Information Coding

Lecturer: Guillaume Beslon
(Lecture adapted from Lionel Morel)

Computer Science and Information Technologies - INSA Lyon

Fall 2025

Remember...

Example: Computing the sum of all integers from 1 to 100?

```
int main()
{
  int x,i;
  x = 0;
  i = 0;
  for (i = 1; i<100;i++)
    {
      x = x+1;
    }
  return x;
```

The **compiler** transforms the code into a sequence of instructions

```c
int main()
{
  int x,i;
  x = 0;
  i = 0;
  for (i = 1; i<100;i++)
  {
    x = x+1;
  }
  return x;
```

```
0x0000000100000f70 <+0>:    push   %rbp
0x0000000100000f71 <+1>:    mov    %rsp,%rbp
0x0000000100000f74 <+4>:    movl   $0x0,-0x4(%rbp)
0x0000000100000f7b <+11>:   movl   $0x0,-0x8(%rbp)
0x0000000100000f82 <+18>:   movl   $0x0,-0xc(%rbp)
0x0000000100000f89 <+25>:   movl   $0x1,-0xc(%rbp)
0x0000000100000f90 <+32>:   cmpl   $0x64,-0xc(%rbp)
0x0000000100000f94 <+36>:   jge    0x100000fb1 <main()+65>
0x0000000100000f9a <+42>:   mov    -0x8(%rbp),%eax
0x0000000100000f9d <+45>:   add    $0x1,%eax
0x0000000100000fa0 <+48>:   mov    %eax,-0x8(%rbp)
0x0000000100000fa3 <+51>:   mov    -0xc(%rbp),%eax
0x0000000100000fa6 <+54>:   add    $0x1,%eax
0x0000000100000fa9 <+57>:   mov    %eax,-0xc(%rbp)
0x0000000100000fac <+60>:   jmpq   0x100000f90 <main()+32>
0x0000000100000fb1 <+65>:   mov    -0x8(%rbp),%eax
0x0000000100000fb4 <+68>:   pop    %rbp
```

The **assembler** transforms the sequence of instructions into (weird) numbers

```
0x0000000100000f70 <+0>:     push   %rbp
0x0000000100000f71 <+1>:     mov    %rsp,%rbp
0x0000000100000f74 <+4>:     movl   $0x0,-0x4(%rbp)
0x0000000100000f7b <+11>:    movl   $0x0,-0x8(%rbp)
0x0000000100000f82 <+18>:    movl   $0x0,-0xc(%rbp)
0x0000000100000f89 <+25>:    movl   $0x1,-0xc(%rbp)
0x0000000100000f90 <+32>:    cmpl   $0x64,-0xc(%rbp)
0x0000000100000f94 <+36>:    jge    0x100000fb1 <main()+65>
0x0000000100000f9a <+42>:    mov    -0x8(%rbp),%eax
0x0000000100000f9d <+45>:    add    $0x1,%eax
0x0000000100000fa0 <+48>:    mov    %eax,-0x8(%rbp)
0x0000000100000fa3 <+51>:    mov    -0xc(%rbp),%eax
0x0000000100000fa6 <+54>:    add    $0x1,%eax
0x0000000100000fa9 <+57>:    mov    %eax,-0xc(%rbp)
0x0000000100000fac <+60>:    jmpq   0x100000f90 <main()+32>
0x0000000100000fb1 <+65>:    mov    -0x8(%rbp),%eax
0x0000000100000fb4 <+68>:    pop    %rbp
```

```
0000f70 55 48 89 e5 c7 45 fc 00 00 00 00 c7 45 f8 00 00
0000f80 00 00 c7 45 f4 00 00 00 00 c7 45 f4 01 00 00 00
0000f90 83 7d f4 64 0f 8d 17 00 00 00 8b 45 f8 83 c0 01
0000fa0 89 45 f8 8b 45 f4 83 c0 01 89 45 f4 e9 df ff ff
0000fb0 ff 8b 45 f8 5d c3 90 90 01 00 00 00 1c 00 00 00
0000fc0 00 00 00 00 1c 00 00 00 00 00 00 00 1c 00 00 00
0000fd0 02 00 00 00 70 0f 00 00 34 00 00 00 34 00 00 00
```

Who reads/writes what?

Programmers...

... usually manipulate complex instructions and numbers (Integers, Reals, etc)

but at the end programs...

... are made of numbers (Integers, Reals, etc)

and machines ....

... only understand **binary**

⇒ We need a way to transform numbers into binary sequences and binary sequences into numbers.

Just do it...

0100101101010010010101001010111010101001010111110111
0100010001001111110010101001101010101110101110100100
1000100010100110001010101001001000100100000100010001
0011101100100101001000101001000101111001001010010001
1111001001001010100010100111001010000001011011010001
1001000101100100101001000101100100011111001001001010
0110001010101001001000100100000001000100010011101100
1001010010001100100100101010001010011100100010000001
0110110100011001000101001001010010001011010010001111
1001001001010011000101010010101000101001110010001000
0001011011010001100100010110010010100100010110100100
011111001001001010011000101010100100100010001...

Bit vectors (aka “Words”)

Basic information: the bit $\in \{0, 1\}$ (for *b*inary dig*it*)

Example of word: 01010101110010011100001111

We work with finite words of predefined length l (why?)

l is the number of bits: in practice $l = 8 * n$

n is the number of bytes (octets in french)

When programming (on a 32 or 64 bits machine):

- $n = 1$ is called a `byte` or a `char`
- $n = 2$ is called a `short`
- $n = 4$ is *usually* called an int (or `float` if it represents a pseudo-real number)
- $n = 8$ is called `long long` (or `double` for a pseudo-real)

Beware, these are only *naming conventions*!
So, be always sure that you know what you are talking about!

Length conventions help

Example with 32 bits words (4 bytes)

01001011010100100101010010101110101010010101111110111
01000100010011111100101010011010101011101011110100100
10001000101001100010101010010010001001000001000010001
00111011001001010010001010010001011110010010100100 01
11110010010010101000101001110010100000010110110100 01
10010001011001001010010001011001000111110010010010 10
01100010101010010010001001000000010001000100111011 00
10010100100011001001001010100010100111001000100000 01
01101101000110010001010010010100100010110100100011 11
10010010010100110001010100101010001010011100100010 00
00010110110100011001000101100100101001000101101001 00
0111110010010010100110001010101001001000100 1...

But we still need to transform binary words into numbers...

Natural numbers in base 2 (Unsigned integers)

Let x be a vector of n bits:
$x_{n-1}, x_{n-2}, \cdots, x_1, x_0$, with $x_i \in 0, 1$ (in base 2).

Using positional notation[1] we can interpret the value of x as a natural number:

$$x = \sum_{i=0}^{n-1} x_i \cdot 2^i$$

2^n different values can be represented:

$$0 \leq x \leq 2^n - 1$$

[1] Note that positional notation is what you use on an everyday basis when you manipulate decimal numbers. Always remember that mechanisms are similar (see the 'protopoly' for a generalization to base $\beta > 1$)

Blackboard examples

LSB: Least Significant Bit (Byte)

LSBit is the bit position in a binary integer giving the units value, that is, determining whether the number is even or odd[2].

1	0	0	1	0	1	0	1

LSByte is the byte in that position of a multi-byte number which has the least potential value.

[2] https://en.wikipedia.org/wiki/Least_significant_bit

MSB: Most Significant Bit (Byte)

MSBit is the bit position in a binary number having the greatest value[3].

1	0	0	1	0	1	0	1

MSByte is the byte (or octet) in that position of a multi-byte number which has the greatest potential value.

[3] https://en.wikipedia.org/wiki/Most_significant_bit

Warning: MSB/LSB not to be confused with “little-endian” and “big-endian”

- MSB/LSB are to be understood relatively to the **Binary code**.
- “Little-Endian” and “Big-Endian” are ways to organise the bytes of a word into the **physical memory** of a computer...

→ Comics vs. Mangas

- “Big-Endian”: The MSByte is stored first (mangas)...
- “Little-Endian”: The LSBytes is stored first (comics)...

→ When you directly look at the memory content, be careful to the “endianness” (aka “boutisme” en Français)...
→ Named after Jonathan Swift’s “Gulliver’s Travel” (1726)

WARNING

When computing with integers, computations are mathematically exact EXCEPT if the value exceeds the limits of the code (given the size of the word)...

In that case... Well, the shit hits the fan and its called an oVerflow “V” (RIP Ariane V 501 flight)

Notations and important values[4]

We write: $(x_{n-1}, x_{n-2}, ..., x_1, x_0)_\beta$
when writing x in base β.
eg:

- $(101)_2 = (5)_{10}$
- $(1010)_2 = (10)_{10}$

for $n = 8$:

- 256 different values
- max: $2^8 - 1 = 255$

for $n = 32$:

- $\approx$ 4 billions different values
- max: $2^{32} - 1 \approx 4 \times 10^9$

for $n = 16$:

- 65.536 different values
- max: $2^{16} - 1 = 65.535$

for $n = 64$:

- **A lot but not a infinite number of values...**
- max: $2^{64} - 1 \approx 16 \times 10^{18}$

[4]These values are fundamental to any serious computer science work! Probably one of the rare things to be known by heart

Fast equivalence: the trick!

$$2^{10} = 1024 \approx 10^3 = 1000$$

Example:

$$2^{32} = 2^{30} \times 2^2 = 2^{10^3} \times 4 \approx 1000^3 \times 4 = 4.000.000.000$$

Binary $\to$ Decimal Conversion

Any number $p \in \mathbb{N}$ can be represented in a unique positional form in base 2, using n bits (with $n = \lfloor log_2(p) \rfloor + 1$):

$$(x_{n-1}x_{n-2}\cdots x_1 x_0)_2 := \sum_{i=0}^{n-1} x_i 2^i.$$

$\Rightarrow$ Binary to decimal conversion is straightforward

NB: bits are numbered from 0 to $n-1$

Binary ← Decimal Conversion

The remainder of the euclidean division of x by 2 gives the right-most digit of its representation in base 2:

$$
\begin{aligned}
x &= \sum_{i=0}^{n-1} x_i 2^i \\
x &= x_{n-1}\cdot 2^{n-1} + x_{n-2}\cdot 2^{n-2} + \cdots + x_2\cdot 2^2 + x_1\cdot 2^1 + x_0 \\
&= \underbrace{\left(x_{n-1}\cdot 2^{n-2} + x_{p-2}\cdot 2^{n-3} + \cdots + x_2\cdot 2^1 + x_1\right)}_{\text{quotient}}\cdot 2 + \underbrace{x_0}_{\text{remainder}}
\end{aligned}
$$

We get all the digits of the binary representation of a natural number x by applying euclidean divisions (by 2) to the successive quotients until we reach 0 as a quotient.

NB: This gives least-significant bits first!

Again, see "poly" for a generalization to any base $\beta > 1$

Binary $\leftarrow$ Decimal Conversion - example

To convert $n = (423)_{10}$ to binary, we

$$\begin{array}{rcrcl}
423 & = & 211 & \times & 2+1 \\
211 & = & 105 & \times & 2+1 \\
105 & = & 52 & \times & 2+1 \\
52 & = & 26 & \times & 2+0 \\
26 & = & 13 & \times & 2+0 \\
13 & = & 6 & \times & 2+1 \\
6 & = & 3 & \times & 2+0 \\
3 & = & 1 & \times & 2+1 \\
1 & = & 0 & \times & 2+1
\end{array}$$

From this we deduce that: $(423)_{10} = (110100111)_2$

Blackboard examples

Fine, we are able to represent natural integers...
What about signed integers?

Signed Integers?

Two ways of representing $-x$:

- 1 bit for the sign, the rest for $|x|$

$$x_{n-1} = \begin{cases} 0 & \text{if } x \geq 0, \\ 1 & \text{if } x < 0. \end{cases} \quad \text{and} \quad (x_{n-2}, x_{n-3}, \ldots, x_1, x_0) = |x|$$

  - Pros: Simple to understand
  - Cons: 2 writings for 0
  - Cons: hardware implementation $\neq$ unsigned integers
- Two's complement
  - Cons: Less easy to understand
  - Pros: numbering and hardware implementation is unchanged
    $\rightarrow$ Used in 99.99% of digital circuits[5]

[5] But other solutions are also used, e.g. for floating values...

Two's Complement - math

- Let x be a vector of n bits: $x_{n-1}, x_{n-2}, \cdots, x_1, x_0$, with $x_i \in 0, 1$
  The value of x interpreted as a signed integer is:

$$x = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i.$$

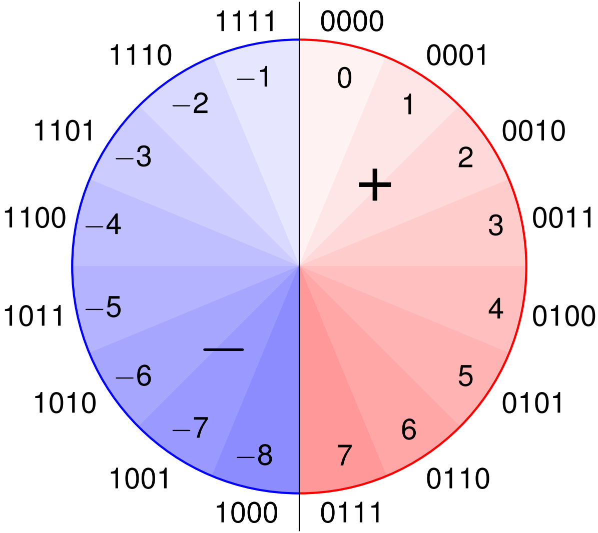
- Still only 2^n different values can be represented for a word of size n
- oVerflow may still happen but NOT AT THE SAME VALUE than for natural integers:

$$-2^{n-1} \leq x < 2^{n-1} - 1$$

e.g. for 32 bits,

$$-2.147.483.648 \leq x < 2.147.483.647$$

Two's Complement - intuition[6]



[6] credit: Benoît Lopez

Two's Complement - compute the opposite of x

- The opposite of a number is obtained by inverting all the bits of the word, then adding 1.
- Mathematically, if:

$$x = x_{n-1}, x_{n-2}, \cdots, x_1, x_0$$

then

$$-x = (\overline{x_{n-1}}, \overline{x_{n-2}}, \cdots, \overline{x_1}, \overline{x_0}) + 1$$

with $\overline{a}$ being the complement of bit a:

$$\overline{a} = \begin{cases} 1, & \text{when } a = 0 \\ 0, & \text{when } a = 1 \end{cases}$$

- This algorithm computes the opposite of a number, being it initially positive or negative!

Blackboard examples

Number Extension

How do we assign a n-bits vector to an m-bits vector?

- $n > m$, truncate $\rightarrow$ lose most significant bits
- $n < m$, requires a sign extension

Consider $(x_{n-1}, x_{n-2}, \ldots, x_1, x_0)$. Let's write it as $(y_{m-1}, y_{m-2}, ..., y_n, y_{n-1}, y_{n-2}, ..., y_1, y_0)$. The value of x is preserved if we take:

$$y_{m-1} = x_{n-1}, y_{m-2} = x_{n-1}, \ldots, y_n = x_{n-1}, y_{n-1} = x_{n-1},$$

$$y_{n-2} = x_{n-2}, \ldots, y_1 = x_1, y_0 = x_0$$

The Hexadecimal: a useful way to deal with binary numbers

Reading binary is a pain in the ass! In practice, everybody uses **hexadecimal**:

binary	hexa	decimal	binary	hexa	decimal
0000	0x0	0	1000	0x8	8
0001	0x1	1	1001	0x9	9
0010	0x2	2	1010	0xA	10
0011	0x3	3	1011	0xB	11
0100	0x4	4	1100	0xC	12
0101	0x5	5	1101	0xD	13
0110	0x6	6	1110	0xE	14
0111	0x7	7	1111	0xF	15

Max values:

- on 1 byte: 0xFF = 255_{10}
- on 2 bytes: 0xFFFF = 65.535_{10}
- on 4 bytes: 0xFFFF.FFFF = $4.294.967.295_{10}$

Useful Notation: 0x...

Remember...

0100101101010010010101001010111010101001010111110111
0100010001001111110010101001101010101110101110100100
1000100010100110001010101001001000100100000100010001
0011101100100101001000101001000101111001001010010001
1111001001001010100010100111001010000001011011010001
1001000101100100101001000101100100011111001001001010
0110001010101001001000100100000001000100010011101100
1001010010001100100100101010001010011100100010000001
0110110100011001000101001001010010001011010010001111
1001001001010011000101010010101000101001110010001000
0001011011010001100100010110010010100100010110100100
01111100100100101001100010101010010010001001...

Same in hexadecimal is more compact and (almost) human friendly...

4B5254AEA95F7444FCA999EBA488A62A92241113B2
5229179291F24A8A72816D19164A4591F24A62A922
40444EC948C92A29C8816D1914948B48F9253152A2
9C8816D19164A45A47C9298AA489...

Other interpretation of bit vectors

As soon as you are able to transform bit vectors into integers, you can code anything providing you have a conversion table and/or a conversion formula
$\rightarrow$ Digital (in French “Numérique”) corresponds to this idea of transforming everything into numbers that computers are able to deal with...

- (Pseudo-)real numbers
- Characters/symbols: encodings such as ASCII or ISO or UTF-8.
- images (PNG, JPEG...), music (MP3, OGG...), video (MOV, AVI, MP4...)
- Instructions = programs in their “final” form (that which is interpreted by HW).
- ...

All this necessitates interpreting bit vectors!

Floating points

Integers have a constant precision but a very low *dynamics*...

- Difficult to represent very large values (e.g. mass of the universe in kilograms)
- Impossible to represent very small values (e.g. mass of an electron in kilograms)
- Encoding of useless (and sometimes misleading) LSB

Solution: scientific notation (3×10^{52} and $9,109 \times 10^{-31}$)

$\Rightarrow$Floating points are the binary analogs to scientific notation. Pseudo-real numbers are represented by a *mantissa* (or *fraction*) and an *exponent* coded by (still finite) bit vectors.

Floating points

Floating points are normalised by the IEEE 754 format. Mainly two formats are used:

- Single precision (32 bits): sign (1 bit), exponent (8 bits, including exponent sign), fraction (23 bits) $\rightarrow$ `float`
- Double precision (64 bits): sign (1 bit), exponent (11 bits, including exponent sign), fraction (52 bits) $\rightarrow$ `double`

Floating points are NOT reals. The maximum number of values one can encode with a `float` is still $2^{32} = 4.294.967.296$!

$\Rightarrow$ Floating points have precision issues that are (1) too difficult to be listed here and (2) extremely dangerous!
e.g., in general $a + b + c \neq b + a + c$ and $b + a - b - a \neq 0$
(RIP Sleipner A offshore platform, 1991)

Character encoding — ASCII

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Character encoding — UTF-8

Character		Binary code point	Binary UTF-8	Hexadecimal UTF-8
$	U+0024	010 0100	00100100	24
¢	U+00A2	000 1010 0010	11000010 10100010	C2 A2
€	U+20AC	0010 0000 1010 1100	11100010 10000010 10101100	E2 82 AC
𐍈	U+10348	0 0001 0000 0011 0100 1000	11110000 10010000 10001101 10001000	F0 90 8D 88

Instruction encoding

```
0x0000000100000f70 <+0>:     push   %rbp
0x0000000100000f71 <+1>:     mov    %rsp,%rbp
0x0000000100000f74 <+4>:     movl   $0x0,-0x4(%rbp)
0x0000000100000f7b <+11>:    movl   $0x0,-0x8(%rbp)
0x0000000100000f82 <+18>:    movl   $0x0,-0xc(%rbp)
0x0000000100000f89 <+25>:    movl   $0x1,-0xc(%rbp)
0x0000000100000f90 <+32>:    cmpl   $0x64,-0xc(%rbp)
0x0000000100000f94 <+36>:    jge    0x100000fb1 <main()+65>
0x0000000100000f9a <+42>:    mov    -0x8(%rbp),%eax
0x0000000100000f9d <+45>:    add    $0x1,%eax
0x0000000100000fa0 <+48>:    mov    %eax,-0x8(%rbp)
0x0000000100000fa3 <+51>:    mov    -0xc(%rbp),%eax
0x0000000100000fa6 <+54>:    add    $0x1,%eax
0x0000000100000fa9 <+57>:    mov    %eax,-0xc(%rbp)
0x0000000100000fac <+60>:    jmpq   0x100000f90 <main()+32>
0x0000000100000fb1 <+65>:    mov    -0x8(%rbp),%eax
0x0000000100000fb4 <+68>:    pop    %rbp
```

```
0000f70 55 48 89 e5 c7 45 fc 00 00 00 00 c7 45 f8 00 00
0000f80 00 00 c7 45 f4 00 00 00 00 c7 45 f4 01 00 00 00
0000f90 83 7d f4 64 0f 8d 17 00 00 00 8b 45 f8 83 c0 01
0000fa0 89 45 f8 8b 45 f4 83 c0 01 89 45 f4 e9 df ff ff
0000fb0 ff 8b 45 f8 5d c3 90 90 01 00 00 00 1c 00 00 00
0000fc0 00 00 00 00 1c 00 00 00 00 00 00 00 1c 00 00 00
0000fd0 02 00 00 00 70 0f 00 00 34 00 00 00 34 00 00 00
```

Instructions can be encoded as binary vectors with two parts: The “opcode” and the “operands”

```
0x0000000100000f70 <+0>:     push   %rbp
0x0000000100000f71 <+1>:     mov    %rsp,%rbp
0x0000000100000f74 <+4>:     movl   $0x0,-0x4(%rbp)
0x0000000100000f7b <+11>:    movl   $0x0,-0x8(%rbp)
0x0000000100000f82 <+18>:    movl   $0x0,-0xc(%rbp)
0x0000000100000f89 <+25>:    movl   $0x1,-0xc(%rbp)
0x0000000100000f90 <+32>:    cmpl   $0x64,-0xc(%rbp)
0x0000000100000f94 <+36>:    jge    0x100000fb1 <main()+65>
0x0000000100000f9a <+42>:    mov    -0x8(%rbp),%eax
0x0000000100000f9d <+45>:    add    $0x1,%eax
0x0000000100000fa0 <+48>:    mov    %eax,-0x8(%rbp)
0x0000000100000fa3 <+51>:    mov    -0xc(%rbp),%eax
0x0000000100000fa6 <+54>:    add    $0x1,%eax
0x0000000100000fa9 <+57>:    mov    %eax,-0xc(%rbp)
0x0000000100000fac <+60>:    jmpq   0x100000f90 <main()+32>
0x0000000100000fb1 <+65>:    mov    -0x8(%rbp),%eax
0x0000000100000fb4 <+68>:    pop    %rbp
```

⇩

```
0000f70 55 48 89 e5 c7 45 fc 00 00 00 00 c7 45 f8 00 00
0000f80 00 00 c7 45 f4 00 00 00 00 c7 45 f4 01 00 00 00
0000f90 83 7d f4 64 0f 8d 17 00 00 00 8b 45 f8 83 c0 01
0000fa0 89 45 f8 8b 45 f4 83 c0 01 89 45 f4 e9 df ff ff
0000fb0 ff 8b 45 f8 5d c3 90 90 01 00 00 00 1c 00 00 00
0000fc0 00 00 00 00 1c 00 00 00 00 00 00 00 1c 00 00 00
0000fd0 02 00 00 00 70 0f 00 00 34 00 00 00 34 00 00 00
```

Take Home Message

1. Binary information is encoded in bit vectors which length is of utmost importance
2. Bit vectors can be interpreted as “integers”
3. Bit vectors can be interpreted as “reals” (floating points)
4. Integers can be interpreted as almost anything (“digitalization”)
5. Numbers in computers don’t behave exactly as in math!
6. Bit vectors can be interpreted as instructions

$\Rightarrow$ So we are able to code **data** AND **instructions**...
$\Rightarrow$ We’ve made a first step towards von Neumann’s machines...

Next step...

Combinatorial Logic and Circuits!!

ie, how to process information encoded in bit vectors?