

מבנה מחשב - סיכום הרצאות למבחן

10 בדצמבר 2025

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

תוכן עניינים

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

| | | | | |
|----|-------|--|-----|--|
| 23 | | <i>Jump Table</i> | 3.6 | |
| 25 | | <i>GDB</i> | 3.7 | |
| 26 | | הרצאה 4 | 4 | |
| 26 | | <i>Assemble process</i> | 4.1 | |
| 27 | | <i>ELF relocatable format</i> | 4.2 | |
| 28 | | <i>ELF executable format</i> | 4.3 | |
| 29 | | 4.3.1 כיצד כותבים וירוס? | | |
| 30 | | <i>Disassembled</i> | 4.4 | |
| 30 | | <i>Linking process</i> | 4.5 | |
| 31 | | <i>Position Independent Code</i> | 4.6 | |
| 32 | | תרגול | 4.7 | |
| 32 | | 4.7.1 ארגומנטים ב- <i>STACK</i> | | |
| 32 | | 4.7.2 <i>Variadic Functions</i> | | |
| 33 | | 4.7.3 קבלת ארגומנטים בשורת ההרצה | | |
| 33 | | 4.7.4 <i>Flow Control</i> | | |
| 34 | | הרצאה 5 - <i>memory hierarchy</i> | 5 | |
| 34 | | 5.1 הקדמה | | |
| 35 | | 5.2 <i>cache</i> | | |
| 36 | | 5.3 <i>organization of a cache Memory</i> | | |
| 39 | | תרגול 5 | 5.4 | |
| 39 | | 5.4.1 <i>static linking</i> | | |
| 40 | | 5.4.2 <i>(Position Independent Code) PIC</i> | | |
| 40 | | 5.4.3 <i>PLT</i> | | |
| 41 | | 5.4.4 <i>Patching</i> | | |
| 41 | | הרצאה 6 | 6 | |
| 41 | | 6.1 <i>miss</i> | | |
| 43 | | 6.2 <i>RAM structure</i> | | |
| 45 | | 6.3 <i>Disk structure</i> | | |
| 45 | | תרגול 6 | 6.4 | |
| 47 | | הרצאה 7 | 7 | |
| 47 | | 7.1 <i>cache friendly code</i> | | |
| 48 | | 7.2 <i>Pipeline friendly code</i> | | |
| 52 | | 7.3 <i>RAM friendly code</i> | | |
| 52 | | 7.4 <i>compiler & optimizations</i> | | |
| 52 | | הרצאה 8 | 8 | |
| 52 | | הרצאה 9 | 9 | |
| 52 | | הרצאה 10 | 10 | |
| 52 | | הרצאה 11 | 11 | |
| 52 | | הרצאה 12 | 12 | |

1 הרצאה 1

1.1 מבוא לקורס

הקורס יתמקד בשני תחומים:

1. מבנה מחשב - חומרה

2. שפת מחשב שנוגעת ישירות בחומרה - *Assembly*

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

אחר החישוב הזה יתקע. זו שפה שהיא כמעט שפת מכונה" - *Low Level*.

האם $x^2 \geq 0$? לא בהכרח - במתמטיקה כן, בוודאי ב \mathbb{R} . בעולם התאורטי, זו טענה נכונה. במציאות: ראינו כבר כי בהינתן מס' מאוד גדול, למשל $x = 1410065407$ נקבל כי $x^2 = -11158137855 < 0$, מדוע? כמו שנלמד במבוא - יש *overflow* והייצוג הופך לשלילי. אז כיצד נתמודד עם זה? נדון בזאת במהלך הקורס.

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

וכאן הדגש: בתאוריה המתמטית - לא ייתכן שיקרו בעיות כאלו. בעולם האמיתי, במחשב: זה לגמרי קורה. ועלינו ללמוד כיצד להתמודד עם זה.

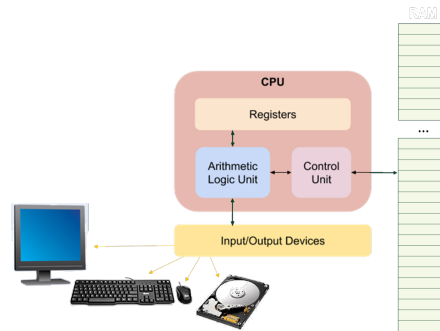
נתבונן בקוד הבא:

```
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 האינטואטיבי, בו אנחנו רצים ללא קפיצות זה שלוש. הקוד הזה הרבה פחות יפה - אבל בהמשך נבין (CPU) שהוא רץ הרבה יותר מהר: וזה כל מה שחשוב, יעילות.

1.2 COMPUTER STRUCTRE

המחשב בנוי מ-CPU, חומרה (Input/Output Devices) וזכרון (RAM).



ה-CPU הוא ה"מוח" של המחשב, הוא יחידת העיבוד המרכזית.

ל-CPU יש יכולת מתמטית חישובית - *Arithmetic Logic Unit*: ALU.

ל-CPU יש זכרון גם כן - *Registers*: בלעדיהם הוא לא מסוגל לעשות כלום והוא תלוי בהם.

הזכרון - RAM: מורכב מבייטים. כל בייט מורכב מ-8 ביטים. מעין מערך שיוכל לספור עד $2^n - 1$ ערכים. כל בייט בזכרון הוא עם ערך כלשהו - גם אם שמנו את זה שם וגם אם לא (יקבל ג'אנק). הביט הימני נקרא *LSB* והביט השמאלי נקרא *MSB*.

Word - שני בייטים רציפים.

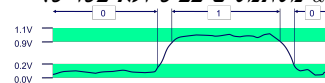
Long/dword - ארבעה בייטים רציפים.

qword - שמונה בייטים רציפים.

1.2.1 מדוע המחשב שלנו בינארי? למה שהמחשב לא יהיה טרינארי?

כאשר מעבירים מידע ממקום למקום (בתוך המעבד) נשלח *signal* חשמלי. אנחנו רוצים לתרגם את המידע שיש בסיגנל החשמלי לביטים. סיגנל מגיע עם רעשים - כמו כאן בתמונה מטה. מגדירים טווח של עוצמת הסיגנל: בין 0 ל-0.2V הוא מתפרש כביט 0, בין 0.9 ל-1.1V מתפרש כ-1, ובכל השאר הוא מתפרש כמצב מעבר.

באופן תאורטי - יכלו את מצבי המעבר להגדיר כמצב השלישי - ואז לעבוד עם ביט" שלישי, במצב טרינארי: מהר מאוד ירדו מרעיון זה, כיוון שהיו הרבה יותר רעשים ותנודות ואז במקום לחשוב שקיבלת מס' 8 קיבלו 9. נוצרו הרבה בעיות כתוצאה מרעיון זה - ולכן בשביל להבטיח את תקינות ה**data** משתמשים בביט ולא בטריט.

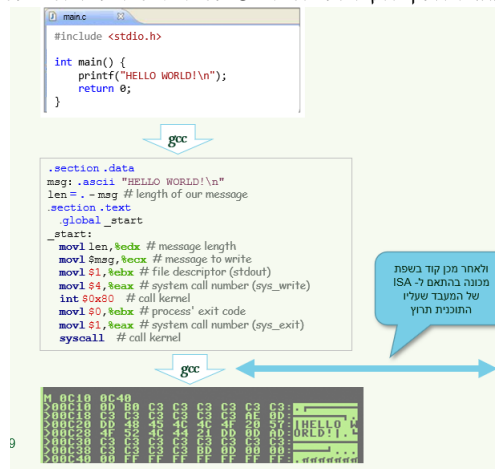


1.2.2 Instruction Set Architecture - ISA

(**מרינה אמרה שהיא שואלת לפעמים במבחן מה זה**). לכל דגם של *CPU* יש אוסף פקודות שהוא יודע לבצע. אוסף פקודות נקרא שפת מכונה. לכל פקודה כזו קיימת פקודה מקבילה שקיימת בשפת אסמבלי. ה**ISA** הוא ספר" שמרכז את כל הפקודות שאותה ארכיטקטורה *CPU* יודע לבצע. ה**ISA** של דגמים שונים של מעבדים יכול להיות שונה. בהינתן שנדע את הספר הנ"ל - נדע איזה פקודות נוכל לכתוב בשביל לכתוב את הקוד שלנו. הפקודות בספר יהיו כתובות הן בשפת אסמבלי והן בשפת מכונה.

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

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



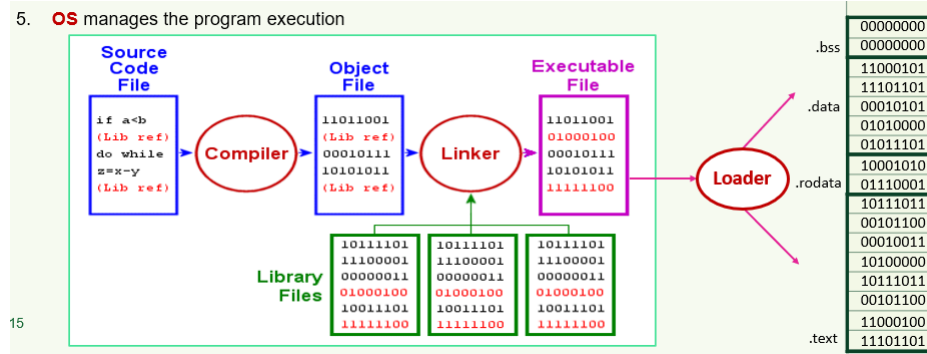
1.3 תהליך ההרצה ותמונת הזכרון

1. **שלב ראשון:** כתבנו קוד. נקרא לו *Sorce*
2. **שלב שני:** הקומפיילר מבצע תהליך קומפילציה, מזהה שגיאות קומפילציה והופך אותו לשפת מכונה.

3. **שלב שלישי:** הלינקר, מבצע תהליך קישור בין הקוד שלי לקודים אחרים שקיימים בספריות אחרות בהם אנחנו השתמשנו בקוד, זהו קוד שכבר מוכן ומקומפל - והוא מאחד אותם לקובץ יחיד שיקרא *Executable* שמוכן לקמפול.

4. **שלב רביעי:** בטרמינל, אנחנו כותבים למשל *a.exe*. >: למעשה, מה שקורה הוא שאנחנו אמרנו למערכת ההפעלה - קחי את הקבצים שכתבתי לך ב*Executable*, תשתמשי ב*Loader* *Loader* : ("טוען" - תפקידו לטעון את הקובץ לזכרון. **ראשית** הוא מוודא שקובץ זה ניתן להרצה ע"י מערכת ההפעלה שלנו. **שנית**, הוא מפרסר ("חותך" ושם בכל מקום בזכרון מה צריך לשבת שם" - בדיוק כמו שבבדיקת מבחן, הבודק מדלג על אזור ההוראות. למעשה ה*Loader* יודע על מה עליו לדלג ומה הוא צריך לקרוא, את מה שהקומפיילר והלינקר עזרו לו לקבל, וזה הפרסור) את מה שכתוב לו, והוא צריך להבין באיזה מיקום של הקובץ מופיע *data* ולשים את זה באזור *data* בזכרון, להבין מה צריך להכנס ל*text* בזכרון ולשים את זה באותו אזור בזכרון וכן הלאה. תפקיד נוסף שלו - הוא לאתחל את המשתנים בהתחלה.

כעת לאחר שהשתמשנו ב*Loader* קיבלנו *Process Image*: מכלול של זכרון (תמונת זכרון) שמוקצה לטובת התהליך ש*Loader* אחראי לו. כלומר, כל הזכרון שמוקצה לטובת התהליך במהלך ריצת התהליך.



bss: משתנה גלובלי (מוקצים ב*data*). כאשר כתבנו משתנה *int x* ולא נתנו לו ערך.
data: משתנים גלובליים שמאותחלים כבר עם ערכים.
rodata: משתנים גלובליים סטטיים שהם כבר *const* והם כעת רק נקראים.
text: אזור הקוד של התוכנית. כמעין ספר". בעת קריאת קוד קוראים מלל.

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

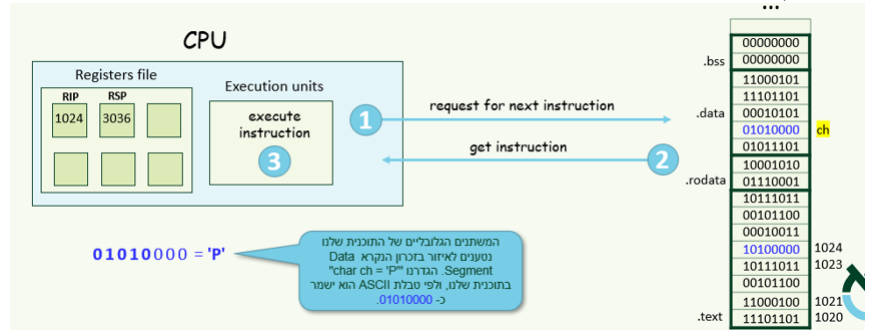
1.3.1 איך CPU יודע מה רצף ביטים מייצג?

ה*CPU* מבקש *instruction* הבא, מקבל אותו ומבצע אותו. ברגע שהוא סיים את ההוראה הנוכחית, הוא מקבל את ההבא ומבצע אותו. הוא מאוד מורכב - אך בפועל הוא עובד בצורה פשוטה. ה*Loader* מכיל מידע אודות היכן ה*main* ב*text*. הוא יודע את ה*instruction* הראשון שהתוכנית צריכה לבצע - בעברית: נקודת כניסה", ה*Loader* יזהה במהלך תהליך הפרסור את המקום הראשון שממנו התוכנית תתחיל.

RIP (*Instruction Pointer*) רגיסטר. הוא הרגיסטר שה*CPU* לא יכול בלעדיו. זהו רגיסטר שמצביע על הבית הראשון של הפקודה הבאה לביצוע. ה*Loader* שם בתוך הרגיסטר ממש *RIP* את הכתובת הזו במהלך תהליך הפרסור. (הערה - גם באסמבלי נוכל לגשת לרגיסטרים בזכרון, מה שאי אפשרות בשפות עילית).

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

מתקדם לכתובת הבאה בזכרון, הוא מקדם את המקום בזכרון באחד (אל הבייט הבא). אם הוא מקבל פקודה - הוא חוזר לשלב 3, ומבצע את הפקודה. אחרת הוא ממשיך להעלות אחד בכל שלב ומתקדם בזכרון.



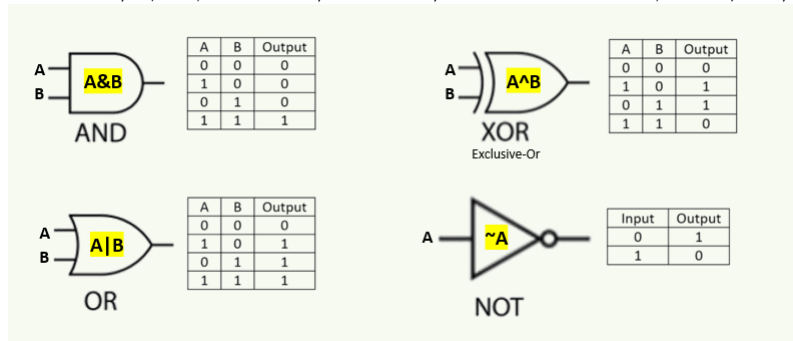
המעבד קורא תחילה (prefix) של פקודה, מזהה שזה פקודה בזכות ISA ומפענח אותה.

RSP (Stack Pointer) - רגיסטר שנמצא בתוך ה-CPU. מצביע על תחילתו (כיוון שאם הערך גדול מדי, לפי מבוא הוא נשמר מלמעלה ללמטה בזכרון) של הערך האחרון שנכנס למחסנית. המחסנית חשובה מאוד כיוון שיש בה משתנים לוקלים, *activation frame* וכתובת חזרה וכן ארגומנטים שפונציה מקבלת (אם וכאשר).

הערה: בתוך ה-CPU ישנם מס' רגיסטרים שיושבים באזור *Registers file*. למשל: *RSP*, *RIP*. המשתנים הגלובליים של התוכנית שלנו נטענים לאזור בזכרון הנקרא *data*, אם הגדרנו אות 'p' למשל בתוכנית - לפי טבלת ASCII הוא ישמר בהתאם אליה.

1.4 Bit Level Operation

בהיתן אוסף ביטים, נפעיל פעולת ביטויז בין ביטים. ישנן מס' פעולות, כדקלמן:



נשים לב כי $true = 1$ וכן $false = 0$ כמובן.

פעולת *Shift Right*: מזיז ביטים ימינה.

פעולת *Shift Left*: מזיז ביטים שמאלה.

דוגמה. **יצירת קבוצה באמצעות ביטויז.** נרצה לייצג וקטור באורך n באמצעות ביט: $\{0, \dots, n-1\}$. כל ערך בוקטור יכול להיות משוייך לקבוצה ויכול להיות שלא. הקבוצה תהיה A . אם $i \in A$ אזי $a_i = 1$. כלומר - אם איבר בקבוצה מיקומו בוקטור יהיה 1.

אם נרצה להוסיף ערך לקבוצה - נבצע פעולת *or*: מדוע? נוסיף ביט עם הערך 1 על 1 וכל השאר אפסים, וכעת כשנעשה *or* נקבל קבוצה חדשה עם 1 בתוכה.
אם נרצה למצוא את הקבוצה המשלימה - \bar{A} : נבצע *not* על הביט.
לחיתוך שתי קבוצות - נשתמש באופרטור *and*, **ולאיחוד** שני קבוצות - נשתמש באופרטור *or*.

חשוב לדעת: פעולות בינאריות הן הפעולות המהירות ביותר שניתן לבצע.

1.5 תרגול 1

ישנם מס' כלים שנועדו להקל את החיים של המתכנת.
Text Editor: *Nano, Notepad, Vim, Emacs*. וכו'. בעיקר כותבים בהם את הקוד.
IDE: סביבת עבודה שגם ניתן לכתוב בה את הקוד וגם להריץ אותו, למשל: *Clion, VSCode, VS*.
Compiler: סט כלים שמכיל בתוכו כמה שלבים שהשלבים האלו יחד מעבירים תוכנית שכתבנו מקובץ קוד לקובץ הרצה שנוכל להריץ על גבי המחשב. עובדים עם קומפיילר *GCC*.
Project Builder: כלי ש"שומר" על סדר בתהליך יצירת קובץ ההרצה, הוא מוודא שכל הקבצים בתוך הפרויקט מועברים כמו שצריך במהלך יצירת קובץ ההרצה. כמו *Makefile, Cmake*.
Computer Interfaces: דרכים בהם ניתן לבצע משימות באמצעות המחשב שלנו. כמו *GUI, GUI, Shell* למשל משמש לפתיחת וסגירת חלונות.

Shell:

תוכנה שבמסגרתה אנחנו יכולים להכניס פקודות ומערכת ההפעלה מריצה אותם בפועל לאחר שמקבלים פלט על הפקודה. פקודות מרכזיות:

1. *ls* - מראה לנו איזה קבצים/תיקיות קיימים בתוך התיקיה שאנחנו נמצאים בה.
2. *pwd* - *present working directory*: מראה את התיב בו אנחנו נמצאים כרגע. למשל *.guy/desktop*.
3. *mkdir* - מאפשר ליצור תיקייה חדשה
4. *cd* - מאפשר לעבור בין תיקיות.
5. *touch* - יוצרת קובץ חדש
6. *cp* - מאפשרת להעתיק קובץ
7. *rm* - מאפשרת למחוק קובץ
8. *history* - מראה לנו את הפקודות שהרצנו עד כה.

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 שמונע את השימוש בעכבר.

מדוע זה רלוונטי אלינו? אם מתחברים מרחוק לשרת לינוקס כלשהו, למשל לשרת של המחלקה: אין לנו אפשרות *GUI* של פתיחה וסגירת חלונות. אם נרצה לערוך קובץ לשרת, נוכל לפתוח אותו עם *Vim*.

ישנה גרסה מתקדמת של *Vim*: *neoVim*, שנכיר.

MakeFile:

כלי שמורכב מכמה וכמה חלקים וחוקים, שהמטרה של כל החוקים הללו יחד היא לוודא שכל הקבצים הרלוונטים בפרויקט שלנו מעורבים ביצירת קובץ ההרצה. זהו קובץ שמאוד מאוד נפוץ ביצירת קבצי *C/C++*

מורכב מאוסף של *rules* שכל אחד מורכב משלושה אלמנטים:
1. *target* - שם *rules* (לרוב קובץ שנוצר כתוצאה מהרצת *command*)

2. *command* - פקודה שתרוץ במהלך הרצה של אותו *rule*.
3. *dependency* - קבצים שצריכים להיות קיימים בשביל *command* של אותו *rule* ירוץ.

כשנרץ את המילה "make" מה שיקרה הוא שה*rule* הראשון בקובץ *makefile* הוא זה שיוּרץ.

2 הרצאה 2

2.1 עוד על numbers

מספר בינארי: נזכר כי בהינתן מס' U המיוצג 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 2 כיוון ש $X + (-X) = 2^n$. כיצד עושים כאן? מכפילים את ה-*MSB* בחזקה ומוסיפים סימן שלילי ומחברים את השאר כמו במספרים חיוביים.

2.2 CPU Flags

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

Example1:

| | |
|----------|------------------|
| 01000000 | ← 64 |
| + | |
| 01000000 | ← 64 |
| ----- | |
| 10000000 | ← 128 or -128 ?? |

unsigned נקבל כי התוצאה 128, *signed* נקבל -128, מה קיבלנו כאן? ה-*CPU* בעצמו לא יודע. בחישוב *unsigned* החישוב נכון - כי אכן $64 + 64$ שווה ל-128. נשים לב כי ה-*MSB* של שני המספרים היא 0, ולכן שני המספרים הינם חיוביים. ולכן אם נקבל -128 התוצאה שגויה.

כלל: *CPU* ביצע חיבור של ביטים, ה-*CPU* לא מתעניין האם התוצאה היא *signed* או *unsigned*. *CPU* ישמור במקום מסויים האם הייתה החלפת סימן, אם לא מעניין אתכם (אתם מחשבים *unsigned*) פשוט אל תסתכלו במידע.

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

| | |
|-----------|---------------|
| 11111111 | ← 255 or -1 ? |
| + | |
| 00000001 | ← 1 |
| ----- | |
| 100000000 | ← 0 ?? |

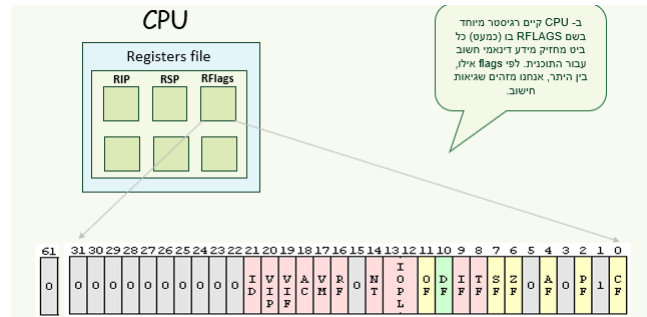
היכן ה-*CPU* רושם רישומים אלו?

Rflags: ישנו רגיסטר מיוחד שה-*CPU* שומר בשם *Rflags* - יש לו ביטים שמתעדכנים בעת חישוב שה-*CPU* עושה. הוא מעדכן את הביטים תוך כדי החישובים. אנחנו מתעניינים בביטים הצהובים. נשים לב שלכל ביט ישנו שם כאן. **ישנו ביט בשם** *CP(CarryFlag)* - יקבל אחד באשר יהיה *carry* (בחיבור או בחיסור) אחרת, יקבל אפס. יאמר (אם וכאשר) כי חישוב *unsigned* שגוי.

ישנו ביט בשם $OF(OverflowFlag)$ - הוא יקבל 1 כאשר תהיה החלפת סימן (כלומר הביט שינה סימן), אחרת יקבל ערך אפס. יאמר (אם וכאשר) כי חישוב *signed* שגוי.

ישנו ביט בשם $ZF(ZeroFlag)$ - אם תוצאת החישוב האחרון יצאה אפס הוא יקבל אחד, אחרת הוא יקבל אפס.

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



2.3 Endianness

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

Big Endian שומרים אותו משמאל לימין (שומרים את ה-*MSB* הכי למטה - כלומר נשמר במקום 00). *Little Endian* שומרים אותו מימין לשמאל (כיצד נזכור? שומרים את ה-*LSB* הכי למטה - כלומר שומרים במיקום 00).

| BIG-ENDIAN | LITTLE-ENDIAN |
|------------|---------------|
| 0x12345678 | 0x12343678 |
| 0x03 78 | 0x03 12 |
| 0x02 56 | 0x02 34 |
| 0x01 34 | 0x01 56 |
| 0x00 12 | 0x00 78 |

אנחנו בקורס לומדים לפי *Little Endian*. (כאשר אנחנו מדפיסים בקורס משהו, זה מתבצע בהדפסה מכתובת 00 כלומר מה שיופס קודם יהיה ה-*LSB*)

2.4 Assembly

אסמבלי נוצרה בשביל לנסות להפסיק לכתוב בשפת מכונה, ולנסות לתרגם את השפה לשפת בני אדם. כשכתבו את השפה לא חשבו על נוחות. על כל פקודת מכונה, לקחו את הפקודה ו"תרגמו" אותה לשפת אסמבלי שלכאורה יותר אנושית. ומכאן המסקנה: פעולה באסמבלי=פעולה של שפת מכונה.

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

אסמבלי לא תומכת בדברים הבאים (ובמה נשתמש במקום?):

- א. *data types* (כן יש - גדלים, 4 בייט, 1 בייט...)
- ב. מערכים (נוכל לעבוד ישירות עם RAM)
- ג. תנאים (גישה ישירה לרגיסטרים - זוהי השפה היחידה שמאפשרת זאת, ולכן **אסמבלי היא השפה הכי יעילה**).
- ד. לולאות
- ה. פונקציות
- ו. ספריות סטנדרטיות

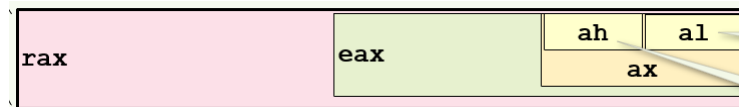
מה שנקרא - בהצלחה שיהיה לנו.

2.5 Registers

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

רגיסטרים לחישוב כללי - *general purpose registers*:

הרגיסטרים הינם $RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, R8, R9, \dots, R15$. נוכל להשתמש בהם לחישובים כלליים. הסיבה של $R8, R9, \dots, R15$ אין שמות הוא שהם נוספו בהרחבה של אסמבלי, שכבר הבינו שלשמות אין משמעות. בעבר היה חסר רגיסטרים ולכן בתוך רגיסטר היו מאחסנים שני נתונים. למשל ברגיסטר ax היה שני תכני מידע שונים - ah, al ובהתאמה $High, Low$. בהמשך, הרחיבו את הרגיסטר ל-32 ביט ולא נתנו שם להמשך המידע וגם בהרחבה ל-64 ביט לא הוסיפו שם. כך נראה רגיסטר - אוסף בזכרון של 64 ביטים.



ישנם המון רגיסטרים - בקורס אנחנו נשתמש ברגיסטרים שצוינו לעיל, ברגיסטר RIP וברגיסטרים של $Rflags$. נזכר ברגיסטרים הבאים:

RIP (Instruction Pointer) רגיסטר. הוא הרגיסטר שה-CPU לא יכול בלעדי. זהו רגיסטר שמצביע על הבית הראשון של הפקודה הבאה לביצוע. ה- $Loadern$ שם בתוך הרגיסטר ממש RIP את הכתובת הזו במהלך תהליך הפרסור. (הערה - גם באסמבלי נוכל לגשת לרגיסטרים בזכרון, מה שאי אפשרות בשפות עילית).

RSP (Stack Pointer) - רגיסטר שנמצא בתוך CPU. מצביע על תחילתו (כיוון שאם הערך גדול מדי, לפי מבוא הוא נשמר מלמעלה ללמטה בזכרון) של הערך האחרון שנכנס למחסנית. המחסנית חשובה מאוד כיוון שיש בה משתנים לוקלים, $activation frame$ וכתובת חזרה וכן ארגומנטים שפונקציה מקבלת (אם וכאשר).

2.6 Basic Instructions & Data types

שורת קוד טיפוסית באסמבלי נראית כך: `movb $0x61, al`. הפקודה הזו מכניסה את הערך $0x61$ לתוך הרגיסטר al . לאחר הפקודה, הרגיסטר al יראה כך:

$$al = 01100001$$

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

נשים לב: \$ חשוב מאוד, ואם לא נכתוב אותו יחשבו שאנחנו מדברים על מקום בזכרון. אם נכתוב עם \$ יהיה ברור כי הכוונה לערך.
כשנרצה לפנות לרגיסטר - נעשה זאת עם % בפנייה לפני השם של הרגיסטר.

הפעולה *mov*: פקודת הזזה. מוסיפים לסוף הפקודה סיומת, בהתאם לגודל הטיפוס שאנחנו מזיזים. אנחנו למעשה מבצעים העתקה של ערך למקום אחר בזכרון.
1. הפקודה *movb*: פעולת הזזה שעובדת על *bytes*, מזיזים בייט ממקום למקום. בייט הוא 8 ביטים.
2. הפקודה *movw*: פעולת הזזה שעובדת על *words*, מזיזים *words* ממקום למקום בזכרון. *word* הוא שני בטים.
3. הפקודה *movl*: פעולת הזזה שעובדת על *longs*, מזיזים *long* ממקום למקום. *long* הוא 4 בטים. (כמו *int* ב C)
4. הפקודה *movq*: פעולת הזזה שעובדת על *qword*, *qword* הוא 8 בטים. (כמו *Long* ב C)

2.6.1 השוואה לשפת מכונה

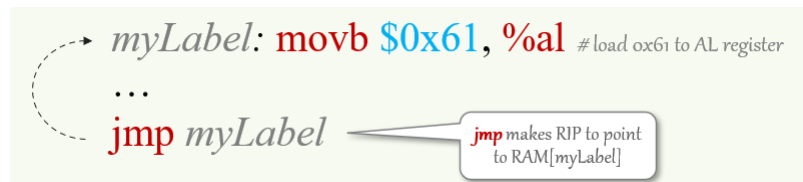
אותה דוגמה מלמעלה, בשפת מכונה תתורגם להיות השורה הבאה:

0xB061

61 זה הערך המספרי אותו צריך להכניס, 0x מעיד שאנחנו בהקסהדצימלי, ו B0 מספר לו שהוא צריך לבצע הכנסה של ערך לתוך רגיסטר *al*. (מאיפה אנחנו יודעים זאת? נפתח ISA).
Immediate הוא ערך נומרי. *Imm8* הוא ערך נומרי בגודל 8.

2.6.2 *jmp* ו *Labels*

בכדי לשים הערות על הקוד באסמבלי - נוכל להשתמש בנקודה פסיק או ב#. נוכל להוסיף לכל פקודה *Label*. *Label* זה מיקום של הפקודה. כאשר נוסיף את הלייבל, הקומפיילר יתרגם אותו לכתובת של הפקודה. **מאחורי הקלעים:** הקומפיילר מוחק את הלייבל ומכניס את הכתובת. בשביל מה נצטרך לייבלים? כעת, נוכל לבצע *jmp* ולהגיע אל הפקודה הזו, ממקום אחר בקוד.
jmp משנה את רגיסטר *RIP* אל *RAM[Label]*.



2.6.3 *Assembler directive*

הנחיה שאנחנו נותנים לקומפיילר של אסמבלי, ששמו *Assembler*.
נתבונן על הפקודה הבאה -

```
buffer: .skip 4 # reserve 4 bytes
```

קומפייטר, תלך לזכרון, ספציפית ל-*section* של משתנים גלובליים, בפרט אל *bss* (המשתנה לא מאותחל), ותקצה לי שם רצף של 4 בתים. מעכשיו, נוכל להתייחס אל 4 הבתים הללו כ-*buffer*. זה כמו מקביל ליצירת משתנה ללא אתחול, וכן מקביל למערך של *chars*. וכן מקביל ליצירת מערך של *shorts*, או מערך של *short* ושני *char* - וכן כל קבוצה של משתנים שנסכמת לגודל של 4 בתים. כעת, 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 מתורגם קוד בשפת *C* לאסמבלי: בתחילה אנחנו מקצים בזכרון 20 בתים (5 פעמים 4 בתים של *int*), אנחנו מאתחלים משתנה ברגיסטר *rbx* שיקבל את הערך אפס שיציין למעשה את $i = 0$. משם אנחנו מתחילים לולאה, באסמבלי יש **פקודת השוואה גנרית**: *cmpq*, באסמבלי יש פקודת השוואה אחת (ולא כמה כמו $\langle \rangle$), מה שהפעולה עושה הוא לחשב את החיסור של *source* מה-*destination*, כלומר הפקודה הבאה תתורגם להיות

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

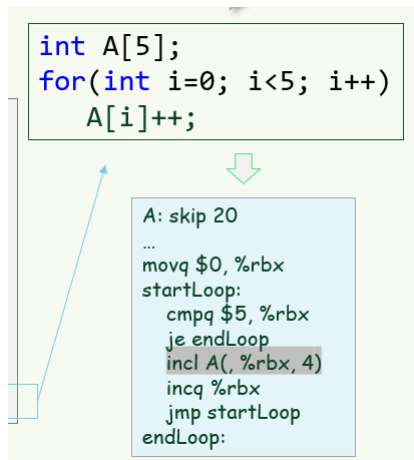
כלומר נבצע $rbx - 5$. נשים לב שתוצאת החיסור לא נשמרת. אם נקבל כי התוצאה חיובי, המשמעות היא כי $dest > source$. אם שלילי, $source > dest$. כיצד נדע האם יש קשר של \geq , \leq ? כלומר - גם שווה, נסתכל על *ZF*.

הפעולה je: אם יש שוויון כלומר $ZP = 1$, בין שני האופרנדים שבוצעו בשורת הקוד הקודמת, אנחנו נבצע *jmp* אל ה-*Label1* שכתוב לאחר *je*. הפקודה הנ"ל ניגשת אל ה-*zeroFlag* ובודקת, אם הוא 1 היא קופצת אל ה-*Label1*.

הפקודה inc: הפקודה למעשה מבצעת *increment*, יש *incl* שמבצע זאת על *long*. וכמובן כמו *mov*, בהתאם לכל סוג משתנה. הסינטקס שלה יהיה כדלקמן

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

כאשר אנחנו כותבים סוגריים באסמבלי - אנחנו ניגשים אל הזכרון. אנחנו הולכים אל *RAM* למקום שהוא $4 * rbx + 2048$ באשר 2048 זו הכתובת של *A*. אנחנו למעשה ניגשים אל הכתובת של *A[0]* ואנחנו מגדילים אותה, בשביל להתקדם לכתובת הבאה (שכבר שמרנו כשהקצנו 02 בתים). כאשר נבצע *inc* על ערך, נוסיף %. כך למשל הפקודה *incq %rbx* תגדיל את הערך *rbx* באחד.



2.7 פקודות בסיסיות באסמבלי

ADD: הפעולה מחברת שני שלמים. נשים לב כי אסור ששני הארגומנטים של הפקודה יהיו בזכרון, אחד מהם חייב להיות *immediat* (קבוע) או רגיסטר. סינטקס יראה כך -

`addq %RBX,%RAX`

SUB: הפעולה מחסרת שני שלמים. נשים לב כי אסור ששני הארגומנטים של הפקודה יהיו בזכרון, אחד מהם חייב להיות *immediat* (קבוע) או רגיסטר. סינטקס -

`subq %RBX,%RAX`

NOT: פעולת ביטוויז. המשלים ל"1" - הופך ביט 1 לביט 0 וביט 0 לביט 1. וסינטקטית - `notb %AL`

NEG: ביטוויז. פעולת המשלים ל-2, הופך את כל הביטים ומוסיף 1 לתוצאה. `notb %AL`

AND: פעולת ביטוויז. ביט במיקום *i* מקבל את הספרה 1 אם שני הביטים של שני הרגיסטרים הוא 1. אחרת, מקבל אפס. סינטקטית - `andb %BL, %AL`

OR: פעולת ביטוויז. ביט במיקום *i* מקבל את הספרה 1 אם לפחות אחד מהביטים של שני הרגיסטרים הוא 1. אחרת, מקבל אפס. סינטקטית - `orb %BL, %AL`

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

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

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

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

TEST: מבצעת פעולת AND על שני אופרנדים אך לא שומרת את התוצאה - רק מעדכנת את דגלי הסטטוס. שימוש נפוץ לבדיקה - האם רגיסטר שווה לאפס:

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

למעשה ישנה פעולת AND על אותו אופרנד עם עצמו. אם הוא היה אפס, נקבל כי ZP כעת ידלק. ולכן קפוף $Label$ אם $R_1 = 0$.

SHIFT: SHL, SHR - פקודות הזזה של ביטים. בביצוע SHL כל ביט זז שמאלה, ביטים שיוצאים מהמקום נזרקים ונשמרים ב- CF , מימין נכנסים אפסים. בביצוע SHR כל ביט זז ימינה, ביטים שיוצאים מהמקום נזרקים ונשמרים ב- CF , משמאל נכנסים אפסים.

SAR, SAL - שיפט אריתמטי, בשיפט רגיל אנחנו מזיזים ומוסיפים אפסים. יתכן כי המספר 0100 (4) יקבל שיפט לשמאל, וכתוצאה מכך יהפוך למספר 1000, שהוא מייצג מספר שלילי (-16), והרי זה לא הגיוני שחילקנו בשתיים מס' חיובי וקיבלנו מס' שלילי. בדיוק זו הסיבה שנשתמש בשיפט אריתמטי -

SAL זהה לחלוטין ל- SHL . עם זאת, SAR מזיז לימין את הביטים, אך הוא משאיר את ביט הסימן בצד שמאל. כלומר: הוא לא מזיז את ביט הסימן. **דוגמה** - 1101 אם נבצע עליו SHR נקבל 0100, ממס' שלילי קיבלנו חיובי, זה לא טוב. לעומת זאת אם נעשה SAR נקבל 1100 (הסימן נשאר).

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

DIV: חילוק בין שני מספרים ב- $unsigned$, $idiv$ זה במצב $signed$.

2.8 תרגול 2

כיצד מאפסים רגיסטרים? מבצעים לרגיסטר xor עם עצמו. שורת הקוד הבא תאפס את הרגיסטר: $xorq \%rbx, \%rbx$ נהוג לאפס את הרגיסטרים בתחילת הריצה.

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

$AddressingModes : A(reg', reg'', i)$ סינטקטית, באשר $i \in \{1, 2, 4, 8\}$ החישוב כדקלמן

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

`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 סוגריים, מתייחסים לזה כ- *Addressing Modes* ומחשבים לפי החישוב לעיל. ניתן להשמיט חלק מהפסיקים, לא חובה שכולם ישתתפו.

נשים לב - ניתן להעביר *eax* (32 ביט) לתוך *rax* למשל (64 ביט), תשמור בחלק התחתון של הרגיסטר ובחלק העליון יהיה זבל. לעומת זאת אם נעשה *rax, rax* זה ישמור בחלק התחתון ויאפס את החלק העליון.

הפקודה *mov* יכולה להעביר מידע מרגיסטר לרגיסטר, ומרגיסטר לזכרון. לא יתכן מצב בו מעבירים מזכרון לזכרון בשורה אחת - לא ייתכן:

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

מה הפתרון? נשתמש ברגיסטר עזר, ואז נוכל להעביר בין כתובות.

הפקודה *movzbl, movsbl* -

ישנם סימונים שכדאי להכיר. $Z = zero, S = sign$ משמעותו היא שאם יהיה בסוף הפקודה *s*, ואנחנו מעבירים למשל *%al, %edx* אנחנו נמלא את שאר המקום שלא התמלא (כי $al < edx$) של *sign*, כלומר אם הסימן היה אחד נוסף אחדות עד *MSB*. אם בסוף הפקודה יהיה *z*, המשמעות שנמלא את שאר המקום שלא התמלא באפסים.

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

2.8.1 Signed&Unsigned

ישנם שני דרכים שונות לייצג מספרים. בשתי השיטות משתמשים בייצוג בינארי - אך מפרשים אותו אחרת.

Signed (המשלים 2): מפרש את הביט השמאלי ביותר, *MSB* כביט סימן. 0 משמע חיובי ו1 משמע שלילי. טווח הערכים יכול לנוע בין $-2^{n-1} - 1 \rightarrow 2^{n-1}$. ב-*Overflow, Underflow*: הביט של הסימן עלול להתהפך, ונקבל תוצאה לא צפויה.

- `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)

Unsigned: מפרש את הביט השמאלי ביותר כביט רגיל. מכאן, טווח הערכים הוא $0 \rightarrow 2^n - 1$.
Overflow, Underflow: זה יכול להפוך לאפס פתאום (מודולו).
 הפקודות מטה משמשות עבור *Unsigned*

- `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)

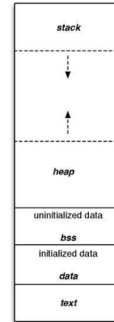
אנחנו קובעים את המשמעות של הרגיסטר - האם אני משתמש בו ככתובת או כערך. אני מחליט מה רגיסטר מחזיק.
 מה זה אומר בכלל? ב *CPU* אין דבר כזה *Singed, Unsigned* ברגיסטרים. המעבד לא יודע אם המספר שאתה שם ברגיסטר הוא חיובי שלילי או בכלל כתובת בזכרון. הוא רואה ביטים בלבד. אותם ביטים יכולים להתפרש באופן שונה בשתי השיטות, ואתה בתור מתכנת בוחר כיצד לפרש זאת. המעבד עצמו לא מבין במה אני בחרתי להשתמש, אבל הפקודות שאני בחרתי להשתמש בהם, הם אלו שמרמזות לו על הכוונה שלי. למשל אם אשתמש ב *idiv* הוא יבין שאני *signed* ואם אשתמש ב *div* הוא יבין שאני *unsigned*.

2.8.2 מבנה של תוכנית:

כעת נדון במבנה הזכרון של תוכנית במהלך זמן ריצה, כלומר איך ה *CPU* מסדר את הזכרון של התהליך.

ישנם כמה חלקים בזכרון:

text - הקוד עצמו, ההוראות שהמעבד מריץ, כל הפונקציות שכתבנו בקוד והקריאות. התוכן כאן לא משתנה בזמן ריצה (*read only*).
data - כאן נשמרים משתנים גלובליים או סטטיים שיש להם ערך התחלתי.
bss - כאן נשמרים משתנים גלובליים או סטטים שלא קיבלו ערך התחלתי. הם מאותחלים אוטומטית ל0 במהלך העלאת התוכנית לזכרון.
heap - כאן מוקצה כל הזכרון שהמתכנת מקצה ידנית, עם *malloc* וכו'. ה *heap* גדל מלמטה למעלה, כלומר לכתובות גדולות יותר.
stack - כאן נשמרים משתנים מקומיים וקריאה לפונקציות. כל פעם שנכנסים אל פונקציה נפתח *activation frame*: כל פריים מכיל כתובת חזרה, פרמטרים לפונקציה ומשתנים מקומיים. ה *stack* גדל מלמעלה למטה, כלומר לכתובות נמוכות יותר.



חשוב לזכור: הרגיסטר אשר מצביע לראש 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 בזיכרון במקום אחד, וזה מאט את התוכנית. לפעמים, כתובת לא מיושרת עלולה לגרום לקריסה.

3 הרצאה 3

3.1 $Jump\&Set$

קפיצה מותנית נעשית בהתאם לערכי ה- $flags$.

| 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 | ZF = 1 |
| JNE | Jump if not equal | ZF = 0 |
| JNZ | Jump if not zero | ZF = 0 |
| JB | Jump if below | CF = 1 |
| JNAE | Jump if not above or equal | CF = 1 |
| JNC | Jump if not carry | CF = 0 |
| JNB | Jump if not below | CF = 0 |
| JAE | Jump if above or equal | CF = 0 |
| JAC | Jump if not carry | CF = 0 |
| JBE | Jump if below or equal | CF = 1 or ZF = 1 |
| JNA | Jump if not above | CF = 1 or ZF = 1 |
| JA | Jump if above | CF = 0 and ZF = 0 |
| JNBE | Jump if not below or equal | CF = 0 and ZF = 0 |
| JL | Jump if less | SF <> OF |
| JNGE | Jump if not greater or equal | SF <> OF |
| JGE | Jump if greater or equal | SF = OF |
| JNL | Jump if not less | SF = OF |
| JLE | Jump if less or equal | ZF = 1 or SF <> OF |
| JNG | Jump if not greater | ZF = 1 or SF <> OF |
| JG | Jump if greater | ZF = 0 and SF = OF |
| JNLE | Jump if not less or equal | ZF = 0 and SF = OF |
| JCXZ | Jump if CX register is 0 | CX = 0 |
| JECXZ | Jump if ECX register is 0 | ECX = 0 |

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

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

תתורגם להיות:

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

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

חשוב לדעת ולזכור: פונקציה תמיד תחזירה את התשובה שלה אל הרגיסטר `rax`! הפעולה `setX` מחזירה את הערך שלה אל רגיסטר בגודל 8 ביטים - ולכן בלית ברירה זה יכנס אל `al`. בהמשך, משתמשים בפקודה `movzbq` ומרחיבים את `al` ל-`rax` באמצעות הוספת אפסים לאחר הערך `al`.

טבלת פעולות `:set`

| | | |
|--------------------|--------------|---------------------------|
| <code>sete</code> | ZF | Equal / Zero |
| <code>setne</code> | ~ZF | Not Equal / Not Zero |
| <code>sets</code> | SF | Negative |
| <code>setns</code> | ~SF | Non-negative |
| <code>setb</code> | CF | Below (unsigned) |
| <code>setae</code> | ~CF | Above or equal (unsigned) |
| <code>seta</code> | ~CF&~ZF | Above (unsigned) |
| <code>seto</code> | OF | Overflow (signed) |
| <code>setno</code> | ~OF | Not Overflow (signed) |
| <code>setg</code> | ~(SF^OF)&~ZF | Greater (signed) |
| <code>setge</code> | ~(SF^OF) | Greater or Equal (signed) |
| <code>setl</code> | (SF^OF) | Less (signed) |
| <code>setle</code> | (SF^OF) ZF | Less or Equal (signed) |

הפקודה `set` לא משנה שום רגיסטרים נוספים פרט ל-`al` בו אנחנו נשתמש. היא מחשבת ערך בינארי שמורכב מביטוי כלשהו שמחזיר לנו את הדרוש.

3.2 `declare initialized data`

נרצה להכריז על משתנים ולהקצות מידע. ישנם כמה אפשרויות -

`skip`: מקצה לנו זכרון לא מאותחל.

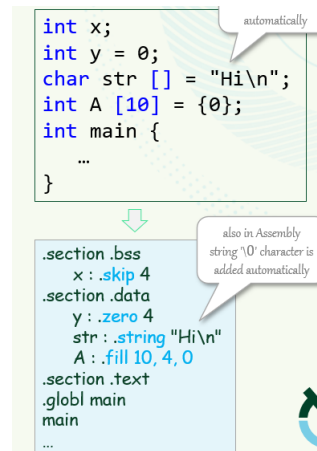
`byte`, `word`, `long`, `quad`: משתמשים באלו להקצות משתנים בגודל המתאים, אך עם **אתחול**.

`zero`: ניתן להקצות זכרון וכן לאפס אותו באותו רגע.

`string`: הקצאת מחרוזת בזכרון.

`fill`: x, y, val : כאשר נרצה להגדיר x אלמנטים בגודל y עם ערך התחלתי val . שימושי ושקול לבניית מערך כמו בדוגמה מטה. זה הרבה יותר יעיל מאשר לולאה אם אנחנו יודעים את כל הערכים ההתחלתיים.

דוגמה:



חשוב לזכור: באסמבלי אי אפשר לבצע השוואה בין שני ערכים מהזכרון, חייבים להעביר את אחד מהם לרגיסטר ואז לבצע השוואה של רגיסטר וערך מהזכרון.

3.3 הפקודה lea

הפעולה טוענת כתובת לארגומנט השני שהיא מקבלת. הכתובת שלה היא הארגומנט הראשון שהיא מקבלת. חשוב להדגיש - הפקודה *lea* בניגוד **לכל** הפקודות האחרות לא ניגשת לזכרון.

לדוגמה:

Examples:

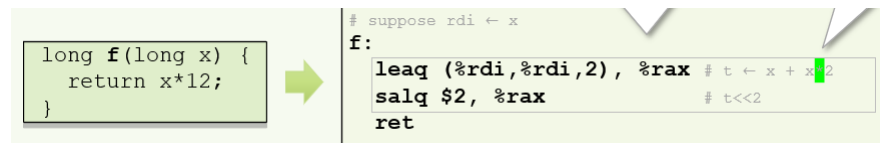
```

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

```

מה שקורה כאן בדוגמה, זה שהארגומנט הראשון הוא *addressing mode* והשני הוא רגיסטר. אנחנו מחשבים את הכתובת לפי *addressing mode* ומכניסים את הכתובת לרגיסטר השני.

במקום הפעולה יכולנו להשתמש בפקודות *mov* או *add*. אז למה צריך להשתמש בפקודה? נסתכל על הדוגמה מטה. כשנרצה להשתמש בפקודת *lea* זה בשביל לבצע חישובים - ולא להתעסק בכתובות. **הפקודה lea היא פקודה הכי מהירה שניתן לבצע במחשב שלנו** - יותר מפעולות *bitwise* (!!!!). מדוע? החישוב מאחורי הקלעים משתמש בחומרה מיוחדת שמייעלת את החישוב.



מה קורה כאן בדוגמה? ראשית אנחנו מחשבים בשורה הראשונה את $3\%rdi$. ומכניסים זאת ל *rax*. לאחר מכן, אנחנו משתמשים בפקודה *salq* שיפט אריתמטי - ומכפילים ב 2^2 , שזה בדיוק 4. שלוש כפול 4, זה אכן 12. נזכר כי הארגומנט הימני ב *addressing mode* מאפשר $i \in \{1, 2, 4, 8\}$ ולכן אי אפשר לטעון שנבצע משהו כמו $(\%rdi, \%rdi, 11)$ - זה יכל להיות נחמד אך לא עובד.

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

הערה חשובה: אם במקום *lea* היינו שמים *mov* והיינו עושים את *lea* על *%rdi*, למשל, $rdi = RAM[\%rsp + 20]$ במקום מה שנקבל עם *lea: %rdi = %rsp + 20*. כמו כן, *destination* ה*lea* תמיד יהיה רגיסטר.

3.4 C Calling Convention

כיצד פונקציות ב*C* וכן באסמבלי צריכות להתנהג? כיצד מעבירים ארגומנטים לפונקציות? כיצד פונקציה מחזירה ערך מוחזר? הדרך לחזור מפונקציה (לייבל) להיכן שקראנו לה, הוא באמצעות הפקודה *ret*. הפקודה מחזירה אותנו להיכן שקראנו לפונקציה. **נשים לב:** הארגומנט הראשון של הפונקציה תמיד הולך אל *rdi*.

3.4.1 Stack Operation

ישנן שתי פקודות באסמבלי בשם *PUSH, POP*. הפקודה *Push* היא פקודה שדוחפת משהו למחסנית, היא שונה מפקודת *mov* שיכולה להכניס לכל מקום בזכרון את הערך, היא יכולה רק למחסנית. הפקודה *Pop* שולפת משהו מתוך המחסנית. באשר *CPU* רואה פקודת *push* הוא לוקח את *rsp* מס' *bytes* כלפי למטה (בהתאם לגודל של *push* - שני בייט, ארבע או שמונה) - **הערה:** אם לא ציינו את מס' הבתים שנדחוף מראש באמצעות תוספת אל *default push* יהיה שיוקצו 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 ה*RSP* מוקפץ מעלה לערך האחרון שהיא לא משכה מהמחסנית.

נשים לב: גם לאחר פעולת *pop* הערכים ששמנו במחסנית נשמרים, עד שנכניס ערכים חדשים במקומם. יתרה מזאת - ניתן לגשת לערכים אלו. בקורס שלנו: זה אסור לחלוטין. זה מגיע ממקום של לחסוך בזכרון ולמנוע מללכת ולשים שם אפסים במקום. **עם זאת, מרינה יותר מרמזה שהיא אוהבת לשאול על זה במבחנים - אז לשים לב.**

נשים לב: בעת דחיפת ערך למחסנית הערך של *rsp* יורד, בעת הוצאת ערך הערך של *rsp* גדל. וכן - אנחנו נשים לב כי המחסנית בנויה הפוך מההגיון. **מדוע?** להזכר תמיד בסיפור של מרינה על הסנדוויץ'. ה*heap* וה*stack* יתחילו "לאכול" אחד את השני משני הצדדים עד ש(אם וכאשר) ייפגשו. מדובר באופטימיזציה (!!) שעשו באסמבלי. אם *heap stack* נפגשים - אזי נגמר לנו המקום בזכרון.

3.5 Calling Convention

כאשר אנו קוראים לפונקציה הארגומנט הראשון נשמר ברגיסטר *rdi* ומשם לפי הסדר לפי הטבלה למטה. הערך המוחזר תמיד דרך *rax*. מכאן נשאלת השאלה - ומה עם גדלים שגדולים מ-8 בייטים? מאחורי הקלעים הקומפילר מתרגם את הגודל הזה כאילו הוא מחזיר פוינטר (אכן פוינטר בגודל 8 בתים) אל הערך הזה.

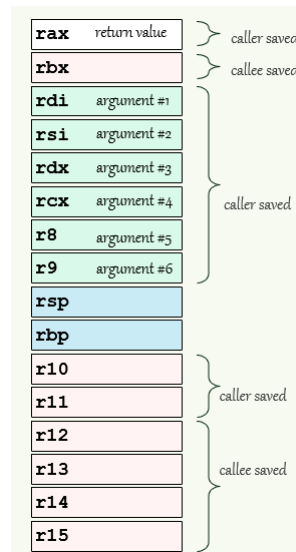
16 הרגיסטרים מתחלקים לשני קבוצות -

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

שהוא לא השתנה. כיצד נדאג שזה יקרה? מי מהפונקציות, הקוראת או הנקראת צריכה לגבות את המידע של הערך הישן במחסנית? מכאן שהזה הופך לתלוי רגיסטר. אם למשל, אנחנו מעוניינים ש- $r10$ ישאר כמו שהיה בפנייה לפונקציה אזי נצטרך לדאוג לכך בפונקציה הקוראת. אם נדבר על $r12$ אזי נצטרך לדאוג לגיבוי בפונקציה הקוראת.

callee saved: "הפונקציה הנקראת". הם: $rax, rdi, rsi, rdx, rcx, r8 - r11$.

caller saved: "הפונקציה הקוראת". הם: $rsp, rbp, rbx, r12 - r15$.



רגיסטר RBP: יש לו תפקיד חשוב בהרצת פונקציות. **תפקידו להחזיק את הכתובת העליונה של stack frame הנוכחי.** מהו *stack frame*? ה"חוצץ" במחסנית של *push*-ים של הפונקציה הנוכחית. לשם מה צריך לשמור את החצה העליון? בעיקר בשביל לשחרר את המידע שדחפנו במהלך הפונקציה. בעת פונקציה: בתחילה אנחנו נדחוף את *rsp* אל המחסנית, כי הוא *callee saved* וצריך לטפל בו בתוך הפונקציה הקוראת. לאחר מכן אנחנו נגדיר $rbp = rsp$ ונגדירו כערך של תחילת הפריים. לבסוף נעשה $rsp = rbp$ ונוציא את *rbp* מהמחסנית, כלומר נחזיר אותו לערך הקודם שלו. נרחיב עליו מטה -

תהליך הקריאה לפונקציה:

בעת קריאה לפונקציה, אנחנו מכניסים את הערכים שישלחו אליה אל הארגומנטים שנשלחים בהתאם. rsi ו- rdi וכן כל מי שעוד נצטרך. לאחר מכן, אנחנו נשתמש בפקודה *call func* שקוראת לפונקציה.

פקודת call - הפקודה מכניסה את return address אל המחסנית (כלומר דוחפים את הrip הנוכחי אל המחסנית). ולאחר מכן, הפקודה *call* קופצת אל הלייבל *func* (כלומר משנים את *RIP* אל הלייבל *func*).

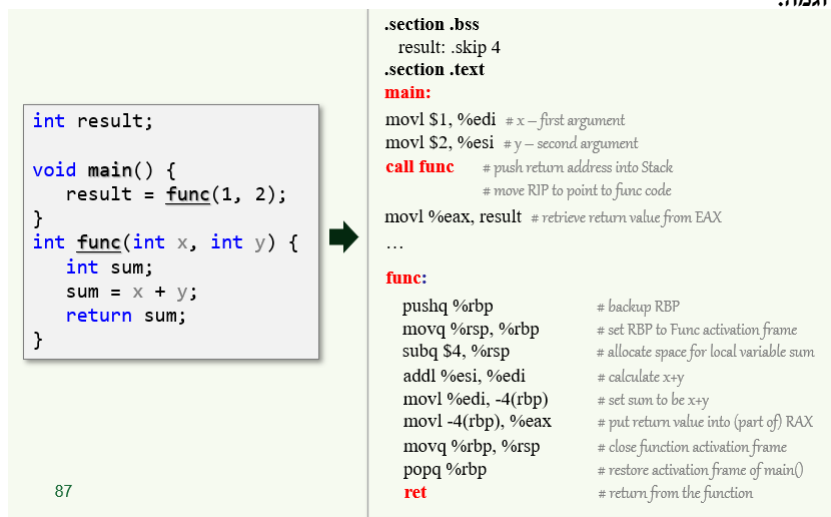
בעת שנכנסים אל הפונקציה, אנחנו צריכים לדחוף למחסנית את הערך הנוכחי של הרגיסטר *rbp*. דבר נוסף, זה לקחת את הערך של *rbp* ולהכניס לו את הערך של *rsp*. כלומר, *rbp* מצביע לאותו מקום שמצביע עליו *rsp*. באסמבלי בחרו שלא להקצות למשתנה לוקאלי שמות. ומכאן: לאחר מכן מורידים מהערך של *rsp* את סך כל הגודל של הבייטים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

```
subq $4,%rsp
```

במקרה בו למשל התייחסנו פונקציה שמקצים בה 4

עם זאת, אם אין למשתנים הלוקאליים שמות. כיצד נדע איך להתייחס אליהם? לשם כך נצטרך את הרגיסטר *rbp*, אנחנו נשמור את הערך הנוכחי של *rbp* במחסנית, ונעזר בו בשביל לדעת לאיזה משתנה אנחנו רוצים לגשת.

דוגמה:



נראה כי בעת החזרת הערך, אנחנו צריכים ומחוייבים (!!) לבצע נקיון ל-*stack*. מי מחוייב? גם מי שקרא לפונקציה וגם הפונקציה עצמה. הפונקציה שמה משתנים לוקליים והזיזה את *rbp*, היא צריכה למחוק משתנים לוקליים ולהחזיר את *rbp* למיקומו. *main* אחראי להעלים כל מיני דברים שיתכן ששם ב-*stack*. כיצד אנחנו מבטלים את המשתנים הלוקליים? באמצעות השורה -

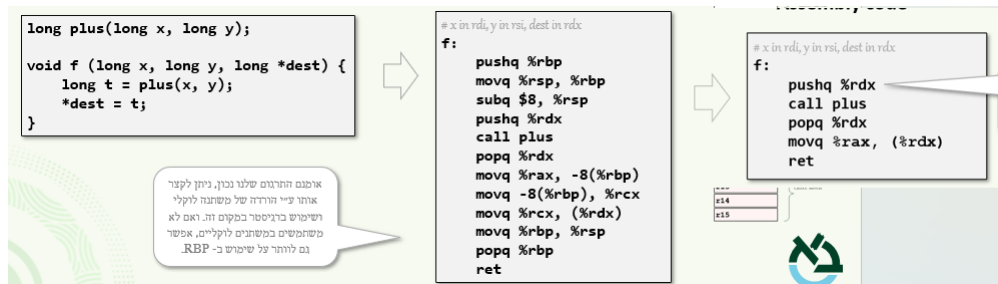
```
movq %rbp, %rsp
```

אנחנו אומרים ל-*rsp* להעלות מעלה לכתובת של *rbp* וכעת *rsp* מצביע לערך הישן של *rbp*, כך אפשר לשחזר את הערך הישן של *rbp*.
סה"כ טיפלנו בחלק שלנו. לאחר מכן מופיעה תמיד הפקודה *ret* - הפקודה לוקחת את הערך ש-*rsp* מצביע עליו ומכניסה את זה לרגיסטר *rip*, וככה אנחנו חוזרים להיכן שקראו לנו. סה"כ - תהליך ארוך שנגמר.

נסתכל על הדוגמה החשובה הבאה:

אנו מתרגמים קוד. נשים לב כי אפשרות אחת היא להשתמש באמת במשתנה הלוקאלי ולקבל את הקוד שמופיע באמצע. נשים לב כי דחפנו את *rdx* ולאחר מכן הוצאנו אותו כי רצינו לשמור את הערך שלו כי יתכן שהוא ישתנה במהלך הפונקציה *plus* כי הוא *caller saved*.

נראה כי את אותו הקוד באמצע ניתן לכתוב גם בצורה שכתבנו בצד ימין. נראה כי נשארנו עם *push, pop* של *rdx* וויתרנו על *rbp* לחלוטין - כי ויתרנו על המשתנה הלוקאלי. **מכאן המסקנה צריך להפעיל שיקול דעת:** אם אפשר לוותר על משתנה לוקאלי - נוותר, ואז לא נצטרך לעבור עם *rbp*.



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

מה ההבדל בין פונקציה לבין `jump`? ההבדל הוא שכאשר אנו קוראים לאיזושהי פונקציה והיא סיימה את פעולתה, בהכרח הקוד יחזור להיות לאחר היכן שנקראה הפונקציה, בניגוד ל`jump`.

ולסיכום - **פקודת `call`**: היא מבצעת `pushq %rip` בשביל לדעת בהמשך להיכן לחזור. וכן מבצעת `jmp target` אל הפונקציה שמעוניינים בה. **פקודת `ret`**: מבצעת `popq %rip`.

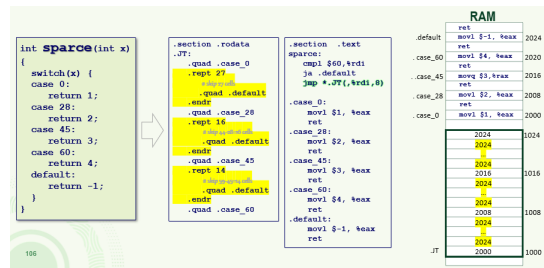
מה נעשה עם פונקציה שמקבלת יותר משישה ארגומנטים? נכניס את השישה הראשונים לרגיסטרים המתאימים ואז את השאר נדחוף למחסנית בסדר הפוך. כלומר אם יש 10 ארגומנטים נכניס את השישה לרגיסטרים ואז נדחוף את 10 למחסנית, 9 8 7 וזהו. הרעיון הוא שהכי יהיה לי קל לגשת אל המשתנה השביעי.

Stack Alignment: יש פונקציות שדורשות `rsp` יהיה כפולה של 16. צריך לדעת, לזכור ואין חובה להבין מדוע. כיצד נוודא ש`rsp` הוא כפולה של 16? נבצע הרצה `GDB` עם הדיבגר ואם במהלך ההרצה אכן `rsp` מתחלק ב16 הוא יהיה כך בכל ההרצות.

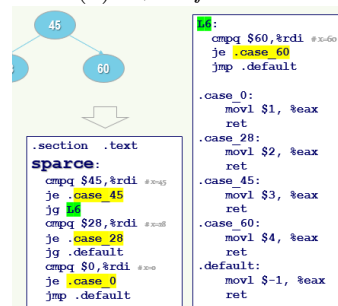
3.6 Jump Table

בשפת `C` קיים מבנה של `switch&case`. כיצד נממש מבנה כזה באסמבלי? נראה כי במקום לכתוב סוויץ' אנד קייס ניתן להמירו ל`if&else`. עם זאת - סיבוכיות של `if&else` היא $O(n)$ באשר n מספר `cases`. שימוש ב`switch&cases` יעלה $O(1)$. כיצד נממש מבנה שכזה בזמן $O(1)$? **בטבלת האש כמובן.**

נקרא לטבלה הזו `JT`, נחלק אותה לשורות. בשורה הראשונה (מלמטה) יופיע הכתובת של `case1`. וכן הלאה בשורה השנייה הכתובת של `case2`. כיצד נגש אל המיקום המדויק? `jmp RAM[JT[op]]`.



נראה כי המרווחים *cases* הם לא רציפים, ניתן להשתמש בטבלה בגודל 60. חבל על המיקום. אז נשתמש ב-*if&else* שיעלה $O(n)$? מה פתאום - נבנה עץ AVL בעלות $O(\log n)$. קל לראות.



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

3.7 GDB

GDB הוא דיבאגר: ניתן להריץ דרכו תוכניות, לעצור אותן במהלך ההרצה, לשנות ערכים של משתנים מסויימים בזמן ריצה, להגדיר *break points* וכו'. בשביל לקמפל אנחנו כותבים לרוב את השורה הבאה:

```
gcc [flags] <source file> -o <output file>
```

אם נוסיף *-g* - נקבל אפשרויות נוספות שאפשר לעבוד איתם:

```
gcc [flags] -g<source file> -o <output file>
```

אם לא נקמפל עם *flag*, לא נקבל אפשרויות אלו. שימוש בזה משנה מעט את קובץ ההרצה אך מוסיף מידע שהופך את השימוש ב-GDB להרבה יותר נוח.

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

```
run arg1 ...
```

כמו כן ניתן להשפיע על מיקום הפלט (להיכן יכתב הקלט) כך:

```
run <input.txt> output.txt
```

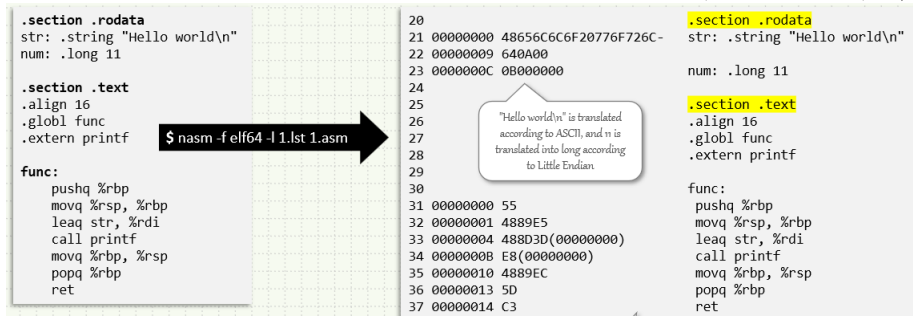
break points: כיצד משתמשים בהם ב-GDB? בשביל להפעיל נכתוב `break [target]` באשר *target* יכול להיות שם של קובץ מקור ומספר שורה, כתובת ספציפית בזכרון וכו'.

4 הרצאה 4

4.1 Assemble process

נרצה לבצע את שלב הקמפול, המרת הקוד לשפת מכונה.

listing file: קובץ שה-GCC יכול לתת לנו, בתוך כל קובץ כזה יש את קוד המקור (צד ימין), העמודה האמצעית היא תרגום לשפת מכונה והעמודה השמאלית היא הגדרת ה"מיקום" היחסי של כל פקודה ביחס ל-*section* שהיא מוגדרת בו. אנחנו נקמפל ונוכל להסתכל בקובץ זה לראות "יישור קו" בין הקמפול שביצענו לקמפול שאמור להיות.



לשים לב: גם *data* מתורגם בזמן קומפילציה, אם הוא ערך נומרי הוא מתורגם לפי *endian* *little* ואם הוא *string* אז הוא מתורגם לפי טבלת אסקי.

מה קורה בעת הליך הקמפול של הקוד הנ"ל? תחילה מגדירים *section* של *rodata*. הקומפיילר יודע שכל עוד לא פתחנו *section* חדש אז אנחנו *rodata*. בעת הקמפול, הקומפיילר מנהל הרבה מאוד טבלאות. אנחנו נתייחס לשתי טבלאות:

Symbol Table: הקומפיילר מכניס לשם מידע שהוא מזהה. בעת קמפול הקוד למעלה, אנחנו נתקלים בסקשן *rodata* ואנחנו נכניס אותו לטבלה. אנחנו נגדיר שהערך שלו בתחילה (*info*) יהיה אפס. לאחר מכן, הקומפיילר נתקל בשם חדש: המחרוזת *str*. היא נכנסת גם כן בשורה חדשה בתוך הטבלה. אנחנו נרשום בטבלה באיזה *section* היינו באשר ראינו את המשתנה *str*. כמו כן, אנחנו נגדיר לו את *location* שיהיה מיקום היחסי שלו בתוך ה-*section*. כמו כן, הקומפיילר צריך לעדכן בעת כניסת *str* את הגודל של סקשן *rodata*, כי התווסף גודל חדש לסקשן. ולכן: הקומפיילר מעדכן בטבלה את *info* שידגל להיות יותר מקום. כך נראית הטבלה -

| 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 כשנכניסים אל *text* מכניסים זאת גם לטבלה וכן *info* שלו יהיה שוב אפס בתחילה. נשים לב כי בעת פתיחת סקשן חדש זה לא אומר שהפסקנו לכתוב בסקשן *rodata*: לכאורה - מותר לפתוח סקשן *rodata* למשל פעם אחת בתחילת הקוד, ועוד הרבה פעמים במהלך הקוד. עם זאת: מרינה לא מסכימה, ואין לכך הצדקה: ואסור בקורס!

נשים לב שבעת הגעה לשורה `global func`, אנחנו נתקלים ב-`type` חדש של `global`: זה אומר, מוגדר בקובץ הזה אבל אנחנו יכולים להשתמש בו בקבצים אחרים. אם הוא לא היה `global` מתוך קובץ אחר לא יכולנו לעשות לו `extern`. כמו כן, הקומפיילר מכניס עבורו בטבלה ב-`info` את `undef`: עוד לא הוגדר. אנחנו לא יודעים מה וכמה יש בו. רק יודעים מה הטיפ שלו ובאיזה סקשן הוא. כמו כן, בעת שימוש בפונקציות כמו `printf` הקומפיילר מעדכן בטבלה היכן הוא פגש אותה, מעדכן כי הטיפ שלה הינו `extern` וה-`info` שלו הוא `undef`.

בעת שמגיעים אל פונקציית `func`: הקומפיילר מעדכן ב-`Name` של `func` שהיא אינה `undef` עוד. יש מקום בו היא מוגדרת. זה בדיוק השלב בזכותו אנחנו לא נקבל על הקוד הזה שגיאת קומפילציה! שכן אם היה נשאר לקומפיילר בתוך הטבלה `undef` זה אומר שיש פונקציה שאין לה `info` (היא לא מוגדרת) והיינו מקבלים שגיאת קומפילציה.

בעת הגעה לשורה של `leaq`:

```

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

```

נראה כי ישנה בעיה. הקומפיילר רואה שיש לו לייבל בשם `str`, ומנסה להבין היכן הוא ממוקם. נראה כי לא ידוע לנו גודל סקשן `text`, וגם בסיום קריאת כל הקוד עדיין לא נדע בהכרח את הגודל של סקשן `text` שכן ייתכן שישנו לאחר מכן את כל הקוד של `extern` שעשינו (ויאריכו אותו או יקטינו אותו). כמו כן, `str` נמצא ב-`rodata` שנמצאת מעל `text`, וכיוון שלא ידוע גודל `text` זה משפיע על `rodata`. אז מה הקומפיילר עושה?

הקומפיילר בונה טבלה נוספת - Relocation Table: הוא מעדכן שם, שבתוך סקשן טקסט, במיקום `0x07` (הביט השביעי, בדיוק איפה שמתחילים למעלה סימני השאלה - אם כי בפועל אלו לא סימני שאלה אלא אפסים), הוא מכניס שם "בקשה" עתידית, יש לך `symbol` בשם `str`, תקצה שם 8 בתים עבורו לעתיד.

| Relocation Table (.reloc) | | | | |
|---------------------------|----------|--------|------|------|
| 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 בקומפילציה קריאה לפונקציות חיצוניות? למשל - כשנהיה בשורה הזו:

```

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

```

אל אותה טבלת `Relocation` אנחנו מכניסים את המיקום (שמתחיל בדיוק באותם סימני השאלה) אליו נרצה להכניס בהמשך את המיקום של `printf`.

בסיום המעבר - הקומפיילר סיים להכין את כל הטבלאות שהוא צריך למען הקמפול, בפרט השתיים שתיארנו, והוא מוכן לעבור לשלב הבא.

4.2 ELF relocatable format

כעת נדון כיצד הקובץ המקומפל נראה באמת - לא איך שאנחנו קימפלנו אותו ידנית. ספוילר - נראה די דומה למה שקמפלנו ידנית.

נוח לראות את הקובץ הזה באמצעות תוכנה `readelf`.

```
$ readelf -a 1.o
```

ELF Header:

```

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

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

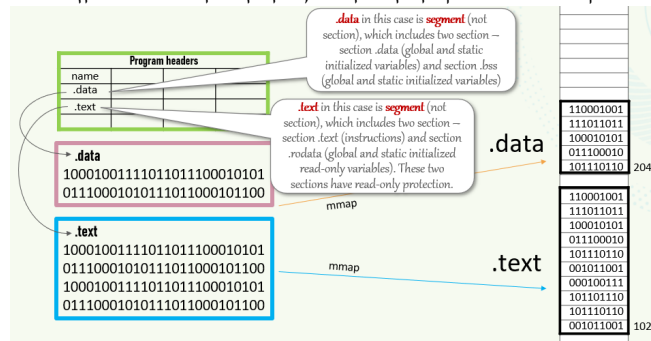
readelf utility allows to observe structure of ELF file (object or executable)

ELF זה מידע שהקומפיילר כותב שיכול לעניין מאוד את *Loader* או *Linker*.
בשורה הראשונה מופיע *Magic*: מספרים, $464c457f = ELF$
Class: מציינים שמדובר ב-*ELF64*, למען *Loader*, שידע כיצד לפרסר את הקובץ ועם איזה קובץ הוא עובד.
Data: הקומפיילר מעדכן כי הקובץ הוא בשיטת המשלים 2 (מס' שליליים בנוסף) וכן עם *Little Endian*.
כמו כן מעדכנים שמדובר בקובץ *Relocatable*, שעוד נדרש לקשר עם קודים שונים (כל *extern*).

לבסוף, מופיעים כל ה-*sections*. נראה כי למרות שקוד המקורי ישנם שני סקשנים, הקומפיילר טוען שיש 7.
תחילה הוא טוען שיש *text*: אותו אחד שאנחנו יצרנו קודם, אכן בגודל של 17. כמו כן, אנו אומרים לו שה-*offset* שלו הוא 40. בקובץ המקומפל - הוא מתחיל בשורה 40.
לאחר מכן אנו מגלים שאת הטבלאות שקודם ראינו, הקומפיילר שומר בתוך *section* חדשים.
rel.text זה ה-*relocationTable* שבנינו וכן *symtab* זו טבלת *symbol*. הוא שומר עוד כמה סקשנים - שאינם בחומר הקורס.

4.3 ELF executable format

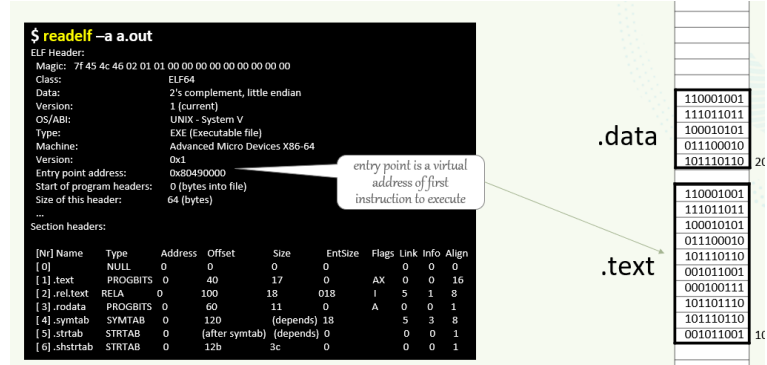
נרצה להבין מה ההבדל בין קובץ מקומפל, לקובץ מקומפל ומלונגקץ: *executable*.



כאן יש לנו *program headers* שמכיל בתוכו שני סקשנים בדיוק, שנקרא להם כעת *segment*.

segment data: סקשן *bss* וסקשן *data*, יהיו בתוך סגמנט זה.
segment text: סקשן *text* וכן סקשן *rodata* - שניהם לא יכולים להשתנות בזמן ריצה.
 וכן, מתבצעת הקצאת זכרון של סגמנט *text* ו*data*.

נסתכל כעת על ה-ELF:



מה השינויים? נראה כי בקובץ הלא מלונק, *Entry point address* היה אפס. כאן נראה שיש שינוי גדול, זה לא אפס. כעת זה מצביע על הכתובת שאנחנו רוצים ש-*Loader* יכניס *RIP*.
 נראה כי *Linker* נותן כתובת של *RIP*, 100 נניח, אותה ה-*Loader* מכניס אל ה-*RAM* באותו מיקום, ומזיז את ה-*RIP* לשם. עם זאת - הזכרון משותף לכל המחשב, ויתכן שתהליך אחר תפס את הכתובת הזו ושמו את הכתובת הזו במיקום אחר, 300 נניח. מה נעשה? מערכת ההפעלה מייצרת לעצמה מיפוי, ובכתובת 100 המערכת כותבת לעצמה: בתהליך *a.out* אם *Loader* יבקש מאה, תפנה אותה ל-300. **מסקנה:** הכתובות הן וירטואליות ואינן באמת אמיתיות. על זאת ועוד נרחיב במערכות הפעלה.

לכתובת אמיתית נקרא **כתובת פיזית** - אין לנו דרך לגשת אליה, ויש לנו **כתובת וירטואלית** - הכתובות שאנחנו חשופים אליהם. יש לנו **כתובת אבסולוטית**: כתובת וירטואלית שקומפילר/לינקר חישב אותה והיה רוצה להשתמש בה אילו היא הייתה פנויה ב-*RAM*. ויש **כתובת relative** - כתובת ביחס למיקום הפקודה בסגמנט כלשהו.

הערה חשובה: כל עוד *gcc* לא נאמר לו אחרת הוא מניח שגודל התוכנית קטן. כלומר: גרסת *32-bit*. ולכן הוא מניח שהתוכנית קטנה ובהתאם שומר כתובות בגודל 4 (במקום 8 בייטס). זה כמובן עוזר לעילות.

ישנן פקודות כמו *call*, *jmp* שהן מלכתכילה *REL* (רלטיביות) - כלומר תבצע חישוב *independent position*. לכן בטבלה של *RelocationTable* בעת פקודות אלו הוא ישים *REL*. לעומת זאת ישנה כתובת *ABS* (אבסולוטית) - למשל כמו *str, %rdi* ולא *(str(%rip))*.

4.3.1 כיצד כותבים וירוס?

אנחנו רוצים להוסיף לקובץ ההרצה שלכם, קטע קוד שאני יצרתי, שאם תריצו את קובץ ההרצה, הקובץ שלי ירוץ גם על הדרך ויעשה בעיות.

הנה רעיון: נבקש ממכם ללחוץ על כפתור כלשהו, אם תלחץ על הכפתור אני אוריד לכם למחשב קובץ הרצה, שיכנס אל *file system*, נחפש את קובץ ההרצה שלכם, ולפי תכנות מונחה עצמים כפי שראינו - ניתן לכתוב לתוך קובץ. נוסיף אל הקובץ את הקובץ המקומפל שלי, ונשמור את הקובץ. כאשר נריץ את קובץ ההרצה הזה, נריץ גם (אולי?) את הקובץ שאני הוספתי לך. נניח שהקובץ שאני הוספתי מוחק את כל ה-*file system*. נשאלות כמה שאלות.

א. האם *Loader* יפרסר זאת? בעת ש-*linker* חישב את הגודל הסופי של סקשן *text*, הם בוודאות לא התחשבו בקוד שאני הוספתי אל קובץ ההרצה - קוד זה לא היה קיים לא בזמן הרצה

ולא בזמן לינקוג'. אז, *Loader* לא אמור להעלות את הקוד הזה כחלק מ-*process image*. לכאורה. בפועל - כאשר *Loader* טוען *process image* הוא לא טוען אותו לפי הגודל המדויק. בפועל, אם ביקשתי מ-*Loader* לבנות סגמנט טקסט בגודל 4bytes הוא בונה סגמנט בגודל 4k-4096bytes (!!!). כאשר אנחנו טוענים מקום לסקשן טקסט אנחנו טוענים את זה פר בלוקים. ולכן, נותר מקום פנוי. ולכן אותו קוד, עם קובץ ההרצה שימחק לי את ה-*file system*, עשוי להתקבל ע"י *Loader*. נניח והתמזל מזלו של הוירוס שלנו, והוא עבר את זה.

ב. האם הקוד הנוסף הזה יתבצע כאשר *process* יתבצע? חד משמעית - לא. בסוף פונקציית *main* יש לנו תמיד *exit*, גם אם כתבנו וגם אם לא מערכת ההפעלה מוסיפה לבד. *exit* זו פניה למערכת ההפעלה שאומרת: אני *process* שסיים לבצע כל מה שהוגדר, ובבקשה מערכת ההפעלה תמחקי את *process* כעת. ולכן, בוודאות, הקוד הנוסף לא יתבצע.

אז מה נעשה? במקום להוסיף את הקוד בנוסף אל *segment text* אנחנו נמחק חלק מהקוד שם, ונכניס במקום את הקוד שלנו. כעת - יש סיכוי שהקוד יתבצע. מדוע יש סיכוי ולא בוודאות? כי יתכן שהמחיקה הרסה דברים חשובים וכעת הקוד לא יתקמפל או תהיה שגיאת זמן ריצה. אז אנחנו לא רוצים אולי - אנחנו רוצים בוודאות ליצור וירוס. כיצד? נדרוס את השורות הראשונות של פונקציית *main*. נכניס אל השורות הראשונות של *main* את הקוד שלי, הוא בוודאות ירוץ. באותה הזדמנות אפשר כמובן למחוק את שאר הקוד (אם כי זה קשה יותר), אך זה לא יפריע לי כי בוודאות הקוד שלי יתבצע, ולא אכפת לי מה יקרה הלאה בקוד המקורי - כי מטרתי להרוס.

פתרון חלופי וקל הרבה יותר: נדביק את הקוד שלנו, ונשנה את ה-*Entry point* להיות למקום שהכנסנו את הקוד. ואז, כאשר *Loader* יקרא את ה-*entry point* הוא ילך אל הקוד המרושע שכתבנו, וסיימנו.

4.4 Disassembled

תהליך של הפיכת קוד משפת מכונה לקוד בשפת אסמבלי. נעיר כי תהליך זה לא חוקי לפי החוק על קוד שהוא לא שלנו. זה מאפשר לנו לחפש באגים למשל בצורה הרבה יותר נוחה מאשר לחפש אותם בקוד של שפת מכונה.

4.5 Linking process

לוקחים כל מיני קבצי *Object* (שלנו ולא שלנו, ספריות סטנדרטיות לדוגמה) ורוצים ללקנץ' אותם לקובץ הרצה יחיד. נשים לב שקומפילר אחד מבצע קומפילציה כל פעם של קובץ אחד. קיבלנו הרבה קבצים מקומפילים, ונרצה לבצע לינקוג' של כל הקבצים לקובץ אחד. נשים לב כי כל שגיאה שתתבצע בשלב זה תקרא כעת **שגיאת לינקוג'**. עד היום שגיאות אלו היו תחת שגיאות קומפילציה". מטרה נוספת של הלינקר היא לפתור את כל הבעיות שנוצרו בזמן הקומפילציה.

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

כללי *Linker*:

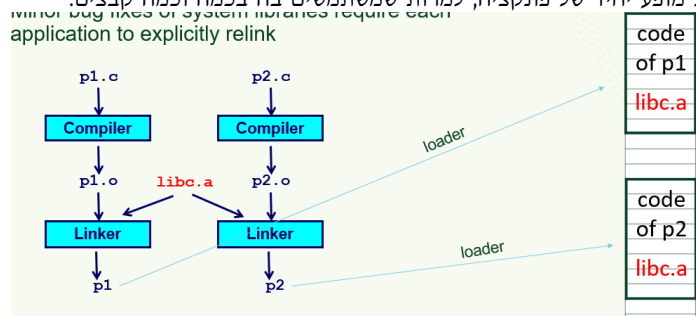
1. *strong* סימבול יכול להופיע רק פעם אחת. אין מצב שיש שני פונקציות למשל באותו שם.
2. *strong symbol* יכול לדרוס *weakSymbol* אם נלקג' את שני הקבצים.
3. אם יש לנו שני *weak symbol*, הלינקר יכול לבחור אחד מהם שרירותי.

חשוב: נניח שבתוכנית אחת הגדרנו $int\ x = 7$, ובתוכנית אחרת הגדרנו $double\ x$ ללא אתחול. לפי הכללים, נוכל להשתמש לכאורה ולתעדף את $x = 7$ שהוא *strong*. כשאנו קוד מתורגם לאסמבלי, הוא מגדיר להם גדלים שונים. על פניו, לכאורה ה-*Linker* יאשר אבל בתוכנית השניה התייחסנו אל $double\ x$ ובראשונה $int\ x$, לכן אנחנו עלולים לקבל בזמן ריצה תוצאה לא טובה. ולכן נקבל כאן שגיאת זמן ריצה.

באמצעות לינקוף אפשר לבנות גם ספרייה בעצמנו: כיצד? נקח למשל פונקציה *printf.c* ונממש אותה, נקמפל ונקבל קובץ הרצה. כמו כן נעשה זאת על עוד פונקציות כמו *atoi.c* וכו'. נקבל הרבה קבצי הרצה וננלקג אותם יחד באמצעות הלינקר לקובץ אחד.

נשים לב כי הלינקר ישתמש לפעמים בספריות סטטיות קיימות, למשל היכן שיש את *printf*, הוא לוקח את הספרייה יחד עם שאר הקבצים שקמפלנו ואותם מכניס לקובץ הרצה.

נשים לב כי יתכן מצב כמו כאן מטה: שני קודים שמשתמשים שניהם בספרייה כלשהי. *Loader* יקח כל קוד יחד עם הספרייה, ונקבל (בצורה הגיונית) שכל קוד כולל את הספרייה ולכן בזכרון שלנו יש כרגע פעמיים את הקוד של *libc.a*, וזה חבל כי אנחנו מבזבזים מקום בזכרון. מה נעשה? אם אנחנו יודעים שיש לנו פונקציה כמו *printf* שמופיעה הרבה פעמים, נשים אותה בתוך **ספרייה דינמית**: אומרים למעשה ל-*Loader* - אתה לא מעלה ליזכרון את שני המופעים של *printf*. אבל, כשאתה רואה *printf* אתה הולך לספרייה הדינמית, מושך משם את הפונקציה וכתוצאה מכך אתה מקבל מופע יחיד של פונקציה, למרות שמשתמשים בה בכמה וכמה קבצים.



נשים לב, איננו יודעים, גם לא ה-*Loader* ולא ה-*Linker*, היכן תמצא ספרייה דינמית שכזו. קוד מסוג זה, יקרא *Position Independent Code*.

4.6 Position Independent Code

קוד שלא תלוי במיקום. מהו קוד שתלוי במיקום? ישנם מיקומים של *sections* שנקבעים בזמן קומפילציה/לינקוף. אם החליט שסקשן *text* מתחיל בכתובת 1024 אז 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 כזה, הוא מפענח את זה בצורה שונה מ-*addressingMode*. הוא מתייחס לביטוי *str(%rip)* כחיסור כתובות *str - rip*. וכך הוא מחשב את המיקום היחסי של *str* ביחס למיקום הנוכחי של *rip* באשר מבצעים את פקודת *leaq*. למעשה הפקודה אומרת: כמה אני צריך לזוז מהמיקום הנוכחי בשביל להגיע אל *str*. הקומפיילר (דגש - אנחנו לא) מחשב את החיסור, מסמנו $x = str - rip$ ואז מתרגם זאת $x(\%rip) = x + \%rip$ בדיוק לפי *addressingMode*.

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

PIC

חשוב לדעת: בין סגמנט *text* לסגמנט *data* ישנו מרווח בין שני הסגמנטים של גודל קבוע (נניח 1000) שהקומפיילר קובע בזמן קומפילציה וידוע בזמן ריצה. מדוע זה חשוב? אם יהיה לנו *loader* *confused*: הוא בנה *process image* במיקום הלא נכון, הוא עדיין ישמור על הרווח הקבוע ובאמצעות הפקודה *str(%rip)* שהיא *position independt*: הוא מריץ את הקוד שמופיע בצד ימין בתמונה, שהוא קוד בלתי תלוי במיקום, ולמרות שה-*loader* התבלבל עדיין הקוד שלי יהיה תקין.

סיבות לשימוש ב-*position independt*:

1. אחת הדרכים לוודא שהקוד שלנו יהיה בטיחותי, היא לפזר את הסגמנטים שלנו במיקומים שונים בכל פעם. ישנם וירוסים שונים שמשתלטים על *process image*. אז הקומפיילר או מערכת הפעלה בונים את ה-*process image* בצורה אקראית - וכך לוירוס יהיה הרבה יותר קשה למצוא את ה-*section's* השונים בקוד.
2. באשר נעשה *dynamic linking* לא נדע היכן נמצאות הספריות הדינאמיות שלנו, ולכן כאשר נשתמש בספריות דינמיות לא נוכל להשתמש בקוד תלוי מיקום שכן המיקום משתנה בהתאם למיקום הספרייה בזכרון (שיכול להשתנות).

* אם מקמפלים עם *no - pie* - זה אומר שמדובר בקוד שהוא לא *position independent* ואז לא צריך (לכאורה) להוסיף *%rip* בסוגריים.

4.7 תרגול

4.7.1 ארגומנטים ב-*STACK*

נניח שהעברנו פרמטרים לפונקציה דרך המחשנית. נראה כי נרצה להשתמש בערכים אלו מהמחשנית. כיצד נעשה זאת? נצטרך "לקפוץ" מעל המיקום הנוכחי (*rbp*) כגודל מה שדחפנו מתחתיהם למחשנית.

4.7.2 *Variadic Functions*

פונקציות שניתן להעביר אליהן מס' לא מוגבל של משתנים. הארגומנט הראשון **לרוב** יציין את מספר הארגומנטים שיועברו.

כיצד נממש פונקציות כאלו ב-C? אנחנו לרוב נכתוב ... במקום הפרמטר האחרון. החתימה תראה כך:

```
int printf(const char* format,...);
```

בשביל לכתוב פונקציות כאלו ב-C נצטרך לעזר 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 מסיימים "לטייל" 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 הנ"ל מחשבת סכום של מספר משתנים שנקבל, באשר הפרמטר שהיא מקבלת הוא מספר המשתנים. בתחילה משתמשים ב-*va_args* על מנת שיהיה אפשר לגשת ובכל שלב מתקדמים לאחד הבא עם *va_args*.

4.7.3 קבלת ארגומנטים בשורת ההרצה

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

$$rdi = (int)argc$$


$$rsi = (char**)argv$$

כלומר, *rdi* יחזיק את מס' המשתנים ו-*rsi* מצביע לערכם של המשתנים עצמם.


4.7.4 Flow Control

כל מבנה של תנאי, לולאה וכדומה ניתן להמיר מ-C לאסמבלי. נראה מספר דוגמאות.

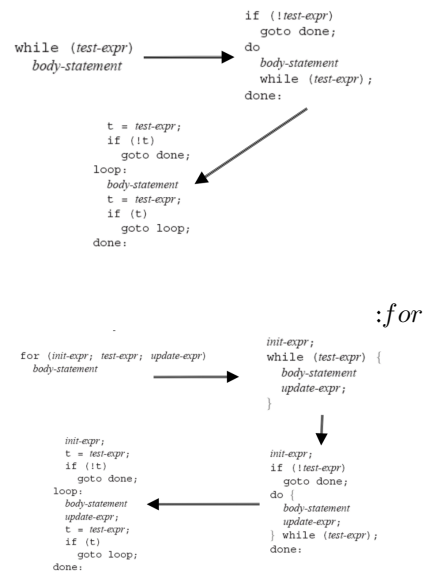
if&else

| | | |
|--|--|--|
| <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

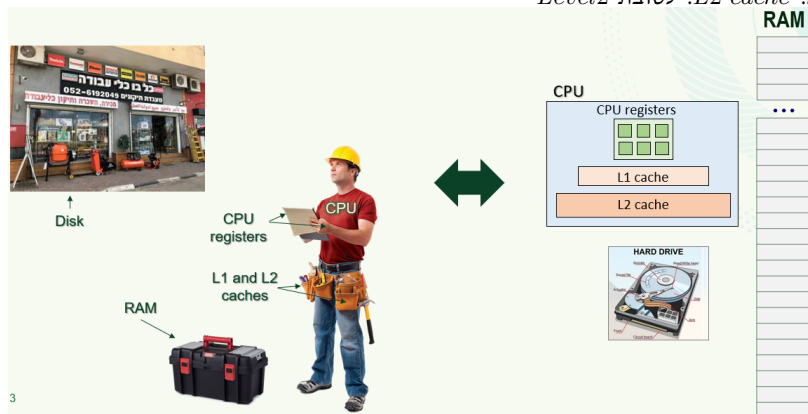


5 הרצאה 5 - memory hierarchy

5.1 הקדמה

אנו מכירים עד היום שינו RAM , ישנם רגיסטרים בתוך ה CPU וכן ישנם דיסקים. ישנם שניים נוספים בתוך ה CPU :

1. $L1$ cache: לטובת $Level1$
2. $L2$ cache: לטובת $Level2$



נניח שיש לנו אדם" בתפקיד CPU . הידיים שלו הם רגיסטרים (אחרת, איך יעבוד?), הדיסק זה לוקח זמן לגשת אליה, ולוקח זמן לגשת אל הדיסק. זמן גישה לדיסק עלותה פי מליון בערך לעומת גישה לרגיסטרים. זו הסיבה - ש CPU לא מסכים ל $loader$ לעבוד עם דיסק, ומכריח אותו לבנות $process$ image ב RAM . החגורה על האדם - היא בדיוק ה $cache$. פחות אטרקטיבי מרגיסטרים, אך עדיין קרוב אליו. ארגז הכלים הינו RAM .

5.2 cache

כל תא בתוך *cache* הינה שורה ונקראת *cache line*. ישנם מס' *levels*. כל *cache line* היא באורך של 64 bytes. ניתן לראות שכל שורה כזו אמורה להחזיק יותר מנתון אחד.

1. *L1 cache*: לטובת *Level1*. פיזית קרוב יותר למעבד, ולכן הוא מהיר יותר אך הוא קטן יותר.
2. *L2 cache*: לטובת *Level2*. פיזית רחוק יותר מהמעבד, ולכן הוא איטי יותר, אך הוא גדול יותר.

מדוע L1 לא גדול יותר אם הוא קרוב יותר ל-CPU? זה לא עובד ככה. אם הוא יהיה גדול יותר בהכרח הוא יהיה יותר רחוק מהמעבד כי אם הוא גדול יש הרבה מאוד מיקום, וחלק מהמיקום יהיה אוטומטית רחוק יותר מהמעבד.

גם L1 וגם L2 הרבה יותר מהירים מה-RAM.

ניתן לקחת נתון מרגיסטר ולהעביר אותו ישירות אל *cache*, ולהפך: ניתן לקחת מ-*cache* ולהעביר לרגיסטר.

מדוע אנחנו זקוקים ל-*cache*? נניח שיש לנו משתנה X בזכרון במיקום 1028. ברמת החומרה - אוטומטית, המעבד יורד למטה בזכרון לכתובת הנמוכה ביותר שקרובה אל X שמתחלקת ב-16: זו 1024, ומשם הוא לוקח 64 בייטים ומכניס ל-*cache line*. מדוע זה לא מיותר? למה לקחת הכל במקום לקחת רק את X ? לא חבל על המקום? לא חבל. נניח שרצינו גם את Y , שבסבירות גבוהה נמצא ב-*cache*. נוכל לחפש אותו ב-*cache*. **נשים לב: גישה ל-RAM עלותה פי מאה מגישה ל-*cache***. מי אמר שבכלל נרצה את Y ? ובכן-

Locality principle: עקרון הלוקאליות. אם משהו נמצא בקרבה של מה שכעת השתמשתי בו (Y נמצא בקרבת X), אזי בסבירות גבוהה אני אשתמש גם ב- Y .

למה טוב *cache*? נניח ויש לנו מערך ולולאה שסוכמת אותו. אם נרצה את $a[0]$ כנראה שלאחר מכן נרצה את $a[1]$. איזה יופי - הוא נמצא ב-*cache line*. אם יגמר לי המיקום ב-*cache line*? נגש שוב לזכרון. המסקנה: באמצעות *cache* אנחנו ניגשים הרבה פחות לזכרון. ניגשים לזכרון רק בשביל להביא *cache line* חדש. גישה לזכרון עלותה 100 ננו שניות.

מסקנה: נרצה לכתוב קוד כמה שיותר סידרתי, ככל שהוא יהיה יותר סידרתי הוא יהיה יותר מהיר. אם נעשה *else if*? זה לא טוב, זה מביא אותנו לגשת יותר לזכרון. מה באשר לפונקציות? לא נשתמש בפונקציות יותר? נשכפל קוד מלא פעמים במקום לקרוא לפונקציה בשביל לשמור על קוד סידרתי? נשים לב שניפוח זה לא טוב - למה לא טוב? סיבוכיות המקום גדלה, ה-*process image* גדל במיקום שלו ב-RAM ולכן בעקיפין זה גם פוגע ב-*cache*. (נדבר זאת כשנדבר על אופטימיזציות).

spatial locality: אם יש משתנה או פקודה לידי בזכרון, אזי בסבירות גבוהה מאוד רצו שיהיה שימוש כעת במשתנה או פקודה זו.

Temporal locality: אם אני משתמש במשתנה, סביר להניח שבזמן הקרוב אני אשתמש בו עוד פעם. למשל אם נחשב סכום של מערך העקרון לא יפעל על $a[0], a[1], \dots$ אך כן יפעל על i ויפעל על sum . לולאות משתמשות טוב בעקרון זה.

התהליך של *cache* למציאת x כלשהו: נקרא את x , *cpu* בודק אם x ב-L1. אם אכן המצב: נביא את x לתוך רגיסטר ב-CPU. (נשאלת השאלה, בתוך L1 יש מס' שורות. האם זה משנה באיזו שורה נביא את x אליה? הרי אנחנו לא רוצים לעבור בכל השורות. החיפוש ב-*cache* הוא יעיל). אחרת, נחפש ב-L2, אם נמצא נביא את x לרגיסטר ב-CPU וכן אנחנו נקדם את x אל L1 (מדוע? בסבירות גבוהה כמו שאמרנו נשתמש בו שוב ונרצה פעם הבאה לגשת אליו מהר יותר). אם לא נמצא שם - אזי נלך לחפש ב-RAM.

נשאלת השאלה - אם הוא בכלל התגלה ב- RAM , לא חבל על הזמן שבזבזנו בחיפוש ב- $L1, L2$?
 אנחנו מניחים שהקוד הוא *cache friendly* - קוד שהמתכנת שכתב אותו מודע ליתרונות של *cache*
 ולכן בסיכוי גבוה מאוד x יתגלה ב- $L1, L2$. אם לא: יתכן, אך הסבירות לכך נמוכה ולכן בטווח הרחוק
 זה אכן משתלם.

איך מגיעים אל $L2$? תמיד מנסים להכניס אל $L1$, אם אין שם מקום אנחנו מכניסים אותו ו"מעבירים"
 את מה שהיה בו אל $L2$. כיצד אנחנו יודעים את מי אנחנו מעבירים במקומו ל- $L2$? אינטואיטיבית -
 את מי שהשתמשנו בו הכי רחוק, ולכן הסבירות שנרצה שוב להשתמש בו כרגע נמוכה.

עדכון ערך ב- $cache$ ולא ב- RAM : ישנו רגיסטר בשם RIR - ששומר את הפקודה הבאה לביצוע.
 אנחנו קוראים אותו מבצעים את הפעולה ונניח שקודם לכן שמרנו $x = 5$ והפקודה הייתה $++$. נראה
 כי נרצה לעדכן את הערך ל-6 ב- $cache$ בלבד. אבל אז נוצרת בעיה: ב- RAM כתוב לי ש- $x = 5$ אבל
 ב- $cache$ הוא 6. כלומר: RAM לא יודע את הערך העדכני של x . זו בעיה - במסגרת הקורס שלנו:
 לא אכפת לנו מזה כיוון שאנחנו מדברים על תוכנות סדרתיות. עם זאת, כאשר נתחיל לכתוב תכנות
 מקבילי זה יהפוך לבעיה קריטית ויעשה את ההבדל בין תוכנית נכונה לשגויה. נדע זאת כרגע - ונאפשר
 את זה. בקורסים עתידיים - נדע כיצד פותרים זאת.

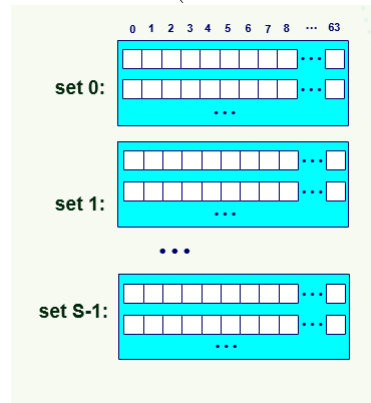
מטריצות: ישנן שתי דרכים לעבור על מטריצה - דרך שורות ודרך עמודות. מעבר על מטריצה
 דרך שורות יהיה הרבה יותר מהיר ממעבר על עמודות! קוד שעובר על עמודות אינו *cache friendly*:
 בכל שלב אנחנו נשתמש רק באיבר הראשון של השורה שנביא וזה יהיה לא מהיר בכלל. אם נעבור
 לפי שורות לפי העקרון שלמדנו אודות השימוש ב- $cache$ השורות יגיעו אחת אחרי השניה והקוד יהיה
 סדרתי. מסקנה: הרבה יותר מהיר!

מבחנית גדלים:

אם מביאים קוד הוא נכנס אל $L1 \text{ instruction cache}$, שהוא בטווח של $16 - 128(KB)$
 אם מביאים $data$ הוא נכנס אל $L1 \text{ data cache}$ שהוא בטווח של $16 - 128(KB)$
 $L2 \text{ cache}$ הוא בגודל $128(KB) - 8(MB)$

5.3 organization of a cache Memory

$cache$ מחולק ל- $set's$. כמה? תלוי בכמה היצרן הגדיר. ראינו 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 $cache \text{ line}$
 (ב- $X86$ אמרנו 64 בייטים) אז אנחנו יודעים לדעת כמה $lines$ יהיו.



נניח שיש לנו תוכנית פשוטה. מוגדר משתנה x ב- RAM ומבצעים לו $x++$.
 נניח כי $|Sets| = 8$. איך נמקם את x ב- $cache$? במקום x נסתכל על המיקום של x . נחליט שאנו מתעלמים מ-6 הביטים הימניים של הכתובת ($2^6 = 64$). נקח את שלושת הביטים הבאים ($2^3 = 8$) הימניים ביותר ממי שנשארו. שלושת הביטים הימניים ביותר שנותרו קובעים באיזה set יכנס $line$ ב- $cache$ שלי.

```

assume S = 8
assume x's address is addr = 0 ... 001000100
skip 6 ( $64 = 2^6$ ) right most bits of addr
take 3 ( $8 = 2^3$ ) next right most bits of addr
    
```

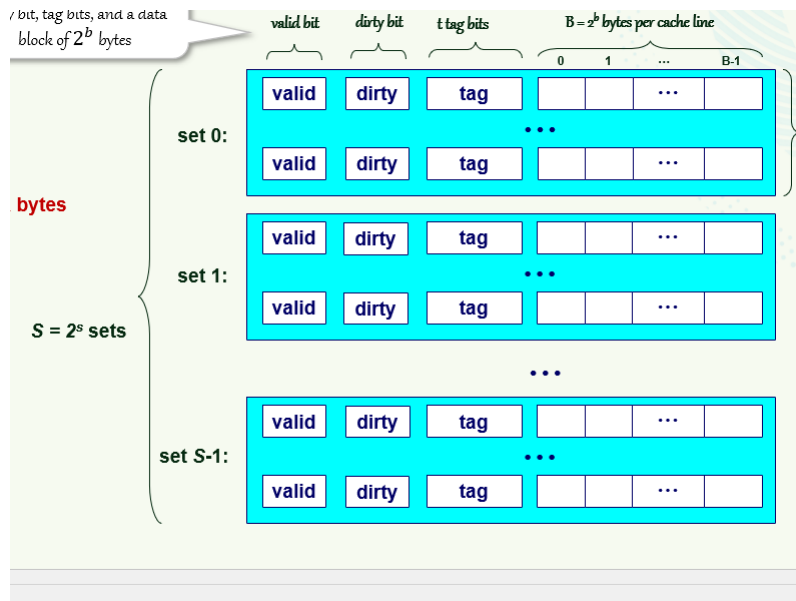
נניח שהכנסנו את x , כיצד cpu ידע באיזה set נמצא x ומה המיקום היחסי שלו באותה $cacheLine$?
 מתמטית - אם מס' מתחלק ב-64 ללא שארית בהכרח 6 ביטים אחרונים שלו הם אפסים.
 אנחנו נסתכל על כתובת שמתחלקת ב-64 והכי קרובה לאיקס הנוכחי, כפי שאמרנו קודם ונקח 64 ביטים. נקח את 6 הספרות האחרונות של כל כתובת והם ייצגו את המיקום היחסי של הכתובת בתוך $cacheLine$. כמו כן: נקח את $logx$ באשר x הוא מס' הקבוצות: נקח את $logx$ הספרות הבאות אחרי 6 הספרות והם ייצגו באיזה קבוצה אנחנו נמצאים מבין x הקבוצות שכן מספיקים $logx$ ספרות לייצוג x מספרים. סה"כ כך תיוצג כל כתובת ב- set 's שבתוך $cache$. נשים לב שכיוון שתחילת ה-64 ביטים הם בכתובת שמתחלקת ב-16 הספרות הימניות שלה יהיו אפסים ומשם נתחיל לייצג את המיקום היחסי בתוך $cacheLine$.
 הערה חשובה: המיקום בתוך אותו set אינו ידוע ולכן בכל set צריך לחפש.

לכל $cacheLine$ מוצמדים:

tag: ראינו כי לקחנו קודם לכן 6 ועוד 3 ביטים, אך נשארו $55 = 64 - 9$ ביטים נוספים, אנחנו נרצה להכניס אותם אל tag ה- CPU רוצה לדעת שה- $cacheLine$ מכיל באמת את x ולכן הוא משתמש ב- tag , שכן יתכנו כתובות שונות עם אותה סיומת של 9 ביטים. כשה- cpu מחפש את x ב- $cache$ הוא מסתכל על שלושת הביטים הכחולים, ניגש ל- set המתאים, באותו set יש מס' שורות ולכל שורה יש tag . cpu בודק שורה שורה את tag השורה וכן שאר הביטים של x (ללא אלו שירדו) ובודק האם אכן ישנה התאמה.

dirty bit: האם $cacheLine$ הוא $dirty$ או לא. $dirty$ הוא אחד שערכו שונה ב- $cache$ ממה שערכו מופיע ב- Ram . מדוע זה חשוב לדעת? אם "נפטר" $dirty bit$ אנחנו חייבים לגשת אל Ram ולעדכן את הערך של הביט כי אחרת - אנחנו נאבד אותו (הוא השתנה ואנחנו לא יודעים זאת בזכרון, ואנחנו מוחקים מה- $cache$). סכנה לאבד אותו לנצח).

valid bit: האם מותר להשתמש בנתון שיושב ב- $cacheLine$ או שאסור. למה שיהיה אסור להשתמש? כל עוד אנו בעולם סידרתי, אין סיבה שיהיה אסור להשתמש וכל השורות הינן $valid$. כשאנחנו עוברים לעולם מקבילי או בעולם שיש בו כמה $process$ - נהיה בבעיה. לכן כרגע, כשאנחנו בעולם סידרתי: כולם $valid$. במקביליות - באשר עוברים בין $process$ שונים אנחנו מסמנים ישירות $invalid$.



Direct Mapped Cache: מצב שיש לנו בדיוק $cacheLine$ אחד בתוך כל set . במצב זה אנחנו גם יודעים איך לחפש את $CacheLines$ ב set כי יש בדיוק אחד בפנים. החסרון המרכזי שאם נביא $cacheLine$ נוסף אל אותה set נהיה חייבים לדרוס אותו. אין מקום אחר בתוך אותה set שנוכל להעביר אותה אליו. היתרון המרכזי הוא שזמן החיפוש כאן הוא בדיוק $O(1)$ - נסתכל על tag בדיוק ונדע האם x שם או לא.

K-associative - Cache: ישנם k $cacheLines$ בתוך כל set .

Direct-Mapped Cache

Simplest kind of cache.

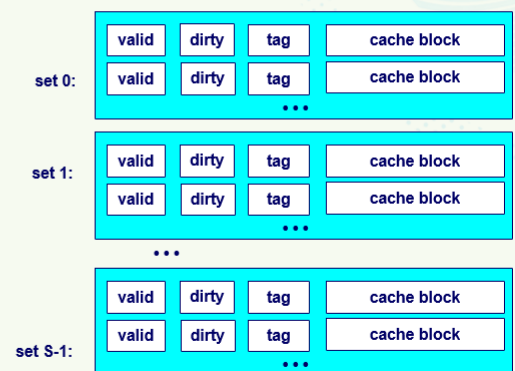
Characterized by exactly one line per set ($E=1$).



22

K-Associative Cache

Characterized by k lines per set.



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

מה שנעשה לרוב: יהיה לא להיות במצב של סט יחיד, ולא במצב של k אלא יותר מאוון.

5.4 תרגול 5

5.4.1 static linking

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

מהם השלבים שקוד שנכתב ב-*C* עובר? מתחילים ב-*main.c*.
א. השלב הראשון הוא *reprocessing* שבמסגרתו כל *define* שיש לנו בקובץ מוחלפים בערכם האמיתי. מתהליך זה יוצא קובץ עם סיומת *i*. *main.i*.
ב. *compiling*: מקבל קובץ ב-*C* ומוציא קובץ באסמבלי *main.s*.
ג. *assembling*: לוקח את קובץ האסמבלי וממיר את הכתובות לשפת מכונה. במהלך תהליך זה מוספות טבלאות כמו שראינו בהרצאה. בסיום שלב זה יש קובץ *main.o*.
ד. *linking*: המטרה של תהליך *linking* שבסיומו מתקבל לנו קובץ הרצה היא שבהינתן *main.o* וקבצי ספרייה נוספים, נרצה ליצור צימוד בין הפונקציות או המשתנים הגלובליים אליהם אנחנו ניגשים לבין המימוש שלהם פיזית (שמופיע בספרייה). *References Resolving*.

library: אוסף של קבצי *.o*. ישנם שני סוגים של ספריות - ההבדל ביניהם הוא בסוג *linking* שיתבצע.

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

חסרונות של static linking:

1. נניח והשתמשנו ב-*printf* בקובץ ההרצה שלנו, ולכן בקובץ ההרצה שלנו הוכנס אליו הקוד של הפונקציה. אממה, אם גילו באג בפונקציה של *c* זה לא ישתנה אצלי. כלומר בשביל שהקוד של הפונקציה יהיה הכי עדכני, נצטרך ליצור קובץ הרצה חדש.
2. חסרון שני - שימוש מיותר מאוד בזכרון. אם למשל אנו כותבים 10 תוכניות שונות שמשתמשות ב-*printf* התקבלו לנו 10 קבצי הרצה שונים שבכל אחד מהם יש *printf* וכשנרץ את קובץ ההרצה יהיה בזכרון 10 מופעים של הפונקציה, חבל על המקום.

ב. *dynamic*: ספרייה *dynamic* היא ספרייה שמופעל עליה *dynamic linking*. לכל פונקצייה או ספרייה בעולם קיים עותק יחיד בזכרון,

נרצה פשוט לגשת אליו באשר אנו מניחים שלכל פונקציה יש צ'אנק כלשהו בזכרון. יש בעיות בכך - אף אחד לא אמר שבכל רגע נתון אנחנו משתמשים בכל הפונקציות שאי פעם נכתבו. חסרון נוסף נובע ממצב בו אולי יורדות גרסאות נוספות לפונקציות, נרצה את ההכי מעודכנת. נראה כי יש מצב שהגרסה העדכנית של *printf* בגודל יותר גדול - יותר בייטים, אבל מתחת לצ'אנק של *printf* הנוכחי יש קוד וגם בצ'אנק מעליו יש קוד לכן נאלץ לחפש מקום חדש בזכרון לכל הפונקציה החדשה. זה לוקח זמן!

לכן נשתמש ב-*dynamic linking* שפועל כך: ב-*static* אמרנו שהצימוד מתרחש באמצעות דחיפה של הקוד לקובץ ההרצה, ב-*dynamic* הצימוד מתרחש בזכרון עצמו. איך? באחת משתי הדרכים הבאות -

א. *load time*: פירוש, שבזמן טעינת התוכנית שלנו לזכרון, מיד הקוד שלנו יטען יחד איתו נקרא ה-*dynamic linker* שקורא את כל הספריות החיצוניות בהם הקוד שלנו מתרחש, הם נטענות לזכרון בדיוק יחד עם התוכנית שלנו.

ב. *run time*: כאן, אנחנו לא נטען ישר את כל הספריות לזכרון, אנחנו נחכה רק לפעם הראשונה שנראה *symbol* בעת הרצת התוכנית, ואז ה-*linker* יטען את ה-*symbol* הנוכחי בזכרון.

כעת נשאל את השאלה הבאה: האם עדיף *load time* או *run time*? אם למשל הקוד שלנו משתמש

בהמון פונקציות, אך בגלל תנאי *if* מזמן רק 3 פונקציות. במקרה זה עדיף לנו כמובן *runTime* לכן: ב-99% מהמקרים נעדיף *runTime*, אך צריך להפעיל שיקול דעת. נראה כי המטרה של *dynamic* לינקינג היא שבכל שלב בזכרון יהיה עותק אחד של כל *symbol*. *dynamic linker* חכם ובעת שיקרא למשל פעם שניה *printf* הוא לא יטען זאת לזכרון אלא יגש לעותק הקודם שהוא טען.

חשוב מאוד: נראה כי אם יש משתנה גלובלי ש-10 תוכניות משתמשות בו - במקרה זה יטענו לזכרון 10 עותקים שונים (!) גם בדינמיק לינקר, בשונה מפונקציות שנטענות רק פעם אחת. לאן *dynamic linker* טוען את הכתובות? לאזור בזכרון שהוא בין *heap* ל-*stack*. נשים לב כי יתכן וכתבתי קוד שמשתמש ב-*scanf* שעוד לא הייתה טעונה בזכרון, לכן הרצתי והפונקציה נטענה לזכרון. נשים לב כי יתכן שנריץ בפעם אחרת את התוכנית, היא תטען למקום אחר בזכרון. אף אחד לא אמר שהמקום שהיא נטענת עליו ישאר קבוע.

ב-*flag* שאנו מעבירים ניתן לקבוע איזה סוג *linking* נבצע וכן האם *runTime* או *loadTime* הוא *default* ו-*dynamic* ו-*runTime*. דיפולט זה נקרא *Lazy binding*.

5.4.2 (Position Independent Code) PIC

כפי שראינו, נרצה שהפונקציות קודם לכן יוכלו להיטען לכל מקום שהוא בזכרון. ננסה להבין איזה טריקים ושטיקים הקומפיילר מבצע כשהוא רואה קובץ *C* עד שהוא הופך אותו לקוד *PIC* באסמבלי: נחלק טריקים אלו לשניים.

א. קוד שניגש ל-*symbol* פנימיים בלבד (רק לפונקציות או משתנים גלובליים או סמליים שהוגדרו באותו קובץ בלבד):

Pc - relative addressing: גישה לכתובות של *symbol* לא באמצעות הכתובת האבסולוטית שלהן בזכרון אלא המרחק שלהן *rip* כרגע. זה נקרא באסמבלי בצורה הזו *str(%rip), %rax*. כיצד הלינקר יודע לחשב את המיקום של כל כתובת? נשים לב כי בין *data* ל-*text* ישנו מרווח קבוע בזכרון. לכן נניח וישנה *global Y* שנמצא כעת ב-*text* אך שמור ב-*data*. אנו יודעים את המיקום היחסי מתחילת *text* עד *global Y* ויודעים את המרחק מתחילת *text* עד תחילת *data* וכן יודעים את מיקומו היחסי של *global Y* ב-*data*. נקח ערך זה, ואת הערך של המרחק מתחילת *text* עד *data* ונחסר את המיקום של *global Y* ב-*text*. זה ההפרש בין המיקומים וכך נדע לעבור בניהם. ב. קוד שניגש ל-*symbol* חיצוניים (יתכן גם פנימיים): כבר בזמן קומפילציה אנו יודעים שיש לנו *symbol* חיצוניים.

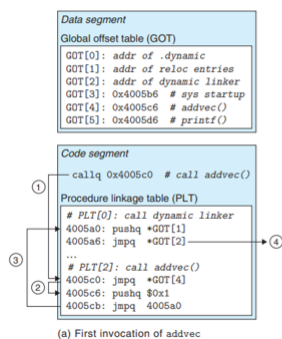
ישנה טבלה בשם *GOT: Global Offset Table*. טבלה שמתווספת ב-*data segment*, בכל שורה ישנם 8 בתים. בעת שהלינקר טען *symbol* הוא יקח את הכתובת של *symbol* שהוא טען וישים בשורה בטבלה. כיצד זה עוזר לנו? אנו מנסים לגשת ל-*symbol* חיצוניים אך לא יודעים היכן הם נמצאים. כעת באמצעות הטבלה נוכל ממש בקוד שלנו לגשת אל *GOT.symbol*. כיצד נוכל בצורה שהיא *PIC* לגשת אל *Got.symbol*? כמו קודם - נוכל לחשב את מיקום. נשים לב כי לפעמים בזמן קומפילציה נוצרים ב-*GOT* סימבולים פנימיים גם - וכן אפשר לגשת אליהם באותה צורה בדיוק (אם כי אין צורך לרשימות *GOT* עבור סימבולים פנימיים, אך זה קורה).

5.4.3 PLT

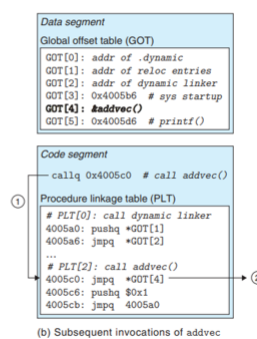
GOT משמשת אותנו לגישה אל סימבולים חיצוניים וכן פנימיים. *PLT* היא טבלה שעוזרת למימוש של *lazyBinding*. היא נמצאת ב-*text segment*. מורכבת מרשימות של 16 בתים שכל אחד מהם מורכב מ-3 פקודות באסמבלי בלבד שמאפשרים את ההבחנה: האם הפונקציה כבר טעונה או שאנחנו צריכים לקרוא ל-*dynamic linker* שיטען אותה.

נראה כי יש הבדל ב-*PLT* בפעם הראשונה שטוענים פונקציה לבין פעמים אחרות. באשר מבצעים קריאה לפונקציה *advac* קוראים לערכה ב-*Got*:

First invocation of *advac*



Subsequent invocations of *advac*



נראה כי אם הלכנו לרשימת *GOT* של *advac* עוד לפני שטענו אותה לזכרון, מתבצע צד שמאל. נראה כי לאחר שטענו אותה לזכרון, במיקום *GOT*[4] אכן ה*dynamic linker* הכניס לשם את הכתובת של *advac* שכבר נטענה. נראה כי תמיד בעת שתקרא *advac* לא משנה באיזה פעם נבצע את *jmpq *Got[4]*.

התפקיד של *PLT* הוא לדעת להבחין האם פונקציה כבר נטענה, ולא צריך לטעון אותה שוב ב*runTime* אלא רק לגשת אליה, או שצריך לטעון אותה.

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

Patching 5.4.4

כיצד נקח קובץ הרצה, ונשנה ממש כמה בתים בו וכתוצאה מכך הוא יתנהג אחרת. (להזכר בדוגמה עם הסטודנטים במבוא שיצרו קובץ חשוב - הרסו אותו כי הם לא חכמים במיוחד, ואני כעת צריך לקחת את הקובץ שהם נתנו לי ולסדר אותו).

6 הרצאה 6

6.1 miss

miss הוא מצב שלא מצאתי *cacheLine* בעת החיפוש ב*cache* אחר מידע ואני נזקק ללכת ל*RAM*.

א. *miss: cold miss* שטוען שהוא לא הכיל עדיין את המידע הספציפי הזה. תמיד יהיה לי *miss* כאלו שכן בהתחלה לא הבאתי ל*cache* כלום. תמיד קורה בפעם הראשונה שניגשים לנתון (נחפש ב*cache* ואין שם כלום). כמעט ואי אפשר לטפל בו - בהמשך נדבר שאם הקוד *cache friendly* אז כן אפשר לטפל בזה.

ב. *conflict miss*: יש לנו מצב של דריסה - מביאים בלוק מהזכרון אל *cache* והוא דורס מידע אחר. באשר *block* מזכרון ממופה למקום תפוס ב*cache* ותופס מידע ששמור שם. זה קורה בגלל אילוצי מיפוי, יש לנו שני נתונים שמתחרים על אותו מקום. זה אומר שהקוד לא *cache friendly* ככל הנראה.

ג. *capacity miss*: מצב שאומר שה*cache* קטן מדי בשביל להכיל את כל המידע שאני צריך. למשל: אם יש לנו הרבה קוד שחוזר על עצמו, הרבה לולאות ומידע. לצערנו ה*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.

דיברנו על כך שה *set* נקבע לפי הביטים האמצעיים. נציע שתי אלטרנטיבות כעת - בואו ונקח את הביטים הימניים ביותר או השמאליים ביותר. האם הם אלטרנטיבות טובות? ההצעה עם הביטים הימניים ביותר - על הפנים. נשים לב שאם ישר נסתכל על השני שורות הראשונות נשים לב שיפגע *locality principle* שכן דברים שיופיעו אחד אחרי השני בקוד יופיעו במקומות שונים.

ההצעה עם הביטים השמאליים ביותר - על הפנים. נניח שיש לנו כתובת $64bits$, אם נסתכל על הביטים השמאליים ביותר והם בהתחלה אפס, אנחנו נקבע שיש המון המון כתובות עד שהביטים השמאליים ביותר יתחלפו, ואז נקבל שיוצר לנו *cacheLine* עומס אדיר של כל הקוד ב *sets* מסויים. לכן, בחרנו בשיטה שראינו עם הביטים השמאליים: ההסתברות להתנגשות תהיה הכי קטנה.

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

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

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

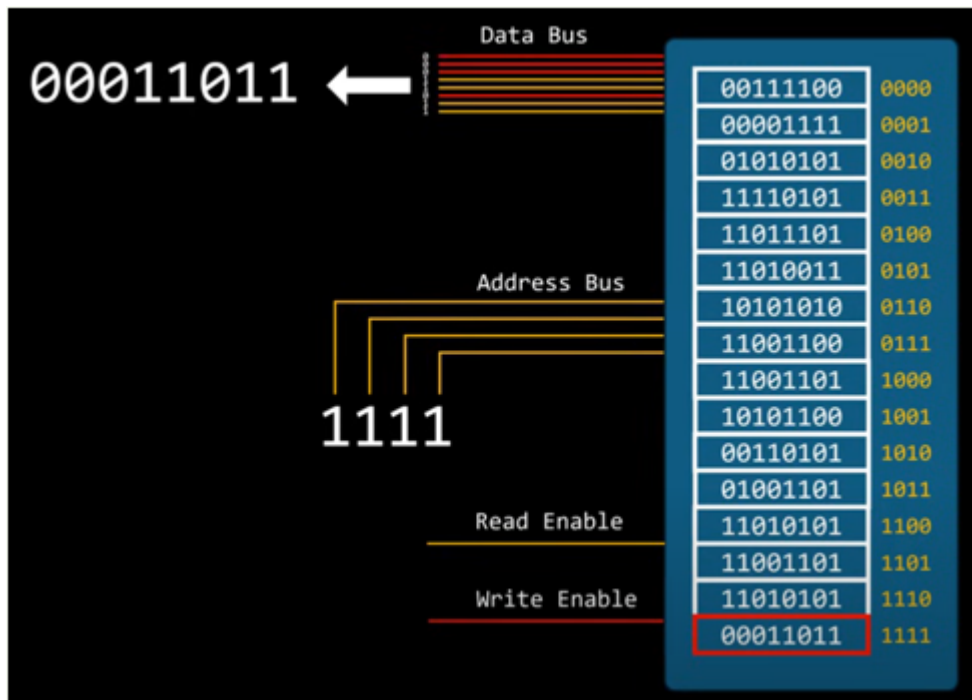
write allocate: כשעושים למשל $x++$, טוענים את הבלוק מהזכרון ל *cache*, אחר כך כותבים אליו בתוך ה *cache* (שם עושים את העדכון) והבלוק נשאר ב *cache* לגישות עתידיות.
no write allocate: כשעושים למשל $x++$ כותבים ישירות לזכרון הראשי. לא טוענים את הבלוק בכלל ל *cache*.

לרוב - נרצה *write back* וכן *write allocate*.

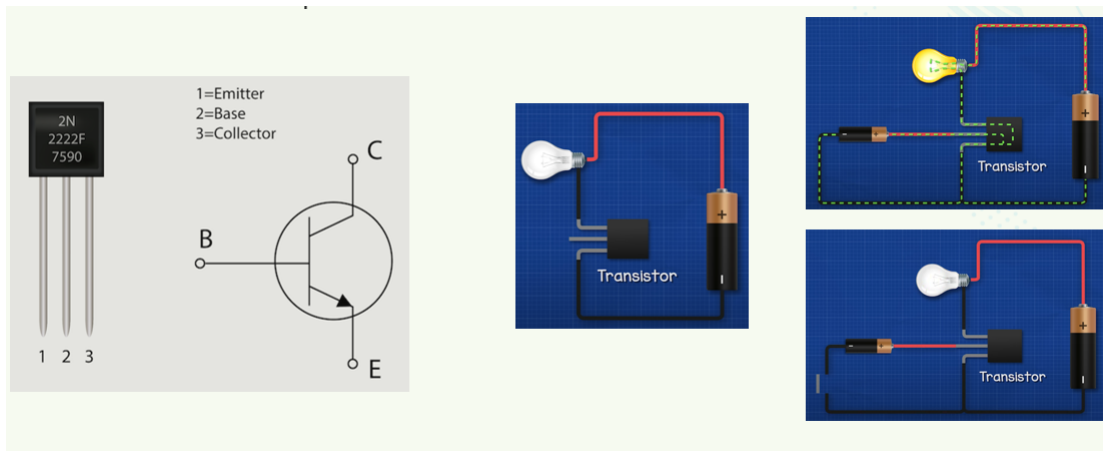
Miss rate: נניח ועשינו 100 גישות ל *cache* ו10 היו *hit* אז אחוז ההצלחה הוא 10%. החלק היחסי של כמה פגענו מתוך כמה ניגשנו לקאש. לרוב ב L_1 מדובר על 10% - 3 וב L_2 פחות מ1%. נשים לב - שזה בתנאי שהקוד הוא *cache friendly*.
Hit Time: כמה זמן לקח לנו לחלץ את הנתון מ *cache*. בדרך כלל: 4 ננו שניות ב L_1 ו10 "ננו שניות" ב L_2 . אם כן, זה עדיף על גישה לזכרון שעלותה 200 - 50 "ננו שניות".

6.2 RAM structure

נדבר על סוגי זכרון שקיימים לנו במחשב. לא נכסה את כולם כמובן.
 נרצה להבין כיצד RAM מחובר לכל דבר אחר במחשב. בינו לבין הCPU באופן ספציפי. מהו BUS? 64 חוטים שעל כל חוט עובר ביט.
 באמצעות *cpu read, write enable* מחליט האם הוא מעוניין לכתוב בזכרון או רק לקרוא ממנו.
 1. אם CPU רוצה לקרוא משהו מRAM הוא שם Address Bus את המידע, מאפשר קריאה ממנו ומשיג את המידע באמצעות הData Bus.
 2. אם CPU רוצה לכתוב משהו לRAM הוא שם Address Bus את המידע, מאפשר כתיבה אליו ושם את המידע בData Bus.
 הdata הזה הולך אל cache - אל החלק הרלוונטי לרגיסטר שהפקודה משתמשת בו.

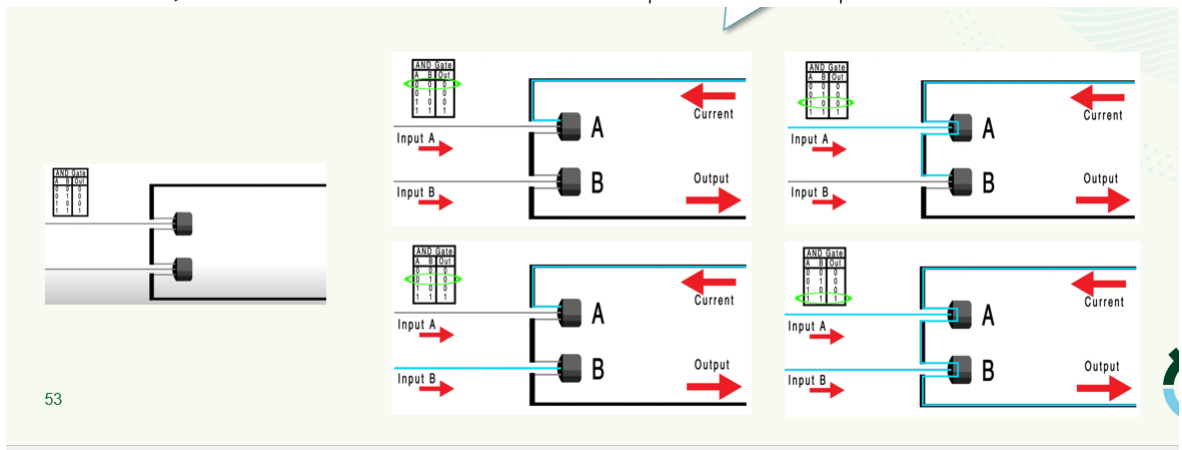


טרנזיסטור הוא רכיב חשמלי שמתפקד כמתג חשמלי או מגבר - הוא יכול להעביר או לחסום זרם חשמלי בהתאם למתח שמופעל עליו, והוא אבן הבניין הבסיסית של כל המעבדים והמעגלים הדיגיטליים המודרניים. כפי שרואים בדוגמה מטה - בשביל להדליק את הנורה צריך לספק לטרנזיסטור חשמל (סוללה כלשהי למשל). השן האמצעית של הטרנזיסטור קובע האם הוא יקבל 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 להבין שניתן לתרגם 8 טרנזיסטורים אל *Byte*! כל ערך של הטרנזיסטור הוא כן או לא, שכן זה כן מעביר חשמל: ביט 1 ולא זה לא מעביר חשמל: ביט 0. כפי שראינו בהרצאה הראשונה - יש או אין חשמל זה לפי טווח מסויים. גם אם יש חשמל לא בטווח מאוד גבוה וגם אם אין זה אומר שיש בטווח מאוד קטן.

מכאן התובנה שכל הזכרון מורכב מטרנזיסטורים. 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. ישנם שני טרנזיסטורים, וכפי שאמרנו הם מייצגים שני *input* שהם ביטים, וכן השן האמצעית זה המידע שעובר לנו מהם. אם יעבור זרם דרך שני הרגיסטרים, משמע שני השנייים האמצעיים יועבר בהם זרם, ולכן "הנורה תדלק", כלומר שני הטרנזיסטורים ידלקו, וכתוצאה מכך יזרום זרם ולכן זה יחזיר 1.

נדבר על שני סוגים של *RAM*.

RAM - *S*: סטטי. החשמל מיוצג (כל ביט) באמצעות 6 טרנזיסטורים - מדוע? החשמל נוטה להחלש והם מקטינים את הירידה בחשמל. (איך? לא רלוונטי לקורס). כתוצאה מכך 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.

RAM - *D*: דינמי. החשמל מיוצג (כל ביט) עם שני טרנזיסטורים בלבד. כן או לא יש

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

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

קעת הבנו מדוע ה-*cache* קטן - הוא עשוי מטכנולוגיה של $S - RAM$ ולכן הוא יקר יותר, ולכן נרצה שהוא יהיה קטן כמה שיותר, בשביל שיהיה מהיר מאוד.

6.3 Disk structure

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

| | SSD (Solid State Drive) | HDD (Hard Disk Drive) |
|-------------------|-----------------------------------|---|
| Price | \$\$\$ | \$ |
| Lifespan | 30-80% bad block in lifetime | 3.5% bad sectors in lifetime |
| Ideal for | High performance processing | Long-term retained data |
| Read/write speeds | up to 12,000 MB/s | up to 200 MB/s |
| Benefits | Higher performance | Less expensive |
| Drawbacks | not as durable / reliable as HDDs | Mechanical components take longer to read-write |
| Energy saving | 2-5 watts | 6-15 watts |

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

6.4 תרגול 6

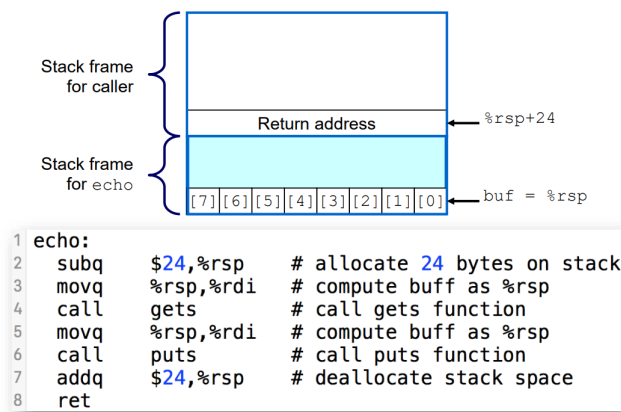
$buffer overflow$: ב-*stack* שומרים מידע וערך חזרה לפונקציה. נראה כי יתכן ותהיה זליגה מהמקום שהוקצה. למשל בפונקציה הבאה - הקצנו ב-*buf* 8 בתים, נניח וה-*input* שהכנסתי אל *get* בגודל 20, אני בהכרח אצא מהמקום שהוקצה לי - כיוון שהפונקציה הנ"ל מניחה כי *dest* הוא מספיק גדול:

```

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

```

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



buffer attack: התקפת *Buffer overflow* מנצלת טעות בפונקציה (כמו *get*) שגורמת לגלישת מאגר. התוקף מזריק למאגר קוד זדוני (*exploitCode*) יחד עם בתים שמשכתבים את כתובת החזרה כך שיצביע על הקוד הזדוני. כשהפונקציה מבצעת `ret`, היא קופצת לקוד התוקף ומבצעת אותו - זו אחת משיטות התקיפה הנפוצות ביותר במערכות מחשב ברשת.

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

7 הרצאה 7

7.1 cache friendly code

נתבונן בדוגמה פשוטה. נניח שיש לנו מטריצה M מגודל $n \times n$. נרצה לחשב סכום של כל איברי המטריצה. אם המטריצה קטנה: לא אכפת לנו איך נקרא אותה. אבל נניח כי n גדול מאוד, זו תהיה הנחה שנרץ בכל מקרה בנושא האופטימיזציות, מניחים:

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

הקוד עובר על המטריצה לפי שורות. מה $Miss\ rate$? נרצה לחשב אותו. על כל 100 גישות ל-cache, כמה פעמים נאלצנו ללכת לזכרון. כיוון שאנחנו מניחים שגודל שורת קאש היא 16 בתים - כל 4 גישות למערך אנחנו צריכים ללכת לram להביא cacheLine חדש ולכן $missRate = \frac{1}{4} \times 100 = 25\%$. אם הקוד יעבור על המטריצה לפי עמודות - $Miss\ rate$ הוא 100%, אנחנו בכל פעם נביא שורה, וכיוון ש N גדול מאוד, כל אלמנט נמצא ב-cache line אחר. לכן בוודאות תמיד נאלץ ללכת לזכרון.

ומה באשר לכפל מטריצות? נשים לב שאלגברית, כפל של שתי מטריצות, הוא לקיחת שורה במטריצה A, ולקחת עמודה במטריצה B, להכפיל אותן ולקבל איבר בודד במטריצת המכפלה. נשים לב שאנחנו בבעיה - מבחינת cache 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 $Miss\ rate$ עבור A, B ו $C = A \times B$?

עבור A - בדיוק 25%, כמו קודם.

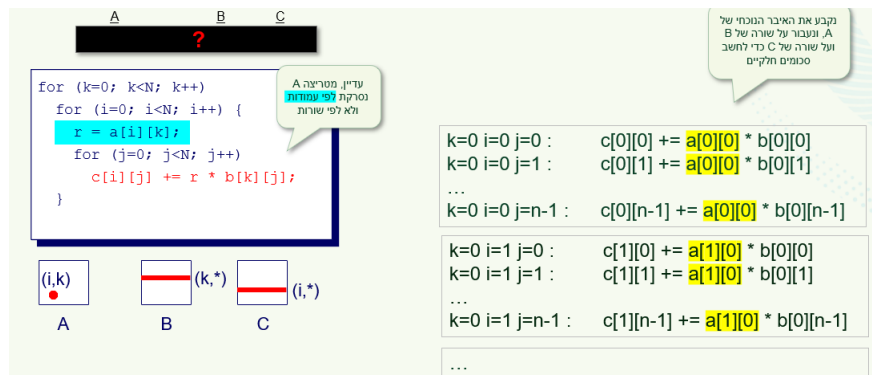
עבור B - בדיוק 100%, לוקחים עמודה, כמו קודם.

עבור C - ניתן לכאורה להתעלם מה $Miss\ rate$. שכן אנו ניגשים אל כל איבר במטריצה C הזו n פעמים (במהלך החישוב, ניגש אליו בלולאה האחרונה n פעמים) ולכן הסיכוי הוא $\frac{1}{n}$ וכיוון ש n גדול מאוד, אזי $Miss\ rate$ שואף לאפס לכן נגיד שהוא 0%.

כיצד נתגבר על העובדה שעוברים לפי עמודות ב? נרצה שאחוז ה $miss$ יהיה לפחות כמו של A,

כלומר 25%. **נתבונן בשיפור הבא:**

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

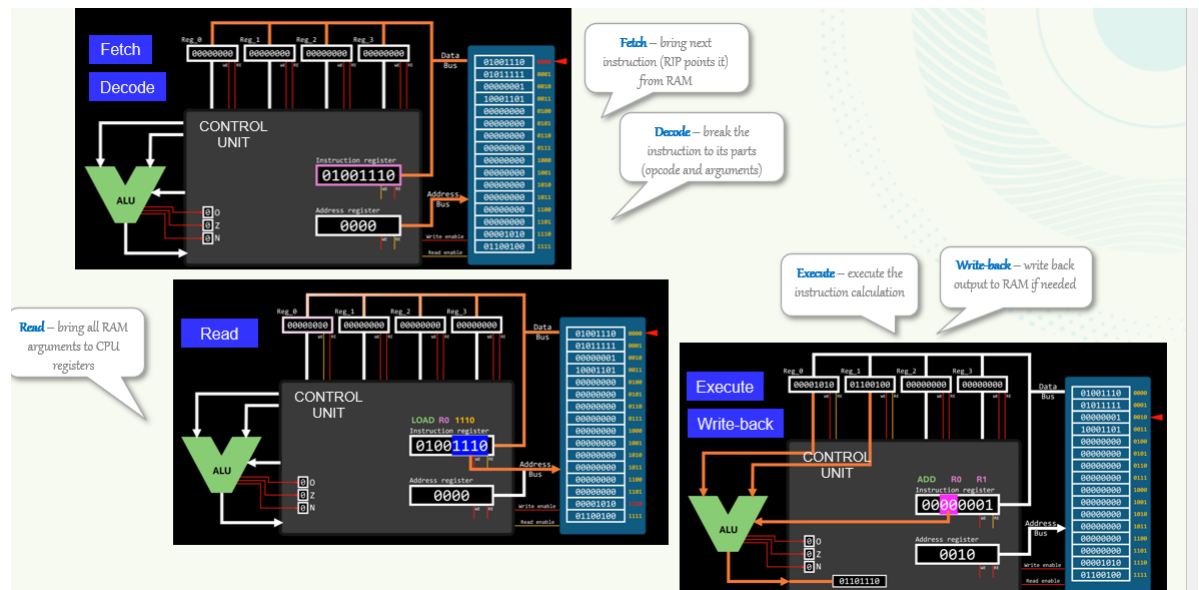


נראה כי בגישה זו עובדים תמיד עם שורות. מה $Miss\ rate$? נשים לב כי עבור A הוא $\frac{1}{n} = 0\%$ (כיוון שאנחנו ניגשים לכל איבר n פעמים, שהרי הסיכוי שלא נגיע הוא $\frac{1}{n}$ - כלומר אתה מביא את האיבר פעם אחת עבור n פעמים) עבור B כעת 25% אך כעת עבור C הוא גם 25% (!) - בכל מקרה: שיפרנו את $Miss\ rate$.

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

Pipeline friendly code 7.2

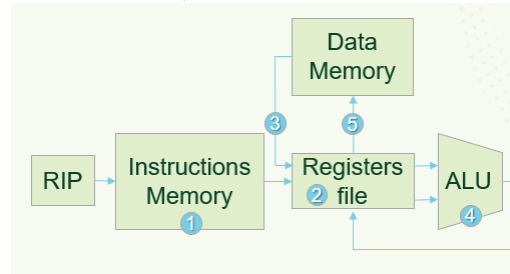
נחזור לדון בחומרה. בחומרה יש ALU , RAM וכן ישנה $Control\ Unit$: יחידה שעושה את כל מה ש- CPU צריך פרט לחישוב עצמו שעושה ALU .



נראה כי הוא אחראי גם לקישור לדטה באס, הוא מביא את ההוראה הבאה וכו'. הוא ממלא לנו את הרגיסטרים שאנחנו משתמשים בהם, והוא מקבל את הפקודות לביצוע. כלומר - הוא שולט על כל

מה שצריך לעשות לפני ואחרי חישוב. **לכל פקודה ישנם 5 שלבים:**
 א. *fetch*: נביא את הפקודה הבאה מהזכרון או *cache* אם התמזל מזלנו באמצעות *RIP*, נשים את הכתובת שיש ב*RIP* בבאס *address*. יהיה מהיר כמובן אם הפקודה ב*cache*.
 ב. *decode*: לוקח את *ISA* ובודק אם ההוראה חוקית או שלא. אם יש לפקודה ארגומנטים בזכרון הוא אחראי לדעת זאת.
 ג. *read*: לא בהכרח תמיד יתבצע, כי לא לכל פעולה יש קריאה מהזכרון. בכל מקרה - מדובר בקריאה מהזכרון.
 ד. *execute*: ביצוע הפעולה. (בהמשך נדון מדוע לא תמיד יתבצע).
 ה. *write*: אם עלינו לכתוב משהו חזרה לזכרון או *cache*, זה יקרה בשלב הזה.

ב *CPU* כל השלבים הללו נעשים ע"י חלקים שונים של החומרה.

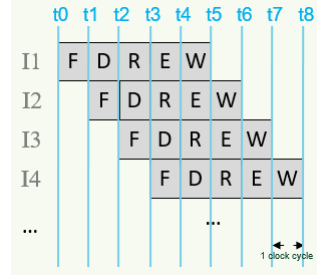


השלבים הללו שכל פקודה מבצעת - הם בלתי תלויים זה בזה.

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



לעומת זאת, אם נבצע אותם בצורת מדרגה, זה יראה כך:

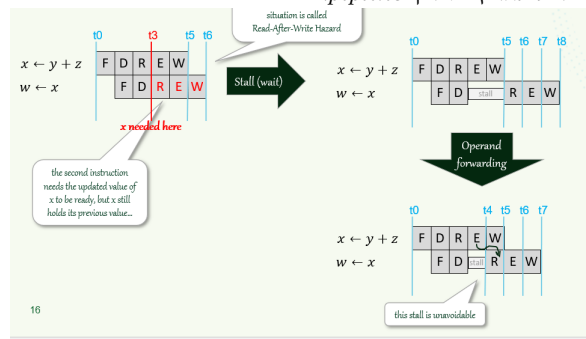


כלומר, לאחר שנסיים *fetch* של ראשון נעבור ישר ל*fetch* של השני. זו צורת *PIPELINE*, זה עדיף מאשר לבצע רצף סדרתי של השלבים ולאחריהם שוב. בצורה זו מס' *cycles* יורד! נראה כי במקרה הראשון עלה לנו 20 *cycles* וכאן רק 8 בלבד. באופן כללי, השיפור יהיה לפי הנוסחה הבאה:

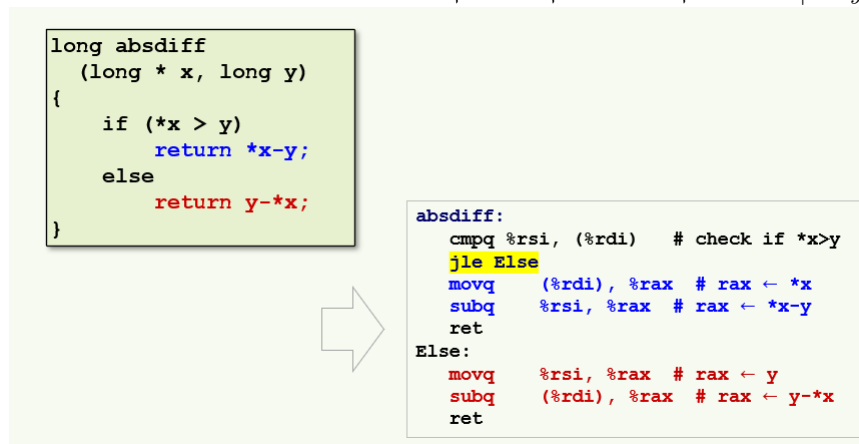
$$\lim_{n \rightarrow \infty} \frac{5n}{4+n} = 5$$

שכן, זמן הריצה הסדרתי יהיה $5n$ (5 כפול n פקודות), וכן $n + 4$ זה זמן הריצה באופן של Pipeline (ממלאים 4 ראשונים ואז מתחילים את השאר), ולכן השיפור שנקבל באופן כללי אם נעבוד לפי pipeline יהיה פי 5.

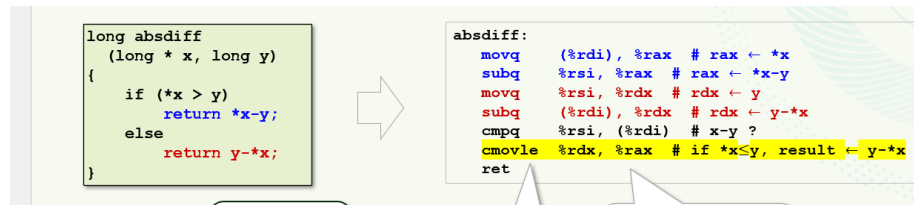
אם כן, נראה ששיפור זה לא תמיד יעבוד. נתבונן בדוגמה מטה: יש תלות בין הפקודות, בשביל לבצע את השורה השנייה צריך לדעת את השורה הראשונה בכדי לדעת את x . נראה כי כאשר נבצע read לפקודה השנייה, זה בדיוק השלב שהפקודה הראשונה מבצעת *execute* - היא בדיוק מחשבת את הערך. זה מצב שנקרא *stall(wait)* - הפעולה הראשונה חייבת לחכות (!). לבסוף - לא תמיד אנחנו נוכל להריץ באופן pipeline.



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



נראה כי ישנה אפשרות נוספת לקוד אסמבלי זה. נרצה שלא לחשב את שני החישובים כמו קודם וקומפר, אלא מראש להכין את הכחול והאדום, ולהשתמש ב *cmovle* כאשר *le* הוא האם קטן שווה מאפס. שקול לכך שנבדוק אם $y > x$ ולהכניס את התוצאה ל *result*.



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

3 מקרים מסוכנים בהם לא נשתמש 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.

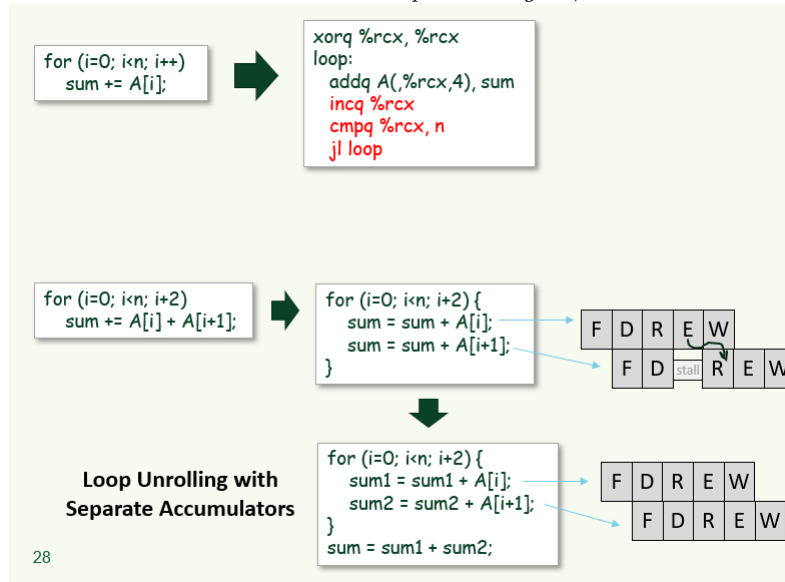
המקרה השני - יתכן שהחישוב יגרם לשגיאת זמן ריצה אם נבצע אותו ללא בדיקת התנאי. למשל,

נגש ל*pointer* שהינו *null* עוד לפני שבדקנו אם הוא לא *null*.

המקרה השלישי - כנ"ל, אנחנו מראש מחשבים גם את *if* וגם את *else*, ואיך נדע לחשב מראש

את $x * 7$ ואת $x + 3$? אנחנו פשוט לא! ולכן אסור להשתמש במקרה זה!

נראה אופטימיזציה נוספת: במקום לרוץ על n נרוץ על $\frac{n}{2}$ איברים. מה קיבלנו כאן? מטפלים בלולאה פעם בשתי איטרציות ולא פעם באיטרציה. וכן אנחנו בוחרים להשתמש בשני סכומים בשביל שנמנע מהבעיה שנוצרת בקוד שבאמצע: הוא חייב לחכות לפני שמתקדם ונוצר *stall* בגלל שהוא חייב לקרוא את הנתונים ולא יכול להתקדם. אם משתמשים בשני משתנים מקבלים את אותה התוצאה בזמן הרבה יותר טוב. זה נקרא *loop unrolling*.



| | |
|-------------------------------------|------------|
| <i>RAM friendly code</i> | 7.3 |
| <i>compiler & optimizations</i> | 7.4 |
| הרצאה 8 | 8 |
| הרצאה 9 | 9 |
| הרצאה 10 | 10 |
| הרצאה 11 | 11 |
| הרצאה 12 | 12 |