Chapter 3

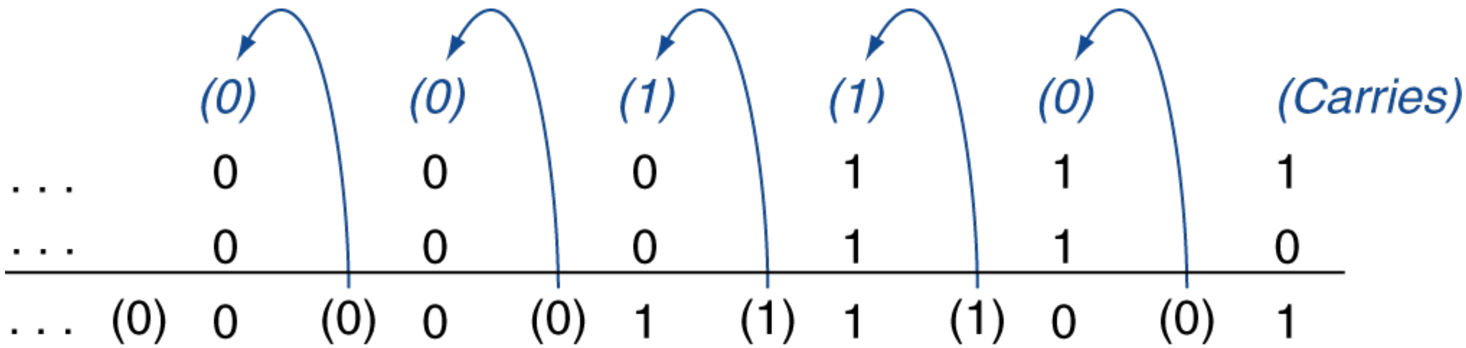
Arithmetic for Computers

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 Addition Example 1

- Consider adding the numbers 7 and 6 represented in 2’s complement using 4 bits. What is the result of the computation?

Integer Addition Example 1

- Consider adding the numbers 7 and 6 represented in 2's complement using 4 bits. What is the result of the computation?

  7: 0 1 1 1

  6: 0 1 1 0

  ----

  1 1 0 1 Result is negative (-3)!

  Overflow.

Integer Addition Example 2

- Consider adding the numbers -7 and -6 represented in 2’s complement using 4 bits. What is the result of the computation?

Integer Addition Example 2

- Consider adding the numbers -7 and -6 represented in 2's complement using 4 bits. What is the result of the computation?

7 → -7: 0 1 1 1 → 1 0 0 0 → 1 0 0 1

6 → -6: 0 1 1 0 → 1 0 0 1 → 1 0 1 0

----

0 0 1 1

Result is positive (3)! Overflow.

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

Integer Subtraction Example 1

- Consider subtracting 7 from -6 assuming that the numbers are represented in 2's complement using 4 bits. What is the result of the computation?

Integer Subtraction Example 1

- Consider subtracting 7 from -6 assuming that the numbers are represented in 2's complement using 4 bits. What is the result of the computation?

```
-6:   1 0 1 0
-7:   1 0 0 1
    ------------
      0 0 1 1   Result is positive (3)!
```

Overflow.

Integer Subtraction Example 2

- Consider subtracting -7 from 6 assuming that the numbers are represented in 2's complement using 4 bits. What is the result of the computation?

Integer Subtraction Example 2

- Consider subtracting -7 from 6 assuming that the numbers are represented in 2's complement using 4 bits. What is the result of the computation?

6 – (-7) = 6 + 7

6: 0 1 1 0

7: 0 1 1 1

----

1 0 0 1 The result is negative (-3). Overflow.

When Overflow Occurs

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

Dealing with Overflow

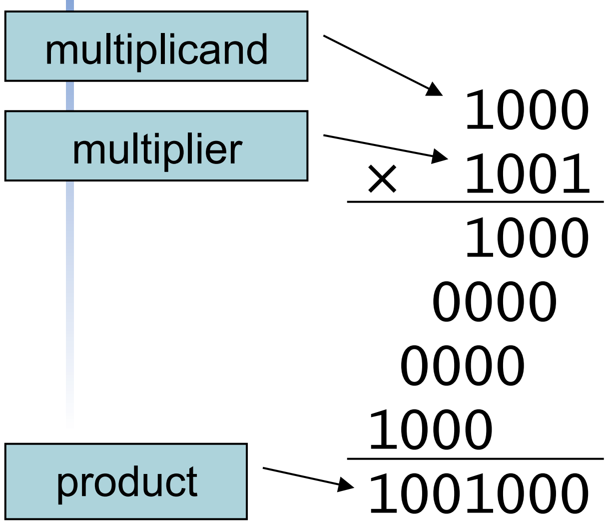
- Some languages (e.g., C) ignore overflow
  - Use MIPS `addu`, `addui`, `subu` instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
  - Use MIPS `add`, `addi`, `sub` instructions
  - On overflow, invoke exception handler
    - Save PC in exception program counter (EPC) register
    - Jump to predefined handler address
    - `mfc0` (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action

Arithmetic for Multimedia

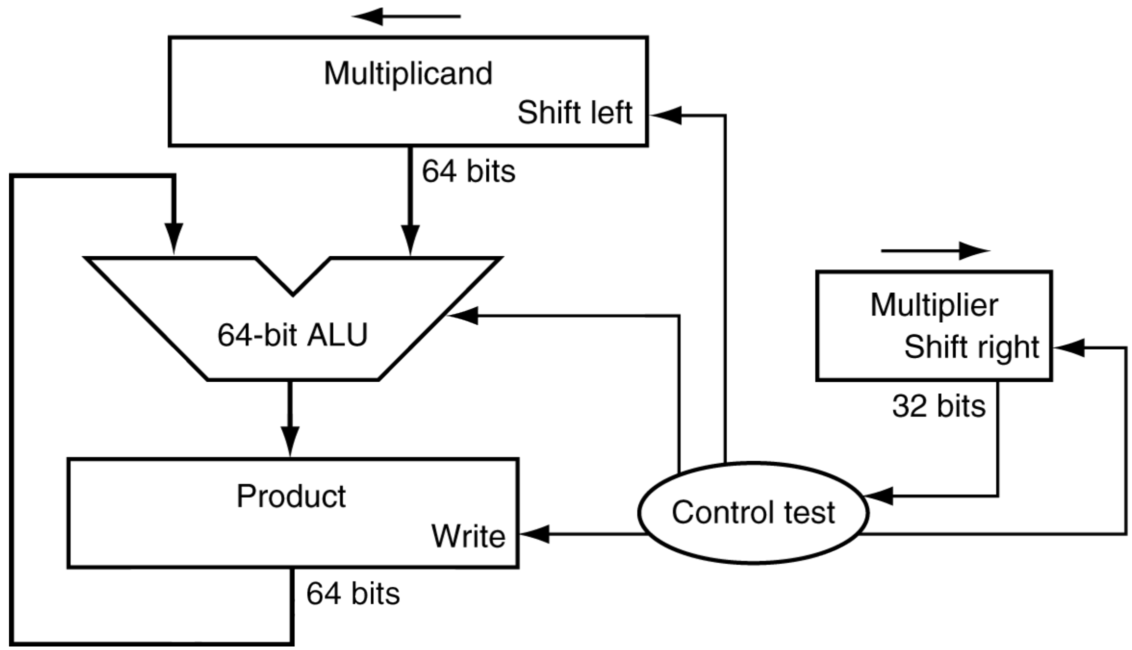
- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
  - SIMD (single-instruction, multiple-data)
- Saturating operations
  - On overflow, result is largest representable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

Multiplication

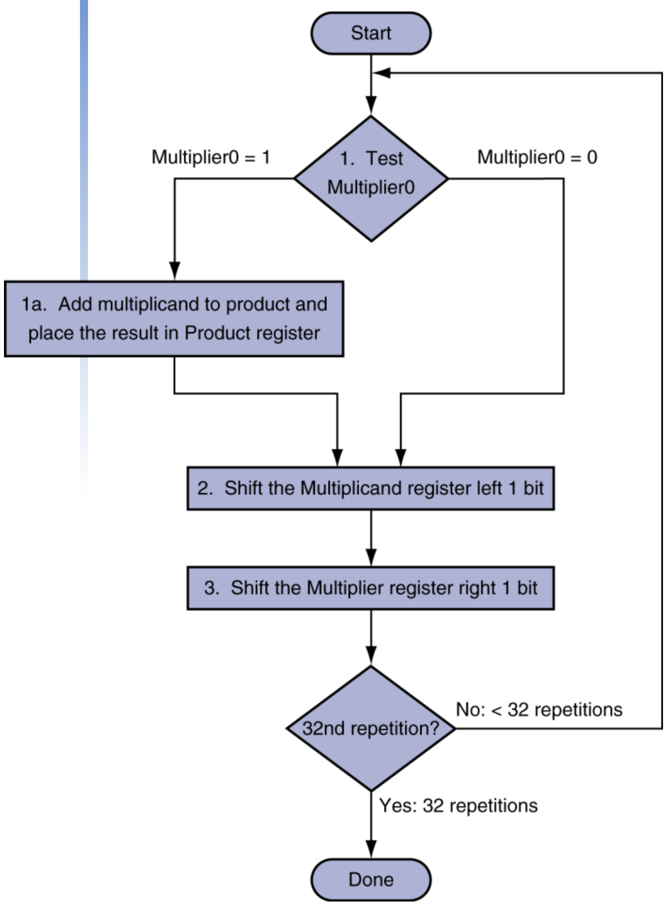
- Start with long-multiplication approach

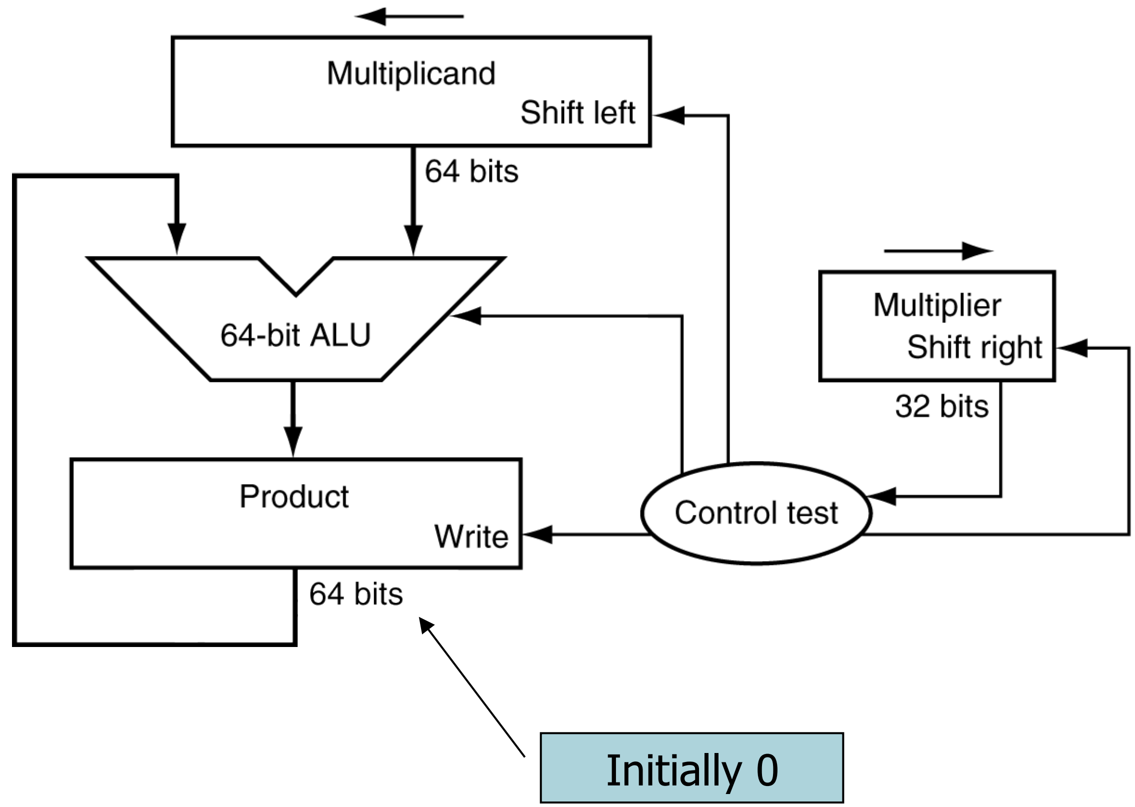


Length of product is the sum of operand lengths



Multiplication Hardware





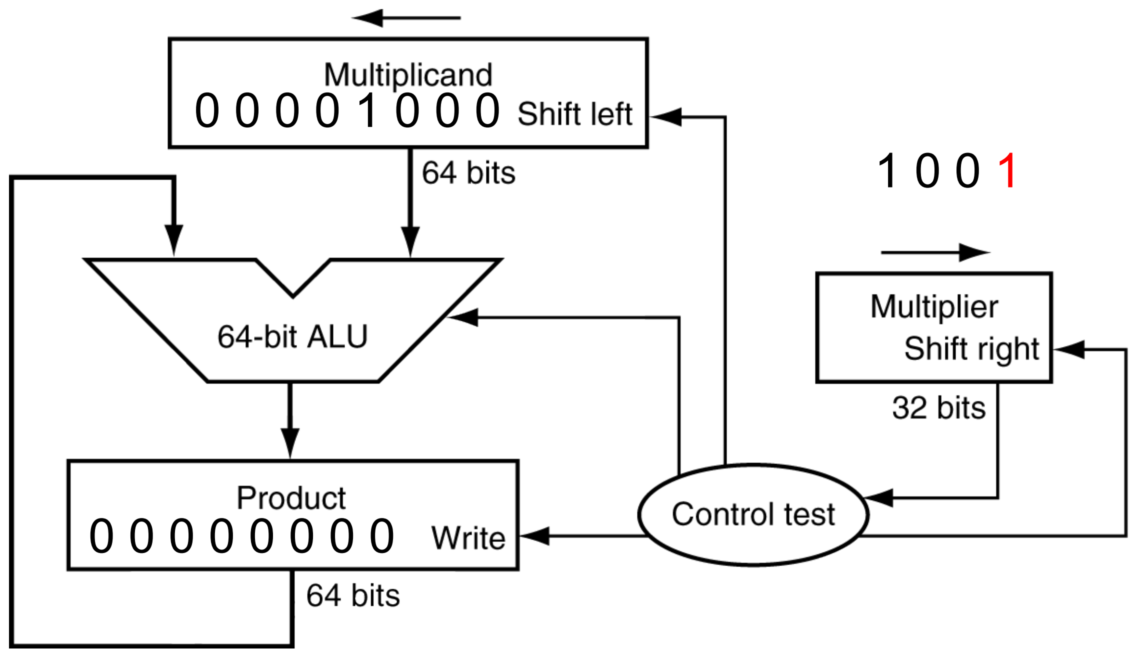
Multiplication

- Start with long-multiplication approach

multiplicand → 1000
multiplier → × 1001

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

product → 1001000

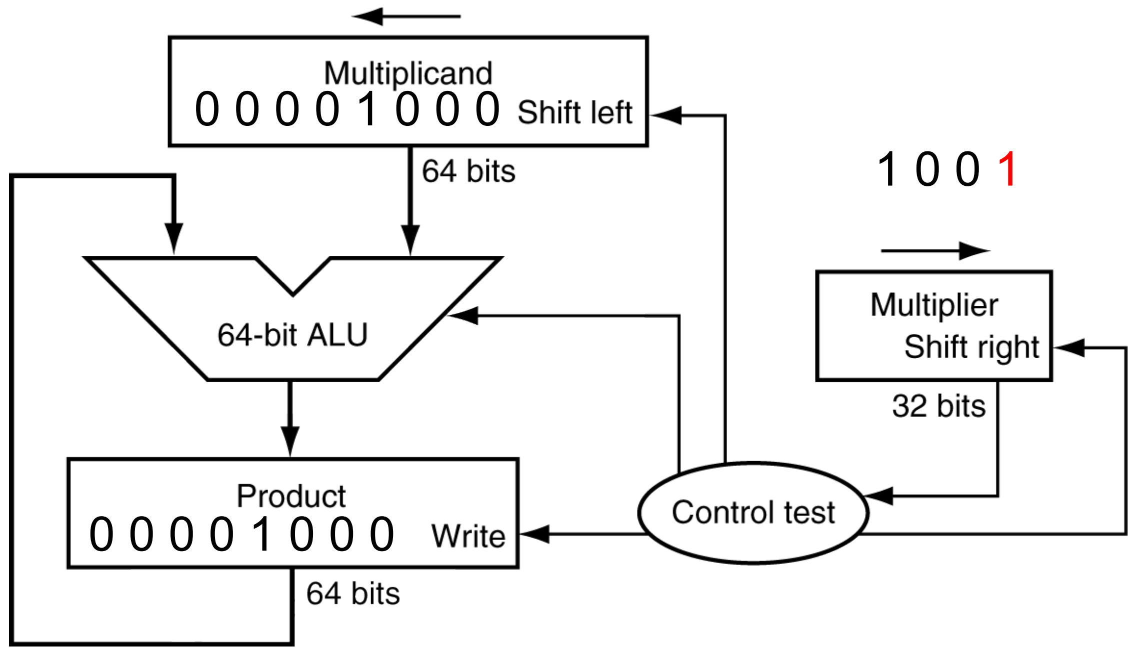


Length of product is the sum of operand lengths

Multiplication

- Start with long-multiplication approach

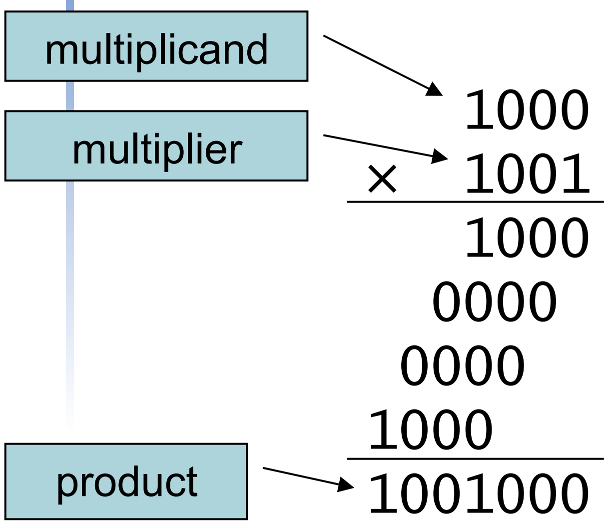
```
multiplicand        1000
multiplier        × 1001
                  ------
                    1000
                   0000
                  0000
                 1000
                 -------
product          1001000
```

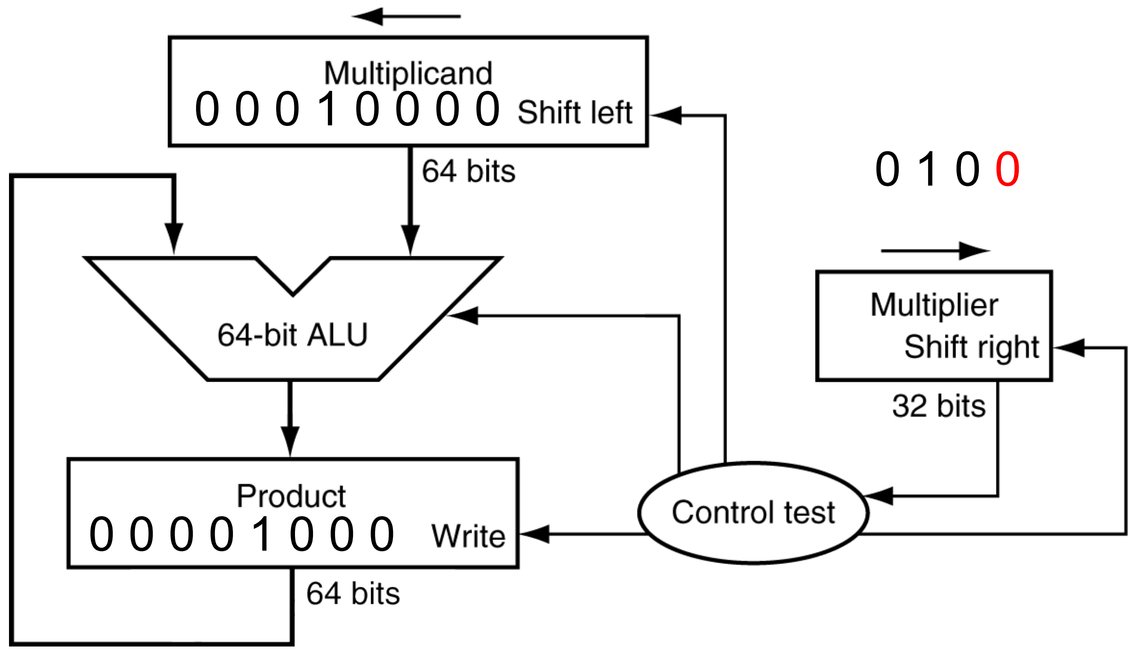


Length of product is the sum of operand lengths

Multiplication

- Start with long-multiplication approach





Length of product is the sum of operand lengths

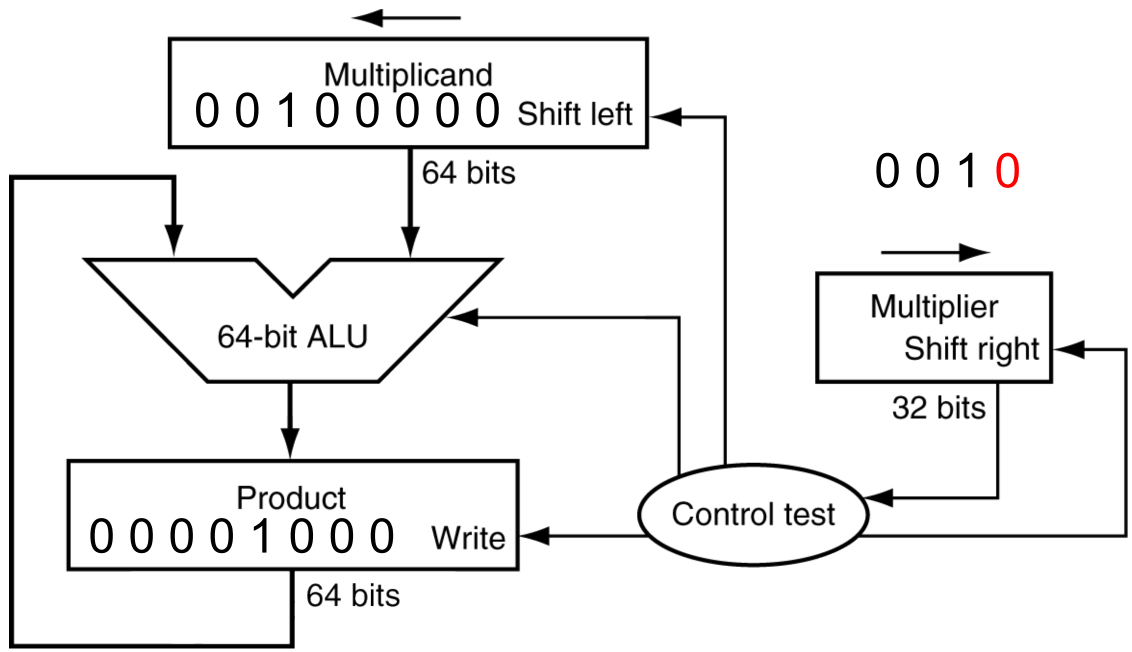
Multiplication

- Start with long-multiplication approach

multiplicand, multiplier, product

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

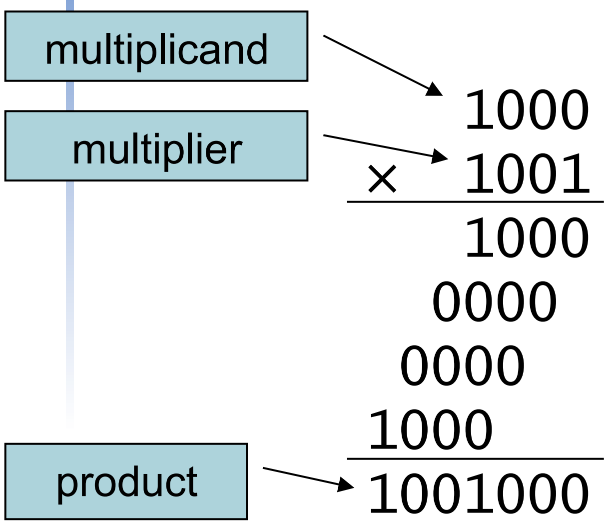
Length of product is the sum of operand lengths



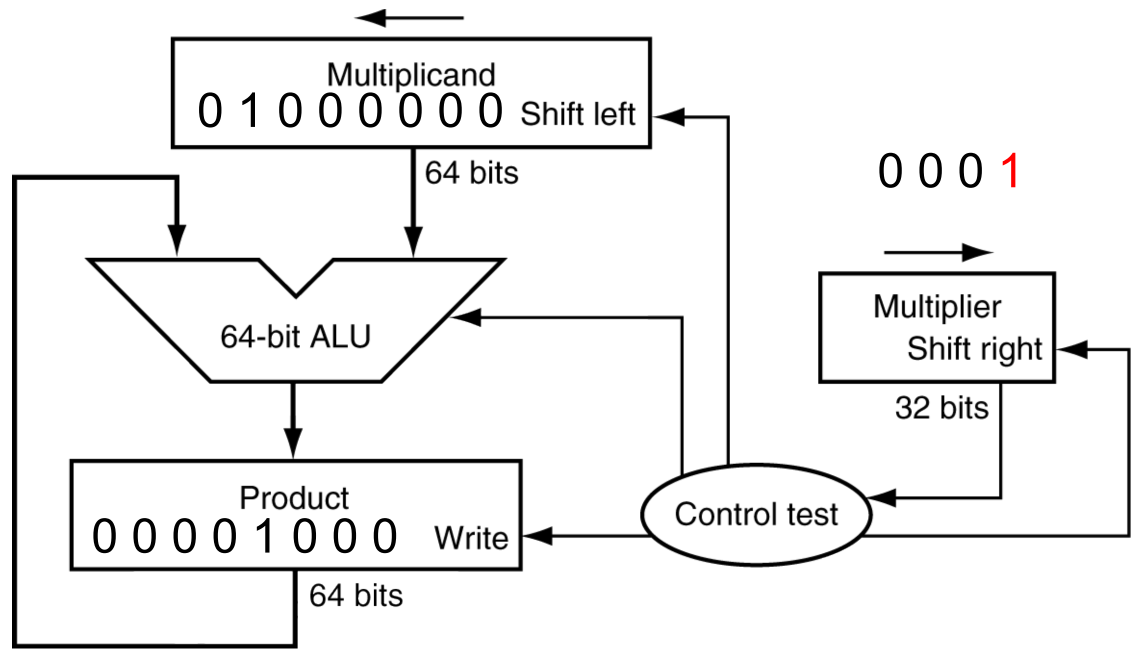
§3.3 Multiplication

Multiplication

- Start with long-multiplication approach

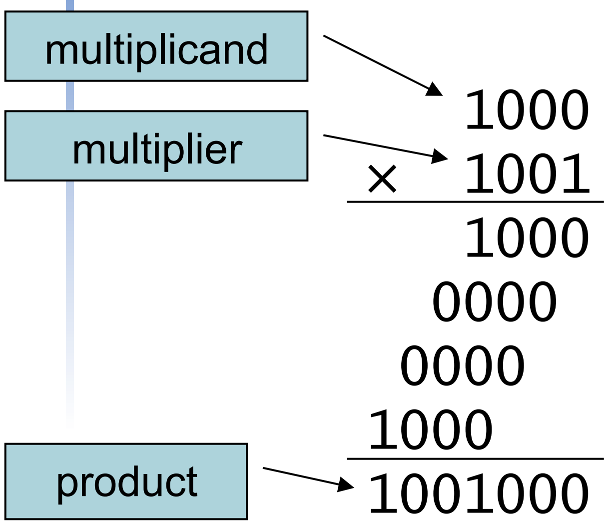


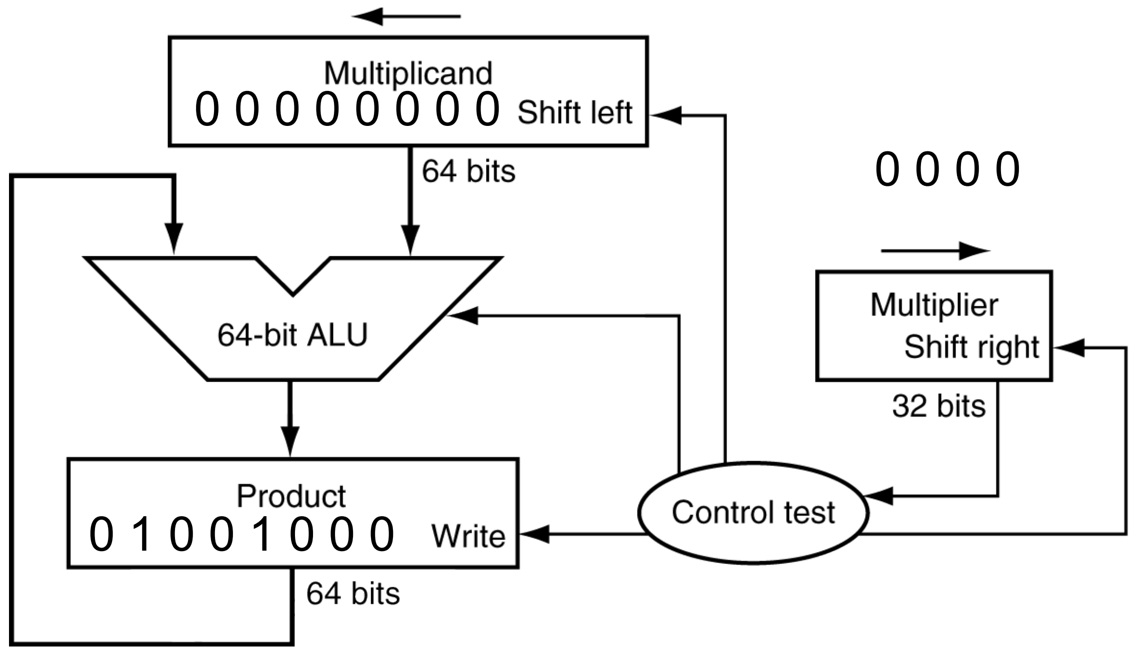
Length of product is the sum of operand lengths



Multiplication

- Start with long-multiplication approach

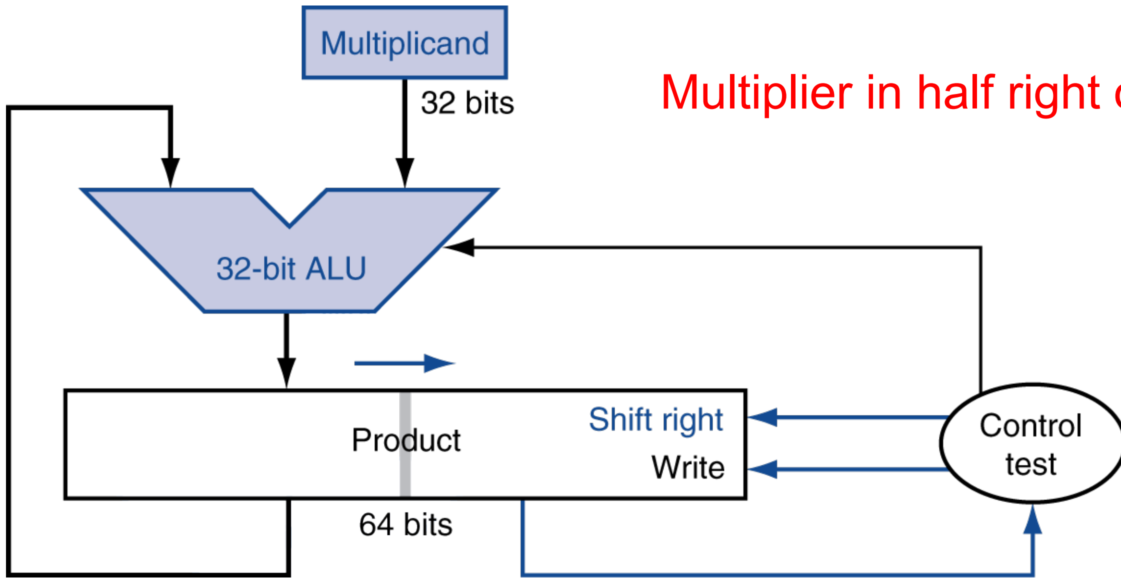




Length of product is the sum of operand lengths

Optimized Multiplier

- Perform steps in parallel: add/shift

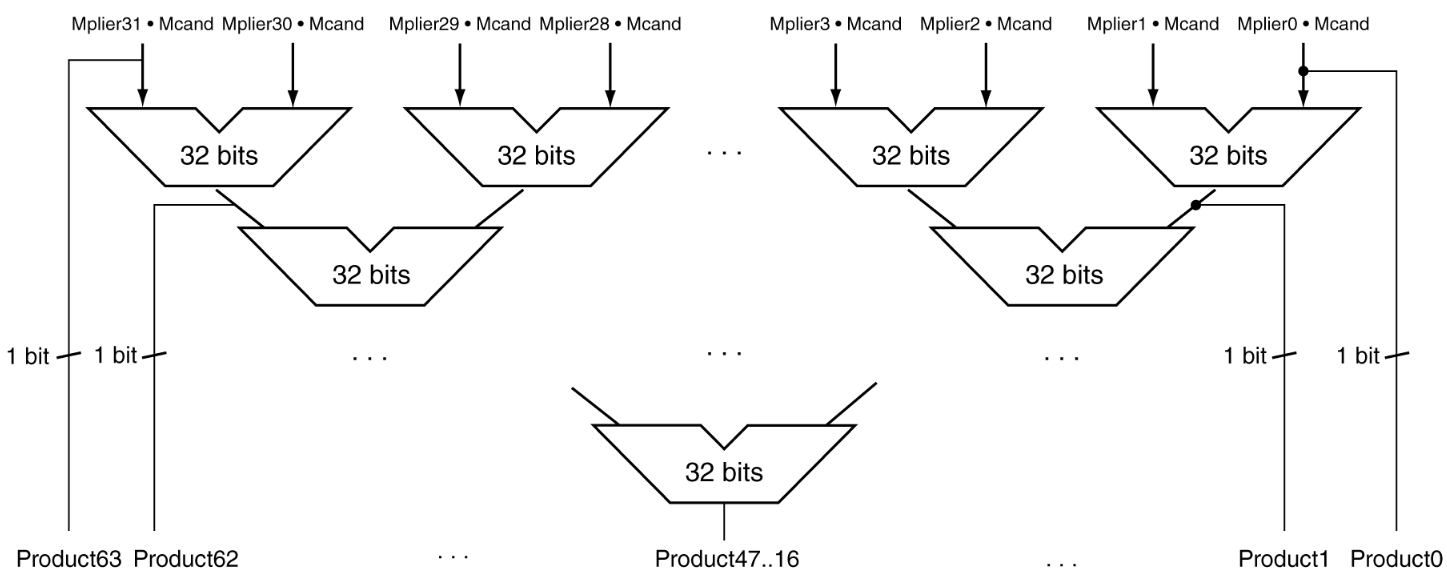


Multiplier in half right of Product register

- One cycle per partial-product addition
  - That's ok, if frequency of multiplications is low

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff



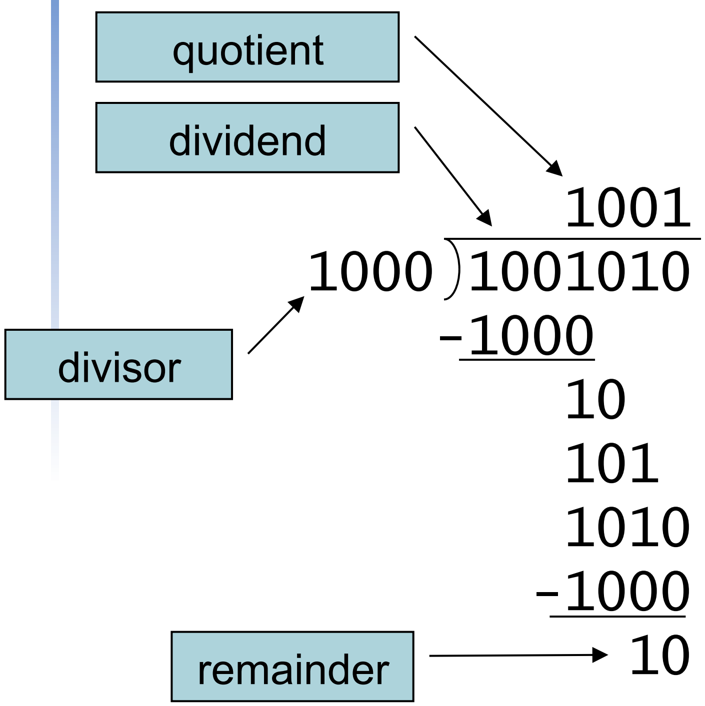
- Can be pipelined
  - Several multiplications performed in parallel

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

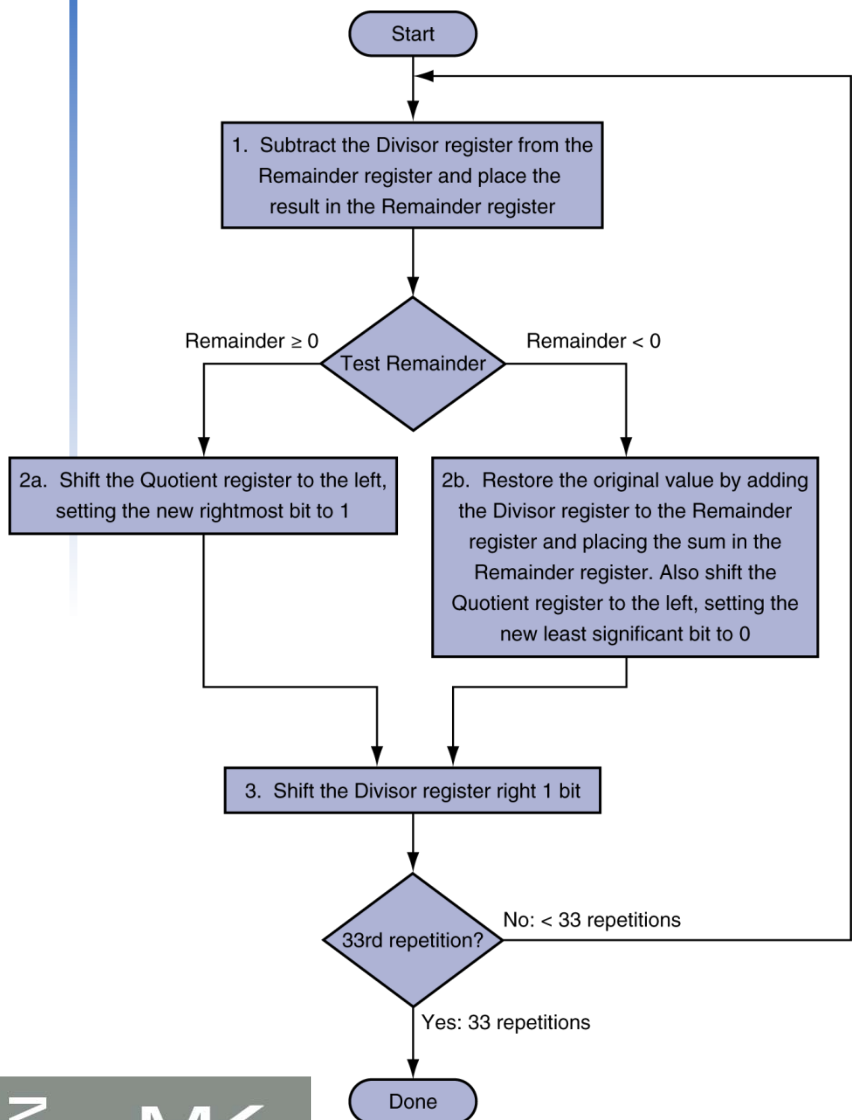
Grammar school algorithm:
Try to see how big a number can be subtracted, creating a digit of the quotient on each attempt.

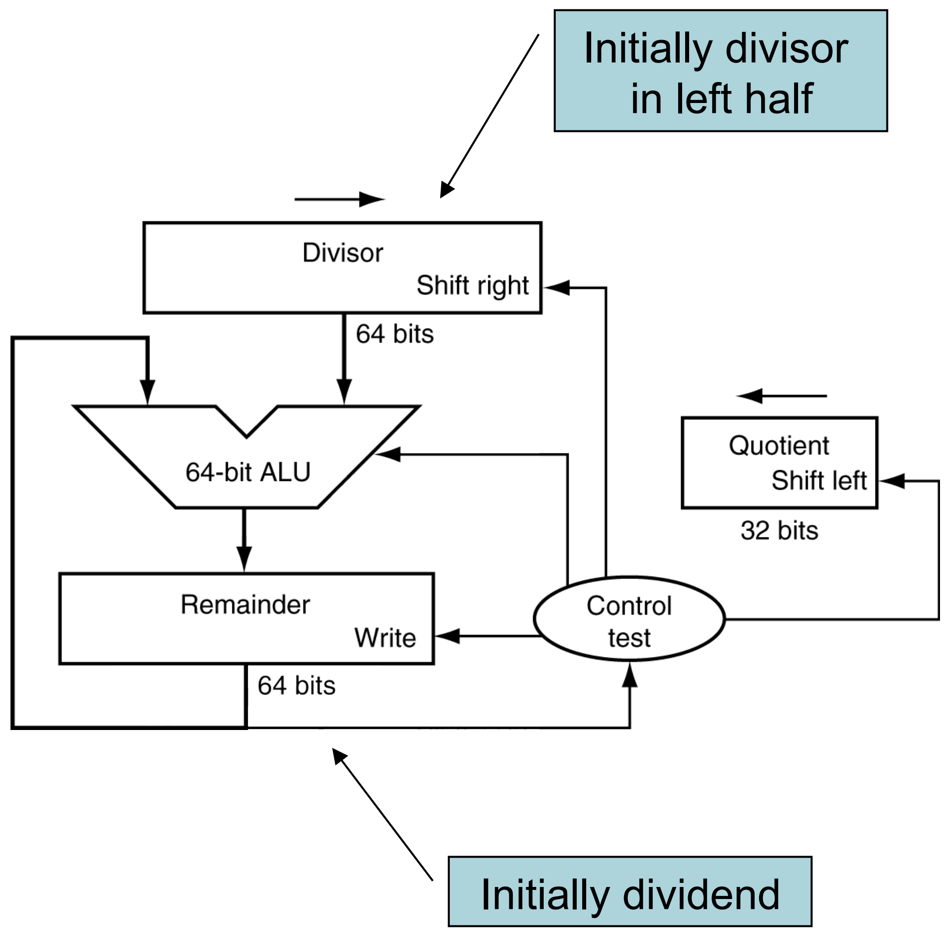


n-bit operands yield *n*-bit quotient and remainder

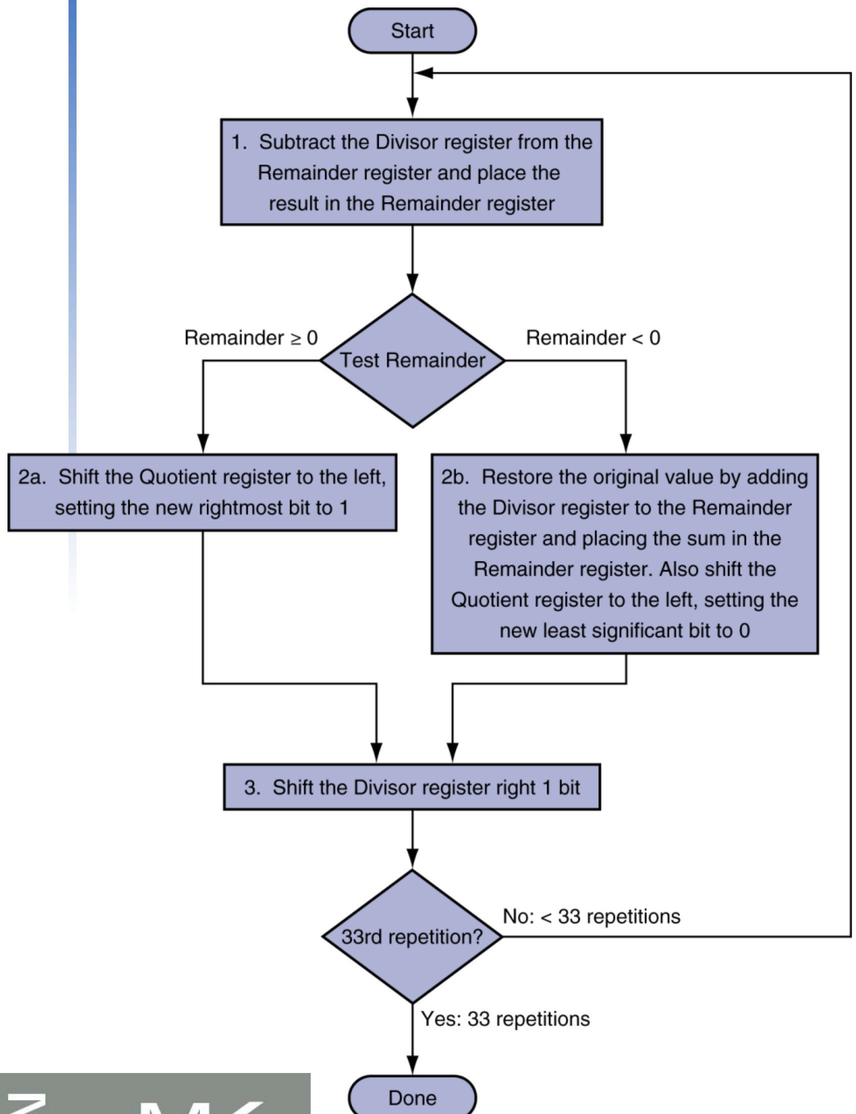
- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using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

Division Hardware



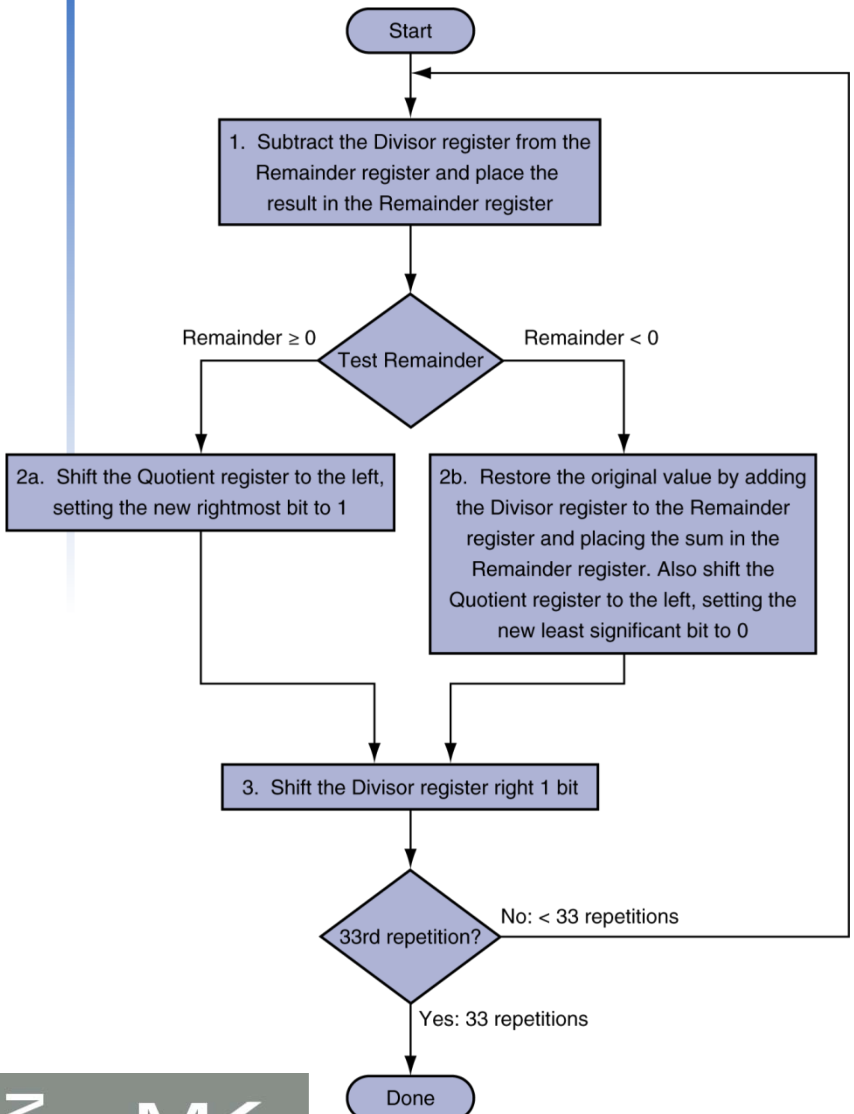


Division Hardware



divisor 12) 85 *Dividend (initially = remainder)*

Division Hardware



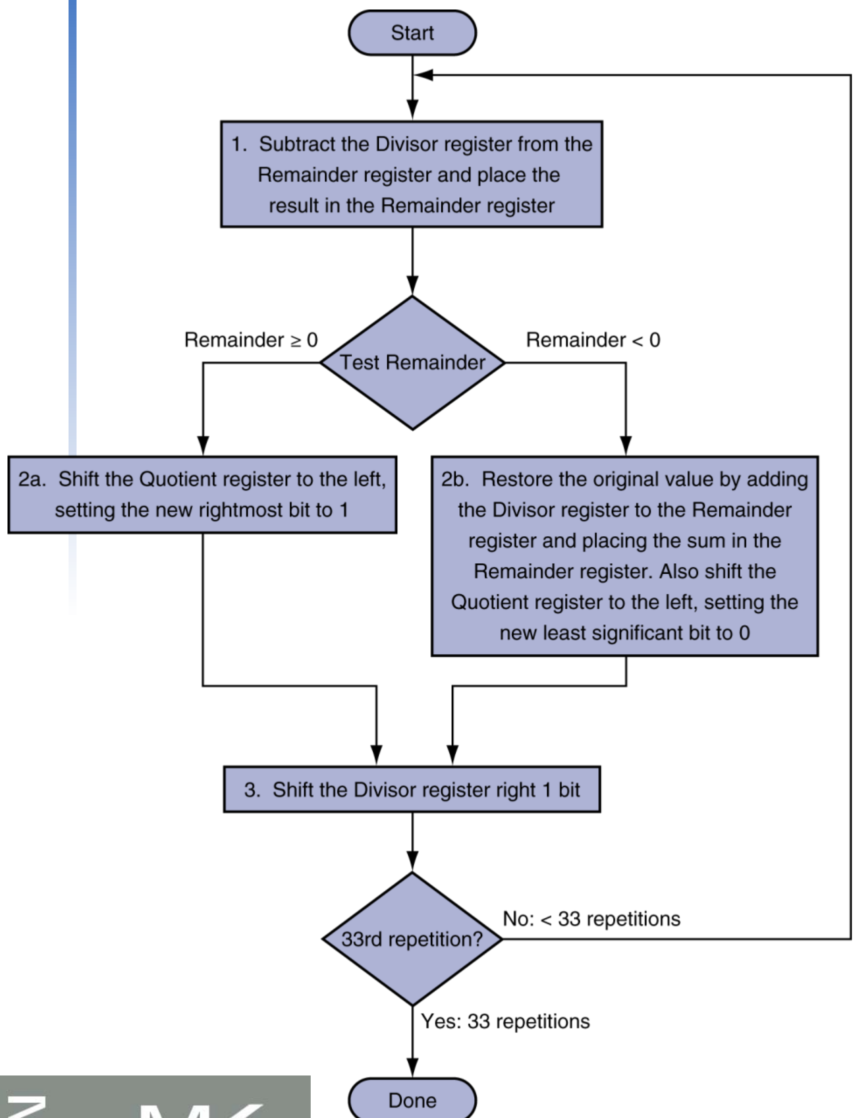
1

divisor 12) 85 *dividend*

-12

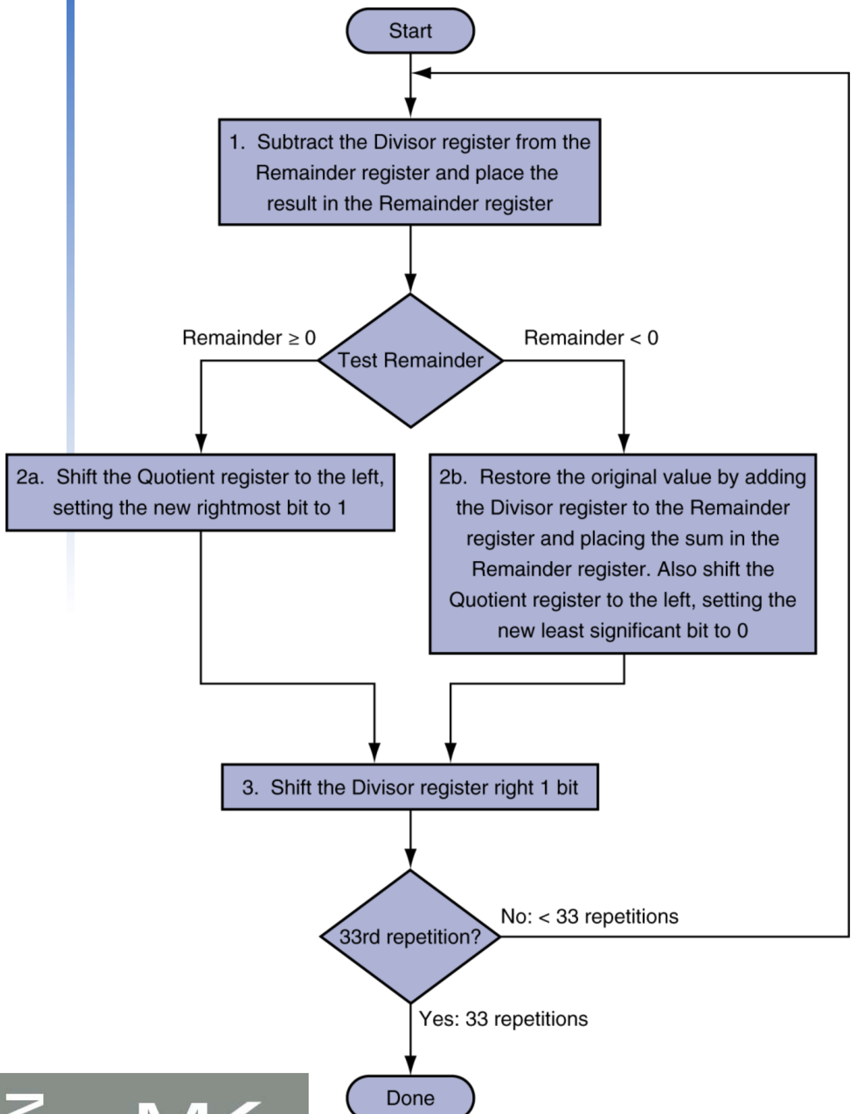
remainder 73

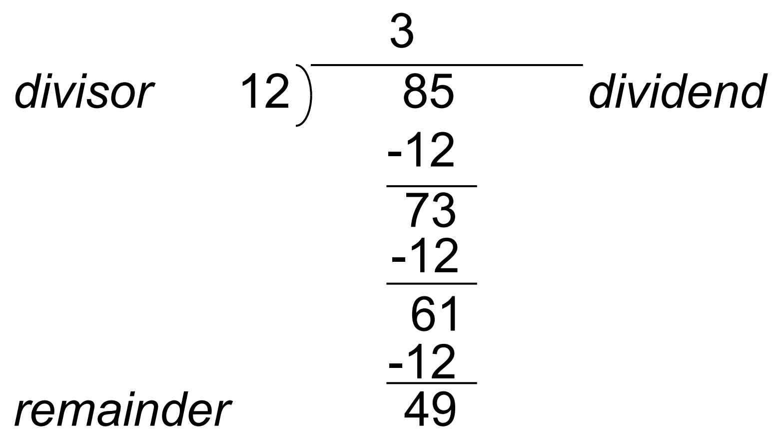
Division Hardware



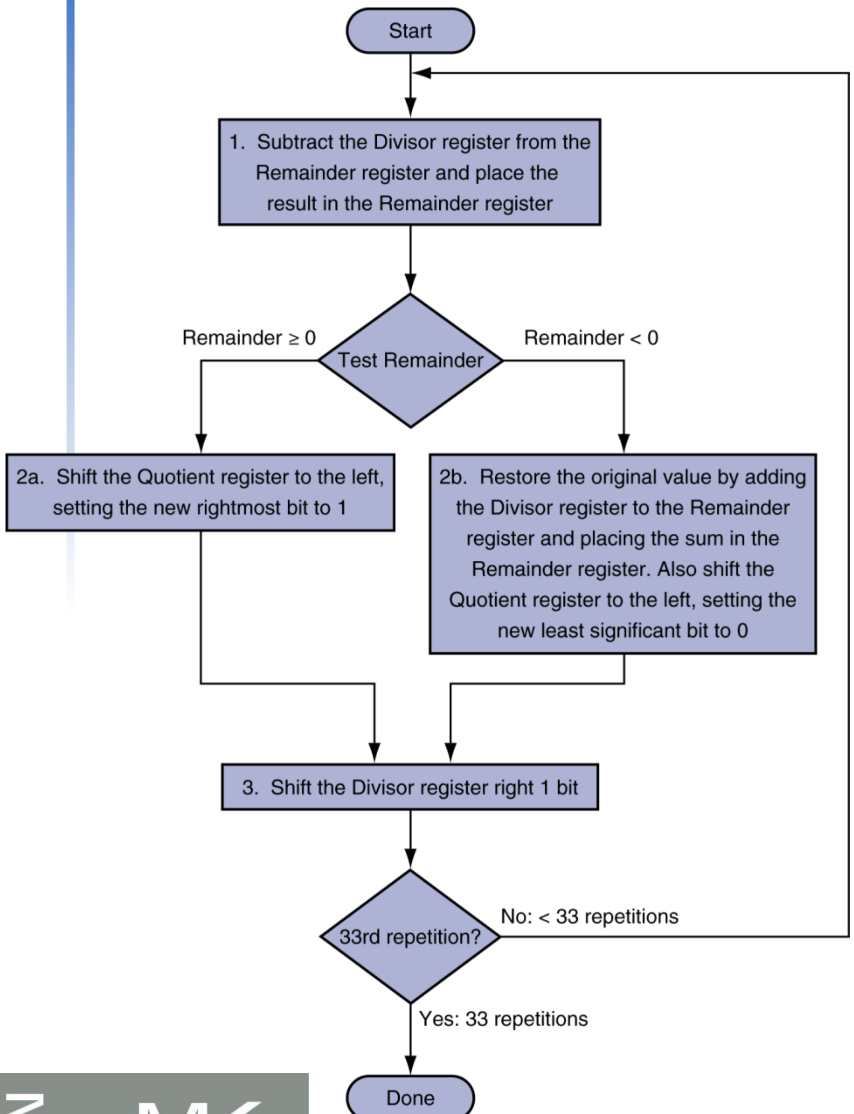

```
                    2
divisor    12 ) 85         dividend
               -12
                73
               -12
remainder       61
```

Division Hardware



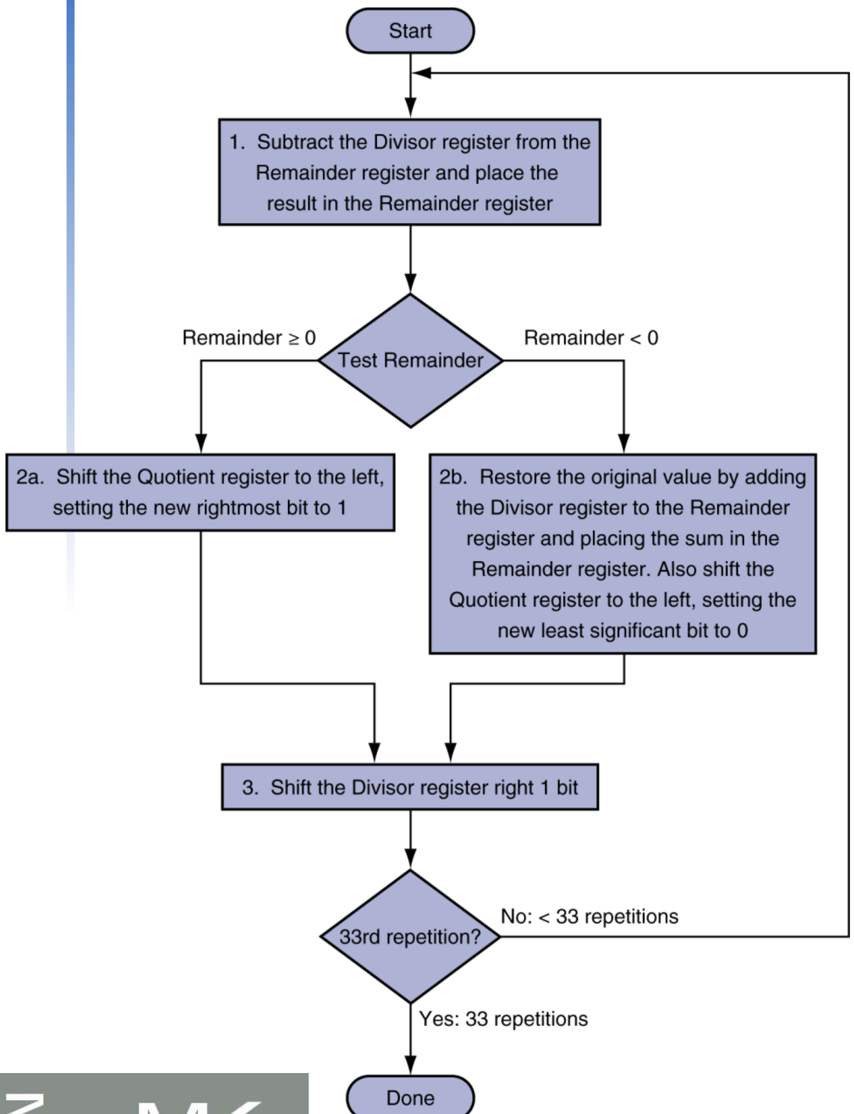


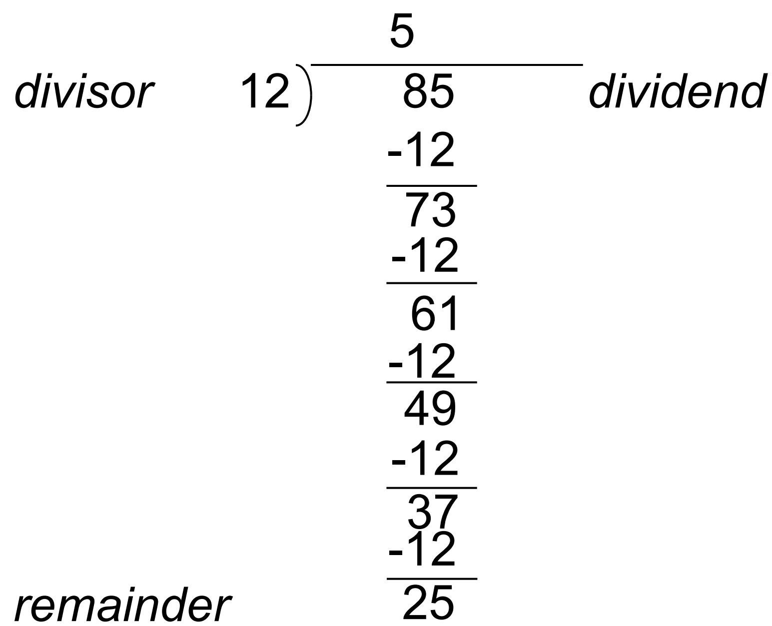
Division Hardware



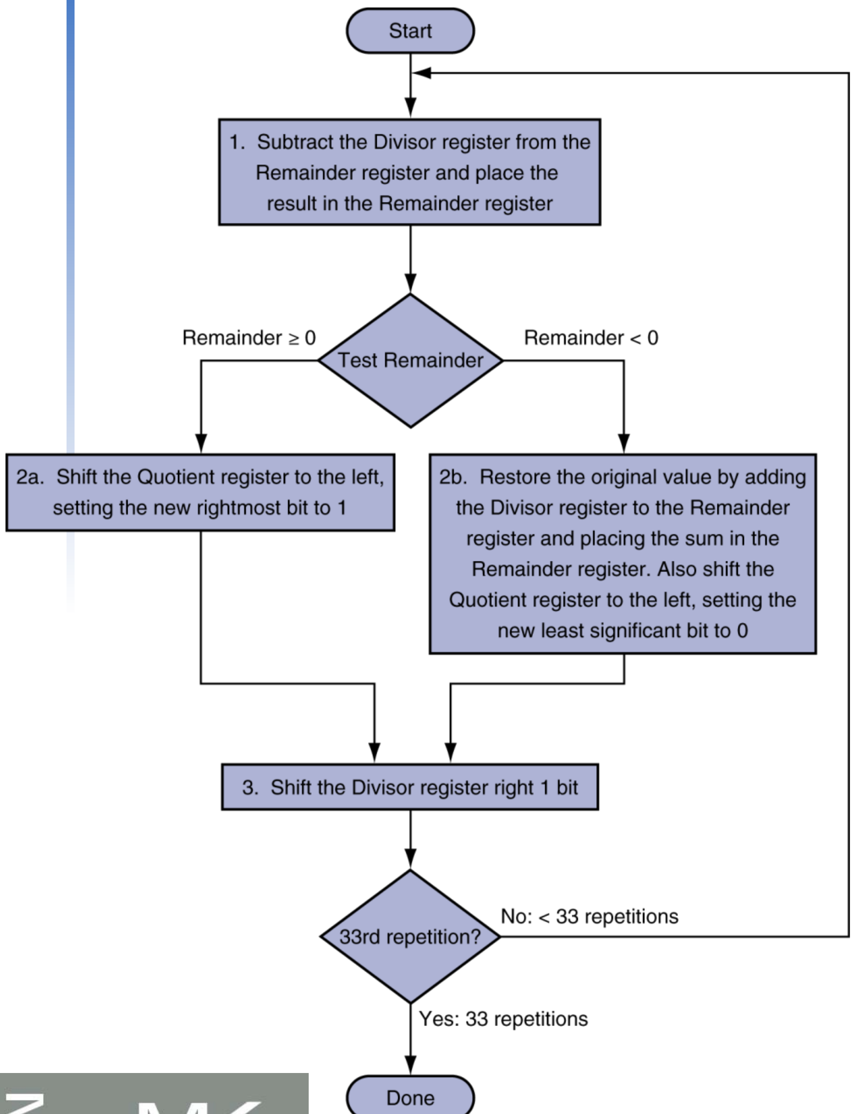

```
                      4
divisor     12 ) 85        dividend
                -12
                 73
                -12
                 61
                -12
                 49
                -12
remainder        37
```

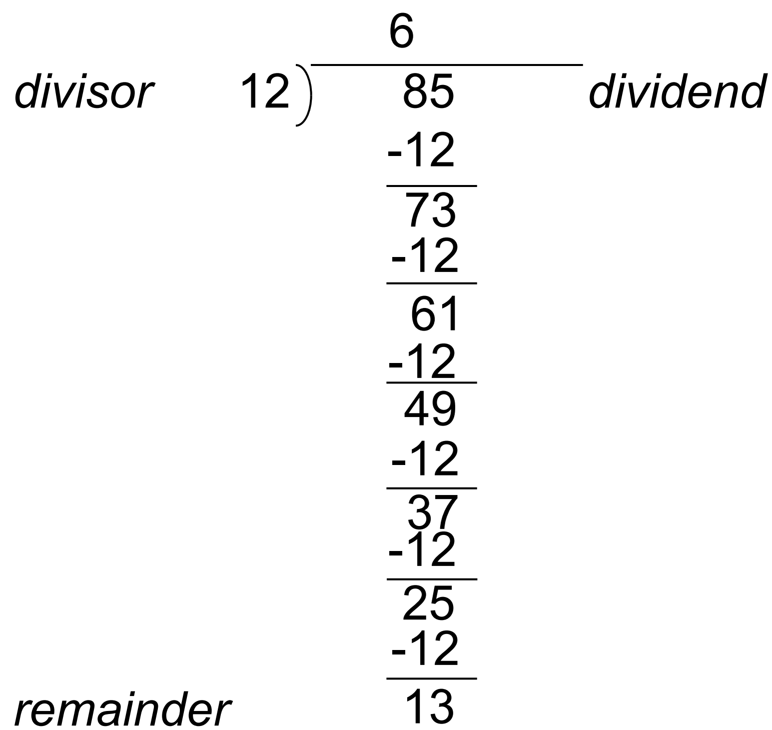
Division Hardware



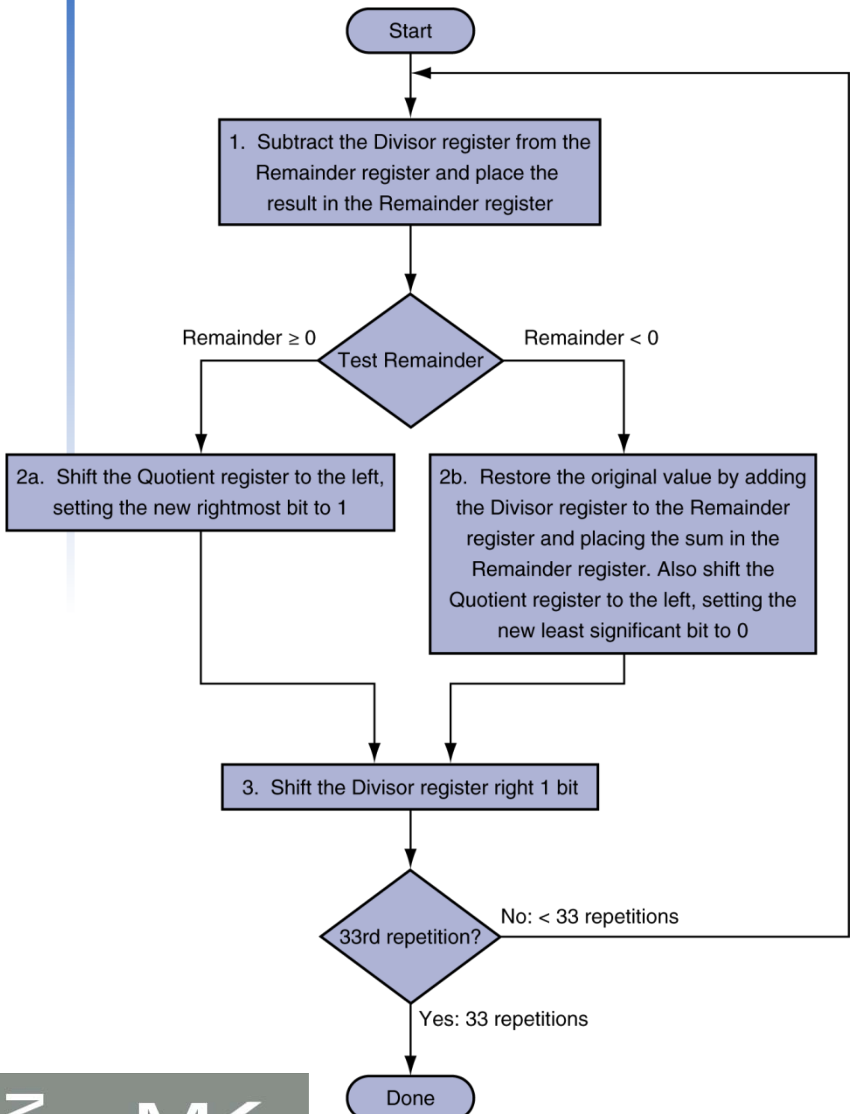


Division Hardware





Division Hardware



7

divisor 12) 85 *dividend*

-12

73

-12

61

-12

49

-12

37

-12

25

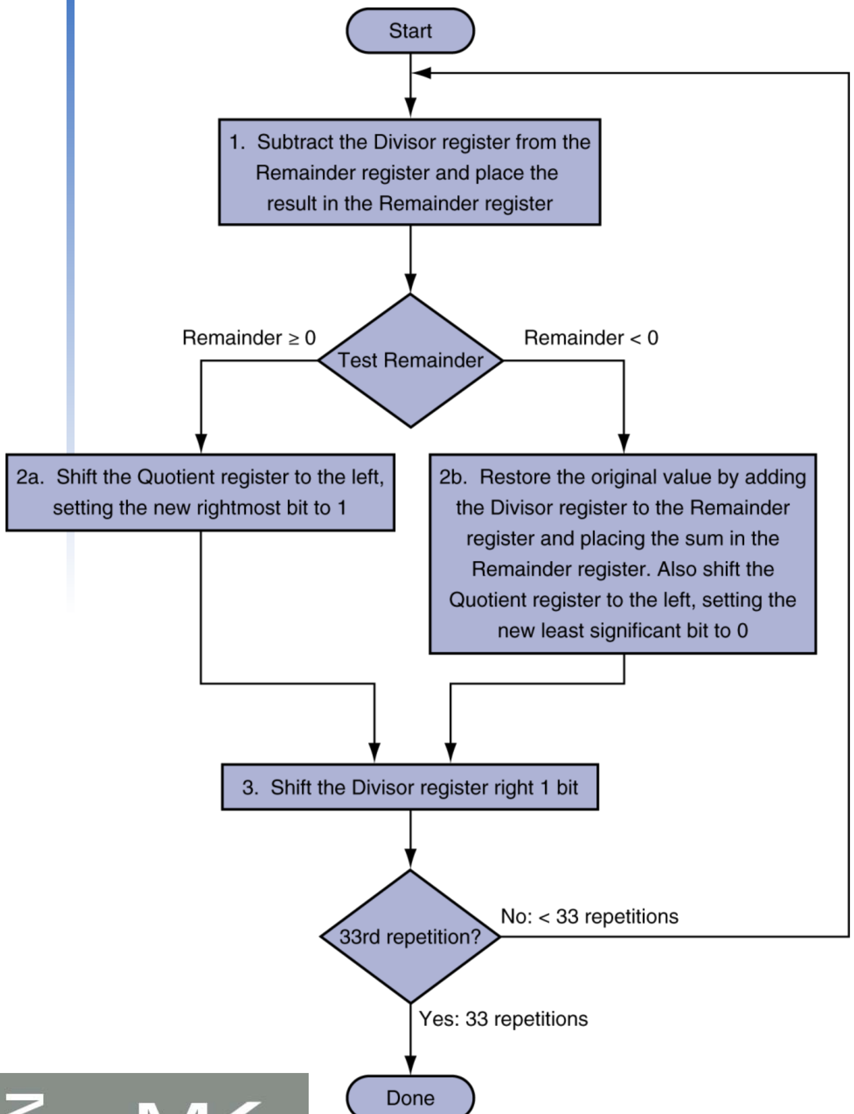
-12

13

-12

remainder 1

Division Hardware



7

divisor 12) 85 *dividend*

-12

73

-12

61

-12

49

-12

37

-12

25

-12

13

-12

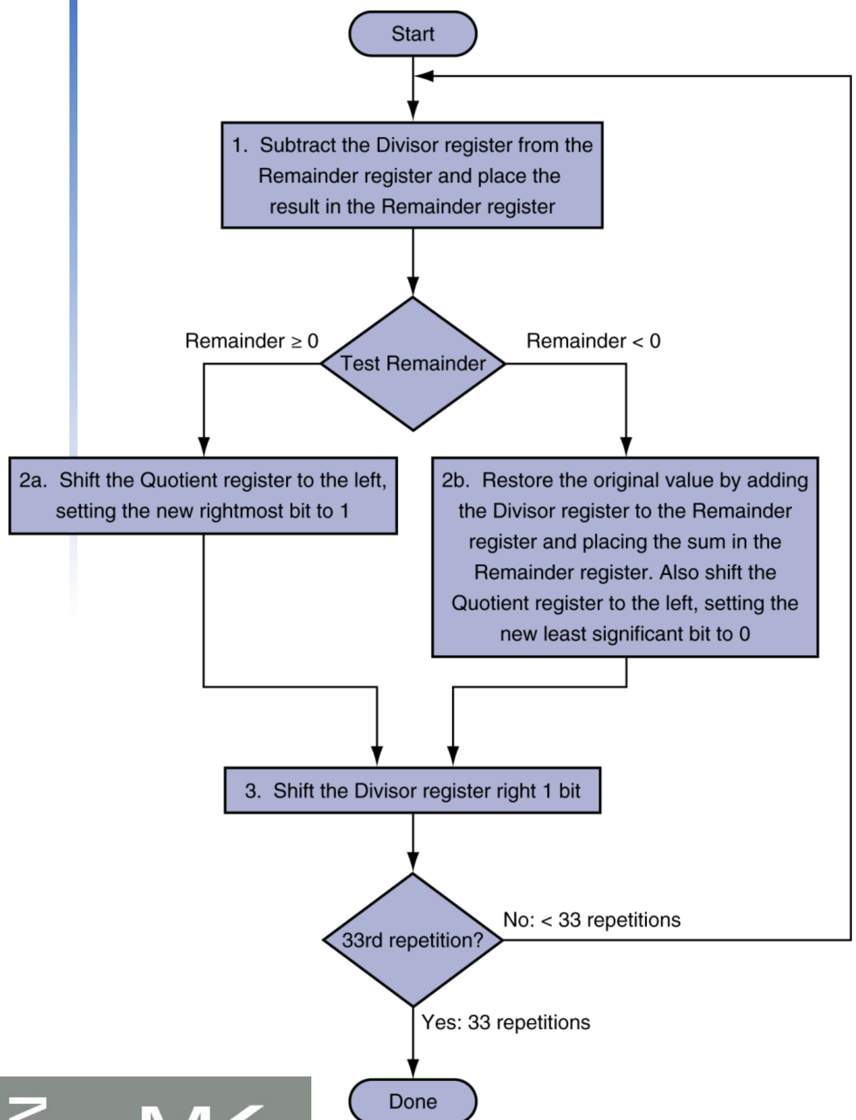
1

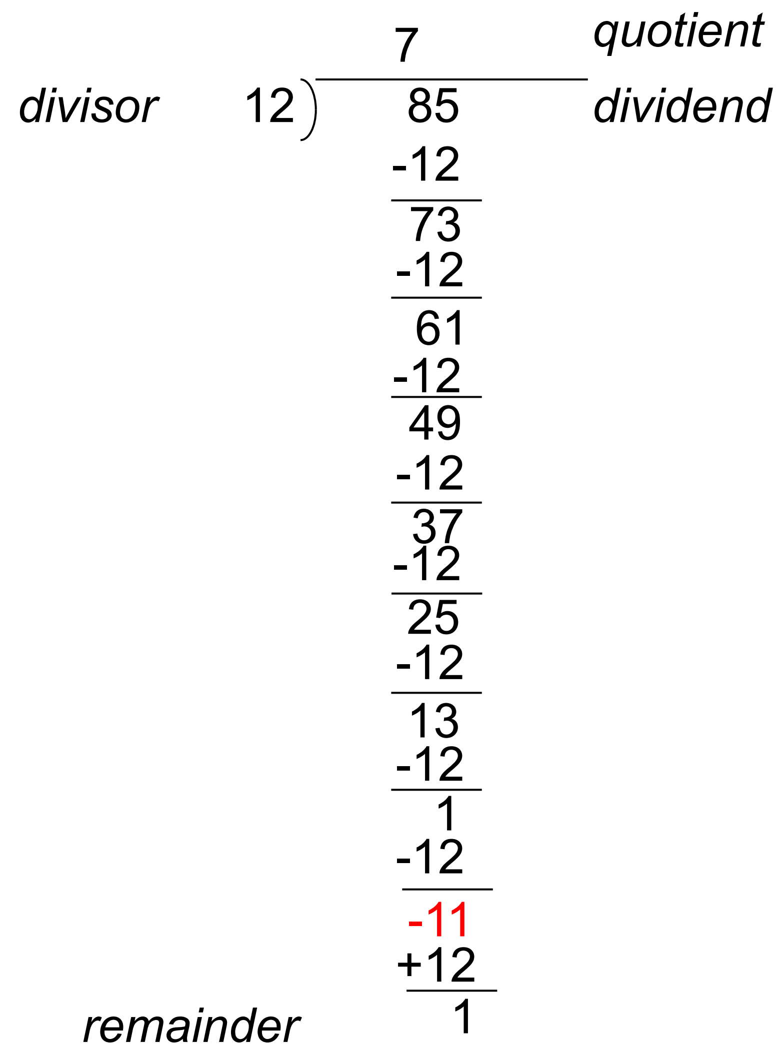
-12

-11

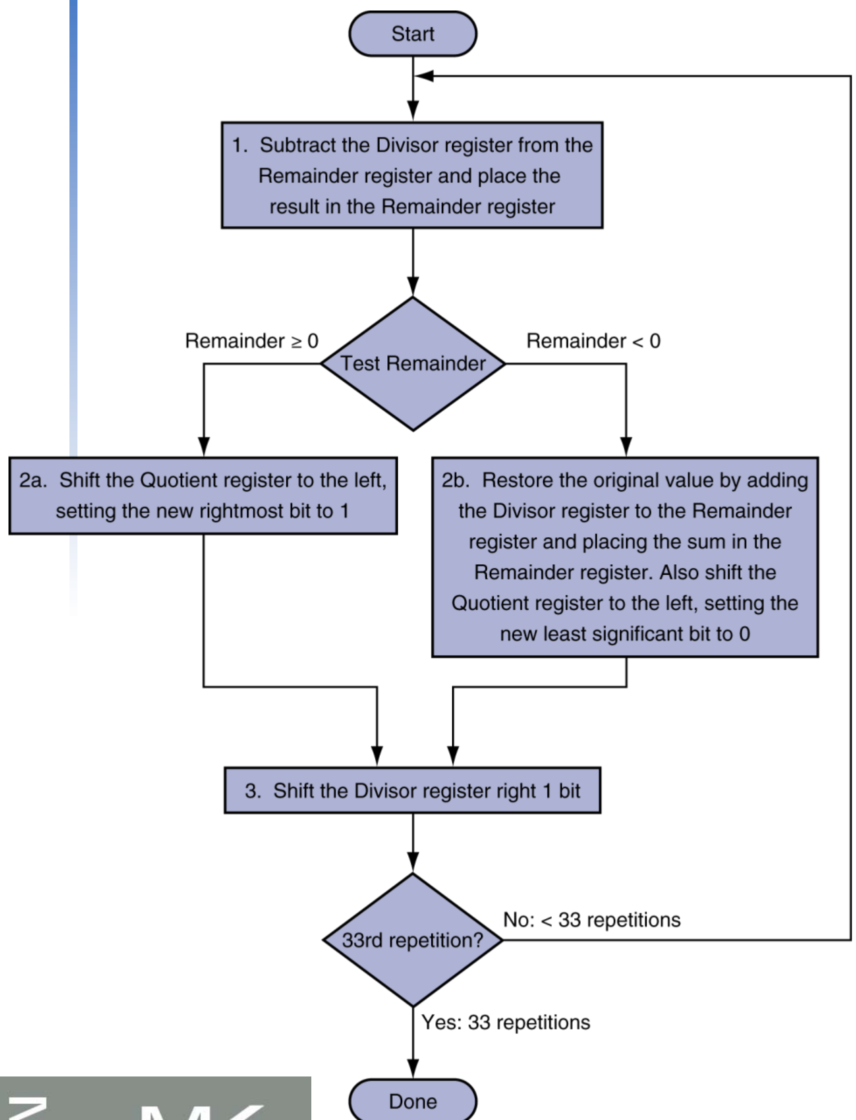
remainder < 0
add divisor to
remainder

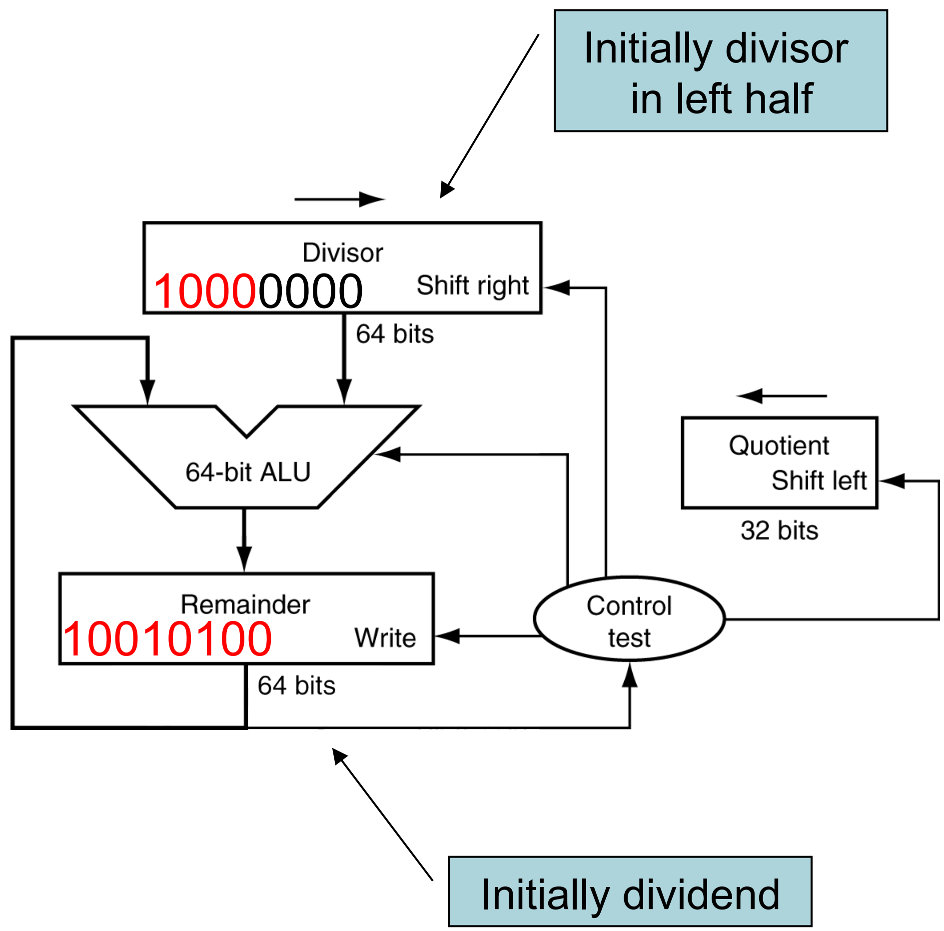
Division Hardware



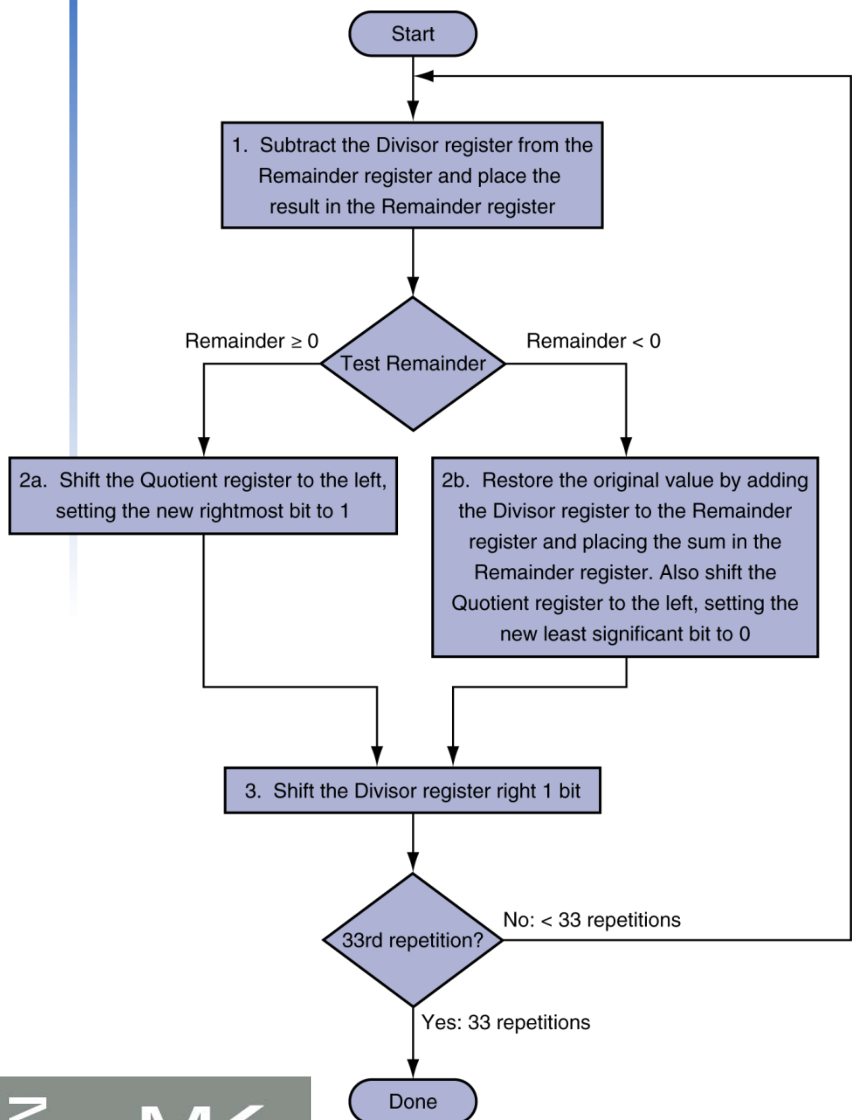


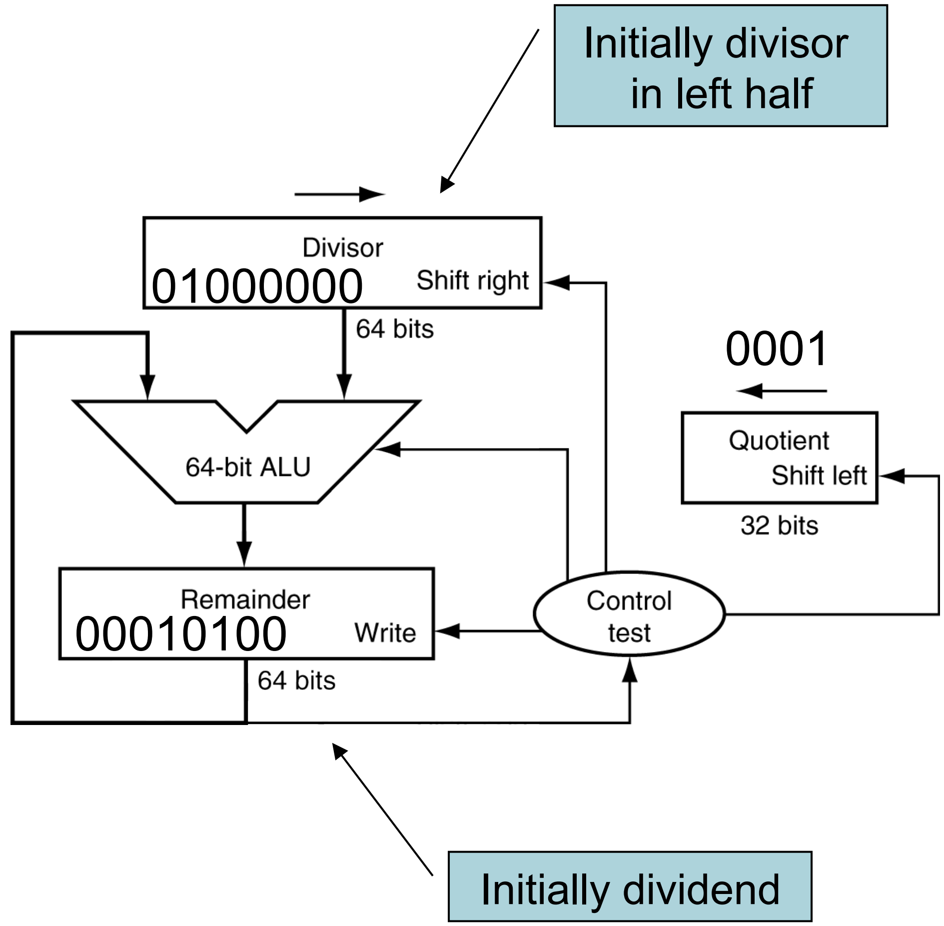
Division Hardware



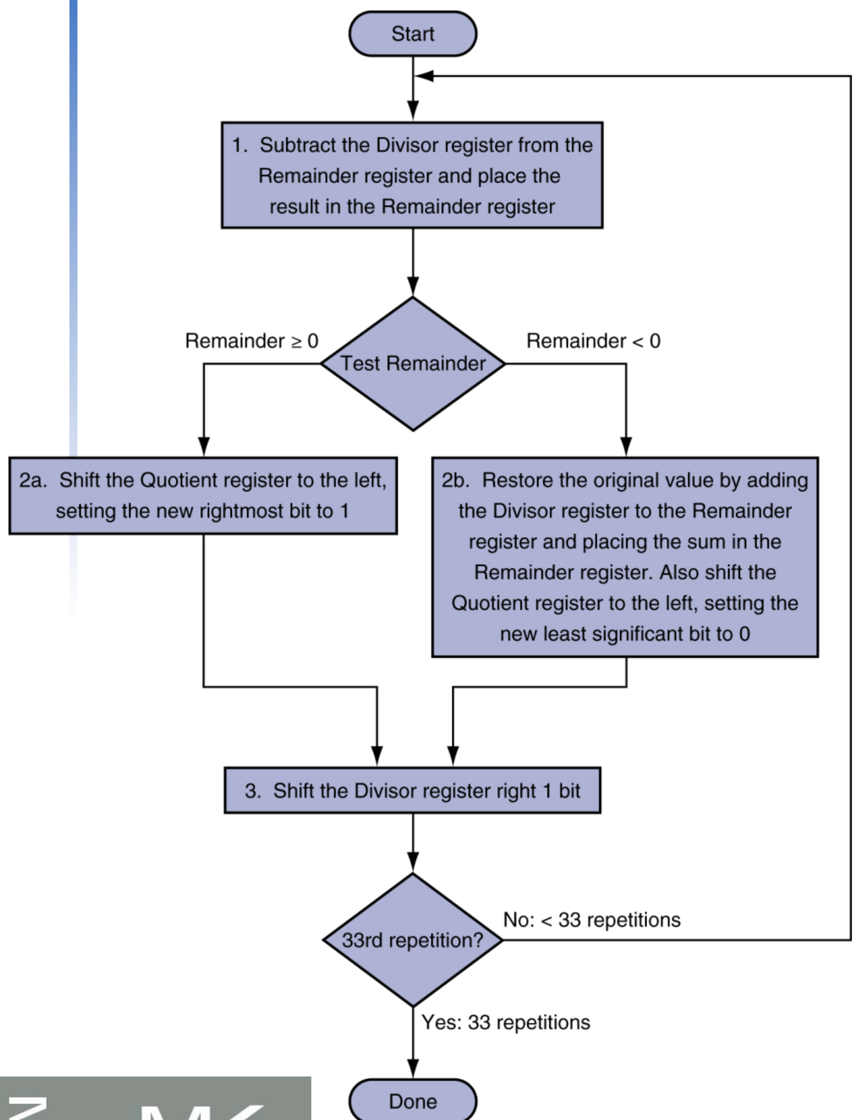


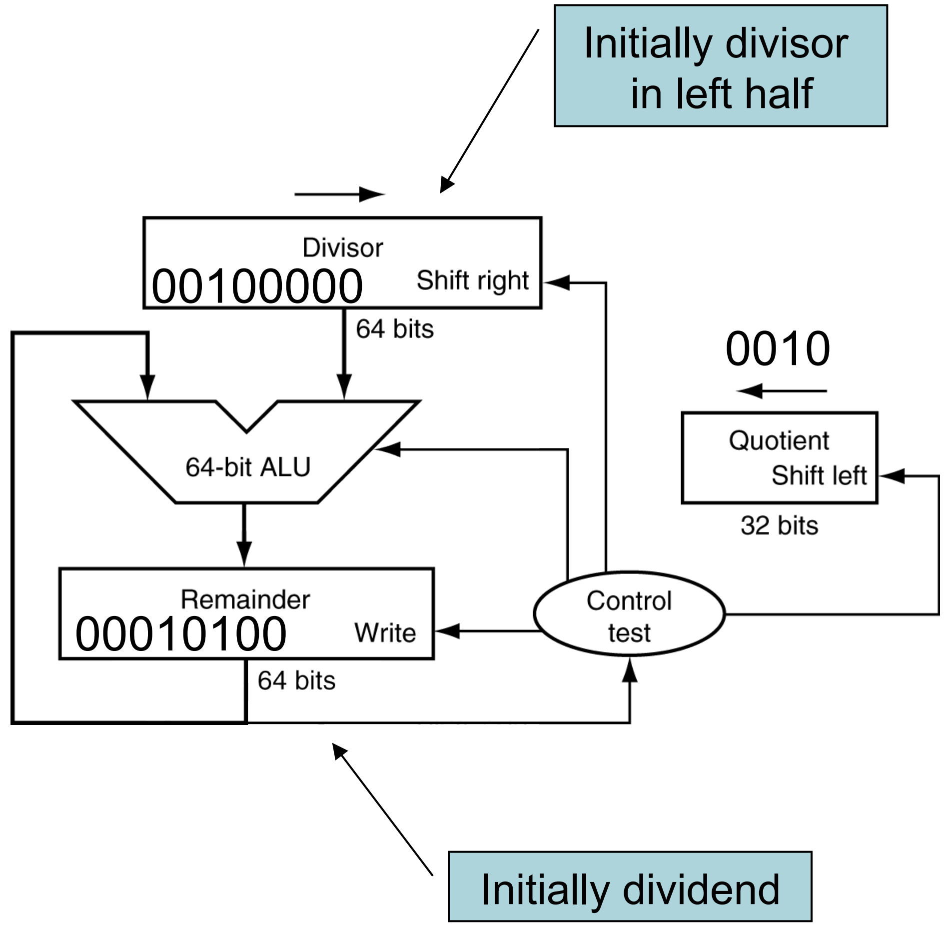
Division Hardware



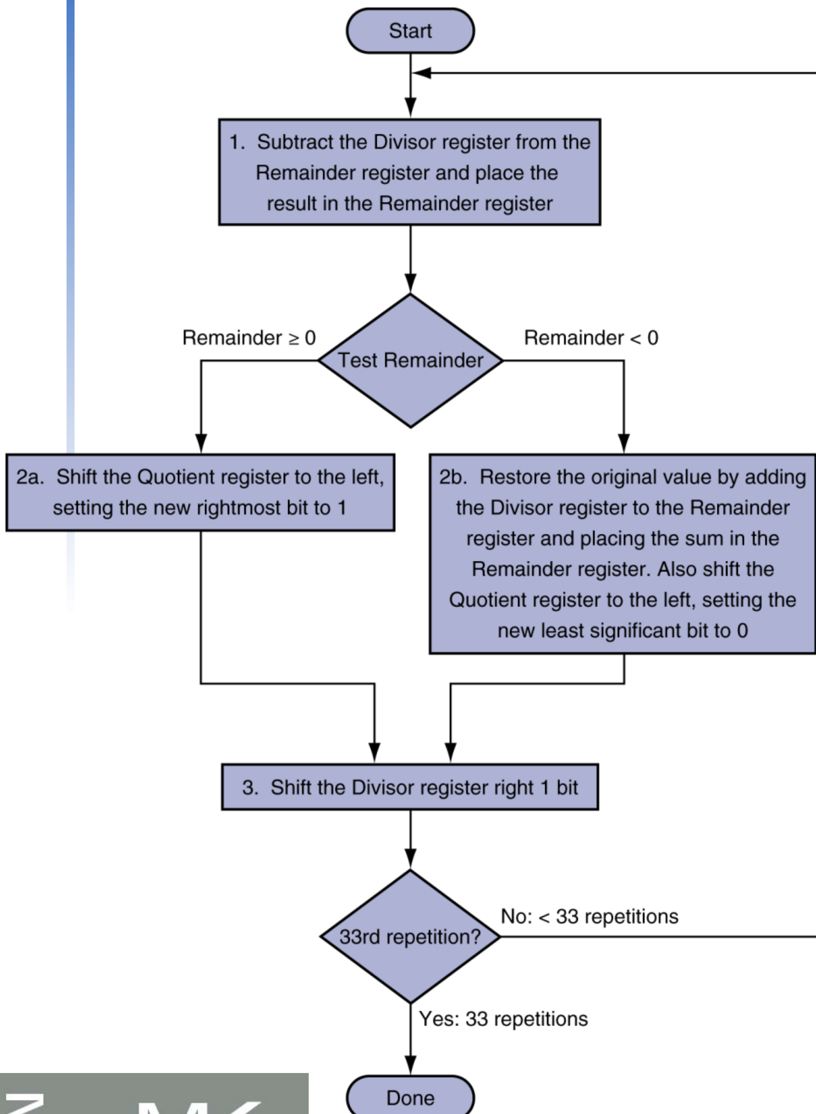


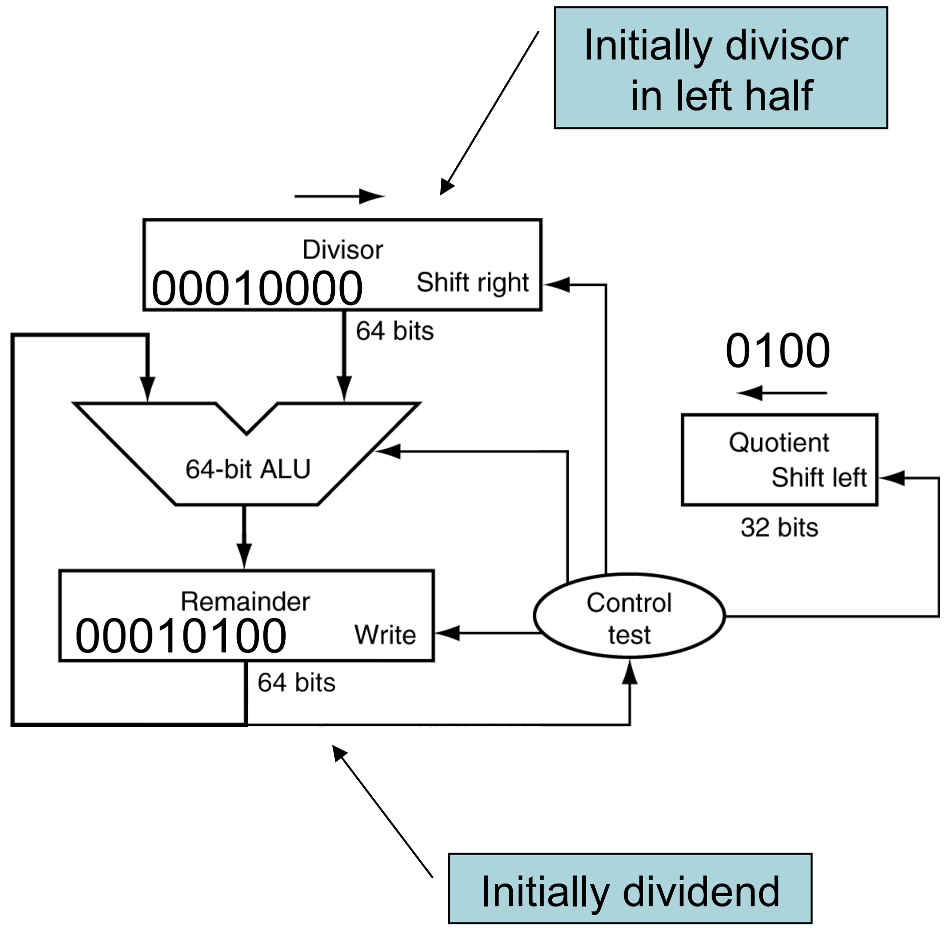
Division Hardware



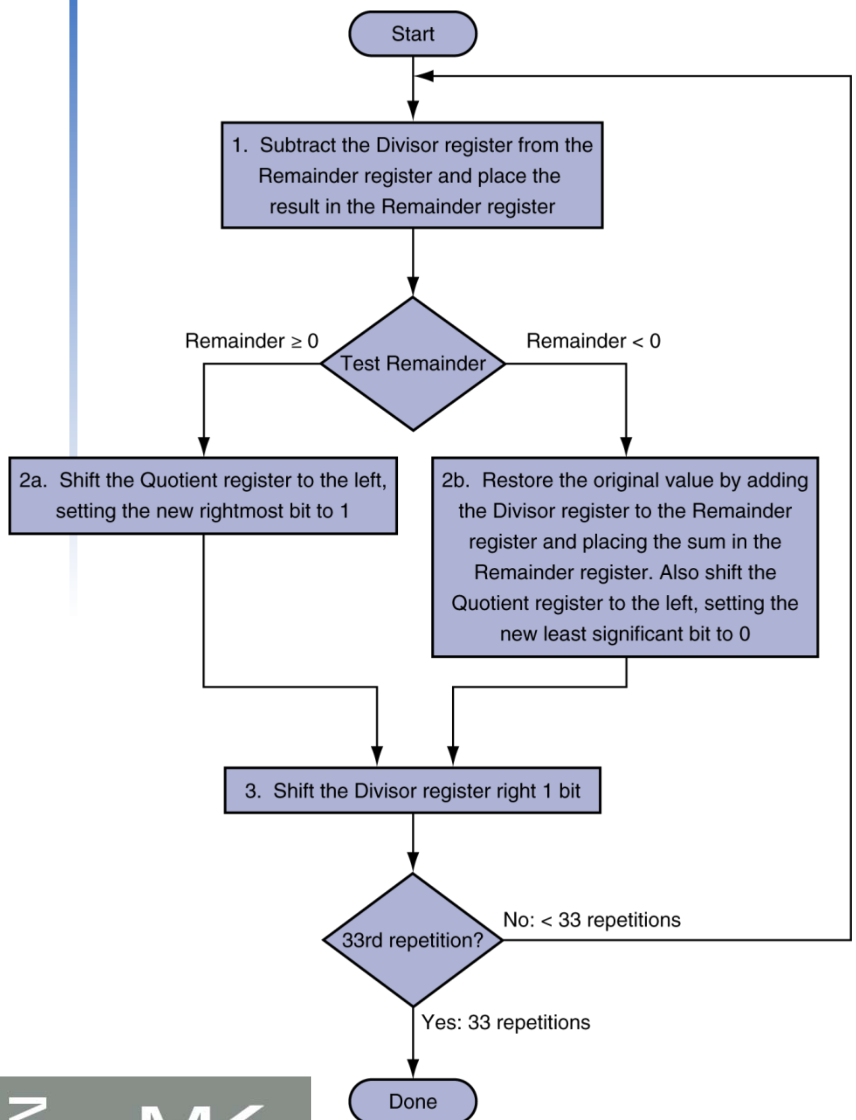


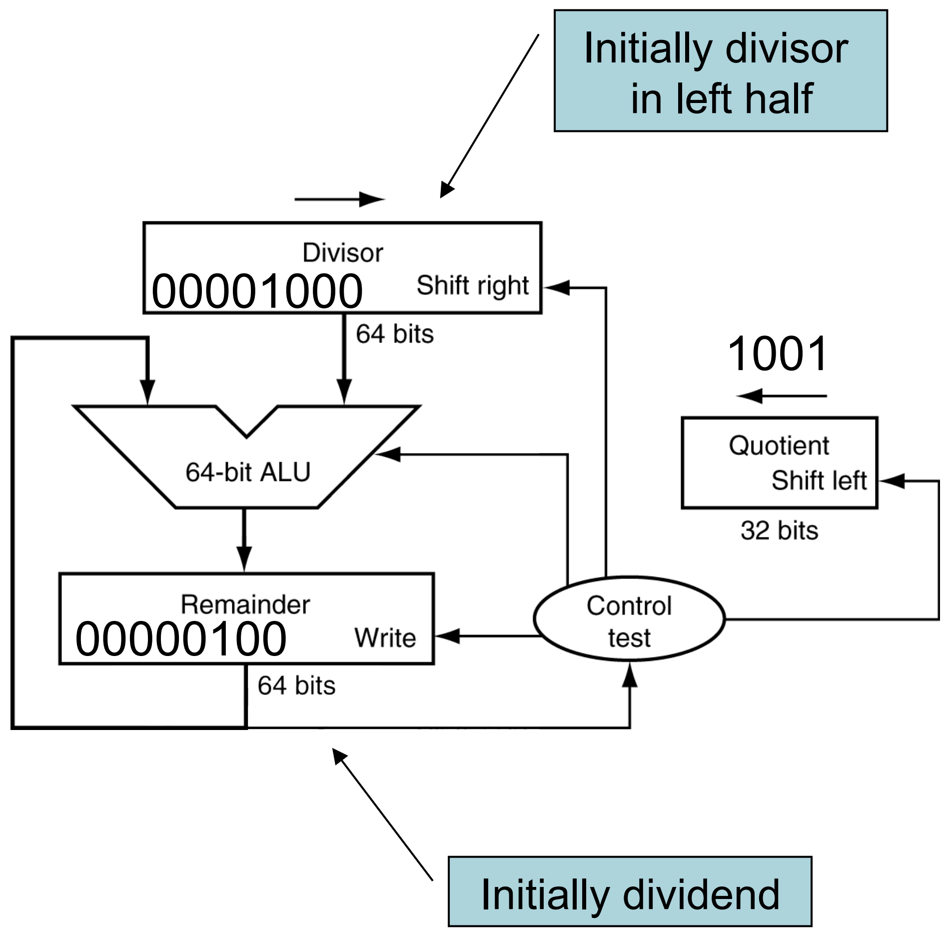
Division Hardware



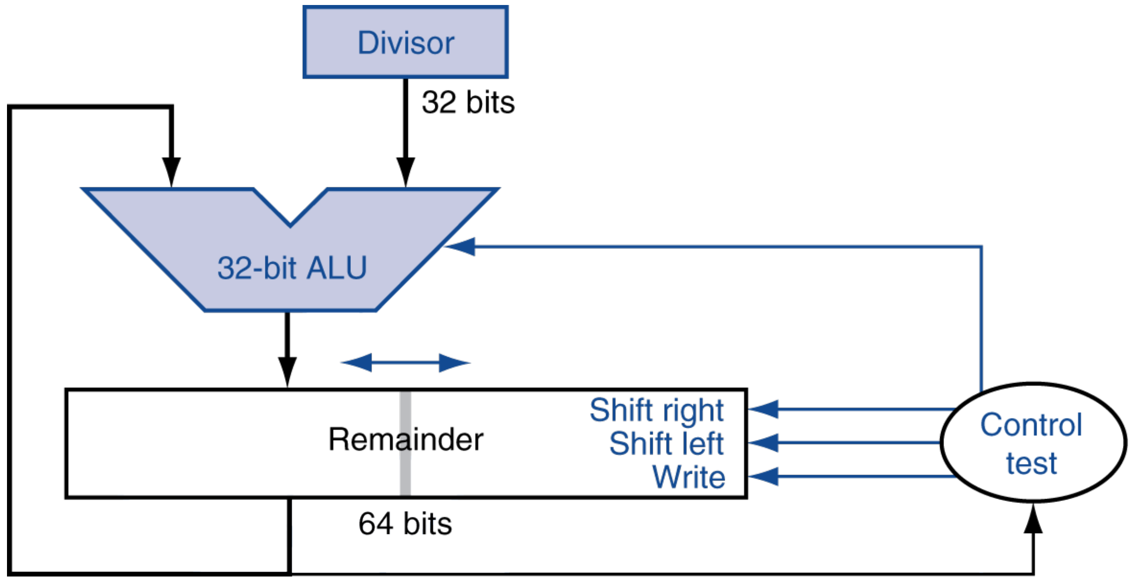


Division Hardware





Optimized Divider



- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both

Faster Division

- Can't use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder
- Faster dividers (e.g., SRT division) generate multiple quotient bits per step
  - Still require multiple steps
  - Uses a lookup table for guessing several quotient bits per step

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
    - E.g., mfhi $s3

      mflo $s2

Floating Point

- Representation for non-integer numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56} ← normalized
  - $+0.002 \times 10^{-4}$ ← not normalized
  - $+987.02 \times 10^{9}$ ← not normalized
- In binary
  - $\pm 1.s_1 s_2 \ldots_2 \times 2^{yyyy}$ $(+\text{-}\ 1 + s_1 \text{ x } 2^{-1} + s_2 \text{ x } 2 \wedge {-2} \ldots)$
- Types `float` and `double` in C

Floating-Point Numbers

- Suppose you are told to use the following representation for floating point numbers using 4 bits: bit 3 (sign), bit 2 (exponent of 2), and bits 1 and 0 (fraction of 2). Assume that numbers are normalized, i.e., the number is $(-1)^{\text{sign}} \times (1 + 2^{\text{exponent}})$.
- What are the possible numbers that can be represented?

Floating-Point Numbers

- Suppose you are told to use the following representation for floating point numbers using 4 bits: bit 3 (sign), bit 2 (exponent of 2), and bits 1 and 0 (fraction of 2). Assume that numbers are normalized, i.e., the number is $(-1)^{sign} \times (1 + 2^{exponent})$.
- What are the possible numbers that can be represented?
- Answer: exponent can be 0 or 1. Fraction can be 00, 11, 10, or 01 (which means 0, $(2^{-1} + 2^{-2} = 0.75)$, $2^{-1} = 0.5$, or $2^{-2} = 0.25$). So, the possible numbers are:
- ±1.0, 1.25, 1.5, 1.75, 2.0, 2.5, 3.0, 3.5
- How do get numbers < 1?

Floating-Point Numbers

- Suppose you are told to use the following representation for floating point numbers using 4 bits: bit 3 (sign), bit 2 (exponent of 2), and bits 1 and 0 (fraction of 2). Assume that numbers are normalized, i.e., the number is $(-1)^{sign} \times (1 + 2^{exponent})$.
- What are the possible numbers that can be represented?
- Answer: exponent can be 0 or 1. Fraction can be 00, 11, 10, or 01 (which means 0, ($2^{-1} + 2^{-2} = 0.75$), $2^{-1} = 0.5$, or $2^{-2} = 0.25$). So, the possible numbers are:
- ±1.0, 1.25, 1.5, 1.75, 2.0, 2.5, 3.0, 3.5
- How do we get numbers < 1?
  - Answer: Need a negative exponent (more about this later)

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 double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

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

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- 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 ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

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: 1011111101000…00
- Double: 1011111111101000…00

Floating-Point Example

- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$

$$0.75_{10} = 3/4_{10} = 3/2^2{}_{10} = 11_2/2^2{}_{10} = 0.11_2 = 1.1_2 \times 2^{-1}$$

  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2
- Single: 1011111101000…00
- Double: 1011111111101000…00

Floating-Point Example

- What number is represented by the single-precision float

  11000000101000…00
  - S = 1
  - Fraction = $01000\ldots00_2$
  - Fxponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$

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

  $= -5.0$

MK MORGAN KAUFMANN

Denormal Numbers

- Exponent = 000...0 ⇒ hidden bit is 0

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{-\text{Bias}}$$

- Smaller than normal numbers
  - allow for gradual underflow, with diminishing precision
- Denormal with fraction = 000...0

$$x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0$$

Two representations of 0.0!

Infinities and NaNs

- Exponent = 111...1, Fraction = 000...0
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Fraction ≠ 000...0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0
  - Can be used in subsequent calculations

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

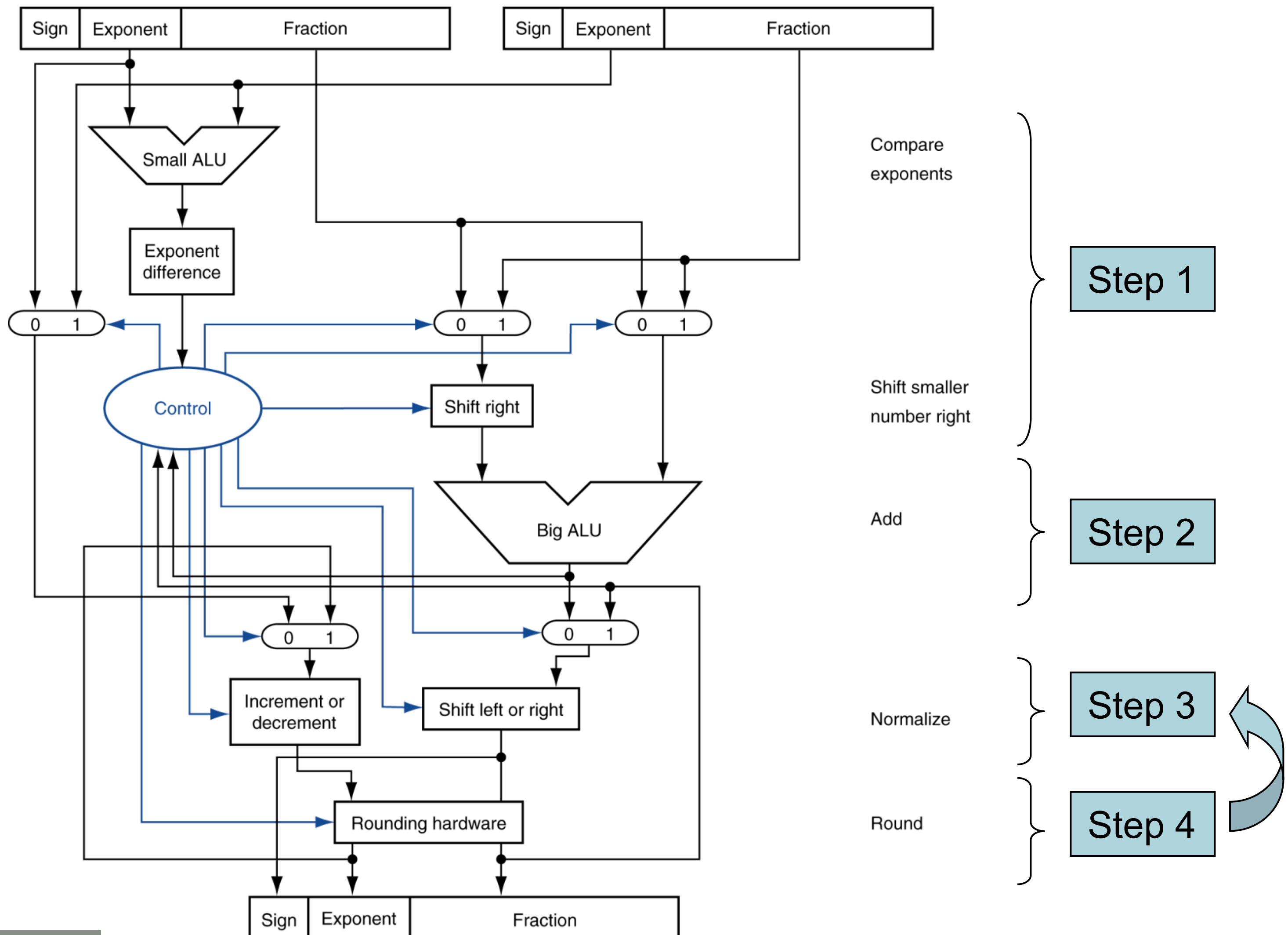
Floating-Point Addition

- 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

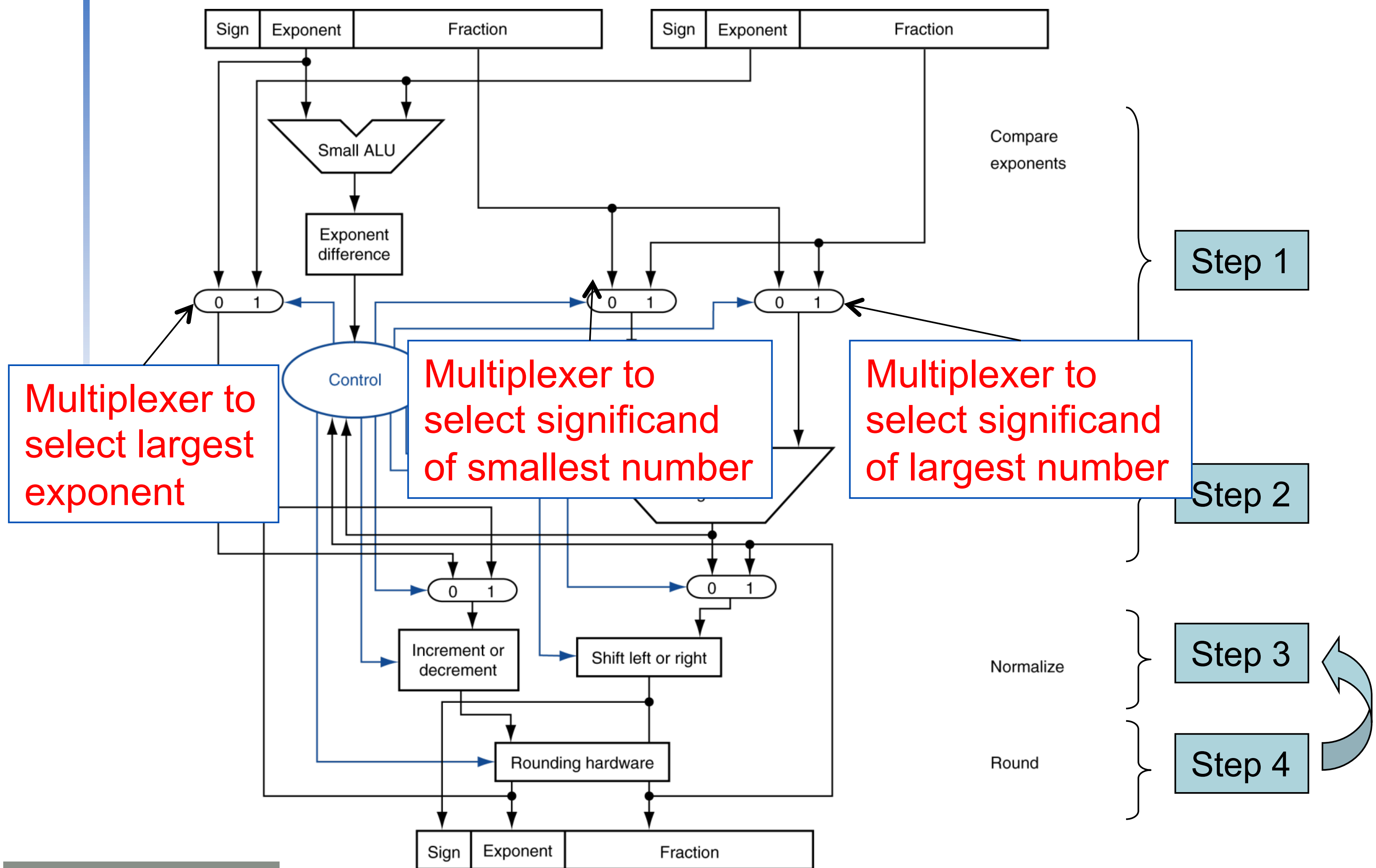
FP Adder Hardware

- 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

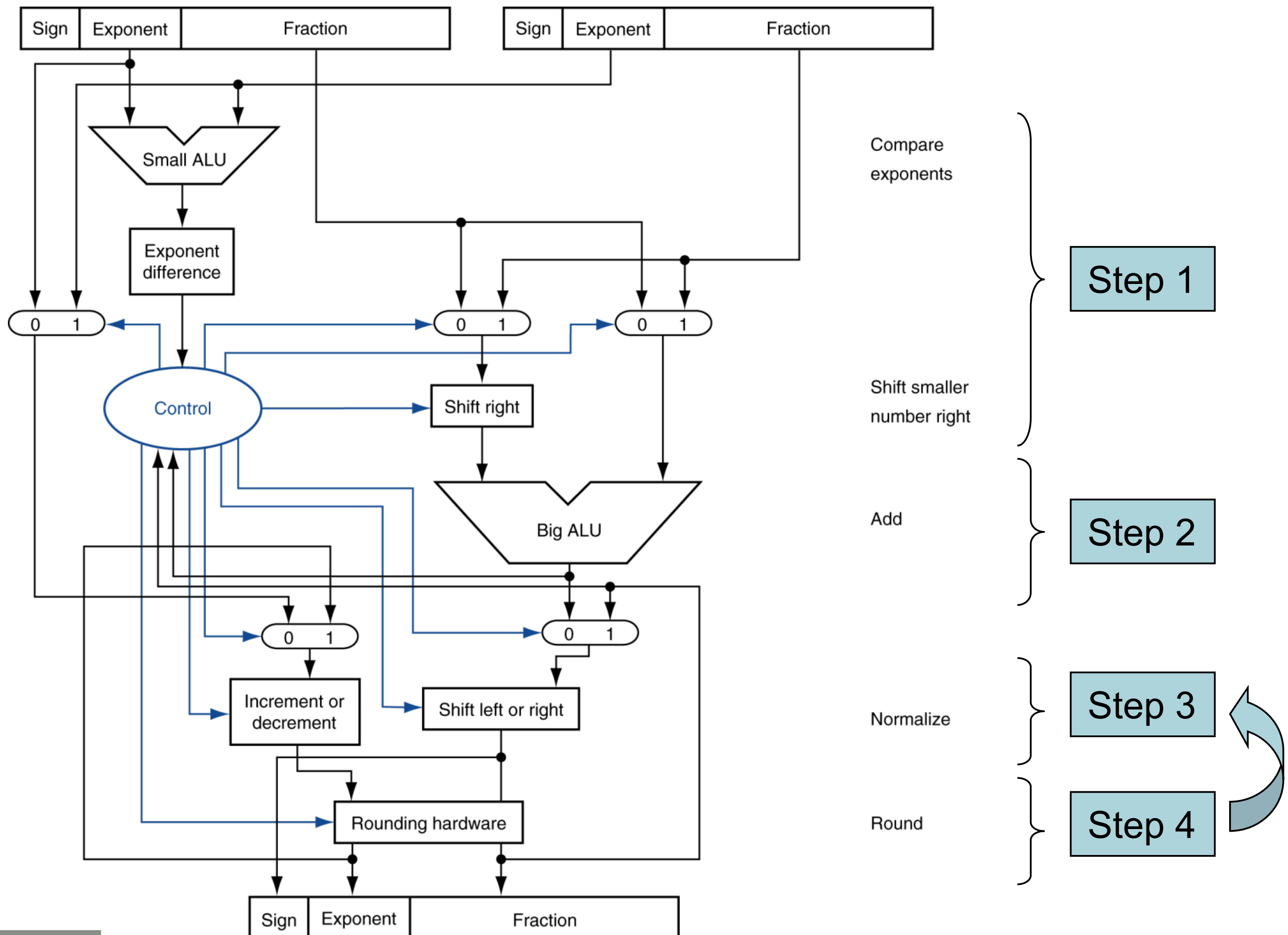
FP Adder Hardware



FP Adder Hardware



FP Adder Hardware



Floating-Point Multiplication

- Consider a 4-digit decimal example
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
  - For biased exponents, subtract bias from sum
  - New exponent = 10 + –5 = 5
- 2. Multiply significands
  - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$
- 3. Normalize result & check for over/underflow
  - 1.0212×10^{6}
- 4. Round and renormalize if necessary
  - 1.021×10^{6}
- 5. Determine sign of result from signs of operands
  - $+1.021 \times 10^{6}$

Floating-Point Multiplication

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5 × –0.4375)
- 1. Add exponents
  - Unbiased: –1 + –2 = –3
  - Biased: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: +ve × –ve ⇒ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$

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 ↔ 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 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`
- Single- and double-precision comparison
  - `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`
- 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)
     lwc1  $f18, const9($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

Storing multi-dimensional arrays

Consider a 3 x 2 matrix stored in memory in *row major order*, i.e., elements are stored row by row. Each element is 4-bytes long. What is the byte offset of element i,j?

Storing multi-dimensional arrays

Consider a 3 x 2 matrix stored in memory in *row major order*, i.e., elements are stored row by row. Each element is 4-bytes long. What is the byte offset of element i,j?

$$\begin{bmatrix} A00 & A01 \\ A10 & A11 \\ A20 & A21 \end{bmatrix}$$

0	A00
4	A01
8	A10
12	A11
16	A20
20	A21

[i,j] = (i * row dim + j) * size element

[1,1] = (1 * 2 + 1) * 4 = 12

[2,0] = (2*2 + 0) * 4 = 16

Absolute address [i,j] = array base address + (i * row dim + j) * size element

Storing multi-dimensional arrays

Write MIPS code to load into $t4, element A [i,j] assuming that The base address of A is in $s0, i is in $s1, j in $s2, each element of A is 4 bytes and A is a 10 x 20 matrix.

Absolute address [i,j] = array base address + (i * row dim + j) * size element

Storing multi-dimensional arrays

Write MIPS code to load into $t4, element A [i,j] assuming that The base address of A is in $s0, i is in $s1, j in $s2, each element of A is 4 bytes and A is a 10 x 20 matrix.

Absolute address [i,j] = array base address + (i * row dim + j) * size element

```
addi    $t1, $0, 20        # $t1 = 20
mul     $t1, $s1, $t1      # $t1 = i * 20
add     $t1, $t1, $s2      # $t1 = i * 20 + j
sll     $t1, $t1, 2        # $t1 = (i * 20 + j) * 4
add     $t1, $t1, $s0      # $t1 = Addr[A] + (i * 20 + j) * 4
lw      $t4, 0 ($t1)       # $t4 = A[i,j]
```

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
  - Not all real numbers in the FP range can be represented.
  - 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

Rounding with Guard Digits

- Add $2.56_{10} \times 10^0$ to $2.34_{10} \times 10^2$ assuming 3 significant decimal digits. Round to the nearest decimal number, first with guard and round digits, and then without them.

- With guard and round digits:

$$
\begin{array}{r}
2.3400 \times 10^2 \\
+\ 0.0256 \times 10^2 \\
\hline
2.3656 \times 10^2 \rightarrow 2.37 \times 10^2
\end{array}
$$

- Without guard and round digits:

$$
\begin{array}{r}
2.34 \times 10^2 \\
+\ 0.02 \times 10^2 \\
\hline
2.36 \times 10^2
\end{array}
$$

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:

```
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+j*n]; /* cij = C[i][j] */
    for(int k = 0; k < n; k++ )
     cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
    C[i+j*n] = cij; /* C[i][j] = cij */
   }
}
```

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:

```
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
```

Fallacy: 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
    - Rounds toward –∞
  - c.f. 11111011_2 >> 2 = 00111110_2 = +62

Pitfall: FP Addition is not Associative

- 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

Fallacy: 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 in 1994
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*
  - Intel recalled the flawed microprocessor at a cost of $500 million!

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
  - Two's complement and IEEE 754 are standard.
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow

Concluding Remarks

- MIPS ISA
  - Core instructions: 54 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent
- Rest of book concentrates on:
  - add, addi, addu, addiu, sub, subu, AND, ANDI, OR, Ori, NOR, sll, srl
  - lui, lw, sw,lhu,sh, lbu,sb,
  - ll, sc
  - beq, bne, j, jal, jr,
  - slt, slti, sltu, sltiu