UTF-32

- Character encoded on **4 bytes** "as it is"
- Obvious problem with **byte sequence** (*endianness*)
  - Example

    A in ASCII 01000001

    A in UTF-32 00000000 00000000 00000000 01000001

    or, 01000001 00000000 00000000 00000000
  - So, not compatible with ASCII
  - Super wasteful
  - But faster text operations (e.g character count)
  - Messes where “null terminated strings” (00000000) are expected

UTF-32 example

Q1: Present in UTF-32 8-bit code (in the solution, the byte order from MSB on the left): 01000000

- Answer : 00000000 00000000 00000000 01000000

Q2: Present in UTF-32 17-bit code (in the solution, the byte order from MSB on the left): 1 00000000 11111111 , In the solution, separate bytes with a space. And If impossible - enter x

- Answer : 00000000 00000001 00000000 11111111

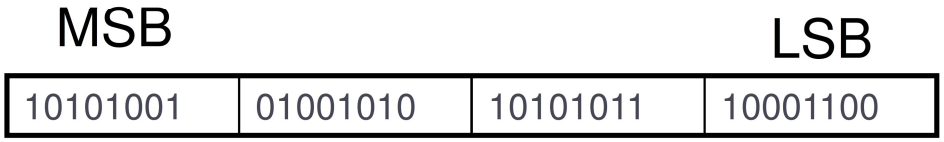
UTF-32 example

Q3: Present in UTF-32 32-bit code (in the solution, the byte order from MSB on the left): 01111111 11111111 11111111 11111111 , In the solution, separate bytes with a space. If impossible - enter x

- Answer : 01111111 11111111 11111111 11111111

Bits and Bytes Terminology

- LS-Bit: least-significant bit
- MS-Bit: most-significant bit



- LSB: least-significant byte
- MSB: most-significant byte

ENDIAN

- In What order do we load/store the bytes of a multi-byte value?
  - Depends whether processor is:
    - "big endian" or "little endian"
- Big Endian:
  - load/store the MSB first
    - i.e., in the lowest address location
- Little Endian:
  - load/store the LSB first
    - i.e., in the lowest address location
- There is no superior endian!

Endian Matters When:

1) You store a multi-byte value to memory
2) Then you load a subset of those bytes

Different endian will give you different results!

Different processors support different endian

- Big endian: motorola 68k, PowerPC (by default)
- Little endian: intel x86/IA-32, NIOS
- Supports both: PowerPC (via a mode)

Eg: must account for this if you send data from big-E machine to a little-E machine

EX: store 0xA1B2C3D4 to addr 0x20

Big Endian

Addr	Value
0x20	0xA1
0x21	0xB2
0x22	0xC3
0x23	0xD4

Little Endian

Addr	Value
0x20	0xD4
0x21	0xC3
0x22	0xB2
0x23	0xA1

Example

- A 64-bit word of hexadecimal value #1122334455667788 was stored at the address 9656 (decimal) in memory of a Little-Endian computer with 8-bit bytes. What is the address of a byte containing value #22?
  - **Solution** : divide the hexadecimal value into bytes to be

    11 22 33 44 55 66 77 88

  - Arrange in Little-Endian : 88 77 66 55 44 33 22 11

Data	88	77	66	55	44	33	22	11
Address	9656	9657	9658	9659	9660	9661	9662	9663

  - The address of a byte containing value #22 is **9662**

Binary Arithmetic

Converting Between Bases (revision)

- Binary to Decimal & Decimal to Binary
- Hexadecimal to Decimal & Decimal to Hexadecimal
- Converting From Binary to hex And From hex to Binary

Converting Binary to Decimal

Example 1: Convert the binary number $(101100)_2$ to its decimal equivalent:

2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	1	0	0

$$(101100)_2 = 1\times2^5+0\times2^4+1\times2^3+1\times2^2+0\times2^1+0\times2^0$$

$$= (44)_{10}$$

Converting Binary to Decimal

Example 2: Convert the binary number $(1101)_2$ to its decimal equivalent:

2^3	2^2	2^1	2^0
1	1	0	1

$$(1001)_2 = 1\times2^3+1\times2^2+0\times2^1+1\times2^0$$

$$= (13)_{10}$$

Converting Decimal to binary

Example 1: Convert the decimal number $(50)_{10}$ to its binary equivalent:

Division	Reminder
50÷ 2 = 25	0
25÷ 2 =12	1
12÷ 2 =6	0
6÷ 2 =3	0
3÷ 2 =1	1
1÷ 2 =0	1

Converting Decimal to binary

Example 2: Convert the decimal number $(23)_{10}$ to its binary equivalent:

Division	Reminder
$23 \div 2 = 11$	1
$11 \div 2 = 5$	1
$5 \div 2 = 2$	1
$2 \div 2 = 1$	0
$1 \div 2 = 0$	1

The result of converting $(23)_{10}$ to binary form is $(10111)_2$.

Converting Hexadecimal to Decimal

Example 1: Convert the hex number $(1A5F)_{16}$ to its decimal equivalent:

16^3	16^2	16^1	16^0
1	A	5	F

$$(1A5F)_{16} = 1\times16^3 + A\times16^2 + 5\times16^1 + F\times16^0$$

$$= (6751)_{10}$$

Converting Hexadecimal to Decimal

Example 2: Convert the hex number $(234)_{16}$ to its decimal equivalent:

16^2	16^1	16^0
2	3	4

$$(234)_{16} = 2\times16^2+3\times16^1+4\times16^0$$

$$= (564)_{10}$$

Converting Decimal to Hexadecimal

Example 1: Convert the decimal number $(56)_{10}$ to its hex equivalent:

Division	Reminder
$56 \div 16 = 3$	8
$3 \div 16 = 0$	3

The result of converting $(56)_{10}$ to hex form is the reminder from bottom to top (start writing it from left to right) which equals $(38)_{16}$

Converting Decimal to Hexadecimal

Example 2: Convert the decimal number $(145)_{10}$ to its hex equivalent:

Division	Reminder
$145 \div 16 = 9$	1
$9 \div 16 = 0$	9

The result of converting $(145)_{10}$ to octal form is $(91)_{16}$.

Converting From Binary to hex And From hex to Binary

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Converting From Binary to hex And From hex to Binary

Example 1: Convert the binary number $(1001001010)_2$ to its hex equivalent:

To do that we will split the binary number into sections of four digits from the right to the left as following

0010	0100	1010
2	4	A

From that we can say that $(1001001010)_2 = (24A)_{16}$

Example 2: Convert the hex number $(A5)_{16}$ to its binary equivalent:

We will select the equivalent value of each number from the table as following:

A	5
1010	0 101

From that we can say that $(A5)_{16} = (10100101)_2$

Representations of numbers

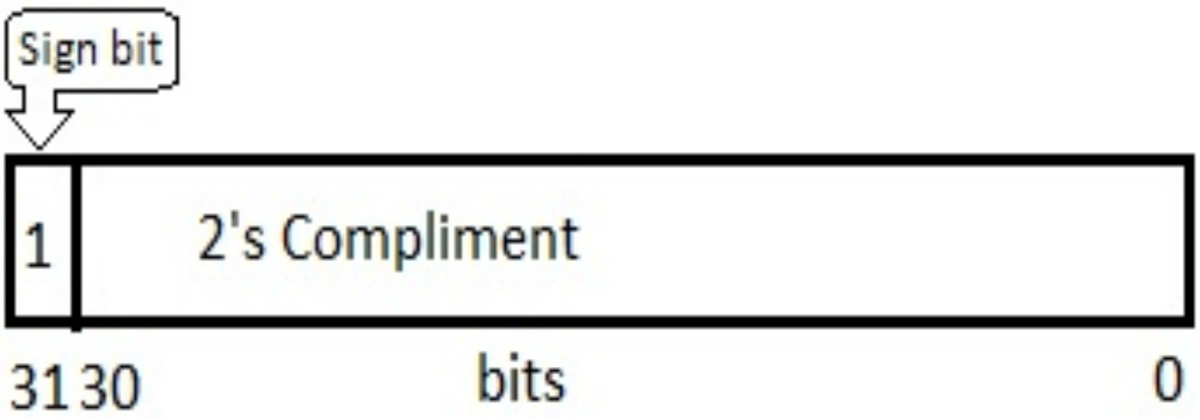
- Integers
- Fractions
- Fixed-Point Numbers
- Floating-Point Numbers

Integers

- Most basic way to store numbers in binary
- Each bit represents a different number in powers of two
- Types
  - Unsigned
  - Signed

Signed Integers

- Sign Magnitude: With the addition of a sign bit to keep track of positive or negative numbers
- 2's complement



Signed Magnitude

- Use left-most bit (called *most significant bit* or MSB) for sign:

  0 for positive

  1 for negative

- Example: $+18_{ten} = 00010010_{two}$

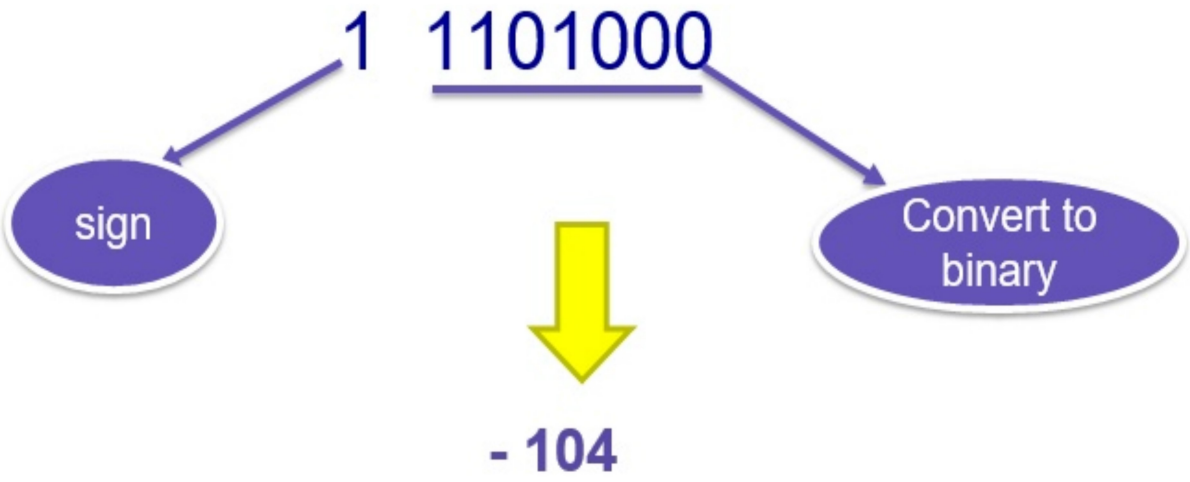
  $-18_{ten} = 10010010_{two}$

  - With the addition of a sign bit to keep track of positive or negative numbers

Example

- The decimal value of the byte in the Sign-Magnitude code, given in hex: #E8 is:

E8



1's Complement Numbers (U1)

Decimal magnitude	Binary number	
	Positive	Negative
0	0000	1111
1	0001	1110
2	0010	1101
3	0011	1100
4	0100	1011
5	0101	1010
6	0110	1001
7	0111	1000

Negation rule: invert bits.

Problem: $0 \neq -0$

Example

- Find 1's complement of binary number

10001.001

01110.110

2's Complement Numbers (U2)

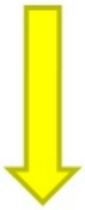
Negation rule: invert bits and add 1

Decimal magnitude	Binary number	
	Positive	Negative
0	0000	
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001
8		1000

Example

- Find 2's complement of binary number

11100011

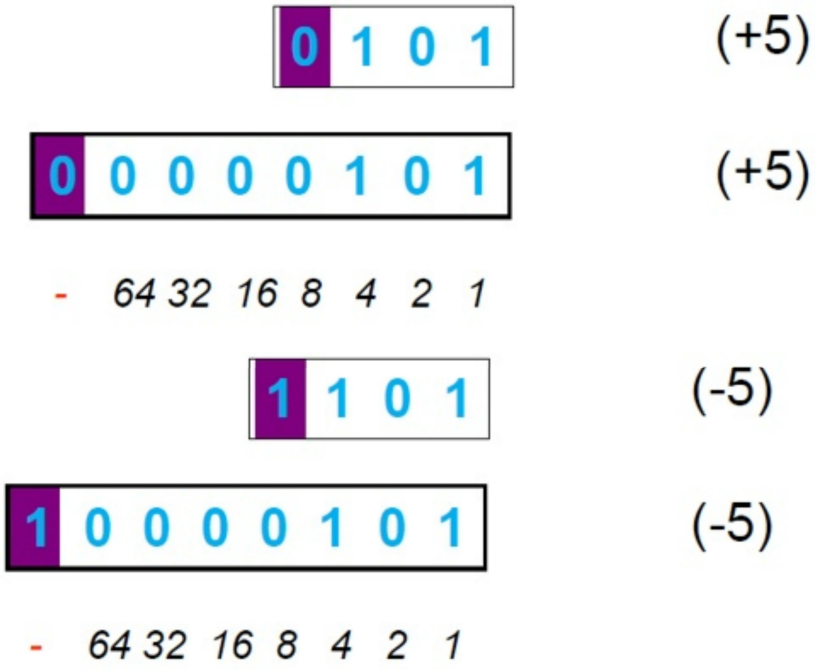


00011101

Three Representations

Sign-magnitude	1's complement	2's complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = - 0	100 = - 3	100 = - 4
101 = - 1	101 = - 2	101 = - 3
110 = - 2	110 = - 1	110 = - 2
111 = - 3	111 = - 0	111 = - 1
		(Preferred)

Extending Signed Magnitude format (8-bit)



Complete the missing places with zeros

Extending U1 format

(+5)

(+5)

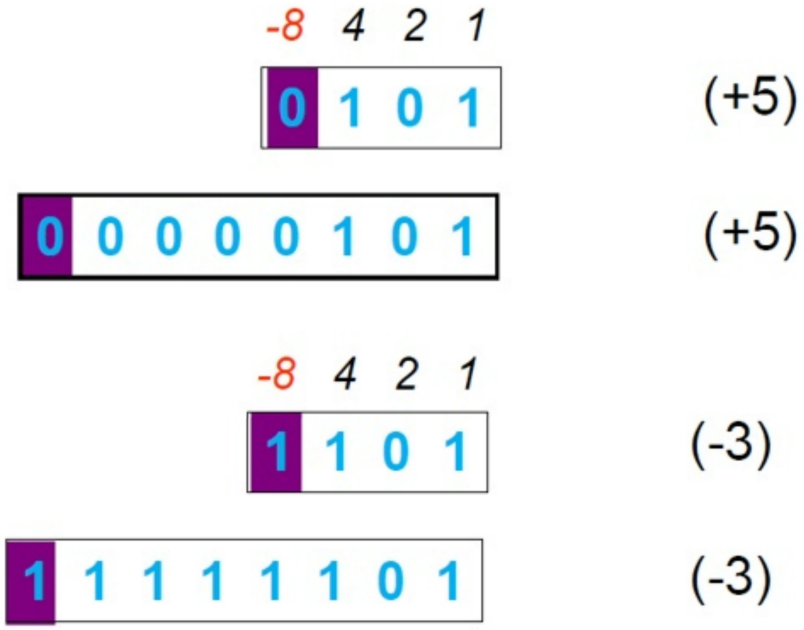
-127 64 32 16 8 4 2 1

(-2)

2=0010

(-2)

Extending U2 format



3=0011

Fractions

- Convert binary integers to decimal we just multiply them by 2^x , while in binary fraction we multiply them by 2^{-x}
- Not accurate and cause rounding errors.
- More complicated numbers multiply fraction by 2

n	2^{-n}
0	1.0
1	0.5
2	0.25
3	0.125
4	0.0625
5	0.03125
6	0.015625
7	0.0078125
8	0.00390625
9	0.001953125
10	0.0009765625
11	0.00048828125
12	0.000244140625
13	0.0001220703125
14	0.00006103515625
15	0.000030517578125

Fractions (from decimal to binary)

2^0	.	2^{-1}	2^{-2}	2^{-3}
0	.	1	1	0

$(0.45)_{10} = (0.011100)_2$

	Integral part
0.45×2= 0.9	0
.9×2= 1.8	1
.8×2 = 1.6	1
.6×2= 1.2	1
.2×2 = 0.4	0
.4×2= 0.8	0
.8×2 = 1.6	1 (discard)

repearted

Example

- The fractional binary number 0.1101 is equal to A/B proper decimal irreducible fraction (specify as A.B, for example, 5/12, specify as 5.12, do not provide a decimal fraction 0.xxxx):

$$0.1101 = \frac{1}{2} \times 1 + \frac{1}{4} \times 1 + \frac{1}{8} \times 0 + \frac{1}{16} \times 1$$

$$= \frac{13}{16}$$

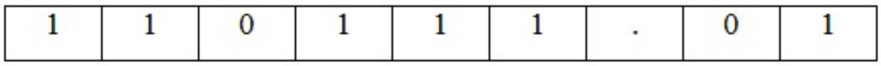
From binary to decimal

Fixed-Point Numbers

- Simply fix the binary point to be at some position of a numeral.
- To define a fixed point type conceptually, all we need are two parameters:
  - width of the number representation (the number of bits used)
  - binary point position within the number

fixed<w,b>

Fixed <8,2> =xxxxxx.xx



Fixed-Point Numbers

- Fixed-point representation of 6.75 using 4 integer bits and 4 fraction bits:

  01101100

  0110.1100

  $2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$

- As with whole numbers, negative fractional numbers can be represented in two ways
  - Sign/magnitude notation
  - Two's complement notation

- Example:
  - -2.375 using 8 bits (4 bits each to represent integer and fractional parts)
    - 2.375 = 0010 . 0110
    - Sign/magnitude notation: 1010 0110
    - Two's complement notation:

      1. flip all the bits: 1101 1001
      2. add 1: + 1

      1101 1010

Binary Addition

Adding binary numbers

- To add binary numbers is relatively simple
- We use the same principles of adding normal decimals
- However, our result needs to fit in with the rules of binary

$$1 + 1 = 10$$

Rules of Binary Addition

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 0 carry 1

1 + 1 + 1 = 1 carry 1

Example

- 1011 + 101

Align the numbers in columns, as you would with a normal decimal

```
  1 1 1 1
  1011
   101
 -----
 10000
```

Example

Example 1: 10001 + 11101

Solution:

```
      1
  1 0 0 0 1
(+) 1 1 1 0 1
-----------
  1 0 1 1 1 0
```

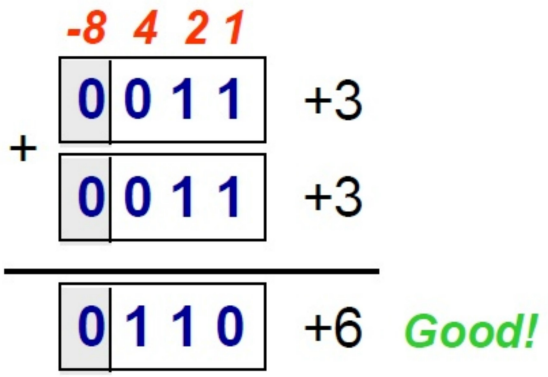
Example

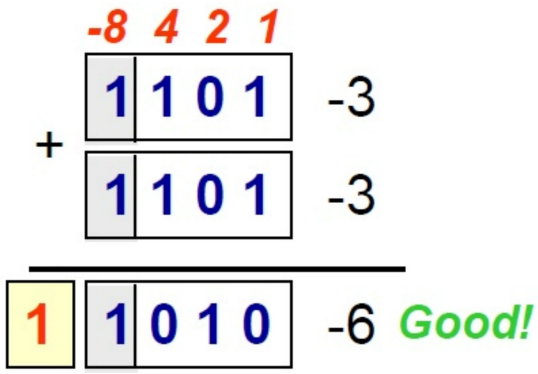
Example 2: 10111 + 110001

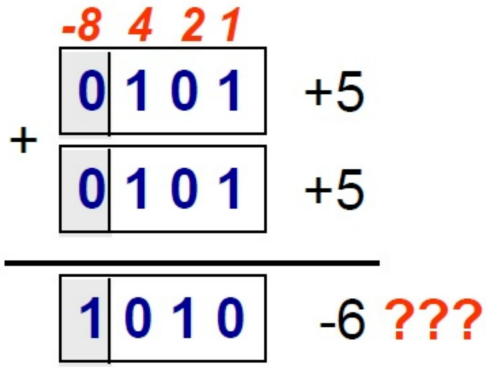
Solution:

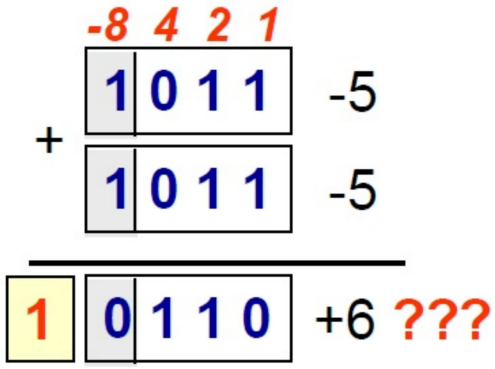
```
      111
    10111
(+) 110001
---------
  1001000
```

What do you think of the following?!!!

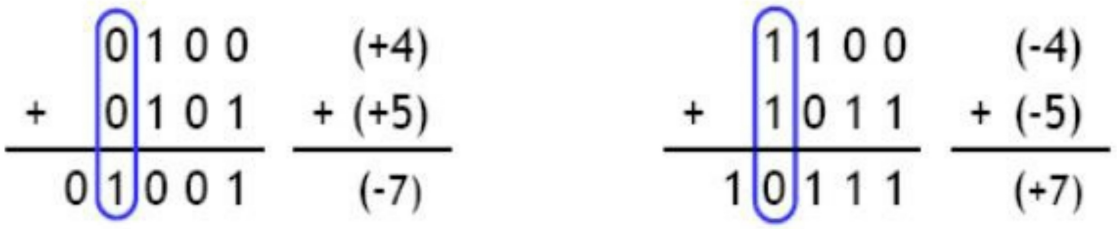






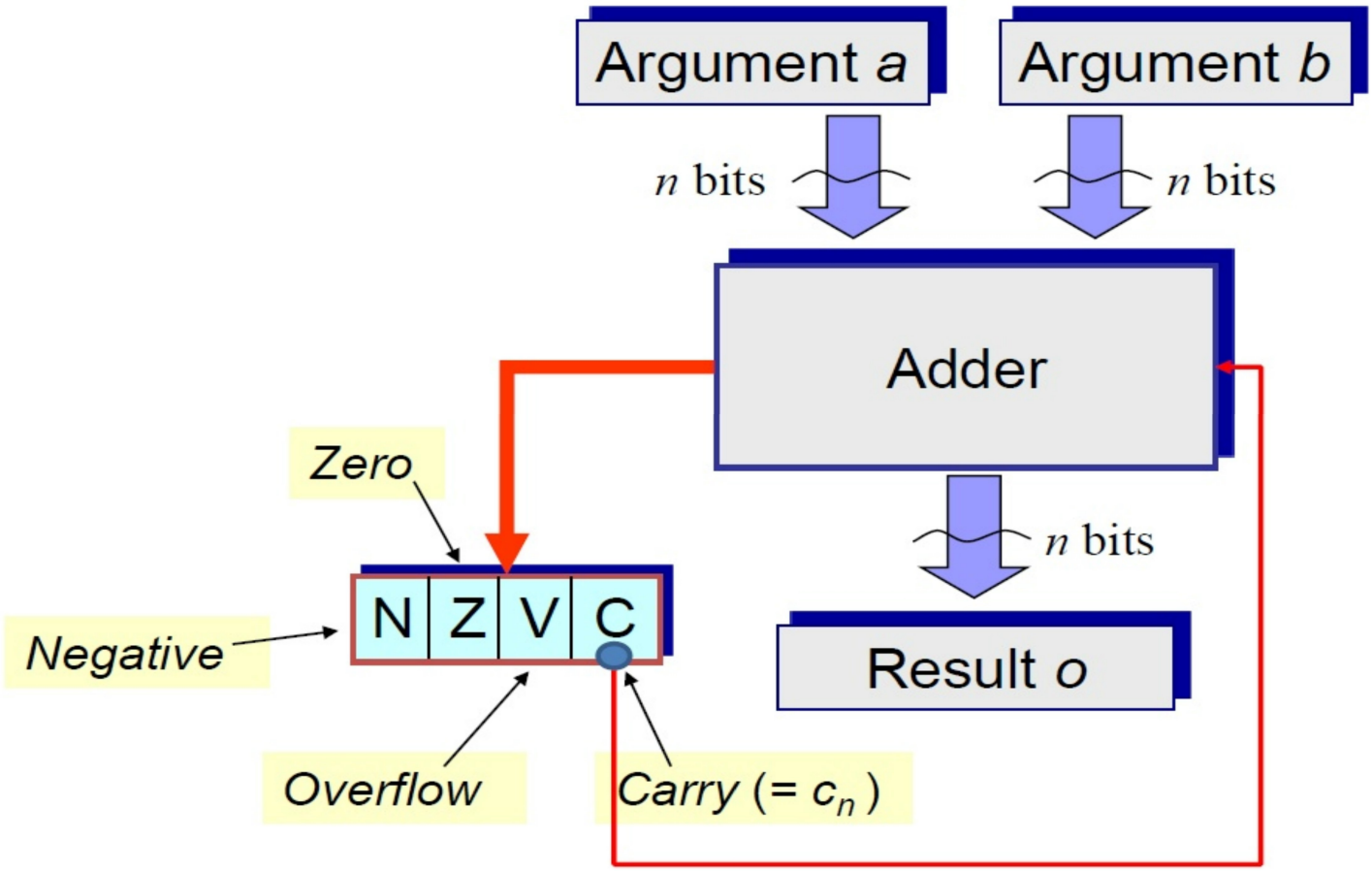


Overflow



- Overflow occurs only in the two situations above.
  1. If you add two *positive* numbers and get a *negative* result.
  2. If you add two *negative* numbers and get a *positive* result.
- Overflow can never occur when you add a positive number to a negative number. (Do you see why?)

Operation of an n-bit adder (condition register!)



Condition Code Register Bits N, Z, V, C

N bit is set if result of operation in negative (MSB = 1)

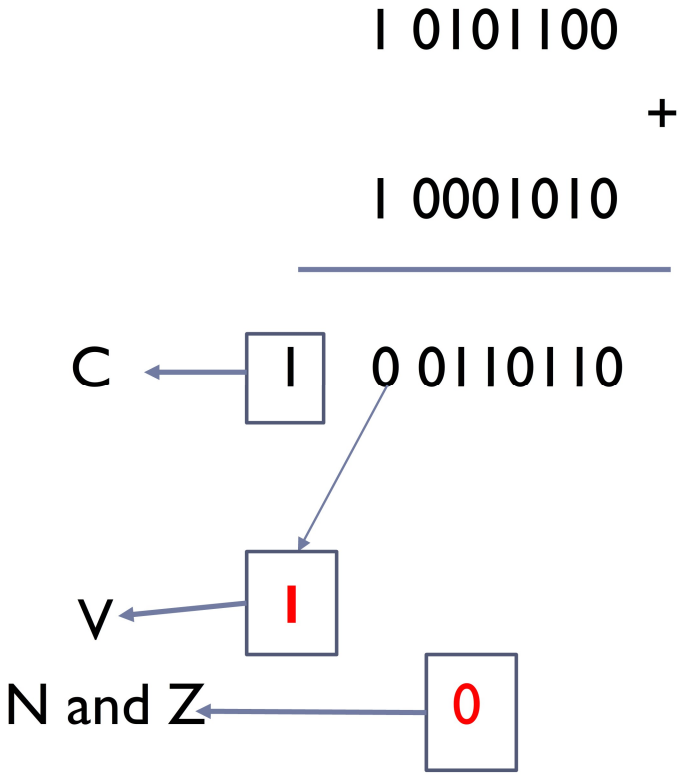
Z bit is set if result of operation is zero (All bits = 0)

V bit is set if operation produced an overflow

C bit is set if operation produced a carry (borrow on subtraction)

Note: Not all instructions change these bits of the CCR

Example (find N,Z,V and C)



Example

- Give the result in decimal and the NZVC conditions as a result of adding two 4-bit numbers in the U2 code

  0110

  100

- Where the "incoming" carry 0, Attention! The numbers are of different lengths!
- Separate the resulting number from the conditions with a point: x.NZVC

Solution :

- first we will extend the length of 100 to 4-bit long

  100 ⟶ 1100

- Second to add the two numbers

  0110 ⟵ [0]

  \+

  1100 Incoming carry

  1 0010

- 1 . 0001