



# **NASM: data and bss**

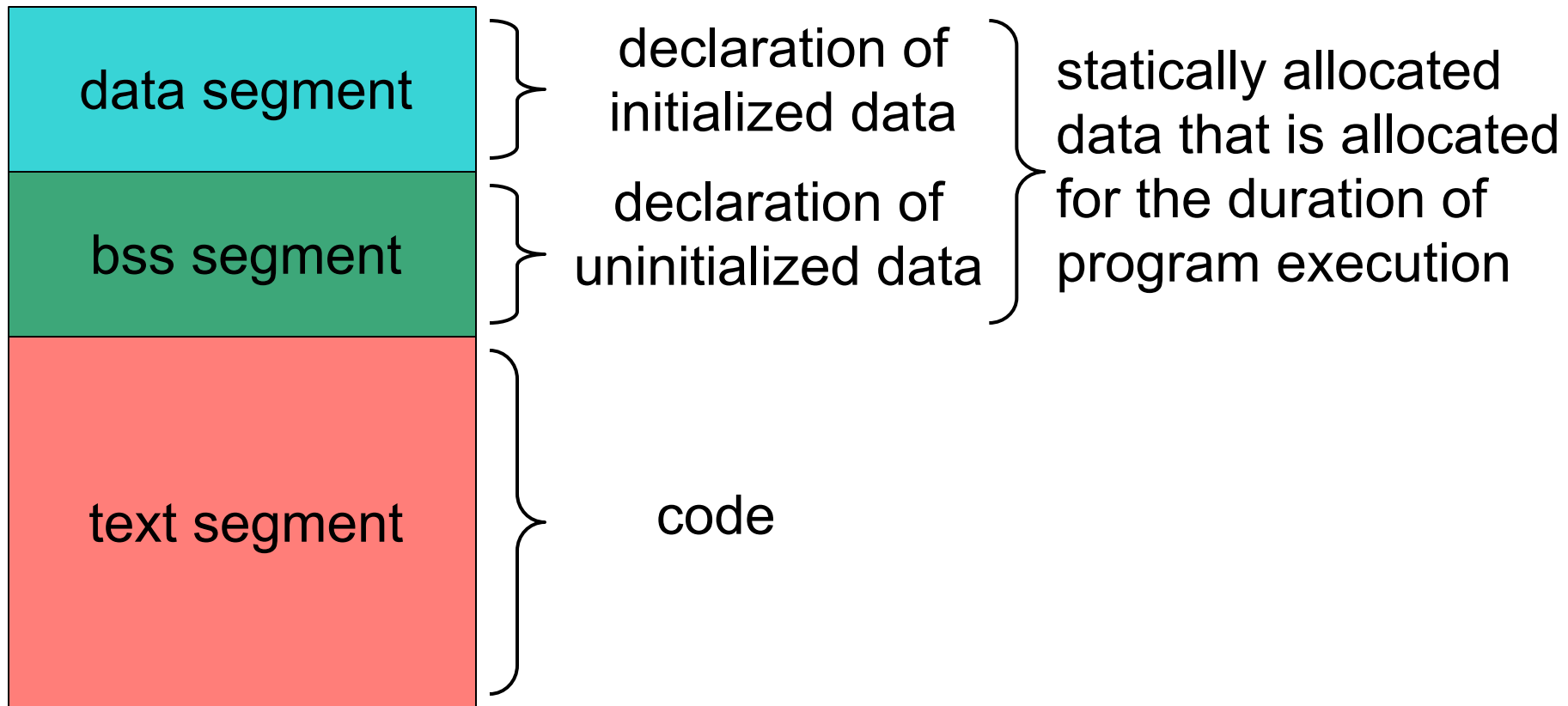
## **Beyond the screencast**

**ICS312**

**Machine-Level and  
Systems Programming**

Henri Casanova ([henric@hawaii.edu](mailto:henric@hawaii.edu))

# NASM Program Structure



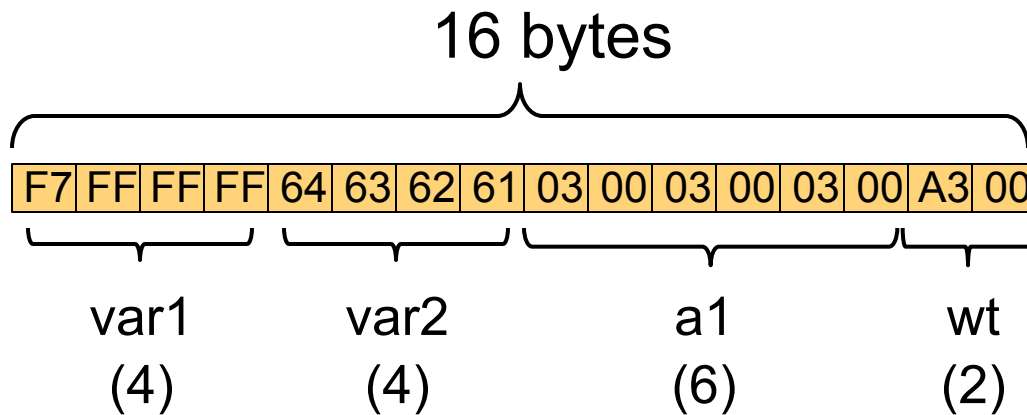
# Practice #1

```
var1      dd      -9
var2      db      "dcba"
a1        times 3    dw      011b
wt        db      0A3h, 0
```

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

# Practice #1

```
var1      dd      -9
var2      db      "dcba"
a1        times 3   dw      011b
wt        db      0A3h, 0
```



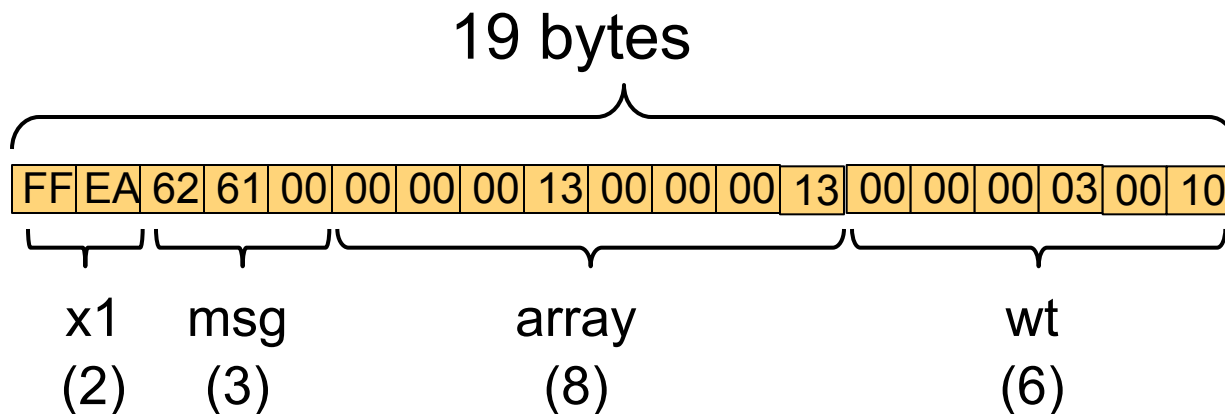
# Practice #2

```
x1          dw      -22
msg         db      "ba", 0
array       times 2      dd      023o
wt          dw      0,011b,020o
```

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

# Practice #2

```
x1          dw      -22
msg         db      "ba", 0
array       times 2   dd      0230
wt          dw      0,011b,0200
```





# More practice?

- Of course we can easily come up with tones of practice examples...
- Should we do one more right now?



# Homework Assignment #2

- Homework Assignment #2 is posted
- With what we've seen you can get started on Exercise #1
- For Exercise #2 we need to go further a bit...
  - We might do it all today





# 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

# Different kinds of 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  $\text{eax} = \text{eax} + \text{ebx}$
- **Memory**: specifies an address in memory.
  - **add eax, [ebx]**
  - means  $\text{eax} = \text{eax} + \text{content (4 bytes) of memory at address ebx}$
- **Immediate**: specifies a fixed value (i.e., a number)
  - **add eax, 2**
  - means  $\text{eax} = \text{eax} + 2$
- **Implied**: not actually encoded in the instruction
  - **inc eax**
  - means  $\text{eax} = \text{eax} + 1$

# 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]`
  - Copy **the 1-byte value 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
  - In the above example, since AL is a 1-byte register, then 1 byte is moved
- **LABELS HAVE NO TYPE!**
  - So although it's tempting to think of them as variables, they are much more limited: just pointers to a byte somewhere

# Moving to/from a register

- Say we have the following data segment

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

- Example: `mov AL, [L]`

- Will copy 1 byte from memory into AL

- Example: `mov AX, [L]`

- Will copy the 2 bytes from memory into AX

- Example: `mov EAX, [L]`

- Will copy the 4 bytes from memory into EAX

- What bytes are written where depends on Little vs. Big Endian...

# Mov and Little Endian

- Consider the following data segment

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

L2      dd      0AABBCDDh

- 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 AABBCDD 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 (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?



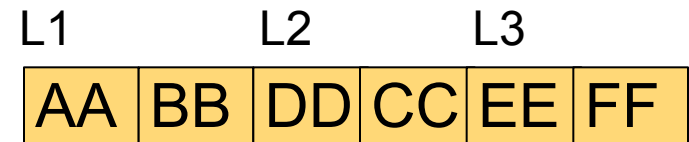
# Solution

## ■ Data segment:

L1     db     0AAh, 0BBh

L2     dw     0CCDDh

L3     db     0EEh, 0FFh



# 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 ; content at L1 is 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
- **Size specifiers**: byte, word, dword, qword (and 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

<code>first</code>	<code>db</code>	<code>00h, 04Fh, 012h, 0A4h</code>
<code>second</code>	<code>dw</code>	<code>165</code>
<code>third</code>	<code>db</code>	<code>"adf"</code>

<code>mov</code>	<code>eax, first</code>
<code>inc</code>	<code>eax</code>
<code>mov</code>	<code>ebx, [eax]</code>
<code>mov</code>	<code>[second], ebx</code>
<code>mov</code>	<code>byte [third], 11o</code>

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

# Example

<code>first</code>	<code>db</code>	<code>00h, 04Fh, 012h, 0A4h</code>
<code>second</code>	<code>dw</code>	<code>165</code>
<code>third</code>	<code>db</code>	<code>"adf"</code>

<code>mov</code>	<code>eax, first</code>
<code>inc</code>	<code>eax</code>
<code>mov</code>	<code>ebx, [eax]</code>
<code>mov</code>	<code>[second], ebx</code>
<code>mov</code>	<code>byte [third], 11o</code>

00	4F	12	A4	A5	00	61	64	66
----	----	----	----	----	----	----	----	----

first

(4)

second

(2)

third

(3)

00	00	00	00
----	----	----	----

eax

00	00	00	00
----	----	----	----

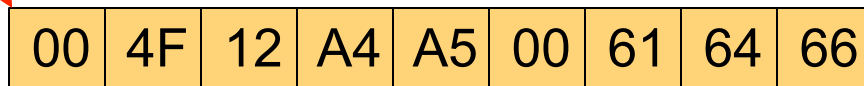
ebx

# 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], 110
```

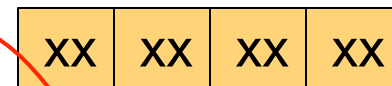
Put an **address** into **eax**  
(this works because  
our addresses are 32-bit  
and thus fit into 4-byte  
registers, just like any  
other 4-byte values!)



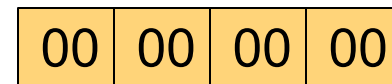
first  
(4)

second  
(2)

third  
(3)



eax



ebx

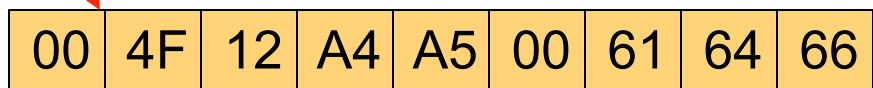


# Example

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], 110
```

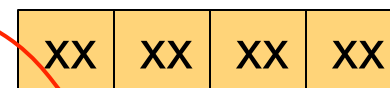
Increment that address  
by 1, thus now pointing  
to the next byte



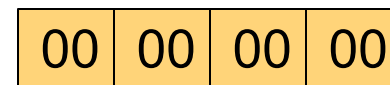
first  
(4)

second  
(2)

third  
(3)



eax



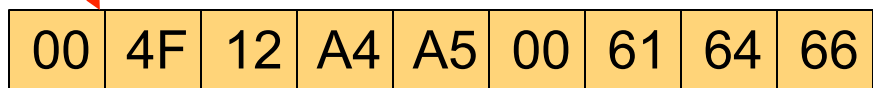
ebx

# 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], 110

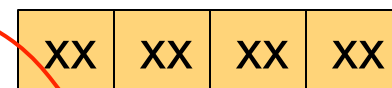
Put the 4 bytes at  
that address into ebx  
(note the Little Endian)



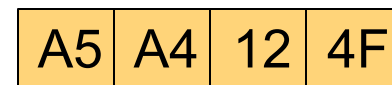
first  
(4)

second  
(2)

third  
(3)



eax



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

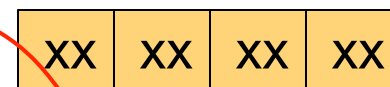
Copy 4 bytes to memory  
at address **second**



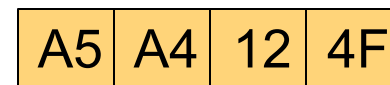
first  
(4)

second  
(2)

third  
(3)



eax



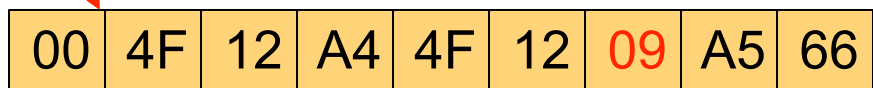
ebx

# 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], 110
```

Write 1 byte at address  
**third**



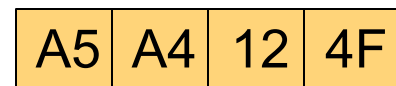
first  
(4)

second  
(2)

third  
(3)



eax



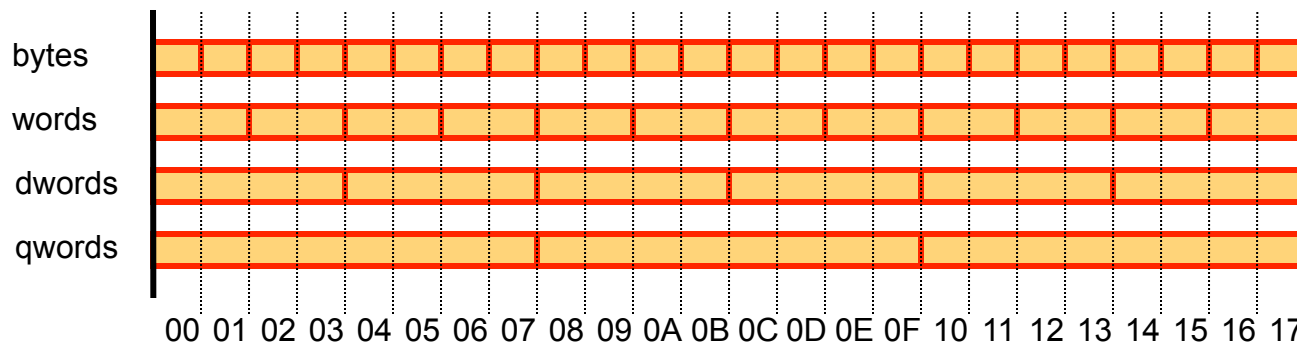
ebx

# 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

# 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 only allow aligned accesses
  - Accessing an X-byte quantity can only be done for an address that's a multiple of X!



# Practice #3

- Consider the following program

var1	dd	179
var2	db	0A3h, 017h, 012h
var3	db	"bca"

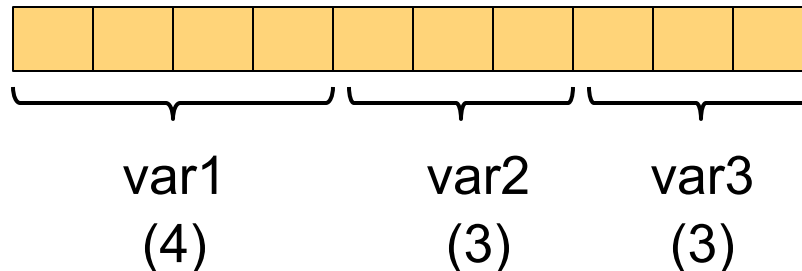
mov	eax, var1
add	eax, 3
mov	ebx, [eax]
add	ebx, 5
mov	[var1], ebx

- What is the final layout of the data segment starting at address var1 on a **Little Endian** Machine?

# Practice #3

var1	dd	179
var2	db	0A3h, 017h, 012h
var3	db	"bca"

mov	eax, var1
add	eax, 3
mov	ebx, [eax]
add	ebx, 5
mov	[var1], ebx

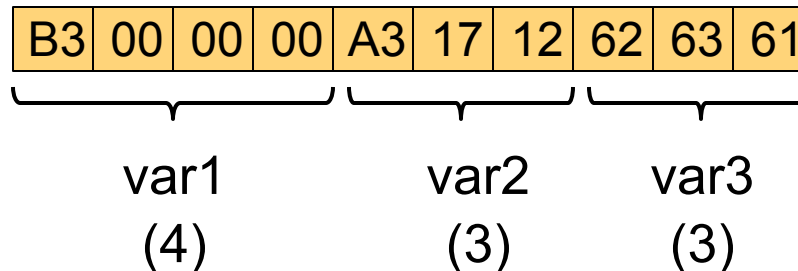




# Practice #3

var1	dd	179
var2	db	0A3h, 017h, 012h
var3	db	"bca"

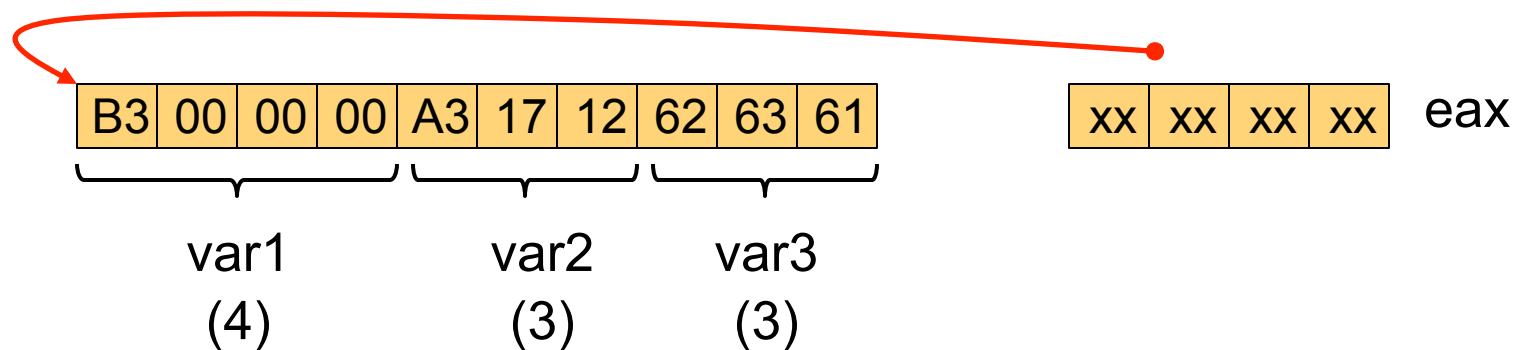
```
mov    eax, var1
add    eax, 3
mov    ebx, [eax]
add    ebx, 5
mov    [var1], ebx
```



# Practice #3

var1	dd	179
var2	db	0A3h, 017h, 012h
var3	db	"bca"

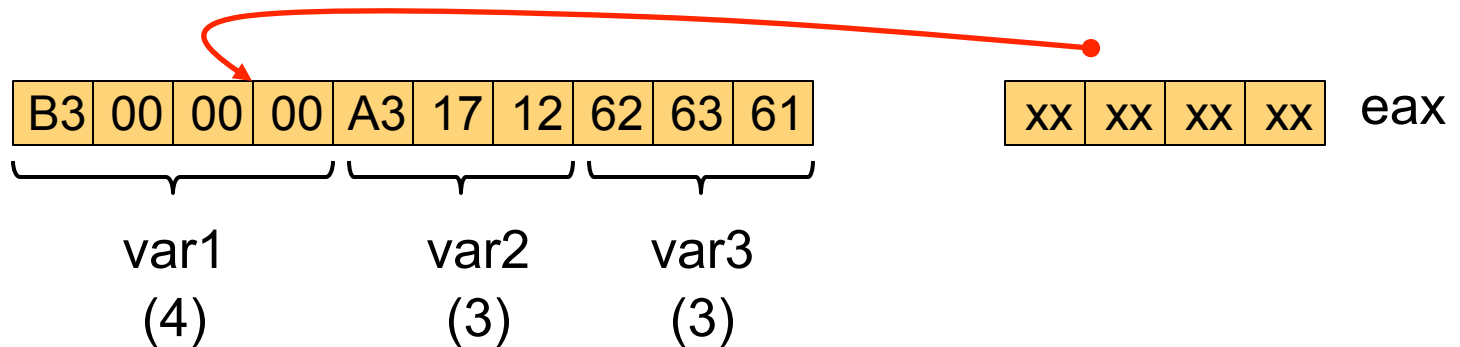
```
mov    eax, var1
add     eax, 3
mov     ebx, [eax]
add     ebx, 5
mov     [var1], ebx
```



# Practice #3

var1	dd	179
var2	db	0A3h, 017h, 012h
var3	db	"bca"

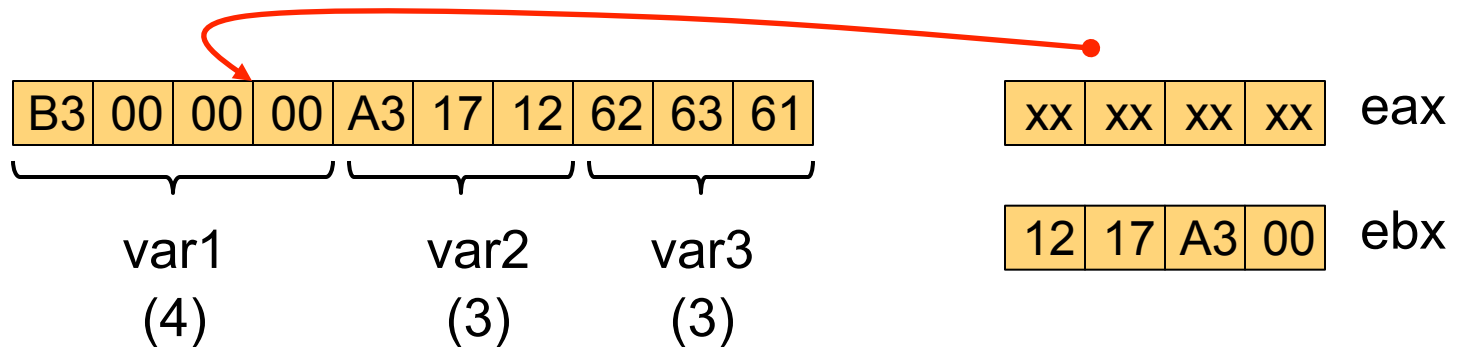
```
mov    eax, var1
add    eax, 3
mov    ebx, [eax]
add    ebx, 5
mov    [var1], ebx
```



# Practice #3

var1	dd	179
var2	db	0A3h, 017h, 012h
var3	db	"bca"

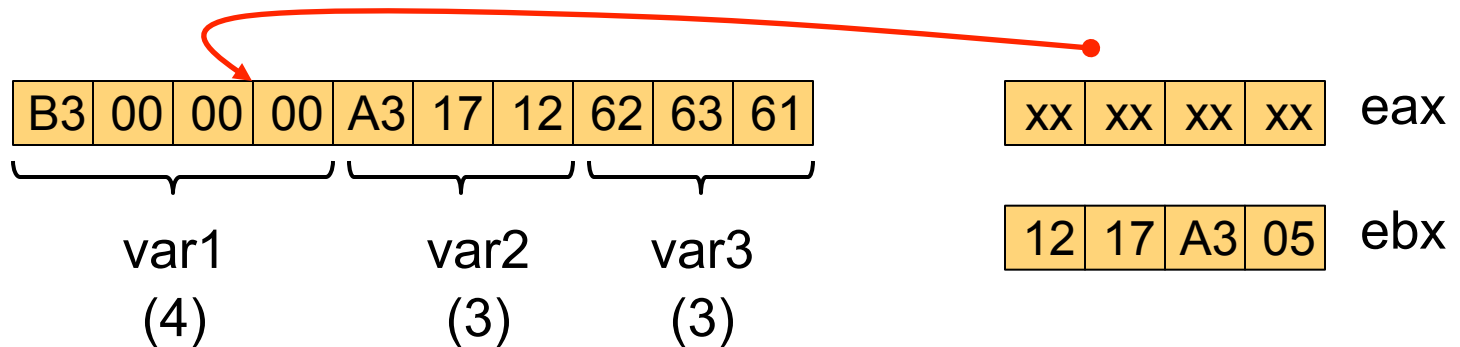
```
mov    eax, var1
add    eax, 3
mov    ebx, [eax]
add    ebx, 5
mov    [var1], ebx
```



# Practice #3

var1	dd	179
var2	db	0A3h, 017h, 012h
var3	db	"bca"

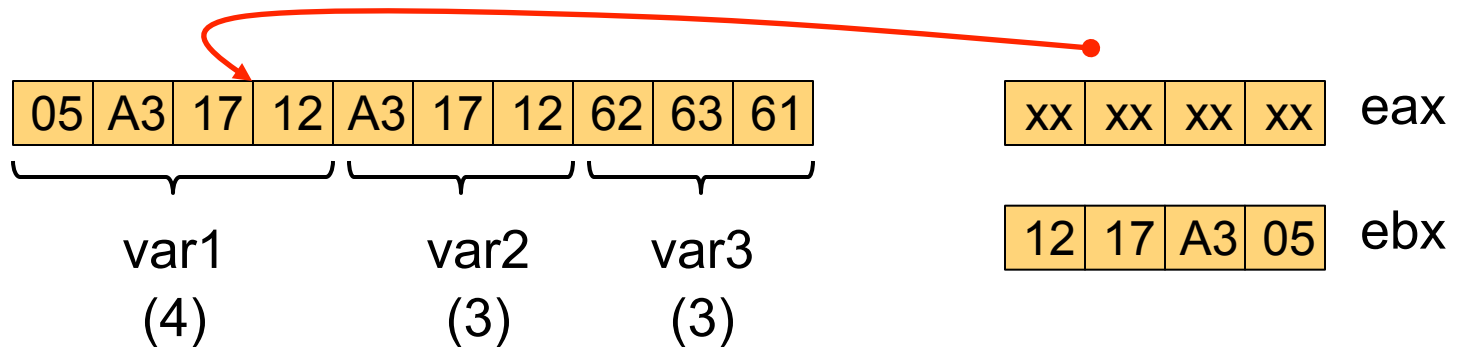
```
mov    eax, var1
add    eax, 3
mov    ebx, [eax]
add    ebx, 5
mov    [var1], ebx
```



# Practice #3

var1	dd	179
var2	db	0A3h, 017h, 012h
var3	db	"bca"

```
mov    eax, var1
add    eax, 3
mov    ebx, [eax]
add    ebx, 5
mov    [var1], ebx
```



# Practice #4

- Consider the following program

```
var1      db      "b", "ca", 0
var2      db      3, 0, 0, 0
var3      times 2 dw      012h
```

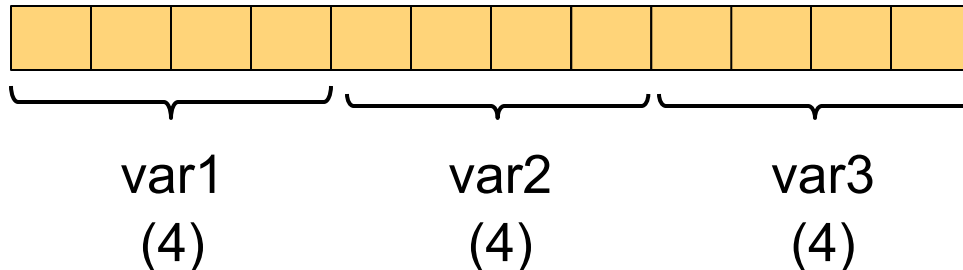
```
mov      eax, var3
mov      ebx, var1
sub      eax, 4
add      ebx, [eax]
mov      dword [ebx], 42
```

- What is the final layout of memory starting at address var1 on a Little Endian Machine?

# Practice #4

```
var1      db      "b","ca",0
var2      times   db 3,0,0,0
var3      times 2 dw      012h
```

```
mov      eax, var3
mov      ebx, var1
sub      eax, 4
add      ebx, [eax]
mov      dword [ebx], 42
```

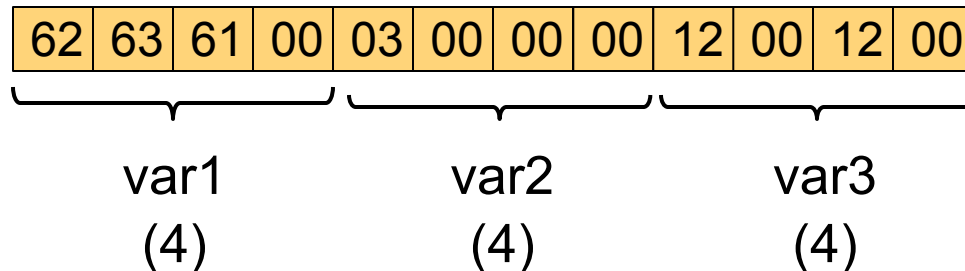




# Practice #4

```
var1      db      "b","ca",0
var2      times   db 3,0,0,0
var3      times 2 dw      012h
```

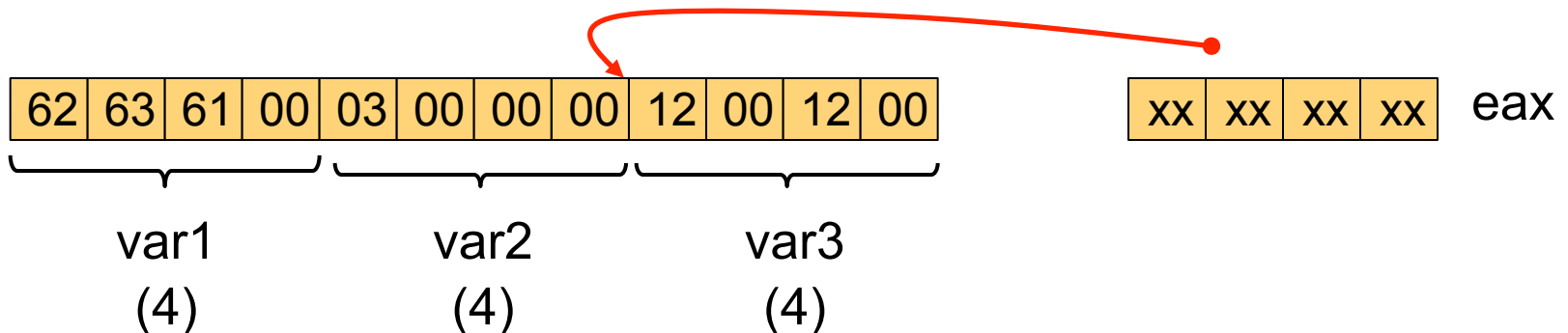
```
mov      eax, var3
mov      ebx, var1
sub      eax, 4
add      ebx, [eax]
mov      dword [ebx], 42
```



# Practice #4

```
var1      db      "b","ca",0
var2      times   db 3,0,0,0
var3      times 2 dw      012h
```

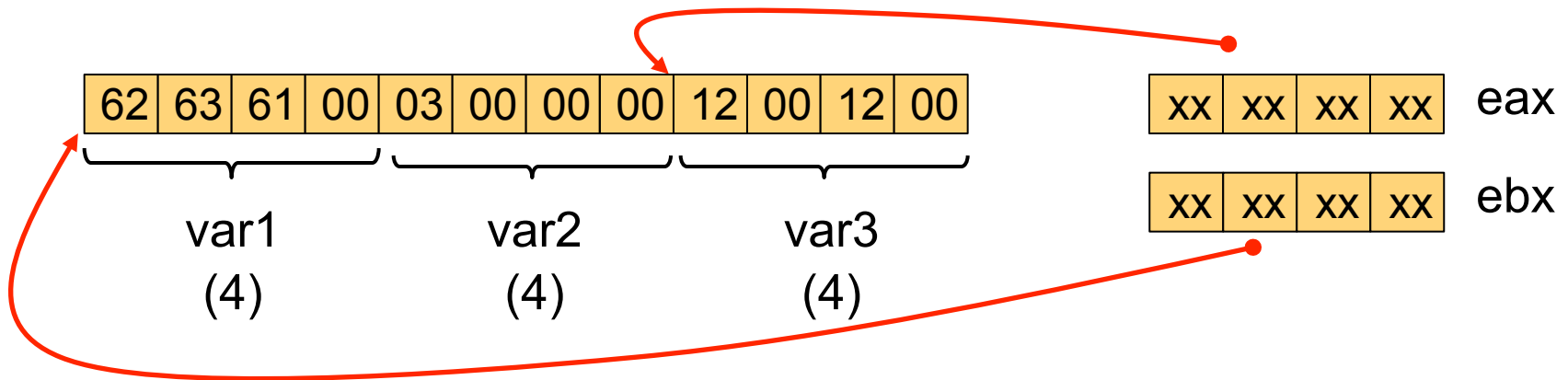
```
mov      eax, var3
mov      ebx, var1
sub      eax, 4
add      ebx, [eax]
mov      dword [ebx], 42
```



# Practice #4

```
var1      db      "b","ca",0
var2      times   db 3,0,0,0
var3      times 2 dw    012h
```

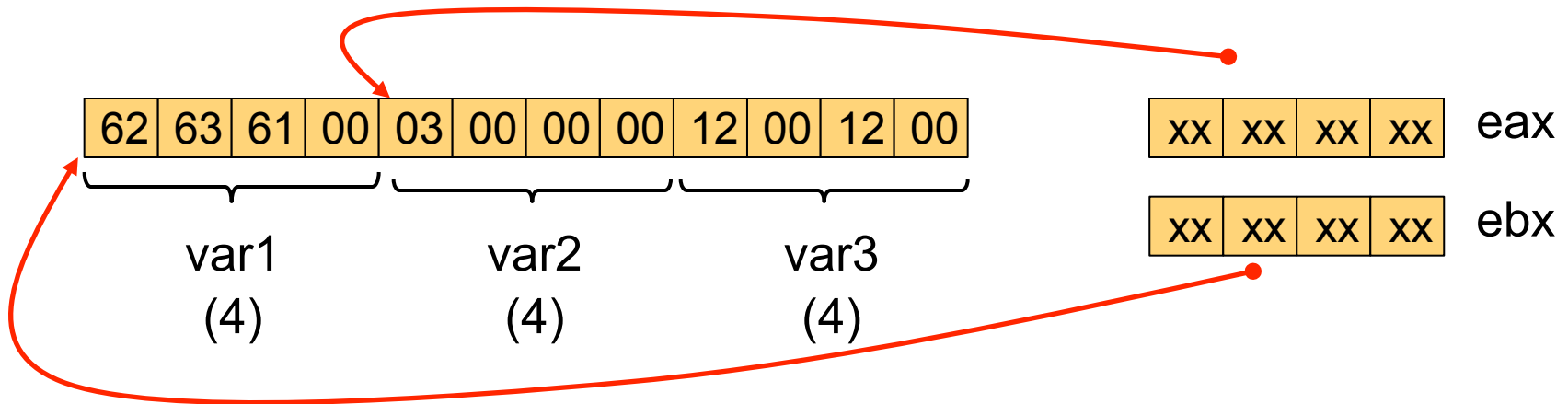
```
mov      eax, var3
mov      ebx, var1
sub      eax, 4
add      ebx, [eax]
mov      dword [ebx], 42
```



# Practice #4

```
var1      db      "b","ca",0
var2      times 3 db  0,0,0
var3      times 2 dw  012h
```

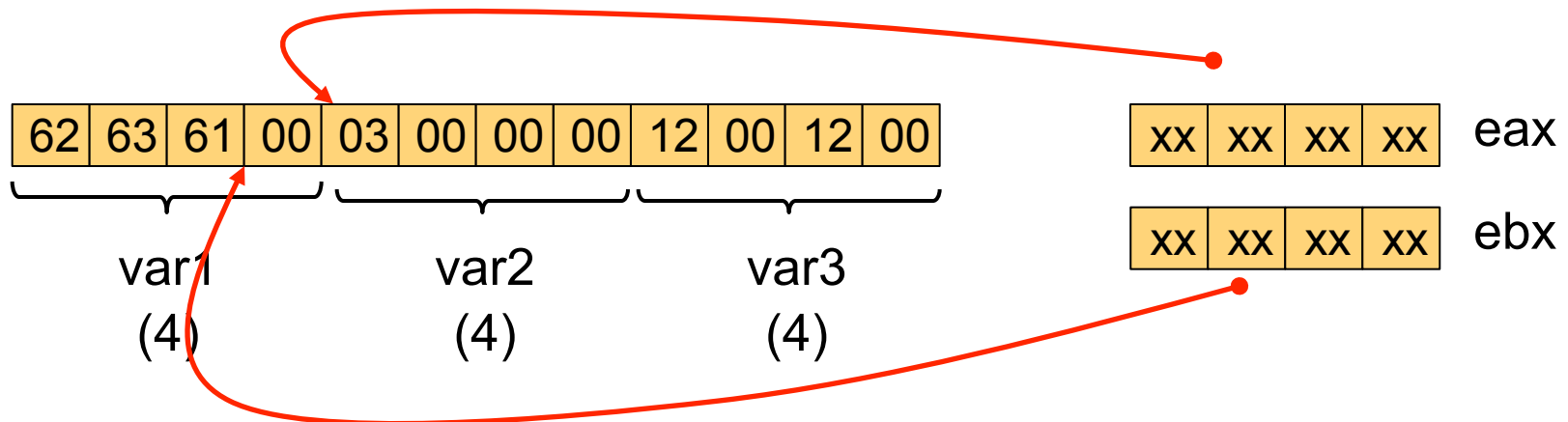
```
mov  eax, var3
mov  ebx, var1
sub  eax, 4
add  ebx, [eax]
mov  dword [ebx], 42
```



# Practice #4

```
var1      db      "b","ca",0
var2      times  db 3,0,0,0
var3      times 2  dw  012h
```

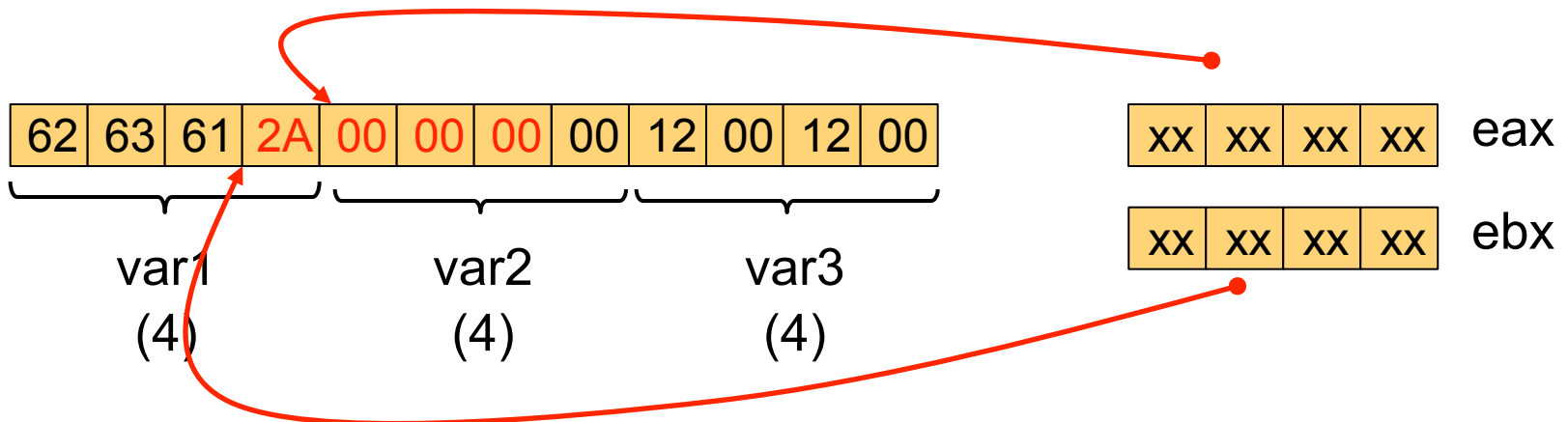
```
mov  eax, var3
mov  ebx, var1
sub  eax, 4
add  ebx, [eax]
mov  dword [ebx], 42
```



# Practice #4

```
var1      db      "b","ca",0
var2      times   db 3,0,0,0
var3      times 2 dw    012h
```

```
mov    eax, var3
mov    ebx, var1
sub    eax, 4
add    ebx, [eax]
mov    dword [ebx], 42
```



# Conclusion

- Should we make up other practices now or are we good?
  - Somebody wants to propose one?
- We can now do **Homework #2**
- Next lecture we'll go through a practice quiz (Zoom poll) on this module...