מבנה המחשב- פרויקט BASYS: דוקומנטציה

שנת הלימודים תשפ"ג, סמסטר ב'

D or on

Noa

Nadav

Daniella

בפרויקט זה נממש אסמבלר וסימולטור, ונכתוב תוכניות בשפת אסמבלי עבור מעבד RISC בשם SIMP. הפרויקט חולק לשלושה חלקים: החלק של האסמבלר ,סימלוץ המעבד, ותוכניות הבדיקה. נפרט על כל חלק בנפרד.
כמו כן- נפרט על העדכונים שביצענו עבור הסימולטור והאסמבלר לפרויקט השני והתאמתם לפסיקות ופעולות על הכרטיס.

האסמבלר

מטרת האסמבלר היא לתרגם את תוכנית האסמבלי שכתובה בטקסט בשפת אסמבלי לשפת המכונה וליצור את תמונת הזיכרון בקובץ memin.txt.

האסמבלר מקבל קובץ קלט program.asm- המכיל את תוכנית האסמבלי, ויוצר קובץ פלט memin.txt המכיל את תמונת הזיכרון, ומשמש בהמשך כקובץ הקלט של הסימולטור.

נסביר את אופן פעולת האסמבלר:
כדי לשמור את תמונת הזיכרון המלאה לקוד האסמבלי הדרוש, האסמבלר מבצע שני מעברים על שורות הקוד.
במעבר הראשון נזהה את כל השורות שמכילות Label, ונשמור במבנה נתונים בשם label_struct (עליו נרחיב בהמשך) את שם הLabel ומיקומה בזיכרון.
במעבר השני נעבור על כל אחת מהשורות, ונחלק אותה לשדות: label, opcode, rs, rd, rt, imm.
בהתאם לשדות שהתקבלו עבור שורה ובעזרת המרה בין שמות הפעולות והרגיסטרים למספרים המתאימים להם, נוכל לכתוב אותה כהוראה בשפת המכונה. נשמור כל הוראה במערך memory.
לבסוף- נייצא כל ערך במערך memory לקובץ memin.txt.

כעת נסביר בנפרד על כל חלקי הקוד:

1. קבועים: נגדיר בהתאם לערכים שהתקבלו בהוראות הפרויקט:

```
#define MAX_LINE_LENGTH 500
#define MAX_LABEL_LENGTH 50
#define MEMORY_SIZE 512
```

2. מבני נתונים:
   a. Label_struct: מבנה נתונים לLabels בקוד. מכיל שדה label_name המקבל את שם הLabel ושדה label_location המקבל את מיקום הLabel בזיכרון:

```
typedef struct {
      char label_name[MAX_LABEL_LENGTH];
      int location;
} label_struct;
```

   b. Line: מבנה נתונים לשורות בקוד וישמש אותנו בריצות על פני הקוד.

```
typedef char line[MAX_LINE_LENGTH];
```

c. Line_fields: מבנה נתונים לכל שורת הוראה בקוד, מכיל שדות עבור: label, opcode, rs, rd, rt, imm.

```
typedef struct {
	char* label;
	char* opcode;
	char* rd;
	char* rs;
	char* rt;
	char* imm;
} line_fields;
```

3. משתנים גלובליים:

מערך Label-ים שישמש לאחסון משתנים מסוג label_struct שהגדרנו קודם:

```
label_struct labels[MEMORY_SIZE];
```

מונה Label-ים שישמש לספירת כמות הLabel-ים בקוד:

```
int label_counter = 0;
```

מערך memory שמציג את תמונת הזיכרון, וישמש לאחסון ההוראות בשפת מכונה לאחר הריצה השניה של האסמבלר.

```
int memory[MEMORY_SIZE];
```

מונה program_length שישמש לספירת גודל התוכנית:

```
int program_length = 0;
```

4. פונקציות:

a. `int opcode_converter(char* opcode)`

פונקציה המקבלת פקודה מסוימת, כגון: add, sub, sll וכדומה, כפרמטר מסוג *char. הפונקציה מתרגמת את הפקודה למספר המתאים לה לפי הטבלה הנתונה בהוראות הפרויקט.

נדגיש כי פעולה זו שונתה כדי להתאים לפרויקט BASYS 2:

התווספו 3 ההוראות החדשות עם הOPCODE המתאים להן: הוראת reti, הוראת in והוראת out.

b. `int register_convertor(char* register_name)`

פונקציה המקבלת רגיסטר מסוים, כגון: $zero, $imm, $t0 וכדומה, כפרמטר מסוג *char. הפונקציה מתרגמת את הרגיסטר למספר המתאים לו לפי הטבלה הנתונה בהוראות הפרויקט.

c. `line_fields parse_line(char* line)`

פונקציה המקבלת שורה (כפרמטר *char), ויוצרת עבורה משתנה מסוג line_fields. אופן הפעולה של הפונקציה הוא כזה:

- נחתוך את השורה מהערות על ידי חיפוש התו '#', וחיתוך כל התווים שאחריו.
- ניצור משתנה result מסוג line_fields המאותחל להיות Null.
- נחתוך את השורה מרווחים מיותרים (תווים " ", "\n", "\t").
- נחפש מופע בשורה של התו: ':', במידה ונמצא כזה נזהה את השורה ככזו המכילה Label, ונשמור את הLabel בשדה המתאים של result.
- נבדוק אם השורה היא שורת ".word". במידה וכן, נשמור בשדה הopcode של השורה ".word", בשדה הrd את כתובת המילה (address) ובשדה הrt את תוכן המילה (data).

- במידה והשורה אינה word., נחתוך אותה במקומות המתאימים ונשמור את הערכים שהתקבלו בשדות המתאימים של result (opcode, rd, rs, rt, imm).
- לבסוף- נחזיר את result.

d. `int get_line_type(line_fields line)`

פונקציה המקבלת פרמטר מסוג line_fields, ובהתאם לערכים בכל אחד מהשדות שלו קובעת מאיזה סוג השורה:

- במידה והשורה ריקה/מכילה הערה בלבד: סוג 1
- במידה והשורה אינה ריקה אך אינה מכילה opcode: זו שורה המכילה Label בלבד, סוג 2.
- במידה והשורה מכילה שדה Label ושדה Opcode ששונה מ".word": זו שורה מלאה המכילה הוראה וLabel ואינה מסוג word., סוג 3.
- במידה והשורה אינה ריקה ושדה הopcode הינו ".word": זו שורה מסוג word., סוג 4.
- במידה והשורה אינה ריקה, אינה מכילה שדה Label אך מכילה Opcode: זו הוראה רגילה ללא Label, סוג 5.

הפונקציה מחזירה את סוג ההוראה כמספר מסוג int.

e. `void first_pass(FILE* input_file)`

הפונקציה שמבצעת את המעבר הראשון על הקוד. מקבלת את קובץ הקלט לאסמבלר.

נאתחל את המשתנה curr_memory המכיל את הכתובת הנוכחית בזיכרון ל0.

- נרוץ על הקוד כל עוד קיימות בו עוד שורות. כל אחת מהשורות נמיר למשתנה מסוג line_fileds ע"י פונקציית parse_line, ונמצא את הסוג שלה באמצעות פונקציית get_line_type.
- במידה והסוג הוא 2 (רק Label): נוסיף את השם של הLabel והכתובת הנוכחית בזיכרון (ערכו הנוכחי של curr_memory), באמצעות שימוש במבנה הנתונים label_struct, למערך labels, ונוסיף 1 למונה הLabelים (label_counter).
- במידה והסוג הוא 3 (שורה המכילה Label וגם הוראה): נוסיף את השם והכתובת בזיכרון למערך labels, נוסיף 1 לlabel_counter ונקדם את curr_memory ב1.
- אחרת- כל עוד השורה לא ריקה ולא מסוג word. נקדם את curr_memory ב1.

בסוף הרצת הפעולה נקבל את המערך Labels כשהוא מכיל משתנים מסוג label_struct עבור כל אחד מהlabelים בקוד, המכיל את שמם ומיקומם בזיכרון.

f. `void second_pass(FILE* input_file)`

הפונקציה מבצעת את המעבר השני על הקוד. מקבלת את קובץ הקלט לאסמבלר.

נאתחל את המשתנה curr_memory המכיל את הכתובת הנוכחית בזיכרון ל0.

- נרוץ על הקוד כל עוד קיימות בו עוד שורות, וכמו קודם נמיר כל שורה למשתנה מסוג line_fields ונמצא את הסוג שלה.

- במידה והסוג הוא 4 (word.)- נקבע את מיקום הadress במערך הmemory להיות הdata של הפקודה (נבצע המרות מהקסדצימלי לדצימלי במידת הצורך).
- במידה והסוג הוא 3 או 5 (כלומר שורת פקודה אם או בלי label)- בעזרת הopcode, register, immediate converters נבצע המרה מהפקודה הנתונה למספרים דצימליים, לפי הוראות הפרויקט.
נבצע קידוד לפורמט הוראות הSIMP הנתון בעזרת shifting (נפצל למקרים בהם ערך הimmediate חיובי או שלילי).
נקבע את מערך הmemory במיקום הנוכחי להיות ערך השורה המקודדת, ונקדם את curr_memory (המיקום הנוכחי בזיכרון) ב1.
- בסוף הפונקציה- נקבע את אורך התוכנית להיות max_adress: שמבטא את הערך הגבוה מבין השניים: הכתובת הגדולה ביותר שנעשה בה שימוש בפונקציית word., או הכתובת הגדולה ביותר שהגענו אליה בזיכרון בעזרת הפקודות האחרות.

בסוף הרצת הפעולה נקבל את המערך memory כשהוא מכיל את השורות המקודדות לכל פקודה במקום המתאים לה בזיכרון. בסוף- ערך המשתנה program_length יהיה אורך התוכנית.

g. `int immediate_converter(char* imm)`
הפונקציה מקבלת ערך immediate של פקודה מסוימת.
אם הערך הינו מילה- מדובר בLabel, ולכן נרוץ על מערך labels עד אשר נמצא התאמה לlabel המתאים, ונחזיר את המיקום שלו בזיכרון.
אם הערך הינו מספר הקסדצימלי- נחזיר את הערך הדצימלי שלו.
אם הערך הוא מספר- נחזיר אותו כמו שהוא.

h. `int HexToInt2sComp(char* h)`
מקבלת מספר הקסדצימלי, ומבצעת המרה למספר דצימלי בtwo's compliment.
מחזירה int.

i. `int HexCharToInt(char h)`
מקבלת ספרה הקסדצימלית, ומבצעת המרה לספרה דצימלית. מחזירה int.

j. `void write_to_file(FILE* input_file, FILE* output_file)`
הפונקציה מקבלת את קובץ הinput (אחת מתוכניות הבדיקה הכתובות באסמבלי) ואת קובץ הoutput (memin.txt). היא מבצעת את המעבר הראשון והשני על הקוד, וכותבת בקובץ הפלט את השורות המקודדות כמספר הקסדצימלי.

k. `int main(int argc, char* argv[])`
בפונקציית הmain מתבצעת פתיחת הקבצים (במידה ואלו לא נפתחים כראוי מתקבלת הודעת שגיאה), כתיבה לקובץ הפלט וסגירת הקבצים.

הסימולטור

תפקיד הסימולטור הוא לדמות את תהליך הלקיחה - פענוח - וביצוע של פקודות המתקבלות מהאסמבלר.
עבור גרסת הסימולטור לPC: הסימולטור מקבל את הקבוץ memin.txt שמכיל את הפקודות שתורגמו מאסמבלי ע"י האסמבלר ומחזיר את קבצי הפלט הבאים:

memout.txt- containing the memory contents at the end of the run.
regout.txt- containing the registers contents at the end of the run in hex8 format, not including R0, R1
trace.txt- containing a line of text for each operation that was done by the CPU in this format in hex8:
PC INST R0 R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15
cycles.txt- containing the number of clock cycles in the program.

עבור גרסת הסימולטור לריצה על כרטיס BASYS: הסימולטור מקבל את ה-memin כמערך מוכן בתוך הסימולטור. מערך זה מכיל את הפקודות שתורגמו ע"י האסמבלר (המעודכן). בגרסה זו לא מוחזר אף קובץ פלט, ולמעשה את הפעולות אנו רואים על גבי הכרטיס.

בתוכנית נשתמש במבנה command הבנוי בצורה הבאה:

```
typedef struct {
	char inst_line[9];
	int opcode;
	int rd;
	int rs;
	int rt;
	int imm;
} command;
```

בקוד עצמו כל אלמנט מבנה זה יכיל פירוק של שורה מ-memin.txt, ז"א יכיל תיאור של פקודה בודדת עם קוד פקודה מתאים, והרגיסטרים הרלוונטים.

בתחילת התוכנית מוגדרים הקבועים הבאים:

```
#define Main_Memory_Max 512 // 2^9
#define Max_string_length 500
#define Max_Label_Len 50
#define CLKS_IN_SECOND (8000000*10/2)
#define CLKS_IN_MILLISECOND (CLKS_IN_SECOND / 1000)
#define SECONDS_IN_DAY (24*3600)
```

והמשתנים הגלובליים הבאים (עם תיאור קצר):

```
char memin_arr[Main_Memory_Max + 1][9]; // array to save memin
lines/information.
int registers_arr[16]; //registers array
static int memin_arr_size; //a constent to save memin actual size
static int pc = 0; //an int variable to save pc position at every clock
cycle.
static int instructions_count = 0;  //an int variable to save the number
of instructions the progrem as made.
char IOregister[15][9] =
{"00000000","00000000","00000000","00000000","00000000","00000000","000000
00","00000000","00000000",
"00000007","00000000","00000000","00000000","00000000","00000000" };
//array to represent registers.
int fib_size = 33; //fib memin array size.
```

```
char fib[Main_Memory_Max + 1][9] = {…}; //fib memin- initialized with
correct values.
int timer_size = 99; //timer memin array size.
char timer[Main_Memory_Max + 1][9] = {…}; //timer memin- initialized with
correct values.

char memin_arr[Main_Memory_Max + 1][9]; //the main memin array- would be
initialized to fib memin/timer memin, in order to the switches state.
int pause_btnl = 0; //variable for pause condition (in order to BTNL
press).
int single = 0; //variable for "single step" condition (in order to BTNR
press).

int reg = 0; //variable for the registers that would be shown on the first
row in the LCD screen
int curr_memory = 0; //variable for the memory places that would be shown
on the first row in the LCD screen
int ready_to_irq = 1; //if equals to 1- we are currently not in an
interrupt, so we are ready to get an interrupt.
int fib_flag = 0; //equals to 1 if fib is running
int reti_count = 0;//a counter for every time we see a reti insrtuction

//variable for every bottom press
int btnl_pressed = 0;
int btnr_pressed = 0;
int btnd_pressed = 0;
int btnu_pressed = 0;
int btnc_pressed = 0;

unsigned int clk32 = 0; // Last 32 bit clock counter sample

char buffer[80]; //Buffer for writing to LCD

//variables for clk_poll() function
unsigned int clk32_prev = 0;
unsigned int clk64_msb = 0;
unsigned long long clk64;
```

כעת נסביר בקצרה על כל פונקציה בקוד (בקוד עצמו קיימת דוקומנטציה מלאה עבור כל פונקציה):

1. `void build_command(char* command_line, command* com)`
   פונקציה שמקבלת מחרוזת המייצגת הוראה שקודדה באסמבלר (מספר הקסדצימלי), ומייצרת משתנה מסוג command בעזרת השמת ערכי הפקודה (ערכי ה inst line, opcode, rd, rs, rt, imm) בשדות המתאימים של משתנה הcommand.

2. `void copy_memin_to_array(FILE* memin)`
   פונקציה שמקבלת את קובץ הקלט לסימולטור- memin.txt, ומעתיקה את כל אחת משורותיו לmemin_array. **(קיימת רק בגרסת הPC של הסימולטור)**

3. `void RegOut(FILE* pregout)`
   פונקציה שמקבלת את קובץ הטקסט regout.txt, וכותבת אליו את תוכן הרגיסטרים בסיום הריצה (ללא הרגיסטרים R0 וR1, כמפורט בהוראות הפרויקט) **(קיימת רק בגרסת הPC של הסימולטור)**

4. `void MemOut(FILE* pmemout)`
   פונקציה שמקבלת את קובץ הטקסט memout.txt, וכותבת אליו את תוכן הזיכרון בסיום הריצה. **(קיימת רק בגרסת הPC של הסימולטור)**

5. `int HexCharToInt(char h)`

פונקציה שמקבלת ספרה הקסדצימלית וממירה אותה לספרה דצימלית.

6. `void Int_to_Hex(int dec_num, char hex_num[9])`
פונקציה שמקבלת מספר דצימלי וממירה אותו למספר הקסצימלי בעל 8 ספרות.

7. `int Hex_to_2sComp(const char* h)`
פונקציה שמקבלת מספר הקסדצימלי ומבצעת המרה למספר דצימלי בtwo's compliment.

8. `void WriteTrace(const command* com, FILE* ptrace)`
פונקציה שמקבלת הוראה כמשתנה מסוג command, וקובץ טקסט trace.txt. הפונקציה כותבת את תוכן הרגיסטרים לקובץ המוצא בפורמט הנתון בהוראות הפרויקט ומפורט בתחילת הדוקומנטציה. **(קיימת רק בגרסת הPC של הסימולטור)**

9. `void clk_counter()`
פונקציה שאינה מקבלת דבר, ואחראית להגדיל את מונה השעון בכל מחזור.

10. `void perform(command* com, FILE* ptrace, FILE* pcycles, FILE* pmemout, FILE* pregout)`
הפונקציה מקבלת פקודה כמשתנה מסוג command ואת כלל קבצי המוצא של הסימולטור שצריך לעדכן (בגרסת הPC). הפונקציה קוראת את הפקודה וע"פ שדה הoppcode שלה מבצעת את אחת מפעולות המעבד, מעדכנת את תוכן הרגיסטרים במערך registers_arr בהתאם ואת הpc.
כאשר מתקבלת פקודת halt- כלומר command בעל ערך 19 בשדה הopcode, הפונקציה מעדכנת את הקבצים כנדרש וסוגרת את התכנית.
בריצה על גבי הכרטיס- הפונקציה לא מקבלת אף קלט מלבד `command* com`. כמו כן- מלבד שינוי תוכן הרגיסטרים והpc, היא אינה מחזירה דבר.

11. `int irq_status_check()`
מטרת הפונקציה היא לבדוק את הסטטוס של כל הפסיקות, לפי התנאי הבא:

irq = (irq0enable & irq0status) | (irq1enable & irq1status) | (irq2enable & irq2status)

הפונקציה מחזירה 1 אם התנאי מתקיים ו-0 אחרת, ובאמצעותה נקבע אם מתקבלת פסיקה.

12. `void led_updater()`
פונקציה שמעדכנת את הלדים לפי מספר הוראות הreti שהתבצעו. כפי שהוגדר בפרויקט- כל פעם שמתבצעת פעולת reti נדלק הלד הבא בתור. **(קיימת רק בגרסת הסימולטור שרצה על הכרטיס)**

13. `static void clk_poll(void)`
פונקציה שמתשאלת את הcounter הפנימי של הכרטיס, ומתאימה את התוצאות להיות 64 ביט במקום 32. **(קיימת רק בגרסת הסימולטור שרצה על הכרטיס)**

14. `void copyAndFillArrays(char source[][9], int sourceSize, char destination[][9], int destinationSize)`
פונקציה שמקבלת את הmemin_array אותו רוצים להריץ (יכול להיות הmemin של FIB או הmemin של הטיימר) ואת גודלו, ומעתיקה אותו למערך הmemin הראשי שאותו תריץ התוכנית (memin_arr). **(קיימת רק בגרסת הסימולטור שרצה על הכרטיס-** שכן בגרסת המחשב מקבלים את הmemin כקובץ.**)**

15. `void check_swtiches(char c_line[9])`
פונקציה שאחראית לבדיקת מצב הSwitch-ים, ועדכון מסך הLCD בהתאם ובהתאם להוראות הפרויקט. מקבלת כקלט את הcommand line- שורת הריצה הנוכחית בקוד הבדיקה. **(קיימת רק בגרסת הסימולטור שרצה על הכרטיס)**

16. `void check_btnu_press()`
פונקציה שבעת ריצתה בודקת את מצב הSwitchים ואת לחיצת הכפתור BTNU, ובמידה ונלחץ והswitchים נמצאים במצבים המתאימים לכך- אז לחיצה על הכפתור מקדמת את מספר הרגסיטר שתוכנו מוצג על המסך/המקום בזיכרון שתוכנו מוצג. **(קיימת רק בגרסת הסימולטור שרצה על הכרטיס)**

כעת נסביר על פונקציית הmain עבור שתי גרסאות הסימולטור.

עבור גרסת הPC: `int main(int argc, char* argv[])`

הפונקציה מקבלת את קובץ הקלט, memin.txt לקריאה ואת ארבעת קבצי הפלט לכתיבה. תחילה, זו בודקת שאכן כלל הקבצים נפתחו כנדרש, אחרת מוחזרת שגיאה והתוכנית מופסקת עם קוד 1.
לאחר מכן, תוכן קובץ הקלט memin.txt מועתק למערך, בעזרת הפונקציה copy_memin_to_array, כך שכל איבר במערך מכיל שורת פעולה על מנת שנוכל לעבוד עליו בלי לפגוע בmemin.txt. לאחר ההעתקה סוגרים את memin.txt.
לאחר שברשותנו מערך הפקודות memin_arr, נבנה עבור כל פקודה אלמנט command מתאים, נתעד בtrace.txt בעזרת פונקציית WriteTrace ונקבצע את הפקודה באמצעות פונקציית perform.
ברגע שמתקבלת פקודת halt נפסיק לקרוא פקודות מהמסמך (המעבד יפסיק לרוץ), נעדכן את קבצע הפלט כנדרש, נסגור אותם ונסיים את ריצת התוכנית.
כמו כן, השינוי של סימולטור זה בניגוד לסימולטור של הפרויקט הקודם מתבטא, בפונקציה זו, בתמיכה בפסיקה 0 וברגיסטרי החומרה: בכל מחזור נעשית בדיקה האם IOreg[12]=IOreg[13], ובמידה שכן- אנחנו נמצאים בפסיקה 0, לכן נדליק את הסטטוס שלה, נבדוק אם אפשר להכנס לפסיקה, ואם כן נשמור את הpc הנוכחי ונקפוץ לשגרת הפסיקה. לבסוף נאפס שוב את IOreg12.

עבור גרסת הכרטיס: `int main()`

נסביר את אופן הפעולה בחלקים.
תחילה- מתבצע איתחול למודולים של הIO: לדים, כפתורים, Switchים וכדומה. בנוסף מתבצעת בדיקה למצב Switch 7- במידה והוא מורם (מצב "1") נריץ את הטיימר. אחרת (מצב "0") נריץ את FIB.
נבצע איתחול למשתנים נוספים: seconds (מספר השניות הנוכחי שיוצג על מסך הLCD) יאותחל ל0, frequency (מספר המחזורים המבוקש לשניה) יאותחל ל1024.
המשתנה frequency_test (מטרתו להיות 1 כשאנחנו מתחילים מחזור שעון חדש ו0 אחרת) יאותחל ל1.
נאתחל את IOregister[12] (שמהווה את מחזור השעון הנוכחי ל0).
רגע לפני הכניסה ללולאה while(1), נבצע תשאול לcounter הכרטיס באמצעות הפונקציה clk_poll(), ונקבל ערך clk64 עדכני לרגע זה. לפיו- נעדכן את last_frequncy וcurrent_frequency. אלו ישמשו אותנו בהמשך לבדיקה אם מחזור שעון חדש התחיל ולמציאת המיקום היחסי של המחזור שלנו ב1024 מחזורי השעון בשניה (current_frequency).

מעתה והלאה כל הפעולות אותן נפרט מתבצעות בתוך לולאת הwhile(1)- זו רצה כל עוד לא לוחצים על כפתור הreset בכרטיס.
בתחילת כל איטרציה של הלולאה מתבצע תשאול לcounter הכרטיס באמצעות הפונקציה clk_poll(), וקבלת ערך השניות (seconds) וlast frequency-.
המעקב אחר השניות נעשה בעזרת שימוש בידע על קצב מחזור שעון בודד של השעון הפנימי בכרטיס, בעזרת שימוש בClks_in_second- ובאופן דומה כתיבת קצב המעבד המסומלץ על הכרטיס ע"י חלוקה של כל שנייה למס' מחזורים מוגדר של1024-. על מנת לוודא שבאמת אנו מריצים מס' פקודות לשנייה כרצוי כל ריצה על לולאת $while(1)$ של ה-$main$ בודקים האם ערך שעון הנוכחי שנשמר ב-$last\_frequency$, התקדם $Clks\_in\_second/1024$, ז"א האם עברנו מחזור שעון בקצב המסומלץ בו אנו מעוניינים, אם כן מועלה 'דגל' (frequency_test) המסמן לבצע פקודה בודדת, ומעדכנים את משתנה הערך $last\_frequency$ העוקב אחר השעון הנוכחי בהתאמה וכך חוזר חלילה לקבלת קצב של 1024 פקודות לשנייה.

כעת מתבצעת חלוקה לשני תנאים, שלשניהם נכנס רק אם התחיל מחזור שעון חדש, לא נמצאים בpause או שנמצאים בsingle_step (סמן לכך רוצים לבצע פקודה בודדת).
התנאי הראשון מתקיים כאשר עוברת שניה (IOreg12==IOreg13), ובו נעשה איפוס לקריאות השעון בIOreg12, והדלקת הסטטוס לפסיקה 0. במידה ואנחנו פנויים לפסיקה (ready_to_irq==1) ומתקיימים שאר התנאים לפסיקה (הנבדקים בפונקציה (irq_status_check), אז שומרים את הPC הנוכחי בIOreg7 וקופצים לשגרת הטיפול בפסיקה השמורה בIOreg6. לבסוף נקרא את ההוראה הנוכחית ונבצע אותה.

אחרת- במידה ולא עברה שניה, נכנס לתנאי השני- שם נבדוק את פסיקות 1 ו-2 באותה הדרך שפירטנו קודם לפסיקה 0 (הדלקת הסטטוס, בדיקת המוכנות והתנאים לפסיקה, שמירת PC וקפיצה לשגרת הטיפול), נקרא את ההוראה הנוכחית ונבצע אותה.

בשני התנאים נבדוק מצב בו BTNL נלחץ ונדליק את pause_btnl בהתאם. במידה וpause_btnl דלוק- נבדוק אם מתרחשת לחיצה על BTNR שמובילה אותנו למצב single step, ונבצע את הפעולות בהתאם- פעולה אחת בכל פעם.
בנוסף נבדוק את מצב הswitchים ונשתמש בפעולה `check_btnu_press()` בכדי לקדם את הרגיסטרים/התא בזיכרון שיוצג בעת עדכון המסך.
לבסוף- נבצע קידום לIOreg12, זאת כל עוד timer enable = 1, או שאנחנו נמצאים בתוכנית הFIB (שם אין שימוש ברגיסטר IOreg11 ולכן היא תמיד רצה).

תוכניות הבדיקה

Timer.asm

מטרת התוכנית להריץ סטופר על המעבד שמסומלץ על הכרטיס, כך שהזמן יודפס על המסך- SSD בבסיס דצימלי בצורה: $MM:SS$ עד ל-$59:59$ ולאחר מכן יתאפס ויתחיל מחדש את הספירה. בנוסף לכך לסטופר יהיו מספר פונקציות, כאשר לוחצים על הכפתור $BTNC$ שעל הכרטיס ייעצר, כך שפקודות והסימולציה ממשיכה לרוץ ברקע, וימשיך בעת קבלת לחיצה נוספת על הכפתור $BTNC$.
בנוסף כאשר נלחץ הכפתור $BTND$ שעל הכרטיס הסטופר יתאפס, כאשר במקרה של כניסה למצב עצירה (לחיצה על $BTNC$) ולאחר מכן איפוס הסטופר ($BTND$) הסטופר לא התאפס ולא יתחיל לרוץ מחדש עד לחיצה נוספת על $BTNC$ לסימון יציאה ממצב העצירה.

כעת נפרט בקצרה על אופן פעולת התוכנית: בעת אתחול התוכנית פותחים וטוענים את המשתנים הרלוונטים למחסנית, ומאפסים/מאתחלים לערכים הרצויים את כלל המשתנים שבהם נשתמש בתוכנית:
טוענים את כתובת הזיכרון של פונקציית $HANDLER$- פונקציה שאחראית על ניתוב במהלך הריצה לפונקציות הנכונות של הטיימר (איפוס, איתחול מחדש, ועדכון זמן) לרגיסטר 6
טוענית את רגיסטר 11, רגיסטר המתאר $timerenable$ אשר מסמן האם ספירת השעון עובדת או לא (קובע את התקדמות רגיסטר 12 החומרתי העוקב אחר מחזורי השעון), ל-1.
לאחר סיום האתחול נכנס ללולאה אינסופית, $FOREVER\_LOOP$, אשר שימושה היא למנוע מצב של התקדמות לא רצויה של הטיימר בעיקר בהתחלה עד שמתקיימת פסיקת ה-0 הראשונה שכן שנייה עוברת כל 1024~ מחזורי שעון מסומלצים ועל כן אם לא נשתמש בה ירוצו פקודות רבות לא רצויות שיגרמו להתנהגות לא רצויה של הטיימר. בנוסף- במהלך ריצתה אנו שולחים לרגיסטר 10 (הרגיסטר ששומר את הערך שמודפס למסך SSD) את ערך הטיימר הנוכחי.
כל 1024 מחזורי שעון הסימולטור מבצע פסיקה 0, כפי שמוגדרת, מה שגורם לקפיצה פעם בשנייה לפונקציית $HANDLER$, משם נקפוץ לאחד משלושה פונקציות:

במקרה שלא התקבל אף קלט מ-$BTNC/BTND$ נכנס ל-$STOPER1$. מטרת חלק זה- הינה לקדם את הזמן בשנייה ולבדוק אם הגענו לשנייה 10 בספרה הראשונה של הסטופר. אם לא- יקודם הערך הנוכחי ששמור ב-$v0$$, ששומר את הערך שמודפס שנשלח בלולאה האינסופית למסך, בשניה.
אחרת- נכנס ל-$STOPER2$ ושם באופן דומה נבדוק האם הגענו לשנייה 60 בספרה השנייה של הסטופר. אם לא- יקודם את הערך ב-$v0$$ ב-7 (זאת על מנת לקדם ב-1 את הספרה השנייה על מסך ה-SSD, שכן ההדפסה והמסך עובדים בבסיס הקסדצימלי והגענו מסטופר אחר ממצב $YY:X9$ ועל כן ע"מ לקבל קידום באחד דצימלי נרצה להוסיף 7 לערך השעון הנוכחי).
באופן דומה, אם לא מתקדמים לספרה השלישית בשעון (המייצגת את הספרה הראשונה של הדקות) מתקיימות בדיקות דומות ל-$STOPER1$ כאשר ההבדל שהקבוע שנוסף ע"מ לקדם את השעון ב-1 יהיה 167 (שכן כעת אנו במיקום ספרה שלישי של ערך הקסדסימלי ומעוניינים להדפיס בתצורה דצימלית).
כנ"ל מתקיים עבור $STOPER4$ המייצג את ערך עשרות הדקות, הספרה הרביעית במסך ה-SSD כאשר הפעם הקבוע 'קפיצה' שלנו מ-$X9:YZ$ ל- $(X+1)0:YZ$ תהיה 1703.
בסוף כל סטופר נתקל בפקודות $reti$ אשר תחזיר אותנו חזרה על הלולאה האינסופית עדכון רגיסטר המסך והמתנה לפסיקת 0 הבאה שתחזיר אותנו אל ה-$HANDLER$.

במקרה של לחיצה על $BTNC$ נעבור לפונקציה $PAUSE$ שתפקידה להכניס את הטיימר למצב עצירה, הפונקציה מורידה את הדגל שהסימולטור הרים בכך ששינה את- $irqstatus1$ ל-1 ל-0, ומשנה את ערך $timerenable$. לבסוף מתבצעת קריאה ל-$reti$ שמחזירה אותנו ללולאה האינסופית.
כך למעשה אם אנחנו במצב ריצה ולוחצים על מנת להיכנס למצב עצירה, הפונקציה תשנה את ה-$timerenable$ ל- 0 מ-1 מה שיגרום לכך שלא תתקיים התקדמות של רגיסטר 12 החומרתי. נזכיר שרגיסטר זה עוקב אחר התקדמות מחזורי השעון ועל כן הסימולטור לא ייכנס לפסיקה 0, זאת בעוד הסימולטור ממשיך לרוץ ברקע. בעת לחיצה כאשר ה-timerenable שווה ל-0, הסימולטור יקפיץ אותנו ל-HANDLER, ויעלה את דגל $irqstatus1$ ל-1 מה שיגרום לשינוי $timerenable$ חזרה ל1,

ובמקביל הורדת הדגל חזרה ל-0, ועל כן חזרת קידום רגיסטר 12 וחזרת ביצוע פסיקות 0 וקידום השעון. לבסוף כמובן תתבצע פקודת $reti$.

במקרה של לחיצה על $BTND$, הסימולטור יקפיץ ל-1 את $irqstatus2$ ועל כן בעת מעבר על פונקציית ה-$HANDLER$ נקפוץ לפונקצייה $INITIALIZE1$ בה נאפס את כלל המשתנים ששומרים את הערכים של ספרות השעון, נאפס את $\$v0$ ששומר את ערך הטיימר הנוכחי, ונאפס את רגיסטר המסך, רגיסטר 10 החומרתי ולבסוף נקרא לפקודת $reti$.

התוכנית לא תוכננה כך שתעצור, על כן לא בוצעה תהליך יציאה: $HALT$, וכל התהליכים הנכללים בזאת (סגירת מחסנית, איפוס רגיסטרים בעת הצורך וכיוצא בזה).