Chapter 3

计算机的算术运算

计算机的算术运算

- **整数上的运算**
  - **加法和减法**
- 数字逻辑入门
- 整数上的乘法
- 实数上的运算（浮点运算）
  - 浮点的表示与运算

整数加法

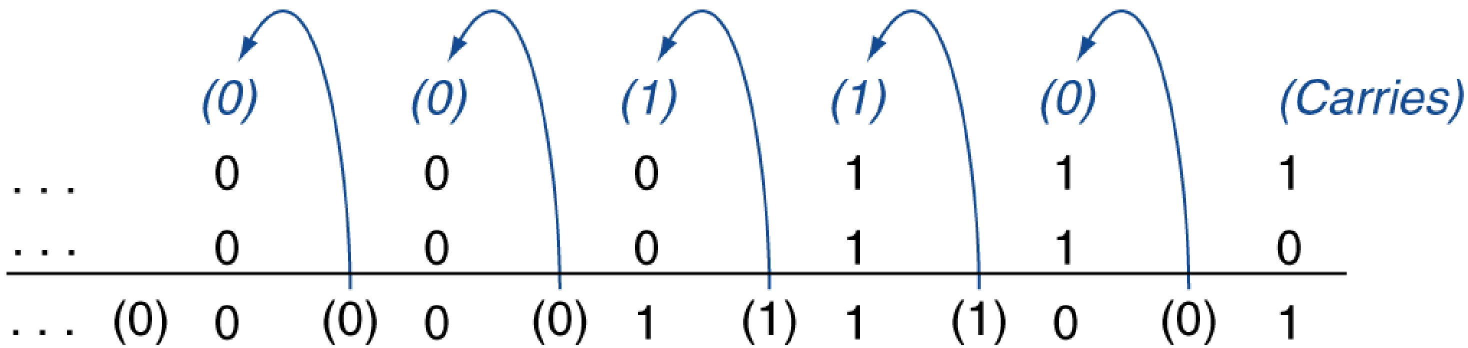
- 例子: 7 + 6

		(0)		*(0)*		*(1)*		*(1)*		*(0)*		*(Carries)*
. . .		0		0		0		1		1		1
. . .		0		0		0		1		1		0
. . .	(0)	0	(0)	0	(0)	1	(1)	1	(1)	0	(0)	1

- 如果结果超出范围，就会溢出。
  - 思考：如何检测?

整数加法

- 例子: 7 + 6



- 如果结果超出范围，就会溢出。
  - 如何检测?
    - 正数+负数，不会溢出
    - 整数+整数，结果为负数，溢出
    - 负数+负数，结果为正数，溢出

整数减法

- 等价于：加上减数的相反数
- Example: 7 – 6 = 7 + (–6)

+7:	0000 0000 … 0000 0111
–6:	1111 1111 … 1111 1010
+1:	0000 0000 … 0000 0001

- 如果结果超出范围，就会溢出。
  - 思考：如何检测?

整数减法

- 等价于：加上减数的相反数
- Example: 7 – 6 = 7 + (–6)

+7:	0000 0000 … 0000 0111
–6:	1111 1111 … 1111 1010
+1:	0000 0000 … 0000 0001

- 如果结果超出范围，就会溢出
  - 如何检测
    - 正数-正数，或者负数-负数，不会溢出
    - 正数-负数，结果为负数，溢出
    - 负数-正数，结果为正数，溢出

溢出

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

加法器的硬件实现?

计算机的算术运算

- **整数上的运算**
  - **加法和减法**
- 数字逻辑入门
- 实数上的运算（浮点运算）
  - 浮点的表示与运算

计算机的算术运算

- 整数上的运算
  - 加法和减法
- **数字逻辑入门**
- 整数上的乘法
- 实数上的运算（浮点运算）
  - 浮点的表示与运算

逻辑函数与真值表

- 逻辑函数
  - 输入输出都是逻辑值
- 最简单的逻辑函数
  - 输出由输入完全决定
    - 也称组合逻辑
- 逻辑函数的两种表示
  - **真值表**
  - 布尔等式
- 例子
    - D：输入有1个1
    - E：输入有2个1
    - F：输入有3个1

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

逻辑函数与布尔等式

- 布尔代数
  - 基于逻辑值的代数系统
  - And运算： A · B
  - Or运算： A+B
  - Not运算： $\overline{A}$
- 布尔等式的例子

$$D = A + B + C$$

$$F = A \cdot B \cdot C$$

  - E呢？

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

逻辑函数与布尔等式

- 布尔代数
  - 基于逻辑值的代数系统
  - And运算： A · B
  - Or运算： A+B
  - Not运算： $\overline{A}$
- 布尔等式的例子

$$D = A + B + C$$

$$F = A \cdot B \cdot C$$

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot \overline{(A \cdot B \cdot C)}$$

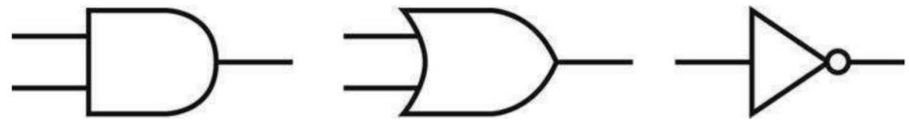
$$\|$$

$$E = (A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$$

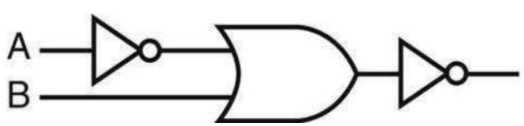
Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

布尔运算的电路实现——门

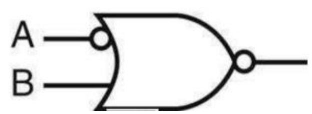
- 布尔代数
  - And运算：与门
  - Or运算： 或门
  - Not运算：非门



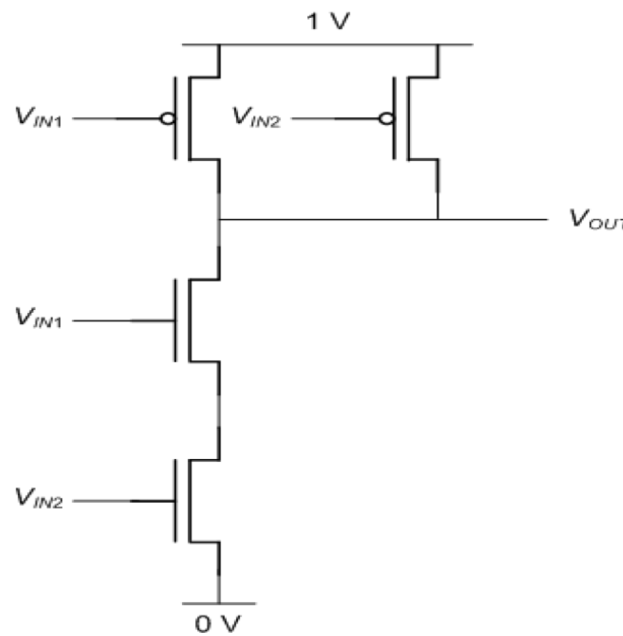
- 例子 $\overline{\overline{A} + B}$



完整版



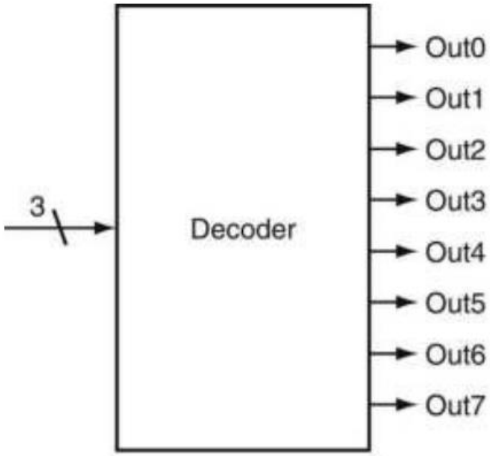
简化版



与非门的实现

更多的门——译码器

- 例子
  - 输入3位信号，输出8（2^3）个1位信号
  - 可以用作地址线



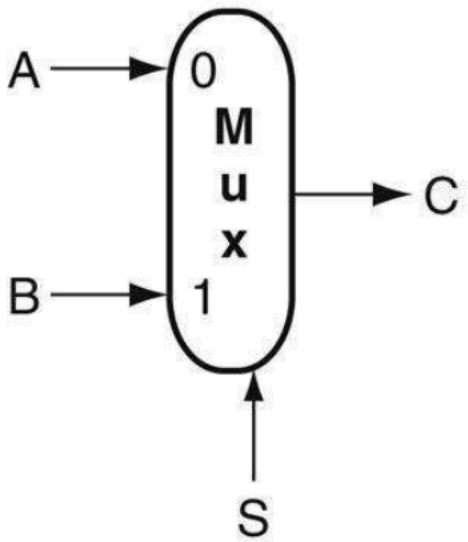
Inputs			Outputs							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

译码器的图例

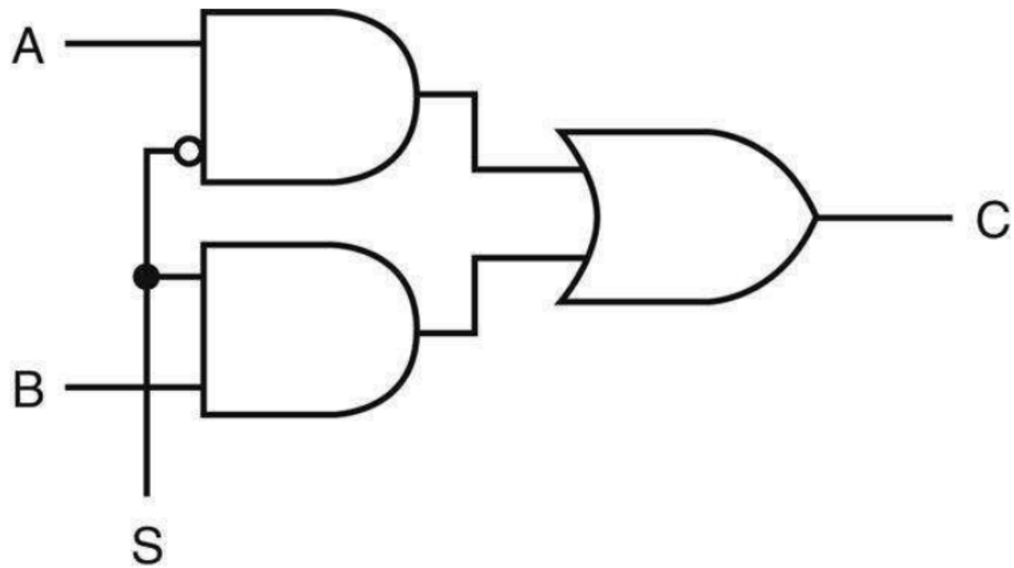
译码器的真值表

更多的门——选择器

- 例子
  - S是控制值，用于选择哪个输入作为输出



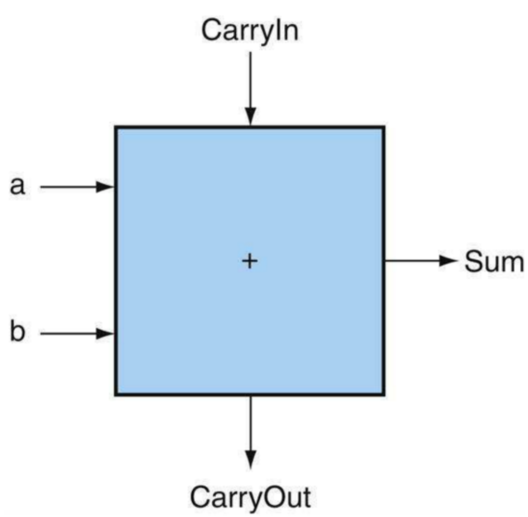
选择器的图例



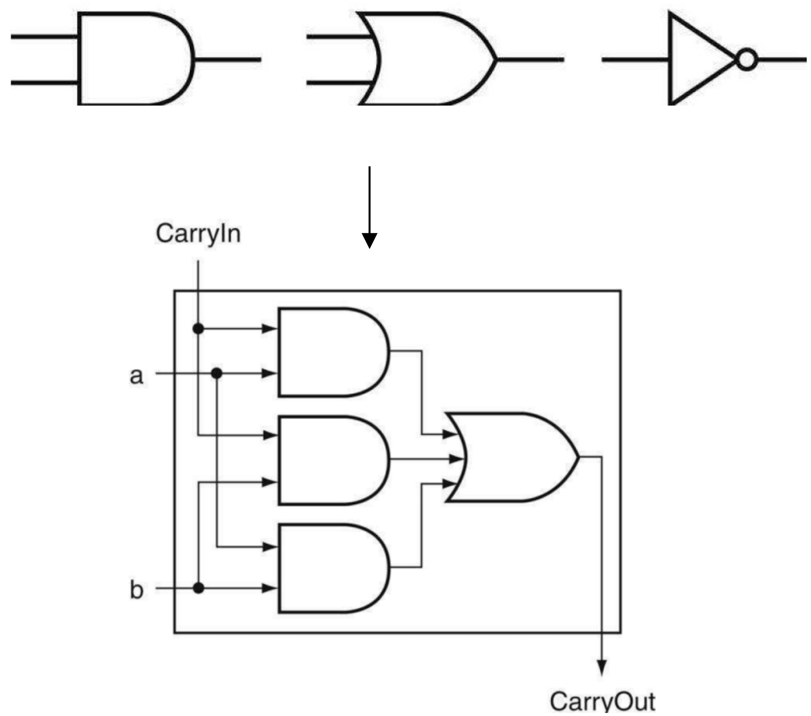
选择器的实现

1-bit 加法器

- $$\text{Sum} = (a \cdot \bar{b} \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$
- $$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$



1-bit加法器

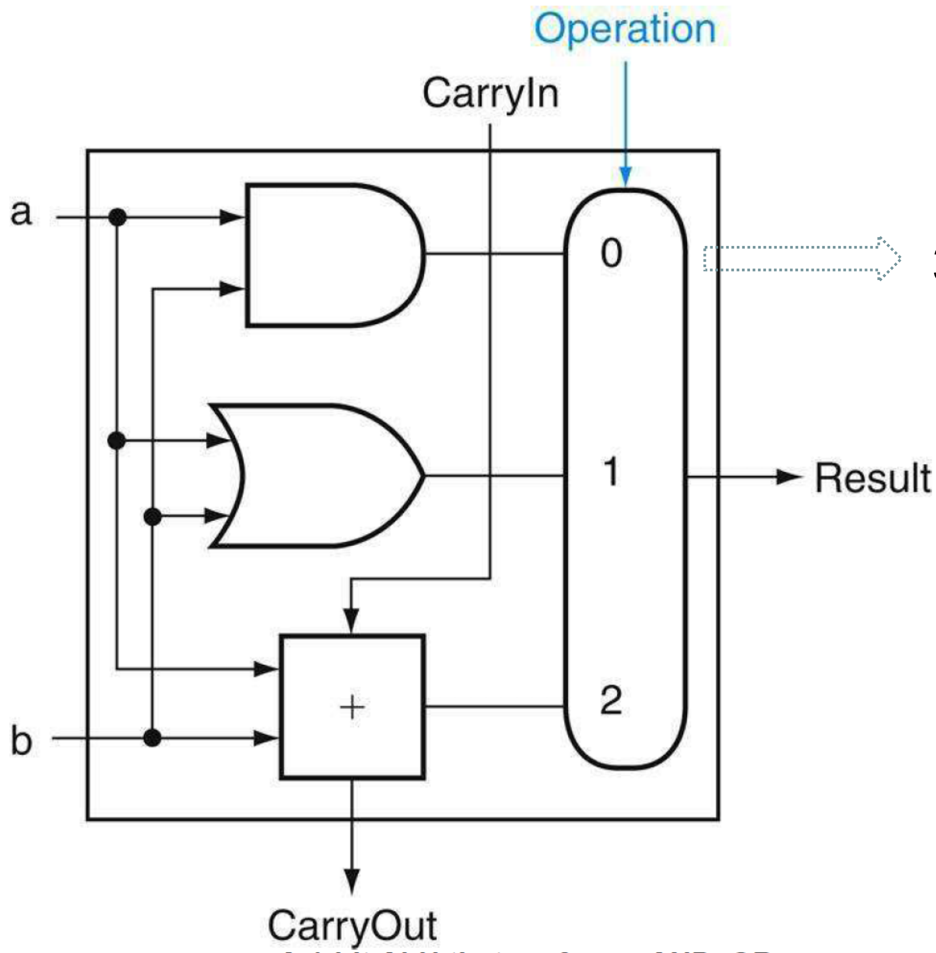


1-bit加法器的CarryOut逻辑

1-bit减法器？ALU？

- 减法器?
  - 将输入b先反相，再+1，然后利用加法器逻辑
- Arithmetic Logic Unit算术逻辑单元
  - 算术运算：add
  - 逻辑运算：and、or

1-bit ALU（附录A.5)



3输入的选择器

FIGURE A.5.6 A 1-bit ALU that performs AND, OR, and addition (see Figure A.5.5).

n-bit ALU

- 直接实现：级联实现
  - 64位加法的速度?
  - 需要串行64次1位加法
- 更快的实现方法（附录A.6）
  - Carry Lookahead adder

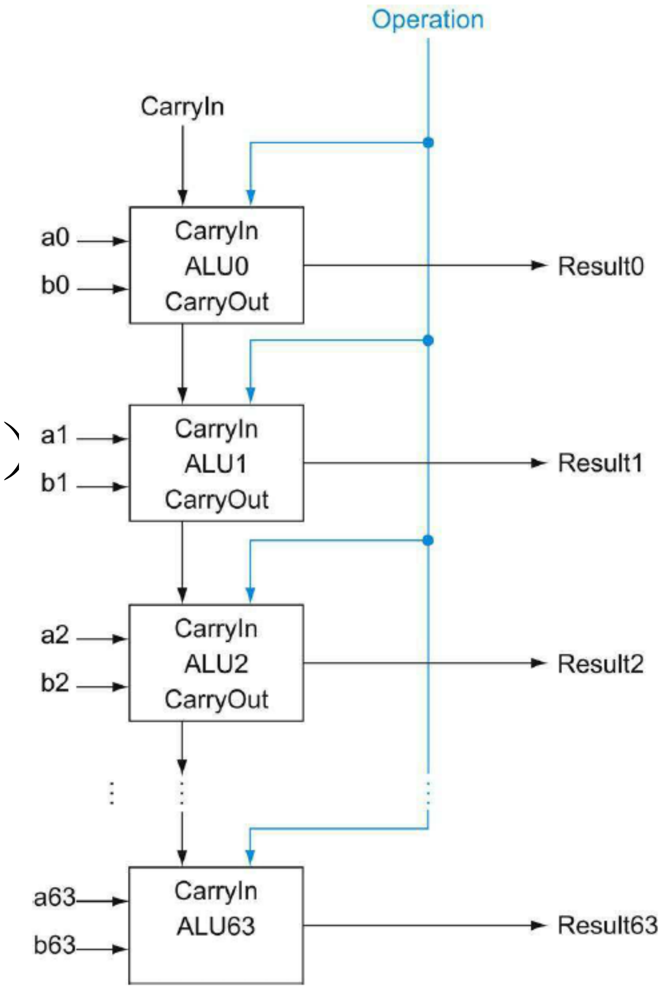


FIGURE A.5.7 **A 64-bit ALU constructed from 64 1-bit ALUs.**

计算机的算术运算

- 整数上的运算
  - 加法和减法
- 数字逻辑入门
- **整数上的乘法**
- 实数上的运算（浮点运算）
  - 浮点的表示与运算

整数乘法运算

- Start with long-multiplication approach

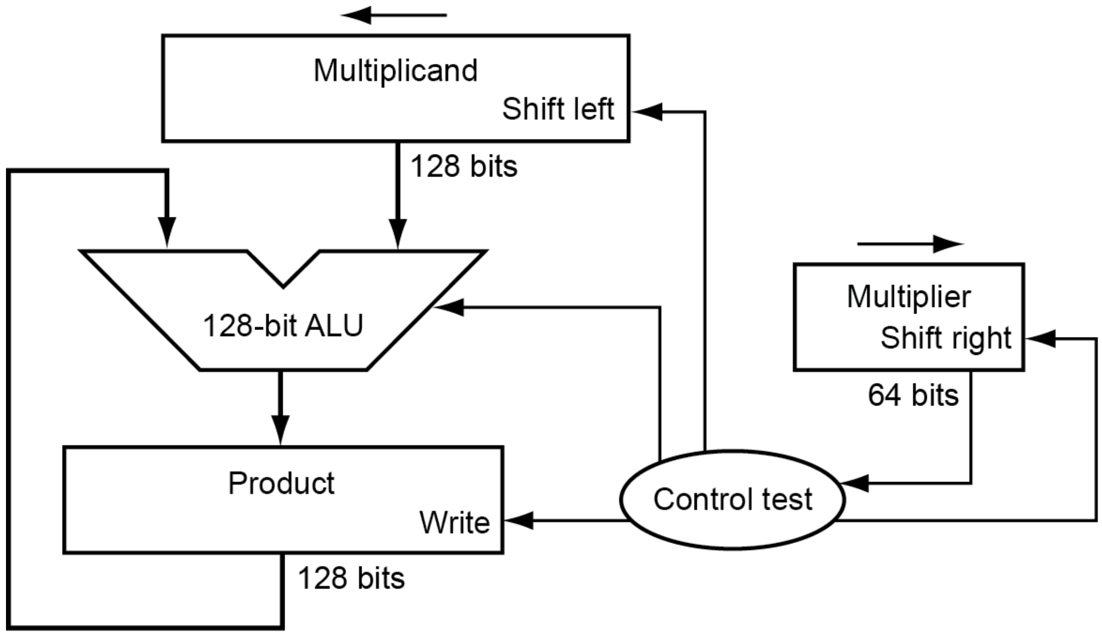
被乘数

乘数

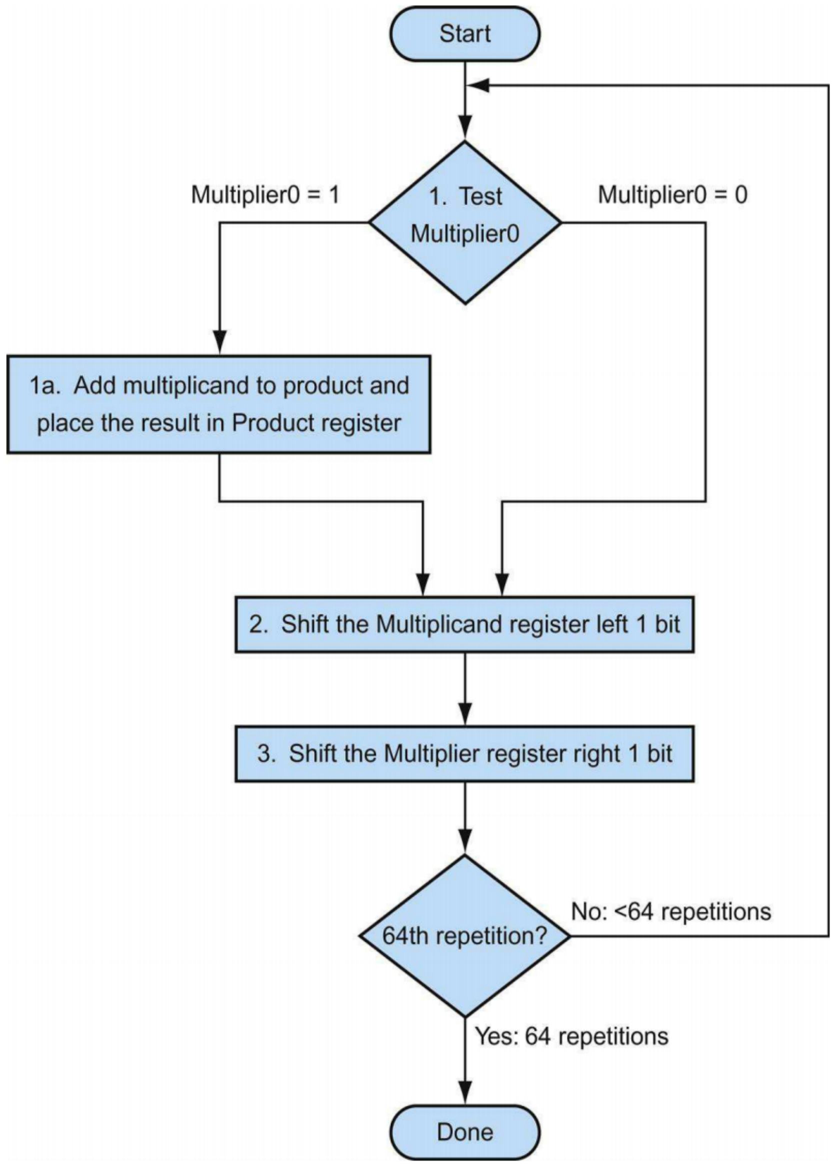
积

```
     1000
×    1001
---------
     1000
    0000
   0000
  1000
---------
  1001000
```

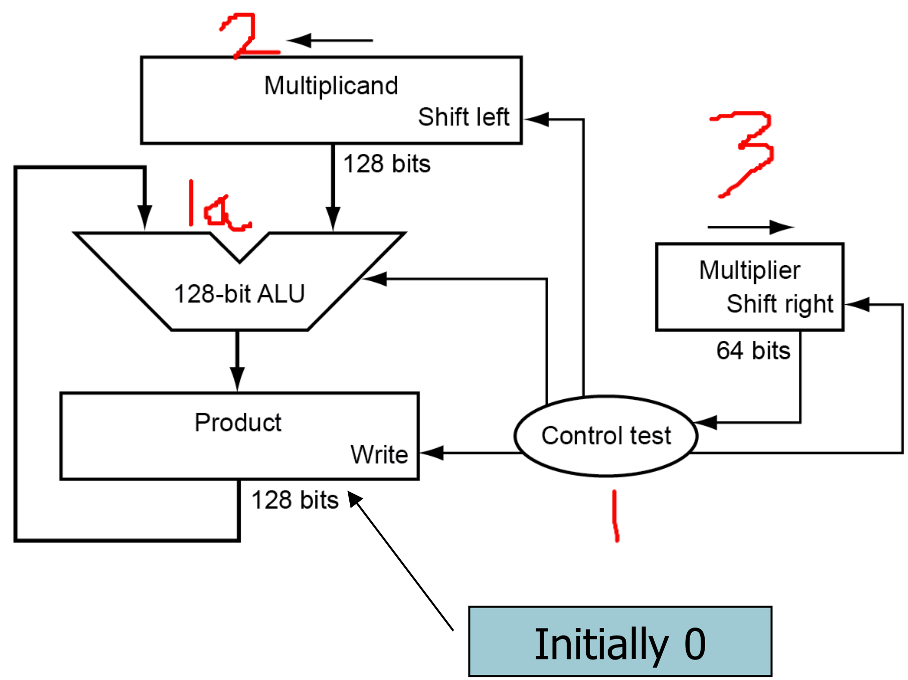
积的长度=两个乘数的长度之和



乘法器

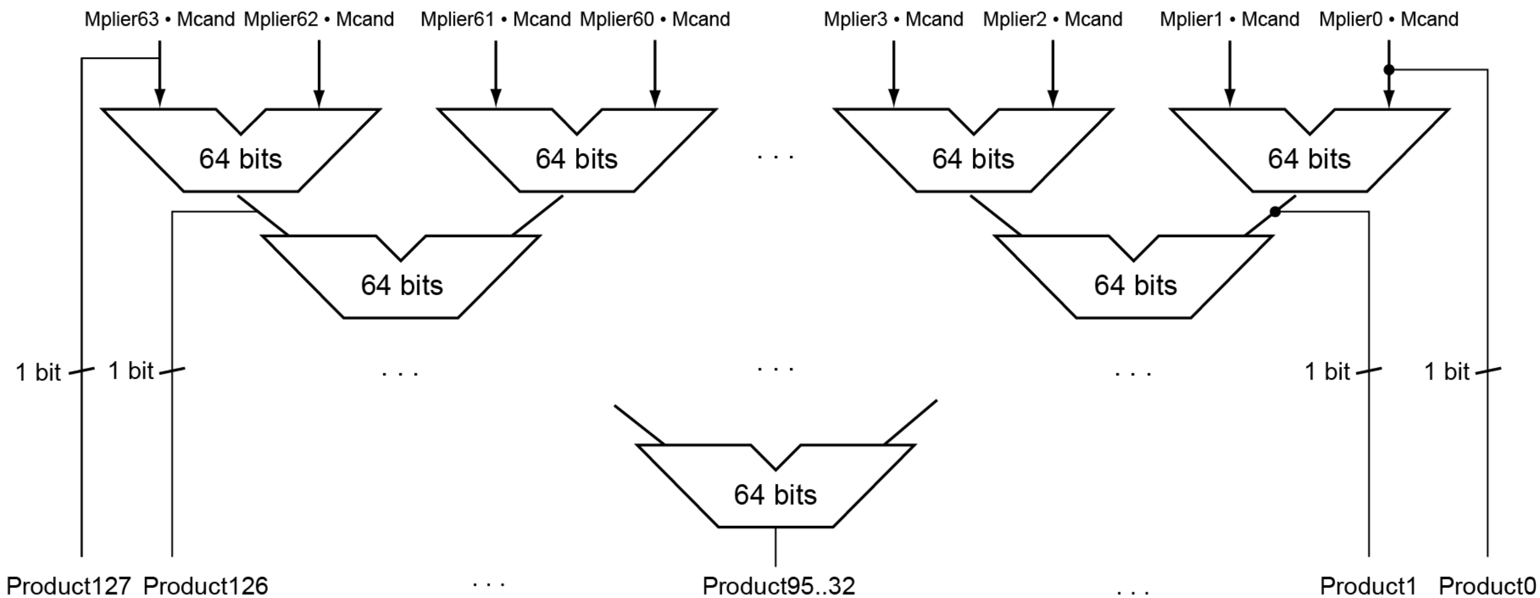


64位加法的速度?
= 64次串行的64位加法



更快的乘法器

- 使用多个加法器
  - 成本/性能的折衷



- 可以流水化
  - 并行完成多个乘法

RISC-V 乘法指令

- 四条乘法指令:
  - mul: multiply
    - 只提供乘积的低32（或64）位
  - mulh: multiply high
    - 只提供乘积的高32（或64）位——限有符号乘法
    - 用来检测结果是否溢出
  - mulhu: multiply high unsigned
    - 只提供乘积的高32（或64）位——限无符号乘法
  - mulhsu: multiply high signed/unsigned
    - 只提供乘积的高32（或64）位——限无符号乘有符号
  - Use mulh result to check for 64-bit overflow

浮点数

- 表示实数
  - 包括很小的数，以及很大的数
- 类似10进制的科学记数法
  - -2.34×10^{56}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$
- 2进制的科学记数法
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- C语言中的 **`float`** 和**`double`** 类型

规格化

非规格化

浮点数标准

- 由IEEE Std 754-1985标准进行定义
- 应对浮点表示的日益五花八门
  - 导致可移植性问题
- 目前该标准被广泛采纳
- 两种表示
  - 单精度 (32-bit)
  - 双精度 (64-bit)

IEEE 浮点数格式

单精度: 8 bits　　　　单精度: 23 bits
双精度: 11 bits　　　　双精度: 52 bits

S	Exponent	Fraction

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: 符号位 (0 ⇒ 非负数, 1 ⇒ 负数)
- 科学记数法中规格化表示的**小数**: 1.0 ≤ |尾数的值| < 2.0
  - **总是**有一个整数部分的1，因此无需硬件保存（隐含位）
  - 因此，实际的小数 = 小数编码Fraction+前面的 “1.”
- 指数编码Exponent: 实际的指数值 + 偏移值,
  - 单精度中偏移值为127;
  - 双精度中偏移值为1023;

单精度浮点数的表示范围

- 指数编码00000000 和11111111 保留另作他用
- 最小的值
  - 指数编码: 00000001
    $\Rightarrow$ 实际的指数 = 1 – 127 = –126
  - 小数编码: 000…00 $\Rightarrow$ 实际的小数 = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- 最大的值
  - 指数编码: 11111110
    $\Rightarrow$ 实际的指数 = 254 – 127 = +127
  - 小数编码: 111…11 $\Rightarrow$ 实际的小数≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

双精度浮点数的表示范围

- 指数编码0000…00 和1111…11保留另作他用
- 最小的值
  - 指数编码: 00000000001
    $\Rightarrow$ 实际的指数 = 1 – 1023 = –1022
  - 小数编码: 000…00 $\Rightarrow$ 实际的小数= 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- 最大的值
  - 指数编码: 11111111110
    $\Rightarrow$ 实际的指数 = 2046 – 1023 = +1023
  - 小数编码: 111…11 $\Rightarrow$ 实际的小数≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

浮点数的精度

- 相对精度
  - 所有的小数编码都重要
  - 单精度: 接近2^{-23}
    - $2^{-23}=10^{-k}$ ==> $\log_{10}(2^{-23}) = \log_{10}(10^{-k})$
      - $k = 23 \times \log_{10}2$
    - $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ 位10进制小数精度
  - 双精度: 2^{-52}
    - $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ 位10进制小数精度

浮点数的例子

- 用2进制格式表示–0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - 符号位 = 1
  - 小数=1.1 -> 小数编码= $1000\ldots00_2$
  - 指数=-1 -> 指数编码= –1 + 偏移
    - 单精度: –1 + 127 = 126 = 01111110_2
    - 双精度: –1 + 1023 = 1022 = 01111111110_2
- 单精度: 1011111101000…00
- 双精度: 1011111111101000…00

浮点数的例子

- 下列二进制表示的浮点数是多少?
  11000000101000…00
  - 符号位 = 1
  - 小数编码= $01000\ldots00_2$ -> 小数= 1.25
  - 指数编码= 10000001_2 -> 指数=129-127
- $x = (-1)^1 \times (1 + 0.01_2) \times 2^{(129-127)}$
  $= (-1) \times 1.25 \times 2^2$
  $= -5.0$

练习

- 写出 63.25 对应的 IEEE 754 标准双精度的浮点格式
- =1.1111101* 2^5
- 指数位:5+1023=1024+4 =10000000100
- 小数位:(45个0)1111101

- 写出二进制 0x0c000000 表示的 IEEE 754 标准浮点数.

- 00011000 24-127=-103 得1.0*2^-103

非规格化数

- 指数编码= 000...0 ⇒ 隐含位为0.
  - 指数 = 1 – 偏移(=-126, 而不是*E* = 0 – ***Bias***)
  - 小数 = 0 + 小数编码(而不是1.0)

- 比规格化数小
  - 最小的规格化数 $1.0000\,0000\,0000\,0000\,0000\,000_{two} \times 2^{-126}$
  - 最小的非规格化数

$$0.0000\,0000\,0000\,0000\,0000\,001_{two} \times 2^{-126}, \ or\ 1.0_{two} \times 2^{-149}$$

- 如何表示零： 000...0
  - 2种零的表示!

无穷与非法数

- 指数编码 = 111…1, 小数编码= 000…0
  - ±**无穷**, x/0.0 = +∞, x/−0.0 = −∞
- 指数编码 = 111…1, 小数编码 ≠ 000…0
  - Not-a-Number (NaN) **非法数**
  - 表示非法或未定义的结果
    - e.g., 0.0 / 0.0
    - e.g., sqrt(–1), ∞ − ∞, ∞ × 0

回顾: 浮点数

- 规格化数
  - 指数E = 指数编码- 127
  - 小数F = **1.**小数编码
  - 符号S = +(0), -(1)
- 非规格化数
  - 指数编码= 000…0, and 指数E = 1 - Bias (not 0-Bias!)
  - F = **0.**小数编码
- 无穷：指数编码=111…1, 小数编码=000…0
- 非法数：指数编码=111…1, 小数编码!=000…0

浮点数加法

- 考虑 4-digit 10进制小数加法的例子
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- 1. 对齐小数点位
  - 对指数小的数进行移位
  - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
- 2. 小数相加
  - $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$
- 3. 规格化运算结果 & 检测有无溢出
  - 1.0015×10^{2}
- 4. 舍入&重新规格化（如果有必要的话）
  - 1.002×10^{2}

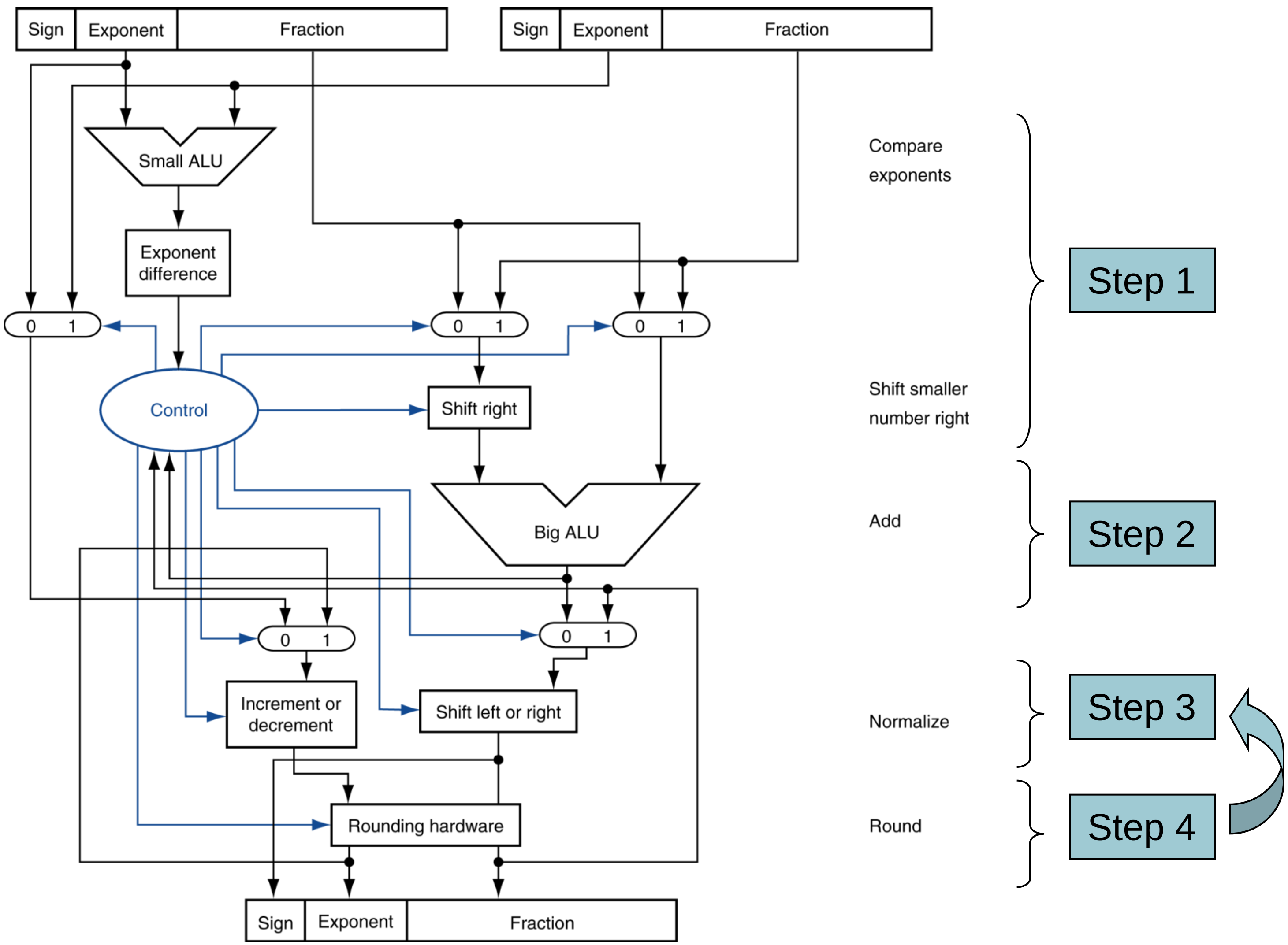
浮点数加法

- 现在考虑 4-digit 2进制浮点加法的例子
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. 对齐小数点位置
  - 对较小的数进行移位
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. 小数部分相加
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2{-}1 = 0.001_2 \times 2^{-1}$
- 3. 规格化运算结果 & 检测有无溢出
  - $1.000_2 \times 2^{-4}$, 没有溢出
    - -126 <= -4 <= 127
- 4. 舍入&重新规格化
  - $1.000_2 \times 2^{-4}$ (无需改变) = 0.0625

浮点加法器的硬件

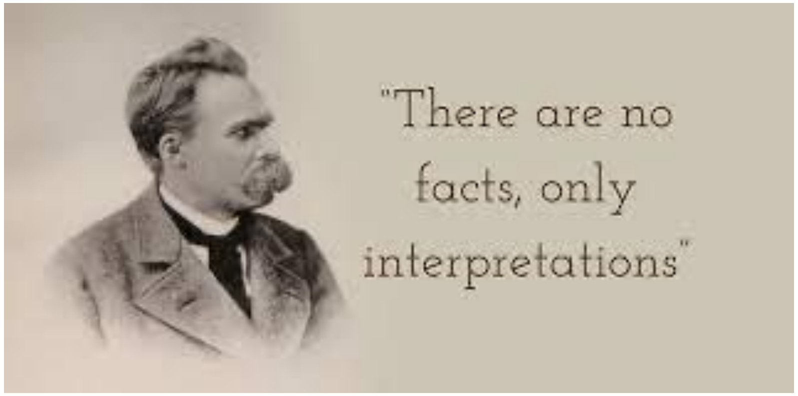
- 比整数加法器复杂很多
- 过程冗长，难以在单个时钟周期内完成
  - 远远长于整数加法的过程
  - 如果单个时钟周期，其他指令都会浪费时间
  - 因此，浮点加法器通常多个时钟周期
  - 可以流水化实现

浮点加法器的硬件


Sign
Exponent
Fraction
Sign
Exponent
Fraction
Small ALU
Exponent difference
0 1
0 1
0 1
Control
Shift right
Big ALU
0 1
0 1
Increment or decrement
Shift left or right
Rounding hardware
Sign
Exponent
Fraction
Compare exponents
Shift smaller number right
Add
Normalize
Round
Step 1
Step 2
Step 3
Step 4

结语

- 比特位本身没有独立的意义
  - 比特位在不同上下文，进行不同的解释，才有意义
    - 无符号编码、补码
    - 浮点编码
    - 指令编码



- 数值的计算机表示
  - 有限的范围、有限的精度
  - 程序中需要考虑这种“有限”性

RISC-V中的浮点指令

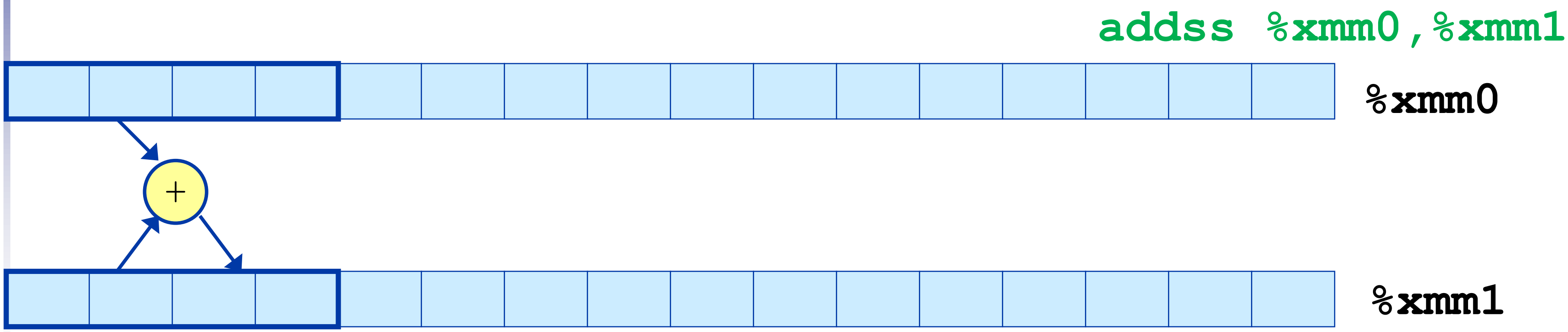
- 使用专门的浮点寄存器: f0, …, f31
  - 双精度寄存器
  - 单精度的值，存放在低32位
- 浮点指令只操作浮点寄存器
- 浮点 load & store 指令
  - `flw`, `fld`
  - `fsw`, `fsd`

字内并行

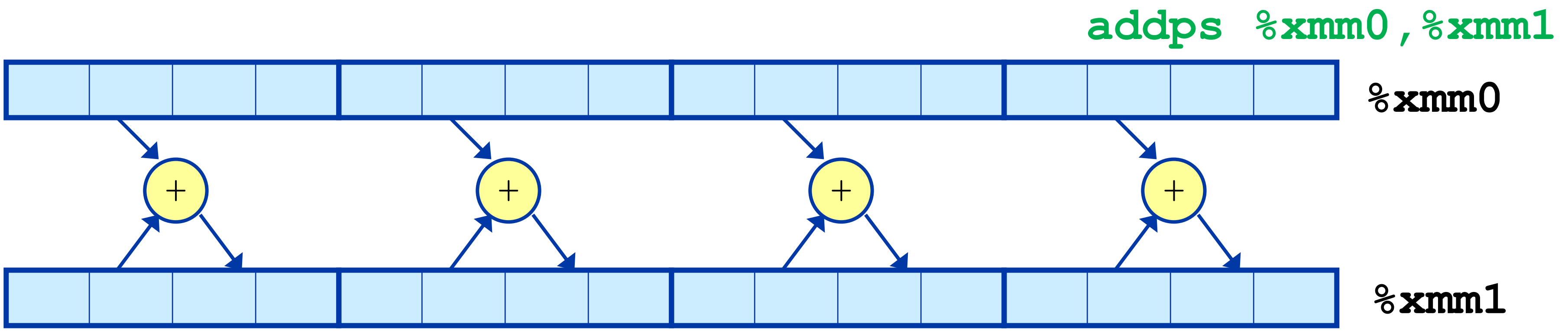
- 图形、音频等多媒体应用，经常会在相邻的、同类数据上，进行同样的运算
  - 字内并行可以**同时**进行这些运算
  - 比如，128-位加法器，可以用于
    - Sixteen 8-bit adds
    - Eight 16-bit adds
    - Four 32-bit adds
- 也称为：数据级并行、向量并行
  - 或者，单指令多数据
  - Single Instruction, Multiple Data (SIMD)

SIMD in x86

- addss：标量计算



- addps：向量计算



矩阵乘法

- 未优化的 C 代码:

```
void dgemm(int n, double* A, double* B, double* C){
    for(int i=0; i <n; ++ i)
        for(int j=0;j <n; ++j ) {
            double cij = C[i*n+j];                  // cij = C[i][j]
                for(int k=0; k<n; ++k) {
                    cij += A[i*n+k*] * B[k*n+j];    //cij += A[i][k] * B[k][j]
               C[i*n+j] = cij;                      // C[i][j] = cij
```

矩阵乘法

- x86 汇编代码:

```
vmovsd (%r10),%xmm0   # Load 1 element of C into %xmm0
mov %rsi,%rcx         # register %rcx = %rsi
xor %eax,%eax         # register %eax = 0
vmovsd (%rcx),%xmm1   # Load 1 element of B into %xmm1
add %r9,%rcx          # register %rcx = %rcx + %r9
vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1, element of A
add $0x1,%rax         # register %rax = %rax + 1
cmp %eax,%edi         # compare %eax to %edi
vaddsd %xmm1,%xmm0,%xmm0 # Add %xmm1, %xmm0
jg 30 <dgemm+0x30>    # jump if %eax > %edi
add $0x1,%r11d        # register %r11 = %r11 + 1
vmovsd %xmm0,(%r10)   # Store %xmm0 into C element
```

矩阵乘法

- 优化的 C 代码:

```c
#include <x86intrin.h>
void dgemm (int n, double* A, double* B, double* C)
{
 for ( int i = 0; i < n; i+=4 )
  for ( int j = 0; j < n; j++ ) {
   __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
   for( int k = 0; k < n; k++ )
     c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
               _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
               _mm256_broadcast_sd(B+k+j*n)));
   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
  }
}
```

矩阵乘法

- 优化的 x86 汇编代码:

```
1. vmovapd (%r11),%ymm0        # Load 4 elements of C into %ymm0
2. mov %rbx,%rcx               # register %rcx = %rbx
3. xor %eax,%eax               # register %eax = 0
4. vbroadcastsd (%rax,%r8,1),%ymm1 # Make 4 copies of B element
5. add $0x8,%rax               # register %rax = %rax + 8
6. vmulpd (%rcx),%ymm1,%ymm1   # Parallel mul %ymm1,8 A elements
7. add %r9,%rcx                # register %rcx = %rcx + %r9
8. cmp %r10,%rax               # compare %r10 to %rax
9. vaddpd %ymm1,%ymm0,%ymm0    # Parallel add %ymm1, %ymm0
10. jne 50 <dgemm+0x50>        # jump if not %r10 != %rax
11. add $0x1,%esi              # register % esi = % esi + 1
12. vmovapd %ymm0,(%r11)       # Store %ymm0 into 4 C elements
```

矩阵乘法的例子(cont)（p118)

- 例子：矩阵乘法
  - python：6小时
  - C（第2章）：提升200倍
  - 向量指令（第3章）：提升8倍
  - 指令级并行（第4章）：提升2倍
  - 访存优化（第5章）：提升1.5倍
  - 线程级并行（第6章）：提升12～17倍
  - 总共提升近5000倍，不到1秒

```
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i]j] += A[i][k] * B[k][j]
```

python版本

```
void dgemm(int n, double* A, double* B, double* C){
    for(int i=0; i <n; ++ i)
        for(int j=0;j <n; ++j ) {
            double cij = C[i*n+j];       // cij = C[i][j]
                for(int k=0; k<n; ++k) {
                    cij += A[i*n+k*] * B[k*n+j];  //cij += A[i][k] * B[k][j]
                C[i*n+j] = cij;              // C[i][j] = cij
```

C版本

结语

- ISAs 支持算术运算
  - 无符号、有符号数据
  - 浮点数据，用来近似实数

- 有限的范围、精度
  - 运算可能导致overflow和underflow