

מחברת תרגול למבחן בארכיטקטורה

ארכיטקטורה

קבוצת פקודות המכונה שמהוות את הממשק בין התוכנה והחומרה, ומתארת חומרה המממשת את הפקודות.
בקורס למדנו 2 ארכיטקטורות: Hack ו-Mips.

2	רכיב החישוב האריתמטי של המחשב ALU (הסבר)	H A C K
5	רכיב החישוב האריתמטי של המחשב ALU (תרגילי מבחן)	
10	אוגרים והתוכן שלהם ב-Hack (הסבר)	
12	אוגרים והתוכן שלהם ב-Hack (תרגילי מבחן)	
15	פיקסלים ב-Hack (הסבר)	
16	פיקסלים ב-Hack (תרגילי מבחן)	
18	בניית CPU (הסבר)	
20	בניית CPU (תרגילי מבחן)	
22	שאלות על Hack Assembly (הסבר)	
24	שאלות על Hack Assembly (תרגילי מבחן)	
26	שפת HDL (הסבר)	M I P S
44	שפת HDL (תרגילי מבחן)	
48	תכנות ב-Mips אסמבלי (הסבר)	
64	תכנות ב-Mips אסמבלי (תרגילי מבחן של אוגר עם Endian)	
67	תכנות ב-Mips אסמבלי (תרגילי מבחן)	
82	מטמון (הסבר)	
88	מטמון (תרגילי מבחן)	

מחברת תרגול למבחן בארכיטקטורה

נערך ונכתב על-ידי דור עזריה

2021

ספר זה לא נבדק על ידי מרצה, יתכן שימצאו טעויות.

לסיכומים נוספים שלי במדעי המחשב ומתמטיקה:

<https://dorazaria.github.io/>

מוזמנים לעקוב אחרי 😊:

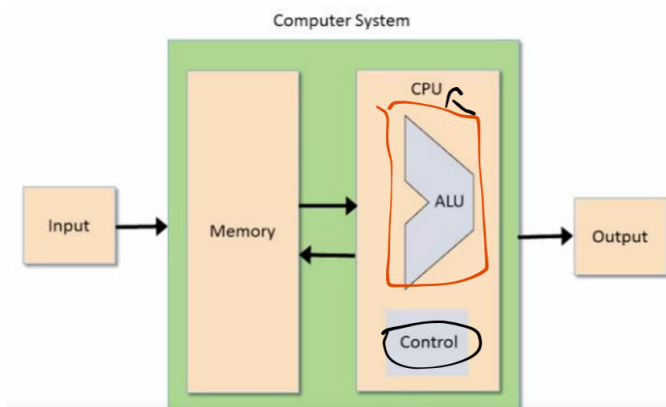
 LinkedIn: <https://www.linkedin.com/in/dor-azaria/>

 GitHub: <https://github.com/DorAzaria>

רכיב החישוב האריתמטי של המחשב ALU (הסבר)

ויקיפדיה

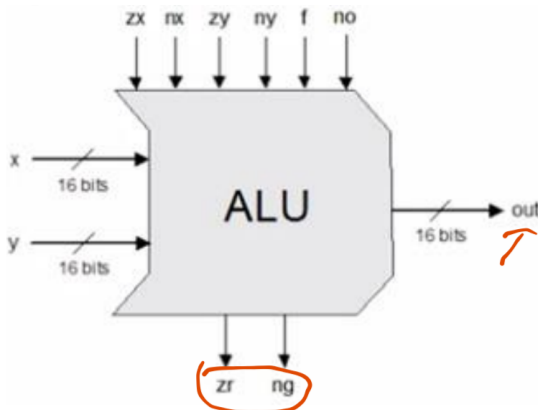
יחידה אריתמטית-לוגית (או בקיצור ALU)
 היא יחידה ביצועית חשובה הקיימת בכל יחידת עיבוד מרכזית הבנויה ממעגל אלקטרוני דיגיטלי. היחידה האריתמטית-לוגית מסוגלת לחשב את תוצאותיהן של מגוון רחב של פעולות אריתמטיות בסיסיות. כמעט כל היחידות האריתמטיות-לוגיות המודרניות משתמשות בשיטת המשלים ל-2 בייצוג מספר בינארי.



- ה ALU צריך לקבל מידע בגודל שני אוגרים (x, y) , לעשות עליהם איזשהו חישוב, ואח"כ להחזיר את הערך המחושב.
- כמו-כן, ה ALU צריך לקבל את zx, nx כדי שיגידו לו איזו פונקציה לחשב.

בניית ה-ALU

מקבל שני קלטים בני 16 ביטים (x, y) , ועוד 6 ביטים שאומרים מה לחשב, ומפיק פלט של 16 ביטים, ועוד שני ביטים שאומרים האם המספר חיובי או שלילי (או אפס).



zx	Zero X
nx	Not X
zy	Zero Y
ny	Not Y
f	F
no	Not Output
zr	Zero Result
ng	NeGative

למה ng, zr?

- ה- ng דלוק כאשר התוצאה של החישוב שלילית.
- ה- zr דלוק כאשר התוצאה של החישוב היא פשוט אפס.
- ה- ALU מפיק ביטים של שליטה, ng, zr, ששולטים על הקפיצות בקוד.
- אם יש קפיצה- נבדוק האם תנאי הקפיצה התקיים, ע"י שניעזר ביציאות ng, zr של ה- ALU.
- המשמעות של קפיצות קוד הן אותם קפיצות של ה- Program Counter.
- ה- PC תמיד יגיד מה הפקודה הבאה בתור, לכן בעזרת ה- ng ו- zr יהיה אפשר להצביע על הפקודה הבאה (הקפיצה בקוד).

מה אפשר לעשות עם השמות האלה?

zx	nx	zy	ny	f	no	out=
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x And y	if no then out=!out	f (X, y)=

(הטבלה הזאת תינתן במבחן כדף עזר).

הפונקציות החוקיות שנרצה לחשב:

$x+y, x-y, y-x, 0, 1, -1, x, y, -x, -y, !x, !y, x+1, y+1, x-1, y-1, x \& y, x|y$

שאלת תרגול

חשבו מה יוציא ה- ALU עבור הקלט הבא:

$zx = 0, nx = 1, zy = 0, ny = 1, f = 0, no = 1$

פתרון

מאחר ו- $zx = 0$ אז זה אומר שהוא לא דלוק, כנ"ל לגבי השאר, לכן נתעסק בכל שם ששווה ל-1 (כלומר דלוק).

- אפשר לראות כי $nx = 1$ כלומר יש כאן פעולת שלילה על x לכן נחשב: $x \rightarrow \bar{x}$.
- בנוסף $ny = 1$ ולכן גם כאן: $y \rightarrow \bar{y}$.
- מאחר ו- $f = 0$ חשוב לשים לב כי אם הוא לא דלוק אז צריך לבצע את ה- else כלומר: $out = \bar{x} \text{ And } \bar{y}$.
- גם ה- $no = 1$ דלוק ולכן נציב שלילה על ה- out שחישבנו: $out = (\bar{x} \text{ and } \bar{y})$.
- ולכן אם נפתח את השלילה נקבל ש- $out = x \text{ or } y$.

שאלת תרגול

חשבו מה יוציא ה-ALU עבור הקלט הבא:

$$zx = 1, nx = 1, zy = 0, ny = 1, f = 0, no = 1$$

פתרון

- אם ה- $zx=1$ אז $x=0$ כלומר $0 \rightarrow x$ כלומר הוא 16 אפסים.
- ה- $nx=1$ לכן נהפוך את כל הביטים כלומר $1 \rightarrow 0 \rightarrow x$ מכיוון ש-16 ביטים של אחדות במשלים ל-2 אז זה 1- (מינוס אחד) אז נקבל כי $1 = 1 \dots 1 = x$.
- ה- $ny=1$ ולכן $y \rightarrow \bar{y}$.
- ה- $f=0$ ולכן y and $1 \dots 1 = out$ ומאחר ואין שום משמעות ל-1.1 עם and אז נקבל $out = \bar{y}$.
- ה- $no=1$ זה אומר שה- out בשלילה ולכן $y = \bar{\bar{y}} = out$.
- ולסיום $out = y$.

רכיב החישוב האריתמטי של המחשב ALU (תרגילי מבחן)

zx	nx	zy	ny	f	no	out=
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x And y	if no then out=!out	f(x,y)=

תשע"ח (2018), סמסטר ב', מועד א', גרסה 2

מה תהיה הפונקציה $f(x, y)$ שיחשב ה-ALU של Hack כאשר:

$zx = 1, nx = 1, zy = 0, ny = 0, f = 1, no = 0$ (ראו דפי עזר).

א. $y-1$ ב. $x-1$ ג. $-x$ ד. $-y$

פתרון:

zx=1	$x = 0 \dots 0$
nx=1	$x \rightarrow \bar{x} \rightarrow 1..1 = -1$
zy=0	-
ny=0	-
f=1	$out = -1 + y = y - 1$
no=0	-

ולכן התשובה היא סעיף **א. $y-1$** .

תשע"ח (2018), סמסטר ב', מועד ב'

מה תהיה הפונקציה $f(x, y)$ שיחשב ה-ALU של Hack כאשר:

$zx = 0, nx = 1, zy = 0, ny = 1, f = 0, no = 1$ (ראו דפי עזר).

א. $y-x$ ב. $x-y$ ג. $x|y$ ד. $x \& y$

פתרון:

zx=0	-
nx=1	$x \rightarrow \bar{x}$
zy=0	-
ny=1	$y \rightarrow \bar{y}$
f=0	$out = \bar{x} \& \bar{y}$
no=1	$out = (\bar{x} \& \bar{y}) = x y$

ולכן התשובה היא סעיף **ג. $x|y$** .

zx	nx	zy	ny	f	no	out=
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x And y	if no then out=!out	f (X,y)=

2019, סמסטר ב, מועד א'

מה תהיה הפונקציה $f(x, y)$ שיחשב ה-ALU של Hack כאשר:

$zx = 0, nx = 1, zy = 0, ny = 0, f = 1, no = 1$ (ראו דפיו עזר).

א. $x-y$ ב. $y-x$ ג. $x-1$ ד. $x+1$

פתרון:

zx=0	-
nx=1	$x \rightarrow \bar{x}$
zy=0	-
ny=0	-
f=1	$out \rightarrow \bar{x} + y$
no=1	$out \rightarrow (\bar{x} + y) \rightarrow \bar{y} + x \Rightarrow out = x - y$

ולכן התשובה היא סעיף א. $x-y$.

תשע"ח (2018), סמסטר ב, מועד א', גרסה 1

מה תהיה הפונקציה $f(x, y)$ שיחשב ה-ALU של Hack כאשר:

$zx = 0, nx = 1, zy = 1, ny = 1, f = 1, no = 1$ (ראו דפיו עזר).

א. $x+1$ ב. $y-x$ ג. $x-1$ ד. y

פתרון:

zx=0	-
nx=1	$x \rightarrow \bar{x}$
zy=1	$y = 0..0$
ny=1	$y = 1..1 = -1$
f=1	$out \rightarrow \bar{x} - 1$
no=1	$out \rightarrow (\bar{x} - 1) \rightarrow x + 1 \Rightarrow out = x + 1$

ולכן התשובה היא סעיף א. $x+1$.

zx	nx	zy	ny	f	no	out=
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x And y	if no then out=!out	f (X,y)=

2019, סמסטר ב', מועד ב'

מה תהיה הפונקציה $f(x, y)$ שיחשב ה-ALU של Hack כאשר:

$zx = 0, nx = 0, zy = 0, ny = 1, f = 1, no = 1$ (ראו דפי עזר).

א. $y-x$ ב. $x-y$ ג. $x+1$ ד. $x-1$

פתרון:

zx=0	-
nx=0	-
zy=0	-
ny=1	$y \rightarrow \bar{y}$
f=1	$out \rightarrow x + \bar{y}$
no=1	$out \rightarrow (x + \bar{y}) \rightarrow \bar{x} - (-y) \rightarrow \bar{x} + y \rightarrow -x + y \Rightarrow out = y - x$

ולכן התשובה היא סעיף **א. $y-x$** .

2020, סמסטר ב', מועד א'

מה תהיה הפונקציה $f(x, y)$ שיחשב ה-ALU של Hack כאשר:

$zx = 1, nx = 1, zy = 0, ny = 0, f = 1, no = 0$ (ראו דפי עזר).

א. $y-1$ ב. $1-y$ ג. -1 ד. 1

פתרון:

zx=1	$x \rightarrow 0..0$
nx=1	$x \rightarrow 0..0 \rightarrow 1..1 = -1$
zy=0	-
ny=0	-
f=1	$out \rightarrow -1 + y \rightarrow y - 1$
no=0	-

ולכן התשובה היא סעיף **א. $y-1$** .

zx	nx	zy	ny	f	no	out=
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x And y	if no then out=!out	f(x,y)=

2020, סמסטר ב, מועד ב'

מה תהיה הפונקציה $f(x, y)$ שיחשב ה-ALU של Hack כאשר:

$zx = 1, nx = 1, zy = 1, ny = 1, f = 1, no = 0$ (ראו דפי עזר).

א. $1 - y$ ב. -1 ג. -2 ד. $x \text{ or } y$

פתרון:

zx=1	$x \rightarrow 0..0$
nx=1	$x \rightarrow 0..0 \rightarrow 1..1 = -1$
zy=1	$y \rightarrow 0..0$
ny=1	$y \rightarrow 0..0 \rightarrow 1..1 = -1$
f=1	$out = x + y = -1 + (-1) = -2$
no=0	-

ולכן התשובה היא סעיף ג. -2.

2021, סמסטר ב, מועד א'

מה תהיה הפונקציה $f(x, y)$ שיחשב ה-ALU של Hack כאשר:

$zx = 1, nx = 1, zy = 0, ny = 0, f = 1, no = 0$ (ראו דפי עזר).

א. $y - 1$ ב. $x - 1$ ג. $-x$ ד. $-y$

פתרון:

zx=1	$x \rightarrow 0..0$
nx=1	$x \rightarrow 0..0 \rightarrow 1..1 = -1$
zy=0	-
ny=0	-
f=1	$out = x + y = -1 + y = y - 1$
no=0	-

ולכן התשובה היא סעיף א. $y - 1$.

zx	nx	zy	ny	f	no	out=
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x And y	if no then out=!out	f(x,y)=

2021, סמסטר ב', מועד ב'

מה תהיה הפונקציה $f(x, y)$ שיחשב ה-ALU של Hack כאשר:

$zx = 0, nx = 1, zy = 0, ny = 1, f = 0, no = 1$ (ראו דפי עזר).

א. $x | y$.ב. $x - y$.ג. $x \& y$.ד. $\neg y$

פתרון:

zx=0	-
nx=1	$x \rightarrow \bar{x}$
zy=0	-
ny=1	$y \rightarrow \bar{y}$
f=0	$out = \bar{x} \& \bar{y}$
no=1	$out = (\bar{x} \& \bar{y}) = x y$

ולכן התשובה היא סעיף **א. $x | y$** .

אוגרים והתוכן שלהם ב-Hack (הסבר)

סוגי פקודות במחשב Hack

ישנם 2 סוגי פקודות (שניהם ברוחב 16 ביטים):

1. פקודות A (מלשון Address)
2. פקודות C (מלשון Compute)

אוגרים במחשב Hack

במחשב Hack יש 3 אוגרים בשימוש ב-CPU (גם הם ברוחב 16 ביטים):

1. אוגר D מלשון Data - משמש אותנו כאוגר זמני כמעט לכל דבר.
2. אוגר A מלשון Address - הערך המספרי של האוגר הזה אומר על איזה אוגר ב-RAM אנחנו מדברים.
3. אוגר M מלשון Memory - הוא לא אוגר ישיר, הוא מסמן את האוגר של ה-RAM שאנחנו מצביעים עליו כרגע (A מצביע עליו) כלומר $M = RAM[A]$.

פקודות A

- הדרך היחידה להכניס קבוע למערכת שהוא לא 1, 0, -1.
- לאחר השימוש בפקודה זו נוכל לגשת למקום בזיכרון עם הכתובת שבאוגר A, (או ל-RAM בעזרת M או ל-ROM בעזרת פקודת C) או להכניס את הקבוע לתוך אוגר D על מנת להשתמש בו בהמשך.
- הפנייה למיקום כלשהו -
 - לדוגמה:

@4

יגרום למחשב להסתכל על אוגר מספר 4.

לכן, לאחר הפקודה הזאת נקבל כי $A=4$.

וגם $RAM[4]$ הוא האוגר ש-M מתייחס אליו.

○ לדוגמה :

איך נשנה את הערך של אוגר 150 ל-5? (זה נקרא שפת Hack אסמבלי):

@5

D=A

@150

M=D

פקודות C

- מורכבת מבחירת יעד התוצאה, החישוב והאם לאחר ביצוע הפקודה נבצע קפיצה ומאיזה סוג (קפיצה למקום אחר בקוד ב-ROM המוצבע על-ידי הערך באוגר A).

עזרים

<i>dest</i>	d1	d2	d3	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
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

<i>jump</i>	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

אוגרים והתוכן שלהם ב-Hack (תרגילי מבחן)

תשע"ח (2018), סמסטר ב', מועד א', גרסה 2

מה יהיה תוכן אוגר D במחשב Hack לאחר ביצוע הפקודות:

```
@200
M = 1
@100
D = A
@200
D = M - D
@L
D; JLE
D = 1
```

(L)

א. 1 ב. -199 ג. -99 ד. -1

פתרון (מפורט לתרגיל ראשון כדי להבין):

שורה 1: זה אומר ש- $A = 200$ ו- M נמצא במקום ה-200.

שורה 2: זה אומר ש- RAM במקום ה-200 שווה ל-1 כלומר $M = RAM[200] = 1$.

שורה 3: עכשיו $A = 100$ ו- M נמצא במקום ה-100.

שורה 4: בגלל ש- $A = 100$ אז $D = 100$.

שורה 5: זה אומר ש- $A = 200$ ו- M נמצא במקום ה-200.

שורה 6: $D = 100 - 1 = 99$.

שורה 7: אין לנו מילה שמורה L במערכת ולכן כנראה שזה שם של שורה כלשהי שהמשתמש הגדיר.

שורה 8: חשבו את הערך של D שזה -99 וקפצו אם הערך קטן או שווה ל-0 כלומר $if -99 \leq 0 \text{ then jump}$.

שורה 9: לא הגענו אליה מכיוון שקפצנו לשורה L איפה שמסומן כ- (L).

לכן הפתרון הוא: **סעיף ג. -99.**

תשע"ח (2018), סמסטר ב', מועד א', גרסה 1

איזה רצף פקודות במחשב Hack יכניס 171 למקום 53 בזיכרון (זה שאלה אמריקאית):

פתרון:

```
@171
D=A
@53
M=D
```

הסבר: שמנו ב-A את 171 והצבנו אותו באוגר הזמני D, שמנו ב-A את 53 וכדי למקם את 171 ב-M פשוט הצבנו.

תשע"ח (2018), סמסטר ב', מועד א', גרסה 1

(אמריקאי) איזה רצף פקודות במחשב Hack יבצע:

if D>9 goto 507

פתרון:

@9

D = D - A

@507

D; JGT

מבחן לדוגמה 1

(אמריקאי) מה יהיה תוכן אוגר D במחשב Hack לאחר ביצוע הפקודות:

```
@100
D = A
@400
M = D
D = D + M
```

פתרון:

1. A=100
2. D=100
3. A=400
4. M = RAM[A]=RAM[400]= D = 100
5. D = 100 + 100 = 200

לכן באוגר D יהיה **200**.

2018 סמסטר ב', מועד ב'

(אמריקאי) מה יהיה תוכן אוגר D במחשב Hack לאחר ביצוע הפקודות (התמונה הממוסגרת):

1. A = 100
2. M=RAM[100]=1
3. D = 1
4. D = 5 - 1 = 4
5. Line L
6. if D > 0 then jump to L

```
@100
M=1
D=M
@5
D=A-D
@L
D; JGT
@3
D=A
(L)
```

לכן באוגר D יהיה **4**.

2021 סמסטר ב', מועד א'

(אמריקאי) מה יהיה תוכן אוגר D במחשב Hack לאחר ביצוע הפקודות (התמונה הממוסגרת):

1. A=200
2. M=RAM[A]=RAM[200]=1
3. D=1+1=2
4. A=100
5. D=100
6. A=200
7. $D = M - D = 1 - 100 = -99$
8. A = 50
9. A=L
10. *if* D + 1 ≤ 0 *then jump*: → - 98 ≤ 0 → *jump to L*
11. Not reached.

```

@200
M = 1
D=M+1
@100
D = A
@200
D = M - D
@50
@L
D+1;JLE
D = 1
(L)

```

לכן באוגר D יהיה **-98**.

2021 סמסטר ב', מועד ב'

(אמריקאי) מה יהיה תוכן אוגר D במחשב Hack לאחר ביצוע הפקודות (התמונה הממוסגרת):

1. A = 200
2. M = RAM[A] = RAM[200] = 1
3. A = 100
4. D = A = 100
5. A = 100 - 100 = 0
6. D = 100 - 0 = 100
7. A = 100 - 0 = 100
8. D = 100 - 100 = 0
9. A = 200
10. D = 1 - 0 = 1
11. L Line
12. *if* D + 1 ≤ 0 → *if* 1 + 1 ≤ 0 → *don't jump*
13. D = 1

```

@200
M = 1
@100
D = A
A=D-A
D=D-A
A=D-A
D=D-A
@200
D = M - D
@L
D+1;JLE
D = 1
(L)

```

לכן באוגר D יהיה **1**.

פיקסלים ב-Hack (הסבר)

- המסך מוגדר ברצף של 8192 אוגרים, החל מכתובת 16384 ועד כתובת 24575.
- התוכן של האוגרים הללו מגדיר את הפיקסלים שעל המסך - ביט 0 משמעו לבן, בין 1 משמעו שחור.
- כמובן, כל אוגר מוגדר להיות 16 ביטים, ולכן כדי להציג משהו על המסך נצטרך לעדכן את כל הביטים שברצוננו.
- כל 32 אוגרים הם שורה.
- בשפה יש כמה מילים שמורות:
 - SCREEN - האוגר הראשון של המסך - 16384.
 - KBD - האוגר של המקלדת - 24576.
 - R0, R1, R2, ..., R15 - כתובות 0,1,2,...,15 בהתאם.

נוסחה לשינוי פיקסל ספציפי על המסך

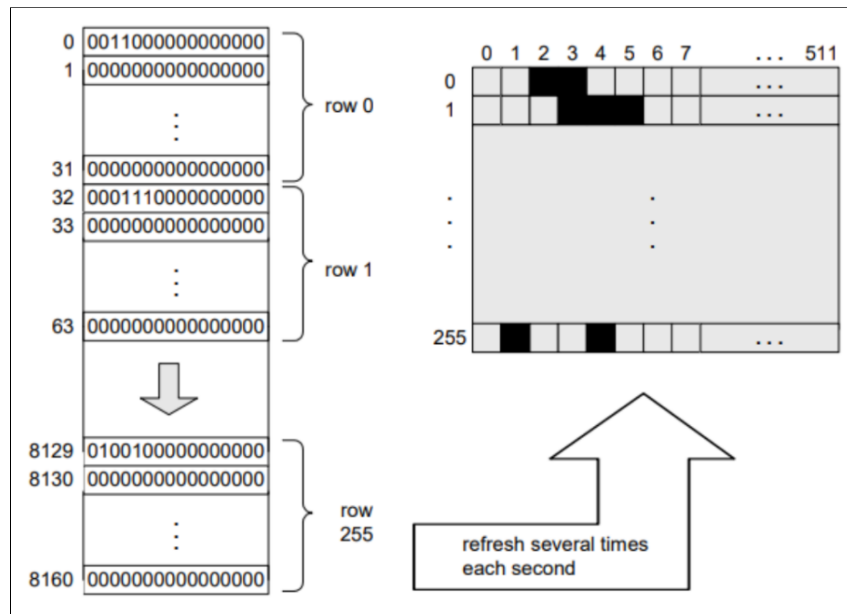
הנוסחה:

$$i = 32 \cdot rows + \frac{cols}{16}$$

כלומר, i היא המילה/האוגר בזיכרון של המסך במחשב Hack.

כדי לדעת איזה ביט באותה מילה צריך לשנות נחשב בנוסחה:

$$col \bmod 16$$



פיקסלים ב-Hack (תרגילי מבחן)

מבחן דוגמה 1

איזו מילה בזיכרון של המסך במחשב Hack צריך לשנות כדי להדליק ביט שנמצא בשורה 200 ובעמודה 100, איזה ביט באותה מילה צריך להדליק?
נניח שהזיכרון של המסך מתחיל בכתובת 0.

א. מילה 3216, ביט 4

ב. מילה 3216, ביט 8

ג. מילה 6406, ביט 8

ד. מילה 6406, ביט 4

פתרון:

לפי הנוסחה: $i = 32 \cdot rows + cols \setminus 16$

אז נקבל שהמילה היא: $6406 \rightarrow 6.25 + 6,400 = 100 \setminus 16 + 32 \cdot 200 = i$
והביט לפי הנוסחה: $cols \bmod 16 = 4$ נקבל כי $100 \bmod 16 = 4$ כלומר הביט הוא 4.
לכן התשובה היא ד.

2018, סמסטר ב, מועד א'

איזו מילה בזיכרון של המסך במחשב Hack צריך לשנות כדי להדליק ביט שנמצא בשורה 100 ובעמודה 200, נניח שהזיכרון של המסך מתחיל בכתובת 0.

פתרון:

לפי הנוסחה המילה היא $3212 \rightarrow 12.5 + 3,200 = 100 \setminus 16 + 32 \cdot 100$
המילה היא 3212.

הסבר:

כל 32 אוגרים הם שורה, ולכן נצטרך להתקדם 3,200 אוגרים בשביל להגיע לשורה 100.
כעת, בשביל עמודה 200 נבצע $12.5 = 200/16$ כלומר, צריך את האוגר ה-13 של השורה.
השורה מתחילה מאוגר 3200, ולכן האוגר ה-13 הוא ה-3212.

2020, סמסטר ב', מועד א'

איזה אוגר בזיכרון של המסך במחשב Hack צריך לשנות כדי להדליק ביט שנמצא בשורה 200 ובעמודה 50, ואיזה ביט צריך להדליק באוגר זה?
(הניחו כי זיכרון המסך מתחיל באוגר מספר 0).

פתרון:

לפי הנוחסא: $6403 = 50/16 + 200 \cdot 32$ לכן האוגר הוא 6403.
הביט שצריך להדליק באוגר זה הוא: $2 = 50 \bmod 16$ כלומר ביט מספר 2.

2020, סמסטר ב', מועד ב'

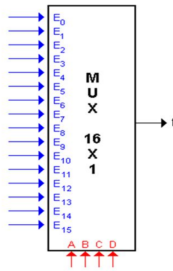
איזה אוגר בזיכרון של המסך במחשב Hack צריך לשנות כדי להדליק ביט שנמצא בשורה 50 ובעמודה 200, ואיזה ביט צריך להדליק באוגר זה?
(הניחו כי זיכרון המסך מתחיל באוגר מספר 0).

פתרון:

לפי הנוחסא: $12.5 + 1,600 = 200/16 + 50 \cdot 32$ לכן האוגר הוא 1612.
הביט שצריך להדליק באוגר זה הוא: $8 = 200 \bmod 16$ כלומר ביט מספר 8.

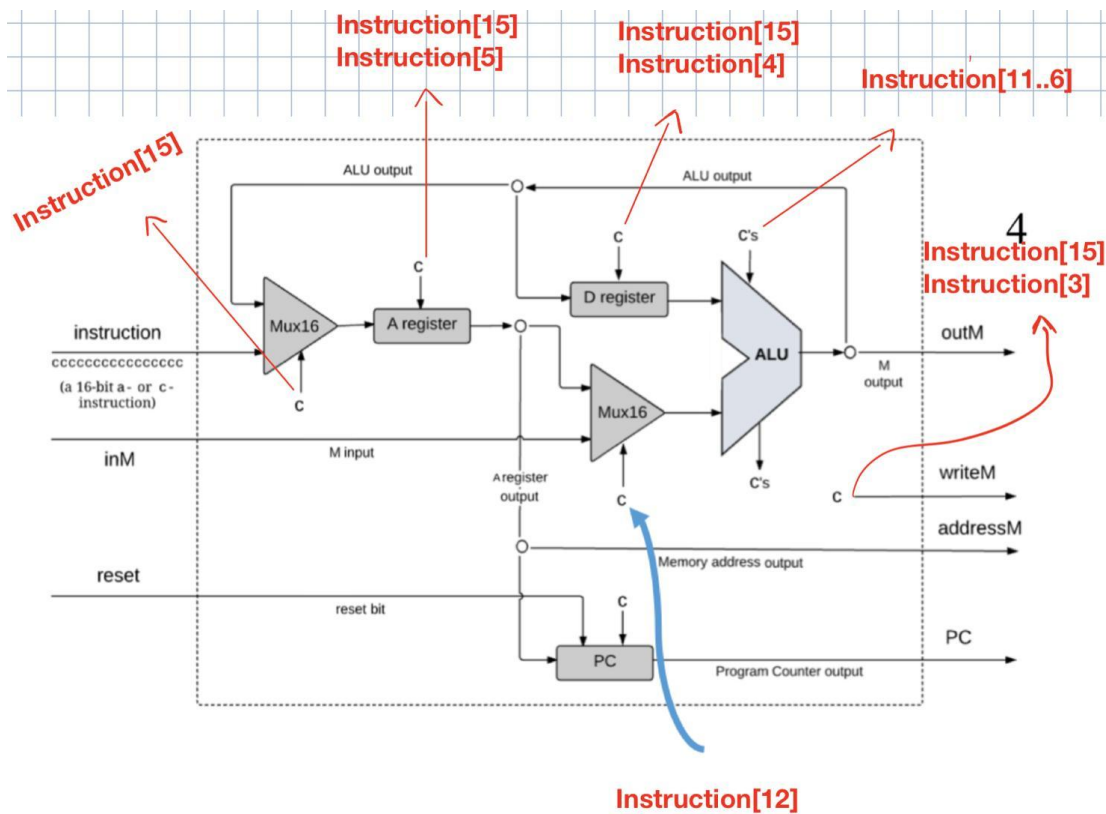
בניית CPU (הסבר)

תזכורות מקורסים קודמים



- PC - Program Counter
- Mux16 - נלמד בעבר בקורס מערכות ספרתיות (מרב):
הוא כינוי להתקן אלקטרוני המממש פעולה בסיסית וחשובה הנקראת ריבוב בה מתבצעת בחירה של אחד מכמה ערוצי קלט לערוץ פלט, בהתאם לערך בכניסות הבקרה.
- התקן המבצע את הפעולה ההפוכה למרוב הוא המפצל, **Demultiplexer**.
- ה-ALU הוא חישוב אריתמטי של המחשב.

ה-CPU נראה כך:



- כל מקום שרשום c זה ביט של בקרה שאומרת לכל רכיב האם להפעיל משהו, ומה להפעיל אם כן.

הכרת פקודות האסמבלי

כדי להתחיל להבין איך נבנה ה-CPU צריך להכיר את הפקודות של אסמבלי.

Symbolic syntax:

dest = comp ; jump

Binary syntax:

1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3

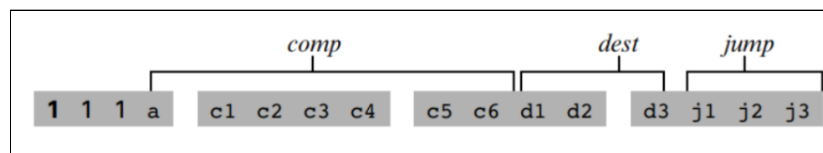
comp		c1	c2	c3	c4	c5	c6
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	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	d1	d2	d3	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
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

נסתכל על ה-Binary syntax וננתח:

- הביט הראשון הוא 1, כך אנו יודעים שזוהי פקודת C.
- שני הביטים הבאים לא מנוצלים (לא רלוונטי אלינו).
- לאחר מכן יש ביט שאומר האם מתייחסים לאוגר A או לאוגר M כאשר A=0 ו-M=1.
- לאחר מכן 6 הביטים הבאים זהים לביטי הבקרה של ה-ALU.
- לאחר מכן (באדום) יש את שלושת הביטים של dest - הם אומרים איזה אוגר/ים משנים בפקודה הנוכחית: אף אחד, אחד מהם, שניים מהם, או את שלושתם.
- לאחר מכן (בירוק) יש 3 ביטים jump - הם אומרים איזה קפיצה לבצע: אף קפיצה, קפיצה מותנית או קפיצה ללא תנאי.

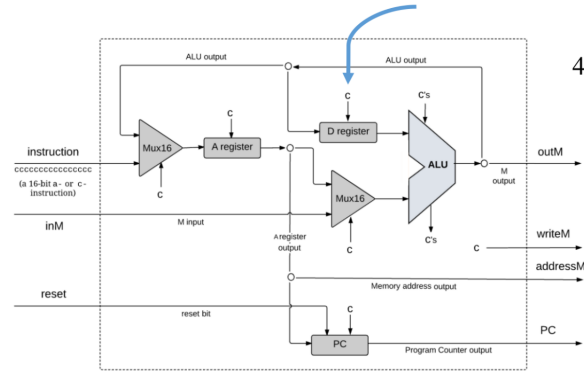


כל הביטים האלו ישמשו לנו לבקרה של ה-CPU, להגיד אילו אוגרים מקבלים ערכים חדשים, אילו אוגרים נשארים עם הערך הקודם, והאם לבצע קפיצה.

בניית CPU (תרגילי מבחן)

2018, סמסטר ב', מועד א'

(אמריקאי) לאיזה ביט בפקודה יחובר קו הבקרה המסומן בחץ בצירוף ה-CPU הבא:



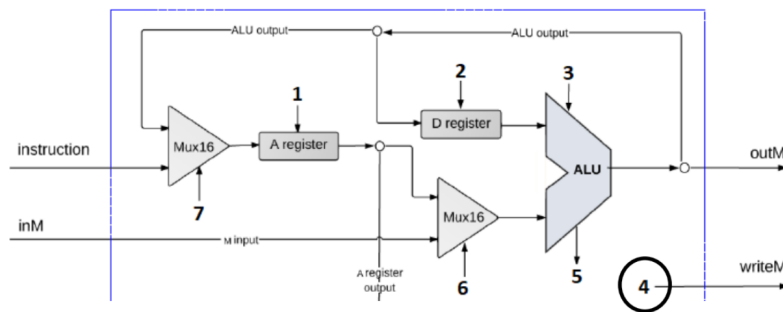
פתרון:

נסמן את Instruction[4] and Instruction[15].

בצער רב, תרגילים כאלה אין דרך חשיבה לפתרון, צריך פשוט לזכור איפה כל דבר מחובר (אני מצטט את המרצה) לפי הציוור שמוצג בתחילת הפרק פה (או המידה ויש לנו במבחן דף עזר של binary syntax ואז יהיה אפשר להבין).

2018, סמסטר ב', מועד ב'

(אמריקאי) לאילו ביט או ביטים באוגר הפקודה יחובר 4?



פתרון:

נסמן את Instruction[3] and Instruction[15].

2019, סמסטר ב', מועד א'

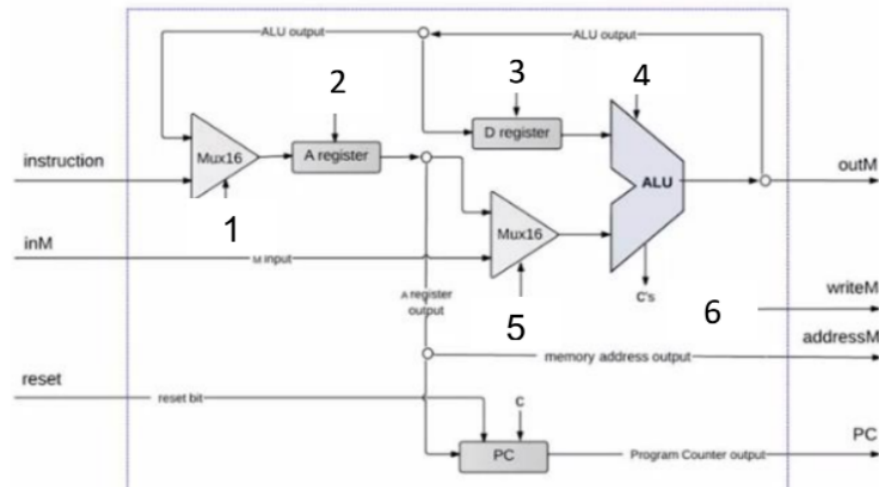
שאלה חוזרת

2019, סמסטר ב', מועד ב'

שאלה חוזרת

2020, סמסטר ב', מועד א'

(אמריקאי) לאילו ביט או ביטים באוגר הפקודה יחובר 5?

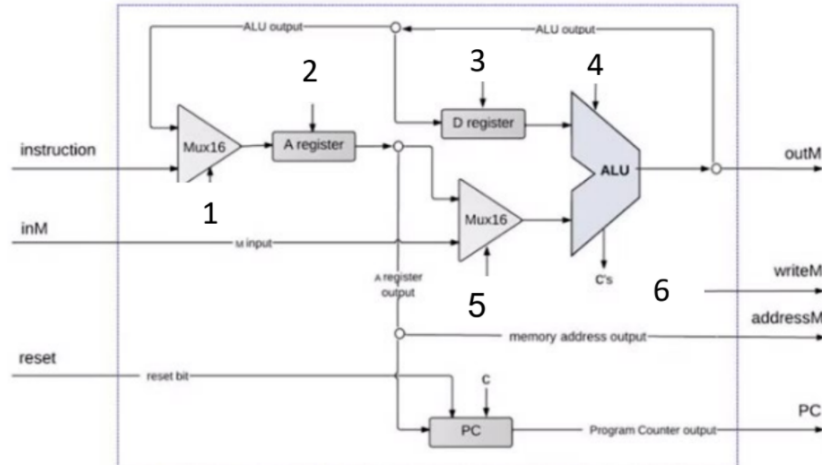


פתרון:

נסמן את Instruction[12].

2020, סמסטר ב', מועד ב'

(אמריקאי) לאילו ביט או ביטים באוגר הפקודה יחובר 2?



פתרון:

נסמן את Instruction[5] and Instruction[15].

2021, סמסטר ב', מועד א'

שאלה חוזרת

2021, סמסטר ב', מועד ב'

שאלה חוזרת

שאלות על Hack Assembly (הסבר)

ההסבר יהיה יחד עם תרגילי מבחן של שנת 2020.

2020, סמסטר ב', מועד א'

(אמריקאי) אילו פקודות יבצעו את הפסאודו קוד $arr[i] = 0$?

פתרון:

```
@arr
D=M
@i
A=D+M
M=0
```

הסבר לפי כל שורה:

1. צריך לגשת ל- arr , וצריך להתקדם ממנו עוד i תאים קדימה.
התא arr זה לא התא שבו מתחיל המערך, זה משתנה שנמצא בתחילת ה-RAM שבו רשום המקום שבו מתחיל המערך, לכן בשורה 2:
2. לכן צריך למשוך את התוכן של מה שכתוב ועכשיו אנחנו יודעים איפה מתחיל המערך arr .
כלומר, ב-D שמרנו את הכתובת הראשונה (התא הראשון) של המערך.
3. לגשת ל- i , הוא גם משתנה שנמצא בתחילת ה-RAM שבו כתוב מה הערך של i , לכן בשורה 4:
4. אנחנו צריכים להוסיף אותו למה שביצענו בשורה 2. כלומר להוסיף לתוכן שלו, למה שיש כרגע ב-D.
נשים ישירות כבר באוגר A, כדי שהאוגר יצביע על $arr[i]$.
- כלומר התא הראשון במערך (D) + המספר צעדים שצריך לזוז עד שנגיע ל- i הנכון (M).
5. צריך לרשום בתא הזה אפס לכן השוונו ל-0.
נזכור כי $M = RAM[A] = arr[i] = 0$.

2020, סמסטר ב', מועד ב'

(אמריקאי) אילו פקודות יבצעו את הפסאודו קוד $arr[i] = 2 \cdot arr[i]$ (הכפלה פי 2 של הערך)?

פתרון:

```
@arr
D=M
@i
A=D+M
D=M
M=D+M
```

הסבר לפי כל שורה:

1. צריך לגשת ל- arr , וצריך להתקדם ממנו עוד i תאים קדימה.
התא arr זה לא התא שבו מתחיל המערך, זה משתנה שנמצא בתחילת ה-RAM שבו רשום המקום שבו מתחיל המערך, לכן בשורה 2:
2. לכן צריך למשוך את התוכן של מה שכתוב ועכשיו אנחנו יודעים איפה מתחיל המערך arr .
כלומר, ב-D שמרנו את הכתובת הראשונה (התא הראשון) של המערך.
3. לגשת ל- i , הוא גם משתנה שנמצא בתחילת ה-RAM שבו כתוב מה הערך של i , לכן בשורה 4:
4. אנחנו צריכים להוסיף אותו למה שביצענו בשורה 2. כלומר להוסיף לתוכן שלו, למה שיש כרגע ב-D.
נשים ישירות כבר באוגר A, כדי שהאוגר יצביע על $arr[i]$.
5. כעת, יש לנו $arr[i]$. עכשיו צריך להכניס את הערך שיש בו פעמיים כלומר $M = M + M$; אבל הדבר הזה לא חוקי, לא קיימת פקודה שכזאת. לכן, נשמור את מה שכתוב בו ב-D כלומר נשמור את $arr[i]$.
6. ועכשיו נוכל להכניס את מה שכתוב לנו באוגר פעמיים, כלומר $D = arr[i]$ וגם $M = arr[i]$ וולכן
קיבלנו $arr[i] = arr[i] + arr[i] = 2 \cdot arr[i]$.

שאלות על Hack Assembly (תרגילי מבחן)

2019, סמסטר ב', מועד א'

(אמריקאי) אילו פקודות יבצעו: $y = x + 10$?

פתרון:

```
@x
D=M
@10
D=D+A
@y
M=D
```

הסבר:

ניגשים ל-x ומציבים אותו ב-D.
זה לא חוקי לבצע D+10 כלומר, נבצע @10 ואז נחבר עם A.
כדי להציב ב-y נשתמש באוגר M.

2019, סמסטר ב', מועד א'

(אמריקאי) אילו פקודות יבצעו: $?v[i] = 1$?

פתרון:

```
@v
D=M
@i
A=D+M
M=1
```

2019, סמסטר ב', מועד ב'

(אמריקאי) אילו פקודות יבצעו: $y = 10 + x$?

פתרון:

אותו הדבר כמו מועד א'.

2021, סמסטר ב', מועד א'

(אמריקאי) אילו פקודות יבצעו: $y = 10 + x$?

פתרון:

שאלה חוזרת ממועד 2019 ב'.

2021, סמסטר ב', מועד ב'

(אמריקאי) אילו פקודות יבצעו: $y = 10 + 2x$?

פתרון:

@x

D=M

D=D+M

@10

D=D+A

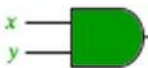
@y

M=D

שפת HDL (הסבר)

תזכורת ממערכות ספרתיות

אני אישית לא עברתי על השערים האלה יותר מדי, רק זכרתי את הגרפים של כל אחד (עוזר בHDL).

Name	Graphic symbol	Algebraic function	Truth table															
AND		$F = x \cdot y$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	F	0	0	0	0	1	0	1	0	0	1	1	1
x	y	F																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$F = x + y$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	F	0	0	0	0	1	1	1	0	1	1	1	1
x	y	F																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
Inverter		$F = x'$	<table><tr><th>x</th><th>F</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	x	F	0	1	1	0									
x	F																	
0	1																	
1	0																	
Buffer		$F = x$	<table><tr><th>x</th><th>F</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	x	F	0	0	1	1									
x	F																	
0	0																	
1	1																	
NAND		$F = (xy)'$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	F	0	0	1	0	1	1	1	0	1	1	1	0
x	y	F																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$F = (x + y)'$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	F	0	0	1	0	1	0	1	0	0	1	1	0
x	y	F																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	F	0	0	0	0	1	1	1	0	1	1	1	0
x	y	F																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= (x \oplus y)'$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	F	0	0	1	0	1	0	1	0	0	1	1	1
x	y	F																
0	0	1																
0	1	0																
1	0	0																
1	1	1																

שפת התכנות HDL

- שפה שמתעסקת ברכיבי מחשב ומממשת אותם.
- לכל רכיב יש את רגלי הכניסה שלו ורגלי היציאה שלו.
- אם הרכיב מורכב, ניתן שמות זמניים לחוטים שנמצאים באמצע, דוגמות בהמשך.
- כל יציאה ניתן לחבר למספר מקומות אחרים כרצוננו, ולכן חייבים לתת שמות ליציאות, כדי שנוכל לזהות אותן.
- יהיו לנו הרבה שיויונים מוזרים כמו $a=a$, $out=out$:
 - ה a השמאלי הוא הכניסה של השער, ו- a הימני הוא מה שאנחנו מחברים אליו.
 - ה out השמאלי הוא היציאה של השער, וה out הימני הוא היציאה של הרכיב.

דוגמה לקוד HDL עם הסברים - בקוד זה מממשים את And3Way.

```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/01/Xor.hdl
```

```
/**
 * And3Way:
 * out = 1 if and only if (a == 1 and b == 1 and c ==1)
 */
```

```
CHIP And3Way {
  IN a, b, c;
  OUT out;

  PARTS:
    And(a=a, b=b, out=ab);
    And(a=c, b=ab, out=out);
}
```

קרייט

הסבר על הרכיב

זה class כמו לכתוב חובה.

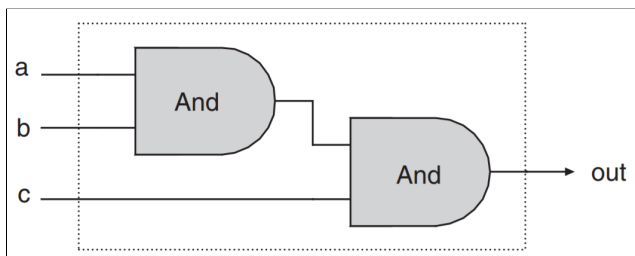
השם של הרכיב

כניסות הרכיב

יציאות הרכיב

ההגדרה של כאן כותבים את המימוש של הרכיב, כל מה שמעל החלק הזה נכתב מראש אנחנו לא מתעסקים בכל מה שמלמעלה

המימוש עצמו



לגבי And3Way - אנחנו בעצם צריכים באופן לוגי לחשב $a \& b \& c$.

רכיב And רגיל מקבל 2 כניסות ומוציא את התשובה כלומר נכנסים $a \& b$ ויוצא הפלט out .

ולכן מספיק להוסיף עוד And אחד שיקיים $c \& (a \& b)$ ומכאן השתמשנו ב- 2 And-ים במימוש.

מימוש דוגמה של Xor.hdl

```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/01/Xor.hdl
```

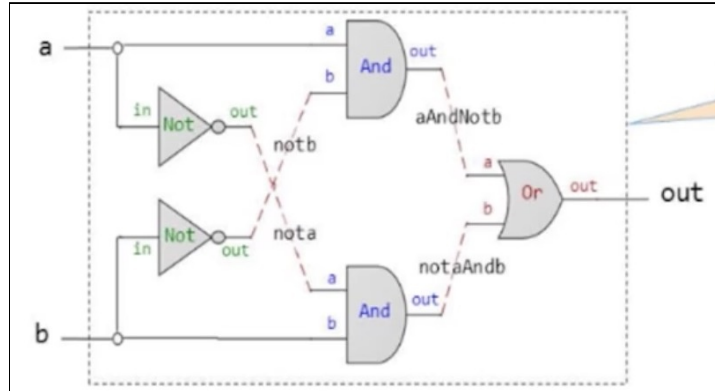
```
/** *
 * Exclusive-or gate: * out = not (a == b)
 */
```

```
CHIP Xor {
  IN a, b;
  OUT out;
```

PARTS:

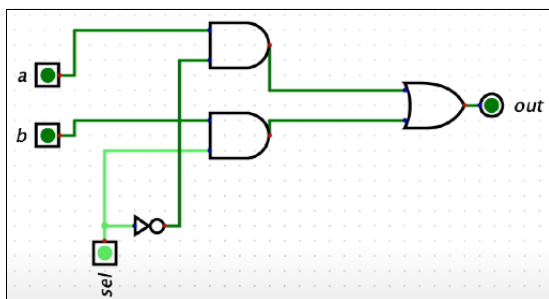
```
Not(in=a, out=nota);
Not(in=b, out=notb);
And(a=a, b=notb, out=aAndNotb);
And(a=nota, b=b, out=notaAndb);
Or(a=aAndNotb, b=notaAndb, out=out);
```

```
}
```



• **Mux** - נלמד במערכות ספרתיות (מרבב):

הוא כינוי להתקן אלקטרוני המממש פעולה בסיסית וחשובה הנקראת ריבוב בה מתבצעת בחירה של אחד מכמה ערוצי קלט לערוץ פלט, בהתאם לערך בכניסות הבקרה. התקן המבצע את הפעולה ההפוכה למרבב הוא מפלג, Demultiplexer.



```
CHIP Mux {
  IN a, b, sel;
  OUT out;
```

PARTS:

```
Not(in=sel, out=nsel);
And(a=sel, b=b, out=c1);
And(a=nsel, b=a, out=c2);
Or(a=c1, b=c2, out=out);
```

```
}
```

בשביל לענות על שאלות של זה במבחן צריך להכיר את המימושים הבאים לצערי הרב:

Not

CHIP **Not** {

IN **in**;

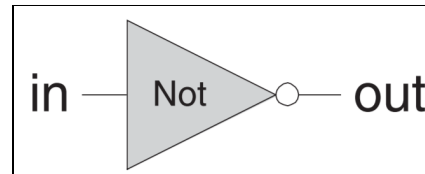
OUT **out**;

PARTS:

// Put your code here:

Nand(a=**in**, b=**in**, out=**out**);

}



Not16

CHIP **Not16** {

IN **in**[16];

OUT **out**[16];

PARTS:

// Put your code here:

Not(in = **in**[0], out=**out**[0]);

Not(in = **in**[1], out=**out**[1]);

Not(in = **in**[2], out=**out**[2]);

Not(in = **in**[3], out=**out**[3]);

Not(in = **in**[4], out=**out**[4]);

Not(in = **in**[5], out=**out**[5]);

Not(in = **in**[6], out=**out**[6]);

Not(in = **in**[7], out=**out**[7]);

Not(in = **in**[8], out=**out**[8]);

Not(in = **in**[9], out=**out**[9]);

Not(in = **in**[10],out=**out**[10]);

Not(in = **in**[11],out=**out**[11]);

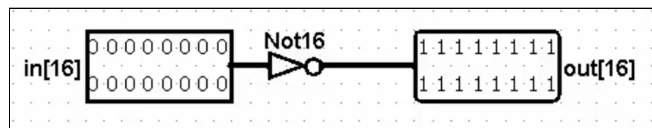
Not(in = **in**[12],out=**out**[12]);

Not(in = **in**[13],out=**out**[13]);

Not(in = **in**[14],out=**out**[14]);

Not(in = **in**[15],out=**out**[15]);

}



And

CHIP **And** {

IN **a**, **b**;

OUT **out**;

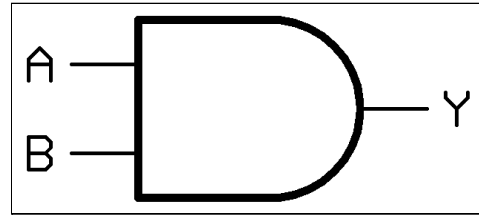
PARTS:

// Put your code here:

Nand(a = **a**, b = **b** , out = **outNand**);

Not(in = **outNand**, out = **out**);

}



And16

CHIP **And16** {

IN a[16], b[16];

OUT out[16];

PARTS:

// Put your code here:

And (a = a[0], b = b[0] , out = out[0]);

And (a = a[1], b = b[1] , out = out[1]);

And (a = a[2], b = b[2] , out = out[2]);

And (a = a[3], b = b[3] , out = out[3]);

And (a = a[4], b = b[4] , out = out[4]);

And (a = a[5], b = b[5] , out = out[5]);

And (a = a[6], b = b[6] , out = out[6]);

And (a = a[7], b = b[7] , out = out[7]);

And (a = a[8], b = b[8] , out = out[8]);

And (a = a[9], b = b[9] , out = out[9]);

And (a = a[10], b = b[10] , out = out[10]);

And (a = a[11], b = b[11] , out = out[11]);

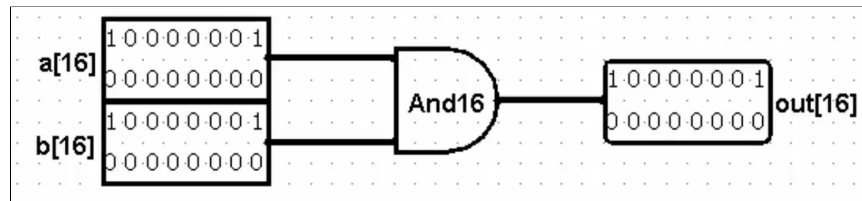
And (a = a[12], b = b[12] , out = out[12]);

And (a = a[13], b = b[13] , out = out[13]);

And (a = a[14], b = b[14] , out = out[14]);

And (a = a[15], b = b[15] , out = out[15]);

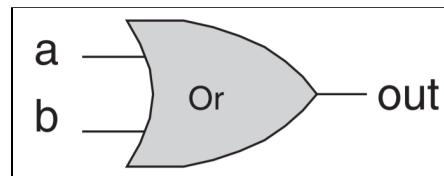
}



Or

```
CHIP Or {
  IN a, b;
  OUT out;

  PARTS:
    Not(in=a, out=notA);
    Not(in=b, out=notB);
    Nand(a=notA, b=notB, out=out);
}
```

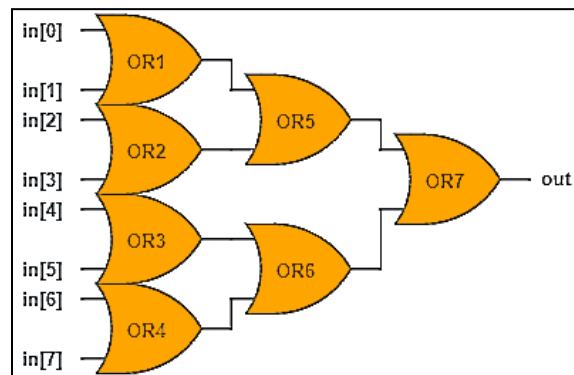


• עבור Or16 באותו השיטה כמו כל ה-16 שמוצג כאן.

Or8Way

```
CHIP Or8Way {
  IN in[8];
  OUT out;

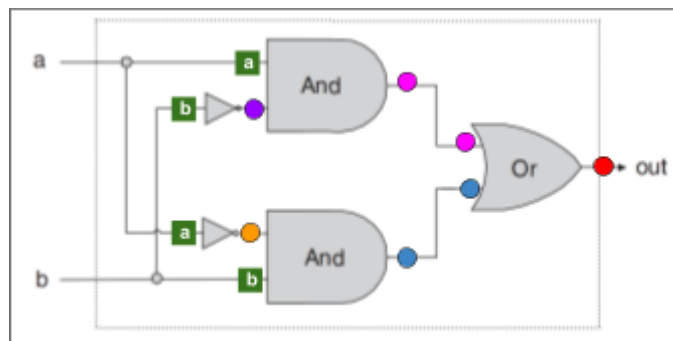
  PARTS:
    Or(a=in[0], b=in[1], out=out1);
    Or(a=out1, b=in[2], out=out2);
    Or(a=out2, b=in[3], out=out3);
    Or(a=out3, b=in[4], out=out4);
    Or(a=out4, b=in[5], out=out5);
    Or(a=out5, b=in[6], out=out6);
    Or(a=out6, b=in[7], out=out);
}
```



Xor

```
CHIP Xor {
  IN a, b;
  OUT out;

  PARTS:
    Not(in = a , out = Nota);
    Not(in = b , out = Notb);
    And(a = a , b = Notb, out = aNotb);
    And(a = Nota, b = b , out = bNota);
    Or(a = aNotb, b = bNota, out = out);
}
```



ריבוב: בחירה של אחד מ-2 ערוצי קלט לערוץ פלט - Mux

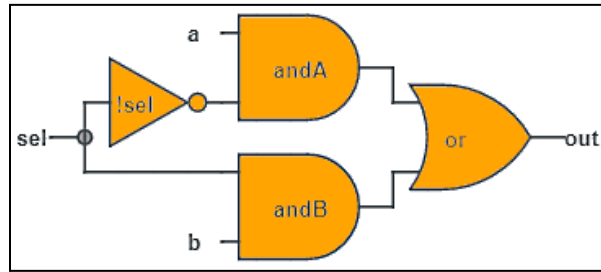
CHIP **Mux** {
 IN **a**, **b**, **sel**;
 OUT **out**;

PARTS:

// Put your code here:

```
Not(in = sel, out = notSel);
And(a = a, b = notSel, out = aNotSel);
And(a = sel, b = b, out = bSel);
Or(a = aNotSel, b = bSel, out = out);
```

}



ריבוב מרובה: בחירה של אחד מ-2 ערוצי קלט ב-16 ביטים לערוץ פלט של 16 ביטים - Mux16

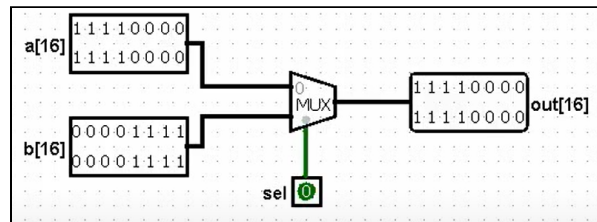
CHIP **Mux16** {
 IN **a**[16], **b**[16], **sel**;
 OUT **out**[16];

PARTS:

// Put your code here:

```
Mux(a = a[0], b = b[0], sel = sel, out = out[0]);
Mux(a = a[1], b = b[1], sel = sel, out = out[1]);
Mux(a = a[2], b = b[2], sel = sel, out = out[2]);
Mux(a = a[3], b = b[3], sel = sel, out = out[3]);
Mux(a = a[4], b = b[4], sel = sel, out = out[4]);
Mux(a = a[5], b = b[5], sel = sel, out = out[5]);
Mux(a = a[6], b = b[6], sel = sel, out = out[6]);
Mux(a = a[7], b = b[7], sel = sel, out = out[7]);
Mux(a = a[8], b = b[8], sel = sel, out = out[8]);
Mux(a = a[9], b = b[9], sel = sel, out = out[9]);
Mux(a = a[10], b = b[10], sel = sel, out = out[10]);
Mux(a = a[11], b = b[11], sel = sel, out = out[11]);
Mux(a = a[12], b = b[12], sel = sel, out = out[12]);
Mux(a = a[13], b = b[13], sel = sel, out = out[13]);
Mux(a = a[14], b = b[14], sel = sel, out = out[14]);
Mux(a = a[15], b = b[15], sel = sel, out = out[15]);
```

}



ריבוב מרובה 4: בחירה של אחד מ-4 ערוצי קלט של 16 ביט לערוץ פלט של 16 ביט - Mux4Way16

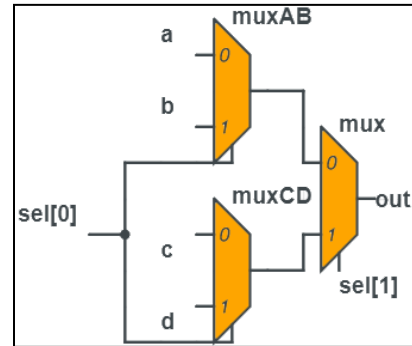
```
CHIP Mux4Way16 {
  IN sel[2], a[16], b[16], c[16], d[16];
  OUT out[16];
```

PARTS:

// Put your code here:

```
Mux16(a = a, b = b, sel = sel[0], out = muxAB);
Mux16(a = c, b = d, sel = sel[0], out = muxCD);
Mux16(a = muxAB, b = muxCD, sel = sel[1], out = out);
```

```
}
```



ריבוב מרובה 8: בחירה של אחד מ-8 ערוצי קלט של 16 ביט לערוץ פלט של 16 ביט - Mux8Way16

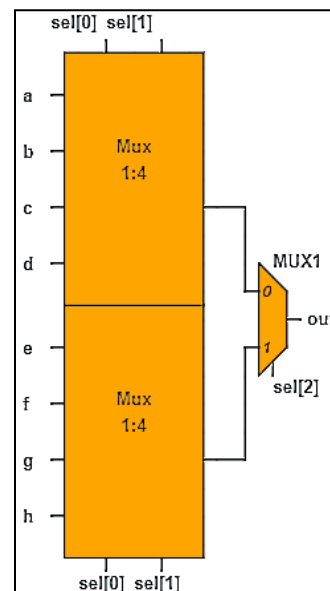
```
CHIP Mux8Way16 {
  IN sel[3], a[16], b[16], c[16], d[16], e[16], f[16], g[16], h[16];
  OUT out[16];
```

PARTS:

// Put your code here:

```
Mux4Way16(a = a, b = b, c = c, d = d, sel[0]=sel[0], sel[1]=sel[1], out = ABCD);
Mux4Way16(a = e, b = f, c = g, d = h, sel[0]=sel[0], sel[1]=sel[1], out = EFGH);
Mux16(a = ABCD, b = EFGH, sel = sel[2], out = out);
```

```
}
```



המפלג: שימושי כאשר דרוש להעביר נתונים ממקור אחד - ל-2 יעדים - DMux

CHIP **DMux** {

IN **in**, **sel**;

OUT **a**, **b**;

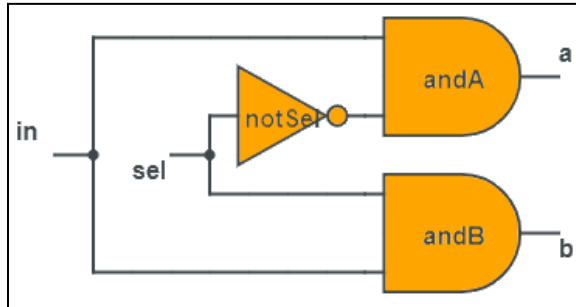
PARTS:

Not (in = **sel**, out = **NotSel**);

And (a = **in**, b = **NotSel**, out = **a**);

And (a = **in**, b = **sel**, out = **b**);

}



מפלג על 4 : שימושי כאשר דרוש להעביר נתונים ממקור אחד - ל-4 יעדים - DMux4Way

CHIP **DMux4Way** {

IN **in**, **sel**[2];

OUT **a**, **b**, **c**, **d**;

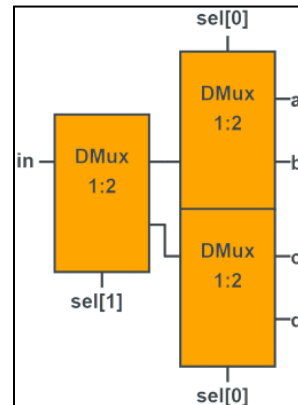
PARTS:

DMux(in=**in**, sel=**sel**[1], a=**abOut**, b=**cdOut**);

DMux(in=**abOut**, sel=**sel**[0], a=**a**, b=**b**);

DMux(in=**cdOut**, sel=**sel**[0], a=**c**, b=**d**);

}



מפלג על 8 : שימושי כאשר דרוש להעביר נתונים ממקור אחד - ל-8 יעדים - DMux8Way

CHIP **DMux8Way** {

IN **in**, **sel**[3];

OUT **a**, **b**, **c**, **d**, **e**, **f**, **g**, **h**;

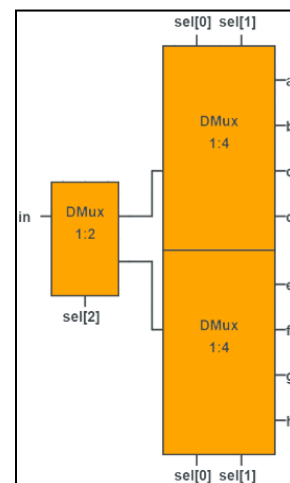
PARTS:

DMux(in = **in**, sel = **sel**[2], a = **abcdOut**, b = **efghOut**);

DMux4Way(in = **abcdOut**, sel[0] = **sel**[0], sel[1] = **sel**[1],
a = **a**, b = **b**, c = **c**, d = **d**);

DMux4Way(in = **efghOut**, sel[0] = **sel**[0], sel[1] = **sel**[1],
a = **e**, b = **f**, c = **g**, d = **h**);

}



חצי מחבר: חיבור שני ביטים - HalfAdder

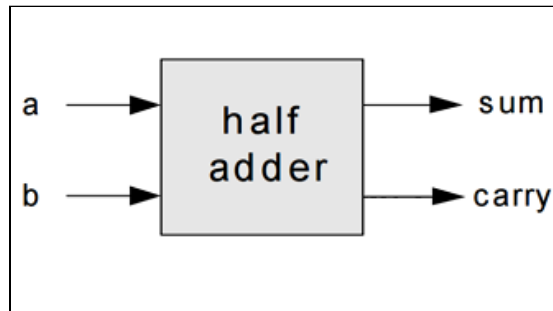
CHIP **HalfAdder** {IN **a**, **b**;OUT **sum**, **carry**;

PARTS:

// Put you code here:

And(a = **a** , b = **b** , out = **carry**);**Xor**(a = **a** , b = **b** , out = **sum**);

}



מחבר מלא: חיבור שלושה ביטים - FullAdder

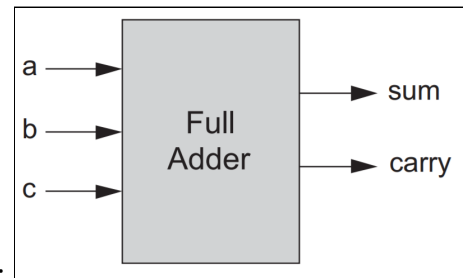
CHIP **FullAdder** {IN **a**, **b**, **c**;OUT **sum**, **carry**;

PARTS:

// Put you code here:

HalfAdder(a = **a** , b = **b** , sum = **s** , carry = **cOut1**);**HalfAdder**(a = **c** , b = **s** , sum = **sum** , carry = **cOut2**);**Or**(a = **cOut1** , b = **cOut2** , out = **carry**);

}



מחבר בין 2 מספרי 16 ביט - Add16

CHIP **Add16** {

IN a[16], b[16];

OUT out[16];

PARTS:

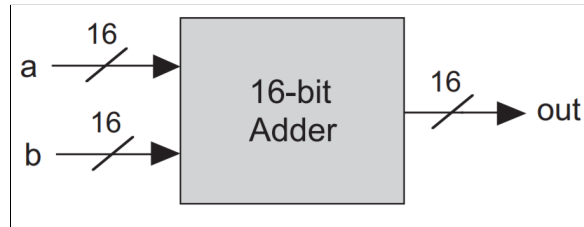
// Put you code here:

```

HalfAdder(a = a[0] , b = b[0] , sum = out[0] , carry = C0);
FullAdder(a = a[1] , b = b[1] , c = C0 , sum = out[1] , carry = C1);
FullAdder(a = a[2] , b = b[2] , c = C1 , sum = out[2] , carry = C2);
FullAdder(a = a[3] , b = b[3] , c = C2 , sum = out[3] , carry = C3);
FullAdder(a = a[4] , b = b[4] , c = C3 , sum = out[4] , carry = C4);
FullAdder(a = a[5] , b = b[5] , c = C4 , sum = out[5] , carry = C5);
FullAdder(a = a[6] , b = b[6] , c = C5 , sum = out[6] , carry = C6);
FullAdder(a = a[7] , b = b[7] , c = C6 , sum = out[7] , carry = C7);
FullAdder(a = a[8] , b = b[8] , c = C7 , sum = out[8] , carry = C8);
FullAdder(a = a[9] , b = b[9] , c = C8 , sum = out[9] , carry = C9);
FullAdder(a = a[10] , b = b[10] , c = C9 , sum = out[10] , carry = C10);
FullAdder(a = a[11] , b = b[11] , c = C10 , sum = out[11] , carry = C11);
FullAdder(a = a[12] , b = b[12] , c = C11 , sum = out[12] , carry = C12);
FullAdder(a = a[13] , b = b[13] , c = C12 , sum = out[13] , carry = C13);
FullAdder(a = a[14] , b = b[14] , c = C13 , sum = out[14] , carry = C14);
FullAdder(a = a[15] , b = b[15] , c = C14 , sum = out[15] , carry = C15);

```

}



הוספה של 1+ למספר בינארי בעל 16 ביט - Inc16

CHIP **Inc16** {

IN in[16];

OUT out[16];

PARTS:

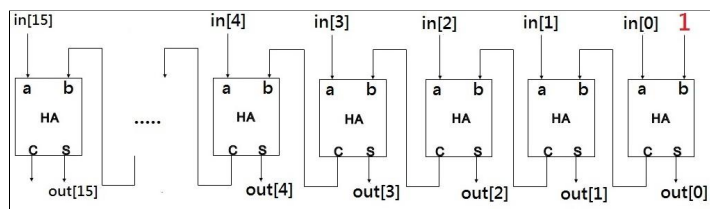
// Put you code here:

```

Add16(a = in , b[0] = true , b[1..15] = false , out=out);

```

}



יחידת החישוב האריתמטי של המחשב - ALU

CHIP ALU {

IN x[16], y[16], zx, nx, zy, ny, f, no;

OUT **out**[16], **zr**, **ng**;

PARTS:

// if zx then x = 0

Mux16(a = x, b[0..15] = false, sel = zx, out = ozx);

// if nx then x = !x

Not16(in = ozx, out = nnx);**Mux16**(a = ozx, b = nnx, sel = nx, out = exx);

// if zy then y = 0

Mux16(a = y, b[0..15] = false, sel = zy, out = ozy);

// if ny then y = !y

Not16(in = ozy, out = nny);**Mux16**(a = ozy, b = nny, sel = ny, out = exy);

// if f then out = x + y

// else out = x & y

Add16(a = exx, b = exy, out = xplusy);**Add16**(a = exx, b = exy, out = xandy);**Mux16**(a = xandy, b = xplusy, sel = f, out = fxy);

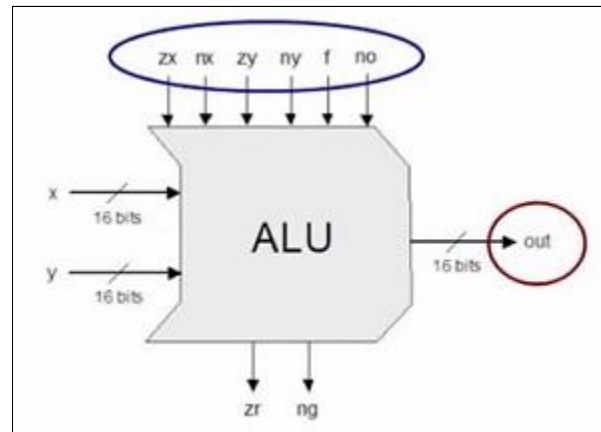
// if no then out = !out

Not16(in = fxy, out = nfxy);**Mux16**(a = fxy, b = nfxy, sel = no, out[0..7] = ret0, out[8..14] = ret1, out[15] = retsign, out = **out**);**Or8Way**(in[0..7] = ret0, out = ret0is0);**Or8Way**(in[0..6] = ret1, in[7] = retsign, out = ret1is0);**Or**(a = ret0is0, b = ret1is0, out = yzr);**Not**(in = yzr, out = **zr**);

// if out < 0 then ng = 1 else ng = 0

And(a = retsign, b = true, out = **ng**);

}



אוגר של ביט יחיד, אם הלאוד דלוק, אז הרכיב יזכור את הכניסה הנוכחית - Bit

CHIP **Bit** {

IN **in**, **load**;

OUT **out**;

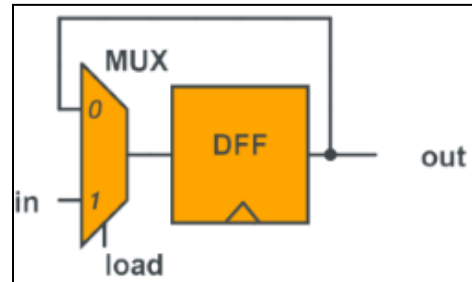
PARTS:

// Put your code here:

Mux(a = **outDFF**, b = **in**, sel = **load**, out = **outMux**);

DFF(in = **outMux**, out = **outDFF**, out = **out**);

}



אוגר של 16 ביטים, אם הלאוד דלוק אז הרכיב יזכור את הכניסה הנוכחית - Register

CHIP **Register** {

IN in[16], load;

OUT out[16];

PARTS:

// Put your code here:

Bit(in = in[0], load = load, out = out[0]);

Bit(in = in[1], load = load, out = out[1]);

Bit(in = in[2], load = load, out = out[2]);

Bit(in = in[3], load = load, out = out[3]);

Bit(in = in[4], load = load, out = out[4]);

Bit(in = in[5], load = load, out = out[5]);

Bit(in = in[6], load = load, out = out[6]);

Bit(in = in[7], load = load, out = out[7]);

Bit(in = in[8], load = load, out = out[8]);

Bit(in = in[9], load = load, out = out[9]);

Bit(in = in[10], load = load, out = out[10]);

Bit(in = in[11], load = load, out = out[11]);

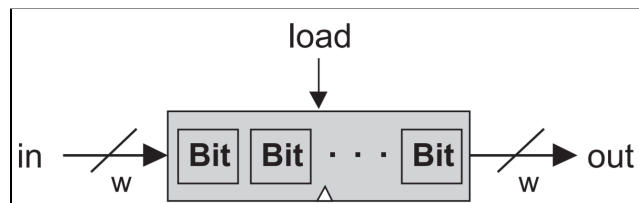
Bit(in = in[12], load = load, out = out[12]);

Bit(in = in[13], load = load, out = out[13]);

Bit(in = in[14], load = load, out = out[14]);

Bit(in = in[15], load = load, out = out[15]);

}



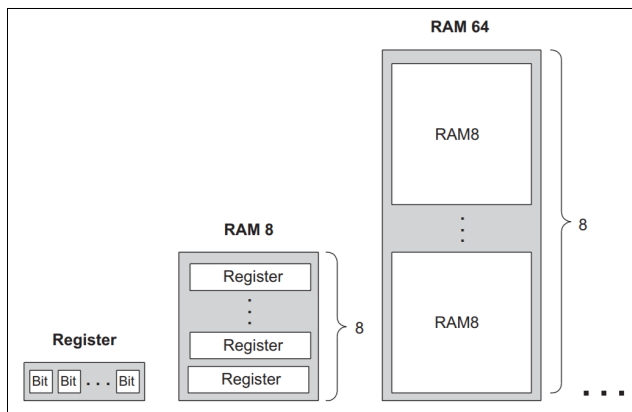
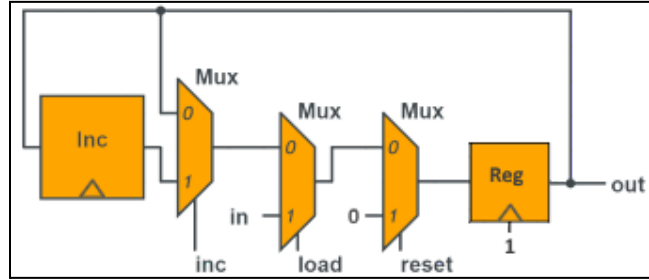
אוגר שאומר מה השורה הבאה שצריך לקרוא בקוד - PC

CHIP **PC** {
 IN **in**[16], **inc**, **load**, **reset**;
 OUT **out**[16];

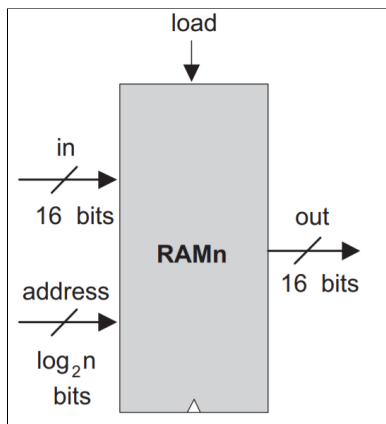
PARTS:

// Put your code here:

```
Inc16(in = was, out = after);
Mux16(a = was, b = after, sel = inc, out = afterInc); // התקדמות באחד
Mux16(a = afterInc, b = in, sel = load, out = afterLoad); // קפיצה בקוד
Mux16(a = afterLoad, b = false, sel = reset, out = afterReset); // איפוס
Register(in = afterReset, load = true, out = was, out = out);
```



נשים לב לסדר המימוש עבור כל בניית רכיב, יש כאן סוג של היררכיה וכל רכיב נבנה על בסיס רכיב אחר שבנינו כפי שאפשר לראות בציור:



לכל מימוש של רכיב RAMn, נשים לב כי גודל המערך address יהיה בעצם בגודל $\log_2 n$.
 למשל עבור RAM8 יהיה לנו address[3]

רכיב הזיכרון של המחשב - זה הרכיב שמאפשר כתיבה לתוך אוגר. זה זיכרון של 8 אוגרים - RAM8

CHIP **RAM8** {

IN **in**[16], **load** , **address**[3];

OUT **out**[16];

PARTS:

DMux8Way(in = **load** , sel = **address** , a = A , b = B ,
c = C , d = D , e = E , f = F , g = G , h = H);

Register(in = **in** , load = A , out = outA);

Register(in = **in** , load = B , out = outB);

Register(in = **in** , load = C , out = outC);

Register(in = **in** , load = D , out = outD);

Register(in = **in** , load = E , out = outE);

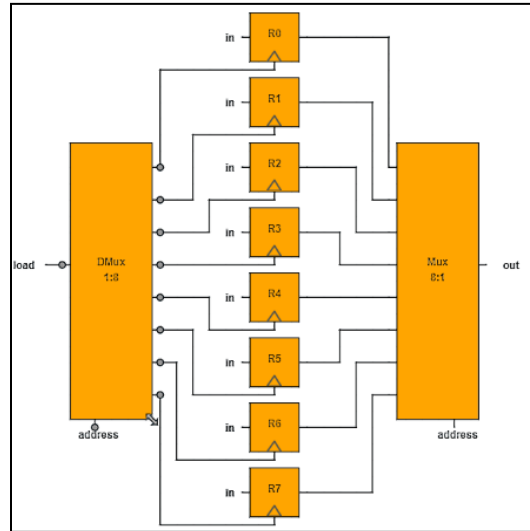
Register(in = **in** , load = F , out = outF);

Register(in = **in** , load = G , out = outG);

Register(in = **in** , load = H , out = outH);

Mux8Way16(a = outA , b = outB , c = outC , d = outD , e = outE , f = outF , g = outG , h = outH ,
sel = **address** , out = **out**);

}



כמו הקודם אבל עבור כתובת בת 6 ביטים. זיכרון של 64 אוגרים - RAM64

CHIP **RAM64** {

IN **in**[16], **load** , **address**[6];

OUT **out**[16];

PARTS:

DMux8Way(in = **load** , sel = **address**[3..5] , a = A , b = B , c = C , d = D , e = E , f = F ,
g = G , h = H);

RAM8(in = **in** , load = A , address = **address**[0..2] , out = outA);

RAM8(in = **in** , load = B , address = **address**[0..2] , out = outB);

RAM8(in = **in** , load = C , address = **address**[0..2] , out = outC);

RAM8(in = **in** , load = D , address = **address**[0..2] , out = outD);

RAM8(in = **in** , load = E , address = **address**[0..2] , out = outE);

RAM8(in = **in** , load = F , address = **address**[0..2] , out = outF);

RAM8(in = **in** , load = G , address = **address**[0..2] , out = outG);

RAM8(in = **in** , load = H , address = **address**[0..2] , out = outH);

Mux8Way16(a = outA , b = outB , c = outC , d = outD , e = outE , f = outF , g = outG ,
h = outH , sel = **address**[3..5] , out = **out**);

}

זיכרון של 512 אוגרים כל אחד מהם בגודל 16 ביטים - RAM512

```
CHIP RAM512 {
  IN in[16], load, address[9];
  OUT out[16];

  PARTS:
    DMux8Way(in = load , sel = address[6..8] , a = A , b = B , c = C , d = D , e = E , f = F , g = G , h = H);
    RAM64(in = in , load = A , address = address[0..5] , out = outA);
    RAM64(in = in , load = B , address = address[0..5] , out = outB);
    RAM64(in = in , load = C , address = address[0..5] , out = outC);
    RAM64(in = in , load = D , address = address[0..5] , out = outD);
    RAM64(in = in , load = E , address = address[0..5] , out = outE);
    RAM64(in = in , load = F , address = address[0..5] , out = outF);
    RAM64(in = in , load = G , address = address[0..5] , out = outG);
    RAM64(in = in , load = H , address = address[0..5] , out = outH);
    Mux8Way16(a = outA , b = outB , c = outC , d = outD , e = outE , f = outF , g = outG , h = outH ,
              sel = address[6..8] , out = out);
}
```

זיכרון של 4 אלף אוגרים כל אחד מהם בגודל 16 ביטים - RAM4K

```
CHIP RAM4K {
  IN in[16], load, address[12];
  OUT out[16];

  PARTS:
    // Put your code here:
    DMux8Way(in= load, sel = address[9..11] , a = A , b = B , c = C , d = D , e = E , f = F , g = G , h = H);
    RAM512(in = in , load = A , address = address[0..8], out = outA);
    RAM512(in = in , load = B , address = address[0..8], out = outB);
    RAM512(in = in , load = C , address = address[0..8], out = outC);
    RAM512(in = in , load = D , address = address[0..8], out = outD);
    RAM512(in = in , load = E , address = address[0..8], out = outE);
    RAM512(in = in , load = F , address = address[0..8], out = outF);
    RAM512(in = in , load = G , address = address[0..8], out = outG);
    RAM512(in = in , load = H , address = address[0..8], out = outH);
    Mux8Way16(a = outA , b = outB , c = outC , d = outD , e = outE , f = outF , g = outG , h = outH ,
              sel = address[9..11] , out = out);
}
```

זיכרון של 16 אלף אוגרים כל אחד מהם בגודל 16 ביטים - RAM16K

CHIP **RAM16K** {

IN **in**[16], **load**, **address**[14];

OUT **out**[16];

PARTS:

// Put your code here:

DMux4Way(in=**load**, sel = **address**[12..13], a = A, b = B, c = C, d = D);

RAM4K(in = **in**, load = A, address = **address**[0..11], out = outA);

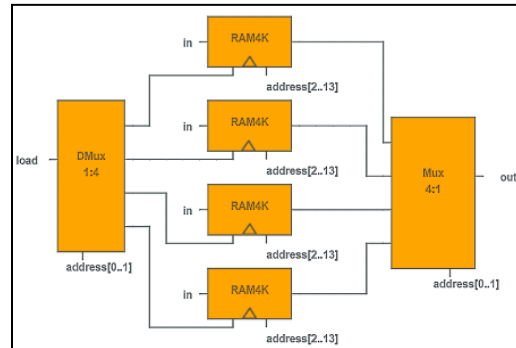
RAM4K(in = **in**, load = B, address = **address**[0..11], out = outB);

RAM4K(in = **in**, load = C, address = **address**[0..11], out = outC);

RAM4K(in = **in**, load = D, address = **address**[0..11], out = outD);

Mux4Way16(a = outA, b = outB, c = outC, d = outD, sel = **address**[12..13], out = **out**);

}



קורא/כותב את הערך שנמצא בכתובת זיכרון כלשהו - Memory

CHIP **Memory** {

IN **in**[16], **load**, **address**[15];

OUT **out**[16];

PARTS:

// Put your code here:

DMux(in=**load**, sel=**address**[14], a=**loadRAM**, b=**loadScreen**); // בוחר האם לטעון מסך או ראם

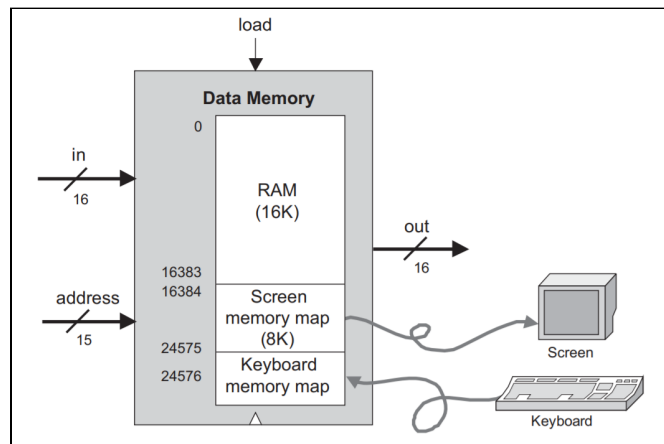
RAM16K(in=**in**, load=**loadRAM**, address=**address**[0..13], out=**outRAM**);

Keyboard(out=**outKeyboard**);

Screen(in=**in**, load=**loadScreen**, address=**address**[0..12], out=**outScreen**);

Mux4Way16(a=**outRAM**, b=**outRAM**, c=**outScreen**, d=**outKeyboard**, sel=**address**[13..14], out=**out**);

}



המחשב מכיל את המעבד, יחידת הזיכרון הראשית, אם ריסט 1 אז נפעיל מחדש - Computer

CHIP **Computer** {

IN **reset**;

PARTS:

// Put your code here:

ROM32K(address=**pc**, out=**instruction**);

CPU(inM = **memOut**, // M = content of RAM[A]

instruction = **instruction**, // Instruction for execution

reset = **reset**, // Signals whether to re-start the current program (reset == 1)

outM = **outM**, // M value output

writeM = **writeM**, // Write to M?

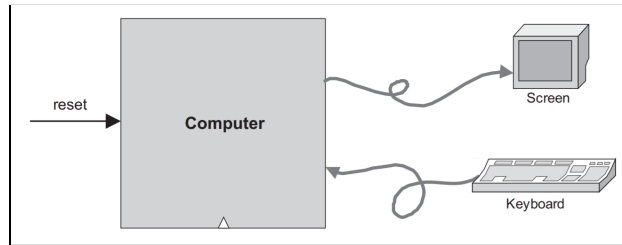
addressM = **addressM**, // Address in data memory (of M)

pc = **pc** // address of next instruction

);

Memory(in=**outM**, load=**writeM**, address=**addressM**, out=**memOut**);

}



שפת HDL (תרגילי מבחן)

2019, סמסטר ב', מועד א'

(אמריקאי) להלן תיאור hdl של Counter Program Hack, מהם שלושת השורות החסרות ?

```

/*
 * If reset(t-1) then out(t) = 0
 * else if load(t-1) then out(t) = in(t-1)
 * else if inc(t-1) then out(t) = out(t-1) + 1
 * else out(t) = out(t-1)
 */
CHIP PC {
  IN in[16],load,inc,reset;
  OUT out[16];
  PARTS:
  Inc16(in=feedback, out=inc_out);
  Mux16(-----);
  Mux16(-----);
  Mux16(-----);
  Register(in=m3, load=true, out=feedback, out=out);
}

```

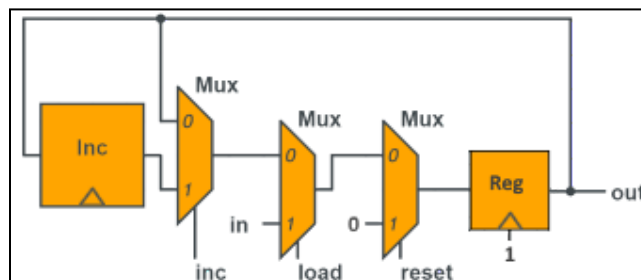
פתרון:

```

Mux16(a = feedback, b = inc_out, sel = inc, out = afterIncr);
Mux16(a = afterIncr, b = in, sel = load, out = afterLoad);
Mux16(a = afterLoad, b = false, sel = reset, out = afterReset);

```

(סעיף א' בטופס)



2019, סמסטר ב', מועד ב'

(אמריקאי) להלן תיאור hdl של Counter Program Hack, מהם שלושת השורות החסרות ?

```

/*
 * If reset(t-1) then out(t) = 0
 * else if load(t-1) then out(t) = in(t-1)
 * else if inc(t-1) then out(t) = out(t-1) + 1
 * else out(t) = out(t-1)
 */
CHIP PC {
    IN in[16],load,inc,reset;
    OUT out[16];
    PARTS:
    Inc16(in=feedback, out=inc_out);
    Mux16(a=feedback, b=inc_out, sel=inc, out=m1);
    Mux16(-----);
    Mux16(-----);
    Register(-----);
}

```

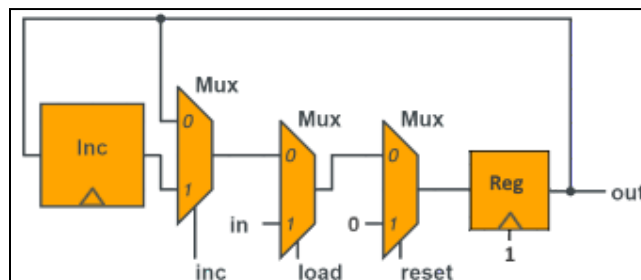
פתרון:

```

Mux16(a=m1, b=in, sel=load, out=m2);
Mux16(a=m2, b=false, sel=reset, out=m3);
Register(in=m3, load=true, out=feedback, out=out);

```

(סעיף א' בטופס)



2020, סמסטר ב, מועד א'

(אמריקאי) להלן קידוד חלקי של הרכיב RAM64. השלימו את שורות הקוד החסרות.

```

**/
* Memory of 64 registers, each 16 bit-wide. Out holds the value
* stored at the memory location specified by address. If load==1, then
* the in value is loaded into the memory location specified by address
( * the loaded value will be emitted to out from the next time step onward .)
/*
CHIP RAM64 {
    IN in[16], load, address[6];
    OUT out[16];
    PARTS :
    DMux8Way(in=load, sel=address[3..5], a=a, b=b, c=c, d=d, e=e, f=f, g=g, h=h);
    RAM8(in=in, address=address[0..2], load=a, out=A);
    RAM8(in=in, address=address[0..2], load=b, out=B);
    RAM8(in=in, address=address[0..2], load=c, out=C);
    RAM8(in=in, address=address[0..2], load=d, out=D);
    RAM8(in=in, address=address[0..2], load=e, out=E);
    RAM8(in=in, address=address[0..2], load=f, out=F);
    RAM8(in=in, address=address[0..2], load=g, out=G);
    _____
    _____
}

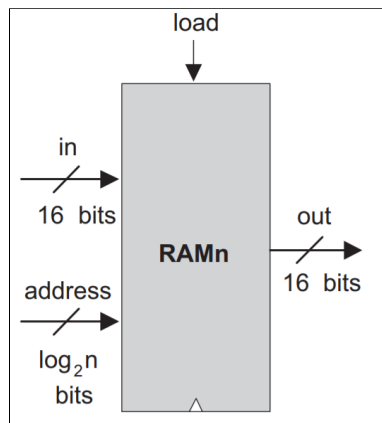
```

פתרון:

```

RAM8(in=in, address=address[0..2], load=h, out=H);
DMux8Way16(a=A, b=B, c=C, d=D, e=E, f=F, g=G, h=H, sel=address[3..5], out=out);

```



2020, סמסטר ב', מועד ב'

(אמריקאי) להלן קידוד חלקי של הרכיב RAM64. השלימו את שורות הקוד החסרות.

```

**/
* Memory of 64 registers, each 16 bit-wide. Out holds the value
* stored at the memory location specified by address. If load==1, then
* the in value is loaded into the memory location specified by address
( * the loaded value will be emitted to out from the next time step onward .)
/*
CHIP RAM64 {
    IN in[16], load, address[6];
    OUT out[16];
    PARTS :
        _____

        _____

    RAM8(in=in, address=address[0..2], load=b, out=B);
    RAM8(in=in, address=address[0..2], load=c, out=C);
    RAM8(in=in, address=address[0..2], load=d, out=D);
    RAM8(in=in, address=address[0..2], load=e, out=E);
    RAM8(in=in, address=address[0..2], load=f, out=F);
    RAM8(in=in, address=address[0..2], load=g, out=G);
    RAM8(in=in, address=address[0..2], load=h, out=H);
    DMux8Way16(a=A, b=B, c=C, d=D, e=E, f=F, g=G, h=H, sel=address[3..5], out=out);
}

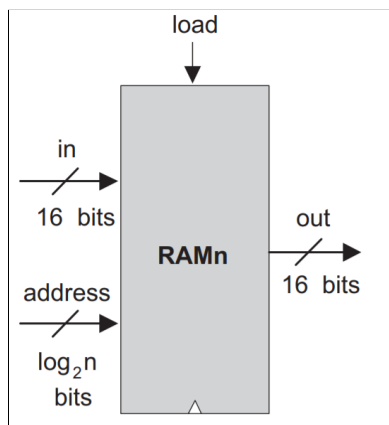
```

פתרון:

DMux8Way(in=load, sel=address[3..5], a=a, b=b, c=c, d=d, e=e, f=f, g=g, h=h);

RAM8(in=in, address=address[0..2], load=a, out=A);

(סעיף ד' בטופס).

**2021, סמסטר ב', מועד א'**

שאלה ממחוזרת ממועד 2019 א'

2021, סמסטר ב', מועד ב'

שאלה ממחוזרת ממועד 2020 א'

תכנות ב- Mips אסמבלי (הסבר)

- Mips זה ארכיטקטורה שהייתה נפוצה בשנות ה-80.
- בקורס למדנו תכנות Mips בשפת אסמבלי.
- אפשר לבצע פעולות על אוגרים, תאי זיכרון וקבועים.
- הגישה לאוגרים (CPU) מהירה יותר מאשר לתאי הזיכרון (RAM).
- בארכיטקטורה של MIPS לאוגרים ותאי הזיכרון יש 32 ביטים (במקום 16 שהכרנו מHACK).

פקודות ב-Mips

נציג את הפקודות בהשוואה לשפת C כדי להבין טוב יותר.

משולב	חיסור	חיבור	
$a = b + c - d$	$a = b - c;$	$a = b + c;$	C
add t, b, c sub a, t, d	sub a, b, c	add a, b, c	Mips Assembly

אוסף האוגרים ב-Mips

- ה-\$ מסמל קריאה לאוגר ב-CPU.
- ה-\$0 הוא אוגר במיקום ה-0 ולרוב הוא יכיל ערך 0.
- כשנגיע לפונקציות אנחנו נשתמש ובין יותר טוב מה ההבדל בין אוגרים זמניים ואוגרי שמירה.
- ה-\$ra - כאשר אנחנו חוזרים מפונקציה, לאן לחזור.
- אוגרים \$s0 - \$s7, הם אוגרי שמירה שנועדו להחזיק משתנים.
- אוגרים \$t0 - \$t9, הם אוגרים זמניים - משמש להחזיק ערכי ביניים במהלך חישוב גדול יותר.

Name	Register Number	Usage
\$0	0	the constant value 0
\$at	1	assembler temporary
\$v0-\$v1	2-3	Function return values
\$a0-\$a3	4-7	Function arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved variables
\$t8-\$t9	24-25	more temporaries
\$k0-\$k1	26-27	OS temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	Function return address

תאי זיכרון

- מלבד האוגרים, יש לנו עוד תאי זיכרון.
- לוקח זמן לגשת אליהם, אך הם מאפשרים לזכור הרבה מידע (יש הרבה תאים).
- לכל תא זיכרון יש כתובת.
- כל תא זיכרון הוא ברוחב **word**, שזה 32 ביט, שזה 4 בתים, והכתובות של תאי הזיכרון אכן גדלות ב-4 בכל פעם, היות ו MIPS הוא addressed-byte.
- קריאה מזיכרון - טוענים מהזיכרון את המילה, ולכן הפקודה היא **lw** מלשון Load Word.
- כאשר אנחנו רוצים לטעון לתוך האוגר \$s0 את הערך שנמצא בזיכרון נכתוב:

`lw $s0, 5($t1)`

כאשר אנחנו רוצים לקרוא מהערך שנמצא בתוך \$t1 (לדוגמה 1124) ועד 5 ולכן נרצה לקרוא ל-1129. חשוב לשים לב שבגלל שהכתובות גדלות ב-4 בכל פעם, יהיה צריך לכתוב פקודות שהכתובות בזיכרון מתחלקות ב-4.

- עבור שמירת ערך לתא בזיכרון נשתמש בפקודה **sw** מלשון Save Word.
- לדוגמה נרצה לשמור עם הערך שנמצא ב-\$s0 אז נכתוב:

`sw $s0, 5($t1)`

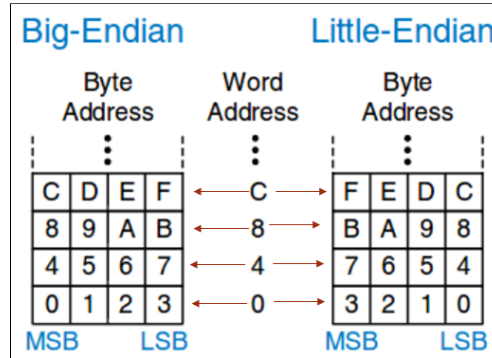
- הפקודות **lb** טוען בית יחיד ו-**wb** שומר בית יחיד.

Word Address	Data								
⋮	⋮								
0000000C	4	0	F	3	0	7	8	8	Word 3
00000008	0	1	E	E	2	8	4	2	Word 2
00000004	F	2	F	1	A	C	0	7	Word 1
00000000	A	B	C	D	E	F	7	8	Word 0

width = 4 bytes

קריאה וכתובת בית

כלומר, אם אנחנו רוצים לקרוא/לכתוב בית, ולא מילה, ניתן לכתוב מספר שאינו כפולה של 4. שתי שיטות:



דוגמה להבדלים בין Big לבין Little (רלוונטי למבחן).

נניח שאוגר \$t0 מכיל את הכתובת 0x23456789.

לאחר שהקוד הבא רץ:

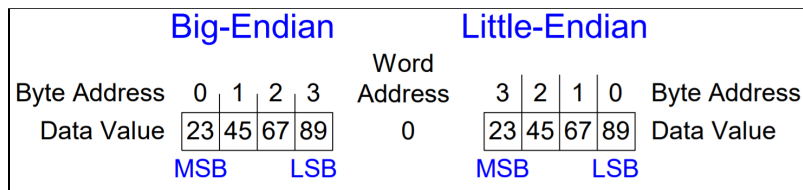
```
sw $t0, 0($0)
```

```
lb $s0, 1($0)
```

איזה ערך נמצא בתוך אוגר \$s0?

Big – endian: 0x00000045

Little – endian: 0x00000067

**פקודות R-Type**

זוהי פקודת registers, היא תקבל (כמעט תמיד) 3 אוגרים.

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

• פעולות על אוגרים:

○ rs, rt - אוגרי המקור (האוגרים שאיתם מבצעים את החישוב).

○ rd - אוגרי היעד (האוגר אליו נשמור את החישוב).

• שדות אחרים:

○ op - תמיד יהיה אפס.

○ funct - מה הפקודה שאותה אנחנו מבצעים.

○ shamt - פקודות shift שנראה בהמשך..

דוגמה:**Assembly Code**

```
add $s0, $s1, $s2
```

```
sub $t0, $t3, $t5
```

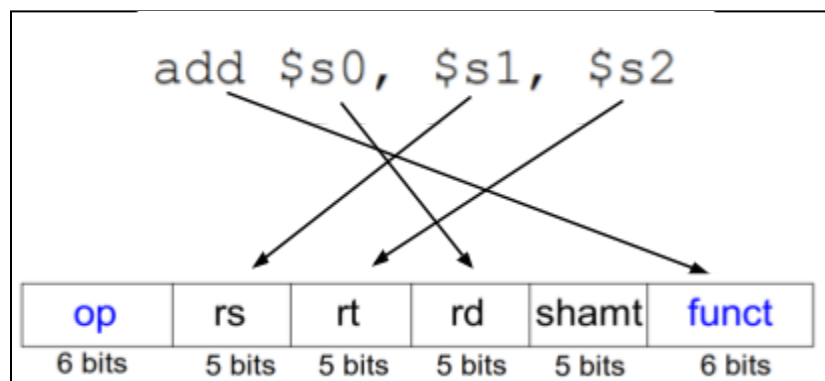
Field Values

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

בשורה הראשונה, אפשר לראות כי הקוד לפעולה של add הוא 32. בנוסף לכל אוגר יש מספר קבוע משלו (כמו שמוצג בטבלה בעמוד הראשון של הפרק של mips פה).

הערה חשובה!

הסדר כתיבה באסמבלי (הקוד השמאלי בתמונה למעלה) שונה מהסדר של ערכי השדות (הטבלה הימנית מלמעלה).



כלומר, הסדר הוא:

add rd, rs, rt

פקודות i-Type

זוהי פקודה על קבוע.

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

• 3 פעולות:

- rs, rt - פעולות על אוגרים
- imm - כמות ההזזות פלוס הWORD שאותה נרצה לבדוק.
- op - יגדיר לנו את הפקודה אותה נרצה לבצע (addi...).

נשים לב לסדר הכתיבה התקין:

addi rt, rs, imm

- נציין כי **addi** זה Add immediate ובו אפשר להוסיף/לחבר באופן מיידי מספרים כלשהם.

דוגמה:

Assembly Code	Field Values			
	op	rs	rt	imm
addi \$s0, \$s1, 5	8	17	16	5
addi \$t0, \$s3, -12	8	19	8	-12
lw \$t2, 32(\$0)	35	0	10	32
sw \$s1, 4(\$t1)	43	9	17	4

חשוב מאוד!

צריך לשים לב לשוני בין האסמבלי למכונה.

• מכונה -

op rs rt imm

• אסמבלי -

addi rt, rs, imm

lw rt, imm(rs)

sw rt, imm(rs)

פקודות J-Type

אלו פקודות של קפיצה

op	addr
6 bits	26 bits

• addr - כתובת בת 26 בתים.

• op - סוג הקפיצה.

פקודות לוגיות

Assembly Code
and \$s3, \$s1, \$s2
or \$s4, \$s1, \$s2
xor \$s5, \$s1, \$s2
nor \$s6, \$s1, \$s2

• פקודת AND משתמשת לאפס ביטים, פקודת OR משמשת לאחד (להעלות) ביטים

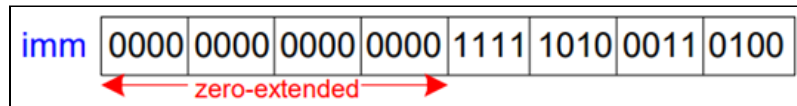
פעולות מיידיות

- **andi, ori, xori** - הן פקודות immediate והן חלק מפקודות i-Type.

Assembly Code

```
andi $s2, $s1, 0xFA34
ori  $s3, $s1, 0xFA34
xori $s4, $s1, 0xFA34
```

- מכיוון שיש ל imm רק 16 ביטים לשימוש המערכת תמיד תרפד את יתר המספר להשלמה ל 32 (הוספה של עוד 16 אפסים).



- 32 WORD- ביטים.
- 8 ביטים זה בית.
- 4 ביטים נקראים ניבל.

פקודות Shift

- **sll**: shift left logical
○ לדוגמה:

`sll $t0, $t1, 5 # $t0 <= $t1 << 5`

- **srl**: shift right logical
○ לדוגמה:

`srl $t0, $t1, 5 # $t0 <= $t1 >> 5`

- **sra**: shift right arithmetic

- ההבדל שעכשיו אנחנו נשכפל את ה MSB עבור כל הזזה במקום להציב 0. ובכך אנחנו שומרים על הסיבית סימן.
- לדוגמה:

`ra $t0, $t1, 5 # $t0 <= $t1 >>> 5`

- **shamt** - נרשום את כמות הביטים אותם נרצה להזיז.
- **sllv, srlv, sra** - כאשר v הוא variable. הוזזה בכמות לא ידועה.
- לדוגמה:

`srlv $t0, $t1, $t2`

דוגמאות ל-shifts (טוב להבנה):

- להזיז שמאלה זה להכפיל פי 2 בכמה שהזזנו.
- להזיז ימינה זה לחלק ב-2 בכמה שהזזנו (בלי שארית).
- עבור **sll** (שמאלה) נקפוץ פעמיים:
 $0x00000004 \rightarrow 0x00000010$ ($4 \rightarrow 16$)
 כלומר, היינו ב-4, וכדי לזוז שמאלה צריך להכפיל ב-2, כיוון שאנחנו קופצים פעמיים אז נכפול פעמיים ב-2 ולכן $16 = 2 \cdot 2 \cdot 4$ כלומר קפצנו ל-16 וב-HEX זה מתורגם ל- $0x00000010$.
- עבור **srl** (ימינה) נקפוץ 3 פעמים:
 $0x0000001f \rightarrow 0x00000003$ ($31 \rightarrow 3$)
 כיוון שאנחנו קופצים ימינה, אנחנו צריכים לחלק ב-2 שלוש פעמים.
 המספר 1f בדצימלי הוא 31 ולכן נקבל: $3 = \frac{31}{2 \cdot 2 \cdot 2}$ (בלי שארית).
- עבור **sra** (ימינה) נקפוץ 3 פעמים:
 $0x0000001f \rightarrow 0x00000003$
- עבור **sra** (ימינה) נקפוץ 2 פעמים:
 $0xffffffff0 \rightarrow 0xffffffffC$ ($-16 \rightarrow -4$)

שימוש של כל 32 ביט

- כאשר אנחנו משתמשים בפקודות `add`, `addi` ... אנחנו משתמשים רק ב-16 ביטים לכן נוכל לבצע הכנסת ערכים על ידי שתי אפשרויות:
1. הכנסת ערך על ידי פונקציה של 16 ביט ואז `SLL` ואז הכנסה נוספת להוספה לביטים הנמוכים.
 2. שימוש בפונקציית **lui** - אנחנו מכניסים לערך העליון ואז שימוש ב **ori** להכנסה של החלק התחתון.

לדוגמה:

- בשפת C נכתוב:
`int a = 0xFEDC8765;`
- בשפת Mips אסמבלי נכתוב:
`# $s0 = a`
`lui $s0, 0xFEDC`
`ori $s0, $s0, 0x8765`

פעולות כפל וחילוק

- פרט לאוגרים שכבר דיברנו עליהם, יש לנו עוד שניים: **hi, lo**.
- הם משמשים להכפלה וחלוקה, שהרי הכפלת 32 ביטים ב 32 ביטים מאפשרת תוצאה עד 64 ביטים.
- החלק העליון של התוצאה יהיה שמור ב **hi**, והחלק התחתון של התוצאה שמור ב **lo**.

`mult $s0, $s1    #result stores at {hi, lo}`

`div $s0, $s1    #result stores at {hi lo}. Hi is the quotient, lo is the remainder`

- נוכל להוציא את המידע מהאוגרים על ידי פוקדת `mov`:
- `mfio $s2`
- `mfhi $s4`
- מוצאים את הערך מהאוגרים ששומרים לאוגרים שאנחנו בוחרים לדוגמה `$.s4`.

פקודות unsigned

`addu, addiu, subu, multu, divu`

תנאים

כשנרצה לבצע פעולות if נשתמש בפקודות הבאות:

- הפקודה **beq** (פקודת i-Type) אומרת if equals למשל:
`beq $s0, $s1, target`
כאשר `target` זה קטע קוד שאליו נרצה לקפוץ במידה והאוגרים שווים.
- הפקודה **bne** (פקודת i-Type) אומרת if not equals למשל:
`bne $s0, $s1, target`
כאשר `target` זה קטע קוד שאליו נרצה לקפוץ במידה והאוגרים לא שווים.

קפיצות ללא תנאי

- הפקודה **j** (פקודת j-Type) תקפוץ לקטע קוד ללא תנאי למשל:
`j target`
- הפקודה **jal** (פקודת j-Type) תקפוץ ללא תנאי לקטע קוד וגם תזכור לאיפה חוזרים שמסיימים את הפונקציה למשל:
`jal target`
- הפקודה **jr** (פקודת R-Type) תקפוץ לכתובת הרשומה אצל האוגר ללא תנאי למשל:
`jr $s0`

תכנות מתקדם

- תנאי if\else
- לולאות for
- לולאות while
- מערכים
- קריאות לפונקציות

נציג את כל אלה בדוגמאות במקביל לשפת C כדי להקל על ההבנה.

תנאי if\else	
C code	MIPS Assembly Code
<pre>if (i == j) f = g + h; f = f - i;</pre>	<pre>bne \$s3, \$s4, L1 add \$s0, \$s1, \$s2 j done L1: sub \$s0, \$s0, \$s3 done:</pre>

לולאת While	
C code	MIPS Assembly Code
<pre>// determines the power // of x such that 2 x = 128 int pow = 1; int x = 0; while (pow != 128) { pow = pow * 2; x = x + 1; }</pre>	<pre>addi \$s0, \$0, 1 add \$s1, \$0, \$0 addi \$t0, \$0, 128 while: beq \$s0, \$t0, done sll \$s0, \$s0, 1 addi \$s1, \$s1, 1 j while done:</pre>

- תזכורת sll זה קפיצה שמאלה.

לולאות For	
C code	MIPS Assembly Code
<pre>// add the numbers from 0 to 9 int sum = 0; int i; for (i=0; i!=10; i = i+1) { sum = sum + i; }</pre>	<pre>addi \$s1, \$0, 0 add \$s0, \$0, \$0 addi \$t0, \$0, 10 for: beq \$s0, \$t0, done add \$s1, \$s1, \$s0 addi \$s0, \$s0, 1 j for done:</pre>

לולאות For עם תנאי less than	
C code	MIPS Assembly Code
<pre>// add the powers of 2 from 1 // to 100 int sum = 0; int i; for (i=1; i < 101; i = i*2) { sum = sum + i; }</pre>	<pre>addi \$s1, \$0, 0 addi \$s0, \$0, 1 addi \$t0, \$0, 101 loop: slt \$t1, \$s0, \$t0 beq \$t1, \$0, done add \$s1, \$s1, \$s0 sll \$s0, \$s0, 1 j loop done:</pre>

- הפעולה **slt** אומרת set less than, כלומר פעולת השוואה של פחות מ... במידה ו-s0 קטן ממש מ-t0 אז t1 יקבל את הערך 1 אחרת, t1 יקבל את הערך 0.

מערכים

- מערך הינו רצף של תאי זיכרון המכיל מידע מאותו "סוג" (כל מידע הוא 32 ביטים, זה לא משתנה).
נגיד, אם המערך מתחיל בכתובת 0x12348000 (כלומר `array[0]`), אז התאים הבאים של המערך ישובו בהמשך הזיכרון: 0x12348008, 0x12348004, כתובת התא הראשון של המערך נקראת `base address`.
- הדבר הראשון שצריך לעשות כדי להכניס מידע לזיכרון, או כדי להוציא אותו, זה להטעין את ערך הזיכרון הרצוי.

דוגמה:

הכניסו את הערכים 1,2,3...10 לתאי הזיכרון שבכתובות 0x10010000, 0x10010008, 0x10010010.

```
lui $s0, 0x1001 #load 0x1001 into upper half of s0
addi $s1, $0, 1 #s1 = 1 להכנסה
addi $t0, $0, 11 # לא כולל 11 זה הגבול כלומר עד 11
loop: # תחילת הלולאה
    beq $s1, $t0, exit # $s1 == 11 נסיים את הלולאה
    sw $s1, 0($s0) #insert s1 data into memory cell s0+0 הערך
    addi $s0, $s0, 8 # הקסדצימלי 8 זה 16 לפי ההגדרה -
    addi $s1, $s1, 1 # i++ זה כמו
    j loop
exit:
```

פונקציות

- קריאה לפונקציה, שאתם מכירים מ java, היא למעשה קפיצה לכתובת של הפונקציה.
 - פקודת הקפיצה (`func jal (link and jump)`).
 - כשקופצים לפונקציה צריך לזכור כמה דברים:
1. לאיפה חוזרים כשמסיימים את הפונקציה? - למשל לשורה שאחרי השורה שקראה לה (+4).
הכתובת הזו נשמרת אוטומטית באוגר **ra** אשר קופצים.
 2. בסוף הפונקציה נקפוץ חזרה בעזרת `jr $sr` (קפוץ לכתובת הרשומה באוגר ra).
אוגרי "פרמטרים לפונקציה" הם האוגרים `$a0 - $a3` ואוגרי "ערך חוזר מפונקציה" הם `$t0 - $t1`, כשנחזור מהפונקציה נעביר את המידע מ `v0` לאוגר שימש כיעד. (`add עם $0`).
 3. צריך לא לדרוס אוגרים קיימים במהלך הפונקציה.
- כמה מילים על **jal** - זהה בדיוק לפקודת `Jump`, רק שלפני ש-PC משתנה לערכו החדש, כתובת הפקודה הבאה (`PC + 4`) נשמרת ברגיסטר RA.
- משמשת בעת קריאה לפונקציה כדי שנזכור ברגיסטר `return address (RA)` לאיפה לחזור בסוף הפונקציה.

תרגיל מבחן לדוגמה:

תרגמו את הקוד הבא לאסמבלי:

```
int main() {
    int x = sum(3, 4, 5, 6);
}

int sum(int a, int b, int c, int d){
    return a+b+c+d;
}
```

פתרון:**main:**

```
addi $a0, $0, 3 # מציב את 3
addi $a1, $0, 4 # מציב את 4
addi $a2, $0, 5 # מציב את 5
addi $a3, $0, 6 # מציב את 6
jal sum # קופץ לפונקציה וגם שומר באוגר אר.איי את הכתובת חזרה למקום הזה אחרי הסיום
add $s0, $v0, $0 # אוגריי "ווי" הם אוגרי החזרה מפונקציה לכן נציב אותם באוגרי "אס"
j done # סוף הקוד - קפיצה לסיום
```

sum:

```
add $t0, $a0, $a1 # סוכם את 3+4 לתוך אוגר טי אפס
add $t1, $a2, $a3 # סוכם את 5+6 לתוך אוגר טי אחד
add $v0, $t0, $t1 # סוכם את שני האוגרים לתוך אוגר השמירה מהפונקציה שהם אוגרי וויי
jr $ra # וקופץ חזרה לכתובת שנשמרה באוגר אר.איי
```

done: # סוף קודקריאה לפונקציות בתוך פונקציות (המחסנית)

- בפונקציות יש לנו 2 מצבים שבהם נרצה לשמור נתונים:
 1. עבור אוגרי S אנחנו נרצה לשמור את הנתונים שלהם מהיכן שהפונקציה נקראת. ועבור אוגרי T נרצה שבתוך הפונקציה לשמור את האוגרים (לרוב לא נצטרך לשמור את אוגרי T כי הם זמניים).
 2. כאשר אנחנו קופצים מפונקציה לפונקציה אנחנו צריכים לשמור את הערך של ra לאן חוזרים ולא לדרוס אותו. כי כל פעם אנחנו זוכרים רק את הפונקציה האחרונה שנקראה.
- לכן יהיה לנו **מחסנית** שאיתה נשתמש לזכור.
- המחסנית יושבת מכתובת 0x7ffffffc ומטה (מכיוון שזאת מחסנית LIFO, אז גם הזיכרון מנוצל מלמעלה למטה)

Address	Data		Address	Data
7FFFFFFC	12345678	←\$sp	7FFFFFFC	12345678
7FFFFFF8			7FFFFFF8	AABBCCDD
7FFFFFF4			7FFFFFF4	11223344 ←\$sp
7FFFFFF0			7FFFFFF0	
⋮	⋮		⋮	⋮
⋮	⋮		⋮	⋮

- האוגר \$sp (stack pointer) הוא מצביע לראש המחסנית וגדל "למטה" מהכתובת הגבוהה ביותר אל הכתובת הנמוכה ביותר במחסנית.

שיטה לשמור אוגרים במחסנית

נחזור לפונקציה sum הדומה מהעמוד הקודם, בתחילת הפונקציה, נקצה מקום במחסנית לכל האוגרים שאנחנו רוצים לשמור בצד (באדום):

sum:

```

addi $sp, $sp, -8    # נקצה מקום במחסנית ל-2 אוגרים, כל אוגר תופס נפח של 4
sw $t0, 4($sp)       # store t0 on stack[1]
sw $t1, 0($sp)       # store t1 on stack[2]
add $t0, $a0, $a1    # כמו הפונקציה מהעמוד הקודם
add $t1, $a2, $a3    # כמו הפונקציה מהעמוד הקודם
add $v0, $t0, $t1    # כמו הפונקציה מהעמוד הקודם
lw $t1, 0($sp)       # שחרור נתונים t1 from stack
lw $t0, 4($sp)       # שחרור נתונים t0 from stack
addi $sp, $sp, 8     # deallocate memory from stack
jr $ra

```

שיטה לקריאה של פונקציות מתוך פונקציות בעזרת המחסנית

נחזור לפונקציה sum הדומה מהעמוד הקודם, כשאנחנו רוצים לשמור את הזיכרון של לאן לחזור ומאיפה לחזור, נקצה word אחד בתוך המחסנית (בירוק) ונבקש לשמור בתוכה את אוגר ra (שידוע כאוגר ששומר את הכתובת חזרה מהפונקציה):

sum:

```

addi $sp, $sp, -4    # נקצה תא אחד במחסנית שלנו בשביל לשמור את כתובת החזרה
sw $ra, 0($sp)       # נשמור באוגר אר.איי את כתובת החזרה לפונקציית המיין
add $t0, $a0, $a1    # כמו הפונקציה מהעמוד הקודם
add $t1, $a2, $a3    # כמו הפונקציה מהעמוד הקודם
add $v0, $t0, $t1    # כמו הפונקציה מהעמוד הקודם
jal sum2             # נקפוץ לפונקציה הבאה
lw $ra, 0($sp)       # עכשיו האוגר אר.איי יקבל את הכתובת לשורה הזאת וישמר במחסנית
addi $sp, $sp, 4     # נמחק מהזיכרון
jr $ra

```

- הערה חשובה - ברגע שנגיע בקוד לשורה של קפיצה לפונקציה למשל **jal** אז האוגר ra באופן אוטומטי ומייד יתעדכן "להצביע" על הכתובת בשורה אחרי ה-**jal**.

רקורסיה

פונקציית רקורסיה של חישוב עצרת בשפות High Level:

```
int factorial(int n) {
    if (n <= 1)
        return 1;
    else
        return (n * factorial(n-1));
}
```

פונקציית רקורסיה של חישוב עצרת בשפת MIPS Assembly:

0x90 factorial:	addi \$sp, \$sp, -8	# הקצאת מקום ל-2 אוגרים במחסנית
0x94	sw \$a0, 4(\$sp)	# הכנסת האוגר הראשון זה עבור האיבר ברקורסיה
0x98	sw \$ra, 0(\$sp)	# הכנסת אוגר אר.איי לשמירת כתובת חזרה
0x9C	addi \$t0, \$0, 2	# נציב גבול עצירה לרקורסיה עד לערך 2
0xA0	slt \$t0, \$a0, \$t0	# אם ה-aן שלנו קטן מ-2 אז טיס יהיה 1 ואז נעצור
0xA4	beq \$t0, \$0, else	# אם טיס לא 1 אז נרצה להכנס שוב רקורסיבית
0xA8	addi \$v0, \$0, 1	# זה הקריאה האחרונה, נכניס לפה 1 ונחזיר אותו
0xAC	addi \$sp, \$sp, 8	# נשחרר מקום מהמחסנית
0xB0	jr \$ra	# קופצים למי שקרא לנו - ונחזור לשורה ה ירוקה
0xB4 else:	addi \$a0, \$a0, -1	# n = n - 1
0xB8	jal factorial	# כאן אנחנו חוזרים שוב ושוב לשורה ה אדומה
0xBC	lw \$ra, 0(\$sp)	# שחרור אוגר אר.איי \$ra
0xC0	lw \$a0, 4(\$sp)	# נשחרר את ה-aן שלנו \$a0
0xC4	addi \$sp, \$sp, 8	# נמחק מהזיכרון, כלומר נפנה מקום \$sp
0xC8	mul \$v0, \$a0, \$v0	# n * factorial(n-1)
0xCC	jr \$ra	# חוזרים אחורה ברקורסיה - שורה ירוקה return

- חשוב מאוד - זה לא הקוד המלא, אנחנו חייבים גם לכתוב פונקציית Main שקוראת לפונקציה factorial ומשתמשת ב-jal כדי שנדע לאן לחזור ומתי. אחרת - הרקורסיה תפעל באופן אינסופי.

Address	Data
FC	
F8	
F4	
F0	
EC	
E8	
E4	
E0	
DC	

הסבר על הקוד הרקורסיבי הקודם עם ציור:

שלב 1 - האיתחול:

בהתחלה, קראנו לפונקציה factorial מה-Main, המחסנית ריקה:
ה-sp מצביע על ראש המחסנית שלנו.

Address	Data
FC	
F8	\$a0 (0x3)
F4	\$ra
F0	\$a0 (0x2)
EC	\$ra (0xBC)
E8	\$a0 (0x1)
E4	\$ra (0xBC)
E0	
DC	

שלב 2 - כניסות הרקורסיביות עד תנאי עצירה:

בדוגמה הזאת, נכנסו רקורסיביות פעמיים, כלומר $n=3$.

אפשר לראות כי בפעם הראשונה קראנו מהפונקציה main-מה:
ה-F8 שמר את $n=3$.

ה-F4 שמר את אוגר ra כדי שנדע את כתובת החזרה שלנו.

בפעם השנייה נשלחנו רקורסיבית שוב לפונקציה:

ה-F0 שמר את $n=2$.

ה-EC שמר את אוגר ra שיזכור לחזור לקריאה הראשונה לפונקציה.

בפעם השלישית (זה הקריאה הרקורסיבית השנייה) נשלחנו שוב לפונקציה:

ה-E8 שמר את $n=1$.

ה-E4 שמר את האוגר שמכיל את כתובת החזרה לקריאה הקודמת.

Address	Data
FC	
F8	\$a0 (0x3)
F4	\$ra
F0	\$a0 (0x2)
EC	\$ra (0xBC)
E8	\$a0 (0x1)
E4	\$ra (0xBC)
E0	
DC	

שלב 3 - חזרה רקורסיבית וחישוב העצרת:

ניתן לחשוב מהקוד שבעצם כל הזמן מחקנו את הנתון ודרסנו כל פעם את ה-n שלנו, אבל כל הנתונים בכל שלב רקורסיבי נשמור במחסנית.

בהתחלה המצביע sp מצביע לתחתית בכתובת E4 וכעת הוא

יחזור למעלה במחסנית + חישוב העצרת בכל שלב עד שהמצביע יגיע לראש המחסנית ושם יחזור לפונקציה - ה-ra בראש המחסנית זוכר את הכתובת חזרה לפונקציית ה-main.

תרגום קוד (חשוב למבחן)

נתונים:

 $\text{funct}(\text{sub})=34, \text{funct}(\text{add})=32, \text{opcode}(\text{beq})=4, \text{opcode}(\text{sw})=43, \text{opcode}(\text{j})=2$

תרגמו את שורת הקוד הבאה לבינארית (במבחן יבקשו להקסא-דצימלי):

0x00400000: here: add \$s0, \$s1, \$s2

פתרון והסבר:

- שאלות מהסוג הזה אתם תקבלו את ההגדרות של הערכים של כל פקודה (הנתון למעלה).
- נתרגם את זה תחילה לבינארי כאשר אנחנו זוכרים איך פקודות מסוג R-type נראות במבחן אנחנו נקבל את האיורים הבאים כדף עזר (מימין R-Type ומשמאל מספרי האוגרים):

Name	Register Number
\$0	0
\$at	1
\$v0-\$v1	2-3
\$a0-\$a3	4-7
\$t0-\$t7	8-15
\$s0-\$s7	16-23
\$t8-\$t9	24-25
\$k0-\$k1	26-27
\$gp	28
\$sp	29
\$fp	30
\$ra	31

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- נרצה לתרגם את השורה הנתונה לנו ולהתאים אותה לפי הסדר:

here: add \$s0, \$s1, \$s2

- תחילה נסדר את השורה לתוך מבנה R-Type לפי הסדר התקין לכתובות:

נשים לב כי הסדר הפוך ומבלבל וחשוב מאוד לא לאבד את זה, + ה-here שכתוב בשורה לא רלוונטי:

op(6 bits)	rs(5 bits)	rt(5 bits)	rd(5 bits)	shamt(5)	funct(6)
	\$s1	\$s2	\$s0		add

- כעת במקום השמות שכתבנו בטבלה בהתאם לשורה, נציב את המספר המתאים של כל אוגר, נסתכל על הטבלה משמאל של האוגרים ונקח מכל אחד את המספר שלו, למשל s0 המספר שלו הוא 16 ועבור הפונקציה add נתון לנו בתרגיל כי המספר שלו הוא 32 (ראו בתחילת הדף). בנוסף op לא קיים אף פעם לכן נציב עבורו מראש אפסים וגם אין כאן shamt ולכן גם עבורו נשים אפסים:

op(6 bits)	rs(5 bits)	rt(5 bits)	rd(5 bits)	shamt(5)	funct(6)
0	17	18	16	0	32

- נתרגם כל מילה לבינארי ונשמור על אורך של ביטים של כל אוגר, במבחן לא קיים טבלת עזר להמרה:

op(6 bits)	rs(5 bits)	rt(5 bits)	rd(5 bits)	shamt(5)	funct(6)
000000	10001	10010	10000	00000	100000

- נקבל כי תרגמנו לבינארית את השורה הנתונה:

000000 10001 10010 10000 00000 100000

תכנות ב- Mips אסמבלי (תרגילי מבחן של תוכן אוגר עם Endian)

2018, סמסטר ב', מועד א', גרסה 1

(אמריקאי) נניח שמחשב MIPS הוא Big Endian והאוגר \$t0 מכיל 0x23456789, מה יהיה תוכן האוגר \$s0 לאחר ביצוע הפקודות הבאות:

```
sw $t0, 0($0)
lb $s0, 1($0)
```

פתרון:

האוגר \$s0 יכיל את 0x00000045.

הסבר:

בפרק ההסבר הוצג לנו כי word בנוי מ-4 בתים כך שבכל בית יושב צמד מהכתובת כלומר: 0x23456789. הפקודה **lb** טוענת לנו בית יחיד, איזה בית? - בית מספר 1 כי נתון לנו בשורה השניה ש- (\$0) 1. כיוון שמדובר ב-Big Endian אז נבחר בבית מספר 1 (אנחנו מתחילים מ-0 כמו במערך) משמאל לימין ולכן בחרנו ב-45 ולכן ב-\$s0 יהיה 0x00000045.

2018, סמסטר ב', מועד א', גרסה 2

(אמריקאי) נניח שמחשב MIPS הוא Little Endian, מה יהיה תוכן אוגר \$s0 לאחר ביצוע הפקודות הבאות:

```
lui $t0, 0x1234
ori $t0, 0xABCD
sw $t0, 100($0)
lbu $s0, 102($0)
```

פתרון:

```
0x12340000
0x1234ABCD
```

ולכן נקבל:

```
0x00000034
```

2018, סמסטר ב, מועד ב'

(אמריקאי) נניח שמחשב MIPS הוא Big Endian, מה יהיה תוכן אוגר \$s0 לאחר ביצוע הפקודות הבאות:

```
addi $t0, $0, 0x5678
sll $t1, $t0, 16
addi $t0, $t1, 0x1234
sw $t0, 4($0)
lb $s0, 6($0)
```

פתרון:

(שורה 1): תחילה נציב פשוט ל-\$t0 את הכתובת, באופן דיפולטיבי נכנס מימין והשאר אפסים משמאל:

```
0x00005678
```

(שורה 2): נזיז 16 ביטים שמאלה שזה אומר $4 \text{ bytes} = 16/4$ כלומר 4 מקומות שמאלה ונציב ב-\$t1.

```
0x56780000
```

(שורה 3): נוסיף ל-\$t0 את מה שיש ב-\$t1 ונצטרף/נדרוס מימין את 0x1234 ונקבל:

```
0x56781234
```

(שורה 4): שמרנו את הכתובת במיקום 4, לכן הוא המספר ההתחלתי.

(שורה 5): כדי לגשת ל-6 שזה 2 בתים משמאל לימין אז נקבל את:

```
0x00000012
```

ולכן \$s0 יכיל את 0x00000012.

2019, סמסטר ב, מועד א'

(אמריקאי) נניח שמחשב Mips הוא Little Endian, מה יהיה תוכן האוגר \$s0 לאחר ביצוע הפקודות הבאות:

```
lui $t0, 0x89AB
ori $t0, 0xCDEF
sw $t0, 0($0)
lb $s0, 2($0)
```

פתרון:

(שורה 1): נציב ב-\$t0 את upper-את הכתובת ונקבל: 0x89AB0000

(שורה 2): נציב ב-\$t0 מימין את הכתובת ונקבל: 0x89ABCDEF

(שורה 3): נשמור את ה-word של האוגר בכתובת המתחילה ב-0.

(שורה 4): מדובר ב-Little Endian ולכן צריך לגשת לבית 2 (מימין לשמאל כולל 0) ומדובר ב-0x89ABCDEF.

כלומר נבחר את הבית של AB ונרצה לאחסן אותו ב-\$s0, אבל צריך לבדוק קודם אם מדובר בשלילי או חיובי.

כיוון שמדובר ב-lb ולא ב-lbu נהיה חייבים לבדוק, נמיר את AB לבינארי ואם הספרה הראשונה משמאל היא 1

אז המספר שלילי (נלמד במערכות ספרתיות), AB בבינארי זה 1010 1011 יש 1 משמאל ולכן המספר שלילי.

לכן \$s0 יכיל את: 0xFFFFFAB (ה-FFF מייצג שליליות).

2019, סמסטר ב', מועד ב'

(אמריקאי) נניח שמחשב Mips הוא Big Endian, מה יהיה תוכן האוגר \$s0 לאחר ביצוע הפקודות הבאות:

```
lui $t0, 0x89AB
ori $t0, 0xCDEF
sw $t0, 0($0)
lb $s0, 2($0)
```

פתרון:

דומה מאוד לקודם אבל עם Big Endian. אז \$s0 יכיל את: **0xFFFFFCD**.

2020, סמסטר ב', מועד א'

(אמריקאי) נניח שמחשב Mips הוא Big Endian, מה יהיה תוכן האוגרים \$s0, \$s1 לאחר ביצוע הפקודות הבאות:

```
lui $t0, 0x5678
addi $t0, $t0, 0xabcd
sw $t0, 4($0)
lb $s0, 6($0)
lbu $s1, 6($0)
```

פתרון:

אוגר \$s0 יכיל **0xfffffab**, אוגר \$s1 יכיל **0x000000ab**

2020, סמסטר ב', מועד ב'

(אמריקאי) נניח שמחשב Mips הוא Little Endian, מה יהיה תוכן האוגרים \$s0, \$s1 לאחר ביצוע הפקודות הבאות:

```
lui $t0, 0x5678
addi $t0, $t0, 0xabcd
sw $t0, 4($0)
lb $s0, 6($0)
lbu $s1, 6($0)
```

פתרון:

אוגר \$s0 יכיל את - **0x00000078**, אוגר \$s1 יכיל את - **0x00000078**

2021, סמסטר ב', מועד א'

שאלה חוזרת מ-2018 מועד א' גרסה 2.

2021, סמסטר ב', מועד ב'

שאלה חוזרת מ-2018 מועד ב'.

תכנות ב- Mips אסמבלי (תרגילי מבחן)

2018, סמסטר ב', מועד א', גרסה 1

תרגיל:

איך תתורגם פקודת האסמבלי `add $s1, $s2, $s3` לשפת מכונה ?
(ה- opcode של `add` הוא 0 וה- `funct` הוא 32)

פתרון:

```
add s1, s2, s3
s2, s3, s1, add
0 18 19 17 shamt 32
000000 10010 10011 10001 00000 100000
0000 0010 0101 0011 1000 1000 0010 0000
0 2 5 3 8 8 2 0
0x02538820
```

תרגיל:

איך תתורגם פקודת האסמבלי: `addi $t1, $s2, -12` לשפת מכונה ?
(ה- opcode של `addi` הוא 8)

פתרון:

```
addi t1, s2, -12
addi s2, t1, -12
8, 18, 9, -12
001000 10010 01001 1111111111110100
0010 0010 0100 1001 1111 1111 1111 0100
2 2 4 9 f f f 4
0x2249fff4
```

תרגיל:

מהי פקודת האסמבלי `lw` שמתאימה לפקודת המכונה `0x8C0A0020` ?
(ה- opcode של `lw` הוא 35)

א. `lw $t2, 32($0)`

ב.

פתרון:

```
0x8C0A0020
100011 00000 01010 0000000000100000
35 0 10 32
lw $t2, 32($0)
```


תרגיל:**ביצוע הפקודות הבאות:**

```
addi $s1, $0, 8
nor $s1, $s1, $0
and $s0, $s0, $s1
```

פתרון:

סופי - הוא יגרום לאיפוס של ביט 3 באוגר \$s0.
 (שורה 1): מכניס את התוכן 8 לאוגר \$s1 כלומר יהיה לנו 1000 והמשך אפסים משמאל.
 (שורה 2): יהיה לנו 0111 והמשך אחדות משמאל.
 (שורה 3): יהיה לנו באוגר \$s0 פשוט מלא אפסים.
 איור דוגמה לפקודות לוגיות כדי לעזור לפתרון:

		Source Registers							
		\$s1	1111	1111	1111	1111	0000	0000	0000
		\$s2	0100	0110	1010	0001	1111	0000	1011
Assembly Code		Result							
and \$s3, \$s1, \$s2	\$s3	0100	0110	1010	0001	0000	0000	0000	0000
or \$s4, \$s1, \$s2	\$s4	1111	1111	1111	1111	1111	0000	1011	0111
xor \$s5, \$s1, \$s2	\$s5	1011	1001	0101	1110	1111	0000	1011	0111
nor \$s6, \$s1, \$s2	\$s6	0000	0000	0000	0000	0000	1111	0100	1000

תרגיל:**מה יהיה תוכן האוגר \$s1 לאחר ביצוע הפקודות הבאות:**

```
addi $s1, $0, 1
addi $s2, $0, 5
sllv $s1, $s1, $s2
```

פתרון:

(שורה 1): ב-\$s1 ישב 0001 בסוף
 (שורה 2): ב-\$s2 ישב 0101 בסוף
 (שורה 3): נבצע shift שמאלה ב-5 צעדים (תזכורת - כל צעד שמאלה זה להכפיל ב-2), כיוון שאנחנו זזים 5 צעדים שמאלה אז $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 32$ ולכן קיבלנו שהתוכן של \$s1 יהיה 32.

תרגיל:

לאילו פקודות MIPS עשוי הקומפיילר לתרגם את קטע הקוד הבא:

```
if (i == j)
    f = g + h;
else
    f = f - i;
```

פתרון:

(זה שאלה אמריקאית, יש כל מיני קטעי קוד צריך לבחור את הנכון).

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
    bne $s3, $s4, L1
    add $s0, $s1, $s2
    j   done
L1:   sub $s0, $s0, $s3
done:
```

תרגיל:

1. לאילו פקודות MIPS עשוי הקומפיילר לתרגם את קטע הקוד הבא:

```
int sum = 0;
int i;
for (i=1; i < 101; i = i*2) {
    sum = sum + i;
}
```

פתרון:

לגבי השורה של ה-loop, אם s0 קטן מ-t0 אז שים 1 ב-t1.
שורה אחרי, אם t1 שווה ל-0 אז תסים, אחרת תמשיך לרוץ...

```
# $s0 = i, $s1 = sum
    addi $s1, $0, 0
    addi $s0, $0, 1
    addi $t0, $0, 101
loop: slt  $t1, $s0, $t0
      beq  $t1, $0, done
      add  $s1, $s1, $s0
      sll  $s0, $s0, 1
      j    loop
done:
```

תרגיל:

איזו מהפונקציות הבאות שומרת אוגרים על המחסנית כפי שצריך ?

א.	ב.
<pre> proc1: addi \$sp, \$sp, -4 sw \$s0, 0(\$sp) add \$t0, \$a0, \$a1 add \$t1, \$a2, \$a3 sub \$s0, \$t0, \$t1 add \$v0, \$s0, \$0 lw \$s0, 0(\$sp) addi \$sp, \$sp, 4 jr \$ra </pre>	<pre> proc1: addi \$sp, \$sp, -8 sw \$s0, 0(\$sp) add \$t0, \$a0, \$a1 add \$t1, \$a2, \$a3 sub \$s0, \$t0, \$t1 add \$v0, \$s0, \$0 lw \$s0, 0(\$sp) addi \$sp, \$sp, 4 jr \$ra </pre>

פתרון:

התשובה היא (א) כיוון שבשורה הראשונה אנחנו מקצים מקום אחד במחסנית (כל מקום זה 4 בתים). לאחר מכן, בשורה לפני האחרונה עשינו דיאלוקציה (שחררנו זיכרון), כלומר בסוף השימוש מחקנו מהזיכרון את ההקצאה של ה-4 הבתים ולכן זה תקין. מצד שני ב-(ב) הקצנו 2 מקומות ובסוף שחררנו רק מקום אחד והשארנו מקום נוסף לא משוחרר וזה לא תקין.

תרגיל:

לאחר תרגום קטע הקוד הבא לשפת מכונה, מהם 16 הביטים שיהיו בשדה imm של הפקודה bne ?
(ראו דפי עזר, ציור 7)

0x40	loop:	add	\$t1, \$a0, \$s0
0x44		lb	\$t1, 0(\$t1)
0x48		add	\$t2, \$a1, \$s0
0x4C		sb	\$t1, 0(\$t2)
0x50		addi	\$s0, \$s0, 1
0x54		bne	\$t1, \$0, loop
0x58		lw	\$s0, 0(\$sp)

פתרון:

תזכורת- בארכיטקטורת MIPS, ה PC תמיד מתקדם לשורה הבאה, ואז מבצע קפיצה (אם צריך). לכן, לאחר ביצוע הפקודה bne \$1, \$0, loop ה PC יצביע על שורה 0x58. כעת, צריך לחזור לשורה 0x40, ולכן התווית loop צריכה להיות 0x18-, כלומר, מינוס 24.
 $-24 = 1111111111101000$
 אלא שכעת נזכור גם ששני הביטים הימניים נמחקים, ומשתמשים בהם מחדש בצד שמאל. לכן נקבל:
 1111111111101010

תרגיל:

בקטע הקוד הבא, מה יהיה תרגום הפקודה jal לשפת מכונה (ה- opcode של jal הוא 3) ?			
0x0040005C	jal	sum	
...			
0x004000A0	sum:	add	\$v0, \$a0, \$a1

פתרון:

0 0 4 0 0 0 A 0
 0000 0000 0100 0000 0000 1010 0000
 0000 1100 0001 0000 0000 0010 1000
 0 C 1 0 0 2 8
0x0C10028

תרגיל:

נניח שבית מספר 50 בזיכרון מכיל 0xFF, מה יהיה תוכן האוגר \$s0 לאחר ביצוע הפקודה הבאה:

lbu \$s0, 50(\$0)

פתרון:

ב-\$0 יהיה **0x000000FF**.

2018, סמסטר ב', מועד א', גרסה 2תרגיל:

איך תתורגם פקודת האסמבלי: `sw $t1, 100($t2)`
 לשפת מכונה? (ה- opcode של sw הוא 43)

פתרון:

```
sw t1, 100(t2)
sw t2, t1, 100
43 10 9 100
101011 01010 01001 0000000001100100
1010 1101 0100 1001 0000 0000 0110 0100
A D 4 9 0 0 6 4
0xAD490064
```

תרגיל:

מהי פקודת האסמבלי `addi` שמתאימה לפקודת המכונה `0x212AFF9C`?
 (ה- opcode של `addi` הוא 8)

פתרון:

```
0x212AFF9C
2 1 2 A F F 9 C
0010 0001 0010 1010 1111 1111 1001 1100
001000 01001 01010 111111110011100
8 9 10 -100
addi t1 t2 -100
addi $t2, $t1, -100
```

תרגיל:

בקטע הקוד הבא, מה יהיה תרגום הפקודה `j` לשפת מכונה (ה- opcode של `j` הוא 2)?

```
0x00400000      j      there
...
0x0040000C  there:
```

פתרון:

```
0x0040000C
0000 0000 0100 0000 0000 0000 0000 1100
0000 1000 0001 0000 0000 0000 0000 0011
0x08100003
```

תרגיל:**ביצוע הפקודות הבאות:**

```
addi $t1, $0, 32
or  $s1, $s1, $t1
```

פתרון:

הוא יגרום להדלקת ביט 5 של אוגר s1.

תרגיל:

לאילו פקודות MIPS עשוי הקומפיילר לתרגם את קטע הקוד הבא:
(ההבדלים בתשובות מודגשים)

```
int res = 1;
int i;
for (i = 10; i > 0; i--) {
    res = res * 8;
}
# $s0 = i, $s1 = res
```

פתרון:

נזכר כי sll כלומר shift שמאלה, בכל צעד שמאלה אנחנו נכפיל ב-2, כיוון שכאן יש 3 צעדים כלומר $2 \cdot 2 \cdot 2 = 8$ אז אכן אנחנו כל פעם "כופלים" ב-8.

```
addi $s1, $0, 1
addi $s0, $0, 10
loop: slt  $t0, $0, $s0
      beq  $t0, $0, done
      addi $s0, $s0, -1
      sll  $s1, $s1, 3
      j    loop
done:
```

תרגיל:

אילו אוגרים תצטרך הפונקציה הבאה לשמור על המחסנית ?

```
proc1:
    add $t1, $a1, $0
    add $s1, $a2, $0
    add $t3, $t1, $s1
    add $a0, $t3, $0
    jal proc2
    jr  $ra
```

פתרון:

הפונקציה תשמור על המחסנית את האוגר ra שמחזיק את כתובת החזרה מהפונקציה ואת האוגר s1 שנועד לשמור משתנים.

תרגיל:

לאחר תרגום קטע הקוד הבא לשפת מכונה, מהם 16 הביטים שיהיו בשדה imm של הפקודה beq ?
(ראו דפי עזר, ציור 8)

0x40 here:	add	\$t1, \$a0, \$s0
0x44	add	\$t2, \$a1, \$s0
0x48	addi	\$s0, \$s0, 1
0x4C	beq	\$s0, \$t1, here
0x50	sra	\$s0, \$t1, \$t2

פתרון:

כדי לתרגם את הקטע, בחלק של beq אנחנו צריכים להגיע ל 0x40 לכן אנחנו תמיד נתחיל לספור שורה אחת אחרי ה-beq בגלל ה-PC כלומר החל מ-0x50 ונשים לב שיש לנו 4 שורות לחזור אחורה כלומר $16 = 4 \cdot 4$ לכן אנחנו צריכים לחזור אחורה 16- (מינוס).
לכן נקבל 11111110000 כאשר משמאל יש עוד אחדות.
כעת נותר רק להוריד 2 ביטים מימין 111111100.
לכן התשובה היא **1111 1111 1111 1100**.

2018, סמסטר ב', מועד ב'תרגיל:

איך תתורגם פקודת האסמבלי: `lw $s1, 16($s2)` לשפת מכונה? (ה- opcode של lw הוא 35)

פתרון:

```
lw s1 16($s2)
lw s2 s1 16
35 18 17 16
100011 10010 10001 0000000000010000
1000 1110 0101 0001 0000 0000 0001 0000
8 E 5 1 0 0 1 0
0x8E510010
```

תרגיל:

מהי פקודת האסמבלי `ori` שמתאימה לפקודת המכונה `0x36510028`? (ה- opcode של ori הוא 13)

פתרון:

```
0x36510028
3 6 5 1 0 0 2 8
0011 0110 0101 0001 0000 0000 0010 1000
001101 10010 10001 0000000000101000
13 18 17 40
ori s2 s1 40
ori $s1, $s2, 40
```

תרגיל:

ביצוע הפקודות הבאות:			
<code>addi</code>	<code>\$t0,</code>	<code>\$0,</code>	<code>16</code>
<code>nor</code>	<code>\$t0,</code>	<code>\$t0,</code>	<code>\$0</code>
<code>and</code>	<code>\$t1,</code>	<code>\$t1,</code>	<code>\$t0</code>

פתרון:

יגרום לכיבוי ביט 4 של אוגר t1.

תרגיל:

לאילו פקודות MIPS עשוי הקומפיילר לתרגם את קטע הקוד הבא:
 המערך מתחיל בכתובת 0x12345678
 (ההבדלים בתשובות מודגשים)

```
int array[100];
int i;
for (i=0; i < 100; i++)
  array[i] = 0;
```

פתרון:

כל תא הוא 4. לכן צריך לעשות 2 הזזות בכל שלב.

```
loop:
  slt  $t0, $s1, $t2
  beq  $t0, $0, done
  sll  $s2, $s1, 2
  add  $s2, $s2, $s0
  sw   $0, 0($s2)
  addi $s1, $s1, 1
  j    loop
done:
```

תרגיל:

פקודת `sll $s1, $s1, ...` תורגמה לשפת מכונה כך:
 0x001188C0
 פי כמה הוכפל המספר שהיה באוגר \$s1 ?

פתרון:

0 0 1 1 8 8 C 0
 000000 00000 10001 10001 00011 000000

לפי R-Type נשים לב כי ה-shamt שלנו עם 00011 כלומר המספר 3.

לכן המספר שהיה באוגר s1 הוכפל $2 \cdot 2 \cdot 2$ כלומר פי 8.

תרגיל:

לאחר תרגום קטע הקוד הבא לשפת מכונה, מהם 16 הביטים שיהיו בשדה imm של הפקודה `beq` ?
 (ראו דפי עזר, ציור 8)

0x44	here:	add	\$t1, \$a0, \$s0
0x48		addi	\$s0, \$s0, 1
0x4C		beq	\$s0, \$t1, here
0x50		sra	\$s0, \$t1, \$t2

פתרון:

צריך לקפוץ 12- אחורה לכן 10100 (ומלא אפסים משמאל).

נוריד את 2 הביטים הימניים ביותר ונקבל: **1111 1111 1101**.

2019, סמסטר ב', מועד א'תרגיל:

מהי פקודת האסמבלי שתרגומה לשפת מכונה הוא:
0xAD310004

פתרון:

```

0xAD310004
A D 3 1 0 0 0 4
1010 1101 0011 0001 0000 0000 0100
101011 01001 10001 000000000100
43 9 17 4
sw t1, s1, 4
sw $s1, 4($t1)

```

תרגיל:

התכנית הבאה מחברת את המספרים 1 עד 10 ושומרת את התוצאה באוגר \$s1.
 מהם השורות החסרות ?

```

addi $s1, $0, 0
addi $s0, $0, 1
addi $t0, $0, 11
-----
-----
add  $s1, $s1, $s0
addi $s0, $s0, 1
j    loop
done:

```

פתרון:

```

loop: slt $t1, $s0, $t0
      beq $t1, $0, done

```

2019, סמסטר ב', מועד ב'תרגיל:

מהי פקודת האסמבלי שתרגומה לשפת מכונה הוא:
0x8D310004

פתרון:

0x8D310004
 8 D 3 1 0 0 0 4
 1000 1101 0011 0001 0000 0000 0100
 100011 01001 10001 000000000100
 35 9 17 4
 lw t1, s1, 4
lw \$s1, 4(\$t1)

2020, סמסטר ב', מועד א'תרגיל:

תרגמו את הפקודה הבאה להקסה-דצימאלי:

Sw \$t0, 32(\$s0) (opcode(sw) = 43)

פתרון:

sw t0, 32(s0)
 sw s0 t0 32
 43 16 8 32
 101011 10000 01000 00000000000100000
 1010 1110 0000 1000 0000 0000 0010 0000
0xAE080020

תרגיל:

נתון תרגום פקודה בהקסה-דצימאלי. החזירו אותה לשפת MIPS Assembly:

0x02ED2020 (funct(32) = add)

פתרון:

0x02ED2020
 0 2 E D 2 0 2 0
 0000 0010 1110 1101 0010 0000 0010 0000
 000000 10111 01101 00100 00000 100000
 23 13 4 32
 s7 t5 a0 add
add \$a0, \$s7, \$t5

תרגיל:

3. נתון הקוד הבא. תרגמו את פקודת הקפיצה loop j (opcode(j)=2).

```
0x04000000: goThere:
    add $s0, $s1, $s2
0x04000004: sub $t0, $t1, $t2
0x04000008: loop:
    slt $t5, $t0, $s0
0x0400000C: beq $t5, $0, exit
0x04000010: bne $t5, $0, goThere
0x04000014: exit:
    J loop
```

פתרון:

```
0x04000008
0000 0100 0000 0000 0000 0000 0000 1000
0000 1001 0000 0000 0000 0000 0000 0010
0x09000002
```

תרגיל:

נתון הקוד הבא. לאחר תרגום הקוד לבינארית, מהו הערך המספרי (immediate) הרשום בפקודת הקפיצה במקום התווית goThere?

```
0x04000000: goThere:
    add $s0, $s1, $s2
0x04000004: sub $t0, $t1, $t2
0x04000008: loop:
    slt $t5, $t0, $s0
0x0400000C: beq $t5, $0, exit
0x04000010: bne $t5, $0, goThere
0x04000014: exit:
    sll $t0, $t1, 3
```

פתרון:

אנחנו צריכים לקפוץ 5- אחורה כלומר מהשורה האחרונה לראשונה.
לכן הערך המספרי יהיה **1111 1111 1011**.

2020, סמסטר ב', מועד ב'תרגיל:

תרגמו את הפקודה הבאה להקסה-דצימאלי:

Slt \$t1, \$a3, \$t9 (func(slt) = 42)

פתרון:

```

slt $t1, $a3, $t9
$a3 $t1 $t9 slt
op 7 9 23 sham 42
000000 00111 01001 10111 00000 101010
0000 0000 1110 1001 1011 1000 0010 1010
0x00E9B82A

```

תרגיל:

נתון תרגום פקודה בהקסה-דצימאלי. החזירו אותה לשפת MIPS Assembly:

0xAE080020 (opcode(43) = sw)

פתרון:

```

0xAE080020
A E 0 8 0 0 2 0
1010 1110 0000 1000 0000 0000 0010 0000
101011 10000 01000 0000000000100000
43 16 8 32
sw s0 t0 32
sw $t0, 32($s0)

```

תרגיל:פתרון:

נתון הקוד הבא. תרגמו את פקודת הקפיצה exit (opcode(j)=2).

```

0x04000000: goThere:
    add $s0, $s1, $s2
0x04000004: sub $t0, $t1, $t2
0x04000008: loop:
    slt $t5, $t0, $s0
0x0400000C: beq $t5, $0, exit
0x04000010: bne $t5, $0, goThere
0x04000014: exit:
    J exit

```

```

0x04000014
0 4 0 0 0 0 1 4
0000 0100 0000 0000 0000 0000 0001 0100
0000 1001 0000 0000 0000 0000 0000 0101
0x09000005

```

תרגיל:

נתון הקוד הבא. מהו הערך המספרי (immediate) הרשום בקפיצה במקום התווית ?exit

```

0x04000000: goThere:
                add $s0, $s1, $s2
0x04000004:  sub $t0, $t1, $t2
0x04000008: loop:
                slt $t5, $t0, $s0
0x0400000C:  beq $t5, $0, exit
0x04000010:  bne $t5, $0, goThere
0x04000014: exit:
                slt $t0, $t1, 3

```

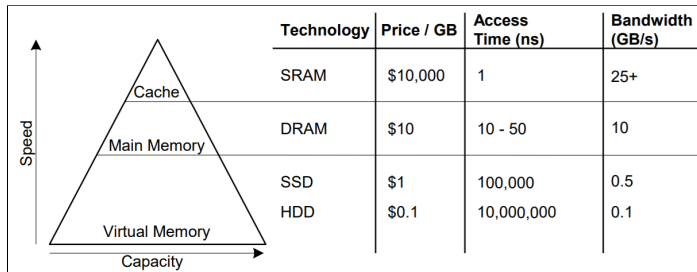
פתרון:

יש לנו רק קפיצה אחת כלומר יהיה רשום לנו 1 כלומר **0001 0000 0000 0000**.

2021, סמסטר ב', מועד א' - שאלה חוזרת

2021, סמסטר ב', מועד ב' - שאלה חוזרת

מטמון (הסבר)



היררכיית זיכרון

- קיימים שלושה גורמים שהיינו רוצים שיהיו ברכיבי זיכרון:
1. מהירות
 2. מקום רב
 3. זול

בפועל, אין רכיב זיכרון שמכיל את שלושתם - רק שניים.

סוגי זכרונות

- זיכרון ה-Cache - מאוד מהיר ומכיל מעט מקום. הבעיה היא שהוא מאוד יקר.
 - הזיכרון המרכזי (Main) - די מהיר, לא יקר, אבל לוקח קצת זמן לגשת אליו.
 - זיכרון וירטואלי - מאוד זול, לוקח הרבה מאוד זמן לגשת אליו והמהירות שלו לא גדולה.
- בפועל, אנחנו משתמשים בשלושתם. כשנחפש מידע כלשהו, ניגש תחילה אל ה-Cache ורק אם לא מוצאים נחפש בזיכרון המרכזי ואז בוורטואלי.

עקרונות הזיכרון

1. מקומיות בזמן - אם היה לאחרונה צורך לשלוף מידע מהזיכרון המרכזי/וירטואלי, כנראה שבקרוב נצטרך להמשיך להשתמש בו. לא נחזיר אותו למרכזי/וירטואלי אלא נשים אותו ב-Cache כדי שהגישה אליו תהיה זריזה.
2. מקומיות במקום - אם היה צורך לשלוף מידע מהזיכרון המרכזי/וירטואלי, כנראה שנצטרך בקרוב גם את מה שבאזור שלו. לכן, נשלוף לא רק אותו אלא גם קצת קדימה וקצת אחורה ונשים הכל ביחד ב-Cache.

מידת ביצועים

- אנחנו נרצה באידיאל שכל פעם, כל מידע שאנחנו צריכים ישב ב-Cache, ככה הזמן חיפוש שלנו יהיה קטן יותר.
- פגיעה Hit - מצאנו ב-Cache את המידע שחיפשנו.
 - פספוס Miss - חיפשנו בזיכרון כלשהו ולא מצאנו.
 - זמן ממוצע לגשת למידע AMAT מחושב על פי הנוסחה (צריך לזכור למבחן - אין בדף נוסחאות):

$$AMAT = T_{Cache} + MR_{Cache} \cdot [T_{MM} + MR_{MM} \cdot T_{VM}]$$

- ה-MR כלומר Missing Rate - אחוז הפספוס (כמות חיפושים שנכשלו חלקי כמות החיפושים).
- ה-MM כלומר Main Memory. ה-VM כלומר Virtual Memory. ה-T כלומר זמן חיפוש (cycles).

תרגיל דוגמה:

חשבו מהו ה-AMAT שבמערכת הבאה:

זמן החיפוש ב-Cache לוקח 1 cycle.

זמן החיפוש ב-MM לוקח 50 cycle.

זמן החיפוש ב-VM לוקח 500 cycle.

התוכנית ביצעה 3000 אחסונים ושליפות מהזיכרון, מתוכן 2300 היו ב-Cache ו-650 ב-MM והשאר ב-VM.

$$AMAT = T_{Cache} + MR_{Cache} \cdot [T_{MM} + MR_{MM} \cdot T_{VM}]$$

$$T_{Cache} = 1, \quad T_{MM} = 50, \quad T_{VM} = 500$$

$$MR_{Cache} = \frac{3000-2300}{3000} = \frac{700}{3000} = \frac{7}{30}$$

$$MR_{MM} = \frac{700-650}{700} = \frac{50}{700} = \frac{1}{14}$$

$$AMAT = 1 + \frac{7}{30} \cdot [50 + \frac{1}{14} \cdot 500] = 21$$

Cache-ה

- C - Capacity - כמה מידע אפשר להחזיק במטמון.
- b - Block size - כמה מידע מביאים למטמון בכל פעם.
- $\frac{c}{b}$ - כמות הבלוקים שיש בזיכרון.
- Set - כמות המחלקות שמחולק הזיכרון.
- N - כמה מידע שיהיה בכל קבוצה ב-Cache.

חלוקת הבלוקים לקבוצות

- Direct Map - כל קבוצה מכילה בלוק זיכרון יחיד. (N=1).
- N-Way - כל קבוצה מכילה N בלוקים.
- Fully Associative - יש קבוצה אחת וכל הבלוקים נמצאים שם.

Direct Map

- כל כתובת בזיכרון ממופה למקום מסוים ב-Cache.
- המיפוי הוא חח"ע אבל לא על.
- המיפוי מתחיל מה-LSB כאשר שני הביטים הימניים ביותר תמיד יהיו 00 למשל 01001000.
- ו-3 הביטים הבאים יתנו לנו את השייכות לקבוצה למשל 01001000 לכן שייך לקבוצה 2.
- ההחלטה לבחור ב-3 ביטים תלויה בגודל ה-Cache שיגדירו בשאלה למשל כאן הגודל הוא 8.
- אם למשל הגודל היה 16 אז היינו בוחרים את ה-4 ביטים הבאים כלומר 01001000.

תרגיל לדוגמה:

תנו שלוש כתובות שמתמפות לקבוצה 2, ושלוש כתובות שמתמפות לקבוצה 5 (בהנחה שיש מקום ל-8 words בזיכרון ה-Cache).

פתרון:

עבור קבוצה 2 (Set) נבחר:

0x00000008, 0x00000028, 0x000000E8

כיוון שאם נתרגם לבינארי את הכתובות, 3 הסיביות המסומנות הן בעצם המספר 2.

000...00**1000**, 000...001**01000**, 000...111**01000**

עבור קבוצה 5 (Set) נבחר:

0x00000014, 0x00000054, 0x000000F4

כיוון שאם נתרגם לבינארי את הכתובות, 3 הסיביות המסומנות הן בעצם המספר 5.

000...000**10100**, 000...010**10100**, 000...111**10100**

מבנה הזיכרון ב-Cache

V	Tag	Data	
			Set 7
			Set 6
			Set 5
			Set 4
			Set 3
			Set 2
			Set 1
			Set 0

- ה-Set זה הקבוצה, (מה שחיפשו בתרגיל הקודם).
 - ה-Data זה מה שמאוחסן שם.
 - ה-Tag מאמת לנו המיקום הנכון (ביטים של התיוג).
 - ה-V - האם התא בשימוש או לא.
- כשמחשב נדלק בפעם הראשונה, כל ה-Cache ריק לחלוטין ואז בכל העמודה של V יהיה 0.

- נשים לב: הגדרנו מקום ל-8 words, כלומר נצטרך **3 ביטים בשביל לתייג לקבוצה** (ה-Set).
- **שני ביטים ימניים תמיד יהיו 0**, זה משאיר **27 ביטים בשביל "תיוג" המידע ב-Cache** כדי שנדע איזה מידע יושב שם (ה-Tag).
- לדוגמה בכתובת הבאה:

0x**00000000000000000000000000000000****000000**

תרגיל לדוגמה:

כמה ביטים יש לקבוצה וכמה ביטים יש לתיוג ב-Cache עם 512 קבוצות.

פתרון:

נחשב $\log_2 512 = 9$ כלומר יש לנו **9 ביטים לקבוצות**, **2 ביטים תמיד 0**, ולכן נשארו **21 ביטים לתיוג**.

למשל:

0x**00000000000000000000000000000000****000000000000**

תרגיל:

הראו כמה פספוסים קרו ב-Cache מקטע הקוד הבא. הניחו כי ה-Cache מאופס בתחילת הריצה.

```
0x0: Addi $t0, $0, 4
0x4: Loop: beq $t0, $0, end
0x8: lw $t1, 0($0)      #bits: 0...00000000
0xc: lw $t2, 4($0)      #bits: 0...00001000
0x10: lw $t3, 8($0)     #bits: 0...00010000
0x14: lw $t4, 0xc($0)   #bits: 0...00011000
0x18: addi, $t0, $t0, -1
0x1c: j Loop
0x20: end:
```

פתרון:

ננתח את הקוד לפי מספר הכניסות ללולאה:

- בהתחלה ה-Cache מאופס, כלומר לא נתון בו כלום ולכן יש לנו $Miss = 4, Hit = 0$.
תמיד בהתחלה יהיה לנו Miss כיוון שאנחנו מאחסנים את הזיכרון ב-Cache והמשמעות של Hit היא השליפה מה-Cache אבל לא שלפנו כלום כי הוא היה ריק, רק אחסנו.
- יהיה לנו $Miss = 4, Hit = 4$ (כולל שלבים קודמים) כיוון שהפעם כן נשמר ב-Cache ארבעת הנתונים שהכנסנו.
- יהיה לנו $Miss = 4, Hit = 8$ (כולל שלבים קודמים).
- יהיה לנו $Miss = 4, Hit = 12$ (כולל שלבים קודמים).
לכן נקבל שאחוז הפספוס ב-Cache הוא:

$$MR_{Cache} = \frac{4}{12+4} = \frac{4}{16} = 25\%$$

תרגיל (כולל דריסה):

הראו כמה פספוסים קרו ב-Cache מקטע הקוד הבא. הניחו כי ה-Cache מאופס בתחילת הריצה.

```
0x0: Addi $t0, $0, 4
0x4: Loop: beq $t0, $0, end
0x8: lw $t1, 0($0)      #bits: 0...00000000
0xc: lw $t2, 0x20($0)   #bits: 0...01000000
0x10: addi, $t1, $t1, -1
0x14: j Loop
0x18: end:
```

פתרון:

ננתח את הקוד לפי מספר הכניסות ללולאה:

- יהיה לנו $Miss = 2, Hit = 0$. אך נשים לב כי הכנסנו את הירוק קודם ואז דרסנו אותו עם הכחול.
- יהיה לנו כבר $Miss = 4, Hit = 0$. כיוון שהירוק רוצה לשלוף מה-Cache אבל הכחול דרס אותו באיטרציה הקודמת ולכן הוא פספס ולא משך מה-Cache אלא דרס את הכחול. עכשיו הכחול מבקש למשוך אבל הירוק דרס אותו ולכן גם פספס.
- $Miss = 6, Hit = 0$.
- $Miss = 8, Hit = 0$.
לכן נקבל שאחוז הפספוס ב-Cache הוא:

$$MR_{Cache} = \frac{8}{8+0} = \frac{8}{8} = 100\%$$

יש לנו כאן 100% פספוס כיוון שאנחנו כל הזמן דורסים את האחר.

N-Way Set

- כל תא בזיכרון ה-Cache מכיל כעת מקום ל-N מילים (Words) במקום ל-Word בודד.
- עדיין, כל כתובת בזיכרון מתמפה לתא יחיד ב-Cache, אלא שעכשיו התא היחיד מכיל כמה מילים.
- זה הפתרון לבעיית הדריסה כפי שהוצג בתרגיל הקודם.

עיקרון ה-Locality

- עובד לפי N-Way.
- אנחנו נגדיר גישה שונה ל-Cache ממה שהוצג עד עכשיו.
- לכל כתובת יהיה 0 ביטים מימין שלא רלוונטים, 2 ביטים אחרי שהם ה-Offset ו-1 ביט שהוא ה-Set.
- היתרון כאן הוא שכאשר נרצה לאחסן ב-Cache, אנחנו נאחסן גם את שאר ה-Words שנמצאים בסביבת ה-Word שאיחסנו וככה אנחנו בעצם נמצמם את ה-Miss שלנו.
- למשל, ללא העיקרון, אם היינו מאחסנים 4 words אז היינו צריכים לאחסן כל אחד מהם ב-Cache וזה עולה לנו ב-4 Miss אבל כעת מספיק ואחסנו word כלשהו והוא כבר יאחסן את ה-"שכנים" שמסביבו מראש וככה אנחנו נקטין את הנזק כי לא נצטרך לאחסן אחר כך את השאר.

```

Addi $t0, $0, 3
Addi $t1, $0, 3
Loop1: Beq $t0, $0, Next
        Lw $t2, 0($0)      #0...00000000 – Set 0
        Lw $t3, 4($0)      #0...00000100 – Set 0
        Lw $t4, 8($0)      #0...00001000 – Set 0
        Lw $t5, 0xc($0)    #0...00001100 – Set 0
        Lw $t6, 0x10($0)   #0...00010000 – Set 1
        Lw $t7, 0x14($0)   #0...00010100 – Set 1
        Addi $t0, $t0, -1
        J Loop1

Next:   Beq $t1, $0, End
        Lw $t2, 0x20($0)   #0...00100000 – Set 0
        Lw $t3, 0x24($0)   #0...00100100 – Set 0
        Lw $t4, 0x28($0)   #0...00101000 – Set 0
        Lw $t5, 0x2c($0)   #0...00101100 – Set 0
        Addi $t1, $t1, -1
        J Next

End:

```

MR

דוגמה:

נתבונן בקוד הבא:

הלולאה Loop1 רצה את כל הצעדים שלה וכשמסיימת היא שולחת ללולאה אחרת בשם Next. (האדום זה ה-Offset והירוק זה ה-Set).

לכן בהתחלה ב-Loop1 נאחסן:

V	Tag	11	10	01	00	Set
1	0...0	[0xc]	[0x18]	[0x14]	[0x10]	Set 1
1	0...0	[0xc]	[0x8]	[0x4]	[0x0]	Set 0

וכעת כשניגש ל-Next ונאחסן (נדרוס) את הכתובות. נשים לב כי אמנם דרסנו אבל אחסנו את כולם במכה אחת כי לכולם יש את אותו ה-Set שהוא 0:

V	Tag	11	10	01	00	Set
1	0...0	[0x1c]	[0x18]	[0x14]	[0x10]	Set 1
1	0...1	[0x2c]	[0x28]	[0x24]	[0x20]	Set 0

לכן יהיה לנו בסה"כ 3 Miss לאורך כל התוכנית. למה? -

1. אחסנו את כל קבוצה 0 במכה אחת בתחילת התוכנית בתוך ה-Loop1.
2. אחסנו את כל קבוצה 1 במכה אחת בתחילת התוכנית בתוך ה-Loop1.
3. אחסנו/דרסנו את כל קבוצה 0 במכה אחת כשניגשנו לפונקציה Next.

חשבו לאן ממופות הכתובות הבאות:

פתרון:

נתרגם לבינארי את 0x00000054 ונקבל:

כלומר **Offset-ה** הוא יהיה 01 **Set-ה** יהיה קבוצה 1.

נתרגם לבינארי את 0x000000F8 ונקבל:

כלומר **Offset-ה** הוא יהיה 10 **Set-ה** יהיה קבוצה 1.

- כאשר ה-Cache מלא, השליפה הבאה שאינה מה-Cache תכניס מידע חדש ל-Cache ותמחק ממנו מידע קיים (כי הוא מלא). כשנצטרך לגשת שוב למידע שהוצאנו/דרסנו יהיה פספוס.
- זה נקרא Capacity Miss.
- נמחק את המידע שאנחנו חייבים לפנות כדי להכניס מידע חדש.
- ב-Least Used Recently - נעיף את ה-Block שהכי לא משומש מהקבוצה הדרושה.
- נקצה ביט של Used כדי שיגידו איזה מידע היה בשימוש לאחרונה.

נתון מטמון בגודל 32 מילים (מילה - 4 בתים), גודל כל בלוק מילה אחת, בכל קבוצה (סט) ארבעה בלוקים. לאיזה קבוצה (סט) במטמון תמופה הכתובת 0x00001234?

נקבע פרמטרים כתבנית לפתרון שאלות מסוג זה:

- גדול המטמון נסמן ב-**C** מלשון Capacity כלומר $C = 32$.
- גודל הבלוק היא מספר המילים בכל בלוק, נסמן ב-**B** מלשון Block כלומר $B = 1$.
- גודל הקבוצה (סט) הוא מספר הבלוקים, נסמן ב-**U** (מלשון Unit) כלומר $U = 4$.
- כדי לדעת כמה קבוצות יש לנו בסה"כ (נסמן ב-**S** מלשון Sets) נשתמש בנוסחה הבאה:

כלומר במקרה שלנו פשוט נציב ונקבל: $32 = 1 \cdot 4 \cdot S$ נחלק ב-4 ונקבל $S = 8$.

- כדי לדעת מהו ה-Offset נחשב לפי הנוסחה: $\log_2(B)$ ובמקרה שלנו $\log_2(1) = 0$.
- כדי לדעת כמה סיביות ייצגו את הקבוצות נחשב בנוסחה $\log_2(S)$ כלומר $\log_2(8) = 3$.
- נתרגם לבינארי את הכתובת ונבחר את הקבוצה (3 סיביות מימין לא כולל ה-2 הראשונות):

- לכן הכתובת תמופה במטמון לקבוצה מספר **101** כלומר קבוצה **5**.

מטמון (תרגילי מבחן)

2018, סמסטר ב', מועד א', גרסה 1

תרגיל:

תכנית מכילה 2000 פקודות שקוראות או כותבות נתונים לזיכרון, מתוכם 1250 נתונים נמצאים במטמון. זמן הקריאה מהזיכרון הוא 100 מחזורי שעון, זמן הקריאה מהמטמון הוא מחזור אחד. כמה מחזורי שעון בממוצע נמשכת קריאה או כתיבה של נתון.

פתרון:

נזכר תחילה בנוסחה:

$$AMAT = T_{Cache} + MR_{Cache} \cdot [T_{MM} + MR_{MM} \cdot T_{VM}]$$

נתאים את הנתונים לכל משתנה:

$$T_{Cache} = 1, T_{MM} = 100$$

$$MR_{Cache} = \frac{2000-1250}{2000} = \frac{750}{2000} = \frac{3}{8}$$

$$AMAT = 1 + \frac{3}{8} \cdot [100 + 0] = 1 + \frac{300}{8} = 1 + 37.5 = 38.5$$

תרגיל:

במטמון ממופה ישירות (Direct Mapped Cache) ישנם 8 סטים בני בלוק אחד, כאשר כל בלוק הוא מלה אחת. מה יהיה אחוז הפגיעה בבצוע התכנית הבאה:

```
addi $t0, $0, 5
loop: beq $t0, $0, done
      lw  $t1, 0x4($0)
      lw  $t2, 0xC($0)
      lw  $t3, 0x8($0)
      addi $t0, $t0, -1
      j   loop
done:
```

פתרון:

ננתח לפי שלבים בלולאה:

1. Miss = 3, Hits = 0

2. Miss = 3, Hits = 3

3. Miss = 3, Hits = 6

4. Miss = 3, Hits = 9

5. Miss = 3, Hits = 12

לכן נקבל: $HR_{Cache} = \frac{12}{15} = 0.8 = 80\%$

תרגיל:

נתון מטמון בגודל 8 מילים (מילה = 4 בתים), גודל כל בלוק מילה אחת, בכל סט 2 בלוקים. לאיזה מקום במטמון תמופה הכתובת 0×24 ?

פתרון:

נסמן את הפרמטרים: $C = 8, B = 1, U = 2 \parallel 8 = 1 \cdot 2 \cdot S \Rightarrow S = 4$
מבחינת Offset נחשב $\log_2(B) = \log_2(1) = 0$ כלומר אין.

מבחינת סיביות קבוצה נחשב $\log_2(S) = \log_2(4) = 2$ כלומר 2 סיביות.

ננתח את הכתובת:

0x00000024

נתרגם לבינארי:

0000 0000 0000 0000 0000 0000 0010 **0100**

לכן, 2 הביטים הראשונים אנחנו נתעלם מהם, ו-2 הביטים הבאים אחריהם ייצגו את המקום במקום בו הכתובת תמופה (כלומר קבוצה/סט מספר **1**).

2018, סמסטר ב, מועד א, גרסה 2תרגיל:

תכנית מכילה 4000 פקודות שקוראות או כותבות נתונים לזיכרון, מתוכם 3500 נתונים נמצאים במטמון. זמן הקריאה מהזיכרון הוא 24 מחזורי שעון, זמן הקריאה מהמטמון הוא 2 מחזורי שעון. כמה מחזורי שעון בממוצע נמשכת קריאה או כתיבה של נתון.

פתרון:

נזכר תחילה בנוסחה:

$$AMAT = T_{Cache} + MR_{Cache} \cdot [T_{MM} + MR_{MM} \cdot T_{VM}]$$

נתאים את הנתונים לכל משתנה:

$$T_{Cache} = 2, T_{MM} = 24$$

$$MR_{Cache} = \frac{4000-3500}{4000} = \frac{500}{4000} = \frac{1}{8}$$

נציב ונחשב:

$$AMAT = 2 + \frac{1}{8} \cdot [24 + 0] = 2 + \frac{24}{8} = 5$$

תרגיל:

נתון מטמון בגודל 64 מילים (מילה = 4 בתים), גודל כל בלוק מילה אחת, בכל סט 4 בלוקים. לאיזה מקום במטמון תמופה הכתובת 0x1234?

פתרון:

נסמן פרמטרים ונחשב: $C = 64, B = 1, U = 4 \parallel 64 = 1 \cdot 4 \cdot S \Rightarrow S = 16$
מבחינת Offset נחשב: $\log_2(1) = 0$ כלומר לא קיים כאן.

מבחינת מספר סיביות לקבוצות נחשב: $\log_2(16) = 4$ כלומר 4 סיביות.

ננתח את הכתובת:

0x00001234

נמיר לבינארי:

0000 0000 0000 0000 0001 0010 0011 0100

לכן, 2 הסיביות הראשונות מימין לא רלוונטיות, ו-4 הסיביות הבאות מייצגות את הקבוצה בה תמופה הכתובת במטמון ולכן הכתובת תמופה במיקום **13**.

תרגיל:

במטמון ממופה ישירות (Direct Mapped Cache) ישנם 8 קבוצות (סטים), בכל קבוצה (סט) יש בלוק אחד, כל בלוק מכיל מילה אחת. מה יהיה אחוז הפגיעה בבצוע התכנית הבאה:

```
addi $t0, $0, 20
loop: beq $t0, $0, done
      lw  $t1, 0x40($0)
      lw  $t1, 0x44($0)
      lw  $t1, 0x4C($0)
      lw  $t2, 0x5C($0)
      lw  $t3, 0x64($0)
      addi $t0, $t0, -1
      j   loop
done:
```

פתרון:

תחילה חשוב לשים לב כי יש כאן דריסה בין 2 כתובות זיכרון - חשוב מאוד לבדוק את זה מראש!.

הכתובת 0x44 ובינארי הוא **1000100**.

הכתובת 0x64 ובינארי הוא **1100100**.

יש כאן התנגשות עם 2 כתובות ולכן נפסוג מכה קשה של פספוסים.

הלולאה תרוץ 20 פעמים, בשלב הראשון נספוג 5 Miss בגלל אחסון (כולל הדריסה).

ובכל שאר השלבים אנחנו נספוג תמיד 2 Miss נוספים ו-3 Hits.

לכן בסופו של יום נקבל: $Miss = 5 + 19 \cdot 2 = 43$, $Hits = 19 \cdot 3 = 57$

ונקבל כי אחוז הפגיעה יהיה:

$$HR_{Cache} = \frac{57}{43+57} = \frac{57}{100} = 0.57 = 57\%$$

2018, סמסטר ב', מועד ב'תרגיל:

תכנית מכילה 1000 פקודות שקוראות או כותבות נתונים לזיכרון, מתוכם 900 נתונים נמצאים במטמון. זמן הקריאה מהזיכרון הוא 10 מחזורי שעון, זמן הקריאה מהמטמון הוא 1 מחזורי שעון. כמה מחזורי שעון בממוצע נמשכת קריאה או כתיבה של נתון.

פתרון:

נזכר תחילה בנוסחה:

$$AMAT = T_{Cache} + MR_{Cache} \cdot [T_{MM} + MR_{MM} \cdot T_{VM}]$$

נתאים את הנתונים לכל משתנה:

$$T_{Cache} = 1, T_{MM} = 10$$

$$MR_{Cache} = \frac{1000-900}{1000} = \frac{100}{1000} = \frac{1}{10}$$

נציב ונחשב:

$$AMAT = 1 + \frac{1}{10} \cdot [10 + 0] = 1 + \frac{10}{10} = 1 + 1 = 2$$

תרגיל:

נתון מטמון בגודל 64 מילים (מילה = 4 בתים), גודל כל בלוק 4 מילים, בכל סט בלוק אחד. לאיזה מקום במטמון תמופה הכתובת 0x1234?

פתרון:

נציב פרמטרים ונחשב: $C = 64, B = 4, U = 1 \parallel 64 = 4 \cdot 1 \cdot S \Rightarrow S = 16$

נחשב Offset ונקבל: $\log_2(B) = \log_2(4) = 2$ כלומר יש לנו 2 סיביות עבור Offset.

נחשב סיביות מייצגות קבוצה: $\log_2(S) = \log_2(16) = 4$ כלומר יש לנו 4 סיביות לייצוג קבוצות.

ננתח את הכתובת:

0x00001234

נמיר לבינארי:

0000 0000 0000 0000 0001 0010 0011 0100

ה-Offset יהיה ה-2 סיביות הראשונות וה-4 סיביות שנותרו יהיו עבור ייצוג הקבוצה שאליה תמופה הכתובת במטמון.

0000 0000 0000 0000 0001 0010 0011 0100

לכן הכתובת תמופה במטמון לקבוצה מספר 3.

תרגיל:

במטמון ממופה ישירות (Direct Mapped Cache) ישנם 16 קבוצות (סטים), בכל קבוצה (סט) יש בלוק אחד, כל בלוק מכיל מלה אחת. מה יהיה אחוז הפגיעה בבצוע התכנית הבאה:

```

addi $t0, $0, 20
loop: beq $t0, $0, done
      lw  $t1, 0x64($0)
      lw  $t1, 0x74($0)
      lw  $t1, 0xA4($0)
      lw  $t2, 0x84($0)
      lw  $t3, 0xE4($0)
      addi $t0, $t0, -1
      j   loop
done:

```

פתרון:

תחילה נשים לב כי מדובר במטמון בגודל 16 ולכן צריך להתייחס ל-4 הסיביות מימין המייצגות את הקבוצות. שנית, נבדוק האם קיימים התנגשויות בין הכתובות. אכן נמצא מקום בו יש 3 כתובות ולכן יהיו כאן דריסות.

הכתובת 0x64 ובבינארי 1001001.

הכתובת 0xA4 ובבינארי 10100100.

הכתובת 0xE4 ובבינארי 11100100.

הלולאה רצה 20 פעמים, ננתח את הנזק:

בשלב הראשון ה-Cache מאופס ולכן נכניס 5 כתובות (2 מתוכם נדרסות) כלומר $Miss = 5, Hits = 0$.

בשלב השני כתובת 0x64 תבקש למשוך אבל היא נדרסה השאר וכן הלאה ל-2 האחרות.

לכן בכל שלב חוץ מהראשון נספוג מכה של 3 פספוסים בגלל הדריסות וגם 2 פגיעות נקיות.

ולכן בסוף הריצה נקבל:

$$Miss = 5 + 3 \cdot 19 = 62, Hits = 2 \cdot 19 = 38$$

ולכן אחוז הפגיעה יהיה:

$$HR_{Cache} = \frac{38}{62+38} = \frac{38}{100} = 0.38 = 38\%$$

2019, סמסטר ב, מועד א'תרגיל:

נתון מטמון בגודל 32 מילים (מילה = 4 בתים), גודל כל בלוק מילה אחת, בכל סט 2 בלוקים. לאיזה קבוצה במטמון תמופה הכתובת 0xABC?

פתרון:

נציב ונחשב: $C = 32, B = 1, U = 2 \parallel 32 = 1 \cdot 2 \cdot S \Rightarrow S = 16$
נבדוק Offset נחשב: $\log_2(B) = \log_2(1) = 0$. Offset כלומר אין.

נחשב מספר סיביות קבוצה: $\log_2(S) = \log_2(16) = 4$ סיביות עבור סימוני קבוצה.

ננתח את הכתובת: 0x00000ABC

נמיר לבינארי:

0000 0000 0000 0000 0000 1010 10**11 1100**

לכן הכתובת תמופה לקבוצה 1111 כלומר קבוצה **F**.

תרגיל:

במטמון ממופה ישירות (Direct Mapped Cache) ישנם 8 קבוצות (סטים), בכל קבוצה (סט) יש בלוק אחד, כל בלוק מכיל מלה אחת. מה יהיה אחוז הפגיעה בבצוע התכנית הבאה:

```
addi $t0, $0, 25
loop: beq $t0, $0, done
lw $t1, 0x40($0)
lw $t1, 0x84($0)
lw $t2, 0x4C($0)
lw $t3, 0x8C($0)
addi $t0, $t0, -1
j loop
done:
```

פתרון:

מאחר ומדובר ב-8 קבוצות אז מספר הסיביות המייצגות קבוצה יהיו 3 סיביות.

נבדוק האם יש התנגשויות במיפוי.

הכתובת 0x4C ובבינארי 100**1100**.

הכתובת 0x8C ובבינארי 1000**1100**.

ננתח את הפגיעות לפי הלולאה שרצה 25 פעמים:

בשלב הראשון ה-Cache מאופס ולכן נאחסן 4 כתובות (2 מתנגשות) ונקבל $Miss = 4, Hits = 0$.

בכל שאר השלבים נקבל תמיד 2 פספוסים ו-2 פגיעות.

לכן בסוף הריצה נגיע למצב הבא:

$$Miss = 4 + 2 \cdot 24 = 4 + 48 = 52, Hits = 24 \cdot 2 = 48$$

ולכן נקבל שאחוז הפגיעה יהיה:

$$HR_{Cache} = \frac{48}{52+48} = \frac{48}{100} = 0.48 = 48\%$$

2019, סמסטר ב, מועד ב'תרגיל:

נתון מטמון בגודל 128 מילים (מילה = 4 בתים), גודל כל בלוק מילה אחת, בכל סט 8 בלוקים. לאיזה קבוצה במטמון תמופה הכתובת $0x5678$?

פתרון:

נציב פרמטרים ונחשב: $C = 128, B = 1, U = 8 \parallel 128 = 1 \cdot 8 \cdot S \Rightarrow S = 16$
 נבדוק Offset נחשב: $\log_2(B) = \log_2(1) = 0$ כלומר אין.
 נבדוק מספר סיביות עבור ייצוגי קבוצות: $\log_2(S) = \log_2(16) = 4$ כלומר **4** סיביות.
 ננתח את הכתובת:

0x00005678

נמיר לבינארי:

0000 0000 0000 0000 0101 0110 01**11 1000**

הכתובת תמופה לקבוצה מספר **14** כלומר **E**.

תרגיל:

במטמון ממופה ישירות (Direct Mapped Cache) ישנם 8 קבוצות (סטים), בכל קבוצה (סט) יש בלוק אחד, כל בלוק מכיל מלה אחת. מה יהיה אחוז הפגיעה בבצוע התכנית הבאה:

```
addi $t0, $0, 20
loop: beq $t0, $0, done
      lw  $t1, 0x40($0)
      lw  $t1, 0x44($0)
      lw  $t1, 0x64($0)
      lw  $t2, 0x4C($0)
      lw  $t3, 0x88($0)
      addi $t0, $t0, -1
      j   loop
done:
```

פתרון:

יש 2 כתובות שמתנגשות במיפוי 0x64 ו-0x44.
 הלולאה רצה 20 פעמים, בשלב הראשון ה-Cache מאופס ולכן נאחסן 5 כתובות ונספוג 5 פספוסים.
 בכל שלב אחר נספוג 2 פספוסים ו-3 פגיעות.
 בסוף הריצה נקבל:

$$Miss = 5 + 2 \cdot 19 = 43, Hits = 3 \cdot 19 = 57$$

ולכן אחוז הפגיעה יהיה:

$$HR_{Cache} = \frac{57}{43+57} = \frac{57}{100} = 0.57 = 57\%$$

2020, סמסטר ב', מועד א'תרגיל:

תכנית מכילה 1500 פקודות שקוראות או כותבות נתונים לזיכרון, מתוכם 1200 נתונים נמצאים במטמון. זמן הקריאה מהזיכרון הוא 60 מחזורי שעון, זמן הקריאה מהמטמון הוא 2 מחזורי שעון. כמה מחזורי שעון בממוצע נמשכת קריאה או כתיבה של נתון.

פתרון:

נזכר תחילה בנוסחה:

$$AMAT = T_{Cache} + MR_{Cache} \cdot [T_{MM} + MR_{MM} \cdot T_{VM}]$$

נתאים את הנתונים לכל משתנה:

$$T_{Cache} = 2, T_{MM} = 60$$

$$MR_{Cache} = \frac{1500-1200}{1500} = \frac{300}{1500} = \frac{1}{5}$$

נציב ונחשב:

$$AMAT = 2 + \frac{1}{5} \cdot [60 + 0] = 2 + \frac{60}{5} = 2 + 12 = 14$$

תרגיל:

נתון מטמון בגודל 32 מילים (מילה = 4 בתים), גודל כל בלוק 2 מילים, בכל סט בלוק אחד. לאיזה קבוצה במטמון תמופה הכתובת 0x1234?

פתרון:

נציב פרמטרים ונחשב: $C = 32, B = 2, U = 1 \mid 32 = 2 \cdot 1 \cdot S \Rightarrow S = 16$.
נבדוק Offset נחשב: $\log_2(B) = \log_2(2) = 1$.
נחשב מספר סיביות עבור ייצוג הקבוצה ונקבל: $\log_2(S) = \log_2(16) = 4$.
ננתח את הכתובת:

0x00001234

נמיר לבינארי:

0000 0000 0000 0000 0001 0010 0011 0100

בכל בלוק יש 2 מילים לכן מספר **סיב אחד** עבור **Offset** ושאר הסיבים עבור ייצוג הקבוצה (**4 סיביות**).
לכן הכתובת תמופה במטמון לקבוצה מספר **0110** כלומר קבוצה **6**.

תרגיל:

במטמון ממופה ישירות (Direct Mapped Cache) ישנם 8 קבוצות (סטים), בכל קבוצה (סט) יש בלוק אחד, כל בלוק מכיל מלה אחת. מה יהיה אחוז הפגיעה בביצוע התוכנית הבאה:

```
addi $t0, $0, 25
loop:
    beq $t0, $0, done
    lw $t1, 0x0064($0)
    lw $t2, 0x00A4($0)
    lw $t3, 0x008C($0)
    lw $t4, 0x00E4($0)
    addi $t0, $t0, -1
    J loop
done:
```

פתרון:

יש 3 כתובות שמתנגשות במיפוי והן: 0x64, 0xA4, 0xE4. הלולאה רצה 25 פעמים, בפעם הראשונה אנו מאחסנים 4 כתובות ל-Cache ולכן נקבל 4 פספוסים. בכל שלב אחר נקבל 3 פספוסים ופגיעה אחת עד סוף הריצה. בסוף הריצה נגיע למצב הבא:

$$Miss = 4 + 24 \cdot 3 = 76, Hits = 24 \cdot 1 = 24$$

אחוז הפגיעה יהיה:

$$HR_{Cache} = \frac{24}{76+24} = \frac{24}{100} = 0.24 = 24\%$$

2020, סמסטר ב', מועד ב'תרגיל:

תכנית מכילה 2500 פקודות שקוראות או כותבות נתונים לזיכרון, מתוכם 2250 נתונים נמצאים במטמון. זמן הקריאה מהזיכרון הוא 100 מחזורי שעון, זמן הקריאה מהמטמון הוא 2 מחזורי שעון. כמה מחזורי שעון בממוצע נמשכת קריאה או כתיבה של נתון.

פתרון:

נזכר תחילה בנוסחה:

$$AMAT = T_{Cache} + MR_{Cache} \cdot [T_{MM} + MR_{MM} \cdot T_{VM}]$$

נתאים את הנתונים לכל משתנה:

$$T_{Cache} = 2, T_{MM} = 100$$

$$MR_{Cache} = \frac{2500-2250}{2500} = \frac{250}{2500} = \frac{1}{10}$$

נציב ונקבל:

$$AMAT = 2 + \frac{1}{10} \cdot [100 + 0] = 2 + \frac{100}{10} = 12$$

תרגיל:

נתון מטמון בגודל 32 מילים (מילה = 4 בתים), גודל כל בלוק מילה אחת, בכל סט ארבע בלוקים. לאיזה קבוצה במטמון תמופה הכתובת $0x1234$?

פתרון:

נציב פרמטרים ונחשב: $C = 32, B = 1, U = 4 \parallel 32 = 1 \cdot 4 \cdot S \Rightarrow S = 8$.
נבדוק Offset נחשב: $\log_2(B) = \log_2(1) = 0$ כלומר אין.

נחשב מספר סיביות עבור ייצוג הקבוצות: $\log_2(S) = \log_2(8) = 3$ כלומר 3 סיביות.

ננתח את הכתובת:

0x00001234

נמיר לבינארי:

0000 0000 0000 0000 0001 0010 0011 **0100**

לכן הכתובת תמופה במטמון לקבוצה מספר 5.

תרגיל:

שאלה 13 - במטמון ממופה ישירות (Direct Mapped Cache) ישנם 8 קבוצות (סטים), בכל קבוצה (סט) יש בלוק אחד, כל בלוק מכיל מילה אחת. מה יהיה אחוז הפגיעה בביצוע התוכנית הבאה:

```
addi $t0, $0, 20
loop:
    beq $t0, $0, done
    lw $t1, 0x0064($0)
    lw $t2, 0x000C($0)
    lw $t3, 0x008C($0)
    lw $t4, 0x00E8($0)
    lw $t5, 0x0040($0)
    addi $t0, $t0, -1
    J loop
done:
```

פתרון:

יש 2 כתובות שמתנגשות במיפוי והן: $0xC$ וגם $0x8C$.
באיטריציה הראשונה נאחסן את כל הכתובות ונקבל 5 פספוסים.
בכל שלב אחר נקבל עוד 2 פספוסים ו-3 פגיעות.
לכן בסוף הריצה נקבל:

$$Miss = 5 + 19 \cdot 2 = 43, Hits = 3 \cdot 19 = 57$$

ולכן אחוז הפגיעה יהיה:

$$HR_{Cache} = \frac{57}{100} = 0.57 = 57\%$$

2021, סמסטר ב', מועד א'תרגיל:

שאלה חוזרת ממועד 2020, סמסטר ב', מועד א'.

תרגיל:

תכנית מכילה 1000 פקודות שקוראות או כותבות נתונים לזיכרון, מתוכם 900 נתונים נמצאים במטמון. זמן הקריאה מהזיכרון הוא 10 מחזורי שעות, זמן הקריאה מהמטמון הוא 1 מחזורי שעות. כמה מחזורי שעות בממוצע נמשכת קריאה או כתיבה של נתון.

פתרון:

נזכר תחילה בנוסחה:

$$AMAT = T_{Cache} + MR_{Cache} \cdot [T_{MM} + MR_{MM} \cdot T_{VM}]$$

נתאים את הנתונים לכל משתנה:

$$T_{Cache} = 1, \quad T_{MM} = 10$$

$$MR_{Cache} = \frac{1000-900}{1000} = \frac{100}{1000} = \frac{1}{10}$$

נציב ונחשב:

$$AMAT = 1 + \frac{1}{10} \cdot [10 + 0] = 1 + \frac{10}{10} = 1 + 1 = 2$$

2021, סמסטר ב', מועד ב'תרגיל:

שאלה חוזרת ממועד 2021, סמסטר ב', מועד א'.

תרגיל:

שאלה חוזרת ממועד 2018, סמסטר ב', מועד ב'.

המרות במחשבון:

1. לוחצים על Menu
2. לוחצים על 3
3. אם נרצה להמיר מ-Dec ל-Bin פשוט נכניס את המספר ב-Dec ואז "=" ואז ללחוץ על Bin ונקבל את התשובה.
4. כנ"ל לגבי המרות בין Hex ל-Bin וכו'.

חישוב מודולו במחשבון:

1. לוחצים על Menu
2. לוחצים על 3
3. במצב DEC בשביל לחשב $A \bmod B$ נחשב באופן הבא לפי הכפתורים $A - A \div B * B$.

- חלאת. -