

פרק 8 פקודות בקרה

למה צריך פקודות בקרה?

- ▶ עד כה עסקנו בתוכניות שמתקדמות פקודה אחרי פקודה
 - העתק ל- ax את הערך 3
 - העתק ל- bx את הערך 4
 - הוסף ל- ax את bx
 - כפול את התוצאה ב-2 והעתק ל- cx
- ▶ לעיתים נרצה שהתוכנית תבצע פקודות רק אם מתקיים תנאי מוגדר
 - אם האיבר הראשון במערך גדול מהאיבר השני, אז $cx=1$
 - אחרת $cx=2$

אבני בנין לפקודות בקרה

- ▶ יכולת להשוות בין שני ערכים
 - פקודת `cmp`
- ▶ יכולת "לקפוץ" לנקודה מוגדרת בקוד בעקבות תוצאת ההשוואה
 - סוגים שונים של פקודות קפיצה:
 - קפיצה בלתי מותנית `jmp`
 - קפיצה מותנית `jb, jbe, ja, jg, je, jne` ועוד
- ▶ יכולת לחזור על פעולה כל עוד מתקיים תנאי מוגדר
 - פקודת `loop`

פקודת cmp



- ▶ קיצור של compare
- ▶ מקבלת שני אופרנדים

cmp operand1, operand2

- ▶ שאלת חשיבה: איך המעבד מבצע
?cmp

פקודת cmp

- ▶ המעבד מבצע חיסור בין האופרנדים
 - בניגוד לחיסור רגיל, התוצאה אינה מועתקת מה-ALU אל אופרנד היעד
 - רק הדגלים מושפעים
- ▶ איך הקוד הבא ישפיע על מצב הדגלים?

הפקודה	דגל האפס - ZF (1 כאשר התוצאה היא אפס)	דגל הסימן - SF (1 כאשר הביט השמאלי של התוצאה הוא 1)	דגל הנשא - CF (1 כאשר המחסר גדול מהמחוסר בייצוג unsigned)
mov ax, 3	<input type="text"/>		
cmp ax, 3	<input type="text"/>		
cmp ax, 2	<input type="text"/>		
cmp ax, 4	<input type="text"/>		

פקודת cmp - המשך

צורות כתיבה חוקיות של cmp ▶

תוצאה	דוגמה	הפקודה
שינוי מצב הדגלים בהתאם ליחס בין האופרנדים	<u>cmp</u> al, bl	<u>cmp</u> register, register
	<u>cmp</u> ax, WordVar	<u>cmp</u> register, memory
	<u>cmp</u> WordVar, cx	<u>cmp</u> memory, register
	<u>cmp</u> ax, 5	<u>cmp</u> register, constant
	<u>cmp</u> ByteVar, 5	<u>cmp</u> memory, constant

פקודת קפיצה לא מותנית - jmp



- ▶ קיצור של jmp
- ▶ מקבלת כתובת
- ▶ שולחת את המעבד, ללא
- ▶ תנאי, לכתובת המוגדרת
- ▶ לאחר ביצוע פקודת ה-jmp
- ▶ ישתנה רגיסטר ה-IP

דוגמה - jmp

CODESEG

mov ax, 1

jmp cs:000Ah

; cs- the segment to jump to

; '0000A' - the offset in the segment.

ax	0001
bx	0000
cx	0000
dx	FFCD
si	0000
di	0000
bp	0000
sp	0100
ds	087C
es	0869
ss	087D
cs	0879
ip	000A



ax	0001
bx	0000
cx	0000
dx	FFCD
si	0000
di	0000
bp	0000
sp	0100
ds	087C
es	0869
ss	087D
cs	0879
ip	0028

קפיצה far-near

```

DATASEG
    address dw 000Ah
CODESEG
    mov     ax, @data
    mov     ds, ax
    mov     ax, 1
    jmp     [address]
    
```

▶ קפיצה near - בתוך אותו סגמנט

- אין צורך לציין את הסגמנט
- לדוגמה:

▶ קפיצה far - לסגמנט אחר

- חייבים לציין לאיזה סגמנט קופצים
 - `jmp cs:000A`
- שימושית אם יש מסגמנט קוד אחד
- בפועל לא נשתמש בה - תמיד נגדיר סגמנט קוד אחד

תרגיל - פקודת jmp

- ▶ נתונה התכנית הבאה (העתיקו אותה לתוך ה- CODESEG שבתכנית base.asm):

```
xor      ax, ax  
add      ax, 5  
add      ax, 4
```

- ▶ בעזרת הוספת פקודת jmp לתכנית, גירמו לכך שבסוף ריצת התכנית $ax=4$.

תוויות Labels



- ▶ שפת אסמבלי מאפשרת לנו לתת תווית Label לשורה בקוד
- ▶ פקודת jmp יכולה לקבל תווית

jmp Start

jmp WaitForKey

jmp PrintResult

- ▶ רצוי שלתוויות יהיו שמות בעלי משמעות
- ▶ למה זה טוב? אפשר לבצע שינויים בתוכנית בלי לכתוב מחדש את פקודות ה-jmp

דוגמאות - Labels

▶ מה יהיה ערכו של ax בסיום הריצה?

```
IncAX: xor    ax, ax  
       jmp    IncAX  
       add   ax, 3  
       inc   ax
```

▶ מה יקרה אם נשתול בתוכנית את הקוד הבא?

```
IncAx: inc    ax  
       jmp    IncAx
```

- ▶ נתונה התכנית הבאה (העתיקו אותה לתוך ה-
CODESEG שבתכנית base.asm):

```
xor    ax, ax  
add    ax, 5  
add    ax, 4
```

- ▶ בעזרת הוספת פקודת `jmp` אל `label` שתגדירו, גירמו
לכך שבסוף ריצת התכנית `ax=4`.

פקודות קפיצה מותנית

▶ קיים אוסף של פקודות קפיצה מותנות

◦ מבוצעות רק אם תנאי מוגדר מתקיים

▶ בדרך כלל ייכתבו אחרי פקודת `cmp`

▶ אופן הפעולה:

◦ בדיקה אם דגל כלשהו, או כמה דגלים, מקיימים תנאי מוגדר.

• לדוגמה, האם דגל האפס שווה ל-1

◦ אם התנאי מתקיים, IP משתנה לכתובת שהוגדרה על-ידי המשתמש.

• בדרך כלל כתובת זו תצוין באמצעות `label`

◦ אם התנאי אינו מתקיים, IP ממשיך לפקודה הבאה

שאלה למחשבה

- ▶ מתכנת תירגם לקוד אסמבלי את אוסף הפעולות הבא:
- "בצע פעולת השוואה בין שני אופרנדים. אם האופרנד הראשון גדול מהשני, בצע קפיצה"
 - המתכנת הכניס לאופרנד הראשון את הערך $00000001b$
 - המתכנת הכניס לאופרנד השני את הערך $10000001b$
 - האם המעבד יבצע קפיצה?

$10000001b$

$00000001b$



- ▶ הערך ב $1\ 000\ 000\ 001$ יכול לייצג שני מספרים:
 - 129_{10} , כמספר unsigned
 - -127_{10} , כמספר signed
- ▶ כדי שהמעבד יידע אם לבצע קפיצה, אנחנו צריכים להורות לו מהו סוג ההשוואה המבוקש
- ▶ בשאלת החשיבה, התשובה אם המעבד יקפוץ או לא תלויה בפקודת הקפיצה שהמתכנת כתב!

קפיצות signed, unsigned

- ▶ כל פקודות הקפיצה מתחילות באות **j** - jump
- ▶ קפיצות unsigned מכילות את האותיות **a** או **b** - קיצור של above או below
- ▶ קפיצות signed מכילות את האותיות **g** או **l** - קיצור של great או less
- ▶ בנוסף, פקודות הקפיצה יכולות להכיל את האותיות:
 - **n** - קיצור של not
 - **e** - קיצור של equal

קפיצות signed, unsigned

סיכום פקודות הקפיצה החוקיות ▶

מספרים Unsigned	מספרים Signed	משמעות הפקודה
JA - Jump if Above	JG - Jump if Greater	קפוץ אם האופרנד הראשון גדול מהשני
JB - Jump Below	JL - Jump if Less	קפוץ אם האופרנד הראשון קטן מהשני
JE - Jump Equal		קפוץ אם האופרנד הראשון והשני שווים
JNE - Jump Not Equal		קפוץ אם האופרנד הראשון והשני שונים
JAE - Jump if Above or Equal	JGE - Jump if Greater or Equal	קפוץ אם האופרנד הראשון גדול או שווה לאופרנד השני
JBE - Jump if Below or Equal	JLE - Jump if Less or Equal	קפוץ אם האופרנד הראשון קטן או שווה לאופרנד השני

תרגילים - פקודות קפיצה

- ▶ כיתבו תוכנית שבודקת אם המשתנה Var1 גדול מ- Var2 (שימו לב- יש להתייחס אל הערכים במשתנים כ-unsigned). אם כן- $ax=1$, אחרת $ax=0$.
- ▶ כיתבו תוכנית שבודקת אם ax גדול מאפס (יש להתייחס לערך של ax בייצוג שלו כ- signed כמובן), ואם כן מורידה את ערכו באחד.
- ▶ **אתגר:** כיתבו תוכנית, שמוגדר בה משתנה בגודל בית בשם TimesToPrintX. תנו לו ערך התחלתי כלשהו (חיובי). הדפיסו למסך כמות 'א'ים כערכו של TimesToPrintX.

הדרכה: קטע הקוד הבא מדפיס למסך את התו 'א':

```
mov    dl, 'x'
mov    ah, 2h
int    21h
```

- החזיקו ברגיסטר כלשהו את כמות הפעמים ש-'א' כבר הודפס למסך.
- צרו label שמדפיס x למסך ומקדם את הרגיסטר ב-1.
- לאחר מכן בצעו השוואה בין הרגיסטר ל-Times2PrintX, ואם לא מתקיים שוויון- קפצו ל-label.
- מה יקרה אם Times2PrintX יאותחל להיות מספר שלילי? או אפס?

פקודת loop



- ▶ ביצוע פעולה - או אוסף פעולות - מספר פעמים מוגדר
- ▶ מה לדעתכם מבצע הקוד הבא?
 - הריצו ובידקו!

```
mov    cx, 10
PrintX:                ; print 'x' to the screen
mov    dl, 'x'
mov    ah, 2h
int    21h
loop   PrintX
```

פקודת סלול- המשך

▶ פקודת סלול מבצעת את הפעולות הבאות- לפי הסדר

הבא:

- מפחיתה 1 מערכו של `cx`
- משווה את `cx` לאפס
- אם אין שוויון (כלומר, ערכו של `cx` אינו אפס) - מבצעת `jmp` ל-
`label` שהגדרנו

▶ קטעי הקוד הבאים זהים:

```
loop    SomeLabel
```

```
dec     cx
cmp     cx, 0
jne     SomeLabel
```

פקודת loop - שאלה למחשבה

```

mov    TimesToPrintX, 0
mov    cx, TimesToPrintX
PrintX:
mov    dl, 'x'
mov    ah, 2h
int    21h
loop   PrintX
    
```

▶ נתון הקוד הבא
▶ כמה פעמים יודפס 'x'
למסך?

- ▶ התשובה: 65,536 פעמים
 - פקודת ה-loop קודם מורידה את ערכו של cx ורק אחר כך בודקת אם הוא גדול או שווה לאפס!
- ▶ הוסיפו שורות קוד כך שהתכנית תעבוד באופן תקין גם עבור $TimesToPrintX=0$

פקודת סלול-פתרון

```
mov    TimesToPrintX, 0
xor    cx, cx
mov    cx, TimesToPrintX
cmp    cx, 0
je     ExitLoop
PrintX:
mov    dl, 'x'
mov    ah, 2h
int    21h
loop   PrintX
ExitLoop:
...
```

תרגילים - פקודת loop

▶ סידרת פיבונצ'י: סדרת פיבונצ'י מוגדרת באופן הבא- האיבר הראשון הוא 0, האיבר השני הוא 1, כל איבר הוא סכום שני האיברים שקדמו לו. צרו תוכנית שמחשבת את עשרת המספרים הראשונים בסדרת פיבונצ'י ושומרת אותם במערך בעל 10 בתים. בסיום התוכנית המערך צריך להכיל את הערכים הבאים:

0,1,1,2,3,5,8,13,21,34

▶ צרו תוכנית שמחשבת את המכפלה $Var1 * Var2$, משתנים בגודל בית שיש להתייחס אליהם כ-unsigned, אך מבצעת זאת ע"י פעולת חיבור. הדרכה:

- יש לבצע פעולת חיבור של $sum = Var1 + sum$ ולחזור עליה ע"י loop כמות של $Var2$ פעמים.

פקודת loop- תרגיל אתגר

צרו תוכנית שמקבלת שני מספרים מהמשתמש ומדפיסה למסך מטריצה של 'x' בגודל המספרים שנקלטו- המספר הראשון הוא מספר השורות והשני הוא מספר העמודות במטריצה. לדוגמה עבור קלט 5 ואחר כך 4, יודפס למסך:

```
XXXX
XXXX
XXXX
XXXX
XXXX
```

הדרכה

```
mov ah, 1h
int 21h
sub al, '0'
```

קליטת ספרה: שורות הקוד הבאות קולטות ספרה מהמשתמש, ממירות אותה למספר בין 0 ל-9 ומעתיקות את התוצאה ל-al

```
mov dl, 'x'
mov ah, 2h
int 21h
```

הדפסה למסך: קטע הקוד הבא מדפיס למסך את התו 'x':

```
mov dl, 0ah
mov ah, 2h
int 21h
```

מעבר שורה: הפקודות הבאות גורמות להדפסה של מעבר שורה:

```

ScrCopy_cap    textequ  <dword ptr [bp+4]>
X_cap          textequ  <word ptr [bp-2]>
Y_cap          textequ  <word ptr [bp-4]>

Capture
    proc
    push        bp
    mov        bp, sp
    sub        sp, 4          ;Allocate room for locals.

    push        es
    push        ds
    push        ax
    push        bx
    push        di

    mov        bx, ScrSeg    ;Set up pointer to SCREEN
    mov        es, bx       ; memory (ScrSeg:0).

    lds        di, ScrCopy_cap ;Get ptr to capture array.

    mov        Y_cap, 0
    mov        X_cap, 0
    mov        bx, Y_cap
    imul       bx, 80        ;Screen memory is a 25x80 array
    add        bx, X_cap     ; stored in row major order
    add        bx, bx        ; with two bytes per element.

    mov        ax, es:[bx]   ;Read character code from screen.
    mov        [di][bx], ax ;Store away into capture array.

    inc        X_Cap        ;Repeat for each character on this
    cmp        X_Cap, 80    ; row of characters (each character
    jb         XLoop       ; in the row is two bytes).

    inc        Y_Cap        ;Repeat for each row on the screen.

```

Page 611

Art of מתוך הספר ▶
Assembly

משמאל לימין: ▶

תוויות והנחיות ◦
לאסמבלר

פקודה ◦

אופרנדים ◦

תיעוד ◦

בתוך לולאה לא ▶
מבצעים הזזה של
טאב

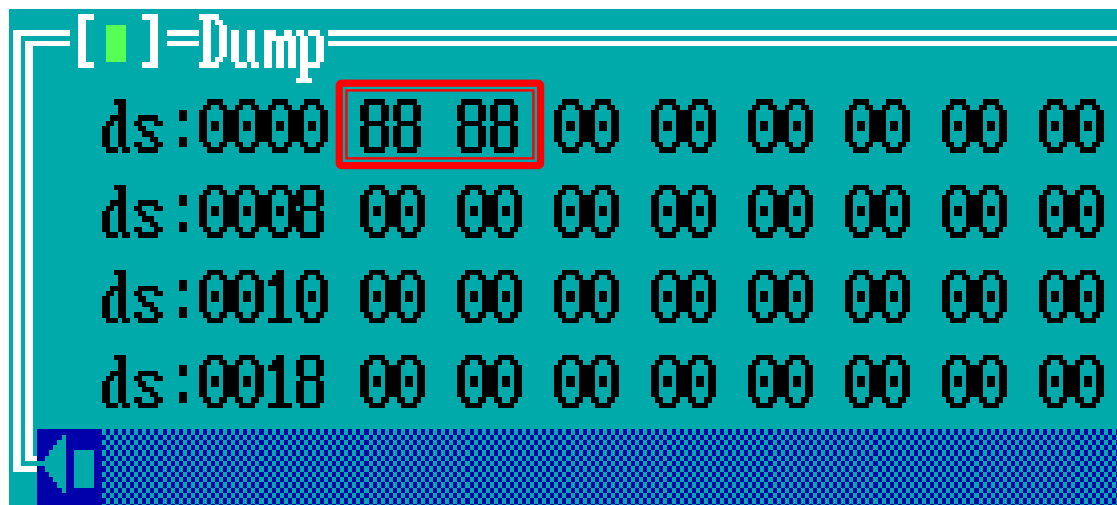
סיכום נושא מספרים signed, unsigned

- ▶ בשלב הראשון למדנו שאין הבדל בדרך בה נשמרים בזיכרון המחשב מספרים signed ו-unsigned
 - הכל נשמר כאחדות ואפסים
 - הפרשנות שלנו היא שמעניקה את הערך הנכון למידע

```

DATASEG:
Var1    db    ?
Var2    db    ?

CODESEG
mov     al, -120
mov     [Var1], al
mov     al, 136
mov     [Var2], al
    
```



סיכום נושא מספרים signed, unsigned

- ▶ בפרקים האחרונים ראינו איך נוצרת הפרשנות:
 - פקודות מתמטיות נפרדות (כגון `mul` / `imul`)
 - סטים נפרדים של פקודות קפיצה מותנות

```
mov    bl, 00000010b
mov    al, 11111011b
imul bl ; -10
```

```
mov    bl, 00000010b
mov    al, 11111011b
mul   bl ; 502
```

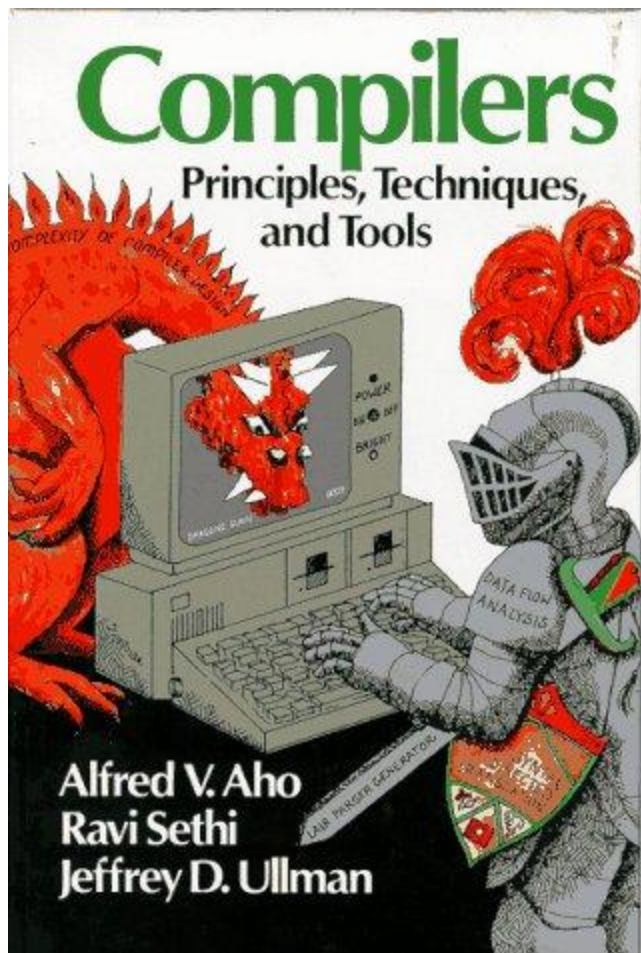
מספרים Unsigned	מספרים Signed	משמעות הפקודה
JA - Jump if Above	JG - Jump if Greater	קפוצ אם האופרנד הראשון גדול מהשני
JB - Jump Below	JL - Jump if Less	קפוצ אם האופרנד הראשון קטן מהשני
JE - Jump Equal		קפוצ אם האופרנד הראשון והשני שווים
JNE - Jump Not Equal		קפוצ אם האופרנד הראשון והשני שונים
JAE - Jump if Above or Equal	JGE - Jump if Greater or Equal	קפוצ אם האופרנד הראשון גדול או שווה לאופרנד השני
JBE - Jump if Below or Equal	JLE - Jump if Less or Equal	קפוצ אם האופרנד הראשון קטן או שווה לאופרנד השני

סיכום נושא מספרים signed, unsigned

אם כך, איך זה שבשפות עיליות יש טיפוסים משתנים מסוג signed ו-unsigned? ◀
 לדוגמה שפת C: ◦

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

סיכום נושא מספרים signed, unsigned



- ▶ הסבר: למדנו שקומפיילר הוא קוד שתפקידו לבצע המרה משפה עילית לשפת אסמבלי
 - הקומפיילר מתאים את פקודות האסמבלי לטיפוס המשתנה שהוגדר בשפה עילית
 - בקוד האסמבלי שנוצר, כבר אין משתנים signed/unsigned

base.asm - מה למדנו?

IDEAL

MODEL small

STACK 100h

DATASEG

CODESEG

start:

mov ax, @data

mov ds, ax

exit:

mov ax, 4c00h

int 21h

END start

- ▶ למדנו את השורות המוקפות בכחול
- ▶ השורות הירוקות:
 - IDEAL - הנחיה לאסמבלר, סגנון כתיבה
 - model - הנחיה לאסמבלר, ארגון הסגמנטים
 - start END - הנחיה לאסמבלר, יש לבצע המרה לקוד מכונה עד לשורה זו ולהתחיל את התכנית מהתווית start
- ▶ השורות המוקפות בחום - נלמד בפרקים הבאים