CS/COE0447: Computer Organization and Assembly Language

Chapter 3

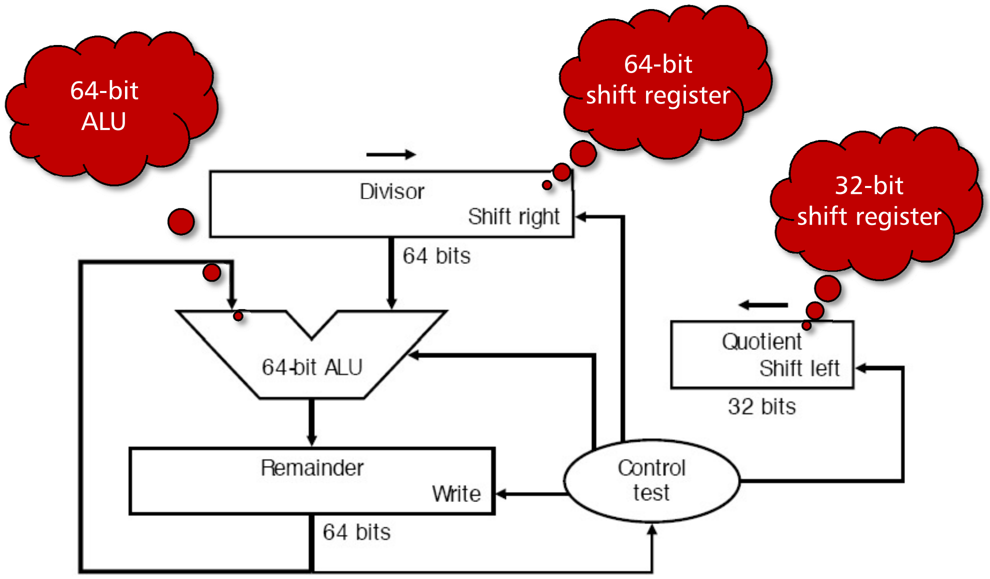
Sangyeun Cho

Dept. of Computer Science
University of Pittsburgh

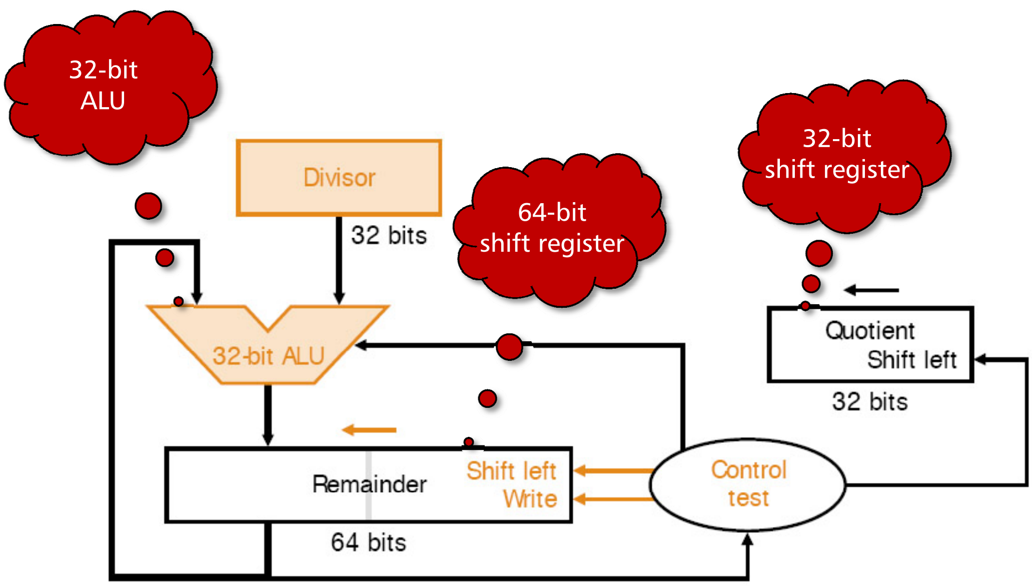
Binary division

- Dividend = divisor × quotient + remainder
- Given dividend and divisor, we want to obtain quotient (Q) and remainder (R)
- We will start from our paper & pencil method

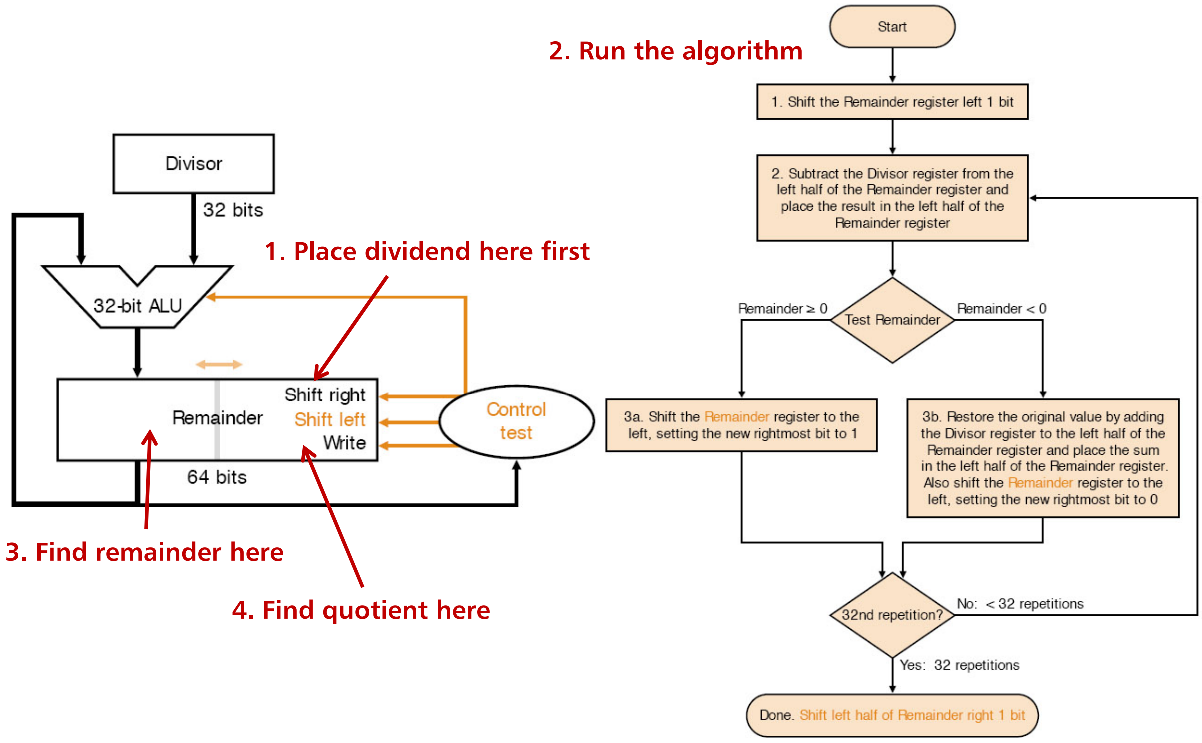
Hardware design 1



Hardware design 2



Hardware design 3



Example

- Let’s do 0111/0010 (7/2) – unsigned

Iteration	Divisor	Hardware design 3	
		Step	Remainder
0	0010	initial values	0000 0111
		shift remainder left by 1	0000 1110
1	0010	remainder = remainder – divisor	1110 1110
		(remainder<0) ⇒ +divisor; shift left; r0=0	0001 1100
2	0010	remainder = remainder – divisor	1111 1100
		(remainder<0) ⇒ +divisor; shift left; r0=0	0011 1000
3	0010	remainder = remainder – divisor	0001 1000
		(remainder>0) ⇒ shift left; r0=1	0011 0001
4	0010	remainder = remainder – divisor	0001 0001
		(remainder>0) ⇒ shift left; r0=1	0010 0011
done	0010	shift “left half of remainder” right by 1	0001 0011

Exercise sheet

Iteration	Divisor	Hardware design 3	
		Step	Remainder
0		initial values	
		shift remainder left by 1	
1			
2			
3			
4			
done		shift "left half of remainder" right by 1	

Restoring division

- The three hardware designs we saw are based on the notion of "restoring division"
  - At first, attempt to subtract divisor from dividend
  - If the result of subtraction is negative – it rolls back by adding divisor
    - This step is called "restoring"

- It's a "trial-and-error" approach; can we do better?

Non-restoring division

- Let's revisit the restoring division designs
  - Given remainder R (R<0) after subtraction
  - By adding divisor D back, we have (R+D)
  - After shifting the result, we have $2\times(R+D)=2\times R+2\times D$
  - If we subtract the divisor in the next step, we have $2\times R+2\times D–D =2\times R+D$
- This is equivalent to
  - Left-shifting R by 1 bit and then adding D!

Example, non-restoring division

- Let's again do 0111/0010 (7/2) – unsigned

Iteration	Divisor	Hardware design 3, non-restoring	
		Step	Remainder
0	0010	initial values	0000 0111
		shift remainder left by 1	0000 1110
1	0010	remainder = remainder – divisor	1110 1110
		(remainder<0) ⇒ shift left; r0=0	1101 1100
2	0010	remainder = remainder + divisor	1111 1100
		(remainder<0) ⇒ shift left; r0=0	1111 1000
3	0010	remainder = remainder + divisor	0001 1000
		(remainder>0) ⇒ shift left; r0=1	0011 0001
4	0010	remainder = remainder – divisor	0001 0001
		(remainder>0) ⇒ shift left; r0=1	0010 0011
done	0010	shift "left half of remainder" right by 1	0001 0011

Exercise sheet

Iteration	Divisor	Hardware design 3, non-restoring	
		Step	Remainder
0		initial values	
		shift remainder left by 1	
1			
2			
3			
4			
done		shift "left half of remainder" right by 1	

Floating-point (FP) numbers

- Computers need to deal with real numbers
  - Fractional numbers (e.g., 3.1416)
  - Very small numbers (e.g., 0.000001)
  - Very larger numbers (e.g., 2.7596×10^9)
- Components in an FP number
  - $(-1)^{\text{sign}} \times$ significand (a.k.a. *mantisa*) $\times 2^{\text{exponent}}$
  - More bits in significand gives higher accuracy
  - More bits in exponent gives wider range
- A case for FP representation standard
  - Portability issues
  - Improved implementations
  - ⇒ IEEE-754

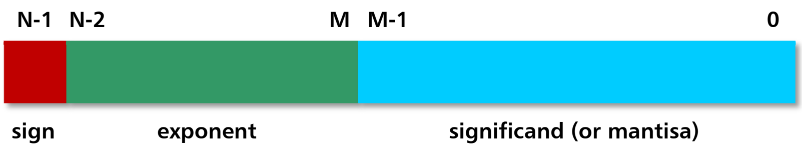
Format choice issues

- Example floating-point numbers (base-10)
  - 1.4×10^{-2}
  - $-20.0 = -2.00 \times 10^{1}$

- What components do we have?
  - Sign
  - Significand
  - Exponent
- Representing sign is easy.
- Significand is unsigned.
- Exponent is a signed integer. What method do we use?

IEEE 754

- A standard for representing FP numbers in computers
  - Single precision (32 bits): 8-bit exponent, 23-bit significand
  - Double precision (64 bits): 11-bit exponent, 52-bit significand



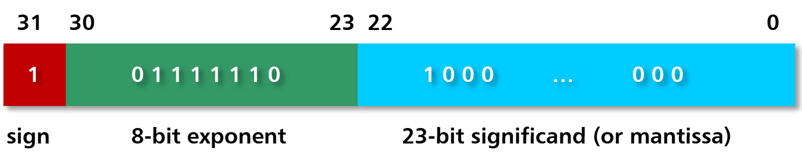
- Leading “1” in significand is implicit (why?)
- Exponent is a signed number
  - “Biased” format – for easier sorting of FP numbers
  - All 0’s is the smallest, all 1’s is the largest
  - Bias of 127 for SP and 1023 for DP
- Hence, to obtain the actual value of a representation
  - $(-1)^{sign} \times (1\#"."\#significand) \times 2^{exponent}$: here “#” is concatenation

Biased representation

- Yet another binary number representation
  - Signed number allowed
- 000…000 is the smallest number
- 111…111 is the largest number
- To get the real value, subtract a pre-determined "bias" from the unsigned evaluation of the bit pattern
- In other words, representation = value + bias
- Bias for the "exponent" field in IEEE 754
  - 127 (SP)
  - 1023 (DP)

IEEE 754 example

- -0.75_{ten}
  - Same as -3/4 or $-3/2^2$
  - In binary, $-11_{two}/2^2_{ten}$ or -0.11_{two}
  - In a normalized form, it's $-1.1_{two} \times 2^{-1}$
- In IEEE 754
  - Sign bit is 1 – number is negative!
  - Significand is 0.1 – the leading 1 is implicit!
  - Exponent is -1 – or 126 in biased representation



IEEE 754 summary

Single Precision		Double Precision		Represented Object
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	non-zero	0	non-zero	+/- denormalized number
1~254	anything	1~2046	anything	+/- floating-point numbers
255	0	2047	0	+/- infinity
255	non-zero	2047	non-zero	NaN (Not a Number)

Denormal number

- Smallest normal: 1.0×2^{Emin}
- Below, use denormal: $0.f \times 2^{Emin}$

- $e = E_{min}-1$, $f != 0$
- #00000000#######################

NaN

- Not a Number
- Result of illegal computation
  - 0/0, infinity/infinity, infinity – infinity, …
  - Any computation involving a NaN

- $e = E_{max}+1$, $f \mathrel{!}= 0$
- #11111111#######################
- Many NaN's

Values represented with IEEE 754

Type	Sign	Exponent	Significand	Value
Zero	0	0000 0000	000 0000 0000 0000 0000 0000	0.0
One	0	0111 1111	000 0000 0000 0000 0000 0000	1.0
Minus One	1	0111 1111	000 0000 0000 0000 0000 0000	−1.0
Smallest denormalized number	*	0000 0000	000 0000 0000 0000 0000 0001	$\pm 2^{-23} \times 2^{-126} = \pm 2^{-149} \approx \pm 1.4 \times 10^{-45}$
"Middle" denormalized number	*	0000 0000	100 0000 0000 0000 0000 0000	$\pm 2^{-1} \times 2^{-126} = \pm 2^{-127} \approx \pm 5.88 \times 10^{-39}$
Largest denormalized number	*	0000 0000	111 1111 1111 1111 1111 1111	$\pm(1-2^{-23}) \times 2^{-126} \approx \pm 1.18 \times 10^{-38}$
Smallest normalized number	*	0000 0001	000 0000 0000 0000 0000 0000	$\pm 2^{-126} \approx 1.18 \times 10^{-38}$
Largest normalized number	*	1111 1110	111 1111 1111 1111 1111 1111	$\pm(1-2^{-24}) \times 2^{128} \approx \pm 3.4 \times 10^{38}$
Positive infinity	0	1111 1111	000 0000 0000 0000 0000 0000	$+\infty$
Negative infinity	1	1111 1111	000 0000 0000 0000 0000 0000	$-\infty$
Not a number	*	1111 1111	non zero	NaN
* Sign bit can be either 0 or 1 .				

FP arithmetic operations

- We want to support four arithmetic functions (+, –, ×, /)

- (+, –): Must equalize exponents first. Why?
- (×, /): Multiply/divide significand, add/subtract exponents.

- Use "rounding" when result is not accurate

- Exception conditions
  - E.g., Overflow, underflow (what is underflow?)

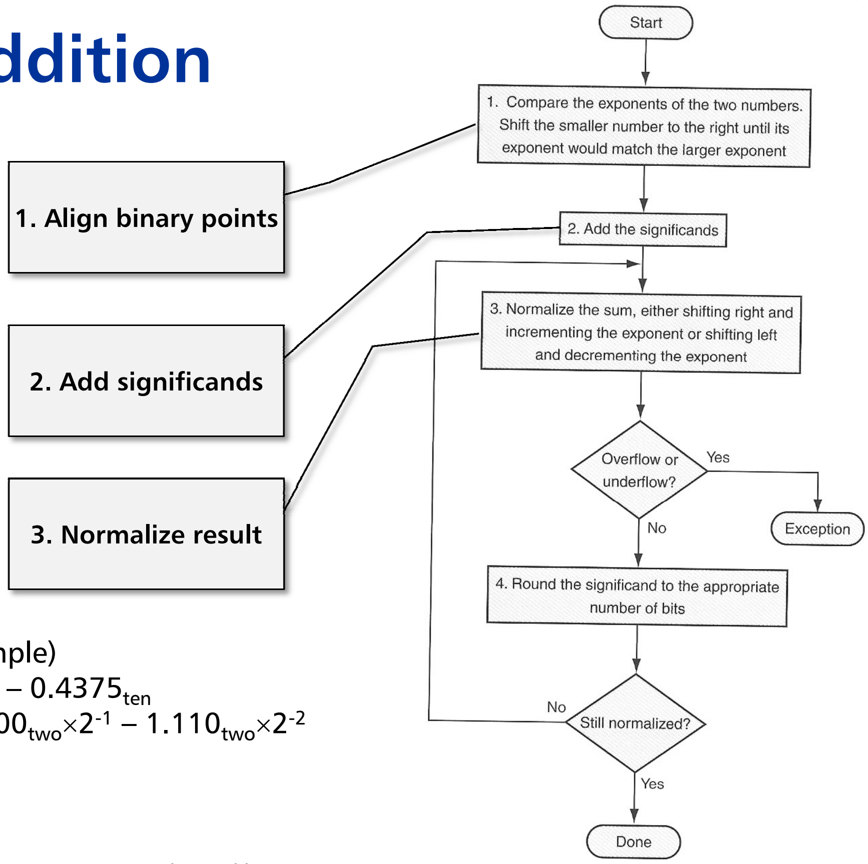
- Error conditions
  - E.g., divide-by-zero

Overflow and underflow

- Overflow
  - The exponent is too large to fit in the exponent field

- Underflow
  - The exponent is too small to fit in the exponent field

FP addition

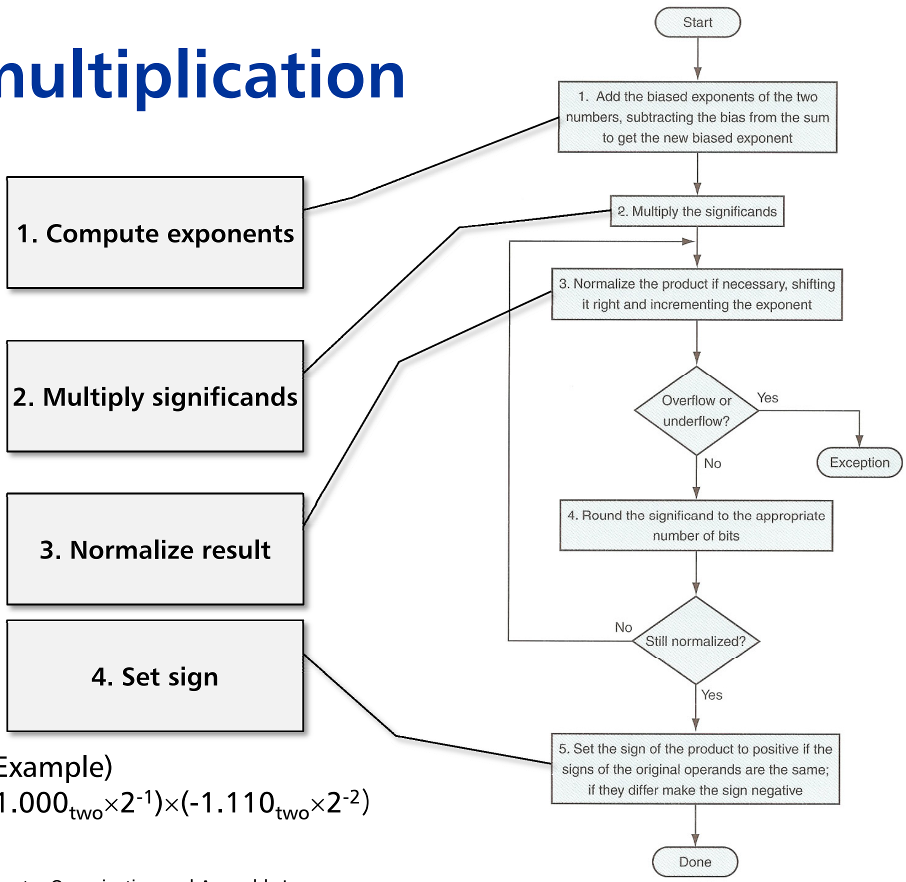


(Example)

$0.5_{ten} - 0.4375_{ten}$

$= 1.000_{two}\times 2^{-1} - 1.110_{two}\times 2^{-2}$

FP multiplication



(Example)

$(1.000_{two}\times 2^{-1})\times(-1.110_{two}\times 2^{-2})$

Accuracy and rounding

- Goal
  - IEEE 754 guarantees that the maximum error is ±½ u.l.p. compared with infinite precision
  - u.l.p. = Units in the Last Place = distance to the next floating-point value larger in magnitude

- Rounding using extra bits
  - Alignment step in the addition algorithm can cause data to be discarded (shifted out on right)
  - Multiplication step
  - IEEE 754 defines three types of extra bits: G (guard), R (round), S (sticky)

Guard bit examples

- Assume 5-bit significand

- Add $1.0000\times2^{0} + 1.1111\times2^{-2}$

- Multiply $1.0000\times2^{0} \times 1.1001\times2^{-2}$

Rounding modes

- IEEE 754 has four rounding modes
  - Round to nearest even (default)
  - Round towards plus infinity
  - Round towards minus infinity
  - Round towards 0

- Round bit is calculated to the right of Guard bit
- Sticky bit is used to determine whether there are any 1 bit truncated below Guard and Round bits
- It can be shown that “Round to nearest even” minimizes the mean error introduced by rounding

Pentium processor divide flaw

- Pentium FP divider algorithm generates multiple bits per step
  - FPU uses MSBs of divisor and dividend/remainder to guess next 2 bits of quotient
  - Guess is taken from a lookup table: -2, -1, 0, +1, +2
  - Guess is multiplied by divisor and subtracted from remainder to generate a new remainder
  - SRT division (Sweeny, Robertson, and Tocher): Used in most CPUs
- Pentium processor table = 7 bits remainder + 4 bits divisor = 11 bits, 2^{11} entries
  - 5 entries of divisors omitted: 1.0001, 1.0100, 1.0111, 1.1010, 1.1101 from the table
  - Fix is just add 5 entries back into the table
  - Eventually, it cost Intel $300M