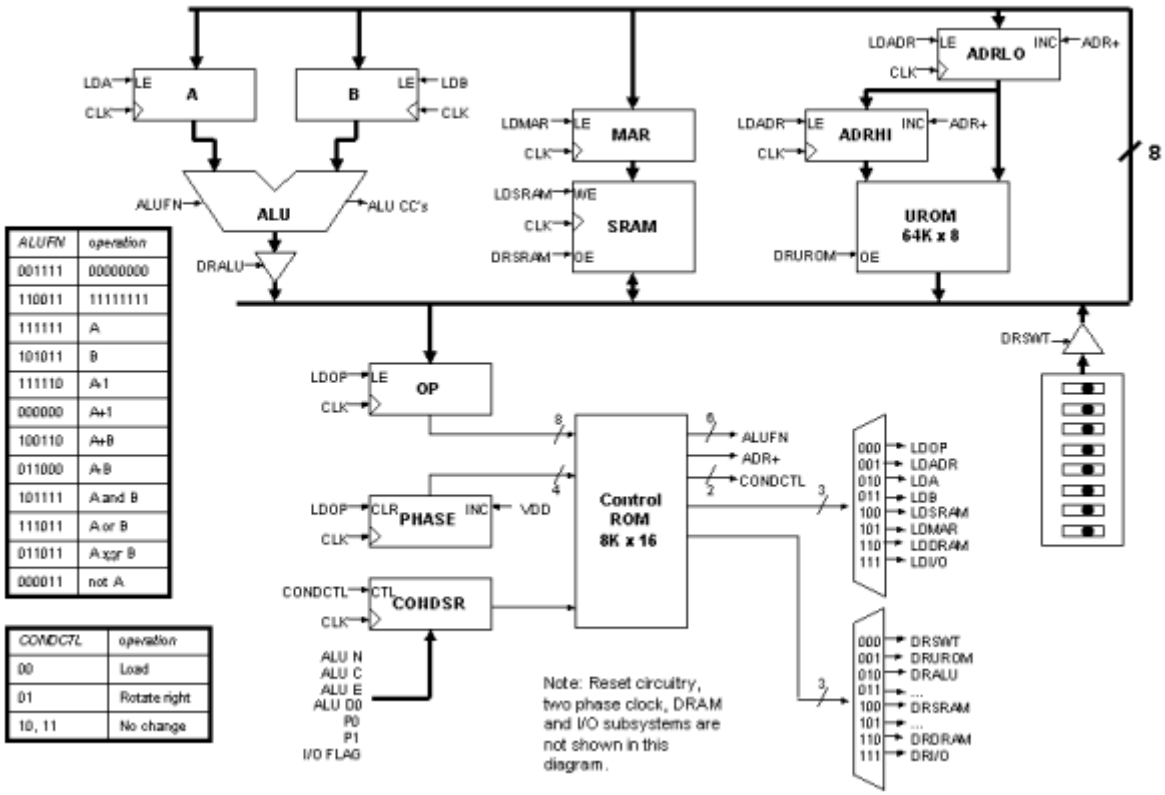
תכן לוגי ומבוא למחשבים

ניר אדר



ALUFN	operation
001111	00000000
110011	11111111
111111	A
101011	B
111110	A-1
000000	A+1
100110	A+B
011000	A-B
101111	A and B
111011	A or B
011011	A xor B
000011	not A

CONDCTL	operation
00	Load
01	Rotate right
10, 11	No change

תכן לוגי ומבוא למחשבים - מהדורה 1

החוברת נכתבה בהתאם לתוכנית הלימוד של הקורס "תכן לוגי ומבוא למחשבים" בטכניון – הגרסה הישנה של הקורס (הגרסה שלומדה עד 2003). זו איננה חוברת רשמית של הטכניון אלא חוברת פרטית שנכתבה על ידי **ניר אדר** והתבססה על ההרצאות, על ספרי הלימוד ועל מקורות נוספים המצויינים ברשימת המקורות.

המחבר איננו אחראי לכל נזק, ישיר או עקיף, שיגרם עקב השימוש במידע המופיע בחוברת, וכן לנכונות התוכן של הנושאים המופיעים בה. עם זאת, המחבר עשה את מירב המאמצים כדי לספק את המידע המדויק והמלא ביותר.

כמו כן, המחבר איננו אחראי על שינויים שיכולו בתוכן הקורס.

<u>תודות</u>

צוות הקורס:

- **ד"ר מיכאל ורנר**, תודה מיוחדת על שעות של הסברים וכן על הערות על החוברת עצמה.
- **ליאור קהתי**, מתרגל שללא שעות הקבלה הרבות בהן עזר לי – חוברת זו לא היתה קיימת.
- **פרופסור רן גינוסר** – על ביקורת וקריאה מעמיקה של חוברת זו.

סטודנטים בקורס:

- **עומר סלע**, **איל רוטמן** שתרמו, העירו והוסיפו לחוברת זו.

ניר אדר
יוני 2003

רשימת מקורות

1. "רישומי הרצאות של מיכאל ורנר", חורף 2002, ניר אדר
2. "רישומי הרצאות תכן לוגי", חורף 1999, מאת המכון לקידום החירש בישראל
3. "רישומי תרגולים של ליאור קהתי", חורף 2002, ניר אדר
4. Ward & Halstead, "Computation Structures", MIT Press
5. "שאלות ותשובות בנושא pipeline", אוניברסיטת MIT http://6004.lcs.mit.edu/currentsemester/tutprobs/pipeline.htm
6. "שאלות ותשובות בנושא מכונות מתוכנתות", אוניברסיטת MIT http://6004.lcs.mit.edu/currentsemester/tutprobs/progmach.htm
7. "שאלות ותשובות בנושא מטא-סטביליות", אוניברסיטת MIT http://6004.lcs.mit.edu/currentsemester/tutprobs/synchronization.htm
8. "Basic DRAM operation", מאת Tom's Hardware http://www.tomshardware.com/mainboard/98q4/981024/ram-01.html

הקדמה

ייצוג מספרים בבסיסים שונים

ייצוג מספרים

ייצוג מספרים בבסיס m נעשה ע"י הספרות 0..(m-1), כשהבסיס גדול מ-10 משתמשים באותיות ABC....
הערך המספרי הוא:

$$\sum_{i=0}^{n-1} d_i b^i = (d_{n-1}...d_0)_b$$

מעבר מבסיס לבסיס

שיטה 1: סכום וכפל
השיטה מומלצת כשבסיס המטרה הוא 10.

$$(\sum_{i=0}^{n-1} d_i b^i)_c$$

החישוב הוא בבסיס c.
דוגמא:

$$100 = 0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 = 4$$

שיטה 2: שיטת החלוקה
השיטה מומלצת כשבסיס המקור הוא 10.
נתונה $(...)_c \leftarrow (d_{n-1}...d_0)$.

מחלקים את מחרוזת הייצוג בערך של בסיס המטרה המיוצג לפי בסיס המקור.
השארית תהווה את ספרותיה של התוצאה.
דוגמא:
נעביר את המספר העשרוני 167 לבסיס אוקטלי:

247_8	=	167_{10}
7	7	20
47	4	2
247	2	0
247 היא המספר בבסיס 8		0 הוא השלם אחרי חילוק ב8

מעבר מבסיס לבסיס כאשר אחד הוא חזקה של השני:

$$726_8 = 111\ 010\ 110_2$$

כל סיפרה בבסיס 8 הפכה ל-3 ספרות בבסיס 2. אם נרצה לעבור מבסיס 2 ל-8 פשוט נקבץ את הספרות.

מספרים בעלי סימן

שיטת המשלים ל-2:

השיטה: הופכים את הביטים ומחברים 1
2Comp(x)
return (1Comp(x)+1)
טווח:

$$-(2^{n-1})\ldots(2^{n-1}-1)$$

דוגמא:
4=0100
-4=1011+1=1100

חיבור וחיסור:

0111 +	7 +
1010 =	-6 =
~~1~~0001	1

התעלמנו מה-carry בחיבור.

1001 +	-7 +
1001 =	-7 =
~~1~~0010	2

גלישה – כאשר מצפים לתוצאה מסימן כלשהו ומקבלים תוצאה מסימן הפוך.

אלגוריתם הבנת מספר:
1. אם הביט השמאלי הוא 0, זהו מספר חיובי.
2. אם הביט השמאלי הוא 1, חסר 1 מהמספר, הפוך את הביטים וזכור שהוא שלילי.

מושגי יסוד

הפשטה ((Abstraction)

הסתרת פרטי מימוש שאינם הכרחיים לניתוח פעולת המערכת.

משטר (Discipline)

מערכת חוקים המונהגים כדי לפשט את התכן ותקפים ברמת הפשטה מסוימת.

מתח תקף (Valid Voltage)

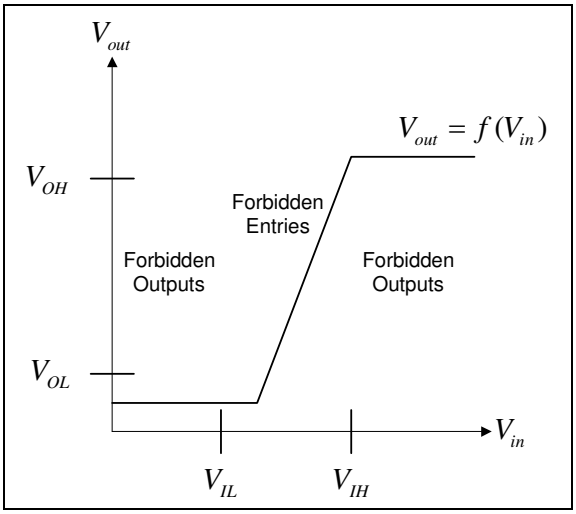
תחום מתח המוגדר כמייצג מצב לוגי (0,1). התחום מוגדר בנפרד עבור הכניסות ועבור היציאות.

תחום אסור (Forbidden Zone)

התחום בין מתחי הכניסה התקפים.
אינו מייצג מצב ולאות מותר להימצא בו רק בזמנים מעבר קצרים.

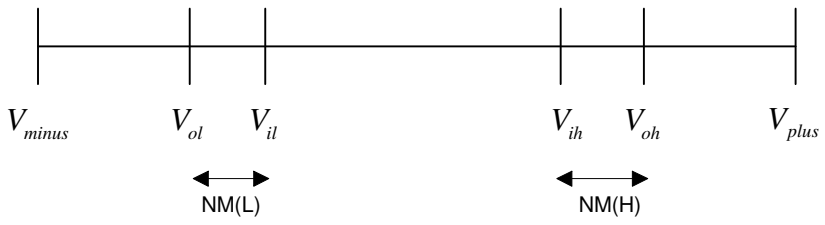
שולי רעש (Noise Margin)

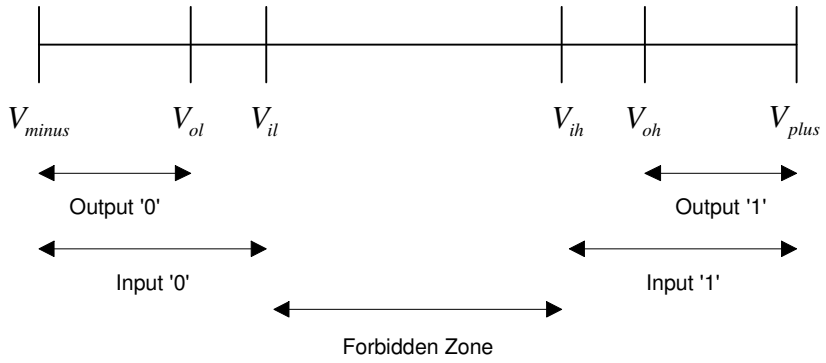
התחום שהוא מתח תקף בכניסה אך איננו מתח תקף ביציאה. תחום זה נועד להבטיח פעולה תקינה בנוכחות רעשים.
רעש הוא שינוי באות החשמלי במעבר בחוטים המקשרים בין הכניסות אל היציאות.



עבור כל רכיב חשמלי מוגדרת 4 רמות המתחים הבאות:

- V_{il} - מתח הכניסה המקסימלי שחייב להיות מתורגם ל-0 לוגי.
- V_{ol} - מתח היציאה המקסימלי המותר לצורך ייצוג 0 לוגי.
- V_{ih} - מתח הכניסה המינימלי שחייב להיתרגם ל-1 לוגי.
- V_{oh} - מתח הכניסה המינימלי המותר לייצוג 1 לוגי.





שולי רעש הוא פרמטר הקובע את הרעש המרבי שניתן לסבול בכניסת שער מסוים תוך כדי שמירה על יציאתו במצב תקף.
הבעיה ששולי רעש באים לפתור היא המקרה בו בתחילה המתח היה תקף אבל בסיום מסלולו לא. לכן נגדיר שולי רעש, תחום שהאות בו עדיין תקף, אך לא נשלח אותות חדשים בתחום זה.

מגדירים שולי רעש גבוה, עבור היציאה ב-"1": $NM_H = V_{OH} - V_{IH}$.

מגדירים שולי רעש נמוך, עבור היציאה ב-"0": $NM_L = V_{IL} - V_{OL}$.

מגדירים שולי רעש של שער, בצורה הבאה: $NM = Min(NM_L, NM_H)$

- קביעת $V_{IL\max}, V_{IH\min}$ והתחום האסור

נהוג לחפש באופיין השער (גרף מתח היציאה כפונקציה של מתח הכניסה) את שתי הנקודות המקיימות $\left|\frac{dVout}{dVin}\right| = 1$ או את שתי הנקודות הראשונות מחוץ לתחום $\left|\frac{dVout}{dVin}\right| > 1$. נקודות אלו יתנו את $(V_{IH\min}, V_{OL\max}), (V_{IL\max}, V_{OH\min})$.

- מתקיים:

$$V_{IH\min} \le V_{IH} \le V_{OH} \le V_{DD}$$
$$0 \le V_{OL} \le V_{IL} \le V_{IL\max}$$

(כאשר V_{DD} מתח מקסימלי באופיין).

- ככל ש NM_L, NM_H קרובים יותר, נקבל NM טובים יותר לשער.

זמנים

<u>זמן עלייה</u>: כמה זמן לוקח לאות לעלות מ'0' לוגי ל'1' לוגי. מוגדר כזמן שלוקח לעבור מ10% מהמתח המייצג 1 לוגי ל90% מהמתח המייצג 1 לוגי. מסומן ב t_r.

<u>זמן ירידה</u>: כמה זמן לוקח לרדת מ'1' לוגי ל'0' לוגי. מוגדר כזמן שלוקח לעבור מ90% מהמתח המייצג 1 לוגי ל10% מהמתח המייצג 1 לוגי. מסומן ב t_f.

tpLH: זמן המעבר בין שינוי כניסה לשינוי היציאה. מוגדר כזמן שלוקח מהרגע שהכניסה ב50% מערכה החדש עד הרגע שהיציאה ב50% מערכה החדש (1 לוגי).

tpHL: זמן המעבר בין שינוי כניסה לשינוי היציאה. מוגדר כזמן שלוקח מהרגע שהכניסה ב50% מערכה החדש עד הרגע שהיציאה ב50% מערכה החדש (0 לוגי).

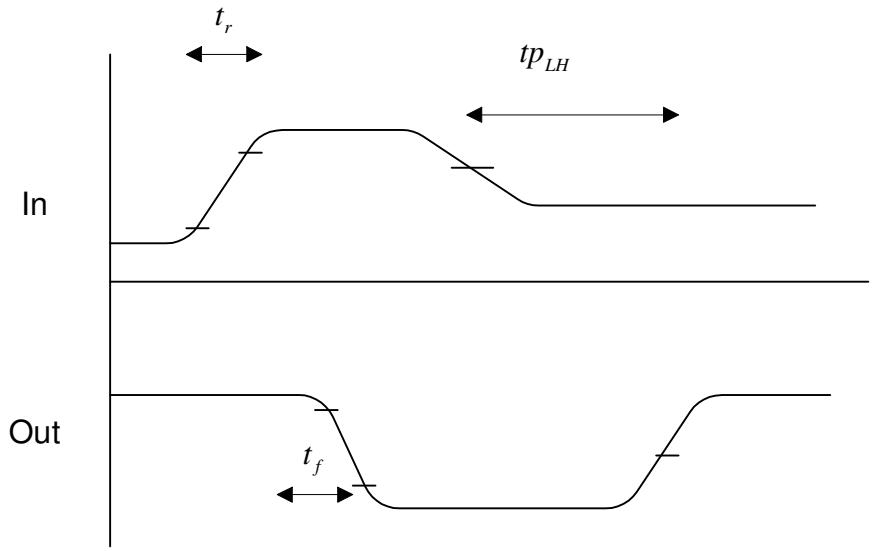
tpLH, tpHL נקראים גם tpd (Time Propagation Delay).

אם יש לנו מספר רכיבים, מחברים את הtpd של הרכיבים השונים.

tcd - Contamination Delay: זמן בו היציאות עדיין נשארות בערך הקודם שלהן לאחר שהשתנה המבוא. בדרך כלל מניחים tcd=0.

ts - Tsetup: הכניסות לרכיב צריכות להיות חוקיות לפחות ts לפני עליית השעון.

th - Thold: הכניסות לרכיב צריכות להישאר חוקיות לפחות th לאחר עליית השעון.



משטר סטטי (Static Discipline)

מעגלים צריכים לקיים משטר סטטי, האומר כי כאשר מתחי הכניסות תקפים, ההתקן הלוגי יפיק (לאחר התייצבות בזמן Tpd) מתחי יציאה תקפים.

התקן צירופים (Combinational Device)

- כניסות ויציאות דיסקרטיות ומקבלים ערכים של 0 או 1.
- פונקציות לוגיות מגדירות יציאות (בעזרת טבלת אמת).
- קיימים זמני שיהוי.
- אין חוגים בחיבורי הרכיבים. (חוג: מעגל בין יציאה של רכיב לכניסה של אותו רכיב).

מערכת צירופית

- כל התקן צירופי הוא גם מערכת צירופית.
- חיבורים בין מספר התקנים צירופיים על ידי החוקים הבאים:

1. מותר לחבר מוצא של התקן אחד למבוא אחד או יותר של התקנים אחרים.
2. אין לחבר מוצאים של רכיבים.
3. כל מסלול במערכת העובר ממבוא למוצא עובר דרך כל נקודה רק פעם אחת.

- בפועל מעגלים צירופיים לא מעשיים.

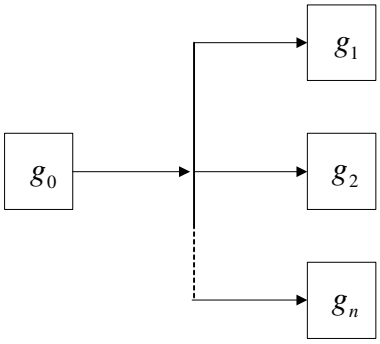
חוברת הרכיבים, משפחות לוגיות

חוברת רכיבים מכילה דפי נתונים עבור רכיבים שונים.
את הרכיבים מחלקים למספר "משפחות לוגיות" הנבדלות בטכנולוגיה בעזרתן הן מיוצרות. המשפחות העיקריות איתן עובדים בתכן לוגי הן TTL וCMOS.
באופן כללי רכיבי הCMOS חסכוניים יותר מבחינת צריכת זרם, אולם הם איטיים יותר מרכיבי TTL.
בד"כ יחידות צירופיות יורכבו על ידי TTL ואילו יחידות עתירות זיכרון ימומשו בדרך כלל על ידי CMOS.
עבור כל משפחה יתכן ותהיינה קיימות מספר תת משפחות.

Fan-Out

Fan-Out מוגדר כמספר המקסימלי של כניסות לוגיות שניתן לחבר ליציאת שער לוגי אחד.
השערים הינם רכיבים פיסיקליים ולכן הם צורכים זרם בכניסתם, והם מוגבלים בכמות הזרם שהם מסוגלים לדחוף ביציאתם.
על מנת שמעגל יתפקד בביטחון, יש להקפיד על כך שסכום הזרמים הנכנסים לכניסות השערים $g_1, g_2, ..., g_n$ יהיה קטן מזרם היציאה שיכול השער g_0 לספק.
כלומר, יש לבדוק כי התנאי הבא מתקיים:

$$I_0(g_0)_{\min} \geq \sum_{k=1}^{n} I_i(k)_{\max}$$



יש לבדוק תנאי זה הן עבור מוצא g_0 ב'1' לוגי והן עבור מוצא g_0 ב'0' לוגי.

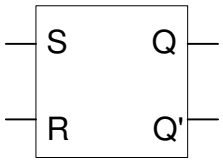
סימונים בדפי נתונים

$I_{IL} = I_{\text{input}}, \max['0']$ $\quad I_{OL} = I_{\text{output}}, \min['0']$
$I_{IH} = I_{\text{input}}, \max['1']$ $\quad I_{OH} = I_{\text{output}}, \min['1']$

התקן עקיבה מתוזמן (Clocked Sequential Device)

- כניסה אחת או יותר, עם ערכים דיסקרטיים.
- כניסת שעון הקובעת מתי מתבצעים מעברי המצבים.
- יציאה אחת או יותר, עם ערכים דיסקרטיים.
- מספר סופי של מצבים דיסקרטיים $S_1,...,S_k$.
- קריטריון למצב הבא, עבור כל מצב נוכחי וכל צירוף כניסות.
- הגדרת יציאות, עבור כל מצב נוכחי וכל צירוף כניסות.
- מפרט זימוני כניסות, הכולל לפחות את T_{setup}, T_{hold}.
- מפרט זימון יציאות, הכולל חסם עליון לזמן הדרוש ליציאות להתייצב אחרי מעבר מצב (T_{pd}) וחסם תחתון לזמן שהיציאות נשארות תקפות לאחר מעבר מצב (T_{cd}).

Set-Reset Latch

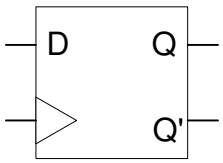


רכיב זה הוא רכיב זיכרון.
Latch הוא רכיב שהכניסות רשאיות להשתנות בכל עת ושהיציאות מושפעות מיידית מהכניסות.

טבלת עירור עבור Set-Reset Latch

R	S	y(t+1)
0	0	y(t)
0	1	1
1	0	0
1	1	?

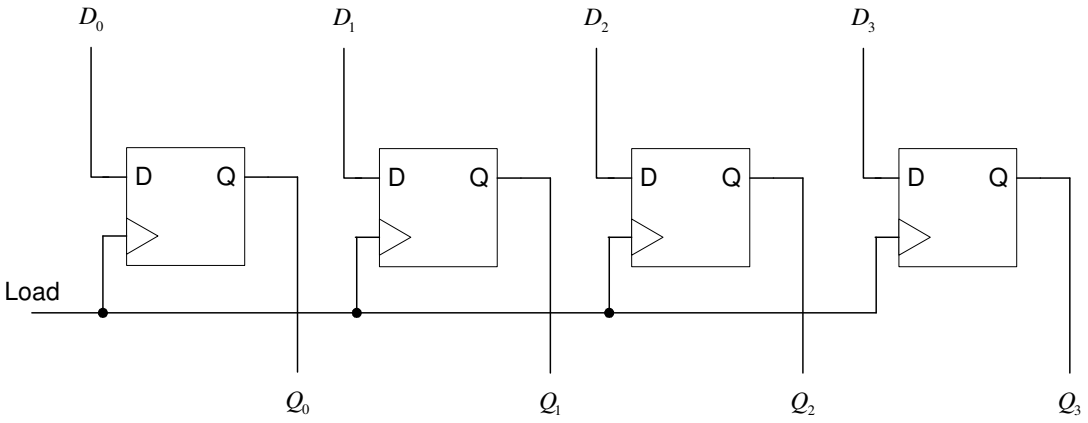
D Flip Flop



כאשר השעון עולה, Q מקבל את הערך של D, ונשאר קבוע עד העלייה הבאה.
הרכיב מעביר את הנתון הנכנס אל היציאה.

רגיסטרים

ניתן לממש בעזרת D-Flip Flop אוגרים (רגיסטרים) המאפשרים קריאה וכתיבה בו זמנית. נביט במימוש הבא:
בכל פולס שעון, סידרה של ארבעה ערכים D_1, D_2, D_3, D_4 נכנסים אל הרכיבים.



המשטר הדינמי (The Dynamic Discipline)

- המעגל יורכב מהתקני עקיבה מתוזמנים ומהתקני צירופים בלבד.
- המעגל יהיה נקי מחוגי צירופים: בכל מסלול סגור צריך להיכלל לפחות רגיסטר אחד.
- השעון המוזן להתקנים המתוזמנים וקובע את מעברי המצבים, יהיה בעל זמן מחזור מספיק ארוך כדי לאפשר לכל הכניסות להתייצב ולקיים את דרישות T_{setup} לפני מעבר המצב הבא.
- בכל יתר הכניסות להתקנים המתוזמנים יסופקו רמות לוגיות תקפות ויציבות לתקופת זמן סביב מעבר המצב, בהתאם למפרט זימון הכניסות להתקנים.

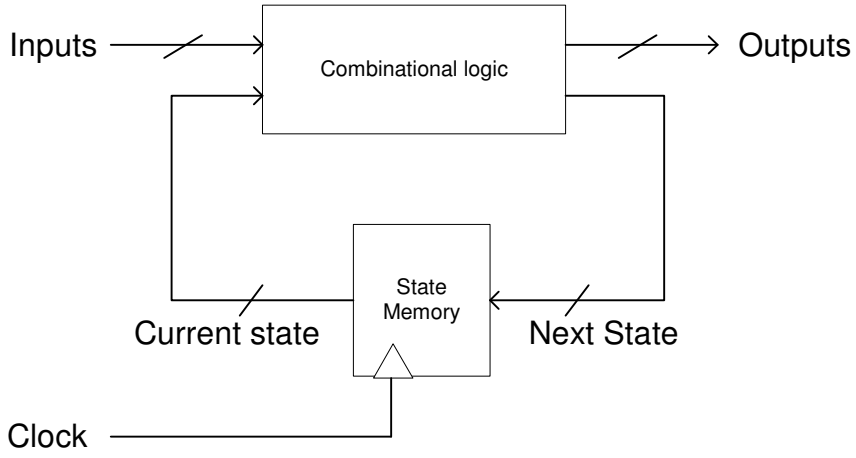
מעגל עקיבה מתוזמן

מערכת עקיבה זוהי מערכת צירופית שהוספנו לה זיכרון. ערכי הפלט תלויים בערכי הקלט וכן בתוכן הזיכרון של המערכת.

דוגמא להמחשה

נתונים שני מנעולים.
האחד הוא מנעול צירופי: צירוף מסוים של מספרים פותח אותו.
סוג המנעול השני הוא מנעול קומבינציה. חייבם לשמור על סדר מסוים של המספרים על מנת שהמנעול יפתח.
סוגי היסטוריה למערכת, במקרה שיש 3 ספרות לצופן:
מצב A - מצב התחלתי, לא הופיע עדיין אף מספר נכון.
מצב B - נצפתה הספרה הראשונה הנכונה.
מצב C - הופיעה כבר הספרה השניה בצופן.

מעגל עקיבה



תנאים לפעילות תקינה של המעגל

א. מחזור השעון מקיים את התנאי: $\text{CLOCK CYCLE} \equiv T > T_{PC\text{-}Q} + T_{pd} + T_{setup}$

ב. כניסות תקפות ויציבות סביב שפת השעון הפעילה, כדי להבטיח כניסות יציבות לרגיסטר. (Ts, Tpd).

ג. Tcd סביב החוג גדול מTh של הרגיסטר. $T_{cd} > T_h$.

במקרה הכללי של מעגלים מורכבים יותר, התנאים צריכים להתקיים עבור כל חוג ובין כל שני רגיסטרים במעגל.

כמן כן ידוע כי T_{cd} הוא הזמן המינימלי בו המוצא שומר על ערכו הקודם אחרי clk ואילו T_{pd} זהו הזמן המקסימלי בו המוצא שומר על ערכו הקודם אחרי clk, ולכן לא יכול להתקיים אף פעם כי $T_{cd} > T_{pd}$.

***f*max**

זהו תדר השעון המקסימלי המותר כך שהמערכת תתפקד נכון.
נפרק את המערכת לשני חלקים: זיכרון ולוגיקה צירופית.
מעבר אקטיבי של השעון גורם לזיכרון לשנות מצב, שינוי המצב "עובר" דרך הלוגיקה וחוזר לכניסות הזיכרון (ככניסות למצב הבא). מכאן שיש זמן מינימלי שיש לחכות בין שני מעברים אקטיביים עוקבים.

מחשבים מהו Tmin - זמן המחזור המינימלי כך שהמערכת תתפקד נכון.
לTmin שלושה חלקים:

א. זמן ההשהיה של הזיכרון מהמעבר האקטיבי עד להתייצבות יציאותיו. (Tpd)

ב. זמן ההתפשטות של המצב החדש דרך הלוגיקה עד לכניסות הזיכרון.

ג. זמן הSetup של הזיכרון.

כלומר מתחילים ממצב התחלתי נתון, כעת מתקבל מעבר אקטיבי של השעון, ושואלים מתי ניתן לתת את המעבר האקטיבי הבא. על מנת למצוא את Tmin יש לבצע את התהליך הנ"ל לכל המצבים של המערכים, אם כי בד"כ ניתן לראות מהם המצבים ההתחלתיים שיתנו Tmin גדול ולבדוק רק אותם.
נסכם:

$$T_{cyc} \geq T_{pd}(clk \to Q) + T_{pd}(CL) + T_{setup}(MEM)$$

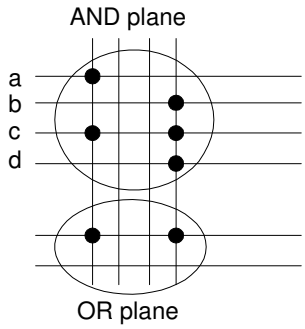
לאחר שחישבנו את Tmin, נוכל למצוא את fmax. $f_{\max} = \frac{1}{T_{\min}}$.

Clock skew

המשטר הדינמי מסוג SGT (synchronous globally timed) מחייב הזנת שעון יחיד בו זמנית לכל הרכיבים המתוזמנים במעגל.
אבל, במקרים רבים נתקלים בקושי בשל מגבלות פיסיקליות, והשעון מגיע בהפרשי זמן שונים לחלקים שונים במעגל.
הפרש הזמן המקסימלי נקרא עיוות שעון (Clock skew).
בדרך כלל נרצה לחשב מהו ה-skew המקסימלי המותר במעגל.
נבדוק את כל המסלולים בין ה-FF השונים, ונראה שאנו עומדים בדרישות T_{hold} שלהם.
על מנת לקיים את המשטר הדינמי, נדרוש: $\sum T_{cd} > T_h + skew$
טעות נפוצה כאשר בודקים מהו ה-skew היא לפספס מסלולים בעלי skew נמוך יותר מהמסלולים אותם בדקנו. בסופו של דבר נבחר את ה-skew המינימלי מבין המסלולים השונים שנמצא.

PAL - Programmed Array Logic

ניתן לייצג פונקציה לוגית בעזרת שני מישורים: מישור AND ומישור OR.



$$y = ac + bcd$$

ברכיבי PAL כל יציאה היא שער OR שכניסותיו הן מספר מכפלות (כלומר מוצא שערי AND) הניתנות לתכנות. חלק מרכיבי ה-PAL מכילים Flip Flops פנימיים שדוגמים את יציאות ה-OR ואז ניתן לממש בעזרתן לא רק פונקציות צירופיות אלא גם מכונות מצבים.

תכן pipeline

שאלה:

נדרש לבנות את המערכת המחשבת את הפונקציה: $f(x) = 243 \cdot x$
לצורך כך יש להשתמש ברכיב הצירופי הבא:

$T_{PD} = 50ns$

<u>סעיף א':</u>

נדרש לממש את הפונקציה כמערכת צירופית.
יש לחשב את זמן המחזור המינימלי בו ניתן להפעיל את המערכת.

<u>פתרון:</u>

נשים לב לעובדה הבאה: $f(x) = 243 \cdot x = 3^5 \cdot x$

המערכת הצירופית:

<u>חישוב זמן המחזור המינימלי:</u>

זמן המחזור נקבע ע"פ השהיית המערכת הצירופית הכוללת:
$T \geq 5 \cdot 50ns = 250ns$

מדדים נוספים של המערכת:

נרצה להיות מסוגלים למדוד את ביצועי המערכת על פי היחס ביצועים מול עלות.
את העלות ניתן למדוד על פי מספר הרכיבים/המעגלים במערכת.
נראה כרגע מספר מושגים בעזרתם נמדוד את יעילות המערכת.

ספיקה – Throughput –

מספר החישובים שהמערכת מבצעת ליחידת זמן (שניות בד"כ)

לדוגמא: במערכת הנתונה הספיקה המקסימלית תהיה:

$$Throughput = \frac{1}{T_{CYCLE}} = \frac{1}{250ns} = 4MHz = 4 \cdot 10^6 \frac{calc}{\sec}$$

שיהוי החישוב (כמיסות) – Latency –

הזמן שלוקח למערכת להשלים את החישוב מרגע קבלת נתון בכניסה.
במערכת צירופית הכמיסות היא הTpd. במערכת צירופית השיהוי והכמיסות זהים.
בדרך כלל נאמר שלמערכת ביצועים טובים אם הכמיסות נמוכה.

לדוגמא: במערכת הנתונה השיהוי הינו $T_{MIN} = 250\text{ns}$.

סעיף ב':

נדרש לבנות מערכת מצונרת המבצעת את אותו החישוב ע"י שימוש ברכיב הצירופי הנ"ל ובנוסף ברכיב זיכרון מסוג D-FF(7474) (שנלמד בתרגול הקודם).

להזכירכם: מתוך דפי הנתונים של D-FF(7474):

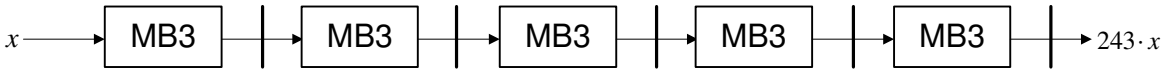
$T_{SETUP} = 20ns$
$T_{HOLD} = 5ns$
$T_{PD} = max\{T_{PLH}, T_{PHL}\} = 40ns$

יש לחשב עבור המערכת את זמן המחזור המינימלי בו ניתן להפעילה, את השיהוי ואת ספיקת המערכת.

הגדרות וכללים

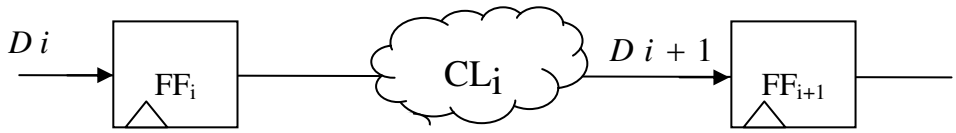
- צינור מדרגה K (K-pipeline) זהו מעגל ללא חוגים הבנוי מלוגיקה צרופית ורגיסטרים כך ש**כל מסלול מכניסה ליציאה** עובר דרך K רגיסטרים בדיוק.
- יש למקם רגיסטר בכניסת המערכת או ביציאתה (לצורך סינכרון עם מערכות אחרות). נהוג למקם את הרגיסטר ביציאה מהמערכת.
- נסמן את מיקומם של הרגיסטרים ב: |

<u>פתרון:</u>



<u>חישוב זמן המחזור:</u>

יש לחפש את המסלול האיטי ביותר בין כל שני רגיסטרים.



$$T \geq \max_{i=1}^{k-1} \{ T_{PD}(FFi) + T_{PD}(CLi) + T_{SETUP}(FF_{i+1}) \}$$

בדוגמא שלנו נקבל כי זמן המחזור המינימלי הוא:

$$Tmin = 40ns + 50ns + 20ns = 110ns$$

<u>חישוב ה- Throughput:</u>

Throughput מקסימלי מתקבל עבור זמן מחזור מינימלי והוא:

$$Throughput = \frac{1}{T_{\min}} = \frac{1}{110ns} = 9.09Mhz \cong 9 \cdot 10^6 \frac{calc}{\sec}$$

<u>חישוב ה- Latency:</u>

במערכת מצונרת השיהוי נקבע ע"פ זמן המחזור ודרגת הצינור.

בדוגמא שלנו נקבל:

$$Latency = K \cdot T_{\min} = 5 \cdot 110ns = 550ns$$

<u>השוואה בין שתי השיטות לבניית המעגל:</u>

	מס' הרכיבים הצירופיים	מס' הרגיסטרים	Throughput	Latency
מערכת לא מצונרת	5	0	4MHz	250ns
מערכת מצונרת	5	5	9.09MHz	550ns

<u>סעיף ג':</u>

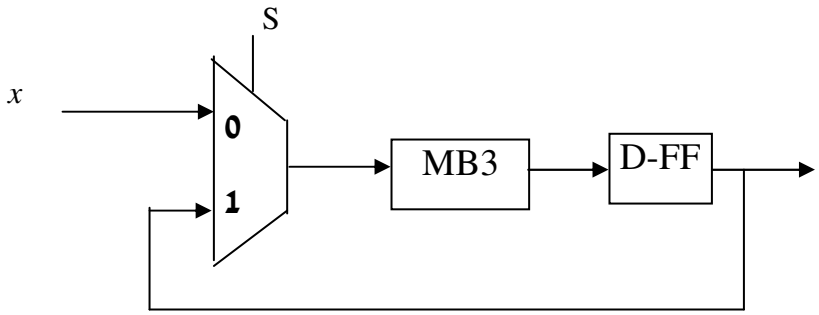
נניח כי נתוני הכניסה, x, מגיעים בקצב די נמוך כך שהשימוש בצינור שבנינו הוא לא כל כך מוצדק, היות והוא צרכן גדול של רכיבים, ולכן הוא מאוד יקר. כמוכן, המערכת הצירופית הטהורה (סעיף א') משתמשת ב-5 רכיבי MB3 , ולכן גם היא מערכת די יקרה.

יש לתכנן מערכת חדשה, זולה יותר, שתבצע את החישוב בעזרת רכיב MB3 בודד. יש לחשב עבור המערכת את זמן המחזור המינימלי בו ניתן להפעילה, את השיהוי ואת ספיקת המערכת.

<u>פתרון:</u>

לצורך תכנון המערכת נשתמש פרט לרכיב MB3 ברגיסטר D-FF(7474) ובנוסף בבורר (Selector 2→1).

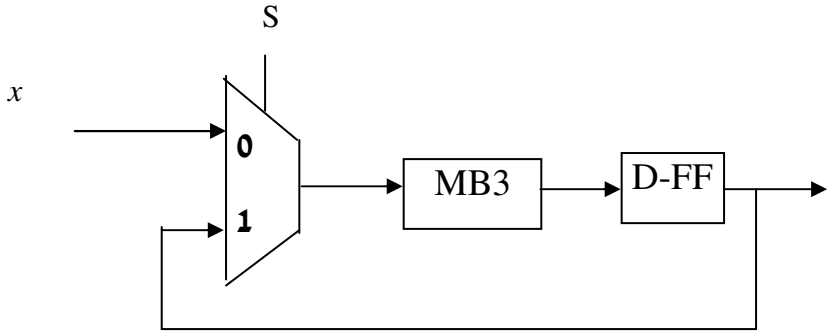
נתוני תזמון עבור הבורר: T_{PD}=15ns



אופן פעולת המערכת:

- במחזור ראשון S=0 והערך x נבחר בבורר.
- בארבעת המחזורים הבאים S=1 וערך התוצאה הקודם מועבר במערכת.
- לאחר חמישה מחזורי שעון הערך במוצא הוא 243x.
- ע"י קביעת S=0 מוכנס למערכת ערך x הבא.

הערה: בשלב זה לא נתעמק בבניית הבקר שמייצר את האות S.



<u>חישוב זמן המחזור המינימלי:</u>

$T_{min} = T_{PD}(D\text{-}FF) + T_{PD}(Sel) + T_{PD}(MB3) + T_{SETUP}(D\text{-}FF) =$
$40ns + 15ns + 50ns + 20ns = 125ns$

<u>חישוב ה- Latency:</u>

יש לשים לב כי חישוב אחד מושלם רק לאחר חמישה מחזורי שעון ולכן:

$$Latency = 5 \cdot T_{\min} = 5 \cdot 125ns = 625ns$$

<u>חישוב ה – Throughput:</u>

יש לשים לב כי זו אינה מערכת מצונרת ע"פ הגדרתה ולכן ה – Throughput אינו נקבע ע"פ $\frac{1}{T_{\min}}$.

כיוון שחייבים לחכות משך זמן של Latency בין כל שתי תוצאות חישוב נקבל:

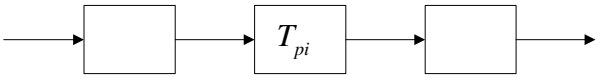
$$Throughput = \frac{1}{Latency} = 1.6MHz = 1.6 \cdot 10^6 \frac{calc}{\sec}$$

<u>השוואה בין שלוש השיטות לבניית המעגל:</u>

	מס' הרכיבים הצירופיים	מס' הרגיסטרים	Throughput	Latency
מערכת לא מצונרת	5	0	4MHz	250ns
מערכת מצונרת	5	5	9.09MHz	550ns
מערכת דמוי-מצונרת	1(MB3)+1(Sel)	1	1.6MHz	625ns

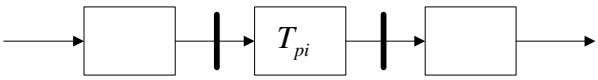
סיכום

<u>מערכת לא מצונרת</u>



$$i = 0,...,k-1$$
$$Latency = \sum_i Tpi$$
$$Throughput = \frac{1}{Latency}$$

<u>מערכת מצונרת</u>



$$i = 0,...,k-1$$
$$Tcl \geq Tpi_{\max}$$
$$Latency = K \cdot Tcl$$
$$Throughput = \frac{1}{Tcl}$$

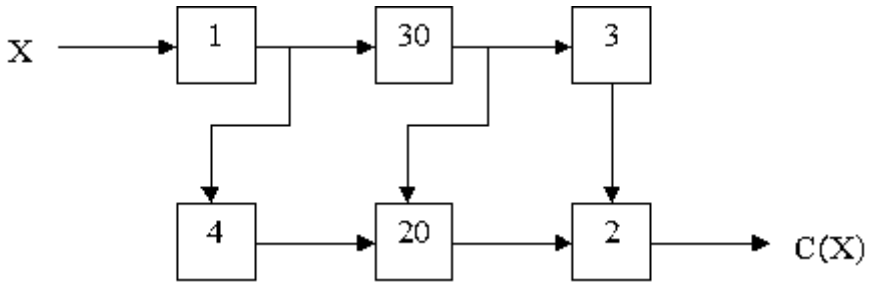
כללים לבניית צינורות

<u>כלל 1</u>: בצינור k, ניתן להוסיף רגיסטר <u>בכל יציאה</u>, ולקבל צינור k+1 בעל אותה פונקציה (ולהיפך).
<u>כלל 2</u>: בצינור k, בו יש מעגל צירופי E, ניתן לבטל רגיסטר בכל יציאה של E ולהוסיף רגיסטר בכל כניסה של E, ולהישאר עם צינור k בעל אותה פונקציה (ולהיפך).

דוגמאות לתרגילים בצינורות

<u>תרגיל 1</u>

נתון המעגל הצירופי הבא, המורכב מ6 יחידות לוגיות. בשרטוט, על כל יחידה רשום ה t_{pd} שלה. t_{cd} עבור כל אחד מהיחידות הוא 0.



א. מהו הlatency ומהו ה-throughput של המערכת הנ"ל?

ב. מקם מספר מינימלי של רגיסטרים אידיאלים על מנת לקבל throughput מקסימלי.

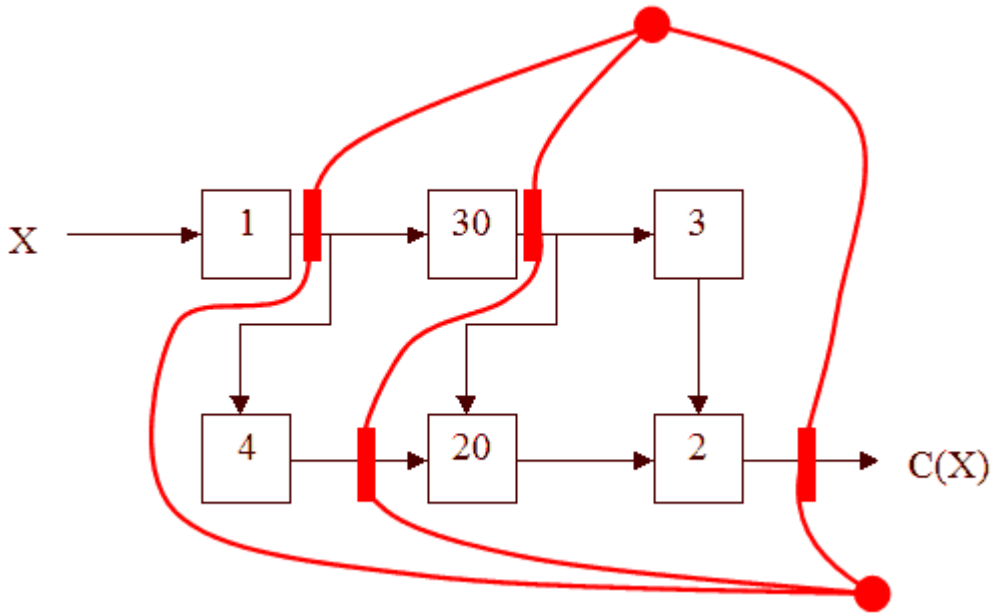
ג. מהו הlatency ומהו ה-throughput של המערכת המצונרת?

<u>תשובה</u>

א. הlatency הוא המסלול הארוך ביותר בין X ל $C(X)$: $1+30+20+2=53$

הthroughput עבור מערכת צירופית הוא $\frac{1}{latency}$ והוא $\frac{1}{53}$.

ב. נשתמש בארבעה רגיסטרים (1 ביציאה):

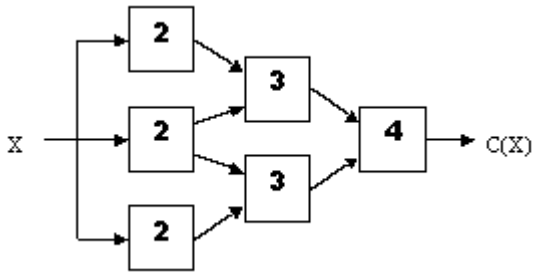


ג. הthroughput הוא 1/(max pipeline stage delay) = 1/30

הLatency: (1/throughput)*(number of pipeline stages) = 30 * 3 = 90

<u>תרגיל 2</u>

נתונה המערכת הבאה, המקבלת אות X ומצפינה אותו:



נניח בשאלות הבאות שכל הרגיסטרים איתם אנו מתעסקים הם רגיסטרים אידיאלים.

א. מהו הLatency של המערכת?

ב. אם נרצה לשפר את הthroughput של המערכת, מהו המספר המינימלי של רגיסטרים שנצטרך להוסיף?

ג. אם אנו צריכים להוסיף בדיוק חמישה רגיסטרים, מהו הthroughput המקסימלי שנוכל לקבל?

ד. אם אנו יכולים להוסיף רגיסטרים כרצוננו, מהו הthroughput המקסימלי שנוכל להשיג?

ה. אם אנו יכולים להוסיף רגיסטרים כרצוננו, מהו הlatency המינימלי שנוכל להשיג?

<u>תשובה</u>

א. הlatency הוא המסלול הארוך ביותר בין X ל $C(X)$: $2+3+4=9$

ב. מספר הרגיסטרים המינימלי שנצטרך להוסיף הוא 3. לפי הכללים שהגדרנו לבניית צינורות, אנו צריכים להוסיף רגיסטר אחד ביציאה. כמו כן, אנו צריכים שבכל אחד מהמסלולים אל היציאה יהיה אותו מספר רגיסטרים, לכן נוסיף עוד רגיסטר בכל אחד מהמעברים בין הרכיבים עם השהיה 3 לרכיב בעל השהיה 4 - כלומר שני רגיסטרים, סך הכל הוספנו 3 רגיסטרים.

ג. הthroughput הטוב ביותר שנוכל להשיג הוא 1/5. נציב 3 רגיסטרים ביציאה, ועוד שני רגיסטרים בין הרכיבים בעלי ההשהיה 3 לרכיב בעל ההשהיה 4.
נשים לב שאם נמקם את הרגיסטרים בצורה שקל יותר לראות: 4 רגיסטרים בין המודולים עם השהיה 2 ל3 ועוד רגיסטר אחד ביציאה, נקבל throughput גרוע יותר - 1/7.

ד. הthroughput הטוב ביותר שנוכל להשיג אם אנו יכולים להוסיף רגיסטרים כרצוננו, הוא 1/4, וזאת על ידי בידוד הרגיסטרים ככה שהשלב הארוך ביותר לוקח זמן 4, ולכן הthroughput יהיה 1/4.

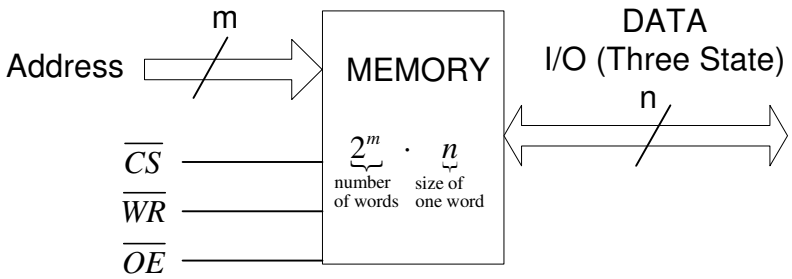
ה. הlatency המינימלי שנוכל להשיג הוא 9. איננו יכולים אף פעם להקטין את הlatency על ידי הוספת pipelines, בדרך כלל כשאנו מוסיפים צינורות הlatency רק עולה.

זיכרון סטטי (Random Access Memory–Static RAM)

ישנם מספר סוגי זיכרון בהם אנו משתמשים.
ROM - Read Only Memory זהו זיכרון קבוע. לאחר שקבעו את הנתונים שבו, הנתונים קבועים, ונשארים ללא צורך במתח.
RAM - Random Access Memory זהו זיכרון לקריאה ולכתיבה.
Hard Disk זהו זיכרון לקריאה וכתיבה לטווח ארוך יותר מהRAM. הוא איטי יותר מהRAM.

אנו נתייחס לשני סוגים של RAM : SRAM וDRAM.
SRAM - Static RAM - אם נכתוב מידע לזיכרון ונבדוק את הזיכרון אחרי תקופה, האות יישאר שם ויישאר תקף.
DRAM - Dynamic Ram - זיכרון יותר זול מהSRAM, יותר מהיר ממנו ויכול להכיל יותר מידע. החיסרון של DRAM הוא שלאחר תקופה, עקב פריקת הקבלים מהם בנוי ה-DRAM, המידע על הDRAM מפסיק להיות תקף.

תיאור זיכרון SRAM



כאשר אין $\overline{CS}$ קווי ה-I/O DATA עוברים למצב HIGH Z.

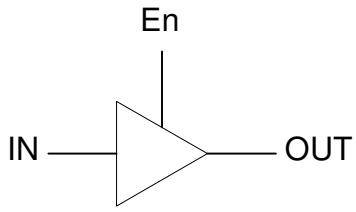
לדוגמא : רכיב HM6116 של חברת HITACHI. זהו RAM המכיל 2048 מילים בנות 8 סיביות כל אחת.

<u>תאור הפינים של הרכיב :</u>

A0-A10 : אחד עשר קווי כתובת למיעון אחת מ2048 המילים בזיכרון.
I/01-I/O8 : שמונת קוי הנתונים לקריאה/כתיבה.
$\overline{OE},\overline{WE},\overline{CS}$: שלושה קווי בקרה הקובעים את אופן פעולת הרכיב.
$\overline{CS}$: קו בקרה Chip Select. כאשר מאולץ על הקו "גבוה", היציאות עוברות למצב High-Z וצריכת הזרם של הרכיב יורדת. לצורך עבודה יש לאלץ "נמוך".
$\overline{WE}$: (WRITE ENABLED) כאשר "גבוה" תבוצע קריאה. כאשר "נמוך" תבוצע כתיבה.
$\overline{OE}$: (OUTPUT ENABLED) כאשר "גבוה" לא מאופשרת יציאת נתונים על קוי I/O.

כאשר $\overline{OE}$ ב-0 מתאפשר OUTPUT. כשהוא ב-1 הוא מוציא Hi-Z ומישהו אחר יכול להוציא OUTPUT לקו.

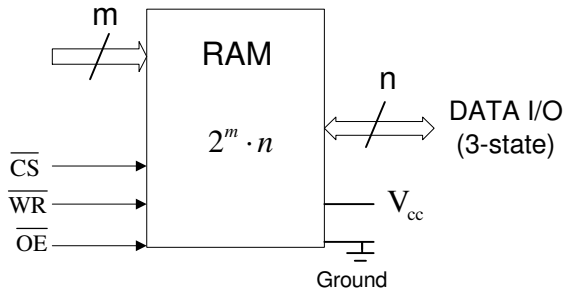
רכיב Three-state



כאשר ה-En ב1, הרכיב מתפקד כחוט רגיל.
כאשר ה-En של הרכיב ב0, היציאה שלו מנותקת, כלומר הרכיב איננו כותב כל ערך לוגי על היציאה.
המצב בו היציאה איננה 0 ואיננה 1 נקרא "High-Z".

היתרון של רכיבים כאלו הוא שניתן לחבר יחד יציאות של מספר רכיבים כאלו, בתנאי שלכל היותר רק רכיב אחד יורשה לכתוב על היציאה המשותפת.
היציאה המשותפת קרויה BUS (עורק).

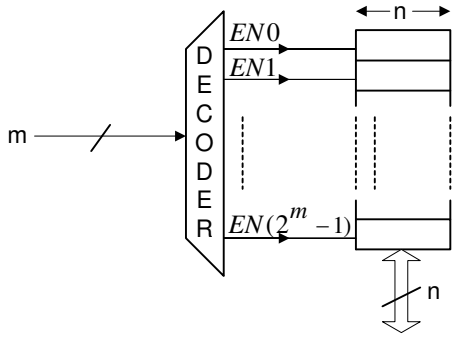
מימושים אפשריים לSRAM



1. **מוט זיכרון:**

אם $m \ll 2^m$ למפענח (Decoder) יהיו 2^m מינטרמים (ללא צימצום)

לדוגמא: $m = 20, n = 8 \quad (1MB) \Leftarrow 2^{20}$ מינטרמים $\Leftarrow$ המון לוגיקה.



2. **מבנה ריבועי:**

ישנם $m = j + k$.

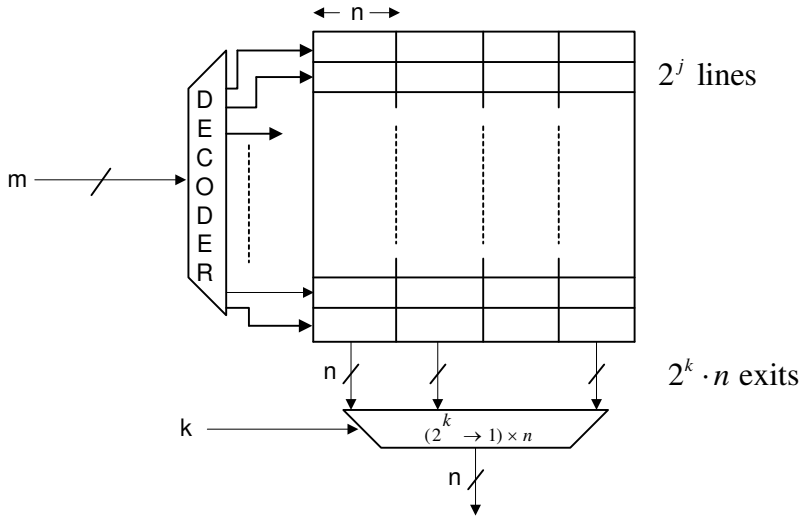
$2^j + 2^k$ מינטרמים במפענח.

לדוגמא:

$m = 20, n = 8$

אם $j = k = 10 \Leftarrow$ יהיו $2 \cdot 2^{10}$ מינטרמים.

$2^{20} \gg 2^{11}$



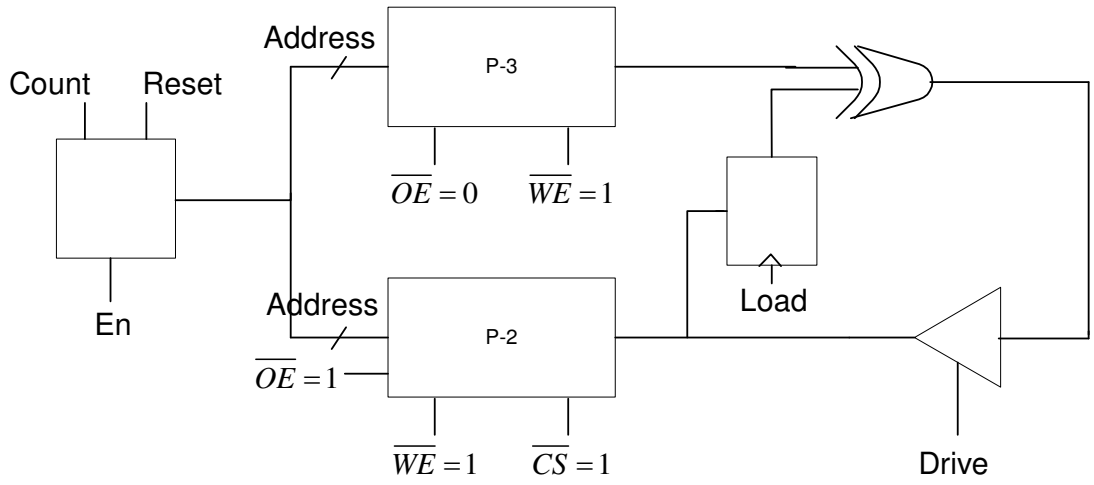
תרגיל

נתון רכיב SRAM מסוג HM6116/P-2. מעוניינים לבצע הצפנה של 2048 הבתים בSRAM זה.
המפתחות עבור ההצפנה נמצאים בSRAM מסוג HM6116/P-3. הלוגיקה עבור ההצפנה של הבית הi (להלן B_i) מבוטאת באמצעות ההשמה $B_i \leftarrow B_i \oplus b_i$, כאשר b_i הינו הבית המתאים (בעל אותה כתובת) בSRAM שמכיל את המפתחות (לדוגמא, אם הנתון עבור כתובת כלשהי הוא 10101101 והמפתח המתאים לאותה כתובת הוא 11101110, אזי הנתון שייכתב הוא 01000011).
עליכם לתכנן מעגל המבצע את הפעולה הנ"ל. המעגל ייגש לכל אחת מהכתובת במרחב הזיכרון של שני הSRAM, יקרא את הערכים הקיימים, יבצע את פעולת ההצפנה ויכתוב את התוצאה לאותה הכתובת בSRAM מסוג HM6116/P-2. שימו לב שפעולת הכתיבה תבוצע רק עבור הSRAM המוצפן, בעוד שערכי SRAM המפתחות ישארו ללא שינוי.

לרשותכם, בנוסף לSRAM-ים ולבקר המעגל:
שערי XOR, רגיסטרים וחוצצים. כמו כן לרשותכם מונה 16 ביט בעל קו reset, קו cnt וקו enable. (קו reset מאפס וקו cnt מקדם באחד, שניהם בעליית enable מ0 ל1).

שרטטו את מסלול הנתונים בלבד. אין צורך לממש את מעגל הבקרה, אלא רק לציין את קווי הבקרה ולהסביר את סדר פעולתם. הניחו כי קו הבקרה $\overline{CS}$ לשני הSRAM-ים פעיל כל הזמן (מקוצר לאדמה).

פתרון



- הDrive הוא הEn של הThree-state, פשוט נתנו לו שם ייחודי.
- הLoad מחובר לכניסת השעון של הרגיסטר. כשניתן שם 1, הרגיסטר ידגום. זהו קו בקרה עליו אנו שולטים.

סדר הפעולות

כל מה שקשור לSRAM מתייחס לP-2. קווי הבקרה של P-3 קבועים כמתואר בשרטוט.
אנו מניחים כי בתחילה $\overline{WE}=1$.

1. להעלות count enable
2. להוריד $\overline{OE}$ (שידחוף)
3. להעלות Load
4. להעלות Drive
5. להעלות $\overline{WE}$
6. להוריד Drive

משימות לביצוע במקביל לפעולות

1. להוריד $\overline{WE}$
2. להוריד Load
3. להוריד count enable

וחוזר חלילה.

סימונים בגרף תזמונים

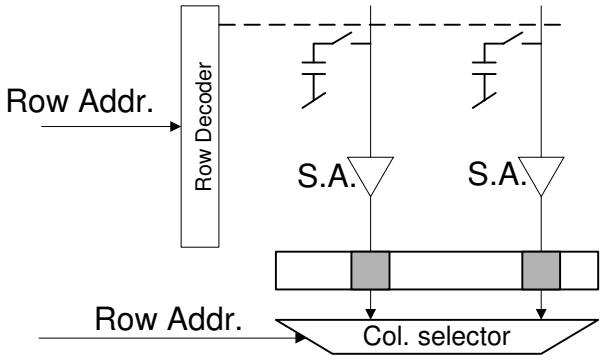
לא ידוע אם יהיה 0 או 1, אבל בהכרח יהיה מצב יציב (אחד מהם)		1 0
ערך לא תקף		1 0
Hi-Z		1 0
ערך בתהליך של שינוי		1 0

זיכרון דינמי (Random Access Memory–Dynamic RAM)

ניתן לחלק זיכרונות RAM מחיקים (כאלה המאבדים את המידע כאשר הם מנותקים מאספקת החשמל) לזיכרונות סטטיים ודינמיים.

- זיכרון סטטי: המידע נשמר בזיכרון כל עוד הרכיב מקבל מתח.
- זיכרון דינמי: אחרי פרק זמן מסוים, אם לא ננקוט פעולה מונעת, המידע עלול להימחק מהזיכרון.

מבנה ועקרונות פעולה



קווי בקרה

ADDR - כתובת

$\overline{W}$ - 0 לכתיבה ו1 לקריאה

Row Address Strobe - $\overline{RAS}$

Columns Address Strobe - $\overline{CAS}$

גישה (קריאה/כתיבה) למידע

בירידת אות ה $\overline{RAS}$ הרכיב דוגם את כתובת השורה. ערכם של הקבלים בשורה הנבחרת נקרא אל זיכרון השורה (שהוא זיכרון סטטי). תהליך הקריאה הרסני - קריאת הקבלים מלווה בפריקתם.

אות ה $\overline{CAS}$ דוגם את כתובת העמודה המשתמשת לבחירת ביט מסוים (במקרה של זיכרון ברוחב ביט אחד) מתוך זיכרון השורה. ערכו של ביט זה נקרא (אם זהו מחזור קריאה) או שערך חדש נכתב לתוכו (אם זהו מחזור כתיבה).

אות ה $\overline{WE}$ קובע אם גישה לזיכרון תהיה קריאה או כתיבה.

בסוף מחזור (עליית $\overline{RAS}$), תוכן זיכרון השורה נכתב חזרה לשורת הקבלים.

נשים לב כי הכתובת מחולקת לכתובת שורה ולכתובת עמודה. כמו כן איננו מספקים בו זמנית את שתי הכתובות. לכן נוכל לחסוך בהדקים - אותם הדקי כתובת משמשים הן לכתובת השורה והן לכתובת העמודה. (חסכון של $\frac{1}{2}\log_2(\text{memory size})$. בDRAM המקובלים כיום, מדובר ביותר מ10 הדקים).

רענון

קבל מציאותי, כמו אלו בDRAM, זולג, ולכן החל מטעינתו המתח עליו יורד בהדרגה. אם לא ננקוט פעולה מונעת, המתח על הקבלים שהיו טעונים ל"1" לוגי ירד מתחת לסף המאפשר הבחנה בינם ובין אלו שמאחסנים "0" לוגי. הפתרון המתבקש הוא, אחת לפרק זמן מסוים, לרענן את תוכן הקבלים.

רענון שורה כולל הורדת השורה אל זיכרון השורה וכתיבתה חזה אל הקבלים, ובכך הם נטענים למטען מלא.
נזכור שבגישה (קריאה/כתיבה) לזיכרון שורה יורדת לזיכרון השורה ונכתבת חזרה לקבלים, ובכך מרוענת אותה שורה. יש לוודא שכל השורות זוכות לרענון, ולא רק אלו שניגשו אליהן.

שיטות רענון - סוגי מחזורי רענון

$\overline{\text{RAS}}$ Only Referesh (ROR)

זוהי הצורה הפשוטה ביותר לרענן את המידע הנמצא בDRAM. יש לספק כתובת שורה, ואז אנו מורידים את ה $\overline{\text{RAS}}$ למשך פרק הזמן הדרוש, ושורת הזיכרון שהוגדרה על ידי הכתובת שסופקה מרוענת.
סוג זה דורש שהמערכת החיצונית לDRAM תעקוב אחרי מספרי השורות שיש לרענן.
בשיטה זו $\overline{\text{CAS}}$ גבוה לאורך כל המחזור ורק ה- $\overline{\text{RAS}}$ יורד.

$\overline{\text{CAS}}$ Before $\overline{\text{RAS}}$ (CBR)

$\overline{\text{CAS}}$ יורד ראשון (שונה מכל שאר מחזורי העבודה). $\overline{\text{RAS}}$ יורד למשך פרק זמן מסוים ומונה פנימי קובע איזה שורה תרוענן.

Self Refresh / Sleep Mode / Auto Refresh

במקרה זה משמש מתנד פנימי לקביעת מחזור הרענון, ומונה פנימי מייצר את כתובות השורות.
בד"כ משמש להתקנים ניידים מבוססי סוללות. צריכת הספק במצב זה נמוכה מאוד. דיאגרמת הזמנים נראית כמחזור CBR ארוך.

שיטות גישה לזיכרון

קריאה מהזיכרון

<u>מחזור קריאה בסיסי</u>

1. יש לספק את כתובת השורה t_{ASR} לפני ירידת $\overline{RAS}$ ולהחזיקו t_{RAH} אחריה.
2. על $\overline{RAS}$ לרדת ולהישאר נמוך לפחות t_{RAS}.
3. יש לספק את כתובת העמודה t_{ASC} לפני ירידת $\overline{CAS}$ ולהחזיקו t_{CAH} אחריה.
4. על $\overline{WE}$ להשאר גבוה במחזור קריאה החל מ t_{RCS} לפני ירידת $\overline{CAS}$.
5. על $\overline{CAS}$ לרדת ולהישאר נמוך לפחות t_{CAS}.
6. יש להוריד את $\overline{OE}$ (אם יש כזה, בDRAM שראינו אין) לפי דרישות התזמון לצורך קריאת הנתון.
7. הנתון מופיע ביציאה בתלות בזמנים בהם $\overline{RAS}$ (t_{RAC}), $\overline{CAS}$ (t_{CAC}), $\overline{OE}$ (t_{OEA}) ירדו, וזמן הספקת כתובת העמודה (t_{AA}).
8. לפני שניתן להתחיל מחזור חדש, $\overline{RAS}, \overline{CAS}$ צריכים לשהות במצב הלא פעיל (הגבוה) זמן מסויים (t_{RP}, t_{CRP} בהתאמה).

<u>Fast Page Mode (FPM) או Ripple Mode</u>

אם ברצוננו לקרוא מספר נתונים מאותה השורה, ניתן להשאירה בזיכרון השורה ולספק רק כתובות עמודה חדשות.
מתחילים כמו במחזור הקריאה הבסיסי משלבים 1 עד 7. כדי לגשת לכתובת אחרת באותה שורה, יש להעלות את $\overline{CAS}$ למשך t_{CP} ולאחר מכן ניתן לחזור על שלבים 3 עוד 7 פעם נוספת.

<u>Extended Data Out או Hyper Page Mode</u>

אופן פעולה זה איננו ממומש בDRAM הנלמד.
EDO דומה לFPM. ההבדל העיקרי הוא שיציאות הנתונים לא מנוטרלות (high-Z) עם עליית $\overline{CAS}$. נתונים ממחזור הקריאה הקודם מצויים ביציאת הנתונים במקביל לתחילת מחזור הקריאה הבא - למעשה יש פה צינור בעל שני שלבים - הספקת הכתובת ושלב הוצאת הנתונים. דבר זה מאפשר זמן מחזור קצר יותר ושיפור של כ40% בביצועים בהשוואה לFPM.

כתיבה לזיכרון

צורת מחזור הכתיבה דומה למחזור הקריאה. $\overline{WE}$ יורד במהלך המחזור למשל לפחות t_{WP}. יש לספק בכניסה נתון יציב t_{DS} לפני ו t_{DH} אחרי המעבר הקובע (ירידת $\overline{WE}$ או ירידת $\overline{CAS}$).

כתיבה מבוקרת $\overline{CAS}$

במקרה זה ירידת $\overline{CAS}$ היא המעבר הקובע.
$\overline{WE}$ יורד לפני ירידת $\overline{CAS}$ לפחות t_{WCS} ונשאר למטה לפחות t_{WCH} אחריה.
צורת עבודה זו מתאימה, למשל, לכתיבה בFPM.

כתיבה מבוקרת $\overline{WE}$

במקרה זה ירידת $\overline{WE}$ היא המעבר הקובע.
צורת עבודה זו מתאימה, למשל, למחזור Read-Modify-Write, בהם קוראים תוכן כתובת, וכותבים ערך חדש לתוך אותה כתובת. $\overline{CAS}$ נשאר נמוך מגישת הקריאה לגישת הכתיבה.

רכיב KM41C16000C-5

רכיב זה הוא רכיב זיכרון מסוג DRAM. הוא מכיל 16 מגה סיביות.

$16 \cdot 2^{20} = 2^{24}$

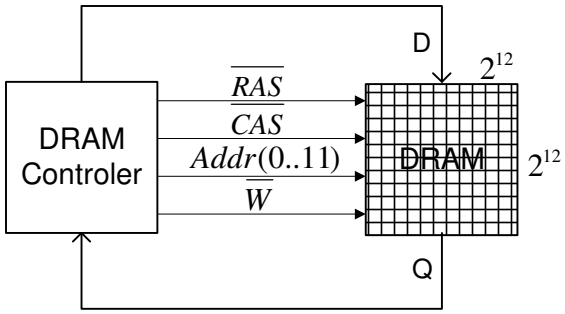
$\log_2^{2^{24}} = 24$

ישנם 24 קווי כתובת ברכיב מסוג זה.

לכן, הDRAM יוכל להיות ריבועי לחלוטין.

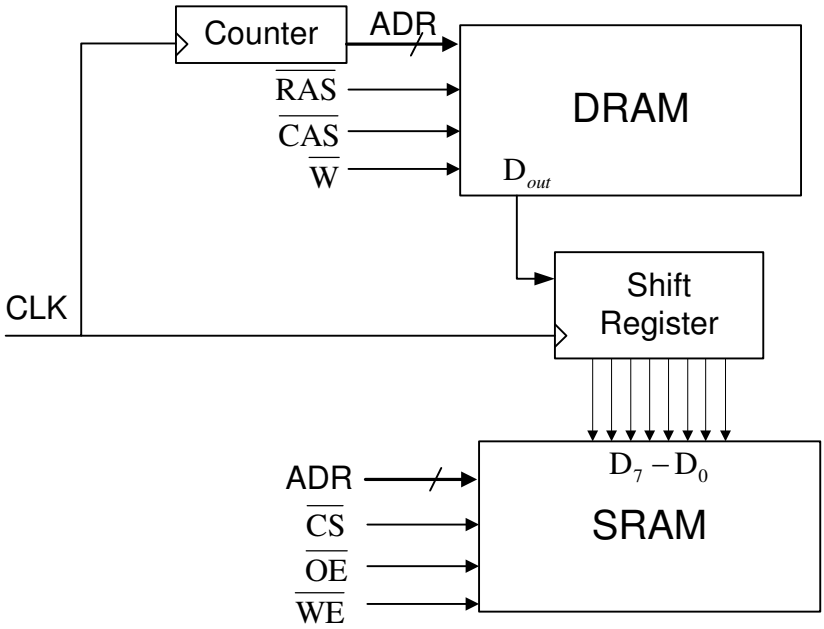
הערה כללית

קווי בקרה הם תמיד 0 או 1, והם אינם Three-State אף פעם, וזאת מכיוון שהם אינם משותפים לרכיבים שונים.



דוגמא

המעגל הבא נועד להעתיק שורה אחת של DRAM מסוג KM41C16000C-5 לזיכרון SRAM מסוג HM6116/P-2.
הקווים $\overline{RAS},\overline{CAS},\overline{W}$ לDRAM וכן $\overline{OE},\overline{CS},\overline{WE}$ לSRAM מגיעים מבקר נפרד שאיננו מופיע בשרטוט.
הקריאה מהDRAM נועדה להתבצע בFast Page Mode.
נניח שכתובת השורה כבר ניתנה לDRAM וכי קו ה $\overline{RAS}$ כבר מופעל.
אות השעון המשותף לCounter ולShift Register הוא בעל זמן מחזור T_{clk}. לאף אחד מהשניים אין כניסת Enable כלשהי. המונה מתקדם ואוגר ההזזה מזיז בכל עליית שעון. נתוני התזמון של המונה ושל אוגר ההזזה זהים:
$T_{su}=10ns,\ T_{pd}=12ns,T_h=0,T_{cd}=0$.



א. הסבר כיצד המעגל פועל. התייחס לנקודות הבאות:
- לשם מה נדרש הShift Register?
- לכמה תאי זיכרון SRAM תועתק שורת DRAM אחת?

ב. תאר במדויק מתי אמור הבקר לתת את האותות הבאים, על מנת שפעולת ההעתקה תסתיים תוך הזמן הקצר ביותר האפשרי:
- $\overline{CAS}$ של DRAM.
- $\overline{WE}$ של SRAM.
- ADR של SRAM.

ג. מהו זמן המחזור המינימלי של השעון שיבטיח פעולה תקינה של המעגל, בהנחה שהבקר מסוגל לתת את כל אותו הבקרה בזמנים מדויקים כרצוננו?

ד. כמה זמן תמשך בסך הכל ההעתקה של שורת DRAM אחת?

פתרון

א.
פעולת המעגל: ערך המונה משמש ככתובת העמודה. בכל מחזור מתקדם המונה וכך מתקדמת כתובת העמודה ונקראת סיבית חדשה מהDRAM. לאחר שמונה מחזורי שעון מכיל אוגר ההזזה ערכים של שמונה סיביות ותוכנו נכתב לתא זיכרון בSRAM. השימוש באוגר נחוץ מכיוון שמהDRAM נקרא המידע סדרתית סיבית אחרי סיבית ולSRAM הוא נכתב במקביל - כל פעם בית שלם.
כל שורת DRAM מכילה 2^{12} סיביות = 4096 סיביות.
כל תא בSRAM מכיל 8 סיביות, ולכן תוכן שורה ייכתב ל $\frac{4096}{8}=512$ בתים.

ב.
תנאי להורדת CAS - כתובת עמודה יציבה.
תנאי לעליית CAS - קיום תנאי Hold של Shift Register.
CAS ירד אחרי לפחות $\underbrace{T_{pd\ Counter}}_{12}+\underbrace{T_{ASC}}_{0}=12ns$ מקבלת הClock Trigger.
CAS יעלה $t_{hold\ register}-t_{off}=0ns$ לפני קבלת הtrigger.
אנו רואים כי מתקיים תנאי t_{cp} מכיוון שיש הבדל של לפחות $12ns$ בין עליית וירידת הCAS.

זמן עליית ה $\overline{WE}$ - לפחות $T_{ow}+T_{pd_reg}=12+35=47ns$ לאחר קבלת ה Clock Trigger.
זמן ירידת ה $\overline{WE}$ הוא t_{wp} לפני עלייתו, כלומר $70-47=23ns$, לפי קבלת ה Clock Trigger של המחזור השמיני.

ADR צריך להשתנות לפחות $\max(105-47,23+20)=58ns$ לפני קבלת ה Clock Trigger של המחזור השמיני ולא לפני $47ns$ אחרי קבלת אותו trigger.

ג.

התנאים שאנחנו צריכים לקיים:
1. זמן מינימלי של החזרת $\overline{CAS}$ במצב נמוך:
$$T_{clk}\geq\left(T_{cas}+T_{ASC}\right)+T_{cas}+\left(T_{hold}-t_{off}\right)=25ns$$
2. קיום T_{su} של הShift Register:
$$T_{clk}\geq\left(T_{cas}+t_{ASC}\right)+t_{cas}+t_{su_reg}=12+13+10=35ns$$
3. תנאי יציבות כניסות הSRAM:
$$T_{clk}\geq\left(T_{pd_reg}+T_{ow}\right)+T_{OH}-T_{cd_reg}=47+5-0=52ns$$

ד.
ישנם 4096 ביטים בשורה ולכן תוך 4096 מחזורי שעון מסתיימת כתיבת שורה ולכן הזמן הוא $4096\cdot T_{clk}\approx 2130ns$.

בקרה

כאשר אנו מתחילים להתעסק עם מערכות ספרתיות מורכבות, אנו צריכים שליטה בסדר ובזמן של הפעולות השונות המתרחשות במערכת.
נחלק את המערכת הספרתית לשני חלקים: נתיבי נתונים, בהם יתבצעו החישובים השונים ובהם יאוכסן המידע, ויחידת בקרה שתהיה אחראית לתזמן את הפעולות בנתיבי הנתונים.
המשטר הדינמי מחייב אותנו לפתח שיטות של מעקב מתי אות הוא תקף, כדי למנוע מטה-סטבילויות או אותות לא תקפים.
היתרון שחיוב זה נותן לנו הוא האפשרות להתייחס לערכי זמן בדידים במקום להתייחס לזמן כרציף.

כעת נרצה לראות משטרים מבוססי זמן נוספים, ולראות את היתרונות והחסרונות בכל אחד מהם.
משטרים מבוססי זמן מתחלקים בשתי נקודות עיקריות:

- משטרים סינכרוניים (synchronous) מול משטרים אסינכרוניים (asynchronous).
- משטרים לוקליים (local) או גלובליים (global).

מערכות סינכרוניות מעניקות לנו צורת הסתכלות בדידה על הזמנים, המבוססת על אות שעון כללי המגיע ליחידות השונות במערכת. התוצאה היא שאירועים "מעניינים" כגון טעינת ערכים לרגיסטרים, קוראות רק בשפה הפעילה של אות השעון.
לעומתן, מערכות אסינכרוניות אינן מסתמכות על שעון. כאשר התקן מסיים פעולה, הוא שולח אות בקרה ומודיע ליחידות אליו הוא מחובר, שהפעולה הסתיימה והנתונים מוכנים.
מערכות אסינכרוניות יכולות לספק ביצועים גבוהים יותר ממערכות סינכרוניות, מכיוון שהמערכת יכולה להמשיך לבצע פעולה ברגע שהנתון מוכן, ולא רק בעליית השעון הבאה. אם זאת, מערכות אסינכרוניות יותר מסובכות לתכנון.

תזמון גלובלי מתייחס לכל המערכת כמקשה אחת.
בהינתן מערכת המסוגלת לבצע מספר פעולות, בעלת תזמון גלובלי, נניח שכל הפעולות לוקחות את אותו זמן קבוע.
תזמון לוקלי, לעומת זאת, מאפשרים למודול לסיים פעולה בזמן משתנה, ולפיכך כאשר עובדים בגישה זו מוסיפים קו בקרה כדי לציין מתי הנתונים מוכנים.
למשל, במערכת בה חלק מהפעולות שרכיב מסוים מבצע לוקחות 100ns וחלק מהן לוקחות 50ns, נאמר שזמן ביצוע פעולה ברכיב הוא 100ns אם אנו עובדים בתזמון גלובלי, לעומת זאת, בתזמון לוקלי, נאמר שלעיתים זמן ביצוע הפעולה הוא 100ns ולעיתים 50ns.
בתזמון לוקלי ננצל את העובדה שחלק מהמערכת יכולה לפעול מהר יותר, כדי להשתמש בנתונים ברגע האפשרי.

נוכל כעת לסכם ולהגדיר ארבעה סוגי משטרי זמנים:

- Synchronous globally timed (SGT) systems - מערכות בהם אותות הזמן נשלחים משעון משותף יחיד. מערכות SGT נוצרות בקלות מרכיבים צירופיים ורגיסטרים, אשר לכולם מחובר שעון בודד. כאשר מתכננים מכונות מצבים, משתמשים למעשה בשיטה זו. שיטה זו נפוצה מאוד עקב פשטות התכנון והמימוש שלה.
- Asynchronous globally timed (AGT) systems - בשיטה זו כל המערכת הינה אסינכרונית. תזמון האותות נעשה באמצעים אנלוגיים כגון קווי השהייה בשיטת "כל מקרה לגופו". שיטה זו בדרך כלל נחשבת בחירה גרועה כאשר מתכננים רכיבים, ולא קיימות מערכות גדולות המשתמשות בשיטה זו.
- Synchronous locally timed (SLT) systems - במערכות המתוכננות בשיטה זו, הרכיבים מקבלים אותות בקרה המסונכרנים עם שעון משותף. מערכות SLT מסוגלות לבצע פעולות בזמן משתנה, אולם כל פעולה לוקחת מספר קבוע של מחזורי שעון. מודולים של SLT ניתנים לחיבור ביחד בקלות ליצירת רכיבי SLT חדשים.
- Asynchronous locally timed (ALT) systems - האותות נקבעים לפי עוצמתם ולא לפי מחזור שעון. מודול של ALT יכול להתחיל לעבוד בכל רגע, ויכול להודיע על סיום הפעולה בכל רגע. ניתן להרכיב רכיבי ALT על מנת ליצור רכיבים חדשים, אולם יש צורך בתשומת לב למשכי אותות הבקרה, שלא יפרו את המשטר הדינמי.

<u>דרגות של סנכרון</u>

באופן מילולי - מערכות סינכרוניות הן מערכות בהן אירועים יכולים להתרחש באותו הזמן בדיוק, ואילו במערכות אסינכרוניות לא. באופן מעשי, נגדיר מערכות סינכרוניות כמערכות שאנו יודעים על קשרים של זמן בין הפעולות השונות במערכת, ואילו במערכות אסינכרוניות איננו יודעים על כאלו. למשל, נוכל לדבר על מערכת שיש בה מספר שעונים, ולא שעון משותף אחד, ולהגיד שהיא מערכת סינכרונית. עם זאת, נוכל להרוויח מעט מהיתרונות של המערכות האסינכרוניות, על ידי כך שנוכל לתת קצב שעון מהיר יותר בקטעים שונים של המערכת.
נביט כעת במספר שיטות לסנכרן את המערכת.

<u>Single Clock Synchronous Systems</u>

שיטה זו היא השיטה הנוקשה ביותר. שיטה זו היא הבסיס למשטרים SLT וSGT.
Single Clock Synchronous Systems: במערכת כולה שעון בודד. המערכת והשעון מקיימים את העקרונות הבאים:

- המערכת מורכבת מרכיבים צירופיים ומרגיסטרים המחוברים אל השעון.
- המעגל חופשי מחוגי צירופים, כלומר, כל חוג מכיל לפחות רגיסטר אחד.
- אות השעון מגיע ונמשך תקופה של t_{cl} , מגיע לכניסת השעון של כל רגיסטר במערכת.
- ה t_{pd} המצטבר של כל מסלול במערכת חייב להיות קטן ממחזור השעון.

כתוצאה ממשטר זה, כניסות הרגיסטרים יציבות בסמוך למעבר הפעיל של השעון, ולכן אנו עומדים בדרישות המשטר הדינמי.
שיטה זו מטילה הגבלות כבדות על המהנדס, אך עדיין שיטה זו מומלצת מכיוון שהיא מפשטת מאוד את תכנון המעגלים.

Multiple-Clock Systems

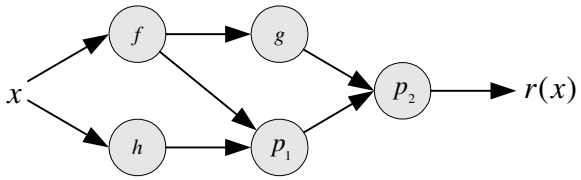
כפי שציינו, הרעיון של מערכות אסינכרוניות לגמרי הוא רעיון גרוע. פתרון המנצל את היתרונות של מערכת אסינכרונית עם היתרונות שנותנת לנו מערכת סינכרונית הוא הוספת מספר שעונים אסינכרונים למערכת, שכל אחד מהם רץ באופן בלתי תלוי באחרים. שיטה זו נפוצה במערכות גדולות בהם יש מספר מודולים שפעולתם אינם תלויה אחד בשני. דוגמא יכולה להיות שני מחשבים, שלכל אחד מהם שעון גלובלי יחיד, המתקשרים ביניהם. מערכת שני המחשבים הינה מערכת בה שני שעונים שאינם מסונכרנים יחדיו.
לשימוש במספר שעונים הפעולים בתדרים שונים, יש את היתרון שהוא מאפשר לנו להפריד את התזמונים של מודולים שונים, וכך להפריד בעיה גדולה למספר בעיות קטנות, שניתן לפתור כל אחת מהן בנפרד.
החיסרון שנובע מהשימוש בשיטה זו היא שאנו נשארים עם מספר מודולים שאין לנו מידע לגבי קשרי הזמן ביניהם. מעבר השעון יכול לקרוא במערכת אחת לפני או אחרי שמעבר השעון במערכת שקדמה לה קרה, ויתכן מצב של מטהסטביליות.
הפתרון הנפוץ לבעיה זו, היא לתת השהייה מספיקה כדי שמצב המטהסטביליות יתייצב. בפתרון זה אנו משלמים בהאטת התקשורת בין הרכיבים השונים.
אם ידועים תדרי השעון של הרכיבים השונים, ניתן לתכנן ולצפות מתי עלולים להיות מצבי מטהסטביליות ולטפל במצבים אלו.

Controlled Asynchrony

בשיטה זו, המודולים מתחילים לפעול כאשר מסופק להם אות חיצוני המורה להם להתחיל לפעול (start). בשלב זה, הרכיב מתחיל לפעול. האות יכול להפעיל גם שעון פנימי, וככה הרכיב מבצע את הפעולה באופן מסונכרן. הרכיב פועל באופן עצמאי ולא תלוי בשאר המערכת. עם השלמת הפעולה, הרכיב יכול להוציא אות שמודיע שהפעולה הושלמה, לעצור את השעון הפנימי שלו ולהמתין לאות הstart הבאה כדי לבצע חישוב נוסף.
בעיות סינכרון נמנעות במערכת כזו, וזאת בתנאי שכל מודול מוכן לקבל באופן אסינכרוני אות start, ושהרכיבים המחוברים לאותו רכיב מסוגלים לקבל אות finish אסינכרוני. המפתח לפתירת הבעיות הוא שלאף מודול אין שעון אסינכרוני שפועל ברגעים שהוא מחכה לכניסת אותות אסינכרוניים.

מודולים ללא בקרה

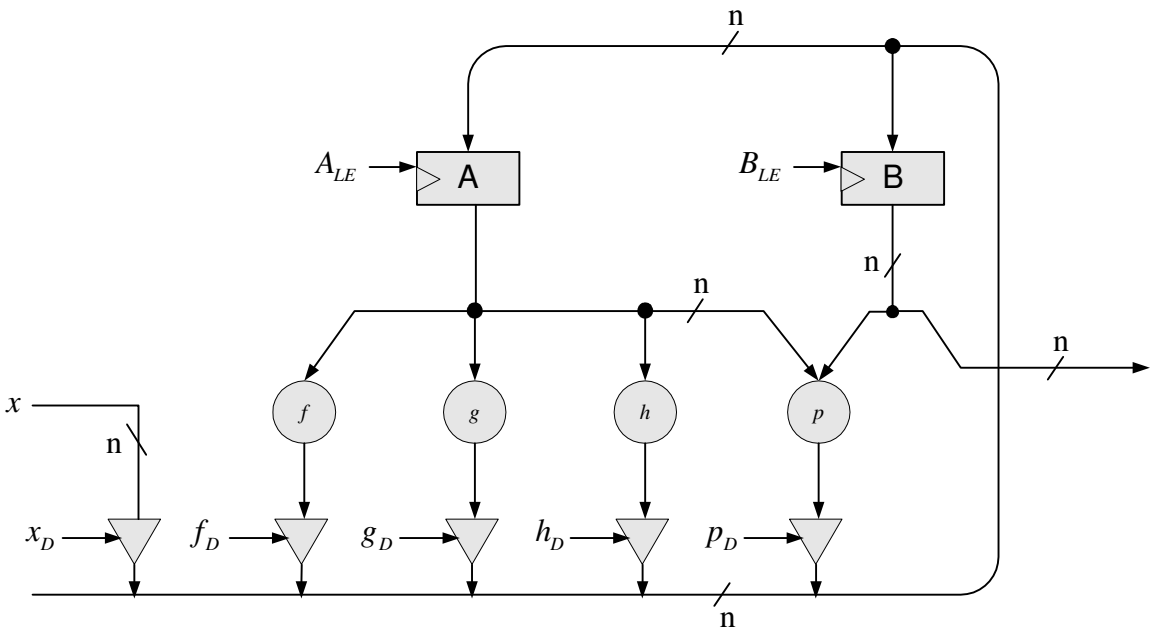
לא תמיד נרצה להשתמש במבני הבקרה שתיארנו לעיל.
נניח למשל שנרצה לחשב את הביטוי $r(x) = p(g(f(x)), p(f(x), h(x)))$ כאשר נתונים לנו מודולים נפרדים המחשבים את p, g, f, h.
נביט במעגל הצירופי הבא:



ביטוי זה מחשב את הביטוי הנדרש.
היתרונות של מימוש כזה של הפתרון הוא שאיננו צריכים אף מבנה בקרה.
אולם, פתרון כזה דורש כי המודולים השונים יהיו רכיבים צירופיים בעצמם, ובכן במידה ויש לנו מערכת המסוגלת לבצע מספר חישובים, נצטרך לבנות מודול נפרד לביצוע כל אחד מהחישובים.
כמו כן, נאלצנו להשתמש בשני רכיבים מסוג p לביצוע החישוב.
בנוסף לכך לא נוכל להפעיל את הביטוי על עצמו בצורה רקורסיבית.
לכן נעדיף לחפש אלטרנטיבות לפתרון זה.

מסלולי נתונים (Data Paths)

הפונקציונליות של המעגל הצירופי נקבעה לגמרי על ידי תכנון קווי הנתונים. רוב המערכות, בניגוד למערכות הצירופיות, מכילות קווי נתונים המסוגלים לבצע מספר פעולות שונות, בהתאם לאותות בקרה הנשלחים אליהן.
נביט כרגע במימוש נוסף לפונקציה $r(x)$.



לעומת המערכת הקודמת, מערכת זו כן דורשת בקרה.
במימוש זה אנחנו משתמשים בthree-state bus. x_D, f_D, g_D, h_D, p_D מאפשרים לנו לדאוג שבכל רגע נתון רק אחד מהאופרטור ישלח מידע אל הbus, על ידי כך שבכל רגע נתון כולם בHigh-Z למעט אחד. בעזרת הבקרה המתאימה (קביעת תדר השעון), המידע יעבור אל אחד משני הרגיסטרים, לצורך המשך החישוב.
היתרונות שהשגנו בשיטה זו: כעת אנו משתמשים רק במודול אחד מכל סוג, ולא ב2 מודולים מסוג p כפי שהיינו צריכים בפתרון הקודם. כמו כן, בעזרת הבקרה המתאימה נוכל לממש מספר פונקציות במערכת הנתונה, ולא רק את $r(x)$. שילמנו בכל שאנו צריכים להוסיף יחידת בקרה למערכת.

Precedence Relations

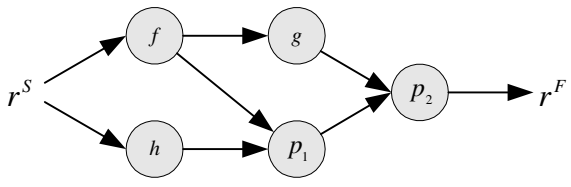
למרות שסדרות שונות של אותו מסלול יכולות לשמש אותנו על מנת לחשב את $r(x)$ קיימות הגבלות חזקות על הסדר שבו אירועים יכולים להתרחש. למשל, הפעולה של g איננה יכולה להתחיל לפני סיום הפעולה של f, מכיוון ש g צריכה את הפלט של f כנתון להתחלת פעולתה.
כדי לבטא יחסים כגון אלו בין מודולים, נשתמש בסימונים הבאים לציון אירועים: f^S, f^F, על מנת לציין את התחלת הפעולה של מודול f (f^S) ואת סיום הפעולה של המודול (f^F).
למשל, כדי לציין שg לא יכול להתחיל עד ש f מסתיים, נשתמש בסימון הבא: $f^F \preceq g^S$, האומר "f^F מקדים את g^S".

סימון: $x \preceq y$ - x קרה לא מאוחר יותר מy - "x מקדים את y".
$\preceq$ זוהי פעולה רפלקסיבית (מתקיים $x \preceq x$) וגם פעולה טרנזיטיבית ($\alpha \preceq \gamma \Leftarrow \alpha \preceq \beta, \beta \preceq \gamma$).

נסכם בטבלה את התלות של הנתונים השונים בפונקציה $r(x)$.

$r^S \preceq f$	$f \preceq g$	$g \preceq p_2$
$r^S \preceq h$	$f \preceq p_1$	$p_1 \preceq p_2$
$p_2 \preceq r^F$	$h \preceq p_1$	

ניתן לבטא את נתונים אלו בגרף, בו חץ ממודול למודול יאמר שפעולת מודול אחד קדמה לאחר.

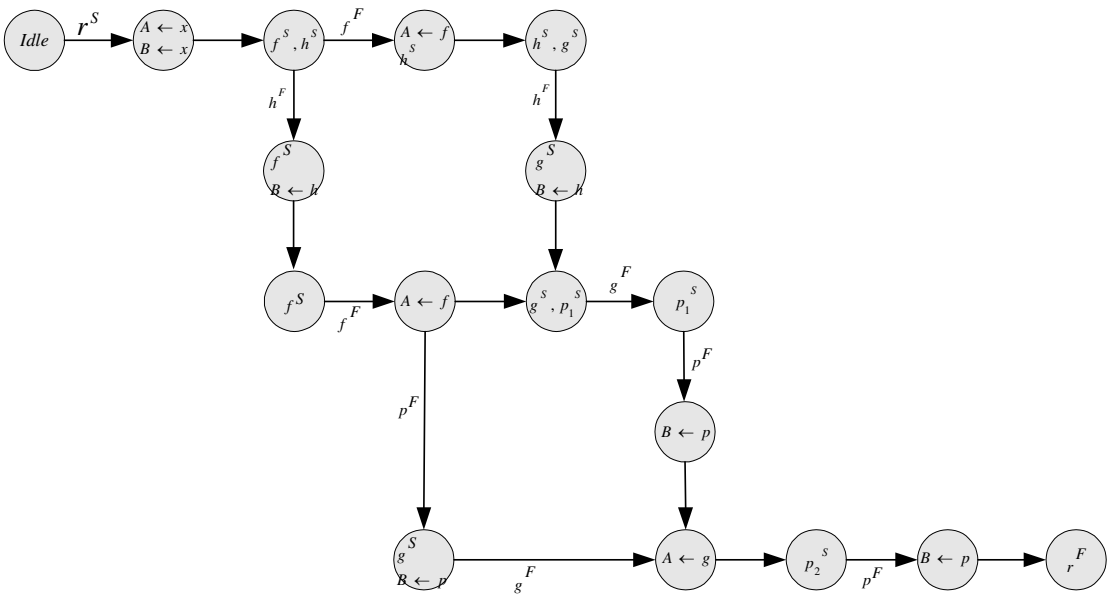


Synchronous globally timed (SGT) Control

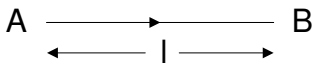
מבנים הנשלטים על ידי שעון גלובלי מתבססים על ההנחה שהזמן הלוקח לביצוע כל פעולה הוא קבוע המובנה אל המערכת. כאשר אנחנו מתכננים מערכות בשיטה זו, אנו צריכים לקחת את הזמן הגרוע ביותר שלוקח לכל פעולה להתבצע (worst case). בגלל שזמני ההתקן קבועים, הרצף וקשרי הזמנים בין האירועים קפואים בתכנון. בזמן התכנון המתכנן קובע את הסדר בו הוא רוצה שהפעולות יתבצעו, והמימוש מתבסס על סדר זה. מערכות SGT יותר פשוטות אך פחות גמישות ממערכות SLGT. אולם, פשטות זו של הSGT מאפשרת במקרים רבים לבצע אופטימיזציה לקבלת תוצאות יותר טובות מהתוצאות של מערכות דומות שימומשו בSLT.
הבקר במערכת מבוססת SGT יכול להיות מורכב מROM ומרגיסטר בלבד, שיקבעו בכל רגע מי מהרכיבים ישדר מידע, ובאיזה סדר. בכל מחזור שעון, הרגיסטר יעביר נתונים חדשים לROM וכך תבחר הפעולה הבאה לביצוע.

Synchronous locally timed (SLT) Control

כפי שכבר הוזכר, החסרונות של הבקרה הגלובלי הן חוסר היכולת לנצל את העובדה למשל, שרכיבים מסוימים מהירים יותר מאחרים, וכמו כן התעלמות מהעובדה שעבור קלטים שונים הרכיבים יכולים לסיים את הפעולה שלהם בזמנים שונים. כאשר אנו מתכננים רכיב כאשר בקרת SGT מול עיננו, אנו מתייחסים לזמנים הגרועים ביותר של כל רכיב.
נוכל לפתור בעיות אלו על ידי כך שלכל רכיב יהיה שעון משלו, ובנוסף נוסיף קווי בקרה שבעזרתם המודולים השונים יודיעו אחד לשני כאשר הם יסיימו את פעולתם. נדגים את מימוש בקרת SLT על הפונקציה $r(x)$ איתה אנו עוסקים.



בשיטות שראינו כעת, SGT וSLT, ישנן מספר נקודות אליהן צריך לשים לב:



אות איננו מתפשט במעגל במהירות אינסופית. במקרה הטוב (בוואקום) האות מתפשט במהירות c (מהירות האור).

$$t = \frac{l}{c}$$

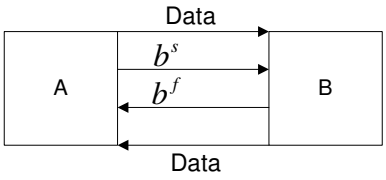
במעגלים מודפסים האותות מתפשטים למעשה בחצי ממהירות האור.
נתעניין ביחס בין זמן מעבר האות בין A לB לבין זמן מחזור השעון. בדרך כלל נרצה לדאוג שהיחס יהיה בערך $\frac{1}{10}$.

נקודה נוספת אליה צריך לשים לב היא שהשעון גם יוצר רעשים והפרעות העלולים להפריע לשאר האותות במערכת.
בנוסף גם השעון צורך הרבה הספק (כמחצית מהכמות הכוללת).

<u>שיטת ALT</u>

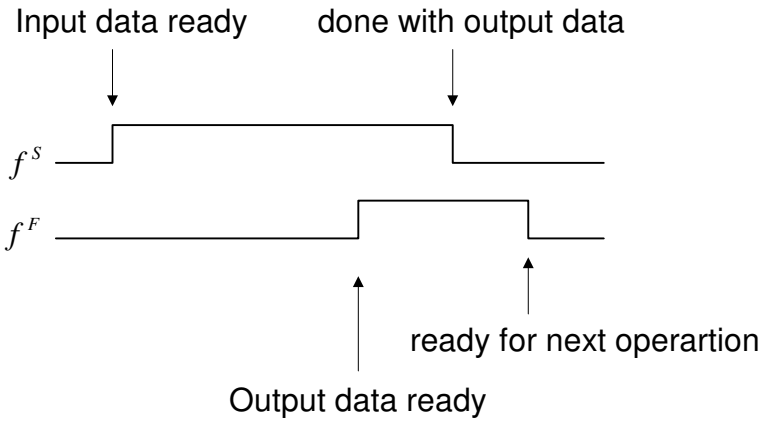
האותות נקבעים לפי עוצמתם ולא לפי מחזור שעון. במערכת כזו לא נרצה שיהיו Hazards (הם עלולים להיחשב כאותות).

בשיטת בקרה זו:



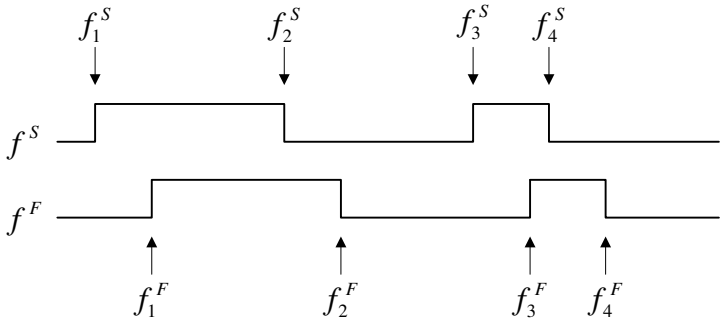
A שולח אות לB להתחיל בחישוב, וB מחזיר אות כאשר הוא מסיים.
מקובל לכנות מנגנון זה בשם "לחיצת יד" – Hand shake.

זמנים עבור שיטה זו:



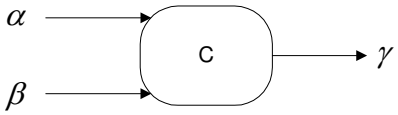
צורת פעולה זו בזבזנית למדי. עוברים 4 אותות עד להשלמת כל חישוב.

נציג דרך שניה לבצע את החישובים:
העלייה או הירידה של השעון היא זו שקובעת את הפקודה. בעליה או ירידה של המקור מודיעים על חישוב חדש ובירידה או עליה של הרכיב השני, הרכיב השני מודיע לרכיב הראשון שהוא מוכן לחישוב נוסף.

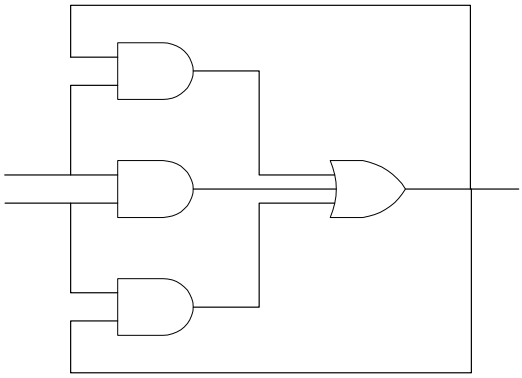


C-Element

זהו רכיב זיכרון השומר מצב כאשר הכניסות זהות.



מימוש אפשרי לC-Element:

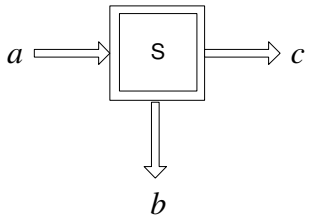


יחידות בקרה שונות

Sequence

עבור רכיב זה מתקיים:

$$a^S \preceq b^S$$
$$b \preceq c$$
$$c^F \preceq a^F$$

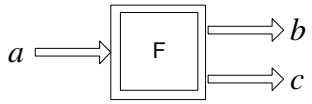


b מופעל לפני c.

Fork

עבור רכיב זה מתקיים:

$$a^S \preceq b^S$$
$$a^S \preceq c^S$$
$$b^F \preceq a^F$$
$$c^F \preceq a^F$$

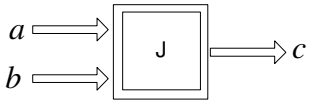


b ו c מופעלים יחדיו.

Join

עבור רכיב זה מתקיים:

$$a^S \preceq c^S$$
$$b^S \preceq c^S$$
$$c^F \preceq a^F$$
$$c^F \preceq b^F$$

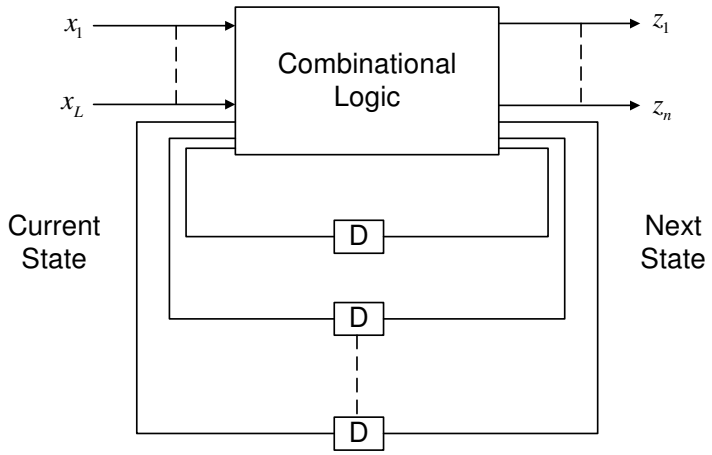


a ו b מופעלים לפני c.

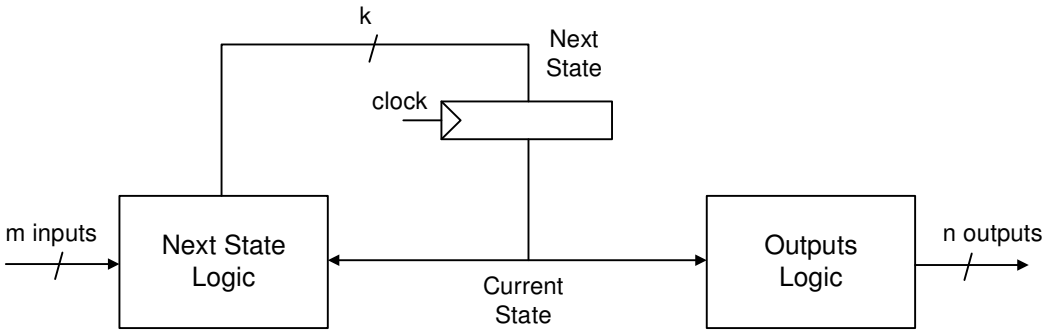
מכונות mealy ומכונת moore - עקרון פעולה

מכונת mealy

- היציאות ($z_1,...,z_n$) הן פונקציה של הכניסות $x_1,...,x_L$ ושל המצב הנוכחי.
- המצב הבא הוא פונקציה של המצב הנוכחי ושל הכניסות.



מכונת moore



- היציאות הן פונקציה של המצב הבא בלבד.
- המצב הבא הוא פונקציה של המצב הנוכחי ושל הכניסות.
- בכל מצב, ברגע שנגיע אליו, נוכל לאמר בוודאות מה יהיו היציאות. אין הכוונה שנוכל להגיד ממש אילו ביטים יצאו החוצה, אלא שנוכל לאמר למשל: "המידע השמור ברגיסטר A במערכת יצא בשלב X".

בקר מיקרו תכנות

הרעיון בבקר מיקרו תכנות הוא מימוש הלוגיקה הרצויה לנו על ידי טבלה בזיכרון. הפונקציות הלוגיות הדרושות ליצור את המצב הבא והתפוקות של המצב הנוכחי ממומשות על ידי טבלה.

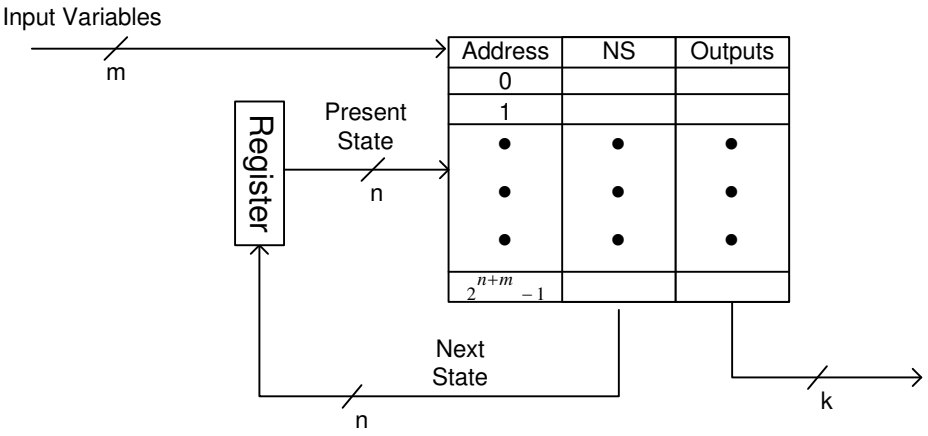
היתרונות לשיטת מימוש זו הם:

1. נוחות תכנון.
2. נוחות עדכון.
3. מודולריות.

מבנה בסיסי של בקר מיקרו תכנות

בבקר זה קיימים 2^n מצבים אפשריים. לכל אחד מהם קיימים 2^m שורות בROM המתאימות לכל אחד מ 2^m הצירופים האפשריים של m משתני הכניסה. בכל מצב יש k תפוקות המוגדרות על פי תוכן הטבלה.

מבנה זה מתאים למימוש בקר למכונת mealy היות והן המצב הבא והן התפוקות הן פונקציה של המצב הנוכחי והכניסות.



כאשר אנו ממשים FSM בעזרת ROM, אם לFSM יש n מצבים, אזי אנו צריכים $\log_2 n$ רגיסטרים במכונה.
מספר הכניסות הכללי נקבע על ידי מספר הכניסות החיצוניות פלוס מספר הFF.
(לכל FF יש כניסה). מספר היציאות הכללי נקבע על ידי מספר היציאות החיצוניות ועל ידי מספר ה-FF (סכומם).
אם שואלים מהו גודל ה-ROM הנדרש למימוש בקר, התשובה היא $2^{m+n}\cdot(n+k)$.

דוגמא

יש לממש FSM בעל 9 מצבים, 4 כניסות ו3 יציאות באמצעות ROM יחיד ורכיבי DFF. למימוש ידרשו:

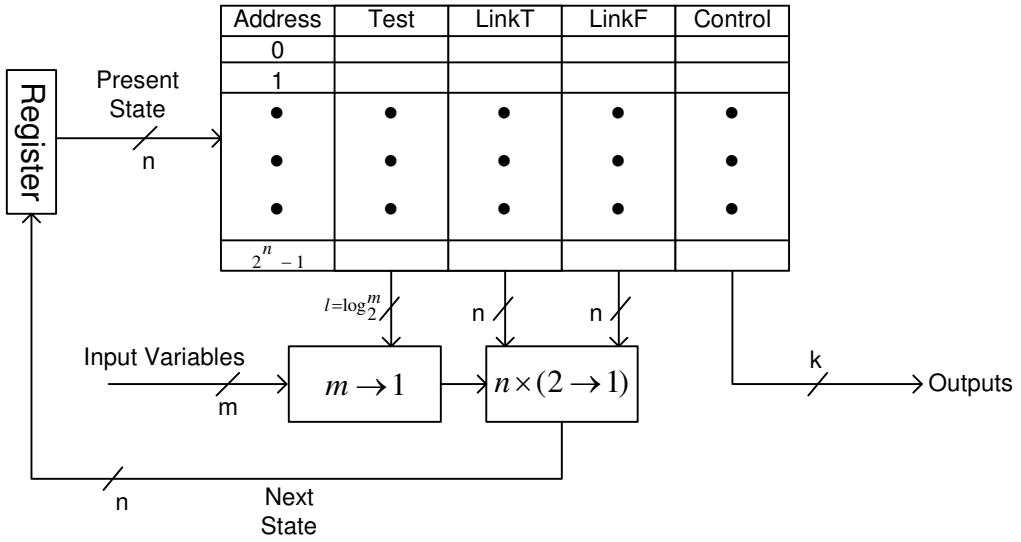
א. 4 רכיבי DFF ו-ROM בגודל $2^8 \cdot 7$
ב. 4 רכיבי DFF ו-ROM בגודל $2^7 \cdot 8$.
ג. 3 רכיבי DFF ו-ROM בגודל $2^9 \cdot 7$.
ד. 3 רכיבי DFF ו-ROM בגודל $2^7 \cdot 9$.

פתרון

מכיוון שיש לנו תשעה מצבים, נצטרך $\lceil \log_2 9 \rceil = 4$ רכיבי DFF. מספר הכניסות הכולל הוא 4 פלוס 4 כניסות נוספות עבור הDFF, ומספר היציאות הכולל הוא 3 פלוס 4 יציאות עבור ה-DFF, ולכן נצטרך 4 רכיבי DFF ו-ROM בגודל $2^8 \cdot 7$.

בקר מיקרו תכנות עם שדות LinkTrue וLinkFalse

נרצה לחסוך בגודל ה-ROM, ולכן נציע מבנה בסיסי אלטרנטיבי לבקר מיקרו מתוכנת.
בשיטה זו נתייחס לטבלה הנמצאת בזיכרון כטבלה בעלת 4 שדות:

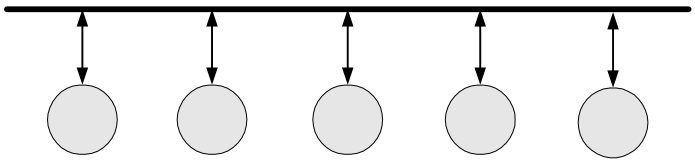


סכימה זו מאפשרת מעבר מכל מצב לאחד משני מצבים המוגדר על ידי שדות LinkTrue וLinkFalse. בכל מצב מוגדר שדה Test הקובע את אות הכניסה שייבדק במצב זה. אם הוא 1 אזי המצב הבא מוגדר על ידי LinkTrue ואם הוא 0 אז על ידי LinkFalse. תפוקות המערכת בכל מצב יהיו שדה הControl.
מבנה זה מתאים למימוש מכונות Moore כיוון ששדה התפוקות הוא פונקציה אך ורק של המצב הנוכחי.
בעיה הקיימת במימוש זה היא ש-Test קובע רק סיגנל בודד שייבדק, ולכן אם אנחנו צריכים לקבוע את המצב הבא כפונקציה של שני סיגנלים או יותר, לא נוכל להשתמש בבקר מסוג זה.

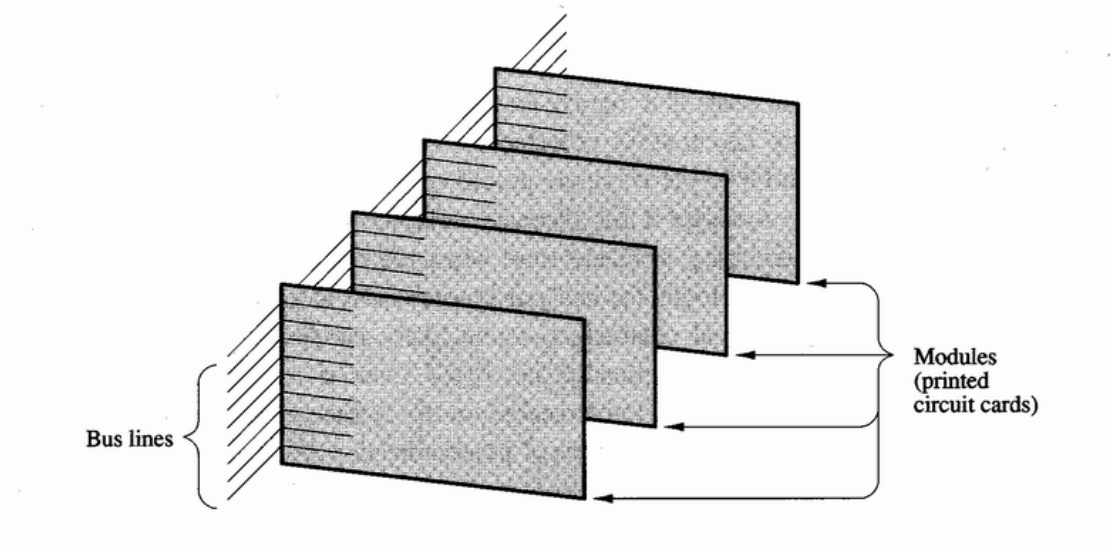
תקשורת במערכת ספרתית

בפרק זה נניח שיש בידינו מערכת עם n מודולים נפרדים, שכל אחד מהם מסוגל לקבל קלט ולהוציא פלט מזמן לזמן. כל output יכול לשמש כ-input של אחת מהיחידות האחרות.
בפרק זה נעסוק בתקשורת בין המודולים הנפרדים השונים.

Shared Bus



עורק משותף (Shared Bus) זוהי חבילה של קווים מוליכים אשר כל היחידות במערכת מחוברת אליה.

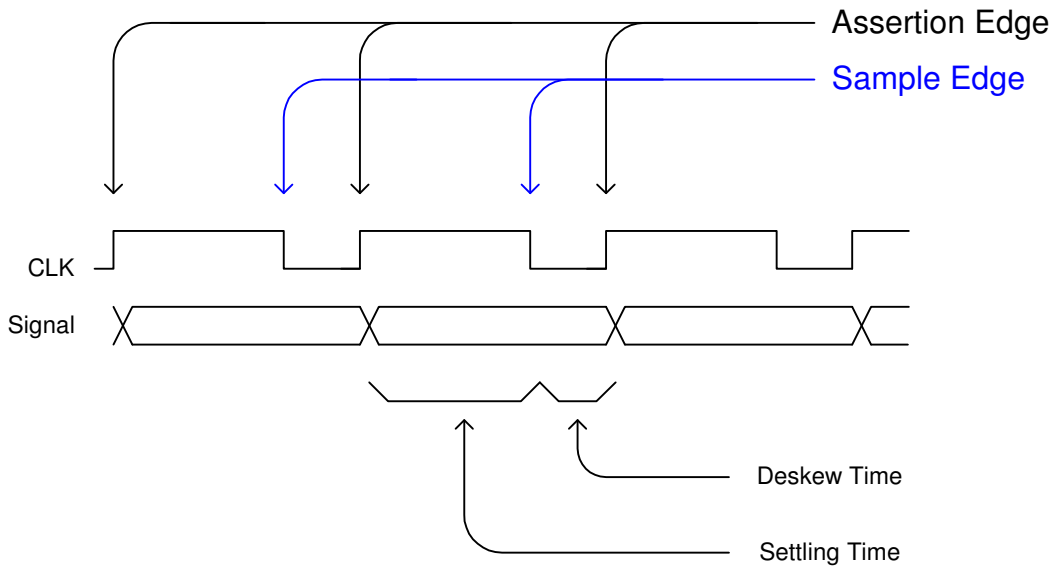


ראינו כבר קודם לכן בקורס bus, שהיה data bus טהור. היו בו רק קווי data. בbus איתו נתעסק כעת יהיו קווי data וגם קווי בקרה.

שיהוי חדש הקשור לbus הינו t_{bus}. שיהוי זה נוצר על ידי העורק והוא איננו הזמן שלוקח לאות לעבור אלא כמה פעמים (5-10) שהאות עובר. שיהוי זה מוסף לצורך הפשטה. בפועל מה שקורה הוא שאנו מחכים עד שהאות ידעך ולכן מתווסף שיהוי זה.

t_{bus} - הזמן שצריך בשביל שאות שמודול מסוים שלח לbus יהפוך לתקף ותקיף עבור שאר המודולים.

תזמונים בbus



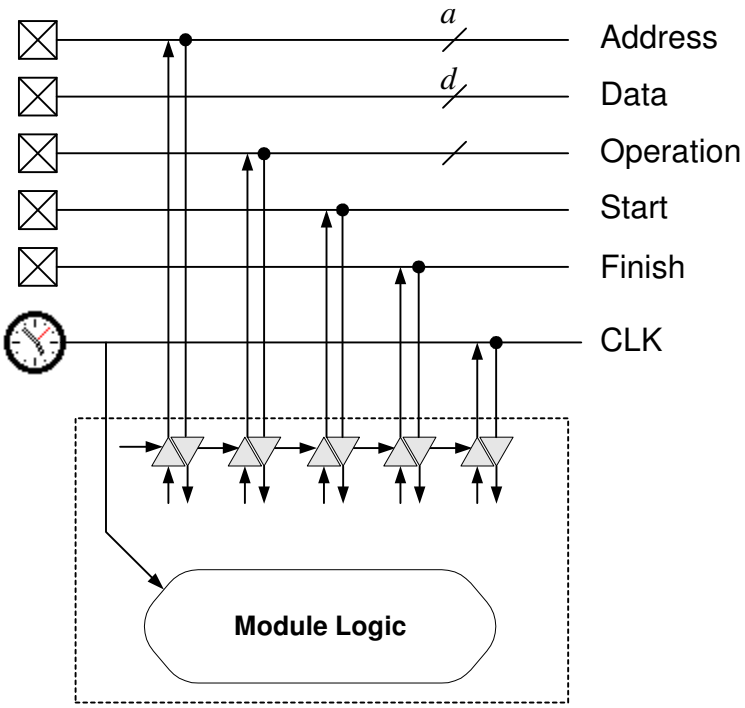
בשרטוט אנו רואים את הקשר בין מעברי השעון ומעברי האותות על עורק מתוזמן.
קו bus בודד שמסומן CLK נושא את אות השעון המשותף ומופעל במחזוריות.
בניגוד לשיטות התזמון שראינו עד כה, בהן רק אחת משפות השעון הייתה פעילה, במקרה זה בשתי השפות קוראים תהליכים. קווי האותות מסונכרנים עם עליית השעון החיובית. Assertion Edge מסמן את הזמנים היחידים בהן התקנים יכולים להיות מופעלים או מופסקים. קווי האותות נדגמים על ידי כל מודול עם ירידת השעון. מעבר הדגימה צריך לבוא לפחות t_{bus} אחרי הAssertion Edge.
Assertion Edge קורא אחרי Deskew time שמיועד לתקן שגיאות שנובעות בהבדלים בזמני השיהוי בין האותות לשעון.

קווי bus אופייניים

הגדרות:

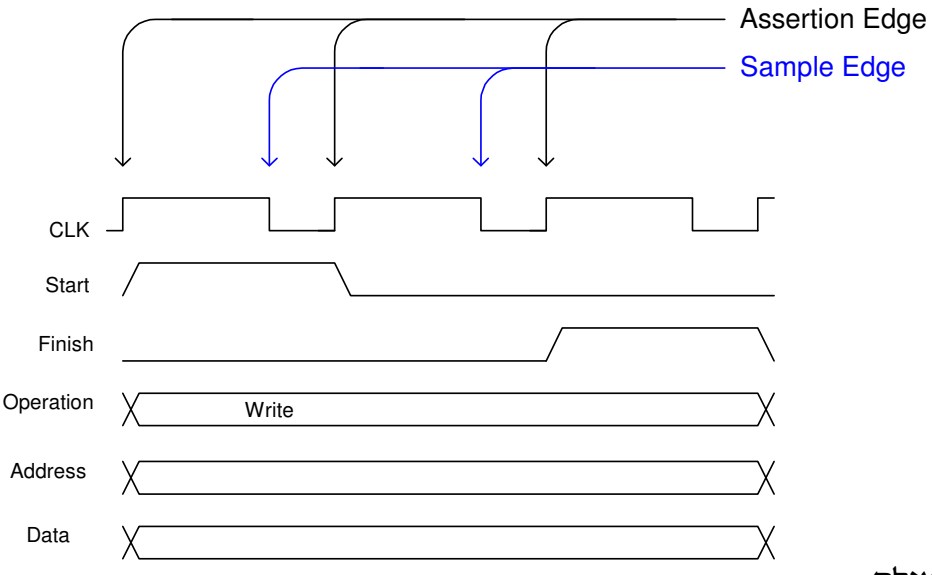
MASTER - הרכיב ששולח הוראות לרכיבים אחרים ומבקש מהם נתונים.
SLAVE - הרכיב שמקבל ומבצע הוראות.

נגדיר כתיבה (**WRITE**) כהוראות (נתונים) שהMASTER שולח לSLAVE.
נגדיר קריאה (**READ**) כנתונים העוברים מהSLAVE לMASTER.



ב-bus ישנם קווי Address - a קווים, ומכאן שמרחב הכתובות הוא 2^a - מרחב המענים שאליהם יכולים להגיע הנתונים.
Operation - בקווים אלו מגדיר הMASTER איזו פעולה הוא רוצה - קריאה/כתיבה וכו'.
Start - ה-MASTER מודיע ל-SLAVE להתחיל לעבוד.
Finish - ה-SLAVE מודיע על סיום העבודה.
ב-Data ה-MASTER שם את הנתונים לכתיבה, או ה-SLAVE שם את הנתונים שהוא מחזיר.

ה-MASTER שולח את כל האותות בעליית השעון. עם ירידת השעון ה-SLAVES דוגמים את האות ורואים מה צרים לעשות.



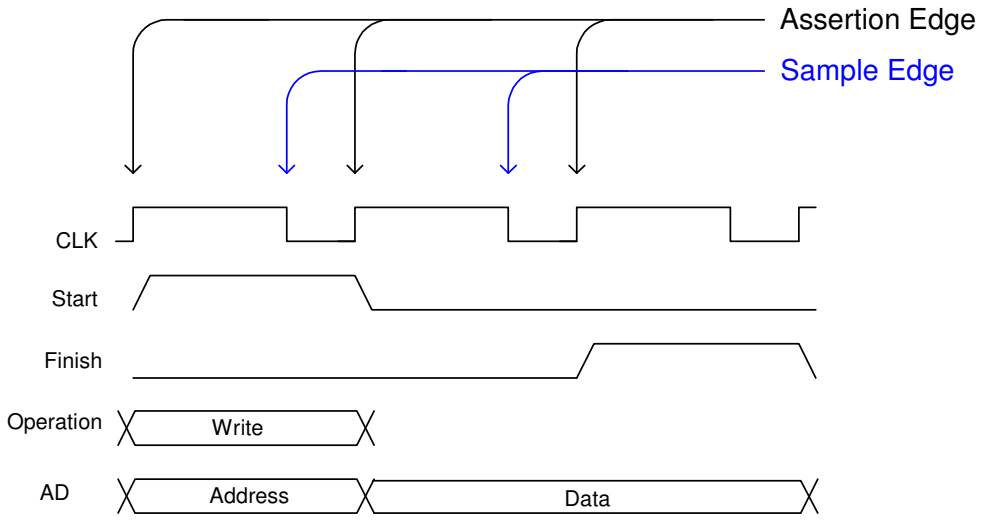
אופן פעולה

תחילת פעולה - MASTER מעלה Start, הנחיית Operation, העלאת Address ו-Data על ה-BUS.
דגימה - ה-SLAVES דוגמים בירידת השעון ומי שהכתובת מתאימה לו קולט את ה-DATA.

הערות:

- ברגע ה-MASTER קיבל Finish הוא ממשיך בפעולתו.
- ה-Slave יכול להשהות את קו ה-Finish כמה שירצה, עד שיגמור את פעולתו.
- אומנם Address ו-Data נשלחים בו זמנית, אך ה-Slave קודם מזהה כתובת, לראות אם מדובר בו, ורק אז קורא את הנתונים.
- לכל Slave יש טווח כתובות משלו והכתובת המתאימה (אשר כוללת בתוכה באיזה מודול מדובר) מתקבלת רק במודול המתאים.

ניתן להציע שיפור מסוים ל-bus שהרגע הצענו.
נכתוב את ה-Address וה-Data באותו הקו.
במחזור השעון הראשון הMASTER שולח כתובת, ואילו הSLAVE מגלה שפקודה עומדת להישלח אליו.
במחזור השעון השני ה-MASTER שולח את ה-DATA וה-SLAVE מקבל ועובד.



bus כזה נקרא multiplexed bus.

Watch-Dog

בשיטה שתיארנו כעת לאופן פעולתו של הbus קיימות מספר בעיות. אחת מהן היא המקרה שהMASTER שולח הוראה לSLAVE שלא קיים, או לSLAVE שהתקלקל ואיננו מסוגל לתפקד.
נוצרת בעיה, כי הMASTER מחכה לאות Finish לפני שהוא שולח הוראות חדשות, ואות Finish איננו מגיע.
Watch-Dog זוהי יחידה ששולחת אות Finish במקרים כאלו על מנת למנוע מה-bus להיתקע. ה-Watch-Dog מתחיל לפעול כאשר נשלחת הוראה לאחד מהיחידות במערכת.
לא נרצה שה-Watch-Dog ישלח אות Finish במידה והיחידה כן תקינה, אולם החישוב של הנתונים לוקח זמן, ולכן נדרוש שהזמן עד שהWatch-Dog ישלח אות Finish יהיה ארוך יותר מזמן החישוב הארוך ביותר של כל אחד מהמודולים האחרים.
הערה: ה-Watch-Dog שולח Error חזרה ל-MASTER במקרה שעבר הזמן המוקצב.

Memory mapped I/O

במערכות מחשב מבוססות עורק משותף, נהוג לא פעם להתממשק אל התקני קלט/פלט בעזרת מיפויים במרחב הזיכרון.
כתובות מסוימות במחרב הכתובות הכללי משויכות לאחד או יותר רגיסטרים פנימיים של התקן קלט פלט ממופה זיכרון. צורת הפניה לכתובות אלו זהה לפניות לזיכרון, דבר המקל על המשתמש/מתכנת.
בדרך כלל מרחב הכתובות הכללי מחולק לאזורים הנבדלים בעזרת מספר מסוים של סיביות בצד הMSB, למשל:

ROM	0x00000000-0x000fffff
Unused	0x00100000-0x00ffffff
SRAM1	0x01000000-0x01ffffff
SRAM2	0x02000000-0x02ffffff
Unused	0x03000000-0x07ffffff
DRAM	0x08000000-0x0fffffff
I/O 1	0x10000000-0x1000ffff
I/O 2	0x10010000-0x100100ff
Unused	0x10010100-0xffffffff

פענוח הכתובת מורכב משתי פעולות:

- זיהוי אזור/התקן (מתבצע בעזרת חומרה חיצונית).
- זיהוי כתובת בתוך ההתקן (מתבצע בד"כ על ידי ההתקן עצמו).

זמני מעברי הנתונים

כאשר אנו באים להחליט על הזמנים בהם יעברו הנתונים בין הMASTER לSLAVES, אנו יכולים לתקוף את הבעיה במספר שיטות. שם כללי לשיטות השונות הוא Memory mapped I/O.

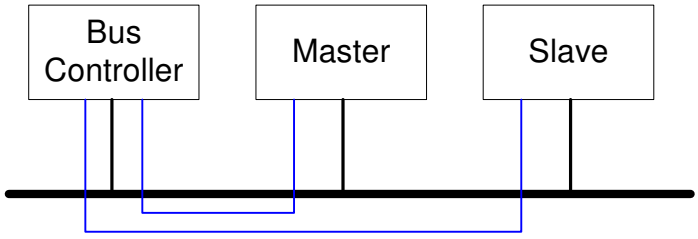
שיטה 1 - "אין תגובה"
יחידת הSLAVE משהה את התשובה שלה (לא שולחת אות Finish) עד שהיא מסיימת את החישוב.
שיטה זו היא הפשוטה ביותר, אולם היא בזבזנית, מכיוון שכל המערכת מחכה עד לתשובה.
מעט מאוד עורקים משתמשים בפועל בשיטה זו.

שיטה 2 - "polling"
יחידת הSLAVE שולחת אות Finish, ובמקבל מחזירה אות המסמל "אין תשובה עדיין". הMASTER שואל את הSLAVE פעם בכמה מחזורים, האם החישוב כבר הסתיים.
שיטה זו מחייבת את הMASTER להתעניין במה שקורה בSLAVES, גם אם אין להם בהכרח את הנתונים הדרושים. שיטה זו לכן גורמת לכמות מסוימת של תעבורה מבוזבזת.

שיטה 3 - "מנגנון פסיקות (interrupts)"
מנגנון המאפשר לMASTER לדעת כאשר הנתונים מוכנים בSLAVES, כך שלא מתבצע משאל מיותר, אלא הMASTER קורא את הנתונים כשהם מוכנים.
שיטה זו היא השיטה הממומשת במרבית העורקים.

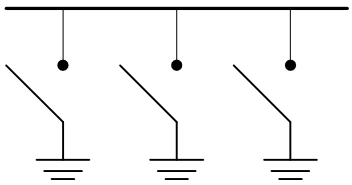
Mastership

נרצה כרגע להרחיב את המודל עליו אנו מדברים.
למרות שיתכן שתהיה מערכת בה יהיה MASTER יחיד קבוע שישלוט בכל המערכת, עורקים מודרניים מאפשרים לשליטה לעבור מרכיב לרכיב, כך שכל הרכיבים יוכלו לבקש שירותים מרכיבים אחרים.
בעורקים כאלו בדרך כלל כל הרכיבים נחשבים לMASTERS פוטנציאלים.
לפני כל פעולה בעורקים כאלו, קיים מנגנון שבוחר מי יהיה הMASTER עבור סידרת הפעולות הבאה ויקבל את השליטה על העורק.
נוסיף למערכת שלנו יחידה נוספת - "Bus Controller", שתיקבע איזה מהרכיבים יקבל בכל רגע את השליטה.

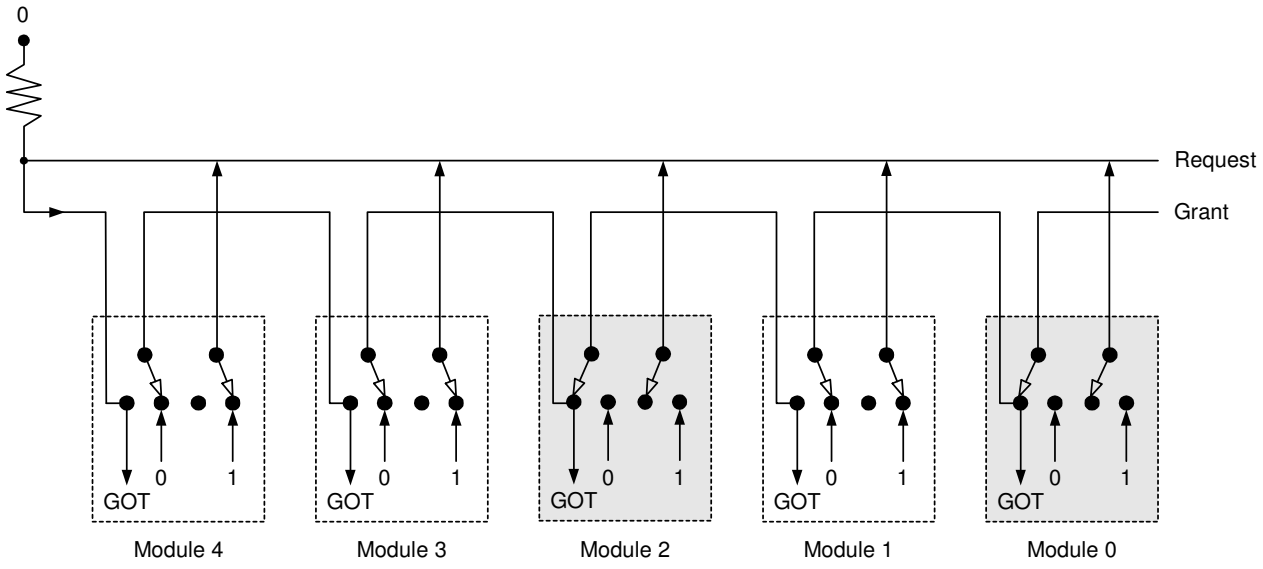


<u>Daisy Chain</u>

בשיטה זו נוסיף לbus שני קווים. נוסיף קו שנקרא Bus Request, שיחובר בצורה דומה לקווי הבקרה שראינו עד כה, ונוסיף קו grant שיהיה "שזור" בין הרכיבים (מכאן שם השיטה). קו הgrant נכנס בכל יחידה אל $grant_{IN}$ ויוצא דרך $grant_{OUT}$.
בצורה זו אנו מאפשרים לכל מודול לקבוע האם להעביר את אות הgrant הלאה אל היחידות הבאות, או לעצור אותו, בהתאם לשאלה האם המודול צריך באותו זמן את הbus או לא. כאשר יחידה לא שולחת בקשה, הgrant מועבר הלאה.
קו הRequest פעיל בנמוך. הסימונים בלוגיקה הם לכן 1 כשהקו מחובר לאדמה ו0 אחרת. כל יחידה מחברת את הRequest עם מתק לאדמה.



ה-bus יהיה 0 רק אם אף יחידה איננה רוצה את העורק.



Daisy Chained Bus

GOT - בעזרתו היחידה בודקת האם היא הבאה שמקבלת את הbus.
בשרטוט 3 יחידות ביקשו את הbus על ידי העברת המתג ימינה.

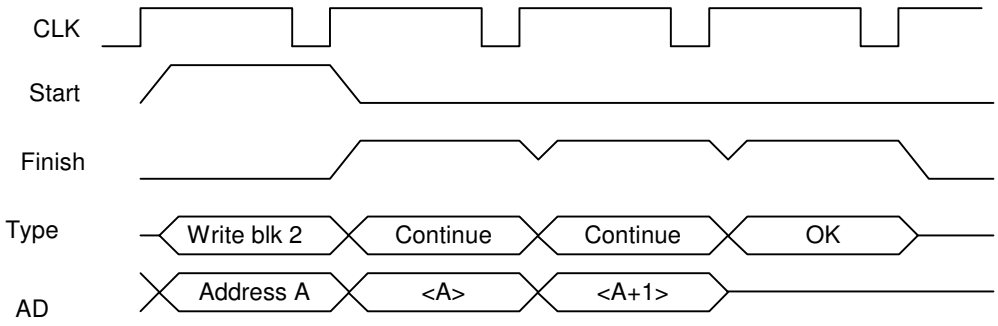
יחידה איננה מתחילה להשתמש בbus ברגע שקיבלה הודעה שהיא הבאה בתור להיות הMASTER. היחידה עוקבת אחרי הקווים של הbus, מחכה לכך שהפעולה שמתבצעת באותו רגע תסתיים, ואז היא שולחת Got לרכיב הבא, ומתחילה לעבוד.

בשיטת Daisy Chain ליחידה השמאלית ביותר יש קדימות, והבאות אחריה יקבלו את הקו רק אחרי שהיא תסיים את פעולותיה.
3,4 מודולים יכולים לסתום את הbus אם הם מהירים (מבקשים בקשות כל הזמן).
דוגמא לתרחיש : מודול 4 ביצע פעולה, ולאחריו 3 ביצע פעולה ומעביר למודול 1 הודעה. באותו רגע מודול 4 מבקש את העורק, ועקב עדיפותו מקבל את העורק.
כדי לפתור את בעיה זו, נצמצם את יכולת כל יחידה להעביר מתגים ימינה.
נאמר שיחידה יכולה לעשות request רק במחזור בו קו הrequest גבוה (0 לוגי).
כך במחזור אחד, מספר רכיבים מבקשים את העורק. רק לאחר שהבקשות של כולם נענו, יהיה מחזור נוסף בו רכיבים יוכלו לבקש את העורק.

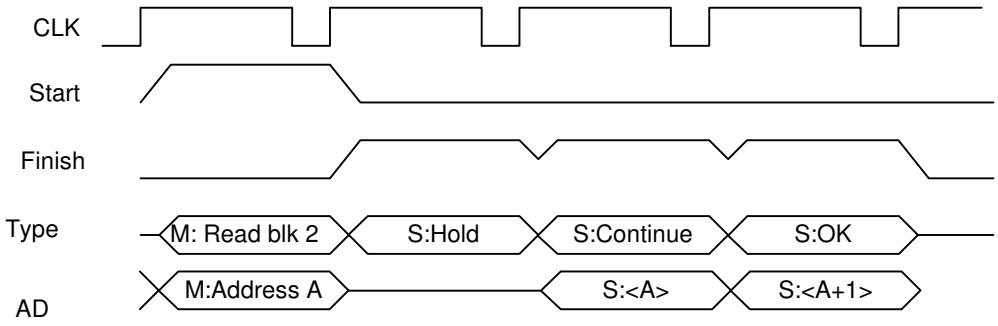
נרצה להיות מסוגלים להעביר על העורק יותר מבית אחד על ידי פקודה בודדת.
נרצה לעשות זאת מכיוון שנרצה לנצל יותר טוב את הbus. אם אנו צריכים להעביר, למשל, סידרה ארוכה של בתים, ועבור כל בית שאנו שולחים אנו צריכים חמישה מחזורי שעון, אנחנו מבזבזים זמן רב.

על מנת שנהיה מסוגלים לבצע העברה של מספר בתים, אנו צריכים מספר דברים:

1. להיות מסוגלים להעביר את כתובת ההתחלה וכתובת הסיום של הבלוק, או לחילופין להיות מסוגלים להעביר את כתובת ההתחלה ואת גודל הבלוק.
2. להוסיף פקודות לרשימת הפקודות שהיחידות מכירות: קריאה של שני בתים, ארבעה בתים, פקודה לכתיבת שניים או ארבעה בתים.



תהליך כתיבת בלוק לזיכרון

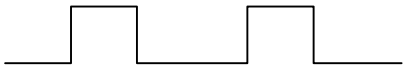


תהליך קריאת בלוק מהזיכרון

פירוש הפעולות:
S:Hold - האות עוד לא מוכן
S:Continue - האות מוכן
S:OK הפעולה הצליחה
<u>תקשורת טורית</u>

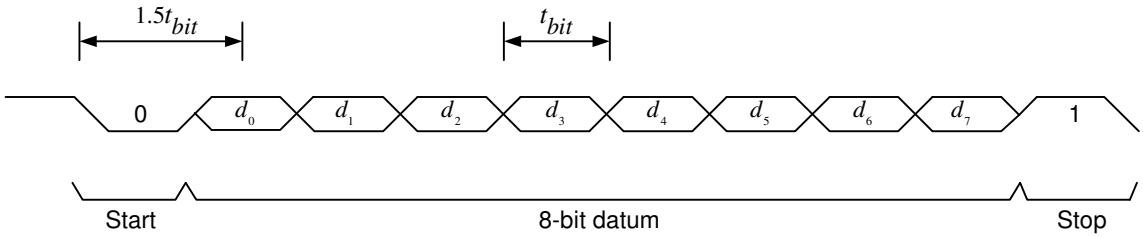
כאשר אנו רוצים לקשר בין מודולים הנמצאים במרחקים גדולים אחד מהשני, ייווצר מצב בו נרצה למזער את מספר הקווים העוברים בין יחידות התקשורת (עקב עלויות). נוכל לצמצם את קווי הבקרה השונים על ידי הגדרת מחרוזות נתונים שישלחו על קו בודד, שמשמעותן תהיה המשמעות של קווי הבקרה השונים - למשל, רצף אותות מיוחד עבור Start, Finish וכו'.
נוכל להגיע למצב שבידנו רק קו בודד, דרכו כל המידע עובר, ללא אף קו בקרה.
כאשר אנו מעבירים מידע בתקשורת טורית, אין שעון המתזמן את הנתונים (כי הרי אנו מעבירים רק קו בקרה אחד, ושעון היה דורש קו בקרה נוסף).
המפתח להעברת המידע הוא שהמודולים המתקשרים ביניהם יסכימו על תדירות שעון מסוימת ויהיו מסוגלים לייצר לעצמם אות שעון בתדירות קרובה לתדירות המוסכמת.

המקלט מקבל את האות וצריך להבין מה התקבל.
למשל:



000 11 000000 11 0000

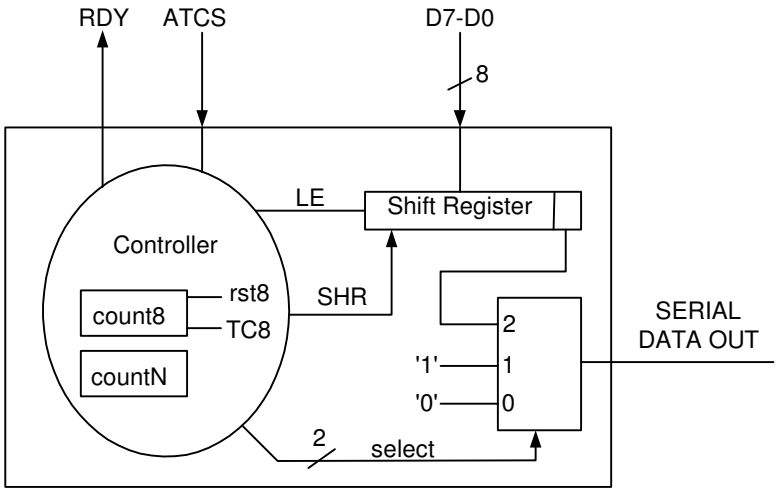
המקלט צריך לקבוע על סמך האות המתקבל מתי ובאיזה קצב לדגום.
תהליך קביעה זה נקרא self timing.
נציג פרוטוקול המשתמש בשיטה זו.
פרוטוקול זה הוא UART - universal asynchronous receiver transmitter.
בפרוטוקול זה, כל 8 ביטים נשלחים כרצף של 10 ביטים: הביט הראשון נקרא start bit, וערכו תמיד 0. בסוף 8 הביטים נשלח ביט נוסף, שהוא תמיד 1, הנקרא stop bit.



t_{bit} הוא הזמן לשליחת ביט בודד.
$\frac{1}{t_{bit}}$ זהו רוחב השידור.

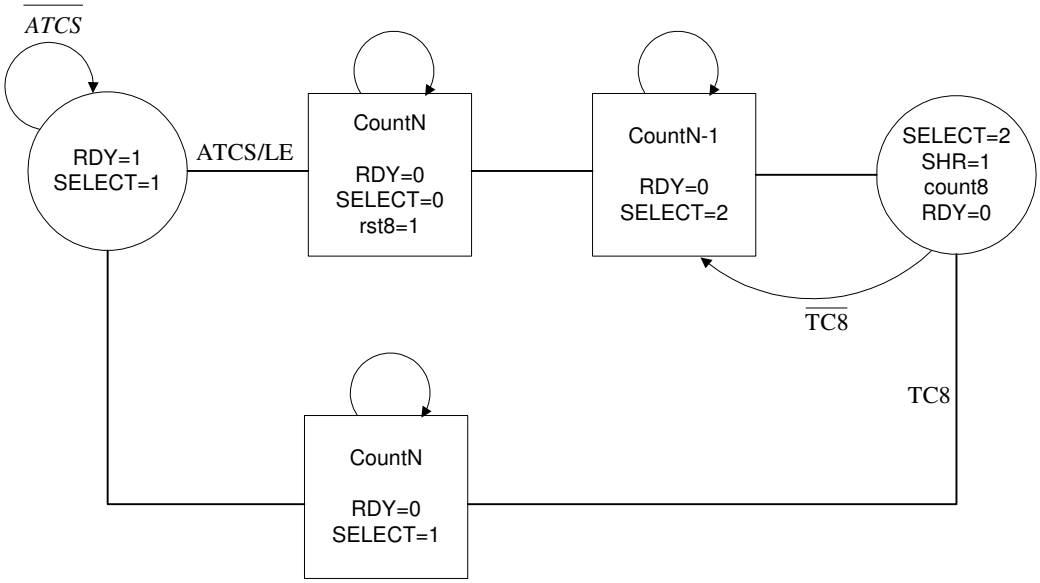
הבית משודר הפוך - d_0 הוא ה-LSB ו- d_7 ה-MSB.

תאור המשדר הטורי האסינכרוני (AT)

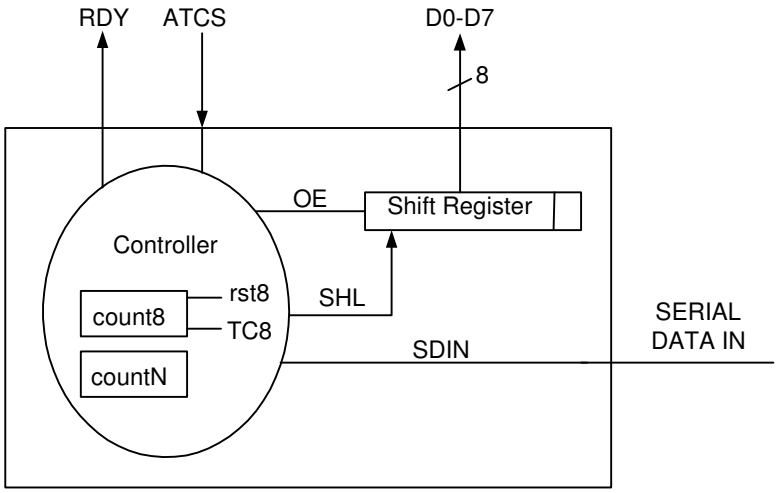


תדר הערוץ הינו תדר השעון של המערכת מחולק בN.

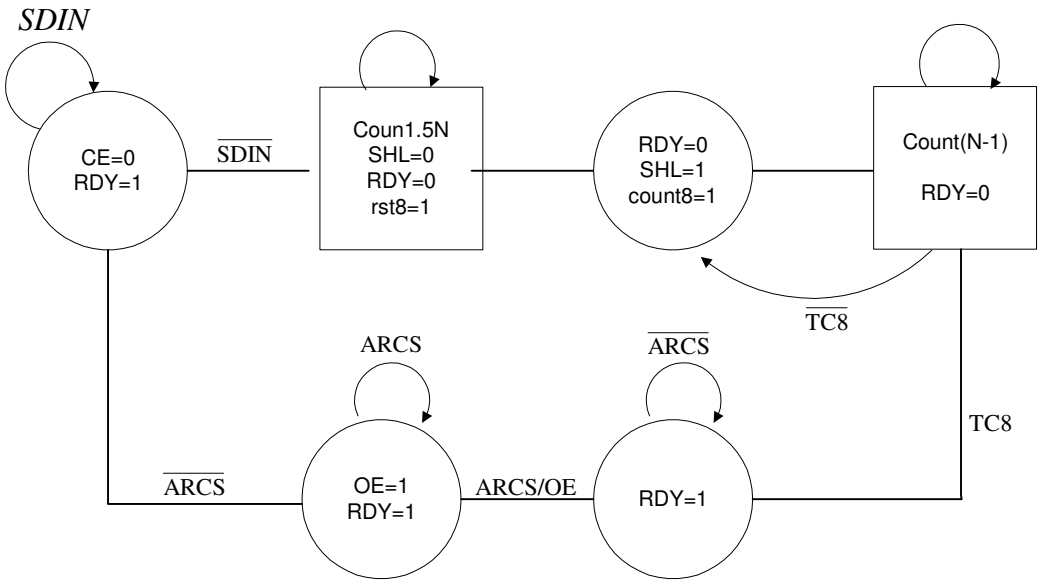
המשדר מוציא RDY=0 כאשר הוא במצב idle, וRDY=1 כאשר הוא מוכן לשדר.



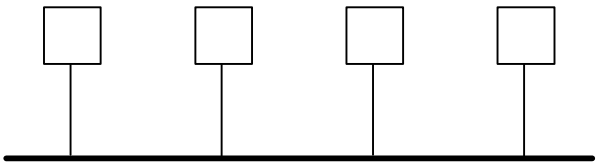
תיאור מקלט טורי אסינכרוני (AR) במערכת דומה



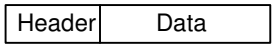
הערה: במימוש המתואר נקבל באוגר ההזזה את D0 משמאל ואת D7 מימין.
SDIN=0 משמע START של שידור טורי המגיע למקלט.
יש לחכות 1.5N כדי לדגום במרכז כל ביט שניקלט (לאחר הזמן שאנו מחכים בהתחלה, אנו ממשיכים לדגום כל 1N את האות).
RDY=1 כאשר המקלט מוכן לקלוט מידע.
לאחר שנקלטו שמונה בתים, אנו מצפים לקבל 1 לסימון סוף הקלט.



LAN - Local Area Network



נתמקד בשיטה משנות השמונים בשם Ethernet.
בשיטה זו כל הרכיבים מחוברים לקו אחד. בכל רגע נתון ניתן להעביר רק הודעה אחת על הקו. מעבירים את האינפורמציה בצרורות (packets) על הקו, בצורה



.

איך נמנע משתי יחידות לשדר בו זמנית?
כל יחידה הרוצה לשדר, בודקת לפני שהיא מתחילה לשדר שאף יחידה אחרת לא משדרת כרגע. (Carrier Sense). אם שתי יחידות מתחילות לשדר בו זמנית, הן עוצרות, ובודקות שוב את העורק אחרי זמן אקראי.

שמות נוספים לשיטה זו:
Carrier Sense, Multiple Access, Collision Detector

ראינו מספר הפשטות עד כה:

FSM
↓
Logic Circuits
↓
Logic Gates
↓
Transistors

מעל לכל רמות הפשטה אלו ישנה רמת הפשטה נוספת, והיא שפות עליות.

בשפה עילית מתכנת יכול, למשל, לבצע פעולת הכפלת מטריצות. הוא מניח שקיימת מכונה ("מכונה וירטואלית") שיודעת לבצע פעולות כגון כפל מטריצות.

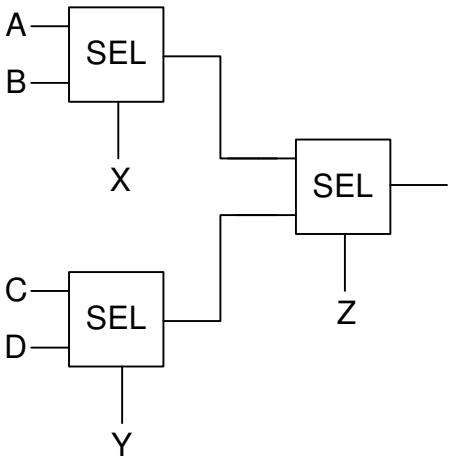
נרצה להשלים את הפער בידע בין רמות ההפשטה הנמוכה לרמת ההפשטה הגבוהה.

דרך 1 - אינטרפטציה

לוקחים כל בלוק ברמת ההפשטה הגבוהה והופכים אותו לבלוקים מקבילים ברמת ההפשטה הנמוכה יותר.

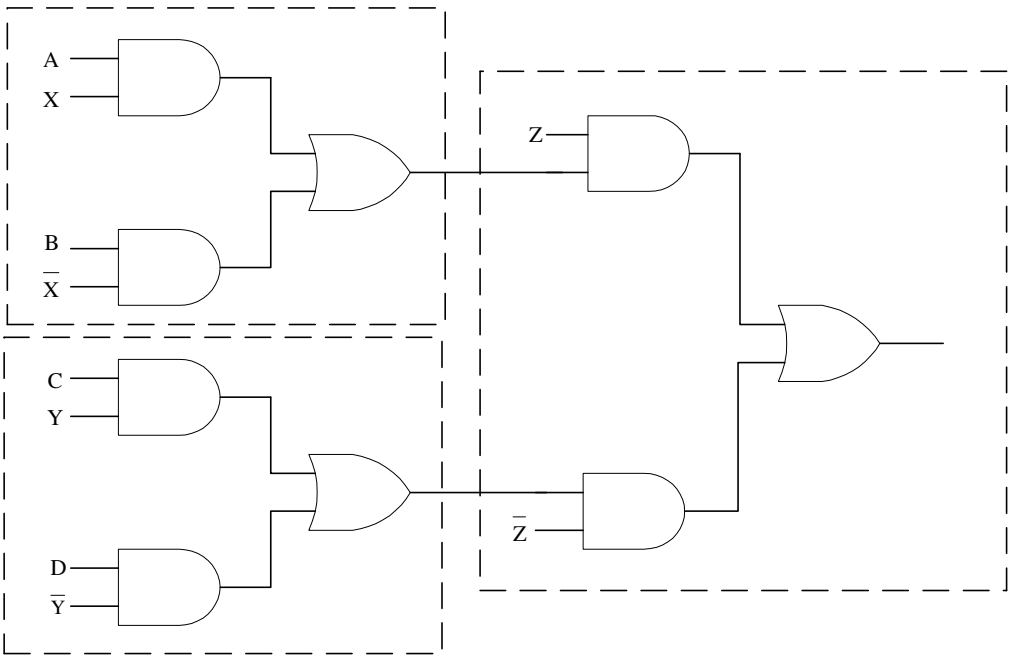
דוגמא

נניח שנתון המעגל הבא:



נרצה לרדת רמת הפשטה:

נחליף כל בלוק SEL באוסף שערים המבצעים את אותה הפעולה.

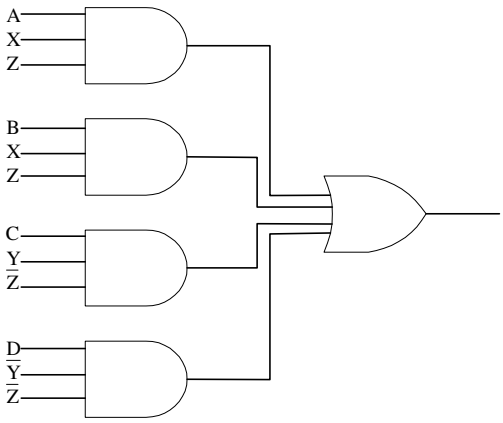


<u>דרך 2</u>

עוברים מהפשטה גבוהה לנמוכה אך לא שומרים על המבנה של הרמה הגבוהה.

<u>דוגמא</u>

נתרגם את אותו מעגל מהדוגמא הקודמת, בלי לשמור על המבנה שלו:

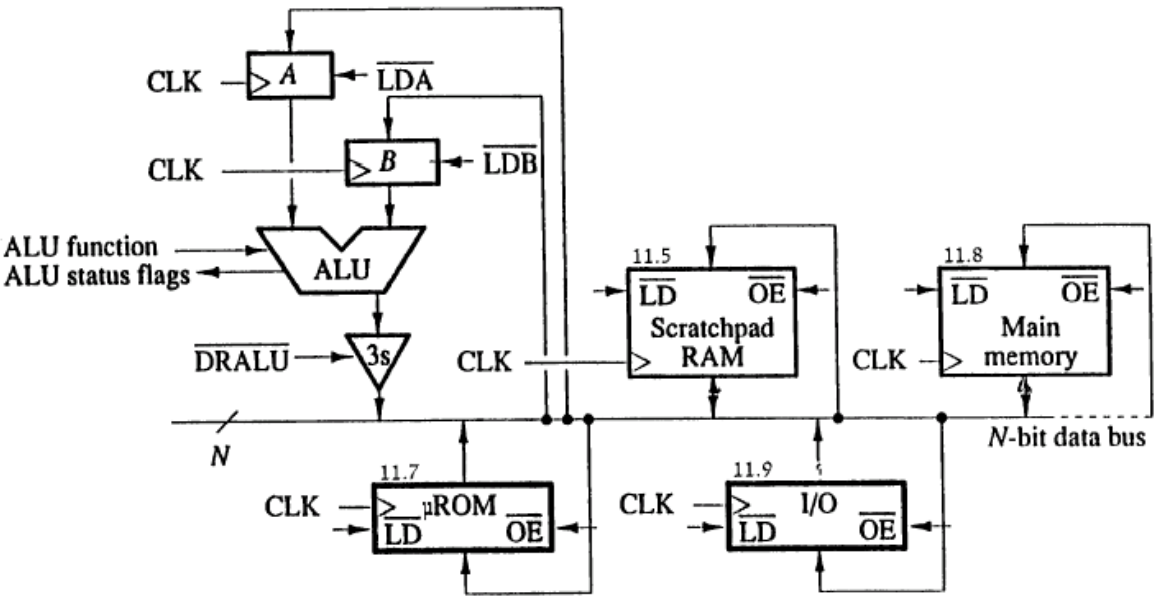


MAYBE

ארכיטקטורת המיקרו אינטרפרטר

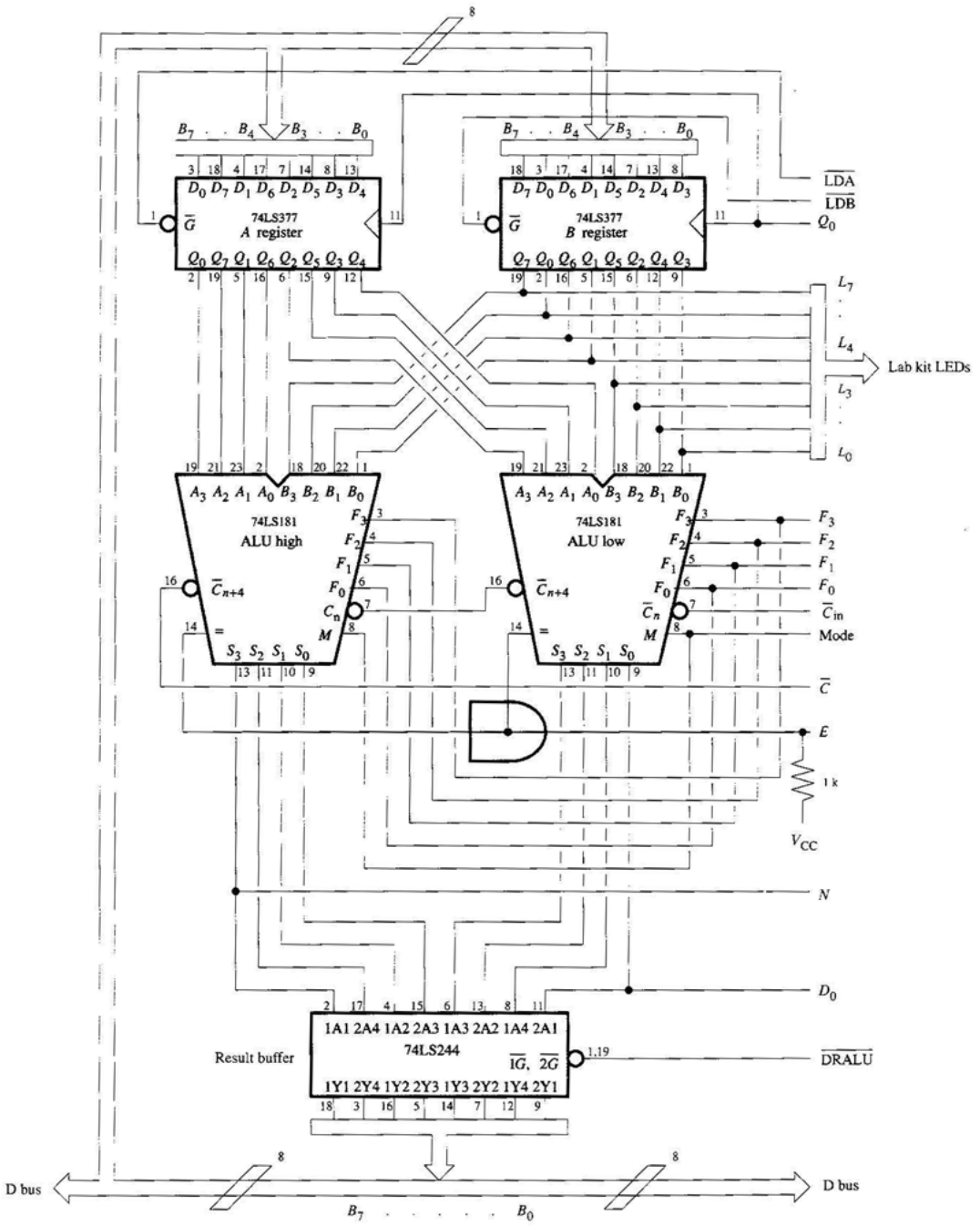
מעבד (CPU) ביחידת מחשב כללית הוא אינטרפרטר המפענח את שפת המכונה, שהיא הוראות מכונה (machine instructions) המקודדות בבינרית, האומרות כיצד תוכנית צריכה לרוץ.
כדי להבין איך האינטרפרטר המפענח את שפת המכונה עובד, נתעסק עם אינטרפרטר פשוט בהרבה של שפה פרימיטיבית הנקראת מיקרוקוד. נראה כיצד מתכנתים אחר כך את האינטרפרטר של שפת המכונה בעזרת אינטרפרטר זה.
כל פקודה בשפת המכונה תתורגם למספר microinstructions.
אנו נעבוד בעזרת microassembler, שזוהי תוכנה הלוקחת קוד שניתן לקריאה והופכת אותו לאותות הביניריים שמפוענחים על ידי הmicroarchitecture.
נשים לב שניתן לכתוב שפות מכונה שונות שיתלבשו על אותה microarchitecture.
הmicroarchitecture שנתאר היא זו של מחשבי הMAYBE, מכונה שתוכננה להדגים את הגישה שבה מתוכננים ארכיטקטורות דומות, אך אם זאת מעוצבת בפשטות כדי שיהיה נוח ללמוד אותה.

נביא סקירה קצרה לתהליך אותו אנו הולכים לעשות. בהמשך נסביר יותר על כל אחד מהדברים המוסברים כאן.
נניח שיש לנו מחשב, וכן גם תכנית של המשתמש הכתובה בשפת מכונה.
התוכנית של המשתמש יושבת בזיכרון הראשי של המחשב (במחשב הMAYBE זהו הDRAM). המחשב צריך להיות מסוגל להפעיל את התכנית הזו, כלומר המעבד צריך להיות מסוגל לקרוא את הפקודות מהDRAM ולבצע אותן.
האינטרפרטר של המחשב עובר על פקודות שפת המכונה וניגש לבצע אותן אחת אחת. הפקודות בשפת מכונה אינן מתייחסות לחומרה באופן ישיר. המיקרו אינטרפרטר לוקח כל פקודה בשפת מכונה, והופך אותה ל"שפה" אותה החומרה מבינה. כל פקודה בשפת מכונה מתורגמת *למיקרו פקודות*.
ברגע ששפת המכונה מוגדרת, והחומרה מוגדרת אז נוכל לכתוב מיקרו קוד עבור כל אחת מהפקודות בשפת המכונה, ואז המיקרו אינטרפרטר יהיה מוגדר. נוכל לקחת כל תוכנית בשפת המכונה ולהריץ אותה.
יש לכתוב למעשה את מיקרו התכנית פעם אחת. אם החומרה ושפת המכונה לא משתנות, אז המיקרו אינטרפרטר קבוע, כיוון שהמיקרו אינטרפרטר צריך לעשות תמיד בדיוק את אותו הדבר – להבין אוסף סופי (מבחינת מספר הפקודות השונות הקיימות) של פקודות. התכנית של המיקרו אינטרפרטר תהיה שמורה בזיכרון ROM. היא איננה אמורה להשתנות. תוכנית זו נשמרת בMicrocode ROM.
כל מיקרו פקודה מפוענחת לסדרה של הוראות שישלחו אל החומרה. כל מיקרו פקודה מפוענחת ל-"ננו קוד", שהן סדרת פעולות קצרה שנשלחת אל החומרה. הננו קוד נשמר על הControl ROM.



מסלול הנתונים מורכב מעורק אחד משותף שרוחבו N סיביות.
עורק זה הוא מסוג three-state, כך שכל תת מערכת מסוגלת לנתק את הרכיב המחובר אליה, ובכך להרשות לרכיבים אחרים לספק את הנתונים על העורק. נשים לב שלרוב הרכיבים במערכת יש קווי בקרה $\overline{OE}$. כאשר קו בקרה זה איננו פעיל, הרכיב מנותק בפועל מהעורק, ונותן לרכיבים האחרים להעלות מידע לעורק.
חשוב לדאוג שבו זמנית רק רכיב אחד ישלח נתונים על העורק, על מנת למנוע מרוץ.
הציור מראה מספר מערכות נפוצות לניתוח מידע המחובר לעורק. לכל יחידה מספר נוסף של קווי בקרה, הכוללים בדרך כלל את הOutput Enabled וכן Load Enabled, המשמש לטעינת רגיסטרים. אות שעון משותף זמין לכל הרכיבים במערכת, ומתזמן כל רגיסטר במערכת עם העורק. הכניסות של כל רגיסטר מחוברות אל העורק. כל רגיסטר שהLoad Enabled שלו פעיל במעבר הפעיל של השעון יטען בנתונים המופיעים על העורק בזמן אותו מעבר שעון.

מעגל הALU



אחד מהחלקים החשובים בשרטוט היא הArithmetic and logic unit (ALU). יחידה זו מסוגלת לבצע חישובים אריתמטיים ובוליאנים שונים על קלט של שני אופרנדים. הפונקציה שהALU מצבע נקבעת על ידי 6 כניסות בקרה המכונות " ALU function select". כמו כן, הALU מוציא קווי מצב המדווחים על תוצאת הפעולה (האם התוצאה יצאה שלילית, אפס וכו'). יציאות אלו משתמשות בדרך כלל למשוב עבור הבקר.

הטבלה הבאה נותנת את הפונקציה המבוצעת על אופרנדים A וB עבור כל צירוף של כניסות הבקרה.

Control inputs				Function performed if $M = 0$ (arithmetic)	Function performed if $M = 1$ (logical)
F_3	F_2	F_1	F_0		
0	0	0	0	$A + 1 - \overline{C}$	$\overline{A}$
0	0	0	1	$(A \text{ OR } B) + 1 - \overline{C}$	$\overline{(A \text{ OR } B)}$
0	0	1	0	$(A \text{ OR } \overline{B}) + 1 - \overline{C}$	$\overline{A} \text{ AND } B$
0	0	1	1	$-\overline{C}$	00000000
0	1	0	0	$A + (A \text{ AND } \overline{B}) + 1 - \overline{C}$	$\overline{(A \text{ AND } B)}$
0	1	0	1	$(A \text{ OR } B) + (A \text{ AND } \overline{B}) + 1 - \overline{C}$	$\overline{B}$
0	1	1	0	$A - B - \overline{C}$	$A \text{ XOR } B$
0	1	1	1	$(A \text{ AND } \overline{B}) - \overline{C}$	$A \text{ AND } \overline{B}$
1	0	0	0	$A + (A \text{ AND } B) + 1 - \overline{C}$	$\overline{A} \text{ OR } B$
1	0	0	1	$A + B + 1 - \overline{C}$	$\overline{(A \text{ XOR } B)}$
1	0	1	0	$(A \text{ OR } B) + (A \text{ AND } B) + 1 - \overline{C}$	B
1	0	1	1	$(A \text{ AND } B) - \overline{C}$	$A \text{ AND } B$
1	1	0	0	$A + A + 1 - \overline{C}$	11111111
1	1	0	1	$(A \text{ OR } B) + A + 1 - \overline{C}$	$A \text{ OR } \overline{B}$
1	1	1	0	$(A \text{ OR } \overline{B}) + A + 1 - \overline{C}$	$A \text{ OR } B$
1	1	1	1	$A - \overline{C}$	A

טבלה זו מתארת את הרכיב 74181 ALU.
ששת קווי הבקרה הקובעים את הפעולה מסומנים $F_3F_2F_1F_0\overline{C}M$. השמות היחודיים לשני קווי הבקרה האחרונים נובעים מהמשמעות המיוחדת שלהם בפיענוח הפעולה.
הביט $\overline{C}$ הוא ההופכי של הcarry הנכנס לביט הנמוך ביותר בפעולות כגון חיבור.
הביט M קובע את המצב (mode) הקובע האם אנחנו מבצעים פעולה אריתמטית (בה לcarry יש משמעות) או פעולה לוגית (בה אנו מתעלמים מהcarry).
בגלל ש $\overline{C}$ הוא ההופכי של הcarry הנכנס, שינוי $\overline{C}$ מ1 בביטויים אריתמטיים בדרך כלל הוא בעל האפקט של הוספת 1 לתוצאה.

הטבלה לעיל איננה כל כך שימושית כשאנו באים לתכנת בMAYBE. היא מכילה 64 פונקציות שרובן לא כל כך שימושיות.
הטבלה הבאה מכילה צירופים שימושיים שונים של קווי הבקרה:

Control inputs						
F_3	F_2	F_1	F_0	$\overline{C}$	M	Output
0	0	1	1	1	1	00000000
1	1	0	0	1	1	11111111
1	1	1	1	1	1	A
1	0	1	0	1	1	B
1	1	1	1	1	0	$A - 1$
0	0	0	0	0	0	$A + 1$
1	0	0	1	1	0	$A + B$
0	1	1	0	0	0	$A - B$
1	0	1	1	1	1	A AND B
1	1	1	0	1	1	A OR B
0	1	1	0	1	1	A XOR B
0	0	0	0	1	1	$\overline{A}$ (1's complement)

במחשב הMAYBE, שני האופרנדים של הALU מנותבים אליו על ידי רגיסטרים (operand registers) של N ביטים כל אחד, שכל אחד מהם מיועד לאחד מהאופרנדים A וB.

נניח שנרצה לחבר שני מספרים בעזרת הALU. נטען את המספרים לרגיסטרים A וB בשני מחזורי פעולה. במחזור הראשון נפעיל את הLoad Enabled של A, ונדגום ערך לתוך הרגיסטר A. במחזור שני נפעיל את הLoad Enabled של B, ונדגום את המספר השני לתוך B. כעת שני הערכים נמצאים ברגיסטרים. במחזור הפעולה האחרון, פונקצית הALU המבוקשת חייבת להיבחר בעזרת קווי הבקרה של הALU. האות $\overline{DRALU}$ צריך להיות פעיל כדי לגרום לתוצאה להופיע על העורק, והLoad Enable של אחד מהרגיסטרים במערכת צריך להיות פעיל כדי לשמור את התוצאה על הרגיסטר. הערה חשובה היא שהרגיסטר יכול להיות אף A או B, כך שבמחזור הבא התוצאה כבר תשב על הALU.

נשים לב כי רק לאחד הרכיבים מותר לכתוב בו זמנית על הbus, אולם אין הגבלה כמה רכיבים יקראו בו זמנית את המידע הנמצא על העורק.

אם זאת, ב-MAYBE רק רכיב אחד יכול לקרוא מה-BUS בו זמנית.

רוחב מסלול הנתונים וגודל הנתונים

למרות שבמבנה הכללי שבשרטוט, אנו מתייחסים למסלולי נתונים באורכים שונים, בדוגמא המוצגת כאן נתמקד במסלולי נתונים בעלי 8 ביטים. מכאן - העורק, הALU וכניסות ויציאות הרגיסטרים הם כולם בני שמונה ביטים כל אחד.
בחירה זו של שמונה ביטים היא בעיקר לצורך שיפור ביצועים. מחשבים בעלי מסלולי נתונים ברוחב 32 ביטים משתמשים לרוב במסלולים אלו לפעולות על שמונה ביטים (כגון קודי ASCII של תווים). פקודות המבצעות פעולות כאלו לרוב מתעלמות מהמידע הלא מנוצל או קובעות את הביטים הלא מנוצלים ל0. יותר מזאת, מחשבים העובדים עם מסלול נתונים ברוחב 8 ביטים, יכולים לעבוד עם נתונים גדולים יותר. ניתן למשל לקחת רצף של פעולות ב8 סיביות כדי לדמות, למשל, פקודה בת 32 סיביות.
טיפול בגדלים שונים של פקודות הוא תהליך איטי, אלא אם כן הוא נתמך בחומר, ולכן רוב המחשבים המודרניים מעדיפים לדבוק בפקודות בגודל קבוע. בדוגמא בה נדון אנו מתעסקים עם מכונה המטפלת בפקודות 8 ביטים בלבד, אולם נוכל לדמות בעזרתה מכונות אחרות בעלות רוחב פקודה שונה. עבור מתכנת שיעבוד במכונות אלו, רוחב הפקודה האמיתי יהיה שקוף - כלומר, המתכנת במכונות שימומשו על הMAYBE לא יהיה מודע לרוחב הפיסיקלי של מסלולי המידע.

מיעון בתים

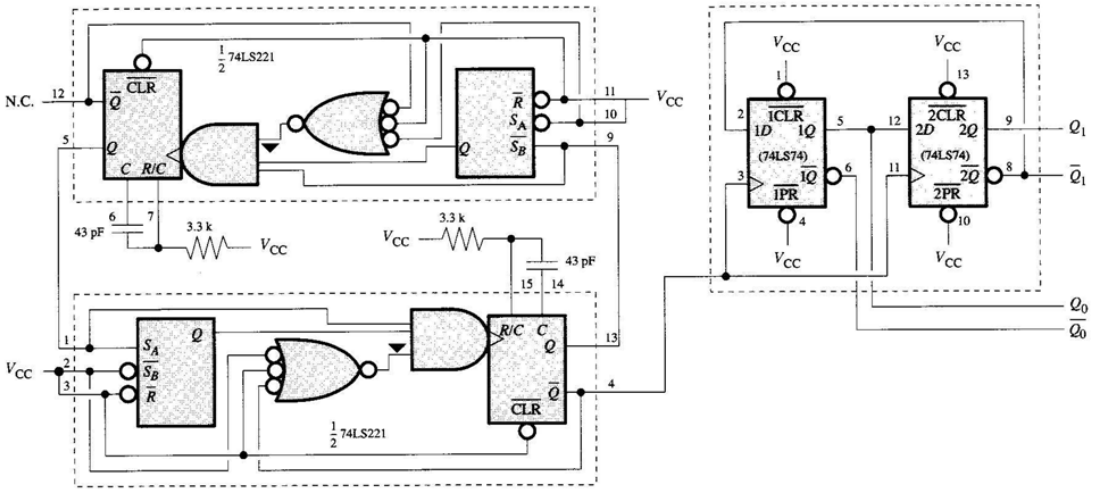
אחד האתגרים בתכנון ארכיטקטורה של מחשב הוא לתכנן את הזיכרון כך שיוכל להחזיק נתונים שונים (אותיות, מספרים, פקודות וכו'), וכן לבצע עליהם פעולות ביעילות.
מחשבים רבים פותרים את הבעיה על ידי התייחסות ליחידות בנות 8 ביטים המכונות בתים. כאשר הנתונים הם גדולים מכדי להתאים לבית בודד, הם מאוכסנים במספר בתים.
כאשר אנו מאכסנים משתנה בן יותר מ8 ביטים במספר בתים בני שמונה ביטים, מתעוררת בעיה חדשה. נביט בבעיה בעזרת דוגמא.
נניח שיש לנו משתנה המייצג מספר שלם חיובי, בן 16 ביטים, למשל, נניח שאנו רוצים לשמור את המספר 13283 בזיכרון.
נביט במספר זה בייצוג בינרי, כאשר הוא מחולק לשני בתים:

00110011	11100011
MSB	LSB

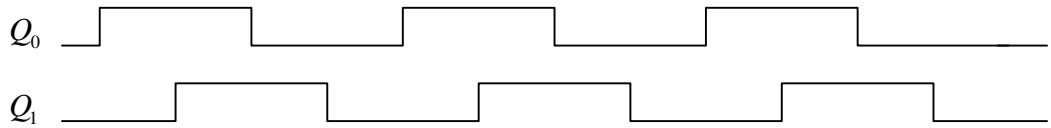
השאלה שמתעוררת היא: אילו מהבתים ישב ראשון בזיכרון, הMSB או הLSB?
ראשון בזיכרון משמעותו בבית בעל הכתובת הקטנה יותר.
נאמר שארכיטקטורות בהן הMSB נמצא בבית הנמוך תוכננו בשיטת "big-Endian" ונאמר כי ארכיטקטורות בהן הLSB נמצא בבית הנמוך תוכננו בשיטה "little-Endian".
אין עדיפות לאף אחד מהבחירות וישנן ארכיטקטורות הפועלות בכל אחת מהגישות. בMAYBE בחרו לעבוד בשיטת little-Endian.

תת מערכות שונות בתוך הMAYBE

שעון המערכת

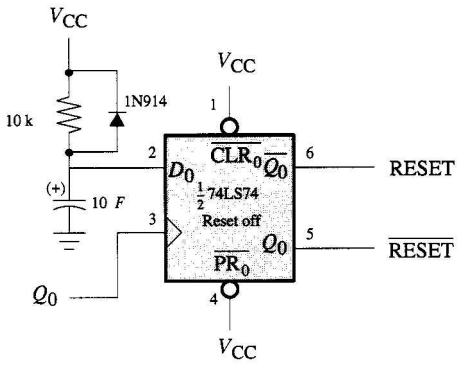


שעון המערכת פועל בתדירות 3.7 MHz.
זהו מונה Gray code המייצר את Q_0 ו Q_1 שהם שני השעונים במערכת. Q_1 מפגר אחרי Q_0 ברגע מחזור.



רוב הפעולות במערכת קשורות ומתוזמנות על ידי Q_0. אנו משתמשים ב Q_1 רק במקרים מיוחדים, בהם צריכים יותר משעון אחד. דוגמא לכך נראה בתהליך הכתיבה לSRAM של הMAYBE.

מעגל הRESET



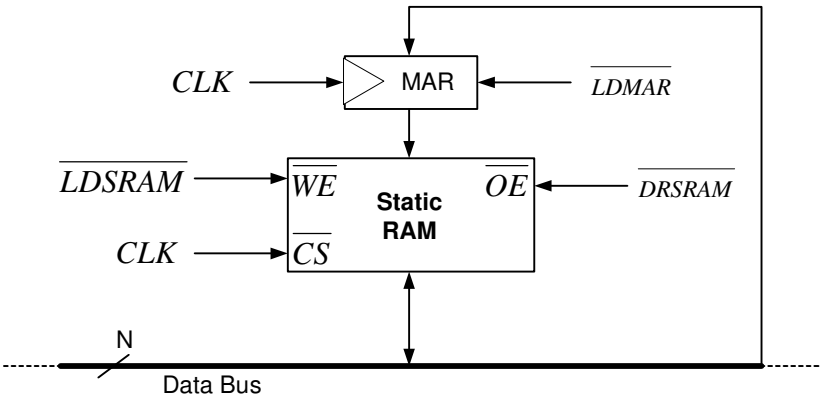
מעגל זה מאתחל את המחשב מיד עם הפעלתו.

נניח כי עד זמן t=0 אין מתח במעגל = המתח בנקודה 2 שווה ל0. עם הדלקת המתח תטען נקודה 2 לפי $V(t)=1-e^{\frac{-t}{RC}}Vcc$ והFF יזהה בכניסה D ערך '1' כאשר $V(t)=Vih$ (כלומר המתח בנקודה 2 הוא המינימלי המזוהה כ'1' בכניסה). לכן מובטח לנו אות RESET מחיבור למעגל ועד לזמן זה.

הRESET דואג לאיפוס הPhase, הCC, והADR ושם 0000 בMicrocode ROM בתור הopcode.

Static RAM

בנוסף לרגיסטרים A וB הנמצאים על הALU ורגיסטרים נוספים הנמצאים על מסלול הנתונים, נרצה זיכרון נוסף בו תוכל הmicroprogram לשמור תוצאות חישובים ומידע מסוגים שונים.
נבחר ברכיב SRAM למטרה זו.



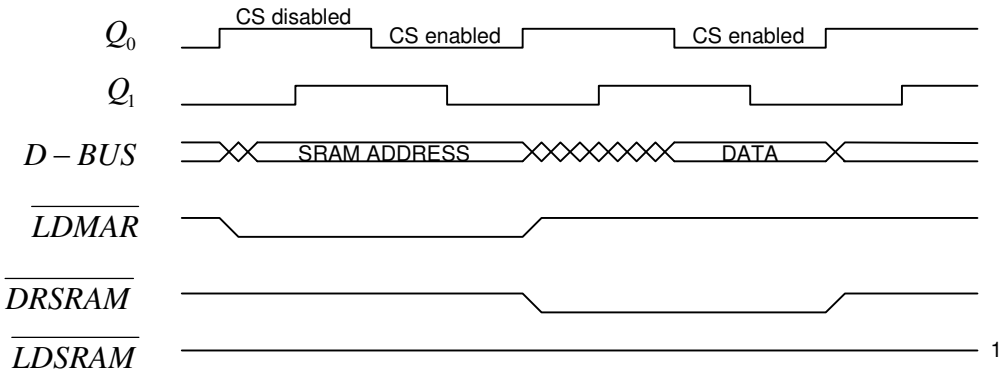
הSRAM בMAYBE

אנחנו מניחים כי רכיב הSRAM מכיל N ביטים עבור כתובות, כאשר כל כתובת מכילה N ביטים. לכן, במקרה של הMAYBE, קיימים 256 כתובות בנות 8 ביטים כל אחת. הרגיסטר MAR מכיל את הכתובת אליה ניגש בSRAM.
אופן פעולת הSRAM:
קריאה מהזיכרון: טוענים את הכתובת המבוקשת לרגיסטר MAR. במחזור השעון הבא מורידים את ה $\overline{OE}$ של הSRAM, כדי להעביר לbus את תוכן הכתובת המבוקשת.
כתיבה לזיכרון: טוענים את הכתובת המבוקשת לרגיסטר MAR. מורידים את $\overline{WE}$ כדי שהנתונים יעברו מהbus אל הכתובת המבוקשת.

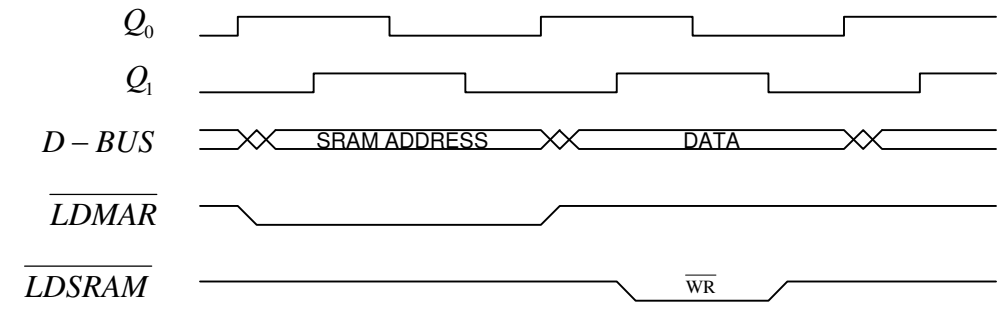
אנו הולכים להשתמש בSRAM בצורה אינטנסיבית. בתוכניות שנכתוב, הSRAM יכיל את המחסנית של התוכנית, בנוסף לנתונים של ה-microprogram.

- הכתובת לSRAM משתנה בעליית Q_0 (אם $\overline{LDMAR}$ היה פעיל במחזור הקודם).
- ה $\overline{CS}$ של הSRAM מחובר לQ_0 ולכן מתקבל $\overline{CS}$ רק בירידת Q_0 ובמחזור הקריאה אנחנו מקבלים את הdata רק לאחר ירידת Q_0.
- נשים לב לתזמון המיוחד של $\overline{LDSRAM}$ במחזור הכתיבה כך שהוא עולה לפני שינוי הDATA ומתקיים תנאי הHOLD.

Read Cycle



Write Cycle



Read Cycle

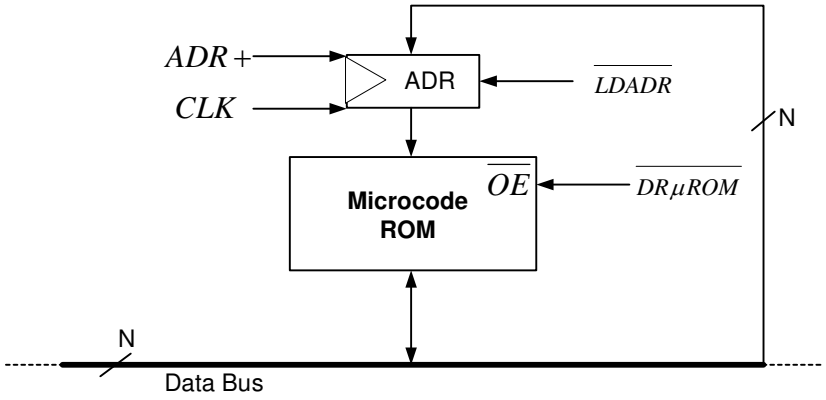
עם עליית השעון Q_0, $\overline{LDMAR}$ מתאפשר. $\overline{LDMAR}$ פעיל ומאפשר את טעינת ה-MAR. הכתובת אל הMAR ניתנת דרך ה-D-BUS מרגיסטר כלשהו. בעליית השעון הבאה הכתובת השמורה ב- MARמשתנה. בירידת השעון $\overline{CS}$ מאופשר, המידע יורד מה-MAR אל ה-SRAM, ומשם ל-BUS, מכיוון ש $\overline{DRSRAM}$ מאופשר.
$\overline{DRSRAM}$ מאופשר כבר מעליית Q_0 ולא יותר מאוחר, אף על פי כן שאין צורך בכך כי הוא "נע" ביחד עם Q_0.

Write Cycle

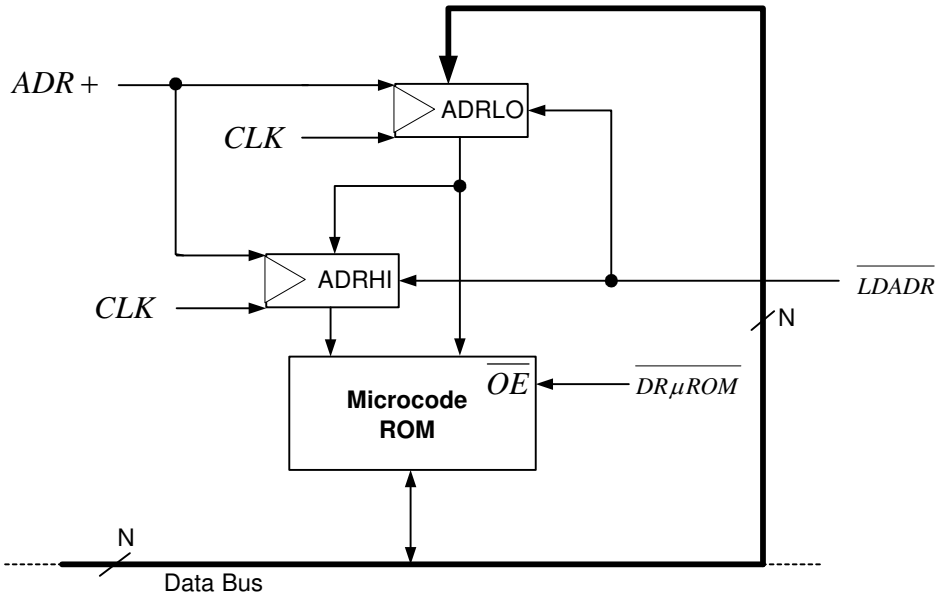
המידע מגיע מה-MAR ומה-BUS. ה-MAR בוחר את הכתובת אליה המידע צריך להיכתב. לכן, השלב הראשו של הכתיבה דומה לשלב הראשון של הקריאה. בעליית Q_0 הבאה, נבחר רגיסטר אחר ל-Load, ולכן $\overline{LDMAR}$ יורד. ה-DATA יגיעו ל-BUS מהרגיסטר שייבחר. לאחר פרק זמן, יהיו ב-BUS נתונים תקפים.
$\overline{LDSRAM}$ מחובר ל-Q_1, ולכן יש הפרש זמנים מסויים מרגע שהנתונים תקפים עד שנכתוב אותם. מיד בעליית Q_1 ה-$\overline{CS}$ מאופשר ומתרחשת כתיבה.
מדוע במחשב MAYBE, $\overline{LDSRAM}$ מחובר עם Q_1, ולא עם Q_0?
התשובה: כדי לספק את תנאי ה-hold של ה-SRAM במחזור הכתיבה.
תוכן ה-SRAM ב-MAYBE: (בהמשך נראה את המשמעות של השדות השונים המוצגים בטבלה זו).

Addr	SRAM CONTENTS
0xFF uSP	<uSP>
0xFE	<Refresh Counter>
0xFD	Temporary Registers
	. . .
Stack | v	Bottom of Stack - Element #1 (Highest u-stack Address)
	Stack element #2
	Stack element #3
	. . .
<uSP>	Top of Stack
	Stack can further grow in the direction | | | v
0x10	Stack Should not grow below this entry
0x0F R15	
0x0E R14	
0x13 . . 0x02	. . .
0x01 R1	
0x00 R0	

Microcode ROM זהו זיכרון לקריאה בלבד המכיל את הmicroinstructions שמפוענחות על ידי המיקרו ארכיטקטורה.
באופן בסיסי, הROM יכל להיות ממומש בדומה לממשק של הSRAM: המערכת יכלה להכיל רגיסטר אחד לכתובת, שניתן לטעון אותו מהbus, בנוסף לROM שאליו מחובר three-state. אולם, שני סיבוכים משנים מעט את ארכיטקטורה זו.
ראשית, בריצה של microprogram טיפוסית, רוב הגישות לROM יהיו ברצף - בית אחרי בית בזיכרון. אנו יכולים להשתמש בALU כדי להוציא רצף כתובות עוקבות, ובעזרתן כל פעם לגשת למילת הפקודה הבאה, אולם פתרון זה איטי ביותר. נפתור בעיה זו על ידי הוספת מונה בינרי, אשר יחליף את רגיסטר הכתובת בחלק מהמקרים. מונה זה יבצע הגדלה של הערך הנמצא ברגיסטר, במקרה ש $ADR+$ פעיל.



הסיבוך השני הוא שנרצה יותר מ256 כתובת עבור הmicrocode שלנו. הפתרון יהיה הרחבת ADR למשל ל16 ביטים.
כעת, כדי לגשת לכתובת בMicrocode ROM נצטרך להעביר 2 בתים עבור הכתובת.
נוסיף לMicrocode ROM שני רגיסטרים, ADRHI,ADRLO , שישמרו את הסיביות הגבוהות של הכתובת המבוקשת ואת הסיביות הנמוכות של הכתובת המבוקשת.
תהליך קריאת כתובת מהROM:
במחזור הראשון נטען את הסיביות הגבוהות אל הROM. הבית ישמר בADRLO.
במחזור השני נטען את הסיביות הנמוכות אל הROM. הסיביות הגבוהות יעברו על הרגיסטר ADRHI, ואילו בADRLO יהיו הסיביות הנמוכות.



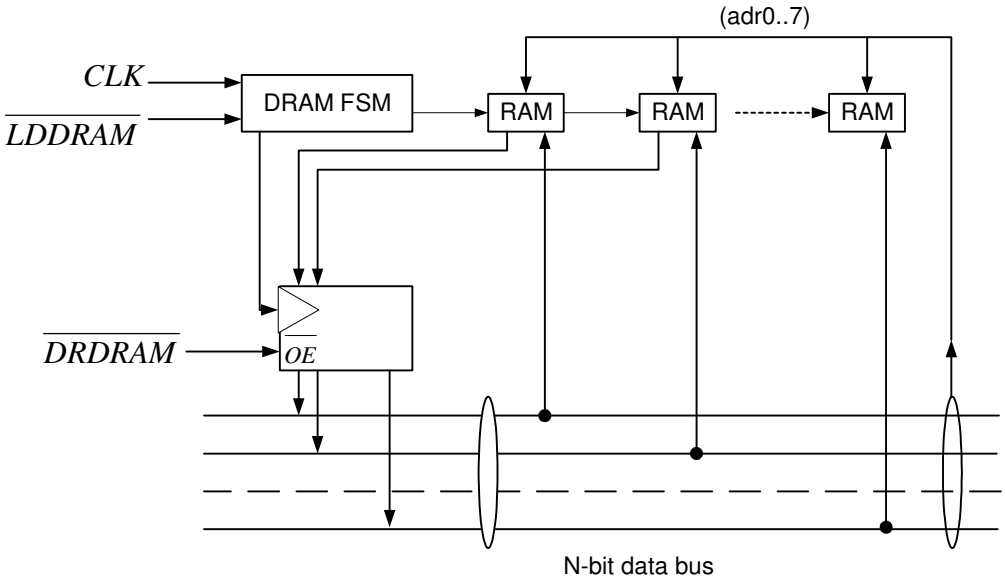
Microcode ROM with 16-bit address register

נעשה שימוש בDRAM כזיכרון ראשי, שכן זהו רכיב זול ובעל קיבול גבוה.
לDRAM יש מעט רגליים והוא תופס מעט שטח על הלוח, ולכן נעדיף אותו על הSRAM.

ישנן מספר בעיות המתעוררות כאשר אנו רוצים להשתמש בDRAM כזיכרון ראשי.

1. בDRAM יש פחות רגליים מאשר כתובות גישה. כל chip הינו מערכת זיכרון בת $(2^n$ by 1)bit, ולכן יש לו כניסה אחת ויציאת 3state אחת.
   ⇐ לצורך הכנסת כתובת נדרש לבצע רצף של inputs:
   ראשית כתיבת שורה $\overline{RAS}$ ואחר כך כתיבת עמודה $\overline{CAS}$.
   $\overline{WRITE}$ קובע האם תבוצע קריאה או כתיבה לDRAM.
2. בעיה נוספת הינה הצורך ברענון המידע.
   רענון שורה מתבצע על ידי הורדת $\overline{RAS}$ ($\overline{LDDRAM}$ שוהה למטה לפאזה אחת).
   בDRAMs רבים הרענון מתבצע על ידי בקר פנימי שבתוך הDRAM, אולם במחשב הMAYBE העדכון נעשה על ידי הMicrocode.

מבנה הDRAM בMAYBE:



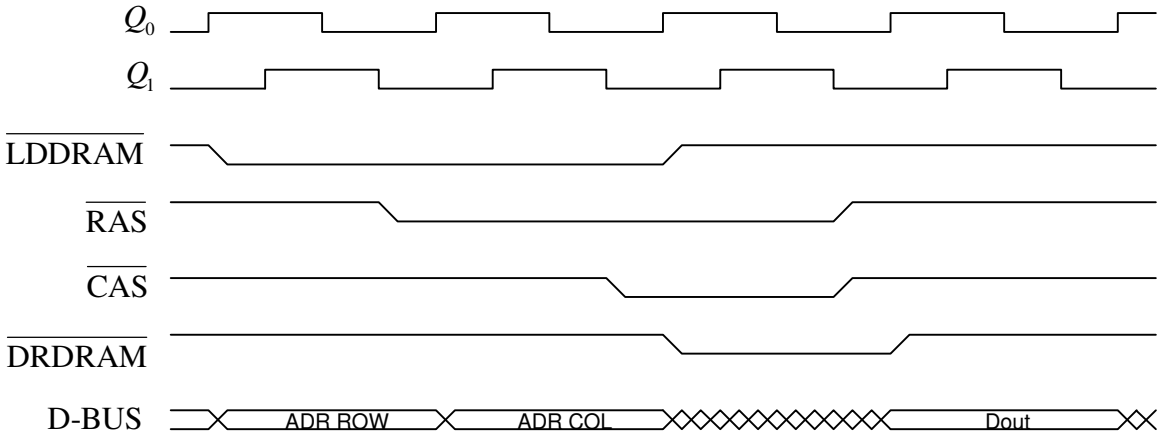
- ה-DRAM מכיל 8 יחידות RAM מסוג (64K by 1)bit וכך נוצר זיכרון בן 64Kbyte.
- לכל יחידת RAM יש 8 כניסות כתובת שמוטענות ב8bit (Row) ואחר כך 8bit (Col) סה"כ מ8 היחידות מקבלים מילה בת 8bit. 8 היחידות נשלטות על ידי FSM. הFSM מתוכננת לביצוע הפעולות הבאות:

Read - כדי לקרוא ערך מDRAM, אנו טוענים את ערכי ADRLO, ADRHI ב2 מחזורים עוקבים, ובמחזור לאחר מכן איננו כותבים אל הDRAM, ועל ידי כך גורמים לbit8 של הכתובת הרצויה להיטען לתוך האוגר של הDRAM. הערך ניתן לשימוש במחזורים הבאים על ידי בחירת האוגר כמקור המידע כלומר קביעת $\overline{DRDRAM}$.

Write - טעינת ערכים לADRLO, ADRHI ולData ב3 מחזורים עוקבים גורמת לפעולת כתיבה. הData נטען לכתובת הרצויה.

Refresh - כדי לרענן שורה טוענים ערך לADRHI במחזור אחד ובמחזור הבא אחריו לא קובעים $\overline{LDDRAM}$. פעולה זו גורמת לרענון השורה שכתובתה מצויה בADRHI.

Read Cycle



הערה: אם נרצה לכתוב את מוצא הDRAM לSRAM, יש למשוך את $\overline{DRDRAM}$ מחזור נוסף.

תרגיל

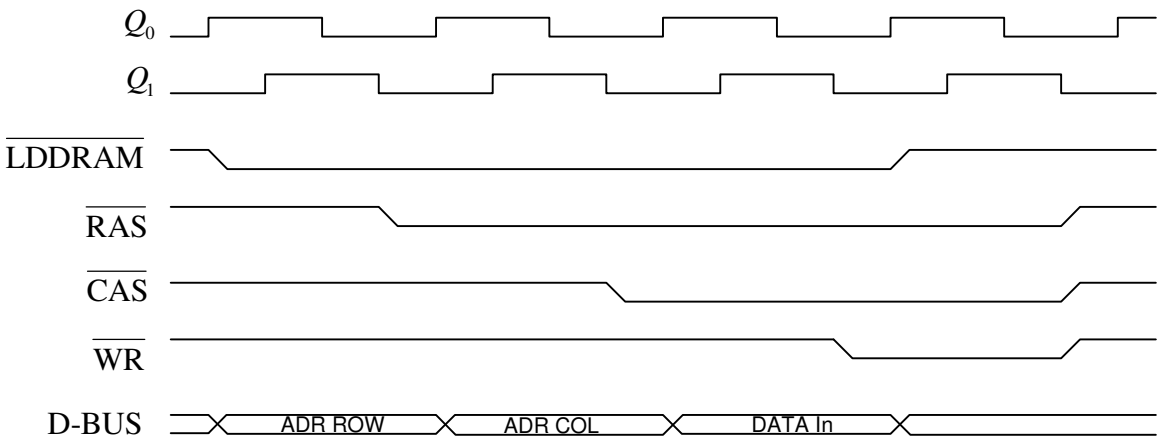
בעת בניית גירסה פירטית של ה-Maybe חיברו בטעות את הקווים, כך שקריאת הערך בזיכרון שהתבצעה עד כה בעליית קו CAS נעשית כעת בעליית קו RAS. כיצד ישפיע הדבר על פעולת המחשב מבחינת ביצוע קריאה מהזכרונות?

א. הקריאות מה-SRAM ומה-DRAM לא תושפענה, והן תעבודנה נכון.
ב. רק הקריאה מה-SRAM תושפע, והיא תהיה שגויה.
ג. רק הקריאה מה-DRAM תושפע, והיא תהיה שגויה.
ד. הקריאות מה-SRAM ומה-DRAM תושפענה שתיהן, והן תהיינה שגויות.

פתרון

ראשית נזכור כל קווי ה-CAS וה-RAS רלוונטים רק ל-DRAM, ולכן לא ייתכן כי ה-SRAM יושפע. האם ה-DRAM יושפע? כמו שניתן לראות, ה-CAS וה-RAS עולים בו זמנית, ולכן לעובדה שקריאת הערך בזיכרון שהתבצעה עד כה בעליית קו CAS נעשית כעת בעליית קו RAS לא תהיה משמעות. התשובה הנכונה: א'.

Write Cycle



בקר ה DRAMממומש בעזרת רגיסטר הזזה 74LS194.
נשתמש באות $\overline{LDDRAM}$ ליצירת מחזורי קריאה, כתיבה וריענון.

1. Refresh - $\overline{LDDRAM}$ פעיל במשך מחזור שעון אחד.
2. Read - $\overline{LDDRAM}$ פעיל במשך שני מחזורי שעון.
3. Write - $\overline{LDDRAM}$ פעיל במשך שלושה מחזורי שעון.

נשים לב שרכיב ה74LS194 מחובר ל $\overline{Q_1}$ ולכן יציאותיו משתנות <u>בירידת</u> Q_1.

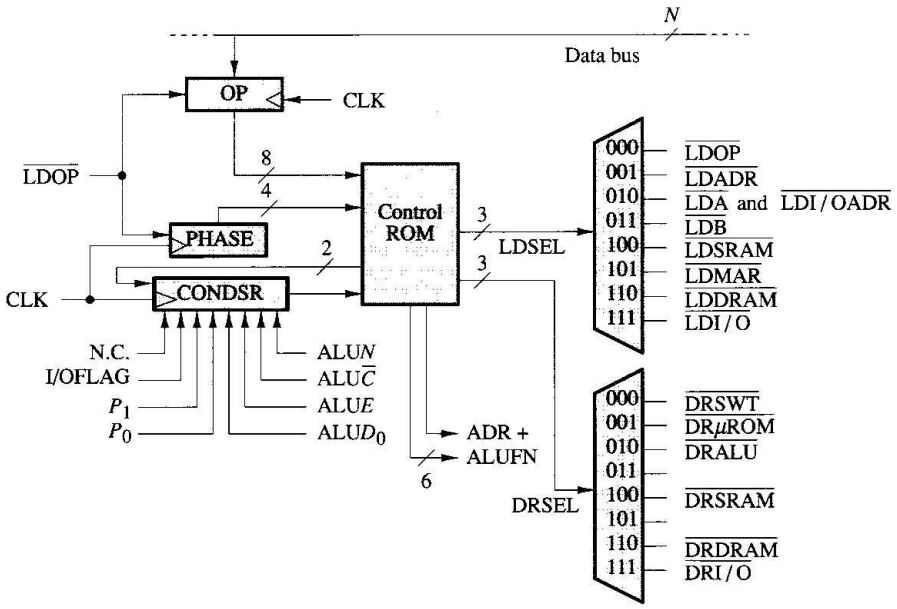
<u>מתגים ואורות (SWITCHS)</u>

נרצה לאפשר לMAYBE לבצע פעולות Input/Output בסיסיות.
לפיכך הוספו לMAYBE מספר רכיבים:

1. שמונה מתגים, מסומנים $S_7...S_0$ המשמשים כיחידת קלט.
   זהו למעשה התקן three-state המכיל 8 ביטים של מידע.
2. באופן דומה, אנו מחברים 8 מנורות לרגיסטר B באופן כזה שכאשר המיקרו תוכנית רוצה להציג ערך על הנורות, היא מציבה את הערך ברגיסטר B, ואחרי תקופה קצרה הנורות מציגות את אותו הערך.

מערכות הבקרה (Control Systems)

לב מערכת הבקרה הינו הControl ROM אשר מקבל מספר כניסות (המייצגות את מצב הרכיבים בData-Path וכן מצבים שקשורים למערכת הבקרה) ומייצר אותות יציאה Drive וLoad.
קווי הבקרה של הROM ($\overline{CS}, \overline{OE}$) קבועים כך שהרכיב מאופשר תמיד.



הערה: ישנם 9 קווי Load אך רק 8 יציאות $\overline{LDA}$ and $\overline{LDI/OADR}$ בעלי יציאה משותפת. דבר זה גורר את התופעה הבאה: הMAR של הUART מתבצעת כאשר טוענים את רגיסטר A, ולהפך.

בתוך הControl ROM ישנם מספר רגיסטרים, עליהם נפרט כעת. כל אחד מהם מחובר לשעון המערכת, ויכול להשתנות בכל מחזור שעון, אם מפעילים אותו. (enable).

COND

אוגר הזזה בן 8 ביטים.
הביט הגבוה של הרגיסטר מחובר לControl ROM ולכן ניתן לבדוק את הביטים המצויים ברגיסטר על ידי ביצוע load ואח"כ מספר מחזורי rotate. דרך זו חוסכת בגודל של הROM אך דורשת יותר מחזורי שעון.

OP

- אוגר בן 8 ביטים המקשר בין מערכת הבקרה לבין מערכת המידע (data, control).
- חיוני ביותר לפענוח הμcode .
- כניסות הOP מחוברות לData Bus.
- יציאות הOP מחוברות לControl ROM.

- ערכו של OP נטען פעם אחת עבור כל μcode שמבוצעת.
ערך OP נטען לפני תחילת ביצוע פקודה הμcode הבאה.
ערכו של הOP הוא קוד המתאר את הפעולה לביצוע.

Phase

אוגר מונה בן 4 ביטים, מקודם כל מחזור שעון. מוצא Phase מחובר רק ל Control ROM. לאוגר Phase יש אות בקרה שגורם לה לאיפוס (0000).
$\overline{LDOP}$ גורם לclr של phase ולטעינת OP.

סיכום

- לתוך הOpcode נכנס מידע מה μcode ROM (מידע זה מציין כתובת ב Control ROM).
- הControl ROM יפרש את הOpCode (+ערך Phase + ערך CC) וייתן אותות בקרה בהתאם.
- ב μROM קוראים פקודות לפי הסדר, אלא אם כן הבקר ייתן כתובת ספציפית לקרוא מתוכה.

סימונים

כדי לתאר את פעולות המכונה, ניצמד לשפה דמוית C.
נשתמש בסימונים הבאים:

$\alpha \leftarrow \beta$ - הנתון β עובר ליעד α.
דוגמא: נניח כי R1 הוא רגיסטר, אזי $R1 \leftarrow 237$ משמעותו שהייצוג הבינרי של 237 יכנס לתוך הרגיסטר.

$\langle ... \rangle$ - התוכן של.
למשל, $\langle R \rangle$ - התוכן של R
$\langle R1 \rangle \leftarrow \langle R2 \rangle$ - הכנס את התוכן של R2 אל R1.
$\langle 100 \rangle \leftarrow \langle 123 \rangle$ - הכנס את התוכן של כתובת 123 לכתובת 100.
$\langle B \rangle \leftarrow \langle A \rangle$ - הכנס את התוכן של A לכתובת שמייצג התוכן של B.

נשים לב שהביטויים בשני צדי החץ מפורשים אחרת. הביטוי בצד שמאל מציין כתובת ואילו הביטוי בצד ימין מציין ערך.

נסמן את הגודל של הנתונים שאנו מעבירים בצורה x^{size}, למשל:
כל הדוגמאות הבאות יעבירו בלוק בגודל 4 בתים מכתובת 123 לכתובת 100.
$\langle 123 \rangle^4 \leftarrow 100^4, \langle 123 \rangle \leftarrow 100^4, \langle 123 \rangle^4 \leftarrow 100$.

תכנות הControl ROM

כאשר אנחנו מתכנתים את הControl ROM, אנו למעשה קובעים מהם הנתונים שיהיו צרובים על הROM.
קביעת הנתונים היא למעשה סוג של תכנות, אם כי תחת הגבלות כבדות, למשל שעבור כל Opcode יכולים להיות לכל היותר 16 מחזורי שעון.
מכיוון שהקוד של הControl ROM הוא דרגה מתחת אפילו המיקרוקוד, נכנה את הקוד שלו בתור ננוקוד.
אנחנו קובעים את תוכן הControl ROM על ידי טבלת אמת, שמגדירה עבור כל צירוף של כניסות הControl ROM פלט של 16 ביטים, בפורמט הבא:

Address increment	ALU inputs						N.C.	COND S/L		Drive select			Load select		
$ADR+$	F_3	F_2	F_1	F_0	$\overline{C}$	M	X	I	S	D_2	D_1	D_0	L_2	L_1	L_0

Control ROM output word

נראה למשל דוגמא לקטע מתוכן הControl ROM, שמכיל את ארבעת התאים שמתעסקים עם השלבים (phases) 0 ו1 של הopcode 1.

Opcode	Phase	COND	=	ADR+	ALU	CC	DRSEL	LDSEL	Comments
00000001	0000	0	=	0	1100 01	11	010	101	$MAR \leftarrow$ 11111111
00000001	0000	1	=	0	1100 01	11	010	101	$MAR \leftarrow$ 11111111
00000001	0001	0	=	0	1111 11	11	100	010	A $\leftarrow$ SRAM
00000001	0001	1	=	0	1111 11	11	100	010	A $\leftarrow$ SRAM

כל שורה מכילה את הinputs הנכנסים אל הcontrol ROM, ולידם את היציאות עבור כניסות אלו.

נשים לב ששתי השורות הראשונות ושתי השורות האחרונות נבדלות ביניהן רק בביט אחד בכניסות, ואילו היציאות שלהן זהות. נוכל לכתוב זאת בצורה קצרה יותר, על ידי "wild-card". שורה עם * תסמן שעבור כל כניסה עבור אותו ביט, תצא אותה תוצאה.
למשל, נוכל לכתוב את הטבלה הקודמת בצורה מקוצרת:

Opcode	Phase	COND	=	ADR+	ALU	CC	DRSEL	LDSEL	Comments
00000001	0000	*	=	0	1100 01	11	010	101	$MAR \leftarrow$ 11111111
00000001	0001	*	=	0	1111 11	11	100	010	A $\leftarrow$ SRAM

נרצה להגדיר פלט ברירת מחדל עבור הControl ROM, שיהיה למעשה ערך שימלא את כל המקומות שלא הוגדר להם תוכן מפורש.
נרצה שקוד זה יגיד למפענח להפסיק לפענח את הפקודה הנוכחית ולעבור לפקודה הבאה. הנה הצעה לדרך לעשות זאת:

Opcode	Phase	COND	=	ADR+	ALU	CC	DRSEL	LDSEL	Comments
********	****	*	==	1	1111 11	11	001	000	Opcode ← μROM; ADR+

פירוש ה== הוא שזהו ערך default, שמוחלף אם ידוע ערך אחר לאותה שורה.
שורה זו היא שורת "begin next microinstruction" טיפוסית. היא אומרת שהרגיסטר OP יטען מה-microinstruction ROM, וכן שהכתובת בmicrocode ROM תגדל ב-1.
(עובדה זו מצוינת על ידי ה1 בעמודה ADR+).

Condition Register

נביט כעת במילה הנטענת אל רגיסטר הבקרה.

C_7	External Communication C_6	Push buttons C_5 C_4	ALU output conditions bits C_3 C_2 C_1 C_0
N.C.	I/OFLAG	P_1 P_0	D_0 E $\overline{C}$ N

ניתן לגשת לכל אחת מהסיביות על ידי rotate ימינה. אחרי כמה rotates, הערך המבוקש יהיה בידנו. מכיוון ש C_0 מחובר אל ה-Control ROM, הוא יכול להשפיע על פענוח הפקודות הבאות, ללא צורך בrotating.

ארבעת הביטים - $D_0, E, \overline{C}, N$ משפיעים על היציאות של ה-ALU.
N הוא ביט הסימן. ביט זה הוא 1 כאשר הפלט של ה-ALU, (כשמתייחסים אליו בתור מספר שלם בן 8 ביטים בשיטת 2`s complement) הוא מספר שלילי.
D_0 הוא אחד כאשר הפלט של הALU הוא לא זוגי.
$\overline{C}$ הוא ההופכי של הcarry של הoutput של הALU.
E הוא אחד, רק כאשר מילת הפלט של הALU מכילה רק 1ים.
ביט זה נקרא ביט השוויון, מאחר ואם משתמשים בפונקצית הALU $A-B-1$ אזי הפלט שלה הוא 1ים אם ורק אם A שווה לB.
P_0, P_1 הוא שני לחצנים, שהם משמשים כמקור לקלט מהמשתמש.
I/OFLAG מגיע מממשק התקשורת של ה-MAYBE.
C_7 לא בשימוש, ואינו מחובר לשום רכיב.

מימוש אותות הCC

$\overline{C} = \overline{C_n} + 4$ של ה-ALU high
E = ממומש על ידי open collector AND במעגל הALU.
N = הסיבית העליונה של התוצאה (MSB)
D_0 = הסיבית התחתונה של התוצאה (LSB)

תחביר המיקרו קוד

שפת המכונה היא שפה המורכבת ממיקרו פקודות.
כל מיקרו פקודה מיוצגת על ידי רצף של בתים השמורים בMicrocode ROM.
אמMicrocode מתוכנת כראוי, הוא צריך לרוץ בלי קשר כיצד הוא מפוענח על ידי החומרה.

למרות שMicrocode, זוהי שפת תכנות נמוכה (low level), היא עושה מספר הנחות והסתרות מידע למתכנת בה, על מנת להפוך את מלאכת התכנות לקלה יותר.
אנו משתמשים בSRAM כאוסף של רגיסטרים וירטואליים הזמינים לשימוש על ידי הmicrocode, ואנחנו מניחים כי כל מיקרו פקודה לוקחת קלט מרגיסטרים אלו ומייצרת פלט לתוכם.
16 הכתובות הראשונות בSRAM משמשות כ16 רגיסטרים וירטואליים הממוספרים $R0, R1, ..., R15$.
כל מיקרו פקודה מקודד לתוך הMicrocode ROM כרצף של בתים.
הבית הראשון בכל רצף הוא הOpcode של הפקודה, שנטען לתוך הOP.
ניתוח הבתים הבאים משתנה מפקודה אחת לאחרת, אך בדרך כלל הבתים הבאים הם שמות של רגיסטרים (כתובות בSRAM) או קבועים של התוכנית.
למשל, נביט במיקרו פקודה add. הopcode של add הוא 0001010 שאחריו הבתים x,y,z המגדירים את שמות הרגיסטרים המשמשים כאופרנדי הקלט, ואחריהם רגיסטר המשמש כאופרנד פלט.
מיקרו פקודה זו מבצעת את הפעולה $z \leftarrow \langle x \rangle + \langle y \rangle$.

נביט בננו קוד המממש את הפקודה add.

Opcode	Phase	COND	=	ADR+	ALU	CC	DRSEL	LDSEL	Comments
00001010	0000	*	=	1	1111 11	11	001	101	*MAR* ← μROM; ADR+
00001010	0001	*	=	0	1111 11	11	100	010	A ← SRAM
00001010	0010	*	=	1	1111 11	11	001	101	*MAR* ← μROM; ADR+
00001010	0011	*	=	0	1111 11	11	100	011	B ← SRAM
00001010	0100	*	=	1	1111 11	11	001	101	*MAR* ← μROM; ADR+
00001010	0101	*	=	0	1001 10	00	010	100	SRAM ← A+B; latch CCs
00001010	0110	*	=	1	1111 11	11	001	000	Opcode ← μROM; ADR+

Nanocode for add (x, y, z) microinstruction

קוד זה מממש בשבעה מחזורי שעון את הפקודה add. נשים לב שהקוד מתחיל כאשר phase=0, כלומר מיד לאחר המחזור שהסתיים בהעלאת הOpcode החדש.
הקוד מסתיים בפקודה האומרת לאינטרפרטר להעלות את הOpcode הבא מהMicrocode ROM.

כדי לפענח ננוקוד נתון, נעזר בטבלאות הבאות:

Address increment	ALU inputs						N.C.	COND S/L		Drive select			Load select		
$ADR+$	F_3	F_2	F_1	F_0	$\overline{C}$	M	X	I	S	D_2	D_1	D_0	L_2	L_1	L_0

Control ROM output word

DRSEL			Source
D_2	D_1	D_0	
0	0	0	Switch Register
0	0	1	Microcode ROM
0	1	0	ALU output
0	1	1	Not connected
1	0	0	Static RAM
1	0	1	Not connected
1	1	0	Dynamic RAM output register
1	1	1	Input/output data

Drive Select Signals

LDSEL			Destination
L_2	L_1	L_0	
0	0	0	OP (Opcode) register
0	0	1	Microcode ROM ADR
0	1	0	ALU A input and I/O address
0	1	1	ALU B input
1	0	0	Static RAM data
1	0	1	Static RAM address (MAR)
1	1	0	Dynamic RAM
1	1	1	Input/output data

Load Select Signals

Inputs		Action
I	S	
0	0	Load condition register
0	1	Shift condition register right
1	0	No change
1	1	No change

Condition shift register control inputs

נוכל להוסיף ולהשתמש בקבועים כאשר נכתוב מיקרו קוד.
המיקרו פקודה cadd מחברת קבוע לכתובת בSRAM.
כאשר פקודת cadd צרובה בזיכרון, ראשית צרוב הOpcode של הפקודה, לאחריו צרוב הקבוע אותו אנו רוצים לחבר, ולאחריו צרובה כתובת הרגיסטר אליו נחבר את הקבוע. כאשר אנו כותבים $\text{cadd}(cx, y, z)$, משמעותו של cx היא שהביטוי שנצרוב במקום cx איננו רגיסטר בSRAM, אלא ערך מספרי קבוע.

Opcode	Phase	COND	=	ADR+	ALU	CC	DRSEL	LDSEL	Comments
00001011	0000	*	=	1	1111 11	11	001	010	$A \leftarrow \mu$ROM; ADR+
00001011	0001	*	=	1	1111 11	11	001	101	$MAR \leftarrow \mu$ROM; ADR+
00001011	0010	*	=	0	1111 11	11	100	011	B ← SRAM
00001011	0011	*	=	1	1111 11	11	001	101	$MAR \leftarrow \mu$ROM; ADR+
00001011	0100	*	=	0	1001 10	00	010	100	SRAM ← A+B; latch CCs
00001011	0101	*	=	1	1111 11	11	001	000	Opcode ← μROM; ADR+

Nanocode for cadd (cx, y, z) microinstruction

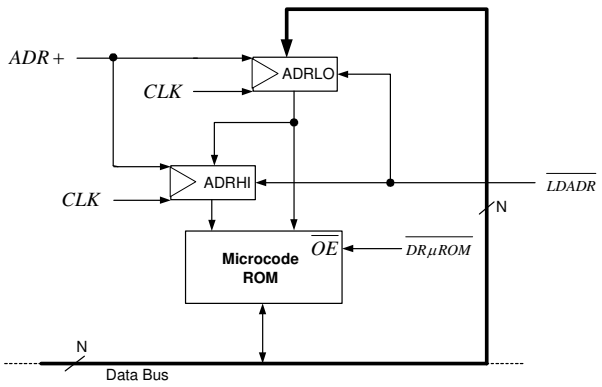
מיקרו פקודות שמבצעות פעולות אריתמטיות או לוגיות מקודדות כדי לטעון את הconditions code register בתוצאת הפעולה.
פעולות אחרות (למעט פקודות קפיצה) עוזבות את הconditions register ללא שינוי.
המיקרו פקודות מקודדות בצורה הבאה: ראשית מקודד הOpcode של הפקודה.
לאחריו מקודדות אופרנדי קלט, ובסוף מקודד אופרנד הפלט.
הטבלה הבאה מסכמת מספר פקודות שימושיות בMAYBE.

Symbolic microinstruction	Micro-ROM encoding	Operation performed
move(x, y)	0x05 x y	$y \leftarrow \langle x \rangle$
cmove(cx, y)	0x06 cx y	$y \leftarrow cx$
add(x, y, z)	0x0A x y z	$z \leftarrow \langle x \rangle + \langle y \rangle$
cadd(cx, y, z)	0x0B cx y z	$z \leftarrow cx + \langle y \rangle$
add2(x, y, z)	0x2C x y z x+1, y+1, z+1	$z^2 \leftarrow \langle x \rangle^2 + \langle y \rangle^2$
cadd2(cx, y, z)	0x2D cx%256 y z cx/256 y+1 z+1	$z^2 \leftarrow cx^2 + \langle y \rangle^2$
sub(x, y, z)	0x1C x y z	$z \leftarrow \langle x \rangle - \langle y \rangle$
cmp(x, y)	0x1D x y	$\langle x \rangle - \langle y \rangle$
and(x, y, z)	0x27 x y z	$z \leftarrow \langle x \rangle$ AND $\langle y \rangle$
or(x, y, z)	0x28 x y z	$z \leftarrow \langle x \rangle$ OR $\langle y \rangle$
xor(x, y, z)	0x29 x y z	$z \leftarrow \langle x \rangle$ XOR $\langle y \rangle$
not(x, y)	0x2A x y	$y \leftarrow$ NOT $\langle x \rangle$ (1's complemet)
neg(x,y)	0x2B x y	$y \leftarrow \left(-\langle x \rangle\right)$ (2's complemet)

מספר סימונים מקובלים, שניתן לראות מהטבלה:
סיפרה בודדת אחרי שם הפקודה מתארת את גודל המקור והתוצאה בבתים.
פעולות ללא מספר פועלות על בית בודד. למשל, add(x, y, z) מבצעת חיבור על בית אחד, ואילו add2(x, y, z) מבצעת חיבור של שני בתים.
אם מבוצעת פעולה בה אחד האופרנדים קבוע, נוסיף לפני שם הפעולה את התחילית c, מלשון constant, למשל, הפקודה cadd בה אופרנד המקור הראשון הוא קבוע.

בקרת זרימה והסתעפויות

שפת המיקרו כוללת מספר פקודות לבקרת הזרימה של התוכנית.
הפקודה הבסיסית ביותר היא קפיצה ללא תנאי.
פקודה זו טוענת ערך חדש לתוך הmicrocode ROM ADR register. ברוב המקרים, הכתובת החדשה היא קבוע שכלול בתוך המיקרו פקודה.
אנו מניחים כי בmicrocode ROM מצוי הopcode של jmp, ולאחריו קודם כל הבית הנמוך, ולאחריו הבית הגבוה של הכתובת החדשה.
על מנת להבין את ההגיון בדבר, נסתכל על השרטוט הבא, ובנוסף על הננו קוד.



נביט בננו קוד המממש את jmp:

Opcode	Phase	COND	=	ADR+	ALU	CC	DRSEL	LDSEL	Comments
0001100	0000	*	=	1	1111 11	11	001	010	A ← μROM; ADR+
0001100	0001	*	=	0	1111 11	11	001	001	ADR ← μROM
0001100	0010	*	=	0	1111 11	11	010	001	ADR ← A
0001100	0011	*	=	1	1111 11	11	001	000	Opcode ← μROM; ADR+

Nanocode for jmp microinstruction

אנו שומרים ברגיסטר A את הבית הראשון (המכיל את הכתובת הנמוכה), אחרי זה הבית השני (המכיל את הבית הגבוה) נכנס אל ADRLO. בשלב הבא אנו מכניסים את התוכן הנוכחי של ה μROM אל ADR, ולאחריו מכניסים את הבית הנמוך, ששמרנו לפני כן בA.

הטבלה הבאה מדגימה ננו קוד אופייני של ננו קוד של פעולת קפיצה מותנית.

Opcode	Phase	COND	=	ADR+	ALU	CC	DRSEL	LDSEL	Comments
00010111	0000	*	=	1	1111 11	11	001	010	A ← μROM; ADR+
00010111	0001	*	=	0	1111 11	01	010	010	Shift CCs
00010111	0010	1	=	0	1111 11	11	001	001	ADR ← μROM
00010111	0011	1	=	0	1111 11	11	010	001	ADR ← A
00010111	0100	1	=	1	1111 11	11	001	000	Opcode ← μROM; ADR+
00010111	0010	0	=	1	1111 11	11	001	010	A ← μROM; ADR+
00010111	0011	0	=	1	1111 11	11	001	000	Opcode ← μROM; ADR+

Nanocode for jnc microinstruction

אנו מביטים בננו קוד של הפקודה jnc, הקופצת אם לא היה carry בתוצאת החישוב האחרון.
ביט הcarry הוא הביט השני במילה, ולכן אנו צריכים לעשות shift אחד על מנת לקבל אותו ב phase 1.
כמו במיקרו פקודה jmp, אחרי הopcode שמור בזיכרון שני בתים, הנמוך והגבוה, של הכתובת אליה קופצים.
אנו מקדמים את הADR, כדי בתים אלו לא יפוענחו בהמשך בתור opcodes.

הטבלה הבאה מסכמת את מיקרו הפקודות השונות המשמשות לקפיצה בMAYBE:

Symbolic microinstruction	Control-ROM encoding	Operation performed
`jmp(`*dst*`)`	0x0C *dstlo dsthi*	Unconditional transfer: ADR ← (*dsthi*, *dstlo*)
`je(`*dst*`)`	0x19 *dstlo dsthi*	`jmp` if ALU output was = 0b11111111
`jne(`*dst*`)`	0x25 *dstlo dsthi*	`jmp` if ALU output was ≠ 0b11111111
`jmi(`*dst*`)`	0x12 *dstlo dsthi*	`jmp` if ALU output was < 0
`jc(`*dst*`)`	0x23 *dstlo dsthi*	`jmp` if carry
`jnc(`*dst*`)`	0x17 *dstlo dsthi*	`jmp` if no carry
`jeven(`*dst*`)`	0x22 *dstlo dsthi*	`jmp` if ALU output was even
`jodd(`*dst*`)`	0x16 *dstlo dsthi*	`jmp` if ALU output was odd
`jready(`*dst*`)`	0x14 *dstlo dsthi*	`jmp` if I/OFLAG is set
`jp0(`*dst*`)`	0x18 *dstlo dsthi*	`jmp` if P_0 is pressed
`jp1(`*dst*`)`	0x15 *dstlo dsthi*	`jmp` if P_1 is pressed
`jnp0(`*dst*`)`	0x24 *dstlo dsthi*	`jmp` if P_0 is not pressed
`jnp1(`*dst*`)`	0x21 *dstlo dsthi*	`jmp` if P_1 is not pressed

dispatch

מיקרו פקודה חשובה נוספת היא dispatch,
שזוהי קפיצה לפי ערך מספרי המועבר לפקודה.
שימוש לפקודה זו היא העברת בקרה כאשר יש למשל, n פקודות מכונה, שעבור כל אחת מהן אנו רוצים להפעיל רצף שונה של מיקרו פקודות.

Symbolic microinstruction	Control-ROM encoding	Operation performed
dispatch(x, tab)	0x80 x tabhi tablo	*tab* is taken as the μROM address of a table of 2-byte μROM addresses, each stored (low, high); the SRAM location *x* contains an index into the table; jumps to the μROM location given in the indexed entry.

tab זו הכתובת הנמוכה של טבלה המכילה 256 כניסות, שכל אחת מהן בגודל שני בתים.
לכן, הכתובת של מיקום בזיכרון המזוהה על ידי $\langle x \rangle$, הוא $tab + \langle x \rangle + \langle x \rangle$.
אנו מניחים שבתא זה ישנה כתובת אחרת בMicrocode ROM אליה ריצת התוכנית תעבור.

שאלות שונות אודות הMAYBE

לפני שנמשיך להכיר את הMAYBE, נציג מספר שאלות הקשורות לאספקטים שונים של הMAYBE בהם עסקנו עד כה.

1. נניח שנחליף בשוגג בין שני חוטים המחברים את הMAR אל הSRAM. האם פעולת הSRAM תשתנה? ומה יקרה אם נחליף בין החיבורים של הADRHI/ADRLO בMicrocode ROM? אם יש שינוי, מה גורם להבדל בין הSRAM לMicrocode ROM?

2. מה, אם בכלל, מונע משני רכיבים בMAYBE לשלוח בו זמנית נתונים על הBUS וליצור התנגשות? האם התנגשויות יתכנו אם ישנן שגיאות תכנות בMicrocode ROM?

3. בהינתן Control ROM גדול מספיק, האם נוכל להיפטר מהLDSEL decoder ומהDRSEL decoder, ולייצר את אותות הבקרה ישירות מיציאות ה Control ROM? אם כן, אילו יתרונות נוכל להפיק מכך?

4. האם פקודת ננוקוד יכולה להיות מושפעת מהמידע המצוי בBUS? אם כן, כיצד תוכנית ננוקוד יכולה להחליט החלטות לפי הנתונים שעל הBUS?

5. בהינתן Control ROM גדול מספיק, האם נוכל להיפטר מה condition shift register, ולחבר את הcondition bits ישירות אל כניסות הControl ROM? אם כן, אילו יתרונות נפיק מכך? כמה כניסות ויציאות נצטרך להוסיף או להוריד לROM? מה יהיה גודל הControl ROM החדש (בביטים)?

6. הננופקודה הבאה בוחרת את הMicrocode ROM בתור מקור הנתונים, וגם מבצעת +ADR בו זמנית. מה מתרחש? האם הכתובת המקורית או הכתובת החדשה בADR משמשות כאשר אנו ניגשים לMicrocode ROM?

Opcode	Phase	COND	=	ADR+	ALU	CC	DRSEL	LDSEL	Comments
********	****	*	==	1	1111 11	11	001	000	Opcode ← μROM; ADR+

תשובות

1.
שינוי הסדר בין החוטים המחברים את הMAR אל הSRAM לא יגרום שום אפקט שניתן לגלותו, מכיוון שיהיה מיקום ייחודי בSRAM עבור כל ערך של MAR.
המיקום הממשי בו ישכנו הנתונים בSRAM ישתנה, אולם לא ניתן לראות את זה.
החלפת ADRHI/ADRLO לעומת זאת תגלה, מכיוון שכאשר שנגדיר את הרגיסטרים על מנת לקרוא את הפקודה הבאה, נראה כי איננו מקבלים את תוכן העוקבת. אם נשנה גם את תוכן הMicrocode ROM כדי להתאים לשינוי בכתובות, אשר השינוי לא יתגלה.

2.
הdrivers של הBUS נשלטים על ידי 3-to-8 decoder. עבור כל צירוף של 3 ביטים, ההתקן מוציא רק יציאה אחת החוצה, לכן לא יתכנו התנגשויות.
שגיאות תכנות עלולות לגרום למידע הלא נכון להיות על הBUS, אך לא יכולות לגרום למספר ערכים להיכתב אל הBUS בו זמנית.

3.
נוכל להיפטר מהdecoders, כל ידי החלפת כל ערך בקרה בן 3 ביטים הנכנס לdecoder בערך בעל 8 ביטים שיהיה מחובר ישירות אל אותות הבקרה של LDxx, DRxx.
כל אות יוכל לעלות באופן עצמי, כאשר יתכן שיעלו אותות נוספים מלבדו.
שיפור זה לא יועיל לאותות DRxx, מכיוון שאיננו רוצים לגרום התנגשויות, אולם נוכל לגרום למספר רגיסטרים שונים להיטען עם אותו הנתון בו זמנית, על ידי העלאת הLDxx שלהם בו זמנית.

4.
הביט הנמוך של כתובת בControl ROM מגיע מרגיסטר הזזה שנטען על ידי הALU בCondition codes. ניתן לכתוב ננופקודות שלא יתעלמו מביט זה (על ידי כתיבת שתי ננופקודות עבור אותו הphase - אחת עבור המקרה שביט זה הוא 0 והשניה עבור המקרה שביט זה הוא 1), וכך ננופקודה תוכל לפעול באופן שונה בהתאם לפלט של הshift register.
על ידי הזזת הshift register לפני הפעלת פקודת הננוקוד המגיבה לביט זה, נוכל לגרום לננו-תוכנית להגיב לכל סיביות הבקרה השונות.

5.
נוכל להיפטר מהcondition shift register, ולחבר את הcondition bits ישירות אל כניסות הControl ROM. נחבר את כל 7 קודי הבקרה כאותות כתובת נוספים, וכתוצאה מכך נוכל להריץ בכל phase אחת מ128 הוראות שונות.
עם 7 כניסות כתובת נוספות, הControl ROM יגדל מ 2^{13} כתובות ל 2^{19} כתובות.
מספר יציאות הControl ROM יקטן ב1, מכיוון שנוכל להקטין הCONDCTL משני ביטים לביט בודד.

6.
הגדלת ADR מתרחשת בסוף מחזור השעון (כלומר, בעליה הבאה של השעון), לכן עבור מחזור השעון הנוכחי, הכתובת המקורית היא זו שבשימוש.

microsubroutines

היכולת לייצר שגרות (פונקציות) היא אחד הכלים הבסיסיים והחשובים של קוד מבני.
פונקציה היא קטע קוד אשר ניתן לקרוא לו מכמה חלקים בתוכנית. לאחר ריצת הפונקציה, הריצה ממשיכה מקטע הקוד שקרא לפונקציה.

על מנת שמכונה כלשהי תתמוך בשגרות, היא צריכה לתמוך בשני מנגנונים:
1. מנגנון להעברת בקרה:
   - קריאה לשגרות.
   - חזרה משגרות.
2. מנגנון להעברת פרמטרים.

העברת בקרה

הפתרון הממומש צריך לכלול פקודות call וreturn עבור השגרות.
הבעייתיות העיקרית ביצירת שגרות היא לנהל את כתובת החזרה (הכתובת אליה חוזרים כשפעולת הפונקציה מסתיימת).
פתרון נאיבי יכול להיות שימוש ברגיסטר מיוחד שישמור את כתובת החזרה.
פתרון כזה לא טוב כי:
1. השיגרה עלולה לשנות את מצב המכונה. מנגנון הקריאה והחזרה צריך לשמור את ערכי הרגיסטרים ולשחזר אותם.
2. לא ניתן לקנן שגרות (עד אין סוף), מכיוון שיש מספר מוגבל של רגיסטרים.

תמיכה במנגנון תמיכה בשגרות ללא חסם עליון ניתנת על ידי מחסנית.

נכנה את המחסנית במחשב הMAYBE בשם "microstack".
הבית העליון בSRAM יכיל את כתובת המחסנית. נכנה אותו בשם μSP (micro stack pointer).
המחסנית נבנית בSRAM כלפי מטה. נסכים כי μSP מכיל את הכתובת הבאה בה אנו רוצים לשים מידע. לכן הפקודה push תשים את המידע בכתובת $\langle \mu SP \rangle$ ולאחריה תקטין את μSP ב1.
באופן דומה, הפקודה pop קודם תגדיל את μSP ב1, ואז תשלוף את המידע מהכתובת החדשה אליה הגענו.

Opcode	Phase	COND	=	ADR+	ALU	CC	DRSEL	LDSEL	Comments
00000011	0000	*	=	0	1100 11	11	010	101	MAR ← 0xFF
00000011	0001	*	=	0	1111 11	01	100	010	A ← SRAM
00000011	0010	*	=	0	111 10	11	010	100	SRAM ← A -1
00000011	0011	*	=	1	1111 11	11	001	101	MAR ← μROM; ADR+
00000011	0100	*	=	0	1111 11	11	100	011	B ← SRAM
00000011	0101	*	=	0	1111 11	11	010	101	MAR ← A
00000011	0110	*	=	0	1010 11	11	010	100	SRAM ← B
00000011	0111	*	=	1	1111 11	11	001	000	Opcode ← μROM; ADR+

Nanocode of push microinstruction

אופן מימוש הפקודה:
ראשית נשים בMAR את כתובת מצביע המחסנית (0xFF). לאחר מכן, ניקח את תוכן תא זה, ונשמור אותו בA. ערך זה הוא הכתובת של התא הפנוי הבא.
במחזור הבא נשמור את A-1 בכתובת 0xFF, כלומר נגרום למצביע המחסנית להצביע לתא הפנוי הבא. ברגיסטר A נשאר המקום הפנוי הקודם.
בשני השלבים הבאים נטען מן הMicrocode ROM את הערך אותו אנו רוצים לדחוף למחסנית, ונשמור אותו זמנית ברגיסטר B. לבסוף, נטען את A, הכתובת הקודמת של מצביע המחסנית, אל הMAR, ואז נשים בכתובת זו את B.
בזאת סוים המימוש של המיקרו פקודה push.

המיקרו פקודה call דומה מאוד לקפיצה ללא תנאי, למעט העובדה שהיא דוחפת את כתובת החזרה למחסנית.
הפקודה return שולפת מהמחסנית את הכתובת אליה חוזרים, ומתקינה אותו על הADR.

נשים לב לעובדה חשובה: בדרך כלל אנו עובדים בשיטת Little Endian, בה הבית הנמוך מכיל את החלק הנמוך של הכתובת, בשמירה במחסנית אנו עובדים בשיטת Big Endian, בה הMSB נשמר בבית הנמוך יותר.
אנו עושים זאת כי על ידי צורת עבודה זו אנו מסוגלים לחסוך מחזור שעון אחד או שניים במיקרו פקודות של call וrtn.

להלן המימוש של call ו rtn:

Opcode	Phase	COND	=	ADR+	ALU	CC	DRSEL	LDSEL	Comments
00000010	0000	*	=	0	1100	11	010	101	MAR ←0xFF
00000010	0001	*	=	0	1111	11	100	010	A ←SRAM
00000010	0010	*	=	0	0000	00	010	010	$A \leftarrow A + 1$
00000010	0011	*	=	0	1111	11	010	101	MAR ←A
00000010	0100	*	=	0	1111	11	100	001	ADR ←SRAM
00000010	0101	*	=	0	0000	00	010	010	$A \leftarrow A + 1$
00000010	0110	*	=	0	1111	11	010	101	MAR ←A
00000010	0111	*	=	0	1111	11	100	001	ADR ←SRAM
00000010	1000	*	=	0	1100	11	010	101	MAR ←0xFF
00000010	1001	*	=	0	1111	11	010	100	SRAM ←A
00000010	1010	*	=	1	1111	11	001	000	Opcode ←μROM; ADR+

Nanocode of rtn microinstruction

Opcode	Phase	COND	=	ADR+	ALU	CC	DRSEL	LDSEL	Comments
00000001	0000	*	=	0	1100	11	010	101	MAR ←0xFF
00000001	0001	*	=	0	1111	11	100	010	A ←SRAM
00000001	0010	*	=	0	1111	11	010	101	MAR ←A
00000001	0011	*	=	1	1111	11	001	100	SRAM ←μROM; ADR+
00000001	0100	*	=	0	1111	10	010	010	$A \leftarrow A - 1$
00000001	0101	*	=	0	1111	11	010	101	MAR ←A
00000001	0110	*	=	1	1111	11	001	100	SRAM ←μROM; ADR+
00000001	0111	*	=	0	1100	11	010	101	MAR ←0xFF
00000001	1000	*	=	0	1111	10	010	100	SRAM ←$A - 1$
00000001	1001	*	=	1	1111	11	001	010	A ←μROM; ADR+
00000001	1010	*	=	0	1111	11	001	001	ADR ←μROM
00000001	1011	*	=	0	1111	11	010	001	ADR ←A
00000001	1100	*	=	1	1111	11	001	000	Opcode ←μROM; ADR+

Nanocode of call microinstruction

הטבלה הבאה מסכמת את הפקודות הקשורות לפונקציות שראינו:

Symbolic microinstruction	Control-ROM encoding	Operation performed
push(x)	0x03 x	Push SRAM $\langle x \rangle$ onto microstack: $\langle \mu SP \rangle \leftarrow \langle x \rangle; \mu SP \leftarrow \langle \mu SP \rangle - 1$
pop(x)	0x04 x	Pop mictostack into SRAM $\langle x \rangle$: $\mu SP \leftarrow \langle \mu SP \rangle + 1; x \leftarrow \langle\langle \mu SP \rangle\rangle$
call(s)	0x01 rlo rhi slo shi	Mictosubroutine call to microcode ROM address s; r is the return location: $\langle \mu SP \rangle \leftarrow rlo; \mu SP \leftarrow \langle \mu SP \rangle - 1;$ $\langle \mu SP \rangle \leftarrow rhi; \mu SP \leftarrow \langle \mu SP \rangle - 1;$ $ADR \leftarrow shi; ADR \leftarrow slo$
rtn()	0x02	Microsubroutine return: $\mu SP \leftarrow \langle \mu SP \rangle + 1; x \leftarrow \langle\langle \mu SP \rangle\rangle;$ $\mu SP \leftarrow \langle \mu SP \rangle + 1; x \leftarrow \langle\langle \mu SP \rangle\rangle$

נסכם את נושא העברת הבקרה השגרות:
ביצוע שגרה כולל קפיצה לתחילת השגרה, ביצוע פעולות השגרה וחזרה לתוכנית הקוראת.
כדי לדעת את כתובת החזרה נרשום אותה במחסנית בזמן ביצוע הcall ונקרא אותה בזמן ביצוע return.

בחומרת ה-MAYBE אין אפשרות לקרוא את כתובת הADR (הPC) אל ה-bus, ולכן יש לרשום את כתובת החזרה כפרמטר לפקודה call.

מנגנון להעברת פרמטרים

כאשר אנו מתכנתים את הMicrocode ROM, יש לנו מספר דרכים להעביר פרמטרים לפונקציה.
נסביר כעת בכלליות את השיטות השונות, ללא דוגמאות. לאחר שנלמד את התחביר של שפת האסמבלי, נשוב ונראה דוגמאות כיצד משתמשים במנגנונים אלו.

הדרך הראשונה להעביר פרמטרים, היא צריבתם אחרי הפקודה.
למשל, נניח כי נצרוב לROM את הצירוף הבא:

```
0x0B 10 0 0
```

0x0B הוא הOpcode של cadd. לאחר הOpcode באים שלושה פרמטרים - הקבוע אותו אנו רוצים להוסיף, רגיסטר המקור ורגיסטר התוצאה.
פעולת פקודה זו היא להוסיף לרגיסטר 0 את הערך 10.
במקרה זה העברנו את הפרמטרים על ידי צריבתם על הMicrocode ROM.

דרך שניה להעביר פרמטרים לפונקציה על ידי רגיסטר קבוע, למשל, נוכל לכתוב פונקציה ולהניח שהיא תמיד תקבל את הפרמטרים שלה בתא 0 בSRAM, למשל, ותחזיר את הפלט שלה בתא 1 בSRAM.

הדרך השלישית להעברת פרמטרים היא דרך המחסנית.
לפני שנכנס לפונקציה, נדחוף בעזרת push את הפרמטרים הרצויים לmicrostack , והשגרה תיגש למחסנית ותקרא את הפרמטרים.
נדגיש כי בהמשך נראה מימוש של מכונת G על מחשב ה-MAYBE. למרות שתיאורטית ניתן להשתמש בדרך זו להעברת פרמטרים, המימוש המוצע איננו עושה שימוש בדרך זו.

שגרות והמחסנית

כאשר נקרא לשגרה, לא נרצה שהיא תשנה את ערכם של הרגיסטרים השונים במהלך פעולתה. מאידך, נרצה לאפשר לשגרה לבצע חישובים ולשמור תוצאות זמניות.
הפתרון: נשמור את הרגיסטרים בהם נשתמש במחסנית בתחילת הפונקציה (על ידי פקודות push), ולפני היציאה מהפונקציה נטען מהמחסנית את ערכי הרגיסטרים המקוריים (על ידי pop).
חוק חשוב שהשגרה צריכה לקיים, הוא שמירה על מצב המחסנית. בסוף השגרה על המחסנית להיות באותו מצב בו היא הייתה בתחילת השגרה.
הסיבה לכך היא שבסוף השגרה, נשלוף מהמחסנית את כתובת החזרה. אם נזבל את המחסנית, לא נחזור אל המקום ממנו הגענו לפונקציה.

העברת נתונים בזיכרון הMAYBE

הטבלה הבאה מסכמת פקודות המאפשרות העברת נתונים בין תאים בSRAM.

Symbolic microinstruction	Control-ROM encoding	Operation performed
move(x, y)	0x05 x y	$y \leftarrow \langle x \rangle$
cmove(cx, y)	0x06 cx y	$y \leftarrow cx$
imove(x, y)	0x0D x y	$y \leftarrow \langle\langle x \rangle\rangle$
movei(x, y)	0x0E x y	$\langle y \rangle \leftarrow \langle x \rangle$

הפקודות imove(x,y) וmovei(x,y) שימושיות כדי ליצור מחסנית נוספת בSRAM.

הטבלה הבאה מציגה פקודות שונות אשר מאפשרות לגשת לDRAM בעזרת מיקרו קוד.

Symbolic microinstruction	Control-ROM encoding	Operation performed
l(adrlo, adrhi, where)	0x0F adrlo adrhi where	where $\leftarrow \langle\langle \text{adrlo} \rangle, \langle \text{adrhi} \rangle\rangle$; loads contents of DRAM location whose address is contained in SRAM locations *adrlo* and *adrhi* into SRAM location *where.*
s(where,adrlo,adrhi)	0x10 where adrlo adrhi	Stores contents of SRAM location *where* into DRAM location whose address is contained in SRAM locations *adrlo* and *adrhi*.
refr()	0x11	Refreshs six rows of dynamic RAM

פקודות אלו מכילות פקודות שמירה וטעינה (l, s) שהפרמטרים שלהם הם כתובות בנות 16 ביט בזיכרון הראשי, שמועברים על ידי שני תאים בSRAM.

פקודת refresh

הפקודה refr מרעננת שש שורות בDRAM. פקודה זו משתמשת בכתובת 0xFE בSRAM בתור refresh counter (RC), שגדל כל פעם ששורה מרוענת. מונה זה שומר את מספר השורה הבאה לה צריך לבצע ריענון. בצורה זו הרצת refr תבטיח שכל 128 השורות רוענו כל 128/6 קריאות לrefr. (למעשה יש 256 שורות, מכיוון שכל שורה נקבעת על ידי כתובת בת 8 ביטים. כל פעולת רענון מרעננת למעשה שתי שורות בו זמנית).

הפקודה עושה שימוש בשיטת RAS only refresh. $\overline{LDDRAM}$ יורד למחזור אחד. הDRAM מתעלם מסיבית הMSB ולכן מבצע רענון ל2 שורות. ניתן לבצע 6 ירידות $\overline{RAS}$ במחזור בן 16 פאזות, בכל מחזור מרוענות 2 שורות, לכן $\frac{256}{2 \cdot 6} = 22$.

הטבלה הבאה מציגה את הננו קוד של הפקודה refr.

Opcode	Phase	COND	=	ADR+	ALU	CC	DRSEL	LDSEL	Comments
00010001	0000	*	=	0	1100	11	010	010	$A \leftarrow$0xFF
00010001	0001	*	=	0	1111	10	010	101	MAR $\leftarrow A - 1$ (= 0xFE)
00010001	0010	*	=	0	1111	11	100	010	$A \leftarrow$SRAM
00010001	0011	*	=	0	1111	11	010	110	DRAM $\leftarrow A$
00010001	0100	*	=	0	0000	00	010	010	$A \leftarrow A + 1$
00010001	0101	*	=	0	1111	11	010	110	DRAM $\leftarrow A$
00010001	0110	*	=	0	0000	00	010	010	$A \leftarrow A + 1$
00010001	0111	*	=	0	1111	11	010	110	DRAM $\leftarrow A$
00010001	1000	*	=	0	0000	00	010	010	$A \leftarrow A + 1$
00010001	1001	*	=	0	1111	11	010	110	DRAM $\leftarrow A$
00010001	1010	*	=	0	0000	00	010	010	$A \leftarrow A + 1$
00010001	1011	*	=	0	1111	11	010	110	DRAM $\leftarrow A$
00010001	1100	*	=	0	0000	00	010	010	$A \leftarrow A + 1$
00010001	1101	*	=	0	1111	11	010	110	DRAM $\leftarrow A$
00010001	1110	*	=	0	0000	00	010	100	SRAM $\leftarrow A + 1$
00010001	1111	*	=	1	1111	11	001	000	Opcode $\leftarrow \mu$ROM; ADR+

פקודת load

הפקודה load שולפת מידע מהDRAM ושמה אותו בSRAM.

$l(adrlo, adrhi, where)$

adrlo, *adrhi* מכילים מספרי רגיסטרים בSRAM.
הפקודות היחידות שאין להם רגיסטרים בSRAM הן פקודות עם קבועים כגון cadd, cmove.

- *adrlo* - כתובת שורה בDRAM.
- *adrhi* - כתובת עמודה בDRAM
- *where* - רגיסטר התוצאה בSRAM.

פקודה זו מבצעת טעינת בית מהDRAM לSRAM. הבית שכתובתו מצויה ב *adrlo*, *adrhi* בDRAM, מועתק לתא *where* בSRAM.

$\overline{LDDRAM}$ צריך לרדת 2 מחזורים על מנת לבצע קריאה - $LDSEL_{110}$.

הטבלה הבאה מציגה את הננו קוד של load.

Opcode	Phase	COND	=	ADR+	ALU	CC	DRSEL	LDSEL	Comments
00001111	0000	*	=	1	1111	11	001	101	MAR ←μROM; ADR+
00001111	0001	*	=	0	1111	11	100	010	*A* ←SRAM
00001111	0010	*	=	1	1111	11	001	101	MAR ←μROM; ADR+
00001111	0011	*	=	0	1111	11	010	110	DRAM ←*A* (RAS)
00001111	0100	*	=	0	1111	11	100	110	DRAM ←SRAM (CAS)
00001111	0101	*	=	1	1111	11	001	101	MAR ←μROM; ADR+
00001111	0110	*	=	0	1111	11	110	100	SRAM ←DRAM
00001111	0111	*	=	1	1111	11	001	000	Opcode ←μROM; ADR+

Nanocode for l(adrlo, adrhi, where) microinstruction

שפת האסמבלי (שפה סימבולית)

נרצה להגדיר שפת תכנות סימבולית שנכנה אותה שפת אסמבלי.
סימנים בשפה שנגדיר יתורגמו למספרים בינריים שיכנסו אחר כך ל Microcode ROM.
האסמבלר היא התוכנה שתבצע את הפעולה של התרגום. האסמבלר יקבל סימנים סימבוליים ויהפוך אותם לרצף מספריים בינריים.
מרכיבי השפה:

1. **מספרים:**
כשנכתוב מספר, הכוונה תהיה לקחת את המספר ולהכניס אותו לתא בMicrocode ROM. נאמר לאסמבלר שאם הוא יתקל במספרים בתוך הסימנים אותם הוא מפענח, הוא יפרש אותם לייצוג הבינרי של אותם המספרים.
אם נרצה לציין שמספר מסוים כבר מופיע בבסיס בינרי, נסמן זאת על ידי התחילית 0b, למשל: 0b01011.
אם נרצה לציין שמספר מסוים מופיע בבסיס הקסדצימלי, נוסיף לפניו את התחילית 0x, למשל: 0xFF.
2. **פעולות אריתמטיות או לוגיות:**
פעולות כגון חיבור, חיסור, כפל, חילוק, שארית ועוד. נוכל לכתוב בשפה ביטויים מתמטיים שונים, למשל 3+4, -8*2, וכו'. החישוב יעשה על ידי האסמבלר. בMicrocode ROM תקודד תוצאת החישוב, ולכן פעולות אריתמטיות שונות לא יאטו את ריצת המיקרו תוכנית.
3. **סימבולים:**
נוכל לתת שמות סימבוליים לקבועים. נוכל למשל לכתוב r1=1, ולהגיד בכך שבכל מקום בו תופיע המחרוזת r1, יושם הערך 1.
4. **נקודה:**
נקודה מייצגת את הכתובת של התא הבא בMicrocode ROM שימולא בתוכן.
למשל, נניח כי בתחילת הקוד שלנו נכתוב את המחרוזת הבאה:
. . . . 2 − 3 +, אזי ארבעת הבתים הראשונים בMicrocode ROM ימולאו ב0,4,2,1, בהתאמה.
5. **תוויות:**
תוויות מזוהות על ידי מילה ונקודותיים אחריה, למשל:
label:
הגדרה זו נותנת לlabel את הערך של "." (נקודה). ניתן אחר כך להתייחס לכתובת בה נמצאת התווית באופן סימבולי בעזרת שם התווית.
6. **הגדרות מאקרו:**
אנחנו משתמשים בפקודות מאקרו כדי לתת שם סימבולי לפקודת מיקרו אחת או יותר. הגדרת מקרו מתחילה ב"macro.", לאחריה שם המאקרו בצירוף הפרמטרים שלו, ולאחריו גוף המאקרו, שזהו קטע הקוד שיושתל בתוכנית בכל מקום בו יופיע צירוף התווים שמציין את המאקרו. למשל, המאקרו cadd מוגדר כך: "macro cadd(cx,y,z) 0x0B cx y z.". מקרו זה יוצר קריאה למיקרו פקודה cadd שתיארנו קודם.
לאחר שהגדרנו את המאקרו, המתכנת יוכל לכתוב, למשל, את הפקודה cadd(7,R2,R1) כאשר הוא ירצה לבצע את הפעולה $R1 \leftarrow 7 + \langle R2 \rangle$.

כאשר אנו כותבים תוכנית, נהוג להפריד את הגדרות המאקרו פקודות מהמיקרו תוכנית, לשים אותם בקובץ נפרד ולצרף אותם אל המיקרו פקודות אל ידי הצהרה כגון " include macros. " בראש קובץ המיקרו קוד. שורה זאת אומרת לאסמבלר להוסיף בנקודה זו את תוכן הקובץ macros. בצורה כזו, תהליך המרת פקודות המיקרו קוד לערכים הבינריים שלהם שקוף מתכנת המיקרוקוד. ניתן בצורה כזו לשנות את הקודים המייצגים את המיקרו פקודות (למשל בשיפור עתידי של הMAYBE), ללא צורך בשינוי המיקרו תוכנית שנכתבה עבור הMAYBE.

הטבלה הבאה מדגימה הגדרת של מספר Macros, בהם אנו מגדירים Macros למספר מיקרו פקודות.

נדגיש את המאקרו WORD - מאקרו המקבל ערך ופורס אותו על שני בתים של שש עשרה סיביות. המאקרו לוקח את הערך, מחלק אותו לMSB וLSB, ושם אותם בMicrocode ROM.

נשים לב למאקרו פקודה dispatch. בפקודה זו אנו שומרים את הכתובת ה Big Endian ולא בLittle Endian כפי שסיכמנו ששומרים את המספרים בMAYBE. הסיבה לכך היא מאילוצי מימוש הפקודה - זו הדרך היחידה בה ניתן לעמוד ב16 ננו פקודות למימוש הפקודה.

```
.macro WORD(x)                  x%256 x/256
.macro move(x,y)                0x05 x y
.macro cmove(cx,y)              0x06 cx y
.macro call(s)                  0x01 (.+4)%256 (.+3)/256 WORD(s)
.macro rtn()                    0x02
.macro push(x)                  0x03 x
.macro pop(x)                   0x04 x

.macro add(x,y,z)               0x0A x y z
.macro sub(x,y,z)               0x1C x y z
.macro cadd(cx,y,z)             0x0B cx y z
.macro addcy(x,y,z)             0x81 x y z
.macro negcy(x, y)              0x32 x y
.macro subcy(x,y,z)             0x33 x y z
.macro caddcy(cx,y,z)           0x82 cx y z
.macro cand(cx,y,z)             0x83 cx y z

.macro jmp(adr)                 0x0C WORD(adr)
.macro jmi(x)                   0x12 WORD(x)
.macro jp1(x)                   0x15 WORD(x)
.macro jnc(x)                   0x17 WORD(x)
.macro jp0(x)                   0x18 WORD(x)
.macro je(x)                    0x19 WORD(x)

.macro dispatch(x,adr)          0x80 x adr/256 adr%256
.macro pdisp(x,p00,p01,p10,p11)     0x08 x WORD(p00) WORD(p01) WORD(p10)
                                        WORD(p11)
```

מיקרו פקודות נוספות בMAYBE

swt(x)

מיקרו פקודה זו לוקחת את ערכי הswitches בMAYBE ושמה אותם בכתובת x בSRAM.

jp1(adrlo, adrhi), jp0(adrlo, adrhi)

המיקרו פקודה jp0 קופצת לכתובת שמועברת אליה אם הלחצן P0 לחוץ.
המיקרו פקודה jp1 קופצת לכתובת שמועברת אליה אם הלחצן P1 לחוץ.

הכוונה שלחצן לחוץ היא שערך P0, למשל, הוא 1, אם P0 לחוץ.

jnp1(adrlo, adrhi), jnp0(adrlo, adrhi)

המיקרו פקודה jnp0 קופצת לכתובת שמועברת אליה אם הלחצן P0 איננו לחוץ.
המיקרו פקודה jnp1 קופצת לכתובת שמועברת אליה אם הלחצן P1 איננו לחוץ.

count(x)

המיקרו פקודה count(x) מקטינה את תוכן הכתובת x בSRAM ב1, ומדליקה את הדגלים בהתאם לתוצאת הפעולה.

cond(x)

המיקרו פקודה cond(x) מעתיקה את תוכן רגיסטר הCC של הMAYBE אל הכתובת x בSRAM.

התחלת התוכנית

תוכנית מיקרו קוד תתחיל בדרך כלל ברצף של הגדרות Macros ומשתנים, בדומה לטבלה הבאה:

```
| Beginning of typical microprogram

.include macros

r0=0
r1=1
r2=2
r3=3
r4=4
r5=5
r6=6
r7=7
init:   pweup()
        cmove(SBASE, SP)
        jmp(top)
top: ...
```

נוסיף הערות לקוד על ידי | בתחילת הערה. מהרגע שהאסמבלר רואה | עד לתחילת השורה הבאה, הוא מתעלם מהכתוב.

השורה הבאה אומרת לאסמבלר להוסיף את ההגדרות של הMacros המתאימים לMicroinstructions, כפי שהם מוגדרים בקובץ macros.uasm.

```
.include macros                    | Definitions of microinstructions
```

לאחריה אנו מגדירים משתנים שונים, כמו R1, ומציינים למעשה שאלו הכתובות המתאימות בSRAM. בדרך כלל הגדרות אלו, שמופיעות בכל התוכניות, מופיעות בקובץ נפרד אותו אנו מצרפים, כך שלא נצטרך להכניסם בכל מיקרו תוכנית.

בדוגמא שלנו, מוגדרים לאחר ההגדרות הנפוצות, מספר משתנים x, framis, הייחודיים למיקרו תוכנית המסוימת בה אנו מתבוננים.

נזכור כי בMAYBE המחסנית גדלה מלמעלה למטה. יש למנוע התנגשות בין המשתנים של התוכנית למחסנית. בדרך כלל נשמור לפחות 40 בתים רק עבור המחסנית, עבור כל כתובות החזרה של הפונקציות בהן התוכנית תשתמש.

השורות הראשונות האמיתיות (אלו שנצרבות לMicrocode ROM) של הMicrocode מבצעת אתחול בסיסי של המערכת. הן נצרבות לMicrocode ROM החל מכתובת 0.

כאשר המכונה מופעלת, הרגיסטרים ADR וPHASE נקבעים ל0, וOP נטען על ידי התוכן של תא 0 של הMicrocode ROM.
קריאה זו (fetch) של המיקרו פקודה היושבת בתא 0 היא לא סטנדרטית, כי הADR לא עולה כאשר קוראים את הפקודה. לכן בphase=0 של הפקודה, ADR עדיין מכיל את הכתובת 0. המיקרו פקודה pwrup מתוכננת להתמודד עם מצב זה. זו פקודה שאינה עושה דבר, מלבד אתחול ADR.

הפקודה cmove(SBASE,SP) מאתחלת את רגיסטר הSP עם כתובת המחסנית של המכונה.
לבסוף, אנו קופצים אל התחלת המיקרו תוכנית, המסומנת על ידי התגית top, בעזרת הפקודה jmp(top).

דוגמא 1

נביט כעת במיקרו תוכנית פשוטה שיכולה להופיע אחרי הקוד ההתחלתי שהצגנו. התוכנית מעלה ב1 את ערכו של רגיסטר כל פעם ש P_1 נלחץ, ומורידה ב1 את ערך הרגיסטר בכל פעם ש p_0 נלחץ.

```
|   Up/down counter microprogram

count       = 32                | Program variable; current count

top:        cmove(0,count)      | Start at zero
loop:       jp0(down)           | P0 pressed
            jp1(up)             | P1 pressed
            jmp(loop)           | Nothing pressed

down:       cadd(-1, count)     | Decrement count
wait0:      jp0(wait0)          | Wait for P0 release
            jmp(loop)

up:         cadd(1, count)      | Increment count
wait1:      jp1(wait1)          | Wait for P1 release
            jmp(loop)
```

לתוכנית זו יש שתי בעיות רציניות:

1. היא איננה מוציאה פלט - לכן קשה לדעת אם היא פועלת כראוי או לא.
2. התוכנית מתעלמת מהצורך לרענן את הDRAM כל תקופת זמן.

כדי להתגבר על הבעיות הנ"ל, נשתמש במיקרו פקודה המיוחדת $\text{pdisp}(x, p_{00}, p_{01}, p_{10}, p_{11})$.
פקודה זו מבצעת את הפעולות הבאות:

1. טוענת את התא x מתוך ה-SRAM אל הרגיסטר B, המחובר ל-LEDים אשר מציגים את ערכו של x.

2. מבצעת מחזור רענון של ה-DRAM.
3. בודקת את מצב P_0 ו P_1, וקופצת לאחת מארבע מקומות. נקפוץ אל כתובת ה Micro ROM, לפי הפרמטר p_{ij} כאשר i הוא המצב של P_1 ו j הוא המצב של P_0. לכן נקפוץ ל p_{00} אם לא נלחץ שום כפתור, p_{10} אם P_0 לחוץ, p_{01} אם P_1 לחוץ ואל p_{11} אם שני הכפתורים לחוצים.

נביט בננו קוד המשמש למימוש pdisp:

```
| pdisp(x, p00, p01, p10, p11)
| Displays <x> in the lights.  Dispatches to (2-byte) location
|   pyz where y is state of P0, z is state of P1.
| Does not put data other than <x> in lights, hence suitable for
|   tight "prompt" loop.
  00001000 0000 *  = 1 1111 11 1   00 001 101   | Latch CCs; MAR <- uROM; ADR+
  00001000 0001 *  = 0 1111 11 1   01 100 011   | B <- SRAM; Shift CCs
  | Following 3 states shift P0 into CCs, and load 0xFE (refresh counter adr)
  |   into SRAM MAR.
  00001000 0010 *  = 0 1100 11 1   01 010 010   | Shift CCs; A <- 0xFF
  00001000 0011 *  = 0 1111 10 1   01 010 101   | Shift CCs; MAR <- A-1 (=0xFE)
  00001000 0100 *  = 0 1111 11 1   01 010 010   | Shift CCs
  | In either P0 case, they refresh one DRAM column.
  | If P0=1, uROM adr is incremented 4 times as well to point to p10 in uROM.
  00001000 0101 n  = n 1111 11 1   11 100 010   | if P0, ADR+; A <- SRAM
  00001000 0110 n  = n 1111 11 1   11 010 110   | if P0, ADR+; DRAM <- A
  00001000 0111 n  = n 0000 00 1   11 010 100   | if P0, ADR+; SRAM <- A+1
  00001000 1000 n  = n 1111 11 1   01 010 010   | if P0, ADR+; Shift CCs

  00001000 1001 n  = n 1111 11 1   11 010 010   | if P1, ADR+ (else NOP)
  00001000 1010 n  = n 1111 11 1   11 010 010   | if P1, ADR+ (else NOP)

  00001000 1011 *  = 1 1111 11 1   11 001 010   | A   <- uROM; ADR+
  00001000 1100 *  = 0 1111 11 1   11 001 001   | ADR <- uROM
  00001000 1101 *  = 0 1111 11 1   11 010 001   | ADR <- A
  00001000 1110 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+
```

בעזרת מיקרו פקודה חדשה זו, נוכל לכתוב את התוכנית שלנו בצורה נכונה יותר.
נביט במימוש המשופר של התוכנית:

```
|    Up/down counter microprogram

count       = 32                          | Program variable; current count

top:        cmove(0,count)                | Start at zero
wait:       pdisp(count,loop,wait,wait,wait,wait) | wait for no buttons pressed
loop:       pdisp(count,loop,up,down,top)

down:       cadd(-1,count)                | Decrement count
            jmp(wait)                     | Return to wait loop
up:         cadd(1,count)                 | Increment count
            jmp(wait)
```

<u>דוגמא 2</u>

נרצה לכתוב תוכנית שתחבר מספרים בני 4 בתים.
נשים לב כי לא ניתן לבצע פעולה כזו על ידי יצירת מיקרו פקודה חדשה, מכיוון שמיקרו פקודה מוגבלת ל16 צעדים שהיא מסוגלת לבצע.
לכן נכתוב Microcode שיבצע את הפעולה.
ראשית נציג מיקרו פקודה חדשה, addcy(x,y,z). פקודה זו זהה לadd(x,y,z), עם שינוי חשוב אחד. addcy(x,y,z) מתייחסת ל $\overline{C}$ מהCondition Register, כלומר היא מתייחסת לנשא (carry) מתוצאת add או addcy קודמת.

נביט בננו קוד של הפקודה:

```
| addcy(x, y, z)
| Like add(x, y, z), but adds in the incoming carry bit.
  10000001 0000 *  = 1 1111 11 1   01 001 101   | MAR <- uROM; ADR+; Shift CCs
  10000001 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  10000001 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  10000001 0011 *  = 0 1111 11 1   11 100 011   | B   <- SRAM
  10000001 0100 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  10000001 0101 c  = 0 1001 c0 1   00 010 100   | SRAM <- A+B+Carry, Latch CCs
  10000001 0110 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM;  ADR+
```

כעת, בעזרת addcy, נוכל לקודד חיבור של מספרים בני n בתים, עי ידי ביצוע n פעולות של חיבור בית בודד. הפעולה הראשונה תהיה פעולת add, ואחריה רצף של n-1 פעולות addcy.

נביט בפונקציה המממשת חיבור של משתנה שלם בן 4 בתים הממוקם ב $R0...R3$ ביחד עם משתנה שני הממוקם ב $R4...R7$.

בשיגרה זו העברנו את הפרמטרים דרך הרגיסטרים. ישנן מספר דרכים נוספות להעביר פרמטרים, שלא נציג כעת. אנו מסכימים מראש כי הקלט והפלט יהיו ברגיסטרים קבועים.
כמו כן הסכמנו כי הפרמטרים יהיו שמורים בשיטת Little Endian.

```
|  4-byte addition microsubroutine:
|  Takes x in R0-3, y in R4-7, and returns their sum in R0-3
|  Note: R0 and R4 are the LOW-ORDER ends of the long integers

add4:       add(R0,R4,R0)          | Add low order; cin = 0
            addcy(R1,R5,R1)        | Remaining adds use
            addcy(R2,R6,R2)        |      previous carry.
            addcy(R3,R7,R3)

        rtn()                      | Return to the caller
```

דוגמא 3

נרצה לכתוב שיגרה אשר תמחק את כל תוכן זיכרון הDRAM (תמלא אותו באפסים).
מכיוון שלוקח יותר זמן לאפס את כל תאי הזיכרון, מהזמן המקסימלי שיכול לעבור ללא רענון הDRAM, נוסיף פקודות refr בתור המיקרו קוד, כדי לרענן את הDRAM.
הערה חשובה לתכנות מיקרוקוד היא לזכור להוסיף מספיק פקודות refr בקטעי קוד ארוכים. אם זאת, ידוע כי כאפקט צדדי של כתיבת/קריאת שורה בDRAM, אותה שורה מתעדכנת, לכן עם מעט תחכום היינו יכולים לחסוך את הקריאה לפקודת refr בצורה מפורשת.

בדוגמא הבאה כן נשתמש בקריאה לrefr, לשם פשטות התוכנית.

```
|  Subroutine to clear DRAM to all zeros

clrdram:    cmove(0,R2)            | Value to write
            cmove(-1, R0)          | adr lo
            cmove(-1, R1)          | adr hi
            refr()                 | For good measure

clr1:       s(R2,R0,R1)            | Store a byte
            cadd(-1,R0,R0)         | Next lower adr
            je(clr2)               | Got back to 0xFF
            refr()                 | For good measure
            jmp(clr1)

clr2:       cadd(-1,R1,R1)         | Decrement high byte
            je(clr3)               | If done
            jmp(clr1)

clr3:       rtn()
```

דוגמא 4

נרצה לכתוב מיקרו פקודה חדשה, בשם show(x). הפקודה מדליקה (מציגה בצורה בינרית) את המספר הנמצא בכתובת x בSRAM על הנורות (LEDs) של MAYBE.

ראשית נממש את הפקודה. נניח כי הopcode שלה הוא 0x70. (opcode זה איננו בשימוש במיקרו קוד שבא עם הMAYBE).

```
| show(x) - show x on LEDs
  01110000 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  01110000 0001 *  = 0 1111 11 1   11 100 011   | B   <- SRAM
  01110000 0010 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+
```

נשים לב לשורה בודדת, למשל השורה הראשונה.

01110000 0000 * = 1 1111 11 **1** 11 001 101

בדוגמאות שראינו עד כה, לא ראינו את הביט המודגש בשורה זו.
ביט זה איננו מחובר במחשב הMAYBE, ולכן אנו מתעלמים מתוכנו. הפקודה לא תשתנה אם הביט יהיה 0 או 1, אם כי יתכנו ווריאציות למחשב הMAYBE, בהן כניסה זו כן תהיה מחוברת, ואז לא נוכל להתעלם מתוכנה.

לאחר שכתבנו את הפקודה, נוסיף מאקרו פקודה, על מנת שנוכל להשתמש בה.

```
.macro show(x)                0x70 x
```

אנו מצהירים על המאקרו show(x). בכל מקום שנקרא לו, האסמבלר ישים 0x70 ולאחריו את ערכה של x בMicrocode ROM.

כעת נפנה למימוש תוכנית הדגמה קצרה, שתדגים את השימוש בפקודה החדשה.

```
.include macros

r0=0

        pwrup()
        cmove(SBASE,uSP)

        cmove(4, r0)
        show(r0)
        jmp(end)

end:
        bpt()
```

אנו שמים את הערך 4 ברגיסטר r0, מציגים אותו על הLEDs ומסיימים את התוכנית.

דוגמא 5

נדרש להגדיר (במיקרו-קוד של מחשב MAYBE) מאקרו לחיבור שני מספרים באורך 4 בתים כל אחד. כל אחד משני המספרים מאוחסן בארבעה רגיסטרים סמוכים (כלומר ארבעה מענים עוקבים ב- SRAM). רשום מאקרו שייתן את המיקרו-קוד באורך מינימלי (כלומר מינימום בתים).

פתרון

```
.macro add4(x, y, z) add2(x,y,z) addcy(x+2,y+2,z+2) addcy(x+3,y+3,z+3)
```

ראשית נחבר את שני הבתים הראשונים על ידי המיקרו פקודה add2. לאחר מכן נחבר אחד אחרי השני את שני הבתים האחרים, בעזרת addcy, כדי לא להתעלם מנשא.

דוגמא 6

נתון קטע המיקרו-קוד הבא:

```
p1:
        call (p2)
        move (R1,R2)
p2:
        pop (R1)
        pop (R2)
        rtn ( )
```

כמו כן נתון, שלפני ביצוע p1, תוכן כל תא ב-SRAM (מלבד מען 0xFF) שווה למענו, כלומר <j> = j עבור j < 0xFF. האם אפשר לדעת על סמך הנתונים היכן תמשיך המיקרו-תוכנית לאחר סיום המיקרו-שגרה p2?

פתרון

לא ניתן על סמך הנתונים לדעת היכן תמשיך המיקרו תוכנית, מכיוון שלא נתון תוכן 0xFF שמכיל את המצביע על המיקרו מחסנית.
כאשר אנו קוראים לp2, בעזרת הפקודה call(p2), כתובת החזרה נדחפת אל המחסנית (שמיקומה לא ידוע לנו כי לא ידוע תוכנו של 0xFF).
p2 שולף את כתובת החזרה מהמחסנית אל R1 ואל R2.
כעת תוכנו של 0xFF זהה לתוכנו לפני הקריאה לפונקציה p2.
הפקודה rtn מבצעת אף היא שני pop מהמחסנית, על מנת לקבל את כתובת החזרה.
נסמן את תוכן 0xFF כBASE. מכיוון שכל תא מכיל את תוכנו, הפקודה תגדיר את כתובת החזרה בתור BASE+1: BASE+2.
אין לנו מידע על תוכן הכתובת 0xFF ולכן המיקרו תוכנית תמשיך למקום בלתי ידוע.

דוגמא 7

נתונה תכנית המיקרו-קוד הבאה:

```
                cmp(0x10,0x11)
                cadd(0x03,0x12,0x12)
                je(somewhere)
conti:          cadd(0x04,0x12,0x12)
somewhere:      cadd(0x05,0x12,0x12)
```

נתון שתוכן ה-SRAM לפני ביצוע הקוד הנ"ל הינו:

Address:	0x10	0x11	0x12	0x13	0x14	0x15
Value:	0x88	0x33	0xFC	0x14	0x11	0x16

מה יהיה תוכן כתובת 0x12 ב-SRAM לאחר ביצוע הקוד הנ"ל?

פתרון

הפקודה cmp(0x10,0x11) משווה בין שני תאים. היא איננה משנה אף כתובת.
הפקודה cadd(0x03,0x12,0x12) מוסיפה את הקבוע 3 לתוכן של תא 0x12 ושומרת את התוצאה בתא 0x12. התוכן של תא 0x12 הוא 0xFC לפני ביצוע החישוב, ולאחרי ביצוע החישוב 0xFC+3=0xFF.
je קופץ אם תוצאת החישוב האחרונה הייתה 0xFF,ולכן אנו קופצים לsomewhere .
אנו מוסיפים כעת 0x05 לתא 0x12. תוכנו של התא היה קודם 1-, ולכן בסופו של דבר, תוכן כתובת 0x12 ב-SRAM לאחר ביצוע הקוד הנ"ל תהיה 0x04.

הערה: E=1 אם בחישוב האחרון הייתה התוצאה 11111111, ולא משנה מה היתה הסיבה לכך.

ה-G-Machine

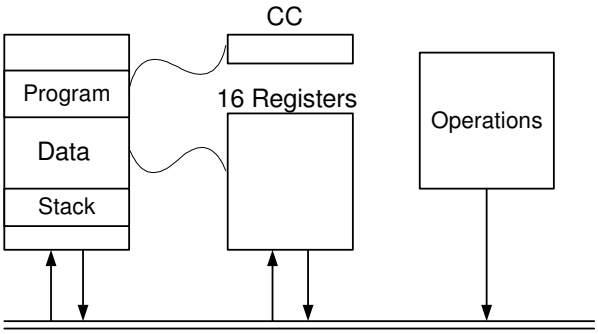
<u>שפת מכונה</u>

נגדיר מכונה וירטואלית שבה אפשר לבצע פקודות יותר מורכבות מאשר בMAYBE.
האינטרפרטר שיפענח פקודות אלו, ייקח אותן ויפרש אותן אחת אחת למיקרו פקודות.
במהלך תכנות בשפה כלשהי, ובפרט בשפת מכונה, מתכנת צריך לעיתים מקום אכסון מהיר, על מנת שיוכל לבצע בו חישובים קריטיים במהירות הגדולה ביותר האפשרית. G-Machine (General Register Machine) היא מכונה המאפשרת למתכנת בשפת המכונה לגשת ולהשתמש ברגיסטרים, שהם התקני זיכרון מהירים.
בחלק זה נכיר את ארכיטקטורה זו, ונראה מימוש של G-Machine על ארכיטקטורת Maybe.

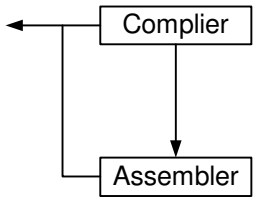
נשים לב לעובדה שעל ידי אינטרפרטרים שונים ניתן לממש מכונות שונות על ה-Maybe, ולא רק את ה-G-Machine.
כדי להפריד בין שפת ה μ של הMAYBE לבין שפת המכונה של G-Machine, נוסיף g לפני הפקודות של G-Machine.

<u>מבנה ה-G-Machine</u>

- 16 רגיסטרים. גודל כל אחד מהם 32bit, ושמותיהם: $GR0,1,2,...,15$.
- זיכרון שגודלו 2^{32} בתים. בזיכרון יושבת התוכנית בשפת מכונה, וכן הנתונים שהתוכנה צריכה על מנת לעבוד.
- הפעולות שמבוצעות ברמה זו הן פעולות כבדות והאינטרפרטר יהפוך כל פעולה לשרשרת פעולות מיקרו.
- האופרנדים יכולים להגיע מהרגיסטרים או מהזיכרון בניגוד לMaybe שם כל האופרנדים מגיעים מהרגיסטרים.



Control Rom	μRom	Main Memory
<u>מיקרו אינטרפרטר</u> מיקרו פקודות ↓ אותות בקרה (ננו פקודות)	<u>אינטרפרטר</u> פקודות בשפת מכונה ↓ מיקרו פקודות	תוכנית בשפת מכונה + נתונים לתוכנית בשפת מכונה



שיטות/אופני מיעון

נרצה לאפשר דרכים שונות לגשת אל האופרנדים בהם אנו משתמשים.
למשל, נרצה להיות מסוגלים לקבל את תוכן רגיסטר כאופרנד, נרצה להיות מסוגלים להתייחס לכתובת בזיכרון, וכו'.
לפיכך, נגדיר שיטות להעביר אופרנדים לפקודות.

האופרנדים יכולים להיות ברגיסטרים או בזיכרון. גישה לרגיסטרים מצריכה 4 ביטים וגישה לזיכרון מצריכה 32 ביטים (רוחב הזיכרון). יש פערים גדולים בצרכים עבור פקודות שמתייחסות אל הזיכרון לבין פקודות שמתייחסות לרגיסטרים.

דרך אפשרית אחת להתייחס אל האופרנדים באופן שונה הוא להוסיף פקודות שונות למכונה, למשל add memory, שתניח כי האופרנד שהיא מקבל הוא כתובת בזיכרון, add register שתניח כי האופרנד שהיא מקבלת הוא רגיסטר וכו'.
החיסרון של פתרון כזה הוא שנצטרך פקודה נפרדת עבור כל צירוף של אופרנדים.

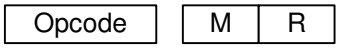
הפתרון שבו נשתמש, הוא הוספת סיביות (M) שתפקידן להגדיר שיטות מיעון (Addressing Modes).

בגישה זו, במקום להעביר לפונקציות את הכתובות של האופרנדים (או את האופרנדים עצמם), בצורה דומה לזו שאנו משתמשים בה בMAYBE, נעביר כתובת כללית (General Address - GA) מהמבנה הבא:

Mode M	Register R	Additional information X (optional)
4 bits	4 bits	(mode dependent)

מבנה זה מאפשר לנו להשתמש לכל היותר ב16 רגיסטרים ולכל היותר ב16 שיטות מיעון.

פקודה תורכב מOpcode באורך שני בתים, ולאחריו האופרנדים השונים, המועברים על ידי GA.

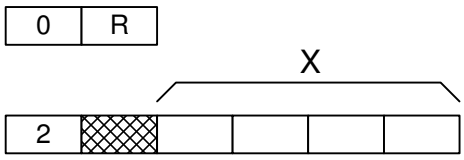


דוגמא

במכונת הG, Mode 0 פירושו שהאופרנד מצוי ברגיסטר, ואילו Mode 2 פירושו כי האופרנד מצוי בזיכרון. נקודד את הGeneral Address כך במקרים הנ"ל:

$M = 0000 \rightarrow$ Operand In Register

$M = 0010 \rightarrow$ Operand In Memory



נניח כי נרצה לבצע את החישוב הבא: $GR4 \leftarrow < GR2 > + < GR3 >$
נשתמש בפקודה gadd4, המבצעת פעולת חיבור על 4 בתים.
בשפת המכונה, תחביר הפקודה יהיה:

gadd4 r(2) r(3) r(4)

r(X) משמעותו שאנו כותבים או קוראים מהרגיסטר X.

הפקודה תקודד בזיכרון באופן הבא:

Byte	Contents	Description
0	00100010	gadd4 opcode
1	00000010	first GA: mode 0 (register), register R2
2	00000011	second GA: mode 0 (register), register R3
3	00000100	third GA: mode 0 (register), register R4

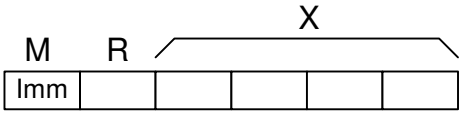
בטבלה הבאה נציג את שיטות מיעון המקובלות במחשבים רבים. לאחר שנסביר שיטות אלו, נציג את קידודן של שיטות המיעון מיעון אלו בG-Machine, ואת השיטות המיוחדות לG-Machine.

Addressing mode	Instruction stream	Assembler syntax	Source datum V	Destination address E
Immediate	X	imm(X)	V=X	
Direct	X	dir(X)	$V=\langle X \rangle$	E=X
Register	R	r(R)	$V=\langle R \rangle$	E=R
Indirect	X	I(X)	$V=\langle\langle X \rangle\rangle$	$E=\langle X \rangle$
Indirect register	R	ir(R)	$V=\langle\langle R \rangle\rangle$	$E=\langle R \rangle$
Indexed	R, X	ix(X, R)	$V=\langle\langle R \rangle+X\rangle$	$E=\langle R \rangle+X$
Stack push		push()		E=SP; $SP \leftarrow \langle SP \rangle+\alpha$
Stack pop		pop()	$SP \leftarrow \langle SP \rangle-\alpha$; $V=\langle\langle SP \rangle\rangle$	
SP-relative	X	ix(X, SP)	$V=\langle\langle SP \rangle+X\rangle$	$E=\langle SP \rangle+X$

הערה

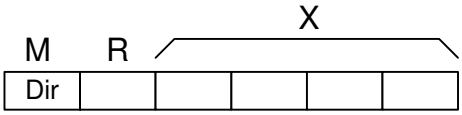
בG-Machine המספרים מאוכסנים בזיכרון בשיטת Little-Endian - הביט הLSB מופיע ראשון והMSB אחרון.
כמו כן, בG-Machine אנחנו לא מדברים על SRAM וDRAM, אלא רק על זיכרון כללי.

Immediate



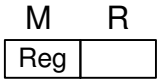
בשיטה זו מתקיים: האופרנד שווה לX.
שיטה זו טובה להגדיר אופרנדים אך לא טובה להגדרת יעד לתוצאת פעולה.
עם זאת, אם ננסה לכתוב ליעד המקודד בשיטת מיעון זו, האינטרפרטר יאפשר לנו לעשות זאת. במקרה כזה המידע ייכתב לתוך X. שיטת פעולה זו איננה מקובלת כיום - בשיטה זו התוכנית משנה את עצמה במהלך הריצה.
מקובל כיום להפריד בין התוכנית לבין הנתונים, ולכן לא נשתמש בשיטה זו בתור יעד לתוצאת פעולה.

Direct



בשיטת מיעון זו, המספר שכתוב בX הוא הEffective Address (EA) של האופרנד - זהו כתובת האופרנד.

Register



בשיטה זו האופרנד הוא רגיסטר. בשדה R נמצא מספר הרגיסטר, ממנו אנו קוראים או כותבים. אין נתונים נוספים (X) בשיטת מיעון זו.

Indirect

בשיטה זו X מכיל את הכתובת של הEffective Address.
מדוע נרצה להשתמש בשיטת מיעון כזו?
שימוש אפשרי הוא כאשר בזמן כתיבת התוכנית לא ניתן לדעת מה תהיה הכתובת של משתנה, ולכן נקצה לכתובת מקום עוד לפני תחילת התוכנית, ונצביע עליו.
שיטת מיעון זו היא הגישה הלא ישירה הפשוטה ביותר הקיימת. שיטות מיעון נוספות, כגון indirect register מכילות בתוכן גם רמה של עקיפות, ומוסיפות עליה דברים נוספים.

Indirect Register

בשיטה זו בתוך הרגיסטר נמצאת הכתובת בזיכרון (EA).

Indexed Addressing

שיטה זו משלבת רמת עקיפות (בעזרת רגיסטר) עם תוספת של קבוע.
הכתובת של האופרנד היא לכן סכום של שני ביטוי: קבוע X השמור עם הפקודה, ומשתנה ברגיסטר R. הכתובת האפקטיבית בשיטה זו היא $E=\langle R\rangle+X$, ואילו התחביר באסמבלי של השיטה הוא ix(X, R).
ישנם שני שימושים עיקריים לשיטה זו:

- פנייה למערכים. במקרה זה אנו מניחים כי המערך מתחיל בכתובת X. ההיסט הרצוי מתחילת המערך מחושב בזמן הריצה ומאוכסן ברגיסטר R. בכל פעם שאנו רוצים לעבור לאיבר הבא, אנו מוסיפים לתוכנו של R את הגודל של איבר בודד, וכך אנו מקבלים בעזרת שיטת מיעון זו את הכתובת של האיבר הבא.

הקוד הבא ממש גישה לאיבר הJ במערך A:

```
gmove4    dir(J)    r(0)      | R0 ← ⟨J⟩
gmult4    imm(4)    r(0) r(0) | R0 ← ⟨R0⟩·4
gmove4    ix(A, 0)  r(1)      | R1 ← ⟨⟨R0⟩+A⟩
```

לא בהכרח נשתמש בשיטת מיעון זו כדי לגשת למערך.
הקוד הבא ממש גישה לאיבר הJ במערך שכתובתו מצויה בR2:

```
gmove4    dir(J)    r(0)      | R0 ← ⟨J⟩
gmult4    imm(4)    r(0) r(0) | R0 ← ⟨R0⟩·4
gadd4     r(2)      r(0) r(0) | R0 ← ⟨R2⟩+⟨R0⟩
gmove4    ir(0)     r(1)      | R1 ← ⟨⟨R0⟩⟩
```

- גישה למבנים (structs). מבנה הוא אלמנט המוכר לנו משפות עיליות. זהו מעין משתנה, אשר מכיל מספר שדות אשר כל אחד מהם הוא משתנה בפני עצמו.
במקרה זה R יכיל את כתובת הבסיס של מבנה כזה, ואילו X יכיל את ההיסט מכתובת הבסיס אל השדה המבוקש.
נביט למשל בביטוי משפת C: p->C, כאשר p הוא מצביע אל מבנה וC הוא שם של אחד מהשדות במבנה. במצב זה נוכל להשתמש בשיטת מיעון Indexed Addressing. בקידוד ix(X, R), X יהיה ההיסט של C מהתחלת המבנה, וR יהיה כתובת הרגיסטר שיכיל את התוכן של p.

הערה

נשים לב כי אם נשתמש בשיטה זו באופן הבא: ix(0, R), אזי שיטה זו זהה לחלוטין לשיטת indirect register. ניתן לנצל עובדה זו על מכונה המממשת את שתי שיטות מיעון אלו.

מחסנית

נוכל להביא אופרנדים מהמחסנים, ולשמור תוצאות במחסנית.
בניגוד למחסנית של הMAYBE, המחסנית של הG-Machine מתחילה מכתובת נמוכה וגדלה אל הכתובות הגבוהות.

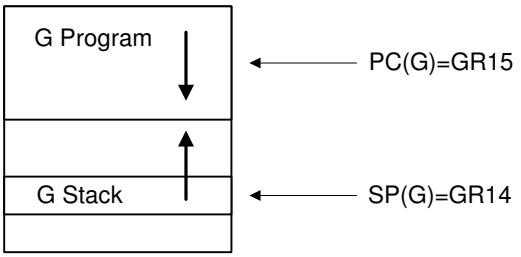
SP-relative addressing

שיטת מיעון זו היא למעשה מקרה מיוחד של indexed addressing, בה הרגיסטר הוא הstack pointer. אנו משתמשים בשיטה זו כדי לגשת לאיבר במחסנית, אך איננו נמצא בראש המחסנית.

G-Machine

עד כה דיברנו על שיטות מיעון כלליות, והדגשנו מעט נקודות הקשורות אליהן ואל הG-Machine. כעת נתרכז במבנה של הG-Machine, ובשיטות המיעון הייחודיות למכונה זו.
G-Machine כוללת מספר תכונות, המאפשרים לה לספק מגוון רחב של מיעונים, בעזרת מספר קטן של מניפולציות:

- בG-Machine הProgram Counter (PC) והStack Pointer (SP) הם רגיסטרים כלליים לכל דבר, ואפשר להשתמש בכל שיטות המיעון עליהם.
בG-Machine, GR14 משמש כSP ואילו GR15 משמש כ-PC. (PC - רגיסטר המכיל את כתובת הפקודה הבאה לביצוע).
- המכונה כוללת שיטות מיעון עקיפות, אשר, כאפקט צדדי, משנות את התוכן של רגיסטר ספציפי. למשל, שיטות המיעון Post Increment ו Pre Decrement. עובדה זו מאפשרת לנו לסרוק מערך איבר איבר, או ליצור בעזרת רגיסטרים מחסניות נוספות בנוסף למחסנית של המכונה.



שיטות המיעון של G-Machine

הטבלה הבאה מציגה את שיטות המיעון הבסיסיות של הG-Machine:

Addressing mode	M field	Effective Address	Assembler syntax
Register	0	$E=Rn$	r(n)
Indirect register	1	$E=\langle Rn\rangle$	ir(n)
Direct	2	$E=X$	dir(X)
Indirect	3	$E=\langle X\rangle$	i(X)
Indexed	4	$E=X+\langle Rn\rangle$	ix(X, n)
Indirect indexed	5	$E=\langle X+\langle Rn\rangle\rangle$	iix(X, n)
Postincrement	6	$E=\langle Rn\rangle; Rn \leftarrow \langle Rn\rangle+\alpha$	posti(n)
Indirect Postincrement	7	$E=\langle\langle Rn\rangle\rangle; Rn \leftarrow \langle Rn\rangle+\alpha$	iposti(n)
Predecrement	8	$Rn \leftarrow \langle Rn\rangle-\alpha;\ E=\langle Rn\rangle$	pred(n)
Indirect Predecrement	9	$Rn \leftarrow \langle Rn\rangle-\alpha;\ E=\langle\langle Rn\rangle\rangle$	ipred(n)

בעזרת שיטות בסיסיות אלו, ובעזרת SP וPC, אנו יוצרים את שיטות המיעון הנוספות.

הטבלה הבאה מציגה את שיטות המיעון הנגזרות משיטות המיעון הבסיסיות:

Addressing mode	M field	General register	Effective Address	Assembler syntax
Immediate	6	PC	$E=\langle PC\rangle; PC \leftarrow \langle PC\rangle+\alpha$	$imm\alpha(X)$
SP-relative	4	SP	$E=X+\langle SP\rangle$	ix(X, SP)
Indirect SP-relative	5	SP	$E=\langle X+\langle SP\rangle\rangle$	iix(X, SP)
Stack (push)	6	SP	$E=\langle SP\rangle; SP \leftarrow \langle SP\rangle+\alpha$	push()
Stack (pop)	8	SP	$SP \leftarrow \langle SP\rangle-\alpha;\ E=\langle SP\rangle$	pop()
PC-relative	4	PC	$E=X+\langle PC\rangle$	rel(X)

כאשר אנו משתמשים בשיטות המיעון הנגזרות משיטות המיעון הבסיסיות, האסמבלר יקודד אותן למעשה לשיטות הבסיסיות, כשרגיסטר הקישור יהיה הPC או הSP.

שיטת המיעון Postincrement

שיטת מיעון זו לא הייתה קיימת בין שיטות המיעון הכלליות שהצגנו. מעצבי ה-G Machine הוסיפו שיטה זו, עבור פישוט מספר פעולות נפוצות, כגון מעבר על מערכים.
בשיטה זו תוכן הרגיסטר הוא הכתובת האפקטיבית של האופרנד.
לאחר חישוב הכתובת האפקטיבית של האופרנד, מקודם הרגיסטר אל כתובתו של האופרנד הבא.
כתובתו של האופרנד הבא נקבעת לפי הפקודה בה אנו משתמשים. אם נשתמש בפקודה gmove1, למשל, עם שיטת מיעון זו, יקודם ערכו של הרגיסטר ב1. אם נשתמש בפקודה gmove4 לעומת זאת, יקודם ערכו של הרגיסטר ב4.
מכאן, האופרנד יקודם ב1, 2, 4 או 8, בהתאם לפקודות בהן נשתמש.

שיטת המיעון Predecrement

שיטה זו היא שיטת המיעון המשלימה לPostincrement.
בשיטה זו, אנו ראשית מקטינים את ערכו של הרגיסטר, ואז מתייחסים אל תוכן הרגיסטר בתור הכתובת האפקטיבית של האופרנד.

הסיבה ששיטות אלו לא הופיעו ברשימת שיטות המיעון הכלליות, היא שבשיטות אלו אנו מניחים כי לשיטת מיעון יכול להיות Side-effect.
כאשר הגדרנו את שיטות המיעון הכלליות, לא הזכרנו את הטענה הזו כלל וכלל, ולכן לא יכולנו לדבר על שיטות אלו.

הערה

חשוב להדגיש כי אין קשר בין הרגיסטרים הכלליים של הG-Machine לבין אלו של הMicrocode. R0 של הG-Machine ממוקם במיקום שונה בSRAM מזה של הMicrocode. גם בין המחסנית של הMicrocode לבין המחסנית של G-Machine אין כל קשר, והן אינן תלויות אחת בשניה.
ארבעה בתים בSRAM ישמשו כרגיסטר אחד של הG-Machine. בנוסף יוגדרו בSRAM כמה רגיסטרים מיוחדים לחישובי ביניים של הG-Machine.

הגדרות המאקרו של שיטות המיעון

נציג כעת את הגדרות המאקרו הממשות את שיטות המיעון שהצגנו.
נדגיש כי הגדרות אלו מיועדת עבור האסמבלי של ה-G-Machine, ובאסמבלי של הMicrocode לא נעשה בהן שימוש.

להלן ההגדרות של המיעונים השונים:

```
| Addressing-mode macros:

.marco      r(R)        R                   | Register
.marco      ir(R)       0x10+R              | indirect register
.marco      dir(X)      0x20 LONG(X)        | Direct
.marco      i(X)        0x30 LONG(X)        | Indirect
.marco      ix(X, R)    0x40+R LONG(X)      | Indexed
.marco      iix(X, R)   0x50+R LONG(X)      | Indirect indexed
.marco      posti(R)    0x60+R              | Postincrement
.marco      iposti(R)   0x70+R              | Indirect Postincrement
.marco      pred(R)     0x80+R              | Predecrement
.marco      ipred(R)    0x90+R              | Indirect predecrement


| Special "derived" addressing modes:

.marco      imm1(X)     posti(PC) X         | Byte immediate
.marco      imm2(X)     posti(PC) WORD(X)   | Word immediate
.marco      imm4(X)     posti(PC) LONG(X)   | Long immediate

.marco      rel(X)      ix(X-.-1, PC)       | PC-relative
.marco      push()      posti(SP)           | Stack push
.macro      pop()       pred(SP)            | Stack pop
```

נדגיש מספר נקודות:

- המאקרו WORD(X) מקבל מספר X ופורש אותו על 2 בתים.
- המאקרו LONG(X) מקבל מספר X ופורש אותו על 4 בתים.
- האופרנד שאנו שולחים לפקודות imm1, imm2, imm4 חייב להיות בגודל המצוין בפקודה. אם לא נספק את האופרנד בגודל הדרוש, האסמבלר לא יעיר כלל, אך יקרא את מספר הבתים שמצוין בפקודה. במצב זה התוכנית שנכתוב לא תעבוד. ניתן עקרונית ליצור אסמבלר שיבדוק האם האופרנדים מתאימים לפקודות בצורה אוטומטית, אך האסמבלר אותו אנו לומדים איננו מכיל מנגנון כזה.
- אם שיטת המיעון שהמשתמש נותן איננה קיימת, dispatch תשלח אותנו אל פונקציה לטיפול בשגיאה בשם GAerr.

פרישת שיטות המיעון בזיכרון הראשי

כל אופרנד מקודד לפחות לבית אחד, המכיל את רגיסטר הקישור ואת שיטת המיעון. בנוסף יכולים להיות, בהתאם לשיטת המיעון, עוד 0, 1, 2 או 4 בתים נוספים לצורך קידוד האופרנד.

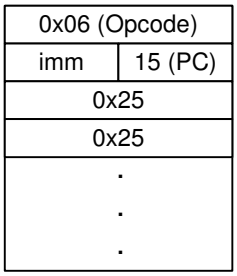
שיטות המיעון	מספר בתים נוספים
posti(n), ir(n), r(n), ipred(n), pred(n), iposti(n), push(), pop()	0
iix(X, n), ix(X, n), i(X), dir(x), rel(X)	4
imm1, imm2, imm4	1, 2, 4

דוגמא

נביט בקידוד הפקודה הבאה בזיכרון:

gmove2 imm2(0x2525), r(4)

Main Memory



המחסנית בG-Machine

השימושים העיקריים של המחסנית בG-Machine הם:

- אכסון משתנים זמניים ותוצאות ביניים של חישובים.
- שמירת כתובות החזרה מפונקציות.
- הקצאת משתנים מקומיים לפונקציות.

אנו ניגשים למחסנית של הG-Machine על ידי שימוש בSP כרגיסטר קישור.

דוגמא קטנה לשימוש במחסנית לצורך אכסון משתנה זמני, היא החלפת תוכנם של שני רגיסטרים. נביט בקוד הבא המחליף בין תוכנם של R0 וR1:

```
gmove4    r(0)    push()    | Push R0 to the stack
gmove4    r(1)    r(0)      | R0 ← ⟨R1⟩
gmove4    pop()   r(1)      | pop into ⟨R1⟩
```

נשים לב, שבדוגמא זו, אחרי פקודת push או pop, ערך הSP משתנה בקפיצות של 4, מכיוון שאנו משתמשים בgmove4, פקודה הפועלת על 4 בתים.

נביט בדוגמא נוספת. הדוגמא מחשבת את הממוצע של R0 וR1, ומחסרת ממוצע זה מתוכן R0 ומתוכן R1.

```
gadd4  r(0)          r(1)          push()  | Stack push ⟨R0⟩+⟨R1⟩
gash4  pop()         imm1(-1)      push()  | Shift right
gsub4  ix(-4, SP)    r(0)          r(0)    | Subtract from R0
gsub4  pop()         r(1)          r(1)    | Subtract from R1 and pop
```

איננו רוצים לשנות ערכים של רגיסטרים אחרים פרט לR0 וR1 במהלך החישוב, ולכן אנו נעזרים במחסנית כדי לבצע אותו. שתי השורות הראשונות מחשבות את הממוצע בעזרת המחסנית. הפקודה **gash4** מבצעת הזזה של ביט אחד ימינה, עקב האופרנד הראשון שלה imm1(-1). אופרנד זה חייב להיות בגודל בית אחד, ולא בגודל של הפקודה, לכן אנו משתמשים בimm1(-1) ולא בimm4(-1).
המשמעות של הביטוי ix(-4, SP) היא גישה אל האיבר העליון במחסנית, ללא הוצאת האיבר מהמחסנית. pop() לאחר מכן ניגש אל האיבר העליון במחסנית, אך כ-Side Effect גם שולף אותו מהמחסנית.

סט הפקודות של הG-Machine

הטבלה הבאה מסכמת את פקודות נבחרות הקיימות בG-Machine.
בטבלה, E_A, E_B, E_C הם הכתובות האפקטיביות של האופרנדים A, B, C.
הסימון α מסמן כי לפקודה יש מספר צורות, על מנת לטפל באופרנדים בגדלים שונים.

Operation	Operand addresses	Function performed
gmoveα	A, B	$E_B^\alpha \leftarrow \langle E_A \rangle^\alpha$
gaddα	A, B, C	$E_C^\alpha \leftarrow \langle E_A \rangle^\alpha + \langle E_B \rangle^\alpha$
gsubα	A, B, C	$E_C^\alpha \leftarrow \langle E_A \rangle^\alpha - \langle E_B \rangle^\alpha$
gmultα	A, B, C	$E_C^\alpha \leftarrow \langle E_A \rangle^\alpha \cdot \langle E_B \rangle^\alpha$
gdivα	A, B, C	$E_C^\alpha \leftarrow \langle E_A \rangle^\alpha \div \langle E_B \rangle^\alpha$
gremα	A, B, C	$E_C^\alpha \leftarrow \langle E_A \rangle^\alpha \text{ mod } \langle E_B \rangle^\alpha$
gnegα	A, B	$E_B^\alpha \leftarrow -\langle E_A \rangle^\alpha$
gandα	A, B, C	$E_C^\alpha \leftarrow \langle E_A \rangle^\alpha \text{ AND } \langle E_B \rangle^\alpha$
gorα	A, B, C	$E_C^\alpha \leftarrow \langle E_A \rangle^\alpha \text{ OR } \langle E_B \rangle^\alpha$
gxorα	A, B, C	$E_C^\alpha \leftarrow \langle E_A \rangle^\alpha \text{ XOR } \langle E_B \rangle^\alpha$
gcomα	A, B	$E_B^\alpha \leftarrow \text{NOT} \langle E_A \rangle^\alpha$
gashα	A, B, C	Arithmetically shift $\langle E_A \rangle^\alpha$ left (right) $\langle E_B \rangle^1$ $\left(\text{or } -\langle E_B \rangle^1\right)$ places. store in $\langle E_C \rangle^\alpha$
glshα	A, B, C	Logically shift $\langle E_A \rangle^\alpha$ left (right) $\langle E_B \rangle^1$ $\left(\text{or } -\langle E_B \rangle^1\right)$ places. store in $\langle E_C \rangle^\alpha$
ghalt		Stop the machine

נשים לב שכאשר נשתמש בפקודות המיועדת לבית או שני בתים, ונשים את התוצאה ב-4 בתים (ברגיסטר למשל), התוצאה תושם בLSB של 4 הבתים, ובבתים הנותרים יתבצע sign extension על מנת לייצג את המספר כראוי בארבעת הבתים.

הטבלה הבאה מסכמת את פקודות הקפיצה ובדיקת האופרנדים האפשריות תחת G-Machine.

Opeation	Operand addresses	Sets CC?	Function performed
gcmpα	A, B	Yes	Set S, Z and C bits in CC according to the result of the operation $\langle E_A \rangle^\alpha - \langle E_B \rangle^\alpha$
gtestα	A	Yes	Set S, Z and C bits in CC according to the result of the operation $\langle E_A \rangle^\alpha - 0$
gjmp	A	No	$PC^4 \leftarrow EA^4$
gj*cond*	A	No	If *cond* matches $\langle CC \rangle$ then $PC^4 \leftarrow EA^4$

בפקודות קפיצה אנו קופצים אל הכתובת האפקטיבית, ולא אל תוכן האופרנד.
למשל, נכתוב "gjmp dir(X)", כאשר נרצה לקפוץ לכתובת X.

לא נוכל להשתמש בכל שיטת מיעון על מנת לבצע קפיצות. אם נשתמש בחלק משיטות המיעון, לא יתקבל ביטוי הגיוני, או שלא יתקבל ביטוי שנוכל להשתמש בו. למשל, נביט בפקודה **gjmp** המשמשת לקפיצה בלתי מותנית.

פקודה	חוקי/לא חוקי	הסבר
gjmp dir(X)	חוקי	קפיצה לכתובת X
gjmp r(2)	לא חוקי	אנו קופצים לכתובת האפקטיבית, ובשיטת מיעון זו אין כתובת אפקטיבית.
gjmp ir(2)	חוקי	נקפוץ אל הכתובת הרשומה בGR2.
gjmp imm4(X)	חוקי, אך בעייתי	המען האפקטיבי הוא התוכן של הPC, והPC באותו רגע בו מפוענחת הפקודה הוא התוכן של הPC, ולכן הוא יוכנס שוב לPC ובמחזור הבא X יבוצע כפקודה.

לG-Machine קיים Condition Register (CC) משלה, השונה מזה של הMAYBE.
בCC זה אנו מנצלים רק שלושה ביטים.
הפקודות השונות בשפת המכונה של G-Machine אינן משנות רגיסטר זה. ישנן מספר פקודות בודדות המשנות את תוכנו, בהתאם לפעולה שהן מבצעים. פקודות אלו הן $\text{gcmp}\alpha$ ו $\text{gtest}\alpha$.
גישה זו, איננה מקובלת כיום במחשבים מודרניים. הגישה המקובלת כיום, היא שכל פקודת ALU תשנה כSide Effect את תוכנו של הCC, וכך נוכל להימנע מפקודות $\text{gcmp}\alpha$ ו $\text{gtest}\alpha$ בתוך התוכנית שלנו.
הקפיצות המותנות מופעלות לאחר השוואה בין שני אופרנדים, שנסמנם a,b.
הCC מכיל שלושה בתים: Z (zero) - מכיל 1 אם $a^\alpha = b^\alpha$. S (signed) מכיל 1 אם מתקיים כי $a^\alpha < b^\alpha$, כאשר אנו מתייחסים לאופרנדים כמספרים בעלי סימן בשיטה המשלים לC .2 מכיל 1 אם החיבור $a^\alpha + \left(-b^\alpha\right)$ גורם לcarry מהMSB. מצב זה מתקיים, אם נתייחס ל a^α, b^α כמספרים ללא סימן, ויתקיים כי $a^\alpha \geq b^\alpha$.

קפיצות מותנות נעשות על ידי המשפחה gj*cond*, כאשר cond הוא אחד מהתנאים המופיעים בטבלה הבאה:

Mnemonic	Branches on	Description
ne	$\overline{Z}$	Not equal (to zero) test
e	Z	Equal
ge	$\overline{S}$	Signed $\geq$
lt	S	Signed $<$
gt	$\overline{Z \ OR \ S}$	Signed $>$
le	$Z \ OR \ S$	Signed $\leq$
hi	C AND $\overline{Z}$	Higher (unsigned $>$)
los	$\overline{C}$ OR Z	Low or same (unsigned $\leq$)
his	C	Higher or same (unsigned $\geq$)
lo	$\overline{C}$	Lowe (unsigned $<$)

דוגמא

נניח שנרצה לממש את הביטוי הבא בשפת C, על גבי G-Machine:

if (a > b) goto here; else goto there;

מימוש אפשרי לשורה זו בG-Machine הוא כלהלן:

```
gcmp4       dir(a)      dir(b)  | set CC from a - b
gjgt        dir(here)           | Go to here if a > b
gjmp        dir(there)          | else go to there
```

דוגמא

נביט בקוד הבא, המחשב את הביטוי A^{16}.

```
        gmove4      imm4(1)     dir(i)              | Initialize loop counter

LOOP:
        gmult4      dir(A)      dir(A)      dir(A)  | Do A = A * A
        gadd4       dir(i)      imm4(1)     dir(i)  | Increment loop counter
        gcmp4       dir(i)      imm4(4)             | Done with loop yet?
        gjle        dir(LOOP)                       | If no, go around again
```

נשים לב שמכיוון שהcounter שלנו נע בין הערכים 1 ל4, אין הבדל בדוגמא אם נשתמש בgjle או בgjlos.

תוכנית זו הייתה יכולה להתרגם לשורה הבאה בC, למשל:

for (i = 1; i <= 4; ++i) A = A * A;

מכיוון שבאסמבלר אין לולאת for, נשתמש בהסתעפויות מותנות על מנת ליצור את אותו האפקט.

הפקודה gtc1 - test and clear

פקודה זו פועלת בדומה לgtest1. היא קובעת את ערכי S, Z, C של הרגיסטר CC בהתאם לתוכן התא שהיא מקבלת. לאחר מכן, פקודה זו מאפסת את התא.
מתי נרצה להשתמש בפקודה כזו? שימוש אפשרי הוא המצב הבא:
נניח שיש לנו התקן, שהביט B מייצג האם ההתקן המבוקש פנוי או עסוק.

	B

כמן כן, נניח שאנחנו עובדים במערכת בה מספר תוכנות יכולות לרוץ בו זמנית.
בשלב כלשהו במהלך ריצת המערכת, שתי תוכנות רוצות להשתמש בהתקן הנ"ל.
התוכנה הראשונה בודקת את הביט, ורואה שההתקן פנוי. באותו רגע, עוד לפני שהתוכנה הראשונה מסמנת בביט זה שההתקן תפוס, התוכנה השניה בודקת אף היא את הביט, ורואה גם שהוא פנוי.
מתעוררת בעיה של התנגשות בין שני תוכנות שבו זמנית ניגשות למידע.
על מנת לפתור זאת, הוספה פעולה אטומית, שבודקת את המידע, ואם ההתקן פנוי תסמן אותו כתפוס מיד אחרי הבדיקה, ואז הבעיה תמנע.

הערה

טבלה A5.1, הנמצאת בעמוד 724 בספר הקורס, מכילה רשימה מלאה של הפקודות של הG-Machine.

מימוש G-Machine על ה-MAYBE

נציג כרגע את עיקרי מימוש G-Machine על מחשב ה-MAYBE.
הקוד הבא מגדיר את המוסכמות כיצד נשתמש ב-SRAM של ה-MAYBE למימוש ה-G-Machine.

```
| 32-bit G-machine archtecture: microcode support

| Machine-level register definitions:
| Each is 4 bytes long.
GR          = 32
GR1         = GR0+4
GR2         = GR1+4
GR3         = GR2+4
GR4         = GR3+4
GR5         = GR4+4
GR6         = GR5+4
GR7         = GR6+4
GR8         = GR7+4
GR9         = GR8+4
GR10        = GR9+4
GR11        = GR10+4
GR12        = GR11+4
GR13        = GR12+4
GR14        = GR13+4
GR15        = GR14+4
SP          = GR14              | G machine's stack pointer
PC          = GR15              | G machine's program counter

| Some SRAM registers reserved for particular purposes:
Op1         = 0x60              | Temporary results and oprands.
Op2         = Op1+4
Op3         = Op2+4
Op4         = Op3+4
Op5         = Op4+4
Op6         = Op5+4

MCond       = Op6+4             | G machine condition codes
MLen        = MCond+1           | Current operand length
MOp         = MCond+2           | Current operand

MReg        = MOp+1             | Register number of current operand
MMode       = MCond+2           | Current addressing mode

mAdrFlag    = MMode+1   | Flags whether the eff adr is a DRAM or SRAM adr.
MEAdr       = MMode+2           | Effective addresses may be 2 bytes wide
MEAdr2      = MMode+3           | (DRAM adrs)

| Fecth a byte from machine's instruction stream into SRAM address x. Increment PC.
.macro fetch(x) l(PC, PC+1, x) cadd2(1, PC, PC)
```

חישוב הכתובת האפקטיבית

על מנת לגשת לאופרנדים השונים של שפת המכונה, נגדיר מספר פוקנציות אשר ישמשו את מפענח הG-Machine:

Meff
פונקציה זו שולפת ומחשבת את הכתובת האפקטיבית של האופרנד הבא המצוי בערוץ הפקודות של הG-Machine (G-machine instruction stream).

LoadGAα,StoreGAα
פונקציות המנתחות שיטות מיעון עבור רגיסטר המקור ועבור רגיסטר היעד, ושולפות או שמות מידע באופרנדים.

להלן הMicrocode המממש את Meff.

```
| Microsubroutines to compute the effective address of an operand
| specified by the next general address in the G instruction stream.
| Performs any register modifications implied by the general address
| encoding (such as postincrement and predecrement).
| Sets MAdrFlag to "SRAM" if the operan is in SRAM (else sets it
| to "DRAM"), where "SRAM" and "DRAM" are arbitrarily
| chosen constants.

SRAM = 0          | There are defined as constats to be used to "flag" whether
DRAM = 1          | the eddective address is an SRAM or DRAM adr.

| MEAdr, MEAdr+1 are set to the effective address (for SRAM adrs, MEAdr+1
| is ignored since there are 1-byte addresses).
| Clobbers Op3, temporary registers R0-R4
| Assumes that MLen is preset to the appropriate operand length.

| Extracts the reister number and mode specified into MReg and MMode
| respectively.

MEff: fetch(MOp)                    | The mode/register byte.
      move(MOp, MReg)               | Copy, for register number.
      cand(0x0F, MReg, MReg)        | Mask off register number bits.
      rotl2(MReg, MReg)             | Multiply by 4
                                    | (4-byte-wide registers)
      cadd(GR0, MReg, MReg)         | Add base address in SRAM where
                                    | the registers begin to get
                                    | the SRAM adr of specified register.
      move(MOp, MMode)              | Copy, for adr mode.
      rotr4(MMode, MMode)           | Extract the mode bits
      cand(0x0F, MMode, MMode)
      dispatch(MMode, AdrTab)
```

```
(continued)


| 16-entry mode dispatch table

AdrTab:
    WORD(hr)        WORD(hir)       WORD(hdir)      WORD(hi)
    WORD(hix)       WORD(hiix)      WORD(hposti)    WORD(hipostin)
    WORD(hpred)     WORD(hipred)    WORD(GAerr)     WORD(GAerr)
    WORD(GAerr      WORD(GAerr)     WORD(GAerr)     WORD(GAerr)
```

Meff נקראת ללא פרמטרים. היא שולפת כתובת כללית מערוץ הפקודות, מפענחת אותו (תוך כדי ביצוע הSide effects אם קיימים), ומשאירה את הכתובת האפקטיבית בשלוש כתובות בSRAM ששמורים למטרה זו.
הכתובת האפקטיבית יכולה להיות מיקום בSRAM (למשל עבור אחד מהרגיסטרים הכלליים) או בDRAM (למשל עבור גישה לזיכרון הראשי של הG-Machine).
הבית MAdrFlag בSRAM מציין לאיזה מהזיכרונות צריך להתייחס.
אם אנו קובעים את MAdrFlag להיות SRAM (קבוע שערכו 0), אזי הכתובת האפקטיבית היא רגיסטר בSRAM. הרגיסטר MEAdr יכיל את הכתובת.
אם MAdrFlag שווה לDRAM (קבוע שערכו 1), אזי הכתובת האפקטיבית היא מיקום בDRAM. הכתובת תישמר ברגיסטרים MEAdr, MEAdr+1.
Meff שולפת את הבית Mode and Register מהDRAM.
לאחר מכן היא מפענחת את השדה R בתור אחד מהרגיסטרים הכלליים GRn, ושפה את התוצאה בתוך המשתנה MReg. לאחר מכאן, היא מפענחת את השדה Mode, ובעזרת dispatch קופצת אל הפונקציה המתאימה שתמצא את הכתובת האפקטיבית לפי שיטת המיעון שהתקבלה. טבלת הפונקציות השונות המשמשת את dispatch היא AdrTab. אם שיטת המיעון בה השתמש המשתמש איננה קיימת, אזי תקרא הפונקציה GAerr. פונקציה זו שמה את Op1 בתור הכתובת האפקטיבית.

נציג את הקוד המשמש לפענוח שיטות המיעון Register ו-Indirect register.

```
| Register mode handler: all operand sizes

hr:
        move(MReg, MEAdr)           | Rn is effective address
        cmove(SRAM, MAdrFlag)       | (Remember that it is in SRAM)
        rtn()

| Indirect register mode handler: Use 2-byte register contents as DRAM adr

hir:
        move(MReg, R0)              | Low 2 bytes on Rn into MEAdr
        imove(R0, MEArr)            | Low adr is <R0>      (=<MReg>)
        cadd(1, R0, R0)             | High adr is <R0>+1   (=<MReg>+1)
        imove(R0, MEArr+1)
        cmove(DRAM, MAdrFlag)       | Eff. adr is in DRAM, adr <MEAdr>
        rtn()
```

בשיטת המיעון register הכתובת האפקטיבית היא בSRAM, ואנו פשוט מעתיקים את MReg אל MEAdr.
בכל שאר שיטות המיעון, הכתובת האפקטיבית היא בDRAM, ואז אנו מרימים את הדגל DAdrFlag.

נציג גם את המימוש של שיטת המיעון indexed:

```
| Indexed mode handler

hix:
        move(MReg, R1)              | Fetch low 2 bytes of register
        imove(R1, R3)               |       into R3, R4
        cadd(1, R1, R1)
        imove(R1, R4)

        FETCH4(Op3)                 | Fetch 4-byte offset
        add2(R3, Op3, MEAdr)        | Add in register contents
        cmove(DRAM, MAdrFlag)       | Eff. adr is in DRAM, adr <MEAdr>
        rtn()
```

אנו משתמשים בOp3 כדי לשמור באופן זמני את ההיסט X של שיטת מיעון זו, שאותו אנו קוראים מערוץ הפקודות.
למרות שאנו קוראים 4 בתים (הגודל של X בזיכרון), אנו משתמשים רק ב2 מהם לכתובת. זוהי דוגמא למקרה בו למעשה הMAYBE מגביל אותנו מעט. אם היינו עובדים מול מכונה עשירה יותר במשאבים, הדרך בה האסמבלר תוכנן הייתה מאפשרת לנו להשתמש בכל ארבעת הבתים עבור הכתובת, ולספק מרחב זיכרון רחב יותר למשתמש.

הקוד הבא הוא הממש את שיטת המיעון Indirect indexed:

```
| Indirect indexed mode handler

hiix:
        move(MReg, R1)                  | Fetch low 2 bytes of register
        imove(R1, R3)                   |       into R3, R4
        cadd(1, R1, R1)
        imove(R1, R4)

        FETCH4(Op3)                     | Fetch 4-byte offset
        add2(R3, Op3, MEAdr)            | Add in register contents

| Following handler executes one level of
| indirection on MEAdr, MEAdr+1, i.e., implements
| MEAdr, MEadr+1 ← ⟨DRAM[⟨MEAdr⟩,⟨MEAdr+1⟩]⟩
hind:
        l(MEAdr, MEAdr+1, R0)           | The indirection: first adr byte
        cadd2(1, MEAdr, MEAdr)
        l(MEAdr, MEAdr+1, MEAdr+1)      | second adr byte
        move(R0, MEAdr)
        cmove(DRAM, MAdrFlag)           | Eff. adr is in DRAM, adr <MEAdr>
        rtn()
```

hiix מבצעת חישוב זהה לזה של שיטת המיעון indexed, ולאחריה hind מוסיפה את רמת העקיפות של שיטת המיעון Indirect indexed.

מימוש שיטות המיעון postincrement וpredecrement מתבסס על המיקרו שגרות increg וdecreg, אשר מגדילות או מקטינות, בהתאמה, את הרגיסטר שמוגדר על ידי בית הMode+Register, בגודל של MLen.
הקוד עבור שיטת המיעון postincrement:

```
| Indirect postincrement handler

hiposti:    move(MReg, R0)          | Fetch low 2 bytes of register
            imove(R0, MEAdr)        |       into MEAdr
            cadd(1, R0, R0)
            imove(R0, MEAdr+1)

            call(increg)            | Increment the register
            jmp(hind)               | Indirect though MEAdr
```

גישה אל האופרנדים

לאחר שחישבנו את הכתובת האפקטיבית של האופרנד על ידי Meff, המפענחים (handlers) של הפקודות של הG-Machine יצטרכו לשלוף או לשמור מידע בכתובות של האופרנדים. הפקודות LEffα, SEffα מספקות לנו את המנגנון לגישות אלו.

LoadGAα, StoreGAα

אנחנו מספקים פקודות הבנויות מעל LEffα, SEffα, אשר משלבות את השגת הכתובת האפקטיבית של האופרנד בעזרת Meff, עם שמירת או טעינת הערכים לזיכרון. פקודות אלו הם LoadGAα, StoreGAα.
כדי לתמוך בפעולות עם מספר אופרנדי מקור, LoadGAα שומרת את Op1 הקודם בOp2, לפני שהיא שומרת בOp1 את האופרנד החדש. לפיכך, שתי קריאות עוקבות ל LoadGAα משאירות את האופרנד הראשון בOp2 ואת האופרנד השני בOp1.
LoadGAα מאריך כל אחד מהאופרנדים לארבעה בתים (sign extension).
חוץ מאשר במקרים מיוחדים, כמו בפקודות קפיצה, LoadGAα ו StoreGAα מספקים את הממשק הסטנדרטי בין שיטות המיעון לMicrocode.

רוב הפונקציות קוראות פעמיים לפונקציה LoadGAα, על מנת לשלוף את האופרנדטים, וכאמור בסוף הקריאה האופרנד הראשון יישב ב-Op2 והשני יישב ב-Op1.
משפחת הפקודות hglshα הן יוצאות דופן מבחינת זו, בכל שבהן האופרנד הראשון נשמר ב-Op1 והשני ב-Op2.

תרגיל

במכונת G, היכן ימצא האופרנד הראשון של הפקודה glsh1 r(2) r(3) r(4)?
א. האופרנד ימצא ב-Op1 לפני הבאת האופרנד השני וב-Op2 אחרי.
ב. האופרנד ימצא ב-Op2 לפני הבאת האופרנד השני וב-Op2 אחרי.
ג. האופרנד ימצא ב-Op2 לפני הבאת האופרנד השני וב-Op1 אחרי.
ד. האופרנד ימצא ב-Op1 לפני הבאת האופרנד השני וב-Op1 אחרי.

תשובה

בהסמך על המידע הידוע לנו לגבי hglshα, ניתן להבין כי האופרנד ימצא ב-Op1 לפני הבאת האופרנד השני וב-Op1 אחרי ההבאה, ולכן התשובה היא ד'.

נביט כעת בסיכום הפקודות השונות בהן מימוש G-Machine על מחשב ה-Maybe משתמש על מנת לפענח את האופרנדים.
נראה את האלגורתים של הפקודות השונות, וכן את הפעולות השונות המתרחשות.

LoadGAα

הפקודה טוענת אופרנד מהכתובת האפקטיבית אל Op1 או Op2.

1. מעתיקה את Op1 לOp2.
2. מכינה פרמטרים לפקודה Meff : מעבירה לרגיסטר Mlen את הגודל α.
3. קוראת לפקודה Meff - לחישוב הכתובת האפקטיבית. (הפקודה מקדמת את הPC).
4. מכינה פרמטרים לפקודה Leffα : מעבירה לרגיסטר R0 את הכתובת Op1.
5. קוראת לפקודת Leffα - המביאה אופרנד מן הכתובת האפקטיבית לOp1.

Meff

הפקודה מפענחת את הכתובת האפקטיבית של האופרנד הבא בזיכרון הראשי.

1. על ידי fetch, תא הMR מעוברת אל Mop והPC מקודם.
2. מעבירה לרגיסטר MREG את המיקום האפקטיבי של הרגיסטר, על פי הנוסחה : $MREG \leftarrow 0x20+4\cdot R$, כאשר R הוא מספר הרגיסטר הכללי.
3. מעבירה לMMode את שיטת קוד המיעון M.
4. MAdtFlag - דגל המציין האם הכתובת בSRAM או בDRAM.

Leffα

לאחר שהכתובת האפקטיבית חושבה על ידי Meff והדגלים האחרים חושבו על ידי הhandlers של שיטות המיעון, נקראת פקודה זו כדי לטעון את האופרנד עצמו.

1. בודקת את MAdrFlag כדי למצוא את מיקום האופרנד.
2. אם מיקום האופרנד בDRAM - נשתמש בפקודה getα כדי לקרוא אותו.
3. אם האופרנד בSRAM - נשתמש בפקודה imovei4.
4. בסופו של דבר, האופרנד ימצא הOp1.

StoreGAα

השיגרה מאכסנת את תוצאת החישוב במקום המתאים.

1. מכינה פרמטרים לפקודה Meff וקוראת לה. (PC מקודם).
2. מכינה פרמטרים לפקודה Seffα וקוראת לה.

Seffα

1. בודקת שאין זבל.
2. בודקת את MAdrFlag כדי למצוא את מיקום האופרנד
3. אם מיקום האופרנד בDRAM - נשתמש בפקודה puta כדי לשמור אותו.
4. אם מיקום האופרנד בSRAM - נשתמש בפקודה imovei4 כדי לשמור אותו.

getα

מעתיקה α בתים מכתובת $(<R0>,<R1>)$ בDRAM, לתוך כתובת בSRAM שמתחילה ב$<R2>$, ומבצעת הרחבת סימן ל4 בתים. הפקודה מגדילה את R2 ב4 ואת $(<R0>,<R1>)$ כמספר בן שני בתים ב4.

putα

מעתיקה α בתים מהSRAM מהכתובת שמתחילה ב$<R2>$ לתוך הכתובת $(<R0>,<R1>)$ בDRAM. הפקודה איננה מבצעת הרחבת סימן.
הפקודה מגדילה את R2 ב α ואת $(<R0>,<R1>)$ כמספר בן שני בתים ב α.

נראה כעת את הMicrocode האחראי לפענח ולבצע את הפקודות השונות שמוגדרות עבור הG-Machine.

נתחיל מדוגמא, להלן הקוד המממש את הפקודה gadd4, המקבלת שלושה אופרנדים, שני אופרנדי מקור ואופרנד תוצאה אחד.

```
| gadd4(x, y, z): three-address add, 4 bytes wide
| z ← ⟨x⟩+⟨y⟩

hgadd4:
        call(LoadGA4)                   | Fetch 1st operand
        call(LoadGA4)                   | Fetch 2nd operand
        add2(Op1, Op2, Op1)             | Add all 4 bytes
        addcy(Op1+2, Op2+2, Op1+2)
        addcy(Op1+3, Op2+3, Op1+3)
        call(StoreGA4)
        rtn
```

פקודות אחרות המקבלות שלושה אופרנדים, ממומשות בצורה דומה. נעשה בכולם שימוש בLoadGA על מנת לקרוא את האופרנדים, ובStoreGA על מנת לשמור את התוצאה.

מלבד פקודות בנות 3 אופרנדים, נוכל לממש גם פקודות המקבלות שני אופרנדים. נוכל למשל להגדיר פקודה g2addα שתקבל שני אופרנדים לחבר ביניהם, ותשים את התוצאה בתוך אחד מהם.

נביט במימוש לפקודה זו:

```
| g2add4(x, y): two-address add, 4 bytes wide
| y ← ⟨x⟩+⟨y⟩
hgadd4:
        call(LoadGA4)                   | Fetch 1st operand
        call(LoadGA4)                   | Fetch 2nd operand
        add2(Op1, Op2, Op1)             | Add all 4 bytes
        addcy(Op1+2, Op2+2, Op1+2)
        addcy(Op1+3, Op2+3, Op1+3)
        call(SEff)                      | Store, reusing second eff. address
        rtn
```

במקום להשתמש בStoreGA, השתמשנו בSEff כדי לשמור את התוצאה בכתובת האפקטיבית האחרונה איתה עבדנו, שהיא הכתובת של y.

כעת נראה מימוש של פקודת ההסתעפות gjmp.
נשתמש במימושה בMeff במקום בLoadGA, עקב העובדה שהכתובת האפקטיבית היא יעד הקפיצה, ולא האופרנד עצמו.
פקודות הסתעפות מותנית נכתבות באופן דומה לgjmp, אך נזכור, שכאשר אנו בודקים את הדגלים השונים כדי להחליט האם לקפוץ או לא, אנו בודקים את הCC של הG-Machine, ולא את זה של הMAYBE.

```
| gjmp x -- unconditional jump to target address x

hgjmp:
        cmove(4, MLen)          | 4-byte jump address
        call(Meff)              | Compute effective address
        ccmp(DRAM, MAdrGFlag)   | Had better be a DRAM location...
        jne(adrerr)             | It is
jmpx:
        move(MEAdr, PC)         | Copy effective address into PC
        rtn()
        ...
        ...
| ERROR: illegal address in call, jump or handler declaration:
adrerr: ...
```

הלולאה הנצחית

הלולאה הנצחית היא למעשה התוכנית הראשית שהמיקרו קוד מבצע.

```
1.  Minit:
2.          cmove4(0,PC)
3.  Go:
4.          refr() refr() refr()
5.          refr() refr() refr()
6.          refr() refr() refr()

7.          call(NextInst)
8.          jmp(Go)

9.  NextInst:
10.         l(PC, PC+1, R0)
11.         cadd2(1, PC, PC)
12.         dispatch(R0, ITab)
```

ה–PC הוא בעצם GR15. כמו שהגדרנו בתחילת הדיון על G-Machine,
"GR15 = PC", כאשר לפני זה GR15 מוגדר בתור 0x5C.

לאחר הפעלת שורה 2 בקוד הנ"ל, SRAM נראה משהו בסגנון השרטוט הבא:

0x5B	משהו
PC = 0x5C	0x00
PC+1 = 0x5D	0x00
PC+2 = 0x5E	0x00
PC+3 = 0x5F	0x00
0x60	משהו
0x61	משהו

כאשר 0x5C-0x5F הם ה- PC שלנו, לפי שיטת ה – Little Endian (קרי, ה – LSB בכתובת הנמוכה וה – MSB בכתובת הגבוהה).

נביט כעת בלולאה:
בשורה 2 אנו דואגים לאיפוס 4 הכתובות 0x5C-0x5F ב – SRAM בעזרת cmove4.
בשורות 4-6 אנו דואגים לרענן 108 שורות DRAM, כי ככל הנראה פקודת ה – G תהיה ארוכה. אם הפקודה ארוכה מהמצופה, יש לדאוג לשרבב כמה פקודות refr() גם בתוך הקוד (לדוגמא, בפקודות כפל וחילוק).
שורה 8 דואגת לחזור לראש הלולאה, ומפה חוזר חלילה.

ועכשיו לבשר – שורה 7. בשורה זו קוראים למיקרו-שגרה המתחילה בשורה 9.
בשורה 10 מוציאים בית (Byte) אחד מתוך ה – Instruction Stream, כלומר מוציאים בית מתוך הקוד של מכונת ה – G שכתבנו ב – DRAM וכותבים אותו לתוך R0.
מה שחשוב להבין זה למה l(PC, PC+1, R0) מוציא את הבית מתוך המקום אליו מצביע ה – PC:
<PC> ו – <PC+1> הם למעשה שני הבתים התחתונים של תוכן רגיסטר GR15.
שורה 10 שולפת בית אחד מתוך כתובת (<PC>, <PC+1>) ב – DRAM, וזה למעשה הבית הראשון בפקודת ה – G. מדוע משתמשים רק בשני הבתים? בגלל המימוש ב – MAYBE. כידוע, ה – DRAM של ה – MAYBE הוא DRAM בעל כתובת של 2 בתים, ולא של 4 בתים, כפי שזה בזיכרון מכונת ה - G. במחשבים ממשיים קיים מנגנון הדואג לטעון לזיכרון הפיזי (מכונת ה – MAYBE) את קטעי הזיכרון של המכונה בשפת מכונה (מכונת ה – G). במכונה שלנו אין מנגנון כזה, ולכן אפשר לנצל רק את 2^{16} הבתים הראשונים של מכונת ה – G (מען של 2 בתים), וכל תוכנם נטען לתוך ה – DRAM.
בשורה 11 מקדמים את (PC, PC+1) ב – 1, כך שה – PC יצביע כעת לבית הבא בקוד מכונת ה – G שלנו (אם יש לפקודת ה - G אופרנדים, (PC, PC+1) יצביע אליהם).
בשורה 12 שולחים את הפקודה להתחיל לעבוד, בצורה שפירטנו בעמודים הקודמים.

ארכיטקטורות General Registers נוספות

G-Machine זוהי ארכיטקטורה מסודרת בצורה יוצאת מהכלל, ביחס לרוב ארכיטקטורות הGeneral Registers הקיימות במציאות.
רוב ההבדלים מהמבנה הפשוט של הG-Machine, נובעים מצרכים ספציפיים, ומשפרים את יעילותן של פקודות מסוימות.

ארכיטקטורות General Registers אחרות יכולות להיות שונות במספר נושאים מה-G Machine:

- Short-form address modes - ארכיטקטורות שונות מאפשרות שיטות מיעון ביניים, מבחינת מספר התאים הדרושים לקודדם. בG-Machine ישנם שני קידודים אפשריים - קידוד קצר ביותר, במקרה שהאופרנד הוא רגיסטר, או קידוד שלוקח ארבעה בתים, בשיטות מיעון אחרות. שיפור אפשרי הוא למשל לפקודה ix(X, R). בהרבה מקרים, X הוא מספר קטן יחסית, ולכן מוגדרת שיטת מיעון דומה נוספת, בה X הוא בגודל בית אחד בלבד. דוגמא נוספת היא הפקודה imm4(X) שבדרך כלל מכילה מספרים קטנים בלבד, ונרצה לא לשמור תאים מבוזבזים. כמו כן, בG-Machine גודל האופרנד צריך להתאים לפקודה. שיפור אפשרי הוא להסיר מגבלה זו.
- Short-form instructions - פעולות שונות נפוצות מעוד בתוכניות. שיפור אפשרי לG-Machine הוא הוספת פקודות נפוצות אל שפת המכונה, על מנת שיהיה ניתן לבצע אותן ביעילות ובמהירות. דוגמא למשל היא איפוס תוכן רגיסטר, או הגדלת ערך רגיסטר ב1.
שיפור נוסף הנכנס לאותה קטגוריה הוא פקודות ההסתעפות. פקודות הסתעפות רבות קופצות למרחקים קצרים בלבד. נוכל להוסיף פקודות קפיצה חדשות, שיקפצו למשל offset מהPC הנוכחי, ויתפסו מעט מקום
- הפחתת הכלליות - G-Machine היא מכונה מאוד כללית, במובן שניתן להשתמש בכל שיטת מיעון בכל פקודה. אם נרצה לגרום לפענוח מהיר יותר של פקודות מסוימות, נוכל להחליט ששיטת המיעון עבורן תהיה קבועה, וכך לחסוך את התהליך של פענוח הכתובת האפקטיבית.

שילוב של אופטימיזציות אלו אל תוך ארכיטקטורות General Registers מאפשר להגיע לביצועים גבוהים יותר, אולם אנו משלמים בהוספת סיבוך לארכיטקטורה.
בפועל, השיפור בביצועים כתוצאה מאופטימיזציות אלו משתלם, ואופטימיזציות אלו מצויות בארכיטקטורות שונות בעולם.

פונקציות בG-Machine

הפונקציות gcall וgrtn, מספקות את המכניזם הנדרש על מנת לנהל פונקציות ב-G Machine.
הפקודה gcall דוחפת את תוכן הProgram Counter (המכיל את כתובת הפקודה שאחרי gcall) אל המחסנית, ואז קופצת לכתובת המוגדרת.
השליטה מוחזרת לתוכנית הקוראת על ידי הפקודה grtn.

Operation	Operand addresses	Function performed
gcall	A	$push4[\langle PC \rangle]$ $PC^4 \leftarrow E_A$
grtn		$PC \leftarrow pop4[\ ]$

העברת פרמטרים לפונקציות

G-Machine מאפשרת לנו להעביר פרמטרים לפונקציות במספר דרכים, בדומה לדרכים שהMicrocode נתן לנו.

- נוכל להעביר פרמטרים דרך ערוץ הפקודות. הפרמטרים יוכלו להופיע מיד לאחר הפקודה.
- נוכל להעביר פרמטרים דרך אחד מ16 הרגיסטרים הכלליים שה-G Machineמספק לנו.
- נוכל גם כן להעביר פרמטרים דרך המחסנית. לפני הקריאה לפונקציה, נדרוש את הפרמטרים הרצויים למחסנית, והפונקציה תקרא אותם משם.
אם נשמור פרמטר במחסנית, נגיע אליו בעזרת שיטת המיעון ix(-n, SP), כאשר n זהו מספר הבתים מראש המחסנית, בו מצוי האופרנד.

מלכודות

מלכודות הן סוג של שגרות מיוחדות.
כאשר אנו קופצים למלכודת, בעזרה gsvc, אנו שומרים במחסנית את הPC, הSP והPSW, שזהו רגיסטר המכיל את סיביות הCC, ומחליפים אותם בערכים חדשים, שנמצאים במקום בזיכרון שמיוחד למלכות.
כאשר אנחנו יוצאים מהפסיקה, בעזרת gtrtn, אנו שולפים מהמחסנית את הערכים ששמרנו, ומחזירים אותם לרגיסטרים.

דוגמא

נתונה שגרה בשפה עילית לחישוב N! :

```
int fact(int N)
{
        int former, Res;
        if (N <= 0)
        {
                Res = 1;
        }
        else
        {
                former = fact(N-1);
                Res = former * N;
        }
        return Res;
}
```

התוכנית הקוראת לשגרה מאחסנת את הערך N במחסנית לפני הקריאה, והשגרה תאכסן באותו מקום את תוצאת החישוב. לשגרה שני משתנים מקומיים : former לאחסון (K-1)!, כאשר K משתנה בכל קריאה של השגרה לעצמה.

נרצה לכתוב תוכנית בשפת G-Machine שתממש את השגרה הנ"ל כפי שהיא, כלומר תוך כדי שמירה על האלגוריתם (כל קטע בשפת G יממש שורה מסוימת בשגרה). נדרוש שתוכנית כגון התוכנית הבאה, תוכל לרוץ בעזרת הפונקציה שנממש :

```
gmove4      imm4(10)    push()      | push N = 10
gcall       dir(fact)               | fact(10)
gmove4      pop()       r(0)        | push 10! into GR0
```

```
| offsets from base of frame for 'fact':

N       =       -12     | N's offset
F       =       0       | former offset
R       =       4       | Res offset
B       =       13      | Register used as base of frame pointer

fact:
        gmove4          r(B)    push()                  | Save old GR13 on the stack
        gmove4          r(SP)   r(B)                    | Use as base of frame pointer
        gadd4           imm4(8)         r(SP)   r(SP)   | Allocate space for former, Res
        gtest4          ix(N, B)                        | Recursion condition
        gjle            dir(base)
        gsub4           ix(N, B)        imm4(1)         push()    | Prepare for the next
call
        gcall           dir(fact)                       | Recursion
        gmove4          pop()           ix(F, B)        | Assign recursive result in former
        gmult4          ix(F, B)        ix(N, B)        ix(R, B)  | Res = former * N
        gjmp            dir(endfact)
base:
        gmove4                  imm4(1)         ix(R, B)
endfact:
        gmove4                  ix(R, B)        ix(N,B) | Move Res to desired place for
                                                        | caller
        gsub4           r(SP)   imm4(8)         r(SP)   | Deallocate local variables
        gmove4                  pop()           r(B)    | Restore GR13
        grtn                                            | Exit
```

דוגמא

תרגיל זה מתייחס למכונת G, הממומשת על בסיס MAYBE.

עבור הפקודה הבאה:

```
gsub4  pop()   imm4(2)   ix(0x4321, 0)
```

א. כיצד הפקודה תקודד בזיכרון?

ב. כמה גישות לזיכרון הראשי דרושות לביצוע הפקודה (כולל קריאת הפקודה עצמה)?

פתרון

א.

gsub			= 0x26
pop()	=	pred(SP)	= 0x80+0x0E = 0x8E
imm4(2)	=	posti(PC) LONG(2)	= 0x6F 0x02 0x00 0x00 0x00
ix(0x4321,0)	=	0x40+0x00 LONG(0x4321)	= 0x50 0x21 0x43 0x00 0x00

הפקודה תקודד בזיכרון כך:

Address	Contents
α	0x26
$\alpha+1$	0x8E
$\alpha+2$	0x6F
$\alpha+3$	0x02
$\alpha+4$	0x00
$\alpha+5$	0x00
$\alpha+6$	0x00
$\alpha+7$	0x40
$\alpha+8$	0x21
$\alpha+9$	0x43
$\alpha+10$	0x00
$\alpha+11$	0x00

ב.

מתרחשות 20 גישות לזיכרון הראשי:

אלמנט	מספר גישות	הסבר
Opcode	1	
pop	5	1 לקריאת MR + 4 לקריאת הערך.
imm4	5	1 לקריאת MR + 4 לקריאת הערך.
ix	9	1 לקריאת MR + 4 לקריאת X + 4 לקריאת הערך.

מחשבים סדרתיים

דיברנו על מספר רמות הפשטה - ננו קוד, אשר ברמה זו דיברנו על חומרה, על הננו קוד המשמש על מנת להפעיל את החומרה, בעזרת הבקר. הזכרנו את רמת ההפשטה של המשטר הדינמי, וטענו שלאחר מעבר השעון, ישתנו ערכי רכיבי המערכת.
הזכרנו כמו כן גם רמת הפשטה נוספת - שפת המכונה והדגמנו את G-Machine בשפת מכונה. הG-Machine היא שפת מכונה. נתעסק כעת בשפת מכונה כללית, ונפרט מה צריכים להיות הדרישות ממנה, ולמה.

שפת מכונה היא סוג של הפשטה. אולם, יש הבדל בינה לבין רמות ההפשטה האחרות עימן התעסקנו. כל פעם שהצגנו רמה חדשה, היא ישבה מעל הרמות הקודמות שהכרנו, ואז התעסקנו ישירות איתה. האסמבלי, לעומת זאת, היא רמה שתשמש כיעד לפעולות של קומפיילרים. מעל האסמבלי תהיה רמת הפשטה נוספת, שהיא השפה העילית, ויהיה קומפיילר, אשר יתרגם את השפה העילית לאסמבלי.
לכן, הדגש העיקרי בשפת האסמבלי לא יהיה הביצועים שנוכל להשיג על ידי תכנות ישירות בשפת האסמבלי, אלא הביצועים שנוכל להשיג כאשר קומפיילר של שפה גבוהה יתרגם אותה לאסמבלי. שפת המכונה צריכה להיות כזו, שיהיה קל לתרגם שפה עילית עליה, מבחינת כמות הפעולות שידרשו כדי לממש את האלמנטים השונים של השפות העיליות.
למשל, כאשר דיברנו על הG-Machine, הזכרנו את שיטות המיעון השונות המשמשות לגישה לאופרנדים. שיטות מיעון כמו index מאפשרת למכונה לתמוך במבני נתונים שונים, כדוגמת רשימה מקושרת, מערך ועוד.

מטרה נוספת לאסמבלר היא לספק ממשק אל פונקציות של מערכת ההפעלה, כגון הגנה על מידע, או ניהול זיכרון ומשאבים אחרים במחשב.
כאשר סקרנו את הG-Machine, ראינו למשל פקודה העוזרת בניהול משאבים, (test and clear).

האסמבלר הוא למעשה הגשר בין רמת ההפשטה של החומרה לבין זו של התוכנה.
נניח לצורך פשטות שהמכונה שלנו היא מכונת Single Sequence Machine, בה פקודות מתבצעות אחת אחרי השניה, ולא במקביל.

<u>מבנה המכונה</u>

המכונה איתה נעסוק, מורכבת משני אלמנטים מרכזיים: מעבד וזיכרון.
המעבד הוא מכונת מצבים, אשר ניתן לשמור בה מידע ברגיסטרים המצויים עליה.
הזיכרון מכיל רצף של כתובות, שניתן להתייחס אליהם בתור רגיסטרים שכל אחד מהם מכיל כמות מסוימת של ביטים. מספר המילים ב"רגיסטר" כזה נקרא רוחב המילה של המחשב. מספר זה הוא פרמטר חשוב בארכיטקטורה של המחשב. לכל כתובת בזיכרון יש מספר מזהה הייחודי לה הנקרא כתובת. מספר הביטים המשמשים לקידוד הכתובת הוא פרמטר חשוב נוסף בארכיטקטורה. מספר זה קובע לכמה כתובות זיכרון נוכל לגשת. למשל, נניח שבמחשב כלשהו אורך כתובת הוא 16 בתים, אזי נוכל לגשת ל2^{16} כתובות. ניתן גם לכתוב וגם לקרוא מהזיכרון, אם כי מחשבים מסוימים מספקים בנוסף לזיכרון לכתיבה וקריאה, גם זיכרון לקריאה בלבד.
תוכנות הן רצף של פקודות הנשמרות בזיכרון. המעבד קורא את השורות אחת אחרי השניה ומבצע אותן אחת אחת. את הכתובת הבאה קובע רגיסטר הPC, המכיל את הכתובת הבאה לביצוע.
<u>גורמים המשפיעים על תכנון שפת המכונה</u>

נציג כעת גורמים שונים, הנלקחים בחשבון ומשפיעים על תכנון שפת המכונה.
נשים לב כי הגורם המשפיע ביותר בתכנון שפת המכונה, הוא הצרכים השונים של השפות העיליות.

סוגי נתונים

כל השפות העיליות מסוגלות להתעסק עם סוגים שונים של מידע.
סוגי מידע שונים יכולים להיות, למשל:

- מספרים: שלמים, נקודה צפה וכו'.
- תווים:
- מצביעים: משתנה שמכיל את כתובתו של משתנה אחר

כפי שאנו יודעים, הזיכרון מכיל רק אפסים ואחדות. לא ניתן לדעת מהתבוננות בזיכרון, איזה סוג מידע נמצא מולנו. אולם, בשפות עיליות קיימים סוגי הנתונים שהצגנו ונוספים, ולפיכך אחד מתפקידי הקומפיילר יהיה לתרגם את התוכנית ואת סוגי הנתונים לתוכנית שתרוץ בשפת המכונה. שפת המכונה מתעסקת עם מספרים בינריים בלבד, אולם נוכל לספק עזרה מסוימת לקומפיילר. נוכל להוסיף פקודות למניפולציות על גדלים שונים של בלוקים (בית, שני בתים, ארבעה בתים וכו'), ובכך לפשט לקומפיילר את תהליך תרגום השפה העילית לשפת מכונה.
כמו כן, נוכל להוסיף פקודות כגון "חבר מספרים שלמים" או "חבר מספרים ממשיים".
גישה נוספת שניתן למצוא בארכיטקטורות שונות היא הוספת tags אל המידע. tag אלו שדות נוספים לכל אלמנט מידע בזיכרון – תווית האומרת מהו תוכן התא - תו, מספר שלם, מצביע וכו'. בשיטה כזו אנו חוסכים פקודות - המכונה מזהה לבד את הנתונים ויכולה לפעול בהתאם לפעולות שמוגדרות על כל אחד מהנתונים. החסרון: לא נוכל בתוכנה להגדיר סוגי נתונים חדשים. אם נרצה להוסיף סוגי מידע חדשים נצטרך לעדכן את החומרה, את המיקרו קוד ואת שפת המכונה.

מבני נתונים:

בעזרת סוגי הנתונים הבסיסיים, נוכל ליצור מבני נתונים שונים.
מספר דוגמאות למבנים נתונים:

- מערכים: מערך זהו אוסף של אלמנטים מסוג בודד. לכל איבר במערך יש אינדקס אשר מזהה אותו באופן יחיד. למשל, נניח שיש לנו מערך A, אזי A[3] יציין את אחד מאברי המערך, ואילו A[5] יציין איבר אחר.
- מבנים: מבנה הוא אוסף של מספר משתנים בסיסיים מסוגים שונים, המקובצים תחת שם אחד. למשל, נוכל להגדיר מבנה בשם בן אדם, אשר השדות שלו יהיו משתנה מסוג מספר שלם, שייצג גיל, ושדה מסוג מחרוזת, שייצג את שם.
- רשימות מקושרות: רשימה מקושרת היא אוסף של אלמנטים, בדומה למערך, בהבדל שהאלמנטים השונים לא נמצאים רציפים בזיכרון, אלא כל איבר הוא מבנה, אשר אחד משדותיו הוא מצביע אל האיבר הבא ברשימה המקושרת. במקרה זה, הוספת שיטות המיעון העקיפות לאסמבלר יקלו מאוד את מימוש הרשימה המקושרת.

פונקציות

אחת מהאפשרויות הנפוצות ביותר בשפות התכנות היא שימוש בפונקציות. נרצה ששפת המכונה תספק תמיכה בפונקציות.
נצטרך לספק מנגנון נוח לשמירת כתובת החזרה, להעברת פרמטרים לפונקציות, ולשמירת משתנים לוקליים של פונקציות.

מחסנית

המחסנית היא חלק חשוב ממימוש שפות התכנות העיליות. המימוש המקובל של מחסנית משתמש באיזור ייעודי בזיכרון על מנת לשמור את תוכן המחסנית, וברגיסטר בשם Stack Pointer, שתפקידו לשמור את מספר האיברים במחסנית. נוכל להתייחס למחסנית כמערך, כאשר SP זהו האינדקס של האיבר הפנוי הבא.
תהליך הכנסת איבר למחסנית יכלול העתקת האיבר ל[Stack[SP ואז קידום SP ב1, על מנת שיצביע למקום הפנוי הבא במחסנית.
שליפת איבר מהמחסנית תקטין ראשית את SP, ואז תחזיר לקורא את הערך המצוי ב[Stack[SP.
בכל רגע נתון, נכנה את האיבר שבראש המחסנית top. האיבר הנמצא ב[stack[0 יקרא הbottom של המחסנית.
במחשבים מודרניים, מנגנון הפונקציות ממומש על המחסנית. הפקודה call בדרך כלל דוחפת את כתובת החזרה מן הפונקציה על המחסנית, ולאחר מכן מעדכנת את הPC. הפקודה return שולפת את כתובת החזרה מראש המחסנית ושמה כתובת זו בPC.
ניתן בעזרת מחסנית גם לקנן שגרות, בתנאי שעומדים **במשטר המחסנית**, האומר שכל המידע שפונקציה הקצתה במהלך ריצתה ישוחרר כאשר יוצאים מהפונקציה. משטר המחסנית מאפשר לפונקציה לדחוף במהלך ריצתה אלמנטים נוספים למחסנית, בתנאי שבסוף ריצת הפונקציה משתנים אלו ישוחררו. בדרך זו, נוכל לספק לפונקציות שלנו **משתנים מקומיים**. הפונקציה תוכל לדחוף ערכים למחסנית ולהשתמש בהם כמשתנים. בסוף ריצת הפונקציה, תשלוף הפונקציה ערכים אלו מהמחסנית. משתנים מקומיים קיימים רק במהלך הריצה הנוכחית של הפונקציה. במהלך הריצה הבאה שלה המשתנים המקומיים יכילו ערכים שונים, וסביר להניח גם שימוקמו במיקום אחר במחסנית.
נגדיר את המושג stack frame, כאוסף כל הערכים שפונקציה שומרת במחסנית במהלך פעולה.
stack frame יחיד כולל:

- משתנים מקומיים שנעשה בהם שימוש במהלך הפונקציה.
- פרמטרים המועברים אל הפונקציה.
- מצב המעבד קודם הקריאה לפונקציה, הכולל ערכים כגון כתובת החזרה וערכים נוספים התלויים במכונה.

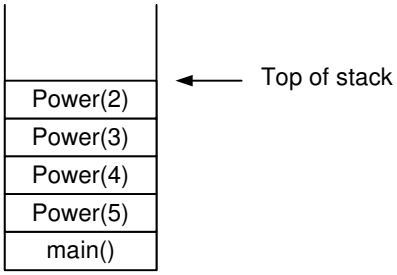
בכל פעם שקוראים לפונקציה, ייווצר stack frame חדש וידחף אל המחסנית. נוכל לדמיין שהמחסנית בנויה מאוסף של stack frames, היושבים אחד על השני, כאשר בתחתית המחסנית נמצא הstack frame של התוכנית הראשית, ומעליו נמצאים הstack frames של הפונקציות המקוננות בתוך הפונקציה הראשית.

למשל, נביט בתוכנית הרקורסיבית הבאה בשפת C:

```
int Power(int n)
{
	if (n == 1) return 1;
	return n*power(n-1);
}

int main()
{
	int result = Power(5);
	printf("5! = %d\n", result);
	return 0;
}
```

תוכנית זו מחשבת את העצרת של 5.
התוכנית מתחילה בפונקציה main, ואז אנו קוראים לפונקציה Power, המחשבת את החזקה באופן רקורסיבי. בשלב מסוים בריצת התוכנית, המחסנית תראה כך:
(נסמן stack frame על ידי שם הפונקציה בצירוף הפרמטרים המגדירים את אותו stack frame).



<u>תהליך הקריאה והחזרה מהפונקציות</u>

תהליך הקריאה והחזרה מהפונקציות, יעשה על ידי פרוטוקול שיכלול את המרכיבים הבאים:

- **תהליך קריאה לפונקציה**, שיכלול את דחיפת הפרמטרים למחסנית, שמירת מצב המעבד על המחסנית והעברת הבקרה אל המחסנית.
- **תהליך הכניסה לפונקציה**, שיכלול הקצאת מקום עבור המשתנים המקומיים של הפונקציה ואיתחולים נוספים.
- **תהליך היציאה מהפונקציה**, שיכלול שחרור המשתנים המקומיים ושחזור מצב המעבד.
- **תהליך החזרה לקורא לפונקציה**, שיכלול שליפת כתובת החזרה מהמחסנית, קפיצה אליה, והחזרת ערכים מהפונקציה לקורא.

פעולת אלו אינן קשורות לתהליכים הפנימיים שמתרחשים בstack frames. ללא שום קשר לפקודות מיוחדות נוספות שמכונות יבצעו בקריאה ובחזרה מפונקציות, הנקודות שצוינו חייבות להיות חלק מכל תהליך של קריאה וחזרה מפונקציה.
<u>מלכודות</u>

מכונות יכולות לאפשר תמיכה במלכודות. מלכודות הינן פונקציות מיוחדות, המאפשרות למתכנת להרחיב במידה מסוימת את הפקודות ששפת המכונה מכירה. מלכודת דומה לפונקציה רגילה, מלבד העובדה שהיא אטומית - עד שהמלכודת מסיימת את פעולתה, המכונה אינה עושה פעולות אחרות.
כדי שנבין את ההבדל בין מלכודת לפונקציה רגילה נסביר כי במערכת בה משתמשים מרובים, למשל, יתכן מצב כי במהלך ביצוע פונקציה, ביצוע הפונקציה ייעצר, המעבד יעבור לכמה מחזורים לבצע פונקציה אחרת שמשתמש אחר ביקש ממנו לבצע, ואז יחזור אל הפונקציה הראשונה. מלכודת לעומת זאת, לא תוכל להיות מופרעת באף מקרה.
הפעולה הבסיסית של מלכודת היא כלהלן: המצב הנוכחי של המעבד נשמר בזיכרון, ומצב חדש נטען לתוך המעבד. לאחר מכן הריצה ממשיכה מהכתובת החדשה של הPC. בסוף ביצוע המלכודת, חוזרים אל הקורא למלכודת על ידי פקודת trap-return מיוחדת.
מצב המעבד, אותו אנו שומרים ומחליפים במהלך הקריאה למלכודת כולל לא רק את המידע הנגיש למשתמש, כגון הPC, אלא גם מידע נוסף הדרוש כדי להגדיר את הפריבילגיות המיוחדות של המלכודת. בדרך כלל מידע זה שמור בבית מיוחד, הנקרא processor status word (PSW). בית זה מכיל, בין היתר, ביט האומר האם המעבד כרגע מבצע פונקציה עם פריבילגיות מיוחדות (מלכודת) או פונקציה רגילה.
המידע על המלכודת - כתובתה של המלכודת, ומצב המעבד הדרוש למלכודת, נשמר באיזור מיוחד בזיכרון הנקרא וקטור המלכודת (trap vector). האזור בו נמצאים ווקטורי המלכודת נקבע מראש על ידי מתכנני האסמבלר.

מכיוון שניתן להתייחס למלכודת כסוג של קריאה לפונקציה, נוכל להשתמש במלכודת H בתוך פונקציה רגילה כלשהי P, זאת כמובן בהנחה שH מקיימת את משטר המחסנית. H תוכל לגשת ללא כל בעיה לערכים שP שמר במחסנית.
לעומת זאת, המצב ההפוך הוא בעייתי יותר. אם נשתמש בפונקציה הרגילה P בתור המלכודת H, אנו מאפשרים לP לשנות את תוכן המחסנית של H, ולמעשה לבצע פעולות בעדיפות גבוהה יותר מהעדיפות הרגילה שלה. לפיכך נמנע ממצב זה - מלכודות הרוצות לקרוא לפונקציות, ישתמשו בסט מיוחד משלהן של פונקציות, ולא יסתמכו על פונקציות רגילות.

Faults

במהלך ביצוע פקודות המכונה, יתכנו תקלות כגון גלישה, ניסיונות לגישה לזיכרון שלא הוקצה, או ניסיונות להריץ פקודות לא חוקיות.
faults הן פונקציות הדומות מעט למלכודות, הבאות לטפל בשגיאות השונות שיכולות להתרחש. בדומה למלכודות, גם לfaults יש עדיפות על פונקציות אחרות.
לכל fault יש ווקטור משלו (Fault Vector), המכיל את הכתובת של הFault, ואת מצב המכונה הרצוי כאשר מגיעים אל הFault. אנו מסיימים fault בעזרת הפקודה trap-return הזהה לפקודה ששימשה אותנו על מנת לצאת מהמלכודות.

בניגוד למלכודות, איננו קוראים לfaults בצורה מפורשת. כאשר מתרחשת שגיאה, הFault המתאים נקרא על מנת לטפל בה.
קטע קוד עשוי לרוץ פעמים רבות ללא כל תקלה, עד אשר צירוף קלטים מסוים יגרום לתקלה, ואז הfault יקרא.

סוג שלישי של פונקציות "מיוחסות", השונה משני הסוגים הקודמים שראינו, הוא פסיקות.
פסיקה היא אות מהתקן חיצוני (למשל מקלדת או מסך) שגורמת למעבד להפסיק לעשות את מה שהוא עושה ולהתחיל לבצע קטע קוד שיטפל בהתקן. המעבד מסיים את ביצוע הפקודה הנוכחית של תוכנית המשתמש, ומיד לאחר כך פונה אל שיגרת הפסיקה.
ההבדל הגדול בין פסיקות למלכודות וfaults, הוא שפסיקות לא נוצרות עקב קריאה מפורשת לפסיקה, או עקב תקלה כלשהי בקוד שגרמה לפסיקה, ולכן בדרך כלל המידע איזו פקודה בוצעה לפני שהגענו אל הפסיקה יהיה לא כל כך רלוונטי.
נשמור עדיין את מצב המעבד הקודם לפני שניגש לביצוע הפסיקה, אך בד"כ לא נתייחס למידע זה.
מבחינת המימוש, פסיקות ממומשות בצורה דומה למלכודות וfaults. לכל פסיקה קיים ווקטור פסיקה (interrupt vector) אשר מכיל את כתובת הפונקציה המטפלת בפסיקה, ואת מצב המעבד הרצוי.
מכיוון שפסיקות מגיעות ממקור חיצוני ולא מתוזמן, הן יכולות לקרוא בכל רגע, דבר המציב קשיים מסוימים כאשר באים לתכנת אותן. לפעמים יהיו מצבים, שנרצה למנוע מפסיקות להתקבל (למשל בזמן ניתוח פסיקה קודמת, או בזמן ביצוע מלכודת). ניתן לבצע זאת על ידי דגל interrupt-enable, שיהיה פעיל אם אנו מעוניינים לקבל פסיקות באותו רגע, או מכובה אם אנו לא רוצים לקבל פסיקות.
גישה נוספת היא יצירת מנגנון עדיפויות, שייתן עדיפות לכל פסיקה, וכן גם עדיפות מיוחדת עבור המעבד, ולפי העדיפויות של כל פסיקה, יקבע האם הפסיקה תתקבל או לא.

שיקולים הקשורים למימוש שפת המכונה

שקיפות

דרישה עיקרית בתכנון הארכיטקטורה היא אי תלות של שפת המכונה בפרטי המימוש שלה. אי תלות זאת בעלת ערך רב, במקרים בהם נרצה לשפר את מימוש המעבד כדי להשיג ביצועים טובים יותר, מבלי להפוך תוכנות שנכתבו בשפת המכונה הישנה לחסרות שימוש. שיפורים כאלו אפשריים, רק אם שפת המכונה איננה תלויה כלל בפרטי המימוש שלה.

היררכיות של זיכרון

במחשב ישנם זיכרונות מסוגים שונים, הנבדלים בביצועים השונים שלהם. בין הזיכרונות השונים ניתן בדרך כלל למצוא:

- מספר רגיסטרים היושבים על המעבד, המשמשים כרכיבי זיכרון מהירים
- זיכרון ראשי, האיטי מהרגיסטרים, ומכיל כמות גדולה של תאים בהם התוכנות יכולות להשתמש.
- זיכרון משני –דיסקים, כוננים קשיחים ועוד. זיכרון זה איטי עוד יותר מהזיכרון הראשי.

בדרך כלל מכונות כוללות שיטות מיעון שונות המשמשות לגשת להתקני הזיכרון השונים.

דוגמא לכך היא למשל הG-Machine, המכילה שיטת מיעון מיוחדת המשמשת לגישה לרגיסטרים הכלליים, ושיטות מיעון אחרות המשמשות על מנת לגשת אל הDRAM.

גישה זו נותנת לנו מספר יתרונות:

- אנו נותנים למכונה אפשרות לבצע פעולות קריטיות במהירות האפשרית, על ידי שימוש ברגיסטרים במקום בזיכרון הראשי.
- אנו יכולים להקטין את גודל הקוד של התוכנית, מכיוון שפעולות שניגשות רק לרגיסטרים יכולות להיות מקודדות על פחות בתים, ובכל אנו חוסכים מקום בזיכרון.

אם זאת, לגישה יש גם מספר חסרונות:

- הפסדת שקיפות המימוש - מספר הרגיסטרים הממשי בכל חומרה הוא פרט הקשור למימוש האסמבלר, שאותו נרצה להסתיר. בG-Machine, מספר הרגיסטרים הוא קבוע, ללא תלות במכונה (16), ולכן נוכל להגיד לכאורה כי G-Machine איננו תלוי במימוש. אולם, אם תהיה לנו חומרה חזקה יותר, בעלת יותר רגיסטרים, נאלץ לשנות את עיצוב הG-Machine אם נרצה לנצל את האפשרויות החדשות.
- סיבוך התכנות - מימוש הקומפיילר ייהפך קשה יותר, מכיוון שכעת הקומפיילר ייאלץ להכיל מנגנון שיקבע עבור כל פקודה, האם לתת לה להשתמש ברגיסטרים של המכונה או בזיכרון הראשי.

מעבדים בעלי ביצועים גבוהים

עד כה, הצגנו מספר הצעות כיצד לתכנן את הארכיטקטורה וכיצד לממש אותה, על מנת לשפר את ביצועי המכונה.
כעת נציג הצעות לשיפור ביצועי המעבד עצמו.
השיפור שנעשה יהיה הוספת רעיונות של עבודה במקביל למחשב אותו אנו מתכננים.
נרשה לעצמנו להמשיך להניח כי המכונה היא Single Sequence Machine, וזאת מכיוון, שלמרות שנרשה כי פקודות שונות ינותחו בו זמנית, נדרוש כי תוצאת ביצוע הפקודות במקביל תהיה זהה לתוצאה שהיינו מקבלים, אם היינו מבצעים את הפקודות אחת אחרי השניה.

נשתמש בשני עקרונות כדי לשפר את ביצועי המעבד:

- Pipelining - כל פקודה של התוכנית עוברת דרך מספרים שלבים (phases) במהלך ביצועה: קריאת הפקודה מערוץ הפקודות (fetch), פענוח הפקודה, פענוח האופרנדים, הרצת הפקודה ושמירת תוצאות החישוב. נוכל לשפר את ביצועי המעבד, על ידי כך שבמקום שנבצע את השלבים אחד אחרי השני, נשתמש בpipelining - ננתח מספר פקודות בו זמנית, כאשר כל פקודה תהיה ברגע נתון בשלב אחר של ביצועה.
- מספר יחידות הרצה - בנוסף לpipeline, נוכל לשלב מספר יחידות להרצת הפקודות. במערכות המשתמשות בשיטה זו, נקצה לכל פקודה יחידת הרצה משלה. במערכות כאלו ישנו מנגנון הבודק תוך כדי ריצה, אילו פקודות אפשר לבצע בו זמנית, שביצוען איננו תלוי בתוצאות של הפעולות האחרות, ומבצע אותן.

שיטות אלו גורמות לשיפור רב בביצועים, אם סט הפקודות הוא פשוט - מבני. מבנה פקודות פשוט מקל על ביצוע pipeline יעיל על ידי מזעור כמות העבודה הדרושה לעשות על פקודה לפני שניתן לנתח אותה, ובכך ניצול הpipeline באופן רציף.

Stored Programs Model

מחשבים מודרניים נקראים בדרך כלל stored programs computers, וזאת במובן שהתוכנות שהם מריצים נשמרות בזיכרון, ליד הנתונים אשר מנותחים על ידי התוכנות. (האלטרנטיבה לשמירת התוכנית בזיכרון, היא שמירתה בROM, או ברכיב אחר המממש אותה).

מכיוון שהתוכנית נשמרת עם הנתונים, אנחנו לכאורה מאפשרים כאן גם תוכנות שהן **Self-modifying programs**. תוכנות כאלו הן תוכנות המתייחסות אל הקידוד שלהן בזיכרון כאל נתונים, ומשנות את עצמן תוך כדי ריצה.
כתיבת תוכניות כאלו איננה מומלצת. האפשרות לשנות את הקוד של התוכנית נובע מההיסטוריה, כאשר היה מחסור בזיכרון ומתכנתים ניסו לחסוך כל מקום אפשרי.

שלבי פענוח שפת המכונה

- שליפת הפקודה מהזיכרון.
- פענוח הפקודה.
- פענוח האופרנדים.
- ביצוע הפקודה.
- החזרת תוצאות הפקודה.

מבנה פקודה

כאשר אנו באים לקודד את הפקודה בזיכרון, נוכל לקודד אותה בצורות שונות. קידוד כל פקודה במספר רב של בתים יהפוך את מימוש האסמבלר לפשוט ומבני יותר, אך ידרוש יותר זיכרון.

מבנה של 3+1 כתובות:

Opcode	Operand1	Operand2	Result	Next Instruction

Opcode – הקוד שיזהה את הפעולה הרצויה לביצוע.
Operand1, Operand2 – אופרנדים – ערכים, כתובות זיכרון בהן הערכים
Result – תוצאה (כתובת)
Next Instruction – כתובת הפקודה הבאה

יתרון: כל פקודה מכילה את כל המידע הדרוש.
חסרון: הפקודה נהיית מאוד רחבה

מבנה פקודה של 3 כתובות (ללא שדה הפקודה הבאה):

OpCode	Operand1	Operand2	Result

יתרון:
1. חיסכון מקום בזיכרון

חסרונות:
1. לא ניתן לפזר את הפקודות בזיכרון
2. צריך לכתוב מנגנון שקובע מה הפקודה הבאה לביצוע - רגיסטר PC.

מבנה פקודה של 2 כתובות:

OpCode	Operand1	Operand2

אחד האופרנדים ישמש גם לקלט וגם לפלט, וכך נוכל לחסוך את שדה הresult.

מבנה פקודה של כתובת אחת:

שימוש בצובר – Accumulator
הצובר הוא רגיסטר, שהוא משתנה של המכונה.
הרעיון של מבני פקודה של כתובת אחת הוא שישנו Opcode, כתובת אחת שהיא האופרנד, והכתובת השניה נאגרת בצובר, במיקום קבוע.
היתרון של שיטה כזו הוא פישוט פענוח האופרנדים, מכיוון שהאופרנד השני מצוי בכתובת קבועה, ולכן אין צורך לחשב את הכתובת האפקטיבית שלו.

RISC/CISC architectures

מכונות מסוג RISC הן מכונות, בהן גודל כל פקודה הוא קבוע, ללא התחשבות בתוכנה. הMAYBE היא דוגמא פשוטה לכך. כאשר צרבנו את הControl ROM, כל פקודה היא בגודל קבוע. למשל, נניח נרצה למצוא מה צריך לבצע כאשר אנחנו מבצעים את הphase 0001 של פקודה שהopcode שלה הוא 0x74, אנו יודעים בדיוק היכן צריך להסתכל בROM כדי למצוא את ההוראות לביצוע.
לעומת זאת, במכונות מסוג CISC הפקודות הן באורך משתנה. אורך הפקודה נקבע על פי הOpcode והאופרנדים השונים המועברים לפקודה. G-Machine היא דוגמא למכונה מסוג CISC. יש לה שיטות מיעון רבות המשפיעים על גודל האופרנדים ויש לה פקודות באורכים שונים.
היתרון של RISC הוא אחידות. אחידות עוזרת בכך שמכיוון שאנו יודעים איפה כל פקודה מתחילה ומסתיימת, ואיפה ממוקמים האופרנדים בזיכרון, ניתן לבצע פעולות שונות במקביל.
החסרון הוא ששפת המכונה של RISC תתפוס מקום רב יותר בזיכרון, כי הרי כל הפקודות יקודדו בצורה כזו שתתאים לפקודה הארוכה ביותר של המכונה.
ארכיטקטורות CICS חוסכות בזיכרון, אולם ניתוח הפקודה נעשה מסובך יותר, וקשה יותר לבצע פקודות במקביל.

```
|=============================================================================
| CTLROM.PROM            MAYBE II  Control ROM Code ...         1985-07-14
|                        I/O Functions added                    1986-10-25 JMM
|=============================================================================
|
|   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
|    Control Instruction Format:  CTLROM-OPCODE  =  CTLROM-HIGH  CTLROM-LOW
|   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
|
|                              .------------------..----------..-------.
|       CTLROM-OPCODE :        |      OPCODE      ||  CLOCK   || Cond. |
|       ~~~~~~~~~~~~~         | 7 6 5 4 3 2 1 0 || 3 2 1 0 ||  Bit   |
|                              `------------------'`----------'`-------'
|                              .------..-------------------------..---.
|       CTLROM HIGH   :        |  1+  ||           ALU           || ? |
|       ~~~~~~~~~~~           |------||---------.-------.------||---|
|                              | ADR+ ||   FUNC  | C in  | Mode || N |
|                              | Bit  || 3 2 1 0 | Bit   | Bit  || C |
|                              `------'`---------^-------^------'`---'
|                               .-----------------..---------------.
|       CTLROM LOW    :         |      CCREG      ||   DATA BUS     |
|       ~~~~~~~~~~             |-------.---------||-------.-------|
|                               | CLOCK | SHIFT   || DRSEL | LDSEL |
|                               | INHIB.| & LOAD  || 2 1 0 | 2 1 0 |
|                               `-------^---------'`-------^-------'
|
|   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
|                    DATA BUS device decoder implementation ...
|   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
|
|                              .--------------..---------------..---------.
|       DRIVER SELECT (DRSEL): | S2 | S1 | S0 || DRIVER DEVICE ||  ABBR.  |
|       ~~~~~~~~~~~~~~~~~~~   |--------------||---------------||---------|
|                              |  0 |  0 |  0 ||   SWITCHES    ||  DRSWT  |
|                              |  0 |  0 |  1 ||   uCODE ROM   || DRuROM  |
|                              |  0 |  1 |  0 ||  ALU output   ||  DRALU  |
|                              |  0 |  1 |  1 ||   ????????   || ( NOP ) |
|                              |  1 |  0 |  0 ||  Static RAM   || DRSRAM  |
|                              |  1 |  0 |  1 ||   ????????   || ( NOP ) |
|                              |  1 |  1 |  0 ||  Dynamic RAM  || DRDRAM  |
|                              |  1 |  1 |  1 || External Comm.||  DRI/O  |
|                              `----^----^----'`---------------'`---------'
|                              .--------------..-------------------..--------.
|       LOADER SELECT (LDSEL): | S2 | S1 | S0 ||   LOADER DEVICE   ||  ABBR. |
|       ~~~~~~~~~~~~~~~~~~~   |--------------||-------------------||--------|
|                              |  0 |  0 |  0 || CTLROM OPCODE Reg.||  LDOP  |
|                              |  0 |  0 |  1 ||  uROM  Addr.  Reg.|| LDADR  |
|                              |  0 |  1 |  0 ||    ALU "A" Reg.   ||  LDA   |
|                              |  0 |  1 |  1 ||    ALU "B" Reg.   ||  LDB   |
|                              |  1 |  0 |  0 ||  Static RAM datum || LDSRAM |
|                              |  1 |  0 |  1 ||  Memory Addr.Reg. || LDMAR  |
|                              |  1 |  1 |  0 ||    Dynamic RAM    || LDDRAM |
|                              |  1 |  1 |  1 ||   External Comm.  || LDI/O  |
|                              `----^----^----'`-------------------'`--------'
|
|======
| NOTE  LDA is also tied to LDI/OADR so that loading the ALU A Reg also loads
|====== the I/O Address Register.

| SRAM allocation conventions:
|-----------------------------
|       0xFF      Stack pointer.  Points to top stack element.
|       0xFE      refresh counter.
|       0xFD-F0   reserved, e.g. for microinstruction temporaries
|       0xEF down microstack.

|>>>>>>>>>>>>>>>>>>>>
|>>> UNUSED OPCODE <<<  Opcode  00000000 is reserved for an I/O hack opcode
|>>>>>>>>>>>>>>>>>>>>

|               C    A       M     I
|               o    D ALU   Co    nS DR  LD    |
|  OPCODE  CLOK n == R FUNC id N   h/ SEL SEL   | COMMENTS
| 76543210 3210 d    + 3210 ne C   bL 210 210   |

| call(rtnlo, rtnhi, subrlo, subrhi)
  00000001 0000 *  = 0 1100 11 1   11 010 101   | MAR <- 0xFF
  00000001 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  00000001 0010 *  = 0 1111 11 1   11 010 101   | MAR <- A
  00000001 0011 *  = 1 1111 11 1   11 001 100   | SRAM <- uROM; ADR+
  00000001 0100 *  = 0 1111 10 1   11 010 010   | A   <- A-1
  00000001 0101 *  = 0 1111 11 1   11 010 101   | MAR <- A
  00000001 0110 *  = 1 1111 11 1   11 001 100   | SRAM <- uROM; ADR+
```

```
  00000001 0111 *  = 0 1100 11 1   11 010 101   | MAR <- 0xFF
  00000001 1000 *  = 0 1111 10 1   11 010 100   | SRAM <- A-1
  00000001 1001 *  = 1 1111 11 1   11 001 010   | A   <- uROM; ADR+
  00000001 1010 *  = 0 1111 11 1   11 001 001   | ADR <- uROM;
  00000001 1011 *  = 0 1111 11 1   11 010 001   | ADR <- A
  00000001 1100 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| rtn()
| NB: This opcode (2) is built into ctlrom code for urom(), below.
| DO NOT CHANGE IT without modifying urom code.
  00000010 0000 *  = 0 1100 11 1   11 010 101   | MAR <- 0xFF
  00000010 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  00000010 0010 *  = 0 0000 00 1   11 010 010   | A   <- A+1
  00000010 0011 *  = 0 1111 11 1   11 010 101   | MAR <- A
  00000010 0100 *  = 0 1111 11 1   11 100 001   | ADR <- SRAM
  00000010 0101 *  = 0 0000 00 1   11 010 010   | A   <- A+1
  00000010 0110 *  = 0 1111 11 1   11 010 101   | MAR <- A
  00000010 0111 *  = 0 1111 11 1   11 100 001   | ADR <- SRAM
  00000010 1000 *  = 0 1100 11 1   11 010 101   | MAR <- 0xFF
  00000010 1001 *  = 0 1111 11 1   11 010 100   | SRAM <- A
  00000010 1010 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| push(x)
  00000011 0000 *  = 0 1100 11 1   11 010 101   | MAR <- 0xFF
  00000011 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  00000011 0010 *  = 0 1111 10 1   11 010 100   | SRAM <- A-1
  00000011 0011 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00000011 0100 *  = 0 1111 11 1   11 100 011   | B   <- SRAM
  00000011 0101 *  = 0 1111 11 1   11 010 101   | MAR <- A
  00000011 0110 *  = 0 1010 11 1   11 010 100   | SRAM <- B
  00000011 0111 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| pop(x)
  00000100 0000 *  = 0 1100 11 1   11 010 101   | MAR <- 0xFF
  00000100 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  00000100 0010 *  = 0 0000 00 1   11 010 010   | A   <- A+1
  00000100 0011 *  = 0 1111 11 1   11 010 100   | SRAM <- A
  00000100 0100 *  = 0 1111 11 1   11 010 101   | MAR <- A
  00000100 0101 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  00000100 0110 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00000100 0111 *  = 0 1111 11 1   11 010 100   | SRAM <- A
  00000100 1000 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| move(x, y)     [move value]    y <- <x>
  00000101 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00000101 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  00000101 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00000101 0011 *  = 0 1111 11 1   11 010 100   | SRAM <- A
  00000101 0100 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| cmove(cx, y)  [move constant]  y <- cx
  00000110 0000 *  = 1 1111 11 1   11 001 010   | A   <- uROM; ADR+
  00000110 0001 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00000110 0010 *  = 0 1111 11 1   11 010 100   | SRAM <- A
  00000110 0011 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| swt(x)         x <- switches
  00000111 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00000111 0001 *  = 0 1111 11 1   11 000 100   | SRAM <- SWT
  00000111 0010 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| pdisp(x, p00, p01, p10, p11)
| Displays <x> in the lights.  Dispatches to (2-byte) location
|   pyz where y is state of P0, z is state of P1.
| Does not put data other than <x> in lights, hence suitable for
|   tight "prompt" loop.
  00001000 0000 *  = 1 1111 11 1   00 001 101   | Latch CCs; MAR <- uROM; ADR+
  00001000 0001 *  = 0 1111 11 1   01 100 011   | B <- SRAM; Shift CCs
  | Following 3 states shift P0 into CCs, and load 0xFE (refresh counter adr)
  |   into SRAM MAR.
  00001000 0010 *  = 0 1100 11 1   01 010 010   | Shift CCs; A <- 0xFF
  00001000 0011 *  = 0 1111 10 1   01 010 101   | Shift CCs; MAR <- A-1 (=0xFE)
  00001000 0100 *  = 0 1111 11 1   01 010 010   | Shift CCs
  | In either P0 case, they refresh one DRAM column.
  | If P0=1, uROM adr is incremented 4 times as well to point to p10 in uROM.
  00001000 0101 n  = n 1111 11 1   11 100 010   | if P0, ADR+; A <- SRAM
  00001000 0110 n  = n 1111 11 1   11 010 110   | if P0, ADR+; DRAM <- A
  00001000 0111 n  = n 0000 00 1   11 010 100   | if P0, ADR+; SRAM <- A+1
  00001000 1000 n  = n 1111 11 1   01 010 010   | if P0, ADR+; Shift CCs

  00001000 1001 n  = n 1111 11 1   11 010 010   | if P1, ADR+ (else NOP)
  00001000 1010 n  = n 1111 11 1   11 010 010   | if P1, ADR+ (else NOP)

  00001000 1011 *  = 1 1111 11 1   11 001 010   | A   <- uROM; ADR+
  00001000 1100 *  = 0 1111 11 1   11 001 001   | ADR <- uROM
  00001000 1101 *  = 0 1111 11 1   11 010 001   | ADR <- A
  00001000 1110 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+
```

```
| pwrup()
  00001001 **** * == 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+
  00001001 0000 *  = 1 1111 11 1   11 010 010   | A   <- A; ADR+

| add(x, y, z)
  00001010 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00001010 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  00001010 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00001010 0011 *  = 0 1111 11 1   11 100 011   | B   <- SRAM
  00001010 0100 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00001010 0101 *  = 0 1001 10 1   00 010 100   | SRAM <- A+B, Latch CCs
  00001010 0110 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| cadd(cx, y, z)
  00001011 0000 *  = 1 1111 11 1   11 001 010   | A   <- uROM; ADR+
  00001011 0001 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00001011 0010 *  = 0 1111 11 1   11 100 011   | B   <- SRAM
  00001011 0011 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00001011 0100 *  = 0 1001 10 1   00 010 100   | SRAM <- A+B, Latch CCs
  00001011 0101 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| jmp(adrlo, adrhi)
  00001100 0000 *  = 1 1111 11 1   11 001 010   | A   <- uROM; ADR+
  00001100 0001 *  = 0 1111 11 1   11 001 001   | ADR <- uROM
  00001100 0010 *  = 0 1111 11 1   11 010 001   | ADR <- A
  00001100 0011 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| imove(x, y)    [move indir value]    y <- <<x>>
  00001101 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00001101 0001 *  = 0 1111 11 1   11 100 101   | MAR <- SRAM
  00001101 0010 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  00001101 0011 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00001101 0100 *  = 0 1111 11 1   11 010 100   | SRAM <- A
  00001101 0101 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| movei(x, y)    [move value indir]    <y> <- <x>
  00001110 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00001110 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  00001110 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00001110 0011 *  = 0 1111 11 1   11 100 101   | MAR <- SRAM
  00001110 0100 *  = 0 1111 11 1   11 010 100   | SRAM <- A
  00001110 0101 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| l(adrlo, adrhi, where) -- DRAM load
  00001111 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00001111 0001 *  = 0 1111 11 1   11 100 010   | A  <- SRAM
  00001111 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00001111 0011 *  = 0 1111 11 1   11 010 110   | DRAM <- A (RAS)
  00001111 0100 *  = 0 1111 11 1   11 100 110   | DRAM <- SRAM (CAS)
  00001111 0101 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00001111 0110 *  = 0 1111 11 1   11 110 100   | SRAM <- DRAM
  00001111 0111 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| s(value, adrlo, adrhi) -- DRAM store
  00010000 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00010000 0001 *  = 0 1111 11 1   11 100 011   | B  <- SRAM (value)
  00010000 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00010000 0011 *  = 0 1111 11 1   11 100 010   | A <- SRAM (row)
  00010000 0100 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00010000 0101 *  = 0 1111 11 1   11 010 110   | DRAM <- A (row)
  00010000 0110 *  = 0 1111 11 1   11 100 110   | DRAM <- SRAM (CAS)
  00010000 0111 *  = 0 1010 11 1   11 010 110   | DRAM <- B
  00010000 1000 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| refr() - refresh 6 DRAM rows.
  00010001 0000 *  = 0 1100 11 1   11 010 010   | A <- 0xFF
  00010001 0001 *  = 0 1111 10 1   11 010 101   | MAR <- A-1 (= 0xFE)
  00010001 0010 *  = 0 1111 11 1   11 100 010   | A   <- SRAM

  00010001 0011 *  = 0 1111 11 1   11 010 110   | DRAM <- A
  00010001 0100 *  = 0 0000 00 1   11 010 010   | A <- A+1

  00010001 0101 *  = 0 1111 11 1   11 010 110   | DRAM <- A
  00010001 0110 *  = 0 0000 00 1   11 010 010   | A <- A+1

  00010001 0111 *  = 0 1111 11 1   11 010 110   | DRAM <- A
  00010001 1000 *  = 0 0000 00 1   11 010 010   | A <- A+1

  00010001 1001 *  = 0 1111 11 1   11 010 110   | DRAM <- A
  00010001 1010 *  = 0 0000 00 1   11 010 010   | A <- A+1

  00010001 1011 *  = 0 1111 11 1   11 010 110   | DRAM <- A
  00010001 1100 *  = 0 0000 00 1   11 010 010   | A <- A+1

  00010001 1101 *  = 0 1111 11 1   11 010 110   | DRAM <- A
  00010001 1110 *  = 0 0000 00 1   11 010 100   | SRAM <- A+1
```

```
  00010001 1111 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| Conditional jumps CLOBBER CCs!

| jmi(adrlo, adrhi) -- jump if N set.
  00010010 0000 *  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+

  00010010 0001 1  = 0 1111 11 1   11 001 001   | ADR <- uROM
  00010010 0010 1  = 0 1111 11 1   11 010 001   | ADR <- A
  00010010 0011 1  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

  00010010 0001 0  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+
  00010010 0010 0  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| jextra(adrlo, adrhi) -- jump if extra (i.e., undefined) flag set.
  00010011 0000 *  = 1 1111 11 1   00 001 010   | A <- uROM; ADR+; Latch CCs

  00010011 0001 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00010011 0010 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00010011 0011 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00010011 0100 *  = 0 1111 11 1   01 010 010   |
  00010011 0101 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00010011 0110 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00010011 0111 *  = 0 1111 11 1   01 010 010   | Shift CCs

  00010011 1000 1  = 0 1111 11 1   11 001 001   | ADR <- uROM
  00010011 1001 1  = 0 1111 11 1   11 010 001   | ADR <- A
  00010011 1010 1  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

  00010011 1000 0  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+
  00010011 1001 0  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| jready(adrlo, adrhi) -- jump if I/O flag set. refresh one row.
  00010100 0000 *  = 0 1100 11 1   00 010 010   | A <- 0xFF

  00010100 0001 *  = 0 1111 10 1   01 010 101   | Shift CCs; MAR <- A-1 (=0xFE)
  00010100 0010 *  = 0 1111 11 1   01 100 010   | Shift CCs; A <- SRAM
  00010100 0011 *  = 0 1111 11 1   01 010 110   | Shift CCs; DRAM <- A
  00010100 0100 *  = 0 0000 00 1   01 010 100   | Shift CCs; SRAM <- A+1
  00010100 0101 *  = 1 1111 11 1   01 001 010   | Shift CCs; A <- uROM; ADR+
  00010100 0110 *  = 0 1111 11 1   01 010 010   | Shift CCs

  00010100 0111 1  = 0 1111 11 1   11 001 001   | ADR <- uROM
  00010100 1000 1  = 0 1111 11 1   11 010 001   | ADR <- A
  00010100 1001 1  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

  00010100 0111 0  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+
  00010100 1000 0  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| jp1(adrlo, adrhi) -- jump if P1 pressed.
  00010101 0000 *  = 1 1111 11 1   00 001 010   | A <- uROM; ADR+; Latch CCs

  00010101 0001 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00010101 0010 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00010101 0011 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00010101 0100 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00010101 0101 *  = 0 1111 11 1   01 010 010   | Shift CCs

  00010101 0110 1  = 0 1111 11 1   11 001 001   | ADR <- uROM
  00010101 0111 1  = 0 1111 11 1   11 010 001   | ADR <- A
  00010101 1000 1  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

  00010101 0110 0  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+
  00010101 0111 0  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| jodd(adrlo, adrhi) -- jump if low ALU bit set.
  00010110 0000 *  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+

  00010110 0001 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00010110 0010 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00010110 0011 *  = 0 1111 11 1   01 010 010   | Shift CCs

  00010110 0100 1  = 0 1111 11 1   11 001 001   | ADR <- uROM
  00010110 0101 1  = 0 1111 11 1   11 010 001   | ADR <- A
  00010110 0110 1  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

  00010110 0100 0  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+
  00010110 0101 0  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| jnc(adrlo, adrhi) -- jump if Carry bit not set
  00010111 0000 *  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+

  00010111 0001 *  = 0 1111 11 1   01 010 010   | Shift CCs

  00010111 0010 1  = 0 1111 11 1   11 001 001   | ADR <- uROM
  00010111 0011 1  = 0 1111 11 1   11 010 001   | ADR <- A
```

```
  00010111 0100 1  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

  00010111 0010 0  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+
  00010111 0011 0  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| jp0(adrlo, adrhi) -- jump if P0 pressed.
  00011000 0000 *  = 1 1111 11 1   00 001 010   | A <- uROM; ADR+; Latch CCs

  00011000 0001 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00011000 0010 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00011000 0011 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00011000 0100 *  = 0 1111 11 1   01 010 010   | Shift CCs

  00011000 0101 1  = 0 1111 11 1   11 001 001   | ADR <- uROM
  00011000 0110 1  = 0 1111 11 1   11 010 001   | ADR <- A
  00011000 0111 1  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

  00011000 0101 0  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+
  00011000 0110 0  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| je(adrlo, adrhi) -- jump if Equal to 11111111 (0xFF)
  00011001 0000 *  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+

  00011001 0001 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00011001 0010 *  = 0 1111 11 1   01 010 010   | Shift CCs

  00011001 0011 1  = 0 1111 11 1   11 001 001   | ADR <- uROM
  00011001 0100 1  = 0 1111 11 1   11 010 001   | ADR <- A
  00011001 0101 1  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

  00011001 0011 0  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+
  00011001 0100 0  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

|>>>>>>>>>>>>>>>>>>>>>
|>>> UNUSED OPCODES <<<  Opcodes  00011010 & 00011011  are NOT currently used.
|>>>>>>>>>>>>>>>>>>>>>                                  ===

| sub(x, y, z) -- z <- x - y
  00011100 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00011100 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  00011100 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00011100 0011 *  = 0 1111 11 1   11 100 011   | B   <- SRAM
  00011100 0100 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00011100 0101 *  = 0 0110 00 1   00 010 100   | SRAM <- A-B, Latch CCs
  00011100 0110 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| cmp(x, y) --   x-y-1; Latch CCs
| This operation produces no result other than in the condition codes.
| After the cmp(x,y) the significance of the condition codes is:
|        E:     1 iff <x> = <y>
|       -C:     0 iff <x> > <y> (unsigned) ... i.e. is asserted active low.
  00011101 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00011101 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  00011101 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00011101 0011 *  = 0 1111 11 1   11 100 011   | B   <- SRAM
  00011101 0100 *  = 0 0110 10 1   00 010 010   | A <- A-B-1, Latch CCs
  00011101 0101 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| jpl(adrlo, adrhi) -- jump if N not set.
  00011110 0000 *  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+

  00011110 0001 0  = 0 1111 11 1   11 001 001   | ADR <- uROM
  00011110 0010 0  = 0 1111 11 1   11 010 001   | ADR <- A
  00011110 0011 0  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

  00011110 0001 1  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+
  00011110 0010 1  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| jnextra(adrlo, adrhi) -- jump if extra (i.e., undefined) flag not set.
  00011111 0000 *  = 1 1111 11 1   00 001 010   | A <- uROM; ADR+; Latch CCs

  00011111 0001 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00011111 0010 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00011111 0011 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00011111 0100 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00011111 0101 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00011111 0110 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00011111 0111 *  = 0 1111 11 1   01 010 010   | Shift CCs

  00011111 1000 0  = 0 1111 11 1   11 001 001   | ADR <- uROM
  00011111 1001 0  = 0 1111 11 1   11 010 001   | ADR <- A
  00011111 1010 0  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

  00011111 1000 1  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+
  00011111 1001 1  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| jnready(adrlo, adrhi) -- jump if I/O ready flag not set. refresh 1 row.
```

```
  00100000 0000 *  = 0 1100 11 1   00 010 010   | A <- 0xFF

  00100000 0001 *  = 0 1111 10 1   01 010 101   | Shift CCs; MAR <- A-1 (=0xFE)
  00100000 0010 *  = 0 1111 11 1   01 100 010   | Shift CCs; A <- SRAM
  00100000 0011 *  = 0 1111 11 1   01 010 110   | Shift CCs; DRAM <- A
  00100000 0100 *  = 0 0000 00 1   01 010 100   | Shift CCs; SRAM <- A+1
  00100000 0101 *  = 1 1111 11 1   01 001 010   | Shift CCs; A <- uROM; ADR+
  00100000 0110 *  = 0 1111 11 1   01 010 010   | Shift CCs

  00100000 0111 0  = 0 1111 11 1   11 001 001   | ADR <- uROM
  00100000 1000 0  = 0 1111 11 1   11 010 001   | ADR <- A
  00100000 1001 0  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

  00100000 0111 1  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+
  00100000 1000 1  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| jnp1(adrlo, adrhi) -- jump if P1 not pressed.
  00100001 0000 *  = 1 1111 11 1   00 001 010   | A <- uROM; ADR+; Latch CCs

  00100001 0001 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00100001 0010 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00100001 0011 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00100001 0100 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00100001 0101 *  = 0 1111 11 1   01 010 010   | Shift CCs

  00100001 0110 0  = 0 1111 11 1   11 001 001   | ADR <- uROM
  00100001 0111 0  = 0 1111 11 1   11 010 001   | ADR <- A
  00100001 1000 0  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

  00100001 0110 1  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+
  00100001 0111 1  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| jeven(adrlo, adrhi) -- jump if low ALU bit not set.
  00100010 0000 *  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+

  00100010 0001 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00100010 0010 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00100010 0011 *  = 0 1111 11 1   01 010 010   | Shift CCs

  00100010 0100 0  = 0 1111 11 1   11 001 001   | ADR <- uROM
  00100010 0101 0  = 0 1111 11 1   11 010 001   | ADR <- A
  00100010 0110 0  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

  00100010 0100 1  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+
  00100010 0101 1  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| jc(adrlo, adrhi) -- jump if Carry bit set
  00100011 0000 *  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+

  00100011 0001 *  = 0 1111 11 1   01 010 010   | Shift CCs

  00100011 0010 0  = 0 1111 11 1   11 001 001   | ADR <- uROM
  00100011 0011 0  = 0 1111 11 1   11 010 001   | ADR <- A
  00100011 0100 0  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

  00100011 0010 1  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+
  00100011 0011 1  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| jnp0(adrlo, adrhi) -- jump if P0 not pressed.
  00100100 0000 *  = 1 1111 11 1   00 001 010   | A <- uROM; ADR+; Latch CCs

  00100100 0001 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00100100 0010 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00100100 0011 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00100100 0100 *  = 0 1111 11 1   01 010 010   | Shift CCs

  00100100 0101 0  = 0 1111 11 1   11 001 001   | ADR <- uROM
  00100100 0110 0  = 0 1111 11 1   11 010 001   | ADR <- A
  00100100 0111 0  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

  00100100 0101 1  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+
  00100100 0110 1  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| jne(adrlo, adrhi) -- jump if Not Equal to 11111111 (0xFF)
  00100101 0000 *  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+

  00100101 0001 *  = 0 1111 11 1   01 010 010   | Shift CCs
  00100101 0010 *  = 0 1111 11 1   01 010 010   | Shift CCs

  00100101 0011 0  = 0 1111 11 1   11 001 001   | ADR <- uROM
  00100101 0100 0  = 0 1111 11 1   11 010 001   | ADR <- A
  00100101 0101 0  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

  00100101 0011 1  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+
  00100101 0100 1  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| urom(adrlo, adrhi, x)
```

```
| read ucode uROM[SRAM[adrlo], SRAM[adrhi]] into SRAM[x]; then forces a rtn().
| NB: This code MUST change iff the opcode for RTN changes!
  00100110 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+; adrlo
  00100110 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM; <adrlo>
  00100110 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+; adrhi
  00100110 0011 *  = 0 1111 11 1   11 100 011   | B   <- SRAM; <adrhi>
  00100110 0100 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+; x
  00100110 0101 *  = 0 1010 11 1   11 010 001   | ADR <- B; <adrhi>
  00100110 0110 *  = 0 1111 11 1   11 010 001   | ADR <- A; <adrlo>
  00100110 0111 *  = 0 1111 11 1   11 001 100   | SRAM <- uROM; <adrlo,adrhi>
  00100110 1000 *  = 0 0011 11 1   11 010 010   | A   <- 0
  00100110 1001 *  = 0 0000 00 1   11 010 010   | A   <- A+1 (= 1)
  00100110 1010 *  = 0 0000 00 1   11 010 010   | A   <- A+1 (= 2, RTN opcode)
  00100110 1011 *  = 0 1111 11 1   11 010 000   | OPCODE <- A; force RTN

| and(x, y, z)
  00100111 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00100111 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  00100111 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00100111 0011 *  = 0 1111 11 1   11 100 011   | B   <- SRAM
  00100111 0100 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00100111 0101 *  = 0 1011 11 1   00 010 100   | SRAM <- A&B, Latch CCs
  00100111 0110 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| or(x, y, z)
  00101000 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101000 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  00101000 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101000 0011 *  = 0 1111 11 1   11 100 011   | B   <- SRAM
  00101000 0100 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101000 0101 *  = 0 1110 11 1   00 010 100   | SRAM <- A|B, Latch CCs
  00101000 0110 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| xor(x, y, z)
  00101001 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101001 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  00101001 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101001 0011 *  = 0 1111 11 1   11 100 011   | B   <- SRAM
  00101001 0100 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101001 0101 *  = 0 0110 11 1   00 010 100   | SRAM <- A xor B, Latch CCs
  00101001 0110 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| not(x, y)
  00101010 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101010 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  00101010 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101010 0011 *  = 0 0000 11 1   00 010 100   | SRAM <- not A; Latch CCs
  00101010 0100 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| neg(x, y)
  00101011 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101011 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  00101011 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101011 0011 *  = 0 0000 11 1   00 010 010   | A   <- not A
  00101011 0100 *  = 0 0000 00 1   00 010 100   | SRAM <- A+1; Latch CCs
  00101011 0101 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| Following are a few 2-byte operators

| add2(xlo, ylo, zlo, xhi, yhi, zhi) -- 16-bit add.
  00101100 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101100 0001 *  = 0 1111 11 1   11 100 010   | A <- SRAM
  00101100 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101100 0011 *  = 0 1111 11 1   11 100 011   | B <- SRAM
  00101100 0100 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101100 0101 *  = 0 1001 10 1   00 010 100   | SRAM <- A+B, Latch CCs
  00101100 0110 *  = 1 1111 11 1   01 001 101   | MAR <- uROM; ADR+; Shift CCs
  00101100 0111 *  = 0 1111 11 1   11 100 010   | A <- SRAM
  00101100 1000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101100 1001 *  = 0 1111 11 1   11 100 011   | B <- SRAM
  00101100 1010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101100 1011 c  = 0 1001 c0 1   00 010 100   | SRAM <- A+B+c, Latch CCs
  00101100 1100 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM;  ADR+

| cadd2(cxlo, ylo, zlo, cxhi, yhi, zhi) -- 16-bit add.
  00101101 0000 *  = 1 1111 11 1   11 001 010   | A <- uROM; ADR+
  00101101 0001 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101101 0010 *  = 0 1111 11 1   11 100 011   | B <- SRAM
  00101101 0011 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101101 0100 *  = 0 1001 10 1   00 010 100   | SRAM <- A+B, Latch CCs
  00101101 0101 *  = 1 1111 11 1   01 001 010   | A <- uROM; ADR+; Shift CCs
  00101101 0110 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101101 0111 *  = 0 1111 11 1   11 100 011   | B <- SRAM
  00101101 1000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101101 1001 c  = 0 1001 c0 1   00 010 100   | SRAM <- A+B+c, Latch CCs
  00101101 1010 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM;  ADR+
```

```
| sub2(xlo, ylo, zlo, xhi, yhi, zhi) -- 16-bit subtract.
  00101110 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101110 0001 *  = 0 1111 11 1   11 100 010   | A <- SRAM
  00101110 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101110 0011 *  = 0 1111 11 1   11 100 011   | B <- SRAM
  00101110 0100 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101110 0101 *  = 0 0110 00 1   00 010 100   | SRAM <- A-B, Latch CCs
  00101110 0110 *  = 1 1111 11 1   01 001 101   | MAR <- uROM; ADR+; Shift CCs
  00101110 0111 *  = 0 1111 11 1   11 100 010   | A <- SRAM
  00101110 1000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101110 1001 *  = 0 1111 11 1   11 100 011   | B <- SRAM
  00101110 1010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101110 1011 c  = 0 0110 c0 1   00 010 100   | SRAM <- A-B-c, Latch CCs
  00101110 1100 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM;  ADR+

| cmp2(xlo, ylo, xhi, yhi) -- 16-bit compare.
| This operation produces no result other than in the condition codes.
| After the cmp(x,y) the significance of the condition codes is:
|        E:     1 iff <x> = <y>
|       -C:     0 iff <x> > <y> (unsigned) ... i.e. is asserted active low
  00101111 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101111 0001 *  = 0 1111 11 1   11 100 010   | A <- SRAM
  00101111 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101111 0011 *  = 0 1111 11 1   11 100 011   | B <- SRAM

  00101111 0100 *  = 0 0110 10 1   00 010 010   | A <- A-B-1, Latch CCs
  00101111 0101 *  = 1 1111 11 1   01 001 101   | MAR <- uROM; ADR+; Shift CCs
  00101111 0110 *  = 0 1111 11 1   11 100 010   | A <- SRAM
  00101111 0111 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00101111 1000 *  = 0 1111 11 1   11 100 011   | B <- SRAM

  00101111 1001 c  = 0 0110 c0 1   00 010 010   | A <- A-B-c, Latch CCs
  00101111 1010 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM;  ADR+

| count(x) -- counts down <x>, setting flags.
| Doesn't disturb <B>, hence lights.
  00110000 0000 * =  1 1111 11 1   11 001 101   | MAR <- ROM; ADR+
  00110000 0001 * =  0 1111 11 1   11 100 010   | A   <- SRAM
  00110000 0010 * =  0 1111 10 1   00 010 100   | SRAM <- A-1; Latch CCs
  00110000 0011 * =  1 1111 11 1   11 001 000   | OP  <- UROM; ADR+

| cond(x) -- copies condition register into SRAM<x>.
|                                  .---------------------------------------.
|        CC Format:   ctl ROM <-   |  N | -C |  E | D0 | P0 | P1 | I/O | ?? |
|                                  `---------------------------------------'
|                                     <---   Direction of shift   <---

  00110001 0*** 0 =  0 1100 10 1   01 010 010   | A   <- A+A (Shift in 0)
  00110001 0*** 1 =  0 1100 00 1   01 010 010   | A   <- A+A+1 (Shift in 1)
  00110001 1000 * =  1 1111 11 1   11 001 101   | MAR <- ROM; ADR+
  00110001 1001 * =  0 1111 11 1   11 010 100   | SRAM<x> <- A
  00110001 1010 * =  1 1111 11 1   11 001 000   | OP  <- UROM; ADR+

| negcy(x, y) - negate with carry input.
  00110010 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00110010 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  00110010 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00110010 0011 *  = 0 0000 11 1   01 010 100   | A   <- not A; shift CCs
  00110010 0100 c  = 0 0000 c0 1   00 010 100   | SRAM <- A+c; Latch CCs
  00110010 0101 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| subcy(x, y, z) --  z <- x - y with carry input.
  00110011 0000 *  = 1 1111 11 1   01 001 101   | MAR <- uROM; ADR+; Shift CCs
  00110011 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  00110011 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00110011 0011 *  = 0 1111 11 1   11 100 011   | B   <- SRAM
  00110011 0100 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  00110011 0101 c  = 0 0110 c0 1   00 010 100   | SRAM <- A-B-1+C, Latch CCs
  00110011 0110 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| readio(r,y) --  y <- [r]  where [N] means "contents of I/O internal reg N"
  00110100 0000 *  = 1 1111 11 1   11 001 010   | A, I/O ADR <- uROM; ADR+
  00110100 0001 *  = 1 1111 11 1   11 001 101   | MAR  <- uROM; ADR+
  00110100 0010 *  = 0 1111 11 1   11 111 100   | SRAM <- I/O
  00110100 0011 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| ireadio(x,y) --  y <- [<x>]  where [N] means "contents of I/O internal reg N"
  00110101 0000 *  = 1 1111 11 1   11 001 101   | MAR  <- uROM; ADR+
  00110101 0001 *  = 0 1111 11 1   11 100 010   | A, I/O ADR <- SRAM
  00110101 0010 *  = 1 1111 11 1   11 001 101   | MAR  <- uROM; ADR+
  00110101 0011 *  = 0 1111 11 1   11 111 100   | SRAM <- I/O
  00110101 0100 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| writeio(x,r) --  io r <- <x>      (i.e., <x> into I/O reg referenced by y)
| >>>>> NOTE  <<<<<  ARGUMENTS ARE REVERSED BY MACRO (see macros.uasm)
  00110110 0000 *  = 1 1111 11 1   11 001 010   | A, I/O ADR <- uROM;ADR+(r)
  00110110 0001 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+      (x)
```

```
  00110110 0010 *  = 0 1111 11 1   11 100 111   | I/O <- SRAM
  00110110 0011 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| writeioi(x,y) --  io <y> <- <x>      (i.e., <x> into I/O reg referenced by y)
| >>>>> NOTE  <<<<<  ARGUMENTS ARE REVERSED BY MACRO (see macros.uasm)
  00110111 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+     (y)
  00110111 0001 *  = 0 1111 11 1   11 100 010   | A, I/O ADR <- SRAM
  00110111 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+     (x)
  00110111 0011 *  = 0 1111 11 1   11 100 111   | I/O <- SRAM
  00110111 0100 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+


|>>>>>>>>>>>>>>>>>>>>>
|>>> UNUSED OPCODES <<<  Opcodes  0011100 - 01111111  are NOT currently used.
|>>>>>>>>>>>>>>>>>>>>>                                  ===


| dispatch(x, tabhi, tablo)
| Takes (tabhi, tablo) as the address in uROM of a dispatch table,
|   whose consecutive entries are each 2-byte uROM addresses stored
|   (low, high).  Uses contents of SRAM register x as an index into
|   this table, and jumps to the indexed entry.  Note that the
|   uROM address of the entry is adr+x+x, since entries are 2 bytes
|   each; phases 0 thru 1011 compute this value and load it into
|   ADR.  Note use of SRAM location beyond stack as a temporary.
|
| NB:  nonstandard order of (tabhi, tablo) arguments
  10000000 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM (= x); ADR+
  10000000 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM (= <x>)
  10000000 0010 *  = 0 1100 11 1   11 010 101   | MAR <- 0xFF
  10000000 0011 *  = 0 1111 11 1   11 100 101   | MAR <- SRAM (= <0xFF>)
  10000000 0100 *  = 0 1100 10 1   00 010 011   | B <- A+A (=<x>+<x>);Latch CCs
  10000000 0101 *  = 1 1111 11 1   01 001 010   | A <- uROM; ADR+;Shift (tabhi)
  10000000 0110 c  = 0 0000 c0 1   11 010 100   | SRAM <- A+Carry    (tabhi+C1)
  10000000 0111 *  = 1 1111 11 1   11 001 010   | A   <- uROM; ADR+     (tablo)
  10000000 1000 *  = 0 1001 10 1   00 010 011   | B <- A+B;Latch CCs(x+x+tablo)
  10000000 1001 *  = 0 1111 11 1   01 100 010   | A <- SRAM;Shift CCs(tabhi+C1)
  10000000 1010 c  = 0 0000 c0 1   11 010 001   | ADR <- A+Carry  (tabhi+C1+C2)
  10000000 1011 *  = 0 1010 11 1   11 010 001   | ADR <- B          (x+x+tablo)
  10000000 1100 *  = 1 1111 11 1   11 001 010   | A   <- uROM; ADR+
  10000000 1101 *  = 0 1111 11 1   11 001 001   | ADR <- uROM
  10000000 1110 *  = 0 1111 11 1   11 010 001   | ADR <- A
  10000000 1111 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM; ADR+

| addcy(x, y, z)
| Like add(x, y, z), but adds in the incoming carry bit.
  10000001 0000 *  = 1 1111 11 1   01 001 101   | MAR <- uROM; ADR+; Shift CCs
  10000001 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  10000001 0010 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  10000001 0011 *  = 0 1111 11 1   11 100 011   | B   <- SRAM
  10000001 0100 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  10000001 0101 c  = 0 1001 c0 1   00 010 100   | SRAM <- A+B+Carry, Latch CCs
  10000001 0110 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM;  ADR+

| caddcy(cx, y, z)
| Like cadd, but adds in the incoming carry bit.
  10000010 0000 *  = 1 1111 11 1   01 001 010   | A   <- uROM; ADR+ Shift CCs
  10000010 0001 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  10000010 0010 *  = 0 1111 11 1   11 100 011   | B   <- SRAM
  10000010 0011 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  10000010 0100 c  = 0 1001 c0 1   00 010 100   | SRAM <- A+B+Carry, Latch CCs
  10000010 0101 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM;  ADR+

| cand(cx, y, z)
  10000011 0000 *  = 1 1111 11 1   11 001 010   | A   <- uROM; ADR+
  10000011 0001 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  10000011 0010 *  = 0 1111 11 1   11 100 011   | B   <- SRAM
  10000011 0011 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  10000011 0100 *  = 0 1011 11 1   00 010 100   | SRAM <- A&B, Latch CCs
  10000011 0101 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM;  ADR+

|>>>>>>>>>>>>>>>>>>>>>
|>>> UNUSED OPCODE  <<<  Opcode  10001000  is NOT currently used.
|>>>>>>>>>>>>>>>>>>>>>                        ===

| rotl1(x, y) -- left rotate x 1 position.
  10001001 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  10001001 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  10001001 0010 *  = 1 1111 11 1   00 001 101   | MAR <- uROM; ADR+; Latch CCs

  10001001 0011 0  = 0 1100 10 1   00 010 100   | SRAM <- A+A+C; Latch CCs
  10001001 0011 1  = 0 1100 00 1   00 010 100   | SRAM <- A+A+C; Latch CCs

  10001001 0100 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM;  ADR+

| rotl2(x, y) -- left rotate x 2 positions.
  10001010 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
```

```
  10001010 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  10001010 0010 *  = 1 1111 11 1   00 001 101   | MAR <- uROM; ADR+; Latch CCs

  10001010 0011 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001010 0100 0  = 0 1100 10 1   00 010 100   | SRAM <- A+A+C; Latch CCs

  10001010 0011 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001010 0100 1  = 0 1100 00 1   00 010 100   | SRAM <- A+A+C; Latch CCs

  10001010 0101 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM;  ADR+

| rotl3(x, y) -- left rotate x 3 positions.
  10001011 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  10001011 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  10001011 0010 *  = 1 1111 11 1   00 001 101   | MAR <- uROM; ADR+; Latch CCs

  10001011 0011 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001011 0100 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001011 0101 0  = 0 1100 10 1   00 010 100   | SRAM <- A+A+C; Latch CCs

  10001011 0011 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001011 0100 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001011 0101 1  = 0 1100 00 1   00 010 100   | SRAM <- A+A+C; Latch CCs

  10001011 0110 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM;  ADR+

| rotl4(x, y) -- left rotate x 4 positions.
  10001100 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  10001100 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  10001100 0010 *  = 1 1111 11 1   00 001 101   | MAR <- uROM; ADR+; Latch CCs

  10001100 0011 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001100 0100 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001100 0101 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001100 0110 0  = 0 1100 10 1   00 010 100   | SRAM <- A+A+C; Latch CCs

  10001100 0011 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001100 0100 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001100 0101 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001100 0110 1  = 0 1100 00 1   00 010 100   | SRAM <- A+A+C; Latch CCs

  10001100 0111 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM;  ADR+

| rotl5(x, y) -- left rotate x 5 positions.
  10001101 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  10001101 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  10001101 0010 *  = 1 1111 11 1   00 001 101   | MAR <- uROM; ADR+; Latch CCs

  10001101 0011 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001101 0100 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001101 0110 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001101 0101 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001101 0111 0  = 0 1100 10 1   00 010 100   | SRAM <- A+A+C; Latch CCs

  10001101 0011 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001101 0100 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001101 0101 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001101 0110 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001101 0111 1  = 0 1100 00 1   00 010 100   | SRAM <- A+A+C; Latch CCs

  10001101 1000 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM;  ADR+

| rotl6(x, y) -- left rotate x 6 positions.
  10001110 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  10001110 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  10001110 0010 *  = 1 1111 11 1   00 001 101   | MAR <- uROM; ADR+; Latch CCs

  10001110 0011 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001110 0100 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001110 0101 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001110 0110 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001110 0111 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001110 1000 0  = 0 1100 10 1   00 010 100   | SRAM <- A+A+C; Latch CCs

  10001110 0011 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001110 0100 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001110 0101 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001110 0110 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001110 0111 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001110 1000 1  = 0 1100 00 1   00 010 100   | SRAM <- A+A+C; Latch CCs

  10001110 1001 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM;  ADR+

| rotl7(x, y) -- left rotate x 7 positions.
  10001111 0000 *  = 1 1111 11 1   11 001 101   | MAR <- uROM; ADR+
  10001111 0001 *  = 0 1111 11 1   11 100 010   | A   <- SRAM
  10001111 0010 *  = 1 1111 11 1   00 001 101   | MAR <- uROM; ADR+; Latch CCs
```

```
  | Following states shift in sign bit (remember, Cin is asserted low)
  10001111 0011 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001111 0100 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001111 0101 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001111 0110 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001111 0111 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001111 1000 0  = 0 1100 10 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001111 1001 0  = 0 1100 10 1   00 010 100   | SRAM <- A+A+C; Latch CCs

  10001111 0011 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001111 0100 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001111 0101 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001111 0110 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001111 0111 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001111 1000 1  = 0 1100 00 1   00 010 010   | A    <- A+A+C; Latch CCs
  10001111 1001 1  = 0 1100 00 1   00 010 100   | SRAM <- A+A+C; Latch CCs

  10001111 1010 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM;  ADR+

|>>>>>>>>>>>>>>>>>>>>>
|>>> UNUSED OPCODES <<<  Opcodes  10001111 - 11111110  are NOT currently used.
|>>>>>>>>>>>>>>>>>>>>>                                    ===

| bpt() --  a do-nothing opcode which has been put in to allow explicit
|   breakpoints in the simulator.
  11111111 0000 *  = 1 1111 11 1   11 001 000   | OPCODE <- uROM;  ADR+
```

```
| Definitions of macros corresponding to ctlrom opcodes for MAYBE II.

| Reserved static RAM locations:

uSP     = 0xFF                      | Microstack pointer.
uRC     = 0xFE                      | Refresh counter.

_DEBUG  = 0xFD                      | Nonzero iff debugging under simulator.
_RESERV = 0xFC                      | Reserved for future uses.

| These SRAM locations are reserved for temporary use by microinstruction
|   macros:

T3      = 0xFB                      | 4 bytes; T0 is lowest address.
T2      = T3-1
T1      = T2-1
T0      = T1-1

SBASE   = T0-1                      | Base of microstack (builds down).

.macro WORD(x) x%256 x/256          | Low-byte followed by high-byte.
.macro LONG(x) WORD(x) WORD(x >> 16) | Low-word followed by high-word.

| Opcode 0 not used for now (reserved for an I/O hack opcode later).

.macro call(s)                  0x01 (.+4)%256 (.+3)/256 WORD(s)
.macro rtn()                    0x02
.macro push(x)                  0x03 x
.macro pop(x)                   0x04 x
.macro move(x,y)                0x05 x y
.macro cmove(cx,y)              0x06 cx y
.macro swt(x)                   0x07 x
.macro pdisp(x,p00,p01,p10,p11) 0x08 x WORD(p00) WORD(p01) WORD(p10) WORD(p11)
.macro pwrup()                  0x09
.macro add(x,y,z)               0x0A x y z
.macro cadd(cx,y,z)             0x0B cx y z
.macro jmp(adr)                 0x0C WORD(adr)
.macro imove(x,y)               0x0D x y
.macro movei(x,y)               0x0E x y

| Copy n-byte SRAM values:

.macro move2(x,y)   move(x,y)   move(x+1,y+1)
.macro move4(x,y)   move2(x,y)  move2(x+2,y+2)

.macro cmove2(cx,y) cmove((cx)%256,y) cmove((cx)/256,y+1)
.macro cmove4(cx,y) cmove2(cx,y)      cmove2(cx >> 16,y+2)

.macro l(adrlo,adrhi,x)         0x0F adrlo adrhi x
.macro s(x,adrlo,adrhi)         0x10 x adrlo adrhi
.macro refr()                   0x11

.macro jmi(x)                   0x12 WORD(x)
.macro jextra(x)                0x13 WORD(x)
.macro jready(x)                0x14 WORD(x)
.macro jp1(x)                   0x15 WORD(x)
.macro jodd(x)                  0x16 WORD(x)
.macro jnc(x)                   0x17 WORD(x)
.macro jp0(x)                   0x18 WORD(x)
.macro je(x)                    0x19 WORD(x)

| Opcodes 0x1A and 0x1B are not used.

.macro sub(x,y,z)               0x1C x y z
.macro cmp(x,y)                 0x1D x y

| Following macros demonstrate an alternative to composing countless
|   nanoinstructions. All could have been written as explicit nanocodes,
|   but at the cost of 1 opcode each and much effort in pounding out
|   the nanobits. Note, however, that each of these expands into
|   2 nanoinstruction calls. You have to pay for your laziness.

.macro ccmp(cx,y)     cmove(cx,T0)   cmp(T0,y)    | Constant compare.
.macro cor(cx,y,z)    cmove(cx,T0)    or(T0,y,z) | Constant OR (inclusive).
.macro cxor(cx,y,z)   cmove(cx,T0)   xor(T0,y,z) | Constant XOR.
.macro csub(x,cy,z)   cmove(cy,T0)   sub(x,T0,z) | Constant subtract.
.macro csubcy(x,cy,z) cmove(cy,T0) subcy(x,T0,z) | Constact sub w/borrow.

.macro jpl(a)                   0x1E WORD(a)
.macro jnextra(a)               0x1F WORD(a)    | Extra condition bit.
.macro jnready(a)               0x20 WORD(a)
```

```
.macro jnp1(a)                  0x21 WORD(a)
.macro jeven(a)                 0x22 WORD(a)
.macro jc(a)                    0x23 WORD(a)
.macro jnp0(a)                  0x24 WORD(a)
.macro jne(a)                   0x25 WORD(a)
.macro urom(l,h,dst)            0x26 l h dst
.macro and(x,y,z)               0x27 x y z
.macro or(x,y,z)                0x28 x y z
.macro xor(x,y,z)               0x29 x y z
.macro not(x,y)                 0x2A x y
.macro neg(x,y)                 0x2B x y

| 16-bit operations. Operands use 2 consecutive SRAM locations.

.macro add2(x,y,z)              0x2C x y z x+1 y+1 z+1
.macro cadd2(cx,y,z)            0x2D cx%256 y z cx/256 y+1 z+1
.macro sub2(x,y,z)              0x2E x y z x+1 y+1 z+1
.macro cmp2(x,y)                0x2F x y   x+1 y+1

.macro ccmp2(cx,y) cmove2(cx,T0) cmp2(T0,y)

.macro count(x)                 0x30 x
.macro cond(x)                  0x31 x

| Opcodes 0x32 & 0x33 are negcy & subcy(x,y,z) below (for no good reason).

| Opcodes 0x34 - 0x39 implement the I/O functions
|.macro iowrite(x)              0x34 x
|.macro ioread(x)               0x35 x
.macro iowrite(x)               writeio(x, 0)
.macro ioread(x)                readio(0, x)
.macro cwriteio(cx, r)          cmove(cx,T0) writeio(T0, r)
.macro readio(r, x)             0x34 r x
.macro ireadio(x, y)            0x35 x y
.macro writeio(x, r)            0x36 r x | NOTE ARG REVERSAL [see ctlrom]
.macro writeioi(x, y)           0x37 y x | NOTE ARG REVERSAL [see ctlrom]

| Opcodes 0x37 - 0x7F not used (for no good reason).

.macro dispatch(x,adr)          0x80 x adr/256 adr%256
.macro addcy(x,y,z)             0x81 x y z
.macro negcy(x, y)              0x32 x y
.macro subcy(x,y,z)             0x33 x y z
.macro caddcy(cx,y,z)           0x82 cx y z
.macro cand(cx,y,z)             0x83 cx y z

| Opcodes 0x84 - 0x88 not used (for no good reason).

.macro rotr7(x,y)               0x89 x y        | Right rotations.
.macro rotr6(x,y)               0x8A x y
.macro rotr5(x,y)               0x8B x y
.macro rotr4(x,y)               0x8C x y
.macro rotr3(x,y)               0x8D x y
.macro rotr2(x,y)               0x8E x y
.macro rotr1(x,y)               0x8F x y

.macro rotl7(x,y)               0x8F x y        | Left rotations.
.macro rotl6(x,y)               0x8E x y
.macro rotl5(x,y)               0x8D x y
.macro rotl4(x,y)               0x8C x y
.macro rotl3(x,y)               0x8B x y
.macro rotl2(x,y)               0x8A x y
.macro rotl1(x,y)               0x89 x y

| Following added for debugging under the simulator:

.macro bpt() 0xFF               | Breakpoint.
```

Control inputs				Function performed if $M = 0$ (arithmetic)	Function performed if $M = 1$ (logical)
F_3	F_2	F_1	F_0		
0	0	0	0	$A + 1 - \overline{C}$	$\overline{A}$
0	0	0	1	$(A \text{ OR } B) + 1 - \overline{C}$	$\overline{(A \text{ OR } B)}$
0	0	1	0	$(A \text{ OR } \overline{B}) + 1 - \overline{C}$	$\overline{A} \text{ AND } B$
0	0	1	1	$-\overline{C}$	00000000
0	1	0	0	$A + (A \text{ AND } \overline{B}) + 1 - \overline{C}$	$\overline{(A \text{ AND } B)}$
0	1	0	1	$(A \text{ OR } B) + (A \text{ AND } \overline{B}) + 1 - \overline{C}$	$\overline{B}$
0	1	1	0	$A - B - \overline{C}$	$A \text{ XOR } B$
0	1	1	1	$(A \text{ AND } \overline{B}) - \overline{C}$	$A \text{ AND } \overline{B}$
1	0	0	0	$A + (A \text{ AND } B) + 1 - \overline{C}$	$\overline{A} \text{ OR } B$
1	0	0	1	$A + B + 1 - \overline{C}$	$\overline{(A \text{ XOR } B)}$
1	0	1	0	$(A \text{ OR } B) + (A \text{ AND } B) + 1 - \overline{C}$	B
1	0	1	1	$(A \text{ AND } B) - \overline{C}$	$A \text{ AND } B$
1	1	0	0	$A + A + 1 - \overline{C}$	11111111
1	1	0	1	$(A \text{ OR } B) + A + 1 - \overline{C}$	$A \text{ OR } \overline{B}$
1	1	1	0	$(A \text{ OR } \overline{B}) + A + 1 - \overline{C}$	$A \text{ OR } B$
1	1	1	1	$A - \overline{C}$	A

Control inputs						Output
F_3	F_2	F_1	F_0	$\overline{C}$	M	
0	0	1	1	1	1	00000000
1	1	0	0	1	1	11111111
1	1	1	1	1	1	A
1	0	1	0	1	1	B
1	1	1	1	1	0	$A - 1$
0	0	0	0	0	0	$A + 1$
1	0	0	1	1	0	$A + B$
0	1	1	0	0	0	$A - B$
1	0	1	1	1	1	A AND B
1	1	1	0	1	1	A OR B
0	1	1	0	1	1	A XOR B
0	0	0	0	1	1	$\overline{A}$ (1's complement)

Address increment	ALU inputs	N.C.	COND S/L	Drive select	Load select
$ADR+$	F_3 F_2 F_1 F_0 $\overline{C}$ M	X	I S	D_2 D_1 D_0	L_2 L_1 L_0

Control ROM output word

DRSEL			Source
D_2	D_1	D_0	
0	0	0	Switch Register
0	0	1	Microcode ROM
0	1	0	ALU output
0	1	1	Not connected
1	0	0	Static RAM
1	0	1	Not connected
1	1	0	Dynamic RAM output register
1	1	1	Input/output data

Drive Select Signals

LDSEL			Destination
L_2	L_1	L_0	
0	0	0	OP (Opcode) register
0	0	1	Microcode ROM ADR
0	1	0	ALU A input and I/O address
0	1	1	ALU B input
1	0	0	Static RAM data
1	0	1	Static RAM address (MAR)
1	1	0	Dynamic RAM
1	1	1	Input/output data

Load Select Signals

Inputs		Action
I	S	
0	0	Load condition register
0	1	Shift condition register right
1	0	No change
1	1	No change

Condition shift register control inputs

	External Communication	Push buttons	ALU output conditions bits
C_7	C_6	C_5 C_4	C_3 C_2 C_1 C_0
N.C.	I/OFLAG	P_1 P_0	D_0 E $\overline{C}$ N

Condition Register

נספח ה' - פקודות הMAYBE

פקודות שימושיות בMAYBE:

Symbolic microinstruction	Micro-ROM encoding	Operation performed
move(x, y)	0x05 x y	$y \leftarrow \langle x \rangle$
cmove(cx, y)	0x06 cx y	$y \leftarrow cx$
add(x, y, z)	0x0A x y z	$z \leftarrow \langle x \rangle + \langle y \rangle$
cadd(cx, y, z)	0x0B cx y z	$z \leftarrow cx + \langle y \rangle$
add2(x, y, z)	0x2C x y z x+1, y+1, z+1	$z^2 \leftarrow \langle x \rangle^2 + \langle y \rangle^2$
cadd2(cx, y, z)	0x2D cx%256 y z cx/256 y+1 z+1	$z^2 \leftarrow cx^2 + \langle y \rangle^2$
sub(x, y, z)	0x1C x y z	$z \leftarrow \langle x \rangle - \langle y \rangle$
cmp(x, y)	0x1D x y	$\langle x \rangle - \langle y \rangle$
and(x, y, z)	0x27 x y z	$z \leftarrow \langle x \rangle$ AND $\langle y \rangle$
or(x, y, z)	0x28 x y z	$z \leftarrow \langle x \rangle$ OR $\langle y \rangle$
xor(x, y, z)	0x29 x y z	$z \leftarrow \langle x \rangle$ XOR $\langle y \rangle$
not(x, y)	0x2A x y	$y \leftarrow$ NOT $\langle x \rangle$ (1's complemet)
neg(x,y)	0x2B x y	$y \leftarrow \left(-\langle x \rangle\right)$ (2's complemet)

פקודות קפיצה:

Symbolic microinstruction	Control-ROM encoding	Operation performed
`jmp(`*dst*`)`	0x0C *dstlo dsthi*	Unconditional transfer: ADR $\leftarrow$ (*dsthi*, *dstlo*)
`je(`*dst*`)`	0x19 *dstlo dsthi*	`jmp` if ALU output was = 0b11111111
`jne(`*dst*`)`	0x25 *dstlo dsthi*	`jmp` if ALU output was $\neq$ 0b11111111
`jmi(`*dst*`)`	0x12 *dstlo dsthi*	`jmp` if ALU output was < 0
`jc(`*dst*`)`	0x23 *dstlo dsthi*	`jmp` if carry
`jnc(`*dst*`)`	0x17 *dstlo dsthi*	`jmp` if no carry
`jeven(`*dst*`)`	0x22 *dstlo dsthi*	`jmp` if ALU output was even
`jodd(`*dst*`)`	0x16 *dstlo dsthi*	`jmp` if ALU output was odd
`jready(`*dst*`)`	0x14 *dstlo dsthi*	`jmp` if I/OFLAG is set
`jp0(`*dst*`)`	0x18 *dstlo dsthi*	`jmp` if P_0 is pressed
`jp1(`*dst*`)`	0x15 *dstlo dsthi*	`jmp` if P_1 is pressed
`jnp0(`*dst*`)`	0x24 *dstlo dsthi*	`jmp` if P_0 is not pressed
`jnp1(`*dst*`)`	0x21 *dstlo dsthi*	`jmp` if P_1 is not pressed

פקודות הקשורות לעבודה עם פונקציות:

Symbolic microinstruction	Control-ROM encoding	Operation performed
push(x)	0x03 x	Push SRAM $\langle x \rangle$ onto microstack: $\langle \mu SP \rangle \leftarrow \langle x \rangle; \mu SP \leftarrow \langle \mu SP \rangle - 1$
pop(x)	0x04 x	Pop mictostack into SRAM $\langle x \rangle$: $\mu SP \leftarrow \langle \mu SP \rangle + 1; x \leftarrow \langle\langle \mu SP \rangle\rangle$
call(s)	0x01 rlo rhi slo shi	Mictosubroutine call to microcode ROM address s; r is the return location: $\langle \mu SP \rangle \leftarrow rlo; \mu SP \leftarrow \langle \mu SP \rangle - 1;$ $\langle \mu SP \rangle \leftarrow rhi; \mu SP \leftarrow \langle \mu SP \rangle - 1;$ $ADR \leftarrow shi; ADR \leftarrow slo$
rtn()	0x02	Microsubroutine return: $\mu SP \leftarrow \langle \mu SP \rangle + 1; x \leftarrow \langle\langle \mu SP \rangle\rangle;$ $\mu SP \leftarrow \langle \mu SP \rangle + 1; x \leftarrow \langle\langle \mu SP \rangle\rangle$

:dispatch

Symbolic microinstruction	Control-ROM encoding	Operation performed
dispatch(x, tab)	0x80 x tabhi tablo	*tab* is taken as the μROM address of a table of 2-byte μROM addresses, each stored (low, high); the SRAM location *x* contains an index into the table; jumps to the μROM location given in the indexed entry.

העברת נתונים בין תאים בSRAM

Symbolic microinstruction	Control-ROM encoding	Operation performed
move(x, y)	0x05 x y	$y \leftarrow \langle x \rangle$
cmove(cx, y)	0x06 cx y	$y \leftarrow cx$
imove(x, y)	0x0D x y	$y \leftarrow \langle\langle x \rangle\rangle$
movei(x, y)	0x0E x y	$\langle y \rangle \leftarrow \langle x \rangle$

גישה לDRAM:

Symbolic microinstruction	Control-ROM encoding	Operation performed
l(adrlo, adrhi, where)	0x0F adrlo adrhi where	where $\leftarrow \langle\langle \text{adrlo} \rangle, \langle \text{adrhi} \rangle\rangle$; loads contents of DRAM location whose address is contained in SRAM locations *adrlo* and *adrhi* into SRAM location *where.*
s(where,adrlo,adrhi)	0x10 where adrlo adrhi	Stores contents of SRAM location *where* into DRAM location whose address is contained in SRAM locations *adrlo* and *adrhi*.
refr()	0x11	Refreshs six rows of dynamic RAM

נספח ו' - שיטות מיעון

שיטות מיעון כלליות:

Addressing mode	Instruction stream	Assembler syntax	Source datum V	Destination address E
Immediate	X	imm(X)	V=X	
Direct	X	dir(X)	$V=\langle X\rangle$	E=X
Register	R	r(R)	$V=\langle R\rangle$	E=R
Indirect	X	I(X)	$V=\langle\langle X\rangle\rangle$	$E=\langle X\rangle$
Indirect register	R	ir(R)	$V=\langle\langle R\rangle\rangle$	$E=\langle R\rangle$
Indexed	R, X	ix(X, R)	$V=\langle\langle R\rangle+X\rangle$	$E=\langle R\rangle+X$
Stack push		push()		E=SP; $SP\leftarrow\langle SP\rangle+\alpha$
Stack pop		pop()	$SP\leftarrow\langle SP\rangle-\alpha$; $V=\langle\langle SP\rangle\rangle$	
SP-relative	X	ix(X, SP)	$V=\langle\langle SP\rangle+X\rangle$	$E=\langle SP\rangle+X$

שיטות המיעון הבסיסיות של ה-G-Machine:

Addressing mode	M field	Effective Address	Assembler syntax
Register	0	E=Rn	r(n)
Indirect register	1	$E=\langle Rn\rangle$	ir(n)
Direct	2	E=X	dir(X)
Indirect	3	$E=\langle X\rangle$	i(X)
Indexed	4	$E=X+\langle Rn\rangle$	ix(X, n)
Indirect indexed	5	$E=\langle X+\langle Rn\rangle\rangle$	iix(X, n)
Postincrement	6	$E=\langle Rn\rangle; Rn\leftarrow\langle Rn\rangle+\alpha$	posti(n)
Indirect Postincrement	7	$E=\langle\langle Rn\rangle\rangle; Rn\leftarrow\langle Rn\rangle+\alpha$	iposti(n)
Predecrement	8	$Rn\leftarrow\langle Rn\rangle-\alpha$; $E=\langle Rn\rangle$	pred(n)
Indirect Predecrement	9	$Rn\leftarrow\langle Rn\rangle-\alpha$; $E=\langle\langle Rn\rangle\rangle$	ipred(n)

שיטות המיעון הנגזרות משיטות המיעון הבסיסיות:

Addressing mode	M field	General register	Effective Address	Assembler syntax
Immediate	6	PC	$E=\langle PC\rangle; PC \leftarrow \langle PC\rangle+\alpha$	$imm\alpha(X)$
SP-relative	4	SP	$E=X+\langle SP\rangle$	ix(X, SP)
Indirect SP-relative	5	SP	$E=\langle X+\langle SP\rangle\rangle$	iix(X, SP)
Stack (push)	6	SP	$E=\langle SP\rangle; SP \leftarrow \langle SP\rangle+\alpha$	push()
Stack (pop)	8	SP	$SP \leftarrow \langle SP\rangle-\alpha;\ E=\langle SP\rangle$	pop()
PC-relative	4	PC	$E=X+\langle PC\rangle$	rel(X)

נספח ז' - פקודות G-Machine

פקודות בסיסיות:

Operation	Operand addresses	Function performed
gmoveα	A, B	$E_B^\alpha \leftarrow \langle E_A \rangle^\alpha$
gaddα	A, B, C	$E_C^\alpha \leftarrow \langle E_A \rangle^\alpha + \langle E_B \rangle^\alpha$
gsubα	A, B, C	$E_C^\alpha \leftarrow \langle E_A \rangle^\alpha - \langle E_B \rangle^\alpha$
gmultα	A, B, C	$E_C^\alpha \leftarrow \langle E_A \rangle^\alpha \cdot \langle E_B \rangle^\alpha$
gdivα	A, B, C	$E_C^\alpha \leftarrow \langle E_A \rangle^\alpha \div \langle E_B \rangle^\alpha$
gremα	A, B, C	$E_C^\alpha \leftarrow \langle E_A \rangle^\alpha \ \text{mod} \ \langle E_B \rangle^\alpha$
gnegα	A, B	$E_B^\alpha \leftarrow -\langle E_A \rangle^\alpha$
gandα	A, B, C	$E_C^\alpha \leftarrow \langle E_A \rangle^\alpha \ \text{AND} \ \langle E_B \rangle^\alpha$
gorα	A, B, C	$E_C^\alpha \leftarrow \langle E_A \rangle^\alpha \ \text{OR} \ \langle E_B \rangle^\alpha$
gxorα	A, B, C	$E_C^\alpha \leftarrow \langle E_A \rangle^\alpha \ \text{XOR} \ \langle E_B \rangle^\alpha$
gcomα	A, B	$E_B^\alpha \leftarrow \text{NOT} \langle E_A \rangle^\alpha$
gashα	A, B, C	Arithmetically shift $\langle E_A \rangle^\alpha$ left (right) $\langle E_B \rangle^1$ $\left(\text{or } -\langle E_B \rangle^1\right)$ places. store in $\langle E_C \rangle^\alpha$
glshα	A, B, C	Logically shift $\langle E_A \rangle^\alpha$ left (right) $\langle E_B \rangle^1$ $\left(\text{or } -\langle E_B \rangle^1\right)$ places. store in $\langle E_C \rangle^\alpha$
ghalt		Stop the machine

פקודות לעבודה עם פונקציות:

Operation	Operand addresses	Function performed
gcall	A	$push4\left[\langle \text{PC} \rangle\right]$ $\text{PC}^4 \leftarrow \text{E}_\text{A}$
grtn		$\text{PC} \leftarrow \text{pop4}[\ ]$

פקודות קפיצה:

Opeation	Operand addresses	Sets CC?	Function performed
gcmpα	A, B	Yes	Set S, Z and C bits in CC according to the result of the operation $\langle E_A \rangle^{\alpha} - \langle E_B \rangle^{\alpha}$
gtestα	A	Yes	Set S, Z and C bits in CC according to the result of the operation $\langle E_A \rangle^{\alpha} - 0$
gjmp	A	No	$PC^4 \leftarrow EA^4$
gj*cond*	A	No	If *cond* matches $\langle CC \rangle$ then $PC^4 \leftarrow EA^4$

קפיצות מותנות נעשות על ידי המשפחה gj*cond*, כאשר cond הוא אחד מהתנאים המופיעים בטבלה הבאה:

Mnemonic	Branches on	Description
ne	$\overline{Z}$	Not equal (to zero) test
e	Z	Equal
ge	$\overline{S}$	Signed $\geq$
lt	S	Signed $<$
gt	$\overline{Z \ OR \ S}$	Signed $>$
le	$Z \ OR \ S$	Signed $\leq$
hi	C AND $\overline{Z}$	Higher (unsigned $>$)
los	$\overline{C}$ OR Z	Low or same (unsigned $\leq$)
his	C	Higher or same (unsigned $\geq$)
lo	$\overline{C}$	Lowe (unsigned $<$)