# ECE437: Introduction to Digital Computer Design

Chapter 3b (mul,div, floating point)

Fall 2016

## Revisiting Computer Arithmetic

- Earlier in semester
  - Integer Representation
  - Addition/Subtraction
  - Logical ops
    - Barrel shifter
- Next:
  - Integer Multiplication
  - Integer Division
  - Floating point numbers
    - Representation
    - Addition/multiplication



## Grade School Multiplication

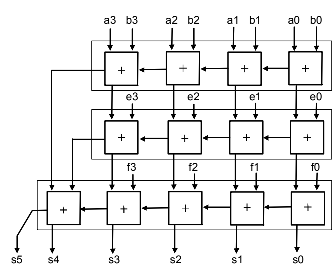
- Multiplicand ($1000_{10}$) and Multiplier ($1001_{10}$)

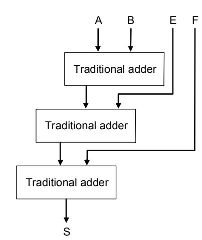
```
      1000
     0000
    0000
   1000
   ----------
   1001000
```
$1001000_{10}$

- Two approaches
  - Purely combinational
  - Sequential



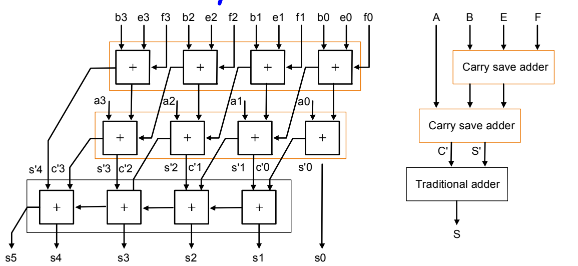
## Adding more than two operands




- Add four numbers (A,B,E,F)
- "Traditional adder" : ripple carry
- Basic block: Full adder
- N-1 adders for N numbers

# Carry-save adders




- Same basic block
- Adds three numbers per stage (using Cins) and output 2 numbers (Cout and S)
  - Number of stages reduced to ~2/3N for N-numbers



# Purely combinational

- Uses carry-save adders (in a tree organization)
  - Called Wallace tree for multiplication
  - Fast but lots of hardware (quite big)
- Partial products
  - AND multiplicand with multiplier bits

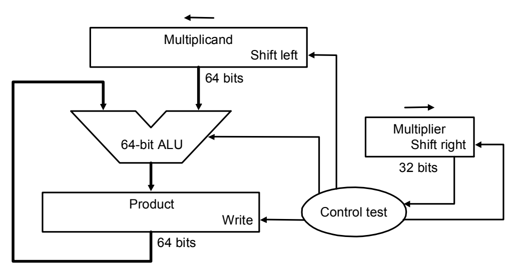


# Grade School Multiplication

- Sequential approach
- Reuse hardware
  - Partial products generated, accumulated into product each cycle



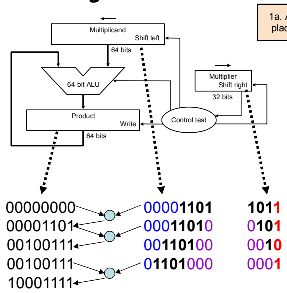
# Grade-School Version

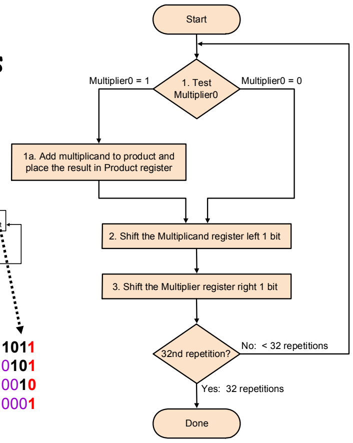



- Naïve implementation
  - 2x 64-bit registers, 1x 32 bit register
  - 1x 64-bit ALU

# Flow chart

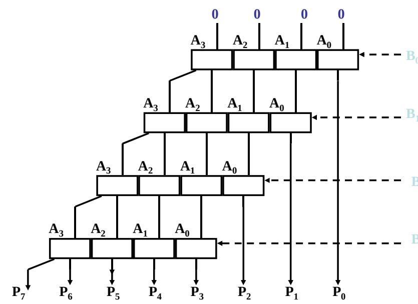
- Repeated shifts and adds
  - E.g. 13*11 = 143





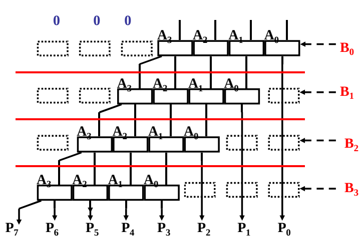

# How does it work?




- Faithful reproduction of grade-school algorithm



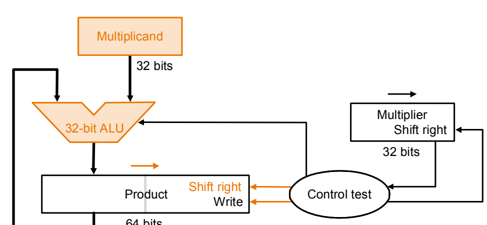
# Version #1




- Additional zeroes (padding)
- Only n valid bits – other bits are zeros



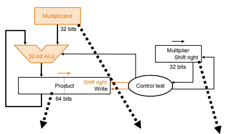
# Optimization # 1




- Insight: Product LSB bits are frozen after each iteration
- Reduced 64-bit ALU to 32-bit ALU
- Very simple optimization – no big deal

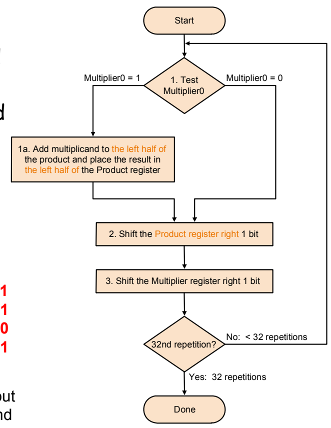
# Flow chart

- Product register shifted right after LSB frozen
- 64-bit product computed with 32-bit adder



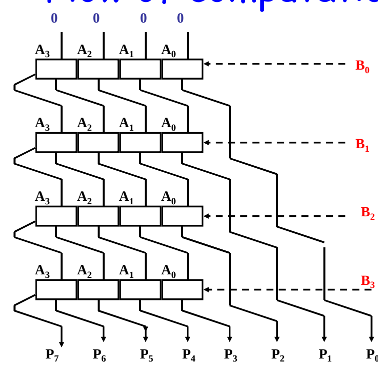

00000000 1101 1011
0|11010000 1101 0101
1|00111000 1101 0010
0|10011100 1101 0001
1|00011110

Each step shows product after add but before shift → needs a shift at the end



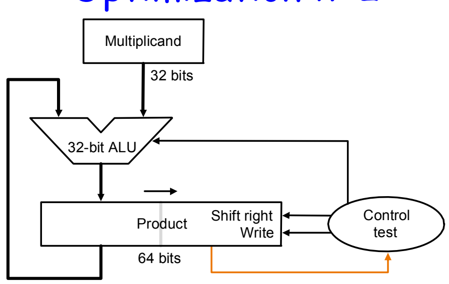



# Flow of computation






# Optimization # 2

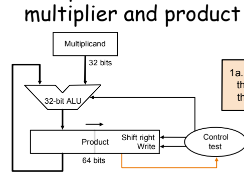



- Insight: bits of multiplier are consumed
  - Put multiplier in not-yet-used part of product
  - 1x 32-bit register, 1x 64-bit register, 32-bit ALU
  - Simple optimization – no big deal



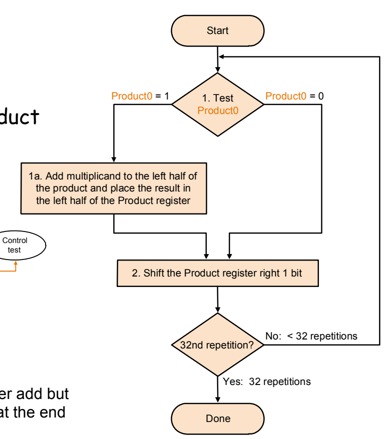
# Flow chart

- Product register manipulates both multiplier and product




00001011 1101
0|11011011 1101
1|00111101 1101
0|10011110 1101
1|00011111

Each step shows product after add but before shift → needs a shift at the end



# Signed Multiplication

- { <+,+>, <-,-> } : +
- { <+,->, <-,+> } : -

- If multiplier negative,
  - Multiplier = - Multiplier
  - Negative ++
- If multiplicand negative,
  - Multiplicand = -Multiplicand
  - Negative ++
- Product = Multiplicand * Multiplier
- If (Negative) sign = '-' , else sign = '+'



# Booth's Algorithm

- Signed/Unsigned integer multiplication
- Previous algo has n steps – can we do better?
- Intuition : exploit run-lengths



# Booth's Algorithm Intuition

- How would you compute the decimal product:
  - 5345 * 999 ?
  - 5345 * (1000 – 1)
  - 5345000 – 5345 = 5339655
- What about
  - 5345 * 9990
  - 5345 * (10000 – 10)
  - 53450000 – 53450 = 53396550
- Now, what about
  - 5345 * 9900990
  - 5345 * (1000000 – 100000 + 1000 – 10)
  - 52915500000 + 5291550 = 52920791550
- Works only when multiplier has only '9's in the decimal system



# Application to Binary

- "1 is the new 9"
- 9 = 10-1 (10 is base of decimal system)
- 1 = 2-1 (2 is the base of binary system)
- For example
  - 0010 * 0011
  - 0010 * (0100 – 0001)

## Run of '1's identification

middle of run

end of run | 0 | 1 | 1 1 | 1 | 0 | beginning of run

| Current bit | Bit to right | Explanation | Example | Action |
|---|---|---|---|---|
| 1 | 0 | Start of run of '1's | 11001110 | sub |
| 1 | 1 | Middle of run of '1's | 11001110 | nop |
| 0 | 1 | End of run of '1's | 11001110 | add |
| 0 | 0 | Middle of run of '0's | 11001110 | nop |



## Works for Signed numbers

- Consider 2's complement number 10110
  - Normal 2's complement view
    - 10110 = -16 + 0 + 4 + 2 + 0 = -10
  - Booth encoding view
    - 10110 = -16 + 8 - 0 - 2 + 0 = -10
    - 10110 :: $\bar{1}10\bar{1}0$ in Booth encoding

  - 1011<u>0</u>(<u>0</u>)
  - 101<u>10</u>(0)
  - 10<u>11</u>0(0)
  - 1<u>01</u>10(0)
  - <u>10</u>110(0)



## Booth's Algorithm

- Example: -4*6
  - 4-bit 2's complement 1100 * 0110
  - Extend multiplicand width
    - -4 = 1111 1100
  - Booth encoding of Multiplier
    - 6 = $10\bar{1}0$ (corresponds to +8 -2 )

| | | |
|---|---|---|
| 1111 1100 | 0 | 0000 0000 |
| 1111 1000 | $\bar{1}$ | 0000 1000 |
| 1111 0000 | 0 | 0000 0000 |
| 1110 0000 | 1 | 1110 0000 |
| | | 1110 1000 |



## The Mathematical Basis

- Recall the table
  - Multiplier = $a_{j-1} - a_j$
- 3-bit numbers
  - A $(a_2, a_1, a_0)$
  - B $(b_2, b_1, b_0)$
- Compute B x A
  - B * { $(a_1 - a_2).2^2 + (a_0 - a_1).2^1 + (0 - a_0).2^0$ }
  - B * { $-a_2.(2^2) + a_1.(2^2-2^1) + a_0.(2^1-2^0)$ }
  - B * { $-a_2.(2^2) + a_1.(2^1) + a_0.(2^0)$ }
  - B * A !!

| Current bit | Bit to right | Multiplier |
|---|---|---|
| 1 | 0 | -1 |
| 1 | 1 | 0 |
| 0 | 1 | +1 |
| 0 | 0 | 0 |

# Summary

- Integer multiplication:
  - Grade school version with minor optimizations
- Sophisticated optimizations
  - Booth's algorithms
    - Can do multiple bits at a time
    - Like CLA for addition
    - We won't go into this topic
  - Remember CLA vs. ripple-carry (grade-school version)



# Recap

- Booth's Multiply Algorithm
  - Identify and exploit runs of '1's
  - Remember analogy of multiplication by 999
    - 0110 normal encoding (4+2 = 6)
    - $10\bar{1}0$ Booth encoding (8-2 = 6)



# Outline

- Integer Division
  - Restoring/Non-restoring
- Floating Point Arithmetic



# Integer Division

- Remember progressive optimization of multiplication hardware
  - Similar Story
- Start with naïve hardware
  - Faithful implementation of grade-school method (long division)
  - Optimize

# Grade School Division

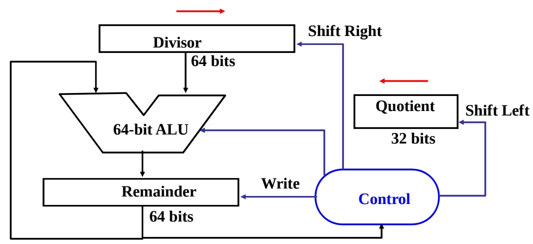
- Consider decimal division
  - Subtract largest possible multiple
  - Shift down one digit
  - "Lather, Rinse and repeat"
  - Binary
    - Exactly two choices: 0 or 1



Dividend = Quotient * Divisor + Remainder
|Dividend| = |Quotient| + |Divisor|



# Version #1

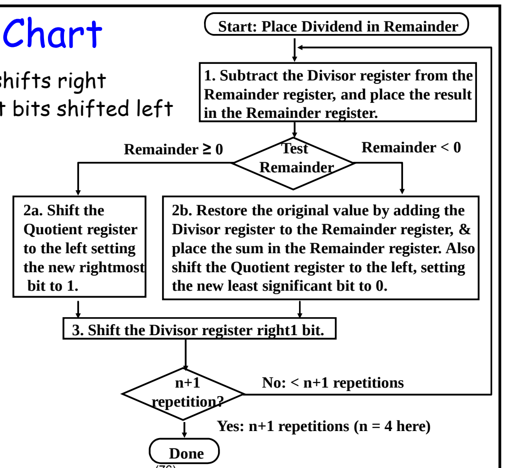



- Dividend initially in Remainder reg
- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg



# Flow Chart

- Divisor shifts right
- Quotient bits shifted left






# Example

- n+1 steps for n-bit quotient
- 7 / 2 = (3,1)
- Use 8(4) bit 2's complement representation
- Register size/ALU size optimizations possible

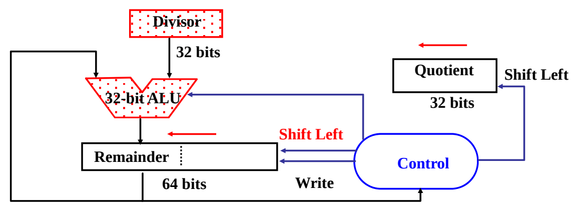
| | Remainder | Divisor | Quotient |
|---|---|---|---|
| | 0000 0111 | 0010 0000 | 0000 |
| | 1110 0111 (after subtraction) | | |
| (restore) | 0000 0111 | 0001 0000 | 0000 |
| | 1111 0111 | | |
| | 0000 0111 | 0000 1000 | 0000 |
| | 1111 1111 | | |
| | 0000 0111 | 0000 0100 | 0000 |
| | 0000 0011 | 0000 0010 | 0001 |
| | 0000 0001 | 0000 0001 | 0011 |

# Observations on Version #1

- 1/2 bits in divisor always 0
  - 1/2 of 64-bit adder is wasted
  - 1/2 of divisor is wasted
- Instead of shifting divisor to right, shift remainder to left?
- Has n+1 steps where 1st step cannot produce a 1 in quotient bit (can n-bit division produce a n+1-bit quotient with MSB 1?)
  - switch order to shift first and then subtract, can save 1 iteration



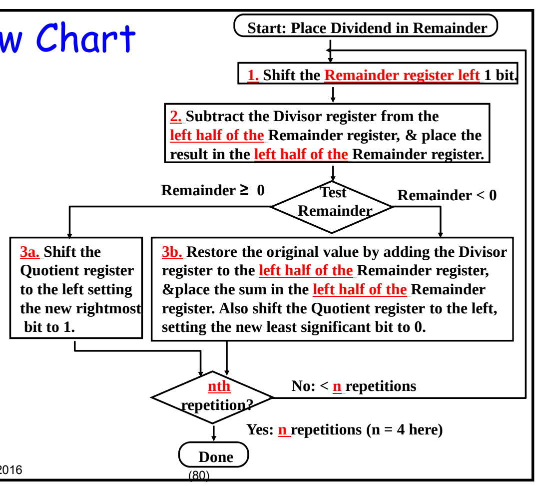
# Version #2




- 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg



# Flow Chart






# Example

- n-steps only
- Because shift is done first

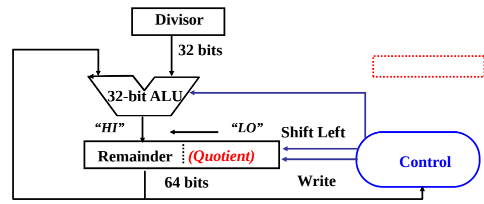
| Remainder | Divisor | Quotient |
|---|---|---|
| 0000 1110 | 1110 | 0000 |
| 1110 1110 | | |
| 0000 1110 | 0010 | 0000 |
| 0001 1100 | 1110 | |
| 1111 1100 | | |
| 0001 1100 | 0010 | 0000 |
| 0011 1000 | 1110 | |
| 0001 1000 | | 0001 |
| 0011 0000 | 1110 | |
| 0001 0000 | | 0011 |

# Observations on Version #2

- Eliminate Quotient register by combining with Remainder as shifted left
  - Start by shifting the Remainder left as before.
  - Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half
  - The consequence of combining the two registers together and the new order of the operations in the loop is that the remainder is shifted left one time too many.
  - Thus the final correction step must right-shift only the left-half of the remainder



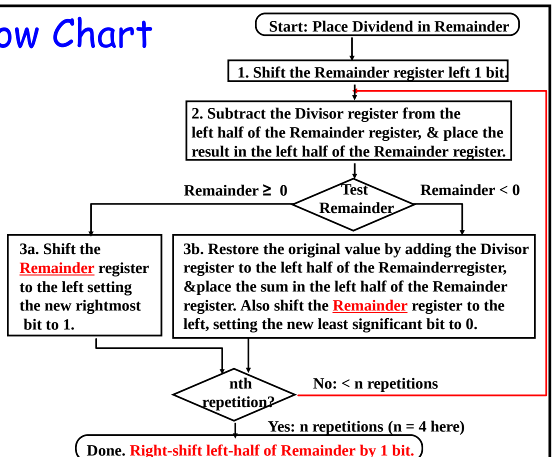
# Version #3




- 32-bit Divisor reg, 32 -bit ALU, 64-bit Remainder reg, (0-bit Quotient reg)



# Flow Chart






# Example

- Remember to right-shift the left-half of remainder register

| Remainder | Divisor |
|---|---|
| 0000 1110 | 1110 |
| 1110 1110 | |
| 0000 1110 | 0010 |
| 0001 1100 | 1110 |
| 1111 1100 | |
| 0001 1100 | 0010 |
| 0011 1000 | 1110 |
| 0001 1000 | |
| 0011 0001 | 1110 |
| 0001 0001 | |
| 0010 0011 | |
| | |
| **0001 0011** | |

## Non-restoring division

- Restoring Division
  - Subtract largest possible multiple
    - Try '1'
    - If too large (i.e. remainder negative), restore
- Alternative
  - Non-restoring division
  - Introduce $\overline{1}$ (Like Booth's)

ECE437, Fall 2016 (86)

## Non-Restoring Division

- With +ve divisor
  - Subtract if positive remainder
  - Add if negative remainder
- Vice versa with –ve divisor

| | Remainder | Divisor | Quotient |
|---|---|---|---|
| - | 0000 0111 | 0010 0000 | 00000 |
| + | 1110 0111 | 0001 0000 | 00001 |
| + | 1111 0111 | 0000 1000 | 00011 |
| + | 1111 1111 | 0000 0100 | 00111 |
| - | 0000 0011 | 0000 0010 | 01111 |
| | 0000 0001 | 0000 0001 | 1111 1 |

Quotient = 0011 => $1 \times 2^1 + 1 \times 2^0$
Quotient = 1$\overline{111}$ 1 => $1 \times 2^4 - 1 \times 2^3 - 1 \times 2^2 - 1 \times 2^1 + 1 \times 2^0$

ECE437, Fall 2016 (87)

## Conversion to standard form

- Non-restoring division uses two unique symbols
  - Can use "0" for $\overline{1}$ (but means the value is -1 and not 0)
- Issues
  - Imperfect division before all bits are computed
  - Re-conversion to standard form

ECE437, Fall 2016 (88)

## Another restoring division

- The key problem in restoring division is the restore add.
- Two of you suggested why not do the usual subtract but NOT store the subtract results if result is negative (and set quotient to 0). Then no need to restore. This is as good as non-restoring division without the drama of 1, 1bar, and conversion etc
- YES this is called non-performing restoring division

ECE437, Fall 2016 (89)

## Outline

- Integer Division
  - Restoring and Non-restoring
- Floating Point Arithmetic



## Floating Point

- Integer arithmetic till now
- How to represent real numbers:
  - Uncountably infinite
  - Realistically, can operate with limited number of significant digits (precision)
  - Remember exponent separately
- Remember scientific norms
  - Planck's constant $6.626068 \times 10^{-34}$ $m^2$ kg/s
  - Avogadro's constant $6.022 \times 10^{23}$
  - Normalization:
    - Not $0.6022 \times 10^{24}$, Not $60.22 \times 10^{22}$
    - MSD in [1,9] range except for 0.0



## FP in hardware

- Binary instead of decimal
- MSB = 1 (equivalent to [1,9] in decimal)
  - $(-1)^s \times f \times 2^e$
  - **s: Sign** stored separately
  - **f:** is of the form 1.F
  - **e:** is the (signed integer) exponent
- Optimizations to save bits
  - Base not stored (implicit 2)
  - MSB not stored (always 1)



## Binary Fractions Recap

- How do you represent $3.125_{(10)}$ in binary?
  - $11.001_{(2)}$
- For integer conversion to binary
  - Repeated division by 2 till quotient goes to zero
- For fraction conversion to binary
  - Repeated multiplication by 2 till fraction goes to zero
- Recurring binary fraction
  - $0.6_{(10)} = 0.1001\ 1001\ 1001\ \ldots_{(2)}$

**0.125 * 2 = 0 . 25**
**0.25 * 2 = 0 . 5**
**0.5 * 2 = 1 . 0**

**0.6 * 2 = 1 . 2**
**0.2 * 2 = 0 . 4**
**0.4 * 2 = 0 . 8**
**0.8 * 2 = 1 . 6**
**0.6 * 2 = 1 . 2**

# IEEE 754

float : 32 bits | S(1) | E (8 bit) | F (23 bit)

double : 64 bits | S(1) | E (11 bit) | F (52 bit)

- Floating point representation standard
- Floating points stored as a packed triple
  - <S, E, F>
  - S : 1 bit
    - 0 → positive
    - 1 → negative
  - E : Biased representation for signed numbers
    - 8 bit (single precision—`float`) (Bias = +127)
    - 11 bit (double precision—`double`) (Bias = +1023)
  - F : (remember the 1 in 1.F is not stored)
    - 23 bit (single precision—`float`)
    - 52 bit (double precision—`double`)



# Examples

- 3.125 in double precision
  - 11.001
  - $(-1)^0 \times 1.1001 \times 2^1$
  - Exponent : 1 with bias of 1023 = 1024 (add bias to actual exponent to get IEEE exponent)
  - 0100 0000 0000 1001 0000 0000 …
  - 0x4009 0000 0000 0000
- 0.6 in single precision
  - 0.100110011001…
  - $(-1)^0 \times 1.001100110011\ldots \times 2^{-1}$
  - Exponent : -1 with bias of 127 = 126
  - 0011 1111 0001 1001 1001…
  - 0x3F19 9999



# FP Representation Worksheet

- Consider two floating point numbers (represented in single precision, IEEE 754 format). A = 0xBFC00000 and B = 0x41600000. If C = A x B, what is the hexadecimal representation of C in IEEE 754 format.



# Solution

float : 32 bits | S(1) | E (8 bit) | F (23 bit)

- A = 0xBFC00000
  = 1011 1111 1100 0000 0000 0000 0000 0000
  = -1 * 1.5 * $2^0$
- B = 0x41600000
  = 0100 0001 0110 0000 0000 0000 0000 0000
  = +1 * 1.75 * $2^3$
- C = -21 = -1 * 1.0101 (binary)* $2^4$
  = 1 *100 0001 1*010 1000 0000 0000 0000 0000
  = 0xC1A80000

# Biased Exponent

- Why?
  - 2's complement arithmetic is elegant
  - Larger numbers appear to have larger magnitude with biased representation
    - Negative numbers appear large in 2's complement
  - Biased so that comparing two FP numbers is like comparing two unsigned integers



# Exceptions

- Specified in great detail in the document IEEE 754-1985
  - 20 pages

| S | E | F | Number |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | Max | 0 | +inf |
| 1 | Max | 0 | - inf |
| X | Max | != 0 | NaN |
| X | 0 | != 0 | Denorm |



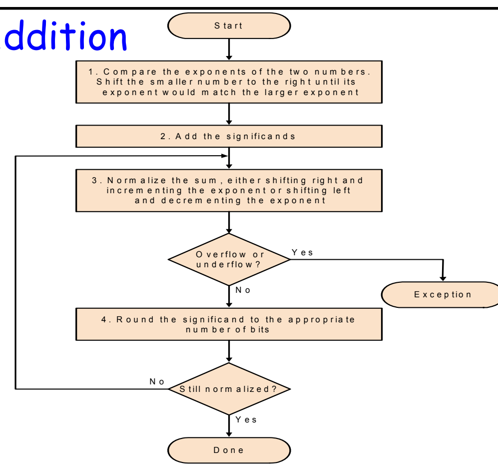
# Floating Point Addition

$9.997 \times 10^{2}$
$+\ 4.6371 \times 10^{-1}$

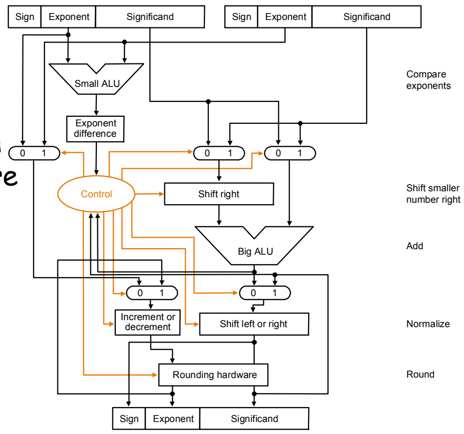
- FP addition
  - Align decimal point → $9.997 \times 10^{2}$ / $+\ 0.0046371 \times 10^{2}$
  - Add → $10.0016371 \times 10^{2}$
  - Normalize → $1.00016371 \times 10^{3}$
- May involve rounding
  - The LSB may be shifted right beyond the limited precision



# FP Addition



- FP Addition Hardware






# Normalization issues

- At most one shift for normalization after add
  - Applies when two operands are of same sign
  - Also the case when subtracting two operands of different signs
- Many shifts for
  - Subtract two numbers of similar sign
  - Add two numbers of dissimilar sign



# Example

- $1.00010 \times 10^{13} - 1.00001 \times 10^{13}$
- $0.00001 \times 10^{13}$
  - Not normalized
- $1.0 \times 10^{8}$
  - Normalize involves multiple digit shifts



# FP multiplication

- Example
  - $A = 3.0 \times 10^{1}$
  - $B = 5.0 \times 10^{2}$
  - $A \times B = (3.0 \times 5.0) \times 10^{(1+2)} = 15.0 \times 10^{3}$
  - $A \times B = 1.5 \times 10^{4}$
- Multiply mantissas
- Add exponents (no overflow/underflow)
- Normalize (and round)
- Set sign (easy, exor sign bits)

# Adding exponents

- Biased representation
- Adding biased numbers
- Raw number

  Actual: $e_1 = 12$, $e_2 = 13$ ; $e_* = (e_1+e_2) = 25$

  IEEE: $E_1 = 12 + 127 = 139$, $E_2 = 13 + 127 = 140$

  $E_* = e_*+127 = E_1 - 127 + E_2 - 127 + 127$

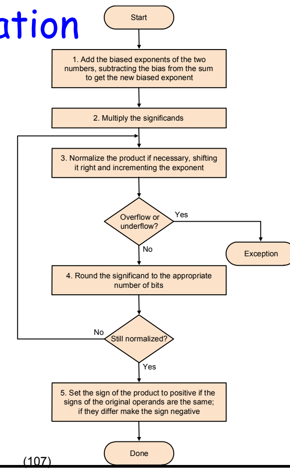
  $E_* = E_1 + E_2 - 127$

  $-127 = -(01111111) = (1000\ 0000) + 1$

1000 1011
1000 1100
1000 0001
---------
1001 1000



# FP Multiplication






# FP Multiplication

- Mantissa multiplication
  - Two non-negative integers
  - Recall
    - Carry save adders in tree (Wallace tree)
    - Booth's multiplication



# FP division

- Exponent computation
  - $E_/ = E_1 - E_2 + 127$

1000 1011
1000 1100
---------

- Raw number

  $e_1 = 12$, $e_2 = 13$ ; $e/ = (e_1-e_2) = -1$

  $E_1 = 12 + 127 = 139$, $E_2 = 13 + 127 = 140$

  $E_/ = e_/+127 = (E_1 - 127) - (E_2 - 127) + 127$

  $E_/ = E_1 - (E_2 - 127)$

  $-E_2 = 1\text{'s } C\ (E_2) + 1$

  $127 = (01111111)$

  $E_/ = E_1 + 1\text{'s}C\ (E_2) + 1000\ 0000$

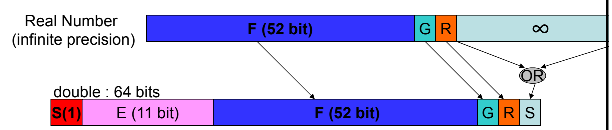
1000 1011
0111 0011
1000 0000
---------
0111 1110

# Rounding

- Decimal conventions
  - [5+,9] -> up
  - 5 -> round to even (4.5 ~ 4, 5.5 ~ 6)
  - [1,5-] -> down
  - 0 -> unchanged
- Binary rounding
  - xxx.1…1… -> up
  - xxx.10000 -> round to even
  - xxx.0xx1x -> down
  - xxx.00000 -> unchanged



# Guard, Round and Sticky bits




- May shift to renormalize
  - Save one bit to right of LSB (Guard bit)
  - This is a significant bit in the normalized representation
  - Round-bit
    - one additional to the right of guard bit
    - Used only for rounding, never becomes significant bit
  - Sticky bit
    - Logical OR of all bits right of round bit

| Round | Sticky | Action |
|---|---|---|
| 1 | 1 | Up |
| 1 | 0 | Even |
| 0 | 1 | Down |
| 0 | 0 | Unchanged |



# Rounding

- IEEE 754
  - Error bounds
  - No more than ½ "units of the last place" (ULP)
  - Errors accumulate
    - Billions of FP ops in a single application
- Effects of limited precision
  - Commutativity, associativity etc. may no longer apply
  - A = (3.1415 + 6.022x$10^{23}$) – 6.022x$10^{23}$
  - B = 3.1415 + (6.022x$10^{23}$ – 6.022x$10^{23}$)
  - Is A==B?
- Ch3 done!