

# **Chapter 3**

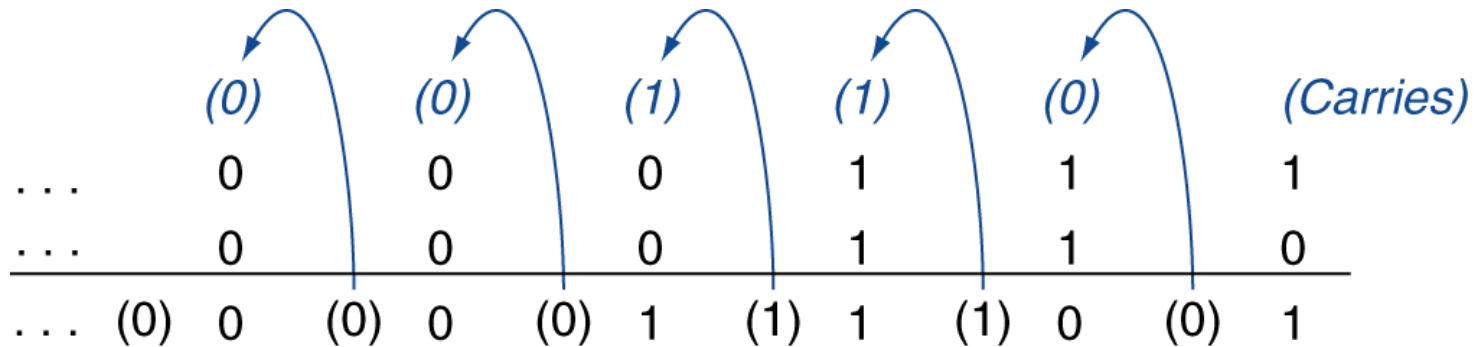
## **Arithmetic for Computers**

# 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

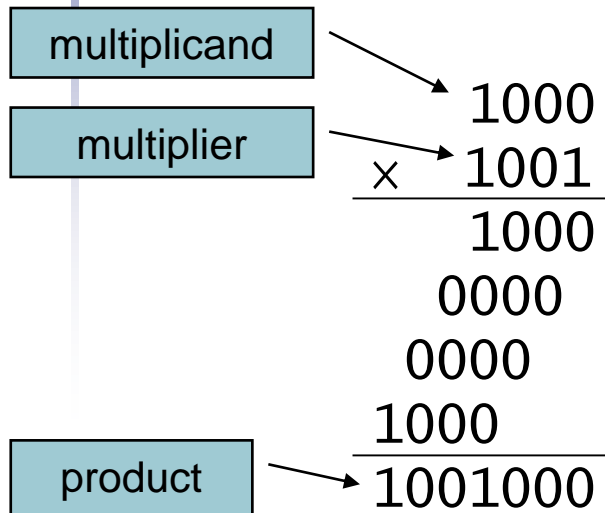
- Some languages (*e.g.*, C) ignore overflow
  - Use MIPS `addu`, `addui`, `subu` instructions
- Other languages (*e.g.*, Ada, Fortran) require raising an exception
  - Use MIPS `add`, `addi`, `sub` instructions
  - On overflow, invoke exception handler
    - Save PC in exception program counter (`EPC`)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

# Arithmetic for Multimedia

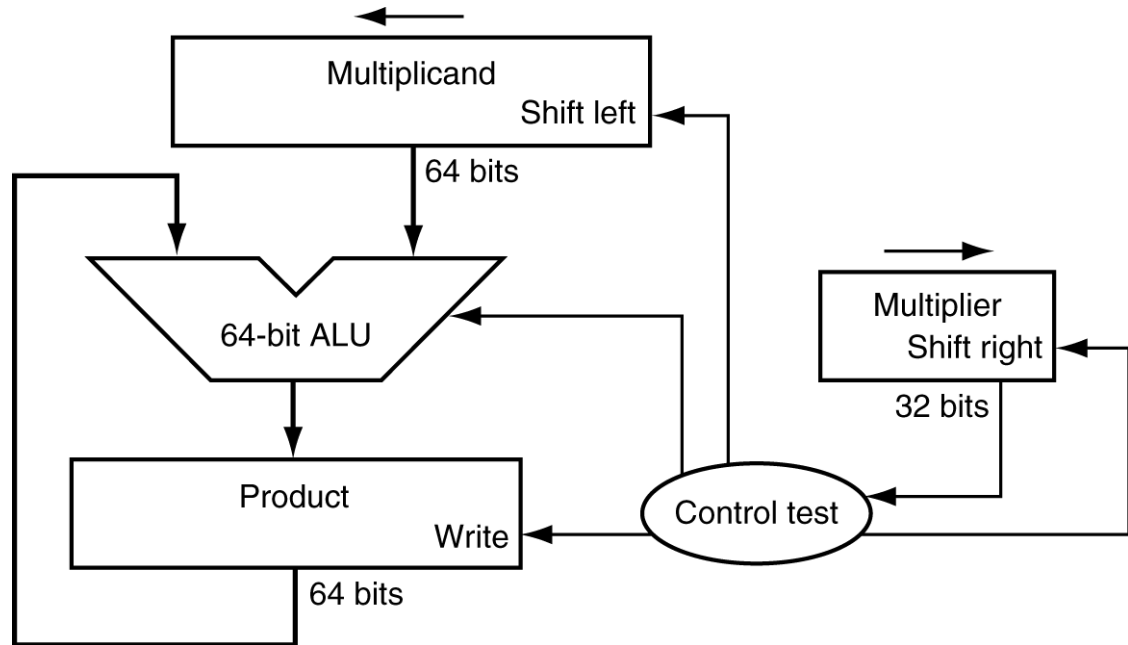
- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
  - SIMD (single-instruction, multiple-data)
- Saturating operations
  - On overflow, result is largest representable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

# Multiplication

- Start with long-multiplication approach

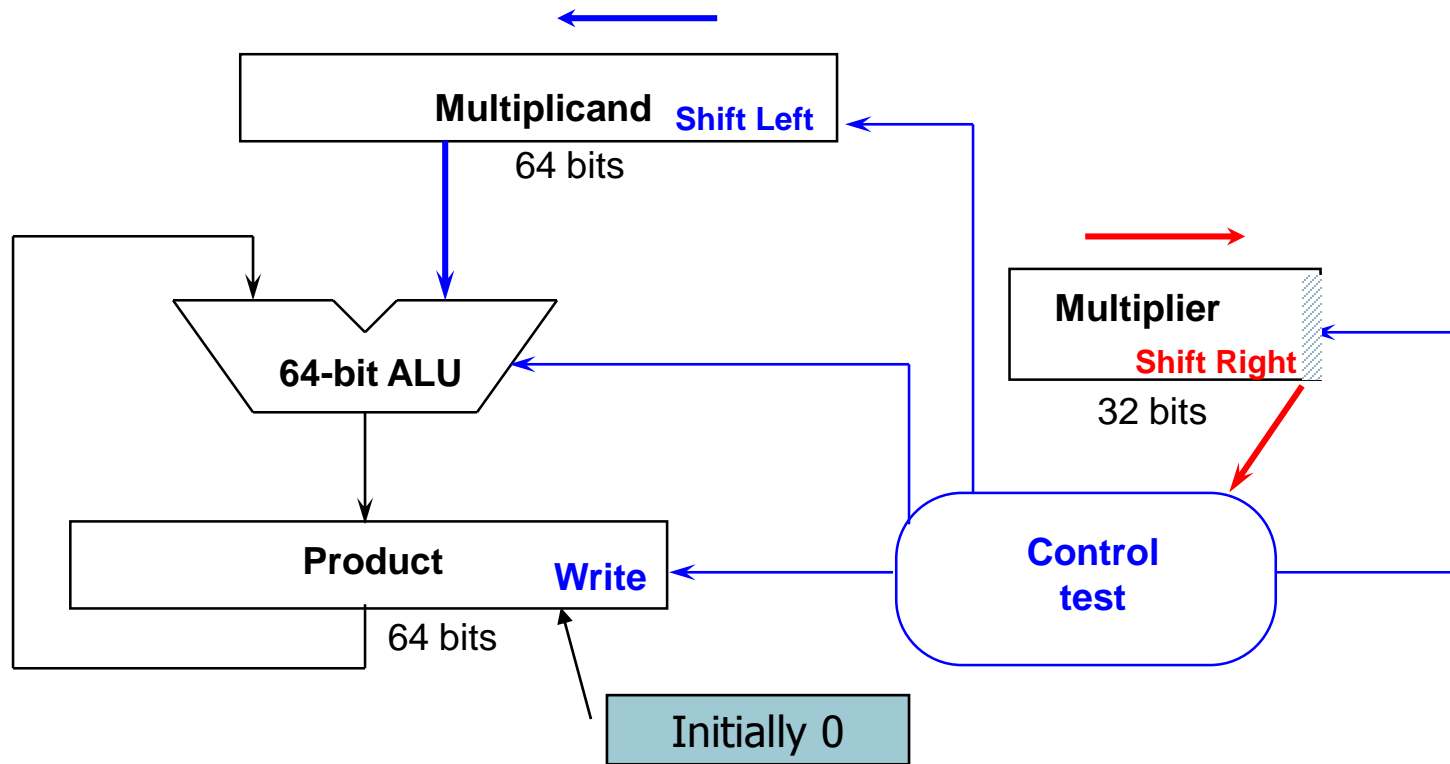


Length of product is the sum of operand lengths



# Multiplication Hardware

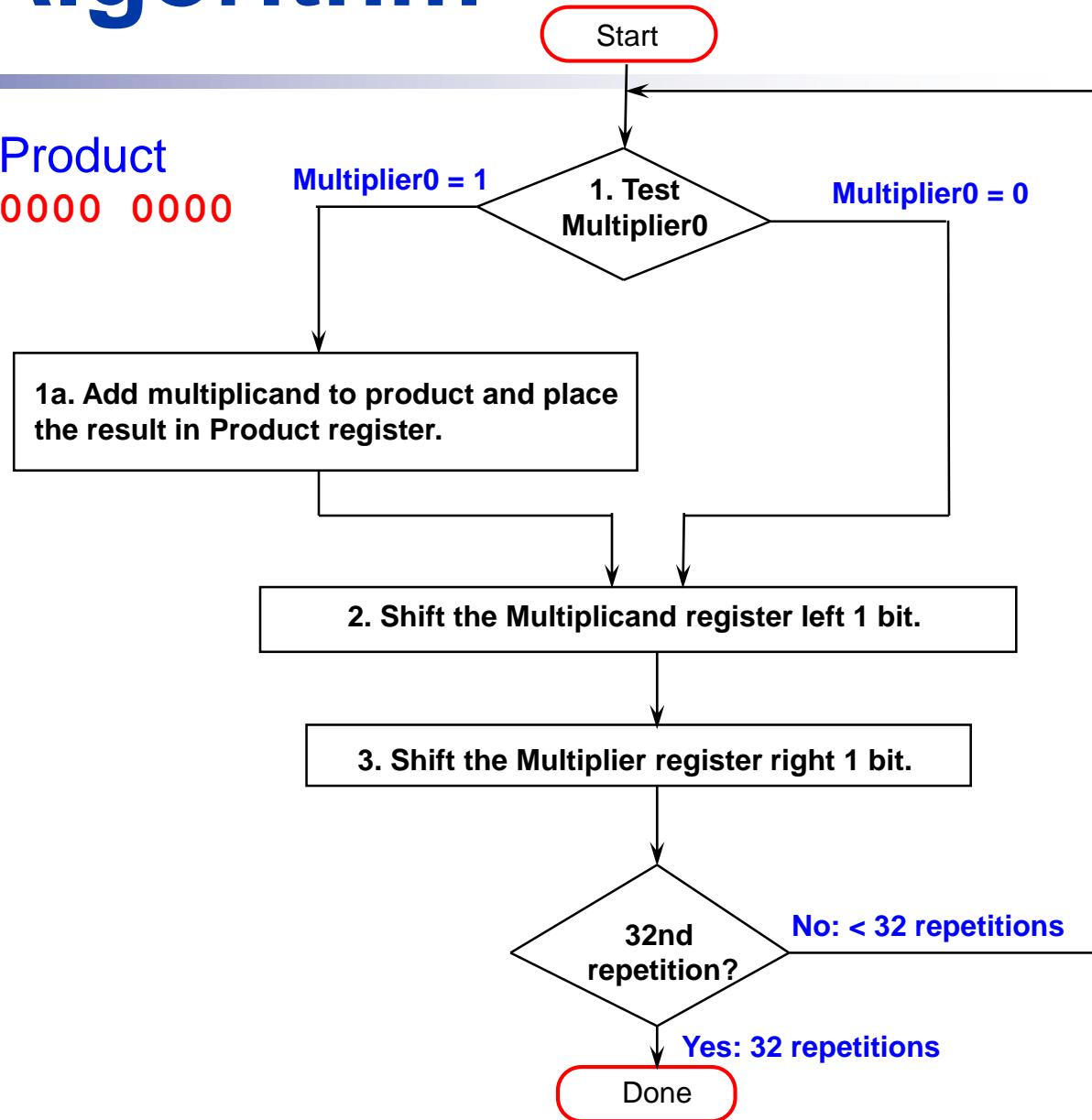
- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit Multiplier reg.





# Multiply Algorithm

Multiplier	Multiplicand	Product
0011	00000010	0000 0000

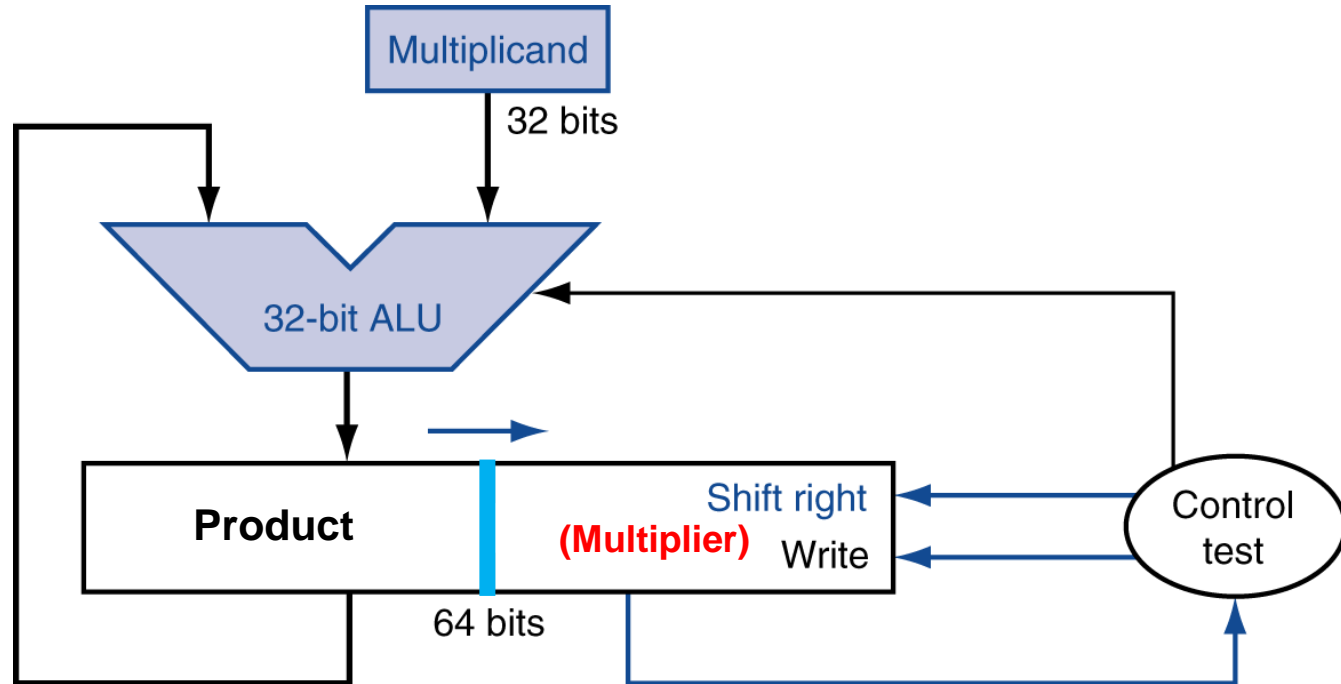


# Multiply Algorithm

	Mplier: 0011	Mcand: 0000 0010	P: 0000 0000
1a. 1=>P=P+Mcand	Mplier: 0011	Mcand: 0000 0010	P: <u>0000 0010</u>
2. Shl Mcand	Mplier: 0011	<u>Mcand: 0000 0100</u>	P: 0000 0010
3. Shr Mplier	<u>Mplier: 0001</u>	Mcand: 0000 0100	P: 0000 0010
1a. 1=>P=P+Mcand	Mplier: 0001	Mcand: 0000 0100	P: <u>0000 0110</u>
2. Shl Mcand	Mplier: 0001	<u>Mcand: 0000 1000</u>	P: 0000 0110
3. Shr Mplier	<u>Mplier: 0000</u>	Mcand: 0000 1000	P: 0000 0110
1. 0=>nop	Mplier: 0000	Mcand: 0000 1000	P: 0000 0110
2. Shl Mcand	Mplier: 0000	<u>Mcand: 0001 0000</u>	P: 0000 0110
3. Shr Mplier	<u>Mplier: 0000</u>	Mcand: 0001 0000	P: 0000 0110
1. 0=>nop	Mplier: 0000	Mcand: 0001 0000	P: 0000 0110
2. Shl Mcand	Mplier: 0000	<u>Mcand: 0010 0000</u>	P: 0000 0110
3. Shr Mplier	<u>Mplier: 0000</u>	Mcand: 0010 0000	P: 0000 0110
	Mplier: 0000	Mcand: 0010 0000	P: 0000 0110

# Optimized Multiplier

- Perform steps in parallel: add/shift



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

# Booth's Algorithm

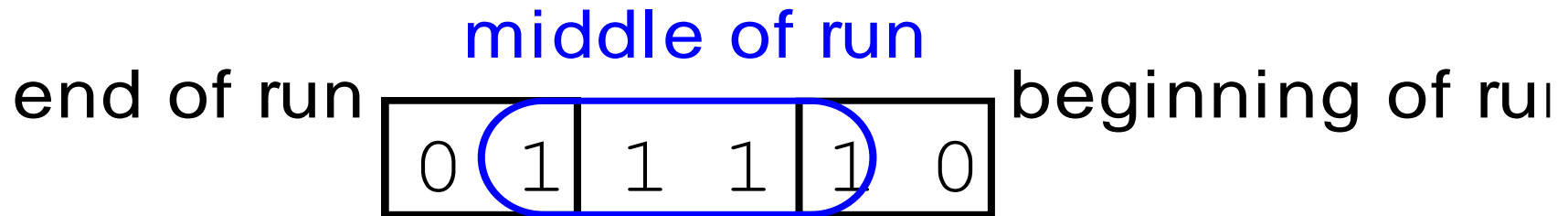
- Example  $2 \times 6 = 0010 \times 0110$

		0010	
x		0110	
+		0000	shift (0 in multiplier)
+		0010	add (1 in multiplier)
+		0010	add (1 in multiplier)
+		0000	shift (0 in multiplier)
		<hr/>	
		00001100	

- ALU with add or subtract gets same result in more than one way

- $6 = -2 + 8$  , or  
 $0110 = -00010 + 01000 (= 11110 + 01000)$

# Booth's Algorithm Insight



Current Bit	Bit to the Right	Explanation	Example
1	0	Beginning of a run of 1s	000111 <u>1</u> 000
1	1	Middle of a run of 1s	00011 <u>11</u> 000
0	1	End of a run of 1s	00 <u>01</u> 111000
0	0	Middle of a run of 0s	0 <u>00</u> 1111000

**Originally for Speed since shift faster than add**

# Booth's Algorithm

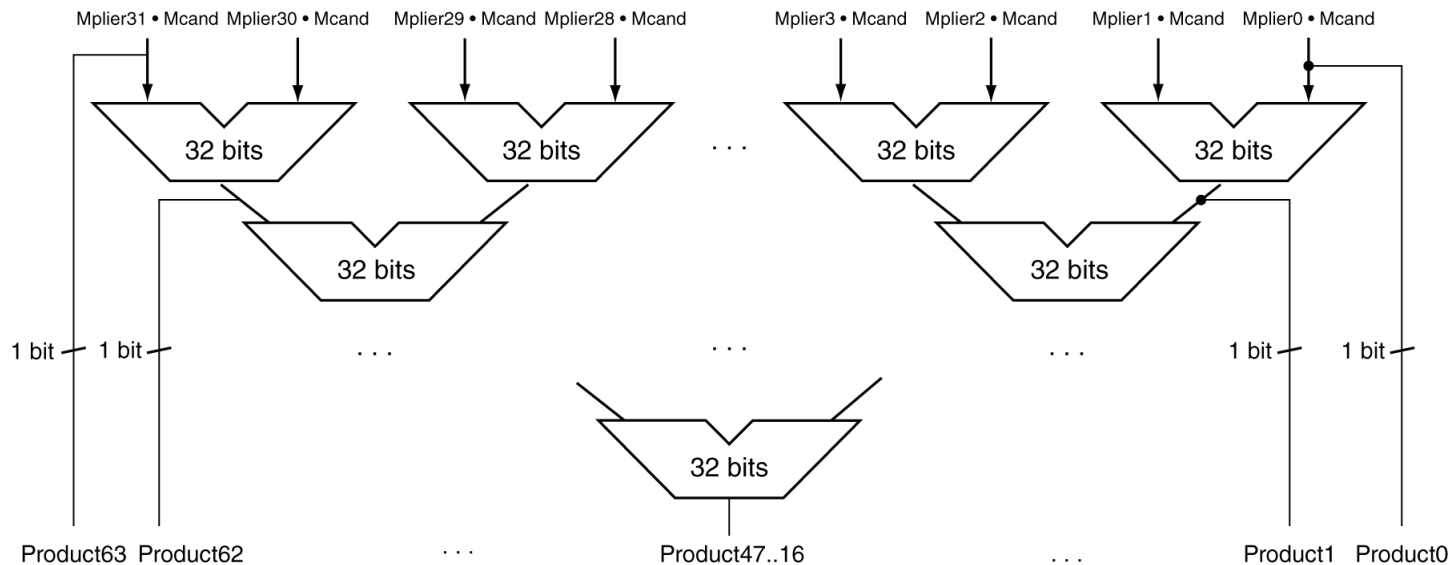
1. Depending on the current and previous bits, do one of the following:
  - 00: a. Middle of a string of 0s, so no arithmetic operations.
  - 01: b. End of a string of 1s, so add the multiplicand to the left half of the product.
  - 10: c. Beginning of a string of 1s, so subtract the multiplicand from the left half of the product.
  - 11: d. Middle of a string of 1s, so no arithmetic operation.
2. As in the previous algorithm, shift the Product register right (arith) 1 bit.

# Example of (2 x -3)

Operation	Multiplicand	Product	next?
0. initial value	0010	0000 1101 0	10 -> sub
1a. $P = P - m$	1110	+ 1110 1110 1101 0	shift P (sign ext)
1b.	0010	1111 0110 1 + 0010	01 -> add
2a.		0001 0110 1	shift P
2b.	0010	0000 1011 0 + 1110	10 -> sub
3a.	0010	1110 1011 0	shift
3b.	0010	1111 0101 1	11 -> nop
4a		1111 0101 1	shift
4b.	0010	1111 1010 1	done

# Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff



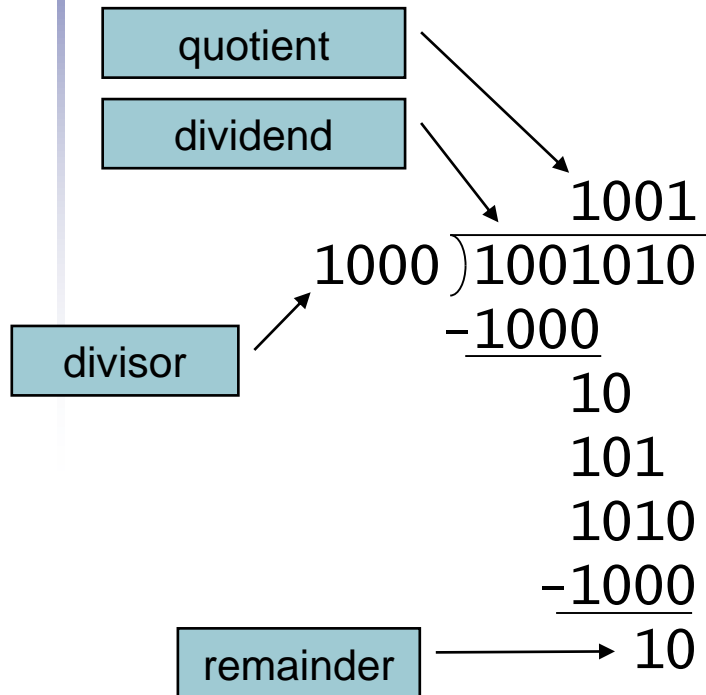
- Can be pipelined
  - Several multiplication performed in parallel



# 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

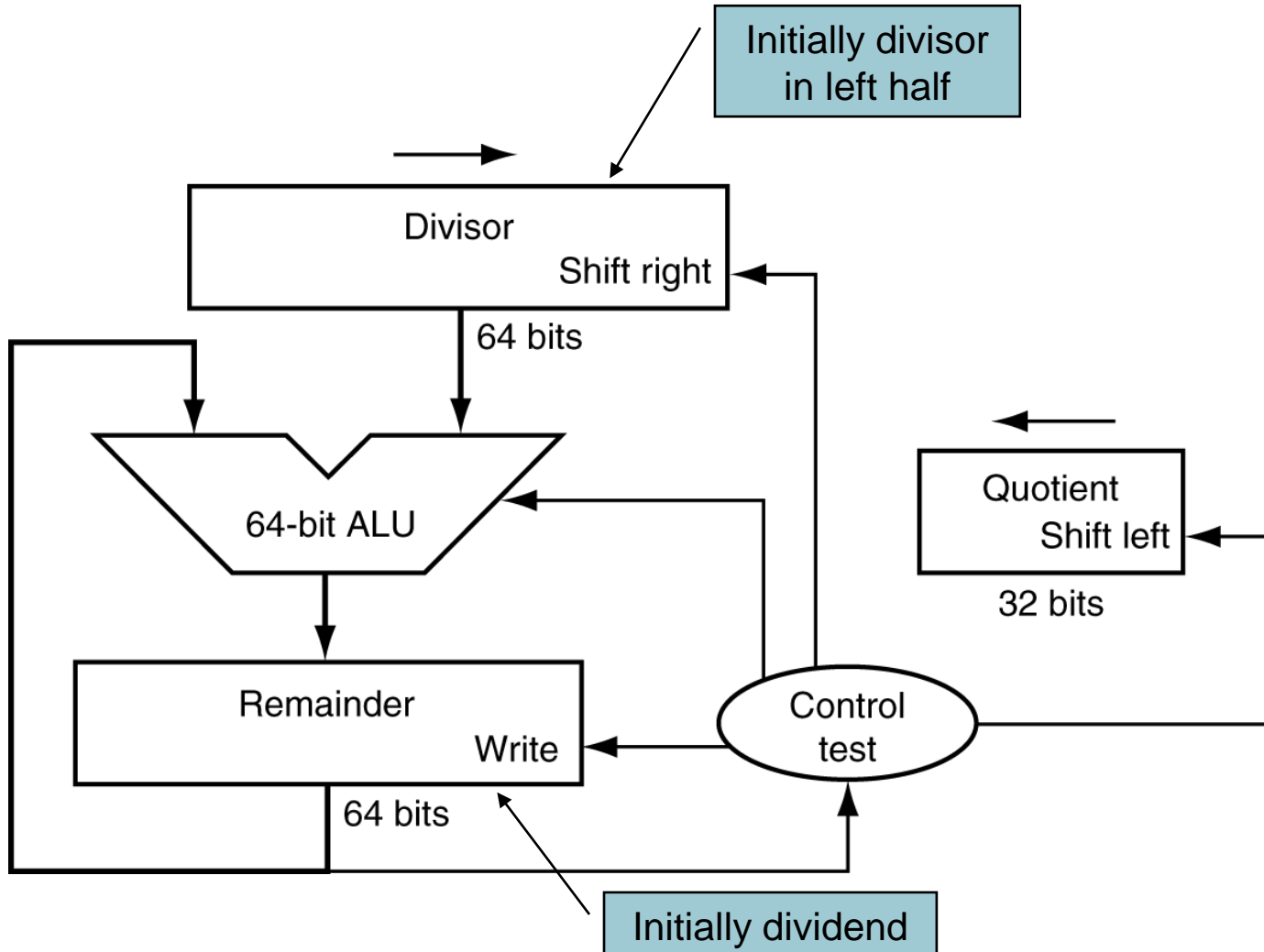
# Division



*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

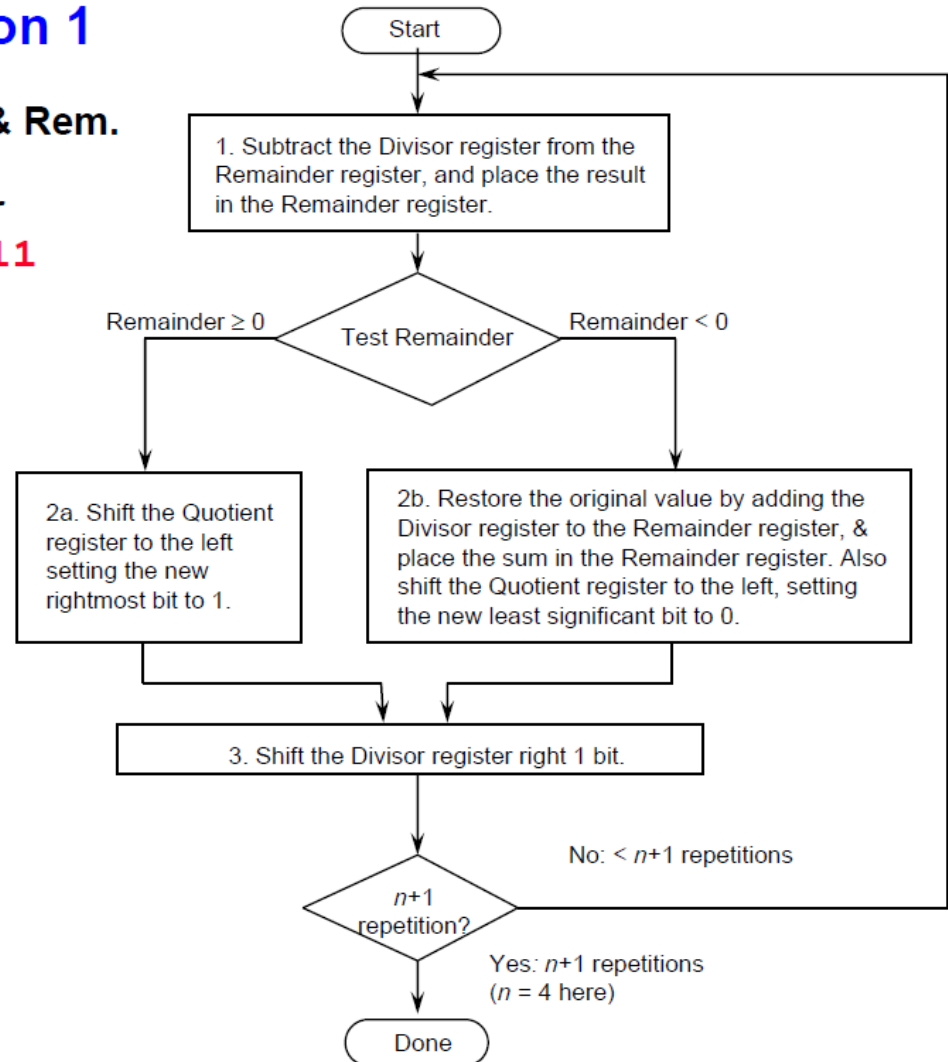


# Divide Algorithm

## Divide Algorithm Version 1

- Takes  $n+1$  steps for  $n$ -bit Quotient & Rem.

Quotient	Divisor	Remainder
0000	0010 0000	0000 0111



# Divide Algorithm

Q: 0000

D: 0010 0000 R: 0000 0111

1: R= R-D Q: 0000

D: 0010 0000 R: 1110 0111

2b: +D, shlQ, 0 Q: 0000

D: 0010 0000 R: 0000 0111

3: Shr D Q: 0000

D: 0001 0000 R: 0000 0111

1: R= R-D Q: 0000

D: 0001 0000 R: 1110 0111

2b: +D, shlQ, 0 Q: 0000

D: 0001 0000 R: 0000 0111

3: Shr D Q: 0000

D: 0000 1000 R: 0000 0111

1: R= R-D Q: 0000

D: 0000 1000 R: 1110 0111

2b: +D, shlQ, 0 Q: 0000

D: 0000 1000 R: 0000 0111

3: Shr D Q: 0000

D: 0000 0100 R: 0000 0111

1: R= R-D Q: 0000

D: 0000 0100 R: 0000 0011

2a: shlQ, 1 Q: 0001

D: 0000 0100 R: 0000 0011

3: Shr D Q: 0000

D: 0000 0010 R: 0000 0011

1: R= R-D Q: 0000

D: 0000 0010 R: 0000 0001

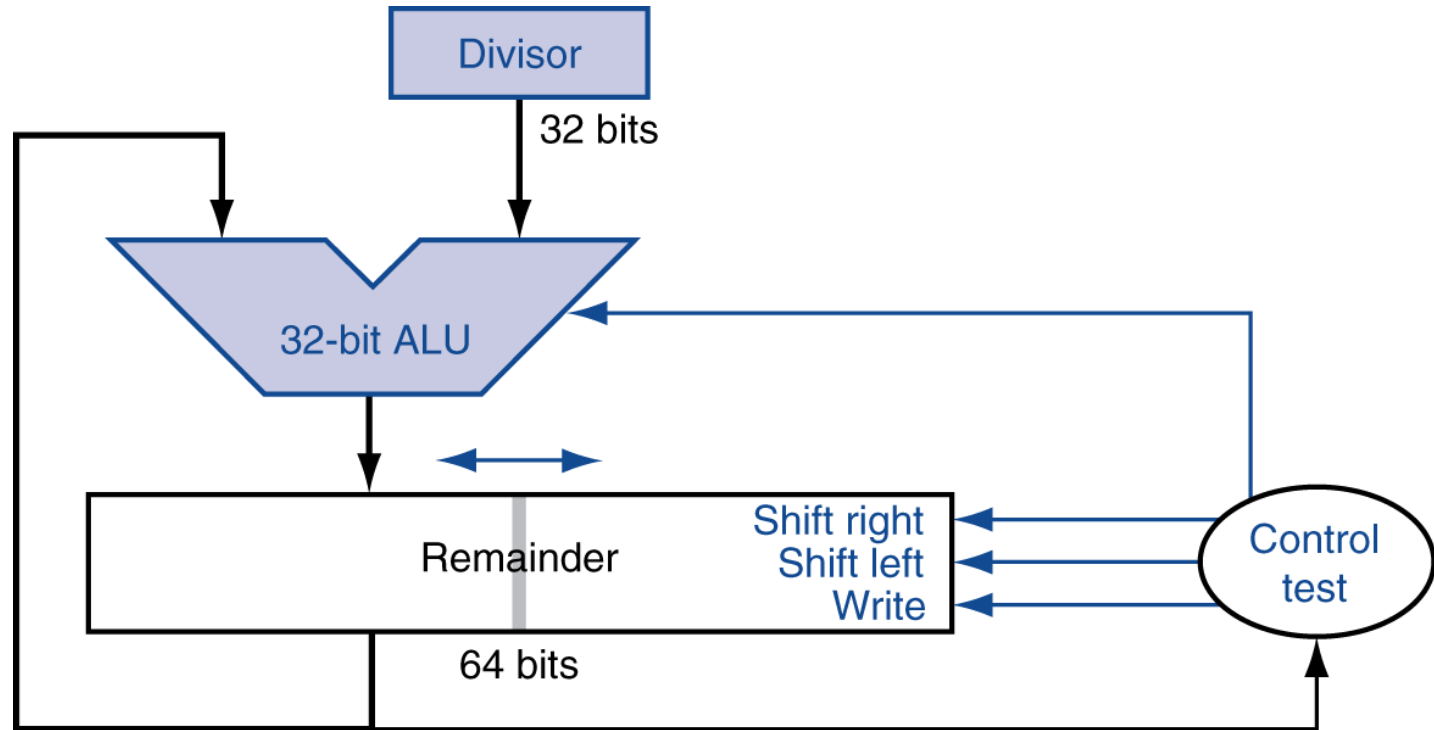
2a: shlQ, 1 Q: 0011

D: 0000 0010 R: 0000 0001

3: Shr D Q: 0000

D: 0000 0001 R: 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 division)  
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



# 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
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- 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  
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 = 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  $\approx 2.0$
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

# Double-Precision Range

- Exponents 0000...00 and 1111...11 reserved
- Smallest value
  - Exponent: 000000000001  
 $\Rightarrow$  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: 111111111110  
 $\Rightarrow$  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

- Represent  $-0.75$ 
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - $S = 1$
  - Fraction =  $1000\dots00_2$
  - Exponent =  $-1 + \text{Bias}$ 
    - Single:  $-1 + 127 = 126 = 01111110_2$
    - Double:  $-1 + 1023 = 1022 = 011111111110_2$
- Single:  $10111111101000\dots00$
- Double:  $10111111111101000\dots00$

# Floating-Point Example

- What number is represented by the single-precision float

11000000101000...00

- $S = 1$

- Fraction =  $01000...00_2$

- Exponent =  $10000001_2 = 129$

- $$\begin{aligned} x &= (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)} \\ &= (-1) \times 1.25 \times 2^2 \\ &= -5.0 \end{aligned}$$



# Extra Bits

- **Floating Point numbers are like piles of sand; every time you move one you lose a little sand, but you pick up a little dirt**
  - **How many extra bits? IEEE: As if computed the result exactly and rounded.**

- **Addition:**

1.xxxxx	1.xxxxx	1.xxxxx
+ 1.xxxxx	0.001xxxxx	0.01xxxxx
<hr/>	<hr/>	<hr/>
1x.xxxxxy	1.xxxxxyyy	1x.xxxxxyyy
post-normalization	pre-normalization	pre and post

- **Guard Digits:** digits to the right of the first  $p$  digits of significand to guard against loss of digits —can later be shifted left into first  $p$  places during normalization.
  - Addition: carry-out shifted in
  - Subtraction: borrow digit and guard
  - Multiplication: carry and guard, division requires guard

# Rounding Digits

- Normalized result, but some non-zero digits to the right of the significand --> the number should be rounded

- e.g.,  $B = 10$ ,  $p = 3$ :

0	2	1.69	=	1.6900	*	10	2-bias
-							
0	0	7.85	=	- .0785	*	10	2-bias
-----							
0	2	1.61	=	1.6115	*	10	2-bias

- One round digit must be carried to the right of the guard digit so that after a normalizing left shift, the result can be rounded, according to the value of the round digit
- IEEE Standard: four rounding modes:
  - round to nearest (default)
  - round towards plus infinity
  - round towards minus infinity
  - round towards 0

# Sticky Bit

- Additional bit to the right of the round digit to better fine tune rounding

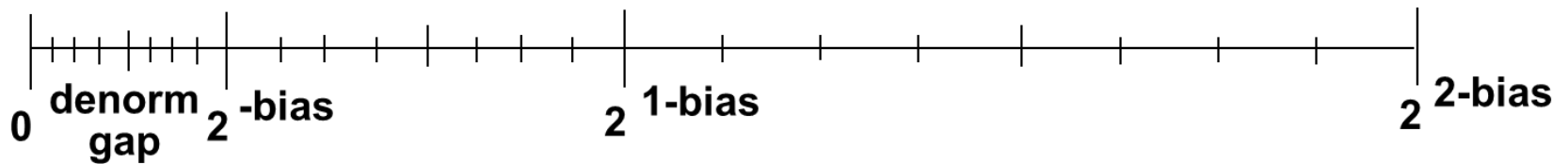
$$\begin{array}{r}
 \text{d0 . d1 d2 d3 . . . dp-1 0 0 0} \\
 + \text{ 0 . 0 0 X . . . X X X S} \\
 \hline
 \text{X X S}
 \end{array}
 \quad \leftarrow \text{Sticky bit: set to 1 if any 1 bits fall off the end of the round digit}$$
  

$$\begin{array}{r}
 \text{d0 . d1 d2 d3 . . . dp-1 0 0 0} \\
 - \text{ 0 . 0 0 X . . . X X X 0} \\
 \hline
 \text{X X 0}
 \end{array}
 \quad \begin{array}{r}
 \text{d0 . d1 d2 d3 . . . dp-1 0 0 0} \\
 - \text{ 0 . 0 0 X . . . X X X 1} \\
 \hline
 \text{generates a borrow}
 \end{array}
 \rightarrow$$

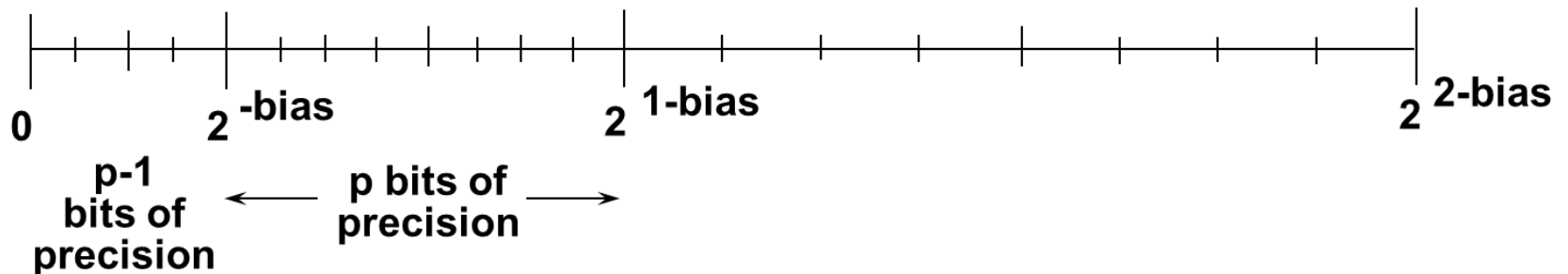
- **Rounding Summary:**
  - Radix-2 minimizes wobble in precision
  - Normal operations in +, -, \*, / require 1 carry/borrow bit + 1 guard digit
  - One round digit needed for correct rounding
  - Sticky bit needed when round digit is B/2 for max accuracy
  - Rounding to nearest has mean error = 0 if uniform distribution of digits are assumed

# Denormalized Numbers

- Normalize a number with hidden bit,  $B=2$ ,  $p=4$ 
  - The gap between 0 and the next representable number is much larger than the gaps between nearby representable numbers.



- IEEE standard uses **denormalized** numbers to fill in the gap, making the distances between numbers near 0 more alike (**Gradual underflow**) => same space, half as many values!



# Denormal Numbers


- Exponent = 000...0  $\Rightarrow$  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!



# Infinites and NaNs

- Exponent = 111...1, Fraction = 000...0

- **$\pm$ Infinity**

S	1 . . . 1	0 . . . 0
---	-----------	-----------

- Can be used in subsequent calculations, avoiding need for overflow check

- Exponent = 111...1, Fraction  $\neq$  000...0

- **Not-a-Number (NaN)**

S	1 . . . 1	non-zero
---	-----------	----------

- Indicates illegal or undefined result

- e.g., 0.0 / 0.0

- Can be used in subsequent calculations

# Floating-Point Addition

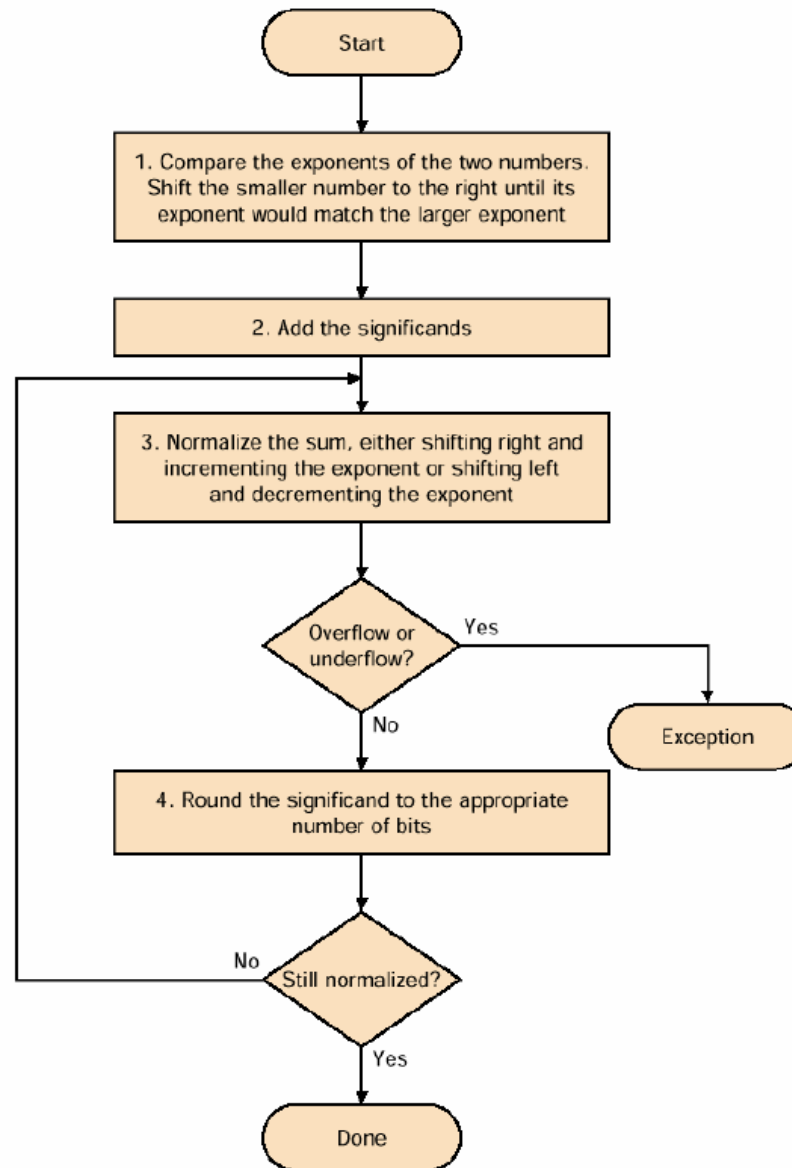
- **Consider a 4-digit decimal example**
  - $9.999 \times 10^1 + 1.610 \times 10^{-1}$
- **1. Align decimal points**
  - Shift number with smaller exponent
  - $9.999 \times 10^1 + 0.016 \times 10^1$
- **2. Add significands**
  - $9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$
- **3. Normalize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

# 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



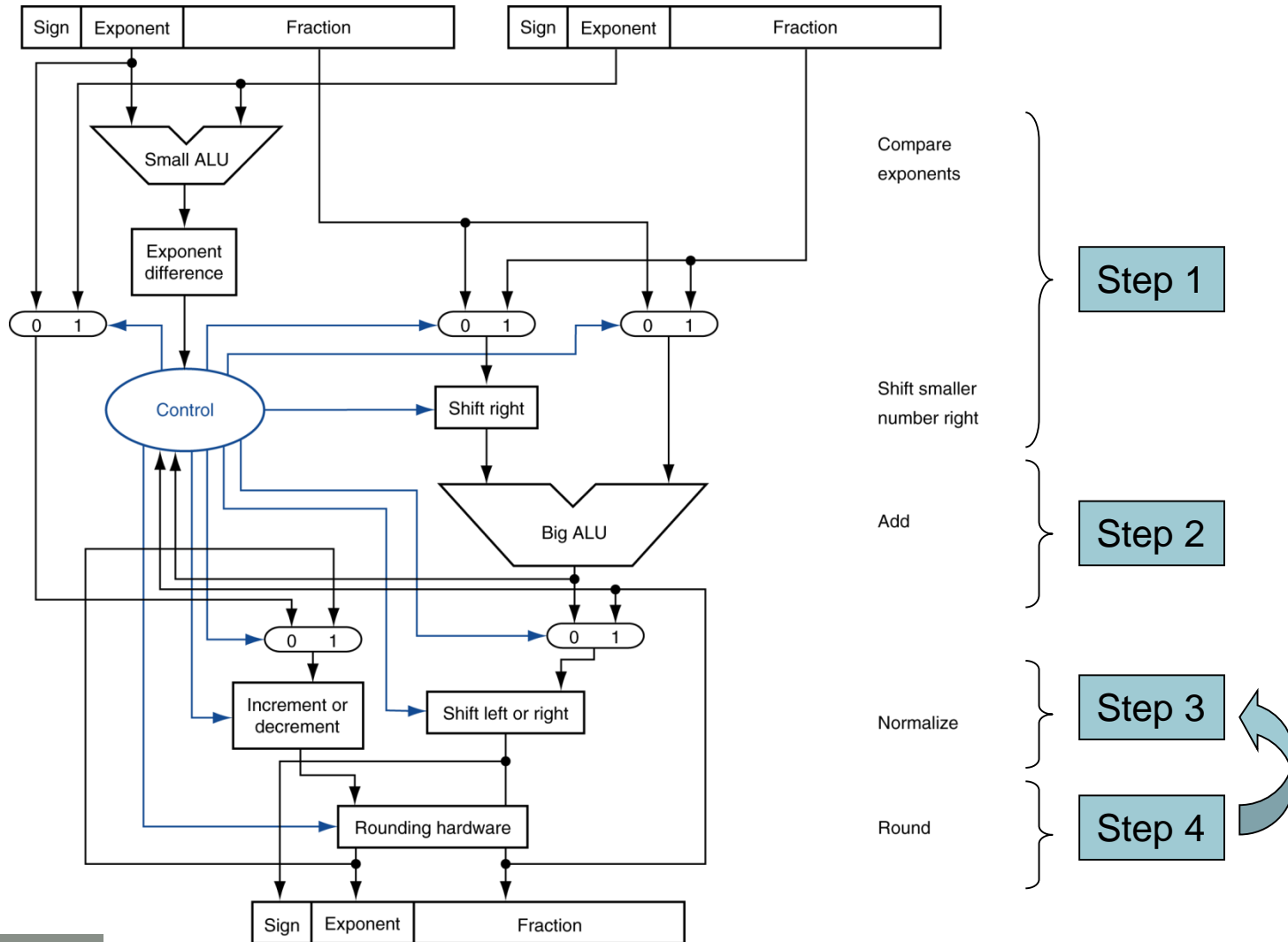
# FP Addition Algorithm



# 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



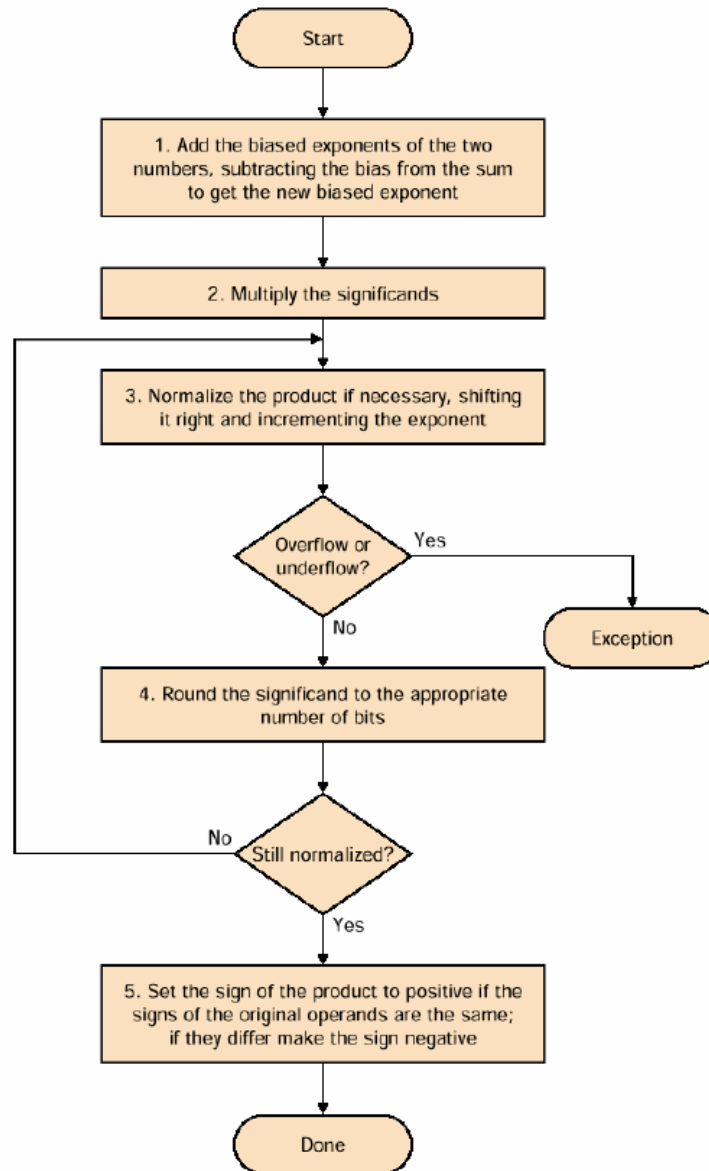
# Floating-Point Multiplication

- **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**
  - $1.0212 \times 10^6$
- **4. Round and renormalize if necessary**
  - $1.021 \times 10^6$
- **5. Determine sign of result from signs of operands**
  - $+1.021 \times 10^6$

# Floating-Point Multiplication

- **Now consider a 4-digit binary example**
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$  ( $0.5 \times -0.4375$ )
- **1. Add exponents**
  - Unbiased:  $-1 + -2 = -3$
  - Biased:  $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- **2. Multiply significands**
  - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- **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  $\times$  -ve  $\Rightarrow$  -ve**
  - $-1.110_2 \times 2^{-3} = -0.21875$

# FP Multiply Algorithm



# 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 (c1)
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision:  $\$f0, \$f1, \dots \$f31$
  - Paired for double-precision:  $\$f0/\$f1, \$f2/\$f3, \dots$ 
    - Release 2 of MIPS ISA supports  $32 \times 64$ -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
  - $lw c1, ld c1, sw c1, sd c1$ 
    - e.g.,  $ld c1 \$f8, 32(\$sp)$



# 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` can be `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)  
     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

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

...

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

# 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
FILD mem/ST(i) FISTP mem/ST(i) FLDPI FLD1 FLDZ	F <del>I</del> ADDP mem/ST(i) F <del>I</del> SUB <del>R</del> P mem/ST(i) F <del>I</del> MULP mem/ST(i) F <del>I</del> DIV <del>R</del> P mem/ST(i) FSQRT FABS FRNDINT	F <del>I</del> COMP F <del>I</del> UCOMP 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  $4 \times 128$ -bit registers
  - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
  - $2 \times 64$ -bit double precision
  - $4 \times 32$ -bit double precision
  - Instructions operate on them simultaneously
    - Single-Instruction Multiple-Data (SIMD)

# 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

## ■ 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:

```

1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double* C)
3. {
4.     for ( int i = 0; i < n; i+=4 )
5.         for ( int j = 0; j < n; j++ ) {
6.             _m256d c0 = _mm256_load_pd(C+i+j*n); /* c0=C[i][j] */
7.             for( int k = 0; k < n; k++ )
8.                 c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
9.                                     _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.                                                    _mm256_broadcast_sd(B+k+j*n)));
11.             _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.         }
13. }

```

# 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

```

# 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