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

```
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
```

declaration of initialized data

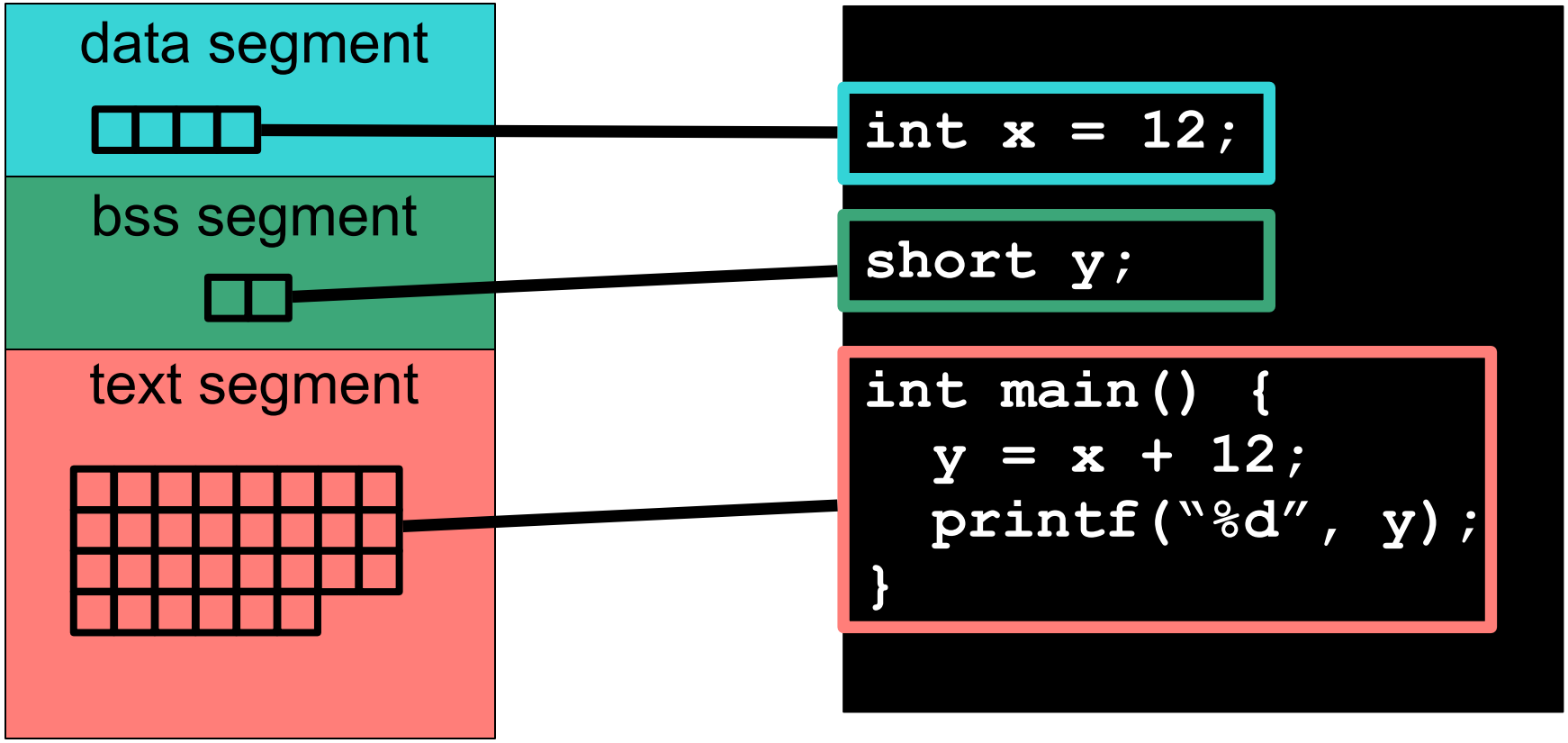
declaration of uninitialized data

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

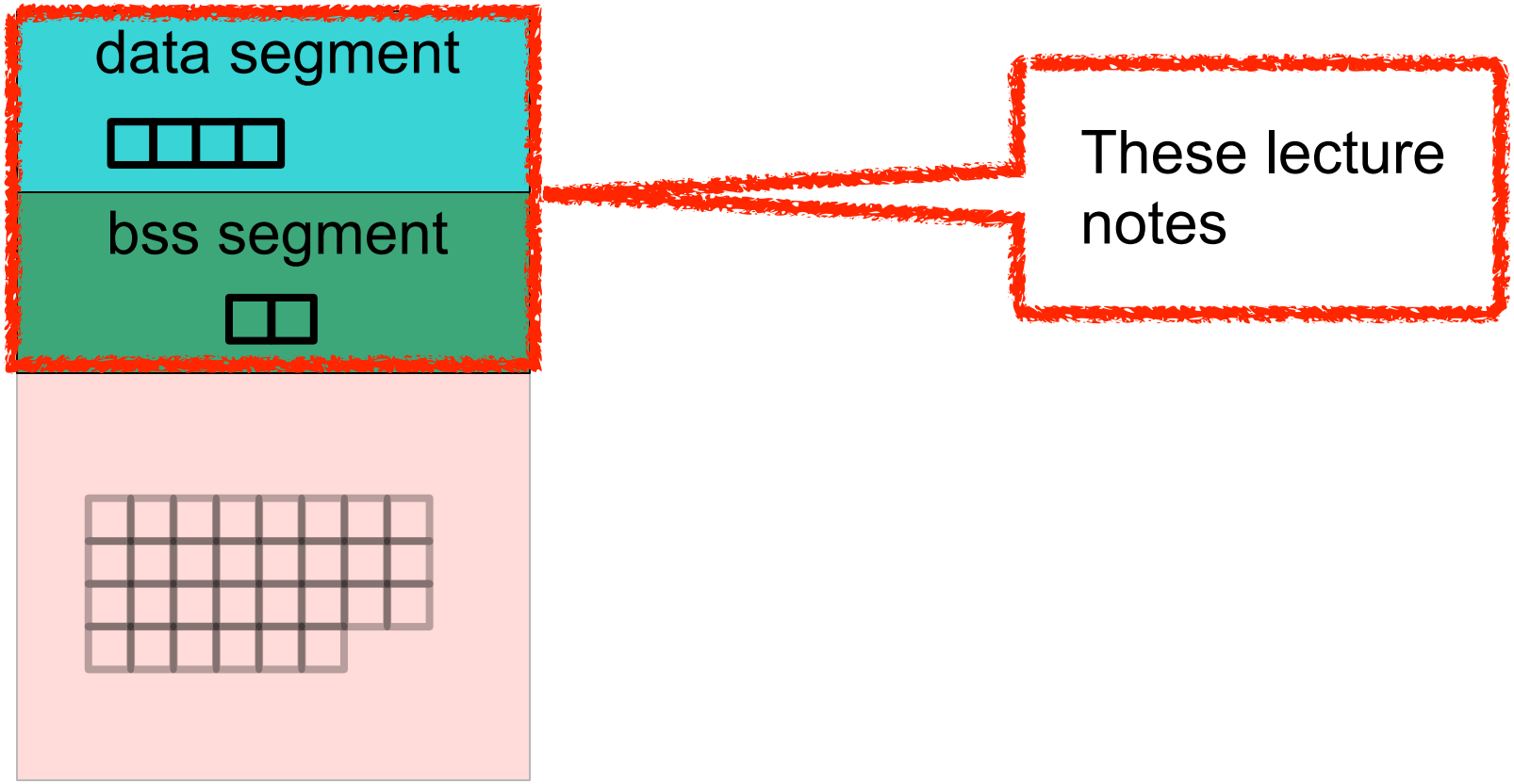
code

- 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 code



Relation to high-level code



The data and bss segments

- Both segments contains data directives that declare pre-allocated zones of memory
- There are two kinds of data directives
  - **DX directives**: initialized data (D = “defined”)
  - **RESX directives**: uninitialized data (RES = “reserved”)
- The “X” above refers to the data size:

Unit	Letter(X)	Size in bytes
byte	**B**	**1**
word	**W**	**2**
double word	**D**	**4**
quad word	Q	8
ten bytes	T	10

The DX data directives

- One declares a zone of initialized memory using three elements:
  - Label: the name used in the program to refer to that zone of memory
    - A pointer to the zone of memory, i.e., **an address**
  - DX, where X is the appropriate letter for the size of the data being declared
  - Initial value, with encoding information
    - default: decimal
    - b: binary
    - h: hexadecimal
    - o: octal
    - quoted: ASCII

DX Examples

- L1 db 0
  - 1 byte, **whose address is named L1**, initialized to 0
- Henri dw 1000
  - 2-byte value, initialized to 1000, **the address of the first of these two bytes is named Henri**

- The label (i.e., the name) is **the address** of a byte
- In the case of multi-byte values, it's the address of the first byte
- Often we say "a 2-byte value named Henri" for simplicity, but that makes it sound like Henri is a variable that contains a 2-byte value
- This is NOT the case. **Henri is a number, which is an address**, which is the address of the first byte of the 2-byte value
- The address of the second byte of the 2-byte value is Henri + 1
- Be prepared for me saying this over and over and for students not really getting it over and over :)

DX Examples

- L1 db 0
  - 1 byte, **whose address is named L1**, initialized to 0
- Henri dw 1000
  - 2-byte word, initialized to 1000, **the address of the first of these two bytes is named Henri**
- L3 db 110101b
  - 1 byte, named L3, initialized to 110101 in binary
- what db **0**A2h
  - 1 byte, named what, initialized to A2 in hex (note the '**0**')
- L5 db 17o
  - 1 byte, named L5, initialized to 17 in octal (1*8+7=15 in decimal)
- L6 dd **0**FFFF1A92h (note the '**0**')
  - 4-byte double word, named L6, initialized to FFFF1A92 in hex
- L7 db "A"

ASCII Code

- Associates 1-byte numerical codes to characters
  - Unicode, proposed much later, uses 2 bytes and thus can encode 2^8 times more characters (room for all languages, Chinese, Japanese, accents, etc.)
- A few values to know:
  - ‘A’ is 65d / 41h
    - ‘B’ is 66d / 42h, etc…
  - ‘a’ is 97d / 61h
    - ‘b’ is 98d / 62h, etc…

DX for multiple elements

- Foo db 0, 1, 2, 3
  - Defines 4 1-byte values, initialized to 0, 1, 2 and 3
  - Foo is a pointer to (i.e., the address of) the first byte
- The above is equivalent (in terms of memory content) to:
  - Stuff db 0
  - What db 1
  - Eggplant db 2
  - Chair db 3
- The only difference is that in the second version we have a name (label) for the address of each of the four bytes

Strings as sequences of chars

- L9 db “w”, “o”, ‘r’, ‘d’, **0**
  - Defines 5 1-byte values, the first 4 being initialized by an ASCII code (i.e., a character)
  - Defines a **null-terminated** string, initialized to “word\0”
  - L9 is a pointer to the beginning of the string (i.e., the address of the first character of the string)
- L10 db “word”, 0
  - Equivalent to the above, more convenient to write

DX with the times qualifier

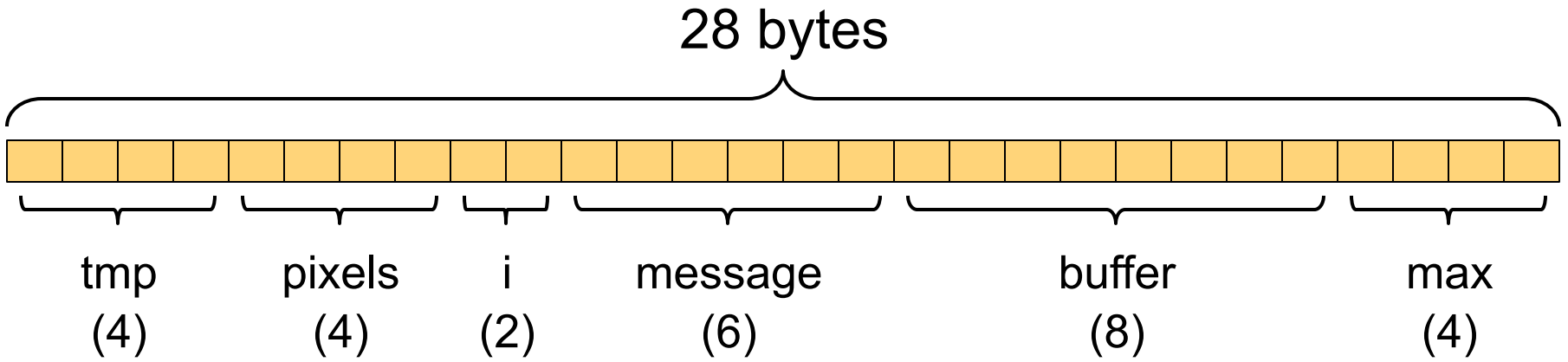
- Say you want to declare 100 bytes all initialized to 0
- NASM provides a nice shortcut to do this, the “times” qualifier
- L11 times 100 db 0
  - Equivalent to L11 db 0,0,0,....,0 (100 times)

Uninitialized Data

- The RESX directive is very similar to the DX directive, but always specifies the number of values to reserve space for
- L20 resw 100
  - 100 uninitialized 2-byte values (so 200 bytes in total)
  - L20 is a pointer to the first (byte of the first) 2-byte value
  - This can't be seen as a 100-element array!
- stuff resb 1
  - One uninitialized byte "named" stuff

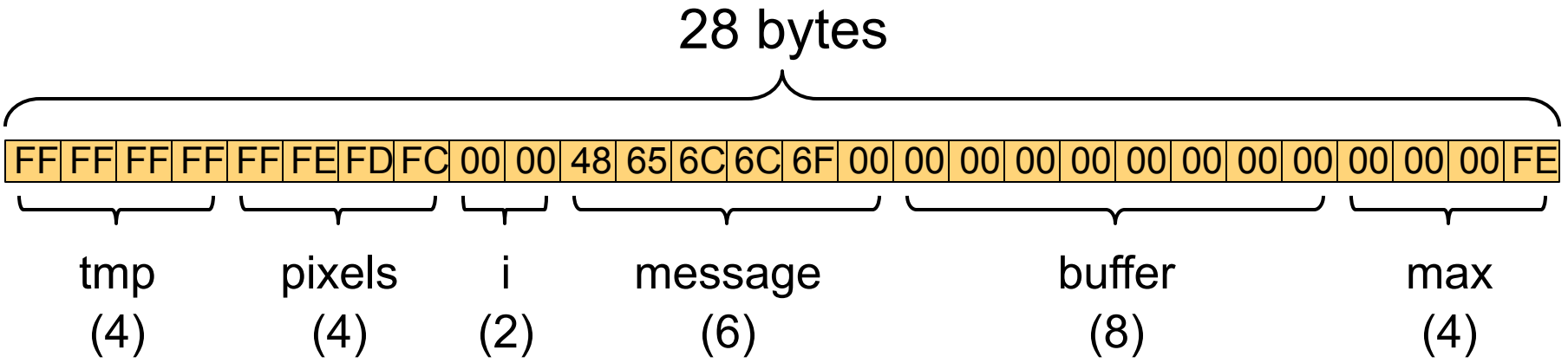
Data segment example

```
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
```



Data segment example

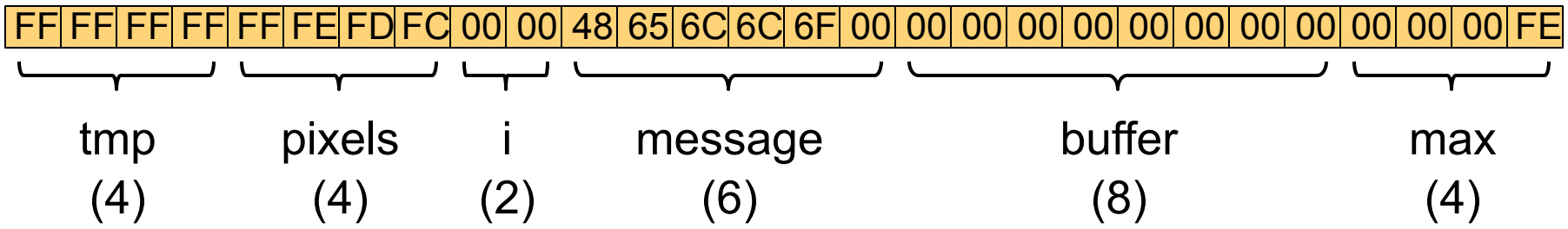
```
tmp         dd     -1
pixels      db     0FFh, 0FEh, 0FDh, 0FCh
i           dw     0
message     db     "H", "e", "llo", 0
buffer      times   8     db   0
max         dd     254
```



Data segment example

```
tmp         dd      -1
pixels
i
message
buffer
max
```

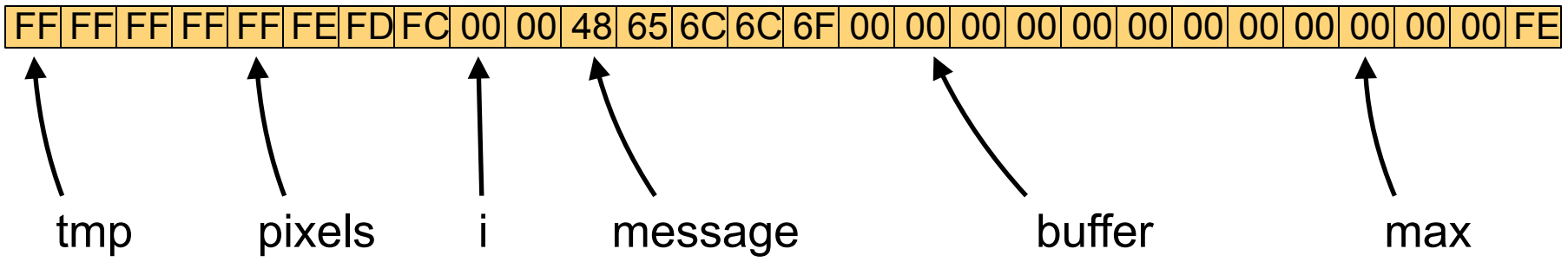
This way of drawing the memory content is actually confusing, because it makes it look like labels are variables. A label is only the address of a byte!
So let's redraw it…



Data segment example

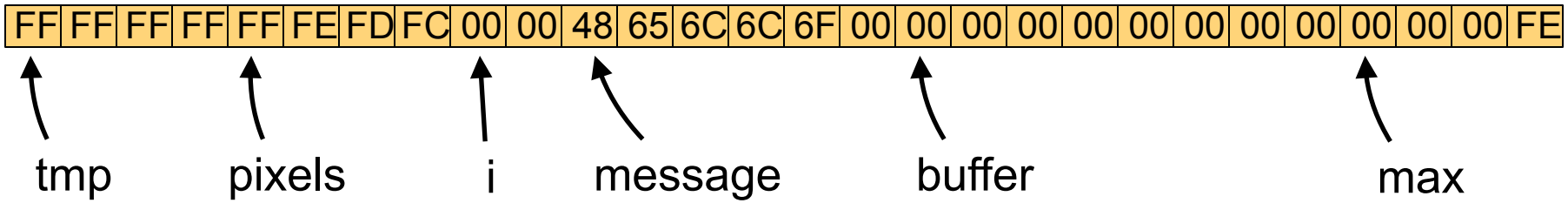
```
tmp             dd      -1
pixels
i
message
buffer
max
```

This is much better, but still a bit strange perhaps because the address of a byte is a number, not a symbol (like “tmp”)…



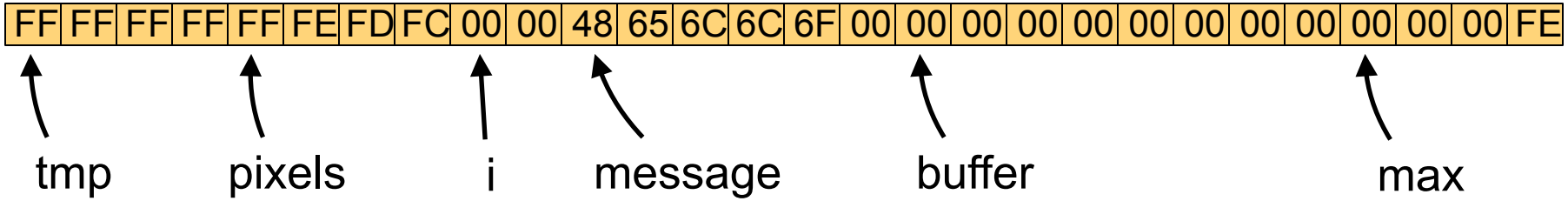
Labels are symbolic

- Labels (L1, Henri, etc.) are symbolic names of 32-bit integers that are addresses of particular bytes
- When the program actually runs, each label then corresponds to an actual 32-bit numerical value
- For instance, say we have this:



Labels are symbolic

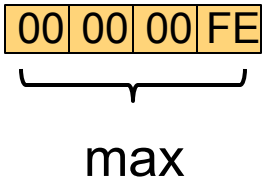
- Labels (L1, Henri, etc.) are symbolic names of 32-bit integers that are addresses of particular bytes
- When the program actually runs, each label then corresponds to an actual 32-bit numerical value
- For instance, say we have this:



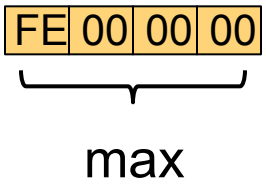
- At run time, at some point, the above bytes are put in RAM somewhere by the OS
- At that point, “tmp” is replaced by a numerical value, say AABBCC00
  - And therefore, “pixels” is replaced by AABBCC04 (because pixels = tmp + 4)
  - And “buffer” is replaced by AABBCD0 (because buffer = tmp + 16)
- When we reason about a memory layout though, we don’t typically give a specific value to the addresses, but just reason relative to the first byte

Endianness?

```
max             dd      254
```

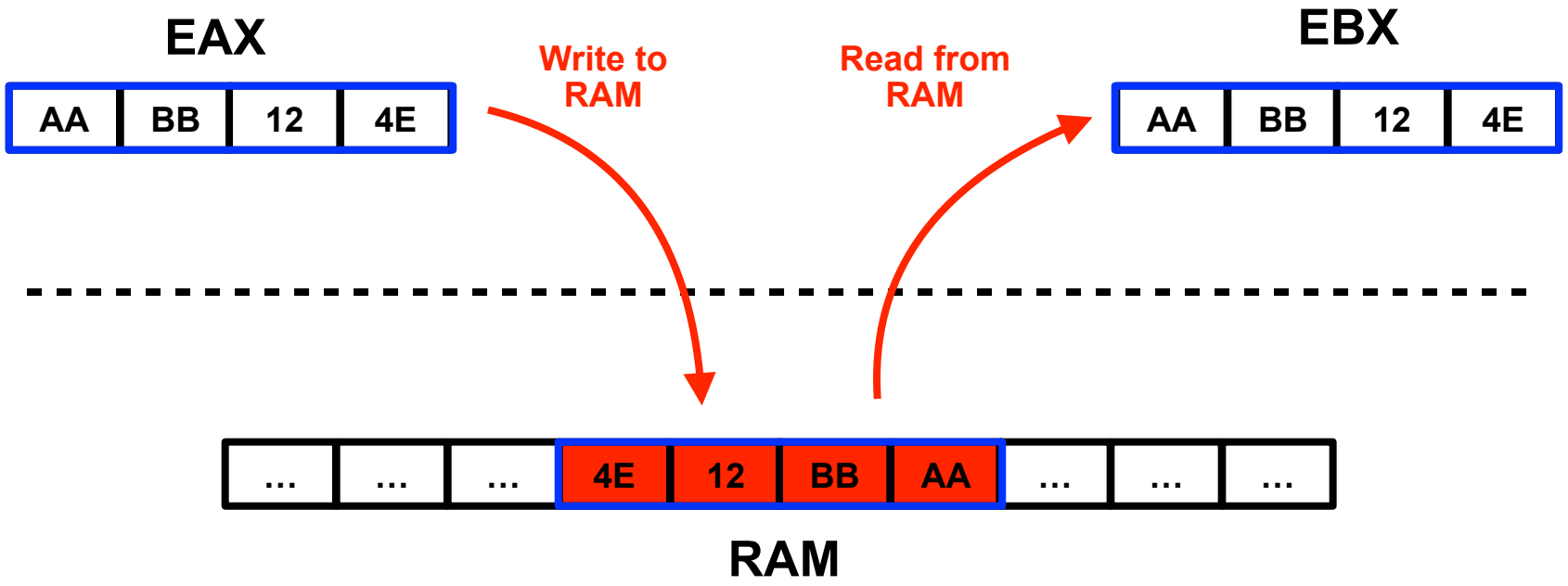


- In the previous slide I showed the above 4-byte memory content for a double-word that contains 254 = 000000FEh
- While this seems to make sense, it turns out that Intel processors do not do this!
  - Yes, the last 4 bytes shown in the previous slide are wrong
- The scheme shown above (i.e., bytes in 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!



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 do this RIGHT NOW just for fun…

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*
- Some processors are configurable (either in hardware or in software) to use either type of endianness (e.g., MIPS)

Example

```
pixels      times 4     db    0FDh
x           dd    0001011100110110000101011101 0011b
blurb       db    "ad", "b", "h", 0
buffer      times  10   db  14o
min         dw    -19
```

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

Example

```
pixels      times 4     db    0FDh
x           dd    0001011100110110000101011101 0011b
blurb       db    "ad", "b", "h", 0
buffer      times  10   db  14o
min         dw    -19
```

- First thing to do: identify the multi-byte quantities

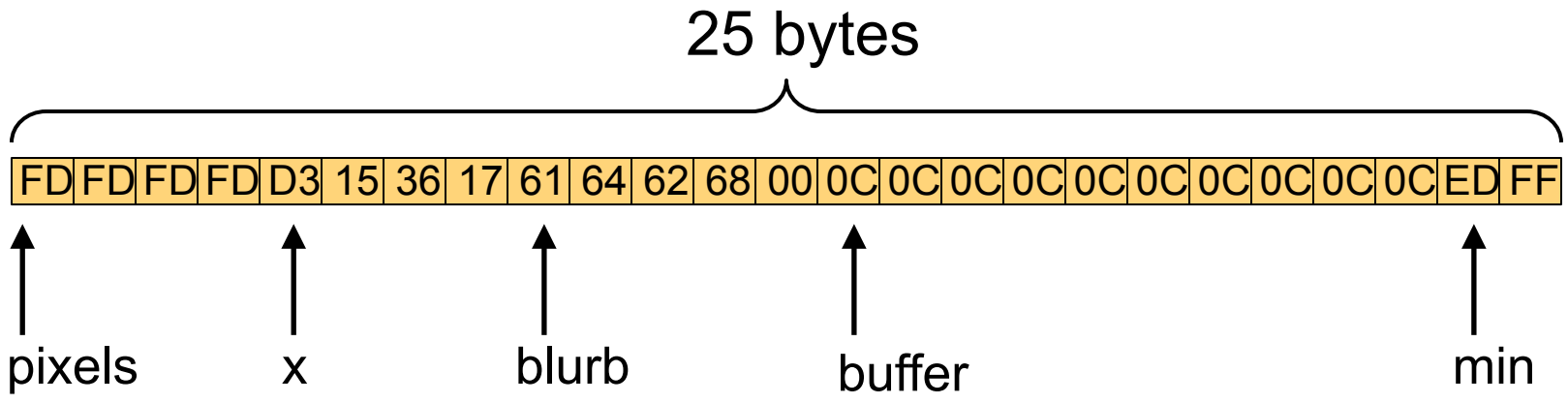
Example

```
pixels      times 4     db    0FDh
x           dd    0001011100110110000101011101 0011b
blurb       db    "ad", "b", "h", 0
buffer      times  10   db  14o
min         dw    -19
```

- First thing to do: identify the multi-byte quantities
  - EVERYTHING THAT'S NOT DECLARED AS "db" IS MULTI-BYTE
    - (L db "stuff" is NOT MULTI-BYTE, it's a sequence of bytes)
- In the above: **x** and **min** above are multi-byte values

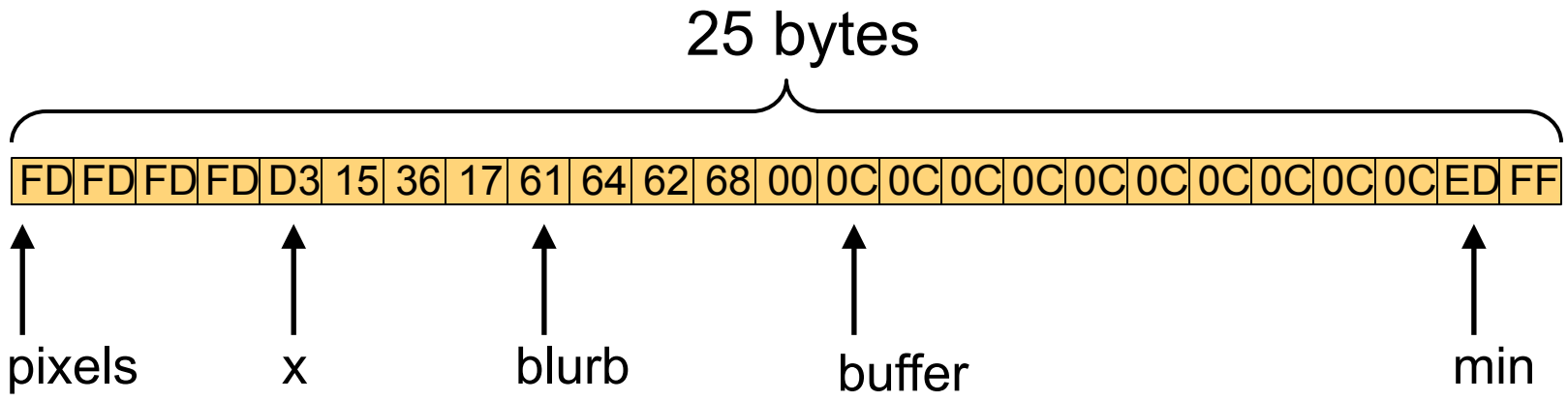
Example

```
pixels      times 4     db    0FDh
x           dd    00010111001101100001010111010011b
blurb       db    "ad", "b", "h", 0
buffer      times  10   db  14o
min         dw    -19
```



Example

```
pixels      times 4     db    0FDh
x           dd    00010111001101100001010111010011b
blurb       db    "ad", "b", "h", 0
buffer      times  10   db  14o
min         dw    -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)
- The data and bss directives for declaring labelled bytes
- The fact that labels are really addresses, and not at all variables
- Little and Big Endianness
- How networks deal with endianness

Conclusion

- Let’s do some of the posted practice problems…
- We can now look at programs that use registers and memory, in the next set of lecture notes