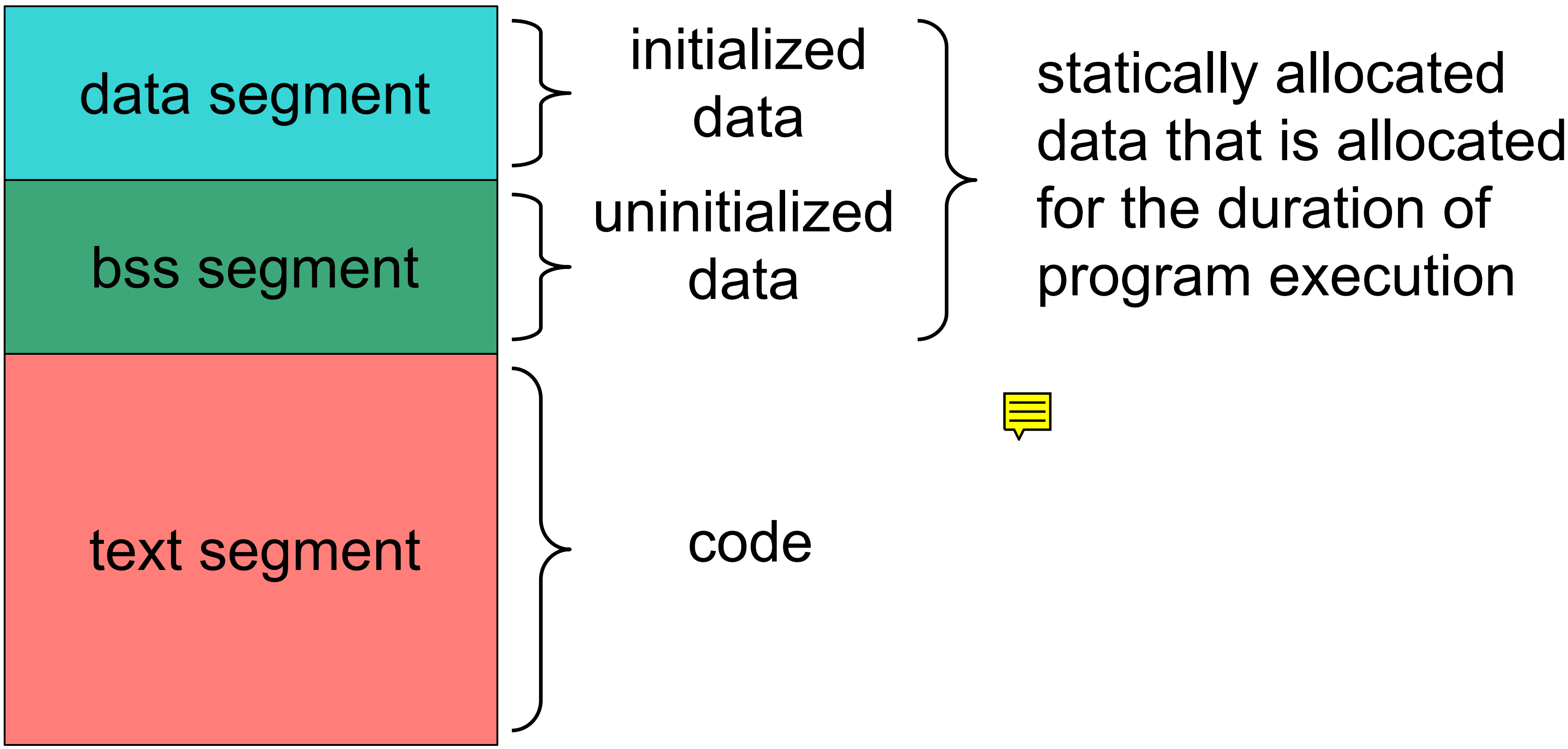
NASM: data and bss (inverted)

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, named L1, initialized to 0
- L2 dw 1000
  - 2-byte word, named L2, initialized to 1000
- L3 db 110101b
  - 1 byte, named L3, initialized to 110101 in binary
- L4 db **0**A2h
  - 1 byte, named L4, 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, ‘B’ is 66d, etc.
  - ‘a’ is 97d, ‘b’ is 98d, etc.
  - ‘ ’ is 32d

DX for multiple elements

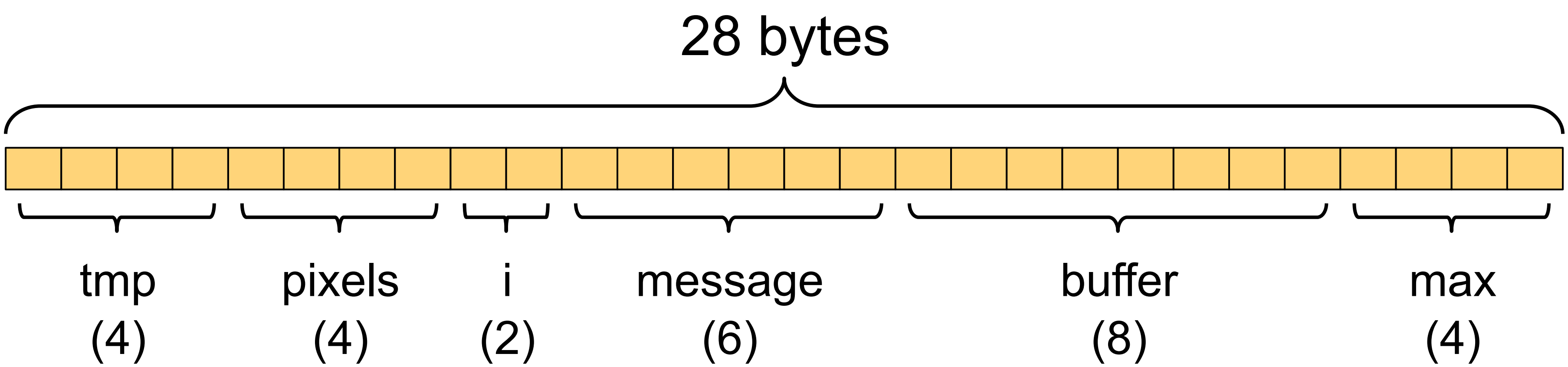
- L8 db 0, 1, 2, 3
  - Defines 4 bytes, initialized to 0, 1, 2 and 3
  - L8 is a pointer to the first byte
- L9 db "w", "o", 'r', 'd', **0**
  - Defines a **null-terminated** string, initialized to "word\0"
  - L9 is a pointer to the beginning of the string
- L10 db "word", 0
  - Equivalent to the above, 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)

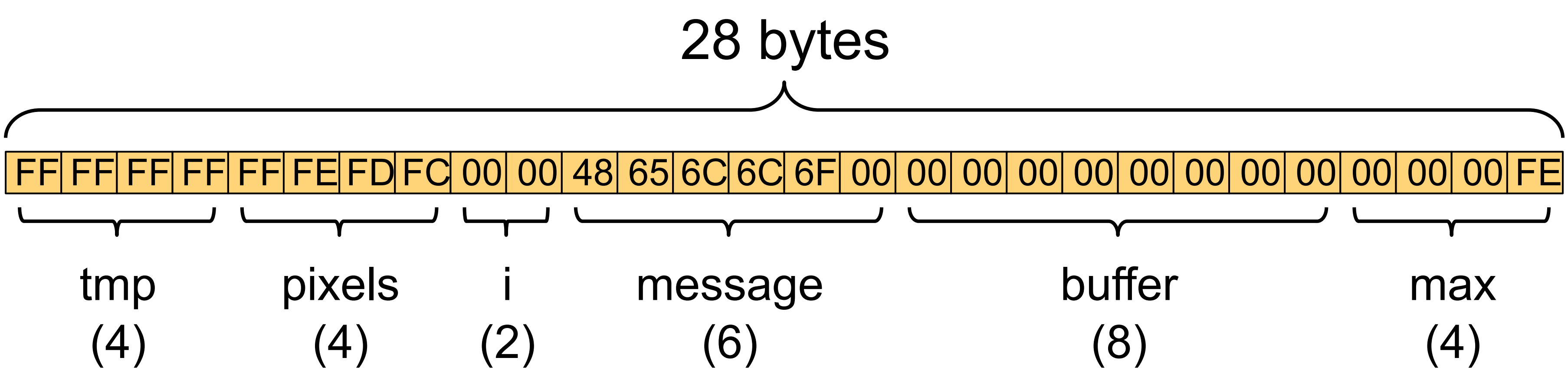
Data segment example

```
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
```

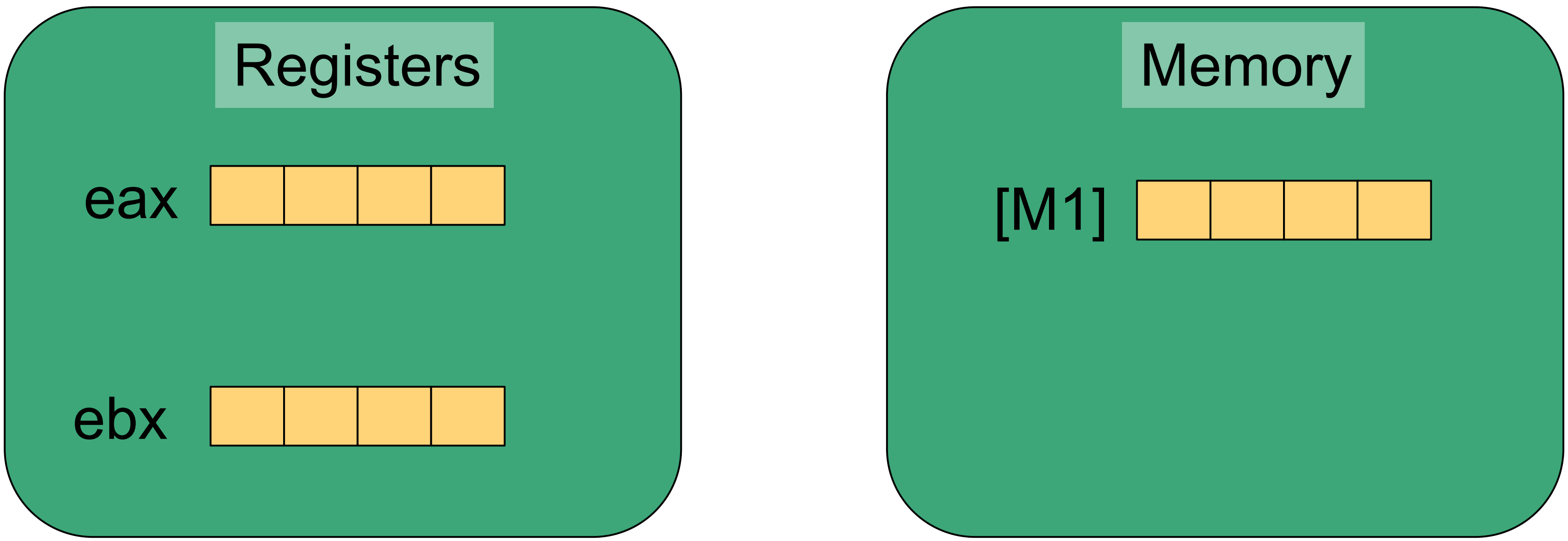
$\underbrace{\boxed{00}\boxed{00}\boxed{00}\boxed{FE}}_{\text{max}}$

- In the previous slide we 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:

$\underbrace{\boxed{FE}\boxed{00}\boxed{00}\boxed{00}}_{\text{max}}$

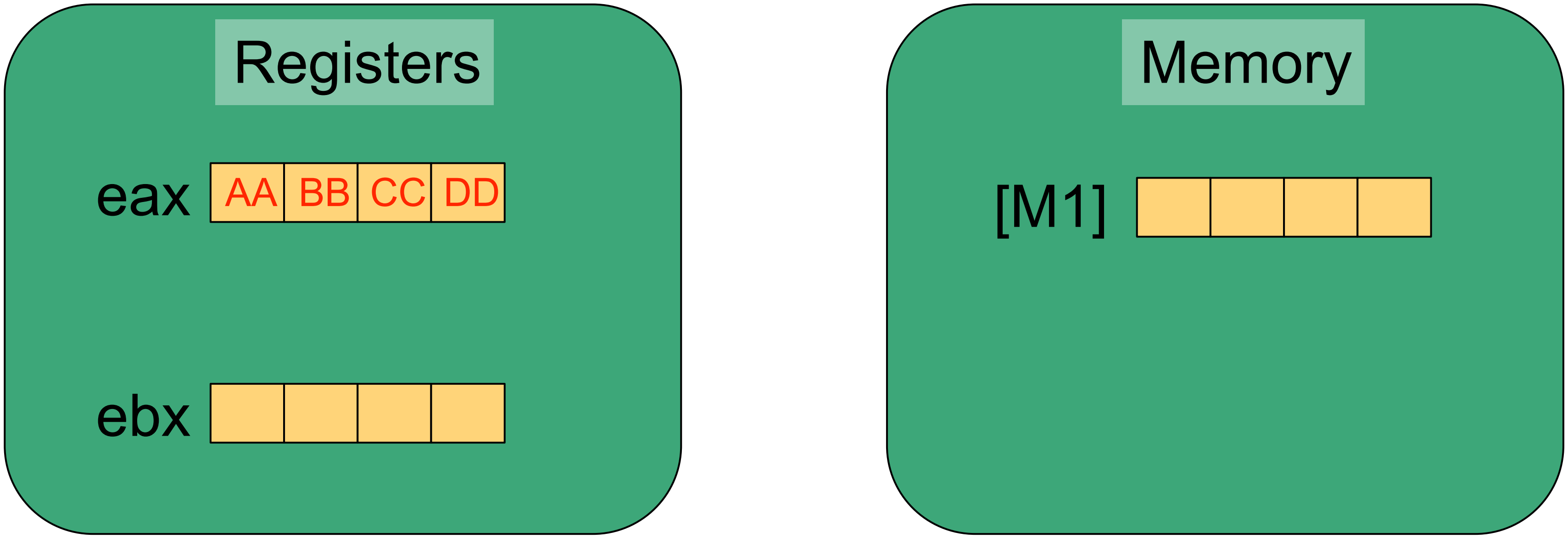
Little Endian

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



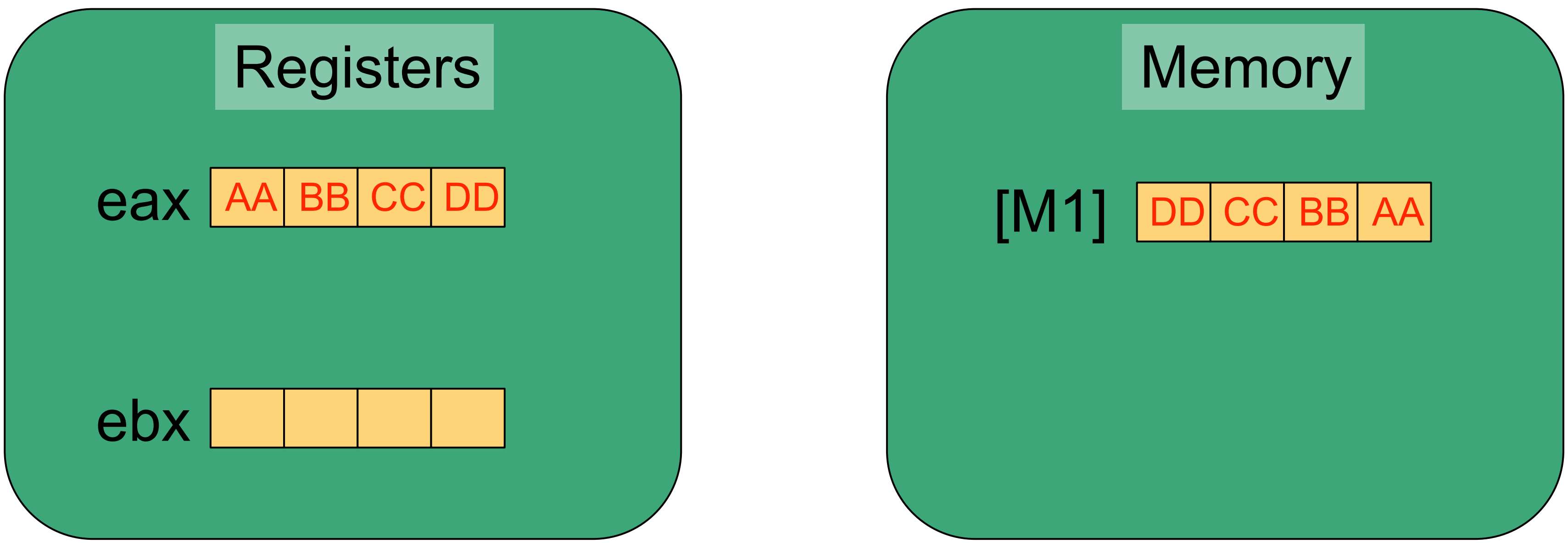
Little Endian

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



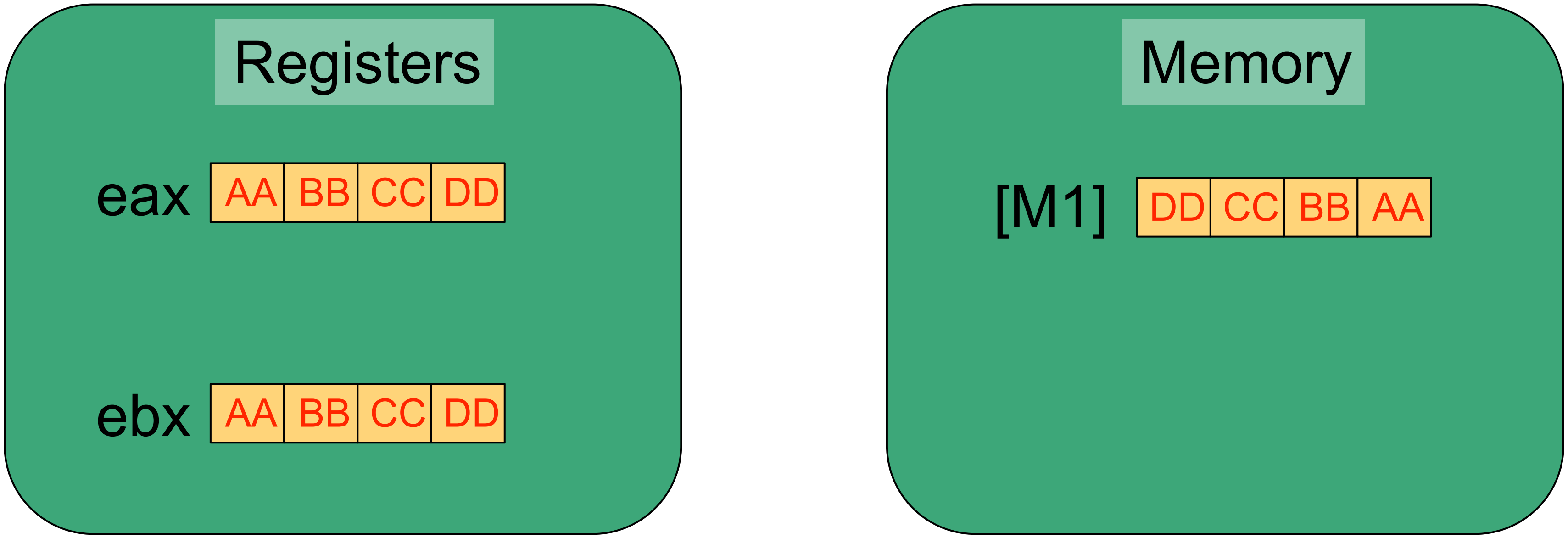
Little Endian

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



Little Endian

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



Little Endian

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

Registers

eax | AA | BB | CC | DD |

ebx | AA | BB | CC | DD |

Memory

[M1] | DD | CC | BB | AA |

In-register byte order and in-memory byte order, within a single multi-byte value, are different!

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)

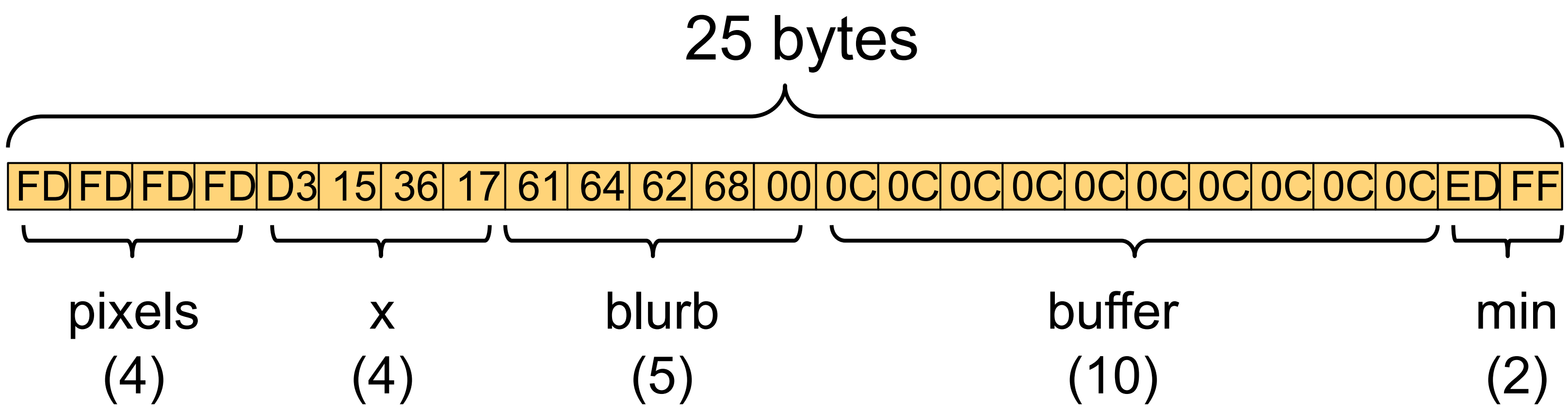
Example

```
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
```



Unitialized 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 words
  - L20 is a pointer to the first word
- L21 resb 1
  - 1 uninitialized byte named L21

Our first instructions

- At this point we need to introduce a few assembly instructions
  - adding integers
  - subtracting integers
  - moving data between registers / memory locations / constants

Simple arithmetic and operands

- Assembly instructions can have operands, and it's important to know what kind of operands are possible
- Register: specifies one of the registers
  - **add eax, ebx**
  - means eax = eax + ebx
- Memory: specifies an address in memory.
  - **add eax, [ebx]**
  - means eax = eax + content of memory at address ebx
- Immediate: specifies a fixed value (i.e., a number)
  - **add eax, 2**
  - means eax = eax + 2
- Implied: not actually encoded in the instruction
  - **inc eax**
  - means eax = eax + 1

Additions, subtractions

- Additions
  - add eax, 4 ; eax = eax + 4
  - add al, ah ; al = al + ah
- Subtractions
  - sub bx, 10 ; bx = bx - 10
  - sub ebx, edi ; ebx = ebx - edi
- Increment, Decrement
  - inc ecx ; ecx++ (a 4-byte operation)
  - dec dl ; dl-- (a 1-byte operation)

The move instruction

- This instruction moves data from one location to another

  mov **dest**, **src**
- Destination goes first, and the source goes second
- At most one of the operands can be a memory operand
  - mov eax, [ebx] ; **OK**
  - mov [eax], ebx ; **OK**
  - mov [eax], [ebx] ; **NOT OK**
- Both operands must be exactly the same size
  - For instance, AX cannot be stored into BL
- Examples:
  - mov ax, ebx ; **NOT OK**
  - mov bx, ax ; **OK**
- This type of "exceptions to the common case" make programming languages difficult to learn and assembly may be the worst offender
  - By contrast, Lisp is known for being very consistent (ICS313)

Use of Labels

- It is important to constantly be aware that when using a label in a program, the label is a pointer, not a value
- Therefore, a common use of the label in the code is as a memory operand, in between square brackets **'[' ']'**
- mov AL, [L1]
  - Move the data at address L1 into register AL
- Question: how does the assembler know how many bits to move?
- Answer: it's up to the programmer to do the right thing, that is load into appropriately sized registers
- Labels do not have a type!
- So although it's tempting to think of them as variables, they are much more limited: just pointers to a byte somewhere in memory

Moving to/from a register

- Say we have the following data segment

  L db 0F0h, 0F1h, 0F2h, 0F3h
- Example: mov AL, [L]
  - AL: Lowest bits of AX, i.e., 1 byte
  - Therefore, value F0 is moved into AL
- Example: mov [L], AX
  - Moves 2 bytes into L, overwriting the first two bytes
- Example: mov [L], EAX
  - Moves 4 bytes into L, overwriting all four bytes
- Example: mov AX, [L]
  - AX: 2 bytes
  - Therefore value F1F0 is moved into AX
  - Note that this is reversed because of Little Endian!!

More About Little Endian

- Consider the following data segment

L1 db 0AAh, 0BBh, 0CCh, 0DDh

L2 dd 0AABBCCDDh

- The instruction: mov eax, [L1]
  puts DDCCBBAA into eax
  - Note that we're loading 4x1 bytes as a 4-byte quantity
- The instruction: mov eax, [L2]
  puts AABBCCDD into eax!!!
  - Meaning that the memory content was DDCCBBAA
- When declaring a value in the data segment, that value is declared as it would be appearing in registers when loaded "whole"
  - It would be confusing to write numbers in little endian in the program
  - So all numerical values you write are in register-order not memory-order

Example

- Data segment:

```
L1    db    0AAh, 0BBh
L2    dw    0CCDDh
L3    db    0EEh, 0FFh
```

- Program:

```
mov eax, [L2]
mov ax, [L3]
mov [L1], eax
```

- What’s the memory content?

Solution

- Data segment:

```
L1      db      0AAh, 0BBh
L2      dw      0CCDDh
L3      db      0EEh, 0FFh
```

L1		L2		L3	
AA	BB	DD	CC	EE	FF

Solution

L1		L2		L3	
AA	BB	DD	CC	EE	FF

```
mov eax, [L2]     ; eax = FF EE CC DD

mov ax, [L3]      ; eax = FF EE FF EE

mov [L1], eax     ; L1 points to EE FF EE FF
```

L1		L2		L3	
EE	FF	EE	FF	EE	FF

Final memory content

Moving immediate values

- Consider the instruction: mov [L], 1
- The assembler will give us an error: "operation size not specified"!
- This is because the assembler has no idea whether we mean for "1" to be 01h, 0001h, 00000001h, etc.
  - Labels have no type (they're NOT variables)
- Therefore the assembler must provide us with a way to specify the size of immediate operands
- mov dword [L], 1
  - 4-byte double-word
- 5 size specifiers: byte, word, dword, qword, tword

Size Specifier Examples

mov	[L1], 1	; Error
mov	byte [L1], 1	; 1 byte
mov	word [L1], 1	; 2 bytes
mov	dword [L1], 1	; 4 bytes
mov	[L1], eax	; 4 bytes
mov	[L1], ax	; 2 bytes
mov	[L1], al	; 1 byte
mov	eax, [L1]	; 4 bytes
mov	ax, [L1]	; 2 bytes
mov	ax, 12	; 2 bytes

Brackets or no Brackets

- mov eax, [L]
  - Puts the content at address L into eax
  - Puts 32 bits of content, because eax is a 32-bit register
- mov eax, L
  - Puts the address L into eax
  - Puts the 32-bit address L into eax
- mov ebx, [eax]
  - Puts the content at address eax (= L) into ebx
- inc eax
  - Increase eax by one
- mov ebx, [eax]
  - Puts the content at address eax (= L + 1) into ebx

Example

```
first       db    00h, 04Fh, 012h, 0A4h
second      dw    165
third       db    "adf"
```

```
mov   eax, first
inc   eax
mov   ebx, [eax]
mov   [second], ebx
mov   byte [third], 11o
```

What is the content of "data" memory after the code executes on a Little Endian Machine?

Example

```
first     db    00h, 04Fh, 012h, 0A4h
second    dw    165
third     db    "adf"
```

```
mov   eax, first
inc   eax
mov   ebx, [eax]
mov   [second], ebx
mov   byte [third], 11o
```

00	4F	12	A4	A5	00	61	64	66
first (4)				second (2)		third (3)		

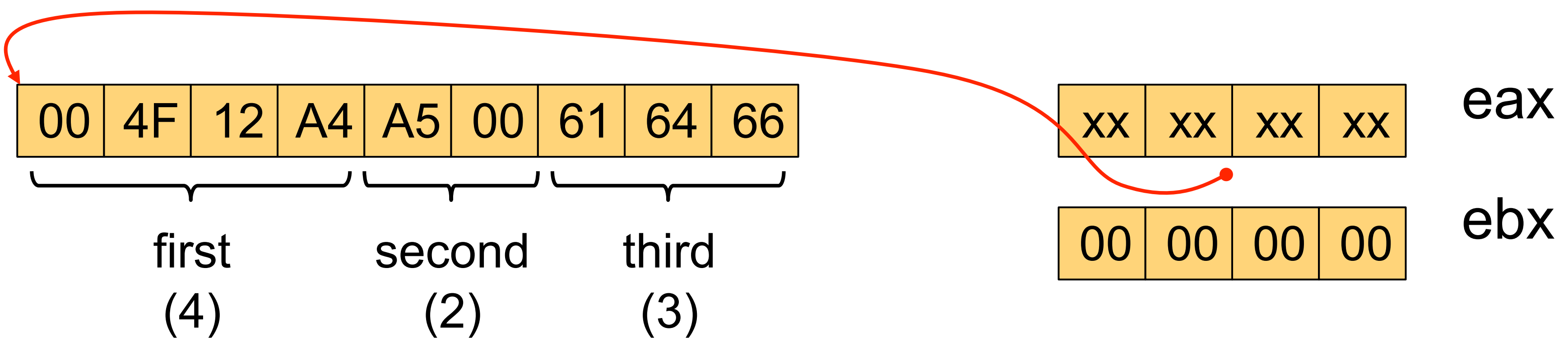
00	00	00	00	eax
00	00	00	00	ebx

Example

```
first       db      00h, 04Fh, 012h, 0A4h
second      dw      165
third       db      "adf"
```

```
mov    eax, first
inc    eax
mov    ebx, [eax]
mov    [second], ebx
mov    byte [third], 11o
```

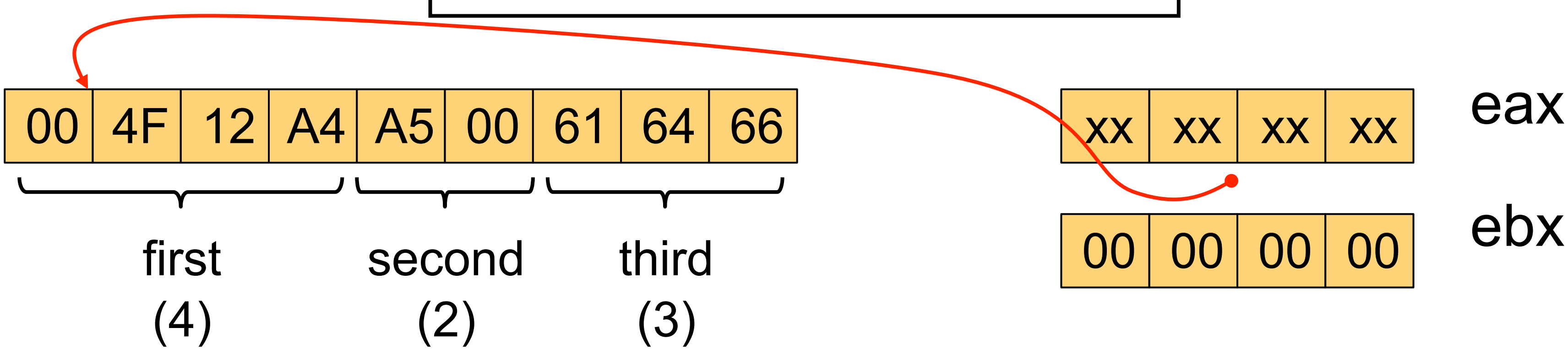
Put an **address** into eax (addresses are 32-bit)



Example

```
first      db    00h, 04Fh, 012h, 0A4h
second     dw    165
third      db    "adf"
```

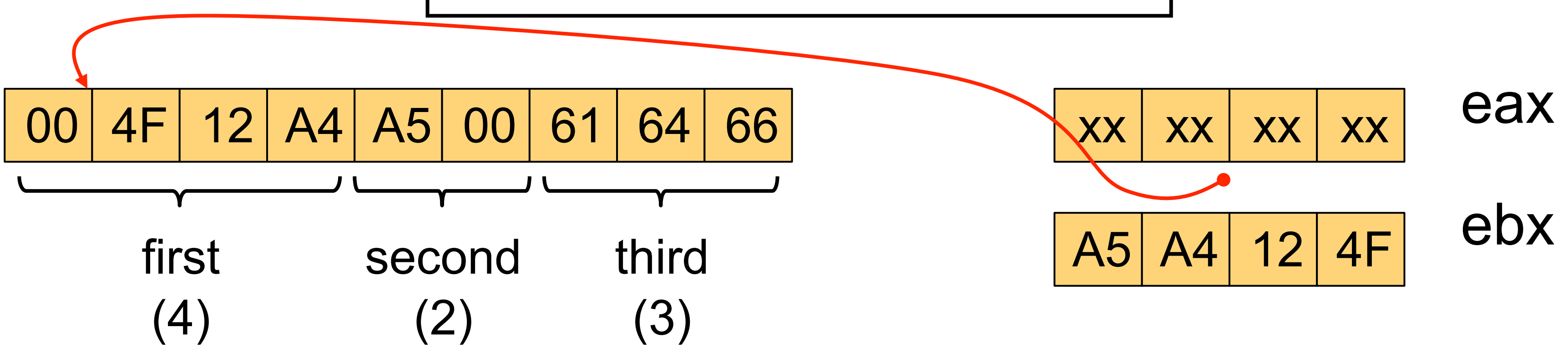
```
mov   eax, first
inc   eax
mov   ebx, [eax]
mov   [second], ebx
mov   byte [third], 11o
```



Example

```
first     db    00h, 04Fh, 012h, 0A4h
second    dw    165
third     db    "adf"
```

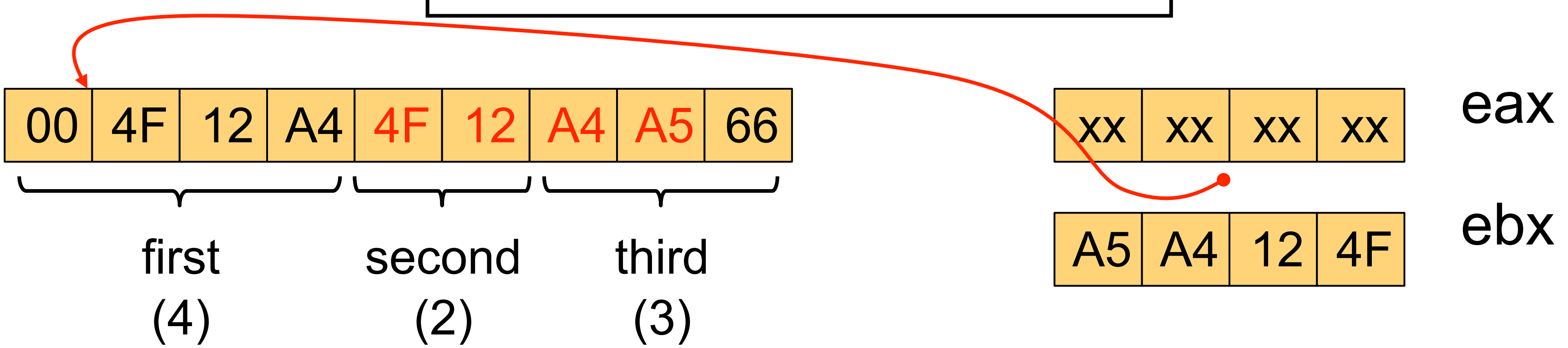
```
mov   eax, first
inc   eax
mov   ebx, [eax]
mov   [second], ebx
mov   byte [third], 11o
```



Example

```
first     db    00h, 04Fh, 012h, 0A4h
second    dw    165
third     db    "adf"
```

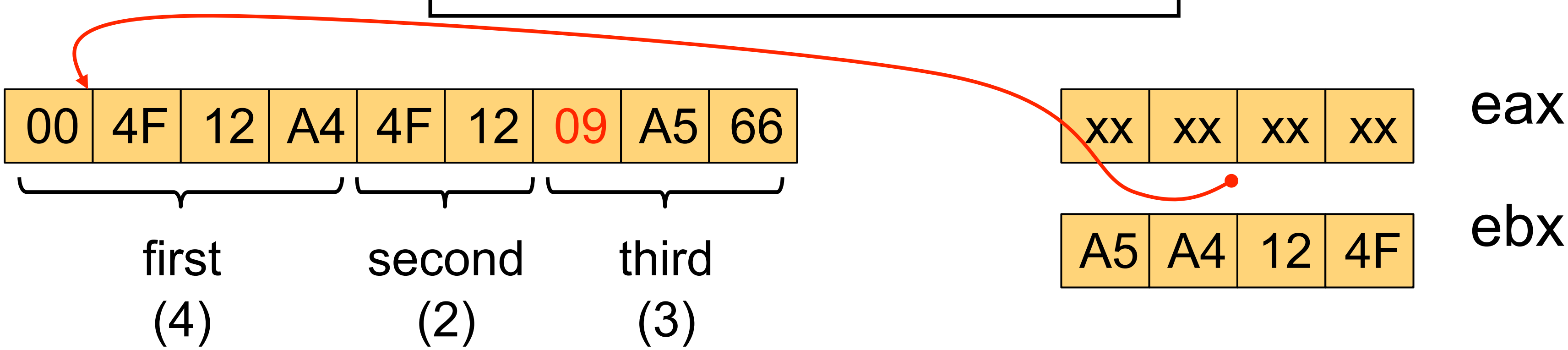
```
mov   eax, first
inc   eax
mov   ebx, [eax]
mov   [second], ebx
mov   byte [third], 11o
```



Example

```
first       db    00h, 04Fh, 012h, 0A4h
second      dw    165
third       db    "adf"
```

```
mov   eax, first
inc   eax
mov   ebx, [eax]
mov   [second], ebx
mov   byte [third], 11o
```

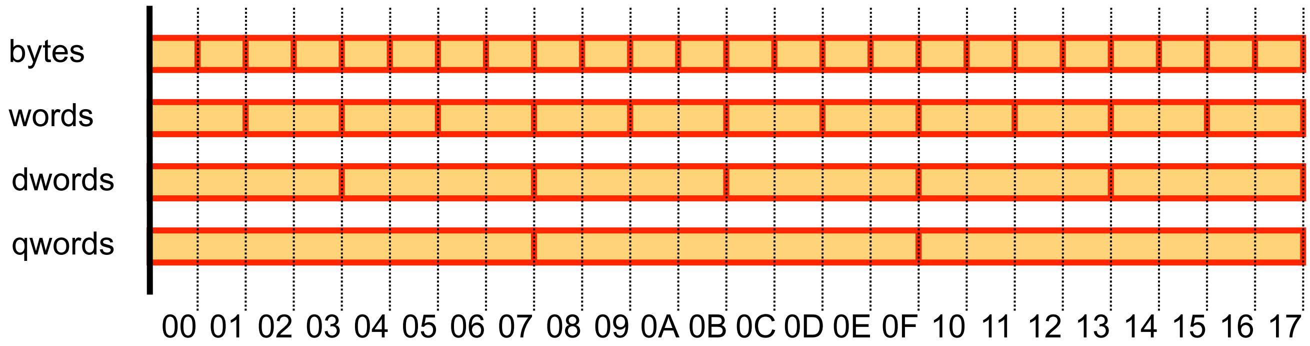


Assembly is Dangerous

- The previous example is really a terrible program
- But it's a good demonstration of why the assembly programmer must be really careful
- For instance, we were able to store 4 bytes into a 2-byte label, thus overwriting the first 2 characters of a string that merely happened to be stored in memory next to that 2-byte label
- Playing such tricks can lead to very clever programs that do things that would be impossible (or very cumbersome) to do with many high-level programming language (e.g., in Java)
- But you really must know what you're doing
- Typically such behaviors are bugs

x86 Assembly is Dangerous

- Another dangerous thing we did in our assembly program was the use of unaligned memory accesses
  - We stored a 4-byte quantity at some address
  - We incremented the address by 1
  - We read a 4-byte quantity from the incremented address!
  - This really removes all notion of a structured memory
- Some architectures only allow aligned accesses
  - Accessing an X-byte quantity can only be done for an address that's a multiple of X!



Conclusion

- It’s important to understand the memory layout