



# Chapter 3

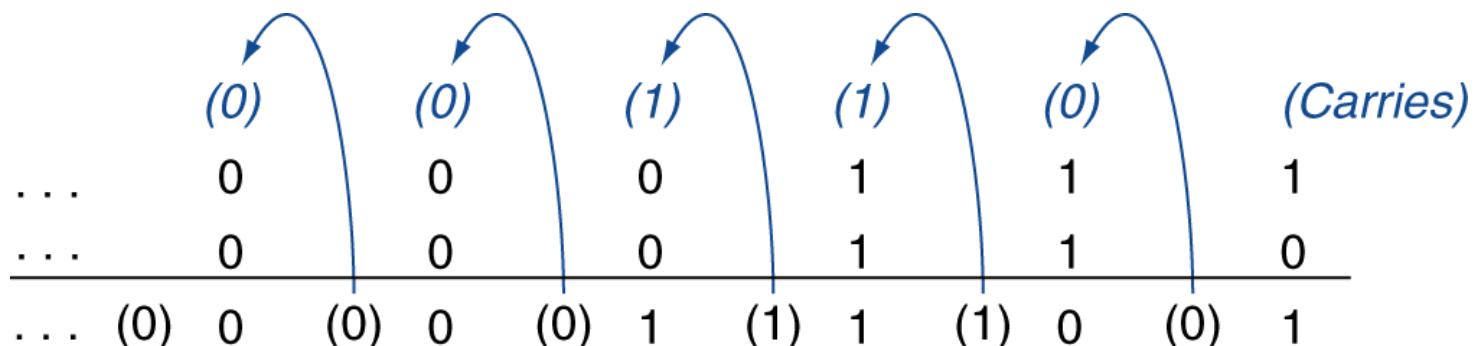
## Arithmetic for Computers

# 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:  $7 - 6 = 7 + (-6)$

$$\begin{array}{r} +7: \quad 0000\ 0000\ \dots\ 0000\ 0111 \\ -6: \quad 1111\ 1111\ \dots\ 1111\ 1010 \\ \hline +1: \quad 0000\ 0000\ \dots\ 0000\ 0001 \end{array}$$

- 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

# Dealing with Overflow

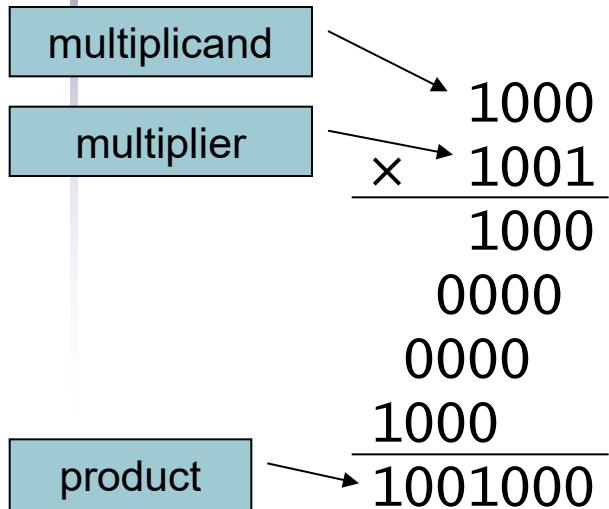
- Some languages (e.g., C) ignore overflow
  - Use MIPS addu, addui, subu instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
  - Use MIPS add, addi, sub instructions
  - On overflow, invoke exception handler
    - Save PC in exception program counter (EPC) register
    -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

# Arithmetic for Multimedia

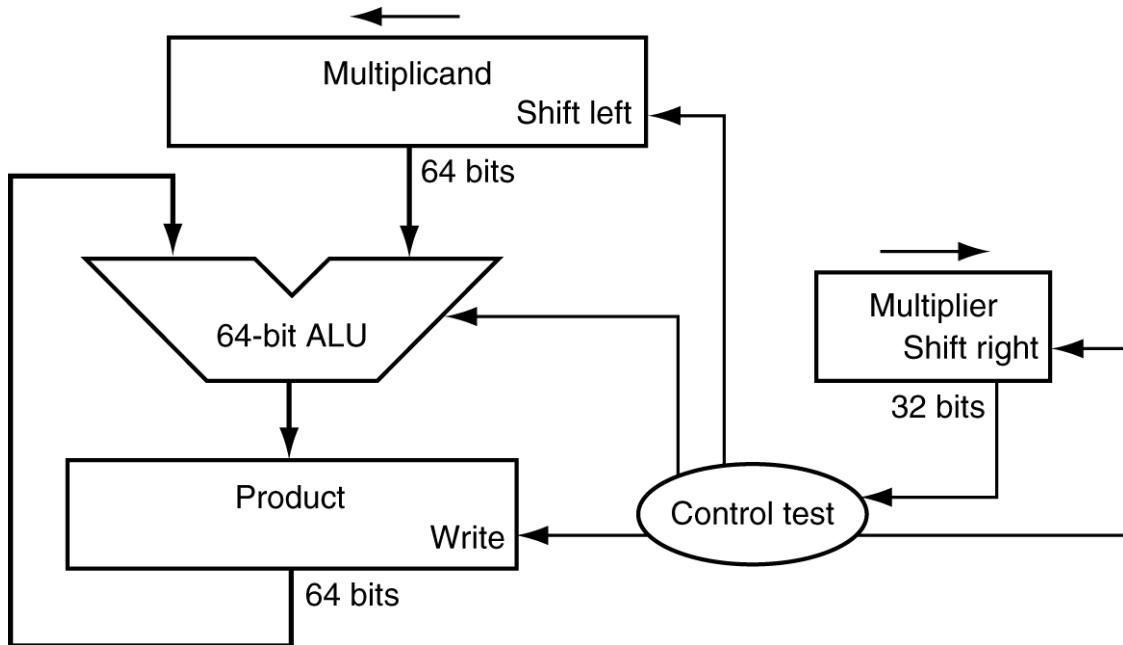
- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
    - Operate on  $8 \times 8$ -bit,  $4 \times 16$ -bit, or  $2 \times 32$ -bit vectors
  - SIMD (single-instruction, multiple-data)
- Saturating operations
  - On overflow,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

# Multiplication

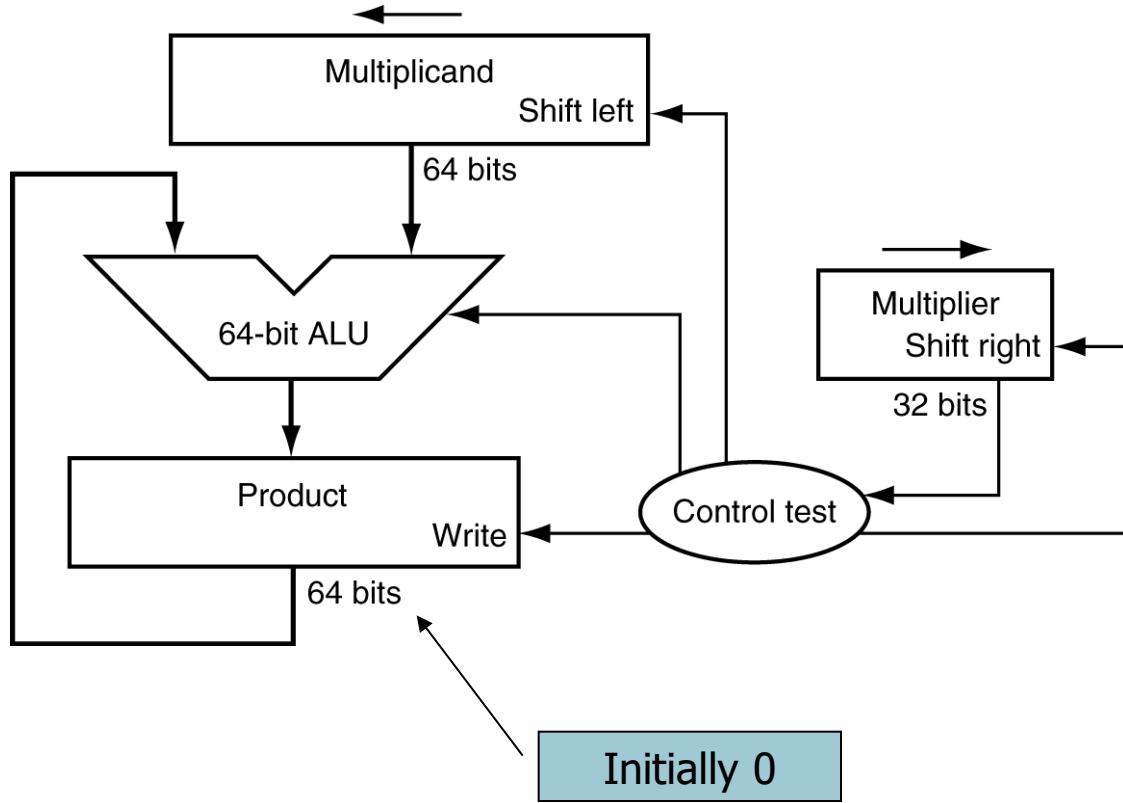
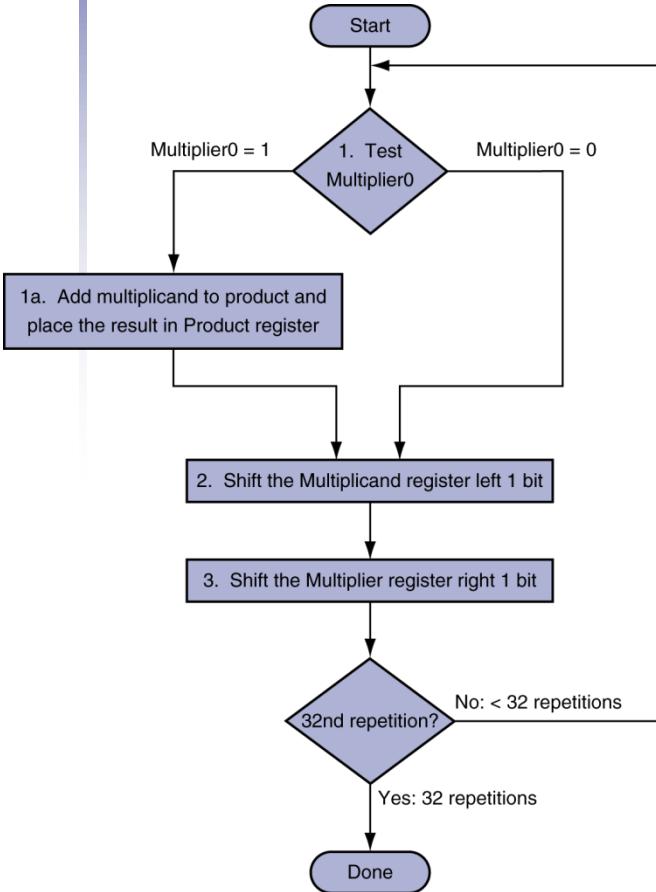
- Start with long-multiplication approach



Length of product is  
the sum of operand  
lengths



# Multiplication Hardware



# A Multiply Algorithm

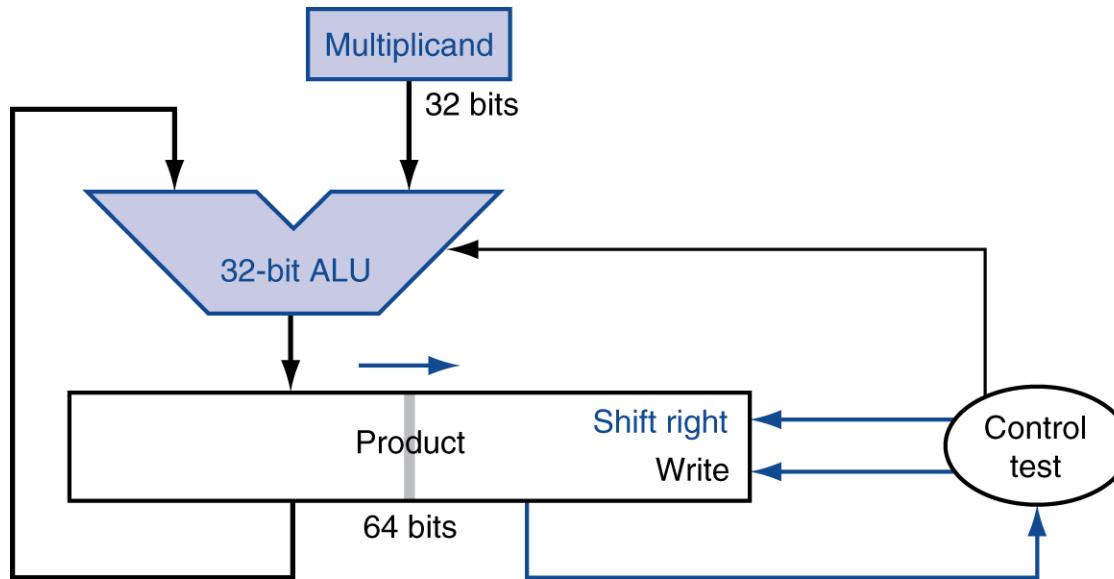
## A Multiply Algorithm

Using 4-bit numbers to save space, multiply  $2_{\text{ten}} \times 3_{\text{ten}}$ , or  $0010_{\text{two}} \times 0011_{\text{two}}$ .

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: $1 \Rightarrow \text{Prod} = \text{Prod} + \text{Mcand}$	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: $1 \Rightarrow \text{Prod} = \text{Prod} + \text{Mcand}$	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: $0 \Rightarrow \text{No operation}$	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: $0 \Rightarrow \text{No operation}$	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

# Optimized Multiplier

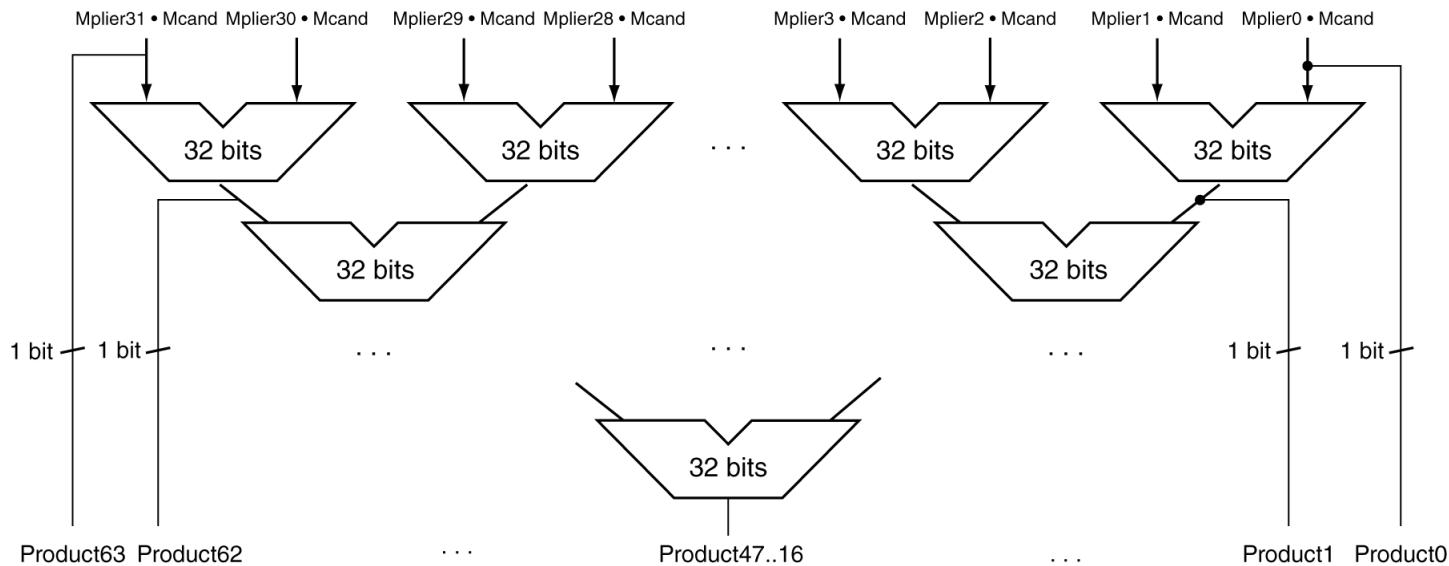
- Perform steps in parallel: addshift



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

# Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff

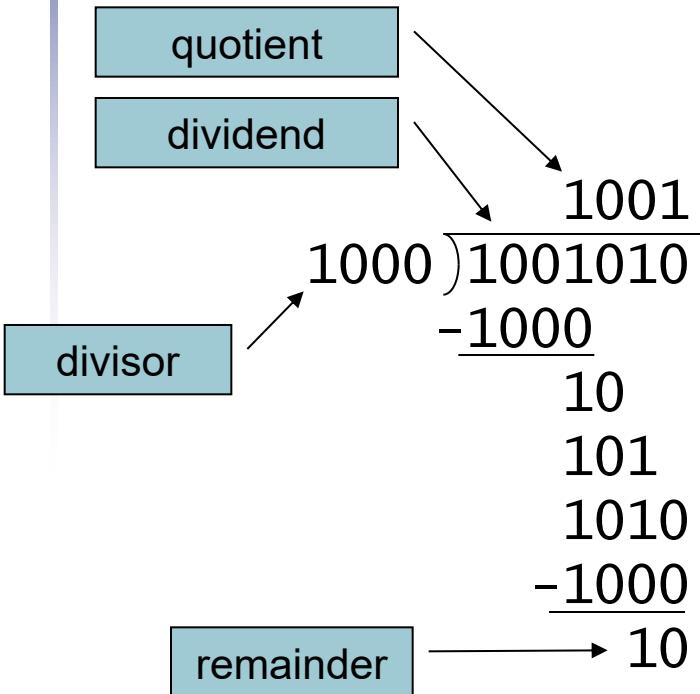


- Can be pipelined
  - Several multiplication performed in parallel

# 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

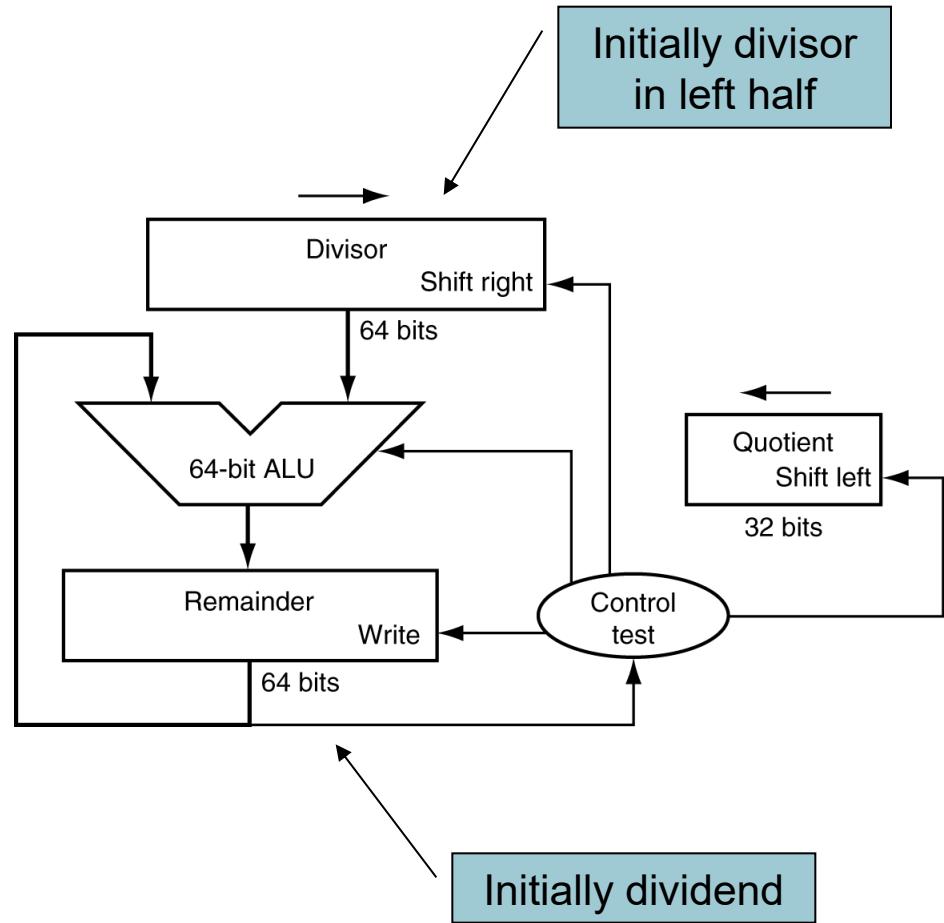
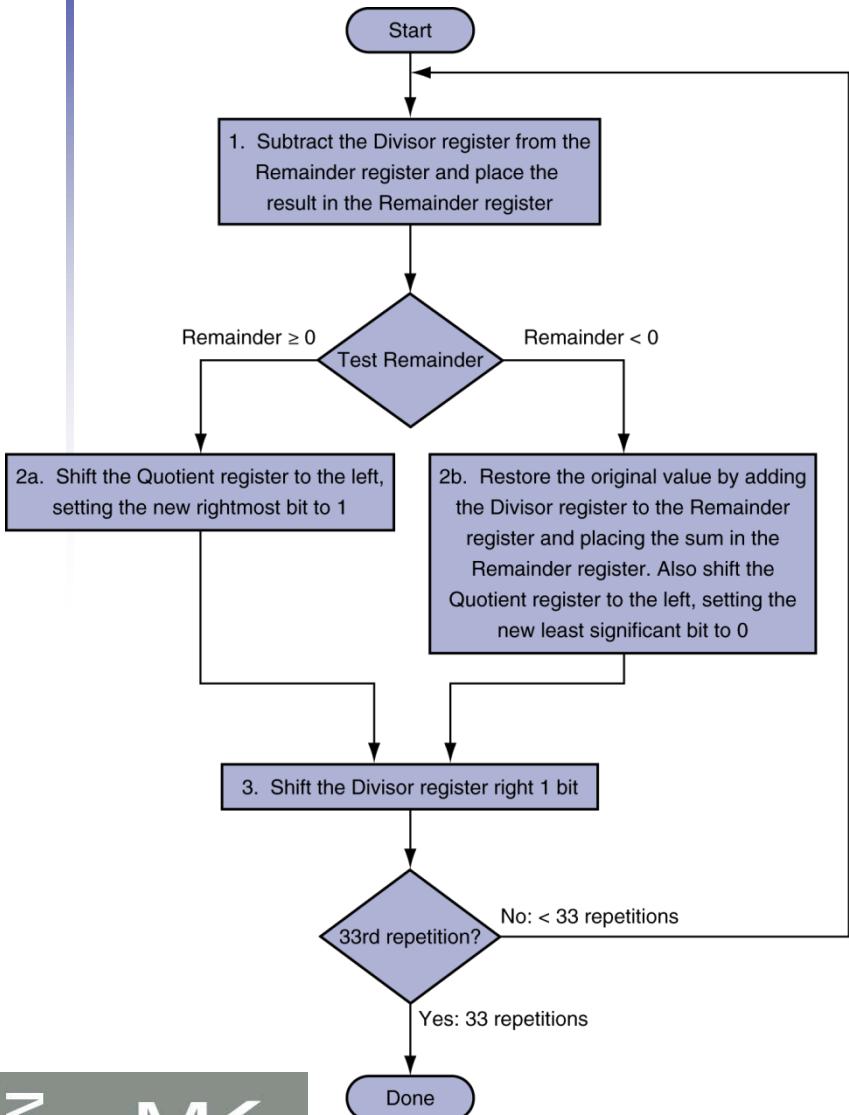
# Division



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

- Check for 0 divisor
- Long division approach
  - If divisor  $\leq$  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



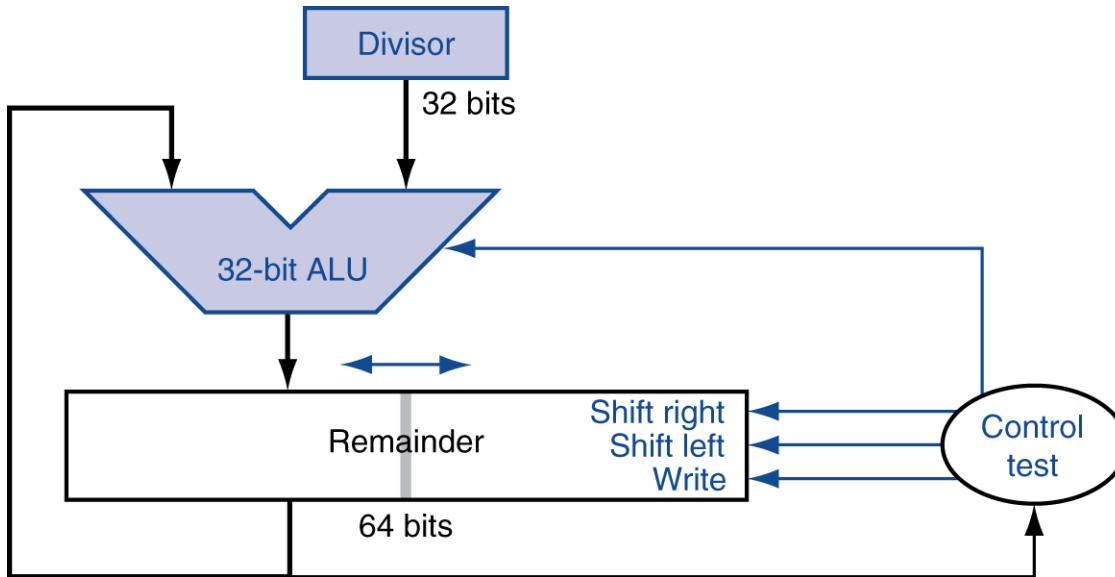
# A Divide Algorithm

## A Divide Algorithm

Using a 4-bit version of the algorithm to save pages, let's try dividing  $7_{\text{ten}}$  by  $2_{\text{ten}}$ , or  $0000\ 0111_{\text{two}}$  by  $0010_{\text{two}}$ .

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	①110 0111
	2b: Rem < 0 $\Rightarrow$ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	①111 0111
	2b: Rem < 0 $\Rightarrow$ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	①111 1111
	2b: Rem < 0 $\Rightarrow$ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	②000 0011
	2a: Rem $\geq$ 0 $\Rightarrow$ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	②000 0001
	2a: Rem $\geq$ 0 $\Rightarrow$ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

# 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 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

## MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; overflow detected
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; overflow detected
	add immediate	addi \$s1,\$s2,100	$\$s1 = \$s2 + 100$	+ constant; overflow detected
	add unsigned	addu \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; overflow undetected
	subtract unsigned	subu \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; overflow undetected
	add immediate unsigned	addiu \$s1,\$s2,100	$\$s1 = \$s2 + 100$	+ constant; overflow undetected
	move from coprocessor register	mfc0 \$s1,\$epc	$\$s1 = \$epc$	Copy Exception PC + special regs
	multiply	mult \$s2,\$s3	Hi, Lo = $\$s2 \times \$s3$	64-bit signed product in Hi, Lo
	multiply unsigned	multu \$s2,\$s3	Hi, Lo = $\$s2 \times \$s3$	64-bit unsigned product in Hi, Lo
	divide	div \$s2,\$s3	Lo = $\$s2 / \$s3$ , Hi = $\$s2 \bmod \$s3$	Lo = quotient, Hi = remainder
	divide unsigned	divu \$s2,\$s3	Lo = $\$s2 / \$s3$ , Hi = $\$s2 \bmod \$s3$	Unsigned quotient and remainder
	move from Hi	mfhi \$s1	$\$s1 = \text{Hi}$	Used to get copy of Hi
	move from Lo	mflo \$s1	$\$s1 = \text{Lo}$	Used to get copy of Lo



# Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - $-2.34 \times 10^{56}$  ← normalized
  - $+0.002 \times 10^{-4}$  ← not normalized
  - $+987.02 \times 10^9$  ← not normalized
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyy}$
- 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

double: 11 bits

single: 23 bits

double: 52 bits

S	Exponent	Fraction
---	----------	----------

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

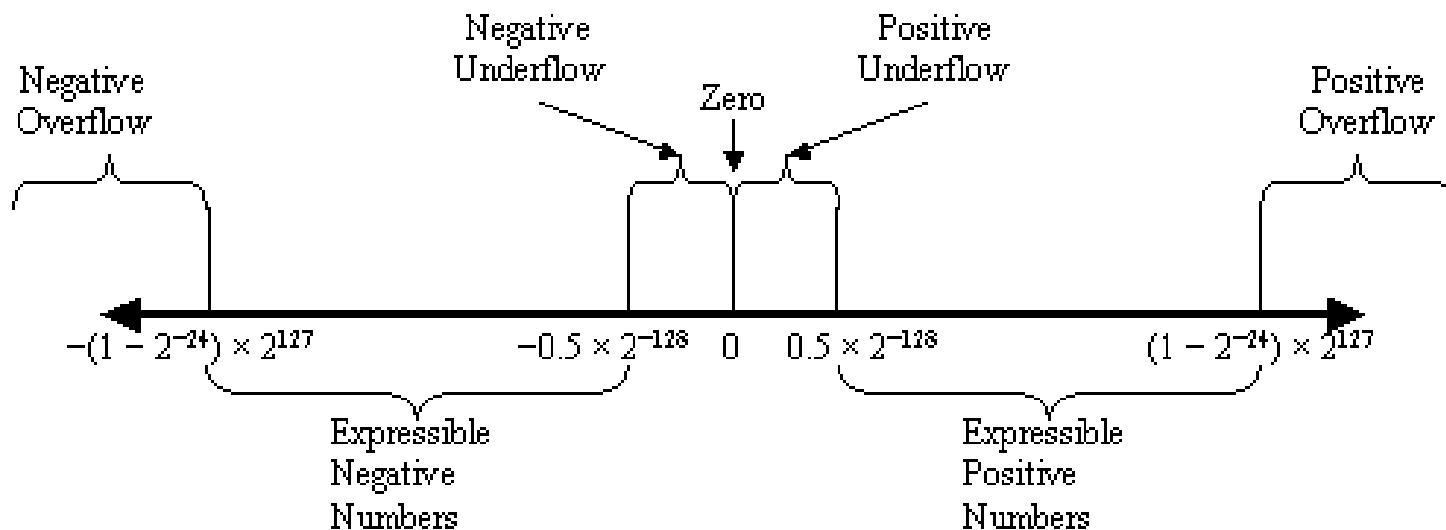
- S: sign bit (0  $\Rightarrow$  non-negative, 1  $\Rightarrow$  negative)
- Normalize significand:  $1.0 \leq |\text{significand}| < 2.0$ 
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

# 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  $\approx 2.0$
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

# Floating-Point

- Overflow: A situation in which a positive exponent becomes too large to fit in the exponent field.
- Underflow: A situation in which a negative exponent becomes too large to fit in the exponent field.



# Double-Precision Range

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

# 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

## Floating-Point Representation

### EXAMPLE

Show the IEEE 754 binary representation of the number  $-0.75_{\text{ten}}$  in single and double precision.

The number  $-0.75_{\text{ten}}$  is also

$$-3/4_{\text{ten}} \text{ or } -3/2^2_{\text{ten}}$$

It is also represented by the binary fraction

$$-11_{\text{two}} / 2^2_{\text{ten}} \text{ or } -0.11_{\text{two}}$$

In scientific notation, the value is

$$-0.11_{\text{two}} \times 2^0$$

and in normalized scientific notation, it is

$$-1.1_{\text{two}} \times 2^{-1}$$

The general representation for a single precision number is

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - 127)}$$

Subtracting the bias 127 from the exponent of  $-1.1_{\text{two}} \times 2^{-1}$  yields

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 000_{\text{two}}) \times 2^{(126 - 127)}$$

# Floating-Point Example

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 000_2) \times 2^{(126-127)}$$

The single precision binary representation of  $-0.75_{10}$  is then

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit                    8 bits                    23 bits

The double precision representation is

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit                    11 bits                    20 bits

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

32 bits

# Floating-Point Example

## EXAMPLE

### Converting Binary to Decimal Floating Point

What decimal number is represented by this single precision float?

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

|

The sign bit is 1, the exponent field contains 129, and the fraction field contains  $1 \times 2^{-2} = 1/4$ , or 0.25. Using the basic equation,

$$\begin{aligned}(-1)^s \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})} &= (-1)^1 \times (1 + 0.25) \times 2^{(129-127)} \\&= -1 \times 1.25 \times 2^2 \\&= -1.25 \times 4 \\&= -5.0\end{aligned}$$

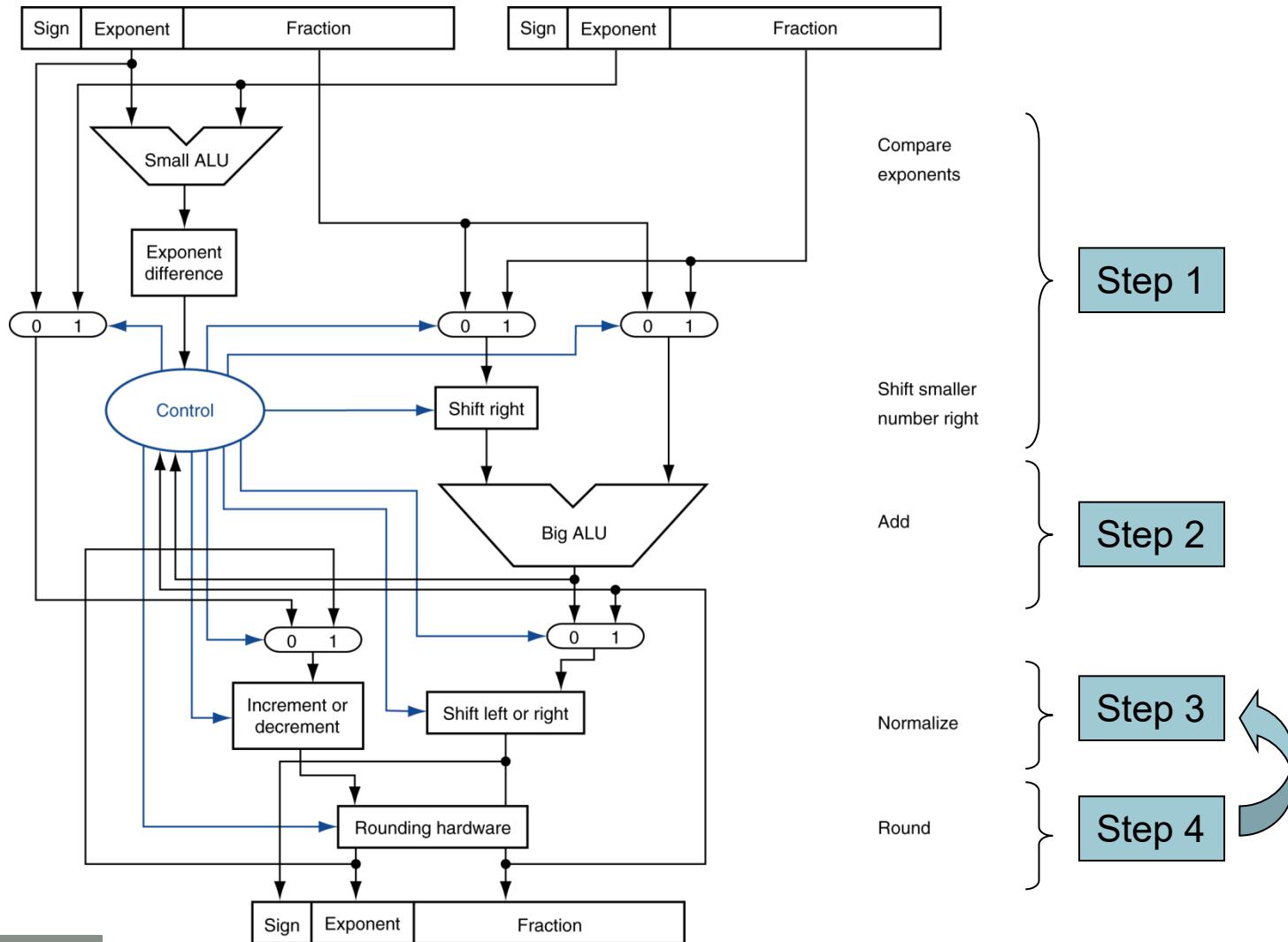
# Floating-Point Addition

- 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
  - $1.0015 \times 10^2$
- 4. Round and renormalize if necessary
  - $1.002 \times 10^2$

# 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



# Floating-Point Addition

## Binary Floating-Point Addition

Try adding the numbers  $0.5_{\text{ten}}$  and  $-0.4375_{\text{ten}}$  in binary using the algorithm in [Figure 3.14](#).

Let's first look at the binary version of the two numbers in normalized scientific notation, assuming that we keep 4 bits of precision:

$$\begin{array}{lll} 0.5_{\text{ten}} = 1/2_{\text{ten}} & = 1/2^1_{\text{ten}} \\ & = 0.1_{\text{two}} & = 0.1_{\text{two}} \times 2^0 \\ -0.4375_{\text{ten}} & = -7/16_{\text{ten}} & = -7/2^4_{\text{ten}} \\ & = -0.0111_{\text{two}} & = -0.0111_{\text{two}} \times 2^0 = -1.110_{\text{two}} \times 2^{-2} \end{array}$$

Now we follow the algorithm:

Step 1. The significand of the number with the lesser exponent ( $-1.11_{\text{two}} \times 2^{-2}$ ) is shifted right until its exponent matches the larger number:

$$-1.110_{\text{two}} \times 2^{-2} = -0.111_{\text{two}} \times 2^{-1}$$

Step 2. Add the significands:

$$1.000_{\text{two}} \times 2^{-1} + (-0.111_{\text{two}} \times 2^{-1}) = 0.001_{\text{two}} \times 2^{-1}$$

Step 3. Normalize the sum, checking for overflow or underflow:

$$\begin{aligned} 0.001_{\text{two}} \times 2^{-1} &= 0.010_{\text{two}} \times 2^{-2} = 0.100_{\text{two}} \times 2^{-3} \\ &= 1.000_{\text{two}} \times 2^{-4} \end{aligned}$$

Since  $127 \geq +4 \geq -126$ , there is no overflow or underflow. (The biased exponent would be  $-4 + 127$ , or 123, which is between 1 and 254, the smallest and largest unreserved biased exponents.)

Step 4. Round the sum:

$$1.000_{\text{two}} \times 2^{-4}$$

The sum already fits exactly in 4 bits, so there is no change to the bits due to rounding.

This sum is then

$$\begin{aligned} 1.000_{\text{two}} \times 2^{-4} &= 0.0001000_{\text{two}} = 0.0001_{\text{two}} \\ &= 1/2^4_{\text{ten}} = 1/16_{\text{ten}} = 0.0625_{\text{ten}} \end{aligned}$$

This sum is what we would expect from adding  $0.5_{\text{ten}}$  to  $-0.4375_{\text{ten}}$ .



# 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  $32 \times 64$
- 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:  lw $f16, const5($gp)  
      lw $f18, const9($gp)  
      div.s $f16, $f16, $f18  
      lw $f18, const32($gp)  
      sub.s $f18, $f12, $f18  
      mul.s $f0, $f16, $f18  
      jr $ra
```

# FP Example: Array Multiplication

- $X = X + Y \times Z$ 
  - All  $32 \times 32$  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

...

```
s11 $t0, $s0, 5          # $t0 = i*32 (size of row of y)
addu $t0, $t0, $s2        # $t0 = i*size(row) + k
s11 $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
```

## MIPS floating-point operands

Name	Example	Comments
32 floating-point registers	\$f0, \$f1, \$f2, . . . , \$f31	MIPS floating-point registers are used in pairs for double precision numbers.
$2^{30}$ memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

## MIPS floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	add.s \$f2,\$f4,\$f6	$f2 = f4 + f6$	FP add (single precision)
	FP subtract single	sub.s \$f2,\$f4,\$f6	$f2 = f4 - f6$	FP sub (single precision)
	FP multiply single	mul.s \$f2,\$f4,\$f6	$f2 = f4 \times f6$	FP multiply (single precision)
	FP divide single	div.s \$f2,\$f4,\$f6	$f2 = f4 / f6$	FP divide (single precision)
	FP add double	add.d \$f2,\$f4,\$f6	$f2 = f4 + f6$	FP add (double precision)
	FP subtract double	sub.d \$f2,\$f4,\$f6	$f2 = f4 - f6$	FP sub (double precision)
	FP multiply double	mul.d \$f2,\$f4,\$f6	$f2 = f4 \times f6$	FP multiply (double precision)
	FP divide double	div.d \$f2,\$f4,\$f6	$f2 = f4 / f6$	FP divide (double precision)
Data transfer	load word copr. 1	lwcl \$f1,100(\$s2)	$f1 = \text{Memory}[\$s2 + 100]$	32-bit data to FP register
	store word copr. 1	swcl \$f1,100(\$s2)	$\text{Memory}[\$s2 + 100] = f1$	32-bit data to memory
Conditional branch	branch on FP true	bc1t 25	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	bc1f 25	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s \$f2,\$f4	if ( $f2 < f4$ ) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d \$f2,\$f4	if ( $f2 < f4$ ) cond = 1; else cond = 0	FP compare less than double precision

## MIPS floating-point machine language

Name	Format	Example						Comments
add.s	R	17	16	6	4	2	0	add.s \$f2,\$f4,\$f6
sub.s	R	17	16	6	4	2	1	sub.s \$f2,\$f4,\$f6
mul.s	R	17	16	6	4	2	2	mul.s \$f2,\$f4,\$f6
div.s	R	17	16	6	4	2	3	div.s \$f2,\$f4,\$f6
add.d	R	17	17	6	4	2	0	add.d \$f2,\$f4,\$f6
sub.d	R	17	17	6	4	2	1	sub.d \$f2,\$f4,\$f6
mul.d	R	17	17	6	4	2	2	mul.d \$f2,\$f4,\$f6
div.d	R	17	17	6	4	2	3	div.d \$f2,\$f4,\$f6
lwcl	I	49	20	2	100			lwcl \$f2,100(\$s4)
swcl	I	57	20	2	100			swcl \$f2,100(\$s4)
bc1t	I	17	8	1	25			bc1t 25
bc1f	I	17	8	0	25			bc1f 25
c.lt.s	R	17	16	4	2	0	60	c.lt.s \$f2,\$f4
c.lt.d	R	17	17	4	2	0	60	c.lt.d \$f2,\$f4
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits

# 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

# Accurate Arithmetic

## Rounding with Guard Digits

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

First we must shift the smaller number to the right to align the exponents, so  $2.56_{\text{ten}} \times 10^0$  becomes  $0.0256_{\text{ten}} \times 10^2$ . Since we have guard and round digits, we are able to represent the two least significant digits when we align exponents. The guard digit holds 5 and the round digit holds 6. The sum is

$$\begin{array}{r} 2.3400_{\text{ten}} \\ + 0.0256_{\text{ten}} \\ \hline 2.3656_{\text{ten}} \end{array}$$

Thus the sum is  $2.3656_{\text{ten}} \times 10^2$ . Since we have two digits to round, we want values 0 to 49 to round down and 51 to 99 to round up, with 50 being the tiebreaker. Rounding the sum up with three significant digits yields  $2.37_{\text{ten}} \times 10^2$ .

Doing this *without* guard and round digits drops two digits from the calculation. The new sum is then

$$\begin{array}{r} 2.34_{\text{ten}} \\ + 0.02_{\text{ten}} \\ \hline 2.36_{\text{ten}} \end{array}$$

The answer is  $2.36_{\text{ten}} \times 10^2$ , off by 1 in the last digit from the sum above.



# Subword Parallelism

- 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
  - $8 \times 80$ -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
<code>FILD</code> mem/ST(i)	<code>FIADDP</code> mem/ST(i)	<code>FICOMP</code>	<code>FPATAN</code>
<code>FISTP</code> mem/ST(i)	<code>FISUBRP</code> mem/ST(i)	<code>FIUCOMP</code>	<code>F2XMI</code>
<code>FLDPI</code>	<code>FIMULP</code> mem/ST(i)	<code>FSTSW</code> AX/mem	<code>FCOS</code>
<code>FLD1</code>	<code>FIDIVRP</code> mem/ST(i)		<code>FPTAN</code>
<code>FLDZ</code>	FSQRT FABS FRNDINT		<code>FPREM</code> <code>FPSIN</code> <code>FYL2X</code>

- 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  $4 \times 128
  - Extended to 8 registers in AMD64/EM64T$
- Can be used for multiple FP operands
  - $2 \times 64$
  - $4 \times 32$
  - 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
vfmaadd231pd  (%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.,  $-5 / 4$ 
    - $11111011_2 \gg 2 = 11111110_2 = -2$
    - Rounds toward  $-\infty$
  - c.f.  $11111011_2 \ggg 2 = 00111110_2 = +62$

# 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