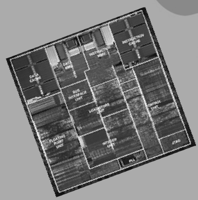
Lecture 5

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

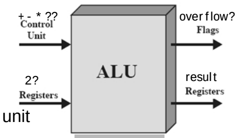
Computer Arithmetic

Ch 9 [Stal10]
Integer arithmetic
Floating-point arithmetic



ALU

- ALU = Arithmetic Logic Unit (*Aritmeettis-looginen yksikkö*)
- Actually performs operations on data
  - Integer and floating-point arithmetic
  - Comparisons (*vertailut*), left and right shifts (*sivuttaissiirrot*)
  - Copy bits from one register to another
  - Address calculations (*Osoitelaskenta*): branch and jump (*hypyt*), memory references (*muistiviittaukset)*
- Data from/to internal registers (latches)
  - Input copied from normal registers (or from memory)
  - Output goes to register (or memory)
- Operation
  - Based on instruction register, control unit



(Sta10 Fig 9.1)

Integer Representation (*kokonaislukuesitys*)

- Binary representation, bit sequence, only 0 and 1
- "Weight" of the digit based on position

$57 = 5*10^1 + 7*10^0$
$= 32 + 16 + 8 + 1$
$= 1*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0$
= 0011 1001
= 0x39 (hexadecimal)
$= 3*16^1 + 9*16^0$

- Most significant bit, MSB (*eniten merkitsevä bitti*)
- Least significant bit, LSB (*vähiten merkitsevä bitti*)

Integer Representation

- Negative numbers?
  - Sign magnitude (*Etumerkki-suuruus*)
  - Twos complement (*2:n komplementtimuoto*)

-57 = 1011 1001
-57 = 1100 0111
Sign (*etumerkki*)

- Computers use twos complement
  - Just one zero (no +0 and -0)
    - Comparison to zero easy
  - Math is easy to implement
    - No need to consider sign
    - Subtraction becomes addition
  - Simple hardware and circuit

+2 = 0000 0010
+1 = 0000 0001
0 = 0000 0000
-1 = 1111 1111
-2 = 1111 1110

Twos complement (*2:n komplementti*)

- Example
  - 8-bit sequence, value -57

57 = 0011 1001	unsigned value (*itseisarvo*)
1100 0110	invert bits (ones complement)
1100 0110	
1	add 1
1100 0111	twos complement

Reject overflow

- Easy to expand. As a 16-bit sequence

57 = 0011 1001 = 0000 0000 0011 1001
-57 = 1100 0111 = 1111 1111 1100 0111

sign extension

Twos Complement Addition

- Twos complement value range (*arvoalue*): $-2^{n-1} \dots 2^{n-1} -1$

8 bits: $-2^7 \dots 2^7-1 = -128 \dots 127$
32 bits: $-2^{31} \dots 2^{31}-1 = -2\ 147\ 483\ 648 \dots 2\ 147\ 483\ 647$

- Addition overflow (*ylivuoto*) easy to detect
  - No overflow, if different signs in operands
  - Overflow, if same sign (*etumerkki*) and the results sign differs from the operands

How would you implement this with and/or gates?

57 = 0011 1001
+ 80 = 0101 0000

137 = 1000 1001 Overflow!

Twos Complement Subtraction

- Subtraction as addition
  - Forget the sign, handle as if unsigned!
  - Complement 2nd term, the subtrahend, then add (*lisää 2:n komplementti vähentäjästä*)
  - Simple hardware

e.g., 1-3= 1 + (-3) = -2

```
3= 0011  →   1100
                1
         -3 = 1101

+1 =  0001
-3 =  1101
-2 =  1110
```

- Check
  - Overflow? (same rule as in addition)
  - sign= 1, result is negative

Twos Complement Negation

- 1: invert all bits
- 2: add 1
- 3: Special cases
  - Ignore carry bit (*ylivuotobitti*)
  - Sign really changed?
    - Cannot negate smallest negative
    - Result in exception
- Simple hardware

```
-57 = 1100 0111
      0011 1000
              1
+57 = 0011 1001
```

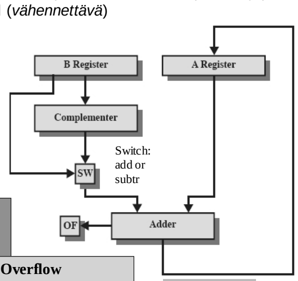
```
-128 = 1000 0000
       0111 1111
               1
       1000 0000
```


Integer Addition (and Subtraction)

- Normal binary addition
  - In subtraction: complement the 2. operand, subtrahend (*vähentäjä*) and add to 1. operand, minuend (*vähennettävä*)
- Ignore carry
  - Check sign for Overflow indication
- Simple hardware function
  - Two circuits: Complement and addition

```
-4-1=?          -4-5=?
  1100 =-4        1100 = -4
+1111 =-1       +1011 = -5
 11011 =-5       10111 = ?
```

Overflow (4-bit 2-compl ints)



(Sta10 Fig 9.6)

Integer Multiplication

- "Just like" you learned at school
  - Easy with just 0 and 1!
- Hardware?
  - Complex
  - Several algorithms
- Overflow?
  - 32 b operands → result 64 b?
- Simpler, if only unsigned numbers
  - Just multiple additions
  - Or additions and shifts
    - E.g., : 5 * => add, shift, shift, add

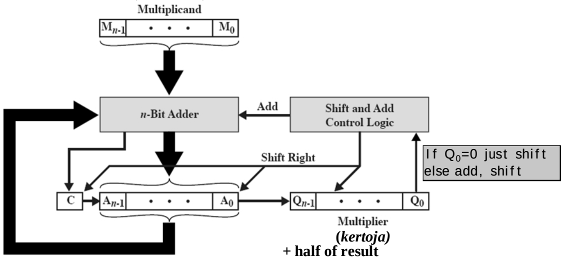
```
    1011     (kerrottava) Multiplicand (11)
   ×1101     Multiplier (13) (kertoja)
    1011  }
   0000   }  Partial products
  1011    }
 1011     }
10001111     Product (143)
```

(Sta10 Fig 9.7)

```
                          multiplier   multiplicant
Example 5*11   5=101, 11 = 1011...
101 add, shift:    add => 1011...
                 shift => 01011..
101 shift:       shift => 001011.
101 add, shift:    add => 110111.
result= 55:      shift => 0110111
```

Discussion?

Unsigned multiplication example



(Sta10 Fig 9.8a)

Unsigned multiplication

Q * M = 1101 * 1011 = 1000 1111, i.e., 13*11 = 143

C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

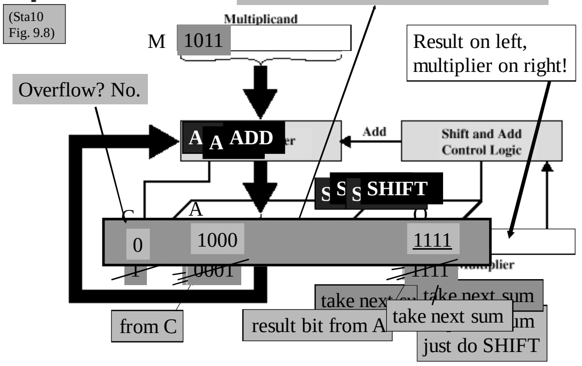
(b) Example from Figure 9.7 (product in A, Q)

(Sta10 Fig 9.8b)

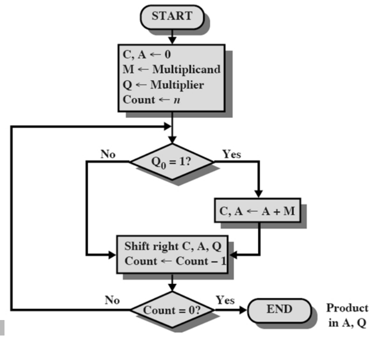
Unsigned Multiplication Example

13 * 11 = ??? = 1000 1111 = 128+8+4+2+1 = 143

(Sta10 Fig. 9.8)



Unsigned multiplication



(Sta10 Fig 9.9)

Multiplication with negative values?

- The preceeding algorithm for unsigned numbers does NOT work for negative numbers
- Could do with unsigned numbers
  - ❶ Change operands to positive values
  - ❷ Do multiplication with positive values
  - ❸ Check signs and negate the result if needed
- This works, but there are better and faster mechanisms available

Booth's Algorithm

- Unsigned multiplication:
  - Addition (only) for every "1" bit in multiplier (*kertoja*)
- Booth's algorithm (improvement)
  - Combine all adjacent 1's in multiplier together,
  - Replace all additions by one subtraction and one addition
  - Example: decimal: 7*x = 8*x +(–x)
  - Binary: 111*x = 1000*x +(-x) =
  - add, shift, shift, shift, complement, add
  - (in reality, the complement/add would be first)

5 * 7 = 0101 * 0111
= 0101 * (1000-0001)

➡

00101000 40
11111011 -5
100100011 = 35

- Works for twos complement! Also negative values!

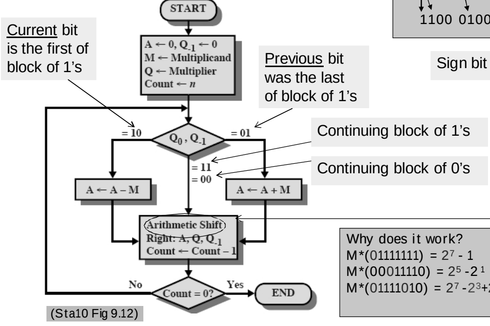
Discussion?

Booth's algorithm

Arithmetic Shift Right:
= fill with sign

1000 1000
1100 0100

Sign bit extending



(Sta10 Fig 9.12)

Why does it work?
M*(01111111) = 2^7 - 1
M*(00011110) = 2^5 -2^1
M*(01111010) = 2^7 -2^3+2^2-2^1

Booth's Algorithm, example

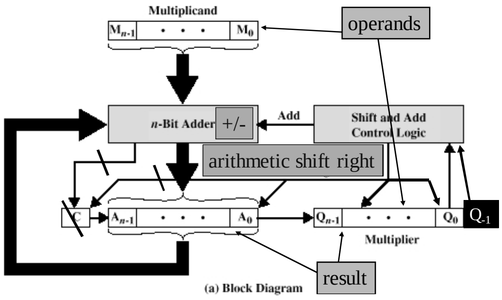
Q * M = 0011 * 0111 = 0001 0101 eli 3*7 = 21

1-0 subtract (*vähennys*)
0-1 add (*lisäys*)

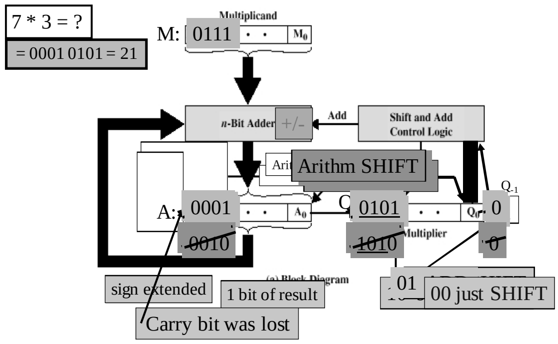
A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	A ← A - M	First Cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	A ← A + M	Third Cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth Cycle

(Sta06 Fig 9.13)

Booth's Algorithm for Twos Complement Multiplication

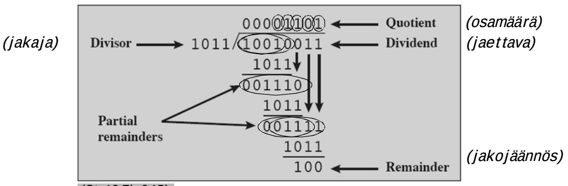


Booth's Algorithm Example



Integer division

- Like in school algorithm
  - Easy: new quotient digit always 0 or 1

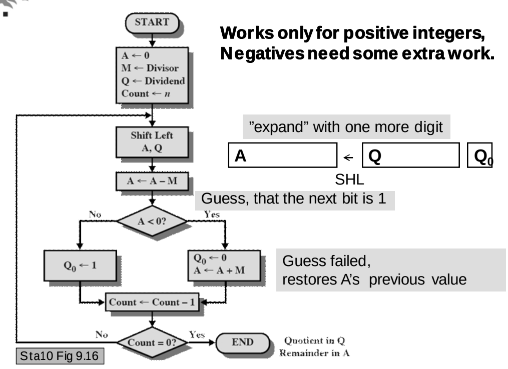


(Sta10 Fig 9.15)

- Hardware needs as in multiplication
  - Consider new digit? -- shift left (etc.)

Integer division

Works only for positive integers, Negatives need some extra work.



Sta10 Fig 9.16

Example: twos complement division

- Division: 7/3 A+ Q = 7 = 0000 0111 M= 3 = 0011

A	Q	
0000	0111	initial value
0000	1110	shift left
1101		subtract M
0000	1110	restore
0001	1100	shift left
1110		subtract M
0001	1100	restore
0011	1000	shift left
0000		subtract M
0000	1001	set Q_0=1
0001	0010	shift
1110		subtract M
0001	0010	restore

Sta10 Fig 9.17

Subtract M = Add (–M)
-M = -3 = 1101

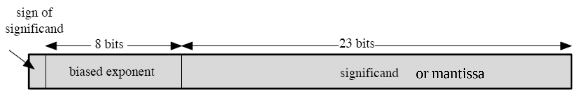
First try, if you can do the subtraction (or add if different signs). If the sign changed, subtraction failed and A must be restored, Q_0 = 0

If subtraction succesful, Q_0 = 1

Q = quotient = 2
A = remainder = 1

Repeat as many times as Q has bits.

Floating Point Representation



- Significant digits *(Merkitsevät numerot)* and exponent *(suuruusluokka)*
- Normalized number (*Normeerattu muoto*)
  - Most significant digit is nonzero >0
  - Commonly just one digit before the radix point (*desim. pilkku*)

$$-0.000\,000\,000\,123 = -1.23 * 10^{-10}$$
$$0.123 = +1.23 * 10^{-1}$$
$$123.0 = +1.23 * 10^{2}$$
$$123\,000\,000\,000\,000 = +1.23 * 10^{14}$$

IEEE 754 (floating point) formats

Parameter	Single	Single Extended	Double	Double Extended
Word width (bits)	32	≥ 43	64	≥ 79
Exponent width (bits)	8	≥ 11	11	≥ 15
Exponent bias	127	unspecified	1023	unspecified
Maximum exponent	127	≥ 1023	1023	≥ 16383
Minimum exponent	–126	≤ –1022	–1022	≤ –16382
Number range (base 10)	10^{-38}, 10^{+38}	unspecified	10^{-308}, 10^{+308}	unspecified
Significand width (bits)*	23	≥ 31	52	≥ 63
Number of exponents	254	unspecified	2046	unspecified
Number of fractions	2^{23}	unspecified	2^{52}	unspecified
Number of values	1.98×2^{31}	unspecified	1.99×2^{63}	unspecified

* not including implied bit

(Stal10 Table 9.3)

32-bit floating point

- 1 b sign
  - 1 = "-", 0 = "+"
- 8 b exponent
  - Biased representation, no sign (*Ei etumerkkiä, vaan erillinen nollataso, talletus vakiolisäyksellä*)
    - Exp=5 → store 127+5, Exp=-5 → store 127-5 (bias127)
- 23 b significant (*mantissa*)
  - In normalized form the radix point is preceeded with 1, which is not stored. (hidden bit, Zuse Z3 1939)
- The binary value of the floating point representation

-1^{Sign} * 1.Mantissa * $2^{Exponent-127}$

Example

23.0 = +10111.0 * 2^0 = +1.0111 * 2^4 =?

127+4=131

sign	exponent	mantissa
0	1000 0011	011 1000 0000 0000 0000 0000

1.0 = +1.0000 * 2^0 = ?

0+127 = 127

sign	exponent	mantissa
0	0111 1111	000 0000 0000 0000 0000 0000

Example

sign	exponent	mantissa
0	1000 0000	111 1000 0000 0000 0000 0000

X = ?

X = $(-1)^0$ * 1.1111 * $2^{(128-127)}$

= 1.1111_2 * 2

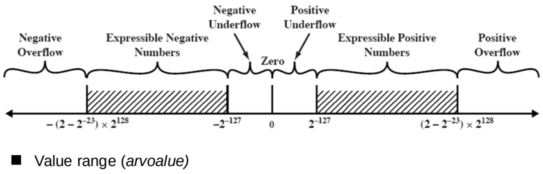
= (1+ 1/2 + 1/4 + 1/8 + 1/16) * 2

= (1 + 0.5 + 0.25 + 0.125+0.0625) * 2

= 1.9375 * 2

= 3.875

Accuracy (*tarkkuus*) (32b)



- Value range (*arvoalue)*
  - 8 b exponent→ 2^{-126} ... 2^{127} ~ -10^{-38} ... 10^{38}
- Not exact value
  - 24 b mantissa → 2^{24} ~ 1.7 * 10^{-7} ~ 6 decimals
- Balancing between range and precision

Numerical errors: Patriot Missile (1991), Ariane 5 (1996)
http://ta.twi.tudelft.nl/nw/users/vuik/wi211/disasters.html

Interpretation of IEEE 754 Floating-Point Numbers

	Single Precision (32 bits)			
	Sign	Biased exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	–0
plus infinity	0	255 (all 1s)	0	∞
minus infinity	1	255 (all 1s)	0	–∞
quiet NaN	0 or 1	255 (all 1s)	≠ 0	NaN
signaling NaN	0 or 1	255 (all 1s)	≠ 0	NaN
positive normalized nonzero	0	0 < e < 255	f	2^{e-127}(1.f)
negative normalized nonzero	1	0 < e < 255	f	-2^{e-127}(1.f)
positive denormalized	0	0	f ≠ 0	2^{e-126}(0.f)
negative denormalized	1	0	f ≠ 0	-2^{e-126}(0.f)

Not a Number

Double Precision similarly

(Sta10 Table 9.4)

NaN: Not a Number

Operation	Quiet NaN Produced by
Any	Any operation on a signaling NaN
Add or subtract	Magnitude subtraction of infinities: $(+\infty) + (-\infty)$, $(-\infty) + (+\infty)$, $(+\infty) - (+\infty)$, $(-\infty) - (-\infty)$
Multiply	$0 \times \infty$
Division	$\frac{0}{0}$ or $\frac{\infty}{\infty}$
Remainder	x REM 0 or ∞ REM y
Square root	$\sqrt{x}$ where $x < 0$

(Sta10 Table 9.6)

Floating Point Arithmetics

- Calculations need wide registers
  - Guard bits - pad right end of significand
  - More bits for the significand (mantissa)
  - Using denormalized formats
- Addition and subtraction
  - More complex than multiplication
  - Operands must have same exponent
    - Denormalize the smaller operand (alignment!)
      - Loss of digits (less precise and missing information)
  - Result (must) be normalised
- Multiplication and division
  - Significand and exponent handled separately

Floating Point Arithmetics

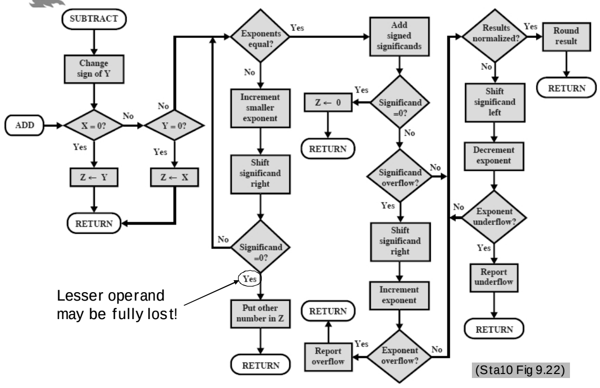
Floating Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$X + Y = \left(X_s \times B^{X_E - Y_E} + Y_s\right) \times B^{Y_E}$ $X - Y = \left(X_s \times B^{X_E - Y_E} - Y_s\right) \times B^{Y_E}$ $X_E \le Y_E$ $X \times Y = \left(X_s \times Y_s\right) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_s}{Y_s}\right) \times B^{X_E - Y_E}$

(Sta10 Table 9.5)

$X = 0.3 \times 10^2 = 30$
$Y = 0.2 \times 10^3 = 200$

$X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$
$X - Y = (0.3 \times 10^{2-3} - 0.2) \times 10^3 = (-0.17) \times 10^3 = -170$
$X \times Y = (0.3 \times 0.2) \times 10^{2+3} = 0.06 \times 10^5 = 6000$
$X \div Y = (0.3 \div 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15$

Floating Point Addition and Subtraction



(Sta10 Fig 9.22)

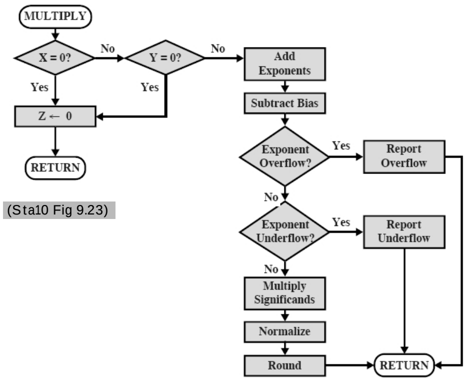
Floating Point Special Cases

- Exponent overflow (*eksponentin ylivuoto*)
  - Very large number (above max) — Programmable option
  - Value ∞ or -∞ , alternatively cause exception
- Exponent underflow (*eksponentin alivuoto*)
  - Very small number (below min) — Programmable option
  - Value 0 (or cause exception)
- Significand overflow (*mantissan ylivuoto*) — Fix it!
  - Normalise!
- Significand underflow (*mantissan alivuoto*)
  - Denormalizing may lose the significand accuracy
  - All significant bits lost? — Ooops, lost some or all data!

Floating Point Rounding (pyöristys)

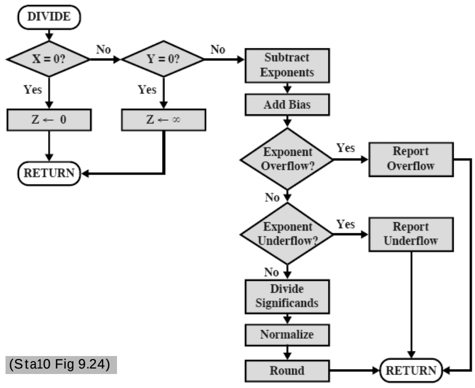
- Example
  - Value has four decimals — 3.1236, -4.5678
  - Represent it using only 3 decimals
  - Normal rounding rule round to nearest value — 3.124, -4.568
  - Always towards ∞ (*ylöspäin*) — 3.124, -4.567
  - Always towards -∞ (*alaspäin*) — 3.123, -4.568
  - Always towards 0 — 3.123, -4.567
- Intel Itanium (e.g.) supports all of these alternatives

Floating Point Multiplication



(Sta10 Fig 9.23)

Floating Point Division



(Sta10 Fig 9.24)

Computer Arithmetics Summary

- Integer ops
  - 2's complement representation
  - Negation, addition, subtraction, multiplication, division
  - Booth algorithm for multiplication
- Floating point ops
  - Complete IEEE format
    - +/- ∞, NaN, denormalized numbers, double
  - Addition, subtraction, multiplication, division
  - Overflows, underflows
  - Rounding
  - Accuracy – beware of early subtractions!

*(1.0666668-1.0666666) * 1.23456 ≠*
*1.0666668*1.23456 - 1.0666666*1.23456*

Try it out with 32-bit IEEE?

Review Questions / Kertauskysymyksiä

- Why we use twos complement?
- How does twos complement "expand" to a large number of bits (8b → 16 b)?
- Format of single-precision floating point number?
- When does underflow happen?
- When can you lose accuracy?