电子系统综合设计

Integrated Electronic System Design

第2章 数字计算电路

主讲：王卫东 陈晓辉

2022年9月27日

引言：关于计算

- 什么是计算？小至四则运算，大至并行计算、云服务系统。
- 服务于计算的电路单元同样规模也存在巨大的差异，如1971年的Intel 4004只提供加法、减法和移位指令，集成了约2300个晶体管；而用于智能手机的单颗芯片可以完成视频解码、图像处理、通信等多种复杂算法，所集成的晶体管数据可达100亿。

1971年，第一款商用处理器，用于Busicom 141-PF计算器

2019年9月，华为发布的麒麟990 5G是一款5G全集成SoC处理器，在约113平方毫米的芯片上集成了多达103亿个晶体管，是世界上第一个晶体管数量过百亿的移动SoC。

引言：人工智能的三个要素

人工智能技术的发展需要三个要素：数据(Data)、算法(Algorithm)和算力(Computing power)。大数据本身并不必然意味着大价值。数据是资源，要得到资源的价值，就必须进行有效的数据分析。今天的人工智能热潮主要就是由于机器学习，特别是其中的深度学习技术取得巨大进展，而且是在大数据、大算力的支持下发挥出巨大的威力。人工智能算法模型对于算力的巨大需求，也推动了芯片业的发展。可以看到，人工智能算法模型的发展，与算力、芯片发展之间，有相互促进的作用。这几方面的要素是互相促进、互相支撑。

{wdwang, cxh}@ustc.edu.cn

来源:2020年8月7日，全球人工智能和机器人峰会（CCF-GAIR 2020）周志华现场演讲

引言：数字计算电路的三个要素

◆数字计算电路单元是电子系统中重要的功能性部件，可用算力、算法、存储三个要素来刻画其离散信号（数据）的运算能力。

◆通俗来说，**算力**指的是运算的能力，如每秒钟可完成多少次浮点数乘法；**算法**可以是固化的ASIC或FPGA电路，也可表现为处理器上运行的代码集合；**存储**则对应算法所需要的输入输出数据，高吞吐量的算法就需要大容量快速的存储电路单元。三个要素对计算性能的影响是交错在一起的，不能割裂开来。

{wdwang, cxh}@ustc.edu.cn

第2章 数字计算电路系统

- 2.1 概述
  - 计算电路的不同层次
  - 基础计算电路
  - 面向应用的计算电路
  - 计算系统
  - “内存墙 (Memory Wall) ”问题
- 2.2 基础计算电路
- 2.3 面向应用的计算电路
- 2.4 异构计算系统
- 2.5 高速计算电路的存储管理

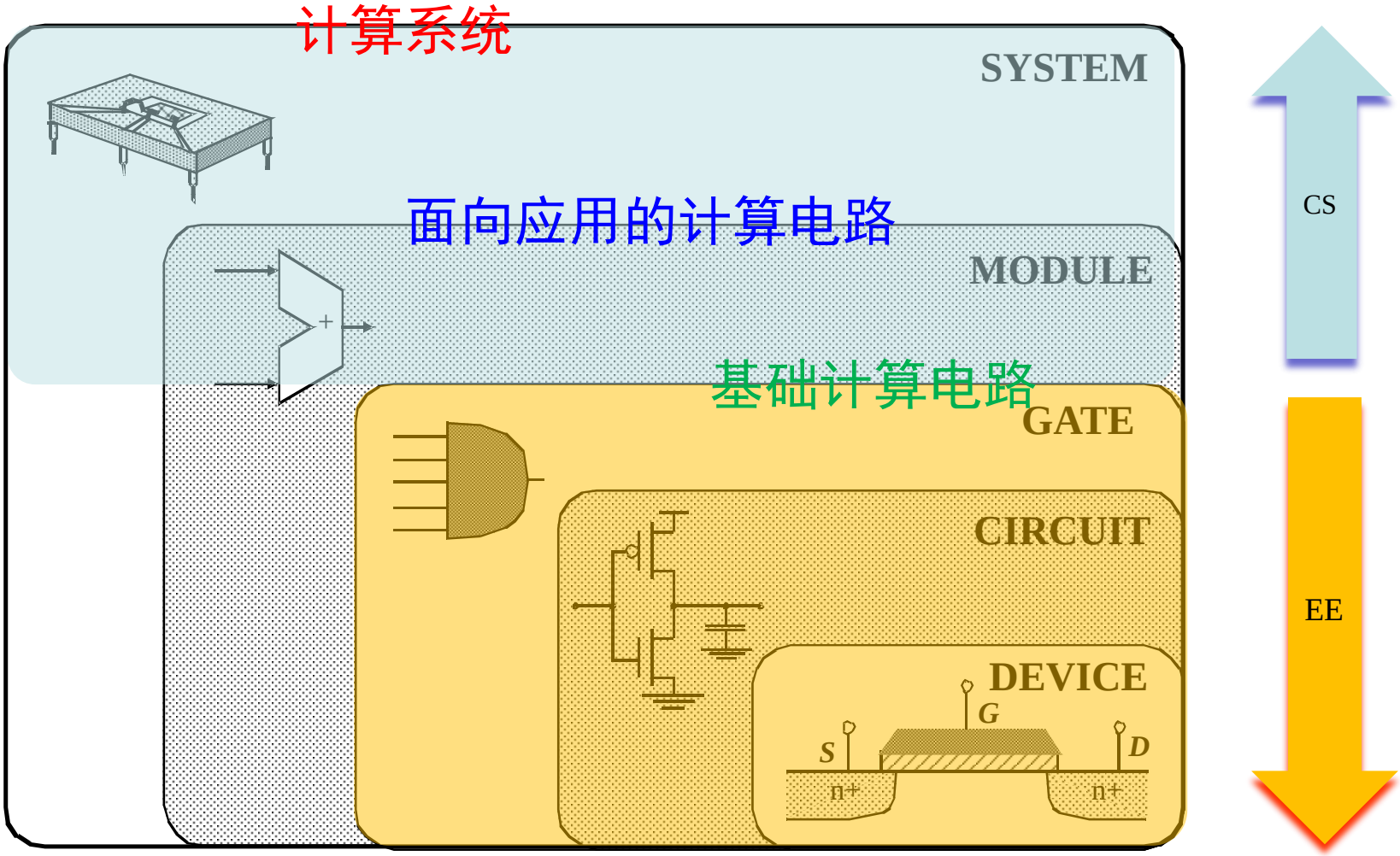
{wdwang, cxh}@ustc.edu.cn

计算电路的不同层次

- 基础计算电路
  - 最底层是诸如整数乘法器、正弦函数计算等基本电路。
- 面向应用的计算电路
  - 其次是组合了多种基础计算电路单元后形成的，可解决特定应用问题的电路模块，如AES (Advanced Encryption Standard)加密算法电路、视频处理加速电路、神经网络加速电路等。
- 计算系统
  - 最高层次为计算系统，如采用英特尔CPU构建的PC、服务于智能平板的SoC (System on Chip)处理器芯片等。
- 传统并行计算系统→云计算→…
  - 已超出了单个电子系统的范畴

{wdwang, cxh}@ustc.edu.cn

系统抽象层次



Picture Source： Digital Integrated Circuits, A Design Perspective，2002
Jan M. Rabaey, Anantha Chandrakasan, Borivoje Nikolic

{wdwang, cxh}@ustc.edu.cn

基础计算电路

◆基础计算电路的目标是把数学运算表达为特定结构的数字逻辑电路单元。设计此类电路单元时往往需要在计算时延和所使用到的逻辑门数目之间进行平衡。

- 例如，二进制的移位运算，如果简单用移位寄存器来实现，移1比特需要一个时钟周期，移n比特计算时延就是n个时钟周期；若需要单个时钟周期内实现任意移位，就需要更多逻辑门来搭建桶形移位寄存器。
- 再如，二进制整数乘法如果用串行移位寄存器实现，位宽n比特的乘数则需要至少n个时钟周期，而DSP(Digital Signal Processor)中往往要实现快速乘法器，无论操作数位宽多少，乘法的计算在一个时钟周期内完成。

{wdwang, cxh}@ustc.edu.cn

面向应用的计算电路

◆面向应用的计算电路，往往会针对特定应用组合多种基础计算单元、添加特殊计算单元、甚至将算法直接体现到电路结构设计中。

- 图形图像处理时，往往待处理的数据类型单一(像素值或者坐标值)，**浮点数运算**占到其处理数据的绝大多数，需要高效率的**超越函数**运算。针对这类应用就需要提高计算电路的**并行度**，增加浮点数加/减法、乘法电路及除法电路，另外常需要存储大量超越函数近似算法的查表信息。
- 与之不同的通信应用中，还有信息安全领域，相关的算法往往对应为数学中的**多项式运算**，并且多数情形需要**迭代**，这些算法的并行实现难度较达，不少算法以DSP的算法或FPGA的IP核方式出现，或是以专用处理器（网络处理器、媒体处理器、加密处理器等）形式出现。

{wdwang, cxh}@ustc.edu.cn

计算系统

- 计算系统是在顶层来看待电路系统。往往一个计算系统内既有多种基础计算电路单元，又有多种面向应用的计算电路单元。
  - 例如，微软把英特尔的CPU和FPGA组合在一起以完成其搜索引擎的算法加速。
  - 各种智能手机的SoC (System on Chip)则把CPU、DSP、GPU组合在单颗芯片内以满足手机在拍照、通信等不同应用场景下的计算需要。
- 集成在同一个计算系统内的不同计算单元设计目标不同，其互联方式和效率往往是系统设计时重点考虑的问题之一。

{wdwang, cxh}@ustc.edu.cn


“内存墙 (Memory Wall) ”问题

- 待处理数据不能及时从存储器传递给计算电路单元。
  - 在过去的几十年中，处理器的性能以每年大约55%的速度快速提升，而内存性能的提升速度则只有每年10%左右。不均衡的发展速度使得目前内存的存取速度严重滞后于处理器的计算速度，这种内存瓶颈导致高性能处理器难以发挥出应有的计算能力。
  - 事实上，科学家们在1994年就分析和预测了这一问题，并将这种严重阻碍处理器性能发挥的内存瓶颈命名为“内存墙(Memory Wall)”。
- 单一考虑“算力”已经不能满足要求了，能否在存储电路单元与计算电路单元之间快速地交换数据影响到了**算力可以发挥的程度**。往往采用高速片内总线和缓存来缓解“内存墙”问题。

{wdwang, cxh}@ustc.edu.cn

小结：数字计算电路

- 计算电路的不同层次
  - 基础计算电路
  - 面向应用的计算电路
  - 计算系统
  - 传统并行计算系统→云计算→…
- 内存墙 (Memory Wall)
  - 算力、算法、存储三个要素，互相交错与支撑
  - 算力对应到运算的能力（整数、浮点小数、各种函数…）
  - 算法对应到固化电路或处理器上运行的代码集合
  - 存储则对应算法所需要的输入输出数据

{wdwang, cxh}@ustc.edu.cn

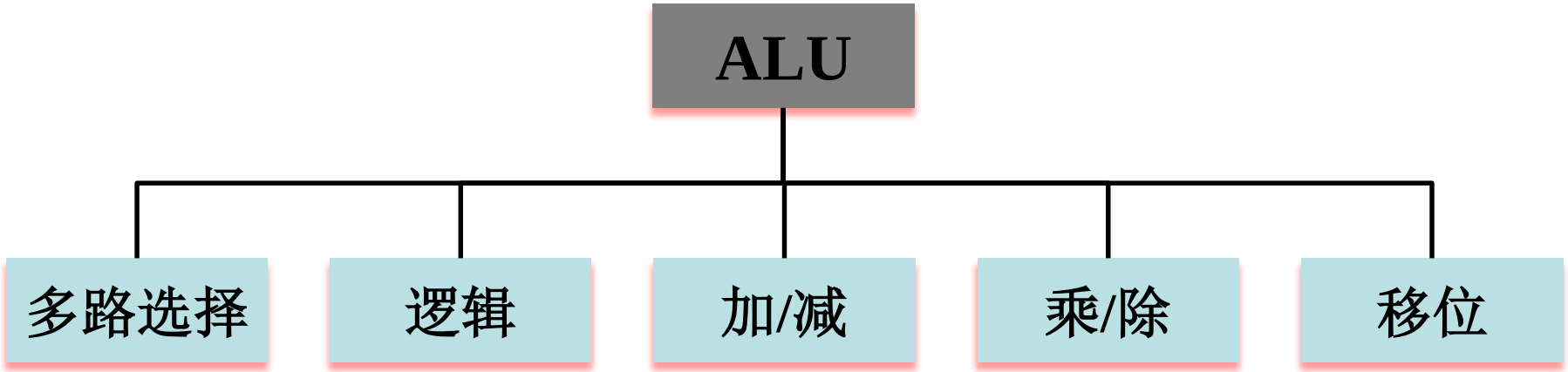
第2章 数字计算电路

- 2.1 概述
- 2.2 基础计算电路
  - 2.2.1 加法电路
  - 2.2.2 从整数到定点小数
  - 2.2.3 乘法电路
  - 2.2.4 除法电路
  - 2.2.5 浮点数运算电路
  - 2.2.6 超越函数的计算
- 2.3 面向应用的计算电路
- 2.4 异构计算系统
- 2.5 高速计算电路的存储管理

{wdwang, cxh}@ustc.edu.cn

通用ALU的基本功能模块

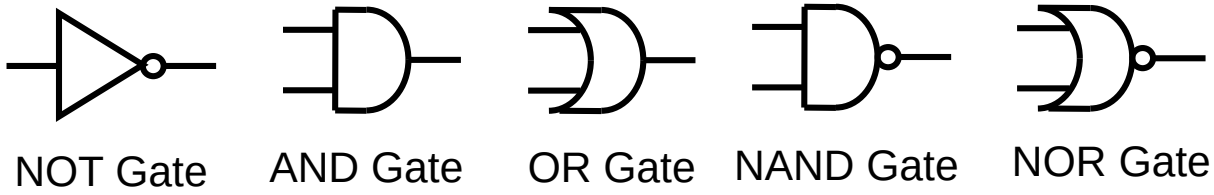
- ALU (Arithmetic & Logical Unit)组成：加/减法器、乘/除法器、逻辑运算器、移位运算器、多路选择器
  - 有些CPU加/减法器、乘/除法器单独实现。加法与减法原理相同。乘法操作是以加法操作为基础的，由乘数的一位或几位译码控制逐次产生部分积，部分积相加得乘积。除法则又常以乘法为基础，即选定若干因子乘以除数，使它近似为1，这些因子乘被除数则得商。

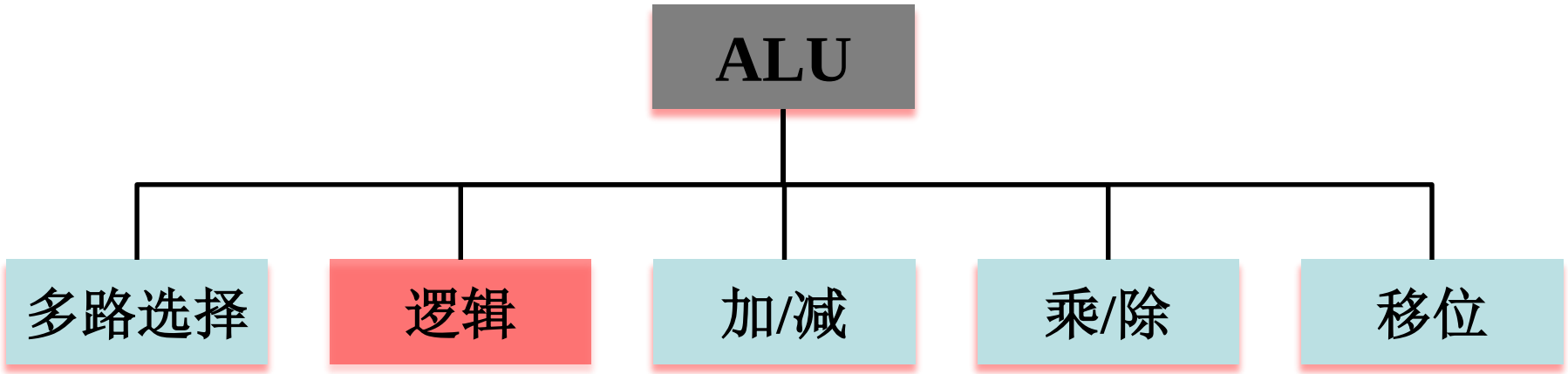


{wdwang, cxh}@ustc.edu.cn

通用ALU的基本功能模块

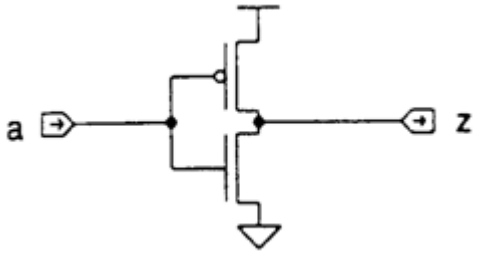
- 逻辑运算器

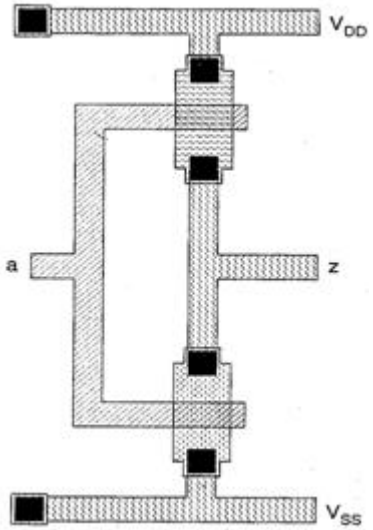




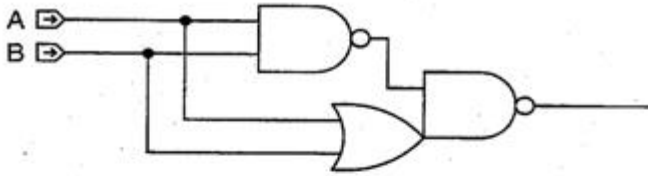
{wdwang, cxh}@ustc.edu.cn

基本逻辑元算单元

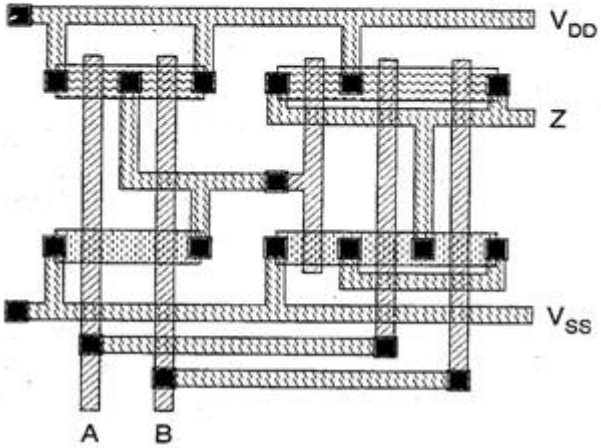


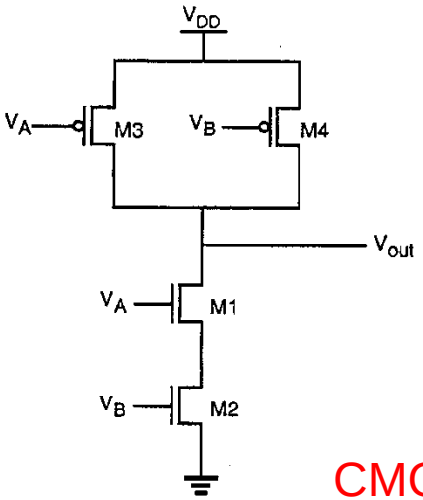


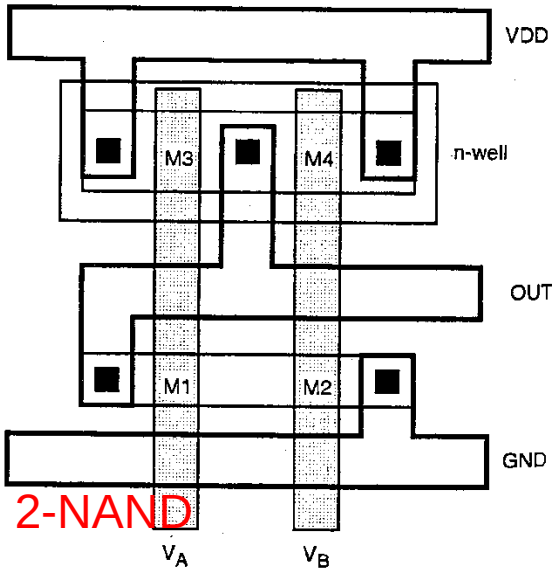
CMOS NOT



CMOS XNOR





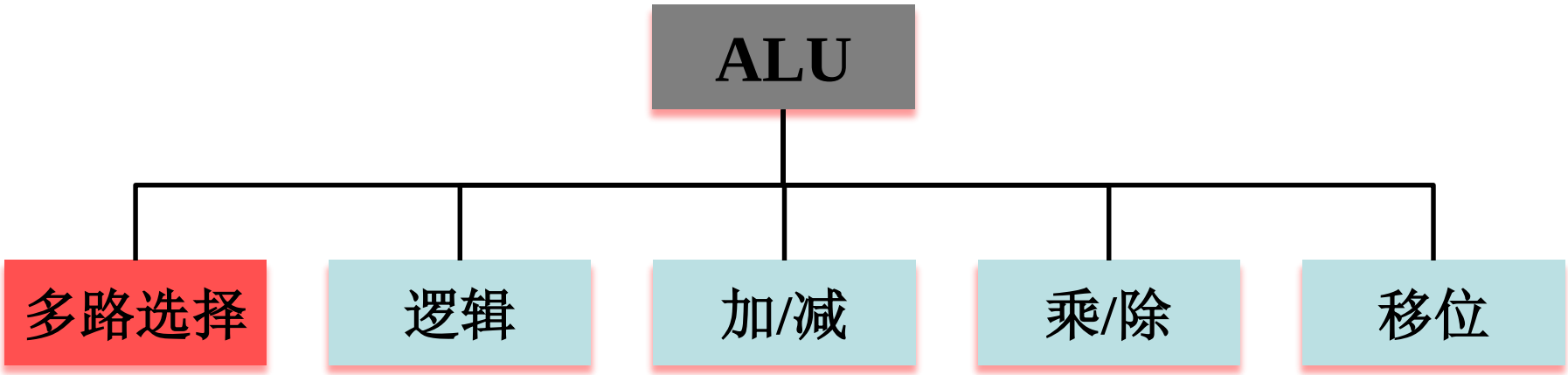


CMOS 2-NAND

{wdwang, cxh}@ustc.edu.cn

通用ALU的基本功能模块

◆多路选择器是算术、逻辑运算模块的核心。它根据来自控制器的指令，将来自控制模块和程序存储器的数据，分配给各种运算模块进行运算，同时要将运算结果和各种标志位输出。

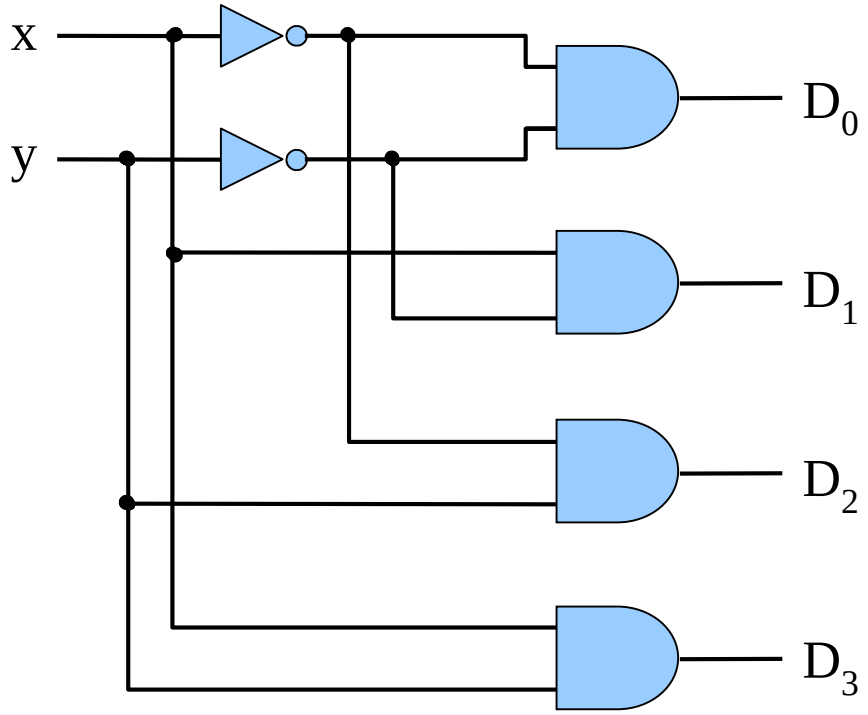


{wdwang, cxh}@ustc.edu.cn

ALU的多路选择：指令译码
Decoder: 2-to-4

- 将运算类型翻译为各运算单元的Enable信号
  - 机器指令：操作码+操作数…

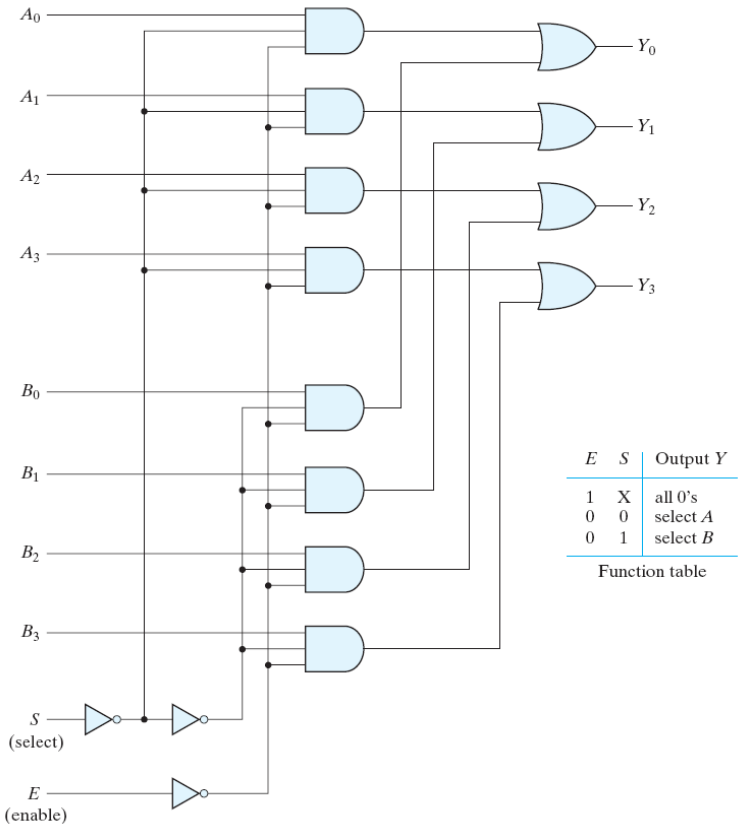
x	y	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1



{wdwang, cxh}@ustc.edu.cn

多路选择：总线切换示意
Multiplexer: 2-to-1 with Enable

- 将操作数送到合适的运算单元



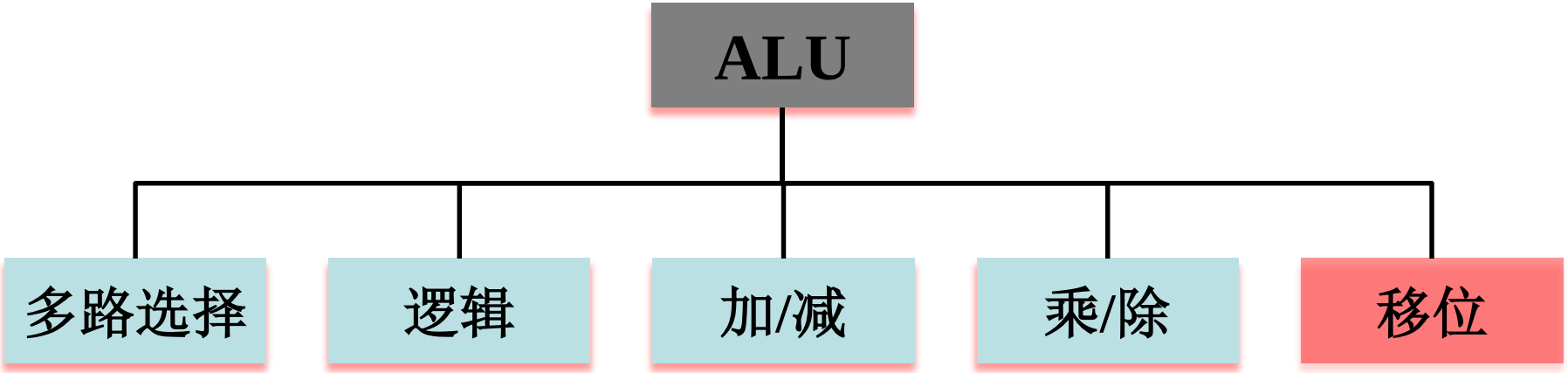
E	S	Output Y
1	X	all 0's
0	0	select A
0	1	select B

Function table

{wdwang, cxh}@ustc.edu.cn

通用ALU的基本功能模块

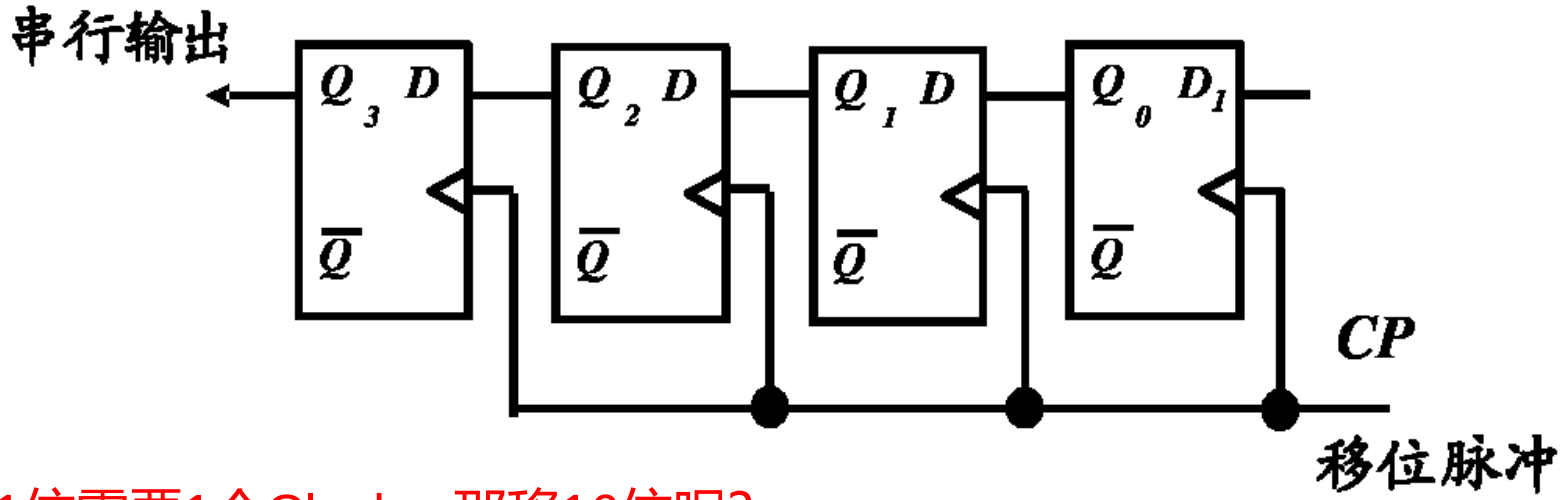
- ALU组成：加/减法器、乘/除法器、逻辑运算器、移位运算器、多路选择器
  - 有些CPU加/减法器单独实现
  - 有些CPU乘/除法器单独实现



{wdwang, cxh}@ustc.edu.cn

数字电路中的移位寄存器

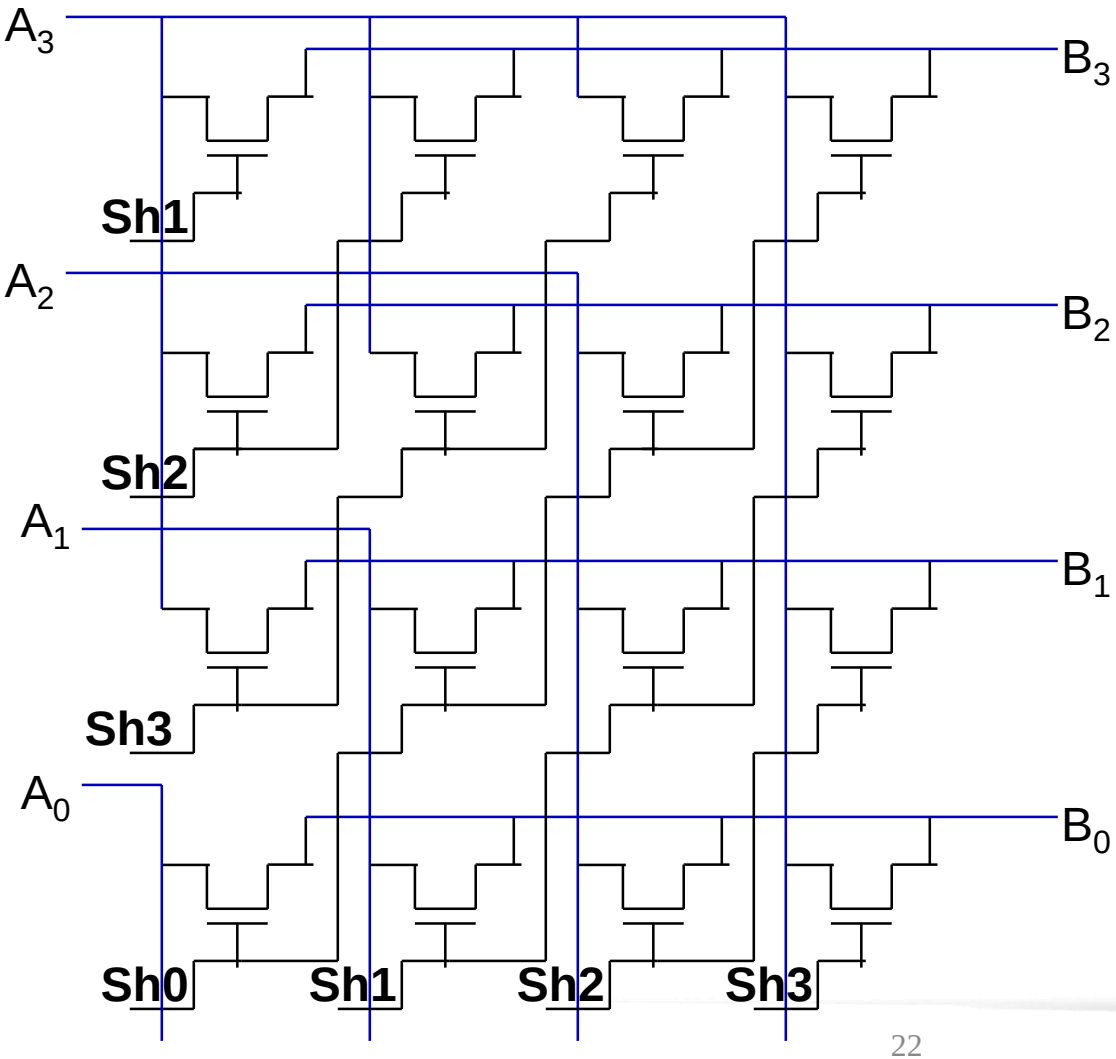
- 移位寄存器（Shift Register）
  - 移位寄存器中的数据可以在移位脉冲作用下一次逐位右移或左移，数据既可以并行输入、并行输出，也可以串行输入、串行输出，还可以并行输入、串行输出，串行输入、并行输出。



移1位需要1个Clock，那移16位呢?

{wdwang, cxh}@ustc.edu.cn

移位能否在1个Clock完成？
4-bit Barrel Shifter



Example:

Sh0 = 1

$B_3B_2B_1B_0 = A_3A_2A_1A_0$

Sh1 = 1

$B_3B_2B_1B_0 = A_3A_3A_2A_1$

Sh2 = 1

$B_3B_2B_1B_0 = A_3A_3A_3A_2$

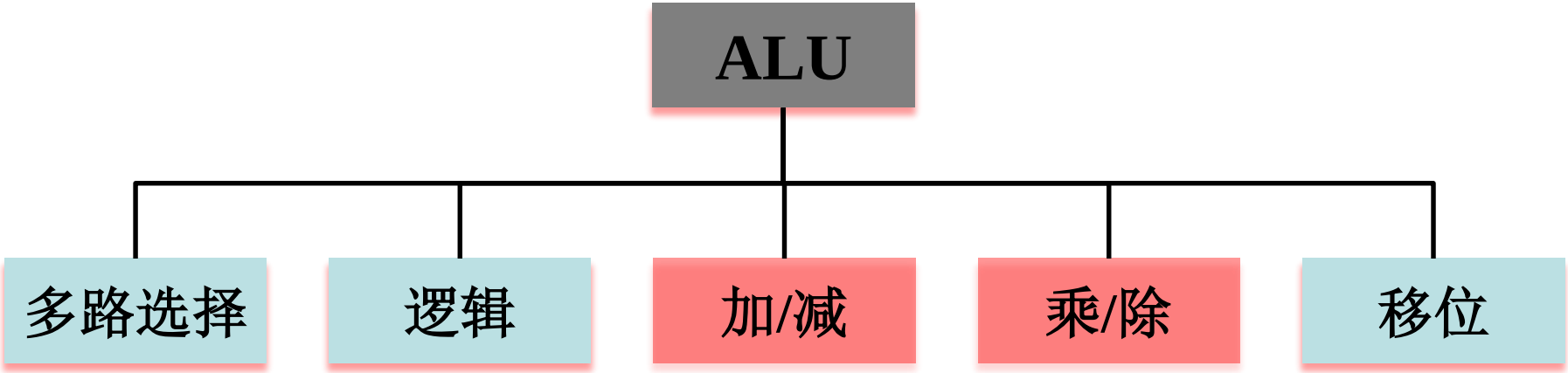
Sh3 = 1

$B_3B_2B_1B_0 = A_3A_3A_3A_3$

{wdwang, cxh}@ustc.edu.cn

小结：ALU构成

- 逻辑运算器：最基本的门
- 多路选择器：MUX + Decoder
- 移位运算：简单的移位寄存器需要多个Clock
- 加法器、减法器、乘法器、除法器



{wdwang, cxh}@ustc.edu.cn

1bit半加器

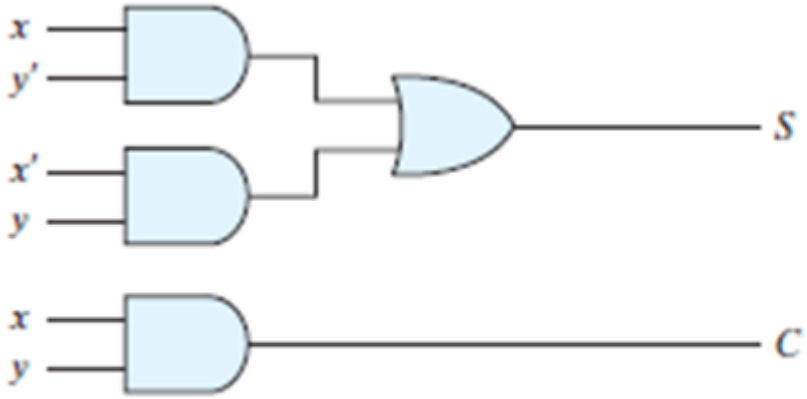
半加器不处理进位
XOR逻辑表达式：A⊕B=A‘B + AB’

```
  0     0     1     1
+ 0   + 1   + 0   + 1
  0     1     1    10
```

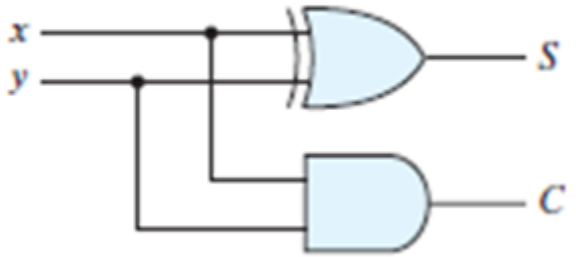
Sum　Carry　Sum

Half Adder

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



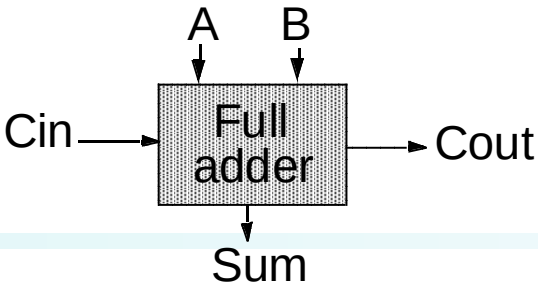
(a) $S = xy' + x'y$
$C = xy$



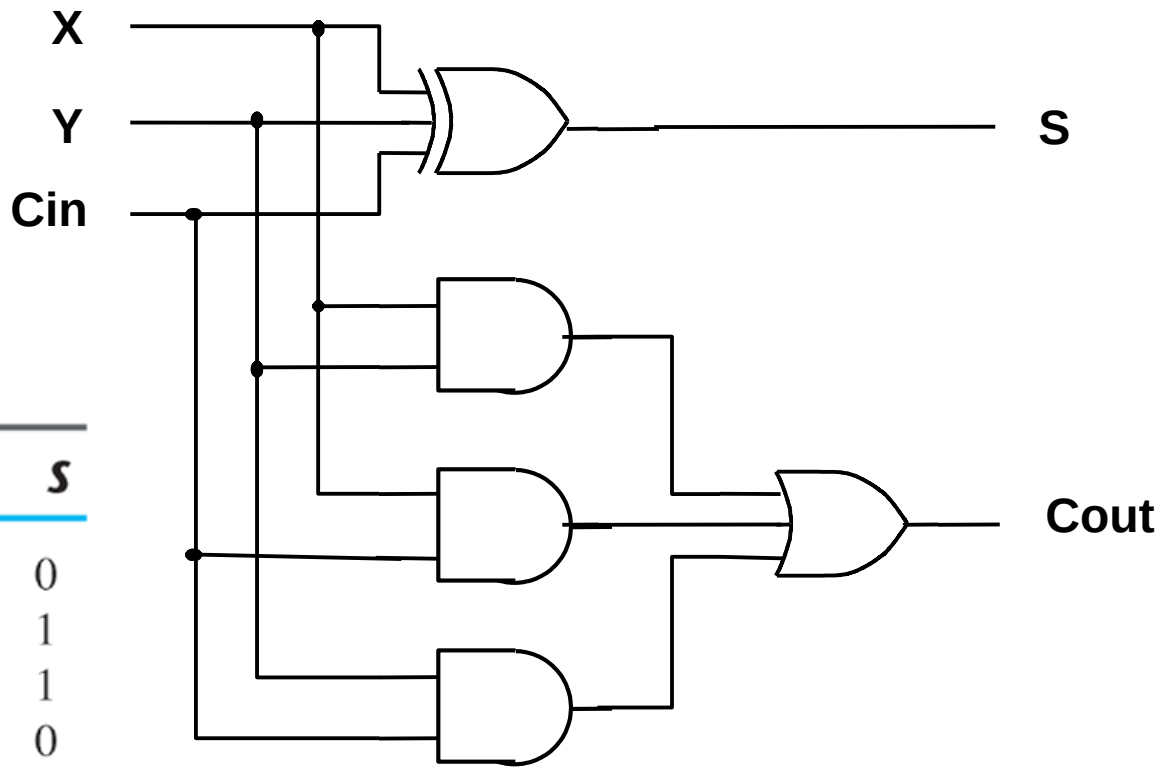
(b) $S = x \oplus y$
$C = xy$

{wdwang, cxh}@ustc.edu.cn

1bit全加器



X、Y为输入
Sum为输出结果
Cin为上次加法进位
Cout为本次加法进位

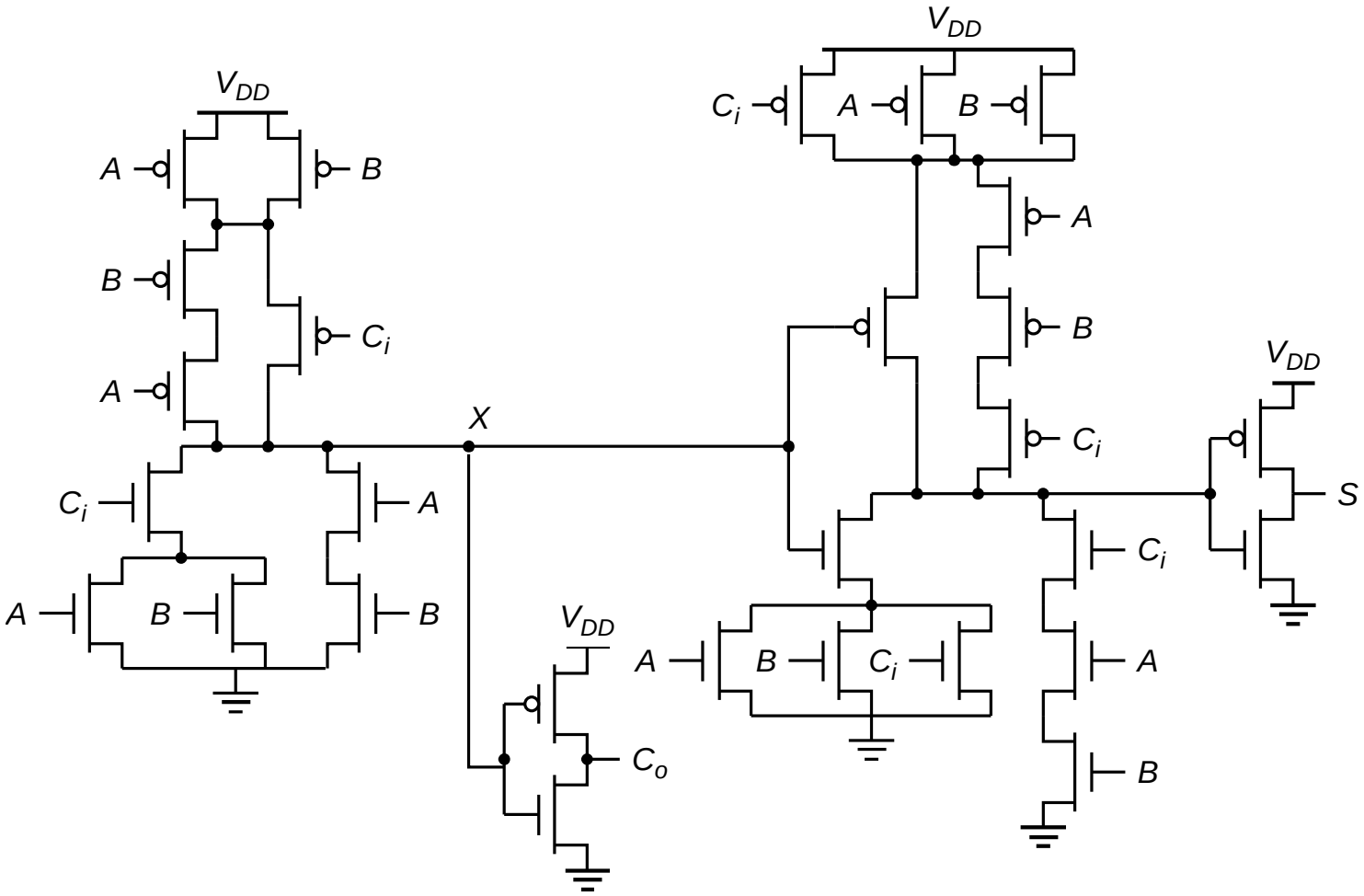


Full Adder

x	y	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

{wdwang, cxh}@ustc.edu.cn

加法器：1bit的全加器的CMOS实现

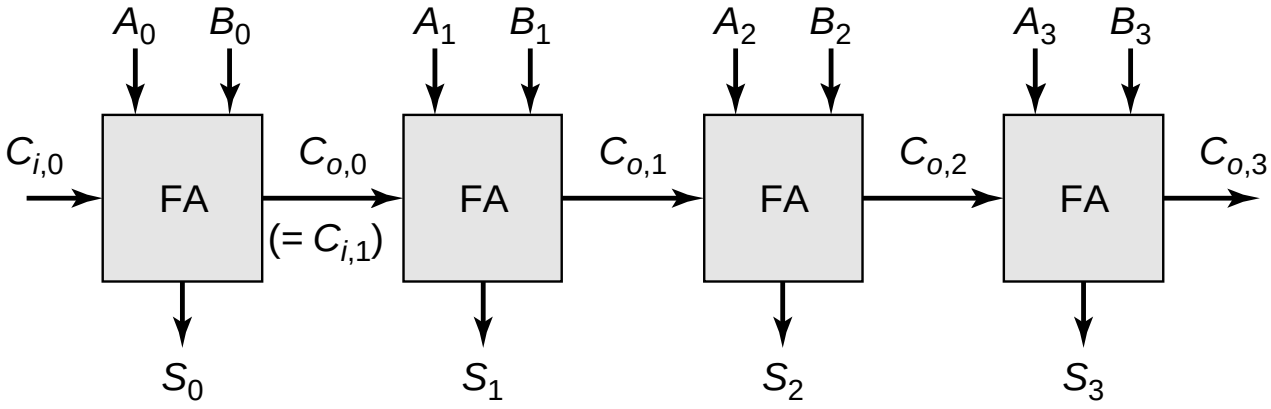


需要28 个MOS管，有更好的办法！

{wdwang, cxh}@ustc.edu.cn

Source http://ece.gmu.edu/~clorie/Spring10/ECE-301/

加法器： Ripple-Carry结构的多bit加法器



本结构加法器的**延时随bit数成线性增长**，与**MOS**管延时有关

$$t_d = O(N)$$

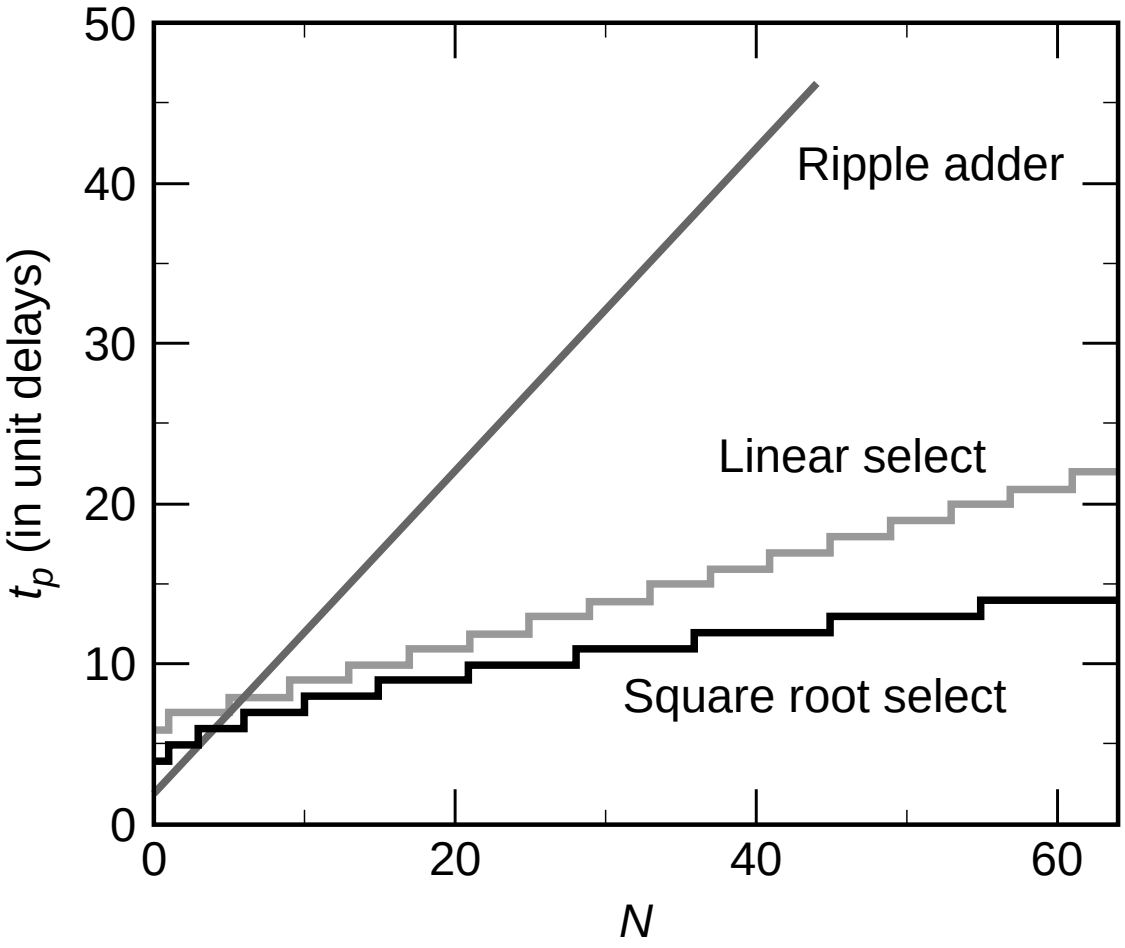
$$t_{adder} = (N-1)t_{carry} + t_{sum}$$

减小延时需要改进！

{wdwang, cxh}@ustc.edu.cn

Source http://ece.gmu.edu/~clorie/Spring10/ECE-301/

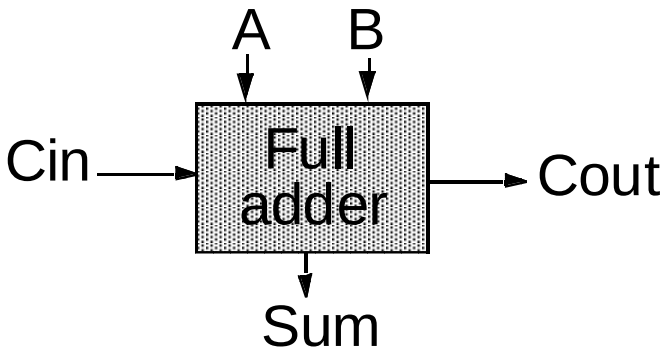
加法器：不同实现方式的时延

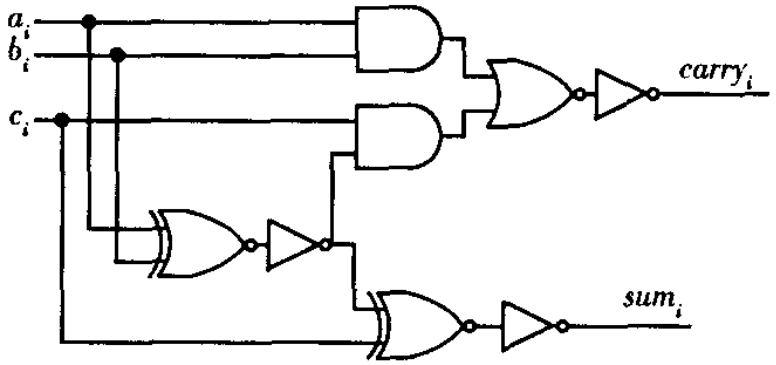


{wdwang, cxh}@ustc.edu.cn

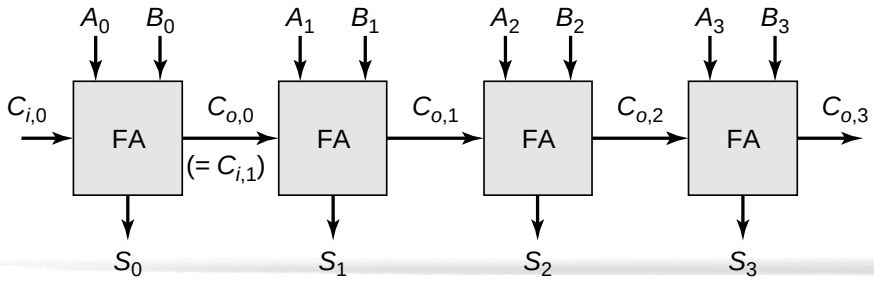
小结：加法器

- 1bit加法：全加器、半加器





- Ripple-Carry结构的多bit加法器
  - 组合电路，延时随bit数成线性增长
  - 改进结构减少MOS管数量



{wdwang, cxh}@ustc.edu.cn

第2章 数字计算电路

- 2.1 概述
- 2.2 基础计算电路
  - 2.2.1 加法电路
  - 2.2.2 从整数到定点小数
    - 如何表示负数？
    - 如何表示小数？
  - 2.2.3 乘法电路
  - 2.2.4 除法电路
  - 2.2.5 浮点数运算电路
  - 2.2.6 超越函数的计算
- 2.3 面向应用的计算电路
- 2.4 异构计算系统
- 2.5 高速计算电路的存储管理

{wdwang, cxh}@ustc.edu.cn

负数在数字电路中如何表示？

- “开”、“关”2种状态→二进制的存储
  - 以8bits整数为例
  - **7←→111，6←→110**

- 那如何表示负整数？
- **$X_sX_1X_2…X_n$**，其中**X_s**表示符号位 **←负数的表示**

- 以8bits整数为例：
  - **+3←→011，+2 ←→ 010**
  - **-3←→111，-2 ←→ 110**

{wdwang, cxh}@ustc.edu.cn

添加符号位带来的问题

- +0 ≠ -0
  - +0 = 0000_0000　-0 = 1000_0000
- 如果符号位参加运算？
  - 1 - 1 = 1 + (-1) ，考虑4bit情形
  - (0001) + (1001) = (1010) = -2

高级语言（如C）中无符号数和有符号数是不同的数据类型

原码表示方法的运算电路设计复杂

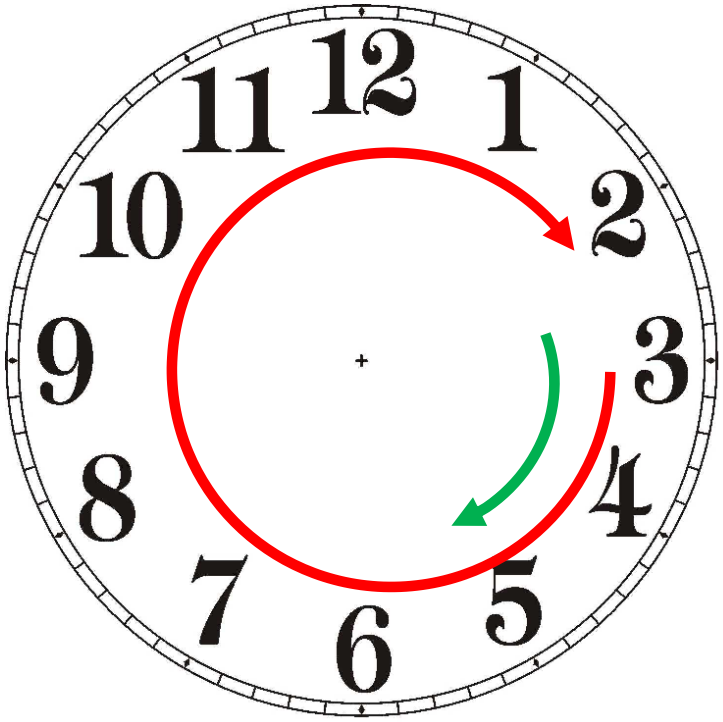
目标：1 - 1 = 1 + (-1)

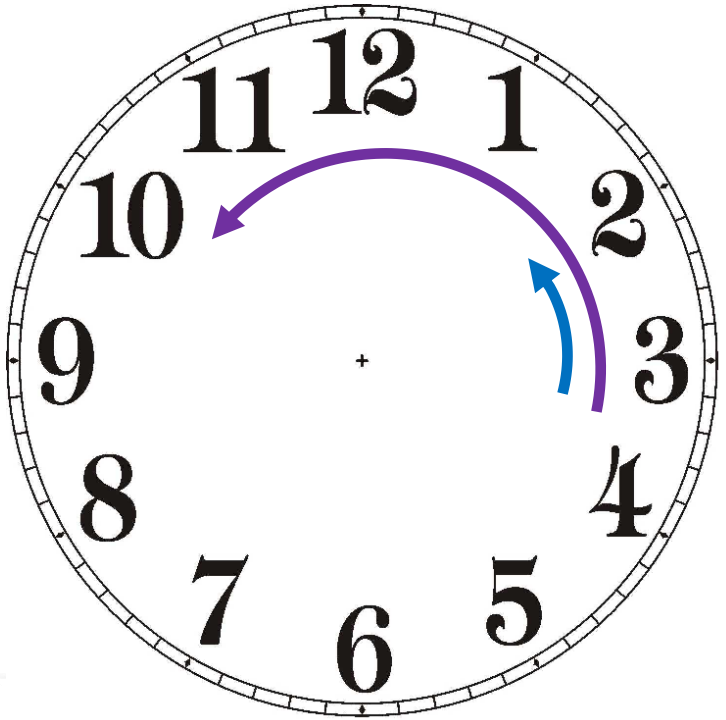
方法：原码？反码？补码？

{wdwang, cxh}@ustc.edu.cn

有限长度存储电路的规律
钟表上的加法和减法

- 3+2 = 5：从3出发，顺时针走3格
- 5+9 = 14 mod 12 = 2：从5出发，顺时针走9格
- 3-2 = 1：从3出发，逆时针走2格
- 3-5 = -2：从3出发，逆时针走5格？

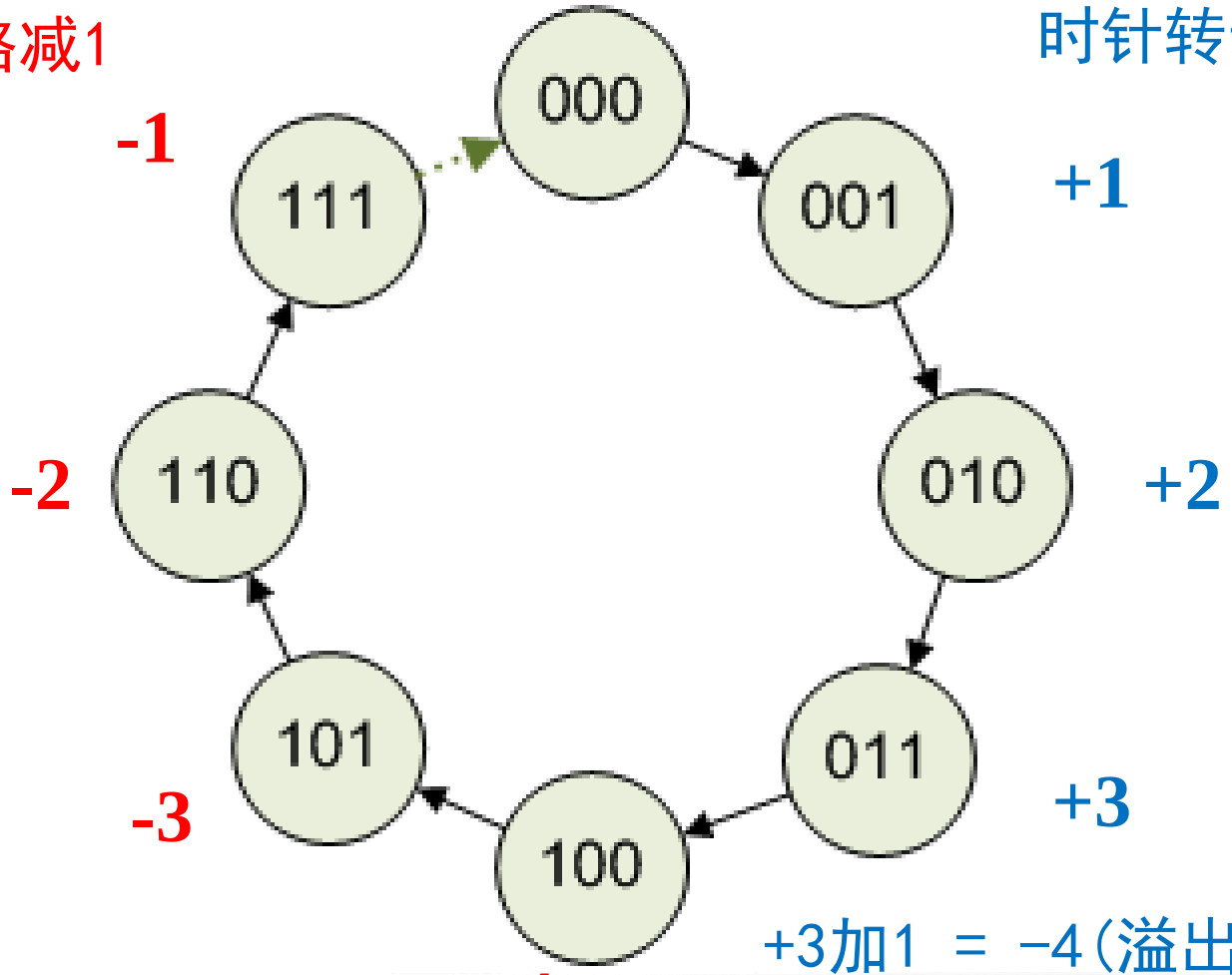




满足钟表运算规则的负数表示，补码最高位作为符号位

钟表哲学中逆时针转动1格减1

钟表哲学中，顺时针转动1格加1



+3加1 = −4（溢出）

−4减1 = +3（溢出）

{wdwang, cxh}@ustc.edu.cn

最高位为符号位：表示正负

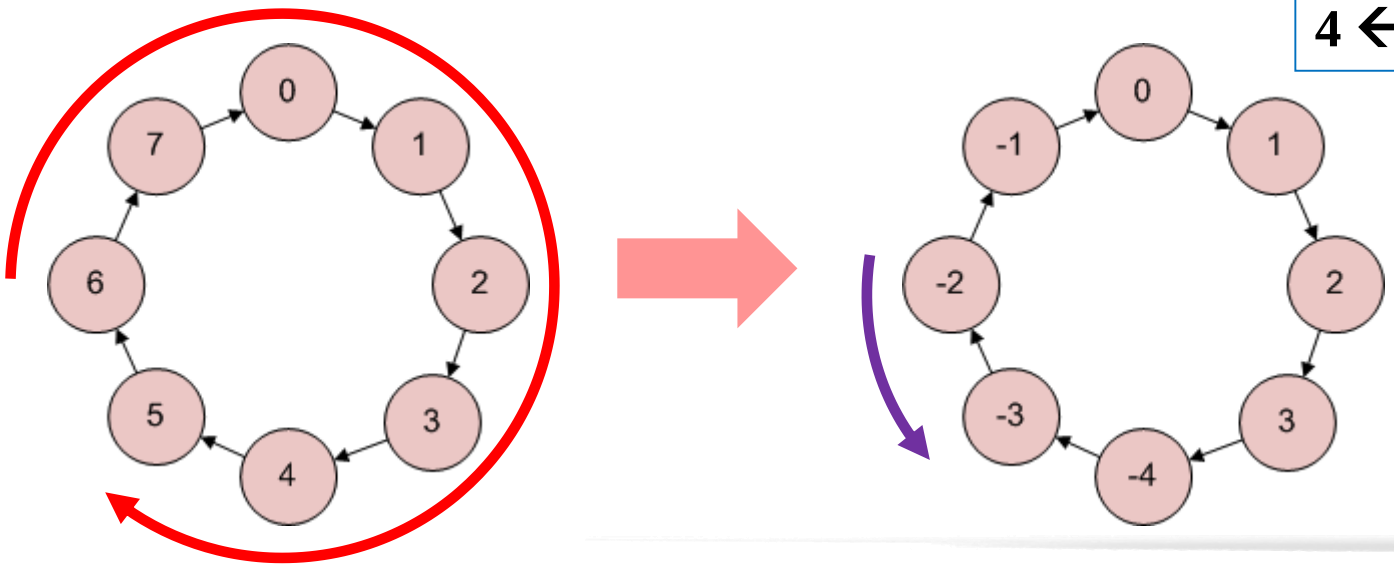
- **unsigned 3-bit integers:**

  6 + 7 → 5

- **signed 3-bit integers:**

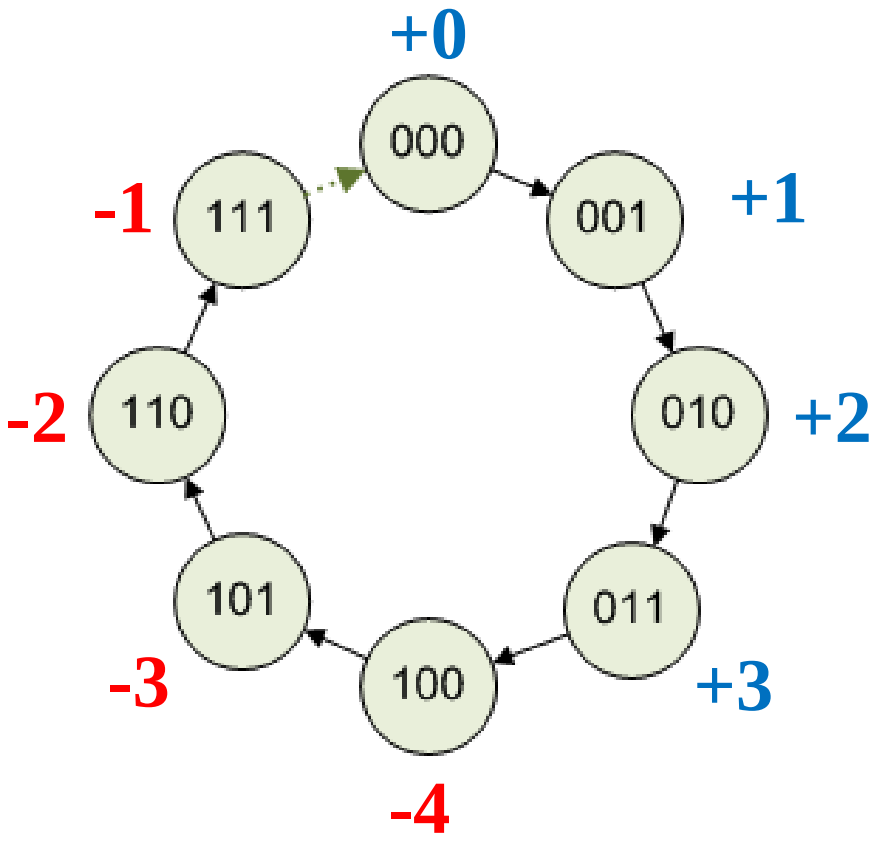
  (-2) + (-1) → (-2) – 1 → -3

7 ←→ -1 ←→ 111
6 ←→ -2 ←→ 110
5 ←→ -3 ←→ 101
4 ←→ -4 ←→ 100



{wdwang, cxh}@ustc.edu.cn

负数取反+1的结果
便捷的补码表示负数的数值计算方法



bit2	bit1	bit0
-2^2	2^{+1}	2^0

0 ←→ 000
1 ←→ 001
2 ←→ 010
3 ←→ 011

正数的有效值是与+0的距离
负数的有效值是与-4的距离

-1 ←→ 111= $-2^2+2^{+1}+2^0$
-2 ←→ 110= -2^2+2^{+1}
-3 ←→ 101= -2^2+2^0
-4 ←→ 100= -2^2

{wdwang, cxh}@ustc.edu.cn

原码、反码和补码的形式化定义

- 原码（true form）编码方案

$$[x]_{原}=\begin{cases} x & 0\le x\le 2^{n-1}-1 \\ 2^{n-1}+|x| & -(2^{n-1}-1)\le x\le 0 \end{cases}$$

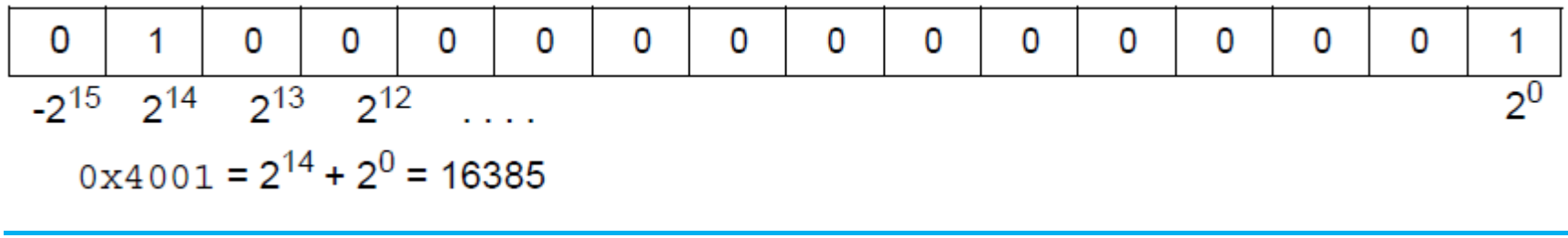
- 反码（inverse code）编码方案

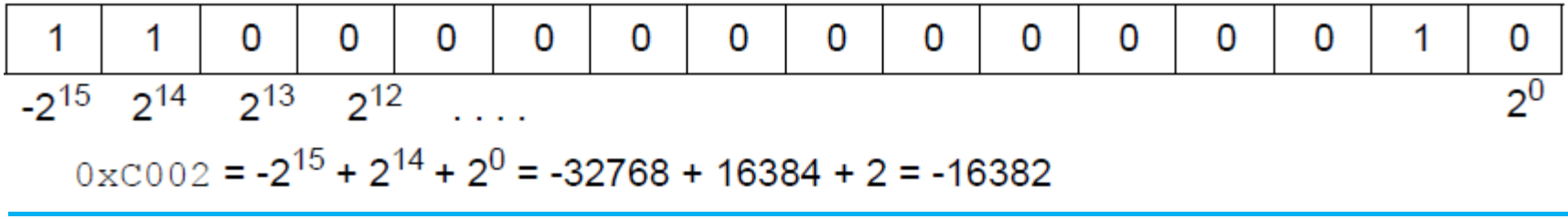
$$[x]_{反}=\begin{cases} x & 0\le x\le 2^{n-1}-1 \\ (2^{n}-1)-|x| & -(2^{n-1}-1)\le x\le 0 \end{cases}$$

- 补码（two‘s complement）编码方案

$$[x]_{补}=\begin{cases} x & 0\le x\le 2^{n-1}-1 \\ 2^{n}-|x| & -2^{n-1}\le x\le 0 \end{cases}$$

16bits整数示例





有符号整数的有效范围：$-2^{15} \sim 2^{15}-1$

即1000_0000_0000_0000 ~ 0111_1111_1111_1111

{wdwang, cxh}@ustc.edu.cn

零扩展和符号扩展

为什么C程序要有强制类型转换

```
short  sh_a;
int  i_b;
i_b = sh_a;
```

- 零扩展（zero-extended）即在高位补“0”
  - 例，8-bit数值“1000 1010”零扩展至16-bit“0000 0000 1000 1010”
- 符号扩展（sign-extended）操作用于保护有符号数的符号位
  - 例，8-bit数值“0000 1010”扩展至16-bit“0000 0000 0000 1010 ”
  - 例，8-bit数值“1000 1010”扩展至16-bit“1111 1111 1000 1010”

```
short int a = -4;
short int b = 8;
print_binary(a);
print_binary(b);
unsigned short int c = a; //类型转换
unsigned int d = a;
print_binary(c);
print_binary(d);
return 0;
```



```
11111111 11111100
00000000 00001000
11111111 11111100
11111111 11111111 11111111 11111100
请按任意键继续. . .
FFFFFFFC
00000008
0000FFFC
FFFFFFFC
```


{wdwang, cxh}@ustc.edu.cn

小结：负数带来的问题

- 补码的表示方法使加减法电路统一
- 补码的表示方法使无符号/有符号数加减法电路统一
  - 如C语言中“unsigned int”和“int”加减法用同一个电路
- 乘、除法电路是否一致？
  - ①符号与数字分离模块
  - ②符号位运算：异或
  - ③乘法运算模块
  - ④符号与数重组模块
- 符号扩展 (Sign extension)
  - 如C语言中强制类型转换涉及符号扩展

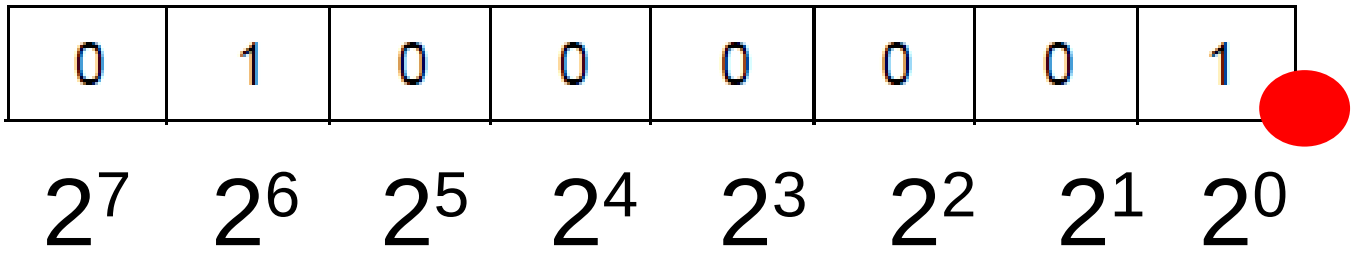
{wdwang, cxh}@ustc.edu.cn

第2章 数字计算电路

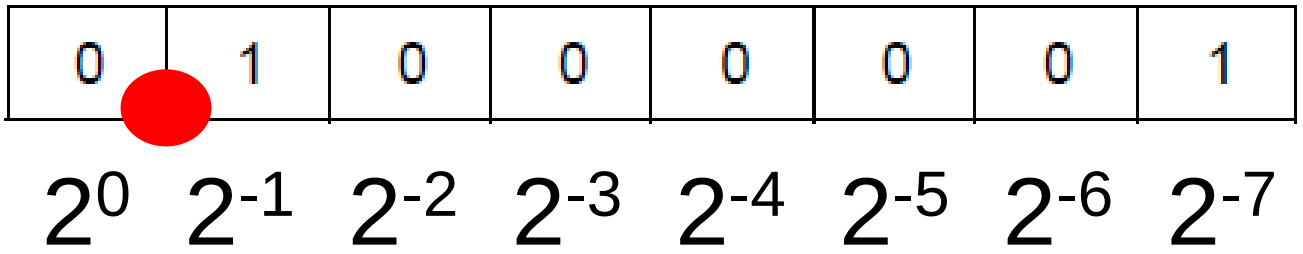
- 2.1 概述
- 2.2 基础计算电路
  - 2.2.1 加法电路
  - 2.2.2 从整数到定点小数
    - 如何表示负数？
    - 如何表示小数？
  - 2.2.3 乘法电路
  - 2.2.4 除法电路
  - 2.2.5 浮点数运算电路
  - 2.2.6 超越函数的计算
- 2.3 面向应用的计算电路
- 2.4 异构计算系统
- 2.5 高速计算电路的存储管理

{wdwang, cxh}@ustc.edu.cn

计算机中表示小数
在二进制数中定义小数点位置



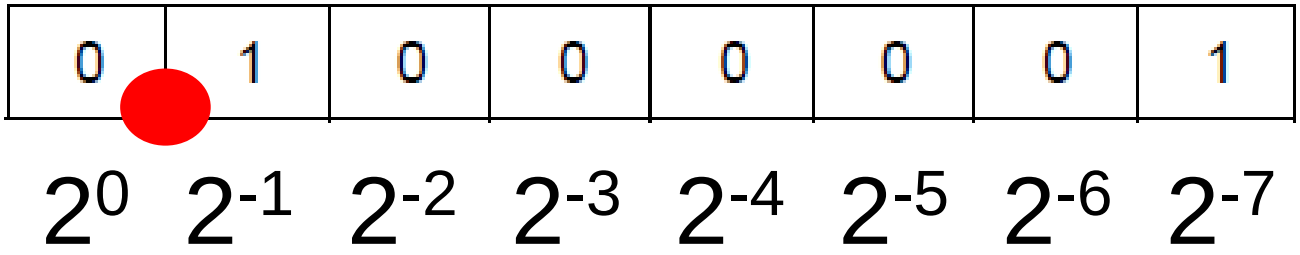
$=2^6 + 2^0$



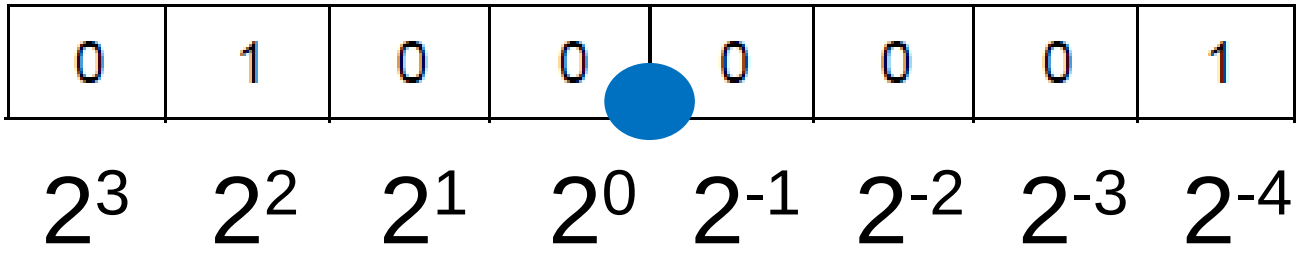
$=2^{-1} + 2^{-7}$

{wdwang, cxh}@ustc.edu.cn

小数点位置不同所代表数值不同



$=2^{-1} + 2^{-7}$



$=2^{2} + 2^{-4}$

{wdwang, cxh}@ustc.edu.cn

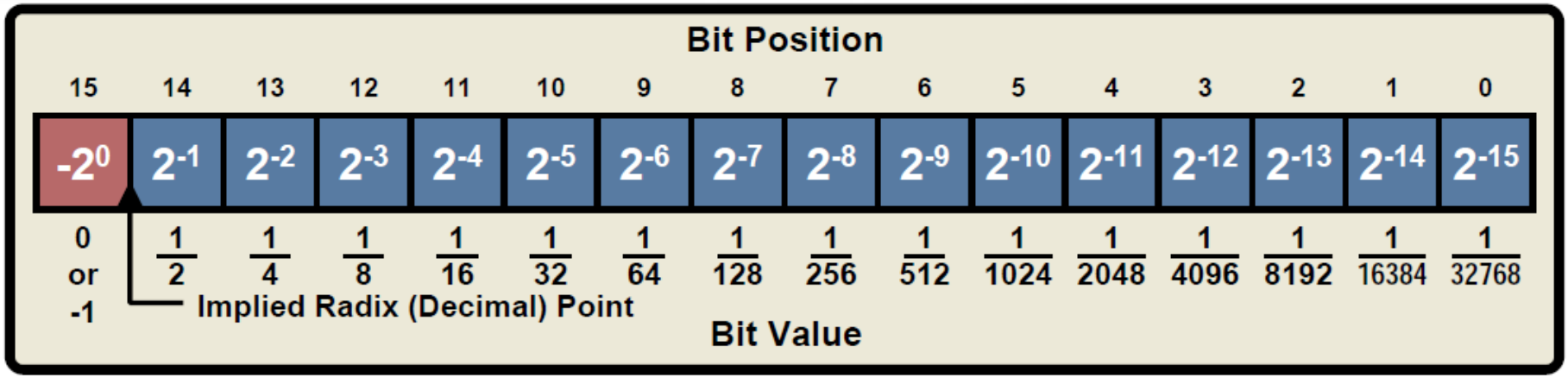
常用的数据表示方式

MSB: Most Significant Bit
LSB: Least Significant Bit

◆**整数**数据表示为带符号的补码值，MSbit 被定义为符号位，N位2 的补码整数的范围为-2^{N-1} 到$2^{N-1}-1$

◆**小数**数据表示为补码，MSbit 被定义为符号位，并**暗示小数点就在符号位**之后。带这种暗示小数点的N位2 的补码小数的范围为-1.0 到（$1-2^{N-1}$）

- 绝对值最大的负数：0x8000 = -1.000000000000
- 绝对值最大的正数：0x7FFF = 0.999969482422

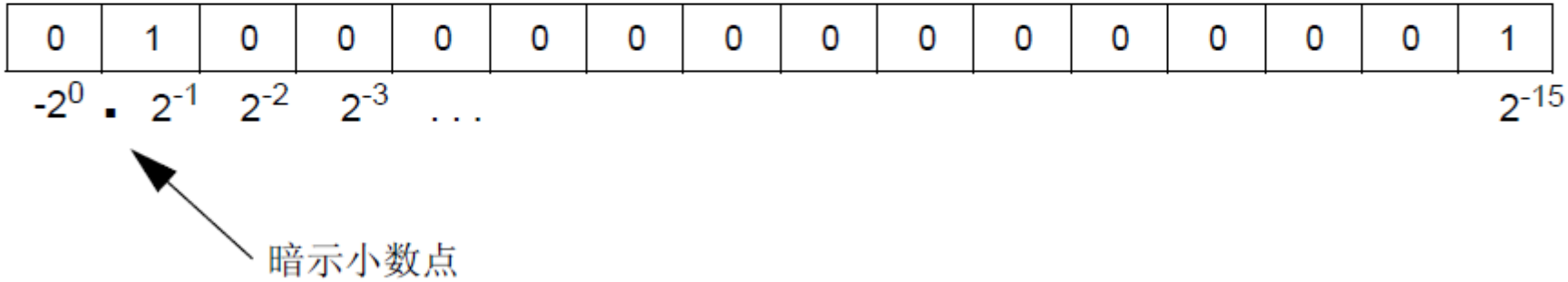


0x4001 的整数和小数表示

整数:

$0x4001 = 2^{14} + 2^{0} = 16385$

1.15 小数:



$x4001 = 2^{-1} + 2^{-15} = .500030518$

{wdwang, cxh}@ustc.edu.cn

0xC002 的整数和小数表示

0xC002 的不同表示

整数:

1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0

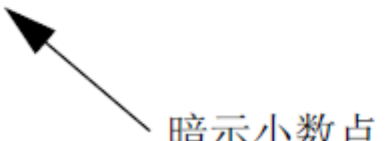
-2^{15} 2^{14} 2^{13} 2^{12} . . . . 2^{0}

0xC002 = $-2^{15} + 2^{14} + 2^{0}$ = -32768 + 16384 + 2 = -16382

1.15 小数:

1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0

-2^{0} . 2^{-1} 2^{-2} 2^{-3} . . . 2^{-15}



0xC002 = $-2^{0} + 2^{-1} + 2^{-14}$ = -1 + 0.5 + 0.000061035 = -0.499938965

{wdwang, cxh}@ustc.edu.cn

其他小数表示方法，及其数据范围

◆Q格式表示法：**Qm.n**，其中m表示整数位的位数，n表示小数位的位数。例如：**Q.15**代表1个符号位+0个整数位+15个小数位=16bit。

◆不同长度的整数和小数数据的表示范围和各自的精度如下

寄存器大小	整数范围	小数范围	小数精度
16 位	-32768 到 32767	-1.0 到 （$1.0-2^{-15}$） （Q.15 格式）	3.052×10^{-5}
32 位	-2,147,483,648 到 2,147,483,647	-1.0 到 （$1.0-2^{-31}$） （Q.31 格式）	4.657×10^{-10}
40 位	-549,755,813,888 到 549,755,813,887	-256.0 到 （$256.0-2^{-31}$） （带 8 保护位的 Q.31 格式）	4.657×10^{-10}

{wdwang, cxh}@ustc.edu.cn

整数和无符号小数的运算是一致的
可用相同的电路予以实现

1	0	1

0	1	0

101+010 → 111

-3 + 2 → -1 (有符号)

5 + 2 → 7 (无符号)

1	0	1

0	1	0

101+010 → 111

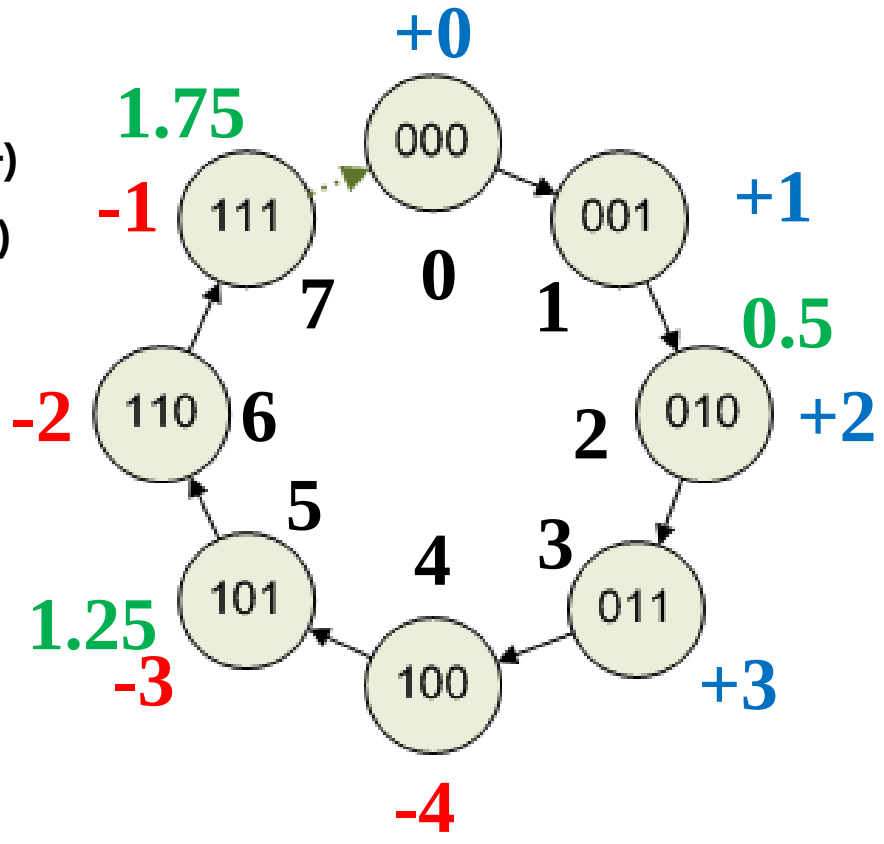
1.25 + 0.5 → 1.75

1	0	1

0	1	0

101+010 → 111

2.5 + 1.0 → 3.5



这种固定小数点位置的方法即定点小数

定点数的优点
有符号小数的运算与整数运算也相同

- a = 1.5 → 001.10 ← A = 6
- b = 0.25 → 000.01 ← B = 1
- c = -0.25 → 111.11 ← C = -0.25
- a + b = 00110 + 00001 = 001.11 → 1.75
- a + c = 00110 + 11111 = 001.01 → 1.25
- A + B = 00110 + 00001 = 00111 → 7
- A + C = 00110 + 11111 = 00101 → 5

- 整数和小数的运算是统一的！

{wdwang, cxh}@ustc.edu.cn

小结：整数和定点小数的关系

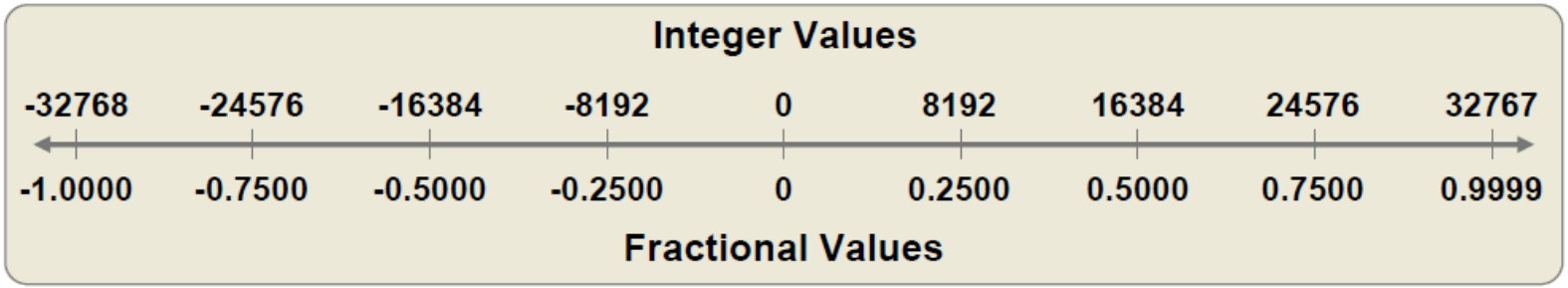
Word Value	Integer Value	Fractional Value
0x8000	-32768	-1.000000
0xA000	-24576	-0.750000
0xC000	-16384	-0.500000
0xE000	-8192	-0.250000
0x0000	0	0.000000
0x2000	8192	0.250000
0x4000	16384	0.500000
0x6000	24576	0.750000
0x7FFF	32767	0.999969

16bit的补码可表示为整数或定点小数。采用相同的算术和逻辑硬件可以同时处理这两种格式的数。其区别在于结果的解释方法不同。

$$x_{FRACTION} = x_{INTEGER} / 2^{N-1}$$

$$x_{INTEGER} = x_{FRACTION} \cdot 2^{N-1}$$

For N=16 bits, $2^{N-1} = 32768$



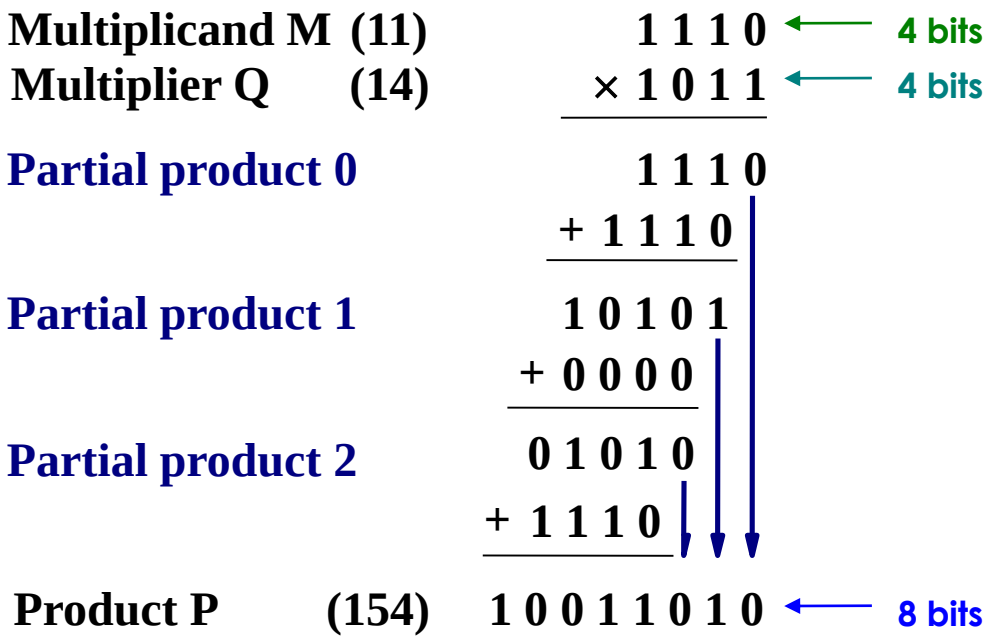
{wdwang, cxh}@ustc.edu.cn

第2章 数字计算电路

- 2.1 概述
- 2.2 基础计算电路
  - 2.2.1 加法电路
  - 2.2.2 从整数到定点小数
  - 2.2.3 乘法电路
  - 2.2.4 除法电路
  - 2.2.5 浮点数运算电路
  - 2.2.6 超越函数的计算
- 2.3 面向应用的计算电路
- 2.4 异构计算系统
- 2.5 高速计算电路的存储管理

{wdwang, cxh}@ustc.edu.cn

乘法器：乘法原理



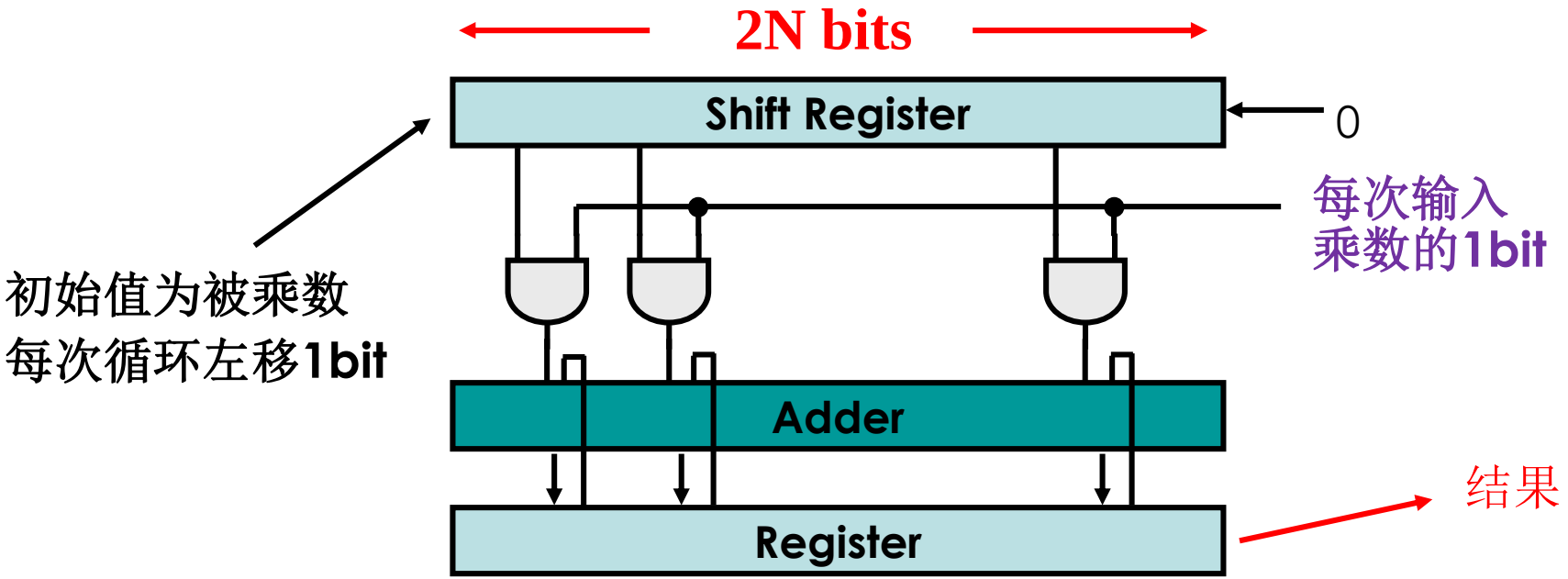
# of bits in P = # of bits in M + # of bits in Q

2个N位数相乘，产生N个部分积，需要N-1次加法

{wdwang, cxh}@ustc.edu.cn

乘法器实现：串行移位乘法器

- 移位寄存器最初存放被乘数
- 每次左移得到一个部分积
- 每次移位将乘数的一个比特送到与门上

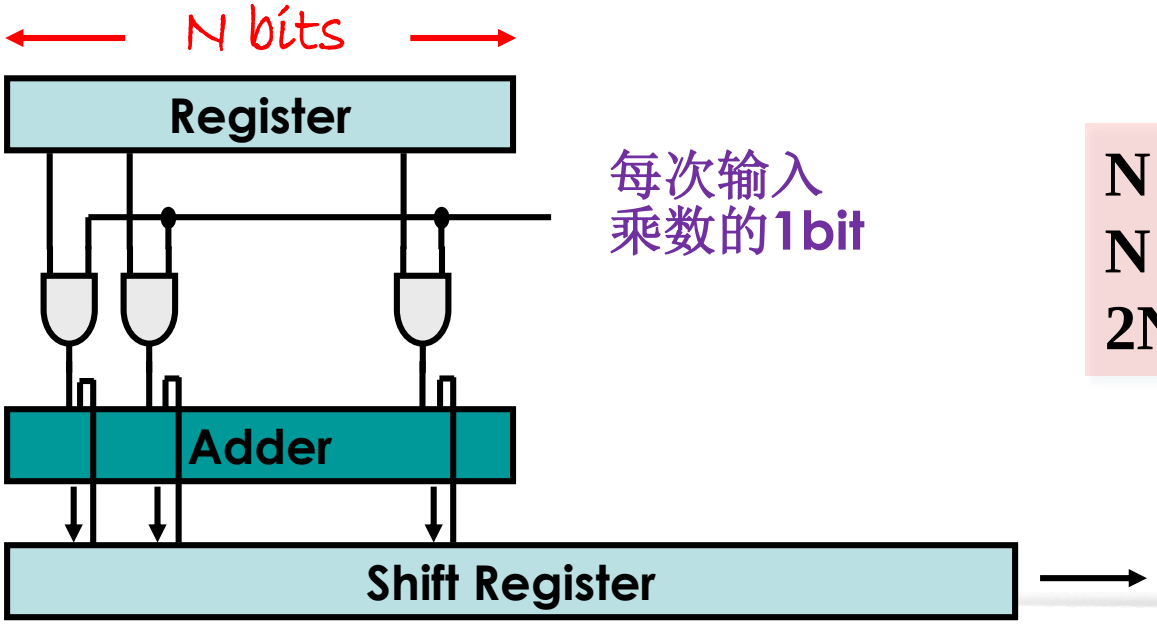


{wdwang, cxh}@ustc.edu.cn

乘法器实现：串行移位乘法器改进

- Shift result register to right
- Uses N AND gates
- Uses N-bit adder

2N bits Register
2N bits Adder
2N bits Shift Register



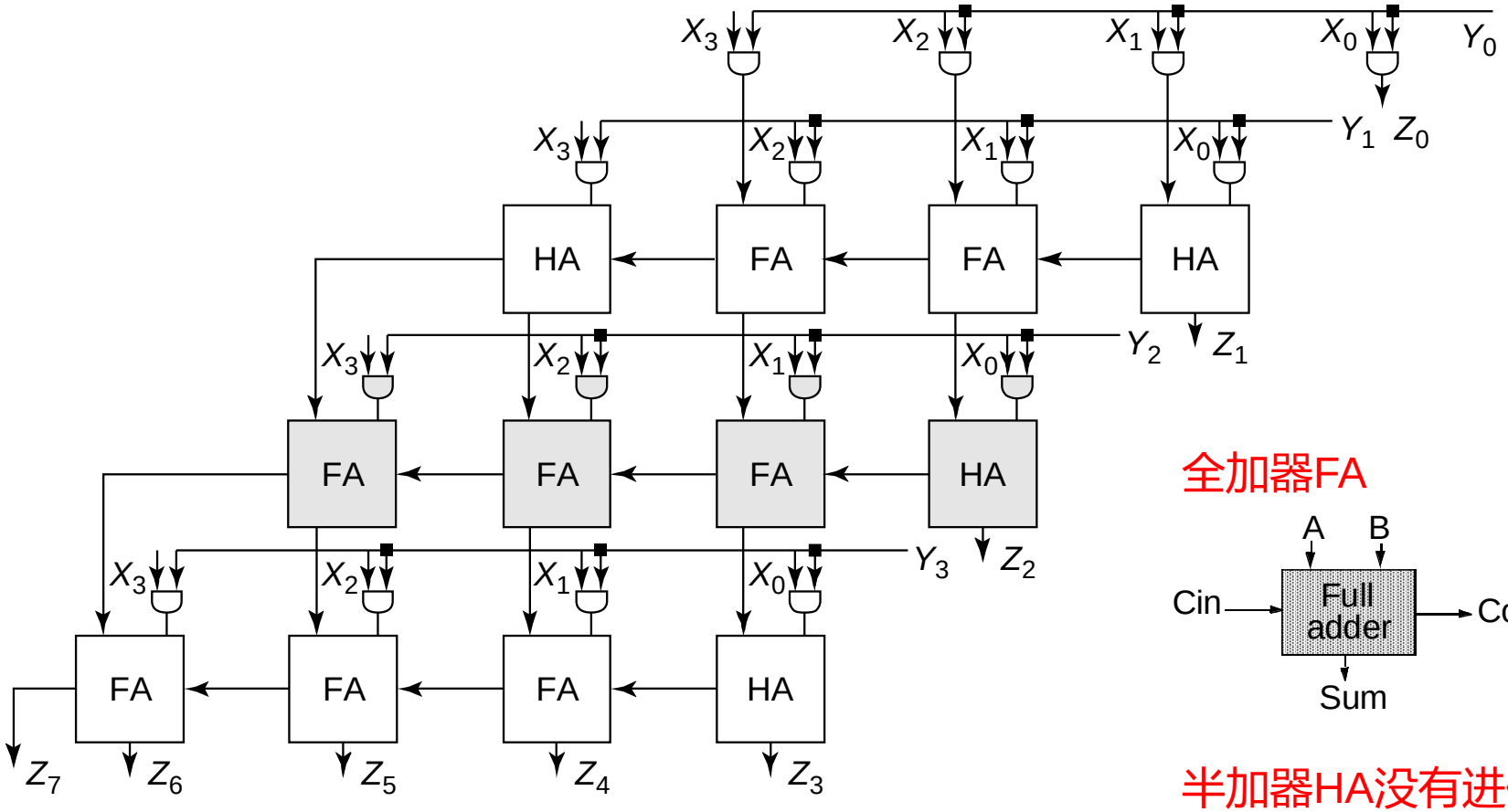
每次输入
乘数的1bit

N bits Register
N bits Adder
2N bits Shift Register

{wdwang, cxh}@ustc.edu.cn

快速乘法器
组合逻辑实现结构：阵列（Array）结构

- 4bit的乘法有3次加法运算



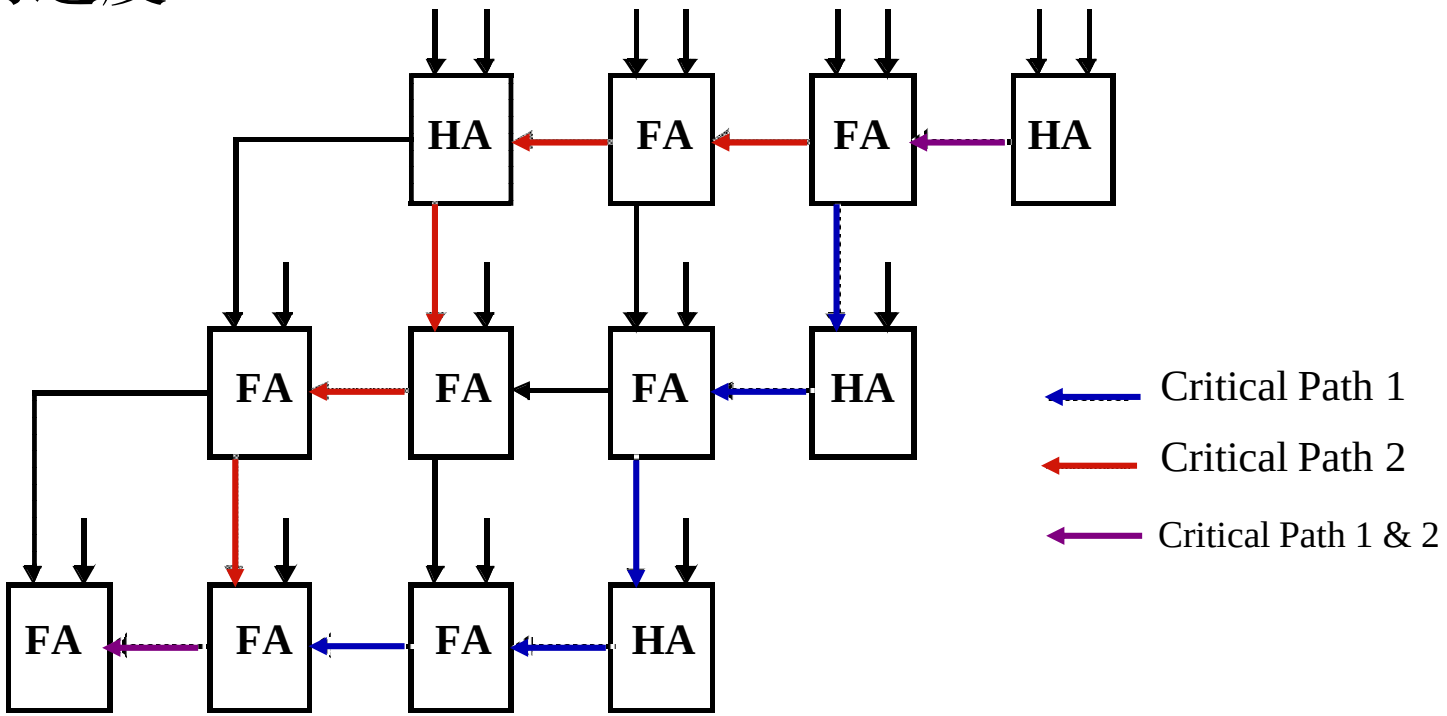
全加器FA

半加器HA没有进位

{wdwang, cxh}@ustc.edu.cn

快速乘法器：Array结构能有多快？
Critical Path

◆阵列乘法采用移位与求和算法，部分乘积项(Partial Product，PP)数目决定了求和运算的次数，直接影响乘法器的速度。

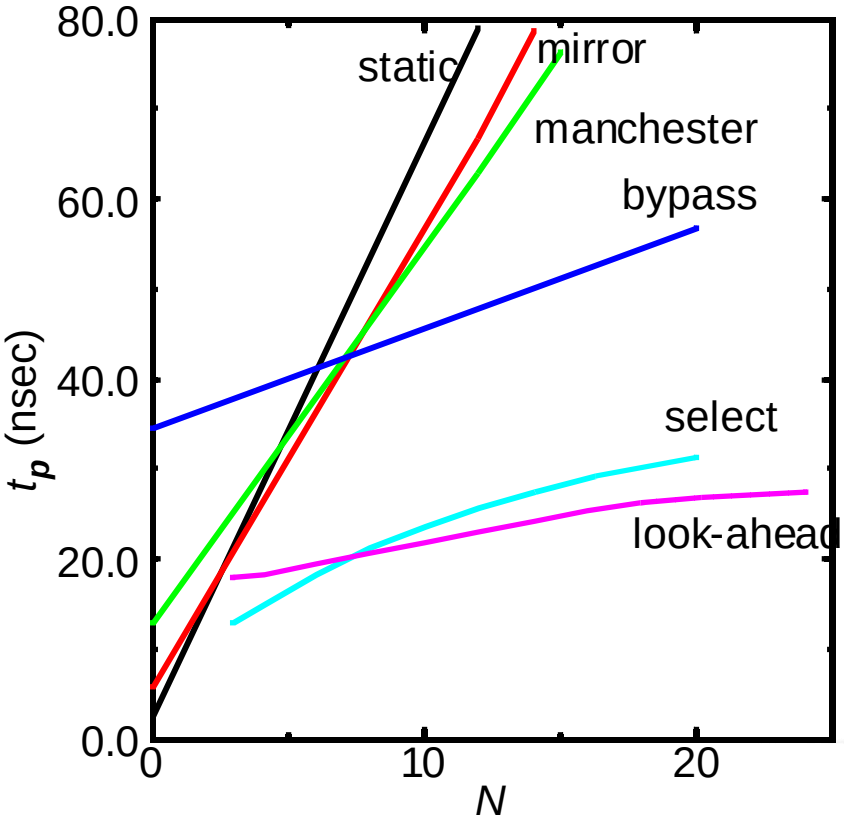


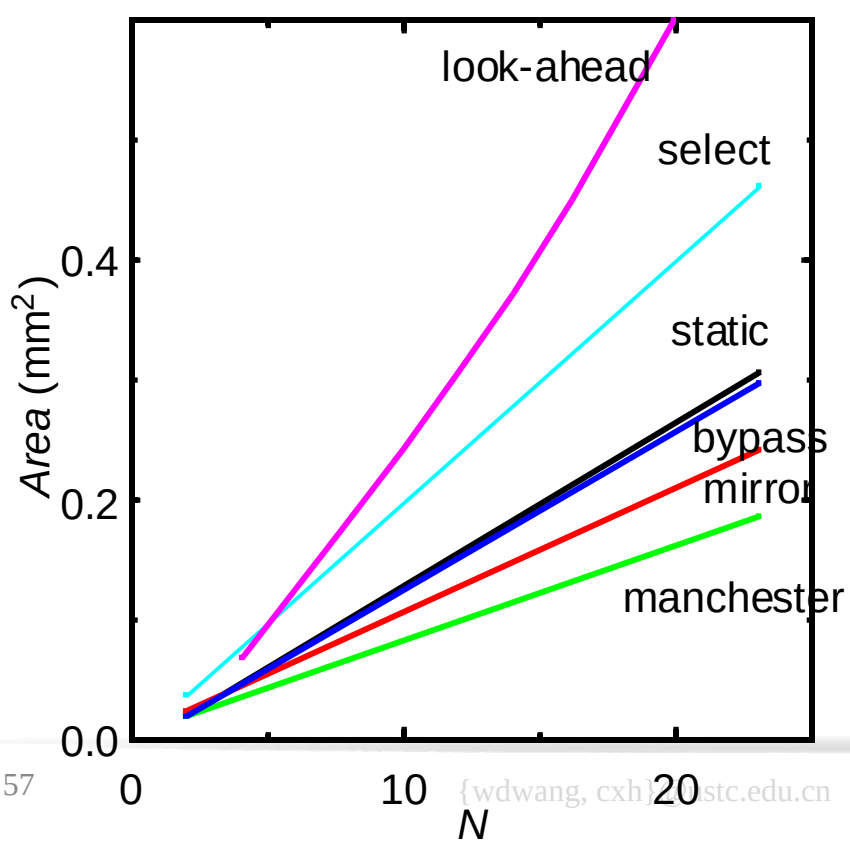
$$t_{mult} \approx [(M-1)+(N-2)]t_{carry}+(N-1)t_{sum}+(N-1)t_{and}$$

{wdwang, cxh}@ustc.edu.cn

快速乘法器有多种实现结构

◆纯组合逻辑构成的乘法器速度快，但占用的硬件资源也相对较多，很难实现宽位数据的乘法器。为了**平衡速度和面积**，有很多改进型如树形乘法器和桶形移位乘法器等。←引入寄存器





{wdwang, cxh}@ustc.edu.cn

小结：乘法器

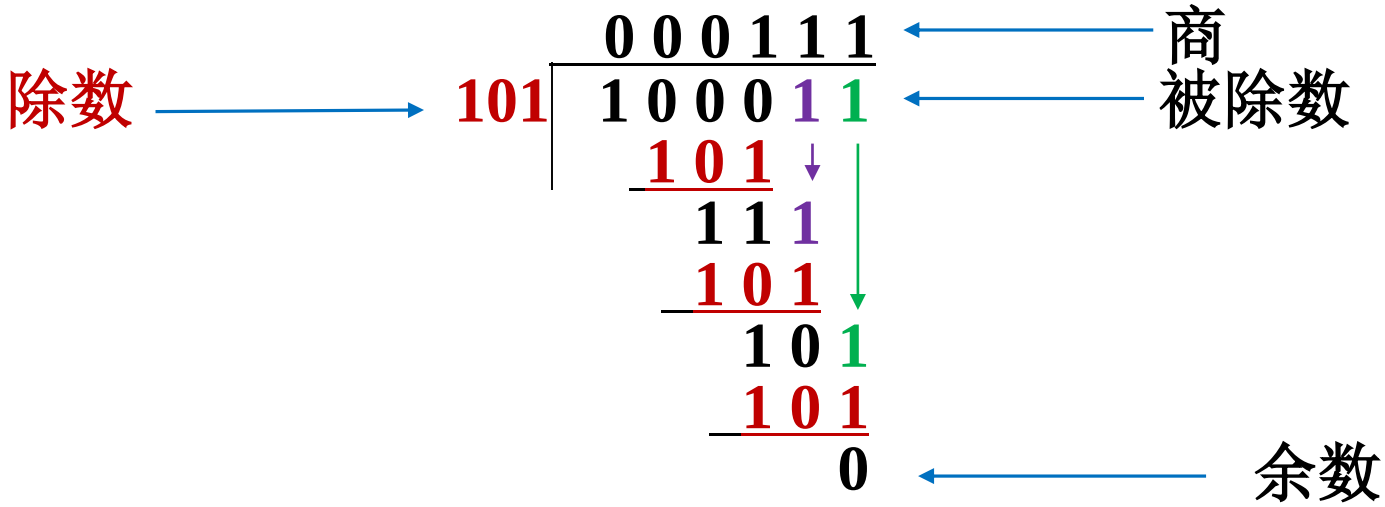
- 乘法器
  - 乘法原理：部分和累加→串行移位乘法器
- 组合逻辑的快速乘法器
  - 组合逻辑的Array结构，延时与位宽呈线性关系
  - 组合电路的Critical Path
- 改进型快速乘法器
  - 速度与乘法器占芯片面积的平衡（Trade off）
  - 改进型常引入寄存器以实现流水线来加快速度

{wdwang, cxh}@ustc.edu.cn

除法器：除法原理

与十进制类似，从除数的最高位开始检查，并定出需要超过除数的位数。找到这个位时商记1，并用选定的被除数减除数。然后把被除数的下一位移到余数上。若余数不够减，则商记0，然后把被除数的下一位移到余数上；若余数够减除数，则商1，余数去减除数，这样反复进行，直至全部被除数的位都下移完为止。

例：100011÷101=?



{wdwang, cxh}@ustc.edu.cn

串行除法器

◆除法**思路**：“用被除数减除数，如果够减，商加1。差再减除数，直到不够减为止。每减一次，商就加1。”

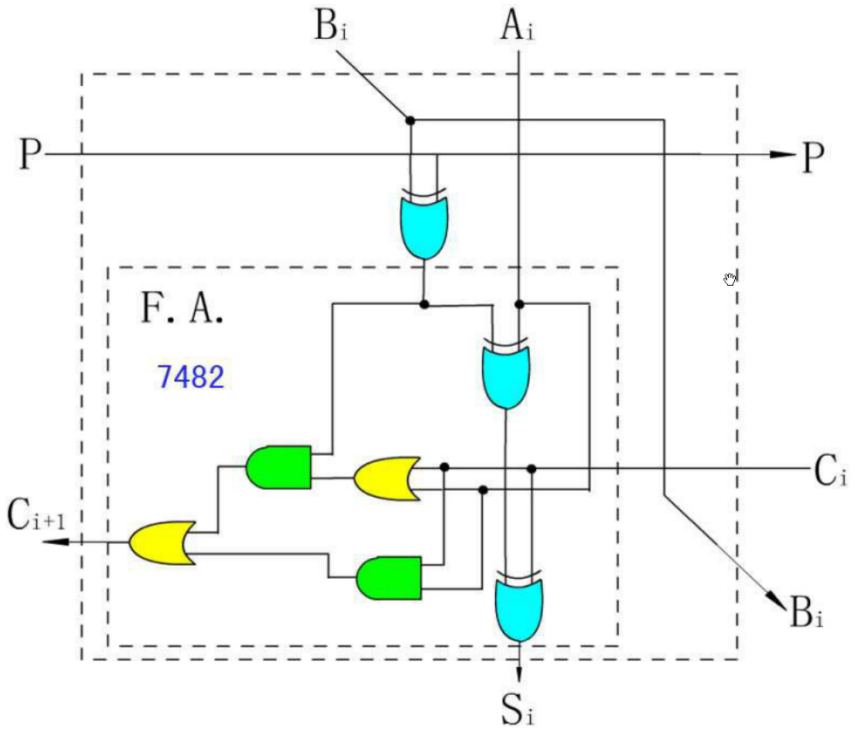
◆人会心算一看就知道够不够减。机器却不会心算，必须先作减法，若余数为正才知道够减；若余数为负才知道不够减。不够减时必须恢复原来的余数，以便再继续往下运算。这种方法称为**恢复余数法**。要恢复原来的余数,只要当前的余数加上除数即可。但由于要恢复余数,使除法进行过程的步数不固定，因此控制比较复杂。

◆实际中常用**不恢复余数法**，又称加减交替法。其特点是运算过程中如出现不够减，则不必恢复余数，根据余数符号，可以继续往下运算，因此步数固定，控制简单

{wdwang, cxh}@ustc.edu.cn

阵列式除法器
可控加法/减法(CAS)单元

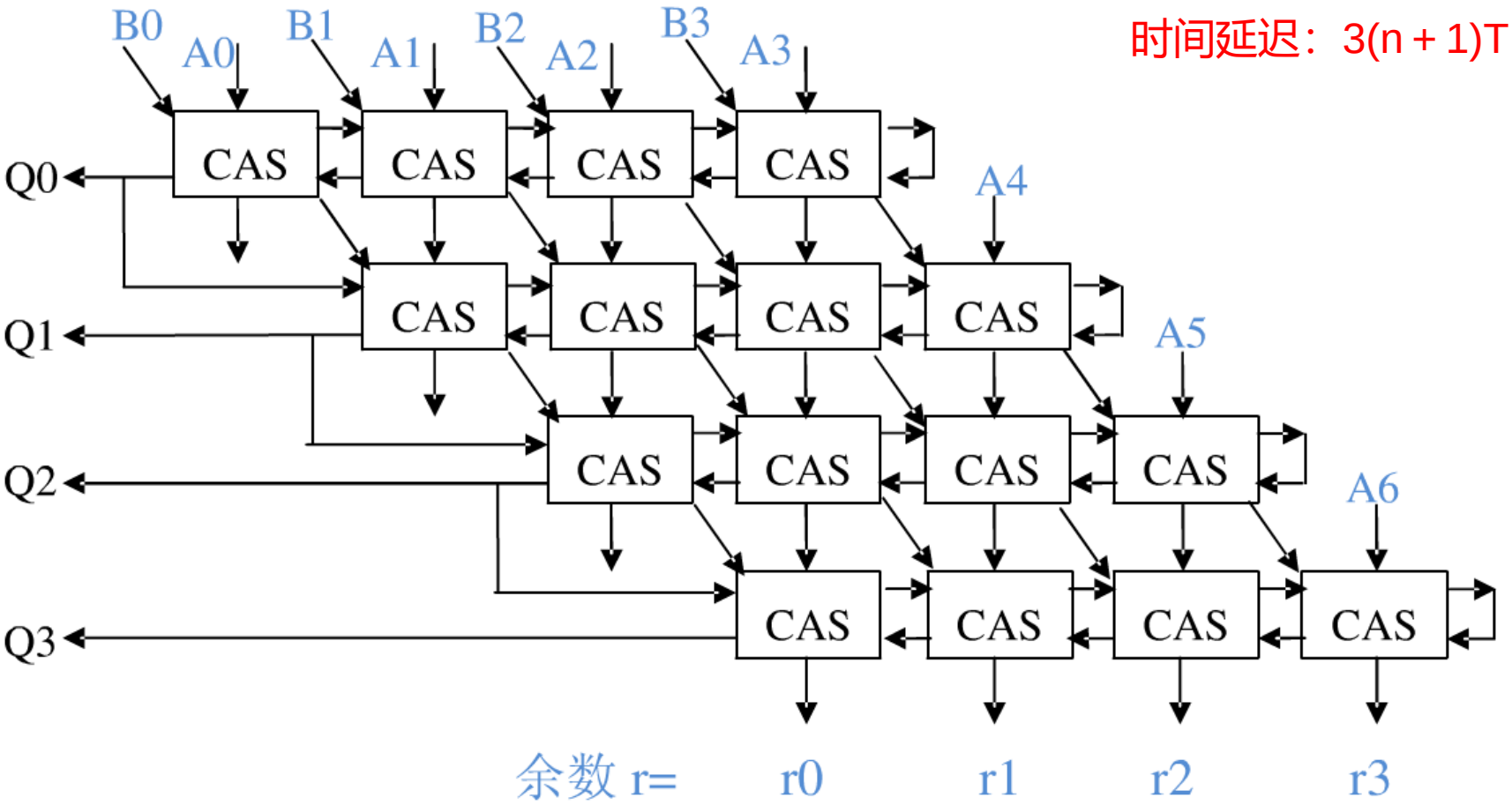
◆可控加法/减法(CAS)单元，它将用于并行除法流水逻辑阵列中，它有四个输出端和四个输入端。当输入线P＝0时，CAS作加法运算；当P＝1时，CAS作减法运算。



本位输入被除数A_i及除数B_i，低位来进位（或借位）信号C_i，加减法控制命令P，输出本位和（差）S_i及进位信号C_{i+1}

{wdwang, cxh}@ustc.edu.cn

阵列式除法器
不恢复余数的阵列除法器



时间延迟：3(n + 1)T

A/B：除数B，被除数A，Q为商，r为余数

{wdwang, cxh}@ustc.edu.cn

小结：除法器

◆**思路：**“用被除数减除数，如果够减，商加1。差再减除数，直到不够减为止。每减一次，商就加1。”

- 多数关于计算电路的教材中都没有关于除法电路的知识点解析，仅笼统概括为除法电路可以用乘法电路来实现。但事实并不简单，在很多MCU、DSP及MPU中有乘法指令，却没有提供除法指令。许多提供了除法指令的处理器，也需要很多个时钟周期才可以完成一次除法运算。

	Dividend	Divisor	Quotient	Remainder	Cycles
DIV r8	AX	r8	AL	AH	25
DIV r16	DX:AX	r16	AX	DX	26-30
DIV r32	EDX:EAX	r32	EAX	EDX	26-38
DIV r64	RDX:RAX	r64	RAX	RDX	38-123

英特尔Silvermont微架构中除法指令的计算时延

{wdwang, cxh}@ustc.edu.cn

第2章 数字计算电路

- 2.1 概述
- 2.2 基础计算电路
  - 2.2.1 加法电路
  - 2.2.2 从整数到定点小数
  - 2.2.3 乘法电路
  - 2.2.4 除法电路
  - 2.2.5 浮点数运算电路
  - 2.2.6 超越函数的计算
- 2.3 面向应用的计算电路
- 2.4 异构计算系统
- 2.5 高速计算电路的存储管理

{wdwang, cxh}@ustc.edu.cn

2.2.5 浮点小数的计算电路

- 浮点小数的表示方法
  - 浮点数的格式
  - 浮点数的精度
- 浮点数的运算
  - 在定点处理器上如何实现浮点运算？
  - 如何设计浮点运算电路？

{wdwang, cxh}@ustc.edu.cn

十进制浮点数

◆浮点数格式使用科学计数法表示实数。科学计数法把数字表示为系数(coefficient)(也称为尾数(mantissa))，和指数 (exponent)两部分。

◆例如，25.92 可表示为 $2.592 * 10^1$，其中 2.592 是系数，值 10^1 是指数。必须把系数和指数相乘，才能得到原始的实数。

◆再如， 0.00172 可表示为 $1.72*10^{-3}$，数字 1.72 必须和 10^{-3} 相乘才能获得原始值。

{wdwang, cxh}@ustc.edu.cn

二进制浮点数

5.25 → 1.0101*2^2

- 下表列出几个二进制小数及它们对应的十进制值：

二进制		十进制分数	十进制值
1. 0101	定点	1+1/4+1/16	1.3125
10.101	形式	2+1/2+1/8	2.625
101.01		5+1/4	5.25
1101.011		13+1/4+1/8	13.375

- 二进制浮点数使用二进制科学计数法的格式表示数值。如，1.0101 乘以 2^2 后，生成二进制值 101.01，这个值表示二进制整数 5，加上分数 (0/2) + (1/4) 。这生成十进制值 5.25 。

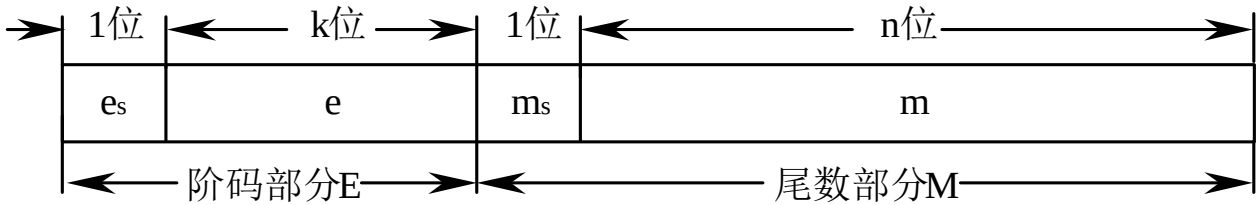
{wdwang, cxh}@ustc.edu.cn

二进制浮点数的规格化

◆编写二进制浮点值时，二进制通常被**规格化**了。这个操作把小数点移动到最左侧的数位，并且修改指针进行补偿。例如 1101.011 变成 $1.101011*2^3$

- →只需要保存两部分

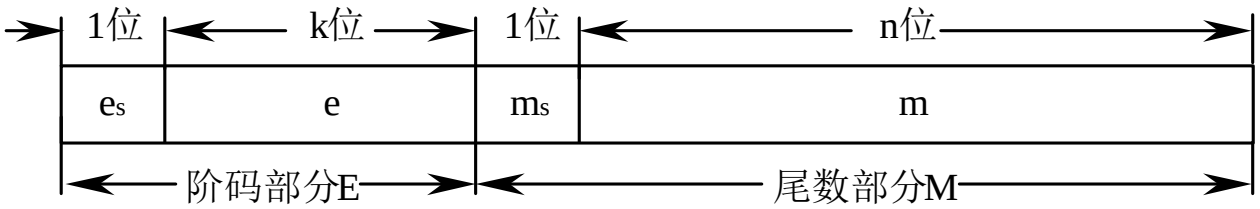
◆浮点数： $N=M\times r^E$ 。式中：r为浮点数阶码的底，与尾数的基数相同，通常r=2。E和M都是带符号数，E叫做阶码，M叫做尾数。



{wdwang, cxh}@ustc.edu.cn

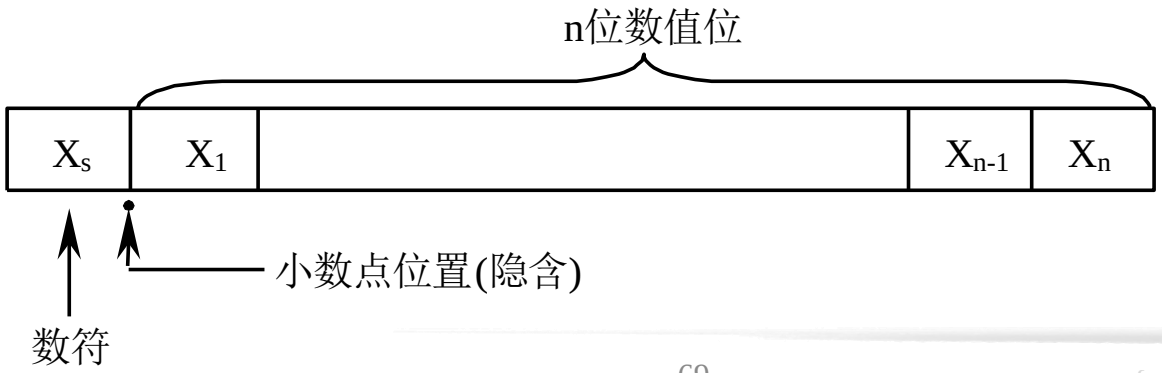
对比：定点与浮点表示的基本思路

◆浮点数： $N=M\times r^E$ 。式中：r为浮点数阶码的底，与尾数的基数相同，通常r=2。E和M都是带符号数，E叫做阶码，M叫做尾数。



◆定点小数：小数点的位置固定在最高有效数位之前，符号位之后，记作$X_s.X_1X_2\ldots X_n$，补码定点小数表示范围：$-1\sim(1-2^{-n})$

- 定点整数：小数点隐含固定在最低位之后，表示范围：$-2^n\sim(2^n-1)$



{wdwang, cxh}@ustc.edu.cn

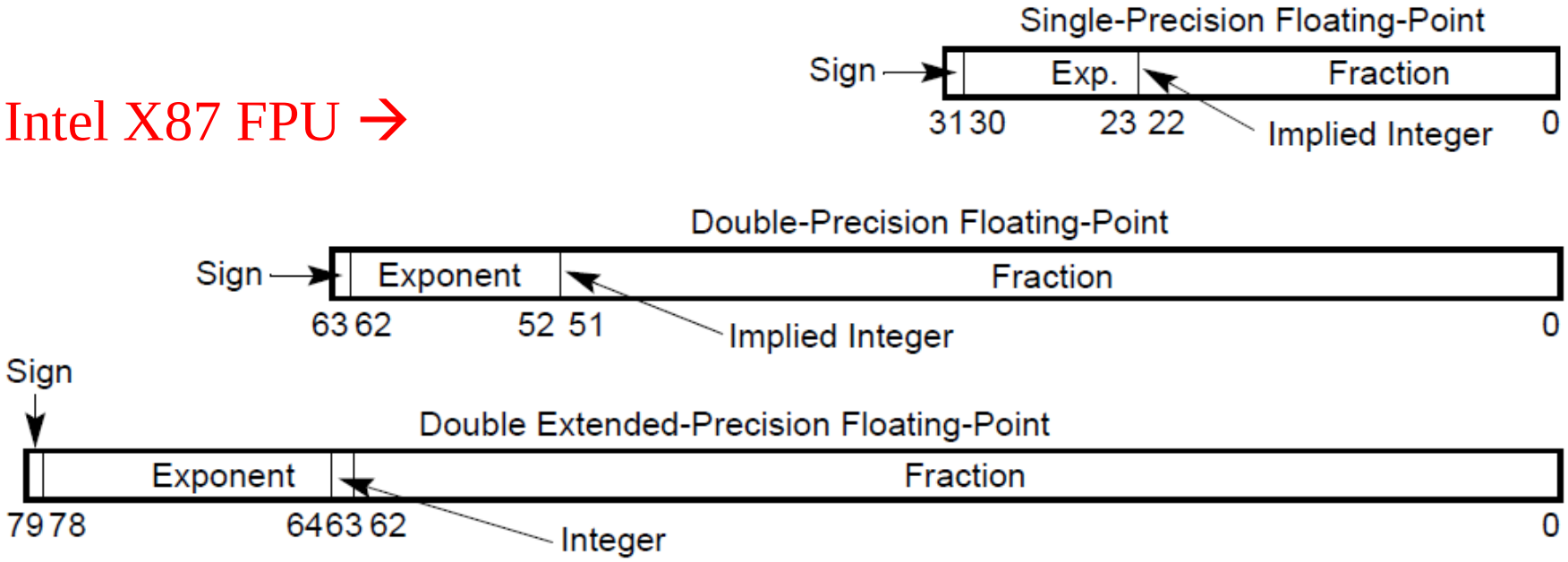
浮点数格式 $N=M\times r^E$，E叫做阶码，M叫做尾数

	Sign	Exponent	Fraction	Bias
Single Precision	1 [31]	8 [30-23]	23 [22-00]	127
Double Precision	1 [63]	11 [62-52]	52 [51-00]	1023

← IEEE Standard 754

偏置量，稍后解释

Intel X87 FPU →



{wdwang, cxh}@ustc.edu.cn

William Kahan

William Kahan, who received the Turing Award in 1989 for "his fundamental contributions to numerical analysis". A primary architect of the Intel 80x87 floating point coprocessor and IEEE 754 floating point standard.

Kahan is now emeritus professor of mathematics and of electrical engineering and computer sciences (EECS) at the University of California, Berkeley.

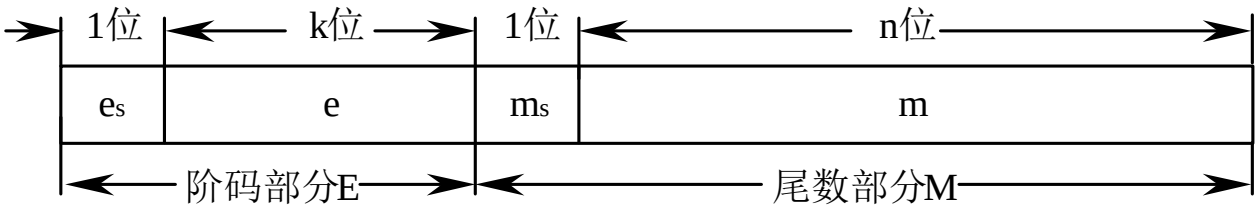
He has been called "The Father of Floating Point," since he was instrumental in creating the original IEEE 754 specification.

William Morton Kahan

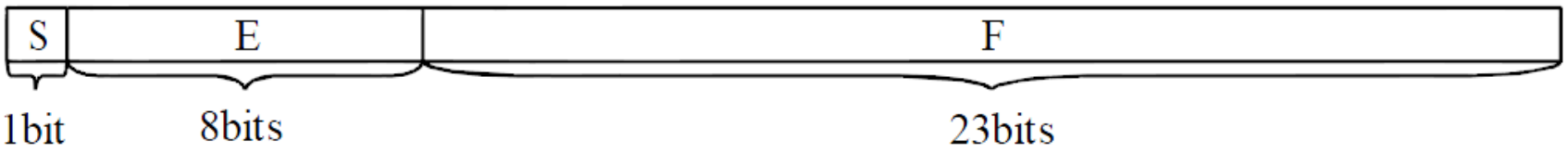
Kahan in 2008

Born	June 5, 1933 (age 88) Toronto, Ontario, Canada
Nationality	Canadian
Alma mater	University of Toronto
Known for	IEEE 754 Kahan summation algorithm
Awards	Turing Award (1989) IEEE Emanuel R. Piore Award[1] (2000) National Academy of Engineering ACM Fellow

IEEE 754 float 32
阶码为何不需要符号位？



◆IEEE 754 是最广泛使用的二进制浮点数算术标准，被许多CPU 与浮点运算器所采用。IEEE 754 规定了多种表示浮点数值的方式，以下为32bits 的float 浮点类型。它被分为3 个部分，分别是符号位S（sign bit）、指数偏差E（exponent bias）和小数部分F（fraction）。



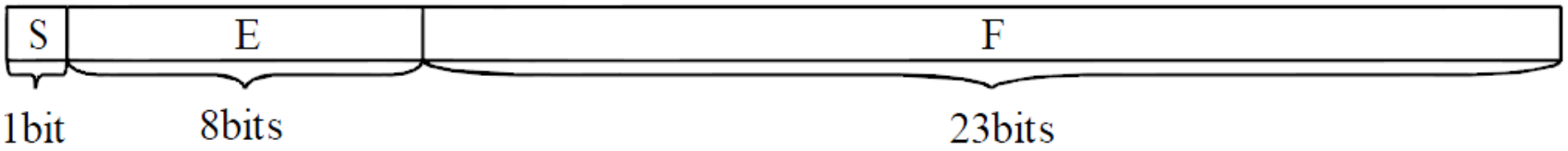
从二进制数换算到浮点数的公式为：

$(-1)^S \times 2^{E-127} \times (1+F)$

{wdwang, cxh}@ustc.edu.cn

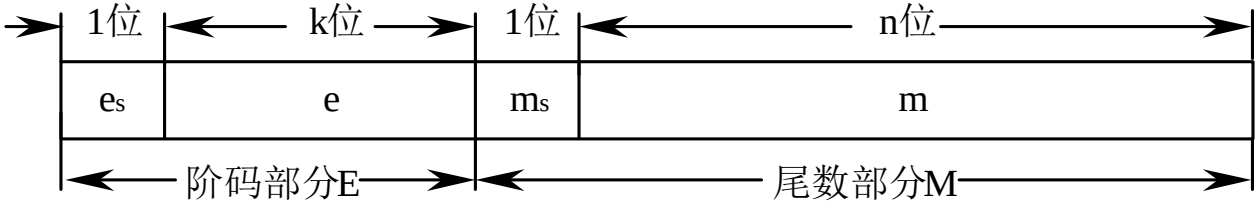
IEEE 754 float 32示例：如何表示0.5

- 为避免计算错误，方便理解，常将E当成二进制真值存储。如将数值-0.5按IEEE754单精度格式存储。
- -0.5（10进制）
- = -0.1（2进制）
- = -1.0×2^{-1}（2进制，-1是指数）
- 这里s=1，M为全0，E-127=-1，E=126（10进制）=01111110（2进制），则存储形式为：
- **1 01111110 00000000000000000000000**
- =BF000000（16进制）

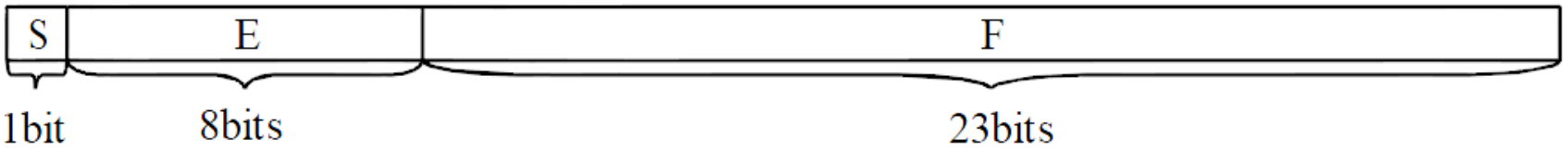


小结：浮点数在数字电路中的表示

- 浮点数格式：科学计数法
  - 十进制：如 0.00172 可表示为 $1.72*10^{-3}$
  - 二进制：5.25 → 101.01
  - 规格化：5.25 → 101.01 → $1.0101*2^2$



- IEEE 754 float
  - 阶码不需要符号位
  - William Kahan, Turing Award 1989



2.2.5 浮点小数的计算电路

- 浮点小数的表示方法
  - 浮点数的格式
  - 浮点数的精度
- 浮点数的运算
  - 在定点处理器上如何实现浮点运算？
  - 如何设计浮点运算电路？

{wdwang, cxh}@ustc.edu.cn

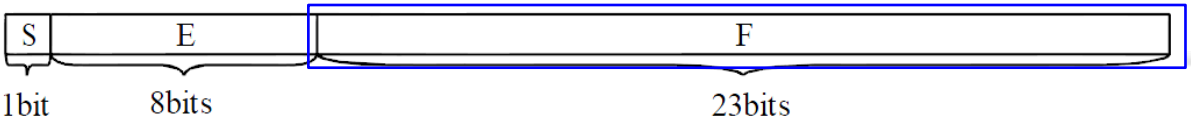
浮点数的精度

Float数如何表示1.00000002?
Impossible!
但是定点数可以

◆可以看到1.xx 这个数量级的最小数是2^{-23}，对应的十进制数值为1.00000011920928955078125，可以精确表示到小数点后23 位，但有些C 语言书上却说float型的有效位只有6~7 位，这是为什么？

二进制小数	十进制小数
2^{-23}	1.00000011920928955078125
2^{-22}	1.00000023841857910156 25
2^{-21}	1.00000047683715820312 5
2^{-20}	1.00000095367431640625
2^{-19}	1.00000190734863281 25
2^{-18}	1.00000381469726562 5

二进制小数对于十进制小数来说相当于是离散的



浮点数的精度：两个例子

```
int main(void)
{
    float a = 9.87654321;
    float b = 9.87654322;

    if(a > b)
    {
        printf("a > b\n");
    }
    else if(a == b)
    {
        printf("a == b\n");
    }
    else
    {
        printf("a < b\n");
    }

    return 0;
}
```

a =0x411E0652
b=0x411E0652
走了a == b 的分支

```
int main(void)
{
    float a = 10.2;
    float b = 9;
    float c;

    c = a - b;

    printf("%f\n", c);

    return 0;
}
```

c=0x3F999998
实际上c 变量的值是1.1999998
将printf 函数%f 格式改为%.7f 格式，
会看到c 变量输出的值是1.1999998

浮点数的ULP

◆任意两个实数之间是有无穷多个实数，而浮点数与实数不同，两个浮点数之间是只包含有限个浮点数的。相邻的两个浮点数之间存在一个距离，这个距离被称为最小精度单位（Unit of Least Precision）或最后位置单位（Unit in the Last Place），简称**ULP**。

例如，Java Math 类中提供了nextUp()方法返回比第一个参数大的最近浮点数。如下java代码可以打印出所有在 1.0 和 2.0 之间的浮点数。

```
1  public class FloatCounter {
2
3      public static void main(String[] args) {
4          float x = 1.0F;
5          int numFloats = 0;
6          while (x <= 2.0) {
7              numFloats++;
8              System.out.println(x);
9              x = Math.nextUp(x);
10         }
11         System.out.println(numFloats);
12     }
13
14 }
```

https://www.ibm.com/developerworks/cn/java/j-math2.html

小结：为何定点数运算的器件多

- 使用同样的内存空间，浮点数却能比定点数表示大得多的范围。但是实际上支持定点数运算的器件却很多。我们为何不使用浮点数来代替定点数呢？
  - 浮点数运算电路实现复杂，有些处理器还专门配置了硬件浮点运算单元用于浮点运算
  - 精度问题：相同字长（比特数）的前提下，浮点数可以表示更大范围内的数，相邻浮点数之间的距离？浮点数无法取代定点数。
- 讨论：鱼和熊掌不可兼得，**浮点数表示了非常大的范围，但它失去了非常准的精度**。

2.2.5 浮点小数的计算电路

- 浮点小数的表示方法
  - 浮点数的格式
  - 浮点数的精度
- 浮点数的运算
  - 在定点处理器上如何实现浮点运算？
  - 如何设计浮点运算电路？

{wdwang, cxh}@ustc.edu.cn

由定点运算部件去完成浮点运算

◆计算机中的“数”有“定点数”和“浮点数”之分，“定点数”的运算部件的设计和实现比较容易，而“浮点数”的运算部件的设计和实现却复杂得多，并且困难得多。

◆因此，较早的计算机许多都不配备浮点运算，而是采用IBM的巴科斯(J.Backus，1977年度图灵奖获得者)发明的**软件**，**由定点运算部件去完成浮点运算**。可这种做法使浮点运算的速度大大降低，也就难以满足某些应用的需要了。

John Backus

John Warner Backus. He directed the team that invented the first widely used high-level programming language (**FORTRAN**) and was the inventor of the Backus-Naur form (**BNF**), the almost universally used notation to define formal language syntax. The IEEE awarded Backus the W.W. McDowell Award in 1967 for the development of FORTRAN. He received the National Medal of Science in 1975, and the 1977 ACM **Turing Award** "for **profound, influential, and lasting contributions** to the design of practical high-level programming systems, notably through his work on **FORTRAN**, and for publication of **formal procedures for the specification** of programming languages."

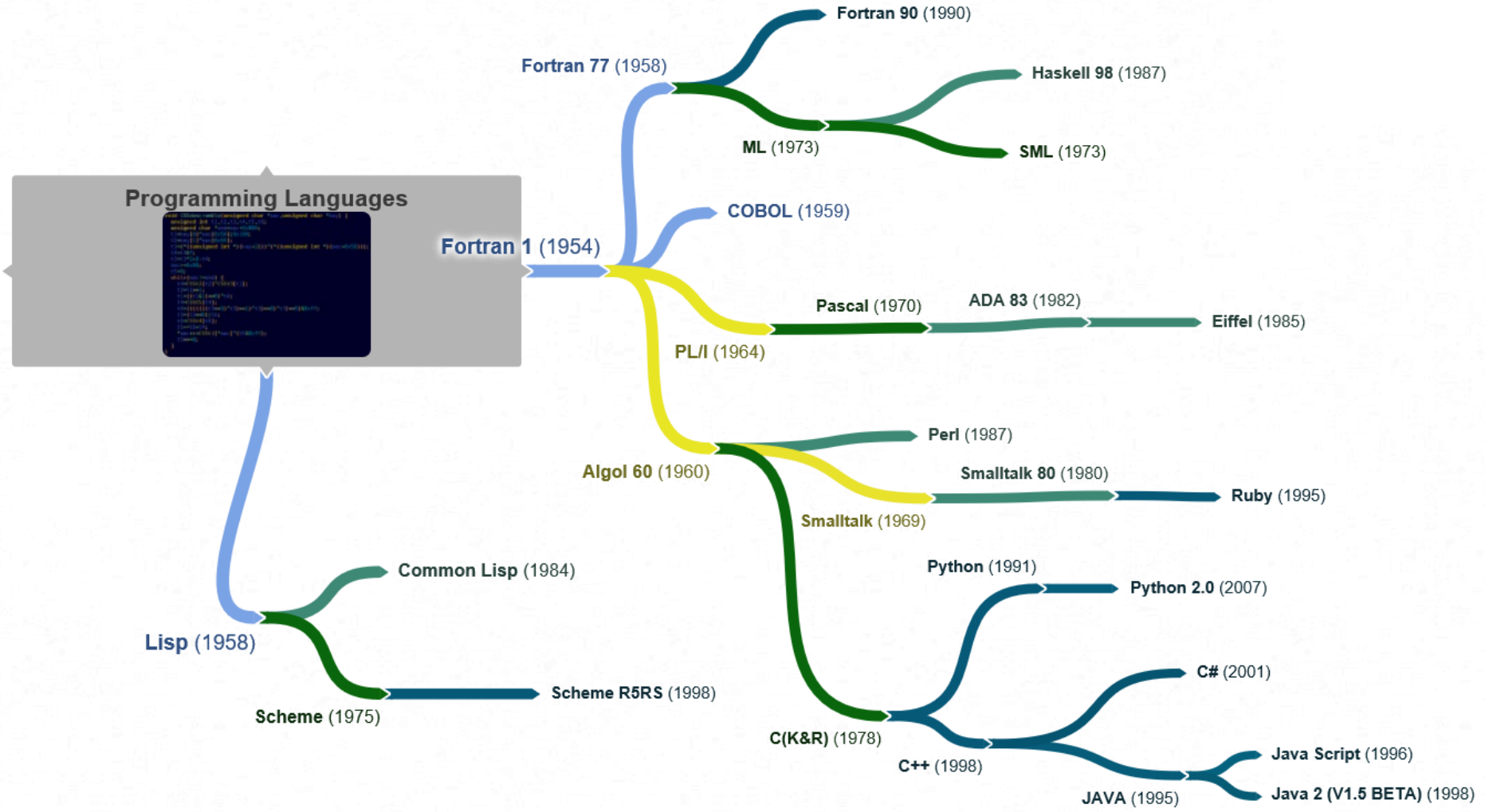
John Backus

Born	December 3, 1924 Philadelphia, Pennsylvania
Died	March 17, 2007 (aged 82) Ashland, Oregon
Fields	Computer Science
Institutions	IBM
Alma mater	Columbia University
Known for	Speedcoding FORTRAN ALGOL Backus-Naur form Function-level programming
Notable awards	ACM Turing Award Draper Prize

FORmula TRANslator

{wdwang, cxh}@ustc.edu.cn

一些常用编程语言



{wdwang, cxh}@ustc.edu.cn

浮点数加法示例

```
123456.7 = 1.234567 × 10^5
101.7654 = 1.017654 × 10^2 = 0.001017654 × 10^5
```

```
Hence:
123456.7 + 101.7654 = (1.234567 × 10^5) + (1.017654 × 10^2)
                    = (1.234567 × 10^5) + (0.001017654 × 10^5)
                    = (1.234567 + 0.001017654) × 10^5
                    =  1.235584654 × 10^5
```

In detail:

例中e为阶码，s为尾数（不是符号位）

阶码对齐

```
  e=5;   s=1.234567      (123456.7)
+ e=2;   s=1.017654      (101.7654)
```

```
  e=5;   s=1.234567
+ e=5;   s=0.001017654   (after shifting)
--------------------
  e=5;   s=1.235584654   (true sum: 123558.4654)
```

This is the true result, the exact sum of the operands.

```
  e=5;   s=1.235585     (final sum: 123558.5)
```

浮点数乘/除法的实现过程

乘法，尾数(significands)相乘，阶码(exponents)相加

```
  e=3;   s=4.734612
× e=5;   s=5.417242
-----------------------
  e=8;   s=25.648538980104 (true product)
  e=8;   s=25.64854        (after rounding)
  e=9;   s=2.564854        (after normalization)
```

除法，尾数(significands)相除，阶码(exponents)相减

小结：浮点数运算规则

按照运算规则编写软件即可在整数CPU上进行浮点数的运算

- 两个浮点数为 $x = 2^{E_x} \cdot M_x$、$y = 2^{E_y} \cdot M_y$，
- 两浮点数进行加减，（1）对阶：必须使它们的阶码相等。（2）求和或求差。（3）规格化。

$$x \pm y = \left(M_x 2^{E_x - E_y} \pm M_y\right) 2^{y}, E_x < E_y$$

- 两浮点数相乘，其乘积的阶码为相乘两数的阶码之和，其乘积的尾数为相乘两数尾数之积。

$$x \times y = 2^{E_x + E_y} \cdot (M_x \times M_y)$$

- 两浮点数相除，商的阶码为被除数的阶码减去除数的阶码所得到的差，尾数为被除数的尾数除以除数的尾数所得的商。

$$x \div y = 2^{E_x - E_y} \cdot (M_x \div M_y)$$

- 由定点运算部件去完成浮点运算
  - John Backus, Turing Award 1977

2.2.5 浮点小数的计算电路

- 浮点小数的表示方法
  - 浮点数的格式
  - 浮点数的精度
- 浮点数的运算
  - 在定点处理器上如何实现浮点运算？
  - 如何设计浮点运算电路？

{wdwang, cxh}@ustc.edu.cn

FP
加法电路

设2 个浮点操作数分别为A、B，

E_a、E_b 为指数操作数，
M_a、M_b 为尾数操作数，
S_a、S_b 为符号位，

E、M、S 分别是结果的指数、尾数和符号位。

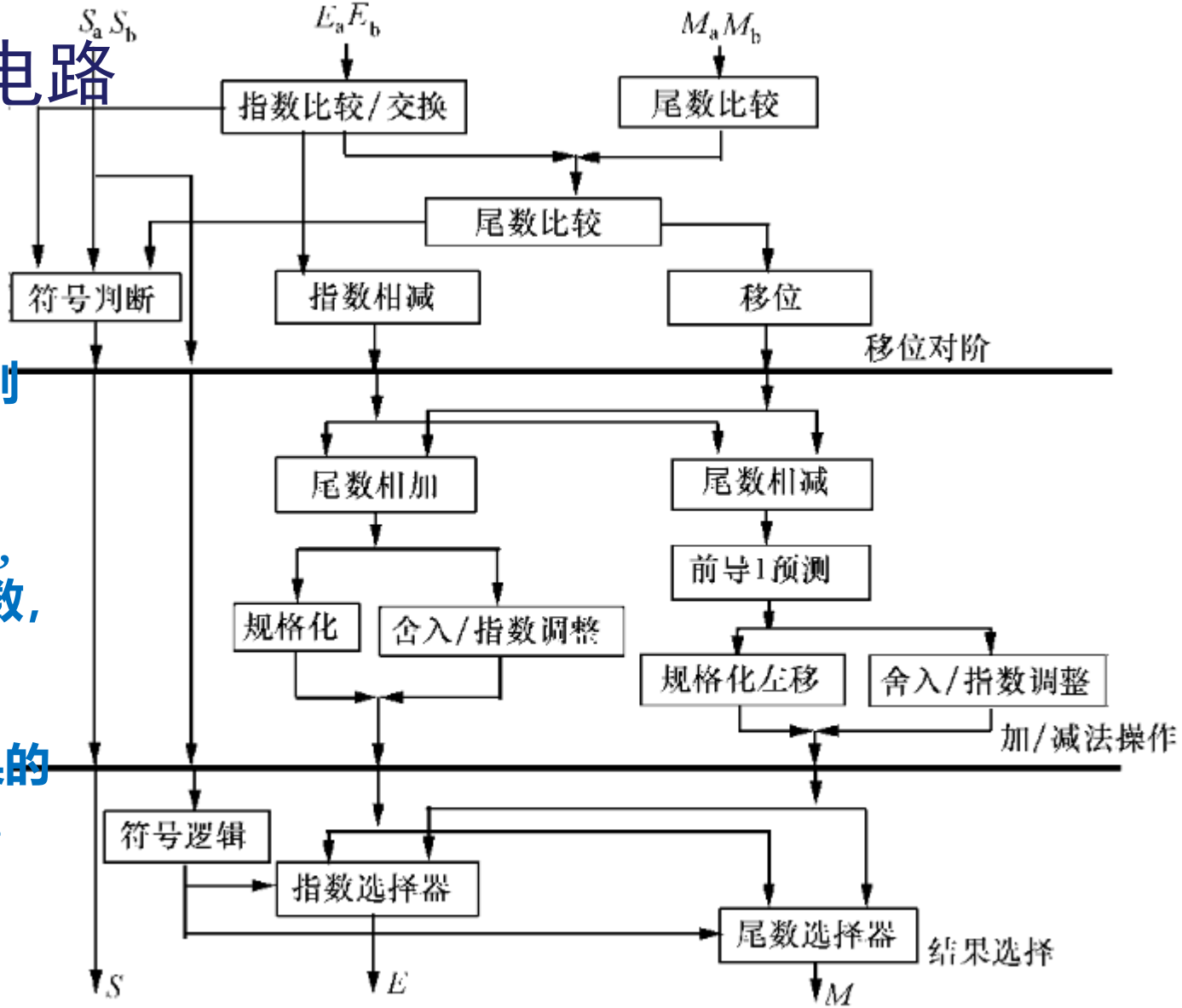


图 2 浮点加法运算单元结构

{wdwang, cxh}@ustc.edu.cn

FP乘法电路

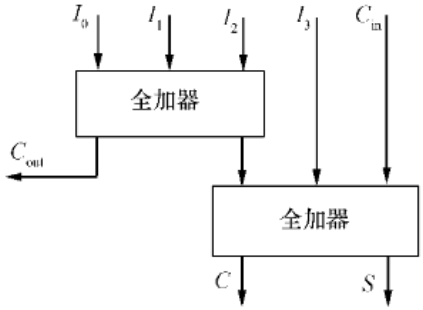


图 7 传统的 4-2压缩单元逻辑

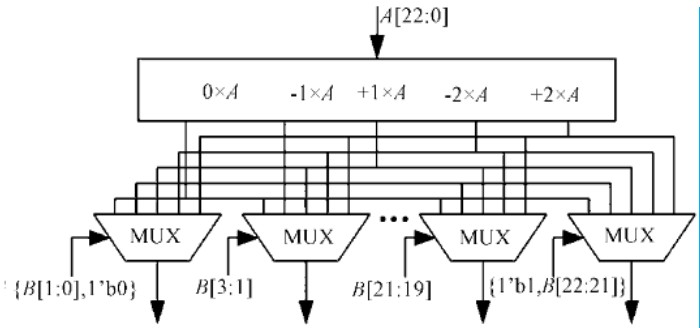


图 6 Booth编码器

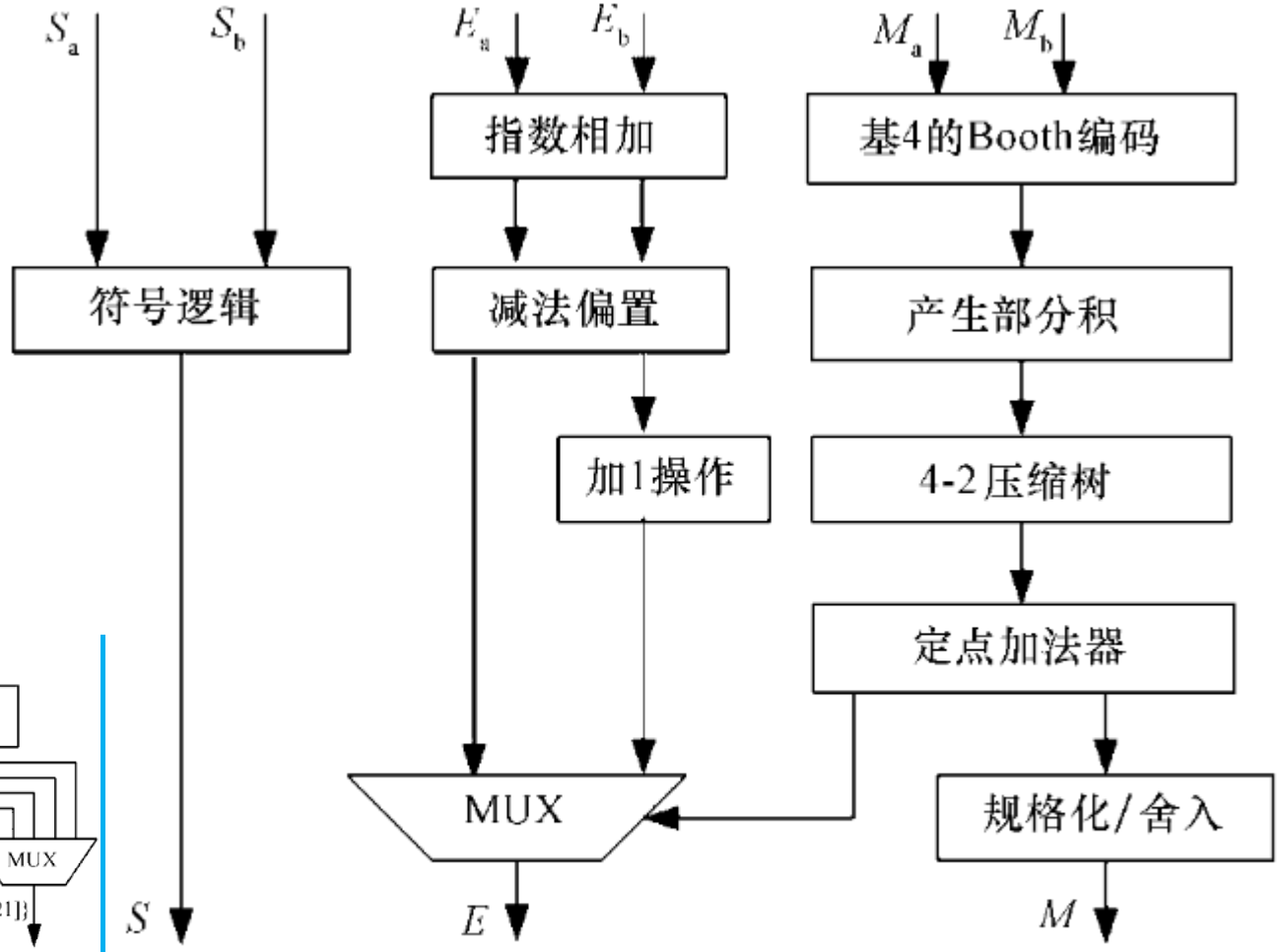


图 5 浮点乘法运算单元结构

高速浮点运算单元的FPGA实现，张小妍等,《信息化研究》,2009年11月

Altera的FPGA对浮点运算的支持

- 由于浮点算法动态范围较大，相对于浮点仿真，大大简化了系统性能验证任务，因此，对于设计人员而言，这种算法通常能够提高性能。在某些应用中，定点算法是不可行的。动态范围要求使用浮点算法的一个常见的例子是矩阵求逆运算
- Altera 提供单精度和双精度浮点 IP 内核，包括：
  - 加法 / 减法、乘法、除法、倒数
  - 指数、对数、平方根、逆平方根
  - 矩阵乘法、矩阵求逆
  - 快速傅立叶变换 (FFT)
  - 整数和分数转换

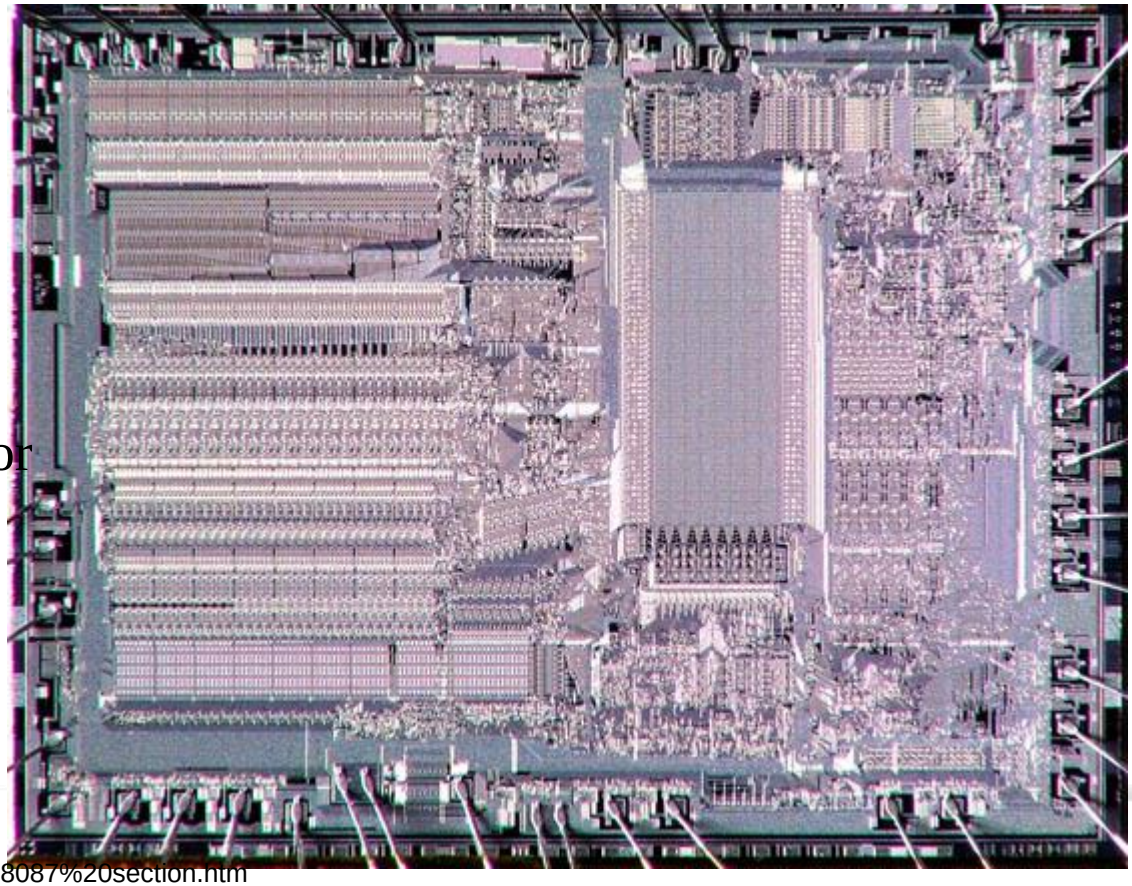
{wdwang, cxh}@ustc.edu.cn

Intel 8087

Intel的8087协处理器

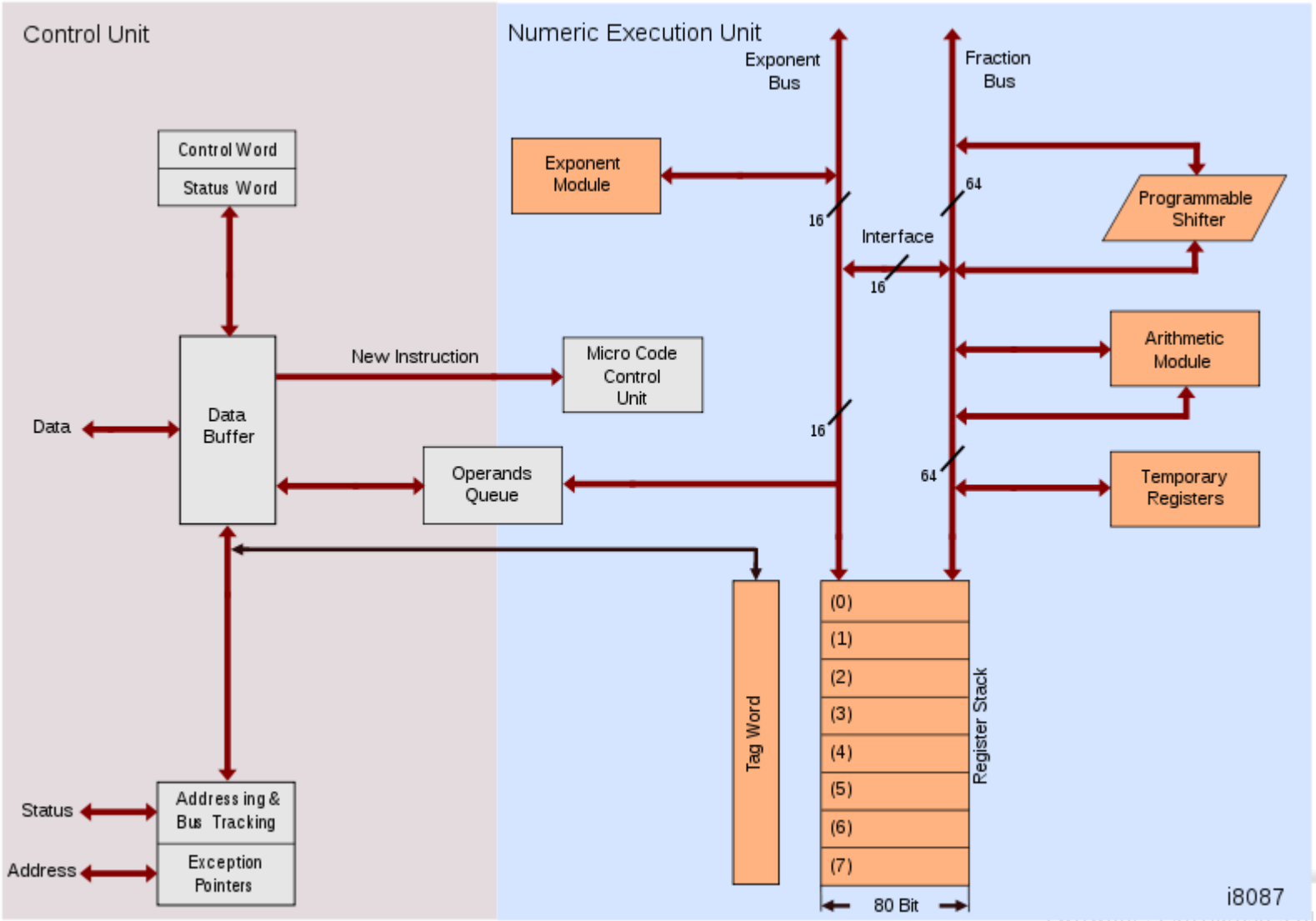
Introduction date: 1978
Frequency:
8087 5,0 MHz
8087-1 10,0 MHz
8087-2 8,0 MHz
8087-3 4,0 MHz
8087-6 6,0 MHz

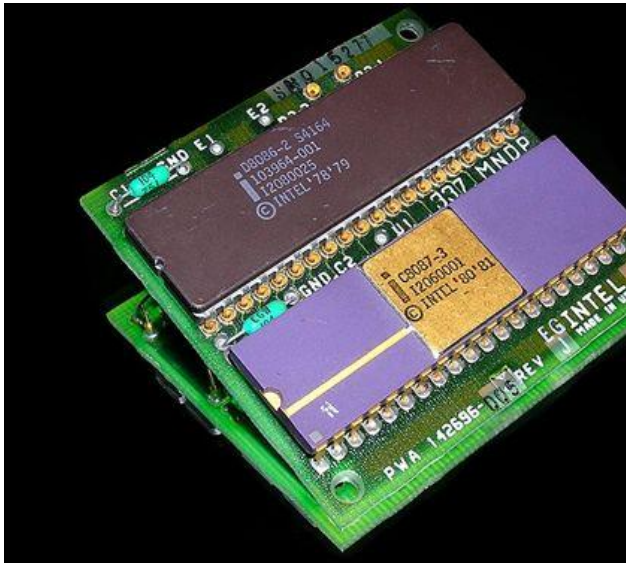
Technology: N Mos
Category: 16 Bit Math-Co Processor
Transistors: **45000**
Instructions: 68



http://www.cpu-galaxy.at/cpu/intel%20cpu/coprozessor/intel%208087%20section.htm

Simplified 8087 microarchitecture





1978年
8087
Transistors: **45000**
+
8086
Transistors: **约6000**

1990年
Intel 80486DX - 具有FPU的i486

Table 3. Execution Times for Selected 8086/8087 Numeric Instructions and Corresponding 8086 Emulation

Floating Point Instruction	Approximate Execution Time (μs)	
	8086/8087 (8 MHz Clock)	8086 Emulation
Add/Subtract	10.6	1000
Multiply (Single Precision)	11.9	1000
Multiply (Extended Precision)	16.9	1312
Divide	24.4	2000
Compare	−5.6	812
Load (Double Precision)	−6.3	1062
Store (Double Precision)	13.1	750
Square Root	22.5	12250
Tangent	56.3	8125
Exponentiation	62.5	10687

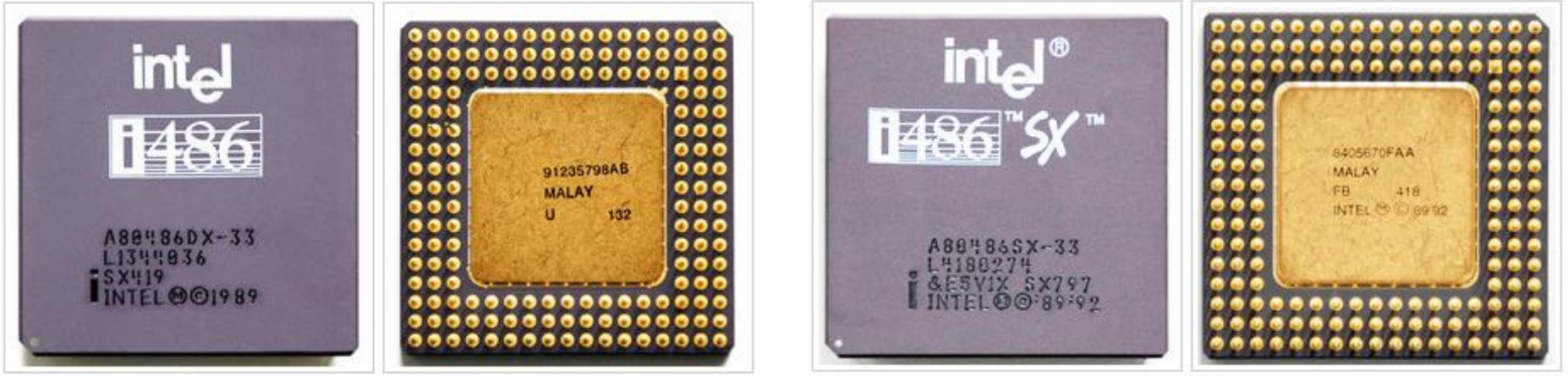
387协处理器

随后在1985年，Intel推出了80386处理器，与之配套推出了387协处理器。387协处理器与386微处理器并行连接，可以构成一个高效的386处理器系统。实际上387芯片相对于386的改变就是增加了8个80位的浮点寄存器，以及16位的控制寄存器、状态寄存器和标志寄存器。这样387协处理器就为386处理器扩充了七十多条指令和多种数据类型，使得386处理器的浮点运算也能够遵循IEEE-754浮点标准。

{wdwang, cxh}@ustc.edu.cn

486后， FPU（浮点处理单元）

1989年，Intel发布了486处理器。486处理器在性能上有很大进步，它首次在CISC体系X86架构中使用了类RICS内核，提高了指令执行速度，当然还有更高的主频和更好的制造工艺。486的处理器有两个不同的系列：有数学协处理器的486DX和无数学协处理器的486SX。**486最大的意义在于它整合了协处理器，协处理器的概念从此消失**而变为CPU内部的FPU（浮点处理器）。这是CPU在发展道路上的里程碑式的一次融合，这次融合显著提升了CPU的浮点处理能力，使得CPU更为全能和强大，是CPU在运算电路方面的一次突破



486时代的两个代表产品486DX(有FPU)和486SX(无FPU)

{wdwang, cxh}@ustc.edu.cn

FP & DP

ACM Announces 2021 A.M. Turing Award Recipient

March 30, 2022

- ACM has named Jack J. Dongarra recipient of the 2021 ACM A.M. Turing Award for pioneering contributions to numerical algorithms and libraries that enabled high performance computational software to keep pace with exponential hardware improvements for over four decades.

https://www.acm.org/articles/bulletins/2022/march/turing-award-2021

{wdwang, cxh}@ustc.edu.cn

FP & DP

- Dongarra创建的开源软件库中，包含很多技术创新
  - **自动调谐（Autotuning）：** Dongarra在2016年的全球超级计算大会上的ATLAS项目中，研究了一种新方法，用于自动找出能生成线性代数内核的算法参数，该线性代数内核在效率上是接近最优的。
  - **混合精度运算（Mixed Precision Arithmetic）：** 在Dongarra2006年递交给全球超级计算大会的论文中，他提出了要「利用32位浮点算法获得64位精度的性能」。他开创了一种办法，可以利用浮点计算的多倍精度来更快输出精确的解决方案。这项研究慢慢地在机器学习应用中越来越基础，最近的例证就是HPL-AL Benchmark，它在全球最先进的超算上实现了前所未有的性能。
  - **批量计算（Batch Computations）：** Dongarra开创了一种范式，用来分割开大密度矩阵的运算，在模拟、建模、数据分析等领域应用非常广泛。该范式可以将大密度矩阵的运算分成包含更小任务量的更多计算，可以各自独立地同时运算。
- 他参与创建的各种标准，包括MPI、LINPACK Benchmark等，为各类不同的计算任务奠定了基础，如天气预测、气候变化，再到分析大规模物理实验数据等等。
- 目前衡量超级计算机性能的最权威榜单Top 500，背后的评测软件算法就是Dongarra开发的。

{wdwang, cxh}@ustc.edu.cn

小结：浮点数运算电路

- 浮点数运算比定点数运算电路复杂度高很多
  - 阶码对齐
  - 指数的加减
  - 尾数的加减乘除
- 如，FPGA所提供的FP运算核

8087
Transistors: **45000**
8086
Transistors: **约6000**

- 外部8087浮点数协处理器
- 后期Intel把8087的功能集成到CPU内部
  - x87 Floating-point Instructions

第2章 数字计算电路

- 2.1 概述
- 2.2 基础计算电路
  - 2.2.1 加法电路
  - 2.2.2 从整数到定点小数
  - 2.2.3 乘法电路
  - 2.2.4 除法电路
  - 2.2.5 浮点数运算电路
  - 2.2.6 超越函数的计算
- 2.3 面向应用的计算电路
- 2.4 异构计算系统
- 2.5 高速计算电路的存储管理

{wdwang, cxh}@ustc.edu.cn

C语言标准库<math.h>

1 三角函数
double sin (double);
double cos (double);
double tan (double);

2 反三角函数
double asin (double); 结果介于[-PI/2, PI/2]
double acos (double); 结果介于[0, PI]
double atan (double); 反正切(主值), 结果介于[-PI/2, PI/2]
double atan2 (double, double); 反正切(整圆值), 结果介于[-PI/2, PI/2]

3 双曲三角函数
double sinh (double);
double cosh (double);
double tanh (double);

4 指数与对数
double exp (double);
double pow (double, double);
double sqrt (double);
double log (double); 以e为底的对数
double log10 (double);

5 取整
double ceil (double); 取上整
double floor (double); 取下整

6 绝对值
double fabs (double);

7 标准化浮点数
double frexp (double f, int *p); 标准化浮点数, f = x * 2^p, 已知f求x, p (x介于[0.5, 1])
double ldexp (double x, int p); 与frexp相反, 已知x, p求f

8 取整与取余
double modf (double, double*); 将参数的整数部分通过指针回传, 返回小数部分
double fmod (double, double); 返回两参数相除的余数

Python math库中超越函数示例

Power & Logarithmic Functions

Function	Description
pow(x,y)	Return- x to the power y value
sqrt(x)	Finds the square root of x
exp(x)	Finds xe, where e = 2.718281
log(x[,base])	Returns the log of x where base is given. The default base is e
log2(x)	Returns the log of x, where base is 2
log10(x)	Returns the log of x, where base is 10

Trigonometric and Angular Conversion Functions

Function	Description
sin(x)	Return the sine of x in radians
cos(x)	It returns the cosine of x in radians
tan(x)	It returns the tangent of x in radians
asin(x)	It returns the inverse of the sine, similarly we have acos, atan also
degrees(x)	It conver angle x from radian to degrees
radians(x)	It convert angle x from degrees to radian

{wdwang, cxh}@ustc.edu.cn

超越函数的计算

- 查表法
- 级数近似

$$f(x)=\sin x=x-\frac{x^3}{3!}+\frac{x^5}{5!}+\ldots+(-1)^{n-1}\frac{x^{2n-1}}{(2n-1)!}+\ldots$$

$$\cos(x)=1-\frac{x^2}{2!}+\frac{x^4}{4!}-\frac{x^6}{6!}+\frac{x^8}{8!}-\ldots$$

←n的取值和需要的精度有关系

x=pi/4 = 0.78539816339744830961566084581988
-x^3/3! = -0.08074551218828078170696957048724
x^5/5! = 0.00249039457019272016001579842157
-x^7/7! =- 0.00003657620418217725078660518698
sin45=sin(pi/4)= 0.707106469575178070817920468567

$$e^x=\sum_{k=1}\frac{x^k}{k!}$$

- CORDIC ←最广泛
  - Many older systems with **integer-only CPUs** have implemented CORDIC to varying extents as part of their IEEE Floating Point libraries. As most modern general-purpose CPUs have floating-point registers with common operations such as add, subtract, multiply, divide, sin, cos, square root, log10, natural log, the need to implement CORDIC in them with software is nearly non-existent. Only microcontroller or special safety and time-constrained software applications would need to consider using CORDIC.

CORDIC algorithm

◆CORDIC (for COordinate Rotation DIgital Computer), also known as Volder's algorithm, or: Digit-by-digit method Circular CORDIC (Jack E. Volder),Linear CORDIC, Hyperbolic CORDIC (John Stephen Walther), and Generalized Hyperbolic CORDIC (GH CORDIC) (Yuanyong Luo et al.), is a simple and efficient algorithm to calculate trigonometric functions, hyperbolic functions, square roots, multiplications, divisions, and exponentials and logarithms with arbitrary base, typically converging with one digit (or bit) per iteration.

◆坐标旋转数字计算机CORDIC算法，通过**移位**和**加减**运算，能递归计算常用函数值，如Sin，Cos，Sinh，Cosh等函数，由J. Volder于1959年公开提出，首先用于导航系统，使得矢量的旋转和定向运算不需要做查三角函数表、乘法、开方及反三角函数等复杂运算。J. Walther在1974年用它研究了一种能计算出多种超越函数的统一算法。

CORDIC
(COordinate Rotation DIgital Computer)

- In early 1956 the aeroelectronics department of Convair, Fort Worth, was given the task of determining the feasibility of **replacing** the **analog** computer-driven navigation system of the B-58 bomber with a **digital** computer.
- Most navigation system specialists agreed that the existing B-58 navigation computer was an ingenuous device utilizing analog resolvers to compute, in real time, the complex trigonometric relationships necessary for navigation over a spherical earth. Each resolver was capable of performing either a rotation of input coordinates or inversely determining the magnitude and angle of the vector defined by the input coordinates—also called vectoring.

The Birth of CORDIC, JACK E. VOLDER,
Journal of VLSI Signal Processing 25, 101–105, 2000

CORDIC计算sin cos示意

单位向量每次旋转γ_i角度，将sin计算转换为arctanγ_i的计算

$$v_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad v_i = R_i v_{i-1} \qquad R_i = \begin{bmatrix} \cos\gamma_i & -\sin\gamma_i \\ \sin\gamma_i & \cos\gamma_i \end{bmatrix}$$

$$\cos\alpha = \frac{1}{\sqrt{1+\tan^2\alpha}} \qquad \sin\alpha = \frac{\tan\alpha}{\sqrt{1+\tan^2\alpha}}$$

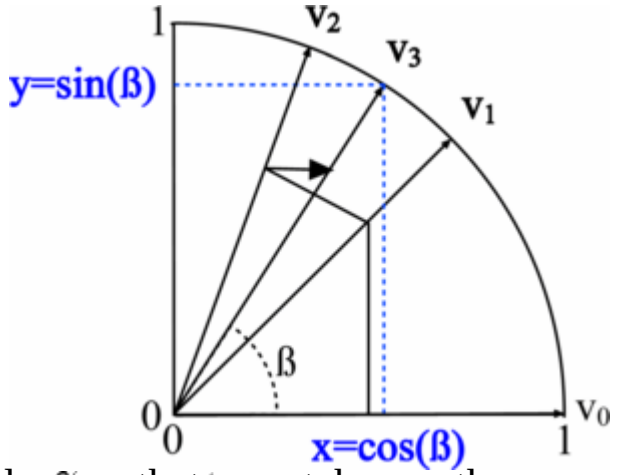
$$R_i = \frac{1}{\sqrt{1+\tan^2\gamma_i}} \begin{bmatrix} 1 & -\tan\gamma_i \\ \tan\gamma_i & 1 \end{bmatrix}$$

$$v_i = \frac{1}{\sqrt{1+\tan^2\gamma_i}} \begin{bmatrix} 1 & -\tan\gamma_i \\ \tan\gamma_i & 1 \end{bmatrix} \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix}$$

$$v_i = K_i \begin{bmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix} \qquad K_i = \frac{1}{\sqrt{1+2^{-2i}}}$$

$$\beta_i = \beta_{i-1} - \sigma_i \gamma_i. \qquad \gamma_i = \arctan 2^{-i},$$

预先计算好

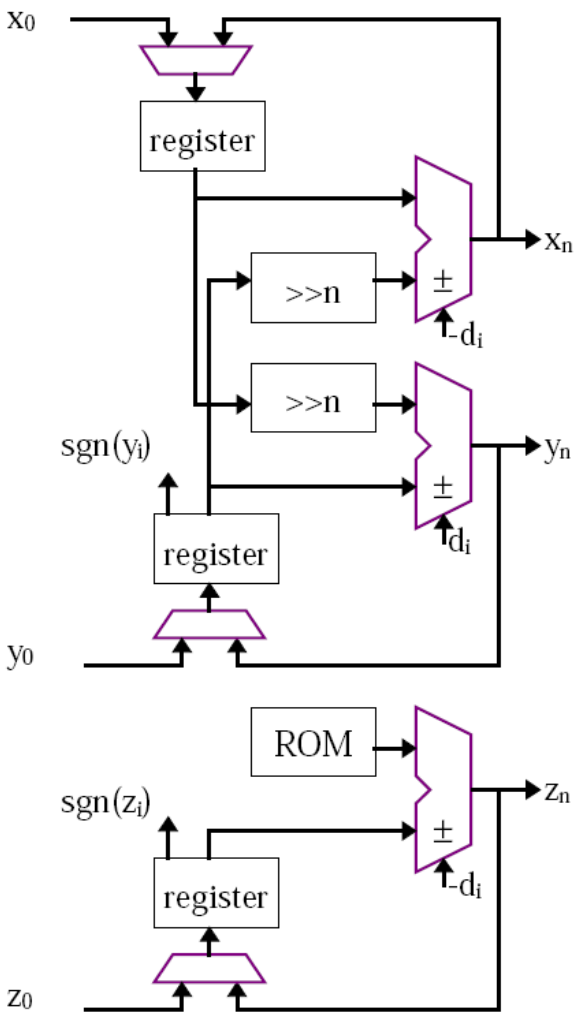


Restricting the angles γ_i so that $\tan\gamma_i$ takes on the values $\pm 2^{-i}$ the multiplication with the tangent can be replaced by a division by a power of two, which is done in digital computer hardware using a bit shift.

σ_i can have the values of -1 or 1 and is used to determine the direction of the rotation

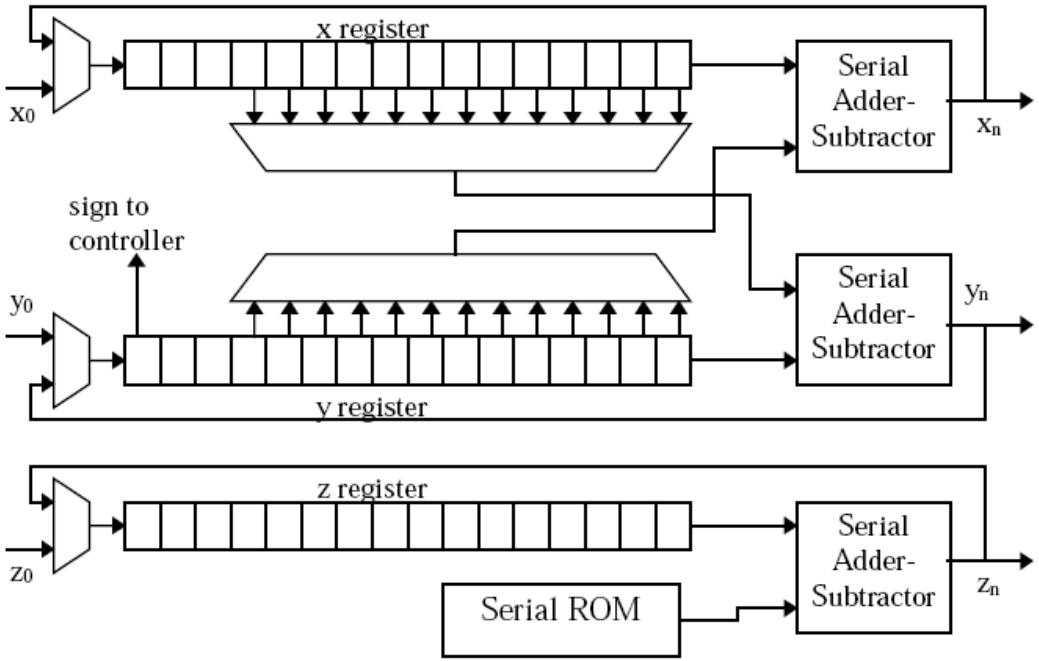
CORDIC can be used to calculate a number of different functions. This explanation shows how to use CORDIC in *rotation mode* to calculate sine and cosine of an angle, and assumes the desired angle is given in radians and represented in a fixed point format. To determine the sine or cosine for an angle , the *y* or *x* coordinate of a point on the unit circle corresponding to the desired angle must be found.

CORDIC算法的实现电路



比特并行(bit parallel)

ROM中存放事先计算好的$\arctan(2^{-i})$表，利用查表法获取

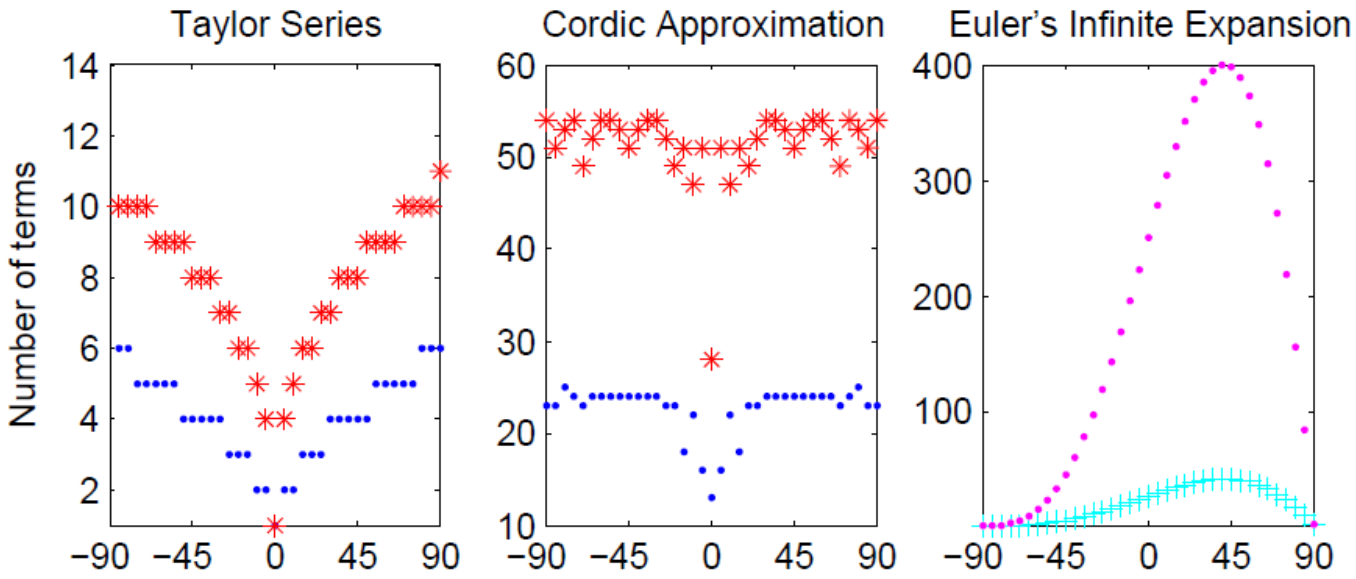


比特串行(bit serial)结构

{wdwang, cxh}@ustc.edu.cn

cos(x)三种方法计算项的对比

The Taylor series requires more terms as x moves away from 0, while the Cordic expansion is not dependent on the value of x. Euler's infinite expansion is far worse.



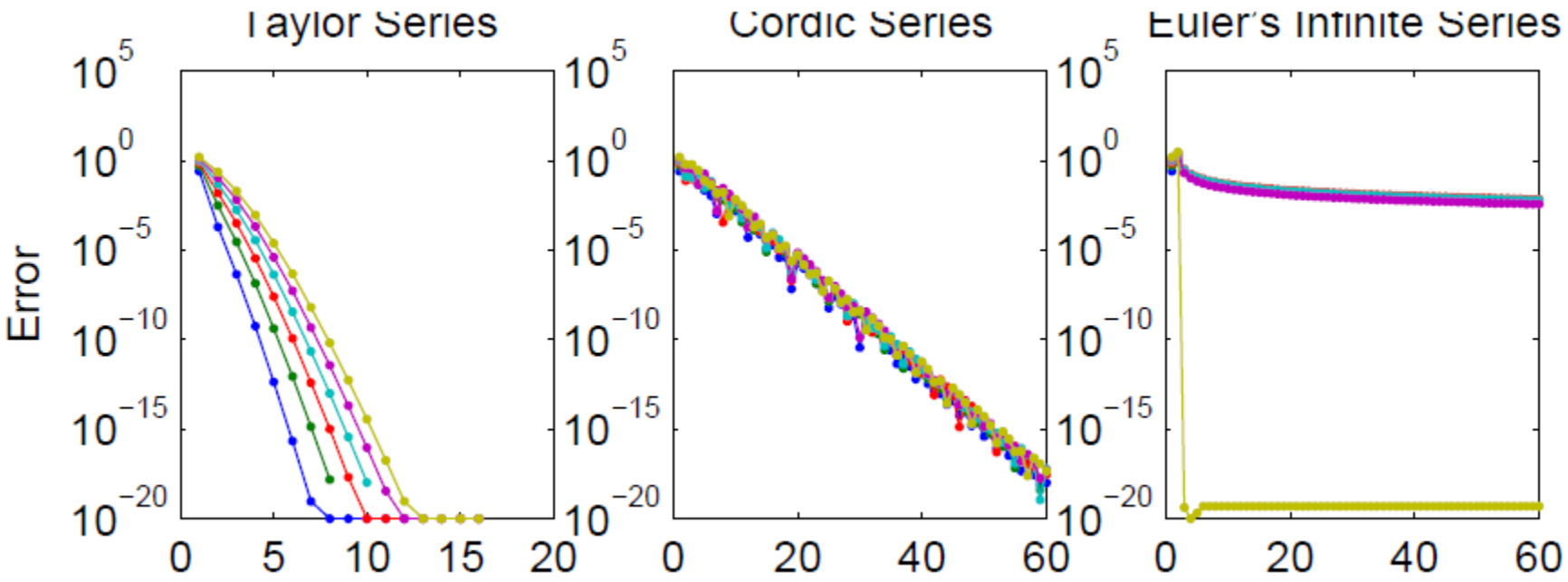
$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \dots$$

蓝色：达到单精度
红色：达到双精度

$$\sin x = \prod_{n-1}^{\infty} \frac{(\pi n - x)\cdot(\pi n + x)}{\pi n \cdot \pi n} = x \cdot \prod_{n-1}^{\infty} \frac{1 - x^2}{\pi^2 n^2}$$

{wdwang, cxh}@ustc.edu.cn

cos(x)三种方法收敛速度



Different values of x are used to demonstrate the dependence of convergence on the input value. The values chosen here are 0, 15, 30, 45, 60, 75, and 90 degrees. The Taylor series converges rapidly, yielding multiple digits per term. The Cordic series yields one bit per term. Euler's infinite series converges surprisingly slowly.

Intel i7中超越函数的运算速度？
x87 Floating-point Instructions

The *Intel® 64 and IA-32 Architectures Optimization Reference Manual* describes how to optimize software to take advantage of the performance characteristics of IA-32 and Intel 64 architecture processors. Optimizations described in this manual apply to processors based on the Intel® Core™ microarchitecture, Enhanced Intel® Core™ microarchitecture, Intel microarchitecture (Nehalem), Intel NetBurst® microarchitecture, the Intel® Core™ Duo, Intel® Core™ Solo, Pentium® M processor families.

FADD	**1μops**
FSUB	3μops
FMUL	5μops
FDIV	32μops
FSQRT	58μops
FCOS	**119μops**
FSIN	119μops
FSINCOS	119μops
FPATAN	147μops
FPTAN	123μops
FYL2X	96μops
FYL2XP1	98μops

Intel i7中超越函数的运算速度？实测结果

sinf → 22 cycles/value on a 1000MHz computer

Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz slc6 glibc 2.12-1.7 gcc 4.6.0 SSE2 (same executible as above)

```
benching              sinf .. ->   22.8 millions of vector evaluations/second ->  22 cycles/value on a 2000MHz computer
benching              cosf .. ->   20.7 millions of vector evaluations/second ->  24 cycles/value on a 2000MHz computer
benching        sincos (x87) .. ->    7.6 millions of vector evaluations/second ->  66 cycles/value on a 2000MHz computer
benching              expf .. ->    1.3 millions of vector evaluations/second -> 385 cycles/value on a 2000MHz computer
benching              logf .. ->   16.6 millions of vector evaluations/second ->  30 cycles/value on a 2000MHz computer
benching             log16 .. ->   49.2 millions of vector evaluations/second ->  10 cycles/value on a 2000MHz computer
benching            atan2f .. ->   17.4 millions of vector evaluations/second ->  29 cycles/value on a 2000MHz computer
benching             atan2 .. ->    7.8 millions of vector evaluations/second ->  64 cycles/value on a 2000MHz computer
benching              sinl .. ->   10.6 millions of vector evaluations/second ->  47 cycles/value on a 2000MHz computer
benching              cosl .. ->   10.9 millions of vector evaluations/second ->  46 cycles/value on a 2000MHz computer
benching              expl .. ->   11.4 millions of vector evaluations/second ->  44 cycles/value on a 2000MHz computer
benching              logl .. ->    7.8 millions of vector evaluations/second ->  64 cycles/value on a 2000MHz computer
benching       cephes_sinf .. ->   29.3 millions of vector evaluations/second ->  17 cycles/value on a 2000MHz computer
benching       cephes_cosf .. ->   26.7 millions of vector evaluations/second ->  19 cycles/value on a 2000MHz computer
benching       cephes_expf .. ->    9.9 millions of vector evaluations/second ->  51 cycles/value on a 2000MHz computer
benching       cephes_logf .. ->   18.3 millions of vector evaluations/second ->  27 cycles/value on a 2000MHz computer
benching            sin_ps .. ->   46.9 millions of vector evaluations/second ->  11 cycles/value on a 2000MHz computer
benching            cos_ps .. ->   46.8 millions of vector evaluations/second ->  11 cycles/value on a 2000MHz computer
benching          sincos_ps .. ->   42.2 millions of vector evaluations/second ->  12 cycles/value on a 2000MHz computer
benching            exp_ps .. ->   36.4 millions of vector evaluations/second ->  14 cycles/value on a 2000MHz computer
benching            log_ps .. ->   34.4 millions of vector evaluations/second ->  15 cycles/value on a 2000MHz computer
```

Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz slc6 glibc 2.12-1.7 gcc 4.6.0 SSE2

小结：超越函数的计算

- 超越函数(Transcendental Functions)
  - 不能用有限次加、减、乘、除、乘方、开方 运算表示
- 查表法
- 级数近似：泰勒级数、欧拉级数
- CORDIC(COordinate Rotation DIgital Computer)
  - 通过移位和加减运算，能递归计算常用函数值
  - J. Walther在1974年用它研究了一种能计算出多种超越函数的统一算法

{wdwang, cxh}@ustc.edu.cn

总结：2.2 基础计算电路

- 四则运算的电路
  - 正整数和负整数用相同电路
  - 整数和定点小数运算用相同电路
  - 乘法原理：部分和累加→串行移位乘法器
  - 快速乘法器速度与芯片面积的平衡
- 如何进行浮点数的运算？
  - 在定点处理器上如何实现浮点运算？软件
  - 如何设计浮点运算电路？
    - 阶码对齐、指数的加减、尾数的加减乘除
- 超越函数如何计算？
  - 查表、级数近似、**CORDIC**(COordinate Rotation DIgital Computer)

{wdwang, cxh}@ustc.edu.cn

电子系统综合设计

Integrated Electronic System Design

第2章 数字计算电路

谢谢！