ארגון המחשב ושפת סף

תכנות מתקדם באסמבלי

תוכן העניינים:

היכרות עם סביבת העבודה של MARS:

הקדמה:

בשיעור זה נתמקד בשני היבטים הקשורים לתכנות באסמבלי של מעבדי ה-MIPS32:

- היכרות עם סביבת העבודה של תוכנת MARS.
- עקרונות בתכנות: שימוש במשתנים, באוגרים, ובלולאות שונות.

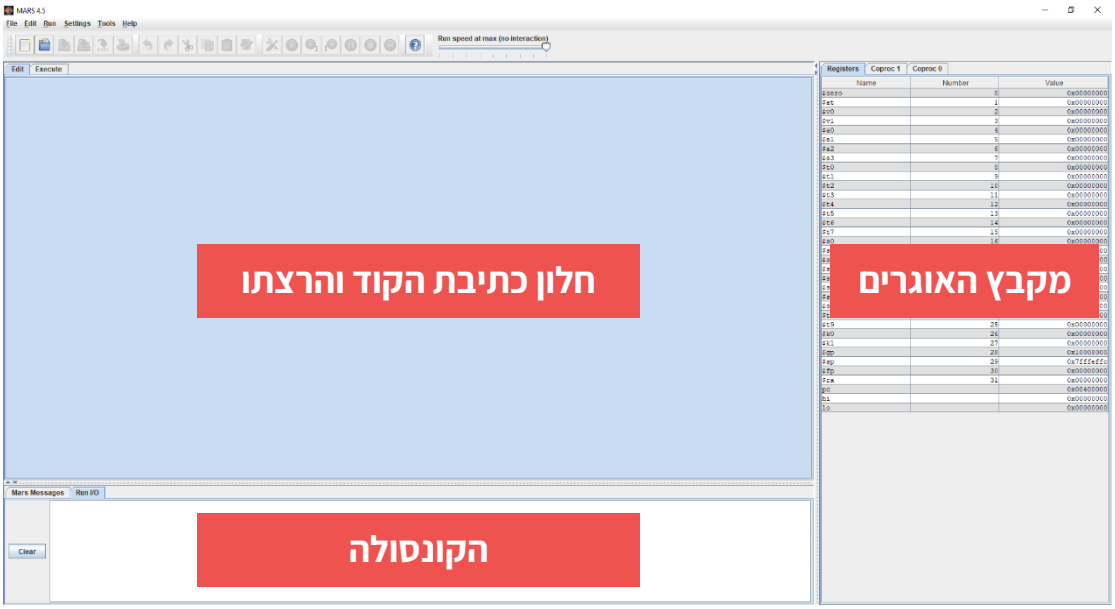
תוכנת ה-MARS:

סביבת העבודה שאיתה נעסוק נקראת MARS (קיצור: MIPS Assembly and Runtime Simulator).
ניתן להוריד את התוכנה דרך הקישור הבא:

MARS MIPS simulator

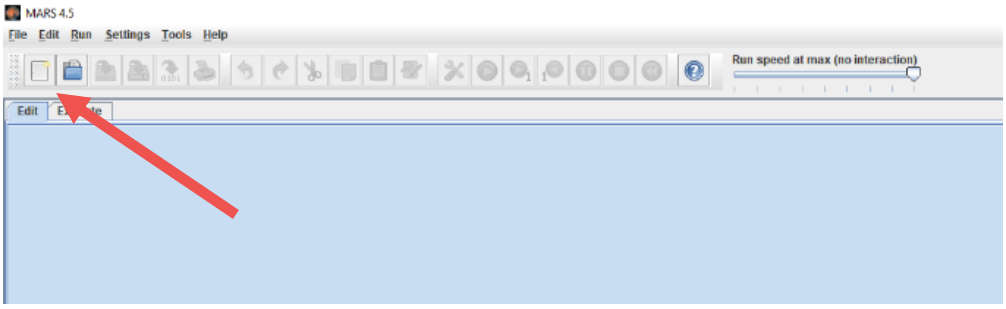
או בלחיצה על האייקון של התוכנה בצד ימין.

התוכנה פועלת כקובץ javascript ולכן יש לפתוח את קובץ ה-Mar4_5.jar בכל פעם מחדש.
החלון המתקבל בפתיחת התוכנה מורכב מ-3 אזורים:

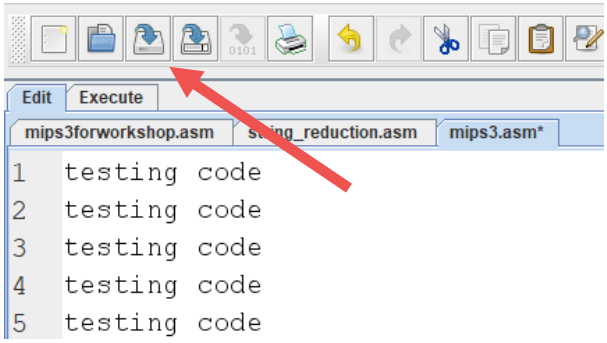


חלון כתיבת הקוד:

כאשר נפתח קובץ חדש כאן נכתוב את התוכנית שלנו.
כדי להתחיל כתיבה של קוד יש ללחוץ על new בסרגל העליון:



לאחר מכן יש לשמור את הקובץ (רצוי במיקום נגיש) באופן הבא:



וכעת אנו מוכנים לכתיבת הקוד.

הקונסולה:

באזור זה נקבל הודעות לגבי שגיאות בקוד, וכן הדפסות של תוכן שנרצה לבצע בקוד שלנו. קיים כפתור clear בצד שבו נשתמש כדי לנקות תוכן קודם בין ריצה לריצה.

מקבץ האוגרים:

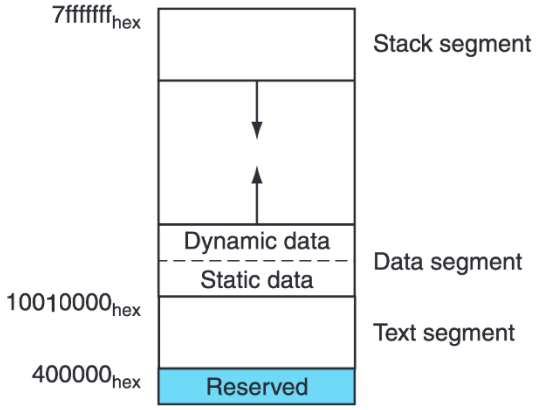
כאן נראה בזמן-אמת את הערכים השמורים בכל אחד מ-32 האוגרים של מקבץ האוגרים (וכן מספר אוגרים נוספים שנלמד עליהם בהמשך) כאשר נריץ את התוכנית שלנו שורה-שורה. הדבר יעזור להבין כיצד התוכנית מתנהגת, כיצד קריאה ויציאה מפרוצדורות מתרחשת וכן למצוא באגים אפשריים ולתקן אותם.

מרחב הזיכרון:

נזכור כי במחשבי ה-MIPS32 מרחב הזיכרון הוא בגודל של $2^{32}\,B = 4GB$.
מרחב הזיכרון מתחיל בכתובת 0x00000000 ומסתיים בכתובת 0xFFFFFFFF.
הזיכרון מורכב מאזורים שונים, הנקראים סגמנטים (segments).

אנו נתעניין בשני סגמנטים עיקריים והם:

- סגמנט/מקטע הנתונים (data segment) מתחיל בכתובת 0x10010000 וכולל שטח לאיחסון נתונים שונים.
- סגמנט/מקטע הקוד (text segment) מתחיל בכתובת 0x00400000 אליו מאותחל אוגר ה-PC ושומר את כל פקודות התוכנית.



שלבים ראשונים בכתיבה של קוד:

הפרדת הסגמנטים בקוד אותו נכתוב מתבצעת באופן הבא:

```
.data
        כאן כותבים את כל הנתונים שעל התוכנית לשמור במרחב זיכרון הנתונים
.text
        כאן כותבים את התוכנית עצמה
```

אתחול של נתונים לזיכרון הנתונים:

ישנם טיפוסים שונים ומגוונים של נתונים שניתן לשמור כמידע בזיכרון הנתונים שניתן יהיה להשתמש במהלך הרצת התוכנית. להלן רשימה חלקית ואופן הכתיבה שלהם בתוך מקטע הנתונים:

```
.data
LABEL1:     .asciiz "Here is your code:"
LABEL2:     .word 2, 3, 4, 5, 6, 7
LABEL3:     .space 3
LABEL4:     .byte 2, 3, -3, 'A', '5', 6, 0x12
```

- טיפוס asciiz.

נועד לכתיבה של מחזורת תווים לפי טבלת ASCII כאשר כל תו הוא בגודל בית.

- טיפוס word.

נועד לכתיבה של מספרים כאשר כל מספר מאוחסן ב-32 ביטים עם סימן.

עקב כך המספרים שניתן להזין הם בתחום: $\left[-2^{31} : 2^{31}-1\right]$.

- טיפוס space.

מגדיר בית ריק (ערך 0) לפי הכמות המבוקשת.

- טיפוס byte.

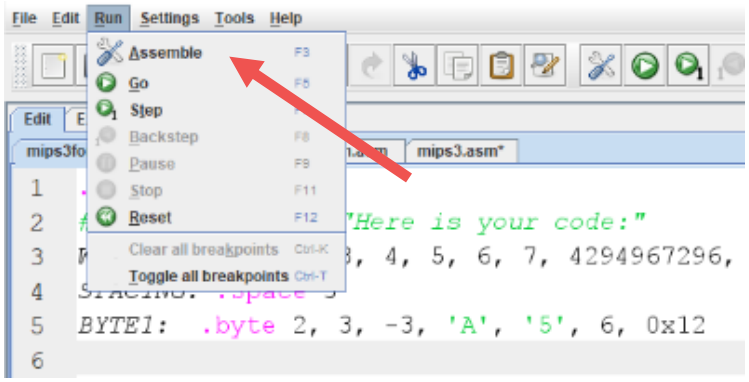
נועד כדי לכתוב מספר או ערך ב-ASCII אשר יישמר בבית בודד.

שמירה והרצה של התוכנית:

כדי להריץ את קטע הקוד יש לבצע את הפעולות הבאות (בכל פעם):

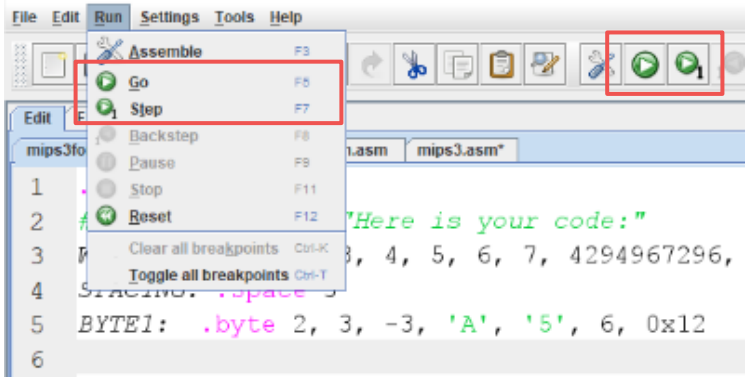
1) שמירה של קטע הקוד כפי שראינו קודם.

2) בסרגל העליון, בחירה באפשרות run ולחיצה על assemble או על F3.



3) לחיצה על כפתור go עבור הרצה של כל התוכנית בפעם אחת.

או לחיצה על הכפתור של step עבור הרצה של שורה-שורה.



כתיבה של התוכנית:

בכתיבת התוכנית נכיר פקודות שונות שלא בהכרח פגשנו בלמידה התיאורטית של שפת אסמבלי. בחלקן נעשה שימוש באופן תדיר ובאחרות נעשה שימוש לפי צורך מסוים.

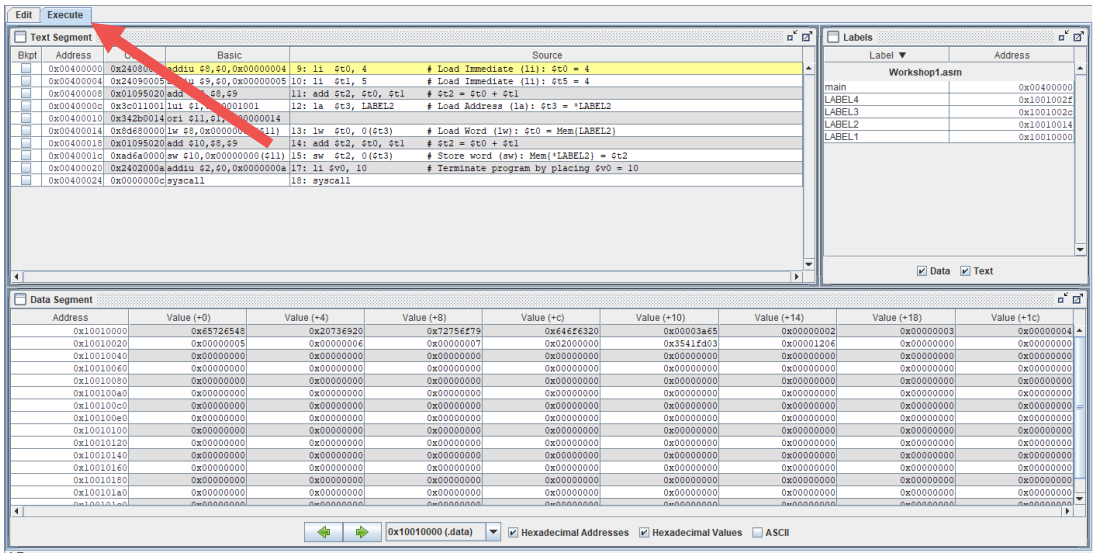
נכתוב למשל את קטע הקוד הבא (לאחר מקטע הנתונים):

```
.text
main:
li   $t0, 4            # Load Immediate (li): $t0 = 4
li   $t1, 5            # Load Immediate (li): $t5 = 4
add  $t2, $t0, $t1     # $t2 = $t0 + $t1
la   $t3, LABEL2       # Load Address (la): $t3 = *LABEL2
lw   $t0, 0($t3)       # Load Word (lw): $t0 = Mem{LABEL2}
add  $t2, $t0, $t1     # $t2 = $t0 + $t1
sw   $t2, 0($t3)       # Store word (sw): Mem{*LABEL2} = $t2


li $v0, 10             # Terminate program by placing $v0 = 10
syscall
```

חלון ה-Execute בתוכנת ה-MARS:

לאחר לחיצה על assemble, ה-MARS בונה את הסימולציה ומייצאת את הפקודות, התוויות ומרחב הנתונים אשר כולם נגישים בזמן אמת בטאב ה-Execute:

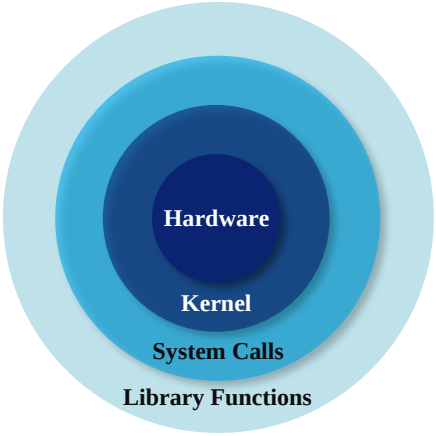


בחלון זה מופיע המידע לגבי כל האזורים והמצביעים שהגדרנו בקוד:

- חלון ה-text segment כותב את כל הפקודות של קוד וכולל את:
  - הכתובת בזיכרון של כל פקודה החל מ-0x00400000.
  - הפקודה בשפת מכונה (עמודה code).
  - הפקודה בשפה בסיסית (לא פסאודו פקודה) בעמודה basic.
  - קוד המקור שכתבנו בעמודה Source.

  בנוסף קיימת אפשרות להזין break points במידה ורוצים לבצע מעקב דקדקני יותר סביב הרצת הקוד.
- חלון התוויות (Labels) הכולל את כל התוויות שהוגדרו בתוכנית, הן במקטע הנתונים והן במקטע הקוד. כל תווית היא למעשה מצביע ששומר את כתובת הנתון או הפקודה המופיעה איתה.
- חלון ה-data segment בו ניתן לראות פיזית את כל מקטע זיכרון הנתונים החל מהכתובת 0x10010000 ולעקוב אחר המידע שהוזן מתוך מקטע הנתונים וכן המידע שיעודכן במהלך הרצת הקוד.

מודל קליפות הבצל של מחשב:



מודל קליפות הבצל של MIPS32 מתאר את שכבות הריצה במערכת המחשב באופן היררכי:

- **Library Functions**: פונקציות ברמה הגבוהה ביותר שזמינות למפתחים. אלו הן פונקציות מוכנות בספריות אשר מקלות על המתכנת ומאפשרות לו לבצע פעולות נפוצות בלי לכתוב קוד מורכב מאפס.
- **System Calls**: קריאות מערכת הן הממשק בין תוכניות המשתמש לבין מערכת ההפעלה. כאשר תוכנית צריכה שירות שדורש הרשאות מיוחדות (כמו גישה לקבצים או חומרה), היא מבצעת קריאת מערכת.
- **Kernel**: גרעין מערכת ההפעלה שמנהל את המשאבים של המחשב. הקרנל מספק שירותים לתוכניות דרך קריאות המערכת ומתקשר ישירות עם החומרה.
- **Hardware**: השכבה הבסיסית ביותר - החומרה הפיזית של המחשב (מעבד, זיכרון, התקני קלט/פלט).

תהליך הקריאה: תוכנית משתמש קוראת ל-Library Function, שלעתים קרובות מממשת את הפעולה באמצעות ה-System Call, אשר מועברת לקרנל, שמבצע את הפעולה הנדרשת דרך פנייה ישירה לחומרה.

בעבודה עם סימולטור MARS נתמקד ב-System Calls כדי לבצע פעולות כמו קריאה/כתיבה מהמסך, גישה לקבצים ופעולות I/O אחרות, מה שיאפשר לנו לתקשר עם "העולם החיצוני" מתוך הקוד שלנו.

פקודת ה-Syscall:

Syscall זו היא פקודת אסמבלי של מעבד ה-MIPS32 היוצרת חריגה (exception) בתוכנית (כלומר: הפסקת התוכנית ומעבר לביצוע קטע קוד המוגדר מראש שיושב ב-kernel).
בהתאם לערכי האוגרים $v0 ו-$a0 (ולעיתים גם $a1), הקרנל יודע מה עליו לבצע.
אופן ביצוע החריגה מנוהל ע"י קטע קוד במערכת ההפעלה הנקרא exception handler.

טבלת ערכים טיפוסיים עבור חריגות באמצעות Syscall:

Service	$v0	Argument / Result
Print Integer	1	$a0 = integer value to print
Print String	4	$a0 = address of null-terminated string
Read Integer	5	$v0 = integer read
Read String	8	$a0 = address of input buffer $a1 = max. number of characters to read
Exit Program	10	
Print char	11	$a0 = character to print
Read char	12	$v0 = character read

הערות:

1) קיימות פקודות נוספות (כגון $v0 = 34 או 35 וכו') אשר מבצעות פעולות נוספות.
אולם לא נעסוק בהן בתכנות קטעי הקוד בקורס שלנו.

2) כל תוכנית שנכתבת חייבת להסתיים עם חריגת ה-Exit program.
בכך המחשב משחרר את המקום בזיכרון אשר הוקצה להרצת התוכנית.

לולאות בתוכנית אסמבלי:

סוגי פקודות קפיצה מותנות פופולריות למימוש לולאות:

בכתיבה של קוד לעיתים נצטרך לממש לולאות כגון: for , if , while וכו.
כדי לבצע זאת נעזר בפקודות הקפיצה המותנות הבאות:

- beq – פקודת branch if equal – נעזר בה כאשר נרצה לבחון תנאי if בכניסה ללולאה.
- bne – פקודת branch in not equal – נעזר בה באותו באופן בכניסה או יציאה מלולאה.
- beqz – פקודת branch if equal to zero – מאפשרת שימוש באוגר אחד כאשר תנאי קפיצה יתקיים אם ערכו שווה לאפס.
- blt – פקודת branch if less than – בה תנאי הקפיצה מתקיים אם ערכו של אוגר אחד קטן משל השני. גם בפקודה זו נשתמש בלולאות לפי הנוחיות שלנו.
- bgt – פקודת branch if greater than – דומה לפקודה הקודמת אך עם תנאי הפוך.

קיימות פקודות נוספות אשר אותן תוכלו לחקור, אך אלו הן הפקודות הטיפוסיות למימוש של לולאות מסוגים שונים לפי הצורך בקוד.

חשוב לציין כי ברוב המקרים, אין העדפה של פקודה אחת על פני השנייה, ולכן ההחלטה לבחור פקודת קפיצה מסוימת היא עליכם.

שלבים בכתיבה נכונה של לולאה:

כתיבה תקינה של לולאה צריכה להיות בנויה באופן הבא:

1) הגדרת המשתנים שהם האינדקסים הרצים בלולאות, או ערכי המשתנים שיש להשוות על מנת לבחון תנאי קפיצה.
2) ביצוע הפעולה שעל הלולאה לעשות.
3) קידום ערך האינדקס, לרוב בסוף הלולאה.

הערה:

אם בלולאה כללית (כגון לולאת for) יש לבדוק תנאים בטרם ביצוע פעולות, יש להיעזר בפקודות הקפיצה המותנות לפי בחירתכם על מנת ליצור את הבדיקה בטרם ביצוע שהלולאה צריכה לעשות ובטרם קידום האינדקס הרץ.

❖ דוגמה:

בקוד הבא מקבלים מערך Raw_Num של מספרים ויש לסכום את כולם.

נבצע זאת ע"י לולאה בשם loop באופן הבא:

```
.data
Raw_Num:    .word 1, 3, -11, 12, 13, 17, -18, 34   # array values
size:       .word 8                                # Store the size
                                                   # of the array

.text
main:
    la $t0, Raw_Num       # Load address of array into $t0
    lw $t2, size          # Load array size into $t2
    li $t1, 0             # Initialize counter i = 0
    li $t3, 0             # Initialize sum = 0

loop:
    beq $t1, $t2, exit    # Exit loop if i == size
    lw $t4, 0($t0)        # Load Raw_Num[i] into $t4
    add $t3, $t3, $t4     # sum += Raw_Num[i]
    addi $t0, $t0, 4      # Move to next element
    addi $t1, $t1, 1      # i++
    bne $t1, $t2, loop    # If i != size, continue loop

exit:
    move $a0, $t3         # Move sum to $a0 for printing
    li $v0, 1             # System call code for print integer
    syscall

    # Exit program
    li $v0, 10            # System call code for exit
    syscall
```

טבלת תווים ASCII:

לנוחיות וסידור ההתעסקות עם מחזורות ובנייה של לולאות עבורן, להלן טבלת ASCII.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

הערות כלליות:

1) נזכור כי כל תו מיוצג בעזרת 8 ביטים, או בית אחד.
למשל: התו '3' מיוצג ע"י 0x33 או 00110011.

2) בטעינה של תו לאוגר נעזר תמיד בפקודת lbu ובשמירה של תו חזרה למחזורת נעזר בפקודת sb.

3) כדי להמיר תו המייצג מספר לערכו במספר נחסר מערך ה-ASCII את הקבוע 0x30 או 48 בעשרוני, בעזרת הפקודה: addi, $t0, $t0, -48.

4) ניתן להיעזר בחוקיויות מתמטיות שונות כאשר אנו נדרשים להמיר תו אחד בתו אחר.
למשל נבחין כי טבלת ASCII בנויה כך שההפרש בין אות קטנה באנגלית לאות גדולה הוא בדיוק 0x20. כך שאם יש להפוך תו 'D' לתו 'd' נוכל פשוט לקחת את ערך ה-ASCII של 'D' להוסיף לו 0x20. נקבל: 0x44 + 0x20 = 0x64 שזה כמובן ערך ה-ASCII של 'd'.

דרך פתרון לשאלות נבחרות:

בוחן לדוגמה 1:

תכנון הלולאה:

בעת כתיבת קוד שמבצע לולאה נגדיר תחילה בעזרת pseudo-code את מה שנרצה לבצע:

```
Char buf[40]     # Allocation in memory
Char buf1[40]    # Allocation in memory
Char char        # char received from the user
int  i = 0       # index for outer loop
int  count = 0   # counting occurrences index

for (i=0 ; i < 40 ; i++)
{
   if (buf[i] === char)
      {
         if (count < 5)
         {
           buf1[i] = '*';
           count += 1;
         }
         else
         {
          buf1[i] = '@';
         }
      }
   else
      {
        buf1[i] = buf[i];
      }
}
```

מימוש הלולאה באסמבלי:

במימוש הלולאה יש לחשבן מספר דברים:

- הזזת התו המתקבל מן האוגר $v0 לאוגר זמני לצורך השימוש שלנו.
- הקצאת אוגרים לכל האינדקסים שיש להיעזר בהם במהלך ריצת הלולאות.
- טעינה ושמירה של תווים בודדים (פקודות: lbu ו-sb בלבד!)
- בחירת פקודות הקפיצה המותנות המתאימות למימוש הלולאות.

בוחן לדוגמה 2:

תכנון הלולאה:

```
Char buf[30]    # Allocation in memory
int  i = 0      # index for outer loop
int  sum = 0    # calculating the final sum of squares
int  square     # temp variable to store the square of a number
int  num = 0    # index to count the number of ASCII numbers

for (i=0 ; i < 30 ; i++)
{
   if (buf[i] >= '0' && buf[i] <= '9')
      {
        square = buf[i] * buf[i];
        sum += square;
        num += 1;
      }
}
```

מימוש הלולאה באסמבלי:

- נגדיר לולאה חיצונית עם אינדקס רץ שיהיה $t1.
- נגדיר שתי פקודות קפיצה שמדלגות על תהליך הריבוע והוספה במידה והערך אינו מייצג מספר.

בוחן לדוגמה 3:

תכנון הלולאה:

```
Char str[] = "Traveling to Cuba!"     # Allocation in memory
char  str1[]                          # empty string
int  inc   = 0    # increment index for outer loop
int  index = 0    # index for str1 string

for (inc=0 ; inc < length(str) ; inc+2)
{
    str1[index] = str[inc];
    str[inc] = '$';
    index += 1;
}
```

בוחן לדוגמה 4:

תכנון הלולאה:

```
int array[] = -4, 300, -600, -750, 1270, -200, 800, 900;
int array1[];

int  index = 0    # index for outer loop
int  temp         # variable to store temp values from array

for (index=0 ; index < length(array) ; index++)
{
    # check if value is divisible by 4:
    if (and(array[index], 3) == 0)
      {
         temp          = array[index] / 4;
         array1[index] = (-1) * temp;
      }
    else
      {
     array1[index] = (-1) * array[index];
      }
    # check if the current value is positive and even:
    if (array1[index] > 0) && (array1[index] mod 2 = 0)
      {
       printf(array1[index]);
      }

}
```

עיקרון חלוקה של מספר נתון בחזקה של 2:

לייצוג בינארי קיים יתרון מובהק בכל הקשור להכפלה וחלוקה של מספר פי חזקות של 2.
לדוגמה, המספר 5 נכתב: $[101]_2$ בעוד שהמספר 10 הוא: $[1010]_2$ והמספר 20 הוא: $[10100]_2$.
הזזה שמאלה ב- k סיביות של מספר בייצוג בינארי שקולה להכפלתו פי 2^k.

באופן דומה, הזזה ימינה ב- k סיביות משמעה חלוקה פי 2^k של מספר נתון.
לפי עיקרון זה נוכל לטעון מיד כי מספר שבו k סיביות ה-LSB הן אפסים, וודאי מתחלק פי 2^k.
למשל: המספר $[11010000]_2$ מתחלק פי 16 שכן $16 = 2^4$, והתוצאה היא: $[1101]_2 = 13$.

כדי לבדוק האם מספר מתחלק פי 2^k נוכל להיעזר **בפעולות לוגיות** (bitwise operations).
בפרט נעזר בפעולת AND.

למשל: כדי לבחון האם מספר מתחלק פי 8, נוכל לקחת את ערכו בפעולת AND עם הערך 7:

$$\begin{array}{r} \$t0 = [01101000\ 11001001\ 00000100\ 10100000]_2 \\ \$t1 = [00000000\ 00000000\ 00000000\ 00000111]_2 \\ \hline [00000000\ 00000000\ 00000000\ 00000000]_2 \end{array}$$

אם התוצאה היא אפס זהותית, סימן שתוצאת המסכה שיצרנו מעידה כי 3 סיביות ה-LSB של המספר הנתון ($\$t0$) הן אכן 0 ולכן המספר הנ"ל מתחלק פי 8 בוודאות.

דוגמה למספר שאינו מתחלק פי 8:

$$\begin{array}{r} \$t0 = [01101000\ 11001001\ 00000100\ 10100010]_2 \\ \$t1 = [00000000\ 00000000\ 00000000\ 00000111]_2 \\ \hline [00000000\ 00000000\ 00000000\ 00000010]_2 \end{array}$$

כעת, התוצאה שהתקבלה אינה אפס – לפחות אחת מ-3 הסיביות ה-LSB אינה אפס.
משמעות הדבר היא כי אם נבצע הזזה ב-3 סיביות ימינה, תהיה קיימת שארית לחלוקה זו.
מכאן כי המספר **אינו** מתחלק פי 8.