第三章

计算机的算术运算

计算机的算术运算

- 整数运算
  - 加减法
  - 乘除法
  - 溢出处理
- 浮点实数运算
  - 浮点表示 和 运算

整数加减法

- 例子:

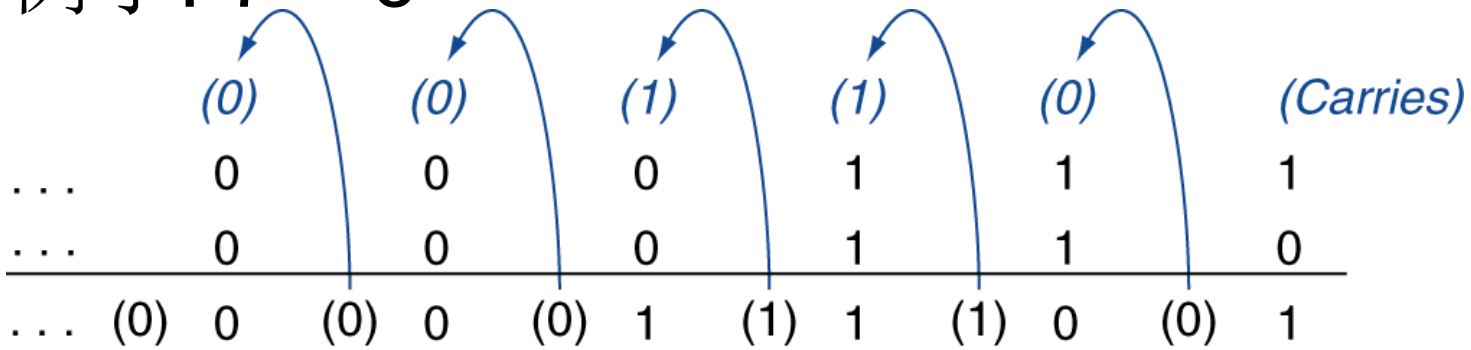
8-bit 数据

127+ 6

- 针对有符号数
- 无符号数一般可以忽略
- 结果超出表达范围，就会发生溢出

整数加减法

- 加法
- 例子: 7 + 6



- 结果超出表达范围，就会发生溢出
  - 正负操作数相加, 不溢出
  - 两个正操作数相加
    - 符号位为1，表示发生溢出
  - 两个负操作数相加
    - 符号位为0，表示发生溢出

整数加减法

- **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

- 结果超出表达范围，就会发生溢出
  - 两个相同符号位的操作数相减，不溢出
  - 负操作数减去正操作数
    - 符号位为0，表示发生溢出
  - 正操作数减去负操作数
    - 符号位为1，表示发生溢出

整数加减法

操作	A	B	溢出条件
A+B	≥0	≥0	< 0
A+B	<0	<0	≥0
A-B	≥0	<0	< 0
A-B	< 0	≥0	≥0

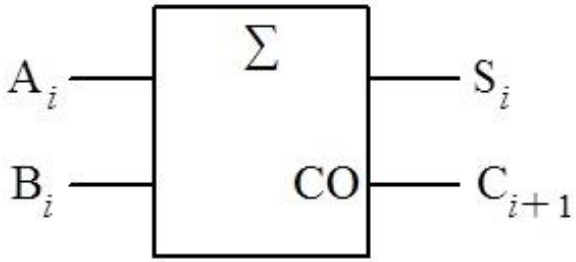
溢出的处理

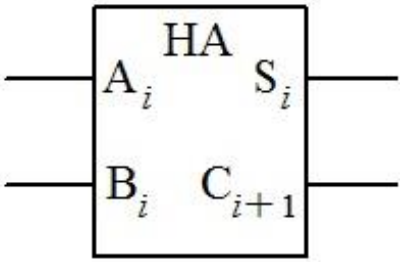
- 有些高级语言(例如, C) 忽略溢出部分
  - 使用 MIPS `addu`, `addiu`, `subu` 指令
- 其它语言 (例如, Ada, Fortran) 需要引发异常
  - 使用 MIPS `add`, `addi`, `sub` 指令
  - 溢出发生时, 调用异常处理程序
    - 保存PC指针到EPC(exception program counter) 寄存器
    - 跳转到预定义的异常处理地址
    - `mfc0` (move from coprocessor reg) 指令可以取回 EPC 值， 恢复PC指针。

整数加减法

- 算术逻辑单元 ALU
  - 半加器

A_i	B_i	C_{i+1}	S_i
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

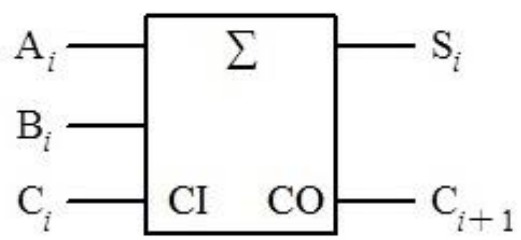


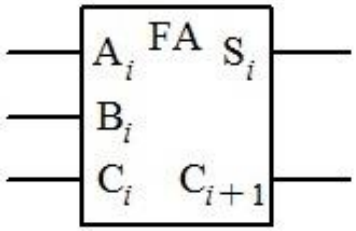


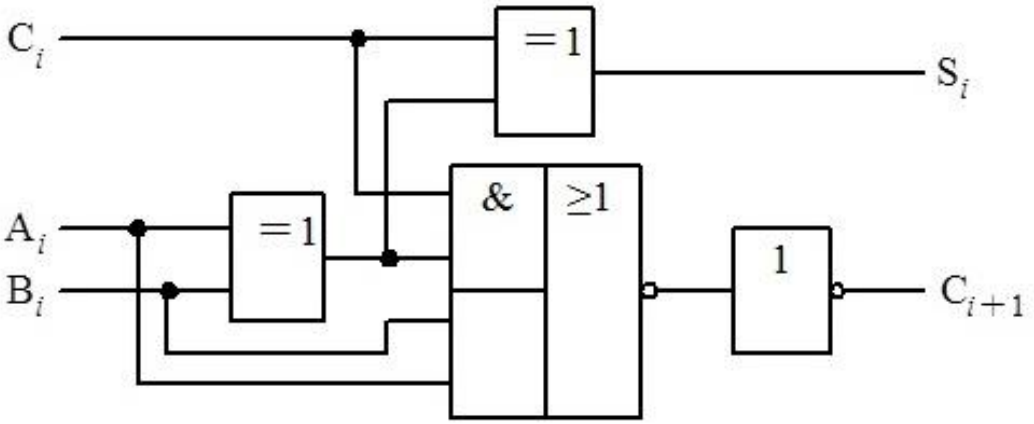
整数加减法

- 算术逻辑单元 ALU
  - 半加器和全加器

A_i	B_i	C_i	C_{i+1}	S_i
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

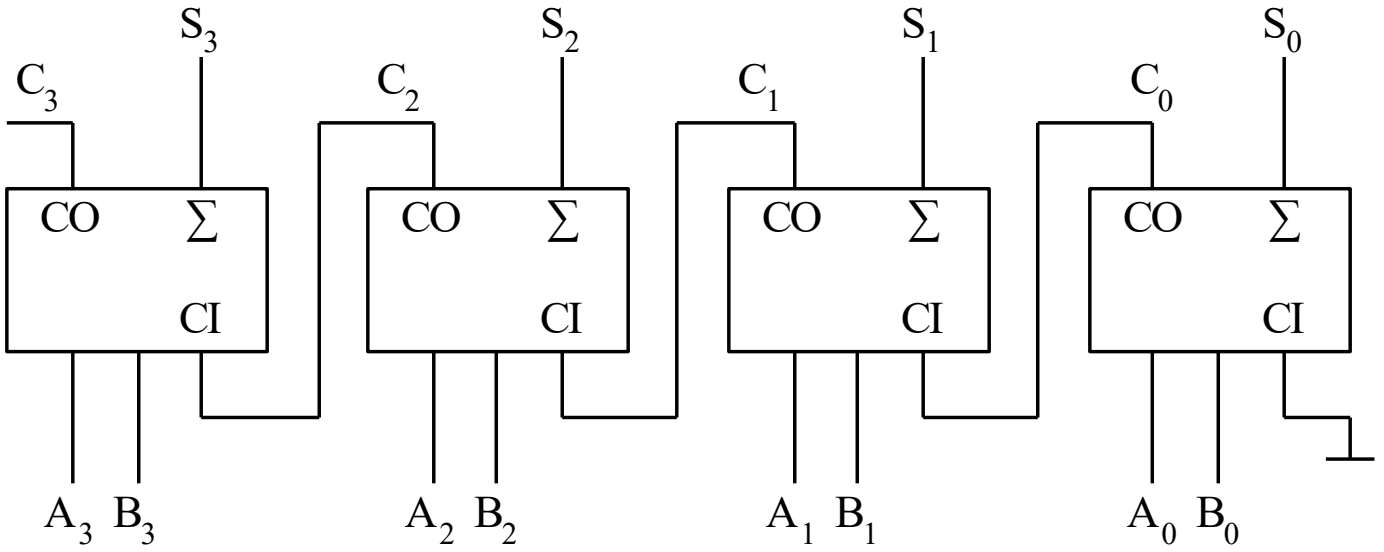






整数加减法

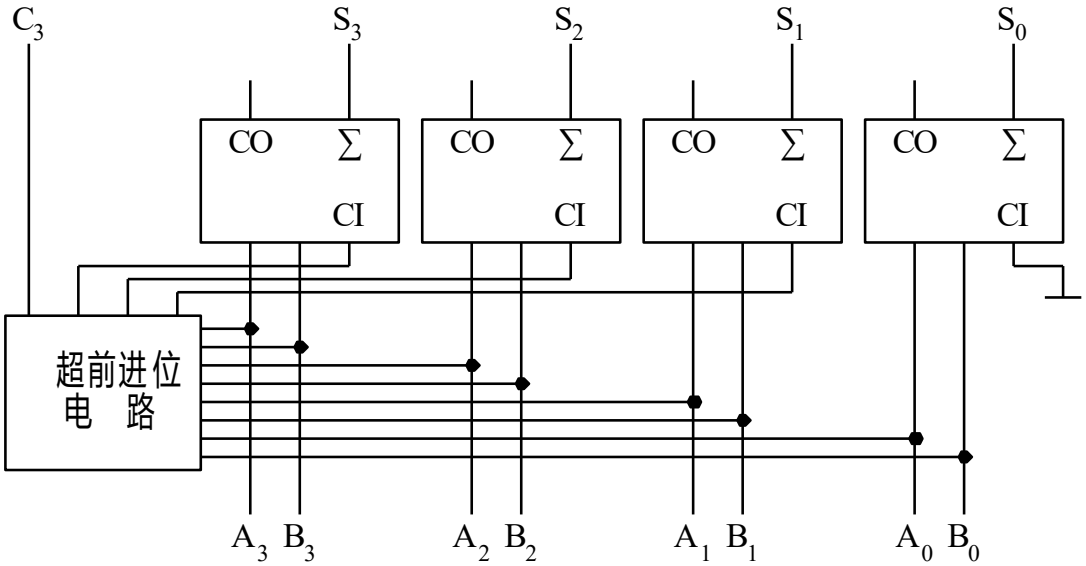
- 算术逻辑单元 ALU
  - 多位全加器



运算时间?

整数加减法

- 算术逻辑单元 ALU
  - 超前进位加法器



$C_0=A_0B_0$

$C_i=A_iB_i+(A_i+B_i)C_{i-1}, i\neq 0$

- 令$G_i=A_iB_i, P_i=A_i+B_i$,利用递归关系可以得到:

$C_i=G_i+P_iC_{i-1}=G_i+P_i（G_{i-1}+P_{i-1}C_{i-2}）=G_i+P_iG_{i-1}+P_iP_{i-1}C_{i-2}$

$=G_i+P_iG_{i-1}+P_iP_{i-1}G_{i-2}+...+P_iP_{i-1}...P_2G_1+P_iP_{i-1}...P_2P_1C_0$

多媒体算术运算

- 图像和媒体处理操作一般针对8-bit和16-bit的矢量数据
  - 使用64位加法器, 分区进位链
    - 8×8-bit, 4×16-bit, or 2×32-bit 矢量
  - SIMD (单指令, 多数据)
- 饱和操作
  - 溢出时,结果为可表示的最大值
    - c.f. 2s-complement 模运算
  - 例如, 音频的剪切，视频中的饱和

乘法

- Start with long-multiplication approach

被乘数

multiplicand

multiplier

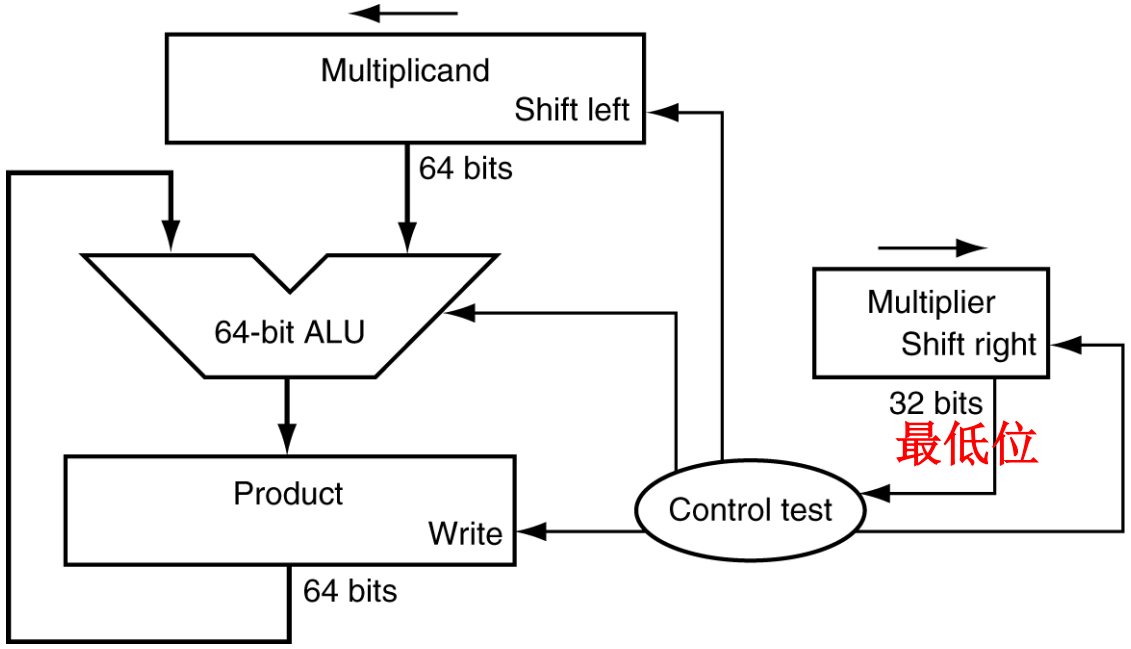
乘数

乘积

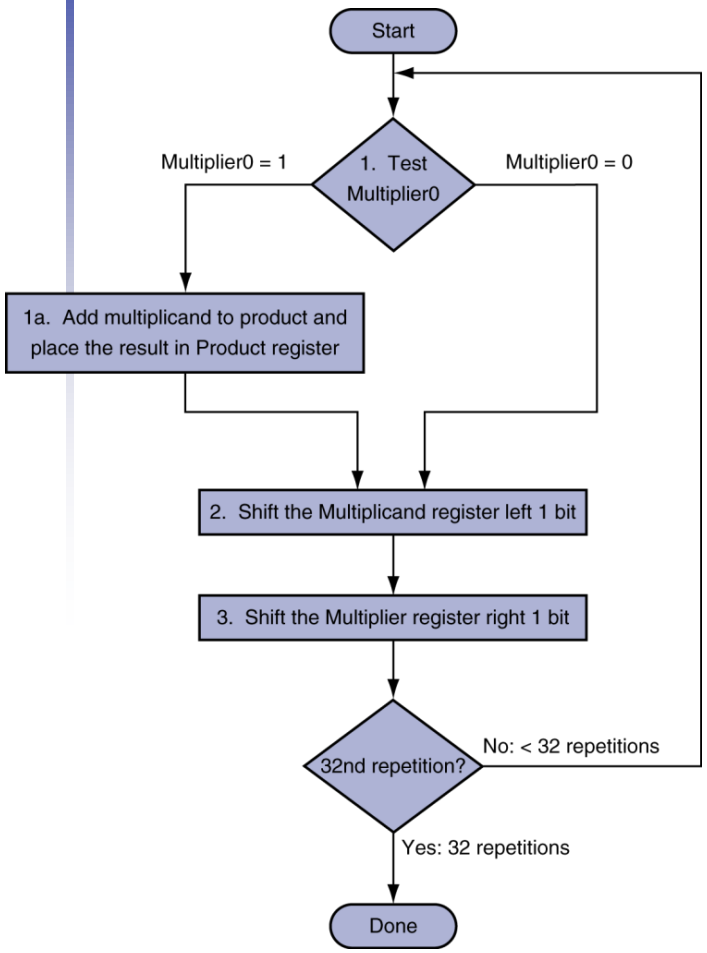
product

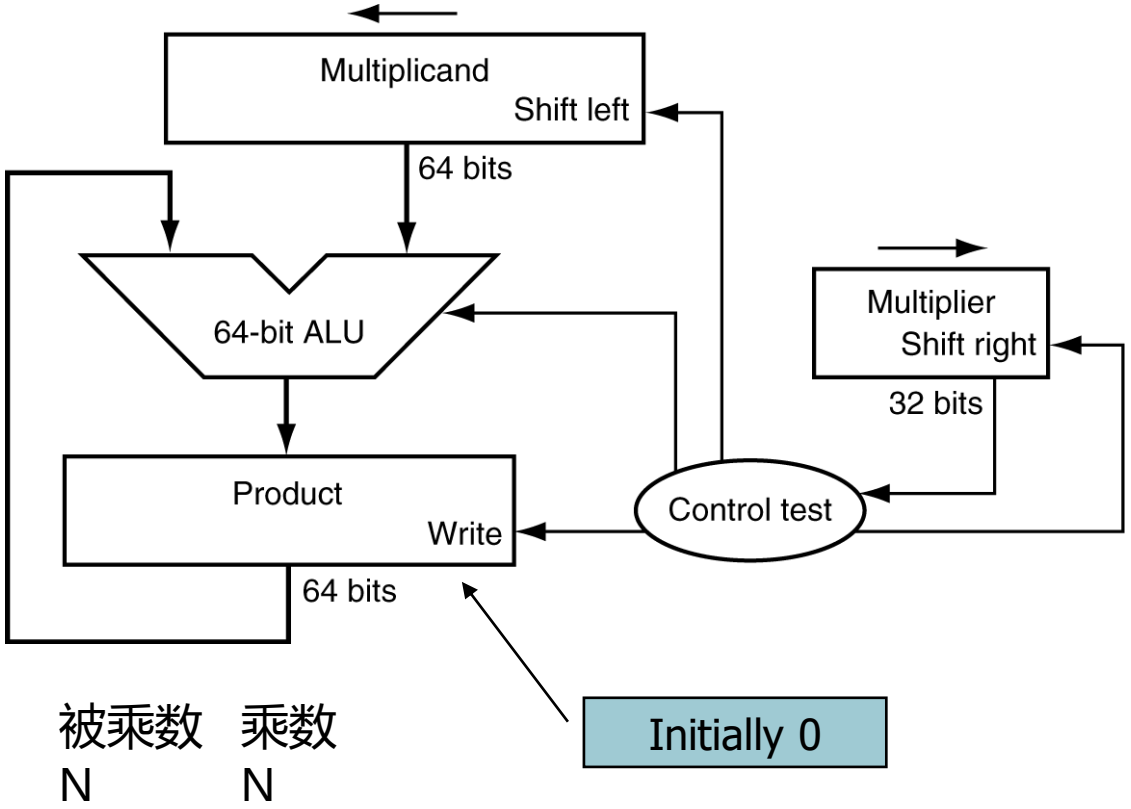
```
  1000
×  1001
  1000
  0000
  0000
 1000
1001000
```

Length of product is the sum of operand lengths



乘法硬件





被乘数 乘数
N N

- 算法优化：一个时钟完成一次积
  - 乘积频率低

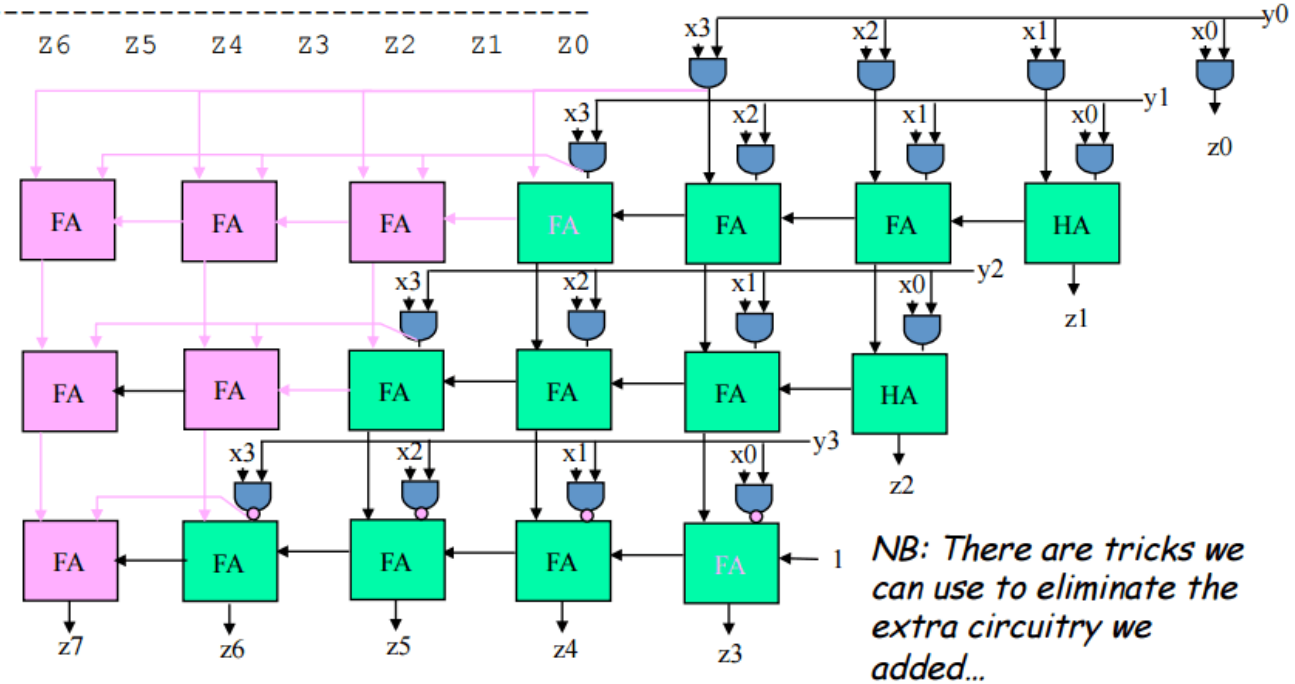
7 X 3 ： 0111 X 011

序号	积	乘数	被乘数	注释
	0000 0000	0011	0000 0111	装载
1	0000 0111	0011	0000 0111	积’=积+被乘数
	0000 0111	0001	0000 1110	被乘数左移1/乘数右移1
2	0001 0101	0001	0000 1110	积’=积+被乘数
	0001 0101	0000	0001 1100	被乘数左移1/乘数右移1
3	0001 0101	0000	0011 1000	被乘数左移1/乘数右移1
4	0001 0101	0000	0111 0000	被乘数左移1/乘数右移1

快速乘法器-（直接法）

Combinational Multiplier (signed!)

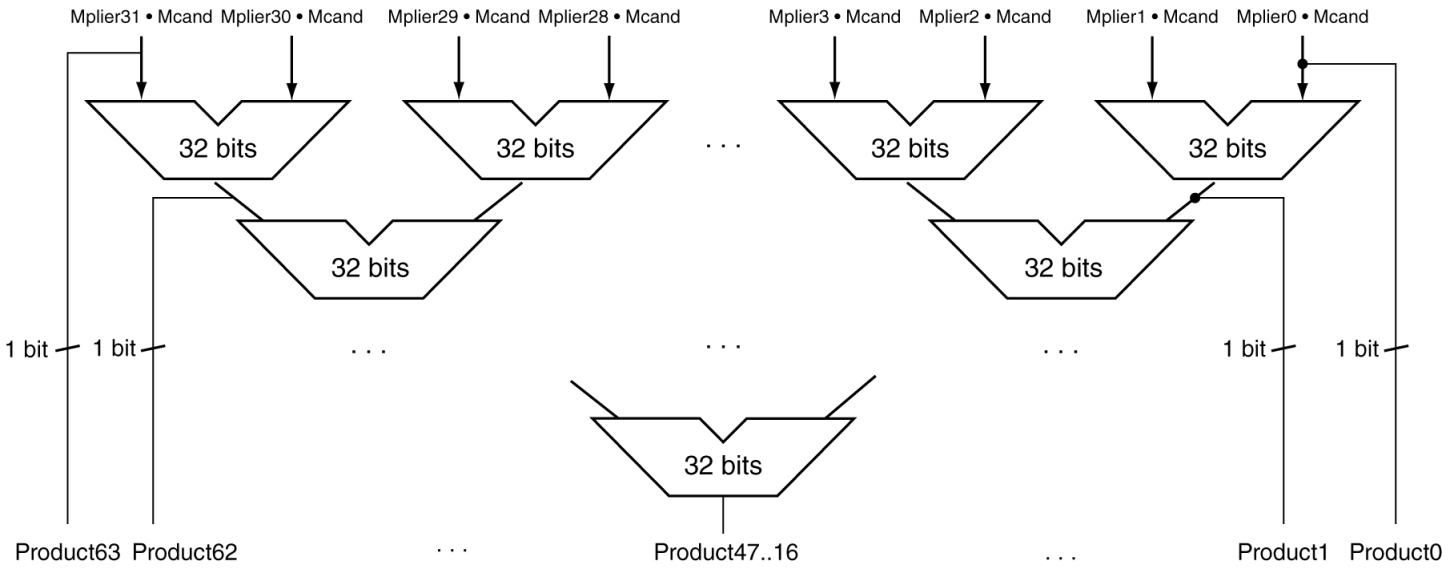
```
                    X3    X2    X1    X0
                *   Y3    Y2    Y1    Y0
                ------------------------
  X3Y0  X3Y0  X3Y0  X3Y0  X3Y0  X2Y0  X1Y0  X0Y0
+ X3Y1  X3Y1  X3Y1  X3Y1  X2Y1  X1Y1  X0Y1
+ X3Y2  X3Y2  X3Y2  X2Y2  X1Y2  X0Y2
- X3Y3  X3Y3  X2Y3  X1Y3  X0Y3
----------------------------------------
    Z7    Z6    Z5    Z4    Z3    Z2    Z1    Z0
```



NB: There are tricks we can use to eliminate the extra circuitry we added…

快速乘法器($\log_2$x高)

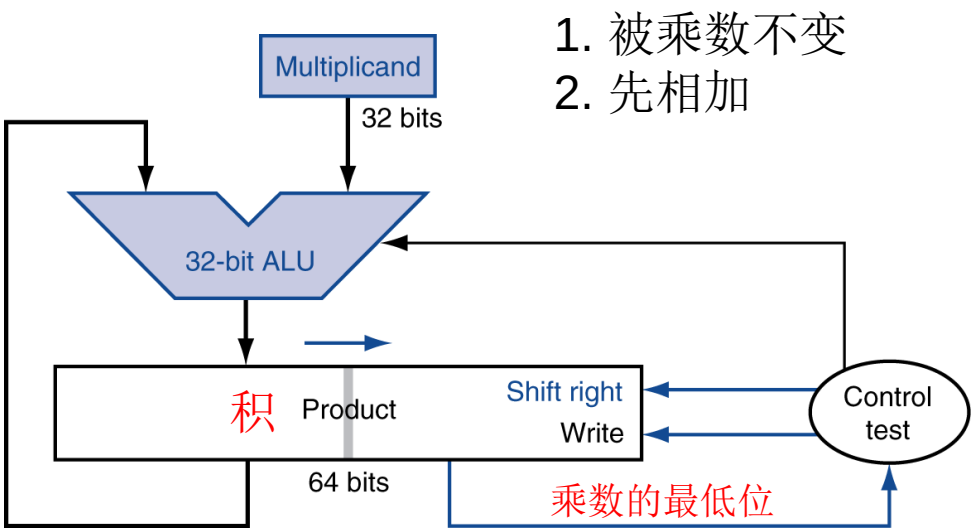
- 使用多个加法器
  - 成本/效率的折中

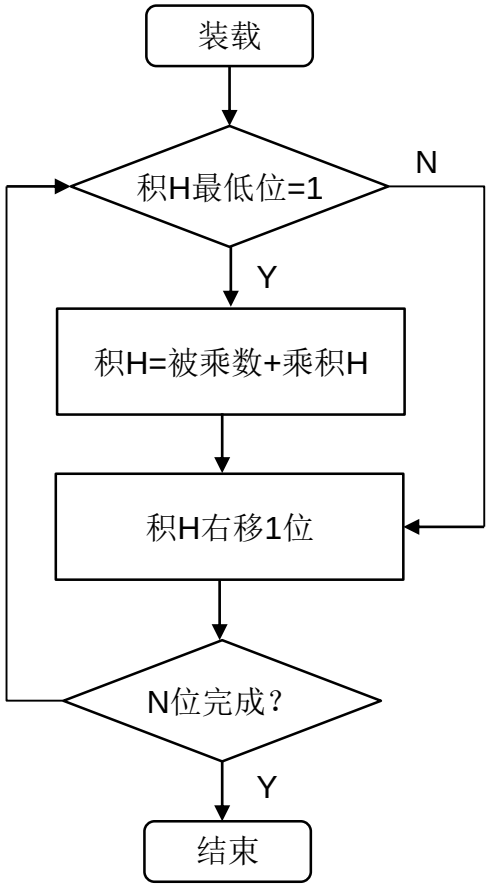


- 可以流水执行
  - 数个乘法并行执行

优化后的乘法器

- 并行执行步骤: add/shift





- 硬件优化=> 32-bit ALU

7 X 3 ： 0111 X 0011

序号	积	乘数	被乘数	注释
	0000 0000	0011	0111	
	0000 0011			装载
1	0111 0011			积’=积H+被乘数
	0011 1001			积’=积右移1
2	1010 1001			积’=积H+被乘数
	0101 0100			积’=积右移1
3	0010 1010			积’=积右移1
4	0001 0101			积’=积右移1

MIPS Multiplication

- 两个32位寄存器保存乘积
  - HI: 高 32 位
  - LO:低 32 位
- 指令
  - `mult rs, rt  /  multu rs, rt`
    - 将64-bit 乘积存到 HI/LO
  - `mfhi rd  /  mflo rd`
    - 传送HI/LO 到 rd寄存器
    - 可以检查HI 的值来判断乘积是否超过32 bits
  - `mul rd, rs, rt`
    - 乘积的低32位存入rd寄存器

优化的乘法器

7 X 3 ： 0011 X 0111

序号	积	乘数	被乘数	注释
	0000 0000	0111	0011	
	0000 0111			装载
1	0011 0111			积’=积H+被乘数
	0001 1011			积’=积右移1
2	0100 1011			积’=积H+被乘数
	0010 0101			积’=积右移1
3	0101 0101			积’=积H+被乘数
	0010 1010			积’=积右移1
4	0001 0101			积’=积右移1

优化的乘法器

- 并行乘法器—BOOTH算法
- 乘数中1的个数决定了加法的次数
- 长1的处理：

0111 1110 = 1000 0000 – 0000 0010

0111 1111 = 1000 0000 – 0000 0001

Y(n)	Y(n+1)	操作
0	0	+0，右移
0	1	+(x)原，右移
1	0	+(-x)补，右移
1	1	+0，右移

右移需要保留符号位不变

优化后的乘法器

- 并行乘法器—BOOTH算法

7 X 3：0111 X 0011，(-7)补 = 1001

序号	P	注释
	0000 0000 0	原始乘积
	0000 0011 0	装载乘数，移位N=0
1	1001 0011 0	10 -> -(被乘数) -> +(被乘数)补
	1100 1001 1	右移，移位N=1
2	1110 0100 1	11 -> 直接右移，移位N=2
3	0101 0100 1	01 -> +(被乘数)原
	0010 1010 0	右移，移位N=3
4	**0001 0101** 0	00 ->直接右移，移位N=4

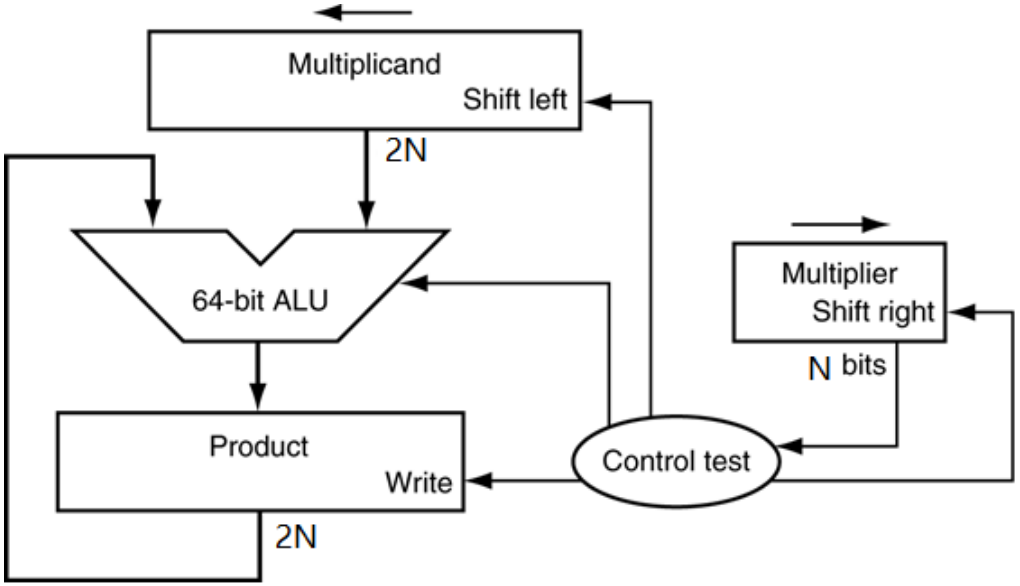
优化后的乘法器

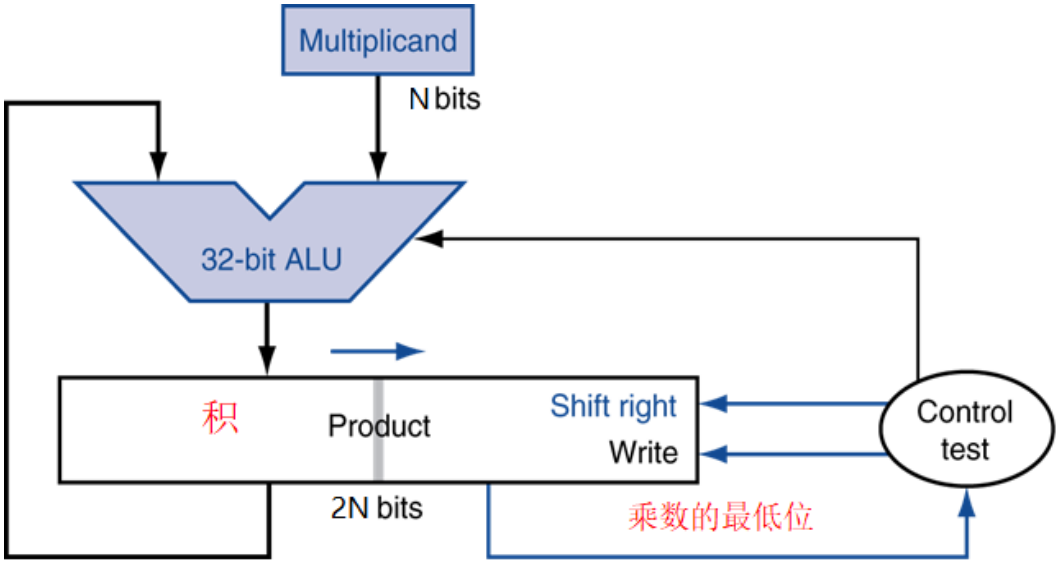
- 并行乘法器—BOOTH算法

3 X 7：0011 X 0111，(-3)补 = 1101

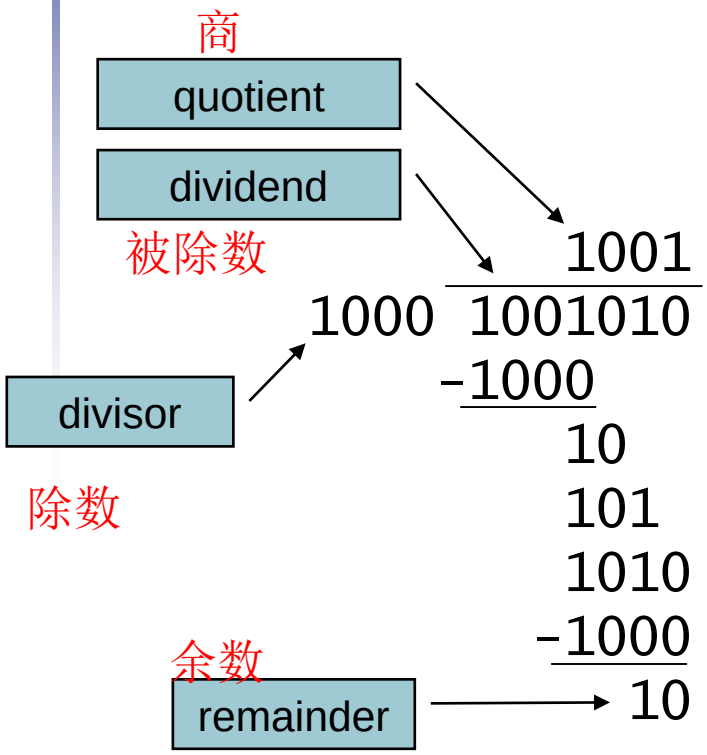
序号	P	注释
	0000 0000 0	原始乘积
	0000 0111 0	装载乘数，移位N=0
1	1101 0111 0	10 -> -(被乘数) -> +(被乘数)补
	1110 1011 1	右移，移位N=1
2	1111 0101 1	11 -> 直接右移，移位N=2
3	1111 1010 1	11 ->直接右移，移位N=3
4	0010 1010 1	01->+(被乘数)原
	0001 0101 0	右移，移位N=4

小结—乘法器


Multiplicand
Shift left
2N
64-bit ALU
Multiplier
Shift right
N bits
Product
Write
Control test
2N


Multiplicand
N bits
32-bit ALU
积
Product
Shift right
Write
Control
test
2N bits
乘数的最低位

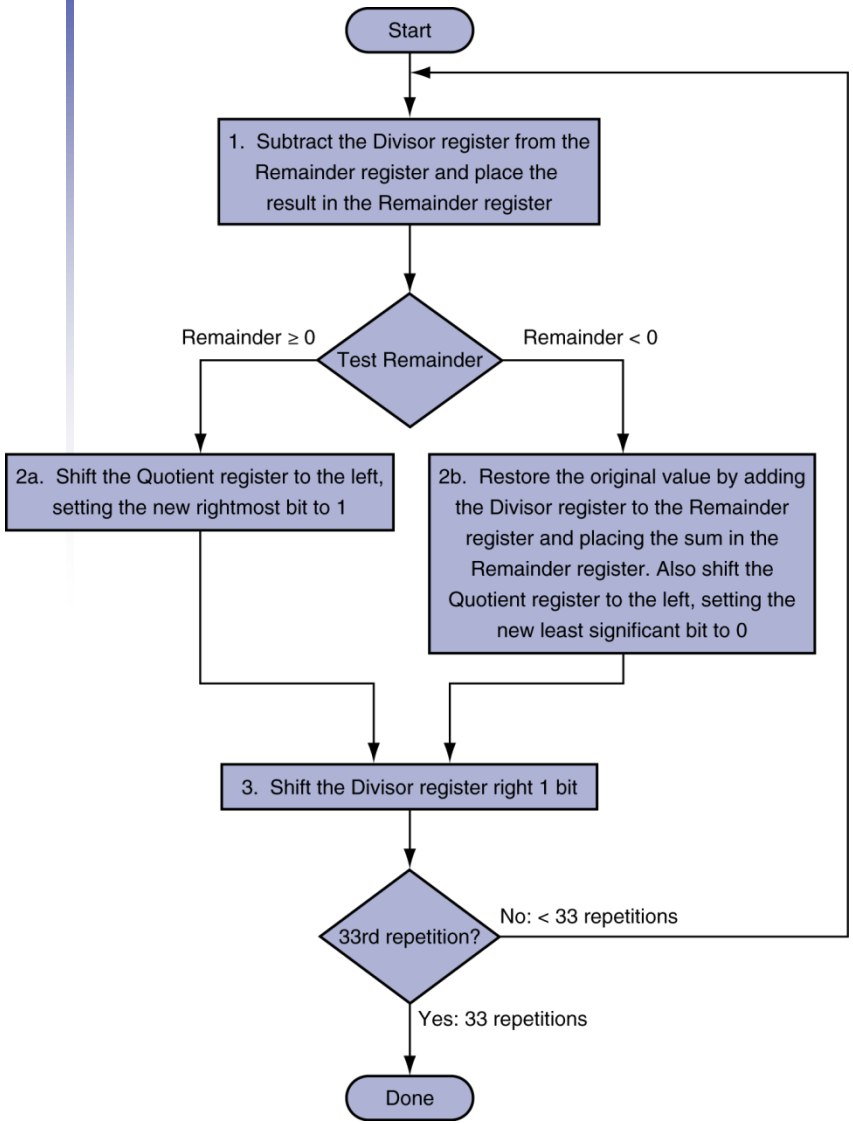
除法

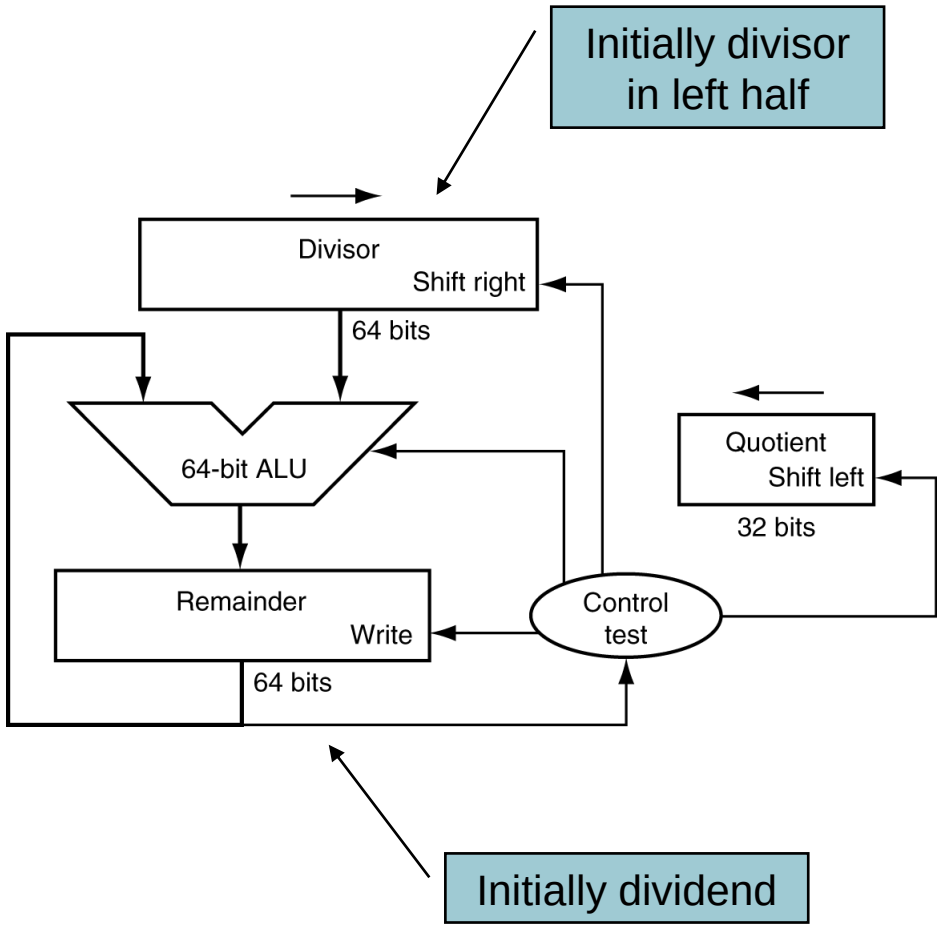


n-bit operands yield *n*-bit quotient and remainder

- 除数判0
- 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作为新的被除数
- 恢复余数法
  - 直接做减法，余数小于零再把除数加回去
- 带符号位的除法
  - 使用绝对值进行除法运算
  - 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
Remainder ≥ 0
Test Remainder
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

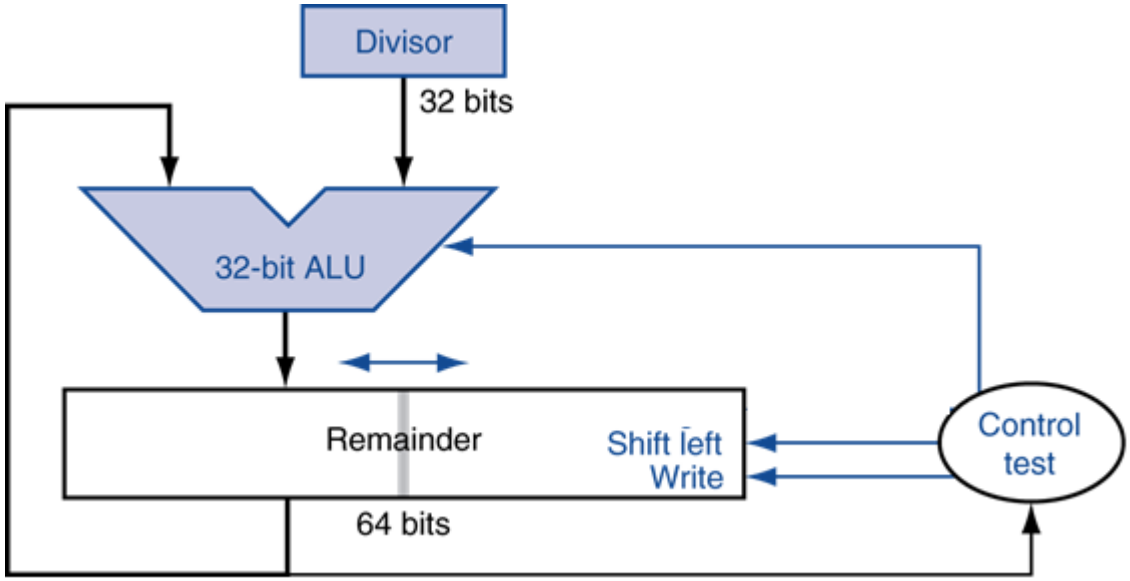

Initially divisor in left half
Divisor
Shift right
64 bits
64-bit ALU
Quotient
Shift left
32 bits
Remainder
Write
Control test
64 bits
Initially dividend

除法器的硬件

74 / 8

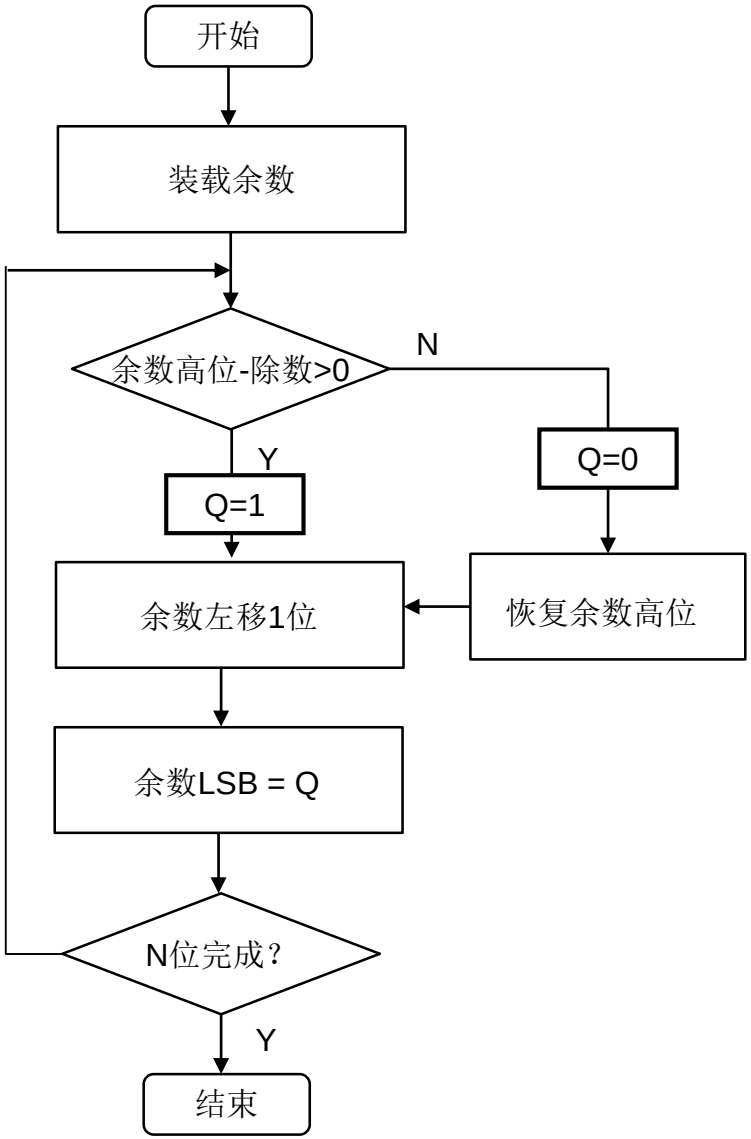
序号	商	除数	余数	注释
	0000	0000 0000	0000 0000	
	0000	1000 0000	0100 1010	装载
1	0000	1000 0000	1100 1010	余数-除数
			0100 1010	恢复余数
	0000	0100 0000	0100 1010	商左移、除数右移
2			0000 1010	余数-除数
	0001	0010 0000	0000 1010	商左移、除数右移
3			1110 1010	余数-除数
			0000 1010	恢复余数
	0010	00001 0000	0000 1010	商左移、除数右移
4			1111 1010	余数-除数
			0000 1010	恢复余数
	0100	0000 1000	0000 1010	商左移、除数右移
5			0000 0010	余数-除数
	1001	0000 0100		商左移、除数右移

优化后的除法器



- 一个时钟完成一次除法
- 和乘法器相似
  - 两者可以使用同样的硬件

优化后的除法器



13 / 4

序号	除数	余数	说明
	0100	0000 1101	装载
1		1100 1101	<0
		0000 1101	余数恢复
		0001 1010	余数左移
2		1101 1010	<0
		0001 1010	余数恢复
		0011 0100	余数左移
3		1111 0100	<0
		0011 0100	余数恢复
		0110 1000	余数左移
4		0010 1000	>0
		0101 0001	余数左移 余数LSB=1
5		0001 0001	>0
		0010 **0011**	余数左移 余数LSB=1

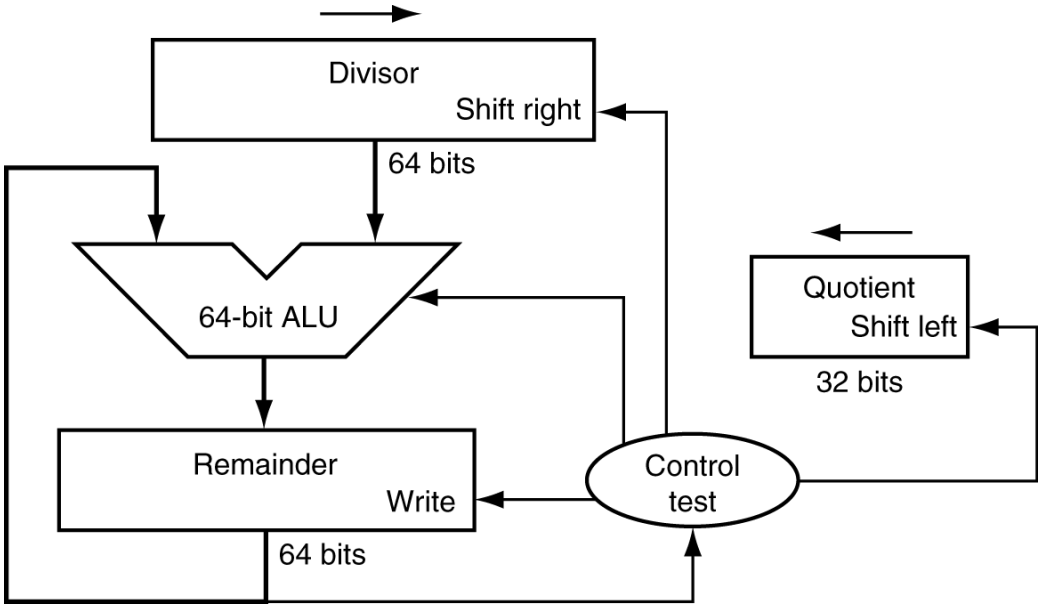
快速除法器

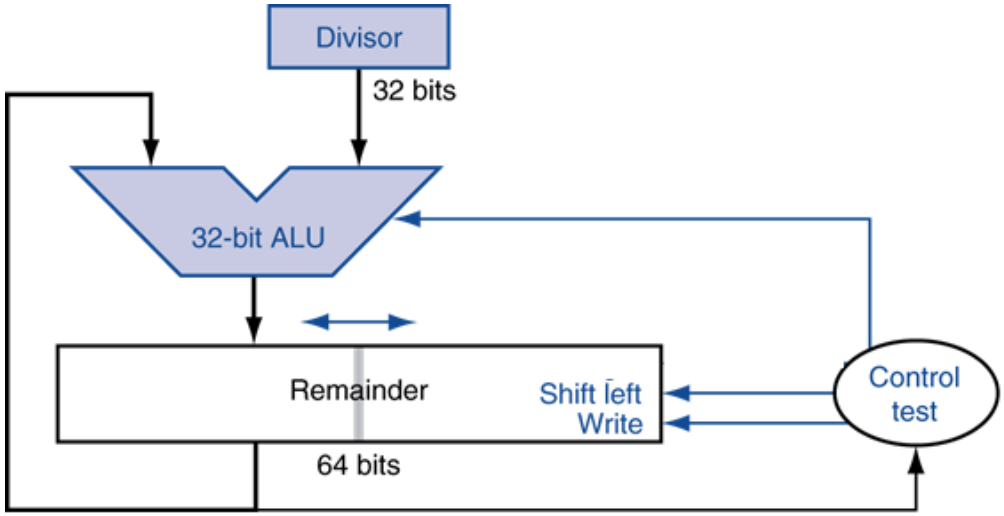
- 不可以使用乘法并行硬件
  - 除法需要考虑余数符号的情况
- 快速除法器(例如,SRT 除法器) 每一步生成多个商
  - 仍需要多步完成

MIPS Division

- Use HI/LO registers for result
  - HI: 32位余数
  - LO: 32位商
- 指令
  - `div rs, rt  /  divu rs, rt`
  - 不溢出或者除0检查
    - 如果需要，软件必须执行检查
  - 使用`mfhi, mflo`指令，访问结果

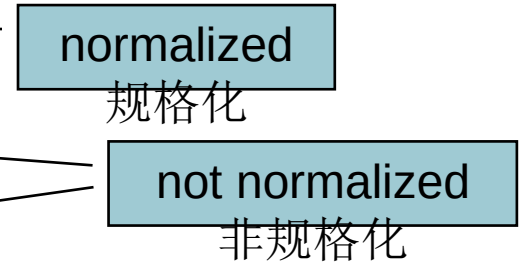
小结—除法器


Divisor
Shift right
64 bits
64-bit ALU
Quotient
Shift left
32 bits
Remainder
Write
Control
test
64 bits


Divisor
32 bits
32-bit ALU
Remainder
Shift left
Write
Control
test
64 bits

浮点数

- 表达非整形的数
  - 包括很小和很大的数
- 和科学计数法类似
  - -2.34×10^{56}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$
- 二进制表示
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- C语言中的类型：float和double



浮点数标准

- IEEE 754标准-1985
- 消除表达的不一致性
  - 科学计算中的可移植性
- 如今被普遍采用
- 两种表现
  - 单精度 (32-bit)
  - 双精度 (64-bit)

浮点数标准-IEEE 754

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

S	阶码（指数+偏移）	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)
- 尾数：有效位的规格化: 1.0 ≤ |有效位| < 2.0
  - 数前总有一个前导的1（作为隐含位可以不表示）
  - 有效位是：**$1+s1x2^{-1}+s2x2^{-2}+\cdots+sNx2^{-N}$**，s1…sN：尾数
- 阶码: 真实指数 + 偏移
  - 确保阶码非负
  - 偏移=>最小为0 : 单精度: 偏移 = 127; 双精度: 偏移= 1203

浮点数标准-IEEE 754

阶码-单精度

- 00000000 和11111111 保留，有特殊用途
- 最小值
  - 阶码: 00000001
    ⇒ 实际指数值= 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
    ⇒ 实际指数值 = 254 – 127 = +127
  - 尾数: 111…11 ⇒ 有效位 ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

浮点数标准-IEEE 754

阶码-双精度

0000…00 and 1111…11 保留，有特殊用途

- 最小值
  - 阶码: 00000000001
    ⇒ 实际指数值= 1 – 1023 = –1022
  - 尾数: 000…00 ⇒有效位 = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- 最大值
  - 阶码 11111111110
    ⇒实际指数值 = 2046 – 1023 = +1023
  - 尾数: 111…11 ⇒ 有效位 ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

浮点数标准-IEEE 754

- 相对精度
  - 尾数的所有位都是有效的
  - 单精度: 大约 2^{-23}
    - 相当于 $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ 精度的小数位数
  - 双精度: 大约2^{-52}
    - 相当于 $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ 精度的小数位数

浮点数标准-IEEE 754

- 表示 –0.75 → $(-0.11)_2$
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - 尾数= $1000\ldots00_2$
  - 阶码 = –1 + Bias
    - 单精度: –1 + 127 = 126 = 01111110_2
    - 双精度: –1 + 1023 = 1022 = 01111111110_2
- 单精度: 10111111101000…00
- 双精度: 10111111111101000…00

浮点数标准-IEEE 754

- 计算下列浮点数的真值

  1100000101000…00

  - S = 1
  - 尾数 = $01000\ldots00_2$
  - 阶码 = $10000001_2 = 129$

- $x = (-1)^1 \times (1 + 0.01_2) \times 2^{(129 - 127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

浮点数加法

- 以一个4位的十进制数为例
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- 1. 对阶
  - 把小阶的值调整到和大阶一致（-1调整到1）
  - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
- 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
  - 1.0015×10^{2}
- 4.进行必要的舍入处理
  - 1.002×10^{2}

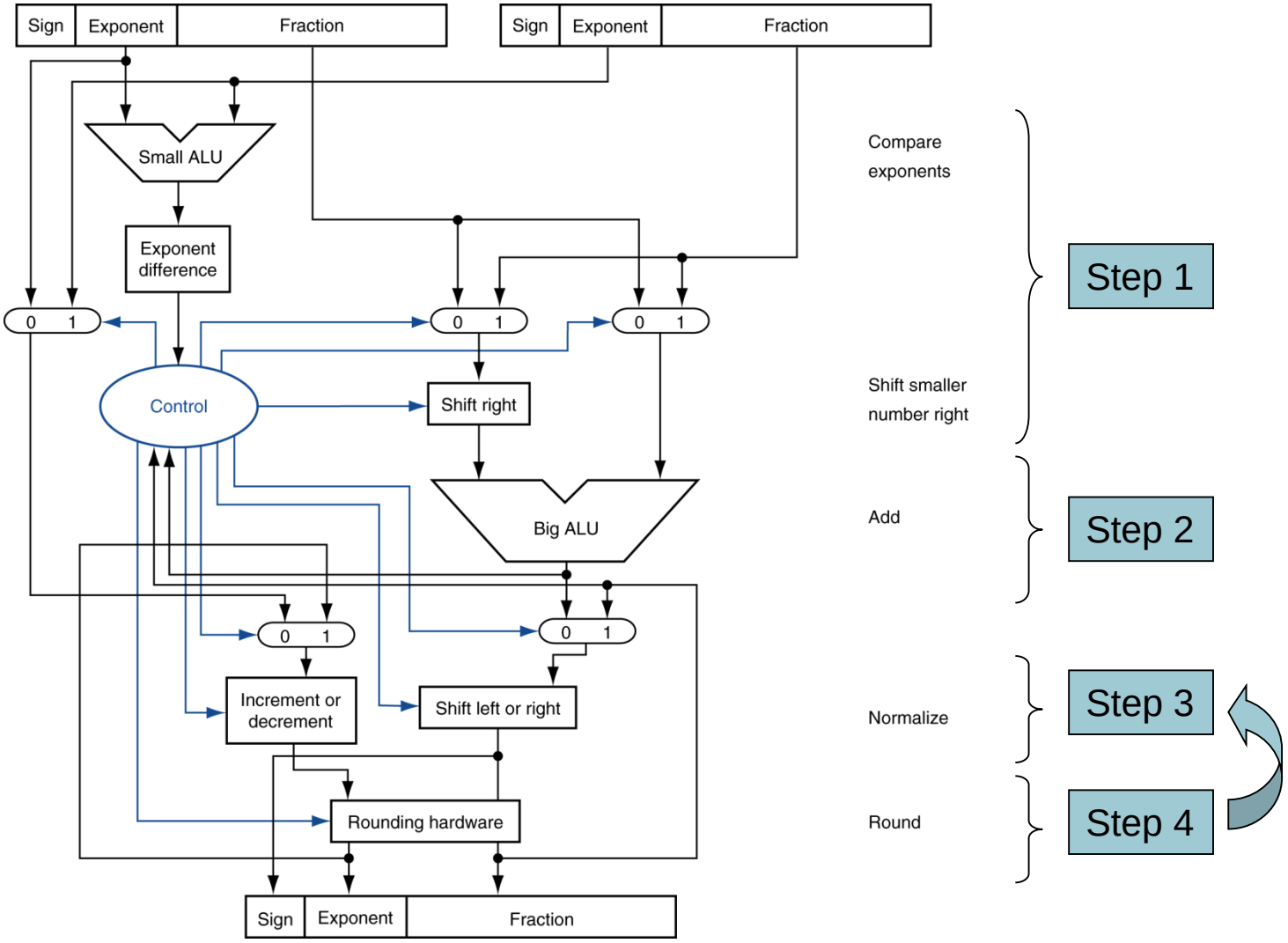
浮点数加法

- 现在计算0.5 + –0.4375，采用4位二进制数
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$
- 1. 对阶
  - 把小阶的值调整到和大阶一致
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. 尾数相加
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- 3. 结果规格化& 检查是否溢出
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. 进行必要的舍入处理
  - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

浮点加法硬件

- 比整数加法器复杂很多
- 如果在一个时钟周期内完成，就会要求时钟周期很长
  - 比整数运算费时许多
  - 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数个时钟周期
  - 可以被流水化

浮点加法硬件


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

浮点运算硬件

- 浮点乘法器与浮点加法器的复杂度相似
  - 但使用乘法器而不是加法器
- 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
- 操作通常需要数个时钟周期
  - 可以被流水化

MIPS中的浮点指令

- 浮点数使用协处理器
  - 通过ISA连接附加处理器
- 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
  - 程序通常不会对浮点数据进行整数操作，反之亦然。
  - 在最小的指令长度影响下，提供更多的寄存器
- 浮点数读取、存储指令
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - 例如., `ldc1 $f8, 32($sp)`

MIPS中的浮点指令

- 单精度运算
  - add.s, sub.s, mul.s, div.s
    - 例如, add.s $f0, $f1, $f6
- 双精度运算
  - add.d, sub.d, mul.d, div.d
    - 例如, mul.d $f4, $f4, $f6
- 单精度和双精度的比较
  - 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
- 浮点条件代码之下的分支
  - bc1t, bc1f
    - 例如, bc1t TargetLabel

浮点示例: ° F to ° C

- C 语言:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - fahr变量存在$f12, 结果存在 $f0, 常量存在全局共享内存

- 编译后的MIPS 指令:

```
f2c: lwc1  $f16, const5($gp)
     lwc2  $f18, const9($gp)
     div.s $f16, $f16, $f18     //5.0/9.0
     lwc1  $f18, const32($gp)
     sub.s $f18, $f12, $f18     //fahr-32.0
     mul.s $f0,  $f16, $f18     // *
     jr    $ra
```

浮点示例: 数组乘法

- a = a + b × c
  - 都是32 × 32 矩阵, 64-bit 双精度元素
- C 语言:

```
void mm (double b[][],
         double c[][], double a[][]) {
  int i, j, k;
  for (i = 0; i! = 32; i = i + 1)
    for (j = 0; j! = 32; j = j + 1)
      for (k = 0; k! = 32; k = k + 1)
        a[i][j] = a[i][j]
                + b[i][k] * c[k][j];
}
```

  - a, b, c 地址存在 $a0, $a1, $a2, and i, j, k 存在 $s0, $s1, $s2

浮点示例: 数组乘法（双精度DGEMM）

双精度数需要相邻的一对浮点寄存器来存储

- MIPS 指令:

循环初始条件

```
    li   $t1, 32        # $t1 = 32 (row size/loop end)
    li   $s0, 0         # i = 0; initialize 1st for loop
L1: li   $s1, 0         # j = 0; restart 2nd for loop
L2: li   $s2, 0         # k = 0; restart 3rd for loop
    sll  $t2, $s0, 5    # $t2 = i * 32 (size of row of a)
    addu $t2, $t2, $s1  # $t2 = i * size(row) + j
    sll  $t2, $t2, 3    # $t2 = byte offset of [i][j]
    addu $t2, $a0, $t2  # $t2 = byte address of a[i][j]
    l.d  $f4, 0($t2)    # $f4 = 8 bytes of a[i][j]
L3: sll  $t0, $s2, 5    # $t0 = k * 32 (size of row of b)
    addu $t0, $t0, $s1  # $t0 = k * size(row) + j
    sll  $t0, $t0, 3    # $t0 = byte offset of [k][j]
    addu $t0, $a2, $t0  # $t0 = byte address of b[k][j]
    l.d  $f16, 0($t0)   # $f16 = 8 bytes of b[k][j]
    …
```

浮点示例: 数组乘法

```
…
    sll  $t0, $s0, 5        # $t0 = i*32 (size of row of c)
    addu  $t0, $t0, $s2     # $t0 = i*size(row) + k
    sll   $t0, $t0, 3       # $t0 = byte offset of [i][k]
    addu  $t0, $a1, $t0     # $t0 = byte address of c[i][k]
    l.d   $f18, 0($t0)      # $f18 = 8 bytes of c[i][k]
    mul.d $f16, $f18, $f16  # $f16 = c[i][k] * b[k][j]
    add.d $f4, $f4, $f16    # f4=a[i][j] + c[i][k]*b[k][j]
    addiu $s2, $s2, 1       # $k k + 1
    bne   $s2, $t1, L3      # if (k != 32) go to L3
    s.d   $f4, 0($t2)       # a[i][j] = $f4
    addiu $s1, $s1, 1       # $j = j + 1
    bne   $s1, $t1, L2      # if (j != 32) go to L2
    addiu $s0, $s0, 1       # $i = i + 1
    bne   $s0, $t1, L1      # if (i != 32) go to L1
```

运算精确

- IEEE 754标准指定舍入控制
  - 额外比特位的精度(guard, round, sticky)
  - 舍入模式的选择
  - 允许程序员微调
- 不是所有的浮点单元都执行所有选项
  - 绝大多数编程语言和浮点库使用默认定义
- 根据硬件复杂性，性能以及市场需求折中

子字并行

- 图形和音频应用可以利用短矢量执行同步操作处理
  - 例如: 128-bit加法器:
    - 16个 8-bit adds
    - 8个16-bit adds
    - 4个 32-bit adds
- 也称为数据级并行，并行向量，或者单指令多数据 (SIMD)
- ARM的NEON/X86的SSE/AVX

x86浮点指令结构

- 最初基于8087浮点协处理器
  - 8 × 80-bit 扩展精度的寄存器
  - 构成一个向下推的栈结构
  - 可按照下标访问 TOS: ST(0), ST(1), …
- 在内存中的浮点数也是32-bit 或者 64-bit
  - 在load/store转换
  - 整数转化也能在 load/store转换
- 生成和优化代码很困难
  - 结果: 浮点性能较差poor FP

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 个寄存器 在 AMD64/EM64T 中
- 可以在浮点乘法操作中使用
  - 2 × 64-bit 双精度
  - 4 × 32-bit 单精度
  - 指令并行操作
    - Single-Instruction Multiple-Data

矩阵乘法

- 未优化的代码:

```
1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.  for (int i = 0; i < n; ++i)
4.    for (int j = 0; j < n; ++j)
5.    {
6.     double cij = C[i+j*n]; /* cij = C[i][j] */
7.     for(int k = 0; k < n; k++ )
8.      cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.     C[i+j*n] = cij; /* C[i][j] = cij */
10.   }
11. }
```

矩阵乘法

- x86汇编指令:

```
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
```

矩阵乘法

- 优化后的C 语言代码:

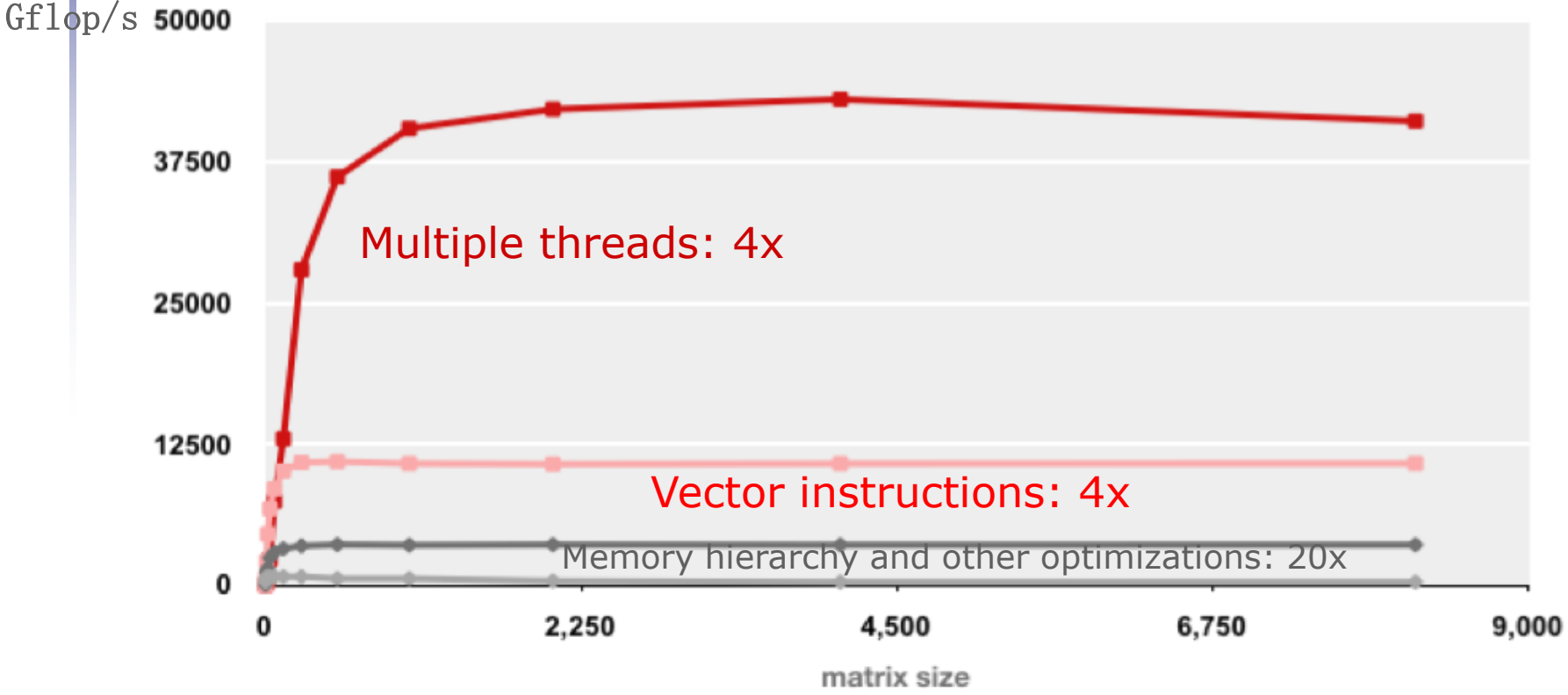
```
1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double* C)
3. {
4.  for ( int i = 0; i < n; i+=4 )
5.   for ( int j = 0; j < n; j++ ) {
6.    __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j]
   */
7.    for( int k = 0; k < n; k++ )
8.     c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
9.               _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.              _mm256_broadcast_sd(B+k+j*n)));
11.   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.  }
13. }
```

矩阵乘法

- 优化后的x86 汇编指令:

```
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 %ymm1,4 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
```

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz



- Reason for 20x: Blocking or tiling, loop unrolling, array scalarization, instruction scheduling, search to find best choice
- ***Effect: fewer register spills, L1/L2 cache misses, and TLB misses***

右移和除法

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

结合律

- 并行程序可以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

- 需要在不同程度的并行下，验证并行程序的可靠性

谁在乎浮点数精确度?

- 对于科学计算很重要
  - 当在某些日常中?
    - “我的余额查了0.0002¢!” ☹
- Pentium FDIV指令的bug
  - 用户还是希望能精确计算
  - See Colwell, *The Pentium Chronicles*

谁在乎浮点数精确度?

- Pentium FDIV指令的bug

正常电脑计算器运算结果：962306957033÷11010046=87402.6282027341
带有 Bug 的奔腾 CPU 运算结果：962306957033÷11010046=87339.5805831329

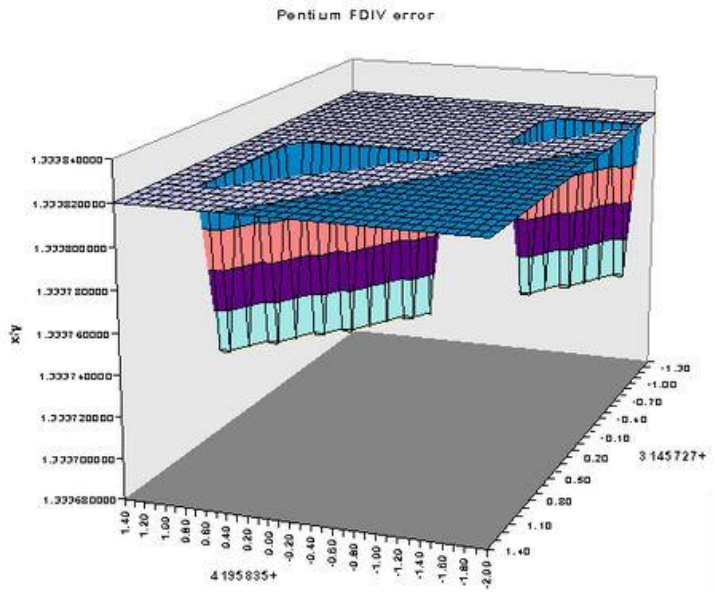
The image below is a 3d graph of the function x/y in the region of 4195835/3145727. Specifically, the region is:

4195833.0 <= x <= 4195836.4

3145725.7 <= y <= 3145728.4

On a 486/66, the function graphs as a monotonically increasing surface with the low point in the foreground corner (where x is low and y is high) and a high point in the background corner (where x is high and y is low). On a Pentium with the FDIV bug there are two triangular areas in the region where an incorrect result is returned. The correct values all would round to 1.3338 and the incorrect values all would round to 1.3337, an error in the 5th significant digit.

An archive of principal papers on the Pentium FDIV bug is available at: http://www.mathworks.com/company/pentium/index.shtml



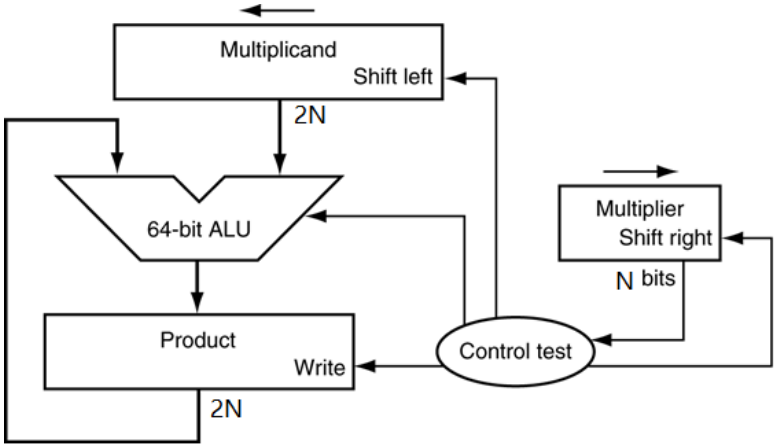
总结

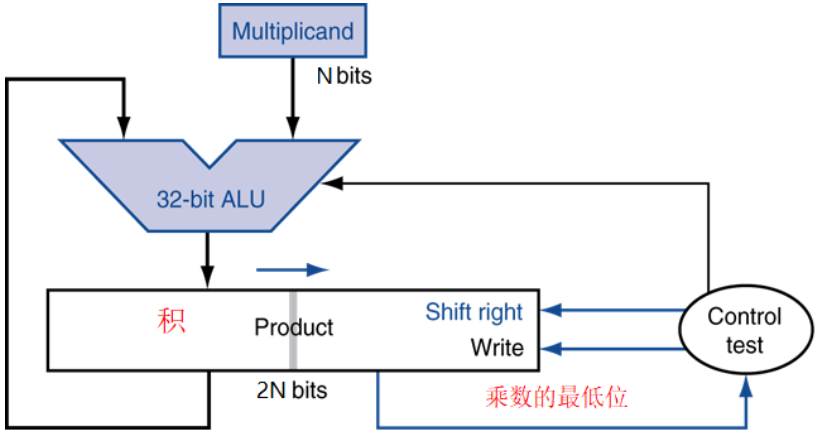
- Bits 没有内在含义
  - 解释依赖于应用的说明
- 数字的计算机表示
  - 有限的范围和精度
  - 方案中需要考虑到这一点

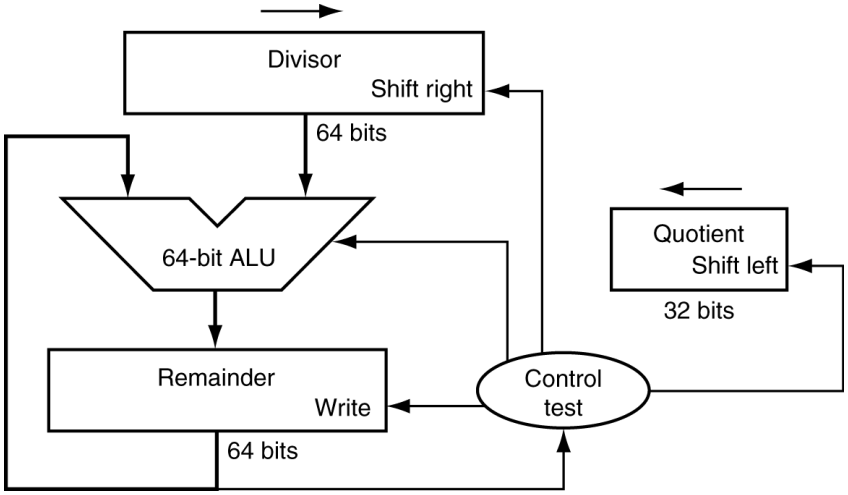
总结

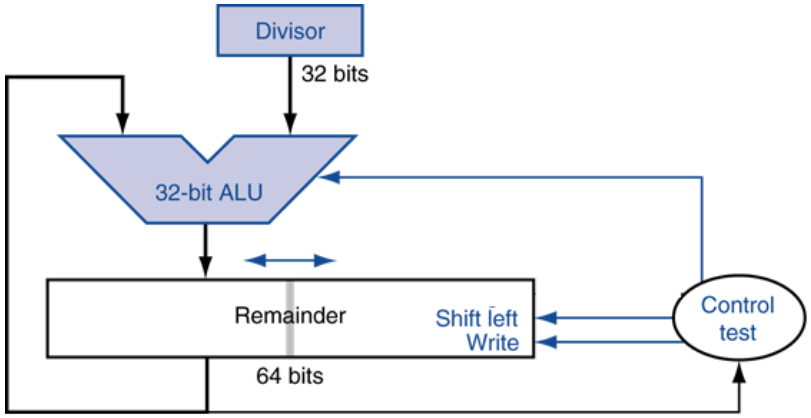
- ISAs 支持的运算
  - 有符号和无符号的整数
  - 和实数类似的浮点数
- 受限的范围和精度
  - 计算可能会溢出
- MIPS 指令结构
  - 核心指令: 54最常用
    - 覆盖100% SPECINT, 覆盖 97% SPECFP
  - 其它指令: 很少使用

小结—乘、除法器


Multiplicand
Shift left
2N
64-bit ALU
Multiplier
Shift right
N bits
Product
Write
Control test
2N


Multiplicand
N bits
32-bit ALU
积
Product
Shift right
Write
Control
test
2N bits
乘数的最低位


Divisor
Shift right
64 bits
64-bit ALU
Quotient
Shift left
32 bits
Remainder
Write
Control
test
64 bits


Divisor
32 bits
32-bit ALU
Remainder
Shift left
Write
Control
test
64 bits