



## האסמבלר של ה - MIPS

שפת אסמבלי

במסמך זה נתאר כיצד לכתוב קוד בשפת אסמבלי באמצעות  
הסימולטור mars

מחבר: אושרי כהן

תאריך יצירה: 20:16 13/03/2012

תאריך עדכון אחרון: 26/05/2015 - 18:59

## תוכן

3	צורת העבודה עם מעבד ה MIPS
3	רגיסטרים
5	הוראות
6	האסמבלי של ה MIPS:
8	זיכרון
11	דוגמאות נוספות של הוראות
11	הוראות immediate
11	addi
11	הוראת li (pseudo שניתן לשימוש במעבדה)
12	הוראות הסתעפות
13	שימוש בהוראות הסתעפות בעזרת תוויות
14	הוראת beq
15	הוראת jr
15	הוראת la (pseudo שניתן לשימוש במעבדה)
17	directive - הנחיה לאסמבלר
21	הוראת load word
22	הוראת store word
23	הוראת mult
23	הוראת div
24	הוראות להעברת תוכן מרגיסטרים לרגיסטרים hi\lo
24	move from HI register – mfhi
24	move from LO register - mflo
24	move to HI register – mthi
24	move to LO register – mtlo
25	הוראת SLT
26	הוראת SLL
26	הוראת SRL
26	הוראות לביצוע פעולות לוגיות על רגיסטרים
27	הוראת LUI
28	נספח א – דוגמה לעבודה עם התוכנית הראשית main
29	נספח ב – חלוקת הקוד על פני יותר מקובץ אסמבלר אחד:

## צורת העבודה עם מעבד ה MIPS

### רגיסטרים

למעבד ה MIPS קיימים 32 רגיסטרים

Name	Number	Use
\$zero	\$0	constant 0
\$at	\$1	assembler temporary
\$v0-\$v1	\$2-\$3	values for function returns and expression evaluation
\$a0-\$a3	\$4-\$7	function arguments
\$t0-\$t7	\$8-\$15	temporaries
\$s0-\$s7	\$16-\$23	saved temporaries
\$t8-\$t9	\$24-\$25	temporaries
\$k0-\$k1	\$26-\$27	reserved for OS kernel
\$gp	\$28	global pointer
\$sp	\$29	<a href="#">stack pointer</a>
\$fp	\$30	<a href="#">frame pointer</a>
\$ra	\$31	<a href="#">return address</a>

מעבד ה MIPS משתמש ברגיסטרים לצורך ביצוע הוראות.

כל רגיסטר מכיל נתון בגודל של מילה (32 סיביות)

מתוך כל הרגיסטרים הנ"ל אסור לנו להשתמש ברגיסטר \$1 בהוראות שנכתוב – רגיסטר זה נמצא בשימוש של הקומפיילר בלבד.

כמו כן רגיסטר \$zero יכיל תמיד את הערך 0 ונוכל להשתמש בו בתנאי שאנחנו לא משנים אותו.

במהלך המעבדות הראשונות נשתדל להשתמש רק ברגיסטרים \$t0 - \$t7 (וכמובן ב \$zero בתור קבוע 0)

בפתרון התרגילים שיינתנו במעבדה חשוב מאוד לצורך הקריאות של הקוד להשתמש דווקא בכינוי של הרגיסטרים ולא בערך המספרי

לדוגמה יש לכתוב \$t0 בתוכנית ולא \$8 (למרות שיש צורך להבין ש \$t0 == \$8)

סדר השימוש ברגיסטרים:

חשוב להשתמש קודם ברגיסטרים \$t0-\$t7

אם אנחנו צריכים יותר רגיסטרים נשתמש גם ב \$t8 \$t9

אם עדיין נצטרך עוד רגיסטרים נשתמש ב \$s0 - \$s7

לשאר הרגיסטרים יש משמעות לעבודה נכונה עם פונקציות וביצוע פעולות מערכת (לדוגמה קלט, פלט) לכן נשתמש בהם רק כשנלמד

את השימוש המתאים להם.

עוד רגיסטרים:

בנוסף נראה בהמשך שיש עוד 3 רגיסטרים פרט ל 32 הרגיסטרים הנ"ל.

רגיסטר pc – מכיל את כתובת ההוראה הבאה שיש לבצע, לא ניתן להשתמש בו בהוראות.

רגיסטר hi ו lo – שימושיים עבור פעולות כפל וחילוק (נראה בהמשך), ניתן להשתמש בהם בקוד בצורה עקיפה בלבד.

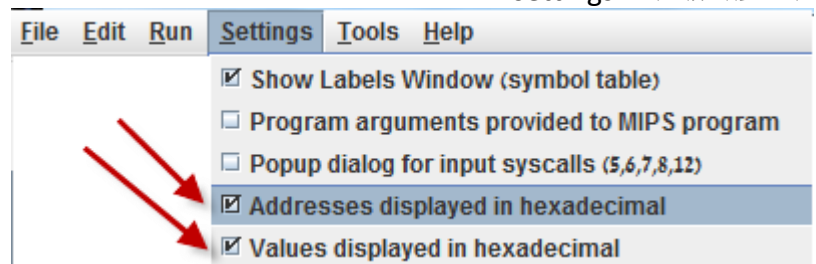
טבלת הרגיסטרים בסימולטור mars:

Registers	Coproc 1	Coproc 0
Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7ffffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

כפי שניתן לראות בעמודה הראשונה משמאל מופיע שם הרגיסטר בעמודה לאחריו מופיע מספר הרגיסטר (0-31) בעמודה הימנית ביותר מופיע תוכן הרגיסטר.

הערה: ניתן לראות את התוכן בצורה עשרונית או Hex

באמצעות התפריט settings



(נראה בהמשך דרך יותר נוחה לעשות זאת)

## הוראות

כל הוראה של mips כתובה בשפת אסמבלי.  
גודל של כל הוראה הוא בדיוק מילה אחת (32 סיביות)

כל הוראה מכילה משמאל לימין את שם הוראה בתחילה  
ולאחריה יופיעו רגיסטרים (מקסימום 3 רגיסטרים תלוי בסוג ההוראה)  
ולאו קבועים (גם תלוי בסוג ההוראה)

### הוראות אסמבלי תיקניות

באופן כללי קיימות לנו 54 הוראות חוקיות/תקניות בשפת אסמבלי, מתוכם 14 הוראות עבור מספרים בייצוג של נקודה צפה שבהן לא נשתמש במהלך הקורס.  
נשארו עמ' 40 הוראות שאיתם עלינו לפתור את כל המטלות של המעבדות.  
(הערה: בהמשך נוסיף עוד 2 הוראות שהן לא תקניות וסה"כ נוכל להשתמש ב 42 הוראות בקוד.)

### הוראות pseudo

למרות האמור לעיל אנו נראה כי הסימולטור mars מרשה לנו להשתמש בעוד הרבה הוראות נוספות.  
הוראות אלו נקראות pseudo ובזמן הידור האסמבלר מתרגם אותן להוראות אחרות (תקניות)  
**אין להשתמש בהן בפתרון המטלות במעבדה (פרט ל 2 הוראות שנרשה בהמשך)**

## האסמבלי של ה MIPS:

כאמור לעיל באסמבלי של ה MIPS קימות 54 הוראות מכונה, מתוכן 14 הוראות ייעודיות עבור מספרים עם נקודה צפה.

בטבלה הבאה מופיעות 40 ההוראות (ללא נקודה צפה)

CATEGORY	NAME	INSTRUCTION SYNTAX	MEANING
Arithmetic	Add	add \$d,\$s,\$t	$\$d = \$s + \$t$
	Add unsigned	addu \$d,\$s,\$t	$\$d = \$s + \$t$
	Subtract	sub \$d,\$s,\$t	$\$d = \$s - \$t$
	Subtract unsigned	subu \$d,\$s,\$t	$\$d = \$s - \$t$
	Add immediate	addi \$t,\$s,C	$\$t = \$s + C$ (signed)
	Add immediate unsigned	addiu \$t,\$s,C	$\$t = \$s + C$ (signed)
	Multiply	mult \$s,\$t	$LO = ((\$s * \$t) \ll 32) \gg 32$ ; $HI = (\$s * \$t) \gg 32$ ;
	Divide	div \$s, \$t	$LO = \$s / \$t$ $HI = \$s \% \$t$
	Divide unsigned	divu \$s, \$t	$LO = \$s / \$t$ $HI = \$s \% \$t$
Data Transfer	Load double word	ld \$t,C(\$s)	$\$t = \text{Memory}[\$s + C]$
	Load word	lw \$t,C(\$s)	$\$t = \text{Memory}[\$s + C]$
	Load halfword	lh \$t,C(\$s)	$\$t = \text{Memory}[\$s + C]$ (signed)
	Load halfword unsigned	lhu \$t,C(\$s)	$\$t = \text{Memory}[\$s + C]$ (unsigned)
	Load byte	lb \$t,C(\$s)	$\$t = \text{Memory}[\$s + C]$ (signed)
	Load byte unsigned	lbu \$t,C(\$s)	$\$t = \text{Memory}[\$s + C]$ (unsigned)
	Store double word	sd \$t,C(\$s)	$\text{Memory}[\$s + C] = \$t$
	Store word	sw \$t,C(\$s)	$\text{Memory}[\$s + C] = \$t$
	Store half	sh \$t,C(\$s)	$\text{Memory}[\$s + C] = \$t$
	Store byte	sb \$t,C(\$s)	$\text{Memory}[\$s + C] = \$t$
	Load upper immediate	lui \$t,C	$\$t = C \ll 16$
	Move from high	mfhi \$d	$\$d = HI$
	Move from low	mflo \$d	$\$d = LO$
	Move from Control Register	mfcZ \$t, \$d	$\$t = \text{Coproprocessor}[Z].\text{ControlRegister}[\$d]$
	Move to Control Register	mtcZ \$t, \$d	$\text{Coproprocessor}[Z].\text{ControlRegister}[\$d] = \$t$
Logical	And	and \$d,\$s,\$t	$\$d = \$s \& \$t$
	And immediate	andi \$t,\$s,C	$\$t = \$s \& C$
	Or	or \$d,\$s,\$t	$\$d = \$s   \$t$
	Or immediate	ori \$t,\$s,C	$\$t = \$s   C$
	Exclusive or	xor \$d,\$s,\$t	$\$d = \$s \wedge \$t$
	Nor	nor \$d,\$s,\$t	$\$d = \sim (\$s   \$t)$
	Set on less than	slt \$d,\$s,\$t	$\$d = (\$s < \$t)$
	Set on less than immediate	slti \$t,\$s,C	$\$t = (\$s < C)$
Bitwise Shift	Shift left logical	sll \$d,\$t,shamt	$\$d = \$t \ll \text{shamt}$
	Shift right logical	srl \$d,\$t,shamt	$\$d = \$t \gg \text{shamt}$
	Shift right arithmetic	sra \$d,\$t,shamt	$\$d = \$t \gg \text{shamt} + \left( \left( \sum_{n=1}^{\text{shamt}} 2^{31-n} \right) \cdot \$2 \gg 31 \right)$
Conditional branch	Branch on equal	beq \$s,\$t,C	if $(\$s == \$t)$ go to $PC+4+4*C$
	Branch on not equal	bne \$s,\$t,C	if $(\$s != \$t)$ go to $PC+4+4*C$
Unconditional jump	Jump	j C	$PC = PC+4[31:28] \cdot C*4$
	Jump register	jr \$s	goto address \$s
	Jump and link	jal C	$\$31 = PC + 8$ ; $PC = PC+4[31:28] \cdot C*4$

## דוגמאות:

### ביצוע חיבור

ב C++ הינו כותבים:

```
c = b + a;
```

בהוראת אסמבלי זה יראה כך:

בהנחה ש:

הערך של a נמצא ברגיסטר \$t2

הערך של b נמצא ברגיסטר \$t1

הערך של c נמצא ברגיסטר \$t0

```
ADD $t0, $t1, $t2
```

### ביצוע חיבור מורכב

ב C++ הינו כותבים:

```
d = b + a + c;
```

בהוראת אסמבלי זה יראה כך:

בהנחה ש:

הערך של a נמצא ברגיסטר \$t3

הערך של b נמצא ברגיסטר \$t2

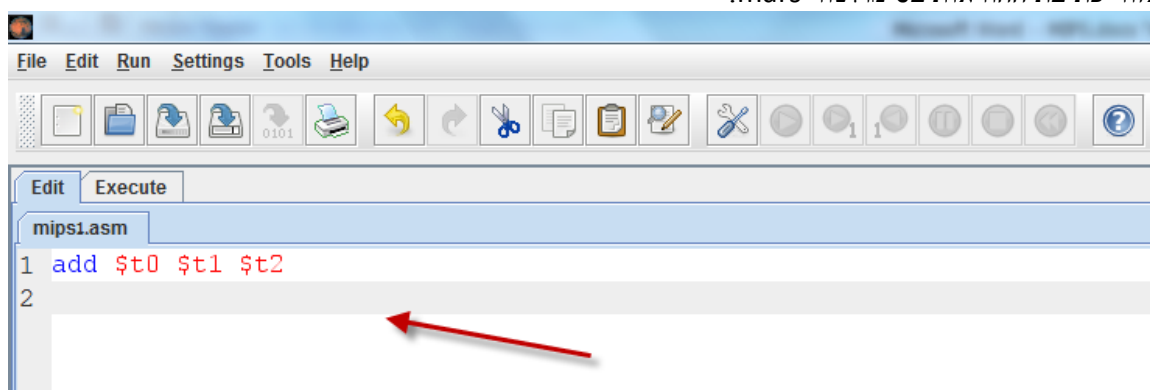
הערך של c נמצא ברגיסטר \$t1

הערך של d נמצא ברגיסטר \$t0

```
ADD $t4, $t1, $t2  
ADD $t0, $t3, $t4
```

שימו לב לשימוש ברגיסטר \$t4 על מנת לשמור את התוצאה הזמנית של החיבור  
האם היה ניתן לוותר עליו?

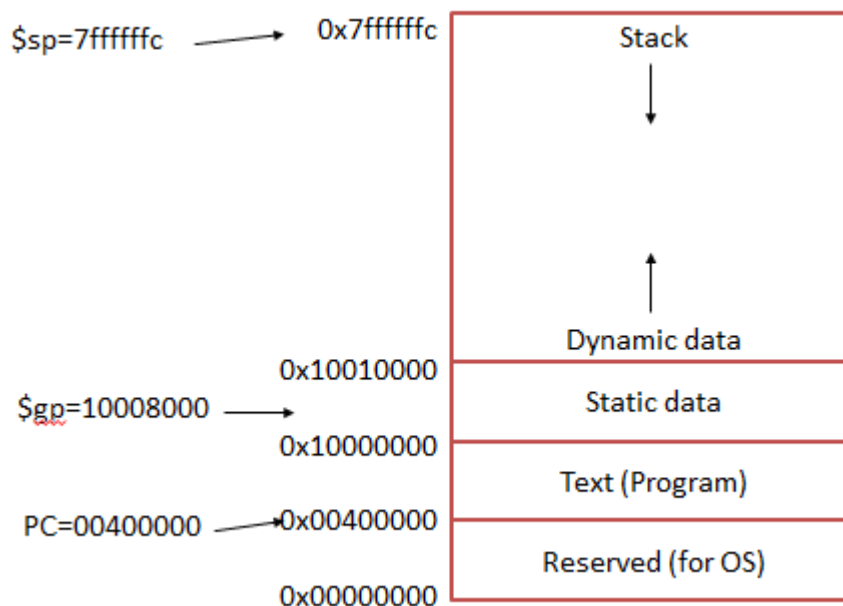
אזור כתיבת ההוראות בסימולטור mars:



## זיכרון


למעבד ה mips יכולת לגשת לזיכרון המורכב מ  $2^{32}$  כתובות שתוכן כל כתובת בית 1 (בית = 8 ביטים)

מפת הזיכרון של מעבד ה mips:

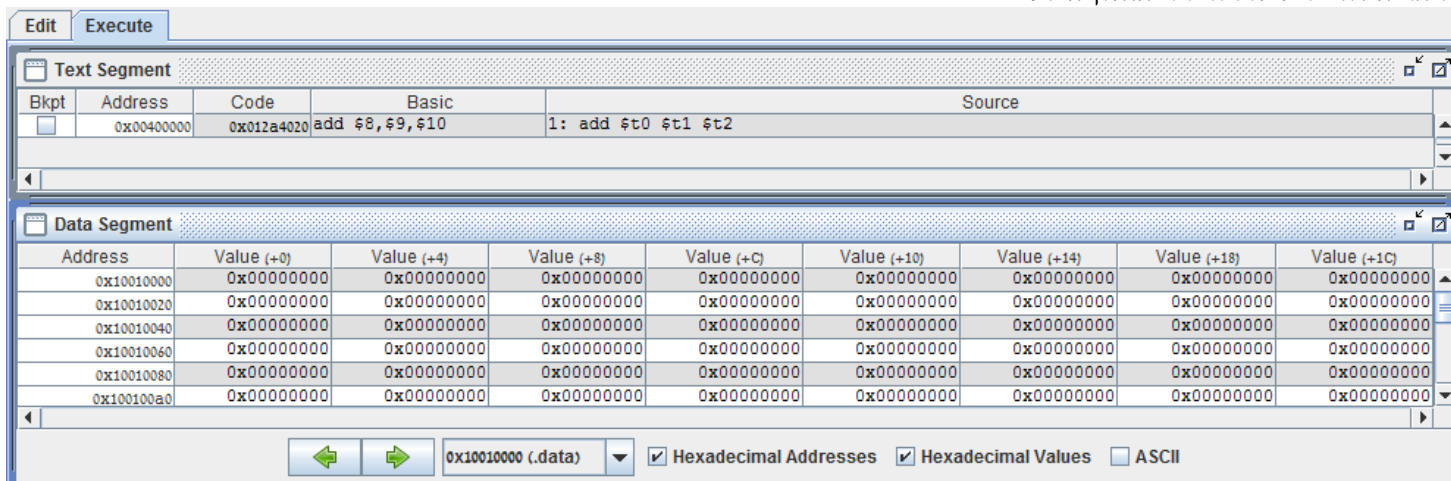


הערה:

הכתובות מופיעות ב Hex , כמובן שאם נכתוב את הכתובת בבינרי כל כתובת תכיל 32 סיביות. כל תוכן של כתובת הינו בגודל של בית (8 סיביות) כל הוראה באסמבלר גודלה 32 סיביות (מילה – 4 בתים) לאחר שהאסמבלר מבצע הידור של קוד האסמבלי, הוראות התוכנית ישמרו בזיכרון החל מכתובת  $0x00400000$  בכתובות בקפיצות של 4 כלומר ההוראה הבאה תישמר בכתובת  $0x00400004$

ביצוע הידור התוכנית בסימולטור מתבצע ע"י לחיצה על הכפתור  או ע"י לחיצה על F3 במקלדת. לפני ביצוע ההידור עלינו לשמור את הקובץ (רצוי לשמור עם סיומת asm על מנת שיהיה קל לפתוח אותו בעתיד)

לאחר הידור נוכל לראות את החלון הבא:



בחלון זה אנו רואים את אזור הזיכרון של התוכנית.



**Text Segment**

בחלק העליון מופיע זיכרון התוכנית (Text Segment)  
 כאמור לעיל זיכרון זה מכיל כתובות בטווח 0x00400000 - 0x10000000

Text Segment				
Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x012a4020	add \$8,\$9,\$10	1: add \$t0 \$t1 \$t2

הסבר:

בעמודת ה source כתובה ההוראה בדיוק כפי שכתבנו אותה  
 בעמודת ה basic כתובה ההוראה בשפת אסמבלי תקנית (לדוגמה אם כתבנו \$t0 אז כאן ייכתב \$8)  
 בעמודת ה code מופיע הקידוד של ההוראה ב Hex (32 סיביות בינריות)  
 בעמודת ה address כתובה הכתובת שבה הוראה זו נמצאת בזיכרון (כתובת זו חייבת להיות כפולה של 4)

הערה:

במידה וכתבנו הוראת pseudo, בעמודת ה basic נראה למה ההוראה תורגמה באמת.

**Data Segment**

בחלק התחתון מופיע אזור זיכרון הנתונים (Data Segment)

Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010180	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

בזיכרון הנתונים נוכל לראות את שאר אזורי הזיכרון (הנגישים)  
 נוכל לעבור בין אזורי זיכרון שונים באמצעות:

current sgp
0x10000000 (.extern)
current sgp
0x10010000 (.data)
0x10040000 (heap)
current ssp
0x90000000 (.kdata)
0xffff0000 (MMIO)

ובתוך אזורי הזיכרון השונים לעבור על כתובות נוספות באמצעות החצים.

## שימו לב

הסימולטור mars מציג את תוכן הכתובות בזיכרון בצורה של מילה (32 סיביות) להוראות אסמבלי זה כמובן מצוין כיוון שכל הוראה גם היא בגודל של מילה. ולכן כל תא מכיל בדיוק הוראה אחת.

ניתן לראות כתובות או ערכי כתובות בצורה Hex ע"י סימון

☒ Hexadecimal Addresses ☒ Hexadecimal Values

או בצורה עשרונית ע"י ביטול הסימון לעיל.

בהמשך נרחיב יותר על זיכרון הנתונים.

## דוגמאות נוספות של הוראות

### הוראות immediate

ניתן להשתמש בערך מידי בהוראה, הכוונה היא שהערך הוא חלק מקידוד ההוראה. בצורה זו ניתן להכניס ערך שהייצוג שלו לא גדול יותר מ 16 סיביות כיוון שכזכור כל הוראה הינה בגודל של 32 סיביות ולכן הוקצה מקום של 16 סיביות בלבד, עבור הוראות immediate

דוגמאות:

#### addi

לדוגמה:

addi \$t0 \$zero 12

הוראה זו מכניסה את הערך 12 לתוכן של רגיסטר \$t0

דוגמה נוספת:

addi \$t0 \$zero 12

addi \$t0 \$t0 3

לאחר ביצוע שתי ההוראות הנ"ל רגיסטר \$t0 יכיל את הערך 15

### הוראת li (pseudo שניתן לשימוש במעבדה)

הוראת pseudo ראשונה שנוכל להשתמש בה:

הוראה זו באה להחליף את ההוראה addi לצורך אתחול רגיסטר בערך מסוים. כלומר במקום לכתוב

addi \$t0 \$zero 12

רק על מנת שהערך 12 יוכנס לרגיסטר.

נוכל לכתוב

li \$t0 12

והאסמבלר יתרגם זאת להוראת addi תקנית.

Text Segment				
Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x2408000c	addiu \$8,\$0,12	1: li \$t0 12
<input type="checkbox"/>	0x00400004	0x21080003	addi \$8,\$8,3	2: addi \$t0 \$t0 3

הסבר:

ניתן לראות שהקוד שכתבנו היה:

li \$t0 12  
addi \$t0 \$t0 3

והאסמבלר תרגם את השימוש ב li ל addiu (ה u מצין unsigned - ללא סימן)

## הוראת קפיצה.

### jump

הוראה זו נכתבת בצורה הבאה

C - קבוע המציין כתובת שאליה יש לקפוץ.

בקוד אסמבלי תקני היה עלינו לכתוב את כתובת ההוראה עצמה שאליה נרצה לקפוץ

לדוגמה:

j 0x0040008

הסימולטור mars לא מאפשר לנו לבצע זאת, אלא אם נשתמש בתווית.

## שימוש בהוראות קפיצה בעזרת תוויות

בעזרת שימוש בתוויות נוכל לתת "כינוי" לכתובת מסוימת.  
לפני השורה שמתייחסת לאותה כתובת נגדיר שם כלשהו ולאחריו נקודתיים ':':

לדוגמה:

```
li $t0 12
j myLabel
addi $t0 $t0 3
myLabel:
```

בדוגמה זו הרגיסטר \$t0 יקבל את הערך 12 ומהשורה השנייה נקפוץ לסוף התוכנית לתווית myLabel

בזמן ההידור האסמבלר מבצע שני דברים.

- עובר על כל הקוד ומחפש את התוויות שהגדרנו, האסמבלר שומר בנפרד עבור כל תווית את הכתובת שבה היא נמצאת.
- עובר שוב על כל הקוד ובכל הוראה שמשתמשת בתווית הוא מתרגם את זה למספר לפי הצורך.

לדוגמה:

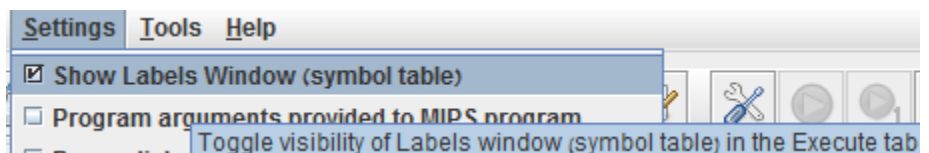
עבור הוראת jump הוא פשוט מעתיק את הכתובת שבה נמצאת התווית להוראה.  
כיוון ש jump מקבלת את כל הכתובת שאליה יש לקפוץ ללא תנאים.

Text Segment				
Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x2408000c	addiu \$8,\$0,12	1: li \$t0 12
<input type="checkbox"/>	0x00400004	0x08100003	j 0x0040000c	2: j myLabel
<input type="checkbox"/>	0x00400008	0x21080003	addi \$8,\$8,3	3: addi \$t0 \$t0 3

ניתן לראות שהוראה בשורה 2

j myLabel תורגמה להוראה j 0x0040000c (שזו בדיוק הכתובת של הוראה שתבוא אחרי ההוראה האחרונה )

ניתן לראות את כל התוויות שהגדרנו בקוד בחלונות Labels  
אם נסמן בתפריט Settings את האפשרות המתאימה.



חלון התוויות:

Labels	
Label	Address
mips1.asm	
myLabel	0x0040000c
<input checked="" type="checkbox"/> Data <input checked="" type="checkbox"/> Text	

## הוראת beq

מבנה ההוראה בצורה הבא:

beq \$rs, \$rt, C

ההוראה הזו פועלת באופן הבא:

אם התוכן של \$rs שווה לתוכן של \$rt ההוראה הבאה שתבצע היא ההוראה שנמצאת במיקום pc+C

באופן דומה ל jump גם עבור הוראת beq נוכל להשתמש בסימולטור רק באמצעות תוויות.

דוגמה:

```
li $t0 12
li $t1 12
beq $t0 $t1 myLabel
sub $t2 $t0 $t1
myLabel: add $t2 $t0 $t1
```

התוצאה לאחר ביצוע אסמבלר:

Text Segment					Labels	
Bkpt	Address	Code	Basic	Source	Label	Address
<input type="checkbox"/>	0x00400000	0x2408000c	addiu \$8,\$0,12	1: li \$t0 12	mips1.asm	
<input type="checkbox"/>	0x00400004	0x2409000c	addiu \$9,\$0,12	2: li \$t1 12		
<input type="checkbox"/>	0x00400008	0x11090001	beq \$8,\$9,1	3: beq \$t0 \$t1 myLabel	myLabel	0x00400010
<input type="checkbox"/>	0x0040000c	0x01095022	sub \$10,\$8,\$9	4: sub \$t2 \$t0 \$t1		
<input type="checkbox"/>	0x00400010	0x01095020	add \$10,\$8,\$9	5: myLabel: add \$t2 \$t0 \$t1		

שימו לב:

בשונה מהוראת jump, הקבוע בהוראת beq מציין כמה הוראות לקפוץ (מספר שלילי מציין קפיצה אחורה) כלומר עבור הוראת beq האסמבלר מחליף תוויות בקבוע באופן הבא:

הוא מבצע חיסור בין הכתובת שבה נמצאת התווית לכתובת שבה נמצאת ההוראה שאחרי הוראת beq (כזכור בעת ביצוע הוראת beq ה pc מקודם כבר ב 4) את התוצאה הוא כמובן מחלק ב 4 כיוון שכל הוראה תופסת 4 בתים.

כדאי גם לשים לב שבמקרה והתנאי לא מתקיים תבוצע שורה 4 וגם שורה 5 .

אם נרצה שבמקרה שהתנאי לא מתקיים תבוצע שורה 4 ונדלג על שורה 5.

נוכל לכתוב זאת כך:

```
li $t0 12
li $t1 12
beq $t0 $t1 myLabel
sub $t2 $t0 $t1
j myExit
myLabel: add $t2 $t0 $t1
myExit:
```

כלומר נוסף תווית אחרי שורה 5 ופשוט נקפוץ אליה לאחר ביצוע שורה 4

## הוראת jr

jr \$rs

הוראה זו מבצעת קפיצה לכתובת ההוראה שנמצאת בתוכן של רגיסטר \$rs

לדוגמה:

```
li $t0 12
li $t1 12
li $t3 0x00400020
beq $t0 $t1 myLabel
sub $t2 $t0 $t1
jr $t3
myLabel: add $t2 $t0 $t1
:myExit
```

הכתובת 0x00400020 זו במקרה הכתובת שבה ההוראה add \$t2 \$t0 \$t1 נמצאת ולכן התוכנית עובדת כמו בדוגמה הקודמת. לאחר ביצוע האסמבלי הקוד ישמר כך:

Text Segment				
Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x2408000c	addiu \$8,\$0,12	1: li \$t0 12
<input type="checkbox"/>	0x00400004	0x2409000c	addiu \$9,\$0,12	2: li \$t1 12
<input type="checkbox"/>	0x00400008	0x3c010040	lui \$1,64	3: li \$t3 0x00400020
<input type="checkbox"/>	0x0040000c	0x342b0020	ori \$11,\$1,32	
<input type="checkbox"/>	0x00400010	0x11090002	beq \$8,\$9,2	4: beq \$t0 \$t1 myLabel
<input type="checkbox"/>	0x00400014	0x01095022	sub \$10,\$8,\$9	5: sub \$t2 \$t0 \$t1
<input type="checkbox"/>	0x00400018	0x01600008	jr \$11	6: jr \$t3
<input type="checkbox"/>	0x0040001c	0x01095020	add \$10,\$8,\$9	7: myLabel: add \$t2 \$t0 \$t1

הערה:

בשורה 3 ההוראה li \$t3 0x00400020 חולקה ל 2 הוראות כיוון שהערך 0x00400020 מכיל יותר מ 16 סיביות. זה יוסבר בהרחבה בהמשך כשנלמד על ההוראה lui

## הוראת la (pseudo) שניתן לשימוש במעבדה

תפקידה של הוראה זו היא לטעון כתובת רגיסטר.

בדומה למה שעשינו לעיל

li 0x00400020

היינו יכולים להשתמש בהוראה la שיכולה לקבל תווית:

```
li $t0 12
li $t1 12
la $t3 myLabel
beq $t0 $t1 myLabel
sub $t2 $t0 $t1
jr $t3
myLabel: add $t2 $t0 $t1
myExit:
```

שימו לב

למרות שהוראת la יכולה לקבל גם מספר ולתפקד כמו li זה לא יהיה נכון לוגית להשתמש בה על מנת לאתחל רגיסטר במספר שהוא לא כתובת.





## directive - הנחיה לאסמבלר

ניתן להשתמש ב directive עבור הנחיות לאסמבלר.  
כל directive מתחיל בנקודה '!'.

בטבלה הבאה מופעים ההנחיות שקימות באסמבלר של ה Mips

.align	Align the next datum on a $2^n$ byte boundary. For example, .align 2 aligns the next value on a word boundary. .align 0 turns off automatic alignment of .half, .word, .float, and .double directives until the next .data or .kdata directive.
.ascii	Store the string in memory, but do not null-terminate it.
.asciiz	Store the string in memory and null-terminate it.
.byte	Store the $n$ values in successive bytes of memory.
.data	The following data items should be stored in the data segment. If the optional argument <i>addr</i> is present, the items are stored beginning at address <i>addr</i> .
.double	Store the $n$ floating point double precision numbers in successive memory locations.
.extern	Declare that the datum stored at sym is size bytes large and is a global symbol. This directive enables the assembler to store the datum in a portion of the data segment that is efficiently accessed via register \$gp.
.float	Store the $n$ floating point single precision numbers in successive memory locations.
.globl	Declare that symbol sym is global and can be referenced from other files.
.half	Store the $n$ 16-bit quantities in successive memory halfwords.
.kdata	The following data items should be stored in the kernel data segment. If the optional argument <i>addr</i> is present, the items are stored beginning at address <i>addr</i> .
.ktext	The next items are put in the kernel text segment. In SPIM, these items may only be instructions or words (see the .word directive below). If the optional argument <i>addr</i> is present, the items are stored beginning at address <i>addr</i> .
.space	Allocate $n$ bytes of space in the current segment (which must be the data segment in SPIM).
.text	The next items are put in the user text segment. In SPIM, these items may only be instructions or words (see the .word directive below). If the optional argument <i>addr</i> is present, the items are stored beginning at address <i>addr</i> .
.word	Store the $n$ 32-bit quantities in successive memory words. SPIM does not distinguish various parts of the data segment (.data, .rdata and .sdata).

הנחיות לשמירת הקוד בזיכרון

.text – מנחה את האסמבלר לשמור את כל מה שבא לאחריו בזיכרון התוכנית (הוראות).

.data – מנחה את האסמבלר לשמור את כל מה שבא לאחריו בזיכרון הנתונים.

לשתי הנחיות אלו יש כתובת וגודל נתון ברירת מחדל שממנה יתחיל האסמבלר לשמור את הנתונים.

עבור .text. כתובת ברירת המחדל היא הכתובת הבאה הפנויה החל מ 0x00400000 וגודל נתון הינו מילה.

עבור .data. כתובת ברירת המחדל היא הכתובת הבאה הפנויה החל מ 0x10010000 וגודל נתון הינו מילה.

גודל הנתון:

עבור הוראות (.text) לא ניתן לשנות את גודל הנתון.

עבור זיכרון הנתונים (.data) ניתן לשנות את גודל הנתון ע"י שימוש באחת מההנחיות הבאות:

.ascii	Store the string in memory, but do not null-terminate it.
.asciiz	Store the string in memory and null-terminate it.
.byte	Store the $n$ values in successive bytes of memory.
.double	Store the $n$ floating point double precision numbers in successive memory locations.
.float	Store the $n$ floating point single precision numbers in successive memory locations.
.half	Store the $n$ 16-bit quantities in successive memory halfwords.
.word	Store the $n$ 32-bit quantities in successive memory words. SPIM does not distinguish various parts of the data segment (.data, .rdata and .sdata).

הערה:

כפי שראינו עד עכשיו – אם לא הגדרנו אף הנחיה כברירת מחדל שמירת הקוד נעשתה לזיכרון התוכנית.

דוגמה:

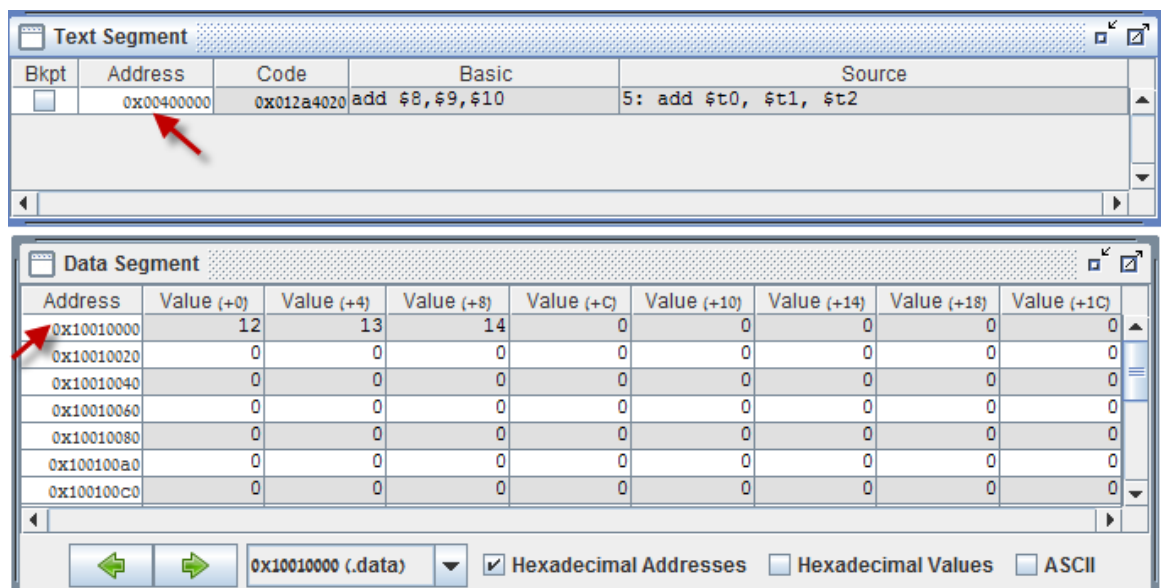
```
.data
12, 13, 14

.text
add $t0, $t1, $t2
```

הסבר:

הנתונים 12 13 14 ישמרו בגודל מילה לזיכרון התוכנית החל מכתובת 0x10010000

וההוראה add \$t0 \$t1 \$t2 תישמר לזיכרון ההוראות החל מכתובת 0x00400000



כדאי לשים לב שבצורה יותר נכונה היה כדאי לרשום זאת בפירוש כך:

```
.data 0x10010000
.word 12, 13, 14

.text 0x00400000
add $t0, $t1, $t2
```

אם נגדיר כך:

```
.data 0x10010000
.word 12, 13, 14

.text 0x00400000
add $t0, $t1, $t2

.data
.word 15, 16, 17

.text
sub $t0, $t1, $t2
```

המספרים 15 ו 16 ו 17 ישמרו מיד לאחר סיום כתיבת המספרים 12 13 14 (ולא ידרסו אותם)  
כנ"ל לגבי ההוראה sub \$t0, \$t1, \$t2

Text Segment				
Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x012a4020	add \$8,\$9,\$10	5: add \$t0, \$t1, \$t2
<input type="checkbox"/>	0x00400004	0x012a4022	sub \$8,\$9,\$10	11: sub \$t0, \$t1, \$t2

Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	12	13	14	15	16	17	0	0
0x10010020	0	0	0	0	0	0	0	0
0x10010040	0	0	0	0	0	0	0	0
0x10010060	0	0	0	0	0	0	0	0
0x10010080	0	0	0	0	0	0	0	0
0x100100a0	0	0	0	0	0	0	0	0
0x100100c0	0	0	0	0	0	0	0	0

דוגמה נוספת:

```
.data 0x10010020
.word 12, 13, 14

.text 0x00400000
add $t0, $t1, $t2

.data
.word 15, 16, 17

.text
sub $t0, $t1, $t2
```

Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0	0	0	0	0	0	0	0
0x10010020	12	13	14	15	16	17	0	0
0x10010040	0	0	0	0	0	0	0	0
0x10010060	0	0	0	0	0	0	0	0
0x10010080	0	0	0	0	0	0	0	0
0x100100a0	0	0	0	0	0	0	0	0
0x100100c0	0	0	0	0	0	0	0	0

הערות:

שימו לב שלמרות שניתן לשנות את מיקום שמירת הוראות התוכנית, אין זה שימושי כיוון שאת ה Pc אנחנו לא יכולים לשנות! שימו לב שהסימולטור מציג לנו בצורה יפה נתונים בגודל מילה ולכן ברוב התוכניות נשתמש בנתונים בגודל מילה.

שימוש בתוויות עבור זיכרון הנתונים:

כפי שהשתמשנו בתוויות עבור כתובות בזיכרון התוכנית ניתן להשתמש בתוויות גם עבור כתובות בזיכרון הנתונים.

דוגמה:

```
.data 0x10010020
lab1:
.word 12
lab2: 13, 14
lab3:

.text 0x00400000
lab4: add $t0, $t1, $t2

.data
.word 15, 16, 17
lab5:

.text
lab6: sub $t0, $t1, $t2
```

Text Segment					Labels			
Bkpt	Address	Code	Basic	Source	Label	Address		
	0x00400000	0x012a4020	add \$8,\$9,\$10	8: lab4: add \$t0, \$t1, \$t2	lab4	0x00400000		
	0x00400004	0x012a4022	sub \$8,\$9,\$10	15: lab6: sub \$t0, \$t1, \$t2	lab6	0x00400004		

Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0	0	0	0	0	0	0	0
0x10010020	12	13	14	15	16	17	0	0
0x10010040	0	0	0	0	0	0	0	0

הוראת load word

הוראה זו טוענת מילה מזיכרון הנתונים לרגיסטר

lw \$rt C(\$rs)

דוגמה:

```
.data 0x10010000
mem:
.word 12, 13, 14

.text
la $t0 mem
lw $t1 0($t0)
lw $t2 4($t0)
add $t3 $t1 $t2
```

הסבר:

בתחילה האסמבלר שומר בזיכרון הנתונים את הערכים 12 13 14 (כל אחד בגודל מילה) ואת ההוראות כרגיל בזיכרון התוכנית.

בהרצת התוכנית:

הכתובת של התווית mem נכנסת לרגיסטר \$t0

לרגיסטר \$t1 נטען הערך שנמצא בכתובת שברגיסטר \$t0 בהיסט של 0

כלומר נטען אליו הערך 12

לרגיסטר \$t2 נטען הערך שנמצא בכתובת שברגיסטר \$t0 בהיסט של 4

כלומר הערך שנמצא בכתובת  $0x10010000+4 = 0x10010004$

שזה 13

ובסוף מבוצע חיבור והתוצאה נשמרת ברגיסטר \$t3

דוגמה נוספת:

```
.data 0x10010000
mem:
.word 1, 2, 3, 4, 5, 6
endMem:

.text
la $t0 mem
la $t3 endMem
li $t2 0

loop:
lw $t1 0($t0)
add $t2 $t2 $t1
addi $t0 $t0 4
bne $t0 $t3 loop
```

הסבר:

התוכנית סוכמת את כל הנתונים החל מהנתון שנמצא בכתובת mem ועד לנתון שנמצא לפני הכתובת memEnd

סכום הנתונים נשמר ברגיסטר \$t2

הוראת store word

הוראה זו טוענת מילה מרגיסטר לזיכרון הנתונים

sw \$rt C(\$rs)

לדוגמה:

```
.data 0x10010000
mem:
.word 1, 2, 3, 4, 5, 6
endMem:

.text
la $t0 mem
la $t3 endMem
li $t2 0

loop:
lw $t1 0($t0)
add $t2 $t2 $t1
addi $t0 $t0 4
bne $t0 $t3 loop
sw $t2 0($t3)
```

הסבר:

כמו קודם רק שהפעם בסוף התוכנית אנו שומרים את התוצאה בכתובת memEnd

הוראת mult

הוראה זו מבצעת כפל.

מכיוון שכפל של 2 מספרים בני 32 סיביות כל אחד יכול לתת מספר בעל 64 סיביות קיימים לנו רגיסטרים hi ו lo שכל אחד מהם מכיל נתון בגודל 32 סיביות ושניהם יחד מכילים את התוצאה באופן הבא:

hi	lo
----	----

לדוגמה:

```
li $t0 4
li $t1 3
mult $t0 $t1
```

התוצאה:

hi		0
lo		12

כלומר המספר הוא 12

0	12
---	----

דוגמה נוספת:

```
li $t0 0x01000000
li $t1 0x01000000
mult $t0 $t1
```

התוצאה:

hi		0x00010000
lo		0x00000000

10000	00000000
-------	----------

התוצאה של  $2^{24} * 2^{24} = 2^{48}$

הוראת div

הוראה זו מבצעת חילוק.

תוצאת החילוק נשמרת ברגיסטרים hi ו lo באופן הבא  
רגיסטר lo יכיל את התוצאה ללא שארית.  
רגיסטר hi יכיל את השארית.

לדוגמה:

```
li $t0 5
li $t1 2
div $t0 $t1
```

התוצאה:

hi		1
lo		2

## הוראות להעברת תוכן מרגיסטרים\לרגיסטרים hi\lo

### העברת התוכן מרגיסטרים lo\hi לרגיסטר אחר:

#### move from HI register – mfhi

פורמט ההוראה: mfhi \$rd  
הוראה זו מעבירה לרגיסטר \$rd את התוכן שנמצא ברגיסטר hi

#### move from LO register - mflo

פורמט ההוראה: mflo \$rd  
הוראה זו מעבירה לרגיסטר \$rd את התוכן שנמצא ברגיסטר lo

### העברת התוכן מרגיסטר מסויים לרגיסטרים lo\hi:

#### move to HI register – mthi

פורמט ההוראה: mthi \$rs  
הוראה זו מעבירה לרגיסטר hi את התוכן שנמצא ברגיסטר \$rs

#### move to LO register – mtlo

פורמט ההוראה: mtlo \$rs  
הוראה זו מעבירה לרגיסטר lo את התוכן שנמצא ברגיסטר \$rs

לדוגמה:

נרצה לשמור את ספרת האחדות של מספר הנתון ברגיסטר \$t0 למיקום בזיכרון:

```
.data  
mem:  
  
.text  
li $t0 1234  
li $t1 10  
la $t2 mem  
  
div $t0 $t1  
mfhi $t1  
sw $t1 0($t2)
```

התוצאה הסופית בזיכרון:

Data Segment					
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+1)
0x10010000	4	0	0	0	0
0x10010020	0	0	0	0	0
0x10010040	0	0	0	0	0

### הסבר:

התוכנית מחלקת 10 את הנתון שנמצא ברגיסטר \$t0, השארית של החלוקה היא ספרת האחדות ומאוכסנת ברגיסטר hi  
בעזרת ההוראה mfhi אנחנו מעבירים את תוצאת השארית לרגיסטר \$t1 על מנת שנוכל לשמור אותו בזיכרון בעזרת פקודת lw



## הוראת SLT

פורמט ההוראה `slt $rd $rs $rt`  
ההוראה מבצעת בדיקה אם התוכן שב `$rs` קטן ממש מהתוכן של `$rt`  
במידה והתוכן קטן ממש אז רגיסטר `$rd` מקבל את הערך 1  
אחרת רגיסטר `$rd` מקבל את הערך 0

דוגמה:

```
li $t0 2 # min number
li $t1 5 # max number
li $t2 0 # count

loop:
slt $t3 $t0 $t1
beq $t3 $zero endLoop
add $t2 $t2 $t0
addi $t0 $t0 1
j loop
endLoop:
```

הסבר:

בדוגמה זו אנו סוכמים את כל המספרים מהערך שמצא ברגיסטר `$t0` ועד לערך שנמצא ברגיסטר `$t1`  
(לא כולל הערך שנמצא ברגיסטר `$t1`)  
בדוגמה זו הערך הסופי ב `$t2` יהיה שווה ל  $2+3+4 = 9$

הערה:

בעזרת הפקודות `slt`, `beq`, `bne` ובעזרת רגיסטר `$0` ועוד רגיסטר עזר נוסף ניתן ליצור עוד פקודות כגון:  
אם גדול מ...  
אם קטן מ...  
אם גדול שווה ל...  
אם קטן שווה ל...

לדוגמה:

אם נרצה להגדיר את התנאי גדול מ... (כלומר עם ערך של רגיסטר מסוים גדול מערך של רגיסטר אחר אז נקפוץ למיקום מסוים)  
הכוונה ל:

```
loop:
if( $t0 > $t1 ) => loop
```

בקוד אסמבלי נשתמש ב:

```
loop:
slt $t3 $t1 $t0
bne $t3 $zero loop
```

שימו לב שהבדיקה אם `$t0` גדול ממש מ `$t1` היא הבדיקה אם `$t1` קטן ממש מ `$t0`

## הוראת SLL

פורמט ההוראה  $\$rd, \$rt, C$   $\$ll$

לדוגמה:

```
li $t0 1
sll $t1 $t0 2
```

בסיום ההרצה הערך של  $t_1$  יהיה 4

## הסבר:

בתחילה רגיסטר  $\$t0$  מכיל את הערך:

[illegible]

לרגיסטר \$t1\$ אנו מכניסים את הערך של \$t1\$ מוזז ב 2 סיביות שמאלה

[illegible]

בדוגמה זו רואים שההזזה שוות ערך לכפולה בחזקה של 2 (כל עוד לא הורדנו אחדות ב MSB של המספר)

במילים אחרות עבור ביצוע הוראה  $\$rd, \$rt, C$

**הוראת SRL**

פורמט ההוראה  $\$rd, \$rt, C$

## הוראות לביצוע פעולות לוגיות על רגיסטרים

and

or

andi

ori

LUI הוראת

פורמט ההוראה: lui \$rd, C

הוראה זו טוענת את הערך C ל 16 סיביות ה MSB של הרגיסטר \$rd וב 16 סיביות LST של הרגיסטר מוכנס 0 (בצורה מתמטית הערך שיהיה ברגיסטר \$rd זהה לערך  $C \cdot 2^{16}$ )

הסבר:

בכל הוראה מסוג I שדה ה immediate הינו 16 סיביות.

בכל הוראה רגילה ההתאמה של ערך ה immediate לתבנית של 32 סיביות הינה להוסיף אפסים או אחדות מובילים משמאל. בהוראה זו ההתאמה של ערך ה immediate ל 32 סיביות נעשית ע"י הוספת אפסים מימין לסיבית ה 15 במספר.

דוגמה:

הוראת addi \$to, \$zero, 0x0101

כידוע גורמת לכך שברגיסטר \$t0 הנתון יהיה :

	MSB 16 סיביות				LSB 16 סיביות			
0 x	0	0	0	0	0	1	0	1

לעומת זאת

הוראת lui \$to, 0x0101

כידוע גורמת לכך שברגיסטר \$t0 הנתון יהיה :

	MSB 16 סיביות				LSB 16 סיביות			
0 x	0	1	0	1	0	0	0	0

הערה:

בסימולטור mars לא ניתן להשתמש במספר שלילי עבור שדה ה immediate בהוראה LUI

## נספח א – דוגמה לעבודה עם התוכנית הראשית main

נוכל להגדיר בסימולטור mars שה pc יתחיל מהתוכנית הראשית main (במידה והגדרנו תווית גלובאלית כזו)

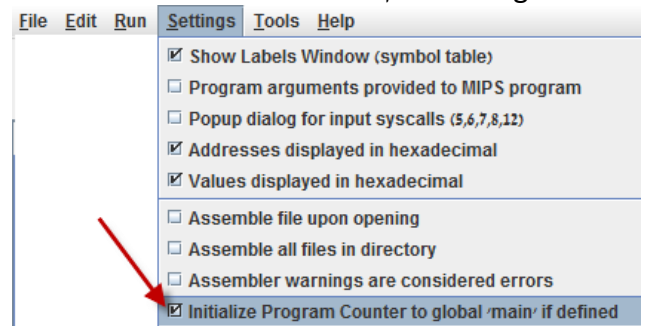
לדוגמה נכתוב את הקוד הבא:

```
.globl main

.text
sum:
    add $v0 $a0 $a1
    jr $ra

main:
    addi $a0 $zero 2
    addi $a1 $zero 3
    jal sum
    add $a0 $zero $v0
    addi $v0 $zero 1
    syscall
```

ובתפריט settings נסמן:



כעת אם נבצע אסמבלר על הקוד נראה שה pc ממוקם בדיוק בכתובת שבה כתבנו את התווית main

The screenshot displays the Mars simulator interface. The 'Text Segment' window shows the assembly code with the following instructions and addresses:

Bkpt	Address	Code	Basic	Source
	0x00400000	0x00851020	add \$2,\$4,\$5	5: add \$v0 \$a0 \$a1
	0x00400004	0x03e00008	jr \$31	6: jr \$ra
	0x00400008	0x20040002	addi \$4,\$0,0x00000002	9: addi \$a0 \$zero 2
	0x0040000c	0x20050003	addi \$5,\$0,0x00000003	10: addi \$a1 \$zero 3
	0x00400010	0x0c100000	jal 0x00400000	11: jal sum
	0x00400014	0x00022020	add \$4,\$0,\$2	12: add \$a0 \$zero \$v0
	0x00400018	0x20020001	addi \$2,\$0,0x00000001	13: addi \$v0 \$zero 1
	0x0040001c	0x0000000c	syscall	14: syscall

The 'Labels' window shows the following labels and addresses:

Label	Address
(global)	
main	0x00400008
mips1.asm	
sum	0x00400000

The 'Registers' window shows the following registers and values:

Name	Number	Value
\$zero	0	0x00000000
pc		0x00400008
hi		0x00000000
lo		0x00000000

A red arrow points from the 'pc' register to the address 0x00400008, which corresponds to the 'main' label in the 'Labels' window.

## נספח ב – חלוקת הקוד על פני יותר מקובץ אסמבלר אחד:

ניצור תיקייה בשם כלשהו נניח "myProgram"  
בתיקייה זו ניצור קובץ asm בשם כלשהו נניח "func.asm" ובו נכתוב את הקוד הבא:

```
.globl sum

.text
sum: add $v0 $a0 $a1
      jr $ra
```

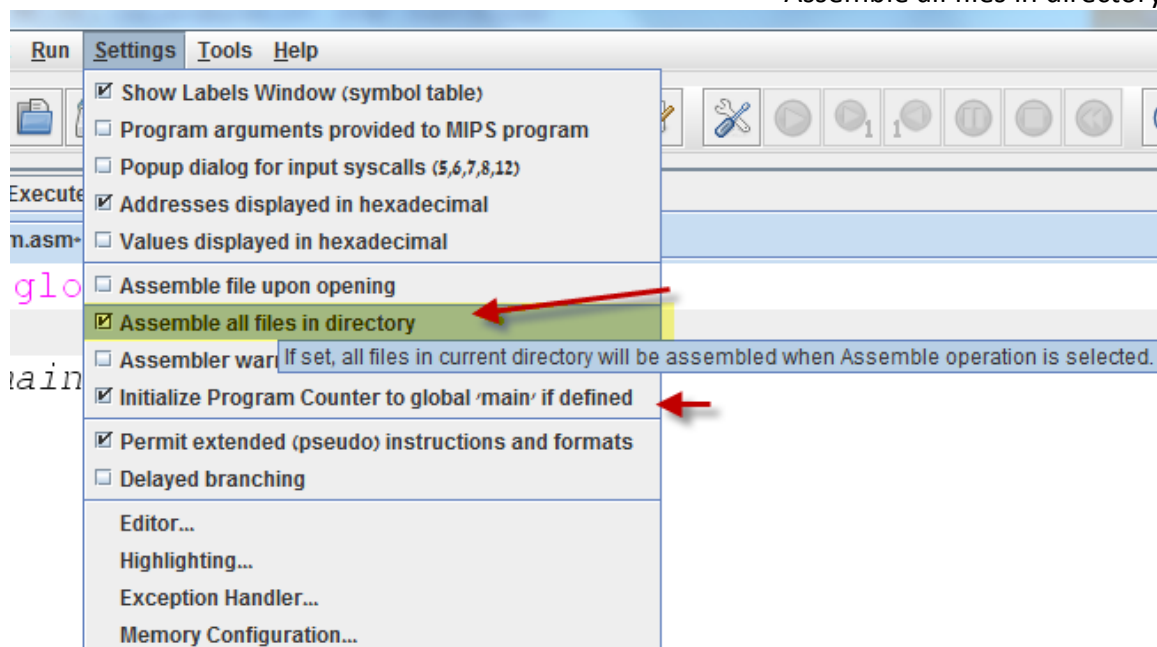
הסבר:  
הגדרנו פונקציה בשם sum והצהרנו על התווית sum שהיא גלובלית – כלומר יהיה ניתן להשתמש בה מקובץ אסמבלי אחר.

כעת ניצור קובץ נוסף באותה תיקיה בעל שם אחר נניח "program.asm" ובו נכתוב את הקוד הבא:

```
.globl main

main: addi $a0 $zero 2
      addi $a1 $zero 3
      jal sum
      add $a0 $zero $v0
      addi $v0 $zero 1
      syscall
```

ובהגדרות נסמן את האפשרות  
Assemble all files in directory



כעת אם נבצע אסמבלר של הקוד נקבל אחת משתי התוצאות:

אם בזמן ביצוע האסמבלר הקובץ "func.asm" היה פעיל נקבל:

Text Segment				
Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x00851020	add \$2,\$4,\$5	4: sum: add \$v0 \$a0 \$a1
<input type="checkbox"/>	0x00400004	0x03e00008	jr \$31	5: jr \$ra
<input type="checkbox"/>	0x00400008	0x20040002	addi \$4,\$0,2	3: main: addi \$a0 \$zero 2
<input type="checkbox"/>	0x0040000c	0x20050003	addi \$5,\$0,3	4: addi \$a1 \$zero 3
<input type="checkbox"/>	0x00400010	0x0c100000	jal 0x00400000	5: jal sum
<input type="checkbox"/>	0x00400014	0x00022020	add \$4,\$0,\$2	6: add \$a0 \$zero \$v0
<input type="checkbox"/>	0x00400018	0x20020001	addi \$2,\$0,1	7: addi \$v0 \$zero 1
<input type="checkbox"/>	0x0040001c	0x0000000c	syscall	8: syscall

וניתן להריץ את הקוד כרגיל.

אבל אם בזמן ביצוע האסמבלר הקובץ "program.asm" היה פעיל נקבל:

Text Segment				
Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x20040002	addi \$4,\$0,2	3: main: addi \$a0 \$zero 2
<input type="checkbox"/>	0x00400004	0x20050003	addi \$5,\$0,3	4: addi \$a1 \$zero 3
<input type="checkbox"/>	0x00400008	0x0c100006	jal 0x00400018	5: jal sum
<input type="checkbox"/>	0x0040000c	0x00022020	add \$4,\$0,\$2	6: add \$a0 \$zero \$v0
<input type="checkbox"/>	0x00400010	0x20020001	addi \$2,\$0,1	7: addi \$v0 \$zero 1
<input type="checkbox"/>	0x00400014	0x0000000c	syscall	8: syscall
<input type="checkbox"/>	0x00400018	0x00851020	add \$2,\$4,\$5	4: sum: add \$v0 \$a0 \$a1
<input type="checkbox"/>	0x0040001c	0x03e00008	jr \$31	5: jr \$ra

ובמצב כזה אם נריץ נקבל לולאה אין סופית.

### הפתרון:

לשים בסוף התוכנית הראשית את הקוד:

```
li $v0 10
syscall
```

שגורם לסיום התוכנית בשורה זו.

(אפשר לשים גם בסוף כל קובץ אבל במקרה של פונקציות תמיד נסיים עם jr \$ra אז אין לזה משמעות)

הקוד המלא:

```
.globl main

main: addi $a0 $zero 2
      addi $a1 $zero 3
      jal sum
      add $a0 $zero $v0
      addi $v0 $zero 1
      syscall

li $v0 10
syscall
```

### בנוסף חשוב לשים לב

רק הקובץ הפעיל נשמר אוטומטית בזמן הקימפול, שאר הקבצים שנמצאים בתיקיה יציגו את הקוד שבהם עד לשמירה האחרונה !!!

הערה:

- במצב כזה נוכל להגדיר תוויות דומות (בתנאי שאינן גלובליות) בשני הקבצים.
- במקרה שנגדיר תווית גלובלית בקובץ אחד בעלת אותו שם של תווית שאינה גלובלית (מקומית) בקובץ אחר, בקובץ האחר ההתייחסות תהיה לתווית המקומית שבו.
- שימו לב שסיומת הקובץ חייבת להיות asm על מנת שזה יעבוד !!!