

# 第一章 计算机系统概论

“计算机系统基础”课程的由来

“计算机系统基础”课程内容概要

计算机系统概述

计算机性能评价

# 主要内容

---

- 课程的由来
- 课程内容概要
- 课程教学安排及考试安排
- 冯·诺依曼结构计算机特点
- 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

# 用“系统思维”分析问题

ISO C90标准下，在32位系统上

以下C表达式的结果是什么？

$-2147483648 < 2147483647$

false (与事实不符) ! Why?

ISO C99标准下为true, Why?

以下关系表达式结果呢？

int i = -2147483648;

i < 2147483647

true! Why?

$-2147483647-1 < 2147483647$ , 结果怎样?

理解该问题需要知道：

编译器如何处理字面量 (常量)

高级语言中运算规则

高级语言与指令之间的对应

机器指令的执行过程

机器级数据的表示和运算

.....

# 用“系统思维”分析问题

```
sum(int a[ ], unsigned len)
{
    int i, sum = 0;
    for (i = 0; i <= len-1; i++)
        sum += a[i];
    return sum;
}
```

当参数len为0时，返回值应该是0，但是在机器上执行时，却发生访存异常。但当len为int型时则正常。Why?

理解该问题需要知道：

高级语言中运算规则

机器指令的含义和执行

计算机内部的运算电路

异常的检测和处理

虚拟地址空间

.....



# 用“系统思维”分析问题

若x和y为int型，当 $x=65535$ 时， $y=x*x$ ；y的值为多少？

$y=-131071$ 。Why?

现实世界中， $x^2 \geq 0$ ，但在计算机世界并不一定成立。

对于任何int型变量x和y， $(x > y) == (-x < -y)$  总成立吗？

当 $x=-2147483648$ ，y任意（除-2147483648外）时不成立

Why?

在现实世界中成立，

但在计算机世界中并不一定成立。

理解该问题需要知道：

机器级数据的表示

机器指令的执行

计算机内部的运算电路

# 用“系统思维”分析问题

main.c

```
int d=100;
int x=200;
int main()
{
    p1( );
    printf ( "d=%d, x=%d\n" , d, x );
    return 0;
}
```

p1.c

```
double d;

void p1( )
{
    d=1.0;
}
```

**打印结果是什么？**

**d=0, x=1 072 693 248**

**Why?**

理解该问题需要知道：  
机器级数据的表示  
变量的存储空间分配  
数据的大端/小端存储方式  
链接器的符号解析规则

.....

# 用“系统思维”分析问题

代码段一：

```
int a = 2147483648;  
int b = a / -1;  
printf("%d, %d \n", a, b);
```

运行结果为

**-2147483648, -2147483648**

Warning: this decimal constant is unsigned  
only in ISO C90[enabled by default]

代码段二：

```
int a = 2147483648;  
int b = -1;  
int c = a / b;  
printf("%d, %d\n", a, c);
```

运行结果为“Floating point exception”，显然CPU检测到了溢出异常

上述结果在Linux上获得，为什么两者结果不同？

在Windows上运算的结果又为何不同？

理解该问题需要知道：

机器级数据的表示

(如：真值和机器数的关系)

机器指令的含义和执行

(如：取负指令、除法指令)

计算机内部的运算电路

(如：除法电路会判是否异常)

编译器如何优化

(如：a/-1可用取负指令实现)

操作系统如何处理异常

(如：除法错异常的处理)

.....

# 用“系统思维”分析问题

以下是一段C语言代码：

```
#include <stdio.h>
main()
{
    double a = 10;
    printf("a = %d\n", a);
}
```

理解该问题需要知道：

IEEE 754 的表示

X87 FPU的体系结构

IA-32和x86-64中过程  
调用的参数传递

计算机内部的运算电路

.....

在IA-32上运行时，打印结果为a=0

在x86-64上运行时，打印出来的a是一个不确定值

为什么？



# 用“系统思维”分析问题

```
double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}
```

对于上述C语言函数， $i=0\sim 4$ 时， $\text{fun}(i)$ 分别返回什么值？

$\text{fun}(0) \rightarrow 3.14$   
 $\text{fun}(1) \rightarrow 3.14$   
 $\text{fun}(2) \rightarrow 3.1399998664856$   
 $\text{fun}(3) \rightarrow 2.00000061035156$   
 $\text{fun}(4) \rightarrow 3.14$ , 然后存储保护错

Why?

理解该问题需要知道：

机器级数据的表示

过程调用机制


栈帧中数据的布局

.....

# 用“系统思维”分析问题

```
void copyij (int src[2048][2048],
             int dst[2048][2048])
{
    int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}
```

```
void copyji (int src[2048][2048],
             int dst[2048][2048])
{
    int i,j;
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
}
```



以上两个程序功能完全一样，算法完全一样，因此，时间和空间复杂度完全一样，执行时间一样吗？

**21 times slower**  
(Pentium 4)  
**Why?**

理解该问题需要知道：  
数组的存放方式  
Cache机制  
访问局部性

.....

# 用“系统思维”分析问题

使用老版本gcc -O2编译时，程序一输出0，程序二输出却是1

Why?

程序一：

```
#include <stdio.h>
double f(int x) {
    return 1.0 / x ;
}
void main() {
    double a, b;
    int i;
    a = f(10) ;
    b = f(10) ;
    i = a == b ;
    printf( "%d\n" , i ) ;
}
```

程序二：

```
#include <stdio.h>
double f(int x) {
    return 1.0 / x ;
}
void main() {
    double a, b, c;
    int i;
    a = f(10) ;
    b = f(10) ;
    c = f(10) ;
    i = a == b ;
    printf( "%d\n" , i ) ;
}
```

# 用“系统思维”分析问题

```
#include <stdio.h>
void main()
{
    int a = 10;
    double *p = (double*)&a;
    printf("%f\n", *p); //结果为不确定值
    printf("%f\n", (double)a); //结果为10.000000
}
```

理解该问题需要知道：

数据的表示

编译（程序的转换）

局部变量在栈中的位置

.....

为什么 `printf("%f\n", *p)` 和 `printf("%f\n", (double)a)` 结果不一样？

不都是强制类型转换吗？怎么会不一样

关键差别在于一条指令：

**fldl** 和 **fildl**

# 你在想什么？

---

- 看了前面的举例，你的感觉是什么呢？
  - 计算机好像不可靠    从机器角度来说，它永远对！你的感觉不可靠！
  - 程序执行结果不仅依赖于高级语言语法和语义，还与其他好多方面有关  
理解程序的执行结果要从系统层面考虑！
  - 本来以为学编程和计算机基本原理就能当程序员，没想到还挺复杂的，并不是那么简单  
学完“计算机系统基础”就会对计算机系统有清晰的认识，  
将其他相关课程的内容联系起来。
  - 要把很多概念和知识联系起来才能理解程序的执行结果  
把许多概念和知识联系起来就是计算机“系统思维”。  
即：站在“计算机系统”的角度考虑问题！

# 系统能力基于“系统思维”

- 系统思维

- 从计算机系统角度出发分析问题和解决问题
- 首先取决于对计算机系统有多了解，“知其然并知其所以然”

基本认识

- 高级语言语句都要转换为机器指令才能在计算机上执行
- 机器指令是一串0/1序列，能被机器直接理解并执行
- 计算机系统是模运算系统，字长有限，高位被丢弃
- 运算器不知道参加运算的是带符号数还是无符号数
- 在计算机世界， $x * x$ 可能小于0， $(x + y) + z$ 不一定等于 $x + (y + z)$
- 访问内存需几十到几百个时钟，而访问磁盘要几百万个时钟
- 进程具有独立的逻辑控制流和独立的地址空间
- 过程调用使用栈存放参数和局部变量等，递归过程有大量额外指令，增加时间开销，并可能发生栈溢出
- .....

只有先理解系统，才能优化系统，并应用好系统!

# 为什么要学习“计算机系统基础”？

---

- 为什么要学习“计算机系统基础”呢？
  - 强化“系统思维”
  - 更好地理解计算机系统，从而编写出更好的程序
  - 编程时少出错
  - 在程序出错时很快找到出错的地方
  - 编写出更快的程序
  - 明白程序是怎样在计算机上执行的
  - 为进一步的学习打下良好基础
  - .....

# 主要内容

---

- 课程的由来
- 课程内容概要
- 课程教学安排及考试安排
- 硬件和软件的基本组成
- 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



# 什么是计算机系统？

## 计算机系统抽象层的转换

程序执行结果

不仅取决于  
算法、程序编写

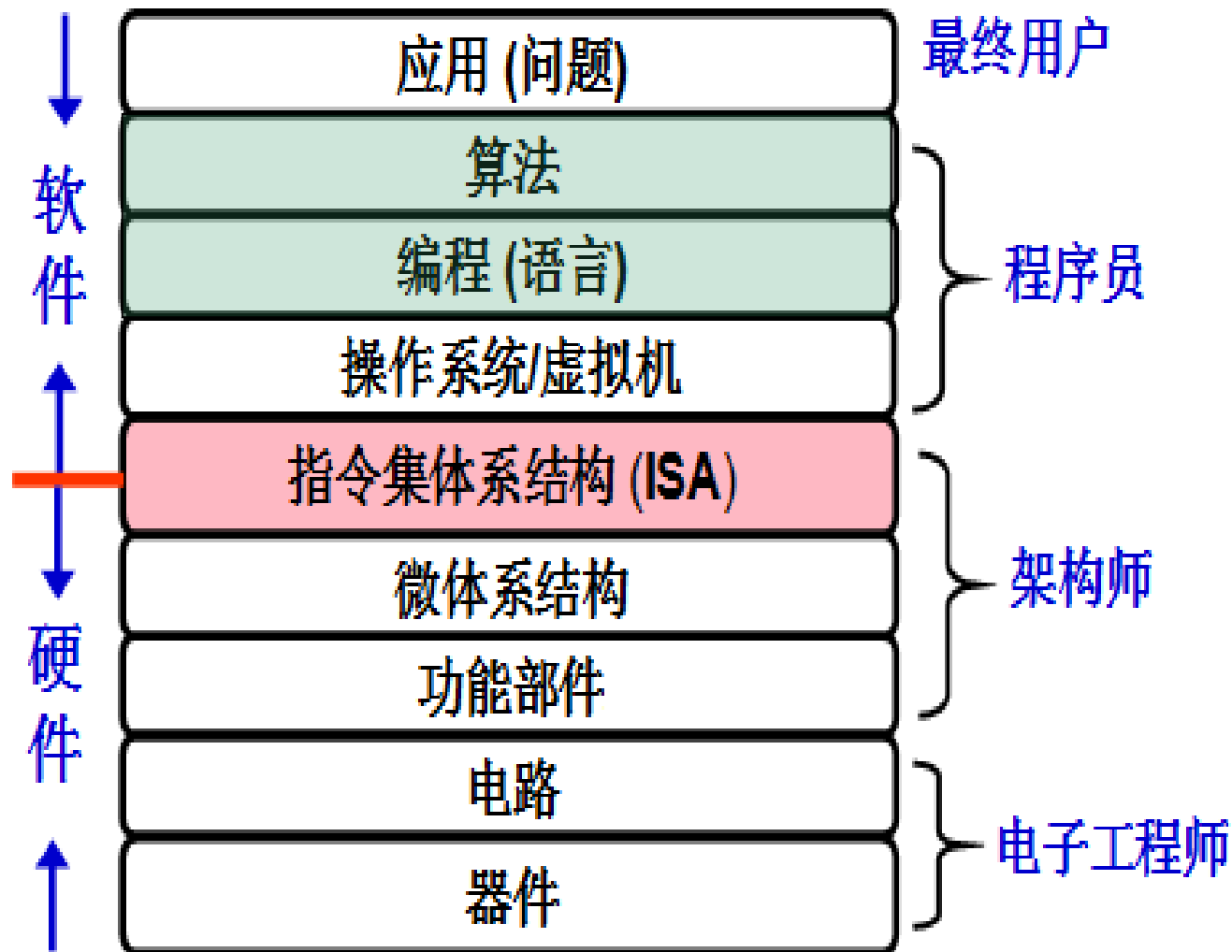
而且取决于  
语言处理系统  
操作系统

ISA

微体系结构

不同计算机课程  
处于不同层次

必须将各层次关  
联起来解决问题



# “计算机系统基础” 内容提要

**课程目标：** 清楚理解计算机是如何生成和运行可执行文件的！

**重点在高级语言以下各抽象层**

- **C语言程序设计层**

- 数据的机器级表示、运算
- 语句和过程调用的机器级表示

- **操作系统、编译和链接的部分内容**

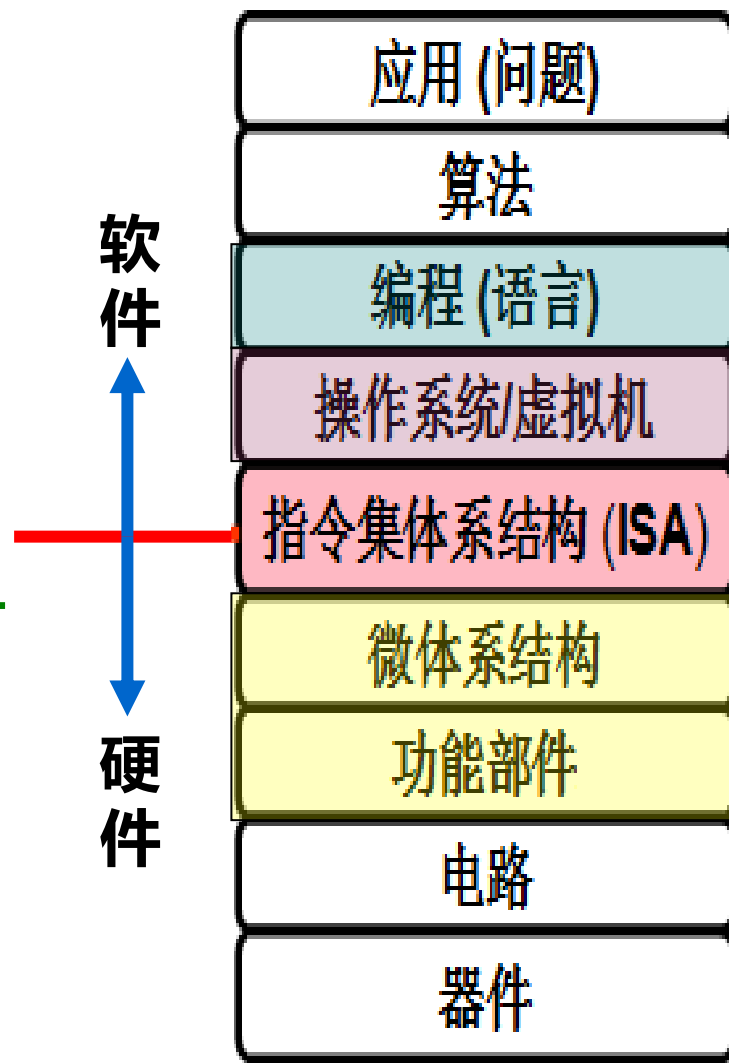
- **指令集体系结构 (ISA) 和汇编层**

- 指令系统、机器代码、汇编语言

- **微体系结构及硬件层的部分内容**

- CPU的通用结构
- 层次结构存储系统

## 计算机系统抽象层



# 课程内容概要

```
/*---sum.c---*/
```

```
int sum(int a[ ], unsigned len)
{
    int i, sum = 0;
    for (i = 0; i <= len-1; i++)
        sum += a[i];
    return sum;
}
```

```
/*---main.c---*/
```

```
int main()
{
    int a[1]={100};
    int sum;
    sum=sum(a,0);
    printf("%d",sum);
}
```

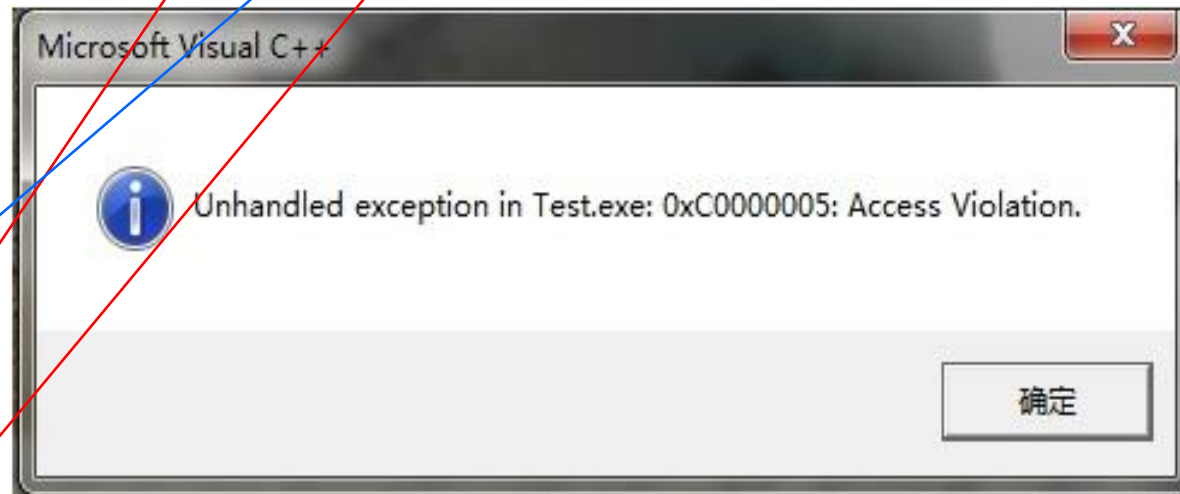
数据的表示

数据的运算

各类语句的转换与表示(指令)

各类复杂数据类型的转换表示

过程（函数）调用的转换表示



链接 (linker) 和加载

程序执行 (存储器访问)

异常和中断处理

输入输出(I/O)

# 课程内容概要

---

## 三个主题:

- **表示 (Representation)**
  - 不同数据类型（包括带符号整数、无符号整数、浮点数、数组、结构等）在寄存器或存储器中如何表示和存储？
  - 指令如何表示和编码（译码）？
  - 存储地址（指针）如何表示以及如何生成复杂数据结构中数据元素的地址？
- **转换 (Translation) 和链接 (Link)**
  - 高级语言程序对应的机器级代码是怎样的？如何合并成可执行文件？
- **执行控制流 (Control flow)**
  - 计算机能理解的“程序”是如何组织和控制的？
  - 如何在计算机中组织多个程序的并发执行？
  - 逻辑控制流中的异常事件及其处理
  - I/O操作的执行控制流（用户态→内核态）

# 计算机系统基础—从程序员角度认识系统

---

- 培养目标:

培养学生的**系统能力**，使其成为一个“**高效**”程序员，在程序调试、性能提升、程序移植和健壮性等方面成为高手；建立扎实的计算机系统概念，将操作系统、编译、组成原理等课程的内容结合起来

- 以 **IA-32+Linux+C+gcc** 为平台（开源项目平台）

- 源于CMU课程:

<https://csapp.cs.cmu.edu/>

**主要内容：描述程序执行的底层机制**

- 学习方法:

在程序与执行机制之间**建立关联**，**强化理解**而不是记忆

将所学内容用C语言程序编程实现出来

# 课程内容概要

---

**前置知识：**C语言程序设计、数字逻辑电路基础

**内容组织：**两大部分

- **第一部分 系统概述和可执行文件的生成（表示和转换）**
  - 计算机系统概述
  - 数据的机器级表示与处理
  - 程序的转换及机器级表示
  - 程序的链接
- **第二部分 可执行文件的运行（执行控制流）**
  - 程序的执行
  - 层次结构存储系统
  - 异常控制流
  - I/O操作的实现

# 主要内容

---

- 课程的由来
- 课程内容概要
- 课程教学安排及考试安排
- 冯·诺依曼结构计算机特点
- 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

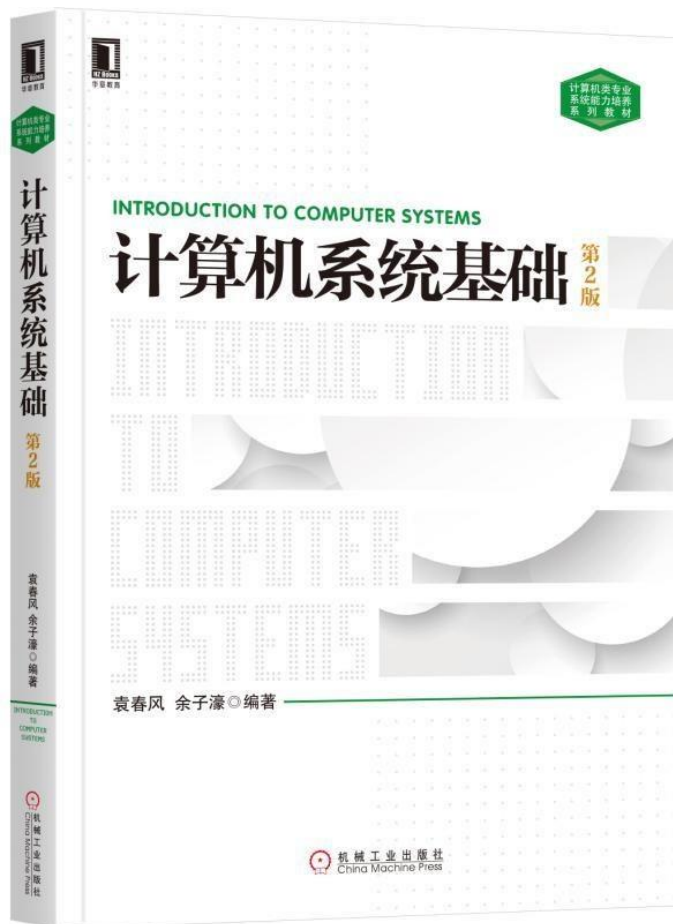
# 课程基本信息

---

- 课程名称
  - 计算机系统基础 (Introduction to Computer Systems)
- MOOC网站 (4门系列课程)
  - <https://www.icourse163.org/course/NJU-1001625001>
  - <https://www.icourse163.org/course/NJU-1001964032>
  - <https://www.icourse163.org/course/NJU-1002532004>
  - <http://www.icourse163.org/course/NJU-1449521162>
- 前导课程
  - C语言程序设计、 (数字逻辑电路, 计算机组成原理, 不是必须的)
- 教材:
  - 《计算机系统基础 (第2版) 》, 袁春风, 机械工业出版社, 2018.7
- 主要参考书:
  - 《深入理解计算机系统》 (第3版) , Randal E. Bryant, david R. O' Hallaron著, 龚奕利, 雷迎春, 译, 机械工业出版社, 2016 年
  - Brian W. Kernighan, Dennis M. Ritchie, The C Programming Language (Second Edition), 北京: 机械工业出版社, 2006



# 课程基本信息



# 课程基本信息

---

- **主要参考书：**

- **计算机系统概论（原书第2版），[美] Yale N. Patt, Sanjay J. Patel 著，梁阿磊, 蒋兴昌, 林凌, 译，机械工业出版社，2018年8月**
- **计算机系统：系统架构与操作系统的高度集成，Umakishore Ramachandran, William D. Leahy Jr著，陈文光, 译，机械工业出版社，2015年7月**
- **计算机系统：嵌入式方法，Ian Vince McLoughlin著，刘雯, 译，机械工业出版社，2020年6月**
- **计算机系统：基础概念及编程实践，钱晓捷, 著，机械工业出版社，2018年9月**

# 实验及考核方式

---

- **实验类型**

- **Homework:** 编程/调试、作业习题
- **Lab:** 二进制逆向工程、数据表示（位操作）、程序表示、二进制炸弹、缓冲区溢出、链接与ELF
- **Project:** 二进制炸弹、缓冲区溢出、链接与ELF

- **考核方式**

- **平时成绩（作业习题、Lab实验）：**  $\geq 50\%$
- **期末考试：**  $\leq 50\%$

# 主要内容

---

- 课程的由来
- 课程内容概要
- 课程教学安排及考试安排
- 冯·诺依曼结构计算机特点
- 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

# 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

1946年，第一台**实际使用的**电子数字计算机**ENIAC**诞生

- 由电子真空管组成
- 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

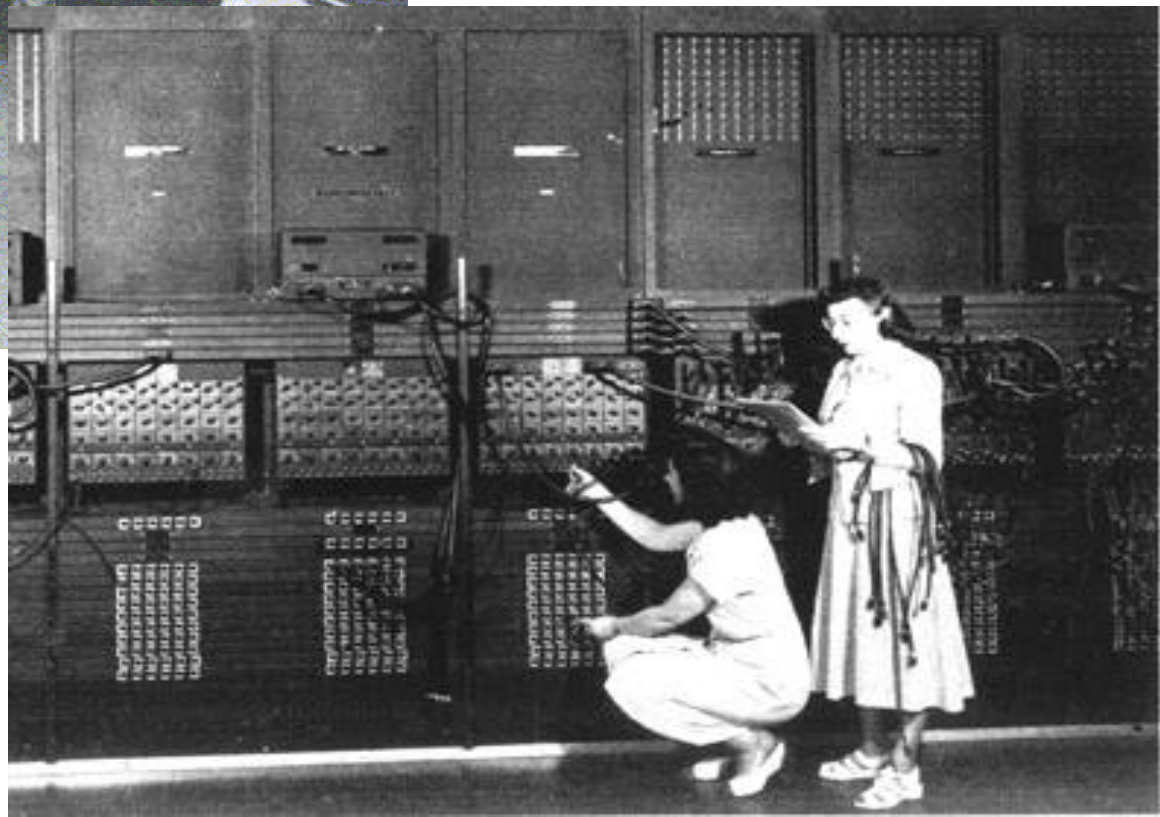
“世界第一台电子计算机”之争：1973年美国明尼苏达地区法院推翻并吊销了莫克利的专利。美国艾奥瓦州立大学约翰·文森特·阿塔那索夫（John Vincent Atanasoff）被称为“电子计算机之父”

# Electronic Numerical Integrator And Computer



占地面积170平方米  
重30吨  
有18000多个真空管  
耗电160千瓦

该机正式运行到  
1955年10月2日，  
这十年间共运行  
80 223个小时





# 冯·诺依曼的故事

- 1944年，冯·诺依曼参加原子弹的研制工作，涉及极为困难的计算。
- 1944年夏的一天，诺依曼巧遇美国弹道实验室的军方负责人戈尔斯坦，他正参与ENIAC的研制工作。
- 冯·诺依曼被戈尔斯坦介绍加入ENIAC研制组，1945年，他们在共同讨论的基础上，冯·诺依曼以“关于EDVAC的报告草案”为题，起草了长达101页的总结报告，发表了全新的“**存储程序通用电子计算机方案**”。
- 一向专搞理论研究的**普林斯顿高等研究院**批准让冯·诺依曼建造计算机，其依据就是这份报告。



**E**lectronic  
**D**iscrete  
**V**ariable  
**A**utomatic  
**C**omputer

# 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是什么？**

**“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。**

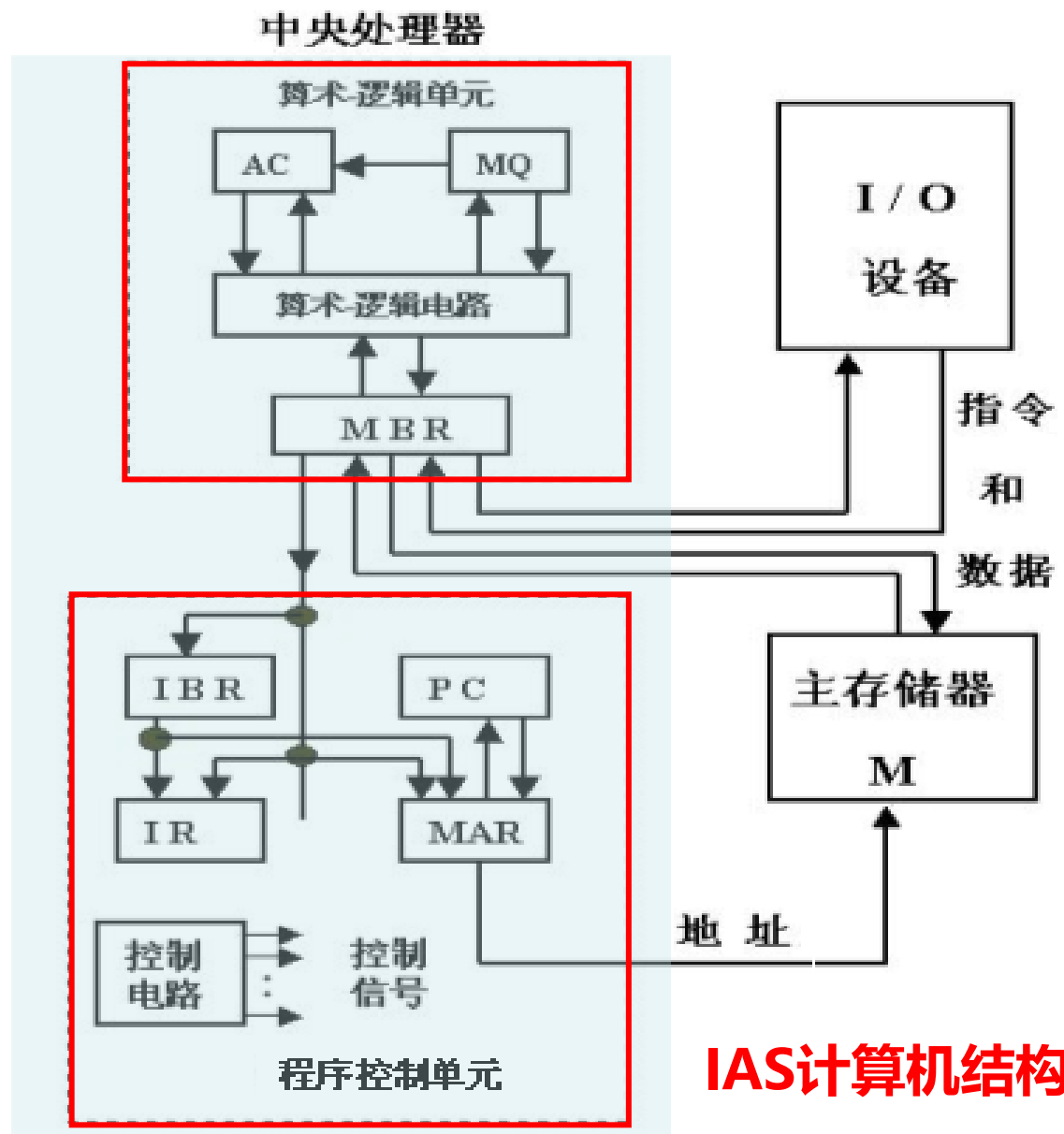


# 你认为冯·诺依曼结构是怎样的？

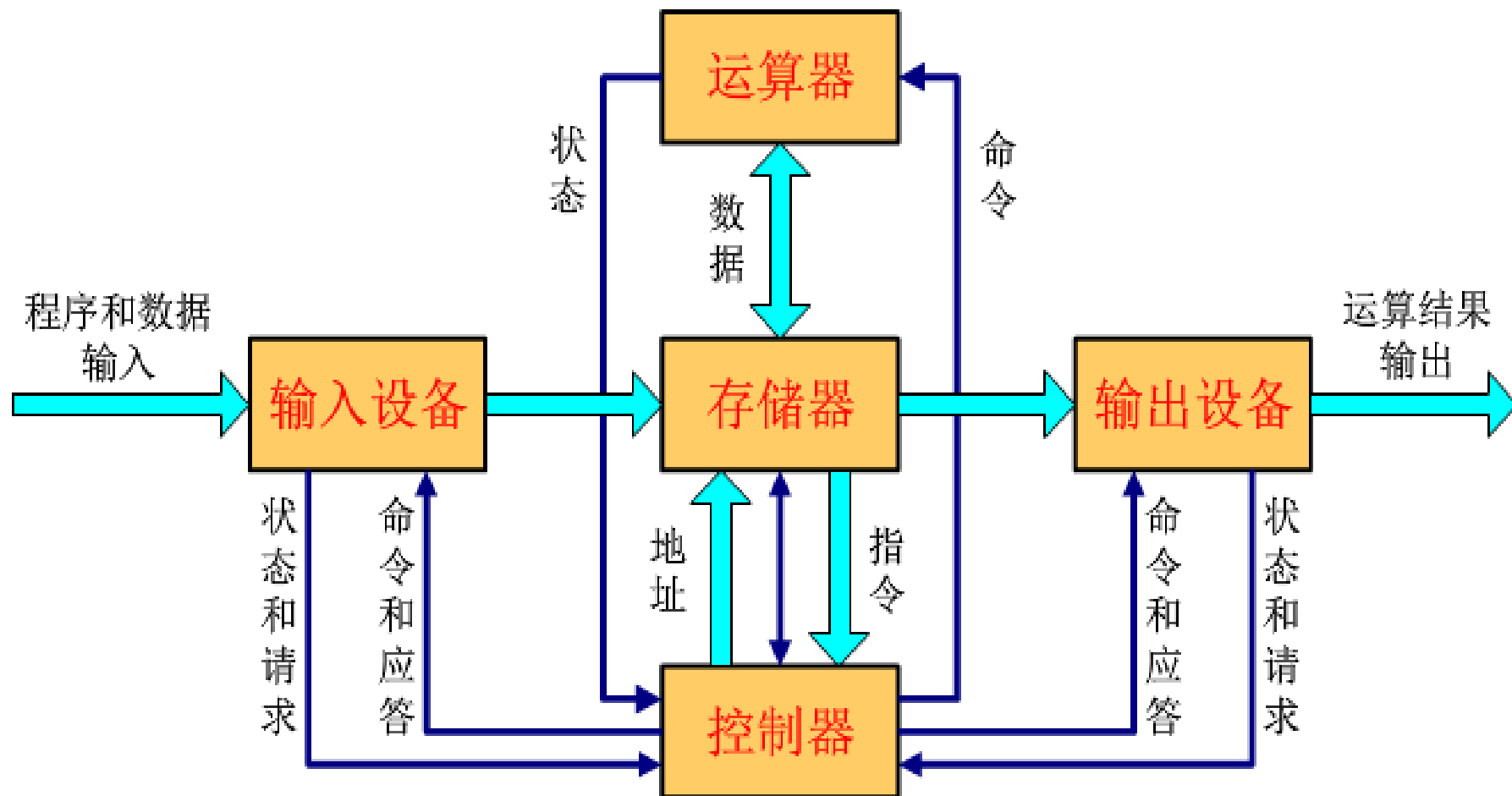
- 应该有个主存，用来存放程序和数据
- 程序由指令构成
- 应该有一个自动逐条取出指令的部件
- 还应该具体执行指令（即运算）的部件
- 指令描述如何对数据进行处理
- 应该有将程序和原始数据输入计算机的部件
- 应该有将运算结果输出计算机的部件

你还能想出更多吗？

你猜得八九不离十了☺



# 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

---

冯·诺依曼结构的主要思想是什么呢？

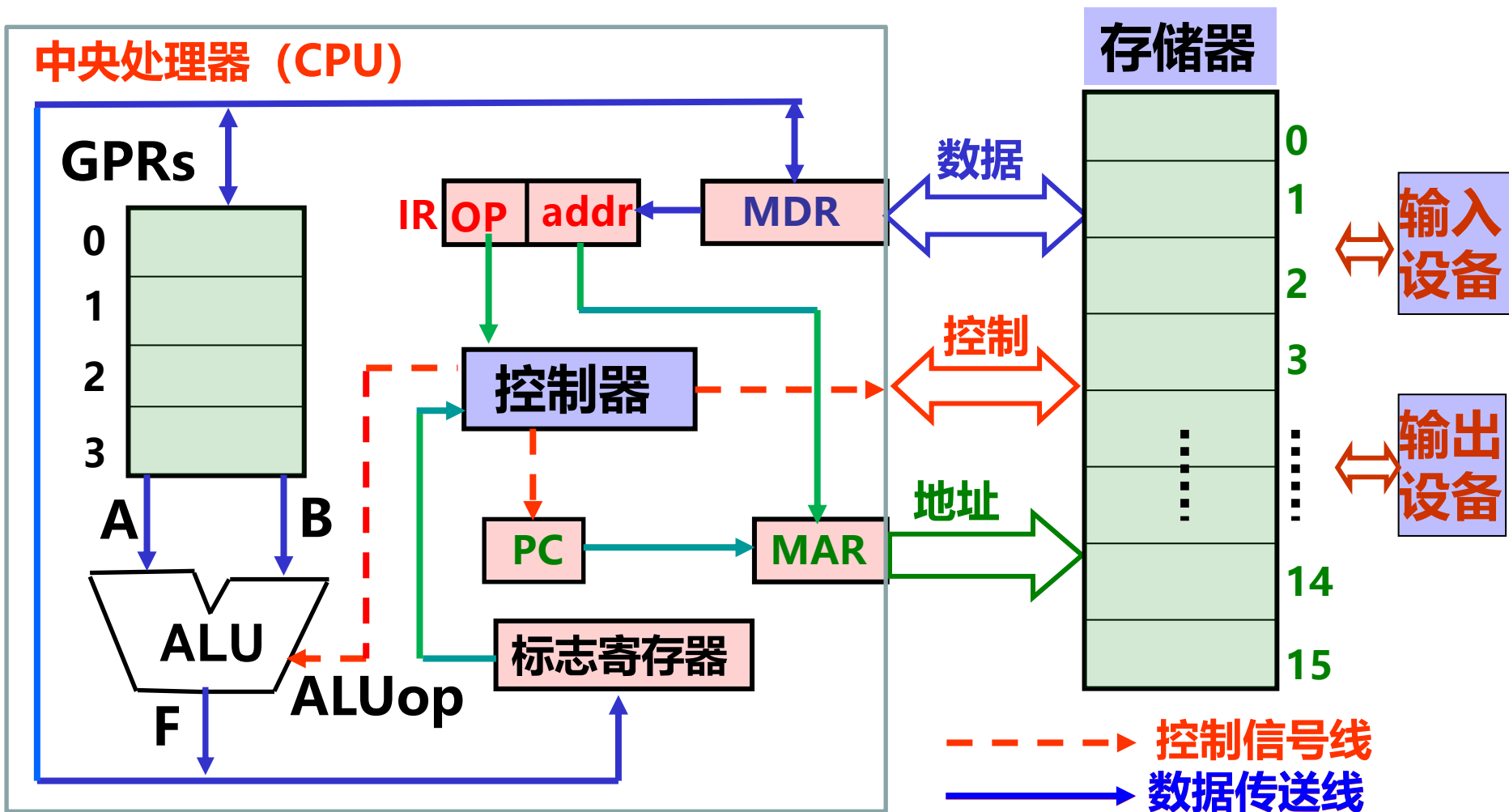
1. 计算机应由运算器、控制器、存储器、输入设备和输出设备五个基本部件组成。
2. 各基本部件的功能是：
  - **存储器**不仅能存放数据，而且也能存放指令，形式上两者没有区别，但计算机应能区分数据还是指令；
  - **控制器**应能自动取出指令来执行；
  - **运算器**应能进行加/减/乘/除四种基本算术运算，并且也能进行一些逻辑运算和附加运算；
  - 操作人员可以通过**输入设备**、**输出设备**和主机进行通信。
3. 内部以**二进制表示**指令和数据。每条指令由操作码和地址码两部分组成。操作码指出操作类型，地址码指出操作数的地址。由一串指令组成程序。
4. 采用“**存储程序**”工作方式。

# 认识计算机中最基本的部件

**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)



# 计算机是如何工作的？

---

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

第三步：取操作数

第四步：指令执行

第五步：回写结果

第六步：修改PC的值

继续执行下一条指令，返回第一步

# 计算机是如何工作的？

- **程序启动前**，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
- 采用“**存储程序**”工作方式：
  - 程序由指令组成，程序被启动后，计算机能自动取出一条一条指令执行，在执行过程中无需人的干预。
- **指令执行过程中**，指令和数据被从存储器取到CPU，存放在CPU内的寄存器中，指令在**IR**中，数据在**GPR**中。

**指令中需给出的信息：**            **IR? GPR?**

**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！**

# 主要内容

---

- 课程的由来
- 课程内容概要
- 课程教学安排及考试安排
- 冯·诺依曼结构计算机特点
- 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

# 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: .....

太原始了，无法忍受，咋办？

用符号表示而不用0/1表示！

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！



# 用汇编语言开发程序

- 若用**符号**表示跳转位置和变量位置，是否简化了问题？

- 于是，汇编语言出现

- 用**助记符**表示操作码
- 用**标号**表示位置
- 用**助记符**表示寄存器
- .....

0: 0101 0110

1: 0010 0100

2: .....

3: .....

4: 0110 0111

5: .....

6: .....

7: .....

sub B

jnz L0

.....

.....

L0: add C

.....

B: .....

C: .....

你认为用汇编语言编写的优点是：

不会因为增减指令而需要修改其他指令

不需记忆指令码，编写方便

可读性比机器语言强

不过，这带来新的问题，是什么呢？

人容易了，可机器不认识这些指令了！

需将汇编语言转换  
为机器语言！


用汇编程序转换

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！

# 进一步认识机器级语言

- 汇编语言源程序由**汇编指令**构成
- 你能用一句话描述**什么是汇编指令**吗？
  - 用助记符和标号来表示的指令（与机器指令一一对应）
- **指令**又是什么呢？
  - 包含操作码和操作数或其地址码  
(**机器指令**用**二进制表示**，**汇编指令**用**符号表示**)
  - 可以描述：取（或存一个数）  
两个数加（或减、乘、除、与、或等）  
根据运算结果判断是否转移执行
- 想象用**汇编语言**编写复杂程序是怎样的情形？  
(例如，用汇编语言实现排序（sort）、矩阵相乘)
  - 需要描述的细节太多了！程序会很长很长！而且在不同结构的机器上就不能运行！

```
sub B
jnz L0
.....
.....
L0: add C
.....
B: .....
C: .....
```



机器语言和汇编语言都是面向机器结构的语言，故它们统称为**机器级语言**

**SKIP**

结论：用汇编语言比机器语言好，但是，还是很麻烦！

# 指令所能描述的功能

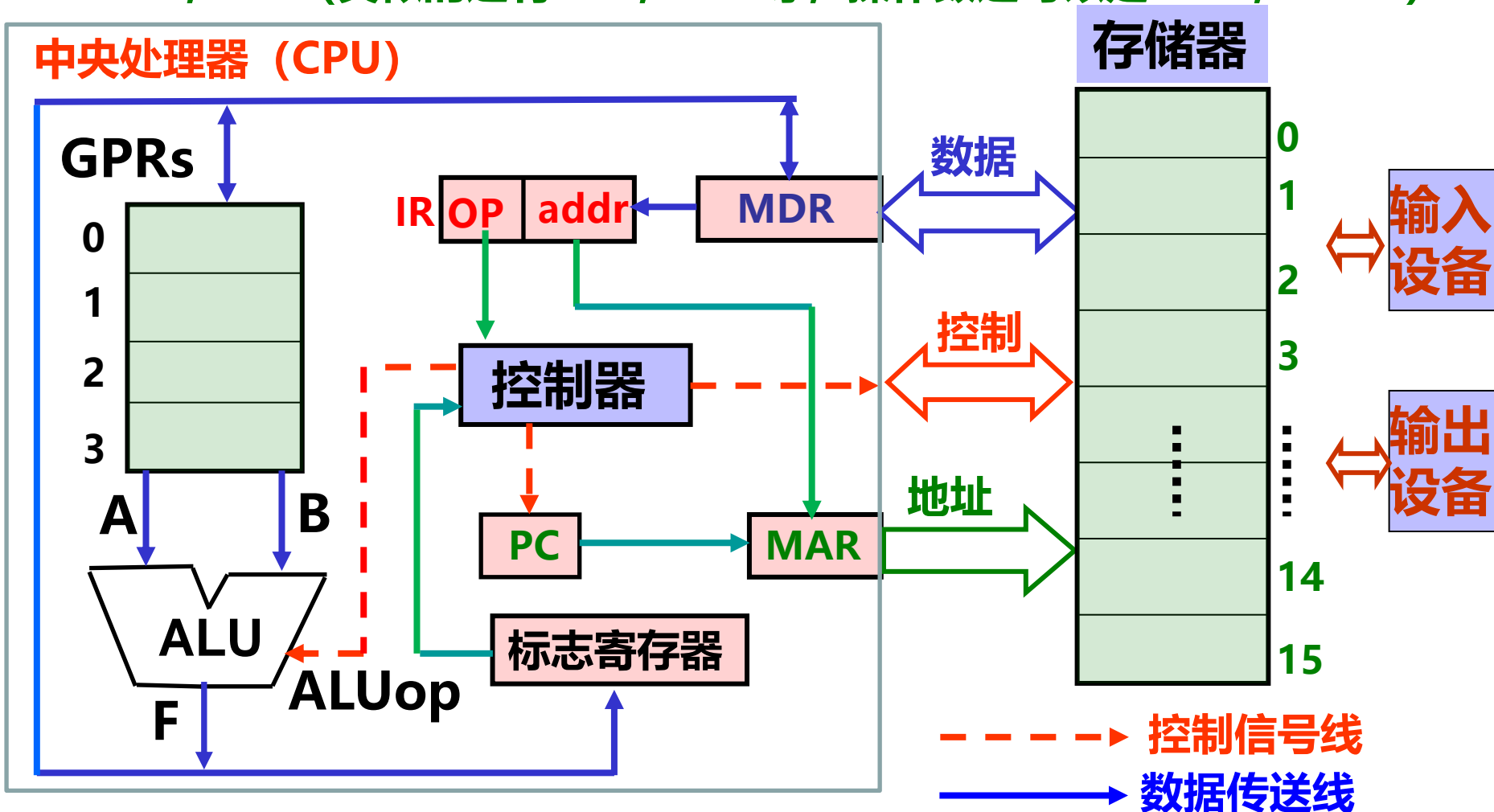
对于以下结构的机器，你能设计出几条指令吗？

Load M#, R# (将存储单元内容装入寄存器)

Store R#, M# (将寄存器内容装入存储单元)

Add R#, R# (类似的还有Sub, Mul等；操作数还可以是“R#, M#”)

[BACK](#)



# 用高级语言开发程序

- 随着技术的发展，出现了许多高级编程语言
    - 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
    - 处理逻辑分为三种结构
      - 顺序结构、选择结构、循环结构
    - 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。
- 现在，几乎所有程序员都用高级语言编程，但最终要将高级语言转换为机器语言程序

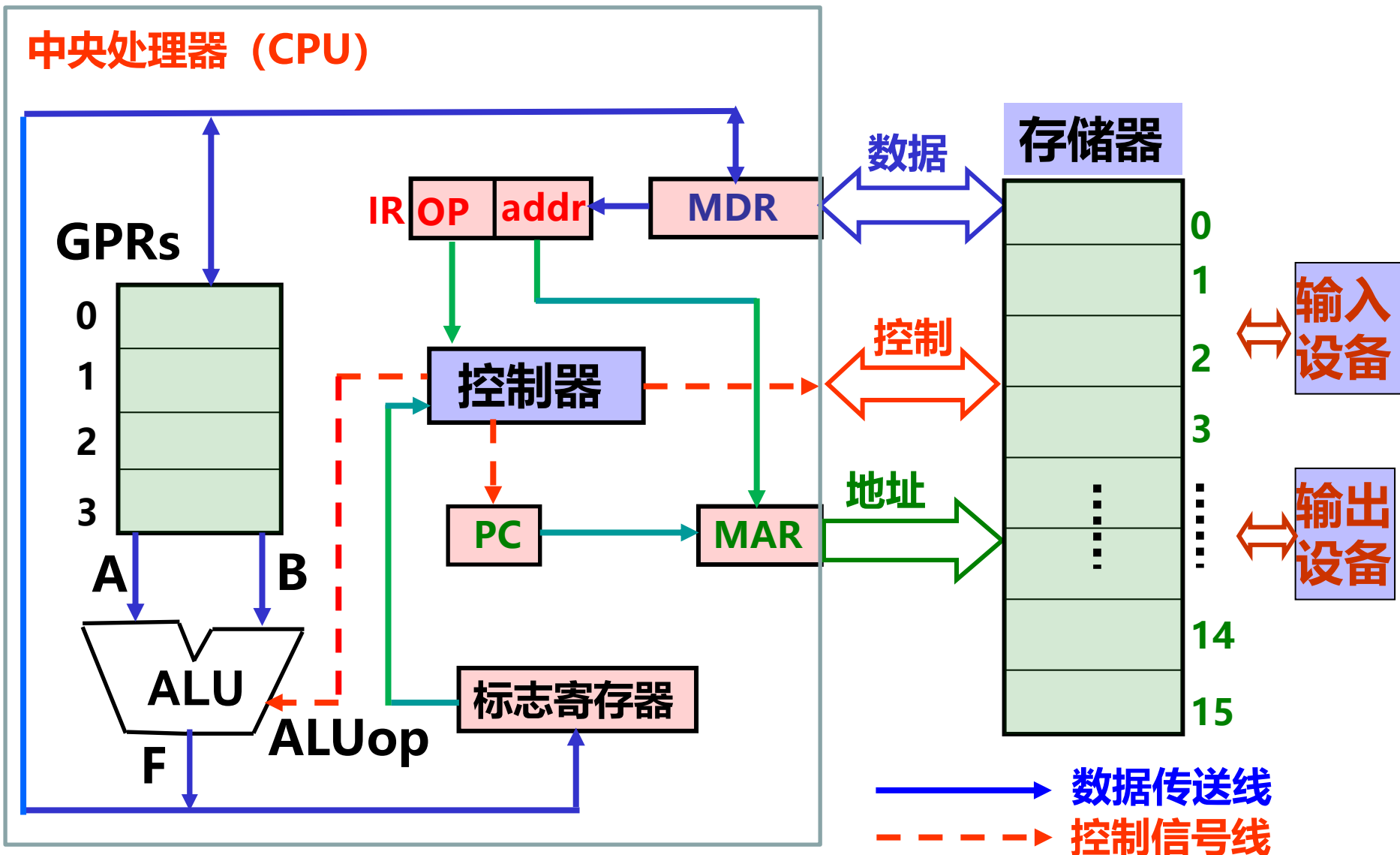
# Software

---

- **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, ...

# 程序和指令执行过程举例

8位模型机M：8位定长指令字，4个GPR，16个主存单元



# 程序和指令执行过程举例

假设模型机M中8位指令，格式有两种：R型、M型

格式	4位	2位	2位	功能说明
<b>R型</b>	op	rt	rs	$R[rt] \leftarrow R[rt] \text{ op } R[rs]$ 或 $R[rt] \leftarrow R[rs]$
<b>M型</b>	op	addr		$R[0] \leftarrow M[addr]$ 或 $M[addr] \leftarrow R[0]$

rs和rt为通用寄存器编号；addr为主存单元地址

**R型**: op=0000, 寄存器间传送 (mov) ; op=0001, 加 (add)

**M型**: op=1110, 取数 (load) ; op=1111, 存数 (store)

问题：指令 1110 0111的功能是什么？

答：因为op=1110，故是M型load指令，功能为：

$R[0] \leftarrow M[0111]$ ，即：将主存地址0111（7号单元）中的8位数据装入到0号寄存器中。

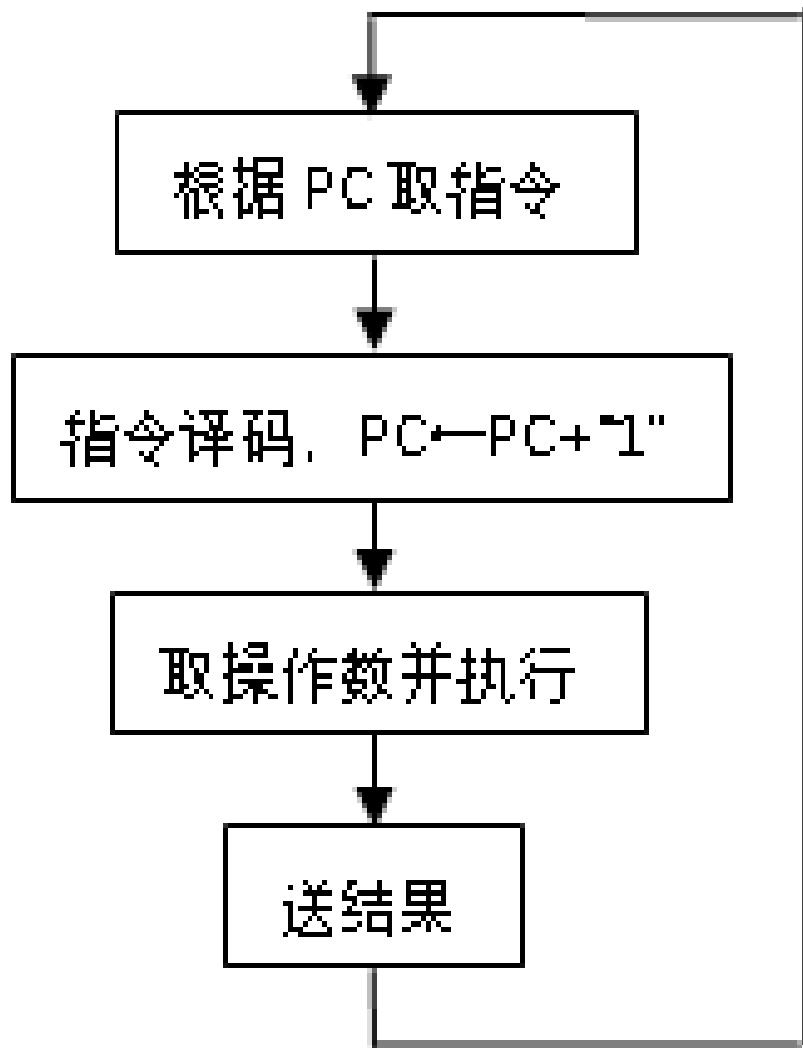
# 程序和指令执行过程举例

若在M上实现“ $z=x+y$ ”，x和y分别存放在主存5和6号单元中，结果z存放在7号单元中，则程序在主存单元中的初始内容为：

主存地址	主存单元内容	内容说明 (li表示第i条指令)	指令的符号表示
0	1110 0110	I1: $R[0] \leftarrow M[6]$ ; op=1110: 取数操作	load r0, 6#
1	0000 0100	I2: $R[1] \leftarrow R[0]$ ; op=0000: 传送操作	mov r1, r0
2	1110 0101	I3: $R[0] \leftarrow M[5]$ ; op=1110: 取数操作	load r0, 5#
3	0001 0001	I4: $R[0] \leftarrow R[0] + R[1]$ ; op=0001: 加操作	add r0, r1
4	1111 0111	I5: $M[7] \leftarrow R[0]$ ; op=1111: 存数操作	store 7#, r0
5	0001 0000	操作数x, 值为16	程序执行过程及其结果是什么?
6	0010 0001	操作数y, 值为33	
7	0000 0000	结果z, 初始值为0	



# 程序和指令执行过程举例



程序执行过程

## 指令I1 (PC=0) 的执行过程

	I1: 1110 0110
取指令	IR ← M[0000]
指令译码	op = 1110, 取数
PC增量	PC ← 0000 + 1
取数并执行	MDR ← M[0110]
送结果	R[0] ← MDR
执行结果	R[0] = 33

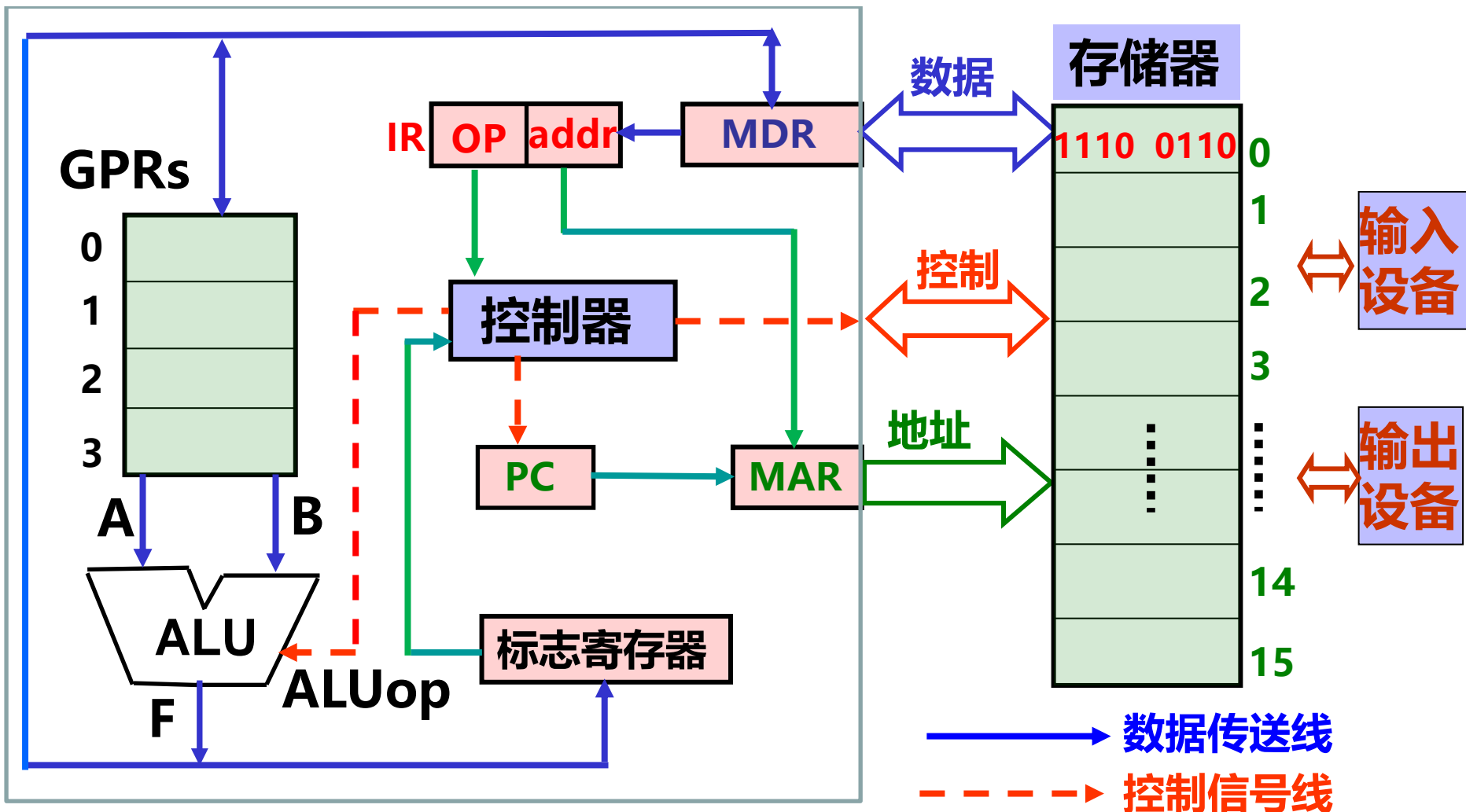
随后执行PC=1中的指令I2

# 程序和指令执行过程举例

指令 1110 0110 功能为  $R[0] \leftarrow M[0110]$ ，指令执行过程如下：

取指  $IR \leftarrow M[PC]$ ：  $MAR \leftarrow PC$ ；控制线  $\leftarrow$  Read；  $IR \leftarrow MDR$

取数  $R[0] \leftarrow M[addr]$ ：  $MAR \leftarrow addr$ ；控制线  $\leftarrow$  Read；  $R[0] \leftarrow MDR$



# 程序和指令执行过程举例

	I2: 0000 0100	I3: 1110 0101
取指令	$IR \leftarrow M[0001]$	$IR \leftarrow M[0010]$
指令译码	op=0000, 传送	op=1110, 取数
PC增量	$PC \leftarrow 0001 + 1$	$PC \leftarrow 0010 + 1$
取数并执行	$A \leftarrow R[0]$ 、mov	$MDR \leftarrow M[0101]$
送结果	$R[1] \leftarrow F$	$R[0] \leftarrow MDR$
执行结果	<b><math>R[1] = 33</math></b>	<b><math>R[0] = 16</math></b>

# 程序和指令执行过程举例

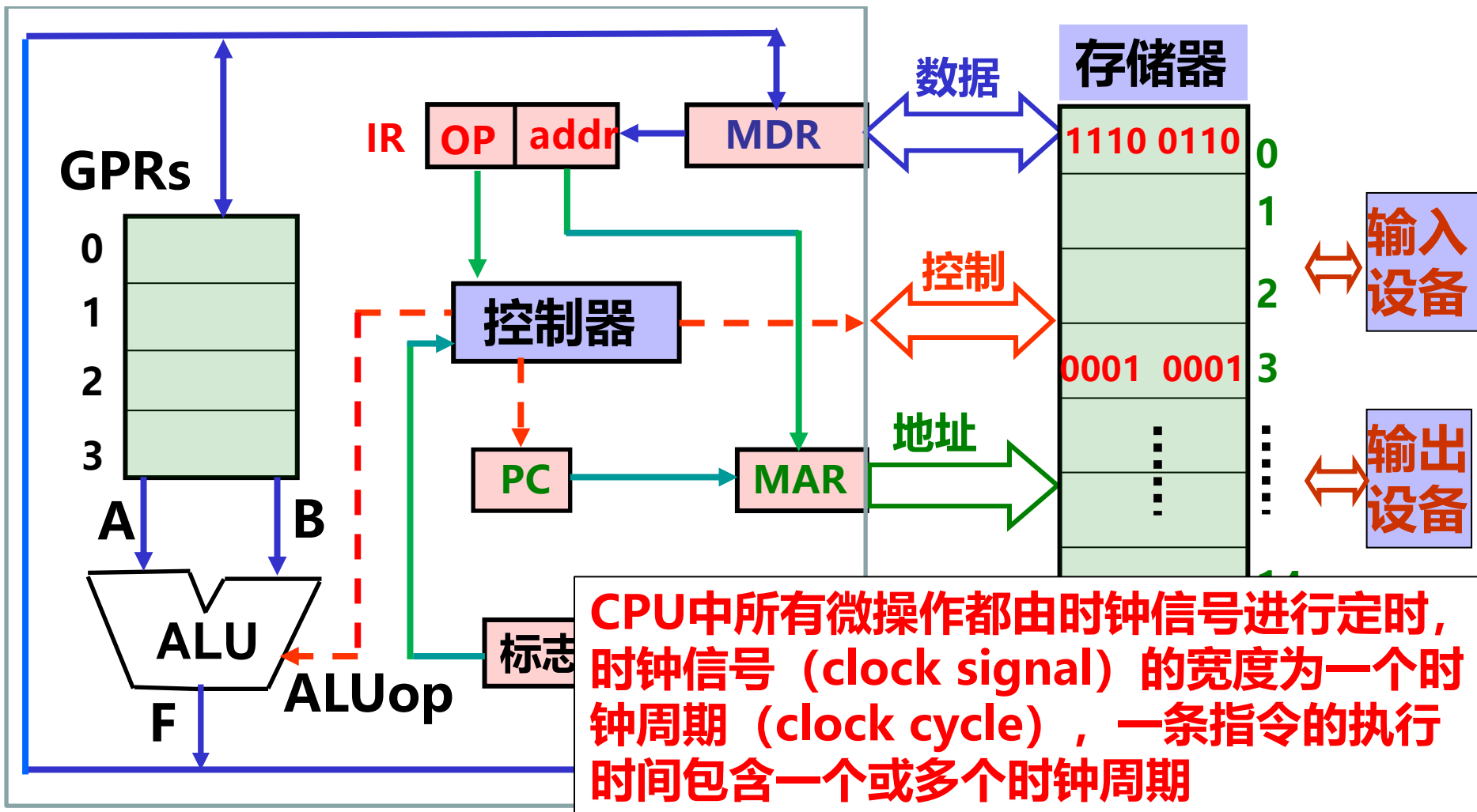
	I4: 0001 0001	I5: 1111 0111
取指令	$IR \leftarrow M[0011]$	$IR \leftarrow M[0100]$
指令译码	op=0001, 加	op=1111, 存数
PC增量	$PC \leftarrow 0011 + 1$	$PC \leftarrow 0100 + 1$
取数并执行	$A \leftarrow R[0]$ 、 $B \leftarrow R[1]$ 、add	$MDR \leftarrow R[0]$
送结果	$R[0] \leftarrow F$	$M[0111] \leftarrow MDR$
执行结果	$R[0] = 16 + 33 = 49$	$M[7] = 49$

# 程序和指令执行过程举例

指令 0001 0001 功能为  $R[0] \leftarrow R[0] + R[1]$ ，指令执行过程如下

**ALU运算  $R[0] \leftarrow R[0] + R[1]$  的微操作（在控制信号的控制下完成）：**

$A \leftarrow R[0]$ ;  $B \leftarrow R[1]$ ;  $ALUop \leftarrow add$ ;  $R[0] \leftarrow F$



# 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");
}
```

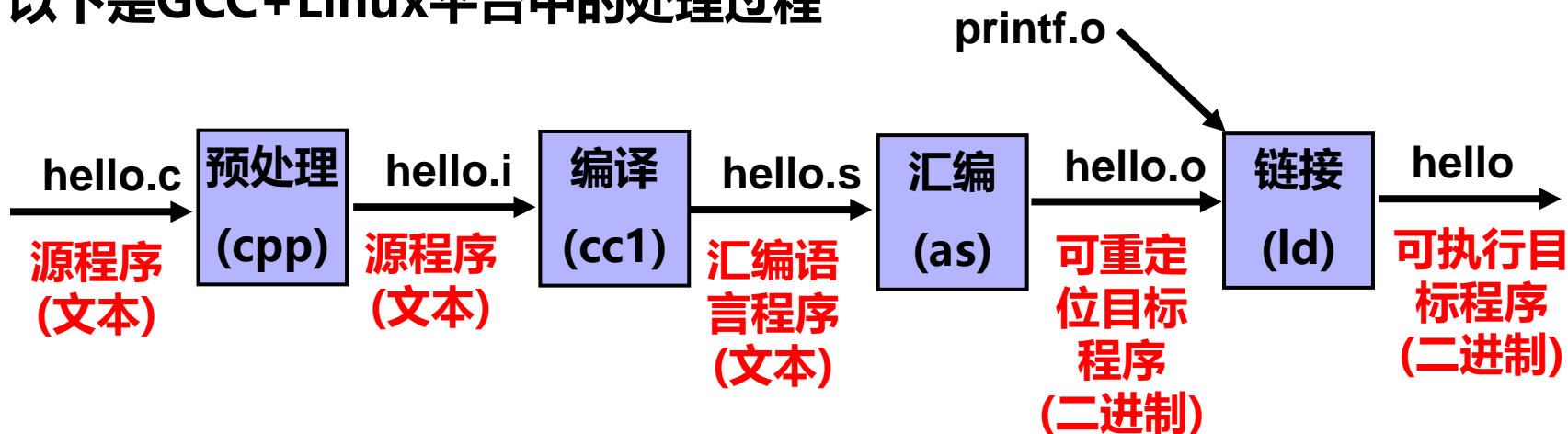
hello.c的ASCII文本表示

```
# 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
```

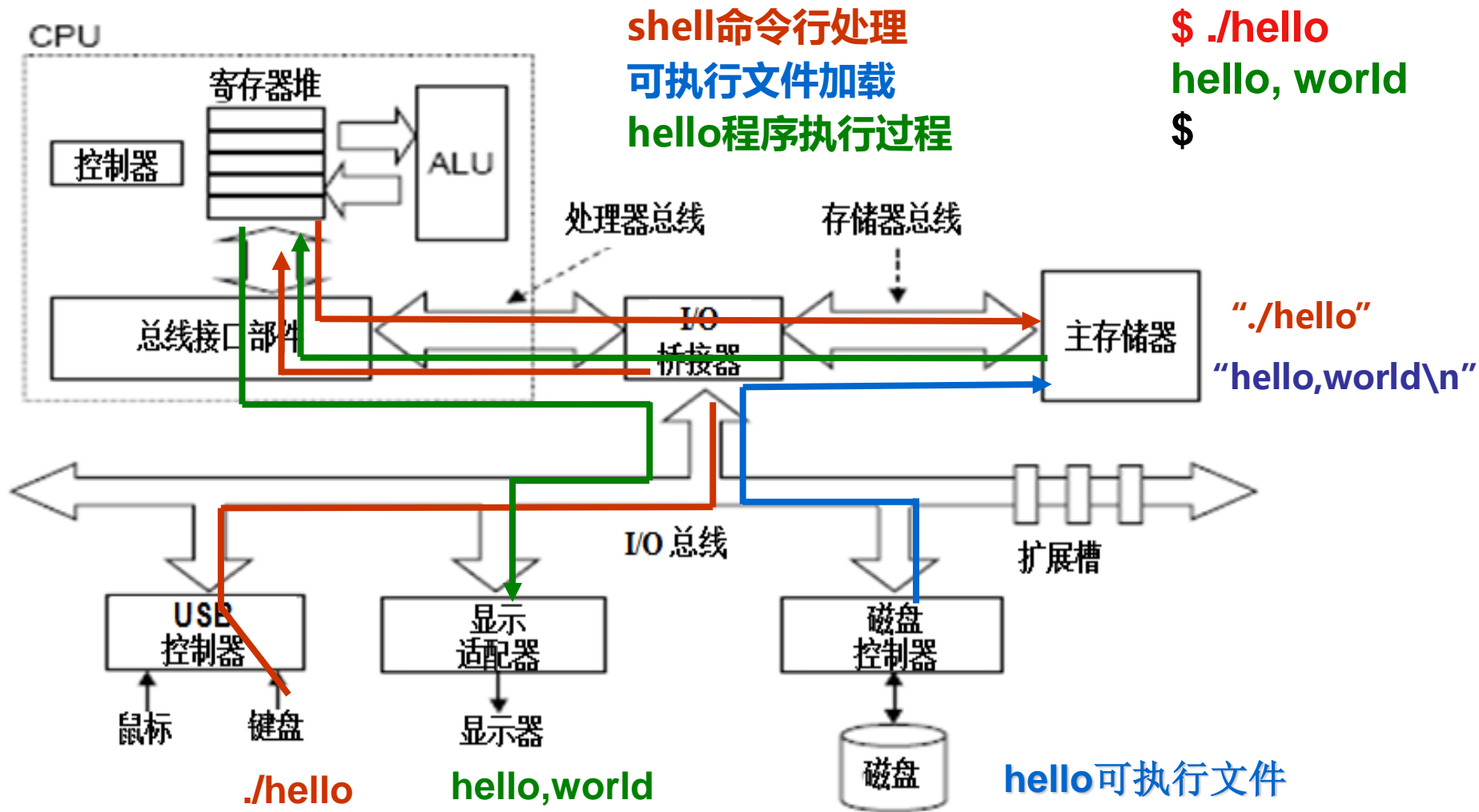
功能：输出 “hello,world”

计算机不能直接执行hello.c!

以下是GCC+Linux平台中的处理过程



# Hello程序的数据流动过程



数据经常在各存储部件间传送，故现代计算机大多采用“缓存”技术！

所有过程都是在CPU执行指令所产生的控制信号的作用下进行的。

# 主要内容

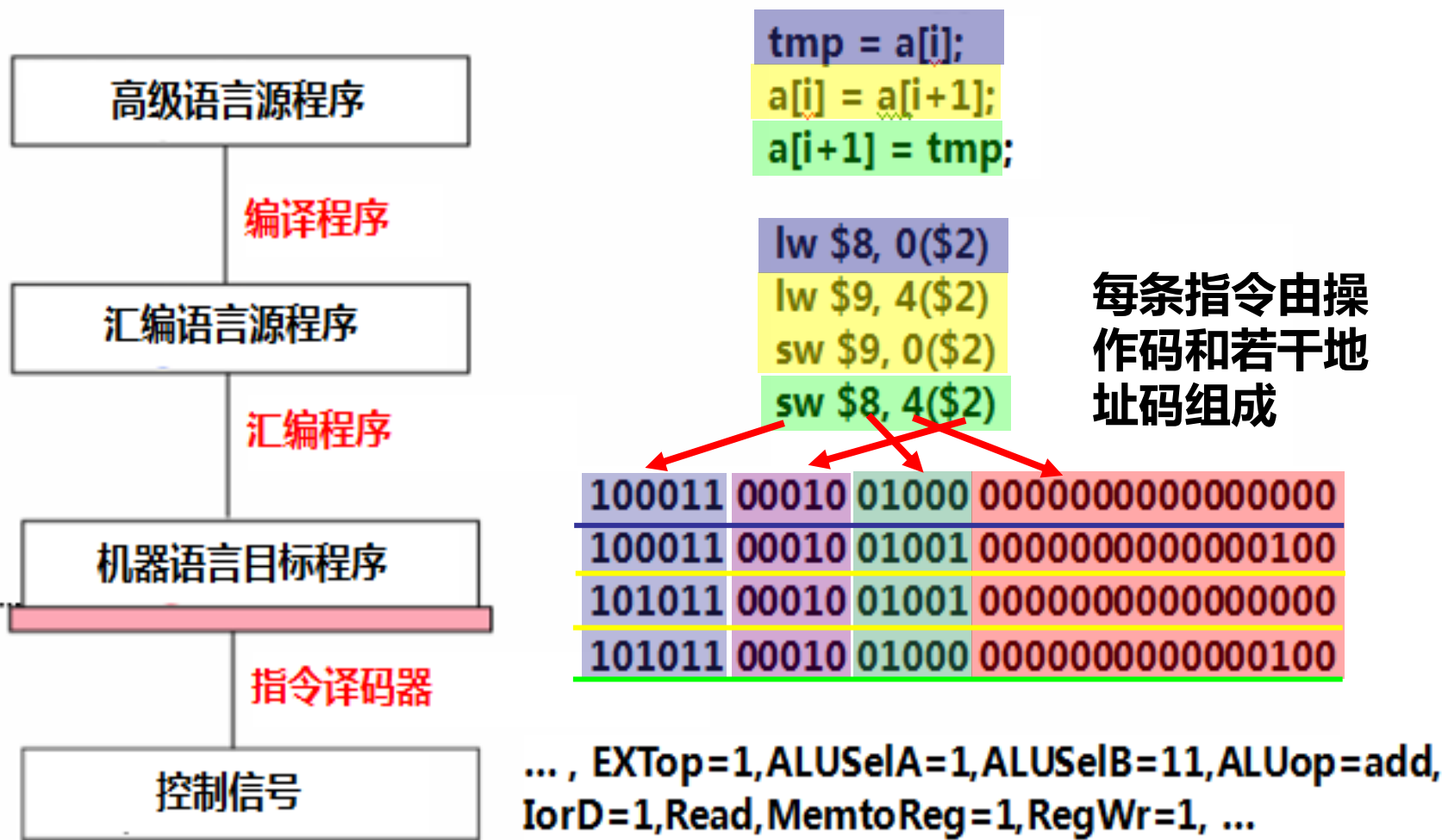
---

- 课程的由来
- 课程内容概要
- 课程教学安排及考试安排
- 硬件和软件的基本组成
- 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



# 不同层次语言之间的等价转换

计算机软件  
计算机硬件



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什么支撑？

- 最早的程序开发很简单（怎样简单？）
  - 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（怎样的环境？）

- 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

- 后来用汇编语言编程

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）

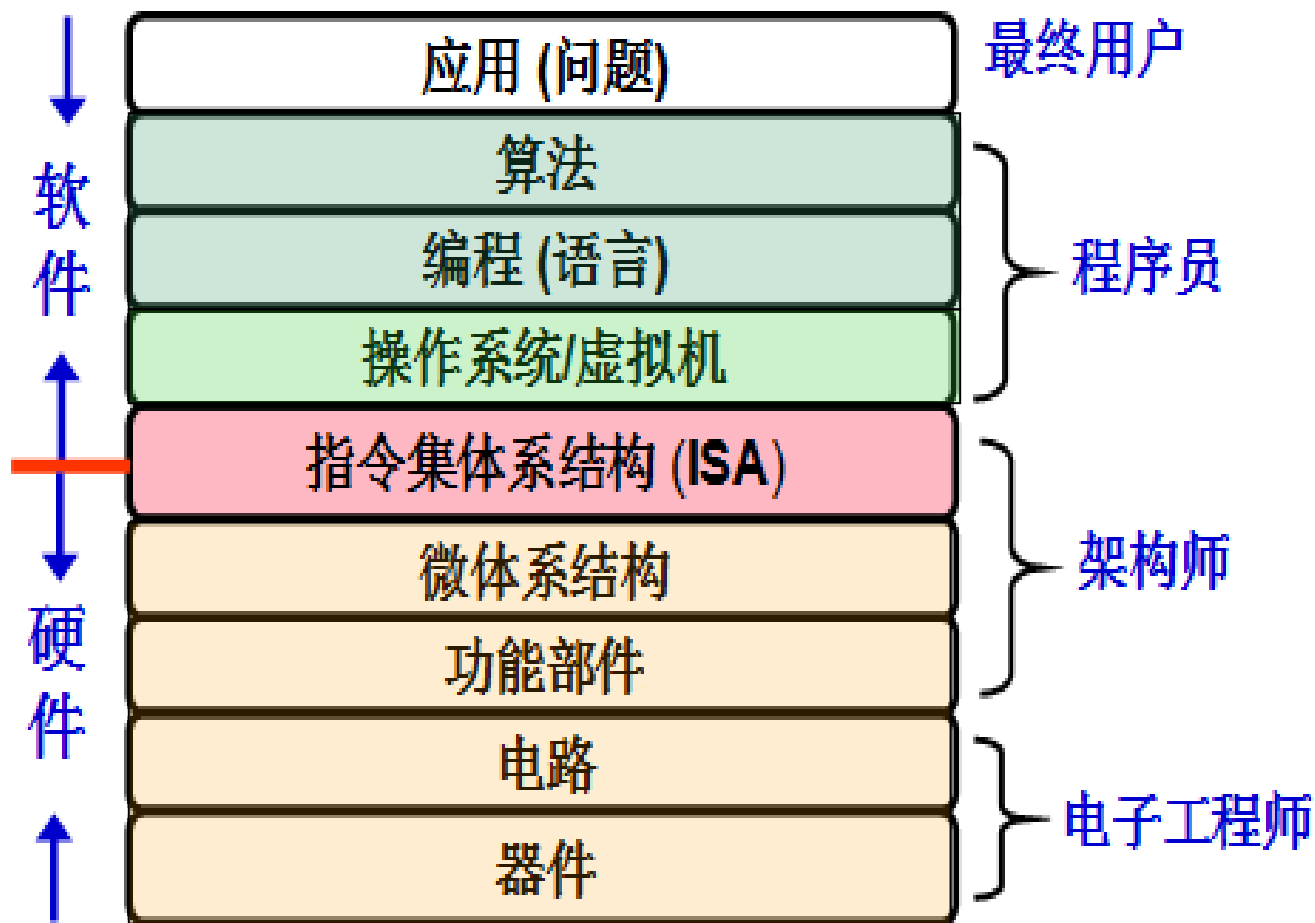
**操作系统**包括人机交互界面、提供服务功能的内核例程

# 计算机系统抽象层的转换

**程序执行结果**  
不仅取决于  
**算法、程序编写**  
而且取决于  
**语言处理系统**  
**操作系统**  
**ISA**  
**微体系结构**

**不同计算机课程**  
**处于不同层次**  
**必须将各层次关**  
**联起来解决问题**

**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！**



**最高层抽象就是点点鼠标、拖拖图标、敲敲键盘，但这背后有多少层转化啊！**

# 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

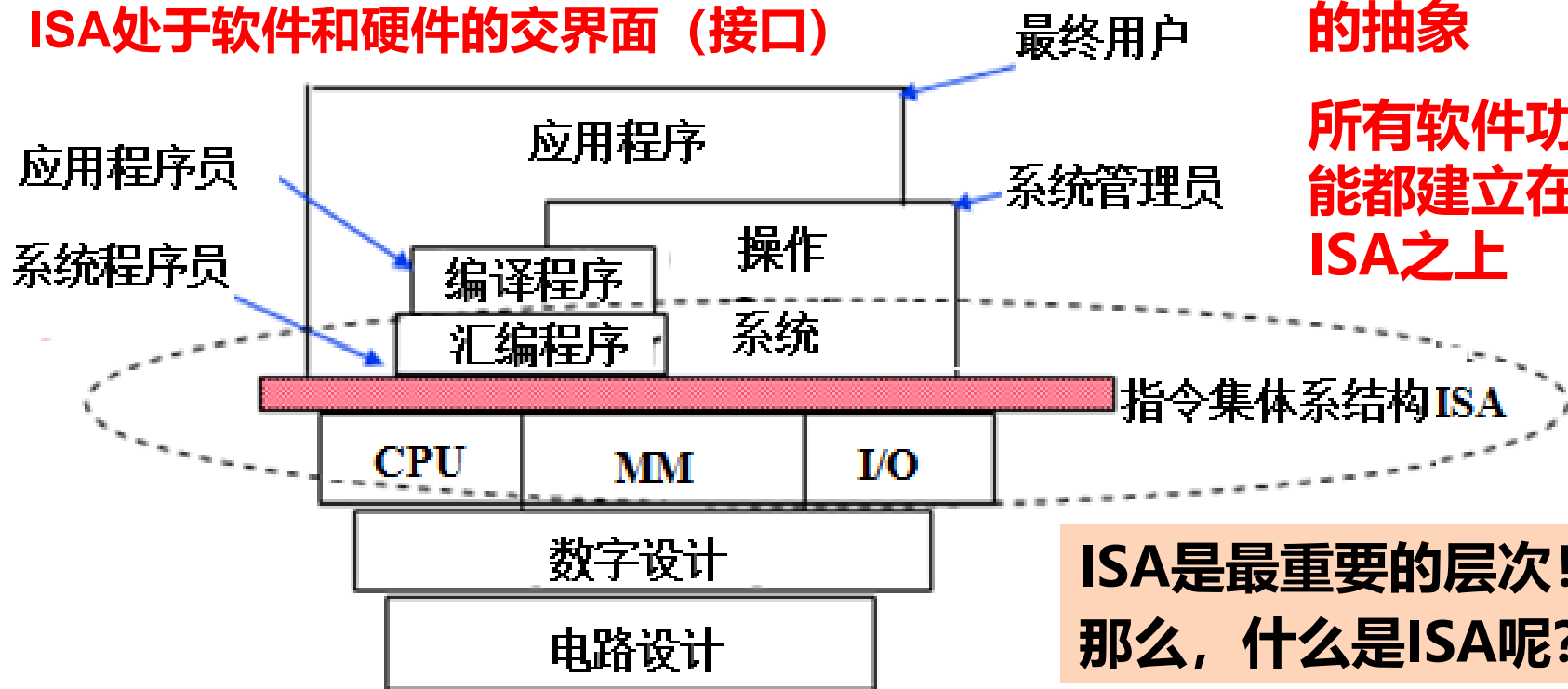
**编译器和汇编器的目标程序**由机器级代码组成

**操作系统通过指令**直接对硬件进行编程控制

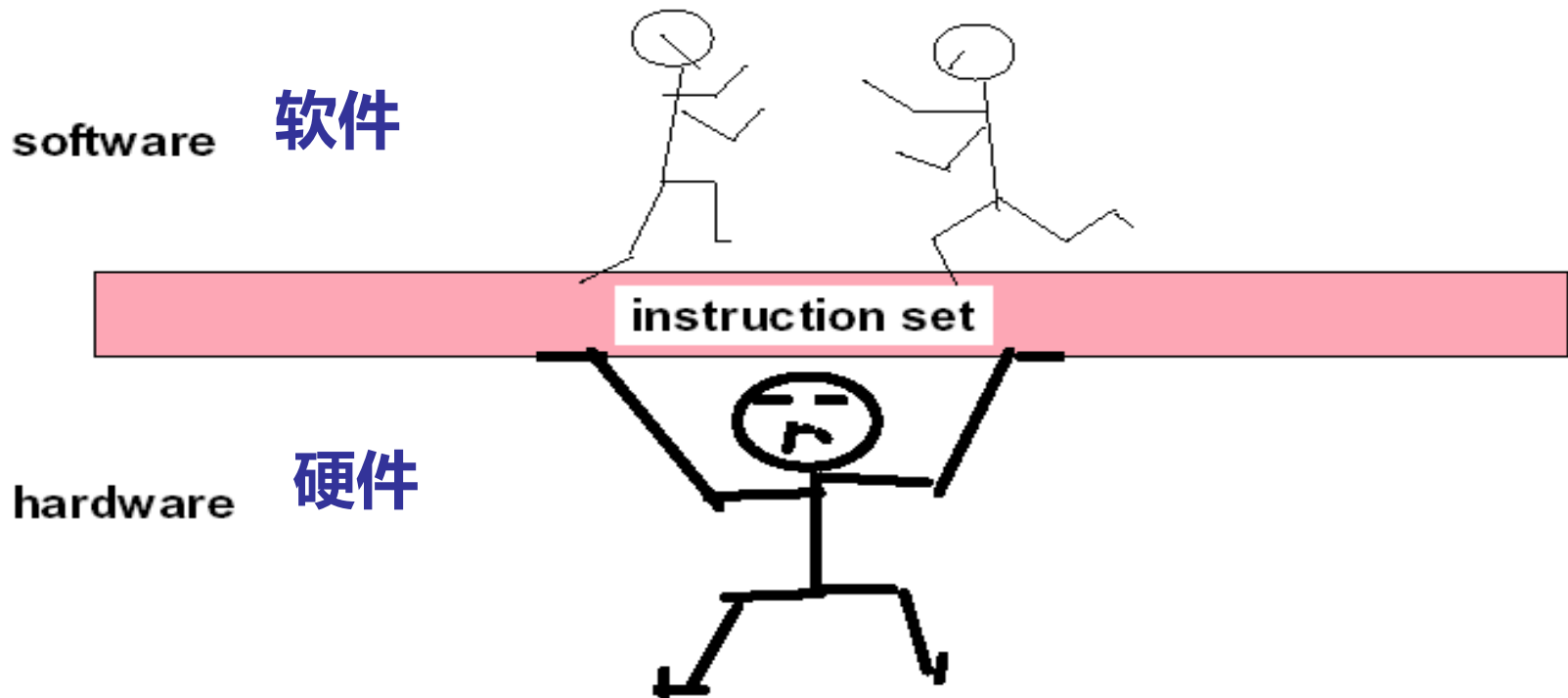
**ISA处于软件和硬件的交界面（接口）**

**ISA是对硬件的抽象**

**所有软件功能都建立在ISA之上**



# 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）

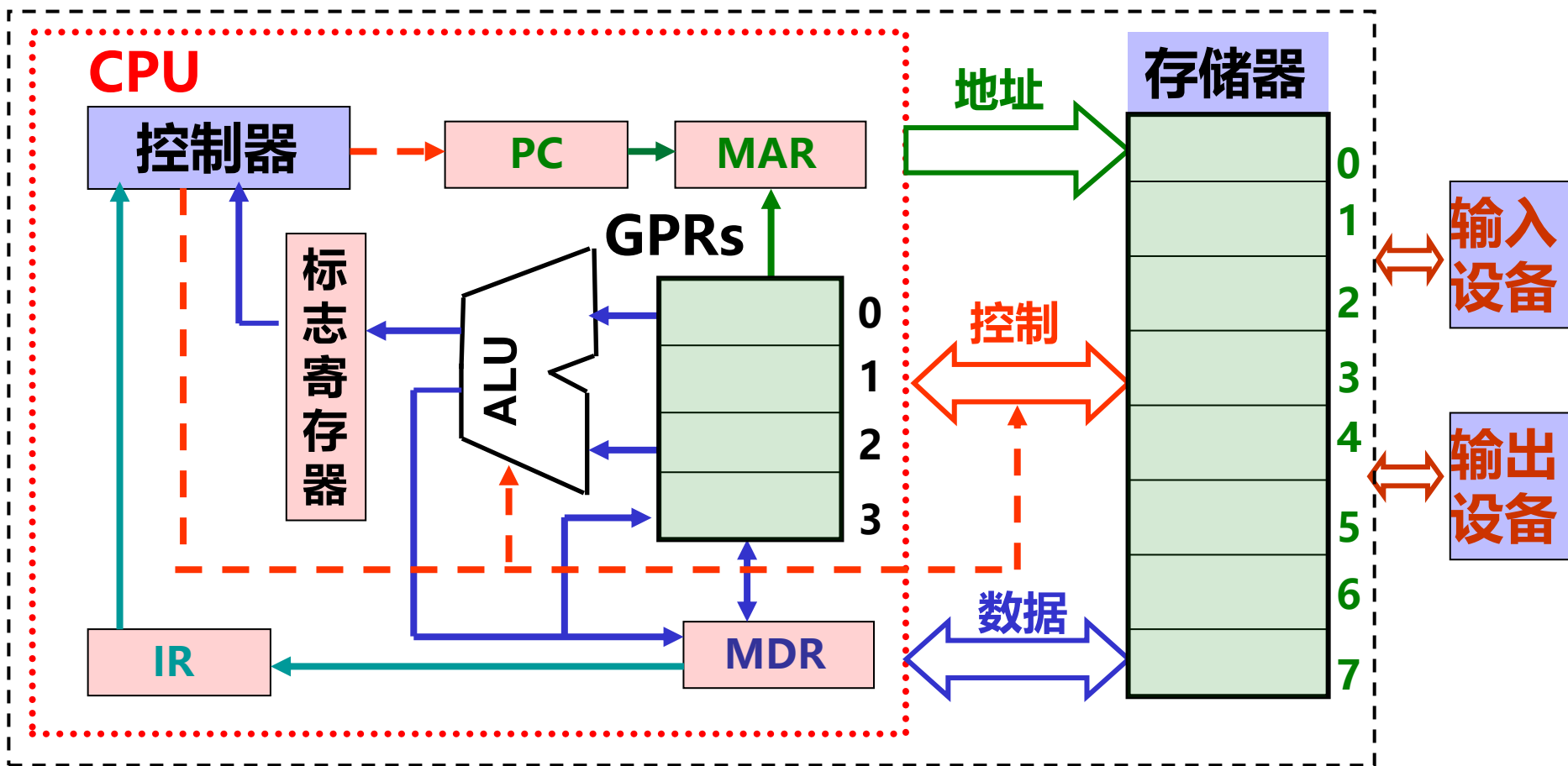
- ISA指Instruction Set Architecture，即指令集体系结构
- ISA是一种规约（Specification），它规定了**如何使用硬件**
  - 可执行的指令的集合，包括**指令格式、操作种类以及每种操作对应的操作数的相应规定**；
  - 指令可以接受的**操作数的类型**；
  - 操作数所能存放的寄存器组的结构，包括每个**寄存器的名称、编号、长度和用途**；
  - 操作数所能存放的**存储空间的大小和编址方式**；
  - 操作数在存储空间存放时按照**大端还是小端方式存放**；
  - 指令获取操作数的方式，即**寻址方式**；
  - 指令执行过程的控制方式，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？**



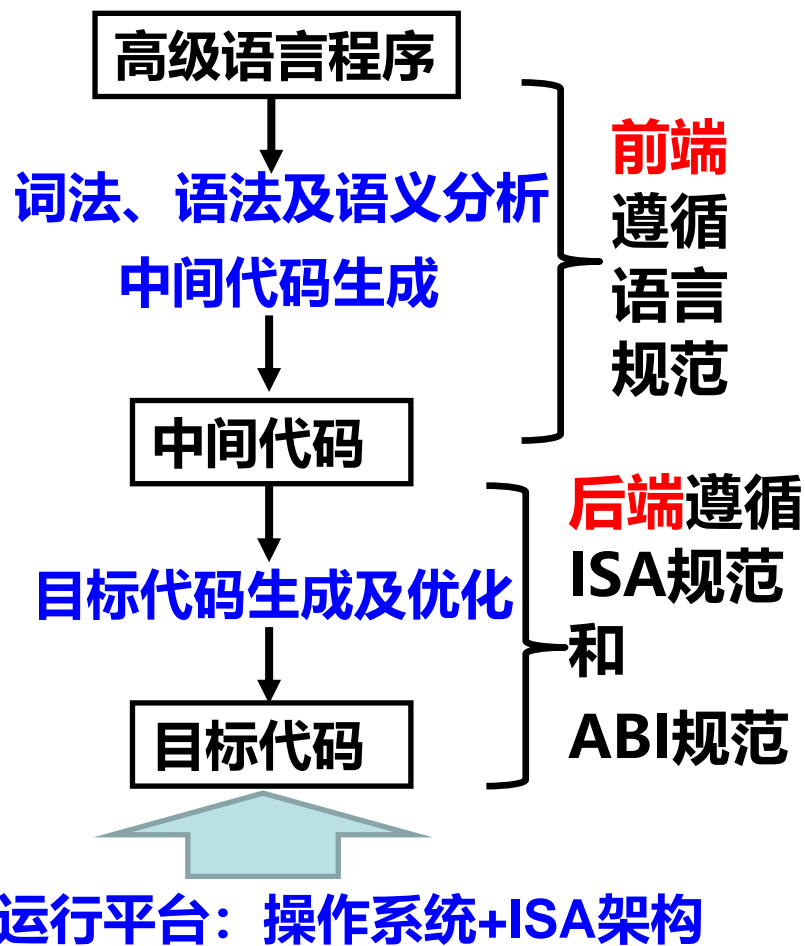
# ISA和计算机组成（微结构）之间的关系



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：

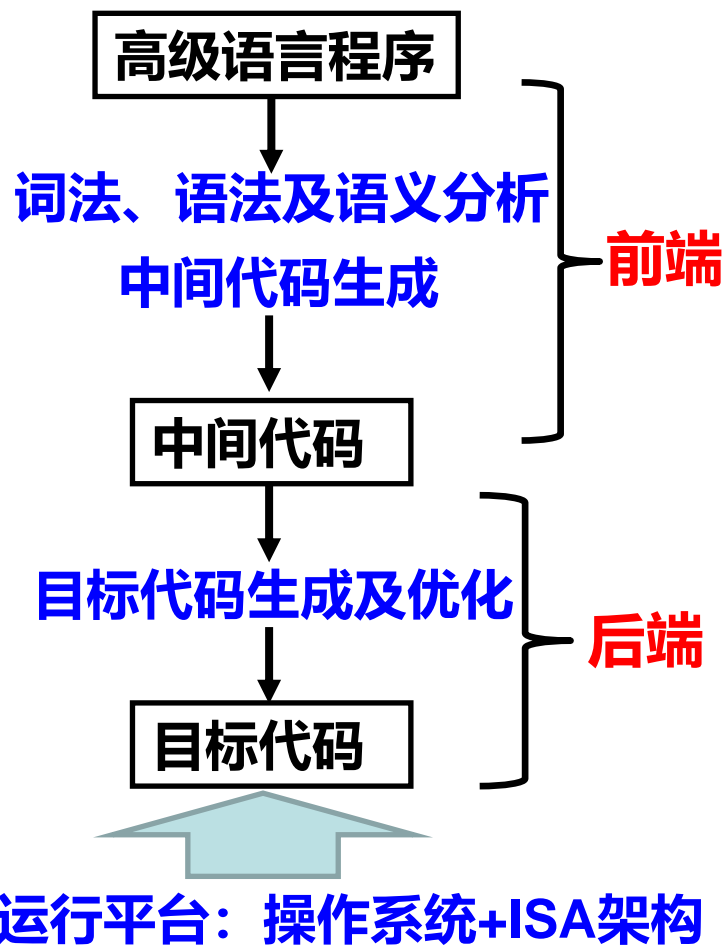
- (1) 程序员不了解语言规范；
- (2) 程序含有未定义行为（undefined behavior）或未确定行为（unspecified behavior）的语句

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规定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

等

本课程所用平台为**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。

本课程大多在讲解ABI和ISA规范！要了解程序的确切行为，最好的方法就是查手册（用于给出规范）！

# 主要内容

---

- 课程的由来
- 课程内容概要
- 课程教学安排及考试安排
- 硬件和软件的基本组成
- 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

# 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)

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

- 基本的性能评价标准是：CPU的执行时间

“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

---

## CPI: Cycles Per Instruction

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

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

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

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而言，其CPI是一个确定的值。但是，对于某一个程序或一台机器而言，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：

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

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：

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

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）

$$\begin{aligned}\text{MIPS} &= \text{Instruction Count} / \text{Time} \times 10^6 \\ &= \text{Clock Rate} / \text{CPI} \times 10^6\end{aligned}$$

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

因为每条指令执行时间不同，所以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？

所以MIPS数不能说明性能的好坏（用下页中的例子来说明）

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

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

- 不一定是程序中最花时间的部分

用MFLOPS数表示  
性能也有一定局限！

# 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 NRCP	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。

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

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

或  $p = 1/(t/n + 1 - t)$

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。