NASM Basics III Using Registers and RAM

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Indirection

- In C, indirection is done with the * operator

```
int *var;   // var is an integer that is
            // the address of some byte in RAM
*var = 2;   // *var is the value at address var
```

- In assembly, indirection is done with [ ]

```
[eax]     ; if eax contains an integer that
          ; is the 32-bit address of some byte
          ; in RAM, then [eax] is the value
          ; at that address.

[ax]      ; Invalid since ax is a 16-bit integer,
          ; and addresses are 32 bits!
```

Memory Reference Operands

- Remember that we had said that instructions can take **operands that are memory locations**
- This is done using the [ ] brackets, for instance:
  - `add eax, [ebx]`
    - eax = eax + **4-byte** content in RAM, where the address of the first byte is the value of ebx (we often will say “at address ebx”)
  - `mov [ecx], dx`
    - Write to RAM, at address ecx, the **2-byte** value in dx (the first byte will be written at address bx, the second byte at address bx + 1)
  - `mov [L1], bh`
    - Write to RAM, at address L1, the **1-byte** value in bh

- In all the above, it’s easy to know how many bytes are read/written because one of the operands is a register
- But what if none of the operands is a register?

Data Size Specifiers

- Say we write in our program: `mov [eax], 12`
- This is *ambiguous*: Do we mean a 1-byte value, a 2-byte value, or a 4-byte value?
- The assembler (in our case NASM) will actually throw an error message that says "operation size not specified"
- We need to specify the data size:
  - `mov byte  [eax], 12   ; writes 0C to RAM`
  - `mov word  [eax], 12   ; writes 000C to RAM`
  - `mov dword [eax], 12   ; writes 0000000C to RAM`
  - `add word  [ebx], 12   ; performs a 2-byte add`
- It's commonplace to forget the size specifier, but since the assembler complains about it, we never run the risk of leaving it ambiguous in our programs

At Most One Memory Operand

- At most one of the operands to an instruction can be a memory location
  - `mov eax, [ebx]                ; OK`
  - `mov [eax], ebx                ; OK`
  - `mov [eax], [ebx]              ; NOT OK`
  - `add dword [eax], 12           ; OK`
  - `add dword [eax], [ebx]        ; NOT OK`

- So if we need, for instance, to copy a 4-byte value from one memory location to another, we have to use 2 instructions and a register:

```
~~mov dword [L2], [L1]~~ ; forbidden
; instead do it in two steps, "wasting" a register
mov edx, [L1]          ; read 4 bytes from RAM
mov [L2], edx          ; write them back to RAM
```

Use of Labels

- In the previous slide, we had things like [L1]
- This makes sense because L1 is an address, not a value
- Therefore, a common use of the label in the code is as a memory operand, in between square brackets **‘[‘ ‘]’**
- LABELS HAVE NO TYPE!
  - It’s tempting to think of them as variables, but they are much more limited: just the address of a byte somewhere
- So, **regardless of how a label was defined**, we can do:
  - `mov al, [L1]` ; a 1-byte copy
  - `mov ax, [L1]` ; a 2-byte copy
  - `mov eax,[L1]` ; a 4-byte copy
- Just to make sures it’s clear, let’s see an example

Labels have NO TYPE

- Say we have the following data segment

```
L    db   0F0h, 0F1h, 0F2h, 0F3h
```

- It seems that the programmer means this as 4-element array of 1-byte values
- But if we do: `mov ax, [L]`
- Then, ax = F1 F0
- That is, although we declared 1-byte values, here we "glue" two of them as a 2-byte value
  - Something that high-level languages often prohibit
- The only thing that matters is what bytes are in RAM, and that some of them have addresses for which we have symbolic names (like L1)
- In fact, there are many equivalent declarations…

Labels have NO TYPE

```
L1    db    0F0h, 0F1h, 0F2h, 0F3h
```

```
L1    dw    0F1F0h, 0F3F2h
```

```
L1    dd    0F3F2F1F0h
```

- The above three declarations are THE SAME
  - They define the exact same 4 consecutive byte values in RAM: F0 F1 F2 F3
  - The address of the first byte is L1
- Each way of writing it may give us some guess about the programmer's intent, but that's it
  - And the programmer could be purposely cryptic

Register-Order Values in Programs

- In the data segment declarations and the code, all immediate values (numerical constants) are written in register order (when written in hex, binary, octal)
  - This should have been obvious all along, but just in case
- Consider the following data segment declaration

```
L1    dd    0AABBCCDDh
```

  - The instruction `mov eax,[L1]` would put AABBCCDD into eax
  - Because the memory content was DDCCBBAA!
- Consider the following instruction:

```
add  eax, 00001h
```

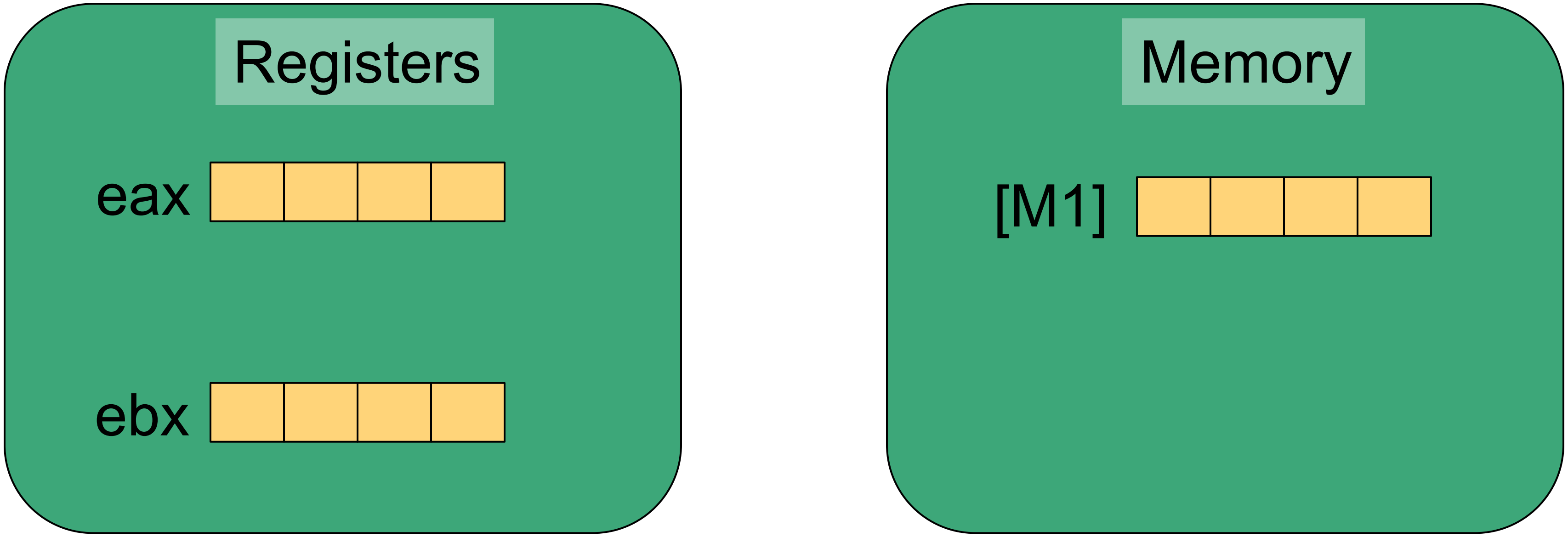
  - The above adds 1 to eax, and **not** 2^8 (i.e., 0100 in hex)

- It would be really confusing to write numbers in (little endian) memory order in the program

Little Endian

- Now that we know how to have memory locations as operands, we can see the Little Endian behavior in assembly

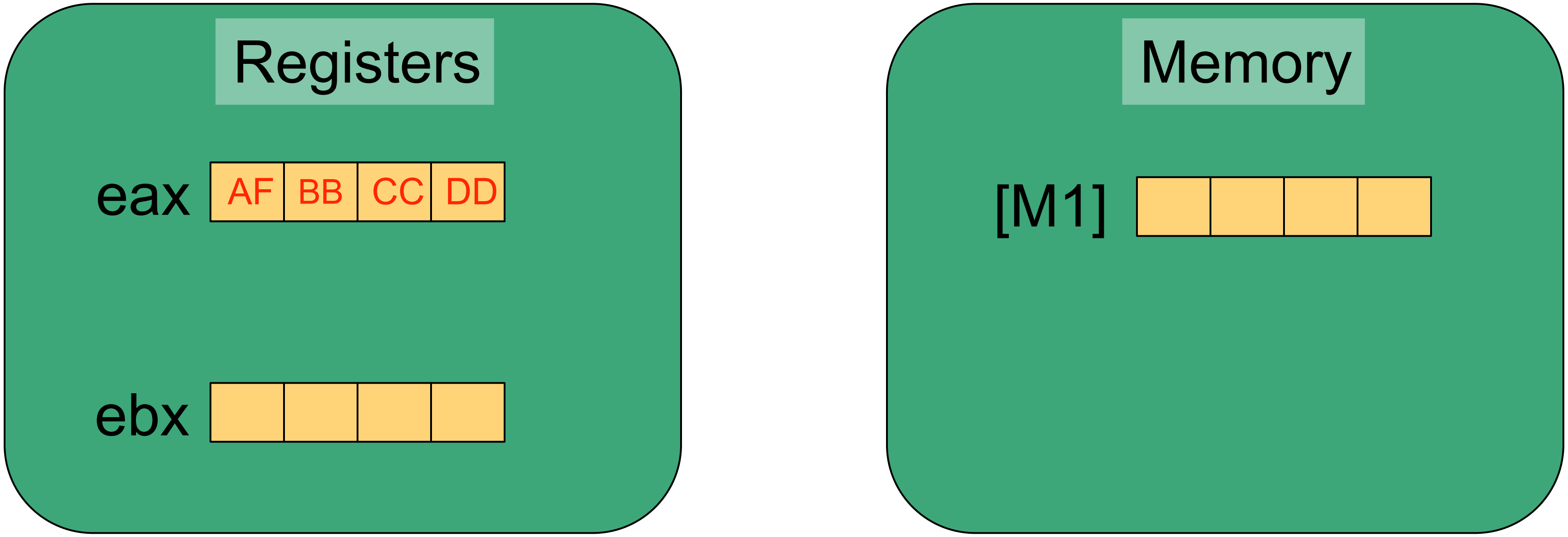
```
mov eax, 0AFBBCCDDh     ; sets value of register EAX
mov [M1], eax           ; copy EAX's value to RAM
mov ebx, [M1]           ; copy value from RAM to EBX
```



Little Endian

- Now that we know how to have memory locations as operands, we can see the Little Endian behavior in assembly

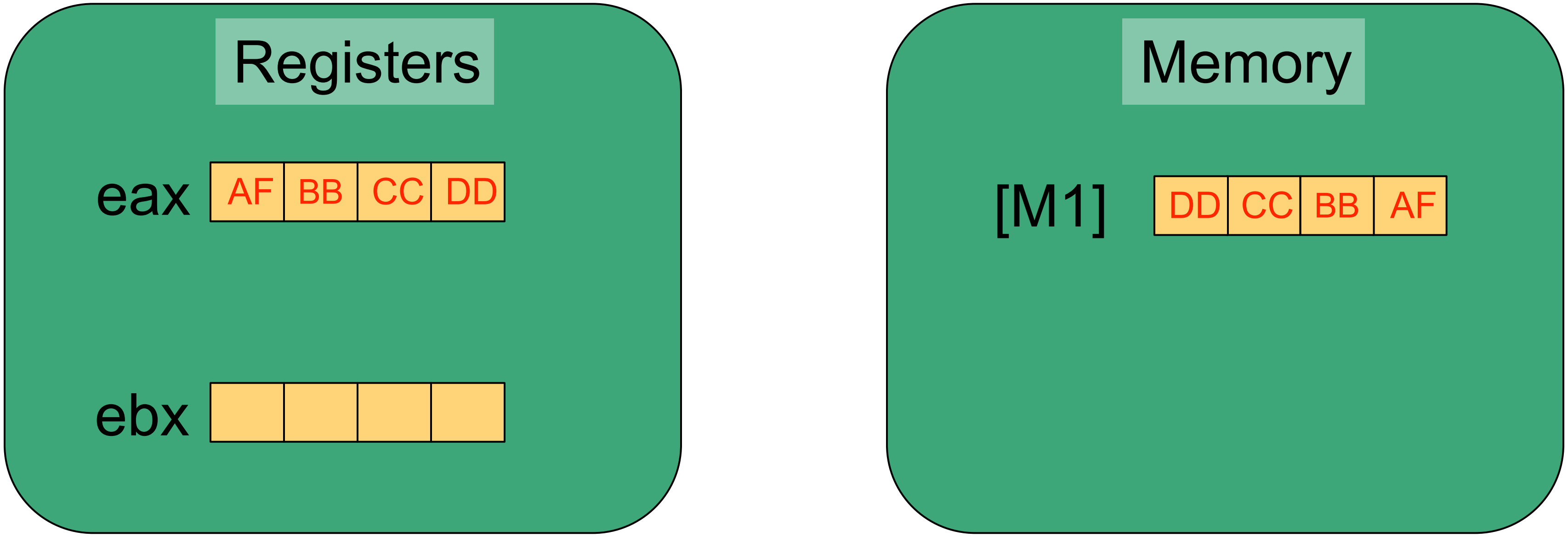
```
mov eax, 0AFBBCCDDh      ; sets value of register EAX
mov [M1], eax            ; copy EAX's value to RAM
mov ebx, [M1]            ; copy value from RAM to EBX
```



Little Endian

- Now that we know how to have memory locations as operands, we can see the Little Endian behavior in assembly

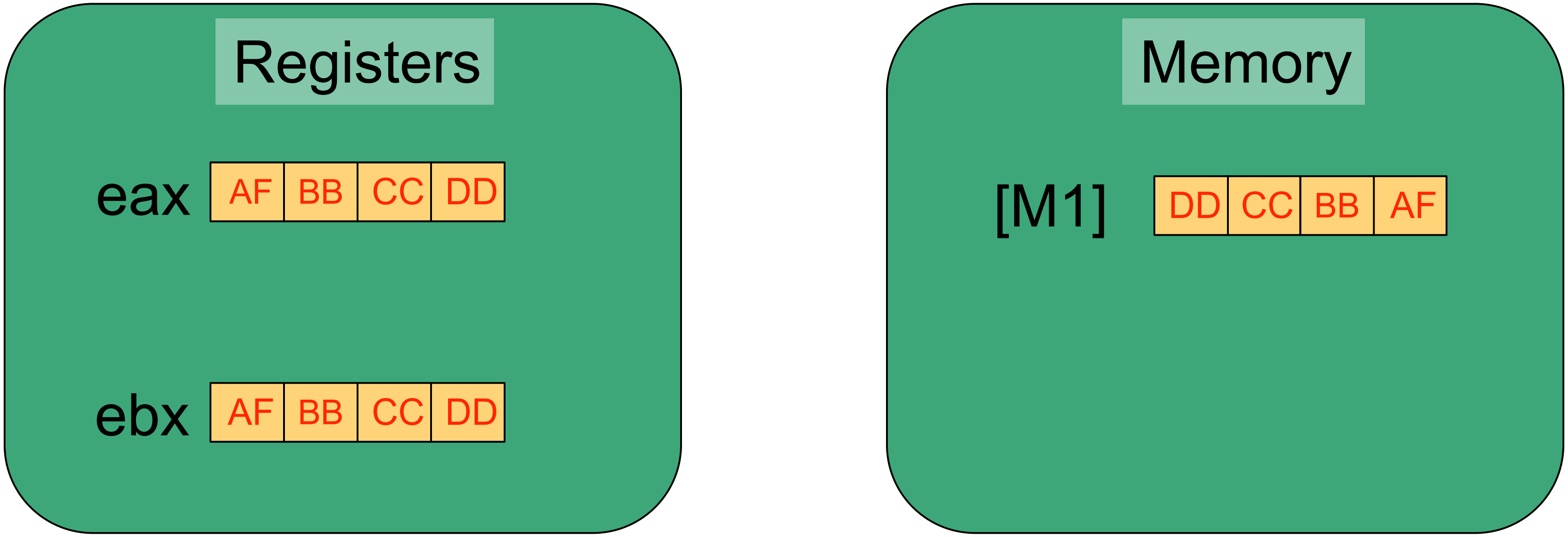
```
mov eax, 0AFBBCCDDh     ; sets value of register EAX
mov [M1], eax           ; copy EAX's value to RAM
mov ebx, [M1]           ; copy value from RAM to EBX
```



Little Endian

- Now that we know how to have memory locations as operands, we can see the Little Endian behavior in assembly

```
mov eax, 0AFBBCCDDh    ; sets value of register EAX
mov [M1], eax          ; copy EAX's value to RAM
mov ebx, [M1]          ; copy value from RAM to EBX
```



Example

- Data segment (little endian):

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

- Program:

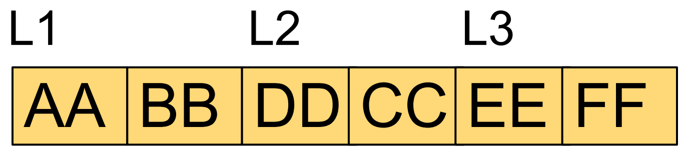
```
mov eax, [L2]
mov ax, [L3]
mov [L1], eax
```

- What’s the final memory content?

Solution (1)

- Data segment (little endian):

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



Solution (2)

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 ; write EE FF EE FF in RAM
```

L1		L2		L3	
EE	FF	EE	FF	EE	FF

Final memory content

Brackets or no Brackets

- mov eax, [L]
  - Copies the content at address L into eax
  - Copies 32 bits of content, because eax is a 32-bit register
- mov eax, L
  - Copies the 32-bit address L into eax
  - eax now contains a number that happens to be an address (we call that a pointer!)
- mov ebx, [eax]
  - Copies the content at the address whose value is stored in eax into ebx
  - In this example, given the above instructions, eax = L
- inc eax
  - Increase eax by one (so now eax = L + 1, given the above instructions)
- mov ebx, [eax]
  - Copies the content at the address whose value is stored in eax (= L + 1 in this example so far) into ebx

Indirection with an Offset

- You can add/subtract a constant offset to the address insides the [ ]

```
mov     eax, L
add     eax, 2
mov     dword [eax], 42
```

```
mov     eax, L
mov     dword [eax+2], 42
```

```
mov     dword [L+2], 42
```

```
mov     dword [M-3], 42
```

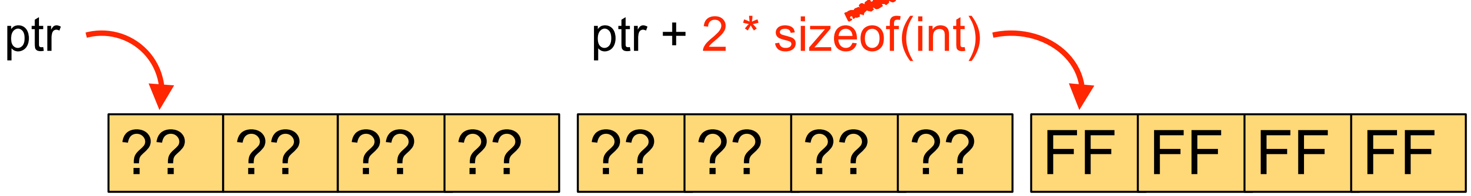
Indirection with an Offset

- In assembly when we say “+2” to an address it’s necessarily adding 2 to the address to “jump over” 2 bytes
  - Because we do not have a notion of data types!
- High-level languages that support pointers (C, C++, Rust, etc) however, try to be helpful because we declared data types in our programs!
- This creates quite a bit of confusion when learning assembly programming **after** learning high-level programming
- So let’s remove that confusion right now with a simple example…

Low-/High-Level Indirection

```
int *ptr;
*(ptr + 2) = -1;
```

The compiler is “helping”: **`ptr`** is a pointer to 4-byte values, so when the user wrote “+2” they really mean “jump over the next two elements” **not** “jump over the next two bytes”




```
mov eax, ptr;
mov dword [eax + 8], -1
```

In assembly we don’t have data types: the only thing we can “talk about” are bytes. So to skip over two 4-byte elements, we have to do +8, **not** +2

Low-level-like High-level Code

```
int *ptr;
*(ptr + 2) = -1;
```

```
int *ptr;
*((int *)((char *)ptr + 8)) = -1;
```

- These two code fragments do the **exact same thing**
  - By casting the `int*` pointer to a `char*` pointer, we “tell” the compiler that `ptr` is now a pointer to 1-byte elements
  - So when we do +8, that means skip over 8 1-byte values
  - Then we cast the pointer back to in `int*` pointer
  - So that we write a 4-byte value (FF FF FF FF) at that address

“Big” Example

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

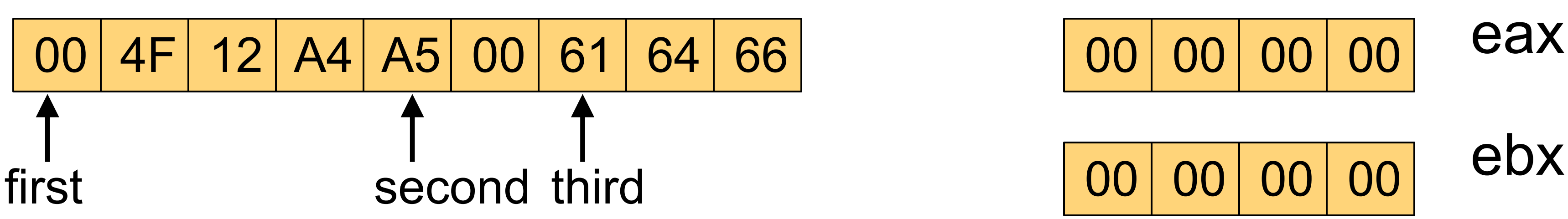
```
mov   eax, first
inc   eax
mov   ebx, [eax]
mov   [second], ebx
mov   byte [third], 11o
```

What is the content of the data segment after the code executes on a **Little Endian** Machine?

"Big" 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
```

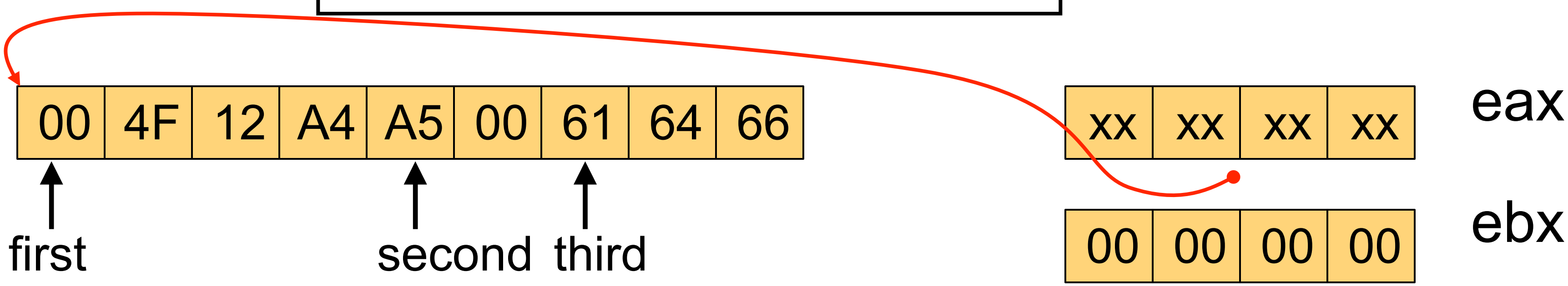


"Big" Example

```
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 (this works because our addresses are 32-bit and thus fit into 4-byte registers, just like any other 4-byte values!)

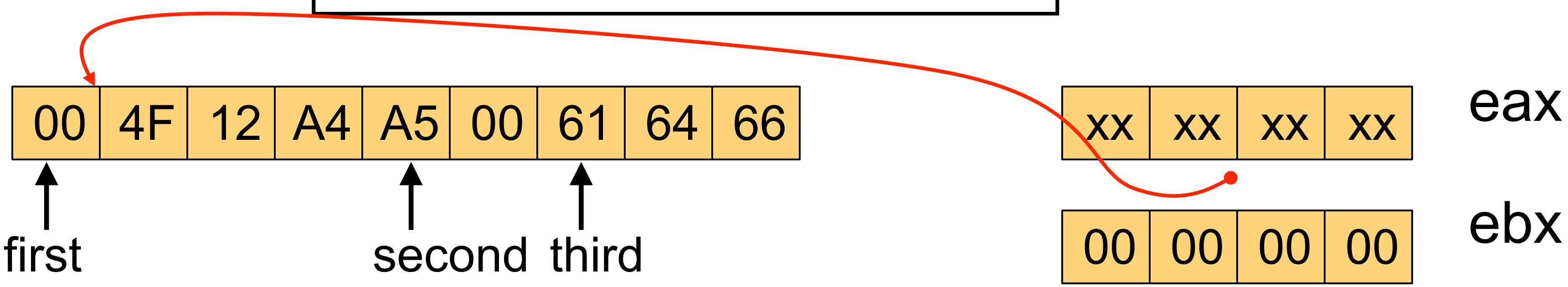


"Big" Example

```
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
```

Increment that address by 1, thus now pointing to the next byte

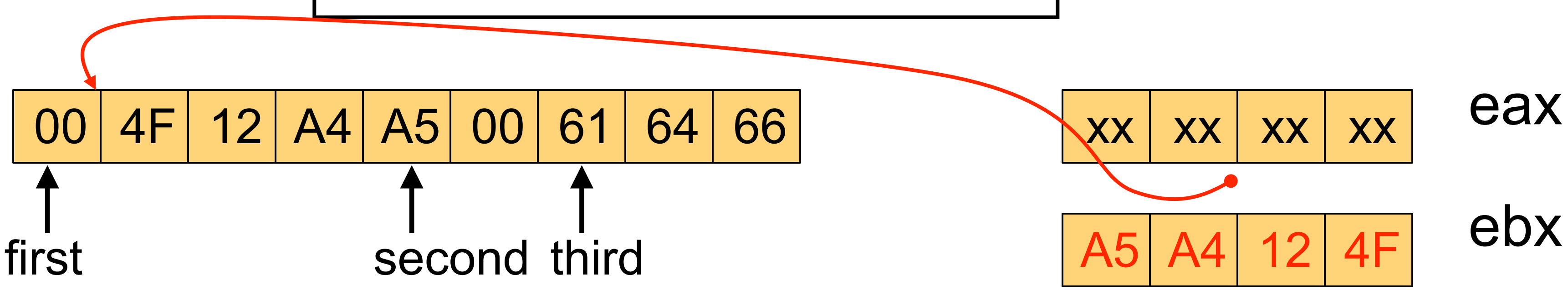


"Big" Example

```
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 the 4 bytes at that address into ebx (note the Little Endian)

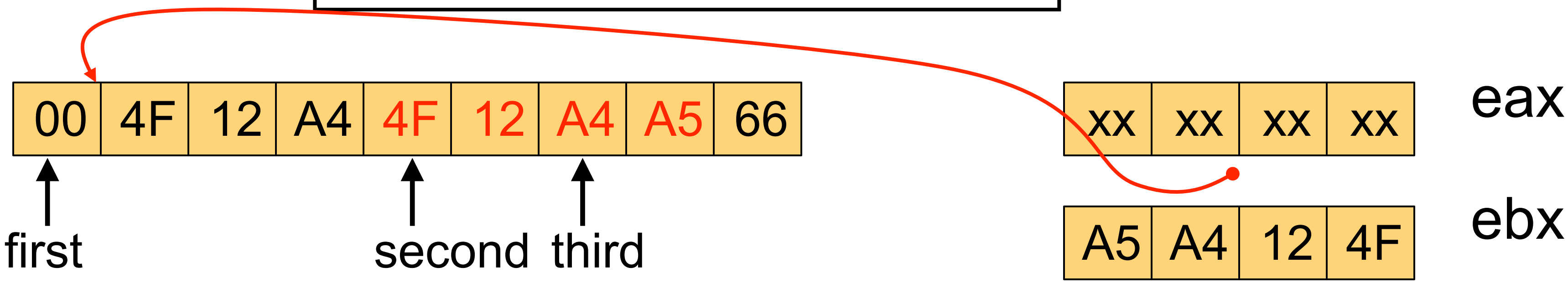


"Big" Example

```
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
```

Copy 4 bytes to memory at address **second**

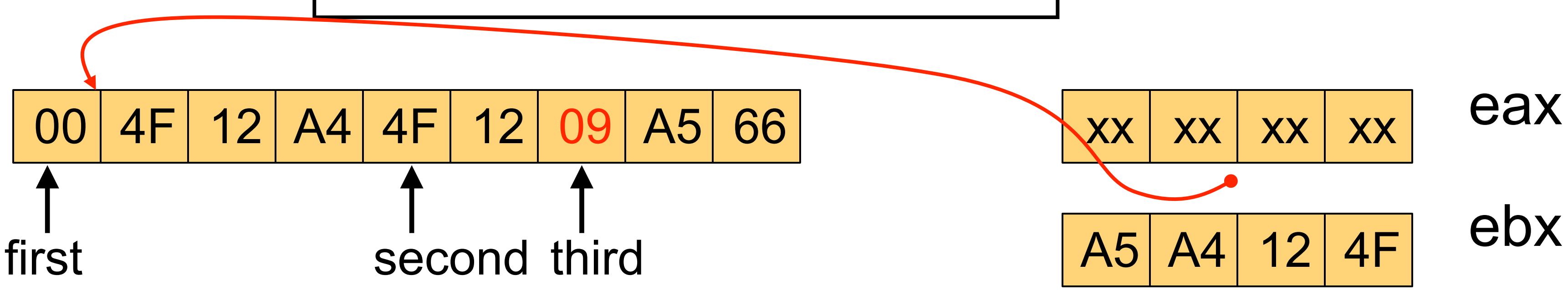


"Big" Example

```
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
```

Write 1 byte at address **third**



Label values

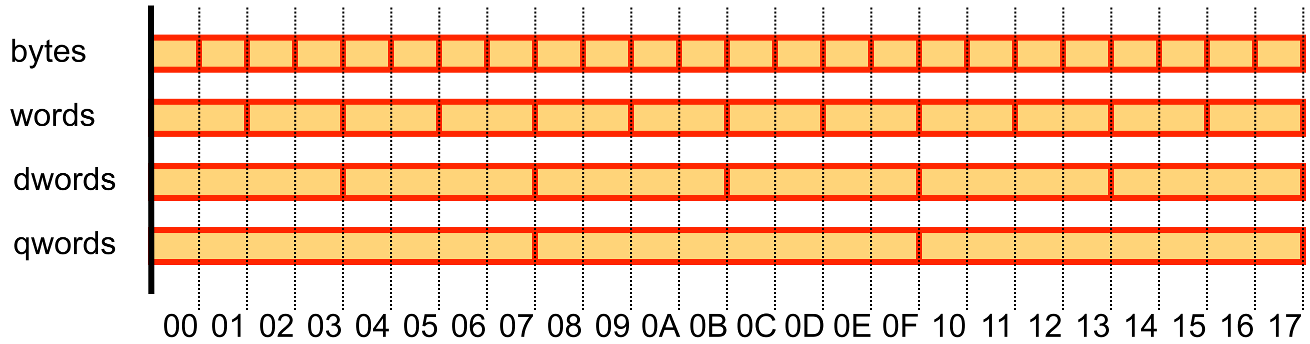
- Note that after the instruction **`mov eax, first`**, I didn't show a value for eax but just the classical "pointer arrow" to the correct byte
- This is because although at runtime the label **`first`** will actually have some numerical value, we don't know what it will be without running the program
- I could have added to the example something like: "oh, and by the way, first = FF FF 12 38"
- Then, we could have known the numerical value of all the addresses that the program manipulates
  - Instead of saying "the address of the 2nd byte" or instead of drawing some pointer arrow, we could have just said FF FF 12 39

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
  - again: LABELS ARE NOT VARIABLES AT ALL
- 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, which you will have, which is why we're doing all this
- Let's try to reproduce that behavior in C, which is done by doing "casting of pointers" as in a few slides ago…

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 (it's only bytes)
- Some architectures (not x86) only allow aligned accesses
  - Accessing an X-byte quantity can only be done for an address that's a multiple of X!



Important Takeaways

- Indirection with the [ ] bracket NASM syntax
  - At most one set of brackets for operands to an instruction
- The need to specify data size when ambiguous
- The fact that labels are not variables, because they have no types
- The difference between an address offset in low-level assembly and in high-level code
- The “danger” / “power” of being able to dereference any address willy-nilly

Conclusion

- Some of the programs we've seen are horrible, and you're thinking "I'll never do that"
- But, you will have bugs and your code will do horrible stuff like that even though you don't mean it to
- So you need to be able to trace/understand such behaviors for debugging purposes
  - And also to reverse-engineer code that's just clever and exploits the "danger" aspect of assembly
- Let's do some of the practice problems for these lecture notes…
- We also have a sample homework assignment

- Next week we'll have **an in-class quiz on this module**
- Our **first midterm** will about about this content
  - Date to be announced (midterm will be after we're done with the next module)