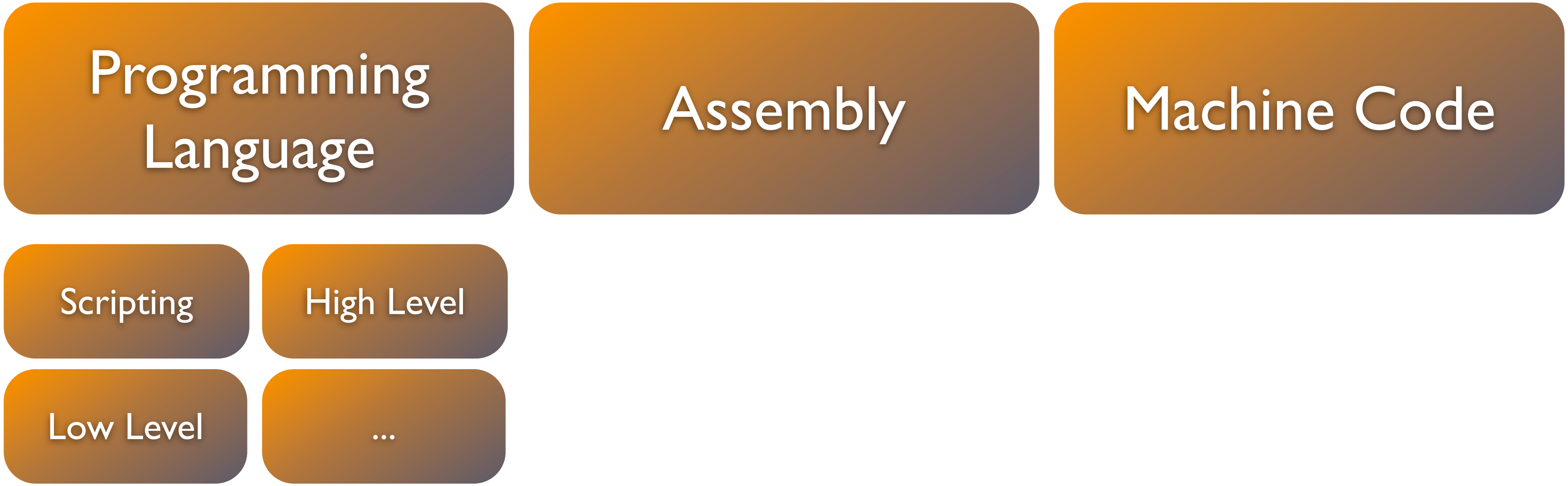
Assembly 101

From Programming Language to Machine Code



Programming Language

Assembly

Machine Code

Scripting

High Level

Low Level

...

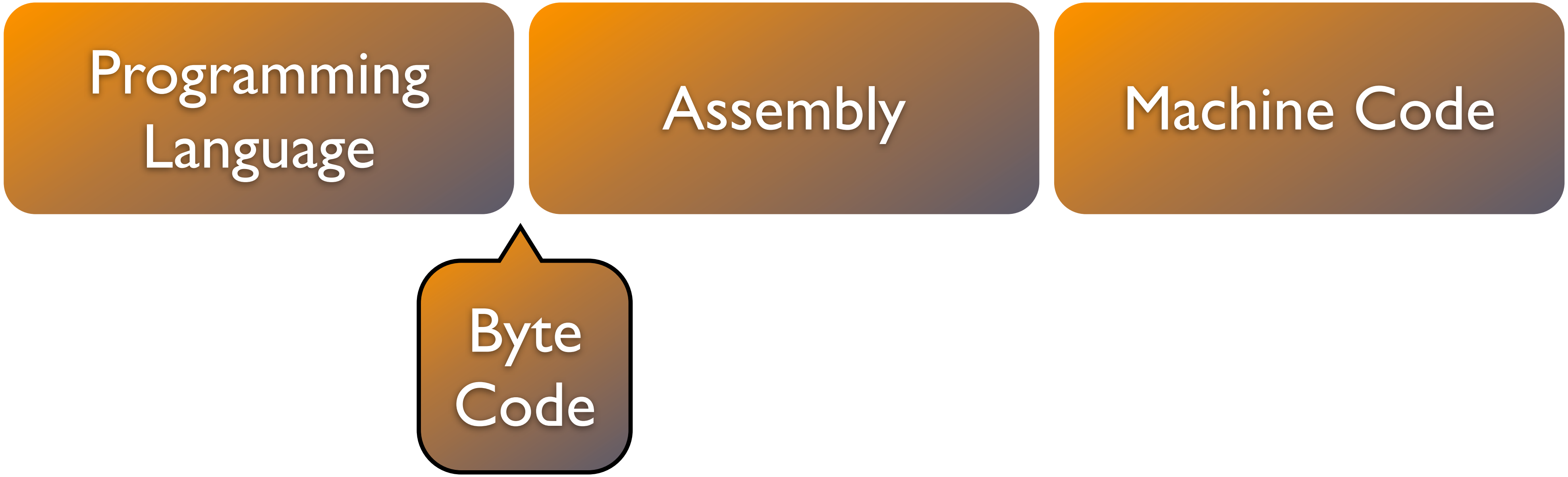
```
int i=5;
++i;
```

```
mov eax, 5
inc eax
```

```
0xB8 0x05000000
0x40
```

High-level to Assembly: Not a 1 to 1 correlation!

From Programming Language to Machine Code



Programming Language

Assembly

Machine Code

Byte Code

C#, Java, ...

From Programming Language to Machine Code

Programming Language	Assembly	Machine Code
Human readable		Understandable by hardware

From Programming Language to Machine Code

Programming Language	Assembly	Machine Code
Human readable	Human readable way to represent machine code	Understandable by hardware

What is Assembly

- A human readable way to view machine code
- Has (almost) a 1 to 1 correlation with machine code
- Extra features - labels, macros, memory layout, etc.
- Architecture Specific

Why do we teach it in a security course

- Reverse engineering
- Some bugs only lie in the assembly
- Shell code

* All will be covered in greater detail throughout the course.

Different Machines, Different Assembly

http://en.wikipedia.org/wiki/List_of_instruction_sets

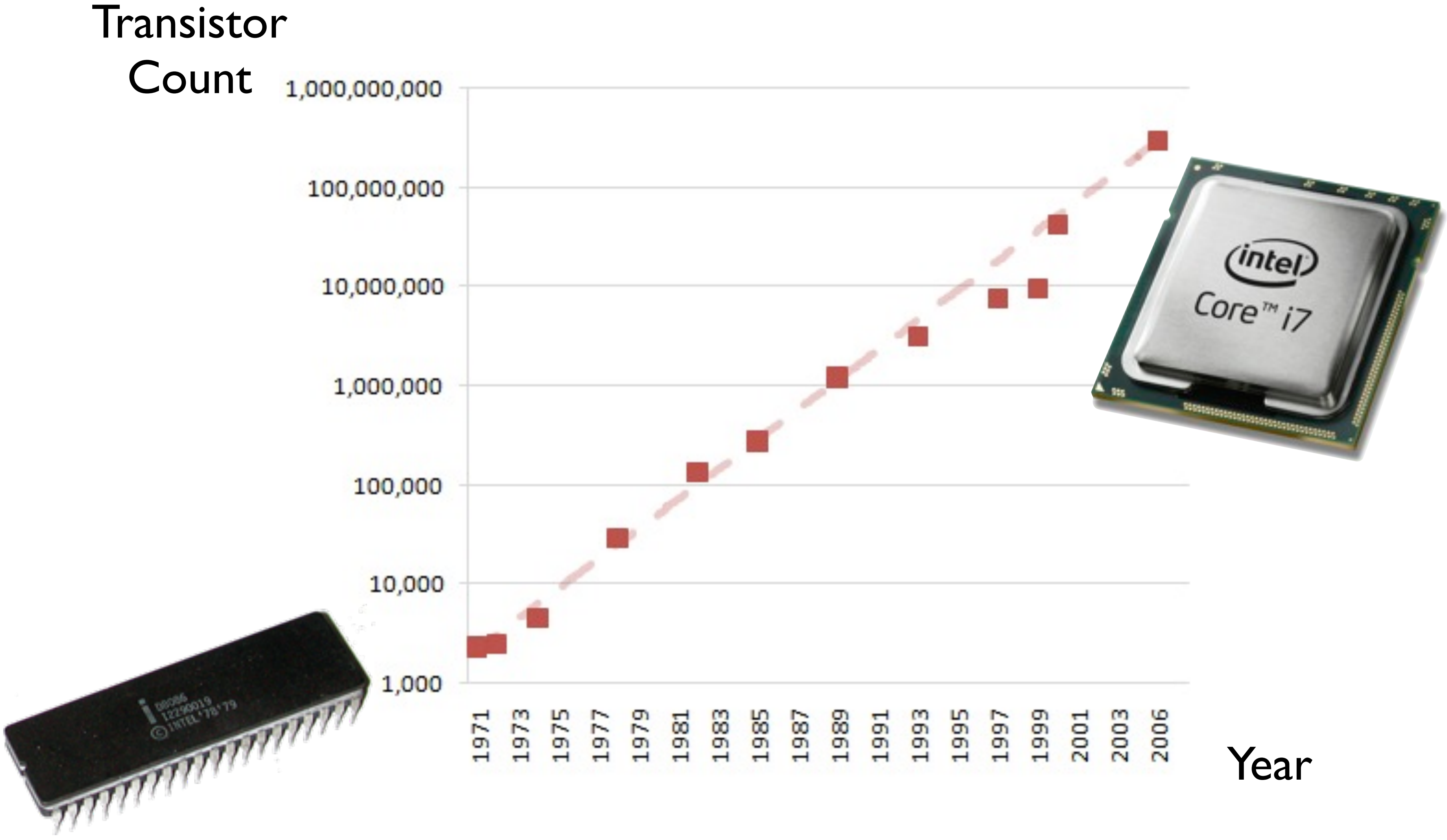
- Many many exist
- Even worse - same machine, different formats

x86

- Why do we zoom in on this one?...
- We will not give a complete instruction set in this lecture!

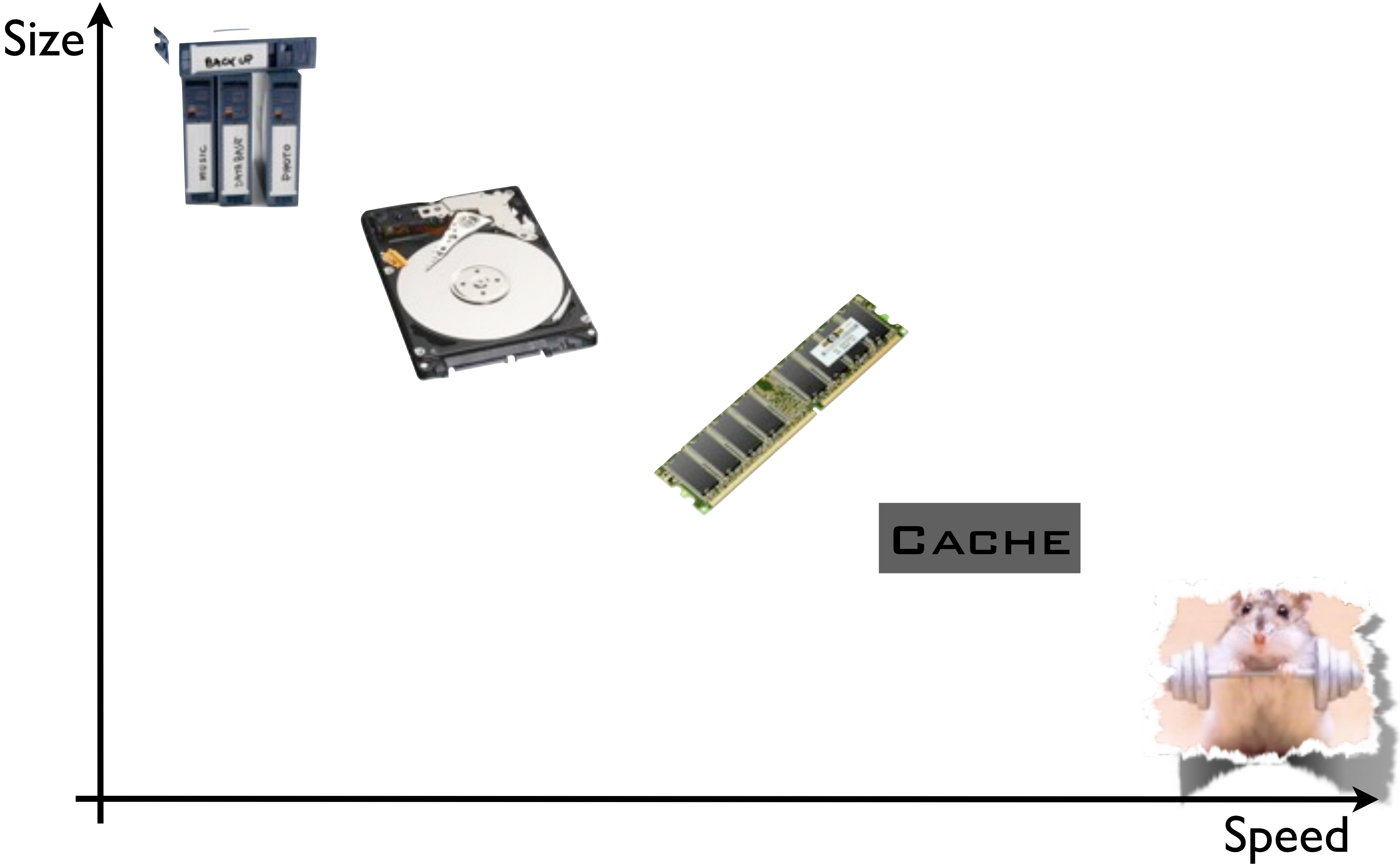
http://www.intel.com/design/intarch/manuals/243191.htm

x86



Transistor Count
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
1971
1973
1975
1977
1979
1981
1983
1985
1987
1989
1991
1993
1995
1997
1999
2001
2003
2006
Year
8086
12290019
©INTEL '78 '79
intel
Core™ i7

Memory Hierarchy



Size

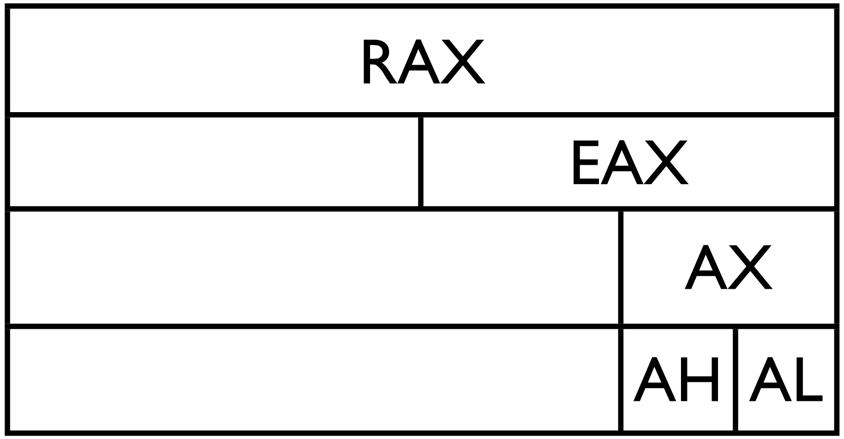
Cache

Speed

Registers

- Hold data
- Can be accessed fast and manipulated

General Purpose	Function Specific
AX, BX, ... EAX, EBX, ... RAX, RBX, ...	ESP, EIP, FLAGS, ...

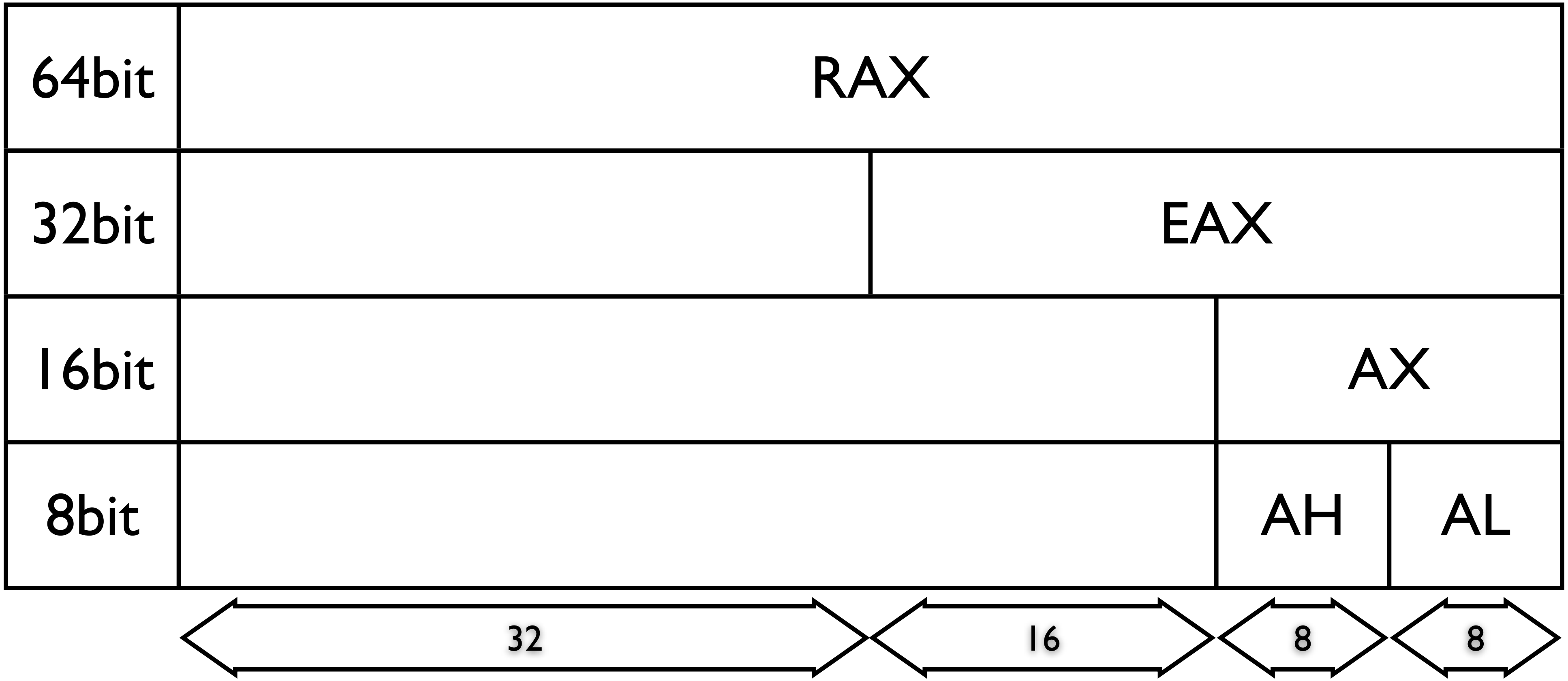

RAX

EAX

AX

AH AL

Parts of a Register



64bit	RAX			
32bit		EAX		
16bit			AX	
8bit			AH	AL
	32	16	8	8

General Purpose Registers

EAX	Accumulator	Arithmetic and other values
EBX	Base	Base for memory access
ECX	Counter	Loop counter
EDX	Data	I/O data

General Purpose Registers

- Are still general purpose
- Why do we care about the “intended” usage?

General Purpose Registers

- Are still general purpose
- Why do we care about the “intended” usage?

```
add al, 5               add bl, 5
```

General Purpose Registers

- Are still general purpose
- Why do we care about the “intended” usage?

```
add al, 5              add bl, 5
04h 05h                80h C3h 05h
```

General Purpose Registers

- Are still general purpose
- Why do we care about the “intended” usage?

```
mov [bx], 7          mov [cx], 7
```

General Purpose Registers

- Are still general purpose
- Why do we care about the “intended” usage?

```
mov [bx], 7           mov [cx], 7
valid                 invalid
```

Segment Registers

CS	Code Segment
DS	Data Segment
ES, FS, GS	Extra Segments
SS	Stack Segment

More on segments later on...

Other Registers

ESI	Source Index
EDI	Destination Index
EIP	Instruction Pointer
ESP	Stack Pointer
EBP	Base Pointer (of stack frame)
EFLAGS	Bit Flags

EFLAGS

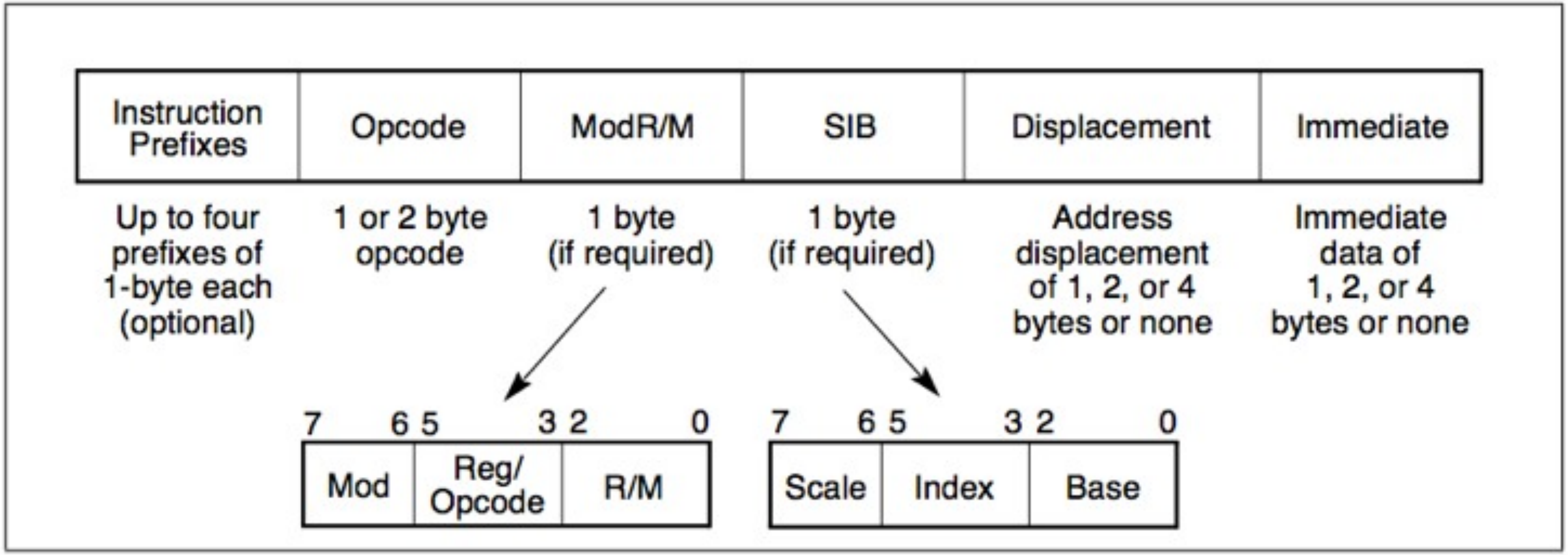
CF	Carry Flag
PF	Parity Flag
ZF	Zero Flag
SF	Sign Flag
TF	Trap (single step) Flag
IF	Interrupt Enabled Flag
DF	Direction Flag
OF	Overflow Flag

And More...

Even More Registers

- Floating point registers
- MMX registers
- SSE registers
- Debug registers
- Control registers
- Test registers
- GDTR, LDTR, IDTR
- More...

Assembly to Machine Code



Instruction Prefixes

Opcode

ModR/M

SIB

Displacement

Immediate

Up to four prefixes of 1-byte each (optional)

1 or 2 byte opcode

1 byte (if required)

1 byte (if required)

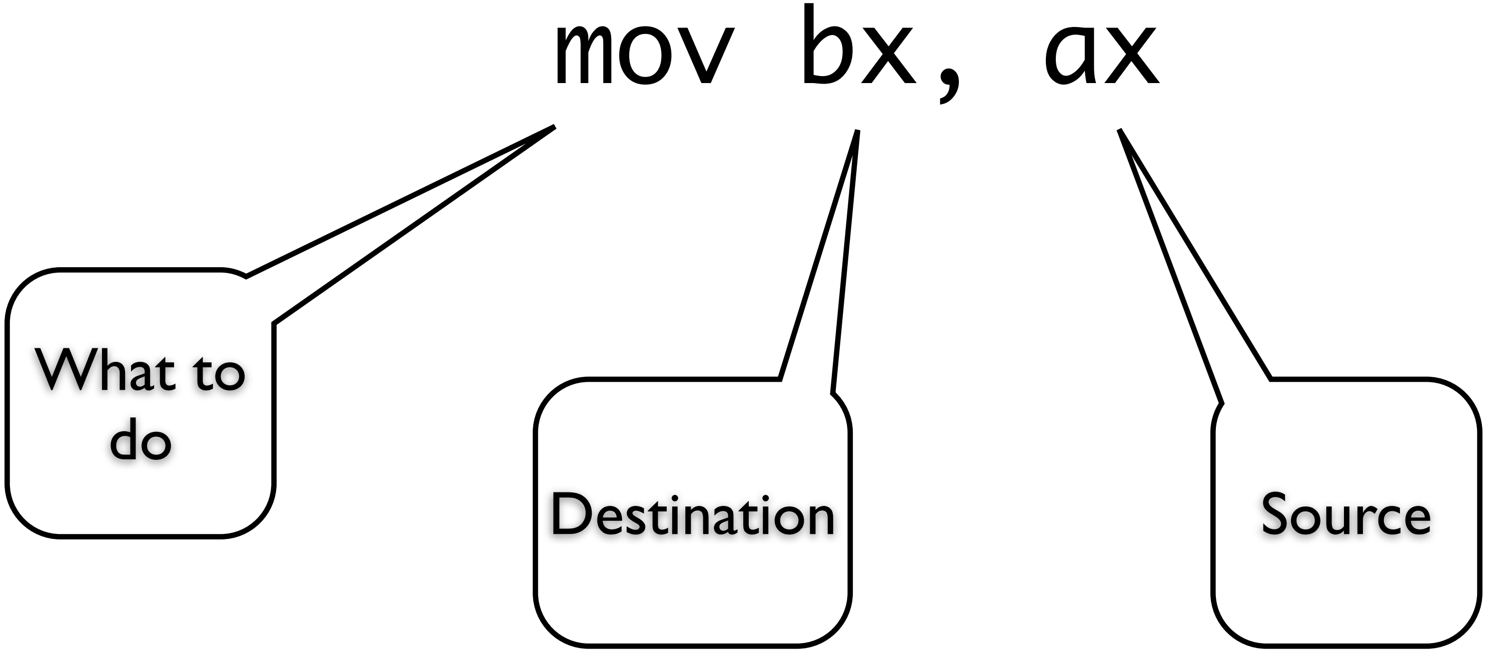
Address displacement of 1, 2, or 4 bytes or none

Immediate data of 1, 2, or 4 bytes or none

Allows complicated instructions such as:
mov eax, [ebp+ebx*4+4]

* [ ] states a dereference

Instruction Structure



```
mov bx, ax
```

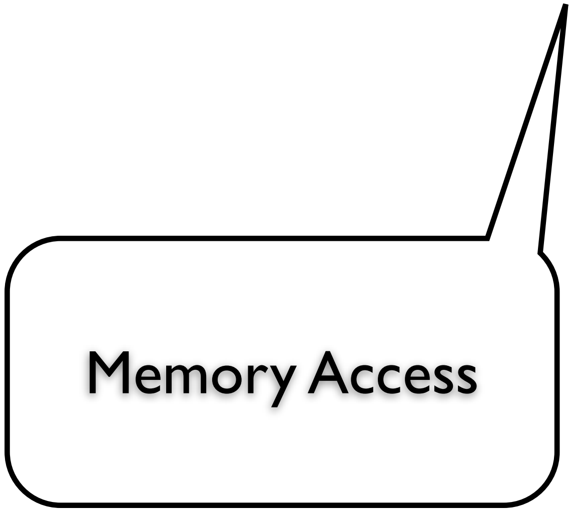
What to do

Destination

Source

Instruction Structure

```
mov [bx], ax
```



Memory Access

Instruction Structure

```
mov [bx], 7
```

What’s the problem here?

Instruction Structure

```
mov byte ptr [bx], 7
```

Specified that one byte will be copied

Instruction Structure

```
mov word ptr [bx], 7
```

Specified that 2 byte will be copied (0x0007)

Types of assembly instructions

- Basic instructions - mov, jmp
- Stack - push, pop
- ALU - add, mul, xor
- Floating point - faddp, fdiv
- SIMD (single instruction, multiple data)
    - MMX, SSE*
- String Operations
- Protection modes, interrupts, many more...

http://en.wikipedia.org/wiki/X86_assembly_language

Must know instructions

- mov
- push
- pop
- int
- db, dw, dd...

```
org 100h

mov dx, [val]
push dx
pop ax
mov ah,4Ch
int 21h

val db 05h
```

Assemble	nasm example.asm -o example.com
Test	C:\> echo %errorlevel%

Must know instructions

- inc
- dec
- add
- sub
- mul*
- div*

- shl
- shr
- and
- or
- not
- xor

* uses ax as first operand

Implementing Functions

- call
- ret
- pusha / popa (pushad /popad)

```
org 100h

mov al, 5 ; we want to print 5
call print_digit
mov al,0
mov ah,4Ch
int 21h
```

```
print_digit:
    pusha
    add ax, '0'
    mov [msg], al
    mov dx,msg
    mov ah,9
    int 21h
    popa
    ret

msg db ' ','$'
```

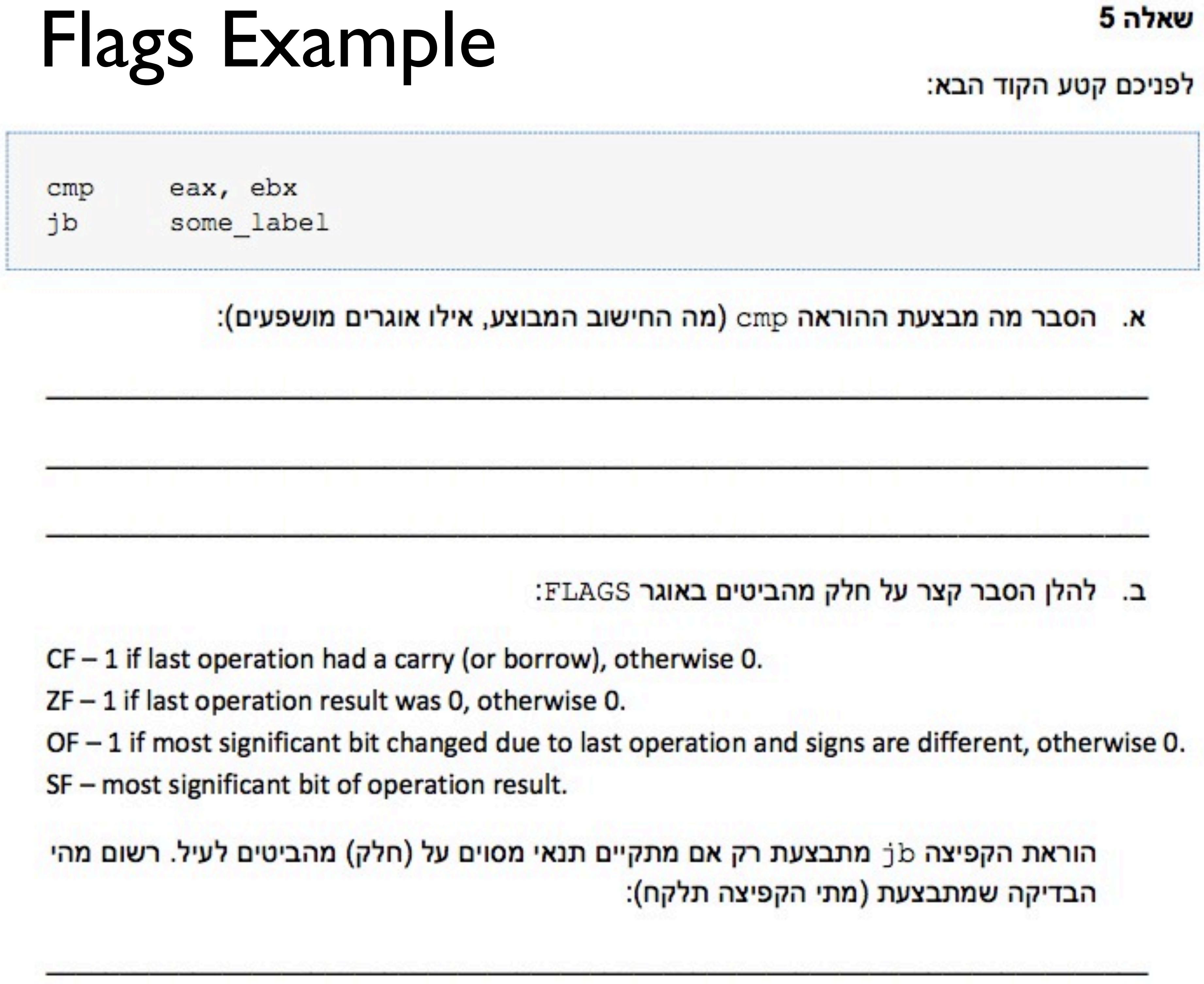
Must know instructions

- jmp
- cmp
- jz / je
- jnz / jne
- jg, jge, jl, jle (signed)
- ja, jae, jb, jbe (unsigned)
- loop

```
   mov cx, 4
   mov ax, 0
add_one:
   inc ax
   dec cx
   jnz add_one
```

```
   mov cx, 4
   mov ax, 0
add_one:
   inc ax
   loop add_one
```

Flags Example



שאלה 5

לפניכם קטע הקוד הבא:

```
cmp     eax, ebx
jb      some_label
```

א. הסבר מה מבצעת ההוראה cmp (מה החישוב המבוצע, אילו אוגרים מושפעים):

____

____

____

ב. להלן הסבר קצר על חלק מהביטים באוגר FLAGS:

CF – 1 if last operation had a carry (or borrow), otherwise 0.

ZF – 1 if last operation result was 0, otherwise 0.

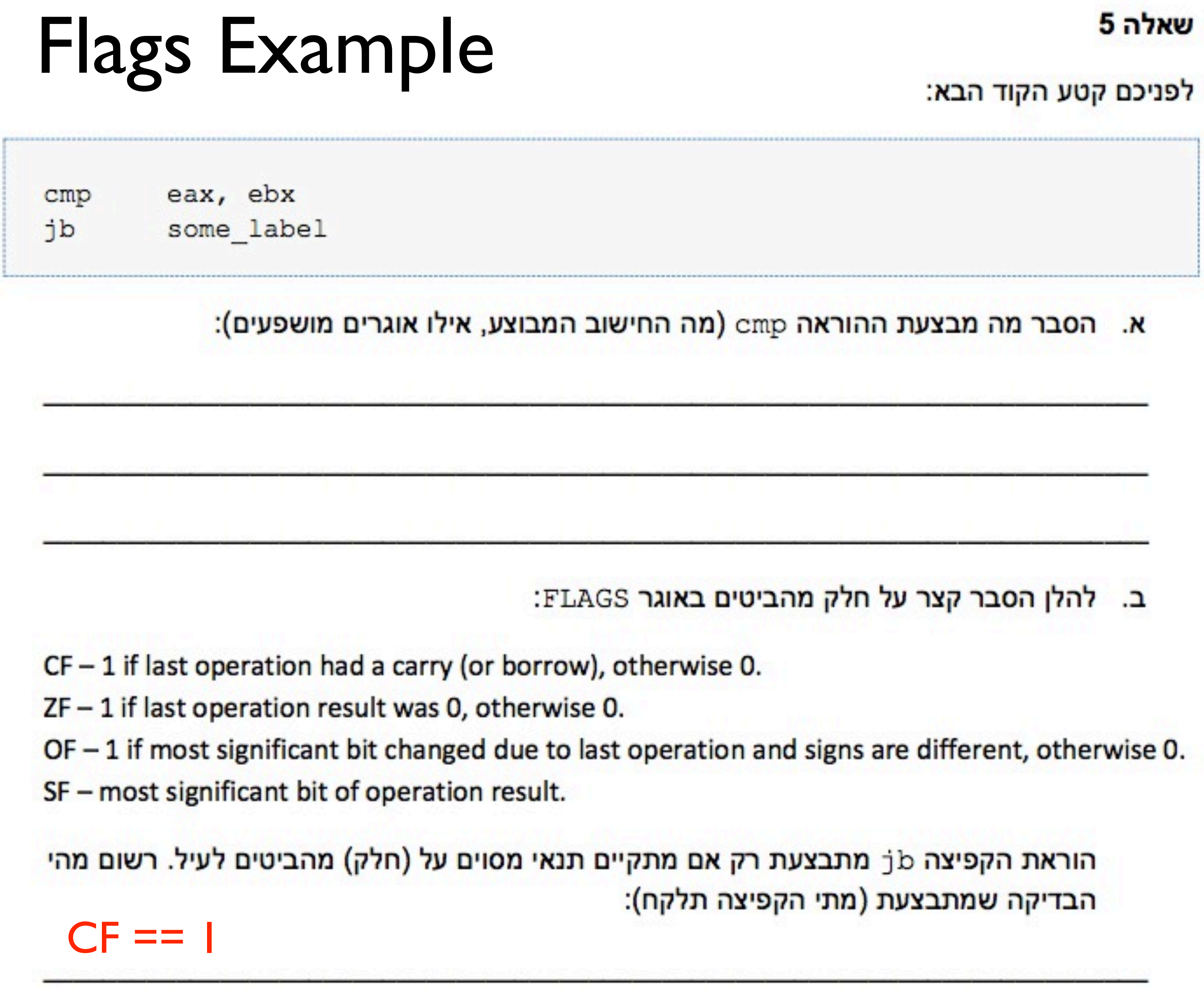
OF – 1 if most significant bit changed due to last operation and signs are different, otherwise 0.

SF – most significant bit of operation result.

הוראת הקפיצה jb מתבצעת רק אם מתקיים תנאי מסוים על (חלק) מהביטים לעיל. רשום מהי
הבדיקה שמתבצעת (מתי הקפיצה תלקח):

____

Flags Example



שאלה 5

לפניכם קטע הקוד הבא:

```
cmp     eax, ebx
jb      some_label
```

א. הסבר מה מבצעת ההוראה cmp (מה החישוב המבוצע, אילו אוגרים מושפעים):

____

____

____

ב. להלן הסבר קצר על חלק מהביטים באוגר FLAGS:

CF – 1 if last operation had a carry (or borrow), otherwise 0.

ZF – 1 if last operation result was 0, otherwise 0.

OF – 1 if most significant bit changed due to last operation and signs are different, otherwise 0.

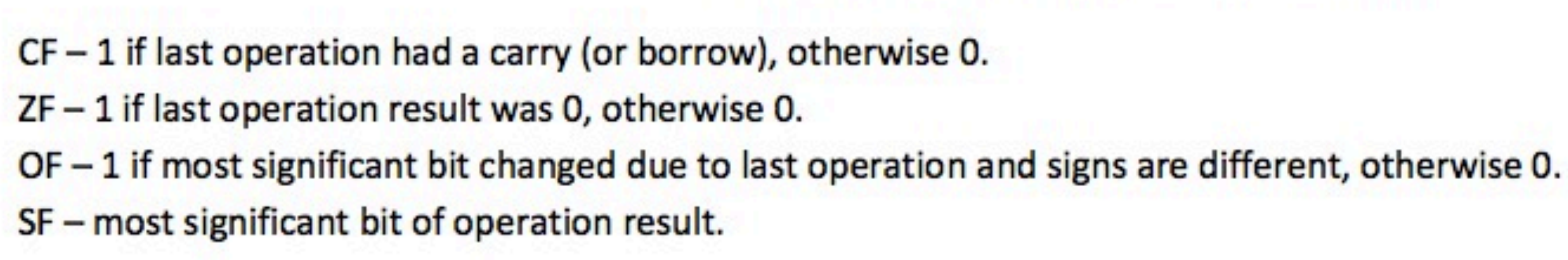
SF – most significant bit of operation result.

הוראת הקפיצה jb מתבצעת רק אם מתקיים תנאי מסוים על (חלק) מהביטים לעיל. רשום מהי
הבדיקה שמתבצעת (מתי הקפיצה תלקח):

CF == 1

____

Flags Example

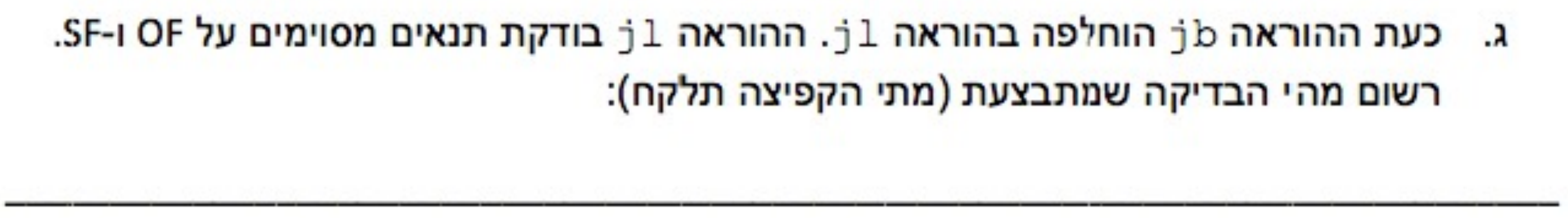


CF – 1 if last operation had a carry (or borrow), otherwise 0.

ZF – 1 if last operation result was 0, otherwise 0.

OF – 1 if most significant bit changed due to last operation and signs are different, otherwise 0.

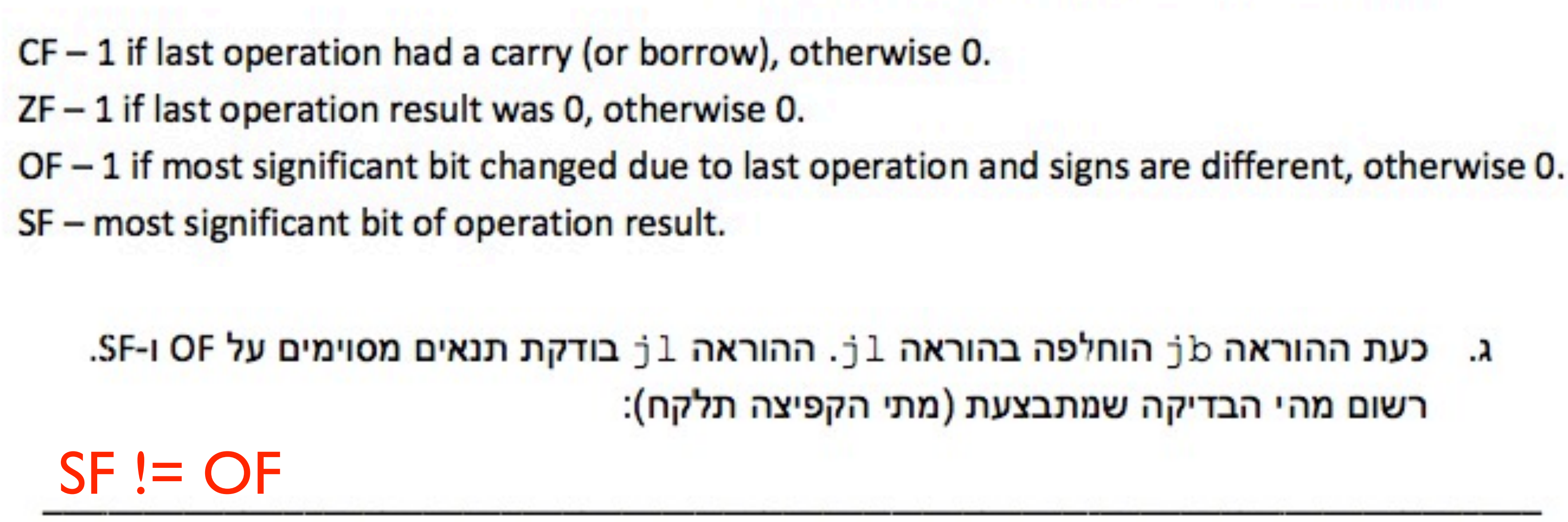
SF – most significant bit of operation result.



ג.   כעת ההוראה jb הוחלפה בהוראה jl. ההוראה jl בודקת תנאים מסוימים על OF ו-SF.
רשום מהי הבדיקה שמתבצעת (מתי הקפיצה תלקח):

____

Flags Example



CF – 1 if last operation had a carry (or borrow), otherwise 0.

ZF – 1 if last operation result was 0, otherwise 0.

OF – 1 if most significant bit changed due to last operation and signs are different, otherwise 0.

SF – most significant bit of operation result.

ג. כעת ההוראה jb הוחלפה בהוראה jl. ההוראה jl בודקת תנאים מסוימים על OF ו-SF.
רשום מהי הבדיקה שמתבצעת (מתי הקפיצה תלקח):

SF != OF

The lea Instruction

```
LEA SI, [EAX * 2 + EBX + 4]
```

The lea Instruction

- Used to implement & semantics
- Can add 3 different operands
- Result can be stored in a 4th register
- Can make code shorter / more readable...

Tricks with lea

Try to multiply eax by 5 using a single instruction...

Tricks with lea

Try to multiply eax by 5 using a single instruction...

```
LEA EAX, [EAX * 4 + EAX]
```

String instructions

- movs, cmps, scas, stos, lods (with b/w/d suffix)
- cld, std
- rep, repe/z, repne/z

String instructions

```
compare_strings:
   xor eax, eax

   lea   esi, [STR1_ADDRESS]
   lea   edi, [STR2_ADDRESS]
   mov   ecx, MAX_BYTES_TO_COMPARE

   repe cmpsb

   jz my_strcmp_end
   ; strings are not equal
   mov   eax, 1
my_strcmp_end:
   ...
```

Segment Registers

- CS, DS, ES, FS, GS, SS
- Different semantics in real/protected mode

```
mov word ptr [bx], 7
```

Implies ↓

```
mov word ptr ds:[bx], 7
```

Segments in Real Mode

```
Segment: 0x1234
Offset:  0x5678

 12340
+ 5678
 -----
 179b8
```

- Can address (a bit more than) 1MB
- No paging in real mode (result is a physical address)

Segments in Protected Mode

- Segment registers are “selectors”
  - bits 0-1: privilege
  - bit 2: GDT / LDT selector
  - bits 3-15: index into GDT / LDT

Segments in Protected Mode

- GDT / LDT entry contains
  - Address of segment
  - Size limit of segment
  - Flags
- Calculated address is virtual (assuming paging is on)

Segments in .COM Files

- Only one segment in .com files
- Limited to 0xFF00 bytes

org 100h

- Assemble all subsequent code starting from address 100h
- Important for .com programming
  - First 100h contains the PSP (Program Segment Prefix)

org 100h

Code	Result	
	Address	Instruction

`org 100h`
`jmp 200h`
	`100h`	`JMP 200h`

org 100h

Code	Result	
	Address	Instruction
org 100h jmp 200h	100h	JMP 200h
org 200h jmp 200h		

org 100h

Code	Result	
	Address	Instruction

`org 100h`
`jmp 200h`
	`100h`	`JMP 200h`

`org 200h`
`jmp 200h`
	`100h`	`JMP 100h`

PSP Example

Offset	Size	Contents
80h	1 byte	Number of bytes on command-line
81h - 0ffh	127 byte	Command-line (terminated by a 0Dh)

http://en.wikipedia.org/wiki/Program_Segment_Prefix

PSP Example

```
org   100h

; int 21h subfunction 9 requires '$' to terminate string
xor   bx, bx
mov   bl, [80h]
mov   byte [bx + 81h], '$'

; print the string
mov   ah, 9
mov   dx, 81h
int   21h

; exit
mov   ax, 4C00h
int   21h
```

http://en.wikipedia.org/wiki/Program_Segment_Prefix

Other stuff

- ; this is a remarks
- some_labels:
- Different assemblers have different extensions

Hello world

```
org 100h
mov dx,msg
mov ah,9
int 21h
mov ah,4Ch
int 21h
msg db 'Hello, World!',0Dh,0Ah,'$'
```

Hello world

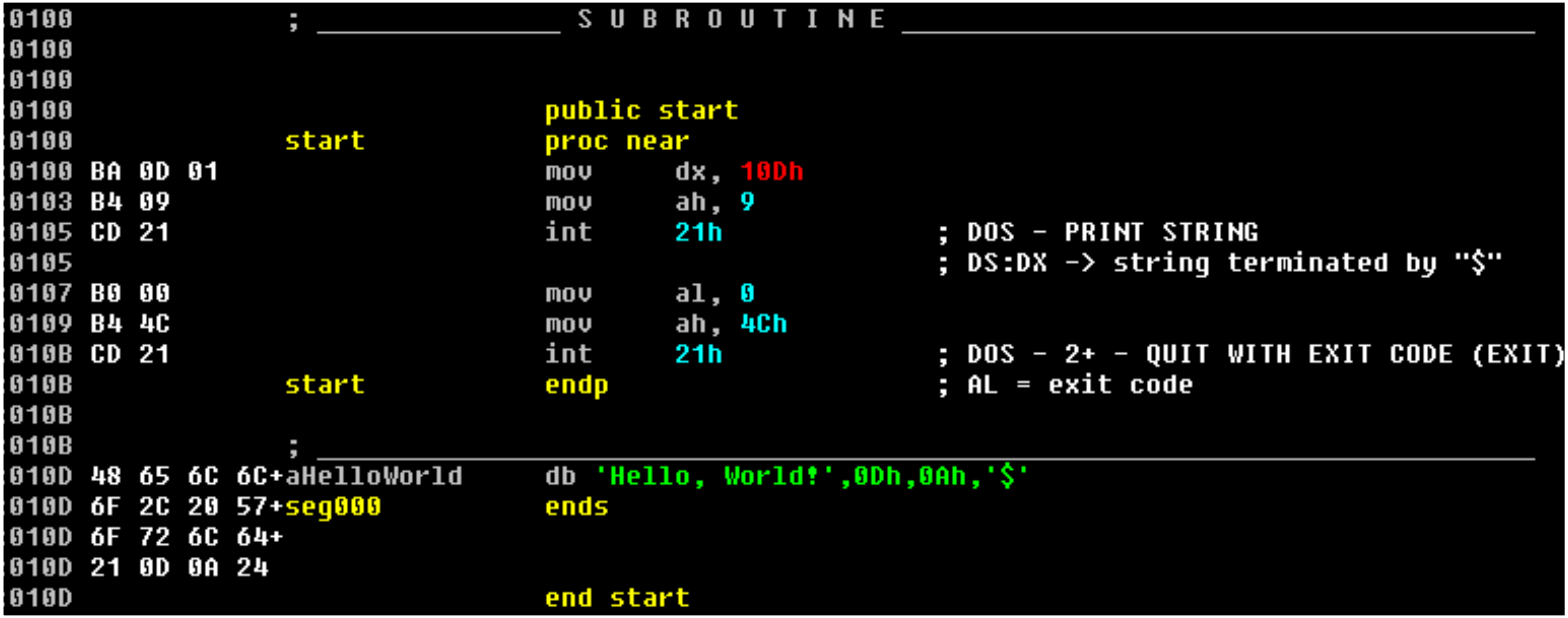
```
org 100h
mov dx,msg
mov ah,9
int 21h
mov al,0
mov ah,4Ch
int 21h
msg db 'Hello, World!',0Dh,0Ah,'$'
```

```
BA0D01B409CD21
B000B44CCD2148
656C6C6F2C20576
F726C64210D0A24
```

```
nasm hello.asm -o hello.com

nasm -h
```
]

Hello world



```
0100             ; ________________ S U B R O U T I N E ________________________________________
0100
0100
0100                             public start
0100             start           proc near
0100 BA 0D 01                    mov     dx, 10Dh
0103 B4 09                       mov     ah, 9
0105 CD 21                       int     21h             ; DOS - PRINT STRING
0105                                                     ; DS:DX -> string terminated by "$"
0107 B0 00                       mov     al, 0
0109 B4 4C                       mov     ah, 4Ch
010B CD 21                       int     21h             ; DOS - 2+ - QUIT WITH EXIT CODE (EXIT)
010B             start           endp                    ; AL = exit code
010B
010B             ; ____________________________________________________________________________
010D 48 65 6C 6C+aHelloWorld     db 'Hello, World!',0Dh,0Ah,'$'
010D 6F 2C 20 57+seg000          ends
010D 6F 72 6C 64+
010D 21 0D 0A 24
010D                             end start
```

Debug Demo

http://www.armory.com/~rstevew/Public/Tutor/Debug/debug-manual.html
http://thestarman.pcministry.com/asm/debug/debug.htm

What do I do (I)

```
org 100h

xor ax, ax
inc ah
mov cx,7

do_it:
   call print_digit*
   mov dl, al
   mov al, ah
   add ah, dl
   loop do_it
```

```
mov al,0
mov ah,4Ch
int 21h
```

* As seen before. Prints the digit in al

What do I do (II)

```
org 100h

mov eax, 5
mov ebx, 7
call swap_v1
call swap_v2

mov ah,4Ch
int 21h
```

```
swap_v1:
   xor eax, ebx
   xor ebx, eax
   xor eax, ebx
swap_v2:
   mov ecx, eax
   mov eax, ebx
   mov ebx, ecx
ret
```

What is the returned value?

What do I do (III)

```
org 100h

db 0BAh, 0Dh, 01h, 0B4h, 09h, 0CDh, 21h, 0B0h
db 00h, 0B4h, 4Ch, 0CDh, 21h, 48h, 65h, 6Ch
db 6Ch, 6Fh, 2Ch, 20h, 57h, 6Fh, 72h, 6Ch
db 64h, 21h, 0Dh, 0Ah, 24h
```

What’s wrong here?

```
org   100h

FIRST_VALUE db 0c3h
SECOND_VALUE db 3ch

; al will contain the result of c3-3c
mov al, [FIRST_VALUE]
sub al, [SECOND_VALUE]

; terminate
mov ah, 4ch
int 21h
```

What's wrong here?

```
org   100h

FIRST_VALUE db 0c3h
SECOND_VALUE db 3ch

; al will contain the result of c3-3c
mov al, [FIRST_VALUE]
sub al, [SECOND_VALUE]

; terminate
mov ah, 4ch
int 21h
```

First instruction is 0xc3,
which means 'ret'!

What's wrong here?

```
org   100h

FIRST_VALUE db 0c3h
SECOND_VALUE db 3ch

; al will contain the result of c3-3c
mov al, [FIRST_VALUE]
sub al, [SECOND_VALUE]

; terminate
mov ah, 4ch
int 21h
```

First instruction is 0xc3, which means 'ret'!

How would you fix this?

Working Version

```
org   100h

; al will contain the result of c3-3c
mov al, [FIRST_VALUE]
sub al, [SECOND_VALUE]

; terminate
mov ah, 4ch
int 21h

FIRST_VALUE db 0c3h
SECOND_VALUE db 3ch
```