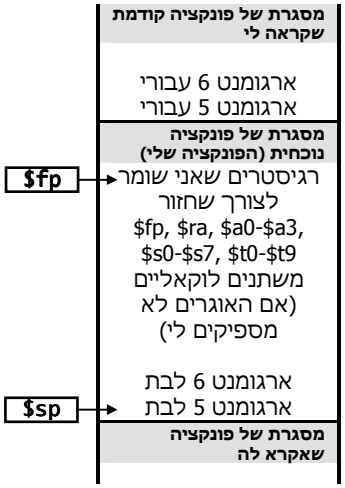
מעבד MIPS R2000

תיאור פקודות לקורס תכן לוגי

אוגרים

שם	מספר	תיאור	נשמר אחרי קריאה לפונק'
$zero	0 (00000)	תמיד 0 (לא ניתן לשינוי)	כן (קבוע)
$at	1 (00001)	שמור לאסמבלר	לא רלוונטי
$v0	2 (00010)	ערך חזרה ראשון של פונקציה	לא
$v1	3 (00011)	ערך חזרה שני של פונקציה	לא
$a0	4 (00100)	ארגומנט ראשון לפונקציה	כן
$a1	5 (00101)	ארגומנט שני לפונקציה	כן
$a2	6 (00110)	ארגומנט שלישי לפונקציה	כן
$a3	7 (00111)	ארגומנט רביעי לפונקציה	כן
$t0	8 (01000)	זמני (לא נשמר)	לא
$t1	9 (01001)	זמני (לא נשמר)	לא
$t2	10 (01010)	זמני (לא נשמר)	לא
$t3	11 (01011)	זמני (לא נשמר)	לא
$t4	12 (01100)	זמני (לא נשמר)	לא
$t5	13 (01101)	זמני (לא נשמר)	לא
$t6	14 (01110)	זמני (לא נשמר)	לא
$t7	15 (01111)	זמני (לא נשמר)	לא
$s0	16 (10000)	משתנה פנימי (נשמר)	כן
$s1	17 (10001)	משתנה פנימי (נשמר)	כן
$s2	18 (10010)	משתנה פנימי (נשמר)	כן
$s3	19 (10011)	משתנה פנימי (נשמר)	כן
$s4	20 (10100)	משתנה פנימי (נשמר)	כן
$s5	21 (10101)	משתנה פנימי (נשמר)	כן
$s6	22 (10110)	משתנה פנימי (נשמר)	כן
$s7	23 (10111)	משתנה פנימי (נשמר)	כן
$t8	24 (11000)	זמני (לא נשמר)	לא
$t9	25 (11001)	זמני (לא נשמר)	לא
$k0	26 (11010)	שמור למערכת הפעלה	לא רלוונטי
$k1	27 (11011)	שמור למערכת הפעלה	לא רלוונטי
$gp	28 (11100)	מצביע לאיזור גלובלי	לא רלוונטי
$sp	29 (11101)	מצביע למחסנית (סוף frame)	כן
$fp	30 (11110)	מצביע לתחילת מסגרת (frame)	כן
$ra	31 (11111)	כתובת חזרה מפונקציה	כן

כתובות גבוהות
תחילת המחסנית



חישוב גודל המסגרת הדרוש (סטטי)

גודל המסגרת נקבע בשורה הראשונה של הפרולוג של הפונקציה (מה שמחסרים מ-$sp).
ניתן לחשב גודל מדוייק או לקחת סדר גודל שבטוח יספיק (פחות יעיל).

צורך	גודל (בתים)	מתי
$fp	4	תמיד
$ra	4	רק אם אני מתכוון לקרוא לפונקציות
$a0-$a3 (של אבא)	4 לכל אוגר	רק אם אני מתכוון להשתמש באוגרים הללו לרוב רק אם אני מתכוון לקרוא לפונקציות שיש להן ארגומנטים
$s0-$s7 (של אבא)	4 לכל אוגר	רק אם אני מתכוון להשתמש באוגרים הללו אילו אוגרים של הפונקציה שקראה לי שאני חייב לשחזר
$t0-$t9 (שלי)	4 לכל אוגר	רק אם אני מתכוון לקרוא לפונקציות ואני שומר באוגרים הזמניים ערכים שאני לא רוצה שייהרסו (הפונקציות לא מחוייבות לשחזר אותם)
העברת ארגומנטים במחסנית	4 לכל ארגומנט	רק אם אני קורא לפונקציות שמקבלות מעל ל-4 ארגומנטים צריך לשמור מקום עבור המספר המקסימלי הדרוש (פחות ה-4 הראשונים)
מחסור באוגרים	כמה שצריך	אם אני צריך לשמור בפונקציה כמות גדולה של משתנים ואין לי מספיק אוגרים (למשל מערך ארוך כמשתנה לוקאלי)

מוסכמות כתיבת פונקציות (סטטי)

התחלה של פונקציה חדשה
צריך לבנות מסגרת חדשה במחסנית (ניתן לשנות את גודלה)
צריך לשמור את $fp הקודם (כאן במילה הראשונה)
צריך לכוון את $fp שיצביע למילה הראשונה במסגרת שלי
אם אני הולך לקרוא לפונקציות צריך לשמור את $ra הקודם (כאן במילה השניה)
אם אני הולך לכתוב על $a0 צריך לשמור את הקודם (כאן במילה השלישית)
כנ"ל לגבי $a1, $a2, $a3
אם אני הולך לכתוב על $s0 צריך לשמור את הקודם (כאן במילה הרביעית)
כנ"ל לגבי $s1, $s2, $s3, $s4, $s5, $s6, $s7

```
# prologue
addi $sp, $sp, -0x20
sw $fp, 0x1c($sp)
addi $fp, $sp, 0x1c
sw $ra, -0x04($fp)
sw $a0, -0x08($fp)

sw $s0, -0x0c($fp)
```

אם נתנו לי ארגומנטים דרך המחסנית צריך לשלוף אותם
הארגומנט החמישי הכי קרוב למסגרת שלי, מילה אחת מעליה
הארגומנט השישי נמצא שתי מילים מעל המסגרת שלי

```
# access my given arguments
lw $t0, 0x04($fp)
lw $t1, 0x08($fp)
```

קוד הפונקציה שלי..

אם אני רוצה לקרוא פתאום לפונקציה אחרת
אם חשוב לי לשמור על אוגרים זמניים (מפחד שישתנו בקריאה) נשמור אותם
כנ"ל לגבי $t1, $t2, $t3, $t4, $t5, $t6, $t7, $t8, $t9
אם צריך להעביר לפונקציה ארגומנטים לשים 4 ראשונים ב-$a0, $a1, $a2, $a3
אם יש ארגומנט חמישי נעביר אותו דרך המחסנית (מילה אחרונה במסגרת שלי)
אם יש ארגומנט שישי נעביר אותו מילה מעל (מילה לפני אחרונה במסגרת שלי)
צריך לקרוא לפונקציה
אם שמרתי אוגרים זמניים (מפחד שישתנו בקריאה) נשחזר אותם
כנ"ל לגבי $t1, $t2, $t3, $t4, $t5, $t6, $t7, $t8, $t9
אם יש ערך חזרה מהפונקציה אפשר להשתמש בו (כנ"ל לגבי $v1)

```
# call another function
sw $t0, -0x10($fp)

addi $a0, $zero, 0x1234
sw $t8, 0x00($sp)
sw $t9, 0x04($sp)
jal sub_function
lw $t0, -0x10($fp)

beq $v0, $zero, label
```

קוד הפונקציה שלי..

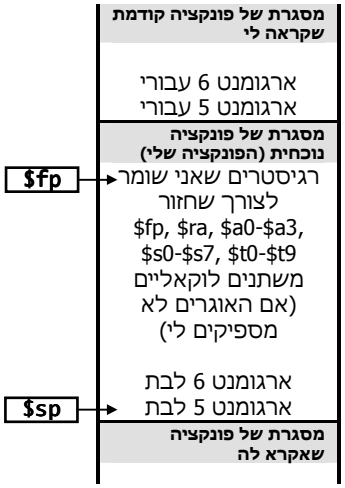
סוף הפונקציה
אם אני רוצה להחזיר ערך צריך לשים אותו ב-$v0 (ניתן גם ערך נוסף ב-$v1)
אם שמרתי את $s0 בפרולוג צריך לשחזר אותו (כאן במילה הרביעית)
כנ"ל לגבי $s1, $s2, $s3, $s4, $s5, $s6, $s7
אם שמרתי את $a0 בפרולוג צריך לשחזר אותו (כאן במילה השלישית)
כנ"ל לגבי $a1, $a2, $a3
אם שמרתי את $ra בפרולוג צריך לשחזר אותו (כאן במילה השניה)
צריך לשחזר את $fp ששמרנו בפרולוג (כאן במילה הראשונה)
צריך לנקות את המסגרת שלי מהמחסנית
צריך לקפוץ חזרה למי שהריץ אותי

```
# epilogue
addi $v0, $zero, 0x1234
lw $s0, -0x0c($fp)

lw $a0, -0x08($fp)

lw $ra, -0x04($fp)
lw $fp, 0x1c($sp)
addi $sp, $sp, 0x20
jr $ra
```

כתובות גבוהות
תחילת המחסנית



חישוב גודל מסגרת התחלתי (דינאמי)

בשורה הראשונה של הפרולוג של הפונקציה נקבע רק גודל מסגרת התחלתי (מינימלי).
אם במהלך הפונקציה צריך מסגרת גדולה יותר (כמו לפני קריאה לפונקציה) אז נגדיל.

צורך	גודל (בתים)	מתי
$fp	4	תמיד
$ra	4	רק אם אני מתכוון לקרוא לפונקציות
$a0-$a3 (של אבא)	4 לכל אוגר	רק אם אני מתכוון להשתמש באוגרים הללו לרוב רק אם אני מתכוון לקרוא לפונקציות שיש להן ארגומנטים
$s0-$s7 (של אבא)	4 לכל אוגר	רק אם אני מתכוון להשתמש באוגרים הללו אילו אוגרים של הפונקציה שקראה לי שאני חייב לשחזר
$t0-$t9 (שלי)	4 לכל אוגר	רק אם אני מתכוון לקרוא לפונקציות ואני שומר באוגרים הזמניים ערכים שאני לא רוצה שייהרסו (הפונקציות לא מחוייבות לשחזר אותם) ניתן להקצות להם מקום במסגרת גם באופן דינאמי
מחסור באוגרים	כמה שצריך	אם אני צריך לשמור בפונקציה כמות גדולה של משתנים ואין לי מספיק אוגרים (למשל מערך ארוך כמשתנה לוקאלי)

מוסכמות כתיבת פונקציות (דינאמי)

התחלה של פונקציה חדשה
צריך לבנות מסגרת חדשה במחסנית (ניתן לשנות את גודלה)
צריך לשמור את $fp הקודם (כאן במילה הראשונה)
צריך לכוון את $fp שיצביע למילה הראשונה במסגרת שלי
אם אני הולך לקרוא לפונקציות צריך לשמור את $ra הקודם (כאן במילה השניה)
אם אני הולך לכתוב על $a0 צריך לשמור את הקודם (כאן במילה השלישית)
כנ"ל לגבי $a1, $a2, $a3
אם אני הולך לכתוב על $s0 צריך לשמור את הקודם (כאן במילה הרביעית)
כנ"ל לגבי $s1, $s2, $s3, $s4, $s5, $s6, $s7

```
# prologue
addi $sp, $sp, -0x20
sw $fp, 0x1c($sp)
addi $fp, $sp, 0x1c
sw $ra, -0x04($fp)
sw $a0, -0x08($fp)

sw $s0, -0x0c($fp)
```

אם נתנו לי ארגומנטים דרך המחסנית צריך לשלוף אותם
הארגומנט החמישי הכי קרוב למסגרת שלי, מילה אחת מעליה
הארגומנט השישי נמצא שתי מילים מעל המסגרת שלי

```
# access my given arguments
lw $t0, 0x04($fp)
lw $t1, 0x08($fp)
```

קוד הפונקציה שלי..

אם אני רוצה לקרוא פתאום לפונקציה אחרת
אם חשוב לי לשמור על אוגרים זמניים (מפחד שישתנו בקריאה) נשמור אותם
כנ"ל לגבי $t1, $t2, $t3, $t4, $t5, $t6, $t7, $t8, $t9
אם צריך להעביר לפונקציה ארגומנטים לשים 4 ראשונים ב-$a0, $a1, $a2, $a3
בשביל ארגומנט 5 נרחיב את גודל המסגרת באופן דינאמי (נוסיף 4 בתים לסופה)
נעביר ארגומנט חמישי דרך המחסנית (מילה אחרונה במסגרת שלי)
צריך לקרוא לפונקציה
ננקה את התוספת הדינאמית למסגרת
אם שמרתי אוגרים זמניים (מפחד שישתנו בקריאה) נשחזר אותם
כנ"ל לגבי $t1, $t2, $t3, $t4, $t5, $t6, $t7, $t8, $t9
אם יש ערך חזרה מהפונקציה אפשר להשתמש בו (כנ"ל לגבי $v1)

```
# call another function
sw $t0, -0x10($fp)

addi $a0, $zero, 0x1234
addi $sp, $sp, -0x04
sw $t8, 0x00($sp)
jal sub_function
addi $sp, $sp, 0x04
lw $t0, -0x10($fp)

beq $v0, $zero, label
```

קוד הפונקציה שלי..

סוף הפונקציה
אם אני רוצה להחזיר ערך צריך לשים אותו ב-$v0 (ניתן גם ערך נוסף ב-$v1)
אם שמרתי את $s0 בפרולוג צריך לשחזר אותו (כאן במילה הרביעית)
כנ"ל לגבי $s1, $s2, $s3, $s4, $s5, $s6, $s7
אם שמרתי את $a0 בפרולוג צריך לשחזר אותו (כאן במילה השלישית)
כנ"ל לגבי $a1, $a2, $a3
אם שמרתי את $ra בפרולוג צריך לשחזר אותו (כאן במילה השניה)
צריך לשחזר את $fp ששמרנו בפרולוג (כאן במילה הראשונה)
צריך לנקות את המסגרת שלי מהמחסנית
צריך לקפוץ חזרה למי שהריץ אותי

```
# epilogue
addi $v0, $zero, 0x1234
lw $s0, -0x0c($fp)

lw $a0, -0x08($fp)

lw $ra, -0x04($fp)
lw $fp, 0x1c($sp)
addi $sp, $sp, 0x20
jr $ra
```

פקודות R-Type

opcode (6) [31:26]	rs (5) [25:21]	rt (5) [20:16]	rd (5) [15:11]	shamt (5) [10:6]	function (6) [5:0]

פקודות לביצוע פעולה בין שני רגיסטרים ושמירת התוצאה ברגיסטר שלישי (rd). ה-opcode תמיד 000000 וסוג הפעולה נקבע בשדה function. אם נסמן את הפעולה בתור * נקבל rt * rs = rd. גם פקודות jr (קפיצה לרגיסטר) ממומשות כאן.

Jump register (jr)

opcode = 000000, function = 001000

```
jr rs
$pc = rs[31:0]
```

קפיצה לכתובת השמורה ברגיסטר rs. תוכן רגיסטר זה מועתק לתוך PC ישירות, כך שהפקודה הבאה שתרוץ היא הפקודה שבכתובת זו. לפי הגדרה שדה rt הוא 00000 כלומר רגיסטר $zero (שימושי לצורך מימוש).

Jump and link register (jalr)

opcode = 000000, function = 001001

```
jalr rs, rd
rd[31:0] = ($pc+4)[31:0] ; $pc = rs[31:0]
```

קפיצה לכתובת השמורה ברגיסטר rs. תוכן רגיסטר זה מועתק לתוך PC ישירות, כך שהפקודה הבאה שתרוץ היא הפקודה שבכתובת זו. לפני הקפיצה, כתובת החזרה (הכתובת הבאה אחרי פקודה זו, כלומר PC+4 המקורי) נשמרת ברגיסטר rd. לרוב רגיסטר זה יהיה RA (רגיסטר 31).

Set less than (slt)

opcode = 000000, function = 101010

```
slt rd, rs, rt
if (rs[31:0] < rt[31:0]) rd[31:0] = 0x0001 ; else rd[31:0] = 0x0000
```

השוואה בין תוכן הרגיסטר rs לתוכן הרגיסטר rt. אם rs קטן ממש מ-rt אז ייכתב הערך 1 לרגיסטר rd, אחרת ייכתב הערך 0 (כל ה-32 ביט של רגיסטר rd ישתנו).

Add (add)

opcode = 000000, function = 100000

```
add rd, rs, rt
rd[31:0] = rs[31:0] + rt[31:0]
```

חיבור התוכן של הרגיסטר rs עם התוכן של הרגיסטר rt ושמירת התוצאה ברגיסטר rd.

Sub (sub)

opcode = 000000, function = 100010

```
sub rd, rs, rt
rd[31:0] = rs[31:0] - rt[31:0]
```

חיסור התוכן של הרגיסטר rt מהתוכן של הרגיסטר rs ושמירת התוצאה ברגיסטר rd.

Or (or)

opcode = 000000, function = 100101

```
and rd, rs, rt
rd[31:0] = rs[31:0] | rt[31:0]
```

ביצוע or ביט-ביט בין תוכן הרגיסטר rs לבין התוכן של rt ושמירת התוצאה ב-rd.

And (and)

opcode = 000000, function = 100100

```
and rd, rs, rt
rd[31:0] = rs[31:0] & rt[31:0]
```

ביצוע and ביט-ביט בין תוכן הרגיסטר rs לבין התוכן של rt ושמירת התוצאה ב-rd.

Xor (xor)

opcode = 000000, function = 100110

```
xor rd, rs, rt
rd[31:0] = rs[31:0] ^ rt[31:0]
```

ביצוע xor ביט-ביט בין תוכן הרגיסטר rs לבין התוכן של rt ושמירת התוצאה ב-rd.

Shift left logical (sll)

opcode = 000000, function = 000000

```
sll rd, rt, shamt
rd[31:shamt] = rt[31-shamt:0] ; rd[shamt-1:0] = 0
```

הזזת כל הביטים של רגיסטר rt שמאלה shamt ביטים. הביטים העליונים נעלמים. בביטים התחתונים נכנסים אפסים. שקול לפעולה Shift left arithmetic ולכן אינה קיימת. שקול להכפלת תוכן רגיסטר rt במספר 2^shamt.

Shift right logical (srl)

opcode = 000000, function = 000010

```
srl rd, rt, shamt
rd[31-shamt:0] = rt[31:shamt] ; rd[31:31-shamt+1] = 0
```

הזזת כל הביטים של רגיסטר rt ימינה shamt ביטים. הביטים התחתונים נעלמים. בביטים העליונים נכנסים אפסים.

Shift right arithmetic (sra)

opcode = 000000, function = 000011

```
sra rd, rt, shamt
rd[31-shamt:0] = rt[31:shamt] ; rd[31:31-shamt+1] = extend(rt[31:31])
```

הזזת כל הביטים של רגיסטר rt ימינה shamt ביטים. הביטים התחתונים נעלמים. בביטים העליונים נכנס ביט הסימן של רגיסטר rt (הביט העליון ביותר, התוכן הרי מיוצג ב-2's complement). שקול לחלוקת התוכן של רגיסטר rt במספר 2^shamt (תוך עיגול למטה).

Rotate left (rol)

pseudo-instruction

```
rol rd, rs, rt
shamt = rt[31:0] ; rd[31:shamt] = rs[31-shamt:0] ; rd[shamt-1:0] = rs[31:31-shamt+1]
```

רוטציה של כל הביטים שבתוך רגיסטר rs שמאלה כמספר הביטים שברגיסטר rt.

Rotate right (ror)

pseudo-instruction

```
ror rd, rs, rt
shamt = rt[31:0] ; rd[31-shamt:0] = rs[31:shamt] ; rd[31:31-shamt+1] = rs[shamt-1:0]
```

רוטציה של כל הביטים שבתוך רגיסטר rs שמאלה כמספר הביטים שברגיסטר rt.

פקודות I-Type

opcode (6) [31:26]	rs (5) [25:21]	rt (5) [20:16]	immediate value (16) [15:0]

פקודות לביצוע פעולה על רגיסטר rt עם קבוע מספרי (immediate value) ושמירת התוצאה ברגיסטר rs. הקבוע המספרי immediate value ברוחב 16 ביט ושמור ב-2's complement. גם פקודות branch ו-load/store ממומשות כאן.

Branch on equal (beq)

opcode = 000100

```
beq rs, rt, label
if (rs[31:0] == rt[31:0]) $pc[31:0] = ($pc+4)[31:0] + 4*sign-extend(value[15:0])
```

אם התוכן של רגיסטר rs שווה לתוכן רגיסטר rt, תתבצע קפיצה יחסית באופן הבא: ראשית PC מקודם ב-4. המספר הקבוע immediate value עובר sign extension לרוחב 32 ביט מ-16 ביט, ואז מוכפל ב-4. התוצאה מתווספת לערכו החדש של PC (כלומר ל-PC+4). אם המספר הקבוע שלילי, תתבצע קפיצה אחורה. אם תוכן הרגיסטרים אינו שווה, לא תתבצע שום קפיצה ו-PC יקודם ב-4 כרגיל כדי להגיע לפקודה הבאה.

Branch on not equal (bne)

opcode = 000101

```
bne rs, rt, label
if (rs[31:0] != rt[31:0]) $pc[31:0] = ($pc+4)[31:0] + 4*sign-extend(value[15:0])
```

זהה לפקודה beq, רק שהפעם מתבצעת הקפיצה אם תוכן שני הרגיסטרים rs ו-rt שונה.

Branch on greater than equal (bge)

pseudo-instruction

```
bge rs, rt, label
if (rs[31:0] >= rt[31:0]) $pc = label
```

זהה לפקודה beq, רק שהפעם מתבצעת הקפיצה אם תוכן הרגיסטר rs גדול או שווה לתוכן rt.

Branch on greater than (bgt)

pseudo-instruction

```
bgt rs, rt, label
if (rs[31:0] > rt[31:0]) $pc = label
```

זהה לפקודה beq, רק שהפעם מתבצעת הקפיצה אם תוכן הרגיסטר rs גדול ממש מתוכן rt.

Branch on less than equal (ble)

pseudo-instruction

```
bge rs, rt, label
if (rs[31:0] <= rt[31:0]) $pc = label
```

זהה לפקודה beq, רק שהפעם מתבצעת הקפיצה אם תוכן הרגיסטר rs קטן או שווה לתוכן rt.

Branch on less than (blt)

pseudo-instruction

```
bgt rs, rt, label
if (rs[31:0] < rt[31:0]) $pc = label
```

זהה לפקודה beq, רק שהפעם מתבצעת הקפיצה אם תוכן הרגיסטר rs קטן ממש מתוכן rt.

Set less than immediate (slti)

opcode = 001010

```
slti rt, rs, value
if (rs[31:0] < sign-extend(value[15:0])) rt[31:0] = 0x0001 ; else rt[31:0] = 0x0000
```

השוואה בין תוכן הרגיסטר rs למספר הקבוע immediate value. אם rs קטן ממש מ-value (אחרי ש-value עובר sign extension ל-32 ביט) אז ייכתב הערך 1 לרגיסטר rt, אחרת ייכתב הערך 0 (כל ה-32 ביט של רגיסטר rt ישתנו).

Load upper immediate (lui)

opcode = 001111

```
lui rt, value
rt[31:16] = value[15:0] ; rt[15:0] = 0x0000
```

כל 16 הביטים של המספר הקבוע immediate value מועתקים ל-16 הביטים העליונים של רגיסטר rt. שאר הביטים של rt (16 הביטים התחתונים) מאופסים. אין שימוש ברגיסטר rs.

Load word (lw)

opcode = 100011

```
lw rt, value(rs)
rt[31:0] = MEMORY[ rs[31:0] + sign-extend(value[15:0]) ]
```

קריאה של 32 ביט מתוך הזיכרון בכתובת שבתוך הרגיסטר rs עם היסט קבוע value. כלומר, הכתובת שממנה תתבצע הקריאה מחושבת ע"י הוספה של immediate value אחרי sign extension לתוכן של הרגיסטר rs. אם value שלילי, הקריאה תתבצע מכתובת שלפני rs. לשים לב שאין הכפלה ב-4 כמו שיש ב-branch.

Load halfword (lh)

opcode = 100001

```
lh rt, value(rs)
rt[31:0] = sign-extend( MEMORY[ rs[31:0] + sign-extend(value[15:0]) ] )
```

קריאה של 16 ביט (שני בתים) מתוך הזיכרון בכתובת שבתוך הרגיסטר rs עם היסט קבוע value. שני הבתים עוברים

sign extension כדי שיימלאו את כל ה-32 ביט של רגיסטר התוצאה rt. הכתובת שממנה תתבצע הקריאה מחושבת ע"י הוספה של immediate value אחרי sign extension לתוכן של הרגיסטר rs. אם value שלילי, הקריאה תתבצע מכתובת שלפני rs. לשים לב שאין הכפלה ב-4 כמו שיש ב-branch.

Load byte (lb)

opcode = 100000

```
lb rt, value(rs)
rt[31:0] = sign-extend( MEMORY[ rs[31:0] + sign-extend(value[15:0]) ] )
```

קריאה של 8 ביט (בית בודד) מתוך הזיכרון בכתובת שבתוך הרגיסטר rs עם היסט קבוע value. הבית הבודד עובר sign extension כדי שיימלא את כל ה-32 ביט של רגיסטר התוצאה rt. הכתובת שממנה תתבצע הקריאה מחושבת ע"י הוספה של immediate value אחרי sign extension לתוכן של הרגיסטר rs. אם value שלילי, הקריאה תתבצע מכתובת שלפני rs. לשים לב שאין הכפלה ב-4 כמו שיש ב-branch.

Store word (sw)

opcode = 101011

```
sw rt, value(rs)
MEMORY[ rs[31:0] + sign-extend(value[15:0]) ] = rt[31:0]
```

כתיבה של 32 ביט מהתוכן של רגיסטר rt לזיכרון בכתובת שבתוך הרגיסטר rs עם היסט קבוע value. כלומר, הכתובת שאליה תתבצע הכתיבה מחושבת ע"י הוספה של immediate value אחרי sign extension לתוכן של הרגיסטר rs. אם value שלילי, הכתיבה תתבצע לכתובת שלפני rs. לשים לב שאין הכפלה ב-4 כמו שיש ב-branch.

Store halfword (sh)

opcode = 101001

```
sh rt, value(rs)
MEMORY[ rs[31:0] + sign-extend(value[15:0]) ] = rt[15:0]
```

כתיבה של ה-16 ביט התחתונים מהתוכן של רגיסטר rt לזיכרון בכתובת שבתוך הרגיסטר rs עם היסט קבוע value. כלומר, הכתובת שאליה תתבצע הכתיבה מחושבת ע"י הוספה של immediate value אחרי sign extension לתוכן של הרגיסטר rs. אם value שלילי, הכתיבה תתבצע לכתובת שלפני rs. לשים לב שאין הכפלה ב-4 כמו שיש ב-branch.

Store byte (sb)

opcode = 101000

```
sb rt, value(rs)
MEMORY[ rs[31:0] + sign-extend(value[15:0]) ] = rt[7:0]
```

כתיבה של ה-8 ביט התחתונים מהתוכן של רגיסטר rt לזיכרון בכתובת שבתוך הרגיסטר rs עם היסט קבוע value. כלומר, הכתובת שאליה תתבצע הכתיבה מחושבת ע"י הוספה של immediate value אחרי sign extension לתוכן של הרגיסטר rs. אם value שלילי, הכתיבה תתבצע לכתובת שלפני rs. לשים לב שאין הכפלה ב-4 כמו שיש ב-branch.

Load address (la) , Load immediate (li)

pseudo-instruction

```
la rt, label                    li rt, value
rt[31:0] = label                rt[31:0] = value[31:0]
```

ב-la שמירת הכתובת label (לא התוכן שבכתובת label) ברגיסטר rt. משמש למשל אם ישנו מערך נתונים ב-data. של התוכנית, ורוצים לקבל מצביע לתחילתו. ב-li שמירת הערך value (32 ביט) ברגיסטר rt. שניהם ממומשים ע"י ביצוע lui (לחלק העליון של הערך המספרי) ואז ori (לחלק התחתון של הערך המספרי).

Add immediate (addi)

opcode = 001000

```
addi rt, rs, value
rt[31:0] = rs[31:0] + sign-extend(value[15:0])
```

התוכן של rs מחובר עם המספר הקבוע immediate value אחרי שהמספר הקבוע עובר sign extension.

Or immediate (ori)

opcode = 001101

```
ori rt, rs, value
rt[15:0] = rs[15:0] | value[15:0] ; rt[31:16] = rs[31:16]
```

הקבוע immediate value מושלם עם אפסים ל-32 ביט, כלומר כל ה-16 הביט העליונים הם 0. מבצעים or ביט-ביט בין תוצאה זו לבין התוכן של rs ושומרים את התוצאה ב-rt.

And immediate (andi)

opcode = 001100

```
andi rt, rs, value
rt[15:0] = rs[15:0] & value[15:0] ; rt[31:16] = 0x0000
```

הקבוע immediate value מושלם עם אפסים ל-32 ביט, כלומר כל ה-16 הביט העליונים הם 0. מבצעים and ביט-ביט בין תוצאה זו לבין התוכן של rs ושומרים את התוצאה ב-rt.

Xor immediate (xori)

opcode = 001101

```
xori rt, rs, value
rt[15:0] = rs[15:0] ^ value[15:0] ; rt[31:16] = rs[31:16]
```

הקבוע immediate value מושלם עם אפסים ל-32 ביט, כלומר כל ה-16 הביט העליונים הם 0. מבצעים xor ביט-ביט בין תוצאה זו לבין התוכן של rs ושומרים את התוצאה ב-rt.

פקודות J-Type

opcode (6) [31:26]	target address (26) [25:0]

פקודות לביצוע קפיצה לכתובת אבסולוטית. מירב הביטים בקידוד הפקודה משמשים לצורך קביעת כתובת היעד של הקפיצה.

Jump (j) opcode = 000010

```
j label
$pc[31:28] = ($pc+4)[31:28] ; $pc[27:2] = (target address)[25:0] ; $pc[1:0] = 00
```

קפיצה אבסולוטית (לא יחסית) לכתובת המקודדת באופן הבא: 2 ביטים תחתונים 00 (כתובת להרצה תמיד מתחלקת ב-4), 26 ביטים הבאים מהפרמטר target address ו-4 ביטים העליונים הם 4 הביטים העליונים של PC+4.

Jump and link (jal) opcode = 000011

```
jal label
$ra = ($pc+4)[31:0] ;
$pc[31:28] = ($pc+4)[31:28] ; $pc[27:2] = (target address)[25:0] ; $pc[1:0] = 00
```

זהה בדיוק לפקודת Jump, רק שלפני ש-PC משתנה לערכו החדש, כתובת הפקודה הבאה (PC+4) נשמרת ברגיסטר RA (רגיסטר 31). משמשת בעת קריאה לפונקציה כדי שנזכור ברגיסטר RA (return address) לאיפה לחזור בסוף הפונקציה.