

Decimal Division

- Remember 4th grade long division?

$$\begin{array}{r}
 43 \quad // \text{ quotient} \\
 12 \overline{)521} \quad // \text{ divisor } \sqrt{\text{dividend}} \\
 \underline{-480} \\
 41 \\
 \underline{-36} \\
 5 \quad // \text{ remainder}
 \end{array}$$

- Shift divisor left (multiply by 10) until MSB lines up with dividend's
- Repeat until remaining dividend (remainder) < divisor
 - Find largest single digit q such that (q*divisor) < dividend
 - Set LSB of quotient to q
 - Subtract (q*divisor) from dividend
 - Shift quotient left by one digit (multiply by 10)
 - Shift divisor right by one digit (divide by 10)

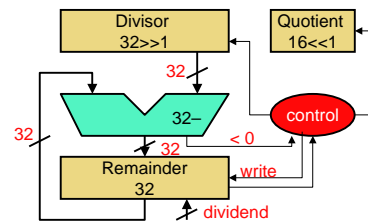
Binary Division

$$\begin{array}{r}
 12 \sqrt{521} = 01100 \sqrt{01000001001} = 43 \\
 \underline{-384} = \quad \quad \quad \underline{-0110000000} \\
 137 = \quad \quad \quad \quad \quad \underline{010001001} \\
 \underline{-0} = \quad \quad \quad \quad \quad \quad \quad \underline{0} \\
 137 = \quad \quad \quad \quad \quad \quad \quad \underline{010001001} \\
 \underline{-96} = \quad \quad \quad \quad \quad \quad \underline{-01100000} \\
 41 = \quad \quad \quad \quad \quad \quad \quad \underline{0101001} \\
 \underline{-0} = \quad \quad \quad \quad \quad \quad \quad \quad \underline{0} \\
 41 = \quad \quad \quad \quad \quad \quad \quad \quad \underline{0101001} \\
 \underline{-24} = \quad \quad \quad \quad \quad \quad \quad \underline{-011000} \\
 17 = \quad \quad \quad \quad \quad \quad \quad \quad \underline{010001} \\
 \underline{-12} = \quad \quad \quad \quad \quad \quad \quad \underline{-01100} \\
 5 = \quad \quad \quad \quad \quad \quad \quad \quad \quad \underline{101}
 \end{array}$$

Hardware for Binary Division

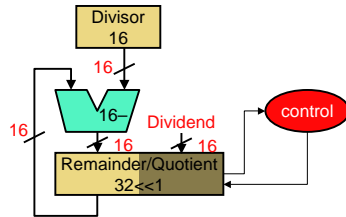
- Same as decimal division, except (again)
 - More individual steps (base is smaller)
 - Each step is simpler
- Find largest bit q such that (q*divisor) < dividend
 - q = 0 or 1
- Subtract (q*divisor) from dividend
 - q = 0 or 1 → no actual multiplication, subtract divisor or not
- One complication: **largest** q such that (q*divisor) < dividend
 - How to know if (1*divisor) < dividend?
 - Human (e.g., ECE 152 student) can eyeball this
 - Computer cannot
 - Subtract and see if result is negative

Simple 16-bit Divider Circuit



- Control algorithm:** repeat 17 times (N+1 iterations)
 - Subtract Divisor from Remainder (Remainder initially = Dividend)
 - Result >= 0? Remainder <= Result, write 1 into Quotient LSB
 - Result < 0? Just write 0 into quotient LSB
 - Shift divisor right 1 bit, shift quotient left 1 bit

Even Better Divider Circuit



- Multiplier circuit optimizations also work for divider
 - Shift Remainder left and do 16-bit subtractions
 - Combine Quotient with right (unused) half of Remainder
 - Booth and modified Booth analogs (but really nasty)
- Multiplication and division in one circuit (how?)

© 2008 Daniel J. Sorin
from Roth and Lebeck

ECE 152

61

Summary of Integer Arithmetic and ALU

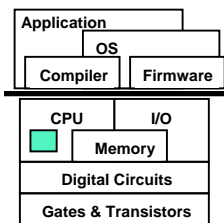
- Addition
 - Half adder full adder, ripple carry
 - Fast addition: carry select and carry lookahead
- Subtraction as addition
- Barrel shifter and shift registers
- Multiplication
 - N-step multiplication (3 refined implementations)
 - Booth's algorithm and N/2-step multiplication
- Division

© 2008 Daniel J. Sorin
from Roth and Lebeck

ECE 152

62

This Unit: Arithmetic and ALU Design



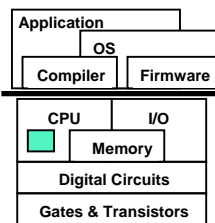
- Integer Arithmetic and ALU
 - Binary number representations
 - Addition and subtraction
 - The integer ALU
 - Shifting and rotating
 - Multiplication
 - Division
- Floating Point Arithmetic
 - Binary number representations
 - FP arithmetic
 - Accuracy

© 2008 Daniel J. Sorin
from Roth and Lebeck

ECE 152

63

Floating Point Arithmetic



- Formats
 - Precision and range
 - IEEE 754 standard
- Operations
 - Addition and subtraction
 - Multiplication and division
- Error analysis
 - Error and bias
 - Rounding and truncation
- Only scientists care?

© 2008 Daniel J. Sorin
from Roth and Lebeck

ECE 152

64

Floating Point (FP) Numbers

- **Floating point numbers**: numbers in scientific notation
 - Two uses
- Use I: real numbers (numbers with non-zero fractions)
 - 3.1415926...
 - 2.1878...
 - 9.8
 - $6.62 * 10^{-34}$
 - 5.875
- Use II: really big numbers
 - $3.0 * 10^8$
 - $6.02 * 10^{23}$

The World Before Floating Point

- Early computers were built for scientific calculations
 - ENIAC: ballistic firing tables
- But didn't have primitive floating point data types
 - Circuits were big
 - Many accuracy problems
- Programmers built **scale factors** into programs
 - Large constant multiplier turns all FP numbers to integers
 - Before program starts, inputs multiplied by scale factor **manually**
 - After program finishes, outputs divided by scale factor **manually**
 - **Yuck!**

The Fixed Width Dilemma

- "Natural" arithmetic has infinite width
 - Infinite number of integers
 - Infinite number of reals
 - Infinitely more reals than integers (head... spinning...)
- Hardware arithmetic has finite width N (e.g., 16, 32, 64)
 - Can represent 2^N numbers
- If you could represent 2^N integers, which would they be?
 - Easy, the 2^{N-1} on either side of 0
- If you could represent 2^N reals, which would they be?
 - 2^N reals from 0 to 1, not too useful
 - 2^N powers of two (1, 2, 4, 8, ...), also not too useful
 - Something in between: yes, but what?

Range and Precision

- **Range**
 - Distance between largest and smallest representable numbers
 - Want big range
- **Precision**
 - Distance between two consecutive representable numbers
 - Want small precision
- In fixed bit width, can't have unlimited both

Scientific Notation

- **Scientific notation**: good compromise
 - Number $[S,F,E] = S * F * 2^E$
 - S: **sign**
 - F: **significand** (fraction)
 - E: **exponent**
 - **"Floating point"**: binary (decimal) point has different magnitude
- + "Sliding window" of precision using notion of **significant digits**
 - Small numbers very precise, many places after decimal point
 - Big numbers are much less so, not all integers representable
 - But for those instances you don't really care anyway
- Caveat: most representations are just approximations
 - Sometimes weirdos like 0.9999999 or 1.0000001 come up
- + But good enough for most purposes

IEEE 754 Standard Precision/Range

- **Single precision**: **float** in C
 - 32-bit: 1-bit sign + 8-bit exponent + 23-bit significand
 - Range: $2.0 * 10^{-38} < N < 2.0 * 10^{38}$
 - Precision: ~7 significant (decimal) digits
- **Double precision**: **double** in C
 - 64-bit: 1-bit sign + 11-bit exponent + 52-bit significand
 - Range: $2.0 * 10^{-308} < N < 2.0 * 10^{308}$
 - Precision: ~15 significant (decimal) digits
- Numbers $> 10^{308}$ don't come up in many calculations
 - 10^{80} ~ number of atoms in universe

How Do Bits Represent Fractions?

- **Sign**: 0 or 1 → easy
- **Exponent**: signed integer → also easy
- **Significand**: unsigned fraction → not obvious!
- How do we represent integers?
 - Sums of positive powers of two
 - S-bit unsigned integer A: $A_{S-1}2^{S-1} + A_{S-2}2^{S-2} + \dots + A_12^1 + A_02^0$
- So how can we represent fractions?
 - Sums of **negative powers of two**
 - S-bit unsigned fraction A: $A_{S-1}2^0 + A_{S-2}2^{-1} + \dots + A_12^{-S+2} + A_02^{-S+1}$
 - More significant bits correspond to larger multipliers

Some Examples

- What is 5 in floating point?
 - Sign: 0
 - $5 = 1.25 * 2^2$
 - Significand: $1.25 = 1*2^0 + 1*2^{-2} = 101\ 0000\ 0000\ 0000\ 0000$
 - Exponent: $2 = 0000\ 0010$
- What is -0.5 in floating point?
 - Sign: 1
 - $0.5 = 0.5 * 2^0$
 - Significand: $0.5 = 1*2^{-1} = 010\ 0000\ 0000\ 0000\ 0000$
 - Exponent: $0 = 0000\ 0000$

Normalized Numbers

- Notice
 - 5 is $1.25 * 2^2$
 - But isn't it also $0.625 * 2^3$ and $0.3125 * 2^4$ and ...?
 - With 8-bit exponent, we can have 8 representations of 5
- Multiple representations for one number
 - Would lead to computational errors
 - Would waste bits
- Solution: choose **normal (canonical) form**
 - Disallow de-normalized numbers
 - IEEE 754 normal form: coefficient of 2^0 is always 1
 - Similar to scientific notation: one non-zero digit left of decimal
 - Normalized representation of 5 is $1.25 * 2^2$ ($1.25 = 1 * 2^0 + 1 * 2^{-2}$)
 - $0.625 * 2^3$ is de-normalized ($0.625 = 0 * 2^0 + 1 * 2^{-1} + 1 * 2^{-3}$)

More About Normalization

- What is -0.5 in **normalized** floating point?
 - Sign: 1
 - $0.5 = 1 * 2^{-1}$
 - Significand: $1 = 1 * 2^0 = 100\ 0000\ 0000\ 0000\ 0000\ 0000$
 - Exponent: $-1 = 1111\ 1111$ (assuming 2's complement for now)
- IEEE 754: no need to represent coefficient of 2^0 explicitly
 - It's always 1
 - + Buy yourself an extra bit of precision
 - Pretty cute trick
- Problem: what about 0?
 - How can we represent 0 if 2^0 is always implicitly 1?

IEEE 754: The Whole Story

- **Exponent**: signed integer → not so fast
- Exponent represented in **excess** or **bias** notation
 - N-bits typically can represent signed numbers from -2^{N-1} to $2^{N-1}-1$
 - But in IEEE 754, they represent exponents from $-2^{N-1} + 2$ to $2^{N-1}-1$
 - And they represent those as unsigned with an implicit $2^{N-1}-1$ added
 - Implicit added quantity is called the **bias**
 - Actual exponent is $E - (2^{N-1}-1)$
- Example: single precision (8-bit exponent)
 - Bias is 127, exponent range is -126 to 127
 - -126 is represented as $1 = 0000\ 0001$
 - 127 is represented as $254 = 1111\ 1110$
 - 0 is represented as $127 = 0111\ 1111$
 - 1 is represented as $128 = 1000\ 0000$

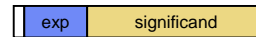
IEEE 754: Continued

- Notice: two exponent bit patterns are "unused"
- **0000 0000**: represents de-normalized numbers
 - Numbers that have implicit 0 (rather than 1) in 2^0
 - Zero is a special kind of de-normalized number
 - + Exponent is all 0s, significand is all 0s
 - There are both $+0$ and -0 , but they are considered the same
 - Also represent numbers smaller than smallest normalized numbers
- **1111 1111**: represents infinity and NaN
 - \pm infinities have 0s in the significand
 - \pm NaNs do not

IEEE 754: To Infinity and Beyond

- What are infinity and NaN used for?
 - To allow operations to proceed past overflow/underflow situations
 - **Overflow**: operation yields exponent greater than $2^{N-1}-1$
 - **Underflow**: operation yields exponent less than $-2^{N-1}+2$
- IEEE 754 defines operations on infinity and NaN
 - $N / 0 = \text{infinity}$
 - $N / \text{infinity} = 0$
 - $0 / 0 = \text{NaN}$
 - $\text{Infinity} / \text{infinity} = \text{NaN}$
 - $\text{Infinity} - \text{infinity} = \text{NaN}$
 - Anything and NaN = NaN

IEEE 754: Final Format



- Biased exponent
- Normalized significand
- Exponent uses more significant bits than significand
 - Helps when comparing FP numbers
 - Exponent bias notation helps there too – why?
- Every computer since about 1980 supports this standard
 - Makes code portable (at the source level at least)
 - Makes hardware faster (stand on each other's shoulders)

Floating Point Arithmetic

- We will look at
 - Addition/subtraction
 - Multiplication/division
- Implementation
 - Basically, integer arithmetic on significand and exponent
 - Using integer ALUs
 - Plus extra hardware for normalization
- To help us here, look at toy "quarter" precision format
 - 8 bits: 1-bit sign + 3-bit exponent + 4-bit significand
 - Bias is 3 ($= 2^{N-1} - 1$)