הפקולטה להנדסה ע"ש איבי ואלדר פליישמן
אוניברסיטת תל אביב

בי"ס להנדסת חשמל

Sequencer Debugger – פרויקט

תכנית עבודה

שם הפרויקט: Sequencer Debugger

מבצע:

דור גולפייז ת.ז 316060805

מיקום ביצוע הפרויקט: Vayyar Imaging Ltd.

לשימוש המנחה:

הנני מאשרת את תכנית העבודה המצורפת

שם: מיכל כרמלי

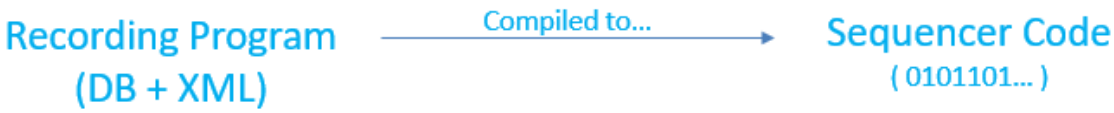
חתימה: מיכל

1. תקציר

בוויאר, אנו מייצרים רדארים למגוון רחב של שימושים – InCar, Breathing monitor, Fall detection, Tracking, etc.

לכל אחד מהשימושים האלו יש משהו אחד במשותף – ה-Sequencer. לכל תוכנית הקלטה, אנו מעבירים שני קבצים כקלט – XML+ DB . בשניהם נעביר את כל הידע הנחוץ לתוכנית הקלטה זו – רוחב פס, טווח תדרים, באילו אנטנות להשתמש כ-transmit/receive, מה יהיה ההגבר לכל פורט שכזה, ורבים נוספים.

מקלט זה, אנו מייצרים קוד Sequencer , שהחברה כתבה ברמתה (עד רמת הopcode).



ה Sequencer הוא קונטרולר שתפקידו לשלוט בפלואו הקלטה הבסיסי ביותר – סריקה על טווח של תדרים (Chirp/Steps), שידור וקליטה, הפעלת/כיבוי מודולים מסויימים ברדאר, כתיבת הפאזורים שהתקבלו ב-DSP וכו'.

זאת במטרה לשחרר את המעבד החיצוני (Host) מביצוע פעולות בזמן אמת, שכן בעבודה עם מעבד חיצוני לא ניתן להסתמך על כך שפעולות ייעשו בדיוק של עד כדי מחזור שעון - שכן מתבצעות אופטימיזציות נוספות שעלולות לשבש את ההליך ולהוביל לקבלת פאזורים לא נכונים ולעוד מגוון רחב של תקלות (לכן גם לא עובדים עם מעבד ARM, והחליטו לכתוב ולממש קומפיילר לסיקוונסר)

כמה דוגמאות לצורכי המחשה –

Opcode name	Copy – opcode = 0x1								
Bits: interpretation	[40] Inc Src base address	[39:25] Src Addr	[24] Add base to Src Addr (en)	[23] Src Type (config/Param)	[22] Inc Dest base address	[21:7] Dest Addr	[6] Add base to Dest Addr	[5:4] DestType	[3:0] Opcode
Description	Copies data from source to destination. If INC is set, then also increments base addr. If 'add base to src/dest addr' is set then actual address is dest/source addr + dest/source base.								

Figure 1 : Sequencer Opcode Example - Copy

```
00006D60  10 47 00 00 00 00 00 00 F4 C6 00 00 20 00 00 00
00006D70  C4 D8 00 00 20 00 00 00 24 D9 00 00 20 00 00 00
00006D80  04 00 00 00 40 00 00 00 00 60 88 00 00 00 00 00
00006D90  81 01 00 00 80 00 00 00 93 00 00 00 20 00 00 00
```

Figure2 : Sequencer Code (Hexadecimal)

לקוד הבינארי המתקבל כפלט, כבר בזמן הקומפילציה, אנו מוציאים גרסה נוספת – גרסת טקסט קריאה יותר, שבעזרתה ניתן להבין קצת טוב יותר מה מבצע ה-Sequencer. דוגמא –

```
Addr(shorts)   Opcode     Description
MainSweep0
000d9c  84 1e 00 00 20 00  Branch  0x1e8, [CALL] < Vayyar::Centipede::CentipedePowerManagementCommand::OnSweepStartF
000d9d  c4 74 00 00 20 00  Branch  0x74c, [CALL] < Vayyar::Centipede::CentipedeTuneFrequencyCommand::TuneFirstFreque
000d9e  64 75 00 00 20 00  Branch  0x756, [CALL] < Vayyar::Centipede::CentipedeDacAdcModule::TurnOnSequence(void) >
000d9f  90 0a 00 6c 04 00  Write   [1:0x0015] = 8d8
000da0  00 08 98 57 00 00  Write   [0:0x3010] = af
000da1  e4 0b 00 00 20 00  Branch  0xbe, [CALL] < Vayyar::SequencerMath::AddShort(const class IILParam &,const class
000da2  00 09 18 00 40 00  Write   [0:0x3012] = 8000
000da3  43 00 00 00 20 00  Wait    4, Tc

class passable_ptr<class ILFunction> __cdecl Vayyar::Centipede::CentipedeDacAdcModule::TurnOnSequence(void)
000756  00 01 08 04 00 00  Write   [0:0x1002] = 8
000757  43 06 00 00 20 00  Wait    100, Tc
000758  80 00 10 22 00 00  Write   [0:0x2001] = 44
000759  03 20 4e 00 20 00  Wait    320000, Tc
00075a  00 01 88 07 00 00  Write   [0:0x1002] = f
00075b  43 06 00 00 20 00  Wait    100, Tc
00075c  80 00 90 7f 00 00  Write   [0:0x2001] = ff
00075d  43 06 00 00 20 00  Wait    100, Tc
00075e  04 00 00 00 40 00  Branch  0x0, [RET]
```

Register

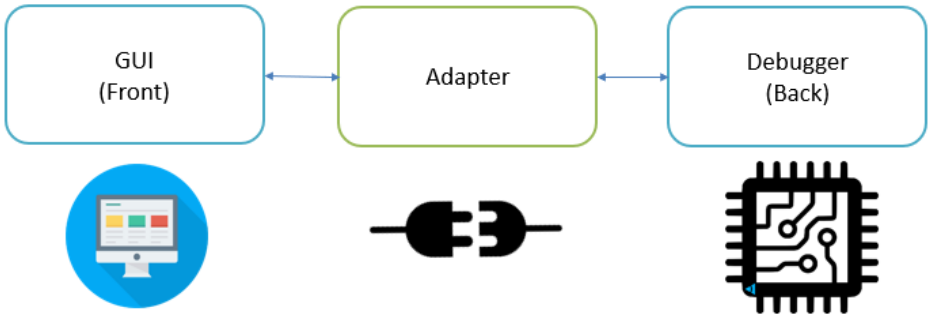
```
"name" : "RX_ADC_N_FIFO_CNTL",
"addr" : 0x1002,
"access" : "read-write",
"description" : "",
"busID" : -1,
"memType" : "Config",
"offset" : 2,
"size" : 2,
"fields" : [
```

Figure3 : Sequencer Code (txt)

כאן נכנס לתפקיד ה-Sequencer Debugger –

כלי שיקשר בין קובץ הטקסט לקובץ הבינארי, ובכך יאפשר להוסיף נקודות עצירה על קטע הטקסט, מה שיכניס את הסיקוונסר ל-Conditional Wait, ובכך יאפשר לראות את מצב הרגיסטרים, לשנות ערכים מסויימים, להציג Call Stack, וכו'.

בעזרת כלי זה, ניתן יהיה לוודא את נכונות הפאזורים, לנסות ולהוכיח היתכנות של POCs, לדבג תקלות בערכי רגיסטרים ב-Dumps ועוד.



2. מוטיבציה

עד היום, הטקסט המצורף למעלה המכיל את קוד הסיקוונסר שנצרב למעבד אך בצורה קריאה וידידותית, שומש לצורכי קריאה ווידוא. לא ניתן להריצו, ואין דבר הקושר בינו לבין הקובץ הבינארי הנצרב ל- Program Memory של ה-Sequencer.

ואם ברצוננו לוודא רצף אופקודים שנכתב עבור הסיקוונסר – יכלנו להריצו על ה-Sequencer Simulator-

כלי המאפשר לסמלץ קוד סיקוונסר (בעזרת API שבו ה-HOST ניגש ישירות למודולים ולרגיסטרים – מן הסתם קבועי זמן איטיים ביותר!)

לצורך השוואה – לאחר בדיקה שביצעתי, הפרשי הזמנים הם קיצוניים ולא פרקטיים – קוד שייקח לסימולטור 4 דקות לסמלץ, יסיים לרוץ על הסיקוונסר ב- 100 מילישניות.

הפרשי הזמנים האלו גורמים לכך שהסימולטור מאבד את המהות שלשמע החברה עברה לעבוד עם סיקוונסר – הוא לא לוקח בחשבון את העובדה שהסיקוונסר עובד בקבועי זמן קצרים בהרבה, וכאשר מדובר ברכיבים חומרתיים שיש לקחת בחשבון את הזמן שלוקח להם להתייצב (PLLs, Voltage measurements, etc) – מבינים שמדובר בכלי שאינו מהווה פיתרון לבעיה – כיצד ניתן לוודא את תקינותו של קוד הסיקוונסר.

לשם כך עלה הרעיון לבנות Sequencer Debugger – כלי דיבאגינג הקושר בין קובץ הטקסט לרצף האופקודים, נותן את היכולת לדבג קוד הרץ ישירות על הצ'יפ, ומאפשר להשתמש בכל כלי שדיבאגרים אחרים בשוק נותנים - מנקודות עצירה ועד החלפת ערכי רגיסטרים תוך כדי ריצת המעבד.

3. תכולת העבודה

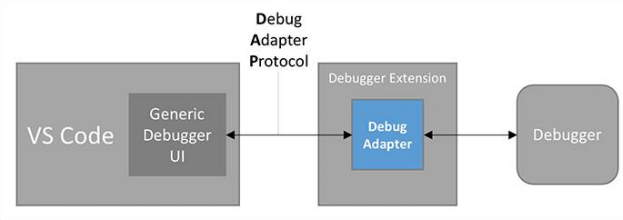
כפי שניתן לראות בדיאגרמה שבתקציר הפרוייקט, ניתן לחלק את הפרוייקט ל-3 חלקים –

1. **GUI** – הסביבה שבה ניתן יהיה לעבוד עם הדיבאגר –סביבה ידידותית למשתמש שנותנת גישה לארגז הכלים שדיבאגרים מספקים - Breakpoints, Watchers, Call stack, local/global variables, etc.
2. **Debugger** – הכלי שיפותח, שמטרתו לממש את כל האבסטרקציות המפורטות לעיל – לדוגמא עבור מימוש של נקודת עצירה - צריבה של Breakpoint Loop לקוד סיקוונסר, החלפת אופקוד ספציפי שבו נרצה לעצור ל- Branch שיעבור ל-Breakpoint Loop, והמתנה לפקודות נוספות בזמן שה- Sequencer ממתין.
3. **Adapter** – תפקידו לחבר בין ה-GUI, למשל אם נחזור לדוגמא המדוברת הוספה של נקודת עצירה על שורה מסויימת בקובץ טקסט, לפקודה שתגיע ל-Debugger בפורמט שהוא ידע להבין, לממש ולצרוב פיזית על ה-Sequencer.

עבור מימוש ה-Adapter + GUI, ישנן 2 אפשרויות מרכזיות העומדות כרגע לנגד עיניי-

- יצירת GUI פייתוני (בעזרת הספריות PyQT / Tkinter), המציג את מסמך הטקסט המתאר את קוד הסיקוונסר כמתואר בתקציר, ומאפשר הצגה נוחה וממשק שימושי ונוח עבור המשתמשים הכולל את כל הכלים שפורטו לעיל.
- התחברות ל-GUI קיים, בעזרת שימוש בפרוטוקולים אבסטרקטיים הקיימים ברשת עבור IDEs / Text editors הקיימים בשוק. לדוגמא- **פרוטוקול DAP.**

DAP – Debug Adapter Protocol



זוהי אבסטרקציה שפותחה ככלי לחבר בין Debugger כלשהו לשפה כלשהי, לבין UI קיים, מה שיהפוך את השימוש בדיבאגר לאינטואיטיבי מאוד עבור משתמשים מהחברה וייקל על התקנת הכלי על סביבות שונות (יקטין את כמות ה-Dependencies).

פרוטוקול זה כבר מומש עבור מספר רב של עורכי טקסט \ IDEs

Cloud Studio eclipse THEIA

מה שאמנם יקשה על פיתוח ה-Adapter, שכן כעת ה-GUI לא מומש על ידי, אך יחסוך הרבה זמן פיתוח שהוא לאו דווקא נחוץ עבור הכלי, ויאפשר לתת את הדגש בחלקים אחרים שכרגע הינם אופציונליים – כמו לאפשר לפתח את קוד הסקוונסר לכדי שפת תכנות לכל דבר – לאפשר להוסיף קטעי קוד ישירות על קובץ הטקסט וממנו ליצור ולצרוב שורת opcode מתאימה במקום המתאים, להוסיף Syntax Highlighting, ולהפוך את קובץ הטקסט שכרגע לא משמש לכלום לשפת תכנות לכל דבר, עם קומפיילר, Code Completion, ועוד.

4. תוצרי הפרויקט

- **סביבת Debugging עבור קוד Sequencer-**

סביבה (בין אם תמומש בעזרת DAP ובין אם בעזרת GUI + Adapter), המאפשרת לעובדי החברה השונים לדבג בזמן ריצה קוד הרץ על המעבד –
מה שייתן לצוותים השונים (System, RF, POC team, etc) את היכולת לעצור באמצע קוד הסיקוונסר, לוודא את נעילת ה-PLL, את מצב הרגיסטרים, מדידות מתחים וטמפ' באיזורים השונים בצ'יפ בחלקים שונים של האלגוריתמיקה, שינוי פרמטרי ה-NCO
(למשל Start/end frequency -> change chirp bandwidth בזמן ריצה), וכו'
וכל זאת בעזרת ממשק נוח ואינטואיטיבי למשתמש.

דרישות מערכת זו –

1. אפשרות לעצור ב-Breakpoint בכל שורה בקוד, להמשיך ריצה עד ה-Breakpoint הבא, Step in/over/out, וכו'
2. הצגת משתנים לוקאליים של כל פונקציה כ-Local Variables – כולל האפשרות לשנות בזמן העצירה את ערכם של רגיסטרים המשומשים בפונקציה הנוכחית ולכתוב את הערך החדש ישירות ל-DRAM של המעבד
3. הצגת Call Stack – מכיוון שקוד סיקוונסר נצרב וקומפל על ידי החברה, נוצר מצב שבו לצורך מתן דוגמא תהליכים כמו Tune Frequency נצרב לקוד המעבד בכל פעם מחדש, עבור כל אחד מהתדרים (יש issue פתוח שמטרתו לכווץ את הקוד בכך שכל פונקציה שכזו תיכתב פעם אחת ותקרא את הארגומנטים שלה ממקום קבוע בקוד – אך זהו תהליך שייקח כמה שנים עד שיקרה וכרגע לא רלוונטי ויש למצוא פיתרון לכך – בדיוק כאן נכנס הדיבאגר)
לכן, במידה והוספתי BreakPoint בתוך אחת מהמתודות של Tune Frequency, ברצוני לדעת באיזה תדר מדובר!
לשם כך ברצוני לממש Call Stack, המאפשר להציג את המחסנית בצורה ידידותית, לקפוץ מקריאה לקריאה, ובכך לאפשר למשתמש להבין מהו התדר המכוייל כרגע וכמה תדרים נותרו לכיול – וזו רק דוגמא אחת לצורך הברור במימוש של Call Stack.

תוצרים אפשריים נוספים –

- Code Navigation – אפשרות לקפוץ מפקודת Branch מסויימת לפונקציה אליה היא קופצת בקטע הטקסט.
- Syntax Highlighting – לשפר את קריאות הקוד ובכך גם את חווית המשתמש.
- Code Writing – הדיבאגר מחייב לקשור את הקוד הבינארי המקומפל לקטע הטקסט – אך לאו דווקא להיפך! ההצעה היא לאפשר כתיבה של קוד סיקוונסר (ללא כתיבת האופקוד כמובן) ישירות לקובץ הטקסט, ולאחר הפעלת סקריפט נוסף הוספת השורה הנ"ל הן לקוד הבינארי והן כשורה מלאה המכילה את הOpcode לקובץ הטקסט. רעיון זה כולל בתוכו המרה של טקסט ל-Opcode הממומש אותו, מטרה שתהיה לא קלה למימוש, אך במידה ואספיק לעמוד בלוחות זמנים צפופים יותר מאלו המתוכננים זו המטרה הבאה. מטרה זו מכילה בתוכה כלים נוספים שיוכלו להתווסף לדיבאגר, למשל -
Smart Completions – השלמה חכמה של קוד סיקוונסר בעת כתיבה.

אבן דרך	פירוט	תאריך יעד	הערות
בחירת סביבת עבודה	הכרעה בין פיתוח GUI לבין פיתוח בעזרת DAP – חקר מעמיק באינטרנט על היתרונות והחסרונות של כל אחד, ניסיון להתחיל לעבוד עם DAP ולראות האם הוא עונה על הדרישות, ותחילת עבודה.	21.11.22	
התחלת פיתוח GUI + Adapter	כתיבת ופיתוח הכלי, היאפשר ביצוע פעולות של כלי דיבאגינג כמו הוספת נקודת עצירה ועריכת ערך משתנה – והעברת פעולות אלו כפקודה אבסטרקטית לדיבאגר	11.12.22	
מימוש Adapter אבסטרקטי	מימוש סופי של ה-Adapter, המכיל את ארגז הכלים הרצוי, והעברת כל פעולה שהתבצעה בו פקודה לדיבאגר, כך שהאבסטרקציה תושלם והדיבאגר יוכל להתחבר ל-GUI במאמץ מינימלי, וכל שעליו לעשות הוא לממש את האבסטרקציות הנ"ל על המעבד	1.1.23	
יצירת Breakpoint Loop	כתיבת קוד סיקוונסר (מרצף אופקודים), שמטרתו להכניס את הסיקוונסר ל-Conditional Wait, והרצת ניסיונות של Branch במקומות שונים בקוד הקופצים אליו.	16.1.23	
הצגת התוכנית למנחה	הצגת תוכנית הוריפיקציה לטובת אישור, תוספות ותיקונים לפני הצגת מצגת האמצע.	17.1.23	
כתיבת מצגת האמצע	הצגת מצגת המתארת את הרעיון, את הצורך ואת אופן המימוש שלו.	20.1.23	
הצגת מצגת אמצע		22.1.23	
פיתוח Debugger	מימוש האבסטרקציות שפורטו לעיל – הכנסת Breakpoint, כתיבה לרגיסטרים, שמירת Call Stack אך כל זה ישירות על המעבד בעזרת צריבת אופקודים בזמן ריצה ישירות ל-IRAM.	19.3.23	
חיבור ה-Debugger ל-GUI	מימוש שליחה וקבלה של פקודות בין ה-GUI, לבין הדיבאגר, מה שיאפשר הן את התקשורת בין השניים ואת ביצוע הפעולות הנעשות ב- GUI ישירות על המכשיר, והן את החלפת ה-GUI בכלי אחר במידת הצורך, בזכות נטרול התלות אחד בשני	16.4.23	
כתיבת טסטים עבור תרחישים שונים והרצתם	כתיבת תרחישי קלט שונים (מכשירים שונים, קוד סיקוונסר שונה וכו') ובדיקתם בדיבאגר.	30.4.23	

ניתוח התוצאות	השוואה וניתוח התוצאות לצד המטרות שהצגתי בתחילת העבודה על הפרויקט	4.5.23	
הרחבת הכלי	פיתוח והרחבת ה-Debugger בעזרת כלל ההצעות שהועלו במסמך זה- Syntax Highlighting, code navigation, etc.	20.5.23	
הגשת פוסטר וסיום העבודה בפרויקט		28.5.23	
המשך - הרחבת הכלי	פיתוח והרחבת ה-Debugger בעזרת כלל ההצעות שהועלו במסמך זה- Syntax Highlighting, code navigation, etc.	20.6.23	
הגשת ספר הפרויקט + מצגת הסיום		29.6.23	