

מבנה מחשב - סיכום הרצאות למבון

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

23	<i>Jump Table</i>	3.6	
25	<i>GDB</i>	3.7	
26	4	הרצאה	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 ורוכסן?	4.3.1	
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	ארגומנטים ב- <i>STACK</i>	4.7.1	
32	<i>Variadic Functions</i>	4.7.2	
33	קבלה ארגומנטים בשורת ההרצתה	4.7.3	
33	<i>Flow Control</i>	4.7.4	
34	<i>memory hierarchy</i>	5	הרצאה
34	הקדמה	5.1	
35	<i>cache</i>	5.2	
36	<i>organization of a cache Memory</i>	5.3	
38	<i>cache</i>	5.4	סיכום
39	תרגול	5.5	
39	<i>static linking</i>	5.5.1	
40	(<i>Position Independent Code</i>) <i>PIC</i>	5.5.2	
41	<i>PLT</i>	5.5.3	
41	<i>Patching</i>	5.5.4	
41	תרגול	6	הרצאה
41	<i>miss</i>	6.1	
43	<i>RAM structure</i>	6.2	
45	<i>Disk structure</i>	6.3	
46	תרגול	6.4	
47	7 + 8 : אופטימיזציות	7	הרצאה
47	<i>cache friendly code</i>	7.1	
48	<i>Pipeline friendly code</i>	7.2	
52	סכימות תאי מטריצה	7.3	
54	<i>RAM friendly code</i>	7.4	
55	<i>compiler & optimizations</i>	7.5	
56	מה הקומפילר לא יכול לעשות?	7.6	
57	שלבים בדרך לכתיבת תוכנית אופטימלית	7.7	
58	<i>Measurement challenge</i>	7.8	
58	תרגול	8	הרצאה
58	9	הרצאה	9
58	10	הרצאה	10
58	11	הרצאה	11
58	12	הרצאה	11

1 הרצאה 1

1.1 מבוא לקורס

הקורס יתמקד בשני תחומיים:

1. מבנה מחשב - חומרה
2. שפת מחשב שנוגעת לשירות בחומרה - *Assembly*

מתי צריך להשתמש באסמבלי? כאשר אנחנו רוצים למשל לחשב חישוב מהיר מאוד - שבכל מקום אחר החישוב הזה ית��ע. זו שפה שהיא כמעט שפת מכונה" - *Low Level*.

האם $0 \geq x^2$? לא בהכרח - במתמטיקה כן, בודאי ב \mathbb{R} . בעולם התאורטי, זו טענה נכונה. במקרה: ראיינו כבר כי ביחסיתן מס' מאד גדול, למשל $x = 1410065407$, נקבל כי $x^2 < 0$ – מודיעו? כמו שנלמד במבוא – יש *overflow* והיצוג הופך לשיליי. אז כיצד נתמודד עם זה? נדון בכך במהלך הקורס.

האם מתקיים $(y + z) + x = y + (z + x)$? בעולם התאורטי, כן. עם זאת – לא כל מספר עשרוני ניתן לשימירה במחשב. למשל אם כמות הספרות אחורי הנקודה גדול מכמות הספרות שאפשר להחזיק, המחשב חותך את הספרות שהוא לא יכול לשמור – אז הוא מקבל מס' שאינו מדויק. וכך, לא יתקיים השוויון הנ"ל במחשב בשל הסטיות הללו.

וכאן הדגש: **בתאוריה המתמטית – לא יוכל شيקו בעיות כאלו. בעולם האמתי, במחשב: זה לא גמור קורה. ועלינו ללמידה כיצד להתמודד עם זה.**

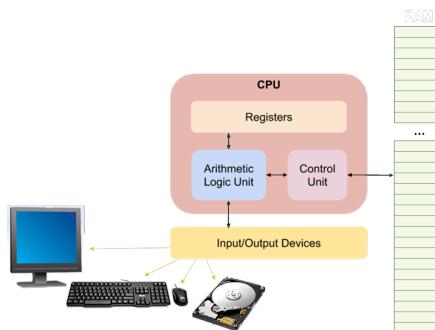
נתבונן בקוד הבא:

```
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 נבון (*CPU*) שהוא רץ הרבה יותר מהר: **זה כל מה שחשיב, עיליות.**

COMPUTER STRUCTURE 1.2

המחשב בנוי מ*מערכת יחידת המחשוב*, *CPU*, *זיכרון* (*RAM*) ו裝置 (*Input/Output Devices*).



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

זיכרון – *RAM*: מרכיב מביתים. כל בית מורכב מ8 ביטים. מעין מערך" שיכול לספר עד

1 – 2^n ערכים. כל בית באכרון הוא עם ערך כלשהו – גם אם שמו את זה שם וגם אם לא (יקבל ג'אנק). הביט הימני נקרא *LSB* והביט השמאלי נקרא *MSB*.

- *Word* – שני בייטים רציפים.
- *Long/dword* – ארבעה בייטים רציפים.
- *qword* – שמונה בייטים רציפים.

1.2.1 מודיע המחשב שלנו בינהiri? למה שהמבחן לא יהיה טרינארי?

כאשר מעבירים מידע ממוקם למקום (בתוך המעבד) נשלח *signal* חשמלי. אנחנו רוצים לתרגם את המידע שיש בסיגנל החשמלי לביטים. סיינל מגע עם רעים – כמו כאן בתמונה טווה. מגדירים טווה של עצמת הסיינל: בין 0 ל/2/0 הוא מתפרש כבית 0, בין 0.9 ל-1.1 מתפרש כ-1, ובכל השאר הוא מותפרש כמצב מעבר.

באופן תאורטי – יכולו את מצבי המעבר להגדיר כמצב השלישי – ואז לעבור עם בית" שלישית, במצב טרינארי: מהר מאוד מופיעו זה, כיון שהוא הרבה יותר רעים ותוננות ואז במקומות לחשוב שקייבת מס' 8 קיבלו 9. נוצרו הרבה בעיות כתוצאה מרעיו זה – וכן בשבייל להבטיח את תקיןויות *dataan* משתמשים בביט ולא בטריטו.

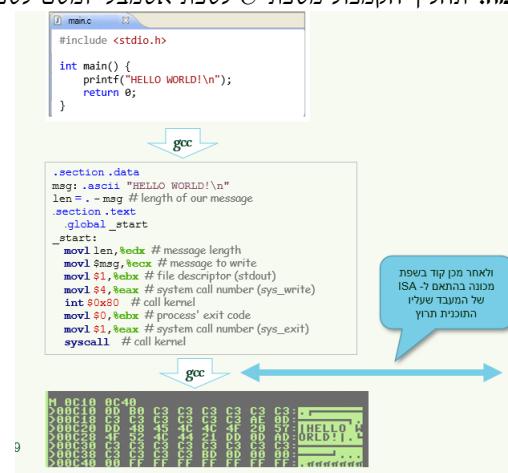


Instruction Set Architecture - ISA 1.2.2

(MRIINA אמרה **שהיא שואלת לפעםם במחון מה זה**). לכל דגם של *CPU* יש אוסף פקודות שהוא ידוע לבצע. אוסף פקודות נקרא שפת מכונה. לכל פקודה כזו קיימת מקבילה שקיימת בשפת *ISA*. *ISA* הוא ספר"ר שמרכזו את כל הפקודות שאותה ארכיטורה / *CPU* ידוע לבצע. *ISA* אסמבלי. דוגמים שונים של מעבדים יכולים להיות שונים. בהינתן שנדע את הספר הנ"ל – נדע איזה פקודות יוכל לכתוב בשבייל לכתב את הקוד שלנו. הפקודות בספר יהיו כתובות הן בשפת אסמבלי והן בשפת מכונה.

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

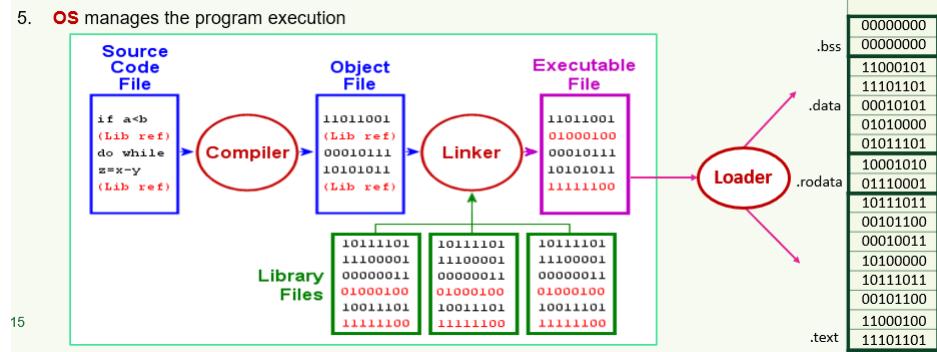
לדוגמא: תהליך הקמפול משפת *C* לשפת אסמבלי ומשם לשפת מכונה.



1.3 תהליכי הרצה ותמונה הזיכרון

1. **שלב ראשון:** כתבו קוד. נקרא לו *Source*
2. **שלב שני:** הקומpileר מבצע תהליכי קומPILEציה, מזהה שגיאות קומPILEציה והופך אותו לשפת מכונה.
3. **שלב שלישי:** הלינקר, מבצע תהליכי קישור בין הקוד של לקודים אחרים שקיים בספריות אחרות בהם אנחנו השתמשנו בקוד, זהו קוד שכבר מוכן ומkompile' - והוא מאחד אותם לקובץ יחיד שיקרא *Executable* שמוון ל�מפל.
4. **שלב רביעי:** בטרמינל, אנחנו כותבים למשל *\a.exe*. > : למעשה, מה שקרה הוא שאנוחנו אמרנו למערכת הפעלה - קח את הקבצים שכתבת לך ב- *Loader*, תשתמשי בהם. *Loader* : (*"טען"* - תפקידי לטעון את הקובץ לזכרון. **ראשית** הוא מודוא שקובץ זה ניתן להרצה ע"י מערכת הפעלה שלנו. **שנייה**, הוא מפרשר ("חוותך ושם בכל מקום בזיכרון מה שצריך לשבת שם") - בדיקות כמו שבבדיקת מבחר, הבודק מודרג על איזו החראות. למעשה *Loader* יידע על מה עליינו לדגל ומה הוא צריך לקרוא, את מה שהקומPILEר והלינקר עשו לו לקובץ, וההפרסר) את מה שכתוב לו, והוא צריך להבין באיזה מקום של הקובץ מופיע *data* ולשים את זה באיזור *data* בזיכרון, להבין מה צריך להכנס *text* בזיכרון ולשים את זה באיזור זכרון וכן הלאה). *תקפיך נוסף שלו* - הוא לאתחל את המשתנים בהתחלה.
cut לאחר שהשתמשנו ב- *Loader* קיבלנו *Process Image*: מכלול של זכרון (תמונה זכרון) שモוקצת לטובת התהליכי Loader שארחיו לו. כמובן, כל הזכרון שモוקצת לטובת התהליכי במהלך ריצת התהליכי.

5. OS manages the program execution



משתנה גלובלי (מוקצים ב- *data*). כאשר כתבנו משתנה *x int* ולא נתנו לו ערך.
משתנים גלובליים שמאוחלים כבר עם ערכים. *data*.

משתנים גלובליים סטטיים שהם *const* והםicut רק נקרים.
אזור הקוד של התוכנית. *כמובן ספר*. בעת קריאת קוד קוראים מיל.

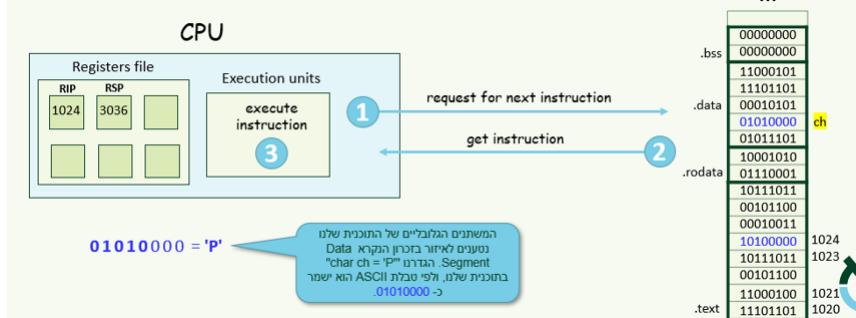
כאשר כתבם פונקציה, נפתח *activation frame* בו מוקצים כל המשתנים של הפונקציה, בעת סיום הפונקציה עם *return* הפריים נסגר לטובת הפריים הבא שייפתח.

1.3.1 איך הCPU ידע מה רצף ביטים מייצג?

הCPU מבקש *instruction* הבא, מקבל אותו ומבצע אותו. ברגע שהוא סיים את הזרואה הנוכחית, הוא מקבל את הבא ומבצע אותו. הוא מאוד מורכב - אך בפועל הוא עובד בצורה פשוטה. *Loadern* מכיל מידע אודות היקן *main* ב- *.text*. הוא ידע את הכתובת *instruction* הראשון שהתוכנית צריכה לבצע - בעברית: נקודת כניסה". *Loadern* יזהה במהלך תהליכי הפרסור את המיקום הראשון שמננו התוכנית תחיל. *(Instruction Pointer) RIP* רגייסטר. הוא הרגייסטר שהCPU לא יכול בלעדיו. זהו רגייסטר שמצויב על הבית הראשון של הפקודה הבאה לביצוע. *Loadern* שם בתוך הרגייסטר ממש את

הכתובת הזו במהלך תהליך הפרשור. (הערה - גם באסמבלי נוכל לגשת לרגיסטרים בזיכרון, מה שאי אפשרות בשפות עילית).

ל-*CPU* יש ביד את *ISA* - הוא מקבל את הכתובת הראשונה בתוכנית והוא פותח את *the ISA*, והוא בודק האם קיימת *ISA* פקדת עם הכתובת זו. אם כן הוא מבצע אותה - ואם לא הוא מתקדם לכטובת הבאה בזיכרון, הוא מקדמי את המיקום בזיכרון אחד (אל הבית הבא). אם הוא מקבל פקדת פקדת - הוא חוזר לשלב 3, ומבצע את הפקדת. לאחר מכן הוא ממשיך להעלות אחד בכל שלב ומתקדם בזיכרון.



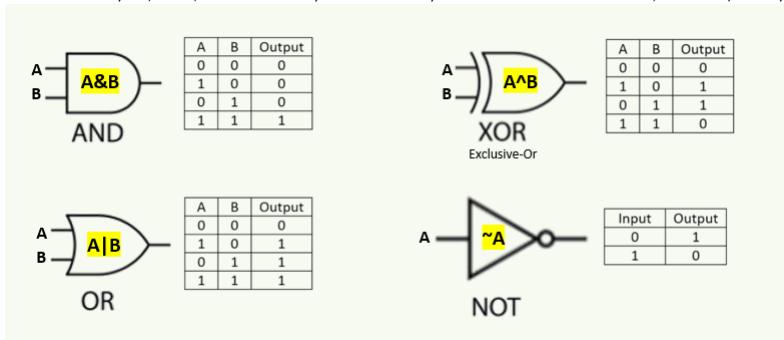
המעבד קורא תחילת (prefix) של פקודת, מזאה שהה פקודת בזכות *ISA* ומפענה אותה.

היעריה: רגיסטר *Stack Pointer* (*RSP*) נמצא בתוכה *CPU*. מצביע על תחילתו (כיוון שאם הערך גדול מדי, לפि מבואו הוא נשמר מלמטה בזיכרון) של הערך האחרון שנכנס למחסנית. המחסנית חשובה מאד כיוון שיש בה משתנים מקומיים, *data* וכתובת חורה וכן ארגומנטרים שפונקציה מקבלת (אם ואחר).

היעריה: בתוך *CPU* ישנו מס' רגיסטרים שיושבים באורו *Registers file*. למשל: *RIP*, *RSP*, *RIP*. המשתנים הגלובליים של התוכנית שלנו נטען לאזור הנקרא *data*, אם הגדרנו את 'p' למשל בתוכנית - לפि טבלת *ASCII* הוא ישרם בהתאם אליה.

Bit Level Operation 1.4

בהינתן אוסף ביטים, נפעיל פעולה ביוטוייז בין ביטים. ישנן מס' פעולות, כדי קלמן:



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

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

דוגמה. **יצירת קבוצה באמצעות ביטויים**. נרצה ליצג וקטור באורך n באמצעות בית: $\{1 - n, \dots, 0\}$. כל ערך בוקטור יכול להיות משוייך לקבוצה יוכל להיות שלא. הקבוצה תהיה A . אם $i \in A$ אז $a_i = 1$. כלומר – אם איבר בקבוצה מיקומו בוקטור יהיה 1. אם נרצה להוסיף ערך לקבוצה – נבצע פעולה *or*: מדוע? נסיף בית עם הערך 1 על 1 וכל השאר אפסים, וcut שונעשה *or* נקבל קבוצה חדשה עם 1 בתוכה. אם נרצה למצוא את הקבוצה המשלימה – \bar{A} : נבצע *not* על הביטוי. **לחיתוך** שתי קבוצות – נשתמש באופרטור *,and*, **ולאיחוץ** שני קבוצות – נשתמש באופרטור *or*.

חשוב לדעת: פעולות ביןaries הן הפעולות המהירות ביותר שנitin לבצע.

1.5 תרגול 1

ישנם מס' כלים שונים להקל את החיכים של המתכונת. *Nano, Notepad, Vim, Emacs :Text Editor* *IDE*: סביבת עבודה שגמינו ניתן לכתוב בה את הקוד גם להרץ אותו, למשל: *Clion, VsCode, VS*. *Compiler*: סט כלים שძקף בתוכו כמה שלבים שהשלבים האלו ייחד מעבירים ומכניס שכתבנו מוקבץ קוד לקובץ הרצה שנוכל להריץ על גבי המחשב. עובדים עם *GCC Project Builder* *Makefile, Cmake* כליל ש"שומר" על סדר בתהליך יצירה קובץ הרצה, הוא מודא לכל הקבצים בתוך הפרויקט מועברים כמו שצרכן במהלך יצירה קובץ הרצה. כמו *Computer Interfaces* *GUI ,GUI, Shell* למשל שימוש לפיתוח וסגירת חלונות.

:Shell

תוכנה שבמסגרתה אנחנו יכולים להכנס פקודות ומערכות הפעלה מריצה אותן בפועל לאחר שמקבלים פלט על הפקודה. פקודות מרכזיות:
1. *ls* – מראה לנו איזה קבצים/תיקיות קיימים בתחום התקינה שאנו נמצאים בה.
2. *pwd* – מראה את הנתיב בו אנחנו נמצאים כרגע. למשל *.guy/desktop*
3. *mkdir* – מאפשר ליצור תיקייה 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 העבודה של לינוקס. השימוש בו נעשה באמצעות המקלדת בלבד, ללא העכבר. כל פעולה שאפשר לעשות על קובץ, קיים עבורה מקלט כלשהו שמנוע את השימוש בעכבר: מדוע זה רלוונטי אלינו? אם מתחברים מרוחק לשרת לינוקס כלשהו, למשל לשרת של המחלקה: אין לנו אפשרות *GUI* של פיתחה וסגירת חלונות. אם נרצה לעורך קובץ לשרת, נוכל לפתחו אותו עם *Vim*.
שנה גרסה מתקדמת של Vim: *neoVim*, שנכיר.

:MakeFile

כל שmorכב מכמה וכמה חלקים וחוקים, שהמטרה של כל החוקים הללו יחד היא לוודא שכל הקבצים הRELוונטיים בפרויקט שלנו מעורבים ביצירת קובץ ההרצה. זהו קובץ שמאוד נפוץ ביצירת קבצי

C/C++

- מורכב נוסף של *rules* שכל אחד מרכיב שלושה אלמנטים:
- 1. שם *rule* (לרוב קובץ שנוצר כתוצאה מה執ת *rule*).
- 2. פקודה שתבצע בהמשך הרצה של *command*.
- 3. קבצים לצרכים להיות קיימים בשביל הרצה של *rule dependency*.

כשנרים את המילה "make" מה שקרה הוא שהראשון בקובץ *makefile* הוא זה שיורץ.

2 הרצתה 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 $U = x_0 x_1 \dots x_{n-1}$

מספרים שליליים: במקרה זה יתקיים $T(x) = -x_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$. שיטה זו נקראת המשלים ל 2 כיוון $2^n \equiv (-X) + X$. כיצד עושים כן? מכפילים את *MSB* בחזקה ומוסיפים סימן שלילי ומוחברים את השאר כמו במספרים חיוביים.

2.2 CPU Flags

באשר *CPU* מבצע חישוב של חיבור הוא לוקח ביטים של האופרנד הראשון, ביטים של האופרנד השני ומוחבר אותם מוביל לדעת האם החיבור הוא *signed* או *unsigned*. חיברנו שני מספרים, כמו בדוגמה כאן מטה: כיצד נדע אם התוצאה שקיבלו נcona או שגואה?

Example1:	$ \begin{array}{r} 01000000 \leftarrow 64 \\ + 01000000 \leftarrow 64 \\ \hline 10000000 \leftarrow 128 \text{ or } -128 ??? \end{array} $
---------------------------	---

במקרה של חיבור של *signed*, 128 – מה קיבלו כאן? *CPU* בעצמו לא יודע. בחישוב *unsigned* החיבור נcone – כי אכן $64 + 64 = 128$ שווה ל 128 . נשים לב כי *MSB* של שני המספרים היא 0 , ולכן שני המספרים הינם חיוביים. וכך נקבל 128 – התוצאה שגואה.

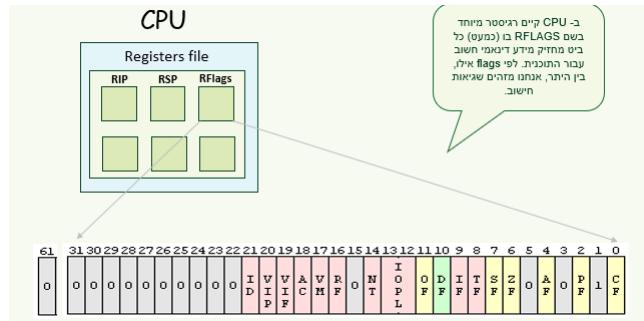
כלל: *CPU* ביצע חיבור של ביטים, **האם התוצאה היא signed או unsigned**. *CPU* ישמור במקומות מסוימים האם הייתה הخلافת סימן, אם לא מעניין אתכם (אתם מחשבים *unsigned*) פשוט אל تستכלו במידע.

דוגמא: בדוגמה כאן מטה נוצר *carry*, *overflow*, אך קיבלו אפס. אם היינו *signed* אכן קיבלו תוצאה נcona. אחרת, *unsigned* או תוצאה שגואה. היה *carry*. נשים לב כי אם היה חיבור של שני סימנים שונים – אפס ואחד, *cpu* לא צריך לבצע הخلافת סימן, הוא רק יזכיר האם היה *carry* או שלא היה.

Example2:	$ \begin{array}{r} 11111111 \leftarrow 255 \text{ or } -1 ? \\ + 00000001 \leftarrow 1 \\ \hline 100000000 \leftarrow 0 ??? \end{array} $
---------------------------	--

היכן הCPU רושם רישומים אלו?

רשותן רגיסטר מיוחד שCPU שומר בשם *Rflags* - יש לו ביטים שמתעדכנים בעט חישוב שהCPU עושה. הוא מעודכן את הביטים תוך כדי החישובים. אנחנו מתעניינים בביטים הכהובים. נשים לב שלכל בית ישנו שם כן. **ישנו בית בשם CP(CarryFlag)** קיבל אחד כאשר יהיה *carry* (בחיבור או בחיסור) אחרית, קיבל אפס. יאמר (אם ואחר) כי חישוב *unsigned* שני. **ישנו בית בשם OF(OverflowFlag)** הוא קיבל 1 כאשר תהיה החלה סימן (כלומר הביט שונה סימן), אחרית קיבל אפס. יאמר (אם ואחר) כי חישוב *signed* שני. **ישנו בית בשם ZF(ZeroFlag)** אם תוכאת החישוב האחרון יצאה אפס הוא קיבל אחד, אחרית הוא קיבל אפס. **ישנו בית בשם SF(SignFlag)** לוחק בית סימן של תוצאה ומעתיק לביט. כלומר לוקחים את *MSB* ומעתיקים אותו ל-*SF*.



Endianness 2.3

ארQUITטורה של המחשב יכולה להיות *Little Endian* או *Big – Endian*. אם יש לנו ערך נומי – מספרי, לא מחרוזת, והערך הנומרי הזה תופס יותר מbyte אחד (כלומר לא *char* בלבד) וכן לא *float*, **מדובר על ערכים שלמים בלבד**.
ב*Big Endian* שומרים אותו משמאלי לימין (שומרים את *MSB* הכי למטה – כלומר נשמר במקום (00) ב*Little Endian* . ב*Little Endian* שומרים אותו מימין לשמאל (כיצד נזכר? שומרים את *LSB* הכי למטה – כלומר שומרים במקום (00)).



אנחנו בקורס לומדים לפי *Little Endian* (כאשר אנחנו מדפיסים בקורס משהו, זה מתבצע בהדפסה מכתובה 00 כלומר מה שיודפס קודם יהיה *LSB*)

Assembly 2.4

אסמבייל נוצרה בשביב למסות להפסיק לכתוב בשפת מכונה, ולנסות לתרגם את השפה לשפת בני אדם. שכחטו את השפה לא חשבו על נוחות. על כל פקודה מכונה, לקחו את הפקודה ו"תרגמו" אותה לשפת אסמבייל שיכאהורה יותר אנושית. ומכאן המסקנה: **פעולה באסמבייל=פעולה של שפת מכונה.** ישנו כמה סטיילים של כתיבה סינטקטית באסמבייל, בקורס נלמד *AT&T* ישנו סטייל של *Intel*.

אסמבייל לא תומכת בדברים הבאים (ובמה משתמש במקום?):

א. *data types* (כן יש - גדים, 4 ביט, 1 ביט...)

ב. מערכים (נוכל לעבוד ישירות עם *RAM*)

ג. תנאים (גישה ישירה לרגיסטרים - זהה השפה היחידה שמאפשרת זאת, ולכן **אסמבייל היא השפה הפייעלה**).

ד. לולאות

ה. פונקציות

ו. ספריות סטנדרטיות

מה שנקרא - בהצלחה תהיה לנו.

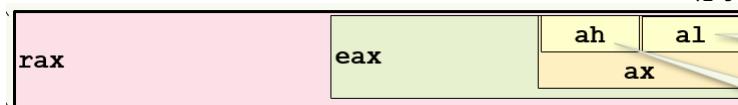
Registers 2.5

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

רגיסטרים לחישוב כללי :*general purpose registers*

הרגיסטרים הניס *RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, R8, R9, ..., R15*. נכל להשתמש בהם לחישובים כלליים. הסיבה לכך *R8, R9, ...R15* אין שמות הוא שם נוסף בהרבה של אסמבייל, שכבר הבינו שלשות אין משמעות.

בעבר היה חסר רגיסטרים ולכן בתוך רגיסטר היו מאחסנים שני נתונים. למשל ברגיסטר *ax* היה שני תכני מידע שונים - *ah, al* ובהתחמלה *High, Low*. בהמשך, הרחיבו את הרגיסטר ל-32 ביט ולא נתנו שם להמשך המידע וגם בהרבה ל-64 ביט לא הוסיפו שם. כך נראה רגיסטר - אוסף בaczron של 64 ביטים.



שניהם המון רגיסטרים - בקורס אנחנו משתמשים ברגיסטרים שצויינו לעיל, ברגיסטר *RIP* וברגיסטרים *Rflags* של *RIP*. נזכירם:

RIP - רגיסטר *Instruction Pointer*. הוא הרגיסטר שה-*CPU* לא יכול בלעדייו. זהו רגיסטר שמצויב על הבית הראשון של הפקודה הבאה לביצוע. *Loader* שם בתוך הרגיסטר מ mish את כתובתה זו במהלך תהליך הפרסור. (עזרה - גם באסמבייל נוכל לגשת לרוגיסטרים בaczron, מה שאי אפשרות בשפות עילית).

RSP - רגיסטר שמנצט בזdeck ה-*CPU*. מצובע על תחילתו (כיוון שאם הערך גדול מדי, לפי מבואו הוא נשמר מלמעלה למטרת בaczron) של הערך האחרון שנכנס למחסנית. המחסנית חשובה מאוד כיון שיש בה משתנים לוקלים, *activitaion frame* וכותבתഴה וכן ארגומנטרים שפונקציה מקבלת (אם וכארש).

Basic Instructions & Data types 2.6

שורת קוד **קובד טיפוסית באסמלבי נראית כז:** *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 משותנה.

נשים לב: \$ חשוב מאוד, ואם לא נכתב אותו יחשבו שאנו שוארים על מוקם בזיכרון. אם כתוב עם \$ יהיה ברור כי הכוונה לערך. כשרצחא לפנות לרגיסטר - נעשה זאת עם % בפניהם לפני השם של הרגיסטר.

- הפעולה mov:** פקודת זהה. מושגים לסוף הפקודה סימות, בהתאם לגודל הטיפוס שאנו חישב. אנתנו למשה ממצאים העתקה של ערך למקום אחר בזיכרון.
1. הפקודה *movb*: פעולה הזה שעובדת על bytes, מזינים בית ממוקם למקום. בית הוא 8 ביטים.
 2. הפקודה *movw*: פעולה הזה שעובדת על words, מזינים ממוקם למקום בזיכרון.
 3. הפקודה *movl*: פעולה הזה שעובדת על long, longs, מזינים long ממוקם למקום. long הוא 4 בתים. (כמו int ב-C)
 4. הפקודה *movq*: פעולה הזה שעובדת על qword, qwords הוא 8 בתים. (כמו Long ב-C)

2.6.1 השוואה לשפת מוכנה

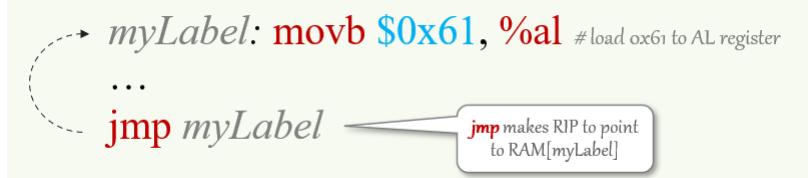
אותה דוגמה מלמעלה, בשפת מוכנה תתרגם להיות השורה הבאה:

$$0xB061$$

זה הערך המספרי אותו צריך להכניס, 0x B061 מעיד שאנו בהפסקה dzimal, ו- B0 מסמן לו שהוא צריך לבצע הכנסה של ערך לתוך רגיסטר *al*. (מיאיפה אנחנו יודעים זאת? נפתח ISA.). *Imm8* הוא ערך נומרי בגודל 8bits. *Label* הוא ערך נומרי.

jmp ו-labels 2.6.2

בכדי לשים הערות על הקוד באסמלבי - נוכל להשתמש בנקודה פסיק או#. נוכל להוסיף לכל פקודה *Label* זה מקום של הפקודה. כאשר נוסיף את הליביל, הקומפיילר יתרגם אותו כתובות של הפקודה. **מאחרוי הקלעים:** הקומפיילר מוחק את הליביל ומוכניס את הכתובת. בשביל מה נctrיך ליבילים? כתע, נוכל לבצע jmp ולהגיע אל הפקודה הזאת, ממוקם אחר בקוד. RIP[Label] jmp RAM[Label]



Assembler directive 2.6.3

הנחייה שאנו ננתנים לקומpileר של אסמבלי, שמו *.Assembler* נתבונן על הפקודה הבאה -

buffer: .skip 4 # reserve 4 bytes

קומpileר, תlk לזכור, ספציפית *section* של משתנים גלובליים, בפרט אל *bss* (המשתנה לא מאוחחל), ותקצתה לי שם רצף של 4 בתים. מעכשיו, יוכל להתייחס אל 4 הבטים הללו *buffer.chars*. זה כמובן מקביל ליצירת משתנה ללא אתחול, וכן מקביל למערך של *shorts*. וכן מקביל ליצירת מערך של שני *shorts*, או מערך של *char* ושיי - וכך כל קבוצה של משתנים שנסמכתה לנודל של 4 בתים. בעת, כאשר נרצה להכניס את הערך 2 אל המשתנה *buffer* נעשה זאת באמצעות הפקודה הבאה:

movl \$2, buffer

Addressing Modes 2.6.4

כעת, נראה כיצד מתרגם קוד בשפת *C* לאסמבלי: בתחילת אנו מקצים בזיכרון 20 בתים (5 פעמים 4 בתים של *int*), אנחנו מאתחלים משתנה ברגיסטר *rbx* שיקבל את הערך אפס שיצין למימוש את *i*. משם אנחנו מתחילה לולאה, *i = 0*.

באסמבלי יש פקודת השוואת גנרייה: *cmpq*, באסמבלי יש פקודת השוואת אחת (*lal* כמו *<>*), מה שהפעולה עשויה הוא לחשב את החיסור של *source* מה*destination*, כלומר הפקודה הבאה תתרגם להיות

cmpq \$5,%rbx

כלומר נבצע *5 - rdx*. נשים לב שתוצאת החיסור לא נשמרת. אם נקבל כי התוצאה חיובית, המשמעות היא כי *sorce > dest*. אם שלילי, *dest > sorce*. כיצד נדע האם יש קשר של $\leq, \geq?$

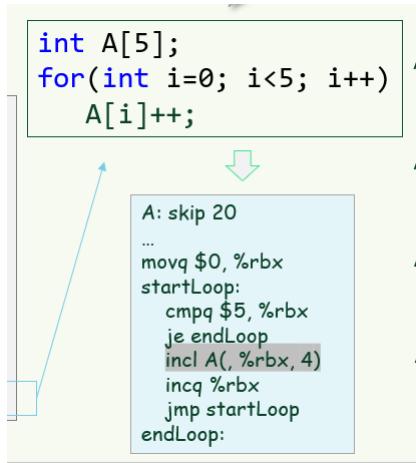
ZF

כלומר - גם שווה, נסתכל על *Label*: אם יש שווון כלומר *ZP = 1*, בין שני האופרנדים שבוצעו בשורת הקוד הקודמת, אנחנו נבצע *jmp* אל *Label* לאחר *je*. הפקודה הנ"ל ניגשת אל *zeroFlag* ובודקת, אם הוא 1 היא קופצת אל *Label*.

הפקודה inc: הפקודה למבצעת מבצעת *inc, increasement*, יש שמבצעת זאת על *long*. וכמוון כמו ב*mov*, בהתאם לכל סוג משתנה. הסינטקס שלה יהיה כלהלן:

inc A(%rbx,4)

כאשר אנחנו כותבים סוגרים באסמבלי - אנחנו ניגשים אל הזכרנו. אנחנו הולכים אל *RAM* מקום שהוא $4 * rdx + 2048$ או הכתובת של *A*. אנחנו למבצעת ניגשים אל הכתובת של *A[0]* ואנו מגדילים אותה, בשביל להתקדם לכטובת הבאה (ש哿 שמרנו כשהקצנו 02 בתים). כאשר נבצע *inc* על ערך, נוסיף %. 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 מחסרת 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 (Conditional jumps) וכוכ' 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 גניסטר שווה לאפס:

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

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

: פקודות הזהה של ביטים. בביטוי $SHL, SHR, SHIFT$ שיצאים מהמקום נזקים ונשמרים ב- CF , מימין ננסים אפסים. בביטוי SHR כל בית זו ימינה, ביטים שיצאים מהמקום נזקים ונשמרים ב- CF , משמאל ננסים 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 במס' חיובי וקיבלו מס' שלילי. בדיקוזו הסיבה שנשתמש בשיפט אריתמטי - זהה לחולוטן SHL . עם זאת, SAR מזין לימיינ את הביטים, אך הוא משאיר את בית הסימן בצד שמאל. כלומר: הוא לא מזין את בית הסימן. **דוגמה** - 1101 אםמבצע SHR על 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 הרגיסטר:

```
xorq %rbx,%rbx
```

נוח לאפס את הרגיסטרים בתחילת הריצעה.

רגיסטר הוא חומרה שצמוד במעבד - ולכן הגישה אליו הרבה הרבה יותר מהירה. **X86 – 64** מכיל 16 רגיסטרים, כל אחד בגודל 64 ביט. **מטרת הרגיסטר RAX** הוא להחזיר ערך חזרה מפונקציות. **RBP** הוא לשימוש המחסנית, **RSP** הוא מצביע על ראש המחסנית. לא להשתמש ברגיסטרים אלו שלא למטרת המחסנית.

סינטקטית, באשר $i \in \{1, 2, 4, 8\}$ הчисוב כדלקמן

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

```

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)

```

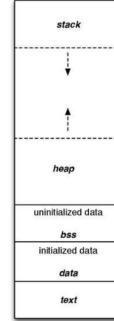
אנחנו קובעים את המשמעות של הרגיסטר - האם אני משתמש בו כתובות או כערוך. אני מחליט מה רגיסטר מחזיק.

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 שכתבנו 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 נוכחות 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, כתובת 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*

sete	ZF	Equal / Zero
setne	\sim ZF	Not Equal / Not Zero
sets	SF	Negative
setsns	\sim SF	Non-negative
setb	CF	Below (unsigned)
setae	\sim CF	Above or equal (unsigned)
seta	\sim CF & \sim ZF	Above (unsigned)
seto	OF	Overflow (signed)
setno	\sim OF	Not Overflow (signed)
setg	\sim (SF \wedge OF) & \sim ZF	Greater (signed)
setge	\sim (SF \wedge OF)	Greater or Equal (signed)
setl	(SF \wedge OF)	Less (signed)
setle	(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* בוגינוד **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* ומכוונים את הכתובת לרегистר השני.

במקומות הפעולה יכולים להשתמש *mov* ו-*add*. אז למה צריך להשתמש בפקודה? נסתכל על הדוגמה מטה. כנראה להשתמש בפקודה *lea* זה בשביל לבצע 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 $3\%rdi$. ומכוונים 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 בבש *addressing mode* מאפשר $t \leftarrow x + x * i$. וכך יתאפשר לבצע $t \leftarrow x + x * 12$.

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 שקרהנו לה, הוא באמצעות הפקודת *ret*. הפקודת מחזירה אותו להיקן שקרהנו 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' ביטים 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.

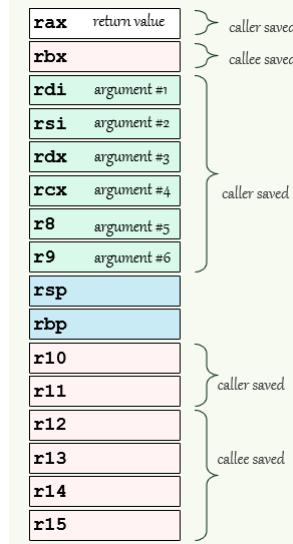
הערך המוחזר תמיד דרך *rax*. מכאן נשאלת השאלה - ומה עם גדלים שגדולים מ-8 ביטיים? מאחורי הקלעים הקומpileר מתרגם את הגודל הזה כאיילו הוא מוחזר פוינטר (אכן פוינטר בגודל 8 בתים) אל הערך זהה.

16 הרגיסטרים מתחלקיים לשני קבוצות

נתבונן בבעיה הבאה. נניח ונחנו משתמשים ברגיסטר *a* כלשהו ואז קוראים לפונקציה, ואותה פונקציה משתמשת באותו רגיסטר. כשנזהור 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` ונדירוו 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 לדוחה 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

```

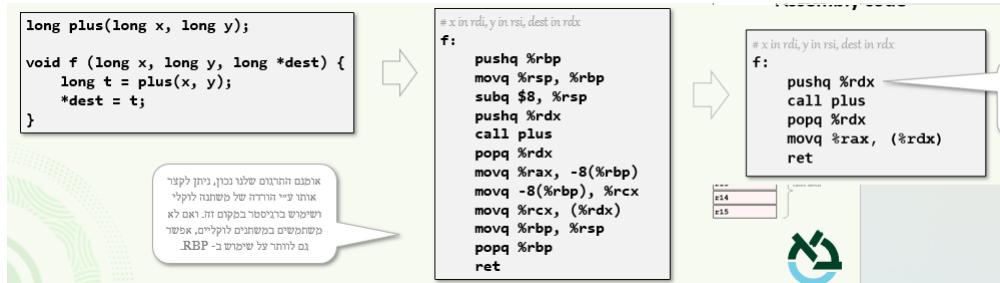
87

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`.

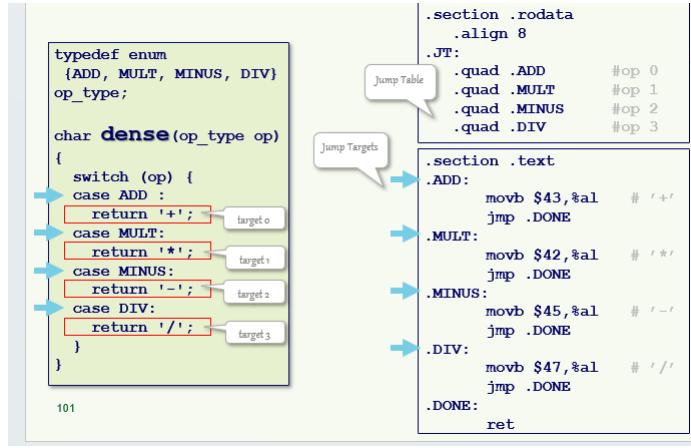
מה נעשה עם פונקציה שמקבלת יותר מושיעת ארגומנטים? נכניס את השיטה הראשונית לרегистרים המותאימים ואז את השאר נדחוף למוחשנית בסדר הפוך. ככלומר אם יש 10 ארגומנטים נכניס את השיטה לרегистרים ואז נדחוף את 10 למוחשנית, 9 8 7 וזהו. הרעיון הוא שהכי יהיה לי קל לגשת אל המשתנה השביעי.

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.

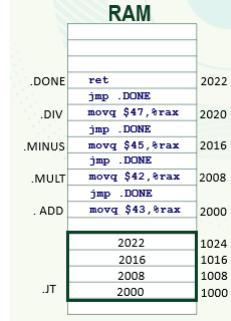
Jump Table 3.6

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)

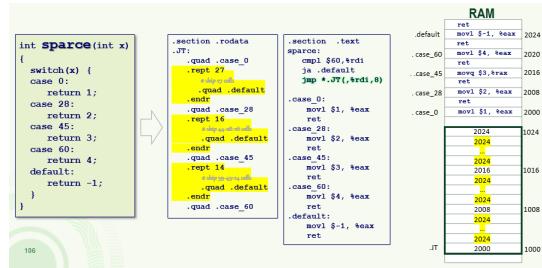
```

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

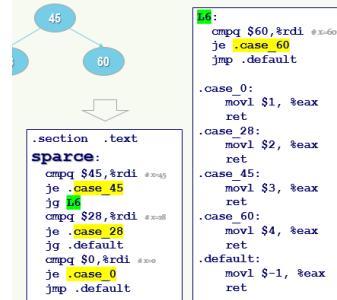
$$2 \times 8 + 1000 = 1016$$

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 במרקחה 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 $O(\log n)$ שלנו?

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 flag, לא נקבל אפשרות כלשהו. שימוש זהה מעת את קובץ ההריצאה
אך מוסיף מידע שהוא יוטר יותר.
```

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 נכתב 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"מיוקום" היחסית של כל פקודה ביחס לsection שהיא מוגדרת 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 *data* קומפיילציה, 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*. הקומפיילר יודע שכל עוד לא פתחנו section חדש אז אנחנו ב*.rodata*. בעת הקומפול, הקומפיילר מנהל הרבה מאוד טבלאות. אנחנו נתיחס לשתי טבלאות:

Symbol Table: הקומפיילר 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, הקומפיילר נתקל בשם חדש: *str*. היא מכнесת גם כן בשורה חדשה בתוךAPS. אנחנו רשותם בטבלה באיזה section הינו באשר ראיינו את המשנהה *str*. כמו כן, אנחנו נגדיר לו את *location* שהוא מקום היחס שלו בתוך *section*. כמו כן, הקומפיילר צריך לעדכן בעת כניסה *str* את הגודל של סשן *rodata*, כי התווסף גודל חדש לסשן. ולכן: הקומפיילר מעדכן בטבלה את *info* שלו להיות יותר מקום.

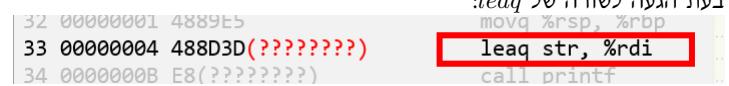
כך נראה הטעינה -

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.

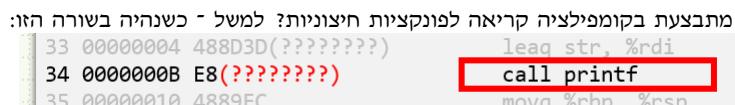
בעת הגעה לשורה של `leaq`

 32 00000001 4889E5
 33 00000004 488D3D(????????)
 34 0000000B E8(????????)
 leaq str, %rdi
 call printf

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, `str` נמצא `text` שמצוות מעלה `rodata`, וכך ידוע גודל `text` זה משפייע על `rodata`. אז מהו הקומPILEר עושים?

הקומPILEר בונה טבלה נספת - Relocation Table: הוא מעדכן שם, שבתוך סקון טקסט, במיקום 0x07 (הבטיח השביעי, בדיק אם השטחים למטה 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 בkompileita קריאה לפונקציות חיצונית? למשל - כשהיא בשורה זו:

 33 00000004 488D3D(????????)
 34 0000000B E8(????????)
 35 00000010 4889EC
 leaq str, %rdi
 call printf
 movq %rsp, %rbp

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 הקמפול, בפרט השתיים שתיארנו, והוא מוכן לעبور לשלב הבא.

ELF relocatable format 4.2

כעת נדון כיצד הקובץ המקומפל נראה באמת - לא איך שאנו קים פלנו 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

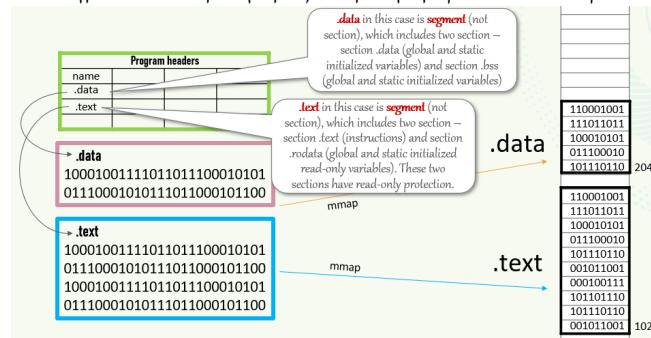
readelf utility allows to observe structure of ELF file (object or executable)

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 הוא עובד. הקומפיילר מעדכן כי הקובץ הוא בשיטת המשלים ל-2 (מס' שליליים בנוסף) וכן עם *Little Endian*. כמו כן מעדכנים שמדובר בקובץ, *Relocatable*, שודנדש לקשר עם קודים שונים (כל *extern*).

לבסוף, מופיעים כל *sections*. נראה כי למאות שקד המקורית ישנו שני סקשנים, הקומפיילר טוען שיש 7. תחילתו הוא טוען שיש *text*: אותו אחד שאנו צרכנו קודם קודם, אכן בגודל של 17. כמו כן, אנו אומרים לו שה*offset* שלו 40. בקובץ המופיע הוא מתחילה בשורה 40. לאחר מכן אנו מולמים שאת הטבלאות שקדם ראיינו, הקומפיילר שומר בתוך *section symbol* זה *relocationTable* *syntab* ו- *rel.text* *relocationTable* *text*. והוא שומר עוד כמה סקשנים - שאינם בחומר הקורס.

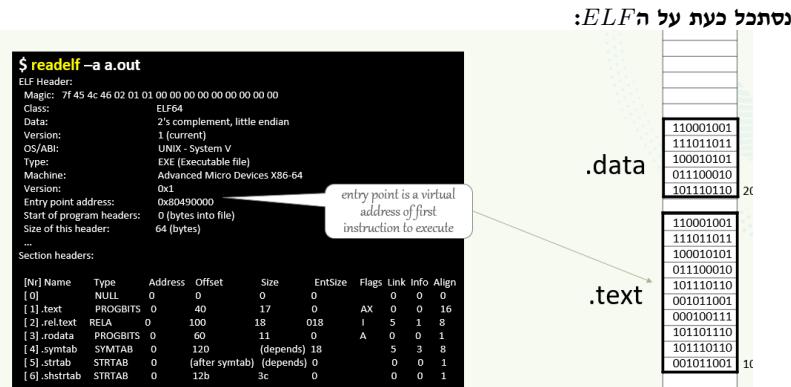
ELF executable format 4.3

נרצה להבין מה הבדל בין קובץ מקומפל, לקובץ מקומפל ומולנגן:



כאן יש לנו *segment* שמכיל בתוכו שני סקשנים בדיק, שנקרו להם כתעט *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* ולכן הוא מנה 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 *str(%rdi)* ולא *str(%rdi)*).

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 תוכנות 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 *Loader* 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 מנגנון טקסט לבנות סגמנט *Loader* בגודל *4bytes* הוא בונה סגמנט בגודל *4k* (!!!). כאשר אנחנו טוענים מקום לסקשן טקסט אנחנו טוענים את זה פר בЛОקים. 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 *process* שאמורת: *ani* שישים לבצע כל מה שהוגדר, ובבקשה מערכת הפעלה תמחקי את *process* כתעט. ולכן, בודאות, הקוד הנוסף לא יתבצע.

אי מה נעשה? במקומות להוסיף את הקוד בנוסף אל *segment text* אנחנו נמחק חלק מהקוד שם, ונכנסים במקומות את הקוד שלנו. כתעט - יש סיכוי שהקוד יתבצע. מודיעו יש סיכוי ולא בודאות? כי יכול שמחיקה הרסה דברים פשוטים וכעת הקוד לא יתאפשר או תהיה שגיאת זמן ריצה. אז אנחנו לא רוצחים אליו - אנחנו רוצחים בודאות ליצור וירוס. כיצד? נדרושים את השורות הראשונות של פונקציית *main*. נכנס אל השורות הראשונות של *main* את הקוד שלו, 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 שלו 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 מקום שהכנסנו את הקוד. ואז, כשה我们会 לינוק את *Loader* הואילך אל הקוד המורושע שתכתבנו, וסימנו.

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 לנו לחפש באגים למשל בצורה הרבה יותר 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 מקומפלים, ונרצה לבצע *linking* של כל הקבצים לקובץ אחד. נשים לב כי כל שגיאה שתתבצע בשלב זה תקרא כתעט **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 לדרוך *strong symbol* אם נקליג' את שני הקבצים.

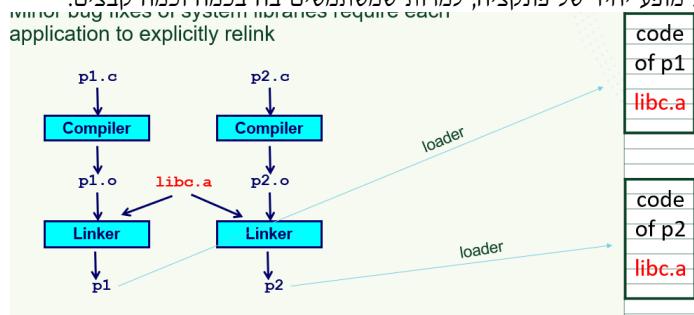
3. אם יש לנו שני *weak symbol*, הלינקר יכול לבחור אחד מהם שרירותי.

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* ולא *Loader*, היקן תמצא ספריה דינמית שכזו. קוד מסווג זה, יקרא *Position Independent Code*.

Position Independent Code 4.6

קוד שלא תלוי במיקום. מהו קוד שתלוי במיקום? ישנים מיקומים של *sections* שנקבעים בזמן הקומPILEציה / 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*. ישנו מתייחס לביטוי *str(%rip)*. והוא מוחשב את המיקום היחסי של *str* ביחס למיקום הנוכחי של *rip* חישור כתובות *str - rip*. וכך הוא מוחשב 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*. הקומפайлר (דges - אנחנו לא) מוחשב את ההיסטוריה, מסמן $x = str - rip$ ואז מותרגם זאת ל-*str + %rip*. בדיק לפי $x(\%rip) = x + \%rip$

<pre>.section .data str: .string "Hi" extern printf .section .text .globl func func: movq \$str, %rdi call printf ret</pre>	PIC <pre>.section .data str: .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 בינה שבין 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. ישנים וירוסים שונים משתמשים על *process image* בזורה אקרים - וכך לווירוס יהיה הרבה יותר קשה למצוא את *section's*' השינויים בקוד.
2. באשר נעשה *dynamic linking* לא נדע היכן נמצאות הספריות הדינמיות שלנו, ולכן מיקום שכנם 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 בערכיהם 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 נמש 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

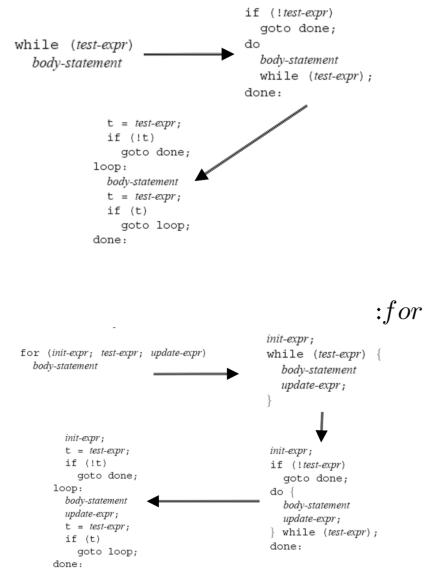
כל מבנה של תנאי, לולה וצדומה ניתן להמיר מ-C לאסמבלי. נראה מספר דוגמאות.
:if&else

<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



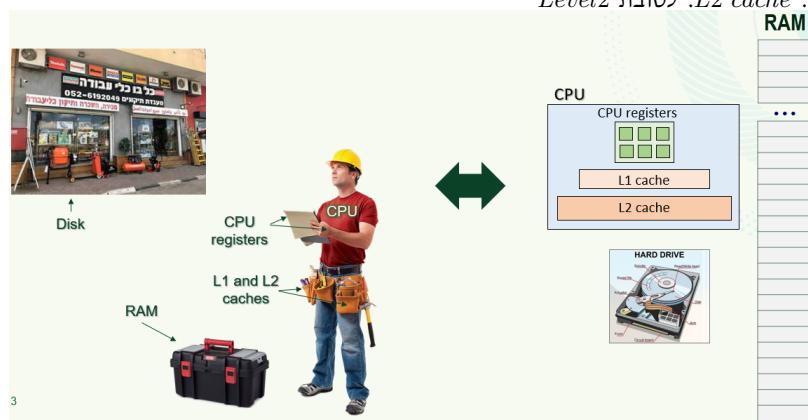
הרצאה 5 - memory hierarchy - 5

5.1 הקדמה

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

Level1 : L1 cache .1

Level2 : L2 cache .2



נניח שיש לנו אדם בתפקיד *CPU*. הידיים שלו הם ורגיסטרים (אחרת, איך יעבד?), הדיסק זה החנות - זה לוקח זמן לגשת אליה, וлокח זמן לגשת אל הדיסק. זמן גישה לדיסק עלותה פי מיליון בערך לעומת גישה לריגיסטרים. זו הסיבה - *shw* לא מסכים *loader* לעבוד עם דיסק, ומכרייה אותו לבנות *RAM process image* - הוא בדיקת *cachen*. פחות אטרקטיבי *RAM* מרגיסטרים, אך עדין קרוב אליו. ארוצו הכלים יהיו

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 מיקום, וחלק מהמיוקם 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 נשתמש 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 שהזמן הקרוב אליו אני משתמש בו עוד פעם. למשל אם נחשב סכום של מערך העקרון לא יפעל על *a[0], a[1], ..., a[n]* אך כן יפעל 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 מהשנה באיזו שורה נבייא את *x* אליה? הרי אנחנו לא רוצים לעבור בכל השורות. החיפוש ב-*cache* הועיל.). אחרת, נחפש ב-*L2*, אם נמצא נבייא את *x* לרגיסטר ב-*CPU* וכן אנחנו נקדם את *x* אל *L1* (מדוע?). בסביבות גבולה כמו שאמרנו השתמש 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 בטווח הרחוק זה אכן משתלב.

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 יודעים את מי אנחנו מעבירים במקומו ל-L2? אינטואטיבית - את מי שהשתמשנו בו הכי רחוק, וכך הסבירות שורצאה 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 לבעיה קריטית ויעשה את ההבדל בין תוכניתנו נכון לשגוי. נדע זאת כרגע - ונאפשר את זה. בקורסים עתידיים - נדע כיצד פתרים זאת.

מטריצות: ישן שתי 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:

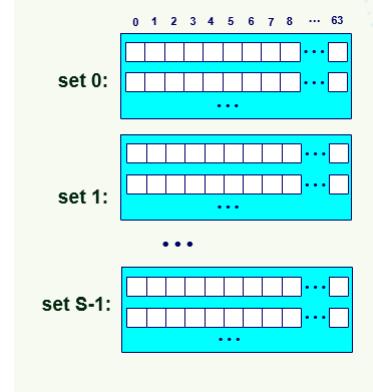
אם מביאים קוד שהוא נכנס אל L1 instruction cache שהוא בטוחה של 16 – 128(KB)

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

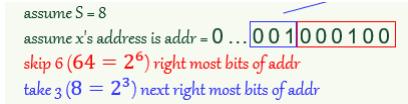
128(KB) – 8(MB) הוא בגודל (L2 cache

organization of a cache Memory 5.3

מבחן גדים cacheן מוחלק ל-'sets'. 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.



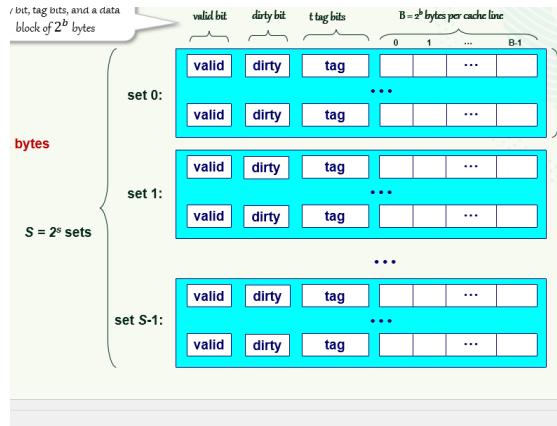
נניח שהוכנסנו את x , כיצד ה-*cpu* ידע באיזה set נמצא x ומה המיקום היחסית שלו באותה מחרשה? אם מס' מתחולק ב-6 לא שארית בהכרח 6 ביטים אחרונים שלו הם אפסים. אנחנו נסתכל על כתובות שמתחלקות ב-64bits וחייב קורובה לאיקס הנוכחי, כי שאמרנו קודם ונkeh 64 ביטים. נkeh את 6 הספרות האחרונות של כל כתובות והם ייצגו את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

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 CPUP. רוחה לדעתה $cacheLine.tag$ מכיל באמצעות את x ולכן הוא משתמש ב-*tag*, שכן יתכו כתובות שונות עם אותה סיומת של 9 ביטים. כשה-*cpu* מוחפש את x (cacheLine.tag), 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.

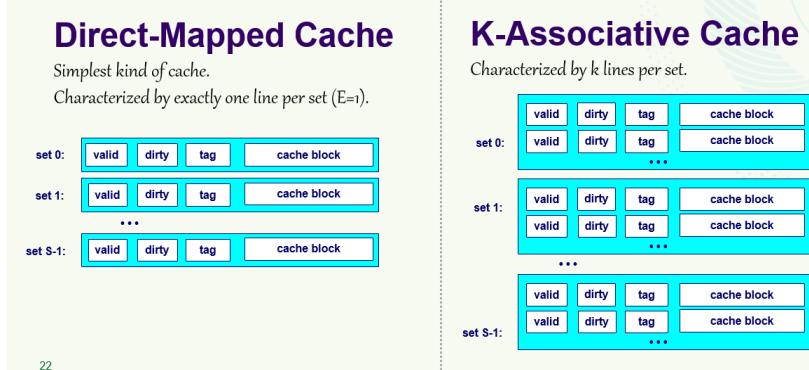
dirty bit: האם *cacheLine* הוא *dirty* או לא. מודיע זה חשוב לדעת? *Ram*. מודיע "פטור" dirty bits אנחנו חיבים לגשת אל *Ram* ולעדכן את הערך של הביט הראשון נאבד אותו (הוא השתנה ונשענו לא יודעים זאת בזיכרון). ואנו מוחקים מהקיים cache. סכנה לאבד אותו לנוכח.

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



מצב שיש לנו בדיקת *cacheLine* אחד בתוך כל *set*: *Direct Mapped Cache* גם יודעים איך לחפש את *CacheLine* ב*set* כי יש בדיקת אחד בפנים. החסרון המרכזי שאם נביא נושא אל אותה *set* נהייה *cacheLine* הנדרש לנו. אין מקום אחר בתוך אותה שנווכל להעבריר אותה אליו. היתרון המרכזי הוא שזמן החיפוש כאן הוא בדיקת $O(1)$ - נסתכל על tag בדיקת *cacheLine* x שם או לא.

.set cacheLines k : K – associative – Cache



למה שלא נkeh set ייחיד? יקח הרבה זמן לחפש בתוך *set*. העובדה שיש לנו הרבה יותר לדעת לאבחן לאיזה *set* שייך איבר, וכך לדעת לצמצם את טווח החיפוש. מה שנעשה לרוב: יהיה לא להיות ממעבב של סט ייחיד, ולא במצב של k אלא יותר מזו.

cache סיכום 5.4

נסמן:

- מס' הסיטים בכל קאש. S
- מס' הביטים הנדרשים לייצוג מס' סט. $s = S$. מתקיים: $2^s = S$
- מס' השורות בכל *set*. E
- גודל כל שורה (כל בלוק). B
- כמות הביטים הנדרשת לייצוג offset בתוך כל בלוק. מתקיים $b = B$

כל בлок בהכרח מתחפה לסט אחד בלבד. אין הגבלה על מי השורה שתכילה בлок מסוים בתוך הסט אליו הוא ממופה.

.*Directed – Mapped* $E = 1$ זה נקרא

.*Associateive* $S = 1$ זה נקרא

.*k – associative* אם $E, S > 1$ אז נקרא

ישנן שתי שיטות כיצד *cache* בוחר איזו שורה לפנות (מפנים שורה כאשר הקаш מלא, וכשצריך להביא בлок חדש מהארכו):

least recently used :*LRU* - מפנים את השורה שלא נגעו בה הכי הרבה זמן, ווקבים אחר זמן הגישה האחרון לכל שורה ואת היכי רחוק מוצאים.
least frequently used :*LFU* - מפנים את השורה שנגישו אליה הכי פעוט פעמים. ווקבים אחר מונה גישות ומוצאים את זה עם הערך המינימלי.

5.5 תרגול 5

5.5.1 static linking

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

מהם השלבים שקו שוכתווב ב-*C* עבור? מתחלים ב-*main*.

א. השלב הראשון הוא *reprocessing* שבמסגרתו כל *define* ו-*include* שיש לנו בקובץ *main.i* מוחלפים בערכם המקורי. מתהליך זה יצא קובץ עם סימט *i*.
ב. מקבל קובץ *C* ומוציא קובץ באסambilי *main.s*: *compiling*
ג. *assembling*: ממקבץ *main.s* לוקח את קובץ האסambilי וממיר את הכתובות לשפת מכונה.
במהלך תהליך זה מוספות טבלאות כמו *شارיאנו* בהרצאה. בסיסים שלב זה יש קובץ *main.o*
ד. *linking*: המטרה של תהליך *linking* שבסיוםו מתקבל לנו קובץ הרצה היא *שהינתן*
main.o וקבצי ספריה נוספים, נרצה ליצור צימוד בין הפונקציות או המשתנים הגלובליים אליהם
אנחנו ניגשים לבין 의미ו שליהם פיזית (שמופיע בספריה). *refrences Resolving*.

library: אוסף של קבצי *o*. ישנו שני סוגי של ספריות - ההבדל בניהם הוא בסוג *linking* שיתבצע.
static: ספרייה *static* היא ספריה שמופעל אליה *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* זה.

חסודות של static linking:

1. נניח והשתמשנו ב-*printf* בקובץ הרצה שלנו, וכך בקובץ הרצה שלנו הוכנס אליו הקוד של הפונקציה. אממה, אם גילו באג בפונקציה של *c* זה לא ישתנה אצל. כלומר בשבייל שהקוד של הפונקציה יהיהichi עדכני, נטרך ליצור קובץ הרצה חדש.
2. חסרון שני - שימוש מיותר בזכרו. אם למשל אנחנו כותבים 10 תוכניות שונות שימושות ב-*printf* התקבלו לנו 10 קבצי הרצה שונים שככל אחד מהם יש *printf* וכשרנץ את קובץ הרצה יהיה בזכרו 10 מופעים של הפונקציה, חביל על המוקם.

dynamic linking: ספרייה *dynamic* היא ספריה שמופעל אליה dynamic linking . לכל פונקציה או ספרייה בעולם קיים עותק יחיד בזיכרון, נרצה פשוט לgesture אליו באשר אנו מניחים לכל פונקציה יש צ'אנק כלשהו בזכרון. יש בעיות בכך - אף אחד לא אמר שבעל רגע נתן אנחנו משתמשים בכל הפונקציות שאי פעם נ כתבו. חדרון נוסך נובע מכך בו אולי יורדות גרסאות נוספות לפונקציות, נרצה את היכי מעודכנת. נראה כי יש מצב שהגרסה העדכנית של *printf* בגודל יותר גדול - יותר ביטים, אבל מתחת צ'אנק של *printf*

הנוכחי יש קוד וגם ב'אנק מעליו יש קוד שכן נאלץ לחפש מקום חדש בזיכרון לכל הפונקציה החדשה. זה לוקח זמן!

לכן נשתמש בdynamic linking שפועל כך: *static* אמרנו שהציגו מתרחש באמצעות דחיפה של הקוד לקובץ הרצחה, ב*dynamic* מתרחש בזיכרון עצמו. איך? באחת משתי הדרכים הבאות:

א. *load time*: פירשו, שבזמן טעינת התוכנית שלו לזכרון, מיד הקוד שלו יטען יחד איתו נקרא *dynamic linker* שקורא את כל הספריות החיצוניתות בהםים הקוד שלו מתרחש, הם נטענות לזכרון. בדיק יחד עם התוכנית שלו.

ב. *run time*: כאן, אנחנו לא נטען ישר את כל הספריות לזכרון, אנחנו נהכה רק לפעם הראשונה שנדרה *symbol* בעת הרצת התוכנית, וזה *linker* יטען את *symbol* הנוכחי בזיכרון.

בעת **נשאלה הבאה**: האם עדיף *load time* או *run time*? אם למשל הקוד שלו מושתמש בהמון פונקציות, אך בغالל תנאי *if* מזמן רק 3 פונקציות. במקרה זה עדיף לנו כמובן *runTime*. לכן: ב-99% מהמקרים עדיף *runTime*, אך צריך להפעיל שיקול דעת.

נראה כי המטרה של *dynamic linking* היא שככל שלב בזיכרון יהיה עותק אחד של כל *symbol*. הוראה כ"י המטרה של *dynamic linker* חכם ובעת שираה למשל פעם שני *printf* הוא לא יטען זאת לזכרון אלא יגע לעותק הקודם שהוא טען.

חשיבות מודולר: נראה כי אם יש משתנה גלובלי של 10 תוכניות משתמשות בו – במקרה זה יטעןו לזכרון 10 עותקים שונים (!) גם בדינמיין לינקר, בשונה מפונקציות שנטענות רק פעם אחת.

לאן *dynamic linker* טוען את הכתובות? לאור בזיכרון שבין *heap* לבין *stack*. נשים לב כי יתכן וכתבתטי קוד שמשתמש ב-*scanf* שעוזר לא היזהה טעונה בזיכרון, לכן הרצתי והפונקציה נתענה לזכרון. נשים לב כי יתרן שירץ בפעם אחרת את התוכנית, היא תעטען למקום אחר בזיכרון. אף אחד לא אמר שהמקומות שהיא נתענת עליה ישר קבוץ.

ב-*flag*'ו שאנו מעבירים ניתן לקבוע איזה סוג *linking* נקבע וכן האם *loadTime* או *runTime* והוא *Lazy bidding*. דיפולט זה נקרא *dynamic default*.

(Position Independent Code) PIC 5.5.2

כפי שראינו, הרצת השפונקציות קודם יוכל להטען לכל מקום שהוא בזיכרון. נססה להבין איזה טריקים ושיקים הקומpileר מבצע כשהוא רואה קובץ *C* עד שהוא הופך אותו לקוד PIC באסמבלי:

מחלק טריקים אלו לשניים.
א. קוד **שנגייש *symbol* פנימיים בלבד** (רק לפונקציות או משתנים גלובליים או סטטיים שהוגדרו **באתו קובץ בלבד**):

ב. קוד **שנגייש *symbol* חיצוניים** (יתכן גם פנימיים): כבר בזמן קומPILEציה אנו יודעים שיש לנו *symbol* *symbol* – *relative addressing* של *symbol* לא באמצעות הכתובות האבסולוטית שלן בזיכרון אלא המרחק שלן *rip* כרגע. זה נקרא באסמבלי בצורה זו *str(%rip), %rax*. כיצד הリンקר יודע לחשב את המיקום של כל כתובות? נשים לב כי בין *data* ל-*text* ישנו מרוחוק בזיכרון. לכן נניח וכיינה *globalY* שמנצאת כתובות *text* עד תחילת כתובות *data*. אנו יודעים את המיקום היחסי מתחילת *text* עד *globalY* וידועים את המרחק מתחילת *text* עד תחילת כתובות *data*. אנו יודעים את מיקומו היחסי של *text* globalY. נקח ערך זה, ואת הערך של המרחק מתחילת *text* עד ונחסר את המיקום של *text*. זה הפרש בין המיקומים וכן נדע לעבור בניהם.

שנה טבלה בשם *GOT*: *Global Offset Table*. טבלה שמתווספת בכל *data segments*, בכל שורה ישנים 8 בתים. בעת שהリンקר טען *symbol* יקח את הכתובת של *symbol* שהוא טען ושיטים בשורה בטבלה. כיצד זה עוזר לנו? אנו מנסים לגשת ל-*symbol* חיצוניים אך לא יודעים היכן הם נמצאים. כתעת באמצעות הטבלה נוכל ממש בקוד שלו לגשת אל *symbol*. *GOT.symbol*. כיצד נוכל בזיכרון *GOT*? כיצד נוכל לגשת אל *symbol*? כמו קודם – נוכל לחשב את מיקומו. נשים לב כי לפחות פעם אחת בזמן PIC

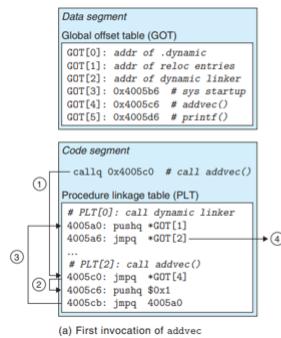
קומpileציה נוצרים ב-GOT סימboleים פנימיים גם - וכן אפשר לשאת אליהם באמצעות צורה בדיק (אם כי אכן אין צורך לרשימת GOT עבור סימboleים פנימיים, אך זה קורה).

PLT 5.5.3

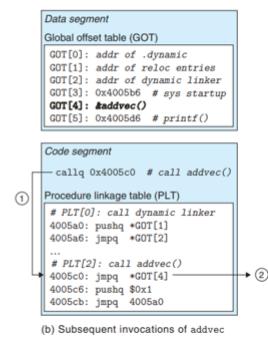
משמשת אותנו לגישה אל סימboleים חיצוניים וכן פנימיים. PLT היא טבלה שעוזרת למיימוש של lazyBiding. היא נמצאת ב-.text segment. מרכיבת מרשימות של 16 בתים שכל אחד מהם מורכב מ-3 פקודות באסמבלי בלבד שמאפשרים את הבדיקה: האם הפונקציה כבר נטונה או שאנו צריכים לקרוא dynamic linker' שיטען אותה.

נראה כי יש הבדל ב-PLT בפעם הראשונה שטוענים פונקציה בין פעמים אחרות. באשר מביצעים קרייה לפונקציה advac קוראים לערכה ב-Got:

First invocation of advac



Subsequent invocations of advac



נראה כי אם הלאנו לרשימת GOT של advac עוד לפני שטענו אותה לאירועו, מתבצע כך שמא. נראה כי לאחר שטענו אותה לאירועו, במקום GOT[4] אכן הינו dynamic linker' הכניס לשם את הכתובת של advac שכך נטעה. נראה כי תמיד בעת שתקרה לא משנה באיזה פעם נבצע את jmpq *Got[4].

התפקיד של PLT הוא לדעת להבחן האם פונקציה כבר נטעה, ולא צריך לטעון אותה שוב ב-runTime אלא רק לנשת אליה, או שරיך לטעון אותה.

סה"כ באמצעות PLT GOT מושתמש בהם לביצוע הטריקים שלו - וכך הוא מבצע PIC Lazy bidding.

Patching 5.5.4

כיצד נכח קובץ הרצתה, ונשנה ממש כמו בתים בו וכתוכאה מכך הוא יתנהג אחרת. (להזכיר בדוגמה עם הסטודנטים במובוא שיצור קובץ חשוב - הרשו אותו כי הם לא חכמים במיוחד, ואניicut צריך לקחת את הקובץ שם נתנו לי ולסדר אותו.)

6 הרצאה 6

miss 6.1

הוא מצב שלא מצאתי cache בעט החיפוש cacheLine miss אחר מידע אני נזקק לכלת RAM.

א. *miss*: *cold miss* שטוען שהוא לא הזכיר עדין את המידע הספציפי זהה. תמיד יהיה לי *miss* ככל שcn בהתחלה לא הבאת *cache* כלום. תמיד קורה בפעם הראשונה שניגשים לנตอน (נחות *cache friendly* ואין שם כלום). **מעט ואי אפשר לטפל בו - בהמשך נדבר שאם הקוד אז אין אפשר לטפל בזה.**

ב. *conflict miss*: יש לנו מצב של דרישת - מבאים בлок מהזיכרון אל *cache* והוא דורש מידע אחר. באשר *block* מזכיר מקום מסוימת *cache* בפוסט והופס מידע שומר שם. זה קורה בغالילוי מיפוי, יש לנו שני נתונים שמתחרים על אותו מקום. זה אומר שהקוד לא *cache friendly* **כל הנראה**.

ג. *capacity miss*: מצב שאומר שההמם קטן מדי בשיבול להכיל את כל המידע שאינו צרי. למשל: אם יש לנו הרבה קוד שחזור על עצמו, הרבה לולאות ומידע. לעזרנו *cache* קטן מדי ולא יכול להכיל את כל *active cache blocks*. למשל אם גודל *cache* הוא $16KB$ והמידע שלנו בגודל $30KB$. **אין לנו מה לעשות איתו.**

דברינו על כך שה set קבוע לפי הביטים האמצעיים. נציג שתי אלטרנטיבות כתע - בואו ונכח את הביטים הימניים ביותר או השמאליים ביותר. האם הם אלטרנטיבות טובות? הצעה עם הביטים הימניים ביותר - על הפנים. נשים לב שגם ישר נסתכל על השני שורות הראשונות ונשים לב שיפגע *locality principle* שכן דברים שייפוי אחד אחרי השני בקוד ייפוי 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 המון כתובות עד שהביטים השמאליים ביותר יתחלפו, אז נקבל שייצור לנו ב-*cacheLine* וומס אדיר של כל הקוד ב-sets מסוימים. לכן, בחרנו בשיטה שראינו עם הביטים השמאליים: ההסתברות להתנגשות תהיה הכינונה.

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

1. *write - hit*: מצב בו אנחנו רוצים לכתוב, למשל לבצע $3 = x$ (כתיבה בלבד) וגילינו כי x נמצא ב-*cache* - יש *hit*.

ל-*cpu* יש שתי אפשרויות: אחת *write - through* היא לעדכן שירותי *RAM* או לחlopen *RAM* לא לעדכן את *RAM* כרגע, לשם כך נשתמש ב-*dirty bit* ונעדכן אותו בהמשך. ב-*CPU* שלנו יש מדיניות של *write - back*. אין לנו רצון לגשת *RAM* הרובה.

2. *write allocate*: כשעושים למשל $x + 1$, טוענים את הבלוק מהזיכרון *cache*, אחר כך כתבים אליו בתחום *cache* (שם עושים את העדכון) והבלוק נשאר ב-*cache* לגישות עתידיות.

3. *no write allocate* : כשבושים למשל $+ x$ כתובים ישירות לזכרון הראשי. לא טוענים את הблוק בכלל *.cache*

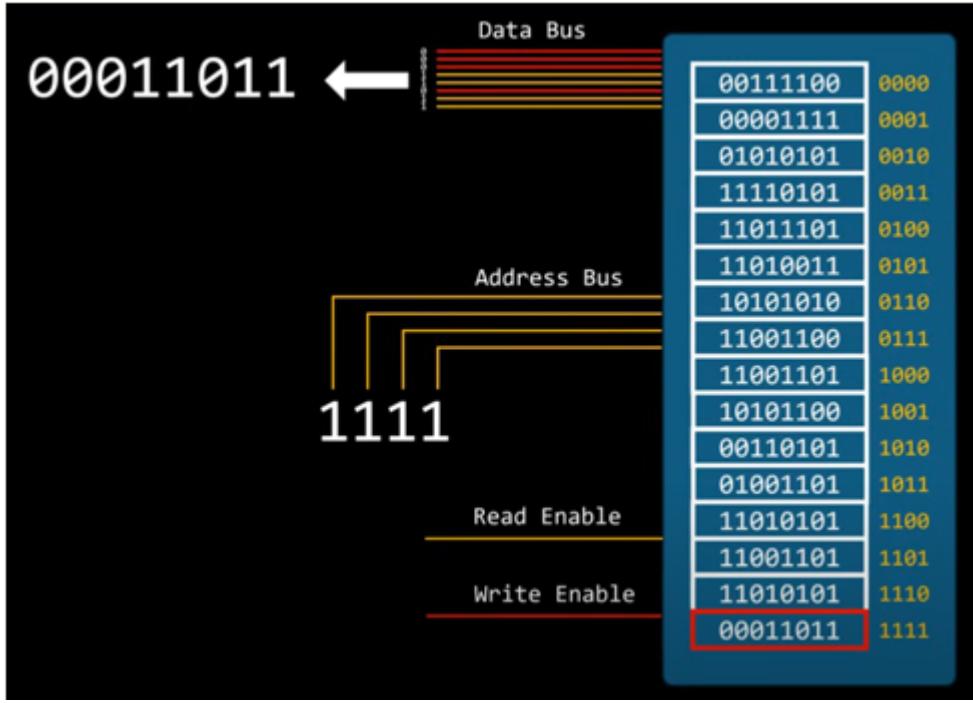
לרוב - נרצה write back וכן write back

הגדרה: *Miss rate*. לדוגמה: נניח ועשינו 100 גישות *cache* ו- 10 הינו *hit* אז אחוז ההצלחה הוא .10%. *miss rate* הוא החלק היחסית של כמה פגעו מותו כמה ניגשו לקаш. לרוב ב- L_1 מדובר על 3 – 10% וב- L_2 1%. *cache friendly* נשים לב – שזה בתנאי שהקובד הוא

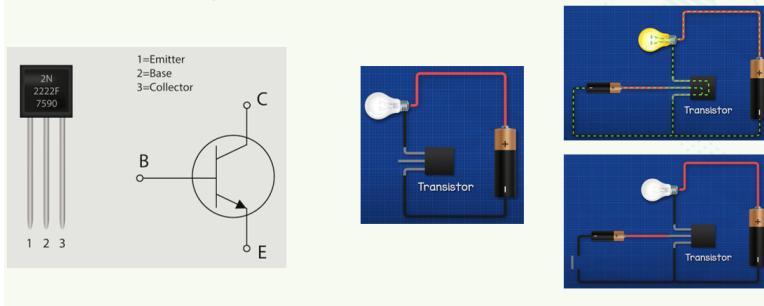
הגדרה: *Hit Time*. כמה זמן לוקח לנו לחלץ את הנתון מה-*cache*. בדרך כלל: 4 ננו שניות "ב- L_1 " ו- 10 ננו שניות "ב- L_2 ". אם כן, זה עדיף על גישה לאזכיר שעולתה 50 – 200 ננו שניות."

RAM structure 6.2

דבר על סוג זכרון שקיים לנו במחשב. לא נcosa את כולם כמובן. נרצה להבין כיצד *RAM* מחובר לכל דבר אחר במחשב. בין בין *CPU* באופן ספציפי. מהו *BUS*? 64 חוטים שעל כל חוט עובר ביט. באמצעות *cpu* *read, write enable* מוחלט האם הוא מעוניין לכתוב בזכרון או רק לקרוא ממנו. 1. אם *CPU* רוצה לקרוא משהו מה-*RAM* הוא שם *Address Bus* את המידע, מאפשר קראיה ממנו ומשיג את המידע באמצעות *Data Bus*. 2. אם *CPU* רוצה לכתוב משהו ל-*RAM* הוא שם *Address Bus* את המידע, מאפשר כתיבה אליו ושם את המידע באמצעות *Data Bus*. *data* זה הולך אל *cache* – אל החלק הרלוונטי לרегистר שהפוקודה משתמש בו.

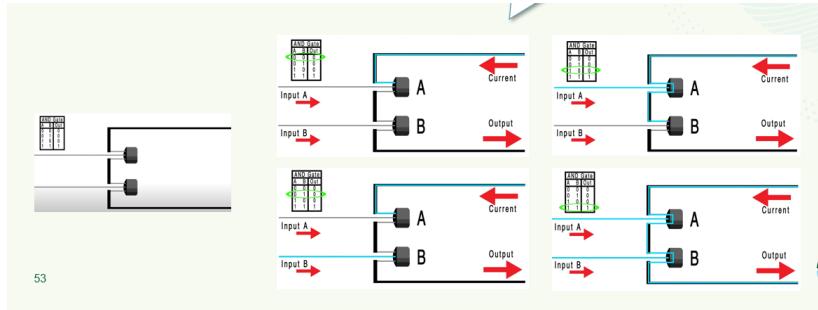


טרנזיסטור הוא רכיב חשמלי שמתפרק כמתג חשמלי או מגבר - הוא יכול להעביר או לחסום זרם חשמלי בהתאם למתח שmorphעל עליו, והוא אבן הבניין הבסיסית של כל המעבדים והמעגלים הדיגיטליים המודרניים. כפי שראים בדוגמה מטה - לצורך הבדיקה את הנורה צריך לספק לטרנזיסטור חשמל (סוללה כלשהי למשל). השן האמצעית של הטרנזיסטור קובע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



באמצעות ההבנה על הטרנזיסטור, שהוא במצב של *on* או *off* האם מעביר חשמל או שלא. נוכל להסביר שניתן לתרגום 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 מאד גבוהה וגם אם אין זה אומר שיש בטוחה מאד קטן.

מכאן התובנה שכל הארון מורכב מטנזיסטוריים. הטרנזיסטורים בונים לנו שערים לוגיים:



לצורך העניין נסתכל על שער לוגי *AND*. אנו מכירים כבר את טבלת האמת. נרצה להבין כיצד שער לוגי נבנה מטרנזיסטורים. ישים שני טרנזיסטורים, וכי שאמורנו הם מיצגים שני *input* שהם ביטים, וכן החן האמצעית זה המידע שעובר לנו מהם. אם עבר זרם דרך שני הרגיסטרים, משמעו שני השינויים האמצעיים יועבר בהם זרם, ולכן "הנורה תדלק", כלומר שני הטרנזיסטורים ידלקו, וכתוצאה לכך יזרום זרם ולכן זה יהיה 1.

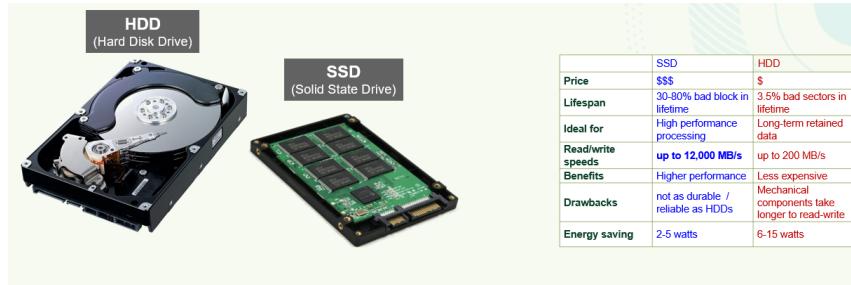
נדבר על שני סוגי של *RAM*: *S – RAM* ו-*ram*. *S – RAM* החשמל מייצג (כל בית) באמצעות 6 טרנזיסטורים - מדוע? החשמל נתוח להחלש וomers מקטינים את הרידיה בחשמל. (איך? לא רלוונטי לקוטס). כתוצאה לכך הוא עולה הרבה יותר - כי משתמשים פי 3 בטרנזיסטורים מאשר *D – RAM*. הוא גם יותר גדול פיזית - הנפח שלו יותר גדול, אך הוא מהיר הרבה יותר. *D – RAM* דינמי. החשמל מייצג (כל בית) עם שני טרנזיסטורים בלבד. כן או לא יש חשמל. כן חשמל 1 אין חשמל 0. והוא איטי הרבה יותר מאשר *S – RAM*.

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

כעת הבנו מדוע *cache* קטן - הוא עשוי טכנולוגיה של *S – RAM* ולכן הוא יקר יותר, ולכן נרצה שהוא יהיה קטן כמה שיותר, בשביל שהיה מהיר מאוד.

Disk structure 6.3

ישנם שני סוגים של 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 לפטופ יש היום. אכן *HDD* יותר זול ואמנם אנחנו צריכים לשמר מלא מלא>Data - עדיף *HDD*.

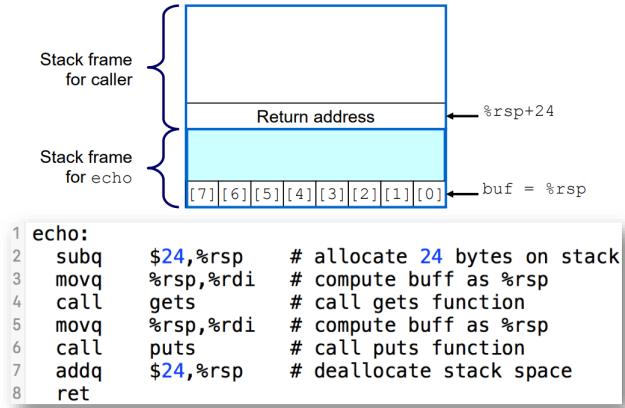
6.4 תרגול 6

ב*buffer overflow*: ב-*stack* שומרים מידע וערך חזרה לפונקציה. נראה כי ניתן ותיה זליגה מהמקום שהוקצה. למשל בפונקציה הבאה - הקצתו ב-8 בתים, נניח וה שהגנשתי אל בגודל 20, אני בהכרח יצא מהמקום שהוקצה לי - כיוון שהfonקציה הנ"ל מניחה כי הוא מספק גודל.

```

1 // Implementation of standard 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 שהקצתנו מקום עובי 24 בתים בלבד. אם נכנס 25 בתים ומעלה מה שיקרה זה שהערכיכים אכן יוכנסו למחסנית, אך הם ידרשו את הכתובת חזרה *rip* ולא יוכל לחזור להיכון שאנו צריכים לחזור בסוף הפונקציה.



מאנר. התוקף מואירק למאגר קוד זמני (*exploitCode*) שגורמת לגילשת פונקציה (*get*) מוצלת טוות בפונקציה (*echo*) ייחד עם בתים שימושתיים את תוצאות החזרה כך שייצב ערך הקוד הזמני. כשהפונקציה מבצעת *ret*, היא קופצת לקוד התוקף ומבצעת אותו - זו אחת ממשיות התקיפה הנפוצות ביותר במערכות מחשב ברשת.

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

7 הרצאה 8 + 7: אופטימיזציות 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: לאACPת לנו איך לקרוא אותה. אבל נניח כי n גדול מאוד, זו תהיה הנחיה שנדריך בכל מקרה בנושא האופטימיזציות, מניחים:

1. שהמטריצה ענקית.
2. מניחים שמספר cache לא מספיק גדול בשbill להכניס את כל המטריצה לתוכו
3. מניחים cold empty cache - כלומר במקרה אין שום דבר שלא רלוונטי לתוכנית.
4. לצורך ההמחשה - נניח 16 בייטס *cacheLine*

הקוד עבר על המטריצה לפי שורות. מה *Miss rate*? נרצה לחשב אותו. על כל 100 גישות *cache* כמה פעמים נאלצנו ללקת לאחרו. כיון שאנו מניחים שגודל שורת קאש הוא 16 בתים - כל 4 גישות למערך אנחנו צריכים ללקת לhabai *ram* *cacheLine* חדש ולכן $\text{missRate} = \frac{1}{4} \times 100 = 25\%$. אם הקוד עבר על המטריצה לפי עמודות - *Miss rate* הוא 100%,因为我们 בכל פעם נביא שורה, וכיון ש- N גדול מאוד, כל אלמנט נמצא *cache line* אחר. לכן בזדאות תמיד נאלץ ללקת לאחרו.

ומה באשר לפביל מטיריות? נשים לב שאלבורי, כפל של שתי מטריצות, הוא לקיחת שורה במטריצה A , ולקחת עמודה במטריצה B , להכפיל אותן ולקבל איבר בודד במטריצת המכפלה. נשים לב שאנו מתייחסים לבודדות *cache* או עוברים על עמודה. זה לא *cache friendly*. 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 $C = A \times B$ ו A, B *Miss rate* עברו?

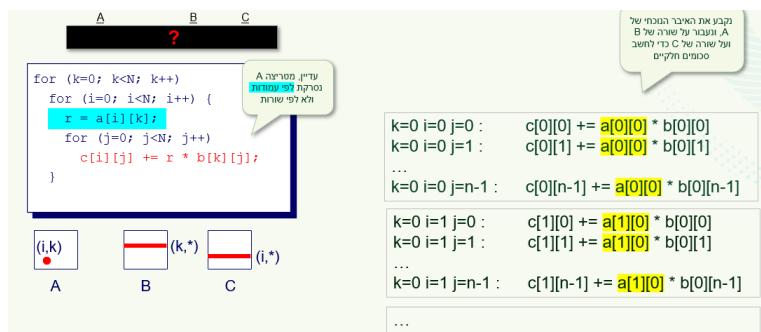
עבור A - בדיק 25%, כמו קודם.

עבור B - בדיק 100%, לוקחים עמודה, כמו קודם.

עבור C - ניתן לכטורה להתעלם מה a . *Miss rate* השניה של איבר C הוא n פעמים (במהלך החישוב, ניגש אליו בלולאה האחורה n פעמים) ולכן הסיכוי הוא $\frac{1}{n}$ וכיוון ש n גדול מאוד, אז $Miss rate$ שואף לאפס שכן נגד שהוא 0%.

כיצד נוגבר על העבודה שעוברים לפני עמודות B ? נרצה שאחוז *miss* יהיה לפחות כמו של A , כלומר כ-25%. **נתבונן בשפирו הבא:**

נראה כי אם נוכל לקבוע את האיבר הנוכחי של A , ונעבור על שורה B ועל שורה של C נוכל לחשב סכומים חלקיים! ככלומר - אנחנו נחשב לכל $[j]$ $C[i][j]$ נחשב אותו בשלבים בהמשך הרציה ובנה אותו באמצעות מעבר על שורות בלבד. זה בדיק כפל שורה-שורה.

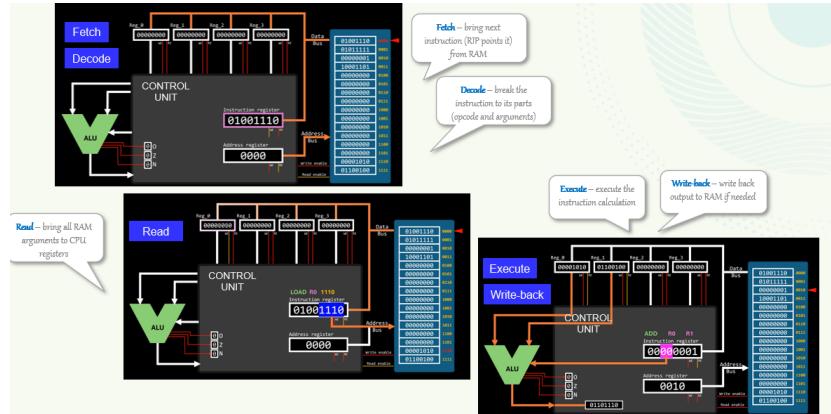


נראה כי בגישה זו עובדים תמיד עם שורות. מה $Miss rate$? נשים לב כי עבור A הוא $\frac{1}{n} = 0\%$ (כיון שאנו ניגשים לכל איבר n פעמים, שהרי הסיכוי שלא נגע אליו $\frac{1}{n}$ - ככלומר אתה מביא את האיבר פעם אחת עבור n פעמים) עבור B כתע עבור C הוא גם 25% (!) - בכל מקרה: שיפרנו את *Miss rate*.

האופציה הטובה ביותר נוספת כפל מטריצות - היא הכפלת לפי בלוקים. זה הכי *cache friendly*.

Pipeline friendly code 7.2

נזכיר לנו בחומרה. בחומרה יש *Control Unit* ו- *RAM* וכן ישנה *ALU*, *RAM* ייחידה שעושה את כל מה *ALU* צריך פרט לחישוב עצמו שעושה *CPU*.

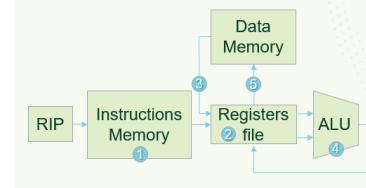


נראה כי הוא אחראי גם ל קישור לדטה בסאס, הוא מביא את ההוראה הבאה וכו'. הוא מלא לו את הריגיטרים שנחנו משתמש בהם, והוא מקבל את הפקודות לביצוע. כמובן - הוא שולט על כל מה צריך לעשות לפני ואחרי חישוב. **כל פקודה ישנים 5 שלבים:**

- : נביא את הפקודה הבאה מהזיכרון או מ-cache, *RIP*.
- : נשים את הכתובת שיש ב-RIP באנס *address*. יהיה מהיר כמוכן אם הפקודה cache.
- : לוקח את *ISA* ובודק אם ההוראה חוקית או שלא. אם יש לפקודה ארגומנטים בזיכרון הוא אחראי לדעת זאת.
- : לא בהכרח תמיד יתבצע, כי לא לכל פעולה יש קריאה מהזיכרון. בכל מקרה מדובר בקריאה מהזיכרון.
- : ביצוע הפעולה. (במהשך נדוע לא תמיד יתבצע).

אם עליינו לכתוב משוחה זירה לזכור או *cache* או זה יקרה בשלב הזה.

בכל השלבים הללו נעשים ע"י חלקים שונים של החומרה.

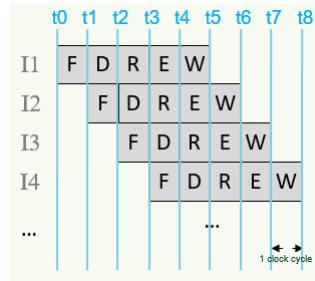


השלבים הללו שכל פקודה מבצעת - הם בלתי תלויים זה בזה.

אם נבצע את הפקודות באופן סדרתי, אחד אחרי השני, זה יראה כך:



לעומת זאת, אם נבצע אותן בסדרת מדרגה, זה יראה כך:

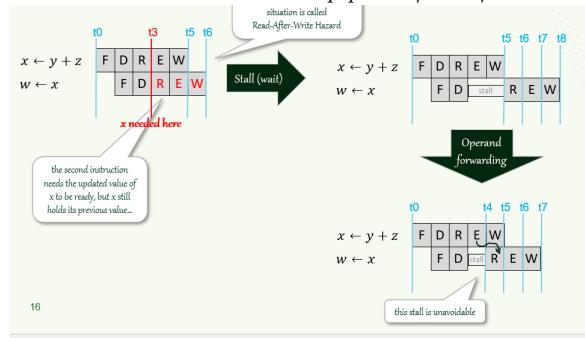


כזכור, לאחר שנסיים של רשות *fetch* ישר *latch* של השני. זו צורת *PIPELINE* זה עדייף מאשר לבצע רצף סדרתי של השלבים ולאחריהםשוב. בצורה זו מס' היסודות יורד! נראה כי במקרה הראשון עליה לנו 20 *cycles* וכן רק 8 בלבד. באופן כללי, השיפור יהיה לפי הנוסחה הבאה:

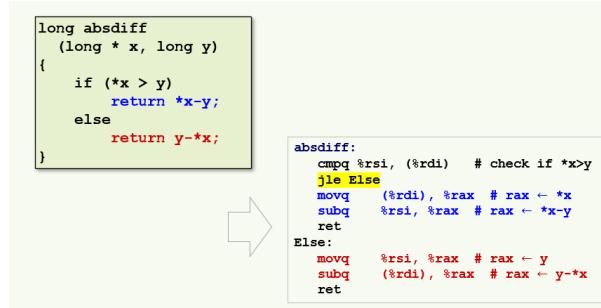
$$\lim_{n \rightarrow \infty} \frac{5n}{4+n} = 5$$

שכן, זמן הריצת הסדרתי יהיה $5n$ (5 כפול n פקודות), וכן $n + 4$ זה זמן הריצת באופן של *Pipeline* (ممלאים 4 ראשונים ואז מתחילה את השאר), ולכן השיפור שנקבל באופן כללי אם נעבד לפוי יהיה פי 5.

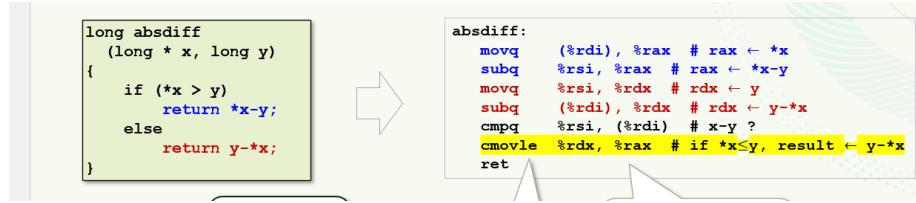
אם כן, נראה שהשיפור זה לא תמיד יעבד. נתבונן בדוגמה מטה: יש תלות בין הפקודות, בשליל לבצע את השורה השנייה כדי לדעת את השורה הראשונה בצדדי לדעת את x . נראה כי כאשר נבצע פקודה השנייה, זה בדיק השלב שהפקודה הראשונה מבצעת *execute* - היא בדיק מחשבת *read* לפקודה השנייה. זה מוביל לשנקרת הפעולה הראשונה חיבת לחכות (!). לבסוף - לא תמיין לנו נוכל להריץ באופן *pipeline*.



לאחר שהבנו מה זה *pipeline friendly*, נרצה לדון במהו קוד עבור *pipeline* נתבונן בדוגמה. באסמבלי יש *conditional move*: ישנה בדוגמה מטה פונקציה, היא מוחזירה את $|x - y|$. בתרגום לקוד אסמבלי נקבל את הקוד הבא:



נראה כי ישנה אפשרות נוספת לארוך אסמבלי זה. נרצה שלא לחשב את שני החישובים כמו קודם וקומו, אלא מראש להזכיר את היחס והאודום, ולהשתמש ב*cmovle* אשר הוא האם קטן יותר *.result*. שקול לכך שנבדוק אם אכן $x > y$ ולהזכיר את התוצאה *.result*.



מדובר זה טוב יותר? במקרה הקודם - היה לנו *if&else* או פגיעה ב*pipeline*, אך אנחנו מעדיפים קוד כמו זה שהוא סדרתי.

3 מקרים מסווגים בהם לא משתמש ב*conditional move*:

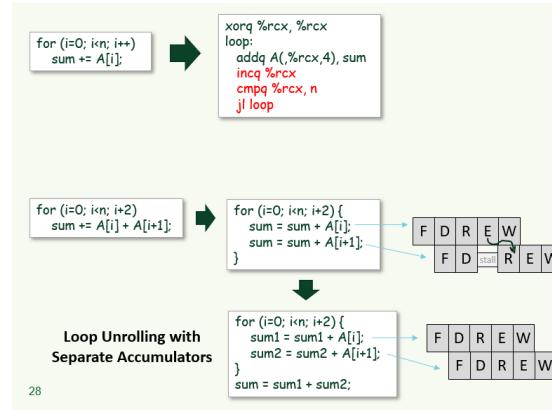
```

val = Test(x) ? Hard1(x) : Hard2(x);
val = p ? *p : 0;
val = x > 0 ? x*=7 : x+=3;

```

המקרה הראשון - אם שני החישובים ממש כבדים, מיותר להשתמש במקרה. במקרה השני - יתכן שהחישוב יגרום לשגיאת זמן ריצה אם נבצע אותו לא בדיקת התנאי. למשל, נגש ל*pointer* שהינו *null* עוד לפני שבדקנו אם הוא לא *null*.
המקרה השלישי - במקרה מושג מושגים גם את *if* וגם את *else*, ואיך נדע לחשב מראש את $x*$ ואת $x+ = 3$? אנחנו פשוט לא! וכך אסור להשתמש במקרה זה!

נראה **אופטימיזציה נוספת**: במקרה לרוץ על n נרוץ על $\frac{n}{2}$ איברים. מה קיבלנו כאן? מטפלים בולאה פעם בשתי איטרציות ולא פעם באיתרציה. וכך אנחנו בוחרים להשתמש בשני סכומים בשבייל שמניע מהבעיה שנוצרת במקרה שbamatz: הוא חייב לחכות לפני שמתקדם ונוצר *stall*啻לו שהוא חייב לקרוא את הנונינים ולא יכול להתקדם. אם משתמשים בשני משתנים מקבלים את אותה התוצאה בזמן הרבה יותר טוב. זה נקרא *loop unrolling*.



7.3 סכימת תא מטריצה

נניח כי בידינו מטריצה ריבועית ונרצה לסקום את כל ערכי המטריצה. נתבונן באופטימיזציה הבאה:

```
int ni=0;  
for(int i=0,i<n,i++)  
    sum+=A[ni+j]  
    ni+=n
```

נבחן כי נוכל להתייחס למטריצה أولי כמערך חד ממדוי, כך נעבור בכל שלב על השורה הראשונה, אח"כ על השניה... וכן הלאה. זה הרבה יותר טוב מהרעיון המקורי. ועדיין - על כל טיפול בולאלה באסמבלי אנו נדרש לבעץ 3 פקודות של טיפול בולאלה: הגדלת *j*, השוואת *j* עם *n* ואם זה קטן עדיין ללבת אל *loop* חוזרת. זה יותר מדי פעולות עבור כל לולאה ולא נרצה זאת.

נבחן בשיפור טוב יותר - עם *loop rolling*.

```
int ni = 0  
for (i=0; i<n; i++)  
    for (j=0; j<n; j+=2)  
        sum += A[ni + j]  
        sum += A[ni + j+1]  
    ni += n
```

כל אכבע: לרוב נרצה לפתחו 4 איטרציות אך צריך לבצע מודידות בפועל בשלב לווזא מס' איטרציות אופטימלי.

כפי שכבר רأינו, נוכל להמיר לשני סכומים:

```
int ni = 0  
for (i=0; i<n; i++)  
    for (j=0; j<n; j+=2)  
        sum1 += A[ni + j]  
        sum2 += A[ni + j+1]  
    ni += n  
    sum = sum1 + sum2
```

נבחן כי בקוד הזה יש לנו $i < n < j$. זה לא טוב. למה? כי אנחנו מבצעים באסמבלי בפועל *cmp* ואז קפיצה מסוימת. יותר שווה לנו לבדוק האם $0 \geq i, j$ נוכל רק לבדוק את *ZF* באסמבלי. קיבל את הקוד העוד יותר טוב הבא:

```
int ni = (n-1)*n  
for (i=n-1; i>=0; i--)
```

```

for (j=n-1; j>0; j-=2)
sum1 += A[ni + j]
sum2 += A[ni + j-1]
ni -= n
sum = sum1 + sum2

```

כלומר, אנחנו מוחללים מהעומדה האחורה וסורקים את המטריצה מהסוף להתחלה. אבל: זה לא cache friendly!

CPU הרבה יותר חכם مما שחשבנו. הוא מצד אחד מציע למערכת הפעלה אך יש לו כל מיini החלטות פנימיות שלו. הוא אומר: ישנה כאן מטריצה של סורקים אותה - אני מזהה (CPU) חכם מאוד) שמדובר במטריצה, מזכינה טוב ידוע שעריך לסרוק אותה לפי הסדר: מלמטה למעלה, והוא אומר - נכון כי ביקשו ממני את $A[0][0]$ ואני מביא לו קאשלין רצף - אבל אני מתוכנן להביא לו כמה שורות נוספות של המטריצה מהזכרנו עוד לפני שהמשתמש בקיש. זה נקרא *prefetchers hardware*. אנחנו ביחסו את האלמנט האחרון של המטריצה - והוא הביא לנו שוב איברים שאנו משתמשים בהם כי כבר השתמשנו בהם או שאינם רלוונטיים. ולכן, באופטימיזציה שהוצעה אנחנו ממש מפסידים - אנחנו סורקים הפוך וזה לא טוב. אז בכלל לא אופטימיזיה!

נתבונן בגרסה הטובה ביותר לסריקת מטריצה לחישוב סכומה:

```

int *p = &A[0][0]
int *end = p + (n-1)*(n-1)
while (p != end)
    sum1 += *p++
    sum2 += *p++
    sum = sum1 + sum2

```

מה קורה כאן? מגדירים כת פוינטר שמחזק את הכתובת של האלמנט הראשון $A[0][0]$. אנחנו מקבלים את הכתובת של האלמנט האחרון על *end*. ולאחר מכן - כל עוד ההתחלה לא שווה לסוף, אנחנו מקדמים את הפוינטר הראשון ומוסיפים את הסכומים, עד שהם נגশים. ככלומר *sum1* מчисב את סכום האלמנטים האי זוגיים *sum2* את סכום האלמנטים הזוגיים. חשב להציג כאן כי ציריך שני סכומים שונים בגלגול *loop unrolling*. הערה חשובה - ניתן לעבור כך על המטריצה כי בזיכרון היא מייצגת כמוון כרצוי אחד של מערך חד ממדי.

יתרונות של גרסה זו:

- * סריקה של האخرון בסדר עולה
- * ניצול מרבי של *pipeline*
- * ביטול ללואות מקוננות
- * *conditional jump* יחיד
- * אין צורך לחשב כתובות כל הזמן

חשוב. יתכן בבדיקה שאלת דומה (מרגע אמרה שזו שאלה שרצה לשים בבדיקה שלנו **כשלת בונוס**). נבחן בפעולת *increment : rax* כלשהו. יש לנו פעולה של 1. *add rax*. הן עושות את אותה הפעולה בבדיקה. בבדיקה *CPU* - מהירות שתי הפקודות זהה. אם כן, אם נשתמש בפעולת התוכניתית תעבור יותר מה. מדוע זה קורה? הקידוד לשפט מכונה של *add* ארוך יותר מהקידוד לשפט מכונה של *increment*: ולכן *add* של *fetch* לוקח יותר זמן. נבחר כי *fetch* לא תמיד (מש לא תמיד) לוקח אחד כמו שהנחנו שקרה - אנחנו זאת בשביל לחשב, בפועל המצב שונה. מבייא פקודה מהזכרנו ואם היא גדולה יותר בזיכרון - יקח לו יותר זמן.

RAM friendly code 7.4

שאנו שורקים את המטריצה והיא יושבת בזיכרון בקצב זה סוג של *RAM friendly*. נניח ויש לנו משתנה $x = 12345678$ שהוא גודל יותר מ-*int*, *Bytes*, במקרה שלנו הוא *int* ולכון *4Bytes*. הם ביטים רציפים בזיכרון - זה מאוד מושה.

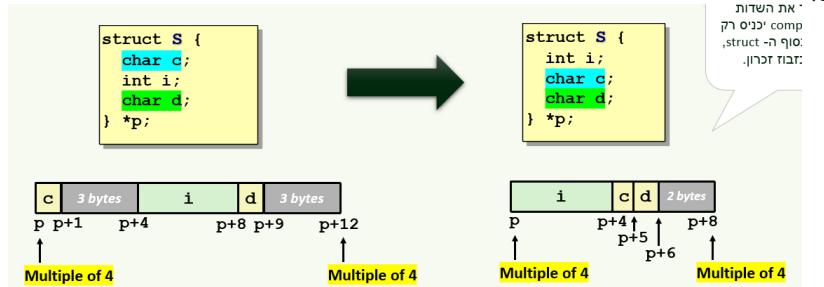
אם הכתובת של x מתחולקת ב-64 לא שארית (64 כי *cacheLine* הוא 64 ביטים), אם נרצה לקרוא את x נדע כי כל *4Bytes* שלו יהיו בשלמותם באוֹת שורת *cache*. יותר מזה: גם אם הכתובת של x מתחולקת ב-4 לא שארית - כל הביטים של x יהיו באותו *cacheLine*. מדוע? נניח 1028 (לא מתחולק ב-64, כן מתחולק ב-4).

נבחן כי כל *Bytes* של x icut יהיה עדין באותה שורת קאש כיון שכינסו בשורה קאש חדשה ב-4 בתים הראשונים. אם למשל x בגודל 2 ביטים, ונמקם אותו במקומות 1022 או 2 ביטים הראשונים יופיעו ב-1022 – 1024 והמסקנה: הם לא יהיו באותה שורת קאש.
יותר מזה – אם יש לנו משתנה x בגודל t ביטים, מספיק למקם אותו בכתובת שמתחלקת ב- t בשביל שהוא באותו שורת קאש.

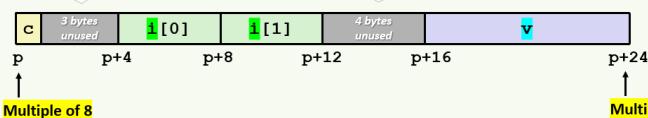
נתבונן במבנה פשוט של struct. מה שהgcc עושה במקרה זה זה שהוא מסתכל על השדה של הסטרuktאט בגודל הגדול ביותר – במקרה שלנו כאן זה *int* ולכן נרצה כי הסטרuktאט יתחל ויסיים בכתובת שמתחלקת בשדה הגדול ביותר בסטרקטט: כמובן גם הכתובת של ההתחלה וגם של סוף הסטרuktאט יצטרכו להתחולק ב-4 במקרה שלו.

כלל: הקומפילר מספיק חכם בשביל לעשות זאת בעצמו, ככלומר הקומפילר דואג שכותבת התחלה והסיום של הסטרuktאט יתחלקו בערך השדה הגדול ביותר בסטרקטט. (מדוע? מודינה אמרה שאנו לא צריכים להבין).

נבחן כי אם נסדר את השדות אחרת, כמו בדוגמה מטה, יש לכך שימושות עבור הקומפילר. הוא יסדר את השדות אחרת ויכניס רק *gap* אחד בסוף *struct* וממנו מבזבז זכרון. נבחן כי באופציה משמאל הוא שם את *char* ולאחר מכן יוצר מכך *int* להתחולק ב-4 הוא יוצר שולץ של 3 בתים מיוחדים עד שהוא שם את *int*, לאחר מכן הוא לא יזוק *gap* עבור *char* שכן הוא יכול לשום *char* בכל מקום, ולאחר מכן הוא שוב לזוק *gap* כיון שהכתובת האחורנית של *struct* הייתה להתחולק 4.

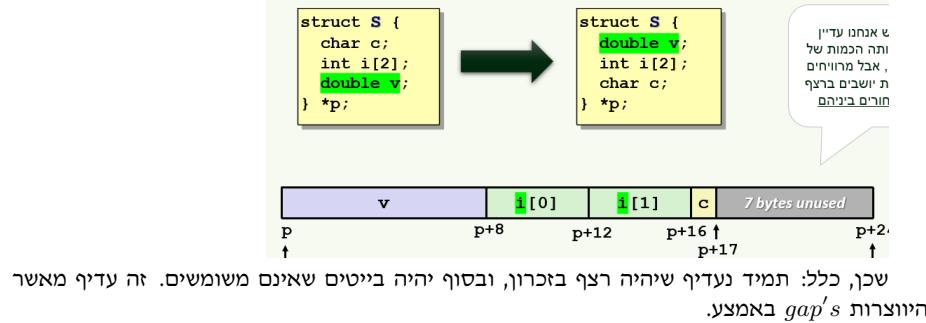


דוגמא נוספת. נניח ויש לנו סטרקטט של *i[2], double : char, int*. הגודל הכללי הוא המערך $8 \times 2 = 16$ וכאן כתובות התחלה והסיום חייבות להתחולק ב-8. אם כן, נראה כי קיבל את המיצב הבא בזיכרון:



כאן למשל, בכל מקרה אין אפשרות לשפר את הסידור. תמיד מקבל שhcottובות האחורונה תסתדרים $.p + 24$.

אם כן, עדיף שנסדר את הSTRUktAT באופן הבא ונקבל:



compiler & optimizations 7.5

הקומפיילר ידע לבצע אופטימיזציות. ובפרט: אופטימיזציות פר בקשה מהמשתמש. אם נכתב לו `-O3` – זה אומר לקומפיילר – לא מגביל אותו, תתרפע עם האופטימיזציות. נתבונן בקוד הבא:

```
#include <stdio.h>
int main() {
    char c = 125;
    while (c > 0) { printf("%d ", c); c++; }
    return 0;
}
```

נבחן כי הקוד מדפיס 125, 126, 127 ולאחר מכן *char*ים יסתהים כי *signed* הוא *negative*. נסתכל על הדוגמה הבאה, מה כתעת ידפס?

```
#include <stdio.h>
int main() {
    char c = 125;
    while (c < c+1) { printf("%d ", c); c++; }
    return 0;
}
```

האם נקבל 125, 126 ? לא יתרה זה נשמע הגיוני, כשהגענו $c = 127$ לא יתקיים הדרוש ונסיים. בפועל תהייה לנו לולאה אינסופית: הקומפיילר נורא חכם. אך לעיתים הוא מניע הנחות שגויות. כאן, ובכל קוד בשפת הקומפיילר יניח שאין *overflow*. מה הכוונה? בדוגמה הקודמת.cn היה *overflow*. למה שהקומפיילר בוני לבצע אופטימיזציות בקוד ואם הוא יהיה שיש אברפלו הוא לא יוכל לבצע את האופטימיזציה הבאה: הרי, תמיד $c < c + 1$ ולכן הקומפיילר הולך לתרגם את השורה ל *while(true)* ולכון נקלט לולאה אינסופית. זו **דוגמא לאופטימיזציה של הקומפיילר**. لكن – כיוון שהוא מניה שאין אברפלו, הוא ביצע את האופטימיזציה וכאנן קיבלו לנו לאינסוף. עברו הקוד שלנו החלטה זו הייתה שגויה אך זה מה שקרה בפועל. זה נקרא *behavior undefined*.

מסקנה: יש קומפיילר, ותמיד עדיף לבצע `gcc -O3`. אך כיוון `gcc` מבצע לעיתים הנחות שלא בהכרח טובות לנו, אז אנחנו צריכים להגדיר לקומפיילר למקרה `-O1` – וכו'. אך במקרה זה, הידע שלנו באופטימיזציות יהיה שימושי – נדרש לכתוב אופטימיזציות בעצמנו ולא לסמוך על הקומפיילר שיבצע אותן.

כלל: קומפיילר או חלופין אנחנו, אסור לו לשנות את התנהגות התוכנית. כלומר: על אותו הקלט, גם התוכנית וגם התוכנית עם האופטימיזציות חייבים להחזיר את אותו הקלט. אחרת – לא ביצענו אופטימיזציה והתוכניות לא שקולות.

7.6 מה הקומפיילר לא יכול לעשות?

דוגמה ראשונה. נתבונן בקוד הבא שמרת מחרוזת *lower case*:

```
void lower(char *s) {
    for (int i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

באשר הקומפיילר מסתכל על הקוד זה הוא מוחפש אופטימיזציה. כאן, ישנה ה-אופטימיזציה שהכי מושפעה על מהירות הקוד. נבחין כי אנחנו מחשבים את *strlen* בכל שלב *i*, ולמה שלא נוציא אותו החוצה? הרי כל פעם אנחנו מחשבים את *strlen* בועלות ($O(n)$). אם כן, הוא לא מתכוון לשנות את זה. מדוע? הקומפיילר ראה שאחנו תוך כדי זמן הריצה משנים את המחרוזת, בדוגמה שלנו אורך המחרוזת לא משתנה, אך הקומפיילר לא עד כדי כך חכם הוא לא ידע זאת! הוא זיהיר, ייתכן כי במקרה מסוים נעדכן $S[i] = null$, ואז אורך המחרוזת יקטן. הקומפיילר זהיר והוא לא מתעסק עם זה. אז מה, אנחנו נכח קוד שכזה בסיבוכיות ($O(n^2)$) ממש לא. אנחנו עדיין נשתמש ב-*gcc* – *o3* ונסנה בעצמנו ידנית את הקוד:

```
void lower(char *s) {
    len=strlen(s)
    for (int i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

דוגמה שנייה. נתבונן בקוד הבא.

```
void f(int *x, int *y) {
    *x += *y;
    *x += *y;
}
```

אופטימיזציה מיידית שעולה לנו היא לבצע:

```
void f(int *x, int *y) {
    *x += 2*y;
}
```

כעת, יש לנו 2 גישות לזכרוון, פעולה חיבור אחת, פעולה השמה ופעולה *shift*, סה"כ באסמבלי תבצעו 5 פקודות. זה הרבה פחות מהקוד המקורי, הקומפיילר עדיין לא מתכוון לשנות את הקוד לקוד זהה. נתבונן בדוגמה הבאה.

$f(&x, &x)$

כלומר, אם קיבלנו פערמים *x*, בגרסה הקודמת קיבל $4x$ ובגרסה שלנו $3x$. שcn, $x + = x$ בשורה הראשונה ייב $2x$, ולאחר מכן $2x + = 2x = 4x$. אם כן, בשורה השנייה נקבע $3x \neq 4x$. כמובן $2x = 3x$. אבל אם נבטיח ולכן, הקומפיילר לא יעשה זאת. ואנחנו: גם לא בהכרח יעשה את השינוי הזה – אלא אם נקבע *Pointer Aliasing*.

דוגמה שלישית. נתבונן בקוד הבא:

```

void compute_total_dest_naive(int* vec, int len, int* dest) {
    *dest = 0;
    for (int i = 0; i < len; i++)
        *dest += *vec++;
}

```

Do compilers carry out such optimizations?

No!
Because of memory aliasing

```

void compute_total_dest_buffer(int* vec, int len, int* dest) {
    int tot = 0;
    for (int i = 0; i < len; i++)
        tot += *vec++;
    *dest = tot;
}

```

על פניו, נראה הגיוני מאד זה אכן חוסך בזמן ריצה בנוו שניות. במקום לגשת לפונקטר בכל איטרציה, ניגשים אליו רק פעם אחת בסוף. מדוע הקומpileר לא עושה זאת בלבד? יתכן מצב שנקרא *memory aliasing*: אם *vec* ו-*dest* יוכלים להפנש (אצלנו, לא) אי אナンנו נקבל כלל אחד מהחוקדים פולטים פלט שונה. לכן הקומpileר לא יודע מה הייתה כוונת הכותב ולפיכך הוא לא מבצע אופטימיזציה זו בעצמו.

דוגמה ריבועית. יותר מדוגמה - ריעו: פירשו לקחת לולאה מסוימת, שהגועה שלה הוא שכפול של כמה פעמים של גורף הולולה המקורית. למשל, במקום לבצע $i + i = 2i$. מה הרעיון בכך? להעזר בקוש, ובוצע כמה שפחות פעודות כמו קידום, בדיקת תנאי עצירה וכדומה. נצורך להבחין כי תנאי העזירה שלנו ציריך לששתנות בהתאם במקורה זה, אם מס' האברים במערך אי זוגי למשול. נבחן גם, שבשביל לשפר עוד יותר את הקוד נוכל להשתמש בשני סכומי עזר *sum1*, *sum2* שיתעדכו במהלך הקוד ובסוף לעדכן את *sum = sum1 + sum2*. **אם שמים *k* פעמים במהלך הלולאה את הקוד הנוכחי, ומשתמשים במשתנה צובר אחד, זה נקרא $\times k$ לפונקציוניג.**

7.7 שלבים בדרך לכנתיבת תוכנית אופטימלית

נרצה לכתוב תוכנית אופטימלית בזמן הריצה, מכוערת ככל שתהייה. העיקר: שתהיה יפה בזמן הריצה. מהם השלבים?

- בחר את רשימת התכונות הטובה ביותר ביותר עברך. "סקר שוק".
- תבחר את האלגוריתם הטוב ביותר (האופטימלי בזמן הריצה) עברך.
- בחר את מבני הנתונים הטובים ביותר (מבחן זמן הריצה) עברך.
- בחר את שפת התכונות הטובה ביותר עברך.
- כתוב את הקוד פשוט ותיקן - קודם כל شيء קוד שעבוד ותיקן.
- בצע אופטימיזציות היכן שהיא טוב ונחוצה. הרץ בדיקות היכן התוכנית מבזבזת הכח הרבה - במקומות אלו, בוצע אופטימיזציות.

* חשוב להשתמש בקומpileר טוב. *gcc* הוא קומpileר טוב.

Measurement challenge 7.8

נרצה למדוד זמן של תוכנית. כיצד נעשה זאת? נוכל בתחלת כל פונקציה לשום *clock* בתחילת ובסוף, ובסוף התוכנית להדפיס את הפרש בין זמן הסיום לזמן ההתחלה ולדעת כמה זמן התוכנית שלנו רצתה.

```
void main() {
    clock_t start = clock();
    ...
    clock_t end = clock();
    printf("Time: %f seconds\n", (end-start));
}
```

The screenshot shows a terminal window titled "marina@vm: ~/Archi". The user runs the command "gcc main.c" and then "time ./a.out". The output shows the following CPU time statistics:

Mode	Time
real	0m1.234s # wall-clock time
user	0m1.230s # CPU time spent in user-mode
sys	0m0.004s # CPU time spent in kernel-mode

A callout bubble points to the "user" mode line with the text "User mode - run our code".

רעיון נחמד, אך פחות טוב: כשאנו מודדים זמן שאיננו צריכים למדוד. מדוע? הרצינו תוכנית, ובו אמונותיה שתוכנית רצתה רצוי דברים אחרים במחשב - לפחות מערכת הפעלה רצתה, ויתכן שגם תוכניות אחרות, כיון שכמויות התוכניות שלנו במחשב קטנה מסופר *CPU* שיש במחשב, מערכת הפעלה נוטנת קצת זמן ריצה לתוכנית אחת, קצת לשנייה, וכך היא מರיצה את כולן במקביל. אך בפועל: יתכן שהחלק מהזמן שהתוכנית הוא רצתה אכן לא אורורה היא בכלל לא רצתה אלא המתוינה *CPU* יריד אותה.

נסתכל על אפשרות שנייה - האפשרות למטה בתמונה. נכתב *time* לפני *a.out* ונתקבל שלושה :

זמן שבאמת התוכנית קיבלה זמן *CPU* - *real*
זמן מותך כל הזמן שהוקצה, כמה זמן היא הייתה ב-*user mode* - *user*
זמן מותך כל הזמן שהוקצה, כמה זמן היא הייתה ב-*kernel mode* - *sys*

שתוכנית רצתה היא מבצעת פקודות - פקודות רגילות ופקודות שאסור לה לבצע (להציג למשל, היא צריכה לבקש מהמערכת הפעלה גישה לדבר עם המסך.). הוא מצב משתמש - התוכנית שולץ רצתה עם הרשות מוגבלות, לא יכולה לשמש שירות לחומרה ולא יכולה לבצע פעולות מסווגות. רוב הקוד נמצא *kernel mode*. רק כשהתקה מצב ליבה - מערכת הפעלה רצתה עם הרשות המלאות ויכולת לגשת לכל החומרה, רק כאשרה מבקש שירות. מתקיים $real \approx user + kernel$

8 הרצתה 9

9 הרצתה 10

10 הרצתה 11

11 הרצתה 12