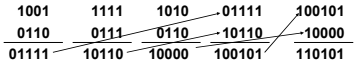
Booth's algorithm (Neg. multiplier)

Itera-tion	multi-plicand	Booth's algorithm	
		Step	Product
0	0010	Initial values	0000 1101 0
1	0010	1c: 10⇒ prod = Prod - Mcand	1110 1101 0
	0010	2: Shift right Product	1111 0110 1
2	0010	1b: 01⇒ prod = Prod + Mcand	0001 0110 1
	0010	2: Shift right Product	0000 1011 0
3	0010	1c: 10⇒ prod = Prod - Mcand	1110 1011 0
	0010	2: Shift right Product	1111 0101 1
4	0010	1d: 11 ⇒ no operation	1111 0101 1
	0010	2: Shift right Product	1111 1010 1

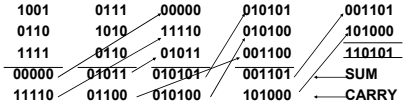
1

Carry-Save Addition

- Consider adding six set of numbers (4 bits each in the example)
- The numbers are 1001, 0110, 1111, 0111, 1010, 0110 (all positive)
- One way is to add them pair wise, getting three results, and then adding them again



- Other method is add them three at a time by saving carry



2

Carry-Save Addition for Multiplication

- n-bit carry-save adder take 1FA time for any n
- For n x n bit multiplication, n or n/2 (for 2 bit at time Booth's encoding) partial products can be generated
- For n partial products n/3 n-bit carry save adders can be used
- This yields 2n/3 partial results
- Repeat this operation until only two partial results are remaining
- Add them using an appropriate size adder to obtain 2n bit result
- For n=32, you need 30 carry save adders in eight stages taking 8T time where T is time for one-bit full adder
- Then you need one carry-propagate or carry-look-ahead adder

3

Division

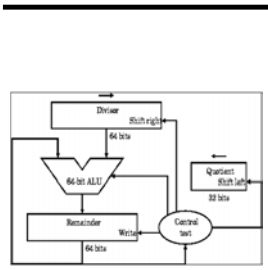
- Even more complicated
  - can be accomplished via shifting and addition/subtraction
- More time and more area
- We will look at 3 versions based on grade school algorithm

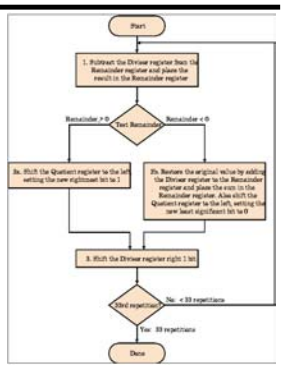
```
0011 | 0010 0010   (Dividend)
-----
```

- Negative numbers: Even more difficult
- There are better techniques, we won't look at them

4

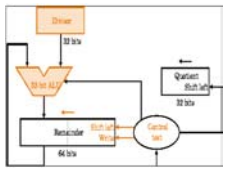
Division, First Version

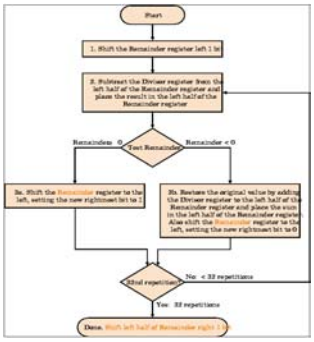




5

Division, Second Version





6

Division, Final Version

Restoring Division

Iteration	Divisor	Divide algorithm	
		Step	Remainder
0	0010	Initial values	0000 0111
	0010	Shift Rem left 1	0000 1110
1	0010	2: Rem = Rem - Div	1110 1110
	0010	3b: Rem < 0 ⇒ + Div, sll R, R0 = 0	0001 1100
2	0010	2: Rem = Rem - Div	1111 1100
	0010	3b: Rem < 0 ⇒ + Div, sll R, R0 = 0	0011 1000
3	0010	2: Rem = Rem - Div	0001 1000
	0010	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0011 0001
4	0010	2: Rem = Rem - Div	0001 0001
	0010	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0010 0011
Done	0010	shift left half of Rem right 1	0001 0011

Non-Restoring Division

Iteration	Divisor	Divide algorithm	
		Step	Remainder
0	0010	Initial values	0000 1110
1	0010	1: Rem = Rem - Div	1110 1110
	0010	2b: Rem < 0 ⇒,sll R, R0 = 0	1101 1100
	0010	3b: Rem = Rem + Div	1111 1100
2	0010	2b: Rem < 0 ⇒ sll R, R0 = 0	1111 1000
	0010	3b: Rem = Rem + Div	0001 1000
3	0010	2a: Rem > 0 ⇒ sll R, R0 = 1	0011 0001
	0010	3a: Rem = Rem - Div	0001 0001
4	0010	2a: Rem > 0 ⇒ sll R, R0 = 1	0010 0011
Done	0010	shift left half of Rem right 1	0001 0011

Floating Point (a brief look)

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., 3.15576×10^9
- Representation:
  - sign, exponent, significand: $(-1)^{sign} \times significand \times 2^{exponent}$
  - more bits for significand gives more accuracy
  - more bits for exponent increases range
- IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit significand
  - double precision: 11 bit exponent, 52 bit significand

IEEE 754 floating-point standard

- Leading "1" bit of significand is implicit
- Exponent is "biased" to make sorting easier
  - all 0s is smallest exponent all 1s is largest
  - bias of 127 for single precision and 1023 for double precision
  - summary: $(-1)^{sign} \times (1+significand) \times 2^{exponent - bias}$
- Example:
  - decimal: $-.75 = -3/4 = -3/2^2$
  - binary: $-.11 = -1.1 \times 2^{-1}$
  - floating point: exponent = 126 = 01111110
  - IEEE single precision: `10111111010000000000000000000000`

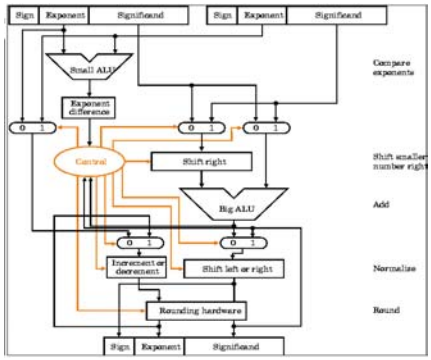
Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have "underflow"
- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round
  - four rounding modes
  - positive divided by zero yields "infinity"
  - zero divide by zero yields "not a number"
  - other complexities
- Implementing the standard can be tricky
- Not using the standard can be even worse
  - see text for description of 80x86 and Pentium bug!

Floating Point Add/Sub

- To add/sub two numbers
  - We first compare the two exponents
  - Select the higher of the two as the exponent of result
  - Select the significand part of lower exponent number and shift it right by the amount equal to the difference of two exponent
  - Remember to keep two shifted out bit and a guard bit
  - add/sub the signifand as required according to operation and signs of operands
  - Normalize significand of result adjusting exponent
  - Round the result (add one to the least significant bit to be retained if the first bit being thrown away is a 1
  - Re-normalize the result

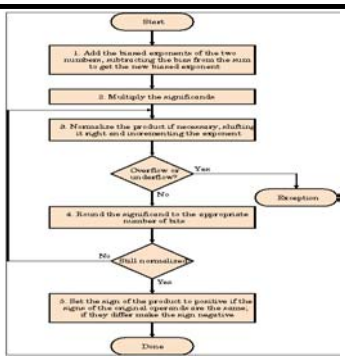
Hardware Organization for Floating Point Add



Floating Point Multiply

- To multiply two numbers
  - Add the two exponent (remember access 127 notation)
  - Produce the result sign as exor of two signs
  - Multiple significand portions
  - Results will be 1x.xxxxx… or 01.xxxx….
  - In the first case shift result right and adjust exponent
  - Round off the result
  - This may require another normalization step

Flow Diagram for Floating-point Multiply



Floating Point Divide

- To divide two numbers
  - Subtract divisor's exponent from the dividend's exponent (remember access 127 notation)
  - Produce the result sign as exor of two signs
  - Divide dividend's significand by divisor's significand portions
  - Results will be 1.xxxxx… or 0.1xxxx….
  - In the second case shift result left and adjust exponent
  - Round off the result
  - This may require another normalization step

Summary on Chapter 4

- Computer arithmetic is constrained by limited precision
- Bit patterns have no inherent meaning but standards do exist
  - two's complement
  - IEEE 754 floating point
- Computer instructions determine "meaning" of the bit patterns
- Performance and accuracy are important so there are many complexities in real machines (i.e., algorithms and implementation)
- We designed an ALU to carry out four function
- Multiplication: Unsigned, Signed, Signed using Booth's encoding, and Carry save adders and their use
- Division
- Floating Point representation
  - Guard and Sticky bit concepts
  - Chopping vs Truncation vs. Rounding in floating point numbers