Computer Architectures

Number Representation and Computer Arithmetics

Pavel Píša, Michal Štepanovský, Miroslav Šnorek

Czech Technical University in Prague, Faculty of Electrical Engineering

English version partially supported by:
European Social Fund Prague & EU: We invests in your future.

Reasons to study computer architectures

- To invent/design new computer architectures
- To be able to integrate selected architecture into silicon
- To gain knowledge required to design computer hardware/systems (big ones or embedded)
- To understand generic questions about computers, architectures and performance of various architectures
- **To understand how to use computer hardware efficiently** (i.e. how to write good software)
  - It is not possible to efficiently use resources provided by any (especially by modern) hardware without insight into their constraints, resource limits and behavior
  - It is possible to write some well paid applications without real understanding but this requires abundant resources on the hardware level. But no interesting and demanding tasks can be solved without this understanding.

More motivation and examples

- The knowledge is necessary for every programmer who wants to work with medium size data sets or solve little more demanding computational tasks
- No multimedia algorithm can be implemented well without this knowledge
- The 1/3 of the course is focussed even on peripheral access
- Examples
  - Facebook – HipHop for PHP → C++/GCC → machine code
  - BackBerry (RIM) – our consultations for time source
  - RedHat – JAVA JIT for ARM for future servers generation
  - Multimedia and CUDA computations

The course's background and literature

- Course is based on worldwide recognized book and courses

  Paterson, D., Henessy, J.: Computer Organization and Design, The HW/SW Interface. Elsevier, ISBN: 978-0-12-370606-5

  - John L. Henessy – president of Stanford University, one of founders of MIPS Computer Systems Inc.
  - David A. Patterson – leader of Berkeley RISC project and RAID disks research
- Our experience even includes distributed systems, embedded systems design (of mobile phone like complexity), peripherals design, cooperation with carmakers, medical and robotics systems design

Topics of the lectures

- Architecture, structure and organization of computers and its subsystems.
- Central Processing Unit (CPU)
- Memory
- Pipelined instruction execution
- Input/output subsystem of the computer
- Input/output subsystem (part 2)
- External events processing and protection
- Processors and computers networks
- Parameter passing
- Classic register memory-oriented CISC architecture
- INTEL x86 processor family
- CPU concepts development (RISC/CISC) and examples
- Multi-level computer organization, virtual machines
- Analog and digital I/O interfacing

The 1. lecture contents

- Number representation in computers
  - numeral systems
  - integer numbers, unsigned and signed
  - floating point representation for real numbers
  - boolean values
- Basic arithmetic operations and their implementation
  - addition, subtraction
  - shift right/left
  - multiplication and division

Motivation: What is the output of next code snippet?

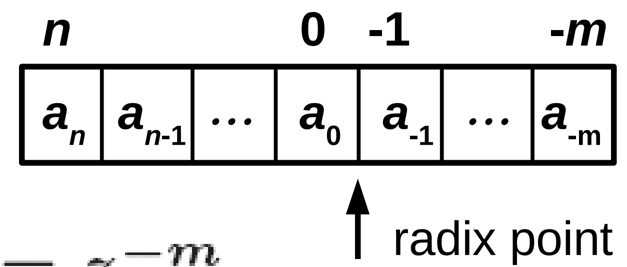
```
int main() {
  int a = -200;
  printf("value: %u = %d = %f = %c \n", a, a,
  *((float*)(&a)), a);

  return 0;
}
```

```
value: 4294967096 = -200 = nan = 8
```

and memory content is: 0x38 0xff 0xff 0xff
when run on little endian 32 bit CPU.

Terminology basics



- Positional (place-value) notation
- Decimal/radix point
- **z** … base of numeral system
- smallest representable number $\varepsilon = z^{-m}$
- **Module** = $\mathcal{Z}$, one increment/unit higher than biggest representable number in the given grid
- **A**, the representable number for given grid, where **k** is natural number in range **⟨0,z^{n+m+1} -1⟩** $\boxed{0 \leq A = k \cdot \varepsilon < \mathcal{Z}}$
- The representation and value

$$A \sim a_n a_{n-1} \ldots a_0, a_1 \ldots a_{-m}$$
$$A = a_n z^n + a_{n-1} z^{n-1} + \ldots + a_0 + a_1 z^{-1} \ldots a_{-m} z^{-m}$$

Integer number representation (unsigned, non-negative)

- The most common numeral system base in computers is z=2
- The value of $\mathbf{a}_i$ is in range {0,1,…z-1}, i.e. {0,1} for base 2
- This maps to true/false and unit of information (bit)
- We can represent number **0** … $\mathbf{2^n-1}$ when **n** bits are used
- Which range can be represented by one byte?

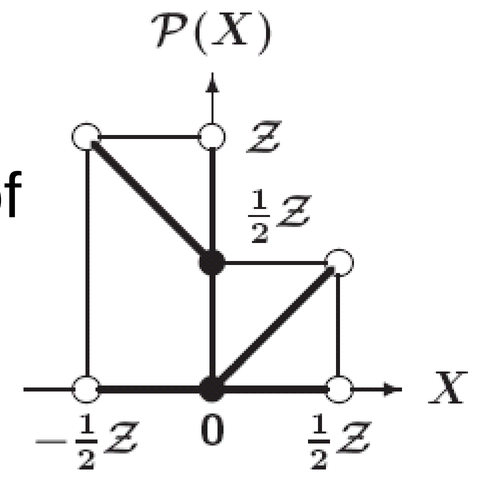
  1B (byte) … 8 bits, $2^8 = 256_d$ combinations, values 0 … $255_d = 0b11111111_b$
- Use of multiple consecutive bytes
  - 2B … $2^{16} = 65536_d$, 0 … $65535_d = 0xFFFF_h$,(h … hexadecimal, base 16, a in range 0, … 9, A, B, C, D, E, F)
  - 4B … $2^{32} = 4294967296_d$, 0 … $4294967295_d = 0xFFFFFFFF_h$

Signed integer numbers

- Work with negative numbers is required for many applications
- When appropriate representation is used then same hardware (with minor extension) can be used for many operations with signed and unsigned numbers
- Possible representations
  - sign-magnitude code, direct representation, sign bit
  - two's complement
  - ones' complement
  - excess-K, offset binary or biased representation

Integer – sign-magnitude code

- Sign and magnitude of the value (absolute value)
- Natural to humans -1234, 1234
- One (usually most significant – MSB) bit of the memory location is used to represent the sign
- Bit has to be mapped to meaning
- Common use 0 ≈ “+”, 1 ≈ “-”
- Disadvantages:
  - When location is **k** bits long then only **k-1** bits hold magnitude and each operation has to separate sign and magnitude
  - Two representations of the value 0



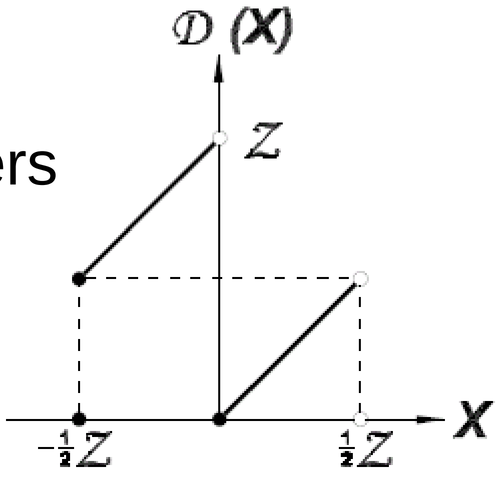
$-2^{n-1}+1 \ldots 0 \ldots 2^{n-1}-1$

Integer – two's complement

- Other option is to designate one half of range/combinations for non-negative numbers and other one for positive numbers
- Transform to the representation

  D(A) = A iff A≥0

  D(A) = Z-|A| iff A<0
- Advantages
  - Continuous range when cyclic arithmetics is considered
  - Single and one to one mapping of value 0
  - Same HW for signed and unsigned adder
- Disadvantage
  - Asymmetric range (-(-1/2Z))



-2^{n-1} … 0 … 2^{n-1}-1

Integers – ones' complement

- Transform to the representation

  $-2^{n-1}+1 \dots 0 \dots 2^{n-1}-1$

  $D(A) = A$ iff $A \geq 0$

  $D(A) = Z-1-|A|$ iff $A<0$ (i.e. subtract from all ones)
- Advantages
  - Symmetric range
  - Almost continuous, requires hot one addition when sign changes
- Disadvantage
  - Two representations of value 0
  - More complex hardware
- Negate (-A) value can be computed by bitwise complement (flipping) of each bit in representation

Integer – biased representation

- Known as excess-K or offset binary as well
- Transform to the representation

  $-K \dots 0 \dots 2^n-1-K$

  $D(A) = A+K$
- Usually K=Z/2
- Advantages
  - Preserves order of original set in mapped set/representation
- Disadvantages
  - Needs adjustment by -K after addition and +K after subtraction processed by unsigned arithmetic unit
  - Requires full transformation before and after multiplication

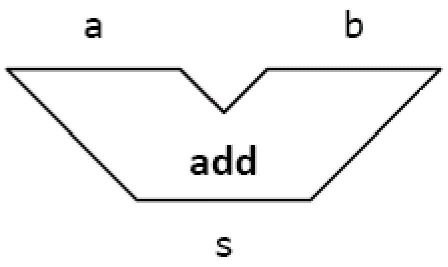
Back to two's complement and the C language

- Two's complement is most used signed integer numbers representation in computers
- Complement arithmetic is often used as its synonym
- “C” programing language speaks about integer numeric type without sign as *unsigned integers* and they are declared in source code as `unsigned int`.
- The numeric type with sign is simply called *integers* and is declared as `signed int`.
- Examples of the values representations when 32 bits are used:
  - 0_D = 00000000_H,
  - 1_D = 00000001_H, -1_D = $FFFFFFFF_H$,
  - 2_D = 00000002_H, -2_D = $FFFFFFFE_H$,
  - 3_D = 00000003_H, -3_D = $FFFFFFFD_H$,
- Considerations about value overflow and underflow from order grit are discussed later.

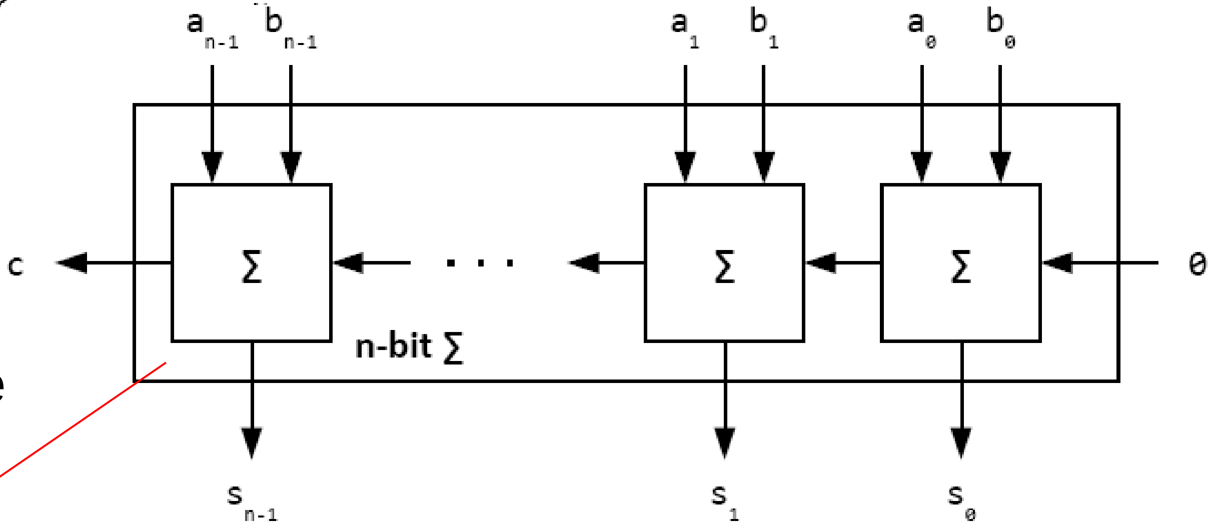
Two's complement – addition and subtraction

- **Addition**
  - 0000000 0000 0111$_B$ ≈ 7_D Symbols use: 0=0_H, 0=0_B
  - + 0000000 0000 0110$_B$ ≈ 6_D
  - 0000000 0000 1011$_B$ ≈ 13_D
- **Subtraction** can be realized as addition of negated number
  - 0000000 0000 0111$_B$ ≈ 7_D
  - + FFFFFFF 1111 1010$_B$ ≈ -6_D
  - 0000000 0000 0001$_B$ ≈ 1_D
- Question for revision: how to obtain negated number in two's complement binary arithmetics?

Hardware of ripple-carry adder

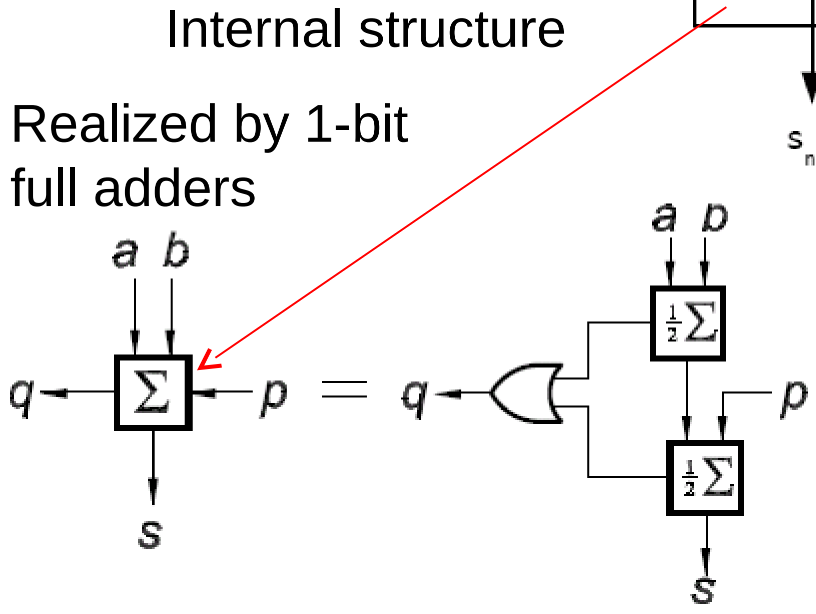


Common symbol for adder

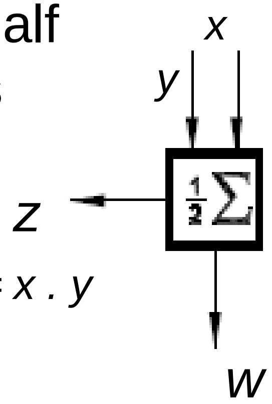


Internal structure

Realized by 1-bit full adders



where half adder is



$z = x \,.\, y$

$w = x \oplus y$

Fast parallel adder realization and limits

- The previous, cascade based adder is slow – carry propagation delay
- The parallel adder is combinatorial circuit, it can be realized through sum of minterms (product of sums), two levels of gates (wide number of inputs required)
- But for 64-bit adder 10^{20} gates is required

Solution #1

- Use of carry-lookahead circuits in adder combined with adders without carry bit

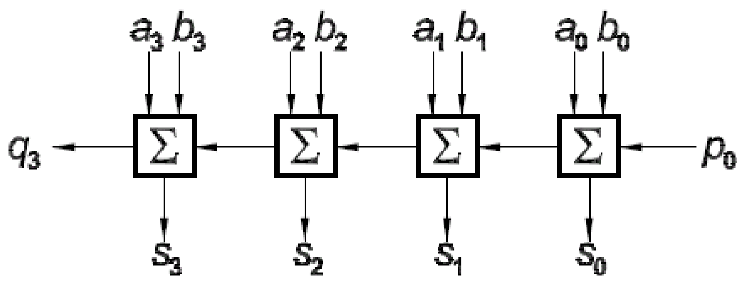
Solution #2

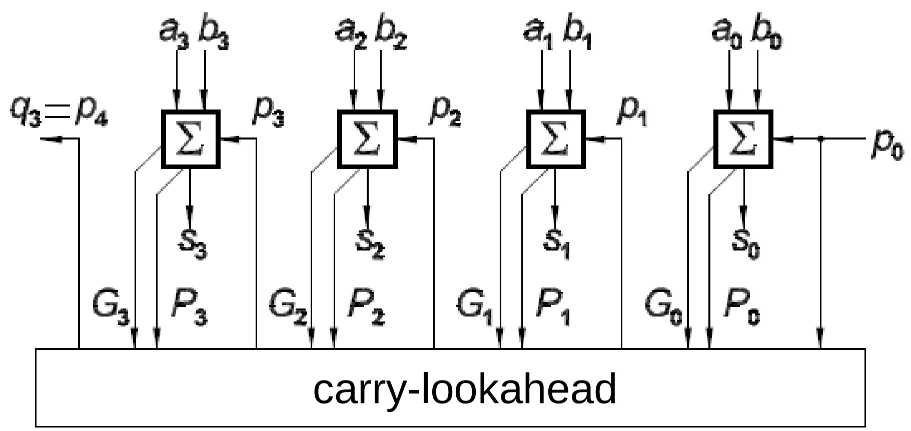
- Cascade of adders with fraction of the required width

Combination (hierarchy) of #1 and #2 can be used for wider inputs

Speed of the adder

- Parallel adder is combinational logic/circuit. Is there any reason to speak about its speed? Try to describe!
- Yes, and it is really slow. Why?
- Possible enhancement – adder with carry-lookahead (CLA) logic!





CLA – carry-lookahead

- Adder combined with CLA provides enough speedup when compared with parallel ripple-carry adder and yet number of additional gates is acceptable
- CLA for 64-bit adder increases hardware price for about 50% but the speed is increased (signal propagation time decreased) 9 times.
- The result is significant speed/price ratio enhancement.

The basic equations for the CLA logic

- Let:
  - the generation of carry on position (bit) j is defined as:

$$g_j = x_j y_j$$

  - the need for carry propagation from previous bit:

$$p_j = x_j \oplus y_j = x_j \bar{y}_j \vee \bar{x}_j y_j$$

- Then:
  - the result of sum for bit j is given by:

$$s_j = c_j \left(\overline{x_j \oplus y_j} \right) \vee \bar{c}_j \left(x_j \oplus y_j \right) = c_j \bar{p}_j \vee \bar{c}_j p_j = p_j \oplus c_j$$

  - and carry to the higher order bit (j+1) is given by:

$$c_{j+1} = x_j y_j \vee \left(x_j \oplus y_j \right) c_j = g_j \vee p_j c_j$$

CLA

The carry can be computed as:

$c_1 = g_0 \vee p_0c_0$

$c_2 = g_1 \vee p_1c_1 = g_1 \vee p_1(g_0 \vee p_0c_0) = g_1 \vee p_1g_0 \vee p_1p_0c_0$

$c_3 = g_2 \vee p_2c_2 = g_2 \vee p_2(g_1 \vee p_1g_0 \vee p_1p_0c_0) = g_2 \vee p_2g_1 \vee p_2p_1g_0 \vee p_2p_1p_0c_0$

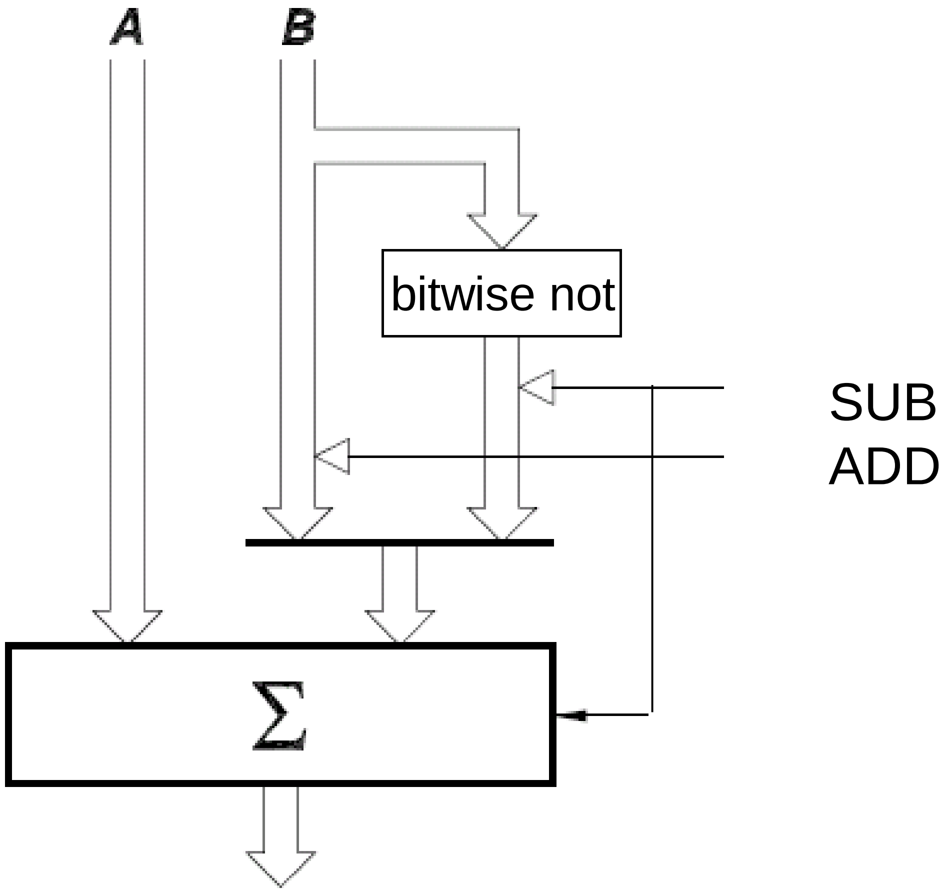
$c_4 = g_3 \vee p_3c_3 = \ldots = g_3 \vee p_3g_2 \vee p_3p_2g_1 \vee p_3p_2p_1g_0 \vee p_3p_2p_1p_0c_0$

$c_5 = \ldots$

Description of the equation for c_3 as an example:

The carry input for bit 3 is active **when** carry is generated in bit 2 **or** carry propagates condition holds for bit 2 and carry is generated in the bit 1 **or** both bits 2 and 1 propagate carry and carry is generated in bit 0

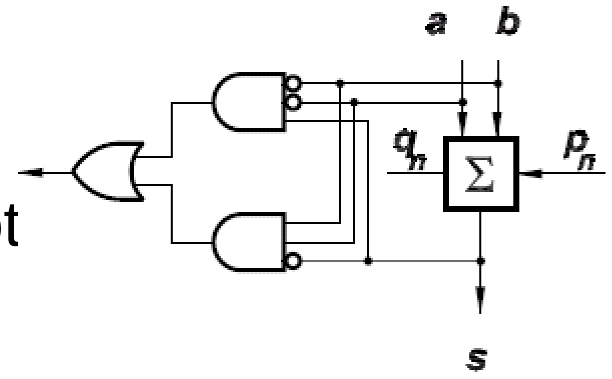
Arithmetic unit for add/subtract operations

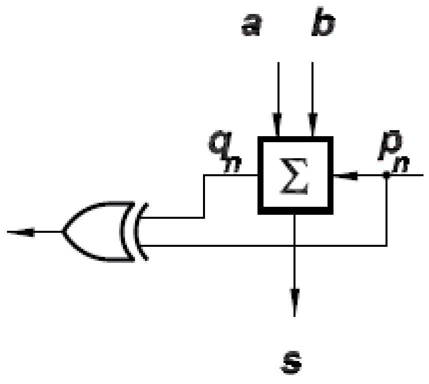


Inspiration: X36JPO, A. Pluháček

Arithmetic overflow (underflow)

- Result of the arithmetic operation is incorrect because, it does not fit into given fraction grid (number of the representation bits)
- But for the signed arithmetics, it is not equivalent to the carry from the most significant bit.
- The arithmetic overflow is signaled if result sign is different from operand signs if both operand has same sign
- or can be detected with exclusive-OR of carry to and from the most significant bit

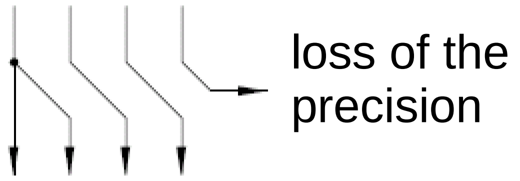


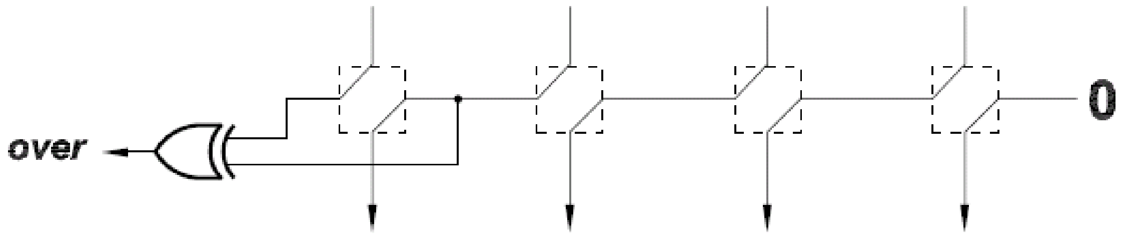


Arithmetic shift to the left and to the right

- arithmetic shift by one to the left/right is equivalent to signed multiply/divide by 2 (movement in the fraction grid)

- Notice difference between arithmetic, logic and cyclic shift operations





- Remark: Barrel shifter can be used for fast variable shifts

Addition and subtraction for the biased representation

- Short note about other signed number representation

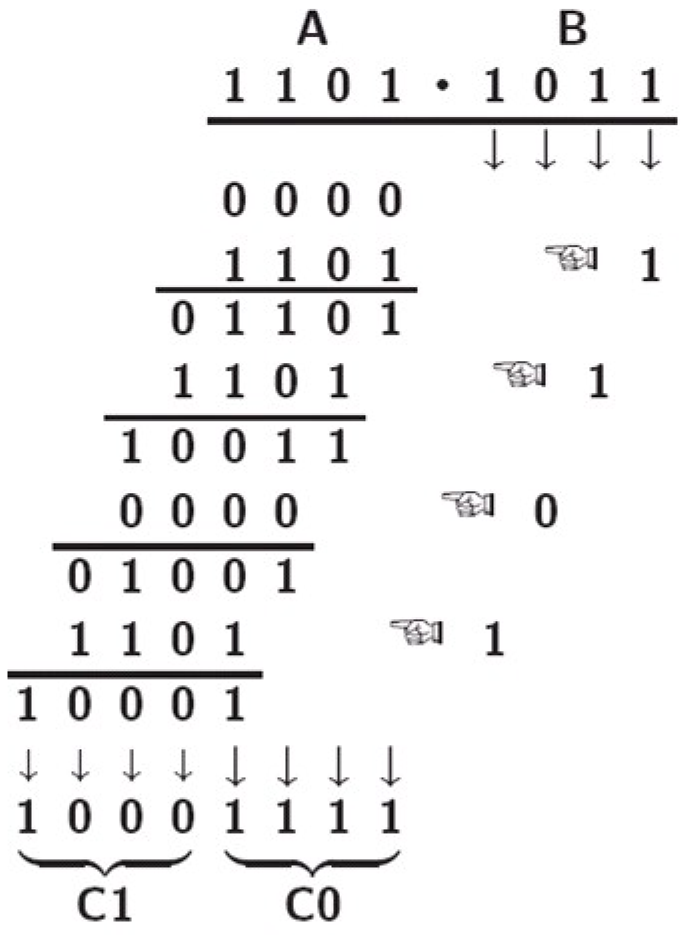
$$\mathcal{A}(A + B) = \mathcal{A}(A) + \mathcal{A}(B) - K$$
$$\mathcal{A}(A - B) = \mathcal{A}(A) - \mathcal{A}(B) + K$$

- Overflow detection
  - for addition:

    same sign of addends and different result sign
  - for subtraction:

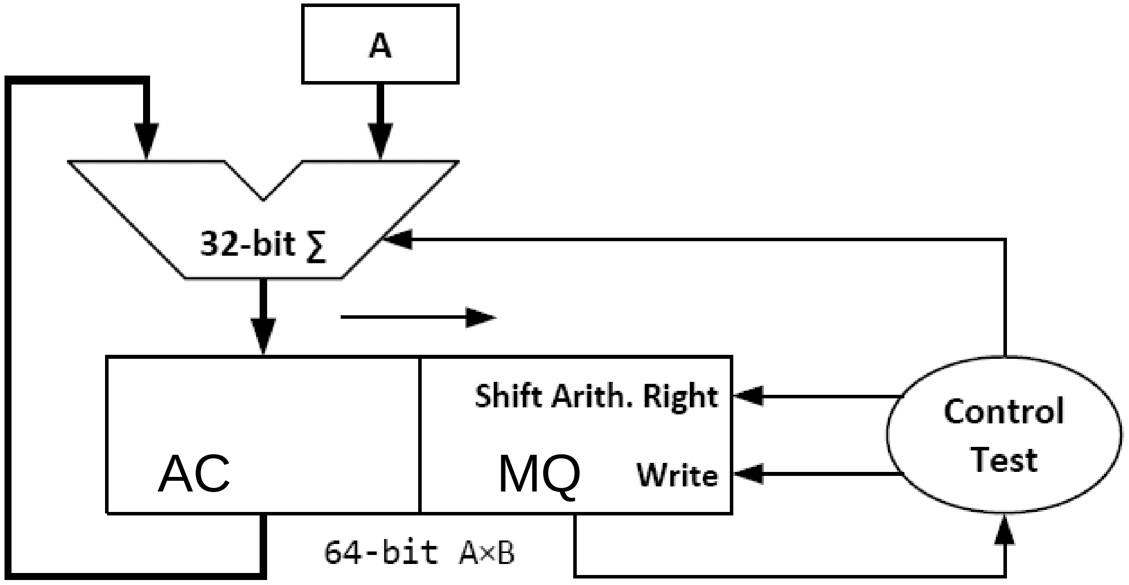
    signs of minuend and subtrahend are opposite and sign of the result is opposite to the sign of minuend

Unsigned binary numbers multiplication



Sequential hardware multiplier (32b case)

```
           A         B
        1 1 0 1 · 1 0 1 1
        -----------------
                  ↓ ↓ ↓ ↓
        0 0 0 0
        1 1 0 1         1
      ---------
      0 1 1 0 1
      1 1 0 1         1
    ---------
    1 0 0 1 1
    0 0 0 0         0
  ---------
  0 1 0 0 1
  1 1 0 1         1
---------
1 0 0 0 1
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
1 0 0 0 1 1 1 1
   C1      C0
```



The speed of the multiplier is horrible

Algorithm for multiplication

```
A = multiplicand;
MQ = multiplier;
AC = 0;

for( int i=1; i <= n; i++)    //  n – represents number of bits
{
   if(MQ_0 = = 1)  AC = AC + A;   //  MQ_0 = LSB of MQ

   SR (shift AC MQ by one bit right and insert information about
carry from the MSB from previous step)
}
end.

when loop ends AC MQ holds 64-bit result
```

Example of the multiply X by Y

Multiplicand x=110 and multiplier y=101.

i	operation	AC	MQ	A	comment
		000	101	110	initial setup
1	AC = AC+MB	110	101		start of the cycle
	SR	011	010		
2	nothing	011	010		because of $MQ_0 == 0$
	SR	001	101		
3	AC = AC+MB	111	101		
	SR	011	110		end of the cycle

The whole operation: x×y = 110×101 = 011110, (6×5 = 30)

Signed multiplication by unsigned HW for two's complement

One possible solution

$C = A \cdot B$

Let A and B representations are n bits and result is 2n bits

$D(C) = D(A) \cdot D(B)$

– (D(B)<<n)   if $A < 0$

– (D(A)<<n)   if $B < 0$

Consider for negative numbers

$(2^n+A) \cdot (2^n+B) = 2^{2n}+2^n A + 2^n B + A \cdot B$

where 2^{2n} is out of the result representation, next two elements have to be eliminated if input is negative

Wallace tree based multiplier

Q=X .Y, X and Y are considered as and 8bit unsigned numbers

(x_7 x_6 x_5 x_4 $x\text{-}_3$ x_2 x_1 x_0). (y_7 y_6 y_5 y_4 y_3 y_2 y_1 y_0) =

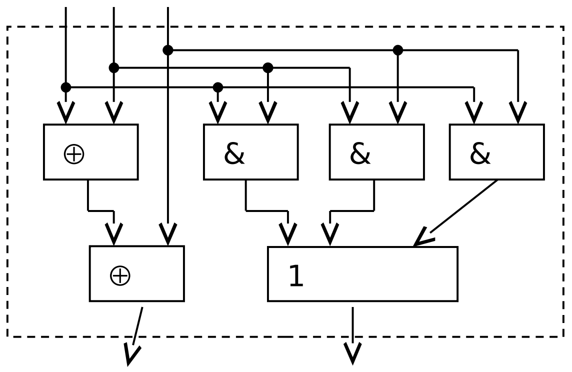
0	0	0	0	0	0	0	0	x_7y_0	x_6y_0	x_5y_0	x_4y_0	x_3y_0	x_2y_0	x_1y_0	x_0y_0	P0
0	0	0	0	0	0	0	x_7y_1	x_6y_1	x_5y_1	x_4y_1	x_3y_1	x_2y_1	x_1y_1	x_0y_1	0	P1
0	0	0	0	0	0	x_7y_2	x_6y_2	x_5y_2	x_4y_2	x_3y_2	x_2y_2	x_1y_2	x_0y_2	0	0	P2
0	0	0	0	0	x_7y_3	x_6y_3	x_5y_3	x_4y_3	x_3y_3	x_2y_3	x_1y_3	x_0y_3	0	0	0	P3
0	0	0	0	x_7y_4	x_6y_4	x_5y_4	x_4y_4	x_3y_4	x_2y_4	x_1y_4	x_0y_4	0	0	0	0	P4
0	0	0	x_7y_5	x_6y_5	x_5y_5	x_4y_5	x_3y_5	x_2y_5	x_1y_5	x_0y_5	0	0	0	0	0	P5
0	0	x_7y_6	x_6y_6	x_5y_6	x_4y_6	x_3y_6	x_2y_6	x_1y_6	x_0y_6	0	0	0	0	0	0	P6
0	x_7y_7	x_6y_7	x_5y_7	x_4y_7	x_3y_7	x_2y_7	x_1y_7	x_0y_7	0	0	0	0	0	0	0	P7
Q_{15}	Q_{14}	Q_{13}	Q_{12}	Q_{11}	Q_{10}	Q_9	Q_8	Q_7	Q_6	Q_5	Q_4	Q_3	Q_2	Q_1	Q_0	

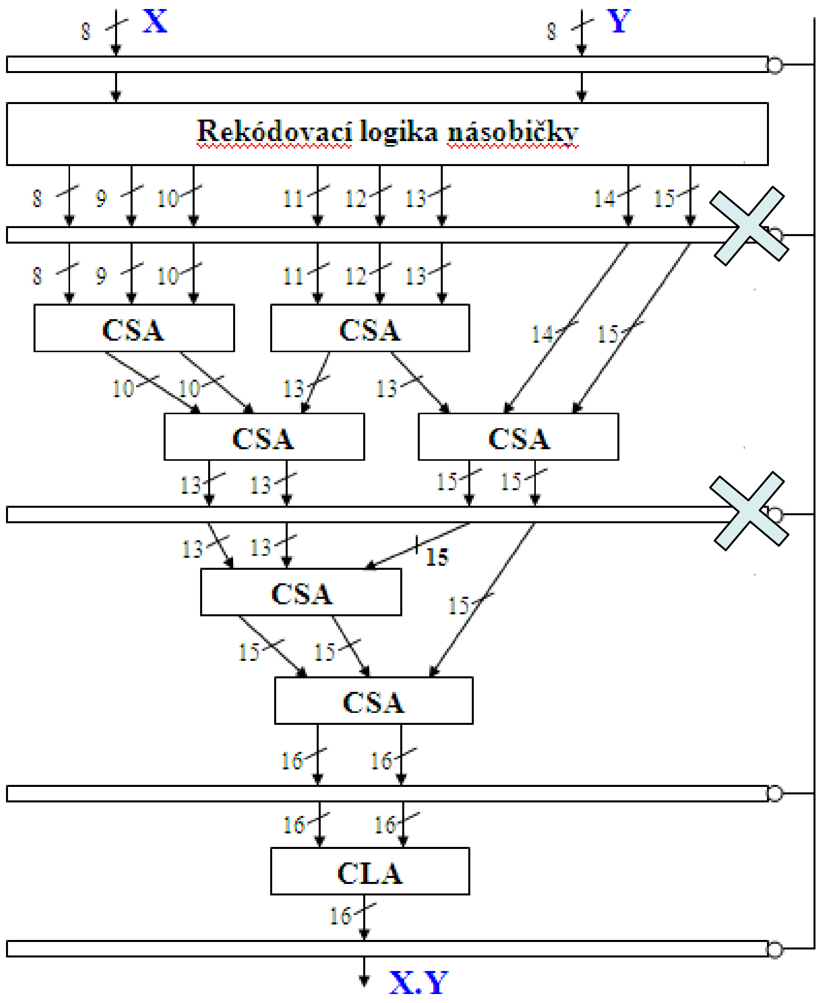
The sum of P0+P1+...+P7 gives result of X and Y multiplication.

Q = X .Y = P0 + P1 + ... + P7

Wallace tree based fast multiplier

The basic element is an CSA circuit (Carry Save Adder)





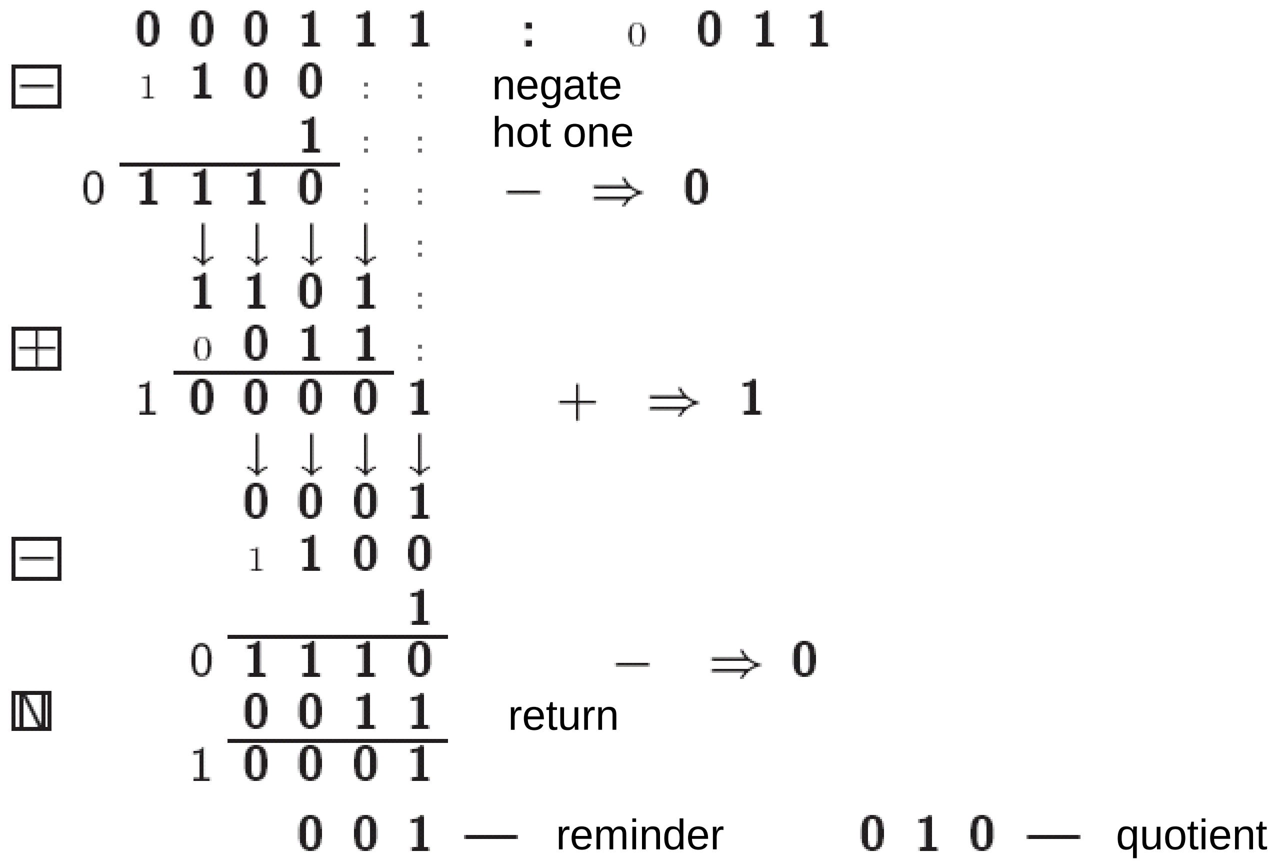
$$S = S^b + C$$

$$S^b_i = x_i \oplus y_i \oplus z_i$$

$$C_{i+1} = x_i y_i + y_i z_i + z_i x_i$$

Hardware divider

$\boxed{1\,1\,1\ :\ 0\,1\,1}$

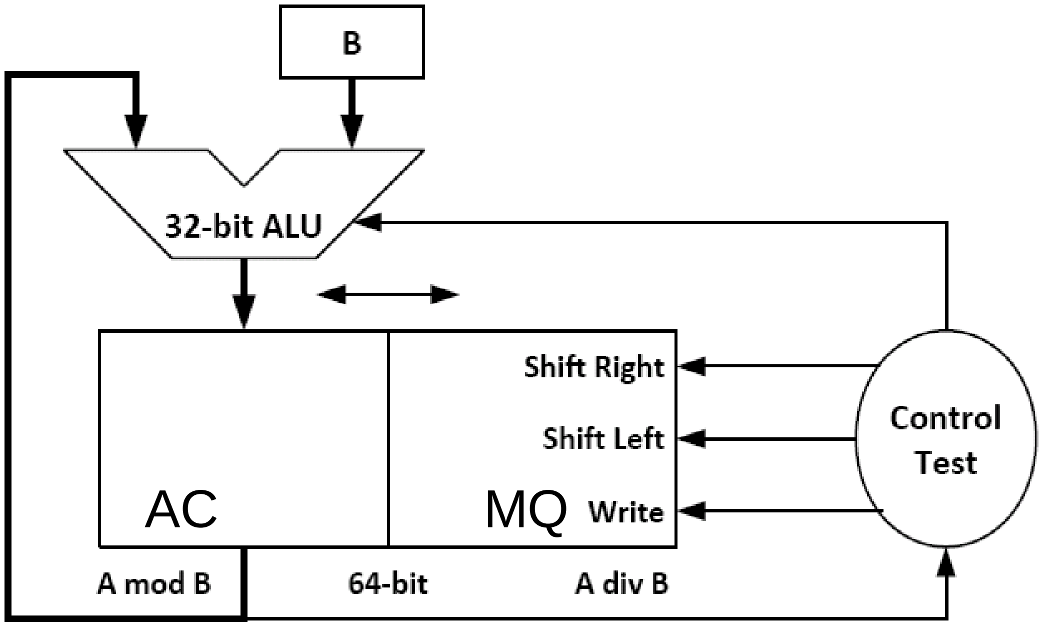


Hardware divider logic (32b case)

$\boxed{1\ 1\ 1\ :\ 0\ 1\ 1}$ divident = quotient × divisor + reminder

```
      0 0 0 1 1 1     :    0 0 1 1
⊟       1 1 0 0   :   :    negate
              1   :   :    hot one
    0 1 1 1 0     :   :    −   ⇒   0
      ↓ ↓ ↓ ↓         :
      1 1 0 1         :
⊞     0 0 1 1         :
    1 0 0 0 0 1                +   ⇒   1
        ↓ ↓ ↓ ↓
        0 0 0 1
⊟       1 1 0 0
              1
      0 1 1 1 0                  −   ⇒   0
N       0 0 1 1    return
      1 0 0 0 1

          0 0 1 — reminder          0 1 0 — quotient
```



Algorithm of the sequential division

```
MQ = dividend;
B = divisor; (Condition: divisor is not 0!)
AC = 0;

for( int i=1; i <= n; i++)          {
   SL (shift AC MQ by one bit to the left, the LSB bit is kept on zero)
   if(AC >= B)   {
          AC = AC – B;
          MQ_0 = 1;         // the LSB of the MQ register is set to 1
    }
}
```

$\rightarrow$ Value of MQ register represents quotient and AC remainder

Example of X/Y division

Dividend x=1010 and divisor y=0011

i	operation	AC	MQ	B	comment
		0000	1010	0011	initial setup
1	**SL**	0001	0100		
	nothing	0001	0100		the if condition not true
2	**SL**	0010	1000		
		0010	1000		the if condition not true
3	**SL**	0101	0000		$r \geq y$
	AC = AC - B; MQ_0 = 1;	0010	0001		
4	**SL**	0100	0010		$r \geq y$
	AC = AC - B; MQ_0 = 1;	0001	0011		end of the cycle

x : y = 1010 : 0011 = 0011 reminder 0001,   (10 : 3 = 3 reminder 1)

Higher dynamic range for numbers (REAL/float)

- Scientific notation, semilogarithmic, floating point
  - The value is represented by:
    - EXPONENT (E) – represents scale for given value
    - MANTISSA (M) – represents value in that scale
    - the sign(s) are usually separated as well
- Normalized notation
  - The exponent and mantissa are adjusted such way, that mantissa is held in some standard range. ⟨0.5, 1) or ⟨1, 2) for considered base z=2
  - Generally: the first digit is non-zero or mantissa range is ⟨1, z)

Standardized format for REAL type numbers

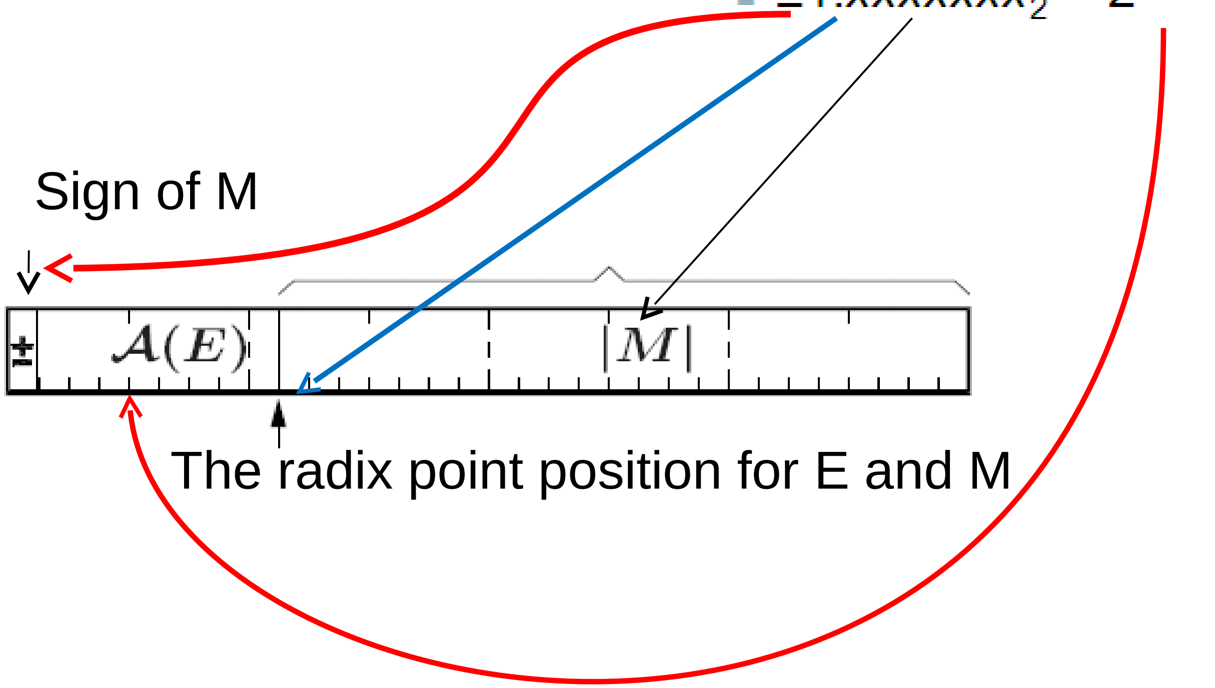
- Standard IEEE-754 defines next REAL representation and precision
  - single-precision – in the C language declared as `float`
  - double-precision – C language `double`

Examples of (de)normalized numbers in base 10 and 2

- -2.34×10^{56} ← normalized
- $+0.002 \times 10^{-4}$ ← not normalized
- $+987.02 \times 10^{9}$ ← not normalized

binary

- $\pm 1.xxxxxxx_2 \times 2^{yyyy}$



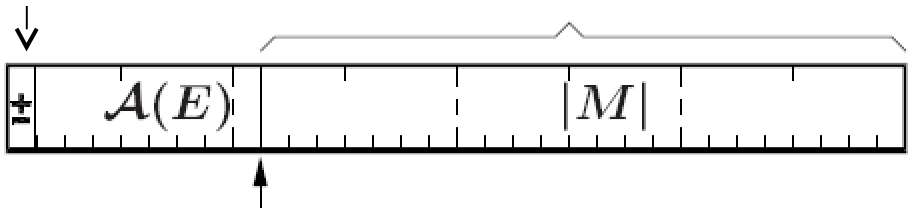
The representation/encoding of floating point number

- Mantissa encoded as the sign and absolute value (magnitude) – equivalent to the direct representation
- Exponent encoded in biased representation (K=127 for single precision)
- The implicit leading one can be ommited due to normalization of m ∈ ⟨1, 2) – 23+1 implicit bit for single

$X = -1^{s}\, 2^{A(E)-127}\, m$ where $m \in \langle 1, 2)$

$m = 1 + 2^{-23} M$

Sign of M



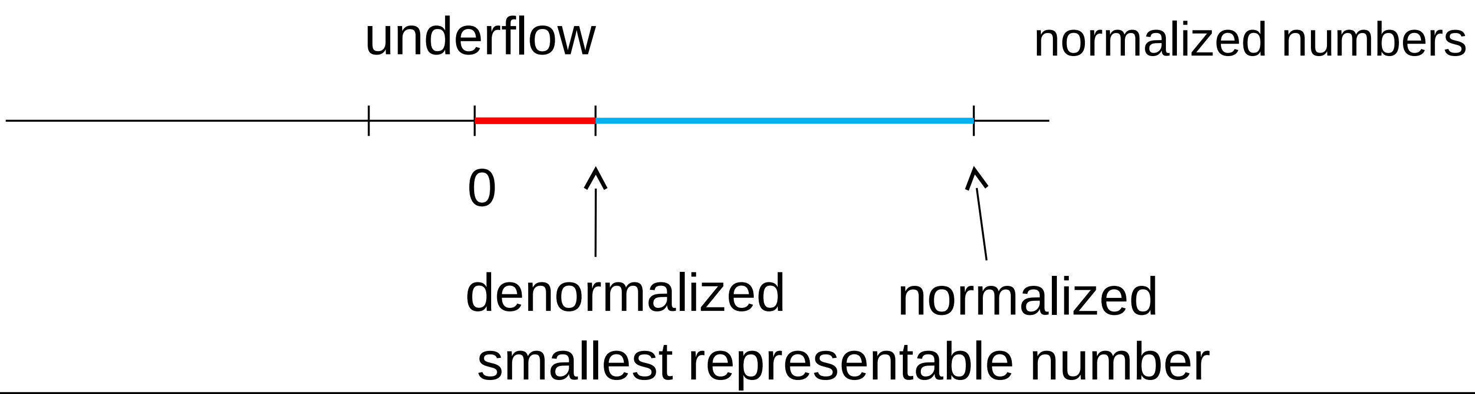
Radix point position for E and M

Implied (hidden) leading 1 bit

- Most significant bit of the mantissa is one for each normalized number and it is not stored in the representation for the normalized numbers
- If exponent representation is zero then encoded value is zero or denormalized number which requires to store most significant bit
- Denormalized numbers allow to keep resolution in the range from the smallest normalized number to zero

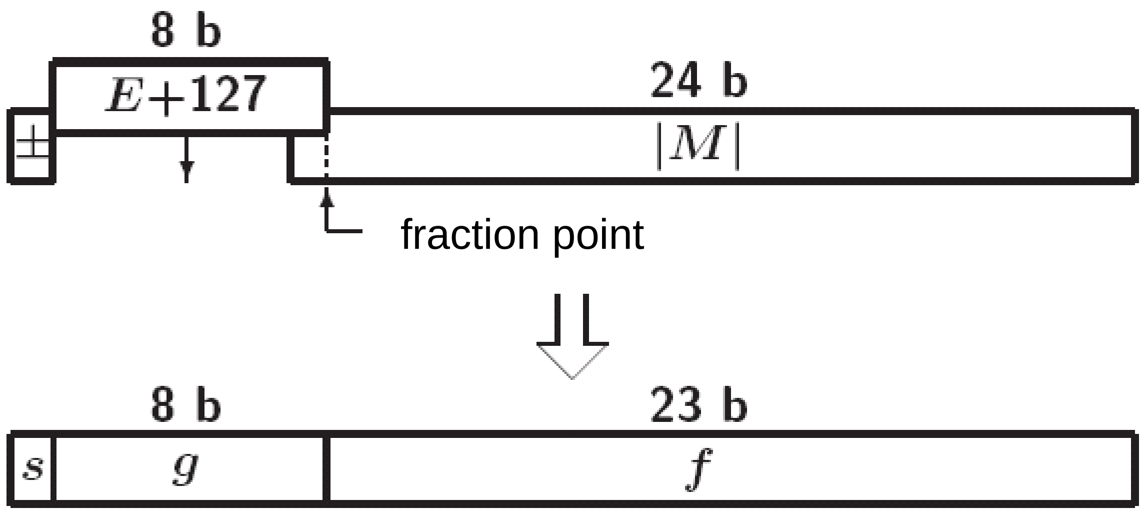
Underflow/lost of the precision for IEEE-754 representation

- The case where stored number value is not zero but it is smaller than smallest number which can be represented in the normalized form
- The direct underflow to the zero can be prevented by extension of the representation range by denormalized numbers



ANSI/IEEE Std 754-1985 – 32b a 64b formats

ANSI/IEEE Std 754-1985 — single precision format — 32b



ANSI/IEEE Std 754-1985 — double precision format — 64b

g . . . 11b f . . . 52b

Representation of the fundamental values

Zero

Positive zero	`0 00000000 00000000000000000000000`	`+0.0`
Negative zero	`1 00000000 00000000000000000000000`	`-0.0`

Infinity

Positive infinity	`0 11111111 00000000000000000000000`	`+0.0`
Negative infinity	`1 11111111 00000000000000000000000`	`-0.0`

Representation corner values

Smallest normalized	`* 00000001 00000000000000000000000`	$\pm 2^{(1-127)}$ $\pm 1.1755\ 10^{-38}$
Biggest denormalized	`* 00000000 11111111111111111111111`	$\pm(1-2^{-23})2^{(1-126)}$
Smallest denormalized	`* 00000000 00000000000000000000001`	$\pm 2^{-23}2^{-126}$ $\pm 1.4013\ 10^{-45}$
Max. value	`0 11111110 11111111111111111111111`	$(2-2^{-23})2^{(127)}$ $+3.4028\ 10^{+38}$

Not a number (NaN)

- All ones in the exponent
- Mantissa not equal to the zero
- Used, where no other value fits (i.e. +Inf + -Inf, 0/0)
- Compare to (X+ +Inf) where +Inf is sane result

IEEE-754 special values summary

sign bit	Exponent representation	Mantissa	Represented value/meaning
0	0<e<255	any value	normalized positive number
1	0<e<255	any value	normalized negative number
0	0	>0	denormalized positive number
1	0	>0	denormalized negative number
0	0	0	positive zero
1	0	0	negative zero
0	255	0	positive infinity
1	255	0	negative infinity
0	255	$\neq 0$	NaN – does not represent a number
1	255	$\neq 0$	NaN – does not represent a number

Comparison

- Comparison of the two IEEE-754 encoded numbers requires to solve signs separately but then it can be processed by unsigned ALU unit on the representations

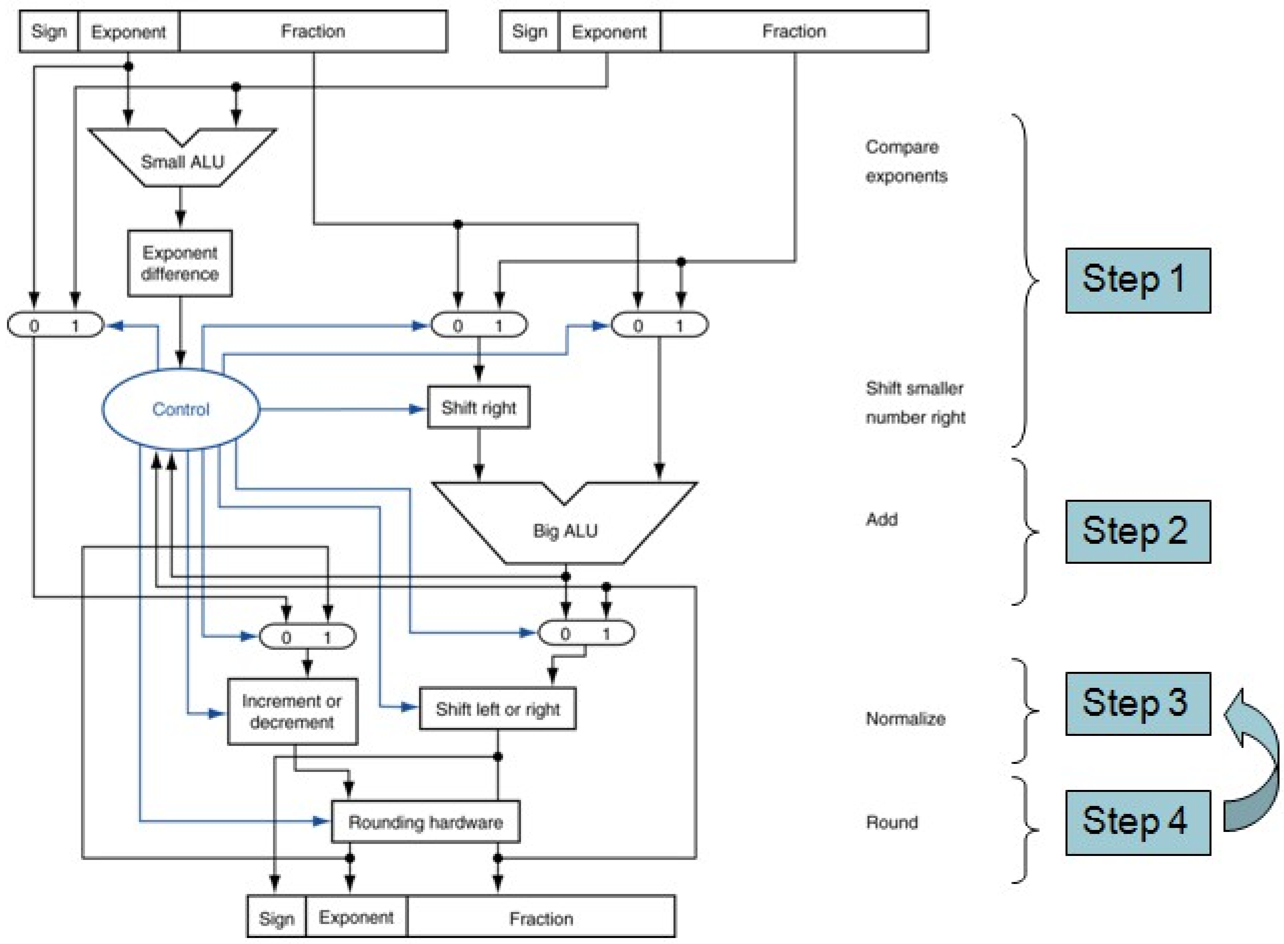
$$A \geq B \Longleftrightarrow A - B \geq 0 \Longleftrightarrow D(A) - D(B) \geq 0$$

- This is advantage of the selected encoding and reason why sign is not placed at start of the mantissa

Addition of floating point numbers

- The number with bigger exponent value is selected
- Mantissa of the number with smaller exponent is shifted right – the mantissas are then expressed at same scale
- The signs are analyzed and mantissas are added (same sign) or subtracted (smaller number from bigger)
- The resulting mantissa is shifted right (max by one) if addition overflows or shifted left after subtraction until all leading zeros are eliminated
- The resulting exponent is adjusted according to the shift
- Result is normalized after these steps
- The special cases and processing is required if inputs are not regular normalized numbers or result does not fit into normalized representation

Hardware of the floating point adder



Multiplication of floating point numbers

- Exponents are added and signs xor-ed
- Mantissas are multiplied
- Result can require normalization

  max 2 bits right for normalized numbers
- The result is rounded

- Hardware for multiplier is of the same or even lower complexity as the adder hardware – only adder part is replaced by unsigned multiplier

Floating point arithmetic operations overview

Addition: $A \cdot z^a , B \cdot z^b , b < a$ unify exponents

$B \cdot z^b = (B \cdot z^{b-a}) \cdot z^{b-(b-a)}$ by shift of mantissa

$A \cdot z^a + B \cdot z^b = [A+(B \cdot z^{b-a})] \cdot z^a$ sum + normalization

Subtraction: unification of exponents, subtraction and normalization

Multiplication: $A \cdot z^a \cdot B \cdot z^b = A \cdot B \cdot z^{a+b}$

$A \cdot B$ - normalize if required

$A \cdot B \cdot z^{a+b} = A \cdot B \cdot z \cdot z^{a+b-1}$ - by left shift

Division: $A \cdot z^a / B \cdot z^b = A/B \cdot z^{a-b}$

A/B - normalize if required

$A/B \cdot z^{a-b} = A/B \cdot z \cdot z^{a-b+1}$ - by right shift