ECE 2049 Lecture 3

Today

- More C examples,
- Memory layout, byte order

Administrivia

- HW1 – due today on Canvas
- Lab 0: Signoff due Thursday by end of office hours (6PM)
- HW2: Online after class, due next Tuesday
- Lab 1: Starts Thursday
- ED Discussion: Ask questions online here!

Office Hours

- Today, Thurs: 4-6PM

ECE2049: Homework 1

Material: Lecture 1
Due: Start of Lecture 3: Tuesday, 24 May 2022 by 2pm EDT

Submission notes:

- For full credit, please show your work and denote your answers with a circle or a box.
- Always write and draw your diagrams neatly! We cannot be expected to GUESS what you meant to write!
- Please see the submission guidelines on the homework page of the course website for details.

1. (5 pts) Please do the following logistical tasks to help you get started with the course:

a. Register for the course discussion board (EdStem) using your WPI email address: https://edstem.org/us/join/VBSzQc

b. Complete the course background survey to provide some information about prior courses you have taken. This will help me calibrate course content to accommodate everyone. You can find the survey here: https://wpi.qualtrics.com/jfe/form/SV_3ZV5f1gTa8swdOm

2. (5 pts) You are given three **16-bit** values shown below. Each of these values can be interpreted as:

- An unsigned number
- A sign-magnitude number
- A two's complement number

Provide the decimal (base 10) equivalent of each value for each of these interpretations. Show your work.

a. 0x4048 ⇒ 0x4048
b. 0x448C
c. 0xDEED

2. (5 pts) A hardware device is responsible for reading the state of 8 relays that control a manufacturing process. The device represents the state of each relay (R0—R7) in an 8-bit value v, with the state of relay R0 is stored in the least significant bit, and the state of R7 in the most significant bit.

If the device returns the value `v = 0x5B`, which relays are on?

HW1

Part 1b

0x 4048 => 0100 0000 0100 1000

- UNSIGNED $2^{14} + 2^6 + 2^3 = \boxed{16456}$

- SIGN-MAG: SIGN BIT IS 0 => +

$2^{14} + 2^6 + 2^3 = \boxed{16456}$

- 2's-COMP

$2^{14} + 2^6 + 2^3 = \boxed{16456}$

Part 1c

0xDEED => 1101 1110 1110 1101

- SIGN-MAG: NUMBER IS NEGATIVE
- 2's COMP: NUMBER IS NEGATIVE.

↳ 1101 1110 1110 1101

0010 0001 0001 0010 COMP

\+ 1

0010 0001 0001 0011

$-(2^{13} + 2^8 + 2^4 + 2^1 + 2^0) = \boxed{-8467}$

0x4048

- WRITE AS
  - UNSIGNED
  - SIGN-MAGNITUDE
  - 2's COMP

FIRST: WRITE 0x4048 AS BINARY

0x4048 => 0100 0000 0100 1000

UNSIGNED: $2^{14} + 2^6 + 2^3 = \boxed{16456}$

SIGN-MAGNITUDE: SAME AS UNSIGNED

=>

- MSB IS SIGN BIT = 0 = POSITIVE

$2^{14} + 2^6 + 2^3 = \boxed{16456}$

2's COMP

- POSITIVE, SO SAME.

0x DEED

UNSIGNED: ✓

SIGN-MAG: ✓

2's COMP: SIGN BIT IS 1, SO NEGATIVE.

DEED => 1101 1110 1110 1101 BIN

0010 0001 0001 0010 COMP

+ 1 ADD

0013 0001 0001 0013

$-(2^{13}+2^{8}+2^{4}+2^{1}+2^{0}) = -8467$

Module 3. Of Integers and Endians & Floating Point Representations

Topics

- Memory organization and endianness
- More data representations: overview of floating point

Last Time

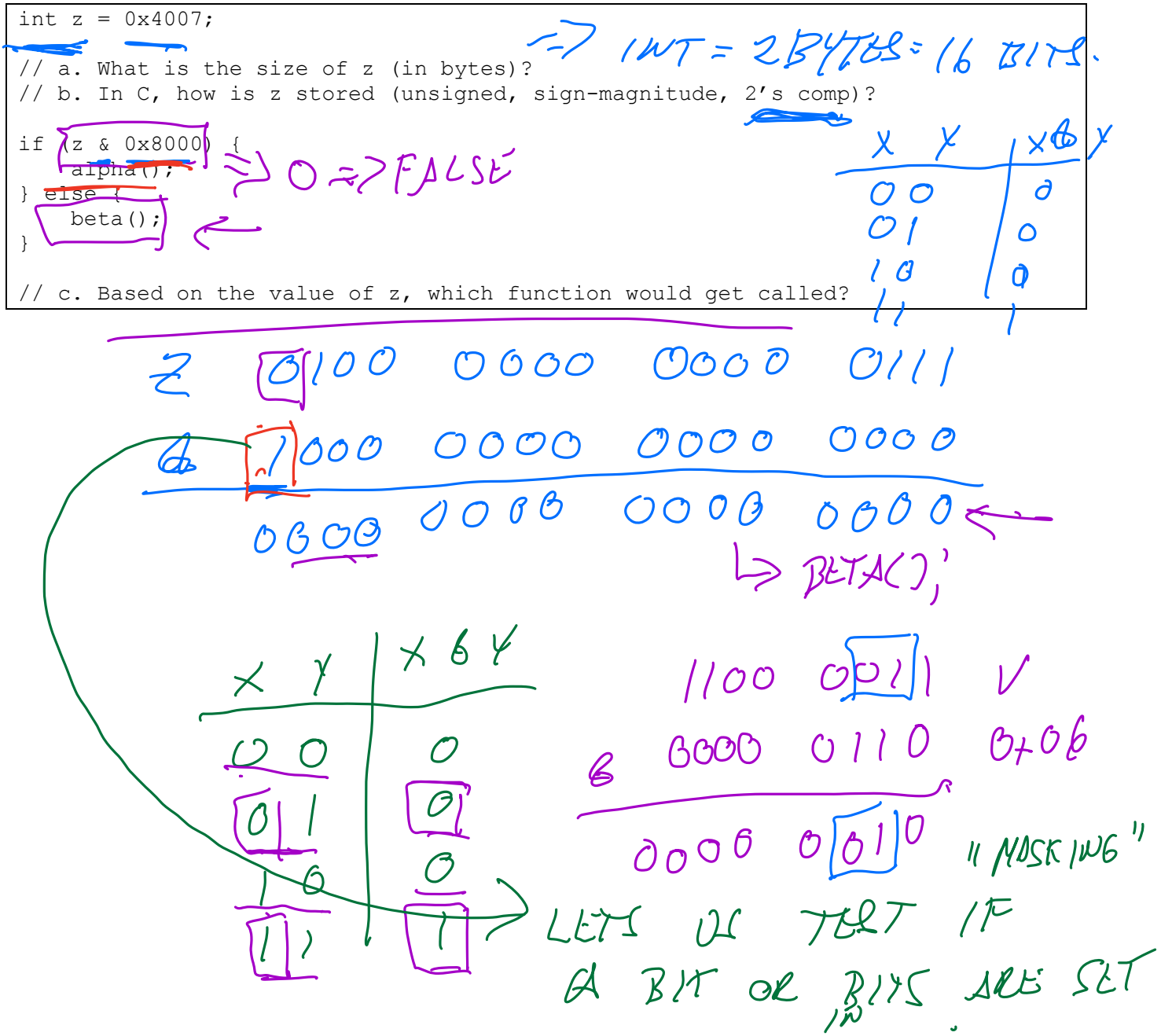
- C programming basics
- Data representations for characters

Warmup: try the following…

```
int z = 0x4007;
// a. What is the size of z (in bytes)?
// b. In C, how is z stored (unsigned, sign-magnitude, 2's comp)?

if (z & 0x8000) {
    alpha();
} else {
    beta();
}

// c. Based on the value of z, which function would get called?
```



IS POSITIVE ( )

– TEST IF A NUMBER x
IS POSITIVE W/O < >
≤ ≥

```
INT IS POSITIVE (INT X){
    // CHECK IF SIGN BIT IS SET,
       RETURN TRUE IF IT IS.

    RETURN !(X & 0x8000)

}
```

x 1 - - - - - - - - - - - - - - -
1000 0000 0000 0000
0x8000

Ex. -1 => 1111 1111 1111 1111
& 1000 0000 0000 0000
1000 0000 0000 0000
RETURN (0xFFFF & 0x8000) => !(TRUE)

-1 => FALSE
>0 => TRUE

2.

INT A = 42;

Memory organization

What does it mean to type "`int a`" in C? This is called variable declaration, which allocates space in the program's memory to store an int.

What do we mean by memory? You can think of memory as a big table of "addresses" that each map to a certain piece of data. This data could be a variable (as above), or it could be a piece of code, a portion of the hardware, etc., but for now let's focus on variables.

On the MSP430, addresses are 16-bits long, and each address refers to one byte.

Recall that the MSP430 is a 16-bit architecture,

ADDRESS	VALUES
0x0000	12h
0x0001	AAh
⋮	⋮
0xFFFE	0xFF
0xFFFF	0x10

$2^{16}-1$ → 16 BITS → 1 BYTE (8 BITS)

(2^{16} POSSIBLE ADDRESSES)(1 BYTE)
$= 65536$ BYTES
$= 16$ KiB
TOTAL MEMORY
(1 KiB = 1024 BYTES)

x86: 32 BIT ADDRESSES
$2^{32} = 4$ GiB

x86-64
2^{64} ≈ EiB

Unfortunately, this is no longer completely true! Newer MSP430 variants (like ours MSP430F5529) utilize 20-bit addresses. Why?

→ EXTEND ADDRESS SPACE W/ HARDWARE CHANGE.

$2^{20} = 1$ MiB
(MSP430X)

(WE WON'T DEAL W/ IT MUCH)

Laying out variables in memory

When you declare variables in your program, they are arranged in memory starting at a certain address. For now, it is sufficient to know that variables in `main` start at address 0x4400. We will discuss why in an upcoming lecture.

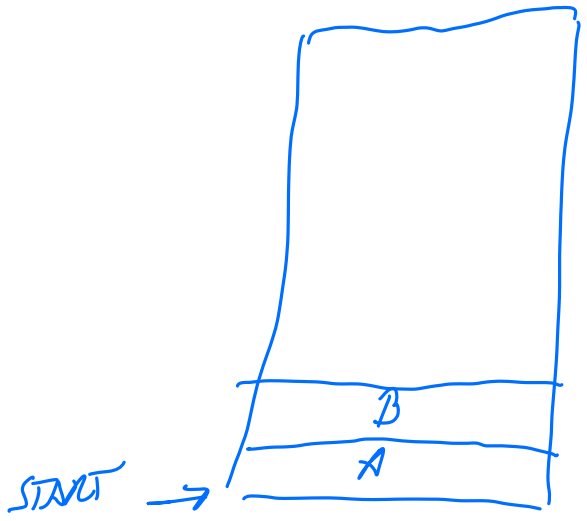
When variables are declared, they are (usually) arranged in order from this starting address. For example:

```
char a = 0x11;
char b = 0x22;
```

…can be arranged in memory as follows:

Address	Data	Variable
0x4401	22h	B
0x4400	11h	A

In our class, we will arrange memory in a table like the one above, with the starting address at the bottom. We use this convention because we are typically representing variables on the program *stack*, which starts at a fixed base address and grows up.



Endianness: Ordering bytes

In the previous example, we have left out an important detail. How do you store variables that are larger than a byte?

As declared on the MSP430, a long is has a size of four bytes:

```
long v = 0xAABBCCDD; // AAh is the  most significant byte (MSB), and
                     // DDh is the least significant byte (LSB)
```

For multi-byte variables, we have a choice–do we arrange the data with the least significant byte first, or with the most significant byte first? Which is correct? Does it matter?

This concept is known as *endianness*, which governs how a processor orders bytes in memory. There are two forms of endianness:

Little Endian (LE)

Little Endian stores the *least significant byte first*, meaning that the memory in this example would be arranged as follows:

Address	Data	Variable
0x4403	AA	v
0x4402	BB	
0x4401	CC	
0x4400	DDh	

Big Endian (BE)

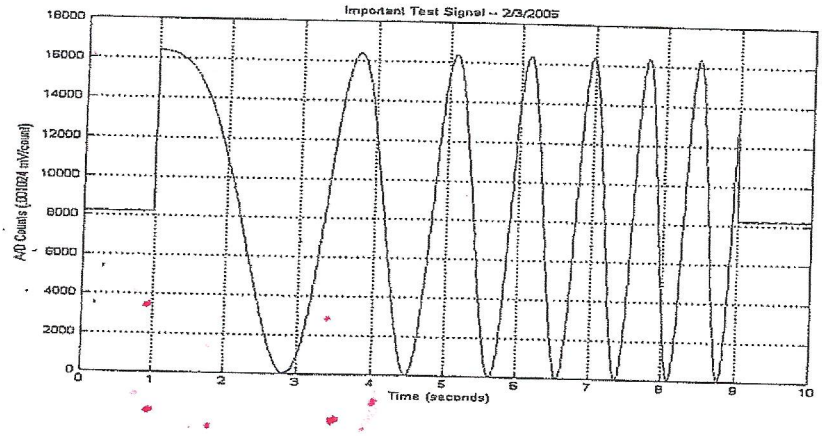
Big Endian stores the *most significant byte first*, as follows:

Address	Data	Variable
0x4403	DD	v
0x4402	CC	
0x4401	BB	
0x4400	AAh	

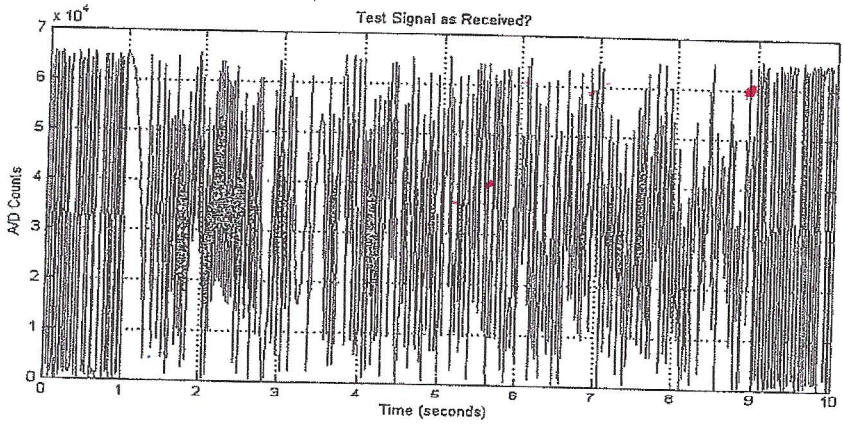
SA

Here is an example of being *Endian-ed*!

A nice plot of a file of unsigned integers as created on a little endian machine.



Below is a plot of the same data file having being read in as unsigned integers on a big endian machine. The data is good! It is the same as above! All that has changed is the endianness of the machine that read the data.



"GARBAGE"

The table below shows the first few unsigned integer values from the data file created on the little endian computer as read by both of the machines. The byte swap is evident in the hexadecimal values.

Read as Little Endian		*Read as Big Endian*	
8178	1FF2 h	61983	F21F h
8193	2001 h	288	0120 h
8194	2002 h	544	0220 h
8182	1FF6 h	63007	F61F h
8201	2009 h	2336	0920 h
8201	2009 h	2336	0920 h

ENDIAN-NESS is Part of Microprocessor Design!
Not a function of the OS!!

Important points on endianness

- Endianness is a fundamental part of the architecture's design. When a processor is designed, it is designed to use a specific byte order–you cannot change this with a compiler setting.
- Is one endianness better than the other? No, they simply reflect different design choices.
- Big endian is read "left to right", which is intuitively easier to read for those accustomed to languages written left to right
- Little endian makes it easier to slice out small portions of a variable (eg, what if you only want the first byte of a long?)

When will you deal with endianness?

Endianness becomes especially important when you need to transfer data between different architectures. Examples include any stored data format or network protocol.

LE: MSP430, x86, APPLE M1

BE: PowerPC, DSP CHIPS, "NETWORK BYTE ORDER" (INTERNET TRAFFIC)

ENDIANNESS MATTERS FOR ALL METHODS BY WHICH DATA IS TRANSFERRED BETWEEN SYSTEMS

(FILE FORMATS, NETWORK...)

More memory layout: Arrays

How do arrays work, anyway?

In C, we can declare arrays and use them as follows:

```
// Declare an array of 5 bytes
char arr[5];
// Declare an array of 5 bytes, and initialize it (set it with some initial values)
char arr[5] = { 0xAA, 0xBB, 0xCC, 0xDD, 0xEE };

// You can access elements of an array by "indexing" into it
// In C, array indexes start at 0
char c = arr[0]; // The first element
char d = arr[4]; // The last element (arr has size of 5, so last index is 5 - 1 = 4
```

You can think of the elements of the array laid out like this:

Index	0	1	2	3	4
Element	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]
Value	AAh	BB	CC	DD	EE
	0x4400	0x4401	0x4402	0x4403	0x4404

Why is it important that array elements are contiguous? (And must contain elements of the same type?)

WANT TO MAKE IT "EASY" TO FIND THE i'TH ELEMENT

$A_i = A_0 + (i * \text{SIZEOF}(A))$

"EASY" MEANS CONSTANT TIME (NO SEARCHING) O(1)

BASE ADDRESS

SIZE OF ONE ELEMENT (INT, CHAR, ETC)

What would happen if we tried to get the 6th element of `arr`?

ARR[1000] = ?

↳ THIS WILL COMPILE, BUT PROGRAM WILL READ WHATEVER DATA IS IN THIS SPACE, BUT IT'S NOT PART OF ARR (NOT WHAT YOU WANT)

NO ERROR CHECKING!

BUFFER OVERFLOW

CLASSIC SECURITY PROBLEM.

How endianness affects arrays (or rather, how it does not)

A fundamental property of arrays is that their elements are stored contiguously in memory in order of their index (as discussed above).
Endianness does not change the order of array elements.

For example, if we laid out the array from the above example on a Big Endian (BE) and a Little Endian (LE) system, it would look like this:

Address	BE	LE
0x4404	EE	EE
0x4403	DD	DD
0x4402	CC	CC
0x4401	BB	BB
0x4400	AA	AA

ARR[4]

ARR[0]

However, endianness *does* affect the ordering of the bytes in each element of the array! In the previous example, the elements were just 1 byte each!

Example: an array of `int`s:

ELEMENTS ARE TYPE INT

```
int iarr[2] = {0x1122, 0x3344};
```

EACH ELEMENT IS 2 BYTES

Here, the memory would be organized as follows:

Address	BE	LE
0x4403	44	33
0x4402	33	44
0x4401	22	11
0x4400	11	22

IARR[1] = 0x4402

IARR[0] = 0x4400

IARR[1]

IARR[0]

DOES CHANGE

DOES NOT CHANGE W/ DIFF. BYTE ORDER

Using addresses as data

We can also have variables that contain memory addresses. These are called *pointers*.

You work w/ pointers all the time!

You can get the address of a variable with the *"address-of" operator* (&):

```
long v = 0x11223344;
long *pv = &v;
```

In this example, we say that `pv` is declared as the type "pointer to long," which is indicated by the "*" before the name `pv`.

How big is `pv`?

, pv holds one memory address

on MSP430 ⇒ 2 bytes (16 bits)

What is the value of `pv`?

— starting address of v

pv = &v = 0x4400

We can lay out these variables in memory as follows:

Address	BE	LE
0x4405	00	44
0x4404	44	00
0x4403	44	11
0x4402	33	22
0x4401	22	33
0x4400	11	44

pv

v

pv contains v's address not its contents.

How big is a pointer?

A pointer is the size of a memory address for a given architecture. On the MSP430, an address has a size of 2 bytes (16 bits).

Type	Size (bytes)
`int`	2
`long`	4
`char`	1
`long long`	8

Type	Size (bytes)
`int *`	2
`long *`	2
`char *`	2
`long long *`	2

ALL HAVE THE SAME SIZE! (ON SAME SYSTEM)

This is one way in which pointers are powerful: a pointer can represent a larger data structure in the program—by passing around the pointer, we can avoid copying or moving the larger data structure.

How are pointers used with arrays?

Whenever you use arrays, you use pointers. Consider the following example:

```
int iarr[10];
int i = iarr[5];
```

When you index into an array, the program actually does the following:

```
int i = *(iarr + 5); // Equivalent to writing iarr[5]
```

Here, the * is the *dereference operator*, which **gets the value at the given address**. This is called *dereferencing* the pointer—it is the opposite of the *address-of* (&) operator.

GET THE DATA AT THIS MEMORY ADDRESS

IARR = STARTING ADDRESS OF IARR = &IARR[0]

Working with Pointers

Pointer math: When performing arithmetic operations on pointers, the address changes in increments based on the type of the pointer.

$A_i = A_0 + i \cdot \text{SIZEOF}(A)$

```
// Example 1: array of char
char carr[4];
// How big is the array?
// Say the starting address is 0x4400, what is the address of carr[3]?
```

(4 ELEMENTS)(1 BYTE/ELEMENT) = 4 BYTES

0x4400 + 3 * SIZEOF(CHAR)

= 0x4403

```
// Example 1: array of int
int carr[4];
// How big is the array?
// Say the starting address is 0x4400, what is the address of iarr[3]?
```

(4 ELEMENTS)(2 BYTES/INT) = 8 BYTES

0x4400 + 3 * SIZEOF(INT)

= 0x4400 + 6 = 0x4406

Passing arrays: Further, when you use the name of an array (either to store or pass to a function), you are *passing a pointer to the first element of the array*. This is the "starting point" of the array used as input to calculate the index.

```
int *ptr = iarr;     // Could also write &iarr[0]
do_thing(iarr, 10); // Same here

void do_thing(int* arr, int size) { // Function takes pointer to array (+ size)
    // . . .
}
```

ARR[0] ⇒ SAME AS IARR[0]

HOW DO YOU KNOW THE SIZE OF AN ARRAY IN C?

NO WAY TO TELL FROM JUST THE POINTER.

UP TO THE PROGRAMMER TO HAVE A CONVENTION

EG:
- ARGUMENT
- NULL TERMINATED STRINGS

Memory organization example

Here's a larger example of memory organization. How would we organize the following variables?

```
unsigned int a = 0x00FF;
long int b[2] = { 65540, -5 };
char c = 'c'; // 'c' = 0x43
```

→ 0x4400 IS START

- VARIABLES ALLOCATED IN ORDER.

STEP 1. WRITE EVERYTHING IN HEX.

B, EACH ELEMENT IS 4 BYTES (TYPE IS LONG INT)

B[0] = 65540

= 65536 + 4 = $2^{16} + 2^2$

31 | 15 | 7 | 2 1 0

0000 0000 | 0001 0000 | 0000 0000 | 0000 0100

00 01 00 04

B[0] = 00 01 00 04h

B[1] = -5 (NEGATIVE SO DO 2'S COMP)

0... 0000 0000 0000 0101 MAGNITUDE

1... 1111 1111 1111 1010 COMP

31 | 15 | 7 | 1 | 0

1111 1111 1111 1111 1111 1011

FF FF F F F B

B[1] = FFFF FFFBh

How many bytes of memory do we require?

A: 2 BYTES

B: (2 ELEMENTS)(4 BYTES/ELEMENT) = 8 BYTES

C: 1 BYTE = 11 BYTES TOTAL.

So using the above information, we can make our table:

Address	BE	LE
0x440C		
0x440B		
0x440A	43	43
0x4409	FB	FF
0x4408	FF	FF
0x4407	FF	FF
0x4406	FF	FB
0x4405	04	00
0x4404	00	01
0x4403	01	00
0x4402	00	04
0x4401	0F	00
0x4400	00	0F

A = 0x000F

B[0] = 00 01 00 04 h

B[1] = FF FF FF FB h

C = 43h

[A | B[0] | B[1] | C

C

B[1]

B[0]

A

ORDER OF VARIABLES ALWAYS THE SAME, BUT ORDER OF BYTES WITHIN A VARIABLE CHANGES W/ ENDIANNESS.

CHAR *STR = "HELLO";

STRINGS IN C:

CHAR *S = "HELLO"

	ASCII	HEX	
4405	'\0'	00h	S[5]
4404	'O'	4F	S[4]
4403	'L'	4C	S[3]
4402	'L'	4C	S[2]
4401	'E'	45h	S[1]
0x4400	'H'	48h	S[0]

INT STRLENGTH(CHAR *S) {

}