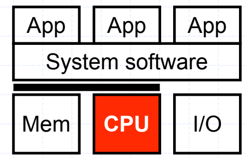
CIS 371
Computer Organization and Design

Unit 7: Floating Point

This Unit: 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

Readings

- P+H
  - Chapter 3.6 and 3.7

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…
  - $6.62 * 10^{-34}$
- Use II: really big numbers
  - $3.0 * 10^{8}$
  - $6.02 * 10^{23}$
- Aside: best not used for currency values

The Land Before Floating Point

- Early computers were built for scientific calculations
  - ENIAC: ballistic firing tables
  - …But didn't have primitive floating point data types
- Many embedded chips today lack floating point hardware
- Programmers built **scale factors** into programs
  - Large constant multiplier turns all FP numbers to integers
  - inputs multiplied by scale factor **manually**
  - Outputs divided by scale factor **manually**
- Sometimes called **fixed point arithmetic**

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 size of 0
- If you could represent 2^N reals, which would they be?
  - 2^N reals from 0 to 1, not too useful
  - Uhh…. umm…

Range and Precision

- **Range**
  - Distance between largest and smallest representable numbers
  - Want big
- **Precision**
  - Distance between two consecutive representable numbers
  - Want small
- In fixed 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: all representations are just approximations
    - Sometimes wierdos 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
  - Used when exact precision is less important (e.g., 3D games)
- **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
  - Used for scientific computations
- 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 → ??
- How do we represent integers?
  - Sums of positive powers of two
  - S-bit unsigned integer A: $A_{S-1}2^{S-1} + A_{S-2}2^{S-2} + \ldots + A_1 2^1 + A_0 2^0$
- So how can we represent fractions?
  - Sums of **negative powers of two**
  - S-bit unsigned fraction A: $A_{S-1}2^{0} + A_{S-2}2^{-1} + \ldots + A_1 2^{-S+2} + A_0 2^{-S+1}$
    - 1, 1/2, 1/4, 1/8, 1/16, 1/32, …
  - 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 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 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 256 representations of 5
- Multiple representations for one number
  - Lead to computational errors
  - Waste bits
- Solution: choose **normal (canonical) form**
  - Disallow de-normalized numbers
  - IEEE 754 normal form: coefficient of 2^0 is 1
    - Similar to scientific notation: one non-zero digit left of decimal
  - Normalized representation of 5 is $1.25 * 2^2$ ($1.25 = \mathbf{1*2^0}+1*2^{-2}$)
  - $0.625 * 2^3$ is de-normalized ($0.625 = \mathbf{0*2^0}+1*2^{-1}+ 1*2^{-3}$)

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
- IEEE754: no need to represent co-efficient of 2^0 explicitly
  - It's always 1
  - + Buy yourself an extra bit (~1/3 of decimal digit) of precision
    - Yeeha
- 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 (`bzero` still works)
    - – 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
  - ± infinities have 0s in the significand
  - ± NaNs do not

IEEE 754: 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 = infinity
  - N / infinity = 0
  - 0 / 0 = NaN
  - Infinity / infinity = NaN
  - Infinity – infinity = NaN
  - Anything and NaN = NaN

IEEE 754: Final Format

- Biased exponent
- Normalized significand
- Exponent more significant than significand
  - Helps comparing FP numbers
  - Exponent bias notation helps there too
- 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

FP Addition

- Assume
  - A represented as bit pattern [S_A, E_A, F_A]
  - B represented as bit pattern [S_B, E_B, F_B]
- What is the bit pattern for A+B [S_{A+B}, E_{A+B}, F_{A+B}]?
  - [S_A+S_B, E_A+E_B, F_A+F_B]? Of course not
  - So what is it then?

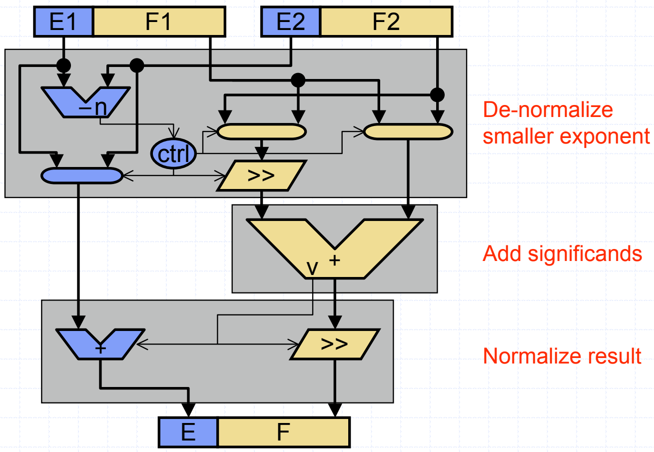
FP Addition Decimal Example

- Let's look at a decimal example first: 99.5 + 0.8
  - $9.95*10^{1} + 8.0*10^{-1}$
- Step I: **align exponents (if necessary)**
  - Temporarily de-normalize one with smaller exponent
  - Add 2 to exponent → shift significand right by 2
  - $8.0*10^{-1} \rightarrow$ **$0.08*10^{1}$**
- Step II: **add significands**
  - Remember overflow, it isn't treated like integer overflow
  - $9.95*10^{1} + 0.08*10^{1} \rightarrow 10.03*10^{1}$
- Step III: **normalize result**
  - Shift significand right by 1 add 1 to exponent
  - $10.03*10^{1} \rightarrow 1.003*10^{2}$

FP Addition Quarter Example

- Now a binary "quarter" example: 7.5 + 0.5
  - 7.5 = 1.875*2² = 0 101 11110
    - 1.875 = 1*2⁰+1*2⁻¹+1*2⁻²+1*2⁻³
  - 0.5 = 1*2⁻¹ = 0 010 10000
- Step I: **align exponents (if necessary)**
  - 0 010 10000 → 0 101 00010
  - Add 3 to exponent → shift significand right by 3
- Step II: **add significands**
  - 0 101 11110 + 0 101 00010 = 0 101 100000
- Step III: **normalize result**
  - 0 101 100000 → 0 110 10000
  - Shift significand right by 1 → add 1 to exponent

FP Addition Hardware



What About FP Subtraction?

- Or addition of negative quantities for that matter
  - How to subtract significands that are not in 2C form?
  - Can we still use an adder?
- Trick: internally and temporarily convert to 2C
  - Add "phantom" –2 in front (–1*2¹)
    - Use standard negation trick
  - Add as usual
  - If phantom –2 bit is 1, result is negative
    - Negate it using standard trick again, flip result sign bit
  - Ignore "phantom" bit which is now 0 anyway
  - Got all that?
- Basically, big ALU has negation prefix and postfix circuits

FP Multiplication

- Assume
  - A represented as bit pattern [S_A, E_A, F_A]
  - B represented as bit pattern [S_B, E_B, F_B]
- What is the bit pattern for A*B [S_{A*B}, E_{A*B}, F_{A*B}]?
  - This one is actually a little easier (conceptually) than addition
    - Scientific notation is logarithmic
    - **In logarithmic form: multiplication is addition**
- [S_A^S_B, E_A+E_B, F_A*F_B]? Pretty much, except for…
  - Normalization
  - Addition of exponents in biased notation (must subtract bias)
  - Tricky: when multiplying two normalized significands…
    - Where is the binary point?

FP Division

- Assume
  - A represented as bit pattern [S_A, E_A, F_A]
  - B represented as bit pattern [S_B, E_B, F_B]
- What is the bit pattern for A/B [$S_{A/B}$, $E_{A/B}$, $F_{A/B}$]?
- [S_A^S_B, E_A–E_B, F_A/F_B]? Pretty much, again except for…
  - Normalization
  - Subtraction of exponents in biased notation (must add bias)
  - Binary point placement
  - No need to worry about remainders, either
- Ironic
  - Multiplication/division roughly same complexity for FP and integer
  - Addition/subtraction much more complicated for FP than integer

Accuracy

- Remember our decimal addition example?
  - $9.95*10^1 + 8.00*10^{-1} \rightarrow 1.00\mathbf{3}*10^2$
  - Extra decimal place caused by de-normalization…
  - But what if our representation only has two digits of precision?
    - What happens to the **3**?
  - Corresponding binary question: what happens to extra 1s?
- Solution: **round**
  - Option I: **round down (truncate)**, no hardware necessary
  - Option II: **round up (round)**, need an incrementer
    - Why rounding up called round?
    - Because an extra 1 is half-way, which "rounded" up

More About Accuracy

- Problem with both truncation and rounding
  - They cause errors to **accumulate**
  - E.g., if always round up, result will gradually "crawl" upwards
- One solution: **round to nearest even**
  - If un-rounded LSB is 1 → round up (01**1** → 10)
  - If un-rounded LSB is 0 → round down (00**1** → 00)
  - Round up half the time, down other half → overall error is stable
- Another solution: **multiple intermediate precision bits**
  - IEEE 754 defines 3: guard + round + sticky
    - Guard and round are shifted by de-normalization as usual
    - Sticky is 1 if any shifted out bits are 1
  - Round up if 101 or higher, round down if 011 or lower
  - Round to nearest even if 100

Numerical Analysis

- Accuracy problems sometimes get bad
  - Addition of big and small numbers
    - Summing many small numbers
  - Subtraction of big numbers
  - Example, what's $1*10^{30} + 1*10^0 - 1*10^{30}$?
    - Intuitively: $1*10^0 = 1$
    - But: $(1*10^{30} + 1*10^0) - 1*10^{30} = (1*10^{30} - 1*10^{30}) = 0$
- **Numerical analysis**: field formed around this problem
  - Bounding error of numerical algorithms
  - Re-formulating algorithms in a way that bounds numerical error
- Practical hints: never test for equality between FP numbers
  - Use something like: if(abs(a-b) < 0.00001) then …

One Last Thing About Accuracy

- Suppose you added two numbers and came up with
  - 0 101 **1**1111 **101**
  - What happens when you round?
  - Number becomes denormalized… arrrrgggghhh
- FP adder actually has more than three steps…
  - Align exponents
  - Add/subtract significands
  - Re-normalize
  - **Round**
  - **Potentially re-normalize again**
  - **Potentially round again**

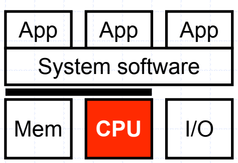
Arithmetic Latencies

- Latency in cycles of common arithmetic operations
- Source: *Software Optimization Guide for AMD Family 10h Processors, Dec 2007*
  - Intel "Core 2" chips similar

	Int 32	Int 64	Fp 32	Fp 64
Add/Subtract	1	1	4	4
Multiply	3	5	4	4
Divide	14 to 40	23 to 87	16	20

- Floating point divide faster than integer divide?
  - Why?

Summary



- FP representation
  - Scientific notation: $S*F*2^E$
  - IEEE754 standard
  - Representing fractions
- FP operations
  - Addition/subtraction: harder than integer
  - Multiplication/division: same as integer!!
- Accuracy problems
  - Rounding and truncation
- Upshot: FP is painful
  - Thank lucky stars P37X has no FP