Chapter 3

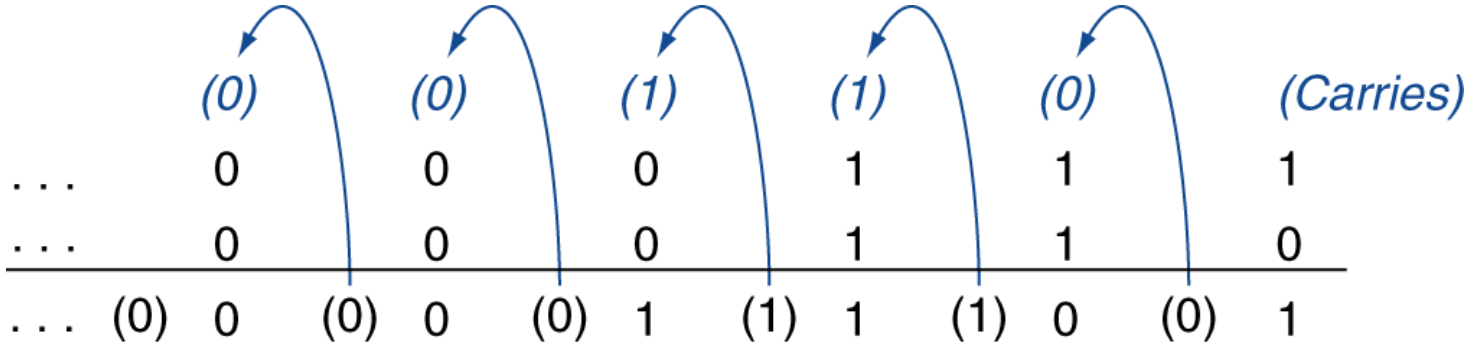
Arithmetic for Computers

Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Representation and operations

Integer Addition

- Example: 7 + 6



- Overflow if result out of range
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if result sign is 1
  - Adding two –ve operands
    - Overflow if result sign is 0

Integer Subtraction

- Add negation of second operand
- Example: 7 – 6 = 7 + (–6)

  +7: 0000 0000 … 0000 0111

  –6: 1111 1111 … 1111 1010

  +1: 0000 0000 … 0000 0001
- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 1
  - 意味着借位占用了符号位

全加器（Full Adder）

- 输入为加数、被加数和低位进位Cin，输出为和F、进位Cout

真值表

A	B	Cin	F	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$F=\overline{A}\cdot\overline{B}\cdot Cin+\overline{A}\cdot B\cdot\overline{Cin}+A\cdot\overline{B}\cdot\overline{Cin}+A\cdot B\cdot Cin$$

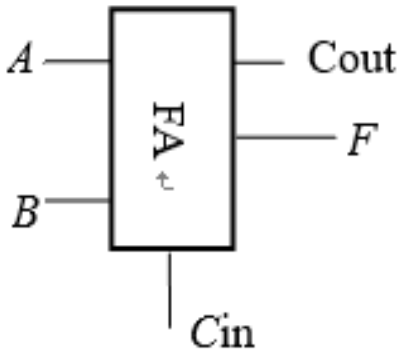
$$Cout=\overline{A}\cdot B\cdot Cin+A\cdot\overline{B}\cdot Cin+A\cdot B\cdot\overline{Cin}+A\cdot B\cdot Cin$$

化简后：

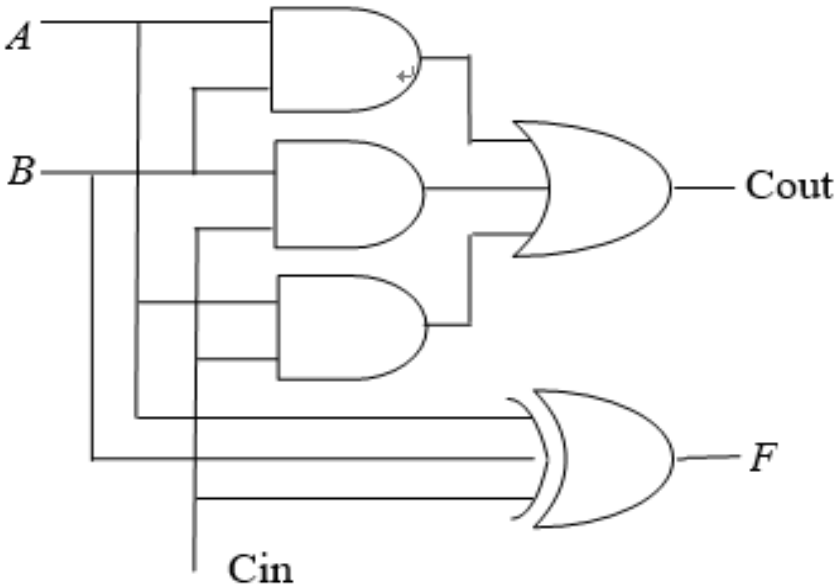
$$F=A\oplus B\oplus Cin$$

$$Cout=A\cdot B+A\cdot Cin+B\cdot Cin$$

逻辑符号



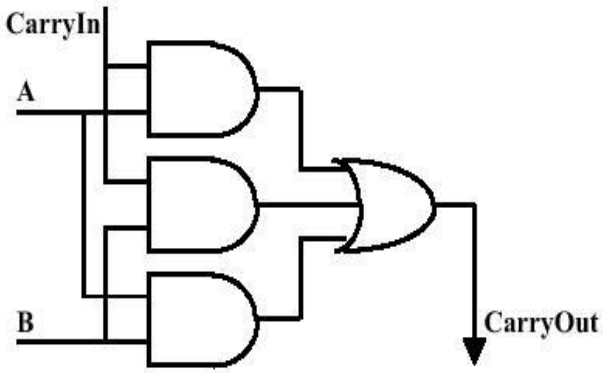
逻辑电路图



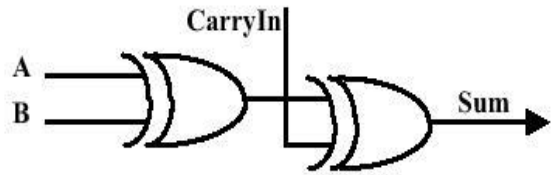
串行进位加法器

CarryOut 和 Sum 的逻辑图

- CarryOut = B & CarryIn | A & CarryIn | A & B



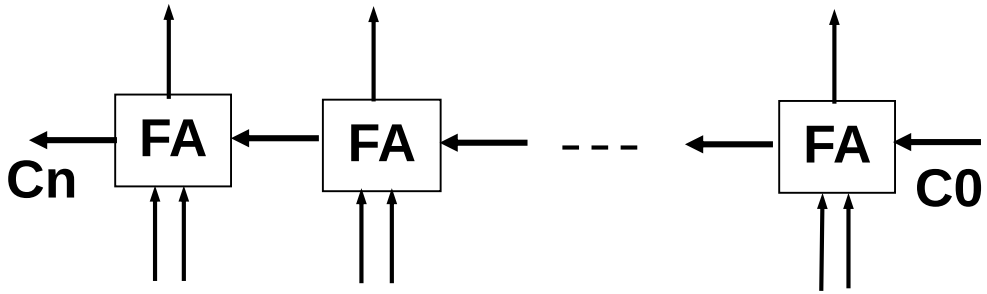
- Sum = A XOR B XOR CarryIn



假定异或门是3个与/或门的延迟，则“和”与“进位”的延迟为多少?

Sum延迟为6个门延迟；Carryout为2个。

n位串行(行波)加法器：



串行加法器的缺点：

进位按串行方式传递，速度慢!

问题：n位串行加法器从C0到Cn的延迟时间为多少？2n级门延迟!

最后一位和数的延迟时间为多少？ 2n+1级门延迟!

先行进位加法器（CLA）

- 为什么用先行进位方式?
  - 串行进位加法器采用串行逐级传递进位，电路延迟与位数成正比关系。因此，现代计算机采用一种先行进位(Carry Lookahead)方式。
- 如何产生先行进位?

  定义辅助函数：$G_i=A_iB_i$ —— 进位生成函数

  $P_i=A_i+B_i$ —— 进位传递函数
- 全加逻辑方程：$F_i=A_i\oplus B_i\oplus C_i$　$C_{i+1}=G_i+P_iC_i$　(i=0,1,…n)

先行进位加法器（CLA）

- 设n=4,则：

$$C_1=G_0+P_0C_0$$

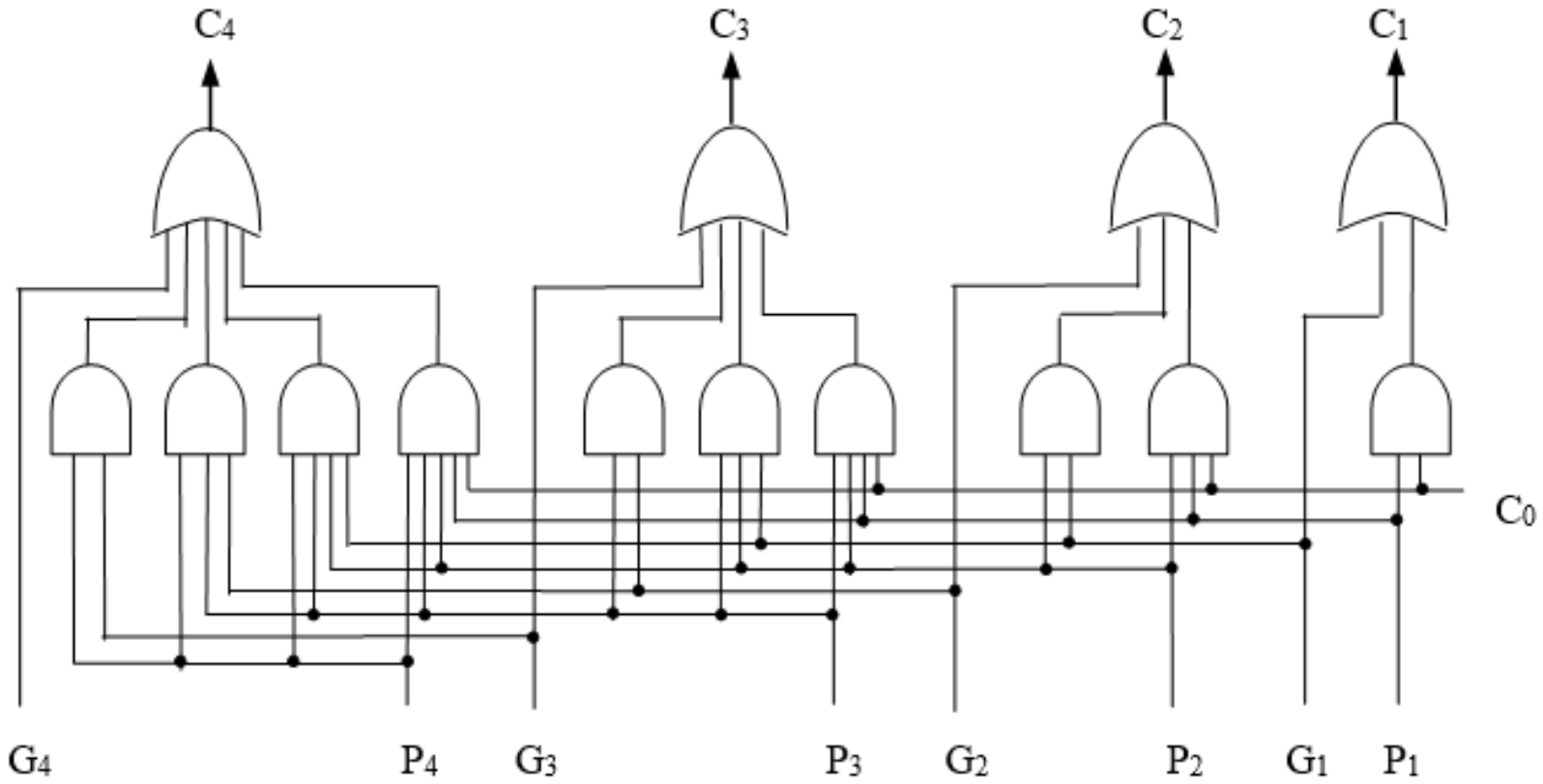
$$C_2=G_1+P_1C_1=G_1+P_1G_0+P_1P_0C_0$$

$$C_3=G_2+P_2C_2=G_2+P_2G_1+P_2P_1G_0+P_2P_1P_0C_0$$

$$C_4=G_3+P_3C_3=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0+P_3P_2P_1P_0C_0$$

- 由上式可知:各进位之间无等待，相互独立并同时产生。
- 通常把实现上述逻辑的电路称为4位先行进位部件（4位CLU）

4位CLU



$C_1 = G_1 + P_1C_0$

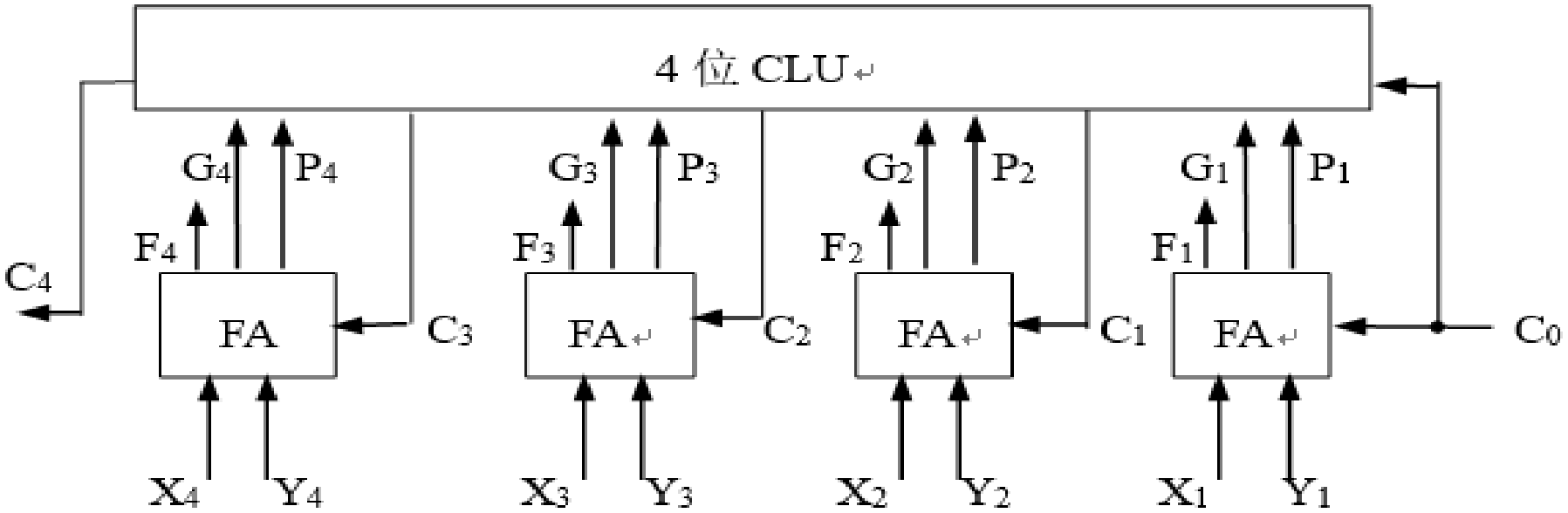
$C_2 = G_2 + P_2C_1 = G_2 + P_2G_1 + P_2P_1C_0$

$C_3 = G_3 + P_3C_2 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1C_0$

$C_4 = G_4 + P_4C_3 = G_4 + P_4G_3 + P_4P_3G_2 + P_4P_3P_2G_1 + P_4P_3P_2P_1C_0$

4位全先行进位加法器CLA

产生所有和数为1+2+3=6级门延迟



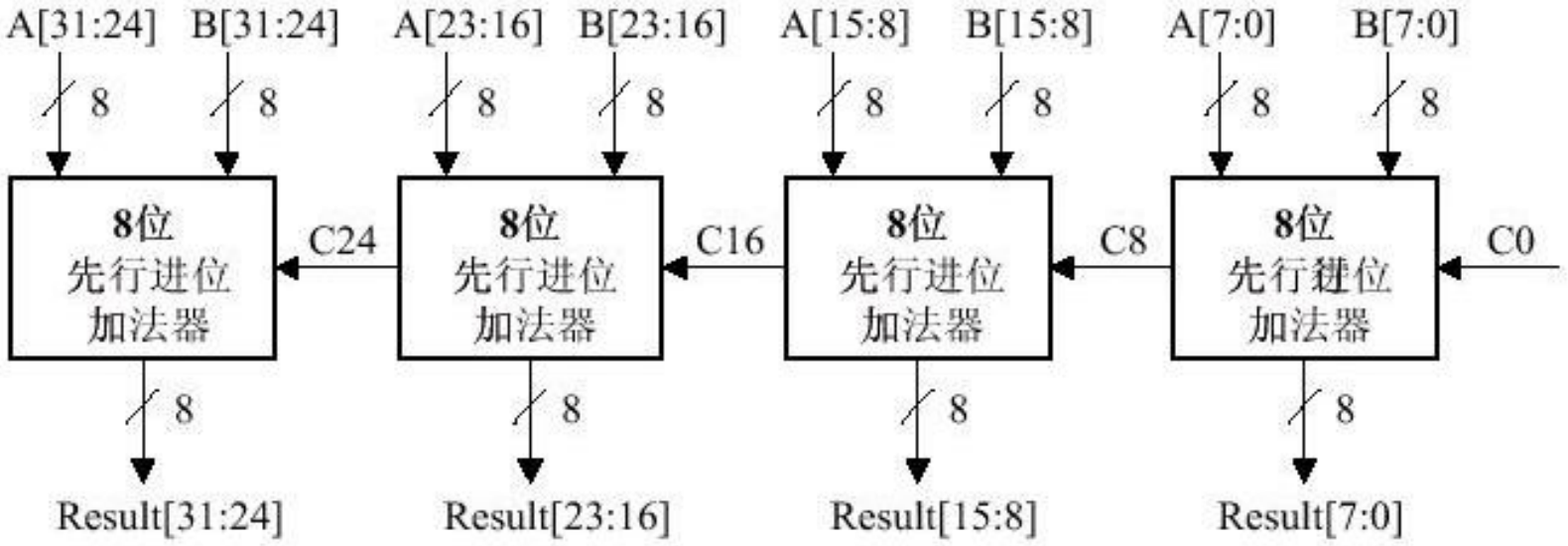
$G_i=X_iY_i$

$P_i=X_i+Y_i$ （或 $P_i=X_i \oplus Y_i$ ）

$F_i=X_i \oplus Y_i \oplus C_{i-1}$

局部（单级）先行进位加法器

- 产生全先行进位加法器的成本太高
  - 想象Cin31的逻辑公式的长度
- 一般性的经验，**关键路径: (A_{7-0}、B_{7-0}、C_0) → C_8→ C_{16} → C_{24}**
  - 连接一些N位先行进位加法器，形成一个大加法器
  - 例：连接4个8位先行进位加法器，形成1个32位局部先行进位加法器

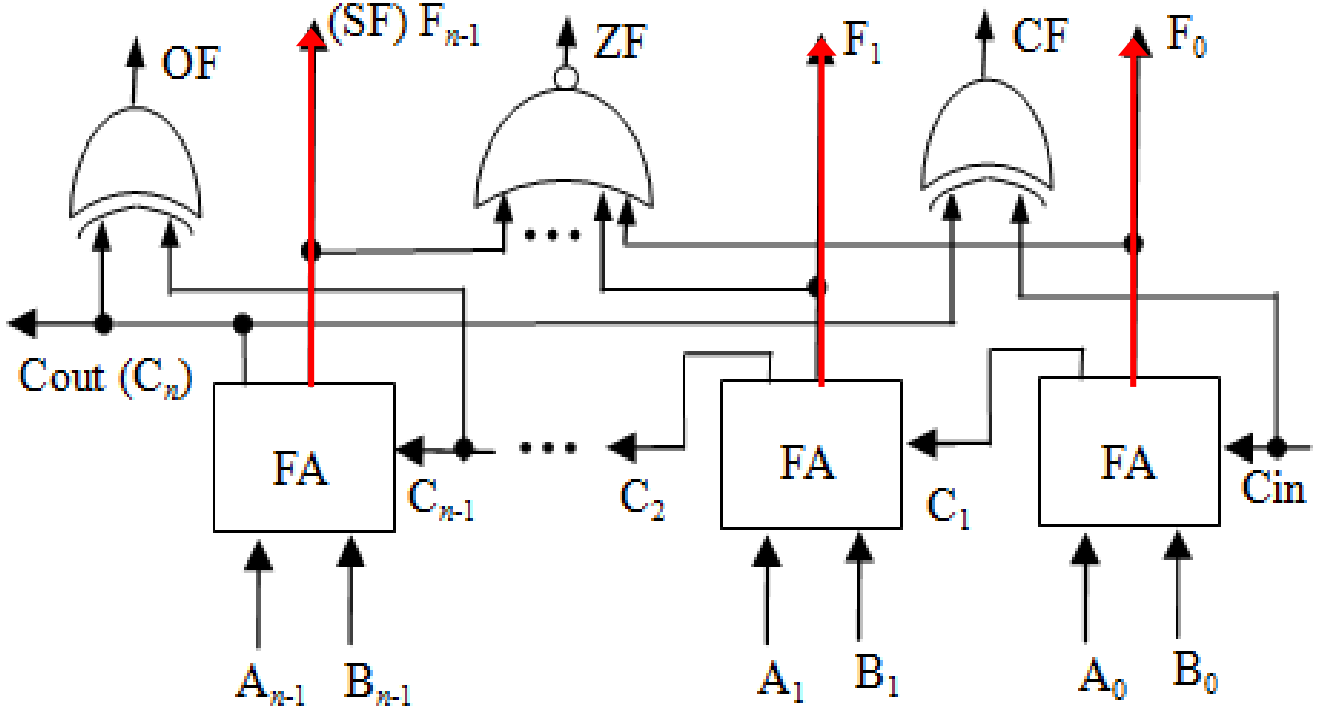


问题：所有和数产生的延迟为多少？

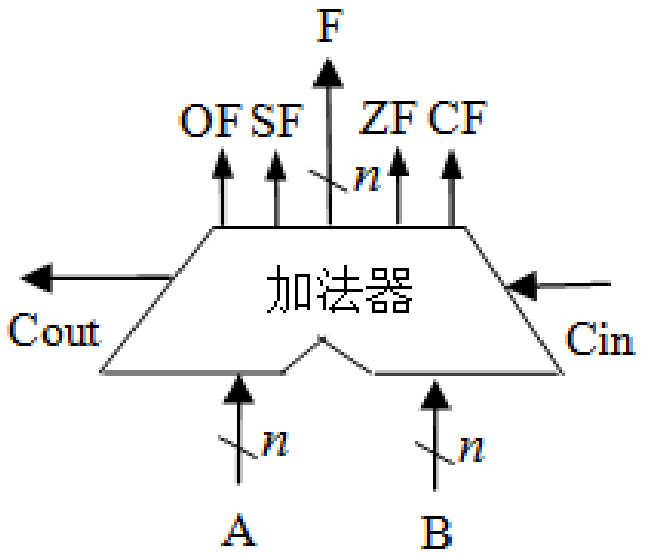
3+2+2+5=12级门延迟

n位带标志加法器

- 两个n位带符号整数（补码）相加，需要判断是否溢出
- 程序中经常需要比较大小，通过（在加法器中）做减法得到的标志信息来判断



带标志加法器的逻辑电路



带标志加法器符号

溢出标志OF：$OF=C_n \oplus C_{n-1}$

符号标志SF：$SF=F_{n-1}$

零标志ZF：=1当且仅当F=0；

进位/借位标志CF：$CF=Cout \oplus Cin$

n位整数加/减运算器

补码加减运算公式

$[A+B]_{补} = [A]_{补} + [B]_{补} \ (\mathrm{mod}\ 2^n)$

$[A-B]_{补} = [A]_{补} + [-B]_{补} \ (\mathrm{mod}\ 2^n)$

–实现减法的主要工作在于：求$[-B]_{补}$

问题：如何求$[-B]_{补}$?

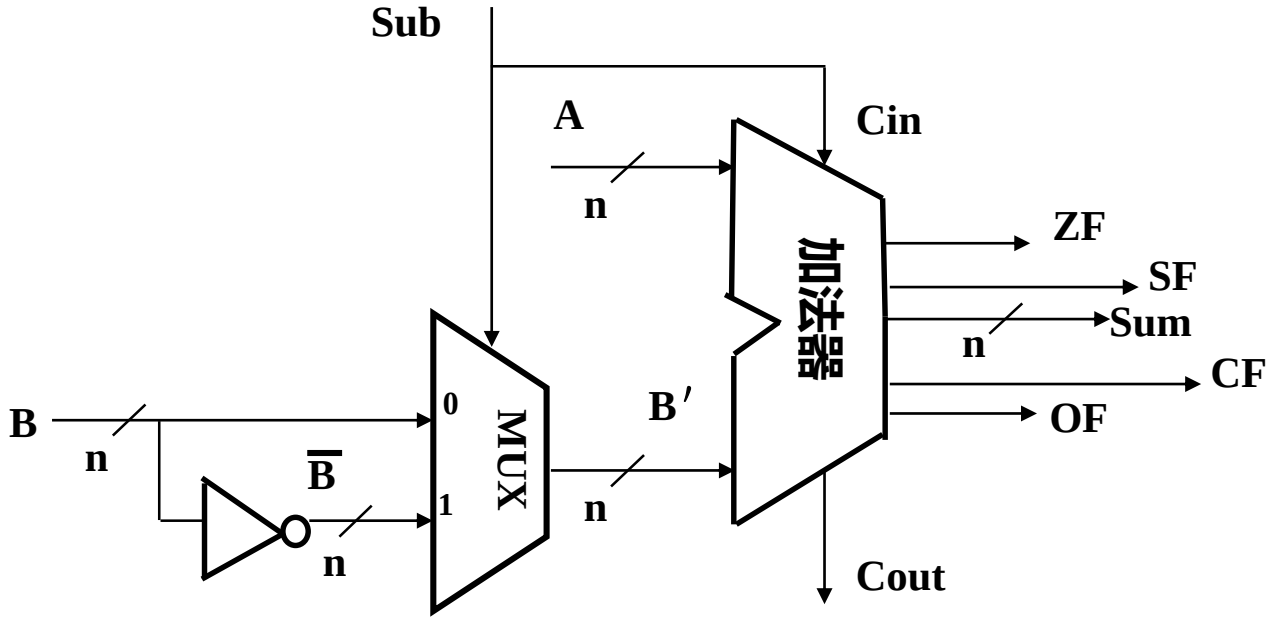
$[-B]_{补} = \overline{[B]_{补}} + 1$

当Sub为1时，做减法
当Sub为0时，做加法

利用带标志加法器，可构造整数加/减运算器，进行以下运算：

无符号整数加、无符号整数减

有符号整数加、有符号整数减



Dealing with Overflow

- 有符号数和无符号数人溢出判断不一样
  - 无符号数根据最高位是否有进位判断溢出，通常不判
  - MIPS：无符号数运算溢出时，不产生“溢出异常”
- Some languages (e.g., C) ignore overflow
  - Use MIPS `addu`, `addui`, `subu` instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
  - Use MIPS `add`, `addi`, `sub` instructions

Dealing with Overflow

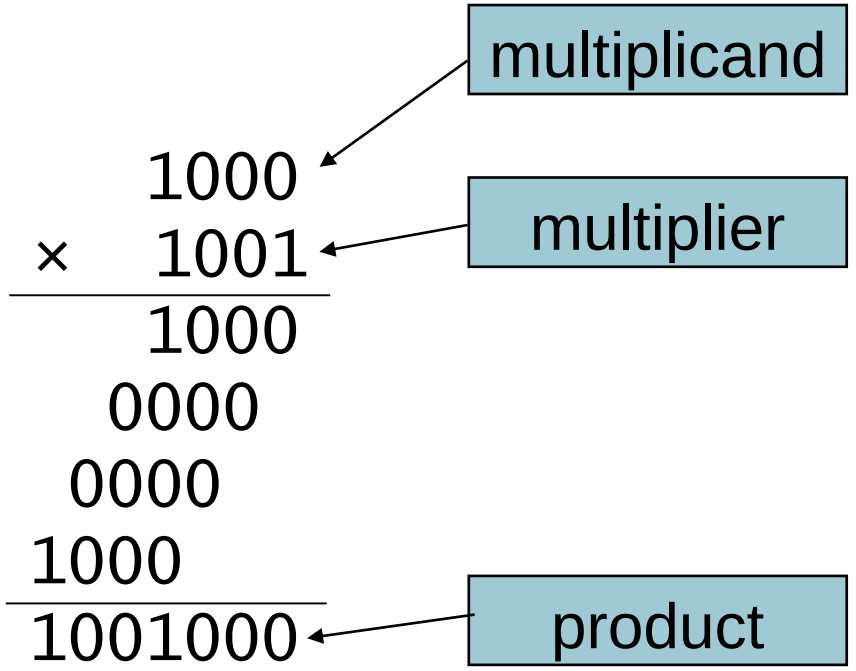
- On overflow, invoke exception handler
  - Save PC in exception program counter (EPC) register
  - Jump to predefined handler address
  - `mfc0` (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action

Arithmetic for Multimedia

- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
  - SIMD (single-instruction, multiple-data)
- Saturating operations
  - On overflow, result is largest representable value
    - c.f. 2s-complement modulo arithmetic
  - E.g., clipping in audio, saturation in video

Unsigned Multiplication

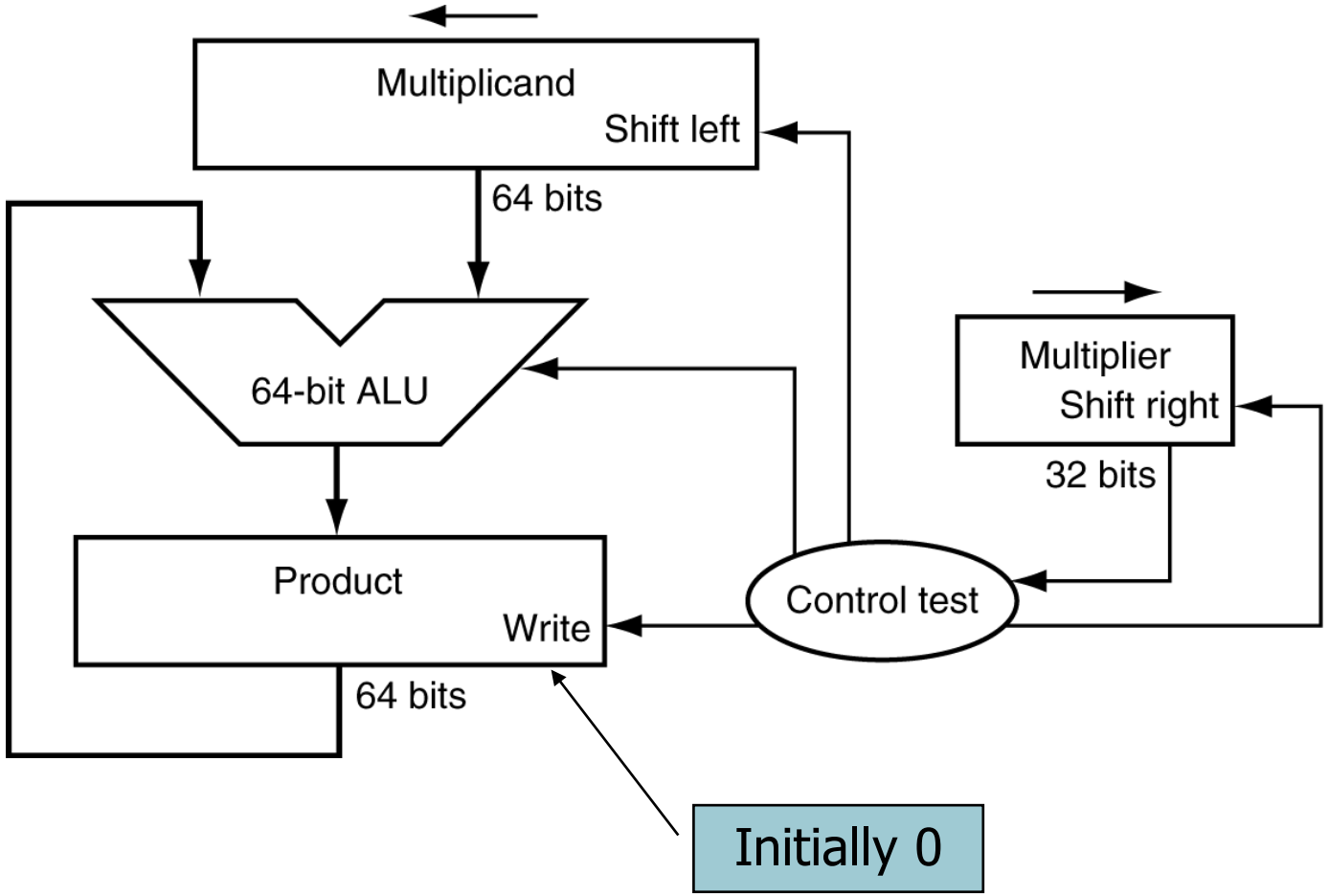
整个运算过程中用到两种操作：加法 + 左移，因而，可用ALU和移位器来实现乘法运算

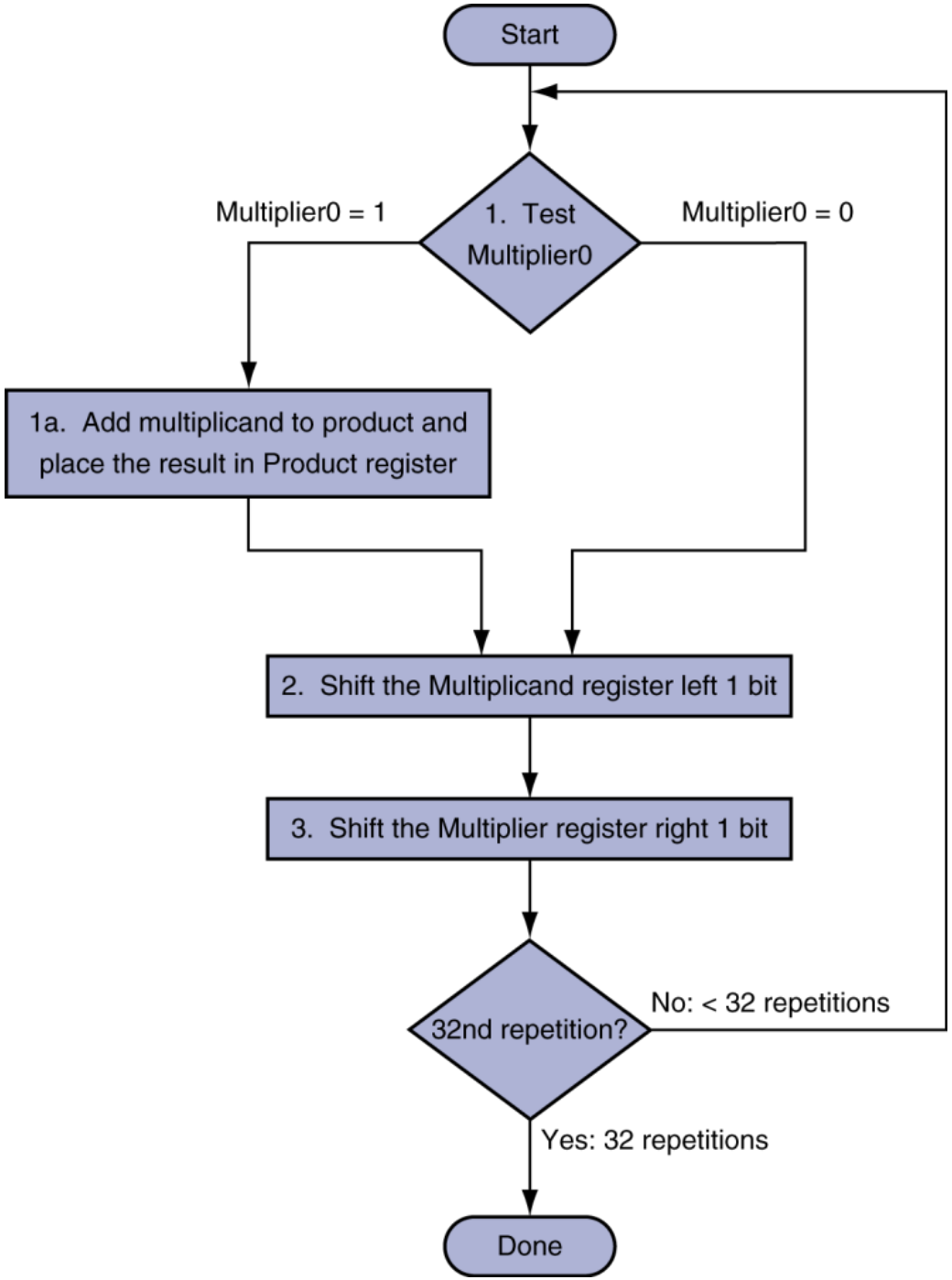


Length of product is the sum of operand lengths

Multiplication Hardware

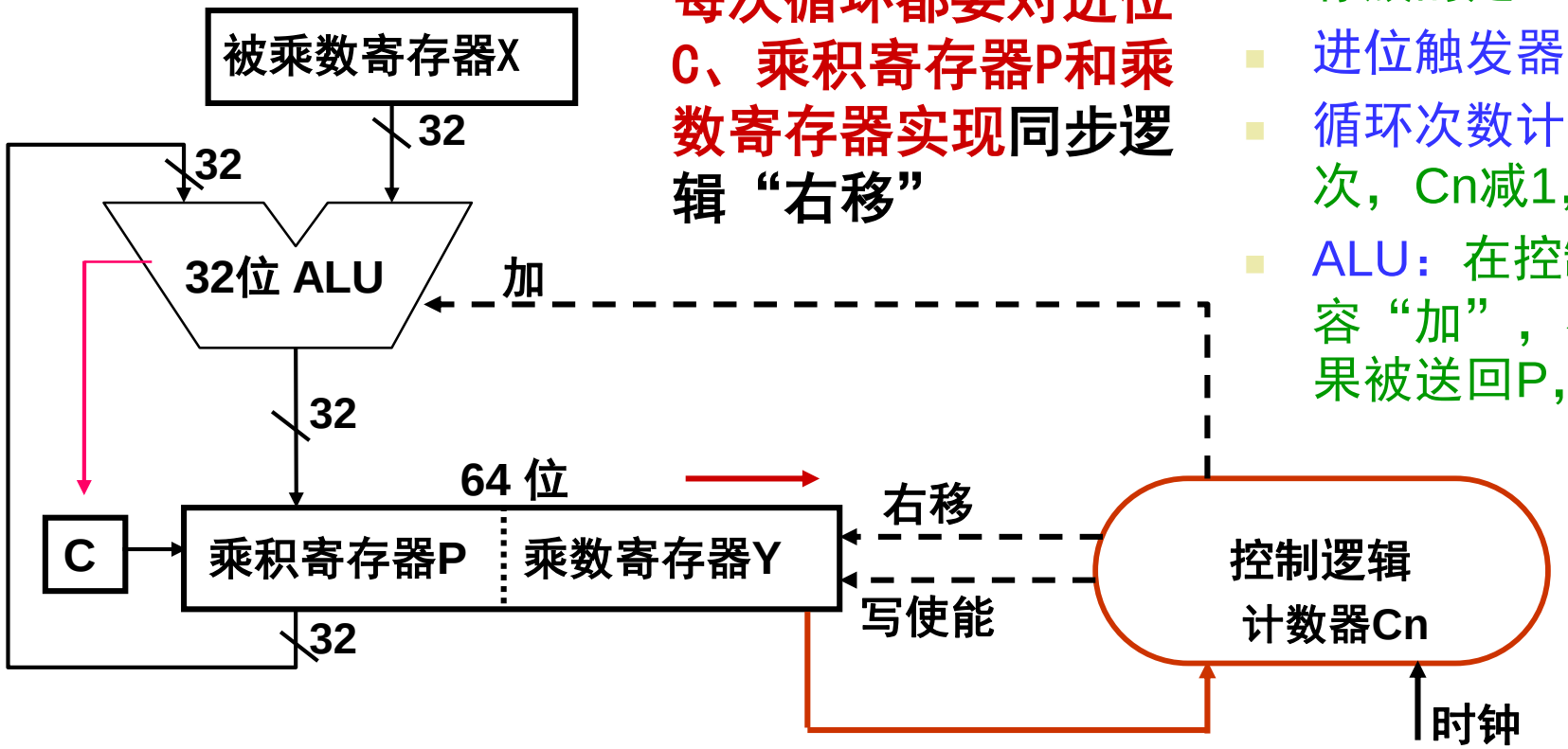
- Start with long-multiplication approach




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

Optimized Multiplier

- Perform steps in parallel: add/shift. One cycle per partial-product addition
  - That's ok, if frequency of multiplications is low



每次循环都要对进位C、乘积寄存器P和乘数寄存器实现同步逻辑“右移”

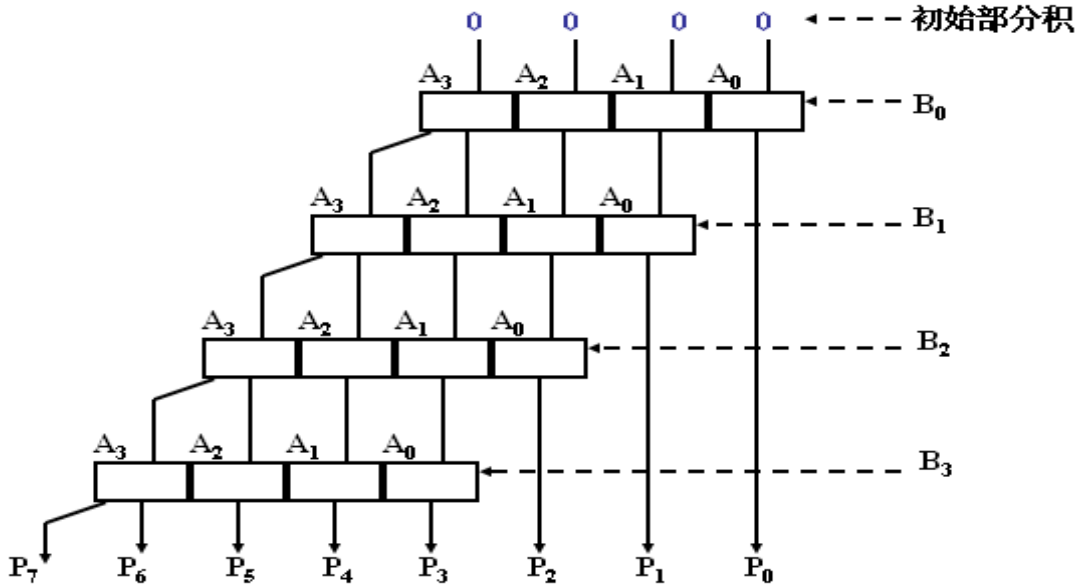
- 被乘数寄存器X：存放被乘数
- 乘积寄存器P：开始置初始部分积 = 0；结束时，存放的是64位乘积的高32位
- 乘数寄存器Y：开始置乘数；结束时，存放的是64位乘积的低32位
- 进位触发器C：保存加法器的进位信号
- 循环次数计数器Cn：初值32，每循环一次，Cn减1，Cn=0时结束
- ALU：在控制逻辑控制下，对P和X的内容“加”，在“写使能”控制下运算结果被送回P，进位在C中

快速乘法器

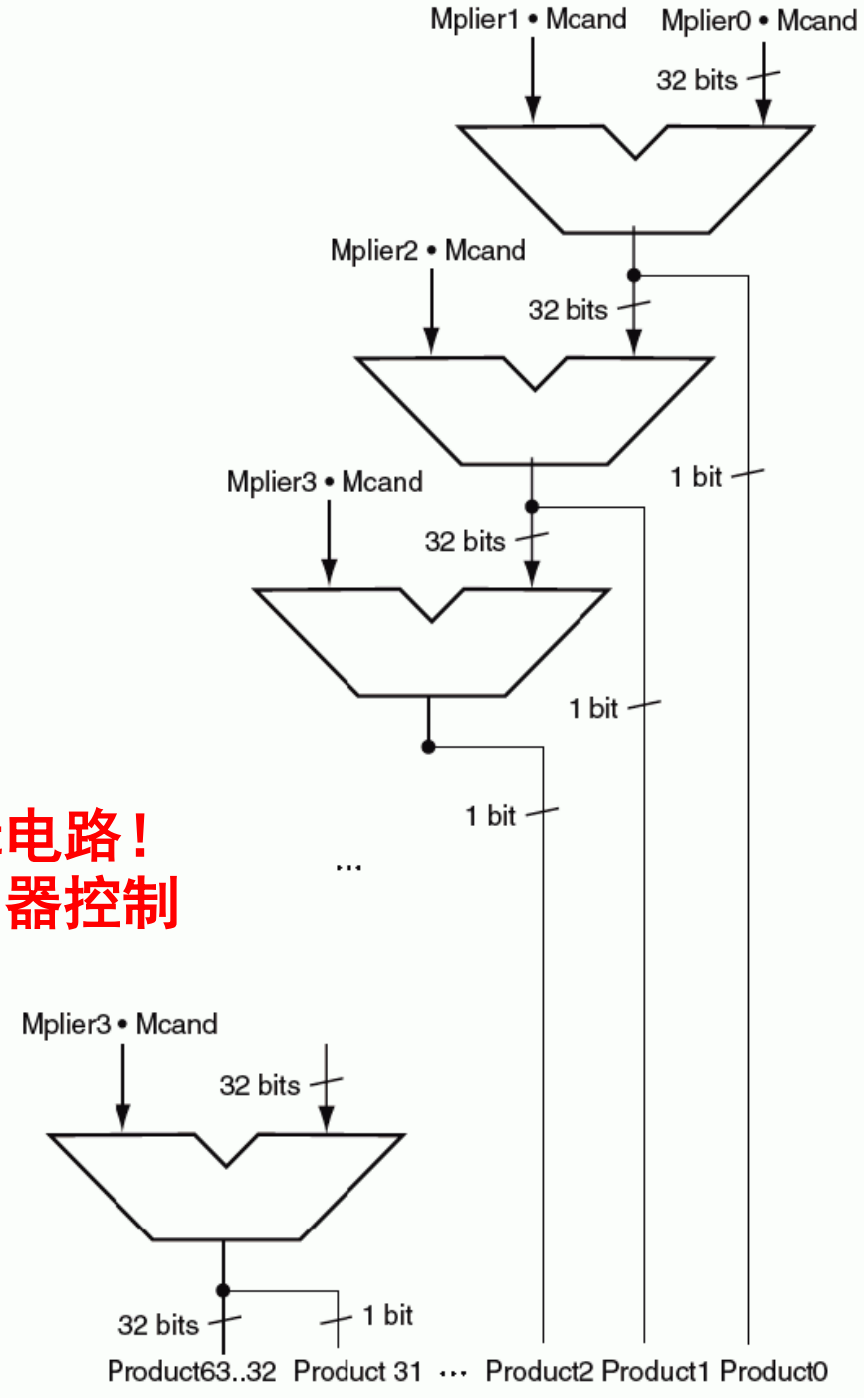
- 前面介绍的乘法部件的特点
  - 通过一个ALU多次做“加+右移”来实现
    - 一位乘法：约n次“加+右移”
    - 两位乘法：约n/2次“加+右移”
  - 所需时间随位数增多而加长，由时钟和控制电路控制
- 设计快速乘法部件的必要性
  - 大约1/3是乘法运算
- 快速乘法器的实现（由特定功能的组合逻辑单元构成）
  - 流水线方式
  - 硬件叠加方式（如：阵列乘法器）

流水线方式的快速乘法器

- 为乘数的每位提供一个n位加法器
- 每个加法器的两个输入端分别是：
  - 本次乘数对应的位与被乘数相的结果（0或被乘数）
  - 上次部分积
- 每个加法器的输出分为两部分：
  - 和的最低有效位(LSB)作为本位乘积
  - 进位和高31位的和组成一个32位数作为本次部分积

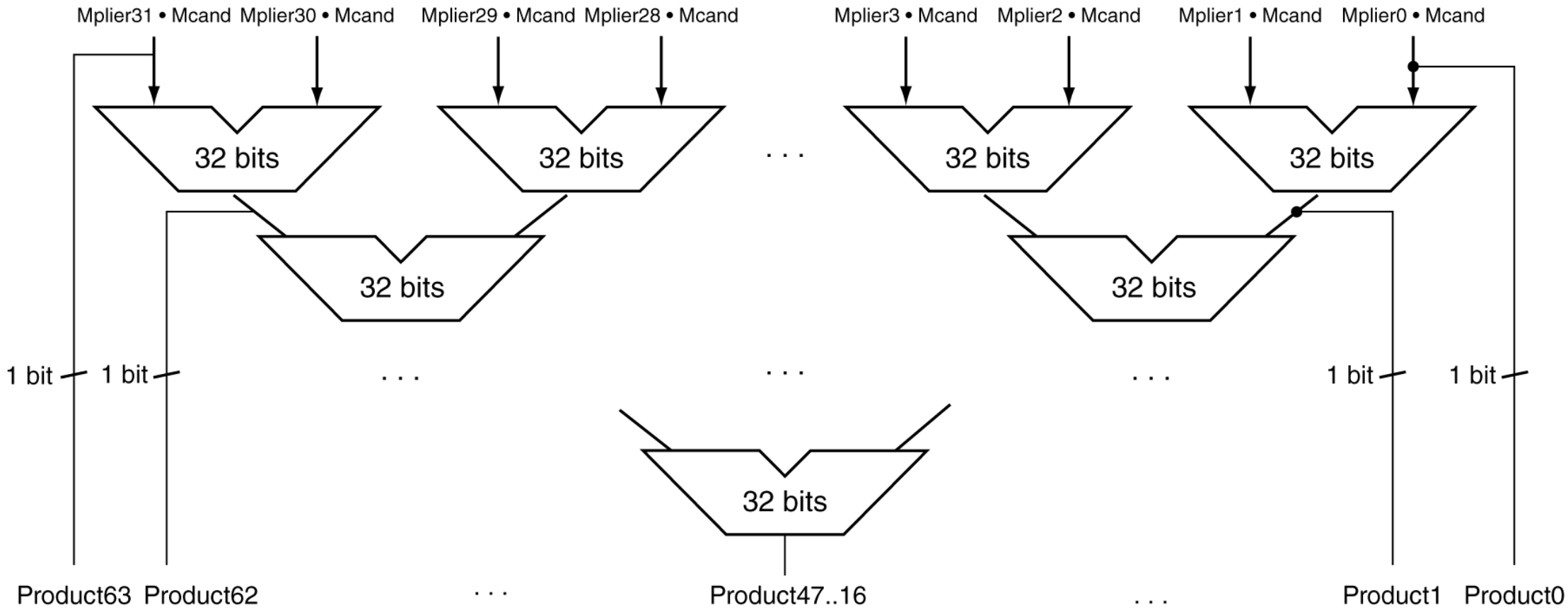


组合逻辑电路！
无需控制器控制



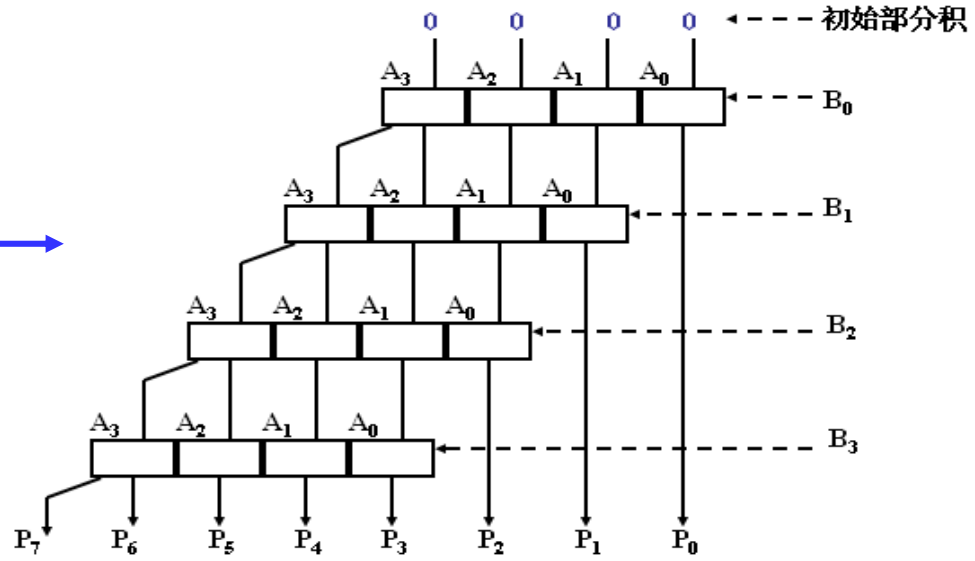
Faster Multiplier

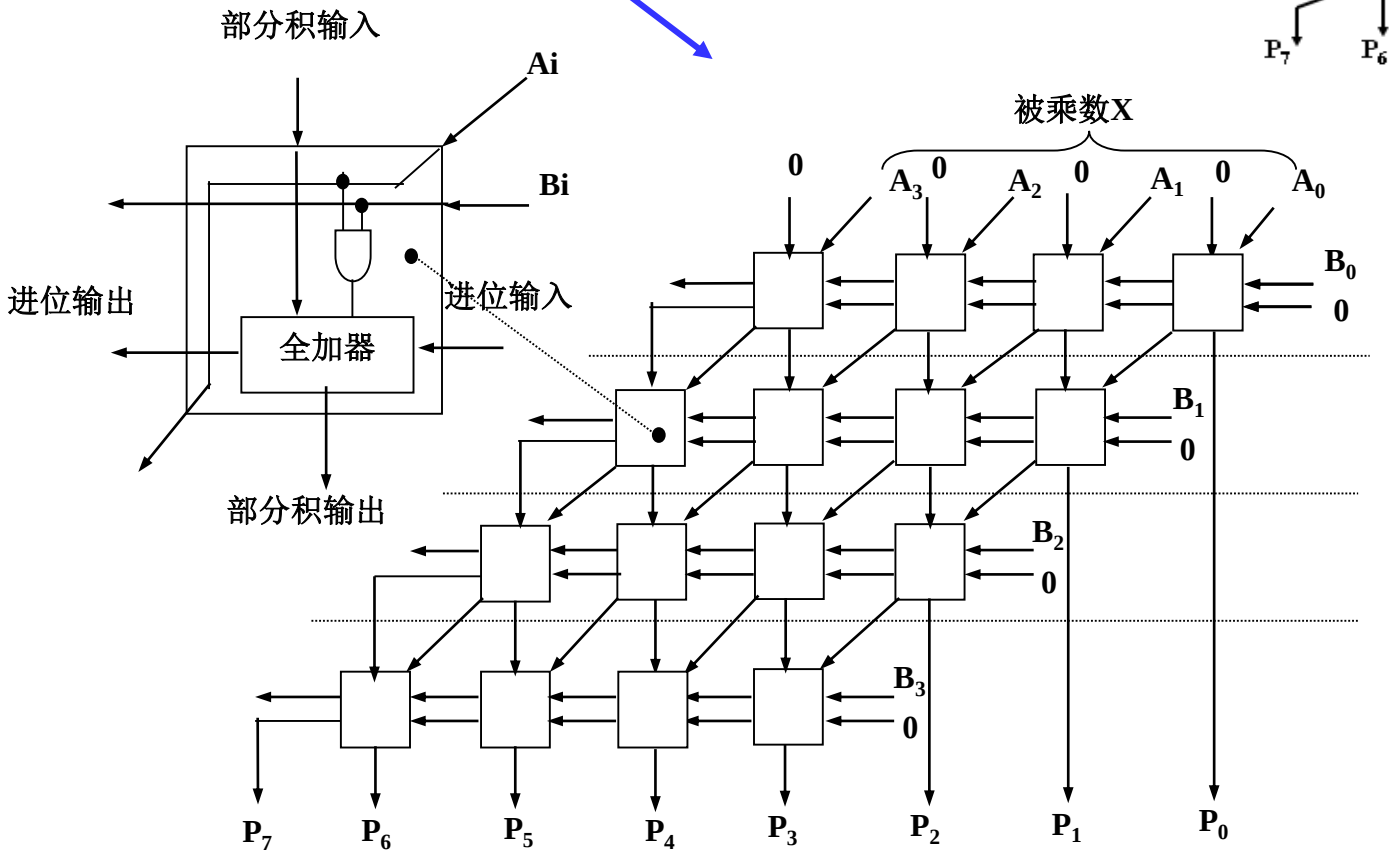
- Uses multiple adders, cost/performance tradeoff
- Can be pipelined, Several multiplication performed in parallel



阵列乘法器

- 手算乘法过程
- 阵列乘法器





速度仅取决于逻辑门和加法器的传输延迟

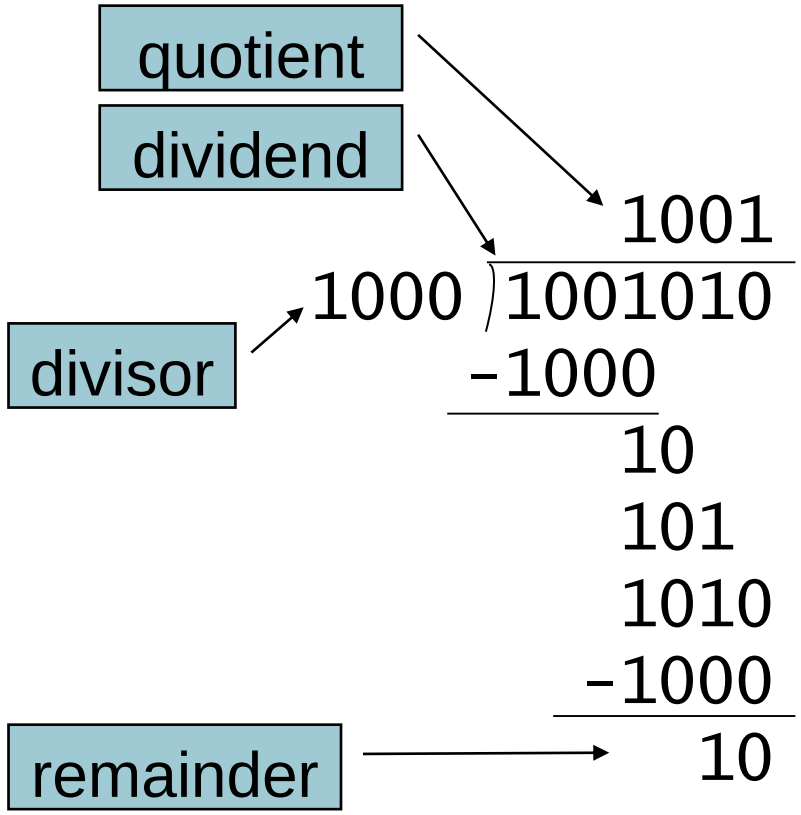
无符号阵列乘法器

增加符号处理电路、乘前及乘后求补电路，即可实现带符号数乘法器。

MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - `mult rs, rt  /  multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd  /  mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product –> rd

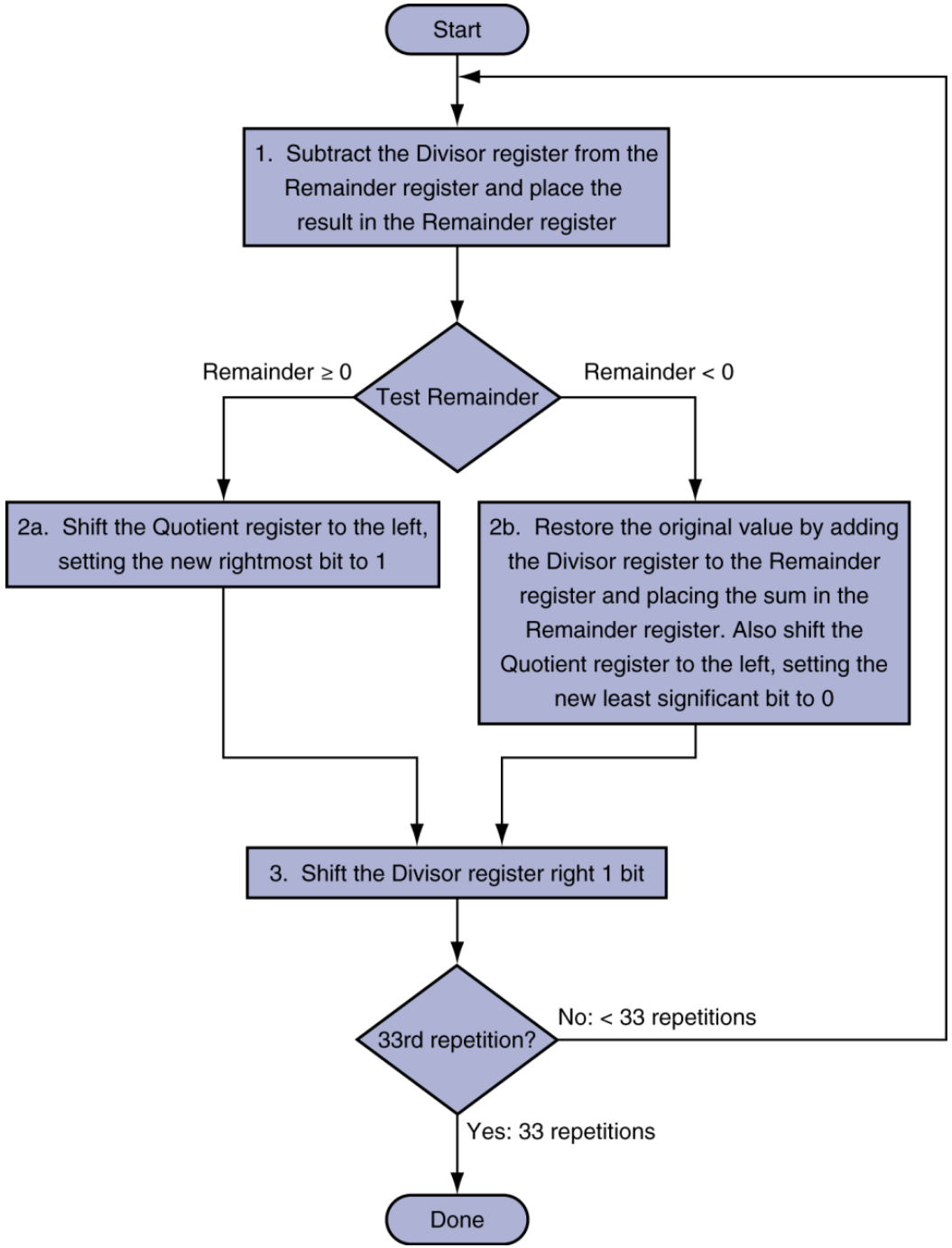
Division



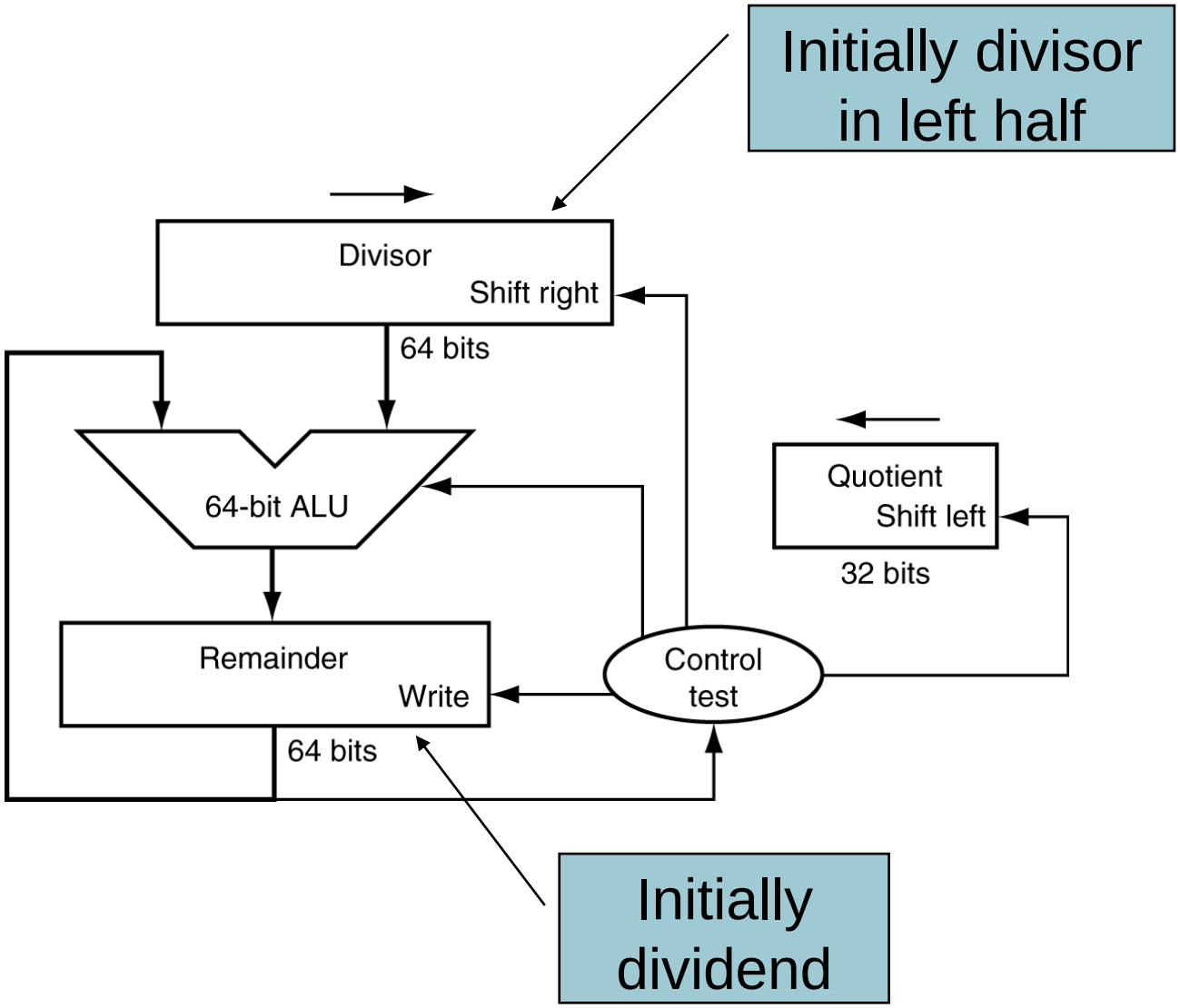
n-bit operands yield *n*-bit quotient and remainder

Division

- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

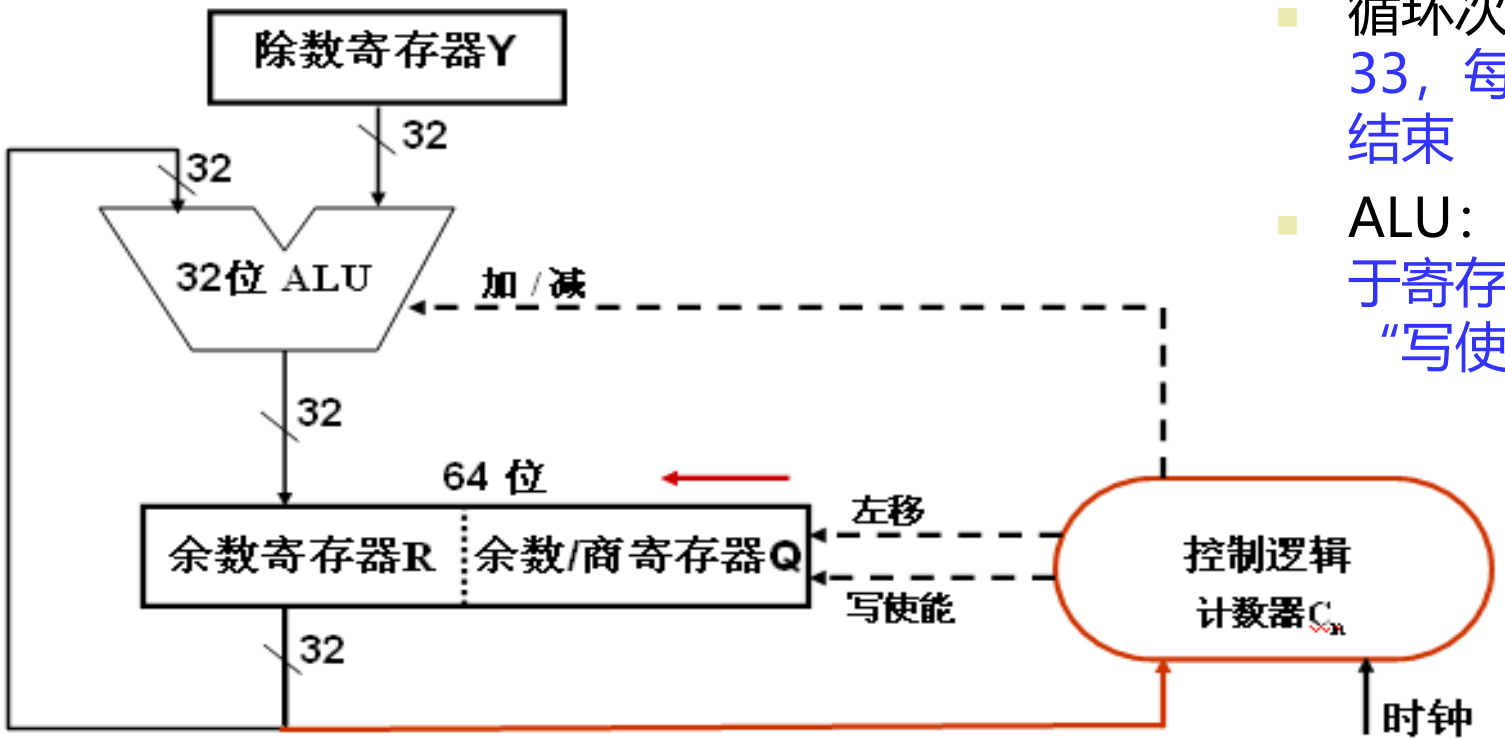

Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to 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

Division Hardware



优化后的无符号除法器

- 优化方法是将移位和减法同时进行
- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both



- 除数寄存器Y：存放除数。
- 余数寄存器R：初始时高位部分为高32位被除数；结束时是余数。
- 余数/商寄存器Q：初始时为低32位被除数；结束时是32位商。
- 循环次数计数器Cn：存放循环次数。初值是33，每循环一次，Cn减1，当Cn=0时，除法结束
- ALU：除法核心部件。在控制逻辑控制下，对于寄存器R和Y的内容进行“加/减”运算，在“写使能”控制下运算结果被送回寄存器R。

R和Q同步“左移”，Q空出位上商，商的各位逐次左移到Q中。由控制逻辑根据加减结果决定商为0还是1

减----试商，加----恢复余数。

Faster Division

- Can’t use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder
- Faster dividers (e.g. SRT devision) generate multiple quotient bits per step
  - Still require multiple steps

MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions
  - `div rs, rt  /  divu rs, rt`
  - No overflow or divide-by-0 checking
    - Software must perform checks if required
  - Use `mfhi`, `mflo` to access result

有符号除法

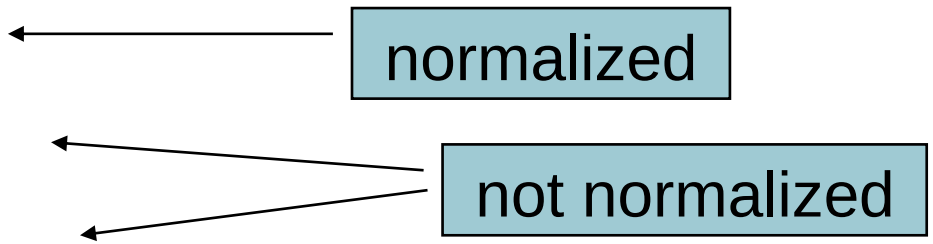
- 下面的公式必须满足：

  被除数=商 * 除数 + 余数

- (-7)/(-2): 商=(-3), 余数=(-1)
  - 商是(-4)且余数是(+1)同样满足公式，但不能取这个结果，否则商的绝对值会根据被除数和除数的符号而改变
- 正确的有符号除法在源操作数的符号相反时商为负，同时非零余数的符号与被除数相同

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C



Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

IEEE Floating-Point Format

single: 8 bits single: 23 bits
double: 11 bits double: 52 bits

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

- S: sign bit (0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ negative)

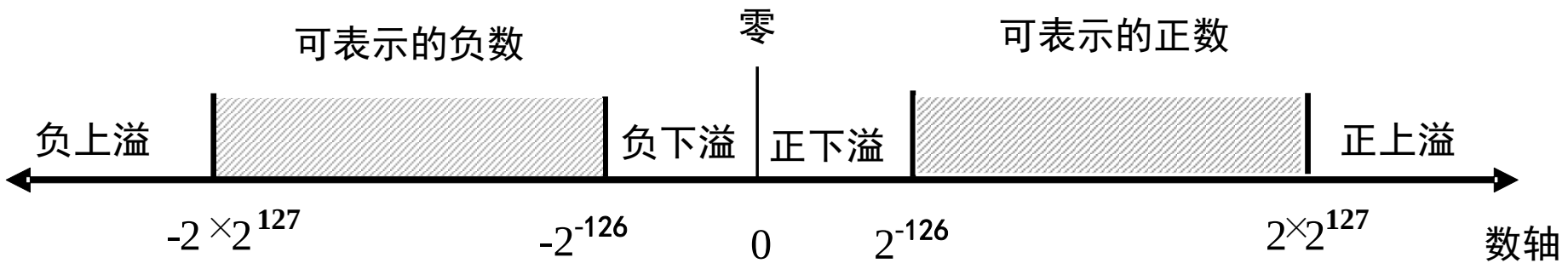
IEEE Floating-Point Format

- Normalize significand: 1.0 ≤ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1023

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Single-Precision Range



机器0：尾数为0 或 落在下溢区中间的数

浮点数范围比定点数大，但数的个数没变多，故数之间更稀疏，且不均匀

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    ⇒ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    ⇒ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - 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

Floating-Point Example

- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2
- Single: 10111111010000…00
- Double: 10111111111101000…00

Floating-Point Example

- What number is represented by the single-precision float
  1100000010100 0…00
  - S = 1
  - Fraction = $01000\ldots00_2$
  - Fxponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$
  $= (-1) \times 1.25 \times 2^2$
  $= -5.0$

Ex: Converting Decimal to FP

-12.75

1. Denormalize: -12.75
2. Convert integer part: 12 = 8 + 4 = 1100_2
3. Convert fractional part: .75 = .5 + .25 = $.11_2$
4. Put parts together and normalize:

   $1100.11 = 1.10011 \times 2^3$
5. Convert exponent: 127 + 3 = 128 + 2 = $1000\ 0010_2$

1	**1000 0010**	**100 1100 0000 0000 0000 0000**

The Hex rep. is 0xc14c0000

Floating-Point Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- 1. Align decimal points
  - Shift number with smaller exponent
  - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
- 2. Add significands
  - $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$
- 3. Normalize result & check for over/underflow
  - 1.0015×10^{2}
- 4. Round and renormalize if necessary
  - 1.002×10^{2}

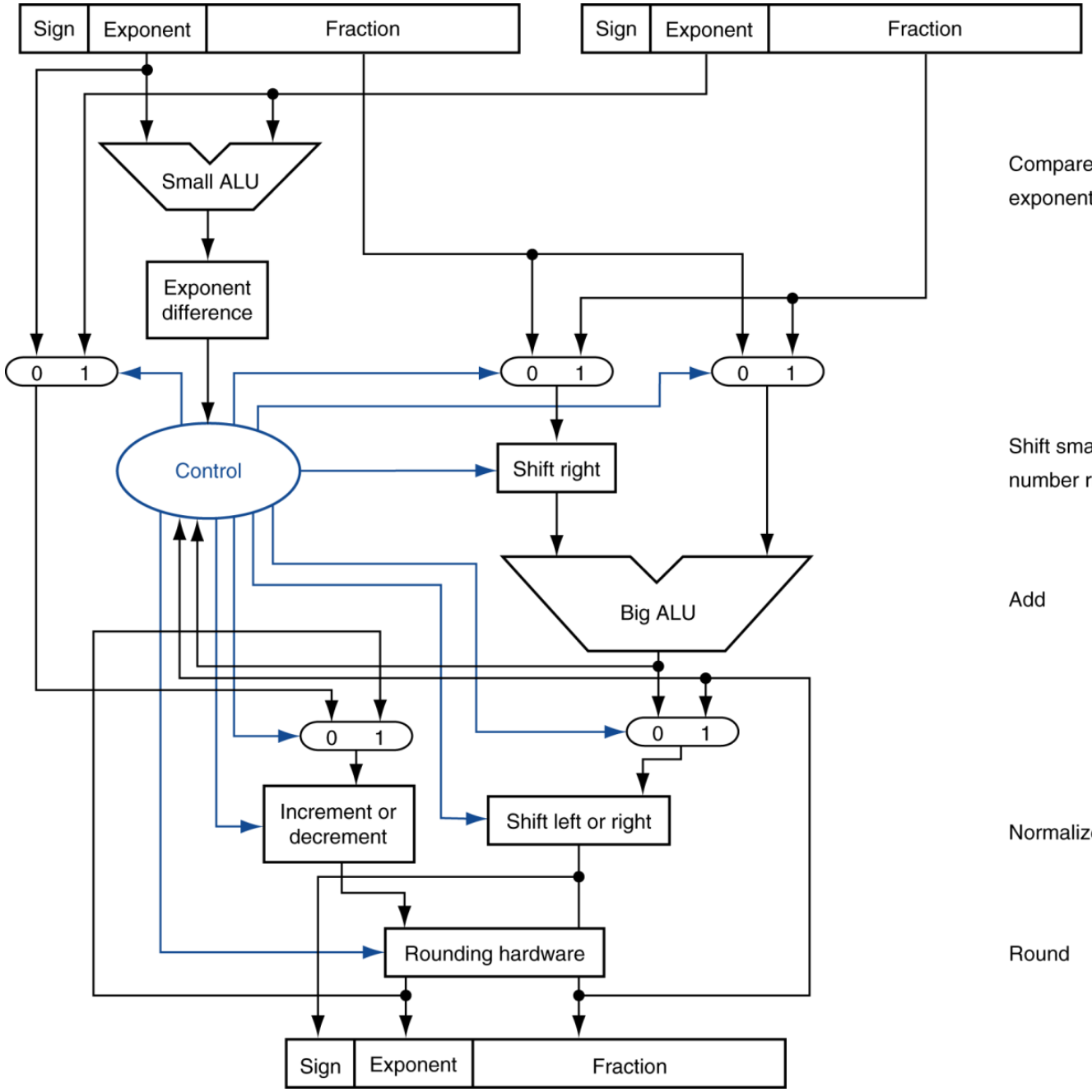
Floating-Point Addition

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-}1 = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined

FP Adder Hardware



Compare exponents

Shift smaller number right

Step 1

Add

Step 2

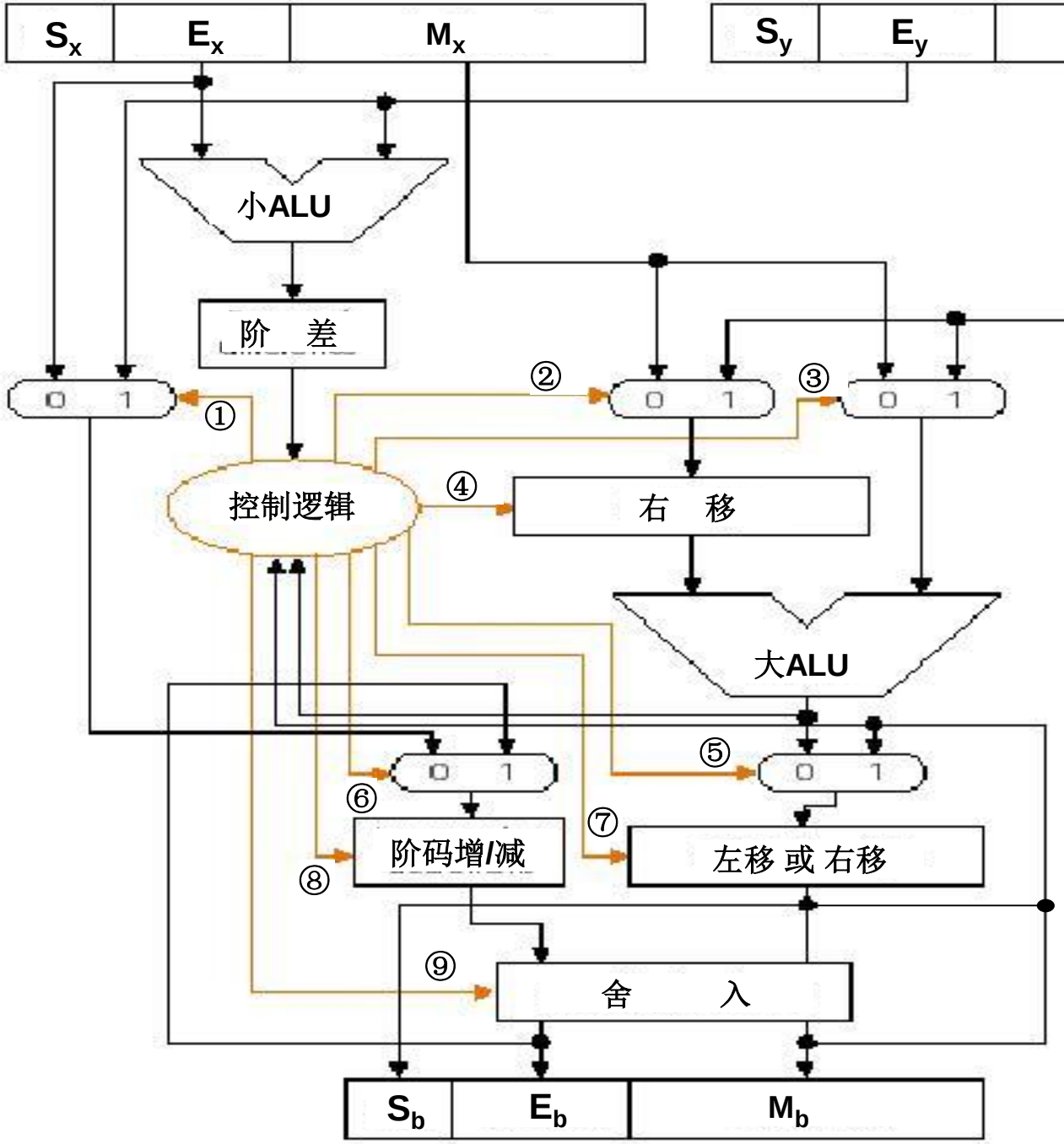
Normalize

Step 3

Round

Step 4

浮点加/减法器



阶码相减

阶小的数的尾数右移

尾数加/减

规格化

舍入

可用流水线方式实现！

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP $\leftrightarrow$ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
    - Release 2 of MIPS ISA supports 32 × 64-bit FP reg's

FP Instructions in MIPS

- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`

FP Instructions in MIPS

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, div.s
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`

FP Instructions in MIPS

- Single- and double-precision comparison
  - `c.`*xx*`.s`, `c.`*xx*`.d` (*xx* is `eq`, neq, `lt`, `le`, gt, ge)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
  - `bc1t`, `bc1f`
    - e.g., `bc1t TargetLabel`

FP Example: °F to °C

- C code:

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

  - `fahr` in $f12, result in $f0, literals in global memory space

- Compiled MIPS code:

```
f2c: lwc1  $f16, const5($gp)
     lwc2  $f18, const9($gp)
     div.s $f16, $f16, $f18
     lwc1  $f18, const32($gp)
     sub.s $f18, $f12, $f18
     mul.s $f0,  $f16, $f18
     jr    $ra
```

FP Example: Array Multiplication

- X = X + Y × Z
  - All 32 × 32 matrices, 64-bit double-precision elements
- C code:

```
void mm (double x[][],
         double y[][], double z[][]) {
  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)
        x[i][j] = x[i][j]
                  + y[i][k] * z[k][j];
}
```

  - Addresses of x, y, z in $a0, $a1, $a2, and i, j, k in $s0, $s1, $s2

FP Example: Array Multiplication

- MIPS code:

```
    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 x)
    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 x[i][j]
    l.d  $f4, 0($t2)    # $f4 = 8 bytes of x[i][j]
L3: sll  $t0, $s2, 5    # $t0 = k * 32 (size of row of z)
    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 z[k][j]
    l.d  $f16, 0($t0)   # $f16 = 8 bytes of z[k][j]
    …
```

FP Example: Array Multiplication

…

```
    sll  $t0, $s0, 5       # $t0 = i*32 (size of row of y)
    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 y[i][k]
    l.d   $f18, 0($t0)     # $f18 = 8 bytes of y[i][k]
    mul.d $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
    add.d $f4, $f4, $f16   # f4=x[i][j] + y[i][k]*z[k][j]
    addiu $s2, $s2, 1      # $k k + 1
    bne   $s2, $t1, L3     # if (k != 32) go to L3
    s.d   $f4, 0($t2)      # x[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
```

Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
  - Extra bits of precision (guard, round, sticky)
  - Choice of rounding modes
  - Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
  - Most programming languages and FP libraries just use defaults
- Trade-off between hardware complexity, performance, and market requirements

浮点数比较运算举例

- 对于以下给定的关系表达式，判断是否永真

```
int x ;
float f ;
double d ;
```

Assume neither d nor f is NaN

自己写程序测试一下！

x == (int)(float) x	否
x == (int)(double) x	是
f == (float)(double) f	是
d == (float) d	否
f == -(-f);	是
2/3 == 2/3.0	否
d < 0.0 ⇒ ((d*2) < 0.0)	是
d > f ⇒ -f > -d	是
d * d >= 0.0	是
x*x>=0	否
(d+f)-d == f	否

例子：Ariana火箭爆炸

- 1996年6月4日，Ariana 5火箭初次航行，在发射仅仅37秒钟后，偏离了飞行路线，然后解体爆炸，火箭上载有价值5亿美元的通信卫星。
- 原因是在将一个64位浮点数转换为16位带符号整数时，产生了溢出异常。溢出的值是火箭的水平速率，这比原来的Ariana 4火箭所能达到的速率高出了5倍。在设计Ariana 4火箭软件时，设计者确认水平速率决不会超出一个16位的整数，但在设计Ariana 5时，他们没有重新检查这部分，而是直接使用了原来的设计。
- 在不同数据类型之间转换时，往往隐藏着一些不容易被察觉的错误，这种错误有时会带来重大损失，因此，编程时要非常小心。

例子：爱国者导弹定位错误

- 1991年2月25日，海湾战争中，美国在沙特阿拉伯达摩地区设置的爱国者导弹拦截伊拉克的飞毛腿导弹失败，致使飞毛腿导弹击中了一个美军军营，杀死了美军28名士兵。其原因是由于爱国者导弹系统时钟内的一个软件错误造成的，引起这个软件错误的原因是浮点数的精度问题。
- 爱国者导弹系统中有一内置时钟，用计数器实现，每隔0.1秒计数一次。程序用0.1的一个24位定点二进制小数x来乘以计数值作为以秒为单位的时间
- 这个x的机器数是多少呢?
- 0.1的二进制表示是一个无限循环序列：0.00011[0011]…，x=0.000 1100 1100 1100 1100 1100B。显然，x是0.1的近似表示，0.1-x

  = 0.000 1100 1100 1100 1100 1100 [1100]… - 0.000 1100 1100 1100 1100 1100B，即为：

  =0.000 0000 0000 0000 0000 0000 1100 [1100]…B

  $=2^{-20}\times 0.1 \approx 9.54\times 10^{-8}$

  这就是机器值与真值之间的误差！

例子：爱国者导弹定位错误

- 已知在爱国者导弹准备拦截飞毛腿导弹之前，已经连续工作了100小时，飞毛腿的速度大约为2000米/秒，则由于时钟计算误差而导致的距离误差是多少？
  - 100小时相当于计数了$100\times60\times60\times10=36\times10^5$次，因而导弹的时钟已经偏差了$9.54\times10^{-8}\times36\times10^5 \approx 0.343$秒
  - 因此，距离误差是2000×0.343秒 ≈ 687米

小故事：实际上，以色列方面已经发现了这个问题并于1991年2月11日知会了美国陆军及爱国者计划办公室（软件制造商）。以色列方面建议重新启动爱国者系统的电脑作为暂时解决方案，可是美国陆军方面却不知道每次需要间隔多少时间重新启动系统一次。1991年2月16日，制造商向美国陆军提供了更新软件，但这个软件最终却在飞毛腿导弹击中军营后的一天才运抵部队。

例子：爱国者导弹定位错误

- 若x用float型表示，则x的机器数是什么？0.1与x的偏差是多少？系统运行100小时后的时钟偏差是多少？在飞毛腿速度为2000米/秒的情况下，预测的距离偏差为多少？
  - 0.1= 0.0 0011[0011]B=+1.1 0011 0011 0011 0011 0011 00B×2^{-4}，故x的机器数为0 011 1101 1 100 1100 1100 1100 1100 1100
  - Float型仅24位有效位数，后面的有效位全被截断，故x与0.1之间的误差为：|x–0.1|=0.000 0000 0000 0000 0000 0000 0000 1100 [1100]...B。这个值等于$2^{-24}\times0.1 \approx 5.96\times10^{-9}$。100小时后时钟偏差$5.96\times10^{-9}\times36\times10^{5} \approx 0.0215$秒。距离偏差0.0215×2000≈43米。比爱国者导弹系统精确约16倍。

例子：爱国者导弹定位错误

- 若用32位二进制定点小数x=0.000 1100 1100 1100 1100 1100 1100 1100 1101 B表示0.1，则误差比用float表示误差更大还是更小？
  - 当x=0.000 1100 1100 1100 1100 1100 1100 1101 B时，与0.1之间的误差约为：|x–0.1|=0.000 0000 0000 0000 0000 0000 0000 0000 00 1100 [1100]...B。这个值等于$2^{-30}\times0.1 \approx 9.31\times10^{-11}$。100小时后时钟偏差$9.31\times10^{-11}\times36\times10^{5} \approx 0.000335$秒。预测的距离偏差仅为$0.000335\times2000 \approx$ 0.67米。

浮点数运算的精度问题

- 从上述结果可以看出：
  - 用32位定点小数表示0.1，比采用float精度高64倍
  - 用float表示在计算速度上更慢，必须先把计数值转换为IEEE 754格式浮点数，然后再对两个IEEE 754格式的数相乘，故采用float比直接将两个二进制数相乘要慢
- Ariana 5火箭和爱国者导弹的例子带来的启示
  - 程序员应对底层机器级数据的表示和运算有深刻理解
  - 计算机世界里，经常是“差之毫厘，失之千里”，需要细心再细心，精确再精确
  - 不能遇到小数就用浮点数表示，有些情况下（如需要将一个整数变量乘以一个确定的小数常量），可先用一个确定的定点整数与整数变量相乘，然后再通过移位运算来确定小数点

Subword Parallellism

- Graphics and audio applications can take advantage of performing simultaneous operations on short vectors
  - Example: 128-bit adder:
    - Sixteen 8-bit adds
    - Eight 16-bit adds
    - Four 32-bit adds
- Also called data-level parallelism, vector parallelism, or Single Instruction, Multiple Data (SIMD)

x86 FP Architecture

- Originally based on 8087 FP coprocessor
  - 8 × 80-bit extended-precision registers
  - Used as a push-down stack
  - Registers indexed from TOS: ST(0), ST(1), …
- FP values are 32-bit or 64 in memory
  - Converted on load/store of memory operand
  - Integer operands can also be converted on load/store
- Very difficult to generate and optimize code
  - Result: poor FP performance

x86 FP Instructions

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

- Optional variations
  - I: integer operand
  - P: pop operand from stack
  - R: reverse operand order
  - But not all combinations allowed

Streaming SIMD Extension 2 (SSE2)

- Adds 4 × 128-bit registers
  - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
  - 2 × 64-bit double precision
  - 4 × 32-bit double precision
  - Instructions operate on them simultaneously
    - Single-Instruction Multiple-Data

Matrix Multiply

- Unoptimized code:

```
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. }
```

Matrix Multiply

- Optimized C code:

```
1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double* C)
3. {
4. for (int i = 0; i < n; i+=8)
5.    for (int j = 0; j < n; ++j)
6.     {
7.       __m512d c0 =  _mm512_load_pd(C+i+j*n); // c0 = C[i][j]
8.          for( int k = 0; k < n; k++ )
9.            {  // c0 += A[i][k]*B[k][j]
10.            __m512d bb = _mm512_broadcastsd_pd(_mm_load_sd(B+j*n+k));
11.            c0 = _mm512_fmadd_pd(_mm512_load_pd(A+n*k+i), bb, c0);
12.           }
13.       _mm512_store_pd(C+i+j*n, c0); // C[i][j] = c0
14.    }
15.}
```

Matrix Multiply

- Optimized x86 assembly code:

```
vmovapd (%r11),%zmm1                    # Load 8 elements of C into %zmm1
mov     %rbx,%rcx                       # register %rcx = %rbx
xor     %eax,%eax                       # register %eax = 0
vbroadcastsd (%rax,%r8,8),%zmm0         # Make 8 copies of B element in %zmm0
add     $0x8,%rax                       # register %rax = %rax + 8
vfmadd231pd   (%rcx),%zmm0,%zmm1        # Parallel mul & add %zmm0, %zmm1
add     %r9,%rcx                        # register %rcx = %rcx
cmp     %r10,%rax                       # compare %r10 to %rax
jne     50 <dgemm+0x50>                 # jump if not %r10 != %rax
add     $0x1, %esi                      # register % esi = % esi + 1
vmovapd %zmm1, (%r11)                   # Store %zmm1 into 8 C elements
```

Right Shift and Division

- Left shift by *i* places multiplies an integer by 2^i
- Right shift divides by 2^i?
  - Only for unsigned integers
- For signed integers
  - Arithmetic right shift: replicate the sign bit
  - e.g., –5 / 4
    - 11111011_2 >> 2 = 11111110_2 = –2
  - c.f. 11111011_2 >>> 2 = 00111110_2 = +62

Associativity

- Parallel programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail

		(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

- Need to validate parallel programs under varying degrees of parallelism

Who Cares About FP Accuracy?

- Important for scientific code
  - But for everyday consumer use?
    - “My bank balance is out by 0.0002¢!” ☹
- The Intel Pentium FDIV bug
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*

Concluding Remarks

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs

Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow
- MIPS ISA
  - Core instructions: 54 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent

作业

- 3.16, 3.19, 3.23, 3.24