

# Chapter 3

## Arithmetic for Computers

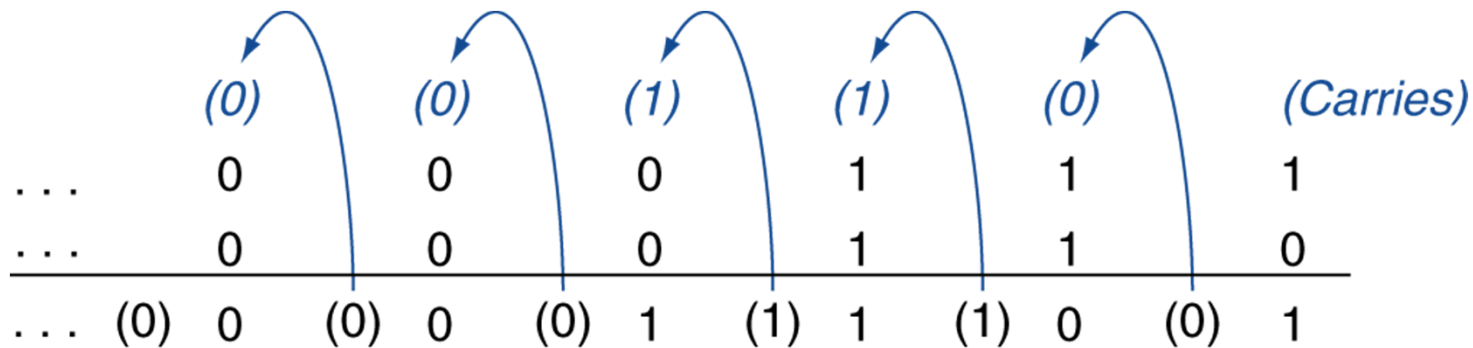
From RISC-V Edition (1<sup>st</sup>) – No Changes

# Arithmetic for Computers

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

# Integer Addition

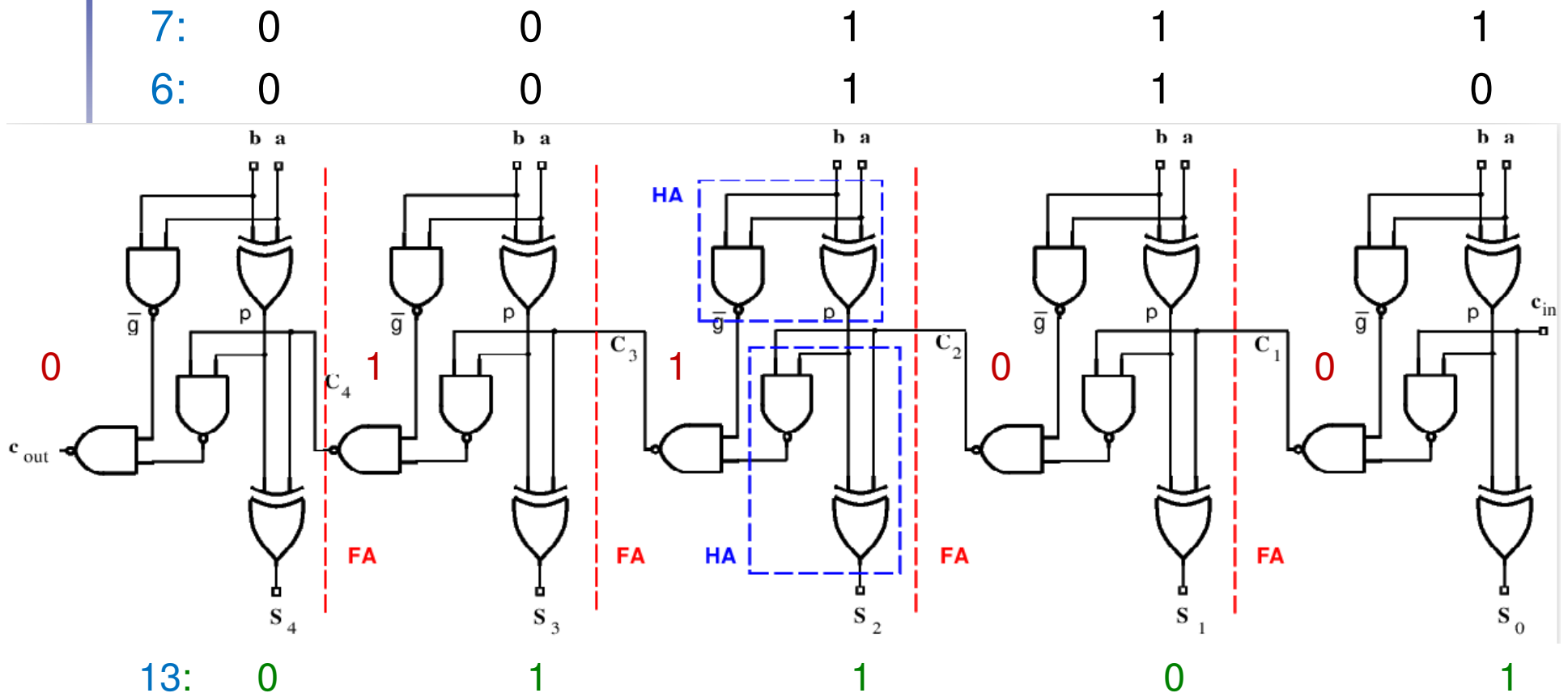
## ■ Example: $7 + 6$



## ■ 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

# Addition by Carry-Ripple Adder



Delay is approx.  $n \times t(\text{carry}) \rightarrow \text{SLOW}$

# 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

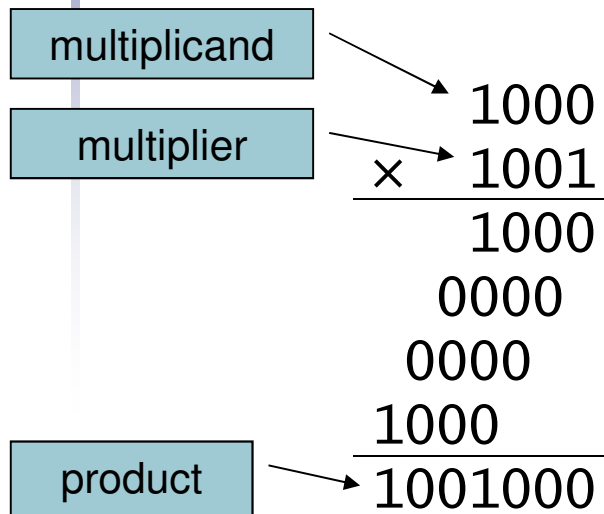
- Overflow if result out of range
  - Subtracting two +ve or two -ve operands, no overflow
  - Subtracting +ve from -ve operand
    - Overflow if result sign is 0
  - Subtracting -ve from +ve operand
    - Overflow if result sign is 1

# Arithmetic for Multimedia

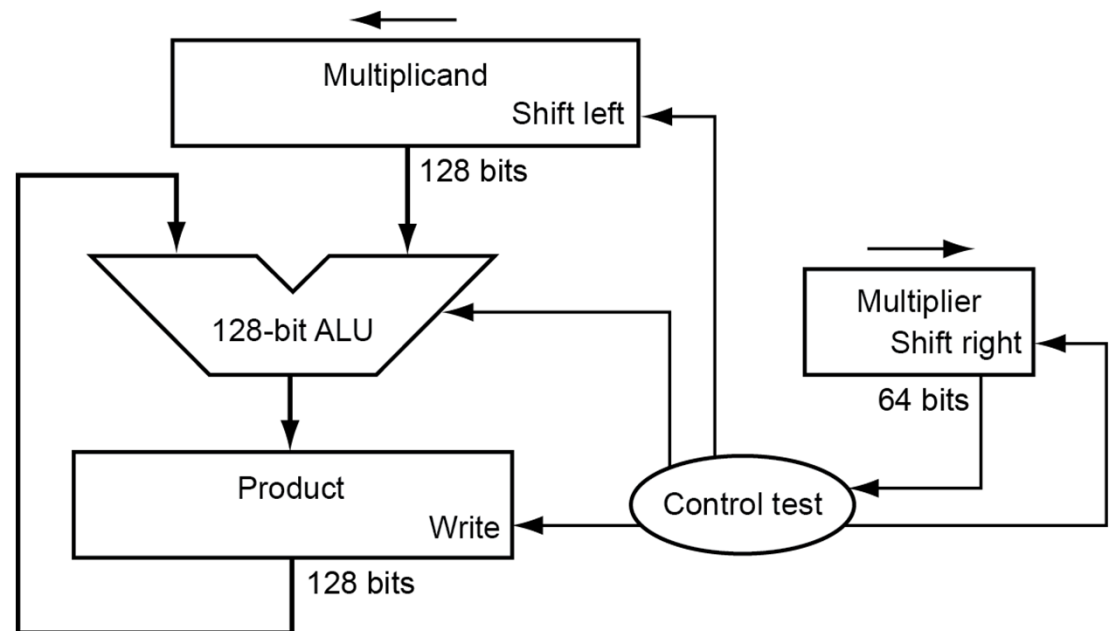
- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
  - SIMD (single-instruction, multiple-data)
- Saturating operations
  - On overflow, result is largest than representable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

# Multiplication

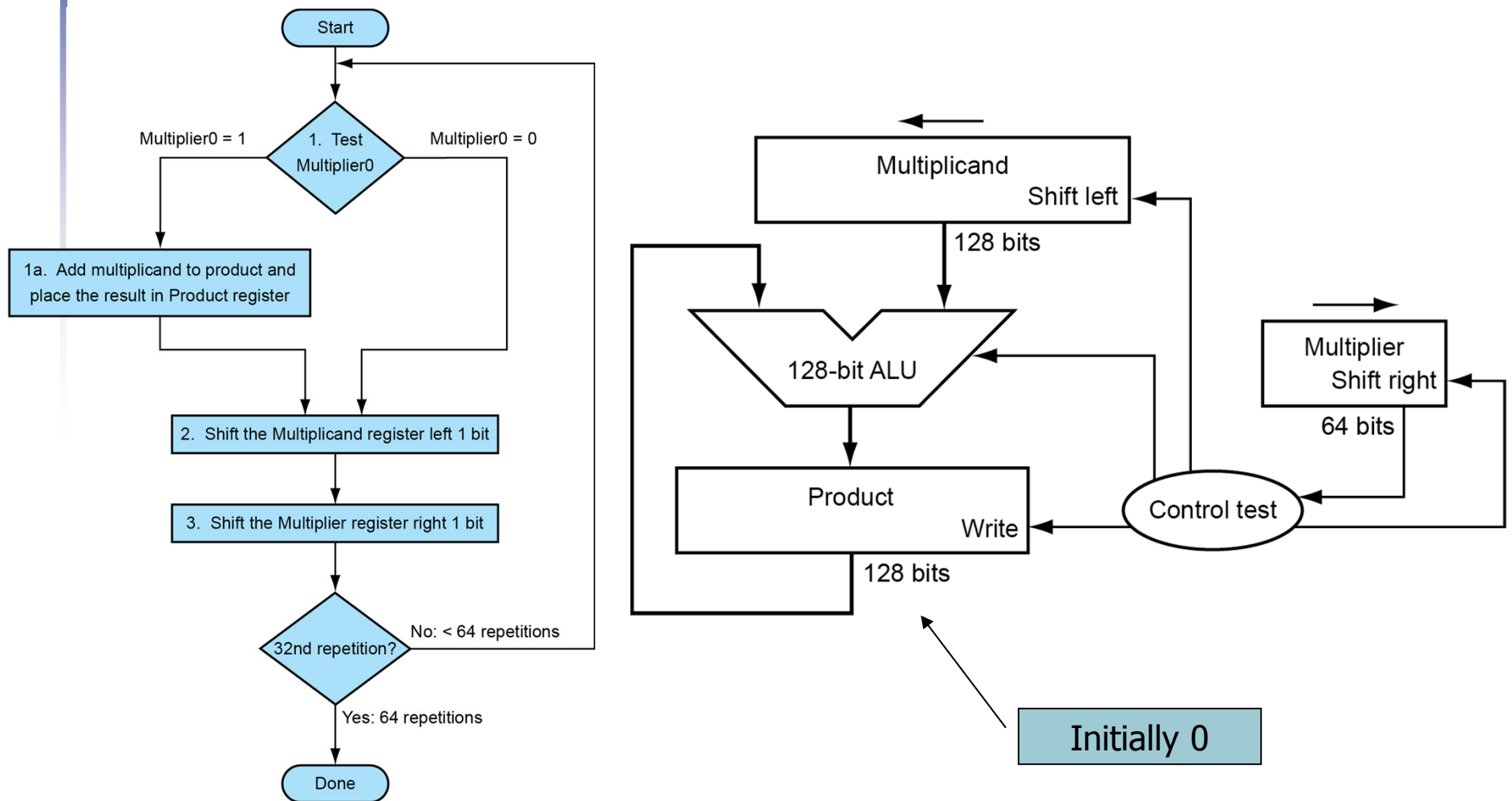
- Start with long-multiplication approach



Length of product is the sum of operand lengths



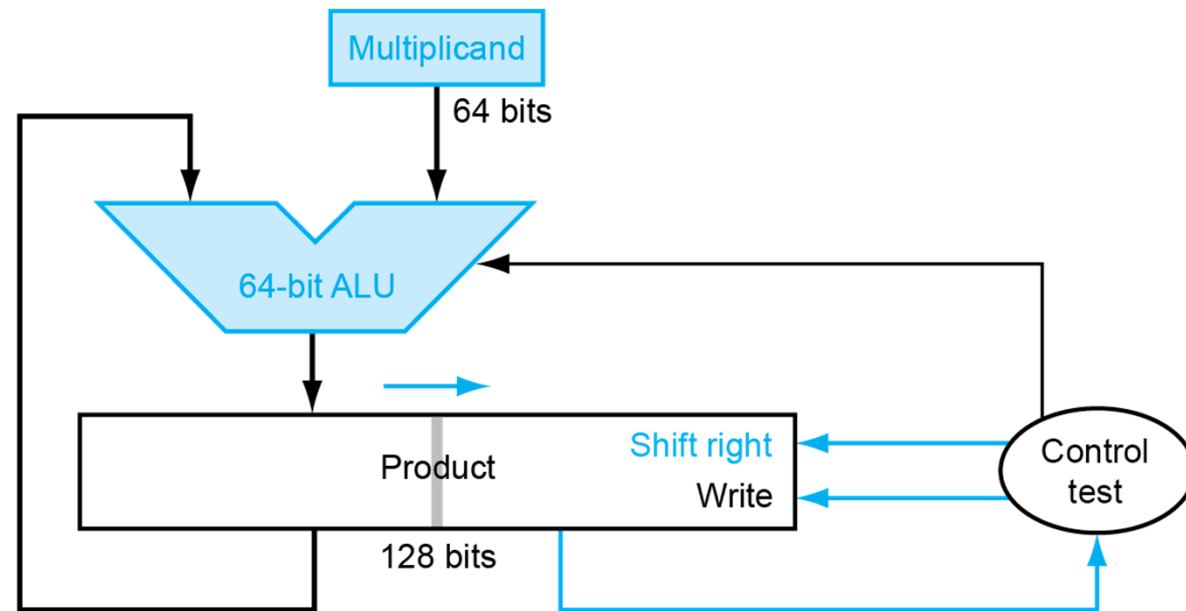
# Multiplication Hardware





# Optimized Multiplier

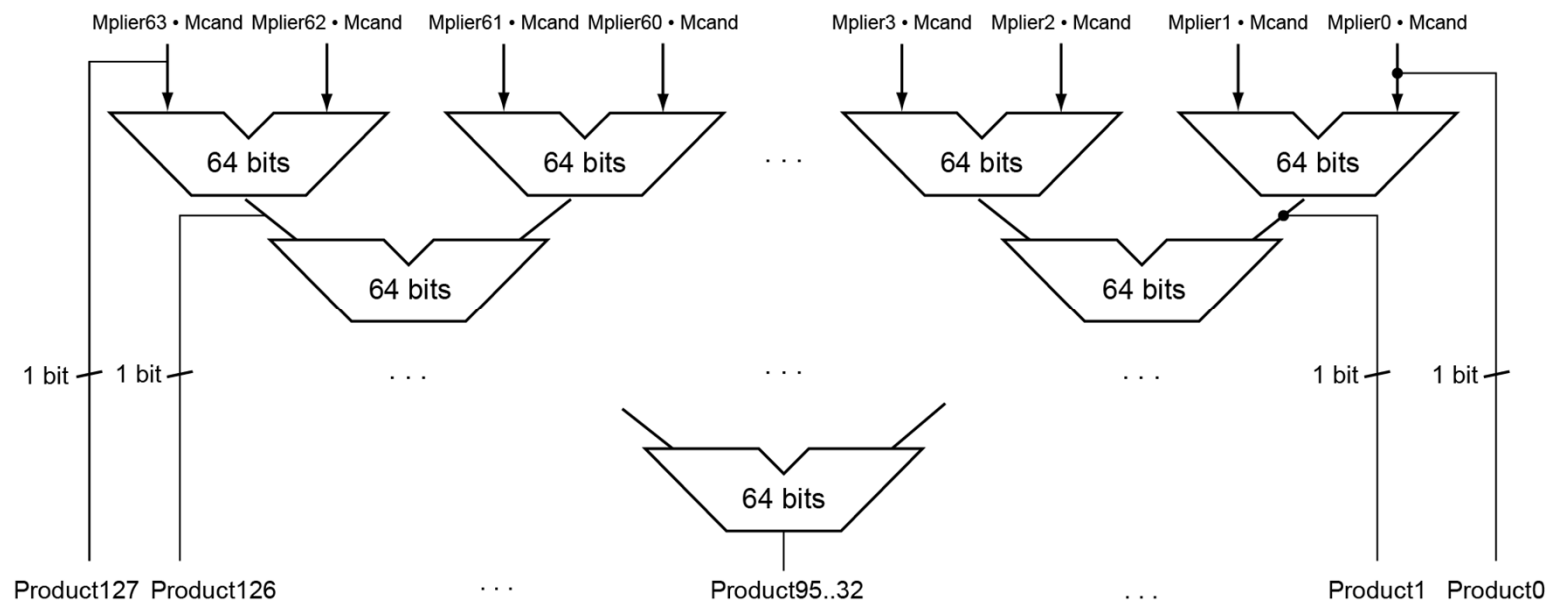
- Perform steps in parallel: add/shift



- 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 multiplications 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

- Algorithm in the book not used (too slow)
  - We skip it
- Can't use parallel hardware as in multiplier
- Faster dividers (e.g. SRT division)  
generate multiple quotient bits per step
  - Still require multiple steps  
From 10 to 28 iterations for binary64

# 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
  - $-2.34 \times 10^{56}$  ← 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

2019

- Defined by IEEE Std 754-1985 (rev. ~~2008~~)
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Common representations
  - Half precision (16-bit) — binary16
  - Single precision (32-bit) — binary32
  - Double precision (64-bit) — binary64

# IEEE Floating-Point Format

single: 8 bits

double: 11 bits

single: 23 bits

double: 52 bits

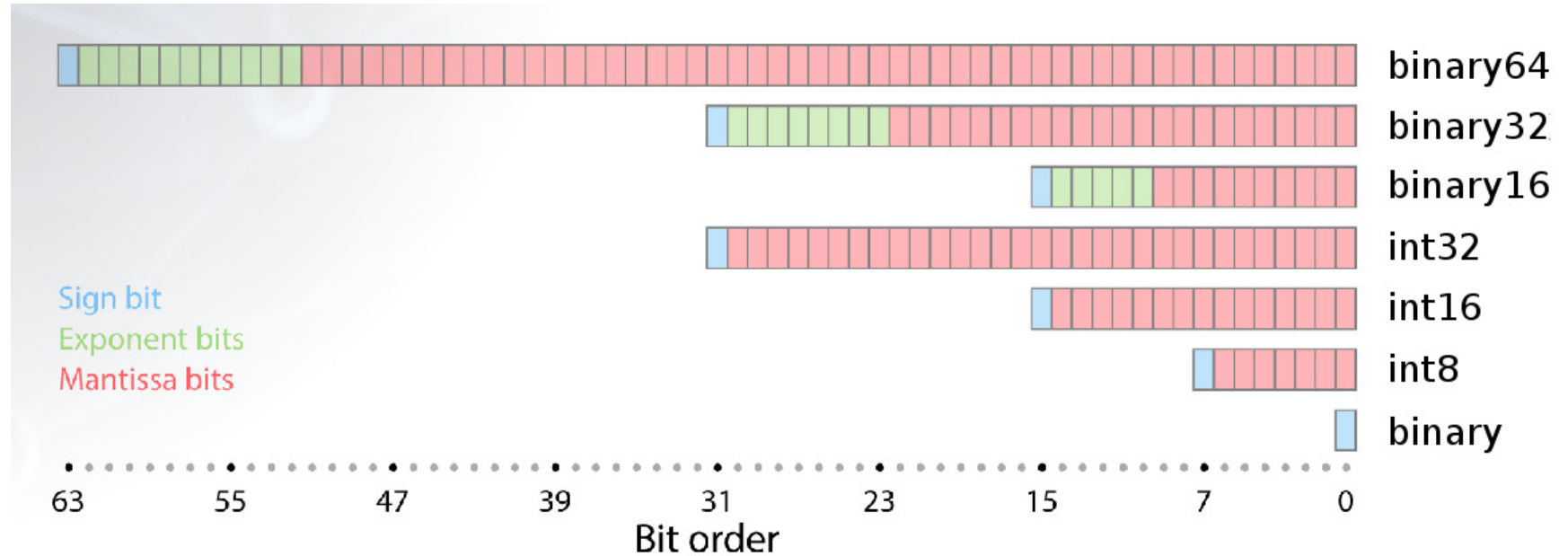
S	Exponent	Fraction
---	----------	----------

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

- 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



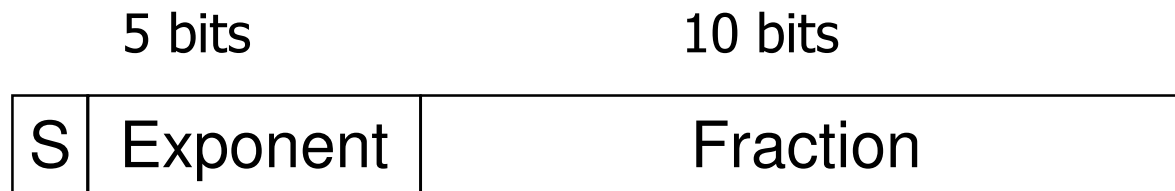
# Floating Point and Integer Formats



Source: Popescu et al, "Flexpoint: Predictive Numerics for Deep Learning", 2018

# Binary16 (introduced in 754-2008)

Recently popular for Deep Learning training



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

- Storage 16 bits
- Exponent 5 bits
- Fraction  $f = 10$  bits
- Bias = 15

# Single-Precision Range

- Exponents **00000000** and **11111111** reserved
- Smallest normalized 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 normalized 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  $\approx 2.0$
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

# Binary16 Range

Small dynamic range  $\approx 4.3 \times 10^{12}$

- Exponents **00000** and **11111** reserved
- Smallest normalized value
  - Exponent: 00000  $\Rightarrow$  actual exponent =  $1 - 15 = -14$
  - Fraction: 0000000000  $\Rightarrow$  significand = 1.0
  - $\pm 1.0 \times 2^{-14} \approx \pm 6.1 \times 10^{-5}$
- Largest value
  - exponent: 11110  $\Rightarrow$  actual exponent =  $30 - 15 = +15$
  - Fraction: 1111111111  $\Rightarrow$  significand  $\approx 2.0$
  - $\pm 2.0 \times 2^{+15} \approx \pm 6.5 \times 10^{+5}$

# Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - **binary16**: approx  $2^{-10}$ 
    - Equivalent to  $10 \times \log_{10} 2 \approx 10 \times 0.3 \approx 3$  decimal digits of precision
  - **binary32** (single): approx  $2^{-23}$ 
    - Equivalent to  $23 \times \log_{10} 2 \approx 23 \times 0.3 \approx 6$  decimal digits of precision
  - **binary64** (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 =  $-1 + \text{Bias}$ 
    - binary16:  $-1 + 15 = 14 = 01110_2$
    - Single:  $-1 + 127 = 126 = 01111110_2$
    - Double:  $-1 + 1023 = 1022 = 011111111110_2$
- binary16:  $1011101000000000$
- Single:  $1011111101000\dots\dots\dots00$
- Double:  $10111111111101000\dots\dots\dots\dots\dots\dots00$

# Floating-Point Example

- What number is represented by the binary32 (single-precision) float

11000000101000...00

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



# Subnormal (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

- Subnormal 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

# Binary32 (single precision) Examples

$S_x$	$E_x$	$M_x$	$V$
0	00000000	0.000000000000000000000000	= 0
0	01111111	1.000000000000000000000000	= $2^{127-127} \cdot (1.0)_2 = 1$
0	01111110	1.000000000000000000000000	= $2^{126-127} \cdot (1.0)_2 = 0.5$
0	10000000	1.000000000000000000000000	= $2^{128-127} \cdot (1.0)_2 = 2$
0	10000001	1.101000000000000000000000	= $2^{129-127} \cdot (1.101)_2 = 6.5$
1	10000001	1.101000000000000000000000	= $-[2^{129-127} \cdot (1.101)_2] = -6.5$
0	00000001	1.000000000000000000000000	= $2^{1-127} \cdot (1.0)_2 = 2^{-126}$
0	00000000	0.100000000000000000000000	= $2^{-126} \cdot (0.1)_2 = 2^{-127}$
0	00000000	0.000000000000000000000001	= $2^{-126} \cdot (0.000000000000000000000001)_2$ = $2^{-149}$ (Smallest positive value)
0	11111111	000000000000000000000000	= $\infty$
1	11111111	000000000000000000000000	= $-\infty$
1	11111111	100000000000000000000000	= $\sqrt{-1}$ (NaN)

# 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
  - $1.002 \times 10^2$

SKIP

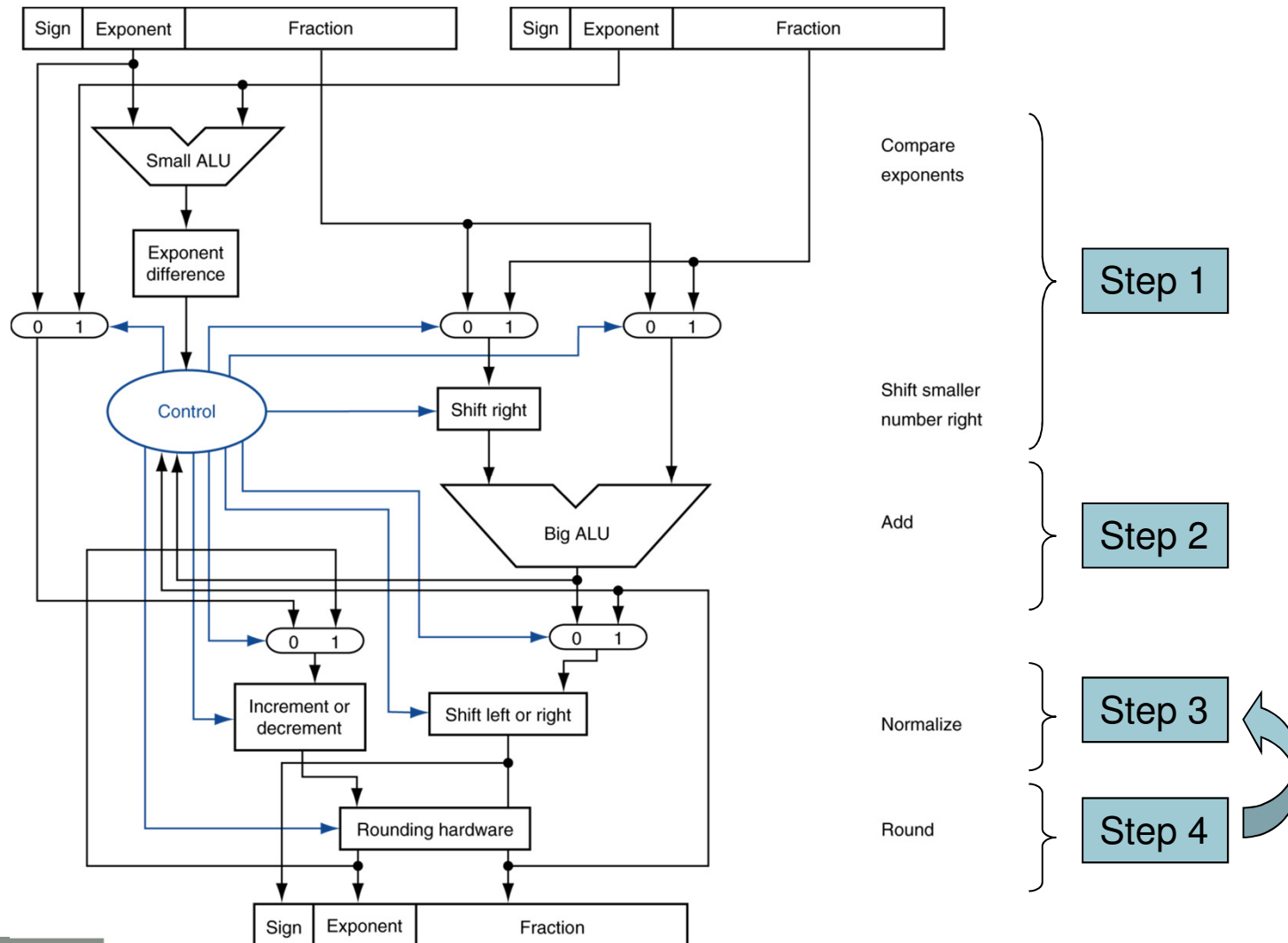
# Floating-Point Addition

- 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



# Rounding

Need extra bits for rounding

❑ Three additional bits sufficient:

guard (G), round (R), and sticky (T)

❑ Examples for binary16 (f=10):

L G R T

$$\begin{array}{r} 1.100111101001 \\ \phantom{1.100111101001} 1 \\ \hline 1.1001111010 \end{array}$$

← Normalized  
← add rounding bit (if R or T = 1)  
← Rounded (f=10 → binary16)

L G R T

$$\begin{array}{r} 0.100111101001 \\ 1.00111101001 \\ \phantom{1.00111101001} 0 \\ \hline 1.0011110100 \end{array}$$

← need normalization  
← Normalized  
← T=0 no rounding bit  
← Rounded (f=10 → binary16)



# Floating-Point Multiplication

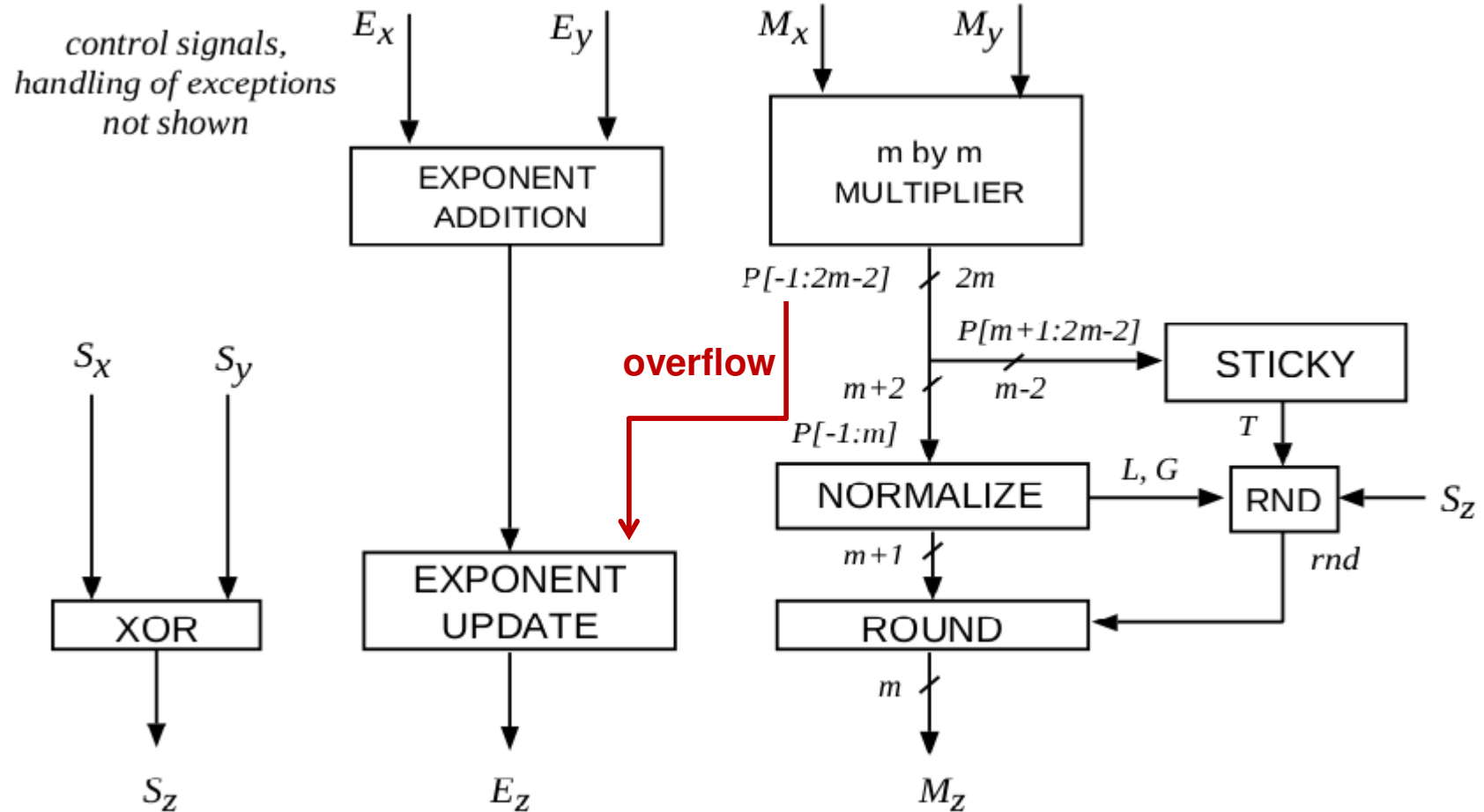
- Consider a 4-digit decimal example
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
  - For biased exponents, subtract bias from sum
  - New exponent =  $10 + -5 = 5$
- 2. Multiply significands
  - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^5$
- 3. Normalize 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
  - $+1.021 \times 10^6$

SKIP

# 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 Multiplier Hardware



Source: Ercegovac & Lang, "Digital Arithmetic", Morgan Kaufmann, 2003.

# 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
  - $\text{FP} \leftrightarrow \text{integer}$  conversion
- Operations usually takes several cycles
  - Can be pipelined (not div, rec & sqrt)

# 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
  - flw, fld
  - fsw, fsd

# FP Instructions in RISC-V

- Single-precision arithmetic
  - `fadd.s`, `fsub.s`, `fmul.s`, `fdiv.s`, `fsqrt.s`
    - e.g., `fadds.s f2, f4, f6`
- Double-precision arithmetic
  - `fadd.d`, `fsub.d`, `fmul.d`, `fdiv.d`, `fsqrt.d`
    - e.g., `fadd.d f2, f4, f6`
- Single- and double-precision comparison
  - `feq.s`, `flt.s`, `fle.s`
  - `feq.d`, `flt.d`, `fle.d`
  - Result is 0 or 1 in integer destination register
    - Use `beq`, `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;
```

Do it at home as Exercise

```
    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



# 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)

# 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

# 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.  $11111011_2 \gg 2 = 00111110_2 = +62$

# 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*

# Disasters Caused by Numerical Errors

On June 4, 1996 an unmanned **Ariane 5** rocket launched by ESA exploded just forty seconds after lift-off



*Source:* <http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html>

# Ariane 5 Explosion

Cause of failure: software error in inertial reference system

- A 64 bit floating-point number (horizontal velocity of rocket with respect to launch platform) was converted to a 16 bit signed integer



- The number larger than max. 16-bit signed integer (32,768) overflowed resulting in “*garbage*”
- The conversion failed causing an “exception” in the guidance software → mission aborted



Cost of rocket and payload estimated to \$500 millions



# 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