Hex Notation

Base 16 (hexidecimal), positional notation for numbers:

"**0x**", "**x**", and "**h**" indicate hex representation for **C**, **LC3as**, and **verilog**, respectively

x a **base-16 digit** (hex). May also represent the **value** of that digit.

x_i the base-16 **digit in the i-th place**. May also represent the **value** of that digit.

hex representation		value
$x_3 \; x_2 \; x_1 \; x_0$	→	$x_3 \cdot 16^3 + x_2 \cdot 16^2 + x_1 \cdot 16^1 + x_0 \cdot 16^0$
1 2 3 4	→	$1 \cdot 16^3 + 2 \cdot 16^2 + 3 \cdot 16^1 + 4 \cdot 16^0$

values of hex digits

Hex Digit	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

Hex Digit	Binary	Decimal
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Hex-2-Binary

(hex digit) ===> (4-bit binary)

$$(1\ 2\ 3\ 4)$$

$$= (1\cdot 16^3 + 2\cdot 16^2 + 3\cdot 16^1 + 4\cdot 16^0)$$

$$= ((2^0)\cdot 16^3 + (2^1)\cdot 16^2 + (2^1+2^0)\cdot 16^1 + (2^2)\cdot 16^0)$$

$$= ((2^0)\cdot (2^4)^3 + (2^1)\cdot (2^4)^2 + (2^1+2^0)\cdot (2^4)^1 + (2^2)\cdot (2^4)^0)$$

$$= ((2^0)\, 2^{12} + (2^1)\, 2^8 + (2^1+2^0)\, 2^4 + (2^2)\, 2^0)$$

$$= ((0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0)\, 2^{12} +$$
$$(0\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0)\, 2^8 +$$
$$(0\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0)\, 2^4 +$$
$$(0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 0\cdot 2^0)\, 2^0)$$

$$= ((0\cdot 2^{15} + 0\cdot 2^{14} + 0\cdot 2^{13} + 1\cdot 2^{12}) +$$
$$(0\cdot 2^{11} + 0\cdot 2^{10} + 1\cdot 2^9 + 0\cdot 2^8) +$$
$$(0\cdot 2^7 + 0\cdot 2^6 + 1\cdot 2^5 + 1\cdot 2^4) +$$
$$(0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 0\cdot 2^0))$$

$$= (0001\ 0010\ 0011\ 0100)$$

$$(b_3\cdot 2^3 + b_2\cdot 2^2 + b_1\cdot 2^1 + b_0\cdot 2^0)(2^4)^k = b_3 b_2 b_1 b_0 \overbrace{0000 \cdots 0}^{4k}$$

$$(b_3\cdot 2^3 + b_2\cdot 2^2 + b_1\cdot 2^1 + b_0\cdot 2^0)(2^4)^{k-1} = b_3 b_2 b_1 b_0 \overbrace{00 \cdots 0}^{4k-4}$$

LC3 Simulators — See execution, check results, debug code

Using a Simulator we can

--- **See Machine's Content**
Registers: R0-R7, IR, PC, MAR, MDR, PSR (in hex notation)
Memory: address/content (in hex, w/ translation to .asm)
Branch conditions: CC (usually as "Z" or "N" or "P")

--- **Alter Machine's Content** (except CC)
Registers
Memory location

--- **Execute instructions**:
STEP (**1 instruction**)
RUN (**w/o stopping**)
STOP (**stop running**)
BREAK (**stop** when PC points to **a particular memory location**)

--- **Set breakpoints**:
Mark memory locations for BREAK

Most things work via double-clicking.
Breakpoint set: click a memory line or square icon.

----

projects/LC3-tools/PennSim.jar

--- Double-click items to change them. Hardware is slightly different from our LC3 and from PP's LC3.
Don't use scroll bars, use up/down arrows on your keyboard.

----

src/LC3-tools/LC3sim: commandline simulator
---See src/Makefile for compiling.

NB--The Makefile compiles the tools, then moves the executables to /bin. I found that LC3sim and LC3sim-tk need to be moved back to src/LC3-tools for them to work. LC3-tools also has other executables (assembler and others) that we will need to use when we get to assembly language programming.

LC3 OS services

An **OS** is **software** that is **pre-loaded into memory**.
Preloading is booting in an actual machine.
The OS provides **services for programs**.

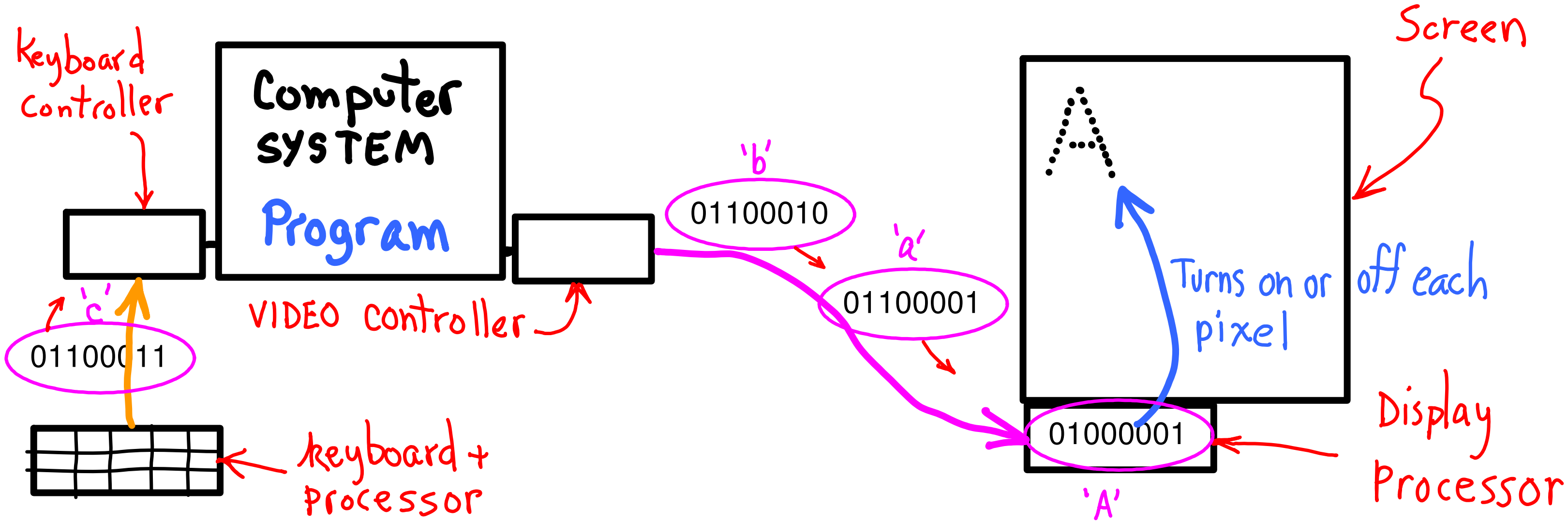
Some LC3 simulators (not ours) preloads a very primitive "OS". It is called "LC3os", and here it is (almost all of it):

TRAP	x25	--**Halt**: stop machine w/ message.
	x20	--**Getc**: one char, **keyboard** ==> **R0**[7:0] (clears **R0** first).
	x21	--**Out**: one char, **R0**[7:0] ==> display.
	x22	--**Puts**: **Mem**[**R0**] ==> display (until x0000 found).
	x23	--**In**: prompts, then one char input ala Getc.
	x24	--**Putsp**: Puts, but for packed data (2 chars per word).

We can load the same OS using our testbenches. The source code is in src/. The instructions to build and load it are in the Makefile.

Bits

what you see is NOT what you get.

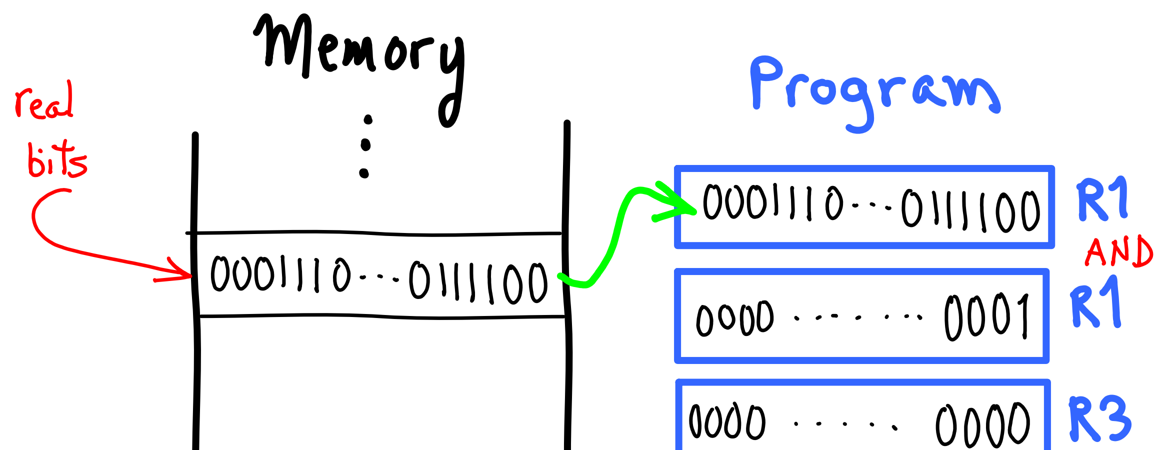


ASCII codes		ASCII codes	
...		...	
x30	**0**	x41	**A**
x31	**1**	x42	**B**
...		x43	**C**
x38	**8**	...	
x39	**9**	x5A	**Z**
...		...	

Program executes
--- **decides** what you need to see
--- **sends codes** to video controller

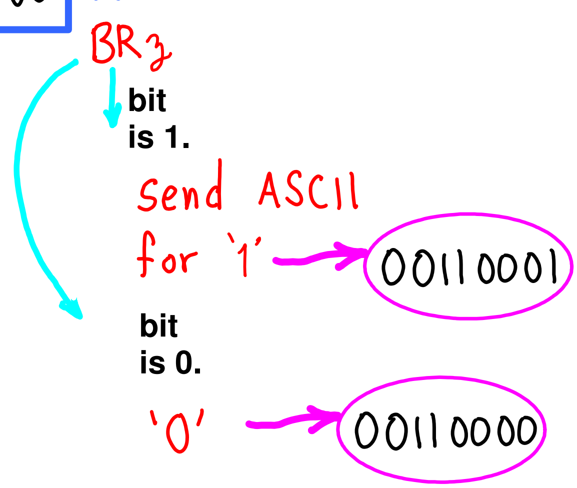
Video controller
--- **sends** to display device

Display processor
--- **turns pixels** on or off



Suppose we want to **see content of memory**.

--- **LD R1**, **<address>**
--- **detect bit**
--- **send ASCII**
--- **detect next bit**
--- **send ASCII**
. . .



DISPLAY

Programmer:

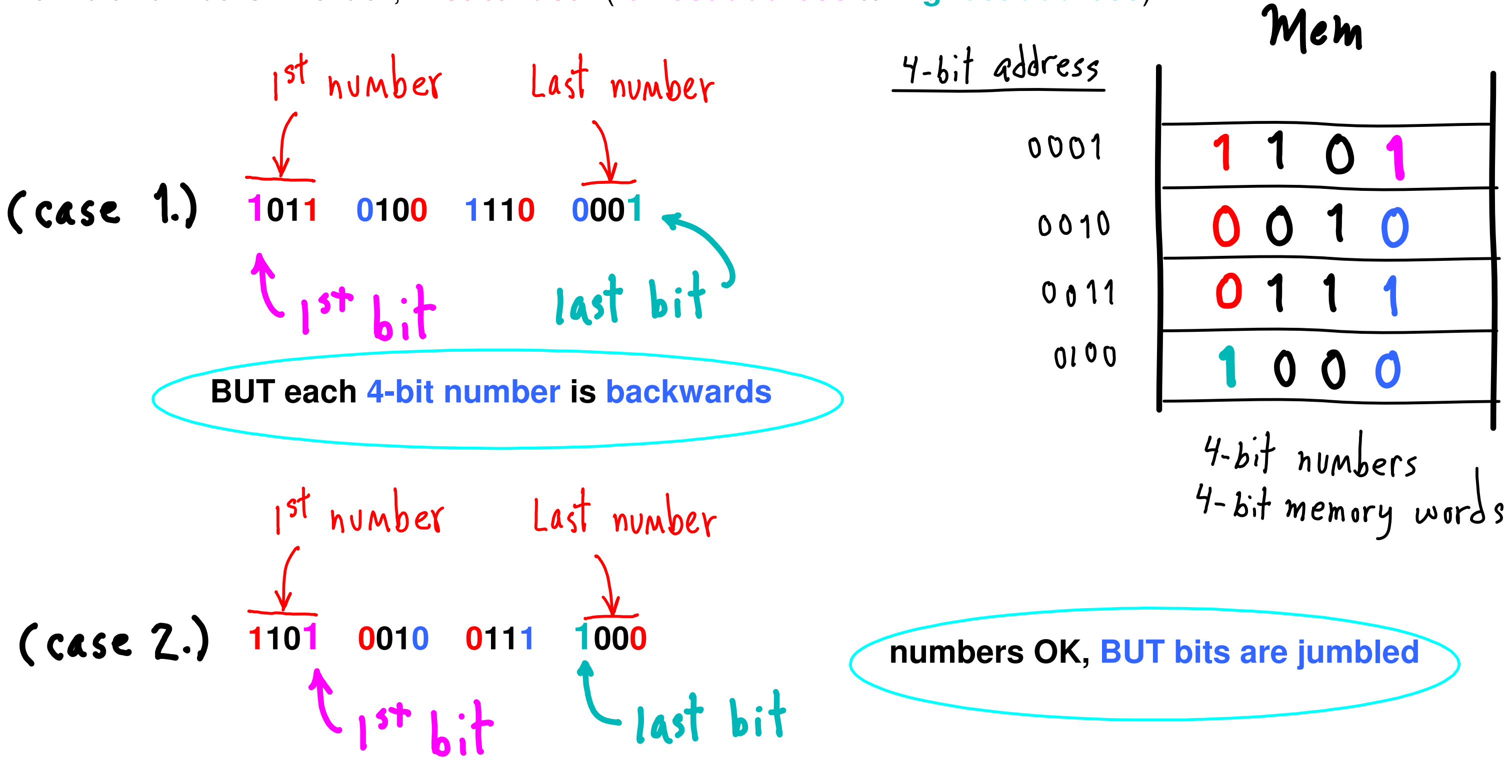
"Decisions, decisions!"
"What **order** should the **bits appear**?"
"Which is **low-order bit**?"

Possibilities (bits)

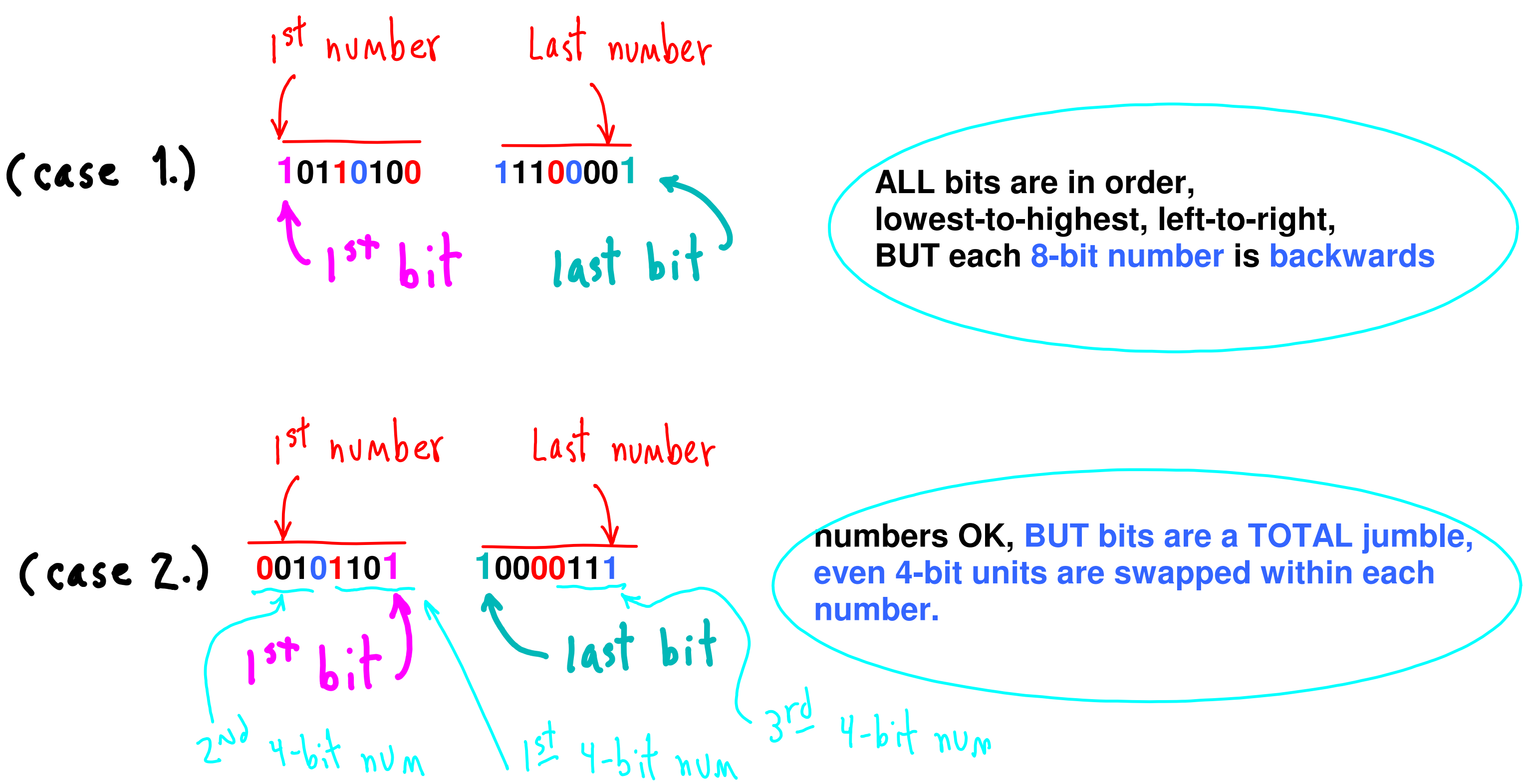
(1.) $b_0\ b_1\ b_2\ \ldots\ b_{15}$ — **Lowest bit** to **Highest bit**,
BUT doesn't look like a number.

(2.) $b_{15}\ \ldots\ b_2\ b_1\ b_0$ — **Highest bit** to **Lowest bit**,
BUT printed in backwards order.

Print 4-bit numbers in order, first-to-last (lowest-address to highest-address)



Print 8-bit numbers in order, first-to-last (lowest-address to highest-address)



Basic problem: **NUMBERS are Arabic** (right-to-left), and our **writing is** left-to-right. In olden times, numbers were expressed in writing, e.g., "**four and twenty blackbirds**", left-to-right, in writing order.
If we **wrote numbers** left-to-right, least-significant-to-most-significant, case (1.) would be perfect.
If we **wrote everything** right-to-left, case (1.) would have all bits reversed, and be perfect, too.

More on the difference between ascii representation of bits and actual bits. At a unix terminal window, enter

```
echo "abcd" | od -t x1
```

You will see the ascii codes for each byte that **echo** sent to **od** (plus an extra byte for an assumed end-of-line), expressed in hex:

```
61   62   63   64   0a
```

We would naturally think of this as the bytes of memory left-to-right. Now enter this,

```
echo "1234" | od -t x1
```

You will again see ascii codes. The "real" bits for the first character, "1", are equivalent to x31, or 00110001 in actual bits. Next change the "x1" to "x2", and to "x4". You will see this,

```
31   32   33   34

  3231    3433

   34333231
```

If you think of the first byte in memory as containing the least-significant bits of a number, it would depend on the number of bytes the number had as to which byte you display first. If the number has 16 bits, then the first 16 bits would be expressed 3231 in hex, but if it was a 32-bit number, you would display 34333231 in hex.

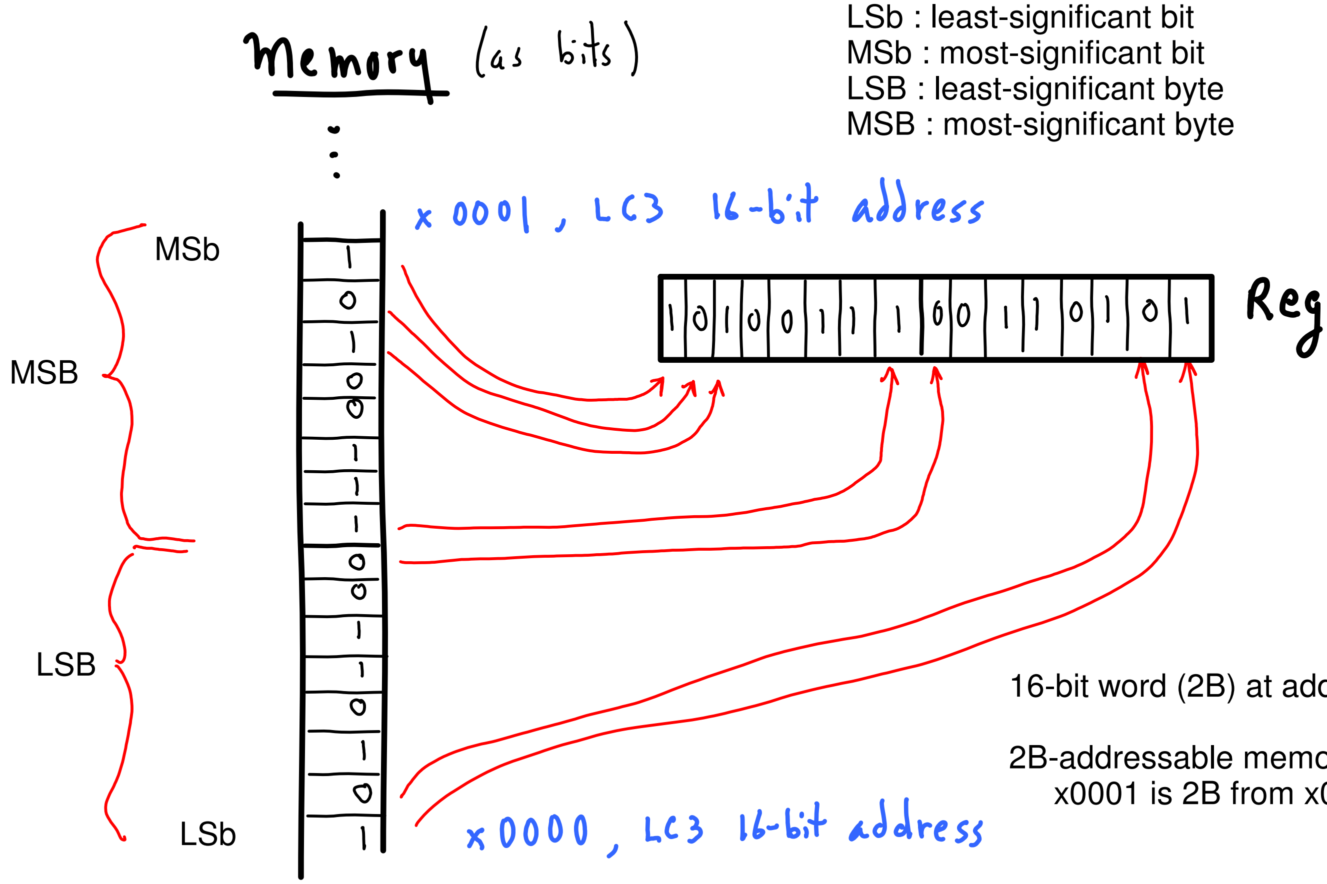
But, if we read things in right-to-left order, thinking of memory as laid out right-to-left, and printing bytes right-to-left, we would have,

```
        "4321"

"4"    "3"    "2"    "1"

34    33     32    31

 3433         3231

    34333231
```

In all cases, the least-significant bit is the rightmost, the least significant byte is the rightmost, and so forth. To accommodate the switching back and forth (and some other less important reasons), some machines put the most-significant byte of a number in the lowest byte address (called "big endian", as opposed to "little endian").

Last word on bit Layout in Mem, reg, ...

LSb : least-significant bit
MSb : most-significant bit
LSB : least-significant byte
MSB : most-significant byte



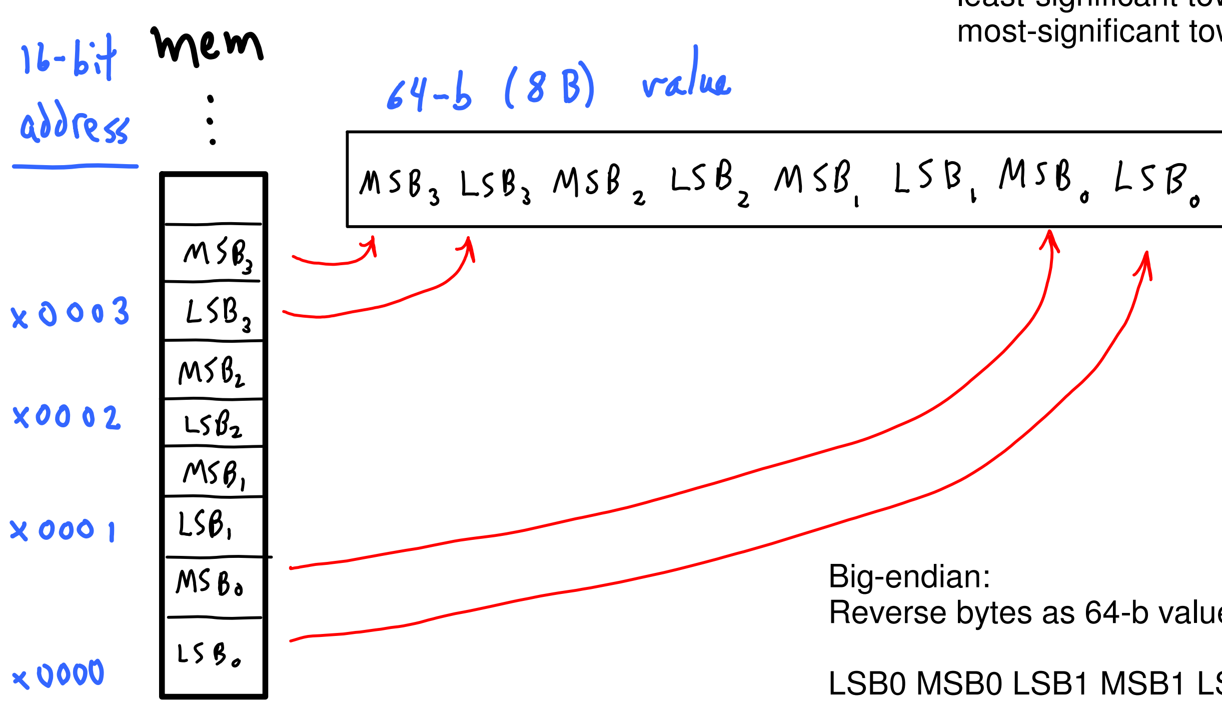
16-bit word (2B) at address x0000

2B-addressable memory:
x0001 is 2B from x0000

1B-addressable memory:
17-bit address 00000000000000000 ==> LSB
17-bit address 00000000000000001 ==> MSB

Small-end of memory: address x0000
Big-end of memory : address xFFFF

Little-endian:
least-significant toward small end
most-significant toward big end



Big-endian:
Reverse bytes as 64-b value:

LSB0 MSB0 LSB1 MSB1 LSB2 MSB2 ...