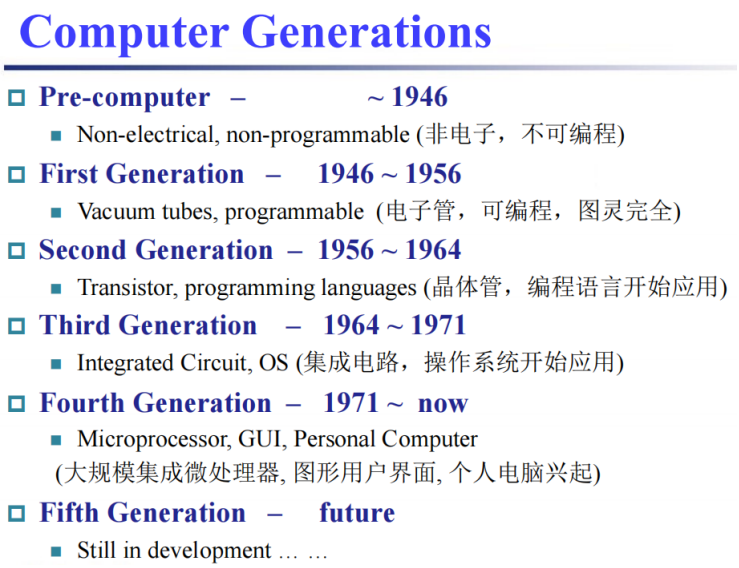
计组知识点整理

Chapter1 - 计算机概要与技术

发展历史



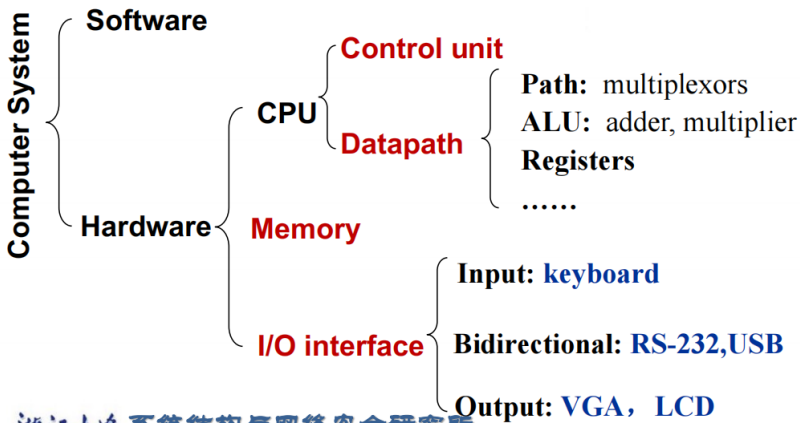
- **图灵完备** - 可以描述可计算的东西
- **冯诺依曼架构特点** - 计算与存储分离，但是数据与指令保存在同一个存储器

计算机定义

1. 电子化实现方式
2. 有指令集
3. 可执行指令
4. 可存储指令与数据
5. 计算能力上图灵完备

计算机组成

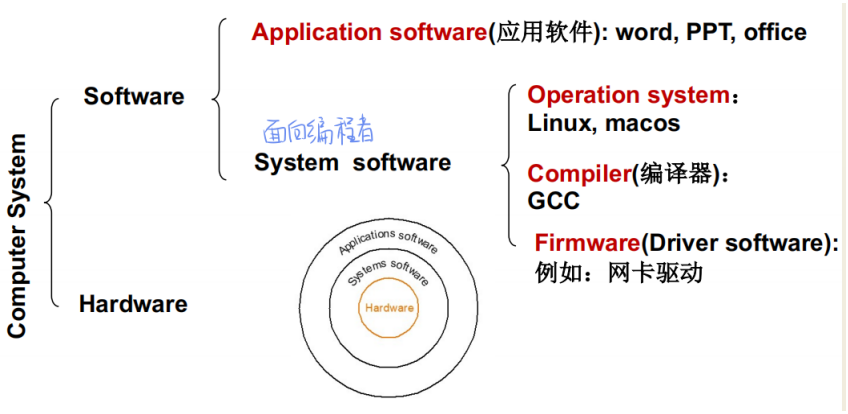
Hardware:



- 主板由一块块芯片组成，分别为CPU、闪存、power controller（启动、关机、待机）和I/O controller
- **Memory:**

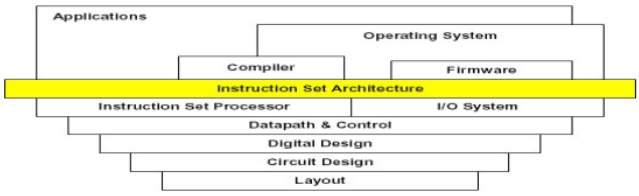
- Main Memory（主存）- Volatile（易失性）
  - DRAM - 动态随机存储器
  - SRAM - 静态随机存储器
- Second Memory - Nonvolatile（非易失性）
  - Flash Memory - 固态硬盘 or 内存
  - Hard Disk - 硬盘

Software:

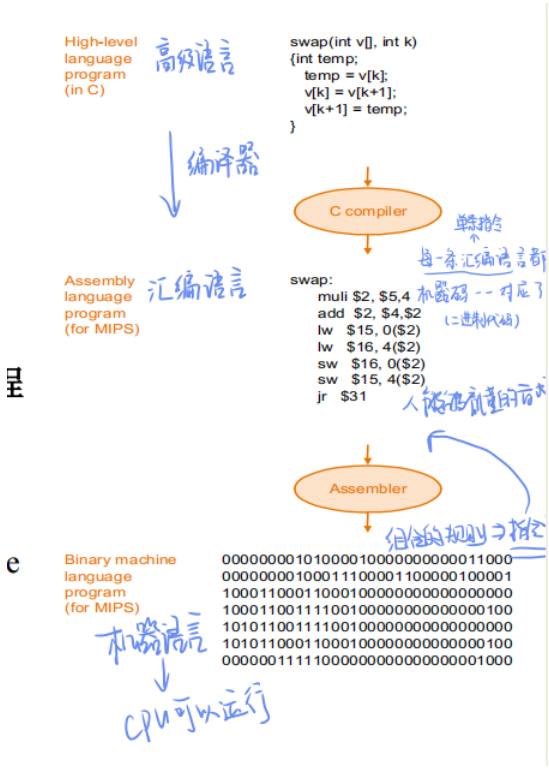


- **OS:**
  - 处理基本IO
  - 分配存储
  - 多进程处理
- **Compiler - 编译器**：把高级语言转换为机器语言
- **Firmware - 驱动**：为某个硬件设计的"software"，使得不同硬件在OS调用的软件界面是一致的

Instruction Set Architecture（指令集） - 是软硬件的桥梁，把硬件向软件隐藏的操作



编程语言层级



处理器生产 - 芯片制造

芯片设计 → 晶圆生产 → 芯片封装 → 芯片测试

$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{Yield}}$$

计算cost：

$$\text{Dies per wafer} \approx \text{Wafer area}/\text{Die area}$$

$$\text{Yield} = \frac{1}{(1+(\text{Defects per area} \times \text{Die area}/2))^2}$$

现状：

- 集成度复杂度高
- 工艺达到物理极限，制造难度大
- 多技术集成
- 挑战 - “内存墙”、“功耗墙”

计算机设计

性能评估（!! 重点 !!）

1. 相关定义：

response time（响应时间）/**execution time**（执行时间）- 完成一项任务要多久

throughput (bandwidth)（吞吐率） - 单位时间内可以完成的工作量

$Performance = 1/\text{Execution time}$

$\rightarrow Performance_X/Performance_Y = \text{Execution time}_Y/\text{Execution time}_X = n$

$\rightarrow$ "X is n time faster than Y"

elapsed time - 总的响应时间，包括处理、IO、编译等所有过程，决定了系统的性能

CPU time - 处理一个给定工作的时间（不包括IO等），包括user CPU time和system CPU time

clock period - 一个时钟周期的时长

clock rate - 一秒几个时钟周期 $\rightarrow = 1/\text{clock period}$

2. CPU time和CPI的计算：

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time} = \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- 指令的数量由：程序内容、ISA（指令集）、编译器决定

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}} \right)$$

- 平均CPI计算：根据每条指令的数量在总数的占比进行CPI的加权平均

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- CPU Time = 一个程序的指令数量 * CPI * 一个时钟周期时长

3. 功耗计算

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

处理"功耗墙"问题的趋势：单核并行→多核

4. SPEC CPU基准测试

1. 运行测试集: 每个测试（或称为基准）在测试机器上运行，产生执行时间。
2. 计算比率: 对于每个基准，都将测试机器的执行时间与在参考机器上运行同一基准的执行时间进行比较，计算出一个性能比率。这个比率通常使用以下公式得出：

   $$\text{基准比率} = \frac{\text{参考机器的执行时间}}{\text{测试机器的执行时间}}$$

   如果测试机器运行得更快，执行时间会更短，从而导致比率更高。
3. 几何平均: 所有基准比率的几何平均值被计算出来，以得到最终的SPEC得分。几何平均值是通过乘以所有比率，然后对该乘积取n次方根（其中n是基准的数量）来计算的。几何平均值的公式是：

   $$\text{SPEC得分} = \left(\prod_{i=1}^{n} \text{基准比率}_i\right)^{\frac{1}{n}}$$

   这里的乘积符号（∏）表示将所有的基准比率相乘。

5. SPEC Power基准测试

不同负载下每个跑多少指标 / 功耗

$$\text{Overall ssj\_ops per Watt} = \left(\sum_{i=0}^{10} \text{ssj\_ops}_i\right) \Big/ \left(\sum_{i=0}^{10} \text{power}_i\right)$$

⇒ 单位功耗跑多少

6. 优化限制

能优化的部分

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

能优化的倍

7. MIPS - 每秒能跑多少百万条指令（不完全和性能成正比，受指令集和指令复杂度影响）

$$\begin{aligned} \text{MIPS} &= \frac{\text{Instruction count}}{\text{Execution time} \times 10^6} \\ &= \frac{\text{Instruction count}}{\frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6} \end{aligned}$$

8. 相关点评

- 软硬件都是逐级抽象的
- 最好的性能评估标准是**执行时间**
- **Power** is a limiting factor - 通过**并行**优化

八大设计思想

1. Design for Moore's Law - 设计紧跟**摩尔定律**

- rapid change in computer design → design for where it will be
- *e.g. 增加了电磁飞机弹射器(这是电力驱动的，而不是目前的蒸汽驱动的模型)，允许由新的反应堆技术提供的电力增加*
- 强调做（for where it will be）这件事情本身

2. Use Abstraction to Simplify Design - 采用**抽象简化**设计

- 低层级的具体实现细节对高层级隐藏
- *e.g. 制造自动驾驶汽车，其控制系统部分依赖于已经安装在基础车辆中的现有传感器系统，如车道偏离系统和智能巡航控制系统，开发者不需要考虑所有模块的细节*

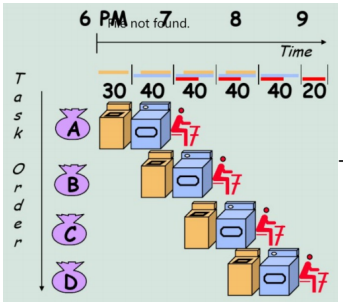
3. Make the **Common Case Fast** - 加速大概率事件

- *e.g. 高楼内快速电梯*

4. Performance via Parallelism - 通过**并行**提高性能

- *e.g. 增加CMOS晶体管的栅极面积（降低电阻）以减少其开关时间*

5. Performance via Pipelining - 通过**流水线**提高性能

- 

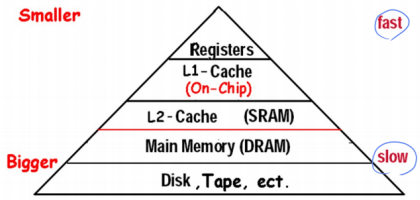
上一条指令未结束时下一条指令开始做
- *e.g. 汽车制造中的装配线(assembly lines)*

6. Performance via Prediction - 通过**预测**提高性能

- 在知道精确的选择结果前先预测并执行下一步（例如if else语句，射击预判）
- *e.g. 包含风力信息的飞机和航海导航系统*
- 强调预测对行为的指导，减少等待时间

7. Hierarchy of Memories - 存储器**层次**

- 存储器分层设计：快、最小和最昂贵的内存位于层次结构的顶部，最慢、最大和最便宜的内存位于底层



- *e.g. 图书馆查书台 - 把经常要用的书放最上面*

8. Dependability via Redundancy - 通过**冗余**提高**可靠**性

- 例如使用备用处理器，若探测到不一样的结果则报ERROR
- *e.g. 悬索桥缆索*

Chapter3 - 计算机算术运算

Pre-introduction

- 课本均以64位为例，但在实际学习过程中大多数没有说明的情况默认为**32位**
- 我们一般使用2's complement表示有符号整数
- 判断是否溢出：两个数字最高位和结果最高位再前面一位（C_i）异或，为0则无溢出

算术计算

减法→使用2's complement变减为加

有关有符号数溢出：同号相加/异号相减情况下结果最高两位异或为`1`即发生溢出

具体过程可参考数逻相关笔记：

二进制计算

补码计算：从低位到高位扫描

- 复制所有最低有效的0
- 复制第一个出现的1
- 接下来所有位取反

对有符号数，反码和补码的计算是保留符号位（最高位）的，也就是说最高位始终为1（表示负权重）。

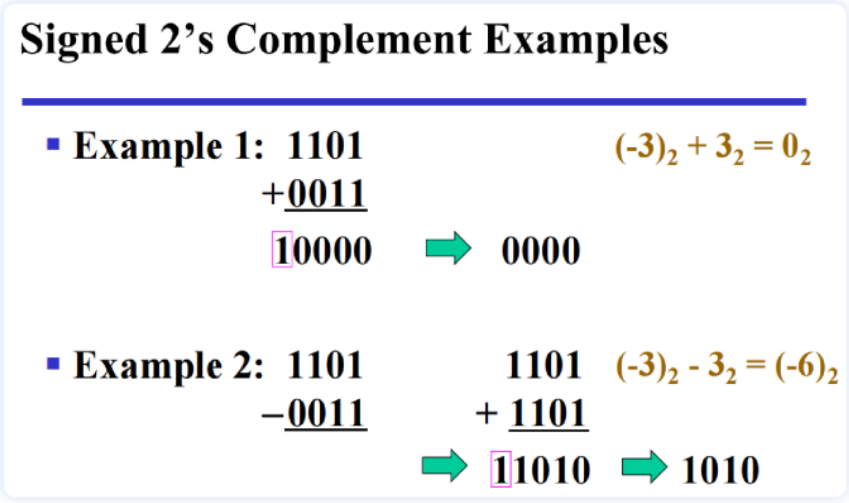
Number	Sign - Mag.	1's Comp.	2's Comp.
+3	011	011	011
+2	010	010	010
+1	001	001	001
+0	000	000	000
–0	100	111	—
–1	101	110	111
–2	110	101	110
–3	111	100	101
–4	—	—	100

Signed-Complement

符号拓展：将符号位复制到更高位。

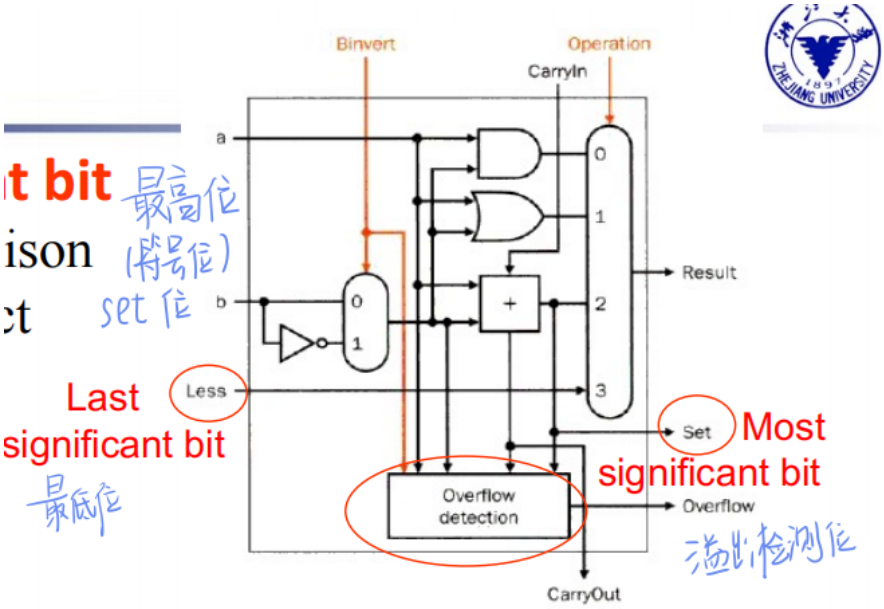
无符号二进制减法：先加上被减数的补码，如果得到的进位是1，则不用修改；否则取结果的补码并加上负号作为最终的答案。

带符号的二进制数运算：先将所有数变成带符号的二进制补码进行运算，再舍弃溢出位、转换回来。运算时减法取减数的无符号二进制补码（我的理解是化减为加）进行计算。



ALU

Extended 1-bit ALU

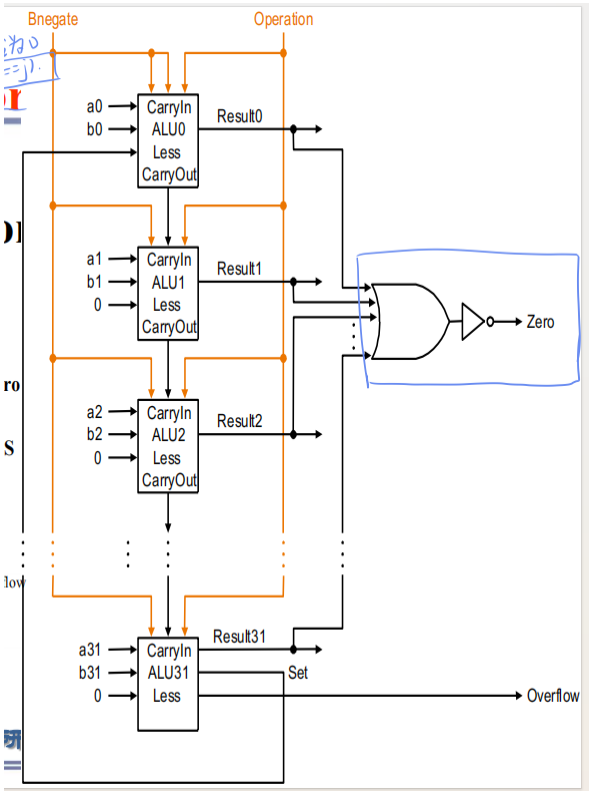


- Operation = 0⟶AND
- Operation = 1⟶OR
- Operation = 2⟶Add/Subtract - 取决于Binvert信号，为`0`时为加，为`1`时对`b`取反加1(cin = 1)为减法
- Operation = 3⟶slt - 比较操作，取`rs-rt`符号位，提前有一个组合逻辑单元专门计算slt，那个逻辑单元的输出是`00001`或者`00000`，取最低位为判断位，其他位均为填充
- 溢出检测：

Operation	Operand A	Operand B	Result overflow	
A+B	≧0	≧0	<0	*(01)*
A+B	<0	<0	≧0	*(10)*
A-B	≧0	<0	<0	*(01)*
A-B	<0	≧0	≧0	*(10)*

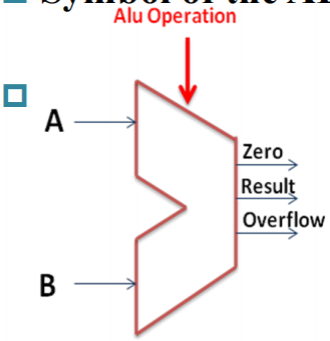
Complete ALU with Zero detection

把1-bit的ALU拓展为32位，实现与或加减比较的基本功能，同时可以附带判断当前结果是否为`0`的功能，make common case like `if(i == j)` fast：



实际CPU中的ALU symbol & control

Symbol of the ALU



Control: Function table

ALU Control Lines	Function
000	And
001	Or
010	Add
110	Sub
111	Set on less than
100	nor
101	srl
011	xor

Fast Adders

1. CLA - Carry Lookahead Adder

$$P_i = A_i \oplus B_i \qquad G_i = A_i B_i$$

$$S_i = P_i \oplus C_i \qquad C_{i+1} = G_i + P_i C_i$$

具体实现参考数逻相关笔记:

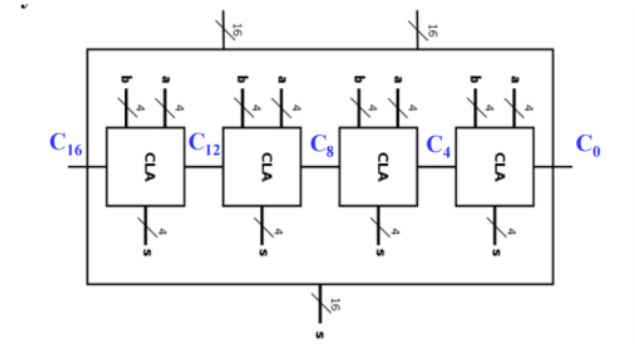
- 使用GP的划分，因为GP的值都只与当下的两个XY输入相关，因此我们对进位传递的优化只需要考虑上一个进位的提前计算，也就是迭代解决这个问题：

$$C_1 = G_0 + P_0 C_0$$

dependency on
with substitutic

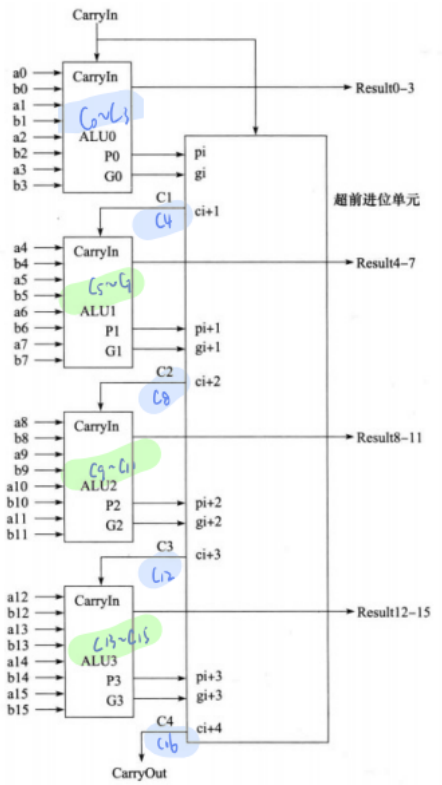
$$C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0)$$
$$= G_1 + P_1G_0 + P_1P_0 C_0$$
$$C_3 = G_2 + P_2 C_2 = G_2 + P_2(G_1 + P_1G_0 + P_1P_0 C_0)$$
$$= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0 C_0$$
$$C_4 = G_3 + P_3 C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0 C_0$$

- 又因为在工艺映射的过程中fan-in的限制，我们采取Group Carry Lookahead Adder - 模块化超前进位加法器的方式，组内提前进位，组间行波进位：



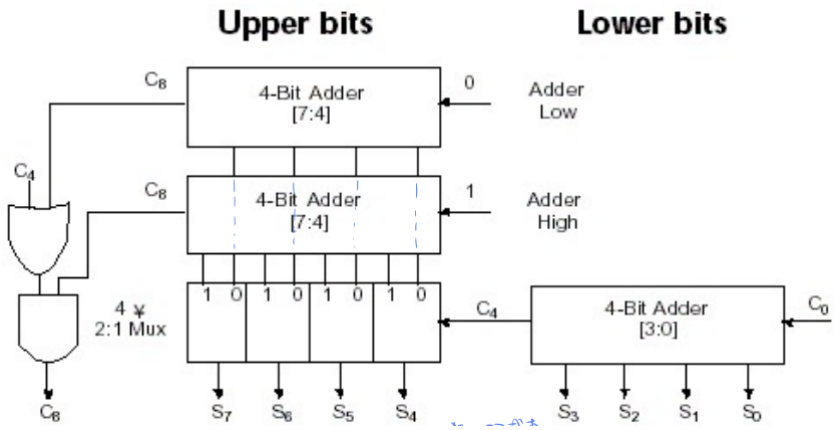
进位信号产生顺序分析：

$$C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 = G_{0\sim3} + P_{0\sim3}C_0$$
$$C_8 = G_7 + P_7G_6 + P_7P_6G_5 + P_7P_6P_5G_4 + P_7P_6P_5P_4C_4 = G_{4\sim7} + P_{4\sim7}C_4$$
$$C_{12} = G_{11}+P_{11}G_{10}+P_{11}P_{10}G_9+P_{11}P_{10}P_9G_8+P_{11}P_{10}P_9P_8C_8 = G_{8\sim11} + P_{8\sim11}C_8$$
$$C_{16}=G_{15}+P_{15}G_{14}+P_{15}P_{14}G_{13}+P_{15}P_{14}P_{13}G_{12}+P_{15}P_{14}P_{13}P_{12}C_{12} = G_{12\sim15} + P_{12\sim15}C_{12}$$



2. CSA - Carry Select Adder

用冗余值代替时间成本，把C_4 = `0`和`1`的情况都算好，然后等C_4的值确定后用其选择对应情况作为计算结果



乘法器

本质是加法器+寄存器实现

V1 - 基础思路

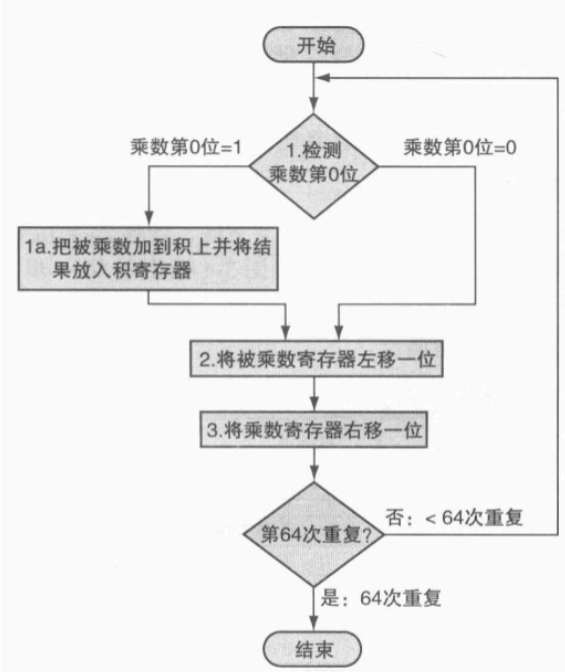
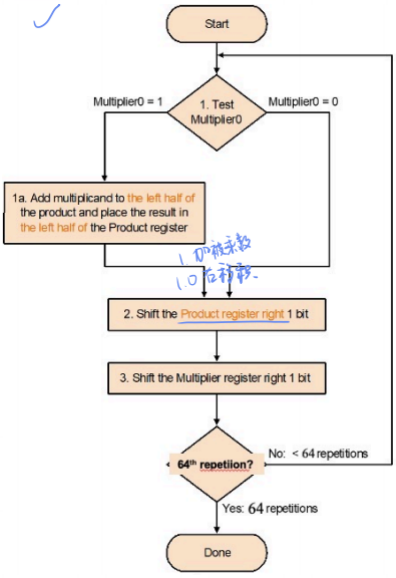


图 3-4　第一种乘法算法，采用了图 3-3 所示的硬件。如果乘数的最低有效位为 1，则将乘数加到积上。如果不是，则执行下一步。在下两步中将被乘数左移和乘数右移。将这三个步骤重复 64 次

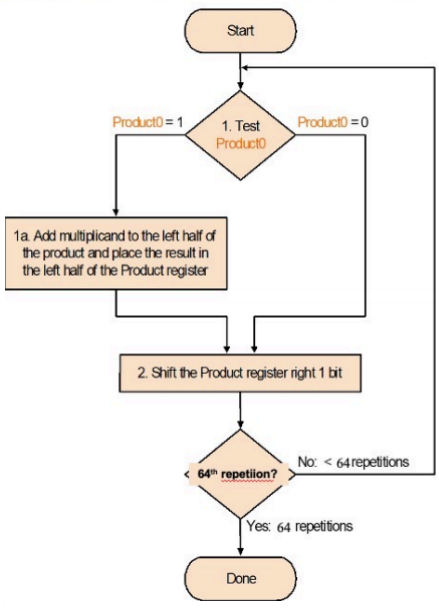
V2 - 将移动被乘数改为移动当前积

修改之后硬件实现的128bit-ALU被优化成了64bit-ALU（实际上是65位，因为要存进位）



我们可以看到，product寄存器的后半部分在最开始是不被使用的，因此这一部分空间我们可以加以利用：

V3 - 把乘数直接放到product寄存器后半部分



例子：

Multiplicand:	0001	Shift out	
Multiplier:×	0111		
	00000111		#Initial value for the product
1	00010111		#After adding 0001, Multiplier=1
	00001011	1	#After shifting right the product one bit
	0001		
2	00011011		#After adding 0001, Multiplier=1
	00001101	1	#After shifting right the product one bit
	0001		#After adding 0001, Multiplier=1
3	00011101		
	00001110	1	#After shifting right the product one bit
	0000		
4	00001110		#After adding 0001, Multiplier=0
	00000111	0	#After shifting right the product one bit

缺陷：计算结束后乘数信息丢失

有符号数乘法

不要用补码，绝对值相乘+判断符号即可

Booth's Algorithm

如果移位操作比加法操作硬件实现更快，那么这个算法就比普通乘法快

步骤（参考GPT）：

1. **初始化**：首先，准备两个操作数，被乘数和乘数，并将乘数进行二进制编码（BOOTH编码）。设置一个累加器（AC）和寄存器（Q），其中AC初始化为0，Q初始化为乘数，还需一个Q-1位，初始化为0。
2. **检查Q的最后两位**：根据Q的最后一位以及Q-1的值进行决策（`00`或`11`时不操作，`01`时**AC**加上被乘数，`10`时**AC**减去被乘数）。
3. **右移**：将AC、Q和Q-1联合右移一位。
4. **重复**：重复步骤2和3，直到完成所有位的处理。
5. **结果**：最终，AC和Q中存储的就是乘法的结果。

例子：

$2 * (-3) = -6 \rightarrow 0010 * 1101 = 11111010$

对照上面的步骤，图中的product左4位为AC，右4位为Q，最后一位为Q-1

iteration	step	Multiplicand	product
0	Initial Values	0010	0000 1101 0
1	1.c:10→Prod=Prod-Mcand	0010	1110 1101 0
	2: shift right Product	0010	1111 0110 1
2	1.b:01→Prod=Prod+Mcand	0010	0001 0110 1
	2: shift right Product	0010	0000 1011 0
3	1.c:10→Prod=Prod-Mcand	0010	1110 1011 0
	2: shift right Product	0010	1111 0101 1
4	1.d: 11 → ***no operation***	0010	1111 0101 1
	2: shift right Product	0010	1111 1010 1

被乘数 Multiplicand 是 0010，乘数 Multiplier 是 1101.
最开始将积 0000 放在高四位，1101 作为乘数放在低四位。 最开始 10，即执行减操作，$0000 - 0010 = 1110$. 答案依然放在高四位，随后右移，以此类推。
注意右移的时候是**算术右移**，bit_{-1} 也可能会改变。

算术右移即符号位保持不变

快速乘法器

为乘数的每一位提供加法器，通过并行树的方式加速计算：

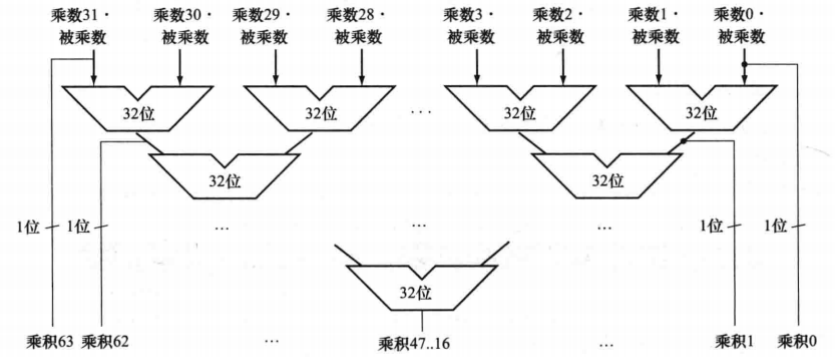


图 3-7 快速乘法器硬件结构。这个结构使用 31 个加法器“展开循环”来实现最小的时延，而不再是使用单个 32 位的加法器 31 次

从n次加法时间优化至$log_2 n$次加法时间，其中n为乘数/被乘数位数。

RISC-V乘法指令

- `mul` - 低64位乘法
- `mulh` - 高64位有符号乘法 - check for 64-bit overflow
- `mulhu` - 高64位无符号乘法
- `mulhsu` - 高64位乘法，一个数有符号一个数无符号

除法器

这一块可能看例子比看流程图更清晰

V1 - 基本实现

初始状态除数放在寄存器左半边，余数放在寄存器右半边

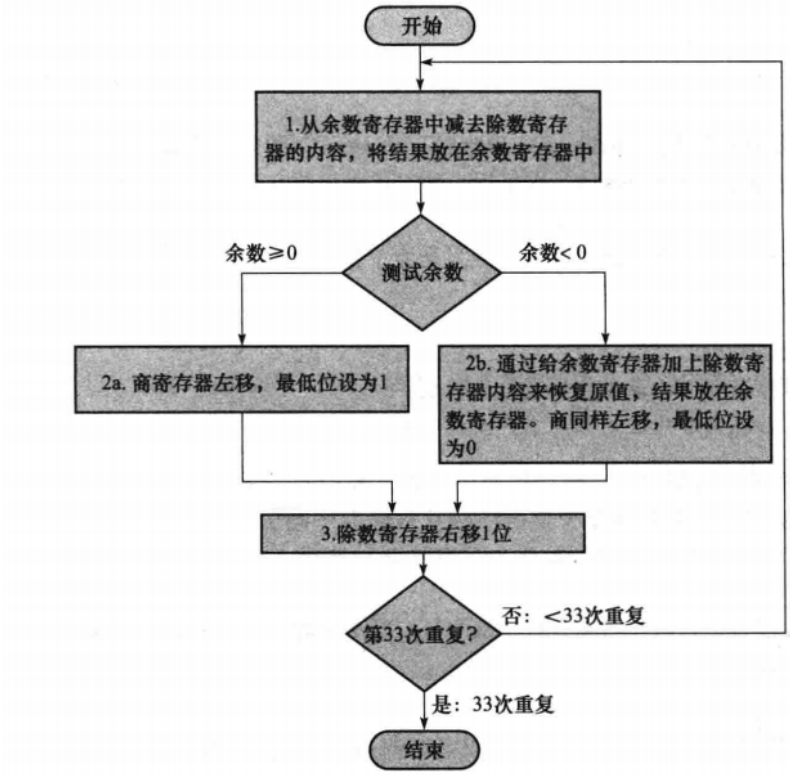


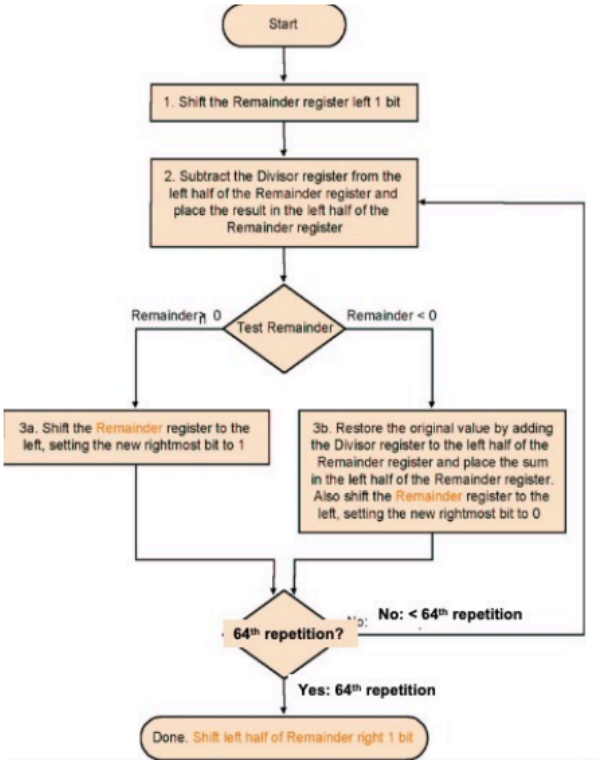
图 3-9　第一种除法算法，其硬件结构见图 3-8。如果余数为正，则将除数从被除数中减去，然后在第 2a 步取商为 1。如果第 1 步之后余数为负，则意味着除数不能从被除数中减去，所以在第 2b 步中商 0 并将除数加到余数上，即做第 1 步减法的逆操作。在第 3 步，进行最后的移位，根据下一个迭代的被除数，将除数适当对齐。这些步骤将要重复 33 次

迭代次数	步骤	商	除数	余数
0	初始值	0000	0010 0000	0000 0111
1	1：余数=余数–除数	0000	0010 0000	**①110 0111**
	2b：余数<0 ⇒+除数，商左移，上最低位上0	**0000**	0010 0000	**0000 0111**
	3：除数右移	0000	**0001 0000**	0000 0111
2	1：余数=余数–除数	0000	0001 0000	**①111 0111**
	2b：余数<0 ⇒+除数，商左移，上最低位上0	**0000**	0001 0000	**0000 0111**
	3：除数右移	0000	**0000 1000**	0000 0111
3	1：余数=余数–除数	0000	0000 1000	**①111 1111**
	2b：余数<0 ⇒+除数，商左移，上最低位上0	**0000**	0000 1000	**0000 0111**
	3：除数右移	0000	**0000 0100**	0000 0111
4	1：余数=余数–除数	0000	0000 0100	**⓪000 0011**
	2a：余数≥0 ⇒商左移，上最低位上1	**0001**	0000 0100	0000 0011
	3：除数右移	0001	**0000 0010**	0000 0011
5	1：余数=余数–除数	0001	0000 0010	**⓪000 0001**
	2a：余数≥0 ⇒商左移，上最低位上1	**0011**	0000 0010	0000 0001
	3：除数右移	0011	**0000 0001**	0000 0001

图 3-10　除法的例子，采用图 3-9 中的算法。图中圈起来的位用于决定下一步的操作

V2 - 寄存器优化实现

把除数寄存器和余数寄存器合并，一个寄存器同时用于存储Reminder和计算结果，最后左半部分是余数右半部分是结果



Example 7/2 for Division V3

- Well known numbers: 0000 0111/0010

iteration	step	Divisor	Remainder
0	Initial Values	0010	0000 0111
	Shift Rem left 1	0010	**0000 1110**
1	1.Rem=Rem-Div	0010	1110 1110
	2b: Rem<0 →+Div,sll R,R_0=0	0010	**0001 1100**
2	1.Rem=Rem-Div	0010	1111 1100
	2b: Rem<0 →+Div,sll R,R_0=0	0010	**0011 1000**
3	1.Rem=Rem-Div	0010	0001 1000
	2a: Rem>0 →sll R,R_0=1	0010	**0011 0001**
4	1.Rem=Rem-Div	0010	0001 0001
	2a: Rem>0 →sll R,R_0=1	0010	**0010 0011**
	Shift left half of Rem right 1		**0001 0011**

浙江大学 ZheJiang University 系统结构与网络安全研究所

同时用于存储Reminder和结果

有符号数除法

记住除数和被除数的符号，如果两者符号相异则商为负。

余数通过计算出商后重写基本公式计算，余数 = 被除数 - 商 * 除数。

快速除法

不能和快速乘法一样使用并行化计算因为相减与否和当前余数寄存器的符号位相关，有一个SRT的除法算法可以通过预测的方式一次计算多位（然后在后面的步骤修正），但是依然是串行算法。

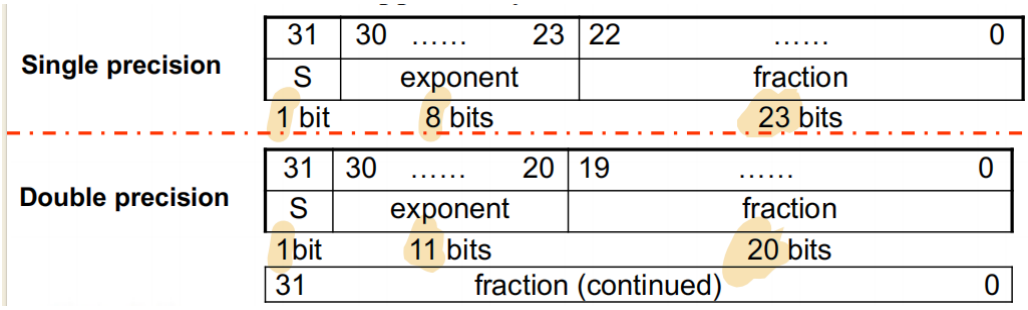
RISC-V除法指令

- 溢出和除0不报ERROR，只会返回特定值（make common case fast）
- `div` & `rem` - 有符号除法的商和余数
- `divu` & `remu` - 无符号除法的商和余数

浮点数

浮点数表示

$$(-1)^{sign} \bullet (1 + significand) \bullet 2^{exponent - bias}$$

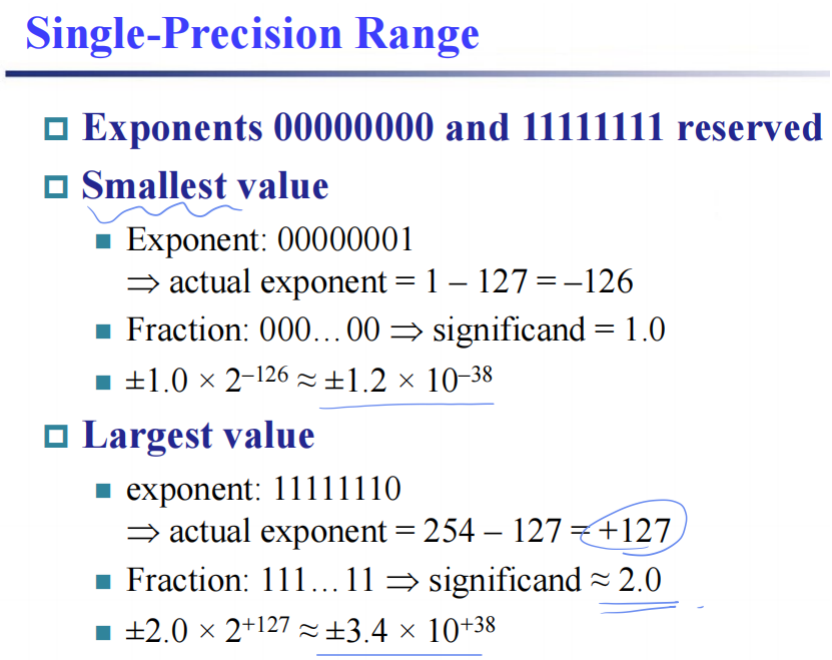


Bias = 127 for 单精度，1023 for 双精度

注意：次数是二进制下的次数，记实际次数为x，由$exp - bias = x$得exp

做题过程中注意是否有hidden 1，是否用2's complement表示有符号数

Problem1 - Range



双精度同理

- Overflow - too big to be represented
- Underflow - too small to be represented

Problem2 - Precision

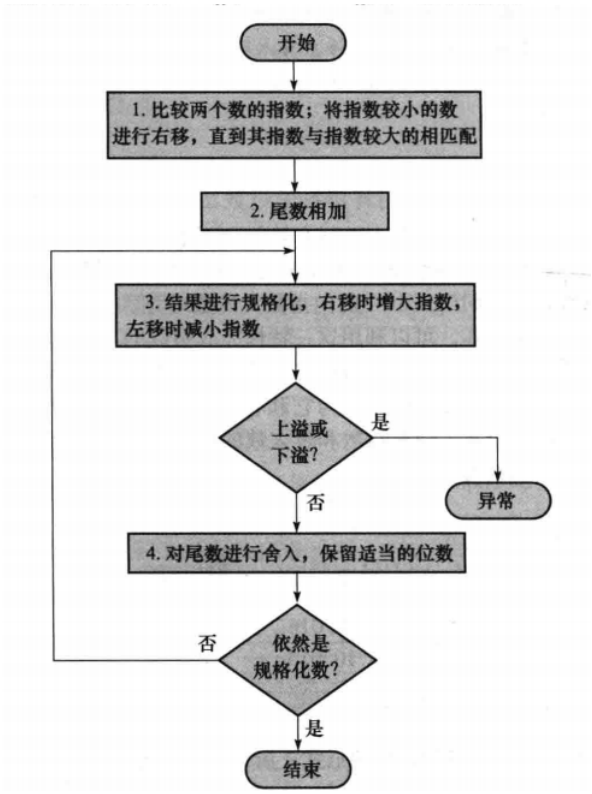
- Single: approx 2^{-23}
  - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
- Double: approx 2^{-52}
  - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Problem3 - Inf & NaN → 会限制range中的exp

- ±Infinity⟶Exp = 111...1，Frac = 000...0
- Not-a-Number⟶Exp = 111...1，Frac != 000...0

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

浮点数加法



Example y=0.5+(-0.4375) in binary

例子：

- $0.5_{10} = 1.000_2 \times 2^{-1}$
- $-0.4375_2 = -1.110_2 \times 2^{-2}$
- **Step1:The fraction with lesser exponent is shifted right until matches**

  $-1.110_2 \times 2^{-2} \rightarrow -0.111_2 \times 2^{-1}$
- **Step2: Add the significands**

$$\begin{array}{r} 1.000_2 \times 2^{-1} \\ +) \; -0.111_2 \times 2^{-1} \\ \hline 0.001_2 \times 2^{-1} \end{array}$$

- **Step3: Normalize the sum and checking for overflow or underflow**

  $0.001_2 \times 2^{-1} \rightarrow 0.010_2 \times 2^{-2} \rightarrow 0.100_2 \times 2^{-3} \rightarrow 1.000_2 \times 2^{-4}$

  正确有效位数
- **Step4: Round the sum**

  $1.000_2 \times 2^{-4} = 0.0625_{10}$

浮点数乘法

$$(s1 \cdot 2^{e1}) \cdot (s2 \cdot 2^{e2}) = (s1 \cdot s2) \cdot 2^{e1+e2}$$

差别：带 bias

例 01 例题·二进制浮点乘法

按照图 3-16 中的步骤，试计算 0.5_{10} 和 $-0.437\,5_{10}$ 的乘积。

例 01 答案

在二进制下，也就是将 $1.000_2 \times 2^{-1}$ 和 $-1.110_2 \times 2^{-2}$ 相乘。

步骤 1：将不带偏阶的指数相加：

$$-1 + (-2) = -3$$

或者，使用带偏阶的表达：

$$\begin{aligned}&(-1 + 127) + (-2 + 127) - 127\\ &= (-1 - 2) + (127 + 127 - 127)\\ &= -3 + 127 = 124\end{aligned}$$

步骤 2：将有效数相乘：

$$\begin{array}{r} 1.000_2 \\ \times \quad 1.110_2 \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1110000_2 \end{array}$$

积是 $1.110\,000_2 \times 2^{-3}$，但是我们需要保存 4 位，所以为 $1.110_2 \times 2^{-3}$。

步骤 3：现在我们检查积以确保其是规格化的，然后检查指数以确定上溢和下溢是否发生。这个积已经是规格化的，并且，因为 $127 \geqslant -3 \geqslant -126$，所以没有上溢和下溢。（使用带偏阶的表达，$254 \geqslant 124 \geqslant 1$，所以指数域可以表达。）

步骤 4：对积舍入没有使其发生变化：

$$1.110_2 \times 2^{-3}$$

步骤 5：因为初始的源操作数的符号相异，所以积的符号为负。因此，积为

$$-1.110_2 \times 2^{-3}$$

为了检查结果，将其转化为十进制：

$$-1.110_2 \times 2^{-3} = -0.001\,110_2 = -0.001\,11_2 = -7/2^5_{10} = -7/32_{10} = -0.218\,75_{10}$$

而 0.5_{10} 和 $-0.437\,5_{10}$ 的积确实是 $-0.218\,75_{10}$。 □

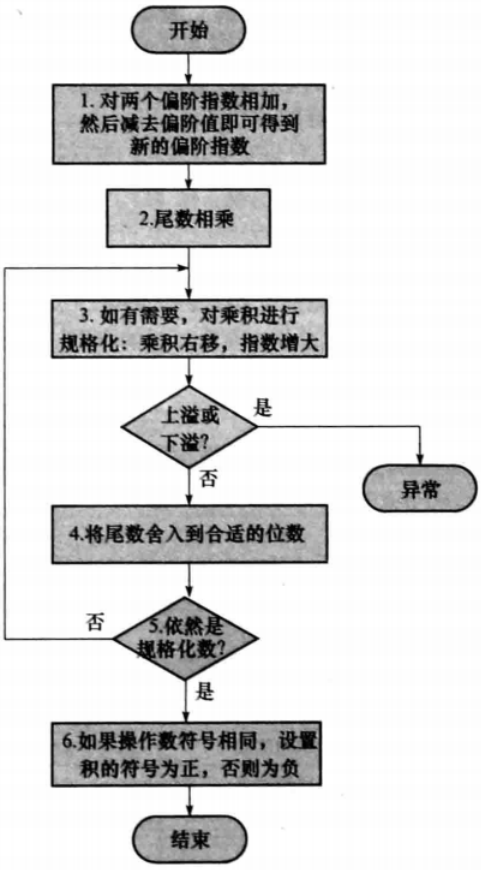


图 3-16 浮点乘法。正常的路径是执行一次步骤 3 和步骤 4，但如果舍入使积变为非规格化数，则需要重复步骤 3

浮点数除法

1. exp位相减（记得处理bias）
2. 尾数相除
3. 规格化
4. 舍入
5. 判断符号

浮点数运算精度问题

加额外的位保证计算精度准确

- Guard - 保护位：在浮点数中间计算中，在右边多保留的两位中的首位；用于提高舍入精度
- Round - 舍入位：在浮点数中间计算中，在右边多保留的两位中的第二位；使浮点中间结果满足浮点格式，得到最接近的数
- *Sticky - 粘滞位：舍入位后面还有1就记为1*

浮点精确性用**尾数最低位（ulp）**衡量，IEEE754保证计算机数的误差在半个ulp以内。

参考资料：马德ppt、计组教材中文版，Hobbitqia计组笔记、hm计组笔记