Lecture 8: Computer Numbers & Arithmetic

- Last Time
  - Role of the Compiler
- Today
  - Take QUIZ 5 before 11:59pm today over Chapter 3 readings
  - Topics
    - Number Representations
    - Computer Arithmetic

Computer Arithmetic

- How do we represent and operate on unsigned/signed integers and real numbers in a finite number of bits?
- What is overflow and underflow?
- How do the arithmetic units work?

Unsigned Binary Integers

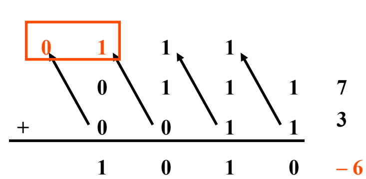
- Given an n-bit number

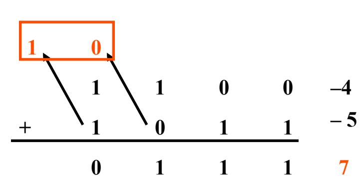
$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295
- What happens if you add 1 to 4,294,967,295?

Overflow Detection

- Overflow: the result is too large (or too small) to represent properly
  - Example: - 8 < = 4-bit binary number <= 7
- When adding operands with different signs, overflow cannot occur!
- Overflow occurs when adding:
  - 2 positive numbers and the sum is negative
  - 2 negative numbers and the sum is positive





Dealing with Overflow

- Some languages (e.g., C) ignore overflow
  - Use MIPS addu, addui, subu instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
  - Use MIPS add, addi, sub instructions
  - On overflow, invoke OS exception handler
    - Save PC in exception program counter (EPC) register
    - Jump to predefined handler address
    - mfc0 (move from coprocessor reg) instruction can retrieve EPC value to an OS reserved register ($k0) to return after corrective action

What About Signed Integers?

- Say we use one bit for the sign and the lower bits for the numbers?
- Example of an 8 bit signed number in 1's complement:

  0000 0001 = 1
  1000 0001 = -1

- What is 1 - 1?
- What is zero?

Signed Integers: 2s-Complement

- Represents: -2^{N-1} to $+2^{N-1}-1$

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

  - Bit 31 is sign bit
    - 1 for negative numbers
    - 0 for non-negative numbers
  - Positive numbers have the same unsigned and 2s -complement representation: $(0)2^{n-1}$ + …
  - Negative numbers are "complement" (0->1, 1->0) of the positive number + 1: - $(1)2^{N-1}$ + …
- *Addition and subtraction need not examine the operand signs! Makes them simpler to implement.*

2s-Complement

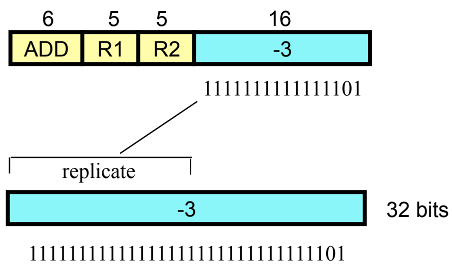
- Example values for 8-bit two's complement integers (most-significant bit on left)

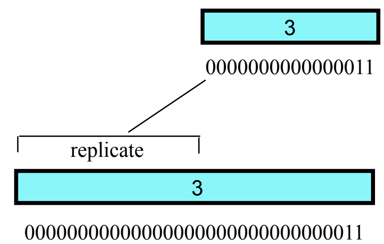
0 1 1 1 1 1 1 1 = 127
0 1 1 1 1 1 1 0 = 126
0 0 0 0 0 0 1 0 = 2
0 0 0 0 0 0 0 1 = 1
0 0 0 0 0 0 0 0 = 0
1 1 1 1 1 1 1 1 = -1
1 1 1 1 1 1 1 0 = -2
1 0 0 0 0 0 0 1 = -127
1 0 0 0 0 0 0 0 = -128

- Now, What is 1 – 1?

Conversion of 16 bit immediates to 32 bits for performing arithmatic

ADD R2, R1, -3

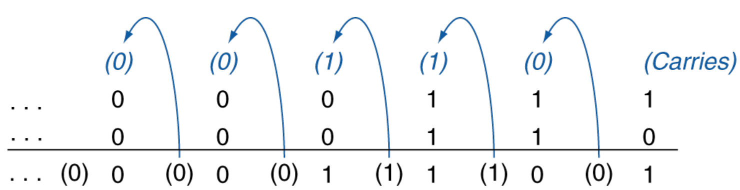




This is called “sign extension”

Integer Arithmetic

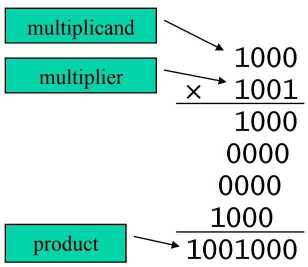
- Leverages what you learned in grammar school
- Carry adder
  - Simple carry over O(n) operations

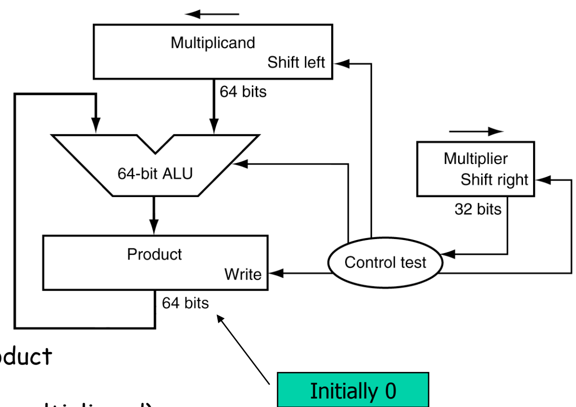


  - Predict carry
  - Guess carry and correct

Multiplication

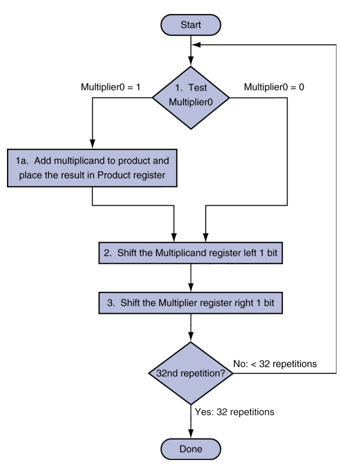
- Start with long-multiplication approach

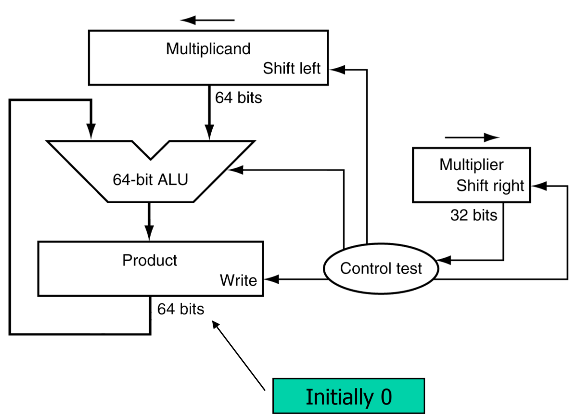




- m bits x n bits = m+n bit product
- Binary makes it easy:
  - •0 => place 0 (0 x multiplicand)
  - •1 => place a copy (1 x multiplicand)

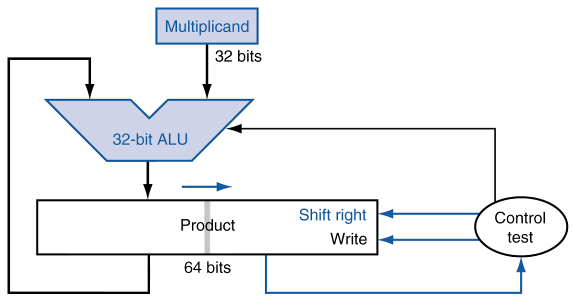
Multiplication Hardware





Optimized Multiplier

- Perform steps in parallel: add/shift

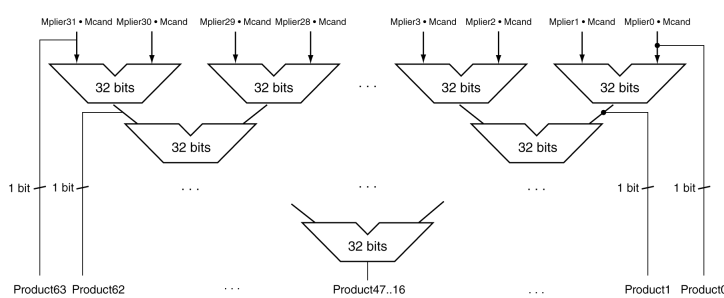


One cycle per partial-product addition

That's ok, if frequency of multiplications is low

Faster Multiplier

- Use multiple adders
  - Cost/performance tradeoff



- Can be pipelined
  - Several multiplications performed in parallel

Floating Point

- Representation for non-integral numbers
  - Types `float` and `double` in C
  - 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
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- IEEE Standard 754-1985
  - Developed in response to divergence of representations
  - Made scientific codes portable

IEEE Floating-Point Format

S	Exponent (single: 8 bits, double: 11 bits)	Fraction (single: 23 bits, double: 52 bits)

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

- 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

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    ⇒ 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: 11111111110
    ⇒ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 ⇒ significand ≈ 2.0
    $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Grammar School 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}

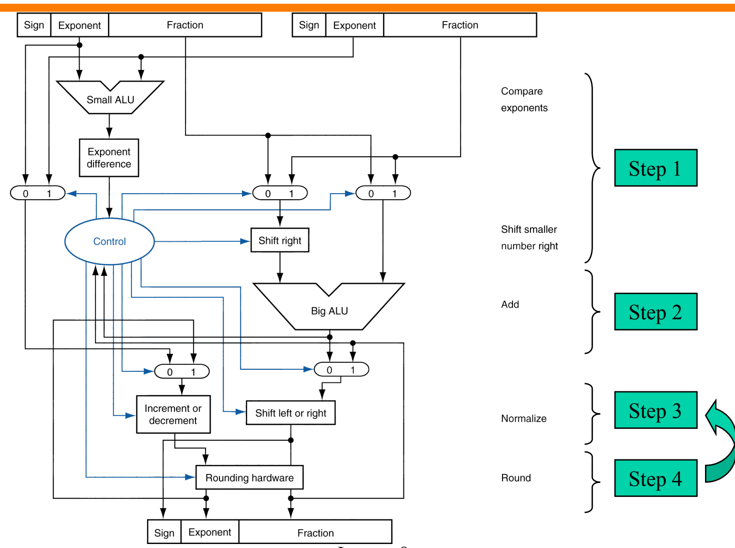
Computer Floating-Point Addition

- 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 Hardware

- Much more complex than integer operations
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP operations usually take several to many cycles
  - Can be pipelined

FP Adder Hardware



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

Summary

- Computer Numbers & Arithmetic
  - Computers have finite resources, but real numbers are infinite
  - Use 2's complement & IEEE 754 FP conventions to standardized meaning of math on computers
  - Optimize data path of add, multiply, and divide to reduce critical path by performing operations in parallel
  - Remember: bits have no inherent meaning!
- Next Time
  - Homework #3 is due 2/16
  - Exam review bring questions to class!
- Reading: review Chapters 1-3
  - No quizzes next week
  - In class open book, open note test 2/18