

תוכן העניינים:

2	מבנה המחשב ותכן לוגי
2	האסמבלי של מעבדי ה-MIPS
2	הקדמה כללית:
2	סיכום כללי:
5	יסודות ה-ISA:
8	פקודות מסוג R-Type:
8	סיכום כללי:
11	שאלות:
12	תשובות סופיות:
13	פקודות מסוג I-Type:
13	סיכום כללי:
14	פקודות שכיחות בפורמט Immediate:
19	שאלות:
20	תשובות סופיות:
21	פקודות מסוג J-Type:
21	סיכום כללי:
23	שאלות:
23	תשובות סופיות:
24	פסאודו קוד:
24	סיכום כללי:
26	שאלות:
26	תשובות סופיות:

מבנה המחשב ותכן לוגי

האסמבלי של מעבדי ה-MIPS

הקדמה כללית:

סיכום כללי:

עיקרון ההפשטה:

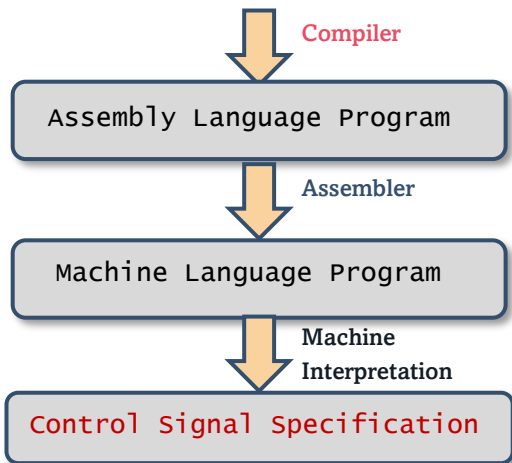
להקל את העיצוב וההבנה של מערכות מורכבות.

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
```

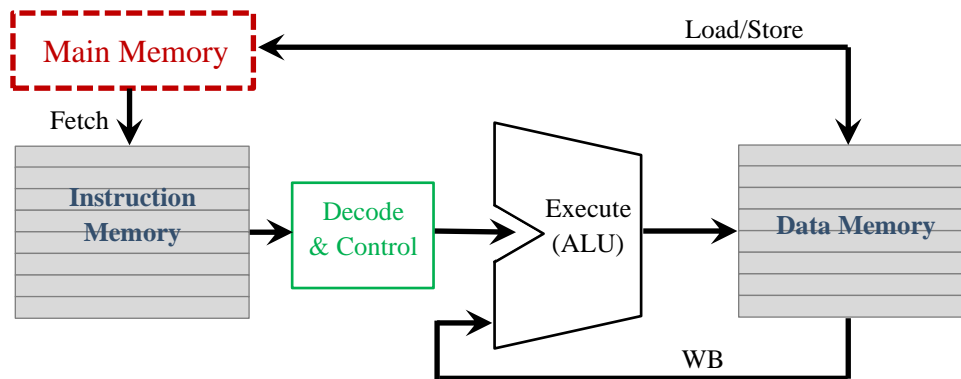
```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

```
ALUOP[0:3] <= InstReg[9:11] & MASK
```



ברצוננו לפשט בעיות מורכבות ולכן:

- הגדרת פונקציות בסיסיות (Functions).
- הגדרת דרך הביצוע של כל פעולת חישוב באמצעות פונקציות אלו.
 - (כלומר, לכנס אותן לסטים של הוראות - Instructions).
- הגדרת ממשק חומרתי (גם לבקרה וגם לזיכרון) שיאפשר:
 - טעינת מידע (Fetch) לפי דרישת הפונקציות שהוגדרו והמידע שיש לטעון.
 - קידוד (Decode) של הפונקציות לפקודות עבור ה-ALU.
 - ביצוע (Execute) של ההוראות באמצעות קווי הבקרה המתאימים.
 - אחסון (Store) של התוצאות המחושבות או כתיבה חזרה לביצוע (Write Back).



מסקנה כללית:

כדי להבין את מבנה המחשב יש להבין את אירגון המחשב ואת מבנה הפקודות:

$$\text{Computer Architecture} = \text{Instruction Set Architecture (ISA)} + \text{Machine Organization (Datapath and Control)}$$

ארכיטקטורה של מחשבים:

במדעי המחשב, ארכיטקטורה של מחשב מתארת את המבנה של מערכת מחשב.

ה-ISA:

ה-ISA הוא מודל אבסטרקטי המגדיר את הפקודות הנתמכות ע"י ה-CPU, סוגי המידע (data types), מבנה ומספר האוגרים (Registers) ותפקידם, החומרה התומכת בזיכרון הראשי, פעולות (פונקציות) בסיסיות, שיטת הכתובות בגישה לזיכרון ולאוגרים ומשפחת השירותים הניתנים כחלק מהתקני ה-I/O המחוברים ל-CPU. ה-ISA מגדיר בנוסף את שפת המכונה שתרוץ על רכיבי החומרה עבור כל פקודה ופקודה באופן שהוא בלתי תלוי במימוש של רכיבי החומרה אלא רק בתפקידם.

מושג ה-MIPS במבנה המחשב:

- החברה MIPS (מתוך ויקיפדיה):

MIPS, formerly MIPS Computer Systems, Inc. and MIPS Technologies, Inc., is an American fabless semiconductor design company that is most widely known for developing the MIPS architecture and a series of RISC CPU chips based on it. MIPS provides processor architectures and cores for digital home, networking, embedded, Internet of things and mobile applications.

- הגדרה של MIPS (מתוך ויקיפדיה):

MIPS (Microprocessor without Interlocked Pipelined Stages) is a family of reduced instruction set computer (RISC) instruction set architectures (ISA) developed by MIPS Computer Systems, now MIPS Technologies, based in the United States.

- המדד MIPS:

MIPS (metric) - Millions Instructions Per Second.

MIPS is an efficiency metric (for uCs)

A more informative metric is MIPS/MHz (MIPS per MHz)

מעבדים ידועים:

- מעבד ה-RISC-V ← מעבד מסוג MIPS32. (נעסוק בו וב-ISA שלו בקורס שלנו)
- מעבד ה-8086 ← מעבד CISC שפותח באינטל. (לא נעסוק בו במסגרת הקורס שלנו)

יסודות ה-ISA:

כלל תכנון 1 - פשטות עדיפה על כמותיות:

- מבנה פקודה ב-ISA למעבדי RISC בנוי בצורה הבאה:
- כל פקודה היא באורך קבוע של 32 סיביות (מילה).
 - ישנו מספר קטן של פורמטים אפשריים לפקודות.
 - בכל הפורמטים של ה-ISA, אורך שדה הקידוד (opcode) זהה ושדות האוגרים חופפים.
 - פקודות אריתמטיות ולוגיות תמיד ידרשו 3 אוגרים.

	Opcode	rs	rt	rd	shamt	funct
Number of bits:	6b	5b	5b	5b	5b	6b

תמיד נבחר בפקודות קצרות כגון: $R1 = R2 \text{ op } R3$

על פני קוד שמאפשר מספר רב של משתנים בפקודה בודדת כגון:

$$R7 = (R1 \text{ op } R2 \text{ op } R3) \text{ op } (R4 \text{ op } R5 \text{ op } R6)$$

אך מימוש חומרתי של מעבד אשר יוכל ליישם פקודה שכזו יהיה מאוד קשה וידרוש זמן חישוב רב.

כלל תכנון 2 - קטן זה מהיר:

נזכור כי קיימת אינטראקציה בלתי ניתנת להפרדה בין המעבד (ה-CPU שכולל את ה-ALU, יחידת האוגרים, ויחידת הבקרה ונתיבי מסלול הנתונים) לבין יחידות הזיכרון (SRAM ו-DRAM). קריאה של מידע מהזיכרון וכתובה חזרה לזיכרון אלו פעולות שדורשות זמן רב. לכן נעדיף לעבוד על מקבץ האוגרים שנמצא בתוך המעבד.

האופרנדים ב-ISA של מעבדי MIPS32:

אופרנד הוא ערך שמוזן בפקודה מסוימת, בין לקבלת מידע ובין להעברת מידע. ב-ISA שלנו נעסוק באופרנדים שהם האוגרים עצמם או מספרים קבועים. (מחשבי CISC כוללים אופרנדים נוספים המאפשרים לבצע פעולות גם בתוך יחידות הזיכרון).

טבלת מקבץ האוגרים של מעבדי ה-MIPS32 מסוג RISC-V:

שם	מספר	שימוש
\$zero	0	קבוע 0
\$at	1	שמור לאסמבלר
\$v0 - \$v1	2-3	תוצאות חישובים
\$a0 - \$a3	4-7	ארגומנטים מועברים לסברוטינות
\$t0 - \$t7	8-15	משתנים זמניים
\$s0 - \$s7	16-23	משתני שפת התכנות העלית
\$t8 - \$t9	24-25	משתנים זמניים
\$k0 - \$k1	26-27	שמור למערכת ההפעלה
\$gp	28	Global Pointer
\$sp	29	Stack Pointer
\$fp	30	Frame Pointer
\$ra	31	Return Address
pc	ללא	כתובת הפקודה הנוכחית
hi	ללא	אוגר עליון למילה בת 64 ביטים
lo	ללא	אוגר תחתון למילה בת 64 ביטים

כלל תכנון 3 - התפשרות בתמורה ליעילות:

ה-ISA של מעבדי MIPS32 (ובהם RISC-V) כולל שלושה פורמטים אפשריים לכתיבה של פקודות תקניות עבור המעבד. כאמור לעיל, על מנת לשמור על הפשטות שבקידוד הפקודה וביצועה, מבנה הפקודות דומה ככל הניתן.

Number of bits:	6b	5b	5b	5b	5b	6b
R	Opcode	rs	rt	rd	shamt	funct
I	Opcode	rs	rt	16 bit address		
J	Opcode	26 bit address				

פקודות מסוג R-Type:

אלו הן פקודות Register והן כוללות הוראות שיש לבצע על 3 אוגרים (קריאה משני אוגרים וכתיבה לאוגר שלישי). מדובר בפקודות אריתמטיות ולוגיות למיניהן.

פקודות מסוג I-Type:

אלו הן פקודות ה-Immediate ומטרתן היא לבצע פעולה על אוגר אחד ביחד עם מספר קבוע ולכתוב אותה לאוגר נוסף. פקודות אלו מהירות ולכן שימושיות ביותר.

פקודות מסוג J-Type:

אלו הן פקודות ה-Jump והן כוללות הנחייה לקפוץ להוראה מסוימת בתוכנה ללא תנאים.

כאשר :

Value	Description
<i>opcode</i>	The operation code field, specifying the operation to be performed
<i>rs</i>	Register Source 1 (first register to read data from)
<i>rt</i>	Register Source 2 (also called: Register Target) is the second register to read data from in R-Type commands, and Target Register to place data in I-Type commands
<i>rd</i>	Register Destination. Used in R-Type commands to write data to
<i>shamt</i>	Shift left/right or arithmetic shift left/right the bits of <i>rd</i> by <i>shamt</i>
<i>funct</i>	Code for the function to be executed

פקודות מסוג R-Type:

סיכום כללי:

פקודות מסוג R-Type:

Bits allocation:	31 - 26	25 - 21	20 - 16	15 - 11	10 - 6	5 - 0
R	Opcode	rs	rt	rd	shamt	funct

- שדה ה-Opcode יחזיק ערך של 0, כלומר: 000000.
- שדות ה-rs וה-rt מציינים את כתובות האוגרים שמהם נקרא את הנתונים.
- שדה ה-rd מציין את כתובת האוגר שאליה נכתוב את התוצאה.
- שדה ה-shamt (או sa) מוקדש להזזת סיביות של rt והשמת התוצאה ב-rd בלבד.
- שדה ה-funct מוקדש לפירוט סוג הפעולה שיש לבצע על האוגרים.

צורת הכתיבה של פקודות אלו (syntax) היא:

```
funct $rd, $rs, $rt # $rd = funct($rs,$rt)
```

להלן מספר מקרים שכיחים:

Operation	Funct value	Assembly Syntax	Performed Action
add	0x20 (32)	add \$t0, \$t1, \$t2	\$t0 = \$t1 + \$t2
sub	0x22 (34)	sub \$t0, \$t1, \$t2	\$t0 = \$t1 - \$t2
and	0x24 (36)	and \$t0, \$t1, \$t2	\$t0 = \$t1 & \$t2
or	0x25 (37)	or \$t0, \$t1, \$t2	\$t0 = \$t1 \$t2
xor	0x26 (38)	xor \$t0, \$t1, \$t2	\$t0 = \$t1 ^ \$t2
nor	0x27 (39)	nor \$t0, \$t1, \$t2	\$t0 = ~ \$t1 \$t2
slt	0x2A (42)	slt \$t0, \$t1, \$t2	if (\$t1 < \$t2) \$t0 = 1 else \$t0 = 0

אוגר האפס:

אוגר האפס מסומן \$zero והוא האוגר הראשון בקובץ האוגרים של המעבד. מטרתו היא לאחסן את הערך אפס כגודל קבוע שניתן להשתמש בו בקוד. לא ניתן לכתוב לאוגר זה אלא רק לקרוא את הערך ממנו.

פקודת sll (Shift Left Logical):

Bits allocation:

	31 - 26	25 - 21	20 - 16	15 - 11	10 - 6	5 - 0
R	Opcode	rs	rt	rd	shamt	Funct
values	0x0	0x0	0x9	0x8	0x3	0x0

```
sll $t0, $t1, 3 # the number in $t1 is shifted left by 3 bits
                # and is written to $t0
```

המספר בשדה ה-shamt מייצג את מספר הביטים שיש להזיז את תוכן האוגר \$t1 שמאלה. הערך המתקבל ייכנס לתור אוגר \$t0. ככלל, בפקודת sll שדה ה-rs יכול תמיד את המספר 0x0.

פקודת srl (Shift Right Logical):

Bits allocation:

	31 - 26	25 - 21	20 - 16	15 - 11	10 - 6	5 - 0
R	Opcode	rs	rt	rd	shamt	Funct
values	0x0	0x0	0x9	0x8	0x3	0x2

```
srl $t0, $t1, 3 # the number in $t1 is shifted right by 3 bits
                # and is written to $t0
```

המספר בשדה ה-shamt מייצג את מספר הביטים שיש להזיז את תוכן האוגר \$t1 ימינה. הערך המתקבל ייכנס לתור אוגר \$t0. ככלל, בפקודת srl שדה ה-rs יכול תמיד את המספר 0x0.

פקודת sra (Shift Right Arithmetic):

Bits allocation:

	31 - 26	25 - 21	20 - 16	15 - 11	10 - 6	5 - 0
R	Opcode	rs	rt	rd	shamt	Funct
values	0x0	0x0	0x9	0x8	0x3	0x3

בפקודה זו ההזזה תתבצע ימינה (עבור ערך shamt חיובי) או שמאלה (עבור ערך shamt שלילי) תוך תשומת לב ל-sign extension לפי סימן המספר שבאוגר המקור.

פקודות כפל:

כאשר מבצעים הכפלה של שני מספרים בני 32 ביטים ניתן לקבל תוצאה שגודלה יהיה עד 64 ביטים לכל היותר. עקב כך, מעבדי RISC-V מקצים שני אוגרים ייעודיים לאחסון תוצאה שכזו. האוגרים הנ"ל נקראים hi ו-lo וכאשר מתרחשת פעולה שתוצאתה גדולה מ-32 ביטים המעבד ישמור את התוצאה בשני אוגרים אלו.

```
mult $t0, $t1 # perform $t0 x $t1 and store the result in: hi, lo
              # as 64bits signed number.

multu $t0, $t1 # perform $t0 x $t1 and store the result in: hi, lo
              # as 64bits unsigned number.

mul $t0, $s1, $s2 # perform $t0 = $s1 x $s2
                  # where $t0 include the low-order 32 bits
                  # of the result
```

פקודות חילוק:

באופן דומה לפעולות הכפל, גם בפעולות חילוק נעזר בשני האוגרים המיוחדים hi ו-lo. אוגר ה-hi ישמור את השארית (remainder) ואוגר ה-lo ישמור את החילוק (quotient).

פקודות השמה של ערכים מתוך אוגרים מיוחדים:

כאשר נרצה לטעון מידע השמור באוגרים המיוחדים hi ו-lo נוכל להיעזר בשתי הפקודות הבאות:

```
mfhi $t0 # $t0 gets the content of hi register.
mflo $t0 # $t0 gets the content of lo register.
```

הפקודות mfhi (move from hi) ו-mflo (move from lo) למעשה טוענות את התוכן הנוכחי השמור בכל אחד מן האוגרים hi ו-lo לתוך אוגר היעד (בדוגמה שלנו - \$t0).

שאלות:

1) לפניכם קטע הקוד הבא:

```
and $s0, $s0, $zero
add $t0, $t1, $s0
add $t2, $t1, $t0
add $t3, $t2, $t0
sub $t4, $t3, $s1
```

- א. (1) מה תפקידה של הפקודה הראשונה בקוד?
 ב. (2) מה תפקידה של הפקודה השנייה בקוד?
 ג. כתבו מה הקוד מבצע מבחינה מתמטית.
 ד. באיזה אוגר יישמר המידע הסופי?
 ד. לפני פקודת החיסור (sub \$t4, \$t3, \$s1) מוסיפים את הפקודות הבאות:

```
add $s1, $t1, $zero
sll $s1, $s1, 2
```

- (לאחריהן תופיע פקודת החיסור).
 בהנחה כי התוכן שבאוגר \$t1 מייצג מספר חיובי, מה תהיה ערכה של
 סיבית ה-MSB באוגר \$t4?

2) היעזרו בטבלאות ערכי השדות של פקודות R-Type וכתבו את הקוד המתקבל
 בשפת מכונה לכל אחת מהפקודות הבאות:
 א.

```
add $t0, $s0, $zero
```

ב.

```
and $t3, $t4, $s2
```

ג.

```
or $s0, $s0, $v1
```

ד.

```
slt $t0, $t7, $a0
```

3) קבעו מה כל קוד הנתון בשפת מכונה מבצע. במידה והקוד אינו תקין נמקו מדוע.

א. $0x02114825$

ב. $0x01ECA827$

ג. $0x00C9502A$

ד. $0x010950E0$

תשובות סופיות:

1) א. (1) הפקודה מאתחלת את: $\$s0 = 0x0$.

א. (2) הפקודה מעתיקה את התוכן של $\$t1$ ל- $\$t0$.

ב. $\$t4 = 3 \times \$t1 - \$s1$. ג. באוגר $\$t4$. ד. $\$t4[31] = 1$.

2) א. $0x02004020$. ב. $0x01925824$. ג. $0x02038025$. ד. $0x01E4402A$.

3) א. $\$t1, \$s0, \$s1$ or . ב. $\$t4, \$t7, \$s5$ Nor . ג. $\$t1, \$a2, \$t2$ Slit . ד. לא תקינה!

פקודות מסוג I-Type:

סיכום כללי:

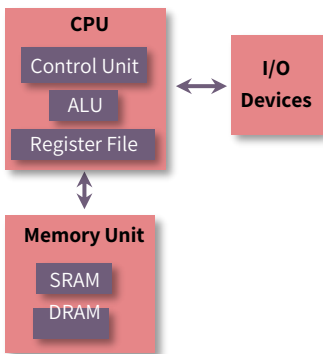
תזכורת:

מחשבים מסוג MIPS32 נקראים כך משום שהארכיטקטורה שלהם תומכת במידע (data) וכתובות (addresses) שהן בנות 32 ביטים, כלומר באורך של מילה.

$$1 \text{ Byte} = 8 \text{ bits} \quad \rightarrow \quad 1 \text{ B} = 8 \text{ b}$$

$$1 \text{ Word} = 4 \text{ Bytes} = 32 \text{ bits} \quad \rightarrow \quad 4 \text{ B} = 32 \text{ b}$$

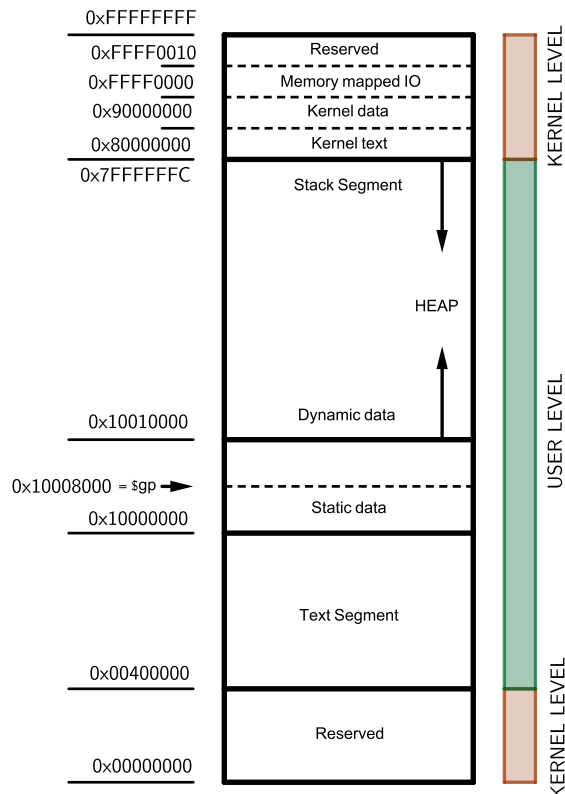
יחידת הזיכרון (Main Memory) ומידע (Data) במחשב MIPS32:



למחשב מסוג MIPS32 שבו מעבד RISC-V מידע מועט ומיידי השמור באוגרים (Registers) שבתוך יחידת העיבוד (ה-CPU). המידע המרכזי (Main Memory) שמור ביחידה הנפרדת מהמעבד עצמו, שהיא יחידת הזיכרון.

מבנה הזיכרון של מחשב MIPS32:

למחשב MIPS32 יש נפח אחסון של 2^{30} מילים או 4GB זיכרון.



איך נשמר הערך של מילה בזיכרון?

מילה הנשמרת בזיכרון מורכבת מ-4 בתים, אותם יש לשמור על פני 4 כתובות סמוכות. נסתכל על הערך: 0xBF014328 אשר יש לשמור בכתובת 0x10003010 בזכרון:

address	value	
	Big Endian	Little Endian
0x10003010	BF	28
0x10003011	01	43
0x10003012	43	01
0x10003013	28	BF

כאשר הבית הגבוה של מילה נשמר בכתובת הנמוכה ביותר אופן השמירה נקרא Big Endian.

כאשר הבית הגבוה של מילה נשמר בכתובת הנמוכה ביותר אופן השמירה נקרא Little Endian.

מעבדי MIPS נעזרים ב-Big Endian בעוד שמעבדי אינטל נעזרים ב-Little Endian.

פקודות שכיחות בפורמט Immediate:

פקודות גישה לזיכרון:

פקודת (lw) load word:

Bits allocation:	31 - 26	25 - 21	20 - 16	15 - 0
I	0x23	rs	rt	Address

דוגמה:

האוגר \$t0 יקבל את התוכן שנמצא בכתובת בזיכרון השווה לערך שנמצא ב-\$t1 בתוספת 0x1000.

```
lw $t0, 0x1000($t1) # $t0 = memory(0x1000+$t1)
```

פקודת (sw) store word :

Bits allocation:

	31 - 26	25 - 21	20 - 16	15 - 0
I	0x2B	rs	rt	Address

דוגמה :

התוכן שנמצא באוגר \$t0 יישמר בכתובת בזיכרון השווה לערך שנמצא ב-\$t1 בתוספת 0x1000.

```
sw $t0, 0x1000($t1) # memory(0x1000+$t1) = $t0
```

הערות :

- (1) לעיתים, ערך האוגר rs נקרא גם base ושדה הכתובת נקרא offset. כך הערך הכתובת המחושב הוא : offset(base).
- (2) ההיסט (offset) חייב להיות מספר המתחלק פי 4 (כלומר שני הביטים הראשונים לעולם יהיו 0). אחרת נקבל היסט שלא מגיע לתחילתה של מילה ותתקבל שגיאת הרצה.
- (3) ערך ההיסט (offset) הוא קבוע המוגדר ב-16 ביטים הנתונים במשלים ל-2. כלומר ניתן ללכת גם 'אחורנית' עם היסט שלילי.
- (4) הכתובת שנבחרת בפקודות lw ו-sw מצביעה על הבית הנמוך ביותר. כלומר המילה שנבחרת (בין אם לטעינה ובין אם כתובת לשמירת מידע) תצביע על הבית הראשון מתוך 4 הבתים שהוא בכתובת הנמוכה ביותר.

פקודות לוגיות מיידיות :

פקודת andi :

בפקודה הבאה האוגר \$t0 יקבל את תוצאת פעולת and של התוכן השמור ב-\$s0 עם 0x1C0. כלומר : \$t0 = AND(\$s0 , 0000 0000 0000 0000 0000 0001 1100 0000).

```
andi $t0, $s0, 0x1C0 # $t0 = $s0 & 0x1C0
```

פקודת ori :

בפקודה הבאה האוגר \$t0 יקבל את תוצאת פעולת or של התוכן השמור ב-\$s0 עם 0x7EF. כלומר: $\$t0 = \text{AND}(\$s0, 0000\ 0000\ 0000\ 0000\ 0000\ 0111\ 1110\ 1111)$.

```
ori $t0, $s0, 0x7EF # $t0 = $s0 | 0x7EF
```

כלל:

בכל הפקודות הלוגיות המיידיות, הערך הקבוע תמיד מרופד באפסים (zero extend).

פקודות אריתמטיות מיידיות:

פקודת addi (Add Immediate):

הפקודה נועדה להוסיף קבוע לערך הנמצא באוגר נתון. בפקודה הבאה האוגר \$t0 יקבל את תוצאת החיבור של התוכן הנמצא ב-\$s0 ביחד עם 0x100. כלומר: $\$t0 = \text{add}(\$s0, 0000\ 0000\ 0000\ 0000\ 0000\ 0001\ 0000\ 0000)$.

```
addi $t0, $s0, 0x100 # $t0 = $s0 + 0x100
```

אוגר ה-PC (Program Counter) שבמקבץ האוגרים במעבד:

המעבד כולל אוגר מיוחד שנקרא PC (Program Counter) ומטרתו היא להצביע על כתובת הפקודה הבאה אותה יש לבצע. ברירת המחדל של אוגר ה-PC היא הערך 0x00400000. היות וכל כתובת היא באורך של מילה, ה-PC יכול את ערך הכתובת של פקודה נוכחית ולאחר שהיא הושלמה, ערכו יקודם ב-4.

פקודות branch :

מדובר בשתי פקודות הכוללות התניה לצורך קפיצה לפקודה הנמצאת במיקום אחר בתכנית.

פקודת (Branch if Equal) beq :

Bits allocation:

	31 - 26	25 - 21	20 - 16	15 - 0
I	0x4	rs	rt	offset

דוגמה :

נניח והפקודה הנ"ל נמצאת בכתובת בזיכרון שתסומן ב-PC. הפקודה הבאה בתכנית נמצאת בכתובת PC + 4. אם התוכן שנמצא באוגרים \$t0 ו-\$t1 זהה, אז יש לקפוץ לפקודה הנמצאת בכתובת:
 $PC + 4 + 4 * 0x39$

```
beq $t0, $t1, 0x39 # if ($t0 == $t1) then: PC = PC + 4 + 4*0x39
```

פקודת bne :

Bits allocation:

	31 - 26	25 - 21	20 - 16	15 - 0
I	0x5	rs	rt	offset

דוגמה :

נניח והפקודה הנ"ל נמצאת בכתובת בזיכרון שתסומן ב-PC. הפקודה הבאה בתכנית נמצאת בכתובת PC + 4. אם התוכן שנמצא באוגרים \$t0 ו-\$t1 שונה, אז יש לקפוץ לפקודה הנמצאת בכתובת:
 $PC + 4 + 4 * 0x39$

```
bne $t0, $t1, 0x39 # if ($t0 != $t1) then: PC = PC + 4 + 4*0x39
```

הערות :

- (1) הערך שנמצא ב-offset הוא בעל סימן, כלומר ניתן לנוע קדימה ואחורה בתכנית.
- (2) טווח הדילוג, עבור 16 ביטים הוא ±16KB (קדימה - מספר חיובי, אחורה - מספר שלילי).
- (3) בכתובת תכנית ב-Assembly, ניתן להזין בשדה ה-offset 'תווית' (label) למיקום בתכנית במקום מספר המציין את הכתובת הפיזית של הפקודה.

פקודות Immediate נוספות:

פקודת lui:

ראינו כי פקודת addi (Add Immediate) מאפשרת לבצע חיבור עם קבוע שגודלו הוא באורך 16 ביטים. במקרים בהם יהיה צורך בחיבור של מספר גדול יותר נעזר בשתי פקודות באופן הבא:

(1) פקודת lui (Load Upper Immediate).

(2) פקודת ori (Or Immediate).

פקודת lui לוקחת את 16 הביטים של Immediate, מזיזה אותם 16 מקומות כך שהם 16 הביטים העליונים של מספר בן 32 ביטים, ומוסיפה 16 אפסים. את הערך הזה כותבים לאוגר היעד.

Bits allocation:

	31 - 26	25 - 21	20 - 16	15 - 0
lui	0x15	0x0	rt	Address / immediate

כאשר נוסף את הפקודה הבאה נקבל טעינה של מילה מלאה (בת 32 ביטים):

דוגמה:

```
lui $t0, 0x178B # the number 0x178B0000 is placed into $t0
ori $t0, $t0, 0xFF41 # the number 0x0000FF41 is ORed with $t0
```

\$t0 = 0x178BFF41

פקודות המתבצעות על בתים:

במידה ונרצה ליישם פקודות על בתים (8 ביטים) ולא על מילה שלמה (32 ביטים) נוכל להיעזר בסט הפקודות העוסקות בפעולות על בתים בלבד.

פקודת lbu (Load Byte Unsigned):

נניח כי: \$s0 = 0x0020B004 וכי התא אליו מצביעים בזיכרון מכיל: 0xABAA00462.

```
lbu $t0, 400($s0) # $t0 gets the value inside memory cell at the
# address of: $s0 + 400DEC, or 0x0020B194
```

התוכן שיוזן לאוגר \$t0 הוא רק את הבית הראשון ב-LSB שלו כאשר הריפוד הוא באפסים ולא לפי סימן (Zero extend): \$t0 = 0x00000062.

שאלות:

1) לפניכם קטע הקוד הבא:

ידוע כי בכל התאים של יחידת הזיכרון מוחזק הערך 0x2.

```
add $s0, $zero, $zero
lw $t0, 0x400($s0)
lw $t1, 0x404($s0)
add $s0, $t1, $t2
sw $s0, 0x400($s0)
```

- מהי הכתובת ומה הוא הערך שייכתבו לזיכרון בסיום התכנית הבאה.
- לאחר שורת ה-add ולפני שורת ה-sw מכניסים פקודה נוספת והיא:

```
add $s0, $s0, $t1
```

האם כעת ייכתב ערך מסוים לזיכרון? אם כן מה הוא ולאילו כתובת?

2) נתון מערך בן שני תאים בזיכרון:

Address	Content
0x00008200	0x00000560
0x00008204	0x000010AA

יש לכתוב קטע קוד שמוסיף לתא הראשון 100, מחסר מהתא השני 50 ומחזיר את הערכים לתאים ההפוכים. (כלומר התוכן המחושב של התא הראשון יחזור לתא השני והתוכן המחושב של התא השני יחזור לתא הראשון).

3) נתונים שני תאים סמוכים בזיכרון.

יש לכתוב תכנית אשר לוקחת את המידע מתאים אלו, ומחזירה לתא הסמוך להם ערך שבו 16 הביטים ה-MSB אלו הם הביטים העליונים של תוכן התא הראשון, ו-16 הביטים ה-LSB הם הביטים הנמוכים של התא השני. ניתן להניח כי האוגר \$s0 מצביע לכתובת התא הראשון בזיכרון.

דוגמה:

נניח ושני תאים סמוכים בכתובות הנ"ל מכילות את התוכן הבא:

Address	Content
0x10008000	0x4AB03482
0x10008004	0x007809C1

אז יש לכתוב לתא שבכתובת 0x1008008 את התוכן: 0x4AB009C1.

- (4) נתון מערך A השמור בזיכרון כאשר כתובת האיבר הראשון במערך, $A[0]$, שמורה ב- s_0 .
 ערכו של הקבוע g שמור באוגר s_1 וערכו של הקבוע h שמור באוגר s_2 .
 ערך האינדקס i אליו מצביעים בגישה הראשונה לפונקציה Loop שמור באוגר s_3 .
 יש לכתוב בשפת אסמבלי קוד המממש את הפונקציה Loop הבאה:

```
Loop: g = g + A[i]
      i = i+1
      if(i != h) goto Loop
```

תשובות סופיות:

- (1) א. לתא שבכתובת $0x404$ ייכנס הערך $0x4$.
 ב. תתקבל שגיאת הרצה היות והכתובת אינה תקינה.
- (2) ראו כתיבת קטע קוד ושיקולים בסרטון הוידאו.
- (3) ראו כתיבת קטע קוד ושיקולים בסרטון הוידאו.
- (4) ראו כתיבת קטע קוד ושיקולים בסרטון הוידאו.

פקודות מסוג J-Type:

סיכום כללי:

פקודות מסוג J-Type:

פקודת jump:

Bits allocation:

	31 - 26	25 - 0
I	0x2	Address

הסבר:

פקודת jump היא קפיצה לפקודה בכתובת יעד ללא תנאי (unconditional jump). כדי לדעת מהי כתובת היעד מוקצים 26 ביטים של שדה ה-address. היות והקפיצות הן בכפולות של 4, ברור כי כתובת היעד תתחיל עם שני אפסים כסיביות ה-LSB. מבחינה הקסדצימלית, ספרת ה-LSB תהיה אך ורק 0x0, 0x4, 0x8 או 0xC. לכן, אין צורך לשמור שני אפסים ומקובל להזיז את ערך ה-address בשני מקומות שמאלה. קיבלנו אפשרות להציג כתובת ע"י 28 סיביות! את 4 סיביות ה-MSB של הכתובת ניקח מכתובת המוצא הנוכחית.

דוגמה:

נניח ופקודת jump נמצאת בכתובת PC = 0x0501013C

```
j 0x0040001C # jump to: PC = (PC + 4)&(0xF0000000) + 0x0040001C<<2
```

קפיצה תתבצע לפקודה שבכתובת: Target Address = 0x11000070.

הערה:

בכתיבת קוד שוטף, אין צורך לבצע חישובים מסוג זה אלא נעזר בתווית (label) שתצביע לכתובת של פקודה מסוימת, למשל:

```
MAIN: add $s0, $t1, $s1
      addi $t0, $s0, 0x40
      j  RUTINE
.....
RUTINE: addi $t0, $s1, 0x8000
        beq $t0, $t2, MAIN
```

השוואה בין פקודות branch ו-jump:

ראינו כי פקודת branch היא קפיצה מותנית בעוד שפקודת jump מתארת קפיצה בלתי מותנית. בנוסף, בפקודת branch יש רק 16 ביטים לשדה ה-Address בעוד שבפקודת jump ישנם 26 ביטים ולמדנו כיצד מתרגמים את יתרת הביטים לכדי כתובת בת 32 ביטים.

Bits allocation:	31 - 26	25 - 21	20 - 16	15 - 0
(I) Branch	0x4	rs	rt	Address
(J) Jump	0x2	Address		

מבחינת הפקודות עצמן:

```

beq $t0, $t1, LABEL # Next instruction is at LABEL if $t0 = $t1
bne $t0, $t1, LABEL # Next instruction is at LABEL if $t0 != $t1
j LABEL # Next instruction is at LABEL
    
```

מסקנה:

קפיצות branch נחשבות מקומיות (לוקליות) בעוד שקפיצות jump מתפרסות על כל התכנית.

שאלות:

1) לפניכם קטע הקוד הבא הכולל לולאה בשם Loop. עליכם לכתוב את הערכים בכל אחד מן השדות של הפקודות בטבלה מלמטה עבור הפקודות המבוצעות בתוך Loop. כמו כן עליכם להשלים את כתובות הפקודות בזיכרון הראשי.

```
Loop: lw $10, 2000($20)
      beq $10, $16, End
      add $18, $18, $19
      j   Loop
End:  VALUE = 1600
```

שימו לב, הכתובות מופיעות בערך עשרוני ויש לכתוב את הערכים בשדות כמספרים עשרוניים.

Address	
60,000	lw
	beq
	add
	j
	End

תשובות סופיות:

1) להלן טבלה מלאה:

Address							
60,000	lw	35	20	10	2000		
60,004	beq	4	10	16	2		
60,008	add	0	18	19	18	0	32
60,012	j	2	15000				
60,016	End						

פסאודו קוד:

סיכום כללי:

הגדרה:

פקודות אלו אנו Pseudo Instruction נועדו לאגד מספר פקודות מכונה על מנת לפשט ולהקל על תהליך הכתיבה של קוד. נשתמש בהן באופן שוטף במהלך כתיבה של קוד מבלי להתחשב בעובדה שהן לא מיושמות בצורה ישירה במעבד אלא כמספר פקודות.

הבדלים עקרוניים בין פקודות פסאודו ופונקציות באסמבלי:

(1) טבע הפקודות ומטרתן:

פקודות פסאודו אינן פונקציות במהותן אלא פירוש מילולי שנועד לפשט את הקוד. בפועל פקודת פסאודו מתבצעת ע"י סדרה של פקודות שה-CPU יודע לבצע. פונקציות לעומת זאת מורכבות מסט של פקודות אשר ה-CPU יודע לבצע ומוגדר עבורן.

(2) ייצוג:

פקודות פסאודו מיוצגות ע"י שורת קוד בודדת עם פורמט כתיבה ייעודי להן בעוד שפונקציות מוגדרות ע"י תוויות המצביעות על כתובות שונות בזיכרון וכן פעולות שונות שעל המעבד לבצע.

(3) שימושיות:

פקודות פסאודו נועדו בעיקר כדי לאפשר למתכנתים לכתוב קוד בשפה עילית ככל הניתן מתוך מטרה לפשט אותו ואת המורכבות שלו. פונקציות נועדו להפוך את הקוד למודולרי ומטרתן היא לאפשר לעקוב אחר התוכנה והשלבים שלה, לחלק את מבנה התוכנה לרמות ועוד.

פקודת move:

```
move $t0, $s3 # $t0 gets the content of $s3: $t0 = $s3.
```

פקודה בסיסית ושכיחה ביותר שכל מטרתה היא 'להזיז' תוכן מאוגר אחד לאחר. בפועל הפקודה ממומשת ע"י פקודת add עם אוגר האפס כך:

```
add $t0, $s3, $zero
```


פקודת li (Load Immediate):

```
li $t0, 0x50 # $t0 gets the value 0x50
```

פקודה המאפשרת לטעון ערך קבוע מיידית לאוגר מבוקש.
בפעול הפקודה מבוצעת ע"י פקודת addi עם אוגר האפס כך:

```
addi $t0, $zero, 0x50
```

פקודת la (Load Address):

```
la $t0, LABEL # $t0 gets the address of LABEL
```

נניח ו-LABEL נמצאת בכתובת 0x12345678, אז הפקודה la מתבצע באופן הבא:

```
lui $t0, 0x1234
```

```
ori $t0, $t0, 0x5678
```

פקודת blt (Branch on Less Than):

```
blt $t0, $t1, LABEL # if $t1 < $t0 then for to:  
# PC = PC + 4 + 4*LABEL
```

הפקודה משלבת שתי פקודות מוכרות והן slt ו-bne באופן הבא:

```
slt $at, $t0, $t1
```

```
bne $at, $zero, LABEL
```

שאלות:

- (1) כתבו קוד הסוכם את כל הספרות הזוגיות של מספר הנמצא באוגר $s0$ ומחזיר את התוצאה לאוגר $t0$.
הגדירו את ה-`data segment` ואת ה-`text segment`, וכן את התוויות בהן תשתמשו בקוד שלכם.

לדוגמה:

- המספר: $s0 = 0x45A03781$ כולל את הספרות הזוגיות הבאות: 4, A, 0, 8.
כאשר נסכום אותם נקבל את הערך 22 (עשרוני) ולכן: $t0 = 0x16$.

תשובות סופיות:

- (1) ראו כתיבת קטע קוד ושיקולים בסרטון הוידאו.