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
$x_3 \; x_2 \; x_1 \; x_0$	$\rightarrow$	$x_3 \cdot 16^3 + x_2 \cdot 16^2 + x_1 \cdot 16^1 + x_0 \cdot 16^0$
$1 \; 2 \; 3 \; 4$	$\rightarrow$	$1 \cdot 16^3 + 2 \cdot 16^2 + 3 \cdot 16^1 + 4 \cdot 16^0$

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

LSb : **L**east-**S**ignificant **bit**
MSb : **M**ost-**S**ignificant **bit**

10110010 11110000

LSB : **L**east-**S**ignificant **Byte**
MSB : **M**ost-**S**ignificant **Byte**

Hex-2-Binary

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

1 2 3 4

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

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

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

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

$$(0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0)\, 2^{0}$$

$$= 0001\ 0010\ 0011\ 0100$$

in general, when each hex digit converted to 4-bit binary:

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

$$+ (b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0)\underbrace{(2^4)^{k-1}}_{16^{k-1}} = b_3 b_2 b_1 b_0 \underbrace{00 \cdots 0}_{4k-4}$$

multiply by 2^n

= Left shift n bit positions

LC3 Simulators

See execution, check results, debug code

Using a Simulator we can

--- **See Machine's Content**
Registers: R0-R7, IR, PC, PSR (in hex)
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.

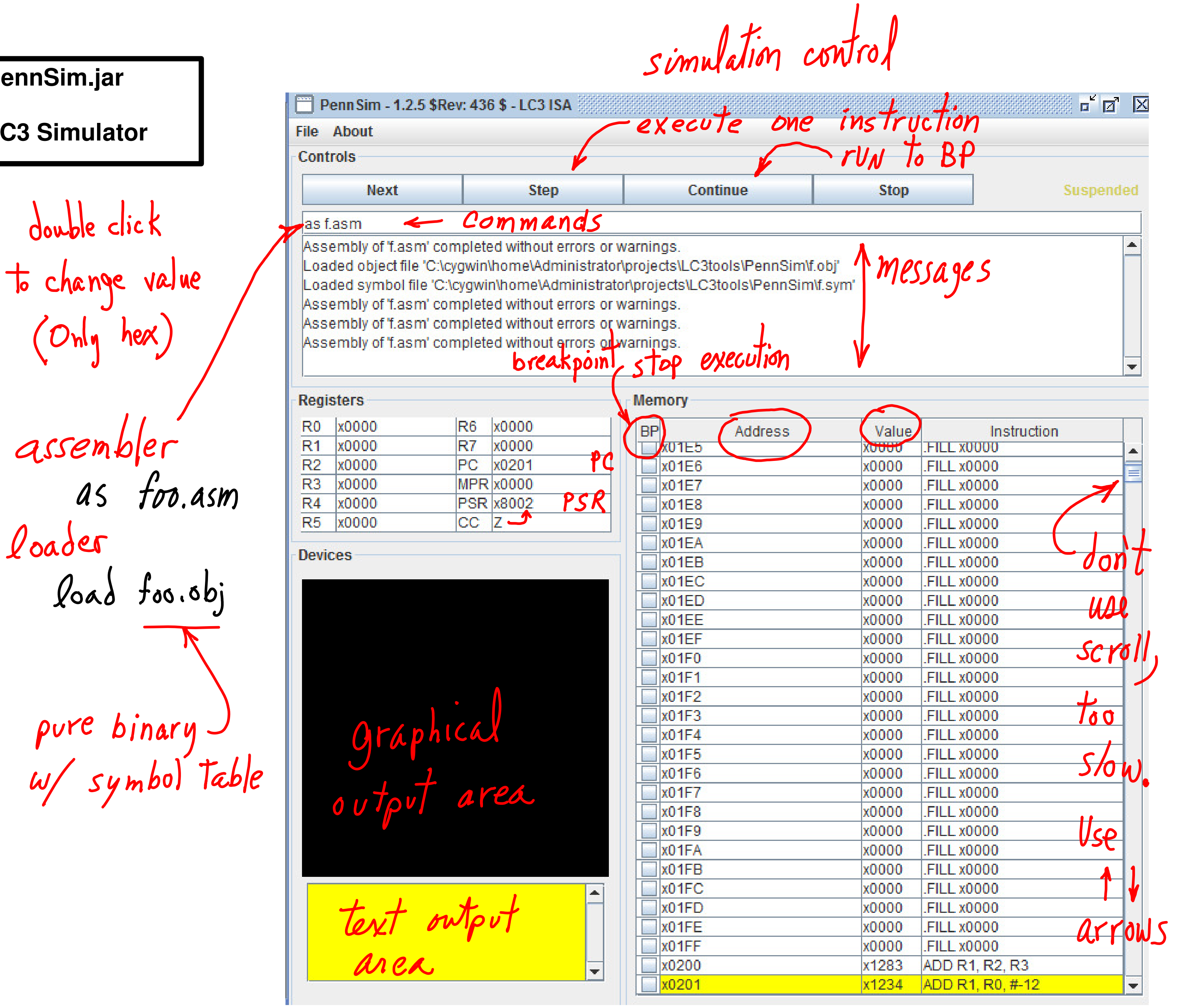
---

projects/LC3-tools/PennSim/PennSim-1-2-5.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.

PennSim.jar

LC3 Simulator

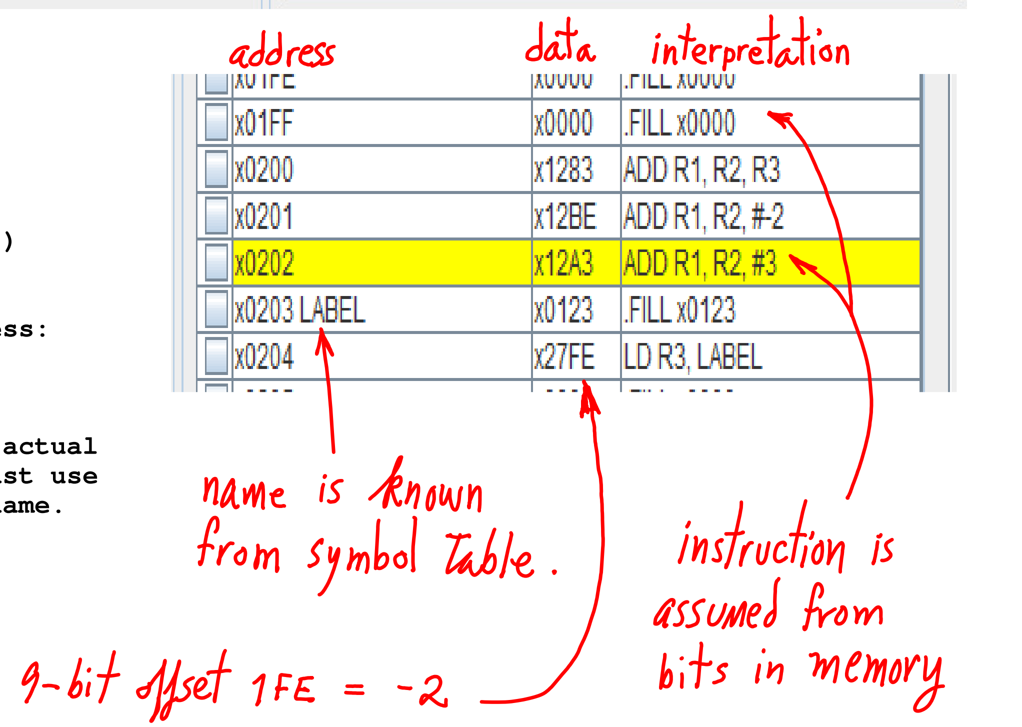



```
;---------------
;-- LC3 assembly
;---------------
.ORIG x0200
    add r1, r2, r3   ;-- comment
    add r1, r2, #-2  ;-- decimal
    add r1, r2, x3   ;-- hex (no "-")

                     ;-- NEW,
                     ;-- named address:
Label
.FILL x0123

    ld r3, Label     ;-- cannot use actual
                     ;-- offsets, must use
                     ;-- address's name.
.END
```

new



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**.

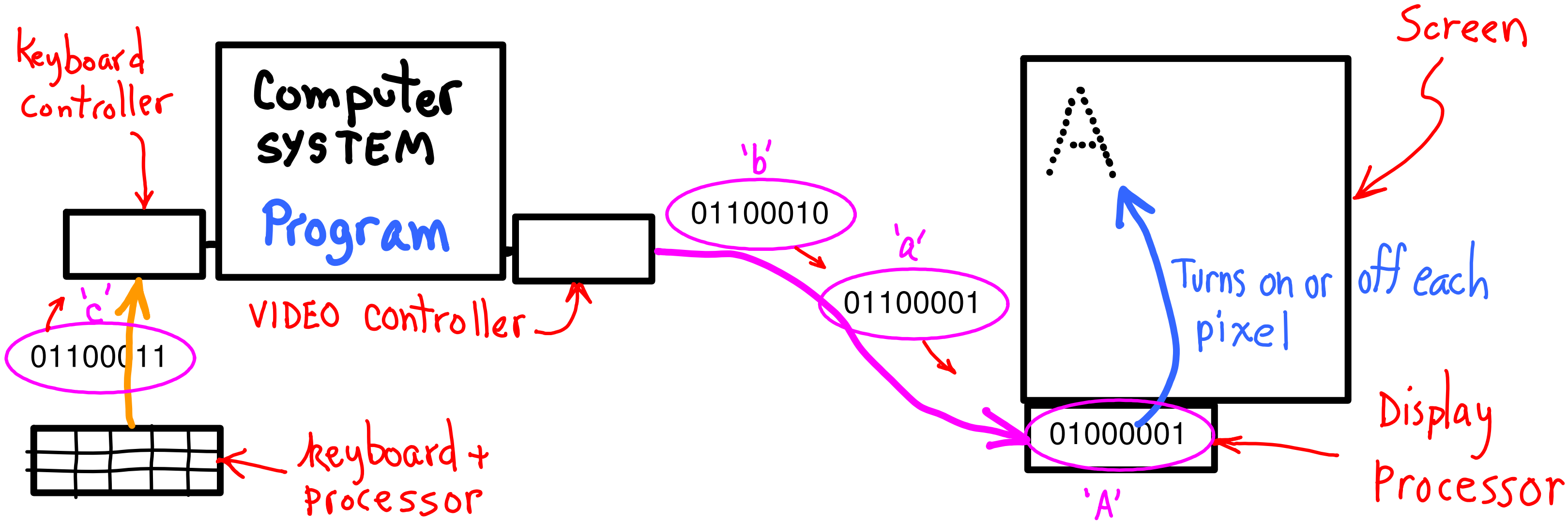
Some LC3 simulators (not ours) preloads a very primitive "OS". It is called "LC3os", and here it is (almost all of it):

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.

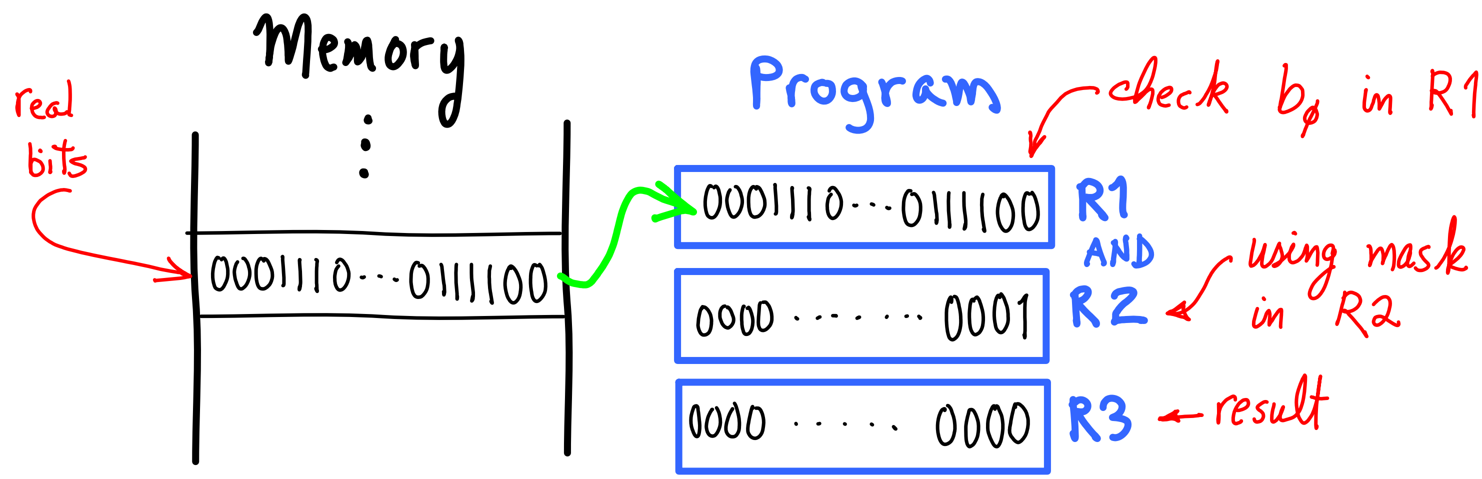


ASCII codes		ASCII codes	
...		...	
x30	**0**	x41	**A**
x31	**1**	x42	**B**
...		x43	**C**
x38	**8**	...	
x39	**9**	x5A	**Z**
...		...	

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

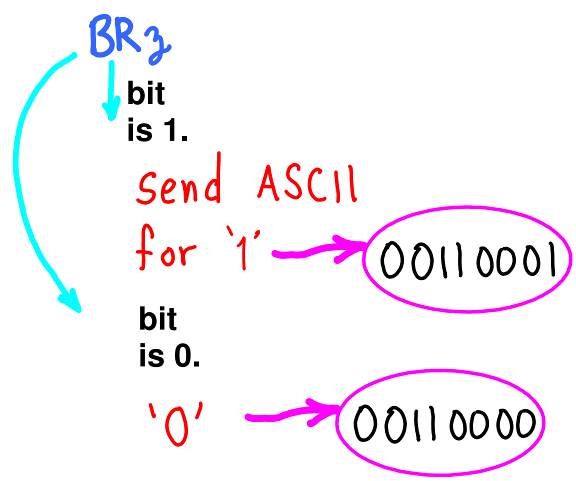
Video controller
--- **sends** to display device

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



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

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



DISPLAY

Programmer:

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

Possibilities (bits)

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

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

Typing

a b c d e f ... z

First typed -- First dispayed

Last typed -- Last dispayed

storage buffer: char ch [26]

a b c d e f g h i j k l m n o p q r s t u v w x y z

ch [0] ... ch [25]

printing

for i=0 to i=25 printf("%c", ch [i])

a b c d e f ... z

First char -- First dispayed

Last char -- Last dispayed

numbers

16-bit register: int n;

11001000

First typed -- First dispayed

Last typed -- Last dispayed

0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0

bit_15 bit_7 bit_0

printing

for i=0 to i=15

if (n is even) printf("0")

if (n is odd) printf("1")

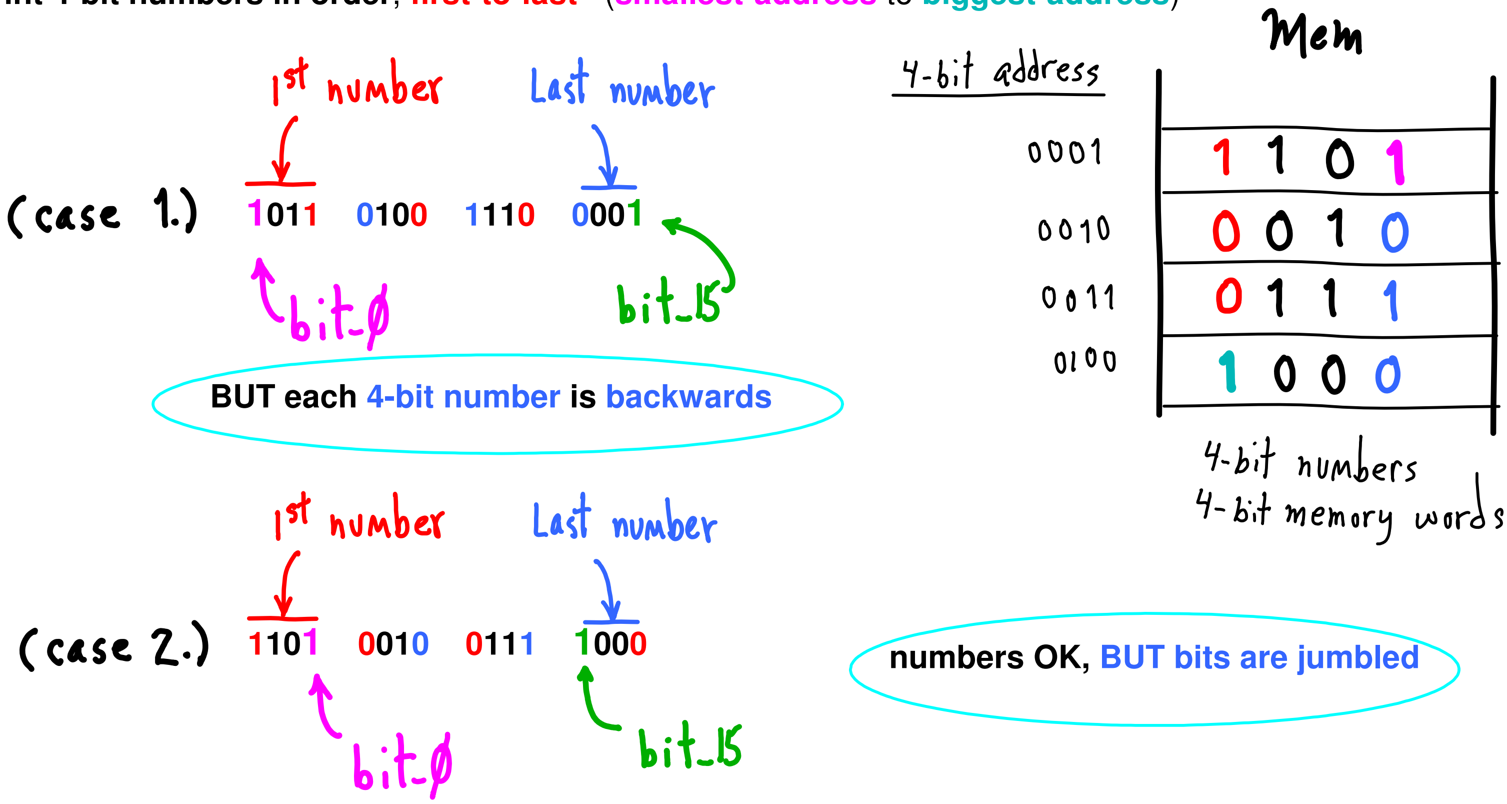
right-shift(n)

0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 0

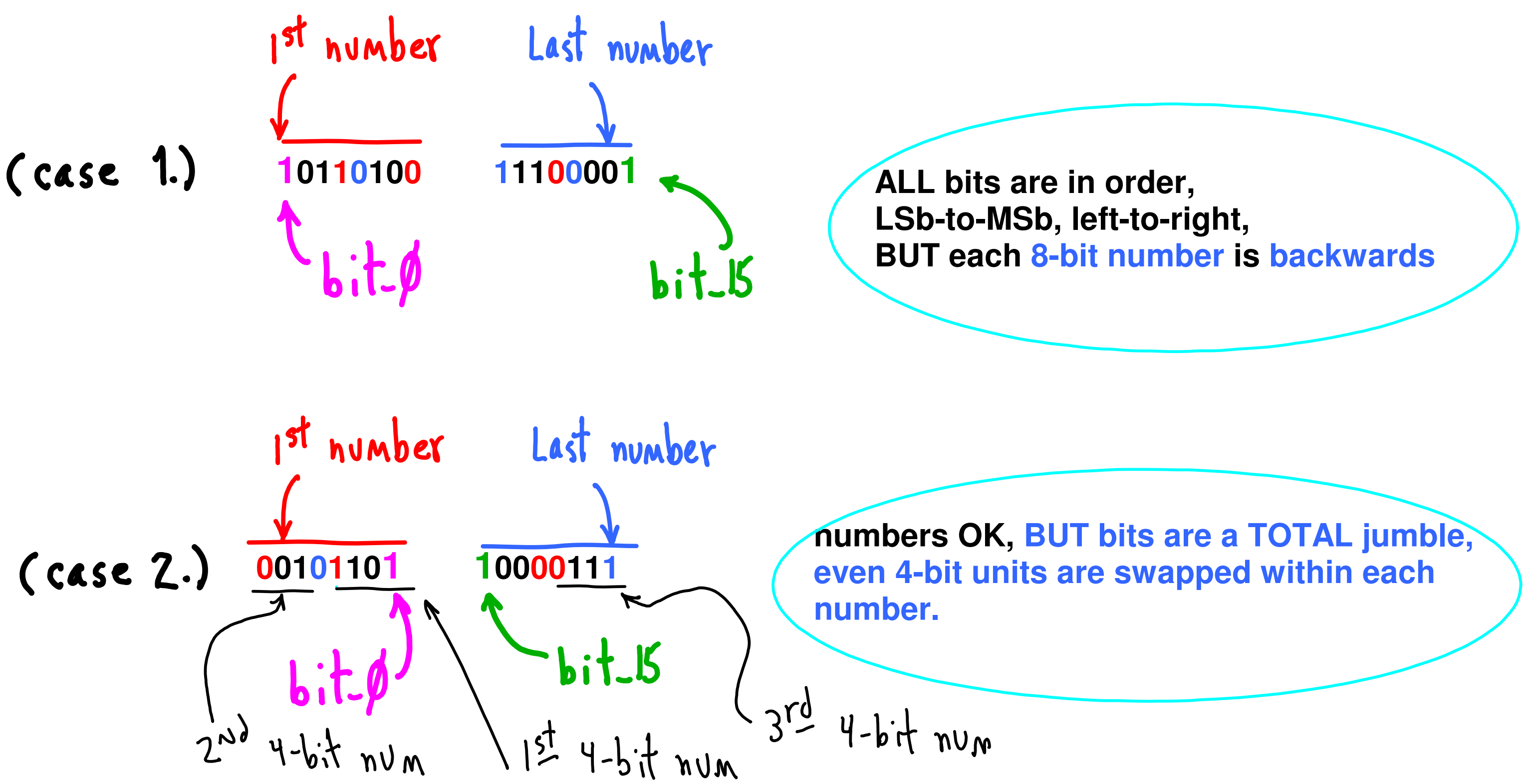
bit_0 bit_15

bits are backward?
LSb on left
MSb on right

Print 4-bit numbers in order, first-to-last (smallest-address to biggest-address)



Print 8-bit numbers in order, first-to-last (smallest-address to biggest-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.

FUN with BITS using unix's "od"

More on the difference between ASCII representation of bits and actual bits.
(For ASCII codes, ee, http://www.asciitable.com/ , or in Patt&Patel appendix.)

At a unix terminal window, enter,

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

"echo" first sends the byte for 'a' (01100001), then for 'b' (01100010), and so on. "od" reads them in that order, as bits, and gives a hex representatin of the bits it received. 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 (as bits, this would be: 01100001 01100010 01100011 0110100 00001010)

We would naturally think of this as the bytes of memory, smallest-address to biggest-address.
Now enter this,

```
echo "1234" | od -t x1
echo "1234" | od -t x2
echo "1234" | od -t x4
```

You will again see ASCII codes. The "real" bits for the first character, '1', are 00110001 (x31).
("-t x2" means, interpret two bytes as one object; "-t x4", four bytes per object.)
You will see this,

31 32 33 34 (as bits: 00110001 00110010 01100011 0110100

3231 3433 (as bits: 001100100110001 001110000110011

34333231 (as bits: 00110100001100110011001000110001

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, ...

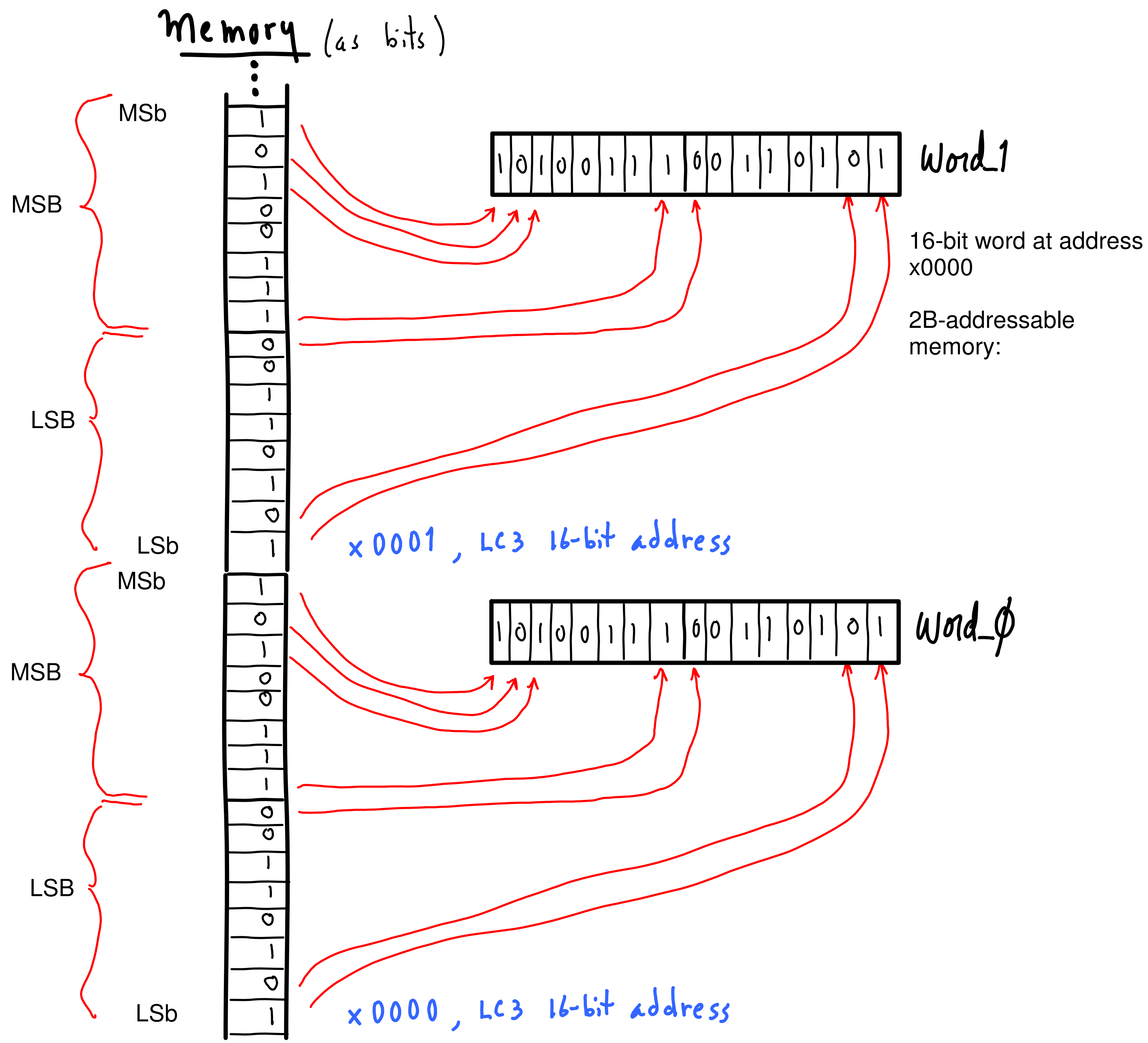
Let's look a memory layed out with the big end at the top and the small end at the bottom:

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

Further, consider memory layed out LSb to MSb, bit by bit, and word by word, starting at address x0000 and going **upward**.

That is, the bits are ordered starting at address x0000:

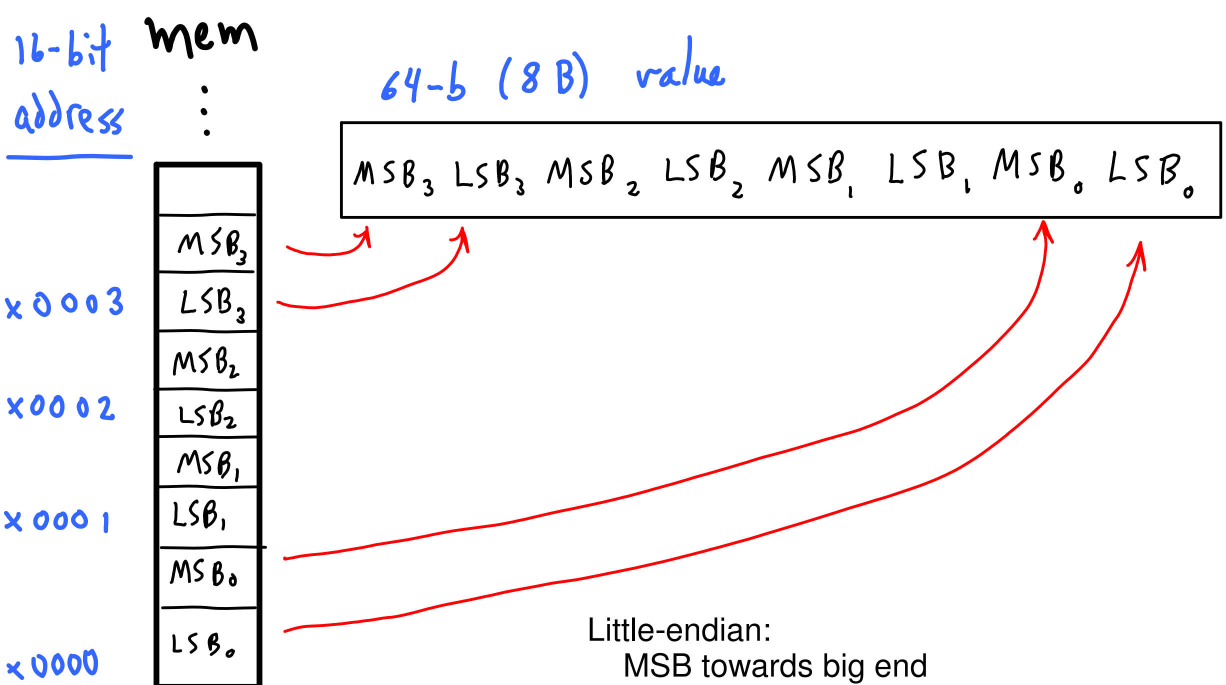
Mem[xFFFF][xF] == MSb of Word_xFFFF
Mem[xFFFF][x0] == LSb of Word_xFFFF
...
Mem[x0001][xF] == MSb of Word_x0001
Mem[x0001][x0] == LSb of Word_x0001
Mem[x0000][xF] == MSb of Word_x0000
Mem[x0000][x0] == LSb of Word_x0000



Suppose we extend LC3 addresses by one bit (17-bit addresses) to have a Byte-addressable memory:

17-bit address	8-bit content
11111111111111111	MSB of Word_xFFFF
11111111111111110	LSB of Word_xFFFF
...	...
00000000000000101	MSB of Word_x0002
00000000000000100	LSB of Word_x0002
00000000000000011	MSB of Word_x0001
00000000000000010	LSB of Word_x0001
00000000000000001	MSB of Word_x0000
00000000000000000	LSB of Word_x0000

How would we layout a 64-bit value?



bits are in order, top-to-bottom, MSb-to-LSb:

bit-63 is MSB3[7]
bit-62 is MSB3[6]
...
bit-2 is LSB0[2]
bit-1 is LSB0[1]
bit-0 is LSB0[0]

Little-endian:
- MSB towards big end
- LSB towards small end

Big-endian:
- LSB towards big end
- MSB towards small end

Big-endian reverse bytes as 64-b value:

MSB LSB
LSB0 MSB0 LSB1 MSB1 LSB2 MSB2 ... LSB3 MSB3

Is there an advantage to big-endian? Well, if you print bytes starting at LSB0, it will come out MSB-to-LSB, left-to-right, and look ok as a number.