סיכום קורס

"מעבדה במיקרו-מחשבים"

“If one has a one which one needs to AND with another one’s one,

Than one ands those two ones, and what one gets is a one”

(סיכום בלי יותר מדי One'ים)

דן אילני

פברואר 06, 2010

פרק 1

ה- ADuC841

- ה- ADuC841 הוא בתכליתו מיקרו-מעבד 8052 עם מקבץ של פריפריות נוספות שהוספו.
- ה- 8052, אם אני לא טועה, זה 8051 עם מקבץ של פריפריות שהוספו.
- כך שלמעשה ל- ADuC841 יש את כל מה שיש ל- 8051, 8052, ועוד דברים נוספים חדשים.

לכל ה- 8052 יש:

- 256 בייטים של זיכרון מידע פנימי.
- אוגרים לפעולות מיוחדות (Special Function Registers= SFRs).
  נמצאים בתוך 128 הבייטים העליונים של זיכרון המידע הפנימי.
- 3 טיימרים / מונים.
- מספר פורטים של I/O.
- UART אחד. (UART = Universal Asynchronous Receiver Transmitter), שמסוגל לתקשר תוך שימוש באחד מהפרוטוקולים שבהם משתמש המחשב.

זיכרון ה- 8052 מאורגן בחמישה אזורים:

- אזור הקוד (חיצוני או פנימי)
- אזור המידע הפנימי.
- אזור נגיש ביט-ביט.
- אזור המידע החיצוני.
- אזור האוגרים לפעולות מיוחדות.

כל אחד מארבעת סוגי הזיכרון הראשונים נגיש ע"י פקודות שונות, ומבוסס על העובדה שהמעבד ייגש לזיכרון לביצוע הפקודה בצורה שונה. (גישה לזיכרון החיצוני, לדוגמא, כרוכה בשינוי המתחים על חלק מהפינים היוצאים מהמעבד. גישה לזיכרון הפנימי לא כרוכה בשינויים כאלו).

סוג הזיכרון החמישי ממופה לתוך תת-חלק באזור המידע הפנימי – סוג הזיכרון השני.

ל- ADuC841 יש כמה פריפריות שאין ל- 8052:

- שמונה ערוצי A/D ושני ערוצי D/A.
- 2048 בייטים (2 קילו-בייט) של זיכרון חיצוני, על השבב של המעבד.
- טיימר נוסף שיכולים להשתמש בו כדי לקבוע את ה- BAUD Rate של ה- UART.
- 62 קילו-בייט של זיכרון תוכנית FLASH/EE פנימי שיכולים לתכנת אותו ע"י הפורט הסיריאלי של ה- ADuC841.
- Watchdog טיימר.
- 4 קילו-בייט של זיכרון FLASH/EE.

בקורס הזה לא עובדים עם ADuC841 פשוט, אלא עם ערכה שלו שנותנת לנו מספר פונקציות שאין לשבב עצמו.
למשל, לכבל שמסופק עם הערכה יש Level Shifter, שלוקח את המוצא של ה UART על השבב ומסית אותו מהרמות מתח של השבב שהן 0 וולט – 5 וולט, לרמות המתח של RS232 שהן (12-) וולט – 12 וולט.

פרק 2

המיקרו-מעבד

2.1 סקירה כללית

- כמו כל אלו החברים במשפחת ה- 8051, גם ליבת ה- ADuC841 יכולה להיחשב ככזו שמורכבת מכמה אזורים של זיכרון ומגוון יחידות עיבוד. תסתכלו על ציור 2.1 בשביל איור סכמטי פשוט של ה- ADuC841.

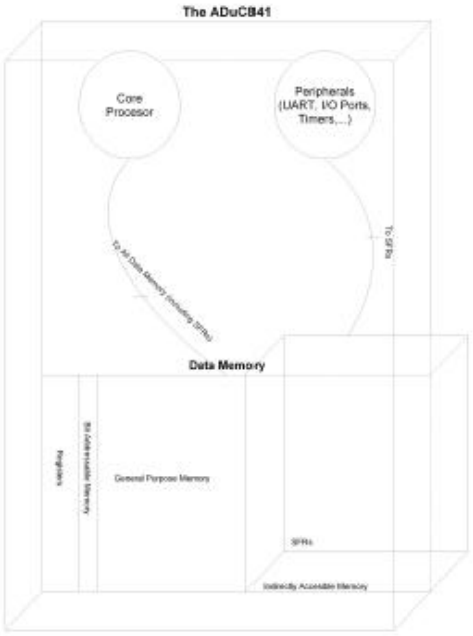


Figure 2.1: A Schematic Description of the ADuC841

2.2 האוגרים

- ל- ADuC841, כמו לכל אלו החברים במשפחת ה- 8051, יש ארבעה "בנקים" של אוגרים שכל אחד מהם מכיל שמונה אוגרים של 8 ביטים. בכל זמן נתון, רק "בנק" אחד של רגיסטרים, פעיל. האוגרים בבנק הפעיל מיוחסים כ- R0 עד R7. המון פקודות מעוצבות כך שיעבדו בצורה הטובה ביותר עם האוגרים הנ"ל. בהתחלה, "הבנק" הראשון של האוגרים פעיל ומצביע המחסנית מצביע על התחלת ה"בנק" השני של האוגרים.
- האוגר היחידי והחשוב ביותר שיש הוא כנראה הסוכם, ה- Accumulator. שלרוב נקרא ACC. ה- Accumulator בא לידי שימוש ברוב הפעולות האריתמטיות ובהרבה מקומות נוספים. כפי שנראה, הרבה פקודות עובדות בצורה הטובה ביותר עם ה- Accumulator, בעוד שפקודות אחרות עובדות *רק* עם ה- Accumulator.

2.3 הזיכרון

- כפי שהזכרנו קודם, ל- ADuC841 יש כמה סוגים של זיכרון. בניגוד לארכיטקטורת Von Neumann ה"סטנדרטית" שבה זיכרון התוכנית וזיכרון המידע נמצאים באותו בלוק פיזי של הזיכרון, ל- ADuC841 יש אזורים נפרדים לזיכרון מידע ולזיכרון תוכנית.
- למעשה, זיכרון המידע וזיכרון התוכנית יכולים להיות מחולקים עוד לזיכרון מידע פנימי, זיכרון מידע חיצוני, זיכרון תוכנית פנימי, וזיכרון תוכנית חיצוני.
- יש מגוון של דרכים שאנחנו יכולים לגשת לזיכרון. השיטה הפשוטה ביותר לגישה למקום בזיכרון היא להגיד למחשב שאתה רוצה לגשת למקום ספציפי ולהגיד לו בדיוק איפה זה. שיטה זו נקראת *גישה ישירה (Direct Addressing)*. הפקודה MOV מזיזה או מעתיקה בייט של מידע ממקום אחד למקום אחר. אם נכתוב את הפקודה MOV 40H, 50H זה יגרום להעתקת תכולת הזיכרון מכתובת 50H אל תוך כתובת 40H בזיכרון המידע הפנימי.
- כפי שהוזכר קודם, ל- ADuC841 יש ארבעת "בנקים" של אוגרים שבכל אחד מהם יש 8 אוגרים. "בנקי" האוגרים האלה ממוקמים בזיכרון המידע מכתובת 00H עד 1FH. (זאת אומרת, האוגרים ממופים לזיכרון מכתובת 00H עד 1FH, וכל בייט מכיל אוגר אחד).
- שיטה שנייה לגשת לזיכרון היא *לא ישירה (Indirect)*. כדי לגשת באופן לא ישיר לבייט, מאחסנים את כתובת הבייט שרוצים לגשת אליו באחד מהאוגרים R0 או R1. לאחר מכן, כשמתייחסים לכתובת הבייט הנ"ל משתמשים בסימן ה-@. לדוגמא, כדי להעביר את תכולת התא בזיכרון שנמצא בכתובת המאוחסנת באוגר R0 אל כתובת 40H, צריך לכתוב MOV 40H, @R0.
- לזיכרון המידע הפנימי יש 2 תת-חלוקות נוספות שמעניינות אותנו. חלק מהכתובות בזיכרון המידע הפנימי יכולות להיות נגישות ביט-ביט. אנחנו נראה שיש מכלול שלם של פקודות שמאפשרות לנו לפעול על ביטים בודדים בקלות. בנוסף, זיכרון המידע הפנימי, מכתובת 80H עד FFH, שמור לאוגרים לפעולות מיוחדות (SFRs). כל גישה לאוגרים האלה חייבת להיעשות בשיטה הישירה. כדי לאפשר 256 בייטים של זיכרון (לשימוש כללי), גישה לא ישירה לזיכרון החל מכתובת 80H עד FFH מתייחסת לזיכרון שונה לחלוטין היושב בכתובות אלו. זיכרון זה מיועד לשימוש כללי ואין לו שום קשר, בשום צורה ובשום אופן ל – SFRs.

2.4 שפת מכונה ושפת אסמבלר

- ה- AduC841 לא באמת מבין פקודות מהצורה של MOV 40H, 50H. פקודות מהצורה הזו כתובות בשפה – שנקראת שפת אסמבלי, שהיא רמה אחת מעל שפת המכונה עצמה – שנקראת באופן מפתיע, שפת המכונה. הסיבה שמשתמשים בשפת אסמבלי היא שבשפת המכונה יש הרבה פקודות דומות ששפת האסמבלי מסוגלת לאגד לפקודה אחת.
- הפקודה MOV A, Rn. הפקודה הזו גורמת לתכולת האוגר Rn להיות מועתקת ל- Accumulator, A. פקודה זו נשמרת בזיכרון התוכנית כמספר: 11101rrr כששלושת הביטים האחרונים מסמלים את מספר האוגר (n) בבינארי. לכן, הפקודה MOV A, R0 נשמרת כ- 11101000. נקודה שחשוב לציין היא שבפקודה של שפת המכונה, העובדה שבייט מועתק לתוך ה- Accumulator היא חלק מהפקודה. (בגלל זה משתמשים ב-A לייצג את ה- Accumulator ולא ב- ACC. אנחנו לא מעוניינים להגיד למעבד שאנו רוצים להעביר למידע ל- Accumulator, אלא, הפקודה שאנו נותנים למעשה אומרת "תעביר מידע מאחד מהאוגרים לשימוש כללי rrr אל תוך ה- Accumulator".)

- אם נרצה להעביר מידע מה- Accumulator לאחד מהאוגרים לשימוש כללי, נכתוב את הפקודה MOV Rn, A. הפקודה בשפת מכונה המתאימה לפקודה זו היא 11111rrr. זה לא נראה יותר מדי דומה לפקודה הקודמת, למרות שהמשמעויות, הן די דומות.
- הפקודה Mov direct, direct, כאשר direct הכוונה למקום ספציפי בזיכרון המידע הפנימי. הפקודה הזו מקודדת כך:
  כתובת ישירה של היעד כתובת ישירה של המקור 10000101
  הפקודה הזו דורשת 3 בייטים בזיכרון התוכנית.
  בנוסף, לפקודה לוקח 3 מחזורי שעון על-מנת להתבצע. (2 הדוגמאות הקודמות דרשו רק בייט אחד של זיכרון ורק מחזור אחד של שעון על-מנת להתבצע.)

2.5 הזיכרון – מנקודת מבט של מתכנת

2.5.1 זיכרון מידע פנימי

- ל- ADuC841 (כמו לכל מעבדי ה- 8051) יש כמה סוגים של פקודות לשימוש בזיכרון. הסוג החשוב ביותר של פקודות למתכנת הוא אלו שעוסקות בזיכרון המידע הפנימי. פקודות אלו הן אלו שהשמות שלהן הם הפשוטים ביותר במידת האפשר. כמו שכבר ראינו קודם, הפקודה ע"מ להזיז מידע בזיכרון המידע הפנימי היא רק MOV. ישנם הרבה MOVים שניתן לבצע, וה- 8051 מאפשר כמעט את כולם. למעשה, הפקודה MOV היא הפקודה המגוונת ביותר שנמצאת בלקסיקון של ה- 8051.
- כבר ראינו קודם איך להזיז מידע מה- Accumulator לאוגר לשימוש כללי. מאוגר לשימוש כללי ל- Accumulator. מכתובת אחת לאחרת. ומכתובת בגישה לא ישירה לכתובת ישירה.
- עוד אפשרות חשובה של MOV, היא האפשרות להעביר קבוע נתון אל תוך כתובת בזיכרון. נניח שנרצה להעביר את המספר 50H אל תוך כתובת 40H.
  אז, נכתוב את הפקודה MOV 40H, #050H. הסימן # גורם למהדר להגיד למעבד להזיז את הערך למקום מסוים. אחת מהדרכים הפשוטות ביותר לכתוב תוכנית שנראית בסדר גמור וגם נכונה סינטקטית (תחבירית) היא לכתוב MOV 40H, 50H כשלמעשה אנו רוצים לכתוב MOV 40H, #050H. שתי הפקודות הנ"ל הן חוקיות.
  הראשונה מעבירה את הערך השמור בכתובת 50H אל כתובת 40H, השנייה מעביר את הערך המספרי 50H אל כתובת 40H. יש הבדל גדול בין הפקודות. מבחינה תחבירית הן כמעט לחלוטין זהות.
- רוב פקודות ה- MOV שאנו חושבים עליהן – חוקיות. אחרות, לעומת זאת, לא. לדוגמה אם כתבנו את הפקודה MOV R0, R1, נגלה שהמהדר לא מכיר את הפקודה הזו.

2.5.2 זיכרון תוכנית

- זיכרון תוכנית ב- ADuC841 הוא פנימי לחלוטין. ישנן שתי פקודות – ורק שתי פקודות – ע"מ להביא מידע מזיכרון התוכנית. אחת הדרכים היא להשתמש בפקודה MOVC A, @A+DPTR, הדרך השנייה היא הפקודה MOVC A, @A+PC.
- הפקודה הראשונה קוראת את הערך שנשמר בתוך זיכרון התוכנית בכתובת הניתנת ע"י חישוב הסכום: הערך הנוכחי של האוגר DPTR (16 ביטים) ועוד הערך הנוכחי של ה- Accumulator, ומעבירה אותו ל- Accumulator (דורסת את הערך שהיה שמור בו קודם).

- הפקודה השנייה עושה את אותו הדבר, אבל משתמשת באוגר ה- PC, (Program Counter), במקום באוגר ה- DPTR.
- הפקודות הללו באות לידי שימוש ע"מ לקרוא ערכים במנות קבועות – כיוון שהערכים השמורים בתוך זיכרון התוכנית לא ניתנים לשינוי בזמן ריצת התוכנית.
  העובדה שקריאת הערכים היא קבועה מודגשת ע"י השם MOVC שע"פ ה- "80C51 Family Architecture" מסמל "MOVe Constant".

2.5.3 זיכרון מידע חיצוני

- בנוסף ל- 256+128 בייטים של RAM שיש לכל ה- 8052, ל- ADuC841 יש 2k נוספים של RAM שמוגדרים כ- RAM חיצוני. כדי לגשת לזיכרון זה, משתמשים בפקודות MOVX.

2.5.4 זיכרון נגיש ביט-ביט

- בנוסף לזיכרון המידע הרגיל, חלקים מסוימים בזיכרון המידע הינם נגישים ביט-ביט. זה אומר שאם מישהו רוצה לגשת לביט ספציפי – אז הוא יכול, במקום לגשת לבייט שלם. אפשר לכתוב אל תוך ביט, לקרוא ביט, ולבצע פעולות על ובשימוש ביט. זה מאפשר יישום של דגלים בלי לבזבז בייט שלם לביצוע עבודה שביט אחד יכול לבצע ביתר קלות.
- הכתובות הנגישות ביט-ביט בזיכרון המידע הן כל הביטים החל מכתובת 20H עד 2FH. בנוסף, כל אוגר לפעולות מיוחדות (SFR) שהכתובת שלו (בהקס-דצימלי) מסתיימת ב-0 או 8, גם הוא נגיש ביט-ביט. אוגרי ה- SFR הנגישים ביט-ביט הם אלו בהם לכל ביט יש חשיבות רבה. ברוב המקרים, כל ביט ישלוט על פונקציה מסוימת של רכיב כלשהו.
- הפקודה ע"מ להדליק ביט (כלומר, שווה 1), היא SETB bit והפקודה לנקות ביט (כלומר, שווה 0) היא CLR bit. בנוסף, ישנו מספר רב של פקודות שבוחנות ביט מסוים ואז קופצות. לדוגמא, הפקודה JB bit, rel, קופצת למקום rel אם הביט דלוק (שווה 1).

2.6 אריתמטיקה

- בביצוע פעולות אריתמטיות בשימוש ה- ADuC841 (או כל אחד מחברי משפחת הפשע 8051), לרוב חייבים להשתמש ב- Accumulator. לדוגמא, ישנן הרבה דרכים לחבר שני מספרים – וכולן כרוכות בשימוש ב- Accumulator. באופן דומה הפקודות שמבצעות חיסור, כוללות את Accumulator כאחד האופרנדים. הרבה מהפעולות הלוגיות פועלות בצורה הטובה ביותר עם ה- Accumulator. ה- Accumulator הוא האוגר החשוב ביותר שיש ל- ADuC841.

כדי לחבר שני מספרים, משתמשים בפקודת ה- ADD. נניח שאנו רוצים לחבר את הערך (תכולת הכתובת בזיכרון) הנמצא ב- 40H עם הערך הנמצא ב- 50H. כיוון שה- ADuC841 לא תומך בחיבור בין שני ערכים בכתובות שונות בזיכרון, נצטרך קודם כל להעביר את אחד מהערכים אל ה- Accumulator, ורק אז לבצע את החיבור. אחת הדרכים לבצע את זה הולכת ככה:

```
MOV A, 40H
ADD A, 50H
```

פקודת ה- ADD תמיד משאירה את הסכום של שני המספרים בתוך ה-Accumulator .
שימו לב שכאשר משתמשים בפקודה ADD, אם הסכום הוא יותר ממה שניתן לשמור בבייט אחד, אז מה שכן ניתן לשמור, נשמר בתוך ה- Accumulator, וביט ה"גלישה", ה- Carry, מודלק גם כן, ע"מ לסמן את הגלישה שהתרחשה. לכן, בדיקה של ביט הגלישה לאחר ביצוע של פעולת חיבור תראה לנו האם התרחשה "גלישה", או לא.

(ביט ה"גלישה" הוא הביט הראשון מתוך ה- Program Status Word, שהוא אחד מה- SFRs.)

- אם נרצה לבצע סכימה על מספר גדול של בייטים, נשתמש בפקודה ADD עבור הבייט הראשון, ובפקודה ADDC עבור כל השאר.
- הפקודה ADDC פועלת בדיוק כמו פקודת ה- ADD, רק שהיא גם מוסיפה 1 במקרה שביט ה"גלישה" דלוק.
- בנוסף לפקודת ה- ADD יש את הפקודה SUBB. למידע נוסף, אתם יודעים לאן ללכת.
- כיוון שבמקרים רבים אנו מעוניינים לספור, למנות אירועים שהתרחשו, הוספה של המספר 1 לערך מסוים או הורדה של המספר 1 מערך מסוים היא פעולה נפוצה ומאד שימושית. מסיבה זו, ל- ADuC841 (ולכל חברי מאפיית ה- 8051), יש פקודות מיוחדות להוספה והחסרה של 1. כדי להגדיל ערך ב-1, משתמשים בפקודה INC, וכדי להחסיר 1 מערך כלשהו, בפקודה DEC.
- הפקודות האלו פועלות על ה- Accumulator, על האוגרים לשימוש כללי, על כתובות בזיכרון או על כתובות בזיכרון הנגישות בצורה לא ישירה.
- בנוסף לחיבור וחיסור, ישנן גם פקודות המבצעות כפל וחילוק. הפקודה האלו שימושיות כאשר נרצה להסיט בייט, במספר ביטים.

2.7 פעולות לוגיות

- בנוסף לפעולות האריתמטיות, ישנו סט של פעולות לוגיות. הפעולות האלו מאפשרות לבצע OR בין שני בייטים, AND בין שני בייטים, XOR בין שני בייטים, וגם לסובב בייטים.
- פקודות הסיבוב (Rotate), מעניינות במיוחד. ישנן ארבע סוגים של פקודות. RL מסובבת את ה- Accumulator ביט אחד שמאלה. למעשה מתייחסים ל- Accumulator כ-"אוגר מעגלי", כך שהביט הראשון שלו מופיע מיד לאחר הביט האחרון.
- בצורה דומה עובדת גם הפקודה RR, נצפה ששם, הכול יזוז ימינה.
- ישנה פקודה נוספת הנקראת RLC שמסובבת שמאלה דרך ביט ה"גלישה". הפקודה הזו מתייחסת ל- Accumulator כאוגר עם גלישה, כלומר כאוגר עם 9 ביטים. ביט ה"גלישה" נחשב כביט הנמוך ביותר. לכן, בפקודה זו, ביט הגלישה מוזן לתוך הביט הנמוך ביותר של ה- Accumulator הבסיסי (8 ביט) והביט הגבוה ביותר של ה- Accumulator הבסיסי, מוזן לתוך ביט ה"גלישה".
- אם נצטרך להסיט מחרוזת של בייטים, נגיד מספר של 32 ביט, הפקודה הנ"ל יכולה להיות מאד שימושית. הפקודה RRC פועלת בדיוק באותן צורה רק מזיזה את הביטים בכיוון ההפוך.

2.8 פעולות על ביטים

- כמו שהזכרנו קודם, ישנן פקודות שמדליקות, ומנקות ביט. אפשרי גם לעשות AND בין ביטים, OR, להזיז ביטים, או להשלים ביטים.
- כשמתעסקים עם ביטים, תפקיד ביט ה"גלישה" דומה לתפקידו של ביט זה עבור התעסקות ה- Accumulator עם בייטים. הרבה פעולות מחייבות שימוש בביט זה.
- לדוגמא, כאשר עושים AND בין שני ביטים, אחד הביטים חייב להיות ביט ה"גלישה". ישנן 2 צורות לביצוע פעולת ה- AND. אחת מהן היא ANL C, bit, שלמעשה אומרת לעשות AND בין ביט ה"גלישה" לביט מסוים אחר.
- השנייה אומרת ANL C, /bit, שלמעשה אומרת לעשות AND בין ביט ה"גלישה" למשלים של ביט מסוים אחר.

- צריך לשים לב שבשתי הפקודות ה-C הוא ממש חלק מהפקודה. שפת המכונה עבור פקודה זו כוללת מידע שאחד מהביטים עליהם עושים AND הוא ביט ה"גלישה". הפקודה גם כוללת את המידע האם אותו הביט שנכנס ל- AND עם ביט ה"גלישה", צריך להופיע כמו שהוא, או בצורת המשלים שלו. לבסוף, עובדה נוספת המראה שביט ה"גלישה" משרת תפקיד זהה לתפקידו ב- Accumulator, (רק כאן עבור פעולות על ביטים), היא שתוצאת הפעולה נשמרת בביט ה"גלישה".
- בנוסף לפקודות שמטפלות בביטים, ישנן מספר פקודות נוספות שגורמות למעבד לקפוץ לכתובת חדשה בהתבסס על מצב של ביט מסוים. לדוגמא, הפקודה JC גורמת למעבד לקפוץ לכתובת חדשה עם ביט ה"גלישה" דלוק. זה בעצם אומר: "קפוץ אם ביט הגלישה דלוק". באופן דומה, יש את הפקודה JNC, שגורמת למעבד לקפוץ עם ביט ה"גלישה" לא דלוק. הדוגמאות ניתנו עבור ביט הגלישה, כמובן שישנן גם פקודות הגורמות למעבד לקפוץ לכתובת חדשה בהתבסס על מצב של ביט כלשהו אחר.

2.9 שליטה בזרימה

- הסט האחרון של פקודות עוסק בשליטה באופן ריצת התוכנית. הפקודה הנפוצה ביותר היא ככל הנראה JMP. לפקודה הזו אין שום תרגום ישיר לשפת מכונה. האסמבלר מתרגם את הפקודה הזו לאחת משלוש הפקודות האפשריות הבאות: LJMP, AJMP, SJMP. כל אחת מהפקודות הנ"ל גורמת לתוכנית לקפוץ לכתובת חדשה. הדרך בה כתובת זו נשמרת – והמיקום (הכתובת) שלה ביחס למיקום (הכתובת) הנוכחי בתוכנית, משתנה בין הפקודות הנ"ל.
- ישנה גם את הפקודה CALL לתת-שגרה. המתכנת בדרך כלל כותב CALL, והאסמבלר מתרגם את זה או ל- ACALL או ל- LACLL. בסופה של כל תת-שגרה, חייבים לחזור לתוכנית הראשית ממנה קראנו לתת-שגרה הזו. עושים זאת ע"י שימוש בפקודה RET.
- הפקודה הבאה היא DJNZ. היא אומרת "תוריד 1 ותקפוץ אם זה לא אפס". פקודה זו שימושית אם רוצים למשל, ליישם לולאה. נניח, שרוצים לעבור דרך לולאה 10 פעמים. אחת הדרכים לעשות זאת היא להשתמש ב- 40H כמו במונה, ואז להשתמש ב DJNZ כדי לספור את מספר הפעמים בהם עברנו בתוך הלולאה. למשל:

```
        MOV 40H, #10;  Move the number 10 into location 40H.
LOOP:   The body of the loop
        goes here.
        DJNZ 40H, LOOP
```

הלולאה משתמשת ב- 40H כמו במונה. נזכר שסימן ה- # הקודם למספר 10 אומר למהדר שאנחנו רוצים להעביר את המספר 10 אל תוך הכתובת בזיכרון.

- בנוסף לפקודות של קפיצה, קריאה, החסרה וקפיצה אם לא אפס, ישנן מספר פקודות של קפיצה ע"פ התניה.
- דוגמא לאחת כזו היא הפקודה CJNE שמשווה בין 2 ערכים וקופצת למקום חדש אם הם לא שווים.
- דוגמא נוספת היא JZ שקופצת אם ה- Accumulator הוא אפס,
- דוגמא נוספת היא JNZ שקופצת עם ה- Accumulator הוא לא אפס.

פרק 3

האסמבלר

3.1 הקדמה

למחשבים יש "שפת אם" – שפת המכונה שלהם – שמותאמת באופן מושלם עבורם. שפת מכונה של מחשב היא באופן כללי אוסף של מספרים שהסדר שלהם הינו בעל משמעות עבור המחשב. אדם הנחוש בדעתו יכול ללמוד לקרוא ולכתוב בשפת מכונה, ולתכנת ברמה הבסיסית ביותר של המכונה, אבל גם אלו בינינו שאוהבים את זה, לרוב לא רוצים, או צריכים, לעבוד ברמה בסיסית שכזו. הרמה הבסיסית ביותר שרובנו נעסוק בה היא שפת האסמבלי. שפת אסמבלי, כפי שכבר ראינו, היא עדיין רמה בסיסית ביותר, אבל כזו שעדיין מאפשרת לנו לעבוד בצורה הרבה יותר נוחה.

התוכנה שלוקחת קוד בשפת אסמבלי וממירה אותה לשפת מכונה נקראת "אסמבלר". בפרק זה נכיר את האסמבלר עבור ה- ADuC841. באופן כללי אסמבלר הופך את חיינו לקלים יותר מכמה בחינות:

- הוא מתרגם פקודות בשפת אסמבלי לפקודות בשפת המכונה.
- הוא מאפשר לנו להעניק שמות סימבוליים לכתובות בזיכרון.
- הוא מאפשר לנו לקבוע היכן משתנים וקטעי קוד ימוקמו, בצורה נוחה יחסית.
- הוא מאפשר לנו להוסיף הערות לתוכנית שלנו.

3.2 חומר עזר, חומר רקע

בקובץ ASM51.pdf

3.3 האסמבלר כמתרגם

תיארנו מספר רב של פקודות שבהן תומך ה- ADuC841. אפילו הקדשנו זמן רב ללימוד פקודות אלו. אבל, לא יותר מדי זמן בניסיון להבין איך המעבד "אוכל" את הפקודות האלו. נתקן את החוסר האיום הזה עכשיו.

נחשוב על הפקודה החשובה MOV. הפקודה הזו אומרת למעבד להזיז מידע ממקום אחד, לאחר. כמו שכבר ראינו, ישנן הרבה פקודות בשפת מכונה המתאימות לסגנון פקודה זה. אז איך הפקודות שאנחנו כותבים באסמבלי מתורגמות לשפת מכונה בצורה המתאימה?

בשפת אסמבלי זה די קל. נתייחס לפקודה MOV 40H, #50. כיוון שכל סימבול בפקודה הזו הינו חד-משמעי, אנחנו יכולים באופן מיידי לקבוע שהפקודה הזו אומרת למעבד להזיז את המספר 50 (בערך דצימלי, כיוון שאין H אחרי המספר 50) אל התא בכתובת 40H. אם נסתכל על ה- "8051 Programmer's Guide" בעמוד 39, נראה שבשפת מכונה זה הולך ככה:

```
0111 0101    0100 0000    0011 0010
```

הבייט הראשון הוא קוד שאומר "תעביר מידע קבוע למקום מסוים". הבייט השני הוא 40H בהצגה בינארית. הבייט השלישי הוא המספר 50 (עשרוני) בהצגה בינארית. עכשיו זה די ברור שתוכנה יחסית פשוטה יכולה לטפל בסוג כזה של תרגום.

כמו שהוזכר בפרק הארכיטקטורה (יש אחד כזה?), ישנן משימות אותן מבצע המהדר. שתי הפקודות JMP ו- CALL צריכות להיות מתורגמות לצורה האמיתית המתאימה. תרגום מסוג כזה, דורש תוכנית מעט יותר חכמה, אבל זה ישים לחלוטין.

האסמבלר מסייע עוד, ומאפשר להפטר מהצורך לעקוב אחר כתובות פיזיות וערכי קבועים. האסמבלר מאפשר להצמיד שמות (תוויות) לכתובות בזיכרון, למקומות בתוך תוכנית, לקבועים ולביטים ספציפיים. זה מאפשר את קריאת התוכנית בצורה הרבה יותר נוחה – מטרה שהמתכנת חייב תמיד לזכור. בנוסף, מאפשר האסמבלר הוספת הערות לתוכנית.

הרבה מהפקודות שגורמות למעבד לבצע קפיצה, למעשה גורמות למעבד לקפוץ מספר x מסוים של בייטים קדימה (או אחורה) בזיכרון התוכנית. בפקודות מסוג זה, האסמבלר מאפשר לכתוב תווית למקום כלשהו בזיכרון התוכנית. האסמבלר אז מחשב כמה הוא המרחק עד אותה תווית, ואז מכניס את הערך הזה במקום המתאים בזיכרון התוכנית. נסתכל לדוגמא על קטע הקוד הבא:

```
        MOV R0, #10
LOOP:   DJNZ R0, LOOP
```

הפקודה בשורה הראשונה טוענת את האוגר לשימוש כללי, R0, במספר 10 (עשרוני). השורה השנייה מתחילה במילה LOOP. סט זה של תווים לא מתורגם לשפת מכונה בכלל! תווים אלו רק אומרים למהדר לקשר את התווית LOOP אל המקום בזיכרון התוכנית בו מתחילה הפקודה DJNZ R0, LOOP.

בנוסף, הפקודה DJNZ דורשת 2 פרטים. הכתובת או האוגר שיש להחסיר ממנו, במקרה זה R0. הפקודה גם דורשת את הכמות היחסית שיש לקפוץ. זה לא ניתן באופן ישיר. מה שכן נתנו למהדר זו תווית. המהדר חייב לקחת את התווית ולחשב כמה רחוק צריך לקפוץ. המהדר מבצע את החישוב הזה ואז מציב את הערך הזה במקום המתאים בזיכרון התוכנית.

כאן ראינו דוגמא טובה איך המהדר חוסך לנו מאמץ – אנחנו לא צריכים לדאוג לגבי חישוב מרחקי קפיצה – ולחסוך כמות רבה יחסית של שגיאות – כיוון שחישוב וביצוע הקפיצה בול למקום הנכון, זה טריוויאלי אבל לא תמיד קל ליישום.

3.4 להגיד לאסמבלר איפה זה

ראינו כבר שיש מספר סוגים שונים של זיכרון. האסמבלר מסוגל לטפל בכולם אבל חייבים להגיד לו על איזה חלק זיכרון הוא עובד. בנוסף, הוא חייב לדעת איפה הוא ממוקם בתוך חלק זיכרון זה.

דיברנו על ארבעה מחלקי הזיכרון. החשוב ביותר הוא זיכרון הקוד. כדי להגיד לאסמבלר שאנו רוצים את מה שבהמשך, לשים בזיכרון הקוד, משתמשים בהכוונה ע"י CSEG. הכוונה זו לא מתורגמת לשפת מכונה בכלל – היא לא אומרת למיקרו-מעבד מה לעשות. היא רק הכוונה שאומרת לאסמבלר איך לפרש את מה שכתוב אחרי אותה הכוונה.

אם לא נגיד לאסמבלר אחרת, זיכרון הקוד הוא הזיכרון אליו פונה האסמבלר כברירת- מחדל. אם נוסיף הכוונה ע"י שימוש ב- AT בסוף הכוונת החלק בזיכרון, נגרום לאסמבלר לזוז אל אותו החלק בזיכרון ולהתחיל מהמקום הנתון באותו החלק. אם נרצה להתחיל מההתחלה של זיכרון הקוד, אז נרשום CSEG AT 0H. כדי לגרום לאסמבלר לשנות למיקום אחר באותו חלק הזיכרון, משתמשים בהכוונה ע"י ORG. כלומר, אם אנחנו כבר נמצאים בזיכרון הקוד ואנחנו רוצים לשנות מיקום לכתובת 20FH, נכתוב ORG 020FH.

שלושת חלקי הזיכרון האחרים הרלוונטיים לנו הם זיכרון המידע, זיכרון המידע החיצוני, והזיכרון הנגיש ביט-ביט. כדי לעבור לזיכרון המידע משתמשים בהכוונה ע"י DSEG. כיוון שכתובות 00H עד 1FH שמורות לארבעת "בנקי" האוגרים הכלליים, לרוב לא נרצה להשתמש בזיכרון שבין 00H ל- 2FH, לשמירת מידע כללי. לכן, כאשר עוברים לחלק זיכרון המידע, לרוב משתמשים בהכוונה, DSEG AT 030H.

ישנן כמה דרכים נוספות בהן המהדר עוזר לנו. נניח שאנחנו רוצים להגדיר מונה בזיכרון המידע. אחת הדרכים היא שנעקוב אחר כל המקומות בזיכרון המידע ולהשתמש במידע זה כפי שאנו רוצים. דרך אחרת היא לתת למהדר לערוך את החשבונות.

נסתכל על קטע הקוד הבא:

```
DSEG AT 30H
COUNTER1:       DS      1
COUNTER2:       DS      1
```

השורה הראשונה אומרת למהדר שההכוונה הבאה מתייחסת לחלק זיכרון המידע ושמהדר צריך להתחיל להקצות משתנים החל מכתובת 30H באותו חלק זיכרון. השורה השנייה קובעת של- COUNTER1 יוקצה למקום הבא בזיכרון, במקרה זה, 30H. DS בא לידי שימוש כאשר מקצים מקומות (בייטים) בזיכרון המידע. המספר 1 קובע שעל בייט אחד להיות מוקצה. (אם היינו כותבים 3, אז היו מוקצים 3 בייטים.) השורה הבאה קובעת שבייט אחד הבא, כלומר 31H, יוקצה ל- COUNTER2. שימו לב שע"י כך, נחסכה מאיתנו הדאגה היכן בדיוק ממוקמים אותם הבייטים. מעתה, ברגע שנרשום COUNTER1 ו- COUNTER2 המהדר יפנה לכתובת 30H ו- 31H בזיכרון המידע הפנימי, בהתאמה.

כדי לעבור לחלק הזיכרון החיצוני, משתמשים בהכוונה ע"י XSEG.

כדי להקצות בייטים (מקום) בחלק הזיכרון החיצוני, משתמשים ב- DS בדיוק כמו קודם.

חלק הזיכרון הרביעי והאחרון שמשמעותי עבורנו, הוא הזיכרון הנגיש ביט-ביט. כדי לעבור לחלק זה של הזיכרון משתמשים בהכוונה ע"י BSEG. בתוך חלק זה משתמשים ב- DBIT ע"מ להקצות מקום (ביטים). דוגמה פשוטה לשימוש בנ"ל נראית כך:

```
BSEG
FLAG:   DBIT    1
```

מה שקורה שם זה להקצות ביט אחד עבור FLAG, וכתובתו מקושרת ל- FLAG.

Header Files 3.5

כאשר מתכנתים לתוך ה- ADuC841, הרבה בייטים וביטים באים לידי ביטוי. כל אוגרי ה- SFR נמצאים בזיכרון בכתובות 080H עד 0FFH. למרות שנוכל (אולי) לזכור בע"פ את כל הבייטים והביטים השונים ואת המיקומים שלהם בזיכרון, דבר זה אינו נחוץ או אפילו רצוי. גם אם ידענו את כל אותם מקומות רלוונטיים בזיכרון, האדם הבא שייקרא את התוכנית עלול לא לדעת למה כל אותם מקומות בזיכרון מיועדים. כדי לכתוב תוכנית קריאה, מומלץ לתת שמות בעלי משמעות לכל אותם מקומות. ניתן לתת שמות למקומות בזיכרון, בייטים, וביטים.

למרות זאת, Analog Devices מספקת סט של קבצים, בדומה לקבצים של שפת C, שמטפלים בעבודה זו בשבילנו. קובץ ה- Header עבור ה- ADuC841 נקרא MOD841. אם אנחנו לא רוצים להשתמש באף אחת מהפריפריות הייחודיות ל- ADuC841, כלומר אנו רוצים לכתוב קוד התואם את ה- 8052, אנחנו יכולים להשתמש בקובץ Header שנקרא MOD52. שני הקבצים משייכים שמות לביטים ובייטים נפוצים בשימוש. ע"י שימוש בקבצים אלו אפשר לכתוב תוכנית שתהיה ברורה יותר ומובנת לכאלו שמכירים את משפחת 8051 וה- ADuC841.

כדי להגיד למהדר לקרוא ולהתייחס לקובץ Header, צריך להוסיף את הפקודה $INCLUDE(MOD52) או $INCLUDE(MOD841) . פקודה זו תופיע בתחילת התוכנית.

יש לשים לב שבמערכת בה אנו משתמשים, כל השמות של אוגרי ה- SFR שמופיעים ב- 8051 כבר מוגדרים. אם כל אוגרי ה- SFR הנמצאים בתוכנית כבר מוגדרים מראש, אין צורך לכלול את קובץ ה- Header.

הערות 3.6

חלק נוסף בכתיבת קוד קריא הוא שימוש בהערות. באסמבלר שלנו, הערות מתחילות בסימן ; . החל מסימן זה, ועד סוף שורה, כל מה שביניהם זה הערה. הערות אלו הן לרוב מה שעומד בין המתכנת למה שעשוי להיות קוד לא- קריא לחלוטין.

חשוב שתהיה הקדמה, לפני שתוכן התוכנית ה"שימושי" מופיע. ההקדמה צריכה לכלול את שם המתכנת, שם הקובץ, תאריך, תאריך עדכון (אם יש כזה), החומרה אליה מיועדת התוכנה, ותיאור קצר של מטרת התוכנית. בנוסף, כל פקודה שייעודה עשוי להיות לא ברור לקורא צריך להיות מלווה בהערות, עד הגעה למצב שבו קורא חסר ניסיון יבין את תכלית הייעוד של הקוד.

3.7 ההכוונה האחרונה

כל תוכנית חייבת להסתיים ב- END. הדבר היחיד שיכול להופיע אחרי END זה הערה, וגם זה, באותה השורה בלבד. אם נמקם קטע קוד או הערות בשורות שלאחר ה- END, המהדר יקטלג זה כשגיאה ולא יקמפל את התוכנית.

3.8 הרצת הקומפיילר (המהדר)

לאורך הקורס, נשתמש ב- Keil integrated development environment, ה- IDE. ה- IDE זו תוכנה שיכולה לעשות הרבה דברים בשביל לעזור לנו. אחת הפונקציות שלה זה המהדר. כדי להריץ את המהדר, צריך "לבנות" את הפרויקט.
אחרי שה- IDE אותחל כהלכה, אחרי שהמהדר מסיים את הריצה, הוא מפיק 2 קבצים. הראשון הוא קובץ hex.* שהוא קובץ בשפת מכונה המתאים להורדה אל ה- ADuC841 . הקובץ השני הוא lst.*, הקובץ הזה מכיל את הקוד המקורי, את הקוד בשפת מכונה, רשימת שגיאות, רשימה של כתובות, ומידע נוסף שעשוי להיות שימושי בזמן Debugging. קובץ זה, עשוי להיות שימושי ביותר.

3.9 הורדה של התוכנית ל- ADuC841

אחרי שאותחל כהלכה, ה- IDE יכול להוריד תוכניות אל ה- ADuC841 . תוכניות נשלחות אל ה- ADuC841 בעזרת ה- UART. ה- UART בא לידי שימוש בכל מיני צורות של תקשורת. לפני שמורידים תוכנית אל תוך ה- ADuC841 , חייבים להגיד למעבד שעומדת להגיע תוכנית חדשה. אומרים למעבד שעומדת להגיע תוכנית חדשה ע"י לחיצה רצופה של כפתור ה- Serial Download בזמן שמאתחלים את ה- ADuC841 . ברגע שזה נעשה, אפשר לעשות LOAD לתוכנית החדשה מתוך ה- IDE. בהנחה שהכול התנהל כשורה, התוכנית אמורה לרוץ כעת באופן מיידי. אם לא, פונים למישהו אחר, לוקחים את התוכנית שלו, וחוזרים על אותן הפעולות שוב, עד שזה עובד.

פרק 4

פסיקות

4.1 הקדמה לפסיקות

מיקרו-מעבדים לרוב מתקשרים עם מגוון פריפריות. במשפחת ה- 8051 ומיקרו-מעבדים דומים זה נעשה לרוב ע"י פסיקות וה-SFRs.

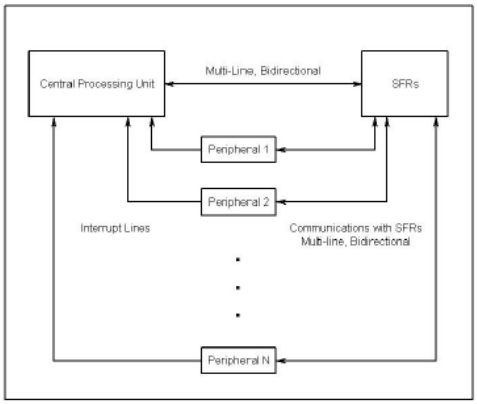


Figure 4.1: A Schematic Description of the Interaction between the CPU, the SFRs, and the Peripherals in the ADuC841. Note that all of these interacations take place *inside* the chip.

- אפשר לחשוב על פסיקה כעל הפרעה לפעולה התקינה של המעבד. ניקח לדוגמה את הפסיקה הפשוטה ביותר, פסיקה המתרחשת כאשר המתח על פין מסוים במעבד משתנה באופן מסוים. כאשר המתח על פין זה משתנה, המעבד קופץ מיד למקום קבוע בזיכרון התוכנית, וממשיך לבצע מה שיש שם. באותו הזמן, המעבד שומר את המקום בו הוא היה קודם לכן (לפני הקפיצה) כך שבתום ביצוע אותן הפקודות שנמצאות במקום הקבוע, הוא יוכל לחזור ולבצע את שאר הפקודות, מהמקום בו הן הופסקו.
- במיקרו-מעבדים במשפחת ה- 8051 כל אחת מסוגי הפסיקות הרבות, מקפיצה את המעבד למקום אחר. כך לדוגמא, External Interrupt Zero, שמתרחשת כאשר משתנה המתח על פין 2 בפורט 3 (P3.2) בצורה מסוימת, מקפיצה את המעבד לכתובת 3 בזיכרון התוכנית.
- סט המקומות אליהן קופץ המעבד בהתרחש כל סוג פסיקה נקרא טבלת וקטורי הפסיקות, Interrupt Vector Table.

- ישנם 8 בייטים בין כל מקום בטבלת וקטורי הפסיקה. כך, שאם כל התגובה (זותומרת כל הטיפול הדרוש) לפסיקה הינו שמונה בייטים או פחות, אז התת-שגרה של הפסיקה (Interrupt Subroutine = ISR = סט הפקודות לביצוע לאחר התרחשות הפסיקה) יכולה להיות ממוקמת בדיוק ע"פ טבלת וקטורי הפסיקה. אחרת, צריך לקפוץ מהמקום המתאים בטבלת וקטורי הפסיקה אל תת-השגרה הזו.
- לאחר סיום הטיפול בפסיקה, כלומר לאחר סיום הפקודות ב- ISR, צריך להגיד למעבד לחזור לביצוע הרגיל של הפקודות, החל מהמקום בו הוא הפסיק. את זה עושים ע"י הפקודה RETI. (Return from Interrupt).

4.2 פסיקות – בקצרה

- יש הרבה פסיקות, סתם שתדעו. הנה חלק מהן, בהן נשתמש בקורס.
- External Interrupt Zero – קשורה למצב של P3.2.
- Timer Zero Interrupt
- External Interrupt One – קשורה למצב של P3.3.
- Timer One Interrupt
- Serial Port Interrupt
- Timer Two Interrupt
- Analog to Digital Interrupt

4.3 למה להשתמש בפסיקות

- המעבד, גם זה שאנחנו עובדים איתו, שהוא לא מהיר במיוחד, עדיין עובד יותר מהר מהתהליכים אותם הוא צריך לבצע.
- דרך אחת לעבוד, היא לתת למעבד להריץ תהליכים, שהם כאמור איטיים יותר ממנו, לחכות כל פעם עד שהתהליך יסתיים, ואז להמשיך לתהליך הבא. דרך עבודה זו היא בזבזנית, טיפשית, מיותרת! היא מבזבזת זמן יקר של המעבד. רוב הזמן של המעבד מתבזבז בלחכות שהתהליך יסתיים, כשלמעשה המעבד מסוגל לבצע עוד פעולות באותו הזמן.
- דרך אחרת, ע"י שימוש בפסיקות, היא לתת לאיזושהי פריפריה לבצע משהו, וכאשר היא מסיימת לבצע את אותו המשהו, להפריע לפעולת המעבד ע"י פסיקה.

  דוגמה לכך היא פסיקה מסוג – Serial Transmit Interrupt.

  מעבד 8051 טיפוסי יכול לבצע מיליון פעולות בשנייה.

  מעבד ADuC841 יכול לבצע 10 מיליון פעולות בשנייה.

  שידור בייט אחד דרך הפורט הסיריאלי במהירות סבירה לוקח בערך חצי מילי-שנייה.

  בחצי מילי-שנייה הזו, יכול המעבד לבצע 5,000 פעולות.

  שידור חמישה בייטים דרך הפורט הסיריאלי במהירות סבירה ייקח בערך 2.5 מילי-שנייה.

  ב-2.5 מילי שנייה האלו, יכול המעבד לבצע 25,000 פעולות.

  אחת הדרכים לשידור חמישה בייטים היא לשדר בייט, לחכות שיסתיים, ואז לשדר את הבייט הבא...

  הדרך הטובה יותר לעשות את זה, היא להשתמש ב- Serial Transmit Interrupt.

  מה שאז קורה זה, שנותנים למעבד להעביר את הבייט הראשון לרכיב פריפריה שמטפל בתקשורת סדרתית, ושהוא ישדר את הבייט הראשון. בינתיים המעבד יחזור לפעולות הרגילות.

כשהרכיב הזה יסיים לשדר, הוא יפריע לפעולת המעבד ע"י הפסיקה הנ"ל, הפסיקה הזו תגרום לקפיצה למקום המתאים בזיכרון, שם תבוצע העברת הבייט הבא המיועד לשידור אל הרכיב בפריפריה כך שישדר את הבייט הבא, וחזרה לפעולה סדירה של המעבד, וכך הלאה.

4.4 אפשור של פסיקות

- כדי לאפשר למעבד להגיב להתרחשות של פסיקות חייבים לעשות כמה דברים.
- חייבים לאפשר התרחשות פסיקות באופן גלובלי.

את זה עושים ע"י הדלקת הביט הכי גבוה באוגר Interrupt Enable Register, שהוא אחד מה- SFR, שנמצא בכתובת A8H. ב- MOD841 וב- MOD52 יש לביט הזה שם, קוראים לו EA. לכן עושים SETB EA.

- חייבים לאפשר התרחשות פסיקות מסוג ספציפי.

לדוגמא, כדי לאפשר פסיקה מסוג External Interrupt Zero, צריך להדליק את הביט הנמוך ביותר של האוגר הנ"ל. שוב- יש לביט הזה שם, קוראים לו EX0. לכן עושים SETB EX0.

- לכל פסיקה יש ביט שלה שצריך להיות מודלק ע"מ לאפשר את התרחשותה.
- לסיכום, כדי לאפשר התרחשות פסיקות, צריך לאפשר התרחשות פסיקות באופן גלובלי, ואז לאפשר את התרחשות הפסיקות באופן ספציפי.
-

פרק 5

מבט פילוסופי על ה- SFRs

5.1 שני סוגים של SFRs

- שני סוגים של אוגרים ממופים אל תוך האזור של ה- SFRs. (מכתובת 80H עד כתובת FFH).
- סוג ראשון משמש להגדרת אופן פעולת המיקרו-מעבד, או לשימוש המיקרו-מעבד בזמן הפעולה שלו.
  ה- Accumulator שנמצא בכתובת E0H ונקרא ACC הוא כזה.
  ה- Interrupt Enable Register שנמצא בכתובת A8H ונקרא IE הוא כזה.
  ה- Stack Pointer שנמצא בכתובת 81H ונקרא SP הוא כזה.
- הסוג השני משמש את המעבד לתקשר עם הפריפריות. האוגרים האלו מעבירים מידע מהמעבד אל הפריפריות החיצוניות האלו, ולהפך.
- במשפחת הפשע של ה- 8052, יש זיכרון רגיל שנמצא בכתובות 80H עד כתובת FFH, לכאורה אותו המיקום של ה- SFRs.
  כל גישה ישירה לכתובות בתחום הנ"ל מיוחסת לקריאה ו/או כתיבה מה- SFRs.
  כל גישה לא ישירה לכתובות בתחום הנ"ל מיוחסת לזיכרון הרגיל שנמצא בתחום הכתובות האלו.

5.2 SFRs שהם נגישים ביט-ביט

- שש עשרה מתוך כל ה- SFRs הם נגישים ביט-ביט.
  ה- Accumulator, ה-SFRs ששולטים על פורט 0,1,2,3, וחלק מה- SFRs ששולטים על הטיימרים.
- לא צריך לזכור איזה SFR מתאים לאיזו כתובת, הקבצים MOD841 ו- MOD52 מגדירים את כל מה שאנחנו צריכים.
- כברירת מחדל, הקומפיילר מזהה את כל השמות של ה- SFRs הנמצאים ב- 8051.

פרק 6

לא מעניין

פרק 7

הקדמה ל- ADuC841

- שורה שמתחילה בסימן ; זו הערה.
- כמנהג, כותבים את כל הפקודות והאופרנדים שלהם באותיות גדולות.
- תדר המעבד הוא $11.0592M[Hz]$.
- כל מחזור שעון הוא $\frac{1}{11.0592M} = 0.0904\mu[Sec]$.

- פין 4 של פורט 3, P3.4 חובר פיזית לדיודת LED.
  כשהמתח על הפין הוא 0 וולט, ה-LED תאיר.

- EQU מחליף את המילה שלפניו במילה שאחריו.
  LED EQU P3.4, כל מקום בו מופיעה המילה LED האסמבלר ישים P3.4.
- DELAY: MOV R7, #8, התווית DELAY היא הכתובת בזיכרון הקוד בה מופיעה הפקודה MOV R7, #8.
- את הפקודות JMP ו-CALL מתרגם האסמבלר לשפת מכונה בעצמו, ע"פ סט שיקולים שלו, כפי שהוסבר כבר קודם.
- CLR – פועלת על ביט ומאפסת אותו.
- SETB – פועלת על ביט ומגדירה אותו להיות שווה לאחד.
- CPL – פועלת על ביט ועושה בעצם את פעולת המשלים.
- MOV – נו באמת.
- לולאות, הפקודה השימושית היא DJNZ. היא יכולה לקחת 3 או ארבעה מחזורי שעון (clock cycles).
  DJNZ R_, LABEL – זה ייקח 3 מחזורי שעון.
  DJNZ DIRECT ADDRESS, LABEL – זה ייקח 4 מחזורי שעון.
  אפשר להשתמש גם בסימן $ בתור תווית. זה גורם לתווית להיות ההתחלה של הפקודה הנוכחית.

פרק 8

המחסנית

8.1 מתי ולמה להשתמש במחסנית

- מחסנית – חוצץ FILO, ראשון נכנס אחרון יוצא, אחרון נכנס ראשון יוצא.
- כתיבה של ערך למחסנית נקראת PUSH.
- קריאה של ערך מהמחסנית נקראת POP.
- ל-8051 וליורשים שלו יש מחסניות כבר Build-in.
- על חלקה העליון של המחסנית מצביע ה- SP. אחד מה- SFRs.
- כברירת- מחדל, מתחילה המחסנית מכתובת 08H וגדלה לכתובות גדולות יותר.
- הגישה למחסנית היא בשיטה הלא- ישירה, ע"י גישה לכתובת עליה מצביע ה- SP.
- כשמופיעה פקודת CALL, או כשמתרחשת פסיקה, ה-8051, כמו גם כל יורשיו, שומרים את המקום הבא, כלומר המקום של הפקודה הבאה לביצוע – המקום לחזור אליו, במחסנית.
  בסוף ביצוע הפקודות שנבעו כתוצאה מהקריאה או מהפסיקה, כשמופיעה הפקודה RET או RETI בהתאמה, המעבד עושה POP מהמחסנית, לכתובת שאליה הוא צריך לחזור, וקופץ לכתובת זו.
- כשמבצעים תת-שגרה, לפעמים רוצים לשמור מידע שנרצה לשחזר ממש לפני החזרה לתוכנית ממנה קראנו לתת-שגרה. דבר זה נפוץ בעיקר בתת-שגרה של פסיקה. אחת הדרכים לעשות את זה היא להשתמש במחסנית. כיוון שהמעבד מצפה למצוא את הכתובת אליה הוא צריך לחזור בראש המחסנית (בעת ביצוע ה-POP) כאשר מופיעה הפקודה RET או RETI, צריך לדאוג לעשות POP לכל הערכים שדחפנו אליה. אם לא נעשה את זה, המעבד יעשה POP למקום שהוא אינו הכתובת אליה הוא צריך לחזור!

פרק 9

פסיקות

9.1 הקדמה כללית

- טיפול בפסיקות ב-8051 הוא די פשוט. ברגע שהתרחשה פסיקה, ביט "דגל" אחד מודלק, המעיד על התרחשות פסיקה, והמעבד קופץ למקום המתאים ע"פ טבלת וקטורי הפסיקה.
  ברוב המקרים, אחרי שהמעבד מבצע את הקפיצה, הוא מאפס בעצמו את אותו ביט "הדגל" שהודלק, כיוון שהוא כבר בביצוע שגרת הפסיקה. אם לא יאופס ביט זה, בעת החזרה לתוכנית הראשית, תתבצע קפיצה נוספת לשגרת הפסיקה כיוון שכאילו "התרחשה פסיקה נוספת". לדוגמה, בעת התרחשות פסיקת External Interrupt Zero, המעבד מדליק את ביט IE0, קופץ למקום המתאים, שהוא 0003H, ואז מאפס את ביט IE0.
- אם פסיקה הופיעה בזמן שהמעבד מטפל בפסיקה אחרת, המעבד מדליק את ה"דגל" הרלוונטי כדי להזכיר לעצמו שהתרחשה פסיקה, אבל ממשיך לבצע את הפסיקה אותה הוא התחיל לבצע. ברגע שסיים לבצע אותה, יקפוץ לביצוע הפסיקה שה"דגל" שלה מודלק.

9.1.1 אפשור פסיקות

- כדי לאפשר פסיקות, צריכים להתבצע שני הדברים שהוסברו קודם. אפשור גלובלי, ואפשור ספציפי.

9.1.2 מעט עובדות לגבי הפינים של הפסיקות החיצוניות

- במקרה של פסיקות חיצוניות, External Interrupt, ישנו ביט נוסף שחייבים להגדירו.
  ביט זה נקרא IT0 – והוא קובע האם External Interrupt Zero הוא Edge triggered או Level triggered.
  אם הערך של ביט זה הוא 1 – אז ה- External Interrupt Zero הוא Edge triggered.
  אם הערך של ביט זה הוא 0 – אז ה- External Interrupt Zero הוא Level triggered.
- כיוון שפין ה- External Interrupt, (P3.2) הוא גם פין קלט/פלט, יש לוודא שפין זה מוגדר בצורה שתאפשר לו לשמש כקלט, ואז דרכו נוכל לגרום להתרחשות פסיקה. אם נכתוב אחד לפורט זה, זה לא ייאלץ ערך כלשהו על הפורט. כיוון שהוא מסתמך על נגדי הפול-אפ שישמרו על המתח גבוה הוא יאפשר למוצא לצוף. כדי להשתמש בפין זה כקלט, צריך לכתוב אחד לתוכו. את זה עושים על-ידי SETB P3.2.

Bounce 9.2

- כשסוגרים מתג, לרוב, המגעים של אותו המתג "קופצים" למשך זמן מסוים, (סדר גודל של מילי-שניות). דרך אחת להתמודדות עם בעיה כזו היא לבנות מעגל שמגיב לשינוי מסוים במתח, ומתעלם מכל שאר השינויים במתח שבאים נניח, 50 מילי-שניות לאחר מכן.

פרק 10

פסיקות טיימרים

10.1 הקדמה כללית

- לפעמים צריכים לעשות משהו כל פרק זמן מסוים. דוגמא לכך זו דגימה של אות בקצב מסוים. דוגמא אחרת היא קבלת נתון מפריפריה מסוימת, לדוגמא מקלדת.
- כדי לבצע משימות מסוג זה, מיקרו-מעבדים ומיקרו-בקרים לרוב באים עם פריפריות מובנות המאפשרות ביצוע פסיקה, כלומר הפרעה לפעולת המעבד, בקצב קבוע.
- הרכיבים האלו פועלים ע"י הגדלת האוגר שלהם באחד כל מחזור שעון, אבל בלי לבזבז את הזמן של המעבד. ברגע שהאוגר של הרכיב הזה "גולש", מתרחשת פסיקה.
- לא חייבים לתת לרכיבים האלו לפעול ע"י הגדלת האוגר שלהם באחד כל מחזור שעון. אפשר לגרום לרכיבים האלו להגדיל את האוגר באחד כל פעם שמתרחש אירוע מסוים, וכך למעשה אפשר למנות מספר אירועים שהתרחשו. (לדוגמה, כל פעם שמשתנה המתח על פין מסוים מגבוה לנמוך). שוב, ברגע שהאוגר שגדל כל פעם, גולש, מתרחשת פסיקה.
- מסיבה זו מתייחסים לטיימרים אלו לפעמים כטיימר ולפעמים כמונים. נתייחס אליהם בשם טיימרים, כדי לא לכתוב כל פעם טיימר/מונה. אם הוא טיימר אז הוא טיימר ואם הוא מונה אז שיהיה מונה. בכל אופן הוא עדיין טיימר.
- ל- ADuC841, כמו ל- 8052 שעליו הוא מבוסס, יש 3 טיימרים לשימוש כללי: Timer 0, Timer 1, Timer 2.
  Timer 1, Timer 2 – יכולים לשמש לקביעת ה- BAUD RATE של ה- UART.
  ל- ADuC841 יש טיימר רביעי נוסף לשימוש מיוחד שנקרא Timer 3, שיכול גם כן לשמש לקביעת ה- BAUD RATE של ה- UART. שזה בעצם בדיוק כמו Timer 1, Timer 2, ממש מיוחד.

10.2 שליטה בטיימרים

- שני SFRs, שנקראים TMOD ו- TCON, שולטים על טיימר 0 ועל טיימר 1. (ביחד, לא כל אחד שולט על טיימר).
- SFR נוסף, שנקרא T2CON, שולט על טיימר 2.

10.2.1 TMOD SFR

- הוא לא נגיש ביט-ביט. צריך לגשת אל כל האוגר (בייט שלם) גם אם רוצים לכתוב אפילו ביט אחד.
- ביט 7 – משמש כ- Gate לטיימר 1, ולא ברור מה זה אומר, לכן לא מעניין אותנו, לכן ביט זה צריך להיות אפס.
- ביט 6 – קובע האם טיימר 1 משמש כטיימר או כמונה.
  אם הוא 1 – מונה.
  אם הוא 0 – טיימר.
- ביט 5 – נקרא M1, קובע את מצב הפעולה של טיימר 1, ביחד עם ביט 4.
  ביט 4 – נקרא M0, קובע את מצב הפעולה של טיימר 1, ביחד עם ביט 5.
  מצב פעולה 1: ביט 5 מאופס, ביט 4 דלוק.

שני האוגרים (כל אחד 8 ביט) TH1 ו- TL1, מסודרים בצורה היוצרת אוגר של 16 ביט.

מצב פעולה 2: ביט 5 דלוק, ביט 4 מאופס.

האוגר TL1, הוא האוגר שמוגדל כל מחזור שעון, (או כל אירוע), ובזמן התרחשות הגלישה של אוגר זה, הוא מעודכן שוב עם הערך שיש בתוך TH1.

מצב זה נקרא מצב עדכון אוטומטי.

- ביט 3 - משמש כ- Gate לטיימר 0, ולא ברור מה זה אומר, לכן לא מעניין אותנו, לכן ביט זה צריך להיות אפס.
- ביט 2 - קובע האם טיימר 0 משמש כטיימר או כמונה.

  אם הוא 1 – מונה.

  אם הוא 0 – טיימר.
- ביט 1 - קובע את מצב הפעולה של טיימר 0, ביחד עם ביט 0.
- ביט 0 – קובע את מצב הפעולה של טיימר 0, ביחד עם ביט 1.

  מצב פעולה 1: ביט 1 מאופס, ביט 0 דלוק.

  שני האוגרים (כל אחד 8 ביט) TH0 ו- TL0, מסודרים בצורה היוצרת אוגר של 16 ביט.

  מצב פעולה 2: ביט 1 דלוק, ביט 0 מאופס.

  האוגר TL0, הוא האוגר שמוגדל כל מחזור שעון, (או כל אירוע), ובזמן התרחשות הגלישה של אוגר זה, הוא מעודכן שוב עם הערך שיש בתוך TH0.

  מצב זה נקרא מצב עדכון אוטומטי.

10.2.2 TCON SFR

- משמש לשליטה על טיימר 0 ועל טיימר 1.
- הוא כן נגיש ביט-ביט.
- ביט 7 – נקרא TF1, מודלק כשטיימר 1 גולש. מאופס אוטומטית ברגע שמתבצעת קפיצה לשגרת הפסיקה.
- ביט 6 – נקרא TR1, מודלק כאשר רוצים להדליק את טיימר 1. הוא מאופס כאשר רוצים לכבות את טיימר 1.
- ביט 5 – נקרא TF0, מודלק כשטיימר 0 גולש. מאופס אוטומטית ברגע שמתבצעת קפיצה לשגרת הפסיקה.
- ביט 4 – נקרא TR0, מודלק כאשר רוצים להדליק את טיימר 0. הוא מאופס כאשר רוצים לכבות את טיימר 0.
- ביט 3,2,1,0 – לא בשימוש ע"י הטיימרים.

10.2.3 הטיימר כפסיקה

- כפי שתיארנו, פסיקות טיימר מתרחשות בקצב קבוע באופן אוטומטי. אבל, כפי שכבר הסברנו, כמו עם כל סוג של פסיקה, על-מנת שהמעבד יוכל לטפל בפסיקה, פסיקות צריך להיות מאופשרות באופן גלובלי (ביט ה- EA), וגם באופן פרטני עבור כל פסיקה בה אנו מעוניינים שהמעבד יטפל.
- פסיקות של טיימר 1 יאופשרו ע"י הביט ET1.
- פסיקות של טיימר 0 יאופשרו ע"י הביט ET0.
- בנוסף, כמובן, צריך לדעת לאן קופץ המעבד בהתאם לכל סוג פסיקה.
- פסיקה של טיימר 0, תגרום למעבד לקפוץ לכתובת 000BH.
- פסיקה של טיימר 1, תגרום למעבד לקפוץ לכתובת 001BH.

10.3 איך מאתחלים טיימרים

- כאמור הטיימר, סופר מחזורי שעון, כל פעם גדל באחד, ואז גולש, ואז יש פסיקה. אבל איך עושים את זה?
- נגיד שאנחנו צריכים לעורר פסיקה 100 שניות, ונגיד שכל מחזור שעון לוקח 5 שניות. אז כמה מחזורי שעון צריך לספור, כדי שיעברו 100 שניות, ואז תתרחש בעצם הפסיקה הרצויה? התשובה היא 20.
  על אותו רעיון, נגיד שאנחנו רוצים לעורר פסיקה כל $1m[Sec]$, ע"י טיימר 0.
- תדר המעבד הוא $11.0592M[Hz]$.
- כל מחזור שעון הוא $\frac{1}{11.0592M}=0.0904\mu[Sec]$.
- לכן, צריך לספור $\frac{1m}{0.0904\mu}=11059.2$, מחזורי שעון. אי אפשר לספור מספר לא שלם, לכן נבחר 11059 לפני שתעבור בערך $1m[Sec]$.
- כיוון שאנו צריכים לספור 11059 מחזורי שעון, נצטרך לעבוד במצב פעולה 1. כיוון שהוא מאפשר ספירה עד $2^{16}=65,536$. מצב פעולה 2 לא יעיל כיוון שהוא מאפשר ספירה עד $2^8=256$.
- אנחנו צריכים לספור 11059 מחזורי שעון, לאחר מכן, שתתרחש גלישה, ותעורר הפסיקה. הפסיקה תתרחש כאשר האוגר שהוא בגודל 16 ביט יגלוש, כלומר כשנגיע לערך המקסימלי שלו. לכן, אנחנו צריכים להתחיל לספור החל מ- $2^{16}-11059=54,477$. שזה D4CDh (בהקס-דצימלי).
- האוגר TH0 יכיל את D4, בהקס-דצימלי.
- האוגר TL0 יכיל את CD, בהקס-דצימלי
- כיוון שזה מצב 1, ולא מצב 2 שהוא מצב עדכון אוטומטי, כל פעם שהאוגר (שבמצב זה הוא 16 ביט) יגלוש, ותתרחש פסיקה, ותתבצע קריאה לתת- שגרת הפסיקה (הפקודות לביצוע בעקבות הפסיקה) יש לאתחל את האוגרים האלו מחדש!
- אם היינו צריכים לספור מספר מחזורי שעון הקטן מ-256, היינו יכולים להשתמש במצב פעולה 2, שהוא מצב העדכון האוטומטי, ולהפטר מהצורך להטעין את האוגרים מחדש. ככה כל פעם שאוגר של 8 ביט גולש, מתרחשת פסיקה, והוא מעודכן אוטומטית עם אותו הערך ההתחלתי שהיה בו קודם, בעזרת האוגר השני.
- עדכון ידני של האוגרים, לא יניב פסיקות במרווחי זמן מדויקים. כדי להשיג דיוק מרבי, יש לבדוק את ערכו של האוגר, בדיוק לפני שמעדכנים אותו, ואז, חייבים להחסיר את אותה הכמות – ערכו של האוגר + הזמן שלוקח לבצע את פעולת החיסור – מהערך שאיתו אנו עומדים לעדכן את האוגר.

פרק 11

סביבת ה- IDE

11.1 הקדמה

לא רלוונטי

11.2 פרויקט

לא רלוונטי

11.3 כתיבת הפרויקט

- ה- IDE, כברירת מחדל, כולל את כל השמות, הפקודות, השמות לאוגרים והשמות ל- SFRs של ה- 8051. זה יעיל כיוון שלא צריך לכלול בתוכנית הגדרות לכל השמות של הרכיבים, אבל זה יעיל כל עוד משתמשים ברכיבים הקיימים ב- 8051.
- ה-8052, וה- ADuC841 מכילים את כל אותם רכיבים עם אותם השמות כמו ב- 8051, ועוד תוספות.
- אם רוצים להשתמש באותם הרכיבים המיוחדים ל-8052 או ל- ADuC841 , שכמובן לא נמצאים ב- 8051, חייבים להגדיר בתוכנית את השמות לכל אותם הרכיבים בהם משתמשים. הדרך הפשוטה ביותר לעשות את זה היא להוסיף בתוכנית את ההכוונה לקובץ MOD841, ע"י הוספת $INCLUDE MOD841 בתחילת התוכנית. כיוון שבדרך זו מגדירים מחדש את כל הקיים ב- ADuC841 (כלומר כולל מה שיש גם ב- 8051), חשוב לבטל ב- IDE את הגדרת אותם רכיבי ה- 8051 כברירת מחדל, ע"מ למנוע כפילות בהגדרות מה שיגרום לשגיאה.

11.4 DEBUGGING

- לאחר שהצלחנו לקמפל את התוכנית בלי לקבל הודעות שגיאה, הגיע הזמן להתחיל לתקן שגיאות שטמונות ביסודה של התוכנית, שגורמות לה להתבצע, אך באופן לא תקין. את זה עושים ע"י DEBUG.
- ה-Debugger יריץ את התוכנית, אם בבת אחת, ואם בקפיצות קטנות (breakpoints) שהגדרנו על-מנת לעקוב ביתר דיוק.
- הוא יראה למשתמש את ערכי האוגרים של ה- 8051, אוגרים של המערכת, יעדכן את המשתמש לגבי מספר מחזורי השעון שחלפו ומספר השניות שחלפו עד כה.
- ע"י חלון ה- Peripherals, אפשר לראות מה עושות כל אחת מהפריפריות על- גבי המעבד. ייתכן שה- A/D וה- D/A לא מוצגים בצורה הכי טובה, השאר מוצגים בצורה טובה.
- Watchlist – מאפשר מעקב אחרי המשתנים בתוכנית. ללכת ל- Watches. ואז Watch #1, ואז ללחוץ על מה שאומר "type F2 to edit", ללחוץ על F2, ואז לכתוב את שם המשתנה אחריו רוצים לעקוב. ה- IDE יציג את ערכו של המשתנה בזמן הביצוע של התוכנית.
- Logic Analyzer - משמש כמו Scope, צריך לפתוח אותו, להגיע ל- Setup, ללחוץ על Insert, ואז לכתוב את שם המשתנה שאנחנו רוצים להציג.

פרק 12

ה- UART

12.1 הקדמה כללית

- לעיתים קרובות צריך לגרום לשני מחשבים, שני מעבדים, לתקשר ביניהם. אחת הדרכים לעשות זאת היא בשימוש ה- UART, Universal Asynchronous Receiver Transmitter.
- פרמטר חשוב שבא לידי ביטוי כשמשתמשים ב- UART הוא ה- BAUD RATE. פרמטר זה, שהיחידות שלו הן סימבולים לשנייה, או במקרה של ה- UART אצלנו, ביטים לשנייה, מסמל את מספר הביטים שיכול ה- UART לשדר או לקלוט בשנייה אחת.
- כשניתנים ל- UART n ביטים לשידור, הוא תמיד יוסיף ביט אחד – ביט התחלה – בתחילת השידור. והוא תמיד יוסיף אחד, אחד וחצי או שני ביטים – ביט(י) סיום – בסוף השידור. הוא גם יכול להוסיף ביט זוגיות בין סוף הרצף לשידור לבין ביט(י) הסיום.
- אם משדרים 8 ביטים באמצעות ה- UART, עם BAUD RATE של 19200, ביט התחלה אחד, ביט סיום אחד, וללא ביט זוגיות, כמה זמן ייקח השידור? אז בסה"כ משדרים 10 ביטים, כל ביט משודר תוך $\frac{1}{19200}$ שניות, 10 הביטים ישודרו תוך פי 10 מזה, כלומר תוך $\frac{10}{19200} = 0.52[sec]$.

12.2 ה- UART

- ה- UART שעל ה- ADuC841 הוא פריפריה של ה- 8051, ויחסית קל לשליטה. החלק המסובך ביותר של השליטה בו, הוא קביעת ה- BAUD RATE. ישנן כמה דרכים לעשות את זה, אנחנו נתייחס רק לחלק.
- על ה- UART שולט אחד מה- SFRs, שנקרא SCON. הוא אוגר הנגיש ביט-ביט.

ביט 7 – נקרא גם SM0.

ביט 6 – נקרא גם SM1.

שני הביטים האלו משמשים לבחירת המצב.

אצלנו ביט 7 יאופס, ביט 6 יודלק.

מצב זה נקרא מצב 1, והוא מכניס את ה- UART לפעולה במצב של 8-ביט BAUD RATE.

ביט 5 – נקרא גם SM2. הוא ביט אפשור התקשורת של המולטי-מעבד. לא נתעסק איתו כי הוא נשמע מפחיד.

ביט 4 – נקרא גם REN. Serial port receive enable bit. חייב להיות דלוק כדי לאפשר קלט ב- Serial Port.

ביט 3 – נקרא TB8, לא משתמשים בו במצב 1, בו אנו עובדים.

ביט 2 – נקרא RB8, לא משתמשים בו במצב 1, בו אנו עובדים.

ביט 1 – נקרא TI, Serial port transmit interrupt flag. ביט זה מודלק ע"י ה- UART בכל פעם שהוא סיים לשדר בייט.

כיוון שיש רק פסיקת Serial port interrupt אחת, למעבד חייבת להיות דרך כלשהי להודיע למשתמש האם הפסיקה נבעה כתוצאה מסיום שידור של בייט מסוים, או כתוצאה מקלט של בייט מסוים. ביט זה וביט 0, מאפשרים לקבוע מה היה הגורם לפסיקה.

ביט 0 – נקרא RI, Serial port receive interrupt flag. ביט זה מודלק ע"י ה- UART בכל פעם שהוא סיים לקלוט בייט.

- יש לשים לב, שביט 1, וביט 0, לא מאופסים ע"י המעבד! לכן, חייבים לאפס אותם בזמן ביצוע שגרת הפסיקה.
- הדבר הבא שצריך לקבוע הוא ה- BAUD RATE. את ה- BAUD RATE קובעים ע"י גרימה לגלישה אצל טיימר 1, טיימר 2 או טיימר 3.
- כאשר משתמשים בטיימר 1, ה- BAUD RATE נקבע כך: $$\text{baud rate} = \frac{2^{SMOD}}{32} \times (\text{Timer 1 overflow rate})$$
כאשר SMOD זה הביט השביעי באוגר PCON.
כאשר PCON זה אחד מה- SFRs שאינו נגיש ביט-ביט.
כשמשתמשים בטיימר 1 כדי לקבוע את ה- BAUD RATE, צריכים לבטל פסיקות של טיימר 1. אנחנו לא רוצים להפריע לפעולת המעבד כל פעם שהאוגר של טיימר 1 "גולש", וכביכול אמורה להתרחש פסיקה.
בנוסף, כיוון שלא מתרחשות פסיקות (שאמורות להתרחש עקב אותה "גלישה", אבל ביטלנו אותן), למעשה לא תהיה הזדמנות לעדכן ידנית את ערכי האוגרים בתוך שגרת הפסיקה כפי שמתאים למצב 1, כיוון שבכלל לא מתרחשת פסיקה, ולא מבוצעת אף שגרת פסיקה! לכן חייבים לעבוד במצב 2 – מצב העדכון האוטומטי – כך שכל פעם שהאוגר של טיימר 1 "גולש" – אמנם לא מתרחשת פסיקה – אבל האוגר הסופר מעודכן אוטומטית ע"י האוגר השני.
- כדי לאפשר פסיקת ה- Serial port interrupt, צריך להדליק את ביט ES – Serial interrupt enable bit.
- כשמתרחשת פסיקת ה- Serial port interrupt, המעבד קופץ לכתובת 0023H.
- כדי לשדר בייט אחד, אחרי שה- UART הוגדר בהתאם לדרישות, צריך להעביר את המידע לאוגר SBUF.
- ה- UART יתחיל לשדר, מיד כשתתחיל הכתיבה לאוגר SBUF. (לכן צריך להגדיר את ה- UART לפני שמעבירים את המידע ל- SBUF הזה).
- כדי לקרוא מידע מה- UART, קוראים מתוך ה- SBUF.
- האוגר ממנו קוראים הוא לכאורה, אותו האוגר אליו כותבים. אבל במציאות, קוראים ממקום שונה לחלוטין מזה שאליו כתבנו. כיוון שאף פעם לא צריך לקרוא מהאוגר המשדר או לכתוב לאוגר הקולט, שני האוגרים כביכול "חולקים" אוגר אחד.

12.3 כתיבת מחרוזת בעזרת ה- UART

- כדי לכתוב מחרוזת ל- UART בלי לבזבז כמויות עצומות של זמן, אפשר להשתמש בפסיקת Transmit interrupt (TI).
- נגיד שאנחנו רוצים לשדר מחרוזת. נשדר את האות הראשונה של המחרוזת, ואז אומרים ל- UART, (ע"י תכנון שגרת פסיקה מתוחכמת), שכל פעם שהוא מסיים לשדר בייט אחד (אות אחת, סימבול אחד) – כל פעם שמתרחשת פסיקת TI, הוא צריך לשדר את הביט הבא. אחרי שידור הביט האחרון, צריכים להתעלם מפסיקת ה- TI הבאה.

פרק 13

שימוש ב- UART, הדפסה של מחרוזת מוכנה מראש

13.1 הקדמה כללית

- כדי לתקשר עם העולם החיצוני, לרוב אנחנו צריכים לשלוח הודעות. ברוב המקרים, להוציא יוצאי דופן בהם מה שנרצה לשדר הוא תלוי מה שקורה בתוכנית, אנו יודעים מראש מה נרצה לשדר ובאיזה עיתוי.
- כדי לשדר הודעה, מחרוזת מוכנה מראש, אנחנו נשמור את המחרוזת שלנו כסוג מחרוזת C-type null terminated string, כלומר לשמור כל מחרוזת אם 0 בסופה.
את המחרוזות נשמור בזיכרון התוכנית, לאחר מכן, נקרא את המחרוזת ונשלח אותה אל ה- UART.
- כדי לעשות את כל זה, נשתמש בכמה פקודות נוספות.
MOVC – משמשת לקריאת מידע מזיכרון התוכנית.
בנוסף, מחרוזות חייבות להישמר בקטע זיכרון הקוד. לכן נשתמש בהכוונת ה- DB.

13.2 הפקודות והטכניקות החדשות

13.2.1 קריאה מתוך זיכרון הקוד

- כדי לקרוא מידע מזיכרון הקוד משתמשים בפקודה MOVC. תחביר הפקודה הוא: MOVC A, @A+DPTR. הפקודה הזו טוענת את ה- Accumulator עם הערך הנמצא בכתובת המחושבת ע"י הסכום A+DPTR.
ה- DPTR, הוא Data pointer register. ה- Accumulator מצוין ע"י A.
ב-8051, 8052 בסיסי, ה- DPTR הוא אוגר של 16 ביט. ב- ADuC841 אוגר זה יכול לתפקד גם כאוגר של 24 ביט.
כברירת- מחדל הוא פועל כאוגר של 16 ביט. אנחנו לא נתעסק עם הפעולה שלו במצב 24 ביט.
הפקודה MOV DPTR, #(16 bit word), טוענת מילה בגודל 16 ביט אל 16 הביטים התחתונים של ה- DPTR.

13.2.2 כמה הכוונות נוספות של האסמבלר

- לפעמים אנחנו צריכים לשמור פרטים מסוימים או לשמור מקום עבור מידע מסוים בחלקים שונים של הזיכרון.
- לרוב, מידע אותו אנו רוצים לשמור יישמר בחלק זיכרון הקוד, ומקום למידע מסוים זמני, יישמר בחלק זיכרון המידע או בחלק הזיכרון הנגיש ביט-ביט.
- כדי לשמור מקום בזיכרון המידע, הראינו כבר שאומרים לאסמבלר לפנות לשם ע"י הכוונת DSEG. אפשר להוסיף לאסמבלר מידע לאחר הכוונה זו, כדי להגיד לו גם מאיפה להתחיל בחלק זיכרון זה. הראינו איך עושים את זה. את זה חשוב לעשות כיוון שהחלק הנמוך ביותר של זיכרון המידע לא אמור להיות בשימוש לזיכרון זמני ופשוט.
32 הבייטים הראשונים של זיכרון המידע מכילים את האוגרים לשימוש הכללי. (ארבעה בנקים של 8 אוגרים בכל אחד). 16 הבייטים הבאים הם זיכרון הנגיש ביט-ביט. לכן, מומלץ ביותר להשתמש בפקודה DSEG AT 30H. (16+32=48 בעשרוני שזה 30 בהקס-דצימלי). זה אומר לאסמבלר להתחיל מהכתובת הנ"ל, שהיא הבייט ה-49.

אחרי שאמרנו לאסמבלר איפה אנחנו רוצים לשמור מקום, אנחנו צריכים להגיד לו כמה מקום לשמור. את זה עושים ע"י הכוונת DS. אם נגיד נרצה להקצות בייט אחד ולפנות אליו בשם MAX, נכתוב: MAX DS 1.

- כדי לשמור מקום בזיכרון הקוד, צריך להשתמש בהכוונת ה- CSEG, ואז ב- DB. זה אומר לאסמבלר לשמור בזיכרון הקוד את כמות הבייטים שמופיעה לאחר ההכוונה. כמו קודם, ניתן לשייך להם גם שמות, בשביל הנוחות.
- נסתכל על הקוד הבא:

```
ORG  0100H
NAME:   DB      'This course is'; Here is the first part of the string.
        DB      ' great.'       ; Here is the second part.  Breaking
                                :it in two does not affect anything.
        DB      13              ; Carriage return.
        DB      10              ; Line feed.
        DB      0               ; 0--the infamous null-termination.
```

בהנחה שאנחנו כבר בזיכרון הקוד, השורה הראשונה קובעת לאסמבלר לזוז למיקום אחר, כעת בכתובת 100H. זה ישמור את המידע במקום שהוא יחסית רחוק מהתוכנית. (שגם שמורה בזיכרון הקוד).

השורה הבאה מייחסת את השם NAME למקום הנוכחי. מכיוון שהשתמשנו בסימן ' יחיד, האסמבלר יודע לשמור את תווי ה- ASCII שמרכיבים את המחרוזת אחד אחרי השני.

המספרים 13 ו-10 מסמלים Carriage return ו- Line feed , לא ברור לי מה זה אומר.

המספר 0, מסמל את סוף המחרוזת.

- כדי לשמור מקום בזיכרון הנגיש ביט-ביט, משתמשים בהכוונה לחלק זיכרון זה ע"י BSEG ואז שומרים מקום ע"י DBIT.

13.2.3 כתיבת מחרוזת אל ה- Serial Port

- כיוון שה- UART הרבה יותר איטי מהמעבד, לא צריכים לכתוב את כל המחרוזת בבת אחת.

מה שעושים זה כותבים ל- UART בייט אחד, ואז ממשיכים בביצוע הרגיל של התוכנית, ואז גורמים ל- UART לשדר את הבייט הבא, ברגע שהתרחשה פסיקת TI, כלומר שהוא סיים לשדר בייט. צריך לזכור לגרום לתוכנית הראשית להתעלם מפסיקות TI המופיעות לאחר שידור הבייט האחרון.

13.2.4 אילוץ של פסיקה

- לפעמים זה נוח "לעבוד" על המעבד ככה שיחשוב כאילו התרחשה פסיקה, למרות שזהו לא המצב.
- כדי לעשות את זה, צריך להדליק את ביט ה"דגל" של הפסיקה הרלוונטית.
- לדוגמא, כדי לגרום לפסיקת TI, צריך להדליק את ביט ה"דגל" של הפסיקה שנקרא ביט ה- TI.

פרק 14

טיימר 3

14.1 הקדמה

- ל- 8051 יש 2 טיימרים: טיימר 0 וטיימר 1.
- ל- 8052 יש 3 טיימרים: טיימר 0, טיימר 1 ועוד טיימר נוסף: טיימר 2.
- ל- ADuC841 יש 4 טיימרים: טיימר 0, טיימר 1, טיימר 2 וטיימר נוסף: טיימר 3.

טיימר זה מתוכנן לעשות דבר אחד ודבר אחד בלבד – לספק שעון ל- UART. (קביעת ה- BAUD RATE).
זה מאפשר פינוי של אחד מהטיימרים האחרים לביצוע פעולות אחרות, שכן אם לא היה טיימר 3, והיינו רוצים להשתמש ב- UART, אזי אחד מבין טיימר 1 או טיימר 2 (שגם יכולים לקבוע את ה- BAUD RATE) היו בשימוש.

14.2 שימוש בטיימר 3

- כדי להשתמש בטיימר 3 ע"מ לתזמן את ה- UART, מגדירים את ה- UART בדיוק כמו שראינו קודם.
  ההבדל היחיד הוא שבמקום לתת לטיימר 1 לקבוע את הקצב, נותנים לטיימר 3 לעשות את זה.
- שני אוגרים קשורים לפעולתו של טיימר 3. הראשון הוא T3CON והשני הוא T3FD.
- T3CON – לא נגיש ביט-ביט, ערך ברירת- המחדל שלו, הוא 0.

  ביט 7 – נקרא גם T3BAUDEN, מאפשר לטיימר 3 לספק את הקצב של ה- UART. צריך להדליק אותו כדי שטיימר 3 יספק את הקצב ל- UART.

  ביט 6,5,4,3 – שמורים, לא לגעת בהם.

  ביט 2 – DIV2

  ביט 1 – DIV1

  ביט 0 – DIV 0

  שולטים בגורם החלוקה.
- T3FD - מכיל מספר בין 0 ל-255.
- ה-BAUD RATE ניתן ע"י:

$$\text{baud rate} = \frac{2 \times f_{core}}{2^{DIV-1}(T3FD + 64)}.$$

Here f_{core} is generally 11.0592 MHz for our board. The recommended method of calculating the two numbers is to first calculate:

$$\texttt{DIV} = \text{floor}\left(\log_2\left(\frac{f_{core}}{16 \times Baud\ \ Rate}\right)\right).$$

Next one calculates T3FD according to the formula:

$$\texttt{T3FD} = \text{round}\left(\frac{2f_{core}}{2^{\texttt{DIV}-1} \times Baud\ \ Rate} - 64\right).$$

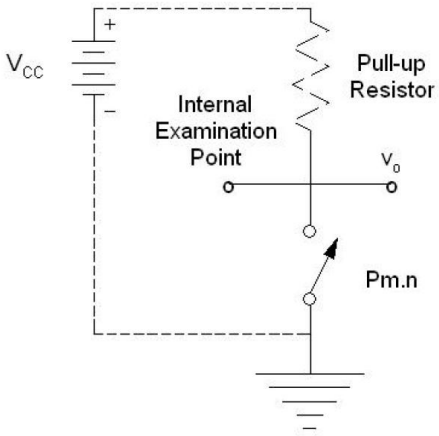
Finally the actual baud rate will be given by:

$$\text{Actual Baud Rate} = \frac{2f_{core}}{2^{\texttt{DIV}-1}(\texttt{T3FD} + 64)}.$$

פרק 15

Parallel Ports

- ל- 8051 ולמעבדים המבוססים עליו יש ארבעה פורטי I/O מקבילים: פורט 0, פורט 1, פורט 2 ופורט 3.
- פין n של פורט m נקרא Pm.n.
- כל פין של כל פורט הוא בעצם מתג עם איזשהו מעגל:



- הפינים של כל פורט נגישים באופן פרטני ע"י גישה לאוגרים הנגישים ביט-ביט P0,P1,P2,P3.
- כדי לגשת לביט השלישי של הפורט השלישי, כותבים P3.2. (הביט הראשון הוא ביט מס' אפס). לכן, ניתן להתייחס אליו כביט השלישי של הפורט השלישי או, הפין השני של הפורט השלישי.
- בעת עדכון ביט מסוים של אחד מהאוגרים הנ"ל, מתרחשים שינויים במעגל שגורמים לשינוי במצבו של הפין המסוים בפורט הרלוונטי.
- כל אחד מהפינים של הפורטים המקבילים 0, 2 או 3, יכול לשמש כקלט או כפלט.
- אצל פורט 1, זה לא ככה.
- איך הם משמשים כפלט? נסתכל על המעגל בציור. המוצא נמדד כמתח V_0. אם מזינים לתוך Pm.n אפס, אז המתג נסגר, ואז המתח V_0הוא אפס, ואז המוצא הוא אפס. אם מזינים לתוך Pm.n אחד, המתג נפתח, ואז המתח V_0 הוא המוצא.
- בפורטים 2 ו-3, מתח המוצא V_0 מחובר למתח V_{CC} דרך נגד. אם ההתנגדות היא גבוהה במיוחד, לא יזרום כמעט זרם דרך הנגד, ולכן המתח V_0 יהיה כמעט V_{CC}.
- V_{CC} הוא תלוי במקור המתח המסופק לרכיב.
- בפורט 0, החיבור לנגד ולמקור מתח אינם נמצאים. המתג פשוט "צף". לכן ניתן לאלץ תוצאה, כלומר, לאלץ מתח אפס ע"י כתיבת אפס לפין הרלוונטי, או לאפשר למתג "לצוף" ע"י כתיבת אחד לפין הרלוונטי. זה בא לידי שימוש אם רוצים לחבר את המעבד לרכיב שלא פועל על אותן רמות מתח כמו המעבד. במקרה כזה, מחברים מקור מתח ונגד אל פין הפלט.
נניח שמקור המתח החיצוני הוא 3.3 וולט, לכן כשכותבים אפס המתח מאולץ לאפס, וכאשר כותבים אחד, המתח קופץ ל-3.3 וולט ע"י מקור המתח והנגד החיצוניים.
לשים לב, שכל פין בפורט 0, יכול לפעול במתח שונה.
- פורט 1, הוא פורט לקלט בלבד, לכן כל הנ"ל לא נוגע אליו כלל. הוא יכול לשמש כקלט דיגיטלי, כאשר הערך אפס הוזן לפין הרלוונטי. או כקלט אנלוגי, כאשר הערך אחד הוזן לפין הרלוונטי. זה חוקי לחלוטין לגרום לחלק מהפינים לעבוד בצורה כזו ולאחרים בצורה אחרת.

- האוגרים P0,P2,P3 שולטים בפורטים 0,2 ו-3 בהתאמה. ראינו שאפשר לכתוב ערך מסוים לפין מסוים בפורט מסוים, ולקרוא ערך אחר לחלוטין, בדיוק מאותו הפין, איך זה יכול להיות? התשובה לכך טמונה בעובדה שכאשר כותבים אפס או אחד לפין מסוים בפורט מסוים, למעשה כותבים אל מין תא בזיכרון ששולט על פתיחה וסגירה של מתגים. כאשר קוראים את המתח על פין מסוים ע"י פקודה מסוימת כמו JB או JNB, זה גורם למעבד ממש לבדוק פיזית את המתח הקיים על הפין הספציפי הזה. אם המתח הזה תואם לרמה לוגית "1" אז הביט נחשב דלוק ואז ה- JB או ה- JNB רואים "1". אם המתח הזה תואם לרמה לוגית "0" אז הביט נחשב כבוי ואז ה- JB או ה- JNB רואים "0".
- לעומת זאת! יש פקודות מסוימות לקריאה מתוך פורטי I/O. הפקודות האלו קוראות את המצב המסוים של המתג! ולא את רמת המתח על הפין. כלומר, את הערך בו הזנו את הפין, (ודרכו גרמנו לפתיחה או לסגירה של המתג) ולא את המוצא שלו.

אחת מהפקודות האלו היא JBC, בצורה JBC bit, label. היא קוראת ערך של ביט מסוים קופצת אם הוא דלוק, ומאפסת אותו לפני שהיא ממשיכה לפקודה הבאה. כאשר משתמשים בפקודה זו כדי לקרוא ערך שקשור לפין של פורט מסוים, היא בודקת את הערך הנתון של המתג, ולא את המתח על אותו פין בפורט מסוים, (בניגוד לJB או JNB, שהסברנו קודם).

- כדי להשתמש בפורטי 0,2 ו-3 כקלט, הדבר הראשון שצריכים לעשות הוא לכתוב 1, לפינים הרלוונטיים. למה? כיוון שכשכותבים לפין כלשהו אפס (גם בפורט 2,3 וגם בפורט 0) אנחנו גורמים לכך שהמתח היחיד שנוכל לקרוא במוצא הוא אפס. בהנחה שכתבנו את הערך אחד, המתג הקשור לאותו הפין הוא פתוח, ואז אפשר לאלץ כל מתח שנרצה במוצא.
- כיוון שפורטים 0,2 ו-3 הינם פורטים דיגיטליים, רמות המתח המופעלים עליהן צריכים להיות או 0 וולט או V_{CC}.
- למעשה, ניתן לגרום בצורה כזו נזק למערכת. אם נלך בדרך בה כאשר כותבים אפס, אנו סוגרים את המתח ובכך מאלצים את המתח במוצא להיות אפס. וע"י כתיבת אחד אנו פותחים את המתג ובכך מאלצים במוצא מתח כלשהו. אם כך, כאשר נקצר את הפין פיזית לאדמה, ונכתוב לאותו פין את הערך אחד, למעשה גרמנו לפתיחה של המתג ולאילוץ מתח מסוים שאינו אפס, על פין שהמתח עליו חייב להיות אפס! זה יכול לגרום נזק. אפשר להתמודד עם זה ע"י נקיטת גישה שנקראת "החגורה והשלייקעס". כאשר מזינים אפס לפין מסוים המתג נסגר והמתח המאולץ במוצא הוא אפס. כאשר מזינים אחד, המתג לא נפתח אלא מאפשרים לו "לצוף", כך לא יכולים לגרום לנזק למערכת ע"י אילוץ של מתח על פין שכרגע מקוצר לאדמה.
- פורטים 2 ו-0 יכולים לשמש גם לדברים נוספים. אם יש כזה, הם יכולים לשלוט על זיכרון חיצוני. ופורט 1 יכול לשמש כקלט אנלוגי או דיגיטלי עבור ה- ADuC841.

פרק 16

פרק 19

ממיר דיגיטלי לאנלוגי

- ישנן הרבה דרכים ליישום ממיר דיגיטלי לאנלוגי.

There are many ways of implementing a digital to analog converter. In this lab we consider a very simple way that uses peripherals available on pretty nearly any microprocessor—including an ordinary 8051 or 8052. We consider a "one-pin" D/A.

The basic idea is to have one pin of one's microprocessor output a signal whose average value is proportional to the value of the digital signal one would like to output. Then one follows the pin with a low-pass filter—perhaps a simple RC circuit. The output of the low-pass filter—which performs a kind of windowed average of its input—should be proportional to the "instantaneous" average of the input to the low-pass.

The simplest way to implement such a system using a microprocessor is to have the microprocessor output a sequence of short pulses whose average voltage is the desired voltage. Typically one uses the output of an I/O pin. Thus the output is either (approximately) 5V or (approximately) 0V. One starts a new pulse every T_S seconds. Supposing that one one wants an output of V_{out} volts, one makes the width of the pulse $(V_{out}/5) * T_S$ seconds. (Modulating the width of a sequence of pulses in this way is called *pulse width modulation*—PWM.) One passes this signal through a low-pass filter whose cutoff frequency is greater than the frequency of the analog signal to be output but is substantially smaller than the frequency of the pulses being used.

Suppose, for example, that one wanted to output 2V for 0.1s and 3V for the next 0.1s. One way to do this would be to have a pulse generator generate 5V pulses every 0.1ms. If for the first 100 ms the pulses were of width 0.04 ms, for the next 100 ms the pulses were of width 0.06 ms, and

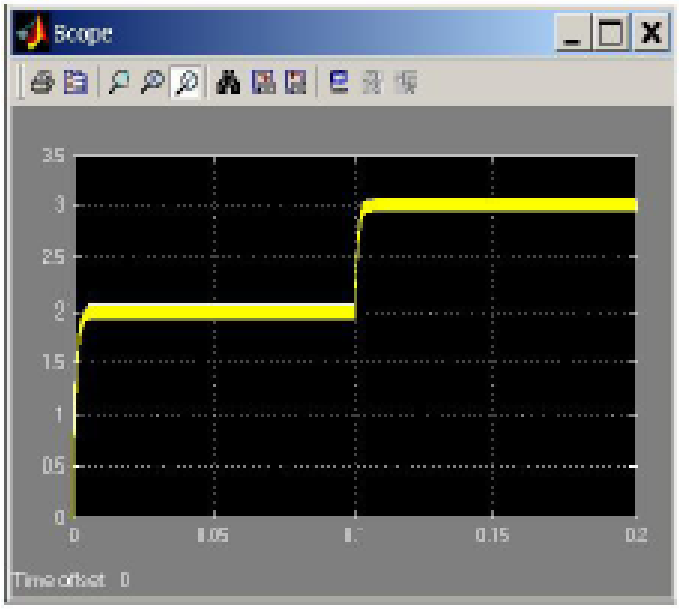


Figure 19.1: The Filtered Output

if one passed the pulses through a low-pass filter whose time constant was considerably larger than $0.1ms$, then the output of the filter should be just the voltages desired. We implemented such a system using Simulink. The output of the filter is given in Figure 19.1. Note that at $t = 0.1s$ there is a transition region. The output of the filter cannot change instantaneously from one value to another—the filter is a low-pass filter. Also, note that the 2V and 3V lines have a certain "thickness." This is because the low-pass filter does not perfectly average its input. When using this type of D/A, one must be prepared (and able) to sacrifice accuracy for ease of implementation.

19.2 The Underlying Theory

Let us examine how a simple RC low-pass filter works. If $i(t)$ is the current flowing through the low-pass filter, then it is easy to see that the voltage drop across the circuit satisfies:

$$Ri(t) + \frac{\int_0^t i(\tau)\,d\tau}{C} = v_{\text{in}}(t).$$

If we set:

$$w(t) \equiv \int_0^t i(\tau)\,d\tau,$$

then we find that $w(t)$ satisfies the equation:

$$Rw'(t) + w(t)/C = v_{in}(t), \qquad w(0) = 0.$$

Solving this equation using the method of variation of parameters, we find that:

$$w(t) = \frac{1}{R}\int_0^t e^{-(t-y)/RC} v_{in}(y)\, dy$$

and we find that the voltage across the capacitor—the output of the filter—satisfies:

$$v_o(t) = \frac{1}{RC}\int_0^t e^{-(t-y)/RC} v_{in}(y)\, dy.$$

If we treat the exponential as having "ended" after three time constants, then we see that the output is the average of the input over (approximately) three time constants. Thus, if one makes the frequency of the AC parts of the signal large relative to the circuits time constant, and if the waveform being output to the DAC also changes slowly relative to the circuits time constant, this method ought to give an output that is proportional to the desired voltage.