

计算机系统基础

2019.4

王多强

群名称：ICS2019

群 号：1015148119



群名称：ICS2019

群 号：1015148119

加入本课堂的学习群



群名称: ICS2019

群 号: 1015148119

关于本课程

本门课“改编”自CMU的《深入理解计算机系统》，阐述计算机系统的基本工作原理。

- C语言、汇编语言：使用程序设计语言编写程序
- 编译：如何将高级语言程序翻译成机器语言程序
- 操作系统：资源管理与作业调度，加载程序并执行
- 计算机组成原理：计算机硬件的组织结构和工作机制
- 数字逻辑：数字电路的设计
-

这些课程都从某个专业方向“深入”地介绍了计算机的重要概念、方法、技术，但**缺少关联性**。

计算机系统是一个有机的整体，软硬件协同才能完成计算任务。而计算机的主要工作是执行程序，那么程序在计算机系统中是如何被执行起来的呢？

- ◆ 程序执行：被加载到内存中的程序放在哪里？程序的地址空间是如何管理的？
- ◆ C程序设计：函数调用是怎么回事？返回地址是什么？返回地址、现场被保存到哪里？参数、返回值是怎么传递？
- ◆ 数据表示：为什么有些浮点数不能精确表示，浮点数的数据表示误差是怎么回事？会产生什么影响？
- ◆

这些问题涉及到计算机系统中不同层面的概念和方法，在以前的课程里并没有很好地关联起来并讲好、讲透。

浮点数表示误差的真实事件：

1991年2月25日，海湾战争中，美国在沙特阿拉伯达摩地区设置的爱国者导弹拦截伊拉克的飞毛腿导弹失败，致使飞毛腿导弹集中美军军营，28名士兵阵亡。

原因：由爱国者导弹系统时钟内的一个软件错误造成。

根本原因是浮点精度问题：

爱国者导弹系统中有一个内置时钟，用计数器实现，理论上每隔0.1秒计数一次。但0.1的二进制表示是一个无限循环序列：

0.00011[0011]...

程序中用24位定点二进制小数x来表示0.1，这样0.1的理论值与x的实际值之间有一个微小的差异：

$$\begin{aligned} & 0.1 - 0.000\ 1100\ 1100\ 1100\ 1100\ 1100\text{B} \\ &= 0.00011[0011]... - 0.000\ 1100\ 1100\ 1100\ 1100\ 1100\text{B} \\ &= 2^{-20} \times 0.1 \approx 9.54 \times 10^{-8} \end{aligned}$$

而在爱国者导弹准备拦截飞毛腿导弹之前，已经连续工作了100小时，相当于计数了 $100 \times 60 \times 60 \times 10 = 36 \times 10^5$ 次。

故，导致的时钟偏差为： $9.54 \times 10^{-8} \times 36 \times 10^5 \approx 0.343$ 秒。

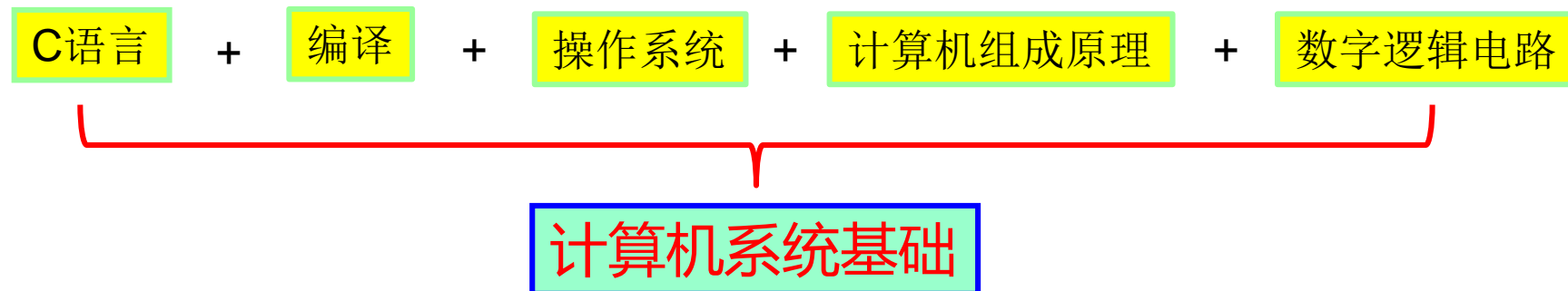
当时飞毛腿的速度大约为2000米/秒，则由于时钟计算误差而导致的距离误差是多少？

$$2000 \times 0.343 \text{秒} \approx 687 \text{米}$$

所以，正是由于这个时钟误差，尽管雷达系统已经侦测到飞毛腿导弹，但爱国者导弹却找不到实际上正在来袭的飞毛腿导弹。致使起初的目标发现被视为一次假警报，而导致拦截失败。

要深入理解计算机系统的工作原理，才能避免类似问题的发生。

本课程将从程序的角度来看计算机内部的工作机制，研究程序是如何在计算机内部被表示和运行起来的。



本门课本质上是一门导论课，涉及很多专门知识。本课程中对这些专门知识的细节不做深入研究，而是在阐述清楚计算机系统中程序运作机制的基础上，引出这些专门知识方面的问题，留待同学们在以后的课程里专门解决。

课程基本信息

- **课程名称**
 - 计算机系统基础 (Introduction to Computer Systems)
- **课程参考网站 (南京大学)**
 - http://cslab.nju.edu.cn/ics/index.php/lcs:Main_page
- **先导课程**
 - C语言程序设计、 (数字逻辑电路, 不是必须的)
- **教材:**
 - 《计算机系统基础》, 袁春风, 机械工业出版社
- **主要参考书:**
 - 《深入理解计算机系统》 (第2版), Randal E. Bryant, david R. O' Hallaron著, 龚奕利, 雷迎春译, 机械工业出版社, 2011 年
 - Brian W. Kernighan, Dennis M. Ritchie, The C Programming Language (second Edition), 北京: 机械工业出版社, 2006
 - 《计算机系统概论》 (原书第2版), Yale N. Patt, Sanjay J. Patel著, 梁阿磊, 蒋兴昌, 林凌译, 机械工业出版社, 2007年



- **学习方式**

- 课堂讲授：24学时
- 课程实验：16学时, Lab1~3

- **考核方式**

- 平时作业：20%
- Lab：20%
- 考试：50%
- 考勤：10%


教材章节：两大部分

- **第一部分 可执行目标文件的生成**

- 第1章 计算机系统概述
- 第2章 数据的机器级表示与处理
- 第3章 程序的转换及机器级表示
- 第4章 程序的链接

- **第二部分 可执行目标文件的运行**

- 第5章 程序的执行
- 第6章 层次结构存储系统
- 第7章 异常控制流
- 第8章 I/O操作的实现



因为课时关系，课程内容无法全部上完。没有讲的部分可以自学或带着问题在以后相关课程里继续学习。

第一章 计算机系统概述

1.1 计算机基本工作原理

计算：在人类文明发展史上一直具有重要的地位并极具现实意义

- ◆ 结绳计数、算盘、机械式计算器、电子计算器、计算机...

ABC：世界上第一台真正意义的电子数字计算机

- ◆ ABC: Atanasoff-Berry Computer (阿塔那索夫-贝瑞电脑)
- ◆ 1935~1939年由美国艾奥瓦州立大学物理系副教授约翰·文森特·阿塔那索夫和克里福特·贝瑞研制成功
- ◆ 国际计算机界公认**约翰·文森特·阿塔那索夫**被称为“电子计算机之父”。

ENIAC：世界上第一台真正实用的电子计算机

- ◆ ENIAC：Electronic Numerical Integrator And Computer，电子数字积分计算机
- ◆ 1946年由美国宾夕法尼亚大学莫齐利、艾克特研制
- ◆ 电子真空管计算机，5000次加法/s，50次乘法/s，用十进制表示信息并运算，采用手动编程——并不是真正意义上的现代电子计算机
- ◆ 用于解决复杂的弹道计算问题，由20分钟缩短到30秒，速度提高40倍。

1.1.1 冯·诺依曼结构基本思想

- 冯·诺伊曼原籍匈牙利，数学家，被称为“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，涉及到极为困难的计算。
- 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，发表了全新的“**存储程序通用电子计算机方案**”。该报告中提出的计算机结构被称为**冯·诺依曼结构**。至此，**标志着现代计算机结构思想的诞生**。



**Electronic
Discrete
Variable
Automatic
Computer**
离散变量自动电子计算机

现代计算机的原型

1946年，普林斯顿高等研究院（the Institute for Advanced Study at Princeton, IAS）开始按照冯·诺依曼的设计实现“存储程序”式计算机，被称为**IAS计算机**。

◆ 但这台计算机直到1951年才完成，使得它并不是世界上第一台存储程序计算机。

世界上第一台存储程序计算机是1949年由英国剑桥大学完成的**EDSAC**。

“存储程序” 工作方式

冯·诺依曼结构最重要的思想是 “存储程序” 工作方式 (Stored-program), 其基本思想是:

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后才能执行程序。一旦程序被启动执行, 计算机能在不需操作人员干预下自动完成逐条指令取出和执行的任務。

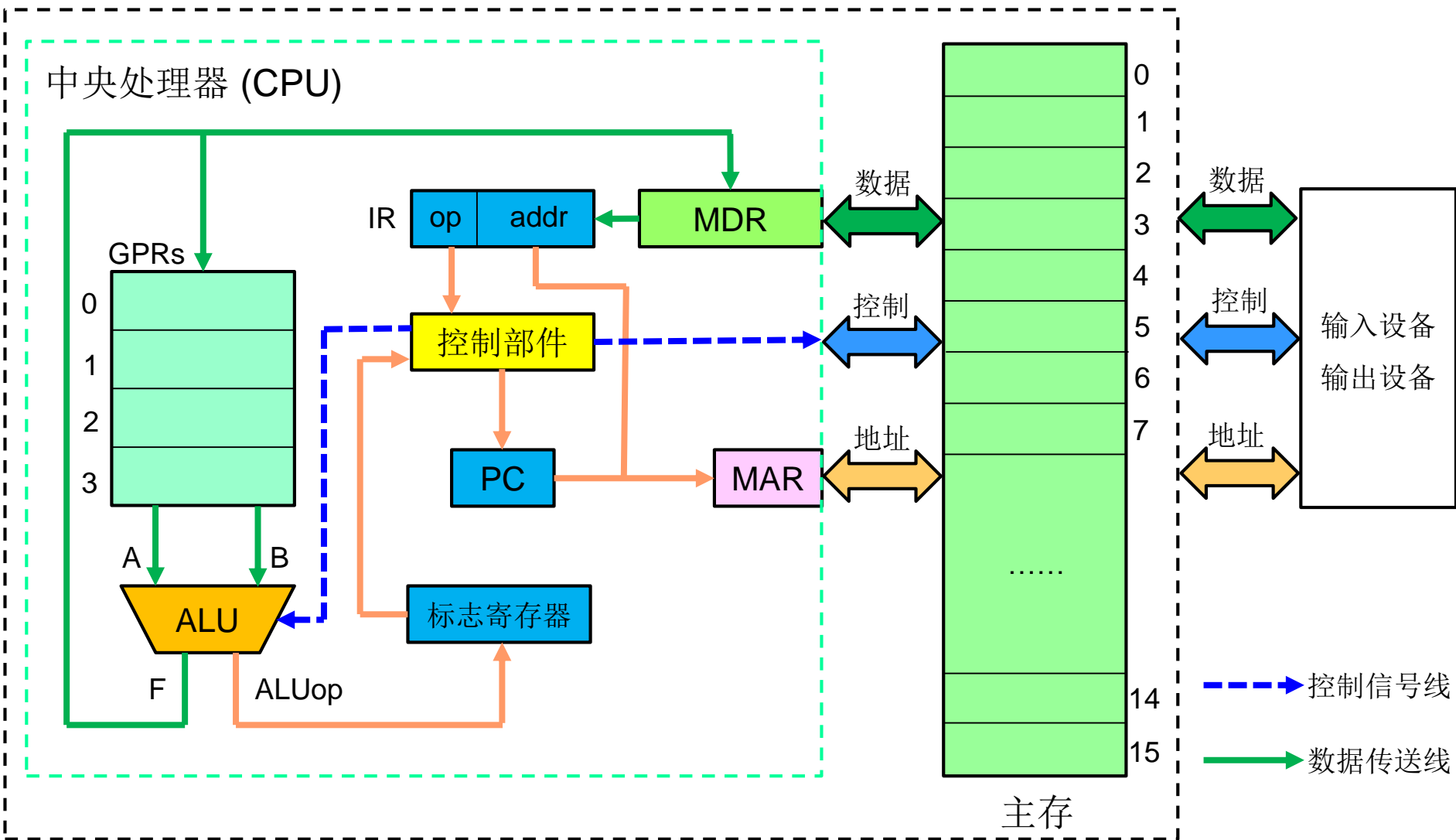
- 冯·诺依曼结构计算机也称为冯·诺依曼机器 (Von Neumann Machine)。
- 现代几乎所有的通用计算机都采用冯·诺依曼结构。

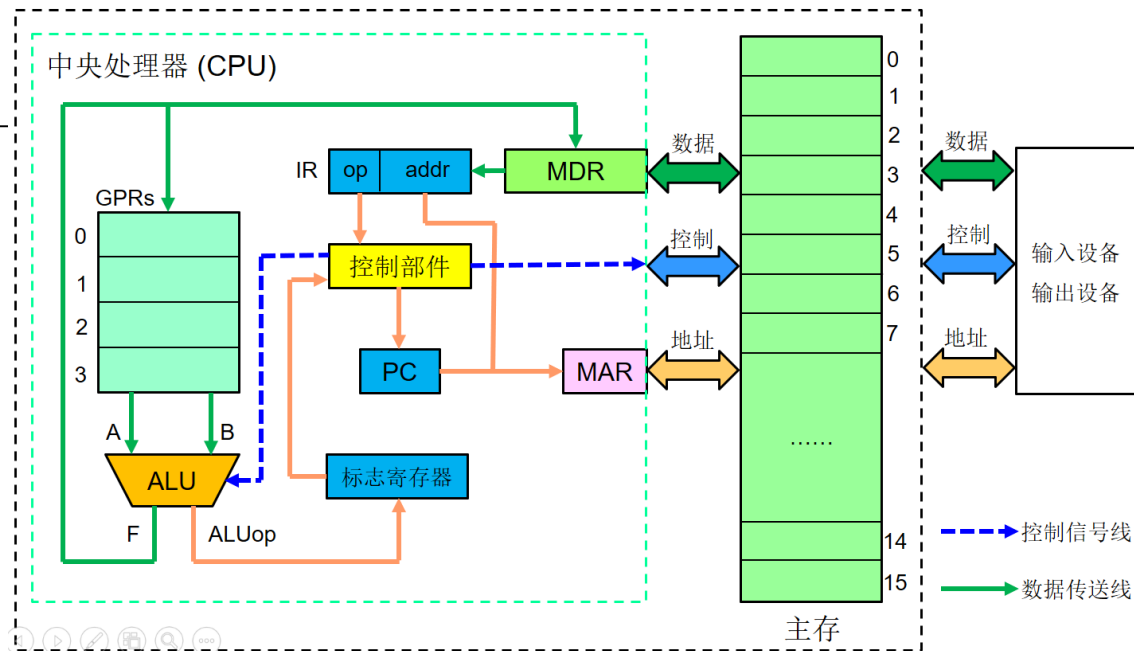
冯·诺依曼结构的基本思想包括以下几个方面：

1. 计算机由**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. **存储器**不仅能存放数据，也能存放指令，两者在形式上没有区别，但计算机应能区分它们；**控制器**应能自动取出指令来执行；**运算器**应能进行加减乘除基本算术运算及逻辑运算和其它附加运算；操作人员可以通过**输入设备**、**输出设备**和主机进行通信。
3. 计算机内部以**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。程序由一串指令组成。
4. 计算机采用“**存储程序**”式工作方式。

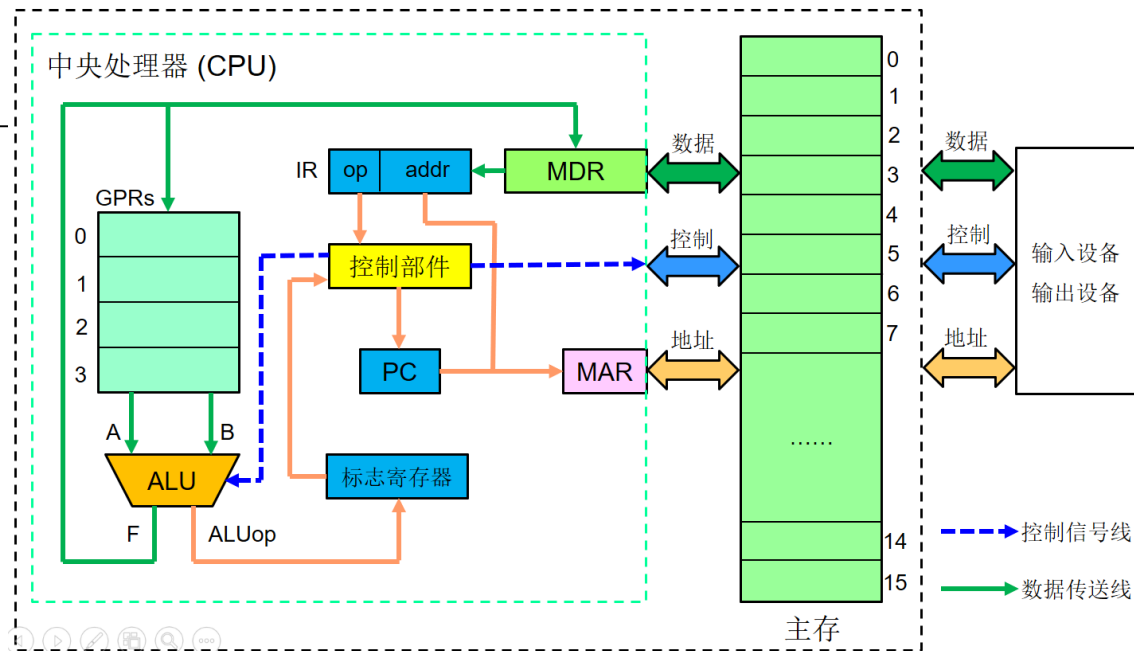
1.1.2 冯·诺依曼机的基本结构

根据冯·诺依曼结构的基本思想，一个模型机的基本硬件结构如图所示。

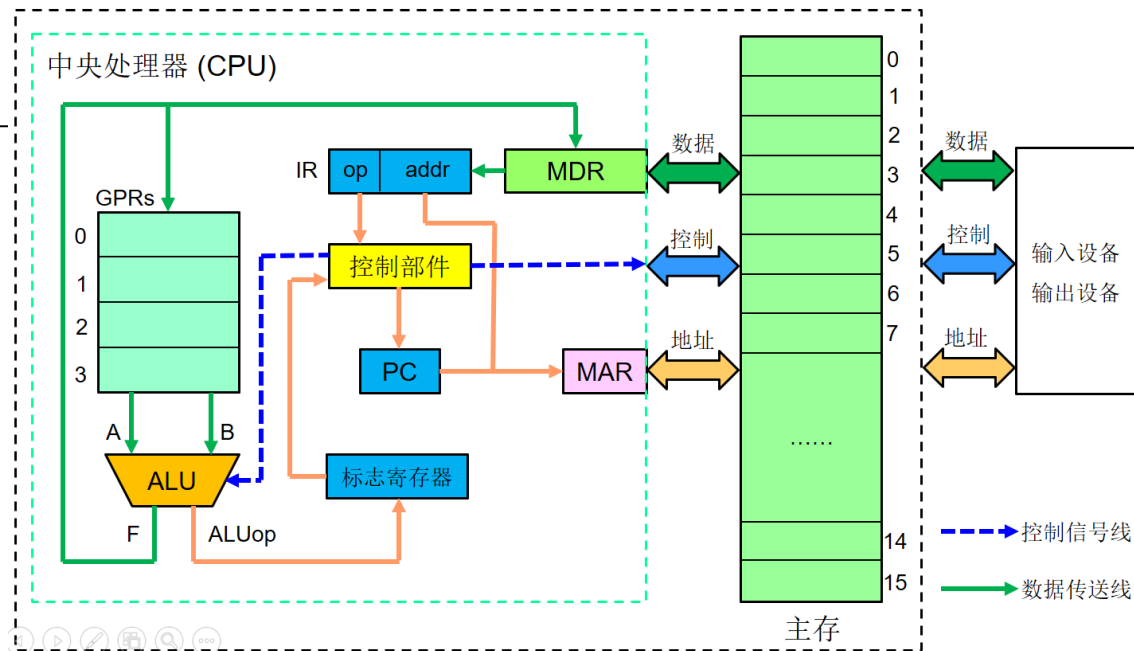




- **主存储器:** 简称主存/内存，用于存放指令和数据；
主存中每个单元都有编号，称为**主存单元地址**，简称**内存地址**
- **ALU:** 运算器，算术逻辑部件 (Arithmetic Logic Unit, ALU)，用来进行算术逻辑运算；
ALUop: ALU操作控制信号；
- **控制器:** 自动逐条取出指令并进行译码 (Control Unit, CU)；
- **输入设备、输出设备:** 与外界交互。



- **通用寄存器:** 临时存放从主存取来的数据或运算的结果；
每个寄存器都有一个编号；
- **标志寄存器:** 用于存放ALU运算后产生的标志信息；
如结果是否为0（ZF）、是否为负（SF）等；
- **指令寄存器（IR）:** 用于临时存放从主存取来的指令；
- **程序计数器（PC）:** 存放下一条将要被执行的指令的地址；
- **控制器、运算器、寄存器构成了CPU的主要部分**



总线： 取指令和存取数据都要通过传输介质传输数据和控制信号，连接不同部件进行信息传输的介质称为总线；

总线又分为：**地址线**（传输地址信息）、**数据线**（传输数据信息）、**控制线**（传输控制信息）。**CPU**访问主存时，先将主存地址、读/写命令分别送到总线的地址线和控制线，然后通过数据线发送和接收数据。

- **CPU**送到地址线的主存地址要首先存放在**主存地址寄存器**（MAR）中；
- 发送到或从数据线取来的信息要先存放在**主存数据寄存器**（MDR）中；

1.1.3 程序和指令的执行过程

计算机的功能通过执行程序实现，程序是由一串指令组成，程序的执行过程就是通过执行一条一条指令完成的。

- “**存储程序**”**工作方式**：

- 程序启动前，指令和数据都存放在存储器中，二者形式上没有差别，都是0/1序列。
- 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GPRs中。

指令：是具有特定格式、用0和1表示的一串0/1序列，用来指示CPU完成一个特定的原子操作。

指令的0/1序列通常被划分为操作码、地址码等字段。

- **操作码字段：**指出指令的操作类型
如取数、存数、加、减、传送、跳转等；
- **地址码字段：**指出指令所处理的操作数的地址，可以是寄存器编号，也可以是主存单元编号等。

一个16位的机器指令如下

100010 DW	mod	reg	r/m	disp8
100010 0 0	01	001	001	11111010

操作码

寻址方式

寄存器编号

立即数(位移量)

例：设一模型机字长为8位；有4个通用寄存器r0~r3，编号分别为0~3；有16个主存单元，编号为0~15，地址编码4位。

- 主存单元、CPU中的ALU、GPRs、IR、MDR等与处理数据相关的单元的宽度都是8位;
- PC和MAR等与地址相关的单元的宽度都是4位;
- 连接CPU和主存的总线中地址线为4位、数据线为8位、控制线若干位。

机器采用8位定长指令格式，即每条指令都有8位；并有R型和M型两种指令格式（操作数类型不同），如图所示：

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

这里，R[r]表示编号为r的寄存器中的内容；M[addr]表示地址为addr的主存单元的内容；←：表示从右向左传送数据。（寄存器传送语言 RTL, Register Transfer Language）

op	rt	rs	$R[rt] \leftarrow R[rt] \text{ op } R[rs]$ 或 $R[rt] \leftarrow R[rs]$
op	addr		$R[0] \leftarrow M[\text{addr}]$ 或 $M[\text{addr}] \leftarrow R[0]$

设指令集中有四种指令：

指令格式	op	含义
R	0000	寄存器间传送(mov)
	0001	加(add)
M	1110	取数(load)
	1111	存数(store)

例：

- **1110 0110**：功能为 $R[0] \leftarrow M[0110]$ ，表示将6号主存单元中的内容取到0号寄存器中；
- **0001 0001**：功能为 $R[0] \leftarrow R[0] + R[1]$ ，表示将0号和1号寄存器的内容相加后将结果送到0号寄存去。

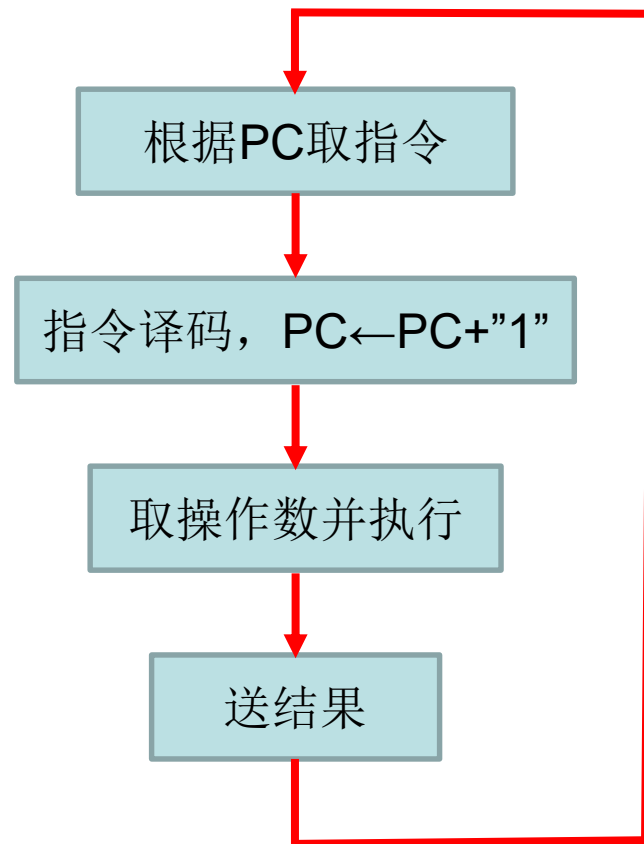
设某程序中有x、y、z三个变量，分别存放在主存5、6、7号单元中。则实现 $z=x+y$ 的机器指令代码及数据在主存单元中的初始内容如下所示：

主存地址	主存单元内容	内容说明	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: $M[0] \leftarrow R[0]$; op=1111;存数操作	store 7#, r0
5	0001 0000	操作数x，值为16	
6	0010 0001	操作数y，值为33	
7	0000 0000	结果z，初始值为0	

程序的执行过程

程序的执行过程就是周而复始地执行一条一条指令的过程。

每条指令的执行过程包括：从主存取指令、对指令进行译码、PC增量计算、取操作数并执行、将结果送主存或寄存器保存。



“存储程序”工作方式

● 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
- 用程序起始地址置PC（PC总指向下一条要执行的指令，这里开始的时候PC=0）

● 开始执行程序

第一步：根据PC取指令

第二步：指令译码，确定操作类型

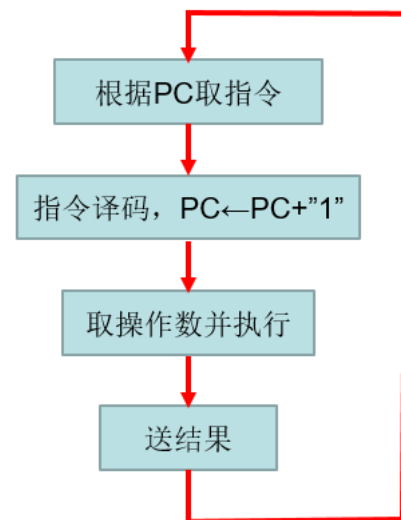
第三步：修改PC的值，使其指向下一条要执行的指令

第四步：取操作数

第五步：指令执行

第六步：回写结果，结果写入寄存器或内存单元

● 按照指令序列，重复上述步骤，直到所有指令执行完毕。



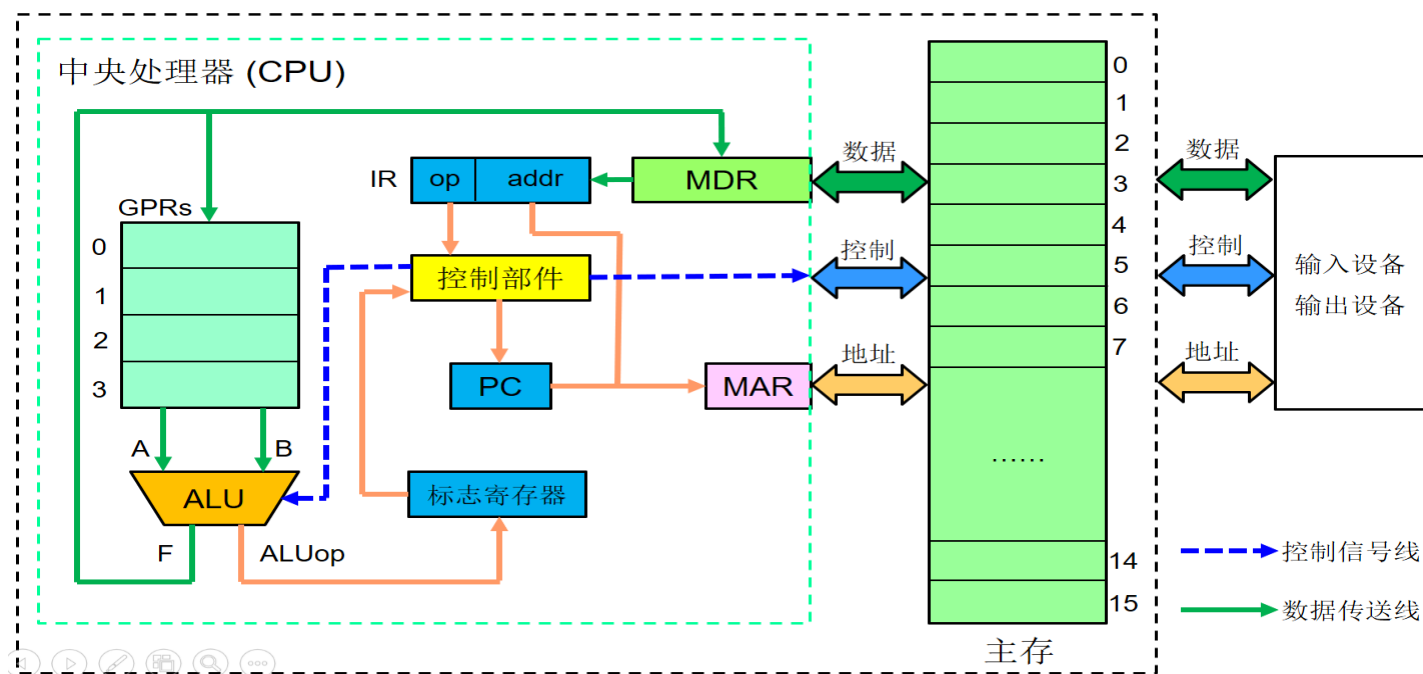
解释程序指令的执行流程

主存地址	主存单元内容	内容说明	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: $M[0] \leftarrow R[0]$; op=1111;存数操作	store 7#, r0
5	0001 0000	操作数x, 值为16	
6	0010 0001	操作数y, 值为33	
7	0000 0000	结果z, 初始值为0	

- ❑ 程序启动前, 指令和数据都存放在存储器中, 置 $PC=0$ 。
- ❑ 开始执行程序, 从第一条指令开始, 计算机自动取出指令, 一条一条执行。
 - 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, 其中指令在IR中, 数据在GPRs中;
 - 对IR中的指令进行译码, 按照指令规定的操作对GPRs中的数据进行处理
 - 同时 $PC \leftarrow PC + "1"$;
 - 保存结果;
- ❑ 周而复始, 按照取指、译码、执行的流程执行完所有的指令, 程序结束。

第一次课内容 (4.22)

- 历史的长河：从ABC到ENIAC到EDVAC
- 冯·诺依曼结构基本思想：**“存储程序”** 工作方式
- 冯·诺依曼机的基本结构：**运算器、控制器、存储器、输入设备、输出设备**五个基本部件。



- 程序和指令的执行过程

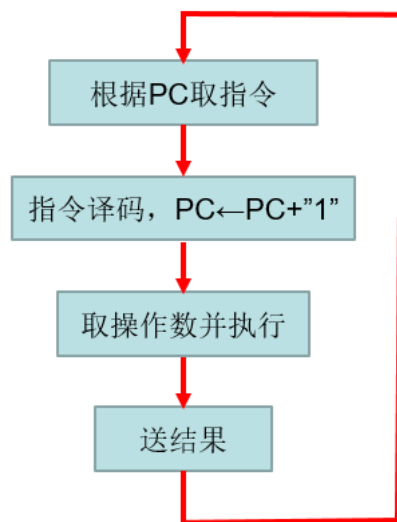
- ◆ 机器指令的结构: **操作码+地址码**

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

- ◆ 程序的执行过程:

从PC被置为程序的第一条指令的地址开始进入“**取指-译码**

-执行”的循环，指导



1.2 程序的开发和执行过程

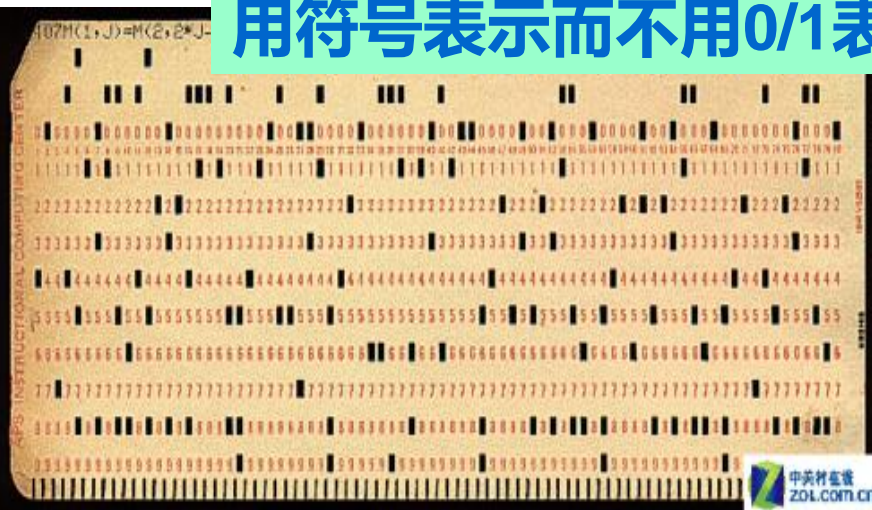
◆ 最早的程序开发是用**机器语言**编写程序，记录在纸带或卡片上



穿孔表示0，未穿孔表示1

太原始了，无法忍受，咋办？

用符号表示而不用0/1表示！



所有信息都是0/1序列！

如：

0: 0101 0110

1: 0010 0100

2:

3:

4: 0110 0111

5:

6:

jxx, 跳转指令

书写、阅读、维护都十分困难。

如：若想在第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。

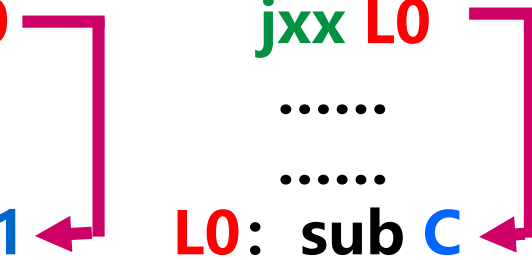
用汇编语言开发程序

- 若用**符号**表示指令和地址，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**表示地址
- 用**助记符**表示寄存器
-

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



用汇编语言编写的优点是：

- 可读性比机器语言大大增强，记忆汇编指令比机器指令容易的多，编写、维护程序的方便性大大提高。

此时，再在第4条指令前加指令，就不用改变add、jxx和sub指令中的地址码！

但有个问题：汇编语言是为方便人编写程序，但机器不认识汇编指令，无法直接执行！



需将汇编语言转换为机器语言！

用汇编程序转换

- 汇编语言源程序由**汇编指令**构成。
 - **汇编指令**是用助记符和标号来表示的指令和地址
 - 每条指令都包含操作码和操作数或其地址码。
- 汇编指令与机器指令一一对应。
 - 机器指令用二进制表示，汇编指令用符号表示。
- 汇编指令只能描述一些基本的操作
 - 取、存一个数
 - 两个数的算术（加、减、乘、除）或逻辑（与、或等）运算
 - 根据运算结果判断是否转移执行等
- 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！而且在不同体系结构的机器上不一定能运行！

结论：用汇编语言比机器语言好，但是，还是很麻烦！

- 汇编语言源程序由**汇编指令**构成。

- **汇编指令**是用助记符和标号来表示的指令
- 每条指令都包含操作码和操作数或其地址码。

- **汇编指令与机器指令一一对应。**

- **机器指令**用二进制表示，**汇编指令**用符号表示。

- **汇编指令：**

- 取、存
- 两个数
- 根据运算结果判断是否转移执行等

机器语言和汇编语言都是面向机器的语言，它们统称为**机器级语言**。

- 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不一定能运行！

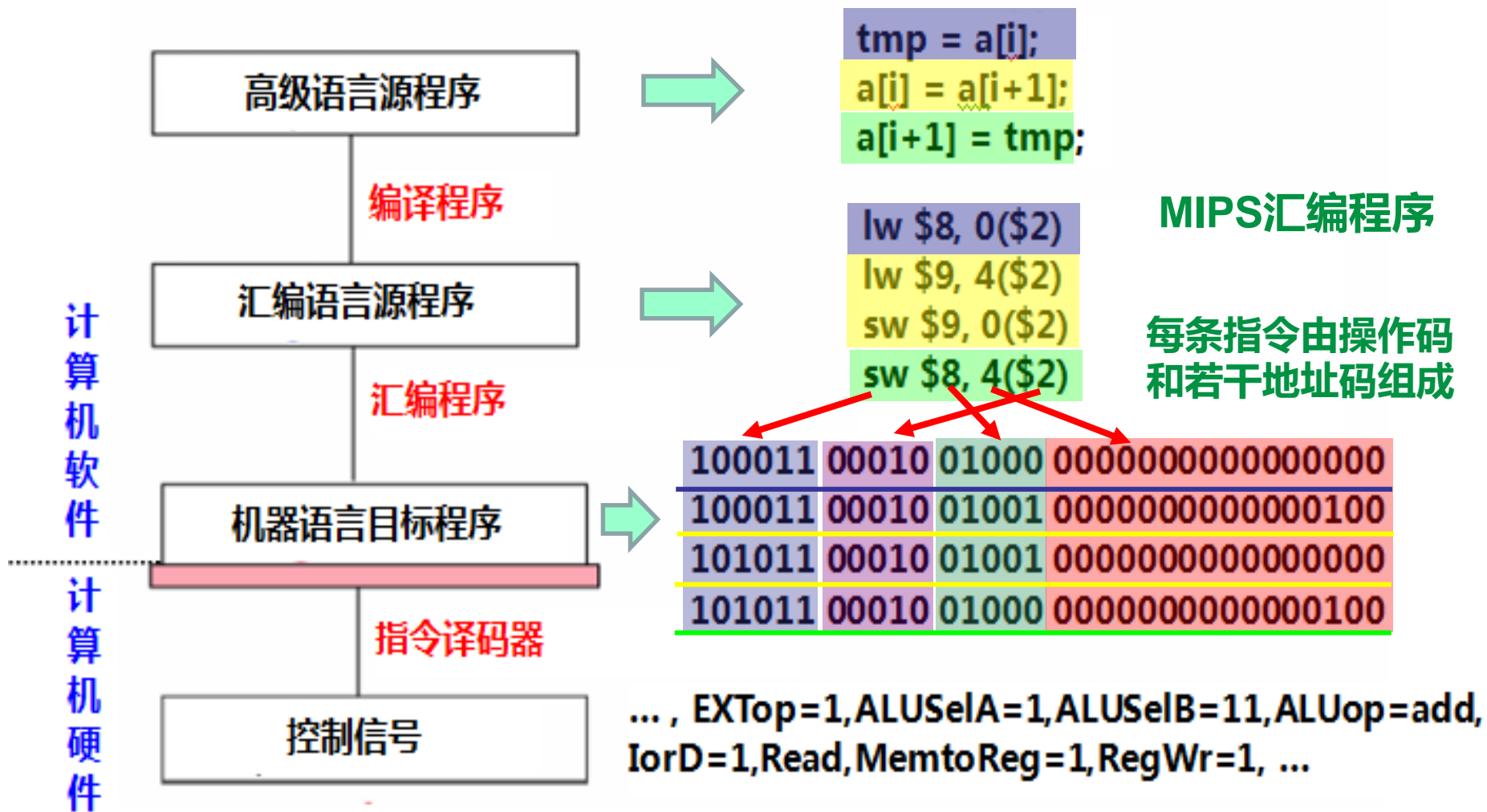
结论：用汇编语言比机器语言好，但是，还是很麻烦！

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（从1950年以后）
 - 它们与具体的机器结构无关
 - 面向问题描述，引入数据类型，相比机器级语言，描述能力大大增强
 - 高级语言中一条语句对应几条、几十条甚至几百条机器指令
 - 处理逻辑分为三种结构：**顺序结构**、**选择结构**、**循环结构**
- 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程序转换为机器语言程序
 - 有两种转换方式：“编译”和“解释”
 - **编译程序(Compiler)**：将高级语言源程序事先转换为机器级目标程序，然后再执行。执行时只要启动目标程序即可
 - **解释程序(Interpreter)**：将高级语言程序的语句逐条翻译成机器指令并立即执行，不生成目标文件。

不同层次语言之间的等价转换

实现数组中相邻两个元素的交换



MIPS: “无内部互锁流水级微处理器”，是世界上一种很流行的RISC处理器，采用RISC指令集。

一个典型高级语言程序的生成、转换过程

“hello world” C源程序

```
#include <stdio.h>

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

功能：输出 “hello,world”

① **编辑源程序**：通过文本编辑器编辑生成源程序，并保存到hello.c文件中。

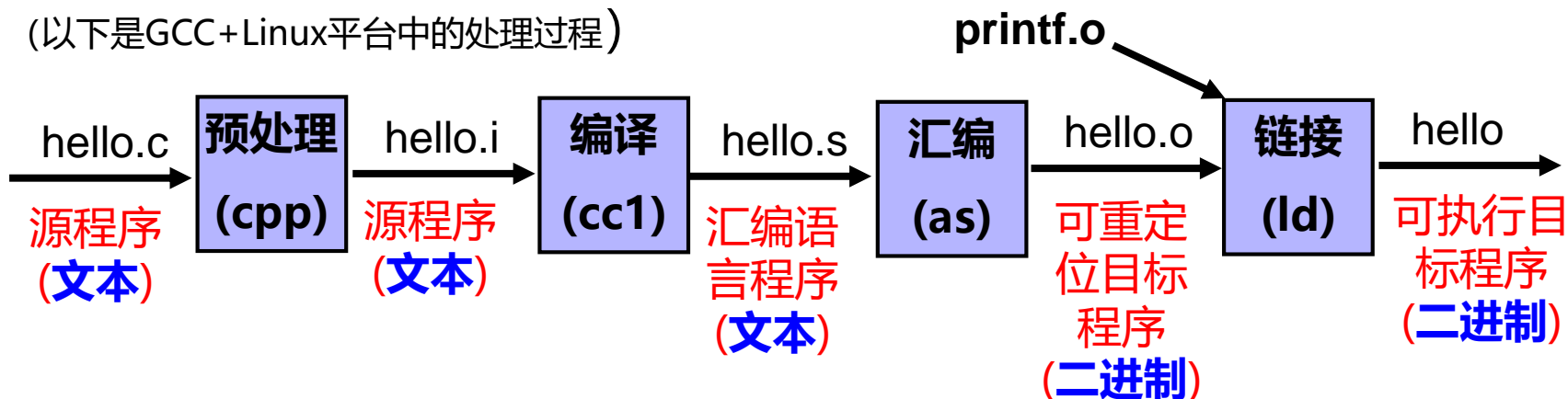
```
# i n c l u d e < s p > < s t d i o .
35 105 110 99 108 117 100 101 32 60 115 116 100 105 111 46
h > \n \n i n t < s p > m a i n ( ) \n {
104 62 10 10 105 110 116 32 109 97 105 110 40 41 10 123
\n < s p > < s p > < s p > < s p > p r i n t f ( " h e l
10 32 32 32 32 112 114 105 110 116 102 40 34 104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

hello.c的ASCII文本表示

计算机不能直接执行hello.c!

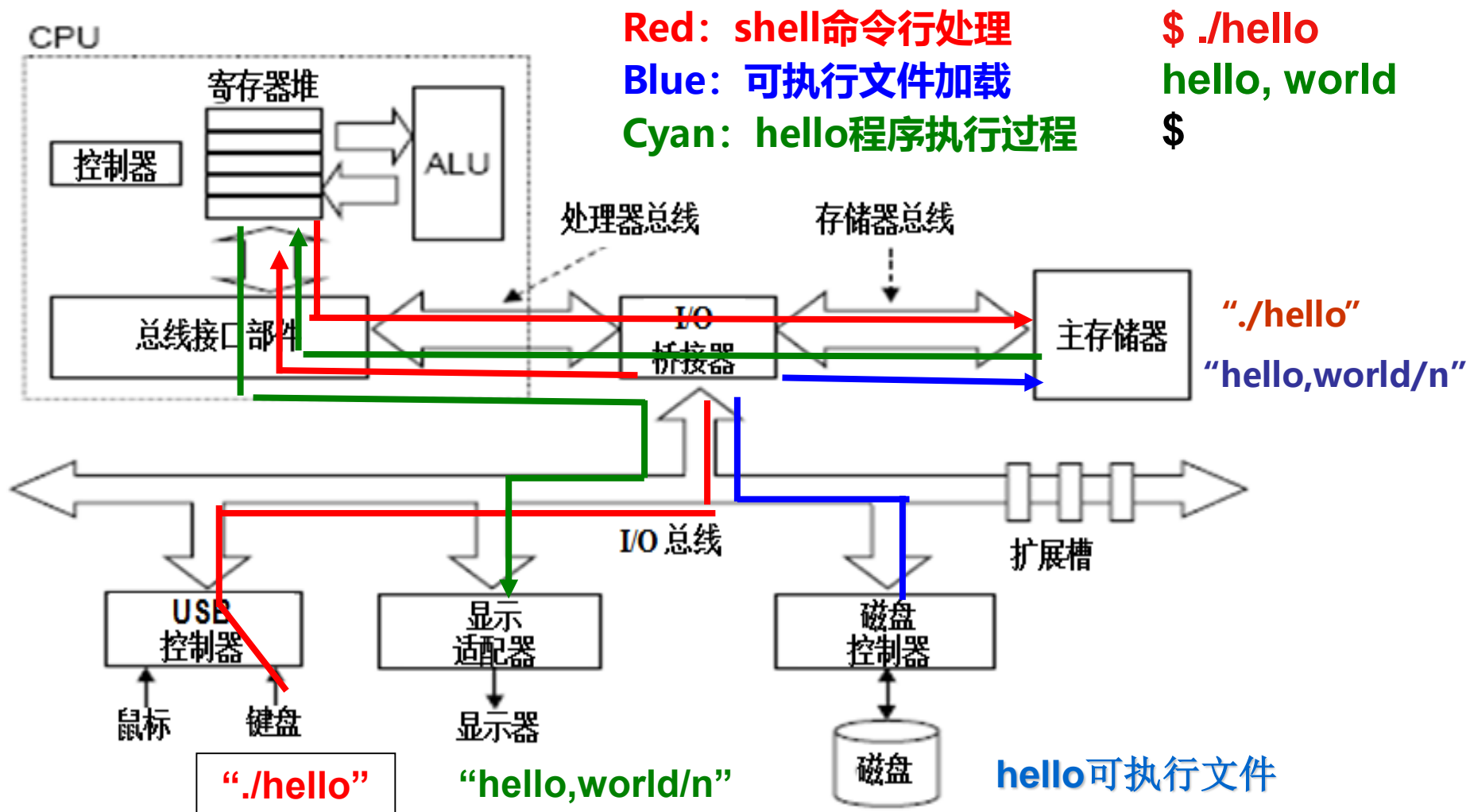
② 编译、链接，得到可执行目标程序

(以下是GCC+Linux平台中的处理过程)



③ 执行

Hello程序执行中的指令/数据流动过程



软件分类

一般软件分为**系统软件**和**应用软件**

- **系统软件** (System software)

对计算机的软硬件资源进行管理，以达到有效利用或简化编程的目的

- **操作系统** (Operating System): 资源管理, 作业调度、用户接口
- **语言处理系统**: 语言编辑和翻译程序、 Linker, Debug, etc ...
- **其他实用程序**: 磁盘碎片整理程序、备份程序等

- **应用软件** (Application software)

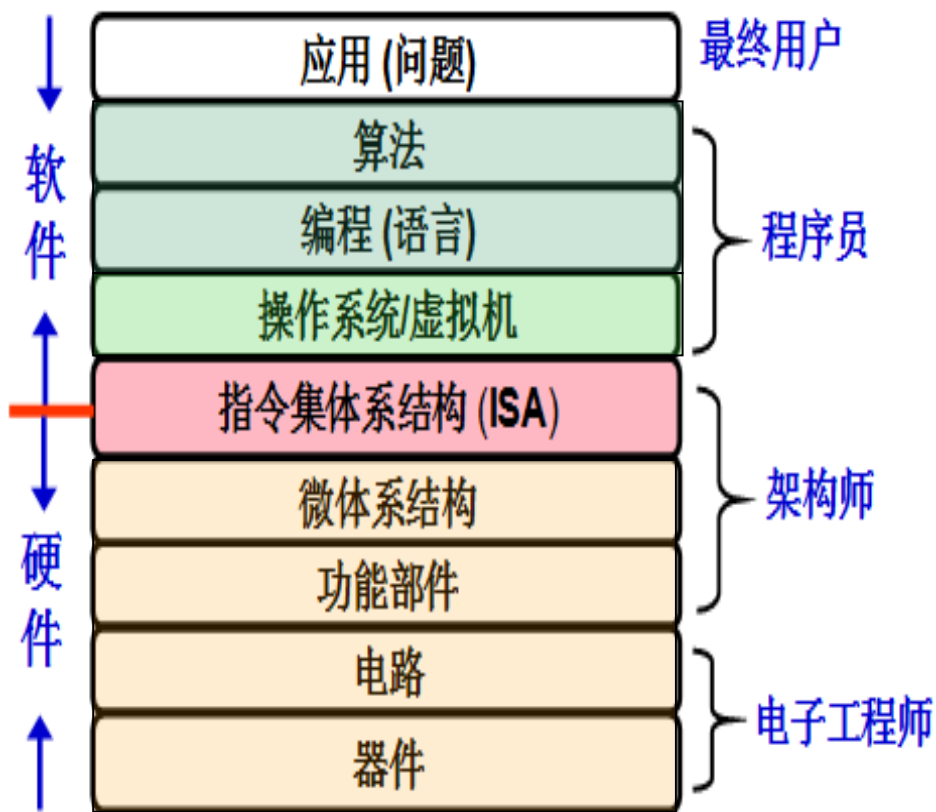
解决具体应用问题或完成具体应用任务的软件

- **各类媒体处理程序**: Word/ Image/ Graphics/...
- **管理信息系统** (MIS)
- **QQ、Game, ...**

1.3 计算机系统的层次结构

如何看待计算机系统?

电子工程师眼里的计算机是一台电子设备，而计算机使用者的眼里，计算机则是一台经过封装的**抽象计算机**，（从最底层的硬件到最高层的应用之间）存在多个抽象层。可描述如下：



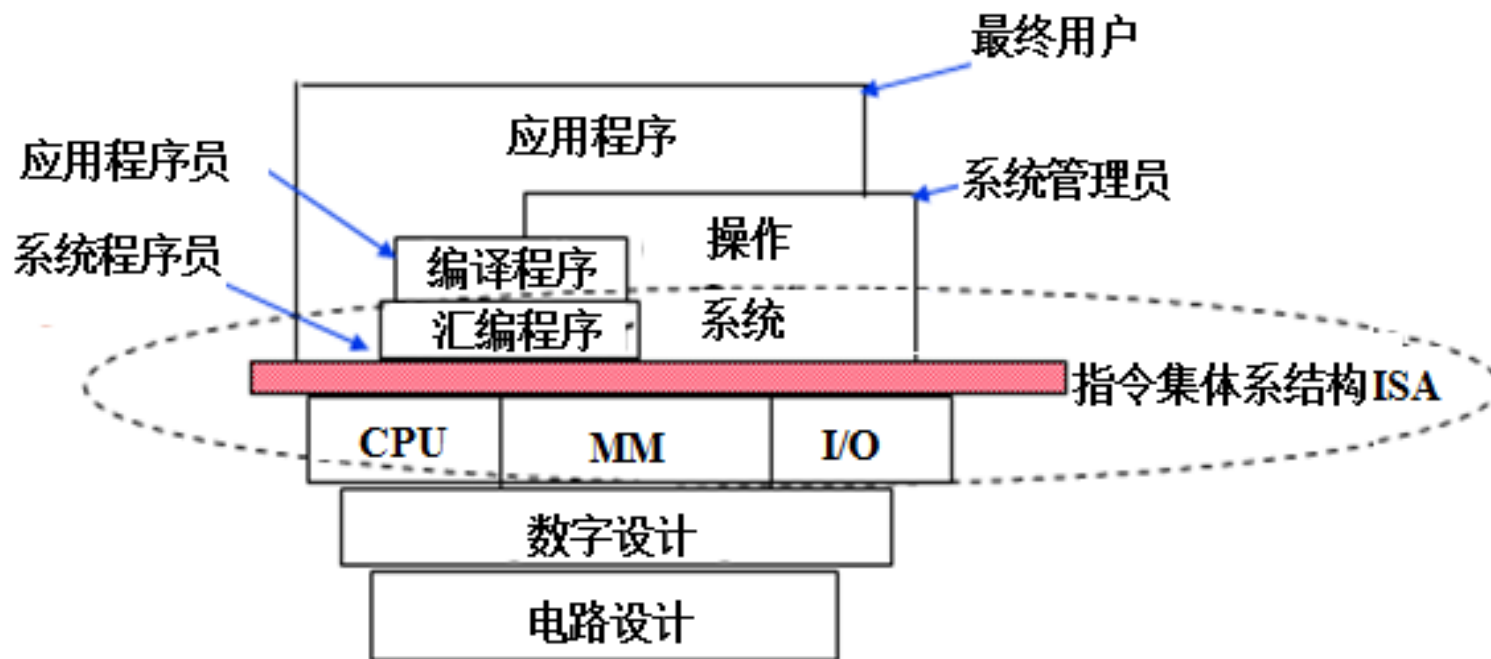
不同的人员看到的计算机是不同的：

- **应用程序员**：配置了操作系统和语言处理环境的抽象机，用来开发应用程序；
- **系统程序员**：在硬件之上由ISA提供了使用说明的抽象机，可开发系统程序；
- **ISA**：是计算机硬件到软件的接口，提供了如何操作计算机硬件的技术规范。

指令集体系结构 (ISA)

- **ISA** (**I**nstruction **S**et **A**rchitecture, 指令集体系结构), 是软件和硬件之间接口的一个完整定义。
- **ISA定义了一台计算机可以执行的所有指令的集合和软件使用硬件的方式方法。**具体包括:
 - 有哪些指令, 每条指令的格式、功能;
 - 指令操作数的类型、寻址方式;
 - 通用寄存器的个数、位数、编号、用途; 控制寄存器的定义;
 - I/O空间的编址方式、输入/输出结构和数据传送方式、存储保护方式;
 - 中断结构、机器工作状态的定义和切换等。

- ISA是计算机系统中**软件和硬件的交界面**。



ISA是对计算机组成的抽象，提供了**底层硬件为上层软件所能感知的部分**。所有的软件都工作在ISA之上。没有ISA，软件就不知道如何使用计算机，也就无从开发程序了。

- 对ISA之上的软件，只要ISA相同（指令集相同），程序就可以运行在具有不同硬件结构的计算机上；
- 对ISA之下的硬件，硬件必须提供ISA所规定的功能，否则就不能完整支持计算机系统的功能。

机器语言程序是不是最低级的程序？ 不是！

◆ ISA之下还有微体系结构

- 微体系结构包含一组微程序，使用微指令编写。微指令是硬件级语言，指挥电子器件和电路完成相应操作。
- 微程序是机器指令的具体实现。机器指令中的操作类型码相当于微程序的编码或“函数名”，执行一条机器指令相当于用指令操作码调用微程序，机器指令中操作数相当于传递给微程序的“参数”。

-
- 平台环境: **IA-32+Linux+C+gcc**
 - 主要内容: **描述程序执行的底层机制**
 - 思路: **在程序与执行机制之间的建立关联**

1.4 计算机性能评价（自学）

- 如何评价计算机的性能？

- 显然，不同的计算机，完成同样的工作所需时间最短的那台性能最好
- 但在多任务系统中，一段时间内可能有多个程序“分时”地轮流使用处理器，所以一个用户程序的执行过程中，可能同时还会有其他用户程序和操作系统程序在执行。
- 一般情况下，**程序的真正执行时间 \neq 结束时间 - 开始时间**
- 通常，一个程序的执行时间，除了程序包含的指令在CPU上执行所用的时间外，还包括磁盘访问时间、存储器访问时间、I/O操作时间以及操作系统运行这个程序所用的额外开销。

- **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。
- **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**

- **系统性能和CPU性能：**

- 系统性能：指系统的响应时间，它与CPU外的其他部分有关；
- CPU性能：特指用户CPU时间，它只包含CPU运行用户程序代码的时间。
- **系统性能和CPU性能是不等价的！**

计算机性能的基本评价指标

计算机有两种不同的性能评价指标

◦ 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比值的倒数来表示！

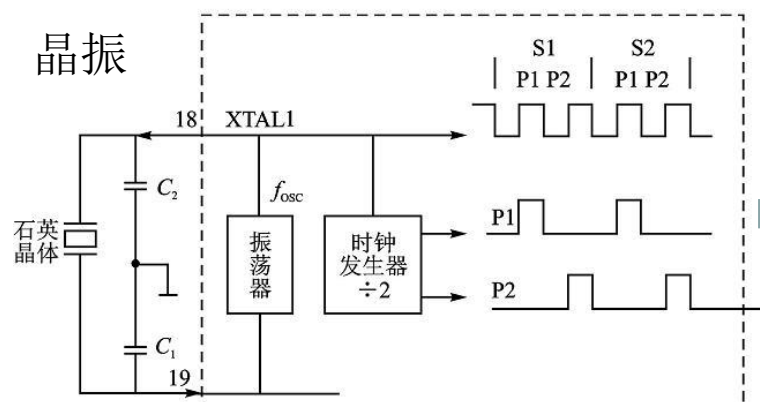
CPU执行时间的计算

● 时钟周期 (clock cycle, tick, clock tick, clock)

- 计算机执行一条指令的过程被分为若干步骤和相应的动作来完成，如取指令、取操作数、执行。
- 每一步动作都要有相应的控制信号进行控制，这些控制信号何时发出、作用时间多长，都要有相应的定时信号进行同步。
- 因此CPU必须能够产生**同步时钟定时信号**
——这就是CPU的主脉冲信号。

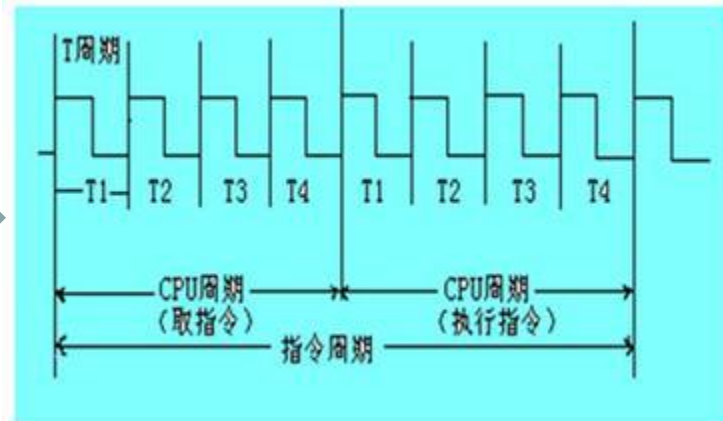
脉冲信号是怎么产生的呢？

晶振是基准，是计算机内部CPU、存储器等部件之间传输和操作数据的“基准口令”。就如同生活中要用[北京时间](#)，这样全中国的人在生活中才不会出问题。



外频

倍频



CPU执行时间的计算

● 时钟频率

- CPU的主脉冲信号的宽度称为时钟周期
- CPU的主频是指CPU主脉冲信号的时钟频率
- CPU的主频 = CPU时钟周期的倒数

一般来说，CPU主频越高，计算机速度越快，但CPU主频和计算机速度并不等价

- 一条指令往往包含多步，需要多个时钟周期才能执行完。

- **CPI (Cycles Per Instruction)**

- 指执行一条指令所需的**时钟周期数**。

- 由于不同指令的功能不同，所需的时钟周期也不同。

- 指令的CPI：对于一条特定的指令而言，其CPI是其执行所需的时钟周期数。这个值是**指令设计时制定**的，是一个确定值；

- 程序的CPI：是该程序中所有指令执行所需的**平均时钟周期数**，此时，CPI是一个平均值；

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

- 机器的CPI：是该机器指令集中所有指令执行所需的平均时钟周期数，此时，CPI也是一个平均值；

- **指令周期**：取出一条指令并执行这条指令的时间

-
- CPU 执行时间、程序CPU时钟周期数、时钟周期、时钟频率、程序指令条数、CPI之间的关系

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

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

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

$$\text{时钟频率} = 1/\text{时钟周期}$$

- CPI 用来衡量以下方面的综合结果
 - Instruction Set Architecture (ISA) and Implementation of that architecture (Organization & Technology)
 - Program (Compiler、Algorithm)

Aspects of CPU Performance

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

思考：三个因素与哪些方面有关？

	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?

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：

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

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：

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

已知CPU时间、时钟频率、总时钟数、指令条数，则程序综合CPI为：

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

● 时钟周期、指令条数、CPI的相互制约关系

- ✓ 时钟周期短→主频高→运算速度快?
- ✓ 指令条数少→程序执行得快?
- ✓ CPI小→指令周期短→程序执行得快?

Example1

程序P在机器A上运行需10 s， 机器A的时钟频率为2GHz。 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1.5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:

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

$$\text{时钟周期数A} = 10 \text{ sec} \times 2\text{GHz} = 20\text{G个}$$

$$\begin{aligned} \text{时钟频率B} &= \text{时钟周期数B} / \text{CPU时间B} \\ &= 1.5 \times 20\text{G} / 6 \text{ sec} = 5\text{GHz} \end{aligned}$$

机器B的频率是A的2.5倍， 但机器B的速度并不是A的2.5倍！

Example2

假设计算机M的指令集中包含A、B、C三类指令，其CPI分别为1、2、4。某个程序P在M上被编译成两个不同的目标代码P1和P2，P1含A、B、C三类指令的条数分别为8、2、2，P2含A、B、C指令的条数分别为2、5、3，问那个代码总指令数少？那个执行的速度快？它们的CPI分别是多少？

Answer:

P1和P2的总指令数分别为12和10，所以P2的总指令数少。

P1的总时钟周期数： $8 \times 1 + 2 \times 2 + 2 \times 4 = 20$

P2的总时钟周期数： $2 \times 1 + 5 \times 2 + 3 \times 4 = 24$

因为两个指令序列在同一台机器上运行，所以时钟周期一样，故P1比P2快

P1的CPI = $20 / 12 = 1.67$ ， P2的CPI = $24 / 10 = 2.4$

用指令执行速度进行性能评估

- 最早用来衡量计算机性能的指标是每秒钟完成的单个运算的指令条数

单位：MIPS (Million Instructions Per Second)

即， $\text{Instruction Count} / \text{Time} \times 10^6$ ，每秒执行多少百万条指令

统计对象：单字长定点指令，如加法运算

缺陷：用MIPS来对不同的机器进行性能比较有时不准确或不客观。

- 不同的机器的指令集不同
- 指令的功能不同，不同的机器上，同样的指令条数所完成的功能可能完全不同；
- 不同机器的CPI和时钟周期不同，同一条指令在不同的机器上所用的时间也不同。

• 浮点操作速度

单位：MFLOPS (Million FLOating-point operations Per Second)

每秒执行多少百万次浮点运算。——按浮点运算的次数来衡量。

其它单位：GFLOPS (10^9)、TFLOPS (10^{12})、PFLOPS (10^{15})、
EFLOPS (10^{18}) 等

- **等效指令速度**

- 设某类指令*i*在程序中所占比例为 w_i ，执行时间为 t_i ，则等效指令的执行时间为：

$$T = w_1 \times t_1 + w_2 \times t_2 + \dots + w_n \times t_n$$

- 若指令执行时间用时钟周期来衡量，则T等于CPI_{综合}。
- MIPS = 主频 / CPI

Example3: MIPS数不可靠!

假定某程序P编译后生成的目标代码有A、B、C、D四类指令组成，它们在程序中所占的比例分别是43%、21%、12%、24%，已知它们的CPI分别为1、2、2、2。现在对P进行编译优化，生成的新目标代码中A类指令条数减少了50%，其他指令的条数不变。问：1) 编译优化前后程序的CPI各是多少？

2) 假定程序在一台主频为50MHz的计算机上运行，则优化前后的MIPS各是多少？

Op	Freq	Cycle	Optimizing compiler	New Freq
A	43%	1	$21.5 / (21.5 + 21 + 12 + 24) = 27\%$	27%
B	21%	2	$21 / (21.5 + 21 + 12 + 24) = 27\%$	27%
C	12%	2	$12 / (21.5 + 21 + 12 + 24) = 15\%$	15%
D	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。

选择性能评价程序 (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！

课后作业

- P.27:

1. 名词解释：ALU，程序计数器，指令寄存器、
控制器、ISA、机器指令

2. (1)、(2)、(3)

- 5.

思考题：3