# פרויקט ISA – דוקומנטציה

מגישים:

- אור לופטה 308256304
- עדי פולק 3085522967

במסמך זה נתאר את צורת הפעולה של התכניות השונות בפרויקט הISA:

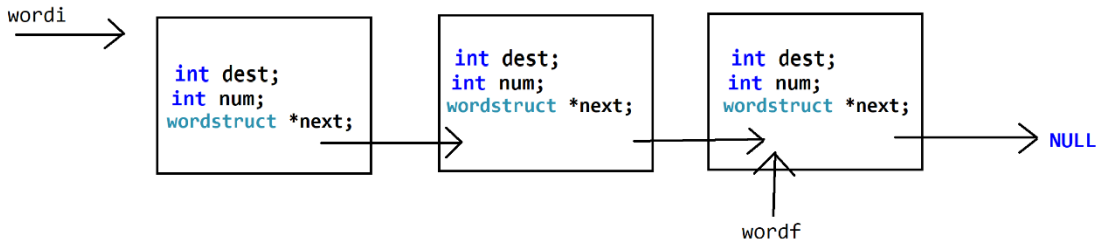
# אסמבלר- דוקומנטציה

האסמבלר עובר שורה-שורה לאורך קוד האסמבלי תוך דילוג על שורות לא תקינות – והעברת הטקסט למערך דו ממדי של chars. כל "שורה" במערך היא string שהוא שורה בטקסט. בנוסף, נספרות מספר השורות.

מתבצע מעבר ראשוני על מערך השורות ובו מזוהות ונמחקות הערות (לפי התו '#'). בנוסף, נשמרות פקודות ה"word." בעזרת רשימה מקושרת של struct מסוג "wordstruct" שהגדרנו. בנוסף, באותה צורה, נוצרת רשימה מקושרת של struct מסוג "label" .

## תרשים מבני הנתונים:

### Wordstruct:




משתנים:

**Dest** - הכתובת בה יש לכתוב את המלה בזיכרון כאשר במעבר נספרות מספר השורות באמצעות המשתנה של הלולאה, ופקודות שמכילות immediate נספרות באמצעות המשתנה immid_command שהרי פקודות אלה נשמרות ב2 שורות.

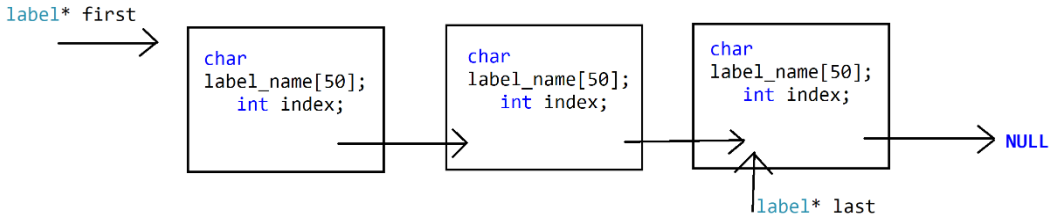
**Num** - המספר אותו יש לכתוב לזיכרון.

**Wordi** - מצביע לתחילת ברשימה

**wordf** - מצביע סוף הרשימה

המילים יווצרו וישמרו באמצעות "make_word", ימוינו באמצעות הפונקציה "sortwordbyindex" ויודפסו בסוף כל ההדפסות של שורות הפקודות באמצעות "add_words_to_print" .

### Label:




משתנים:

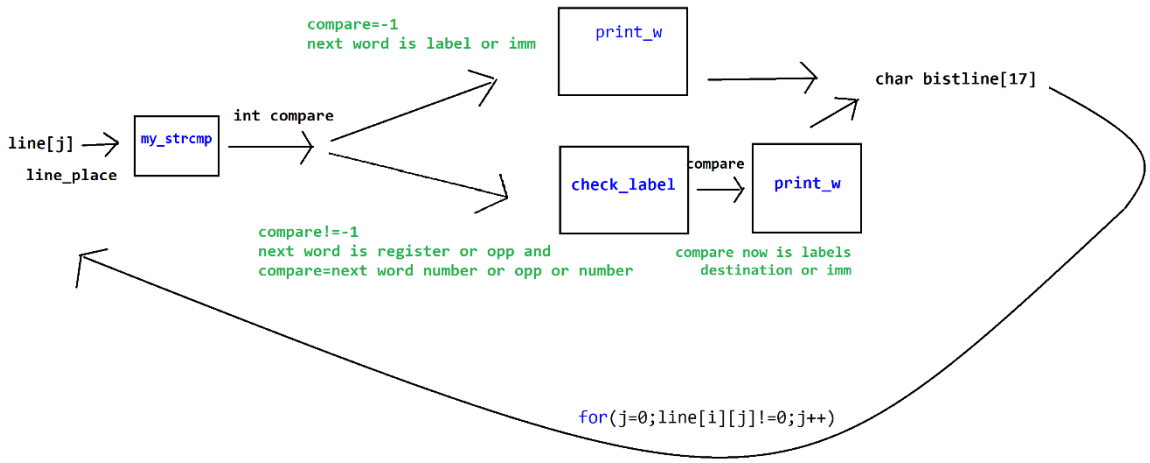
**Label_name** - שם הסימנייה

**index** - השורה שהסימנייה נמצאת בה – מחושבת באמצעות ספירת השורות וספירת 'immid_command'.

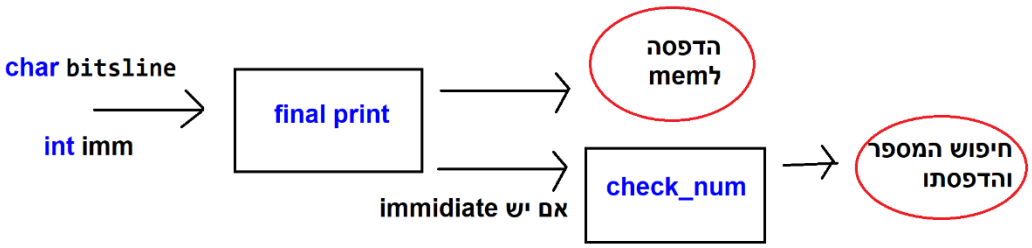
**First-** מצביע לתחילת ברשימה

**last** - מצביע סוף הרשימה

לאחר מכן מתבצע מעבר נוסף על מערך השורות, וכל שורה מתורגמת ראשית ל string (מערך של chars שהוא רצף של '0' או '1') במערך bitsline[17] char שיסתיים ב'\n' , לאחר מכן ל char hex[5]- מערך של המספר בhex, עבור שורת אסמבלי שנמצאת ב line[j]. נדמה את התהליך באמצעות תרשים:

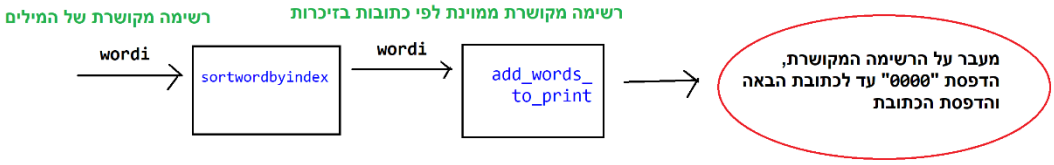



bitsline[17] char יכיל את הקידוד המתאים עבור כל שורה. לשם כך מורצת לולאה על השורה line[j] והמילה הבאה נבדקת באמצעות my_strcmp . אם המילה הבאה היא רגיסטר או פקודה, הפונקציה תחזיר את מספר הרגיסטר או הפקודה, אם המילה הבאה היא לא אחת מאלה, הפונקציה תחזיר 1- ונעביר את השורה בפונקציה check_label שמבצעת בדיקה לlabel. אם המילה הבאה היא שם של label, תיסרק רשימת ה labels מהמעבר הראשון ויוחזר מיקום ה label המבוקש. בנוסף, במהלך המעבר, ייבדק האם יש immediate בשורה באמצעות המשתנה imm בכדי להבין האם להדפיס שתי שורות בהמשך. בשני המקרים יתקבל המספר הבא שיועבר לפונקציה print_w שהופכת את המספר מדצימלי לבינארי וממקמת במקום הנכון ב bitsline. כעת הודפסו 4 תווים לbitsline ומתבצעים קידומי משתנים. כל התהליך חוזר על עצמו בלולאה. הלולאה מסתיימת כאשר מגיעים בbitsline ל16 תווים – אורך שורה תקנית.



מערך הביטים עובר ל final_print שהופך את המערך מבינארי למערך הקסדצימלי ( char hex[5] ) באמצעות decimal_to_hex וידפיס את המערך. הבדיקה imm בודקת האם בשורה היה immediate, במידה וכן, הimmediate נבדק באמצעות check_num ומודפס.

לאחר סיום הדפסת הפקודות מתחילה הדפסת המילים (.word). ההדפסה צריכה להיות לפי כתובות ולכן מבצע מיון לרשימה המקושרת של המילים לפי כתובות- sortwordbyindex. לאחר המיון, הפונקציה add_words_to_print עוברת על רשימת המילים הממוינת ומדפיסה אותם במיקום הנכון. במהלך ההדפסות הקודמות נספרו מספר השורות המודפסות באמצעות int printed, לכן יודפסו שורות אפסים עד למיקום הבית אותו יש להדפיס.




בסיום ההדפסות, הזיכרון שהקצנו באמצעות –
free_word, free_label, free_line

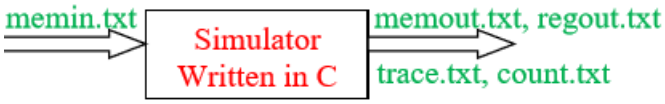
**פונקציות:**

```
struct wordstruct *sortwordbyindex(struct wordstruct*  wordi);
void print_w(char * line,int compare,int place,char *bitsline);
void add_words_to_print(FILE *memory,int printed,struct wordstruct* wordi);
int final_print(char *bitsline,FILE* memory,int imm,char *line,struct label* first);
void swap(struct wordstruct* a, struct wordstruct* b);
void decimal_to_hex(char* hex,int decimal, int neg);
void make_word(char* line,struct wordstruct* last, int *wor);
void free_word(struct  wordstruct* wordi);
void free_label(struct  label* first);
void free_line(char **line);
int my_strcmp(char *line,char *(opps[]), char *(registers[]),int place,int line_place);

int check_label(char *line,int a,struct label* first,char* bitsline );
int str_cmp(char * arr,char * opps); // used inside "my_strcmp" to compare between 1
strings in a specific way.
Int check_num(char *line ,struct label* first);
```

## סימולטור – דוקומנטציה

מטרת הסימולטור הינה לקרוא את הקובץ memin.txt המייצג תמונת זיכרון, ולבצע את פעולות האסמבלי המקודדות בו החל משורה 0. לאחר סיום ביצוע הפעולות, הסימולטור יחזיר כפלט ארבעה קבצים, כמתואר בדיאגרמה הבאה:




בחלק זה יתוארו הפונקציות בהן הסימולטור נעזר ודרך פעולתן. בסופו יפורטו כל הפונקציות שנכתבו עבור הסימלטור, אך ראשית, נתאר את מבני הנתונים בהם נעשה שימוש בסימולטור.

## מבני הנתונים בסימולטור

### Global array: int memArr

כאמור, הסימולטור מדמה פעולת מעבד העובר על פני כתובות שונות בזיכרון נתון, מפענח את הפקודות הכתובות בזיכרון, ומבצע אותן. את הזיכרון הסימולטור יקבל כקובץ טקסט.

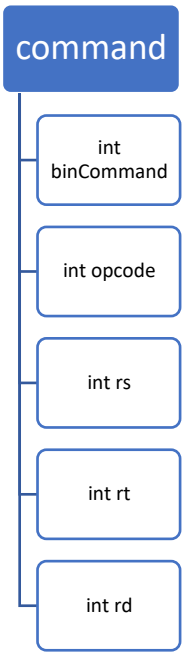
לשם כך הוגדר מערך **גלובלי** של int-ים בשם memArr וגודלו כגודל הזיכרון המדומה – 65536 תאים.

כאשר הקובץ נקרא על ידי הסימולטור, תוכנו מועבר אל המערך הנ"ל, כמפורט בתיאור הפונקציה void memToArr.

### Struct: command

פיענוח ההוראה אותה נרצה לבצע ייעשה אל תוך מבנה נתונים בשם command שנכתב לסימולטור. מבנה נתונים זה מכיל חמישה משתנים מסוג int המייצגים את הפרמטרים השונים של כל פקודה: המספר הגולמי המייצג את הפקודה במלואה (אך ללא ערך הimmediate במידה ונעשה בו שימוש), הopcode של הפקודה, וערכי הrs, rt ו-rd של הפקודה.

עבור פירוט על הדרך בה הנתונים מחושבים מתוך ערך הפקודה הגולמי, ראה פירוט בהמשך על הפונקציות int scanArr ו- int excecute_command .




**הפונקציות בסימולטור** – הסבר מפורט

כעת נתאר את הפונקציות השונות בהן נעשה שימוש בסימולטור.

**Int main(int argc, char *argv[])**

מטרת פונקציה זו היא לדאוג לפתיחת הקבצים השונים, לקרוא לתתי-הפונקציות שמבצעות את פעולות הסימולטור , ובסיום ריצן לסגור את הקבצים בהם אין יותר שימוש. הדיאגרמה הבאה ממחישה את פעולת פונקציה זו, שעוברת קובץ-קובץ ועבור כל אחד מבצעת את השרשרת המחזורית: פתיחת קובץ – ביצוע פונקציה רלוונטית – סגירת קובץ.


בנוסף, נוצרים בmain מספר משתנים ומערכים בהם נעשה שימוש לאורך התכנית בפונקציות השונות. המרכזיים שבהם הינם:

- Int count – משתנה המחזיק את מספר הפעולות שמבוצעות בתכנית
- Int numOfLines – משתנה המחזיק את מספר השורות שייכתבו בסופו של דבר אל הקובץ memout. אנו מניחים ששורות ריקות בקבצי הטקסט של הזיכרון משמען ערך '0', ולכן, עבור קריאות, נחזיק את משתנה זה ונכתוב לקובץ הזיכרון בסיום רק את מספר השורות שכתוב במשתנה.
- int regs[16] – מערך זה מכיל את הרגיסטרים השונים לאורך ריצת התכנית.

**Void memToArr(FILE *memin, int *numOfLines)**

פונקציה זו מקבלת מצביע לקובץ הטקסט המתאר את הזיכרון, וממלאת בעזרתו את המערך הגלובלי המייצג את הזיכרון. הקריאה מהקובץ נעשית ישירות כמספר הקסדצימלי.

לצורך תצוגה נוחה יותר בהמשך הריצה, הפונקציה משתמשת במשתנה בשם numOfLines. משתנה זה מחזיק לאורך ריצת הסימולטור את מספר השורות המינימלי אותו יצטרך לכתוב לזיכרון בסיום התוכנית, שכן אין צורך לכתוב לקובץ הפלט memout.txt את כל שורות הזיכרון – שורות חסרות משמען תאים המכילים אפסים בלבד.

**Int scanArr(int *pc, int *regs, FILE *trace, int *numOfLines)**

פונקציה זו אחראית על סריקת מערך הזיכרון. היא מתחילה את הסריקה מכתובת 0, אותו היא מחזיקה במשתנה ה-pc, ורצה בלולאה אינסופית. בכל איטרציה של הלולאה היא לוקחת את השורה בזיכרון אותה יש לבצע, מכניסה אותה לשדה הbinCommand של משתנה מסוג command (struct – ראה פירוט בחלק המתאים), ולאחר מכן קוראת לפונקציה execute_command (ראה פירוט בהמשך). פונקציה זו מבצעת את הפעולה הנוכחית מהזיכרון, ומחזירה ערך עבור flag המציין האם יש להמשיך את הלולאה או להפסיק (דבר שקורה כאשר מתבצעת פעולת halt).

בנוסף, מתבצעים בלולאה של scanArr קידום ערך הpc וספירת הפעולות שמבוצעות בריצת התוכנית. הספירה מתבצעת בcounter אותו הפונקציה מחזירה לפונקציית הmain.

**Void fillRegisters**

פונקציה זו אחראית על כתיבה לקובץ regout.txt המכיל את מצב הרגיסטרים בסיום ריצת התכנית.

הכתיבה נעשית באמצעות לולאה הרצה על מערך הרגיסטרים, ומדפיסה אותם בקידוד המורחב לשמונה ספרות הקסדצימליות.

**Void copyToMemout(FILE *memout, int numOfLines)**

בדומה לfillRegisters, פונקציה זו מעתיקה אל קובץ טקסט את תמונת הזיכרון בסיום הריצה. היא מבצעת זאת בעזרת משתנה בשם numOfLines שרץ לאורך כל התכנית וסופר את מספר השורות המינימלי שניתן לכתוב אל הקובץ ללא פגיעה במידע. זה נעשה באמצעות ההנחה ששורות שאינן נכתבות אל קובץ הזיכרון – משמען ערך '0'.

**Branch commands**

ישנן מספר פונקציות שבודקות האם תנאי מסוים מתקיים, ואם כן – מבצעות קפיצה אל כתובת בזיכרון (כולן פונקציות void, בסיום המסמך מופיע פירוט של כולן):

- beq – פונקציה המבצעת קפיצה אם ערך rs שווה לערך rt
- bgt – פונקציה המבצעת קפיצה אם ערך rs גדול מערך rt
- ble – פונקציה המבצעת קפיצה אם ערך rs קטן או שווה לערך rt
- bne – פונקציה המבצעת קפיצה אם ערך rs שונה מערך rt

קפיצה בזיכרון מתבצעת אך ורק אם הכתובת אליה קופצים קטנה מnumOfLines, אחרת התכנית תתקע (פקודת 0000). בנוסף, מתבצע בפקודה זו קידום של ערך הPC אל המקום המתאים.

**Void sra(int *rd, int *rs, int *rt)**

פונקציה זו מבצעת shift ימינה תוך שמירה על סימן המספר בייצוגו העשרוני. שמירת הסימן מתבצעת באופן הבא:

ראשית כל נבצע את ההזזה. לאחר מכן נחזור לרגיסטר שמכיל את הערך המקורי, לפני ההזזה, ונעתיק את הMSB שלו אל הMSB של הרגיסטר עם המספר שאחרי ההזזה, באמצעות פקודות AND ו-OR.

**Void lw(int *memPtr, int *regDes)**

פקודה זו טוענת מהזיכרון ערך אל תוך רגיסטר. כיוון שערכי הזיכרון מיוצגים על ידי 4 ספרות הקסדצימליות, בעוד הרגיסטר מכיל 8 ספרות כאלו, טעינת מספר שלילי דורשת פעולה מיוחדת, שכן העתקה פשוטה תגרום לכך שהמספר ברגיסטר ייקרא כמספר חיובי (הביט השמאלי ביותר ברגיסטר יהיה 0 ולא 1 כנדרש). נבחרה השיטה הבאה:

ראשית, מתבצעת בדיקה האם המספר בזיכרון הינו שלילי על ידי פעולת AND עם מספר שמכיל '1' בביט מס' 15 ו'0' בכל האחרים. בצורה זו נדע האם המספר אותו אנו טוענים אמור להיות שלילי. אם הוא אינו שלילי, הטעינה התבצעה כראוי. אם לא, נהפוך את כל 16 הביטים השמאליים ל'1' על ידי פעולת OR עם המספר ההקסדצימלי 0xFFFF0000.

**Void sw(int *memPtr, int *regDes, int *numOfLines, int 8ewline)**

פונקציה זו שומרת בזיכרון ערך הנמצא באחד הרגיסטרים. כיוון שהייצוג בזיכרון הינו ע"י 16 סיביות (ולא 32 כמו ברגיסטרים), נאפס את 16 הביטים השמאליים ע"י פעולת AND עם 0xFFFF.

בנוסף, נעדכן בפונקציה זו את מספר השורות המינימלי אותן צריך לכתוב לזיכרון.

**Void lhi(int *regDes, int *regSrc)**

פונקציה זו מכניסה לחצי השמאלי של ערך רגיסטר כלשהו rd את החצי הימני של רגיסטר כלשהו rs. לשם כך, נלקח ערך הרגיסטר rs, מתבצעת הזזה שלו שמאלה ב16 ביטים (כדי להעביר את החצי הימני שלו לחצי השמאלי של ערך הרגיסטר המעודכן), ולבסוף חיבור של החצי הימני של רגיסטר rd המקורי. כך נקבל את rd המעודכן.

**Int excecute_command(struct command newComm, int *regs, int *pc, FILE *trace, int *numOfLines)**

הפונקציה מקבלת struct מסוג command, שהערך היחיד שמלא בו בתחילתה הינו הbinCommand, כלומר, הייצוג הגולמי של הפקודה בזיכרון. תפקידה לבצע את הפקודה, ולכן היא מקבלת פרמטרים נוספים בהן ניתן לעשות שימוש בעת ביצוע הוראה:

- מצביע לקובץ הtrace אותו יש לעדכן בכל ביצוע הוראה.
- מספר השורות המינימלי אותו נרצה לכתוב לקובץ הmemout בסיום הריצה

- המצביע לפקודה הנוכחית בזיכרון (PC)
- מערך הרגיסטרים במעבד

ראשית, הפקודה מעדכנת את הרכיבים השונים של הstruct באמצעות פעולות AND (לאיפוס החלקים בפקודה הגולמית שלא מעניינים אותנו) ו-shift.

לאחר מכן, נרצה לבדוק האם הפקודה כוללת שימוש בimmediate. אם כן, נעדכן את ערך הPC וניקח מהזיכרון את המספר שמייצג את ערך הimmediate. לאחר מכן, נבדוק האם הוא מייצג מספר שלילי. אם כן, נהפוך את 16 הביטים השמאליים שלו ל'1', שכן המספר מיוצג בזיכרון ע"י 16 ביטים אך לאחר הטעינה הוא מיוצג ע"י 32, ונרצה לשמור עליו שלילי.

לפני ביצוע הפקודה נכתוב בקובץ הtrace את ערך הPC המקורי, ואת הפקודה שתתבצע בנוסף לimmediate בו היא משתמשת. נכתוב את ערכי הרגיסטרים לפני ביצוע הפקודה, כנדרש, ולבסוף נבצע את הפקודה. כל פקודה מיוצגת על ידי פונקציה נפרדת, והקריאה לפונקציה מתבצעת בהתאם לopcode שפוענח. לסיום הפונקציה תחזיר '1' בכדי להמשיך לקריאת הפקודה הבאה בscanArr (הפונקציה שקראה לה), אלא אם כן הפקודה שבוצעה הינה פקודת halt שעוצרת את התכנית – ואז היא תחזיר '0'.

## רשימת הפונקציות המלאה בסימולטור

```
void add(int *rd, int *rs, int *rt);

void sub(int *rd, int *rs, int *rt);

void and(int *rd, int *rs, int *rt);

void or (int *rd, int *rs, int *rt);

void sll(int *rd, int *rs, int *rt);

void sra(int *rd, int *rs, int *rt);

void beq(int *rd, int *rs, int *rt, int *pc, int *numOfLines);

void bgt(int *rd, int *rs, int *rt, int *pc, int *numOfLines);

void ble(int *rd, int *rs, int *rt, int *pc, int *numOfLines);

void bne(int *rd, int *rs, int *rt, int *pc, int *numOfLines);

void jal(int *rd, int *pc, int *r15, int *numOfLines);

void lw(int *memPtr, int *regDes);

void sw(int *memPtr, int *regDes, int *numOfLines, int newLine);

void lhi(int *regDes, int *regSrc);

void copyToMemout(FILE *memout, int numOfLines);
```

```
void fillRegisters(FILE *regout, int *regs);


void memToArr(FILE *memin, int *numOfLines);

int excecute_command(struct command newComm, int *regs, int *pc, FILE *trace, int
*numOfLines);

int scanArr(int *pc, int *regs, FILE *trace, int *numOfLines);
```

# תכניות הבדיקה באסמבלי – דוקומנטציה

בדיקת תקינות התכנית התבצעה באמצעות 3 תכניות אסמבלי.

## תכנית div

תכנית זו מבצעת חלוקה בין שני מספרים שלמים חיוביים. לכן, עם טעינת המספר המחולק מהזיכרון, "נתקן" אותו במידה והMSB שלו בקידוד לארבע ספרות הקסדצימליות (8 ביט) הינו '1'. הסיבה היא שהטעינה מהזיכרון לרגיסטר (בן 16 ביט) תהפוך אותו לשלילי, ונרצה להחזיר אותו לייצוג שלו כמספר חיובי.

החלוקה מתבצעת באמצעות לולאה, בה בכל איטרציה אנו מפחיתים את המספר המחלק מהמספר המחולק, עד שהמספר המחולק נהיה קטן יותר מהמחלק. כאשר מצב זה מזוהה, מספר פעולות החיסור הינו תוצאת החילוק השלם, והמספר המחולק לאחר פעולות החיסור הינו השארית.

בוצעו בדיקות עבור שני מספרים כאשר יש שארית וכאשר אין שארית, ועבור מספרים שהMSB שלהם בייצוג 8 ביט הינו '1'.

## תכנית bubble

התכנית מבצעת מיון בועות עבור מספרים שליליים וחיוביים כאחד. משתמשים בoffset בסריקת המערך כדי לגשת לאיברים השונים. התכנית מבצעת את אלגוריתם מיון הבועות כהגדרתו ללא שינויים מיוחדים.

התבצעו בדיקות עבור מספרים חיובים, שליליים ואפסים, כאשר פקודות הword. מפוזרות לאורך התכנית, וחלק מהבתים בזיכרון מכילים מספרים זהים, לעיתים בייצוג עשרוני ולעיתים בייצוג הקסדצימלי.

## תכנית pascal

מטרת התכנית הינה חישוב השורה ה-n במשולש פסקל.

בתחילה מתבצעת כתיבה לזיכרון של השורה ה-0 במשולש. לאחר מכן נסרק קלט התכנית, והתכנית מתחילה בחישוב של כל השורות של משולש פסקל, עד שהיא מגיעה לשורה המבוקשת בקלט.

החישוב מתבצע עבור כל שורה מהסוף להתחלה, כך שאין דריסה של מידע באמצע תהליך החישוב. כלומר – כל איבר חדש דורס איברים מהשורה הקודמת לו, אבל אלו רק איברים שאין בהם צורך להמשך חישוב אותה שורה.