

# מבנה מחשב - סיכום הרצאות למבון

4 בדצמבר 2025

הסיכום נכתב תוך כדי הרצאות סמס א' תשפ"ו (2026) ובכן יתכן שנפלו טעויות תוך כדי כתיבת הסיכום, ככה שהשימוש על אחריותכם.  
גיא ערד-און.

## תוכן עניינים

2	.....	הרצאה 1	1
2	.....	מבוא לקורס	1.1
3	.....	COMPUTER STRUCTURE	1.2
4	.....	מדוע המחשב שלנו ביינארי? למה שהוא טרינארי?	1.2.1
4	.....	Instruction Set Architecture - ISA	1.2.2
4	.....	תהליך הרצאה ותמונה הזכרון . . . . .	1.3
5	.....	איך CPU יודע מה רצף ביטים מייצג? . . . . .	1.3.1
6	.....	Bit Level Operation	1.4
7	.....	תרגול 1	1.5
8	.....	.....	הרצאה 2
8	.....	numbers	2.1
8	.....	CPU Flags	2.2
9	.....	Endianness	2.3
9	.....	Assembly	2.4
10	.....	Registers	2.5
10	.....	Basic Instructions & Data types	2.6
11	.....	השוואה לשפת מכונה . . . . .	2.6.1
11	.....	jmp ו Labels	2.6.2
11	.....	Assembler directive	2.6.3
12	.....	Addressing Modes	2.6.4
13	.....	פקורות בסיסית באסמבלי . . . . .	2.7
14	.....	תרגול 2	2.8
15	.....	Singed&Unsigned	2.8.1
16	.....	מבנה של תוכנית: . . . . .	2.8.2
17	.....	.....	הרצאה 3
17	.....	Jump&Set	3.1
18	.....	declare initialized data	3.2
19	.....	lea הפקודה	3.3
20	.....	C Calling Convention	3.4
20	.....	Stack Operation	3.4.1
20	.....	Calling Convention	3.5

23	.....	<i>Jump Table</i>	3.6
25	.....	<i>GDB</i>	3.7
26	.....	4 הרצתה	4
26	.....	<i>Assemble process</i>	4.1
27	.....	<i>ELF relocatable format</i>	4.2
28	.....	<i>ELF executable format</i>	4.3
29	.....	4.3.1 כיצד כתובים ורוכס?	
30	.....	<i>Disassembled</i>	4.4
30	.....	<i>Linking process</i>	4.5
31	.....	<i>Position Independent Code</i>	4.6
32	.....	תרגול	4.7
32	.....	4.7.1 ארגומנטים בSTACK	
32	.....	<i>Variadic Functions</i>	4.7.2
33	.....	4.7.3 קבלת ארגומנטים בשורת ההרצתה	
33	.....	4.7.4 Flow Control	
34	.....	4.7.4 memory hierarchy	5 הרצתה
34	.....	5.1 הקדמה	
35	.....	5.2 cache	
36	.....	5.3 organization of a cache Memory	
39	.....	5.4 תרגול	
39	.....	5.4.1 static linking	
40	.....	5.4.2 (Position Independent Code) PIC	
40	.....	5.4.3 PLT	
41	.....	5.4.4 Patching	
41	.....	6 הרצתה	6
41	.....	6.1 miss	
43	.....	6.2 RAM structure	
45	.....	6.3 Disk structure	
45	.....	6.4 תרגול	
47	.....	7 הרצתה	7
47	.....	7.1 cache friendly code	
48	.....	7.2 Pipeline friendly code	
48	.....	7.3 RAM friendly code	
48	.....	7.4 compiler & optimizations	
48	.....	8 הרצתה	8
48	.....	9 הרצתה	9
48	.....	10 הרצתה	10
48	.....	11 הרצתה	11
48	.....	12 הרצתה	12

## 1 הרצתה 1

### 1.1 מבוא לקורס

הקורס יתמקד בשני תחומיים:

1. מבנה מחשב - חומרה

2. שפת מוחשב שנוגעת לשירות בחומרה - *Assembly*

מתי צריך להשתמש באסמבלי? כאשר אנחנו רוצים למשל לחשב חישוב מהיר מאוד - בכל מקום

אחר החישוב זהה יתקע. זו שפה שהיא כמעט שפת מוכנה" - *Low Level* -

האם  $0 \geq x^2$ ? לא בהכרח - במתמטיקה כן, בודאי ב $\mathbb{R}$ . בעולם התאורטי, זו טענה נכונה. במציאות: ראיינו כבר כי בהינתן מס' מאד גדול, למשל  $x = 1410065407$ , נקבל כי  $x^2 = 0 < 11158137855$ , מודיע? כמו שמלמד מבואה - יש *overflow* והיצוג הופך לשלייל. אז כיצד נתמודד עם זה?ណון בזאת במהלך הקורס.

האם מתקיים  $(y + z) + x = y + (z + x)$ ? בעולם התאורטי, כן. עם זאת - לא כל מספר עשרוני ניתן לשימירה במחשב. למשל אם כמהות הספרות האחוריenkodaה גדול מכמות הספרות שאפשר להחזיק, המחשב חותך את הספרות שהוא לא יכול לשמור - וזה הוא מקבל מס' שאינו מדויק. ולכן, לא יתקיים השוויון הנ"ל במחשב בשל הסיטיות הללו.

וכאן הדגש: **בתאוריה המתמטית - לא יתכן שיקרו בעיות כאלה.** בעולם האמתי, במחשב: זה לغمרי קורה. ועלינו לומוד בידך להתמודד עם זה.

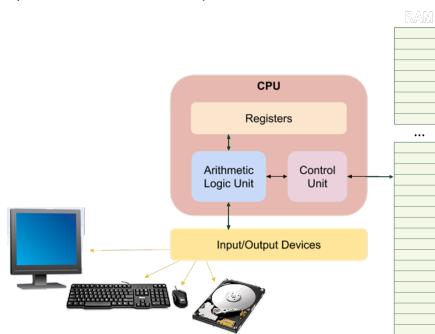
**תבונן בקוד הבא:**

```
for (i = 0; i < 2049; i++)
for (j = 0; j < 2049; j+=3){
    B[i][j] = A[i][j];
    B[i][j+1] = A[i][j+1];
    B[i][j+2] = A[i][j+2];
```

קוד זה רץ מהר יותר מאשר הקוד האינטואטיבי, בו אנחנו רצים ללא קפיזות זה שלוש. הקוד הזה הרבה פחות יפה - אבל בהמשך נבין (*CPU*) שהוא יותר מהר: **זה כל מה חשוב, עילוות.**

## COMPUTER STRUCTURE 1.2

המחשב בנוי מ*CPU*, *RAM*, *Registers* (וחדרון) *(Input/Output Devices)*.



ה*CPU* הוא ה"מוח" של המחשב, הוא יחידת העיבוד המרכזית. *CPU* יש יכולת מתמטית חישובית - *Arithmetic Logic Unit*: *ALU*. *CPU* יש זכרון גם כן - *Registers*: בcludיהם הוא לא מסוגל לעשות כלום והוא תלוי בהם.

**הזיכרון - *RAM*:** מרכיב מבנייטים. כל בית מורכב מ-8 ביטים. מעין מערך" שיכול לספור עד  $2^n - 1$  ערכים. כל בית בזכרון הוא עם ערך כלשהו - גם אם שמו את זה שם וגם אם לא (יקבל גאנק). הביט הימני נקרא *LSB* והביט השמאלי נקרא *MSB*. *Word* - שני ביטים רציפים. *Long/dword* ארבעה ביטים רציפים. *qword* - שמונה ביטים רציפים.

### 1.2.1 מודיע המחשב שלנו בינהרי? למה שהמחשב לא יהיה טרינארי?

כאשר מעבירים מידע ממוקם למקום (בתוך המעבד) נשלח *signal* חשמלי. אנחנו רוצים לתרגם את המידע שיש בסיגנל החשמלי לביטים. סיג널 מגע עם רעשין - כמו כאן בתמונה מטה. מדדרים טובות של עצמת הסיגנל: בין 0 ל-2/0 הוא מתרפרש כביט 0, בין 0.9 ל-1.1 מתרפרש כביט 1, ובכל השאר הוא מתרפרש כמצב מעבר.

באופן תאורטי - יכלו את מצבי המעבר להגדיר כמצב השלישי - וזה לעובוד עם בית" שלישית, במצב טרינארי: מהר מאוד ירדו מרעיוון זה, כיון שהה הרבה יותר רעשיות ותונדות וזה במקומות לחשוב קיבלת מס' 8 קיבלו 9. נוצרו הרבה בעיות כתוצאה מרעיוון זה - ולכן בשביל להבטיח את תקינות

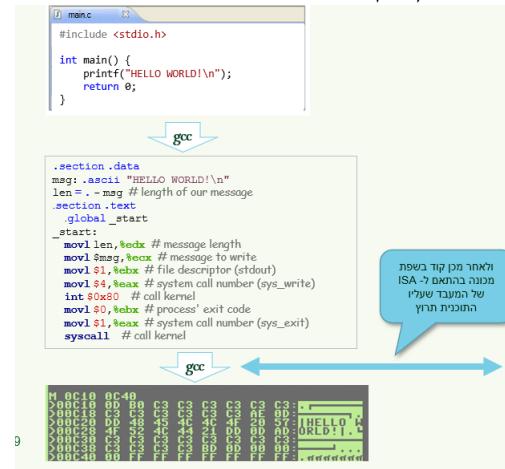


### Instruction Set Architecture - ISA 1.2.2

(מרינה אמרה **שהיא שואלת פעמים במבחן מה זה**). לכל דגם של *CPU* יש אוסף פקודות שהוא ידוע לבצע. אוסף פקודות נקרא שפת מכונה. לכל פקודה כזו קיימת פקודה מקבילה שקיימות בשפת אסמבלי. ה*ISA ספר* שומרכו את כל הפקודות שאחותה ארכיטורה / *CPU* ידוע לבצע. ה*ISA ספר* ידוע לבצע. דוגמים שונים של מעבדים יכול להיות שונה. בהינתן שנדע את הספר הנ"ל - נדע איך פקודות נוכנש לכתוב בשביל לכטוב את הקוד שלנו. הפקודות בספר יהיו כתובות הן בשפת אסמבלי והן בשפת מכונה.

הקומפיילר הוא שיצטרך את *ISA* בשביל לדעת לתרגם את הקוד לשפת אסמבלי. קוד בשפת *C* יתורגם לשפת אסמבלי באמצעות הקומפיילר, ולאחר מכן מנקן *ISA* יעוז לתרגם את הקוד לשפת מכונה.

לדוגמא: תהליך הקומפלול משפת *C* לשפת אסמבלי ומשם לשפת מכונה.



### 1.3 תהליכי ההרצה ותמונה הזיכרון

1. שלב ראשון: כתבו קוד. נקרא לו *Source*

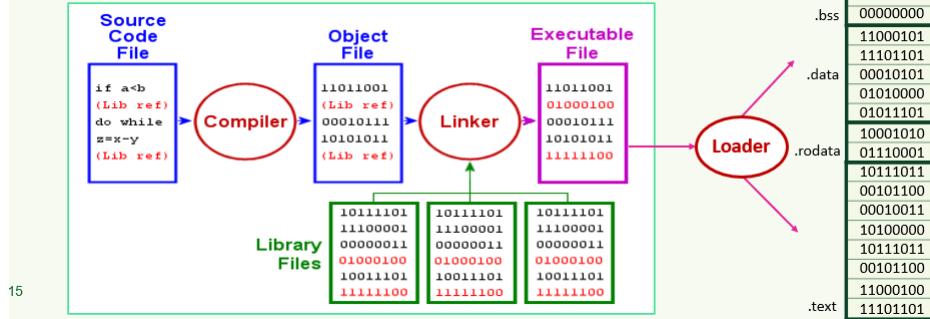
2. שלב שני: הקומפיילר מבצע תהליכי קומפיילציה, מזהה שגיאות קומפיילציה והופך אותו לשפת מכונה.

3. **שלב שלישי:** הילינקר, מבצע תהליכי קישור בין הקוד שליל קודים אחדים שקיים בספריות אחרות בהם אנחנו השתמשנו בקוד, זהו קוד שכבר מוכן ומקומפל - והוא אחד אותם לקובץ יחיד שיקרא *Executable* שמוון ל�מפול.

4. **שלב רביעי:** בטרמינל, אנחנו כתובים למשל *a.exe*.> : למעשה, מה שקרה הוא שאנחנו אמרנו למערכת הפעלה - קחי את הקבצים שכתבתי לך *Executable*, תשתמשי בו *Loader* ("טוען") - תפקידו לטען את הקובץ לארכו. **ראשית** הוא מודוא שקובץ זה ניתן להרצה ע"י מערכת הפעלה שלנו. **שנית**, הוא מפרשר ("חותך ושם בכל מקום בזיכרון מה שצריך לשבת שם") - בדיק כómo שבבדיקת מבחר, הבודק מודלג על אזור ההרואות. **למעשה** יודיע על מה עליו לדגל ומה הוא צריך לארו, את מה שהקובומפイルר והילינקר עזרו לו לקביל, וזה הפרטור) את מה שכתוב לו, והוא צריך להבין באיזה מקום של הקובץ מופיע *data* ומשם את זה באזורי *data* בזיכרון, להבין מה צריך לћhnנס *text* בזיכרון ולשים את זה באותו אזור בזיכרון וכן הלאה). **תקpid נסף שלו** - הוא לאתחל את המשתנים בהתחלה.

cut לאחר שהשתמשנו ב-*Loader* קיבלנו *Process Image*: מכלול של זכרון (תמונה זכרון) שמקורו לטובות התהיליך *Loader* אחראי לו. כלומר, כל הזכרון שמקורו לטובות התהיליך במהלך ריצת התהיליך.

5. OS manages the program execution



**bss:** משתנה גלובלי (מוקצים ב-*data*). כאשר כתבנו משתנה *x int* ולא נתנו לו ערך.

**data:** משתנים גלובליים שמאותחלים כבר עם ערכים.

**rodata:** משתנים גלובליים סטטיים שהם כבר *const* ומסicut רק נקרים.

**text:** אזור הקוד של התוכנית. כמוין ספר". בעת קריאת קוד קוראים מלל.

כאשר כתובים פונקציה, נפתח activation frame בו מוקצים כל המשתנים של הפונקציה, בעט סיום הפונקציה עם *return* הפריים נסגר לטובות הפריים הבה שייפתח.

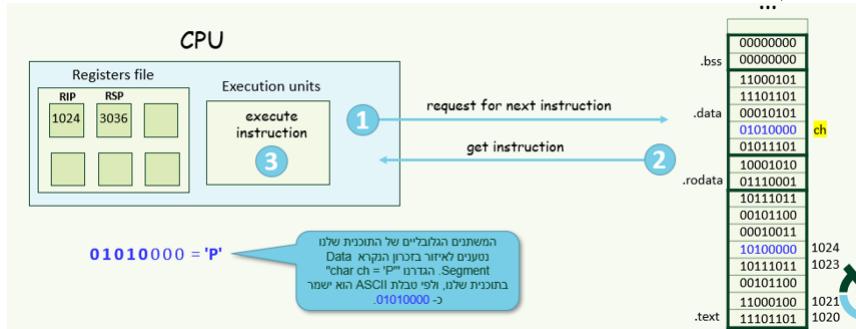
### 1.3.1 איך ה-CPU יודע מה רצף ביטים מייצג?

הCPU מבקש *instruction* הבא, מקבל אותו ומבצע אותו. ברגע שהוא סיים את ההוראה הנוכחית, הוא מקבל את הhabba ומבצע אותו. הוא מאוד מורכב - אך בפועל הוא עובד בצורה פשוטה. *Loader* מכיל מידע אודות היכן *main* ב-*.text*. הוא יודיע את הכתובת *instruction* הראשו שהתוכנית צריכה לבצע - בעברית: נקודת כניסה". *Loader* יזהה במהלך תהליך הפרטור את המיקום הראשון שמננו התוכניתית תתחילה.

מציביע על הבית הראשון של הפקודה הבאה לביוץ. *Loader* שמש בתוך הרגיסטיר משמש *RIP* את הכתובת הזו במהלך תהליך הפרטור. (הערה - גם באסמבלי נוכל לגשת לרегистרים בזיכרון, מה שאי אפשרות בשפות עילית).

לCPU יש ביד את *ISA* - הוא מקבל את הכתובת הראשונה בתוכנית והוא פותח את *ISA*, והוא בודק האם קיימת ב-*ISA* פקודה עם הכתובת זו. אם כן הוא מבצע אותה - ואם לא הוא

מתקדם לכתובת הבאה בזיכרון, הוא מבודד את המיקום בזיכרון באחד (אל הביט הבוא). אם הוא מקבל פקודה - הוא חוזר לשלב 3, ומבצע את הפקודה. לאחר מכן הוא ממשיך להעלות אחד בכל שלב ומתקדם בזיכרון. ...



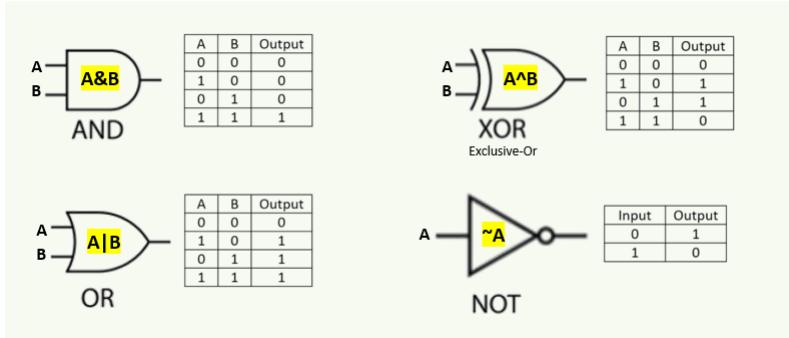
המעבד קורא תחילת (prefix) של פקודה, מזהה שזה פקודה בזאת ה-ISA ומפענה אותה.

**הערכה:** בתוכה CPU ישנו מס' רגיסטרים שיושבים באורו Registers file.Registers file נמצאת בתוכה CPUP (Stack Pointer) RSP. מצביע על תחילתו (כיוון שאם הערך גדול מדי, לפי מבואו הוא נשמר מלמעלה למטרת בaczro) של הערך האחרון שנכנס למחסנית. המחסנית החשובה מאוד כיון שיש בה משתנים לקלט, data, Segment ונתונים לא ASCII וטבלה活性תאion frame וכתובות חורה וכן ארגומנטרי שפונקציה מקבלת (אם וכאר).

**הערכה:** בתוכה CPU ישנו מס' רגיסטרים שיושבים באורו Registers file.Registers file נמצאת בתוכה CPUP (Stack Pointer) RSP, RIP. המשתנים הגלובליים של התוכנית שלו נטען לאורו בaczro הנקרא p', אם הגדרנו אותו ASCII הטענו בתוכנית - לפי טבלת ASCII הוא ישר בהתאם אליה.

## Bit Level Operation 1.4

בහינתן אוסף ביטים, נפעיל פעולות ביוטיז בין ביטים. ישנו מס' פעולות, כדקמן:



נשים לב כי  $1 = \text{true}$  וכן  $0 = \text{false}$  כמובן.

פעולות Shift Right: מזיז ביטים ימינה.  
פעולות Shift Left: מזיז ביטים שמאליה.

דוגמא. **יצירת קבוצה באמצעות ביטויי.** נרצה ליציג קटור באורך  $n$  באמצעות ביט:  $\{1, \dots, n\}$  באמצעות ביט. כל ערך בקטור יכול להיות משוויך לקבוצה ויכול להיות שלא. הקבוצה תהיה  $A$ . אם  $i \in A$  אז  $a_i = 1$ . כלומר, איבר בקבוצה מיקומו בקטור יהיה 1.

אם נרצה להוסיף ערך לקובוצה - נבצע פעולה *or*: מודיע? נסיף ביט עם הערך 1 על 1 וכל השאר אפסים, וcutת כשנעשה *or* נקבל קובוצה חדשה עם 1 בתוכה.  
 אם נרצה למצוא את הקובוצה המשילמה -  $\bar{A}$ : נבצע *not* על הביתו.  
**לחיוך שתי קבוצות** - נשתמש באופרטור *,and*, **ולאייחוד שני קבוצות** - נשתמש באופרטור *or*.  
**חשיבות לדעתך:** פעולות בינהיות המהירות ביותר שניתן לבצע.

## 1.5 תרגול 1

ישנם מס' כלים שנעמדו להקל את החיים של המתכנת.  
 סבבב עבדה שגם ניתן לכתוב בה את הקוד גם להריץ אותו, למשל: *Nano*, *NotePad*, *Vim*, *Emacs :Text Editor*.  
*Clion*, *VsCode*, *VS*: **IDE**  
 סט כלים שמכיל בתוכו כמה שלבים שהשלבים הללו יחד מעבירים תוכנית שתכתבנו **Compiler**  
 מקובץ קוד לקובץ הרצה שנוכל להריץ על גבי המחשב. עובדים עם *компилятор* *GCC*  
**בתוך פרויקט** מועברים כמו *צרייך* במלחץ ייצור קובץ ההרצה. כמו מודא לכל הקבצים *Project Builder*  
*Makefile*, *Cmake* וכו' במאפשר ייצור קובץ ההרצה. כמו *דריכים* בהם ניתן לבצע משימות באמצעות המחשב שלנו. כמו *Computer Interfaces*  
 למשל משמש לפתיחה וסגירת חלונות. *GUI*, *GUI*, *Shell*

:*Shell*

תוכנה שבמיסגרטה אנחנו יכולים להכנס פקודות ומערכת הפעלה מריצה אותן בפועל לאחר שמקבלים פלט על הפקודה. פקודות מרכזיות:  
 1. *ls* - מראה לנו איזה קבצים/*תיקיות* קיימים בתחום התקיה שאנו נמצאים בה.  
 2. *pwd* - מראה את הנתיב בו אנחנו נמצאים כרגע. למשל *.guy/desktop*  
 3. *mkdir* - מאפשר ליצור תיקיה חדשה.  
 4. *cd* - מאפשר לעבור בין תיקיות.  
 5. *touch* - יוצרת קובץ חדש.  
 6. *cp* - מאפשרת להעתיק קובץ.  
 7. *rm* - מאפשרת למחוק קובץ.  
 8. *history* - מראה לנו את הפקודות שהרכינו עד כה.

:*Vim*

עורץ טקסט שקיים בסביבת עבודה של לינוקס. השימוש בו נעשה באמצעות המקלדת בלבד, ללא העכבר. כל פעולה שאפשר לעשות על קובץ, קיים עבורה קיצור מקלדת כלשהו שמוע את השימוש בעכבר.  
 מדוע זה רלוונטי אלינו? אם מתחברים מרוחק לשרת לינוקס כלשהו, למשל לשרת של המחלקה: אין לנו אפשרויות *GUI* של פתיחה וסגירת חלונות. אם נרצה לעורך קובץ לשרת, נוכל לפתח אותו עם *Vim*.  
 ישנה גרסה מתקדמת של *Vim*, *neoVim*, שנכיר.

:*MakeFile*

כל שmorכב מכמה וכמה חלקים וחוקים, שהמטרה של כל החוקים הללו יחד היא לוודא שכל הקבצים הRELIONUTIMNS בפרויקט שלנו מעורבים ביצירת קובץ ההרצה. זה קובץ שמאוד נפוץ ביצירת קבצי *C/C++*  
 מרכיב מיאוסף של *rules* שכל אחד מרכיב משלווה אלמנטים:  
 (commandn *target* .1) לרוב קובץ שנוצר כתוצאה מהרצת *commandn*.

פקודה שתoruץ במהלך הריצה של אותו rule .2 –*command* קבצים שורכים להיות קיימים בשביל rule של אותו dependency .3 –rule

כשריצ' את המילה "make" מה שקרה הוא rule הראשון בקובץ makefile והוא זה שירץ.

## 2 הריצה 2

### 2.1 עוד על numbers

**מספר בינארי:** נזכר כי בהינתן מס'  $U$  המוצג בצורה בינארית, מתקיים  $U(X) = \sum_{i=0}^{n-1} x_i x^i$  באשר  $U = x_0 x_1 \dots x_{n-1}$ . שיטה זו נקראת  $T(x) = -x_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$ . מספרים שליליים: במקרה זה יתקיים  $T(x) = -x_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i = (-X) + X$ . כיצד עושים כן? מכפילים את ה-MSB בחזקה ומוסיפים סימן שלילי ומוחברים את השאר כמו במספרים חוביים.

### 2.2 CPU Flags

באשר CPU מבצע חישוב של חיבור הוא לוקח ביטים של האופרנד הראשון, ביטים של האופרנד השני ומחבר אותם מוביל לדעת האם החיבור הוא signed או unsigned. חיברנו שני מספרים, כמו בדוגמה כאן מטה: כיצד נדע אם התוצאה שקיבלנו נכונה או שנגיה?

<b>Example 1:</b>	$\begin{array}{r} 01000000 \\ + 01000000 \\ \hline 10000000 \end{array} \quad \begin{matrix} \leftarrow 64 \\ \leftarrow 64 \\ \leftarrow 128 \text{ or } -128 ?? \end{matrix}$
-------------------	---

ב-unsigned נקבל כי התוצאה 128, ב-signed נקבל -128, מה קיבלנו כאן? CPU בעצם לא יידע. בחישוב unsigned החישוב נכון – כי אכן  $64 + 64 = 128$ . נשים לב כי MSB של שני המספרים היא 0, ולכן שני המספרים הינם חוביים. וכך נקבל 128 – התוצאה נכונה.

כלל: CPU ביצע חיבור של ביטים, האם התוצאה היא signed או unsigned ישמור מקום מסוים האם הייתה החלפת סימן, אם לא מעוניין אתכם (אתם מחשבים unsigned) פשוט אל תסתכלו במידע.

**דוגמא:** בדוגמה כאן מטה נוצר overflow, שכן קיבלנו אפס. אם היינו ב-signed תוצאה נכונה אחרת, ב-unsigned תוצאה שנגיה. היה carry. נשים לב כי אם היה חיבור של שני סימנים שונים – אפס ואחד, cpus לא צריך לזכור החלפת סימן, הוא רק יזכור האם carry היה או שלא היה.

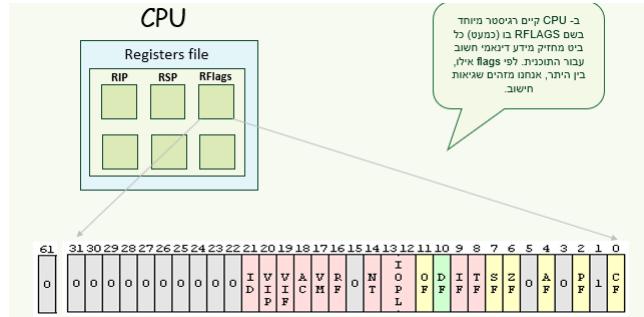
$\begin{array}{r} 11111111 \\ + 00000001 \\ \hline 100000000 \end{array} \quad \begin{matrix} \leftarrow 255 \text{ or } -1 ? \\ \leftarrow 1 \\ \leftarrow 0 ?? \end{matrix}$
--

**היכן הCPU רושם רישומים אלו?**: ישנו גנרטור מיוחד ש-*Rflags* שומר בשם *Rflags* – יש לו ביטים שמתעדכנים בעט חישוב שהCPU עושה. הוא מעדכן את הביטים תוך כדי החישובים. אנחנו מותעניינים בביטים הכהובים. נשים לב שכל ביט ישנו שםCAN. **שנו ביט בשם CP(CarryFlag)** – CP קיבל אחד באשר יהיה carry (בחיבור או בחיסור) אחרית, קיבל אפס. יאמר (אם וכאשך) כי חישוב unsigned שנגי.

**ישנו בית בשם** *OF*(OverflowFlag) הוא יקל 1 כאשר תהייה ההפצת סימן (כלומר הביט שינה סימן), אחרת יקל ערך אפס. אמרו (אם ואחר) כי חישוב *signed* שגוי.

**ישנו בית בשם** *ZF*(ZeroFlag) אם תוצאה החישוב האחרון יצא אפס הוא יקל אחד, אחרת הוא יקל אפס.

**ישנו בית בשם** *SF*(SignFlag) לוקח בית סימן של תוצאה ומעתיק לביט. כלומר לוקחים את *SF* ומעתיקים אותו ל*MSB*.



### Endianness 2.3

ארכיטקטורה של המחשב יכולה להיות *Big – Endian* או *Little – Endian*. אם יש לנו ערך נומירי - מספרי, לא מחרוזת, והערך הנומרי הזה תופס יותר bytes אחד (כלומר לא *char* בלבד) וכן לא במקומות 00. ב-*Big Endian* שומרים אותו משמאלי לימין (שומרים את *MSB* הכי למטה - כלומר נשמר במקומות 00). ב-*Little Endian* שומרים אותו מימין לשמאל (יכיז נזכיר? שומרים את *LSB* הכי במקומות 00). כלומר שומרים במקומות 00).



אנחנו בקורס לומדים לפי *Little Endian*. (כasher אנחנו מדפיסים בקורס משהו, זה מתבצע בהדפסה מכתובות 00 כלומר מה שיודפס קודם יהיה *the LSB*

### Assembly 2.4

asmobilי נוצרה בשבייל לנשות להפוך לכותב בשפת מוכנה, ולנותות לתרגם את השפה לשפת בני אדם. שכחtabו את השפה לא חשבו על נוחות. על כל פקודות מוכנה, לקחו את הפקודת ו"תרגמו" אותה לשפת asmobil שיכאורה יותר אנושית. ומכאן המשקנה: **פעולה באסמבלי=פעולה של שפת מוכנה**.

ישנם כמה סטיילים של כתיבה סינטקטית באסמבלי, בקורס נלמד *AT&T* ישנו סטייל של *Intel*.

**אסמבלי לא תומכת בדברים הבאים (ובמה משתמש במקרים?):**

- א. כן יש - גדים, 4 ביטים, 1 ביט(...).
- ב. מערכים (nocell לעובד שירות עם *RAM*)
- ג. תנאים (גישה ישירה לרגיסטרים - זהה השפה היחידה שמאפשרת זאת, וכך **אסמבלי היא השפה הפייעלה**).
- ד. LOLAOT
- ה. פונקציות
- ו. ספריות סטנדרטיות

מה שנקרה - בהצלחה תהיה לנו.

## Registers 2.5

כפי שדנו קודם לכן, ב-*CPU*, ישנו איזור בשם *registers file*: **רץ של 64 ביטים**.  
ישנם רגיסטרים נוספים שנדרשו בהם כעת.

**רגיסטרים לחישוב כללי** :*general purpose registers* היגיסטרים הינם *RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, R8, R9, ..., R15*. נוכן השתמש בהם לחישובים כלליים. הסיבה לכך שהוא הוא שם נוסף נושא בהרחבה של אסמבלי, שכבר הבינו שלশמות אין משמעות. בעבר היה חסר רגיסטרים ולאחר מכן רגיסטר היו מאחסנים שני נתונים. למשל ברגיסטר *ax* היה שני תכני מידע שונים - *ah, al* ובהתחממה *Low, High*. בהמשך, הרחיבו את הרגיסטר ל-32 ביט ולא נתנו שם לה Mishuk המידע וגם בהרחבה ל-64 ביט לא הוסיףו שם. כך נראה רגיסטר - אוסף בזיכרון של 64 ביטים.



ישנם המונחים **REGISTERS** - בקורס אנחנו משתמשים ברגיסטרים שמצוינו לעיל, ברגיסטר *RIP* וברגיסטרים *Rflags* של נאכל ברגיסטרים הבאים:

**RIP** (Instruction Pointer) הינו הרגיסטר הראשון של ה-CPU לא יכול בלבדיו. זהו רגיסטר שמצויב על הבית הראשון של הפקודה הבאה לביצוע. **Loadern** שם בתוך הרגיסטר מושך RIP את כתובת הזו במהלך תהליך הפרסור. (הערה - גם באסמבלי נוכל לגשת לרגיסטרים בזיכרון, מה שאי אפשר בשמות עליית).

**RSP** (Stack Pointer) - רגיסטר שנמצא בתוך CPU. מצבע על תחילתו (כיון שאם הערך גדול מדי, לפי מבואו הוא נשמר מלמעלה למטרה בזיכרון) של הערך האחרון שנכנס למחסנית. החסנית חשובה מאוד כיון שיש בה משתנים לוקלים, כתובות חרזה וכן ארגומנטרים שפונקציה מקבלת (אם וכאשא).

## Basic Instructions & Data types 2.6

שורת קוד **טיופסית באסמבלי נראה כך:** *movb \$0x61, al*. הפקודה זו מכניסה את הערך 0x61 לתוך הרגיסטר *al*. לאחר הפקודה, הרגיסטר *al* יראה כך:

$$al = 01100001$$

**נדגש:** רק *al* מכניס אל עצמו ערכים, לא כל הרגיסטר משתנה.

נשים לב: \$ חשוב מאד, ואם לא נכתב אותו יחשבו שאנו מדברים על מקום בארכון. אם נכתבם עם \$ יהיה ברור כי הכוונה לערך.  
כשנרצה לפרט לרגיסטר - נעשה זאת עם % בפניה לפני השם של הרגיסטר.

**הפעולה mov:** פקודת זהה. מוסיפים לסוף הפקודה סימות, בהתאם לנודל הטיפוס שאנו מזיאים.  
אנו מעשה מבעיים העתקה של ערך למקום אחר ביכרונו.  
1. הפקודה *movb*: פעולה הזה שעובדת על *bytes*, מזיאים בית מוקם. בית הוא 8 ביטים.  
2. הפקודה *movw*: פעולה הזה שעובדת על *words*, מזיאים *words* מקום מקום בארכון.  
3. הפקודה *movl*: פעולה הזה שעובדת על *longs*, מזיאים *long* מקום מקום. הוא 4 בתים. (כמו *int* ב-C)  
4. הפקודה *movq*: פעולה הזה שעובדת על *qword*, *qword* הוא 8 בתים. (כמו *Long* ב-C)

### 2.6.1 השוואה לשפת מוכנה

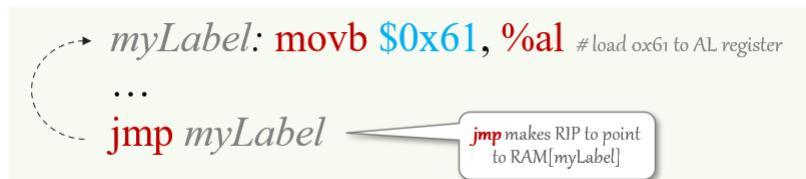
אותה דוגמה מלמעלה, בשפת מוכנה תתרגם להיות השורה הבאה:

0xB061

זה הערך המספרי אותו צריך להכניס, 0x מעיד שאנו בקשר הדצימלי, ו-B0 מסמן לו שהוא צרך לבצע הכנסה של ערך לתוך רегистר *al*. (מאייה אנחנו יודעים זאת? נפתח ISA).  
הערך הוא ערך נומירי. *Imm8* הוא *Immediate*.

### jmp ו-labels 2.6.2

בכדי לשים הערות על הקוד באסמבלי - נוכל להשתמש בנקודה פסיק או #. נוכל להוסיף לכל פקודה זו מקום של הפקודה *Label*. *Label* אשר נסיף את הליבל, הקומפיילר יתרגם אותו כתובות של הפקודה. **מאתורי הקלעים:** הקומפיילר מוחק את הליבל ומוכנס את הכתובת. בשביל מה נctrar ליבלים? כתע, נוכל לבצע *jmp* ולהגיע אל הפקודה זו, מקום אחר בקוד.  
*jmp* משנה את רегистר *RIP* אל *RAM[Label]*



### Assembler directive 2.6.3

הנחיה שאנו נתונים לקומפיילר של אסמבלי, ששמו *Assembler* נקבע על הפקודה הבאה -

buffer: *.skip 4* # reserve 4 bytes

קומpileר, תולך לזכרו, ספציפית ל-section של משתנים גלובליים, בפרט אל *bss* (המשתנה לא מאותחל), ותקצה לי שם רציף של 4 בתים. מעכשו, נוכל להתייחס אל 4 הבטים הללו כ*buffer*. זה כמובן מקבל ליצירת משתנה ללא אתחול, וכן מקבל למערך של *chars*. וכן מקבל ליצירת מערך של שני *shorts*, או מערך של *char short* ושני - וכך כל קבוצה של משתנים שנסכמת לנודל של 4 בתים. בעת, כאשר נוצרה להנגיש את הערך 2 אל המשתנה *buffer* נעשה זאת באמצעות הפקודה הבאה:

```
movl $2, buffer
```

#### Addressing Modes 2.6.4

כעת, נראה כיצד מותרגם קוד בשפת *C* לאסטלבלי: בתחילת אונחנו מקצים בזכרו 20 בתים (5 פעמיים 4 בתים של *int*), אנחנו מתחילה משתנה ברגיסטר *rbx* שיקבל את הערך אפס שיצין למשה את *i = 0*. משם אונחנו מתחילים לולאה, *baseReg* בasmbleyi יש פקודת השוואת (*cmpq*, *baseReg*, *sorce*) מה שהפעולה עשויה הוא חישוב אמצעי (*destination*, *source*), כלומר הפוקודה הבאה תתרגם להיות

```
cmpq $5,%rbx
```

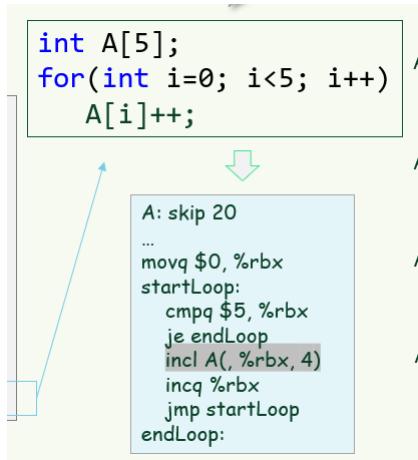
כלומר נבצע  $5 - rbx$ . נשים לב שתוצאת החישוב לא נשמרת. אם קיבל כי התוצאה חיובי, המשמעות היא כי *dest > sorce*. אם שלילי. *dest < sorce*. כיצד נדע האם יש קשר של  $\leq, \geq$ ?  
כלומר - גם שווה, נסתכל על *ZF*.

**הפעולה je:** אם יש שוויון כלומר  $1 = ZP$ , בין שני האופרנדים שבוצעו בשורת הקוד הקודמת, אנחנו נבצע *jmp Label* שכותוב לאחר *je*. הפוקודה הניל' ניגשת אל *zeroFlag* ובזוקת, אם הוא 1 היא קופצת אל *Label*.

**הפקודה inc:** הפקודה למשה מבצעת *inc*, *increasement*, יש שמבצע זאת על *long*. וכמוון כמו ב*mov*, בהתאם לכל סוג משתנה. הסינטקטש שלה יהיה כדלקמן

```
incl A(%rbx,4)
```

כאשר אונחנו כתבים סוגרים באסטלבלי - אנחנו ניגשים אל האזכור. אנחנו הולכים אל *RAM* למקום שהוא  $4 * rdx + 2048$  באשר  $2048 = A[0]$  הכתובת של *A*. אנחנו למשה הכתובת של *A[0]* ואונחנו מגדילים אותה, בשבייל להתקדם לכתובת הבאה (שכבר שמרנו כשהקנו 02 בתים).  
כאשר נבצע *inc* על ערך, נוסף %. כך למשל הפקודה *incq %rbx* תגדיל את הערך *rbx* באחד.



## 2.7 פקודות בסיסיות באסמבלי

*ADD*: הפעולה מוחברת שני שלמים. נשים לב כי אסור שני הארגומנטים של הפקודה יהיו בזיכרון, אחד מהם חייב להיות *immediate* (קבוע) או רגיסטר. סינטקטיס יראה כך -

`addq %RBX,%RAX`

*SUB*: הפעולה מוחסרת שני שלמים. נשים לב כי אסור שני הארגומנטים של הפקודה יהיו בזיכרון, אחד מהם חייב להיות *immediate* (קבוע) או רגיסטר. סינטקטיס -

`subq %RBX,%RAX`

*NOT*: פעולה ביטויה. המשלים ל-0" - הופך בית 1 לbit 0 ובית 0 לbit 1. סינטקטית -

`%AL`

*notb %AL*: ביטויה. פעולה המשלים ל-2", הופך את כל הביטים ומוסיף 1 לתוכאה. *NEG*

*AND*: פעולה ביטויה. בית במקומות *i* מקבל את הספירה 1 אם שני הביטים של הריגיסטרים הוא 1. אחרת, מקבל אפס. סינטקטית -

`andb %BL, %AL`

*OR*: פעולה ביטויה. בית במקומות *i* מקבל את הספירה 1 אם לפחות אחד מהביטים של שני הריגיסטרים הוא 1. אחרת, מקבל אפס. סינטקטית -

`orb %BL, %AL`

*INC*: מגדיל את הערך, שקול לפעולה `++`. סינטקטית -

`incq %RAX`

*DEC*: מחסיר את הערך באחד, שקול לפעולה `--`. סינטקטית -

: פקודת  $CMP(Compare)$  משמשת להשוואה בין שני ערכים באSEMBLY. הפקודה מבצעת חישור בין שני האופרנדים  $A - B$  או  $CMP(A, B)$  מחשב את  $A - B$  לא שומרת את התוצאה - היא רק מעדכנת את דגלי הסטטוס(*flags*) כמו  $ZeroFlag(ZF)$ ,  $SignFlag(SF)$ ,  $CarryFlag(CF)$ ,  $OverflowFlag(OF)$ . דגליים אלה משמשים את פקודות הקפיצה המותנית (*conditional jumps*)  $JE, JNE, JG, JL, JGE, JLE$ .

וכו' כדי לקבע אם לבצע קפיצה או לא.

אם ההשוואה ייצאת שלילי, הפלאג  $SF$  מקבל את הסימן 1 ואז יודעים כי  $A < B$ , אם יוצאת חיובי הפלאג  $SF$  מקבל את הסימן 0 ואז יודעים כי  $A > B$ . אם קיבלו גם כי  $1 = B$  אז  $ZF = 1$ .

: מבצעת פעולה  $AND$  על שני אופרנדים אך לא שומרת את התוצאה - רק מעדכנת את דגלי הסטטוס. שימוש נפוץ לבדיקה - האם גנייסטר שווה לאפס:

```
TEST R1, R1 ;  
JZ label ;
```

למעשה ישנה פעולה  $AND$  על אותו אופרנד עם עצמו. אם הוא היה אפס, נקבל כי  $ZP$  כתידלק. וכן קפוץ *Label* אם  $R_1 = 0$ .

: פקודות הזהה של ביטים. בביצוע  $SHL, SHR, SHIFT$  שיוצאים מהמקום נזקים ונשמרים ב- $CF$ , מימין לנכנים אפסים. בביצוע  $SHR$  כל בית זו ימינה, ביטים שיוצאים מהמקום נזקים ונשמרים ב- $CF$ , משמאל לנכנים אפסים.

- שיפט אריתמטי, בשיפט רגיל אנחנו מזינים ומוסיפים אפסים. יתכן כי המספר 0100 (4) קיבל שיפט לשמאלי, וכתוצאה לכך יופיע מספר 1000, שהוא מייצג מספר שלילי (-16).

והרי זה לא הגיוני שחילקנו בשתיים מס' חיובי וקיבלו מס' שלילי. בדיקוזו הסיבה שנשתמש בשיפט אריתמטי -

$SAL$  זהה לחולוטן  $SHL$ . עם זאת,  $SAR$  מזין לימיינ את הביטים, אך הוא משאיר את בית

הסימן בצד שמאל. ככלומר: הוא לא מזין את בית הסימן.

**דוגמה** - 1101 אם לבצע עליו  $SHR$  נקבל 0100, ממש' שלילי קיבלו חיובי, זה לא טוב. לעומת זאת אם נעשה  $SAR$  נקבל 1100 (הסימן נשאר).

: כפל בין שני מספרים *unsigned*. נשים לב כי *imul* זה למצב שיש *signed*.

: חילוק בין שני מספרים *idiv*, *unsigned*. זה במצב *signed*.

## 2.8 תרגול 2

כיצד מאפסים רגיסטרים? מבצעים לרגיסטר *xor* עם עצמו. שורת הקוד הבא תczę את הרגייסטר:

`xorq %rbx,%rbx`

נוח לאפס את הרגייסטרים בתחילת הריצעה.

**רגיסטר** הוא חומרה שצמוד במעבד - ולכן הגישה אליו הרבה הרבה יותר מהירה.  $X86 - 64$  מכיל 16 רגייסטרים, כל אחד בגודל 64 ביט.

מטרת הרגייסטר *RAX* הוא להחזיר ערך חזרה מפונקציות.

*RSP* הוא לשימוש המחשנית, *RBP* הוא מצביע על ראש המחשנית. לא להשתמש ברגיסטרים אלו שלא למטרת המחשנית.

$A + reg' + i \times reg''$ , כאשר  $i \in \{1, 2, 4, 8\}$  : *Addressing Modes*

$$A + reg' + i \times reg''$$

```

movl $1, 0x604892 # address is constant value (RAM[0x604892] = 1)
movl $1, (%rax) # address is in register %rax (RAM[%rax] = 1)
movl $1, -24(%rbx) # address = -24 + %rbx (RAM[%rbx - 24] = 1)
movl $1, 8(%rax, %rdi, 4) # address = 8 + %rax + %rdi * 4 (RAM[8 + %rax + %rdi * 4] = 1)
movl $1, (%rax, %rcx, 8) # address = %rax + %rcx * 8 (RAM[%rax + %rcx * 8] = 1)
movl $1, 0x8(%rdx, 4) # address = 8 + %rdx * 4 (RAM[8 + %rdx * 4] = 1)
movl $1, 0x4(%rax, %rcx) # address = 4 + %rax + %rcx (RAM[4 + %rax + %rcx] = 1)

```

תמיד כאשר אנחנו רואים סוגרים, מתייחסים להז *Addressing Modes* ומחברים לפי החישוב לעיל. ניתן לחסמי חלק מהפסיקים, לא חובה שכולם ישתתפו.

נשים לב - ניתן להעביר מידע *eax* למשל ( 64 ביט), נשמר בחלק התיכון של הריגיסטר ובחולק העליון יהיה זבל. לעומת זאת אם נעשה *rax, rax* זה ישמר בחלק התיכון ויאפס את החלק העליון.

הפקודה *mov* יכולה להעביר מידע מריגיסטר לריגיסטר, ומרגיסטר לaczון. לא ניתן מצב בו מעבירים מזכרון לaczון בשורה אחת - לא יתכן:

```
mov (%rax),(%rbx)
```

מה הפתרון? נשתמש בריגיסטר עזר, אז נוכל להעביר בין כתובות.

הפקודה *-movzbl, movsbl* שנסם סיומות *mov* שכדי להזכיר. *Z = zero, S = sign*. *s*, ואנו מubbyים למשל *%al, %edx* אנחנו נמלא את שאר המיקום שלא הتمלא (כי *MSBn*) *sign* (*al < edx* ), ככלומר אם הסימן היה אחד נסיף אחדות עד *the MSB*. אם בסוף הפקודה יהיה *z* המשמעות שנמלא את שאר המיקום שלא הتمלא באפסים.

*Branches*: ישנו ריגיסטר *Rflags* בגודל 64 ביט, שמחזיק *flags* שונים. אין לנו דרך לשנות אותן. אנחנו מעוניינים רק לקבל את ערכי הדגמים שלו לאחר פעולות אריתמטיות.

### Singed&Unsigned 2.8.1

ישנים שני דרכים שונות לייצג מספרים. בשתי השיטות משתמשים בייצוג בינורי - אך מפרשים אותו אחרת.

(**המשלים 2**): מפרש את הביט השמאלי ביותר, *MSB* כביט סימן. 0 משמע חיובי ו-1 משמע שלילי. טווח הערכים יכול לנوع בין  $-2^{n-1} \rightarrow 2^{n-1}$ . *Overflow, Underflow*: של הסימן עלול להתחזק, ונקבל תוצאה לא צפואה.

```

        · jmp target # unconditional jump
        · je target # jump equal (ZF=1)
        · jne target # jump not equal (ZF=0)

        · js target # jump signed (SF=1)
        · jns target # jump not signed (SF=0)

        · jg target # jump greater than (ZF=0 and SF=OF)
        · jge target # jump greater or equal (SF=OF)
        · jl target # jump less than (SF!=OF)
        · jle target # jump less or equal (ZF=1 or SF!=OF)

 $.2^{n-1} \rightarrow Unsigned$ : מפרש את הבית השמאלי ביוטר כבית רגיל. מכאן, טווח הערכים הוא 0
 $Overflow, Underflow$ : זה יכול להפוך לאפס פתאום (מודולו).
 $- Unsigned$ : הפקודות מיטה משומשות עבור

ja target # jump above (CF=0 and ZF=0)
jae target # jump above or equal (CF=0)
jb target # jump below (CF=1)
jbe target # jump below or equal (CF=1 or ZF=1)

```

**אנחנו** קובעים את המשמעות של הרגיסטר - האם אני משתמש בו כתובות או כערוך. אני מחליט מה רגיסטר מחזיק.

מה זה אומר בכלל? ב-*CPU* אין דבר כזה *Singed, Unsigned* ברגיסטרים. המעבד לא יודע אם המספר שאתה שם ברגיסטר הוא חיובי שלילי או בכלל כתובות בזכרון. הוא רואה ביטים בלבד. אותן ביטים יוכלים להתפרש באופן שונה בשתי השיטות, אתה בטור מתכוון בוחר כיצד לפרש זאת. המעבד עצמו לא מבין למה אני בחרתי להשתמש, אבל הפקודות שאני בחרתי להשתמש בהם, הם אלו שמרמזות לו על הכוונה שלי. למשל אם משתמש ב-*idiv* הוא יבין שאני *signed* ואם משתמש ב-*div* הוא יבין שאני *unsigned*.

### 2.8.2 מבנה של תוכנית:

כעת נדונ במבנה הזכרון של תוכנית במהלך זמן ריצה, קלומר איך *CPU* מסדר את הזכורן של התהילה.

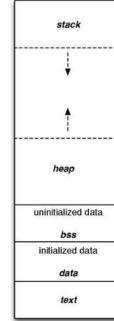
ישנם כמה חלקים בזכרון:

*text* - הקוד עצמו, ההוראות שהמעבד מרים, כל הפונקציות שכתבו בקוד והקריאה. התוכן כאן לא משתנה בזמן ריצה (*read only*).

*data* - כאן נשמרים משתנים גלובליים או סטטיים שיש להם ערך ההתחלתי.  
*bss* - כאן נשמרים משתנים גלובליים או סטטיים שלא קיבלו ערך ההתחלתי. הם מאותחלים אוטומטית ל-0 במהלך העלאת התוכנית לזכרון.

*heap* - כאן מוקצת כל הזכרון שהמתכונת מקצת ידנית, עם *malloc* וכו'. גודל מלמטה למעלה, קלומר לכתובות גדולות יותר.

*stack* - כאן נשמרים משתנים מקומיים וקריאה לפונקציות. כל פעם שנכנסים אל פונקציה נפתח *activation frame* : כל פריים מכל כתובותഴרה, פרמטרים לפונקציה ומשתנים מקומיים. גודל מלמעלה למטה, קלומר לכתובות נמוכות יותר.



**חשוב לזכור:** הרегистר אשר מצביע בראש המחסנית *rsp* חייב להתחלק ב-16. מדוע? מעבדים מודרניים קוראים נתונים ב-*"chunks"* של 16, 32 או 64 בתים. אם הכתובת לא מיושרת, המעבד צריך לקרוא שני אזורים באותו מקום אחד, וזה מאט את התוכנית. לעיתים, כתובות לא מיושרת עלולה לגרום לתקלה.

## 3 הרצתה 3

### 3.1 Jump&Set

קפיצה מותנית נעשית בהתאם לערכי *flags*

Instruction	Description	Flags
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if sign	SF = 1
JNS	Jump if not sign	SF = 0
JE	Jump if equal	ZF = 1
JZ	Jump if zero	
JNE	Jump if not equal	ZF = 0
JNZ	Jump if not zero	
JN	Jump if not none	CF = 1
JNAE	Jump if not above or equal	
JC	Jump if carry	
JNB	Jump if not below	CF = 0
JAE	Jump if above or equal	
JNC	Jump if not carry	
JBE	Jump if below or equal	CF = 1 or ZF = 1
JNA	Jump if not above	
JNBE	Jump if not below or equal	
JA	Jump if above	CF = 0 and ZF = 0
JNL	Jump if not below or equal	
JNGE	Jump if not greater or equal	SF <> OF
JGE	Jump if greater or equal	SF = OF
JNL	Jump if not less	
JL	Jump if less	SF = OF
JNG	Jump if not greater	SF <> OF
JG	Jump if greater	ZF = 0 and SF = OF
JNLE	Jump if not less or equal	
JCXZ	Jump if CX register is 0	CX = 0
JECXZ	Jump if ECX register is 0	ECX = 0

בדומה לפקודת *jump* ישנה פקודה שקופה *setX* שמחזירה את ערכי הרגיסטרים. למה זה טוב לי? זה טוב לי עבור בניית פונקציה שהיא פרדיקט: מחזירה לי כן או לא. למשל, פונקציה כזו:

```
long func(int x,int y)
return x<y
```

תרגומם להivot:

```
gt:
cmpl %esi,%edi
setg %al
movzbq %al,%rax
ret
```

מה קורה כאן? אנחנו מקבלים את המספרים מהפונקציה ועושים להם *cmp*. רегистר *al* יכול  
כעת את התשובה האם  $y > x$ .

**חשיבות לדעת ולזכור:** פונקציה תמיד תחזיר את התשובה שלה אל הרегистר *rax*! הפעולה *X*  
מחזירה את הערך שלה אל רегистר בגודל 8 ביטים - וכך בליית ביריה זה יכנס אל *al*.  
בහמשך, משתמשים בפקודה *movzbq* ומרחיבים את *al* ל*rax* באמצעות הוספת אפסים לאחר  
הערך *al*.

טבלת פעולות :*set*

<b>sete</b>	ZF	Equal / Zero
<b>setne</b>	$\sim$ ZF	Not Equal / Not Zero
<b>sets</b>	SF	Negative
<b>setsns</b>	$\sim$ SF	Non-negative
<b>setb</b>	CF	Below (unsigned)
<b>setae</b>	$\sim$ CF	Above or equal (unsigned)
<b>seta</b>	$\sim$ CF & $\sim$ ZF	Above (unsigned)
<b>seto</b>	OF	Overflow (signed)
<b>setno</b>	$\sim$ OF	Not Overflow (signed)
<b>setg</b>	$\sim$ (SF $\wedge$ OF) & $\sim$ ZF	Greater (signed)
<b>setge</b>	$\sim$ (SF $\wedge$ OF)	Greater or Equal (signed)
<b>setl</b>	(SF $\wedge$ OF)	Less (signed)
<b>setle</b>	(SF $\wedge$ OF)   ZF	Less or Equal (signed)

פקודת *set* לא משנה שום רגיסטרים נוספים פרט ל*al* בו אנחנו משתמשים. היא מחשבת ערך  
בinneriy שמורכב מביטויו כלשהו שמחזר לנו את הדורש.

## declare initialized data 3.2

נרצה להזכיר על משתנים ולהקצות מידע. ישנו כמה אפשרויות :

*skip* ..: מקצתה לנו זכרון לא מאוחROL.  
*byte*, *word*, *long*, *quad* ..: משתמשים באלו להקצות משתנים בגודל המתאים, אך עם אתחול.  
*zero* ..: ניתן להקצות זכרון וכן לאפס אותו באופן רגע.  
*string* ..: רקצאת מהרזהת בזכרון.  
*fill : x, y, val* ..: כאשר נרצה להגיד *x* אלמנטים בגודל *y* עם ערך התחלתי *val*. שימושי וشكול  
לבניית מערך כמו בדוגמה מטה. זה הרבה יותר יעיל מאשר לולאה אם אנחנו יודעים את כל הערכים  
התחלתיים.

דוגמא:

```

int x;
int y = 0;
char str [] = "Hi\n";
int A [10] = {0};
int main {
    ...
}

.section .bss
x : .skip 4
.section .data
y : .zero 4
str : .string "Hi\n"
A : .fill 10, 4, 0
.section .text
.globl main
main ...

```

**חשיבות לזכור:** באסמבלי אי אפשר לבצע השוואה בין שני ערכיים מהאזורון, חיבים להעביר את אחד מהם לרегистר ואז לבצע השוואה של רגיסטר וערך מהאזורון.

### 3.3 הפקודה lea

הפעולה טוונת כתובות לארגומנט השני שהוא מקבלת. הכתובת שלה היא הארגומנט הראשון שהיא מקבלת. חשוב להזכיר - הפקודה *lea* בוגיון **לכל** הפקודות האחרות לא ניגשת לזכרו.

לדוגמה:

#### Examples:

```

leaq (%rbx,%rcx), %rax ; RAX = RBX + RXC
leaq 16(%rsp), %rax # RAX = RSP + 16

```

מה שקרה כאן בדוגמה, זה שהารוגומנט הראשון הוא *addressing mode* והשני הוא רегистר. אנחנו מחשבים את הכתובת **לפי** *addressing mode* ומכניםים את הכתובת לרегистר השני.

במקומות הפעולה יכולים להשתמש *mov* ו-*add*. אז למה צריך להשתמש בפקודה? נסתכל על הדוגמה מטה. כנראה להשתמש בפקודת *lea* זה בשביל לביצוע חישובים - ולא להתעסק בכתובות. **הפקודה היא פקודה הבנייה שniton לבצע במחשב שלו** - יותר מפעולות **(!!!!)bitwise**. מדוע? החישוב מאחוריו הקלים משתמש בחומרה מיוחדת שמייעלת את החישוב.

```

# suppose rdi <- x
f:
    leaq (%rdi,%rdi,2), %rax # t <- x + x*2
    salq $2, %rax # t<<2
    ret

```

מה קורה כאן בדוגמה? ראשית אנחנו מחשבים בשורה הראשונה את  $3\%rdi$ . ומכניםים זאת  $rax$ . לאחר מכן, אנחנו משתמשים בפקודה בשם *:salq* - שיפט אריתמטי - ומכפילים ב $2^2$ , שזה בדיקת  $i \in \{1, 2, 4, 8\}$ . שולש כפול 4, זה אכן 12. נזכר כי הארגומנט הימני ב-*lea* מאפשר  $t = x + x * i$ .

ולכן אי אפשר לטעון שנבצע משוחה כמו  $(\%rdi, \%rdi, 11)$  - זה יכול להיות נחמד אך לא עובד.

**מסקנה:** אנחנו יכולים להשתמש בפעולות כמו *mul*, *div*. עם זאת, פקודת כמו *lea* היא פקודה שמייעלת מאוד את החישוב.

**הערה חשובה:** אם במקומות הינו שמים *mov* והינו עושים את *lea* על  $0x20(\%rsp)$ ,  $\%rdi$  במקומות הינו מקבלים כי  $rdi = RAM[\%rsp + 20]$  במקומות מה שנקלט עם *lea* כמו כן, *lea destination* יהיה רגיסטר.

### C Calling Convention 3.4

כיצד פונקציות ב-C וכן באסמבלי צרכות להתנהג? כיצד מעבירים ארגומנטים לפונקציות? כיצד פונקציה מחזירה ערך מוחזר? הדרך לחזור מפונקציה (ליבל) להיקן שקראננו לה, הוא באמצעות הפקודת *ret*. הפקודת מחזירה אותו להיקן שקראננו לפונקציה.

**נשים לב:** הארגומנט הראשון של הפונקציה תמיד הולך אל *rdi*.

#### Stack Operation 3.4.1

ישנן שתי פקודות באסמבלי בשם *PUSH*, *POP*.  
**Push** הפקודה שדוחפת משוח למחסנית, היא שווה למפקdot *mov* שכלה להכנסיה לכל מקום בזיכרון את הערך, היא יכולה רק למחסנית. הפקודה *Pop* שולפת משוח מתוך המחסנית.  
באשר *CPU* רואה פקודות *push* הוא לוקח את *rsp* מס' *bytes* כלפי למטה (בהתאם לגודל של *push* - שני בייט, ארבע או שמונה). **הערה:** אם לא ציינו את מס' הבטים שנדוחף מראש באמצעות *default push* יהיה שיקוצו 8 בתים, כמו כן הוא מכניס את הערך של הרגיסטר למחסנית לפי *LittleEndian*. כמו כן, תמיד *RSP* מצביע על הערך האחרון שהוכנס למחסנית.  
פקודת *pop* מושכת מס' בייטים מהמחסנית בהתאם לגודלה, ובאופן אוטומטי *RSP* מוקף מעלה לערך האחרון שהוא לא מושכח מהמחסנית.

**נשים לב:** גם לאחר פעולה *pop* הערכים ששימנו במחסנית נשמרים, עד שנכניס ערכים חדשים במקומם. יתרה מזאת - ניתן לגשת לערכים אלו. בקורס שלנו: זה אסור להלטין. זה מגע ממוקם של לחסוך בזיכרון ולמנוע מלבד ולשים שם אפסים במקום. **עם זאת,** מרינה יוטר מרים **שהיא אוהבת לשאול על זה ב מבחנים - אז לשים לב.**

**נשים לב:** בעת דחיפת ערך למחסנית הערך של *rsp* יורץ, בעת הוצאת ערך הערך של *rsp* גדל. וכן - אנחנו נשים לב כי המחסנית בוניה הפקץ מההיוון. **מדוע?** להזכיר תמיד בסיפור של מרינה על הסנדוויץ. **heap** והstack יתחלו "לאכול" אחד את השני משיינ הצדדים עד ש(אם וכasher) ייפגשו. מדובר באופטימיזציה (!!) שעשו באסמבלי. אם *heap* ו-*stack* נפגשים - אין נגמר לו המקום בזיכרון.

### Calling Convention 3.5

כאשר אנו קוראים לפונקציה הארגומנט הראשון נשמר ברגיסטר *rdi* ומישם לפי הסדר לפי הטבלה למטה.

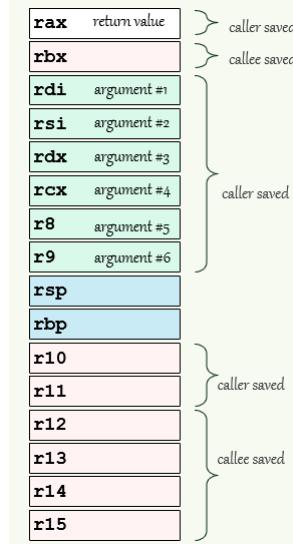
הערך המוחזר תמיד דרך *rax*. מכאן נשאלת השאלה - ומה עם גדלים שגדולים מ-8 בייטים? מאחרי הקלעים הקומפיילר מתרגם את הגודל הזה כאילו הוא מוחזר פוינטר (אכן פוינטר בגודל 8 בתים) אל הערך זהה.

#### 16 הרגיסטרים מתחלקיים לשני קבוצות

נתבונן בבעיה הבאה. נניח ונחנו משתמשים ברגיסטר *a* כלשהו ואז קוראים לפונקציה, ואותה פונקציה משתמשת באותו רגיסטר. כשנוזור מהפונקציה נרצה להשתמש במידע של הרגיסטר *a* בידיעה

שהוא לא השתנה. כיצד נdag שזה יקרה? מי מהפונקציה, הקוראת או הנקראת צריכה לגבות את המידע של הערך הישן במחסנית? מכאן שהזה הופך לתליי רגיסטר. אם למשל, אנחנו מעוניינים ש`r10` ישאר כמו שהוא בפנוייה לפונקציה אז נדרש לדאג לכך בפונקציה הקוראת. אם נדבר על `r12` אז נדרש לדאג לגיבוי בפונקציה הקוראת.

`.rax, rdi, rsi, rdx, rcx, r8 – r11` הם callee saved  
`.rsp, rbp, rbx, r12 – r15` הם caller saved



**רגיסטר RBP:** יש לו תפקיד חשוב בהרצת פונקציות. **תפקידו להחזיק את הכתובת העליונה של stack frame הנוכחי.** מהו "הוציא" במחסנית של `push`-ים של הפונקציה הנוכחיית. לשם מה צריך לשמור את החזחה העליון? בעיקר בשבייל לשחרר את המידע שדחיפנו במהלך הפונקציה. בעת פונקציה: בתחילת אונחנו נדחוף את `rsp` אל המחסנית, כי הוא callee saved ונדיר `rbp = rsp` ונגידיוו כערך של תחילת לטפל בו בתוך הפונקציה הקוראת. לאחר מכן מכך אונחנו נגידיר `rbp = rsp` ונדיריוו כערך של תחילת הפירים. לבסוף נעשה `rbp = rsp` ונווציא את `rsp` מהמחסנית, כלומר נחזיר אותו לערך הקודם שלו. נרחיב עליו מטה -

#### תהליך הקוריאה לפונקציה:

בעת קרייה לפונקציה, אנחנו מכניסים את הערכים שישלחו אליה אל הארגומנטים שנשלחים בהתאם. ו`rsi` וכן `rdi` וכן כל מי שעוד נדרש. לאחר מכן, אנחנו משתמש בפקודה `call func` שקוראת לפונקציה.

**פקודה call - הפקודה מכניסה את return address אל המחסנית (כלומר דוחפים את RIP הנוכחי אל המחסנית).** ולאחר מכן, הפקודה `call func` קופצת אל הליביל `func` (כלומר משנים את RIP אל הליביל `func`).

בעת שנכנסים אל הפונקציה, אנחנו צריכים לדוחו למחסנית את הערך הנוכחי של הרגיסטר `rbp`. דבר נוסף, זה לחתת את הערך של `rbp` ולהכניס לו את הערך של `rsp`. כלומר, `rbp` מציביע לאותו מקום שמציביע עליו `rsp`. באסמבלי בחרו שלא להקצות למשתנה לוקאלי שמות. ומכאן: לאחר מכן מורידים מהערך של `rsp` את סך כל הגודל של הביטים שנשתמש בהם במהלך התוכנית. כלומר מבצעים את השורה הבאה:

`subq $4,%rsp`

במקרה בו למשל התייחסנו לפונקציה שמקצים בה 4

עם זאת, אם אין למשתנים הlokאליים שמות. כיצד נדע איך להתייחס אליהם? לשם כך נדרש את הרегистר *rbp*, אנחנו נשמר את הערך הנוכחי של *rbp* במחסנית, ועזר בו בשביל לדעת לאיזה משתנה אנחנו רצים לגשת.

דוגמה:

```

int result;

void main() {
    result = func(1, 2);
}
int func(int x, int y) {
    int sum;
    sum = x + y;
    return sum;
}

```

```

.section .bss
result: .skip 4
.section .text
main:
    movl $1, %edi # x - first argument
    movl $2, %esi # y - second argument
    call func      # push return address into Stack
                    # move RIP to point to func code
    movl %eax, result # retrieve return value from EAX
    ...

func:
    pushq %rbp          # backup RBP
    movq %rsp, %rbp     # set RBP to Func activation frame
    subq $4, %rsp        # allocate space for local variable sum
    addl %esi, %edi      # calculate x+y
    movl %edi, -4(%rbp)  # set sum to be x+y
    movl -4(%rbp), %eax   # put return value into (part of) RAX
    movq %rbp, %rsp      # close function activation frame
    popq %rbp            # restore activation frame of main()
    ret                  # return from the function

```

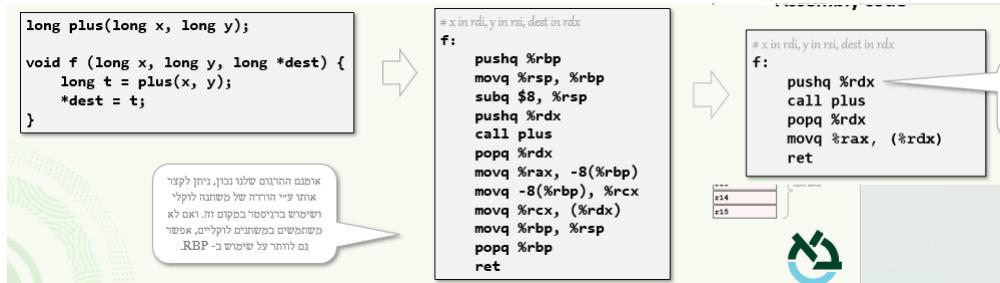
87

נראה כי בעת החזרת הערך, אנחנו צריכים ומהווים (!!) לבצע נקיון ל-*stack*. מי מהווים? גם מי שקרה לפונקציה וגם הפונקציה עצמה. הפונקציה שמה משתנים לוקליים והזיהה את *rbp*, היא צריכה למחוק משתנים לוקליים ולהציג את *rbp* למיקומו. *main* אחראי להעלים כל מיני דברים שיתיכן שם ב-*stack*. כיצד אנחנו מבטלים את המשתנים הлокליים? באמצעות השורה -

movq %rbp,%rsp  
אנחנו אומרים ל-*rsp* להעלות מעלה לכתובת של *rbp* וכעת *rsp* מצביע לערך הישן של *rbp*, וכך אפשר לשחזר את הערך הישן של *rbp*. סה"כ טיפלנו במקרה של *rip*, לאחר מכן מופיעה תמיד הפקודה *ret* הפקודה לוקחת את הערך שצביע עליו ומכוינה אותו לרגעיסטר *rip*, וככה אנחנו חוזרים להיכן שקרהו לנו. סה"כ - תהליך ארוך שנגמר.

#### נסתכל על הדוגמה החשובה הבאה:

אנו מתרגמים קוד. נשים לב כי אפשרות אחת היא להשתמש במשתנה הлокאלי ולקבל את הקוד שמוופיע באמצעות. נשים לב כי דחפנו את *rdx* ולאחר מכן הוציאנו אותו כי רצינו לשמר את הערך שלו כי יתכן שהוא ישנה במהלך הפעולה *plus* כי הוא *caller saved*.  
נראה כי את אותו הקוד באמצעות נתנו לכתוב גם בצוותה שכתבנו בצד ימין. נראה כי נשארנו עם *push, pop* של *rdx* ויתרנו על *rbp* לחוטין - כי ויתרנו על המשתנה הлокאלי. **מכאן המשקנה שצרי**  
**להפעיל שיקול דעת:** אם אפשר לוותר על משתנה לוקלי - נותר, ואז לא נדרש לעבורי עם *rbp*.



חשוב לזכור: אם אנחנו לא נשים לפני משתנה `§`, למשל `a movl a` מתייחסים לכתובת של משתנה `a`.

מה ההבדל בין פונקציה לבין `jmp`? ההבדל הוא שכאשר אנו קוראים לאיזושהי פונקציה והיא סימנה את פעולתה, בהכרח הקוד יחוור להיות לאחר היכן שנקרה הפונקציה, בוגיגוד `jmp .`

ולסיקום - **פקודת pushq %rip**: היא מבצעת `call` בשביב לדעת בהמשך להיכן לחזור. וכן מבצעת `jmp target` אל הפונקציה שמשמעותו `popq %rip`. **פקודת ret**: מבצעת `ret`.

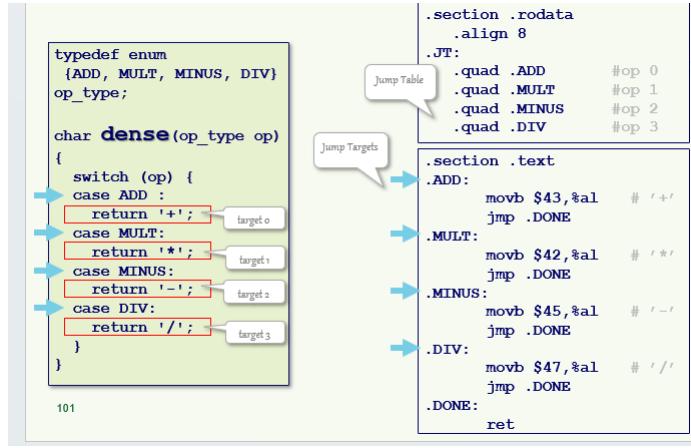
מה נעשה עם פונקציה שמקבלת יותר מושיעת ארגומנטים? נכניס את השיטה הראשונית לרегистרים המותאימים ואז את השאר נדחוף למוחשנית בסדר הפוך. ככלומר אם יש 10 ארגומנטים נכניס את השיטה לרегистרים ואז נדחוף את 10 למוחשנית, 9 8 7 וזהו. הרעיון הוא שהכי יהיה לי קל לגשת אל המשתנה השביעי.

*Stack Alignment*: יש פונקציות שדורשות `rsp` יהיה כפולה של 16. צריך לדעת, לזכור ואיין חובה להבון מדוע. כיצד נודא `rsp` הוא כפולה של 16? נבצע ריצה עם הדיבגר או במאלה. הריצה אכן `rsp` מתחולק ב-16 הוא יהיה כך בכל ההריצות.

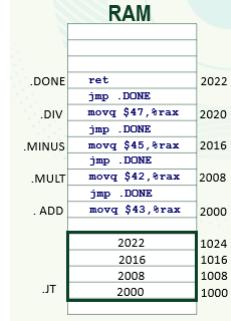
### Jump Table 3.6

בשפת *C* קיים מבנה של `switch&case`. כיצד נממש מבנה זה באסמלבי? וראה כי במקום לכתוב סוייז' אנד קיס ניתן להמירו `if&else`. Um zat - סיבוכיות של `if&else` של  $O(n)$  באשר  $n$  מספר ה`cases`. שימוש מבנה שכזה בזמן ( $O(1)$ )? **בטבלה האש כMOVN**.

נקרא לטבלה האו  $JT$ , נחלק אותה לשורות. בשורה הראשונה (מלמטה) יופיע הכתובת של `case1`. `jmp RAM[JT[op]]` יופיע אל המיקום המדויק? `case2`. כיצד נרש אל המיקום המדויק?



באסמבלאי, נגיד ליבלים שונים לכל אחד מה`cases`, ב`rodata` אנחנו נבנה  $JT$  בו נרכז את כל הכתובות של הליבלים השונים (`cases`). כיצד זה נראה בזכרון?



כמו כן, נזכיר את הפונקציה `CD`

```

dense: cmpl $3,%rdi
ja .DONE
jmp *.JT(%rdi,8)

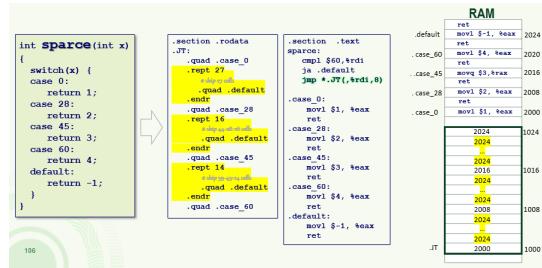
```

כלומר, אם החישוב הינו חוקי נקפוץ לippet המותאים. נראה כי החישוב למשל עבור  $rdi = Minus$  יתקיים כי הכתובת שנתקבל בקפיצה תהיה

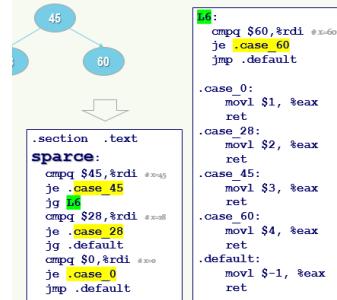
$$2 \times 8 + 1000 = 1016$$

שם מופיעה הכתובת 2016, זה בדיקת המיקום בו שמורה פעולת מגнос. **מדוע הוספנו \* ? כי אנחנו ניגשים פעמיים לזכרון.** כל פעם שניגש פעמיים לזכרון אנחנו נוסיף \*. ככלומר סה"כ אנחנו ניגשים אל  $RAM[RAM[1016]]$

בעיה נוספת - ומה אם `switch&case` שלנו נראה ככה?



נראה כי המרוויחים בcases הם לא רציפים, ניתן להשתמש בטבלה בגודל 60. חבל על המיקום.  
ואנו משתמש *if&else* שיעלה  $O(n)$ ? מה פתאום - נבנה עץ AVL בעלות  $O(\log n)$ . **כל לראות.**



נשים לב - אנחנו במקורה זה יודעים מראש את מבנה העץ, ולכן אנחנו לא צריכים לבנות עץ בפועל. אלא ממש לבצע את החיפוש הבינארי המתאים על העץ הספציפי הזה. ומה סיבוכיות הזמן שלנו?  $O(\log n)$

## GDB 3.7

**GDB הוא דיבאגר:** ניתן להריץ דרכו תוכניות, לעצור אותן במהלך ההריצה, לשנות ערכיהם של משתנים מסוימים בזמן ריצה, להגדיר *break points* וכו'!  
בשביל לקמפל אנחנו כתובים לרוב את השורה הבאה:

```
gcc [flags] <sorce file> -o <output file>
אם נוסיף g – נקבל אפשרות נוספת לעבוד איתם:  
gcc [flags] -g<sorce file> -o <output file>
אם לא נקמפל עם hflag, לא נקבל אפשרות כלשהול. שימוש זהה מטה את קובץ ההריצאה
אך מוסיף מידע שהופך את השימוש ב-GDB לרבה יותר נוח.
```

**כיצד טוענים קובץ הריצה ?GDB run** *gdb <file>*. אם אנחנו רוצים להריץ את התוכנית כתוב *run* או פשוט *r*. אם אין באגים – היא תרוץ, אחרת היא תקרוס ותציג לנו מודיע. כמו כן: ניתן באמצעות *run* לתת ארגומנטים *main* בשורה אחת כך:

```
run arg1 ...
כמו כן ניתן להשפיע על מקום הפלט (להיכן יכתב הקלט) כך:  
run <input.txt> output.txt
כיצד משתמשים בהם GDB? break points: בשביל להפעיל נכתוב break[target] באשר יכול להיות שם של קובץ מקור ומספר שורה, כתובות ספציפית באחרון וכו'.
```

## הרצאה 4

### Assemble process 4.1

נרצה לבצע את שלב הקומפול, המרת הקוד לשפת מכונה.

קובץ שהנקרא *listing file* יכול לתת לנו, בתוך כל קובץ זה יש את קוד המקור (צד ימין), העמודה האמצעית היא תרגום לשפת מכונה והעמודה השמאלית היא הגדרת ה"מייקום" היחסית של כל פקודה ביחס לsection שהיא מאוגדרת בו. אנחנו נקEMPL ונוכל להסתכל בקובץ זה לראות "יישור קו" בין הקומפול שביצענו לקומפול שאמור להיות.

```
.section .rodata
str: .string "Hello world\n"
num: .long 11

.section .text
.align 16
.globl func
.extern printf

func:
    pushq %rbp
    movq %rsp, %rbp
    leaq str, %rdi
    call printf
    movq %rbp, %rsp
    popq %rbp
    ret

$ nasm -f elf64 -I 1.lst 1.asm
```

```
20 00000000 48656C6C6F20776F726C-
21 00000009 640A00
22 0000000C 0B000000
23 0000000D 0B000000
24
25
26 "Hello world\n" is translated
27 according to ASCII, and \n is
28 translated into long according
29 to little Endian
30 00000000 55
31 00000001 4889E5
32 00000004 488D3D(00000000)
33 00000008 E8(00000000)
34 0000000B 4889EC
35 00000010 4889EC
36 00000013 5D
37 00000014 C3
```

**לשימם לב:** גם מותרים בזמן קומpileציה, אם הוא ערך נומירי הוא מתרגם לפי *little endian* ואם הוא string אז הוא מתרגם לפי טבלת ASCII.

מה קורה בעת הליק הקומפול של הקוד הנ"ל? תחילת מגדרים section של *rodata*. הקומפילר יודע שכל עוד לא פתחנו section חדש אז אנחנו ב*rodata*. בעת הקומפול, הקומפילר מנהל הרבה מאוד טבלאות. אנחנו נתיחס לשתי טבלאות:

*Symbol Table*: הקומפילר מכניס שם מידע שהוא מהה. בעת קומפול הקוד לעמלה, אנחנו נתונים בסשן *rodata* אנחנו מכניס אותו לטבלה. אנחנו נגדיר שהערך שלו בתחילת (*info*) יהיה לאחר מכן, הקומפילר נתקל בשם חדש: *str*. היא מכнесת גם כן בשורה חדשה בתוך אפס. כאמור מכאן, הקומפילר ביאזיה section הינו באשר ראיינו את המשנהה *str*. כמו כן, אנחנו יכולים רשותם בטבלה ביאזיה section שהיינו שלו בתחום *location* *rodata*. כמו כן, הקומפילר צריך לעדכן בעת כניסה *str* את הגודל של סשן *rodata*, כי התווסף גודל חדש לסשן. ולכן: הקומפילר מעדכן בטבלה את *info* שלו להיות יותר מקום.

כך נראה הטעינה -

Symbol Table (.symtab)				
Name	Section	Location	Type	Info
.rodata			section	0x10
str	.rodata	0		
num	.rodata	0x0c		
.text			section	0x01
func	.text	0	global	
printf	.text		extern	undef

בעת פתיחת section חדש, למשל כשנכנים אל *text* מכניםים זאת גם לטבלה וכן *info* שלו יהיה שוב אפס בתחילת. נשים לב כי בעת פתיחת סשן חדש זה לא אומן שהפסנו לכתוב בסשן *rodata*: לכואורה - מותר לפתח סשן *rodata* פעם אחר בתחילת הקוד, ועוד הרבה פעמים במהלך הקוד. עם זאת: **מרינה לא מסכימה, ואין לך הצדקה: ואסור בקורס!**

נשים לב שבעת הגעה לשורה `global func`. אנחנו נתקלים ב>New global type `undefined` חדש של `func`: זה אומר, מוגדר בקובץ זהה אבל אנחנו יכולים להשתמש בו בקבצים אחרים. אם הוא לא היה `global` מตอน קובץ אחר לא יכולנו לעשות לו `extern`. כמו כן, הקומpileר מכניס עבورو בטבלה `info` את `undefined`: עד לא הוגדר. אנחנו לא יודעים מה וכמה יש בו. רק יודעים מה הטיפ שלו ובאיזה סקן הוא. כמו כן, בעת שימוש בפונקציות כמו `printf` הקומpileר מעדכן בטבלה היכן הוא פונש אוטה, מעדכן כי הטיפ שלה הינו `extern` והוא `undefined`.

בעת שמנגנים אל פונקציית `func`: הקומPILEר מעדכן `Name` של `func` לשורהiana שהיא אינה עוד. יש מקום בו היא מוגדרת. זה בדוק השלב בזאתנו לא קיבל על הקוד הוראה `kompileita`! שכן אם היה נואר לkompilerl בתוך הטבלה `undefined` זה אומר שיש פונקציה שאין לה `info` (היא לא מוגדרת) והיינו מקבלים שגיאות kompileita.

בעת הגעה לשורה של `leaq`  

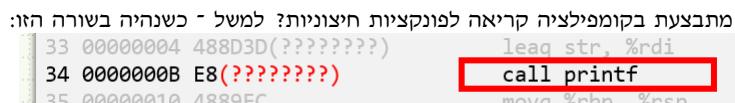
 32 00000001 4889E5  
 33 00000004 488D3D(????????)  
 34 0000000B E8(????????)

נראה כי ישנה בעיה. הקומPILEר רואה שיש לו לייבל בשם `str`, ומנסה להבין היכן הוא ממוקם. נראה כי לא ידוע לנו גודל סקון `text`, וגם בסיום קריאת כל הקוד עדין לא נדע בהכרח את הגודל של סקון `text` שכן יתכן שישנו לאחר מכן את כל הקוד של `extern` שעשינו (ויארכו אותו או יקטיינו אותו). כמו כן, נמצא `str` נמצא `text` rodatab שמצוות מעלה, וקיים שלא ידוע גודל `text` זה משפייע על `rodata`. אז מהו הקומPILEר עושים?

**הקומPILEר בונה טבלה נספת - Relocation Table**: הוא מעדכן שם, שבתוך סקון טקסט, במייקום `0x07` (הבטיח השביעי, בדוק אם שמתחלים למעלה סימני השאלה) אם כי בפועל אלו לא סימני שאלה אלא אפסים), הוא מכניס שם "בקשה" עתידית, יש לך בשם `str`, תקצת שם 8 בתים עבورو לעתיד.

Relocation Table (.reltab)				
Section	Location	Symbol	Size	Type
.text	0x07	str	4	REL

הערה. יש טעות בתמונה ומדובר ב 8 בתים ולא ב 4 כמו שמופיע בתמונה.

כיצד מתבצעת בkompilets קרייה לפונקציות חיצונית? למשל - כשהיא בשורה זו:  

 33 00000004 488D3D(????????)  
 34 0000000B E8(????????)  
 35 00000010 4889EC

אל אותה טבלה `Relocation` אנחנו מכניסים את המיקום (שמתחיל בבדיקה באמצעות סימני השאלה) אליו נרצה להכנס בהמשך את המיקום של `printf`.

בסוף המעבר - הקומPILEר סיים להזכיר את כל הטלילות שהוא צריך למען הקטוף, בפרט השתיים שתיארנו, והוא מוכן לעبور לשלב הבא.

## ELF relocatable format 4.2

כעת נדונן כיצד הקובץ המומפל נראה באמת - לא איך שאנו קיימים אותו ידנית. ספויילר - נראה די דומה למה שקיימנו ידנית.  
 נוח לראות את הקובץ הזה באמצעות תוכנה `readelf`

```
$ readelf -a 1.0
ELF Header:
  Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF64
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  Type: REL (Relocatable file)
  Machine: Advanced Micro Devices X86-64
  Version: 0x1
  Entry point address: 0x0
  Start of program headers: 0 (bytes into file)
  Size of this header: 64 (bytes)
...
Section headers:
```

[Nr]	Name	Type	Address	Offset	Size	EntSize	Flags	Link	Info	Align
[ 0]	NULL		0	0	0	0		0	0	0
[ 1]	.text	PROGBITS	0	40	17	0	AX	0	0	16
[ 2]	.rel.text	RELA	0	100	18	018	I	5	1	8
[ 3]	.rodata	PROGBITS	0	60	11	0	A	0	0	1
[ 4]	.symtab	SYMTAB	0	120	(depends)	18		5	3	8
[ 5]	.strtab	STRTAB	0	(after symtab)	(depends)	0		0	0	1
[ 6]	.shstrtab	STRTAB	0	12b	3c	0		0	0	1

**readelf utility** allows to observe structure of ELF file (object or executable)

זה מידע שהקומפיילר כותב שיכול לעניין מאוד את *Loader* או *Linker* בשרה הראשונה מופיע *ELF*: *Magic* מספרים, *Class*: *ELF64*, *Type*: *Relocatable file*, *Machine*: *Advanced Micro Devices X86-64*. מציין שמדובר ב-*ELF64*, *Loader*, שידע כיצד לפרש את הקובץ ועם איזה קובץ הואעובד.

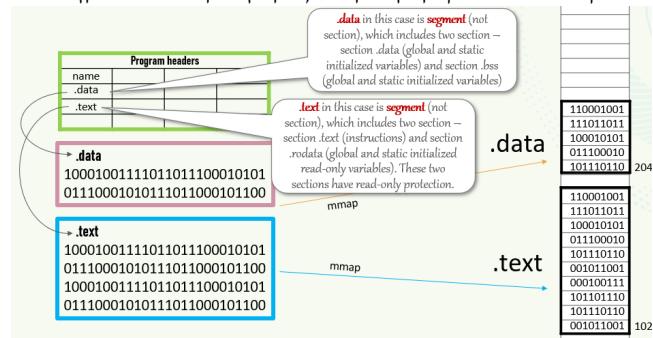
הקומפיילר מעדכן כי הקובץ הוא בשיטת המשלים ל-2 (מס' שליליים בנוסף) וכן עם *Data Little Endian*.

כמו כן מעדכנים שמדובר בקובץ *Relocatable*, שודנדרש לקשר עם קודים שונים (*extern*). *symbol relocationTable* *rel.text*.

לבסוף, מופיעים כל *sections*. נראה כי למאות שקוד המקורית ישנו שני סקשנים, הקומפיילר טוען שיש 7. תחילתו הוא טוען שיש *text*: אותו אחד שאנוינו צרכנו קודם, אכן בגודל של 17. כמו כן, אנו אומרים לו שה*offset* שלו 40. בקובץ המקומפל - הוא מתחילה בשורה 40. לאחר מכן אנו מולמים שאת הטבלאות שקודם ראיינו, הקומפיילר שומר בתוך *section symbol* זה *relocationTable* *syntab* זו טבלת *rel.text* שבסביבו וכנובנו *mmap* - שאינם בחומר הקורס.

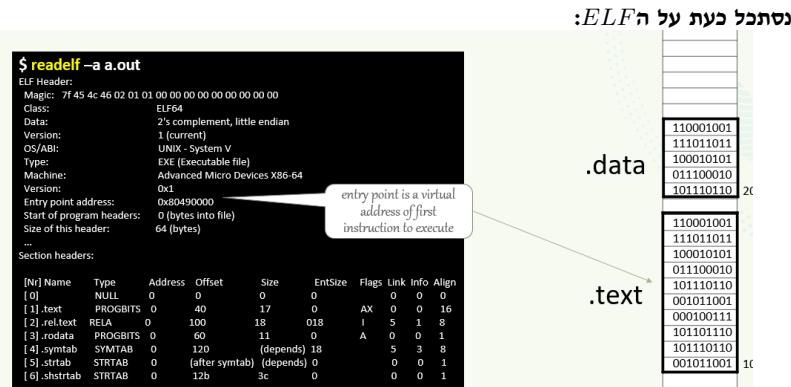
### ELF executable format 4.3

נרצה להבין מה הבדל בין קובץ מקומפל, לקובץ מקומפל ומונגאץ': *.executable*



כאן יש לנו *segment* שבתוכו שני סקשנים בדיק, שנקרו להם כתעת *program headers*

סקשן *bss* וסקשן *data*, יהי בתוכך סגמנט זה.  
סקשן *text* וכן סקשן *rodata* - שניים לא יכולים להשתנות בזמן ריצה.  
וכן, מתבצעת הקצת אץ' כרונ' של סגמנט *.text* ו- *data*



מה השינויים? נראה כי בקובץ הלא מולוקץ', *Entry point address* היה אפס. כאן נראה שיש שינוי גדול, זה לא אפס. עצה זה מביע על הכתובת שאנו חנו רוצים ש-*RIP Loader* יכניס ל-*RAM*.  
נראה כי *Linker* נותן כתובות של *RIP*, 100 נניח, אותה ה-*RAM* מכניס אל *RAM* באותו מקום, ומאי את *RIP* שלהם. עם זאת - האזכור משוחח לכל המחשב, ויתכן שתהילך אחר תפס את הכתובת הזה ושמו את הכתובת הזה במקומות אחרים, 300 נניח. מה נעשה? מערכת הפעלה מייצרת עצמה מיפוי, ובכתובות 100 המערכת כתובות עצמה: בתהליך *a.out* אם יבקש מה-*Loader* כתובות *a.out* על זאת ועוד נרחב במערכות אחרות ל-300. **מסקנה:** הכתובות הן וירטואליות ואינן באמות אמיתיות. על הכתובות במערכות הפעלה.

לכתובות אמיתיות נקרא **כתובת פיזית** - אין לנו דרך לגשת אליה, ויש לנו **כתובת אבסולוטית** - הכתובות שאנו חשופים אליהם. יש לנו **כתובת אבסולוטית**: הכתובות וויטואליות שקומפיילר/לינקר חישב אותן והරיצה להשתמש בה אילו היא הייתה פניה ב-*RAM*. ויש **כתובת relative** - הכתובות ביחס למיקום הפקודה בסגמנט כלשהו.

הערה חשובה: כל עוד *gcc* לא נאמר לו אחרת הוא מניח שגודלו התוכנית קטן. כלומר: גרסת 32 – *bit* ולכון הוא מניה שהתוכנית קטנה ובהתאם שומר כתובות בגודל 4 (במוקום 8 ביביטס). כמו כן עוזר לעילות.

ישנן פקודות כמו *jmp call*, *jmp REL* (רלטיביות – כלומר תבע חישוב *independent* *RelocationTable* ב-*position*). لكن בטבלה של *RelocationTable* על פקודות אלו הוא ישים *ABS* (absolute). *(str(%rip)* – לשמש כמו *%rdi* ו- *str* ולא *%rdi* ו- *str*).

#### 4.3.1 כיצד כתובים וירוס?

אנחנו רוצים להוסיף לקובץ ההרצה שלכם, קטע קוד שאין יצרתי, שאם תרצו את קובץ ההרצה, הקובץ שיירוץ גם על הדרכך ויעשה בעיות.

**הנה רעיון:** נבקש ממכם ללחוץ על כפתור כלשהו, אם תלחץ על הכפתור אני אוריד לכם למחשב קובץ הריצה, שיכנס אל *file system*, נחפש את קובץ ההרצה שלכם, ולפי תכונת מונחה עצמים כפי שראיםנו – ניתן לכתוב לתוך קובץ. ווסף אל הקובץ את הקובץ המקורי המקומופל שלו, ושמור את הקובץ. כאשר נריץ את קובץ ההרצה הזאת, נריץ גם (אולי?) את הקובץ שאינו הוסיף לכך. נניח שהקובץ שאינו הוסיף מוחק את כל *file system*. נשאלות כמה שאלות.

א. האם *Linker* יפרסר זאת? בעת *Linker* שיחשב את הגודל הסופי של סקשן *text*, הם בודאות לא התחשבו בקוד שאין הוסיף אל קובץ ההרצה – קוד זה לא היה קיים לא בזמן הריצה

ולא בזמן לינוקו. אז, לא אמרו להעלות את הקוד הזה כחלק *process image*. לכאורה, בפועל - כאשר *process image Loader* טוען *Loader* הוא לא טוען אותו לפי הגודל המדויק. בפועל, אם ביקשתי מוקם סגמנט טקסט לבנות סגמנט *Loader* בגודל *4bytes* הוא בונה סגמנט בגודל *4k* (!!!). כאשר אנחנו טוענים מוקם לסקשן טקסט אנחנו טוענים את זה פר בлокים. ולכן, יותר מקום פנוי, ולכן אותו קוד, עם קובץ הרצה שימחק לי את *file system*, עשוי להתקבל ע"י *Loader*.

ב. האם הקוד הנוסף הזה יתבצע כאשר *process* יתבצע? ח"ד משמעית - לא. בסוף פונקציית *main* יש לנו תמיד *exit*, גם אם תבנו אותו וגם אם לא מערכת הפעלה מוסיפה לבד. לעומת זאת, מערךת הפעלה שאמרתה: *Ani process* שסיים לבצע כל מה שהוגדר, ובבקשה מערךת הפעלה תמחקי את *process* כתע. ולכן, בודאות, הקוד הנוסף לא יתבצע.

אי מה נעשה? במקום להוסיף את הקוד בנוסף אל *segment text* אנחנו נמחק חלק מהקוד שם, ונכנס במקום את הקוד שלנו. כתע - יש סיכוי שהקוד יתבצע. מודיעו יש סיכוי ולא בודאות? כי אכן שהמחיקה הרסה דברים פשוטים וכעת הקוד לא ית侃פל או תהיה שגיאת זמן ריצה.  
אז אנחנו לא רוצחים אליו - אנחנו רוצחים בודאות ליצור וירוס. כיצד? נדרוס את השורות הראשונות של פונקציית *main*. נכנס אל השורות הראשונות של *main* את הקוד של, והוא בודאות ירוץ. באותו הזמן אפשר אפסון ממון למחוק את שאר הקוד (אם כי זה קשה יותר), אך זה לא יפריע לי כי בודאות הקוד שלי יתבצע, ולא אכפת לי מה יקרה הלאה בקוד המקורי - כי מטרתי להרוס.

**פתרון חלופי וכל הרבה יותר:** נבדיק את הקוד שלנו, ונשנה את *entry point* להיות למקומות שהכנינו את הקוד. ואז, כשה我们会 לינוק את *Loader* הואילך אל הקוד המורושע שתבנו, וסיימנו.

#### Disassembled 4.4

תהליך של הפיכת קוד משפט מכונה לקוד בשפת אסמבלי. נעיר כי תהליך זה לא חוקי לפי החוק על קוד שהוא לא שלו. זה מאפשר בוגדים רבים יותר נוחה מאשר לחפש אותם בקוד של שפת המכונה.

#### Linking process 4.5

לוקחים כל מיין קבצי *Object* (שלנו ולא שלו, ספריות סטודנטיות לדוגמה) ורוצחים ל�נצ' אוטם לקובץ הרצה היחיד. נשים לב שקובומפイル אחד מבצע קומPILEציה כל פעם של קובץ אחד. קיבלונו הרבה קבצים מקומפליים, ונרצה לבצע *linking* של כל הקבצים לקובץ אחד. נשים לב כי כל שגיאה שתתבצע בשלב זה תקרא כתע **שגיאת לינוקו!**. עד היום שגיאות אלו היו תחת שגיאות קומPILEציה". מטרת נוספת של הלינקר היא לפתור את כל הבעיה שנוצרו בזמן הקומPILEציה.

**התנשויות symbols:** בקובץ *a* יש לי משתנה בשם *x* וגם בקובץ *b* יש לי משתנה בשם *x*. נוצרה התנשות: יש לי שני משתנים בשם *x*? האם זו שגיאת לינוקו?  
**הגדרה - strong:** נאמר כי *symbol* *strong* אם הוא שם של פונקציה או שם של משתנה גלובלי מאוחROL.  
**הגדרה - weak:** נאמר כי *symbol* *weak* אם הוא שם של משתנה גלובלי לא מאוחROL (*bss*).

כללי :Linker

1. סמלול יכול להופיע רק פעם אחת. אין מצב שיש שני פונקציות למשול באותו שם.

2. סמלול יכול לדروس *strong symbol* אם נקליג' את שני הקבצים.

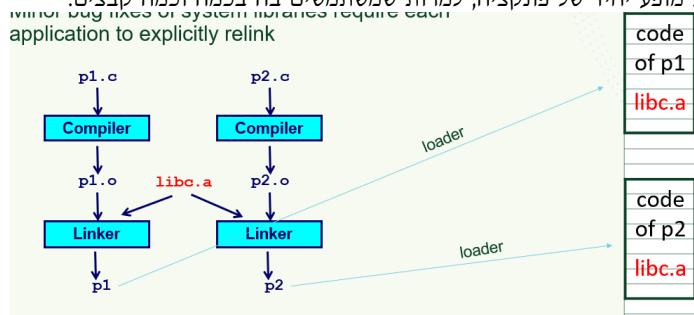
3. אם יש לנו שני *weak symbol*, הלינקר יכול לבחור אחד מהם שרירותי.

**חשוב:** נניח שבתוכנית אחת הגדנו  $x = 7$ , ובתוכנית אחרת הגדנו  $x = \text{double}$  ללא אתחול. לפי הכללים, יוכל להשתמש לכאן ולהתעדף את  $x = 7$  שהוא *strong*. בשאותו קוד מותרגם לאסמבלי, הוא מגדיר להם גדים שונים. על פניו, לכארה *Linker* יאשר אבל בתוכנית השנייה התיחסנו אל  $x$  כ-*double* ובראשו כ-*int*, לנו אנחנו יכולים לקבל במקרה ריצה תואמת לא טובה. וכך קיבל כאן שגיאת זמן ריצה.

**באמצעות linking' אפשר לבנות גם ספרייה עצמאית:** כיצד? נכח למשל פונקציה *printf.c* ונממש אותה, נקمل ונקבל קובץ הרצה. כמו כן נעשה זאת על עוד פונקציות כמו *atoi.c* וכו'. נקבל הרבה קבצי הרצה וננלק' אותם יחד בammedot linking' להילינקר לקובץ אחד.

שים לב כי הילינקר ישמש לפעמים בספריות סטטיות קיימות, למשל היקן שיש את *printf*, הוא לוקח את הספריה יחד עם שאר הקבצים שמקפלנו ואוטם מכניס לקובץ הרצה.

שים לב כי ניתן מצב כאן מטה: שני קודים שימושים שניהם בספריה כלשהי. *Loader* יקח כל קוד יחד עם הספריה, ונקבל (צורה הגיונית) של קוד כולל את הספריה ולבן בכרכו שלו יש כרגע פעמיים את הקוד של *libc.a*, והחולב כי אנחנו מבזבזים מקום בזכרון. מה נעשה? אם אנחנו יודעים שיש לנו פונקציה כמו *printf* שמשופיעה הרבה פעמים, נשים אותה בתוך **ספריה דינמית**: אומרים למשה *Loader* - אתה לא מעלה לנו איזורן את שני המפעמים של *printf*. אבל, כשאתה רואה *printf* אתה הולך בספריה הדינמית, מושך משם את הפונקציה וכותזאה מכך אתה מקבל מופע ייחודי של פונקציה, למרות שימושיהם בה במקביל ומהם הקבצים.



שים לב, איינו יודעים, גם לא *Linker*, היקן תמצא ספריה דינמית שכזו. קוד מסוג זה, יקרא *Position Independent Code*.

#### Position Independent Code 4.6

קוד שלא תלוי במיקום. מהו קוד שתלוי במיקום? ישנים מיקומים של sections שנקבעים בזמן הקומPILEZA / linking'. אם gcc החלט שסקשן *text* מתחילה בכתובת 1024 או 1024 הוא המיקום ההתחלתי של הסקשן.

נסתכל על הקוד הבא. נראה כי ישנו הביטוי *str(%rip)*. מה כתוב כאן למעשה? כאשר ה-*CPU* רואה קוד זה, הוא מפענה את זה בצורה שונה מזו *addressingMode*. הוא מתייחס לביטוי *str(%rip)* כיחסור כתובות *str - rip*. וכך הוא מחשב את המיקום היחסי של *str* ביחס למיקום הנוכחי של *rip* באשר ממבצעים את פקודת *leaq*. למעשה הפקודה אומרת: כמה אני צריך לזרז מהמיקום הנוכחי בשבייל להציג אל *str*. הkompiiler (דges - אנחנו לא) מחשב את היחסור, מסמן  $x = str - rip$  ואות מותרגם זאת ל-*str + (%rip)* - בדיק לפי  $x + %rip$

<pre>.section .data <b>str:</b> .string "Hi"  extern printf .section .text .globl func func:     movq \$str, %rdi     call printf     ret</pre>	<b>PIC</b> <pre>.section .data <b>str:</b> .string "Hi"  extern printf .section .text .globl func func:     leaq str(%rip), %rdi     call printf     ret</pre>
---	---

**חשוב לדעת:** בין סגמנט *text* לSEGMENT *data* ישנו מרוח בין שני הSEGMENTים של גודל קבוע (נניח 1000) שהКОМПИЛЯР קבע בזמן קומpileציה וידעו בזמן ריצעה. מודיע זה חשוב? אם יהיה לנו *loader*: *confused* הוא בנה *process image* במקומות הלא נכון, הוא עדין ישמר על הרוח הקבוע ובאמצעות הפוקודה (*position independent str(%rip)*) הוא מרים את הקוד שMOV' פועל בצד ימין בתמונה, שהוא קוד בלתי תלוי במקומות, ולמרות שהקוד שלו יהיה תקין.

#### סיבות לשימוש *position independent*:

1. אחת הדרכים לוודא שהקוד שלנו יהיה בטיחותי, היא לפזר את הSEGMENTים שלנו במקומות שונים בכל פעם. ישנו וירוסים שונים שימושים על *process image* בצהורה *dynamic linking* – וכך לוירוס יהיה הרבה יותר קשה למצוא את *section's*' השוניים בקוד.
2. באשר נעשה *dynamic linking* לא נדע היכן נמצאות הספריות הדינמיות שלנו, ולכן מיקום שכך המיקום משתמש בספריות דינמיות לא יוכל להשתמש בקוד תלוי מיקום שכן המיקום משתנה בהתאם למיקום הספרייה בזיכרון (שיכול להשתנות).

\* אם מקפלים עם *-no pie* – זה אומר שמדובר בקוד שהוא לא *position independent* ואז לא צריך (לכואלה) להוסיף *%rip* בסוגרים.

## 4.7 תרגול

### 4.7.1 ארגומנטים ב-STACK

נניח שהעברית פרטורים לפונקציה דרך המחסנית. נראה כי נרצה להשתמש בערכיים אלו מהמחסנית. כיצד נעשה זאת? נצטרך "לקפוץ" מעל המיקום הנוכחי (*rbp*) בגודל מה שדחפנו מתחתיים למחסנית.

### 4.7.2 Variadic Functions

פונקציות שנitin להעיבר אליהן מס' לא מוגבל של משתנים. הארגומנט הראשון **לרוב** יציין את מספר הארגומנטים שיועברו.  
כיצד נממש פונקציות כאלה ב-C? אנחנו לרוב נכתוב ... במקומות הפרטור האחרון. החתימה תראה כך:

בשביל לכתוב פונקציות כאלה ב-C נדרשעזר בתיקייה *stdarg.h*. שם יש את הפונקציות הבאות:  
**va\_start**: מאפשר לגשת אל הארגומנטים של הפונקציה  
**va\_arg**: מאפשר לגשת אל הארגומנט הבא של הפונקציה  
**va\_end**: באשר מסייםים "לטיל" על ארגומנטים שהפונקציה קיבלה.  
 קוד שכזה יראה כך:

```

1 int sum(int count, ...) {
2     va_list args;
3     va_start(args, count);
4     int sum = 0;
5     for (int i = 0; i < count; i++) {
6         int num = va_arg(args, int);
7         sum += num;
8     }
9     va_end(args);
10    return sum;
11 }

```

באשר הפונקציה ה"ל' מחשבת סכום של מספר משתנים שנקלט, באשר הפעמטור שהיא מקבלת הוא מספר המשתנים. בתחילת משתמשים בargs על מנת שייהי אפשר לגשת ובכל שלב מתקדים .va\_args לאחד הבא עם

#### 4.7.3 קבלת ארגומנטים בשורת הרצאה

כמו ב-C, גם באסמבלי אם נכתב עם קמפול הקובץ מ' משתנים נוכל להשתמש בהם בפונקציה. תמיד יתקיים כי:

$$rdi = (\text{int})\text{argc}$$

$$rsi = (\text{char}^{**})\text{argv}$$

כלומר, יחזיק את מס' המשתנים וrsi מצביע לערךם של המשתנים עצמם.

#### Flow Control 4.7.4

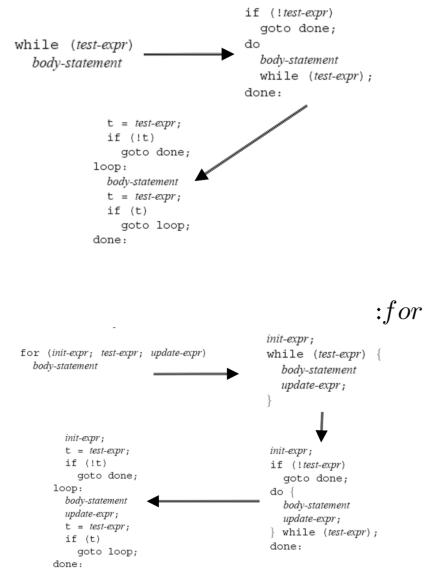
כל מבנה של תנאי, לולה וצדומה ניתן להמיר מ-C לאסמבלי. נראה מספר דוגמאות.  
:if&else

<pre> if (test-expr)     then-statement else     else-statement </pre>		<pre> t = test-expr; if (t)     goto true; else-statement goto done; true:     then-statement done: </pre>
--	--	--

:do while

<pre> do     body-statement while (test-expr); </pre>		<pre> loop:     body-statement     t = test-expr;     if (t)         goto loop; </pre>
---	--	--

:while



## הרצאה 5 - memory hierarchy - 5

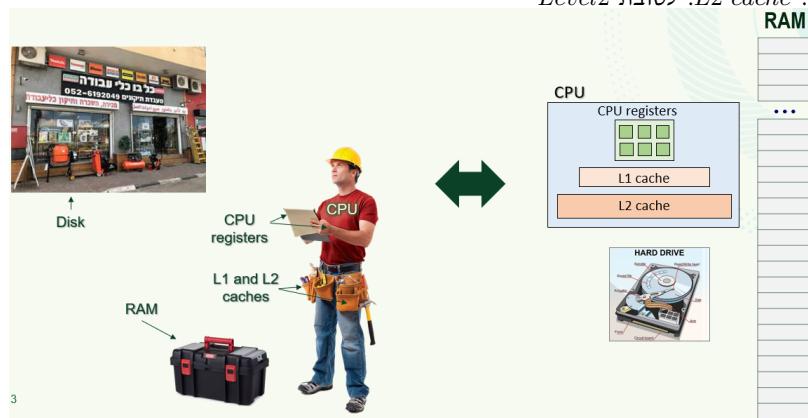
### 5.1 הקדמה

אננו מכירים עד היום שישנו *RAM*, ישם ורגיסטרים בתוך *CPU* וכן דיסקים. ישם שניים

:*CPU* נספחים בתוך *the CPU*

*Level1* : לטובת *L1 cache* .1

*Level2* : לטובת *L2 cache* .2



נניח שיש לנו אדם" בתפקיד *CPU*. הידיים שלו הם רגיסטרים (אחרת, איך יעבדו?), הדיסק זו החנות - זה לוקח זמן לגשת אליה, וлокח זמן לגשת אל הדיסק. זמן גישה לדיסק עלותה פי מיליון בערך לעומת גישה לרוגיסטרים. זו הסיבה - שסבירה - לא מסכים *loader* לעבוד עם דיסק, ומזכיר אותו לבנות *RAM process image*. החגורה על האדם - היא בדיקת *cachen*. פחות אטרקטיבי מרגיסטרים, אך עדין קרוב אליו. ארוצו הכלים יהיו *RAM*.

## cache 5.2

כל תא בתוך *cache* הינה שורה ונקראת *cache line*. ישנו מס' *levels*. כל *cache line* הוא באורך של 64 bytes. ניתן לראות שכל שורה כזו אמורה להחזיק יותר מנתון אחד. *Level1*: לטובות קרוב יותר למעבד, ולכן הוא מהיר יותר אך הוא קטן יותר. *Level2*: לטובות רחוק יותר מהמעבד, ולכן הוא איטי יותר, אך הוא גדול יותר.

מדוע *L1* לא גדול יותר אם קרוב יותר *CPU*? זה לא עובד ככה. אם הוא יהיה גדול יותר בהכרח הוא יהיה יותר רחוק מהמעבד כי אם הוא גדול יש הרבה מאוד מקום, וחלק מהמקומות יהיה אוטומטיות רחוק יותר מהמעבד.

גם *L2* הרבה יותר מהירים מה*RAM*.

ניתן לחתך נתון מריגיסטר ולהעביר אותו ישירות אל *cache*, ולהפוך: ניתן לחתך *cache* ולהעבירו לרגיסטר.

מדוע אנחנו זוקקים ל*cache*? נניח שיש לנו משתנה *X* בזיכרון במיקום 1028. ברמת החומרה - אוטומטית, המעבד יורד למיטה בזיכרון לכתובה הנמוכה ביותר שקרובה אל *X* שמתחלקת ב-16: זו 1024, ומשם הוא ילוך 64 ביטים ומכנים ל*cache line*. מדוע זה לא מיותר? למה לחתך הכל במקומות לחתך רק את *X*? לא חבל על המיקום? לא חבל. נניח שרצינו גם את *Y*, שבסביבות גבולה נמצאת *cache*. נוכל לחשוף אותו ב*cache*. **נשים לב: גישה ל*RAM* עלותה פי מה מגישה ל*cache*.** מי אמר שככל נרצה את *Y*? ובכן?

**Locality principle:** עקרון הלוקאליות. אם משהו נמצא בקרבה של מה שכתעת השתמשתי בו (*Y*) נמצא בקרבת (*X*), אז בסביבות גבולה אני משתמש גם ב*Y*.

למה טוב *cache*? נניח יש לנו מערך וללאו שוכמת אותו. אם נרצה את *a[0]* נקרה שלאחר מכן נרצה את *a[1]*. איזה יופי - הוא נמצא בזיכרון *cache line*. אם יגמור לי המיקום לפונקציות? נגששוב לזכור. המשקנה: באמצעות *cache* אנחנו ניגשים הרבה פעות לזכרון. ניגשים לזכרון רק בשביל להביא *cache line* חדש. גישה לזכרון עלותה 100 nano שניות.

**מסקנה:** נרצה לכתוב קוד כמה שייתור סידرتוי, ככל שהוא יהיה יותר מהיר. אם נעשה *if* זה לא טוב, זה מביא אותנו לgesture יותר לזכרון. מה באשר לפונקציות? לא נשטמש בפונקציות יותר? נשכפל קוד מלא פעמים במקומות לקרווא לפונקציה בשביל לשומר על קוד סידרטוי? נשים לב שניפוח זה לא טוב - למה לא טוב? סיבוכיות המיקום גדול, *process image* גדול במיקום שלו ב*RAM* וכן בעקביפין זה גם פוגע כשנדבר על אופטימיזציות).

**spatial locality**: אם יש משתנה או פוקודה לידי בזיכרון, אז בסביבות גבולה מאד רצוי שייהיה שימוש כתעת במשתנה או פוקודה זו.

**Temporal locality**: אם אני משתמש במשתנה, סביר להניח שבזמן הקרוב אני משתמש בו עוד פעם. למשל אם נחשב סכום של מערך העקרון לא יפעיל על *a[0], a[1], ..., a[n]* אך כן יפעיל על *sum*. לולאות משתמשות טוב בעקרון זה.

**התהיליך של cache למציאת *x* כלשהו:** נקראת *x*, *cpu* בודק אם *x* ב-*L1*. אם אכן המצב: נבייא את *x* לתוך רגיסטר ב*CPU*. (נשאלת השאלה, בתוך *L1* יש מס' שורות. האם זה מה שנותה באיזו שורה נבייא את *x* אליה? הרי אנחנו לא רוצים לעבור בכל השורות. החיפוש ב*cache* היעיל.). אחרת, נחפש ב-*L2*, אם נמצא נבייא את *x* לרגיסטר ב*CPU* וכן אנחנו נקדם את *x* אל *L1* (מדוע?). בסביבות גבולה כמו שאמרנו נשתמש בו שוב ונרצה פעם הבאה לגשת אליו מהר יותר). אם לא נמצא שם - איזי נלך לחפש ב*RAM*.

נשאלת השאלה - אם הוא בכלל הtgtlaה ב-RAM, לא חבל על הזמן שbezino בחיפוש ב-L1, L2? אנחנו מניחים שהקוד הוא cache friendly - קוד שהמתכנת שכתב אותו מודע ליתרונות של cache וכן בסיכוי גבוהה מאוד  $x$  יתגלה ב-L1, L2. אם לא: תכו, אך הסבירות לכך נמוכה ולכנן בטעות הרחוק זה אכן משtell.

איך מגיעים אל L2? תמיד מנסים להכניס אל L1, אם אין שם מקום אנחנו מכנים אותו ו"מעבירים" את מה שהוא בו אל L2. כיצד אנחנו ידעים את מי אנחנו מעבירים במקומו ל-L2? אינטואטיבית - את מי שהשתמשנו בו הכי רחוק, וכך הסבירות שגרה שוב להשתמש בו כרגע נמוכה.

**עדכון ערך ב-cache ולא ב-RAM:** ישנו רגיסטר בשם RIR - שומר את הפקודה הבאה לביצוע. אנחנו קוראים אותו מבצעים את הפעולה ונניח שקדם לכך שמרנו  $x = 5$  והפקודה הייתה  $++x$ . נראה כי נרצה לעדכן את הערך ל-6/cache בלבד. אבל אז נוצרת בעיה: ב-RAM כתוב לי  $x = 5$  אבל(cache) הוא 6. ככלומר: RAM לא ידע את הערך העדכני של  $x$ . זו בעיה - במסגרת הקורס שלנו לא אפשר לנו מזיה כיוון שאנו מדברים על תוכנות סדרתיות. עם זאת, כאשרначיל לכתוב תוכנות מקבילים זה יהפוך לבעה קритית ויעשה את ההבדל בין תוכניתנו נכון לשגוי. נדע זאת כרגע - ונאפשר את זה. בקורסים עתידיים - נדע כיצד פתרים זאת.

**מטריצות:** ישן שני דרכים לעبور על מטריצה - דרך שורות ודרך עמודות. מעבר על מטריצה דרך שורות יהיה הרבה יותר מהיר מאשר דרך עמודות! קוד שעובר על עמודות אינו cache friendly: בכל שלב אנחנו משתמש רק באיבר הראשון של השורה שנביא וזה יהיה לא מהיר בכלל. אם נעבור לפי שורות לפי העקרון שלדנו אוזות השימוש cache בחרשות ייעו אחת אחרי השניה והקוד יהיה סדרתי. מסקנה: הרבה יותר מהיר!

#### מבחן גדים:

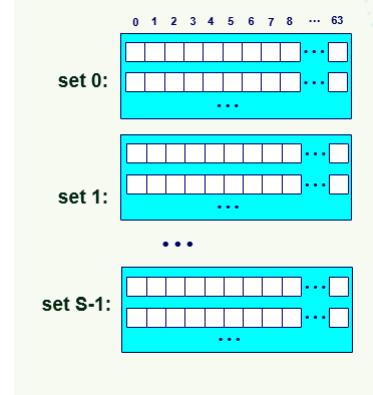
אם מבאים קוד שהוא נכנס אל L1 instruction cache שהוא בטוחה של 16 – 128(KB)

אם מבאים קוד שהוא נכנס אל L1 data cache שהוא בטוחה של 16 – 128(KB)

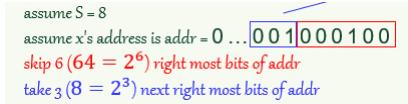
128(KB) – 8(MB) הוא בגודל (L2 cache

### organization of a cache Memory 5.3

מוחלך cache' ל-'set'. כמה? תלי בכמה היכרן הגדייר. ראיינו כי אנו יודעים את הגודל של L1 ולכנן יודעים מה גודל כל set. וכן אם אנו יודעים גודל כל set ואנו יודעים את גודל כל cache line (ב-X86 64 בייטים) אז אנחנו יודעים לדעת כמה יהיו.



נניח שיש לנו תוכנית פשוטה. מוגדר משתנה  $x$  ב-RAM ומבצעים לו  $+x$ . נניח כי  $8 = |Sets|$ . איך נמוקם את  $x$  בזיכרון?  $x$  נמוקם על המיקום של  $x$ . נחליט שאנו מתעלמים מ-6 הביטים הימניים של הכתובת ( $64 = 2^6$ ). נkeh את שלושת הביטים הבאים (3) הימניים ביותר ממי שנשארו. שלושת הביטים הימניים ביותר שנטרו קובעים באיזה set נכנס cache שלו.



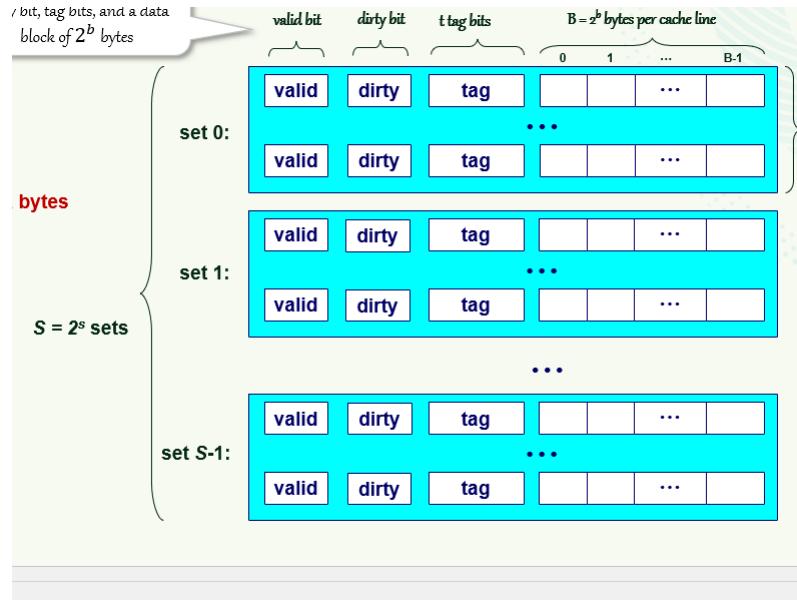
נניח שהוכנסנו את  $x$ , כיצד ה-*cpu* ידע באיזה set נמצא  $x$  ומה המיקום היחסי שלו באותה מחרשה? אם מס' מתחולק ב-6 לא שארית בהכרח 6 ביטים אחרונים שלו הם אפסים. אנחנו נסתכל על כתובות שמתחלקות ב-64bits וחייב קורובה לאיקס הנוכחי, כי שאמרנו קודם ונkeh 64 ביטים. נkeh את 6 הספרות האחרונות של כל כתובות והם ייצגו את המיקום היחסי של הכתובת בתחום *cacheLine*. כמו כן: נkeh את  $\log_2$  באשר  $x$  הוא מס' הקבוצות: נkeh את  $\log_2$  הספרות הבאות לאחרי 6 הספרות והם ייצגו באיזה קבוצה אנחנו נמצאים מבין  $x$  הקבוצות שכן מספיקים  $\log_2$  ספרות ליציג  $x$  מספרים. ס"כ בז' תיוג כל כתובות ב- $s$  שבטו *set* *cache*. נשים לב שכיוון שתחלת ה-64 ביטים הם כתובות שמתחלקות ב-16 הספרות הימניות שלה יהיו אפסים ומשם נתחיל ליציג את המיקום היחסי *cacheLine*.

הערה חשובה: המיקום בתחום *set* אינו ידוע ולכן בכל *set* צריך לחפש.

**לכל cacheLine מוצמדים:** *tag*: ראיינו כי לcheinנו קודםplen 6 ועוד 3 ביטים, אך נשארו  $55 - 9 = 46$  ביטים נוספים, אנחנו נרצה להכניס אותם אל *CPUs.tag*. רוצה לדעתה ש-*tag* מכיל באמצעות את  $x$  ולכן הוא משתמש ב-*tag*, שכן יתכו כתובות שונות עם אותה סיומת של 9 ביטים. כשה-*cpu* מוחפש את  $x$  הוא מסתכל על שלושת הביטים הכהולים, נגש *set* המתאים, באותו *set* יש מס' שורות וכל שורה יש *tag*. *cpu* בודק שורה שורה את *tag* השורה וכן אחר הביטים של  $x$  (לא אלו שירדו) ובודק האם אכן ישנה התאמה.

**dirty bit**: האם *cacheLine* הוא *dirty* או לא. מודיע זה חשוב לדעת? *Ram*. מודיע "פטור" *dirty bits* אנחנו חיבים לגשת אל *Ram* ולעדכן את הערך של הביט הראשון אחרת - אנחנו נאבד אותו (הוא השתנה ואנו לא יודעיז את זכרונו). ואנו מוחקים מה-*cache*. סכנה לאבד אותו לנוכח.

**valid bit**: האם מותר להשתמש בנתון שישוב בתחום *cacheLine* או אסור. למה שיהיה אסור להשתמש? כל עוד אנו בעולם סיידרתי, אין סיבה שהוא אסור לשימוש וכל השורות הין. כשאנו עוברים לעולם מקבילי או בעולם שיש בו כמה *process* - נהיה בעיה. לכן כרגע, כשאנו עולם סיידרתי: **column valid**. במקבילות - באשר עוברים בין *process* שונים אנחנו מסמנים ישירות *invalid*.



מצב שיש לנו בדיקת cacheLine אחד בתוך כל  $set$ . במצב זה אנחנו נסחזר את  $CacheLine$  כי יש בדיקת אחד בפניהם. החסרון המרכזי שams נביא גם יודעים איך לחשוף את  $set$  נסחזר את  $CacheLine$  נסחזר אל אותה  $set$  נסחזר אותו. אין מקום אחר בתוך אותה  $set$  נסחזר שנווכל להעבריר אותה אליו. היתרונות המרכזיות הוא שזמן החיפוש כאן הוא בדיקת  $O(1)$ . נסתכל על tag בדיקת  $x$  שם או לא.

$.set$   $K$  – associative – Cache

## Direct-Mapped Cache

Simplest kind of cache.

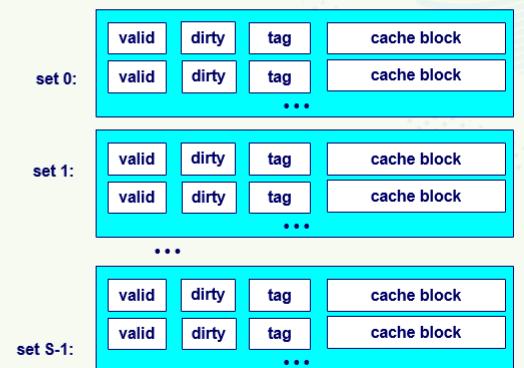
Characterized by exactly one line per set ( $E=1$ ).



22

## K-Associative Cache

Characterized by  $k$  lines per set.



למה שלא נקח  $set$  יחיד? יקח הרבה זמן לחפש בתוך  $set$ 's. העובדה שיש לנו הרבה שוניים עוזרת לדעת לאבחן לאיזה  $set$  שייך איבר, וכך לדעת לצמצם את טווח החיפוש.

מה שנעשה לרוב: יהיה לא להיות במצב של סט ייחד, ולא במצב של  $k$  אלא יותר מאשר.

## 5.4 תרגול 5

### 5.4.1 static linking

**ישור קו:** *symbol* מייצג פונקציה, משתנה גלובלי או משתנה סטטי.

- מהם החלבים שקוד שנכתב ב-*C* עוזר? מתחילה ב-*c.c*.
- א. השלב הראשון הוא *reprocessing* שבמסגרתו כל *define* ו-*include* שיש לו בקובץ *.main.i* מוחלפים בערכם המקורי. מתחליך זה יצא קובץ עם סימות *.main.s*.
  - ב. מתקבל קובץ ב-*C* ומוציא קובץ באסמבלי *.main.s*: *compiling*.
  - ג. השלב הבא הוא *assembling*: ממקבל קובץ האסמבלי וממיר את הכתובות לשפת מכונה. במהלך תהליך זה מוספות טבלאות כמו *strings* בהרצאה. בסיסים שלב זה יש קובץ *.o main.o*.
  - ד. המטרה של תהליך *linking* שבסיום מתקבל לנו קובץ הרצה *main* או *main.o* וקבצי ספריה נוספים, רצאה ליצור צמוד בין הפונקציות או המשתנים הגלובליים אליו. אנחנו ניגשים לבין השימוש שלהם פיזית (שמופיע בספריה).

*library*: אוסף של קבצי *.o*. ישנו שני סוגי של ספריות - ההבדל בניהם הוא בסוג *linking* שיתבצע.

א. ספרייה *static* היא ספריה שמופעל עליה *static linking* - אם אנו ניגשים ל-*symbol* מסויים במהלך שלב ג', המימוש של אותו קוד, למשל המימוש של *scanf* ממש יוכנס אל קובץ הרצה שלנו בסוג *linking*.

**חסרונות של static linking**:

1. נניח והשתמשנו בקובץ הרצה שלנו, וכך בקובץ הרצה שלנו הוכנס אליו הקוד של הפונקציה. אולם, אם גילו באג בפונקציה של *c* זה לא ישנה אצלם. כלומר בשבייל שהקוד של הפונקציה יהיה הכיל ערך, נטרך ליצור קובץ הרצה חדש.

2. חסרון שני - שימוש מיותר באוצרון. אם למשל אנו כותבים 10 תוכניות שונות שימושות ב-*printf* התקבלו לנו 10 קבצי הרצה שונים שככל אחד מהם יש וכשרץ את קובץ הרצה יהיה באוצרון 10 מופעים של הפונקציה, חבל על המוקם.

ב. ספרייה *dynamic linking* . לכל פונקציה או ספרייה בעולם קיים עותק ייחד בזיכרון, נרצה פשוט לנשתן אליו באשר אנו מניחים שלכל פונקציה יש צ'אנק כלשהו בזיכרון. יש בעיות בכך - אף אחד לא אמר שבעל רגע נתנו לנו מושגים מסוימים בכל הפונקציות שאי פעם נכתבו. חסרון נוסף ובע מ מצב בו אולי יורדות גרסאות מסוימות לפונקציות, נרצה את המכבי מעודכנת. נראה כי יש מצב שהגרסה העדכנית של *printf* בגודל יותר גדול - יותר בייטים, אבל מתחת צ'אנק של *printf* הנוכחי יש קוד וגם בצ'אנק מעליו יש קוד שכן נאלץ להפוך מקום חדש בזיכרון לכל הפונקציה החדשה. זה לוקח זמן!

לכן נשמש *dynamic linking* שפועל כך: ב-*static* אמרנו שהציגו מתרחש באמצעות דחיפה של הקוד לקובץ הרצה, ב-*dynamic* הציגו מתרחש באוצרון עצמו. איך? באחת משתי הדרכים הבאות -

א. פירשו, שבזמן טעינת התוכנית שלו לaczron, מיד הקוד שלנו יטען ייחד אליו נקרא *dynamic linker* שקורא את כל הספריות החיצוניתות בהם הקוד שלנו מתרחש, הם נטענות לaczron. בדיק ייחד עם התוכנית שלנו.

ב. *run time*: כאן, אנחנו לא נטען ישר את כל הספריות לaczron, אנחנו נחבר רק לפחות הפעם הראשונה *symbol* בעת הרצת התוכנית, ואז ה-*linker* יטען את *symbol* הנוכחי בaczron.

**בעת נשאל את השאלה הבאה:** האם עדין *run time* או *load time*?

בהמונ פונקציות, אך בغالל תנאי `if` זמן רק 3 פונקציות. במקרה זה עדיף לנו כמובן `runTime` لكن: ב-99% מהמקרים נעדיף `runTime`, אך צריך להפעיל שיקול דעת.  
נראה כי המטרה של *dynamic linker* לינקינג היא שככל שלב בזכרון יהיה עותק אחד של כל `symbol`.  
נראה כי המטרת *dynamic linker* היא למשל פעם שנייה `printf` הוא לא יטען אלא ינש  
לעתוק הקודם שהוא טען.

**חשיבות מואוד:** נראה כי אם יש משתנה גלובלי ש-10 תוכניות משתמשות בו - במקרה זה יטעןו לזכרון 10 עותקים שונים (!) גם בדינמי לינקר, בשונה מפונקציות שנטענות רק פעם אחת.  
לאן *dynamic linker* טוען את הכתובות? לאזרז בזכרון שהוא בין `heap` ו-`stack`. נשים לב כי תיכון כתבת קוד שמשתמש ב-`scanf` שעד לא היתה טעונה בזכרון, לכן הרצתי והפונקציה נתענה לזכרון. נשים לב כי יתכן שנרצה בפעם אחרת את התוכנית, היא תעטען למקום אחר בזכרון. אף אחד לא אמר שהמקומות שהיא נתענת עליו ישאר קבוע.

ב-`bs`'`flag` שאנו מעבירים ניתן לקבוע איזה סוג *linking* נקבע וכן האם `runTime` או `Lazy bidding` זה נקרא `runTime` *dynamic default* הוא.

#### (Position Independent Code) PIC 5.4.2

כפי שראינו, הרצת הפונקציות קודם יוכל להטען לכל מקום שהוא בזכרון.  
נססה להבין איזה טריקים וטיקטים הקומpileר מבצע כשהוא רואה קוד `C` עד שהוא הופך אותו לקוד *PIC* באסמבלי:  
מחלק טריקים אלו לשניים.

**א. קוד שניגש `symbol` פנימיים בלבד (רק לפונקציות או משתנים גלובליים או סטטיים שהוגדרו באותו קוד בלבד):**

*Pc – relative addressing*: גישה לכתובות של `symbol` לא באמצעות הכתובות האבסולטית שלון בזכרון אלא המרחק שלhn `rip` כרגע. זה נקרא באסמבלי בצורה זו `%rip`, `%rax`.  
כיצד הלינקר יודע לחשב את המיקום של כל כתובות? נשים לב כי בין `data` `text` ישנו מרוחק קבוע בזכרון. לכן נניח וישנה `globalY` שנמצא כתה אך שומר `data`. אנו יודעים את המיקום היחסי מתחילה `text` עד `globalY` וידועים את המרחק מתחילה `text` עד תחילת `data` וכן יודעים את מיקומו היחסי של `data` `globalY`. נחץ ערך זה, ואת הערך של המרחק מתחילה `text` עד `data` ונחסר את המיקום של `text` `globalY`. זה ההפרש בין המיקומים וכך דע לעבר בינהם.  
**ב. קוד שניגש `symbol` חיצוניים (תיכון גם פנימיים):** כבר בזמן קומPILEציה אנו יודעים שיש לנו `symbol` חיצוניים.

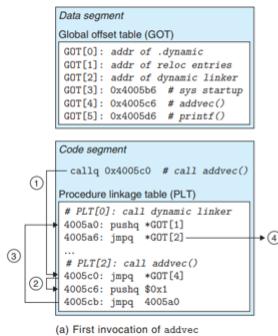
ישנה טבלה בשם *GOT*: *Global Offset Table*. טבלה שמתווספת בכל `data segment`,  
שורה ישנים 8 בתים. בעיטה שלילינקר טען `symbol` הוא יקח את הכתובת של `symbol` שהוא טען  
וישם בשורה בטבלה. כיצד זה עוזר לנו? אנו מנסים לגשת ל-`symbol` חיצוניים אך לא יודעים היכן הם  
נמצאים. כתה באמצעות הטבלה וכל מכם בקוד שנינו לגשת אל `GOT.symbol`. כיצד נוכל בצורה  
שהיא *PIC* לגשת אל `symbol`? כמו קודם - ככל לחשב את מיקום. נשים לב כי לפעמים בזמן  
קומPILEציה נוצרם ב-*GOT* סימבולים פנימיים גם - וכך אפשר לשאת אליהם צורה בדיק (אם  
כי אכן אין צורך לרשימות *GOT* עבור סימבולים פנימיים, אך זה קורה).

#### PLT 5.4.3

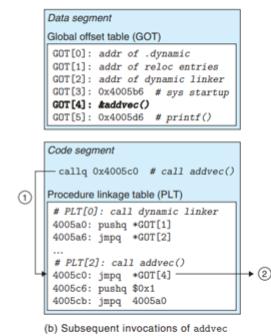
משמשת אותנו לגישה אל סימבולים חיצוניים וכן פנימיים. *PLT* היא טבלה שעוזרת למימוש  
של `lazyBiding`. היא נמצאת ב-`text segment`. מורכבת מרשימות של 16 בתים שככל אחד מהם  
מורכב מ-3 פקודות באסמבלי בלבד שמאפשרים את הבדיקה: האם הפונקציה כבר טעונה או שאחננו  
צריכים לקרוא dynamic linker' שיטען אותה.

נראה כי יש הבדל בפעם הראשונה שטוענים פונקציה לבין פעמים אחרות. באשר מבצעים  
קריאה לפונקציה `advac` קוראים לערך `Got`.

### First invocation of addvec



### Subsequent invocations of addvec



נראה כי אם הלאנו לרשימת *GOT* של *advac* עוד לפני שטענו אותה לארכוון, מותבצע כך שמא. נראה כי לאחר שטענו אותה לארכוון, במיקום *GOT[4]* אכן *dynamic linker* הכניס לשם את הכתובת של *advac* שכבר נטענה. נראה כי תמיד בעת שתקרה *advac* לא משנה באיזה פעם נבצע את *.jmpq \*Got[4]*.

התפוקיד של *PLT* הוא לדעת להבחן האם פונקציה כבר נטענה, ולא צריך לטעון אותה שוב *bequests* *runTime* אלא רק לגשת אליה, או צריך לטעון אותה.

סה"כ באמצעות *PLT* ו*GOT* הקומפיילר משתמש בהם לביצוע הטריקים שלו - וכך הוא מבצע *PIC* ומישיג *Lazy bidding*.

### *Patching 5.4.4*

כיצד נקבע קובץ הרצתה, ונשנה ממש כמו בתים בו וכתוכאה מכך הוא יתנהג אחרת. (להזכר בדוגמה עם הסטודנטים במובוא שיצירו קובץ חשוב - הרסו אותו כי הם לא חכמים במיוחד, ואני כעת צריך לחתת את הקובץ שם נתנו לי ולסדר אותו.)

## 6 הרצתה 6

### *miss 6.1*

הוא מצב שלא מצאתי *cache* בעט החיפוש *cacheLine* אחר מידע אני נזק ללקת *RAM*.

א. *miss : cold miss* שטוען שהוא לא הכיל עדין את המידע הספציפי זהה. תמיד יהיה לי *miss* ככל שcn בהתחילה לא הבאת *cache* כלום. תמיד קורה בפעם הראשונה שנגשים לנו (נחפש *cache friendly* ואין שם כלום). **כמעט ואי אפשר לטפל בו - בהמשך נדבר שאותן הקוד אן אין אפשר לטפל בו.**

ב. *conflict miss*: יש לנו מצב של דרישת - מבאים בлок מהזכרנו אל *cache* והוא דורש מידע אחר. באשר *block* מזכיר מומפה למקומות תפוס *cache* ותווסף מידע שומר שם. זה קורה בכל אילוצי מיפוי, יש לנו שני נתונים שמתחרים על אותו מקום. **זה אומר שהקוד לא cache friendly**.

ג. *capacity miss*: מצב שאומר שה**cache** קטן מדי בשbill להכיל את כל המידע שאינו צריך. למשל: אם יש לנו הרבה קוד ש חוזר על עצמו, הרבה LOLAOOT ומידע. לצערנו *cache* קטן מדי ולא

יכול להכיל את כל *active cache blocks* הווה  $16KB$  והמידע שלו בגודל  $30KB$ . **אין לנו מה לעשות איתו.**

דיברנו על כך שה $set$  קבוע לפי הביטים האמצעיים. נציג שתי אלטרנטיבות כתע - בואו ונכח את הביטים המניעים ביותר או השמאליים ביותר. האם הם אלטרנטיבות טובות?

ההצעה עם הביטים המניעים ביותר - על הפנים. נשים לב שם ישר נסתכל על השני שורות הראשונות נשים לב שיפגע *locality principle* שכן דברים שיופיעו אחד אחרי השני בקוד יופיעו במקומות שונים.

ההצעה עם הביטים השמאליים ביותר - על הפנים. נניח שיש לנו כתובות  $64bits$ , אם נסתכל על הביטים השמאליים ביותר והם בהתחלה אפס, אנחנו נקבע שיש המונוטוניות כתובות עד שהביטים השמאליים ביותר יתחלפו, אז נקבל שיופיעו לנו *cacheLine* שכן דברים שיופיעו אחד אחרי השני בקוד מופיעים.

לכן, בחרנו בשיטה שראינו עם הביטים השמאליים: ההסתברות להתגשות תהיה הכי קטנה.

High-Order Bit Indexing	Low-Order Bit Indexing
0000	0000
0001	0001
0010	0010
0011	0011
0100	0100
0101	0101
0110	0110
0111	0111
1000	1000
1001	1001
1010	1010
1011	1011
1100	1100
1101	1101
1110	1110
1111	1111

*write - hit*: מצב בו אנחנו רוצים לכתוב, למשל לבצע  $x = 3$  (כתיבה בלבד) וגילינו כי  $x$  נמצא *hit - cache* ב-*cpu* יש שתי אפשרויות: אחת *write - through* היא לעדכן ישירות את *RAM* או לחלופין *write - back*: לא לעדכן את *RAM* כרגע, לשם כך נשתמש ב-*dirty bit* ונעדכן אותו בהמשך. *CPU* שלנו יש מדיניות של *write - back*. אין לנו רצון לנשתתף *RAM* הרבה.

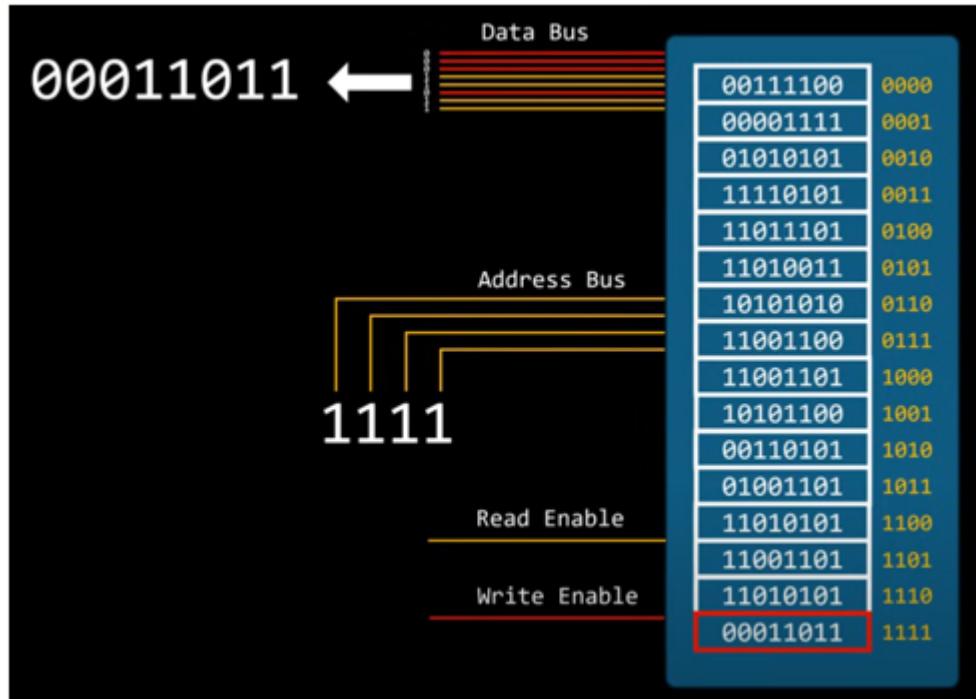
*write allocate*: כשבושים למשל  $+x$ , טוענים את הבלוק מהזיכרון ל-*cache*, אחר כך כותבים אליו בתוך *cache* (שם עושים את העדכון) והבלוק נשאר ב-*cache* לגישות עתידית.  
*no write allocate*: כשבושים למשל  $+x$  כותבים ישירות לאזרכון הראשי. לא טוענים את הבלוק בכלל ל-*cache*.

**רוב - נרצה write back וכן write allocate**

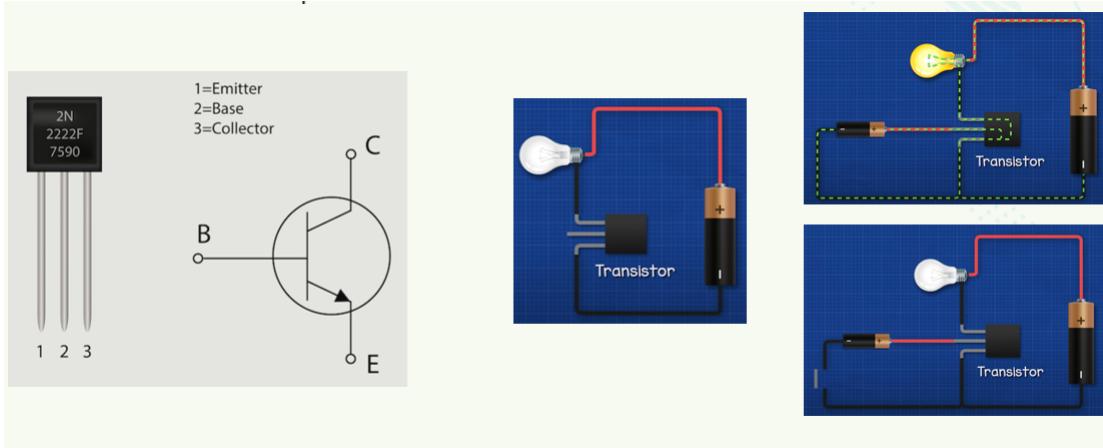
*Miss rate*: נניח עשינו 100 גישות 101 הוו *hit* אז אחוז ההצלחה הוא 10%. החלק היחסי של כמה פגעו מתוכן כמה נגשנו לאחסן. לרוב ב- $L_1$  מדובר על 10% – 10% ובסך מ-3%.  
 נשיכת לב - שזה בתנאי שהקוד הוא *cache friendly*.  
*Hit Time*: כמה זמן לוקח לנו להלץ את הנתון מ-*cache*. בדרך כלל: 4 ננו שניות" ב- $L_1$  ו-10 ננו שניות" ב- $L_2$ . אם כן, זה עדיף על גישה לאזרכון שעולמתה 200 – 50 "ננו שניות".

## RAM structure 6.2

נדבר על סוגי זכרון שקיים לנו במחשב. לא נססה את כולם כמובן. נרצה להבין כיצד *RAM* מחובר לכל דבר אחר במחשב. בינו לבין ה-*CPU* באופן ספציפי. מהו *BUS*? 64 חוטים שלל כל חוט עובר ביט. באמצעות *cpun read, write enable* מוחלט האם הוא מעוניין לכתוב בזכרון או רק לקרוא ממנו. 1. אם *CPU* רוצה לקרוא משהו *RAM* הוא שם *Address Bus* את המידע, מאפשר קריאה ממנו ומשיג את המידע באמצעות *Data Bus*. 2. אם *CPU* רוצה לכתוב משהו *RAM* הוא שם *Address Bus* את המידע, מאפשר כתיבה אליו ושם את המידע *Data Bus* ב- *cache* - אל החלק הרלוונטי לרגיסטר שהפקודה משתמשת בו.

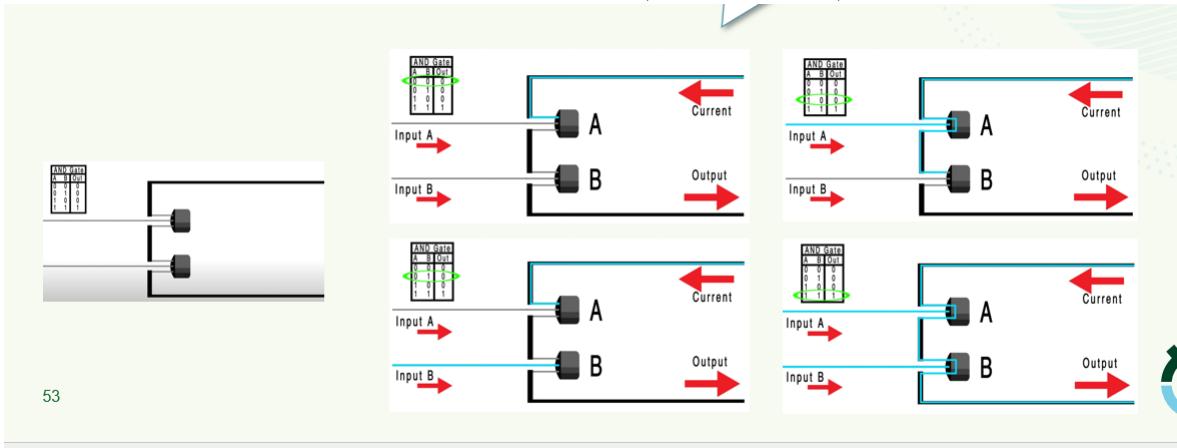


טרנזיסטור הוא רכיב חשמלי שמתפרק כמתג חשמלי או מגבר - הוא יכול להעביר או לחסום זרם חשמלי בהתאם למתח שmorphעל עליו, והוא אבן הבניין הבסיסית של כל המעבדים והמעגלים הדיגיטליים המודרניים. כפי שראיתם בדוגמה מטה - בשbill לחדיל את הנורה צריך לספק לטרנזיסטור חשמל (סוללה כלשהי למשל). השן האמצעית של הטרנזיסטור קובע האם הוא יקבל חשמל. אם קיבל בשן האמצעית - הוא עבר את החשמל.



באמצעות ההבנה על הטרנזיסטור, שהוא במצב של *off* או *on* מעביר חשמל או שלא. נוכל להבין שניתן לתרגם 8 טרנזיסטורים אל *Byte*! כל ערך של הטרנזיסטור הוא כן או לא, שכן זה כן מעביר חשמל: ביט 1 ולא זה לא מעביר חשמל: ביט 0. כפי שראינו בהרצאה הראשונה - יש או אין חשמל זה לפי טווח מסוים. גם אם יש חשמל לא בטוחה מאוד גבוהה וגם אם אין זה אומר שיש בטוחה מאוד קטן.

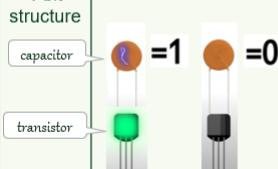
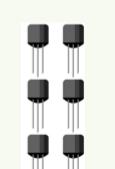
מכאן התובנה שכל האוצרו מורכב מטנזיסטורים. הטרנזיסטורים בונים לנו שערים לוגיים:



לצורך העניין נסתכל על שער לוגי *AND*. אנו מכירים כבר את טבלת האמת. נרצה להבין כיצד שער לוגי נבנה מטנזיסטורים. ישים שני טנזיסטורים, וכי שאמורנו הם מייצגים שני *input* שהם ביטים, וכן השן האמצעיתו המידיע שעובר לנו מהם. אם עبور זרם דרך שני הרגיסטורים, משמעו שני השינויים האמצעיים יועבר בהם זרם, וכך "הנורה תדלק", כלומר שני הטרנזיסטורים ידלכו, וכותזאה מכז זרם זרם ולכן זה ייחזר 1.

דבר על שני סוגי של *RAM*.  
*S – RAM*: *ram* סטטי. החשמל מיוצג (כל ביט) באמצעות 6 טנזיסטורים - מודיע? החשמל נוטה להחלש והמקטינים את הירידה בחשמל. (איך? לא רלוונטי לקורס). כתוצאה לכך הוא עולה הרבה יותר - כי משתמשים פי 3 בטרנזיסטורים מאשר *D – RAM*. הוא גם יותר גדול פיזית - הנפה שלו יותר גדול, אך הוא מהיר הרבה יותר.  
*D – RAM*: *ram* דינמי. החשמל מיוצג (כל ביט) עם שני טנזיסטורים בלבד. כן או לא יש

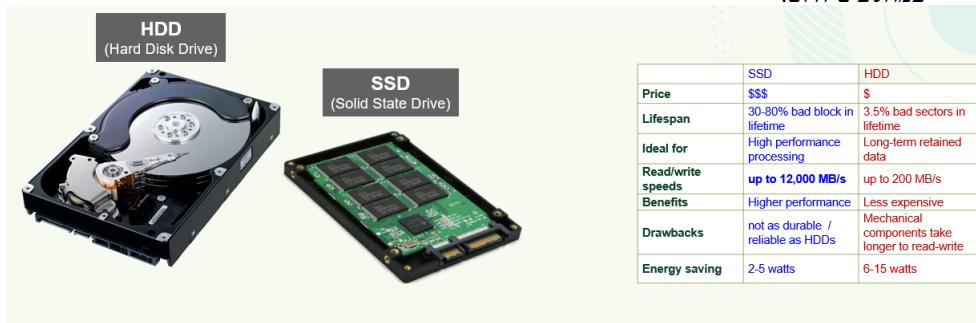
חסמל. כן חשמל 1 אין חשמל 0. הוא איטי הרבה יותר מאשר  $S - RAM$ .

	<b>DRAM</b>	<b>SRAM</b>
access	slow (~100 nsec)	fast (~10 nsec)
capacity	high	low
cost	\$	\$\$\$
1 bit structure		
usage	RAM	Cache

כעת הבנו מדוע *cachen* קטן - הוא עשוי טכנולוגיה של  $S - RAM$  ולכן יקר יותר, וכן נראה שהוא יהיה קטן מכמה שיטות, בשל שהינה מהיר מאוד.

### Disk structure 6.3

ישנם שני סוגי דיסקים: HDD הוא הדיסק הישן, SSD הוא הדיסק החדש הטכנולוגיה החדשה שיש במחשבים היום.



ברור כי SSD יקר יותר, אך לא עד כדי כך יקר כי בכל לפתוף יש היום. אכן HDD יותר זול ואם אנחנו צריכים לשמר מלא מלא נתונים - עדיף HDD.

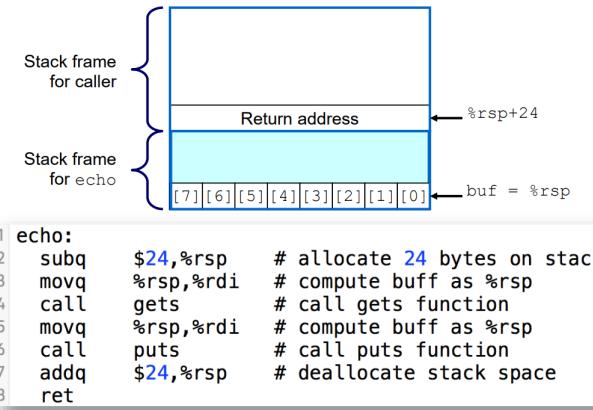
### תרגול 6.4

בבLOCK *buffer overflow*: בstack שומרים מידע וערך חוזר לפונקציה. נראה כי ניתן ותיהה זליגה מה מקום שהוקצתה. למשל בפונקציה הבאה - הקצתו ב-8buf *get input* והכנסתי אל בגודל 20, אני בהכרח יצא מהמקום שהוקצתה לי - כיון שהfonקציה הנ"ל מנהה כי מספק גודל:

```

1  /* Implementation of library function gets() */
2  char* gets(char* s)
3  {
4      int c;
5      char* dest = s;
6      while((c = getchar()) != '\n' && c != EOF)
7          *dest++ = c;
8      if (c == EOF && dst == s)
9          /* No characters read */
10         return NULL;
11     *dest++ = '\0'; /* Terminate string */
12     return s;
13 }
14
15 /* Read input line and write it back */
16 void echo()
17 {
18     char buf[8]; /* Way too small! */
19     gets(buf);
20     puts(buf);
21 }
```

דוגמה נוספת, היא כאן. נשים לב שהקצנו מקום עבור 24 בתים בלבד. אם נכניס 25 בתים ומעלה מה שיקרר זה שהערכים אכן יוכנסו למחסנית, אך הם ידרשו את הכתובת חזרה *rip* ולא יוכל לחזור להיכן שאנו צריכים לחזור בסוף הפונקציה.



את מושיות התקיפה הנפוצות ביותר במערכות מחשב ברשת. *Buffer overflow attack* (באנגלית *buufer attack*) הנקפה מאגר קוד זמני (*exploitCode*) יחד עם שימושים שימושיים את כתובות החזרה כך שיצביע על הקוד החדשני. כשהפונקציה מבצעת *gets*, היא קופצת לקוד התקוף ובמצעת אותו - זו

מדוע יש לנו חלקיים של *data*, *text* וכו'? בעיקר בשbill הגנה ממתקפות זדוניות שכאלו.

## 7 הרצאה 7

### cache friendly code 7.1

- נתבונן בדוגמה פשוטה. נניח שיש לנו מטריצה  $M$  בגודל  $n \times n$ . נרצה לחשב סכום של כל איברי המטריצה. אם המטריצה קטנה: לא אכפת לנו איך נקרא אותה. אבל נניח כי  $n$  גדול מאוד, זו תהיה הנחה שנדרץ בכל מקרה בנושא האופטימיזציות, מניחים:
1. שהמטריצה ענקית
  2. מניחים *cache* לא מספיק גדול בשבייל להכניס את כל המטריצה לתוכו
  3. מניחים *cold empty cache* - כלומר אין שום דבר שלא רלוונטי לתוכנית.
  4. לצורך ההמחשה - נניח 16 ביטס *cacheLine*

**הקוד עבר על המטריצה לפי שורות?** מה *Miss rate*? מה *cache* גישות ל-*cache* כמה פעמים נאלצנו ללקת לזכרו. כיון שאנו מניחים שגודל שורה קаш היא 16 בתים - כל 4 גישות למערך אנחנו צריכים ללקת *ram* להביא *cacheLine* חדש ולכן  $\text{missRate} = \frac{1}{4} \times 100 = 25\%$ . אם הקוד יעבור על המטריצה לפי עמודות - *Miss rate* הוא 100%, אנחנו בכל פעם נביא שורה, וכיון ש-*N* גדול מאוד, כל אלמנט נמצא *cache line* אחר. לכן בודדות תמיד נאלץ ללקת לזכרו.

**ומה באשר לנכפל מטריצות?** נשים לב שאלגברית, כפל של שתי מטריצות, הוא לקיחת שורה במטריצה *A*, ולקחת עמודה במטריצה *B*, להכפיל אותן ולקבל איבר בודד במטריצת המכפלה. נשים לב שאנו בבעיה - מבחינת *cache friendly* אנו עוברים על עמודה. זה לא נכון. אם נסתכל על קוד שמכפיל מטריצות, זה נראה כך:

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (k=0; k<n; k++)
            c[i][j] += a[i][k] * b[k][j];
```

מה  $C = A \times B$  עבר *Miss rate*  $A$  ו-  $B$  ?  
עבור  $A$  - בדיק  $25\%$ , כמו קודם.

עבור  $B$  - בדיק  $100\%$ ,因为我们 *cache* עמודה, כמו קודם.

עבור  $C$  - ניתן לכורא למתעלם מההישוב. *Miss rate* של איבר  $C$  במטריצה  $C$  הוא  $\frac{1}{n}$  פעמים (במהלך החישוב, ניגש אליו בולולאה האחורונה  $n$  פעמים) ולכן הסיכוי הוא  $\frac{1}{n}$  וכיון ש-*n* גדול מאוד, אז *Miss rate* שואף לאפס שכן נגד שהוא  $0\%$ .

<i>Pipeline friendly code</i>	<b>7.2</b>
<i>RAM friendly code</i>	<b>7.3</b>
<i>compiler &amp; optimizations</i>	<b>7.4</b>
<b>הרצאה 8</b>	<b>8</b>
<b>הרצאה 9</b>	<b>9</b>
<b>הרצאה 10</b>	<b>10</b>
<b>הרצאה 11</b>	<b>11</b>
<b>הרצאה 12</b>	<b>12</b>