

מבנה מחשב - סיכום הרצאות למבון

6 בינואר 2026

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

תוכן עניינים

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

23	<i>Jump Table</i>	3.6	
25	<i>GDB</i>	3.7	
26	4	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	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	קדמה	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	<i>files&system – Calls&shell</i>	9 + 10	הרצאה
58	<i>Unix files</i>	8.1	
60	<i>System calls</i>	8.2	
62	<i>Unix processes</i>	8.3	
64	: שורת הפקודה <i>Unix shell</i>	8.4	
65	שימוש בקוד אסמבלי בתוך קוד ב- <i>C</i> (מהתרגול)	8.5	
66	(<i>Undefined Behavior</i> מהתרגול)	8.6	
67	(<i>Macros</i> מהתרגול)	8.7	
68	<i>Co(operating) – routines - Coroutines</i>	10 – 11	הרצאה
70	<i>yield</i>	9.1	הfonקצייה
72	תרגול	9.2	

74	10	הרצאה אחרת: נושאים מתקדמים
74	10.1	היסטוריה של ארכיטקטורת המחשב

1 הרצאה 1

1.1 מבוא לקורס

הקורס יתמקד בשני תחומים:

1. מבנה מחשב - חומרה

2. שפת מחשב שנוגעת לשירות בחומרה - *Assembly*

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

אם $0 \leq x^2$? לא בהכרח - במתמטיקה כן, בודאי ב \mathbb{R} . בעולם התאורטי, זו טענה נכונה.
במציאות: ראיינו כבר כי בינהנן מס' מאד גדול, למשל 1410065407 נקבל כי $x^2 = x$ אז $x^2 < 0$ – מודיע? כמו שנלמד במובא – יש *overflow* והיצוג הופך לשילילי. אז
כיצד נתמודד עם זה?ណו בזאת במהלך הקורס.

האם מתקיים $(y + z) + x = y + (z + x)$? בעולם התאורטי, כן. עם זאת – לא כל מספר עשרוני ניתן לשימירה במחשב. למשל אם כמות הספרות אחרי הקודה גודל מכמות הספרות שאפשר להחזיק, המחשב חותך את הספרות שהוא לא יכול לשמר – וזה הוא מקלט מס' שאינו מדויק. ולכן לא יתקיים השוויון הנ"ל במחשב בשל הסטיות הללו.

וכאן הדגש: **בתאוריה המתמטית – לא יוכל שיקרו בעיות כלל.** בעולם האמיתי, במחשב: זה למורי קורה. ועלינו ללמוד כיצד להתמודד עם זה.

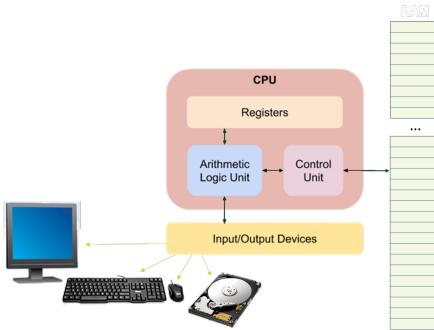
נתבונן בקוד הבא:

```
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 האינטואיטיבי, בו אנחנו רצים ללא קפיצית זה שלוש. הקוד הזה הרבה פחות יפה – אבל בהמשך נבין (*CPU*) שהוא רץ הרבה יותר מהר: זה **כל מה חשוב**, עילוות.

COMPUTER STRUCTURE 1.2

המחשב בנייתו מ*RAM*, *CPU* וחומרה (*Input/Output Devices*) זכרנו.



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

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

1.2.1 מודיע המחשב שלנו בגיןאי? למה שהמחשב לא יהיה טרינארי?

כאשר מעבירים מידע ממוקם למקום (בתוך המעבד) נשלח signal החשמלי. אנחנו רוצים לתרגם את המידע שיש בסיגנל החשמלי לביטים. סיגנל מגע עם רעשנים - כמו כן בתמונה מטה. מוגדרים טווות של עצמתה הסיגנלית. בין 0 ל/255 הוא מותפרש כбит 0, בין 0.9 ל-1.1 מותפרש כ-1, ובכל השאר הוא מהתפרש כמצב מעבר.

באופן תאורטי - יכלו את מצבבי המעביר להגדיר כמצב השילishi - ואז לעבור עם בית" שלישי, במצב טרינארי: מהר מאד ירדו מרעינו זה, כיון שהוא הרבה יותר רעשנים ותנוונות ואז במקומות לחושב שקייבלת מס' 8 קיבלו 9. נוצרו הרבה בעיות כתוצאה מרעינו זה - ולכן בשbill להבטיח את תקינות

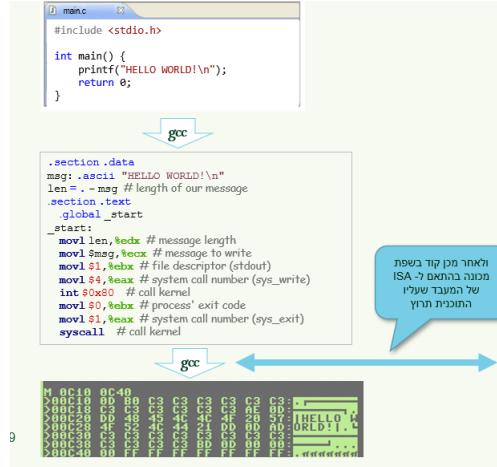


Instruction Set Architecture - ISA 1.2.2

(מרינה אמרה שהיא שואלת **לפעמים ב厰**). לכל דגם של **CPU** יש אוסף פקודות שהוא ידוע לבצע. אוסף פקודות נקרא שפת מכונה. לכל פקודה כזו קיימות פקודה מקבילה שקיימת בשפת **ISA**. **ISA** הוא ספר שמרכז את כל הפקודות שאוטה ארכיטקטורה / **CPU** יודע לבצע. An אסמבלי. דוגמים שונים של מעבדים יכול להיות שונה. בהינתן שנדע את הספר הנ"ל - נדע איך פקודות נוכנש כתוב בשbill לכטוב את הקוד שלו. הפקודות בספר יהיו כתובות הן בשפת אסמבלי והן בשפת מכונה.

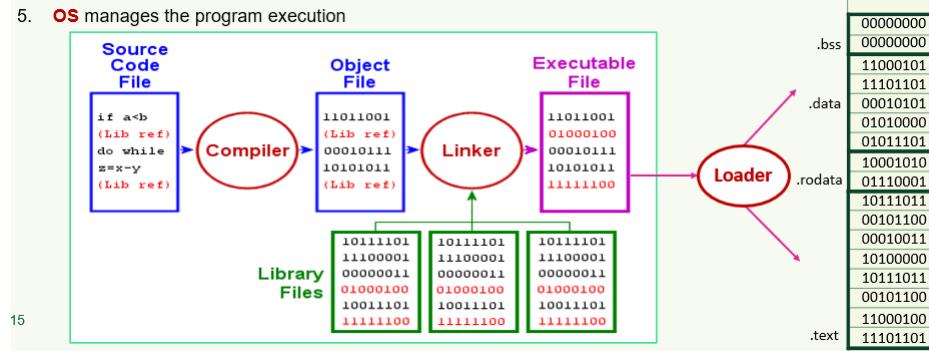
הקומפפיילר הוא זה שייצטרך את **ISA** בשbill לדעת לתרגם את הקוד לשפת אסמבלי.
 קוד בשפת **C** יתורגם לשפת אסמבלי באמצעות הקומפפיילר, ולאחר מכן **ISA** יעזר לתרגם את הקוד לשפת מכונה.

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



1.3 תהליכי ההרכבה ותמונות 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 שלו לקודים אחרים שקיים בספריות אחרות בהם אנחנו השתמשנו בקוד, זה קוד שכבר מוכן ומוקומפל - והוא אחד אותם לקובץ יחיד. שיקרא *Executable* שמוון ל�ומפל.
4. **שלב רביעי:** בטרמינל, אנחנו כותבים למשל *a.exe*.>: למעשה, מה שקרה הוא שאנחנו אמרנו למערכת הפעלה - תקpidו לטוען את הקובץ לאירוע. **ראשית** הוא מודוא שקובץ זה ניתן להרכבה : *Loader* ("טוען" - תקpidו לטוען את הקובץ לאירוע. **שנית**, הוא מפרשר ("חותך ושם בכל מקום בזיכרון מה שצריך לשבת שם") מערכות הפעלה שלנו. **שלישית**, הבודק מודלג על אזור ההואות. **לפעשה Loader** יודיע על מה עליו - בדוק כמו שבדיקת מבחן, את מה שהקומPILEר והLINKER עזרו לו לקובץ, וזה הפרסרור) את מה שתוב לו, והוא צריך להבין באיזה מקום של הקובץ מופיע *data* ומשם את זה באיזה מקום *data* בזיכרון, להבין מה צריך לћנס *text* בזיכרון ולשים את זה באיזור זכרונו וכן הלאה). **תקpid נסף שלו** - הוא לאתחל את המשתנים בהתקלה.
5. **Process Image**: כתע לאחר שהשתמשנו ב-*Loader* קיבלנו *Process Image*: מכלול של זכרו (תמונה זכרו) שמקצתה ל佗ת התהליכי *Loader* אחראי לו. כלומר, כל זכרו שמקצתה ל佗ת התהליכי במהלך ריצת התהליכי.



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

data: משתנים גלובליים שמאוחתלים כבר עם ערכים.

const: משתנים גלובליים סטטיים שהם כבר נקבעו וهم כתעתך נקרים.

rodata: איזור הקוד של התוכנית. כמוין ספר". בעת קריאת קוד קוראים מיל.

text: איזור הקוד של התוכנית. כתעתך קוראים מיל.

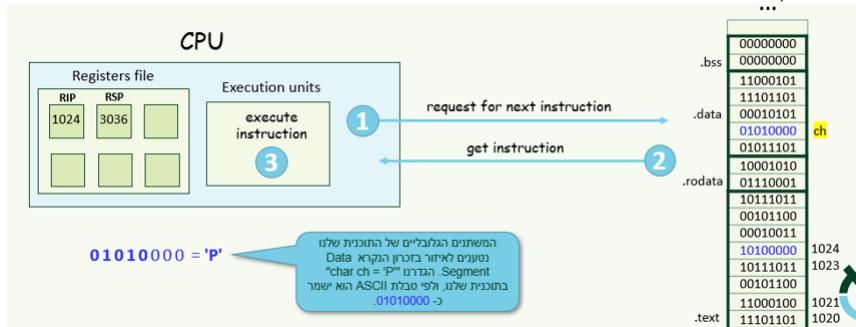
כאשר כתבבים פונקציה, נפתח ה-*activation frame* בו מוקצים כל המשתנים של הפונקציה, בעת סיום הפונקציה עם return ה-*פריים* מסר לשובת ה-*פריים* הבא שייפתח.

1.3.1 איך ה-CPU יודע מה רצף ביטים מייצג?

ה-CPU מבקש *instruction* ה-*Loader*, מקבל אותו ומבצע אותו. הרג'יסטר שהוא סימס את ההוראה הנוכחיית, הוא מקבל את הhabba ומבצע אותו. הוא מאוד מורכב - אך בפועל הוא עובד בצורה פשוטה. *Loader* מכיל מידע אודות ה-*icon* ה-*main* ב-*.text* הוא יודע את ה-*instruction* הראשון הראוי את השתוכננות צרכי להצע - בעברית: נקודת כניסה", *Loader* יהיה במהלך תהליך ה-*פרוסר* את המיקום הראשון שממנו השוכננות מתחילת.

(*Instruction Pointer*) RIP שמצוין על הבית הראשון של הפקודה הבאה לביצוע. *Loader* שם בתוך ה-*רג'יסטר* ממש RIP את הכתובת הזה במהלך ה-*פרוסר*. (עזרה - גם באסמבלי נוכל לגשת לרג'יסטרים בזיכרון, מה שאי אפשרות בשפות עילית).

ל-ISA יש ביד את ISA - הוא מקבל את הכתובת הראשונה בתוכנית והוא פותח את ה-ISA והוא בודק אם קיימת ISA פקודה עם הכתובת זו. אם כן הוא מבצע אותה - ואם לא הוא מתקדם לכטובת הבאה בזיכרון, הוא מגדם את המיקום בזיכרון אחד (אל הביט הבוא). אם הוא מקבל פקודה - הוא חוזר לשלב 3, ומבצע את הפקודה. אחריה הוא ממשיך להעלות אחד בכל שלב ומתקדם בזיכרון. ...



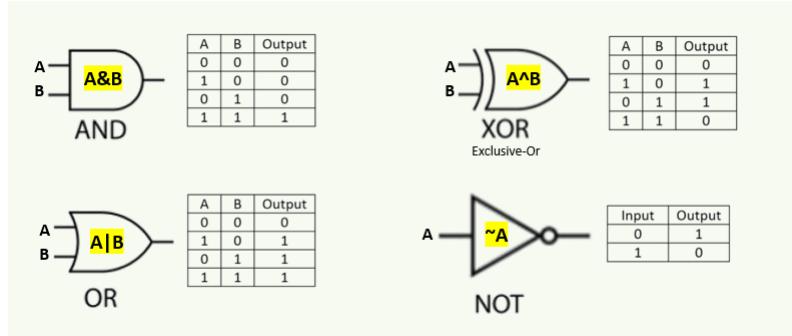
המעבד קורא תחילה (prefix) של פקודה, מזיהה שזה פקודה בזכות ה-ISA ומפענה אותה.

ה-*Registers file* (Stack Pointer) RSP נמצא בתוך CPU. מצבע על תחילתו (כיוון שם הערך גדול מדי, לפי מבוא הוא נשמר מלמעלה למטה בזיכרון) של הערך האחרון שנכנס למחסנית. המחסנית חשובה מאד כיון שיש בה משתנים לוקלים, *activitaion frame* וכטובה חזרה וכן ארגומנטרים שפונקציה מקבלת (אם כמובן).

הערה: בתוך CPU ישנו מס' רג'יסטרים שיושבים באיזור Registers file. למשל: *RSP, RIP*. המשתנים הגלובליים של התוכנית שנשמרו נטען לאיזור בזיכרון הנקרא *data*, אם הגדרנו אותן 'p' למשל בתוכנית - לפי טבלת ASCII הוא ישמר בהתאם אליה.

Bit Level Operation 1.4

בහינתן אוסף ביטים, נפעיל פעולה ביוטוי בין ביטים. ישנו מס' פעולות, כלהלן:



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

פעולה *Shift Right*: מזיז ביטים ימינה.

פעולה *Shift Left*: מזיז ביטים שמאליה.

דוגמה. **יצירת קבוצה באמצעות ביוטוי.** נרצה ליציג וקטור באורך n באמצעות ביט: $\{1, n, \dots, 0\}$. כל ערך בוקטור יכול להיות משוייך לקבוצה יוכל להיות שללא. הקבוצה תהיה A . אם $i \in A$ אז $a_i = 1$. כלומר - אם איבר בקבוצה מיקומו בוקטור יהיה 1. אם רצחה לחושף ערך לקבוצה - נבצע פעולה *or*: מודיע? גסיף ביט עם הערך 1 על 1 וכל השאר אפסים, וכך נקבעה *or* נקבע קבוצה חדשה עם 1 בתוכה. אם רצחה למזויא את הקבוצה המשלימה - \bar{A} : נבצע *not* על הביט. **לחיתוך** שני קבוצות - נשתמש באופרטור *,and*, **ולאיחוד** שני קבוצות - נשתמש באופרטור *.or*.

חשוב לדעת: פעולות בינאריות הן הפעולות המהירות ביותר שנitin לביצוע.

1.5 תרגול 1

ישנו מס' כלים שנועדו להקל את החיכים של המתכנת.

Nano, Notepad, Vim, Emacs :Text Editor: סביבת עבודה שגם ניתן לכתוב בה את הקוד גם להריץ אותו, למשל: *Clion, VsCode, VS*: סט כלים שמכיל בתוכו כמה שלבים שהשלבים האלו יחד מעבירים תוכנית שתכתבנו *Compiler*. מקובץ קוד לקובץ הרצה שנוכל להריץ על גבי המחשב. עובדים עם קומפイルר *GCC*: *Project Builder*: כלי ש"שומר" על סדר תהליכי יצירה קובץ ההרצה, הוא מודא לכל הקבצים בתוך הפרויקט מועברים כמו שירץ במהלך יצירה קובץ ההרצה. כמו *Makefile, Cmake*: דרכיהם בהם ניתן לבצע מושימות באמצעות המחשב שלנו. כמו *Computer Interfaces GUI, GUI, Shell* למושל משמש לפיתוח וסגירת חלונות.

:Shell

תוכנה שבמסגרתה אנחנו יכולים להכניס פקודות ומערכות הפעלה מריצה אותם בפועל לאחר שמקבלים פלט על הפקודה. פקודות מסוימות: *ls* - מראה לנו איזה קבצים /תיקיות קיימים בתחום התקיה שאנו נמצאים בה. *pwd* .1 - מראה את הנוכחיPWD. *pwd* .2 - מראה את הנתיב בו אנחנו נמצאים כרגע. למשל *.guy/desktop* *.mkdir .3* - מאפשר ליצור תיקיה חדשה

- .4 - אפשר לעבור בין תקיות *cd*
- .5 - יוצרת קובץ חדש *touch*
- .6 - מאפשרת להעתיק קובץ *cp*
- .7 - מאפשרת למחוק קובץ *rm*
- .8 - מראה לנו את הפקודות שהרכינו עד כה *history*

:*Vim*

עורך טקסט שקיים בסביבת עבודה של לינוקס. השימוש בו נעשה באמצעות המקלדת בלבד, ללא העכבר. כל פעולה שאפשר לעשות על קובץ, קיים עבורה קיצור מקלדת כלשהו שמנע את השימוש בעכבר.
מודיע זה רלוונטי אילו? אם מתחברים מרוחק לשרת לינוקס כלשהו, למשל לשרת של המחלקה: אין לנו אפשרות *GUI* של פיתחה וסיגרת חלונות. אם רצתה>User קובץ לשרת, יוכל לפתח אותו עם *Vim*.
ישנה גרסה מתקדמת של *Vim*, *neoVim*.

:*MakeFile*

כל שמורכב מכמה וכמה חלקים וחוקים, שהמטרה של כל החוקים הללו יחד היא לוודא שכל הקבצים הרלוונטיים בפרויקט שלנו מעורבים ביצירת קובץ ההרצה. זה קובץ שמאוד נפוץ ביצירת קבצי *C/C++*: מרכיב מסווג של *rules* שכל אחד מרכיב משולשה אלמנטים:
.1 - שם *rule* (*rule*) קובץ שנוצר כתוצאה מהרצה של *command*
.2 - פקודת שתורץ במהלך הריצה של אותו *command*
.3 - קבצים לצרכים להיות קיימים בשביל *rule* של אותו *rule* ירץ.

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

2 הרצאה 2

2.1 עוז 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 ל2 כיון ש $-X = 2^n - X$. 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 האופrnd הראשון, ביטים של האופrnd השני ומחבר אותם מבליל דעת האם החיבור הוא *signed* או *unsigned*. חיברנו שני מספרים, כמו בדוגמה כאן מטה: כיצד נדע אם התוצאה שקיבלו נconaה או ששגניה?

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

ב-*unsigned* קיבל כי התוצאה 128, ב-*signed* נתקבל -128, מה קיבלנו כאן? ה-*CPU* בעצם לא יודע. בחישוב *unsigned* החישוב נכון - כי אכן $64 + 64 = 128$ שווה ל-128. נשים לב כי ב-*MSB* של שני המספרים היא 0, ולכן שני המספרים הינם חיוביים. וכך גם קיבל -128 – התוצאה שגوية.

כלל: *CPU* ביצע חיבור של ביטים, *the CPU לא מתעניין האם התוצאה היא signed או unsigned*. ישמר במקומות מסוימים האם הייתה החלפת סימן, אם לא מעוניין אתכם (אתם מחשבים *unsigned*) פשוט אל تستכלו במידע.

דוגמה: בדוגמה כאן מטה נוצר *overflow*, שכן קיבלנו אפס. אם היו ב-*signed* אכן קיבלנו תוצאה נכונה. אחרת, ב-*unsigned* זו תוצאה שגوية. היה *carry*. נשים לב כי אם היה חיבור של שני סימנים שונים - אפס ואחד, *the CPU* לא צריך לזכור החלפת סימן, הוא רק יזכור האם היה *carry* או שלא היה.

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

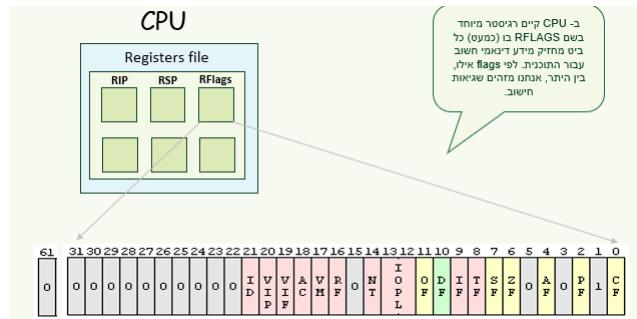
היכן ה-*CPU* רושם רישומים אלו?

שנו גנרטור מייחד שאר *CPU* שומר בשם *Rflags* - יש לו ביטים שמתחדכנים בעת חישוב שה-*CPU* עושה. הוא מדכן את הביטים תוך כדי החישובים. נשים לב שלכל בית ישנו שםcano. אנחנו מתעניינים בביטים הרצובים. ישנו ביט אחד אשר יהיה *carry* (בחיבור או בחישור) אחרת, **שנו ביט בשם CP(CarryFlag)** - קיבל אחד באשר יהיה *carry* (בחיבור או בחישור) אחרית, קיבל אפס. אמרו (אם וכאשר) כי חישוב *unsigned* שונה.

שנו ביט בשם OF(OverflowFlag) הוא קיבל 1 כאשר תהייה החלפת סימן (כלומר הביט שינה סימן), אחרת קיבל אפס. אמרו (אם וכאשר) כי חישוב *signed* שונה.

שנו ביט בשם ZF(ZeroFlag) - אם תוצאה החישוב האחרון יצאה אפס הוא קיבל אחד, אחרת הוא קיבל אפס.

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

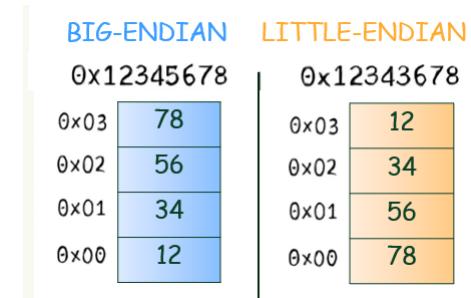


Endianness 2.3

ארQUITוRA של המחשב יכולה להיות *Big – Endian* או *Little Endian*. אם יש לנו ערך נומרי - מספרי, לא מוחזרות, והערך הנומרי הזה תופס יותר מ-1 byte אחד (כלומר לא *char* בלבד) וכן *float*, **מדוברים על ערכים שלמים בלבד**.

ב-*Big Endian* שומרים אותו משמאלי לימין (שומרים את ה-*MSB* הראשון – כלומר נשמר

במקום 00). בLittle Endian שומרים אותו מימין לשמאל (כיצד נזכיר? שומרים את ה-LSB הכי למטה - קלומר שומרים במקומות 00).



אנחנו בקורס לומדים לפי Little Endian. (באשר אנחנו מדפיסים בקורס משהו, זה מתבצע בהדפסה מ כתובת 00 קלומר מה שידפס קודם יהיה LSB).

Assembly 2.4

אסמבייל נוצרה בשביל לננות להפוך לכטוב בשפת מכונה, ולנסות לתרגם את השפה לשפת בני אדם. כשבתו את השפה לא חשב על נוחות. על כל פקודה מכונה, לקחו את הפוקודה ו"תרגמו" אותה לשפת אסמבייל שלcacura ויתר אוניות. ומכאן המסקנה: **פיעולה באסמבייל=פיעולה 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 במקומות?):**
- data types (כן יש - גדלים, 4 ביט, 1 ביט...)
 - מערכות (יכול לעבד ישרות עם RAM)
 - תנאים (גישה ישירה לרегистרים - זהה השפה היחידה שמאפשרת זאת, ולכן **אסמבייל היא השפה הפיעולה**).
 - ולואות
 - פונקציות
 - ספריות סטנדרטיות

מה שנקרה - בהצלחה תהיה לנו.

Registers 2.5

כפי שדרנו קודם לכן, בCPU ישנו איזור בשם registers file: **רץ' של 64 ביטים.** ישנו רגיסטרים נוספים שנדרשו בהם כתעת.

רגיסטרים ליחסוב כללי general purpose registers: הרגיסטרים הינם RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, R8, R9, ..., R15. נוכל להשתמש בהם לחישובים כלליים. הsticeה של R15 אין שמות הוא שמים נוספים בהרחבה של אסמבייל, שכבר הבינו שלשםות אין משמעות. בעבר היה חסר רגיסטרים ולכן בתוכך רגיסטר היו מאחסנים שני נתונים. למשל ברגיסטר ax היה שני תכני מידע שונים ah, al - ובהתקופה High, Low ah, al. בהמשך, הרחיבו את הרגיסטר ל-32 ביט ולא נתנו שם להמשך המידע וגם בהרחבה ל-64 ביט לא הוסיפו שם. כך נראה רגיסטר - אוסף בארכון של 64 ביטים.



שנים המון ורегистרים - בקורס אנחנו נשתמש ברגיסטרים שציינו לעיל, ברגיסטר *RIP* וברגיסטרים *Rflags* של *al*:
זכר ברגיסטרים הבאים:

RIP (Instruction Pointer). הוא הרגיסטר שהCPU (*Instruction Pointer*) RIP שמצויב על הבית הראשון של הפקודה הבאה לביצוע. *Loader*. שם בתוך הרגיסטר ממש את הכתובת הזו במוליך תהליך הפרסור. (הערה - גם באסמבלי נוכל לגשת לרגיסטרים בזיכרון, מה שאי אפשרות בשפות עלייתן).

RSP (Stack Pointer) - רגיסטר שנמצא בתוך CPU. מצבע על תחילתו (כיון שאם הערך גדול מדי, לפי מבוא הוא נשמר מלמעלה למטרת בזיכרון) של הערך האחרון שנכנס למחסנית. המחסנית חשובה מאוד כיון שיש בה משתנים לוקלים, *activation frame* וכותבת חזרה וכן ארגומנטרים שפונקציה מקבלת (אם וכאשר).

Basic Instructions & Data types 2.6

שורת קוד טיפוסית באסמבלי נראה כך: *movb \$0x61, al*. הפקודה הזו מכניסה את הערך 0x61 לתוך הרגיסטר *al*. לאחר הפקודה, הרגיסטר *al* יהיה כך:

$$al = 01100001$$

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

נשים לב: \$ חשוב מאד, ואם לא נכתבו אותו ייחסו שאנו מדברים על מקום בזכרון. אם כתוב \$ יהיה ברור כי הכוונה לערך. כשרנצה לפנות לרגיסטר - נעשה זאת עם % בפנייה לפני השם של הרגיסטר.

הפעולה mov: פקודת זהה. מוסיפים לסוף הפקודה סימות, בהתאם לגודל הטיפוס שאנוחנו מזיאים. אנחנו למשה ממציעים העתקה של ערך למקום אחר בזכרון.
 1. הפקודה *movb*: פעולה הזה שעובדת על bytes, מזיאים ביט 8 בתים.
 2. הפקודה *movw*: פעולה הזה שעובדת על words, מזיאים words מקום זיכרון.
 3. הפקודה *movl*: פעולה הזה שעובדת על long, מזיאים long מקום זיכרון. long הוא 4 בתים. (כמו int ב-C)
 4. הפקודה *movq*: פעולה הזה שעובדת על qword, qword הוא 8 בתים. (כמו Long ב-C)

2.6.1 השוואה לשפת מכונה

אותה דוגמה מלמעלה, בשפת מכונה תתרגם להיות השורה הבאה:

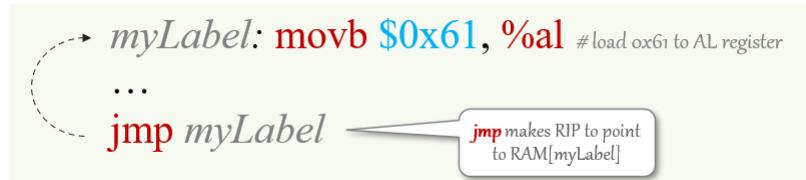
$$0xB061$$

זה הערך המספרי אותו צריך להכניס, 0xB061. מספר לו שהוא צריך לבצע הכנסה של ערך לתוך רגיסטר *al*. (מאיפה אנחנו יודעים זאת? נפתח ISA.).
Imm8 הוא ערך נומיри. *Imm* הוא ערך נומירי בגודל .8

jmp ו-*Labels* 2.6.2

בכדי לשים העורט על הקוד באסמלבי - נוכל להשתמש בנקודה פסיק או `#`. נוכל להוסיף לכל פקודה *Label*. זה מקום של הפקודה. כאשר נוסיף את הליבל, הקומפילר יתרגם אותו כתובות של הפקודה. **מאתוויי הקליעים**: הקומפילר מוחק את הליבל ומכניס את הכתובת. בשביל מה נctrar ל'יבלים? כתע, נוכל לבצע `jmp` ולהגע אל הפקודה זו, ממוקם אחר בקוד.

`RAM[Label]` משנה את רегистר *RIP* אל `jmp`



Assembler directive 2.6.3

הנעה שאנונו נותנים לקומפילר של אסמלבי, ששמו *.Assembler*, נקבע על הפקודה הבאה -

`buffer: .skip 4 # reserve 4 bytes`

קומפילר, תlk לזכור, ספציפית *section* של משתנים גלובליים, בפרט אל *bss* (המשתנה לא מאוחחל), ותקצתה לי שם רצף של 4 בתים. מעכשיו, נוכל להתייחס אל 4 הבטים הללו *buffer.chars*. זה כמוון מקביל ליצירת משתנה ללא אתחול, וכן מקביל למערך של *shorts*. וכן מקביל ליצירת מערך של שני *shorts*, או מערך של *char* - וכן כל קבוצה של משתנים שננסכמת לנודל של 4 בתים. כתע, 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 מתרגמים קוד בשפת *C* לאסמלבי: בתחילת אוננו מקרים בזcron 20 בתים (5 פעמיים 4 בתים של *int*), אנחנו מתחילה משתנה ברגיסטר *rbx* שיקבל את הערך אפס שיצין למעשה את *i*. משם אוננו מתחילה לולאה,

באסמלבי יש פקודת השוואת גנרייה: `cmpq`, באסמלבי יש פקודת השוואת אחת (<=) מה שהפעולה עשויה הוא לחשב את החיסור של *sorce* מה*destination*, כולם הפקודה הבאה תתרגם להיות

`cmpq $5,%rbx`

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

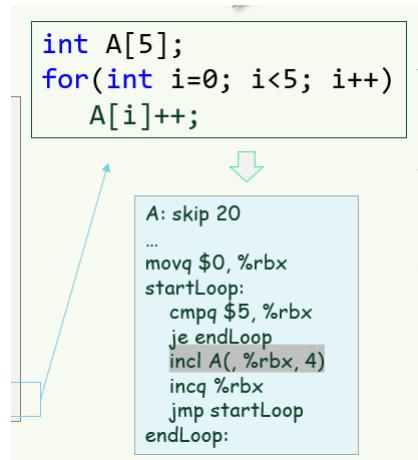
ZF

הפעולה `je`: אם יש שוויון כלומר $ZP = 1$, בין האופרנדים שבוצעו בשורת הקוד הקודמת, אנחנו נבצע `jmp` אל *Label* *je*. הפקודה הנ"ל ניגשת אל *zeroFlag* ובודקת, אם הוא 1 היא קופצת אל *Label*.

הפקודה `inc`: הפקודה למבצעת מבצעת *increasement*, יש *incl* שמבצע את על *long*. וכמוון כמוו ב*mov*, בהתאם לכל סוג משתנה. הסינטקס שלה יהיה כדלקמן

`incl A(%rbx,4)`

כאשר אנחנו כותבים סוגרים באסמבלי - אנחנו ניגשים אל האזכור. אנחנו הולכים אל RAM למקום שהוא $2048 + 4 * rbx$ או הכתובת של A . אנחנו למשה ניגשים אל הכתובת של $A[0]$ ואנחנו מגדילים אותה, בשביל להתקדם לכמות הבאה (שכבר שמרנו כשהקצנו 02 בתים). כאשר נבצע $incq %rbx$ 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 גליסטר. 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 גליסטר. סינטקטיס -
`subq %RBX,%RAX`

NOT: פעולה ביטויה. המשלים ל"0" - הופך בית 1 לbit 0 ובית 0 לbit 1. וסינטקטית -
`notb %AL`

NEG: ביטויה. פעולה המשלים ל"2", הופך את כל הביטים ומוסיף 1 לתוצאה. `notb %AL`

AND: פעולה ביטויה. בית במיקום i מקבל את הספרה 1 אם שני הביטים של שני הרגיסטרים הוא 1. אחרת, מקבל אפס. סינטקטית -
`andb %BL, %AL`

OR: פעולה ביטויה. בית במיקום i מקבל את הספרה 1 אם לפחות אחד מהbeitים של שני הרגיסטרים הוא 1. אחרת, מקבל אפס. סינטקטית -
`orb %BL, %AL`

`incq %RAX` : מגדיל את הערך, שקול לפעולת `++`. סינטקטית -

`decq %RAX` : מחסיר את הערך באחד, שקול לפעולת `--`. סינטקטית -

CMP : פקודת *CMP(Compare)* המשמשת להשוואה בין שני ערכים באסמלבי. הפקודה מבצעת חישור בין שני האופרנדים $A - B$ או $CMP(A, B)$ מחשב $A - B$ אך לא שומרת את התוצאה - היא רק מדכנת את דגלי הסטטוס(*flags*) כמו $ZeroFlag(ZF)$, $SignFlag(SF)$, $CarryFlag(CF)$, $OverflowFlag(OF)$. דגלי אלה משמשים את פקודות הקפיצה המותנית (*conditional jumps*) כמו JE , JNE , JG , JL ועוד. כדי לקבוע אם לבצע קפיצה או לא.

אם ההשוואה יוצאת שלילי, הפלאג SF מקבל את הסימן 1 ואז יודעים כי $A < B$, אם ייצא איזי $A = B$. אם קיבלו גם כי $1 > B$. אם קיבלו גם כי $1 < A$.

TEST : מבצעת פעולה *TEST* על שני אופרנדים אך לא שומרת את התוצאה - רק מדכנת את דגלי הסטטוס. שימוש נפוץ לבדיקה - האם רגיסטר שווה לאפס:

`TEST R1, R1 ;`

`JZ label ;`

למעשה ישנה פעולה *TEST* על אותו אופרנד עם עצמו. אם הוא היה אפס, נקבל כי ZP כתעתיד. ולכן. וכך קפוץ *Label* אם $R1 = 0$.

SHIFT : פקודות הזהה של ביטים. בביצוע SHL כל בית זו שמאליה, ביטים שיוצאים מהמקום נזרקים ונשמרים ב- CF , מימין נכנסים אפסים. בביצוע SHR כל בית זו ימינה, ביטים שיוצאים מהמקום נזרקים ונשמרים ב- CF , משמאלי נכנסים אפסים.

SAL, SAR - שיפט אריתמטי, בשיפט רגיל אנחנו מזיאים ומוסיפים אפסים. יתכן כי המספר (4) 0100 יקבל שיפט לשמאלי, וכותזאה מכח יהפוך למספר 1000, שהוא מייצג מספר שלילי (-16). והרי זה לא הגיוני שהילכנו בשתיים מס' חיובי וקיבלו מס' שלילי. בדיקוזו הסיבה ששנתחמש בשיפט אריתמטי -

SAL זהה לחולוטן SHL . עם זאת, SAR מזיא לימין את הביטים, אך הוא משאיר את בית הסימן בצד שמאל. ככלומר: הוא לא מזיא את בית הסימן.

דוגמה - 1101 אם נבצע עלייו SHR נקבל 0100, ממש מאלי קיבלו חיובי, זה לא טוב. לעומת זאת אם נעשה SAR נקבל 1100 (הסימן נשאר).

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

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

2.8 תרגול 2

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

`xorq %rbx,%rbx`

זהו גלגול לאפס את הרגיסטרים בתחילת הריצה.

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

$i \in \{1, 2, 4, 8\}$ החישוב 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 לה *Addressing Modes* ומחברים לפי החישוב לעיל. ניתן להשミニ חלק מהפיסיקים, לא חובה שכולם ישתתפו.

נשים לב - ניתן להבהיר *eax* (32 ביט) לתוכה *rax* למשל (64 ביט), תשמור בחלק התחתון של הרגיסטר ובחלק העליון יהיה אブル. לעומת זאת אם נעשה *rax, rax* זה ישמור בחלק התחתון ואפס את החלק העליון.

הפקודה *mov* יכולה להבהיר מידע מרגיסטר לרוגיסטר, ומרגיסטר לאזכרונו. לא יתכן מצב בו מעבירים מזכרו לזכרו בשורה אחת - לא יתכן:

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

מה הפתרון? נשתמש ברוגיסטר עאר, וזה יוכל להבהיר בין כתובות.

הפקודה *movzbl, movsbl* משמשת ל*mov* שכדי להכיר. $Z = zero, S = sign$. $Z = zero$ שאמם יהיה בסוף הפקודה *s*, ואנחנו מעבירים למשל *al, %edx* אשר נמלא את שאר המיקום שלא הتمלא (כי *MSB* ב*sign* של *al*, ככלומר אם הסימן היה אחד נוסף אחדות עד *MSB*). אם בסוף הפקודה יהיה *z*, המשמעות שנמלא את שאר המיקום שלא הتمלא באפסים.

: *Branches* ישנו רוגיסטר *Rflags* בגודל 64 ביט, שמחזק *flags* שונים. אין לנו דרך לשנות אותו. אנחנו מעוניינים רק לקבל את ערכי הדגלים שלו לאחר פעולות אריתמטיות.

Singed&Unsigned 2.8.1

ישנם שני דרכים שוניםלי ליציג מספרים. בשתי השיטות משתמשים בייצוג בינארי - אך מפרשים אותו אחרת.
(המשלים 2): מפרש את הביט השמאלי ביותר, *MSB* כבית סימן. 0 משמע חיובי ו-1 משמע שלילי. טווח הערכים יכול לנوع בין $-2^{n-1} - 2^{n-1} \rightarrow 2^{n-1}$. *Overflow, Underflow*: *Overflow* הוא מצב של הסימן עלול להתחפה, ונקלט תוצאה לא צפופה.

```

        · 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)

```

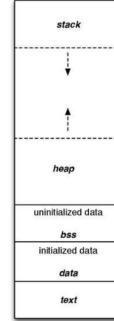
אנחנו קובעים את המשמעות של הרגיסטר - האם אני משתמש בו כתובות או כערוך. אני מחליט מה רגיסטר מחזיק.

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 התהיליך.

ישנם כמה חלקים בזכרון:
 - *text* - הקוד עצמו, ההוראות שהמעבד מרים, כל הפונקציות שכתבנו בקוד והקריאהות. 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, כתובות לא מיושרת עלולה לגרום לתקלה.

3 הרצתה 3

3.1 Jump&Set

קפיצה מותנית נעשית בהתאם לערכי *flags*

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

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

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

תרגומם להivot:

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

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

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

טבלת פעולות :*set*

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 ערכאים 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* ומכניםים 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 שnitן לבצע במחשב שלו - יותר מפעולות bitwise**. מדוע? החישוב מאחוריו הקלים משתמש בחומרה מיוחדת שמייעלת את החישוב.

```

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

```

מה קורה כאן בדוגמה? ראשית אנחנו מחשבים בשורה הראשונה את $3\%rdi$. ומכניםים זאת rax . לאחר מכן, אנחנו משתמשים בפקודה בשם *:salq* - שיפט אריתמטי - ומכפילים ב 2^2 , שזה בדיק $i \in \{1, 2, 4, 8\}$. שולש כפול 4, זה אכן 12. נזכר כי הארגומנט הימני ב-*lea* מאפשר $t \leftarrow x + x * i$. וכך בדיק $i \in \{1, 2, 4, 8\}$ יוכל אי אפשר לטעון שנבצע משוחה כמו $(\%rdi, \%rdi, 11)$ - זה יכול להיות נחמד אך לא עובד.

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

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

C Calling Convention 3.4

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

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

Stack Operation 3.4.1

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

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

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

Calling Convention 3.5

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

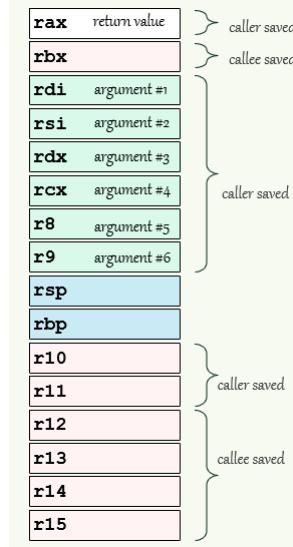
הערך המוחזר תמיד דרך *rax*. מכאן נשאלת השאלה - ומה עם גדלים שגדולים מ-8 בייטים? מאחרי הקלעים הקומפיילר מתרגם את הגודל הזה כאילו הוא מוחזר פוינטר (אכן פוינטר בגודל 8 בתים) אל הערך זהה.

16 הרגיסטרים מתחלקיים לשני קבוצות

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

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

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



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

תהליך הקוריאה לפונקציה:

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

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

בעת שנכנסים אל הפונקציה, אנחנו צריכים לדוחו למחסנית את הערך הנוכחי של הרגיסטר `rbp`. דבר נוסף, זה לחתת את הערך של `rbp` ולהכניס לו את הערך של `rsp`. כלומר, `rbp` מציביע לאותו מקום שמציביע עליו `rsp`. באסמליל 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

```

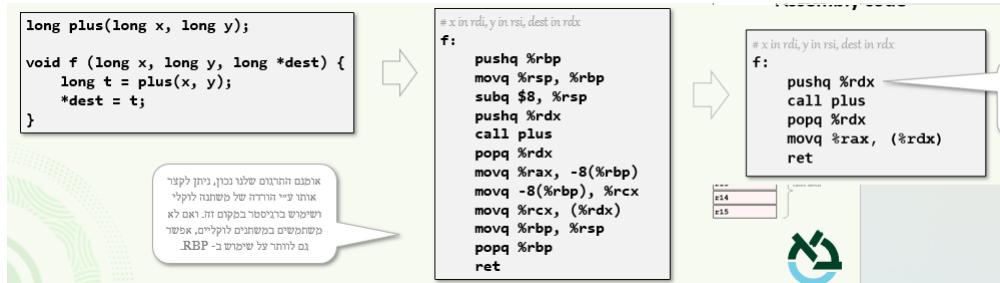
87

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

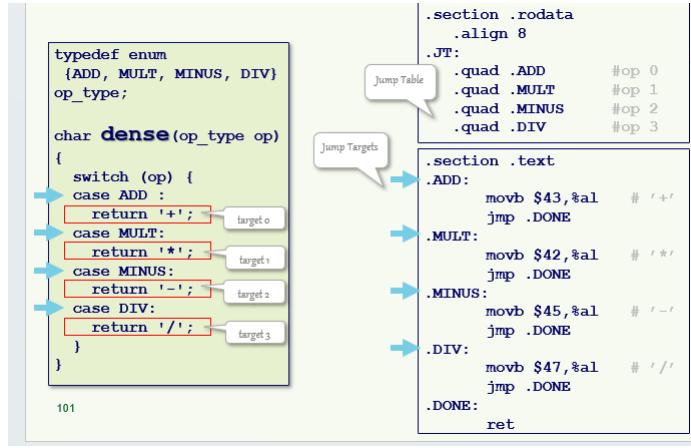
מה נעשה עם פונקציה שמקבלת יותר מושיעת ארגומנטים? נכניס את השיטה הראשונית לרегистרים המותאימים ואז את השאר נדחוף למוחשנית בסדר הפוך. ככלומר אם יש 10 ארגומנטים נכניס את השיטה לרегистרים ואז נדחוף את 10 למוחשנית, 9 8 7 וזהו. הרעיון הוא שהכי יהיה לי קל לגשת אל המשתנה השביעי.

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 החרצאות.

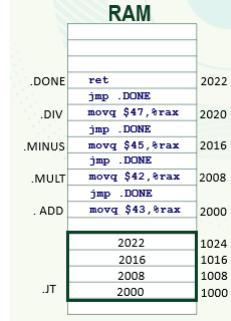
Jump Table 3.6

בשפת *C* קיים מבנה של `switch&case`. כיצד נממש מבנה זה באסמלבי? וראה כי במקומות לכתוב סוייך' אנד קיס ניתן להמירו `if&else`. עם זאת - 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 (מלטפה) יופיע הכתובת של `case1`. `jmp RAM[JT[op]]` נשלח אל המיקום המדויק? `case2`. וכך הלאה בשורה השנייה הכתובת של `case2`.



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



כמו כן, נזכיר את הפונקציה `CD`

```

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

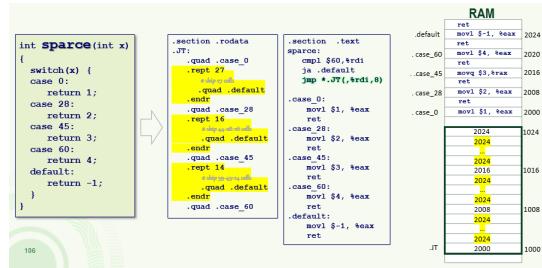
```

כלומר, אם החישוב הינו חוקי נקפוץ לippet המותאים. נראה כי החישוב למשל עבור יתקיים כי הכתובת שנקלט בקפיצה תהיה

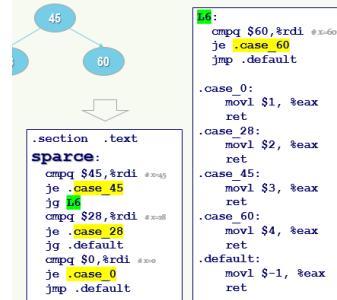
$$2 \times 8 + 1000 = 1016$$

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

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



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



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

GDB 3.7

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

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

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

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

הרצאה 4

Assemble process 4.1

נרצה לבצע את שלב הקומפול, המרת הקוד לשפת מכונה.

קובץ שהנקרא *listing file* יכול לתת לנו, בתוך כל קובץ זה יש את קוד המקור (צד ימין), העמודה האמצעית היא תרגום לשפת מכונה והעמודה השמאלית היא הגדרת ה"מייקום" היחסית של כל פקודה ביחס לsection שהיא 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 קומpileציה, אם הוא ערך נומירי הוא מתרגם לפי *little endian* ואם הוא string אז הוא מתרגם לפי טבלת ASCII.

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

Symbol Table: הקומפילר מכניס שם מידע שהוא מהה. בעת קומפול הקוד לעמלה, אנחנו נתונים בסשן *rodata* אנחנו מכניס אותו לטבלה. אנחנו נגדיר שהערך שלו בתחילת (*info*) יהיה לאחר מכן, הקומפילר נתקל בשם חדש: *str*. היא מכнесת גם כן בשורה חדשה 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, הקומפילר צריך לעדכן בעת כניסה *str* את הגודל של סשן *rodata*, כי התווסף גודל חדש לסשן. ולכן: הקומפילר מעדכן בטבלה את *info* שלו להיות יותר מוקם.

כך נראה הטעינה -

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

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

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

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

בעת הגעה לשורה של `leaq`

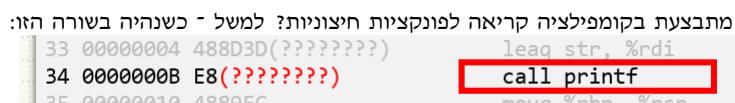
 32 00000001 4889E5
 33 00000004 488D3D(????????)
 34 0000000B E8(????????)

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

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

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

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

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

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

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

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

ELF relocatable format 4.2

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

```
$ readelf -a 1.0
ELF Header:
  Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF64
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  Type: REL (Relocatable file)
  Machine: Advanced Micro Devices X86-64
  Version: 0x1
  Entry point address: 0x0
  Start of program headers: 0 (bytes into file)
  Size of this header: 64 (bytes)
...
Section headers:
[Nr] Name     Type      Address  Offset   Size    EntSize Flags Link Info Align
[ 0] .null    NULL      0        0       0       0       0      0  0   0   0
[ 1] .text    PROGBITS 0        40      17      0       AX    0  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   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  0   1
```

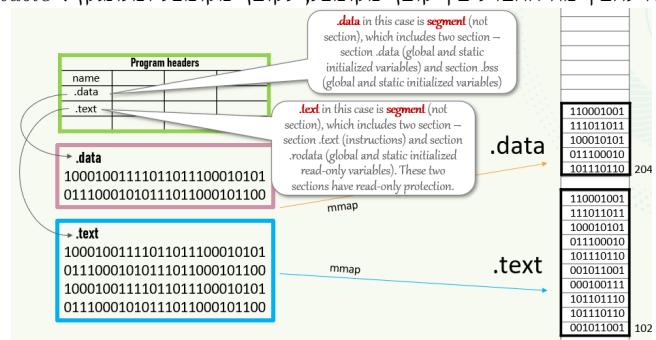
readelf utility allows to observe structure of ELF file (object or executable)

זה מידע שהקומפיילר כותב שיכול לעניין מאוד את *Linker* או *Loader* בשרה הראשונה מופיע *ELF*: Magic מספרים, מודובר *ELF64*, מען *Loader*, מידע 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*, שעוד נדרש לחבר עם קודים שונים (*extern*).
sections כל הנקורים ישים שני סקשנים, הקומפיילר טוען כי למורות שקודם הנקורים ישים שני סקשנים, הקומפיילר טוען שיש.

תחילתו הוא טוען שיש *text*: אותו אחד שאנו צרכנו קודם קודם, אך בגודל של 17. כמו כן, אנו אומרים לו שה*offset* שלו 40. בקובץ המקומפל - הוא מתחילה בשורה 40. לאחר מכן אנו מולמים שאת הטבלאות שקדם ראיינו, הקומפיילר שומר בתוך *section symbol* זה *relocationTable* *rel.text* שבסביבו *syntab* זו טבלת *symbol* *relocationTable* *rel.text* - שאינם בחומר הקורס.

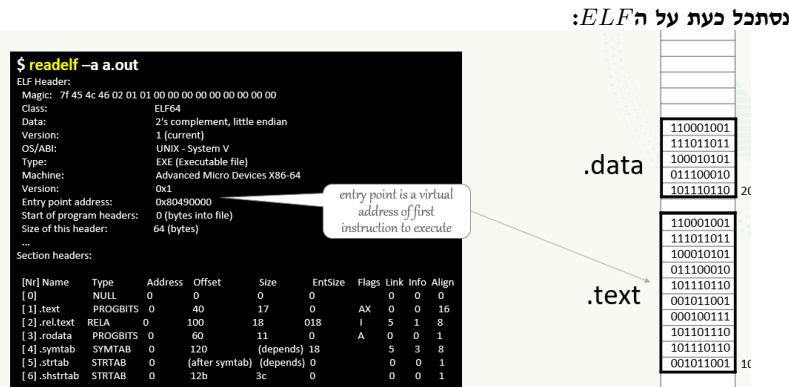
ELF executable format 4.3

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



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', היה אפס. כאן נראה שיש שינוי גדוֹל, זה לא אפס. כתעת 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 המוחשב, ויתכן שתהילך אחר תפס את הכתובת הזה ושמו את הכתובת הזה במקומות אחר, 300 נניח. מה נעשה? מערכת הפעלה מייצרת עצמה מיפוי, ובכתובות 100 המערכת כתובות עצמה: בתהליך אם *Loader* יבקש מהה, תפנה אותה ל-300. **מסקנה:** הכתובות הן וירטואליות ואינן באמות אמיתיות. על זאת ועוד נרחב 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 שkompiiler-linker חישב אותן והරיצה להשתמש בה אילו היא הייתה פניה ב-*RAM*. ויש **כתובת relative** - הכתובת ביחס למיקום הפקודה בסגמנט כלשהו.

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

ישנן פקודות כמו *call*, *jmp* שהן **RELATIVE** – כלומר תבע חישוב *independent* (רלטיביות) – כתובות וויטואליות *RelocationTable* (*position*). לכן בטבלה של *RelocationTable* על פקודות אלו הוא ישם *ABS* (absolute). לעומת כתובות *ABS* (אבסולוטית) – למשל כמו *%rdi* ו- *str(%rip)*.

4.3.1 כיצד כתובים וירוס?

אנחנו רוצים להוסיף לקובץ ההרצה שלכם, קטע קוד שאין יצרתי, שאם תרצוו 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 לתוכך קובץ. ווסף אל הקובץ את הקובץ המקורי המקומופל שלו, ושמורו את הקובץ. כאשר נריץ את קובץ ההרצה הזאת, נריץ גם (אולי?) את הקובץ שאינו הוסיףתי לך. נניח שהקובץ שאנו הוסיףתי מוחק את כל *file system*. נשאלות כמה שאלות.

אם *Loader* יפרסר זאת? בעת *linker* שיבא את הגודל הסופי של סקשן *text*, הם בודאות לא התחשבו בקוד שאין הוסיףתי אל קובץ ההרצה – קוד זה לא היה קיים לא בזמן הריצה.

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

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

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

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

Disassembled 4.4

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

Linking process 4.5

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

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

כללי :Linker

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

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

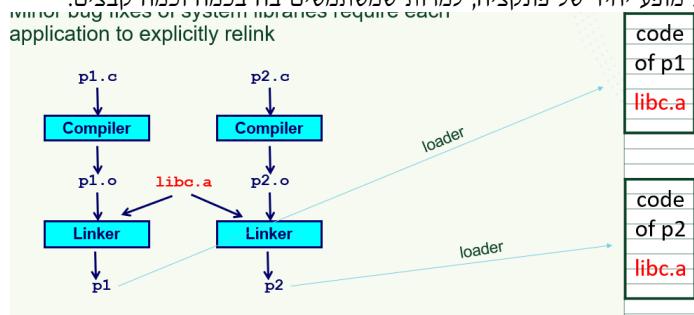
3. אם יש לנו שני *weak symbol*, הלינקר יכול לבחור אחד מהם שרירותי.

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

באמצעות linking' אפשר לבנות גם ספרייה עצמאית: כיצד? נכח למשל פונקציה *printf.c* ונממש אותה, נקمل ונקבל קובץ הרצה. כמו כן נעשה זאת על עוד פונקציות כמו *atoi.c* וכו'. נקבל הרבה קבצי הרצה וננלק' אותם יחד בammedoot 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 לבצע 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 בזכרו. מה נעשה? אם אנחנו יודעים שיש לנו פונקציה כמו *printf* שמשופיעה הרבה פעמים, נשים אותה בתוך **ספריה דינמית**: אומרים למשה *Loader* - אתה לא מעלה לזכרו את שני המפעעים של *printf*. אבל, כשאתה רואה *printf* אתה הולך בספריה הדינמית, מושך משם את הפונקציה וכותזאה מכך אתה מקבל מפעע 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 הקומPILEZA / linking'. אם gcc החלט שסקשן *text* מתחילה בכתובת 1024 או 1024 הוא המיקום ההתחלתי של הסקשן.

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

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

סיבות לשימוש *position independent*:

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

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

4.7 תרגול

4.7.1 ארגומנטים ב-STACK

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

4.7.2 Variadic Functions

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

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

```

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

```

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

4.7.3 קבלת ארגומנטים בשורת הרצאה

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

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

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

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

Flow Control 4.7.4

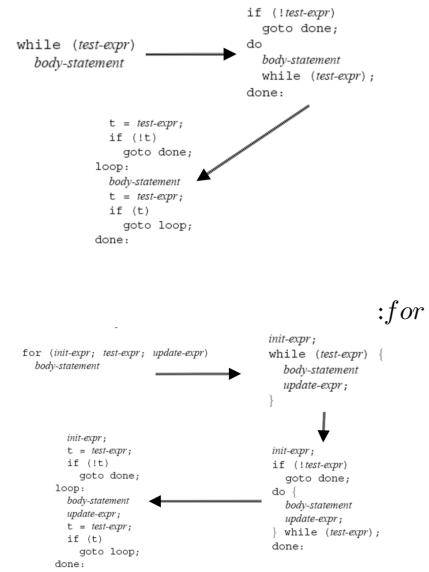
כל מבנה של תנאי, לולה וצדומה ניתן להמיר מ-C לאסמבלי. נראה מספר דוגמאות.
:if&else

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

:do while

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

:while



הרצאה 5 - memory hierarchy - 5

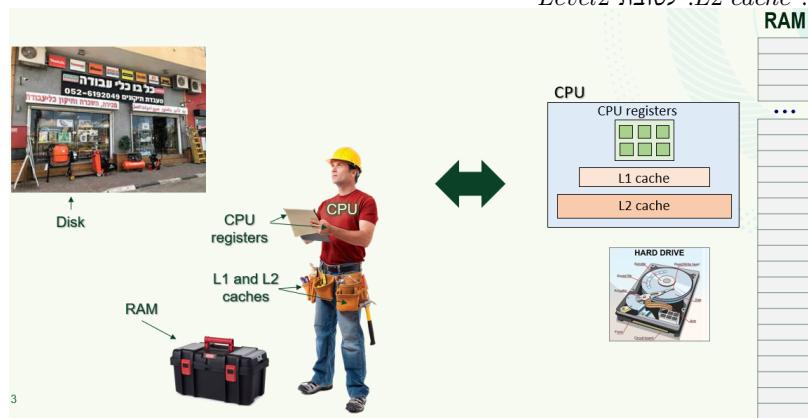
5.1 הקדמה

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

:*CPU* נספחים בתוך *the CPU*

Level1 : לטובת *L1 cache* .1

Level2 : לטובת *L2 cache* .2



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

cache 5.2

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

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

גם *L2* הרבה יותר מהיוקם 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. ברמות החומרה - אוטומטית, המעבד יורד למיטה בזיכרון לכתובה הנומוכה ביותר שקרובה אל *X* שמתחלקת ב-16: זו 1024, ומשם הוא ילוך 64 ביטים ומוכנס 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 שסוכמת 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 סידורתי, 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 סידורתי? 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 מה שנותה באיזו שורה נבייא את *x* אליה? הרי אנחנו לא רוצים לעבור בכל השורות. החיפוש ב-*cache* הוא עיליל.). אחרת, נחפש ב-*L2*, אם נמצא נבייא את *x* לרגיסטר ב-*CPU* וכן אנחנו נקדם את *x* אל *L1* (מדוע?). בסביבות גבולה כמו שאמרנו נשתמש בו שוב ונרצה פעם הבאה לגשת אליו מהר יותר). אם לא נמצא שם - איזי נלך לחפש ב-*RAM*.

נשאלת השאלה - אם הוא בכלל הtgtlaה ב-RAM, לא חבל על הזמן שbezינו בחיפוש ב-L1, L2? אנחנו מניחים שהקוד הוא cache friendly - קוד שהמתכונת שכותב אותו מודע ליתרונות של cache ולכן בסיכוי גבוה מאוד x יתגלה ב-L1, L2. אם לא: תכון, אך הסבירות לכך נמוכה ולכן בטוחה הרחוק זה אכן משתלב.

איך מגיעים אל L2? תמיד מנסים להכניס אל L1, אם אין שם מקום אנחנו מכניסים 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, ולכן הסבירות שנרצהשוב 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 מזיהה כיוון שאנו מדברים על תוכנות סדרתיות. עם זאת, כאשרначיל לכתוב תוכנות מקביליות זה יהפוך לבעיה קריטית ויעשה את ההבדל בין תוכניתנו נכונה לשגوية. נדע זאת כרגע - ונאפשר את זה. בקורסים עתידיים - נדע כיצד פתרים זאת.

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

מבחן גדים:

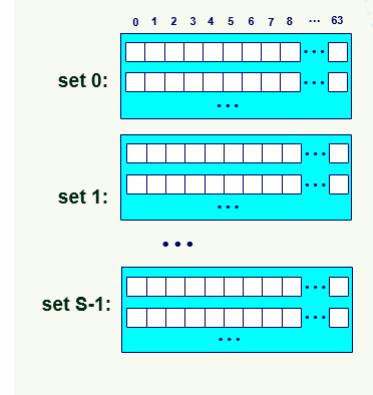
אם מביאים קוד שהוא נכנס אל L1 instruction cache שהוא ב佗וה של 16 – 128(KB)

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

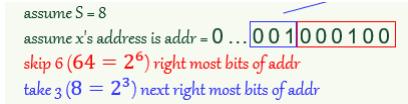
128(KB) – 8(MB) הוא בגודל (L2 cache

organization of a cache Memory 5.3

מבחן גדים cacheן מוחולק לש' set. כמה? תלי בכמה היכרן הגדייר. ראיינו כי אנו יודעים את הגודל של L1 ולכן אנחנו 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.



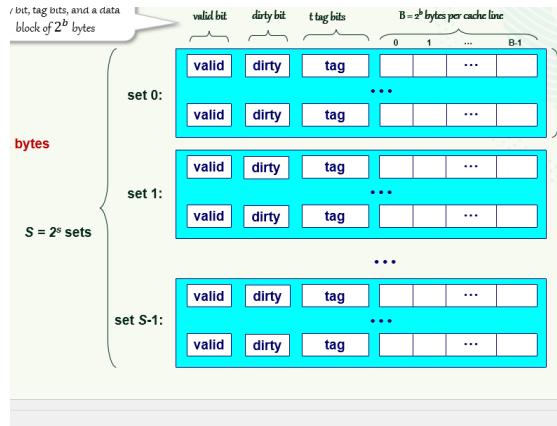
נניח שהוכנסנו את x , כיצד ה-*cpu* ידע באיזה set נמצא x ומה המיקום היחסית שלו באותה מחרשה? אם מס' מתחולק ב-6 לא שארית בהכרח 6 ביטים אחרונים שלו הם אפסים. אנחנו נסתכל על כתובות שמתחלקות ב-64bits והכי קרובות לאיקס הנוכחי, כפי שאמרנו קודם ונקה 6 ביטים. נkeh את 6 הספרות האחרונות של כל כתובות והם ייצגו את המיקום היחסית של הכתובת בתוך cacheLine. כמו כן: נkeh את \log_2 באשר x הוא מס' הקבוצות: נkeh את \log_2 הספרות הבאות אחרי 6 הספרות והם ייצגו באיזה קבוצה אנחנו נמצאים מבין x הקבוצות שכן מספיקים \log_2 ספרות ליציג x מספרים. ס"כ בז"כ תיוזג כל כתובות ב- set 's שבסותה x ב-64bits הם בכתובות 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 קודם לבן 6 ועוד 3 ביטים, אך נשארו $55 - 9 = 46$ ביטים נוספים, אנחנו נרצה להכניס אותם אל *CPUs.tag*. רוצה לדעתה מהילא *cacheLine* מכיל באמצעות את x ולכן הוא משתמש ב-*tag*, שכן יתכו כתובות שונות עם אותה סימות של 9 ביטים. כשה-*cpu* מוחפש את x (cacheLine) 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.

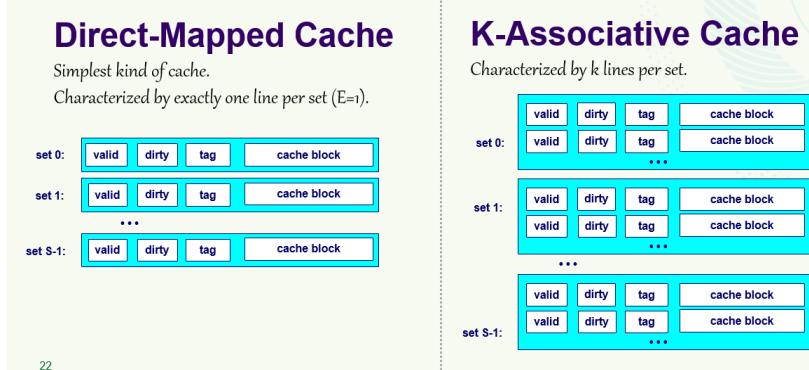
dirty bit: האם *cacheLine* הוא *dirty* או לא. מודיע זה חשוב לדעת? *Ram*. מודיע "פטור" *dirty bits* אנחנו חיבים לגשת אל *Ram* ולעדכן את הערך של הביט*i* אחרית - אנחנו נאבד אותו (הוא השתנה ונחננו לא ידועים זאת בזיכרון). ואנו מוחקים מה-*cache*. סכנה לאבד אותו (לצח).

valid bit: האם מותר להשתמש בנתון שישוב *cacheLine* או שאסור. למה שיהיה אמור להשתמש? כל עוד אנו בעולם סיידרתי, אין סיבה שהוא אסור להשתמש וכל השורות הין. כשנחננו עוברים לעולם מקבילי או בעולם שיש בו כמה *process* - נהיה בעיה. לכן כרגע, כשנחננו בעולם סיידרתי: *column valid*. במקבילות - באשר עוברים בין *process* שונים 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*'s. 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):

לזמן *least recently used*: *LRU* - מפנים את השורה שלא נגעו בה הכי הרבה זמן, ווקבים אחר זמן הגישה האחרון לכל שורה ואת היכי רחוק מוצאים.

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? מתחילהם ב-*main*.

א. השלב הראשון הוא *reprocessing* שבמסגרתו כל *define* ו-*include* שיש לנו בקובץ *main.i* מוחלים בערכם המקורי. מתהלך זה יוצא 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 אליו.

אנחנו ניגשים לבין 의미ו של *פיזית* (שמופיע בספריה).

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

א. *static*: ספרייה *static* שמופעל אליה - אם אנו ניגשים ל- *symbol* מסויים במהלך *scan* ממש יוכנס אל קובץ הרצה שלו 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 כלל. ככלומר בשבייל שהקוד של הפונקציה יהיהeki 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* וכשרירץ את קובץ הרצה יהיה בזיכרון 10 מופעים של הפונקציה, חביל על המיקום.

ב. *dynamic linking*: ספרייה *dynamic* שמופעל אליה . לכל פונקציה *dynaminc linking* . או ספרייה בעולם קיים עותק יחיד בזיכרון,

הרצת פשוט לגשת אליו באשר אנו מניחים לכל פונקציה יש צ'אנק כלשהו בזכרון. יש בעיות בכך - אף אחד לא אמר שככל רגע נתנו לנו מושגים 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 בייטים, אבל מתחת צ'אנק של *printf*

הנוכחי יש קוד וגם ב'אנק מעליו יש קוד וכן 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 שנראתה *symbol* בעת הריצת התוכנית, ואז *linker* יטען את *symbol* הנוכחית בזיכרון.

בעת **נשאלה הבאה**: האם עדיף *load time* או *run time*? אם למשל הקוד שלו מושתמש בהמון פונקציות, אז בغالל תנאי *if* מזמן רק 3 פונקציות. במקרה זה עדיף לנו כמובן *runTime*. לכן: ב-99% מהמקרים עדיף *runTime*, אז 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 שהוא בין *heap* ל-*stack*. נשים לב כי יתכן וכתבתטי קוד שמשתמש ב-*scanf* שעוזר לא היזהה טעונה בזיכרון, لكن 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*'שאנו מעבירים ניתן לקבוע איזה סוג *linking* נקבע וכן האם *loadTime* או *runTime* הוא *Lazy bidding*. דיפולט זה נקרא *dynamic default*.

(Position Independent Code) PIC 5.5.2

כפי שראינו, רצחה שהפונקציות קודם יוכל להטען לכל מקום שהוא בזיכרון. נססה להבין איזה טריקיים ושיקיים הקומpileר מבצע כשהוא רואה קובץ *C* עד שהוא הופך אותו לקוד PIC באסמבלי:

מחלק טריקיים אלו לשניים. א. קוד שנגish *symbol* פנימיים בלבד (וק *לפונקציות או משתנים גלובליים או סטטיים שהוגדרו באותו קובץ בלבד*):

ב. קוד שנגish *symbol* חיצוניים (יתכן גם פנימיים): כבר בזמן קומPILEציה אנו יודעים שיש לנו *symbol* *symbol* – *relative addressing* של *symbol* לא באמצעות ה абсолютית של *symbol*. 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 מרוחק קבוע בזיכרון. לכן נניח וישנה *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 ומחרס את המיקום של *text*. זה הפרש בין המיקומים וכן נדע לעבור בניתם.

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

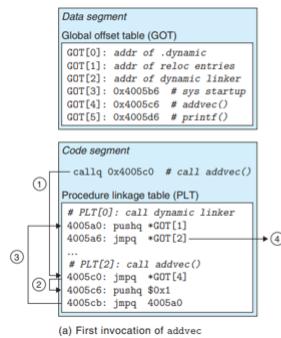
קומpileציה נוצרים ב-GOT סימboleים פנימיים גם - וכן אפשר לשאת אליהם באמצעות צורה בדיק (אם כי אכן אין צורך לרשימת GOT עבור סימboleים פנימיים, אך זה קורה).

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.

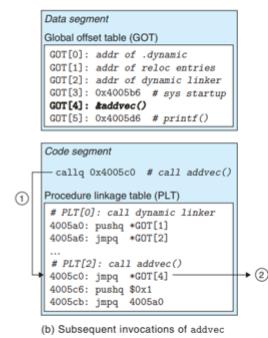
נראה כי יש הבדל ב-PLT בפעם הראשונה שטוענים פונקציה בין פעמים אחרות. באשר מביצעים קרייה לפונקציה advac קוראים לערכה ב-Got:

First invocation of advac



(a) First invocation of advac

Subsequent invocations of advac



(b) Subsequent invocations of advac

נראה כי אם הלאנו לרשימה GOT של advac עוד לפני שטענו אותה לאירועו, מתבצע צד שמאל. נראה כי לאחר שטענו אותה לאירועו, במקום GOT[4] אכן הינו dynamic linker' לשם המשם את הכתובת של advac שכך נטעה. נראה כי תמיד בעת שתקרה לאויה 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 שיצרך 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 וכותזאה מכך הוא יתנהג אחרת. (להזכיר בדוגמה עם הסטודנטים במובוא שיצור קובץ חשוב - הרשו אותו כי הם לא חכמים במיוחד, ואניicut צריך לקחת את הקובץ שהם נתנו לי ולסדר אותו.)

6 הרצתה 6

miss 6.1

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

א. *miss*: *cold miss* שטוען שהוא לא הכיל 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 ואפשר לטפל בו - בהמשך נדבר האם הקוד אז.cn אפשר לטפל בזה.**

ב. *conflict miss*: יש לנו מצב של דרישת - מבאים בлок מהזיכרון אל *cache* והוא דורש מידע אחר. באשר *block* מזכיר מומפה למקום תפוס *cache* וטופס 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 שייפויו אחד אחרי השני בקוד ייפויו 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* **ουמס אדר' של כל הקוד במסויים.** 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**.

לcpu יש שתי אפשרויות: אחת *write - through* היא לעדכן שירותי *RAM* או לחlopen *RAM* לא לעדכן את *RAM* כרגע, לשם כך נשתמש ב-*dirty bit* ונדען 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 *cache* לגישות עתידיות.

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

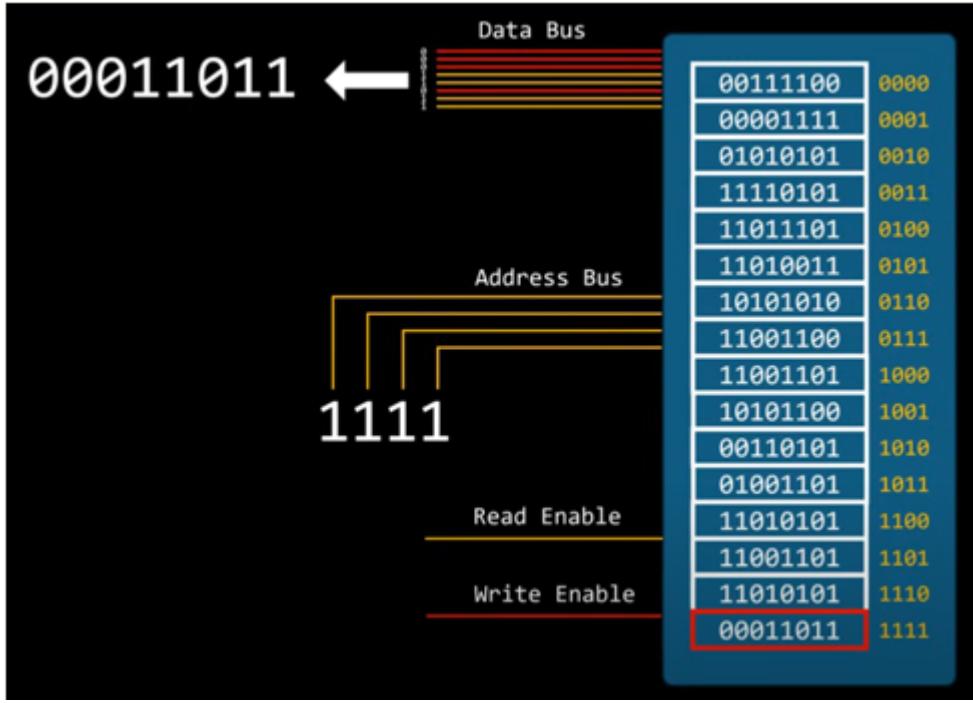
לרוב - נרצה write back וכן write back

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

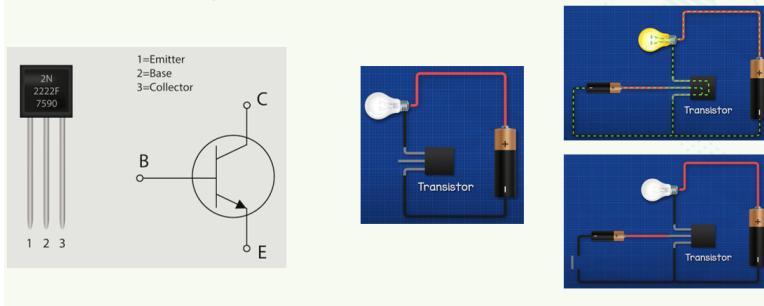
הגדרה: *Hit Time*. כמה זמן לוקח לנו לחלץ את הנתון מה-*cache*. בדרך כלל: 4 ננו שניות "ב- L_1 " ו- 10 ננו שניות "ב- L_2 ". אם כן, זה עדיף על גישה לאזכיר שעולתה 50 – 200 ננו שניות."

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.

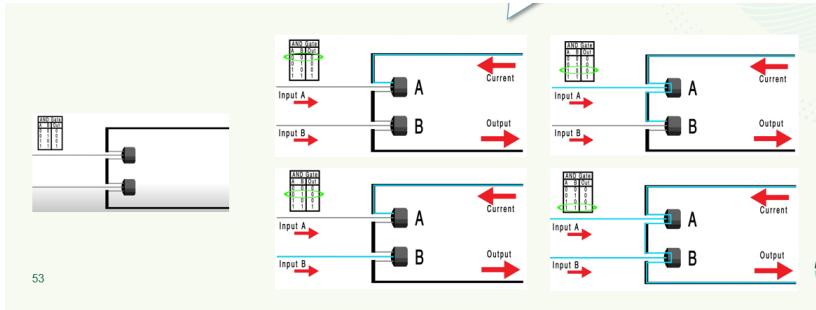


טרנזיסטור הוא רכיב חשמלי שמתפרק כמתג חשמלי או מגבר - הוא יכול להעביר או לחסום זרם חשמלי בהתאם למתח שモפעל עליו, והוא אבן הבניין הבסיסית של כל המעבדים והמעגלים הדיגיטליים המודרניים. כפי שראים בדוגמה מטה - לצורך הבדיקה את הנורה צריך לספק לטרנזיסטור חשמל (סוללה כלשהי למשל). השן האמצעית של הטרנזיסטור קובע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

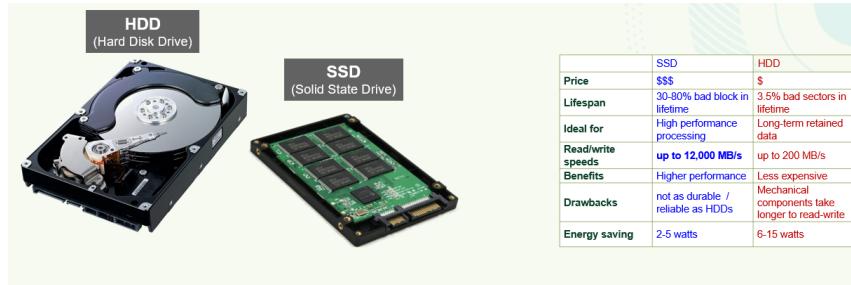
נדבר על שני סוגי של *RAM*: *S – RAM* ו-*ram*. *S – RAM* החשמל מייצג (כל בית) באמצעות 6 טרנזיסטורים - מדוע? החשמל נתוח להחלש וomers מקטינים את הרידיה בחשמל. (איך? לא רלוונטי לקוטס). כתוצאה לכך הוא עולה הרבה יותר - כי משתמשים פי 3 בטרנזיסטורים מאשר *D – RAM*. הוא גם יותר גדול פיזית - הנפח שלו יותר גדול, 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 מטכנולוגיה 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*.

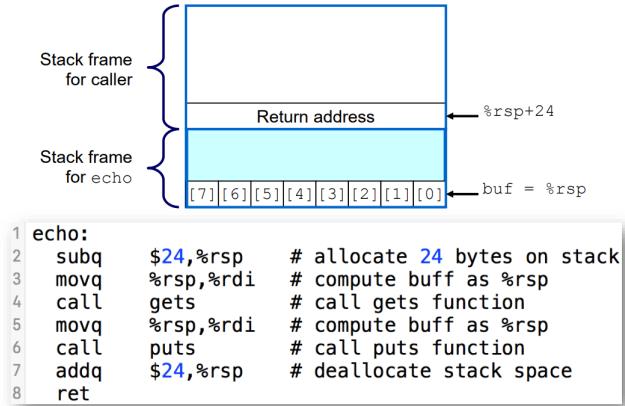
6.4 תרגול 6

בוגד *stack buffer overflow*: ב-*stack buffer overflow* מושרים מידע וערך חזרה לפונקציה. נראה כי ניתן ותיה זליגה מהמקום שהוקצה. למשל בפונקציה הבאה - הקצתו ב-8 בתים, נניח וה-*input* שהכנסתי אל בוגד 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 שיקירה 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* וכיו? 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 $missRate = \frac{1}{4} \times 100 = 25\%$. אם הקוד עבר על המטריצה לפי עמודות - *Miss rate* הוא 100%, אנחנו בכל פעם נביא שורה, וכיון ש- N גדול מאוד, כל אלמנט נמצא *cache line* אחר. לכן בזדאות תמיד נאלץ ללקת לאחרו.

ומה באשר לפביל מטיריות? נשים לב שאלבורי, כפלי של שתי מטריצות, הוא לקיחת שורה במטריצה A , ולקחת עמודה במטריצה B , להכפיל אותן ולקבל איבר בודד במטריצת המכפלה. נשים לב שאנו מוחים בבעיה - מבחינת *cache friendly* אנו 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?

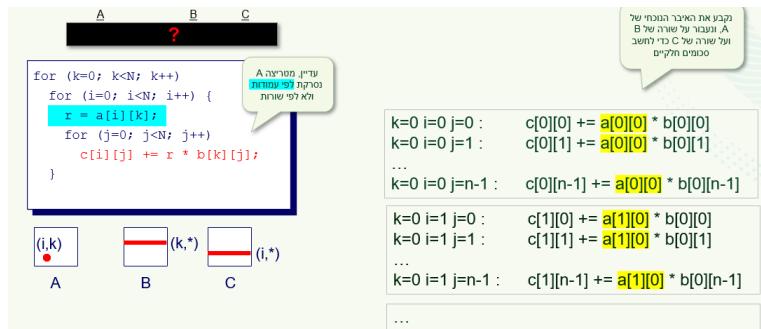
עבור A - בדיק 25%, כמו קודם.

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 נגיד שהוא 0%.

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

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

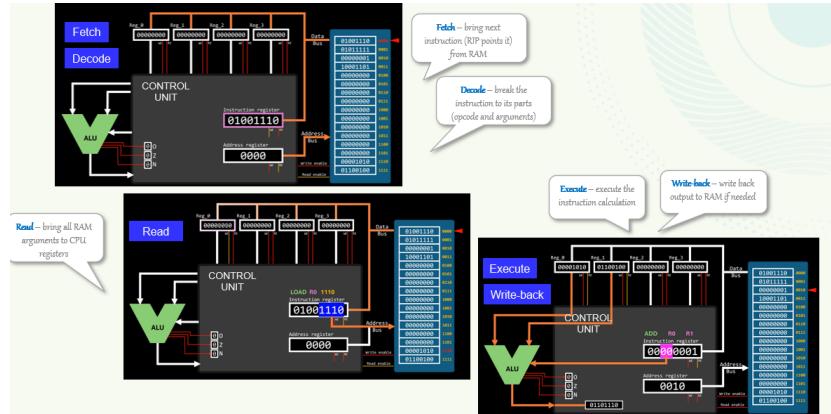


נראה כי בגישה זו עובדים תמיד עם שורות. מה *Miss rate*? נשים לב כי עבור A הוא $\frac{1}{n} = 0\%$ (כיון שאנו ניגשים לכל איבר n פעמים, שהרי הסיכוי שלא נגע אליו $\frac{1}{n}$ - כמובן אתה מביא את האיבר פעם אחת עבור n פעמים) עבור B כתע עבור C הוא גם 25% (!) - בכל מקרה: שיפרנו את *Miss rate*.

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

Pipeline friendly code 7.2

נזכיר לדון בחומרה. בחומרה יש *Control Unit* ו *ALU, RAM* וכן ישנה *Cache* ש羞שה את כל מה *ALU* ש *RAM* צריך פרט לחישוב עצמו ש羞שה *ALU*.

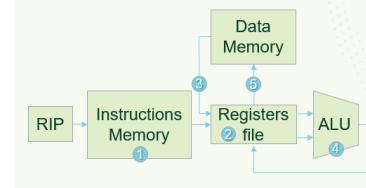


נראה כי הוא אחראי גם ל קישור לדטה בסאס, הוא מביא את ההוראה הבאה וכו'. הוא מלא לו את הריגיטרים שנחנו משתמש בהם, והוא מקבל את הפקודות לביצוע. כמובן - הוא שולט על כל מה צריך לעשות לפני ואחרי חישוב. **כל פקודה ישנים 5 שלבים:**

- : 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.

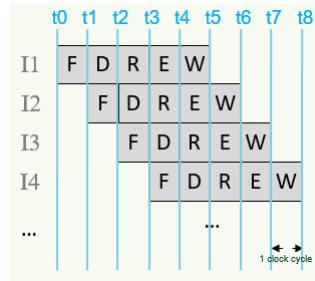


השלבים הללו שכל פקודה מבצעת - הם בלתי תלויים זה בזה.

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



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

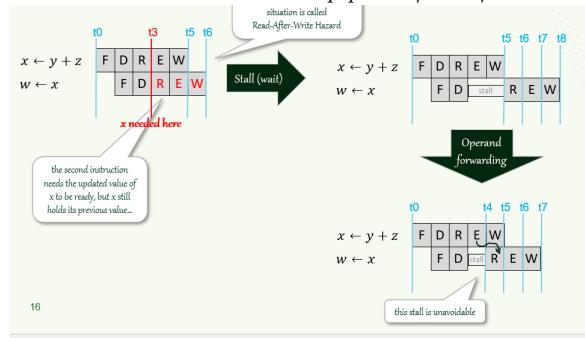


כזכור, לאחר שנסיים של רשות *fetch* ישר *latch* של השני. זו צורת *PIPELINE* זה עדייף מאשר לבצע רצף סדרתי של השלבים ולאחריהםשוב. בצורה זו מס' היסודות יורד! נראה כי במקרה הראשון עליה לנו 20 *cycles* וכן רק 8 בלבד. באופן כללי, השיפור יהיה לפי הנוסחה הבאה:

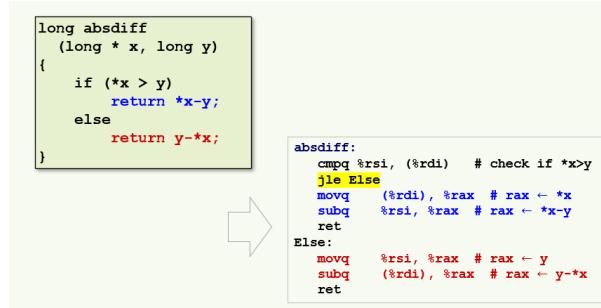
$$\lim_{n \rightarrow \infty} \frac{5n}{4+n} = 5$$

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.

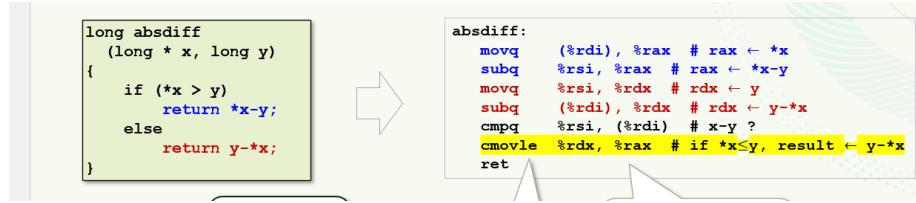
אם כן, נראה שהשיפור זה לא תמיד יעבד. נתבונן בדוגמה מטה: יש תלות בין הפקודות, בשליל לבצע את השורה השנייה כדי לדעת את השורה הראשונה בצדדי לדעת את x . נראה כי כאשר נבצע פקודה השנייה, זה בדיק השלב שהפקודה הראשונה מבצעת *execute* - היא בדיק מחשבת *read* לפקודה השנייה. זה מוביל לשנקרת הפעולה הראשונה חיבת לחכות (!). 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 מוחזרת 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 *pipeline*, אך אנחנו מעדיפים קוד כמו זה שהוא סדרתי.

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

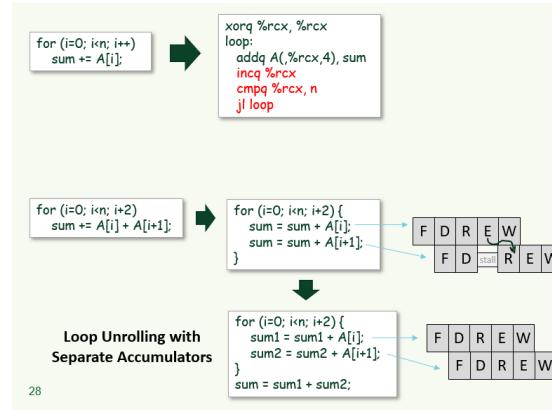
```

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* שהוא *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 באיטרציה. וכך אנחנו בוחרים להשתמש בשני סכומים בשבייל שמניע מהבעיה שנוצרת במקרה שbamatz: הוא חייב לחכות לפני שמתקדם ונוצר stall 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 $0 \geq i, j \geq !i$. 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. מביא 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.

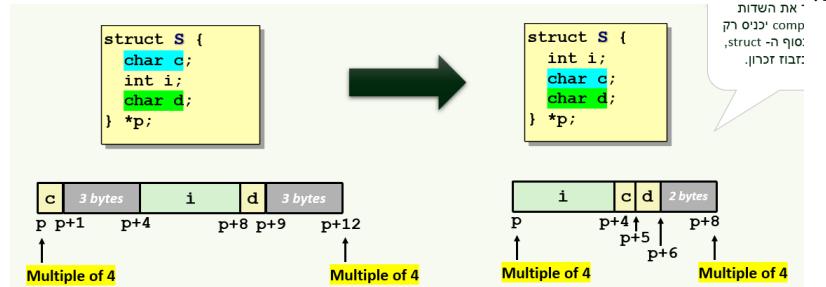
אם הכתובת של x מתחולקת ב-64 לא שארית (64 כי *cacheLine* הוא 64 ביטים), אם נרצה לקרוא את x נדע כי כל *4Bytes* שלו יהיו בשלמותם באוֹת שורת *cache*. יותר מזה: גם אם הכתובת של x מתחולקת ב-4 לא שארית - כל הביטים של x יהיו באותו *cacheLine*. מדוע? נניח 1028 (לא מתחולק ב-64, כן מתחולק ב-4).

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.

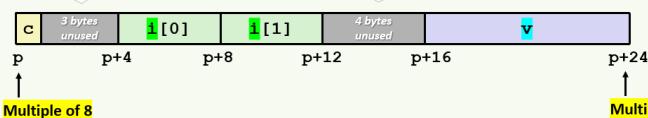
נתבונן במבנה פשוט של struct. מה שהgcc עושה במקרה זה זה שהוא מסתכל על השדה של הסטרuktאט בגודל הגדול ביותר – במקרה שלנו כאן זה *int* וכן אנחנו נרצה כי הסטרuktאט יתחל ויסיים בכתובת שמתחלקת בשדה הגדול ביותר בסטרקטט: כמובן גם הכתובת של ההתחלה וגם של סוף הסטרuktאט יצטרכו להתחולק ב-4 במקרה שלו.

כלל: הקומפילר מספיק חכם בשביל לעשות זאת בעצמו, ככלומר הקומפילר דואג שכותבת התחלה והסיום של הסטרuktאט יתחולקו 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 זיכרון. נבחן כי באופציה משמאל הוא שם *char* ולאחר מכן מוכן כיון *short* צרי להתחולק ב-4 הוא יוצר של 3 בתים מיוחדים עד שהוא שם *int*, לאחר מכן הוא לא זוקק *gap* עבור *char* שכן הוא יכול לשמש *char* בכל מקום, ולאחר מכן הוא שוב זוקק *gap* כיון שהכתובת האחורנית של *struct* הייתה להתחולק 4.

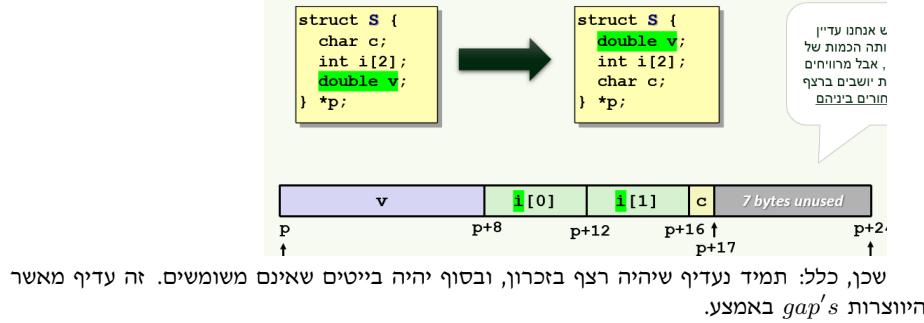


דוגמא נוספת. נניח ויש לנו סטרקטט של *i[2], double : char, int*. הגודל הכללי הוא $8 \times 2 + 4 = 20$ וכאן כתובות התחלה והסיום חייבות להתחולק ב-8. אם כן, נראה כי קיבל את המצב הבא בזיכרון:



כאן למשל, בכל מקרה אין אפשרות לשפר את הסידור. תמיד מקבל שכתובת האחורונה תסתהים כאן ב-24+. *p*.

אם כן, עדיף שנסדר את הSTRUktAT באופן הבא ונקבל:



שכן, כלל: תמיד נעדיף שיהיה רצף בזיכרון, ובסוף יהיה בייטים שאיןם משומשים. זה עדיף מאשר היוצרות *gap's* באמצע.

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 יסתהים כי *char* הוא *signed*. כאשר נגין נבע `c++` ו-`c` הפוך לשילילי. נסתכל על הדוגמה הבאה, מה CUT ידפס?

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

האם נקבל 125, 126 ? לא נראה זה נשמע הגיוני, כשנגיע `c = 127` לא יתקיים הדרוש ונסיים. בפועל תהייה לנו לאויא אינסופטי: הקומפיילר נורא חכם. אך לפעם הוא מניח הנחות שגויות. כאן, ובכל קוד בשפת הקומפיילר יניח שאין *overflow*. מה הכוונה? בדוגמה הקודמת.cn היה *overflow*. למה שהקומפיילר יניח זאת? הקומפיילר בוני לבצע אופטימיזציות בקוד ואם הוא יניח שיש אברפלול הוא לא יוכל לבצע את האופטימיזציה הבאה: הרי, תמיד $c < c + 1$ ולכן הקומפיילר הולך לתרגם את השורה ל- `while(true)` ולכון נקל לו להאל אינסופטי. זו דוגמה לאופטימיזציה של הקומפיילר. لكن – כיוון שהוא מניח שאנו אברפלול, הוא ביצע את האופטימיזציה וכאנ קיבלו לו להאל אינסופטי. 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. אם זאת, הקומפיילר עדיין לא מתכוון לשנות את הקוד לקוד הזה. נתבונן בדוגמה הבאה.

$f(&x, &x)$

כלומר, אם קיבלנו פערמים *x*, בגרסה הקודמת קיבל $4x$ ובגרסה שלנו $3x$. שכן, $x + x + x + x = 4x$ ו- $x + x + x = 3x$. בשורה הראשונה ניב $2x = x$, ולאחר מכן $2x + y == 2x + 2x = 4x$. אם כן, בשורה השנייה נקבע $3x \neq 4x$. כמובן $2x = 3x$. אבל אם נבטיח ולכן, הקומפיילר לא יעשה זאת. ואניונו: גם לא בהכרח נעשה 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 בתחילת כל פונקציה לשים *clock* בתחילת ובסוף, ובסוף התוכנית להדפיס את הפרש בין זמן הסיום לבין הначלה ולדעת כמה זמן התוכנית שלנו רצתה.

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

A screenshot of a terminal window titled "marina@vm:~/Archi\$". The terminal shows the command "gcc main.c" followed by "time ./a.out". The output includes:

```
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 - run our code" line in the output.

רעיון נחמד, אך פחות טוב:usch, שאנו מודדים אכן שאלינו צרכיים למדוד. הרצנו תוכנית, ובו זמינותה התוכנית רצתה רצוי אחרים במחשב - לפחות מערכת הפעלה רצתה, ויתכן גם תוכניות אחרות, כיון שכמות התוכניות שלנו במחשב קתנה ממספר ה *CPU* שיש במחשב, מערכת הפעלה נותרת קצר זמן ריצה לתוכנית אחת, קצר לשנייה, וכך היא מರיצה את כולן במקביל. אך בפועל: יתכן שהחלק מהזמן שהתוכנית הוזעקה רצתה כאן לאורורה היא בכלל לא רצתה אלא המתוינה שה *CPU* יירץ אותה.

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

real - הזמן שבאמת התוכנית קיבל זמן *CPU*
user - מזמן כל הזמן שהוקצה, כמה זמן היא הייתה ב *user mode*
kernel - מזמן כל הזמן שהוקצה, כמה זמן היא הייתה ב *kernel mode*

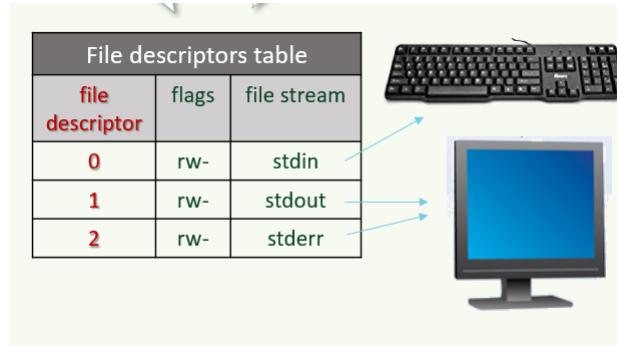
שתוכנית רצתה היא מבצעת פקודות - פקודות ריגולות ופקודות שאסור לה לבצע (להדפיס למשל, היא צריכה לבקש מהמערכת הפעלה גישה לדבר עם המסך.). הוא מצב משמש - התוכנית שולץ רצתה עם הרשות מוגבלות, לא יכולה לשמש שירות לחומרה ולא יכולה לבצע פעולות מסווגות. רוב הקוד נמצא *kernel mode*. רק כאותה מזעקה רצתה עם הרשות המלאות ויכולת לגשת לכל החומרה, רק כאותה מבקשת שירות. מתקיים $\text{real} \approx \text{user} + \text{kernel}$.

8 הרצתה : 9 + 10 files&system – Calls&shell

כיצד הקוד שלנו קורא למערכת הפעלה בשbill לבצע פעולות שאין יכול לבצע?

Unix files 8.1

שאנו פותחים קובץ הוא מקבל טבלה שנקראת *File description table* והוא מספר הקובץ שנפתח עד מה (*מספר שלם*) *file descriptor* בכל תוכנית שאנו מרים היא מקבלת כבר שלווה קבועים שפותחים, באשר מערכת הפעלה פתחה אותן: *stdin* - כל מה שאנו כותבים על המקלדת, מהקובץ אנחנו מושכים את הקריאה מהמקלדת ועוד קובץ שנפתח. *stdout* - המסרך שלנו, להציג דברים על המסרך למשל *printf*. *stderr* - משתמשים בו רק בשbill הדפסה דחופה, למשל עבור *error* שצרי להדפיס באופן דוחף למסך. תמיד, בעת שימושה הפעלה מרים שלושת הקבצים הללו נפתחים.



אנחנו נרצה להשתמש בקבצים אלו. נוכל למשל לכתוב את השורה הבאה:
`fprintf(stdout,"Hi\n");`

כאן אנחנו משתמשים בהדפסה של *file*, *descriptor* לו במי אנחנו רוצים להשתמש ומה להדפיס. אנחנו יכולים ליצור קבצים מסווגים, כמו בדוגמה הבאה: יצרנו קובץ בשם *a.txt*, והוא מקבל *descriptor* *a*.txt, ואילו אנחנו מדפיסים את הקובץ. תמיד כרגע - צריך לסגור את הקובץ. אנחנו מדפיסים אל תוך הקובץ, ובשביל שנראה הדפסה למסך צריך לעשות *cat* לקובץ. נבחון כי נתנו תיאור *w+*, בטבלה בתמונה מטה נראה שיש תיאורים מוכרים לשינויים נפוצים כמו כתיבה קרייה וכו'.

```

r      read only mode
r+    read and write
w      write only, truncated or created
w+    read and write, truncated or created
a      write only, appends or creates
a+    read and write, appends or creates

#include <stdio.h>
#include <unistd.h>
void main() {
    FILE* fd = fopen("a.txt", "w+");
    fprintf(fd, "print to a.txt\n");
    fclose(fd);
}

[Terminal]
marina@vm:~/Archi$ gcc main.c
marina@vm:~/Archi$ ./a.out
marina@vm:~/Archi$ cat a.txt
print to a.txt
marina@vm:~/Archi$ 
```

דוגמה נוספת: כאן בדומה ישנו *O_WRONLY|O_CREAT* : מילים שמורות ב' *C* - שטוענות, נרצה לפתוח קובץ לקריאה בלבד, ואם הוא לא קיים אז אנחנו ניצור חדש. כלומר - אם הוא קיים הקובץ הנ"ל הוא פתוח, ואחרת יוצר חדש בשם זה. לאחר מכן, אנחנו כותבים: מצינים את *file descriptor* של הקובץ אליו אנחנו כותבים, את מה שנרצה לכתוב ואת אורך המחרוזת שנרצה לכתוב. פונקציית *write* מוחזירה לנו את *cnt*, שהוא מס' הבתים שהייתה הצליחה לכתוב. לבסוף - מדפיסים כמה בתים הצלחנו לכתוב. לבסוף: כמובן סוגרים,

שלא יהיה דליפות זכרון.

מדובר אחד התלמידנו: מה זה 0644? (לא בחומר של הקורס)

```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

void main() {
    int cnt;
    int fd = open("a.txt", O_WRONLY | O_CREAT, 0644);

    cnt = write(fd, "123456789\n", strlen("123456789\n"));
    printf("written %d bytes\n", cnt);

    close(fd);
}
```

Family.read:

- are system calls
- are not formatted I/O: we have a non-formatted byte stream

Family.write:

- are functions of the standard C library (<stdlib.h>)
- use an internal buffer
- are formatted I/O (with the "%..." parameter) for some of them
- use always the Linux buffer cache

More details [here](#).

: נתבונן בדוגמה מטה. בקטע הקוד הראשון משמאלי - הקוד כותב לתוך הקובץ. בקטע הקוד השני, קודם לכן מנקים את הקובץ (ממה שהיה בו קודם) - ואנו כותבים בתוכו. ומה ההבדל? נניח ויגילינו כי הקובץ כבר קיים, אז בקטע הקוד משמאלי אנהנו כתבוו בתחילתו של קובץ, אך לא מחקנו את תוכון הקויים. בקטע הקוד מימין, כן ישנו ניקוד של התוכן הקויים.

```
void main() {
    int cnt;
    int fd = open("a.txt", O_WRONLY | O_CREAT, 0644);
    cnt = write(fd, "Hi", strlen("Hi"));
    close(fd);
}
```

```
int cnt;
int fd = open("a.txt", O_WRONLY | O_TRUNC | O_CREAT, 0644);
cnt = write(fd, "Hi\n", strlen("Hi\n"));
close(fd);
```

Note that the previous content of the file is not truncated, and only two first characters are overwritten.

Note that the file is completely overwritten.

System calls 8.2

מה זה *system calls*? נניח ויש אפליקציה שלנו (*User space*) - אנחנו רוצים לכתוב אל הקובץ

- זה השלב של *System call*: מערכת הפעלה נקראת ורק היא יכולה לנשט אל חומרה. *kernel* יהיה מערכת הפעלה. אנחנו - היוזר, חייבים לבקש ממנו.

נסתכל על דוגמה, התחילה שמוופיע מטה.

שלב ראשון: נניח וכתבנו פונקציה ב-C.

שלב שני: הפונקציה *fwrite* היא פונקציה שמופיעה בתוך ספרייה בשם *glibc* (ספרייה סטנדרטית).

- היא עשויה *buffering* עיבוד ובסוף קוראת ל(*write*) הנמוכה יותר.

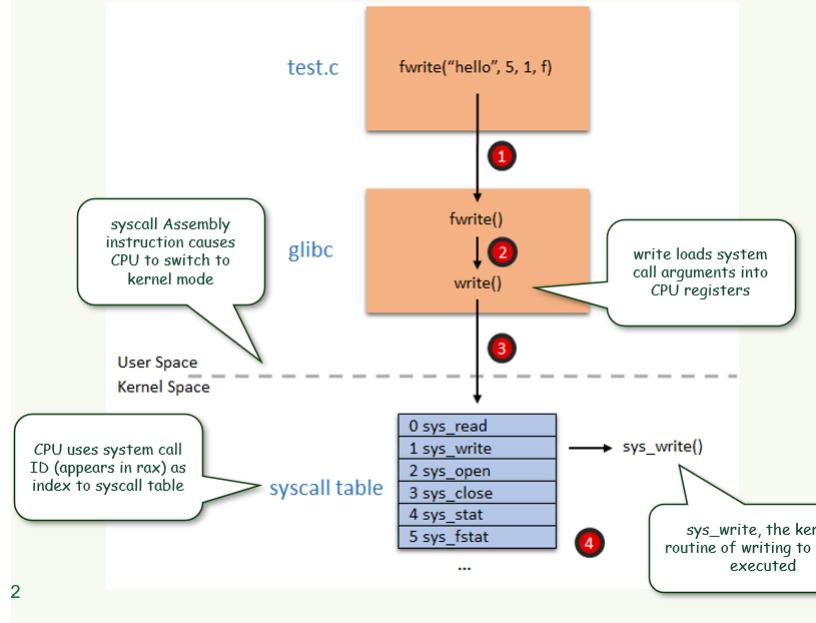
שלב שלישי: פונקציית *write* - זו בעצם בתוך *glibc* תפקידה היא להזכיר את הקריאה (לכתייה) למערכת הפעלה.

שלב רביעי: זה השלב שבו עוברים לאסמבלי. מתרגמים את הקוד לאסמבלי, ואז ישנה הפקודה *mode* *syscall* ו קופצת לקוד של ה kernel.

שלב חמישי: בתוך ה kernel - ה kernel אמרה: בוא נראה מה רוצים מני. זה השלב שמערכת הפעלה נקראת למעשה. ישנה טבלה בשם *syscall table* שאומרת: מה אני בטור מערכות הפעלה יכול

לעשות? מערכת הפעלה לוחחת את *rax*, מוצאת את הפונקציה המתאימה לערך בטבלה שמתאים (במקרה שלנו *rax = 1*)

בשלב האחרון, הפונקציה `sys_write` בקרנל ניגשת לחומרה ממש וכותבת. לאחר מכן מתקבל ערך חזרה ב-`rax`, הkerןל חזר ליוור מוד והקוד שלק משמש בכתובות שבה קריאנו לפונקציה.



נעבור לאסמבלי - נרצה למלא הרי ורגיסטרים, ואת זה אפשר רק באסמבלי. ראשית, מגדרים מהו `char* buffer` (שהוא ה-`char` שלנו ממקודם). בתחילת סקשן `txt` אנחנו מתחלים את `rax` ל-0, (`lmah?` ככה). השורה הבאה מצינית שאחננו רוצים לקרוא מקובץ `stdin`. בשורה השלשית - אנחנו מעבירים את הכתובת של אפר `lrsi`. אנחנו מעוניינים לקרוא ביט אחד, אז קוראים `l`.

- system call number (in rax): 0
- arguments:
 - `rdi`: file descriptor (to read from it)
 - `rsi`: pointer to buffer (to keep a read data into it)
 - `rdx`: maximal number of bytes to read (maximal buffer size)
- return value (in rax):
 - number of bytes received (0 if EOF)
 - On errors: negative number

```
.section .bss
buffer: .space 1

.section .text
.globl _start
_start:

    movq $0, %rax      # system call number (sys read)
    movq $0, %rdi      # file descriptor (stdin)
    leaq buffer(%rip), %rsi   # buffer to keep the read data
    movq $1, %rdx      # bytes to read
    syscall            # call kernel

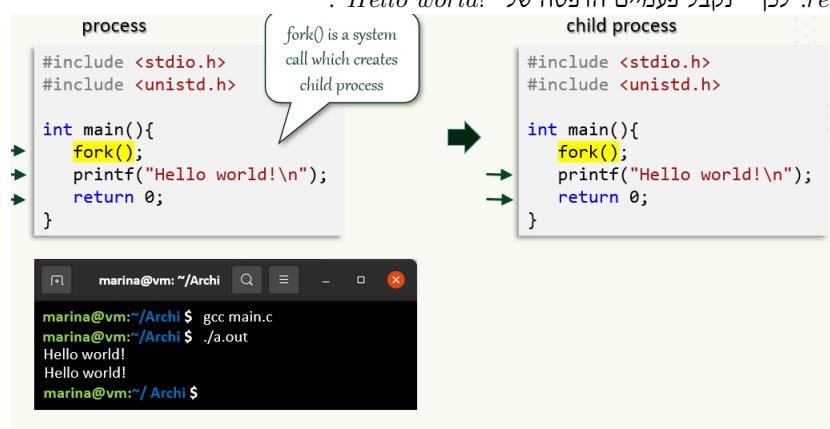
    movq $60, %rax     # system call number (sys exit)
```

נבחן כי ישם המונן *system calls* - בקשות שניתן לבקש מערכת הפעלה. לכל אחד ישנו שם שזה ב-*.rax*. כמה בדיק יש?

Unix processes 8.3

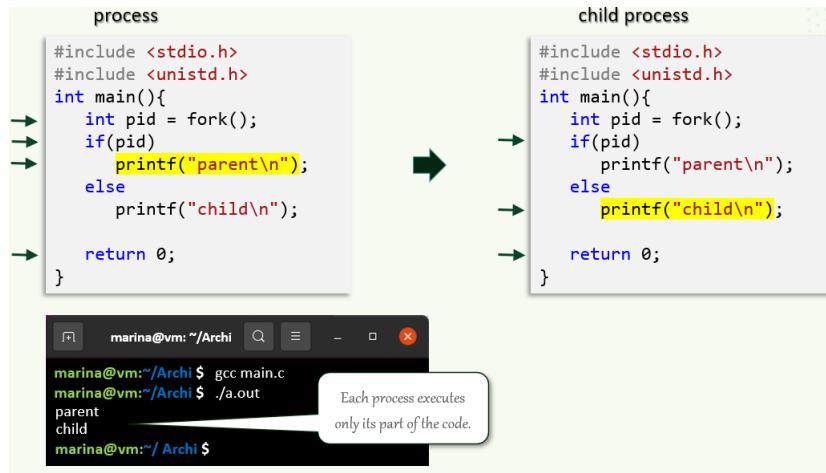
מה אנחנו יודעים על *process* ? תחילה? כשאנו מרכיבים **תובנית** (קובץ בינהר) אנחנו מבקשים ממערכת הפעלה לחת את הקובץ שלנו ולהריץ אותו - וזה מערכות הפעלה מבצעת *process*. ואז, מערכת הפעלה נותנת שירות עבור *process image*.

נתבונן בתוכנית הבאה - קטרה אך כבר לא מוכרת לנו: מהו *system call fork()*? מהו מיחוד הוא אומר למערכת הפעלה: שכפל אוoui. מה הכוונה? מי מבקש שכפול? (!) - מערכת הפעלה תשכפל את *process* שכרגעה. מה הכוונה בשכפל *process*? במלוא המובן של המיליה - שני *process* זהים לחלוטין שרכסים יחד. כתע בקוד הזה: שניים יבצעו *printf* ושניהם יבצעו *return*. אך נקבל פגמים הדפסה של "Hello world!".



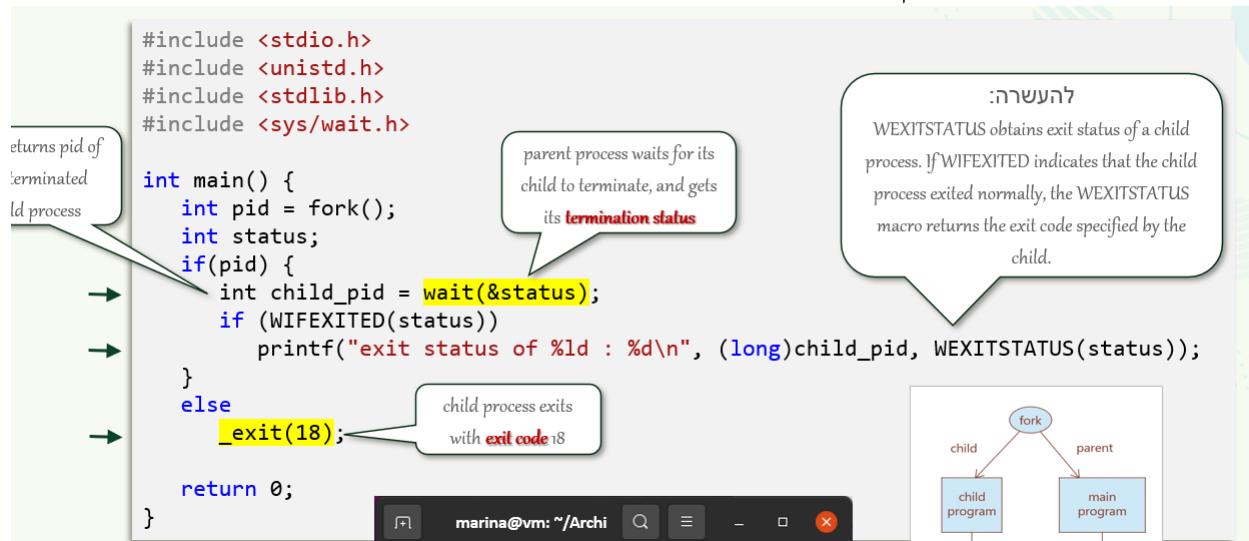
קיבלו שני *process* - החל מל-*fork* הם יירצו יחדיו, נקבל את *parent* ואת *child* - *child* הוא המשוכפל. מדובר ב**ריציה במקביל של שני התהליכים**. אם במחשב יש לנו כמה הם *cpu's* מושוכפים, מדבר ב**ריציה במקביל של שני התהליכים**. אם במחשב יש לנו כמה הם *cpu's* מושוכפים, מדבר ב**ריציה במקביל של שני התהליכים**. כדרכו של *system call* מוחזר גם כאן ערך: מסטבר, כי כל *process* ישנה תעוזת זהות - ככה מערכת הפעלה מזהה ומבדילה בין *process*. תעוזת זהות נקראת *pid* - מסוג *int*, אנחנו יכולים לтипיס אותו אם נסתכל על הערך המוחזר של *fork*. ישנו הבדל בין הערך המוחזר ב-*fork* שמקבל האבא לבין הערך המוחזר ב-*fork* שמקבל הבן. פקודה החשומה של *pid* מתבצעת בנפרד עבור הבן והאב. נראה כי הבן קיבל ערך מוחזר של אפס והאב קיבל ערך של *.id*.

נראה כי האבא ידפיס *parent*, הבן ידפיס *child* כיון שהבן יהיה *id* של האבא - וכך הוא יכנס *else*. ושוב נבהיר: מרגע ההשמה של *fork*, הקוד יתבצע במקביל.



הערה חשובה: מתי *fork* נכשל? כאשר נרצה לעשות יותר מדי *process* (יותר מ-65,000!) אז מערכת הפעלה תגיד לא יכשל. שכן, מערכת הפעלה רוצה להגן על המחשב. אם פותחים יותר מדי *process* המחשב יהיה איטי מאוד.

נתבונן בדוגמה מטוה. מערכת הפעלה - אל תתני לי לתקדם. אני יצרתי *process* בן בשבייל שהוא יבצע עבודה כלשהי. אבל - אנחנו לא רוצים ל התקדם הללו עד שהוא יסיים, אך אומרים למערכת הפעלה: אל תתני לי זמן ל*cpu*, תבצעי CUT את הבן, וכשהבן יחייר נמשיך את הקוד שלו.



נתבונן בדוגמה הבאה - מערך של פוינטורים של *char*. מוגדר מערך שיש בו פוינטר לסטRING *ls*, פוינטר לסטRING *l* – ופויינטר שהוא *null*. לאחר מכן, מבצעים *fork*: רצים *process* בו. אנחנו אומרים – אם אנחנו אבא: נחכה. ואחרת – אם אנחנו בן: נבצע *execvp* – הוא מבקש ממ מערכת הפעלה להחליף לו *process image* – הוא מורכב מקוד ומARGV, וכך *process image* קובע מה הקוד הולך לעשות. מה בדיק אם מנסים לעשות כן? מה הכוונה להחליף?

תוכנית אחרת. כלומר: יכול לבקש ממערכת הפעלה להפעיל *process* אחר - במקרה שלנו: *.elf* דהינו *ls argv[0]* בן יהיה *execv*.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
    char* argv[3] = { "ls", "-l", 0 };
    int pid = fork();
    if (pid)
        wait(NULL);
    else
        execvp(argv[0], argv);
    return 0;
}
```

this is indeed “ls -l” Shell command

execvp() loads **new process image** for child process from **ls.exe** file. *argv* is sent as a command-line argument for the new process.

marina@vm:~/Archi \$ ls -l
total 144
-rwxrwxr-x 1 marina marina 17328 Oct 23 15:09 a.out
-rw-rw-r-- 1 marina marina 203 Sep 24 11:31 main.c
marina@vm:~/Archi \$

שורת הפקודה 8.4 : Unix shell

נתבונן בקוד הבא. (*getCommand* מדפס את *\$* למשל, *typePrompt* היא פונקציה שקיימת *shell* ומחכה לפקודה ופרמטרים. לאחר מכן מבצעים *fork* - רוצים להריץ תהליך חדש: בונים *(fork = 0)*. נזכיר כי הילד יחזר 0 *fork > 0* כי הילד יחזר 0 *child process* אחרות: זה הילד, ולכן אנחנו מנסים שוב לשנות את *process image*).

```
while (TRUE) {
    typePrompt();
    getCommand (&command, &parameters);

    if (fork() > 0)
        /* Parent code – wait for child to exit */
        wait();
    else
        /* Child code – execute command */
        execvp (command, parameters);
}
```

נתבונן בדוגמה הבאה: עליינו לכתוב תוכנית שמקבלת כארגומנט קובץ ב-*C*, למשל *a.outmain.c* נרצה שהתוכנית תкомפלט את הקובץ, ותרץ אותו. נתבונן בתוכנית שתעבדו:

```
int main(int argc, char ** argv) {
    int pid = fork();
    if(pid) {
```

```

wait(NULL);
char* argvChild[2] = { "./main.exe", 0 };
execvp(argvChild[0], argvChild); }
else {
char* argvChild[5] = { "gcc", "-o", "main.exe", argv[1], 0 };
execvp(argvChild[0], argvChild); }
return 0;

```

מה קורה לנו?
הפייטול: `fork()` ברגע שהתוכנית מגיעה לשורה `int pid = fork();`, נוצר עותק כמעט זהה של התהיליך הנוכחי.
בתהיליך הבן: הפונקציה מחזירה 0.

1. תהיליך האב: הפונקציה מחזירה את ה-*PID* (מזהה התהיליך) של הבן (מספר חיובי).
2. תהיליך הבן (*else*): שלב הקימפול - הבן נכנס לבlok ה-*else* כי אצלו *pid* = 0, הוא מגדיר מערך ארגומנטים להרצת הקומפיילר: `gcc -omain.exe`, הוא משתמש ב-`execvp`. הפקודה זו מחליפה את התוכן של תהיליך הבן בתוכנית `gcc`.
3. תהיליך האב (*if*): שלב המסתנה וההרצת האב נכנס לבlok ה-*if*(*pid*) כי אצלו ה-*PID* חיובי `wait(NULL)`: וויי שורה קריטית. האב עוצר ומהכח עד שתהיליך הבן (הקימפול) יסתתיים. בלי זאת, האב היה מסה להרץ קובץ שעוד לא נוצר.
לאחר שהבן סיים, האב ממשיך ומגדיר מערך להרצת הקובץ שנוצר: `main.exe`. גם הוא משתמש ב-`execvp`, מה שהופך את תהיליך האב עצמו לתוכנית שהרגע קומפליה.

8.5 שימוש בקוד אסטטלי בתוך קוד ב-C (מהתרגול)

נרצה לבצע לתוכנו לוחטמייע קוד של אסטטלי בתוך קוד של סי. מדוע? גישה לרגיסטרים ספציפיים מה שאי אפשר בסי, פקודות מיוחדות שאין *C* כמו *flags*, אופטימיזציה במקומות ספציפיים.
ישנן שתי דרכים של כתיבת אסטטלי בתוך סי:
אופציה ראשונה - תחביר כמו ("assembly - code") : מה הבעה בכתביה זו? הקומפיילר לא בטוח ממה עשית. הוא עלול לדرس רגיסטרים, לא יודע איזה משתנים השתנו וקשה לתחזק.
אופציה שנייה - הגדרה המורחבת. תחביר כללי הוא מהצורה הבאה:

```
asm("assembly - code" : output-list : input-list : clobber-list)
```

מה כל חלק עושים?
קוד האסטטלי עצמו: *assembly code*
משתנים שהקוד כותב אליהם: *output list*
נקודות: *[name]"constraint"(expr)*

לדוגמה:
long result;
asm("movq %%rax, %0" : "=r" (result))
כאן למשל, בין Ci expr מותאר את שם המשתנה שהקוד כותב אליו. ישנו נפקודים:

Constraint	Meaning
"=r"	Update value stored in a register
"+r"	Read and update value stored in a register
"=m"	Update value stored in memory
"+m"	Read and update value stored in memory
"=rm"	Update value stored in a register or in memory
"+rm"	Read and update value stored in a register or in memory

משתנים שהקוד קורא מהם, הפורמט זהה לש *input list* ומזהירה הבאה: *constraint*"(*expr*)". דוגמה:

```
long x = 5, y = 10;  
asm("addq %1, %0" :  
    "=r" (result) // output :  
    "r" (x), "r" (y) // inputs );
```

רегистרים או דגלים שהקוד משנה: *clobber list*

(מחריג) Undefined Behavior 8.6

הכוונה היא בהתנהגות שלא מוגדרת בפרט של שפת C. הקומפיילר במצב זה יכול לעשות מה שהוא ירצה. כל הרשימה הבאה היא דוגמאות להתנהגות שאינן מוגדרות:

Division by zero

Signed integer overflow

Indexing an array outside of its bounds (either index too large or negative)

Null pointer dereferencing

Modifying a string literal

Not returning from a value-returning function (other than main)

Shifting by a negative value or a value greater than the number of bits in the value

למה התנהגות לא צפואה מסוכנת? הקומפיילר מניח תמיד כי התנהגות לא צפואה לא קורית!
נתבונן על דוגמה בקוד הבא:

```
int main() {  
    char buf[50] = "y";  
    for (int j = 0; j < 9; j++) {  
        printf("%d\n", j * 0x20000001);  
        if (buf[0] == 'x') break;  
    }  
}
```

הקומפיילר עשו (וכך יעשה) את האופטימיזציה הבאה על מנת לחסוך בחישוב הכפל ובודיקות:

```
for (int i = 0; i <= 0x20000001; i++) {  
    if (buf[0] == 'x') break; // ... }
```

מה הוא עושה? הוא רואה שהמספר שבתנאי גדול מאוד - זה יותר גדול מINT_MAX
אזי, הוא אומר: ? בזמנים לא עברו את הערך הזה - ולכן הוא משנה את ערך הלולאה *true*:
ולאלה אין סופית!

לכן, עליינו להניח כי הקומפיילר לא עד כדי כך חכם - צריך להמנע מהתנהגות לא מוגדרת, כי
הקומפיילר מניח שאין כזו בקוד שלנו.

Macros 8.7 (מהתרכזול)

זכור כי *macro* היא תבנית שמחלייפה טקסט בזמן *preprocessing* (לפני הקומpileציה). ישנו שני

סוגים:

א. החלפה פשוטה (*Object like macro*)
למשל, נניח והגדרנו

```
#define Pi 3.1415
```

במקרה פשוט זה, בכל מקום בקוד שבו מופיע *Pi* זה יוחלף בערך 3.1415. פשוט ותמיד יקרה.

ב. החלפה עם פרמטרים (*Function like macro*)
נתבונן בדוגמה הבאה:

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

```
int x = MAX(5, 10); // → int x = ((5) > (10) ? (5) : (10));
```

זו דוגמה להחלפה עם פרמטרים.

נבחן כי במקרה זה ישנו סכנות רבות עבורהנו.

דוגמה ראשונה - חוסר של סוגרים. נניח והגדרנו את הדפיין הבא:

```
# define SQUARE(x) x*x
```

```
int y= Square(2+3)
```

מה שיקרה במקרה זה הוא $2 + 3 * 2 + 3 = 2 + 6 + 3 = 11$ אך בפועל היוו אמורים לקבל 25.
כיצד נפתרו? נו, כמובן, נוסיף סוגרים.

דוגמה שנייה - Token concatenation: מה זה לעשות? האופרטור `#` מזדיבק שני טוקונים להיות טוקן אחד. כלומר -

```
#define CONCAT(a, b) a##b
```

```
int xy = 42;
```

```
int result = CONCAT(x, y);
```

מה קורה כאן? מוחלף בסך הכל השם, כלומר *xy* → *x*, *y* ונקבל

טוקנה נוספת:

```
#define VAR(num) variable_##num
```

```
int VAR(1) = 10; int VAR(2) = 20; int VAR(3) = 30;
```

במצב זה, לאחר preprocessing נקבל:

```
int variable_1 = 10; int variable_2 = 20; int variable_3 = 30;
```

חשוב: התוצאה חייבת להיות טוקן יחיד. לכן דברים כמו `BAD(a,b)a## + b` של *defined* לא יעבדו.

דוגמה שלישית - Variadic macros

מדובר במקאו שמקבל מספר משתנה של ארגומנטים. תחביר בסיסי יראה כך:

```
#define MACRO_NAME(fixed_args, ...) replacement _ VA_ARGS _ _
```

נתבונן בדוגמה הבאה:

```
#define PRINT(...) printf(_ _ VA_ARGS _ _)
```

```
PRINT("Hello\n");
```

```
PRINT("x = %d\n", 42);
```

```
PRINT("x = %d, y = %d\n", 10, 20);
```

לאחר preprocessing נקבל:

```
printf("Hello\n");
```

```
printf("x = %d\n", 42);
printf("x = %d, y = %d\n", 10, 20);
```

חשיבות: סדר ההרחבה ורקורסיה של מקרו:

הכל הבא חשוב מאוד: ארגומנטים מתרחבים לפני החלפה במקרו, אלא אם הם *stringified* או *concatenated*.
1. דוגמה ראשונה -

```
#define DOUBLE(x) ((x) * 2)
#define VALUE 10
int result = DOUBLE(VALUE);
או הרחבה רגילה. במקרה זה, VALUE מתרחב תחילה, ואז DOUBLE מתרחב. קיבל בכלל
(10) * 2 = 20 מקרה
2. דוגמה שנייה -
```

```
#define STR(x) #x
#define VALUE 100
STR(VALUE)
כואז, STR(VALUE) לא יתרחב בגל שיער # ולכן נקבל בדיקת STR(VALUE) - STR(VALUE)
```

9 הרצאה 10–11 : *Co(operating)-routines - Coroutines*

נרצה לראות כיצד פונקציה *f* שתשתף פעולה עם פונקציה *g* : ומה הכוונה בשיטות פעולה בין פונקציות זו לא הכוונה של "ובאו ונחשב בידך" - אלא עזרה מבון אחר. מהו המבון الآخر הוא? במשמעותם עד היום הרצינו תוכניות בקרה סדרתיות, למשל באופן הבא:

```
void f() { char ch=1; while (ch < 10) printf("%d", ch++); }
void g() { char ch = 'a'; while (ch < 'j') printf("%c", ch++); }
void main() { f(); g(); }
```

התוכנית זו רצתה די פשוטו: היא קוראת ל-*f*, קופצים אל *f* ומבצעים את *f*, בונים מבון *activation frame*, מבצעים קוד של *f*, חוזרים, בונים, בונים *activation frame* עבור *g*, מבצעים את *g* וחוזרים. מה אנחנו נרצה? נרצה לבנות תוכנית שעשויה "קצת *f*, קצת *g*" באופן איטרטיבי - הרצה

לסיוגין.

דוגמה מהחיים: יש לנו עבודה במבנה מחשב, ועבודה באלגוריתמים: אז נזગ בין המטלות, נעשה שאלת באלו, שאלת במבנה מחשב, וכן הלאה. אבל - צריך לעשות את זה בזיהירות, כי אם כל עשר שינוי נחליפין בין המטלות אנחנו לא נסימם שום דבר. לכן: נתקצב שעתיים לאלה, ואז שעתיים למבנה

מחשב, ואז שוב שעה באלה, ושלוש שעות במבנה מחשב.

למה זה טוב, בקוד? נניח שMRIינה הייתה דורשת כל יומיים שנדווח על ההתקדמות במטללה, ונניח גם צבי ידרוש זאת: נניח ונגייע אל MRIינה - ונאמר לה שסיימנו, ולצבי נגיד: לא התחלנו. הוא יensus - עדיף שנתקדם חצי בכל מטללה.

איך זה קשור לקוד? יש לנו הרבה תוכניות שרשות במחשב שלנו -فتحנו *gmail* וווצאפ. אם המחשב היה אמר - קודם נרץ את *gmail* ואז נרץ את וויצופ : מחשב שכזה היה על הפנים. למה? כי *gmail* לעולם לא תפסק לרווי, וגם לא ויצופ. لكن מערכת הפעלה חייבת לבצע *interleaving* - הרצה בו זמינית. לכן בשביבו *user* יהיה מרוצה, מערכת הפעלה חייבת לבצע *interleaving* - הרצה

לסיוגין (זה שונה מממקביל).

יש בפנינו מלא בעיות - אם השתמשנו בפונקציה מסוימת ברגיסטרים, וקיצנו לפונקציה אחרת, היא השתמשה גם ברגיסטרים, הערכים השתנו.

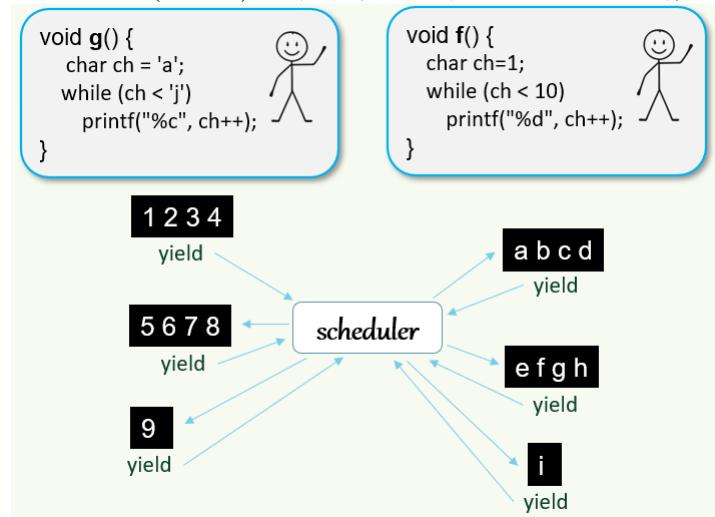
כל קוד יקרא coroutine ולכל coroutine ישנו מצב - *:state*

בתוך כל *state* נחזיק: *stack*. ערך של *stack*

- ב. ערכי הרегистרים
 ג. פלאגים (*Rflags*) - נרצה לחזור לש' *flag* שהוא לנו.
 ד. *stack pointer* *RSP* - ייחזק מצביע לפונקציה הרגולומית.
 ה. *instruction pointer* *RIP* - מצביע לפקודה של.

אם נרצה לקרוא *coroutine* *switch*, רגע לפני הקפיצה עליינו לקבע את *the state*, ולשמור אותו, לשחרר את *the state* של הפונקציה הבאה, וכן הלאה. מעריכים כאלו נקרים contexts switch בשמירינה מוגעה להרצאה שלנו אחרי ההרצאה במנוח'ץ - היא צריכה להחליף את מה שיש לה באש בחומר של אSEMBLY, ולאחרינו היא תחליף את מה שיש לה בראש בחומר של מקבילות. הקפיצה הזאת היא בדיקת *switch*.

הזמן שאנו נוהנים *coroutine* לרווח בין פונקציה יקרא טיים-קאונטו. כיוון שקשה למדוד זמן, נרצה לומר לכל פונקציה: תעשי מספר מסוים של פקודות, ואז נعبر לפונקציה הבאה. למשל כאן, אנחנו נוהנים לכל פונקציה להריץ 4 פקודות (4 הריצות):



הפונקציה למבצעת מותרת על זמן ריצה של התוכנית של עצמה - ונותנת אותה לתוכנית אחרת. זה נקרא *yield* (יתור מושך).

בתחילת, נאלץ לשמר בזיכרון את המידע של כל *coroutine* בתוך סשן *:data*. נקצתה מקום בגודל 128 ביטס, *ccb1*, יציג את *state* של *coroutine* *g()*, *ccb2*, יציג את *state* של *coroutine* *f()*, וכך הלאה. כמו כן, מגדרים מחסנית לכל *coroutine* וכן מגדרים משתנה גלובלי בשם *current_ccb*.
 בהמשך, נוצר לכל *coroutine* את התו הראשון להדפסה. (זה 49).(char1, char2). והפונקציה אכן תדפיס אחד ו-97, וזה מופיע את מס' הפעולות שביצעו עד כה ב-*coroutine* *g()*. *done1*, *done2* מסויים. הם שני *flag's* שטוענים האם יש לנו לזרום לחזור ל-*coroutine* *f()* או אחד מהם (או שניהם) סיימו.

```

ccb_scheduler: .space 128      # Scheduler CCB
ccb1: .space 128    # coroutine1 CCB
ccb2: .space 128    # coroutine2 CCB
stack1: .space 4096      # coroutine1 Stack
stack2: .space 4096      # coroutine2 Stack
current_ccb: .quad 0        # Pointer to current CCB

```

```

char1: .byte 49, 10   # char for func1 ('1' to '9'), '\n' for printing
char2: .byte 97, 10   # char for func2 ('a' to 'i'), '\n' for printing
quantum: .byte 0      # Instruction quantum counter
done1: .byte 0        # Flag for coroutine1 completion
done2: .byte 0        # Flag for coroutine2 completion

```

cut בסקשן `:text`: נגידר את הפונקציות. בתחילת, NATCHILAH, NATCHILAH את `quantum` של כל אחד מהם ליהי 4, שכך רוץ 4 איטרציות מכל אחד. `func2_loop` ו`func1_loop` בודקים האם הגענו כבר לטו האחרון - אם כן, רוץ להפסיק את הרצאה. אחרת, אנחנו עוד לא בסוף. נרצה להעביר 1 `rax` בשבייל כתובו, ו-1 `rdi` (ריצה לכתוב במסך). אנחנו קוראים `inc`, מוצאים מכך, ומורידים `msvcpr` / תוו הבא, ומורידים `msvcpr` בשבייל לציין `quantum` 1. אנחנו מגדילים את המספר לאפס המשך בולאה, אחריה קרא `yield`: מדבר בפונקציה שתפקידו נדון בה. לבסוף, אם סימנו אנחנו מגיעים אל `done` ומכנים שמייננו עם הפונקציה זו: וכך לא ניתן לפונקציה הנ"ל זמן ריצה.

```

func1:
    # Reset quantum
    movb $4, quantum(%rip)
func1_loop:
    cmpb $57, char1(%rip)  # Check if char > '9'
    jg func1_done

    # Prints the next digit from ['1','9']
    movq $1, %rax  # sys_write
    movq $1, %rdi  # stdout
    leaq char1(%rip), %rsi
    movq $2, %rdx  # length
    syscall
    incb char1(%rip)  # move to the next char
    decb quantum(%rip) # decrement quantum
    jnz func1_loop
    call yield  # Yield if quantum exhausted
    jmp func1 # On resume, go to next loop iteration

func1_done:
    movb $1, done1(%rip) # Mark coroutine1 as done
    call yield

```

```

func2:
    # Reset quantum
    movb $4, quantum(%rip)
func2_loop:
    cmpb $105, char2(%rip) # Check if char2 > 'i'
    jg func2_done

    # Prints the next character from ['a','i']
    movq $1, %rax  # sys.write
    movq $1, %rdi  # stdout
    leaq char2(%rip), %rsi
    movq $2, %rdx  # length
    syscall
    incb char2(%rip)  # move to the next char
    decb quantum(%rip) # decrement quantum
    jnz func2_loop
    call yield  # Yield if quantum exhausted
    jmp func2 # On resume, go to next loop iteration

func2_done:
    movb $1, done2(%rip) # Mark coroutine2 as done
    call yield

```

9.1 הפונקציה `yield`

בתחילת, אנחנו צריכים לבנות ב-`main` את כל המערכת עלייה דיברנו קודם. בתחילת NATCHILAH מתחילה מתחלים את `pointer` *scheduler*, בהמשך מתחלים את 1 *coroutine*. נבחן שאנו מתחלים בכל שלב את *the scheduler* להיות ראש המוחסנית (המחסנית הראשית גדלה מלמעלה למטה).

```

.global main
main:
    # Initialize scheduler CCB
    leaq ccb_scheduler(%rip), %rax
    movq %rax, current_ccb(%rip)

    # Initialize coroutine1 - set up stack with return address to func1
    leaq stack1(%rip), %rax
    addq $4088, %rax  # Near top of stack
    movq $func1, (%rax)  # Put func1 address on stack as return address
    movq %rax, ccb1+8(%rip)  # Store rsp in CCB (pointing to return address)
    movq $func1, ccb1+64(%rip)  # Store func1 address in CCB

    # Initialize coroutine2- set up stack with return address to func2
    leaq stack2(%rip), %rax
    addq $4088, %rax  # Near top of stack
    movq $func2, (%rax)  # Put func2 address on stack as return address
    movq %rax, ccb2+8(%rip)  # Store rsp in CCB (pointing to return address)
    movq $func2, ccb2+64(%rip)  # Store func2 address in CCB

```

הערה. נושא זה לא מסוכם (בעניין יותר קל לראות את המציגת). - מציגת של *couroutines* בעמוד

.18 – 19

הפונקציה *pushfq* דוחפת למחסנית את כל *Rflags*. פקודה שכדי להכיר וברור מדוע עוזרת בMOVED של *.state* בMOVED של *:state* כך משמרים

```

scheduler:
    # Save scheduler context (scheduler acts as a coroutine too)
    movq current_ccb(%rip), %rax
    movq %rbx, (%rax)  # Save rbx
    movq %rsp, 8(%rax)  # Save rsp
    movq %rbp, 16(%rax)  # Save rbp
    movq %r12, 24(%rax)  # Save r12
    movq %r13, 32(%rax)  # Save r13
    movq %r14, 40(%rax)  # Save r14
    movq %r15, 48(%rax)  # Save r15
    pushfq  # Save flags
    popq 56(%rax) # Save flags to scheduler ccb
    movq (%rsp), %rcx  # Save return address (rip)
    movq %rcx, 64(%rax)

```

לסיכום:

מיומש *Coroutines* מותבצע על ידי יצירת בלוק בקרה(*CCB*) לכל פונקציה, השומר את ה-*State* המלא שלו: רגיסטרים, דגלים(*Rflags*) ומחסנית ייעודית. התוכנית מבצעת *Interleaving* – הרחאה לシリוגן על ידי הגדרת *Quantum*, מונה פקודות שקובע מתי פונקציה תעצור ותקרה ל-*ContextSwitch* בעת הוויתור, מותבצע *RSwitch* ב-*CCB*, וה-*Yield* מוחלף

כדי לטעון את המצב של המשימה הבאה. המעבר möglich "זוגוג" בין Möglichות כך שהמשתמש מקבל תחזקה של עבודה במקביל. בסיום כל משימה מעודכן דגל *Done*, המאותת להפסקה *Scheduler* להפקות להקצותות לה זמן ריצה.

9.2 תרגול *syscall*

היא קריית מערכת *syscall*. מדובר בכלים שימושיים לתוכנית מחשב וגיליה לבקש שירותים ממערכת הליבה - *kernel*. מערכת הפעלה צריכה להגן על משאים כמו החומרה, וכן תוכנית משתמש לא יכולה לגעת ישירות בדיסק הקשיח או בזכרון של תוכנית אחרת - היא חייבת לעבור דרך השומר: השומר הוא מערצת הפעלה. ישנו שתי רמות, המעבד מדבר למשחק שמנשר בין תוכניות משתמש למערכת הפעלה. עובד בשני מצבים עיקריים:

1. מצב מוגבל בו רצوت האפליקציות שלו.
2. מצב עם גישה מלאה לחומרה. קריית מערכת *syscall* היא הנקודת שבה התוכנית עוברת מ*user mode* ל*kernel mode*.

איך המערכת יודעת איו פועלה לעשות?

לכל קריית מערכת ב*Linux* ישנו מספר ייחודי. למשל *open* של קובץ הוא מס' 2 וסגירה *close* הוא מס' 3. כמו כן, באסמבלי כשןקרה לפונקציה אנחנו נctrיך להעיבר לה בריגיסטרים:

1. דרך *rax* את מס' הייחודי של קריית המערכת.
2. דברים שקשורים לפונקציה עצמה - כרגע, דרך *rdi, rsi, rdx, r10, r9, r8*
3. קרייה *syscall* (לא *call* כמו בפונקציה רגילה!).

ערך החזרה יהיה אל *rax* אם המספר חיווי: הצלחנו. אחרת: נקלט 1 – הייתה שגיאה. כדי להבין מהי השגיאה בדיק, בודקים משתנה גלובלי בשם *errno*.

כעת, נבין מה קורה "מתוחת למכסה המגנו" של מערכת הפעלה לאחר שביצענו קרייה.

1. מערכת של מצביעים: טבלת קריאות המערכת היא למעשה מעשה מערך שבו כל תא מכיל כתובות בזכרון של פונקציה כלשהי.
2. הפרדה בזכרון: הזכרן מוחולק *user space* (שם רצوت האפליקציות של היוזר) ו-*kernel space* – שם נמצא מערכת הפעלה וטבלה זו.
3. תהליך חיפוי: המערכת משתמשת במס' שטמנו ב*rax* כדי Indeks למערך בשיביל למצוא את הפונקציה המתאימה.
4. המעבר: ברגע שהקרייה מתבצעת, המעבד עובד ל*kernel mode* ו קופץ שירותות כתובות *Table[Syscall_id]*

בידם המערכת מנהלת קבצים?

1. *File descriptors (FD)*: מדובר במספר שלם שמייצג קובץ פתוח. המספרים אלו ספציפיים לכל תהליך ונמצאים בטבלה הפנימית שלו.
תמיד לכל תהליך ישנו שלושה ערוצים שפתוחים אוטומטית:
 מס 0: עבור *stdin* (קלט סטנדרטי – מקלדת)
 מס 1: עבור *stdout* (פלט סטנדרטי – מסך)
 מס 2: עבור *stderr* (שגיאות – מסך)

דוגמאות ראשונה:

נתבונן בדוגמה מטה השימושית (שאמצעוותה נלמד עוד על *syscall*):

1. תחביר: הפונקציה `open` מקבלת את שם הקובץ - `filename`, דגלים להגדרת אופן הפתיחה, וההרשאות במקורה של יצרת קובץ חדש. נבהיר - לא תמיד יהיה את החלק השלישי, זה תלוי בהרשאות שנתנו לנו.

2. ההרשאות: `O_create` יוצר קובץ אם הוא לא קיים, `O_WRONLY` מציין כתיבה בלבד, אם הקובץ כבר קיים, `O_TRUNC` מוחק את תוכן שלו (הקובץ עדין נשאר).

3. המספר 0644 מייצג הרשאות לינוקס סטנדרטיות. לבסוף: תמיד תמיד - צריך לבצע `close(fd)` בשביל לסגור את המשאב.

הערה חשובה: נבחין כי מהרגע שביצענו קובץ, הוא ייקבל `fd` ואנחנו נתיחס אליו בשם זה מעתה ואילך. לכן שנסגור `close(fd)` נעשה זאת עם `fd` שהתקבל אל משתנה `.fd`.

```
#include <unistd.h>
#include <fcntl.h>

int main() {
    const char *filename = "example.txt";
    int fd = open(filename, O_CREAT | O_WRONLY | O_TRUNC, 0644);

    // Simplified: no error checking or further operations
    close(fd); // Always clean up resources
    return 0;
}
```

כעת נתבונן בקלט קוד זהה לחלוtin, רק הפעם באסמבלי:

1. שם הקובץ נשמר ב-`rax`, `data`, `open` מקבל את הערך 2 שמייצג את `rsi`, `open` מקבל את הדגלים הרלוונטיים, 0101 מייצין אכן את הדגלים הרלוונטיים - קריאה וכ כתיבה, `rdx` מקבל את `mode` = 0644 לבסוף, `fd` חוזר ב-`rax`, מעבירים אותו אל הקריאה למערכת הבאה, `close`, שהרי המספר שלא הוא .3

```
.section .data
filename: .asciz "example.txt"      # Null-terminated string for the filename

.section .text
.global _start

_start:
# Syscall to open the file
mov $2, Rax          # syscall number for open (2)
lea filename(%rip), Rrdi # filename pointer
mov $0101, Rsi        # O_CREAT | O_WRONLY | O_TRUNC (flags)
mov $0644, Rdx        # mode (permissions)
syscall              # invoke syscall

# File descriptor is now in Rax, no error checking

# Syscall to close the file
mov $3, Rax          # syscall number for close (3)
mov Rax, Rrdi         # use the file descriptor returned by open
syscall              # invoke syscall

# Exit syscall
mov $60, Rax          # syscall number for exit (60)
xor Rrdi, Rrdi        # status code 0
syscall              # invoke syscall
```

כיצד נטפל בשגיאות ?`syscall`

קריאהות המערכת שישתיימו בכשלון ייחרו 1-. אק, כאשר קריאה נכשלת נרצה לדעת מודיע היא נכשלת. מערכת ההפעלה מעדכנת משתנה גלובלי בשם `errno` עם קוד שגיאה ספציפי שמייצג את סיבת הכשלון. ישן פונקציית עזר שיעוזרות לנו: כמו `perror()` שמדפסה למסך מחרוזות שלחה, ביצירוף התיאור של השגיאה שהתקבלה ע"י `errno`, וכן `strerror()` שמחזירה מחרוזת שמתארת את קוד השגיאה.

:`fork`
תהליך היא תוכנית שנמצאת בביצוע. בኒוגד לקובץ הרצה שיושב על הדיסק - תהליך חי בזיכרון. תהליך יכול להיות במצבים שונים: רץ, ממתין וכו'. בלינוקס, תהליכיים נוצרים במבנה היררכי. תהליך קיים יוצר תהליך חדש (ילד).
ישנם שני מזהים: `PID` הוא מזהה תהליך ייחודי, `(Process ID)`, וכן `PPID` הוא מזהה תהליך ההורה.

הפקודה *fork*: משכפלת את התהליך הנוכחי, לאחר הקוריאה ישנים שני תהליכי – אב והורה, שרצים במקביל ומשיכים מאותה הנקודה בדיק. ערך החזרה הוא הדרך להבדיל ביןיהם: בתהליכי הילד *fork* מוחזירה אפס, בתהליכי הורה () מוחזירה את *PID* של הילד שנוצר. במקרה של שגיאה: חזרה 1 – בקוד, נשתמש ב*if*, על ערך החזרה בשבייל להבדיל ביןיהם. לאחר תהליכי *fork*, שני התהליכים הם העתיקים כמעט זהים (מלבד ה-*PID* וה-*PPID*). הם חולקים את אותן כתובות ייכרעו וקצתם כתווים ברגע השכפול, אבל מדובר בשני מרחבי זיכרונו נפרדים – שינוי משתנה אצל הילד לא ישפיע על הורה.

משפחה הפונקצייתית *exec()*: אם () יוצר עותק של התהליך המקורי, משפחת *exec()* המשמשת כדי להחליף את התוכנית הנוכחיית בתוכנית חדשה למורי. הפעולה – התהליך הנוכחי (הילד לרוב) טווען קובץ הרצה חדש לתוך הזכרון שלו ומוחליל להריצץ אותו. מה קורה באזרען? כל הקוד והנתונים הקיימים נמחקים ומוחלפים בחדשים (אצל הילד), *PID* לא משתנה. אם *exec* מצליח – הוא מעולם לא יחזיר לתוכנית המקורית – כי היא לא קיימת באזרען, הוא חוזר רק אם הייתה שגיאה.

הפקודה *wait*: בתור הורים אחראים – علينا לדעת מה קרה ליד שלנו. המטרה: לගרים לתהליכי הורה לעצור ולהמתין, עד אליו יש אחד MILFIZ יסיט את ריצתו. זה מונע מצב שבו ההורה מסיים לפניו הילד או שhayild הווקן לוומבי.

תהליכי זומבי: כאשר הילד מסיים את ריצתו, הוא לא נעלם למורי מהמערכת עד שההורה קורא *wait()*. בתקופת הביניים הזה, הוא תופס מקום בין תהליכי ונקרא זומבי.

10 הרצתה אחרונה: נושאים מתקדמים

10.1 היסטוריה של ארכיטקטורת המחשב

בתחילת הדרך, שנות ה-40 והחמשים של המאה הקודמת, הדור הראשון של המחשבים – בני אדם הפעילו את המחשב, ברמת להפעיל ולכבות בטיטים. זה היה יקר מאד דאי. ברמת מדיניה קונה מחשבה – לא אוניברסיטה ובטע לא אדם פרטיו. המחשבים דאי היו בעלי יכולת מינימיליסטית מאד, ועשה הרבה טעויות, וכן היה מורכב מאלפי נוריות: נורות נשروفות, וזה הרובה בלאי. אך עדין – לפני זה לא היה כלום ולכן מדובר היה בפריצת דרך. בעיקר השתמשו בו דאי לסימולציות של נשק. לא הייתה מערכת הפעלה (האנשים הפעילו), לא היו שפות תכנות כי לא היו תוכניות, אין זכרו כי ישנים רגיסטרים.

בஹשך הדרך, הבינו שרוצים לשומר קוד בתוכה המחשב: הגע מחשב *EDVAC*. בשנת 1952 התקדמנו אל טיפ מגנטית – האב הקדמון של דיסק מגנטי, וכעת על הטיפ היה ניתן לרשום *data, output* ואף קוד. כיצד הדפיסו? עם מדפסת. ממש התקדמו לטרניזיטור – 1955 – 1965, מה היה הבעה בו? התעסקות עם הטיפים, רצוי להפטר מזה.