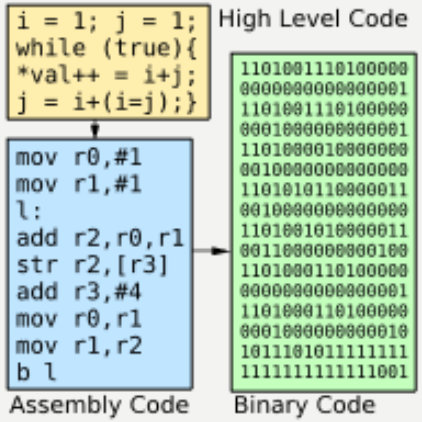
מבוא לאסמבלי



תרגול 1 – הנדסה לאחור – אביב תשפ"ב

אסמבלי אינטל (32 סיביות)

- x86-32bit – IA32 – הוצג לראשונה ב-1985 ע"י Intel
  - מעבד 80386
- גרסה קודמת וגרסה נוכחית
  - גרסת 16 סיביות הוצגה ב-1978 – מעבד 8086
  - גרסת 64 סיביות הוצגה ב-2003
  - אנו נתייחס רק לגרסת 32 סיביות
- מרחב הכתובות הוא 2^{32} בתים = 4GB
- Little Endian
- תומך ב-Protected mode
  - זיכרון וירטואלי
    - עם תמיכה בהרשאות קריאה/כתיבה/הרצה לכל דף זיכרון
  - תמיכה בהרשאות מ"ה (kernel+user modes)


"yay programming"
me
ASSEMBLY

מבנה הפקודות

Two operands:	mov eax, DWORD PTR [esp+8]
[prefix] inst dest, src	add eax, DWORD PTR [esp+8]
Single operand:	
[prefix] inst operand	inc ebx
No operands:	
inst	nop

- prefix – ניתן לצרף לפקודות מסוימות, נלמד בהמשך.
- inst – ה-opcode של פקודת המכונה.
- dest – תוצאת החישוב (וגם קלט אם יש צורך בשניים) – אוגר או הפנייה לזיכרון.
- operand / src – ערך, אוגר או הפנייה לזיכרון.
- אין שתי הפניות לזיכרון באותה הפקודה – למשל, לא קיים mov [eax],[esp].

מבנה הפקודות
הערות

- מסיבות היסטוריות הערך המספרי של ה-opcode של פקודות מכונה דומות עם אופרנדים שונים אינו בהכרח זהה.
- חלק מהפקודות ניגשות לרגיסטרים קבועים שאינם רשומים בפקודה, למשל mul ו movsb.
  - mul מכפילה את eax באופרנד, והמכפלה נכנסת לצרוף הרגיסטרים edx:eax (הערך הנמוך ב-eax).
  - movsb ודומותיה משתמשות ברגיסטרים esi ו edi כמצביעים לקלט והפלט, וב-ecx כמונה וחסם למספר ההפעלות עם rep. אין להן אופרנדים משלהן!

אסמבלי בסגנון ATT

- קיימת גרסה נוספת לצורת רישום האסמבלי למעבדי אינטל
  - דומה למבנה האסמבלי של PDP
  - כמובן, ניתן לתכנת בדיוק את אותן פקודות מכונה, בצורת כתיבה שונה
  - סדר האופרנדים הפוך
  - גודל המילה הוא חלק מ-inst, ולא מ-src ו-dest
  - זו ברירת המחדל של gcc
  - **לא נשתמש בגרסה זו בקורס**

Two operands:	movl (esp+8), %eax
inst src, dest	addl (esp+8), %eax
Single operand:	
inst operand	incl %ebx
No operands:	
inst	nop

קידומות - PREFIXES

- יש פקודות להן ניתן להוסיף קידומת ומשפיע על הביצוע של הפקודה. הקידומת מוסיפה בית (או שניים) לפני הפקודה.

- קידומת נפוצה הנרשמת במפורש היא rep וגרסאותיה.
  - מפעילה את הפקודה הבאה בלולאה.
  - לשם כך יש פקודות מכונה מיוחדות שיודעות לפעול בהתאם.
  - למשל repnz movsb – העתקת מחרוזת.

- movsb מעתיקה בית מהכתובת esi לכתובת edi ומקדמת את האוגרים.
- repnz/repne – מבצעות את הפקודה שלאחריהן לכל היותר ecx פעמים, עד אשר נלדק ה ZF (Zero Flag).

אוגרים

רגיסטרים

- eax, ebx, ecx, edx – רגיסטרים לשימוש כללי.
- esi, edi - רגיסטרים לשימוש כללי. במקור רגיסטרים לשמירת אינדקס.
- Stack pointer and base pointer - esp, ebp.
- Instruction pointer - eip.
- eflags - רגיסטר הדגלים.

- cs, ds, ss, es, fs, gs.
  - Segment registers (נקראים גם segment selectors).
  - מייצגים מרחבי זיכרון המשמשים למטרות שונות, הרשאות שונות.
  - פעלו באופן שונה בארכיטקטורות קודמות בנות 16 סיביות.

מוסכמות

- eax - מכיל את ערך החזרה של פונקציות.
- ecx - משמש כאינדקס של לולאות.
- esi, edi - כתובות מקור ומטרה במערכים.
- ebp - Base pointer – בסיס לגישה לפרמטרים ולמשתנים מקומיים.

- cs, ds, ss, es, gs - חלונות (Windows) מפנה אותם לאותו סגמנט, כך שניתן להתעלם מהם
- fs:[0] - משמש לטיפול בחריגות
  - fs הוא סגמנט של נתוני thread

הרגיסטרים
EAX, EBX, ECX, EDX

- ניתן לגשת ישירות למילה התחתונה של כל אחד מהרגיסטרים הללו, וכן לכל אחד משני הבתים התחתונים שלהם בגישה ישירה, על ידי שם ייחודי
- לדוגמא, ax, ah, al במקרה של eax

eax			32-bit register
	ax		16-bit register
	ah	al	Two 8-bit registers

- ובדומה בשלושת האחרים

ebx		bh	bl	and also bx
ecx		cx		and also cl, ch
edx		dh	dl	and also dx

דוגמאות

	eax							
					ax			
					ah		al	

mov eax, 87654321h	8	7	6	5	4	3	2	1
mov ax, 9876h	8	7	6	5	9	8	7	6
mov ah, 23h	8	7	6	5	2	3	7	6
xor al, al // Set the low 8 bits (al) to zero								
	8	7	6	5	2	3	0	0
dec eax	8	7	6	5	2	2	F	F
inc al	8	7	6	5	2	2	0	0

סיומת h במספר מייצגת בסיס 16 (הקסדצימלי).

רגיסטרים ופקודות נוספים

- במשך השנים הוספו מגוון פקודות ורגיסטרים לארכיטקטורה
  - במידת הצורך, פנו לתיעוד של אינטל
- xmm0-xmm7 הם רגיסטרים גדולים יחסית - רגיסטרי 128 סיביות השייכים להרחבת המולטימדיה של המעבדים – SSE המאפשרת פעולות SIMD
- כלומר הרגיסטרים הנ"ל משמשים את המעבד לביצוע פקודות floating point, ולכן הם גם משתלבים בתוך ההגדרות של calling conventions שונים.

זיכרון

	16 GB memory
	32 GB memory
Me In an Exam	0 GB memory

גדלי מילה

- **Byte** מייצג בית בן 8 סיביות
- **Word** מייצג מילה בת 16 סיביות
- **DWORD** מייצג מילה כפולה בת 32 סיביות

- כתובות ומצביעים הם **DWORD**

שיטות מיעון

- תמיכה בגישה ל-4GB בתי זיכרון (2^{32} בתים)
- ערך בסוגריים מרובעים [...] מייצג גישה לכתובת שבסוגריים
- בחירת גודל המילה אליה ניגשים ע"י הקידומות
  - DWORD PTR
  - WORD PTR
  - BYTE PTR

שיטות מיעון

- מבחר קטן שליטות מיעון (יחסית ל - PDP)
  - אין שיטת מיעון double-indirect
  - אין הגדלה אוטומטית של רגיסטר דרך שיטת מיעון
  - אין גישה לזיכרון בשני אופרנדים של אותה פקודת מכונה
  - צירופי רגיסטרים מוגבלים בגישה לזיכרון:

$$\left[\begin{Bmatrix}\text{eax}\\ \text{ebx}\\ \text{ecx}\\ \text{edx}\\ esi\\ edi\\ ebp\end{Bmatrix}+\text{offset}\right],\qquad \left[\begin{Bmatrix}\text{eax}\\ \text{ebx}\\ \text{ecx}\\ \text{edx}\\ esi\\ edi\\ ebp\\ esp\end{Bmatrix}+\begin{Bmatrix}1\\ 2\\ 4\\ 8\end{Bmatrix}*\begin{Bmatrix}\text{eax}\\ ebx\\ \text{ecx}\\ edx\\ \text{esi}\\ \text{edi}\\ ebp\end{Bmatrix}+\text{offset}\right]$$

דוגמאות

```
mov eax, 01005000h
mov BYTE PTR [eax], 0          // set one byte at 0x01005000 to zero
mov WORD PTR [eax], 0          // ..two bytes..
mov DWORD PTR [eax], 0         // ..four..

mov ebx, DWORD PTR [eax+4]                  // load from [01005004h]
mov ecx, DWORD PTR [eax+4*esi]              // useful for arrays
mov edx, DWORD PTR [ebx+8*edi+0x3030]       // two dimensional array
lea   eax, [eax*8+eax]                      // multiply eax by 9
```

Not possible:

```
mov [eax], [ebx]                // only one indirect access
mov [eax+ebx+esi], 0            // only two registers
mov [eax-ebx], 0                // only addition...
```

חלוקה לסגמנטים

- כאשר ניגשים לכתובת מסוימת בזיכרון, הכתובת היא בסגמנט דרכו ניגשים - cs, ds, ss, es, fs, gs. לדוגמא mov eax, fs:[24h] ייגש לכתובת 24h (כלומר 36) באזור הזיכרון ש-fs מצביע אליו.

- הגישה היא תמיד דרך האוגרים האלו, גם אם לא צוין במפורש
  - לדוגמא mov eax, [esi] הוא קיצור ל-mov eax, ds:[esi]

- בחלונות ב-IA32 רוב אוגרי הסגמנט מפנים לאותו סגמנט
  - כך שבעצם אין שימוש במנגנון הסגמנטציה
  - אוגרי הסגמנטים מכילים אינדקסים לתוך טבלת GDT
    - מכילה כתובות, הרשאות, גודל, וכו' של הסגמנטים השונים
    - חפשו Global Descriptor Table למידע נוסף
    - אנו לא נתעמק ב-GDT ובסגמנטים

קריאות לפונקציות והמחסנית

קריאות לפונקציות

- call operand

```
mov eax, 01005000h
call eax                    // jump to 01005000h
call [eax]                  // jump to the address at [01005000h]
call 01005000h              // jump to 01005000h
```

```
                                sub esp , 4
call ~=  push eip ~=            mov [esp] , eip
         Jump
```

- Returning from a function: ret

```
ret
```

```
ret ~= pop eip ~=  mov eip, [esp]
                   add esp , 4
```

CALLING CONVENTIONS
C CALLING CONVENTION (CDECL)

- פרמטרים מועברים במחסנית מהסוף להתחלה (מימין לשמאל)
- ערך החזרה מוחזר ב-eax
- הפונקציה הקוראת אחראית לניקוי המחסנית
- דוגמא

C:

```
var = someFunction(1,2,3);
If (var…
```

asm:

```
push 3
push 2
push 1
call someFunction
add esp , 0xC
tst eax
…
```

- יתרונות?
  - נוח לפונקציה הנקראת
  - מאפשר מספר פרמטרים משתנה
    - למשל printf("%d%d%c%s\n", a, b, c, str)

CALLING CONVENTIONS

- שיטה אחרת ש-Visual C משתמש בה לעיתים:
  - העברת פרמטר ראשון דרך ecx

```
asm:
    mov ecx, 1
    call someFunction
    tst eax
    …
```

- יש גם שיטות אחרות:
  - Pascal – פרמטרים מועברים משמאל (לפי הסדר, מספר פרמטרים קבוע)
  - חלק מהמהדרים מעבירים פרמטרים גם באוגרים ומשתנים גלובליים
  - ב-++C נהוג שהפרמטרים מועברים במחסנית בעוד שהמחלקה מועברת ב ecx (הכתובת של this)
  - stdcall – נרחיב בשקף הבא.

CALLING CONVENTIONS

C CALLING CONVENTION (STDCALL)

- מאוד דומה ל:cdecl
- פרמטרים מועברים במחסנית מהסוף להתחלה (מימין לשמאל)
- ערך החזרה מוחזר ב-eax
- **ההבדל:** הפונקציה **הנקראת** אחראית לפינוי המחסנית בסיומה
- **כל פונקציות הAPI של windows מוגדרות עם stdcall**
- ניתן לזהות אותן בassembly באמצעות הפקודה retn
- יתרונות?
  - נוח לפונקציה הקוראת (ובדכ למתכנת)

דוגמא

```
push 2          // param 2
push 1          // param 1
call someFunction
add esp,8
```

someFunction:

```
push ebp
mov ebp , esp     // ebp will point to the frame start

sub esp  , x      // make room for local variables
…
code
…

mov esp , ebp
pop ebp
ret
```

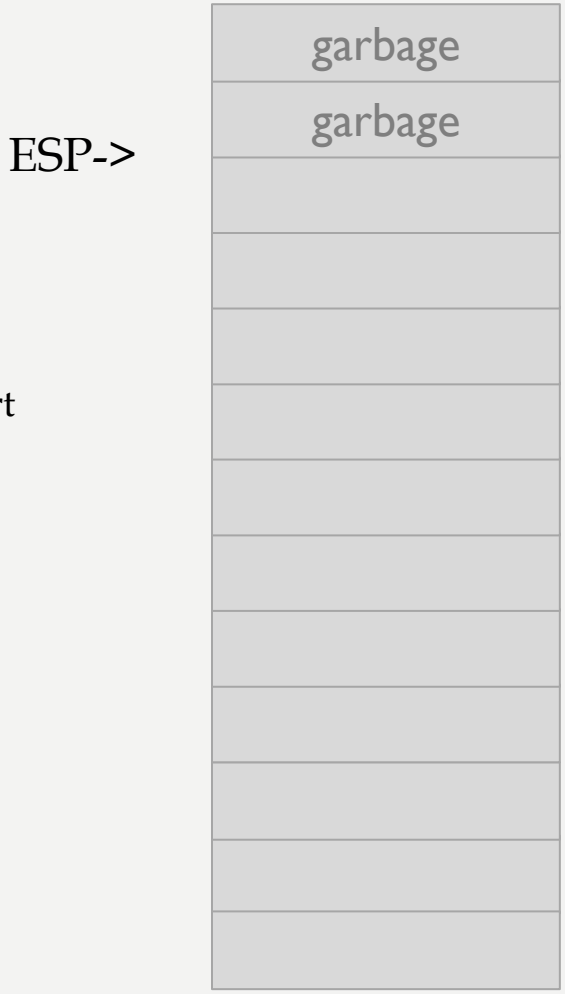
דוגמא

```
→ push 2            // param 2
  push 1            // param 1
  call someFunction
  add esp,8


  someFunction:
  push ebp
  mov ebp , esp     // ebp will point to the frame start

  sub esp  , x      // make room for local variables
  …
  code
  …

  mov esp , ebp
  pop ebp
  ret
```



דוגמא

```
push 2          // param 2
push 1          // param 1
call someFunction
add esp,8


someFunction:
push ebp
mov ebp , esp     // ebp will point to the frame start

sub esp  , x      // make room for local variables
…
code
…

mov esp , ebp
pop ebp
ret
```

ESP->

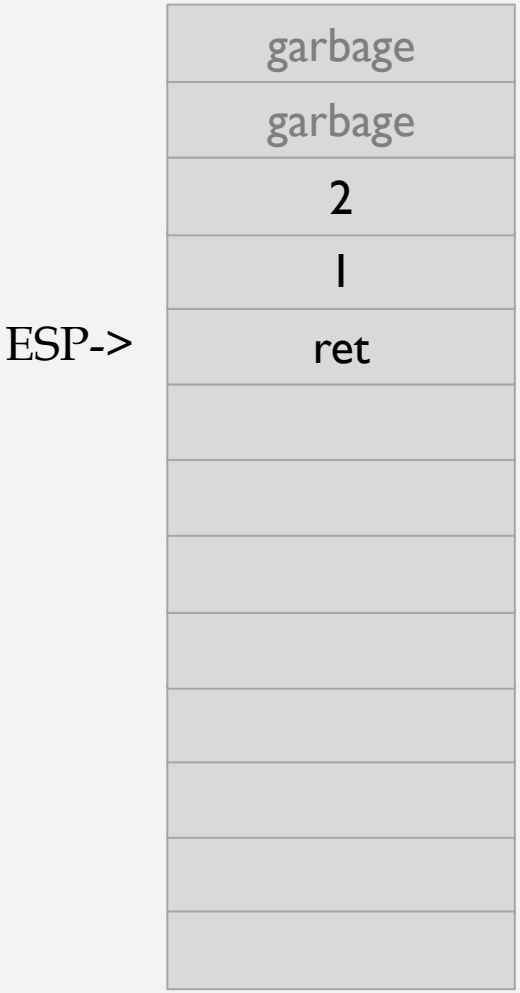
דוגמא

```
push 2          // param 2
push 1          // param 1
call someFunction
add esp,8

→ someFunction:
push ebp
mov ebp , esp     // ebp will point to the frame start

sub esp  , x      // make room for local variables
…
code
…

mov esp , ebp
pop ebp
ret
```



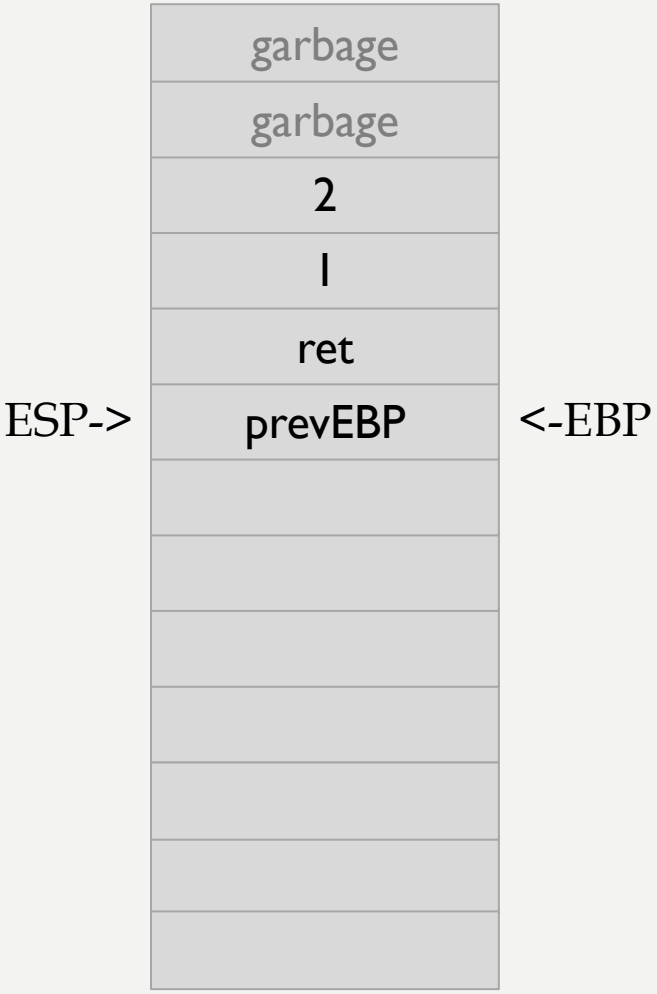
דוגמא

```
push 2          // param 2
push 1          // param 1
call someFunction
add esp,8


someFunction:
push ebp
mov ebp , esp     // ebp will point to the frame start

→ sub esp  , x      // make room for local variables
…
code
…

mov esp , ebp
pop ebp
ret
```



דוגמא

```
push 2          // param 2
push 1          // param 1
call someFunction
add esp,8
```

someFunction:

```
push ebp
mov ebp , esp     // ebp will point to the frame start

sub esp  , x      // make room for local variables
…
→ code
…

mov esp , ebp
pop ebp
ret
```

Note:

You can access the function arguments and the local variable using the EBP register.

- EBP+x stands for arguments
- EBP-x stands for locals

Using ESP is more complex (you should know it from PDP)

depends on the code

דוגמא

```
push 2          // param 2
push 1          // param 1
call someFunction
add esp,8
```

someFunction:

```
push ebp
mov ebp , esp     // ebp will point to the frame start

sub esp  , x      // make room for local variables
…
code
…

→ mov esp , ebp
  pop ebp
  ret
```

Note:

The two commands "mov esp,ebp" and "pop ebp" can be replaced by the "leave" command.

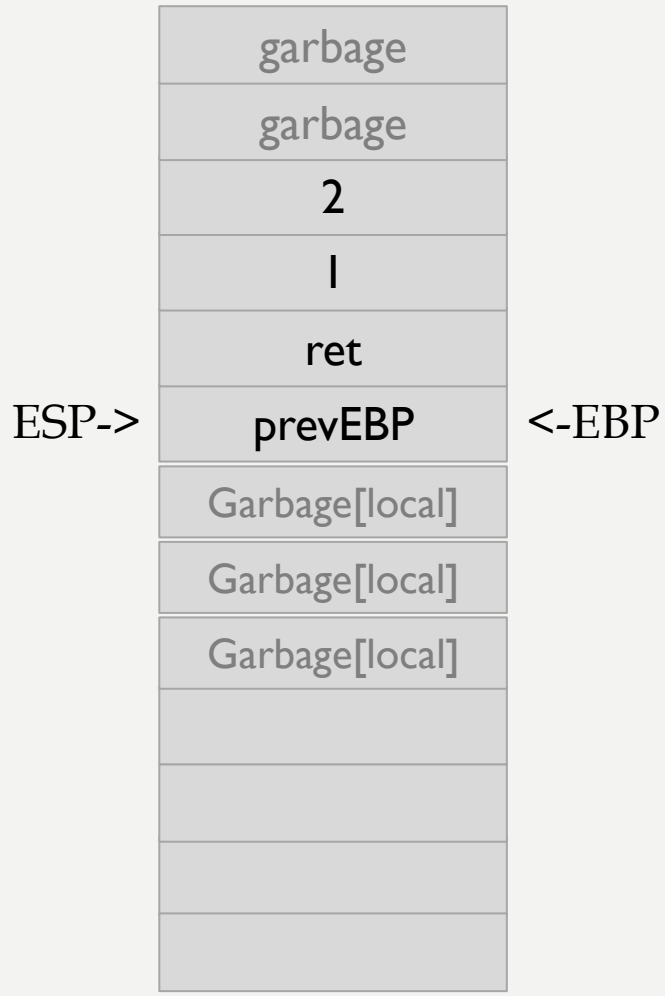
דוגמא

```
push 2          // param 2
push 1          // param 1
call someFunction
add esp,8


someFunction:
push ebp
mov ebp , esp     // ebp will point to the frame start

sub esp  , x      // make room for local variables
…
code
…

mov esp , ebp
pop ebp
ret
```



דוגמא

```
push 2          // param 2
push 1          // param 1
call someFunction
add esp,8


someFunction:
push ebp
mov ebp , esp     // ebp will point to the frame start

sub esp  , x      // make room for local variables
…
code
…

mov esp , ebp
pop ebp
ret
```

דוגמא

```
push 2          // param 2
push 1          // param 1
call someFunction
→ add esp,8


someFunction:
push ebp
mov ebp , esp     // ebp will point to the frame start

sub esp  , x      // make room for local variables
…
code
…

mov esp , ebp
pop ebp
ret
```

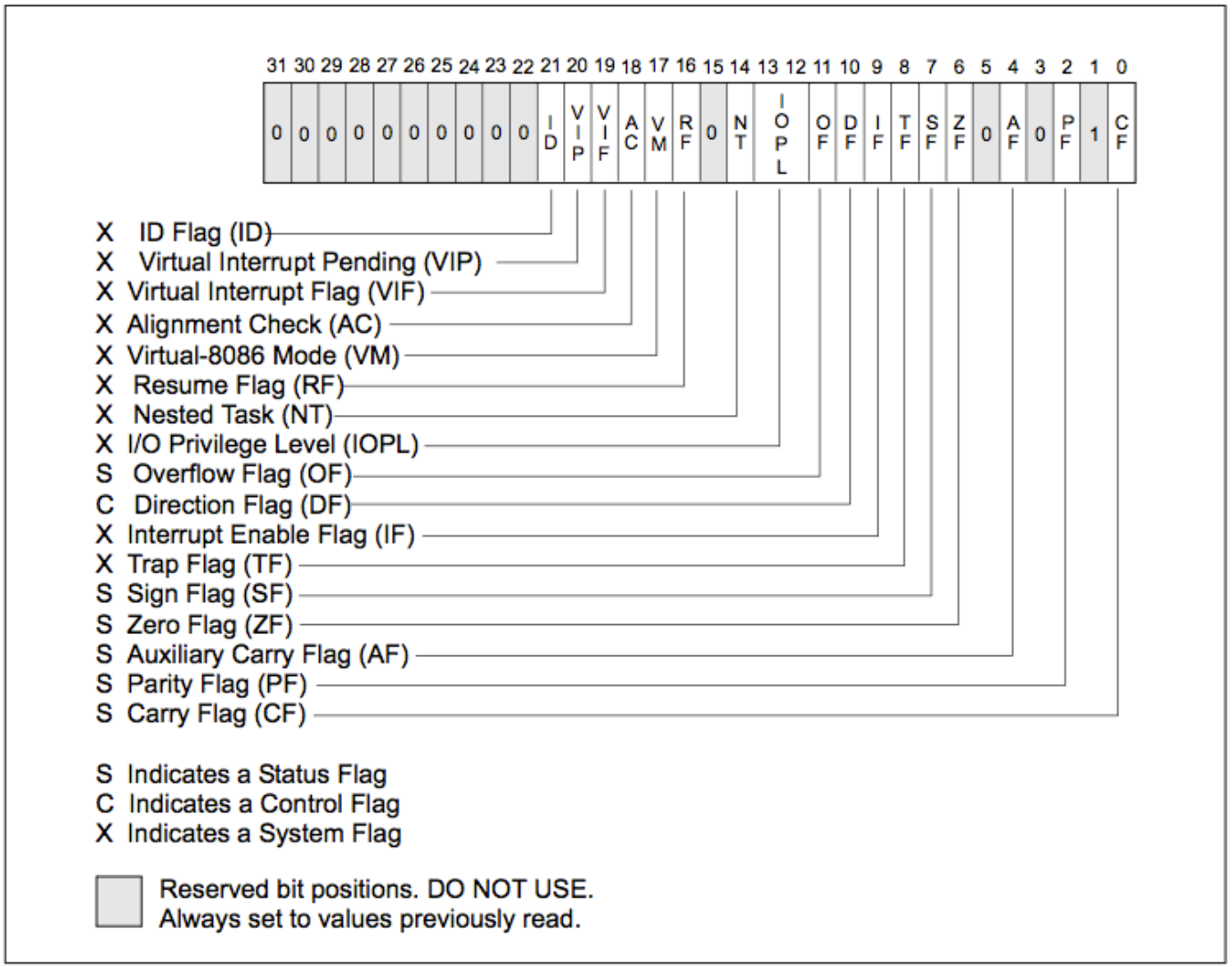
CONDITIONAL JUMPS

רגיסטר EFLAGS

- רגיסטר eflags מכיל סיביות דגלים המציגות מצבים שונים בחישוב, על פיהם ניתן לבצע פעולות מותנות, למשל:

  - ZF (Zero Flag)
    - Set if last result of the operation was zero.
  - CF (Carry Flag)
    - Set if there was a carry or borrow.
  - SF (Sign Flag)
    - Set if the destination is negative.
  - OF (Overflow Flag)
    - Set on overflow.

רגיסטר EFLAGS



קפיצות מותנות

Instruction	Descriptions	eflags
JE	Jump if equal	ZF = 1
JNE	Jump if not equal	ZF = 0
JGE	Greater or equal (signed)	SF = OF
JG	Greater (signed)	SF = OF and ZF = 0
JL	Jump if less (signed)	SF != OF
JLE	Jump if less or equal (signed)	SF != OF or ZF = 1
JB	Jump if below (unsigned)	CF = 1
JA	Jump if above (unsigned)	CF = 0 and ZF = 0
JBE	Below or equal (unsigned)	CF = 1 or ZF = 1
JAE	Above or equal (unsigned)	CF = 0

דוגמא - תנאי עם AND

```
        cmp a,b
        jle code2
        cmp c,d
        jle code2
code1:
        …
        jmp code3
code2:
        …
        jmp code3
Code3:
```

→ ?

דוגמא - תנאי עם AND

```
        cmp a,b
        jle code2
        cmp c,d
        jle code2
code1:
        …
        jmp code3
code2:
        …
        jmp code3
Code3:
```

→

```
if (a<=b) goto code2;
if (c<=d) goto code2;
code1:
..
goto code 3;
code2:
..
goto code 3;
code3:
```

דוגמא - תנאי עם AND

```
        cmp a,b
        jle code2
        cmp c,d
        jle code2
code1:
        …
        jmp code3
code2:
        …
        jmp code3
Code3:
```

→

```
if (a<=b) goto code2;
if (c<=d) goto code2;
code1:
..
goto code 3;
code2:
..
goto code 3;
code3:
```

זה אולי תקין/עובד, אך לא נראה כמו קוד סי לגיטימי שהמתכנת כתב.

דוגמא - תנאי עם AND

```
        cmp a,b
        jle code2
        cmp c,d
        jle code2
code1:
        …
        jmp code3
code2:
        …
        jmp code3
Code3:
```

```
If ((a>b) && (c>d)) {
        code1…
} else {
        Code2…
}
Code3…
```

דוגמא – תנאי עם OR

```
        cmp a,b
        jg code1
        cmp c,d
        jle code2
code1:
        …
        jmp code3
code2:
        …
        jmp code3
Code3:
```

```
If ((a>b) || (c>d)) {
        code1…
} else {
        Code2…
}
Code3…
```

פסיקות

פסיקות בחלונות

- וקטור הפסיקות מפנה את המעבד לכתובות לטיפול בפסיקות.
  - Interrupt Dispatch Table (IDT).
  - השגרה לטיפול בפסיקה מספר x מופנית מהכניסה בעלת אינדקס x בווקטור הפסיקות.
- וקטור הפסיקות מאותחל ע"י מערכת הפעלה.
- השגרה בדרך כלל תרוץ ב-"ring 0" (הרשאות kernel).

פסיקות תוכנה

- ניתן לקרוא לפסיקות גם מתוכנה
- על מנת ליזום פסיקות תוכנה ניתן להשתמש בפקודת המכונה int val
- int val קוראת לשגרה שנמצאת באינדקס שמספרו val ב-IDT
- מאד שימושי לקריאות למ"ה

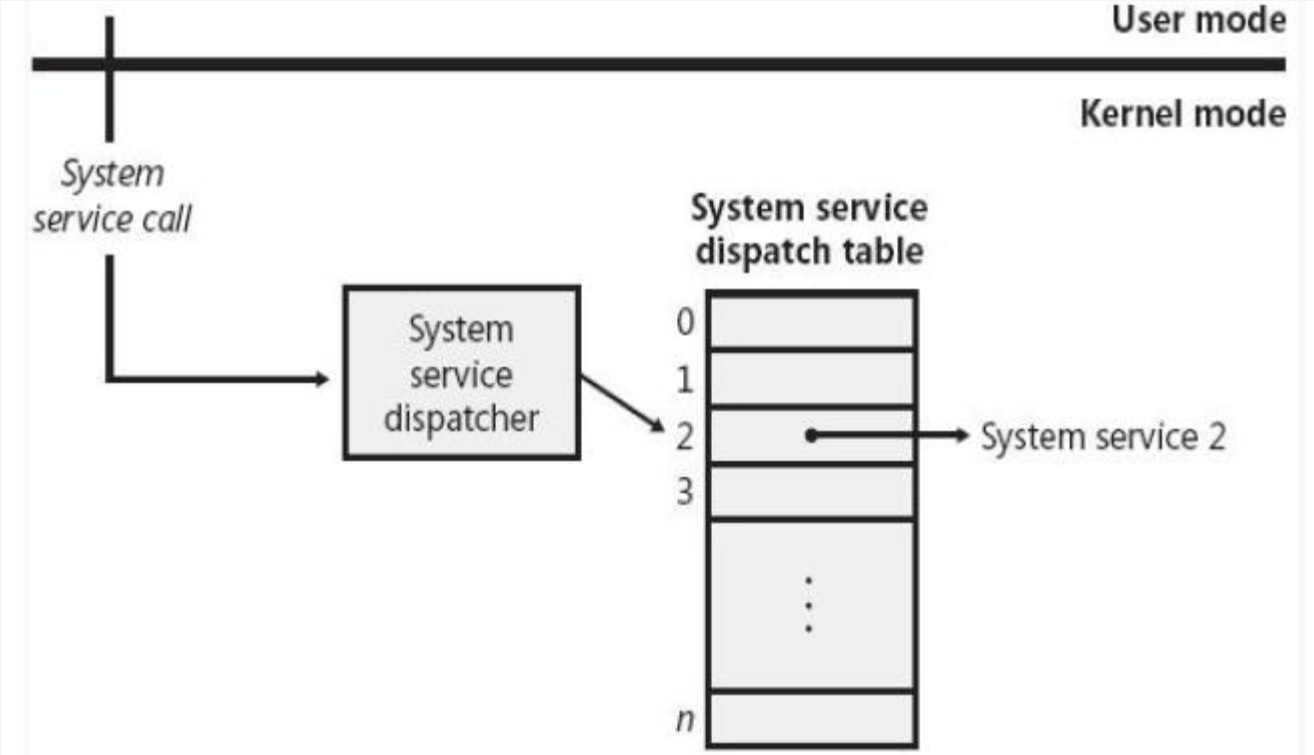
קריאות מ"ה – INT 2E

- פסיקת תוכנה 2E משמשת לקריאה למ"ה.
- לפני הקריאה יש לשים ב eax את מספר הפונקציה וב edx כתובת לפרמטרים.
- השגרה ב IDT באינדקס 2E נקראת KiSystemService.
- השגרה קוראת לפונקציה המתאימה שכתובתה נמצאת בטבלת ה-SSDT (טבלה של מ"ה) באינדקס eax.

SYSCALL/SYSENTER

- כל פונקציה (כמעט) משתמשת בשירות של מערכת ההפעלה.
- שימוש ב-int 2E ומעבר דרך ה-IDT לוקח זמן - נקרא בשם syscall.
- הפקודה sysenter קוראת ישירות לפונקציה הרלוונטית במערכת ההפעלה (kifastSystemCall) ללא שימוש ב-IDT.
- למידע נוסף מומלץ לעיין בפרקים הרלוונטיים בספר Windows Internals.

SYSCALL/SYSENTER



דוגמאות

פקודות חשובות

- add, sub
- mul, div (use fixed registers)
- and, not, xor, or, neg
- shl, shr, rol, ror
- lea
- call, jmp, ret, leave
- Conditional branch: e.g., ja, jl, jne
- rep, repne, repnz +
  - stos, scas, movsd, movsw, movsb
- nop

- *You should know these instructions fluently!*
- *Consult the Intel instruction set – link in the site.*

דוגמא 1

```
encrypt:
        push ebp
        mov ebp, esp
        sub  esp, 16
        mov DWORD PTR [ebp-4], 0
        jmp  .L2
.L3:
        mov eax, DWORD PTR [ebp-4]
        add  eax, DWORD PTR [ebp+8]
        movzx     edx, BYTE PTR [eax]
        add  edx, 1
        mov BYTE PTR [eax], dl
        add  DWORD PTR [ebp-4], 1
.L2:
        mov eax, DWORD PTR [ebp-4]
        cmp eax, DWORD PTR [ebp+12]
        jl      .L3
        leave
        ret
```

→ ?

דוגמא 1

```
encrypt:
        push ebp
        mov ebp, esp
        sub  esp, 16
        mov DWORD PTR [ebp-4], 0
        jmp  .L2
.L3:
        mov eax, DWORD PTR [ebp-4]
        add  eax, DWORD PTR [ebp+8]
        movzx     edx, BYTE PTR [eax]
        add  edx, 1
        mov BYTE PTR [eax], dl
        add  DWORD PTR [ebp-4], 1
.L2:
        mov eax, DWORD PTR [ebp-4]
        cmp eax, DWORD PTR [ebp+12]
        jl      .L3
        leave
        ret
```

```
int encrypt(char *array , int n) {
        for (int i = 0; i < n ; i++)
                array[i]++;
}
```

דוגמא 2

```
mov     eax, OFFSET FLAT:.LC1
lea     edx, [esp+42]
mov     DWORD PTR [esp+4], edx
mov     DWORD PTR [esp], eax
call    scanf

lea     eax, [esp+42]
mov     DWORD PTR [esp+28], -1
mov     edx, eax
mov     eax, 0
mov     ecx, DWORD PTR [esp+28]
mov     edi, edx
repnz scasb
mov     eax, ecx
not     eax

lea     edx, [eax-1]
mov     eax, OFFSET FLAT:.LC2
mov     DWORD PTR [esp+4], edx
mov     DWORD PTR [esp], eax
call    printf
```

→ ?

דוגמא 2

```
mov     eax, OFFSET FLAT:.LC1
lea     edx, [esp+42]
mov     DWORD PTR [esp+4], edx
mov     DWORD PTR [esp], eax
call    scanf

lea     eax, [esp+42]
mov     DWORD PTR [esp+28], -1
mov     edx, eax
mov     eax, 0
mov     ecx, DWORD PTR [esp+28]
mov     edi, edx
repnz scasb
mov     eax, ecx
not     eax

lea     edx, [eax-1]
mov     eax, OFFSET FLAT:.LC2
mov     DWORD PTR [esp+4], edx
mov     DWORD PTR [esp], eax
call    printf
```

```
#include <stdio.h>

int main () {
  char buffer[36];
  scanf("%s" , &buffer);
  printf("%d" , strlen(buffer));
  return 0;
}
```

דוגמא 3

```
functionname:
        push    ebp
        mov     ebp, esp
        push    ebx
        sub     esp, 20
        cmp     DWORD PTR [ebp+8], 1
        je      .L2
        cmp     DWORD PTR [ebp+8], 2
        jne     .L3
.L2:
        mov     eax, 1
        jmp .   L4
.L3:
        mov     eax, DWORD PTR [ebp+8]
        sub     eax, 1
        mov     DWORD PTR [esp], eax
        call    functionname
        mov     ebx, eax
        mov     eax, DWORD PTR [ebp+8]
        sub     eax, 2
        mov     DWORD PTR [esp], eax
        call    functionname
        add     eax, ebx
.L4:
        add     esp, 20
        pop     ebx
        pop     ebp
        ret
```

→ ?

דוגמא 3

```
functionname:
        push    ebp
        mov     ebp, esp
        push    ebx
        sub     esp, 20
        cmp     DWORD PTR [ebp+8], 1
        je      .L2
        cmp     DWORD PTR [ebp+8], 2
        jne     .L3
.L2:
        mov     eax, 1
        jmp .   L4
.L3:
        mov     eax, DWORD PTR [ebp+8]
        sub     eax, 1
        mov     DWORD PTR [esp], eax
        call    functionname
        mov     ebx, eax
        mov     eax, DWORD PTR [ebp+8]
        sub     eax, 2
        mov     DWORD PTR [esp], eax
        call    functionname
        add     eax, ebx
.L4:
        add     esp, 20
        pop     ebx
        pop     ebp
        ret
```

```
int fibonacci(int n) {
  if ((n==1) || (n==2))
    return 1;
  else
    return fibonacci(n-1) +
                    fibonacci(n-2);
}
```