B35APO: Computer Architectures

Lecture 02. Integer and Floating Point Numbers

Pavel Píša
pisa@fel.cvut.cz

Petr Štěpán
stepan@fel.cvut.cz

3. března, 2025

Outline

Repetition and Fundamentals from Previous Lecture

The last lecture introduced:

- The bit (logical value) representation by voltage level
- Byte (logical values vector) representation using parallel bit signals/wires/conductors
- To represent arithmetic unsigned integer values, weights of power two are assigned to the parallel signals
- Positional (place-value) notation / numeral system is introduced
- The representation has been used to implement operation of addition of two non-negative numbers
- Logical bit shift has been introduced (it is correspondent to multiply and divide by power of two for binary number representation)

The Current Lecture Topics

- The ranges which can be represented by integer numbers and their storage in memory
- Multiplication and division of integer non-negative numbers
- Signed numbers (range split for negative part) and respective operation
- Arithmetic and unsigned overflow
- Real numbers representation and operations

Quiz 1

How fast can the sum of two n-bit numbers be calculated, and how many transistors do we need?

A in constant time ($O(1)$) with linear number of transistors ($O(n)$)

B in constant time ($O(1)$) with exponential number of transistors ($O(2^n)$)

C in logarithmic time ($O(\log n)$) with linear number of transistors ($O(n)$)

D in logarithmic time ($O(\log n)$) with cubic number of transistors ($O(n^3)$)

E in logarithmic time ($O(\log n)$) with exponential number of transistors ($O(2^n)$)

Non-negative Integer Numbers

Non-negative integer numbers representation
C-language standard (ISO/IEC 9899:TC3) defines:

type	min	max	informative bytes
unsigned char	0	255	1
unsigned short	0	65 535	2
unsigned long	0	4 294 967 295	4
unsigned long long	0	18 446 744 073 709 551 615	8

- The standard defines minimal ranges, `unsigned int` at least $2^{16} - 1$ (2 bytes), but usually 4 bytes today.
- To find actual size in basic addressable units (C char) use `sizeof(int)`, for range `UINT_MAX`
- For exact size use `uintX_t` and `intX_t` (where `X` is 8, 16, 32, or 64), i.e. `uint8_t`, `int64_t`
- Some exact size types can be missing but guaranteed `[u]int_fastX_t` and `[u]int_leastX_t`, i.e. `uint_least8_t`, `int_fast64_t`

Non-negative Integer Numbers in C-Language

The constant values (integer literal) in C-language source:

- decimal number – has to start by digit '1' to '9' except for '0'
- octal number – starts by digit '0'
- hexadecimal – starts by '0x', continues by '0' – '9' and 'a' to 'f'
- binary – starts by '0b' (GNU compiler extension / C++14 / C23)

Example: 252 == 0xfc == 0374 == 0b11111100

Remark: The hexadecimal digits mapping to bytes is straightforward, each digit (nibble) represents four bits and two digits expressed number fits into single byte, i.e. 0x123456 fits into three bytes (24 bits rounded, exact minimum 21 bits)

Non-negative Integer Numbers in Memory

- Computer memory works with addressable units/cells (usually bytes)
- There are the two basic options for storing longer numbers in memory.

The number 0x12345678 for fixed sized integer types (unsigned int):

address	Big-endian	Little-endian
400	0x12	0x78
401	0x34	0x56
402	0x56	0x34
403	0x78	0x12

- Motorola and IBM processors started with big-endian, Intel processors are usually little-endian.
- It's important when you read/receive (Internet) serialized data by bytes, for example, you have to agree on order between systems
- RISC V - little-endian, MIPS - big-endian but later even little-endian
- Bitcoin - DER signatures big-endian, transaction hash-endian

Non-negative Integer Numbers – Quiz 2

```c
#include <stdio.h>
int main() {
  unsigned char p[] = {0,0,0,0};
  *(int*)p=10;
  printf("%02x,%02x,%02x,%02x\n", p[0],p[1],p[2],p[3]);
}
```

What is the output of the above program on Intel processors?

A nothing, the code cannot be translated

B random result, p (unsigned char *) cannot be casted to *int

C 0a,00,00,00

D 00,00,00,0a

Non-negative Integer Number Multiplication

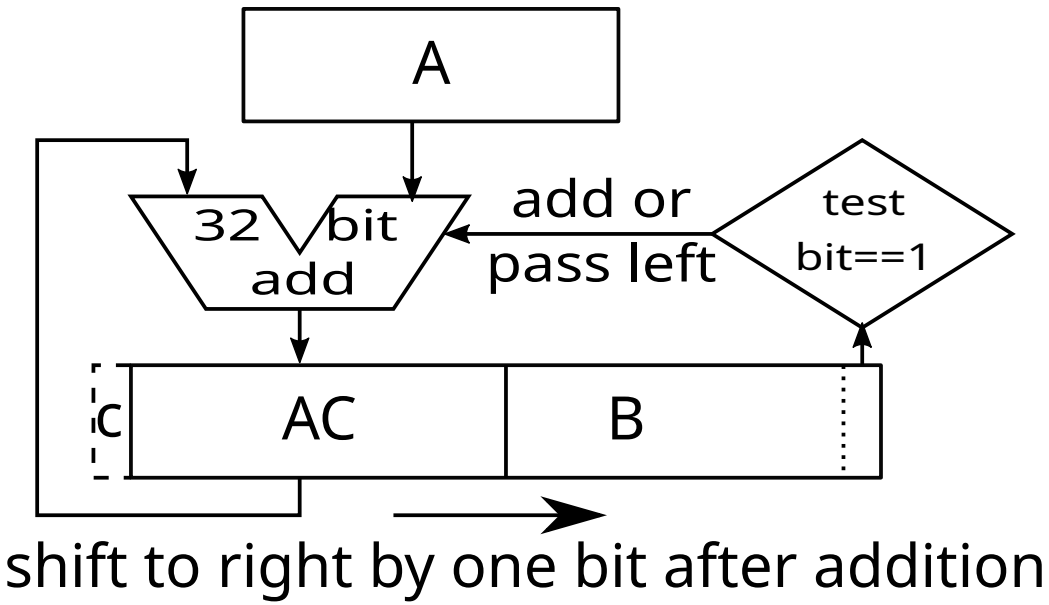
The same principle which you have learned at basic school for decimal numeral system

```
  153
  *45
-------
  765  5
 612   4
-------
 6885
```

```
      10011001
       *101101
-----------------
      10011001  1
     00000000   0
    10011001    1
   10011001     1
  00000000      0
 10011001       1
-----------------
1101011100101
```

Sequential Integer Number Multiplication

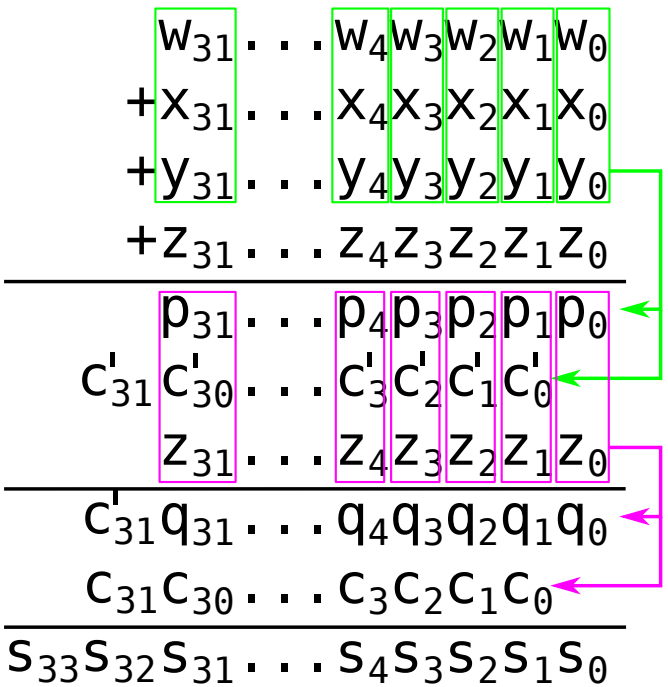
The realization of the algorithm form previous slide with shift register and adder:
(inputs A,B 32-bit, result 64-bit)



- The result will be available after 32 cycles in AC and B
- It is slow, even adder and addition required 32 times (64 times for 64-bit systems).

Fast Multiplication – Wallace Tree Motivation

Potential for speedup – delayed carry (Carry Save Adder).
The optimization of the sum of four 32-bit integers:



- r=w+x and s=y+z can be computed in parallel and then is computed r+s – time equivalent to two full additions
- The carry chain can be delayed (carry is not propagated until the last step):
  - step 1 – use unchained `full adders` and proceed $w_i + x_i + y_i = c'_i p_i$
  - step 2 – use `full adders` again for bits $p_i + c'_{i-1} + z_i = c_i q_i$
  - step 3 – regular adder (i.e. CLA) for 32-bit numbers ($s_0 = q_0$, $c'_3 2$ added to q)

Fast Multiplication – Wallace Tree

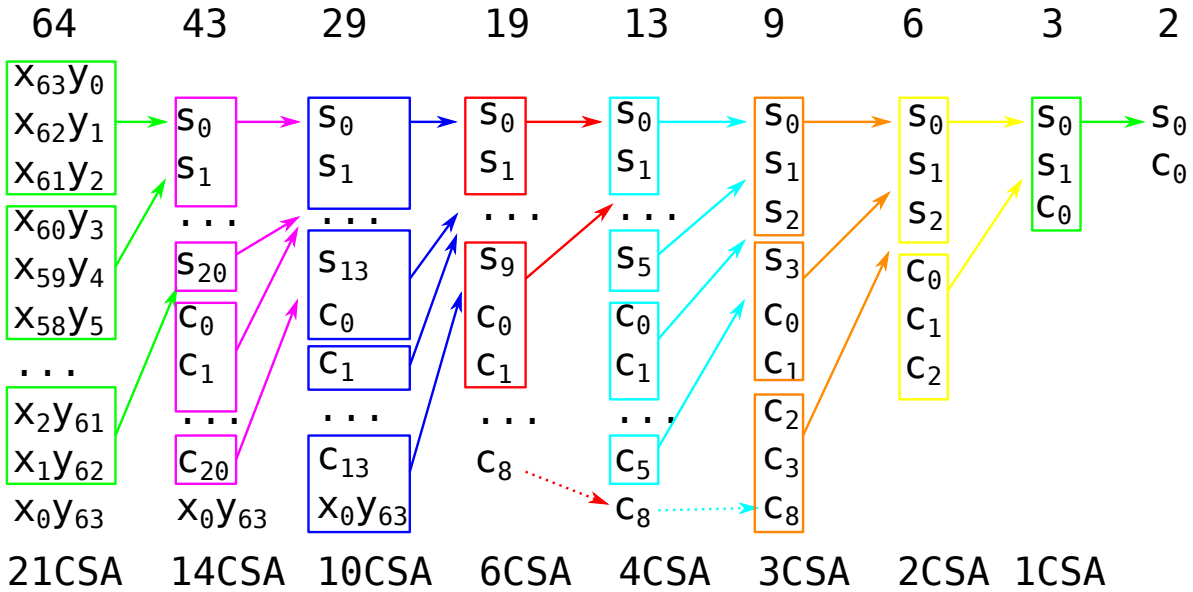
Try to apply the described principle to sum fast 32 or 64 values:

$$
\begin{array}{cccccccccc}
 & & & & x_{63}...x_4x_3x_2x_1x_0 & * & y_{63}...y_4y_3y_2y_1y_0 & & & \\
\hline
 & 0 & 0 & 0 & \cdot\ x_{63}y_0 & \cdot & x_2y_0 & x_1y_0 & x_0y_0 & \\
 & 0 & 0 & 0 & \cdot\ x_{62}y_1 & \cdot & x_1y_1 & x_0y_1 & 0 & \\
 & 0 & 0 & 0 & \cdot\ x_{61}y_2 & \cdot & x_0y_2 & 0 & 0 & \\
 & & ... & & \cdot\ ... & \cdot & & ... & & \\
 & 0 & 0 & x_{63}y_{61} & \cdot\ x_2y_{61} & \cdot & 0 & 0 & 0 & \\
 & 0 & x_{63}y_{62} & x_{62}y_{62} & \cdot\ x_1y_{62} & \cdot & 0 & 0 & 0 & \\
 & x_{63}y_{63} & x_{62}y_{63} & x_{61}y_{63} & \cdot\ x_0y_{63} & \cdot & 0 & 0 & 0 & \\
\hline
q_{127} & q_{126} & q_{125} & q_{124} & & q_{63} & & q_2 & q_1 & q_0
\end{array}
$$

- Actual one bit multiplication is trivial: $x_i \cdot y_j = x_i \text{ and } y_j$
- The most demanding is to sum central column with 64 single bit values
- The adders will be run in parallel on all bits which map to their three inputs and carry will be processed in following steps
- The first phase requires 1323 adders

Fast Multiplication – Wallace Tree

The longest, central column in more detail:



- After 8 counting steps, i.e. 16 gate delays, the two bits are ready for final adder
- The column on the right of the central one are already partially summed and 8 the last carry signals have been promoted
- Two 120-bit numbers (sum and carry) remain to add, which can also be done in 30 gate delays
- Result - we multiply two numbers for the price of time corresponding to two 64-bit additions

Sequential Integer Number Division

Division in the binary system can be done the same way as manual decimal division:

```
 240:11=21
-22
---
  20
 -11
---
   9
```

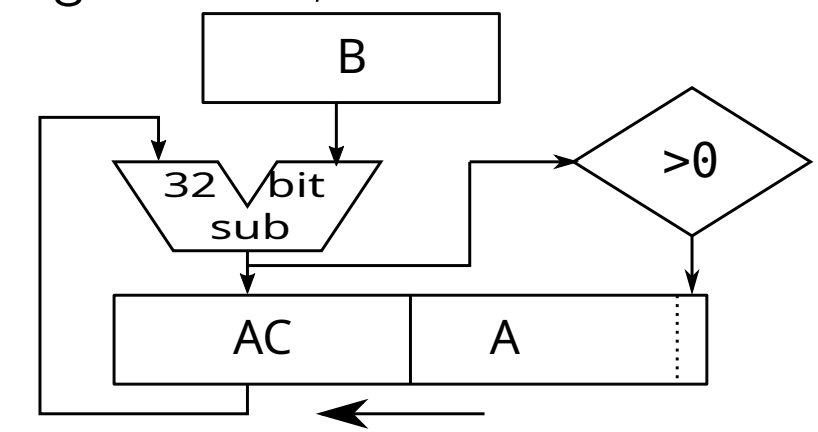
```
 11110000:1011=10101
-1011
-----
  1000
  10000
  -1011
-------
    1010
    10100
    -1011
---------
     1001
```

The both evaluation of 240 by 11 result in 21 and the remainder is 240%11=9.

Sequential Integer Number Division

The `A/B` operation, A is 64-bit, B is 32-bit:

The input A is stored into two registers AC,A



shift left by one bit after each iteration

- Result: A register integral ratio, AC reminder – modulo
- The A register is shifted only in the last step, AC is not shifted – why?
- There exists an even faster algorithm – High Radix Division (it is complex, above focus of our subject)
  - It estimates more bits by approximation and iteration to enhance precision follows
  - 1994 – Pentium FDIV bug – incorrect implementation of Sweeney, Robertson, and Tocher (SRT) algorithm – two bits estimated per single cycle

Outline

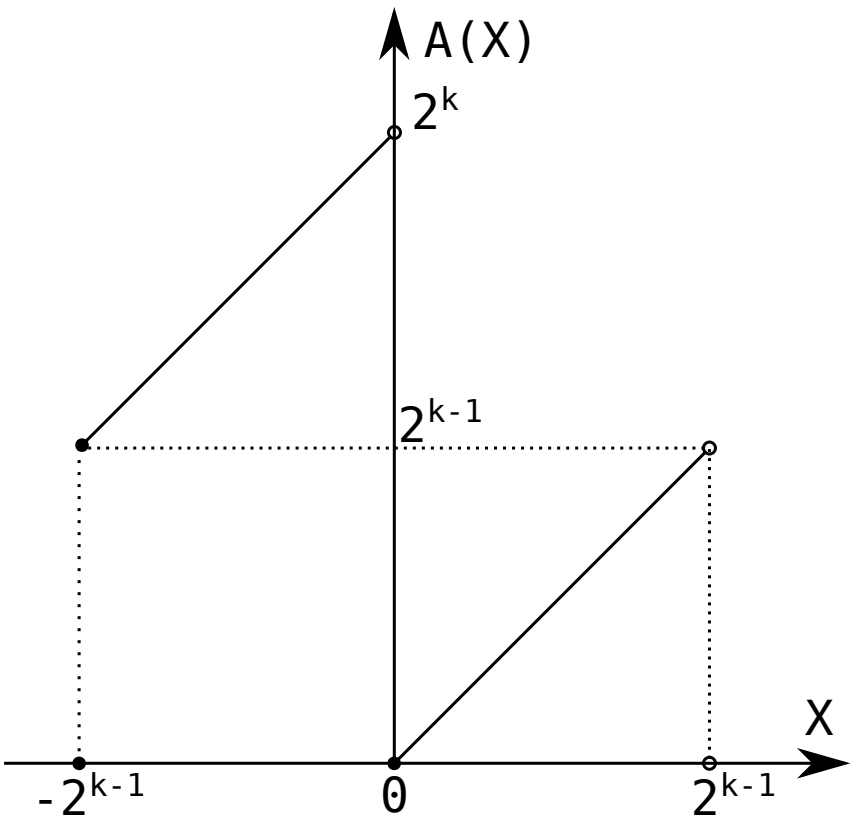
Signed Numbers – Way to Include Negative Ones

The sign has to be encoded into numeral representation:

- simple way - the most significant bit represents sign
  - The absolute value the rest but 0 and -0 even that represents the same value
  - The addition complicated and unsigned adder is hard to reuse
- two's complement (complement to module) – most frequent in use
  - the X arithmetic value representation by k-bit binary evaluates to $X \bmod 2^k$
    - if $X \geq 0$, the representation is the same X
    - if $X < 0$, the value is represented by $2^k - |X|$
  - advantages: the exactly same adder can be used for signed and unsigned types.
  - -1 represented by 8-bit two's complement binary 11111111
  - 5+(-1) is 101+11111111=100000100, the bit 8 (9-th) does not fit into representation, so the results is 101+11111111=100, i.e. 4 in decimal

Two's Complement (Complement to Module)

- the k bits can represent the range $< -2^{k-1}, 2^{k-1} - 1 >$
- let X is arithmetic value, A(X) is unsigned binary value in the two's complement:



8-bit A(X)	Arithmetic value
00000000	$0_{(10)}$
00000001	$1_{(10)}$
...	...
01111110	$126_{(10)}$
01111111	$127_{(10)}$
10000000	$-128_{(10)}$
10000001	$-127_{(10)}$
10000010	$-126_{(10)}$
...	...
11111101	$-3_{(10)}$
11111110	$-2_{(10)}$
11111111	$-1_{(10)}$

Additive Inverse (Opposite Number)

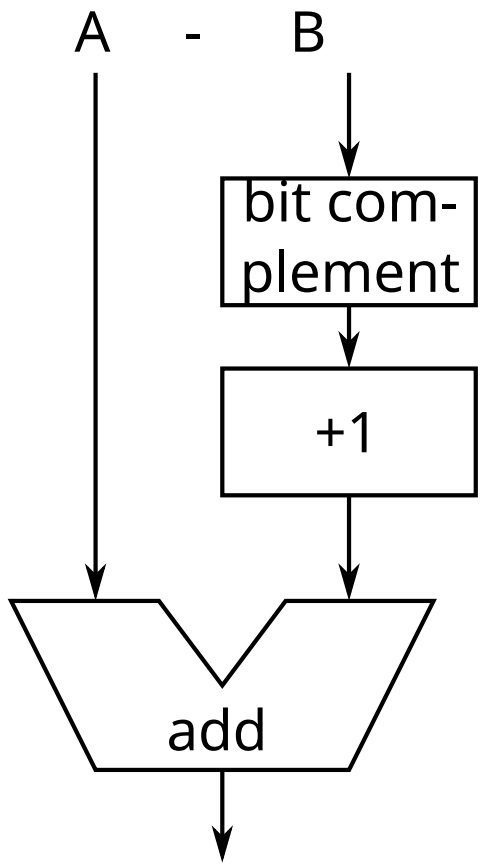
- The addition of the numbers represented by the two's complement is same as for non-negative ones and subtraction `A-B` can be realized as addition where inverse of `B` is added, i.e. `A+(-B)`
- The idea how to compute inverse (often `neg` instruction) `-B` from `B` comes from
  - two's complement negative values are encoded as $X = 2^k - |X|$
  - if we negate (complement) each bit individually we get $(2^k - 1) - X$, because $2^k - 1$ is represented by k ones, no borrow from more significant bits are required
- Final algorithm is:
  1. negate, complement all bits of the input X
  2. add one to the result

Example:
53=0b00110101 bit complement gives -54=0b11001010 and result after addition of one is -53=0b11001011

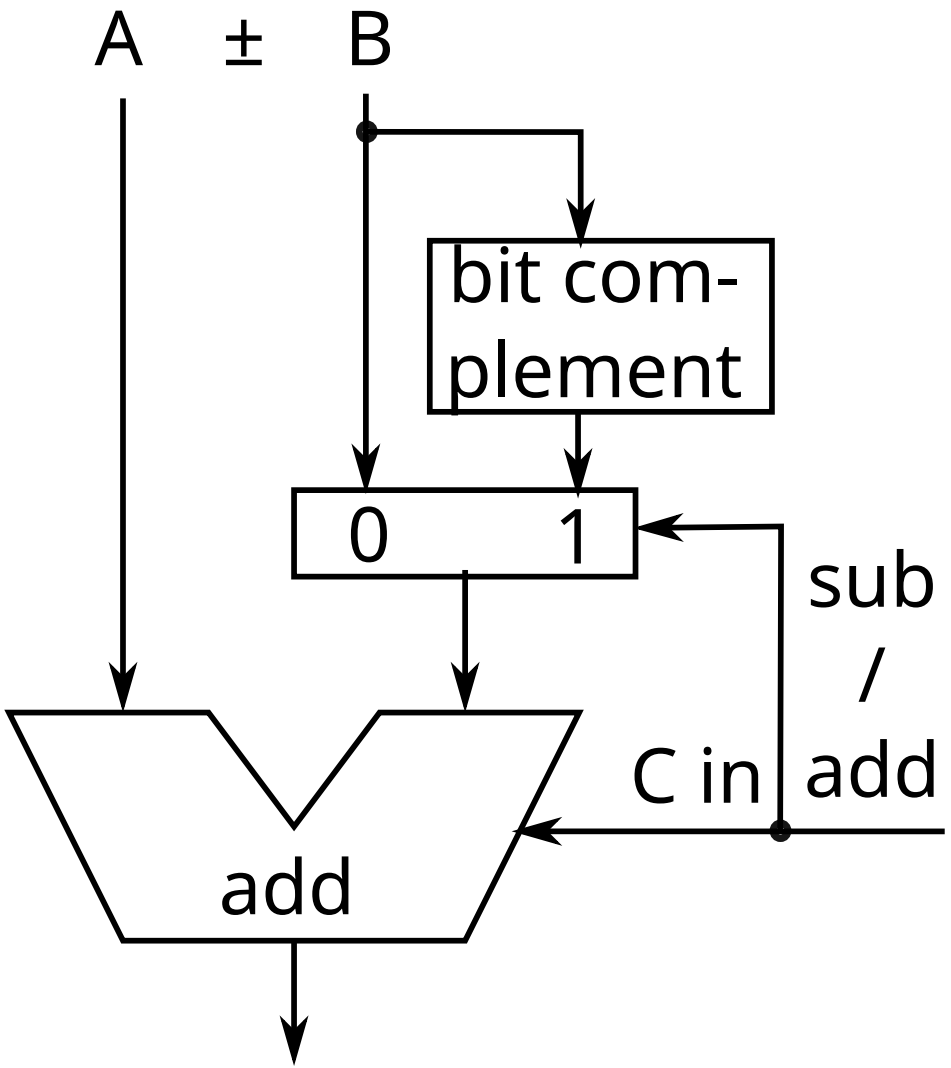
Integer Subtraction

It can be solved:

- by a special circuit similar to an addition with all acceleration possibilities as for addition
- or from the two's complement and inverse number we can convert it to addition and same hardware



Single Unit for Addition and Subtraction



Multiplication and Division in Two's Complement

- rule for adjustment of the result of multiplication based on unsigned $M \cdot N$, $(A(M) \cdot A(N))$ multiplication of two M, N two's complement k-bit numbers :

$$A(M \cdot N) = A(M) \cdot A(N) \begin{cases} -A(M) \cdot 2^k & \text{when } N < 0 \\ -A(N) \cdot 2^k & \text{when } M < 0 \end{cases}$$

- because two's complement representation of $A(M) = 2^k + M$, then result of multiplication for for two negative numbers is $(2^k + M) \cdot (2^k + N) = 2^{2 \cdot k} + 2^k \cdot M + 2^k \cdot N + M \cdot N$
- today's fast multipliers and divisors compute usually with absolute values and for the sign
  - it is stored in the most significant bit
  - inverse number computation is fast

Signed Numbers in C-language

Integer numbers representation in C

Next types are defined by standard with minimal ranges:

type	min	max	byte count
char	-128	127	1
short	-32 768	32 767	2
long	-2 147 483 648	2 147 483 647	4
long long	-9 223 372 036 854 775 808	9 223 372 036 854 775 807	8

- The C standard defines required minimum by one higher than two's complement to not eliminate processor computing in one's complement – but that is not practically used today
- To be sure about actual range of `int` use `INT_MIN` and `INT_MAX`

Signed Numbers – Quiz 3

Consider the following program:

```
#include <stdio.h>
int main() {
  unsigned char a=150u, b=120u, c;
  char sa=-100, sb=-80, sc;

  c=a+b;
  sc=sa+sb;
  printf("c=%u sc=%d\n", c, sc);
}
```

What will be printed:

A c=270 sc=-180
B c=14 sc=-76
C c=14 sc=76
D c=-14 sc=-76
E Numeric error

Overflow for Unsigned Numbers

Unsigned char is 8-bit represented number typically which is reason that next operation overflows:

```
 150 =   1001 0110
+120 =   0111 1000
-------------------
  14 =   0000 1110
 270 =1  0000 1110
```

The result does not fit in 8-bit representation, the most significant bit is lost, and the result is only 14.

If we want to signal overflow in addition we can use:

- C23 bool ckd_add(type1 *result, type2 a, type3 b) – addition of two numbers with overflow signalling
- GNU GCC 5+, Clang 3.8+ __builtin_add_overflow(a, b, result) – both versions even for sub and mul
- Can be detected on conventional C by check if the result is smaller than both inputs

Arithmetic Overflow for Signed Numbers

The overflow in operations with signed numbers is more complex.
Examples what results in overflow:

```
-112 = 10010000
+ 45 = 00101101
-----------------
 -67 = 10111101
```

CORRECT

```
 -12 = 11110100
+ -20 = 11101100
-----------------
 -32 =111100000
```

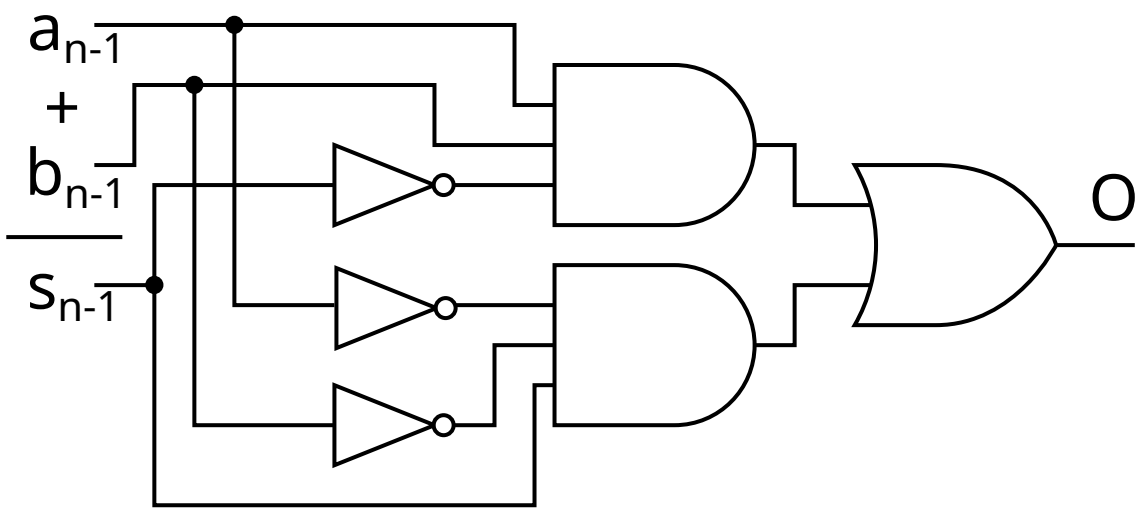
CORRECT

```
 -90 = 10100110
+ -42 = 11010110
-----------------
 124 =101111100
```

OVERFLOW

- The arithmetic overflow for operation with signed numbers is present if auxiliary carry to the most significant result bit differs from the carry out from this bit:
  - overflow = c_n xor c_{n-1}; c_n carry from most significant result bit, c_{n-1} is carry into the most significant bit
- The second option i to check that addition result of the two positive numbers is positive and for two negative numbers stay negative:
  - overflow = (a_{n-1} and b_{n-1} and (not s_{n-1})) or ((not a_{n-1}) and (not b_{n-1}) and s_{n-1}); a_{n-1}, b_{n-1} are the most significant bits of addends; s_{n-1} is MSB of result

Arithmetic Overflow – Quiz 4



When two numbers with opposite signs are added:

A can only overflow in two's complement representation

B can only overflow in representation other two's complement representation

C cannot only occur in representation in two's complement representation

D cannot occur in any representation of signed numbers

Other Representations of Signed Numbers

Excess-K (offset binary):

- for k-bit representation, offset K is usually $K = 2^{k-1}$ or $K = 2^{k-1} - 1$
- representation/code for number X is $A(X) = X + K$
- arithmetic value is obtained from $D(A) = A - K$
- the range of the represented arithmetic values is $< -K, 2^k - K - 1 >$

for k=8 and K=127

A(X)	Value
00000000	$-127_{(10)}$
00000001	$-126_{(10)}$
...	...
01111110	$-1_{(10)}$
01111111	$0_{(10)}$
10000000	$1_{(10)}$
10000001	$2_{(10)}$
...	...
11111110	$127_{(10)}$
11111111	$128_{(10)}$

Excess-K – Arithmetic Operations

- the addition and subtraction operations processed directly representation:
- $A(X+Y) = (X+Y)+K = (X+K)+(Y+K)-K = A(X)+A(Y)-K$
- $A(X-Y) = (X-Y)+K = (X+K)-(Y+K)+K = A(X)-A(Y)+K$
- multiplication is even more complex:
- $A(X \cdot Y) = (X \cdot Y)+K = (X+K)\cdot(Y+K)-(X+K+Y+K)\cdot K+K^2+K = A(X)\cdot A(Y)-(A(x)+A(y))\cdot K+K^2+K$

- Overflow:
  - for addition, same sign inputs and opposite sign on output

Other Representations of Signed Numbers

One's complement:

- negative number is represented by bit complement of its absolute value. For k-bit representation:
  - for $X \geq 0$, representation is $A(X) = X$
  - for $X < 0$, representation is $A(X) = 2^k - 1 - |X|$
- disadvantages: two representations of the value zero (-0, +0) complicate addition (hot one correction)

Binary coded decimal (BCD) representation

- another representation of integer numbers, each decimal digit maps to nibble
  - the number `1234` representation printed in hexadecimal form gives `0x1234`
- advantages: simple conversion to and from decimal input/output
- disadvantages: ineffective storage – space waste, complex computation

Outline

Real Numbers

- Integer number X in k-bit binary representation can be split into bits where each b_i is multiplied by corresponding power of two (weight), that is $X = \sum_{i=0}^{k-1} b_i 2^i$
- Real number X is again similar sum of $k + j$ bits b_i which are multiplied by powers of two (2), but we add negative powers of two: $X = \sum_{i=-j}^{k} b_i 2^i$
- Remember grammar school math lessons $2^{-j} = \frac{1}{2^j}$

							$\frac{1}{2}$	$\frac{1}{4}$			
				4	2	1	2	4			
				=	=	=	=	=			
2^i	2^{i-1}	2^{i-2}	...	2^2	2^1	2^0	2^{-1}	2^{-2}	...	2^{-j+1}	2^{-j}
b_i	b_{i-1}	b_{i-2}	...	b_2	b_1	b_0.	b_{-1}	b_{-2}	...	b_{-j+1}	b_{-j}

Fixed Point Real Numbers

Fixed point numbers:

- similar as signed excess-K representation
- real number is represented by k-bit signed integer number, the fixed number of fractional bits is chosen, where $0 \leq s \leq k$
- representation for arithmetic value X is $A(X) = \lceil X \cdot 2^s \rceil$
- decoding of the representation to arithmetic value $D(A) = \frac{A}{2^s}$
- the range of represented numbers $\langle -\frac{2^{k-1}}{2^s}, \frac{2^{k-1}-1}{2^s} \rangle$
- absolute precision of the representation is $\pm\frac{1}{2^s}$.

Special case for fractional numbers:

- if numbers from range $\langle 0, 1)$ should be represented (usually control systems)
- ten $s = k$ and we can directly use unsigned integer numbers representation
- the better precision can be achieved than for the same bit size `float`, or `double` representation

Fixed Point Real Numbers – Operations

- addition and subtraction is equivalent to the same operation on the number representation
- multiplication requires fix-up (normalization) step:
  - $A(X \cdot Y) = (X \cdot Y) \cdot 2^s = \frac{(X \cdot 2^s) \cdot (Y \cdot 2^s)}{2^s} = \frac{A(X) \cdot A(Y)}{2^s}$
- division is similar:
  - $A(\frac{X}{Y}) = (\frac{X}{Y}) \cdot 2^s = \frac{(X \cdot 2^s) \cdot 2^s}{(Y \cdot 2^s)} = \frac{A(X) \cdot 2^s}{A(Y)}$
- It is not so significant complication because multiplication by 2^s is shift by s bits to the left and division by 2^s is (arithmetic) shift by s to the right.

Some SIMD instructions set extensions provide implementation of operations for computation in fixed point numbers representation.

Floating Point Real Numbers

It is equivalent to scientific notation format for decimal real numbers writing: $-123000000000000.0 = -1.23 \cdot 10^{14} = -1.23\text{E}14$
$-0.000000000000123 = -1.23 \cdot 10^{-13} = -1.23\text{E} - 13$
Significand (mantissa) $\in \langle 1; 10)$ for normalized form

Binary representation only changes base for exponent to 2:
$110110000000000.0_2 = 1.1011_2 \cdot 2^{14} = 1.1011_2\text{E}14 = 29696_{10}$
$-0.0000000000000011101_2 = -1.1101_2 \cdot 2^{-16} = -1.1101_2\text{E} - 16 \approx$
$\approx 0.00002765_{10}$
Significand (mantissa) $\in \langle 1; 10_2)$ that is $\in \langle 1; 2_{10})$ for normalized form

As in scientific notation the number has to start by single non-zero digit before decimal point, the binary representation has to start by single one (exception is zero) before binary point

IEEE-754 Floating Point Standard

Standard IEEE-754 defines how to encode real numbers into 32 (C float), 64 bits (C double)
32-bit representation of the real number composes of:

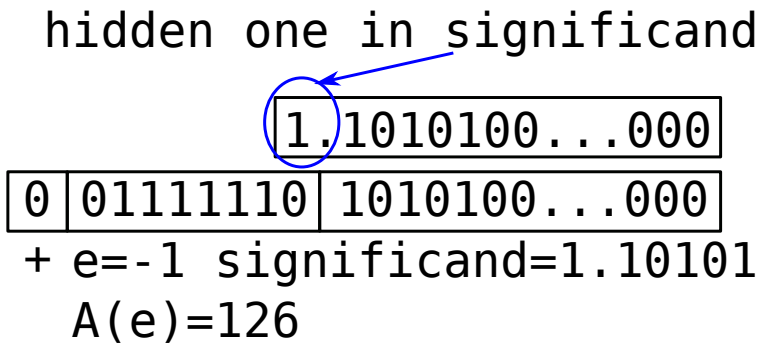
- 1 bit for the sign (both +0 and -0 are defined)
- 8 bits two based exponent in excess-k representation (K=127)
- 23 bits to represent significand fractions (plus implicit MSB one)

64-bit representation of the real number composes of:

- 1 bit sign (both +0 and -0 are defined)
- 11 bits two based exponent in excess-k representation (K=1023)
- 52 bits to represent significand fractions (plus implicit MSB one)

IEEE-754

Example: Real number $0.828125_{(10)} = 0.5 + 0.25 + 0.0625 + 0.015625 = 2^{-1} + 2^{-2} + 2^{-4} + 2^{-6} = 0.110101_{(2)}$.
The number is converted into scientific like binary format:
$0.110101 = 1.10101\mathrm{E} - 1$.
Exponent is $e = -1$, its excess-k representation ($\mathrm{K} = 127$) is
$A(-1) = -1 + 127 = 126$.
The space for leading MSB of significand (hidden one) is not reserved because it is guaranteed by format definition (except for zero and some corner cases):



You can experiment with
https://www.h-schmidt.net/FloatConverter/IEEE754.html

IEEE-754 – Normalized Range

Normalized number is each number which can be written as `1.XXXXX E` exp and power exponent fits in range -126 till 127 for 32-bit representation. That is exponent in excess-k format is in range 1 to 254.

Denormalized numbers are used to cover range near zero and can be written as `0.XXXXX E -126`, for example, even 0.0, the interval is defined for 32-bit representation as $(-1.17549\text{E}-38, 1.17549\text{E}-38)$, that is $(-2^{-126}, 2^{-126})$.

When exponent is encoded as all ones 1, that is 255, i.e. exponent arithmetic value is 128, then special value is represented

- if all significand bits are 0, then **infinite** (value to big to represent) is stored, it can be -inf, or *inf* according to sign.
- if significand is non-zero, then nothing about arithmetic value can be considered *NaN* – **not a number**, some error in the computation, for example square root of a negative real number.

IEEE-754 – Overview

Encoding table:

Exponent	Significand	Value
00000000	0	0.0 – zero
00000000	non-zero	denormalized numbers around 0
00000001	0	the smallest normalized number (with hidden one)
1 to 254	any value	normalized numbers (with hidden one in significand)
11111111	0	infinity
11111111	non-zero	NaN – error value

The **smallest normalized number not equal to zero** is:

- exponent = 0 (-126), significand=000...0001, value = $2^{-23+(-126)} \approx 1.4E-45$

Normalized number with the **smallest absolute value**:

- exponent = 1 (-126), significand=000...0000, value = $2^{-126} \approx 1.17E-38$

Normalized number with the **biggest absolute value**:

- exponent = 255 (127), significand=111...1111, value = $(2-2^{-23})2^{127} \approx 3.4E38$

IEEE-754 2008 Revision

The 2008 revision defines 16-bit real numbers encoding (half precision) and 128-bit encoding (quad precision).
16-bit real number IEEE-754 representation:

- 1 bit sign (the both +0 and -0 are defined)
- 5 bits two based exponent in excess-k representation (K=15)
- 10 bits significand fractions (plus implicit MSB one)

128-bit real number IEEE-754 representation:

- 1 bit sign (the both +0 and -0 are defined)
- 15 bits two based exponent in excess-k representation (K=16383)
- 112 bits significand fractions (plus implicit MSB one)

IEEE-754 - Comparison

Comparison of the real numbers (equal, greater than):

- A positive number is greater than the negative one, check sign the first if different positive is greater than negative except for zero, where $+0$ and -0 are equal
- When signs are removed, then absolute numbers values can be compared in the representation format (as they are in memory) same as unsigned numbers of the same size and endianness
  - This is possible thanks to exponent excess-K (offset binary) representation
  - Greater exponent than the represented number is greater, when exponents are equal greater significand value represents greater number.

Remark: the offset in exponent k_e-bits representation is chosen as $2^{(k_e-1)/2} - 1$ which ensures that reciprocal value to the smallest normalized number fits into representation (does not overflow to Inf, ∞)

IEEE-754 – Addition / Subtraction

1. The number with bigger exponent value is selected, significands extracted and for normalized numbers extended by implicit MSB one
2. Significand of the number with smaller exponent is shifted right by exponent difference – the significands are then expressed at same scale
3. The signs are analyzed and significands are added (same sign) or subtracted (smaller number from bigger)
4. The resulting significand is shifted right (max by one) if addition overflows or shifted left after subtraction until all leading zeros are eliminated (result can be even zero, then encode zero directly)
5. The resulting exponent is adjusted according to the shift (increment exponent by one for each right shift by bit, decrement exponent by one for each left shift by bit)
6. Result is normalized after these steps and sign is copied from larger source
7. The special cases and processing when inputs are not normalized numbers or result does not fit into normalized representation

IEEE-754 – Addition Example

Example: add two real numbers 31.5+0.75 in their binary floating point representations

$31.5_{(10)} = 11111.1_{(2)} = 1.11111\text{E}4 \qquad 0.75_{(10)} = 0.11_{(2)} = 1.1\text{E}-1$

Both numbers have to be converted to the same binary exponent 4 and then significands are added:

```
  1.11111
  0.000011
-----------
 10.000001
```

The result has to be normalized (significand $\in \langle 1; 2)$) by incrementing exponent to 5 which corresponds to binary fraction point by one position left (rounding can be required to fit in defined bits for significand):
$10.000001\text{E}4 = 1.0000001\text{E}5$

The binary floating point number represents 32.25 decimal.

IEEE-754 – Multiplication

1. Exponents are added and signs xor-ed
2. Significands are multiplied
3. Result can require normalization, max 1 bit right shift and increment exponent by one for normalized input numbers
4. The result is rounded
5. Special care has to be taken for normalized inputs and or result in out of normalized range

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

IEEE-754 – Multiplication Example

Example: multiply two real numbers $0.375 \cdot 1.5$ in their binary representations

$0.375_{(10)} = 0.011_{(2)} = 1.1\text{E} - 2 \qquad 1.5_{(10)} = 1.1_{(2)} = 1.1\text{E}0$

The significands multiplication:

```
   11  ≡  1.1
  *11  ≡  1.1
-------------
   11
  11
-------------
 1001
```

result with two binary fractional digits `10.01`

```
  375  ≡  0.375
  *15  ≡  1.5
-------------
 1875
 375
-------------
 5625
```

result with four decimal fractional digits `0.5625`

Exponent addition $-2 + 0 = -2$, but result of significands multiplication requires normalization, that is exponent is incremented to -1.

The correct result is obtained $10.01\text{E} - 2 = 1.001\text{E} - 1 \equiv 0.5625_{(10)}$

IEEE-754 – Summary

Real numbers:

- the floating point real numbers allows to represent values in large dynamic range with almost constant relative precision when exponent allows normalized form:
  - float – absolute represented value from $1.175494351\text{E}-38$ to $3.402823466\text{E}+38$
  - double – absolute represented value from $2.2250738585072014\text{E}-308$ to $1.7976931348623158\text{E}+308$
- the relative precision by number of valid decimal digits:
  - float – 6-7 valid decimal digits (increment $2^{\langle -23;-24)} : 1$)
  - double – 15-16 valid decimal digits (increment $2^{\langle -52;-53)} : 1$)

WARNING: next `while` loop is infinite, never ends:

```
float a=1.0, step=5e-8;
while (a*a<1.01) {
  a+=step;
}
```