

Chapter 3

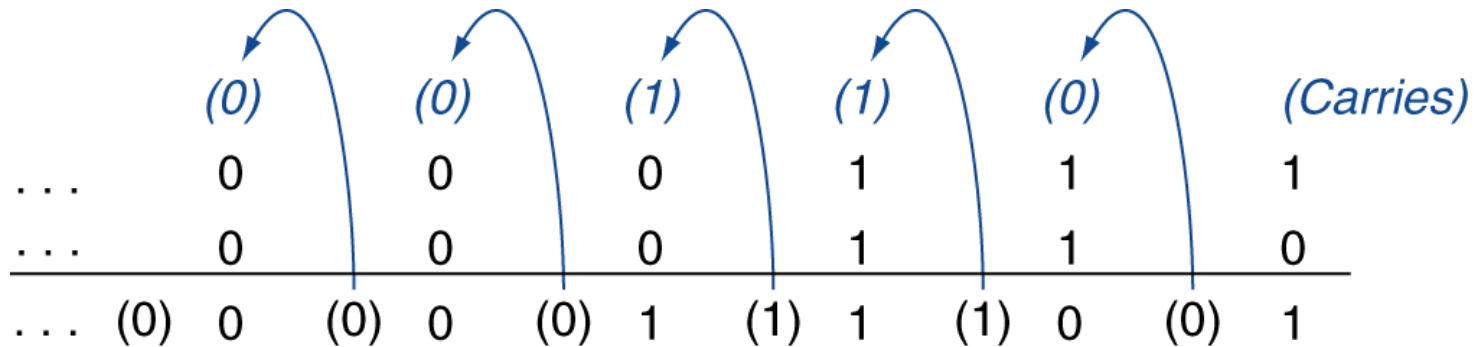
Arithmetic for Computers

Arithmetic for Computers

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

Integer Addition

■ Example: $7 + 6$



■ Overflow if result out of range

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

Integer Subtraction

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

+7:	0000 0000 ... 0000 0111
-6:	1111 1111 ... 1111 1010
<hr/>	
+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

Overflow

- Overflow means,
 - Result is too large for a finite computer word
 - Adding **two n-bit** numbers does not yield an n-bit number
 - e.g. add two **unsigned** 3-bit numbers,
 - 111
 - + 001
 - -----
 - 1000

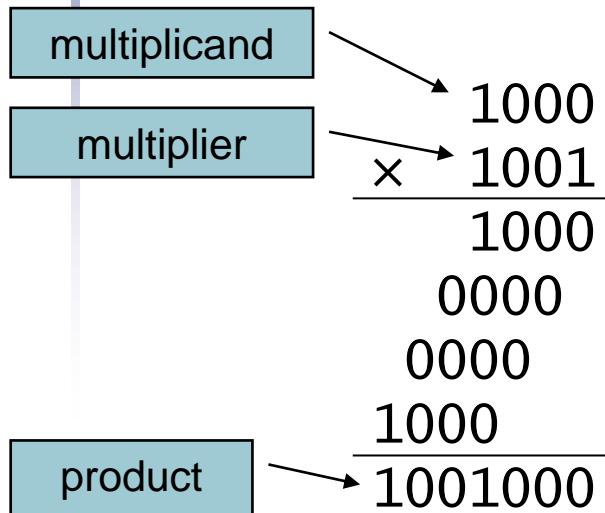
Operation	Operand A	Operand B	Overflow if result
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A - B	≥ 0	< 0	< 0
A - B	< 0	≥ 0	≥ 0

Arithmetic for Multimedia

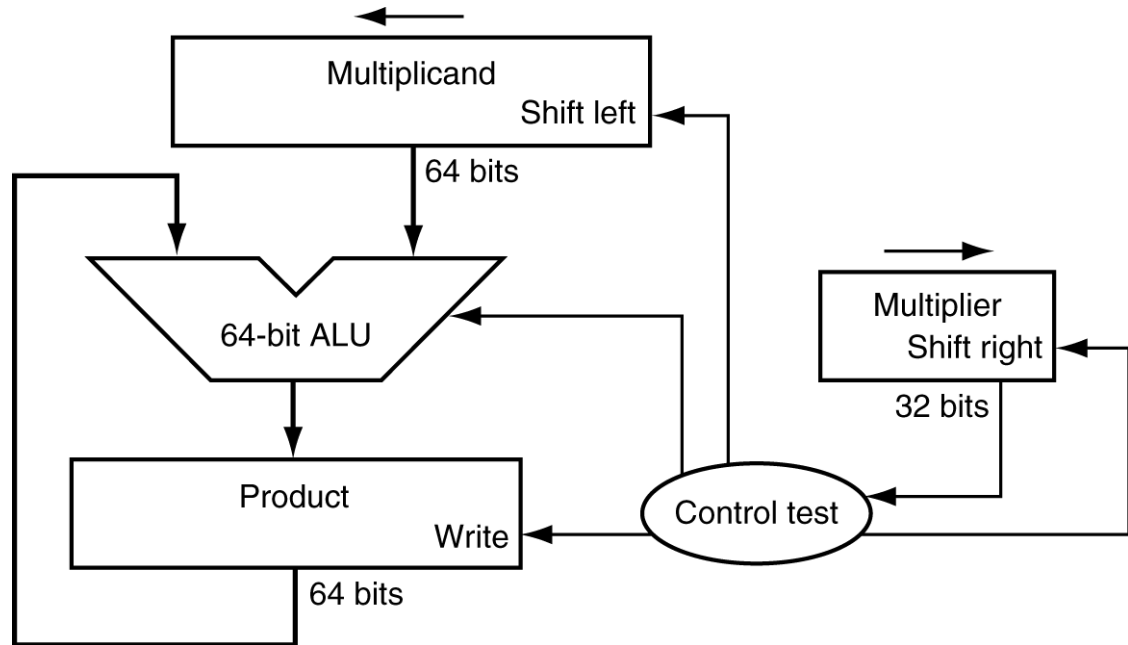
- Graphics and media processing operates on vectors of 8-bit and 16-bit data
 - Use 64-bit adder, with partitioned carry chain
 - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
 - SIMD (single-instruction, multiple-data)
- Saturating operations
 - On overflow, result is largest representable value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

Multiplication

- Start with long-multiplication approach

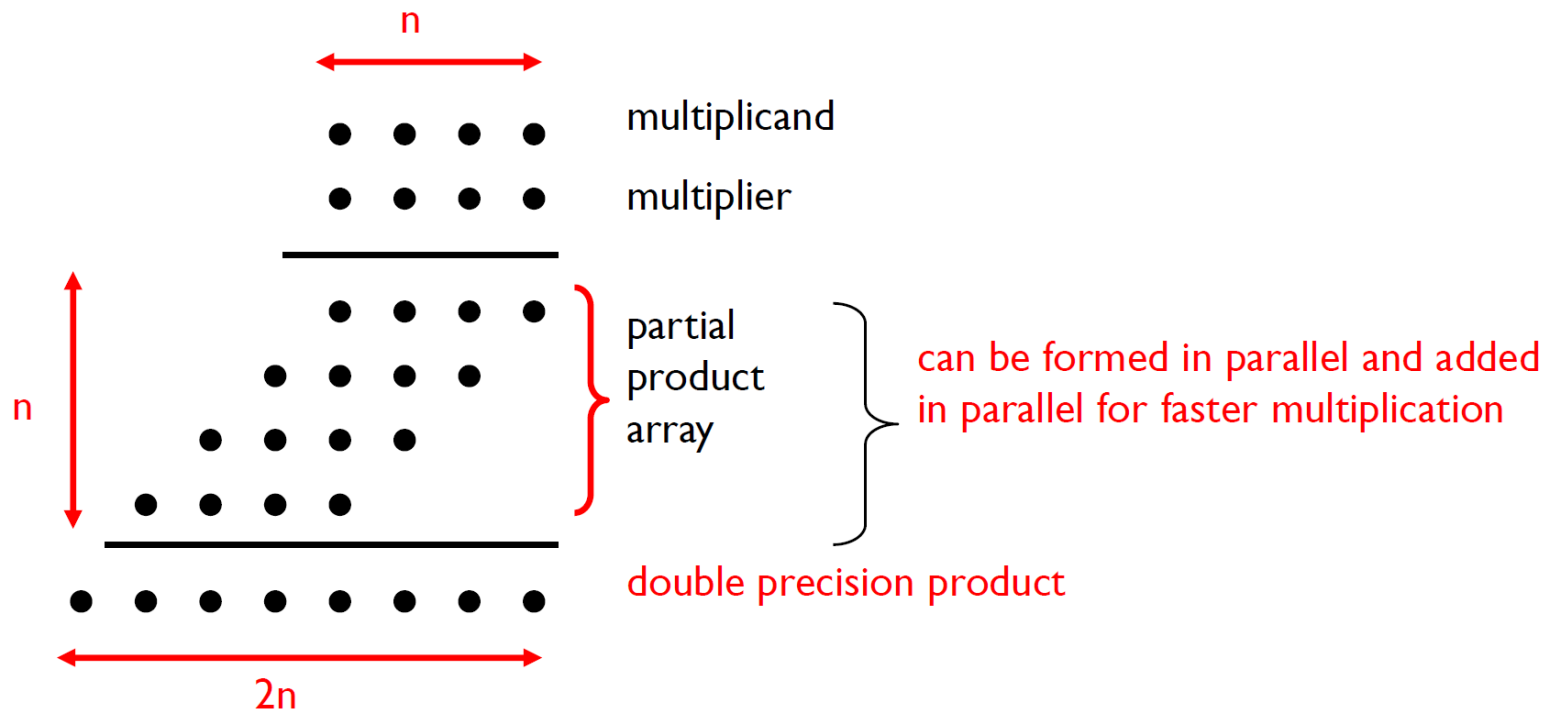


Length of product is the sum of operand lengths



Multiplication (2)

- Binary multiplication is,
 - Just a *bunch* of shifts and adds



Integer Multiplication

- Pencil and paper binary multiplication

$$\begin{array}{r} 1000 \quad (\text{multiplicand}) \\ \times 1001 \quad (\text{multiplier}) \\ \hline 1000 \\ 00000 \\ 000000 \\ +1000000 \\ \hline 1001000 \quad (\text{product}) \end{array}$$

(partial products)

- Key elements:
 - Examine **multiplier** bits from right to left
 - Shift **multiplicand** left one position each step
 - Simplification: each step, **add multiplicand** to running product total, but only if **multiplier bit == 1**

Integer Multiplication (2)

- Initialize product register to 0

1000 (multiplicand)

1001 (multiplier)

00000000 (running product)

Integer Multiplication (3)

- Multiplier bit == 1:
 - Add multiplicand to product

1000 (multiplicand)

1001 (multiplier)

00000000

+1000

00001000

(new running product)

Integer Multiplication (4)

- Shift multiplicand left



10000

(multiplicand)

1001

(multiplier)

00001000

(running product)

Integer Multiplication (5)

- Multiplier bit == 0:
 - Do nothing

10000 (multiplicand)

1001 (multiplier)

00001000 (running product)

Integer Multiplication (6)

- Shift multiplicand left

→ 100000 (multiplicand)
 1001 (multiplier)

 00001000 (running product)

Integer Multiplication (7)

- Multiplier bit == 0:
 - Do nothing

100000 (multiplicand)

1001 (multiplier)

00001000 (running product)

Integer Multiplication (8)

- Shift multiplicand left

→ 1000000 (multiplicand)
1001 (multiplier)

00001000 (running product)

Integer Multiplication (9)

- Multiplier bit == 1:
 - Add multiplicand to product (100 0000)

1000000 (multiplicand)

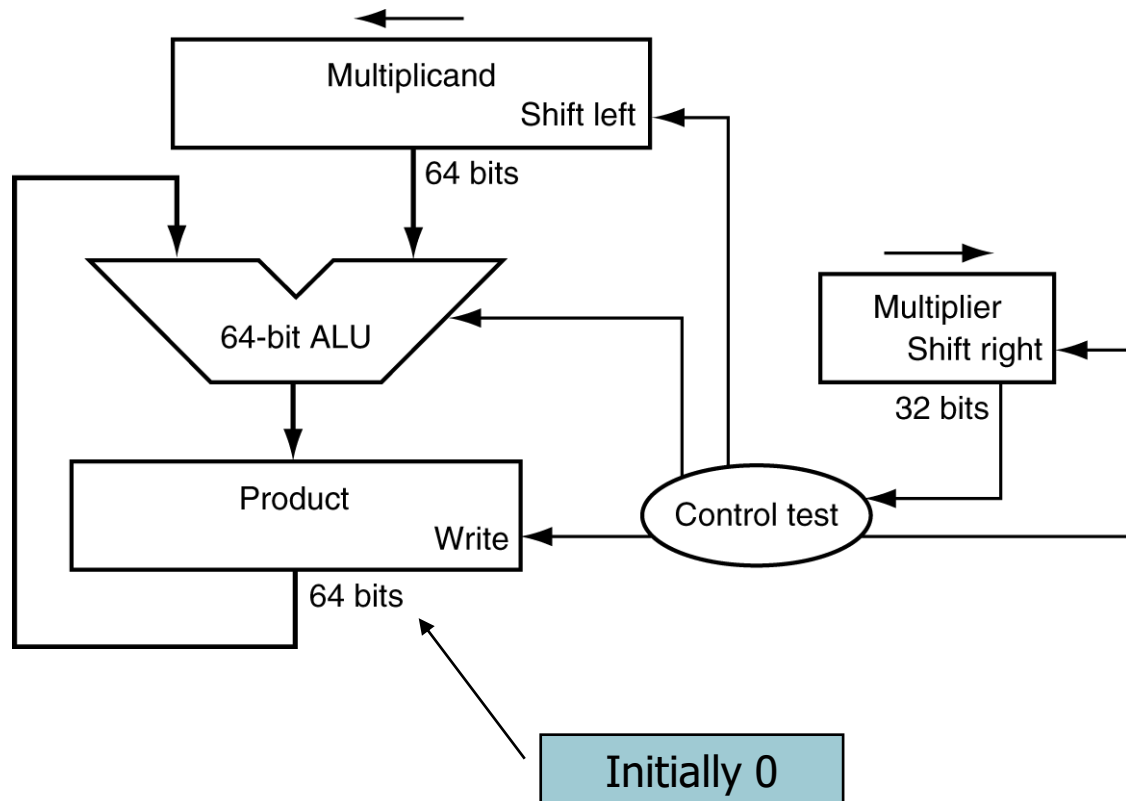
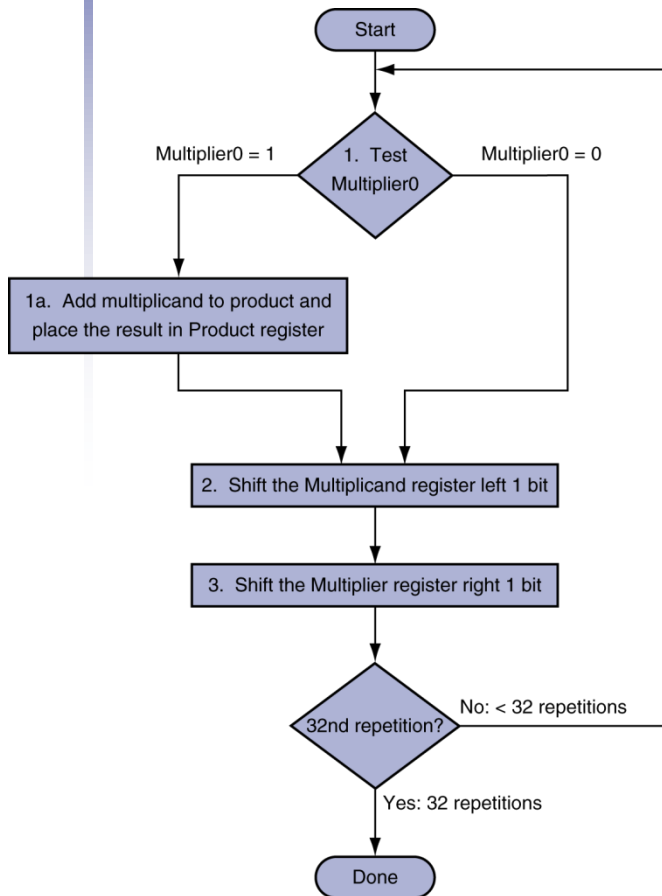
1001 (multiplier)

00001000

+1000000

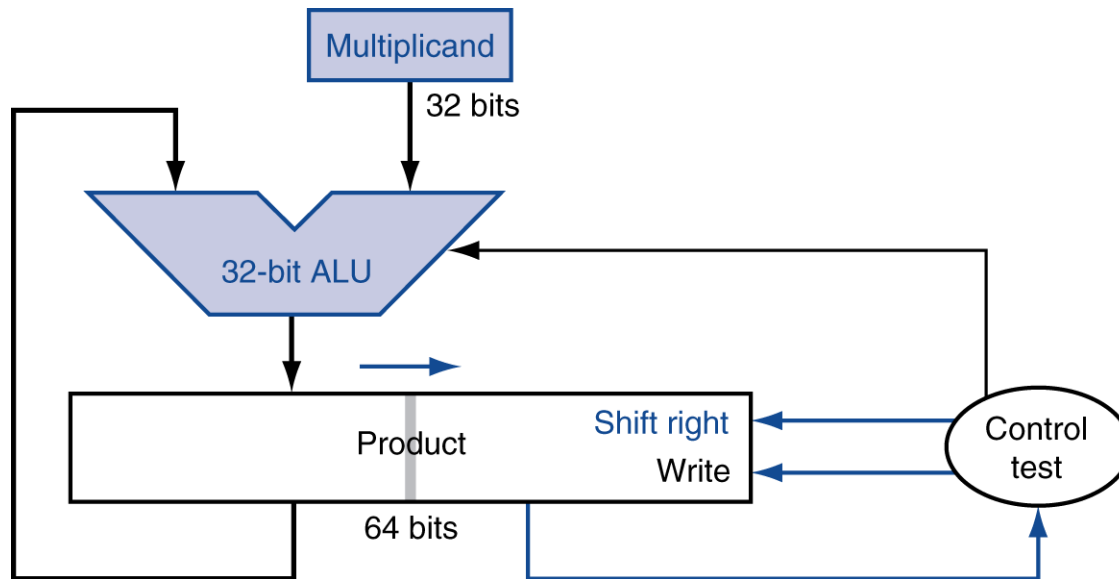
01001000 (product)

Multiplication Hardware



Optimized Multiplier

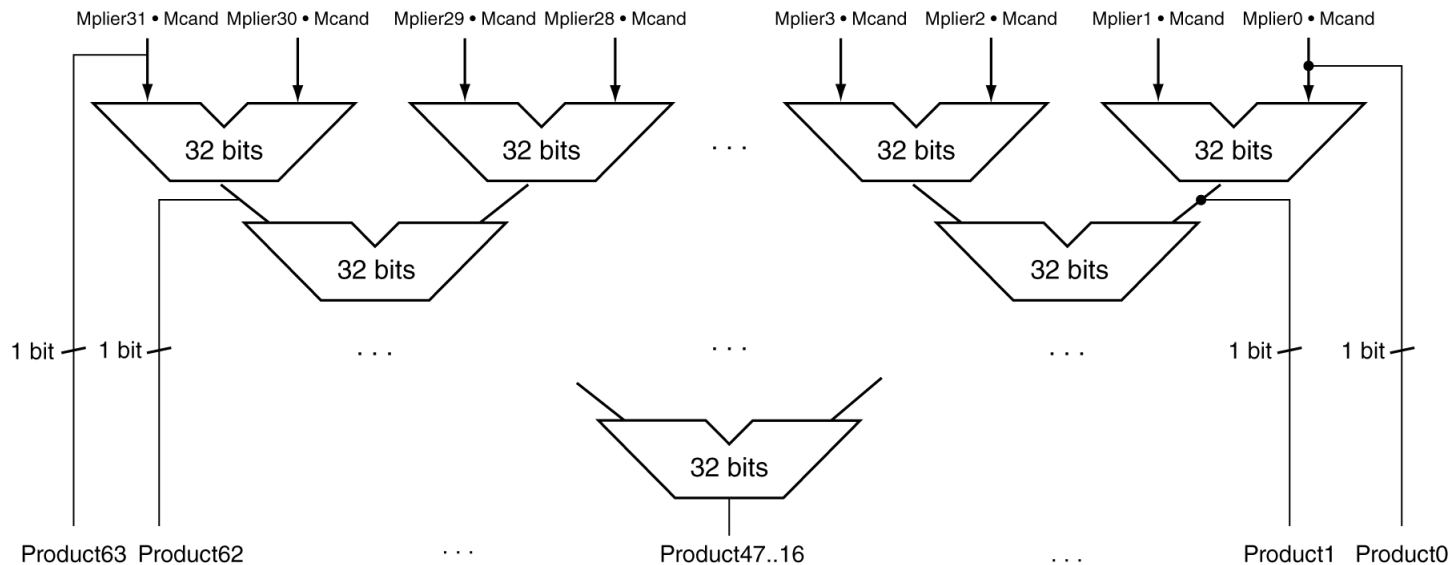
- Perform steps in parallel: add/shift



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

Faster Multiplier

- Uses multiple adders
 - Cost/performance tradeoff

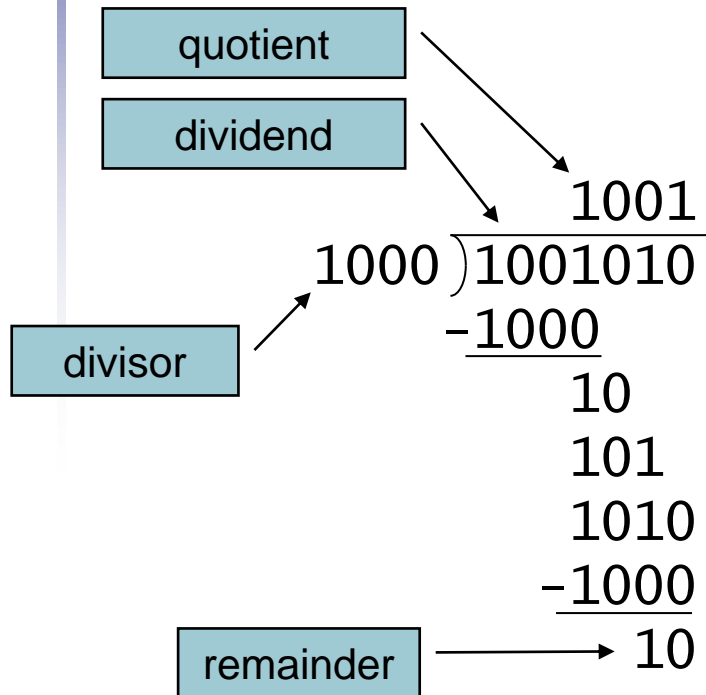


- Can be pipelined
 - Several multiplication performed in parallel

LEGv8 Multiplication

- Three multiply instructions:
 - MUL: multiply
 - Gives the lower 64 bits of the product
 - SMULH: signed multiply high
 - Gives the upper 64 bits of the product, assuming the operands are signed
 - UMULH: unsigned multiply high
 - Gives the lower 64 bits of the product, assuming the operands are unsigned

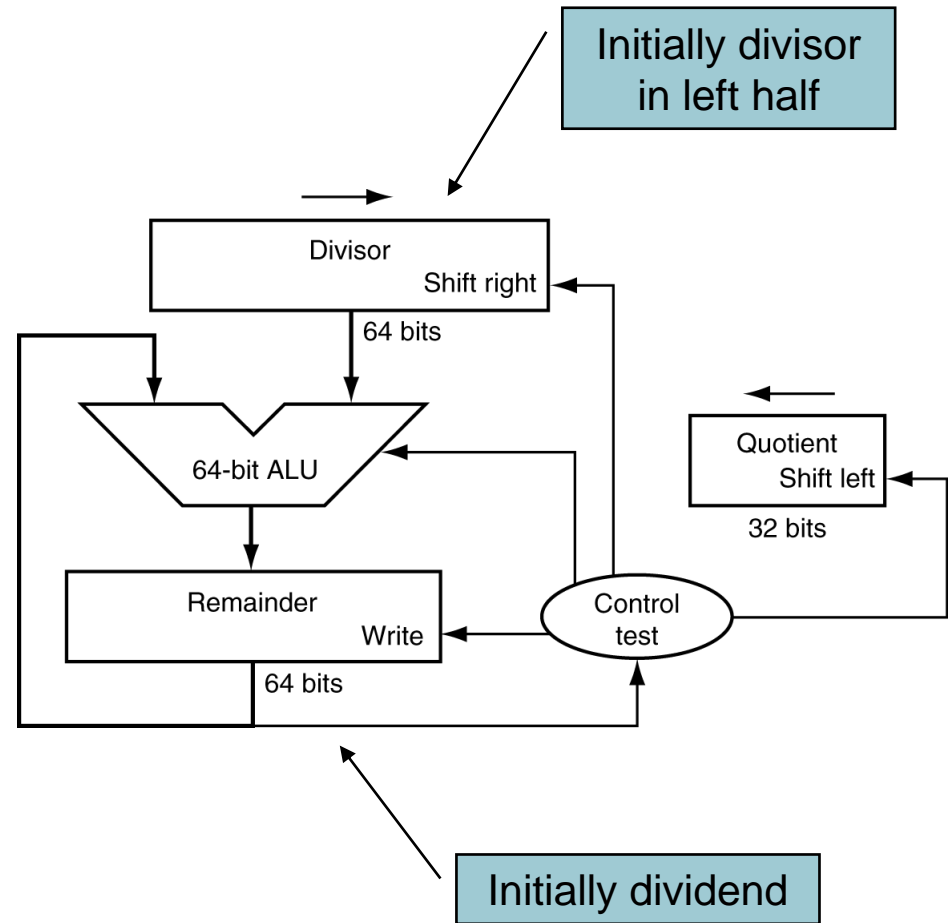
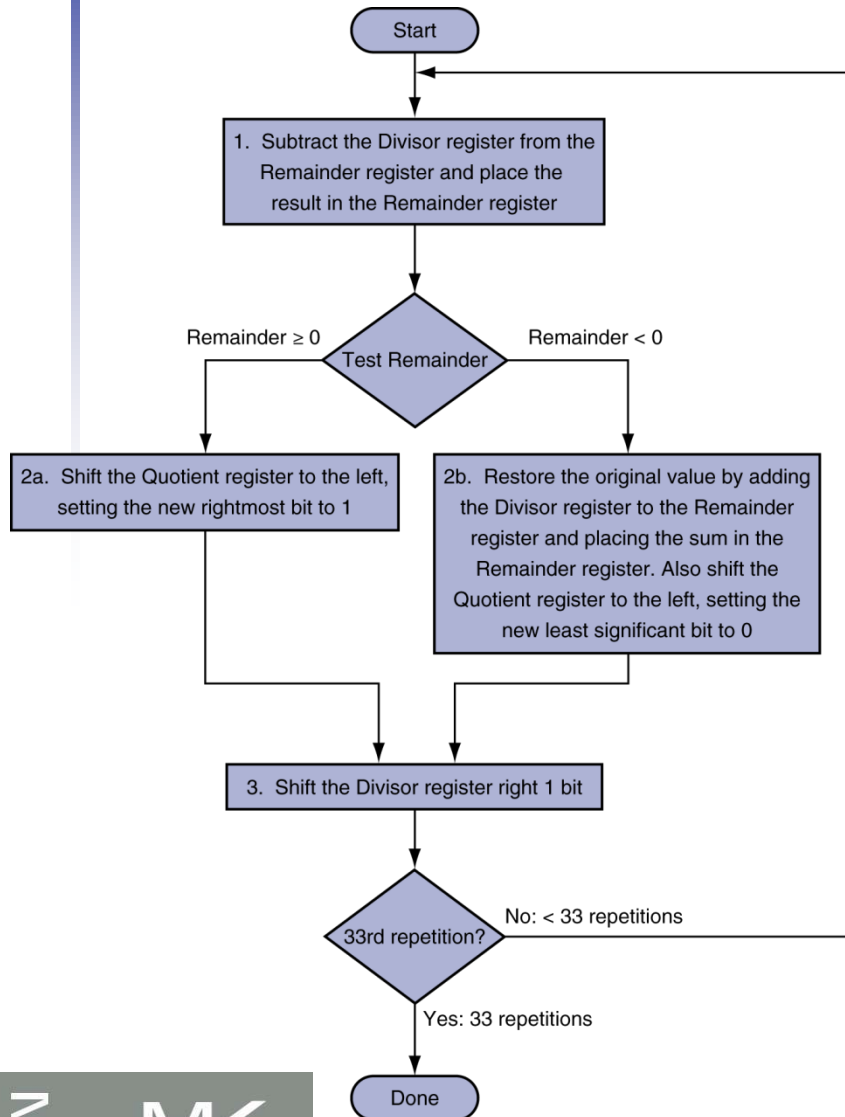
Division



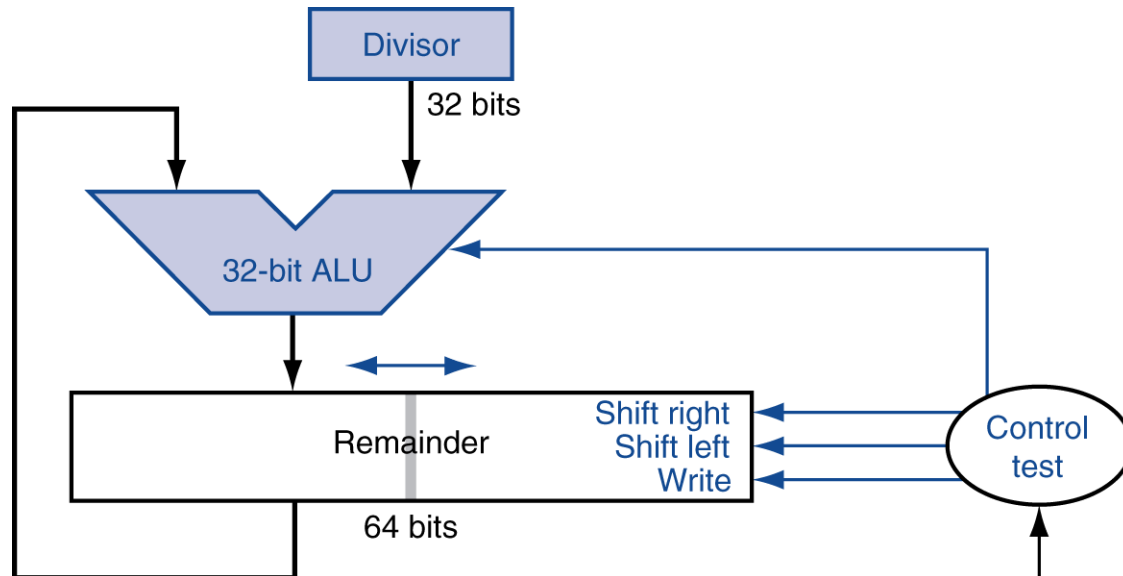
n-bit operands yield *n*-bit quotient and remainder

- 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



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

LEGv8 Division

- Two instructions:
 - SDIV (signed)
 - UDIV (unsigned)
- Both instructions ignore overflow and division-by-zero

Floating Point

- Representation for non-integral 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
 - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- 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 ≈ 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 ≈ 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: $101111111111101000\dots00$

Floating-Point Example

- What number is represented by the single-precision float

11000000101000...00

- $S = 1$
 - Fraction = $01000...00_2$
 - Exponent = $10000001_2 = 129$
- $$\begin{aligned}x &= (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)} \\&= (-1) \times 1.25 \times 2^2 \\&= -5.0\end{aligned}$$

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
 - 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
 - 1.0015×10^2
- 4. Round and renormalize if necessary
 - 1.002×10^2

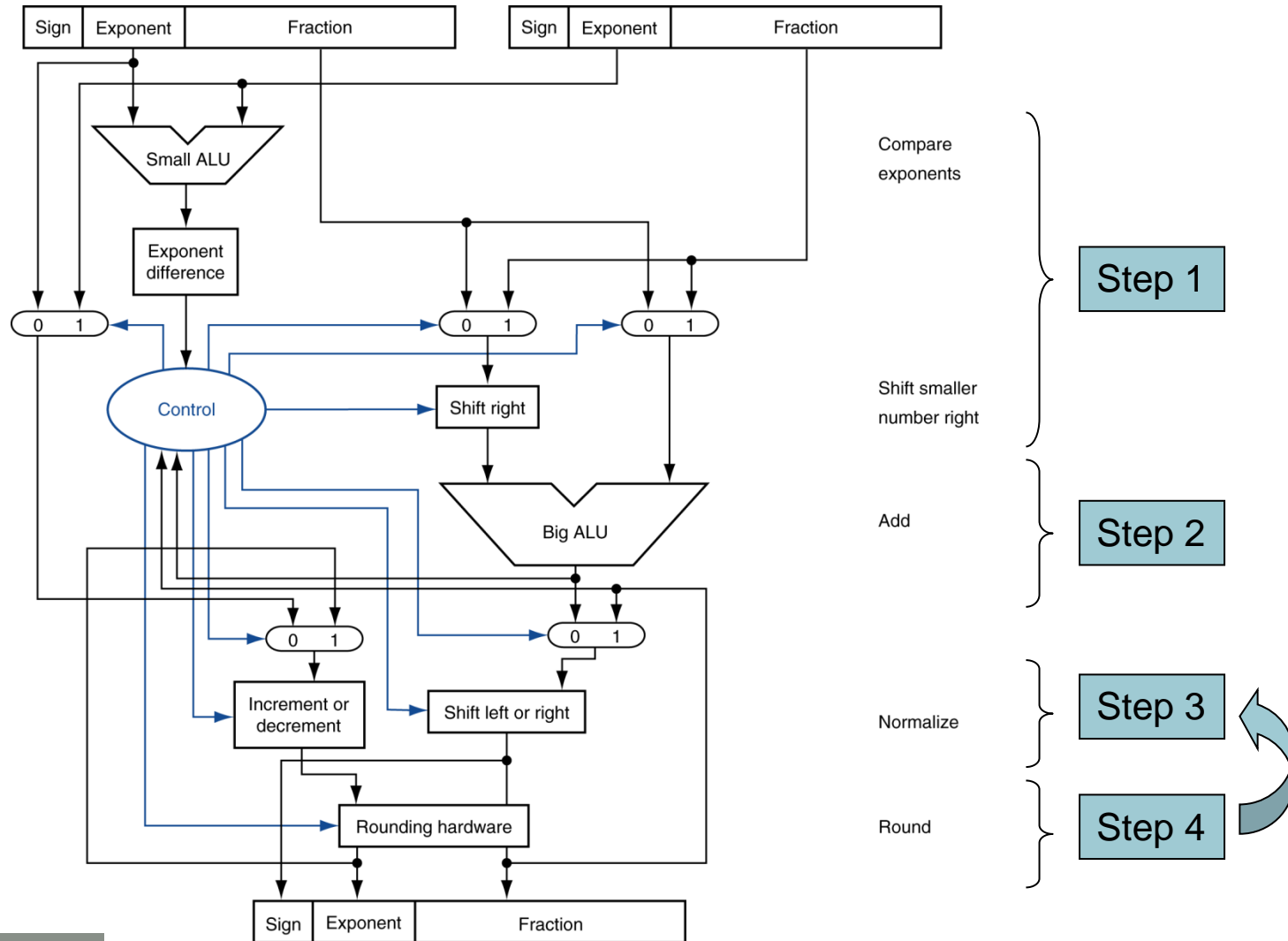
Floating-Point Addition

- 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



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
 - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- 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 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 LEGv8

- Separate FP registers
 - 32 single-precision: S0, ..., S31
 - 32 double-precision: DS0, ..., D31
 - S_n stored in the lower 32 bits of D_n
- 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
 - LDURS, LDURD
 - STURS, STURD

FP Instructions in LEGv8

- Single-precision arithmetic
 - FADDS, FSUBS, FMULS, FDIVS
 - e.g., FADDS S2, S4, S6
- Double-precision arithmetic
 - FADDD, FSUBD, FMULD, FDIVD
 - e.g., FADDD D2, D4, D6
- Single- and double-precision comparison
 - FCMPS, FCMPPD
 - Sets or clears FP condition-code bits
- 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 S12, result in S0, literals in global memory space

- Compiled LEGv8 code:

f2c:

```
LDURS S16, [X27,const5]    // S16 = 5.0 (5.0 in memory)  
LDURS S18, [X27,const9]    // S18 = 9.0 (9.0 in memory)  
FDIVS S16, S16, S18        // S16 = 5.0 / 9.0  
LDURS S18, [X27,const32]   // S18 = 32.0  
FSUBS S18, S12, S18        // S18 = fahr - 32.0  
FMULS S0, S16, S18        // S0 = (5/9)*(fahr - 32.0)  
BR LR                     // return
```

FP Example: Array Multiplication

- $X = X + Y \times Z$
 - 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 X0, X1, X2, and
i, j, k in X19, X20, X21

FP Example: Array Multiplication

■ LEGv8 code:

mm: . . .

LDI X10, 32	// x10 = 32 (row size/loop end)
LDI X19, 0	// i = 0; initialize 1st for loop
L1: LDI X20, 0	// j = 0; restart 2nd for loop
L2: LDI X21, 0	// k = 0; restart 3rd for loop
LSL X11, X19, 5	// x11 = i * 2 5 (size of row of c)
ADD X11, X11, X20	// x11 = i * size(row) + j
LSL X11, X11, 3	// x11 = byte offset of [i][j]
ADD X11, X0, X11	// x11 = byte address of c[i][j]
LDURD D4, [X11,#0]	// D4 = 8 bytes of c[i][j]
L3: LSL X9, X21, 5	// x9 = k * 2 5 (size of row of b)
ADD X9, X9, X20	// x9 = k * size(row) + j
LSL X9, X9, 3	// x9 = byte offset of [k][j]
ADD X9, X2, X9	// x9 = byte address of b[k][j]
LDURD D16, [X9,#0]	// D16 = 8 bytes of b[k][j]
LSL X9, X19, 5	// x9 = i * 2 5 (size of row of a)

FP Example: Array Multiplication

...

ADD X9, X9, X21	// X9 = i * size(row) + k
LSL X9, X9, 3	// X9 = byte offset of [i][k]
ADD X9, X1, X9	// X9 = byte address of a[i][k]
LDURD D18, [X9,#0]	// D18 = 8 bytes of a[i][k]
FMULD D16, D18, D16	// D16 = a[i][k] * b[k][j]
FADDD D4, D4, D16	// f4 = c[i][j] + a[i][k] * b[k][j]
ADDI X21, X21, 1	// \$k = k + 1
CMP X21, X10	// test k vs. 32
B.LT L3	// if (k < 32) go to L3
STURD D4, [X11,0]	// = D4
ADDI X20, X20, #1	// \$j = j + 1
CMP X20, X10	// test j vs. 32
B.LT L2	// if (j < 32) go to L2
ADDI X19, X19, #1	// \$i = i + 1
CMP X19, X10	// test i vs. 32
B.LT L1	// if (i < 32) go to L1

Accurate Arithmetic

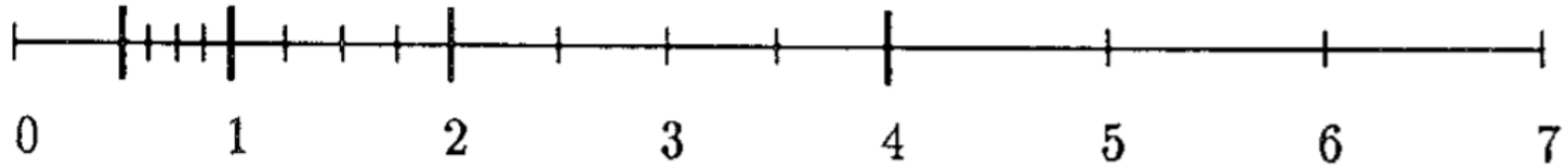
- IEEE Std 754 specifies additional rounding control
 - Extra bits of precision (guard, round, sticky)
 - Choice of rounding modes
 - Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
 - Most programming languages and FP libraries just use defaults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

Basic Principle of IEEE FP

“Each of the computational operations ... shall be performed as if it first produced an intermediate result correct to infinite precision and unbounded range, and then rounded that intermediate result to fit in the destination’s format”
(IEEE 754-2008 §5.1)

- So the machine result of $x*y$ is the rounding of the “real” result of $x*y$
- This is simple and easy to reason about
- ... and quite surprising that it can be implemented in finite hardware

Normalized Number Line



- Representable with 2 f bits and 2 e bits:
(So minimum e is -1 and maximum e is 2)

$$1.00_2 \times 2^{-1} = 0.5$$

$$1.01_2 \times 2^{-1} = 0.625$$

$$1.10_2 \times 2^{-1} = 0.75$$

$$1.11_2 \times 2^{-1} = 0.875$$

$$1.00_2 \times 2^0 = 1$$

$$1.01_2 \times 2^0 = 1.25$$

$$1.10_2 \times 2^0 = 1.5$$

$$1.11_2 \times 2^0 = 1.75$$

$$1.00_2 \times 2^1 = 2$$

$$1.01_2 \times 2^1 = 2.5$$

$$1.10_2 \times 2^1 = 3$$

$$1.11_2 \times 2^1 = 3.5$$

$$1.00_2 \times 2^2 = 4$$

$$1.01_2 \times 2^2 = 5$$

$$1.10_2 \times 2^2 = 6$$

$$1.11_2 \times 2^2 = 7$$

- Same relative precision for all numbers
- Decreasing absolute precision for large ones

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)

ARMv8 SIMD

- 32 128-bit registers (V0, ..., V31)
- Works with integer and FP
- Examples:
 - 16 8-bit integer adds:
 - `ADD V1.16B, V2.16B, V3.16B`
 - 4 32-bit FP adds:
 - `FADD V1.4S, V2.4S, V3.4S`

Other ARMv8 Features

- 245 SIMD instructions, including:
 - Square root
 - Fused multiply-add, multiply-subtract
 - Conversion and scalar and vector round-to-integral
 - Structured (strided) vector load/stores
 - Saturating arithmetic

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	F ^I ADDP mem/ST(i) F ^I SUBRP mem/ST(i) F ^I MULP mem/ST(i) F ^I DIVRP mem/ST(i) FSQRT FABS FRNDINT	F ^I COMP F ^I 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 4×128 -bit registers
 - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
 - 2×64 -bit double precision
 - 4×32 -bit double precision
 - Instructions operate on them simultaneously
 - Single-Instruction Multiple-Data

Matrix Multiply

■ Unoptimized code:

```
1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.   for (int i = 0; i < n; ++i)
4.     for (int j = 0; j < n; ++j)
5.       {
6.         double cij = C[i+j*n]; /* cij = C[i][j] */
7.         for(int k = 0; k < n; k++ )
8.           cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.         C[i+j*n] = cij; /* C[i][j] = cij */
10.      }
11. }
```

Matrix Multiply

■ 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	0.00E+00	-1.50E+38
y	1.50E+38		1.50E+38
z	1.0	1.0	
		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