Arithmetic for Computers

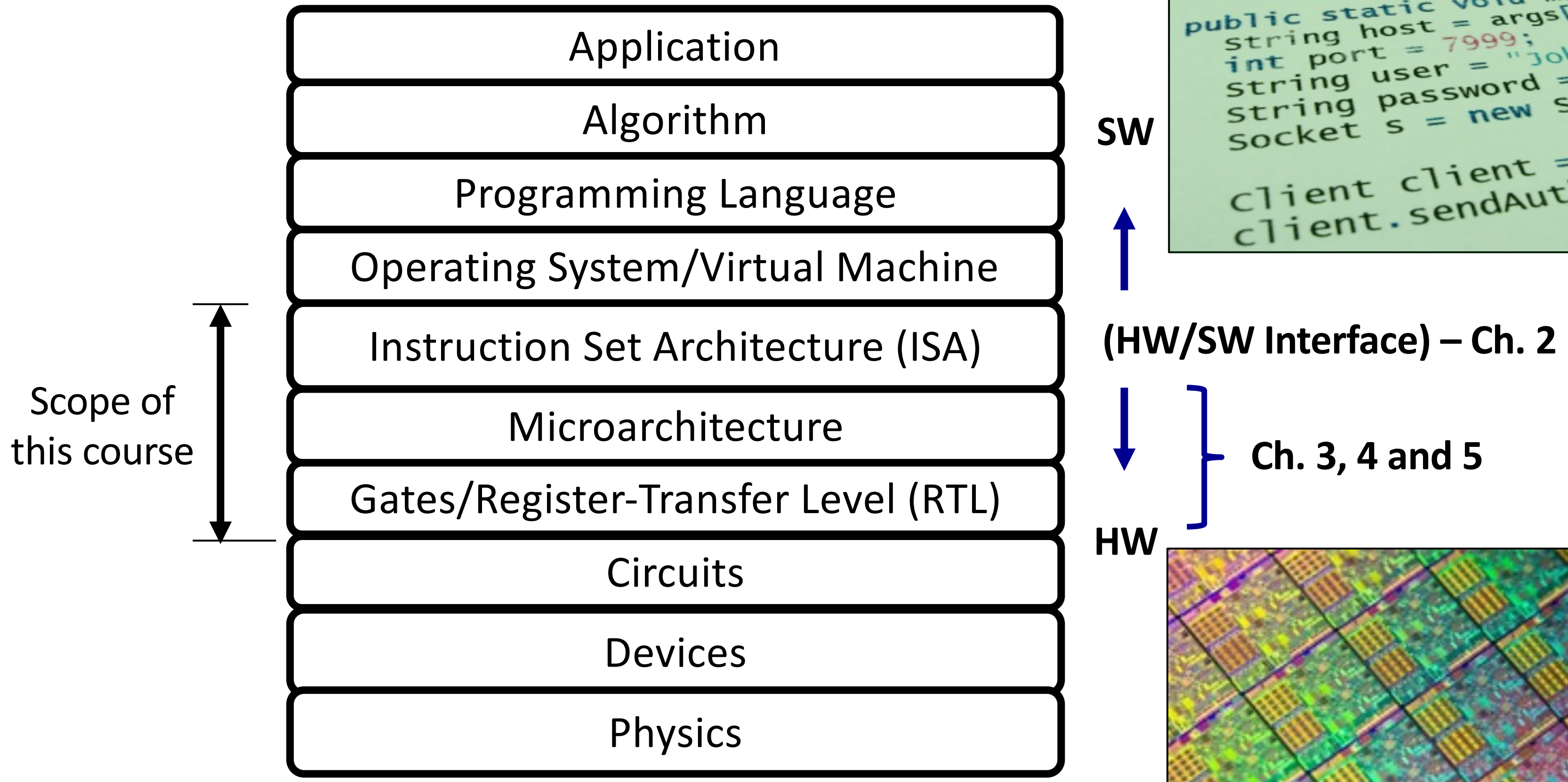
Lecture 3
September 20th, 2023

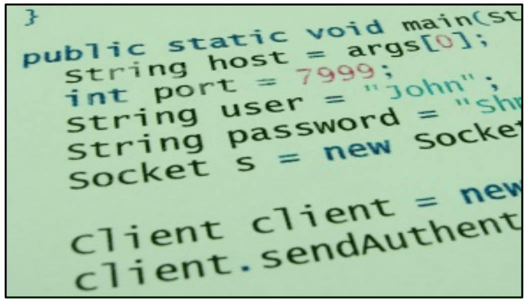
Jae W. Lee (jaewlee@snu.ac.kr)
Computer Science and Engineering
Seoul National University

***Slide credits**: [CS:APP3e] slides from CMU; [COD:RV2e] slides from Elsevier Inc.*

Where Are We?

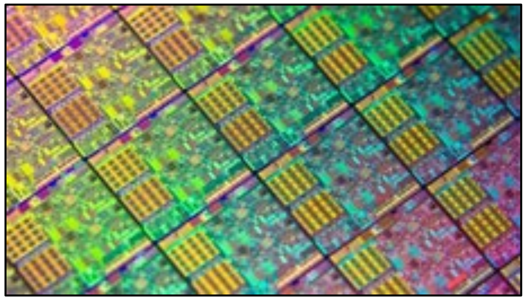
- **Abstraction Layers in Modern Systems**


Application
Algorithm
Programming Language
Operating System/Virtual Machine
Instruction Set Architecture (ISA)
Microarchitecture
Gates/Register-Transfer Level (RTL)
Circuits
Devices
Physics
Scope of this course
SW
(HW/SW Interface) – Ch. 2
Ch. 3, 4 and 5
HW



```
}
public static void main(St
    String host = args[0];
    int port = 7999;
    String user = "John";
    String password = "Sh
    Socket s = new Socke

    Client client = new
    client.sendAuthent
```



* **Sources: CS 252 lecture notes from Prof. Kubiatowicz (UC Berkeley).**
Coursera lecture notes for HW/SW Interface from Profs. Borriello and Ceze (Univ. of Washington).

Introduction

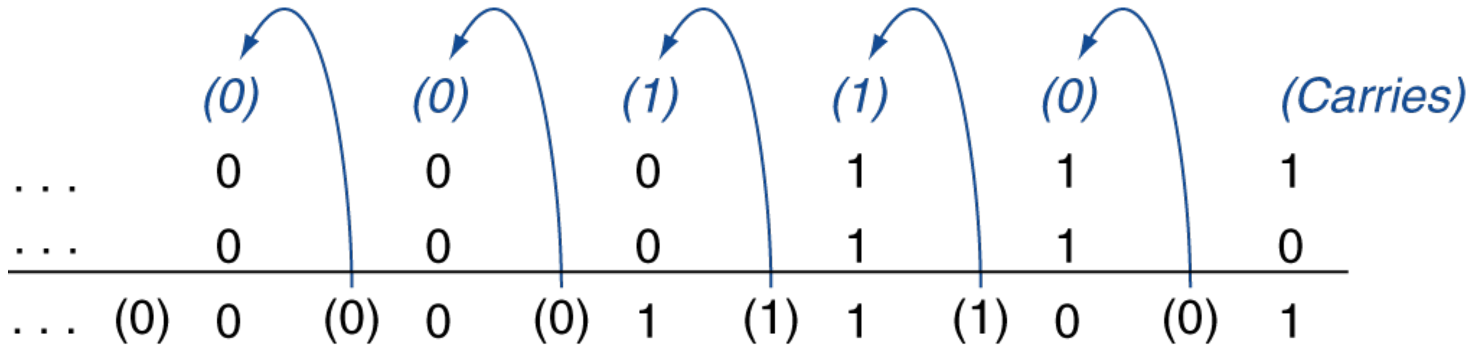
- **Bits are just bits**
  - No inherent meaning: conventions define relationship between bits and numbers
  - *n*-bit binary numbers: 0 ~ 2^n-1 decimal numbers
- **Of course it gets more complicated**
  - Numbers are finite (overflow)
  - Negative numbers
  - Fractions and real numbers
  - Precision and accuracy
  - Error propagation, ...

Arithmetic for Computers: An Overview

- **Operations on integers**
  - Textbook: COD 3.1-3.4
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- **Floating-point real numbers**
  - Textbook: P&H 3.5
  - Representation and operations
- **Subword parallelism**
  - Textbook: P&H 3.6

Integer Addition

- **Example: 7 + 6**



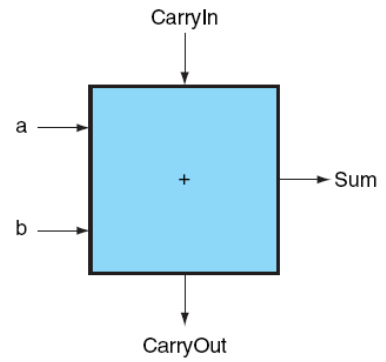
- **Overflow if result out of range**
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if result sign is 1
  - Adding two –ve operands
    - Overflow if result sign is 0

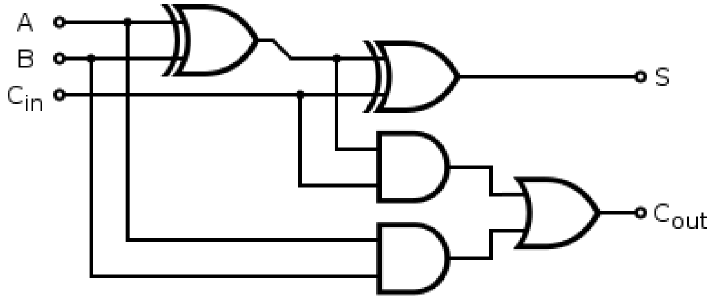
Integer Subtraction

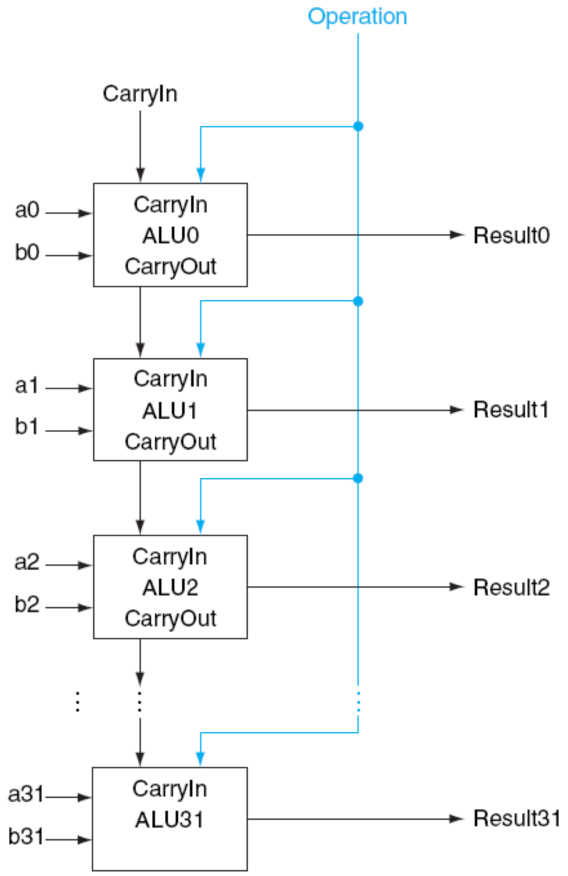
- **Add negation of second operand**
- **Example: 7 – 6 = 7 + (–6)**
  - +7: 0000 0000 … 0000 0111
    –6: 1111 1111 … 1111 1010
    +1: 0000 0000 … 0000 0001
- **Overflow if result out of range**
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 1

Simple Adder

- *n*-bit ripple-carry adder

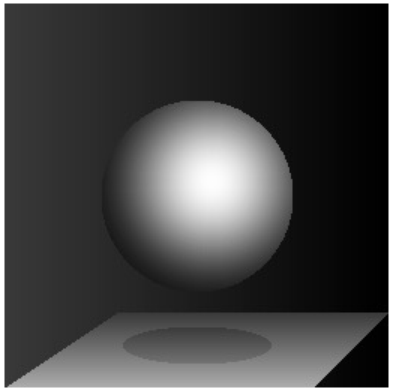


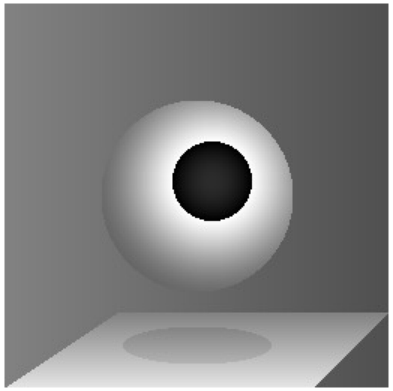




Arithmetic for Multimedia

- **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
    - cf. 2s-complement modulo arithmetic
  - e.g., clipping in audio, saturation in video



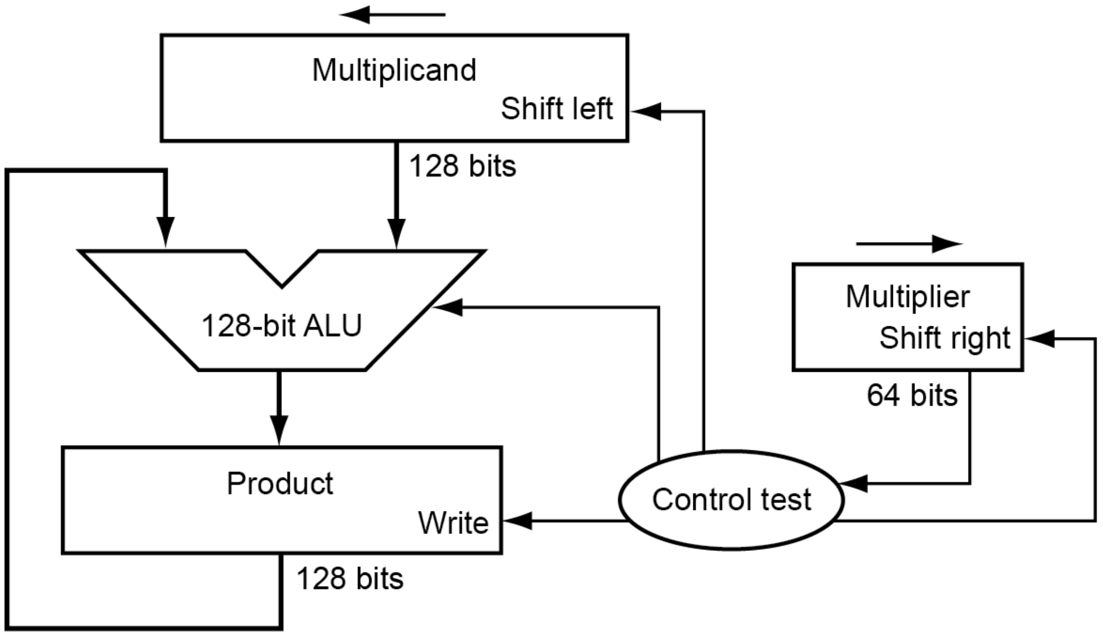


Multiplication

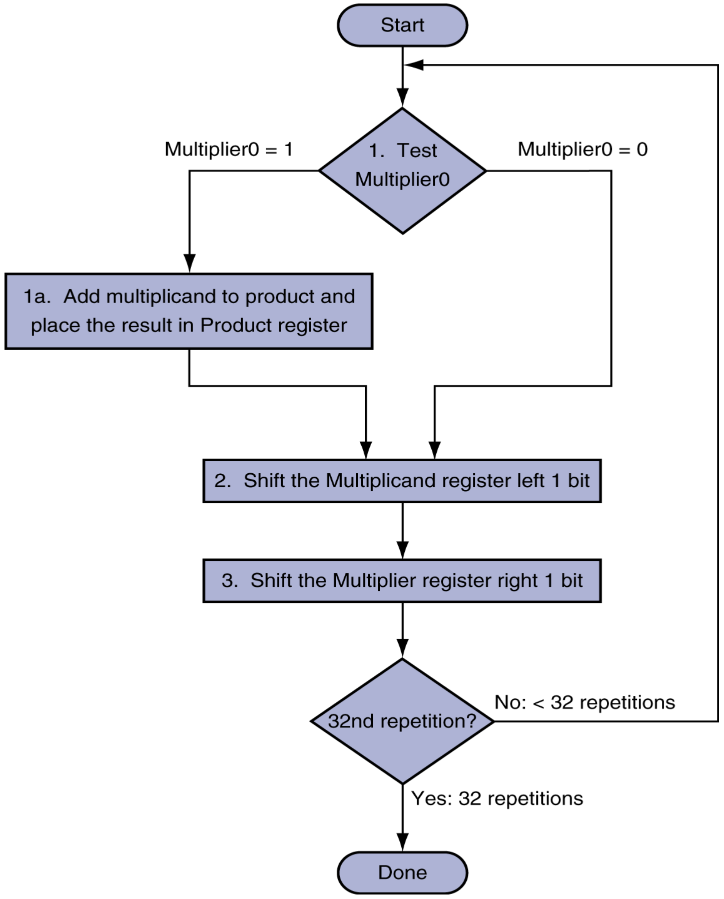
- Start with long-multiplication approach

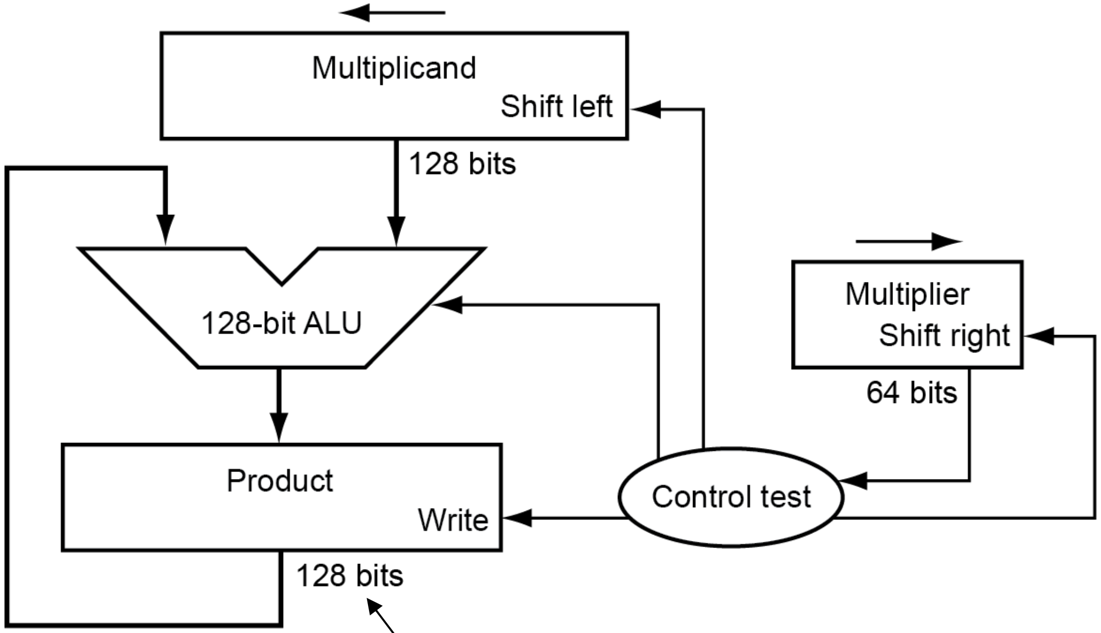
```
multiplicand  →      1000
multiplier    →  ×   1001
                 --------
                     1000
                    0000
                   0000
                  1000
                 --------
product       →   1001000
```

Length of product is the sum of operand lengths



Multiplication Hardware

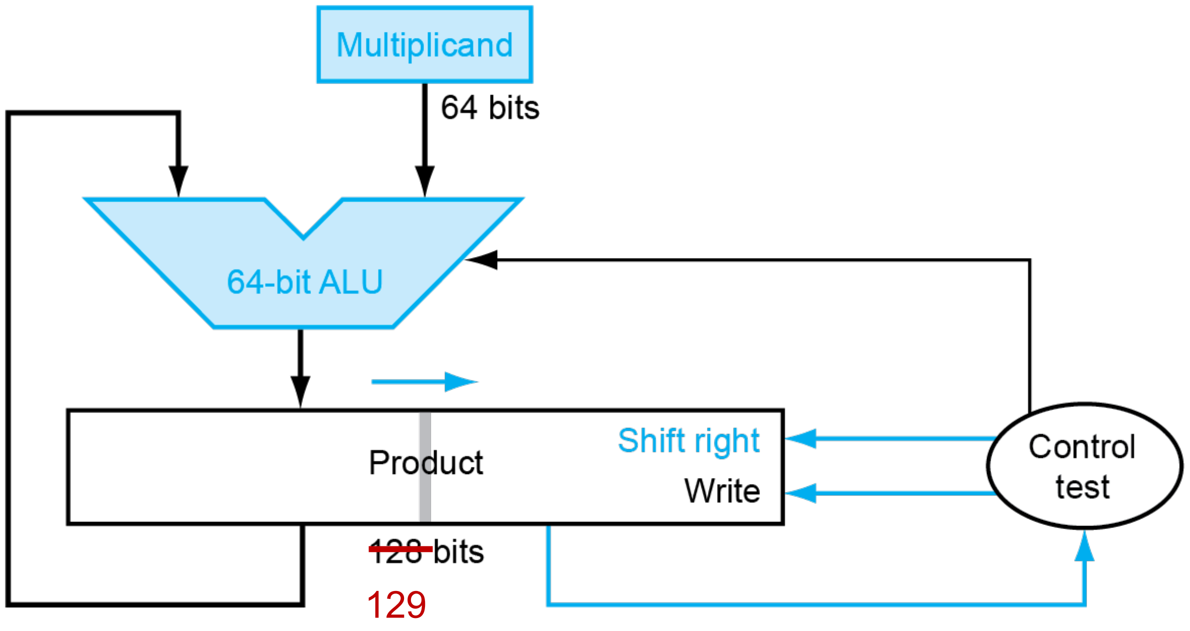

Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done



Initially 0

Optimized Multiplier

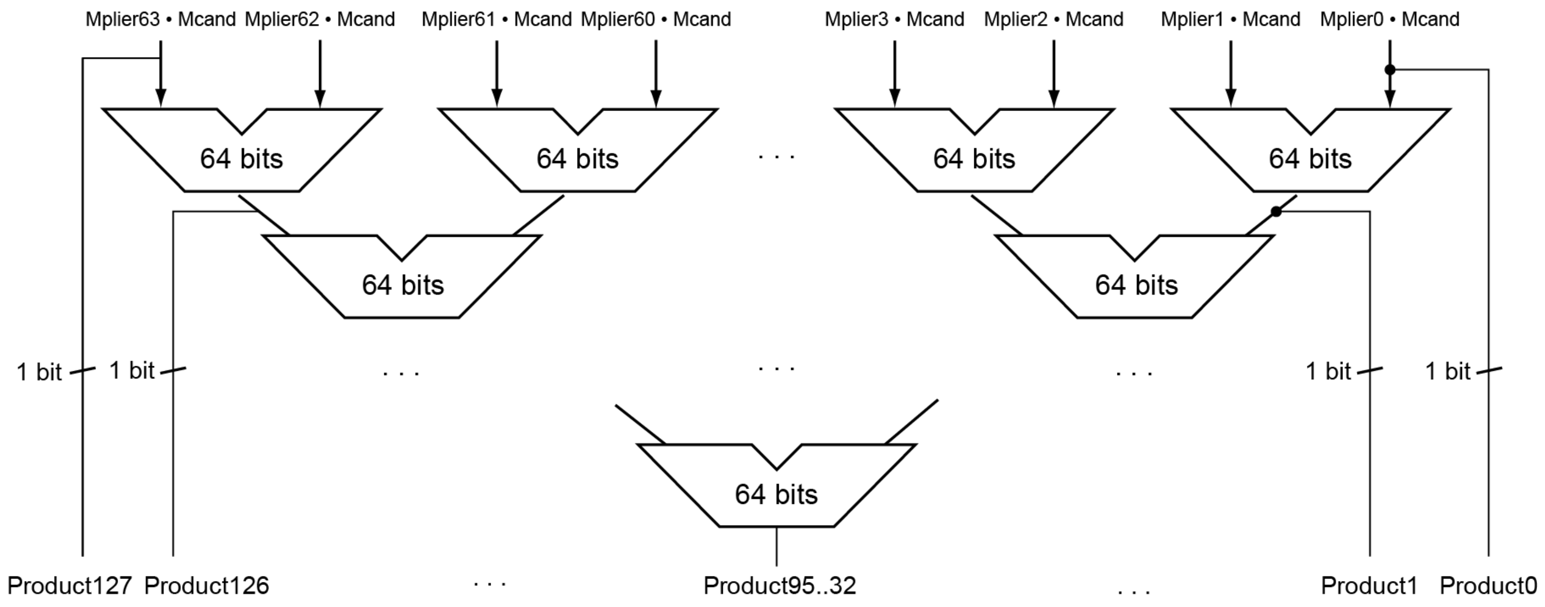
- **Perform steps in parallel: add/shift**



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

Faster Multiplier

- **Uses multiple adders**
  - Cost/performance tradeoff

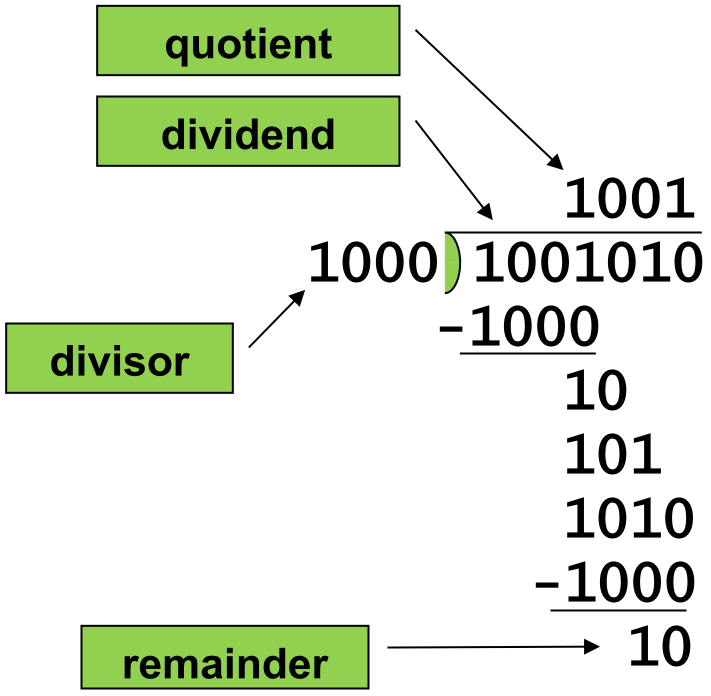


- **Can be pipelined**
  - Several multiplication performed in parallel

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

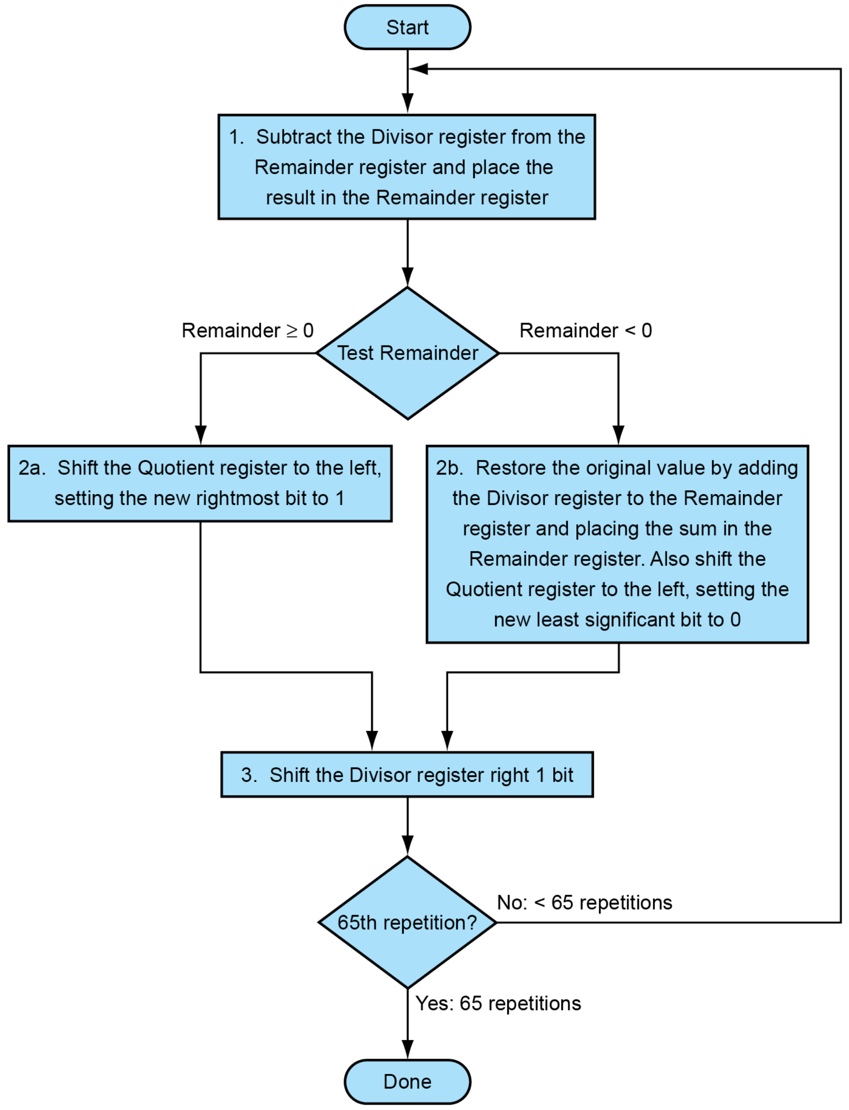
Division



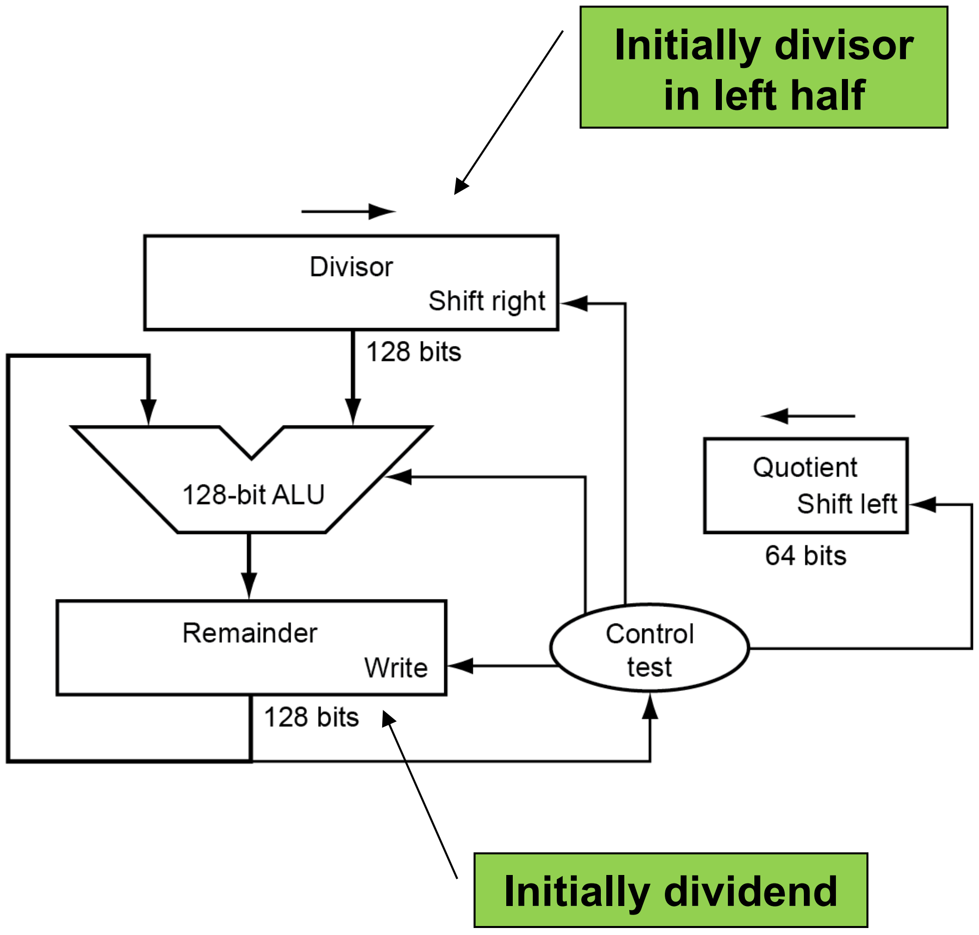
***n*-bit operands yield *n*-bit quotient and remainder**

- **Check for 0 divisor**
- **Long division approach**
  - If divisor ≤ 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

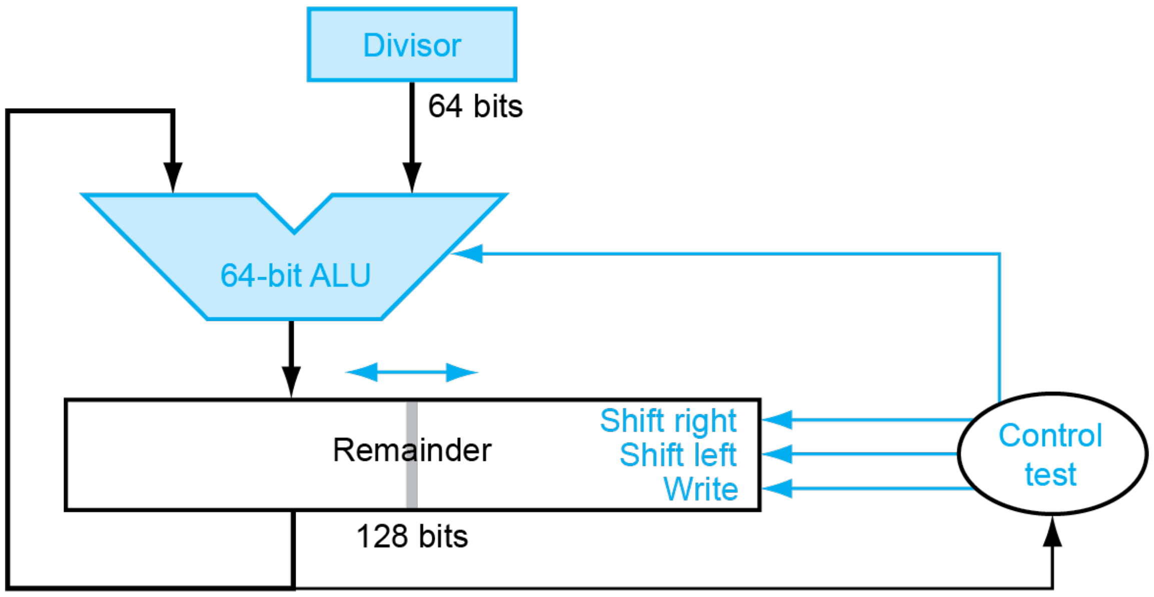
Division Hardware



Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
65th repetition?
No: < 65 repetitions
Yes: 65 repetitions
Done



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

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 >> 2 = 11111110_2 = –2
    - Rounds toward –∞
  - c.f. 11111011_2 >>> 2 = 00111110_2 = +62

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

Arithmetic for Computers: An Overview

- **Operations on integers**
  - Textbook: COD 3.1-3.4
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- **Floating-point real numbers**
  - Textbook: P&H 3.5
  - Representation and operations
- **Subword parallelism**
  - Textbook: P&H 3.6

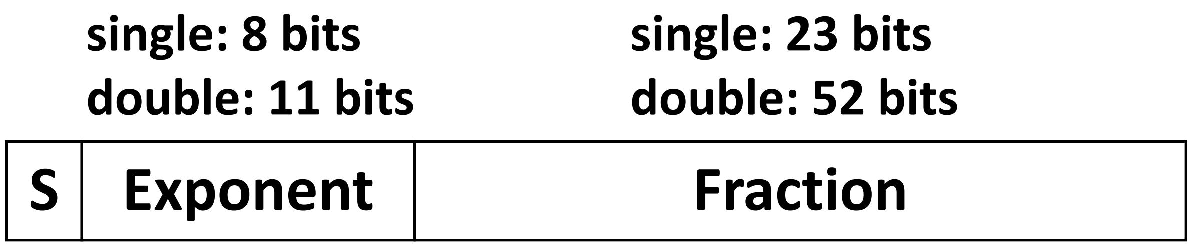
Floating Point

- **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



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

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

Single-Precision Range

- **Exponents 00000000 and 11111111 reserved**
- **Smallest value**
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- **Largest value**
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 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: 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 ≈ 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: approximately 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approximately 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 = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2
- **Single: 1011111101000…00**
- **Double: 101111111111101000…00**

Floating-Point Example

- **What number is represented by the single-precision float**

  11000000101000…00

  - S = 1
  - Fraction = $01000…00_2$
  - Fxponent = 10000001_2 = 129

- **$x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$**

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

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!

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 ≠ 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}

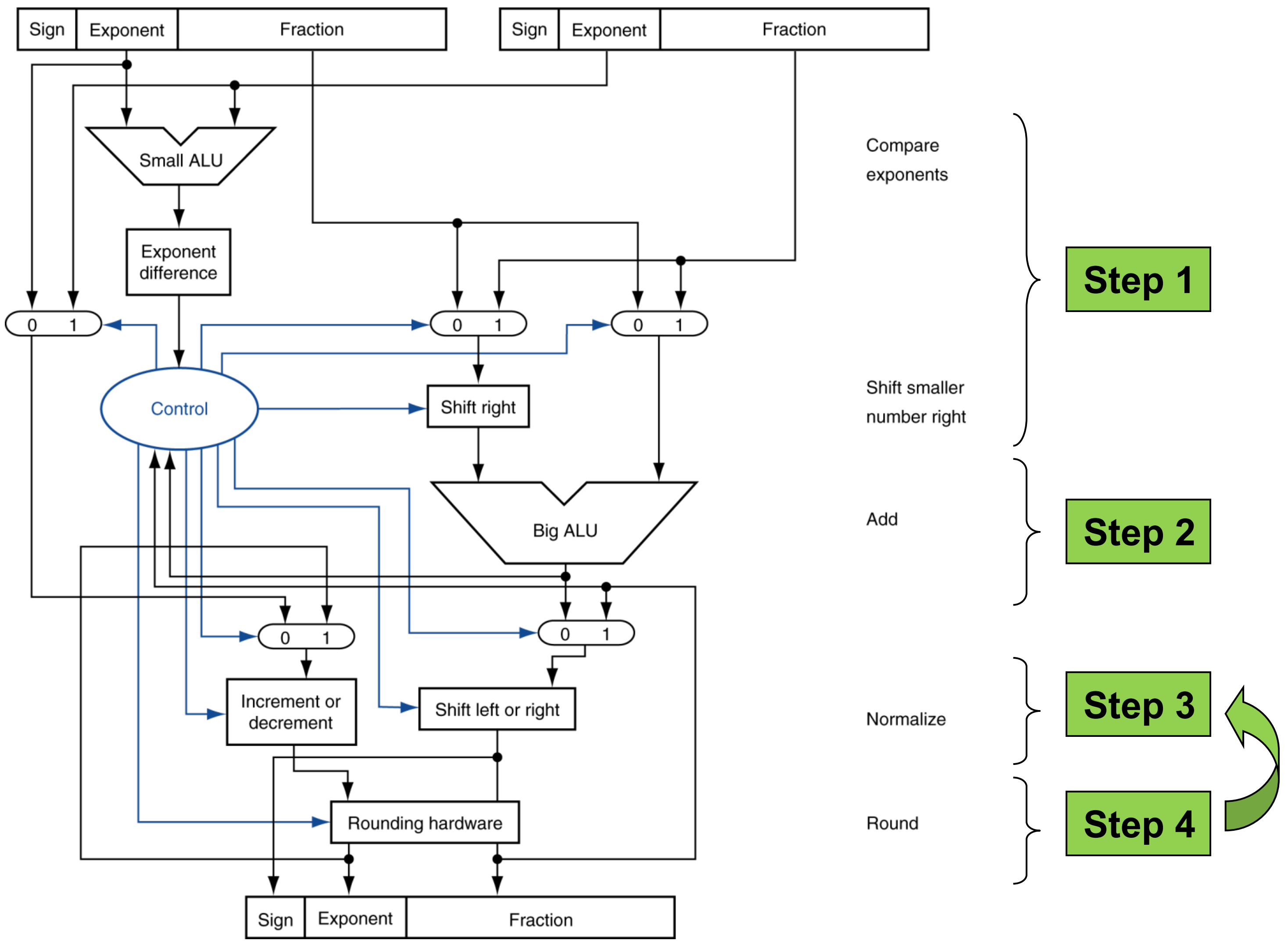
Floating-Point Addition

- **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 × –ve ⇒ –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 ↔ 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**
  - `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:
 flw    f0,const5(x3)  // f0 = 5.0f
 flw    f1,const9(x3)  // f1 = 9.0f
 fdiv.s f0, f0, f1  // f0 = 5.0f / 9.0f
 flw    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 × B
  - All 32 × 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:...
       li    x28,32       // x28 = 32 (row size/loop end)
       li    x5,0         // i = 0; initialize 1st for loop
L1:    li    x6,0         // j = 0; initialize 2nd for loop
L2:    li    x7,0         // k = 0; initialize 3rd for loop
       slli  x30,x5,5     // x30 = i * 2**5 (size of row of c)
       add   x30,x30,x6   // x30 = i * size(row) + j
       slli  x30,x30,3    // x30 = byte offset of [i][j]
       add   x30,x10,x30  // x30 = byte address of c[i][j]
       fld   f0,0(x30)    // f0 = c[i][j]
L3:    slli  x29,x7,5     // x29 = k * 2**5 (size of row of b)
       add   x29,x29,x6   // x29 = k * size(row) + j
       slli  x29,x29,3    // x29 = byte offset of [k][j]
       add   x29,x12,x29  // x29 = byte address of b[k][j]
       fld   f1,0(x29)    // f1 = b[k][j]
```

FP Example: Array Multiplication

…

```
slli   x29,x5,5     // x29 = i * 2**5 (size of row of a)
add    x29,x29,x7   // x29 = i * size(row) + k
slli   x29,x29,3    // x29 = byte offset of [i][k]
add    x29,x11,x29  // x29 = byte address of a[i][k]
fld    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[k][j]
addi   x7,x7,1      // k = k + 1
bltu   x7,x28,L3    // if (k < 32) go to L3
fsd    f0,0(x30)    // c[i][j] = f0
addi   x6,x6,1      // j = j + 1
bltu   x6,x28,L2    // if (j < 32) go to L2
addi   x5,x5,1      // i = i + 1
bltu   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 FP units implement all options**
  - Most programming languages and FP libraries just use defaults
- **Trade-off between hardware complexity, performance, and market requirements**

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

Arithmetic for Computers: An Overview

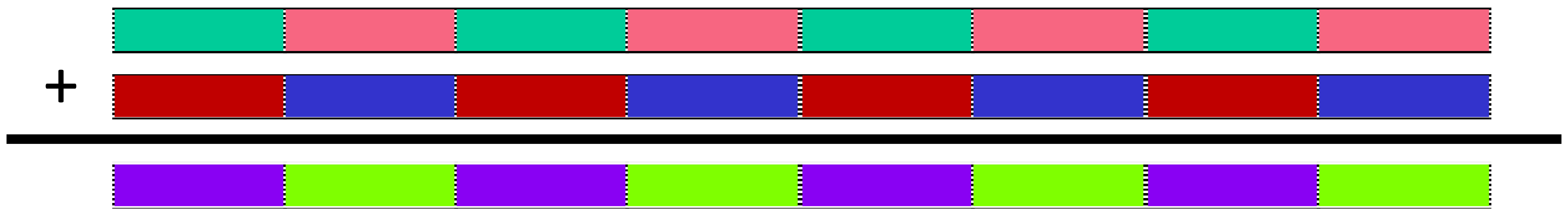
- Operations on integers
  - Textbook: COD 3.1-3.4
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Textbook: P&H 3.5
  - Representation and operations
- **Subword parallelism**
  - Textbook: P&H 3.6

Subword Parallellism

- **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)**

SIMD Extensions

- **Example: Intel MMX (1997)**
  - 57 additional 80x86 instructions (1st since 386)
    - similar to Intel 860, Mot. 88110, HP PA-71000LC, UltraSPARC
  - 3 data types: 8 8-bit, 4 16-bit, 2 32-bit in 64bits
    - reuse 8 FP registers (FP and MMX cannot mix)
  - short vector: load, add, store 8 8-bit operands



  - Claim: overall speedup 1.5 to 2X for 2D/3D graphics, audio, video, speech, comm., ...
    - use in drivers or added to library routines; no compiler

SIMD Extensions

- **Example: Intel MMX instructions**
  - Move 32b, 64b
  - Add, Subtract in parallel: 8 8b, 4 16b, 2 32b
    - opt. signed/unsigned saturate (set to max) if overflow
  - Shifts (sll,srl, sra), And, And Not, Or, Xor in parallel: 8 8b, 4 16b, 2 32b
  - Multiply, Multiply-Add in parallel: 4 16b
  - Compare = , > in parallel: 8 8b, 4 16b, 2 32b
    - sets field to 0s (false) or 1s (true); removes branches
  - Pack/Unpack
    - Convert 32b<–> 16b, 16b <–> 8b
    - Pack saturates (set to max) if number is too large

SIMD Extensions

- **Streaming SIMD Extension 2 (SSE2) (2001)**
  - 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

- **Up-to-date version: AVX-512 (2015)**
  - Advanced Vector Extensions
  - 512-bit SIMD registers
  - Targets machine learning, high-performance computing, etc.

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