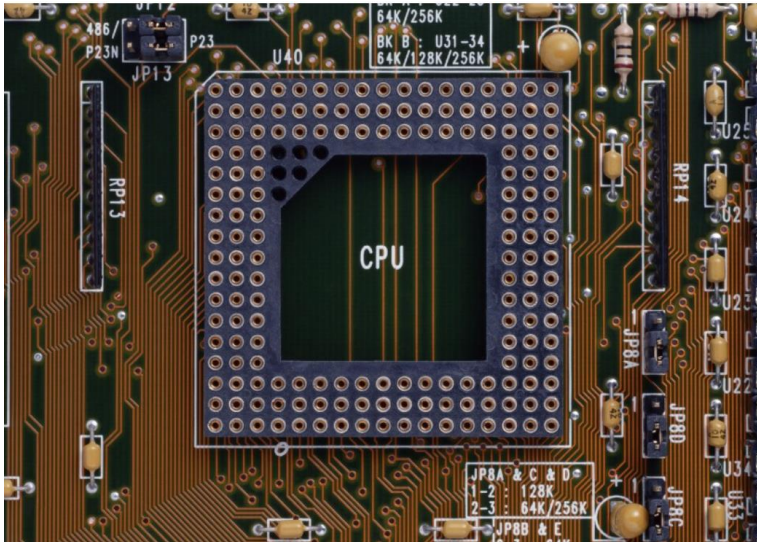
当乘法运算时间占比 $\leq 80\%$ ，则无论如何对乘法部件进行改进，都不能使整体性能提高到原来的5倍。

总结

中央处理器，CPU，Central Processing Unit。

计算机的运算核心和控制核心，是信息处理、程序运行的最终执行单元。

算术逻辑部件，ALU，Central Processing Unit。



总结

中央处理器 , CPU, Central Processing Unit.

算术逻辑部件 , ALU, Arithmetic Logical Unit

通用寄存器 GPRs, General Purpose Registers

程序计数器 PC , Program Counter

指令寄存器 IR, Instruction Register

控制器

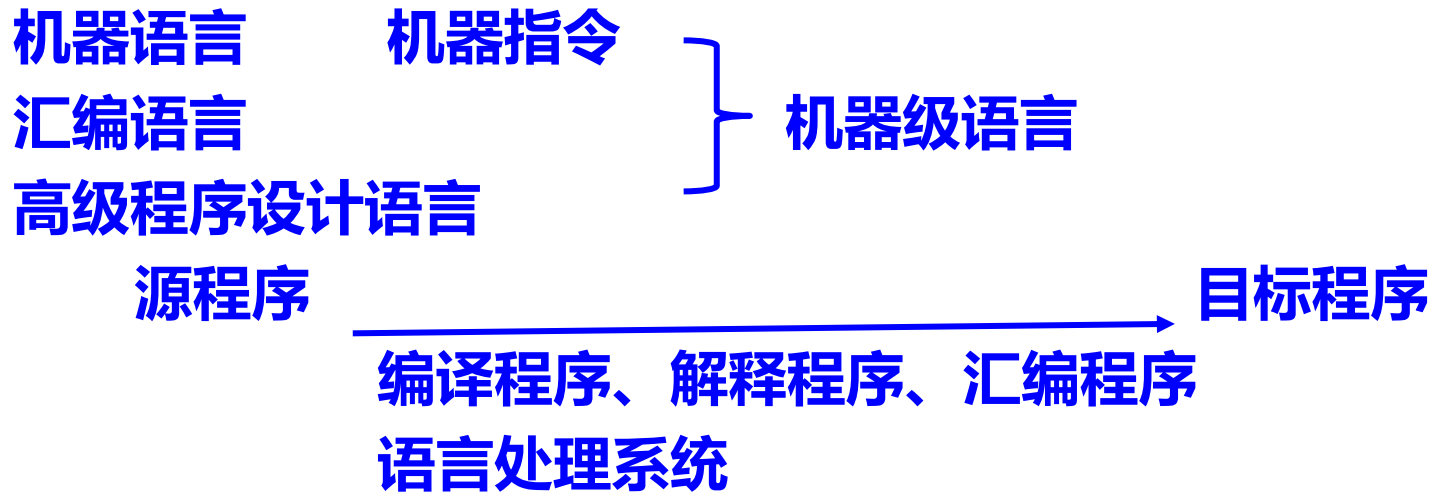
主存地址寄存器 MAR, Memory Address Register

主存数据寄存器 MDR, Memory Data Register

主存储器 Memory

总线 Bus

总结



最终用户、系统管理员、应用程序员、系统程序员

指令集体系结构 ISA, Instruction Set Architecture
微体系结构 透明
微操作
设备控制器

总结

系统性能 响应时间 吞吐率

用户CPU时间 系统CPU时间

CPU性能 时钟周期 主频 CPI

基准程序 SPEC基准程序集 SPEC 比值

MIPS 峰值 MIPS 相对 MIPS

MFLOPS