Computer Organization and Design

More Arithmetic: Multiplication, Division & Floating-Point

Henry Fuchs

Slides adapted from Montek Singh, who adapted them
from Leonard McMillan and from Gary Bishop
Back to McMillan & Chris Terman, MIT 6.004 1999

Tuesday, March 31, 2015

Lecture 12

Topics

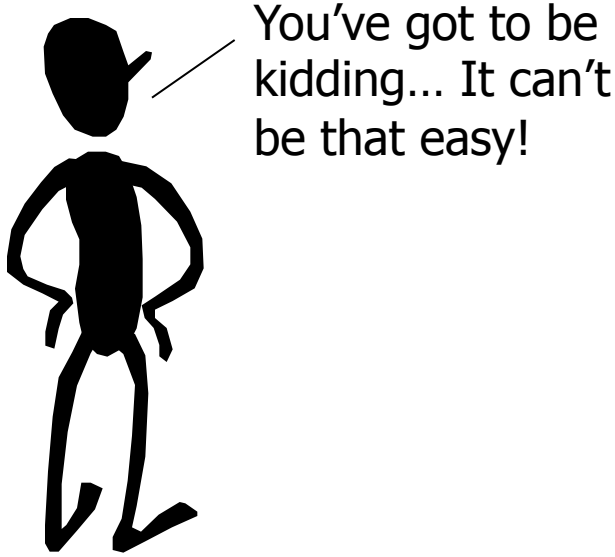
- Brief overview of:
  - integer multiplication
  - integer division
  - floating-point numbers and operations

Binary Multipliers

The key trick of multiplication is memorizing a digit-to-digit table… Everything else is just adding

×	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

×	**0**	**1**
0	**0**	**0**
1	**0**	**1**



Reading: Study Chapter 3.1-3.4

Binary Multiplication

The "Binary" Multiplication Table

X	0	1
0	0	0
1	0	1

Binary multiplication is implemented using the same basic longhand algorithm that you learned in grade school.

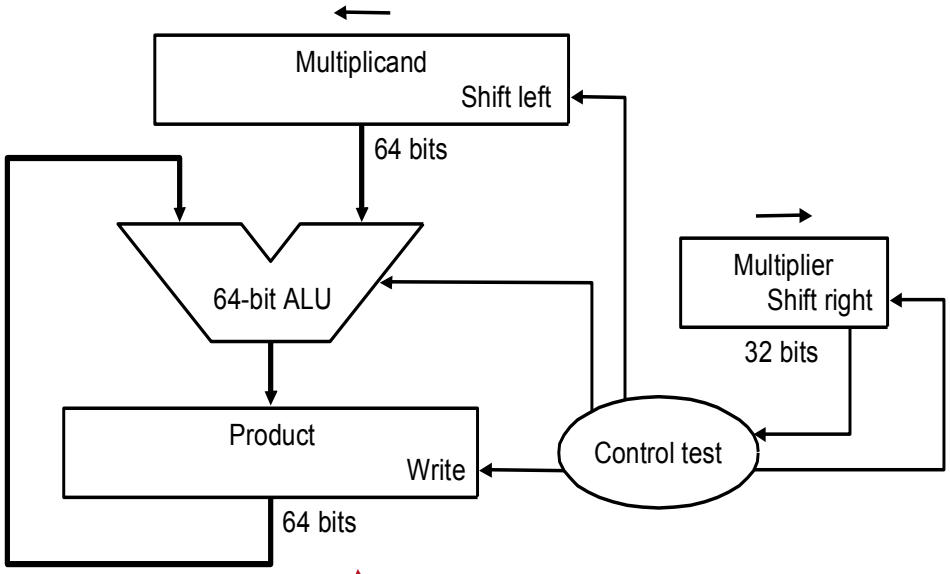
$$\begin{array}{rcccc} & A_3 & A_2 & A_1 & A_0 \\ \times & B_3 & B_2 & B_1 & B_0 \\ \hline & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\ & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 \\ & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 \\ + & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 \\ \hline \end{array}$$

A_jB_i is a "partial product"

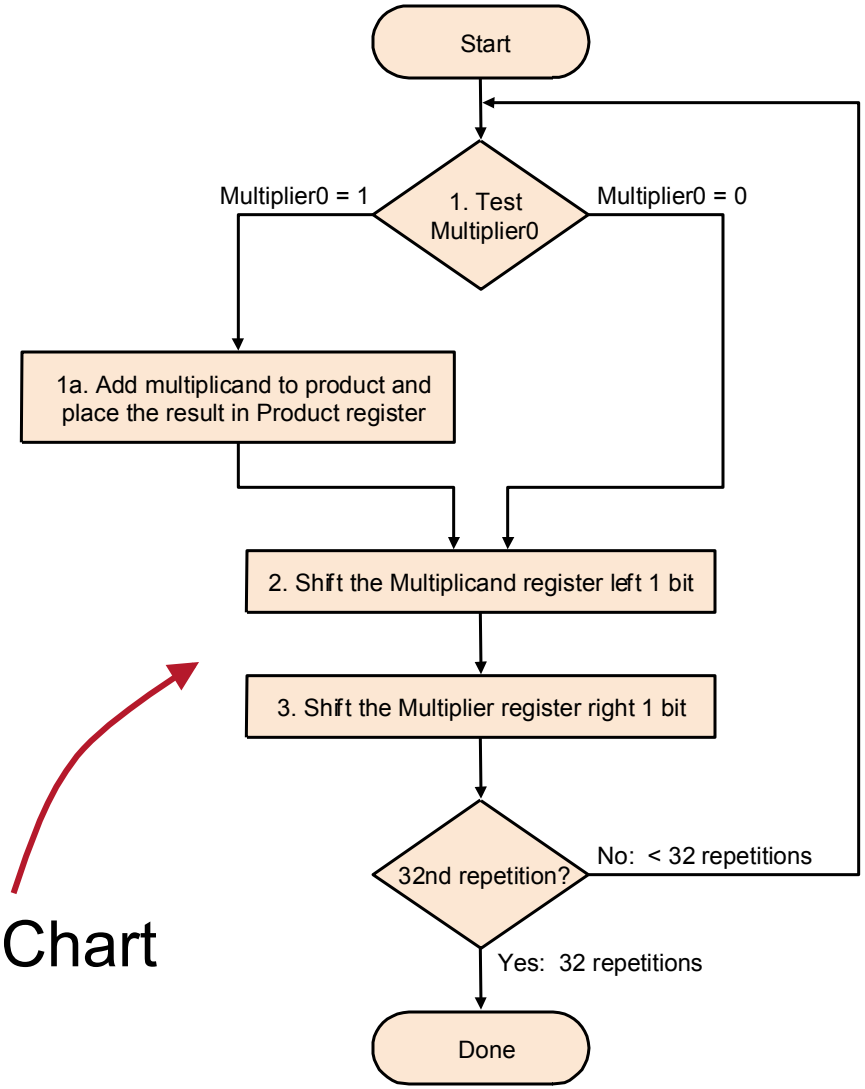
Multiplying N-digit number by M-digit number gives (N+M)-digit result

Easy part: forming partial products (just an AND gate since B_I is either 0 or 1)
Hard part: adding M, N-bit partial products

Multiplication: Implementation

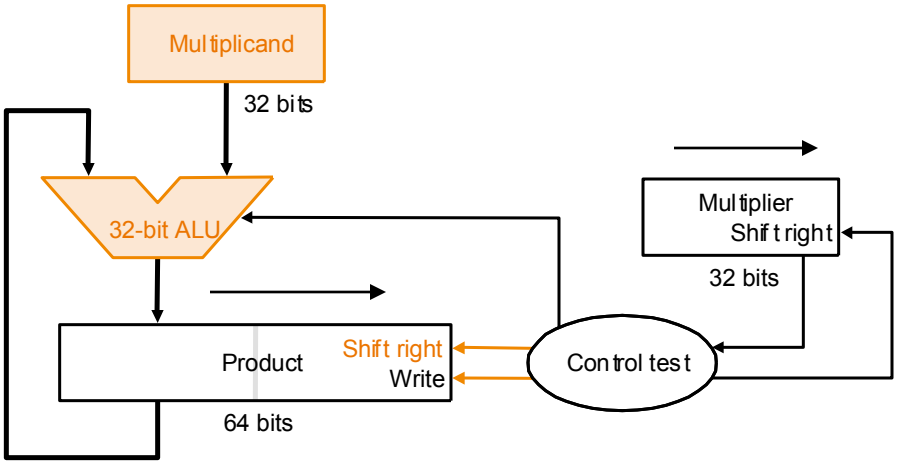


Hardware Implementation



Flow Chart

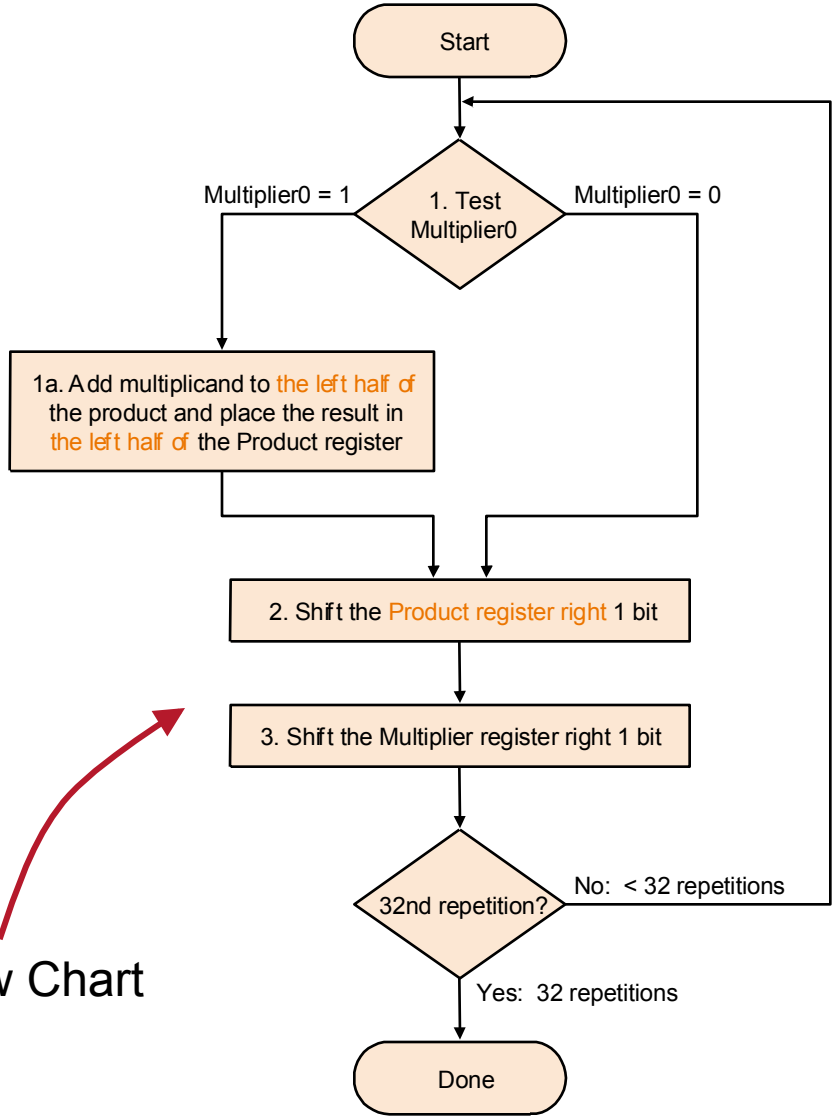
Second Version



More Efficient Hardware Implementation

32-bit ALU, rather than 64-bit ALU

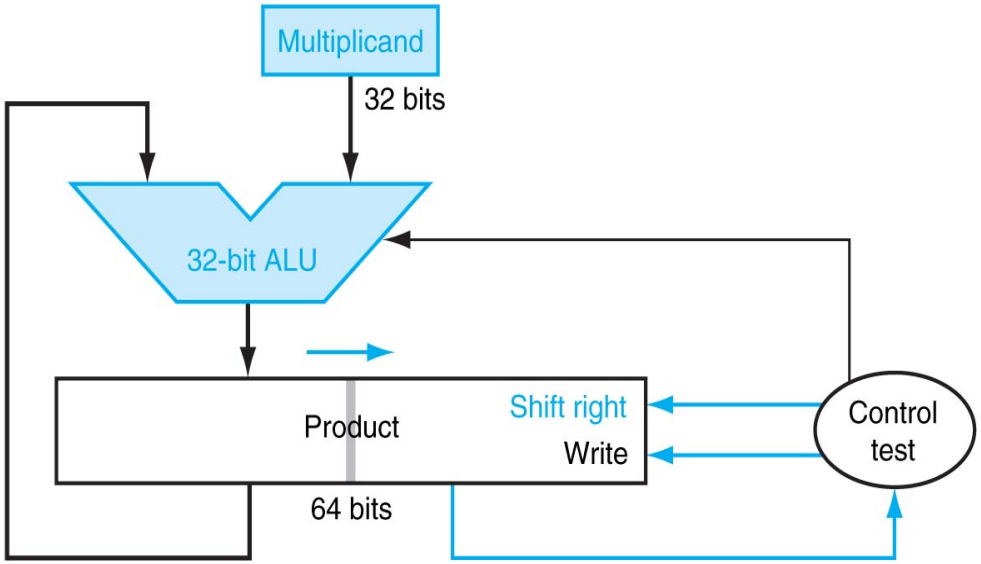
Flow Chart

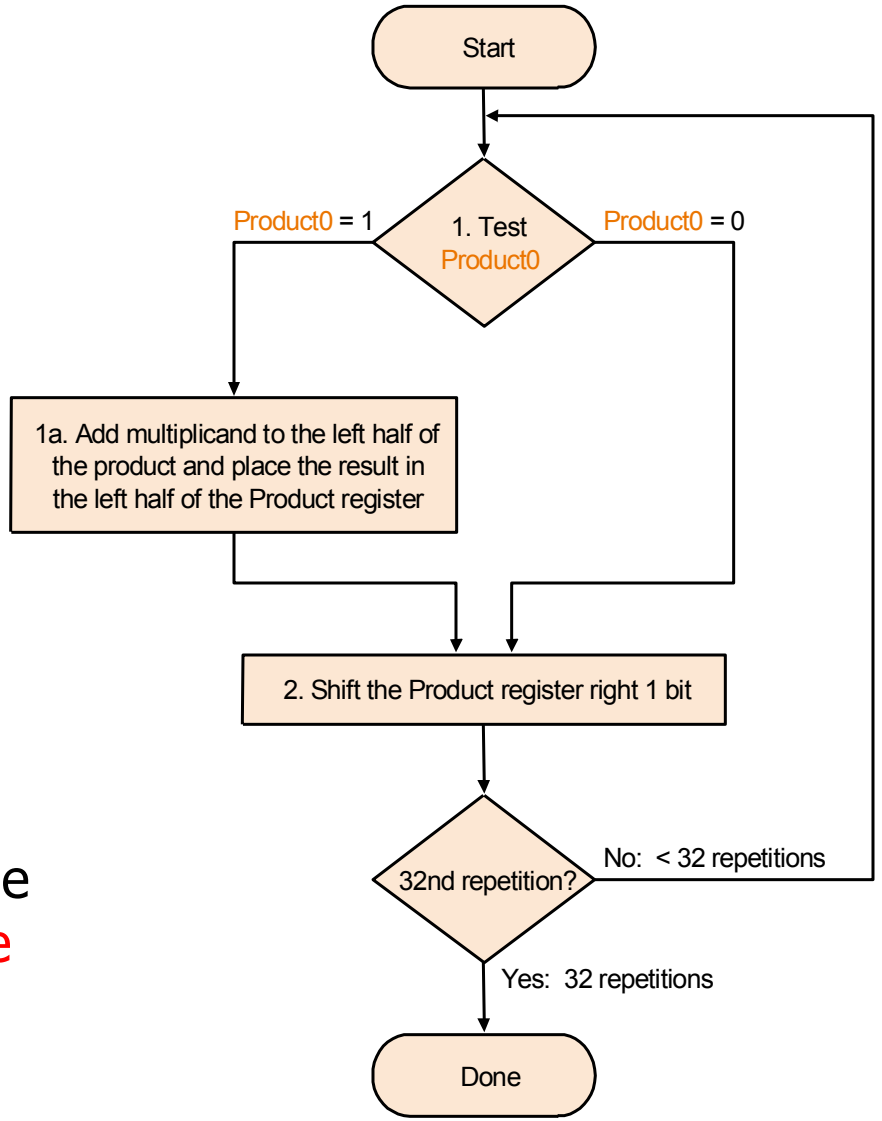


Example for second version

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial	1011	0010	0000 0000
1	Test true shift right	1011 0101	0010	0010 0000 0001 0000
2	Test true shift right	0101 0010	0010	0011 0000 0001 1000
3	Test false shift right	0010 0001	0010	0001 1000 0000 1100
4	Test true shift right	0001 0000	0010	0010 1100 0001 0110

Final Version





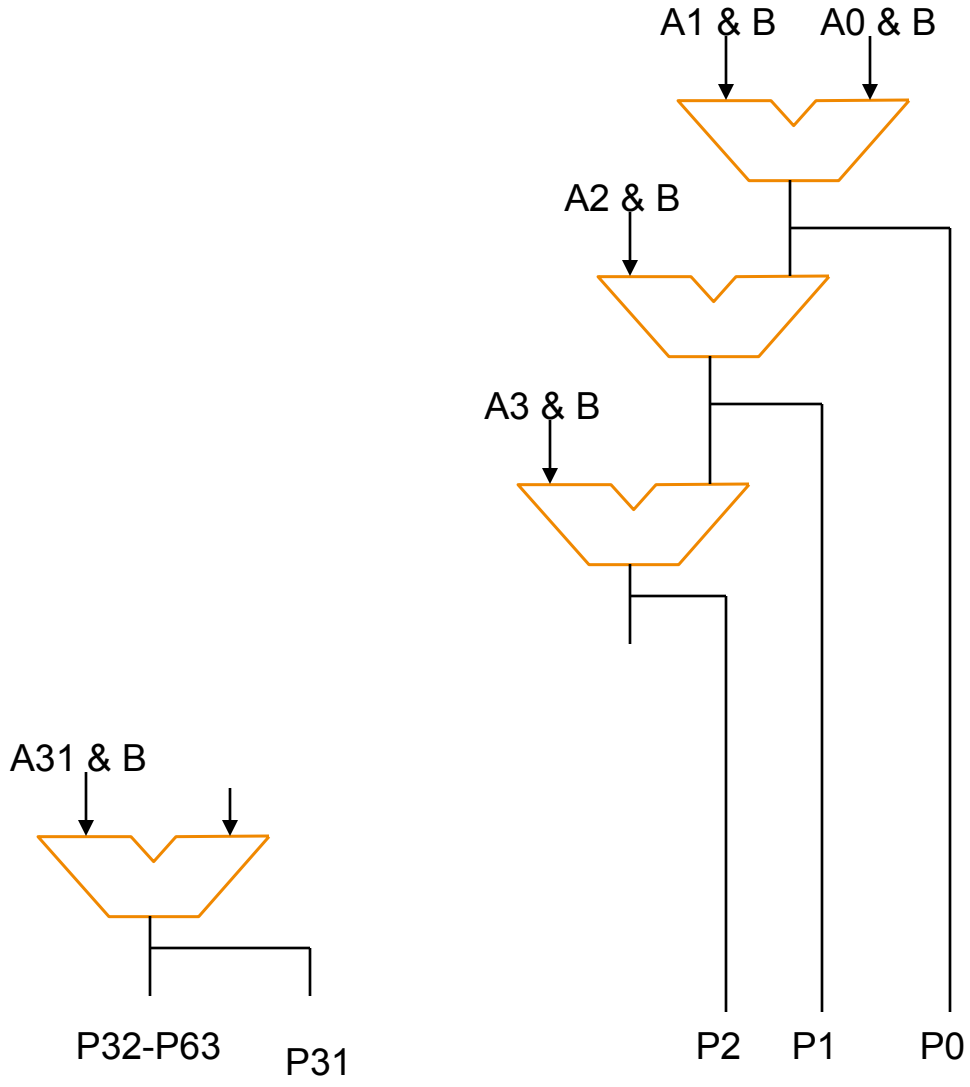
The trick is to use the lower half of the product to hold the multiplier during the operation – lose bits of multiplier as we increase the Product

Even More Efficient
Hardware Implementation!

What about the sign?

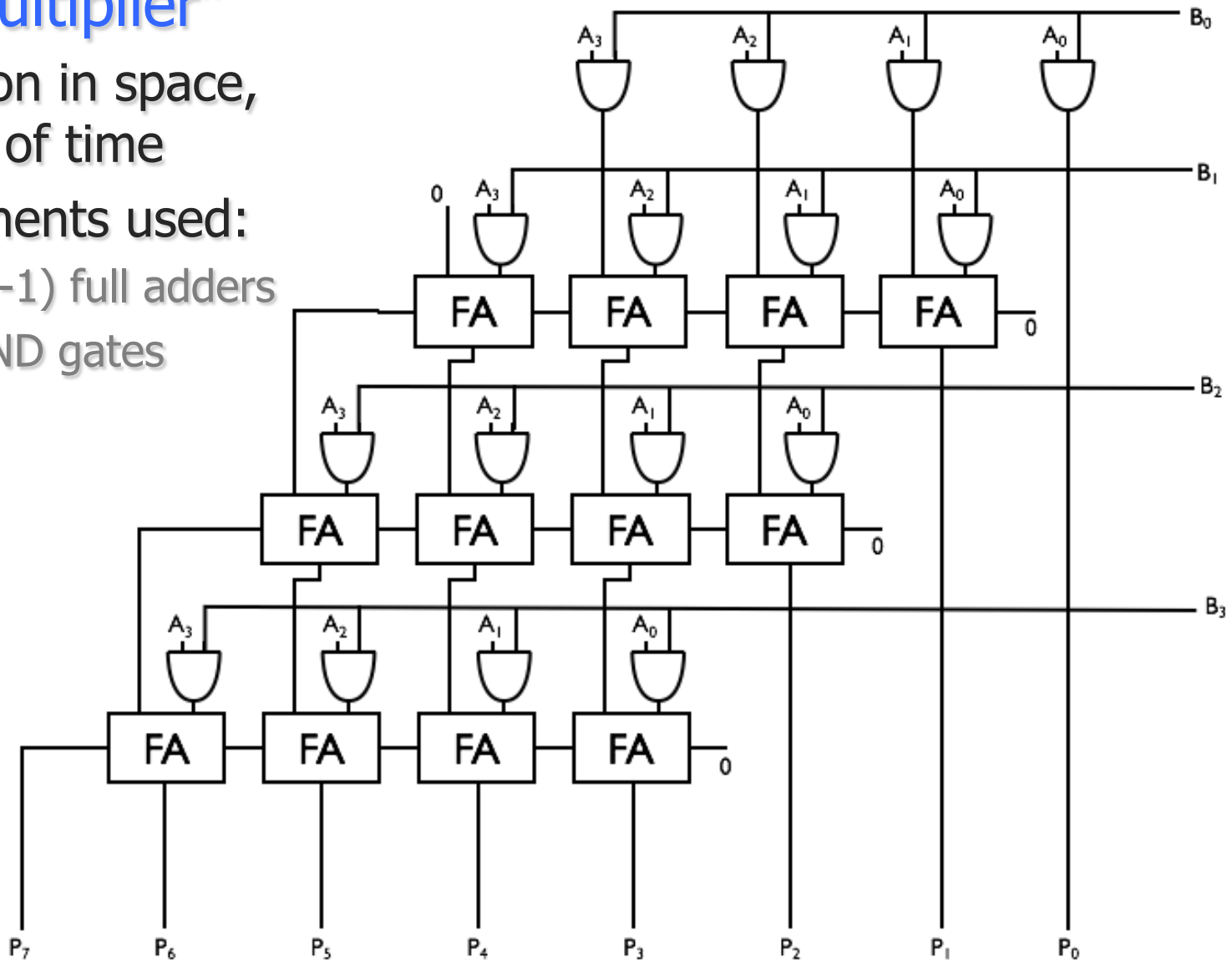
- Positive numbers are easy
- How about negative numbers?
  - Please read signed multiplication in textbook (Ch 3.3)

Faster Multiply



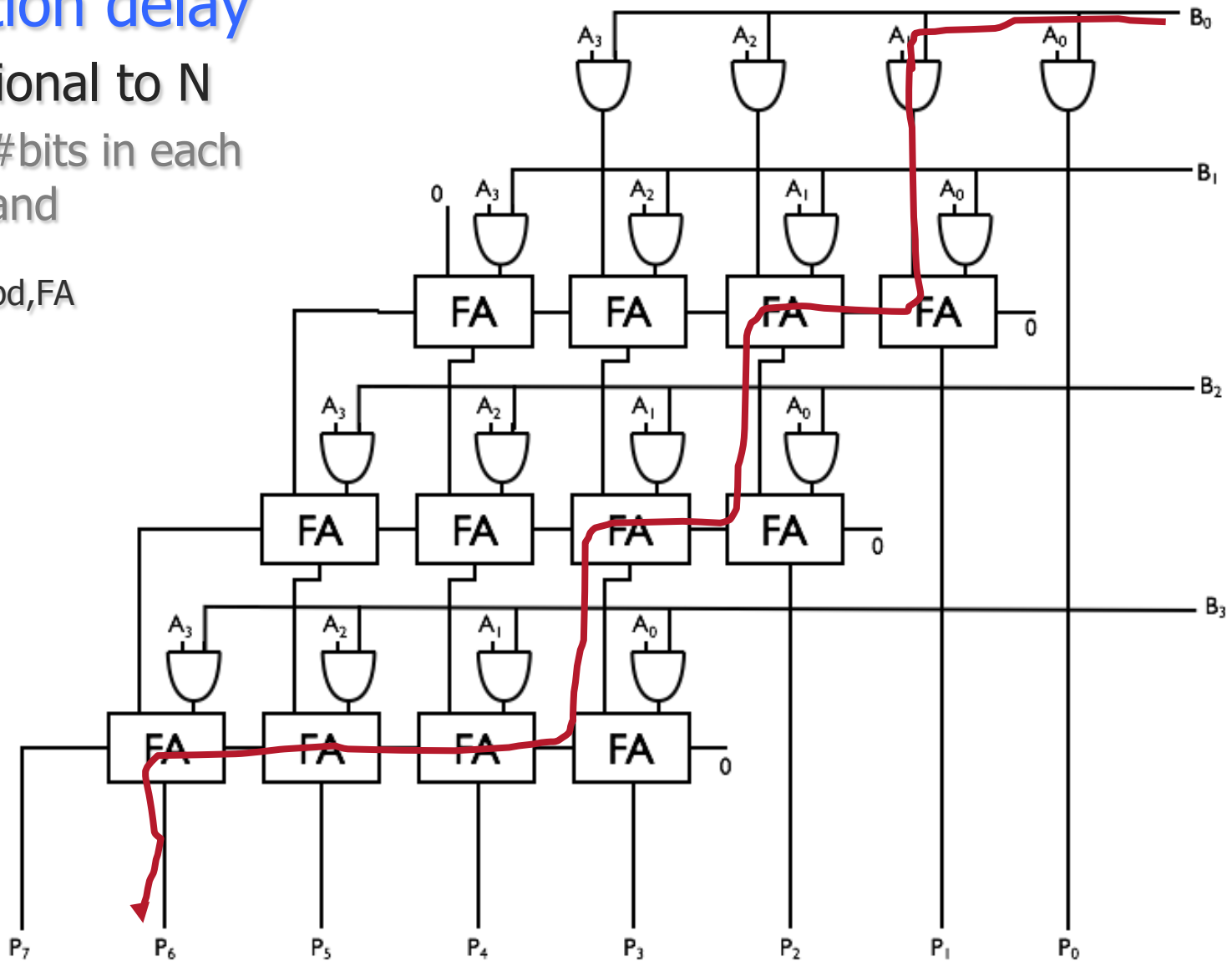
Simple Combinational Multiplier

- "Array Multiplier"
  - repetition in space, instead of time
  - Components used:
    - N*(N-1) full adders
    - N^2 AND gates



Simple Combinational Multiplier

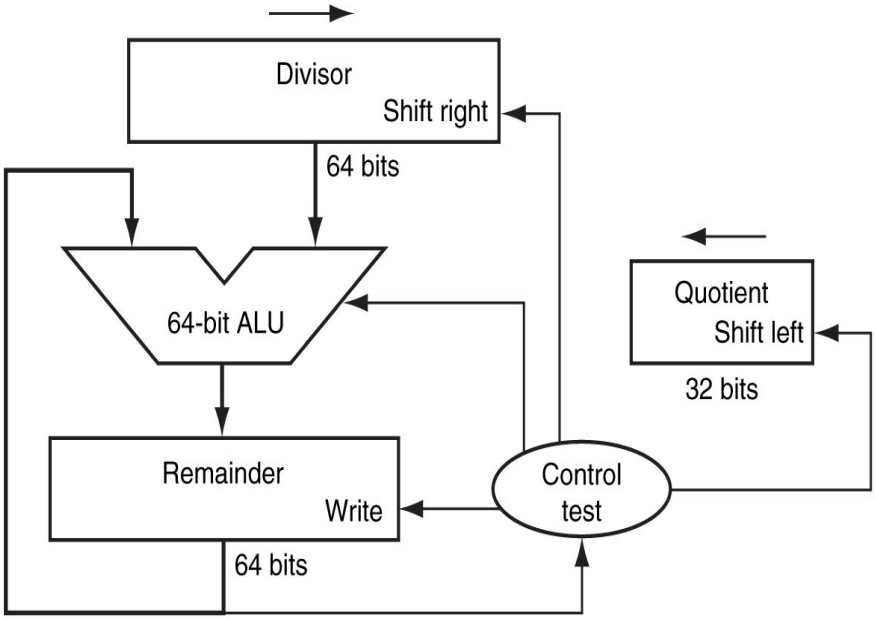
- Propagation delay
  - Proportional to N
    - N is #bits in each operand
  - ~ $3N*t_{pd,FA}$



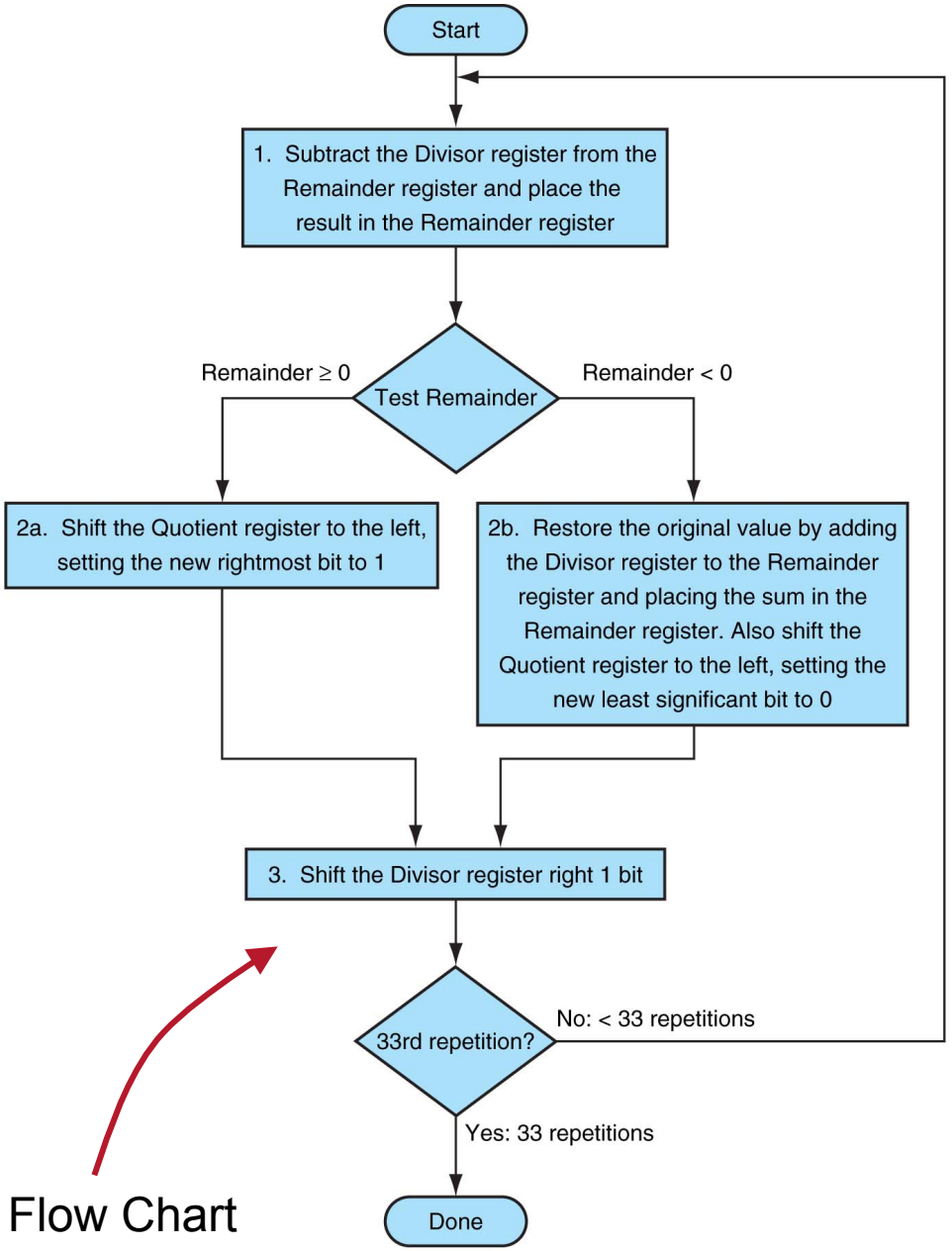
Even Faster Multiply

- Even faster designs for multiplication
  - e.g., "Carry-Save Multiplier"
    - covered in advanced courses

Division



Hardware Implementation



Flow Chart

See example in textbook (Fig 3.11)

Floating-Point Numbers & Arithmetic

Floating-Point Arithmetic

Reading: Study Chapter 3.5
Skim 3.6 and 3.8

Why do we need floating point?

- Several reasons:
  - Many numeric applications need numbers over a huge range
    - e.g., nanoseconds to centuries
  - Most scientific applications require real numbers (e.g. π)

- But so far we only have integers. What do we do?
  - We could implement the fractions explicitly
    - e.g.: ½, 1023/102934
  - We could use bigger integers
    - e.g.: 64-bit integers
  - Floating-point representation is often better
    - has some drawbacks too!

Recall Scientific Notation

Recall scientific notation from high school

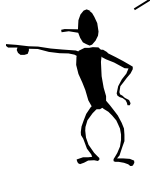
- Numbers represented in parts:
  - $42 = 4.200 \times 10^{1}$
  - $1024 = 1.024 \times 10^{3}$
  - $-0.0625 = -6.250 \times 10^{-2}$

Significant Digits

Exponent

Arithmetic is done in pieces

	1024	1.024×10^{3}
-	42	-0.042×10^{3}
=	982	0.982×10^{3}
=		9.820×10^{2}



Before adding, we must match the exponents, effectively "denormalizing" the smaller magnitude number

We then "normalize" the final result so there is one digit to the left of the decimal point and adjust the exponent accordingly.

Multiplication in Scientific Notation

- Is straightforward:
  - Multiply together the significant parts
  - Add the exponents
  - Normalize if required
- Examples:

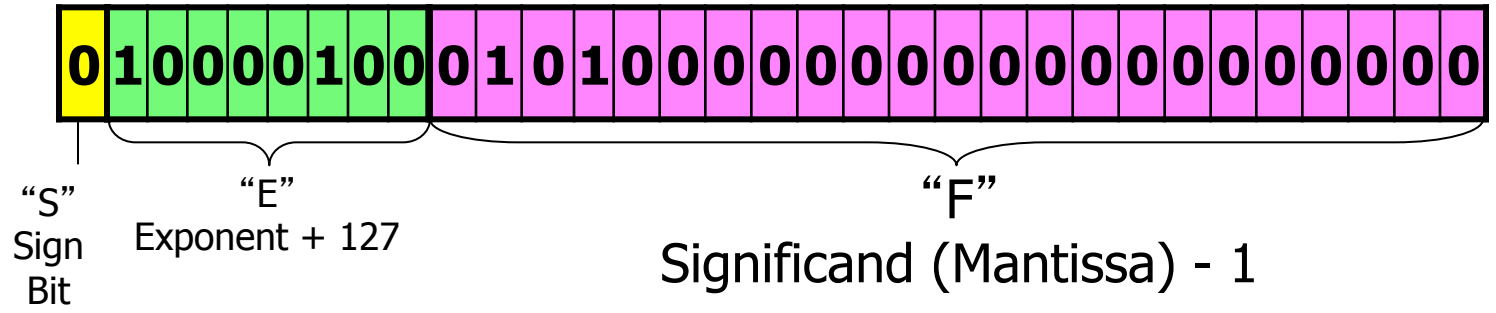
1024	1.024×10^{3}
x 0.0625	6.250×10^{-2}
= 64	6.400×10^{1}
42	4.200×10^{1}
x 0.0625	6.250×10^{-2}
= 2.625	26.250×10^{-1}
=	2.625×10^{0} (Normalized)

In multiplication, how far is the most you will ever normalize?

In addition?

Binary Floating-Point Notation

- IEEE single precision floating-point format
  - Example: (0x42280000 in hexadecimal)



- Three fields:
  - Sign bit (S)
  - Exponent (E): Unsigned "Bias 127" 8-bit integer
    - E = Exponent + 127
    - Exponent = 10000100 (132) – 127 = 5
  - Significand (F): Unsigned fixed-point with "hidden 1"
    - Significand = "1"+ 0.01010000000000000000000 = 1.3125
  - Final value: $N = -1^{S}(1+F) \times 2^{E-127} = -1^{0}(1.3125) \times 2^{5} = 42$

Example Numbers

- One
  - Sign = +, Exponent = 0, Significand = 1.0
    - $1 = -1^0 (1.0) \times 2^0$
    - S = 0, E = 0 + 127, F = 1.0 – '1'
    - 0 01111111 00000000000000000000000 = 0x3f800000
- One-half
  - Sign = +, Exponent = -1, Significand = 1.0
    - $½ = -1^0 (1.0) \times 2^{-1}$
    - S = 0, E = -1 + 127, F = 1.0 – '1'
    - 0 01111110 00000000000000000000000 = 0x3f000000
- Minus Two
  - Sign = -, Exponent = 1, Significand = 1.0
    - $-2 = -1^1 (1.0) \times 2^1$
    - 1 10000000 00000000000000000000000 = 0xc0000000

Zeros

How do you represent 0?

- Sign = ?, Exponent = ?, Significand = ?
  - Here's where the hidden "1" comes back to bite you
  - Hint: Zero is small. What's the smallest number you can generate?
    - Exponent = -127, Signficand = 1.0
    - $-1^0 (1.0) \times 2^{-127} = 5.87747 \times 10^{-39}$

IEEE Convention

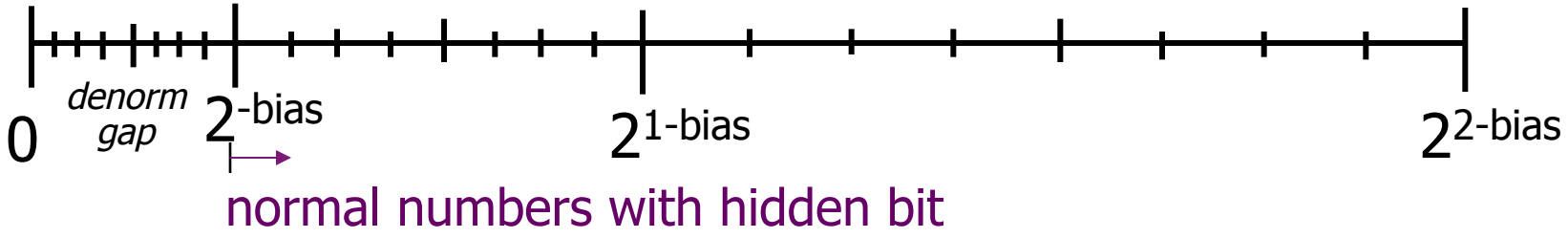
- When E = 0 (Exponent = -127), we'll interpret numbers differently…
  - 0 00000000 00000000000000000000000 = 0 not 1.0×2^{-127}
  - 1 00000000 00000000000000000000000 = -0 not -1.0×2^{-127}

Yes, there are "2" zeros. Setting E=0 is also used to represent a few other small numbers besides 0. In all of these numbers there is no "hidden" one assumed in F, and they are called the "unnormalized numbers". WARNING: If you rely these values you are skating on thin ice!

Infinities

- IEEE floating point also reserves the largest possible exponent to represent "unrepresentable" large numbers
  - Positive Infinity: S = 0, E = 255, F = 0
    - 0 11111111 00000000000000000000000 = +∞
    - 0x7f800000
  - Negative Infinity: S = 1, E = 255, F = 0
    - 1 11111111 00000000000000000000000 = -∞
    - 0xff800000
  - Other numbers with E = 255 (F ≠ 0) are used to represent exceptions or Not-A-Number (NAN)
    - √-1, -∞ x 42, 0/0, ∞/∞, log(-5)
  - It does, however, attempt to handle a few special cases:
    - 1/0 = + ∞, -1/0 = - ∞, log(0) = - ∞

Low-End of the IEEE Spectrum



- "Denormalized Gap"
  - The gap between 0 and the next representable normalized number is much larger than the gaps between nearby representable numbers
  - IEEE standard uses denormalized numbers to fill in the gap, making the distances between numbers near 0 more alike
    - Denormalized numbers have a hidden "0" and…
    - … a fixed exponent of -126
    - $X = -1^{S}\ 2^{-126}\ (\mathbf{0}.F)$
      - Zero is represented using 0 for the exponent and 0 for the mantissa. Either, +0 or -0 can be represented, based on the sign bit.

Floating point AIN'T NATURAL

- It is CRUCIAL for computer scientists to know that Floating Point arithmetic is NOT the arithmetic you learned since childhood

- 1.0 is NOT EQUAL to 10*0.1 (Why?)
  - 1.0 * 10.0 == 10.0
  - 0.1 * 10.0 != 1.0
  - 0.1 decimal == 1/16 + 1/32 + 1/256 + 1/512 + 1/4096 + … ==
    - 0.0 0011 0011 0011 0011 0011 …
  - In decimal 1/3 is a repeating fraction 0.333333…
  - If you quit at some fixed number of digits, then 3 * 1/3 != 1

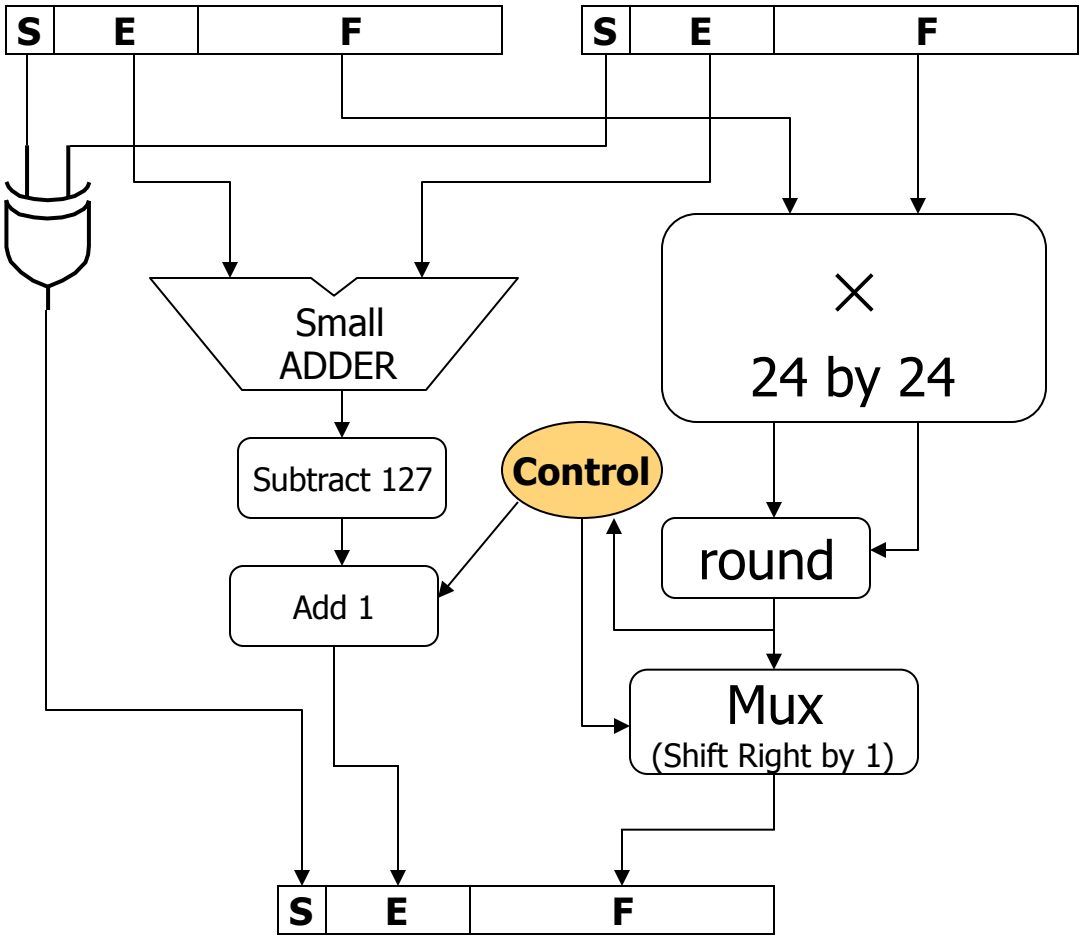
- Floating Point arithmetic IS NOT associative
  - x + (y + z) is not necessarily equal to (x + y) + z

- Addition may not even result in a change
  - (x + 1) MAY == x

Floating Point Disasters

- Scud Missiles get through, 28 die
  - In 1991, during the 1st Gulf War, a Patriot missile defense system let a Scud get through, hit a barracks, and kill 28 people. The problem was due to a floating-point error when taking the difference of a converted & scaled integer. (Source: Robert Skeel, "Round-off error cripples Patriot Missile", SIAM News, July 1992.)
- $7B Rocket crashes (Ariane 5)
  - When the first ESA Ariane 5 was launched on June 4, 1996, it lasted only 39 seconds, then the rocket veered off course and self-destructed. An inertial system, produced a floating-point exception while trying to convert a 64-bit floating-point number to an integer. Ironically, the same code was used in the Ariane 4, but the larger values were never generated (http://www.around.com/ariane.html).
- Intel Ships and Denies Bugs
  - In 1994, Intel shipped its first Pentium processors with a floating-point divide bug. The bug was due to bad look-up tables used to speed up quotient calculations. After months of denials, Intel adopted a no-questions replacement policy, costing $300M. (http://www.intel.com/support/processors/pentium/fdiv/)

Floating-Point Multiplication



Step 1:

Multiply significands

Add exponents

$E_R = E_1 + E_2 - 127$

(do not need twice the bias)

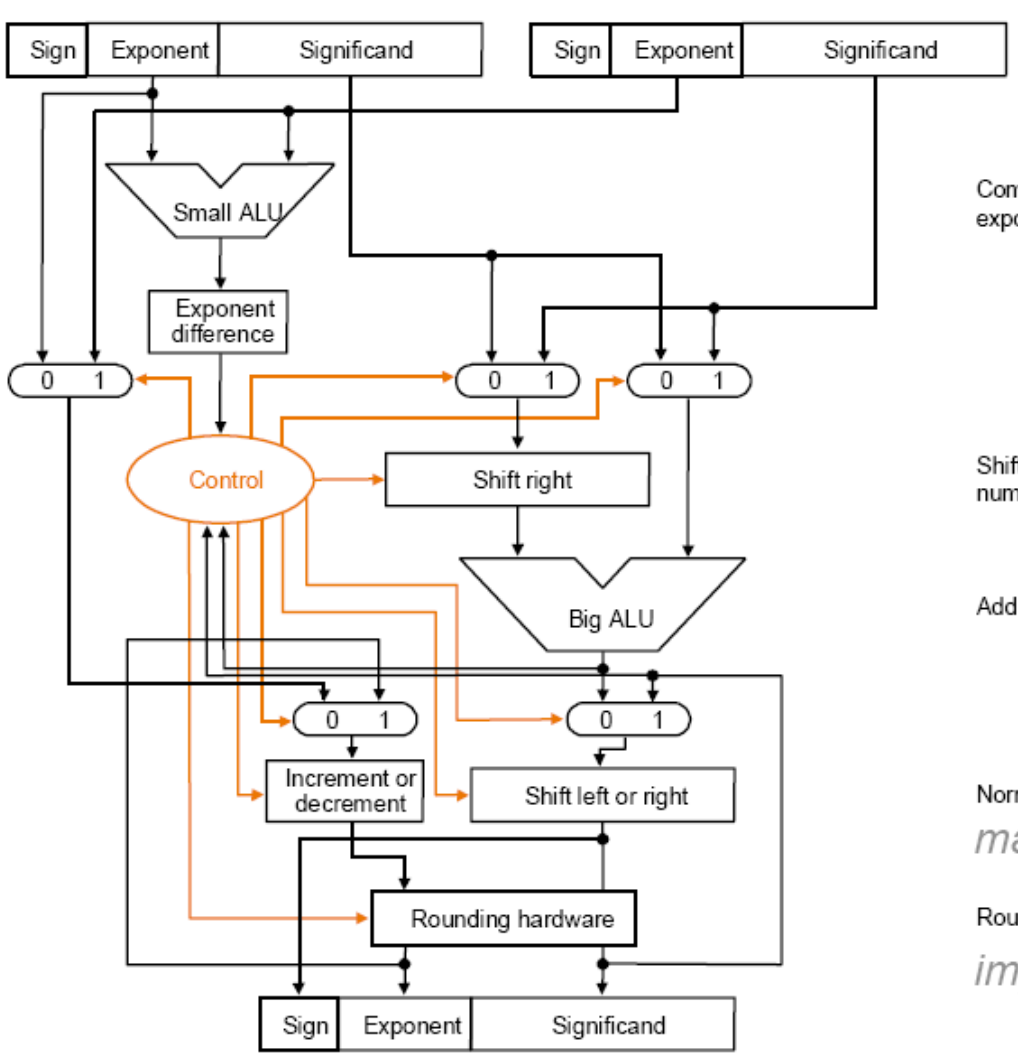
Step 2:

Normalize result

(Result of [1,2) *[1.2) = [1,4)

at most we shift right one bit, and fix exponent

Floating-Point Addition



MIPS Floating Point

- Floating point "Co-processor"
  - Separate co-processor for supporting floating-point
    - Separate circuitry for arithmetic, logic operations
- Registers
  - F0…F31: each 32 bits
    - Good for single-precision (floats)
  - Or, pair them up: F0|F1 pair, F2|F3 pair … F30|F31 pair
    - Simply refer to them as F0, F2, F4, etc. Pairing implicit from instruction used
    - Good for 64-bit double-precision (doubles)

MIPS Floating Point

- Instructions determine single/double precision
  - add.s $F2, $F4, $F6 // F2=F4+F6 single-precision add
  - add.d $F2, $F4, $F6 // F2=F4+F6 double-precision add
    - Really using F2|F3 pair, F4|F5 pair, F6|F7 pair
- Instructions available:
  - add.d fd, fs, ft # fd = fs + ft in double precision
  - add.s fd, fs, ft # fd = fs + ft in single precision
  - sub.d, sub.s, mul.d, mul.s, div.d, div.s, abs.d, abs.s
  - l.d fd, address # load a double from address
  - l.s, s.d, s.s
  - Conversion instructions: cvt.w.s, cvt.s.d, …
  - Compare instructions: c.lt.s, c.lt.d, …
  - Branch (bc1t, bc1f): branch on comparison true/false

Next

- Sequential circuits
  - Those with memory
  - Useful for registers, state machines
- Let’s put it all together
  - … and build a CPU