# שפת MIPS

## הקדמה

MIPS היא שפת אסמבלי שהומצאה בשנות ה-80. שפה זו כוללת מספר קטן של פקודות ביחס לשפות אסמבלי אחרות, המטרה היא לשמור על פשטות למתכנת ומהירות של התרגום לשפת מכונה. עקרונות MIPS הם:

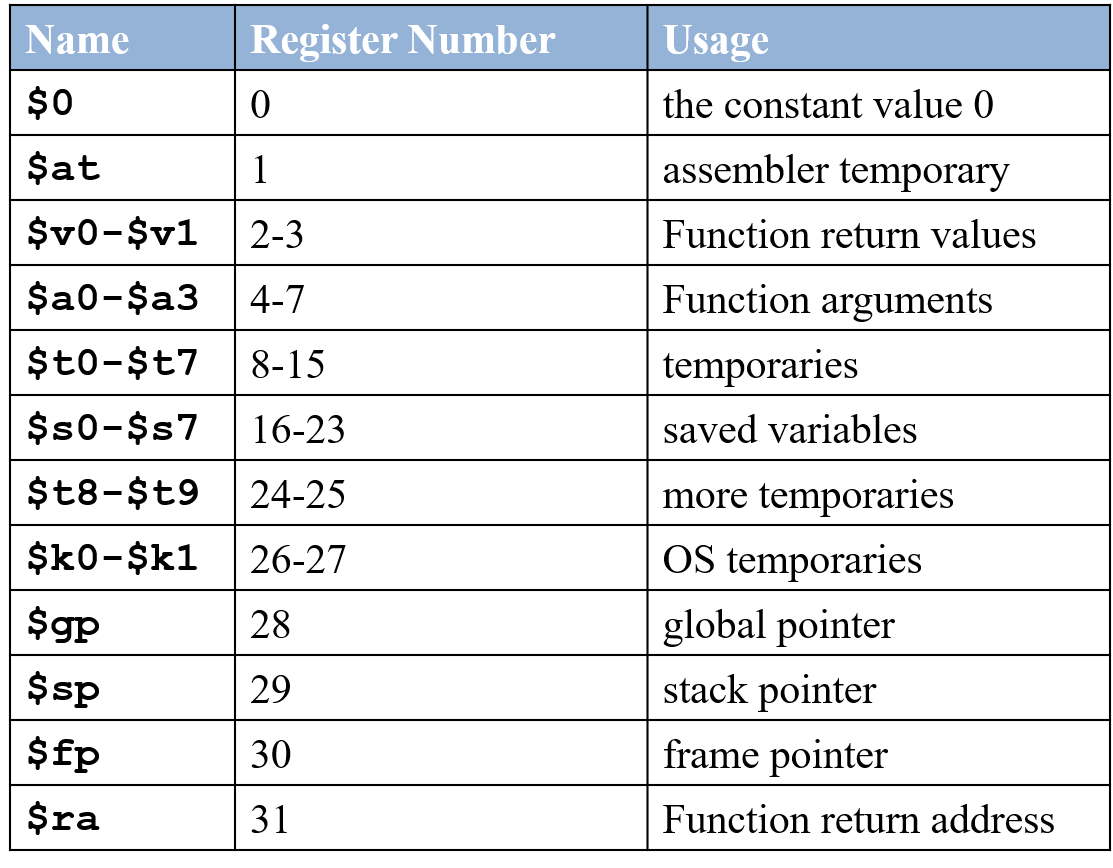
* כל פקודה כוללת שתי מקורות ויעד אחד.
* פעולה מסובכת צריך לפרק למספר פקודות פשוטות.
* הארכיטקטורה היא של 32-bit, כלומר כל אוגר ותא זיכרון הוא בגדול 32-bit, וכן כל שורת פקודה היא באורך 32.

## זיכרון - מושגים בסיסיים

בהמשך נלמד יותר על הזיכרון כיצד הוא מחולק ומהם רכיביו, אמנם כעת נפרט רק מושגים בסיסיים.

**אוגרים:**

בMIPS יש 32 אוגרים מיוחדים שכל אחד הוא באורך 32-bit. השימוש המיועד לכל אוגר, וכן השם והמספר שלו נמצאים בטבלה למטה. הגישה לכל אוגר נעשית באמצעות הסימון $. נלמד על כולם בהמשך.

****

הגישה לאוגרים היא מאוד מהירה בגלל מספרם המצומצם, אמנם החיסרון הוא שבתוכניות ארוכות יש לנו מעט מדי אוגרים שנוכל להשתמש בהם. הפתרון הוא להשתמש בתאי הזיכרון, אמנם הגישה אליהם היא איטית יותר.

**תאי זיכרון:**

בMIPS כתובות הזיכרון מחולקות לבתים bytes, כל בית הוא שמונה ביטים, ולכל בית יש כמובן כתובת ייחודית המתחיל מ-0 עד מספר הבתים בזיכרון. מילה בזיכרון המכילה 32 ביטים היא בעצם ארבעה כתובות בזיכרון. חשוב תמיד לזכור שכתובות בMIPS הם לפי בתים ולא מילים, ולכן הגישה לתא זיכרון הבא תהיה בכתובת של תא הזיכרון הקודם ולהוסיף 4.

**כתובות פקודות:**

לכל פקודה בתוכנה שלנו יש כתובת המאוחסנת בתאי זיכרון הנמצאים ב-ROM. כל כתובת כזו גם היא באורך 32 ביט ומאוחסנת בארבעה תאי זיכרון באורך 4 bytes, כך שהכתובות של הפקודות הם בקפיצות של 4. באמצעות מונה התוכנה (PC) המעבד יודע מה הפקודה הבאה לבצע, כאשר ברירת המחדל היא לקפוץ לכתובת הבאה אלא אם כן הורנו לו אחרת באמצעות פונקציות jump. חשוב לשים לב שהכתובות הם כפולות של 4.

## קריאה וכתיבה מהזיכרון

**קריאה מהזיכרון:**

קוראים מהזיכרון מילה באמצעות פקודת lw (load word) בפורמט הבא: **lw $s0, 4($t1)**. כאשר $so הוא כתובת הזיכרון שבה נשים את הערך שנקרא. ו- 4($t1) הוא גם כתובת בזיכרון שממנו נקרא, כאשר הכתובת מחושבת כך: $t1 + 4. חשוב לזכור שכתובות של מילה הם בקפיצות של 4 ולכן הערך הקבוע (מחוץ לסוגריים) חייב להיות כפולה של 4. ניתן גם לרשום את המספר הקבוע בערך הקסדצימלי, אך יש לציין לפני כן זאת באמצעות סימון 0x. הקבוע המקסימלי שניתן לרשום בפקודה זו הוא בעל 16-bit בלבד בשיטת משלים ל-2.

ניתן לקרוא חצי מילה באמצעות פקודת lh (load half) שגם היא בפורמט זהה lh $0 2($t1). בפקודה זו הכתובת צריכה להיות כפולה של 2. וכן ניתן גם לקרוא בית בודד ולא מילה באמצעות פקודת lb (load byte) שגם היא בפורמט דומה: lb $0, 3($t1). בפקודה זו הכתובת לא צריכה להיות כפולה של 4.

**כתיבה בזיכרון:**

כותבים בזיכרון מילה באמצעות פקודת sw (store word) בפורמט הבא: **sw $s0, 4($t1)**. כאשר $so הוא כתובת הזיכרון שממנה נקרא ערך כלשהו. ו- 4($t1) הוא הכתובת בזיכרון שבה נכתוב את הערך שקראנו, כאשר הכתובת מחושבת כך: $t1 + 4. בכל שאר הדברים פקודה זו זהה לקריאה מהזיכרון.

ניתן גם לכתוב בבית בודד ולא במילה באמצעות פקודת sb (store byte) שגם היא בפורמט דומה: lb $0, 4($t1). בפקודה זו הכתובת לא צריכה להיות כפולה של 4.

**Big-Endian & Little-Endian:**

כבר אמרנו שמילה בMIPS היא באורך 32 ביט ומכילה 4 כתובות בזיכרון, כאשר כל כתובת מייצגת byte אחד שהוא 8 ביטים. מכאן עולה השאלה הבאה: כאשר אנו מאחסנים מילה באורך 32 ביט בזיכרון איזה מידע נשמר בכתובת 0 (byte 0)? האם 8 הביטים הכי גדולים במילה (MSB) או 8 הביטים הכי קטנים (LSB)? התשובה היא שיש שתי סוגי מערכות בMIPS:

Little-Endian - היא מערכת שבה בכל מילה בזיכרון בית 0 הוא ה-LSB ובית 4 הוא ה-MSB.

Big-Endian - בדיוק ההיפך, כל מילה בזיכרון בית 0 הוא ה-MSB ובית 4 הוא ה-LSB.

## יצירת קבועים

ניתן ליצור אוגר שמחזיק ערך קבוע, לדוגמא 5, באמצעות הפקודה addi $s0, $0, 5. אמנם כך הקבועים מוגבלים ל-16 ביטים. מה נעשה כדי לטעון ערכים גדולים יותר? נשתמש בפקודת lui המכניסה ערך קבוע ל-16 הביטים העליונים לאוגר ואז להפעיל פקודת ori כדי לאחד את האוגר עם ה-16 ביטים התחתונים.

lui $so, 0xFEDC

ori $s0, $s0, 0x8765 # $s0 = 0xFEDC8765

## פעולות אריתמטיות

**הוספה:**

פקודת הוספה מתבצעת בפורמט הזה: add a, b, c. כאשר a הוא כתובת היעד בה תישמר תוצאת החיבור, ו-b ו-c הם האוגרים או תאי הזיכרון אותם אנו רוצים לחבר, כלומר מתבצע a = b + c.

כאשר רוצים לחבר מספר מידי עם אוגר או תא זיכרון, נשתמש בפקודה addi a, b, 3, כאשר a היא כתובת היעד, b הוא הערך שאנו מוסיפים, ו-3 הוא הערך השני שמוסיפים שהוא קבוע. כלומר מתבצע a = b + 3. נשים לב כי בחיסור אין צורך בפעולה דומה subi מפני שב-addi ניתן להוסיף גם ערכים שליליים -3, כך שיתקבל בעצם חיסור.

**חיסור:**

פקודה מאוד דומה לחיבור sub a, b, c. כאשר a הוא כתובת היעד ומחסירים את c מ-b, כלומר a = b – c.

חיסור מספר מידי מאוגר מתבצע באמצעות addi כמו שציינו.

**כפל:**

פקודת כפל בין שני אוגרים מתבצעת בפורמט הבא: **mul a, b, c**. כאשר a הוא כתובת היעד בה תישמר תוצאת הכפל, ו-b ו-c הם האוגרים או תאי הזיכרון אותם אנו רוצים לכפול, כלומר מתבצע . פעולה הכפל אפשרית אך ורק בין אוגרים בלבד ולא אוגר ומספר קבוע.

הבעיה בפקודת הכפל היא שאוגר a הוא בגודל 32 ביטים בלבד, ולכן פקודה זו היא מאוד מוגבלת, שהרי הכפלה של 32 ביטים ב-32 ביטים מאפשר תוצאה עד 64 ביטים. ולכן יש פקודה נוספת **mult b, c** המכפילה בין b ל-c ושומרת את התוצאה בשני אוגרים מיוחדים lo ו-hi שהם אוגרים ידועים השמורים בזיכרון. ב-lo ישמרו 32 הביטים התחתונים בתוצאת ההכפלה, וב-hi ישמרו 32 הביטים העליונים בתוצאת ההכפלה. מיד לאחר שהשתמשנו בפקודת זו, "נמשוך" את המידע השמור ב-lo ו-hi אל שני אוגרים אחרים באמצעות הפקודה mflo $s0 שתעביר את המידע מ-lo אל $s0, ופקודת mfhi $s1 שתעביר את המידע מ-hi אל $s1.

**חילוק:**

פקודת החילוק פועלת באופן דומה ל-mult, בפורמט הבא **div b, c**. גם כאן התוצאה תישמר בשני האוגרים מיוחדים lo ו-hi. ב-lo ישמרו 32 הביטים של המספר השלם בתוצאת ההכפלה, וב-hi ישמרו 32 הביטים של השארית בתוצאת ההכפלה. מיד לאחר שהשתמשנו בפקודת זו, "נמשוך" את המידע השמור ב-lo ו-hi באמצעות mflo ו-mfhi. החילוק אפשרי בין שני אוגרים בלבד ולא אוגר ומספר קבוע.

## פעולות לוגיות

**על אוגרים בלבד:**

פעולות לוגיות כמו: and, or, xor, nor בין שני אוגרים שכל אחד מהם מכיל 32 ביטים נעשית בפורמט הבא: func a, b, c, כאשר func היא הפעולה הלוגית, a הוא היעד שאליו תיכנס התוצאה, b ו-c הם האוגרים שעליהם נפעיל את הפעולה הלוגית. השימושים הנפוצים של פעולות לוגיות הם:

* נשתמש בפעולת and כאשר נרצה לאפס ממספר b ביטים מסוימים. נבנה מספר c שבו נשים בביט במקום כלשהו 0 אם את הביט באותו המקום ב-b נרצה למחוק, אחרת נשים בהם 1. כעת נעשה פעולת and בין b ל-c ונקבל את התוצאה המבוקשת ב-a. לדוגמא: xF234012F and 0x000000FF = 0x0000002F0.
* נשתמש בפעולת or כאשר נרצה לשנות ביטים מסוימים ב-b ל-1. נבנה מספר c שבו במקומות אלו יהיה 1 ובכל שאר הביטים יהיה 0. לדוגמא: xF2340000 0r 0x000012BC = 0xF23412BC0.
* נשתמש בפעולת nor כדי להפוך את מספר b לשלילה שלו not b. נפעיל פעולת nor בין b למספר שכולו אפסים ונקבל את התוצאה המבוקשת ב-a. כלומר: not b = b nor $0.

**עם מספרים מיידים:**

פקודות דומות לפעולות על אוגרים בלבד. הפקודה היא בצורה דומה אלא ש-c בפקודות אלו הוא מספר מידי עם 16 ביטים, ופקודות אלו יקראו andi, ori, xori. אין צורך בפקודת nori.



## פעולות הזזה shift

**הזזות לוגיות:**

פקודות sll או srl שמזיזות את כל המספר הבינארי ימינה או שמאלה מספר פעמים. בהזזה שמאלה מתווסף ביט מצד ימין ובהזזה ימינה מתווסף ביט מצד שמאל, ביטים אלו הם תמיד 0. Sll מזיז את המספר שמאלה ו-srl מזיז את המספר ימינה. פקודות אלו הם בפורמט sll a, b, 3 כאשר a הוא אוגר היעד שיאחסן את התוצאה, b הוא האוגר שבו המספר הבינארי שמזיזים, ו-3 או כל מספר אחר הוא מספר ההזזות. בהזזה שמאלה של n פעמים מתקיים: . ובהזזה ימינה n פעמים מתקיים: .

**הזזות אריתמטיות:**

פקודה אחת שנקראת sra הדומה להזזה לוגית ימינה, אלא שבה הספרה 1 הגדולה ביותר נכתבת שוב ושוב.

לדוגמא: אם $s0 = 10110011, אז לאחר הפקודהsra $s0, $s0, 2 , יתקיים: $s0=11101100.

**הערה** - לכל שלושת פקודות ההזזה שלמדנו יש פקודות דומות שבמקום להזיז מספר קבוע של פעמים שאנו מכניסים בפקודה, הם מזיזות בכמות לא ידועה, כלומר בכמות של משתנה. פקודות אלו הם: sllv, srlv, srav (כאשר v הוא variable). לדוגמא: srlv $t0, $t1, $t2, אומר להזיז ימינה את המספר שב-$t1 $t2 פעמים.

## תרגום לשפת מכונה

כל שורת פקודה היא 32 ביט סך הכל. יש שלושה סוגי פקודות:



**R-Type**:

פקודות על אוגרים מהצורה הבאה:

op - מציין את סוג הפקודה, ב-R-Type כל ששת הביטים הם 0.

rs, rt, rd - הם האוגרים עליהם מבצעים את הפקודות. בתרגום לשפת מכונה נרשום בתיבה המתאימה את הכתובת שלהם בזיכרון, שהם המספרים של כל אוגר שפירטנו בסעיף ב' אלא שצריך לתרגמם מעשרוני לבינארי. rd הוא אוגר היעד בו נשים את החישוב, rs ו-rt הם שני האוגרים עליהם מפעילים את הפונקציה. נשים לב כי סדר האוגרים בשפת המכונה שונה מהסדר בפקודה בMIPS שהוא func rd, rs, rt.

shamt - מספר ההזזות שיש בפקודה. בפקודות אלו תמיד rs = 0.

func - זו הפונקציה המופעלת על האוגרים, תתורגם בעזרת קוד נתון.



המרה מעשרוני

המרה להקסה

**I-Type**:

פקודה הכוללת מספרים מידיים. פקודה זו היא מהצורה:

op - מציין את סוג הפקודה. ב-I-Type אין op קבוע כמו ב-R-Type אלא תלוי בסוג הפקודה.

rs, rt - הם האוגרים שמבצעים איתם את הפקודה. סדר האוגרים בשפת מכונה שונה מהסדר בפקודת MIPS שהוא func rt, rs, imm או func rt, imm(rs). בתרגום לשפת מכונה נרשום בתיבה המתאימה את הכתובת שלהם בזיכרון, שהם המספרים של כל אוגר שפירטנו בסעיף ב' אלא שצריך לתרגמם מעשרוני לבינארי.

imm - הוא המספר המידי בפקודה המכיל 16 ביטים.



**J-Type**:

פקודות מסוג קפיצה. פקודה זו היא מהצורה:

op - מציין את סוג הקפיצה.

addr - מציין את הכתובת אליה צריך לקפוץ. הכתובת היא באורך 26 ביטים.

**הערה** - אם נרצה להעביר את הפקודה הבינארית להקסדצימלי, נחלק את 32 הביטים לתאים כך שבכל תא יהיו 4 ביטים, סך הכל נקבל 8 תאים. ואז נתרגם כל תא להקסדצימלי, ונחבר את כל התאים. חשוב לזכור לשים ליד המספר שקיבלנו 0x המציין שזהו מספר הקסדצימלי.

## קפיצות

קפיצות הם פקודות המאפשרות לנו לבצע את הפקודות של התוכנה באופן לא רציף. באמצעות קפיצות אנו יכולים לבצע: תנאים, לולאות, פונקציות ועוד.

**תוויות:**

כאשר אנו משתמשים בפקודות קפיצה אנו צריכים לציין בפקודה היכן לקפוץ. ניתן לעשות זאת באמצעות כתובת הפקודה שאליה אנו רוצים לקפוץ (בדר"כ רשומה בהקסדצימלי). אך הדרך המקובלת יותר היא באמצעות תווית, שהיא שם שאנו מחליטים מהו, בדרך כלל שם המגדיר את הפעולה שאנו רוצים לבצע כמו loop, else, target, end וכו'. נגדיר את התווית מעל השורה שאליה נרצה לקפוץ על ידי כך שנרשום את שם התווית ולאחריה נקודותיים, לדוגמא "loop:". ניתן גם לקפוץ אל כתובת הרשומה באוגר מסוים כמתואר בהמשך.

**סוגי קפיצות:**

יש לנו 2 סוגי קפיצות: קפיצה מותנית וקפיצה שאינה מותנית. קפיצות מותנות מבוססות על השוואה. יש קפיצות שמשוות בין שני אוגרים bne, beq, bgt, blt,, ועוד, שבהם פורמט הפקודה הוא **beq $s3, $t2, loop**, המשווה בין $s3 ו-$t2 וקופצת ל-loop אם התנאי מתקיים. יש קפיצות שמשוות בין אוגר ל-0 כמוbgtz, bltz, bgez, blez, שבהם פורמט הפקודה הוא **bgtz $t1, target**, המשווה בין $t1 ל-0 וקופצת ל-target אם התנאי מתקיים.

קפיצות שאינן מותנות הן פקודות j, jr, jal, לכל אחת יש תפקיד שונה. j היא פקודה הקופצת אל מספר המציין כתובת או תווית, בפורמט j loop. פקודת jr קופצת אל כתובת הכתובה באוגר מסוים, בפורמט jr $s0. פקודת jal משמשת לפונקציות, נלמד עליה בהמשך.

**תרגום קפיצות לשפת מכונה:**

**קפיצה מותנית** - היא פקודת I-type, כי היא משווה בין שני אוגרים ואינה רק נותנת כתובת לקפיצה. האסמבלר בקפיצה מותנית לא מתרגם את התווית אל כתובת שורת הפקודה שאנו רוצים לקפוץ אליה, אלא לכמה שורות עלינו לקפוץ פחות 1. מורידים 1 מכיוון שבכל מקרה ברירת המחדל היא לקפוץ שורה 1. אם קופצים למעלה אזי התווית תתורגם למספר שלילי בשיטת משלים ל-2.

בדוגמא else נמצא 4 שורות מתחת לקפיצה המותנית (0x10, 0x14, 0x18, 0x1c, 0x20), לכן הוא יתורגם ל-3.

0x10 beq $t0, $0, else

...

0x20 else: (do something)

**פקודת jr** - היא פקודת R-type. \*\*צריך לפרט יותר כיצד.

**פקודות j, jal** - הם פקודות J-type. אם כן, בפקודת אלו יש לנו אופציה לכתובת באורך 26 ביטים, אמנם היות והכתובות הם תמיד כפולות של ארבע שני הביטים האחרונים תמיד יהיו 00, ולכן החליטו לוותר עליהם ולקבל עוד שני ביטים לכתובת, המערכת תשלים את שני האפסים אוטומטית. סך הכל ניתן לקפוץ לכתובת באורך של 28 ביטים. בדוגמא sum יתורגם לבינארית וכל המספר יזוז 2 ביטים ימינה, כל שהכתובת שתתקבל בהקסדצימלי היא: 0x00100028.

0x0040005c jal sum

...

0x004000a0 sum: (do something)

## תנאים ולולאות

תנאי if - נעשה זאת על ידי התנאי ההופכי. אם התנאי לא מתקיים נבצע את הפעולות הנצרכות, אם לא נדלג עליהם.

# $s0 = f, $s1 = g, $s2 = h

# $s3 = i, $s4 = j

bne $s3, $s4, L1

add $s0, $s1, $s2

L1: sub $s0, $s0, $s3

if (i == j)

f = g + h;

f = f – I;

תנאי if/else - כמו בתנאי if אלא שנוסיף קפיצה שתדלג על ה-else אם התנאי לא מתקיים, כדי לא לבצע את שני המצבים.

bne $s3, $s4, Else

add $s0, $s1, $s2

j done:

else:

sub $s0, $s0, $s3

done:

if (i == j)

f = g + h;

else

f = f – I;

לולאת while - סדרה של פעולות שבהתחלה בודקים תנאי, אם מתקיים מדלגים על כל הפעולות, ואם לא מבצעים את כל הרצף שבסופם יש קפיצה את לתחילת הרצף.

# $s0 = pow

addi $s0, $0, 1

addi $t0, $0, 128

while:

beq $s0, $t0, done

sll $s0, $s0, 1

j while

done:

int pow = 1;

while (pow != 128) {

pow = pow \* 2;

}

לולאת for - כמו בלולאת while אלא שיש איזשהו i שמקדמים כל איטרציה.

# $s0 = i, $s1 = sum

addi $s1, $0, 0

add $s0, $0, $0

addi $t0, $0, 10

for:

beq $s0, $t0, done

add $s1, $s1, $s0

addi $s0, $s0, 1

j for

done:

int sum = 0;

int i;

for (i=0; i!=10; i = i+1) {

sum = sum + i;

}

**הערה** - אם התנאי הנדרש אינו האם שני אוגרים שווים או לא או השוואה ל-0, נוכל להשתמש בפקודה slt בפורמט **slt $1, $2, $3**. פקודה זו בודקת אם $2<$3, אם כן אזי $1 יקבל את הערך 1, אחרת יקבל את הערך 0. כעת נוכל לבדוק האם $1 שווה 0 או לא.

## מערכים

מערך הינו רצף תאי זיכרון המכילים מידע. לכל מערך יש כתובת בסיס (base address) שהוא בעצם arr[0], וכל שאר התאים במערך הם ביחס אליו בכפולות של 4. לכן שמגדירים מערך יש קודם כל להכניס לתוך אוגר את הכתובת הראשונה במערך. מכיוון שהכתובת היא באורך 32 ביטים בדרך כלל, וניתן להכניס לתוך אוגר רק 16 בתים בכל פקודה, נשתמש בפקודת lui ואז בפקודת ori כמו שלמדנו ביצירת קבועים.

כדי לגשת לכל התאים במערך נעשה זאת באמצעות לולאת for עם אינדקס נוסף שבכל איטרציה יגדל ב-4, כך נוכל לגשת אל הכתובת הבאה במערך. לדוגמא אם נרצה להכניס את הערכים 1, 2, …, 10 אל מערך בגודל 10 שמתחיל בכתובת 0x10011000, נבצע זאת בדרך הבאה:

lui $s0, 0x1001 # load 0x1001 into upper half of s0

ori $s0, $s0, 0x1000 # load 0x1000 into lower half of s0

addi $s1, $0, 1 # $s1 is loop index

addi $s2, $0, 0 # $s2 is array index

addi $t1, $0, 10 # stop condition

for:

ble $s1, $t1, done # if $s1<=10 go again to done

sw $s1, 0x10011000($s2) # insert $s1 data into memory

addi $s1, $s1, 1 # s1++

addi $s2, $s2, 4 # $s2 = $s2+4

j for

done:

שימו לב שאין שימוש בכתובת הבסיס $s0 בקוד זה, אמנם לרוב כן נשתמש בכתובת הבסיס.

## פונקציות

כל פונקציה היא בעצם כתובת בזיכרון שלאחריה יש רצף מסוים של פקודות. כל שורת פקודה שהמחשב מבצע היא בעצם חלק מפונקציה. הפונקציה הראשית של התוכנה היא פונקציית main, וממנה אפשר לקרוא לפונקציות נוספות, כלומר לקפוץ לכתובת של הפונקציה הרצויה ולבצע את רצף הפקודות שלה. מתוך פונקציה אפשר לקפוץ לפונקציה אחרת ואפילו לפונקציה שאנו נמצאים בה (רקורסיה). בקריאה לפונקציה, הפונקציה שממנה קראנו נקראת caller, והפונקציה שקראנו לה נקראת callee. לאחר סיום כל הפקודות של הפונקציה נחזור לשורת הפקודה האחרונה שממנה הפעלנו את הפונקציה.

קפיצה לפונקציה נעשית באמצעות הפקודה **jal func**, כאשר func היא תווית המסמלת את שמה של הפונקציה ומורה לאן בזיכרון צריך לקפוץ.

**משתני פונקציה:**

כל פונקציה צריכה להחזיק את המידע הבא:

* **כתובת החזרה בסיום הפונקציה** - כלומר כתובת שורת הפקודה שאחרי השורה שקראה לפונקציה. הכתובת הזו נשמרת אוטומטית באוגר ra כאשר קופצים לפונקציה, שהוא אחד מהאוגרים המיוחדים. בסוף הפונקציה נקפוץ חזרה בעזרת פקודת jr $ra כדי לקפוץ לכתובת הרשומה באוגר $ra.
* **פרמטרים לפונקציה** - הם האוגרים שנשלחים לפונקציה בתור פרמטרים, ועליהם הפונקציה מבצעת פעולות. ניתן להכניס עד ארבעה פרמטרים, הנשמרים בארבעת האוגרים המיוחדים $a0-$a3. לפני שקוראים לפונקציה צריך לאתחל אותם לערכים הרצויים.
* **ערך חזרה מהפונקציה** - הוא הערך שאנו רוצים להחזיר מהפונקציה הנוכחית אל הקודמת. יש רק ערך אחד כזה הנשמר באוגר המיוחד $v0. לאחר שנחזור מהפונקציה נעביר מיד את המידע מאוגר זה אל אוגר אחר.

**שימוש במחסנית:**

צריך לא לדרוס אוגרים קיימים במהלך הפונקציה. ולכן בתחילת הפונקציה נקצה מקום במחסנית לכל האוגרים שאנחנו רוצים לשמור בצד. בסוף הפונקציה נחזיר חזרה את הערכים ששמרנו במחסנית לאוגרים שדרסנו, ונשחרר את הזיכרון שהקצנו. האוגרים שאנו רוצים לשמור הם:

* אוגרי s - שמשתמשים בהם גם בפונקציה הקוראת וגם בפונקציה הנקראת, נצטרך לשמור אותם במחסנית בפונקציה הנקראת כדי לא לדרוס אותם.
* אוגרי t - ייעודם הוא אוגרים זמניים, ולכן לא צריך לשמור אותם במחסנית בפונקציה הנקראת. אמנם אם הפונקציה הקוראת רוצה לשמור עליהם היא צריכה להכניס אותם למחסנית, ולא הפונקציה הנקראת.
* כתובת החזרה $ra - כאשר מתוך הפונקציה צריך לקפוץ לפונקציה אחרת, נשמור גם כן את הכתובת חזרה במחסנית, כדי שלא תידרס בקריאה לפונקציה השנייה.

השימוש במחסנית נעשה באופן הבא: מחסרים מראש המחסנית $sp את מספר האוגרים שאנו רוצים לשמור כפול 4 (כל אוגר הוא 4 כתובות). ומכניסים את האוגרים למחסנית. בסוף הפונקציה נחזיר מהמחסנית את כל האוגרים בסדר הפוך ממה שהכנסנו, ונשחרר את הזיכרון שהקצנו באמצעות הוספה ל$sp. לדוגמא:

Main:

addi $a0, $0, 3

addi $a1, $0, 4

addi $a2, $0, 5

addi $a3, $0, 6

jal Sum

add $s0, $v0, $0

j done

Sum:

addi $sp, $sp, -8 # space for 2 registers

sw $s0, 4($sp) # store S0 on stack[1]

sw $s1, 0($sp) # store S1 on stack[2]

add $s0, $a0, $a1 # $s0 = a+b

add $s1, $a2, $a3 # $s1 = c+d

add $v0, $s0, $s1 # $v0 = a+b+c+d

lw $s1, 0($sp) # restore S1 from stack

lw $s0, 4($sp) # restore S0 from stack

addi $sp, $sp, 8 # deallocate memory

jr $ra

done:

Int main (){

Int x = sum (3, 4, 5, 6);

}

Int sum (int a, int b, int c, int d){

return a+b+c+d;

}

**פונקציה רקורסיבית:**

מאוד דומה לפונקציה שקוראת לפונקציה כמו שהסברנו, אלא שבפונקציה רקורסיבית הפונקציה הקוראת צריכה לשמור גם את הפרמטרים שהיא קיבלה $a0-$a3 במחסנית.

Mult:

addi $sp, $sp, -12 # space for 3 registers

sw $a1, 8($sp) # store a1 (b) in stack

sw $a0, 4($sp) # store a0 (a) in stack

sw $ra, 0($sp) # store return address

addi $t0, $0, 2 # t0=2

slt $t0, $a1, $t0 # if a1<2 t0=1, else t0=0

beq $t0, $0, Else # a0<2

add $v0, $a0, $0 # return a

addi $sp, $sp, 12 # deallocate memory

jr $ra # return to caller function

Else:

addi $a1, $a1, -1 # b--

jal mult # recursive

lw $ra, 0($sp) # restore ra

lw $a0, 4($sp) # restore a

lw $a1, 8($sp) # restore b

addi $sp, $sp, 12 # deallocate memory

add $v0, $v0, $a0 # return a + func\_result

jr $ra

int mult (int a, int b){

if (b<=1)

return a;

else

return a + mult (a, b-1);

## מבנה הזיכרון

כל המידע שאנו משתמשים בו, פרט ל-35 האוגרים המיוחדים, נמצא בזיכרון. הזיכרון שאנו לומדים נמצא בכתובות 0x00000000 - 0xffffffff, שהם בתים, שזה 4GB. הזיכרון מחולק ל-4 חלקים (בתמונה משמאל ניתן לראות את החלוקה לכתובות):

1. **זיכרון שמור (Reserved)** - בשימוש על ידי מערכת ההפעלה.
2. **הוראות (Instructions)** - שם שמורות כל הפקודות של התוכנה. מכונה גם text.
3. **מידע סטטי/גלובלי (Static/Global Data)** - כל המידע שמוקצה לפני ריצת התוכנית.
4. **הקצאות דינמיות (Heap) -** כל המידע שמוקצה במהלך ריצת התוכנית. מחולק ל-2: מידע רגיל שהקצינו במהלך התוכנית ומידע ששמרנו במחסנית. ההקצאות הדינמיות מתחילות ב-0x10010000 ומטפסות כלפי מעלה, והמחסנית מתחילה ב-0x7fffffc ומטפסת כלפי מטה.

## משתנים גלובליים/סטטיים

כדי להשתמש במשתנים גלובליים נרשום בתחילת התוכנית .data"", זה מורה למעבד שאנו כותבים לחלק הזיכרון static data. ולאחר מכן נצהיר על המשתנים שאחריהם נקודותיים (כמו תוויות, לדוגמא counter:). נוכל לאתחל משתנים אלו לערכים אם נרשום לאחר ההצהרה ". word 3", כך נאתחל את המשתנה הגלובלי ל-3. לאחר ההצהרה על כל המשתנים נרשום .text"", זה מורה למעבד שאנו מתחילים את הקוד.

משתנים אלו הם משתנים שחיים לכל אורך התוכנית, ולאחר שהצהרנו עליהם נוכל להשתמש בהם בקוד בשם שהצהרנו עליהם. שימו לב, כמו שיש אוגר מיוחד $sp שמצביע לראש המחסנית, יש אוגר מיוחד $gp שמצביע על המקום הפנוי הראשון ב-static data.

.data # allocate memory

f: .word 0 # allocate f=0

g: .word 4 # allocate g=4

array: .space 12 # allocate 12 bytes of data called "array"

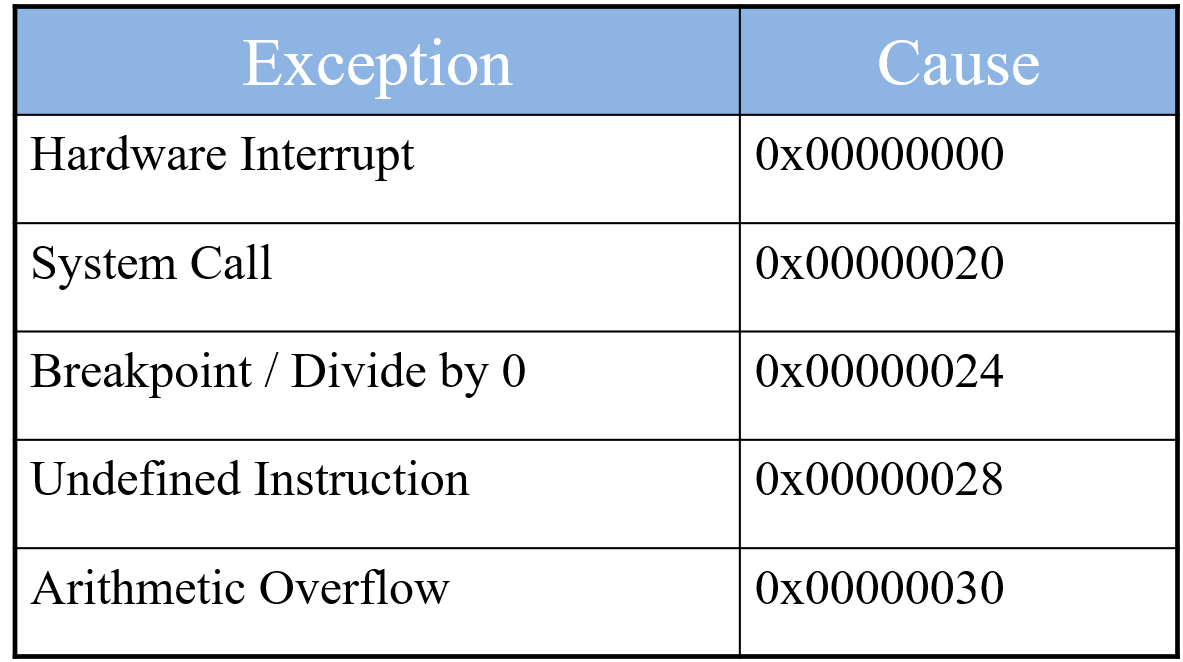
.text # code

…

## חריגות

חריגות יכולות לקרות משתי סיבות:

* בגלל החומרה, לדוגמא: הקשה על המקלדת. חריגה זו נקראת "interrupt" (הפרעה).
* בגלל התוכנה, לדוגמא: פקודה לא מוגדרת. חריגה זו נקראת "trap" (מבוי סתום).

כאשר המעבד נתקל בחריגה הוא מכניס את סיבת החריגה לאוגר בשם cause, ואת כתובת החריגה לאוגר בשם EPC. סיבות החריגה והערכים שיכנסו לאוגר cause מצורפות בטבלה. ולאחר מכן המעבד מפעיל קופץ לפונקציית Exception handlerשנמצא בזיכרון בכתובת 0x80000180.

ה-Exception handler מבצע את הפעולות הבאות:

1. שומר את כל האוגרים שהוא הולך להשתמש בהם במחסנית, כדי לא לדרוס ערכים השמורים בהם.
2. קורא את הודעת השגיאה מ-cause באמצעות פקודת mfc0 $k0, Cause.
3. מתמודד עם השגיאה.
4. מחזיר מהמחסנית את כל האוגרים ומשחרר את הזיכרון שהקצה.
5. חוזר לתוכנה מאיפה שהפסקנו באמצעות הפקודות: mfc0 $k0, EPC

jr $k0

## float ו-double

ניתן לעבוד בMIPS גם עם מספרים לא שלמים:

**float** - אוגר שמכיל 32 ביטים שחלקו מיועד לחלק השלם וחלקו לשארית. טיפוס זה הוא ברמת דיוק של 8 ספרות אחרי הנקודה.

**double** - מכיל שני אוגרי float, 64 ביטים סך הכל. כך שיכולת הדיוק שלו היא פי 2 משל float, כלומר 16 ספרות אחרי הנקודה.

יש סך הכל 32 אוגרים מיוחדים כאלו. שמם, מספרם ויעודם נמצא בטבלה:

הפקודות על אוגרים אלו זהה לפקודות ולפורמט שאנו מכירים אלא שצריך להוסיף ".s" בכל פקודה בשביל float, ובשביל double צריך להוסיף ".d".

פקודות float - add.s, sub.s, div.s, neg.s, abs.s, וכו'.

פקודות double - add.d, sub.d, div.d, neg.d, abs.d, וכו'.

בתרגום לשפת מכונה נקראות פקודות F-Type, והם מהצורה:

כאשר ה-OP של שניהם הוא 17 (010001). ה-COP של float הוא 16 (10000), וה-COP של double הוא 17 (10001).

## רכיבי הזיכרון

קיימים שלוש תכונות שהיינו רוצים שיהיו ברכיבי הזיכרון: זמן גישה קצר ומהיר, מקום רב, זול. אמנם בפועל אין רכיב זיכרון שמכיל את כל שלושת תכונות אלו, אלא רק חלק. לכן אנו משתמשים בשלושה רכיבי זיכרון: cache (מטמון) מאוד מהיר אך יקר מאוד. זיכרון מרכזי הוא ממוצע, לא יקר אך לוקח זמן לגשת אליו. זיכרון וירטואלי הוא מאוד זול אך לוקח זמן רב לגשת אליו.

**מדידת ביצועים:**

השימוש ברכיבי זיכרון אלו הוא בצורה היררכית, כלומר שמחפשים מידע קודם כל מחפשים אותו במטמון, אם לא מוצאים מחפשים בזיכרון המרכזי, ואם גם שם לא מוצאים מחפשים בזיכרון הווירטואלי. אם המידע נמצא ברמת זיכרון כלשהי נאמר שיש פגיעה ברמת זיכרון זו, אחרת נאמר שיש פספוס ברמת זיכרון זו. יש נוסחה לחישוב זמן גישה ממוצע לזיכרון - AMAT (Average Memory Access Time):

- זמן קריאה מהמטמון. נמדד במחזורים (cycles). בדרך כלל מחזור 1.

- אחוז הפספוס במטמון, כלומר כמות החיפושים שלא נמצאו במטמון חלקי סך החיפושים הכללי.

- זמן קריאה מהזיכרון המרכזי. גם כן נמדד במחזורים.

- כמות החיפושים שלא נמצאו בזיכרון המרכזי חלקי החיפושים בו, שהוא מספר הפספוסים במטמון.

- זמן קריאה מהזיכרון הווירטואלי. אם לא נתון הכוונה שווה ל-0.

## cache

זוהי הרמה הכי גבוהה בזיכרון. לרוב מחזיק את רוב המידע שהתוכנה משתמשת בו, וכן מחזיק את המידע הכי עדכני מבחינת שימוש.

**עקרון המקומיות:**

עקרון שלפיו נדע איזה מידע לשמור במטמון. מחולק ל-2:

**מקומיות בזמן** - אם היה לאחרונה צורך לשלוף מידע מהזיכרון המרכזי/וירטואלי (באמצעות פקודת lw), כנראה שבקרוב נצטרך להמשיך להשתמש בו. לכן, את כתובת הזיכרון שממנה שלפנו את המידע ואת המידע עצמו לא נחזיר לזיכרון מרכזי/וירטואלי, אלא נשים אותם במטמון כדי שהגישה אליהם תהיה זריזה.

**מקומיות במקום** - אם היה צורך לשלוף מידע מהזיכרון המרכזי/וירטואלי, כנראה שנצטרך בקרוב גם את מה שבאזור שלו. לכן, נשלוף לא רק אותו אלא גם קצת קדימה וקצת אחורה, ונשים הכל יחד במטמון. גודל המידע שנמשוך תלוי בכמה בתים ניתן להכניס למטמון בבת אחת (נקרא בלוק, בסעיף הבא נלמד על כך), כאשר ניגש למילה כלשהי כל המילים בגודל בלוק יכנסו למטמון.

**מונחים:**

קיבולת - כמה בתים (כתובות) יש במטמון. מסומן ב-c.

גודל בלוק - כמה בתים ניתן להכניס למטמון בבת אחת. מסומן ב-b. כמות הבלוקים במטמון תחושב: .

רמת השייכות - כמה בלוקים נכנסים בקבוצה. מסומן ב-N.

קבוצות - המטמון מחולק לקבוצות. כמות הקבוצות מסומנת ב-S, ומחושבת כך: .

לכל כתובת מילה בזיכרון, מכל רמות הזיכרון, מתאימה קבוצה במטמון. יש שלוש שיטות כיצד לחלק את הבלוקים לקבוצות:

**מיפוי ישיר - Direct map:**

כל קבוצה היא בגודל בלוק אחד. לדוגמא: אם במטמון יש 32 בתים (8 מילים), וכל בלוק הוא בגודל 4 בתים (מילה 1), אזי מספר הקבוצות הוא 8. נמספר כל קבוצה במטמון מ-0 עד S-1.

כיצד נדע איך להתאים כל כתובת בזיכרון לקבוצה במטמון? אם נחלק כל אחת מהכתובות ב-S ונבחן את השארית של כל חלוקה כזו, נקבל בדיוק S מחלקות שקילות. כל מחלקת שקילות כזו תיכנס לקבוצה אחרת במטמון בהתאמה. דרך נוספת להסתכל על חלוקה זו, היא לחשב כמה ספרות בינאריות צריך כדי לייצג את כל הקבוצות במטמון, נעשה זאת באמצעות החישוב . כל הכתובות של מילים בזיכרון הם כפולה של 4, לכן כל הכתובות נגמרות ב-00. באמצעות d הספרות שלאחר שתי הספרות האלו נתאים כל מילה אל הקבוצה המתאימה לה במטמון. לפי מספר הביטים שנותרו נדע כמה מילים יש בכל קבוצה. בהמשך לדוגמא הקודמת אם יש בזיכרון בתים, וצריך 3 ספרות בינאריות לייצג 8 קבוצות במטמון, אזי בכל קבוצה יהיו מילים. 27 הספרות האלו נקרא "התיוג" (Tag) של הקבוצה, כך אנו בעצם יודעים איזה כתובת ספציפית יושבת בקבוצה במטמון.



מקובל לצייר מטמון כמו בתמונה משמאל. כאשר Tag מורה איזה כתובת של מילה יושבת בקבוצה, Data מורה על איזה מידע יושב במטמון, ו-v מורה האם התא הזה בשימוש או לא. אם התא בשימוש אזי v=1 אחרת v=0. בכל הדלקה של המחשב המטמון תמיד מאופס לכל כל הטור של v הוא 0.

כאשר אנו טוענים מילה מכתובת מסוימת בזיכרון, 27 הביטים הראשונים מכתובת זו יכנסו ל-Tag, המידע ששלפנו יכנס ל-Data, ו-v יהפוך ל-1. אם נטען שתי מילים המתאימות לאותה קבוצה במטמון בזה אחר זה, המילה השנייה תדרוס את הראשונה.

כאשר בלוק מכיל יותר ממילה אחת כל מילה בקבוצה תקבל מספר בינארי כדי שנוכל להבדיל בין המילים בקבוצה. אם לדוגמא יש מטמון המכיל 8 מילים ובלוק הוא בגודל 4 מילים, כך שיש 2 קבוצות, אזי חלוקת הביטים בכתובת תיעשה כך: 2 הביטים הראשונים תמיד 00, שני הביטים אחריהם זו החלוקה הפנימית בתוך הבלוק של המילים (Block Offset), והביט הבא אחריו מורה על איזו קבוצה מתאימה במטמון. כל שאר 27 הביטים הם התיוג (Tag).

**N-Way:**

כל קבוצה במטמון מכילה N כלשהו של בלוקים ולא בלוק אחד. אם לדוגמא המטמון מכיל 8 מילים ובלוק הוא בגודל מילה, אז אפשרי שכל קבוצה תכיל N=2 בלוקים. גם כאן כל כתובת בזיכרון מתאימה לקבוצה אחת במטמון, כמו שהסברנו לעיל, אלא שעכשיו כל קבוצה מכילה כמה מילים. לכן רק כאשר נכניס לקבוצה את כתובת המילה ה-N+1 אחת המילים הקיימות יימחקו.

כאשר מתקבלת בקשה לשליפת זיכרון עם כתובת מסויימת, תיבדק רק הקבוצה המתאימה לכתובת זו לראות אם באמת קיימת שם (הבדיקה היא רק על ה-Tag ולא על כל הכתובת!), כיצד המידע יישלף? בעזרת רכיב MUX שיאפשר למידע הנכון לצאת. שיטה זו אמנם מאפשרת יותר מקום במטמון כך שיהיו פחות פספוסים, אך פעולת השליפה וההשוואה בביטים של Tag ארוכה יותר, ולכן יכולה לקחת יותר זמן.

**Fully:**

קבוצה יחידה במטמון שכל המידע נכנס אליה. בחיפוש מידע צריך לחפש בכל הקבוצה בשביל למצוא את הTag- הנכון, כי אין חלוקה לקבוצות. רק כאשר טוענים יותר מילים מהמקום במטמון תתרחש דריסה.

מטמון כזה דורש הרבה חומרה בשביל לבצע את ההשוואות בין כל הביטים כדי למצוא את ה-Tag הנכון. מטמון כזה יעיל בעיקר כשמדובר על מטמון עם מעט זיכרון יחסית, כדי שכמות ההשוואות לא תגרום יותר נזק מתועלת.

## פספוסים

יש שלושה סוגי פספוסים:

1. פספוס הכרחי - בפעם הראשונה שניגשים למידע המטמון ריק ולכן בוודאי שיהיה פספוס.
2. פספוס קונפליקט - כאשר ניגשים לשני מילים המתאימות לאותה קבוצה במטמון השני דורס את הראשון. לכן אם ניגש למידע הראשון נקבל פספוס.
3. פספוס קיבולת - כל גישה למידע שאינו נמצא במטמון תכניס אותו למטמון. אמנם כאשר המטמון כבר מלא צריך למחוק ממנו מידע כדי להכניס את המידע החדש. לכן אם ננסה לגשת למידע שמחקנו יהיה פספוס. כיצד נדע מה למחוק? אם בקבוצה המתאימה למידע החדש יש בלוק יחיד, אז נמחק את מה שצריך כדי להכניס את המידע החדש. אם יש יותר מבלוק אחד נמחק מידע מהבלוק הכי לא משומש בקבוצה.

השפעות הפרמטרים של המטמון (c, b, N) על הפספוסים:

* אם נגדיל את קיבולת המטמון c נצמצם את פספוסי הקיבולת והקונפליקט, אך לא נשפיע על הפספוסים ההכרחיים.
* אם נגדיל את גודל הבלוק b נצמצם את הפספוסים ההכרחיים, אולם מצד שני זה עלול להגדיל את פספוסי הקונפליקט כי יותר כתובות יתאימו לאותה קבוצה.
* אם נגדיל את כמות הבלוקים בכל קבוצה N נוכל לצמצם את פספוסי הקונפליקט, אך אין השפעה על שני הפספוסים האחרים.