EE 3613: Computer Organization

Arithmetic for Computers – 3
Multiplication, Division and Floating Point Representation

Avinash Karanth
Department of Electrical Engineering & Computer Science
Ohio University, Athens, Ohio 45701
E-mail: karanth@ohio.edu
Website: http://oucsace.cs.ohiou.edu/~avinashk/ee461a.htm

Acknowledgement: Srinivasan Ramasubramanian

Course Administration

- All lecture slides covered so far are online (except this set)
- Homework 2B is due this Friday Sept 25 by 11:59 PM EST. Recall the instructions on homework 2B (zipped, single file, naming convention)
- Exam 1 is scheduled for **Friday Oct 2** via blackboard (proctortrack)
    - Review on Wed Sept 30
    - Topics are Performance Metrics, Instruction Set Architecture and Computer Arithmetic
    - Homework 2 graded material and solutions will be available next week
    - On-boarding needed for those who have not completed it (from now to next Monday); everyone needs to complete the on-boarding by Sept 28th

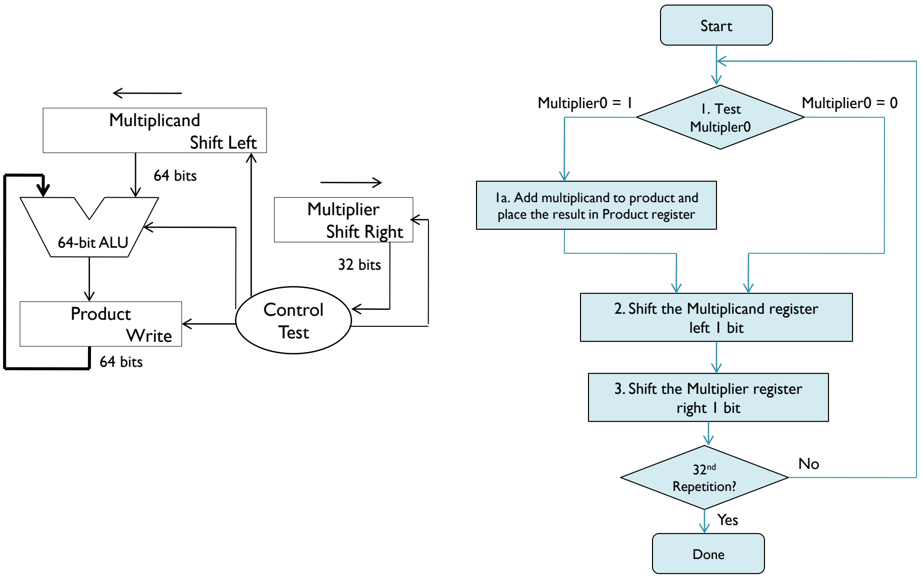
Multiplication

- More complicated than addition
  - Accomplished via shifting and addition
- More time and area
- Lets look at 3 versions based on grade school

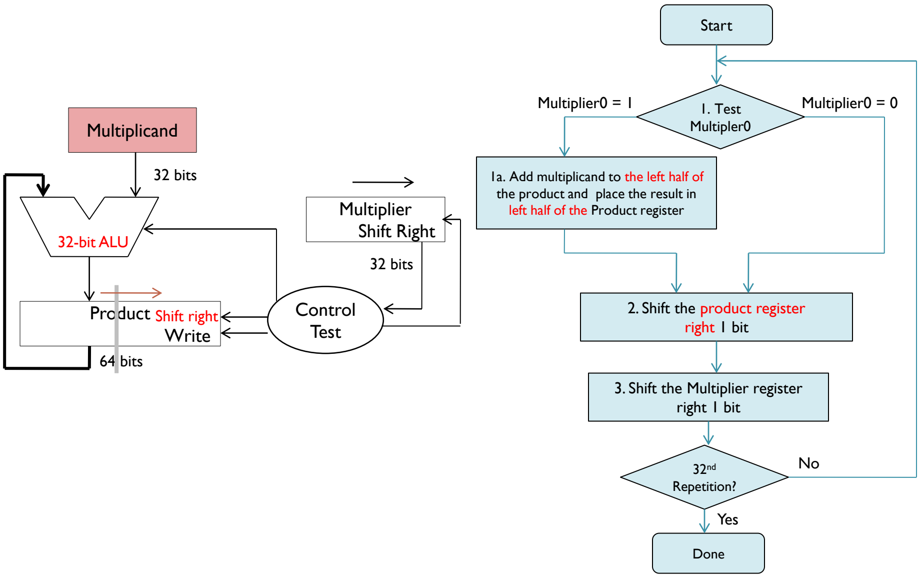
  0010 (multiplicand)

  X 0011 (multiplier)
- Negative numbers: convert and multiply
- Other technique like Booth's encoding can be better

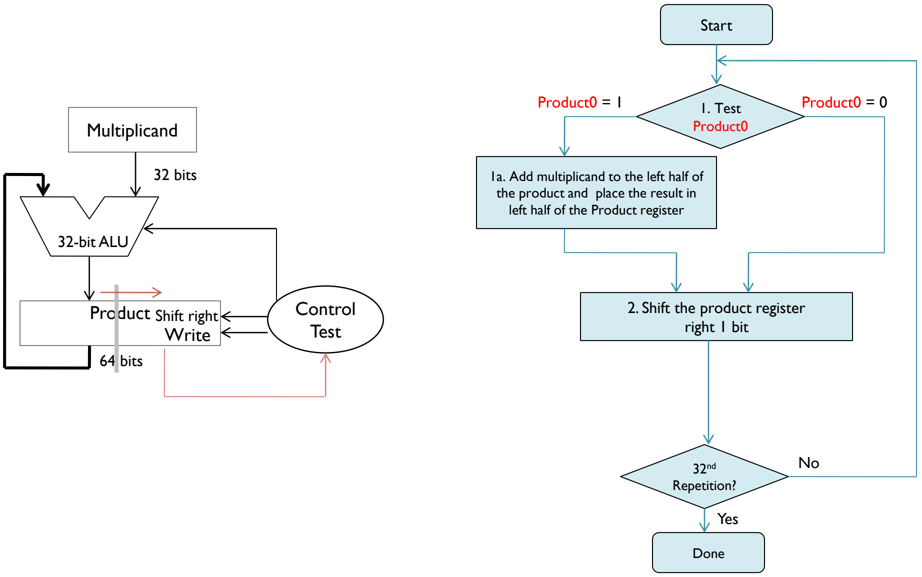
Multiplication Implementation



Second Version



Final Version



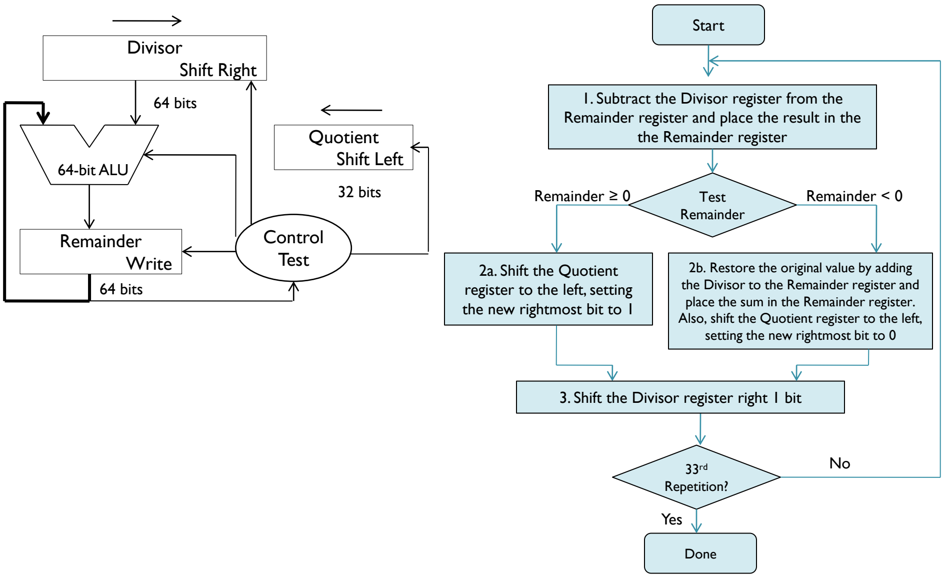
Multiplication Example: 0010 x 0011

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1a: 0 ➔ No Operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1a: 0 ➔ No Operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

Division

- Even more complicated
- Can be accomplished via shifting and addition/subtraction
- 1001010 ÷ 1000
- We will look at ONE version! Others refer to book
- Negative numbers are more difficult
  - There are better techniques, we will not be looking at them

Division Implementation



Start

1. Subtract the Divisor register from the Remainder register and place the result in the the Remainder register

Test Remainder

Remainder ≥ 0

Remainder < 0

2a. Shift the Quotient register to the left, setting the new rightmost bit to 1

2b. Restore the original value by adding the Divisor to the Remainder register and place the sum in the Remainder register. Also, shift the Quotient register to the left, setting the new rightmost bit to 0

3. Shift the Divisor register right 1 bit

33rd Repetition?

No

Yes

Done

Example: 7 ÷ 2 or 0000 0111 ÷ 0010

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem - Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ➔ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem - Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ➔ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem - Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ➔ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem - Div	0000	0000 0100	0000 0011
	2b: Rem ≥ 0 ➔ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem - Div	0001	0000 0010	0000 0001
	2b: Rem ≥ 0 ➔ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Floating Point Numbers

- Used to represent
  - Numbers with fractions Eg. 3.1416
  - Very small numbers Eg. 0.00000001
  - Very large numbers Eg. 3.15576 x 10^9
- Representation
  - Sign, exponent, significand: $(-1)^{sign} \times (1+\text{significand}) \times 2^{exponent-bias}$
  - 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

- Sign bit: (0 is positive, 1 is negative)
- Significand/Mantissa: (store 23 most significant bits after the decimal point), leading 1 is implicit
- Exponent: used biased base 127 encoding
  - Add 127 to the value of the exponent to encode

Examples

- $(-1)^{\text{sign}} \times (1 + \text{Significand}) \times 2^{(\text{exponent-bias})}$

31	29	28	.....	24	23	22	...	1	0
Sign									

8bits - Exponent

23bits - Significand

- Convert $-.75_{10}$ to binary
- Convert 10.625_{10} to binary
- Convert 1 1000 0001 $01000000000000000000000_2$ to decimal

Conversion Procedure

- The rules for converting a decimal number into floating point are as follows:
  - A. Convert the absolute value of the number into binary, perhaps with a fractional part after the binary point
  - B. Append x 2^0 to the end of the binary number (which does not change the value)
  - C. Normalize the number. Move the binary point so that it is one bit from the left. Adjust the exponent of two so that the value does not change.
  - D. Place the mantissa into the mantissa field of the number. Omit the leading one, and fill with zeros on the right.
  - E. Add the bias to the exponent of two and place it in the exponent field. The bias is $2^{k-1} - 1$, where k is the number of bits in the exponent field. For IEEE 32-bit, $k = 8$, so the bias is $2^{8-1} - 1 = 127$.
  - F. Set the sign bit, 1 for negative, 0 for positive, according to the sign of the original number.

Example: Convert 2.625_{10}

- A) The integral part is easy, $2_{10} = 10_2$. The fractional part can be converted by multiplication. (This is the inverse of the division method.)
    - 0.625 × 2 = 1.25   1
    - 0.25 × 2 = 0.5   0
    - 0.5 × 2 = 1.0   1
    - So $0.625_{10} = 0.101_2$, and $2.625_{10} = 10.101_2$
- B) Add an exponent part: $10.101_2 = 10.101_2 \times 2^0$
- C) Normalize: $10.101_2 \times 2^0 = 1.0101_2 \times 2^1$
- D) Mantissa: 0101
- E) Exponent: $1 + 127 = 128 = 1000\ 0000_2$.
- F) Sign is 0.
- RESULT → 0 1000 0000 01010000000000000000000 → 0x40280000

Reverse Conversion

- 0 1000 0000 01010000000000000000000
- $(-1)^{\text{sign}} \times (1 + \text{Significand}) \times 2^{(\text{exponent} - \text{bias})}$
- Exponent: $1000\ 0000_2 = 128$
- Significand: $0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = 0.25 + 0.0625$
- $(-1)^0 \times (1 + 0.3125) \times 2^{(128 - 127)} = 1 \times 1.3125 \times 2 = 2.625_{10}$

Floating-Point Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- Align decimal points
  - Shift number with smaller exponent
  - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
- Add significands
  - $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$
- Normalize result & check for over/underflow
  - 1.0015×10^{2}
- Round and renormalize if necessary
  - 1.002×10^{2}

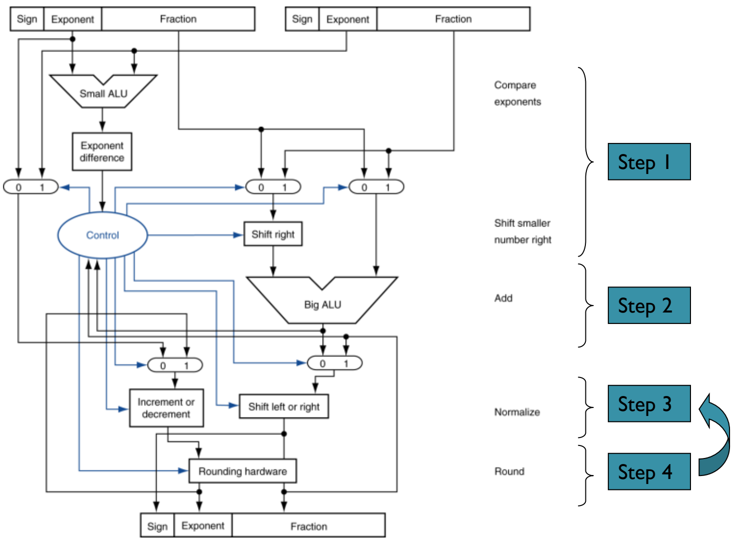
Floating-Point Addition

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ $(0.5 + -0.4375)$
- Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) $= 0.0625$

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
    - Much longer than integer operations
    - Slower clock would penalize all instructions
- FP adder usually takes several cycles
    - Can be pipelined

FP Adder Hardware



Floating Point Complexities

- Operations are somewhat complicated
  - Overflow and underflow
  - IEEE 754 keeps two extra bits, guard and round
  - Four rounding modes
  - Positive divide by zero yields "infinity"
  - Zero divide by zero yields "NaN – not a number'
  - Other complexities
- Implementing the standard can be tricky
- We will not be doing floating point multiplication or division – try on your own ☺

Summary

- Computer arithmetic is constrained by limited precision
- Bit patterns have no inherent meaning but standards do exist
- Computer instructions determine "meaning" of bit patterns
- Performance and accuracy are important so there are many complexities in real machines (algorithms and implementations)
- We designed an ALU to carry out 4 functions
- Multiplication, Division and floating point representation