סיכום נאנד

1 הרצאה 1:

- **מעבר מפונקציה לטבלת אמת:** בכדי לעבור מפונקציה לטבלת אמת נצטרך להציב את ערכי כל המשתנים ולבדוק איזה ערך מתקבל מהפונקציה עבור כל מצב. ניתן לפשט זאת אם בפונקציה מופיע האורטור $x\ and\ y$ וכך בכל מקום שבו מופיע משתנה $x = 0$ או $y = 0$ נדע שערך הפונקציה הינו 0.

- **מעבר מטבלת אמת לפונקציה DNF:** נסתכל על השורות בטבלה עבורת מתקבל הערך 1. נבנה פונקציה המארת את המצב הנוכחי ונתייחס רק לשורה הזו. נעשה זאת בצורה הבאה - אם $x = 0, y = 0 \Rightarrow 1$ נמיר זאת ל $not(x)\ AND\ not(y)$ כלומר - בכל פעם נשתמש רק באופרטורים $NOT\ AND$.

- **השער $Namd$:** זהו שער לוגי המתאר מצב של $x\ Nanx\ y \Rightarrow NOT(x\ AND\ y)$.

- **למה:** כל פונקציה בוליאנית ניתנת לייצוג בעזרת השער $Nand$ בלבד.

2 הרצאה 2:

- **חיבור מספרים בינארים:** בכדי לחבר מספרים בינארים נשתמש באותו האופן של חיבור דצימלי. נזכור את המספר הבא ונעלה אותו למעלה ונכתוב אותו בייצוג בינארי.

- $overflow$: מצב שמתרחש כאשר חיברנו שני מספרים והמספר שקיבלנו גדול יותר במספר הביטים מהמספרים המחוברים. אזי אם אורך המילה במחשב מוגבל - הביט הנוסף לא יוצג ויספר.

- **ייצוג:** עבור מספר בעל n ביטים - ניתן לייצג את כל המספרים מ 0 עד 2^{n-1}.
  באופן כללי: 2^n זה המספר 1 ומימינו יש n אפסים. והמספר 2^{n-1} זה המספר המיוצג בעזרת n 1־ים.

- **ייצוג מספרים שליליים:** נייצג את המספרים השליליים בעזרת אותם ייצוגים בינארים. עבור $n = 4$ המספרים הראשונים ייצגו את 7־0. והמספרים הבאים ייצגו את 8־ עד 1־ . הקוד של $-x$ יהיה $2^n - x$
  מעבר לדצימלי: עבור חיוביים נחזיר רגיל. עבור שליליים - נחזיר $x - 2^n$.

- **חיסור מספרים בינארים:** באותו האופן של חיבור, נשתמש בשיטת הייצוג הקודמת של מספרים שליליים ונעבוד באופן הבא: $(x + (-y))\ mod\ 2^n = x - y$ (עבור nמספר הביטים).
  אם יש אוברפלואו - נתעלם מהביט הנוסף, זאת המקבילה ללבצע מודולו בדצימלי.

- **מעבר למספר ההופכי בבינארי:** נחסר מהמספר 2^nאת הספרה 1, למעשה זה להפוך את כל הביטים מ 1 ל 0 ולהיפך. ולאחר מכן נוסיף 1 לתוצאה. המספר שיתקבל הוא ההופכי.

- **הוספת 1 ל** x**:** $x+1$ = נתחיל מהביט הימני ביותר ונהפוך אותו, אם קיבלנו 1 - נעצור. אחרת - נשיך לביט הבא. נעצור כשהביט שהפכנו הפך ל 1.

- ALU **- יחידה אריתמטית לוגית:** זה הצ'יפ הראשי שמנהל את כל החישובים של המחשב (הוא נמצא בתוך המעבד המרכזי - CPU). אנו מכניסים לו שני ערכים - x, y ו 6 $control\ bits$ ועל פיהם הצ'יפ מחשב את מה שנרצה.

3 הרצאה 3:

- **ייצוג זמן במחשב:** בכדי להשתמש בזיכרון אנו צריכים לייצג זמן במחשב.

- **ציפ זכרון - אוגר** $Register$**:** יש ציפים המסוגלים להתייחס לזמן וכך לזכוא דברים. נייצג אותם בדרך הבאה: יש להם in שאליו מכניסים קלט, בנוסף קיים משתנה $load$ שמעדכן האם הציפ פתוח להכנסת מידע או לא. אם כן - נכניס מידע וביחידת זמן הבאה המידע החדש ייצא מהציפ, אחרת - מידע לא יוכל להיכנס והמידע ששמור ייצא מ out.

- $\boldsymbol{RAM}$**:** הזכרון המרכזי של המחשב. הוא מכיל n אוגרים, ויש לו קלט - in, , פלט - out וקלט של כתובת מ 1 עד n.

- **מונה -** PC**:** הוא נמצא בתוך המעבד המרכזי - CPU. הוא רגיסטר ששומר את הפקודות הבאות שהמחשב צריך לבצע.

- DFF **- שער פליפ פלופ:** שער שהתכונה שלו היא להוציא בזמן t את מה שנשמר בו בזמן $t-1$.

- **כיצד נממש** $register$**:** נממש ציפ עם שעק DFF שמוחזר לעצמו ב in דרך שער לוגי Mux. כך למעשה נוכל לשלוט על הוצאה, כתיבה ושמירת ערך לאורך זמן.

- **כיצד נממש** RAM**:** נשתמש בכמות אוגרים נדרשת, נחבר בכניסה שער $DMux$ וביציאה שער Mux, כך נוכל לשלוט על כתיבה והוצאת מידע.

4 הרצאה 4:

- **יחידות הזכרון והפקודות:** המחשב אותו נבנה בקורס - $hack$ משתמש בשתי יחידות זכרון - RAM: משמש לזכרון דאטה. ROM: משמש לזכרון פקודות. בנוסף המחשב משתמש בשלשה רגיסטרים :
A**:** רגיסטר המקבל כתובת בזיכרון וניגש לכתובת הזו ב RAM.בכדי לגשת לרגיסטר a נשתמש בפקודה @x כאשר x מייצג כתובת בזכרון. ניתן להשתמש ברגיסטר זה בתור יחידת זכרון נוספת ולא רק ע"מ לגשת לזכרון.
M**:** הרגיסטר אותו בחרנו לפי הפקודה שנתנו לרגיסטר A. כלומר זהו רגיסטר x.
D**:** רגיסטר נסוף שקשור לחלק של הפקודות בזכרון - ROM. בשונה מרגיסטר A לא ניתן לכתטב עליו ישירות, אלא נצטרך לכתוב קודם על A ואחת לכתוב את הפקודה $M=A$.

- **סימבולים - משתנים:** ניתן להגדיר $@sum$ במקום לתת מספר קונקרטי של כתובת בזכרון, וכך הסמל sum יתורגם לכתובת בזכרון מאחורי הקלעים.

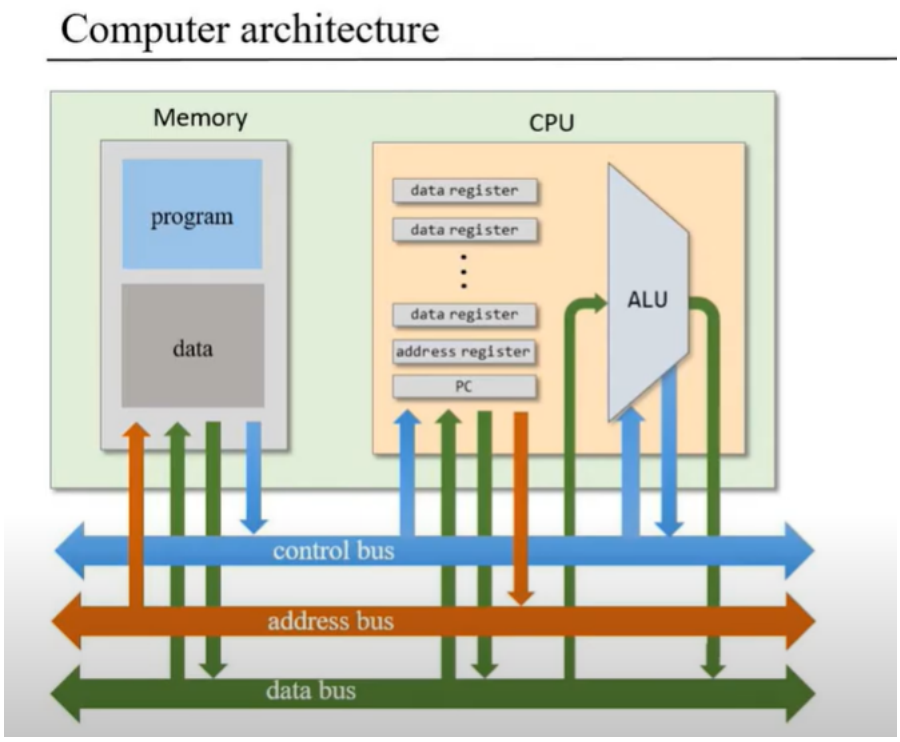
- **פקודות קפיצה:** פקודות שאומרות למחשב לקפוץ לכתובת הבאה בזכרון אם מתרחש תנאי כלשהו (או ללא תנאי).
הפקודות יתחילו כך: ראשית נטען את הכתובת אליה נרצה לקפוץ אחכ נכתוב את הפקודה - $D; J..$ כאשר D יסמן את התנאי אותו נבדוק מול 0. (אם במקום D מופיע 0 אזי הקפיצה תתרחש ללא תנאי)

5 הרצאה 5:

- PC: רכיב שנמצא ברגיסטרים שב CPU והוא שומר את הכתובת של הפעולה הבאה אותה צריך לבצע.

- **מעבר הנתונים במחשב:** מתרחש בעזרת שלש נתיבים מרכזיים:
$data\ bus$: מחבר בין הזכרון לרגיסטרים שיושבים בתוך ה CPU ול ALU למעשה ב"צינור" זה עוברים הקלטים והפלטים מה ALU.
$control\ bus$: מחבר בין הזיכרון ל ALU ומעביר לו את הביטים של הקונטרול שאומרות לו איזה פעולות צריך לבצע על הקלטים.
$address\ bus$: מעביר את הכתובת הבאה מה PC אל הזכרון. לאחר מכן המידע יוצא אל ה ALU בעזרת ה $control\ bus$.

- **סכימה ככללית:**

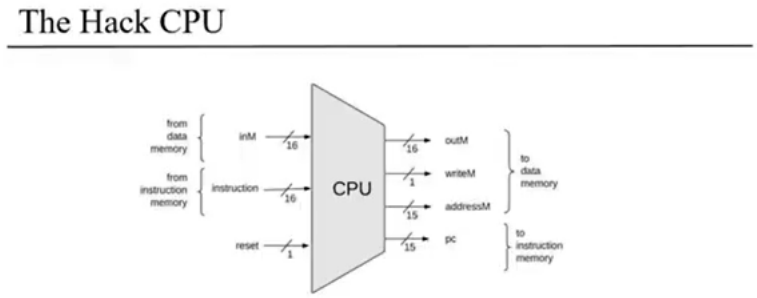


- $cash$: זכרון ביניים שמכיל בלוק של פקודות, ונמצא בין הזכרון ל CPU.

- $fetch - execute$: לולאה שבה אנו שולפים פקודה מהזכרון - $fetch$. ואחכ מבצעים את הפקודה ששלפנו ומחשבים מה הפקודה הבאה - $execute$.
  ע"י השעון ניתן להחליט איזה פעולה לבצע בכל פעם, כך הלולאה לא תיעצר ותימשך תמיד.

- $\boldsymbol{Feetch - Execute\ clash}$: התנגשות שנוצרת בין ה $fetch$ ל $execute$ משום שהזכרון צריך להוציא במקבי גם מילהוגם כתוב בזכרון. ומצד שני הוא צריך להכניס את הכתובת שעליה צריך לפעול בזכרון וגם את הכתובת שצריך כדי לבצע את הפקודה הבאה.
  יש שתי גישות בכדי לפתור את ההתנגשות:
  1: שני מעגלים: ניצור שני מעגלים עם Mux בכניסה ו $DMux$ ביציאה שינתנו לנו את הדאטה לפי הצורך בכל פעם - לפי תקתוקי השעון.
  2: הפרדת זכרון: במקום יחידת זכרון אחת שתייצג את הפקודות ואת הדאטה, נפריד אותן לשתי יחידות זכרון. כך למעשה נוכל לנהל בכל במעגל יחיד. החסרון הוא שאם יהיה לנו זכרון מיותר לא נוכל להפנות אוו לשימוש השני. לכן שיטה זו נפוצה במחשבים שצריכים לבצע סוג פעולות אחד בלבד.

- **מעבד -** CPU: מבצע את הפקודה הנוכחית, בנוסף הוא יודע לחשב איפה נמצאת הפקודה הבאה שהוא צריך לבצע. יש לו שלש כניסות של קלטים, וארבע יציאות של פלטים.
  דיאגרמת CPU:



6 הרצאה 6 - אסמבלר:

- **בשפת** $hack$ **יש לנו:**
  פקודות - טיפוס A/C.
  סמלים $simbols$ - מוגדרים מראש (חלק מהשפה), מסמנים תוויות, מסמנים משתנים.
  הערות או שורות ריקות.
  כשנתרגם לקוד בינארי נצטרך להתמודד עם שלשת הדברים הללו.

- **תרגום פקודות מטיפוס** A: נקח את המספר הדצימלי ונמיר אותו לבינארי, במידת הצורך נוסיף 0־ים כדי להגיע ל 16 ביט.

- **תרגום פקודות מטיפוס** C:
  פקודות מטיפוס C נתרגם לפי הטבלאות הבאות כאשר הביטים מסמנים את הפקודה ואת היעד.

Translating C-instructions

Symbolic syntax: *dest* = *comp* ; *jump*

comp is mandatory.
If *dest* is empty, the = is omitted; If *jump* is empty, the ; is omitted

Binary syntax: 1 1 1 a c c c c c c d d d j j j

(bit positions: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0)

comp		c c c c c c
0		1 0 1 0 1 0
1		1 1 1 1 1 1
-1		1 1 1 0 1 0
D		0 0 1 1 0 0
A	M	1 1 0 0 0 0
!D		0 0 1 1 0 1
!A	!M	1 1 0 0 0 1
-D		0 0 1 1 1 1
-A	-M	1 1 0 0 1 1
D+1		0 1 1 1 1 1
A+1	M+1	1 1 0 1 1 1
D-1		0 0 1 1 1 0
A-1	M-1	1 1 0 0 1 0
D+A	D+M	0 0 0 0 1 0
D-A	D-M	0 1 0 0 1 1
A-D	M-D	0 0 0 1 1 1
D&A	D&M	0 0 0 0 0 0
D|A	D|M	0 1 0 1 0 1
a == 0	a == 1	

dest	d d d	effect: the value is stored in:
null	0 0 0	the value is not stored
M	0 0 1	RAM[A]
D	0 1 0	D register
DM	0 1 1	D register and RAM[A]
A	1 0 0	A register
AM	1 0 1	A register and RAM[A]
AD	1 1 0	A register and D register
ADM	1 1 1	A register, D register, and RAM[A]

jump	j j j	effect:
null	0 0 0	no jump
JGT	0 0 1	if *comp* > 0 jump
JEQ	0 1 0	if *comp* = 0 jump
JGE	0 1 1	if *comp* ≥ 0 jump
JLT	1 0 0	if *comp* < 0 jump
JNE	1 0 1	if *comp* ≠ 0 jump
JLE	1 1 0	if *comp* ≤ 0 jump
JMP	1 1 1	Unconditional jump

Example: Symbolic: `M=1` → Binary: `1110111111001000`

Nand to Tetris / www.nand2tetris.org / Chapter 6 / Copyright © Noam Nisan and Shimon Schocken

- **תרגום הערות:** כל שורה שמתחילה ב // או שורה ריקה נתעלם ממנה.

- **תרגום סמלים** $simbols$**:**

סמלים מוגדרים: בשפת $hack$ יש לנו 23 סמלים שיש להם מספר מוגדר מראש בשפה:

The Hack language features
23 *predefined symbols*:

symbol	*value*
R0	0
R1	1
R2	2
...	...
R15	15
SCREEN	16384
KBD	24576
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

Translating @*preDefinedSymbol* :

סמלים שמסמנים תוויות: , לדוג' $(LOOP)$: יוגדרו בתוך סוגריים עגולים, למעשה הם מסמנים את הכתובת של הפקודה הבאה בתכנית.

נתרגם אותם באופן הבא - נספור את מספר השורות החוקיות בתכנית (לא כולל הערות, שורות ריקות והתוויות עצמן) וכאשר ניתכל בתווית נחליף את התווית במספר השורה שבה מופיעה פקודת התוית.

סמלים המסמנים משתנים: נחלק להם כתובות החל מ 16 והלאה. כך שאפ ראינו אותו פעם ראשונה - נתן לו ערך, אחרת - ניתן לו את הערך השמור.

- **כיצד נבצע:**

שלב אתחול: נבנה מילון (טבלה) שבה נשים את כל המילים השמורו אצלנו בתכנית (בשפה לנו יש 23 סמלים שמורים).
ריצה ראשונה: נמספר את השורות, וכל פעם שניתקל ב "(" נוסיף את התווית למילון עם המספר המתאים לה.
ריצה שניה: נעבור על הקלט, כשנמצא סימן נבדוק אם הסימן נמצא בטבלה - אם כן נמשיך, אחרת נתן לו אינדקס החל מ 16 ונקדם את האינדקס.
ריצה שלישית: נתרגם את השדות של פקודות C לבינארי.

7 הרצאה 7:

- **כיצד נבנה קומפיילר:** אנו נכתוב על המחשב שלנו בשפת ג'ק, אותה נתרגם לאסמבלי ואת האסמבלי נתרגם לשפת מכונה.

- **מכונת מחסנית:** חלק מזכרון המחשב שבו נשמור ערכים שנוכל לבצע עליהם פעולות אריתמטיות (חיבור, חיסור ושלילה). פעולות לוגיות (גדול, קטן ושווה. and, or, not) ופעולות $pop, push$ בכדי להכניס ולהוציא ערכים מהמחסנית. הפעולות האריתמטיות והלוגיות פועלות על שני הערכים שבראש המחסנית. התכנית תשמור מצביע שיצביע לסוף המחסנית (התא הריק הבא).

- **מצביע למחסנית:** המצביע יצביע על התא הריק הבא במחסנית - הטופ.

- **פעולת** pop: כשנקרא ל pop הערך ששמור בראש המחסנית ייצא ויאוחסן בערך שהצמדנו למילה pop. לדוגמה הפקודה $pop\ y$ תאחסן לנו ב y את הערך ששמור בראש המחסנית.

- **פעולות אריתמטיות:** כשנבצע פעולות אריתמטיות על המחסנית, מה שיקרה מאחרוי הקלעים זה - מספר הערכים שאנו צריכים ישלפו מהמחסנית (שני ערכים אם האופרטור בולאני, ואחד אם הוא אונארי). נפעיל עליהם את האופרטור ונעשה $push$ לערך החדש.

- **מילים שמורות:** כשנרצה להוסיף משתנה למחסנית נגדיר את הסוג שלו - $static, local, global$. כך הקומפיילר ידע לאיזו מחסנית להכניס את המשתנה, (למעשה יש לנו 8 סוגים שונים של מחסניות).

- $memmory\ segments$: כשנרצה להוסיף למחסנית ארגומנט של פונקציה נכתוב כך: $push\ argument\ n$ כאשר n מייצג את מספר הארגומנט. אין צורך לכתוב את הערך של הארגומנט אלא רק את מספר הארגומנט.
באותו האופן, כשנרצה להכניס ערך למשתנה לוקאלי נכתוב $pop\ local\ n$ כאשר n מייצג את מספר המשתנה הלוקאלי אותו אנו רוצים לעדכן.
כשנרצה להוסיף למחסנית קבוע נכתוב כך: $push\ constant\ x$ כאשר x הוא הערך.

- **כיצד עובד הזכרון:**
1 - מחסנית: בתא $RAM[0]$ ישמר לנו המצביע למחסנית - $SP(stack\ pointer)$. המחסנית תאוחסן החל מ $RAM[256]$.
2 - משתנים לוקאלים: בתא $RAM[1]$ יאוכסן המצביע LCL אשר יצביע למחסנית שתצביע למשתנים לוקאלים (לא

אכפת לנו היכן המחסנית תאוחסן כל עוד יש לנו מצביע שנשמר). **נשים לב:** בשונה מ SP, המצביע LCL יצביע **לתחילת** המחסנית (ולא לתא הפנוי הבא).

כשנרצה להכניס ערך למחסנית של המשתנים הלוקאלים נעשה זאת כך: $pop\ local\ n$, כעת נבצע $RAM[1]+n$ וזה יביא אותנו לתא הפנוי במחסנית הלוקלית, התא שבו נאחסן את הערך החדש.

3 - ארגומנטים : $RAM[2]$ ישמור מצביע לארגומנטים של המתודה הנוכחית עליה אנו עובדים. (יצביע **לתחילת** המחסנית (ולא לתא הפנוי הבא)).

4 - $This$: ישמר ב $RAM[3]$ והוא ישמור לנו את משתני המחלקה - שדות, האובייקט הנוכחי. (יצביע **לתחילת** המחסנית (ולא לתא הפנוי הבא))

5 - $That$: ישמר ב $RAM[4]$, במידה ואנו עובדים במטודה הנוכחית על מערך, כתובת המערך תישמר ב $That$ (יצביע **לתחילת** המחסנית (ולא לתא הפנוי הבא)).

- **מחסנית למשתנים זמניים - $temp$:** משתנים זמניים שאנו צריכים לעשות עליהם חישובים ישמרו במקטע $RAM[5]-RAM[12]$.

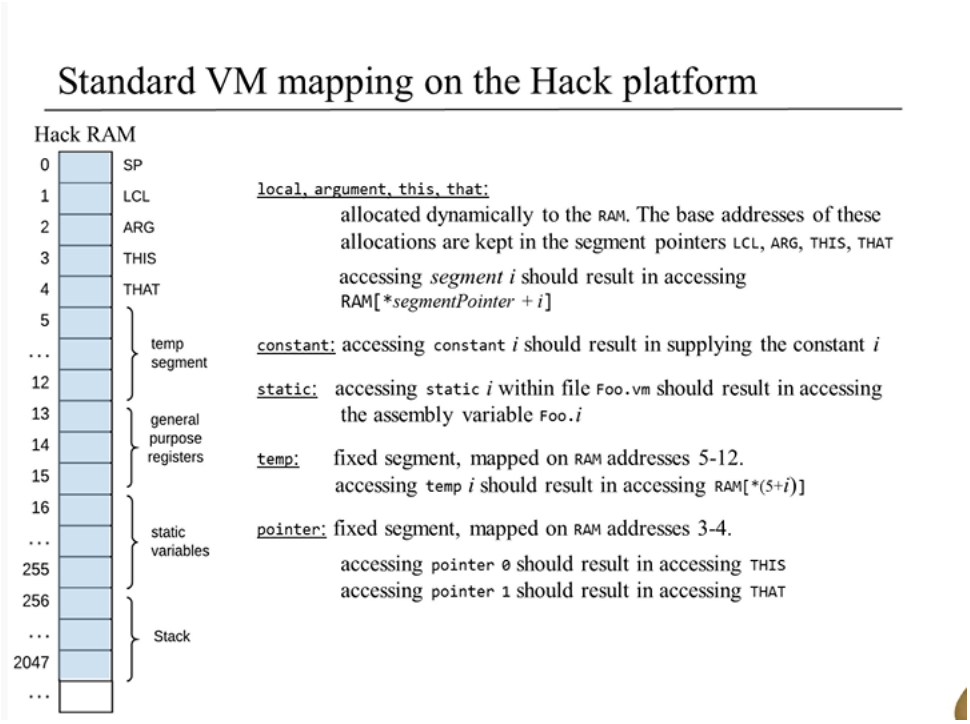
- **מחסנית למשתנים סטטים:** כשנרצה לשמור משתנים סטטים נתרגם אותם לאסמבלי כך:

$$ststic\ 5 \Rightarrow @Foo.5\ M=D$$

כל המשתנים הסטטים ישמרו לנו בסקופ גלובלי בין $RAM[16]$ ל $RAM[255]$ ולכל התכנית תהיה גישה אליהם. למעשה הם יתמפו באופן אוטומטי בלי שנצטרך לשמור להם מקום מסויים בזכרון.

- **מצביעים - $pointer$:** המצביע $pointer$ יכול לקבל שני ערכים בלבד - 0,1 . כשנכניס $pointer\ 0$ נתייחס ל $This$ (משתני מחלקה), וכשנכניס $pointer\ 1$ נתייחס ל $That$ (המערך שעליו המטודה עובדת).

- **כך נתרגם משפת ג'ק בעזרת VM:**



Symbol	*Usage*
SP	This predefined symbol points to the memory address within the host RAM just following the address containing the topmost stack value.
LCL, ARG, THIS, THAT	These predefined symbols point, respectively, to the base addresses within the host RAM of the virtual segments `local`, `argument`, `this`, and `that` of the currently running VM function.
R13–R15	These predefined symbols can be used for any purpose.
`Xxx.`*i* symbols	The `static` segment is implemented as follows: each static variable *i* in file `Xxx.vm` is translated into the assembly symbol `Xxx.`*i*. In the subsequent assembly process, these symbolic variables will be allocated to the RAM by the Hack assembler.

8 הרצאה 8:

- $Branchimg\ commands$: פקודות אלו נותנות לנו את חופש הפעולה לקפוץ למקומות שונים בקוד אם תנאי כלשו מתקיים או לא.

הפקודות הן - $label, goto, if-goto$. יש שני סוגים:

1 **Unconditional branching** : פקודת $goto\ LABEL$ כשהמחשב יראה את הפקודה הזו הוא יקפוץ ישירות ל $label$ שכתוב לאחר הפקודה.

2 **Conditional branching**: פקודת $if-goto$ מתבצעת רק אם התנאי מתקיים, לכן נצטרך קודם כל לעשות $push$ לתנאי, ואז לשים את הפקודה. כך אם הפקודה מתקיימת נלך ל $label$ הנכון.

8.1 $Functions$:

- **פקודת** $call$: לפני שנקרא לפונקציה שמקבלת n ארגומנטים - נעשה $push$ למחסנית של n הארגומנטים, לאחר מכן נכתוב $call\ funcName\ n$ כך המערכת יודעת על כמה ארגומנטים מהמחסנית צריך להפעיל את הפונקציה.
מאחורי הקלעים הפונקציה תפעל על n ארגומנטים, ותעשה $push$ לערך שהפונקציה החזירה.

- **פקודת** $function$: כשנקרא לפונקציה נעשה זאת כך $function\ funcName\ n$ כאשר n מייצג את מספר המשתנים הלוקאלים שהפונקציה משתמשת בהם.

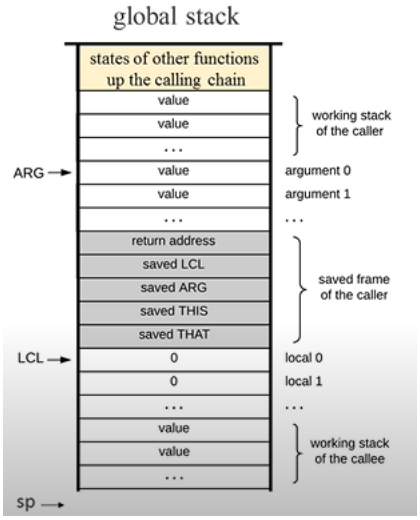
- **פקודת** $return$: בסוף פונקציה אנו רוצים שהפונקציה תחזיר את הערך שהיא חישבה לפונקציה שקראה לה. לכן הקומפיילר יקח את הערך האחרון מהמחסנית של הפונקציה שנקראה $(callee)$ וישים אותו במחסנית של הפונקציה הקוראת $(caller)$.

- **כיצד נממש** $return$ **ו** $call$:

For each function `call` during run-time, the implementation has to ...
- Pass parameters from the calling function to the called function;
- Determine the return address within the caller's code;
- Save the caller's return address, stack and memory segments;
- Jump to execute the called function;

For each function `return` during run-time, the implementation has to...
- Return to the caller the value computed by the called function;
- Recycle the memory resources used by the called function;
- Reinstate the caller's stack and memory segments;
- Jump to the return address in the caller's code.

- **מימוש:** נגדיר $states$ עבור כל פונקציה שיכיל את המשתנים הלוקאלים ואת הארגומנטים שהפונקציה צריכה. נעשה זאת עבור כל פונקציה ברגע שהיא נקראת, ונמחק\נשחרר אותו בסוף הריצה.
נממש את זה בשיטת $LIFO$, בעזרת מחסנית. כאשר נקרא לפונקציה מסויימת נדאג שהמצביע ARG יצביע לארגומנט הראשון שאנו צריכים להכניס לפונקציה (עשינו לפניכן $push$ לכל הארגומנטים שנצטרך). כך למעשה כל מה שמעל ARG שייך ל $caller$, ומה שמתחת שייך ל $callee$.
כעת אנו צריכים לשמור את ה $memory\ segments$ ואת ערך ההחזרה של ה $caller$. נעשה זאת עם מחסנית בה נשמור את הפויינטרים של הפונקציה $(LCL, ATG, THIS, THAT)$ ונמקם בסוף את המצביע LCL.
לאחר מכן נוכל לקפוץ ל $callee$ ולבצע את הפעולה שלה.
כך תיראה המחסנית:



- **מימוש חזרה:** כשנחזור מהפונקציה נעדכן את כל הפויינטרים לפפויינטרים השמורים של ה $caller$, לאחר מכן נשים את ערך ההחזרה של הפונקציה ב $argument\ 0$ ונזיז את SP שיצביע אחרי ערך ההחזרה.

9 הרצאה 9:

10 הרצאה 10:

- **קומפיילר:** נבנה קומפיילר שיתרגם לנו משפת ג'ק ל VM, את הקומפיילר שלנו נבנה בשני חלקים, כאשר חלק אחד יתרגם ל XML וחלק אחר יתרגם ל VM קוד.

- $tokenizing$ - **אסימון:** אנו רוצים לקחת את התווים שיש לנו בקוד הג'ק ולתרגם אותם לאסימונים שיהיו בעלי משמעות שאותם נוכל לתרגם ל VM.\

- **אסימונים - טוקנס:** בשפת ג'ק יש לנו 5 סוגים של אסימונים:
  מילים שמורות: 20 מילים שמורות.
  סמלים: סוגריים וכו.
  מספרים: אינטג'רים 0-32767.
  סטרינגים: ""
  מזהים: משמשים לשמות משתנים ופונקציות "_" או שמות שלא מתחילים בספרה.

keyword: 'class'|'constructor'|'function'|
'method'|'field'|'static'|'var'|'int'|
'char'|'boolean'|'void'|'true'| 'false'|
'null'| 'this'| 'let'|'do'|'if'|'else'|
'while'|'return'

symbol: '{'|'}'|'('|')'|'['|']'|'.'|','|';'|'+'|'-'|'*'|
'/'|'&'|'|'|'<'|'>'|'='|'~'

integerConstant: a decimal number in the range 0 ... 32767

StringConstant: '"' a sequence of Unicode characters,
not including double quote or newline '"'

identifier: a sequence of letters, digits, and
underscore ('_') not starting with a digit.

- **תרגום סטרינג לטוקן:** כשנתרגם סטרינג נפריד אותו באופן הבא:

$$\langle token\ classification \rangle\ str\ \langle /token\ classification \rangle$$

- **דקדוק:** כללים שאומרים לנו באיזה סדר אנו יכולים לסדר את האסימונים בכדי שהשפה תהיה חוקית.
יש חוקים סופיים הכוללים קבועים, וחוקים אינסופיים שעובדים רקורסיבית.

- **ייצוג משפטים:** כאשר נתון לנו משפט או סטרינג במחשב ואנו רוצים לתרגם אותו ולרק אותו לאסימונים, אנו יכולים לעשות זאת בעזרת עץ, או בעזרת קובץ XML שהוא למעשה ייצוג של האסימונים בצורה היררכית.

- **דקדוק** $LL(k)$**:** שיטה זו אומרת מהו ה $k \in \mathbb{N}$ מספא האסימונים הברים עליהם אנו צריכים להסתכל בשביל לדעת באיזה סוג של משפט אנו נמצאים.
בתרגום משפת ג'ק $k = 1$, כלומר - מספיק להסתכל על הטוקן הראשון בשביל לדעת לאיזה מטודה לקרא ואיך לנתח את המשפט.

- $terms$**:** ביטויים שנצטרך לעבוד לפעמים קצת יותר קשה בכדי לתרגם אותם.
כאשר ביטוי מכיל קבוע, סטרינג או מילה שמורה נטפל בו כרגיל. אך כאשר הביטוי מכיל של של משתנה נצטרך להסתכל על הטוקן הבא $(LL(2))$ בכדי לדע אם שם המשתנה מכיל $[], (), .$ ובכל אחד מהמקרים נצטרך לטבל בנפרד.

- **כללי הדקדוק:**

Program structure:	A Jack program is a collection of classes, each appearing in a separate file. The compilation unit is a class. A class is a sequence of tokens structured according to the following context free syntax:
class:	**'class'** className '{' classVarDec* subroutineDec* '}'
classVarDec:	(**'static'** | **'field'**) type varName (',' varName)* ';'
type:	**'int'** | **'char'** | **'boolean'** | className
subroutineDec:	(**'constructor'** | **'function'** | **'method'**) (**'void'** | type) subroutineName '(' parameterList ')' subroutineBody
parameterList:	((type varName) (',' type varName)*)?
subroutineBody:	'{' varDec* statements '}'
varDec:	**'var'** type varName (',' varName)* ';'
className:	identifier
subroutineName:	identifier
varName:	identifier
Statements:	
statements:	statement*
statement:	letStatement | ifStatement | whileStatement | doStatement | returnStatement
letStatement:	**'let'** varName ('[' expression ']')? '=' expression ';'
ifStatement:	**'if'** '(' expression ')' '{' statements '}' (**'else'** '{' statements '}')?
whileStatement:	**'while'** '(' expression ')' '{' statements '}'
doStatement:	**'do'** subroutineCall ';'
ReturnStatement	**'return'** expression? ';'
Expressions:	
expression:	term (op term)*
term:	integerConstant | stringConstant | keywordConstant | varName | varName '[' expression ']' | subroutineCall | '(' expression ')' | unaryOp term
subroutineCall:	subroutineName '(' expressionList ')' | (className | varName) '.' subroutineName '(' expressionList ')'
expressionList:	(expression (',' expression)*)?
op:	'+' | '-' | '*' | '/' | '&' | '|' | '<' | '>' | '='
unaryOp:	'-' | '~'
KeywordConstant:	**'true'** | **'false'** | **'null'** | **'this'**

11 הרצאה 11:

- **שלבים בקומפילציה:**

1: תחילה נטפל ב $class - level - code$ זה החלק של הגדרת המחלקה ושדות המחלקה. בחלק זה נטפל בשדות ומשתנים סטטיים

2: $subrutine$ -מטודות המוגדרות בקלאס. בחלק זה נטפל במשתנים מקומיים וארגומנטים.

11.1 קמפול $subrutines$:

11.1.1 $procedural\ code$:

1. **משתנים:**

כאשר נרצה לתרגם משתנים ל VM קוד נתרגם אותם באופן הבא: תחילה נרצה לדעת האם הם שדות, משתנים גלובלים או מקומיים. עבור כל משתנה נרצה לשמור את $name, type, kind(static/local...), scope$. המידע הנחוץ על המשתנה.

בכדי לסווג כל משתנה נשתמש בטבלאות שישמרו לנו את המידע - כל טבלה תאתחל שם של משתנה עם $typr, kind$ בעמודה האחרונה של הטבלה נשמור מספר שמייצג את המיקום של המשתנה.

בכל פעם שנאתחל טבלה של מטודה נשמור משתנה כך:

$name$	$type$	$kind$	$\#$
$this$	$class\ name$	$argument$	0

שורה זו תייצג לנו את האובייקט הנוכחי עליו או עובדים, מתחת נשמור את שאר הארגומנטים והמשתנים של המטודה.

כדי לשמור על $scops$ כמו שצריך, נשמור את הטבלאות ב $linked\ list$ כך שהסקופ האחרון נשמר ראשון. כך כשנראה משתנה נתחיל לחפש אותו בסקופ הפנימי ביותר ונעלה למעלה בהיררכיה עד לסקופ של המחלקה.

2. **ביטויים** $expressions$**:**

כשניתקל בביטויים נטפל בהם באופן רקורסיבי:

מספר - n**:** נכתוב $push\ n$.

משתנה x**:** נכתוב $push\ x$.

ביטוי עם אופרטור בינארי: לדוגמה עבור $x + y$ נבצע באופן הבא:

$$push\ x$$
$$push\ y$$
$$output\ op$$

ביטוי עם אופרטור אונארי: לדוגמה עבור $\sim x$ נבצע באופן הבא:

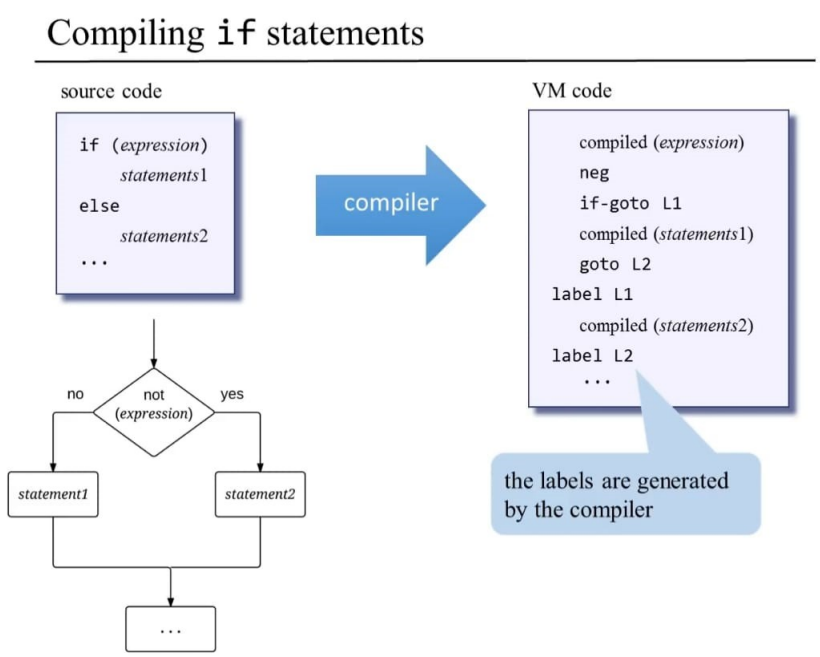
$$push\ x$$
$$output\ op$$

פונקציה: נבצע $push$ לארגומנטים של הפונקציה ואחכ נכתוב את הקריאה לפונקציה $call\ f$.

3. $flow\ of\ control$ **- לולאות:**

בחלק זה נטפל ב $let, do, return, while, if$.

כשנרצה לטבפל בלולאות $if, while$ נעשה זאת באופן הבא: נקח את התנאי של הלולאה ונפעיל עליו שלילה, אם קיבלנו $true$ - נלך ל $else$, אחרת - ניכנס ללולאה.



11.1.2 אובייקטים:

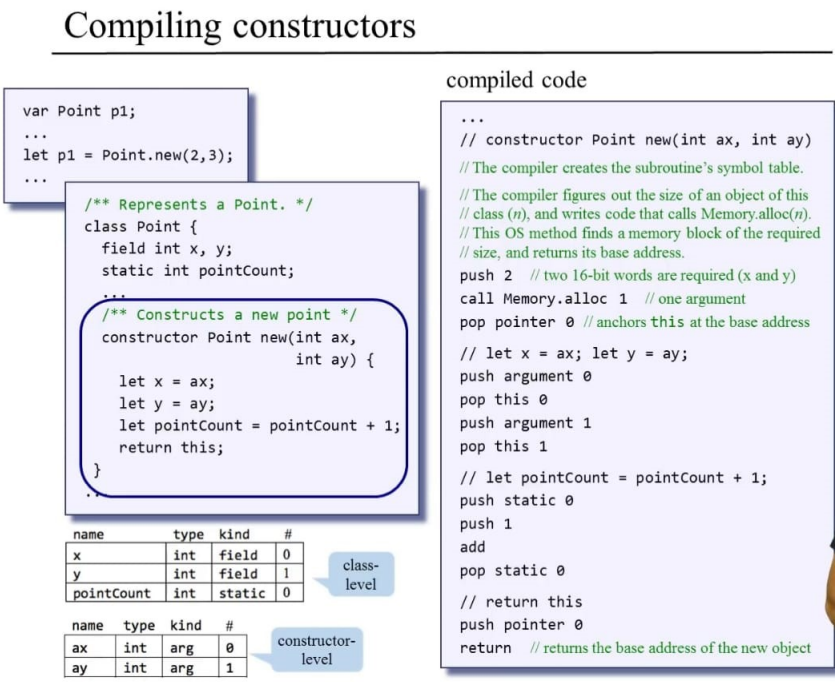
- **אחסון בזכרון:** למחשב יש שתי יחידות זכרון $stack, heap$. כאשר משתנים מקומיים וארגומנטים ישמרו ב $stack$, כאשר מערכים ואובייקטים ישמרו ב $heap$.

- $this, that$**:** מייצגים אובייקטים ומערכים, ממוקמים ב $heap$. הכתובות שלהם ישמרו ב $RAM3$ ו $RAM4$ בהתאמה, כאשר הפויינטרים יהיו 0 ו 1 בהתאמה.

- **אובייקטים ו** $this$**:** כאשר נצטרך לעבוד על אובייקט שנשמר ב $heap$, אנו נשמור מצביע לאובייקט ב $RAM3 = this$.

- **יצירת אובייקט חדש:** כשנרצה ליצור אובייקט חדש לדוגמה $P1 = Point.new(2,3)$ אנו נצטרך לשמור את שתי הנקודות - 2,3 ולעשות הם $push$, אחכ נקרא לקונסטרקטור כמו שאנו מטפלים במטודות, ולבסוף נבצע $pop\ P1$ כדי שהערך שחזר מהקונסטרקטור ישמר ב $P1$.

- **איך נקמפל קונסטרקטור:** הקונסטרקטור צריך לקבל גישה לשדות המחלקה בכדי לייצר את האובייקט, לכן נעשה זאת באמצעות $this$.
תחילה נבדוק כמה משתנים מוגדרים כמשתני מחלקה ונשמור להם מקום בעזרת $alloc$. ונחזיק מצביע לזכרון שהוקצה ב $pointer\ 0$.

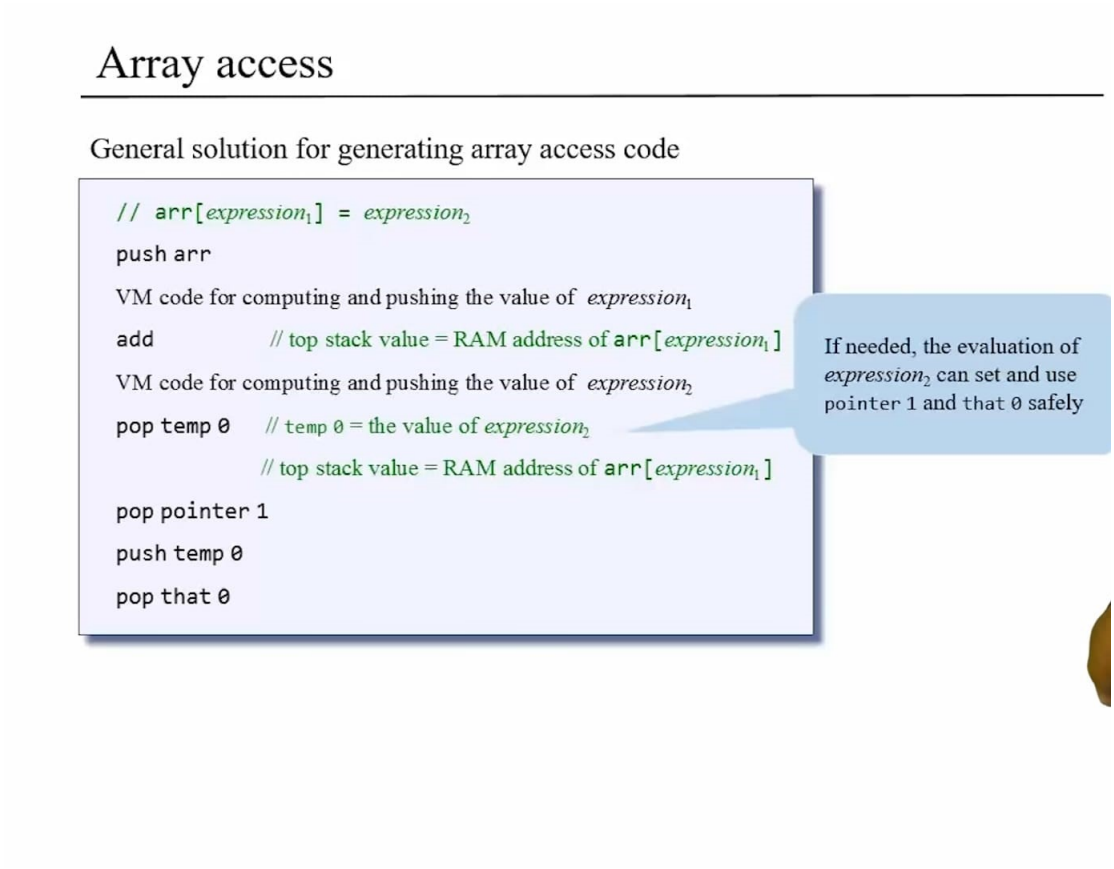


- **טיפול באובייקטים**: כשנצטרך להפעיל פונקציה על אובייקט, לדוגמה $obj.foo(x1, x2...)$, בתרגום - נעביר את האובייקט עצמו בתור אחד מהארגומנטים של הפונקציה (הראשון מבניהם).
לאחר מכן נשמור את שדות המחלקה ב $this$ כמו שהזכרנו מקודם בכדי שתהיה לנו גישה ונוכל לחשב מה שצריך.

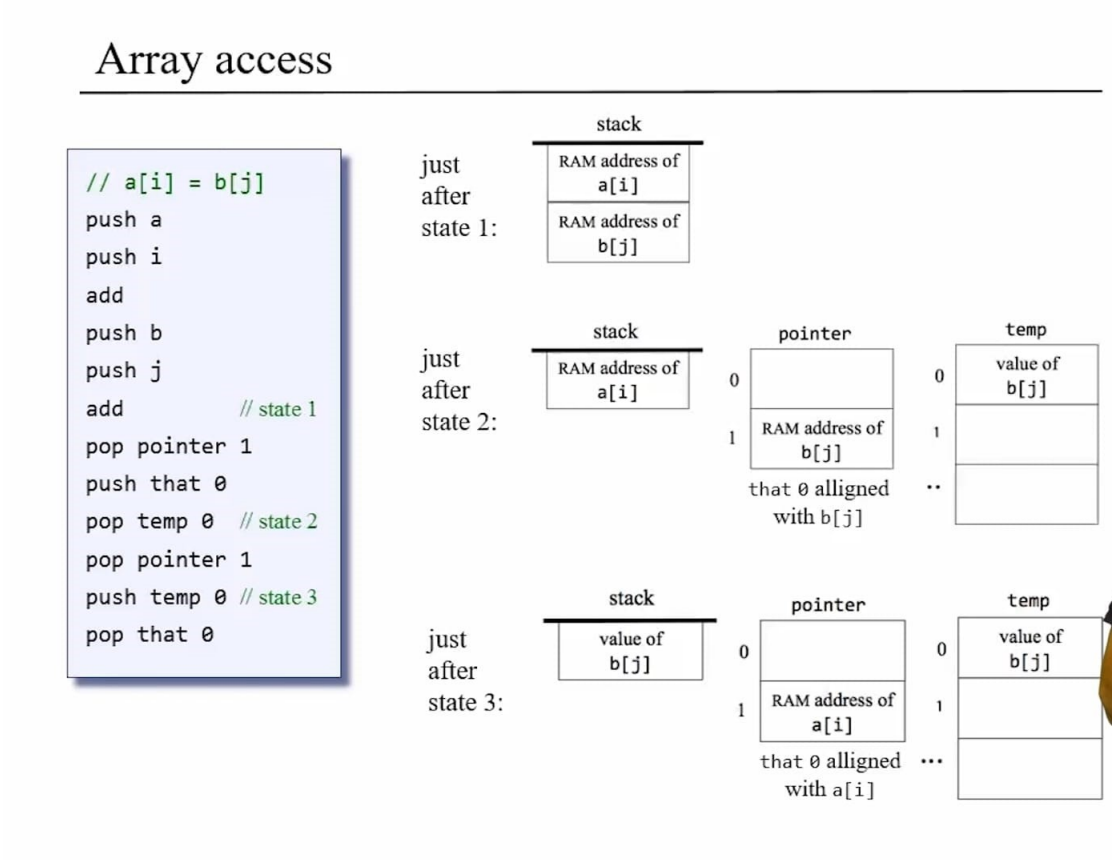
- **טיפול במטודות** $void$: מכיוון שמטודה תמיד צריכה להחזיר ערך, כאשר ניתקל במטודה $void$ - נבצע $push\ constant\ 0$ ואח"כ נעשה $return$.

11.1.3 מערכים:

- **מערכים ו** $that$: כאשר נצטרך לעבוד על מערך שנשמר ב $heap$, אנו נשמור מצביע לאיבר הראשון שלו (הכתובת של תחילת המערך) ב $RAM4 = that$.
- כשנרצה לאתחל מערך לדוגמה $let\ arr = Array.new(n)$ נטפל בשורה זו באותו האופן שטיפלנו באובייקטים.
- **כך נבצע השמה של ערך חדש למערך:** אלגוריתם כללי



- **כך נבצע השמה של ערך חדש למערך במקום ה** i - $a[i] = x$:



12 הרצאה 12 - מערכות הפעלה:

- **מערכת ההפעלה תנהל את הדברים הבאים:**

1: זכרון. **2:** עבודה עם קבצים. **3:** דרייברים - מדפסת, מסך, עכבר וכו. **4:** מולטי טסקינג. ועוד דברים נוספים.

- **יעילות** OS: כשאנו מתכננים מערכת הפעלה אנו צריכים להתחשב בזמני הריצה ולדאוג לכך שהיא תהיה יעילה, משום שזמן ריצה טוב למערכת ההפעלה ישליך על כך שזמן הריצה של כל התכניות יהיה טוב יותר ולהיפך.

- **ספריות:** ביחידה הנוכחית נממש פונקציות של ספריות שמערכת ההפעלה עובדת איתן, לדוגמה הספריה $math$ שבתוכה נמצאות הפונקציות $multiply, devide, sqrt$ ועוד. נצטרך לממש אותן בצורה יעילה.

12.1 המחלקה $Math$:

- **מימוש ל** $multiply$: נממש ככפל של שני מספרים בינארים.

1: נשים את שני המספרים אחד מתחת לשני

2: נמלא באפסים מצד שמאל.

3: נעשה שיפט לשמאל על המספר העליון סה"כ w פעמים. עבור w מספר הביטים של המספר השני.

4: נסכום את התוצאה.

פסואודו קוד:

```
// Returns x*y, where x, y ≥ 0
multiply(x, y):
  sum = 0
  shiftedX = x
  for  i = 0 ... w - 1   do
    if  ((i'th bit of y) == 1)
        sum = sum + shiftedX
    shiftedX = shiftedX * 2
  return sum
```

הערה: האלגוריתם מתמודד גם עם מספרים שליליים ואוברפלואו.

כדי לממש את השורה עם החץ ניצור מערך סטטי שישמור את המספר הנוכחי בתוכו ובכל פעם נבדוק האם האיבר ה i שווה ל 1 .

- **מימוש ל** $divide$**:** נממש כמו חילוק ארוך.

פסואודו קוד:

```
// Returns the integer part of x / y,
// where x ≥ 0 and y > 0
divide (x , y):
    if  (y > x) return 0
    q = divide (x, 2 * y)
    if ((x – 2 * q * y) < y)
        return 2 * q
    else
        return 2 * q + 1
```

חילוק מספרים שליליים: נעשה ערך מוחלט על המספרים, נחלק ולאחר מכן נסיף את הסימן.

אורפלואו: נוסיף את התנאי - $\text{if } (y > x) \text{ or } (y < 0) \text{ return } 0$, כך ברגע ש y ישנה סימן נדע שהגענו לאוברפלואו.

- **מימוש ל** $sqrt$

פסואודו קוד:

```
// Compute the integer part of  y = √x
// Strategy: find an integer y such that  y² ≤ x < (y+1)²  ( for  0 ≤ x < 2ⁿ)
// by performing a binary search in the range  0 ... 2^(n/2) – 1

sqrt (x):
   y = 0
   for  j = n/2 - 1 ... 0  do
      if  (y + 2^j)² ≤ x  then  y = y + 2^j
   return y
```

אוברפלואו: יכול להתרחש כאשר אנו מחשבים את $(y+2^j)^2 \leq x$. הפתרון הוא להחליף את התנאי באופן הבא: $(y+2^j)^2 \leq x \text{ and } (y+2^j)^2 > 0$.

12.2 ניהול זכרון - המחלקה $Memory$:

- **הפונקציה** $peek(address)$: מקבלת כתובת ומחזירה את הערך השמור בכתובת.
  הפונקציה $poke(address, val)$: מקבלת כתובת וערך, ומכניסה את הערך לכתובת.

כיצד נממש:
אנו צריכים לגשת ל RAM בעזרת השפה ג'ק.
נאתחל מערך סטטי שיקרא ram. ובמטודה $init$ נאתחל את המערך להיות 0. כך למעשה כל פעם שנכניס כתובת למערך יתבצע החישוב הבא $0 + addr$ וכך נגיע למקום הרצוי במערך.

- **הפונקציות** $alloc(size), deAlloc(obj)$: פונקציות שמקצות ומשחררות זכרון עבור אובייקטים ומערכים.

כיצד נממש:
$alloc(size)$: מקבלת אורך, מחפשת בזכרון מקום פנוי בגודל הרצוי ומחזירה לפונקציה שקראה לה מצביע למקום הנ"ל. כדי לנהל את הזכרון נשתמש בטכניקה שנקראת $Heap\ management$.

$Heap\ management$: ראשית נאתחל משתנה $\text{free} = \text{heapBase}$ שייצג את התאים הריקים ב $heap$.
כשהפונקציה $deAlloc$ **תיקרא:** נשתמש בלינקד ליסט כדי לעקוב אחר כל המקומות בזיכרון שזמינים לנו. כל רשימה תחזיק מצביע לתא ההראשון ואת הגודל המערך.
כך נאתחל את הרשימה המקושרת - $freeList$:

Implementing the heap / *freeList* (on the Hack platform):

```
class Memory {
   ...
   static Array heap;
   ...

   // In Memory.init:
   ...
   let heap = 2048; // heapBase
   let freeList = 2048;
   let heap[0] = 0;        // next
   let heap[1] = 14335;    // length
   ...
```

כשהפונקציה $alloc$ **תיקרא:** תחילה נחפש ברשימה המקושרת האם יש מקום פנוי שמתאים לגודל המערך אותו אנו רוצים לאתחל +2. (שני מקומות ל $size$ ומצביע), אם מצאנו - נכניס לרשימה. אחרת - הפונקציה תחזיר מצביע שיצביע ל $free$ ולאחר מכן נעדכן את $free$ שיצביע לסוף הזכרון שכרגע הקצנו.

- **פסאודו קוד:**

```
init:
   freeList = heapBase
   freeList.size = heapSize
   freeList.next = 0

// Allocate a memory block of size words
alloc(size):
   search freeList using best-fit or first-fit heuristics
         to obtain a segment with segment.size ≥ size + 2
   if no such segment is found, return failure
         (or attempt defragmentation)
   block = base address of the found space
   update the freeList and the fields of block
         to account for the allocation
   return block

// de-allocate the memory space of the given object
deAlloc(object):
   append object to the end of the freeList
```

12.3 המחלקה $Screen$:

- מחלקה זו תתעסק בגרפיקה ובציור על המסך.
- **יש שתי שיטות לשמירת תמונה:**

$vector$: נשמור את ההוראות כיצד לצייר את הציור. (שיטה זו עדיפה כי היא חוסכת במקום ומתאימה לכל רזולוציה ע"י התאמות קטנות).

$bitmap$: נשמור מפה עם הביטים שצריכים להיות צבועים בשחור - 1 . וביטים לבנים יסומנו ב 0 .

- **הפונקציה** $drawPixel$: נבצע חישוב איזה פיקסל צריך לצבוע, אח"כ ניגש לכתובת ונצבע את הביט הנדרש.

OS Screen class

```
...
// Sets pixel (x,y) to the current color
function void drawPixel(int x, int y) {
   address = 32 * y + x / 16
   value = Memory.peek[16384 + address]
   set the (x % 16)th bit of value to the current color
   do Memory.poke(address,value)
}
```

- **הפונקציה** $drawLine(x1, y1, x2, y2)$: כשאנו מציירים שורה, אנו יכולים לצבוע פיקסל רק מימין או מעל הפיקסל בו אנו נמצאים.

ראשית נגדיר:

```
drawLine(x1, y1, x2, y2)
let:
  x = x1;
  y = y1;
  dx = x2 - x1;
  dy = y2 - y1;
```

בנוסף נגדיר a - כמה פיקסלים זזנו ימינה, b- כמה פיקסלים עלינו למעלה $diff$ יגדיר לנו אם לזוז ימינה אן למעלה. נאתחל את כולם ל 0.

נגדיר גם $\text{let } diff = a^*dy - b^*dx$

פסואודו קוד:

```
a = 0; b = 0; diff = 0;
while ((a ≤ dx) and (b ≤ dy))
   drawPixel(x+a, y+b);
   // decide if to go right, or up;
   if (diff < 0) { a = a+1; diff = diff + dy;}
   else          { b = b+1; diff = diff - dx;}
```

צריך להוסיף לאלגוריתם התמודדות עם קווים לכל הכיוונים, בנוסף צריך להוסיף קו אנכי ואופקי בתור מקרים מיוחדים.

- **הפונקציה** $drawCircle(x, y, r)$**: נבצע זאת בעזרת הפונקציה** $drawLine$**.**

פסואודו קוד:

drawCircle (x, y, r)

for each $dy = -r$ to r do:

drawLine ($x - \sqrt{r^2 - dy^2}, y + dy$, $x + \sqrt{r^2 - dy^2}, y + dy$)

צריך לטפל באוברפלואו, בנוסף צריך לדאוג ש $r \leq 181$.

12.4 המחלקה $Output$:

- המחלקה אחראית על כתיבת טקסט על המסך.
- במחלקה זו נצטרך לדאוג ל $corsor$ - סמן שמסמן למשתמש איפה יוקלד התא הבא.

נממש כך:

Must be managed as follows:

- if asked to display `newLine`: move the cursor to the beginning of the next line
- if asked to display `backspace`: move the cursor one column left
- if asked to display any other character:
  display the character, and
  move the cursor one column to the right

12.5 המחלקה $Keyboard$:

- ה RAM שמייצג את המקלדת הוא 24576.

- **הפונקציה** $keyPressed$: פונקציה שבודקת האם נלחץ מקש כלשהו במקלדת.
  נשתמש במטודה $peek$ כדי לראות מהו התו שנמצא ברגיסטר שמייצג את המקלדת.

- **הפונקציה** $readChar$: פונקציה שצריכה לקרוא תווים שהמשתמש מכניס מהמקלדת.
  פסואודו קוד:

```
/** Waits until a key is pressed and released;
echoes the key on the screen, advances the cursor,
and returns the key's character value. */
readChar():
  display the cursor
  // waits until a key is pressed
  while (keyPressed() == 0):
        do nothing
  c = code of the currently pressed key
  // waits until the key is released
  while (keyPressed() ≠ 0):
        do nothing
  display c at the current cursor location
  advance the cursor
```

- **הפונקציה** $readLine$: פונקציה שצריכה לקרוא תווים שהמשתמש מכניס מהמקלדת עד שנלחץ התו אנטר.
  פסואודו קוד:

gets a string

```
/** Displays the message on the screen, gets the next
line (until a newLine character) from the keyboard,
and returns its value, as a string. */
readLine():
  str = empty string
  repeat
     c = readChar()
     if (c == newLine):
        display newLine
        return str
     else if (c = backSpace):
           remove the last character from str
           do Output.backspace()
        else
           str = str.append(c)
  return str
```

- **הפונקציה** $readInt$: אותו דבר כמו קריאת סטרינג, רק שצריך לקרוא ספרות בלבד.

12.6 המחלקה $String$:

- **אתחול:** נייצג כל סטרינג באמצעות מערך, משתנה נוסף ישמור את האורך העשווי של הסטרינג.

```
/** Represents a String object. Implements the String type. */
class String {
   field Array str;
   field int length;

   /** Constructs a new empty String with a maximum length. */
   constructor String new(int maxLength) {
      let str = Array.new(maxLength);
      let length = 0;
      return this;
   }

   ...
```

- **הפונקציה** $length$**:** מחזירה את האורך הנוכחי של הסטרינג.

- **הפונקציה** $setInt$**:** מחליפה מספר לסטרינגים ע"י הערך אסקי שלהם.

פסואודו קוד:

int to string:

```
// Returns the string representation of
// a non-negative integer
int2String(val):
    lastDigit = val % 10
    c = character representing lastDigit
    if (val < 10)
        return c (as a string)
    else
        return int2String(val / 10).append(c)
```

- **הפונקציה** $intValue$**:** מקבלת סטרינג ומחזירה את ערך האינטי שלו.

פסואודו קוד:

string to int:

```
// Returns the integer value of a string
// of digit characters, assuming that str[0]
// represents the most significant digit.
string2Int(str):
    val = 0
    for (i = 0 ... str.length) do
        d = integer value of str[i]
        val = val * 10 + d
    return val
```

- **שלשת הפונקציות האחרונות:** newLine, backSpace, doublequote: return $128, 129, 34$

12.7 המחלקה $Array$:

- מערכת ההפעלה צריכה לדאוג רק ליצירת ומחיקת מערך, בכל השאר מטפל הקומפילר.
- **הפונקציה** new: נממש אותה כפונקציה ולא כקונסטרקטור, נקרא ל $Memory.alloc$ ונחזיר את הכתובת.

12.8 המחלקה Sys:

- **הפונקציה** $init$: צריכה לאתחל את המערכת (לקרוא לכל מחלקה שיש בה מטודה $init$ לדוג' $do\ Math.init$), ואח"כ לקרוא ל $Main.main$.
- **הפונקציה** $halt$: מדמה מצב שהמחשב נעצר, ניתן לממש עם לולאה אינסופית.
- **הפונקציה** $wait(time)$: פונקציה שתחכה $time$ זמן ואחכ תפעל (במילי שניות). נממש בעזרת לולא עם דיליי פקטור שנממש בעזרת קבוע.
- **הפונקציה** $error$: פונקציה שמדפיסה שגיאה למסך.

13 הרצאה 13:

- ח