Chapter 3

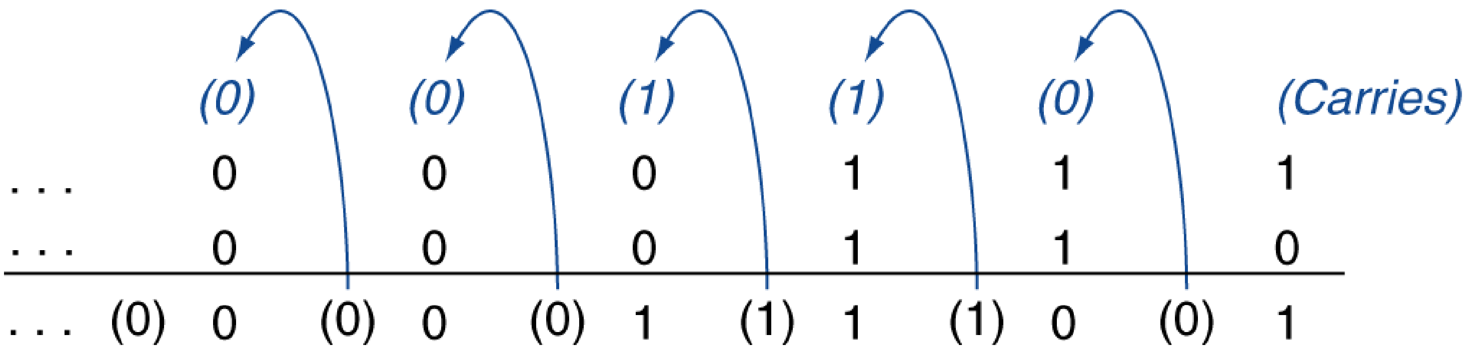
算术运算

算术运算

- 整数运算
  - 加减法
  - 乘除法
  - 溢出处理
- 浮点运算
  - 浮点表示
  - 操作

3.2整数加法

- 例子: 7 + 6



- 结果超出表示范围就会发生溢出(定义)
  - 正数和负数相加，不会溢出
  - 两个正数相加
    - 符号位为1，表示发生溢出
  - 两个负数相加
    - 符号位为0，表示发生溢出

整数减法

- 减法：加上加数的负值
- 例如: 7 – 6 = 7 + (–6)

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

- 结果超出范围就会溢出
  - 相同同符号位相减，不会溢出
  - 一个负数减去一个正数
    - 符号位为0，表示发生溢出
  - 一个正数减去一个负数
    - 符号位为1，表示发生溢出

多媒体算术运算

- 饱和操作
  - 溢出时，结果保持为最大值
  - E. g. ，　音频的剪切，视频中的饱和

3.3乘法

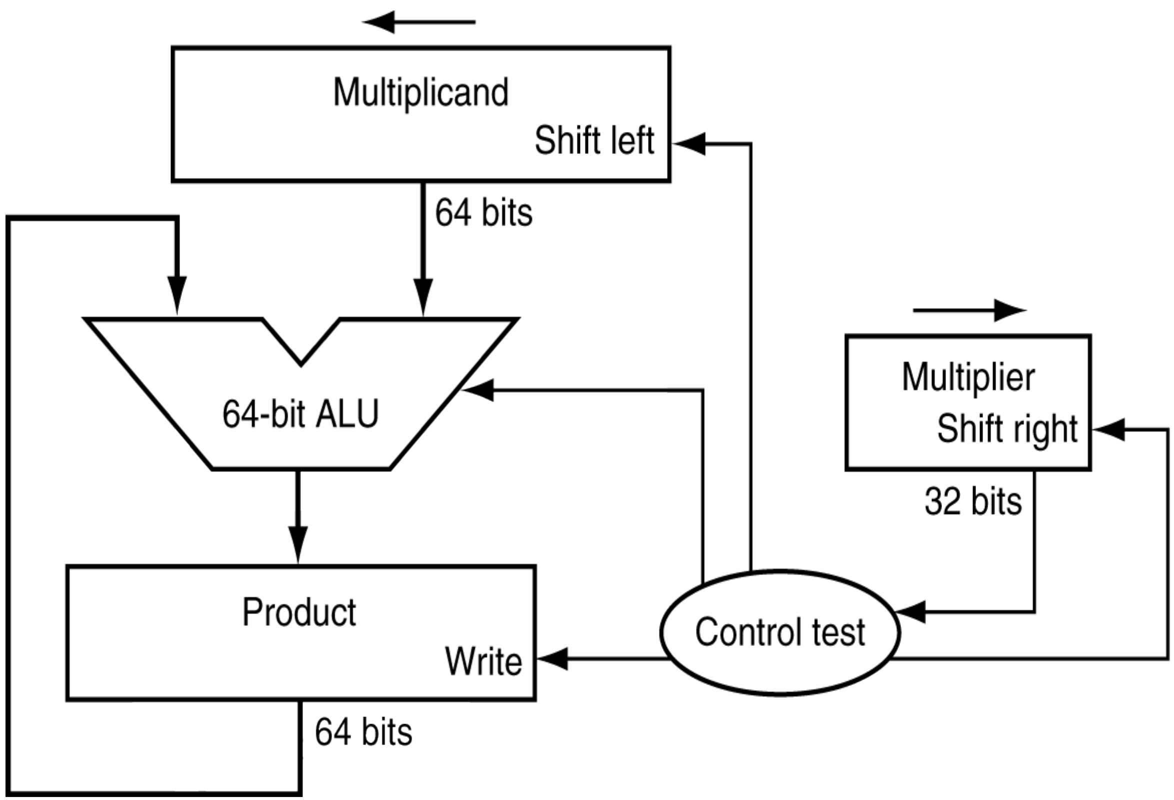
Multiplicand 被乘数

Multiplier乘数

Product 乘积

```
     1000
  ×  1001
  -------
     1000
    00000
   000000
  1000000
  -------
  1001000
```

乘积的位数是乘数和被乘数之和



（1）手算

$$
\begin{array}{r}
0.1101 \\
\times\, 0.1011 \\
\hline
1101 \\
1101 \\
0000 \\
+\quad 1101 \\
\hline
0.10001111
\end{array}
$$

部分积

上符号：1.10001111

问题：1）加数增多（由乘数位数决定）。

2）加数的位数增多（与被乘数、乘数位数有关）。

改进：将一次相加改为分步累加。

（2）分步乘法

每次将一位乘数所对应的部分积与原部分积的累加和相加，并移位。

设置寄存器：

A：存放**部分积累加和**、**乘积高位**

B：存放**被乘数**

C：存放**乘数**、**乘积低位**

设置初值：

$$A = 00.0000$$

$$B = |X| = 00.1101$$

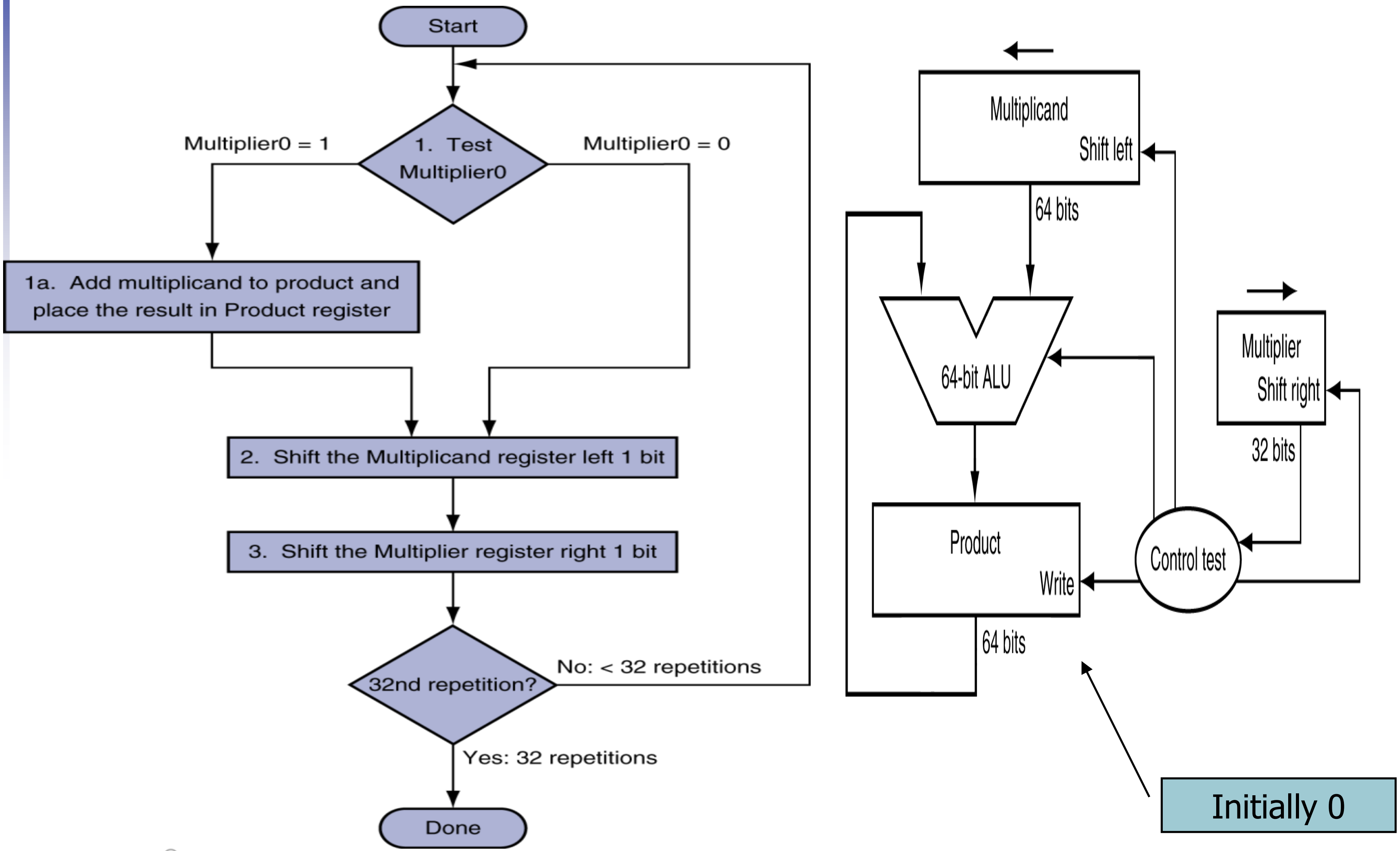
$$C = |Y| = \ .1011$$

步数	条件	操作	A	C C_n
			00.0000	.1011
1)	C_n=1	+B	+ 00.1101	
			00.1101	
		→	00.0110	1.101
2)	C_n=1	+B	+ 00.1101	
			01.0011	
		→	00.1001	11.10
3)	C_n=0	+0	+ 00.0000	
			00.1001	
		→	00.0100	111.1
4)	C_n=1	+B	+ 00.1101	
			01.0001	
		→	00.1000	1111

$$
\begin{array}{r}
0.1101 \quad \text{—B} \\
\times\, 0.1011 \quad \text{—C} \\
\hline
1101 \\
1101 \\
0000 \\
+\,1101 \\
\hline
0.10001111
\end{array}
$$

$X_原 \times Y_原 = 1.10001111$

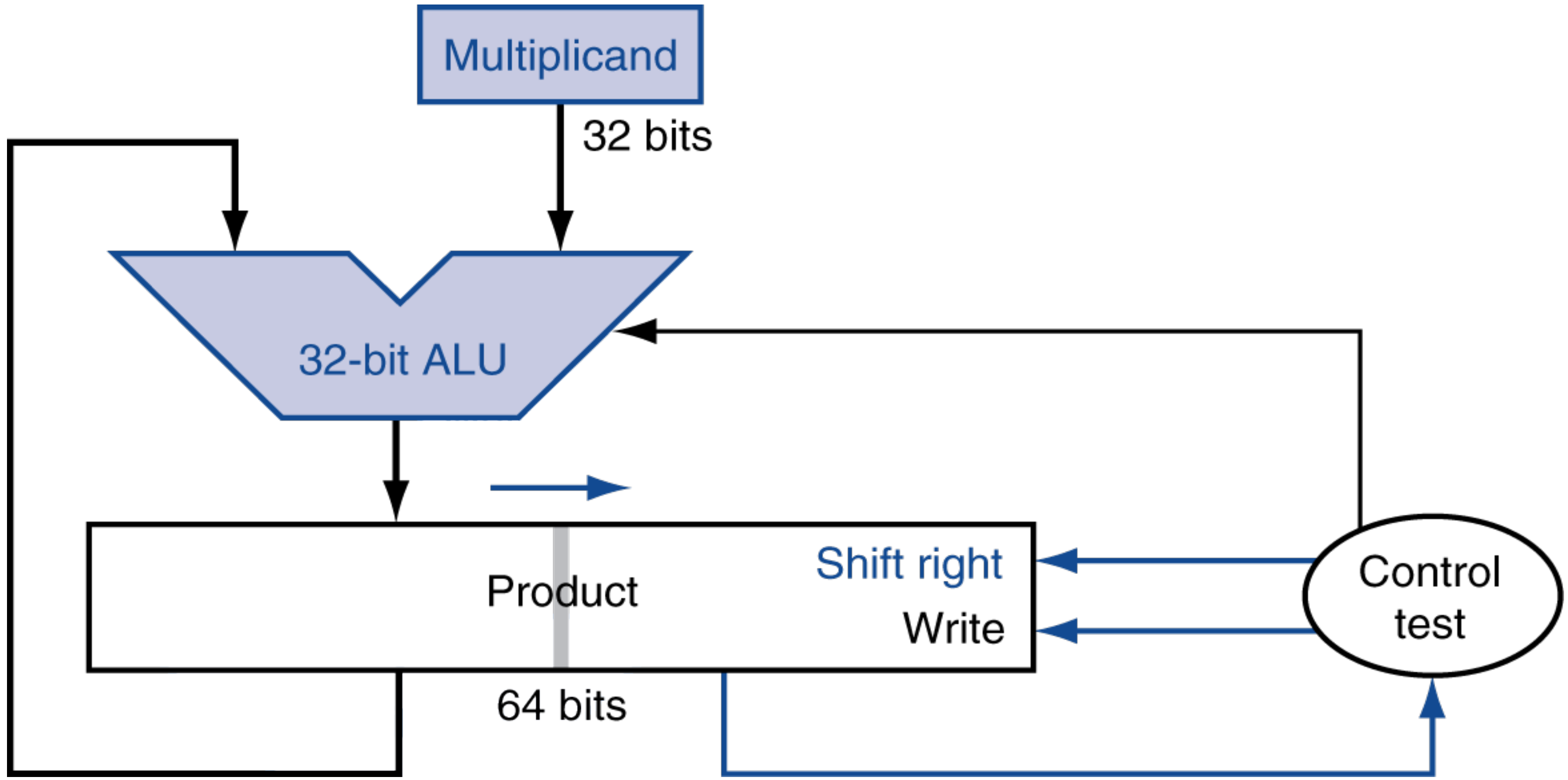
乘法硬件



Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done
Multiplicand
Shift left
64 bits
64-bit ALU
Multiplier
Shift right
32 bits
Product
Write
Control test
64 bits
Initially 0

改进后的乘法器

- 操作并行: add/shift

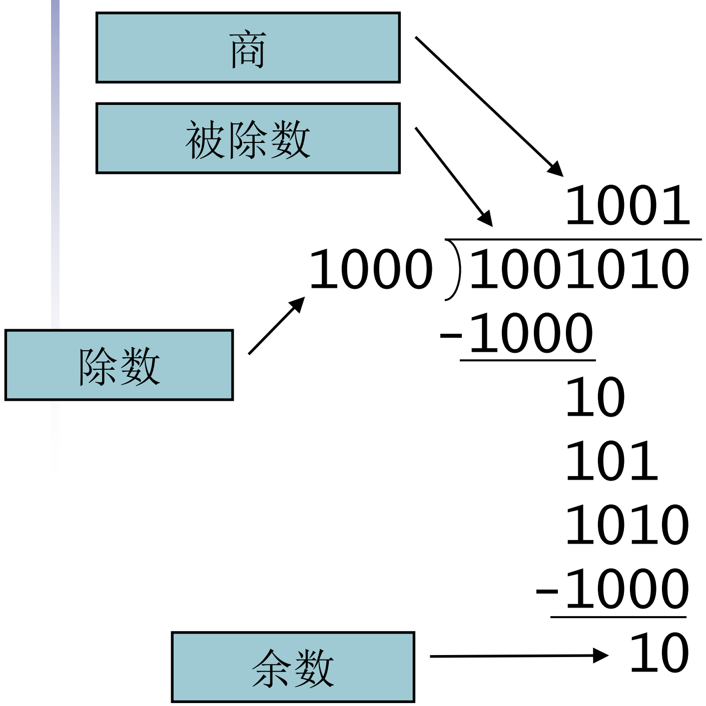


- 一个时钟完成一次积:频率较低

圆圈圈起来的是下一步要检测的位

迭代	步骤	乘数	被乘数	积product
0	初始值	0011	0000 0010	0000 0000
1	1:1➔积=积+被乘数	0011	0000 0010	0000 0010
	2：左移被乘数	0011	0000 0100	0000 0010
	3：右移乘数	0001	0000 0100	0000 0010
2	1:1➔积=积+被乘数	0001	0000 0100	**0000 0110**
	2：左移被乘数	0001	0000 1000	0000 0110
	3：右移乘数	0000	0000 1000	0000 0110
3	1：无操作	0000	0000 1000	0000 0110
	2：左移被乘数	0000	0001 0000	0000 0110
	3：右移乘数	0000	0001 0000	0000 0110
4	1：无操作	0000	0001 0000	0000 0110
	2：左移被乘数	0000	0010 0000	0000 0110
	3：右移乘数	0000	0010 0000	**0000 0110**

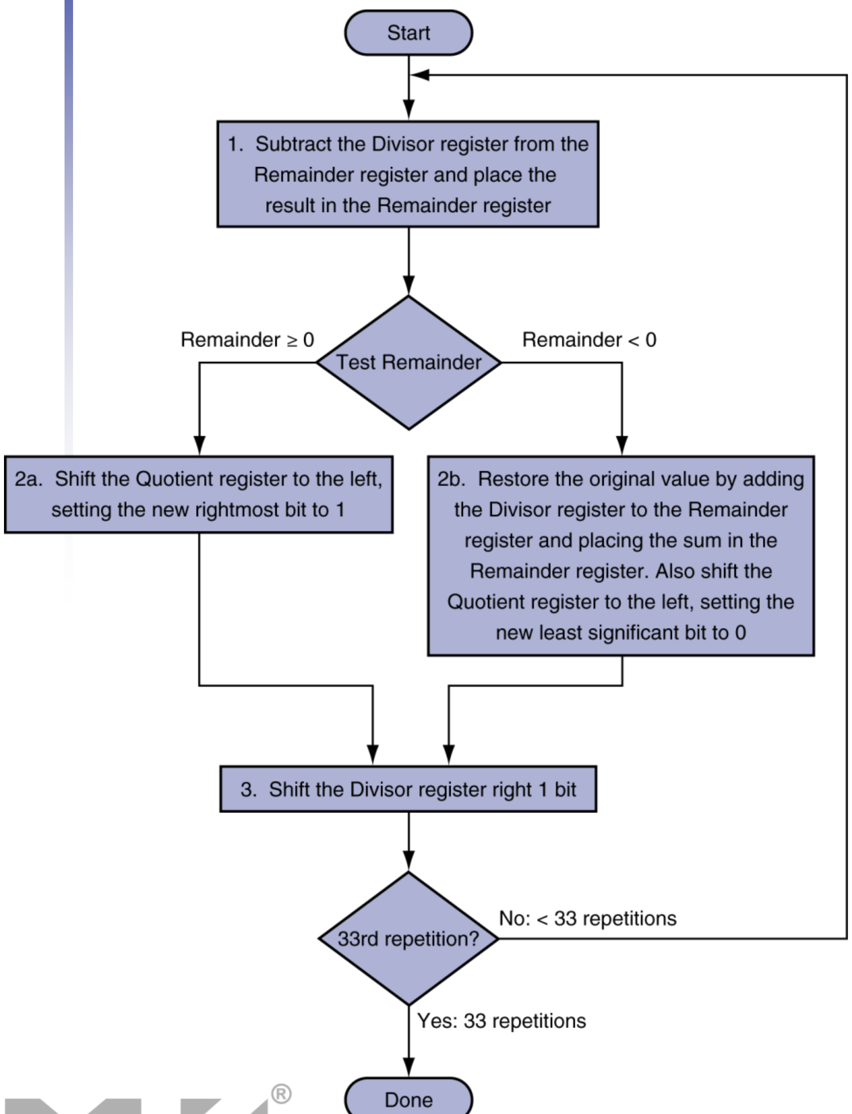
3.4除法



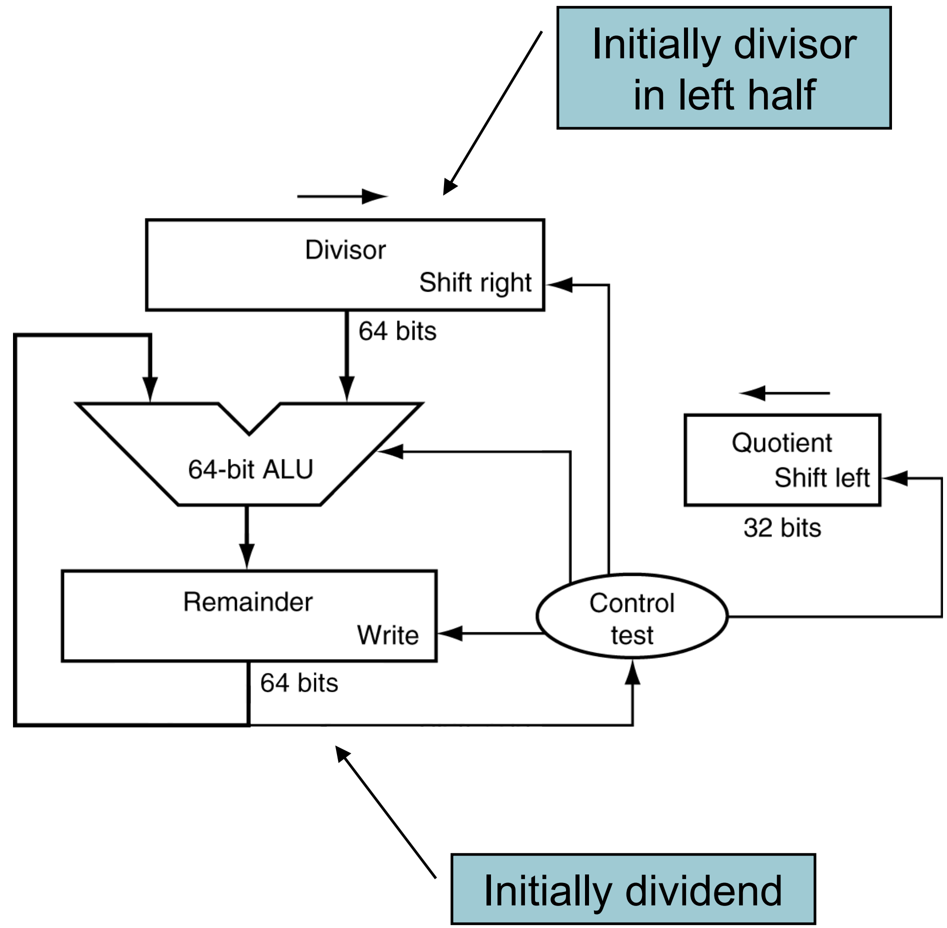
n-bit 操作产生 *n*-bit
商和余数

- 除数判0，（无效操作）
- 除法步骤
  - 除数 ≤ 被除数
    - 商添加1，执行减法
  - 否则
    - 商添加0，从被除数中提取下一个bit
- 恢复余数法
  - 直接执行减法，结果小于0后再把除数加回去
- 带符号位的除法
  - 用绝对值进行除
  - 根据需要调整商和余数的符号

除法器的硬件

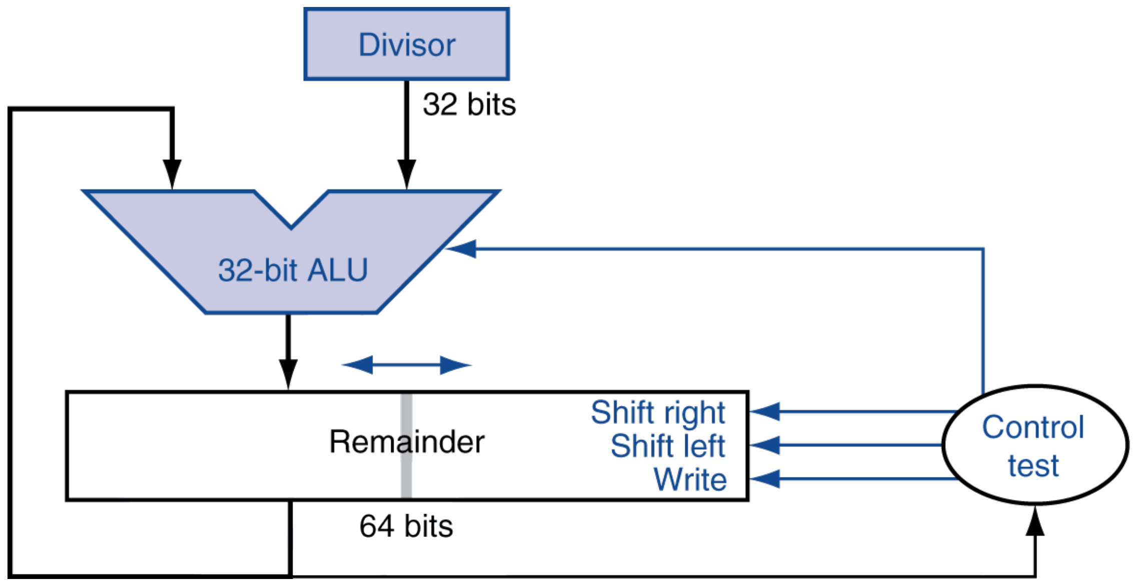


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done



Initially divisor in left half
Divisor
Shift right
64 bits
64-bit ALU
Quotient
Shift left
32 bits
Remainder
Write
Control test
64 bits
Initially dividend

改进后的除法器



- 一个时钟周期做一次减法
- 和乘法器类似
    - 乘法和除法可以共用同样的硬件

3.5浮点运算

- 表达非整形的数
  - 可以表达很小和很大的数
- 和科学计数法类似
  - -2.34×10^{56}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$

规格化

非规格化

- 二进制表示
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- C语言中的类型：float和double

3.5.1浮点表示：IEEE754

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

S	阶码	尾数

$$x = (-1)^S \times (1 + 尾数) \times 2^{(阶码 - 偏移)}$$

- S：符号位(0 ⇒ 非负数，1 ⇒ 负数)
- 有效位的规格化：1.0 ≤ |有效位| < 2.0
  - 数前总有一个前导的1，此外作为隐含位可以不表示。
  - 有效位是“1. 尾数”
- 阶码：移码表示：真实的指数 + 偏移
  - Ensures exponent is unsigned
  - Single：偏移= 127；Double：偏移= 1203

单精度浮点数的范围

- 阶码00000000和11111111特殊用途
- 最小值
  - 阶码：00000001
    ⇒ 指数值 = 1 – 127 = –126
  - 尾数：000…00 ⇒ 有效位= 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- 最大值
  - 阶码：11111110
    ⇒ 指数值= 254 – 127 = +127
  - 尾数：111…11 ⇒ 有效位≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

IEEE754标准

- 对于阶码为0或255的情况，而IEEE754标准有特别的规定：阶码是1—254，如果 E 是0 并且 M 是0，则这个数的真值为±0（正负号和数符位有关） 如果 E = 255 并且 M 是0，则这个数的真值为±∞（同样和符号位有关） 如果 E = 255 并且 M 不是0，则这不是一个数（NaN）。
- 短浮点数和长浮点数（不含临时浮点数）的存储在尾数中隐含存储着一个1，因此在计算尾数的真值时比一般形式要多一个整数1。对于阶码E的存储形式因为是127的偏移，所以在计算其移码时与人们熟悉的128偏移不一样，正数的值比用128偏移求得的少1，负数的值多1，为避免计算错误，方便理解，常将E当成二进制真值进行存储。例如：将数值-0.5按IEEE754单精度格式存储，先将-0.5换成二进制并写成标准形式：-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

浮点数精度

- 相对精度（分辨率）
  - 尾数的每一部分都有意义
  - 单精度：大约2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 7$ decimal digits of precision
  - 双精度：大约2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

浮点数的示例

- 如何表示–0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - 尾数= $1000...00_2$
  - 阶码= –1 + 偏移
    - 单精度：$-1 + 127 = 126 = 01111110_2$
    - 双精度：$-1 + 1023 = 1022 = 01111111110_2$
- 单精度：1 $\underbrace{01111110}_{8位}$ $\underbrace{1000...00}_{23位}$
- 8位 23位
- 双精度：1 01111111110 1000...00

浮点数的示例

- 计算下列浮点数的真值
  11000000101000…00
  - S = 1
  - 尾数= $01000\ldots00_2$
  - 阶码 = 10000001_2 = 129
- $x = (-1)^1 \times (1 + .01_2) \times 2^{(129 - 127)}$
  $= (-1) \times 1.25 \times 2^2$
  $= -5.0$

浮点数加法

- 以一个4位的十进制数为例：
  - $9.999 \times 10^{1} + \underline{1.610} \times 10^{-1}$
- 1. 对阶
  - 把小阶的值调整到和大阶一致(−1调整到1)
  - $9.999 \times 10^{1} + \underline{0.016} \times 10^{1}$
- 2. 尾数相加
  - $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$
- 3. 结果规格化& 检查是否溢出
  - $1.00\underline{15} \times 10^{2}$
- 4. 进行必要的舍入处理
  - $1.00\underline{2} \times 10^{2}$

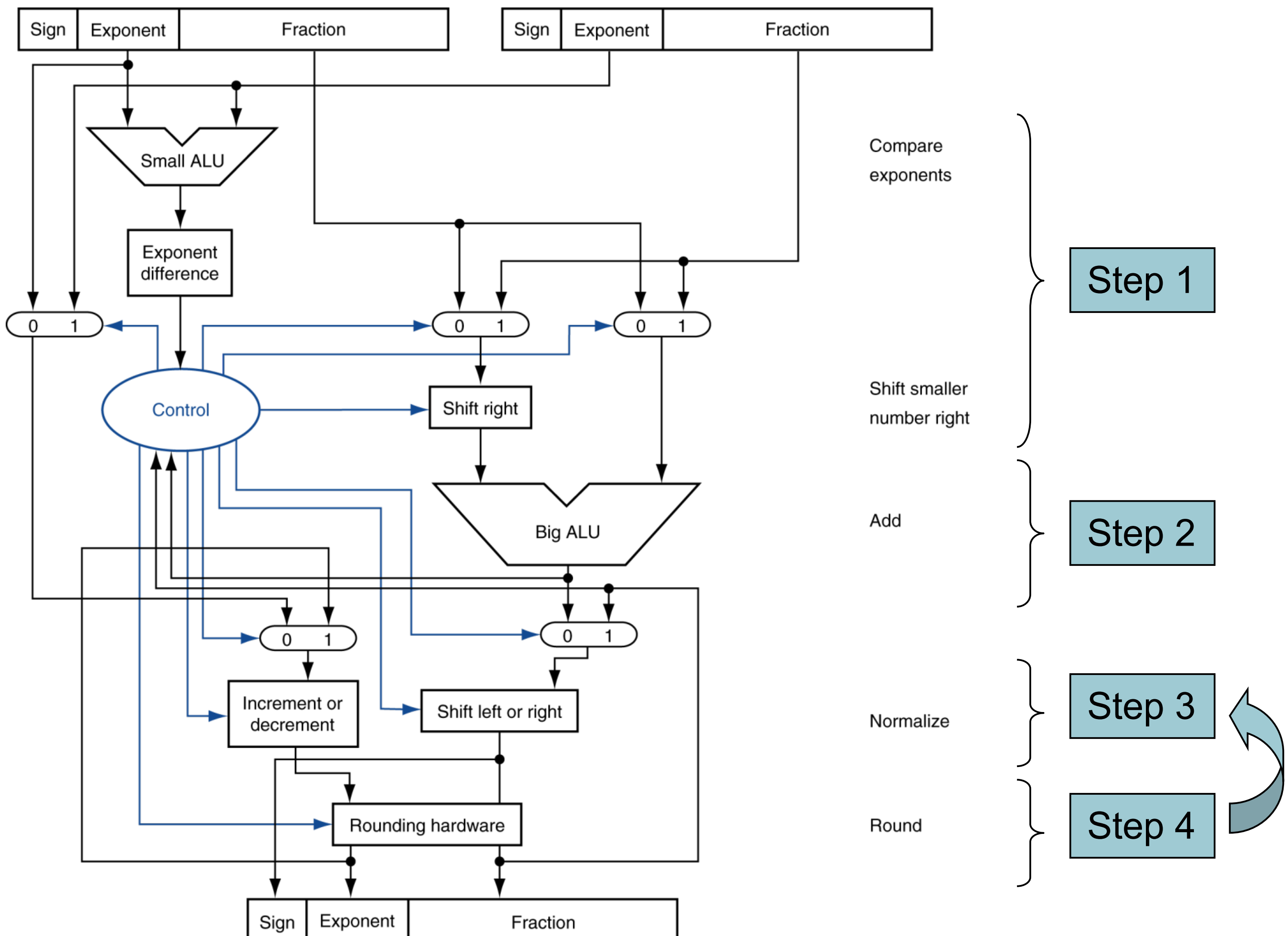
浮点数加法

- 现在计算一个4位的二进制数
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. 对接
  - 小阶对大阶
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. 尾数相加
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- 3. 规格化 & 检查溢出
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. 进行必要的舍入
  - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

浮点加法的硬件实现

- 比整数复杂很多
- 如果在一个时钟周期内完成，就会要求时钟周期非常的长
  - 比整数运算费时
  - 较慢的时钟会对所有的指令产生影响
- 浮点加法器通常需要花费几个时钟周期
  - 可以被流水化

浮点加法


Sign
Exponent
Fraction
Sign
Exponent
Fraction
Small ALU
Exponent difference
0 1
Control
Shift right
Big ALU
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

浮点运算硬件

- 浮点乘法和加法的硬件复杂度类似
  - 有效位上进行乘法而不是加法
- 浮点运算通常需要的操作是
  - 加法，减法，乘法，除法，求倒数，平方根
  - 浮点数和整数间的转换
- 通常需要多个时钟周期
  - 很容易用流水实现

MIPS中的浮点指令

- 浮点数使用协处理器
  - 通过ISA相连的附属处理器
- 独立的浮点寄存器
  - 32个单精度：$f0，$f1，… $f31
  - 配对为双精度：$f0/$f1，$f2/$f3，…
    - Release 2 of MIPs ISA supports 32 × 64-bit FP reg's
- 浮点指令只操作浮点寄存器
  - 程序通常不会在浮点寄存器上进行整数操作，或在整数寄存器上进行浮点操作
  - 因此可以提供更多的寄存器，而不影响指令的长度
- 浮点数读取、存储指令
  - lwc1，ldc1，swc1，sdc1
    - e.g.，ldc1 $f8，32($sp)

MIPS中的浮点指令

- 单精度
  - add.s, sub.s, mul.s, div.s
    - e.g., add.s $f0, $f1, $f6
- 双精度
  - add.d, sub.d, mul.d, div.d
    - e.g., mul.d $f4, $f4, $f6
- 比较
  - c.*xx*.s, c.*xx*.d (*xx* is eq, lt, le, ...)
  - Sets or clears FP condition-code bit
    - e.g. c.lt.s $f3, $f4
- 分支
  - bc1t, bc1f
    - e.g., bc1t TargetLabel

内在含义

The BIG Picture

- Bits没有固有的含义
  - 由在上面操作的指令决定如何使用
- 计算机中数的表示
  - 有限的范围和精度
  - 在编程时需要处理这些细节

子字并行

- 视频和音频的应用可以采用短向量的并行操作
  - Example: 128-bit adder:
    - Sixteen 8-bit adds
    - Eight 16-bit adds
    - Four 32-bit adds
- 也被称为数据级并行，向量或SIMD（单指令多数据）

结合律

- 并行程序可能进行操作顺序的调整
  - 依赖特定的顺序可能会导致错误的结果

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- 因此需要验证结果的可靠性（在计算机中，浮点数的加法不满足结合律）

x86 浮点指令结构

- 8087浮点协处理器上扩展
  - 8 × 80-bit 扩展精度的寄存器
  - 构成一个向下推的栈结构
  - 可以按照下标访问 TOS: ST(0), ST(1), ...
- 在内存中的浮点数也是32-bit 或 64-bit
  - 在load/store自动转换
  - 整数转换也能自动完成
- 代码生成和优化很困难
  - 浮点性能较差

x86 浮点指令

Data transfer	Arithmetic	Compare	Transcendental
FILD mem/ST(i) FISTP mem/ST(i) FLDPI FLD1 FLDZ	FIADDP mem/ST(i) FISUBRP mem/ST(i) FIMULP mem/ST(i) FIDIVRP mem/ST(i) FSQRT FABS FRNDINT	FICOMP FIUCOMP FSTSW AX/mem	FPATAN F2XMI FCOS FPTAN FPREM FPSIN FYL2X

- 可选的组合
  - I：使用整数
  - P：计算完毕后弹栈
  - R：源操作数和目的操作数反转
  - 并不是所有的组合都可用。

Streaming SIMD Extension 2 (SSE2)

- 增加4 × 128-bit 寄存器
  - 扩展为8个 AMD64/EM64T
- 可以在浮点操作中使用
  - 2 × 64-bit 双精度
  - 4 × 32-bit 单精度
  - 指令并行操作
    - Single-Instruction Multiple-Data（单指令多数据）

Matrix Multiply

- x86 assembly code:

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

Matrix Multiply

- Optimized C code:

```
#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 */
  }
}
```

Matrix Multiply

- Optimized x86 assembly code:

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

右移和除法

- 左移i位和乘以2^i*是同样的结果*
- 右移是否和除2^i*相同*?
  - 只对无符号数
- 有符号数
  - 算数右移：高位需要补入符号位
  - e.g.，–5 / 4
    - 11111011_2 >> 2 = 11111110_2 = –2
    - 低位直接舍弃
  - c.f. 11111011_2 >>> 2 = 00111110_2 = +62

浮点数精确度?

- 在科学计算中很重要
  - 当在某些日常生活中?
    - “我的余额差了0.0002¢!” ☹
- Pentium FDIV指令的bug
  - 用户还是希望能精准计算
  - See Colwell, *The Pentium Chronicles*

总结

- 指令支持的运算
  - 有符号和无符号的整数
  - 和实数类似的浮点数
- 受限的范围和精度
  - 计算可能会溢出
- MIPS指令结构
  - 核心指令：54最常用
    - 覆盖100% SPECINT，覆盖97%SPECFP
  - 其他指令：很少使用