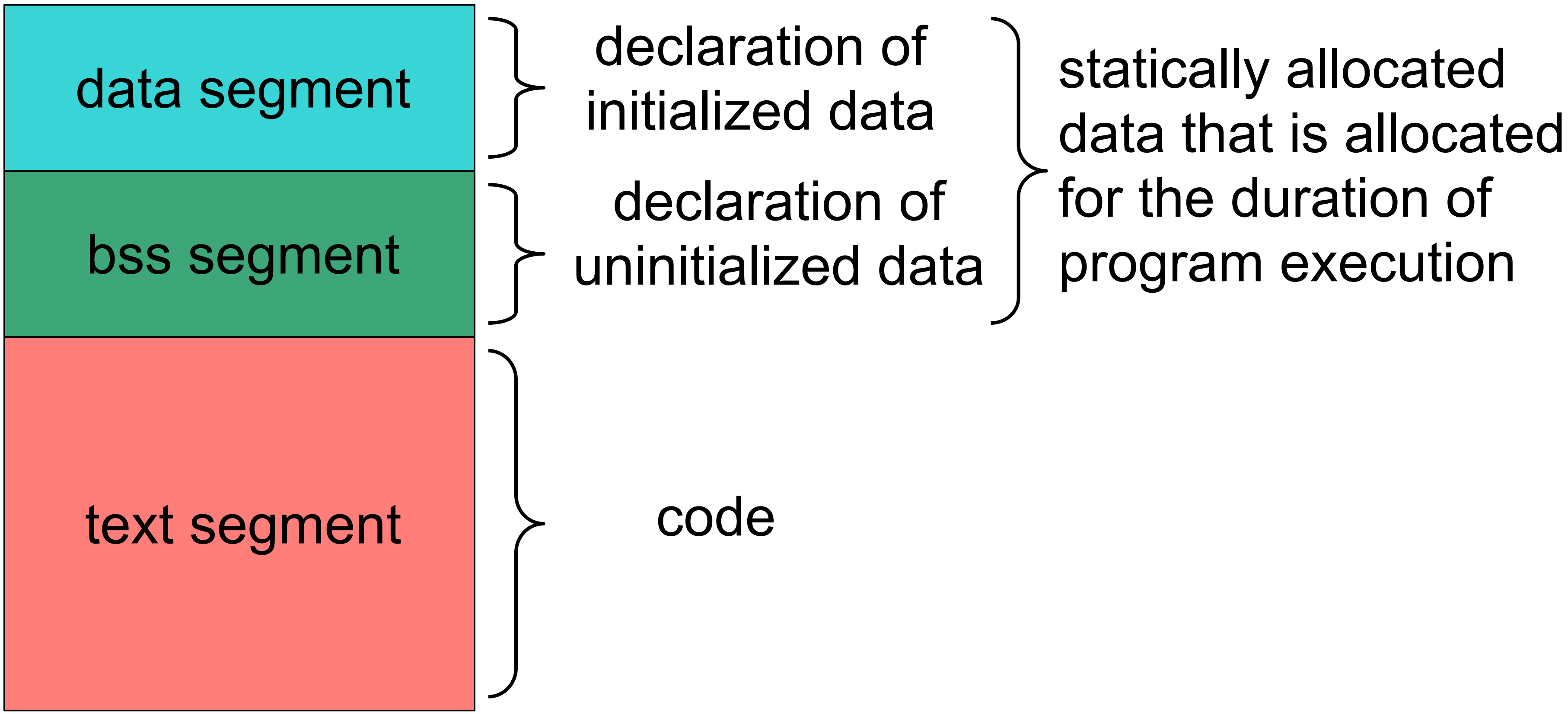
NASM: data and bss Introduction

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

NASM Program Structure



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 word, named Henri, initialized to 1000
- 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”
  - 1 byte, named L7, initialized to the ASCII code for “A” (65d)

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

- L8 db 0, 1, 2, 3
  - Defines 4 1-byte values, initialized to 0, 1, 2 and 3
  - L8 is a pointer to (i.e., the address of) the first byte
- The above is equivalent (in terms of memory content) to:
  - L8 db 0
  - L9 db 1
  - L10 db 2
  - L11 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

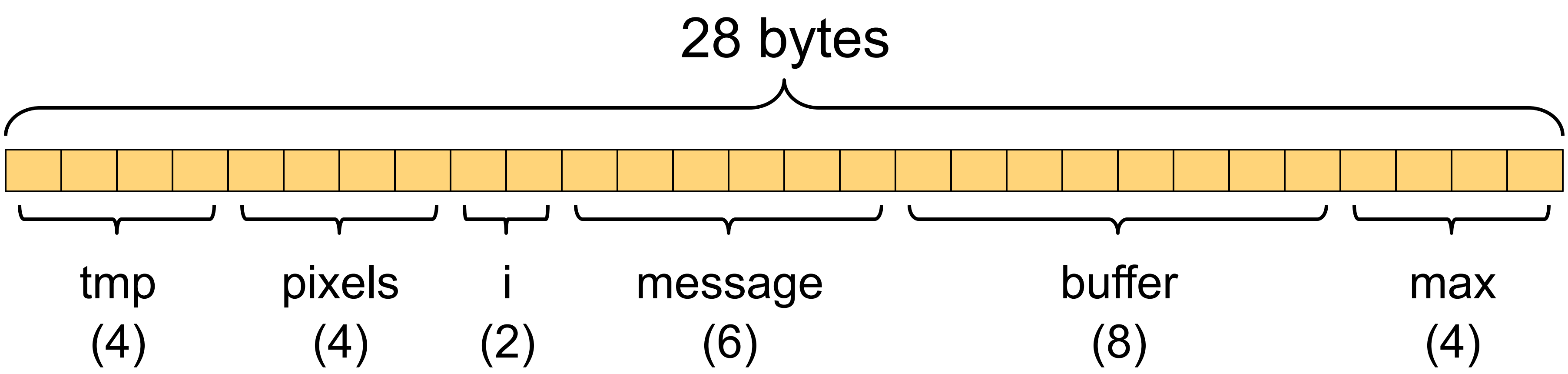
- Say you want to declare 100 bytes all initialized to 0
- NASM provides a nice shortcut to do this, the “times” qualifier
- L11 times 100 db 0
  - Equivalent to L11 db 0,0,0,....,0 (100 times)

Uninitialized Data

- The RESX directive is very similar to the DX directive, but always specifies the number of memory elements
- L20 resw 100
  - 100 uninitialized 2-byte values
  - L20 is a pointer to the first 2-byte value
- stuff resb 1
  - 1 uninitialized byte named stuff

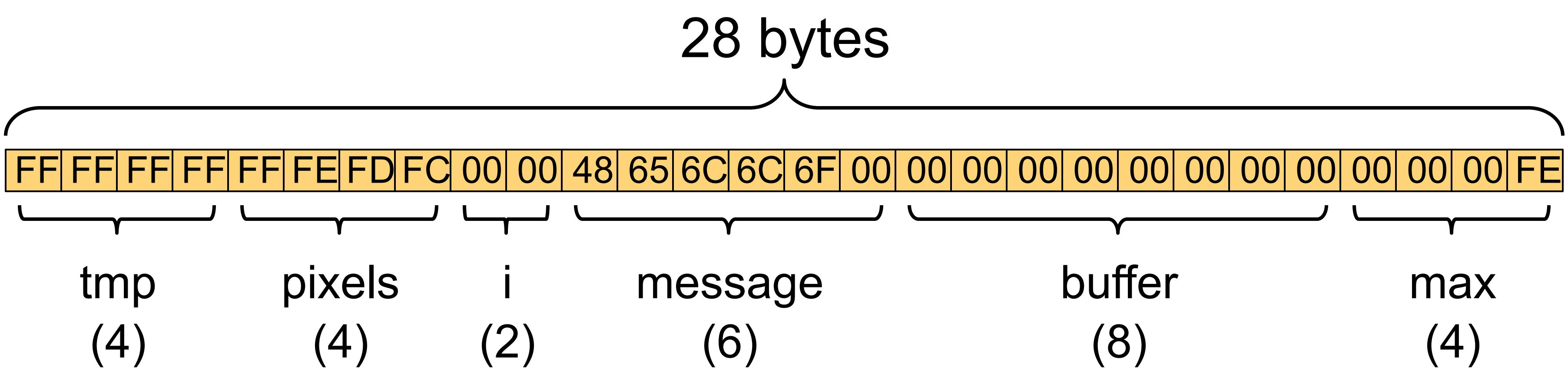
Data segment example

```
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
```



Endianness?

```
max              dd      254
```

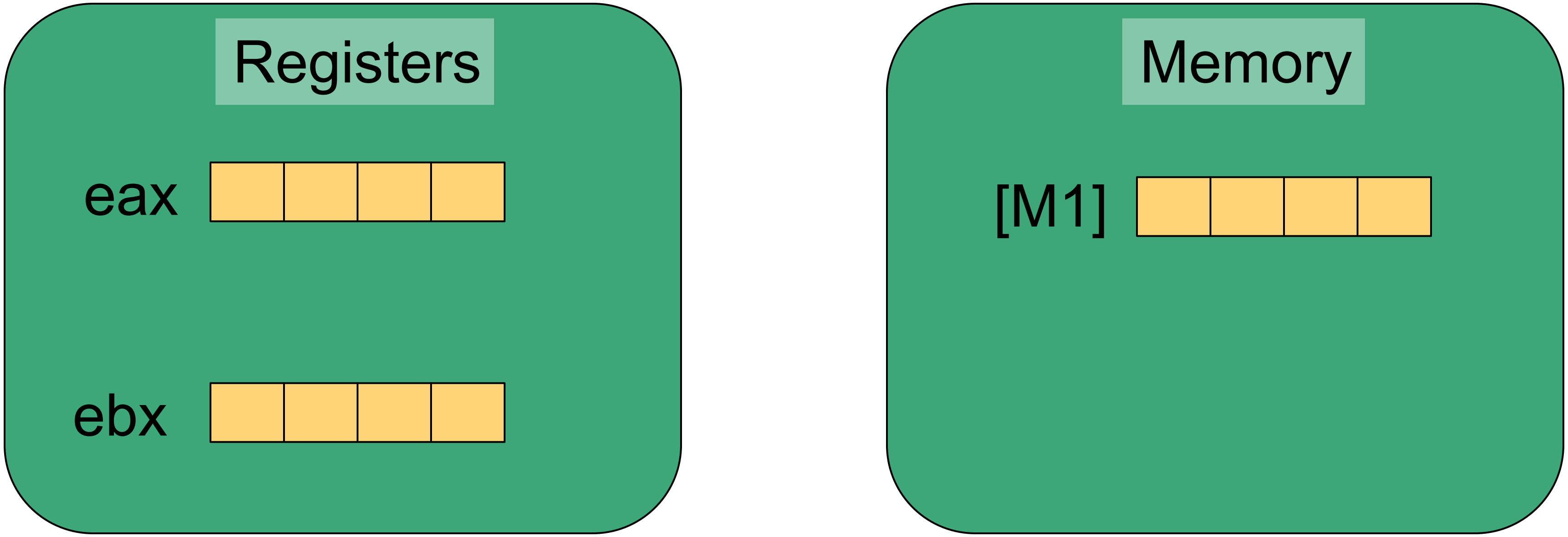
$\underbrace{\texttt{00|00|00|FE}}_{\text{max}}$

- 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:

$\underbrace{\texttt{FE|00|00|00}}_{\text{max}}$

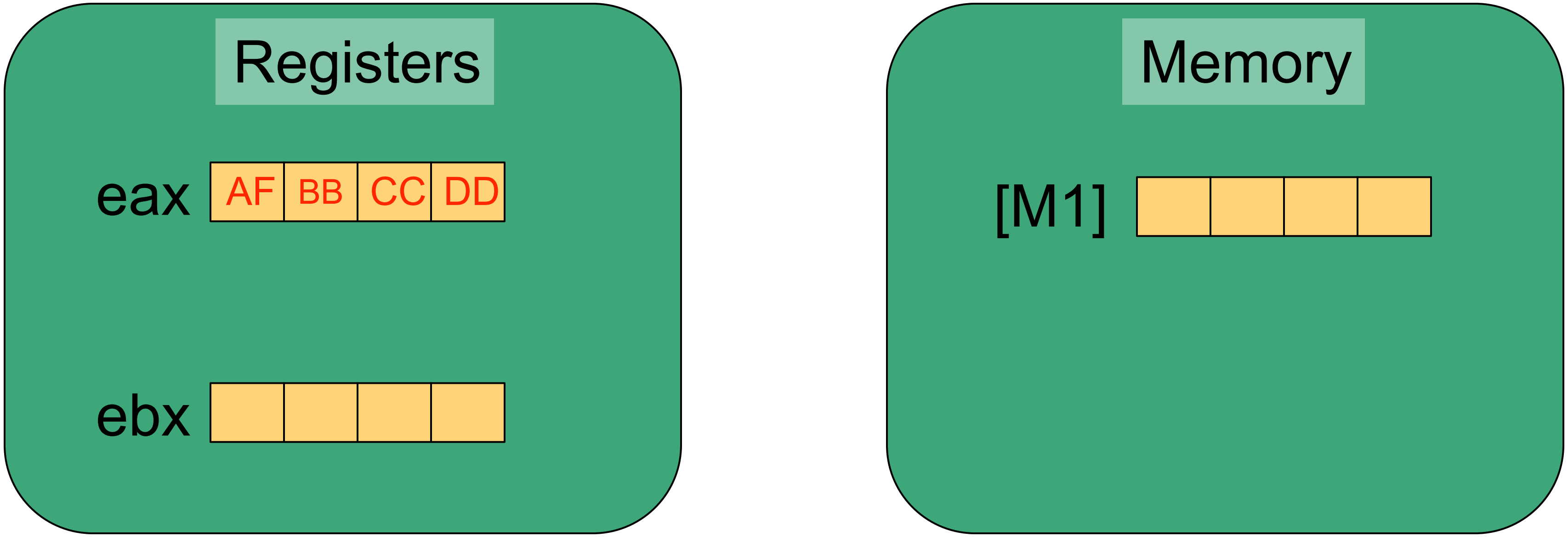
Little Endian

```
mov eax, 0AFBBCCDDh
mov [M1], eax
mov ebx, [M1]
```



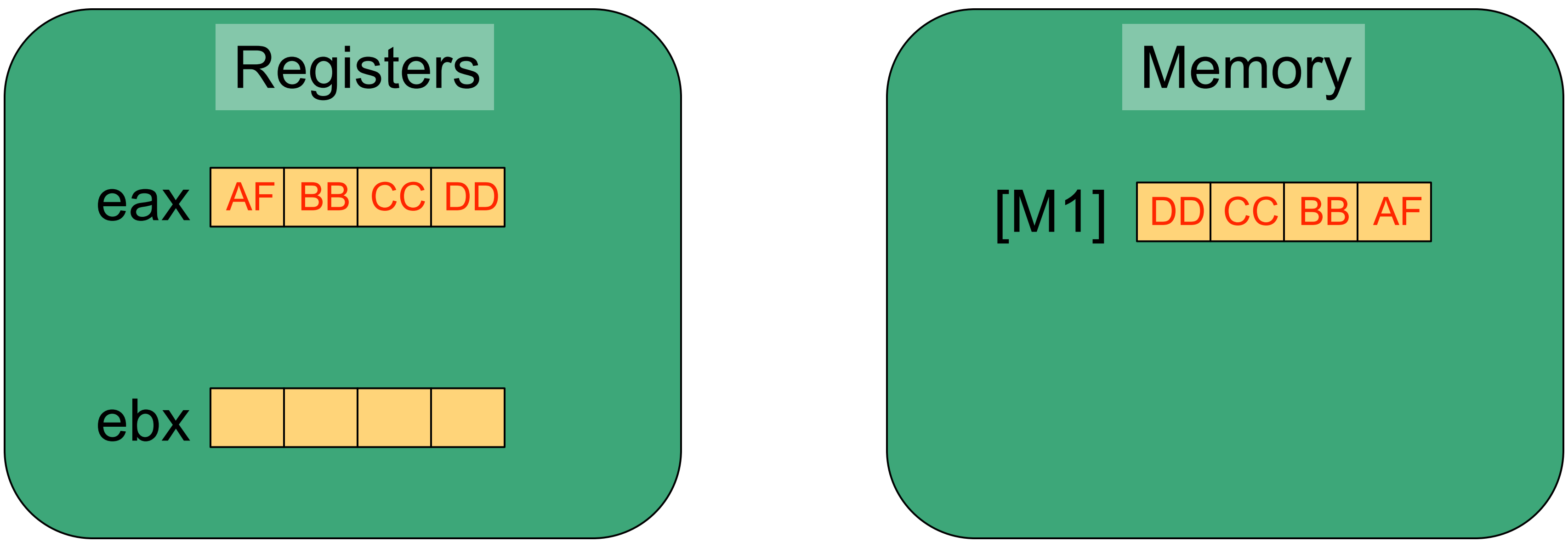
Little Endian

mov eax, 0AFBBCCDDh
mov [M1], eax
mov ebx, [M1]



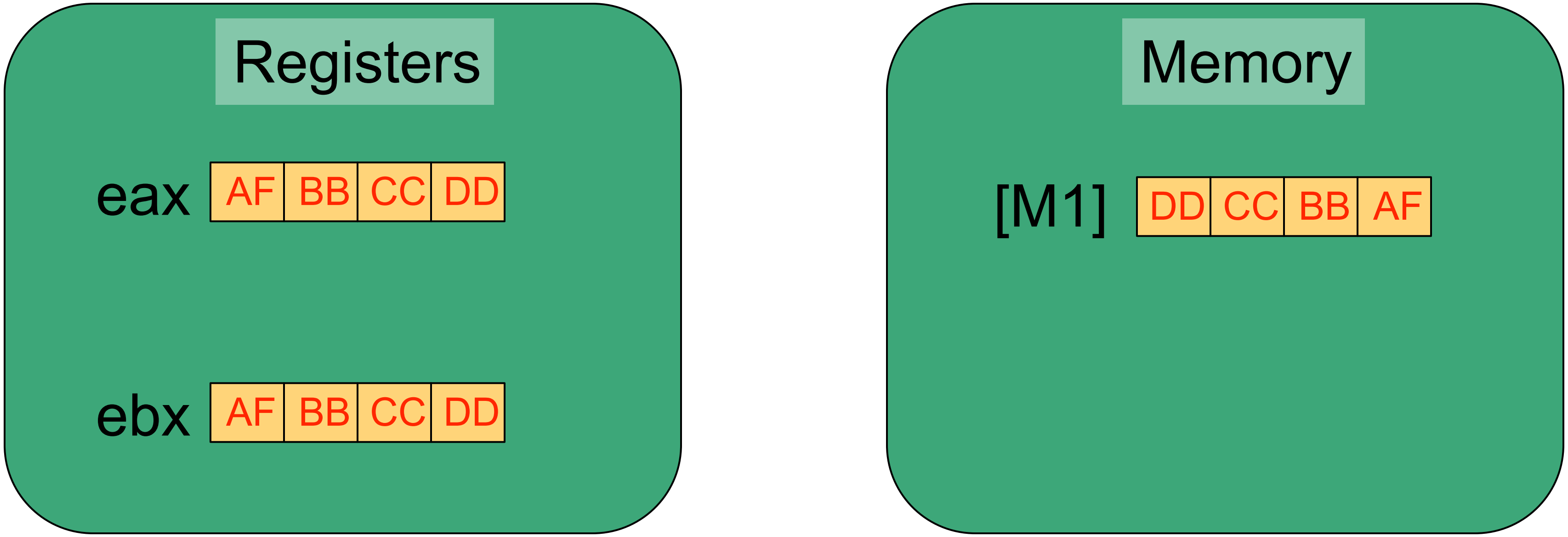
Little Endian

mov eax, 0AFBBCCDDh
mov [M1], eax
mov ebx, [M1]



Little Endian

mov eax, 0AFBBCCDDh
mov [M1], eax
mov ebx, [M1]



Little Endian

```
mov eax, 0AFBBCCDDh
mov [M1], eax
mov ebx, [M1]
```

Registers

eax | AF | BB | CC | DD

ebx | AF | BB | CC | DD

Memory

[M1] | DD | CC | BB | AF

In-register byte order and in-memory byte order, within a single multi-byte value, are 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
  - Although 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!!!
- This 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 processor)

Example

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

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

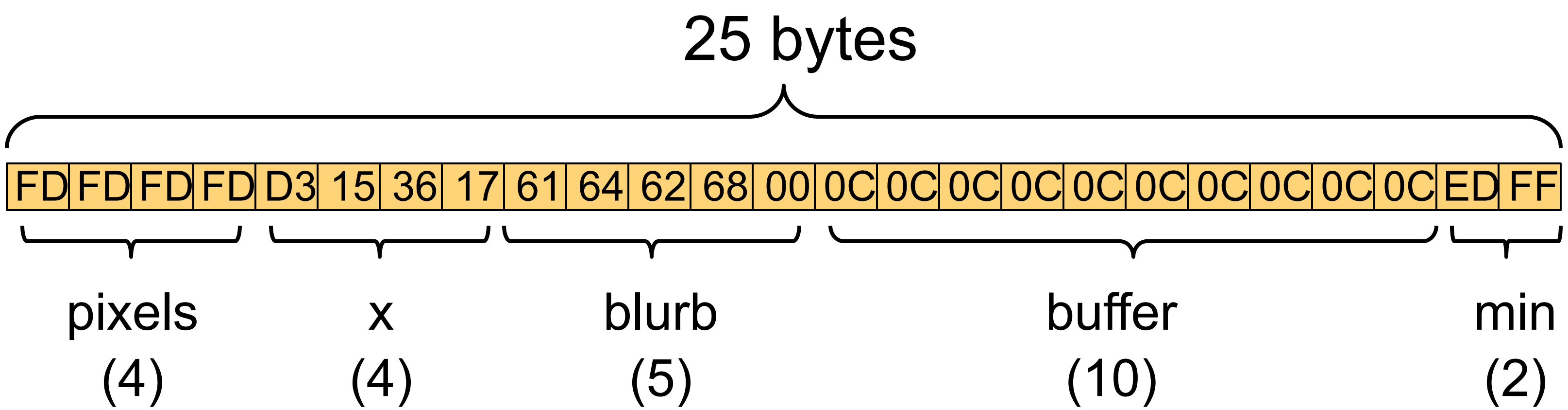
Example

```
pixels      times 4     db    0FDh
x           dd    00010111001101100001010111010011b
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)
- In the above: **`x`** and **`min`** above are multi-byte values

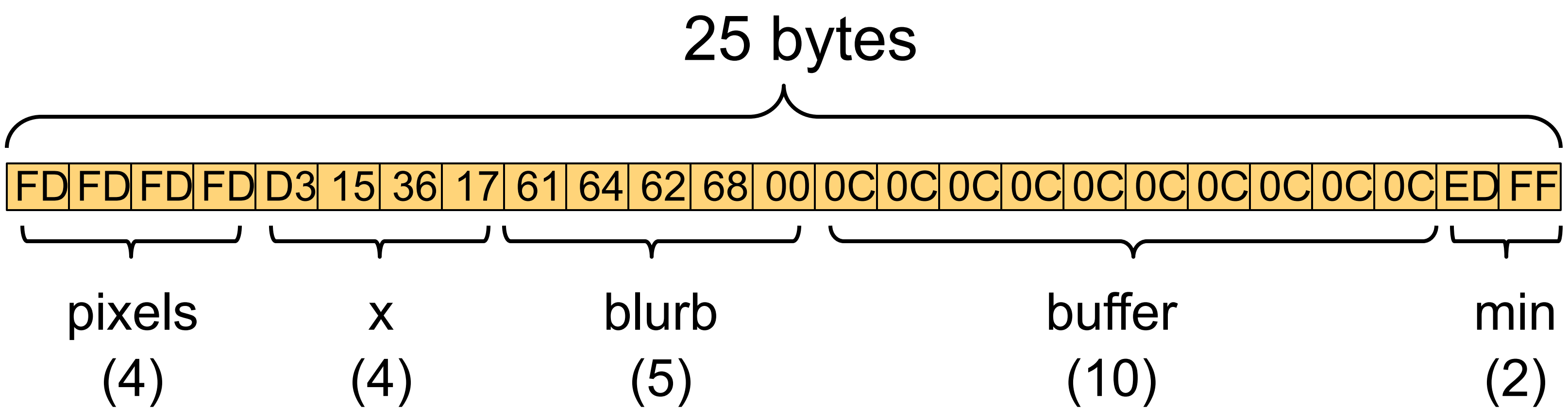
Example

```
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 with each byte are NOT reversed

What we’ll do in Class

- We’ll do some “what are the bytes in RAM given the data segment specification?” practice
- Then I’ll keep lecturing about this, which will lead to more in-class practices

- Make sure you really have your terminology down pat before you come to class
  - The ‘b’, ‘w’, ‘d’ things
  - Syntax of the data segment declarations, including the “times” qualifier
  - Little-Endian weirdness