



# 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:  $7 + 6$

A binary addition diagram for the sum  $7 + 6$ . The numbers are represented as binary fractions:  $7 = 0.111_2$  and  $6 = 0.110_2$ . The sum is  $13 = 1.111_2$ . The diagram shows the addition process column by column from right to left. Blue arrows labeled '(0)' indicate no carry, while blue arrows labeled '(1)' indicate a carry. The carries are labeled '(Carries)' above the columns.

0	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0	1
+	0	0	0	0	0	0	0	0
<hr/>								
0	0	0	0	1	1	1	0	1

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

$$\text{Example: } 7 - 6 = 7 + (-6)$$

+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

# 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



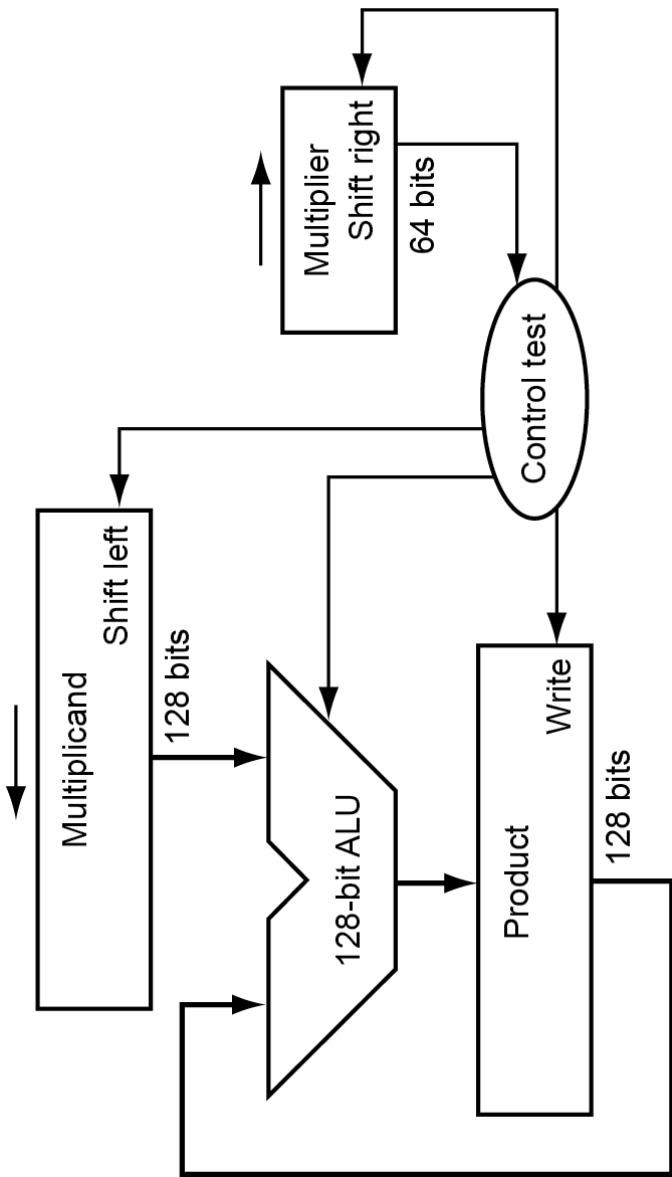
# Multiplication

Start with long-multiplication approach

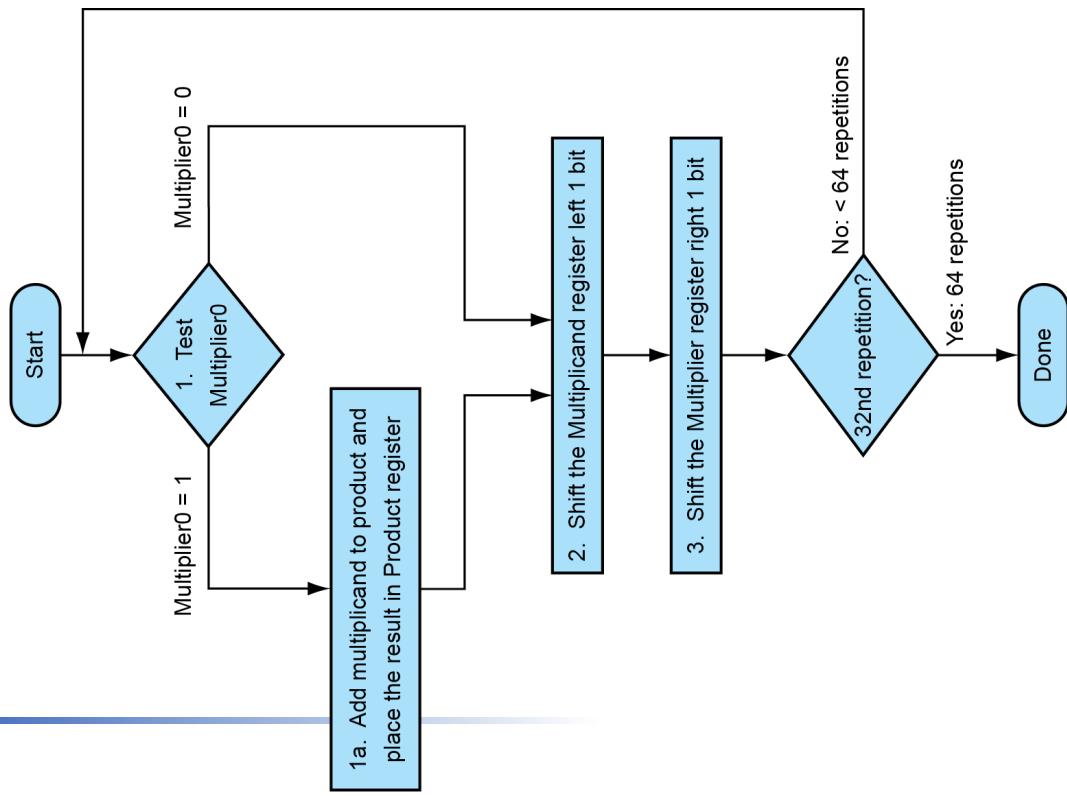
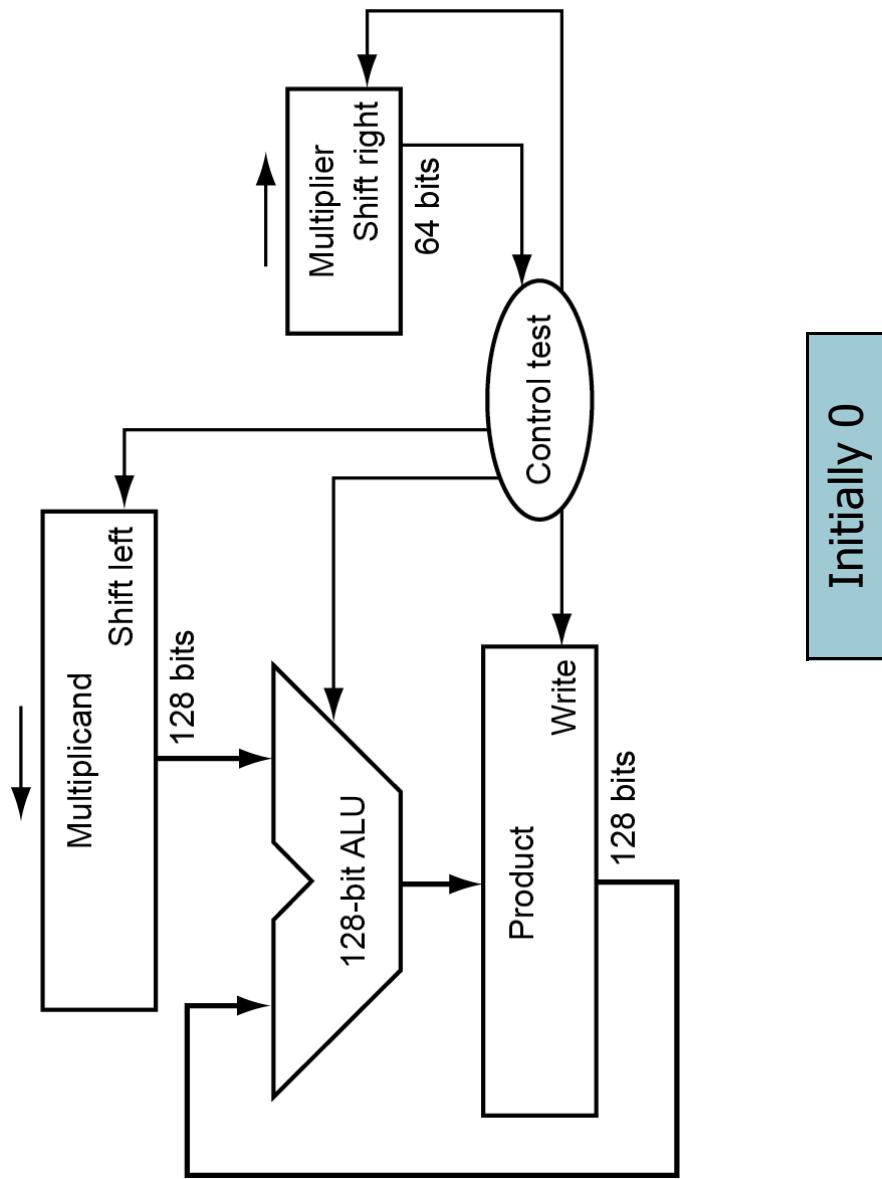
$$\begin{array}{r} \text{multiplicand} & 1000 \\ \times \text{multiplier} & 1001 \\ \hline & 1000 \\ & 0000 \\ & 0000 \\ & 1000 \\ \hline & 100100 \\ & 0 \end{array}$$

product

Length of product is  
the sum of operand  
lengths

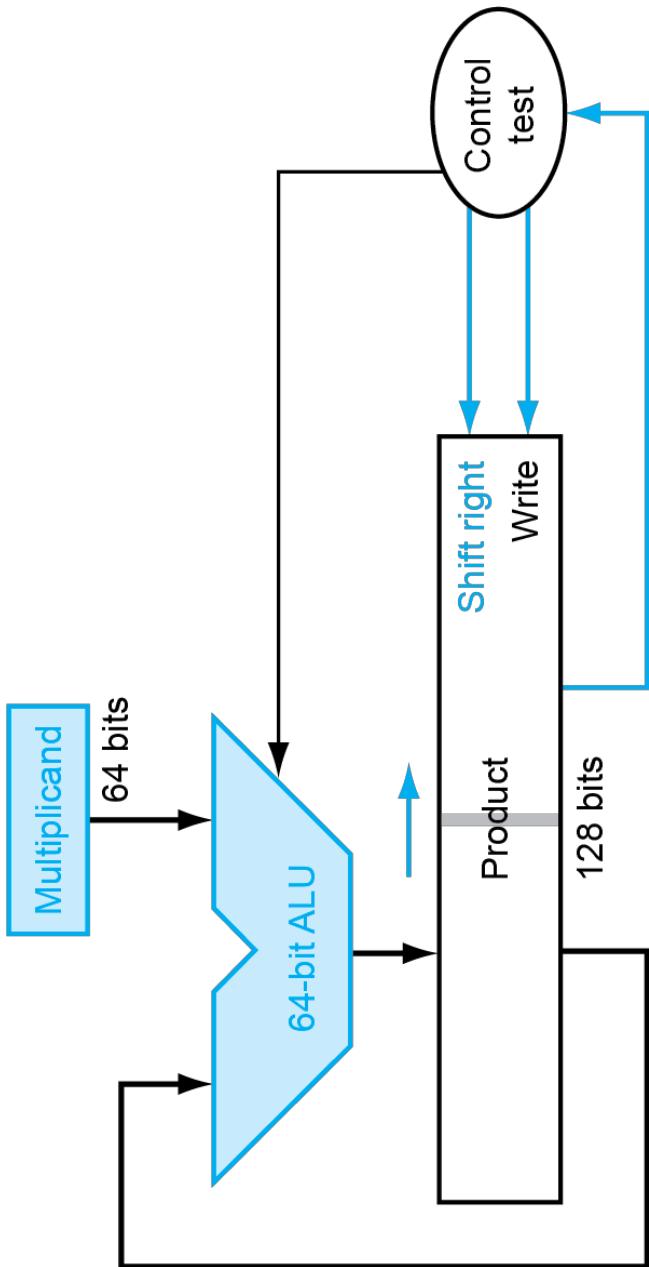


# Multiplication Hardware



# Optimized Multiplier

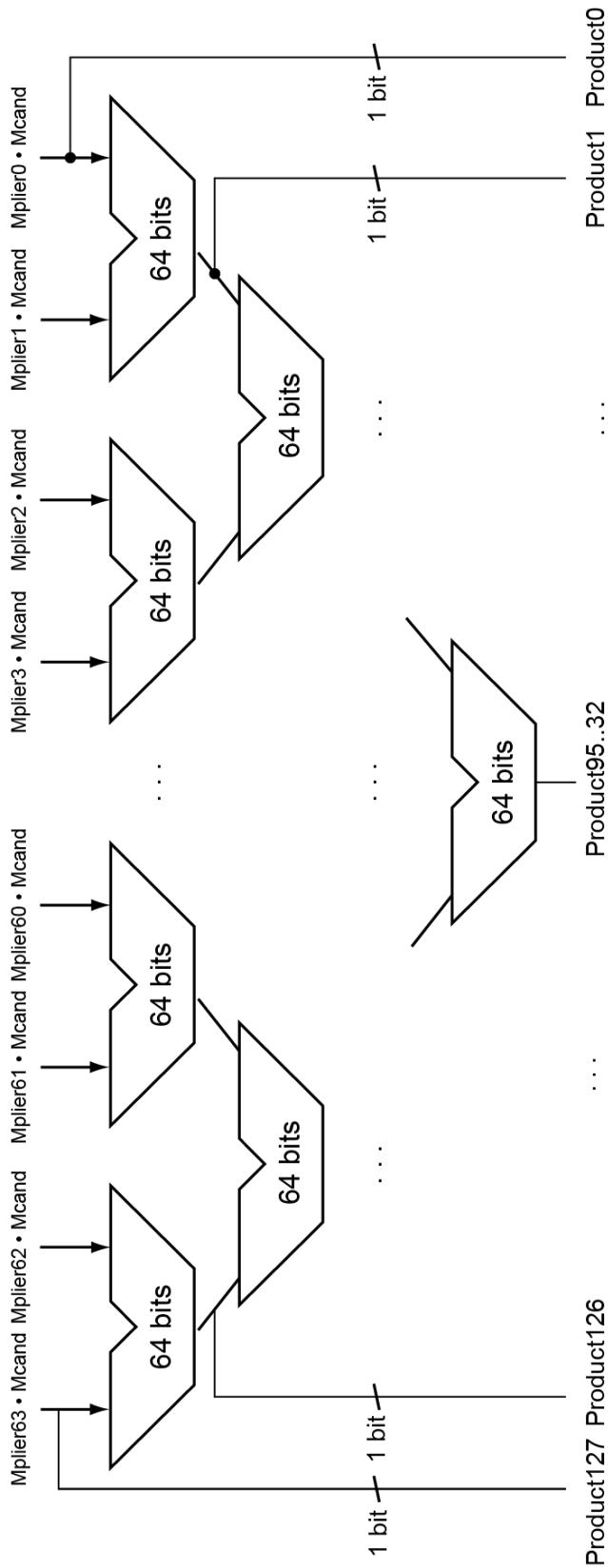
Perform steps in parallel: addshift



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

# RISC-V Multiplication

Four multiply instructions:

**mul: multiply**

Gives the lower 64 bits of the product

**mulh: multiply high**

Gives the upper 64 bits of the product, assuming the  
operands are signed

**mulhu: multiply high unsigned**

Gives the upper 64 bits of the product, assuming the  
operands are unsigned

**mulhsu: multiply high signed/unsigned**

Gives the upper 64 bits of the product, assuming one  
operand is signed and the other unsigned

**Use mulh result to check for 64-bit overflow**

# Division

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

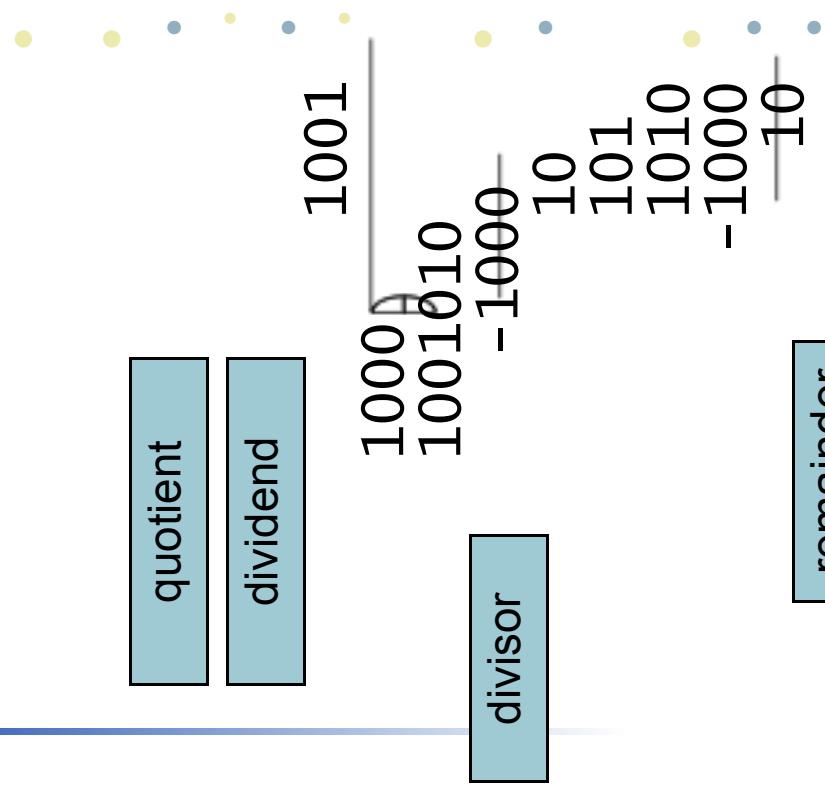
Restoring division

Do the subtract, and if remainder goes  $< 0$ , add divisor back

Signed division

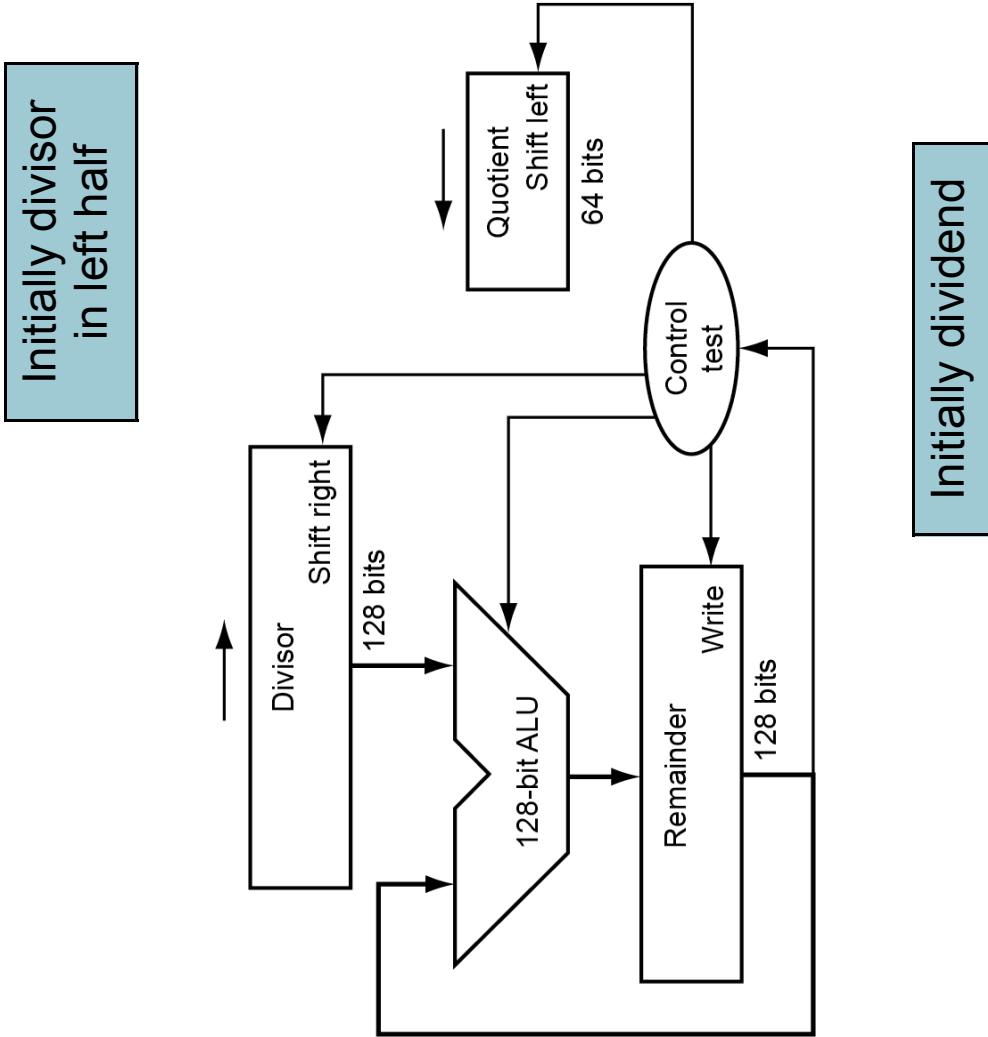
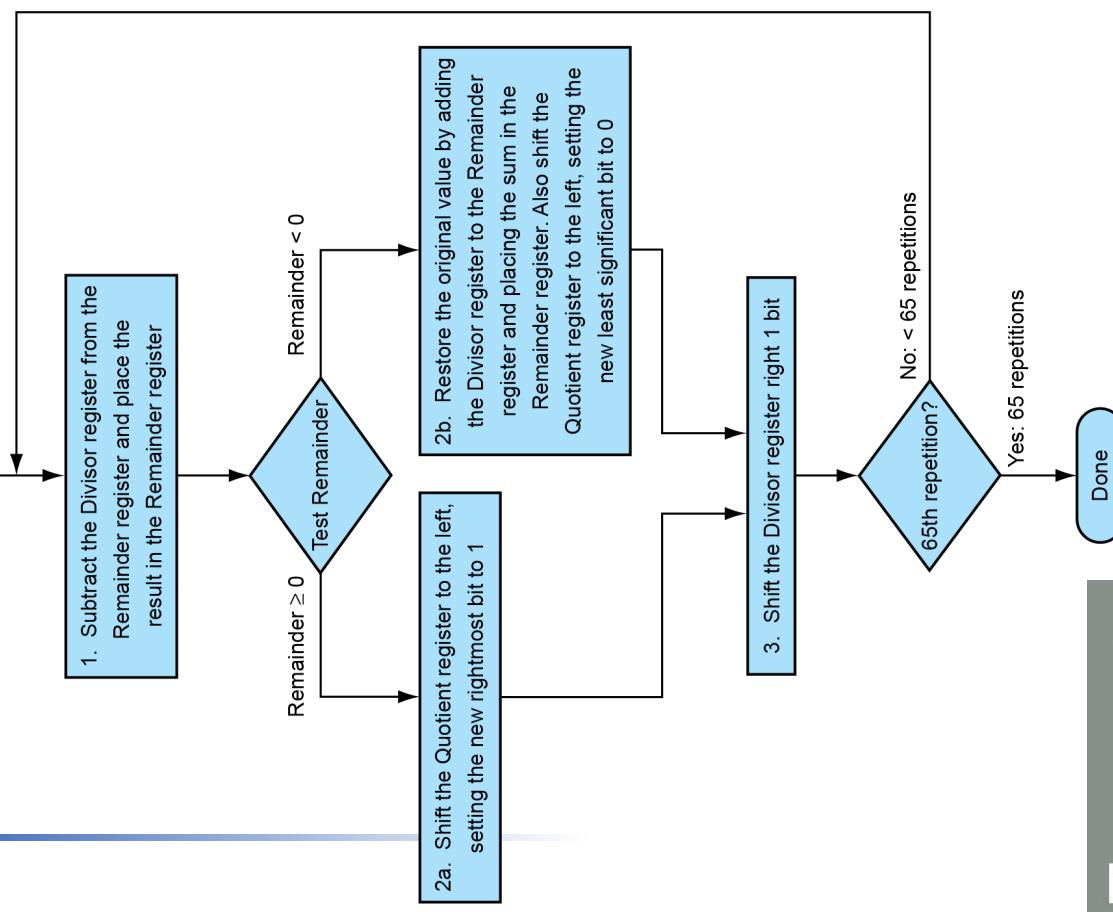
Divide using absolute values

Adjust sign of quotient and remainder as required

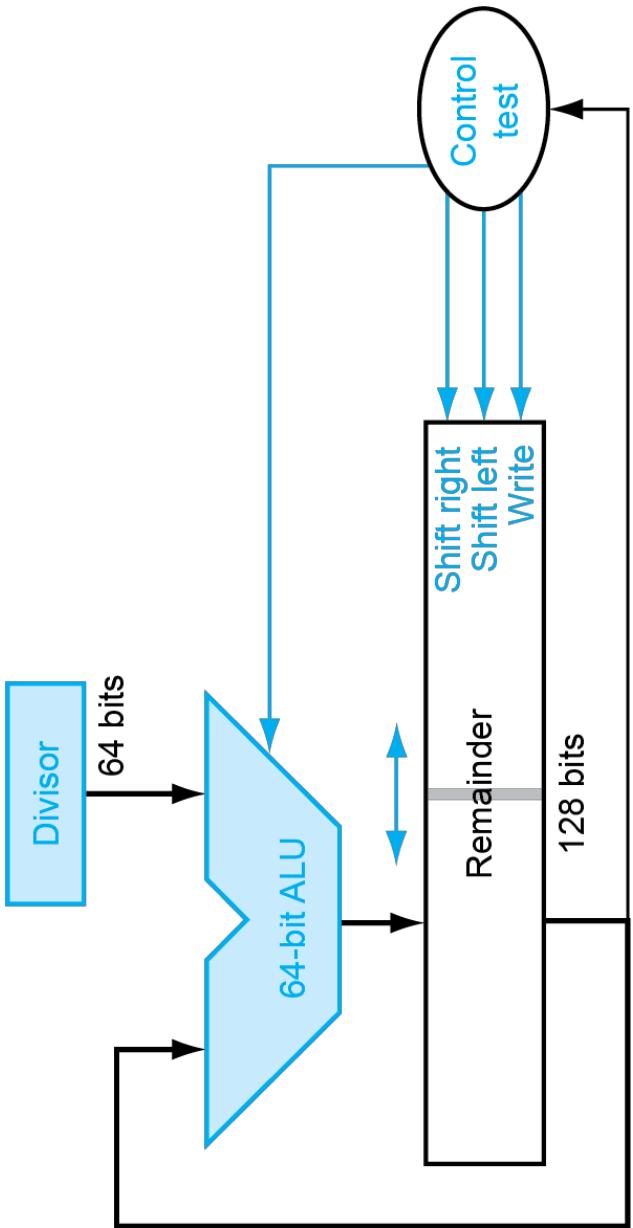


$n$ -bit operands yield  $n$ -bit quotient and remainder

# 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 devision)
  - generate multiple quotient bits per step
  - Still require multiple steps



# RISC-V Division

Four instructions:

div, rem: signed divide, remainder  
divu, remu: unsigned divide, remainder

Overflow and division-by-zero don't produce errors

Just return defined results

Faster for the common case of no error

# Floating Point

Representation for non-integral numbers  
Including very small and very large numbers

Like scientific notation

$$-2.34 \times 10^{56}$$

$$+0.002 \times 10^{-4}$$

$$+987.02 \times 10^9$$

normalized

not normalized

In binary

$$\pm 1.xxxxxxx2 \times 2^{yyyy}$$

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

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

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

S: sign bit (0  $\square$  non-negative, 1  $\square$  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 1111111 reserved

Smallest value

Exponent: 00000001

actual exponent =  $1 - 127 = -126$

Fraction: 000...00  significand = 1.0

$\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$

Largest value

Exponent: 1111110

actual exponent =  $254 - 127 = +127$

Fraction: 111...11  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

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: 111111110

actual exponent =  $2046 - 1023 = +1023$

Fraction: 111...11  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  $-0.75$

$$-0.75 = (-1)1 \times 1.12 \times 2^{-1}$$

$S = 1$

Fraction =  $1000\dots002$

Exponent =  $-1 + \text{Bias}$

Single:  $-1 + 127 = 126 = 01111110_2$

Double:  $-1 + 1023 = 1022 = 0111111102$

Single:  $10111101000\dots00$

Double:  $101111101000\dots00$



# Floating-Point Example

---

What number is represented by the single-precision float  
**1 1000000101000...00**

$$S = 1$$

$$\text{Fraction} = 01000\dots002$$

$$\text{Exponent} = \color{green}{100000012} = 129$$

$$\begin{aligned}x &= (-1)^1 \times (1 + 012) \times 2(129 - 127) \\&= (-1) \times 1.25 \times 22 \\&= -5.0\end{aligned}$$



# 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

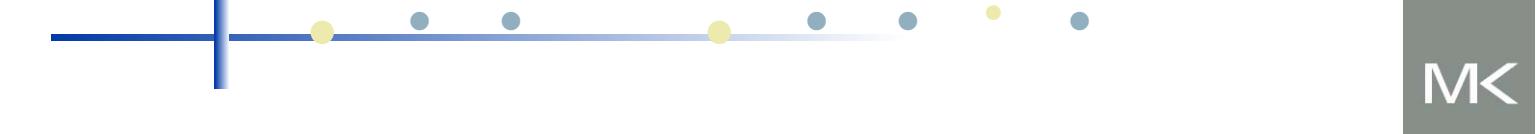
Exponent = 111...1, Fraction = 000...0  
±Infinity

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

Exponent = 111...1, Fraction  $\neq$  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

$$1.0015 \times 10^2$$

4. Round and renormalize if necessary

$$1.002 \times 10^2$$



# Floating-Point Addition

---

Now consider a 4-digit binary example

$$1.0002 \times 2^{-1} + -1.1102 \times 2^{-2} (0.5 + -0.4375)$$

1. Align binary points

Shift number with smaller exponent

$$1.0002 \times 2^{-1} + -0.1112 \times 2^{-1}$$

2. Add significands

$$1.0002 \times 2^{-1} + -0.1112 \times 2^{-1} = 0.0012 \times 2^{-1}$$

3. Normalize result & check for over/underflow

$$1.0002 \times 2^{-4}, \text{with no over/underflow}$$

4. Round and renormalize if necessary

$$1.0002 \times 2^{-4} (\text{no change}) = 0.0625$$



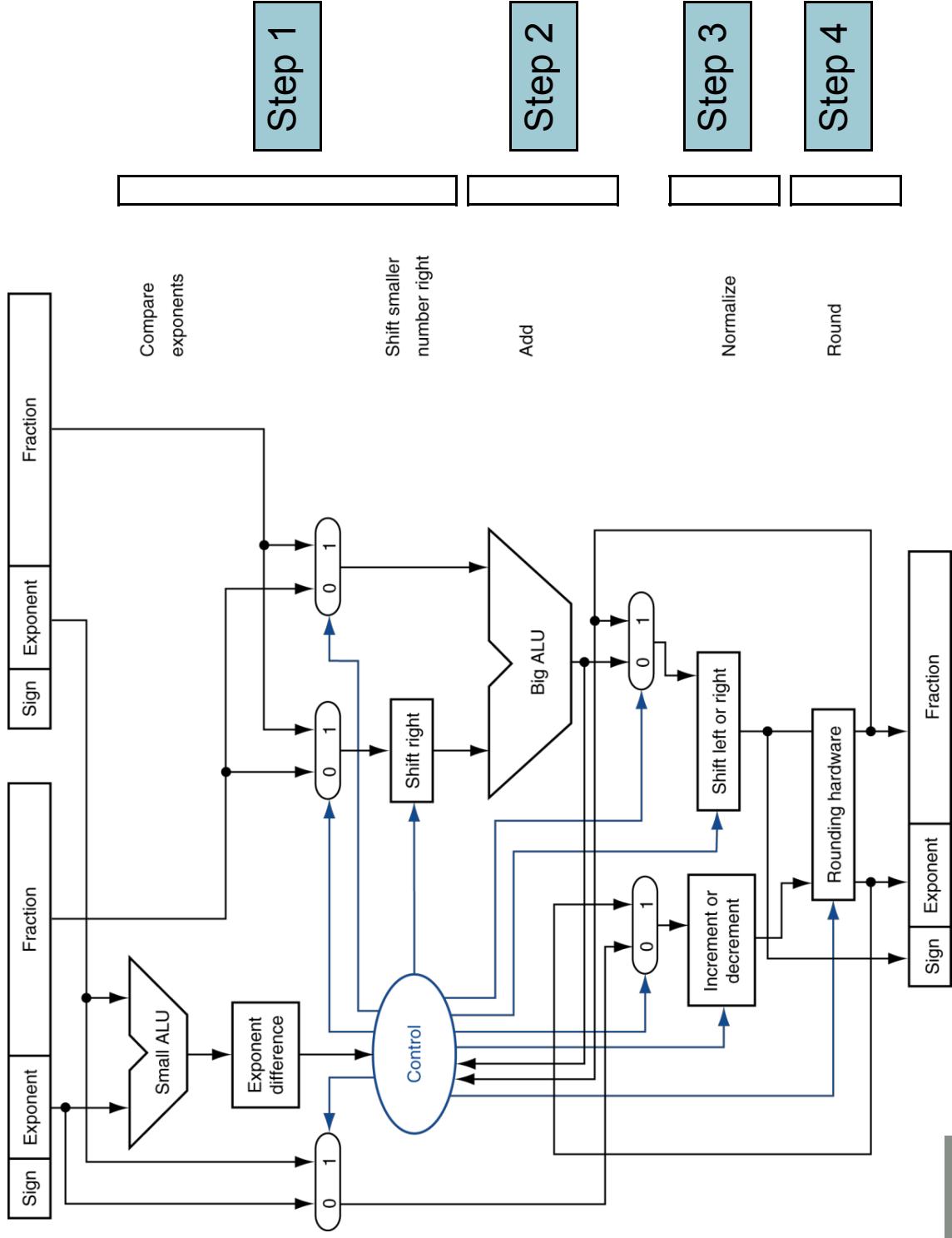
# FP Adder Hardware

---

- 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



# Floating-Point Multiplication

---

Consider a 4-digit decimal example

$$1.110 \times 1010 \times 9.200 \times 10^{-5}$$

## 1. Add exponents

For biased exponents, subtract bias from sum

$$\text{New exponent} = 10 + -5 = 5$$

## 2. Multiply significands

$$1.110 \times 9.200 = 10.212 \quad \square \quad 10.212 \times 10^5$$

## 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

$$1.0002 \times 2^{-1} \times -1.1102 \times 2^{-2} (0.5 \times -0.4375)$$

## 1. Add exponents

Unbiased:  $-1 + -2 = -3$

Biased:  $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$

## 2. Multiply significands

$$1.0002 \times 1.1102 = 1.1102 \quad \square \quad 1.1102 \times 2^{-3}$$

## 3. Normalize result & check for over/underflow

$1.1102 \times 2^{-3}$  (no change) with no over/underflow

## 4. Round and renormalize if necessary

$$1.1102 \times 2^{-3}$$
 (no change)

## 5. Determine sign: +ve $\times$ -ve $\square$ -ve

$$-1.1102 \times 2^{-3} = -0.21875$$

# 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  $\square$  integer conversion

Operations usually takes several cycles  
Can be pipelined

# FP Instructions in RISC-V

- Separate FP registers: f0, ..., f31
  - double-precision
  - single-precision values stored in the lower 32 bits
- 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
  - f1w, f1d
  - fsw, fsd



# FP Instructions in RISC-V

Single-precision arithmetic  
`fadd.s`, `fsub.s`, `fmu1.s`, `fdiv.s`, `fsqrt.s`

e.g., `fadds.s f2, f4, f6`

Double-precision arithmetic  
`fadd.d`, `fsub.d`, `fmu1.d`, `fdiv.d`, `fsqrt.d`

e.g., `fadd.d f2, f4, f6`

Single- and double-precision comparison  
`feq.s`, `fne.s`, `fle.s`,  
`feq.d`, `fne.d`, `fle.d`

Result is 0 or 1 in integer destination register  
Use `bne`, `bne` to branch on comparison result

Branch on FP condition code true or false  
`B.cond`

# FP Example: °F to °C

C code:

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

fahr in f10, result in f10, literals in global memory space

Compiled RISC-V code:

```
f2c:  
    f1w    f0, const5(x3)    // f0 = 5.0f  
    f1w    f1, const9(x3)    // f1 = 9.0f  
    fdiv.s f0, f0, f1      // f0 = 5.0f / 9.0f  
    f1w    f1, const32(x3)  // f1 = 32.0f  
    fsub.s f10, f10, f1    // f10 = fahr - 32.0  
    fmul.s f10, f0, f10   // f10 = (5.0f/9.0f) * (fahr-32.0f)  
    jalr  x0, 0(x1)        // return
```

# FP Example: Array Multiplication

$$C = C + A \times B$$

All  $32 \times 32$  matrices, 64-bit double-precision elements

C code:

```
void mm (double c[],  
         double a[], double b[]) {  
    size_t 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)  
                c[i][j] = c[i][j]  
                          + a[i][k] * b[k][j];  
}
```

Addresses of  $c$ ,  $a$ ,  $b$  in  $x10$ ,  $x11$ ,  $x12$ , and  
 $i$ ,  $j$ ,  $k$  in  $x5$ ,  $x6$ ,  $x7$

# FP Example: Array Multiplication

- RISC-V code:

```
mm: . . .
    l1i    x28,32          // x28 = 32 (row size/loop end)
    l1i    x5,0             // i = 0; initialize 1st for loop
L1:   l1i    x6,0             // j = 0; initialize 2nd for loop
L2:   l1i    x7,0             // k = 0; initialize 3rd for loop
s11i  x30,x5,5           // x30 = i * 2**5 (size of row of c)
add   x30,x30,x6          // x30 = i * size(row) + j
s11i  x30,x30,3           // x30 = byte offset of [i][j]
add   x30,x10,x30          // x30 = byte address of c[i][j]
f1d   f0,0(x30)           // f0 = c[i][j]
L3:   s11i  x29,x7,5           // x29 = k * 2**5 (size of row of b)
add   x29,x29,x6          // x29 = k * size(row) + j
s11i  x29,x29,3             // x29 = byte offset of [k][j]
add   x29,x12,x29          // x29 = byte address of b[k][j]
f1d   f1,0(x29)           // f1 = b[k][j]
```

# FP Example: Array Multiplication

```
...
    s11i    x29,x5,5      // x29 = i * 2**5 (size of row of a)
    add     x29,x29,x7    // x29 = i * size(row) + k
    s11i    x29,x29,3      // x29 = byte offset of [i][k]
    add     x29,x11,x29   // x29 = byte address of a[i][k]
    f1d    f2,0(x29)      // f2 = a[i][k]
    fmul.d f1, f2, f1    // f1 = a[i][k] * b[k][j]
    fadd.d f0, f0, f1    // f0 = c[i][j] + a[i][k] * b[i][j]
    addi   x7,x7,1        // k = k + 1
    b1tu   x7,x28,L3     // if (k < 32) go to L3
    fsd    f0,0(x30)      // c[i][j] = f0
    addi   x6,x6,1        // j = j + 1
    b1tu   x6,x28,L2     // if (j < 32) go to L2
    addi   x5,x5,1        // i = i + 1
    b1tu   x5,x28,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 FPU 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)	F <sub>I</sub> ADD P mem/ST(i)	F <sub>I</sub> COMP	FPATAN
FISTP mem/ST(i)	F <sub>I</sub> SUB RP mem/ST(i)	F <sub>I</sub> UCOMP	F2XMI
FLDPI	F <sub>I</sub> MUL P mem/ST(i)	FSTSW AX/mem	FCOS
FLD1	F <sub>I</sub> DIV RP mem/ST(i)		FPTAN
FLDZ	FSQRT		FPREM
	FABS		FPSIN
	FRNDINT		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  $4 \times 1$  28-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



# 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 ci_j = C[i+j*n]; /* ci_j = C[i][j] */
7.         for (int k = 0; k < n; k++)
8.             ci_j += A[i+k*n] * B[k+j*n]; /* ci_j += A[i][k] * B[k][j] */
9.         C[i+j*n] = ci_j; /* Ci_j = ci_j */
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 = %r9 + %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, %r11           # register %r11 = %r11 + 1
12. vmovepd %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] */ /
   _mm256_mull_pd(_mm256_load_pd(A+i+k*n) ,
   _mm256_broadcast_sd(B+k+j*n)) );
9.             _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
10.         }
11.     }
12. }
```

# 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.,  $-5 / 4$

$\textcolor{teal}{1}1110112 \gg 2 = \textcolor{teal}{1}111102 = -2$

Rounds toward  $-\infty$

c.f.  $\textcolor{teal}{1}1110112 \gg 2 = \textcolor{teal}{001}11102 = +62$

# 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	1.50E+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 0.00002¢!” ☺

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

