

计算机是如何运行的

作者: 李晓峰

邮箱:CY_LXF@163.COM

个人课程主页:[HTTPS://BUUER_XXTXIAOFENG.GITEE.IO/LXF/](https://BUUER_XXTXIAOFENG.GITEE.IO/LXF/)

前言

2014 年开始在高校任教，慢慢的觉得学信息类的学生应该对计算机的运行机理有一个大致的了解，于是在上各种课程时，都会把计算机的运行机理抽出或多或少的时间进行讲解，这个讲解过程中需要查阅大量的资料，涉及面非常广，也看到一些国内外学者写的类似这样的书籍，但是或多或少与我的思路有点不同，也希望按自己的思路进行整理编著一本尽量薄的书，使得刚开始学习的学生或者想对计算机运行机理进行了解的人，能够快速有个了解。

在根据思路整理材料的过程，在互联网上共享出来的大量资料，对此书的编著提供了非常大的帮助，能直接用的我就直接引用了，并且给出了链接，在此表示感谢，如有侵权，请联系我，我会替换掉相应内容。

另外写出来的文字尽量短，但是对于很多初学者来说，可能一句话里面就有很多基本概念，希望读者在读的时候能够真的理解了这句话在讲什么，不了解的概念可以通过在互联网搜一下。通过了解一个个基本概念，并且通过本书能够将这些概念的基本关系构建起来，对于初学者来说，应该会对计算机系统一个系统认识。

Hope so.

李晓峰

Contents

1	数的表示与运算	1
1.1	十进制 (Decimal)	1
1.2	二进制 (Binary)	3
1.3	八进制 (Octal)	3
1.4	十六进制 (Hexadecimal)	3
1.5	布尔代数 (Boolean Algebra)	4
2	中央处理器 CPU	11
2.1	晶体三极管	11
2.2	与非门	15
2.3	锁存器	17
2.3.1	RS 锁存器	17
2.3.2	D 锁存器 (电平触发)	17
2.3.3	D 触发器 (边沿触发)	19
2.4	加/减法器	19
2.5	冯诺依曼体系结构	21
2.6	算术逻辑单元 ALU	22
2.7	指令执行	23
2.7.1	PC 寄存器	26
2.8	现代 CPU 架构	27
2.8.1	8086	27
2.8.2	Intel Skylake (client) 微架构	29
2.9	启动	32
2.9.1	Firmware Support Package (FSP)	38
2.9.2	UEFI	39
3	电子设计自动化 (EDA: Electronic Design Automation)	41
3.1	电路逻辑设计	41
3.1.1	HDL/VHDL	41
3.1.2	门电路 (图)	42

3.2 仿真验证 (逻辑)	43
3.3 逻辑综合	44
3.4 形式验证	44
3.5 可测性设计 (DFT:Design ForTest)	45
3.6 布局规划 (Floor Plan)	45
3.7 时钟树综合 (CTS: Clock Tree Synthesis)	45
3.8 布线 (Place & Route)	46
3.9 寄生参数提取	46
3.10 版图物理验证	46
4 机器码/指令集	49
5 汇编语言	55
6 数据存储	59
6.1 寄存器 (Register)	59
6.2 高速缓存 (cache)	60
6.3 内存 (memory)	60
6.4 外部存储 (external storage)	61
6.4.1 外设访问	61
6.4.2 硬盘 (harddisk)	61
6.4.3 光盘 (Compact Disc)	63
6.4.4 磁带 (magnetic tape storage)	64
6.5 存储的逻辑形式	64
6.5.1 文件系统 (File System)	65
7 可执行程序/文件	69
7.1 PE format	70
8 高级语言	71
8.1 C	71
8.2 VB	74
8.3 C++	74
8.4 C#	77
8.5 Java	77
8.6 Python	82
9 操作系统	87
9.1 API	88
9.2 系统调用 (System Calls)	90
9.3 Windows 运行时 (Windows Runtime)	92
A IC 产业链	99
B 网络安全产业链	101

CONTENTS

Appendices	99
-------------------	-----------

List of Figures

2.1	三极管逻辑示意图 ²	12
2.2	三极管测试电路 ²	12
2.3	三极管逻辑示意图 ²	13
2.4	三极管测试电路 ²	13
2.5	集成电路中的三极管在电子显微镜下的照片	14
2.6	三极管的两种封装	14
2.7	非运算电路的逻辑符号和基本电路 ⁵	16
2.8	与运算电路的逻辑符号和基本电路 ⁵	16
2.9	RS 锁存器的原理图 ⁶	18
2.10	D 型锁存器的原理图 ⁷	18
2.11	D 型边沿触发器的原理图	19
2.12	全加器的逻辑电路实现	21
2.13	1 bit ALU 逻辑电路图	23
2.14	由 1 bit ALU 组成 4 bit ALU 逻辑电路图 ¹⁰	25
2.15	由 1 bit ALU 组成的 1 bit CPU	26
2.16	8086 系统架构图 ¹²	29
2.17	8086 系统架构图 ¹²	30
2.18	Skylake(client) 微架构 ¹⁴	31
2.19	Skylake(client) 双核 SOC 框图 ¹⁴	31
2.20	Skylake(client) 四核 SOC 框图 ¹⁴	32
4.1	ARM 实现的指令集	50
4.2	龙芯架构 32 位精简版典型指令编码格式	54
4.3	龙芯架构 32 位精简版指令码表截取，红框部分是逻辑“与”、“或”、“异或”指令。	54
5.1	Elliott 903 纸带计算机数据的读入	55
6.1	机械硬盘内部结构	62
6.2	机械硬盘内部结构	63
6.3	外置光驱和光盘	64
6.4	磁带机和磁带	65

LIST OF FIGURES

8.1	丹尼斯·里奇 (Dennis MacAlistair Ritchie) 被世人尊称为“C 语言之父”“Unix 之父”，C 语言的诞生是现代程序语言革命的起点，今天，C 语言依旧在系统编程、嵌入式编程等领域占据着统治地位，而 Unix 在操作系统的发展历史上也是一个里程碑的系统，其不仅还用于大型机上，而且 Linux 和 MacOS 都是在其基础上发展而来。其获得美国国家技术奖章和图灵奖	74
8.2	本贾尼·斯特劳斯特卢普 (Bjarne Stroustrup, 1950 年 6 月 11 日-)，丹麦人，计算机科学家，在德克萨斯 A&M 大学担任计算机科学的主席教授。他最著名的贡献就是开发了 C++ 程序设计语言。其个人网站是 https://www.stroustrup.com/index.html	76
8.3	詹姆斯·高斯林 (James Gosling)，1955 年 5 月 19 日出生于加拿大，Java 编程语言的共同创始人之一，一般公认他为“Java 之父”。1977 年获得了加拿大卡尔加里大学计算机科学学士学位，1983 年获得了美国卡内基梅隆大学计算机科学博士学位。	78
8.4	Java 的 logo	81
8.5	吉多·范罗苏姆 (Guido van Rossum)，荷兰计算机程序员，他作为 Python 程序设计语言的作者而为人们熟知。在 Python 社区，吉多·范罗苏姆被人们认为是“仁慈的独裁者 (BDFL:Benevolent Dictator For Life)”，意思是他还关注 Python 的开发进程，并在必要的时刻做出决定。他在 Google 工作，在那里他把一半的时间用来维护 Python 的开发。2020 年 11 月 12 日，64 岁的 Python 之父 Guido van Rossum 在自己社交网站上宣布：由于退休生活太无聊，自己决定加入 Microsoft 的 DevDiv Team	83
8.6	python 的 Logo	85
A.1	IC 产业链	100
B.1	网络信息安全产品及服务产业链	102

Chapter 1

数的表示与运算

1.1 十进制 (Decimal)

十进制是以十为基础的计数系统，我国最迟在商代已经使用了十进制，从现已发现的商代陶文和甲骨文中，可以看到当时已能够用一、二、三、四、五、六、七、八、九、十、百、千、万等十三个数字，十进位值制的记数法是古代世界中最先进、科学的记数法，对世界科学和文化的发展有着不可估量的作用，是中国文明对世界的一个重大贡献，正如李约瑟所说的：“如果没有这种十进位制，就不可能出现我们现在这个统一化的世界了。”

在计算数学方面，中国大约在商周时期已经有了四则运算，到春秋战国时期整数和分数的四则运算已相当完备。其中，出现于春秋时期的正整数乘法歌诀“九九歌”，堪称是先

进的十进位记数法与简明的中国语言文字相结合之结晶，这是任何其它记数法和语言文字所无法产生的。从此，“九九歌”成为数学的普及和发展最基本的基础之一，一直延续至今。

“十进制”顾名思义就是“到十进一的系统”，在计数时从 0 到 1... 到 9，到十以后进一位，同时本位循环到 0，那么就是 10，也就是说最右边为最低位，那么从右到左，第二个位置上有 1 表示有一个 10。而 23 表示 2 个 10 和一个 3。而第三个位置如果有 1 表示 10 个 10，也就是 10^2 ，第四个位置上有 1 表示有 10 个 (10 个 10)，也就是 10^3 ，在十进制中，我们将 10 称为“基数”，简称“基”，最后计数的多少统一表示：

$$\sum_{\text{所有非空位置}} \text{基}^{\text{位置号}-1}$$

如果基数为 b ，数字位置从 0 开始算起，共有 $n + 1$ 位数，那么 b 进制系统的计数可以表示为：

$$\sum_{k=0}^n a_k b^k$$

其中 a_k 为在位置 k 上的数字，且 $a_k \in \{0, 1, \dots, b - 1\}$, $k = 0, 1, \dots, n$ 。

1.2 二进制 (Binary)

二进制的基为 2，二进制每一位上的数的范围是 {0, 1}，那么二进制的计数可以表示为：

$$a_n a_{n-1} \dots a_1 a_0 = \sum_{k=0}^n a_k 2^k$$

比如二进制数 $1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 0 + 1 = 13$ 。

1.3 八进制 (Octal)

八进制的基为 8，八进制每一位上的数的范围是 {0, 1, 2, 3, 4, 5, 6, 7}，那么八进制的计数可以表示为：

$$a_n a_{n-1} \dots a_1 a_0 = \sum_{k=0}^n a_k 8^k$$

比如八进制数 $1101 = 1 \times 8^3 + 1 \times 8^2 + 0 \times 8^1 + 1 \times 8^0 = 512 + 64 + 0 + 1 = 577$ 。

比如二进制数 $3170 = 3 \times 8^3 + 1 \times 8^2 + 7 \times 8^1 + 0 \times 8^0 = 3 \times 512 + 1 \times 64 + 7 \times 8 + 0 = 13 = 1536 + 64 + 56 = 1656$ 。

1.4 十六进制 (Hexadecima)

十六进制的基为 16，十六进制每一位上的数的范围是 {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}，那么十六进制的计

数可以表示为：

$$a_n a_{n-1} \dots a_1 a_0 = \sum_{k=0}^n a_k 16^k$$

比如十六进制数 $1101 = 1 \times 16^3 + 1 \times 16^2 + 0 \times 16^1 + 1 \times 16^0 = 4353$ 。

比如十六进制数 $D2FA = 13 \times 16^3 + 2 \times 16^2 + 15 \times 16^1 + 10 \times 8^0 = 54010$ 。

1.5 布尔代数 (Boolean Algebra)

布尔代数是用来研究逻辑的过程中产生的，1854年，英国数学家乔治·布尔 (George Boole, 1815 – 1864)^②发表了《思维规律》这部杰作，布尔代数问世了，数学史上树起了一座新的里程碑。下面我们引用阮一峰“布尔代数入门”^③帖子中的内容，对布尔代数进行简要说明。

逻辑推理通常都是用文字来表达，比如最著名的“三段论”：

所有人都要死的，
苏格拉底是人，
所以，苏格拉底是要死的。

²关于布尔的介绍可以参考斯坦福网站上的材料，链接<https://meinong.stanford.edu/entries/boole/>

³“布尔代数入门”，网址<http://www.ruanyifeng.com/blog/2016/08/boolean-algebra.html>

1:



乔治·布尔 (George Boole, 1815 – 1864)¹

这种推理能不能用数学来表示，或者能不能用符号来表示？布尔对其进行了系统研究。

首先，发明了“集合论”(set theory)，他认为，逻辑思维的基础是一个个集合 (Set)，每一个命题表达的都是集合之间的关系 (并 \cup 、交 \cap 、差 $-$)。比如，所有人类组成一个集合 R ，所有会死的东西组成一个集合 D ，“所有人都是要死的”，这个命题用集合论的写法就是： $R \cap D = R$ 。把苏格拉底看作集合 S ，“苏格拉底是人”，表示为 $S \cap R = S$ ，我们把第一个式子带入第二个，会形成以下推理：

$$\begin{aligned} & S \cap R \\ &= S \cap (R \cap D) \\ &= (S \cap R) \cap D \\ &= S \cap D \\ &= S \end{aligned}$$

最后推导的两步意思是：苏格拉底与会死的东西的交集，就是苏格拉底，即苏格拉底也属于会死的东西。

上面这个例子很直观，我们看看下面这个例子。

“哲学家都是有逻辑头脑的，一个没有逻辑头脑的人总是很顽固。”，根据这个描述，我们能推出什么结论？

首先我们提取出来四个集合：哲学家 (philosopher) 集合，

记为 P , 有逻辑 (logic) 的人的集合, 记为 L , 无逻辑人的集合, 记为 NL , 顽固 (stubborn), 记为 S 。

"哲学家都是有逻辑头脑的", 我们描述为: $P \cap L = P$ 。

“没有逻辑头脑的人总是很顽固”, 我们描述为: $NL \times S = NL$ 。

我们看看哲学家和无逻辑的交集:

$$\begin{aligned} & P \cap NL \\ &= (P \cap L) \cap NL \\ &= P \cap (L \cap NL) \\ &= P \cap \emptyset \\ &= \emptyset \end{aligned}$$

其中 \emptyset 表示空集, 这个推理结论说哲学家与没有逻辑的人的交集, 是一个空集, 也就是说哲学家没有没有逻辑的。

我们看看无逻辑和顽固的交集, 下面的 \tilde{P} 表示 “非哲学家”:

$$\begin{aligned}
 & NL \cap S \\
 & = NL \cap S \cap (P \cup \tilde{P}) \\
 & = NL \cap S \cap P \cup NL \cap S \cap \tilde{P} \\
 & = \phi \cap S \cup NL \cap S \cap \tilde{P} \\
 & = NL \cap S \cap \tilde{P} \\
 & = NL
 \end{aligned}$$

通过上面的推理我们知道 $NL \cap S \cap \tilde{P} = NL \Rightarrow S \cap \tilde{P} = NL \cap$ 其他情况，结论就是顽固的人与非哲学家之间有交集。通俗的表达就是：一些顽固的人，不是哲学家，或者一些不是哲学家的人，很顽固。

由此可见，集合论可以帮助我们得到直觉无法得到的结论，保证推理过程正确，比文字推导更可靠。

既然命题可以用集合表达，那么逻辑推导可以表示为一系列集合运算吗？答案是肯定的。

我们看一个例子，来说明逻辑推导和集合运算之间的对应关系，假设有这样一个命题。

一名顾客走进宠物店，对店员说：“我想要一只公猫，白色或黄色均可；或者一只母猫，除了白色，其他颜色均可；或者只要是黑猫，我也要。”

这名顾客的要求用集合论表达，就是下面的式子。

$$\text{公猫} \cap (\text{白色} \cup \text{黄色}) \cup \text{母猫} \cap \text{非白色} \cup \text{黑猫}$$

现在店员拿出一只灰色的公猫，请问是否满足要求？

我们先定义布尔代数里面的运算法则（如图1.1），这个法则和集合的运算法则很像。

变量 A	变量 B	与 ($A \wedge B$ or $A \bullet B$ or A AND B)	或 ($A \wedge B$ or A OR B)	非 (\tilde{A} or $\neg A$ or $NOT A$)
0	1	0	1	0
0	0	0	0	1
1	1	1	1	0
1	0	1	1	1

Table 1.1: 布尔值函数的定义

在布尔代数中我们规定，个体属于某个集合用 1 表示，不属于就用 0 表示。灰色的公猫属于公猫集合，就是 1，不属于白色集合，就是 0。“灰色的公猫”是否满足“顾客要求”，我们可以等价为以下的布尔代数运算过程。

$$1 \cap (0 \cup 0) \cup 0 \cap 1 \cup 0 = 0$$

我们通过这个运算得到结论，灰色的公猫不满足要求。我们可以从这个示例看到，布尔代数是一种计算命题真伪的数学方法。

在大学计算机类的专业课程设置中，布尔代数^②内容的学习通常会设置在“离散数学”课程中^③。

2:



布尔代数译本

1938 年克劳德·艾尔伍德·香农 (Claude Elwood Shannon, 1916 年 4 月 30 日—2001 年 2 月 24 日) 在 MIT 获得电气工程硕士学位，硕士论文题目是《A Symbolic Analysis of Relay and Switching Circuits》(继电器与开关电路的符号分析)。当时他已经注意到电话交换电路与布尔代数之间的类似性，即把布尔代数的“真”与“假”和电路系统的“开”与“关”对应起来，并用 1 和 0 表示。于是他用布尔代数分析并优化开关电路，这就奠定了数字电路的理论基础，从此布尔代数就成为电路和计算机运行的必不可少的逻辑工具。哈佛大学的 Howard Gardner 教授说，“这可能是本世纪最重要、最著名的一篇硕士论文。”

1948 年，香农在 Bell System Technical Journal 上发表了《A Mathematical Theory of Communication》一文，这是一个划时代的文章，创建了信息时代的理论基础——信息论，香农因而被称为“信息论之父”。

1949 年，香农发表了其另外一篇重要文章《Communication Theory of Secrecy Systems》(保密系统的通信理论)，该文刚一发表就引起轰动，香农也随之为美国政府聘为密码事务顾问。这篇论文为对称密码系统的研究建立了一套数学理论，从此密码术变成为密码学了，由一门艺术或技艺变成一门真正的科学。



Chapter 2

中央处理器 CPU

2.1 晶体三极管

三极管一个迷人的特性是其开关特性，也就是当输入到某个值时，有输出。

晶体三极管（以下简称三极管）按材料分有两种：锗管和硅管。而每一种又有 NPN 和 PNP 两种结构形式，但使用最多的是硅 NPN 和锗 PNP 两种三极管（逻辑示意图见2.1），（其中，N 是负极的意思（代表英文中 Negative），N 型半导体在高纯度硅中加入磷取代一些硅原子，在电压刺激下产生自由电子导电，而 P 是正极的意思（Positive）是加入硼取代硅，产生大量空穴利于导电）。两者除了电源极性不同外，其工作

原理都是相同的。¹

我们以 NPN 为例搭建一个测试电路 (如图2.2), 发现一个非常有意思的物理现象 (如图2.3、2.4), B、E 两点间的电压 U_{BE} 达到某个数值时 (记为 U_{BER}), 三极管导通, U_{CE} 增大, 特性曲线右移, 但当 $U_{CE} > 1.0V$ 后, 特性曲线几乎不再移动。²

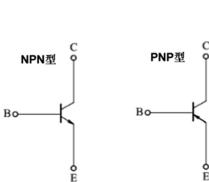


Figure 2.1: 三极管逻辑示意图²

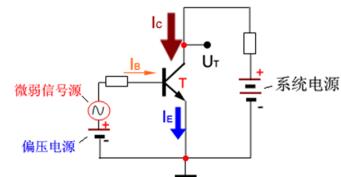


Figure 2.2: 三极管测试电路²

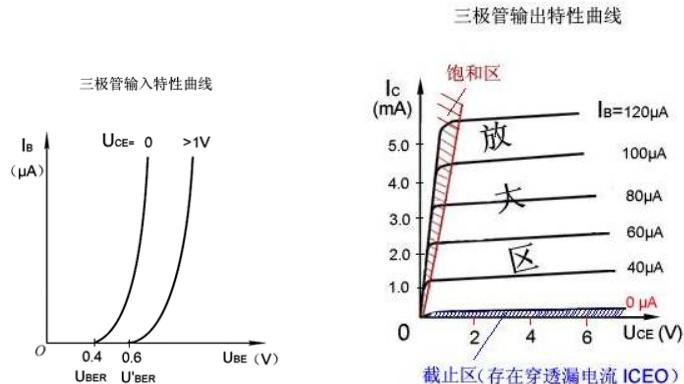
当 $I_B = 0$ 时, I_C 趋近 0, 称为三极管处于截止状态, 相当于开关断开; 当 $I_B > 0$ 时, I_B 轻微的变化, 会在 I_C 上以几十甚至百多倍放大表现出来; 当 I_B 很大时, I_C 变得很大, 不能继续随 I_B 的增大而增大, 三极管失去放大功能, 表现为开关导通。

我们从三级管的电特性上可以看出, 三极管核心功能一是放大功能: 小电流微量变化, 在大电流上放大表现出来; 二是开关功能: 以小电流控制大电流的通断。²

在集成电路中如何使得集成的基本组成单元越来越小, 对

¹ 引自 <https://baike.baidu.com/item/>

² 引自 “史上最详细图解三极管” <http://www.elecfans.com/d/652842.html>

Figure 2.3: 三极管逻辑示意图²Figure 2.4: 三极管测试电路²

于集成电路小型化和低功耗都非常关键，图2.5³是集成电路在电子显微镜下的照片。也有一些三极管作为独立芯片，如2.6⁴。

³此图引自http://www.semiconductor-today.com/news_items/2018/jun/epfl_290618.shtml

⁴来自<https://www.digikey.cn/product-detail/zh/microchip-technology/TC8020K6-G/TC8020K6-G-ND/4902536>

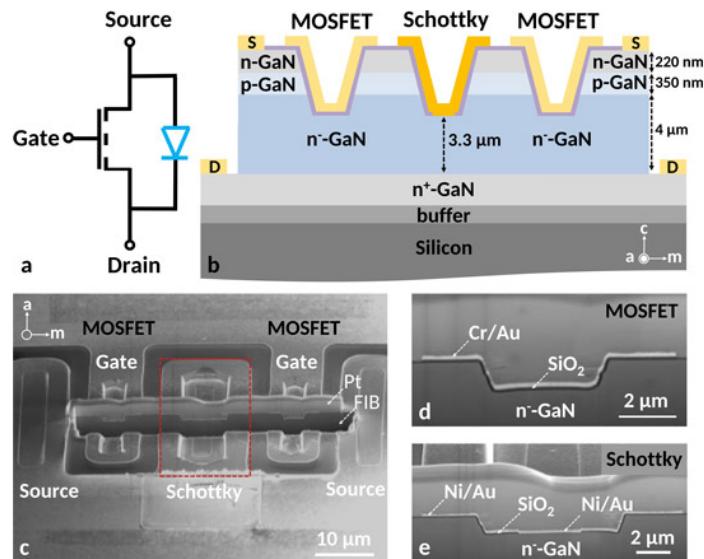


Figure 1: (a) Equivalent circuit. (b) Schematic of integrated vertical MOSFET-Schottky barrier diode (SBD). Scanning electron microscope images of (c) integrated vertical MOSFET-SBD, and (d) cross-sections of integrated vertical MOSFET, and (e) integrated vertical SBD.

Figure 2.5: 集成电路中的三极管在电子显微镜下的照片



Figure 2.6: 三极管的两种封装

2.2 与非门

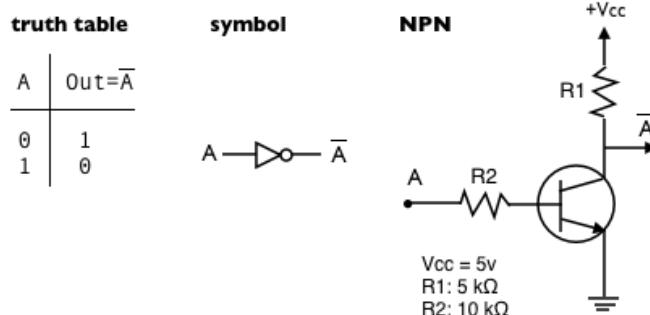
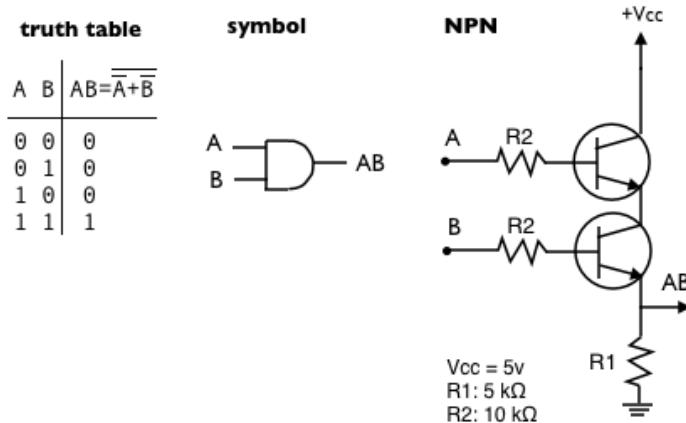
布尔代数 $\langle B, \text{and}, \text{or}, \text{not}, 0, 1 \rangle$ 的三个基本运算与或非都可以用三极管表示，这也就意味着，任何布尔代数预算都可以用三极管搭建电路来实施。

我们知道布尔代数的定义了一个代数结构，元素为 0、1，三个基本运算为与、或、非，分别记为 $\&$ 、 $-$ 、 $+$ ，而或可以用非运算表示，所以如果我们可以用三极管电路表示 0、1，表示非和与运算，是不是就意味着可以搭建任何的代数运算电路？

我们知道，在数字电路中我们应用的三极管的开关特性，那么我们可以用地电平（通常是 0v）表示 0，用导通电压（通常有 5v、3.6v）表示 1。

非运算电路如2.7⁵所示，输入为高电平，输出为低，输入为低电平，输出为高，真值表同非运算。与运算电路如2.8所示，两个输入一个输出，其真值表与与运算相同。

⁵ 图片引自“NPN and PNP logic gates” <http://codeperspectives.com/computer-design/npn-pnp-logic-gates/>

NotFigure 2.7: 非运算电路的逻辑符号和基本电路⁵**NPN And**Figure 2.8: 与运算电路的逻辑符号和基本电路⁵

2.3 锁存器

可以用三极管搭建一个电路，进行某个状态的锁存，也就是记忆，解决了计算过程中的存储问题。

2.3.1 RS 锁存器

RS 锁存器 (RS Latch) 实现的电路如图2.9⁶所示：

当 $R=1$ 时，输出为 0，故 R 又称为直接置“0”端，或“复位”端 (reset)

当 $S=1$ 时，输出也为 1，故 S 又称为直接置“1”端，或“置位”端

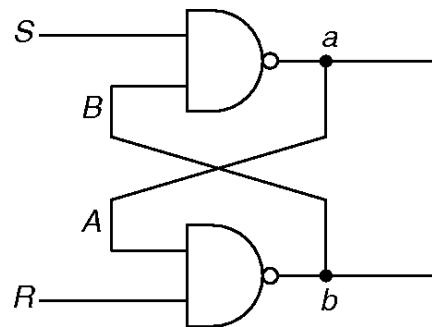
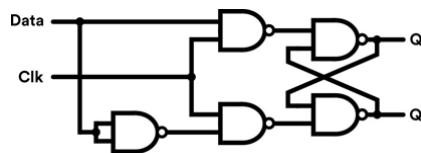
当 $R=S=0$ 时，输出保持不变，这个特性实现了 1bit 的存储

2.3.2 D 锁存器 (电平触发)

D 锁存器 (D-type Latch) 也具有存储功能，电路如图2.10⁷所示：

⁶引自 <http://www.c-jump.com/CIS77/CPU/Storage/lecture.html>

⁷引自 <https://arith-matic.com/notebook/logic-gates-registers>，其他参考见“Constructing Memory Circuits (Registers) with SR-latches” www.mathcs.emory.edu/~jallen/Courses/355/Syllabus/2-seq-circuits/memory.html

Figure 2.9: RS 锁存器的原理图⁶Figure 2.10: D 型锁存器的原理图⁷

Clk 是时钟信号，当 Clk=1 时， $Q=Data$ ；当 Clk=0 时，不管 Data 是什么值， Q 保持不变。电路中 $Q' = \text{not } Q$ ，可以看出 D 型触发器也对状态有存储功能。

2.3.3 D 触发器 (边沿触发)

D 型触发器 (D-type flip-flop) 实现原理如图2.11所示，在这里我们要注意到图中非门的延时效果，我们可以看到只有 clock 端从 0 变成 1 时，才完成一次数据的锁存，也就是在 clock 的上升沿完成锁存，所以我们称其为边沿触发器。

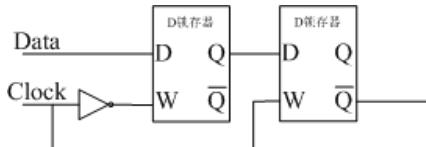


Figure 2.11: D 型边沿触发器的原理图

2.4 加/减法器

可以用门电路构建基本的算术运算，为高级的运算奠定基础。

我们先看看一位 (1bit) 加法器的真值表，如表2.1，其中 A、B 是输入， C_{in} 是进位输入，也可以看做输入，S 是加法器的输出， C_{out} 是进位输出。

Table 2.1: 全加器的真值表

C_{in}	A	B	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

全加器的逻辑电路实现如图2.12，同样我们可以实现减法器等等，任何算术运算电路。

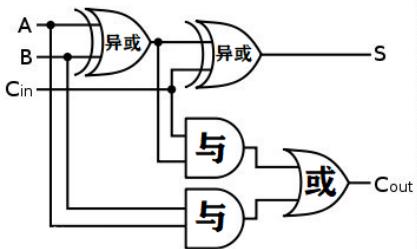


Figure 2.12: 全加器的逻辑电路实现

2.5 冯诺依曼体系结构

冯诺依曼和他的同事共同设计了第一台现代意义上的计算机 ENIAC，虽然是一台电子管计算机，但其提出的设计思想被一直沿用下来。

目前的计算机体系结构时冯诺依曼体系结构 (von Neumann architecture)^④，冯诺依曼结构也称普林斯顿结构，是一种将程序指令存储器和数据存储器合并在一起的计算结构，冯诺依曼提出了计算机制造的三个基本原则，即采用二进制逻辑、程序存储执行以及计算机由五个部分组成（运算器、控制器、存储器、输入设备、输出设备），这套理论被称为冯·诺依曼体系结构⁸。

⁸引自<https://baike.baidu.com/item/%u5e02>

4:



冯·诺依曼 (John von Neumann)

A von Neumann architecture machine, designed by physicist and mathematician John von Neumann (1903–1957) is a theoretical design for a stored program computer that serves as the basis for almost all modern computers. A von Neumann machine consists of a central processor with an arithmetic/logic unit and a control unit, a memory, mass storage, and input and output.⁹

2.6 算术逻辑单元 ALU

ALU 是 Arithmetic and Logic Unit 的缩写，中文通常翻译为“算术逻辑单元”，我们为了用最简单的例子来说明基本原理，下面我们举一个 1bit ALU 的示例。

首先我们要确定设计的 ALU 包括那些算术、逻辑单元，在这里我们假设包括三个单元：加、与、或非、异或，前面我们已经介绍了这四种运算的电路实现。

在每次执行时，我们还要告诉 ALU 执行那种操作，是加？与？或？异或？我们可以同时进行四种运算，然后选择一个输出，也就是我们可以使用数据选择器 (Multiplexer，简写为 MUX) 来实现运算的选择，电路如图2.13¹⁰所示，

我们假设 MUX 的逻辑如表2.2所示，这时整个 ALU 单元的输入为 M1 M0 B0 A0，如果我们从坐到右排列这四位，并且认为左为最高位，那么知道当输入的四比特是"00**" 形式时，就表示执行加运算，如输入"0010"，ALU 执行"1+0" 运算，而输入的"0010" 是 ALU 可以执行的编码，也就是我们通常说的

¹⁰ 图片来自于https://www.exploreembedded.com/wiki/ALU_in_Detail

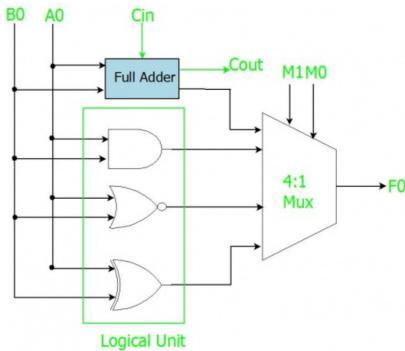


Figure 2.13: 1 bit ALU 逻辑电路图

“机器码”，下面我们把所有类型的机器码都写出来，如表2.3所示。

在上面的例子中，我们在不考虑优化的前提下，可以直接用 1 bit ALU 构建一个 4 bit ALU，如图2.14所示。

2.7 指令执行

下面我们在 1 bit ALU 的基础上构造一个 1 bit CPU(Central Processing Unit, 翻译为中央处理器，或，中央处理单元)，ALU 现在已经能够执行机器码了，根据冯诺依曼结构，CPU 需要完成取指从，然后进行执行，程序是顺序执行的，所以我们需要来记录指令存储的地址，每次执行完后，这个地址需要加 1，得到下一条指令的地址，我们把 CPU 具有存储功能的单元叫寄存器 (register)，而记录取指令的寄存器，我们称为程序计数器 (Program Counter，简写为 PC)，PC 的

Table 2.2: 1 bit ALU 中的 MUX 逻辑

M1	M0	F0 输出
0	0	全加器结果
0	1	与结果
1	0	或非结果
1	1	异或结果

Table 2.3: 1 bit ALU 可执行的机器码

运算名	机器码/指令集			
加	0	0	a	b
与	0	1	a	b
或非	1	0	a	b
异或	1	1	a	b

a,b 是运算的操作数，因为是 1bit，所以可能的取值为 0 或 1.

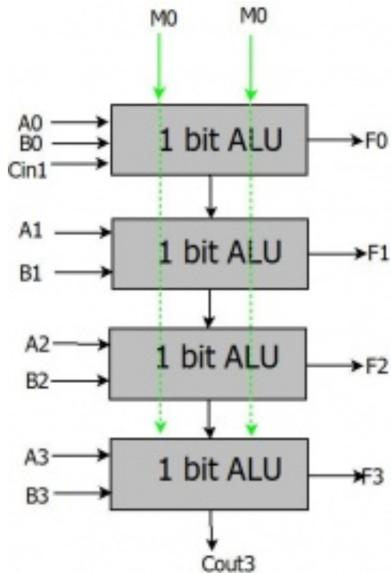


Figure 2.14: 由 1 bit ALU 组成 4 bit ALU 逻辑电路图¹⁰

位数决定了其可以寻址的范围。

2.7.1 PC 寄存器

我们假设 PC 寄存器是 3bit，这也就意味着 PC 的寻址能力为 $2^3 = 8$ ，也就是最多有 8 个存储单元，从上面我们 1 bit ALU 的机器指令设计中，我们可以看到一个指令为 4bit，我们可以将每个存储单元设计为 4 比特宽。

我们简化一下冯诺依曼体系架构，执行指令简化为两步，首先要根据 PC 中的地址取指令，用一个时钟，然后在 ALU 中执行，再用一个时钟，执行完一个指令，也就是两个时钟后，PC 要加一，接着再执行，实现框图如2.15所示。

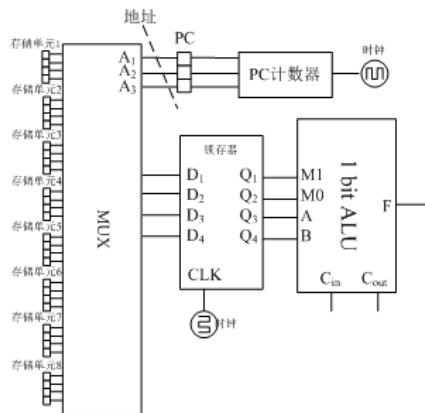


Figure 2.15: 由 1 bit ALU 组成的 1 bit CPU

2.8 现代 CPU 架构

从前面给出的一个极简 1 bit CPU 可以看出，要想让 CPU 能够进行更大规模运算、结构更加简单、运行速度更快、能够适应更广泛的运算，需要至少在以下几方面进行深化设计：

- 连接线路的复用——总线设计技术。
- 能够更快执行复杂运算——增加内部存储单元，我们称为寄存器，使得在运算能够直接在寄存器间进行，这也意味着需要有加载数据到寄存器的指令和寄存器运算指令。
- 执行过程中需要程序跳转——JMP 语句和函数调用，以及相应的现场保护思想、堆栈管理技术、增加相应的寄存器。
- 在执行中断时保存返回现场——现场保护思想、堆栈管理技术、增加相应的寄存器。
- 为了增加程序和数据读取速度——增加了 Cache(缓冲区)。

2.8.1 8086

8086¹¹是 1978 年英特尔 (Intel) 公司推出的世界上第一个 16 位微处理器，芯片上有 4 万个晶体管，采用 HMOS 工艺制造，用单一的 +5V 电源，时钟频率为 4.77MHz 10MHz。8086

¹¹ 8086 的英文介绍可参考 "Internal Architecture of 8086" <https://www.eeeguide.com/internal-architecture-of-8086/>

有 16 根数据线和 20 根地址线，它既能处理 16 位数据，也能处理 8 位数据，可寻址的内存空间为 1MB，每一个存储单元可以存放一个字节（8 位）二进制信息。¹²

8086 系统架构图如2.16所示，总线接口单元 (BIU, bus interface unit) 包含 4 个段地址寄存器、16 位的指令指针寄存器 IP、20 位的地址加法器、6 字节的指令队列缓冲器。其中 4 个段地址寄存器包含：CS(code segment)16 位的代码段寄存器；DS(data segment)16 位的数据段寄存器；ES(extra segment)16 位的扩展段寄存器；SS(stack segment)16 位的堆栈段寄存器。执行单元执行部件 (EU, execute unit) 包括 8 个通用寄存器 (AX、BX、CX、DX、BP、SP、SI、DI)、4 个数据寄存器 (AX、BX、CX、DX)、2 个地址指针寄存器 (BP,base pointer; SP, stack pointer)、2 个变址寄存器 (SI, source index; DI, destination index)、标志寄存器 FR(flags register)、算术逻辑单元 ALU(arithmetic logic unit)。EU 负责全部指令的执行，同时向 BIU 输出数据（操作结果），并对寄存器和标志寄存器进行管理。在 ALU 中进行 16 位运算，数据传送和处理均在 EU 控制下执行。¹²

BIU 和 EU 的协同：

1. BIU 和 EU 可以并行工作，提高 CPU 效率。BIU 监视着指令队列。当指令队列中有 2 个空字节时，就自动把指令取到队列中。
2. EU 执行指令时，从指令队列头部取指令，然后执行。如

¹² 内容引自<https://baike.baidu.com/item/8086>

需访问存储器，则 EU 向 BIU 发出请求，由 BIU 访问存储器。

3. 在执行转移、调用、返回指令时，需改变队列中的指令，要等新指令装入队列中后，EU 才继续执行指令。

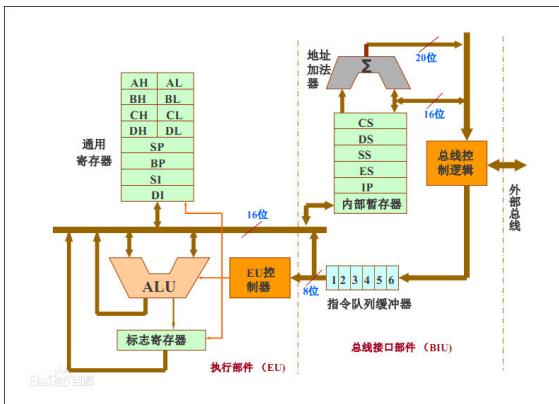


Figure 2.16: 8086 系统架构图¹²

8086 的封装采用双列直插 (通常说的 DIP, dual in-line package), 如图2.17所示，同一芯片会有不同的封装，虽然封装里面的集成电路模块是一样的，有时同一集成电路的不同封装，根据应用的要求，在管脚的引出上会有些许的差别。

2.8.2 Intel Skylake (client) 微架构

随着 CPU 不断加入新的加速执行机制，CPU 的性能不断提高，CPU 的结构也越来越复杂，CPU 已经是一个微系统，这种复杂的 CPU 通常称为微架构 (Microarchitectures)。Intel 的微架构及其对应工艺有

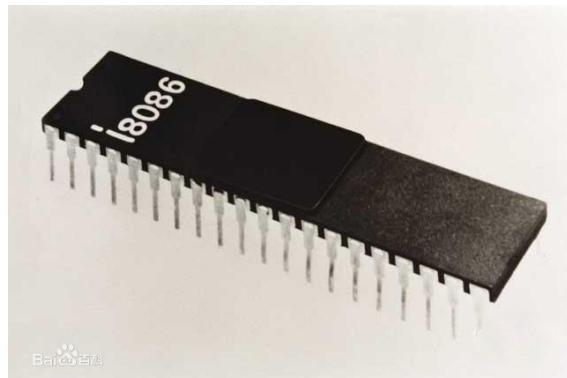


Figure 2.17: 8086 系统架构图¹²

P6(65nm),core(65nm,45nm),Nehalem(45nm,32nm),Sandy Bridge(32nm,22nm),Haswell(22nm,14nm),Skylake(14nm,10nm),Ice Lake(10nm,7nm)。¹³

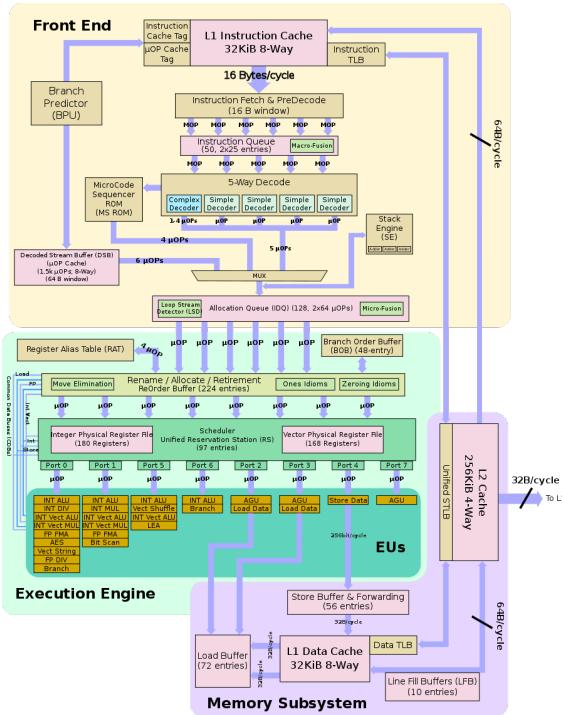
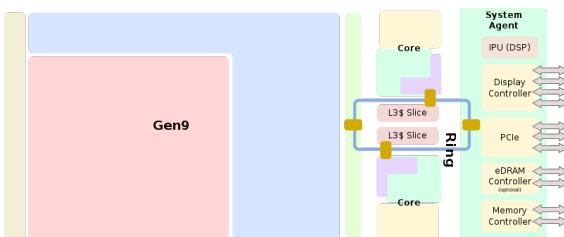
下面我们看看 skylake 的微架构¹⁴，其单核微架构如图2.18所示。

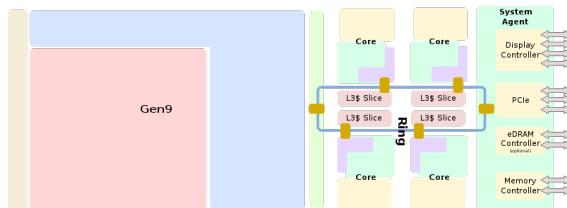
Intel 的 Celeron CPU 就是利用 Skylak(client) 微架构搭建的双核 SOC，总体框图如图2.19所示。

Intel 的 Core i3 CPU 就是利用 Skylak(client) 微架构搭建的四核 SOC，总体框图如图2.20所示。

¹³此列表信息摘录自<https://jcf94.com/2018/02/13/2018-02-13-intel/>, 此贴的信息来源其称来自维基百科。

¹⁴详细介绍可参考[https://en.wikichip.org/wiki/intel/microarchitectures/skylake_\(client\)](https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client))

Figure 2.18: Skylake(client) 微架构¹⁴Figure 2.19: Skylake(client) 双核 SOC 框图¹⁴

Figure 2.20: Skylake(client) 四核 SOC 框图¹⁴

2.9 启动

计算机在上电后，CPU 的程序执行寄存器都有一个初始值，这个系统的执行就是从这里开始，而这些信息是需要从 CPU 的设计厂家中获得。在解释计算机启动过程之前，我们先看两个基本概念¹⁵。

- BIOS, 全称 **Basic Input/Output System**, 基本输入输出系统

BIOS 是直接与硬件打交道的底层代码，它为操作系统提供了控制硬件设备的基本功能。BIOS 包括有系统 BIOS（即常说的主板 BIOS）、显卡 BIOS 和其它设备（例如 IDE 控制器、SCSI 卡或网卡等）的 BIOS，其中系统 BIOS 是本文要讨论的主角，因为计算机的启动过程正是在它的控制下进行的。BIOS 一般被存放在 ROM(只读存储芯片)之中，即使在关机或掉电以后，这些代码也不会消失。

- 内存的地址

¹⁵ tham 的“操作系统启动过程——启动引导 + 硬件自检 + 系统引导 + 系统加载 + 系统登录”，网址<https://www.cnblogs.com/tham/p/6827151.html>

内存的每一个字节都被赋予了一个地址，以便 CPU 访问内存。32MB 的地址范围用十六进制数表示就是 0 ~ 1FFFFFFH 其中 0~FFFFFH 的低端 1MB 内存非常特殊，因为最初的 8086 处理器能够访问的内存最大只有 1MB，这 1MB 的低端 640KB 被称为基本内存，而 A0000H~BFFFFH 要保留给显示卡的显存使用，C0000H~FFFFFH 则被保留给 BIOS 使用，其中系统 BIOS 一般占用了最后的 64KB 或更多一点的空间，显卡 BIOS 一般在 C0000H~C7FFFH 处，IDE 控制器的 BIOS 在 C8000H~CBFFFH 处。

下面我们来看看计算机启动的基本流程^{15 16}:

1. 当我们按下电源开关时，电源就开始向主板和其它设备供电，此时电压还不太稳定，主板上的控制芯片组会向 CPU 发出并保持一个 RESET（重置）信号，让 CPU 内部自动恢复到初始状态，但 CPU 在此刻不会马上执行指令。当芯片组检测到电源已经开始稳定供电了（当然从不稳定到稳定的过程只是一瞬间的事情），它便撤去 RESET 信号（如果是手工按下计算机面板上的 Reset 按钮来重启机器，那么松开该按钮时芯片组就会撤去 RESET 信号），CPU 马上就从地址 FFFF0H 处开始执行指令，从前面的介绍可知，这个地址实际上在系统 BIOS 的地址范围内，无论是 Award BIOS 还是 AMI BIOS，放在里面的只是一条跳转指令，跳到系统 BIOS 中真正的启动代码处。

¹⁶ 郑瀚 Andrew_Hann 博客的帖子“Windows 启动过程 (MBR 引导过程分析)”，网址<https://www.cnblogs.com/LittleHann/p/6974928.html>

2. 系统 BIOS 的启动代码首先要做的事情就是进行 POST (Power — On Self Test, 加电后自检), POST 的主要任务是检测系统中一些关键设备是否存在和能否正常工作, 例如内存和显卡等设备。由于 POST 是最早进行的检测过程, 此时显卡还没有初始化, 如果系统 BIOS 在进行 POST 的过程中发现了一些致命错误, 例如没有找到内存或者内存有问题 (此时只会检查 640K 常规内存), 那么系统 BIOS 就会直接控制喇叭发声来报告错误, 声音的长短和次数代表了错误的类型。在正常情况下, POST 过程进行得非常快, 我们几乎无法感觉到它的存在, POST 结束之后就会调用其它代码来进行更完整的硬件检测。
3. 接下来系统 BIOS 将查找显卡的 BIOS, 前面说过, 存放显卡 BIOS 的 ROM 芯片的起始地址通常设在 C0000H 处, 系统 BIOS 在这个地方找到显卡 BIOS 之后就调用它的初始化代码, 由显卡 BIOS 来初始化显卡, 此时多数显卡都会在屏幕上显示出一些初始化信息, 介绍生产厂商、图形芯片类型等内容, 不过这个画面几乎是一闪而过。系统 BIOS 接着会查找其它设备的 BIOS 程序, 找到之后同样要调用这些 BIOS 内部的初始化代码来初始化相关的设备。
4. 查找完所有其它设备的 BIOS 之后, 系统 BIOS 将显示出它自己的启动画面, 其中包括有系统 BIOS 的类型、序列号和版本号等内容。
5. 接着系统 BIOS 将检测和显示 CPU 的类型和工作频率,

然后开始测试所有的 RAM，并同时在屏幕上显示内存测试的进度，我们可以在 CMOS 设置中自行决定使用简单耗时少或者详细耗时多的测试方式。

6. 内存测试通过之后，系统 BIOS 将开始检测系统中安装的一些标准硬件设备，包括硬盘、CD - ROM、串口、并口、软驱等设备，另外绝大多数较新版本的系统 BIOS 在这一过程中还要自动检测和设置内存的定时参数、硬盘参数和访问模式等。
7. 标准设备检测完毕后，系统 BIOS 内部的支持即插即用的代码将开始检测和配置系统中安装的即插即用设备，每找到一个设备之后，系统 BIOS 都会在屏幕上显示出设备的名称和型号等信息，同时为该设备分配中断、DMA 通道和 I/O 端口等资源。
8. 到此，硬件设备检测完毕，系统 BIOS 会重新清屏并在屏幕上显示出一个系统配置表，其中概略列出系统中安装的各种标准硬件设备，及他们使用的资源和一些相关参数。
9. 接下来系统 BIOS 将更新 ESCD (Extended System Configuration Data，扩展系统配置数据)。ESCD 是系统 BIOS 用来与操作系统交换硬件配置信息的一种手段，这些数据被存放在 CMOS (一小块特殊的 RAM，由主板上的电池来供电) 之中。通常 ESCD 数据只在系统硬件配置发生改变后才会更新，所以不是每次启动机器时我们都能够看到“Update ESCD…Success”这样的信息，不过，某些主板的系统 BIOS 在保存 ESCD

数据时使用了与 Windows 系统不相同的数据格式，于是 Windows 在它自己的启动过程中会把 ESCD 数据修改成自己的格式，但在下一次启动机器时，即使硬件配置没有发生改变，系统 BIOS 也会把 ESCD 的数据格式改回来，如此循环，将会导致在每次启动机器时，系统 BIOS 都要更新一遍 ESCD，这就是为什么有些机器在每次启动时都会显示出相关信息的原因。

10. ESCD 更新完毕后，系统 BIOS 的启动代码将进行它的最后一项工作，即根据用户指定的启动顺序从软盘、硬盘或光驱启动。

- BIOS 将磁盘第一个物理扇区 (MBR^⑤) 加载到内存。
- 接着将系统控制权交给 MBR 来进行 (其实就是跳转到加载到内存中 MBR 程序部分)。
- MBR 运行后，搜索 MBR 中的分区表，查找活动分区 (Active Partition) 的起始位置。MBR 将活动分区的第一个扇区中的引导扇区 (分区引导记录) 载入到内存。MBR 检测当前使用的文件系统是否可用。
- MBR 查找操作系统的启动器^⑥，并将控制权转交给启动器，由启动器完成操作系统的启动。

下面我们摘录 Intel 的一篇技术文档 “Minimal Intel Architecture Boot Loader”¹⁷ 中的内容，对计算机的启动有一个

¹⁷<https://www.intel.com/content/www/us/en/intelligent-systems/intel-boot-loader-development-kit/minimal-intel-architecture-boot-loader.html>

5: MBR(Master Boot Record)——主引导记录，位于启动磁盘的第一个扇区，其中主要包含引导代码(Boot Code)和分区表(Partition Table)数据。引导代码主要用于引导系统。而分区表则主要用于标识基本分区和扩展分区。

6: 微软基于 NT 内核的操作系统启动器是 ntldr 文件；DOS 和 win9x 的系统启动器就是分区引导记录，分区引导记录将负责读取并执行 IO.SYS，Windows 9x 的 IO.SYS 首先要初始化一些重要的系统数据，然后就显示出我们熟悉的蓝天白云，在这幅画面之下，Windows 将继续进行 DOS 部分和 GUI (图形用户界面) 部分的引导和初始化工作。

大致概况了解。

Power-Up (Reset Vector) Handling

When an IA bootstrap processor (BSP) powers on, the first address that is fetched and executed is at physical address 0xFFFFFFFF0, also known as the reset vector. This accesses the ROM / Flash device at the top of the ROM -0x10. The boot loader must always contain a jump to the initialization code in these top 16 bytes.

Mode Selection

The processor must be placed into one of the following modes:

- Real Mode
- Flat Protected Mode
- Segmented Protected Mode (Not Recommended for Firmware)

Refer to the Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A section titled "Mode Switching" for more details.

2.9.1 Firmware Support Package (FSP)

Firmware Support Package (FSP) 是固件支持包，通常由 CPU 的厂家提供，包含了初始化 CPU 的程序，可以用来供开发者快速开发启动程序。

下面是 Intel 关于 FSP 的描述。

Intel Firmware Support Package (Intel FSP) provides key programming information for initializing Intel silicon and can be easily integrated into a boot loader of the developer's choice. It is easy to adopt, scalable to design, reduces time-to-market, and is economical to build. Components include:

- CPU, memory controller, and Intel® chipset initialization functions as a binary package: Provides silicon initialization ingredients, preserves existing features and frameworks, and fits into existing boot loaders
- Integration guide: Describes the APIs available to communicate with Intel FSP and to integrate it with a boot-loader solution

Intel FSP is designed for integration into a variety of boot loaders, including coreboot and TianoCore (open source Unified Extensible Firmware Interface (UEFI)).¹⁸

¹⁸此 FSP 介绍引自<https://software.intel.com/content/www/us/en/develop/articles/intel-firmware-support-package.html>

2.9.2 UEFI

Unified Extensible Firmware Interface (UEFI) 是统一可扩展固件接口规范，其前身是 Extensible Firmware Interface (EFI)，其定义了操作系统和硬件平台固件之间的接口，这个接口包括数据表和一些操作系统可以调用的程序接口。

下面是 Intel 网站上关于 UEFI 的介绍材料。

The Unified Extensible Firmware Interface (UEFI) Specification, previously known as the Extensible Firmware Interface (EFI) Specification, defines an interface between an operating system and platform firmware. The interface consists of data tables that contain platform-related information, boot service calls, and runtime service calls that are available to the operating system and its loader. These provide a standard environment for booting an operating system and running pre-boot applications. The UEFI Specification was primarily intended for the next generation of Intel® architecture-based computers, and is an outgrowth of the Intel Boot Initiative (IBI) program that began in 1998. Intel's original version of this specification was publicly named EFI, ending with the EFI 1.10 version.

In 2005, The Unified EFI Forum was formed as an industry-wide organization to promote adoption and continue the development of the EFI Specification. Using the EFI 1.10 Specification as the starting point, this industry group released the follow on specifications, renamed Uni-

fied EFI.

Find out more information about UEFI, the UEFI Forum, and the current version of the UEFI Specification at the UEFI Forum Website.¹⁹ ²⁰

¹⁹此英文介绍引自<https://software.intel.com/content/www/us/en/develop/articles/unified-extensible-firmware-interface.html>

²⁰UEFI Forum Website <https://uefi.org/>

Chapter 3

电子设计自动化 (EDA: Electronic Design Automation)

本章介绍的主要工具框架来自“IC 设计完整流程及工具”¹.

3.1 电路逻辑设计

3.1.1 HDL/VHDL

HDL(Hardware Description Language) 是硬件描述语言，是对硬件电路进行行为描述、寄存器传输描述或者结

¹<https://zhuanlan.zhihu.com/p/87314723>

构化描述的一种语言，我们常用的两种硬件描述语言有 VHDL(Very High-Speed Integration Circuit HDL) 和 Verilog。

VHDL 诞生于 1982 年。在 1987 年底，VHDL 被 IEEE 和美国国防部确认为标准硬件描述语言。自 IEEE 公布了 VHDL 的标准版本，IEEE-1076（简称 87 版）之后，各 EDA 公司相继推出了自己的 VHDL 设计环境，或宣布自己的设计工具可以和 VHDL 接口。此后 VHDL 在电子设计领域得到了广泛的接受，并逐步取代了原有的非标准的硬件描述语言。

Verilog HDL 是由 GDA(Gateway Design Automation) 公司的 Phil Moorby 在 1983 年末首创的，最初只设计了一个仿真与验证工具，之后又陆续开发了相关的故障模拟与时序分析工具。1985 年 Moorby 推出它的第三个商用仿真器 Verilog-XL，获得了巨大的成功，从而使得 Verilog HDL 迅速得到推广应用。1989 年 CADENCE 公司收购了 GDA 公司，使得 Verilog HDL 成为了该公司的独家专利。1990 年 CADENCE 公司公开发表了 Verilog HDL，并成立 LVI 组织以促进 Verilog HDL 成为 IEEE 标准，即 IEEE Standard 1364-1995。²

3.1.2 门电路(图)

门电路是用以实现基本逻辑运算和复合逻辑运算的单元电路。常用的门电路在逻辑功能上有与门、或门、非门、与非门、或非门、与或非门、异或门等几种。

²文字介绍来源于 <http://yanqin.spaces.eepw.com.cn/articles/article/item/50571>

另外还有一个常用的基本门电路叫传输门，可以模拟“开关”的动作，物理上通过 MOS-FET 实现，利用了其栅电压控制 MOS 管导通的原理；当 CP 为 1，A 的数据可以传到 B 端，当 CP 为 0 时，其内部晶体管截止，可以把电路中的通路临时关断。

实际应用的门电路都是集成电路。在集成电路设计过程中，将复杂的逻辑函数转换为具体的数字电路时，不管是手工设计还是 EDA 工具自动设计，电路都可以用七种基本逻辑（与、或、非、与非、或非、同或、异或）的图形表示，在电路术语中这些逻辑操作符号被称作门，对应的具体电路就叫做门电路。

利用门电路可以组成任意复杂的逻辑功能电路。³

3.2 仿真验证 (逻辑)

仿真验证工具 Mentor 公司的 Modelsim，Synopsys 的 VCS，还有 Cadence 的 NC-Verilog 均可以对 RTL 级的代码进行设计验证，该部分个人一般使用第一个-Modelsim。该部分称为前仿真，接下来逻辑部分综合之后再一次进行的仿真可称为后仿真。

³ 如何利用门电路设计一个逻辑功能电路，这是首先要解决的问题，在《数字逻辑电路》课程里，我们会学习一些基本方法，比如利用真值表设计逻辑电路的门电路表示，但是紧接着会有另外一个问题，就是“化简问题”，如果用最少的门电路实现？硬件描述语言的综合过程，就是生成此硬件描述语言所描述的逻辑功能的门电路表示，里面其实已经解决了“化简”问题。

3.3 逻辑综合

仿真验证通过，进行逻辑综合。逻辑综合的结果就是把设计实现的 HDL 代码翻译成门级网表 netlist。综合需要设定约束条件，就是你希望综合出来的电路在面积，时序等目标参数上达到的标准。逻辑综合需要基于特定的综合库，不同的库中，门电路基本标准单元（standard cell）的面积，时序参数是不一样的。所以，选用的综合库不一样，综合出来的电路在时序，面积上是有差异的。一般来说，综合完成后需要再次做仿真验证（这个也称为后仿真，之前的称为前仿真）逻辑综合工具 Synopsys 的 Design Compiler，仿真工具选择上面的三种仿真工具均可。

3.4 形式验证

这也是验证范畴，它是从功能上（STA 是时序上）对综合后的网表进行验证。常用的就是等价性检查方法，以功能验证后的 HDL 设计为参考，对比综合后的网表功能，他们是否在功能上存在等价性。这样做是为了保证在逻辑综合过程中没有改变原先 HDL 描述的电路功能。形式验证工具有 Synopsys 的 Formality。前端设计的流程暂时写到这里。从设计程度上来讲，前端设计的结果就是得到了芯片的门级网表电路。

3.5 可测性设计 (DFT:Design ForTest)

芯片内部往往都自带测试电路，DFT 的目的就是在设计的时候就考虑将来的测试。DFT 的常见方法就是，在设计中插入扫描链，将非扫描单元（如寄存器）变为扫描单元。关于 DFT，有些书上有详细介绍，对照图片就好理解一点。DFT 工具 Synopsys 的 DFT Compiler。

3.6 布局规划 (Floor Plan)

布局规划就是放置芯片的宏单元模块，在总体上确定各种功能电路的摆放位置，如 IP 模块，RAM，I/O 引脚等等。布局规划能直接影响芯片最终的面积。工具为 Synopsys 的 Astro.

3.7 时钟树综合 (CTS: Clock Tree Synthesis)

简单点说就是时钟的布线。由于时钟信号在数字芯片的全局指挥作用，它的分布应该是对称式的连到各个寄存器单元，从而使时钟从同一个时钟源到达各个寄存器时，时钟延迟差异最小。这也是为什么时钟信号需要单独布线的原因。CTS 工具，Synopsys 的 Physical Compiler.

3.8 布线 (Place & Route)

这里的布线就是普通信号布线了，包括各种标准单元（基本逻辑门电路）之间的走线。比如我们平常听到的 0.13um 工艺，或者说 90nm 工艺，实际上就是这里金属布线可以达到的最小宽度，从微观上看就是 MOS 管的沟道长度。工具 Synopsys 的 Astro.

3.9 寄生参数提取

由于导线本身存在的电阻，相邻导线之间的互感，耦合电容在芯片内部会产生信号噪声，串扰和反射。这些效应会产生信号完整性问题，导致信号电压波动和变化，如果严重就会导致信号失真错误。提取寄生参数进行再次的分析验证，分析信号完整性问题是十分重要的。工具 Synopsys 的 Star-RCXT.

3.10 版图物理验证

对完成布线的物理版图进行功能和时序上的验证，验证项目很多，如 LVS (Layout Vs Schematic) 验证，简单说，就是版图与逻辑综合后的门级电路图的对比验证；DRC (Design Rule Checking)：设计规则检查，检查连线间距，连线宽度等是否满足工艺要求，ERC (Electrical Rule Checking)：电气规则检查，检查短路和开路等电气规则违例；等等。工具为 Synopsys 的 Hercules 实际的后端流程还包括电路功耗分析，以及随着制造工艺不断进步产生的 DFM (可制造性设计) 问题，在此不说了。物理版图验证完成也就是整个芯片设计阶

段完成，下面的就是芯片制造了。物理版图以 GDSII 的文件格式交给芯片代工厂（称为 Foundry）在晶圆硅片上做出实际的电路，再进行封装和测试，就得到了我们实际看见的芯片。



Chapter 4

机器码 / 指令集

机器码就是 CPU 能够直接执行的二进制串，我们把机器能执行的一个二进制串就称为一条机器指令。指令集就是 CPU 所能执行的所有机器指令的集合。

我们也可以换个角度来看指令集，指令集是在说明如何用软件操控 CPU。

有很多种指令集定义，而这些指令集定义是有知识产权的。

图4.1是 ARM 官网上的截图，其说明 ARM 支持的指令集，每个指令集都会衍生出来一系列或者几个系列的 CPU 芯片¹。

¹初学者注意理解这句话，这句话的意思是：定义了一个指令集，而我可以用各种不同的设计来实现这个指令集，只要 CPU 能够执行这些指令，我们就说他支持这个指令集，所以两个 CPU 支持同样的指令集，但是他们运

Arm Instruction Set Architecture

The Arm architecture supports three instruction sets: A64, A32 and T32.

- The A64 and A32 instruction sets have fixed instruction lengths of 32 bits.
- The T32 instruction set was introduced as a supplementary set of 16-bit instructions that supported improved code density for user code. Over time, T32 evolved into a 16-bit and 32-bit mixed-length instruction set. As a result, the compiler can balance performance and code size trade-off in a single instruction set.

Explore these instruction sets:



Figure 4.1: ARM 实现的指令集

在 AMD 开发者网站 (<http://developer.amd.com/>) 中，其对 AMD 64 架构的 CPU 的指令集描述如下²:

1.1.3 Instruction Set

The AMD64 architecture supports the full legacy x86 instruction set, with additional instructions to support long mode (see Table 1-1 on page 2 for a summary of operating modes). The application- programming instructions are organized into four subsets, as follows:

General-Purpose Instructions—These are the basic x86 integer instructions used in virtually all programs. Most of these instructions load, store, or operate on data located in the general-purpose registers (GPRs) or memory. Some of the instructions alter sequential program flow by branching

行效率可能差别很大

² 来源于文档“AMD64 Architecture Programmer’s Manual-Volumes 1-5”，文档地址:<https://www.amd.com/system/files/TechDocs/40332.pdf>

to other program locations.

Streaming SIMD Extensions Instructions (SSE)—These instructions load, store, or operate on data located primarily in the YMM/XMM registers. 128-bit media instructions operate on the lower half of the YMM registers. SSE instructions perform integer and floating-point operations on vector (packed) and scalar data types. Because the vector instructions can independently and simultaneously perform a single operation on multiple sets of data, they are called single-instruction, multiple-data (SIMD) instructions. They are useful for high-performance media and scientific applications that operate on blocks of data.

Multimedia Extension Instructions—These include the MMX technology and AMD 3DNow! technology instructions. These instructions load, store, or operate on data located primarily in the 64-bit MMX registers which are mapped onto the 80-bit x87 floating-point registers. Like the SSE instructions, they perform integer and floating-point operations on vector (packed) and scalar data types. These instructions are useful in media applications that do not require high precision. Multimedia Extension Instructions use saturating mathematical operations that do not generate operation exceptions. AMD has de-emphasized the use of 3DNow! instructions, which have been superceded by their more efficient SSE counterparts. Relevant recommendations are provided in Chapter 5, “64-Bit Media Programming” on

page 239, and in the AMD64 Programmer’s Manual Volume 4: 64-Bit Media and x87 Floating-Point Instructions.

x87 Floating-Point Instructions—These are the floating-point instructions used in legacy x87 applications. They load, store, or operate on data located in the 80-bit x87 registers.

龙芯的 CPU 芯片开始使用的 MIPS 指令集架构，2019 年 MIPS 开源，这虽然对于龙芯是利好消息，但是龙芯很注重核心知识产权的自主研发，所以并没有停止其研发的步伐，龙芯在 2020 年 8 月 13 日的全国计算机体系结构学术年会（ACA2020）上，龙芯中科董事长、中科院计算技术研究所研究员胡伟武作了名为《指令系统的自主与兼容》的特邀报告。在报告中，他透露了龙芯的新动向——研发既“自主”又“兼容”的 LoongArch 指令集。如果最终达成目标，这将是一个自带“完整”生态，且中国人能牢牢掌握的体系³。

2020 年，龙芯中科基于二十年的 CPU 研制和生态建设积累推出了龙芯指令系统（LoongArch³），包括基础架构部分和向量指令、虚拟化、二进制翻译等扩展部分，近 2000 条指令。龙芯指令系统不包含 MIPS 指令系统。龙芯指令系统具有较好的自主性、先进性与兼容性。龙芯指令系统从整个架构的顶层规划，到各部分的功能定义，再到细节上每条指令的编码、名称、含义，在架构上进行自主重新设计，具有充分的自主性。龙芯指令系统摒弃了传统指令系统中部分不适

³丢掉幻想！龙芯中科将出 LoongArch 自主指令集，深度兼容主流体系，网址<https://baijiahao.baidu.com/s?id=1675553831699313296&wf=r=spider&for=pc>

应当前软硬件设计技术发展趋势的陈旧内容，吸纳了近年来指令系统设计领域诸多先进的技术发展成果。同原有兼容指令系统相比，不仅在硬件方面更易于高性能低功耗设计，而且在软件方面更易于编译优化和操作系统、虚拟机的开发。⁴

龙芯架构 32 为精简版中的所有指令均采用 32 位固定长度，且指令的地址要求 4 字节边界对齐。当指令地址不对齐时将触发地址错例外。指令编码的风格是所有寄存器操作数域都从第 0 比特开始从低到高依次摆放。操作码都是从 31 比特开始从高到低依次摆放。如果指令中含有立即数操作数，那么立即数域位于寄存器域和操作码域之间，根据不同指令类型有不同的长度。具体来说，包含 9 中典型的指令格式编码，即 3 种不含立即数的编码格式 2R、3R、4R，以及 6 中含立即数的编码格式 2RI8、2RI12、2RI14、2RI16、1RI21、I26，图4.2列举了这九种典型编码格式的具体定义，图4.3截取了指令码一览表的一部分，其中红框里面是逻辑“与”、“或”、“异或”指令码。⁵

⁴ 来源于<http://www.loongson.cn/loongArch>

⁵ 参考“龙芯架构 32 位精简版参考手册”，手册下载地址在龙芯官网<http://www.loongson.cn/FileShow>

Figure 4.2: 龙芯架构 32 位精简版典型指令编码格式

Figure 4.3: 龙芯架构 32 位精简版指令码表截取，红框部分是逻辑“与”、“或”、“异或”指令。

Chapter 5

汇编语言

直接用机器码或者机器指令进行编程会是一个很繁琐的过程(并不是不可以),早期人们用纸带编程,其实就是在纸带上通过有孔、无孔表示0或1,直接在纸带上“扎出”机器码,而计算机从纸带读入指令,从而进行运算。



Elliott 903 计算机(纸带)

纸带数据读入

Figure 5.1: Elliott 903 纸带计算机数据的读入

为了提高开发效率，人们用助记符代替机器指令的操作码，用地址符号或标号代替指令或操作数的地址，汇编语言对应着不同的机器指令，特定的汇编语言和特定的机器语言指令集是一一对应的，不同平台¹之间不可直接移植，汇编语言与机器指令相比，人书写更方便，但是需要通过汇编器(Assembler)将汇编语句转换成机器指令，用汇编语言写的程序通常会有多个，人们编写的一套工具来完成汇编程序到机器指令序列的转换，这个工具我们称为编译器，有不同的汇编语言编译器，如 MASM、NASM 等。当然还有高级语言的编译器，比如 C 语言的编译器 gcc。图4.3是龙芯指令集 LoongsonArch 一览表的部分截取，可以看到每条机器码都对应一条汇编指令。为了方便编程，汇编语言还会定义一些其他语句，如伪指令、段定义、宏定义等。

我们通常还会看到一个名词“编辑器(Editor)”，编辑器通常只用来编辑文本的软件，比如 windows 操作系统自带的 notepad，以及第三方的 notepad plus 等。

还有一个概念就是 IDE，英文全称是 integrated development environment，翻译过来就是“集成开发环境”，从字面意思就可以猜到，这个软件就是把你开发中用到的软件或者功能都集成在一起了，这类软件有 visual Studio、Eclipse、code::block 等，通常一个开发环境会针对一种或者几种开发语言。另外需要强调的一点是，程序的调试是非常重要的，所以通常 IDE 中会集成调试工具，使得程序调试方便些。

下面是一个在命令行输出“Hello World”的完整的汇编

¹此处平台是指不同指令集

程序，操作系统是 Linux，编译器是 Nasm。

```

1 ; Define variables in the data section
2 SECTION .DATA
3     hello:    db 'Hello world!',10
4     helloLen: equ $-hello
5
6 ; Code goes in the text section
7 SECTION .TEXT
8     GLOBAL _start
9
10 _start:
11     mov eax,4          ; 'write' system call = 4
12     mov ebx,1          ; file descriptor 1 = STDOUT
13     mov ecx,hello      ; string to write
14     mov edx,helloLen   ; length of string to write
15     int 80h             ; call the kernel
16
17     ; Terminate program
18     mov eax,1          ; 'exit' system call
19     mov ebx,0          ; exit with error code 0
20     int 80h             ; call the kernel

```

编译和运行过程如下所示。

```

1 # 编译(Compile)
2 nasm -f elf64 hello.asm -o hello.o
3
4 # 链接(Link)
5 ld hello.o -o hello
6
7 # 运行(Run)
8 ./hello

```



Chapter 6

数据存储

6.1 寄存器 (Register)

寄存器是 CPU 内部用来存放数据的存储区域，用来暂时存放参与运算的数据和运算结果。其实寄存器就是一种常用的时序逻辑电路，但这种时序逻辑电路只包含存储电路。寄存器的存储电路是由锁存器或触发器构成的，因为一个锁存器或触发器能存储 1 位二进制数，所以由 N 个锁存器或触发器可以构成 N 位寄存器。寄存器是中央处理器内的组成部分，是高速存储部件，它们可用来暂存指令、数据和位址。

CPU 内部的寄存器数量通常比较少，ARM 7 的寄存器只有 38 个。

6.2 高速缓存 (cache)

Cache 存储器是位于 CPU 与内存间的一种容量较小但速度很高的存储器，通常由 SRAM (Static Random Access Memory 静态存储器) 组成。CPU 的速度远高于内存，当 CPU 直接从内存中存取数据时要等待一定时间周期，而 Cache 则可以保存 CPU 刚用过或循环使用的一部分数据，如果 CPU 需要再次使用该部分数据时可从 Cache 中直接调用，这样就避免了重复存取数据，减少了 CPU 的等待时间，因而提高了系统的效率。Cache 又分为 L1Cache (一级缓存) 和 L2Cache (二级缓存)，L1Cache 主要是集成在 CPU 内部，而 L2Cache 集成在主板上或是 CPU 上。

Cache 的使用是系统根据设计时确定的算法自行进行的，无需干预。

6.3 内存 (memory)

内存 (Memory) 是计算机的重要部件之一，也称内存储器和主存储器，它用于暂时存放 CPU 中的运算数据，与硬盘等外部存储器交换的数据。它是外存与 CPU 进行沟通的桥梁，计算机中所有程序的运行都在内存中进行，内存性能的强弱影响计算机整体发挥的水平。只要计算机开始运行，操作系统就会把需要运算的数据从内存调到 CPU 中进行运算，当运算完成，CPU 将结果传送出来。

内存按工作原理分类，分为只读存储器 (Read Only Memory，缩写为 ROM) 和随机存储器 (Random Access Memory，

缩写为 RAM)。ROM 一般用于存放计算机的基本程序和数据，如 BIOS ROM，现在常用的只读存储器是闪存 (Flash Memory)，可以控制反复擦写。现在大部分 BIOS 程序就存储在 Flash ROM 芯片中，U 盘和固态硬盘 (SSD) 也是利用闪存原理做成的。随机存储器表示既可以从中读取数据，也可以写入数据，当机器电源关闭时，存于其中的数据就会丢失，我们通常购买或升级的内存条就是用作电脑的内存。

6.4 外部存储 (external storage)

6.4.1 外设访问

可以终端执行 “mmc devmgmt.msc” 看一下控制面板中键盘这个设备的属性。CPU 对外设的访问也是读、写两个操作，一种方式是讲外设的访问看成是与内存的访问方式一样，也就是将外设中存储数据的寄存器映射到内存的地址空间，这样 CPU 可以对其直接读写，第二种方式是外设的内存存储数据的寄存器独立编址，这是需要专用的 CPU 指令来访问。Intel X86 系列的 CPU 设计时，I/O 的访问采用独立编址，ARM 体系结构采用的是内存映射方式。

6.4.2 硬盘 (harddisk)

机械硬盘

机械硬盘即是传统普通硬盘，主要由：盘片，磁头，盘片转轴及控制电机，磁头控制器，数据转换器，接口，缓存等几个部分组成。机械硬盘有多个盘片，机械硬盘中所有的盘片都装在一个旋转轴上，每张盘片之间是平行的，在每个盘片

的存储面上有一个磁头，磁头与盘片之间的距离只有 $0.1\text{m}\sim 0.5\text{m}$ ，较高的水平已经达到 $0.005\text{m}\sim 0.01\text{m}$ 。磁头可沿盘片的半径方向运动，加上盘片每分钟几千转的高速旋转，磁头就可以定位在盘片的指定位置上进行数据的读写操作。信息通过离磁性表面很近的磁头，由电磁流来改变极性方式被电磁流写到磁盘上，信息可以通过相反的方式读取。硬盘作为精密设备，尘埃是其大敌，所以进入硬盘的空气必须过滤。图6.1是机械硬盘内部构造¹²。

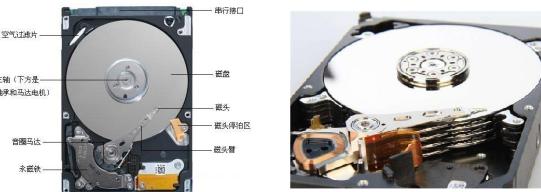


Figure 6.1: 机械硬盘内部结构

固态硬盘 (SSD:Solid State Disk)

固态硬盘是一种电子硬盘，其内部没有机械结构，是利用存储芯片构建的外部存储，构成固态硬盘的存储芯片有一个必备条件，就是要满足掉电非易失。目前固态硬盘的存储介质分为两种，一种是采用闪存（FLASH 芯片）作为存储介质，另外一种是采用 DRAM 作为存储介质。固态硬盘在接口的规范和定义、功能及使用方法上与普通硬盘的完全相同。图6.2是在某购物网站上销售的两款 SSD 硬盘³。

¹ 图片来自 <http://pic.baike.soso.com/ugc/baikepic2/25617/20170515140722-894063905.jpg/0>

² 图片来自 http://www.lotpc.com/yjzs/5362_2.html

³ 信息采集于 2021 年 9 月 17 日



Figure 6.2: 机械硬盘内部结构

6.4.3 光盘 (Compact Disc)

高密度光盘 (Compact Disc, 简写为 CD) 通常简称为光盘, 是一种光学存储介质, 利用激光原理进行读、写的设备, 可以存放各种文字、声音、图形、图像和动画等任意数字信息。CD 光盘容量通常为 650MB/700MB/800MB/890MB,DVD 光盘的容量通常为 4.7GB/8.6GB, 分不可擦写光盘, 如 CD-ROM、DVD-ROM 等, 可擦写光盘, 如 CD-RW、DVD-RAM 等。

光驱是读写光盘的设备, 通常分为内置型和外置型, 光驱也分为只读型和可读性型。

图6.3是一款外置的可读性光驱和光盘的图片。



Figure 6.3: 外置光驱和光盘

6.4.4 磁带 (magnetic tape storage)

磁带存储器 (magnetic tape storage) 以磁带为存储介质，磁带机是读取磁带信息的设备，磁带存储系统通常是用来做数据备份，磁带存储利用压缩技术容量不断扩大。“2015 年由 IBM、惠普和甲骨文等公司组建的信息存储行业协会 (INSIC) 公布了一个「国际磁带存储路线图」，预测 2025 年磁带储存密度能达到每平方英寸 91 GB，到 2028 年将突破每平方英寸 200 GB。2015 年 IBM 与 Fujifilm 研发出容量为 220 TB 的单盒磁带，成本只有硬盘的十五分之一。⁴”

图6.4是在目前⁵在售的磁带机和磁带。

6.5 存储的逻辑形式

存储设备使用不同的物理存储方式，但是这些物理的存储单元需要以一种逻辑的方式组织起来存储数据，设备的驱动程序屏蔽了设备的操作细节，而存储的逻辑形式进一步对

⁴数 据 来 源 于 <https://baijiahao.baidu.com/s?id=1626791570893436394&wfr=spider&for=pc>

⁵截图时间为 2021 年 9 月 17 日。



Figure 6.4: 磁带机和磁带

细节进行屏蔽，是的数据操作更加方便。

6.5.1 文件系统 (File System)

在计算机中，文件系统 (file system，通常简写为 fs) 控制着数据如何存储和获取，没有文件系统，数据将会作为一个整体存放在存储介质上，我们无法知道一部分数据在哪里结束，下一个数据从哪里开始。通过将数据分块，给每一块一个名字，数据很容易分离和标识，借用我们通常办公中数据管理时的名词“文件”，我们将计算机里的这么一组数据也称为文件，用来存储这么一组组数据和名字的逻辑规则和结构我们称为“文件系统”。⁶

这里有许多不同的文件系统，每一个都有不同的结构和逻辑，不同的存取速度，灵活性和安全性，可存储文件大小等等，一些文件系统是为特殊应用设计，比如 ISO 9660 文件系统是为光盘存储系统而设计。

文件系统可以被用于各种不同存储媒介的存储设备上，

⁶此段解释翻译自<https://encyclopedia.thefreedictionary.com/file+system>

比如机械硬盘、SSD 硬盘、磁带、内存 (RAM⁷) 等等。

目前常见的文件系统有：

- FAT(File Allocation Table) 系列，这一系列有 FAT12、FAT16、FAT32，其中 FAT32 采用 32bit 文件分配表，FAT32 文件大小的限制是 4GB。
- NTFS(New Technology File System) 是微软设计的文件操作系统，首先使用于 Windows NT 3.1 操作系统。
- exFAT (Extensible File Allocation Table)
- CDFS(Compact Disc File System)，也称为 ISO 9660
- Ext(extended file system)，于 1992 设计并用于 Linux 内核，这一系列还有 Ext2(the second extended file system)、Ext3(the third extended file system)、Ext4(the fourth extended file system)。Ext 是 GNU/Linux 系统中标准的文件系统，Ext3 是一种日志式文件系统，是对 Ext2 系统的扩展，它兼容 Ext2，Ext4 是 Ext3 的改进版。
- HFS(Hierarchical File System),HFS 是苹果公司设计的文件系统。
- HPFS (High Performance File System), HPFS 是 OS/2 操作系统首先使用的文件系统。
- APFS(APple File System) , APFS 是苹果公司 2017 年

⁷为了提高程序运行速度，目前也有把 RAM 作为一个临时文件系统来存储数据的方法。

开始使用的文件系统。

- VMFS(Virtual Machine File System), 是 VMWare 设计的一种高性能的群集文件系统，它使虚拟化技术的应用超出了单个系统的限制。VMFS 的设计、构建和优化针对虚拟服务器环境，可让多个虚拟机共同访问一个整合的群集式存储池，从而显著提高了资源利用率。
- ZFS(Zettabyte File System), 也叫动态文件系统（Dynamic File System），是 OpenSolaris 开源计划的一部分，后被重新命名为 OpenZFS。ZFS 是一款 128bit 文件系统，总容量是现有 64bit 文件系统的 1.84×10^{19} 倍，其支持的单个存储卷容量达到 16EiB(2^{64} byte，即 $16 \times 1024 \times 1024$ TB)；一个 zpool 存储池可以拥有 2^{64} 个卷，总容量最大 256ZiB (2^{78} byte)；整个系统又可以拥有 2^{64} 个存储池。
- JFS(JOURNAL FILE SYSTEM), JFS 是一种字节级日志文件系统，借鉴了数据库保护系统的技术，以日志的形式记录文件的变化。JFS 通过记录文件结构而不是数据本身的变化来保证数据的完整性。这种方式可以确保在任何时刻都能维护数据的可访问性。该文件系统主要是为满足服务器（从单处理器系统到高级多处理器和群集系统）的高吞吐量和可靠性需求而设计、开发的。JFS 文件系统是为面向事务的高性能系统而开发的。在 IBM 的 AIX 系统上，JFS 已经过较长时期的测试，结果表明它是可靠、快速和容易使用的。



Chapter 7

可执行程序 / 文件

用汇编语言写了程序，然后也编译为机器码了，现在就有一个问题“编译后的机器指令是以二进制方式存储在文件中的，我们如何让他执行起来？”，我们知道 CPU 在上电的时候都有一个缺省启动地址，从这个缺省启动地址开始，最终会启动操作系统，我们通常的操作环境都是在操作系统中来操控计算机，所以上面那个问题，我们可以转换为“操作系统中，编译后的机器指令是以二进制方式存储在文件中的，我们如何让他执行起来？”，我们以 windows 为例，看看我们是如何执行一个程序的，比如我们要运行 windows 中的记事本，我们按“win+r”，然后在运行窗口中输入“cmd”，启动命令行终端，在命令行终端中，我们输入“notepad.exe”，操作系统会运行记事本。下面我们概念性地解释一下这个过程。

当你输入 notepad.exe 并回车后，命令行终端知道你是

要执行这个程序，也就是说你输入的文件是一个包含有机器指令的二进制文件，操作系统需要把你这个文件加载到内存中，加载的内存地址他知道，等他把程序都加载进去后，他执行程序跳转指令，跳转到程序的第一行就可以开始执行你要执行的程序了。但是这里有个问题需要解决，操作系统在加载程序的时候，他需要知道如何把文件中的内容加载到内存中，这里面有地址问题，有数据存储的问题，为了使得加载能够顺利进行，定义了可执行程序/文件的格式，这个格式其实就是告诉操作系统如何将文件中包含的代码和数据加载到内存里面，我们通常看到的可执行程序/文件的格式定义有 windows 使用的 PE(Portable Executable) 格式¹，Linux 中使用的 ELF(Executable and Linkable Format) 格式。

7.1 PE format

"PE Format" 网页地址<https://docs.microsoft.com/en-us/windows/win32/debug/pe-format>

"Inside Windows : An In-Depth Look into the Win32 Portable Executable File Format" 网页地址
<https://docs.microsoft.com/en-us/archive/msdn-magazine/2002/february/inside-windows-win32-portable-executable-file-format-in-detail>

PEview 工具

¹利用工具进行 PE 文件格式的分析，可以帮助深入了解 PE 文件格式，可以参考博客文章“PE 文件格式分析”，网址<https://www.cnblogs.com/lqerio/p/11117579.html>

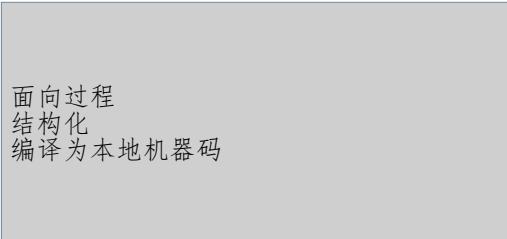
Chapter 8

高级语言

汇编语言与机器指令相比，程序员能够更好理解程序的语义，也更好记忆，但是离算法的基本语义单元相比，仍然粒度太小，这就使得利用汇编语言编写实际算法效率任然很低^⑦。所以计算机研究人员不断努力，设计出各种不同的计算机语言，希望能够使得程序开发效率更高。

7: 这句话希望读者能够好好理解一下其含义

8.1 C



面向过程
结构化
编译为本地机器码

C 语言诞生于美国的贝尔实验室，由 D.M.Ritchie(C 大的介绍见8.1)以 B 语言为基础发展而来，在它的主体设计完成后，Thompson 和 Ritchie 用它完全重写了 UNIX，且随着 UNIX 的发展，c 语言也得到了不断的完善。为了利于 C 语言的全面推广，许多专家学者和硬件厂商联合组成了 C 语言标准委员会，并在之后的 1989 年，诞生了第一个完备的 C 标准，简称“C89”，也就是“ANSI c”，截至 2020 年，最新的 C 语言标准为 2017 年发布的“C17”。

C 语言之所以命名为 C，是因为 C 语言源自 Ken Thompson 发明的 B 语言，而 B 语言则源自 BCPL 语言。

1967 年，剑桥大学的 Martin Richards 对 CPL 语言进行了简化，于是产生了 BCPL（Basic Combined Programming Language）语言。20 世纪 60 年代，美国 AT&T 公司贝尔实验室（AT&T Bell Laboratory）的研究员 Ken Thompson 闲来无事，手痒难耐，想玩一个他自己编的，模拟在太阳系航行的电子游戏——Space Travel。他背着老板，找到了台空闲的机器——PDP-7。但这台机器没有操作系统，而游戏必须使用操作系统的一些功能，于是他着手为 PDP-7 开发操作系统。后来，这个操作系统被命名为——UNIX。

1970 年，美国贝尔实验室的 Ken Thompson，以 BCPL 语言为基础，设计出很简单且很接近硬件的 B 语言（取 BCPL 的首字母）。并且他用 B 语言写了第一个 UNIX 操作系统。

1971 年，同样酷爱 Space Travel 的 Dennis M.Ritchie 为了能早点儿玩上游戏，加入了 Thompson 的开发项目，合作开发 UNIX。他的主要工作是改造 B 语言，使其更成熟。

1972 年，美国贝尔实验室的 D.M.Ritchie 在 B 语言的基础上最终设计出了一种新的语言，他取了 BCPL 的第二个字母作为这种语言的名字，这就是 C 语言。

1973 年初，C 语言的主体完成。Thompson 和 Ritchie 迫不及待地开始用它完全重写了 UNIX。此时，编程的乐趣使他们已经完全忘记了那个"Space Travel"，一门心思地投入到了 UNIX 和 C 语言的开发中。随着 UNIX 的发展，C 语言自身也在不断地完善。直到 2020 年，各种版本的 UNIX 内核和周边工具仍然使用 C 语言作为最主要的开发语言，其中还有不少继承 Thompson 和 Ritchie 之手的代码。¹

C 语言编译器有很多，比如鼻祖级编译器 gcc。

C 语言的 IDE 也很多，比如针对单片机的有 Keil，基于通用 PC 的有 Visual Studio，开源的也有很多，比如 Code::Blocks，也有些程序开发者使用一些通用的编辑器，比如 Notepad++，Sublime 编写源代码，然后利用编译器进行编译，用调试器(如 gdb) 进行调试。

初学者通过会把下载、安装相应的软件，然后编写一个最简单的“Hello world”程序，来检验其基本开发环境是否构建成功，这种思维方式是我们常用的一种工程思维。

¹以上关于 C 语言的介绍文字，来源于百度百科，网址<https://baike.baidu.com/item/c%E8%AF%AD%E8%A8%80>

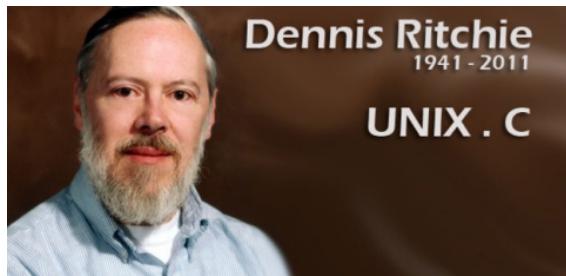


Figure 8.1: 丹尼斯·里奇 (Dennis MacAlistair Ritchie) 被世人尊称为“C 语言之父”“Unix 之父”，C 语言的诞生是现代程序语言革命的起点，今天，C 语言依旧在系统编程、嵌入式编程等领域占据着统治地位，而 Unix 在操作系统的发展历史上也是一个里程碑的系统，其不仅还用于大型机上，而且 Linux 和 MacOS 都是在其基础上发展而来。其获得美国国家技术奖章和图灵奖。

8.2 VB

8.3 C++

20 世纪 70 年代中期，Bjarne Stroustrup(C 大的图片介绍见8.2) 在剑桥大学计算机中心工作^⑤。他使用过 Simula 和 ALGOL，接触过 C。他对 Simula 的类体系感受颇深，对 ALGOL 的结构也很有研究，深知运行效率的意义。既要编程简单、正确可靠，又要运行高效、可移植，是 Bjarne Stroustrup 的初衷。以 C 为背景，以 Simula 思想为基础，正好符合他的设想。1979 年，Bjame Sgoustrup 到了 Bell 实验室，开始从事将 C 改良为带类的 C (C with classes) 的工作。1983 年该语言被正式命名为 C++。自从 C++ 被发明以来，它经历了 3 次主要的修订，每一次修订都为 C++ 增加了新的特征并作了一些修改。第一次修订是在 1985 年，第二次修订是在 1990 年，而第三次修订发生在 c++ 的标准化过程中。在 20 世纪

8: 有些介绍资料说他是在读博士期间接触到 Simula，如果是这样，那么他应该是在剑桥大学读的博士

90 年代早期，人们开始为 C++ 建立一个标准，并成立了一个 ANSI 和 ISO (International Standards Organization) 国际标准化组织的联合标准化委员会。该委员会在 1994 年 1 月 25 日提出了第一个标准化草案。在这个草案中，委员会在保持 Stroustrup 最初定义的所有特征的同时，还增加了一些新的特征。在完成 C++ 标准化的第一个草案后不久，发生了一件事情使得 C++ 标准被极大地扩展了：Alexander Stepanov 创建了标准模板库 (Standard Template Library, STL)。STL 不仅功能强大，同时非常优雅，然而，它也是非常庞大的。在通过了第一个草案之后，委员会投票并通过了将 STL 包含到 C++ 标准中的提议。STL 对 C++ 的扩展超出了 C++ 的最初定义范围。虽然在标准中增加 STL 是个很重要的决定，但也因此延缓了 C++ 标准化的进程。委员会于 1997 年 11 月 14 日通过了该标准的最终草案，1998 年，C++ 的 ANSI/ISO 标准被投入使用。通常，这个版本的 C++ 被认为是标准 C++。所有的主流 C++ 编译器都支持这个版本的 C++，包括微软的 Visual C++ 和 Borland 公司的 C++Builder。²

关于 C++ 版本迭代的较详细信息，可以参考 github 上 xiaodongQ 维护的一个文档“C 和 C++ 的历史版本迭代整理”³。

²以上关于 C++ 历史介绍文字，来源于百度文库，网址：<https://baike.baidu.com/item/C++>

³<http://xiaodongq.github.io/2019/10/18/CC++/>



Figure 8.2: 本贾尼·斯特劳斯特卢普 (Bjarne Stroustrup, 1950 年 6 月 11 日-), 丹麦人, 计算机科学家, 在德克萨斯 A&M 大学担任计算机科学的主席教授。他最著名的贡献就是开发了 C++ 程序设计语言。其个人网站是 <https://www.stroustrup.com/index.html>。

8.4 C#

C# 读作 C Sharp。最初它有个更酷的名字，叫做 COOL。微软从 1998 年 12 月开始了 COOL 项目，Delphi 语言的设计者 Hejlsberg 带领着 Microsoft 公司的开发团队，开始了第一个版本 C# 语言的设计，直到 2000 年 2 月，COOL 被正式更名为 C#，在 2000 年 9 月，国际信息和通信系统标准化组织为 C# 语言定义了一个 Microsoft 公司建议的标准，最终 C# 语言在 2001 年得以正式发布。

C# 一种由 C 和 C++ 衍生出来的面向对象的编程语言、运行于.NET Framework 和.NET Core(完全开源，跨平台)之上的高级程序设计语言，C# 源代码会编译为 MSIL(Microsoft Intermediate Language)，也称为通用中间语言 (CIL: Common Intermediate Language)，是一组与平台无关的指令，在程序执行时，通过 JIT(just-in-time) 编译器 (compiler) 将 MSIL 编译为本地码 (native code)，而运行时由安装在系统中的 CLR ((common language runtime) 提供运行所需环境⁴。

8.5 Java

20 世纪 90 年代，硬件领域出现了单片式计算机系统，这种价格低廉的系统一出现就立即引起了自动控制领域人员的注意，因为使用它可以大幅度提升消费类电子产品（如电视

⁴关于程序的运行过程可以参考微软官方网站上的介绍，网址为 <https://docs.microsoft.com/en-us/dotnet/standard/manage-d-execution-process>

机顶盒、面包烤箱、移动电话等)的智能化程度。Sun 公司为了抢占市场先机,在 1991 年成立了一个称为 Green 的项目小组,帕特里克·詹姆斯·高斯林(见图 8.3)、麦克·舍林丹和其他几个工程师一起组成的工作小组在加利福尼亚州门洛帕克市沙丘路的一个小工作室里面研究开发新技术,专攻计算机在家电产品上的嵌入式应用⁵。

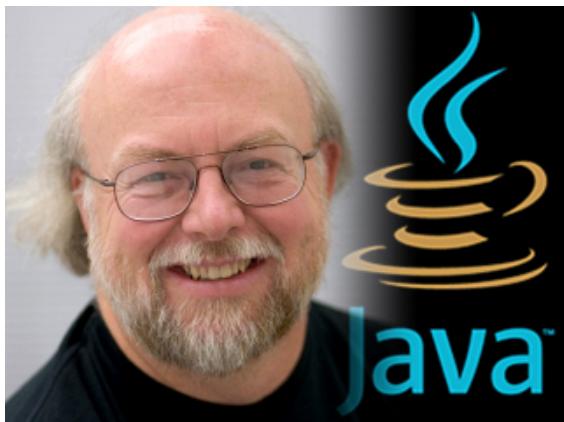


Figure 8.3: 詹姆斯·高斯林 (James Gosling), 1955 年 5 月 19 日出生于加拿大, Java 编程语言的共同创始人之一, 一般公认他为“Java 之父”。1977 年获得了加拿大卡尔加里大学计算机科学学士学位, 1983 年获得了美国卡内基梅隆大学计算机科学博士学位。

由于 C++ 所具有的优势, 该项目组的研究人员首先考虑采用 C++ 来编写程序。但对于硬件资源极其匮乏的单片式系统来说, C++ 程序过于复杂和庞大。另外由于消费电子产品所采用的嵌入式处理器芯片的种类繁杂, 如何让编写的程序跨平台运行也是个难题。为了解决困难, 他们首先着眼于语

⁵本节关于 Java 的发展介绍文字来自于 <https://baike.baidu.com/item/java/85979>

言的开发，假设了一种结构简单、符合嵌入式应用需要的硬件平台体系结构并为其制定了相应的规范，其中就定义了这种硬件平台的二进制机器码指令系统（即后来成为“字节码”的指令系统），以待语言开发成功后，能有半导体芯片生产商开发和生产这种硬件平台。对于新语言的设计，Sun 公司研发人员并没有开发一种全新的语言，而是根据嵌入式软件的要求，对 C++ 进行了改造，去除了留在 C++ 的一些不太实用及影响安全的成分，并结合嵌入式系统的实时性要求，开发了一种称为 Oak 的面向对象语言。

由于在开发 Oak 语言时，尚且不存在运行字节码的硬件平台，所以为了在开发时可以对这种语言进行实验研究，他们就在已有的硬件和软件平台基础上，按照自己所指定的规范，用软件建设了一个运行平台，整个系统除了比 C++ 更加简单之外，没有什么大的区别。1992 年的夏天，当 Oak 语言开发成功后，研究者们向硬件生产商进行演示了 Green 操作系统、Oak 的程序设计语言、类库和其硬件，以说服他们使用 Oak 语言生产硬件芯片，但是，硬件生产商并未对此产生极大的热情。因为他们认为，在所有人们对 Oak 语言还一无所知的情况下，就生产硬件产品的风险实在太大了，所以 Oak 语言也就因为缺乏硬件的支持而无法进入市场，从而被搁置了下来。

1994 年 6、7 月间，在经历了一场历时三天的讨论之后，团队决定再一次改变了努力的目标，这次他们决定将该技术应用于互联网。他们认为随着 Mosaic 浏览器的到来，因特网正在向高度互动的远景演变，而这一远景正是他们在有线电视网中看到的。作为原型，帕特里克·诺顿写了一个小型万维网浏览器 WebRunner。

1995 年，互联网的蓬勃发展给了 Oak 机会。业界为了使死板、单调的静态网页能够“灵活”起来，急需一种软件技术来开发一种程序，这种程序可以通过网络传播并且能够跨平台运行。于是，世界各大 IT 企业为此纷纷投入了大量的人力、物力和财力。这个时候，Sun 公司想起了那个被搁置起来很久的 Oak，并且重新审视了那个用软件编写的试验平台，由于它是按照嵌入式系统硬件平台体系结构进行编写的，所以非常小，特别适用于网络上的传输系统，而 Oak 也是一种精简的语言，程序非常小，适合在网络上传输。Sun 公司首先推出了可以嵌入网页并且可以随同网页在网络上传输的 Applet（Applet 是一种将小程序嵌入到网页中进行执行的技术），并将 Oak 更名为 Java（LOGO 见图 8.4）（在申请注册商标时，发现 Oak 已经被人使用了，再想了一系列名字之后，最终，使用了提议者在喝一杯 Java 咖啡时无意提到的 Java 词语）。1995 年 5 月 23 日，Sun 公司在 Sun world 会议上正式发布 Java 和 HotJava 浏览器。IBM、Apple、DEC、Adobe、HP、Oracle、Netscape 和微软等各大公司都纷纷停止了自己的相关开发项目，竞相购买了 Java 使用许可证，并为自己的产品开发了相应的 Java 平台。

1996 年 1 月，Sun 公司发布了 Java 的第一个开发工具包（JDK 1.0），这是 Java 发展历程中的重要里程碑，标志着 Java 成为一种独立的开发工具。9 月，约 8.3 万个网页应用了 Java 技术来制作。10 月，Sun 公司发布了 Java 平台的第一个即时（JIT）编译器。

1997 年 2 月，JDK 1.1 面世，在随后的 3 周时间里，达到了 22 万次的下载量。4 月 2 日，Java One 会议召开，参



Figure 8.4: Java 的 logo。

会者逾一万人，创当时全球同类会议规模之纪录。9月，Java Developer Connection 社区成员超过 10 万。

1998 年 12 月 8 日，第二代 Java 平台的企业版 J2EE(Java 2Enterprise Edition) 发布。1999 年 6 月，Sun 公司发布了第二代 Java 平台（简称为 Java2）的 3 个版本：J2ME（Java2 Micro Edition，Java2 平台的微型版），应用于移动、无线及有限资源的环境；J2SE（Java 2 Standard Edition，Java 2 平台的标准版），应用于桌面环境；J2EE（Java 2Enterprise Edition，Java 2 平台的企业版），应用于基于 Java 的应用服务器。Java 2 平台的发布，是 Java 发展过程中最重要的一个里程碑，标志着 Java 的应用开始普及。

1999 年 4 月 27 日，HotSpot 虚拟机发布。HotSpot 虚拟机发布时是作为 JDK 1.2 的附加程序提供的，后来它成为了 JDK 1.3 及之后所有版本的 Sun JDK 的默认虚拟机（也就是我们现在所说的 Java 运行环境 (JRE;java runtime environment))。

随后 Java 得到更广泛的业界支持。

2006 年 11 月 13 日，Java 技术的发明者 Sun 公司宣布，将 Java 技术作为免费软件对外发布。Sun 公司正式发布的有关 Java 平台标准版的第一批源代码，以及 Java 迷你版的可执行源代码。从 2007 年 3 月起，全世界所有的开发人员均可对 Java 源代码进行修改。2009 年，甲骨文 (Oracle) 公司宣布收购 Sun。2010 年，Java 编程语言的共同创始人之一詹姆斯·高斯林从 Oracle 公司辞职。2011 年，甲骨文公司举行了全球性的活动，以庆祝 Java7 的推出，随后 Java7 正式发布。2014 年，甲骨文公司发布了 Java8 正式版。

8.6 Python

Python 由荷兰数学和计算机科学研究学会的吉多·范罗苏姆 (Guido van Rossum) 于 1990 年代初设计，作为一门叫做 ABC 语言的替代品，Python 由于是一种解释型语言，所以很容易实现跨平台，最初被设计用于编写自动化脚本 (shell)，随着版本的不断更新和语言新功能的添加，逐渐被用于独立的、大型项目的开发。

1989 年圣诞节期间，在阿姆斯特丹，Guido 为了打发圣诞节的无趣，决心开发一个新的脚本解释程序，作为 ABC 语言的一种继承。之所以选中 Python (大蟒蛇的意思) 作为该编程语言的名字，是取自英国 20 世纪 70 年代首播的电视喜剧《蒙提·派森的飞行马戏团》(Monty Python's Flying Circus)。

ABC 是由 Guido 参加设计的一种教学语言^⑨。就 Guido 本人看来，ABC 这种语言非常优美和强大，是专门为非专业程序员设计的。但是 ABC 语言并没有成功，究其原因，Guido

⑨：在国外很多教编译原理的课程都会有一个渐进练习，就是随着课程的推进，会让大家实现编译器的不同功能，最后课程结束，大家就编写了一个语言的编译器。



Figure 8.5: 吉多·范罗苏姆 (Guido van Rossum)，荷兰计算机程序员，他作为 Python 程序设计语言的作者而为人们熟知。在 Python 社区，吉多·范罗苏姆被人们认为是“仁慈的独裁者 (BDFL: Benevolent Dictator For Life)”，意思是他还关注 Python 的开发进程，并在必要的时刻做出决定。他在 Google 工作，在那里他把一半的时间用来维护 Python 的开发。2020 年 11 月 12 日，64 岁的 Python 之父 Guido van Rossum 在自己社交网站上宣布：由于退休生活太无聊，自己决定加入 Microsoft 的 DevDiv Team。

认为是其非开放造成的。Guido 决心在 Python 中避免这一错误。同时，他还想实现在 ABC 中脑海中闪现过的想法，但未曾实现的东西。

就这样，Python 在 Guido 手中诞生了。可以说，Python 是从 ABC 发展起来，主要受到了 Modula-3（另一种相当优美且强大的语言，为小型团体所设计的）的影响。并且结合了 Unix shell 和 C 的习惯。

Python(Python 的 Logo 如图8.6) 目前（指 2021 年）已经成为最受欢迎的程序设计语言之一。自从 2004 年以后，python 的使用率呈线性增长。Python 2 于 2000 年 10 月 16 日发布，稳定版本是 Python 2.7，Python 3 于 2008 年 12 月 3 日发布，不完全兼容 Python 2.0，2011 年 1 月，它被 TIOBE 编程语言排行榜评为 2010 年度语言。

Python 在执行时，首先会将.py 文件中的源代码编译成 Python 的 byte code（字节码），然后再由 Python Virtual Machine（Python 虚拟机）来执行这些编译好的 byte code。这种机制的基本思想跟 Java，.NET 是一致的。然而，Python Virtual Machine 与 Java 或.NET 的 Virtual Machine 不同的是，Python 的 Virtual Machine 是一种更高级的 Virtual Machine。这里的高级并不是通常意义上的高级，不是说 Python 的 Virtual Machine 比 Java 或.NET 的功能更强大，而是说和 Java 或.NET 相比，Python 的 Virtual Machine 距离真实机器的距离更远。

目前 python 有着丰富的库支持，得到广泛应用，如：Web 和 Internet 开发、科学计算和统计、人工智能、桌面界面开

发、网络爬虫、数据统计、脚本编写等等。⁶



Figure 8.6: python 的 Logo。

⁶以上 python 的介绍文字来自于<https://baike.baidu.com/item/Python/407313>



Chapter 9

操作系统

操作系统（operation system，缩写为 OS），我们给出几种定义，通过这些定义，大家可以理解计算机操作系统概念。

操作系统（operation system，简称 OS）是管理计算机硬件与软件资源的计算机程序。操作系统需要处理如管理与配置内存、决定系统资源供需的优先次序、控制输入设备与输出设备、操作网络与管理文件系统等基本事务。操作系统也提供一个让用户与系统交互的操作界面。¹

其实操作系统就是用来操作计算机（软硬件资源）的一个系统。

operating system :The low-level software that supports a computer's basic functions, such as scheduling tasks and

¹<https://baike.baidu.com/item/192>

controlling peripherals.²

An operating system is the primary software that manages all the hardware and other software on a computer. The operating system, also known as an “OS,” interfaces with the computer’s hardware and provides services that applications can use.³

An Operating System (OS) is a software that acts as an interface between computer hardware components and the user. Every computer system must have at least one operating system to run other programs. Applications like Browsers, MS Office, Notepad Games, etc., need some environment to run and perform its tasks. The OS helps you to communicate with the computer without knowing how to speak the computer’s language. It is not possible for the user to use any computer or mobile device without having an operating system.<https://www.guru99.com/operating-system-tutorial.html>

9.1 API

在微软的官方网站上，我们可以看到 Windows 提供的 API 信息，有时候厂家还会有一些接口不公开。在<https://docs.microsoft.com/zh-cn/windows/win32/api>

²https://www.lexico.com/definition/operating_system

³<https://www.howtogeek.com/361572/what-is-an-operating-system/>

iindex/windows-api-list中给出的 win32 API 有：

1. 用户界面: Windows UI API 创建并使用 windows 来显示输出、提示用户输入，并执行其他支持与用户交互的任务。大多数应用程序至少创建一个窗口。
2. Windows 环境 (Shell)
3. 用户输入和消息传递
4. 数据访问和存储
5. 诊断
6. 图形和多媒体: 利用图形、多媒体、音频和视频 api，应用程序可以合并格式化文本、图形、音频和视频。
7. 设备
8. 系统服务: 系统服务 api 使应用程序能够访问计算机的资源和基础操作系统的功能，如内存、文件系统、设备、进程和线程。
9. 安全和标识: 安全和标识 api 在登录时启用密码身份验证，可以对所有可共享的系统对象、特权访问控制、权限管理和安全审核进行任意保护。
10. 应用程序安装和维护
11. 系统管理员和管理
12. 网络和 internet

9.2 系统调用 (System Calls)

先我们引用一些技术网站上关于系统调用的解释和说明，大家通过阅读这些材料来了解系统调用的概念。

~~~~~

The operating system provides two main functions, acting as an interface between the user and the hardware. And to allocate resources appropriately. For the most part, allocating resources happens in the background and is not visible to the user. But the former part of acting as an interface happens on the GUI. For example, this includes reading, writing, creating, and deleting files on any directory. The user interface of an operating system is primarily for dealing with abstractions. All **this happens by various system calls.**

System calls differ from one OS to another, but the underlying concept remains the same.

**Definition:** In computing, there is an interface between the operating system and the user program, and it is defined by a set of extended instructions that the operating system provides, and that set of instructions is called system calls.

In other words, a system call is a way for programs to interact with the operating system. It is a way in which a computer program requests a service from the kernel of the

operating system.

~~~~~<sup>4</sup>

操作系统的主要功能是为管理硬件资源和为应用程序开发人员提供良好的环境来使应用程序具有更好的兼容性，为了达到这个目的，内核提供一系列具备预定功能的多内核函数，通过一组称为系统调用 (system call) 的接口呈现给用户。系统调用把应用程序的请求传给内核，调用相应的内核函数完成所需的处理，将处理结果返回给应用程序。

现代的操作系统通常都具有多任务处理的功能，通常靠进程来实现。由于操作系统快速的在每个进程间切换执行，所以一切看起来就会像是同时的。同时这也带来了很多安全问题，例如，一个进程可以轻易的修改进程的内存空间中的数据来使另一个进程异常或达到一些目的，因此操作系统必须保证每一个进程都能安全的执行。这一问题的解决方法是在处理器中加入基址寄存器和界限寄存器。这两个寄存器中的内容用硬件限制了对储存器的存取指令所访问的储存器的地址。这样就可以在系统切换进程时写入这两个寄存器的内容到该进程被分配的地址范围，从而避免恶意软件。

为了防止用户程序修改基址寄存器和界限寄存器中的内容来达到访问其他内存空间的目的，这两个寄存器必须通过

⁴ 资料来源于<https://technobyt.org/system-calls-in-operating-systems-simple-explanation/#:~:text=Definition%3A%20In%20computing%2C%20there%20is%20an%20interface%20between,for%20programs%20to%20interact%20with%20the%20operating%20system.>

一些特殊的指令来访问。通常，处理器设有两种模式：“用户模式”与“内核模式”，通过一个标签位来鉴别当前正处于什么模式。一些诸如修改基址寄存器内容的指令只有在内核模式中可以执行，而处于用户模式的时候硬件会直接跳过这个指令并继续执行下一个。

同样，为了安全问题，一些 I/O 操作的指令都被限制在只有内核模式可以执行，因此操作系统有必要提供接口来为应用程序提供诸如读取磁盘某位置的数据的接口，这些接口就被称为系统调用。

当操作系统接收到系统调用请求后，会让处理器进入内核模式，从而执行诸如 I/O 操作，修改基址寄存器内容等指令，而当处理完系统调用内容后，操作系统会让处理器返回用户模式，来执行用户代码。

*****⁵

9.3 Windows 运行时 (Windows Runtime)

对于 Windows Runtime APIs 的理解，我们引用 Windows Runtime 的介绍资料，让大家理解 Windows Runtime。

~~~~~

Windows Runtime, or WinRT?

---

<sup>5</sup> 材料来源<https://baike.baidu.com/item/>

They are interchangeable. Don't mistake it for Windows RT - it has nothing to do with Windows Runtime/WinRT, and it's a horrible name. I'm going to use WinRT for the rest of the article, for the sake of brevity.

### WinRT is not a runtime

A runtime is a collection of helper libraries that provide support for a platform or a language. WinRT, on the other hand, fundamentally, is an ABI (Application Binary Interface). It provides a set of binary-level (read: in terms of machine instructions, memory layout, etc, as opposed to source code level) common protocols that describes how components can talk to each other, including:

1. How to make calls
2. How to pass data structure
3. What is the set of primitive types
4. What is a structure/object

In Windows 8+, where WinRT is supported, the entire WinRT ecosystem consists of the following:

1. The OS (Windows, XBOX, phone) provides a hierarchy of object-oriented APIs that expose OS functionality, following WinRT ABI, typically implemented in C++ (not C++/CX). These APIs are often called WinRT APIs because they are available in Windows and is usually the only WinRT

types you care about. But WinRT API is not WinRT itself - it is simply a collection of types/methods that follows WinRT ABI. You could implement your own WinRT APIs, for example.

2. Visual Studio provides a set of compilers/languages, such as C#, VB.NET, JS, C++/CX, C++, that understands the ABI so that they can interact with the new set of APIs that is based on WinRT.

3. Applications, written using one or more of those languages, and running as UWP (Universal Windows Apps) or Desktop Bridge (sandboxed desktop apps).

4. 3rd party libraries and controls.

How do you call a WinRT method?

For C++ code, calling these functions are very straightforward - just make a v-table call to the underlying method using a predetermined offset (if it is the 3rd method, then it is offset 2 \* pointer\_size). It might be easier to explain in terms of C, if you are not familiar with COM:

```
1      struct IFoo_Vtbl
2      {
3          void *pFoo;
4      }
5
6      IFoo *p = GetFoo();
7      IFoo_Vtbl **ppVtbl = p;
8      IFoo_Vtbl *pVtbl = *ppVtbl; // first pointer
9          -value pointed by p is the v-table pointer
((foo_func_ptr)(pVtbl->pFoo))(arg1, arg2, ...);
```

In C#, there are a lot more involved. .NET / CLR is responsible for generating stubs that calls from managed code into native code, flipping a few internal states, converting managed data structure to native data structures (such as strings, etc), convert native WinRT objects into RCWs (Runtime Callable Wrapper), and managed objects into CCWs back to native. If you are familiar with .NET COM interop, you'll immediately recognize that RCWs/CCWs are used in COM interop. Here in WinRT they serve exactly the same purpose - as proxies that can be consumed by the other side. This is another evidence that WinRT is built on top of COM.

~~~~~<sup>6</sup>

⁶ 材料来源于<https://yizhang82.dev/what-is-winrt>



Appendices



Appendix A

IC 产业链

集成电路作为半导体产业的核心，市场份额达 83%，由于其技术复杂性，产业结构高度专业化。随着产业规模的迅速扩张，产业竞争加剧，分工模式进一步细化。目前市场产业链为 IC 设计、IC 制造和 IC 封装测试。

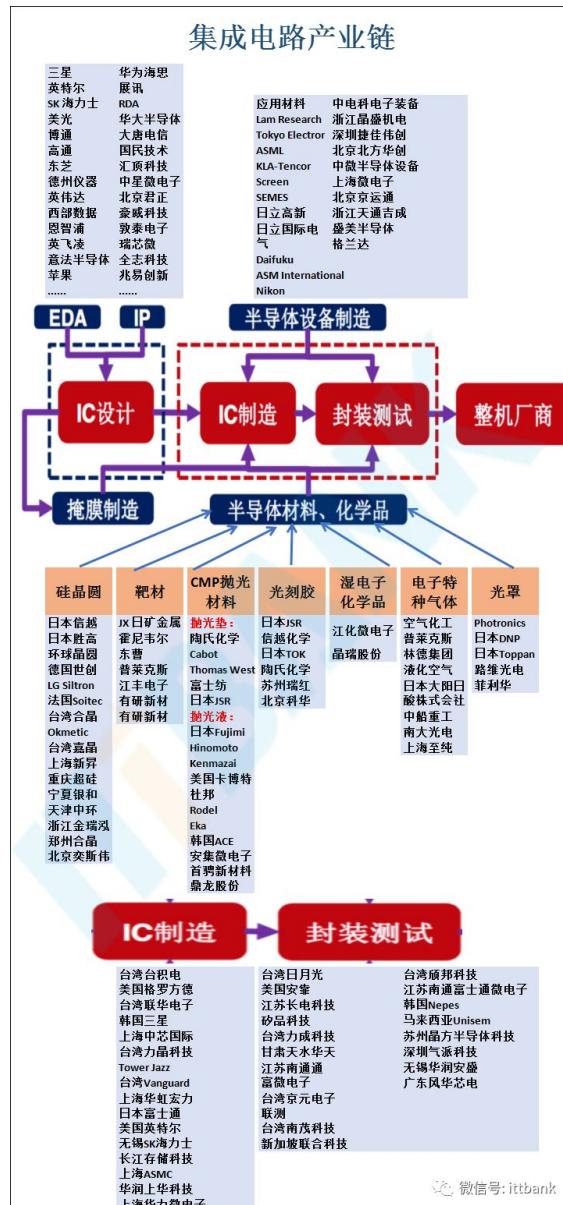


Figure A.1: IC 产业链

Appendix B

网络安全产业链

从网络安全行业产业链来看，上游主要为工控机、服务器、存储器、芯片及操作系统、数据库等软硬件厂商。产业链上游市场竞争充分，主要参与者均为成熟的全球化厂商，产品更新快，产量充足，产品价格相对稳定，且产品性价比呈上升趋势。中游为提供安全产品、安全服务、安全集成的厂商，并形成安全网关、数据安全、身份管理、云安全、物联网安全、应用安全和安全管理等诸多细分领域。下游则是政府、金融、电信、能源等各行业的企业级用户。国内网络安全行业产业链如B.1所示。¹

¹ 来源https://www.sohu.com/a/372840041_120217338



Figure B.1: 网络信息安全产品及服务产业链

Bibliography