NASM Basics II
Data declarations

ICS312
Machine-Level and Systems Programming

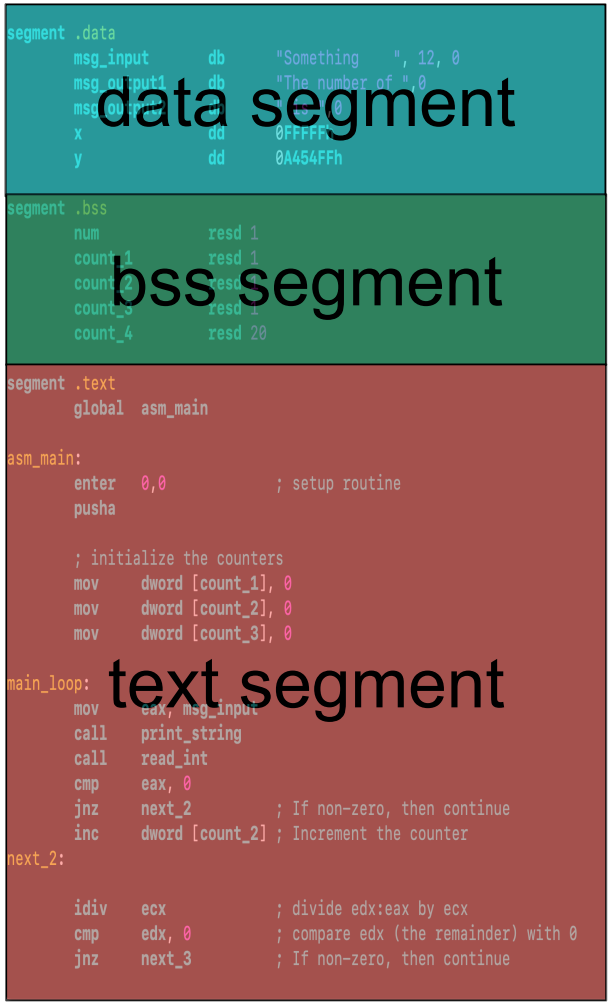
Henri Casanova (henric@hawaii.edu)

NASM Program Structure

```
segment .data
        msg_input       db      "Something    ", 12, 0
        msg_output1     db      "The number of ",0
        msg_output2     db      " is ",0
        x               dd      0FFFFFh
        y               dd      0A454FFh

segment .bss
        num             resd 1
        count_1         resd 1
        count_2         resd 1
        count_3         resd 1
        count_4         resd 20

segment .text
        global  asm_main

asm_main:
        enter   0,0             ; setup routine
        pusha

        ; initialize the counters
        mov     dword [count_1], 0
        mov     dword [count_2], 0
        mov     dword [count_3], 0

main_loop:
        mov     eax, msg_input
        call    print_string
        call    read_int
        cmp     eax, 0
        jnz     next_2          ; If non-zero, then continue
        inc     dword [count_2] ; Increment the counter
next_2:

        idiv    ecx             ; divide edx:eax by ecx
        cmp     edx, 0          ; compare edx (the remainder) with 0
        jnz     next_3          ; If non-zero, then continue
```

- This is a typical program written in assembly
- Before we know what any of this means (if you can even see it), we need to learn the overall structure
- There are 3 main sections or “segments”…

NASM Program Structure



declaration of initialized data

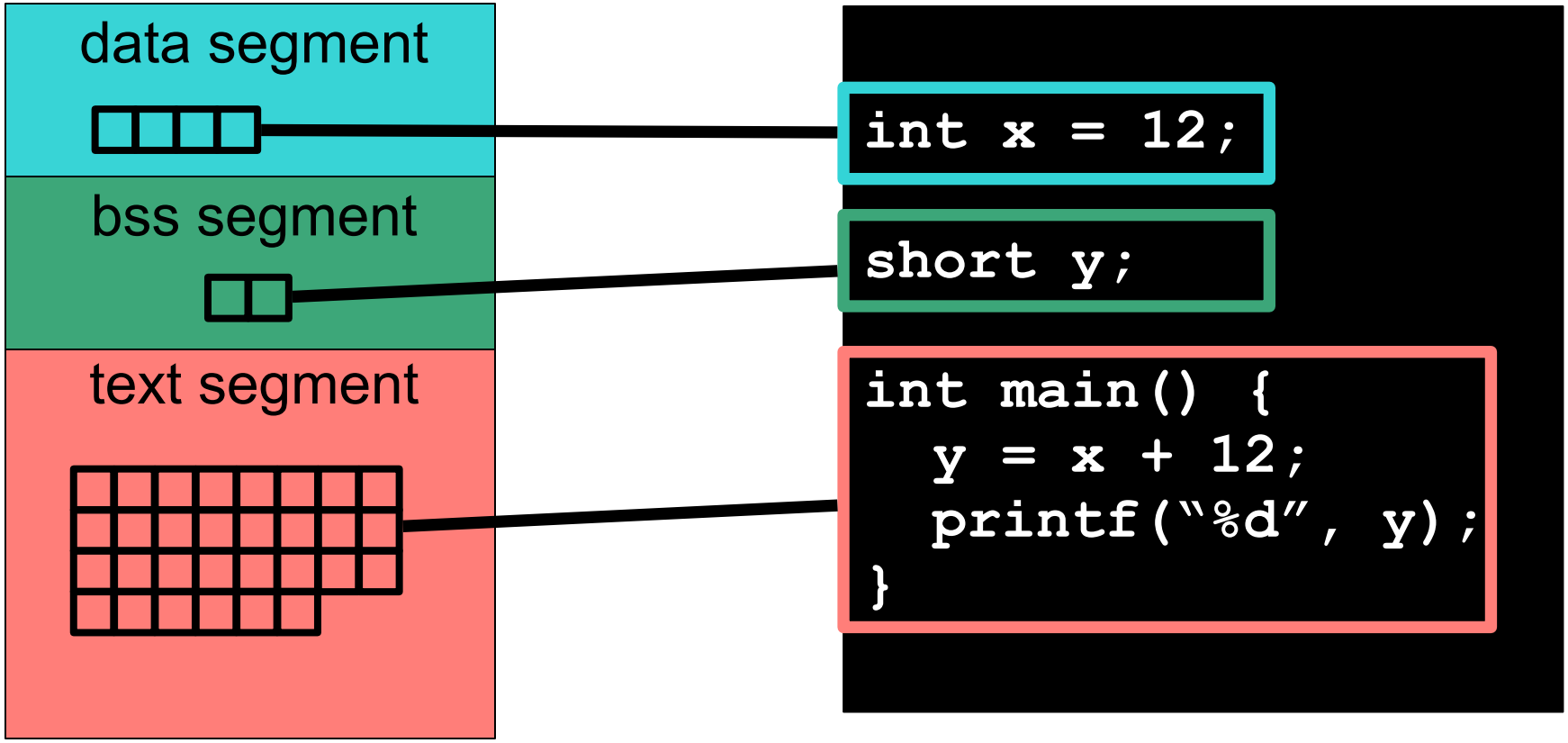
declaration of uninitialized data

statically allocated data that is allocated for the duration of program execution

code

- We'll see later that in source code these sections don't have to be in this order and can be split up and interleaved
- But for now, let's assume that all programs follow this exact structure

Relation to high-level C code



Disclaimer

- We are about to see some details about the NASM syntax to declare global data (note I am not saying “variables”)
- We need to go through this because sometimes in live coding I’ll use this syntax
  - And if you do write assembly code for the sample programming assignments, of course you’ll need this content
- BUT, you’ll only be quizzed/tested on high-level code declarations
  - So you don’t need to memorize the assembly syntax for data declaration
  - If it’s on a quiz/exam, there will be a description of what the syntax means
- We are going to show everything from a high-level code perspective, using C

Declaring Integers

- In C you can declare an integer global variable like:

```
// <data type>  <variable name>
unsigned int var;
```

- As humans we think “var is an unsigned integer”:
  - We can write code to use/modify its value, and the language/compiler/interpreter can limit what’s allowable
  - Some languages are really permissive, some really restrictive:

```
// OK in C, not OK in Java
var = 'a';
```

- But the CPU has no notion of “data type” or “variable name”…

High-Level vs. Low-level

```
unsigned int var;
```

- From a low-level perspective the declaration above **only** means: “I need 4 bytes somewhere in RAM”
- We “lose” the notion of data type
- The code (i.e., the instructions) will use these 4 bytes with instructions that assume some particular data type
- But that’s all dealt with by the compiler, not by the CPU
- Let’s see how the above declaration is done in assembly….

Declaring Uninitialized Integers

```
<name>    res<#bytes> <#repeats>
```

- The name is a symbolic name for the **ADDRESS** of where the first of the requested bytes will be
- The number of bytes is **d for 4 bytes**, **w for 2 bytes**, **b for 1 byte**
- The number of repeats is the number of a values for which space is needed

`unsigned int var;`	`var resd 1`
`int var;`	`var resd 1`
`unsigned short var;`	`var resw 1`
`short var;`	`var resw 1`
`unsigned char var;`	`var resb 1`
`char var;`	`var resb 1`

Declaring Uninitialized Integers

```
<name>    res<#bytes> <#repeats>
```

- The name is a symboli[illegible] the first of the requested bytes will be
- The number of bytes is [illegible] 1 byte
- The number of repeats [illegible] ch space is needed

Note that we lose all notion of data type!

`unsigned int var;`	`var resd 1`
`int var;`	`var resd 1`
`unsigned short var;`	`var resw 1`
`short var;`	`var resw 1`
`unsigned char var;`	`var resb 1`
`char var;`	`var resb 1`

Example: Individual values

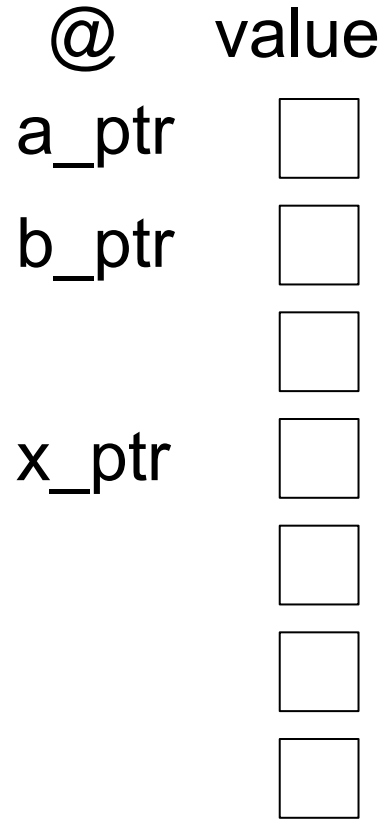
- C:

```
char a;
unsigned short b;
int x;
```

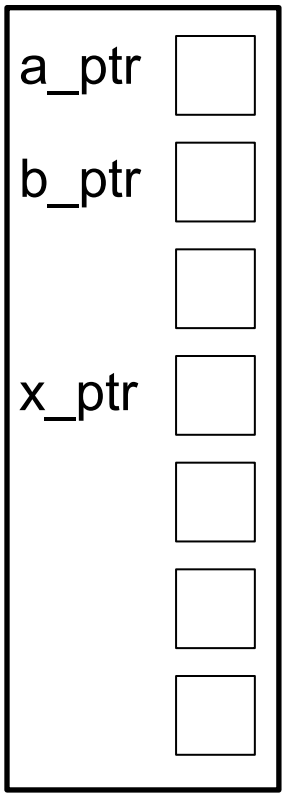
- NASM:

```
; I am using "_ptr" to
; make it crystal clear
; names are addresses,
; not values
a_ptr resb 1
b_ptr resw 1
x_ptr resd 1
```

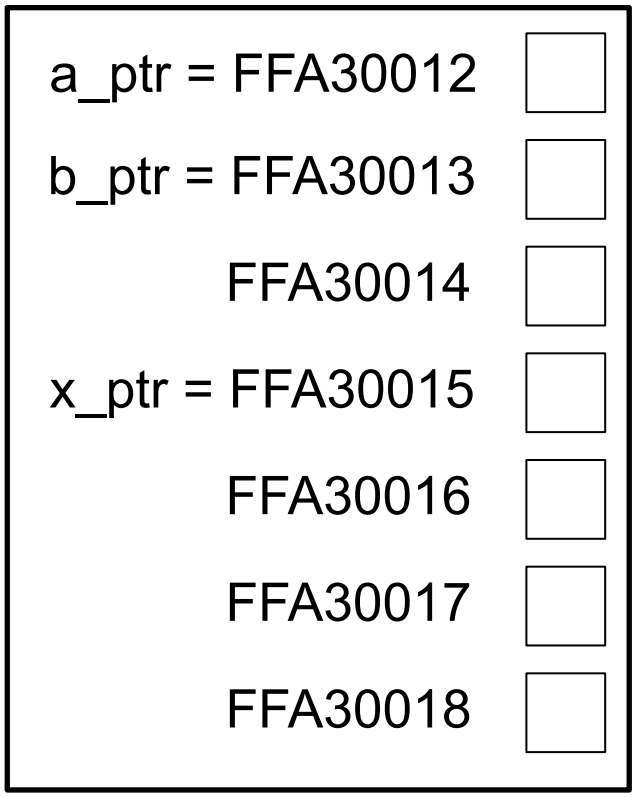
- 7 bytes in RAM:



Example: Individual values



- Our assembly program can use these symbolic addresses in instructions
  - Just like a C program will use symbolic variable names (for values)
- Once the program runs, these symbols correspond to actual addresses
  - See example on the right



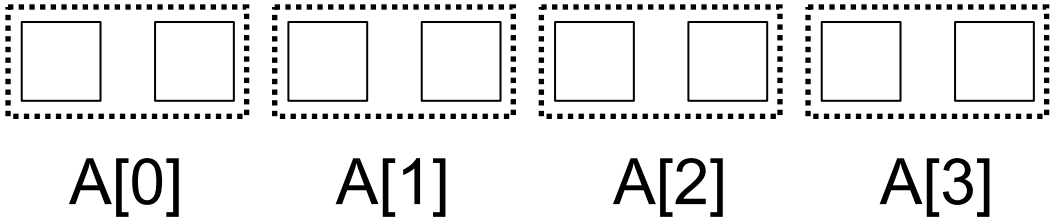
“compile time”

“runtime” example

Example: Array

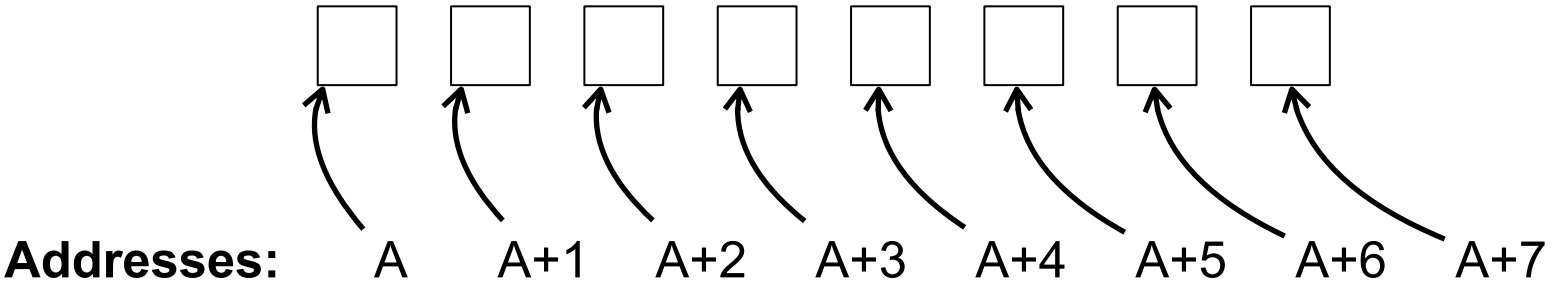
- An array is **just** a bunch of bytes that are contiguous in RAM
- C:

```
signed short A[4] // 4 * 2 bytes in RAM
```



- NASM:

```
A resw 4 ; 4 * 2 bytes in RAM
```



Declaring Initialized Integers

```
<name>   d<#bytes> <value>, <value>, …
```

- The name is a symbolic name for the **ADDRESS** of where the first of the requested bytes will be
- The number of bytes is **d for 4 bytes**, **w for 2 bytes**, **b for 1 byte**
- The values can be in decimal, binary (b), octal (o), or hex (h)
  - No need to have leading 0's in the declaration (they will be added)
  - But hex has a weird "phantom" 0 for parsing purposes

```
short var = 12;
short var = 0b1100; // base 2
short var = 012;    // base 8
short var = 0xC;    // base 16
```

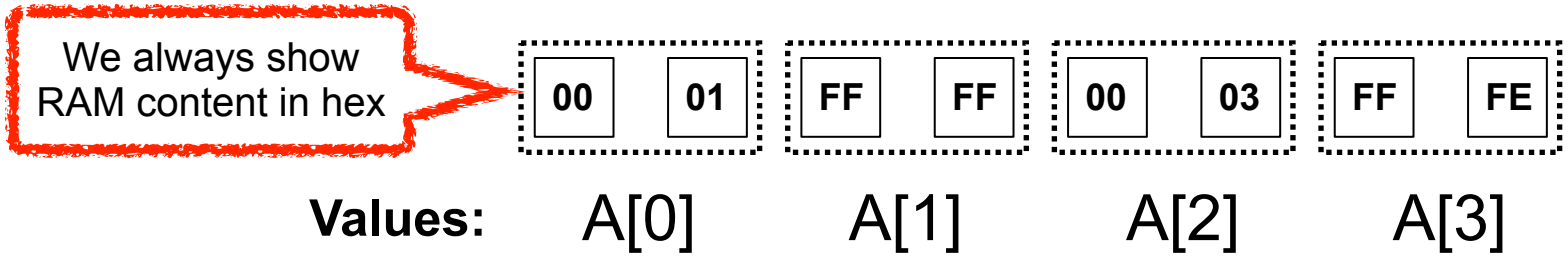
```
var dw 12
var dw 1100b
var dw 14o
var dw 0Ch
```

This leading 0 is not really part of the number

Initialized Array of Integers

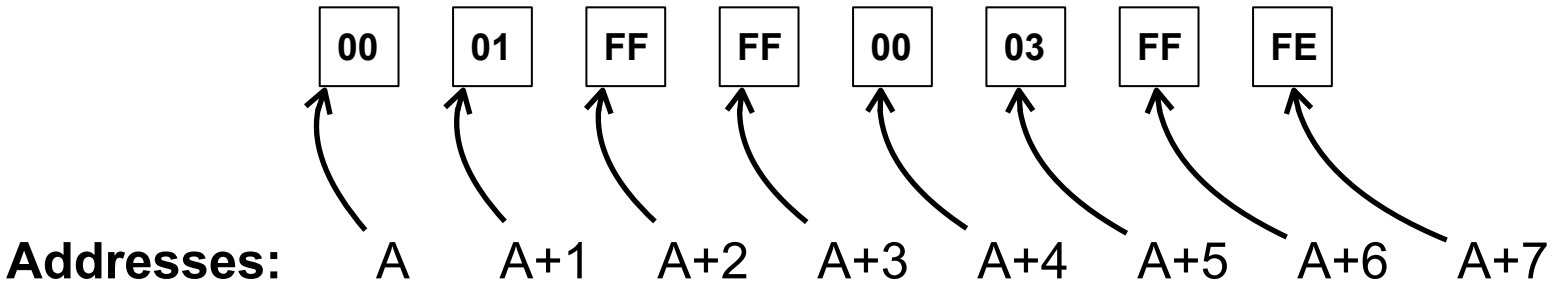
- C:

```
signed short A[4] = {1, -1, 3, -2}
```



- NASM:

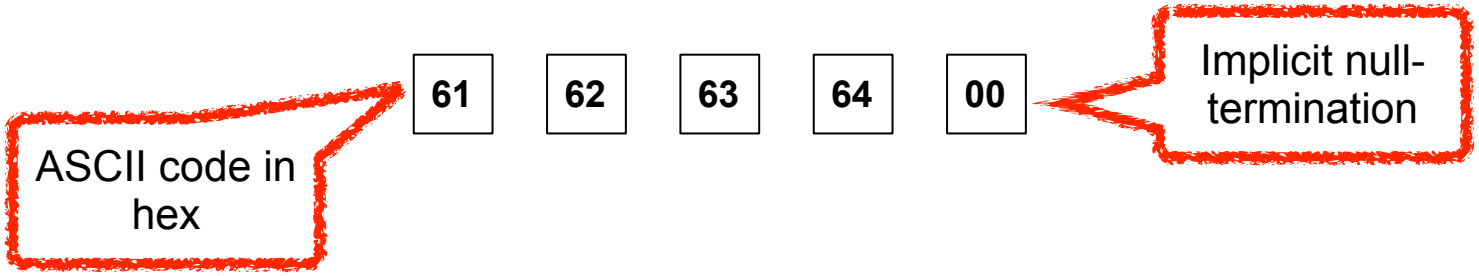
```
A dw 1, 0FFFFh, 11b, -2
```



Character strings

- A string is **just** an array of ASCII codes
- C:

```
char *str = "abcd";
```



- NASM:

```
str db "a", "b", "c", "d", 0

str db 061h, 062h, "c", 'd', 0

str db "abcd", 0
```

explicit null-termination

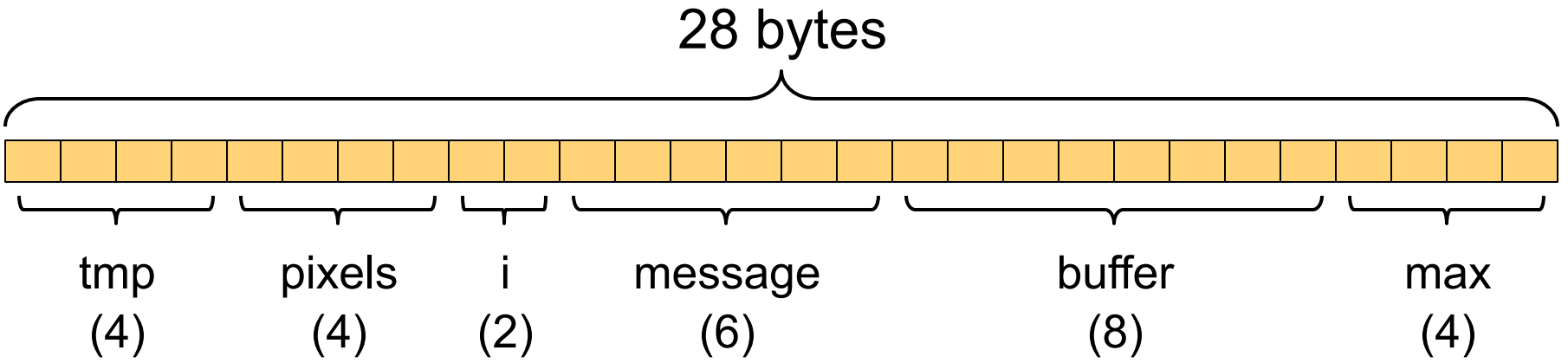
Data/Bss segment example

```
int tmp = -1;
unsigned char pixels[4] = {0xFF, 0xFE, 0xFD, 0xFC};
signed short i = 0;
char *message = "Hello";
unsigned char buffer[6];
unsigned int max = 254;
```

- What's the corresponding memory content?

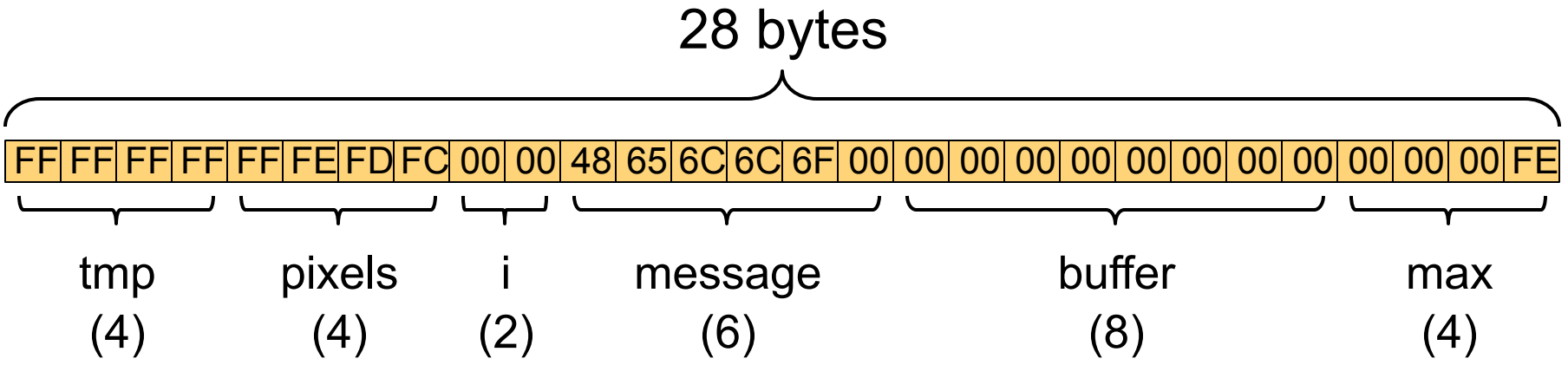
Data/Bss segment example

```
int tmp = -1;
unsigned char pixels[4] = {0xFF, 0xFE, 0xFD, 0xFC};
signed short i = 0;
char *message = "Hello";
unsigned char buffer[6];
unsigned int max = 254;
```



Data/Bss segment example

```
int tmp = -1;
unsigned char pixels[4] = {0xFF, 0xFE, 0xFD, 0xFC};
signed short i = 0;
char *message = "Hello";
unsigned char buffer[6];
unsigned int max = 254;
```



In NASM

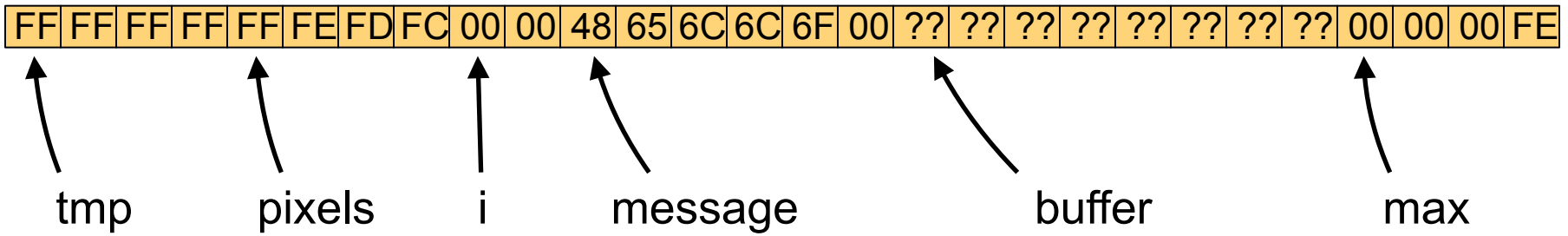
```
int tmp = -1;
unsigned char pixels[4] = {0xFF, 0xFE, 0xFD, 0xFC};
signed short i = 0;
char *message = "Hello";
unsigned char buffer[6];
unsigned int max = 254;
```

- Could be as follows in NASM (there are many options):

```
tmp         dd    -1
pixels      db    0FFh, 0FEh, 0FDh, 0FCh
i           dw    0
message     db    "H", "e", "llo", 0
buffer      resb  8
max         dd    254
```

Data segment example

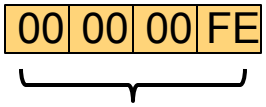
```
tmp         dd    -1
pixels      db    0FFh, 0FEh, 0FDh, 0FCh
i           dw    0
message     db    "H", "e", "llo", 0
buffer      times  8    db  0
max         dd    254
```



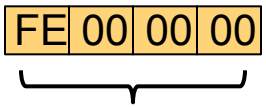
- In NASM, the names are now addresses, meaning they are 32-bit integers
- In high-level lingo we call them **pointers!**

Endianness?

```
int  max =  254;
```

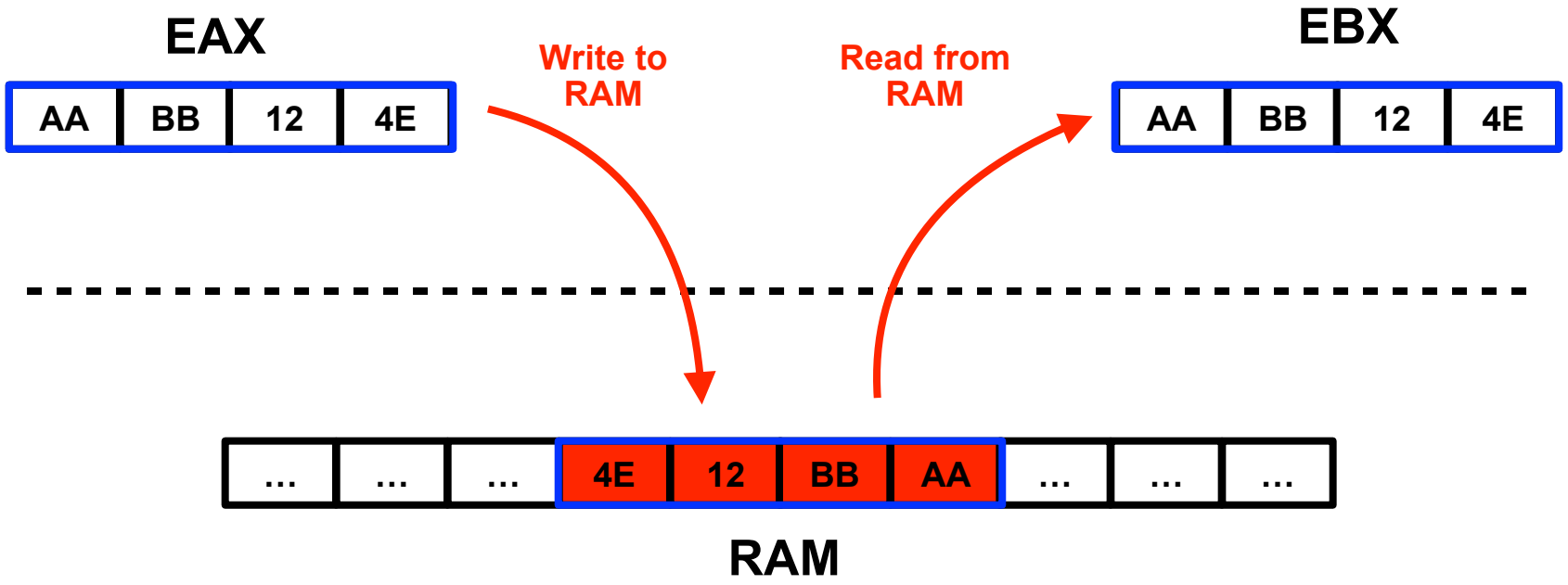


- In the previous example I showed the above 4-byte memory content for a double-word that contains 254_{10} = 000000FEh
- While this seems to make sense, it turns out that Intel processors do not do this!
  - Yes, the last 4 bytes shown in the previous slide are wrong
- The scheme shown above (i.e., bytes in memory follow the "natural" order): Big Endian
- Instead, Intel processors use Little Endian:



Register vs. Memory order

- In registers, values are always in the “correct” order (i.e., the Big Endian order), which makes sense mathematically
- On a Little Endian machine, each time you write a register value to RAM or you read a RAM value into a register, then the byte order is reversed!
- In declaration, integer values are written/assumed in register order, but they values in memory are in RAM order



Little/Big Endian

- Motorola and IBM processors use(d) Big Endian
- Intel/AMD uses Little Endian (used in this class)
- When writing code in a high-level language one rarely cares
- But in languages that expose addresses, like in C, one can definitely expose the Endianness of the computer
- And thus one can write C code that's not portable between an IBM and an Intel!!!
- Let's live-code a small C program that tells us whether our machine uses Little Endian or Big Endian…

Little/Big Endian

- Endianness only matters when writing **multi-byte** quantities to memory and reading them differently (e.g., byte per byte)
- When writing assembly code one often does not care, but we'll see several examples when it matters, so it's important to know this *inside out*
  - *And we just saw that it can matter in C*
- Some processors are configurable (either in hardware or in software) to use either type of endianness (e.g., MIPS)

Example

```
unsigned char pixels[4] = {0xFD, 0xFD, 0xFD, 0xFD};
int x = 0b00010111001101100001010111010011;
char *blurb = "adbh";
char buffer[10] = {014,014,014,014,014,014,014,014};
signed int min = -19;
```

- What is the layout and the content of the data memory segment on a Little Endian machine?
  - Byte per byte, in hex

Example

```
unsigned char pixels[4] = {0xFD, 0xFD, 0xFD, 0xFD};
int x = 0b00010111001101100001010111010011;
char *blurb = "adbh";
char buffer[10] = {014,014,014,014,014,014,014,014};
signed int min = -19;
```

- First thing to do: identify the multi-byte quantities
  - Because endianness only matters for multi-byte quantities, within which by order is changed

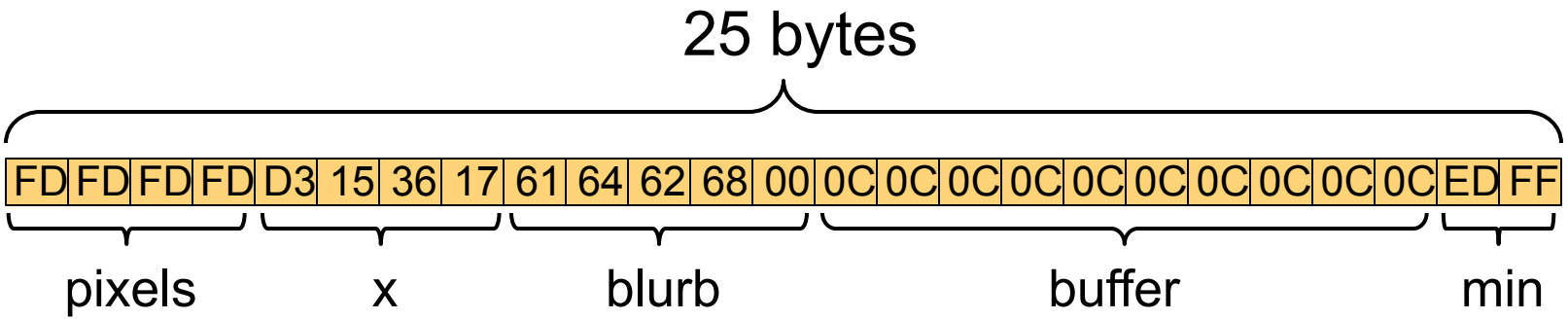
Example

```
unsigned char pixels[4] = {0xFD, 0xFD, 0xFD, 0xFD};
int x = 0b00010111001101100001010111010011;
char *blurb = "adbh";
char buffer[10] = {014,014,014,014,014,014,014,014};
signed int min = -19;
```

- First thing to do: identify the multi-byte quantities
  - EVERYTHING THAT'S NOT DECLARED AS "char" IS MULTI-BYTE, AND BYTE ORDER WILL BE "REVERSED" DUE TO LITTNE ENDIAN
- In the above: **`x`** and **`min`** above are multi-byte values

Example

```
unsigned char pixels[4] = {0xFD, 0xFD, 0xFD, 0xFD};
int x = 0b00010111001101100001010111010011;
char *blurb = "adbh";
char buffer[10] = {014,014,014,014,014,014,014,014};
signed int min = -19;
```



Note that bits within byte are NOT reversed

What about Networks??

- Say you have a network with
  - Machines that communicate messages that contain integers
  - Some of them use Big Endian and some use Little Endian
  - This happens all the time, everywhere
- If you've taken any OS/networking course, you know that data is read/written from/to RAM and then written/read to/from the network "as is" (i.e., it doesn't come from registers)
- **We have a problem:** on a Little Endian machine the 4 hex bytes "00 00 00 10" in RAM mean 2^{31}, while on a Big Endian machine, they mean 16
- Somebody has to "lose", and it turns out the network order is defined as the Big Endian order: every multi-byte quantity on the network must always be in the Big Endian order
- And so, when doing networking, **ALL** Little Endian machines must swap bytes when sending / receiving
  - That's ok, it doesn't take much time at all
- But how do we do this in practice?

Network Byte Order

- Say we need to exchange messages that contain this data structure:

```
struct {
  int a;
  short b;
} my_data;
```

- These are 6 contiguous bytes, and so, we can send them over the network doing something like this:

```
send_to_network(&my_data, 6);
```

  - See a networking course for the gory details about low-level communication APIs

- Let's just assume we have hidden all networking stuff in the above function, which is defined as:

```
void send_to_network(void *ptr, int num_bytes);
```

Network Byte Order

```
struct {
  int a;
  short b;
} my_data;

send_to_network(&my_data, 6);
```

- The above code will send the data over to the network exactly as it is stored in RAM
- So on a Little Endian machine, that's wrong!
- What we need to do is "package" our 6 bytes into a buffer, where each multi-byte value has been
- Do do that, all systems provide helper functions
- Let's see the code…

Network Byte Order

```
struct {
  int a;
  short b;
} my_data;

// Create a 6-byte buffer
char buffer[6];

// Reverse the bytes of 4-byte int into a new variable
int net_a = htonl(my_data.a);
// Reverse the bytes of the 2-byte short into a new variable
short net_b  = htons(my_data.b);

// Copy the reversed values into the buffer
memcpy(buffer, &net_a, sizeof(net_a));
memcpy(buffer + sizeof(net_a), &net_b, sizeof(net_b));

// Send over to the network in network byte-order
send_to_network(buffer, 6);
```

Network Byte Order

- Of course, the same shenanigans have to happen on the receiving side
- On a Big Endian machine, the **`hton*()`** functions just do nothing
- On a Little Endian machine, the **`hton*()`** functions perform the byte reversal
- That way, the C code on the previous slide will work on any machine and is thus portable
  - With some overhead of course…
- Wouldn't the world be nicer if everybody did the same endianness? :)

Important Takeaways

- The three sections of a NASM program (data, bss, and text)
- Declaration of numbers if different bases
- The fact that labels in assembly declarations are really just addresses, and not at all values like in high-level code (which correspond to variables with data types)
- Little and Big Endianness
- How networks deal with endianness

Conclusion

- Let's do some of the posted practice problems…
- We now have two things:
  - Places to put data: registers (previous set of lecture notes)
  - Places to declare data: RAM (this set of lecture notes)
- We can now look at programs that use/ modify data in registers and in memory, in the next set of lecture notes…