Lecture 6: Arithmetic

COS / ELE 375

Computer Architecture and Organization

Princeton University
Fall 2015

Prof. David August

Multiplication

Computing Exact Product of w-bit numbers x, y

- Need 2w bits

Unsigned: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$

Two's Complement:

min: $x * y \ge (-2^{w-1})(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$

max: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$

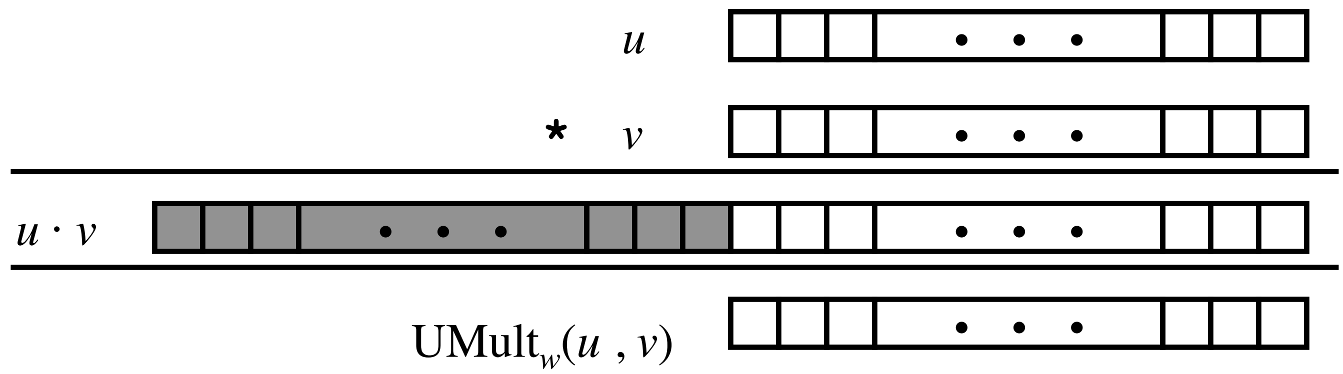
- Maintaining Exact Results
  - Need unbounded representation size
  - Done in software by *arbitrary precision* arithmetic packages
  - Also implemented in Lisp, ML, and other languages

Unsigned Multiplication in C

Operands: *w* bits

True Product: 2*w bits

Discard *w* bits: *w* bits



- Standard Multiplication Function
  - Ignores high order w bits
- Implements Modular Arithmetic
  - $\text{UMult}_w(u, v) = u \cdot v \bmod 2^w$

Unsigned Multiplication

<u>Binary makes it easy:</u>

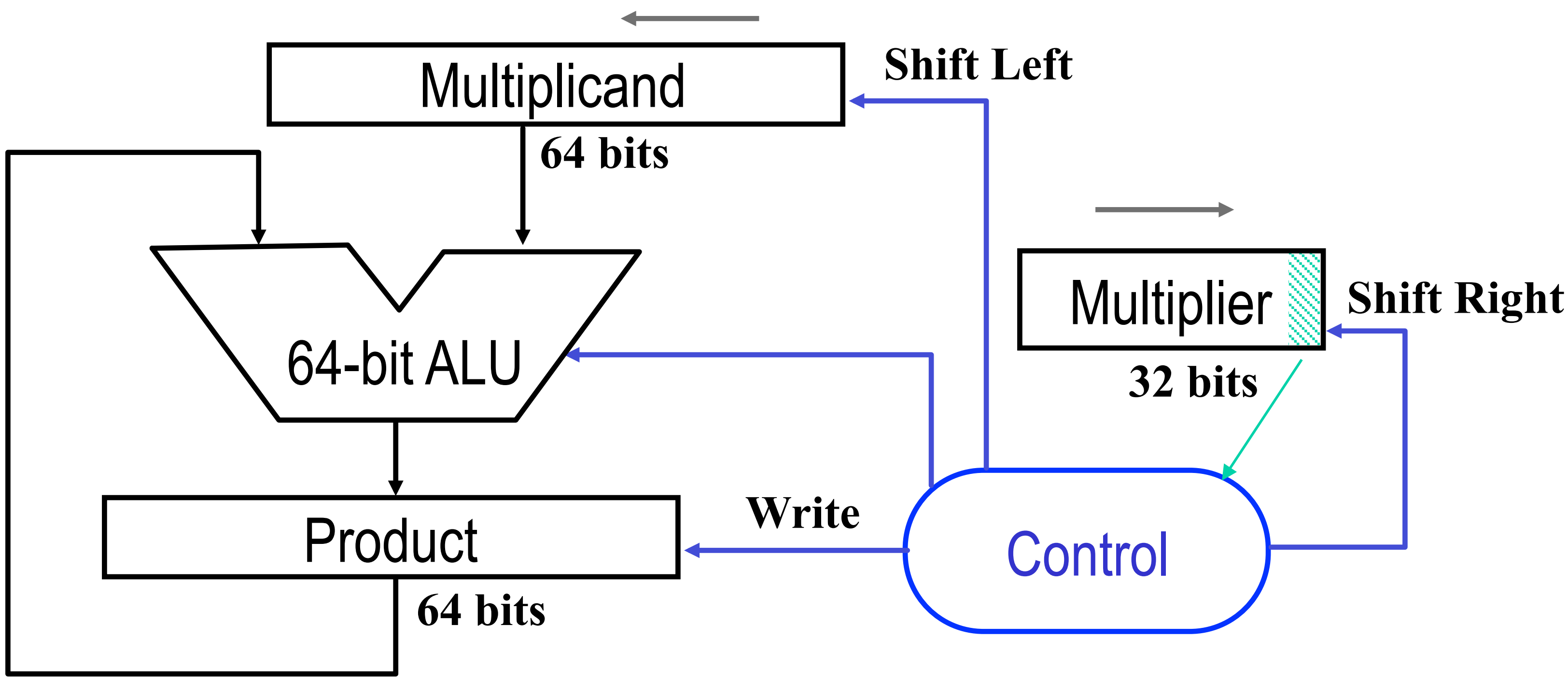
- 0 => place 0 (0 x multiplicand)
- 1 => place a copy (1 x multiplicand)

<u>Key sub-parts:</u>

- Place a copy or not
- Shift copies appropriately
- Final addition

Unsigned Shift-Add Multiplier (Version 1)

Straightforward approach:



Algorithm (Version 1)

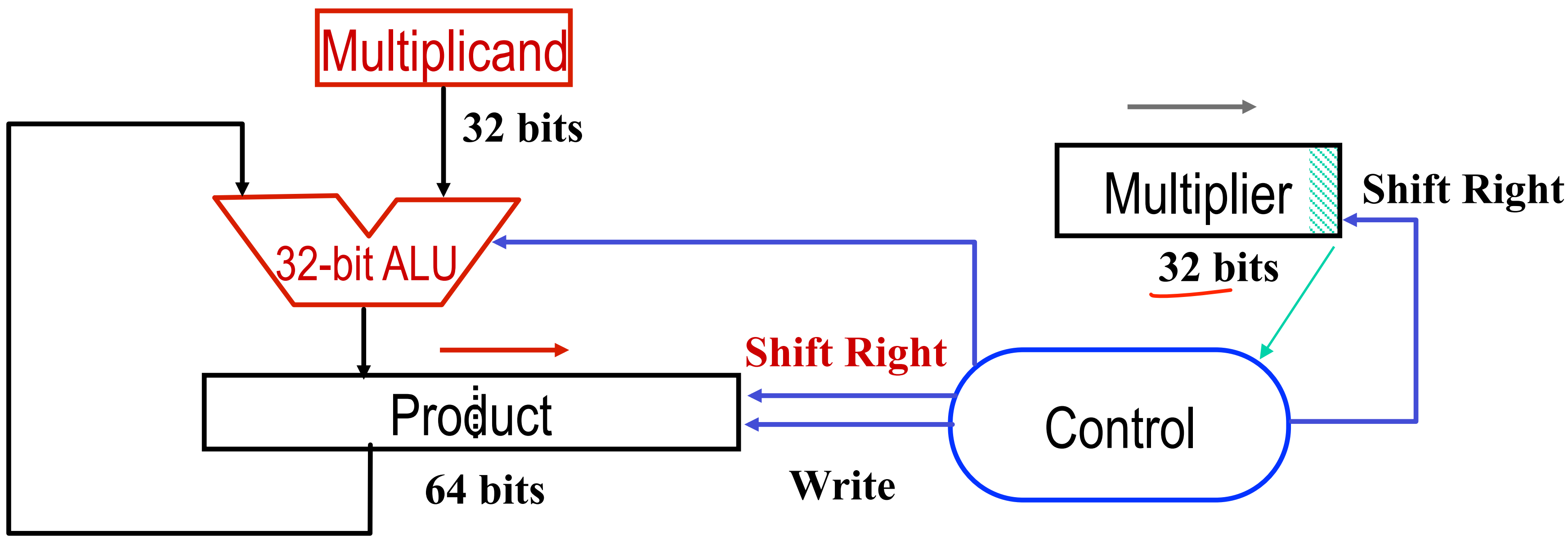
```
for (i = 0; i < 32; i++) {
  if(MULTIPLIER[0] == 1)
     PRODUCT = PRODUCT + MULTIPLICAND;
  MULTIPLICAND << 1;
  MULTIPLIER >> 1;
}
```

Unsigned Multiplier (Version 2)

Observation: Half of bits in the Multiplicand were always 0

Improvement: Use a 32-bit ALU (faster than a 64-bit ALU)

Shift product right instead of shifting multiplicand



Algorithm (Version 2)

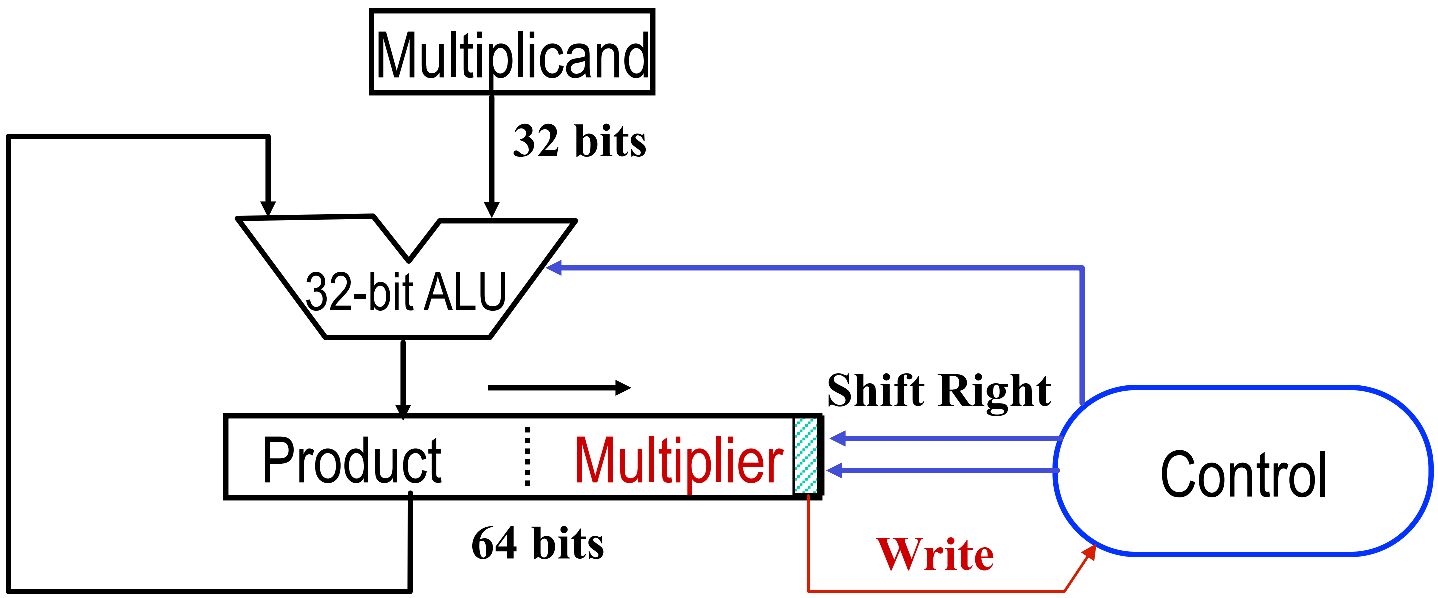
```
for (i = 0; i < 32; i++) {
  if(MULTIPLIER[0] == 1)
      PRODUCT[63:32] += MULTIPLICAND;
  PRODUCT >> 1;
  MULTIPLIER >> 1;
}
```

Unsigned Multiplier (Final Version)

Observation: Multiplier loses bits as Product gains them

Improvement: Share the same 64-bit register

Multiplier is placed in Product register at start



Algorithm (Final Version)

```
PRODUCT[31:0] = MULTIPLIER;
for (i = 0; i < 32; i++) {
  if(PRODUCT[0] == 1)
     PRODUCT[63:32] += MULTIPLICAND;
  PRODUCT >> 1;
}
```

Signed Multiplication

Solution 1:

Compute multiplication using magnitude, compute product sign separately

Solution 2:

Same HW as unsigned multiplier except sign extend while shifting to maintain sign

Solution 3:

A potentially faster way: Booth's Algorithm…

Andrew D. Booth

- During WWII: X-ray crystallographer for British Rubber Producers Research Association
- Developed a calculating machine to help analyze raw data

- 1947: At Princeton under John von Neumann at IAS
- Back in Britain: Developed Automatic Relay Computer with Magnetic Drum

Booth's Algorithm Key Idea

Look for strings of 1's:

$2 \times 30 = 00010_2 \times 011110_2$

$30 = -2 + 32$

$011110 = -\ 000010 + 100000$

To multiply:

- Add 000010 four times (w/ shifts)

  - OR -

- Add 100000 once and subtract 000010 once (w/ shifts)

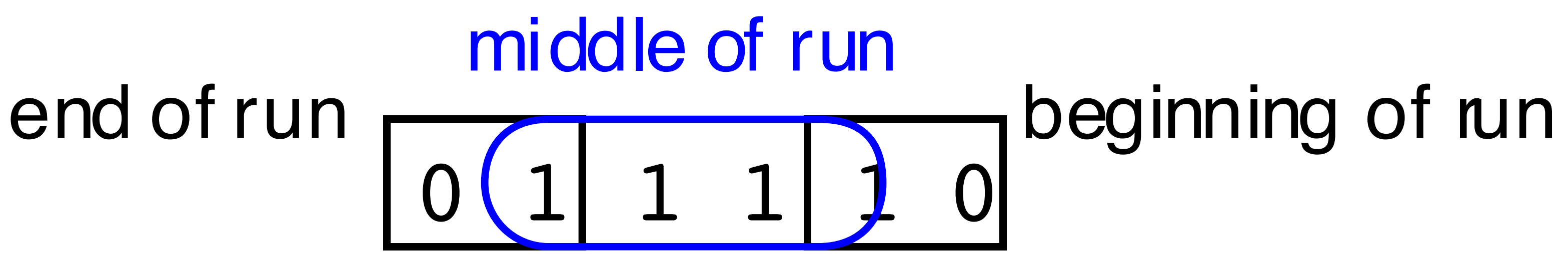
When is this faster?

Booth's Algorithm

To multiply:

Each string of 1s: subtract at start of run, add after end

Current Bit	Bit to the Right	Explanation	Example	Operation
1	0	Start of 1s	00110	sub (00010)
1	1	Middle of 1s	00110	none
0	1	End of 1s	00110	add (01000)
0	0	Middle of 0s	00110	none



Multiplication: Summary

- Lots more hardware than addition/subtraction
- Large column additions “final add” are big delay if implemented in naïve ways → Add at each step
- Observe and optimize adding of zeros, use of space
- Booth’s algorithm deals with signed and may be faster

- Lots of other efforts made in speeding multiplication up
  - Consider multiplication by powers of 2
  - Special case small integers

“Float” by Frank Ortmanns

Representations

What can be represented in N bits?

Unsigned: 0 → 2^n-1

Signed: -2^{n-1} → 2^{n-1} - 1

<u>What about:</u>

Very large numbers?	9,349,787,762,244,859,087,678
Very small numbers?	0.0000000000000000000004691
Rationals?	2/3
Irrationals?	SQRT(2)
Transcendentals?	e, PI

Pattern Assignments

Bit Pattern	Method 1	Method 2	Method 3
000	0	0	0
001	1	1	0.1
010	e	2	0.2
011	pi	4	0.3
100	4	8	0.4
101	-pi	16	0.5
110	-e	32	0.6
111	-1	64	0.7

What should we do? Another method?

The Binary Point

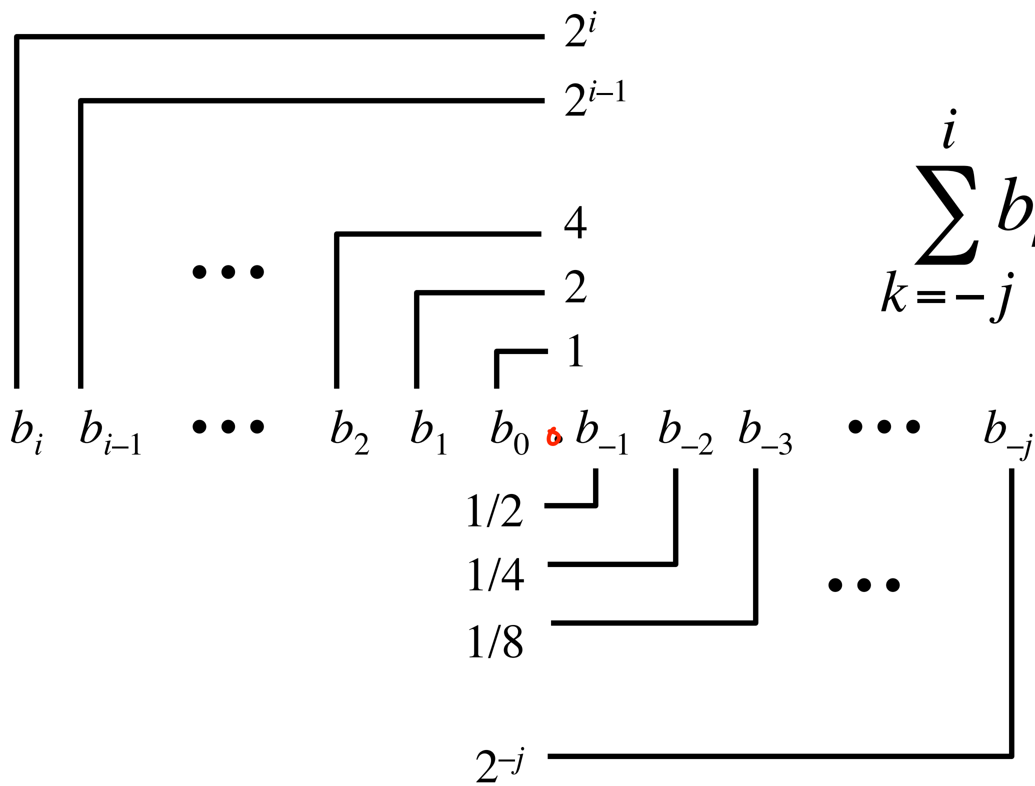
$$101.11_2 = 4 + 1 + ½ + ¼ = 5.75$$

Observations:

- Divide by 2 by shifting point left
- $0.111111\ldots_2$ is just below 1.0
- Some numbers cannot be exactly represented well

  $1/10 \rightarrow 0.0001100110011[0011]^*\ldots_2$

Obvious Approach: Fixed Point



$$\sum_{k=-j}^{i} b_k \cdot 2^k$$

Fixed Point

In w-bits (w = i + j):

- use i-bits for left of binary point
- use j-bits for right of binary point

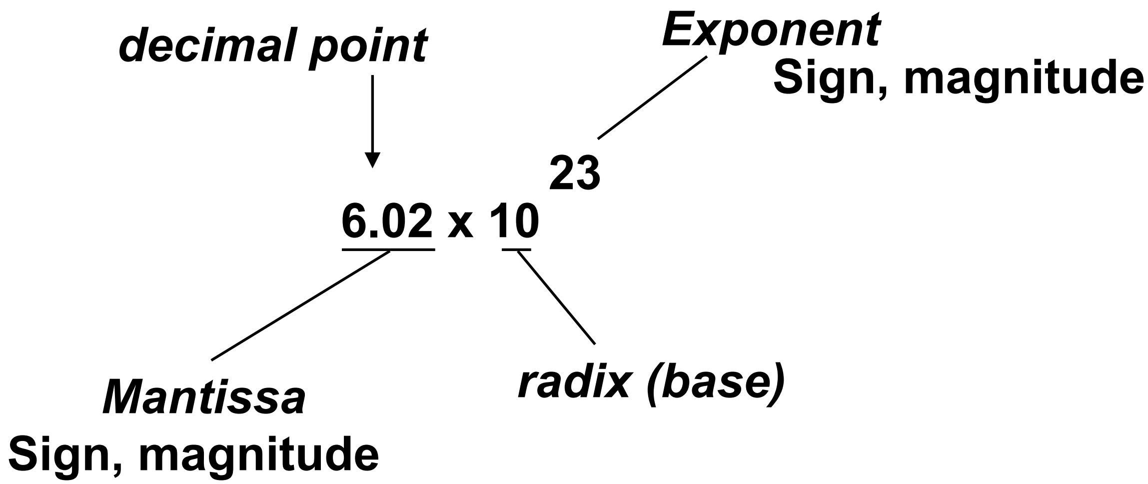
Qualities:

- Easy to understand
- Arithmetic relatively easy to implement...
- Precision and Magnitude:

    16-bits, i=j=8: 0 → 255.99609375

    Step size: 0.00390625

Another Approach: Scientific Notation



- In Binary:

$$\text{radix} = 2$$

$$\text{value} = (-1)^{s} \times M \times 2^{E}$$

- How is this better than fixed point?

IEEE Floating Point

IEEE Standard 754

- Established in 1980 as uniform standard for floating point arithmetic
- Supported by all major CPUs
- In 99.999% of all machines used today

Driven by Numerical Concerns

- Standards for rounding, overflow, underflow
- Primarily numerical analysts rather than hardware types defined standard

This is where it gets a little involved…

IEEE 754 Floating Point Standard

- Single precision: 8 bit exponent, 23 bit significand
- Double precision: 11 bit exponent, 52 bit significand

- Significand *M* normally in range [1.0,2.0) → Imply 1
- Exponent *E* biased exponent → B is bias ($B = 2^{N-1} - 1$)

$$N = (-1)^s \times 1.M \times 2^{E - B}$$

s	E	M

- Bias allows integer comparison (almost)!

  0000…0000 is most negative exponent

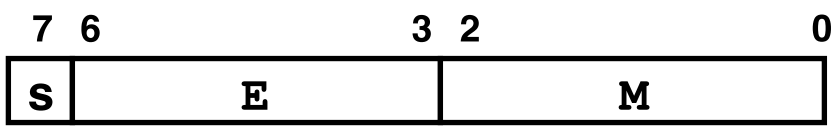
  1111…1111 is most positive exponent

IEEE 754 Floating Point Example

Define Wimpy Precision as:

1 sign bit, 4 bit exponent, 3 bit significand, B = 7

Represent: -0.75



IEEE 754 Floating Point Special Exponents

There’s more!

Normalized: E ≠ 000…0 and E ≠ 111…1

- Recall the implied 1.*xxxxx*

Special Values: E = 111…1

- M = 000…0:
  - Represents +/- ∞ (infinity)
  - Used in overflow
  - Examples: 1.0/0.0 = +∞,   1.0/−0.0 = −∞
  - Further computations with infinity possible
  - Example: X/0 > Y may be a valid comparison

IEEE 754 Floating Point Special Exponents

Normalized: E ≠ 000…0 and E ≠ 111…1

Special Values: E = 111…1

- M ≠ 000…0:
  - Not-a-Number (NaN)
  - Represents invalid numeric value or operation
  - Not a number, but not infinity (e.q. sqrt(-4))
  - Examples: sqrt(–1), ∞ − ∞
  - NaNs propagate: f(NaN) = NaN

IEEE 754 Floating Point Special Exponents

Normalized: E ≠ 000…0 and E ≠ 111…1

- Recall the implied 1.$xxxxx$

Denormalized: E = 000…0

- M = 000…0
    - Represents value 0
    - Note the distinct values +0 and –0

IEEE 754 Floating Point Special Exponents

Normalized: E ≠ 000…0 and E ≠ 111…1

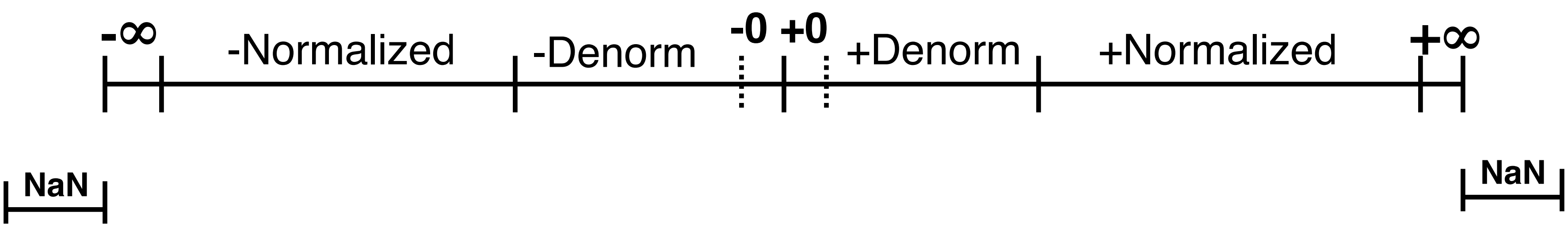
- Recall the implied 1.*xxxxx*

Denormalized: E = 000…0

- M ≠ 000…0
  - Numbers very close to 0.0
  - Lose precision as magnitude gets smaller
  - "Gradual underflow"

Exponent	$-Bias + 1$
Significand	$0.xxx...x_2$

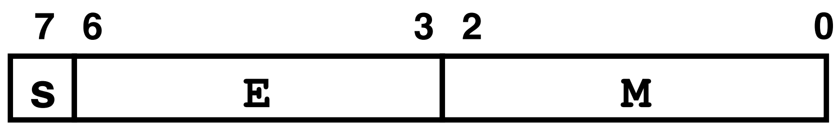
Encoding Map



Wimpy Precision

Define Wimpy Precision as:

1 sign bit, 4 bit exponent, 3 bit significand, B = 7



E = 1-14: Normalized

E = 0: Denormalized

E = 15: Infinity/ NaN

Dynamic Range

	S	E	M	exp	value	
Denormalized numbers	0	0000	000	n/a	0	
	0	0000	001	-6	1/512	← closest to zero
	0	0000	010	-6	2/512	
	…					
	0	0000	110	-6	6/512	
	0	0000	111	-6	7/512	← largest denorm
Normalized numbers	0	0001	000	-6	8/512	← smallest norm
	0	0001	001	-6	9/512	
	…					
	0	0110	110	-1	28/32	
	0	0110	111	-1	30/32	← closest to 1 below
	0	0111	000	0	1	
	0	0111	001	0	36/32	← closest to 1 above
	0	0111	010	0	40/32	
	…					
	0	1110	110	7	224	
	0	1110	111	7	240	← largest norm
	0	1111	000	n/a	inf	


USA
40043
USA
40043

Is Rounding Important?

- June 4, 1996: Ariane 5 rocket.
- Converted a 64-bit floating point to a 16-bit integer.
- The overflow wasn't handled properly.

Rounding Modes in IEEE 754

Always round to nearest, unless halfway

Round toward Zero

Round Down

Round Up

Nearest Even - Default for good reason

- Others are statistically biased
- Hard to get anything else without assembly

Rounding Binary Numbers

"Even" when least significant bit is 0

Halfway when bits to right of rounding position = $100\ldots_2$

Example: Round to nearest 1/4 (2 bits right of point)

Value	Binary	Rounded	Action	Rounded
2-3/32	10.00011_2	10.00_2	(<1/2—down)	2
2-3/16	10.00110_2	10.01_2	(>1/2—up)	2-1/4
2-7/8	10.11100_2	11.00_2	(1/2—up)	3
2-5/8	10.10100_2	10.10_2	(1/2—down)	2-1/2

IEEE 754 Rounding

"Floating Point numbers are like piles of sand; every time you move one you lose a little sand, but you pick up a little dirt."

- How many extra bits?
- IEEE Says: As if computed exactly then rounded.

- Guard and round bit - 2 extra bits used for computation

- Sticky bit - 3rd bit, set when a 1 is shifted to the right
  Indicates difference between 0.10…00 and 0.10…01

Arithmetic

Comparison:

- Nice property for 0 equality: All 0 bits means +0.
- Same as integers except
  - Compare sign bits
  - Consider +0 == -0 and NaN's

Addition:

1. Align decimal point by shifting (remember implied 1)
2. Add significands
3. Normalize significand of sum
4. Round using rounding bits

Arithmetic

<u>Multiplication:</u>

1. Add exponents - be careful of double bias!
2. Multiply significands
3. Normalize significand of product
4. Round using rounding bits
5. Compute sign of product, set sign bit

*Nobody was hurt in the making of this photograph

The FDIV (Floating Point Divide) Bug

- July 1994: Intel discovers the bug in Pentium
- Sept. 1994: Math professor (re)discovers it
- Nov. 1994: Intel says it's no biggie for non-techies
- Dec. 1994: IBM says it is, stops selling Pentium PCs
- Dec. 1994: Intel apologizes, offers recall

- Recall cost roughly $300M dollars
- Fix in July 1994 would have cost $300K dollars

- April 1997: Intel finds, announces, fixes another floating point bug

What was the FDIV Bug?

- Floating point DIVide
- Uses a lookup table to guess next 2 bits of quotient
- Table had bad values

Enrichment: Devise such a scheme from what is available in the book and your knowledge of algebra.

At Intel, quality is job 0.999999998.

Q: How many Pentium designers does it take to screw in a light bulb?

A: 1.99995827903, but that's close enough for nontechnical people.

This lecture was brought to you by Apple.

The Importance of Standards

For over 20 years, everyone has been using a standard that took scientists and engineers years to perfect.

The IEEE 754 standard is more ubiquitous than just about anything out there.

In defining Java, Sun ignored it…

How Java's Floating-Point Hurts Everyone Everywhere
by W. Kahan and J. Darcy
http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf

(since been fixed)

Summary

- Phew! We made it through Arithmetic!
- Datapath and Control next time!!