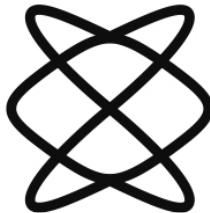


החוג למדעי המחשב (0368)
מבנה מחשבים (2159)
(קורס ארוכה)

מרצה: עמית ברמן
מתרגלים: אפיק טומי קרכען /ודן פריליאן
תשפ"ג, סמסטר ב' (2023)

מסכם: רועי מעין



The Raymond and
Beverly Sackler Faculty
of Exact Sciences
Tel Aviv University



פרק 1 - מבוא

3	מבוא
14	מעגלים
23	תיזמו

פרק 2 – ארכיטקטורת מחשב

34	MIPS ISA
41	MIPS Design
64	איכרונו

פרק 3 – שיפור ביצועים

76	Single Thread
83	שאלות מסכימות



1 - מבוא

מבוא

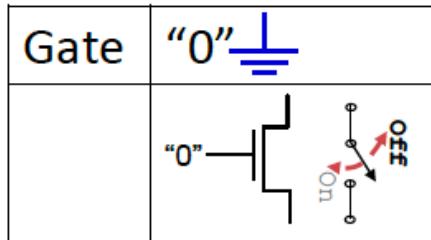
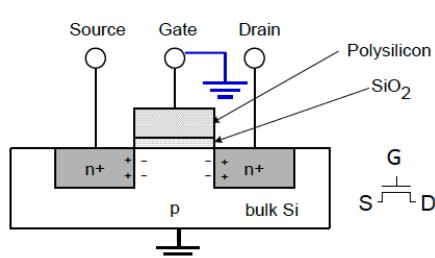
רוב השיעור הראשון היה בגדיר העשרה (אלקטרוניקה, מושגים בסיסיים בחישול ובפיזיקה). החלק שהועבר בסופו על רון כהן בקורס: ברמת מה הם עושים ואיך ה-*NOT* מורכב מהם, אין צורך לדעת איך הם עובדים מאחוריהם.

טרנזיסטור

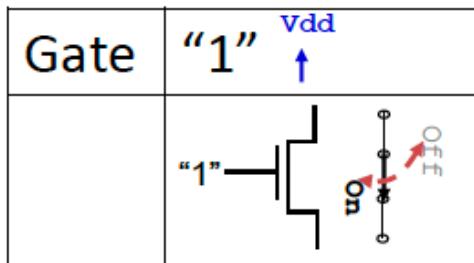
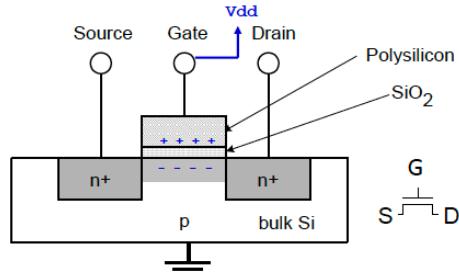
טרנזיסטור הוא רכיב שימושים בו בתפקיד **mpsych** (switch), ללא שום דבר מכני צריך לו זו בו, **הבול חשמלי** (אין צורך לדעת את כל הרקע באלקטרוניקה ובפיזיקה – זה היה בהרצאה הראשונה בגדיר העשרה, מאחורי הקלעים לא מעוניין, רק מה הוא עשה ואין מרכיבים בעודתו ורכיבים אחרים).

עבור טרנזיסטור MOS-N (P באמצעו ושני N מצדדיו):

- כאשר המפסק פתוח – אין זרימה, "0". זה קורה כאשר ה-*gate* מחובר לאדמה (0). נאמר שהטרנזיסטור מנותק (לא מעביר זרם בין ה-Source ל-Drain).



- כאשר המפסק סגור – יש זרימה, "1". זה קורה כאשר ה-*gate* מחובר ל-Vdd (1) (מתוך). נאמר שהטרנזיסטור מקוצר (יש מעבר חופשי במעט של מטען מה-Source ל-Drain).



וככל גם ליצור טרנזיסטור P-MOS (N באמצעו ושני P מצדדיו), ופה קורה בבדיקה ההפר:

- כאשר נתחבר לאדמה (0) – יש זרימה והmpsych סגור.
- כאשר נתחבר למתח (1) – אין זרימה והmpsych פתוח.

נסכם את ההבדלים (נשים לב ש-S-MOS מסומן עם עיגול קטן):

Gate	"0"	"1"
N-MOS		
P-MOS		

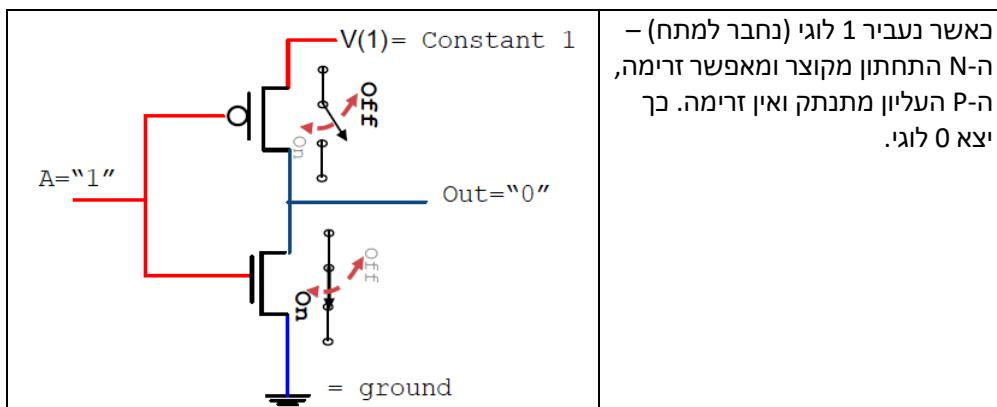
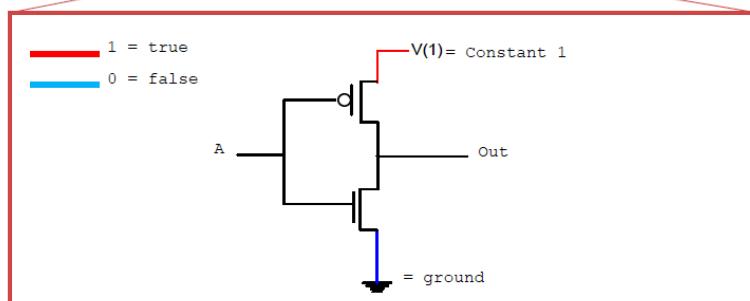
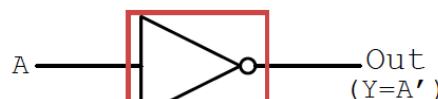
מעגלים פשוטים:

נוכל ליצור שורשת של מפסקים, כאשר הכל בה חשמלי והמפסקים משפיעים זה על זה. כך ניצור פונקציה לוגית מתרנויסטורים. לשפחota המעגלים שמוצרים על ידי סימטריה בין N-L-P נקרא CMOS, Complementary MOS, כי הם משלימים זה את זה).

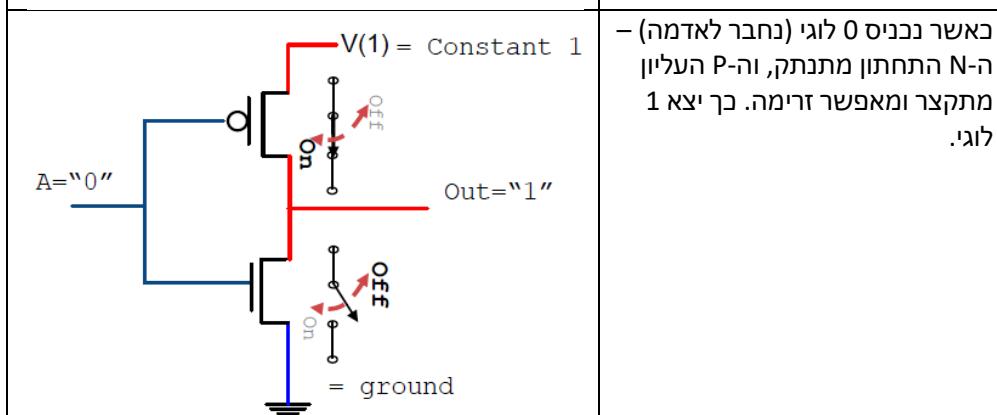
מחפר (Inverter): שער Not, הופך 0 ל-1 ו-1 ל-0. יש שני טרנזיסטורים, אחד P ואחד N. העיגול בשרטוט מסמן היפוך.

Inverter (“Not” Gate)

A	Y
0	1
1	0



כאשר נטען 1 לוגי (נחבר למתח) – ה-N התח桐ן מקוצר ומאפשר זרימה, ה-P העלון מתנתק ואין זרימה. כך יצא 0 לוגי.



כאשר נטען 0 לוגי (נחבר לאדמה) – ה-N התח桐ן מתנתק, וה-P העלון מתקצר ומאפשר זרימה. כך יצא 1 לוגי.

מעגלים לוגיים הם משורזרים, פلت של מעגל אחד הוא קלט של מעגל אחר. תמיד יש input-output. נשים לב שגם ה-P יאפשר זרם (עם הקלט 0) ונחבר את ה-outקע output לאדמה, יהיה פיצוץ כי תהיה זרימה מהסוללה לאדמה (ולא ל-out).



שערים לוגיים בסיסיים

$\text{Not}(x) = x'$	
$\text{And}(x, y) = x \cdot y$	
$\text{Or}(x, y) = x + y$	

שער NOT (NAND): בהתבסס על שער-h-AND, נוסיף עיגול קטן לשרטוט ונקבל את ה-NAND. פועלות זו מבחינה בוליאנית היא:

$$\text{NAND}(x, y) = (x \cdot y)'$$

בניה את השער בצורה הבאה:

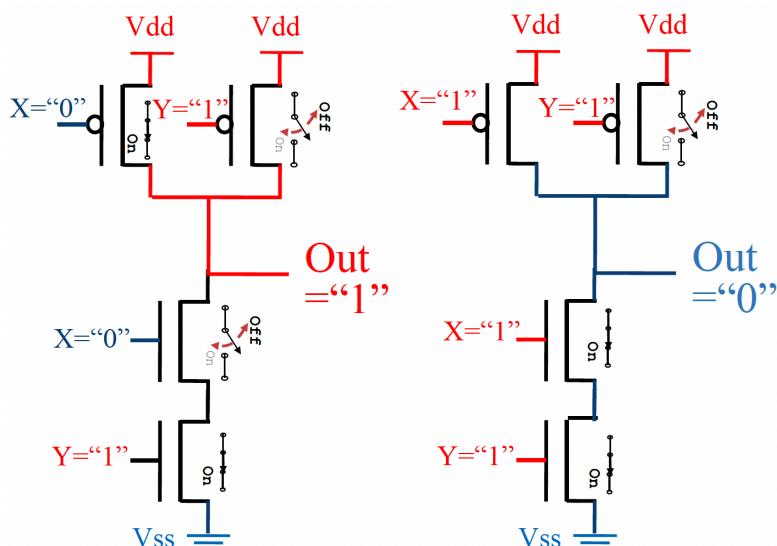
- P (מחובר למתח) – נחבר 2 במקביל (OR), מחוברים לקלט y, x .
- N (מחובר לאדמה) – נחבר 2 בטור (AND), גם הם מחוברים לקלט y, x . כאשר y הוא התיכון, $-x$ הוא העליון.

מצבים שונים:

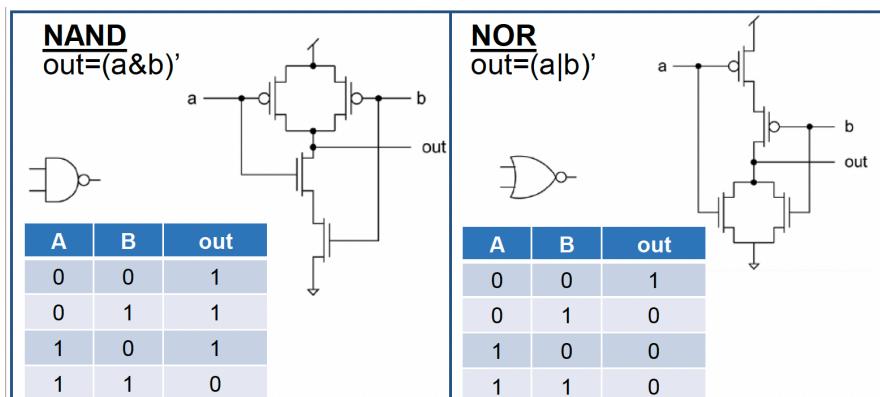
- $x = 0, y = 1$ – נקבל כי באחד הטרנזיסטורים של ה-P זורם זרם בגל הערך 0, הם פועלים במקביל ולכן יש כאן OR ומספריק אחד מהם קיבל זרם. נקבל ב- $\text{Out} = 1$.

- ב-N שמחובר לאדמה זורם זרם בתיכון ($1 = y$) אבל **בעליו** ($0 = x$) לא זורם, לכן סה"כ לא יזרום זרם. אין חיבור לאדמה. נישאר עם ה-1 שמנגיע מה- P וזה יהיה הפלט.

- $x = 1, y = 1$, נקבל כי ב-P אין זרם בשני הטרנזיסטורים, אין זרם מה- P , ולכן נקבל **0**. ב-N שני הטרנזיסטורים פועלים ולכן הוא בולע את כל הזרם לאדמה (זה המצב היחיד שבו ה-N מזרים זרם לאדמה).



שער NOR (Not OR): מימוש הפוך ל-NAND. את ה-P נחבר בטור, ואת ה-N במקביל.



◆ Rules for CMOS: (Complementary Metal-Oxide-Semiconductor)

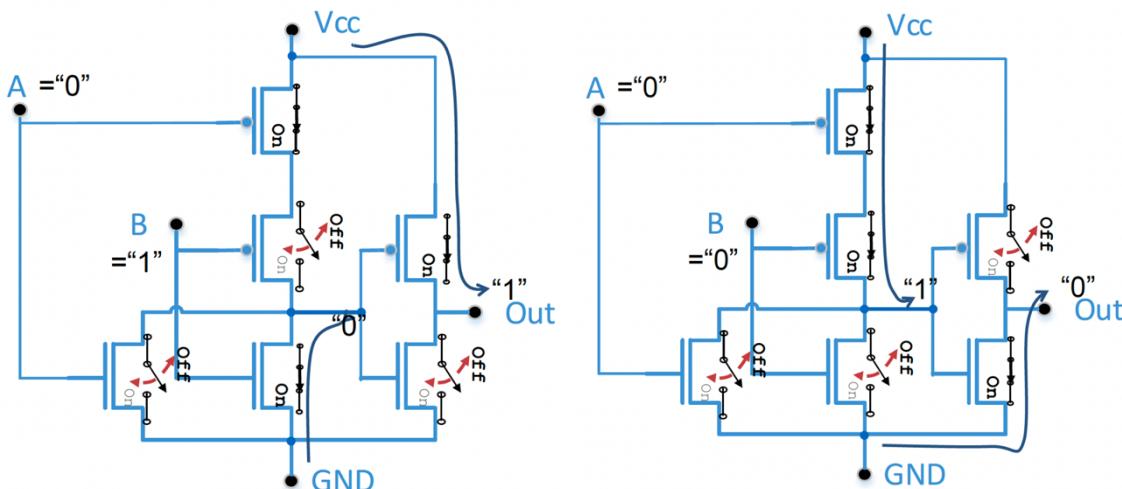
- PMOS and NMOS are always in pairs
- PMOS always to V(1), NMOS always to GND(0)

כעת נוכל למש את שערי AND ו-OR באמצעות NAND או NOR (4 טרנזיסטורים), בתוספת של NOT (2 טרנזיסטורים), וכך נקבל סה"כ מערכת של **6 טרנזיסטורים**.



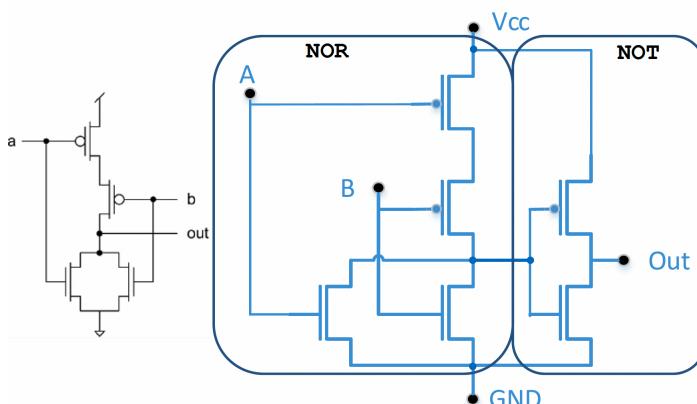
תרגיל – ניתן להוות מעגל לפי טבלה אמת (מעקב אחריו הזרמים):

- למשל עבור $A = 0, B = 0$: יש שני P שמחוברים בטור למתח, העליון מקבל 0 (זירימה), אבל ב-N כיוון שיש 2 במקביל אז יש זירימה – ה-N מחובר לאדמה ולכן 0. לאחר מכן, יש P שדרכו יש זירימה ו-N שדרכו אין זירימה, לכן סה"כ המתח יזרום דרך ה-P ונתקבל בפלט 1.
- עבור $A = 0, B = 1$: בשני ה-N אין זירימה, אבל בשני ה-P יש זירימה ולכן 1. לאחר מכן, דרך ה-P אין זירימה, ודרך ה-N כן יש זירימה, לכן סה"כ הפלט מוחבר לאדמה דרך ה-N ונתקבל בפלט 0.



או לפי זהוי תבניות מוכרכות:

$$OUT = \text{NOT}(\text{NOR}(A,B)) = \text{OR}(A,B)$$



יצוג מספרים (תרגול 1)

יצוג מספרים חיוביים בבסיסים שונים:

$$(a_4a_3a_2a_1a_0 \cdot a_{-1}a_{-2}a_{-3})_{10} =$$

$$a_4 \cdot 10^4 + a_3 \cdot 10^3 + a_2 \cdot 10^2 + a_1 \cdot 10^1 + a_0 \cdot 10^0 +$$

$$a_{-1} \cdot 10^{-1} + a_{-2} \cdot 10^{-2} + a_{-3} \cdot 10^{-3}$$

$$\sum_{i=-m}^n a_i r^i$$

שיטת ספירה – שיטה להציגת של מספרים באמצעות קבוצה נתונה של סימנים, הקרוים ספרות.

בסיס – תחום הספרות הייחודיות הנמצוא בסיסה של שיטת ספירה. הבסיס המקובל לספירה הוא בסיס 10 (עשרוני), עם הספרות 0 עד 9. עבור מספר כללי:

בסיסים שונים – בסיס 2 (Binary), בסיס 8 (Octal), בסיס 16 (Hexadecimal).

המחשב שלנו מורכב ממוגלים לוגיים אלקטרוניים, שם נוח להסתפק בהבחנה בין שתי רמות מתח בלבד, גובה ונמוך, המוצגת על ידי 1 ו-0 בהתאם. לכן הבסיס הטבעי לעבד אותו הוא הבסיס הבינארי.

פעולות חשבוניות:

- הכללים הנהוגים בפעולות החשבון לפי בסיס 2 זהים לאלו הנהוגים בבסיס 10.
- יש לשימוש לב להשתמש רק במספרות $\{ -r, \dots, 0, 1 \}$. למשל, במס' 4 מתקיים $10 = 1 + 3$.



纽带 בין בסיסים שונים:

- מעבר מבסיס z לבסיס 10: נתנו לנו מספר $a_{m-1} \dots a_1 a_0. a_{-1} a_{-2} \dots a_{-n}$ בסיס z . ערך המספר בסיס 10 הוא:

$$\sum_{i=-n}^m a_i r^i$$

$$(AD1.F)_{16} = 10 \cdot 16^2 + 13 \cdot 16^1 + 1 \cdot 16^0 + 15 \cdot 16^{-1}$$

$$= (2769.9375)_{10}$$

- מעבר מבסיס 10 לבסיס z : האלגוריתם הוא לאותל $0 = i$. כל עוד $0 > N$ נרשום $r \mod N$,nde נעדכן את $N = \left\lfloor \frac{N}{r} \right\rfloor$. ונקדם את $1 + i = i$. נוח לרשום בצורה טבלה את השאריות ואת השלמים ולבסוף לקבל את התוצאה.
לדוגמה, עבור $N = 59, r = 5$:

i	שארית	שלם
59		
11	4	0
2	1	1
0	2	2

- ונקבל: $(59)_{10} = (a_2 a_1 a_0)_5 = (214)_{10} = (a_2 a_1 a_0)_5 = (59)_{10} = 50 + 5 + 4 = 50 + 5 \cdot 5^0 + 4 \cdot 5^1 + 1 \cdot 5^2 + 0 \cdot 5^3 = (214)_5$.
- בסיסים שונים חזקות של 2: כל ספרה בסיס אוקטלי ניצג על ידי 3 ספרות בינאריות, וכל ספרה בסיס הקסדצימלי ניצג על ידי 4 ספרות בינאריות. בולמר, **במעבר מבסיס z לבסיס 2, נוכל להתאים כל ספרה ל- $\lceil z \rceil$ ספרות ביבנארית.**
- **מעבר מכל בסיס לכל בסיס:** נعبור דרך בסיס 10.
 - **דוגמה ראשונה:** מבסיס 16 לבסיס 4 דרך בסיס 2.

$$(DE.CAF)_{16} = (?)_4$$

$$(DE.CAF)_{16} = (1101 \text{ } 1110. \text{ } 1100 \text{ } 1010 \text{ } 1111)_2 = \\ (11 \text{ } 01 \text{ } 11 \text{ } 10. \text{ } 11 \text{ } 00 \text{ } 10 \text{ } 10 \text{ } 11 \text{ } 11)_2 = \\ (3 \text{ } 1 \text{ } 3 \text{ } 2. \text{ } 3 \text{ } 0 \text{ } 2 \text{ } 2 \text{ } 3 \text{ } 3)_4$$

בשים לב שאנו מתחילה את קיבוץ הספרות מהנקודה ימינה לחילוק השברי (בקצה הימני מוסיפים אפסים אם נדרש), ומהנקודה שמאלה לחילוק השלם (בקצה השמאלי מוסיפים אפסים אם נדרש).

- **דוגמה שנייה:** מבסיס 16 דרך 4 דרך 10. את החלוק שלם נסיק **מלמטה למעלה**, בעוד את החלוק השברי נסיק **מלמטה למעלה** (נכפיל בכל פעם את 4 = $\lceil 4 \rceil$ בשבר, נשמר את החלוק שלם ונמשיך לחישוב עם השבר, עד שנתקבל שבר שהוא 0).

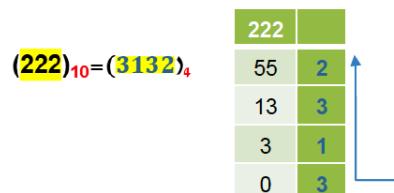
$$(DE.CAF)_{16} = (13 \cdot 16^1 + 14 \cdot 16^0 + 12 \cdot 16^{-1} + 10 \cdot 16^{-2} + 15 \cdot 16^{-3})_{10} = \\ (222.79272460937)_{10}$$

עתה נניר את המספר השברי 0.79272460937 מעשרוני לבסיס 4:
נרשום בתוצאה את החלוק שלם **מלמטה למעלה**.

$$\begin{aligned} (0.170898437484 \text{ (ושבר 3)} &= 4 * 0.79272460937 \\ (0.68359374992 \text{ (ושבר 0)} &= 4 * 0.17089843748 \\ (0.73437499968 \text{ (ושבר 2)} &= 4 * 0.68359374992 \\ (0.93749999872 \text{ (ושבר 2)} &= 4 * 0.73437499968 \\ (0.7499999488 \text{ (ושבר 3)} &= 4 * 0.93749999872 \\ (0.7499999488 \text{ (ושבר 0)} &= 4 * 0.7499999488 \end{aligned}$$

$$(0.79272460937)_{10} = (0.302233)_4$$

תחילה נניר את המספר השלם 222 מעשרוני לבסיס 4:
נרשום בתוצאה את השאלית **מלמטה למעלה**.



$$(222)_{10} + (0.79272460937)_{10} = (3132)_4 + (0.302233)_4 = (3132.302233)_4$$

יצוג מספרים ממשיים:

1. **נקודה קבועה (Fixed Point)** – קיבוע מקום הנקודה העשויונית, ולפיכך מספר הביטים שמייצגים את החלק השברי, ומספר הביטים שמייצגים את החלקשלם. לדוגמה:

$$(1010.1100)_2 = 10.75$$

$$2^3 + 2^1 = 8 + 2 = 10$$

$$\begin{aligned} 2^{-1} + 2^{-2} = \\ 0.5 + 0.25 = 0.75 \end{aligned}$$

כדי לחבר שני מספרים נוכל לבצע ליעזר בינהר, לרפוד באפסים, לחבר ואז לחזור לבסיס עשרוני.

2. **נקודה צפה (Floating Point)** – נרצה הצגה שטוחה הייצוג שלו מאוד גדול. עבור מספרים גדולים נרצה דיקון קטן, ועבור מספרים קטנים נרצה דיקון גדול. נחלק את רצף הביטים לשני קטעים: Mantissa-Exponent. נקבל: $N = r^E \cdot M$.

יצוג מספרים עם סימן בסיס 2:

שיטת גודל וסימן (Sign and Magnitude)	ערךון (0), שלילי (1).	טווח ייצוג חיוביים שליליים																
$N = (-1)^{a_{n-1}} \sum_{i=0}^{n-2} a_i 2^i$	הביט השמאלי ביותר מייצג את סימן המספר: חיובי (0) , שלילי (1) .	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>000</td><td>001</td><td>010</td><td>011</td><td>100</td><td>101</td><td>110</td><td>111</td> </tr> <tr> <td>0</td><td>1</td><td>2</td><td>3</td><td>-0</td><td>-1</td><td>-2</td><td>-3</td> </tr> </table> <p>טווח הייצוג: $-(2^{n-1} - 1), \dots, -1, 0, 1, \dots, + (2^{n-1} - 1)$</p>	000	001	010	011	100	101	110	111	0	1	2	3	-0	-1	-2	-3
000	001	010	011	100	101	110	111											
0	1	2	3	-0	-1	-2	-3											
$N = -a_{n-1}(2^{n-1} - 1) \sum_{i=0}^{n-2} a_i 2^i$	גם כאן יש ביט המציג האם המספר חיובי או שלילי. הפעם, אם המספר שלילי שאר הביטים מצינים את ערך המשלים הבינארי שלו (אם המספר חיובי – מצינים את הערך המוחלט שלו). $a_{n-1} \quad a_{n-2} \quad \dots \quad a_0$ <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>sign</td> <td></td> <td>לחיובי-ערך בינהר</td> <td>לשלילי-המשלים</td> </tr> </table> $11011 \rightarrow 00100 = 2^2 = 4$	sign		לחיובי-ערך בינהר	לשלילי-המשלים	שיטת המשלים לאחד (1's Complement)												
sign		לחיובי-ערך בינהר	לשלילי-המשלים															
$N = -a_{n-1} \cdot 2^{n-1} \sum_{i=0}^{n-2} a_i 2^i$	דמיון לשיטה הקודמת, אך הפעם אם המספר שלילי שאר הביטים מצינים את ערך המשלים הבינארי ועוד אחד (על מנת למנוע את הייצוג הקבול של אפס). $11100 \rightarrow 00011 \rightarrow 00100 = 2^2 = 4$	שיטת המשלים לשתיים (2's Complement)																

$$(-5) - 6 = (-5) + (-6) = -11$$

$$\begin{array}{r}
 11010 \\
 + 11001 \\
 \hline
 110011
 \end{array}$$

המשלים

שלילי

$$\begin{array}{r}
 \xrightarrow{+1} \\
 10100
 \end{array}$$

$$(01011)_2 = (11)_{10}$$

דוגמאות :

$$9 - 4 = (+9) + (-4) = 5$$

$$\begin{array}{r}
 01001 \\
 + 11011 \\
 \hline
 100100
 \end{array}$$

המשלים

$$\begin{array}{r}
 \xrightarrow{+1} \\
 00101
 \end{array}$$

- יצוג של 5 – למשל, יהיה 1 עבור מספר שלילי, וניקח את המשלים של 1010 שווה 0101, ונקבל $11010 = 1010 + 1$.
- חיבור וחיסור: מוחברים את שני המספרים. **אם יש נושא (carry)** מוחקם אותו (הביט השמאלי ביותר), ומוסיפים אחד לתוצאה:

משלים-2:

- חישוב משלים (שיטת ראשונה) – **משלים + מוסיפים 1 לתוצאה.**
- חישוב משלים (שיטת שנייה) – מתקדים מימין ומשאים אפסים ללא שינוי, עד הסופה הראשונה שהיא 1. אחרת מחליפים כל ספרה במשלים שלה (NOT).
- לחברות וחישור: לחברות את שני המספרים, אם יש נשא (carry) מוחקים אותו (**ולא מוסיפים אחד**).
- **galishah (overflow)** – אם התוצאה לא ניתן לייצוג נקלט גלישה. **סמן התוצאה יהיה הפוך לסימני שני המספרים (תייחס רק אם שני המספרים שוויים סימן).**

סיכום:

מספר בינרי	ערך (ללא סימן)	Sign/ Magnitude	1's Complement	2's Complement
000	0	0	0	0
001	1	1	1	1
010	2	2	2	2
011	3	3	3	3
100	4	0	-3	-4
101	5	-1	-2	-3
110	6	-2	-1	-2
111	7	-3	0	-1

אלגברה בولיאנית (תרגול 2)אלגברה ופונקציות בוליאניות:

אלגברה בוליאנית היא מבנה אלגברי המוגדר על קבוצת איברים B בצירוף שני אופרטורים ביןארים: חיבור (+) וכפל (\cdot), בקר שמתיקיות 6 האקסיום של Huntington לכל $B \in z, y, x$: סגורות, איבר ייחודה, חילוף, פילוג, משלים, לפחות שני איברים בקבוצה. אנחנו נעבד עם הקבוצה $\{0, 1, B\}$, זו תהיה אלגברה בוליאנית דוארכית.

- חוק פילוג במקרה שלנו: $(z + x)(y + x) = zy + x$. זה עובד רק במקרה שבו המשתנים בסוגרים זהים! יש לנו כאן x פעמיים, אם אחד מהם היה z זה לא היה עובד.
- משלים: $1' = 0, x + x' = x' \cdot x$.
- האופרטור + מקביל ל-OR, האופרטור · מקביל ל-AND.
- קידימות אופרטורים: $NOT \rightarrow AND \rightarrow OR \rightarrow NOT$.

x	y	$x \cdot y$	$x + y$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

x	Not(x)
0	1
1	0

כללים אלו זהים לפועלות AND, OR ו-NOT בהתאם.

משפטים יסודים:

1. דה-מורגן:

$$\begin{aligned} a. (x \cdot y)' &= x' + y' \\ b. (x + y)' &= x' \cdot y' \end{aligned}$$

2. אידempוטנטיות:

$$\begin{aligned} a. x \cdot x &= x \\ b. x \cdot 0 &= 0 \end{aligned}$$

3. אסוציאטיביות:

$$\begin{aligned} a. (x + y) + z &= x + (y + z) \\ b. (x \cdot y) \cdot z &= x \cdot (y \cdot z) \end{aligned}$$

4. צמצום:

$$\begin{aligned} a. x + x \cdot y &= x \\ b. x \cdot x + x \cdot (x + y) &= x(1 + y) = x \cdot 1 = x \end{aligned}$$

5. דברים נוספים:

$$\begin{aligned} a. x + x' &= 1 \\ b. x \cdot x' &= 0 \\ c. x(x + y)' &= 0 \\ d. x + x'y &= (x + x')(x + y) = x + y \end{aligned}$$

פונקציה בولיאנית: פונקציה בוליאנית היא ביטוי אשר מכיל משתנים בינהירים (ערבים 0 או 1), את שני האופרטורים הבינהירים AND ו-OR, האופרטור האונארי NOT, סוגרים וסימן שווון. עבור ערכים נתונים של המשתנים, הפונקציה מקבלת ערך 0 או 1.

x	y	y'	F
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

נתן להציג באופן בסיסי פונקציה:

• בביטוי אלגברי - $xy' = F(x, y)$.

• בפעולות טבלת אמת. נניח שיש לנו n משתנים בינהירים של הפונקציה, מספר השורות בטבלת האמת הוא 2^n . נציג את כל הקומבינציות האפשרות של המשתנים.

הציג בסכום מכפלות סטנדרטיות (SoP - minterms): הציג פונקציה בוליאנית **בסכום של מכפלות**, שבכל אחת מהמכפלות נמצאים כל משתני הפונקציה. בהינתן סכום מכפלות, לדוגמה: $xyz' + x'y'z + x'y'z' = F(x, y, z)$, **מתי הסכום יהיה 1?**

- תנאי לסכום: מספיק שאחת המכפלות (minterm) מקבלת ערך 1 כדי שערך הפונקציה יהיה 1.
- תנאי למינימל בודד: כאשר כל הליטרלים (משתנים או שלילתם) הם 1 אזי ערך המכפלה הוא 1. כך מקבל עבור פונקציה זו: מכפלה ראשונה תהיה 1 כאשר $x = 0, y = 0, z = 0$, המכפלה השנייה תהיה 1 כאשר $x = 1, y = 0, z = 0$, המכפלה השלישייה תהיה 1 כאשר $x = 0, y = 1, z = 0$, והמכפלה הרביעית תהיה 1 כאשר $x = 1, y = 1, z = 0$.

בהינתן טבלת אמת: נסתכל בכל המקומות בהם הפונקציה מקבלת 1. נסיף מכפלה לסכום בה עבור ערכי המשתנים z, y, x יתקבל 1. למשל עבור $z = 0, y = 0, x = 0$ נסיף את המכפלה $x'y'z$. **מספר את השורות מ-0 עד 7** במקורה זה, בהתאם ליצוג העשורי של המספר הבינארי שמודגש באמצעות הספרות z, y, x .

x	y	z	F ₁
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

כתיב של F₁ בסכום מכפלות סטנדרטיות:

$$\begin{aligned} F_1(x, y, z) &= x'y'z + x'yz + \\ &\quad xy'z' + xyz' \\ &= m_1 + m_3 + m_4 + m_6 \\ &= \sum(1, 3, 4, 6) \end{aligned}$$

האודות של הפונקציה בטבלת האמת מייצגים את המכפלות הסטנדרטיות!

			מכפלה סטנדרטית Min term
סימון	גורם	y	x
m_0	$x'y'z'$	0	0
m_1	$x'y'z$	0	0
m_2	$x'yz'$	0	1
m_3	$x'yz$	1	0
m_4	$xy'z'$	1	0
m_5	$xy'z$	1	0
m_6	xyz'	1	1
m_7	xyz	1	1



הצגה כמכפלת סכומים סטנדרטיבים (PoS - maxterms): הצגת פונקציה בولיאנית כמכפלה של סכומים, כך שבסכום אחד מתקבלים נמצאים כל משתני הפונקציה. בהינתן מכפלת סכומים, לדוגמה: $(x' + y + z')(x' + y' + z)(x + y + z') = F(x, y, z)$, **מהו הסכום יהיה?**

- תנאי למכפלה: מספיק שאחד הסכומים (**maxterm**) קיבל את הערך 0 כדי שערך הפונקציה יהיה 0.
- תנאי לסכום בודד: באשר כל הליטרלים (משתנים או שלילתם) הם 0 אז ערך הסכום הוא 0. כך נקבל עבור פונקציה זו: סכום ראשון יהיה 0 כאשר $x = 0, y = 0, z = 0$, הסכום השני יהיה 0 כאשר $x = 1, y = 1, z = 0$, והסכום השלישי יהיה 0: $x = 1, y = 0, z = 1$.

בהינתן טבלת אמת: נסתכל בכל המיקומות בהם הפונקציה מקבלת 0. נוסיף **סכום** למכפלה בו עבור ערכי המשתנים z, y, x יתקבל 0. למשל עבור $0 = x, y = 0, z = 0$ נוסיף את הסכום $z + y + z'$.

x	y	z	F_1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

כתב של F_1 כמכפלת סכומים סטנדרטיבים:

$$\begin{aligned} F_1(x, y, z) &= (x + y + z) \cdot (x + y' + z) \cdot \\ &\quad (x' + y + z') \cdot (x' + y' + z') \\ &= M_0 \cdot M_2 \cdot M_5 \cdot M_7 \\ &= \prod(0, 2, 5, 7) \end{aligned}$$

האפסים של הפונקציה בטבלת האמת

מייצגים את הסכומים הסטנדרטיבים!

			סכום סטנדרטיבי Max term	
סימון	גורם	z	y	x
M_0	$x + y + z$	0	0	0
M_1	$x + y + z'$	0	0	1
M_2	$x + y' + z$	0	1	0
M_3	$x + y' + z'$	0	1	1
M_4	$x' + y + z$	1	0	0
M_5	$x' + y + z'$	1	0	1
M_6	$x' + y' + z$	1	1	0
M_7	$x' + y' + z'$	1	1	1

מצום ביטויים בוליאניים באמצעות אלגברה בוליאנית:

נTHONה לנו הפונקציה $y'uz + x'y'z + x'yz + xy' = F_1(x, y, z) = x'y'z + x'yz + x'yz + xy' = x'z(y' + y) + xy' = x'z \cdot 1 + xy' = x'z + xy'$. נרצה להראות שהוא שקולה לפונקציה $F_1(x, y, z) = x'y'z + x'yz + x'yz + xy' = x'z(y' + y) + xy' = x'z \cdot 1 + xy' = x'z + xy'$ באמצעות פיתוח אלגברי נקי:

$$F_1(x, y, z) = x'y'z + x'yz + xy' = x'zy' + x'zy + xy' = x'z(y' + y) + xy' = x'z \cdot 1 + xy' = x'z + xy'$$

דוגמא נוספת:

$$\begin{aligned} F(x, y, z) &= (x + y)[x'(y' + z')]' + x'y' + x'z' = (x + y)[(x + (y' + z'))' + x'y' + x'z'] \\ &= (x + y)(x + yz) + x'y' + x'z' = x + yyz + x'y' + x'z' = x + yz + x'y' + x'z' \\ &= x + yz + x'y' + x'z' = \dots = \dots = 1 \end{aligned}$$

מצום ביטויים בוליאניים באמצעות מופות קרמן:

שלב בנית המפה הינו חד-פערמי, מפה של n משתנים תשמש אותנו לכל פונקציה בעלת n משתנים. הטבלה תכיל 2^n משבצות.

		y			
		0	0	1	1
x	0	m_0	m_1	m_3	m_2
	1	m_4	m_5	m_7	m_6
		0	1	1	0
				z	

		y			
		0	0	1	1
x	0	$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
	1	$xy'z'$	$xy'z$	xyz	xyz'
		0	1	1	0
				z	

			מכפלות סטנדרטיביות	
סימון	גורם	z	y	x
m_0	$x'y'z'$	0	0	0
m_1	$x'y'z$	0	0	1
m_2	$x'yz'$	0	1	0
m_3	$x'yz$	0	1	1
m_4	$xy'z'$	1	0	0
m_5	$xy'z$	1	0	1
m_6	xyz'	1	1	0
m_7	xyz	1	1	1



(1) מפת קרנו עבור 3 משתנים:

נכטו טבלת אמת, ונרצה להמיר את זה למפת קרנו. אפשר לעשות את זה **לפי השבלונה**: נציב m_i במקום המתאים (אפשר גם להציב ידנית את גורמי המכפלה במשבצת המתאימה בטבלה, לפי המיקום שבו z, y, x אמורים להיות 0 או 1).

		z			
		00	01	11	10
x \ yz		x'y'z'	x'y'z	x'yz	x'yz'
0	0	x'y'z'	x'y'z	x'yz	x'yz'
	1	xy'z'	xy'z	xyz	xyz'

- **ריבוע אחד** מייצג מכפלה אחת, התורמת גורם בועל **3 ליטרים**.
- **שני ריבועים** סמוכים מייצגים גורם בועל **2 ליטרים** (אחרי פישוט).
- **ארבעה ריבועים** סמוכים מייצגים גורם בועל **1 ליטר אחד**.
- **שמונה ריבועים** סמוכים מייצגים את הפונקציה השווה ל-1.
- ניתן לראות לפיה הסימונים של המשתנים,iziaה ריבוע יצא ע, המשלים שלו יהיה 'ע ו'ו'.

תכונות:

- כל משבצת מתאימה ל-minterm אחד של המשתנים.
- כל שני ריבועים סמוכים במאפה נבדלים במשטנה אחד בלבד.
- קבוצות מרכיבות מ-1-ים בלבד (פרט לצירוף אדיש). צריך לנסות את כל ה-1-ים.
- מספר האיברים בקבוצה הוא חזקה של 2 (כולל 1): 2, 4, 8, ... 16, 32, ...
- קבוצות לא יכולות להיות אלכסוניות וחיברות להיות מלכניות.
- על קבוצות להיות גדולות ככל האפשר (עדיף שמנינה על ריבועה).
- הטבלה היא **מעגלית**.

דוגמאות:

- עבור הפונקציה $F(x, y, z) = \Sigma(2,3,4,5)$ נקבל שתי קבוצות, ولكن $x'y + xy' = x'y + xy' + z' = F(x, y, z) = \Sigma(2,3,4,5,6)$.
- עבור הפונקציה $F(x, y, z) = \Sigma(0,2,4,5,6)$ נקבל שתי קבוצות (אחד מהן מעגלית, מותר לקבוצות לחפות) ולכן $F(x, y, z) = xy' + z'$

		z			
		00	01	11	10
x \ yz		0	0	1	1
0	0	1	1	0	0
	1	1	0	0	0

		z			
		00	01	11	10
x \ yz		1	0	0	1
0	0	1	1	0	1
	1	1	0	0	1

(2) מפת קרנו עבור 4 משתנים:

השבלונה קצת שונה, אותן הכללים תקפים. עבור הפונקציה $F(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$ יש לנו שמייניה, ועוד שתי ריביעיות מעגליות שיחפפו עם השמייניה, כי אנחנו יכולים ליצור קבוצות בגודל חזקות של 2. נקבל $y' + xz' + w'z'w$.

		00	01	11	10
wx \ yz		w'x'y'z'	w'x'y'z	w'x'yz	w'x'yz'
00		w'xy'z'	w'xy'z	w'xyz	w'xyz'
01		wxy'z'	wxy'z	wxyz	wxyz'
11		wx'y'z'	wx'y'z	wx'yz	wx'yz'
10		wx'y'z'	wx'y'z	wx'yz	wx'yz'

		y			
		00	01	11	10
wx \ yz		m_0	m_1	m_3	m_2
0	0	m_4	m_5	m_7	m_6
	1	m_{12}	m_{13}	m_{15}	m_{14}
1	0	m_8	m_9	m_{11}	m_{10}
	1				

		y			
		00	01	11	10
wx \ yz		1	1		1
0	0	1	1		1
	1	1	1	1	1
1	0	1	1	1	1
	1	1	1	1	1
10	0	1	1	1	1
	1	1	1	1	1

(3) צירופים אדישים:

צירוף אדיש זה איבר שנראה לו **דיוקה** (בשניהם לנו הוא יהיה 0 ובשניהם לנו הוא יהיה 1). אם הוא עוזר לנו למצוא קבוצות גדולות הוא יהיה 1, ואם הוא לא עוזר לנו הוא יהיה 0. נסמן איבר זה ב-Φ.

נזכיר לדוגמה מוקדם, ונניח שיש לנו איבר אדיש: $(7)\Sigma = d(x, y, z)$ אז נוכל להסתכל עליו כ-1 ולבנות בעזרתו ביטוי אחר:

		00	01	11	10
		z			
x	y	1	0	0	1
0	1	1	1	0	1

$$F(x, y, z) = z' + xy'$$

		00	01	11	10
		z			
x	y	1	0	0	1
0	1	1	1	0	1

$$F(x, y, z) = z' + x$$

(4) פישוט למכפלת סכומים:

- נזכור שכל פונקציה ניתן לבתוב בסכום מכפלות, ובמכפלת סכומים את **קבוצות האפסים**.
- כאשר במכפלת סכומים אנחנו מוחפשים את **קבוצות האפסים**.
- כדי למצוא את הפונקציה נוכל לרשום את F ' המשלים של הפונקציה (לפי קבוצות האפסים שמצאנו), ואז לעשות עליו NOT כדי לדעת למה שווה F .

הערות:

- במה פונקציות שונות מיוצגות על ידי המפה: ^[4] 2, תלוי בcomaות הצירופים האדישים (כל הקומבינציות שלהם).
- אם הביטוי שנתקבל הוא ייחיד? יכול להיות שהוא דומה (כמו ליטרלים) באמצעות הצירופים האדישים.

פונקציות שלמות:

פונקציה שלמה: פונקציה שביתן למש באמצעות NOT-AND, או NOT-OR.

נתון כי הפונקציה Nand היא פונקציה שלמה: $Nand(x, y) = (x \cdot y)'$. נראה כי אפשר באמצעות הפעלה/הרכבה שלה להגיע ל- AND NOT וגם ל- NOT AND.

- עבור NOT נבצע: $Not(x) = (x \cdot x)' = x' = Not(x)$.
- עבור AND נבצע: $And(x, y) = ((x \cdot y)' \cdot (x \cdot y)')' = (x \cdot y)'' = x \cdot y = And(x, y)$.

טענה: הפונקציה $z = x' + y \cdot f(x, y, z) = x' + y$ והפונקציות הקבועות 0, 1 הינן קבוצה שלמה.

הוכחה: נראה כי ניתן למש באמצעות NOT.

- עבור NOT נבצע: $Not(y) = f(x, 0, 0) = x' + 0 \cdot 0 = x' = Not(y)$.
- עבור AND נבצע: $And(x, y) = f(1, x, y) = 1' + x \cdot y = 0 + x \cdot y = And(x, y)$.
- עבור OR (יצורי קצת) נבצע: $f(f(x, 0, 0), y, 1) = f(x', y, 1) = x + y \cdot 1 = x + y = Or(x, y)$.



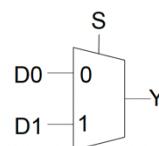
מעגלים

מעגלים צירופיים

: (Multiplexer) MUX

- ♦ 2:1 multiplexer chooses between two inputs

S	D ₁	D ₀	Y
0	X	0	0
0	X	1	1
1	0	X	0
1	1	X	1



$$\text{MUX}(S, D_0, D_1) = S \cdot D_1 + S' \cdot D_0$$



סוג של מפסק, רק רמה אחת מעלה. יש לנו **RGL S** שבוחרת בין שני קלטים. אם $S = 0$ נרצה לקבל את הקלט של D_0 (לא משנה מה הערך של D_1). אם $S = 1$ נרצה לקבל את הקלט של D_1 (לא משנה מה הערך של D_2).

נמשש את המעגל באופן הבא: נחבר את S_1, D_1 באמצעות AND, נחבר את S_2, D_2 באמצעות AND אשר נבצע לפניו NOT על S , כדי לקבל את הערך מהעיגן של S . לבסוף נבצע OR. ניתן לראות את התאמיות בין בתיב הפונקציה למימוש הלוגי שלו.

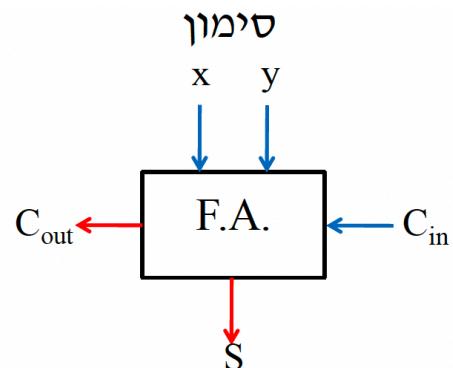
יש לנו סה"כ 4 שערים לוגיים (ט', ט"ו – AND, ט"ז – NOT, ט"ז – OR, ט"ז – OR). סה"כ נקבל $20 = 6 \cdot 4 = 24$.

נשים לב כי $(S' \cdot D_0) \cdot (S' \cdot D_1) + S' \cdot D_1 + S \cdot D_0 = [(S \cdot D_1) + S' \cdot D_0] \cdot S$ ולכן נוכל להפוך את שני AND ל-NAND, ואת האחרון גם ל-NAND. כך נוכל לחסוך בטרמיניסטים.

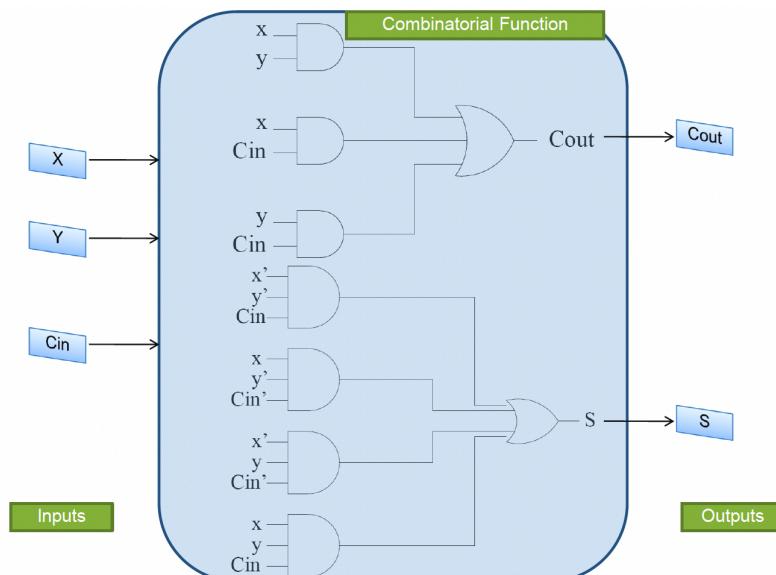
: (Full Adder)

למחבר המלא 3 כניסה: x, y, C_{in} ויציאות: $S, C_{out}, x, y, C_{in}, C_{out}$. הוא מבצע את החישוב $C_{in} + y + x$. התוצאה ניתנת ב-S והנשא ב- C_{out} . זו ייחידה בסיסית שմבוצעת חיבור. הקלט שמתפרק הוא שני ביטים.

x	y	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

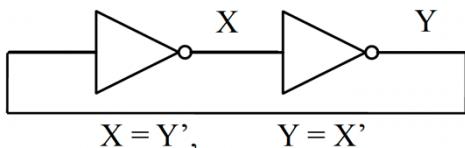


- מה שמיוחד בכך הוא שיש שתי יציאות, שתי פונקציות, שתי פונקציות. **כל פונקציה בזו מבצע חישוב נפרד, מפת קרנו נפרדת.**
- אם נרשום מפת קרנו לפונקציה שמניבבה את S, נקבל 4 גורמים בסכום לפי PSo. נוכל להבחן בעדינות שמדובר בשער XOR. כך נחשוך הרבה טרמיניסטים.



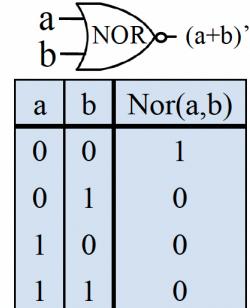
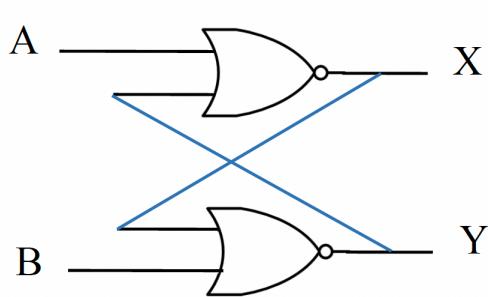


:(Set-Reset) SR Latch



נסתכל על המעגל הבא, אשר מתקף בזיכרון. אם נכניס $0 = X$ מקבל $Y = 1$ ולאחר מכן שוב את הערך 0 ב- X . כך גם אם היינו מכניסים את ה- 1 , הינו שומרים על הערך 1 . יש כאן **שמירת מצב**.

בצדי נוכל להחליף מצב? לשנות על הערך שנכנס לולאה? נהפוך את ה-NOT ל-NOR
וכוסיפ שני בניות בנות A, B .



- מה נשתרם? אם $A = B = 0$ זה תליי רק בנסיבות הקודמות Y, X . הם לא משפיעים, וזה בדיקת המעגל הקודם.
- הופך ל-NOT כמו שהוא קודם: אנו מביצעים $(x) \text{NOR}(0,0)$ וזה פשוט הופך את הערך של x .
- מה נשתנה? ניתן לשנות על ערך היציאה.

נתאר מצב געוי של המעגל: $(X(t+1))' = \text{NOR}(A, Y(t)) = \text{NOR}(B, X(t))$, ובאופן דומה $(X(t+1))' = \text{NOR}(A, Y(t)) = \text{NOR}(B, X(t))$. בעת אנחנו מסתבלים על 4 בניות, ו-2 יציאות (X - Y בזמן מתקדם יותר).

עבור 0=A=B: נקבע את ערך הלולאה להיות $0 = Y = 1, X = 1$. לא משנה מה היו הערבים שהו קודם לכן, נאלץ את הלולאה לעבור למצב זה.

עבור 0=A=1=B: באופן סימטרי נקבע את ערך הלולאה להיות $1 = Y = 0, X = 0$.

מצב בעייתית: עבור 1=A=B=0: התוצאה תהיה $0 = Y = X = 1$ בסופו של דבר. אם מכאן אנחנו העבור למצב 0 ומה התרחש תחולפה מהירה בין $0 = Y = X = 1$ לבין $1 = Y = X = 0$. כאן אנחנו ייכנסו לולאה כזאת של בזבוז אנרגיה (מיוגר מחריר, מעבר בין מצבים). לכן, רצוי שהערבים של A, B יהיו שונים זה מזה. **נגביל את השימוש ברכיב למצבים בהם $X' = Y$** .

A	B	$X(t)$	$Y(t)$	$X(t+1)$	$Y(t+1)$
0	0	0	0	1	1
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	0	0
0	1	0	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
0	1	1	1	0	0

A	B	$X(t)$	$Y(t)$	$X(t+1)$	$Y(t+1)$
1	0	0	0	0	1
1	0	0	1	0	1
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	0	0

אפשר לתאר את טבלת המעברים:

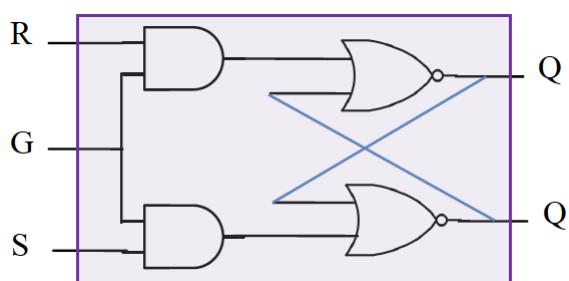
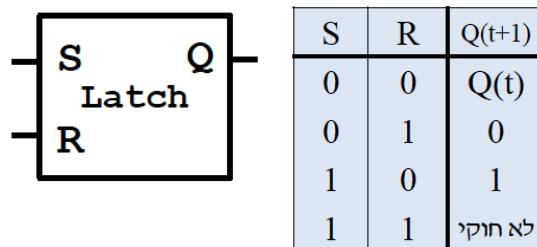
	A=0		A=0		A=1		A=1	
	B=0	B=1	B=0	B=1	B=0	B=1	B=0	B=1
	X=0	X=1	X=0	X=1	X=0	X=1	X=0	X=1
	Y=0							
	Y=1							



B	A	X(t+1)	Y(t+1)
0	0	X(t)	Y(t)
0	1	0	1
1	0	1	0
1	1	0	0

בנהה ש- Y = X
לא חוקי

נכנה את $S = A = R, B = Q$ ונתייחס לפולט שלנו בתור $Q = X$ ואז $Q' = Y$. את ערך הולולה נקבע לפי Q (לכן לרוב לא סמן את Q'). כך קיבלנו **יחידת זיכרון בסיסית**, יש פה יכולת לשמר על מצב קודם, על אף שהוא קצר מגושם – זה עובד.

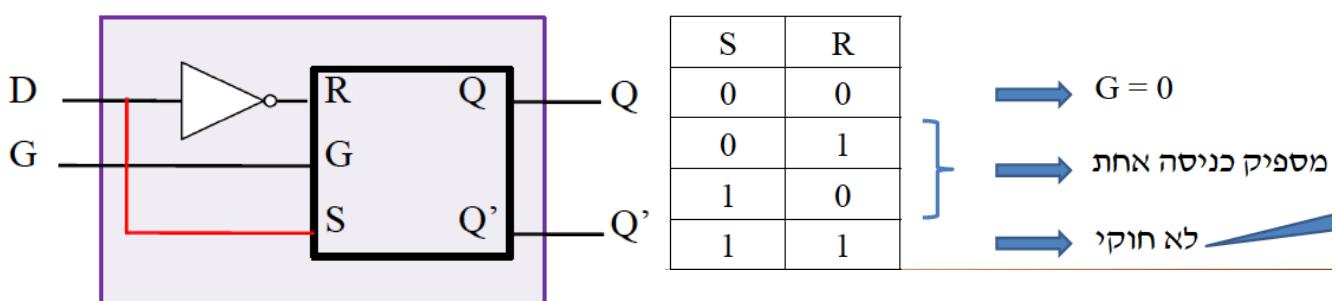


- :Gated SR Latch
- אנחנו רוצים להמשיך ולשכל את הרכיב שבנינו. השכול הראשון הוא Gate. אונחנו נוסיף שני AND-ים ובכניסה G שמשמנת Gate חדש שהוספנו:
 - אם $G = 1$ אז אנחנו מקבלים את ההתחנות הקודמת. ה-Latch פתוח ופועל אותו דבר.
 - אם $G = 0$ אז נאלץ באן $S = R = 0$, וה-Latch נעלם (לא משנה מה הערבים של R, S).

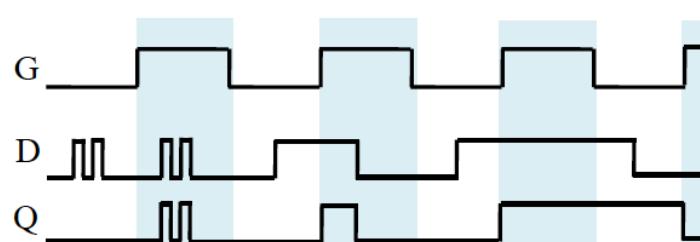
:Gated D-Latch

השכול הבא הוא **למנוע את המצב הלא חוקי** שהוא לנו קודם $R = S = 1$. בעת אנחנו מוסיפים Data. אפשר לבדוק כי מספיקה לנו כניסה אחת, עברו קליטים כמו $S = 0, R = 1, D = 0$ או $S = 1, R = 0, D = 1$.

- D נשמר ברכיב הזיכרון כאשר השער פתוח ($G = 1$) ומתקיים $D = Q$ (בין אם D הוא 1 או 0).
- D לא ישמר ברכיב הזיכרון כאשר השער סגור ($G = 0$) כי הוא מונע שינוי של הזיכרון לא משנה מה הקלט, **Q שומר מצב**.



קיבלנו ייחידת זיכרון. בעת, באופן טיפוסי נרצה לחבר את-h-Gate לשעון (clock). נראה דרך נוספת לייצג את המידע, על תרשימים זמינים. ציר הזמן הוא כמו ציר ה-x ימינה, ובציר ה-y יש לנו 1 לוגי ו-0 לוגי. **Q** נראה בדיק במו D, ורק כאשר G הוא 1 (החלק השוקף). בכל מצב שבו G הוא 0 אנו בשמירת מצב, ו-Q שומר על מצבו הקודם (החלק הנעלם).



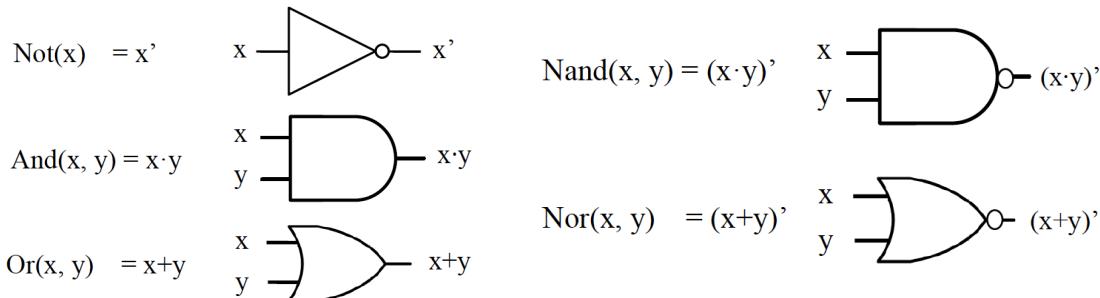


מעגלים צירופיים ויחידות סטנדרטיות (תרגול 3)

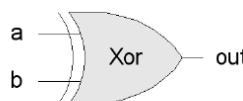
שערים לוגיים:

משתנים בינאריים יכולים לקבל שני ערכים: 0 ו-1. ניתן לייצג ערכאים אלו על ידי מתחים חשמליים (מתוך גובה מייצג 1 לוגי, ומתח נמוך מייצג 0 לוגי). ניתן כי ניתן לבנות שערים לוגיים פשוטים במתחאים אלו, וביצעו את הפעולה הלוגית הדורשה, למשל ע"י טרנזיסטורים בטכנולוגיית CMOS. בעת, מהשערים שבנינו נבנה מעגלים חשמליים שיביצעו פונקציות מסוימות בוליאניות נתונות.

שערים שכבר ראיינו:



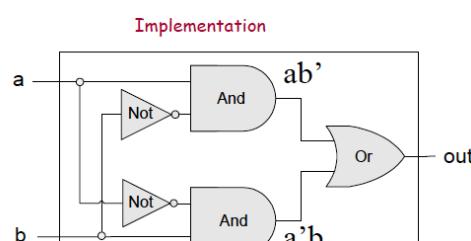
שער נוסף שחשוב להזכיר הוא **XOR**: מחזיר 1 אם "מ הערכים שונים, ו-0 אם הם זהים (שניהם 0 או שניהם 1). אפשר גם להתייחס לשער זה בתורה חיבור עם מודולו 2.



$$\text{out} = ab' + ba' = a \oplus b$$

(פשוט חיבור מודולו 2)

a	b	$\text{Xor}(a,b)$
0	0	0
0	1	1
1	0	1
1	1	0



$$\text{Xor}(a,b) = \text{Or}(\text{And}(a,\text{Not}(b)),\text{And}(\text{Not}(a),b))$$

Reduction or



Reduction nor



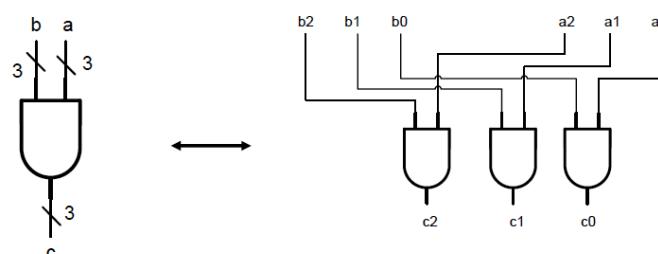
Reduction xor



אם יש לנו מספר כניסה, נוכל לבצע דיזקציה לערך סופי אחד:

- or – קיבל 1 כאשר בכניסה כלשהו יש 1.
- nor – קיבל 1 כאשר בכל הכניסות יש 0.
- xor – קיבל 1 כאשר יש מספר אי-זוגי של 1.

בנוסף, יש לנו **bitwise operation**: כאשר כל זוג כניסה לא תלוי בשני, מטפלים בכל בית בנפרד.



$$c_i = a_i \cdot b_i$$

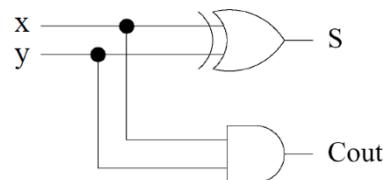
a, b, c are 3 bit variables

$$i = 0, 1, 2$$

הפעולה כאן ברמה של סיביות בודדות

**מעגלים צירופיים:**

מחברים – מעגל צירופי המבצע חיבור של שני ביטים נקרא **חצ'י מ לחבר (Half Adder)**. מעגל צירופי המבצע חיבור של שלושה ביטים (שני ביטים וושא המתקבל מהחיבור) נקרא **מחבר מלא (Full Adder)**. באמצעות שני חמוצים אפשר ליצור מחבר מלא.



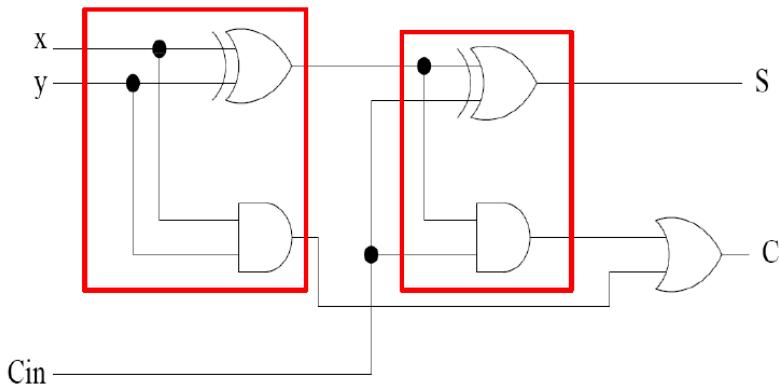
נמשח חצ'י מ לחבר בצורה הבאה:

$$\begin{aligned} S &= xy' + x'y = Xor(x, y) \\ C_{out} &= xy = And(x, y) \end{aligned} \quad \bullet$$

אם נכתוב טבלת אמת למחבר המלא, נוכל לחלץ את שתי הפונקציות (לחפש 1-ים בטבלה) ולצמצם באמצעות מפת קרכנו:

		C _{in}			
		00	01	11	10
		x	y	C _{in}	S
x	0	0	1	0	1
	1	1	0	1	0
y	0	1	1	1	$xy' C_{in}' + x'y' C_{in}$ + $xy C_{in} + x'y C_{in}'$
	1	1	1	1	$C_{out} = yC_{in} + xC_{in} + xy$

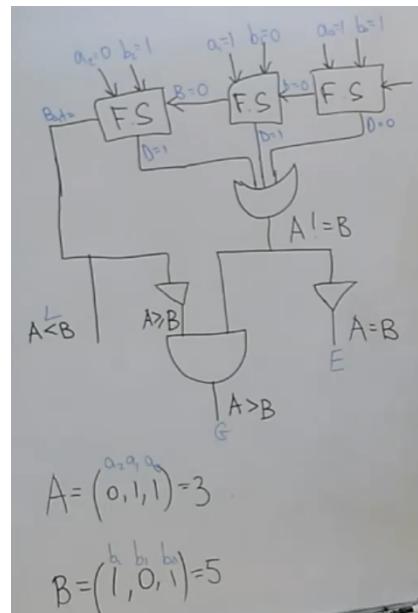
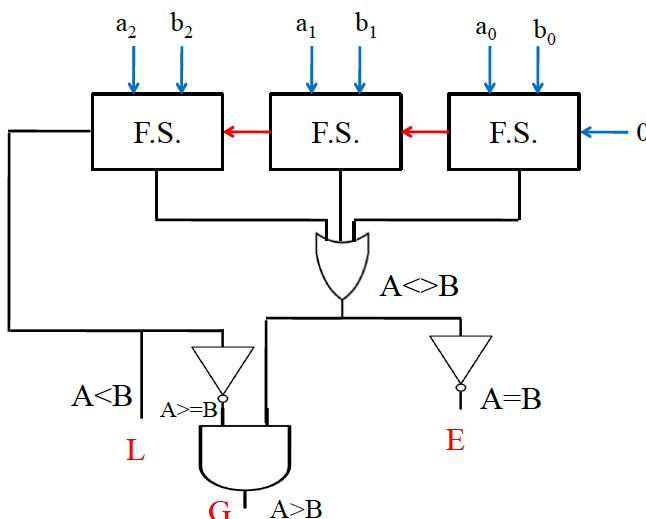
ואז נמשח באמצעות השערים הלוגיים הרלוונטיים (AND, OR). ניתן גם למשח באמצעות שני חמוצים מחברים:



מחברים – גם כאן קיימים חמוצים מחסרים ומחסרים מלאים. חצ'י מחסר, מחסר שני ביטים ויוצר את ההפרש, ויש לו יציאה המציינת אם הושאל 1. מחסר מלא מבצע חיסור בין שני ביטים אבל בנוסף יש בו התחשבות בכר ש-1 יכול היה להיות משאלא בעמדה הקודמת, הקטנה בחישובות. **במקרה 3 נידרש לבנות באליה.**

יחסויים – משווה הוא מעגל אשר נדרש להשוות בין שני מספרים ולציין את היחס ביניהם (גדול, קטן, שווה). בהתאם לכך, למעגל 3 יצרות עבר כל תוצאה אפשרית (שם נשים 1 או 0). כאשר נרצה להשוות בין שני מספרים:

- שיטה ראשונה – משווה את הספירה המשמעותית בין שני המספרים, נמשך לספירה הباء, וכך הלאה.
- שיטה שנייה – באמצעות מחסרים. נשים לב כי מתקנים למשל: $0 > A - B \Leftrightarrow A - B > 0$ וכן נובל להציב תנאי עבור המקדים האחרים. נשרשר מחסרים מלאים מימי' לשמאלי במו בחישור בינהר.
- נרצה לבנות שני מעגלים שמקבלים את A, B , בשכל אחד הוא 3 ביטים ומחזירים L, E, G, C בהתאם לכל מצב אפשרי: גדול, קטן, שווה.
- אם קיבל **ones-worrow** שהוא 1, אז התוצאה היא שלילית. אם זה קורה לאחרון אז קיבל כי $1 = L$.
- ב-OR אנו בעצם מקבלים את תוצאה חיסור כל הספרות. אם המספרים שוויים הכל יהיה 0, ואם לפחות ספרה אחת שונה נקבל 1, ואזណע שהמספרים שונים זה מזה. בצד ימין נקבל את המשלים של השווים: ואז קיבל $1 = E$ כאשר הם שוויים, ו-0 = E כאשר הם שונים.
- את G קיבל על ידי שילוב של $B \geq A$ וגם כאשר $B \neq A$ ולכן $B > A$.

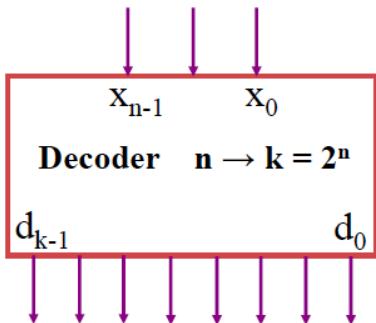


יחידות סטנדרטיות:

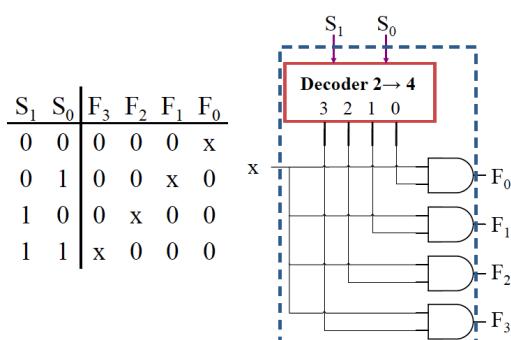
1 – מפענחים (Decoders) בינהרי לעשרוני:

מפענה מכיל n כניסות, $k = 2^n$ יציאות, ומתרגם את הערך מביינארי לעשרוני. נניח שקיבלנו את $x_5x_4x_3x_2x_1x_0 = 101010$, וזה הערך d_5 ב-Decoder 3. זה למשל $8 \rightarrow 3$. Decoder 3 מקבלים

כדי למשתמש את המפענה, נחבר את היציאות עבור d_7, d_2, d_1, d_0 בירק עבורם אנחנו מקבלים 1-ים בייצוג הביניארי.

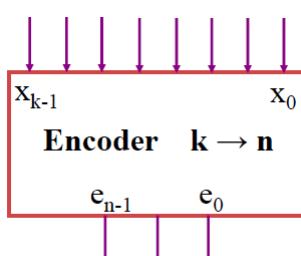


DeMux 1 → 4



2 – מפלגים (Demultiplexers) בינהרי לעשרוני:

מפלג הוא מעין switch, המקבל מידע דרך קו ייחיד ומשדר את המידע הזה באחד מ- 2^n קווים יציאה. הבחירה של קו יציאה מסויימת מבקורת ע"י ערכי הביטים של א' קווי בחירה. יש לנו כינסה אחת שמתפקדת במפסק (x) ועוד n כניסות, 2^n יציאות. **השינוי** ב-Decoder לא משנה איזה קלטים נוכנים, נקבל 1 באחת היציאות. **אם** אנחנו מעוניינים שיבול היציאות יהיה 0, אנו מוסיפים את הכניסה שהוא מתפקדת במפסק שלנו.



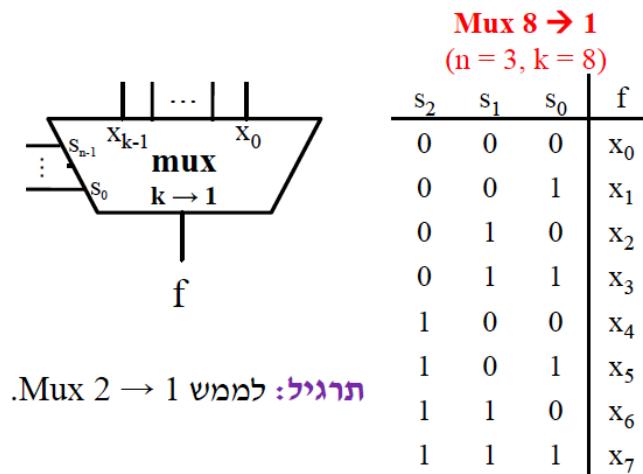
הופכי למפענה. יש לנו n כניסה $2^n = k$ כניסה ו- n יציאות. נתרגם מבסיס עשרוני לביניארי.

3 – מקודדים (Encoders) עשרוני לביניארי:

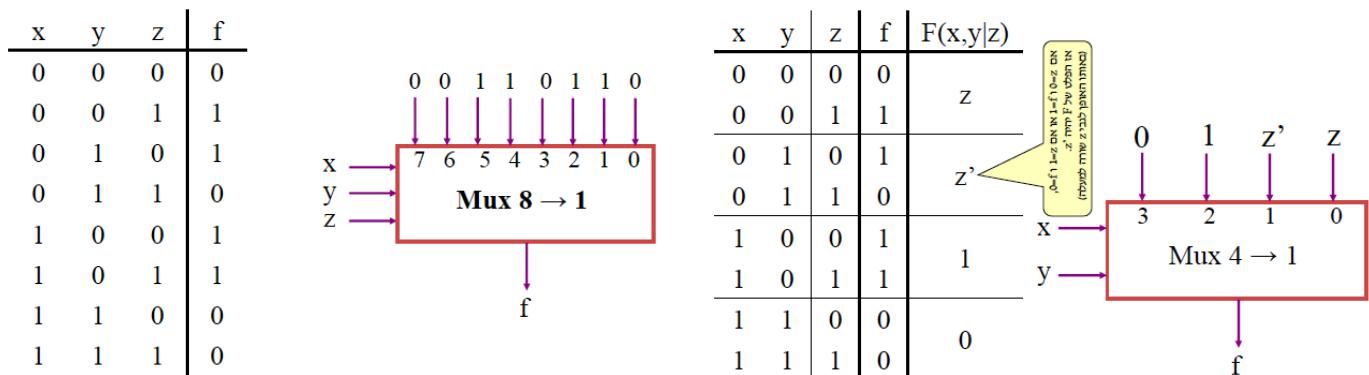
ריבוב הוא צירוף של שני אוטות או יותר משני ערוצים או יותר ושידורים כפלט יחיד.

יש לנו n כניסה $2^n = k$ כניסה ו-מ-קווי בחירה, קו יציאה 1 (f). אנו מתרגמים מבסיס בינהרי לעשרוני את מה שמוספי בעקבות הבחירה, ובוחרים את הכניסה באנדקס שתורגם לנו. למשל בקווי הבחירה קיבלנו את הערך 3, אז בפלט נקבל $f_3 = f$. הרעיון הוא לנתח אל הפלט את הקלט שמספרו הסדרי נתון בסיס 2 ע"י מ-קווי הבחירה.

4 – מרובבים (Multiplexers) בינהרי לעשרוני בקווי הבחירה:



נוכל לממש פונקציהبولיאנית בעזרת מרובבים. איך נממש את זה עם $\text{Mux } 4 \rightarrow 1$?

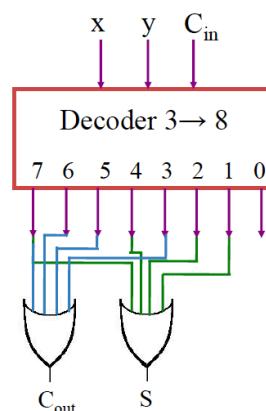


תרגיל – מימוש מחבר מלא בעזרת יחידות סטנדרטיות:

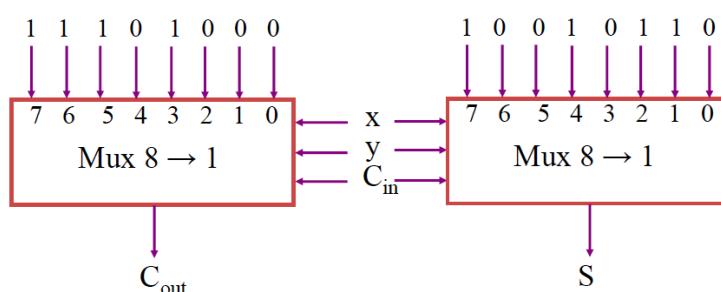
כאן אנו משתמשים במפענה $8 \rightarrow 3$ ושעריו OR על מנת לתפוס את כל ערכי ה-1 בטבלת האמת ל-S-ל- C_{out} .

טבלת האמת של מחבר מלא:

x	y	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

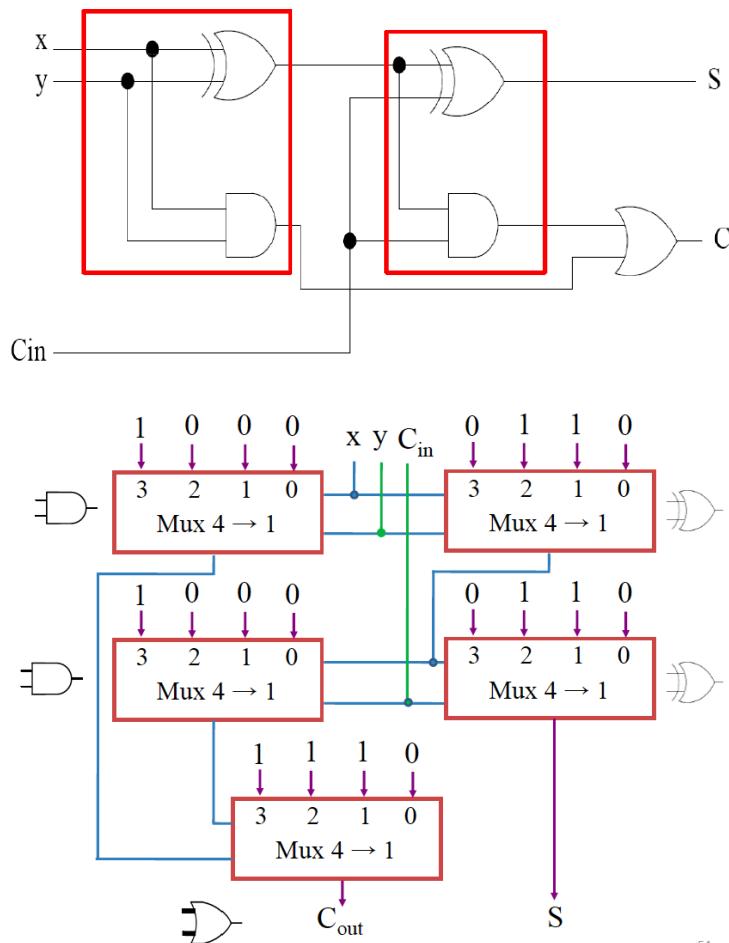


כאן אנו משתמשים במרבי $1 \rightarrow 8$. נשבץ בכל מרבי את ה-1ים המתאימים בקווים הבניות.





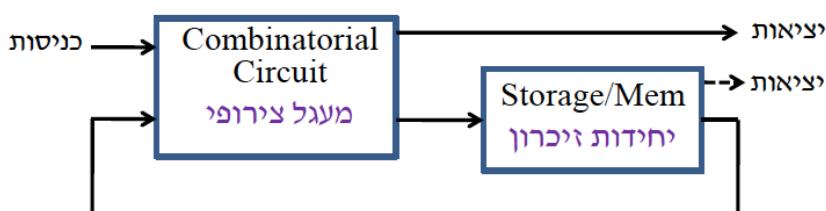
בأنנו עוברים למრבי 1 → 4. נשים לב שמחבר מלא אפשר למשתמש באמצעות שני חריצים: עם שער XOR ושער AND. לכן, אנו ניצור 1 → 4 *Mux* מתאימים לערכו האמת של השערים הלוגיים האלה. לבסוף יש לנו שער OR כדי לקבל את ה-*carry* הסופי.



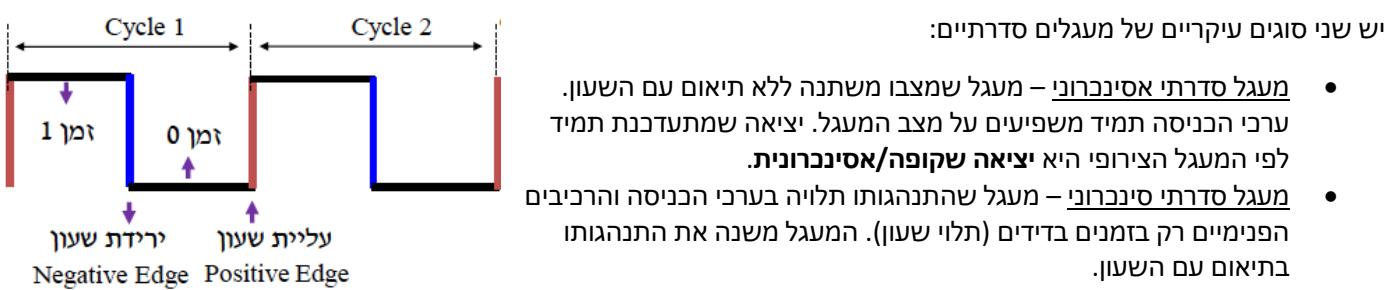
54

מעגלים סדרתיים (Flip Flop)

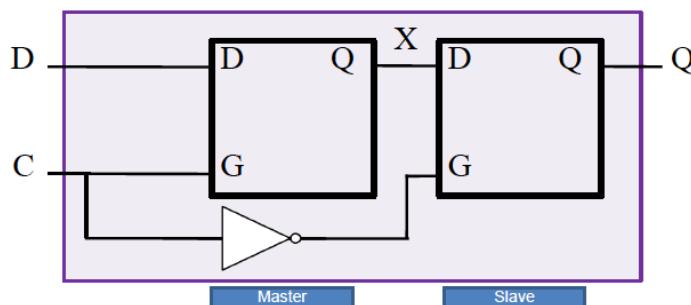
במעגלים סדרתיים, יש מעגל צירופי בשילוב עם ייחדות זיכרון. הערך של היציאות נקבע על ידי הערך של הבניות והערך של ייחדות הזיכרון. יש יציאה שקופה (מתעדכנת תמיד לפני המعال הצירופי), ויש יציאה שקשורה לחידת הזיכרון ולא תמיד מתעדכן (רק באשר ה-*gate* שווה ל-1).



יש שני סוגי עיקריים של מעגלים סדרתיים:



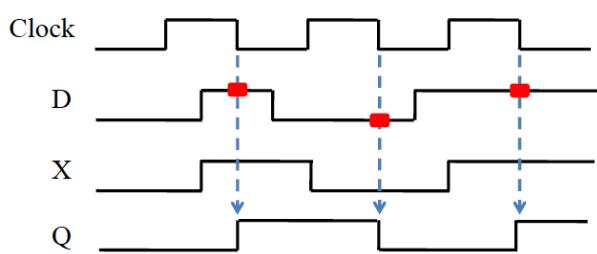
חסרון של **Latch**: יתכן שהרכיב ישנה את ערכו יותר מפעם אחת באותו מחרוז שעון. אין לנו יכולת לקבוע את קצב העברת המידע, ברגע שפותחים את *gate* כל הלוגיקה רצתה. נרצה שbullet מחזור שעון המידע יוזד בתא אחד ימינה ולא "ירוץ" קדימה. ניתן להתמודד עם בעיה זו על ידי כך שה-*gate* יקבל את הערך 1 בבדיקה למן הנחוצ להזוז בית אחד בלבד: זה קשה מאוד להשגה! **רצתה עברו לממבנה שהוא_triggered (ולא מבנה Transparent)**.



Edge triggered D Flip Flop: מימוש אפשרי הוא על ידי שני Master-Slave Latches במבנה תחלתי. אנו יודעים שבמצב התחלתי Gate-to-clock $X = Q = 0$. נחבר את ה-Gate ל- C (Master) ו את היפוך שלו ל-Gate (Slave). במבנה זה **Q יהיה הפלט הסופי** של ה-FF, ו-X הוא הפלט של ה-Master בלבד.

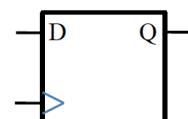
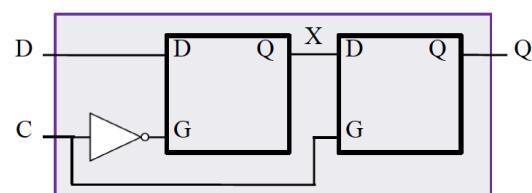
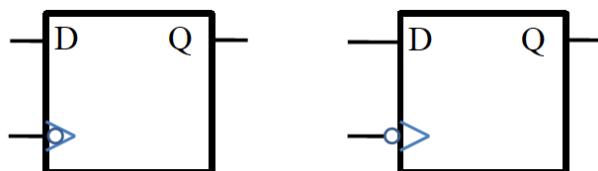
נעקש אחרי השינויים לפि מצב השעון:

- $C = 0$: ה-**Master** נועל (שומר מצב), ה-**Slave** אمنם פתוח אבל לא קורה בו כלום).
- $C = 1$: ה-**Master** נפתח ואז מקבל $X = D$ (X מחקה את D).
- $C = 0$: ה-**Master** נnell (X שומר מצב) וה-**Slave** נפתח ונקלב $Q = X$ (Q ממחקה את X).
- $C = 1$: ה-**Master** נפתח ואז מקבל $X = D$ (X מחקה את D).
- ה-**Slave** נnell (Q שומר מצב).



נשים לב שברגע **ירידת השעון** (מ-1 ל-0) קיבל כי Q מתחלף ובעצם $D = Q$. בנוסף, לרגע אחד קטן מאוד שני-h-**Latches** שkopים (פתוחים) ו- D יכול לעשות את דרכו ל- Q . מה שהיא בירידה, הערך בזמן זה שמאפשר לעבור **M-D** ל-X ול-Q: זה מה שמכונה **droben yridah** (Negative Edge).

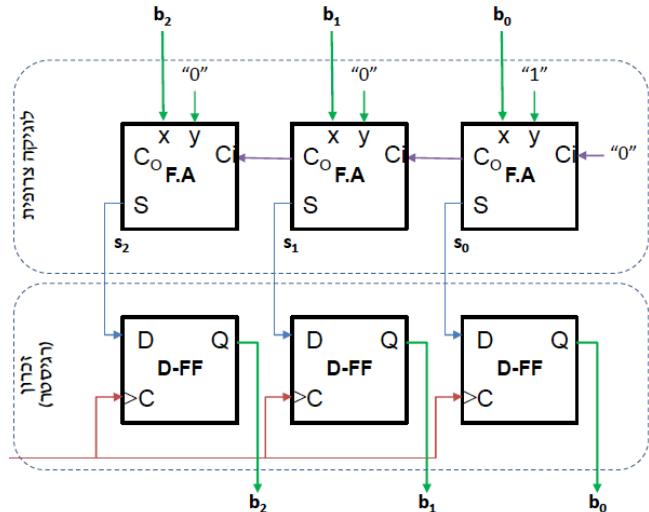
נקבל **droben upla** (Positive Edge) אם נשים את ה-NOT על ה-**master**. כלומר, הערך של D בזמן עליה הוא זה שייצא ב- Q .



דוגמה של Counter

למה צריך FF? באמצעות אוסף של FF לשמר מילימ ורגיסטרים, למשל 32 FF שמחזיקים ערך בינארי של מספר עם 32 ביטים. ברגע אחד של שינוי בשעון כל הביטים היללו ותעדכו, עד שינוי השעון הבא. הריגיסטר מחזק את כל 32 הביטים הללו.

יש לנו כאן מודול שמנון כמה עלויות שעון היו.



- בהתחלה נניח כי $0 = b_0 = b_1 = b_2$ ו $s_0 = 1, s_1 = 0, s_2 = 0$. המספר 000 הפך להיות 001 (הוספנו לו 1) שזה **הערך 1**.

בעליית השעון הראשון (droben עליה):

$$b_0 = s_0 = 1, b_1 = s_1 = 0, b_2 = s_2 = 0$$

- איך זה משפיע על המודול הנוכחי: עבשו יש לנו המספר 001 ואנו מוסיף 1, ונקבל **1 = s₁**, ושזה **הערך 2**.
- בעליית השעון הבא: נעבען את הזיכרון וננעל שם את 010: $b_0 = 0, b_1 = 1, b_2 = 0$.

הריגיסטר של שלושת ה-FF שומר לנו את התוצאה האחורית, והוא

מת לחבר לוגיקה הצירופית לתוכה

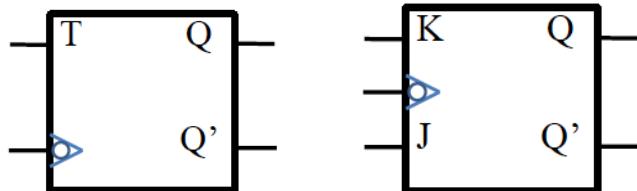
ה-**Adders**, כך שאנו מוסיפים **1** למספר האחרון שהוא בזיכרון **1**, וככה מבצעים **increment in counter**.

אחרי שהגענו למספר 7 (הбитים 111) אנחנו מוסיפים 1 וחוזרים ל-0. כך קיבלנו מודול שופר מ-0 עד 7 בזיכרון מחזורי, בקצב של עלויות שעון.

clk	0	1	2	3	4	5	6	7	0	1
b	0	1	2	3	4	5	6	7	0	1
s	1	2	3	4	5	6	7	0	1	2

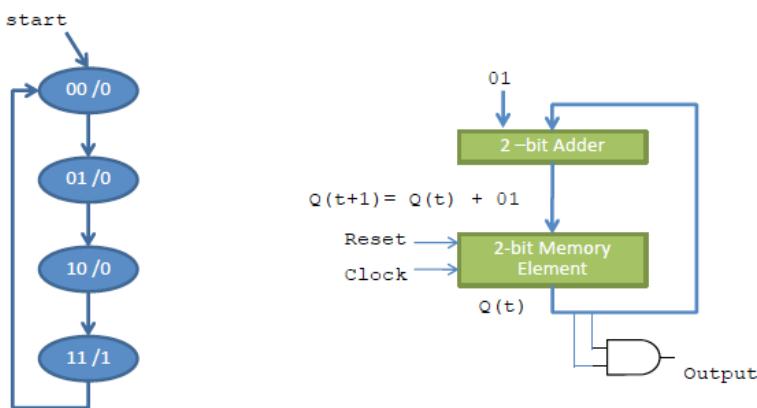
משמעותים נוספים של FF:

- JK Flip Flop – כל אופציה מלבד 1 = J = K = 1 הוא הופך מצב (לא שומר מצב).
- T Flip Flop – ניקח את K, J ונחבר ביניהם לבסיסה אחרת: FF זה רק שומר מצב ($T = 0$) או הופך מצב ($T = 1$). עם זה אפשר למשש אותו בשני פא 2 יותר גדול, בעזרת T-FF עם 1 בבסיסה (הופך מצב בכל ירידת שעון): זה מחלק תדר. אם לחבר עוד T-FF שוב נחלק את התדר פי 2.

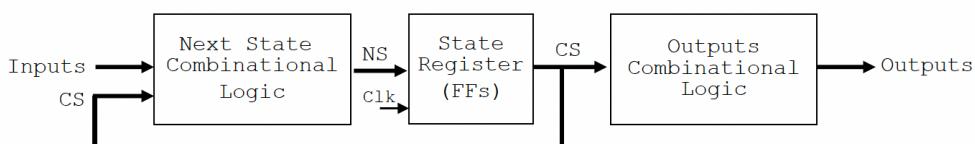
תזכרון**FSM**מבנה מצבים (FSM):

למוכנה יש מצב פנימי (זיכרון), היא יודעת בתוך מה היא במצבה ברגע. מקבלים החלטה לא רק לפני הקלט, אלא אף המצב הפנימי.

- הדוגמה הבci פשוטה ל-FSM הוא שער מסתובב (Rotating Gate).
- דוגמה נוספת היא vending machine: ל-4 מצבים צריך FF 2 כדי לטבעאות (2 ביטים, בערך \log_2 של כמות המצבים).
- ברגע עברו מונה בינהרין:

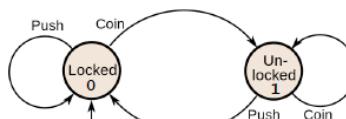


יש שני סוגים של מוכנות: **Moore** (יציאה סינכרונית: תלויות בזיכרון ובקלט) ו-**Mealy** (יציאה סינכרונית: מושפעת רק מהזיכרון בעת עליית שעון). בקורס זהה נעבד בעיקר בעיקר עם מוכנות Moore: נכתבות את ה-Output על המצבים (הוא תלוי רק במצב).

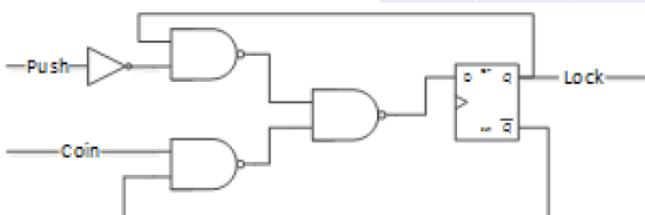
Moore Machine outputs depend on current state onlyהשער המסתובב:**Rotating Gate #3****Next-State logic****Implantation with D-FF :**

- Out : D
- In : Q (CS), Coin, Push
- D = Coin & Q' | Push' & Q

Current State	Coin	Push	Next State
0	0	X	0
0	1	X	1
1	X	0	1
1	X	1	0

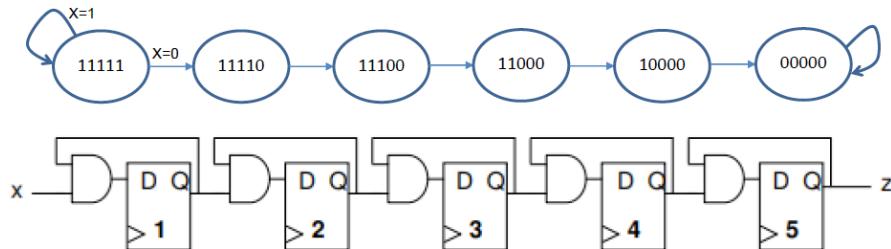


- ניקח את הגרף של המצבים, נמיר אותו לטבלת אמת. הכניסות (הקלטים): Push, Coin, Locked . הייצאה: Un/Locked (ניזג במצבות 0 ו-1). Next State של FF הוא הה-Input .pushנו אותו לפוי $D = C \cdot Q' + P' \cdot Q$. Is can Nand(Nand, Nand) And, Or



דוגמיה ממבחן:

- ◆ Brute force – not good (5 DFF → 32 states)
- ◆ But...
 - Initial state of 11111
 - As long as X is 1, state doesn't change
 - Once X is 0, state changes to 11110 → 11100 → 11000 → 10000 → 00000
- ◆ 6 states → equivalent FSM must use at least 3 DFF
- ◆ Equivalent FSM: mapping of states, and outputs, for all possible input sequences (and initial state)



- מצב התחלתי 11111 (31).
- עבור 1 = x נישאר באותו מצב, כל האחדים נשארים.
- עבור 0 = x קיבל 0 = Q_1 עם עליית השעון (Drvbn עלייה). נשים לב כי בשאר המיקומות יהיו 1 כי רק FF_1 התעדכן. Q_2 לא מספיק להגיב באזונה עליית השעון, הוא עדין רואה 1 (ס"ב 15).
- ברגע עליית השעון הבא, יהיה 0, $Q_2 = Q_3 = Q_4 = Q_5 = 1$.
- אם 1 = x הערך יישאר אותו דבר כי עדיןמבצע AND עם 0. x בבר לא רלוונטי מהרגע שהוא יד ל-0 פעם אחת. ככל input כבכל עליית שעון הערך 0 יחולחל בין ה-FF-ים.
- ה-stateflow שהוא הוא: 0 → 15 → 7 → 3 → 1 → 31 (ותישאר עם מצב 0). **6 מצבים = 3-FF-ים.**

Timing and Frequencyזמן תגובה:

באשר אנו מדברים על תזמון חיצים למדוד את ה-delay בין t_1 (זמן שבו קיבלנו פלט) ל- t_0 (זמן שבו העברנו את הקלטים). זה **זמן תגובה**: תוך כמה זמן יקרה שינוי בפלט לאחר השינוי בקלט?

למה זה חשוב לנו? אם יש לנו 2 FF (כבה המחשב בנו), אם האות לא מספיק להגיע מ-FF אחד לשני אז המנגל לא עובד בצורה תקינה. אנחנו עושים ניתוח זמינים של מעגלים כדי לדעת אם המנגל שתיכנו הוא תקין. לעיתים הלוגיקה נבונה אבל בשנפעיל את המחשב זה לא יעבוד בהלהה כי האות לא הספיק לחול.

- **זמן תגובה של קלט/פלט:** נגיד סוף ב-תקוף: ברגע שההעולה על מתח 0 (V_{IH}) אנו מחשבים את זה ב-1 לוגי. נגיד גם סוף מתאים עבור ה-פלט: V_{OH}
- **זמן תגובה:** זה זמן התגובה. יש שם לכל סוג מעבר: t_{PLH} מנגום לגובה, $-t_{PHL}$ מגובה לנמוך. אנו מודדים: (max delay) t_{pd} – כמה זמן אנו יכולים לסרוק על התוצאה שלנו שהיא תקפה ולא מושפעת מהבחןήה הקודמת? אחרי כמה זמן היציאה יציבה בערכה החדש? minimum time interval between outputs?
- – כמה זמן לאחר שינוי הכניסה אנו יכולים לקבל פלט?
- (min delay) t_{cd} – כמה זמן לאחר שינוי הכניסה אנו יכולים לקבל פלט?
- – או ערך חדש? minimum time interval between inputs (undefined)

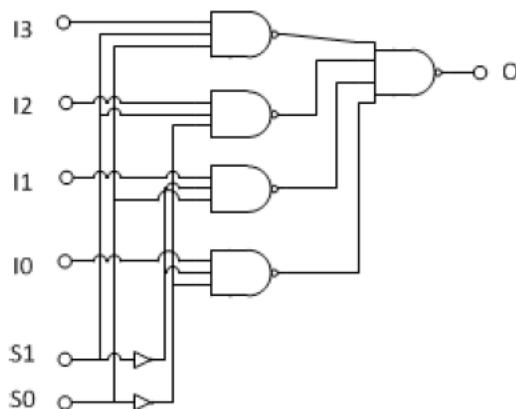


דוגמה: מה זמן התגובה של המעגל? נctrיך לבדוק את כל המסלולים מהבנisa ליציאה, ולקחת את האורך מבניhim (Worst Case). נתון לנו עבור כל רכיב זמן מינימלי t_{cd} ומקסימלי של t_{pd} . המסלול הארוך ביותר הוא 23ns .

Gate	tpd min	tpd max
Inv	2nS	5nS
NAND 3in	4nS	8nS
NAND 4in	5nS	10nS

Min Delay : min of all path :
 $S_0 \rightarrow \text{Inv } 0 \rightarrow \text{Nand3} \rightarrow \text{Nand4} \rightarrow \text{Out}$
 $2 + 4 + 5 = 11\text{nSec}$
 $S_1 \rightarrow \text{Inv } 0 \rightarrow \text{Nand3} \rightarrow \text{Nand4} \rightarrow \text{Out}$
 $2 + 4 + 5 = 11\text{nSec}$
 $I_0 \rightarrow \text{Nand3} \rightarrow \text{Nand4} \rightarrow \text{Out}$
 $4 + 5 = 9\text{nSec}$
 $S_0 \rightarrow \text{Nand3} \rightarrow \text{Nand4} \rightarrow \text{Out}$
 $4 + 5 = 9\text{nSec}$
etc for all possible path to output....

Solution : all path are 11 or 9 \rightarrow Min Delay = 9



Answer : Circuit delay is: Min 9nSec – Max 23nSec

:Latch

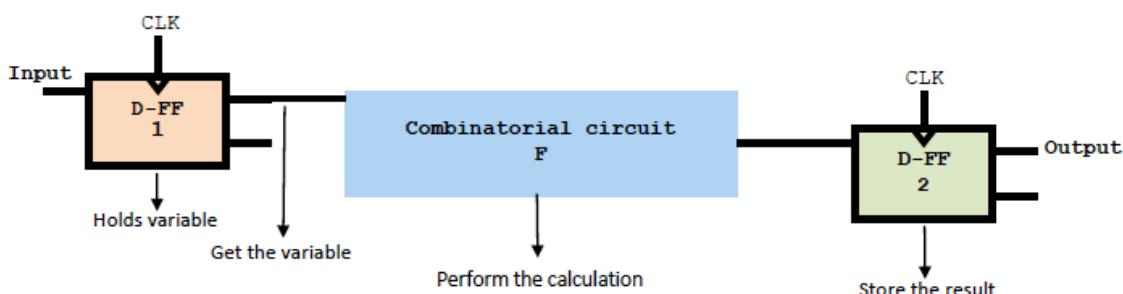
באן החישוב קצר יותר מסובן. כאשר $G = 1$ זה מתנהג כמו לוגיקה צירופית ונוכבל להגדיר לו t_{pd} וגיל – הכל פתוח/שкроף. זה מסתבר באשר נסתכל על **תהליכי הסגירה** של ה-gate. לדוגמה: ה-Data שלנו הוא 0, והוא משתנה ל-1 **משה קרוב לירידת השעון**. S הפע-מ-0 ל-1. R אמרו להתחלף מ-1 ל-0. ברגע נעלמת השעון, מה שה- Latch רואה זה $1 = S = R$ ביוון שלוקח זמן ל- Inverter להפוך את זה ל-0 (יש עיכוב ב-Inversion זהה). יש לנו שתי דרישות:

1. אנו צריכים שה-Data **ויה יציב זמן מסוים לפני נעלמת השעון** – לדרישת זו אנו קוראים t_{setup} (כדי לעמוד בדרישה זו אנחנו צריכים **לדעת מה הרגע הביא מאוחר שבו האות שלו יציב**, ולכן **זה יLER יחיד עם חישובי max**: t_{pd}). השינוי לא יכול לקרות בזמן נעלמת השעון. זה פרק הזמן לפני הנעלמה שבו ה-FF בבר לא יוכל לקבל את ה-**input**. יש לנו דרישת סימטרית – אסור לו-Data להשתנות מהר מדי **אחרי נעלמת השעון**. יכול להיות שהוא שזה צריך יופיע על הנעלמה הנכchia (ולא רק על הנעלמה הבהה שתקרו בהמשך). יש זמן מסוים לאחר נעלמת השעון שבו ה-Data צריך להיות – לדרישת זו אנו קוראים t_{hold} (זה יLER יחיד עם חישובי **min delay**: t_{cd}).
2. יש לנו דרישת שמייניביזיה (Master-Slave) – לזה נקרא t_{valid} וגם לו יש \min (במה זמן אחרי השינוי בזיכרון אפשר לסמור על היזיה). (switch)

:Flip Flops

אותה דרישת כמו ב-Latch. ההבדל הוא, שאין מסלול שкроף מה-D ל-Q. יש לנו t_{setup}, t_{hold} עבור ה-Latch הראשון. **בנוסף**, יש לנו **שינוי ב-Clock שמכתיב שינוי ביציאה** (ה-Master גנעל, ה-Slave נפתח) – לזה נקרא t_{valid} וגם לו יש \min (במה זמן אחרי השינוי בזיכרון, מובטח שהערך הזה ישמר: **keep old value**) ו- \max (במה זמן אחרי השינוי בזיכרון אפשר לסמור על היזיה).

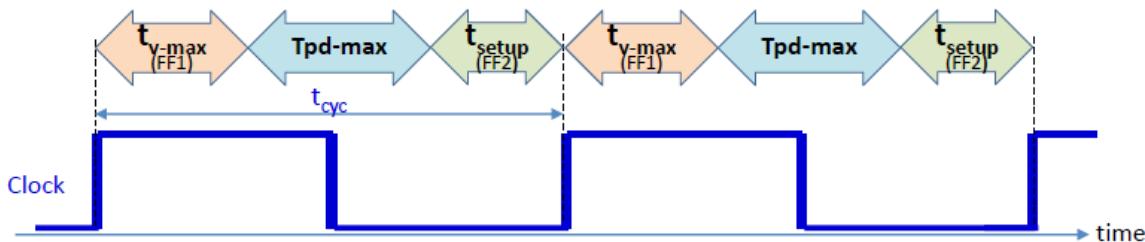
יש לנו סכמה טיפוסית למעגל לוגי: רכיב זיכרון ראשוני FF1 (input), שמייצר ברגע שינוי השעון Data, שנכנס לתוך המעגל. מתבצע חישוב כלשהו, שמחולל לתוך FF2 ונשמר בו. אנו רוצים לדעת אם המעבר בין ה-FF-ים חוקי.





מה הנוסחה לבדיקה האם המעגל עומד בדרישת t_{setup} ? אנו צריכים שהסכום של השינוי ב-FF (t_{value}), זמן החישוב (t_{pd}), ו- t_{setup} , יהיה קטן מזמן המחזור. רק אז, האות שולמו יהיה מוכן מספיק זמן מראש:

- ◆ $t_{cyc} = 1 / f$: the time from one rising edge to the next
- ◆ Must meet :
 - $t_{cyc} \geq t_{v_max} + t_{pd_max} + t_{setup} = t_{path}$
 - FF-1 valid + comb. propagation + FF-2 setup

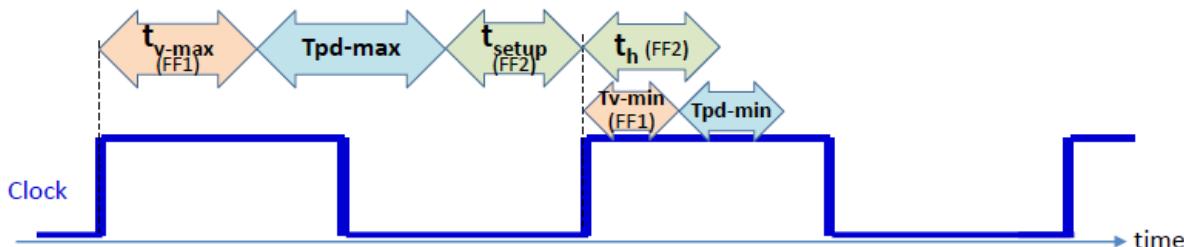


בנוסף, איך לבדוק האם המעגל עומד בדרישת t_{hold} ? זה לא אותו הדבר. אנחנו רוחים לוודא שאחריו השינוי בשעון, השינוי לא מתבצע בצורה מהירה מדי (ערכו היישן יציב זמן מסוים). הכיו מהר השינוי יחולח בזמן $t_{value-min}$ של FF1, זמן החישוב המינימלי t_{pd-min} וזמן t_{hold} צריך להיות קטן מהסכום המינימלי הזה.

◆ What about t_h ?

- Calculate the t_{path_min} for all possible paths (min time for a change):

$$t_{path_min} = t_{v_min}(\text{FF1}) + t_{pd_min}(\text{comb. logic})$$
- We must meet the requirement: $\min\{\text{all } t_{path_min}\} \geq t_h$



חישוב max frequency של מעגל:

◆ Tin (Q/S0) \rightarrow FF setup

$$t_{cyc} \geq t_{v(max)} + t_{pd(max)} + t_{set} = 7 + 23 + 3 = 33$$

◆ Tin (Q/S0) \rightarrow FF hold

$$T_{v(min)} + T_{pd(min)} \geq t_h$$

$$2 + 9 \geq 5$$

\rightarrow hold time requirement met (OK)

◆ Tcycle min = 33nSec

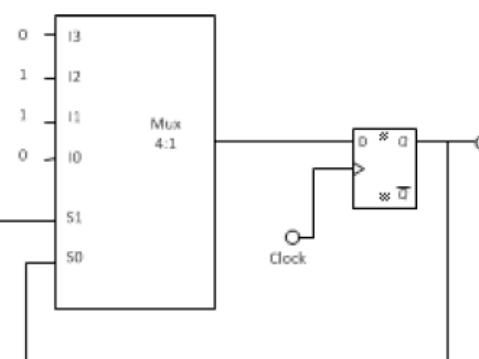
$$F_{max} = 1/33nSec = 30MHz$$

◆ What about the input?

- Need to specify t_v for the input relative to clock and then it can be handled like a signal coming from an output

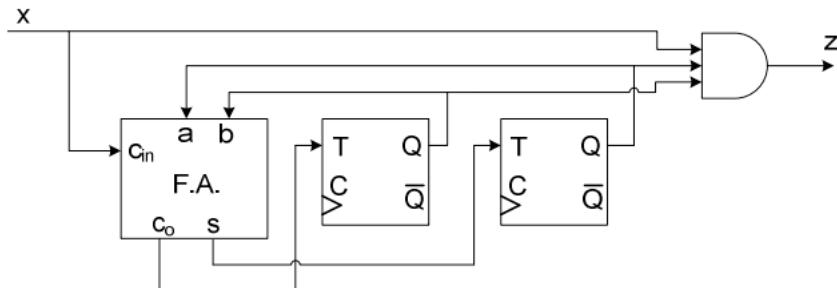
Gate	Parameter	Min	Max	unit
FF	T_v	2	7	nSec
	T_s [input]	3		
	T_h [input]	5		
Mux	T_{pd}	9	23	

בהתאם לדרישת ה- t_{setup} , נכתוב את אי השוויון של זמן המחזור. בבדיקה גם את t_{cyc} ו- t_h .



זמןן מעגלים עם קלט ופלט:

נתונים לנו כל הזמןנים עבור כל הרכיבים במעגל. נתון לנו כי עבור הקלט X מתקיים $t_{v_min} = 0, t_{v_max} = 15$ (מתיחסים אליו בתווך פלט של FF קודם בתחילת). לבדוק האם המעגל עומד בדרישות הבאות על הפלט Z : $t_{setup} = 9, t_{hold} = 0$ (מתיחסים אליו בתווך קלט של FF שקיים בהמשך)?

פתרון:

נעבור על כל **המסלולים האפשריים (רכיב זיכרון (FF) – פונקציה קומבינטורית – רכיב זיכרון (FF))** ובזוויק אם הם תקינים. השיטה הכי טובה לפיה שנראה ברגע זה להתחיל מהסוף ולעקוב אחריה, מכשה נדע אילו זמנים חיברים לכלול, ועל אילו זמנים יש לבצע max (פיזול למספר אפשרויות).

דרישת ה- t_{setup} :

$$\begin{aligned} T1(Z) &= t_{setup} + \max\{t_{pd_and_max} + t_{v_ff1}, t_{pd_and_max} + t_{v_ff0}, t_{pd_and_max} + x_{valid}\} \\ &= 9 + \max\{(3+4), (3+4), (3+15)\} = 9+18 = 27nSec \end{aligned}$$

$$\begin{aligned} T2(T_{ff1}) &= t_{set_ff1} + t_{pd_fa_s} + \max\{x_{valid}, t_{v_ff1}, t_{v_ff0}\} \\ &= 7 + 12 + \max\{15, 4, 4\} = 34nSec \end{aligned}$$

$$\begin{aligned} T3(T_{ff0}) &= t_{set_ff0} + t_{pd_fa_co} + \max\{x_{valid}, t_{v_ff1}, t_{v_ff0}\} \\ &= 7 + 8 + \max\{15, 4, 4\} = 30nSec \end{aligned}$$

- מסלול שmagiu עד Z : בהכרח נגמר ב- Z , ואז יש 3 אפשרויות: מ-FF0, מ-FF1, ומ-X ישירות.
- מסלול שmagiu עד FF1: בהכרח נגמר ב-FF1, י יצא מ-s של ה-FA ואז יש 3 אפשרויות: י יצא מ-FF1, י יצא מ-FF0, י יצא מ-X.
- מסלול שmagiu עד 0: בהכרח נגמר ב-0, י יצא מ-c0 של ה-FA ואז יש 3 אפשרויות כמו קודם.

דרישת ה- t_{hold} : **t_{z_hold} requirement is OnSec:**

$$\begin{aligned} t_{pd_and_min} + \min\{t_{v_x_min}, t_{v_ff_min}\} &= 1 + \min\{0, 0\} = 1nSec \geq 0 \quad (t_{z_hold}) \\ \rightarrow \text{thus we don't have hold-time problem with the output} \end{aligned}$$

FF0 and FF1 inputs' hold is 2nSec:

$$\begin{aligned} t_{pd_fa_min} + \min\{t_{v_ff_min}, t_{v_x_min}\} &= 3 + \min\{0, 0\} = 3 \geq 2 \quad (t_{hold_ff}) \\ \rightarrow \text{thus we don't have a hold-time problem with the flip flops} \end{aligned}$$

דוגמאות מבחן: 2/2019

mobutah sheh kenisah yiziba sec 10achri ultiat shuvon. Drishot ha-setup shel moutzaim hineim

min-timing 0.0sec. Nitn lehniyah shain beuyit min-timing 1. Lmi tdr fuolah goba yoter?

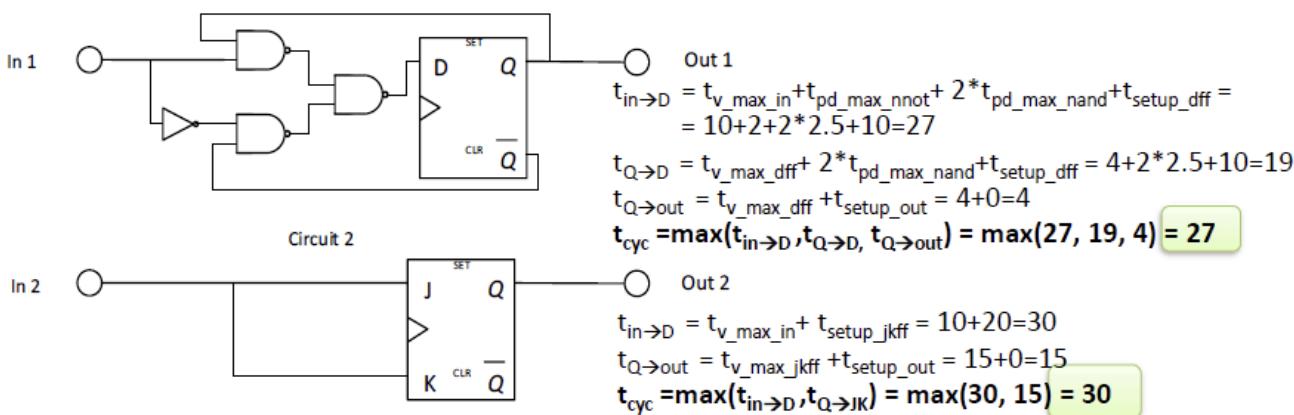
t_{setup} of outputs

All times in [pico-sec]	$t_{\text{PD-min}}$	$t_{\text{PD-max}}$	$t_{v-\text{min}}$	$t_{v-\text{max}}$	t_{hold}	t_{setup}
NOT	1	2				
2 Input NAND	1	2.5				
D Flip Flop			1	4	0	10
JK Flip Flop			1	15	0	20

nbch at cl mssolim shel otot mdgima (Q,in) lsimosh (out, FF_{inputs})

Tshuba: Mugal 1 (f=1/27pSec)

Circuit 1

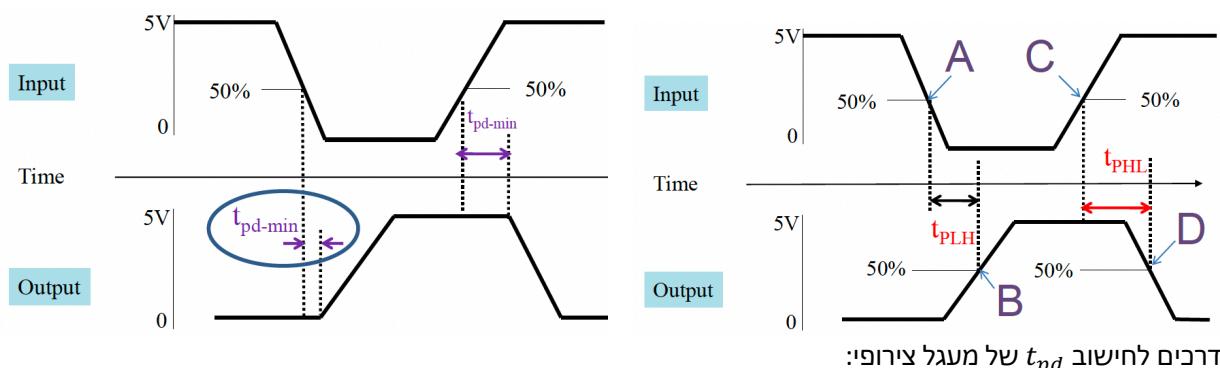


תזמון מגלים (תרגול 4)

maachori ha-mugelim shavnu ud ba umadim shinui matchim (0-l-1, 1-l-0). Lemtach lokh zman lehshatnot vbo tor ha-zman yisheh muver b'matz la-mogder. hiziyot ainin matudbanot midit batgeba leshinim b'kenisot. Nrzta lezmon at ha-rekibim br shivutu lnu shcaher negid shahiziah ha-tzivah ha-uruk ba acen yihya mogder.

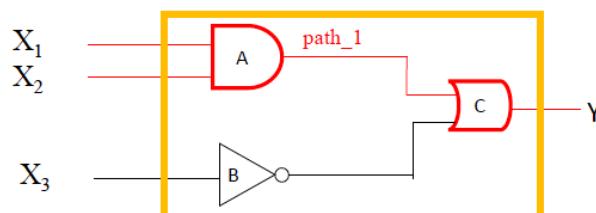
תזמון של מugal翟ופין:

שם	סימן
זמן עלייה (rise)	t_r
זמן ירידת (fall)	t_f
propagation low-to-high	t_{PLH}
היציאה הגעה לאוטו ערך של 50% בעלייה מ-0 ל-1 עד הרגע בו	
mcnisah leziah: mahragu bo ha-kenisa ha-gava la-50% maheur bin 0 l-1 ud hregu bo	
mcnisah leziah: mahragu bo ha-kenisa ha-gava la-50% maheur bin 0 l-1 ud hregu bo	t_{PHL}
המקסימום בין שני הזמנים הקודמיים: $\{t_{PLH}, t_{PHL}\}$. Zeo meshr ha-zman	t_{pd} t_{pd-max}
maheur bin 0 l-1 ud hregu bo	
minimum mahragu b'iztu hashinui b'kenisa bo mobutu lnu shahiziah bar ha-shatnot.	
maheur bo ha-kenisa ha-gava la-50%, ud shahiziah matchila lehshatnot. minimum	t_{cd} t_{pd-min}
bin cel ha-zmimim hallo, yehi zeho meshr ha-zman ha-maksimal mi-biztu shinui b'kenisa bo	
mobutu lnu bi hiziyah temshet ha-shatnot.	
zman shlokha lcl achd ha-rekibim lheragsh shenkens alio match.	



1. דרך מסורבלת: לחפש את המסלול האיטי ביותר בכל ידי סקירת כל הנסיבות האפשריות, וכל שינוי הביטויים האפשריים ומציאת זמן העיכוב הגבוה ביותר. אם ערך הפלט אינו משתנה עבור שינוי בית מסוים, אין צורך בחישוב הזמן.
2. הדרך הנבחרת: לעבד ברמת מסלולים, t_{pd} של המסלול הארוך ביותר יהיה t_{pd} של המעגל כולו.

The function : $X_1 * X_2 + X_3$



נחשב t_{pd_max} (ניקח מקסימום - המסלול הארוך ביותר לפי חיבור זמן t_{pd_max} של הרכיבים במסלול):

Data in ns	X_2	A	B	C
t_{PHL}	-	100	90	80
t_{PLH}	-	110	70	100
t_{ed}	-	12	8	10
t_r	14	20	12	18
t_f	15	17	13	19

$$t_{pd}(\text{path_1}) = t_{pd}(A) + t_{pd}(C) = 110 + 100 = 210 \text{ ns.}$$

$$t_{pd}(\text{path_2}) = t_{pd}(B) + t_{pd}(C) = 90 + 100 = 190 \text{ ns.}$$

$$\Rightarrow T_{pd} \text{ of circuit} = 210 \text{ ns}$$

נחשב t_{pd_min} (ניקח מינימום – המסלול המהיר ביותר לפי חיבור זמן t_{pd_min} של הרכיבים במסלול):

Data in ns	X_2	A	B	C
t_{PHL}	-	100	90	80
t_{PLH}	-	110	70	100
t_{pd_min}	-	12	8	10

$$t_{pd_min}(\text{path_1}) = t_{pd_min}(A) + t_{pd_min}(C) =$$

$$= 12 + 10 = 22 \text{ ns}$$

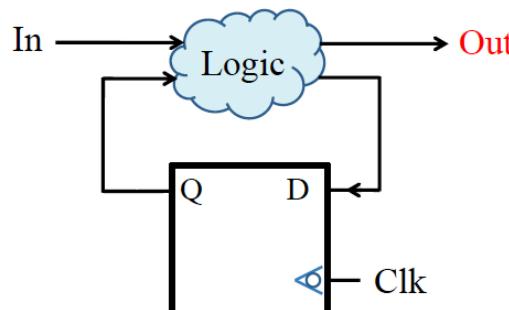
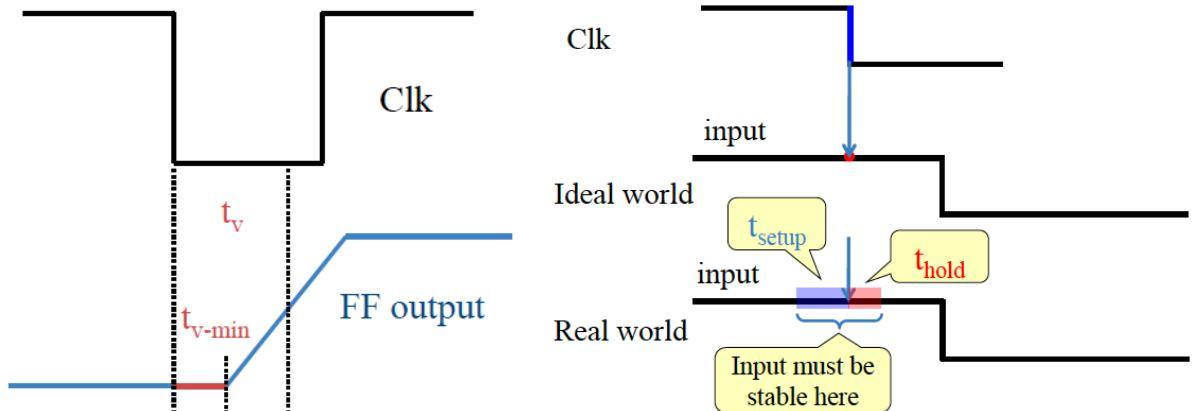
$$t_{pd_min}(\text{path_2}) = t_{pd_min}(B) + t_{pd_min}(C) =$$

$$= 8 + 10 = 18 \text{ ns}$$

זמןון של מעגל סדרתי:

רכיב FF שפועל בעלייה שעון לא יושפע ממשינויים בכניסת המידע המתרחשים בתחום הזמן שבין שתי עלויות שעון, גם אם הערך בכניסת המידע של הרכיב השתנה, הרכיב "אינו רואה זאת" עד לרגע עליית השעון – רק ערך בכניסת המידע ברגע עליית השעון הוא זה שקובע. עליינו להכניס כמה אילוצי זמינים:

סימן	תיאור
t_{setup}	משך הזמן שיש להחזיק את הבנייה של ה-FF קבועה לפני מעבר השעון
t_{hold}	משך הזמן שיש להחזיק את הבנייה של ה-FF קבועה אחרי מעבר השעון
t_v t_{pcq}	משך הזמן ה минимальн בו אנו בטוחים כי אחרי מעבר השעון – היציאות ישתנו ויתיצבו
t_{v-min} t_{ccq}	משך הזמן ה מקסימל בו אנו בטוחים כי אחרי מעבר השעון – היציאה טרם השתנה



- $T_{min-clock}$: מינימום הזמן של מחזור השעון על מנת שנספק לביצוע שינוי ברכיב הדיזנגוף וביציאות בצורה יציבה. אנו דורשים שהיציאה של רכיב הדיזנגוף תהיה יציבה אחרי השינוי, היציאות של המעלג ישתנו, ונחזיק את הבניוסות יציבות:

$$T_{min-clock} = t_{v(1)} + t_{pd}(Logic)(2) + t_{setup}(3)$$

- זמן המתנה: על מנת שהוא יהיה יציב הוא חייב לקיים:

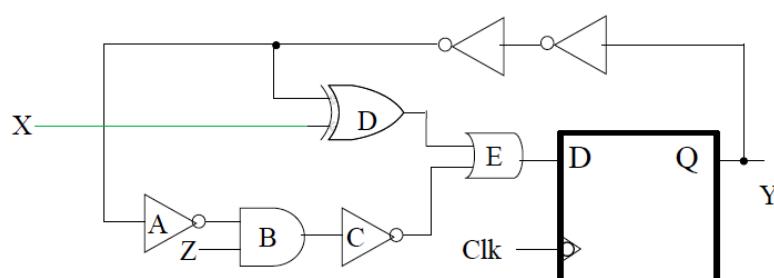
$$t_{hold}(1) \leq t_{v-min}(2) + t_{pd-min}(Logic)(3)$$

- עדמתי ניתן לשנות את הקלט? לפני ירידת השעון, הבנייה D חייבת להיות יציבה ומוכנה לדגימה. הקלט חייב להתייצב עד בזמן $t_{setup} + t_{pd}(Logic)$ לפני ירידת השעון.
- מתי הפלט של המעלג יציב? אחרי ירידת השעון, היציאה של רכיב הדיזנגוף תשתנה. כמה זמן צריך לחכות עד שהיציאה מתהייצבת?
 $t_v + t_{pd}(Logic)$.

- **להוסיף DELAY עיי הוספת שערים במסלול**

הבעיתי! (הוספנו שני שער NOT שבזמנים לא עושים דבר)

- **חסרונו : תדר העבודה ירד!**





אנליזה וсинטזה של מעגלים סינכרוניים (תרגול 5)

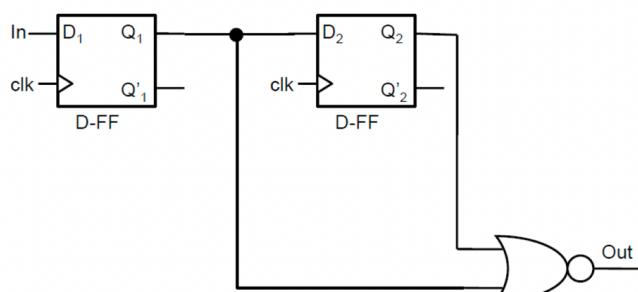
הקדמה:

סוג	מעגל צירופי	מעגל סינכרוני
אנליזה	אפשר בקלות לבנות את היציאות שלו כפונקציות של הרכיבים המבנה את המעגל וננתה לו.	מבצע תהליכי טכני שבסופו קיבל מוכנת מצבים (Mealy או Moore) המתארת את המעגל וננתה לו.
סינטזה	כတבנו טבלת אמת מתאימה, הוצאים ממנה את הביטוי האופטימלי שמייצג את טבלת האמת באמצעות מפורת קרנו, ועל פי הפונקציה הבוליאנית שקיבלנו מימשנו את המעגל.	צייר מוכנת מצבים שמתארת את התנהגות המעגל, ולאחר מכןמבצע תהליכי טכני שבסופו קיבל תיאור מלא של המעגל ונצייר אותו.

אנליזה:

נתון מעגל בleshו ורוצים להבין מה הוא עושה (לנתח את התנהגותו):

1. הצגת הרכישות לרכיבי הזיכרון ויציאות המעגל (בעזרת היציאות של רכיבי הזיכרון והרכישות למעגל). בשלב זה נראה האם הפלט שלנו תלוי ברכישות (Mealy) או רק בזכרון (Moore).
2. טבלת מעברים.
3. טבלת מצבים סימבולית – כינוי המצבים בשמות (A, B).
4. אוטומט מצבים.
5. ניתוח האוטומט: בעזרת סדרת בוחן.

דוגמה לאנליזה (Moore):שלב 1:

$$\begin{aligned} D_1 &\leftarrow In & \bullet \\ D_2 &\leftarrow Q_1 & \bullet \\ Out &\leftarrow (Q_1 + Q_2)' & \bullet \end{aligned}$$

במקרה זה ניתן לראות כי הפלט תלוי רק בזכרון (Moore).

שלב 2:

	In=0		In=1		
	Q ₁	Q ₂	D ₁	D ₂	Out
0	0	0	0	0	1
0	1	0	0	0	0
1	0	0	1	0	1
1	1	0	0	1	0

נפריד עבור 0 ו-1 בכניסה: הם משפיעים ישירות על D_2 , D_1 וכן נמלא את הערכים הללו. את ערכי D_2 נקבל מ- Q_2 .שלב 3:

	NS		
PS	In=0	In=1	Out
A	A	C	1
B	A	C	0
C	B	D	0
D	B	D	0

	שמות המצבים		
	Q ₁	Q ₂	
A	0	0	
B	0	1	
C	1	0	
D	1	1	

ניתן שמות לכל קומבינציה של Q_2 , Q_1 וככנה אותן: D, C, B, A. נחליף כל רצף בקידוד המצב שלו.

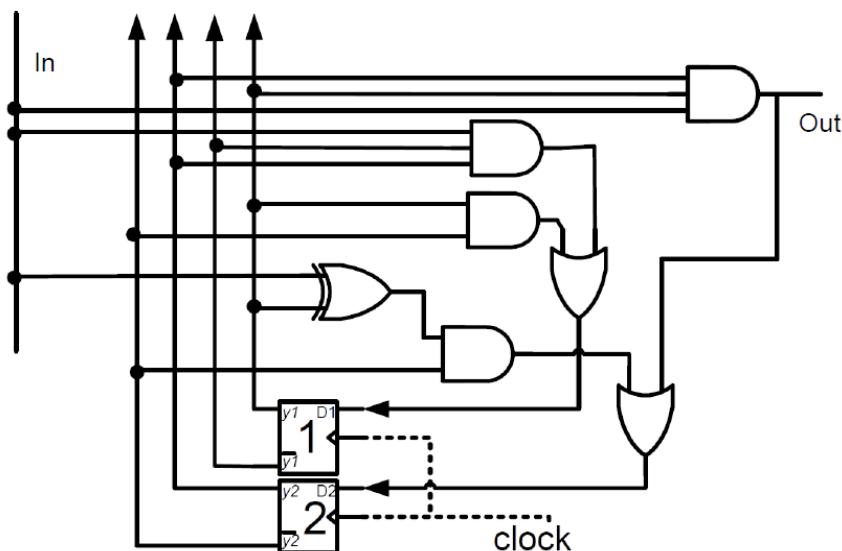
שלב 4:

ראינו כבר כי מדובר במבנה Moore. לכן, בכל מצב באוטומט אנחנו נכתוב State, Out (אין קשר לקלטים). את ה-state אנו אוחנו **ושמיים על המצב ממנו יצאנו**. למשל מ-A ל-C ה-input הוא 1 וה-output הוא 1 (רשום בתוך העיגול של A).

שלב 5:

In	0	0	1	0	0	1	1	0	1	0	1	0	0	0
State	A	A	A	C	B	A	C	D	B	C	B	C	B	A
Out	φ	φ	1	0	0	1	0	0	0	0	0	0	0	1

- ננתה את האוטומט בעזרת סדרת בוון. נניח שהמתחילו מהמצב A – המטריה היא לעבור בכל מצב לפחות פעם אחת.
- סדרה זו מראה שכאשר מופיעה המחרוזת 0,0 בזמנים $t, t+1$ המעגל פולט 1 בזמן $t+2$: כיוון שזו מבנה Moore **השינוי לא יקרה באותו מחזור, המוכנה תפלוט 1 אחרי קליטה של פעמיים 0**.
- נשים לב כי בהתחלה הכנסנו פעמיים 0 והפליט שלנו הוא φ. ה-state'ו שלנו לא מוגדר בזווית שיש לנו שני FF-ים בדוגמה שלנו. במצב ההתחלתי, אנחנו לא יודעים מה יש בזיכרון, ולכן לא נוכל לדעת מה יהיה ה-output. **שני מחזורי השעון הראשונים לא מוגדרים**.

דוגמה נוספת של אנליזה (Mealy):שלב 1:

– **הציגת הבניות לריבבי הזיכרון**, בעזרת היציאות של רכיבי הזיכרון והבניות למעגל.

- $D_1 \leftarrow y_1 \cdot y_2 + In \cdot y_1' \cdot y_2'$
- $D_2 \leftarrow y_2' (In \text{ XOR } y_1) + In \cdot y_1 \cdot y_2$

– **הציגת יציאות המעגל**, בעזרת היציאות של רכיבי הזיכרון הכניסות למעגל.

- $Out \leftarrow In \cdot y_1 \cdot y_2$

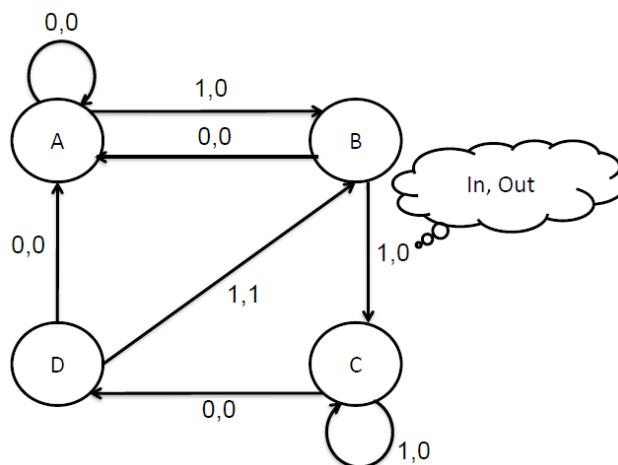
נעקב אחרי החיבורים במעגל ונסיק מה מגיע ל- D_1, D_2 (נכך מהם אחרה ונעקב), ומה מגיע ל- Out .

שלב 2 :3 + 2

	NS,Out	
PS	In=0	In=1
A	A,0	B,0
B	A,0	C,0
C	D,0	C,0
D	A,0	B,1

	In=0		In=1					
	y_1	y_2	D_1	D_2	Out	D_1	D_2	Out
0,0	0	0	0	0	0	0	1	0
0,1	0	1	0	0	0	1	0	0
1,0	1	0	1	1	0	1	0	0
1,1	1	1	0	0	0	0	1	1

יש כאן עוד עמודה נפרדת ל-Out עברו כל תז – כיון שכאן אנחנו במבנה Mealy.

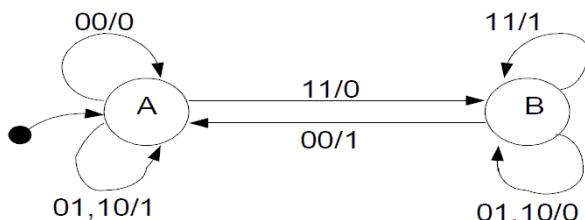
שלב 4 :

בכל מעבר בין מצבים באוטומט, על כל קשת נרשום *Input, Output*. בתור כל מצב לא נרשום את הפלט, רק את המצב.

שלב 5 :

In	1	1	0	1	1	0	1	0	1	1
State	A	B	C	D	B	C	D	B	A	B
Out	Φ	Φ	0	1	0	0	1	0	0	0

הפעם, נראה שכאשר מופיעה המחרוזת 1,1,0,1,0,1, t, t+1, t+2, t+3 בזמן $t+3$.

סינזה:

נתונה ההתנגדות הרציה של המעגל, רצים למשם אותו:

1. בניית אוטומט בהתאם לדרישות.
2. טבלת מצבים (כולל A, B ו-Zero).
3. קידוד מצבים (למשל A זה 0, B זה 1) ובבחירה סוג FF.
4. טבלת מעברים ופלט.
5. הגדרת פונקציות הכניסה של ה-FF-ים ויציאת המעגל.
6. בניית (שרוטט) המעגל.

NS (Next State), z (Output)

ראינו דוגמה למסכם בינהי טורי.

PS (Present State)	Input			
	00	01	11	10
A	A,0	A,1	B,0	A,1
B	A,1	B,0	B,1	B,0



2 – ארכיטקטורת מחשב

MIPS ISA

מבוא

המחשב מורכב ממעבד (יחידת החישוב) ו邏輯 (data לעיבוד, והתוכנה עצמה – הפקודות). היזירון הוא מערך של גודל של FF-ים (ביטים). אנו עובדים עם גודל שנקרא Word וקוראים/כותבים את כלו. גודל זה שווה ל-4 בתים (כלומר 32 ביטים). כאשר נרצה לкопא לכתובת הבאה בזיכרון נבצע תוספת של 4 בתים (32 ביטים) לכתובת.

.instruction Fetch Unit לוקחת מהזיכרון פקודות לביצוע: כל פקודה היא בגודל מילה. שמורה לנו הכתובת הנוכחית של ה-.operation + operands increment | immediate (4+4) כדי להציג לפקודה הבאה בזיכרון. לפקודה יש תבנית ברורה: זיכרונו יוסיף של FF-ים (כל אחד הוא אוסף של 32 ביטים): זיכרונו עבודה שהוא בתוך המעבד.

אנו כותבים קוד בשפת level high: למשל **C**. לאחר מכן, ישנו הקומpileר שמרתגם את הקוד לשפת **asm**. האסambilר מתרגם את הפקודות לקוד **binario**. ה-linker מחבר חלקים שונים של תוכנה לפורמט שנוכל לטעון ל זיכרון. תרגום של אסambilר לשפת מכונה הוא 1:1. **MIPS** הוא **RISC** – ממומש באמצעות פקודות פשוטות (בניגוד ל-CISC).

ISA

באסambilר המשתנים הם רגיסטרים. יש 32 רגיסטרים הממוספרים מ-0 עד 31. לכל אחד מהם יש תפקיד, יש קונבנצייה לשימוש בו. הפעולות מתחזקות אך ורק דרך הרגיסטרים. כל רגיסטר מורכב מ-32 ביטים (מילה).

Name	Register number	Usage	Preserved on call?
\$zero	0	the constant value 0	n.a.
\$v0-\$v1	2-3	values for results and expression evaluation	no
\$a0-\$a3	4-7	arguments	no
\$t0-\$t7	8-15	temporaries	no
\$s0-\$s7	16-23	saved	yes
\$t8-\$t9	24-25	more temporaries	no
\$gp	28	global pointer	yes
\$sp	29	stack pointer	yes
\$fp	30	frame pointer	yes
\$ra	31	return address	yes

FIGURE 2.18 MIPS register conventions. Register 1, called \$at, is reserved for the assembler (see Section 2.10), and registers 26–27, called \$k0–\$k1, are reserved for the operating system.

כל פקודה באסambilר מבצעת פעולה אחת, מסוימת:

דוגמאות	סוג
арitmetyka – חיבור חיסור וכו' add \$s0, \$s1, \$s2 sub \$s0, \$s1, \$s2 addi \$s0, \$s1, 30 addi \$s0, \$s1, -30 (subi doesn't exist)	
ذיכרון – טעינה מדיכרון לרגיסטר (load) או שטירה מרגיסטר ל זיכרון (store). כל העבודה עם הכתובות היא ב-bytes. ביןיהם שעדים עם מילום (4 בתים), בלי offset לנכץין יהיה בפולה של 4: 4 בתים שבור מילה 1 (האינדקס הראשון במערך), 8 בתים עבור 2 מילים (האינדקס השני במערך). אנו משתמשים ברגיסטר בתור מצביע (מכיל לנו בתובת זיכרון, לא את הערך עצמו).	
החלטה/ברכה – פקודות שנוגעות ב-current address וממשאות לוגיקה של if-else. ניתן גם להשתמש בפקודה slt שמבצעת השמה ל-1 אם תנאי מתקיים, אחרת 0. j Label beq reg1, reg2, Label bne reg1, reg2, Label slt rd, rs, rt (if rs < rt) rd = 1; else rd = 0; slli rt, rs, constant	



דוגמה לימוש לולה:

- ◆ C code: // array is array of integers (32bits elements)


```
i = 0;
while (array[i] != 0) i = i + 1;
```
- ◆ Assigned registers for our MIPS code:


```
Address of array is in register $a0
i will be in register $v0
```
- ◆ MIPS assembly:


```
add $v0, $0, $0          // i = 0
loop:
    lw $t0, 0($a0)        // temp0 = array[i]
    beq $t0, $0, finish   // if array[i] == 0 finish
    addi $v0, $v0, 1       // i = i + 1
    addi $a0, $a0, 4       // advance $a0 to next int
    j loop
finish:
```

Calling Convention

קונבנציית הקריאה לפונקציה:

- Push \$s0 to stack: `addi $sp, $sp, -4`
`sw $s0, 0($sp)`
- Pop from stack into \$s0: `lw $s0, 0($sp)`
`addi $sp, $sp, 4`
- Load the 4th int from stack: `lw $t0, 12($sp)`

- הפרמטרים – מועברים דרך `$a0, ..., $a3`
- ערך החזרה – מועבר דרך `$v0, $v1`
- **רגיטרים שהfonקציה יכולה לשנות - ו-** `$t0, ..., $t9`
- **רגיטרים שהfonקציה לא יכולה לשנות -** `$a0, ..., $a3`
- בתובת החזרה היא נשמרת ב-`$ra`
- מצביע המחסנית נשמר ב-`$sp` (מי שמבצע push למחסנית חייב לבצע pop)

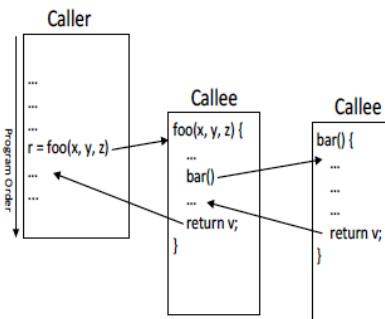
ישנו מקום מיוחד בזיכרון שנקרא **מחסנית**. יש לנו stack pointer שמצוין למקומ הנוכחי במחסנית (ה-top), וכרגע פעולות של `push`, `pop`, `addi` או `sw` מושגут באמצעות `lw` ו-`addi` ו-`sw` ו-`lw` ביחס ל-`$sp`. **אנחנו שומרים במחסנית** דברים נחוצים עבור הפונקציה. דוגמאות לעובדה עם המחסנית (המחסנית גדולה באשר אנו מורדים את ערכו של `$sp` – **המחסנית היא הפוכה**).

שתי פקודות רלוונטיות שמשתמשות ב-`$ra` בהקשר זה:

```
// int foo(int x, y, z)
foo:
    addi $sp, $sp, -12           // make room for $s0, $s1, $ra
    sw $s0, 0($sp)              // save $s0 value to the stack
    sw $s1, 4($sp)              // save $s1 value to the stack
    sw $ra, 8($sp)              // save $ra value to the stack
    ...
    // foo changes $s0 and $s1
    ...
    // foo calls another function - bar
    jal bar

    ...
    lw $s0, 0($sp)              // restore $s0 from stack
    lw $s1, 4($sp)              // restore $s1 from stack
    lw $ra, 8($sp)              // restore $ra from stack
    addi $sp, $sp, 12            // deallocate space from the stack
    jr $ra                      // return

bar:
    addi $sp, $sp, -4           // make room for $s0
    sw $s0, 0($sp)
    ...
    // bar changes $s0
    ...
    lw $s0, 0($sp)              // restore $s0 from the stack
    addi $sp, $sp, -12           // deallocate space from the stack
    jr $ra
```



- מוצעת **jal \$ra, label** jump and link קופצת בתובת החדש (label) ושומרת את בתובת הפוקודה הבאה בתוור כתובת החזרה (`ra`). כדי לשומר את בתובת החזרה שאחננו צירבים לחזור אליה לפני קרייה לפונקציה בפועל push לערך של `ra` הנוכחי (כדי לא לאמוד אותו לפני שהוא ידרס).

- מוצעת **jr \$ra** jump register. השתמש בכך כדי לחזור בסיום הפונקציה למי שקרה לנו (השומרה ב-`$ra`). דוגמה (בסוף הפונקציה `bar` צריך לבצע 4 `addi` ולא 12).



שפת מוכנה

ישנו מיפוי בין אסמלבי לשפת המוכנה – אוסף של הוראות המובן בצורה ישירה על ידי המעבד. הוראות שפת המוכנה הוא רצף ביטים שנקרא קוד בינארי. ישנו שלושה מבנים של פקודות:

- add \$s0, \$s1, \$s2
- sub \$s0, \$s1, \$s2

- addi \$s0, \$s1, 30
- lw \$t0, 8(\$s2)
- sw \$t0, 8(\$s2)
- beq \$s3, \$s4, True
- bne \$s3, \$s4, False

- j Exit

→ R-Type

sources and dest are registers

→ I-Type

Instructions that require immediate values

→ J-Type

Jump instructions

Format	Encoding	Example																																				
R-Type (Register) 3 רגיסטרים	<table border="1"> <thead> <tr> <th>opcode</th><th>rs</th><th>rt</th><th>rd</th><th>shamt</th><th>funct</th></tr> </thead> <tbody> <tr> <td>31-26</td><td>25-21</td><td>20-16</td><td>15-11</td><td>10-6</td><td>5-0</td></tr> <tr> <td>6</td><td>5</td><td>5</td><td>5</td><td>5</td><td>6</td></tr> </tbody> </table> <p>opcode – always 000000 for r-type rs/rt – source registers rd – dest register shamt – shift amount (for shift operations) funct – specific operation (add/sub/or/and)</p> <p>.קדם רושמים את rd (היעד), ואז את rs, rt (הקלטים).</p>	opcode	rs	rt	rd	shamt	funct	31-26	25-21	20-16	15-11	10-6	5-0	6	5	5	5	5	6	<p>Example: add \$8, \$16, \$17</p> <table border="1"> <thead> <tr> <th>opcode</th><th>rs</th><th>rt</th><th>rd</th><th>shamt</th><th>funct</th></tr> </thead> <tbody> <tr> <td>0</td><td>16</td><td>17</td><td>8</td><td>0</td><td>32</td></tr> <tr> <td>000000</td><td>10000</td><td>10001</td><td>01000</td><td>00000</td><td>100000</td></tr> </tbody> </table> <ul style="list-style-type: none"> Add is funct 32 (table in the book) <p>Assembly: add \$8, \$16, \$17 = Machine code: 0x02114020</p>	opcode	rs	rt	rd	shamt	funct	0	16	17	8	0	32	000000	10000	10001	01000	00000	100000
opcode	rs	rt	rd	shamt	funct																																	
31-26	25-21	20-16	15-11	10-6	5-0																																	
6	5	5	5	5	6																																	
opcode	rs	rt	rd	shamt	funct																																	
0	16	17	8	0	32																																	
000000	10000	10001	01000	00000	100000																																	
I-Type (Immediate) 2 רגיסטרים ומספר	<table border="1"> <thead> <tr> <th>opcode</th><th>rs</th><th>rt</th><th>immediate</th></tr> </thead> <tbody> <tr> <td>31-26</td><td>25-21</td><td>20-16</td><td>15-0</td></tr> <tr> <td>6</td><td>5</td><td>5</td><td>16</td></tr> </tbody> </table> <p>נשים לב כי בעת כתיבת הפקודה אנו קודם רושמים את rt, ורק אז את rs ואת ה-offset של rs (זה ה-immediate). (זה ה-immediate).</p>	opcode	rs	rt	immediate	31-26	25-21	20-16	15-0	6	5	5	16	<p>Ex: addi \$9, \$16, -34</p> <table border="1"> <thead> <tr> <th>opcode</th><th>rs</th><th>rt</th><th>Immediate</th></tr> </thead> <tbody> <tr> <td>8</td><td>16</td><td>9</td><td>-34</td></tr> <tr> <td>001000</td><td>10000</td><td>01001</td><td>1111111111011110</td></tr> </tbody> </table> <p>Ex: lw \$9, 32(\$16)</p> <table border="1"> <thead> <tr> <th>opcode</th><th>rs</th><th>rt</th><th>Immediate</th></tr> </thead> <tbody> <tr> <td>0x23</td><td>16</td><td>9</td><td>+32</td></tr> <tr> <td>100011</td><td>10000</td><td>01001</td><td>0000000000100000</td></tr> </tbody> </table>	opcode	rs	rt	Immediate	8	16	9	-34	001000	10000	01001	1111111111011110	opcode	rs	rt	Immediate	0x23	16	9	+32	100011	10000	01001	0000000000100000
opcode	rs	rt	immediate																																			
31-26	25-21	20-16	15-0																																			
6	5	5	16																																			
opcode	rs	rt	Immediate																																			
8	16	9	-34																																			
001000	10000	01001	1111111111011110																																			
opcode	rs	rt	Immediate																																			
0x23	16	9	+32																																			
100011	10000	01001	0000000000100000																																			
J-Type (Jump)	<table border="1"> <thead> <tr> <th>opcode</th><th>target</th></tr> </thead> <tbody> <tr> <td>31-26</td><td>25-0</td></tr> <tr> <td>6</td><td>26</td></tr> </tbody> </table> <p>nextPC = nextPC & 0xf0000000 target * 4</p> <p>כלומר, אנו מאפסים את כל 28 הביטים התחתיות של הכתובת, נשאיר רק את ה-4 העליונים (זה ה-page, המספר שלו נשאר קבוע). ואז ניקח את הכתובת של target ונכפיל ב-4 (איפסנו את שני הביטים הראשונים) וביצע OR. אנו לוקחים את ה-target ומלבישים את זה על ביטים 2 עד 28 ב-PC.nextPC.</p>	opcode	target	31-26	25-0	6	26	<p>Ex: j 1024</p> <table border="1"> <thead> <tr> <th>opcode</th><th>target</th></tr> </thead> <tbody> <tr> <td>0x2</td><td>256</td></tr> <tr> <td>000010</td><td>00000000000000000000000000000000100000000</td></tr> </tbody> </table>	opcode	target	0x2	256	000010	00000000000000000000000000000000100000000																								
opcode	target																																					
31-26	25-0																																					
6	26																																					
opcode	target																																					
0x2	256																																					
000010	00000000000000000000000000000000100000000																																					



(תרגול 6) MIPS

סוגי הרגיסטרים ב-MIPS:

המשתנים באסמבלי הם ורגיסטרים – כל רגיסטר הוא אוסף של FF. יש 32 רגיסטרים, כל אחד מורכב מ-32 ביטים (4 בתים), אשר נקראים מילה (word).

שם	רגיסטר	שימוש
\$zero	0	תמיד מחזיר ערך אפס (לא באמת רגיסטר)
\$v0 - \$v1	2 - 3	תוצאות חישובים שונים ורצים להחזיר למחזיר return
\$a0 - \$a3	4 - 7	משמשים להעברת ארגומנטים למתחוזות
\$t0 - \$t7	8 - 15	משתנים זמינים למתחוזות מיותר לדפוס
\$s0 - \$s7	16 - 23	משתנים לשימוש. למתחוזות אסור לדפוס אותם (כלומר, חייבות לוודא שהם מוכלים את ערכם המקורי לפני סיום המתחודה).
\$t8 - \$t9	24 - 25	משתנים זמינים נוספים
\$gp	28	יכול להציג ערך מסוים בזיכרון
\$sp	29	מצביע לראשית המחסנית.
\$fp	30	שומר על מבנה של המתחודה
\$ra	31	מצביע לפקודה אליה יש לחזור בסיום ריצת מתחודה.

הפקודות השונות ב-MIPS:

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	three register operands
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	three register operands
Data transfer	load word	lw \$s1,100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Data from memory to register
	store word	sw \$s1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Data from register to memory
Logical	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \$s3$	three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim (\$s2 \$s3)$	three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,100	$\$s1 = \$s2 \& 100$	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,100	$\$s1 = \$s2 100$	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Shift left by constant
	shift right logical	srl \$\$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Shift right by constant
Conditional branch	branch on equal	beq \$s1,\$s2,L	if($\$s1 == \$s2$) go to L	Equal test and branch
	branch on not equal	bne \$s1,\$s2,L	if($\$s1 != \$s2$) go to L	Not equal test and branch
	set on less than	slt \$s1,\$s2,\$s3	if($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than; used with beq, bne
	set on less than immediate	slti \$s1,\$s2,100	if($\$s2 < 100$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than immediate; used with beq, bne
Unconditional jump	jump	j L	go to L	Jump to target address
	jump register	jr \$ra	go to \$ra	For procedure return
	jump and link	jal L	$\$ra = PC + 4$; go to L	For procedure call

- פקודת sll (shift left logic): בוצע $\$s1, \$s2, x \ll z$ כאשר המשמעות היא $x \ll z = \$s1 = \$s2 \cdot 2^x$ או במילים אחרות אנחנו מכפילים בחזקת 2 המתאימה: $\$s1 = \$s2 \cdot 2^x$. ביצוע $\ll z$ ב-2, ביצוע הכפלת ב-4.
- פקודת sra (shift right arithmetic): הסיביות שהו בצד ימין נעלמות לנו. נרף בצד שמאל בהתאם לסיבית השמאלית ביותר. אם המספר מתחילה ב-0 נרף ב-0, אחרת ב-1.

תרגום הפקודות לשפת מכונה:

הפקודות מתחולקות ל-3 סוגים:

- R-Type: פעולות בין רגיסטרים.

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
opcode	rs	rt	rd	shamt	funct
31-26	25-21	20-16	15-11	10-6	5-0

add \$4, \$3, \$2 000000 00011 00010 00100 00000 100000

- I-Type: פעולות עם immediate, מיוצגות בשיטת המשלים ל-2.

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate
31-26	25-21	20-16	15-0

lw \$5, 8(\$6) 100011 00110 00101 0000 0000 0000 1000

- J-Type: פקודות קפיצה (ה-offset נתן ב밀ימ' – עבר כתובת 4000 כתוב 1000).

6 bits	26 bits
opcode	offset – target address
31-26	25-0

j 15 000010 00 0000 0000 0000 0000 0000 1111

דוגמאות לשימוש בפקודות:

a = b - A[6]; (in C)

גישה לבתוות בזיכרון (תמיד ה-offset שלנו יהיה כפולה של 4, כי כל איבר=מילה=4 בתים).

a : \$s0, b : \$s1, base of A : \$s2.

לולאה:

lw \$t0, 24(\$s2) # \$t0 = A[6]

נתרגם תחיליה לפסאודו-קוד. במקום התנינה עם if goto .

sub \$s0, \$s1, \$t0 # \$s0 = b - A[6]

נקצה לכל אחד מה משתנים רגייסטר.

נכפיל את ה-i – שלנו ב-4 (2 ללס), ונוסף לבתוות הבסיס שהוא A.

נתען את האיבר במערך באמצעות \$i.

בבצע add עבר g, וגם עבר i, ובבדוק את

התנאי באמצעות bne.

טיפול באוי שווין:

```

do {
    g = g + A[i];
    i = i + j;
} while (i != h);

```

“i” represents the loop index, and we
#want to loop through the addresses in
#Bytes (not words)

Loop: sll \$t1,\$s3,2 # \$t1 = 4*i
add \$t1,\$t1,\$s5 # \$t1 = addr of A[i] if (g<h) goto Label; # g:\$s0, h:\$s1
lw \$t1,0(\$t1) # \$t1 = A[i]
add \$s1,\$s1,\$t1 # g = g + A[i]
add \$s3,\$s3,\$s4 # i = i + j
bne \$s3,\$s2,Loop # go to L1 if i!=h slt \$t0,\$s0,\$s1 # \$t0 = 1 if g<h
bne \$t0,\$0,Label # goto Label

g	h	i	j	Base of A
\$s1	\$s2	\$s3	\$s4	\$s5





פונCTION קוד (נבדוק ערך א כלשהו ונקוב אחריו הפקודות):

```

begin: addi $t0, $zero, 0 # $t0=0
        addi $t1, $zero, 1 # $t1=1
loop:   slt $t2, $a0, $t1 # If n<$t1 then $t2=1 else $t2=0
        bne $t2, $zero, finish # If n<$t1 then goto finish
        add $t0, $t0, $t1 # $t0 = $t0 + $t1
        addi $t1, $t1, 2 # $t1 = $t1 + 2
        j loop # goto loop
finish: add $v0, $t0, $zero # $v0 = $t0

```

- $n = 12$:
- $1 + 3 + 5 + \dots + 11$

סכום המספרים הא-זוגיים מ-1 עד n .

המרת קוד:

- המשתנים יהיו $a = \$a0, b = \$a1, value = \$t0, A = \$s0$
- נשים לב שעבור בדיקת התנאי $a < 0$ אנחנו שמים את **הערך הבוליאני של ההשוואה באמצעות slt**, ואז עושים **beq** אל **מול 0**, כלומר נבצע קפיצהalla רק אם קיבלנו 0 (שקר). מיד אחר כך נרשום את גוף ה-if עצמו.

int mult(int a, int b) {

Begin: add \$t0, \$zero, \$zero

int value;

slt \$t1, \$a0, \$zero

value = 0;

beq \$t1, \$zero, Loop

if ($a < 0$) {

sub \$a0, \$zero \$a0

$a = -a$;

sub \$a1, \$zero \$a1

$b = -b$;

Loop: beq \$a0, \$zero, Finish

}

add \$t0, \$t0, \$a1

while ($a \neq 0$) {

sub \$a0, \$a0, 1

value += b;

j Loop

$a--$;

}

A[4] = value;

Finish: sw \$t0, 16(\$s0)

}

Calling Convention

תוכנות הרצות על MIPS משתמשות בחלוקת הבאה:

- $.\$a0 - \$a3$: וегистרים לפרמטרים לפונקציה (arguments).
- $.\$t0 - \$t3$: וегистרים עבור ערכי החזרה מהפונקציה (values).
- $.\$ra$: רגיסטר אשר זכר לאן יש לחזור בסיום ביצוע הפונקציה (return address).

פקודות:

- מבצעת גם jump לכתובת מסוימת, וגם שומרת את כתובת החזרה ב-\$ra – jal
- jr \$ra – jr משמש לחזרה מהפונקציה, מבצע jr \$ra – jr



נדיר קונבנצייה איחודית עברו מישוקה לפונקציה (caller) ובעור הפונקציה שנקרהת (callee):

callee	caller	
אם משתמש ברגיסטרים שמורים (a), לפני השימוש בהם צריך לשמור אותם למחסנית ולשוחזר בהמשך לפני סיום הפונקציה.	mobutah_shishmeho_rgistrim_shumorim(a). uber_beshar_arum_ho_rachah_lehashmush_benhem, ho_crir_leshemor_lmehsinit_leshazarot_benhem_barhemshar_lesim_fonktsiya.	רגיסטרים שמורים
אם יש קריאה לעוד פונקציה פנימית, צריך לשמר למחסנית את \$ra ולשוחזר לפני סיום הפונקציה.	zogag_lehabur_prmtrim_lefonktsiya_shaho_kora_lahe_b-\$a3-\$a0(\$a0)_(nmehsinit_bmidat_hatzor)	קריאה לפונקציה
חווד ל-caller באמצעות \$ra	koraa_lefonktsiya_bamatzuot_jal	קפייה
שם את ערכו החזרה ברגיסטרים \$t1-\$t0-\$u0-\$u1.	uravi_chzora_shel_fonktsiya_imzao_barrgistrim \$t1-\$t0-\$u0-\$u1.	ערכי חזרה

Leaf_example:

```
addi $sp, $sp, -4
sw $s0, 0($sp)
add $t0, $a0, $a1
add $t1, $a2, $a3
sub $s0, $t0, $t1
add $v0, $s0, $zero
lw $s0, 0($sp)
addi $sp, $sp, 4
jr $ra
```

דוגמא – callee

- תחילה, מקצת מקום במחסנית לאיבר אחד (4 בתים) ושומר את הרגיסטר \$s בזווון שהוא משמש בו (הוא שומר لكن צריך לשמר ולשוחזר אותו אחר כר).
- מניחים שהארגומנטים ניתנו ברגיסטרים הייעודיים (a), וכן נוכל להשתמש ברגיסטרים הזמןניים (t).
- כדי להחזיר את הערך הסופי נעתק אותו לרגיסטר \$v.
- לבסוף, נשוחר את הערך של \$s מוקדם.
- נקפוץ לבתוות החזרה.
- במדהה יש קריאה לעוד פונקציה בתחום callee, callee צריך לשמר למחסנית את הרגיסטר החזרה לפונקציה של הפונקציה הקוראת ולשוחזר לפני החזרת השילטה ל-caller. בתכלס: לשמר במחסנית את \$ra לפני SMBCTIMES ja, ולשוחזר אותו בסוף.

דוגמא – caller

```
addi $sp, $sp, -4
sw $t1, 0($sp)
addi $a0, $s0, 0
addi $a1, $s1, 0
addi $a2, $s2, 0
addi $a3, $s3, 0
```

- ה-caller משתמש ברגיסטר הזמןי \$t1 גם אחרי קריאה לפונקציה leaf. אך הוא צריך לשמר את ערכו ולשוחזר אחרי הקריאה לפונקציה (לא mobutah_shaho_ismer).
- אך אנו מזמנים מקום במחסנית ושמירים את הערך בהתאם.
- אנו מעבירים את הפרמטרים לפונקציה ברגיסטרים המתאימים (a).
- בצע jal לפונקציה leaf.
- נשוחר את \$t1.
- ערך החזרה של הפונקציה נמצא ב-\$t1 וכן נוכל לעבוד איתו.

jal Leaf_example

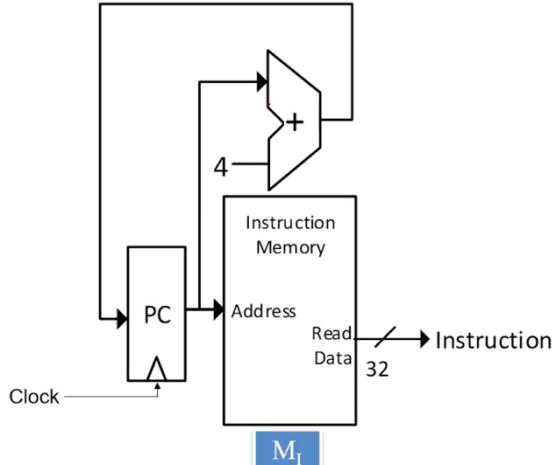
```
lw $t1, 0($sp)
addi $sp, $sp, 4
add $s6, $v0, $s0
add $s6, $s6, $t1
```



MIPS Design

Single Cycle

קריאה פקוודת:



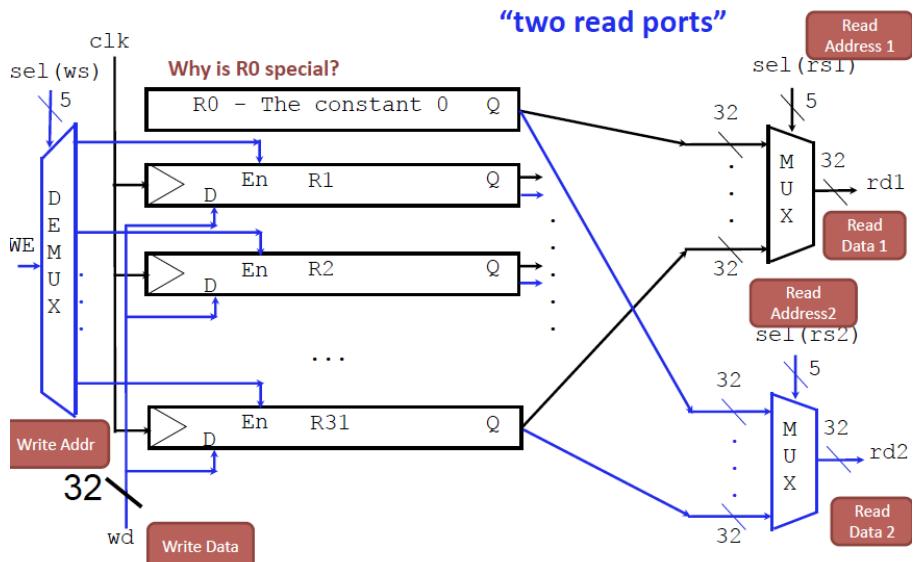
באשר אנחנו מבצעים fetch, אנחנו הולכים ל-PC (Program Counter) ששמור לנו את הכתובת בtower ה-PC (Program Counter) בתוך ה-Instruction Memory. ושמים את הכתובת בתוך ה-PC ב-4 ביטים. יש לנו מנגנון בזיכרון בכל fetch כזה אנו מקדמים את ה-PC ב-4 ביטים. שקדם ב-4 את הכתובת. לאחר מכן, אנו קוראים את הפקודה שהיא באורך 32 ביטים מהזיכרון והיא יוצאה דרך ה-Read Data.

נשים לב כי ה-PC מתעדכן רק בשינוי השעון הבא. לכן אנו מצפים לקבל ערך חדש בכל שינוי שעון. אנו צריכים לוודא שפעולות החיבור לא קורית לפחות מידי (min delay) או מהר מדי (max delay).

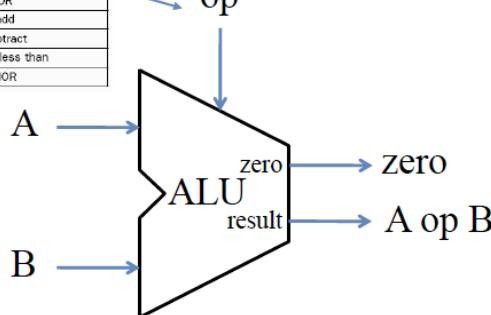
פענוח ופקודות בין רגיסטרים:

ישנו רכיב בשם **Register File** – סוג של זיכרון שביכולת כל הרגיסטרים (R0 – R31). אנו רוצים לבצע שתי פעולות מול רגיסטר כלשהו: קוראה ובתיבה.

- קוראה** – נחבר את כל הבניוסות של 32 הרגיסטרים ל-**xsout** (כל כניסה כזו מזורימה 32 ביטים של מידע), ובאמצעות 5 קווים בחירה (כדי לבחור מספר עשרוני בין 0 ל-31) נוכל לבחור את הרגיסטר שמננו נרצה לקרוא. בلومר זהו $\rightarrow 1 \rightarrow xsout$.
- בפעולות **type=rs** יש לנו **2 רגיסטרים לקריאה**, ולכן **בשתי xsout**, אחד עבור **rs1** והשני עבור **rs2**.
- בתיבה** – כאן ניעזר בכיוון ההפקה: **demux** 1 → 32 → **WE** = **WriteEnabled**. יש לנו 5 קווים בחירה (**ws** – בחירת הרגיסטר שאלוי אנחנו כתבים), ו-32 יציאות. ה-**(Write Data)** (**wd**) שמגייע ממוקם אחר הולך כל הרגיסטרים תמיד, ורק כאשר יהיה **Enabled** ברגיסטר המסויים, אז ה-**wd** יכנס אליו.



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

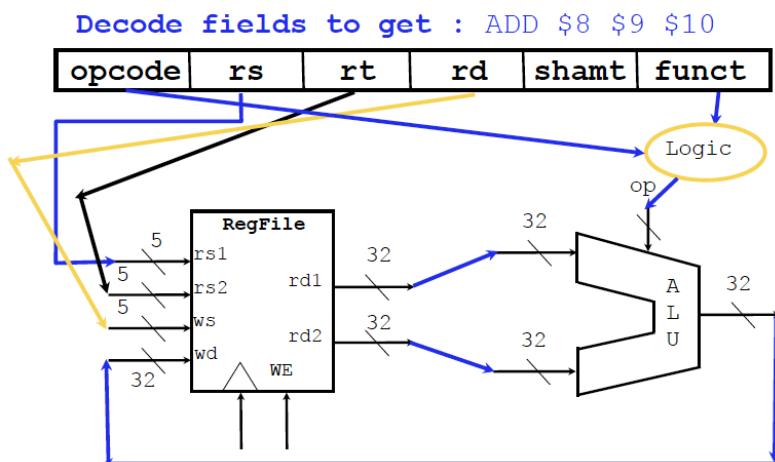


– רכיב שמבצע את כל פעולות החישוב: **ALU** (A, B, op) = $A \text{ op } B$. הוא מקבל שני מספרים לבצע עליהם את החישוב, ואת סוג הפעולה לבצע (**op**).

ה-**op** לא מגע ישירות מה-**opcode** של הפקודה, הוא שילוב עם השדה **funct** שמצוין איזו פעולה יש לבצע.



לאחר שאחננו מבצעים מסוימים לפקודה, אנחנו מפינים את rd ל-Port Register File Read Ports rs, rt ל-Port Register File Write Ports wd. התוצאה תיבנש ל-wd של Register File. כאן הכל מתחבר **בעור פקודה** נTHONה **מסוג R**:

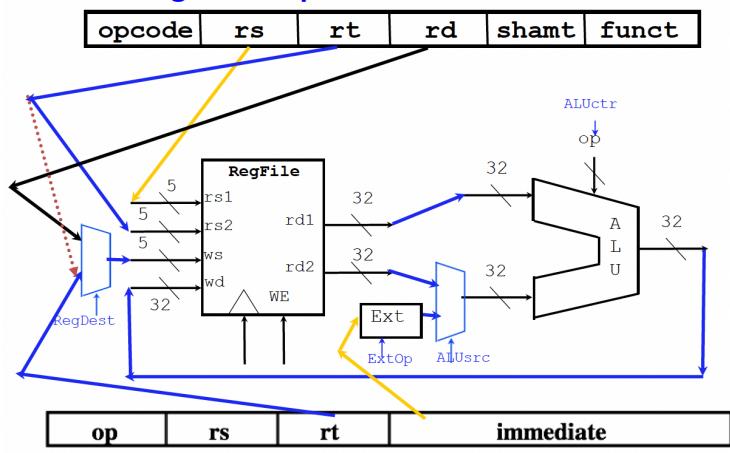


פקודות על IMMEDIATE

הפעם ה-*rt* יכול להיות גם רגיסטר שלו או כתובים (בפקודת load). כדי לשלב עם מה שעשינו קודם, יש לנו **aux בצד ימין** שבוחר האם אנחנו עובדים עם כתובים immediate (יש תהליך של sign extend מ-16 ביט ל-32 ביט בעת הצורך) או read port נוסף.

בנוסף, **aux בצד שמאל** שבוחר האם *rt* הוא לקריאה (store) או לכתיבה (load).

The merged data path ...



Decoder has two control methods – selected according to opcode

פקודות קרייה/כתביה:

op	rs	rt	immediate
----	----	----	-----------

Syntax: LW \$1, 32(\$2)

Action: \$1 = M[\$2 + 32]

Syntax: SW \$3, 12(\$4)

Action: M[\$4 + 12] = \$3

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate
31-26	25-21	20-16	15-0

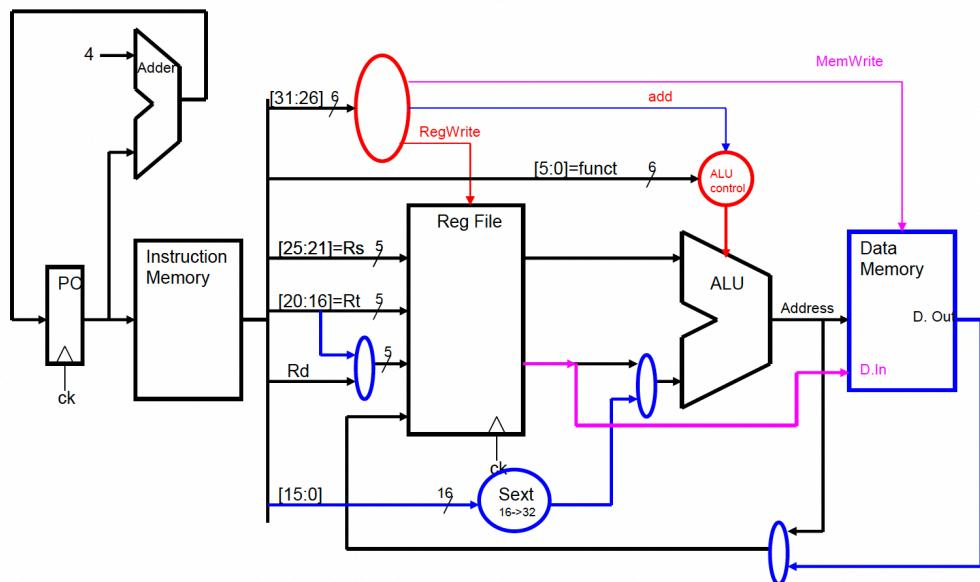
- אנחנו מוסיפים את ה-offset ל-*rs* תמיד. ה-ALU מחשב את הכתובת על ידי חיבור *rs* + offset.
- (**reg write**) – ה-*rt* הוא הרגיסטר שנחננו כתובים אליו (קוראים מ-*rs* שהוא כתובות בזיכרון).
- (**mem write**) – ה-*rt* הוא הרגיסטר שנחננו כתובות בזיכרון.
- **קוראים** ממנו (כתובים ל-*rs* שהוא כתובות בזיכרון).

Load: IR[op] = 100010 = ‘load’

Reg[IR[rt]] \leftarrow M_D[Reg[IR[rs]] + SiEx(immediate)]

Store: IR[op] = 101011 = ‘store’

M_D[Reg[IR[rs]] + SiEx(immediate)] \leftarrow Reg[IR[rt]]

פקודות בקרה:

יש לנו שני סוגי של פקודות:

- – לוקח את הכתובת הנוכחיית ומחשב offset ממנו. **branch**
- – הנה הכתובת שאליה צריך לקפוץ. **jump**

בפקודה branch נעשה rs=rt ובמידה ונקבל 0 (המ שווים) נעדכן את ה-PC לערך immediate (הכתובת הבאה לביצוע).

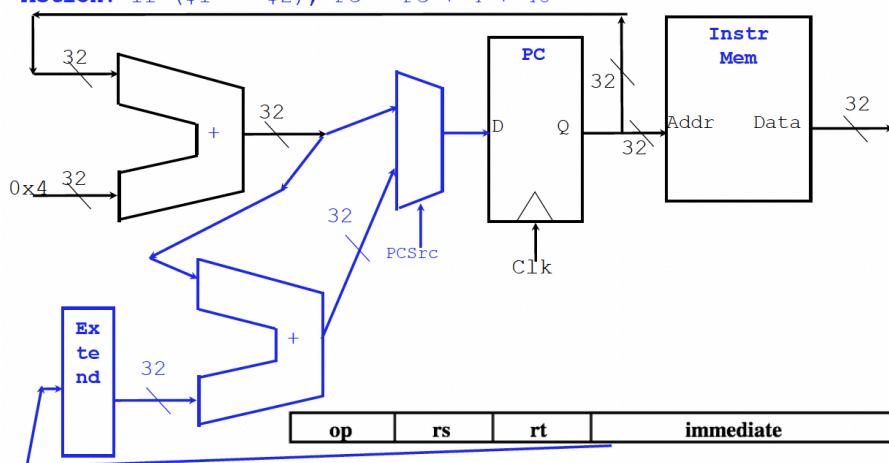
```

if Reg[ IR[rs] ] ≠ Reg [ IR[rt] ]
    PCnext = (PCcurrent + 4) + immediate<<2
Else
    PCnext = (PCcurrent + 4)

```

יש לנו אך שבור האם אנחנו לוקחים את הכתובת+4 (התקדמות רגילה לפקודה הבאה) או את הכתובת+4 בתוספת ה-4. נשים לב כי יש לנו כאן שני רכיבי ALU לטובות שני החישובים האלה, ו-asm שմכريع ביניהם.

Syntax: BEQ \$1, \$2, 12
Action: If (\$1 != \$2), PC = PC + 4
Action: If (\$1 == \$2), PC = PC + 4 + 48

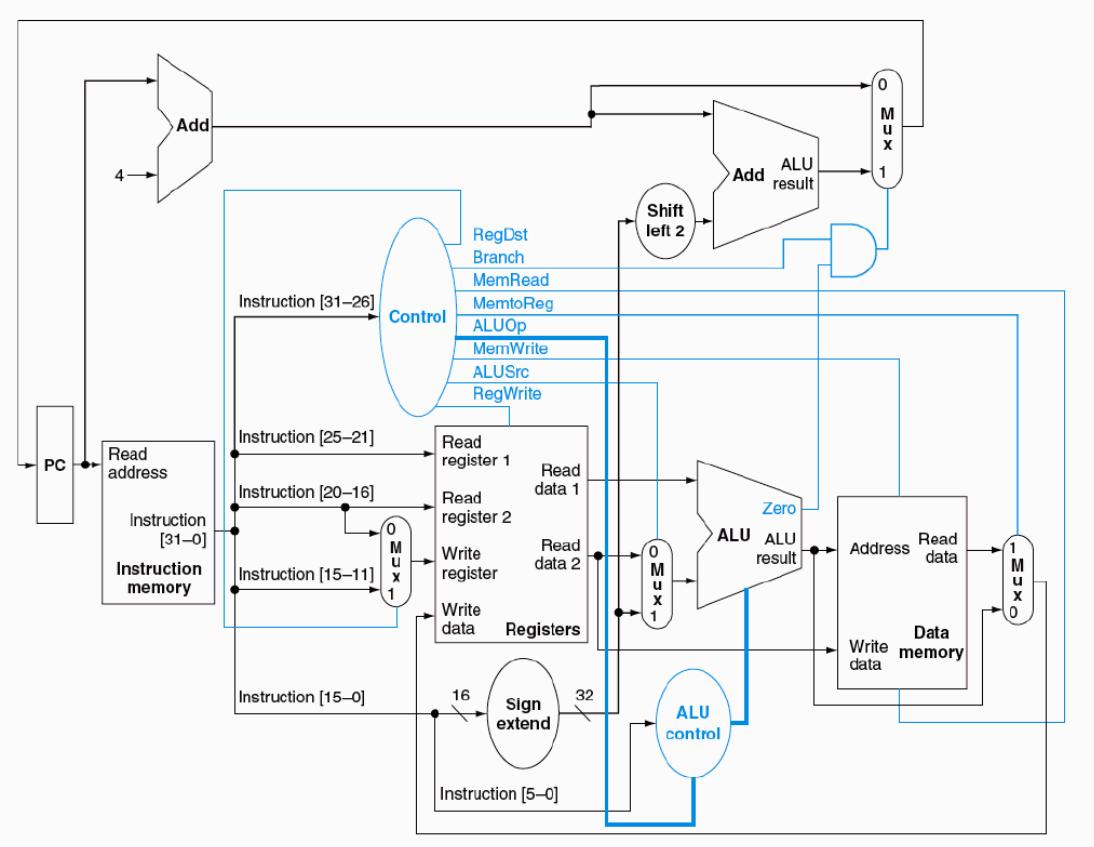


בפקודת jump נלקח את 26 הביטים של ה-PC, offset, נבצע offset שמאליה פעמיים (להכפיל ב-4) וכן נקבל 28 ביטים. נוסיף את 4 הביטים השמאליים של ה-PC הנוכחי וכן נקבל את הכתובת החדשה.



ביצועים של Single Cycle

חזרה על לבן חסר כאן dump בشرطו:



1. **שלב fetch:** ככל יושב סביב ה-PC שאומר לנו איפה נמצאת הכתובת שאנו רוצים למשוך ממנה את ה-instruction הבא (הכתובת הבאה לביצוע, מתוך ה-instruction memory).
2. **שלב decode-controller:** מפצלים את הביטים של הפקודה לשדות השונים שלה, הפיזול זהה לאו דווקא דור, יש גם חפיפה.
ה-controller: מייצר את אותות הקריאה ושולט על כל שאר הרכיבים שיש בארכיטקטורה – איזה כניסה ב-axmu לבוחר, לאפשר כתיבה לרגיסטר או לא.
3. **שלב execute:** הביצוע עצמו של החישוב (העיבוד של המעבד) באמצעות ה-ALU. התוצאה יכולה לשמש בתור כתובת בזיכרון למשול (פקודות מסוג `ws`, `wa`).

ביצועים:

נרצה למדוד כמה זמן לוקח לבצע פעולה כלשהי. כדי לנתח את ה-CPU Time, יש לנו 3 מרכיבים עיקריים שאפשר לשפר בנפרד:

1. **IC (Instruction Count)** – כמות הפקודות שצריך לבצע בתוכנה. כדי לצמצם את המספר הזה, צריך לשנות את ה-ISA (لتמוך ביוטר פקודות).
2. **CPI (Cycles Per Instruction)** – כמה מהזורי שעון לוקח לבצע כל פקודה. לעיתים מודדים $\frac{1}{CPI}$.
- a. לא תמיד מואוד קל לדעת בדיק מהו אותו CPI: יש הרבה אינטראקציה בין הפקודות, הסדר ביניהן חשוב, מצב הזיכרון באותו רגע. הרבה דברים משפיעים על ה-CPI.
b. חישוב כללי:

$$CPI = \frac{\#cyc}{IC} = \frac{\sum_i CPI_i * IC_i}{IC} = \sum_i CPI_i * \frac{IC_i}{IC} = \sum_i CPI_i * F_i$$

$\cdot \frac{1}{cyc}$

.3 – זמן של מהזורי שעון, נמדד ב- μ sec. תדר השעון נמדד ב- $\frac{1}{cyc}$.



דוגמה מספקית: נתון לנו כמה זמן לוקח כל שלב. למשל פקודת R-Type לא ניגשת ליזכרון. פקודת Branch גם לא ניגשת ליזכרון וגם לא מבצעת write. יש פה redundancy, ה-**single cycle** מואוד לא יעיל: בזמן מחזור אנחנו צריכים להתחשב ב-worst case. זה מאד בזבזני. אולי אפשר לתכנן אחרת, כדי לנצל את העובדה שרבות הפוקודות לא צריכהות 600ps.

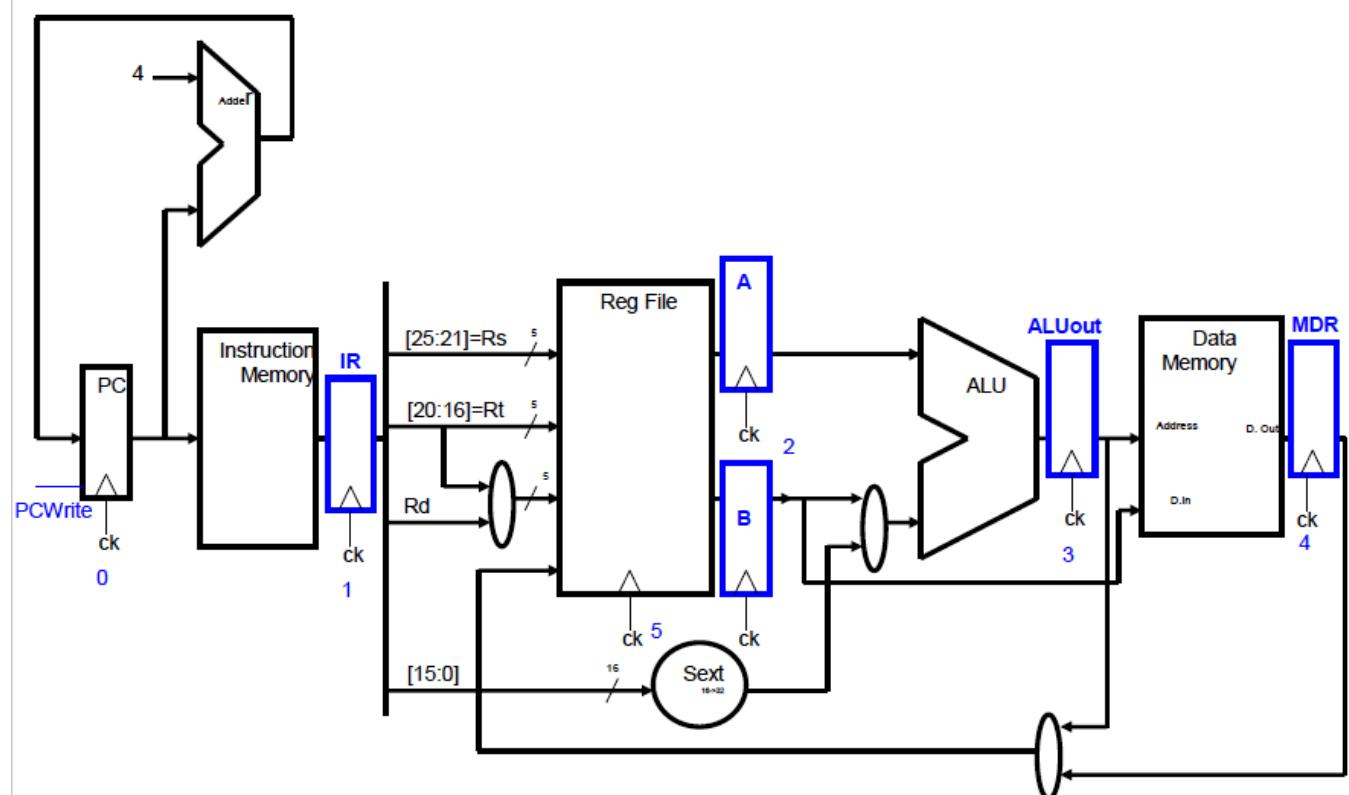
Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

Each instruction type takes a different time to execute

מה אפשר לעשות? **נשבר פעולה אחת בכמה מחזורי שעון שהם קטנים יותר:** נניח כל מחזור כזה יהיה 200ns במקום 600ns (זמן המסלול הארוך ביותר). נבצע כל שלב במחזור שעון נפרד: פקודת קומפונס תבצע במחזור שעון אחד (200). מנגד, פעולה load word תיקח 5 מחזורי שעון ותבצע בזמן גורע יותר (1000). ה-**CPI** עלה.

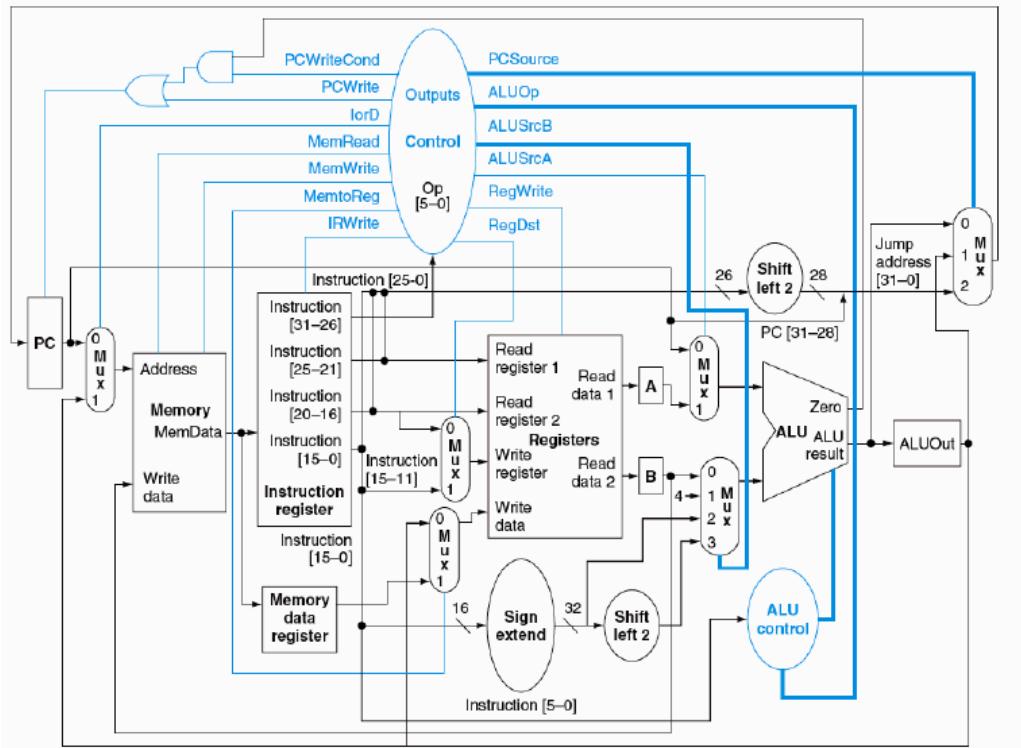
למשל, עבור פקודת **lw**, כדי לשבר את ה-cycle היחיד: **נוסיף רגיסטרים (בכחול)** והם יעזרו לנו באופן הבא:

- במחזור השעון הראשון, ה-PC מתעדכן בכתובת החדש, הכתובת הזאת נכנסת ליזכרון, הוא שולף את הפוקודה הבאה והיא נרשמת ב-**IR** – זה מחזור שעון אחד.
- אנחנו צריכים לוודא שהקידום ב-4 לכתובת ב-PC קורה רק פעם אחת עבור הפוקודה, לא בכל מחזור שעון. לשם כך יש לנו את ה-**PCWrite** שנאפשרר אותו רק במחזור הראשון. עד שלא נעשה fetch עוד פעם לא נאפשר עדכו.
- במחזור השני, הפוקודה מה-**IR** גורמת לעדכו של הרגיסטרים ומשם זה ממחכה ב-**A-B**.
- במחזור השלישי, מבצעים את ה-motion של הפוקודה – ה-data execution של הפוקודה, הוא כבר ממחכה ברגיסטרים הכהולים הקודמים. התוצאה ממחכה ב-**ALUout**.





אפשר ליעיל בכך במספר דברים: לא צריך שני ALU-ים (יש אחד שפועל רק בשלב fetch), הזיכרון (data memory) יהיה צריך להיות מסוגל גם לקרוא וגם לכתוב באותו מוחזור שנון והפעם ה-IR עובד רק בשלב fetch וה-MDR עובד רק בשלב האחרון.

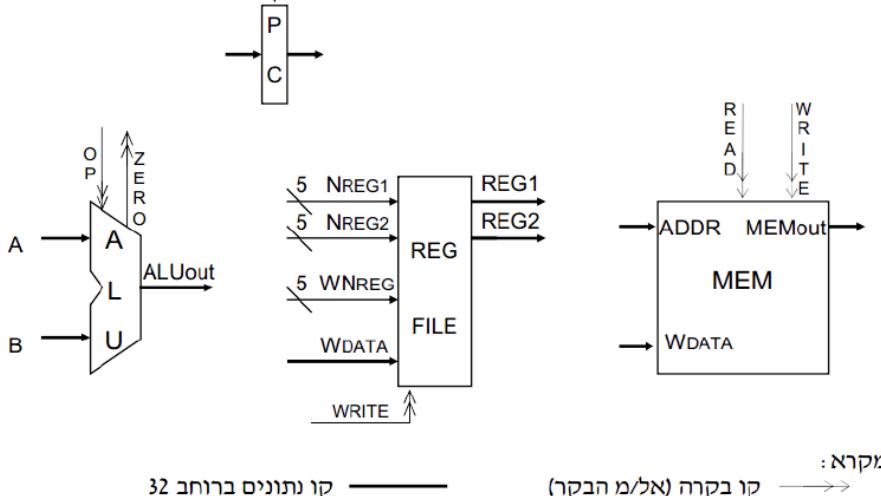


SC (תרגול 7)

קונפיגורציות MIPS:

- SC – פקודה מבוצעת במוחזור שנון יחיד בעל זמן מוחזור ארוך מאוד. חסרון: חוסר ייעילות. יש פקודות שלא צריבות להיות ארכות כמו פקודות אחרות. זמן המוחזור הוא אחד ונקבע על פי הפוקודה הארוכה ביותר.
- MC – פקודה מוחולקת למספר שלבים, כל שלב מתבצע במוחזור שנון אחד (קצר).
- – **שלבים שונים של פקודות שונים מתבצעים במקביל.**

אגרים ויחידות עיקריות:



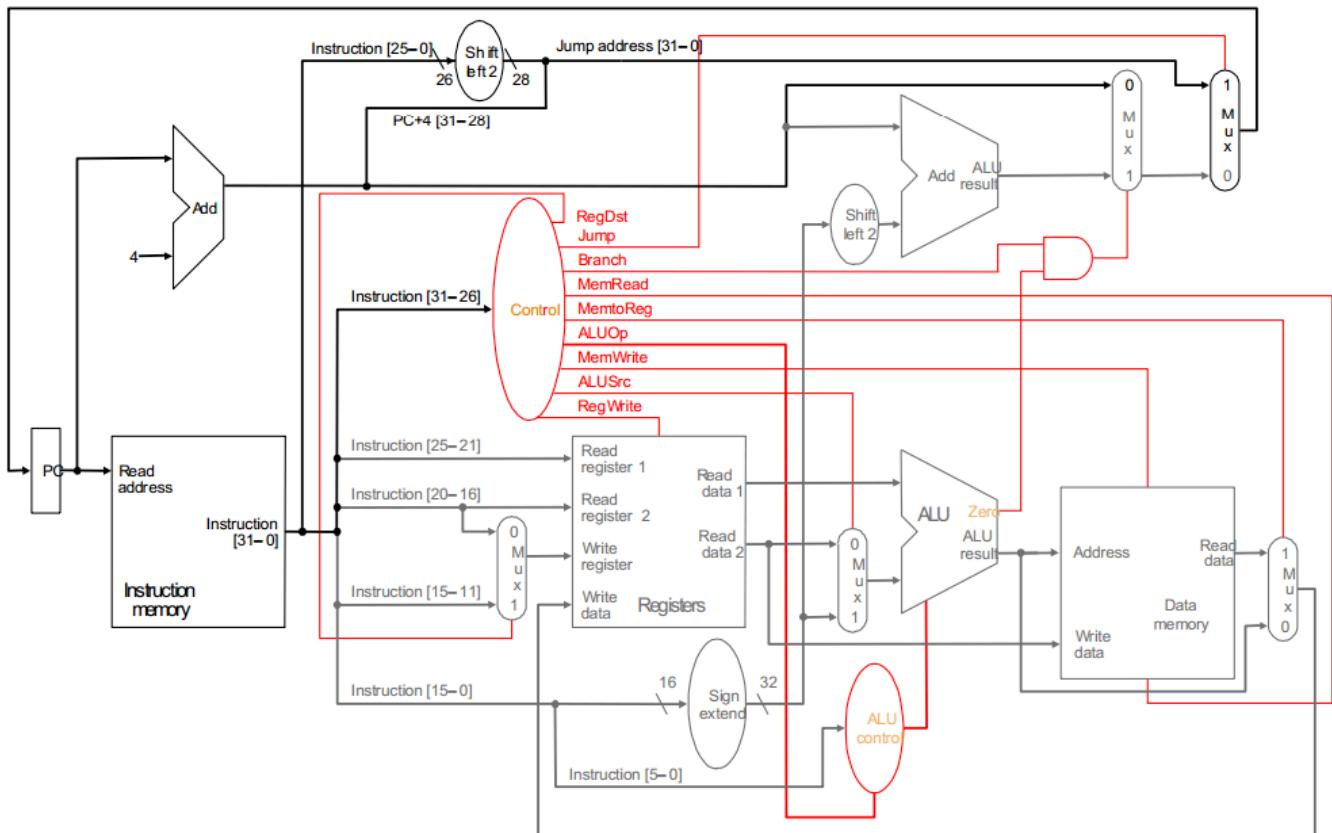
• PC – המיקום שלנו בתוכנה, הפוקודה **הבאה לביצוע.**

• – **Instruction Memory** – זיכרון **הפוקודות.**

• **Registers File** – שני רגיסטרים **לקריאה, רגיסטר אחד לכתיבה. יש כניסה של data-הFILE לכתיבה. שתי יציאות של מדע. אותן בקרה של לאפשר בקרה או לא.**

• **ALU** – מקבל שני מספרים ופעולה **לביצוע ומחשב את התוצאה. בנוסף אומרת האם היא אפס או לא.**

• **Memory**

השלבים ב-SC (שרוטט מלא):

מספר	שלב	תיאור
1	Fetch פקודה	<p>נקבל ביציאה של ה-PC לפקודה הבאה (4 בתים, בכל מחזור שעון). הכתיבת PC תבצע בתחילת מחזור השעון הבא.</p>
2	:Decode מפענחים את הפקודה, הופכים אותה בקרה לאותות בקרה	<p>ראיינו 3 פורמטים לפקודות: J,R. אחרי שקראננו את הפקודה, אנחנו רוצים לפצל אותה לכמה חלקים. ה-e-opcode קיים בכל סוג פקודה, וה-controller יודע לפחות ליזיר את אותן הפעולות בהתאם לפקודה.</p> <ul style="list-style-type: none"> הבדל בין R ל-J הוא קיום השדה rd. בפקודות מסוג I אין rd, ואם נרצה לבתוב מדובר ברגיסטר rt. לכן יש לנו את השדה rd. בפקודות מסוג I אין rd, ואם נרצה לבתוב מדובר ברגיסטר rt לבכיבתיה, ואם הוא מקבל 1 הוא יכנס את rd לבכיבתיה. נשים לב כי funct-ל-ALU, אבל כל funct-immediate עובר sign extend ומשיר להלבה. <p>עבור פקודות הקומפונס: נפצל את הביטים שמקודדים את המיקום אליו נרצה לкопץ (ביטים 0 עד 25). הביטים האלה יעברו shift left 2 כי אנחנו מקודדים את הכתובת-b-words, ולכן נרצה לכפול אותם ב-4. משם, נשרות לביטים הדרושים מה-PC+4 PC+4 וכהה תיקבע הכתובת-h-kmpn. מאוחר יותר יש את שמחילט אם אנחנו מבצעים את הקפיצה או לא. ברגע אנחנו רק מחשבים את בתובת הקפיצה.</p>
3	:Execute ביצוע שלבים מתמטיים – חישובים.	<p>קיבלו את כל השדות הקודמים. כעת נצטרך שה-ALU ידע לבצע את החישובים הנדרשים. לכן יש לנו עוד mux (ALUSrc): מחליט האם הארגומנט השני ל-ALU הוא רגיסטר (0) בפקודה או immediate (1) בפקודה (R).</p> <ul style="list-style-type: none"> ה-ALU צריך לדעת מה לבצע. בשביל זה יש לנו אותן בקרה funct-R. אם מדובר למשל בפקודה R נתחייב בשדה funct. במקביל, יש לנו התיחסות לפעולות branch: לא נרצה להשתמש funct-immediate לשום חישוב, רק לבכנתה החזרה. נפצל את המסלול לעוד shifter (עליה למעלה) ונחבר אותו לכתובת הנוכחית PC+4. לא מתבצע החלטה הסופית האם עושים branch או לא. תבוצע בשלב מאוחר יותר.
4	:Memory כתובים או קוראים.	<p>אפשר לנו לבתוב לזכרון ולקרוא ממנו – לא בו זמינות באותו מחזור שעון (בניגוד ל-Register File). יהיו לנו 2 אOTES בקרה האם לבתוב (MemWrite) או לקרוא (MemRead).</p>

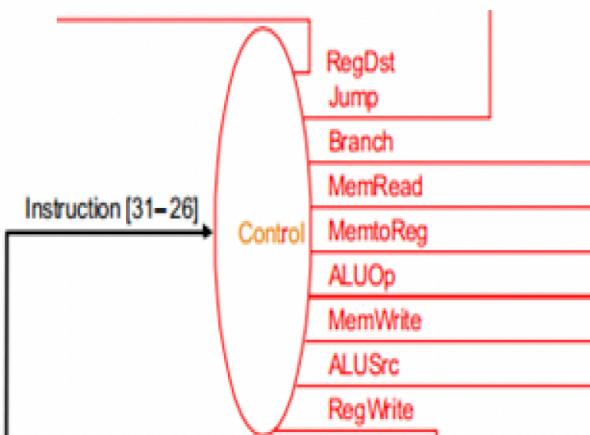
	<ul style="list-style-type: none"> ▪ ניתן את התוצאה של ה-ALU ל-Address במידה שבאמת חושבה בתובת. ואז יוכל לכתב אונ' לקרוא באמצעות הכתובת שקיבלו. ▪ כמובן, אנחנו מוצאים את ההחלטה, האם עושים jump/branch. יש לנו And שלוקח את אות הבקרה零 ואות Branch零. אם שניהם 1, רק אז קיבל את תוצאה החישוב של כתובת הקפיצה שלנו. זה קשור ל-beq. 		
5	<ul style="list-style-type: none"> ▪ יש לנו את memory ואת ALU. צירכום לבחור את התוצאה של אחד מהם, ולבחור האם כותבים אותה לריגיסטר או לא. ▪ יש לנו Auswahl לבחירה זו – 0 זה ALU ו-1 זה memory. ▪ נבחר בזיכרון עבור word load. ▪ יש אות בקרה עבור הirection Register File. לא נרצה לכתוב אותם לריגיסטר. למשל בפקודת branch. לא מעוניין אותנו מה יש בזיכרון, או מה תוצאה החישוב. לכן נקשייח את ה-Register File ולא נבצע אליהם כתיבה. 	:Write Back בתיבת התוצאה חזרה אל ה-Register File	

רכיבי הבקרה של SC:

- מקבל את opcode של הפקודה ומיציר אותן בקרה שמחולקים אותן לטיפול הרלוונטי.
- זה שאומרים ל-ALU AILO פקודות לבצע. מקבל את שדה funct (אם הוא רלוונטי) ואת שני הביטים של ALUOp ומיציר ארבעה ביטים של הפקודה לביצוע.

אותות הבקרה היוצאים מה-controller:controller

– קבוע האם התוצאה תיכתב לריגיסטר rd (אם מדובר בפקודת I-type I) או



לרגיסטר rd (אם מדובר בפקודת R-type).

לפי ה-X, שאומר באילו ביטים נשתמש בשביל הכתובת

– האם הפקודה היא פקודת jump.

– האם הפקודה היא פקודת Branch.

.register file – האם יש קריאה מהזיכרון ל.

MemToReg – האם כתיבה לקובץ

הרגיסטרים היא מהזיכרון או ישירות מה-ALU.

(יכול להיות שלא נכתב ולא נקרא)

ALUop1,ALUop0 – 00 אם מדובר בפקודת גישה לזכרון. 01 אם מדובר בפקודת beq. 10 אם מדובר בפקודה מסווג R-type .

– קבוע האם יש כתיבה לזכרון.

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
opcode	rs	rt	rd	shamt	funct
31-26	25-21	20-16	15-11	10-6	5-0

– קבוע האם הערך השני שיוכנס לרכיב ה-ALU הוא רגיסטר AluSrc (פקודה מסווג R-type) או immediate (פקודה מסווג I-type)

– קבוע האם כתיבה לריגיסטר היעד

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate
31-26	25-21	20-16	15-0

6 bits	26 bits
opcode	offset – target address
31-26	25-0

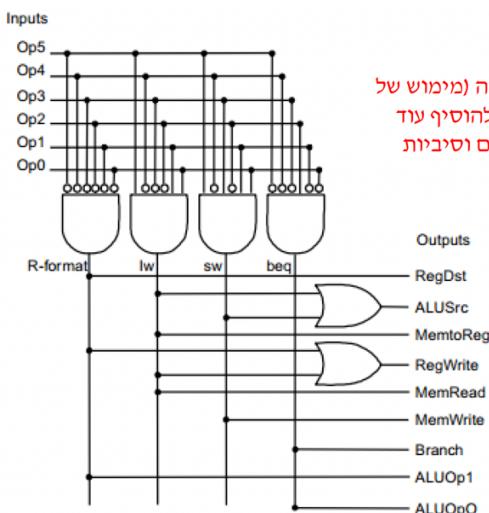
28

IMPLEMENTATION OF THE CONTROL:

מימוש ייחידת ה-control: כל אחת מהסיביות של opcode ישלחות לשער AND עם 6 כניסה, חלק מהכניסות עם NOT וחלק לא, בהתאם למה שה-control מיצג. למשל עבור פקודת R נרשם 00000000000000000000000000000000 ואז לפי ה-NOT על כל הביטים קיבל 1111111111111111 ואז בהתאם ויגע אותן לאותות הבקרה המתאימות. היתרון – קל למשת טיפול באותות בקרה חדשים. מושגים ב-selketos ומחברים לקווים הרלוונטיים. בcn"ל עבור פקודת חדשה, מושגים שער חדש וקובעים כניסה מתאימות.



משמעות יחידת ה-ALU: ALU Control output – מתרגם את ה-ALUOp ואת סוג הפעולה ל-ALU המתאים.



Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control output
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

תמייה בפקודות נוספות:

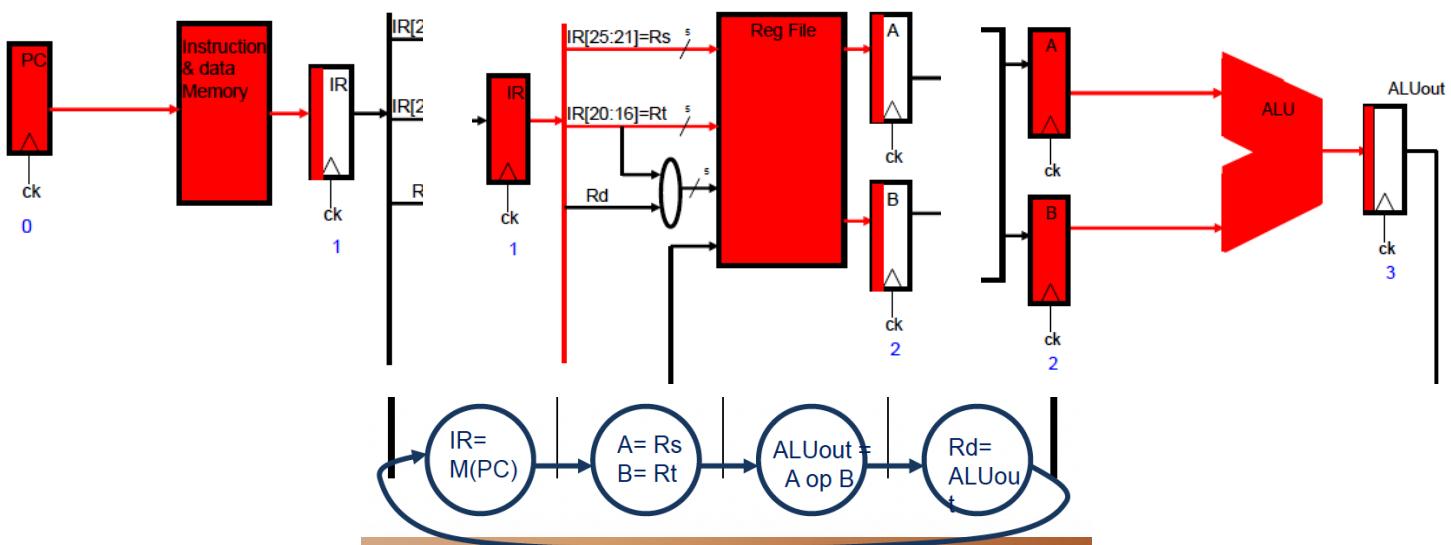
- פקודת lw: תבצע: $.lwreg \$s0, \$s1, \$s2: \$s0 = \text{Memory}[\$s1 + \$s2]$
- אותות הבקרה – RegDst=1 כי רגיסטר היעד מקודד בסיביות 11-15. MemRead=1 כי זה lw.
- MemToReg=1 כי נרצה לטעון מהזיכרון לרגיסטר. ALUOp=1. ALUOp1=00 כמו lw ו-ALUOp=10 כמו בפקודת R עם 3 רגיסטרים כי יהיה 0.
- אין צורך לעשות شيئاً נוספים מחוץ לבקר!
- פקודת jal: היא מדכנת את ה-PC, ובמציעים קומפוננטת כתובת מסויימת.
- שינוי ראשון: ניקח את תוצאת החיבור PC+4 PC+4 ונרצת להביא אותה ל-RA. נוסיף את הכתובת בתור koutן נוסף ל-PC+4/4. mux (MemToReg) שקבע לנו מה ערך הכתובת לרגיסטרים: ALU/זיכרון הופך ל-ALU/זיכרון PC+4/4.
- שינוי שני: בשלב write back נרצה לכתוב לרגיסטר RA. לכן נרצה להכניס עוד קלט ל-mux (RegDst), שהוא רגיסטר 31 (RA).

Multi Cycle

R-Type

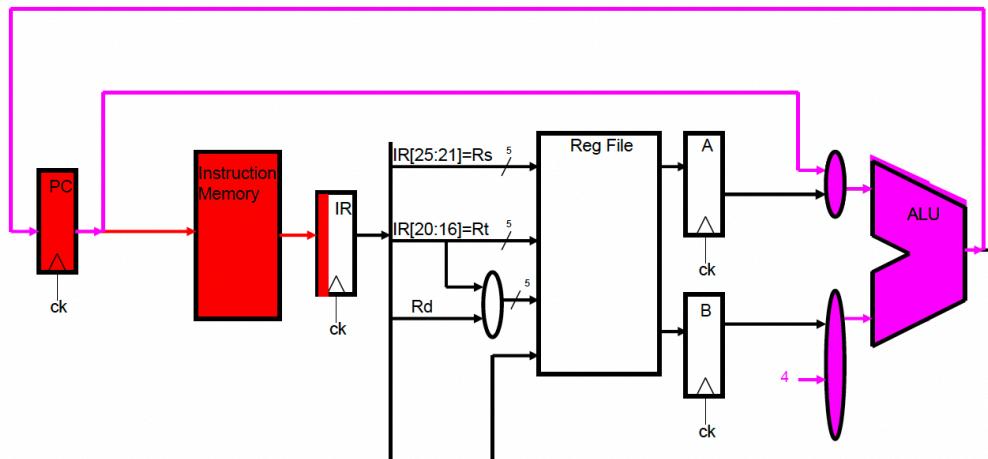
ב-single cycle, PC מתקדם ב-4 והפקודה מफצילה לשולש שדות: rd, rs (עבורים ל-read ports), rt (עבורים ל-write port) של ה-register file (ALU). נקלט את A ו-B שעוברים ל-ALU, התוצאה שלה נכתבת בחזרה לregister file.

ב-multi cycle נחלק את הפקודה לשלבים שונים ב-4 מחזורי שעון שונים: fetch, decode, execute, write back. נשים לב שה-PC מתעדכן רק במחזור 0 של כל פקודה (בשלב fetch בלבד). ולכן השאר הדברים בשרשראת לא ישתנו, הערכיהם ישמרו במסלול ימינו שם.





במקביל, במחזור השעון הראשון קורה המסלול הסגול向前 לקידום הכתובת של ה-PC ב-4, ובמחזור השלישי ה-ALU כבר פניו ליביצוע ה-**execute** עצמו. אנו משתמשים באותו ALU פעמיים, בשלבים שונים של הפקודה.

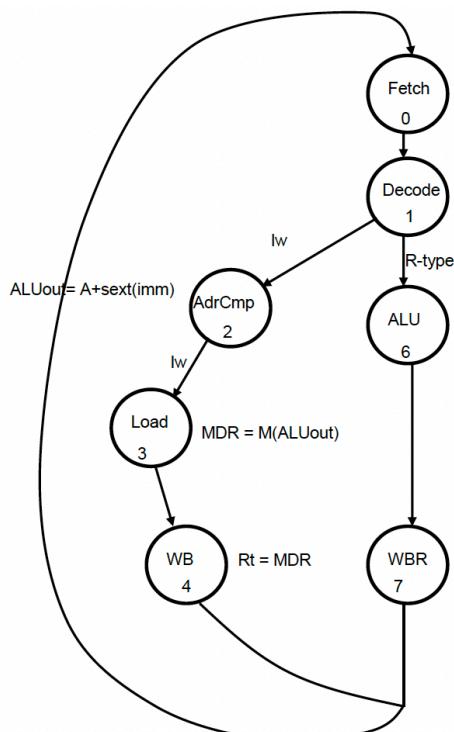


הוספת פעולה Iw:

לפעולות Iw יש 5 מחזוריים:

- fetch מתרחש באותה צורה: המסלול האדום והסגול קורים במקביל: שילוף הפקודה + עדכון ה-PC ב-4.
- decode מתרחש באותה צורה, שומרים את rs ו- rt ב-A ו-B.
- execute אנחנו לוקחים את ה-immediate (בוחרים ב- rt את הכניסה הנכונה) ואת z ומבצעים חיבור.
- memory (שלב נוסף) ה-ALUOut הולך לזכרון, ולקוח את הכתובת המתאימה. התוצאה מוחבה ב- MDR .
- write back לוקחים את הכתובת מה- MDR .

נשים לב שה-IR וה- MDR לא נמצאים בשימוש באותו הזמן. אפשר לאחד אותם לכדי זיכרון אחד. אנחנו מתחילה לקבל מבונת מצבים מסוג Moore: הפלטים לא תלויים בקלט.



הוספת פעולה sw:

מואוד דומה ל-Iw רק ש:

4 מחזוריים.

- אין בו את הצורך ב-write back.
- ה-data שנחכנו רוצים לבצע יוצא rt (מחבה לנו ב-B), ונכתב לזכרון.

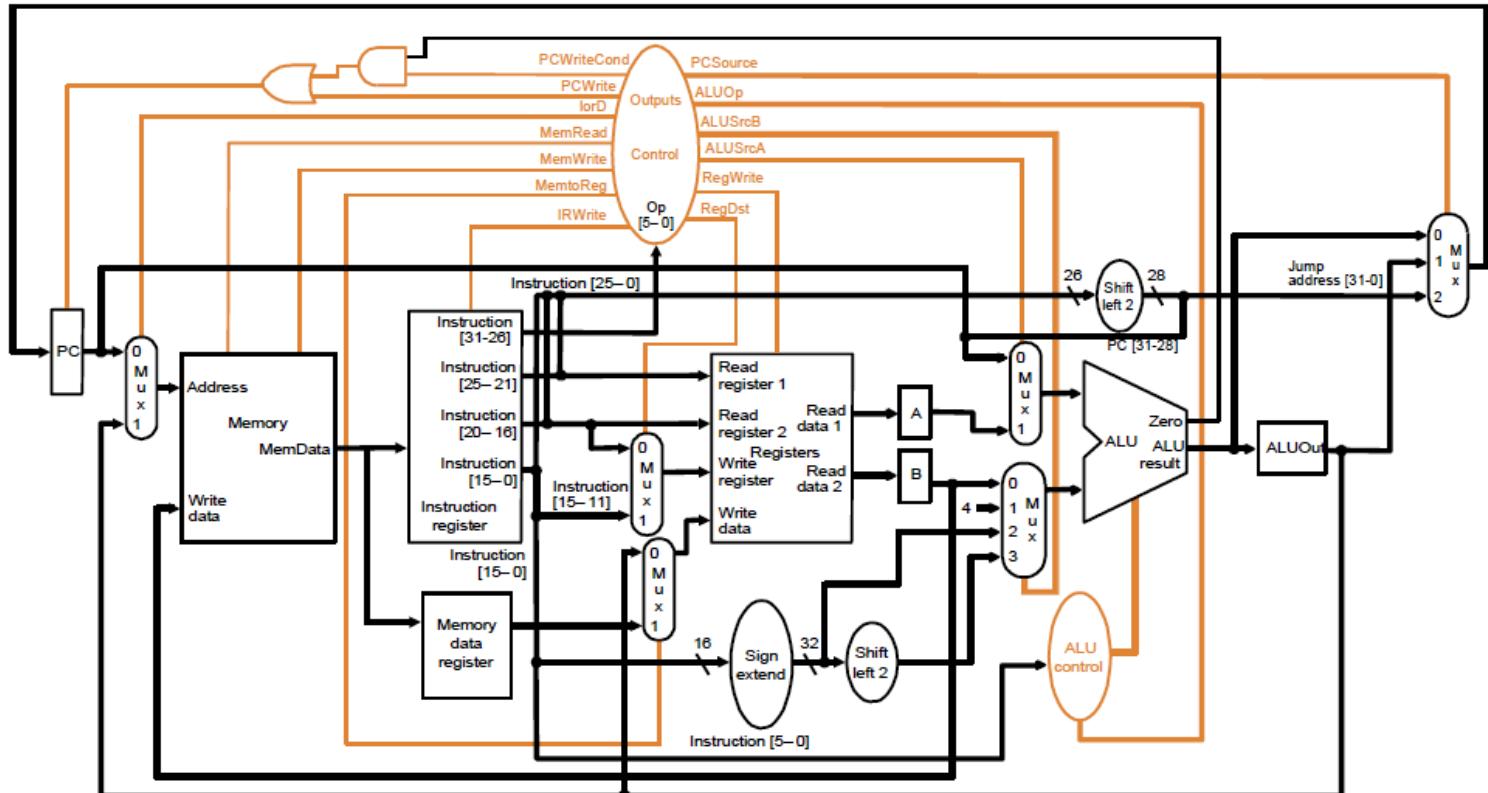
הוספת פעולה branch:

- מה שנוסיף הוא אקסט בתחילת ה-PC: האם מבצעים פשוט תוספת של 4, או שמקבלים בתובת קפיצה אחרת. אנחנו לוקחים את הכתובת המעודכנת של הפקודה הבאה (כולל 4), וצריכים להוסיף להחליף בה מהו בעורת ה-ALU.
- שלב ה-fetch ברגיל: פקודה חדשה נשמרת ב-IR, **ומבצעים 4 PC+4 במקביל**.
- שלב ה-decode: מקבלים את המספרים A ו-B שצריך להשוות. שימוש ראשון ב-ALU: **מחשבים את בתובת הקפיצה על בסיס immediate-h-type** ו-**הכתובת המעודכנת של הפקודה הבאה (PC+4)**. זה מחכה ב-ALUOut ALU (למקרה הצורך).
- שלב ה-execute: שימוש שני ב-ALU לחישוב עצמו של ההשוואה בין A ל-B (אם יצא 0 או לא). אם צריך מעדכנים את ה-PC לכתובת שאליה אנחנו קופצים (branch if equal), אחרת לא מעדכנים.

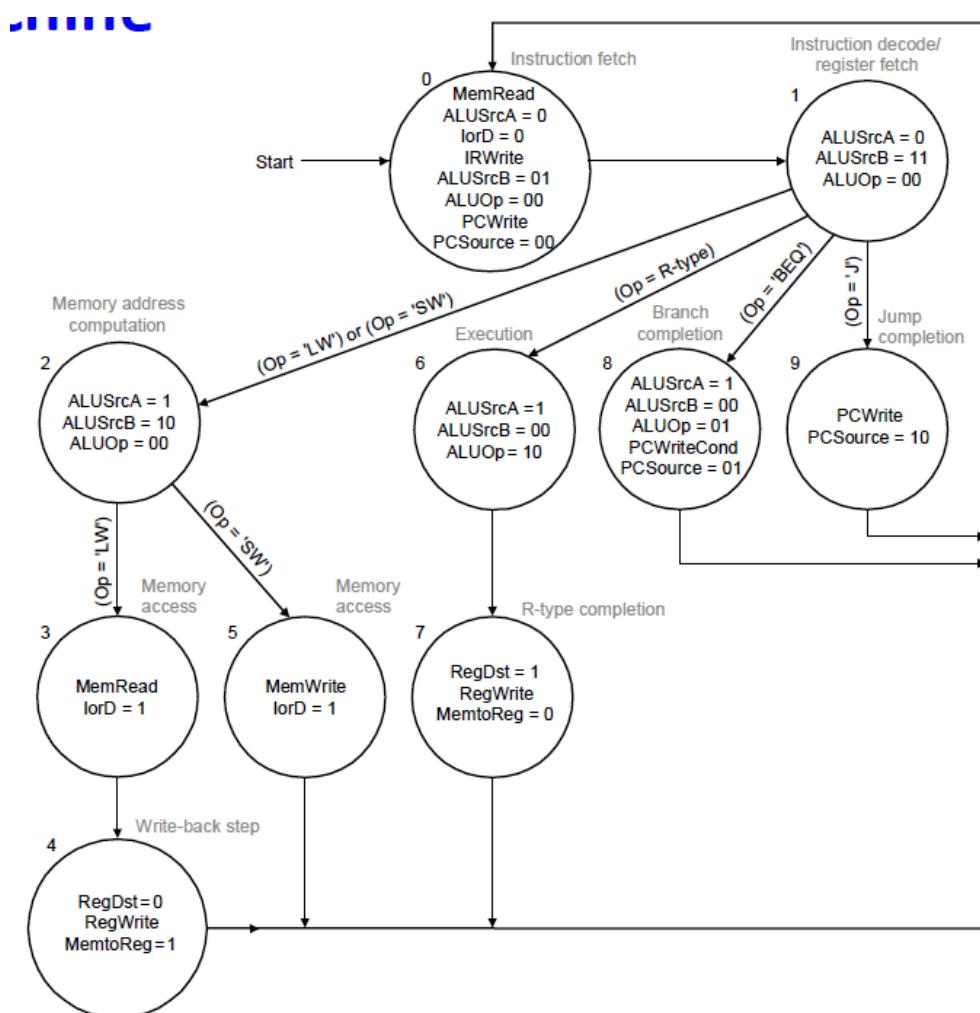
הוספת פעולה branch: כמעט שקול ל-fetch. עושים decode-if-branch. עוד מחזור לעשوت את ה-execute: מחשבים את הכתובת החדשה ובעדרת ה-אקסט מעדכנים את ה-PC.



משמעות מלא עם ה-controller, שהוא מكونת מצבים, לכל שלב הוא מוציא אוסף אחר של אותות בתום:



מבנה המצביעים המלאה של ה-controller עם פירוט של האותות:

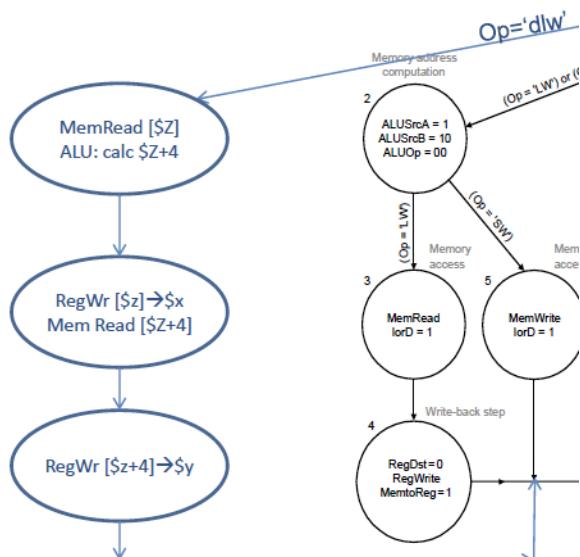




דוגמה מבחן (2017): עקב תקלה ב-controller של מעבד Multi Cycle MIPS מתקיים תמיד $0 = \text{Source1}$.

- נשים לב שיש לו שני ביטים ואנו יודעים שהбит השמאלי 0, בلومר PC יכול לקבל ערכים רק של 00 או 01.
- הברק יכול לבצע ב-00 (PC + 4), ב-01 (branch) אבל לא ב-10 (jump).
- המעבד יrisk נכון תוכניות ללא קפיצות מחלות (j).**

דוגמה מבחן (2018): רוצים להוסיף למעבד Multi Cycle MIPS את הפקודה `dlw $x,$y,$z`. (double load word). הפקודה קוראת לתוך $\$x$ את הערך שנמצא בכתובת $\$z$ בזיכרון, ולתוך $\$y$ את הערך שנמצא בכתובת $\$z+4$ בזיכרון. היא מחליפה שתי פקודות רגילות של `lw`.



- האם נכנו להשתמש בפורמט של פקודה `lw` בשינוי האופקود עבור הפקודה החדש? **לא.** בפקודה החדשה צריך להזכיר 3 גיסטרים. נצטרך להשתמש בפקודה מסוג R ולא בפורמט של `lw` שהוא מסוג עם 2 גיסטרים - `i.immediate`.
- הוסף את הפקודה בצוරה הייעלה ביותר בבדיקה CPI:

- או פקודה מסוג R.

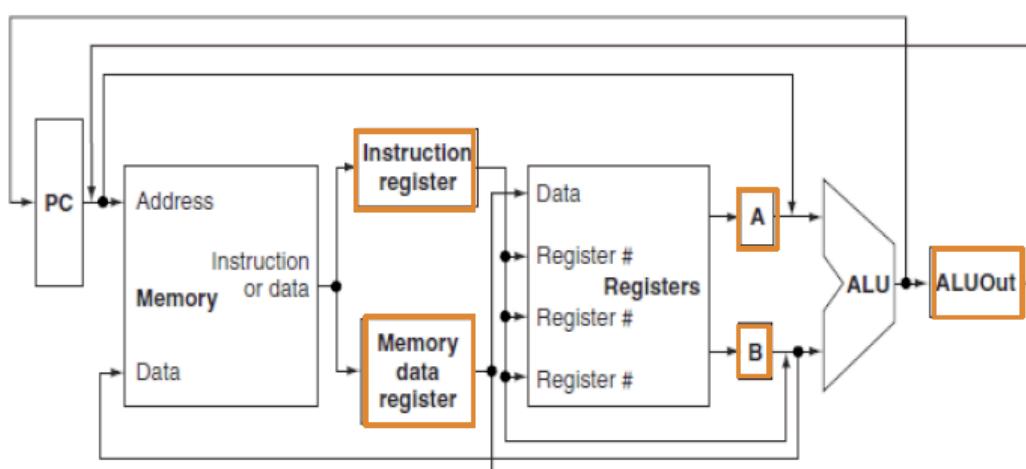
- בשלב execute: לבתייה הראשונה מבין השיטים לא צירע לעשות פעולה חישוב. לכן, בזמן שאנו קוראים מהזיכרון את הכתובת עבור הפקודה הראשונה שבה אין צורך ב-ALU (את $\$z$ נקרא מה-memory), **מחשב בבר את $4+z$ עבור הפקודה השנייה במחזור הבא (הכתובת $4+z$)**.
- מחזור לאחר מכן נכתוב את הפקודה הראשונה (reg write). ונקרא את השניה (את $4+z$ מהזיכרון).
- במחזור האחרון נכתוב את הפקודה השנייה.

(תרגול 8 MC)

השינויים שמוספו מ-SC ל-MC:

הבדל העיקרי הוא שב-SC כל פקודה משלטת על כל החומרה בין אם היא צריכה חלק מסוים או לא, וב-MC ניעול את השימוש בחומרה, אך שנחלק כל פקודה למספר שלבים, כל שלב יבוצע במחזור שיעון (קצר יחסית ל-SC). כל פקודה לוקחת מספר מחזורי שיעון. השינויים שנוספו:

- רגיטרי בינים –** יקפיו את המצב לאחר כל שלב: רגיטר לשימרת הפקודה (Instruction), שני רגיסטרים לשימירת הערכים מקובץ הרגיטרים (A, B), רגיטר לשימרת תוצאה ה-ALU (ALUOut), ורגיטר לשימרת הערך מהזיכרון (Memory). כך ניתן יהיה להשתמש בערך שהושב במחזור השיעון הקודם, **במחזור השיעון שבא אחריו**. הם שומרים "ערבי" בין שלבים שונים של ביצוע הפקודה, בין מחזורי שיעון שונים.
- אחדות יחידות –** אם עושים כל פעולה במחזור שיעון אחד, אז יש אפשרות לעשות ניצול נוסף של הרכיבים. אם במחזור אחד אנחנו קוראים ובאחר כותבים, אפשר פשוט לאחד בין הרכיבים **לרכיב אחד**, לא משתמשים בשנייהם באותו מחזור שיעון.
- זיכרון –** משתמשים באותה יחידה עבור זיכרון נתונים (data) וזכרון פקודות (instructions).
- ALU –** כל החישובים ב-ALU אחד (בניגוד ל-SC שם חישובי PC והקפיצה בפקודת beq לא עברו דרך ALU נפרד).





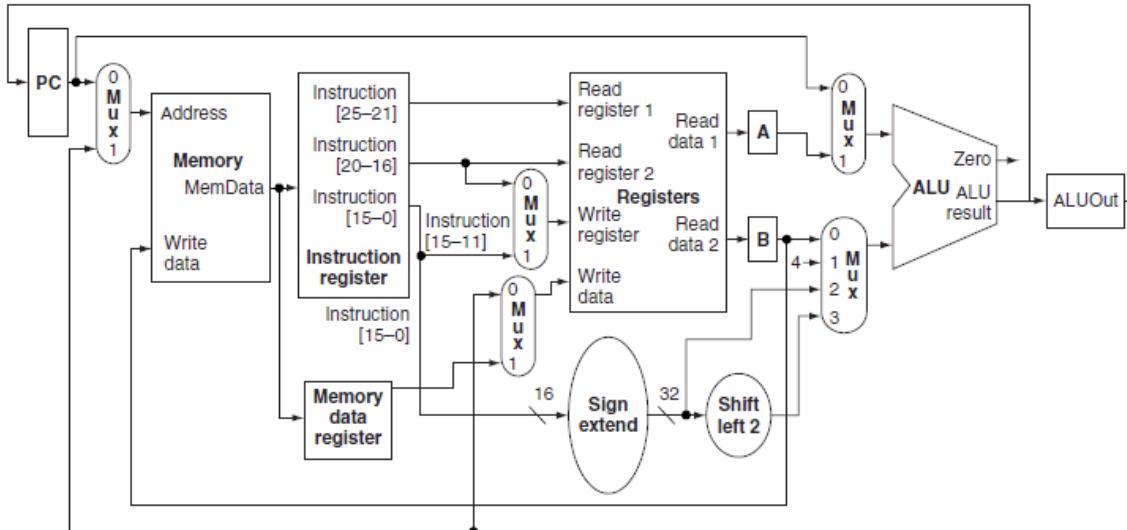
הוספת אסט-ים – איחוד הרכיבים (ALU ו-ZERO) בתבוצע באמצעות מטאימים, עבור המצביעים השונים:

זיכרון – האם מדובר בכתובת של הפקודה הבאה או במידע (כתובת נתונה)? ○

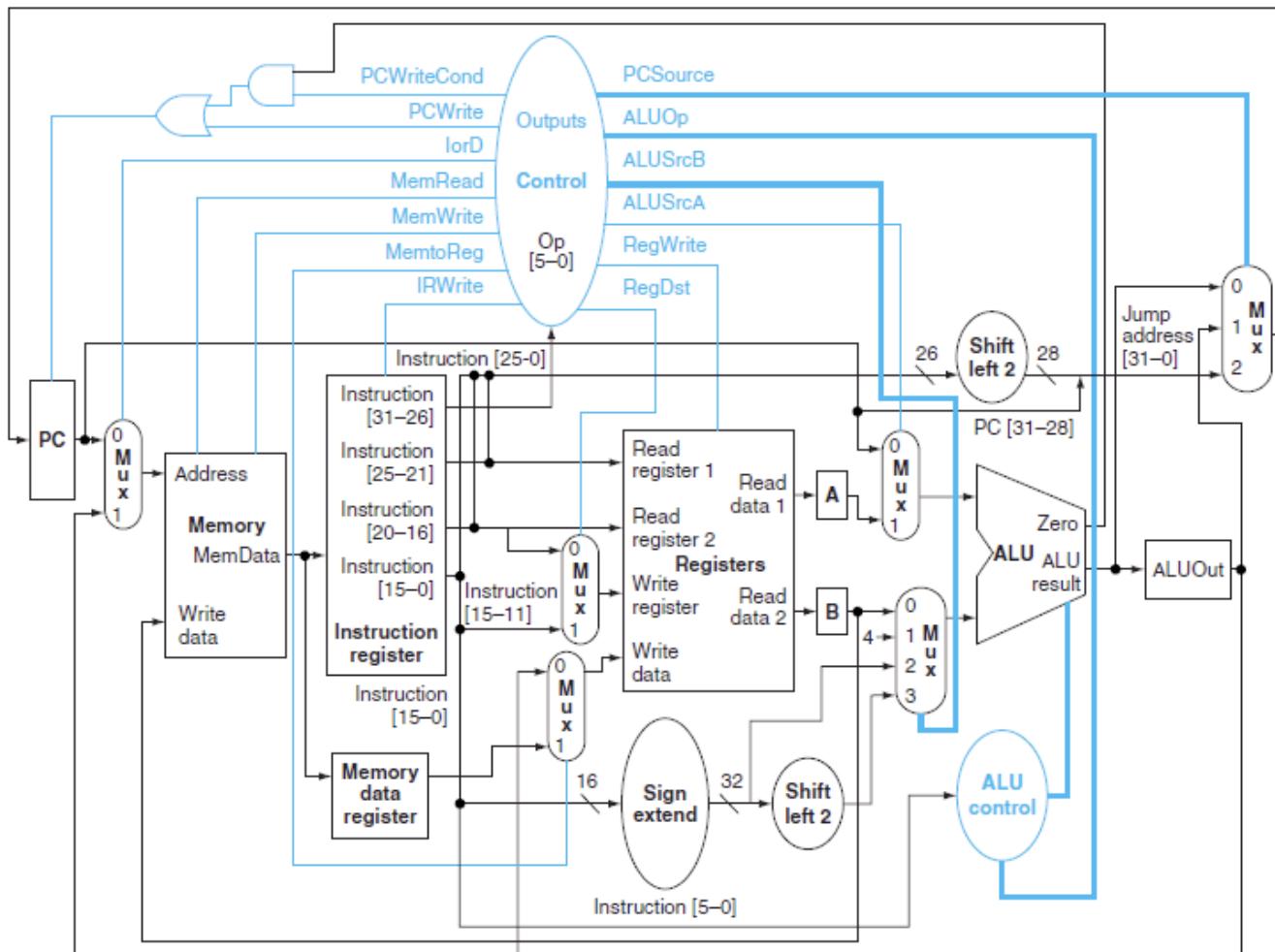
ALU (A) – האם מבצעים חישוב או מדמים את ה-PC? ○

ALU (B) – האם מבצעים חישובים על רגיסטרים, חישוב כתובות גישה ל זיכרון, או חישוב כתובות קפיצה? נשים לב שיש שם גם את הערך הקבוע 4 עבור חישוב 4.

PC+4



ה-ctrl עם קווי בקרה החדשים:



אותות הבקשה:

משמעות	אות בקרה
כתיבה לרגיטר ז' (I) או rd (R)	RegDest
האם יש כתיבה לרגיטר היעד	RegWrite
האם יש קריאה מהזיכרון	MemRead
האם יש כתיבה לזכורן	MemWrite
האם כתיבה לקובץ הרגיטרים היא מהזיכרון או מה-ALU	MemToReg
פעולה ALU – 00 לחיבור, 01 לחיסור, 10 לביצוע func	ALUop
האם יש כתיבה ל-Register	IRWrite
כניסה ALU ראשונה: מרגיטר או PC	AluSrcA
כניסה ALU שנייה: רגיטר (R-type + תנאי beq), 4 (חיבור ל-PC), Imm (חישוב כתובת זכרון), 4*Imm (חישוב כתובת קפיצה)	AluSrcB
כתובת לזכרון: של פקודה או נתון	IorD
האם תבצע כתיבה ל-PC Register	PCWrite
האם הכתיבה ל-PC Register היא מותנית (יהי 1 בפקודות branch)	PCWriteCond
מעבר לפקודה הבאה, מעבר לפקודה מ-ALU (beq), או jump	PCSource

:ALU Control

Signal name	Value (binary)	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSource	00	Output of the ALU (PC + 4) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address (IR[25:0] shifted left 2 bits and concatenated with PC + 4[31:28]) is sent to the PC for writing.

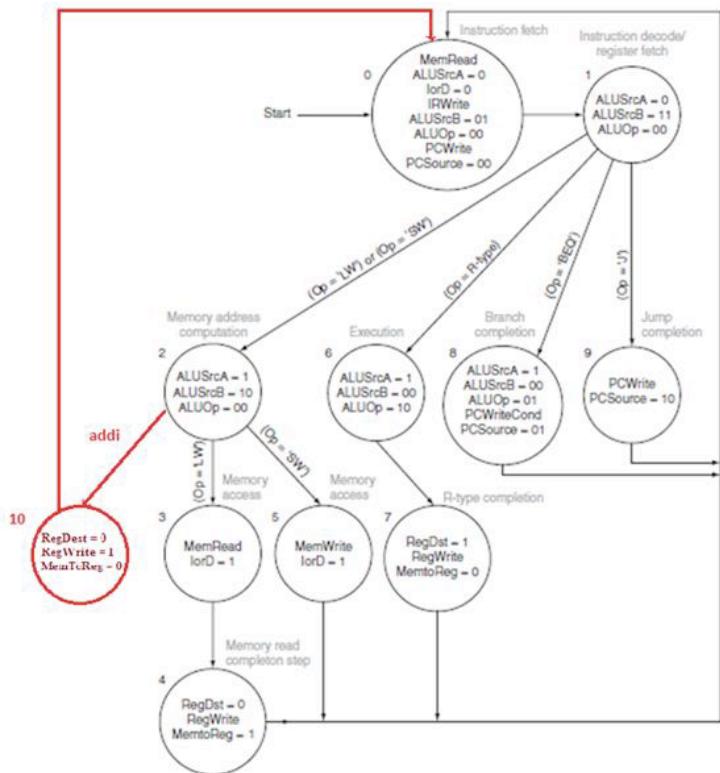
ה-controller יהיה מכונת מצבים. מצב = אותות בקרה עבור פקודה+שלב. זהו משאנו סטטי שלא ניתן לשנות אותו, מכונת מצבים מגדרה לנו אילו אוטו בקרה אמורים להיות במחזור ספציפי של אותה הפקודה. לא כל פקודה מבצעת את כל השלבים, לכן לא כל פקודה נבער בכל המצבים:

מצב	שלב	תיאור
0	fetch	חישוב PC+4 שנכנס חזרה ל-PC, נקראת פקודה מהזיכרון ומאוחסנת ב-IR.
1	decode	מתבצע חיבור של PC+offset*4 (חישוב עתידי ל-beq), קוראים רגיטרים ל-B, A, ALUOut = A + immediate:immediate .MDR = Memory[ALUOut]
2	lw/sw computation	חיבור של רגיטר עם כתובת נתון מהזיכרון:lw
3	lw	קריאה של כתובת נתון מהזיכרון:lw
4	lw write back	בתיבת של המידע שנקרא לתוכן הרגיטר: MDR = MDR .rt = MDR
5	sw	בתיבת של כתובת נתון לתוכן הרגיטר: Memory[ALUOut] = B
6	R-type execution	.ALUOut = A op B: החישוב הדורש לפקודה ב-ALU
7	R-type completion	בתיבת של התוצאה משלב 6 לתוך קובץ הרגיטרים.
8	beq	מחסרים בין הרגיטרים ובודקים את תנאי הקפיצה (הכתובת חושבה קודם).
9	jump	בתיבת של שדה ה-jump ל-PC.

תמייה בפקודות נוספות

מעדים לעשות שינויים רק ב-controller: זה שינויים במוכנות מצבים, והתקנת אוטות בקרה פר מצב. **נוסיף תמייה ב-addi.** נצורך להוסיף את הפקודת למוכנות המצבים.

- המצים 0,1,2 מתאימים לפקודת addi.
- יש צורך להוסיף מצב חדש, מצב מס' 10 שיצא ממצב 2 ובו תבוצע הכתיבה לריגיסטר (דומה למצב 7). אמנם, בנויגוד למצב 7 שם הכתיבה הייתה ל-r_{rd}, בפקודת אנחנו addi **נותרים את התוצאה ל-rt!** לכן .RegDest = 0
- בדומה למצב 7: .RegWrite = 1, MemToReg = 0

הערכת ביצועים:

- נחשב את ה-CPU על ידי מכפלה של הגורמים הבאים: IC (כמות הפקודות הכלולות), CPI (מספר מחזורי השעון בפועל עבור פקודה), -clock cycle.
- נחשב את ה-PI-CPI באופן הבא: כאשר CPI_i הוא זמן הביצוע הממוצע של פקודה מסווג i . I_i הוא מספר הפקודות מסווג i שבוצעו בתוכנית.

$$CPI = \sum_{i=1}^n CPI_i \times \frac{I_i}{Instruction\ count}$$

- חוק אמדל – נועד לבדוק בכמה אונחים משפרים את זמן הריצה בתוצאה מושני מסויים:

$$Speedup_{overall} = \frac{ExecTime_{old}}{ExecTime_{new}} = \frac{1}{[(1-Fraction_{Enhanced}) + \frac{Fraction_{Enhanced}}{Speedup_{Enhanced}}]}$$

- דוגמה 1 – משפרים את פקודות R פי שניים, המהווים 10% מהתוכנית:

$$ExTime_{new} = ExTime_{old} \times (0.9 + 0.1 / 2) = 0.95 \times ExTime_{old}$$

$$Speedup_{overall} = \frac{1}{0.95} = 1.053$$

- דוגמה 2 – הוראות שטטפלות בנקודה צפה ירצו פי 2.5 יותר מהר (מהוות 15%), גישה לדיברן פי 3 יותר מהר (מהוות 20%), וחיבור/חיסור פי 1.5 יותר לאט (מהוות 40%). נוריד את כל האחדים ששינויו, ונוסיף את כל השיפורים. נשים לב כי עברו הפעולה שריצה פי 1.5 יותר לאט, אנחנו מכפילים ולא מחלקים.

$$Speedup_{overall} = \frac{1}{(1-0.15-0.2-0.4)+0.15/2.5+0.2/3+0.4\times 1.5} \approx 1.024$$



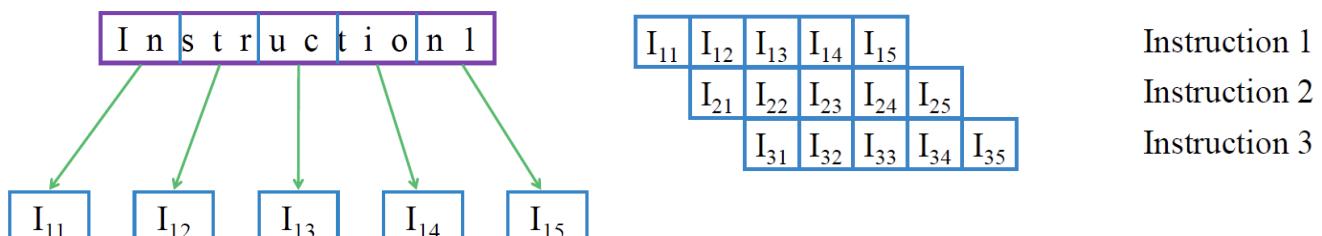
Pipelining

אם נגרום לכל המשאים, לכל הרכיבים במערכת לעבוד כל הזמן, יוכל לקבל את ה-pipeline (כמו בדוגמה הביבסה).

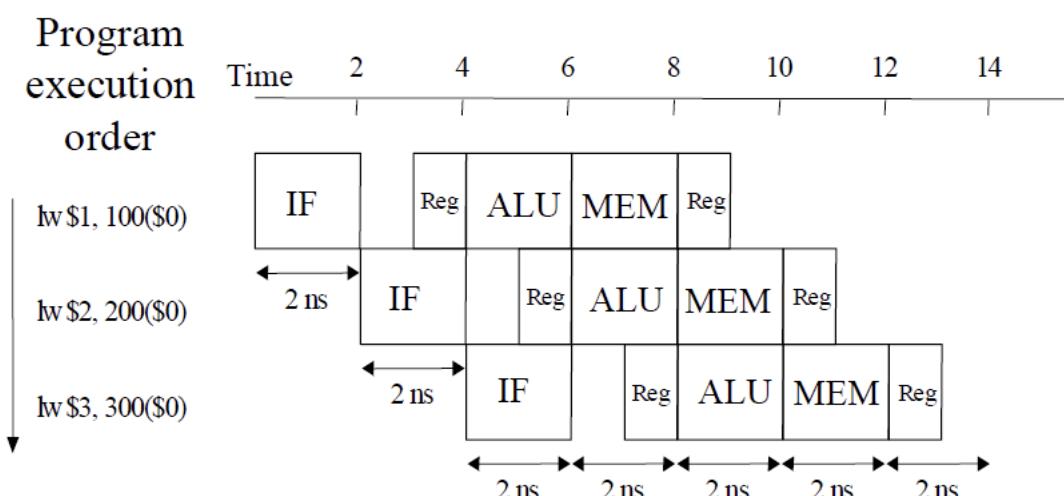
- כמות הזמן מהרגרע שנכנסה פקודה למערכת עד שהיא תצא. **b-Cycle Multi Cycle** הוא יותר טוב (יותר נמוך), כי יש פקודות שאפשר לסיים תוך 3 cycles.
- סוג של ממוצע/amortized throughput, כמה פקודות מסוימים במחזור לאורך כל ההרצה. דומה ל-IPC (כמה instructions可以在一个时钟周期内完成) – פקודה אחת יכולה לצאת החוצה יותר מהר, אבל ב-**b-Pipeline** אפשר לבצע ב-**b-Cycle**.

MIPS Pipeline

נרצה לפחות פקודה לשלבים, ולבצע אותם במקביל:



.fetch → decode → execute → memory → write back: כבר בבראשית



ה-latency של כל פקודה הוא 10ns, וה-throughput הוא $\frac{3}{7}$ בין שלקוח לנו 7 מחזורי לבצע 3 פקודות. בהמשך בהנחה שיש הרבה יותר אחרי השניה זה יהיה פשוט $\frac{1inst}{2ns}$, פקודה 1 כל מחזור.

הערות:

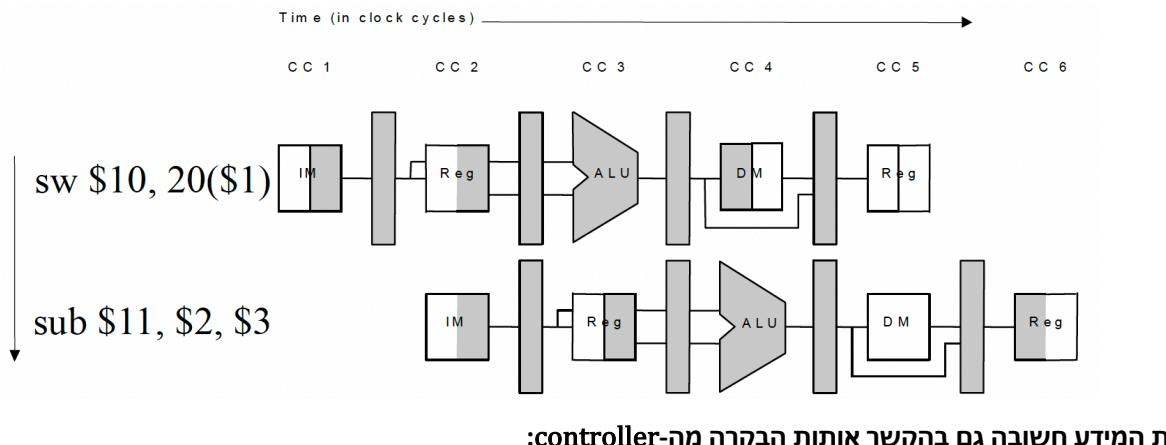
- בהתמשך נעסק בזה: נשים לב כי בתרשימים אנחנו כבר מציררים את זה בכיה שהקדירה מה-Register file קורית בחצי השני של המחזורי, והכטיבה (write back) קורית בחצי הראשון של המחזורי. הגישה ל-Register file יותר מהירה מגישה ל ذיכרון ולבן זה לוקח רק חצי זמן מחזור.
- אנחנו נCallCheck קצר **ארכיטקטורה של ה-single Cycle** (הו לנו שני U: לקידום ה-PC ולשלב ה-execute, ויזכרנו "מפוץ" "Instruction + Data"). האופטימיזציות שביצענו היו עבור Multi Cycle. עם זאת, עדין יש כאן נקודת דמיון ל-**Multi Cycle**, ישנים רגיסטרים ששומרים לנו את המידע בין ה-cycle.

כדי להעביר מידע בין שלבים ב-pipeline, נוסף **רגיסטרים עבור מידע שאינו משתמש בו מיד**, אלא יჩכה לנו למחזור הרלוונטי. כמו שסביר ברגע ל-MC אנחנו לא מרים פקודה-פקודה באופן סינכרוני, אלא במקביל, אנחנו זוקקים למשהו שיחזק לנו את המידע בין השלבים השונים, ולא יהיו מושפע מפקודות שרצות במקביל. **אך** שמאלו הוא לבכתייה, וצד ימין הוא לקרה.

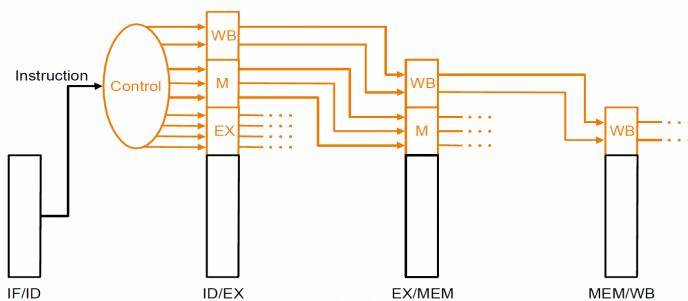
נשים לב שצריך לשמור את שדה ה-pc (הריגסטר אליו אנחנו רוצים לבכתייה) ולנקחת אותו איתנו בכל הדריך. בשנגייל-**write back** אנחנו נדע לאן לבכתייה.



דוגמה נוספת: sw: מבצעים כתיבה לזכרון ולא קרייה, ולא כתובים זהה לרגיטרים אז write back לא עושים שום דבר. לעומת זאת ב-bus שהוא פקודת R, בשלב ה-memory, לא עושים שום דבר ולבסוף כתובים לרגיטר.



שמירת המידע החשובה גם בהקשר אותות הבקשה מה-controller:



ב-MC היה לנו ALU אחד והשתמשנו בו 3 פעמים לאורך חישוב הפקודת: fetch (ביצוע PC+4), decode (חישוב בתובת הקפיצה אם היה צורך לkapo), execute (חישוב). כאן יש לנו ALU אחד ושני adders.

Pipeline Hazards

– מחסום שמנוע מאייתנו לבצע את אחד השלבים ב-cycle המועד לו: **hazard**

- structural hazard – שימוש באותו משאב לשתי פקודות שונות: לא נוכל גם לקרוא ו גם לכתוב לזכרון בו זמנית.
- data hazard – נרצה לבצע פעולה, אבל היא תלויה בתוצאה של פקודה אחרת שעדיין לא הסתיימה.
- control hazard – אם נחליט שנחננו צירcisム לkapo (לבצע פעולה לא לינאריות מביתת הסדר בזיכרון), מה קורה אם מה שעשינו עד כה. רק לקראת סוף ה-pipeline נחליט לבצע branch, מה עם הפקודות שנכנסו בנתויים?

Structural Hazards

1. קרייה/כתביה לזכרון: באוטה נקודת מחזור נרצה לכתוב לזכרון, וגם לבצע fetch ולקראן ממנו.
2. קרייה/כתביה ל-register file: write back register-file: ב- write back:嶙א בשלב decode.

נניח שנרצה לבצע 1000 sw ופקודה אחת אחרת 1000 wl. האם אנחנו עלולים לקרוא את מה שנכתב? במקרים את המידע הנקודט? נניח שנחננו רוחצים לכתוב את המידיע 17, וברגיסטר ברגע 5. בתחילת מחרור השעון, המספר 5 עדיין יושב ברגיסטר, אך יש לנו את MEM/EXデータ. שמאנו יוצא data לכתובת שנחננו רוחצים לקרוא, והוא מהכתובת 5. במקביל יש לנו את MEM/WB שאותו לכתוב את 17. זה עדיין לא נכנס ל-FF, הוא עומד ומחייב בכניסה. רק בעליית השעון הבאה, ה-5 יתחלף ב-17. הבעיה: אם **באותו מחזור יש לנו גם גם כתיבה וגם קרייה של אותו רגיסטר – מה שנתקבל בקרייה הוא את המידע הישן!** הכתיבה נכתב בפועל רק בעליית המחרור הבאה, בסוף המחרור הנכובי המידע יהיה פשוט מוקן לכתביה, עומד ומחייב.

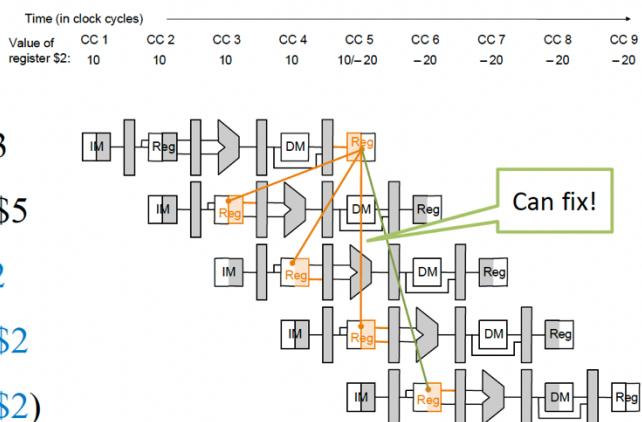
פתרון: כדי לאפשר קרייה וכתיבה לרגיטרים שונים בו זמנית, נדרש להפריד פורטוי קרייה וכתיבה. **בבעץ bypass.** יש לנו לוגיקה שבודקת ה-write register וה-read register האם אותה הכתובת, אל תיקח את ה-data מהזיכרון **אלא תיקח אותו מה-data.** נסיף כאן אונט שיעזר לבחור האם צריך לנתק את ה-data שעתידי להיבט בבר עכשו. זה מאפשר לנו להציג יותר מהר, לקרוא data שבסופול עדיין לא נרשם.

Data Hazards

- .1. (Read after Write) RAW: נרצה לקרוא מרגיסטר שקדם בכתב.
- .2. (Write after Read) WAR: נרצה לכתב לרגיסטר שקדם לקרוא – אין בעיה ב-pipelining.
- .3. (Write after Write) WAW: נרצה לכתב לרגיסטר שקדם נכתב – אין בעיה ב-pipelining.

:register file

sub \$2, \$1, \$3
and \$12, \$2, \$5
or \$13, \$6, \$2
and \$14, \$2, \$2
sw \$15, 100(\$2)



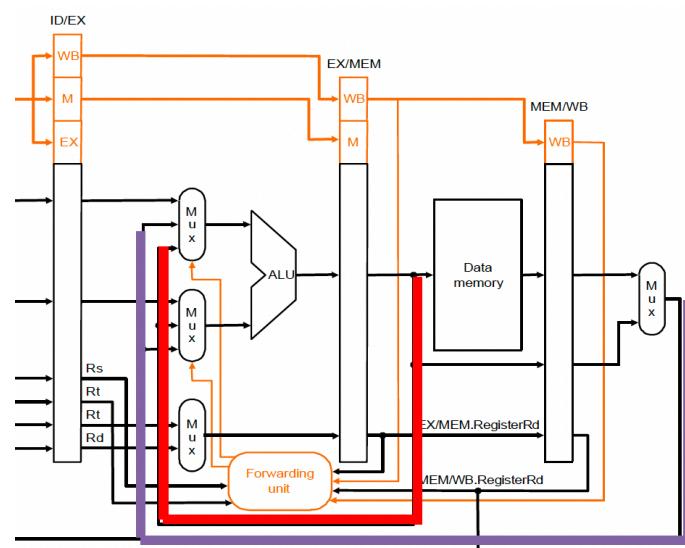
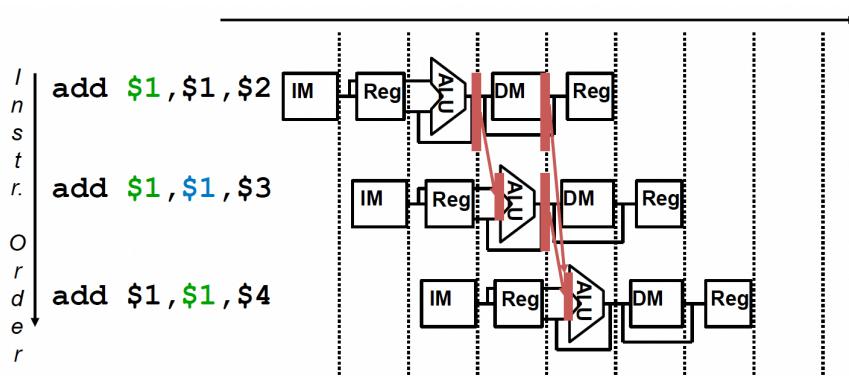
נתיחיל מהסוף להתחלה.

- ws – הכתיבה ל-\$2 מתקבצת במחזור 5 אבל הכתיבה מתבצעת במחזור 6, לכן ה-data בבר כתוב: **אין בעיה**.
- or – אנחנו בבר קוראים מ-\$2 אבל הוא עוד לא נשמר ברגיסטר. אבל, אפשר לעשות bypass ישירות ל-ALU (אחרי הריגיסטר), ולא לקרוא ערך ישן: **פתרנו**.
- and – כאן אנחנו עוד בשלב אחרת, או אפשר לרשום את הערך זמן מוחזר אחד קודם: **בעיה**.
- אמנם, אפשר לעשות bypass ספציפיibi מדויב בפקודה מסווג R (פקודת sw, גם עבר זה היה עובד) זה כבר יושב במבנה ל-DM, אפשר לנקחת ממנו.

:memory

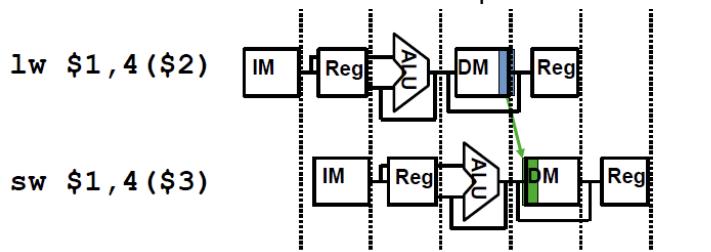
תוכנית: להוסף **sop** בין שתי פקודות תלויות אם אנחנו מזהים מצב של RAW. פתרון קל למימוש, אבל לא יעיל.

– להשתמש בתוצאות זמניות, לא לחכות להם יכטו לתוכה הריגיסטרים. יש לנו יחידה בשם **register forwarding** –_forwarding unit שדואגת לכך. ה-add השלישי לוקח את התוצאה של ה-add השני (לא הראשון), והשני לוקח מהראשון.

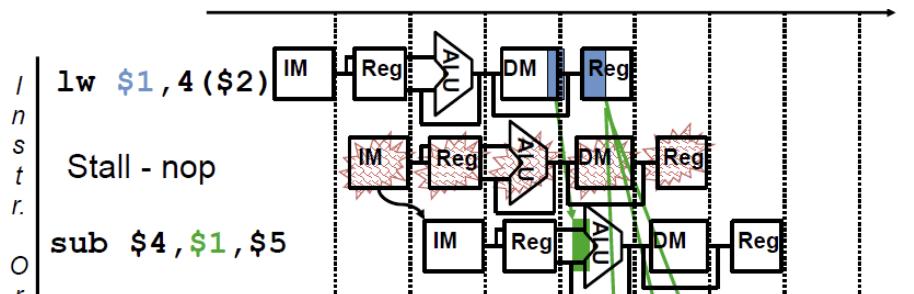




- אותן בעיות קוראות גם ביצירון: Io לכתובת \$1 ואז Aw ש1 \$1, יש לנו AW. בנויגוד לפעולות R וחישוב מעלה רגיסטרים, זו פעולה זיכרון ולכן ה-data לא מוכן ולא יוכל לעשות forwarding, הוא מוכן רק לאחר המעבר דרך הזיכרון. יוכל לעשות forward וולפטור את הבעה מה-DM הראשון לשני.



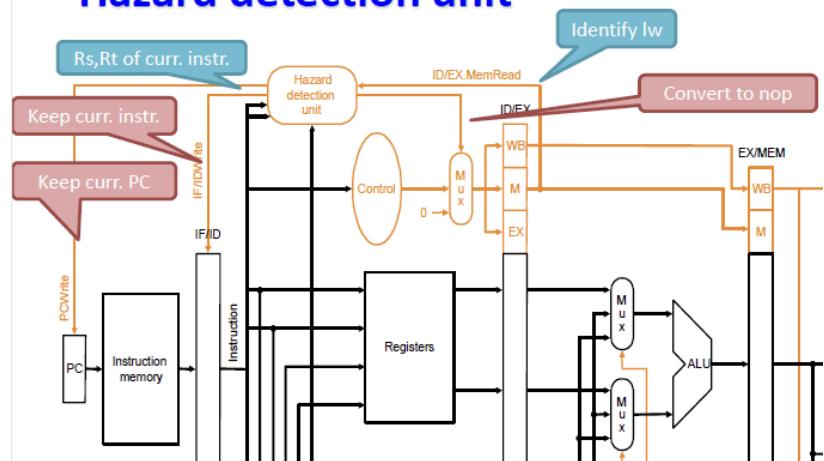
- אבל, אם יש לנו sub אחרי lw, יש לנו load-use data hazard, ולכן נדרש לבצע nop לפני ה-sub. אין לנו ברירה אלא לחכות מוחדר אחד.



ל-nop זהה נקרא load delay slot.

- יש לנו unit hazard שיעודו למשם את ה-nop הזה. הבעה היא שבכל fetch ה-PC מתקדם ב-4, גם ב-4ות.
- אפשר למשם את זה בחומרה: לא צריך לנפח את הפקודות ב-4ות ולהגדיל את הקוד. **אם נזזה את המצביע העייתי (lw) ואחריו פקודה R, יצא ה-PC לא התעדכן!** נשים לב שכן הוספנו את 0 ל-xmu, אם ה-nop יזהה מצב בעייתי, נחלחל קדימה רק את 0, שזו הפקודהnop.

Hazard detection unit



:Control Hazards

עבור פקודות branch ו-jump: אנחנו מכנים באופן לינארי פקודות לתוכה-pipeline. מתי אנחנו יודעים שאחנמו ציריים לקפוץ? רק בסוף שלב execute. יש שתי פקודות שנקנסו ועבורו decode-and-fetch, שאמם אנחנו קופצים לאמורים לבצע אותן.שתי הפקודות הללו צריכים להפוך ל-stall כדי לעצור את ההתקדמות של ה-PC. אפשר להוריד את זה ל-stall אחד.

פתרון ראשון: לא לבצע stall מיידית כאשר אנו רואים פקודה branch, רק אם הבנו שצריך לקפוץ (assume not taken). נשמר את ה-PC המקורי של תחילת התהילה. אם מבצעים את הקפיצה, צריך לטפל בשלושה cycles (שלוש פקודות). אחרת, אין עולות.

פתרון שני: נכניס special branch comparator register file. נקרא מה register file וונחשב את בתובת הקפיצה וגם את תנאי הקפיצה, בשלב decode. לא נחכה ל-ALU בשלב הבא לחישוב. **נקבל את החלטה בסוף ה-decode** בлокם בסוף ה-execute, אז חסכנו לנו cycle אחד.

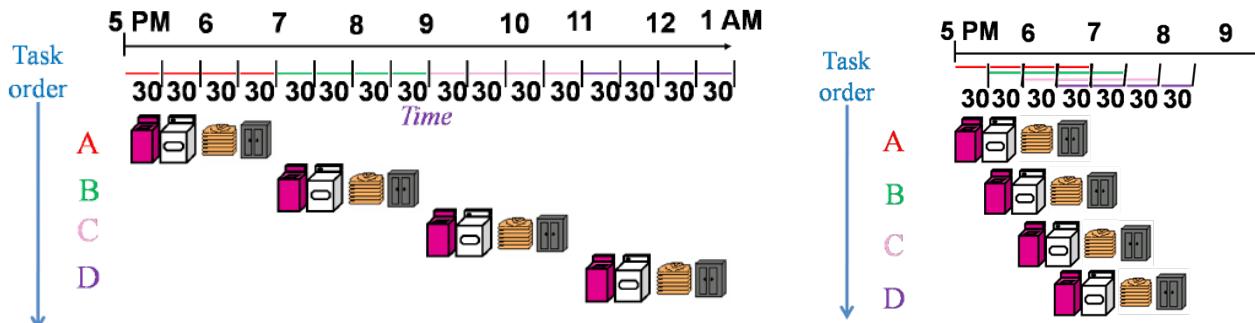
-

(תרגול 9) Pipelining

עקרון ה-Pipeline

- Latency – זמן ביצוע פקודה.
- Throughput – מהירות (מספר תוצאות החישוב ביחידת זמן).
- CPI – מספר מחזורי השעון ממוצע פרט פקודה.

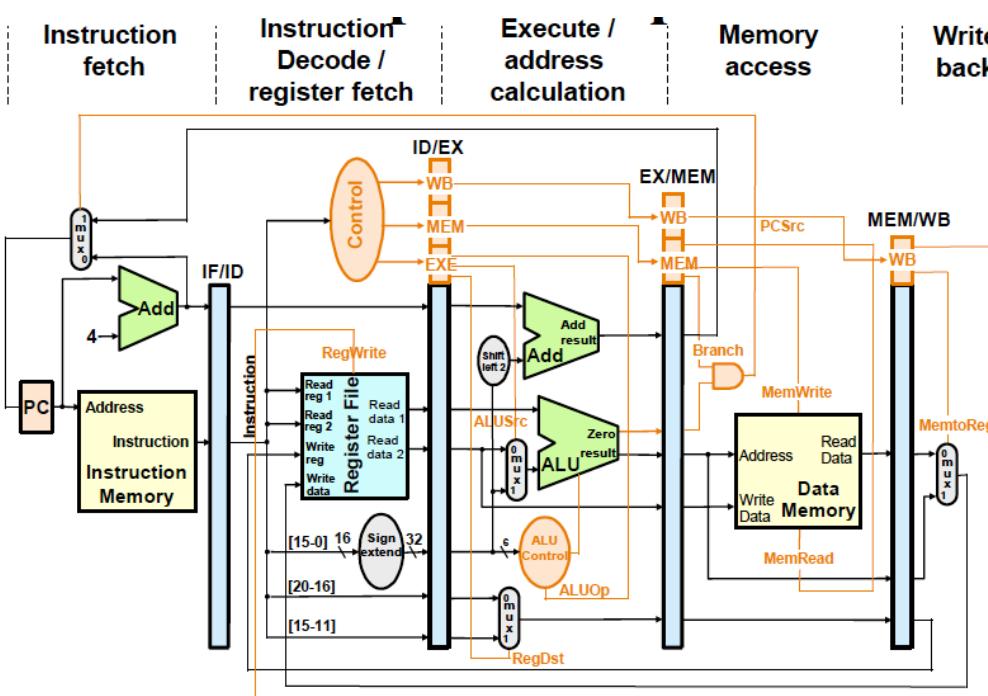
למשל באן, ה-Throughput הוא האפשרות שהוא שעתית (זמן לביצוע בביסה אחת), אבל ה-Throughput משתפר מ"חצי בביסה לשעה (בביסה כל שעתיים), לשתי בביסות לשעה (משייםים תהליך כל חצי שעה).



עקרון ה-pipeline תכלס: עד בה-B-MC/SC לא מתחלים לבצע פקודה חדשה לפני שימושים אחרים בפוקודה ישנה, בעת אנחנו רוצים לבצע כל פקודה ל-5 שלבים **בלתי תלויים**, כך שבcut ניתן לבצע שלבים שונים מפקודות אחרות **באוטו הזמן!**

מימוש Pipeline MIPS

- **יעילות** – כל פקודה תיקח אותו מספר שלבים (5), בחלק מהשלבים פקודות מסוימות לא יעשו כלום. בעצם זה מאפשר לנו נצלות מקסימלית. לא קיירנו זמן ביצוע פקודה מסוימת, אבל המערכת מוציא פקודה כל מחזור שעון. ככל מרוחקו משבי ביונים, כל פעם יש רק **חלק** של החומרה בפעולת (Multi), וכל מחזור שעון (**אחד**, שהוא קצר מאוד) יוצאה פקודה חדשה (Single).
- **אורך שלב pipeline** – נניח שהגיעה לישרונות הפקודות או הנזונים לocket אסן, גישה ל-ALU לocket אסן, וקריאה/ כתיבה לרגיטרים לocket אסן. אורך השלב יהיה אסן. **זמן המחוור יהיה אסן זמן ביצוע הפקודה האחרון ביתר** (רוצים שלכל המערכת יהיה אוטו שעון).

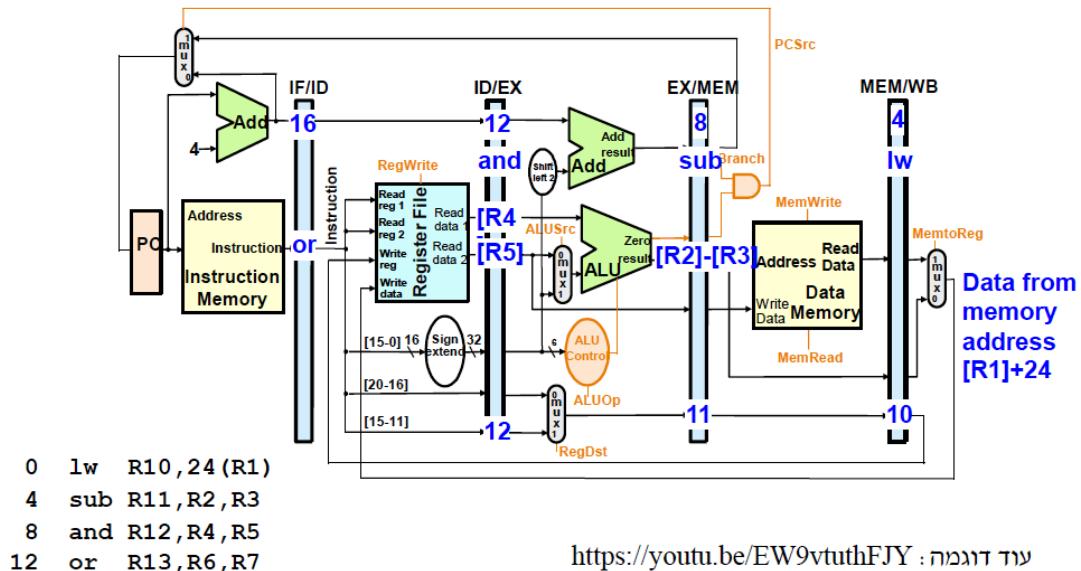


- **ביצוע** – אם פקודה לא משתמשת בחלק מהשלבים, היא תחכה. כל פקודה חייבת לפחות בכל 5 השלבים ואין דילוגים.

- **מימוש** – איך נממש את העקרון? באמצעות הוספה של רגיטרים. כאן אוטה הבקה נשמר בזיכרון קצת שונה. אחרי שלב ה-*fetch-h*-*decode* נשמר רק את מה שנשאר וכו'.

**הבדלים מ-MC:**

- ב-pipeline יש לנו פיצול חזרה של ה-data memory (כמו בארכיטקטורה של SC): ה-data instruction וה-data memory ב-SC.
- שני זכרונות נפרדים.
- ב-pipeline ברגיסטרים בין השלבים לא נשמרת ורק התוצאה של השלב הקודם, אלא גם כל סיביות הבקשה שהפקודה צריכה במלל השלבים הבאים לביצועה. כיוון שבעל מוחזר שנון יוצאה פקודה חדשה, בכל מוחזר שנון ה-controller צריך לטפל בפקודה אחרת ולהעביר את המידע על הפקודה הקודמת הלאה.
- ב-pipeline מספר מוחזרים השנון עברו כל פקודה יהיה קבוע (5), בניגוד ל-MC שם מספר מוחזרים השנון כל פקודה הוא שונה.
- ב-pipeline בכל מוחזר שנון יוצאה פקודה חדשה, בניגוד ל-MC, בה כל פקודה רצתה בנפרד על המעבד ורק במקרה מסוים לבצע פקודה יוצאה הפקודה הבאה.
- ב-pipeline לא ניתן להשתמש במשאבי מחשב בו זמןית ובשלבים שונים, בניגוד ל-MC.

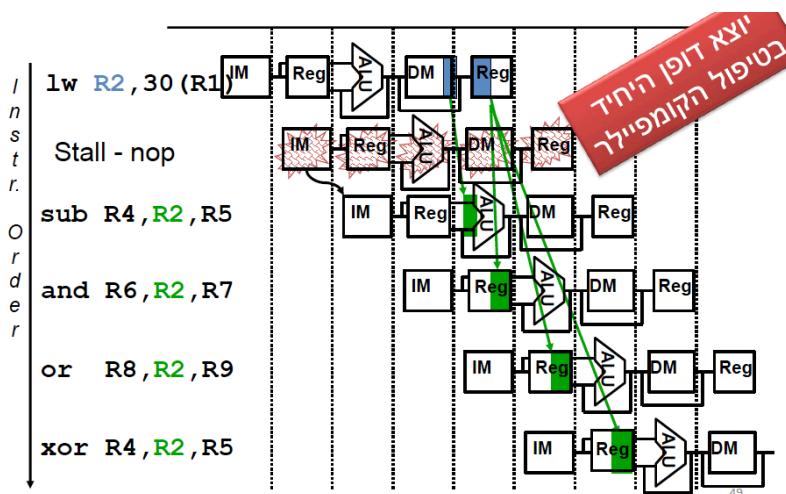
עוד דוגמה : <https://youtu.be/EW9vtuthFJY>**:Hazards**

במעבד ה-pipeline MIPS יש לנו בעיות הנובעות מכך שליעיטים יש תלות בין הפקודות:

- Data Hazard (סיכון נתונים) – כאשר הוראה מאוחרת ב-pipeline תליה בתוצאת הוראה מוקדמת ב-pipeline שעדיין לא מוכנה. הסוג העיקרי שכירgeo בו הוא RAW (Read after Write).
- Structural Hazard (סיכון מבני) – נובע מkonflikt הנוצר כאשר החומרה אינה יכולה להיענות לבקשתות סותרות של שלבים שונים של ה-pipeline.
- Control Hazard (סיכון בקרה) – בתוצאה מביצוע הוראות branch או הוראות אחרות המשנות את ערכו של PC.

:Data Hazard

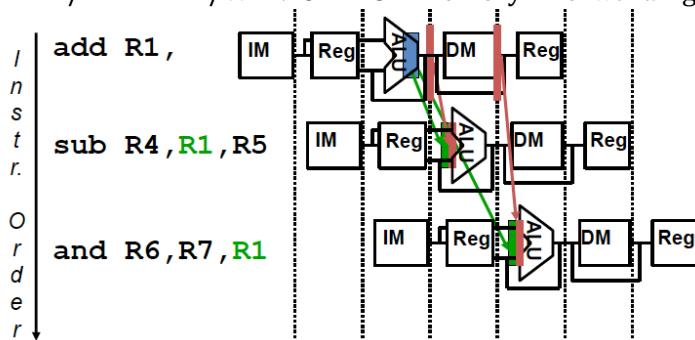
ניתן לפתור את התלוויות בזמן קומpileציה עם הוספת **sopm** בין שתי פקודות תלויות. פתרון שני הוא באמצעות **forwarding** המתבצע ע"י תוספת לחומרה.lopforwarding היא טכניקה המעבירת את התוצאה הנחוצה מהמקום בו היא נוצרת – **ישירות למקום בו היא נוצרת** (לא המתנה לבכיבת הוראות קרייה ומילוי).



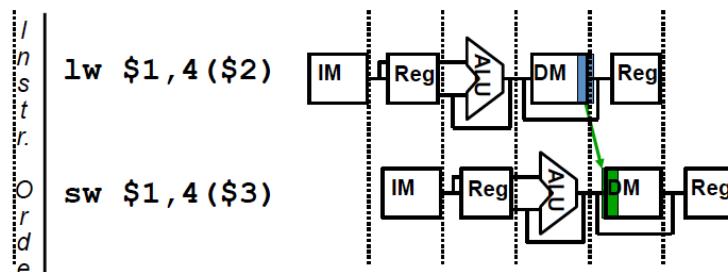
- למשל מ-sub ל-and יכול לבצע forwarding מה-ALU (תוצאה זמנית), לא נוכחה שההתוצאה תירשם.
- **יצא דופן היחיד בטיפול הקומpileר: m-w ל-and** המקרה כבר בעיתי יותר. נדע את התוצאה רק בסוף שלב memory. במקרה זה מוסיפים קצת אחד. memory. במקורה בזיה מוסיפים קצת אחד.
- כאשר יש פקודה קרייה מהזיכרון לתוכן רגיסטר ולאחר מכן פקודה חישובית משתמשת באותו רגיסטר, המידע שאנו צריכים נמצא רק בסוף שלב memory של הפקודה הראשונה, ואנו צריכים אותו לתחלת שלב ה-ALU של הפקודה השנייה. במקרה זה מעוף לבד לא יעזר – נctruck השניה של מוחזר אחד.



- במקרה הבא: בין שתי הפקודות הראשונות נבצע forwarding EX/MEM ל-ALU ל-ID. בין הפקודה הראשונה והשלישית נבצע forwarding memory-mWB ל-ALU מסוג EX/WB ל-ID.



- במקרה של ws לאחר W_o, נבצע memory-forwarding ל-memory forwarding מסוג EX/MEM/WB

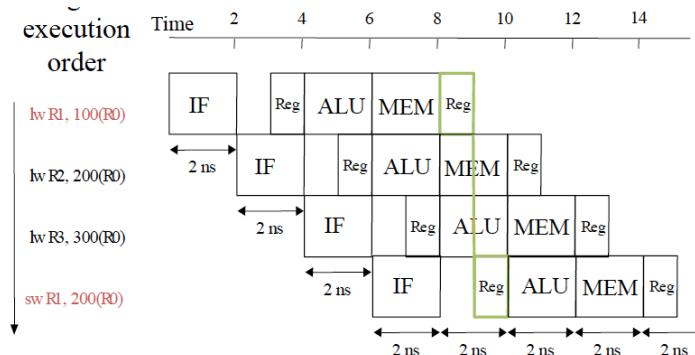


:Structural Hazard

ונוצר בכך חומרה צריכה לתמוך בדרישות סותרות של שלבים שונים ב-pipeline.

- רכיב שאינו מוחלק לשככים בראוי, ונוחז בשככים שונים.
- אי הפרדה בין זיכרון ההוראות לבין הנתונים.
- גישה לקובץ הרגיסטרים לקריאה ולכטיבה באותו מקום שערן.

השניים הראשוניים נמנעים במעבד MIPS pipeline. ניתן למנוע את השלישי ע"י כך שמבצעים תמיד בחצי מחזור השעון הראשון כתיבה, ובבחצי מחזור השעון השני קיראה.



:Control Hazard

בביצוע פקודת branch, יתכן שבמסוף הפקודה לא נקבע ויש להמשיך בבחירה מהמקום שה-PC מצביע אליו, או שכך נקבע ויהיה ערך חדש ל-PC שעדיין לא ידוע. יש לנו בעיה כי עד שלא מתבצעת ההשוואה של ה-branch (בדיקה תנאי הקפיצה) אנו לא יודעים מה הפקודה הבאה שצריכה להזע.

- אפשר לחתך שניים-שלושים stalls (הקפאה). זו פגעה תמידית ב-CPI ולא יעיל.
- אפשר להניח של קופצים – ברגע שמחשבים את כתובות המטרה, קופצים – למעשה טענים פקודות מהמייקום החדש. אפשר להקדים את חישוב כתובת המטרה לשלב decode-all stall. זו הנחה סבורה (בלולאות קופצים הנזון). נשים לב שאם לא קופצנו, הכל הפיך – עוד לא כתבנו מידע.
- אפשר להניח שלא קופצים – וلتענו את הפקודות הבאות. כאן אין צורך לחישוב יעד הקפיצה, וגם כאן אם מגלים שצריך לקפוץ אז זה בסדר – ועוד לא כתבנו שום מידע (בשלב MEM או WB) וכן פשוט מרוקנים את pipeline. בכל מקרה אפשר לבדוק את התנאי בסוף שלב decode (נחסוך מחזור שעון אחד של אי זדאות).
- יש גם שיטה מתחרמת – delayed branch – המימוש שלה לא ברמת המעבד אלא ברמת הקומפיילר.

דוגמאות:

1) נניח ש- $1 = CPI$ באופן אידיאלי, ויש forwarding מלא (ראינו 3 סוגי: $MEM \rightarrow ALU$, $ALU \rightarrow ALU$, $ALU \rightarrow MEM$: forwarding). נניח שבתוכנית יש 20% פקודות branch. נניח שאנו ממקפיאים (ה-pipeline) עוצר עד שכחובת המטרה מחושבת ועד שמתיקבלת החלטה האם לבצע את הקפיצה או לא – עושים 3 stalls).

$$. CPI_{new} = CPI_{ideal} + \frac{avg\ stall}{instr}$$

2) רצים להוסיף לסת הפקודות פעולה כפלה, יש שתי אפשרויות:

(א) להוסיף לשלב execute לוגיקה שתחשיב את פעולה הכפלה.

(ב) להוסיף לשלב נוסף ל-pipeline שבו תהיה לוגיקה שמחשבת את פעולה הכפלה (כך שיהיה בערך 6 שלבים).

ברנאה שמחזר השעון קבוע על פי שלב execute, ושאפשרות (א) מאטאת את השלב הזה ב-20%, מה תהיה ההשפעה על הביצועים בהנחה שאין hazards?

- (א) זמן מחזר השעון יגדל ב-20% וכן latency של פקודה בודדת יגדל אף הוא ב-20%. במנוחת העבודה פר ייחידת זמן throughput (throughput) עברו התוכנית יקטן ב-20%. לפני השני: במנוחת העבודה היא כמעט פקודה 1 כל מחזר שעון. לאחר השני: משומם שמחזר השעון גדל ב-20%, נתקבל כי במנוחת העבודה ביחס למחזר השעון הישן היא $\frac{1}{1.2}$, כלומר האטמו את המערכת ל-83% ממה שהיה קדם.

- (ב) מחזר השעון נשאר אותו דבר. בערך נוסף מחזר שעון עברו כל פקודה. latency של פקודה בודדת יגדל ב-20%. במנוחת העבודה פר ייחידת זמן throughput (throughput) עברו התוכנית לא תשתנה, המערכת תוציא פקודה 1 כל מחזר שעון.

איזו אפשרות יותר יקרה בהנחה שהלוגיקה שמבצעת את פעולה הכפלה היא זהה בשתי האפשרויות?

- אפשרות (ב) יקרה יותר מכיוון שבנוסף לוגיקה לביצוע פעולה הכפלה, הדרישה לשני הפתרונות, מצוריבה הוספת רגיסטר לשלב הנוסף ב-pipeline.

3) עברו קטעי הקוד הבאים, רשמו האם הם יחייבו stall, האם ניתן לפטור אותם בעזרת forwarding או האם ניתן להריםם ללא forwarding stalls.

(א) זה מצב interlock, אחרי פקודה `lw` אנחנו משתמשים בתוכן של הרגיסטר. ככל המוקדם המידע הזה נוצר אחרי שלב `lw`. זה מחייב memory decode, אנחנו צריכים לו בשלב ה-`lw`, אנחנו צריכים `lw` בתחילת ה-`decode`.

`lw $t0,0($t0)`
`add $t1,$t0,$t0`

(ב) לא מפריע, אבל `$t1` ב-`lw` מוגדר `ALU` forwarding מסוג `ALU → MEM`. תוצאה החיבור מוכנה בברור בסוף מחזר השעון השלישי. צריך אותה רק בתחילת המחזר החמישי.

<code>add \$t1,\$t0,\$t0</code>	<code>add \$t1,\$t0,\$t0</code>
<code>addi \$t2,\$t0,5</code>	<code>addi \$t2,\$t0,5</code>
<code>addi \$t4,\$t1,5</code>	<code>addi \$t4,\$t1,5</code>

(ג) אין צורך ב-`stalls` או .forwarding

<code>addi \$t1,\$t0,1</code>	<code>addi \$t1,\$t0,1</code>
<code>addi \$t2,\$t0,2</code>	<code>addi \$t2,\$t0,2</code>
<code>addi \$t3,\$t0,2</code>	<code>addi \$t3,\$t0,2</code>
<code>addi \$t3,\$t0,4</code>	<code>addi \$t3,\$t0,4</code>
<code>addi \$t5,\$t0,5</code>	<code>addi \$t5,\$t0,5</code>



זיכרון

Cache Memory

רקע:

יש 3 מיקומים עיקריים של הזיכרון: זיכרון שיושב בתוך המעבד עצמו – רגיסטרים, cache, RAM – זיכרון העבודה. לבסוף יש את יחידות האחסון – HDD/SSD. ישנו פער גדול בין השיפורים ב-CPU לבין השיפורים ב-זיכרון: המחשבים הולכים ונוהים יותר מהירים, אבל זה לא קורה בזכרון, הוא לא הופך להיות מהיר יותר.

הפתרון הוא Caching – נמחיש את העקרון דרך דוגמת הספרים. ספרים יוכלים להימצא online (amazon), בספרייה, או על-ID המיטה. נשים לב כי ככל שהיא צאת היא בגל גישה מהירה יותר מהקדמתה, אבל מכילה פחות קיבולת של ספרים. ספר שגורץ למשך הרבה ייה בערך הגישה הבי מהירה, ואפשר לנחל את הגישה למידע ולהחליף מקום של ספרים (מה יותר נגש ומה פחות). קראו איזה הרבה מה-CPU על-ID בCaching: להבין איזה מידע לשיט באיזה מחרך מה-CPU ועל-ID בק' למצער את זמן הגישה באופן ממוצע.

היררכיית הזיכרון:

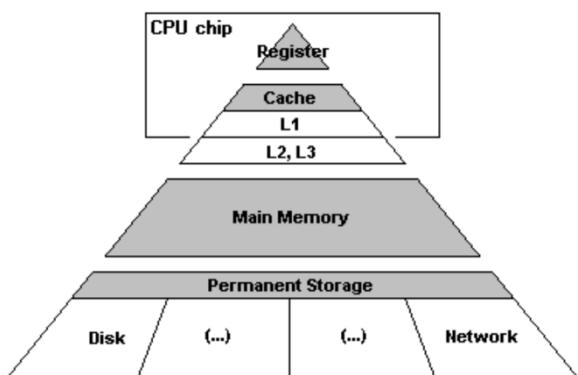


Fig. 2. Typical modern memory hierarchy

יש כמה טכנולוגיות וכל אחת מהן מחייבת נקודת tradeoff בין המחיר לבין המהירות. הדברים הכי יקרים (CPU), לאחר מכן יש לנו RAM, SSD, disk on key, HDD ועוד הללו... זהה ההיררכיה של הזיכרון. השתמש ב-Logic-over DRAM, Cache-over DRAM, Cache-over Main Memory ו-Storage-over DRAM. עבור כל שאר ה-Storage.

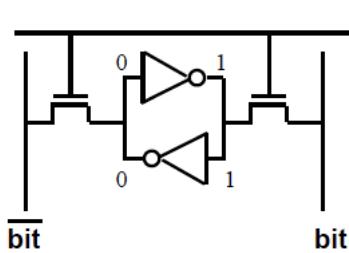
אנחנו שואפים לקבל את ה-capacity של Amazon עם מהירות גישה של השידה שליד המיטה (באיו) שביל פעם שנרצה לקרוא ספר הוא יהיה קרוב אליו. אנחנו רצים למצער את ה-TAT: זמן גישה ממוצע לזכרון.

זיכרון מטמון (Cache):

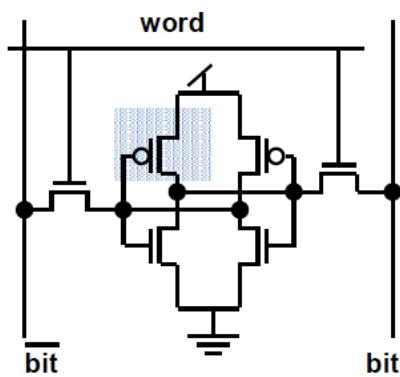
תפקידו להיות קטן, מהיר וקרוב יותר ל-CPU. אנחנו נהנו איזה data צריך לשבת באיזה היררכיה, על-ID מעבר יעיל של data בין היררכיות השונות כדי שהמחשב יוכל לקבל את המידע שהוא רוצה במינימום זמן גישה. ננצל את העובדה שיש לנו מקומיות בזמן ובמקום (space and time locality).

איך בנו זיכרון? ראיינו בבר-FF – ביט זיכרון בודד (בנוי משני latches) שבל אחד עם 8 NAND-ים, סה"כ מגעים ל-36 טרנזיסטורים.

עבור ביט אחד 6 טרנזיסטורים. משתמשים יותר עבור ה-Static SRAM



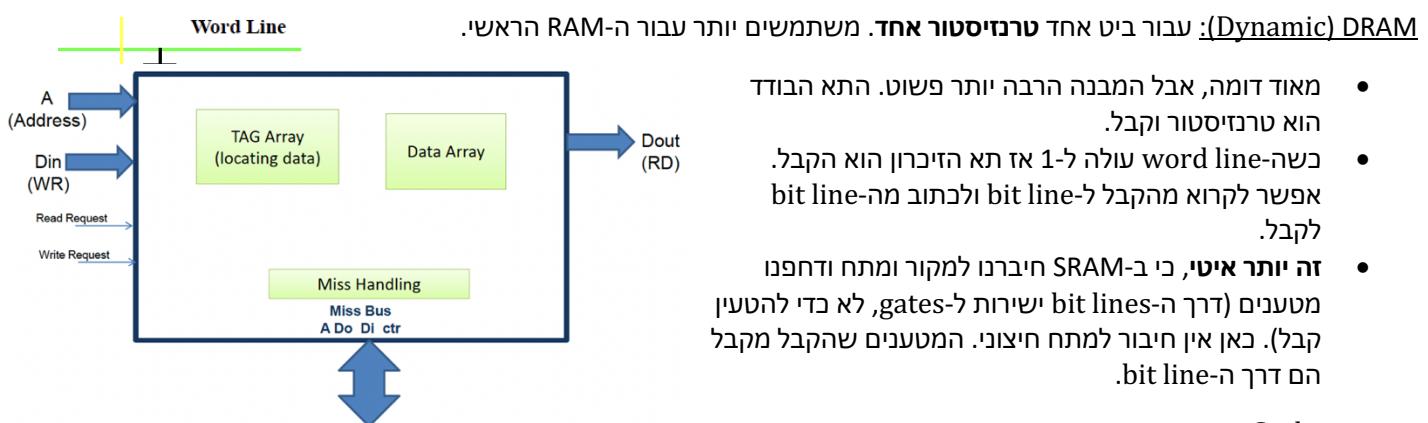
- הכל מתחילה מהלולה של inverters בפנים (זו בעצם זיכרון פשוטה ביחס האפשרית). הבעה היא שאנו יכולים לשלוט על איזה מידע מתייצב בלולאה הזאת. יש לנו bit line ו-word line.
- מה עושים ב-file register? אנחנו לוקחים את data-line ומכנים אותו בכל הרגיסטרים. לוקחים במקביל את address-line ומכנים אותו select-line. select-line יכתוב רק ברגיטר הרצוי כי עליו יש write enable.write enable. בוא זה דומה – ה-data יושב על bit-line והוא משותף לכל ה"רגיסטרים" (ביט מס' 3 בכל הרגיסטרים מקבל את אותו החוט). במקביל ה-word-line הוא כמו write enable ברגיסטרים, האם המילה הזה צריכה להיות מופעלת או לא.



- נזכיר שככל מהפוך (inverter) בנוי משני טרנזיסטורים (במו שראינו בתחלת הקורס ממש). נניח שהיינו רוצים להטען 1 לוגי לתא מסוים בזיכרון. אנחנו מחברים את bit-line למתח (ואת bit-line הפורק לאדמה), ובאשר ה-word-line גם 1 זה מתח. שני הטרנזיסטורים עוברים למצב פעיל. המתח מטען את gates ב-1 מצד ימין. במקביל מהצד השני ה-0 מטען את gates מצד שמאל. אנחנו מאלצים את התא להכיל 1 והלוואה שומרת על הערך הזה.

- למה לא עושים את זה ב-FF? נניח שלפני כן היה שומר 0 בימין ו-1 בשמאלי. מימין אנחנו נגע לקצר. זה מצב בעייתי. הטרנזיסטור הצד ימין חזק יותר, והשמאלי חלש יותר. הם לא באוטו סדרי חזק.

- קריאה: לא דוחפים מידע bit-line. בוחרם word, המידע יצא החוצה (למטה). אם הצד ימין גדול מצד שמאל מוצאים 1 לוגי, ואם הצד שמאל גדול מצד ימין מוצאים 0 לוגי. זה נעשה על ידי ה-sense amp.



עקרונות Cache:

יש שני סוגים של **חוורתיות (locality)** שאפשר לנצל בשיקולי בחירת ה-data. אם יש בתובת שהשתמשתי בה:

- חוורתיות בזמן (temporal locality) – סביר להניח שאשתמש בה שוב בזמן הקרוב.
 - חוורתיות במקום (spatial locality) – סביר להניח שאשתמש בכתובות נסיפות קרובות לה (למשל קטע קוד, מערך).
- קובוצת העבודה (working set) – יש לנו טווח זמנים Δt , כל הכתובות שנגatty בהן באותו הזמן נקראות ה-working set שלנו.

טרמינולוגיה:

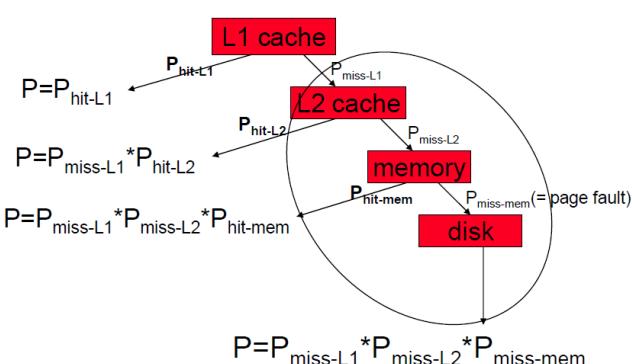
- hit – הספר נמצא בשידה.
- miss – הספר לא כאן, צריך ללבת לספריה ולהביא אותו.
- block – אם נרצה לקרוא את הדף הראשון בספר, אנחנו מבאים את גודל היחידה הבסיסית שנעובד אליה ב-cache.
- cache miss penalty – כמה זמן לוקח לנו ללבת לספריה ולהביא את הספר. ככל מהחזרה שעונת עליה לנו miss.

כיוון ש-not all misses were born equal, אנחנו מסוגים אותם לרמות שונות:

- compulsory – כל גישה ראשונה לבlok בזיכרון היא חייבת לדורות冷 start. זה miss שחייב לדורות.
- capacity – אם היה לי cache יותר גדול, הmiss הזה לא היה קורה.
- conflict – שתי בתובות שאמורות לתמפות לאוות מקום בתחום ה-cache והן דורשות אותה השניה. (הדריך להבדיל בין לסוגים אחרים, הוא על ידי זה שנהפרק את ה-cache בראש שלו ל-fully associative cache, ואם זה עדין קורה זה capacity. אם זה נפתר זה conflict).
- coherency – עד כמה ה-data שיושב ב-cache וזה שיושב בזיכרון תואמים אחד לשני.

ביצוע Cache:

מה הסיכוי ל-hit/miss?



- נניח שיש לנו מעבד שיש לו $CPI = 1.1$ (ideal CPI על כל 1000 פקודות יהו 1100 מחזוריים, ללא miss-im), 50% לפעולה אריתמטית, 30% לקריאה, 10% לכתיבה. מפעולות הזיכרון מקבלות miss עם עונש של 50 מחזורי שעון.
- נחשב את ה-CPI החדש: נוסף את ה-CPI האידיאלי וכוסיף את כמות ה-stall לכל פקודה (בגלל miss). ניקח את 30% של גישה לזיכרון בפועל 10% של miss (על גישה לזכרון). כפול 50 (העונש על כל miss).
- אם נוסיף ש-1% מהפעולות מקבלות miss עם עונש של 50, ונקבל כי 65% מהזמן המעבד ב-stall.

אינטגרציה לתוכר MIPS:

אנחנו צריכים לעטוף את זיכרון ה-cache ביחידות הבאות:

- כניות: בתובת, מידע כתיבה, write/read enabled.
- יציאות: מידע לקריאה.

- $CPI = \text{ideal CPI} + \text{average stalls per instruction} = 1.1(\text{cycle}) + (30\% \text{ (ld/st of instrs)} \times 10\% \text{ (miss prob)} \times 50 \text{ (cycle/miss)}) = 1.1 \text{ cycle} + 1.5 \text{ cycle} = 2.6$
- So 58% of the time (1.5 of 2.6) the processor is stalled waiting for memory

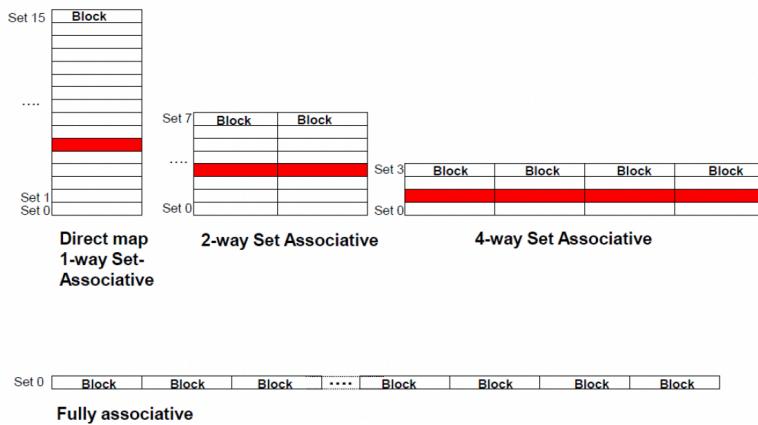


- miss handling
- SRAM: data array
- tag array: מה בתובת המקור בזיכרון הראשי, [-tag bit] (אם המידע אמיתי).

ה-cache הוא שkopf מבית ה-MIPS. אנחנו צריכים להוסיף את **היכולת של cache לדבר עם ה-memory** ש**נמצא מחוץ ל-cache**. הרעיון הוא שככל הסיפור זהה הוא שkopf מבית ה-MIPS, הוא עובד באילו הוא עובד מעל הזיכרון הראשי (מורחב בתובות של הזיכרון הראשי). ה-cache בדרך עשויה משחקים כדי להפוך את התהילה ליותרiesel, ועשויה את כל החולקות הפנימיות. יהיה לנו cache שונה ל-instruction ו-data, הם עובדים בצורה אחרת.

Cache Design

רקע:



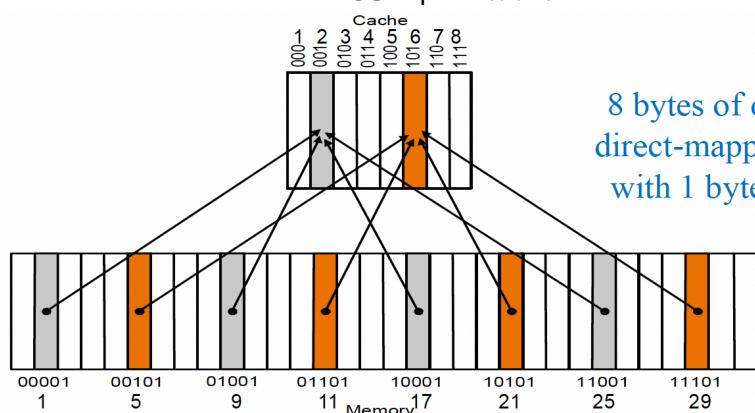
בהתובת מהזיכרון הראשי, איפה אני שם אותה ב-cache? נועד בפונקציית hash, אנחנו מחלקים לתאים.

1. **fully associative** – אין מיפוי, כל בתובת cache על בסיס מקום פנו.
2. **direct map** – כל בתובת בזיכרון יש אך ורק בתובת אחת זהה חוקי למפות אלה. יש מקום מסוים ב-cache ייעודי אחד ויחיד לכל בתובת מהזיכרון. **אם יש שתי בתובות שמתמפות לאוותה נקודה (לפי ה-hash), אחות דורשת את השניה.**
3. **4-way set N** – למשל ב-4-way set – לכל בתובת יש אפשרויות למיפוי, ומחליטים בין ה-4 הללו. צריך לנחל את המיפוי תוך אפשרות.

Direct Mapped Cache

כל בתובת בזיכרון ממופה **לבלוק ספציפי ב-cache**.

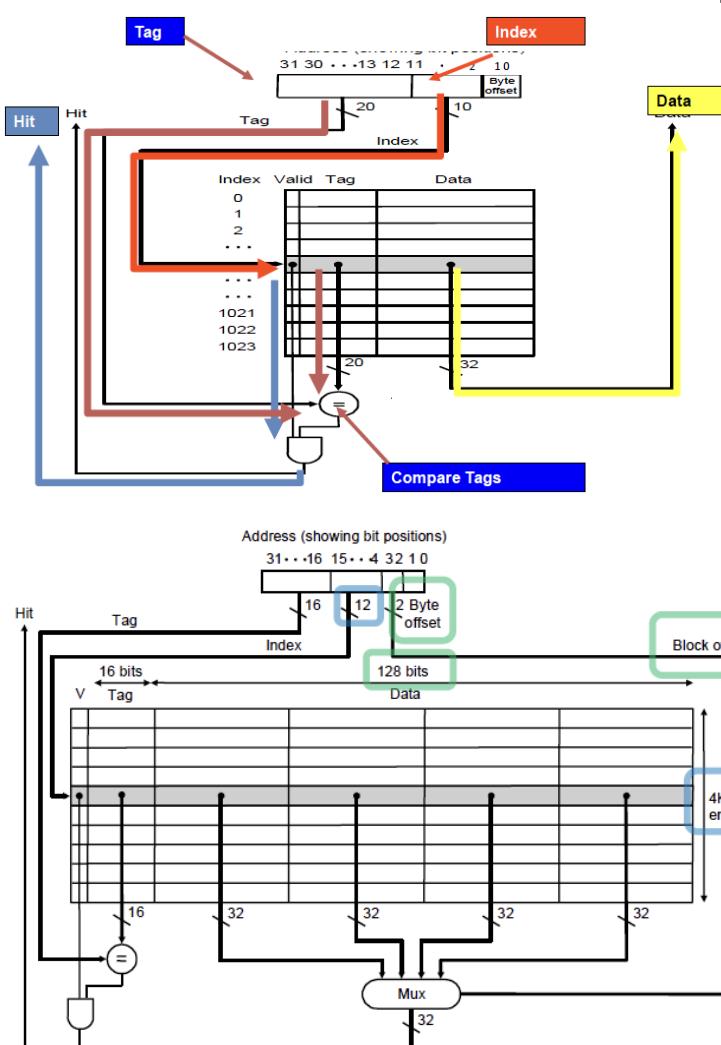
- איך אנחנו קובעים את המיפוי מכתובת בזיכרון לכתובת cache? נניח שיש לנו 8 תאים, כדי לתאר תא הכתובות צירכה להיות 3 ביטים. או שניקח את 3 הביטים הימניים, או את 3 הביטים השמאליים ולהחליט על הכתובות ב-cache. **נדיף** **לקחת את ה-3 הימניים:** הם מעתכדים כאשר אנו מתקדמיים בזיכרון כל פעם, אם נסתכל על השמאליים לא יוכל לראות את השינויי בין הכתובות. ה-3 least significant bits מוצלח(cache). יתרון: פשוטות.
- חסרון: תאים 9, 17, 25, 25 מומופים לאוותה הכתובות ב-cache. יתרון: פשוטות.



8 bytes of data in a direct-mapped cache with 1 byte blocks

טרמינולוגיה:

- **index (גישה לבלוק)** – הכתובות ב-cache שנמפה אליה, לאיזה בלוק לגשת. ביון שככל בלוק הוא 2 bytes וגודל הזיכרון הוא 8 bytes יש **4 בלוקים: נזדקק ל-2 ביטים**.
- **offset (גישה למילה בתוך בלוק, ורק כאשר הבלוק גדול מהמילה)** – נניח שגודל מילה הוא 1 byte, וכל בלוק הוא 2 bytes. נכתוב באן במא offset בתוך הבלוק אני מעוניין, לאיזה מילה לגשת בתוך הבלוק. הבלוק כולל את 0 ו-1, אנחנו רצים רק את 1 (לכן צריך רק בית אחד). תזכורת: **תמיד מוצאים ומכוונים בבלוק במלואו**.



• מאיפה המידע הגיע במקור – כתובת בזיכרון הראשי.

ארכיטקטורה ב-MIPS:

- לכל רשומה ב-cache יש לנו את **המידע עצמו** שהגיע מהזיכרון הראשי (**data**), את **tag** (data), את **tag** (כתובת המקור) וסיבית **valid** (האם המידע נמצא ב-**cache** או לא).
- בහינת כתובת, נרצה לדעת אם היא נמצא ב-cache או לא. נלק לטא הנכון לפי ה-index, ולהשוות את ה-tag שיש ב-cache עם ה-tag של ה-index (וגם מובן pid). איך אנחנו פונים ל-tag של ה-index? comparator, באמצעות mux שיעשב מעל ה-comparator ניקח את ה-tag מה index והזמין מה שיצא החוצה. אם היה Hit נקבל את המידע ונמשיך ל-**RAM**.
- אחרת צריך לגשת ל-**RAM**.

הרחבת לבlokים של 4 מילימ:

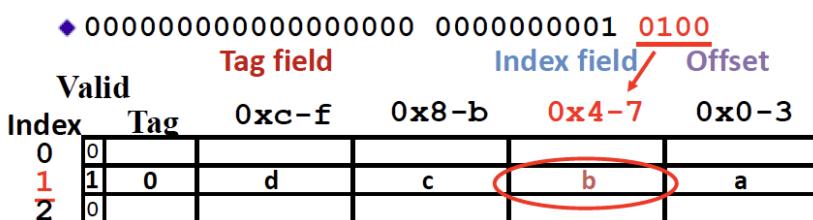
- נלקח את 2 ביטי ה-**offset**, ונשתמש בהם כ-select לעוד mux שבחור בתוך הבלוק את המילה הנכונה (קודם גודל המילה וגודל הבלוק היו אותו דבר, לא היינו צריכים מה היותר בהרחבה? קודם אנחנו מנצלים רק temporal, ועכשו אנחנו מנצלים גם spatial (מביאים את כל הבלוק, כתובות קרובות לכתובת שאנו צריכים).
- מה היתרון בהרחבה? הקודם אנחנו מוציאים רק 16bits, ועכשו אנחנו מוציאים 4K entries.

:cache conflict

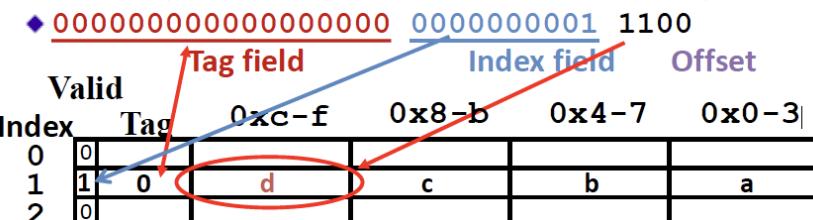
- יש לנו אוסף של גישות לזכירון, ואנחנו רוצים לבדוק מה יהיה ב-cache. גודל ה-cache הוא **16KB** cache. כי כל מילה היא 4 ביטים, ובכל בלוק הוא **4 מילימ (16 ביטים)**. לכן יש לנו $\frac{4KB}{4} = 1024$ בלוקים, **1024 כתובות שונות** ב-cache.
- לכן יהיו לנו 4 ביטים ל-offset (אנחנו עובדים עם מילים, לכן נזדקק ל-2 ביטים בלבד, בתוספת 2 ביטים לאחר מכן shift left), 10 ביטים ל-index, ו-18 ביטים ל-tag. למשל, עבור הכתובת 0x000000014: מדובר בכתובת 20. עלינו לגשת הבלוק השני (באינדקס 1, 16 ביטים), ולהוסיף עוד מילה (4, 1 offset).

Memory Address (hex)	Value of Word	4 Addresses:
		4 Addresses divided (for convenience) into Tag, Index, Byte Offset fields
00000010	a	- 0x000000014, 0x00000001C, 0x000000034, 0x00008014
00000014	b	
00000018	c	
0000001C	d	
...	...	
00000030	e	0000000000000000 0000000001 0100
00000034	f	0000000000000000 0000000001 1100
00000038	g	0000000000000000 0000000001 0100
0000003C	h	0000000000000000 0000000001 0100
...	...	
00008010	i	0000000000000000 0000000001 0100
00008014	j	0000000000000000 0000000001 0100
00008018	k	0000000000000000 0000000001 0100
0000801C	l	0000000000000000 0000000001 0100

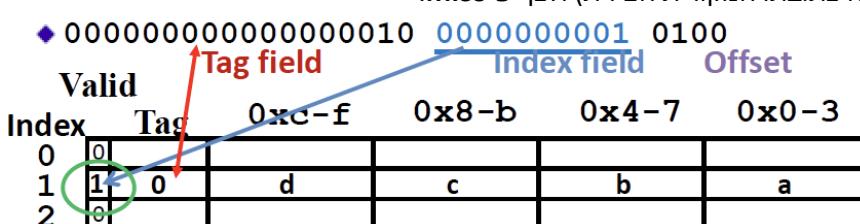
- בשניגש לכתובת הראשונה, היא לא קיימת ב-cache ולכן נשלוף מהזיכרון הראשי, נעדכן את ה-tag ואת ה-valid ל-1.
- לאחר מכן נוכל לגשת ב-offset ולקרוא את המילה הרצiosa בתוך הבלוק.



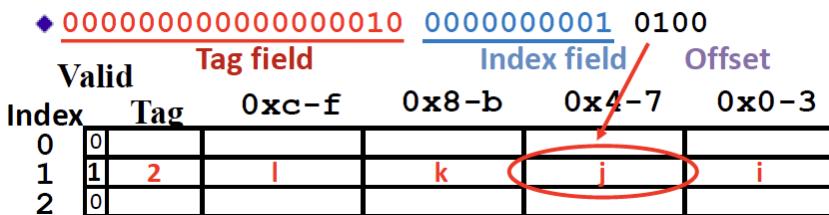
בשניגש לכתובת השנייה, הבלוקobar קיים ב-cache עם 1 valid וכן יהיה לנו HIT (חסכון גישה ליזיכרן!).



בשניגש לכתובת השלישית, נתען את בלוק 4 (באיינדקס 3) כי שקרה בשניגשנו לכתובת הראונונה. בשניגש לכתובת הרביעית, ניפול על אותו הבלוק. אנו רואים שה-tag שונה (תזכורת: מצין לנו מאייפה בזיכרון הראשי. **MISS**).



לכן, נחליף את הבלוק באיןדקס 1 עם מידע חדש מזיכרון ונעדכן את ה-tag.



– זה מקרה של conflict ולא של associativity. אין נפתרור את זה? על ידי העלאה ה-**associativity**. יש עוד הרבה מקום פנוי, לא חסר מקום ב-cache. התכנון לא טוב, אנו כופים את 4 ואת 0 להתמכות באותו בלוק (אסוציאטיביות).

Memory Reference Order →						
Address	0	4	0	4	0	4
hit/miss	M	M	M	M	M	M
0	0	4	0	4	0	4
1						
2						
3						

N-Way Set Associative Cache

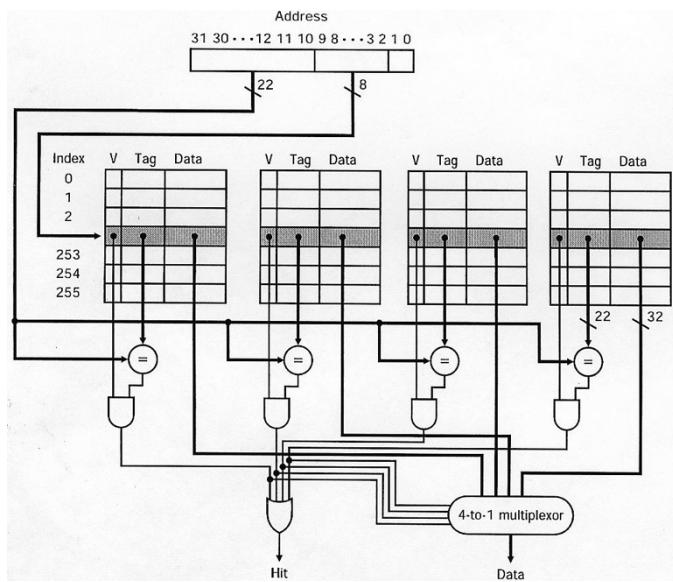
– ההפרק הגמור מ-direct map זה **fully associative**. הכל בביבול בשורה אחת/בטור אחד (או שלכל כתובות יש רק בלוק אחד שהוא יכול להיבנות אליו), או שבל כתובות יכולה לשבת בכל מקום פנוי ב-cache, לבב data tag וצריך להשווות את כל ה-tags בשעושים קרייה. לעומת זאת miss מסוג conflict, אבל העלות היא גבוהה.

אם צורך בשדה index, יש רק tag offset ו-tag. אם יכול להיות שנקלט miss capacity? capacity? כן! אם יש לנו cache שגודלו 4 מילימ', ואנו חסמו ניגשים ל-5 מילימ'. בפועל מוחזרי. אחרי שנכנים 4, יש 4 אחרים שמחליפים אותם. פעם הבאה שנקלט 4 הראשונים נקלט capacity.



Two-way set associative.

Set	Tag	Data	Tag	Data
0				
1				
2				
3				



נמצא את 8 כי הרגע השתמשנו ב-0. ואז בשנחות את 8 שוב יהיה לנו MISS כי אין מקום פנוי ונוציא את 0 (זה מסוג conflict כי הרגעה השתמשנו ב-0). ואז בשנחות את 8 שוב יהיה לנו HIT כי הרגע השתמשנו ב-0.

אם אנחנו ב-set 1 (1 = N) אז יהיה לנו רק MISS-ים.

בתיבה ל-cache: כתיבה יש שתי גישות:

write-through – כל פעם שנכתבו לזכרון, נכתבו אותו גם ל-cache וגם ל-cache בו זמינות.

write-back – הגישה היותר מקובלת. כתיבות קודם כל-cache, מוסיפים bit dirty ב-0 לשם ספק

אינדיקציה שהזיכרון צריך להתעדכן בשמחליפים את הבלוק זה.

- fully N-way set associative

N-way. יש לנו גם מיפוי בהינתן כתובת לאיזה תא היא יכולה ללבת (direct), (fully). אבל במקרה הזה יש מקום למספר בתיבות, אין מקום ספציפי אחד (fully). בשנעשה קריאה נצטרך לבצע חיפוש בתוך כל הכתובות האפשריות בתא (8 comparators). **בעת כאשר נגש ל-cache במקומות index גע ל-set ולא לבlok יחיד: הוא מכיל N בלוקים שכל אחד מהם יש לו (tag, data).** עבור המקרה של $1 = N$ זה direct map (מקרה פרטי).

בכל שנדיל את N, יהיו לנו פחות misses. בבר way-2 חוסף לנו הרבה conflict misses.

באשר קיבל כתובת, נסתכל על ה-index שמודול אותו ל-set המתאים. עבור-set זהה צריך להשווות מול 4 tag-ים ולבן בחומרה ציריך 4 comparators. מתרגמים את זה ל-HIT ביט אחד, ומוצאים גם data ייחיד (בעזרת א斧-ים).

block replacement policy: אם יש MISS?

לקחת את הבלוק מהזיכרון הראשי ולהכנס ל-cache. ברגעוד direct, есть יכולות בחירה בין הבלוקים.

- random – זה לא רע.

- FIFO – לפי סדר הibernה.

- LRU – הכי טוב. מי שהשתמשו בו יותר אחרה.

דוגמה של LRU (נפח ה-cache הוא 4 blocks של 1 byte):

- אם אנחנו נזכיר 0 (fully associative) שיש MISS נכנים את הכתובת, ויש לנו מקום להכניס. אין conflict. אין 2-way associative (2 = N) אז בשנחות לבתוות 6, אין מקום פנוי ב-0 וכאן נוציא את 8 כי הרגע השתמשנו ב-0. ואז בשנחות את 8 שוב יהיה לנו HIT כי הרגע השתמשנו ב-0.

אם אנחנו ב-set 1 (1 = N) אז יהיה לנו רק MISS-ים.

בתיבה ל-cache: כתיבה יש שתי גישות:

write-through – כל פעם שנכתבו לזכרון, נכתבו אותו גם ל-cache וגם ל-cache בו זמינות.

write-back – הגישה היותר מקובלת. כתיבות קודם כל-cache, מוסיפים bit dirty ב-0 לשם ספק

אינדיקציה שהזיכרון צריך להתעדכן בשמחליפים את הבלוק זה.

(תרגול 10 Cache)

הקדמה:

הזיכרון מחולק למספר רמות – התחרונה היא **הגדולה והאיתית ביותר** (disk), והעלונה היא הרמה הקטנה והמהירה ביותר (registers). כל המידע ברמת זיכרון נתונה, נמצא בכל אחת מהרמות שמתוחיה (**עקרון ה-cb**): כל רמת זיכרון מוכלת ברמה הגדולה יותר. אם שינוי מידע מסוים ב-cache ולא שמרנו אותו ברמות שמתוחתי, אז בטעות אפשר לדرس אותן.

- פגיעה (Hit) – מצב שבו רצינו לקרוא מידע והميدע בבר נמצא ב-cache. יחס הפגיעה (Hit rate) הוא אחוז הפגיעה מתוך

כל הגישות לזכרון שהוא לנו.

החסṭאה (Miss) – המידע לא נמצא ב-cache, יש להביא את המידע הנחוצה מהזיכרון ברמה הנמוכה והאיתית יותר. יחס

החסṭאה (Miss rate) הוא אחוז הפגיעה מתוך כל הגישות לזכרון שהוא לנו: $MR = 1 - HR = 1 - Hit Rate$.

- זמן – זמן גישה ל-cache ולזכרון אינם תלויים בינם לבין עצמם, ולכן המטרה שלנו היא להביא למינימום את ה-Miss rate. נסה

- זמן – זמן גישה ל-cache ולזכרון אינם תלויים בינם לבין עצמם, ולכן המטרה שלנו היא להביא למינימום את ה-Miss rate. נסה

- לדאוג לכך שהפקודות והנתונים שאלהם אנחנו ניגשים יהיו ב-cache. ה-Miss rate יושפע ממדדי הibernה וההוצאה.

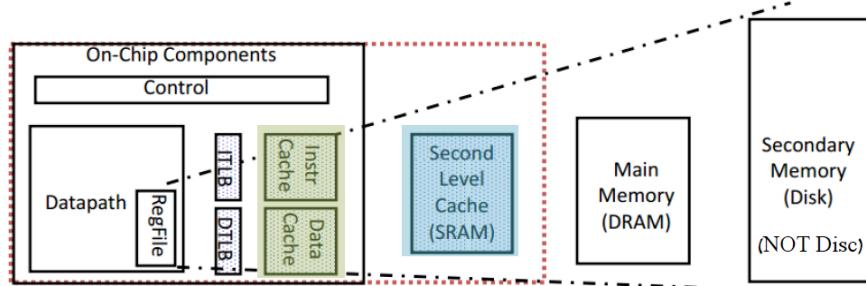
$$\begin{aligned} t_{\text{effective}} &= \text{Hit Rate} \times t_{\text{cache}} + \text{Miss rate} \times (t_{\text{cache}} + t_{\text{mem}}) \\ &= t_{\text{cache}} + \text{Miss rate} \times t_{\text{mem}} \end{aligned}$$

תמיד בשיש גישה לזכרון מבזבזים זמן על t_{cache} בין אם פגענו או החטפנו.

איפה נכנס ה-cache?

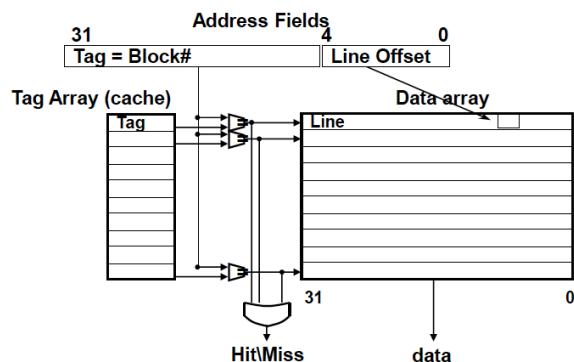
- 1 – שני זיכרונות ברמה ראשונה: לזכרון הפקודות, ולזכרון הנתונים (כמו ב-SC, Pipeline).

2 – זיכרון גדול יותר גם לפקודות וגם נתונים (כמו ב-MC).



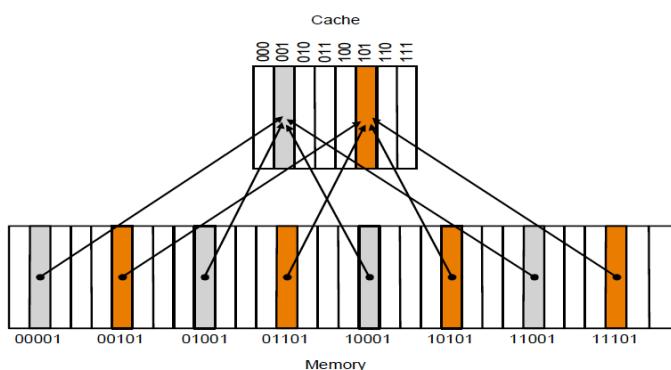
- **מבנה בסיסי** – הזיכרון מחולק לבלוקים. כשמבאים כתובות ל-cache מביבאים את כל הבלוק (בגלל לוקאליות במקום). אנו מחזקים טבלה ובה כתובות הבלוק המקורי (tag), ותוכן הבלוק (data).

:Associative Mapping



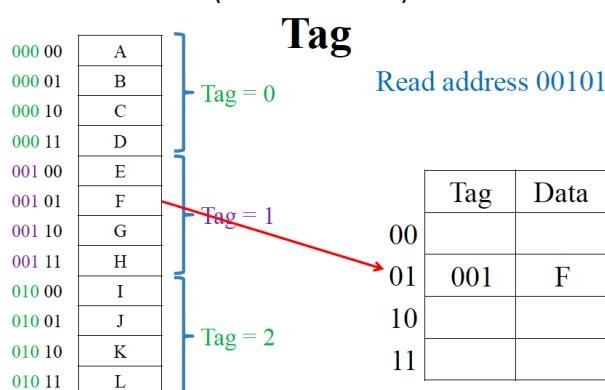
- **כל בלוק יכול להתמפות לכל שורה ב-cache**, **fully associative cache**. data array בתוכו תוכן הבלוק עצמו. הכתובת מניבתאותו לשם.
- **יתרנו** – **יחסית גבוהה**. זאת מפני שתמיד נוכל למצוא את המיקומות הפנויים ב-cache ופושט למלא אותם.
- **חסרנו** – **זמן החיפוש גבוה**, $O(n)$ גבוה. עלינו להשוו את ה-tag של הכתובת המבוקשת עם כל ה-tag-im של הבלוקים.
- **כਮון** אנו שומרים גם ביט valid שマーה לנו האם המידע זהה מעודכן או לא, הוא נמצא בכל השיטות.

:Direct Mapping

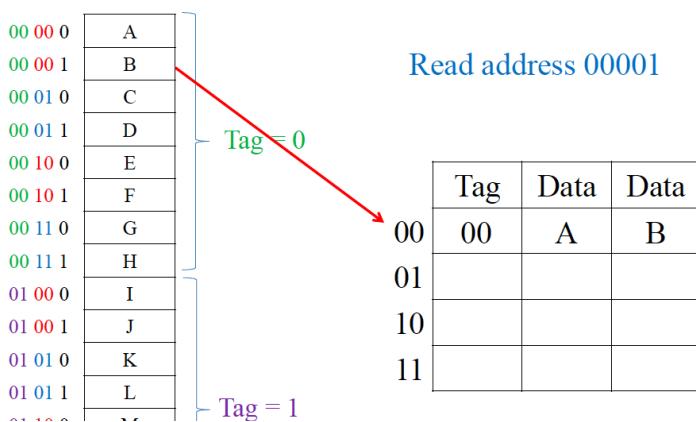


- **כל בלוק יש שורה יחידה ב-cache**, **cache**, למעשה מדובר במודול.
- אם במעמון יש 8 שורות למשל, אז ניקח בלוק מהזיכרון לשורה מס' $block \mod 8$. **לאוטו מקום cache יוכלו להתמפות כמה כתובות מהזיכרון**.
- נחלק את הכתובת בזיכרון ל-3 שדות:
 - offset – באיזה byte byte בתוכו הבלוק נמצא המידע.
 - index – באיזה בלוק הgiיגן נמצא המידע.
 - tag – מאיפה הבלוק הגיגן משמש כדי להבדיל בין מקומות cache.

- **דוגמה 1:** יש לנו 4B של מידע, באשר כל בלוק מכיל 1B. מה גודל השדות אם יש לנו 5 ביטים לייצוג כתובות?
- offset – אין צורך בו, 0 ביטים.
- index – יש לנו 4B לחלק לבלוקים בגודל 1B כלומר 4^2 בלוקים, 2 ביטים.
- tag – מה שנשאר לנו לייצוג הוא 3 ביטים (השMAILIM ביות).

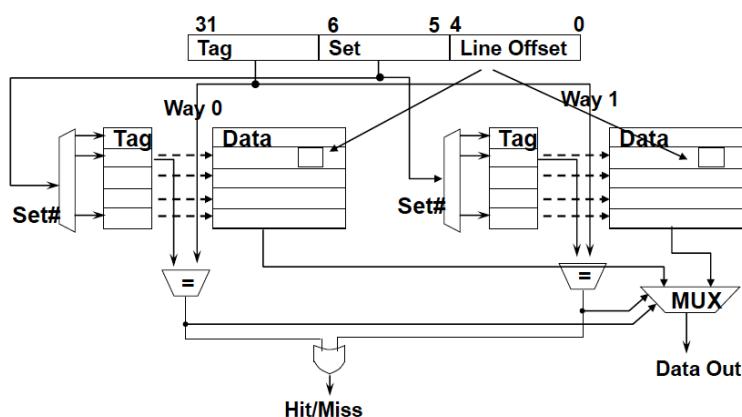


- **דוגמה 2:** יש לנו 8B של מידע, באשר כל השדות אם יש לנו 5 ביטים לייצוג כתובות?
- offset – יש 2 אופציות ל-byte בתוכו הבלוק. لكن נצטרך ביט 1.
- index – יש לנו 8B לחלק לבלוקים של 2B ולכן יש 4 בלוקים ושוב 2 ביטים.
- tag – מה שנשאר לנו לייצוג הוא 2 ביטים.



יתרנו – פשוטות בימוש. ניגשים לשורה הרלוונטיות עם $index$, משווים tag וולוקחים את המידע ב- $offset$.

- חסרו – אם במקרה משתמשים לעיתים תכופות ב-2 כתובות שמתמפות לאותו בלוק cache, יוצר מצב שבו נביא בלאק מהזיכרון, ומיד אחר כך נביא בלוק אחר שידروس אותו וחוזר חיללה.



N-Way Mapping

- בלוק מהזיכרון יומפה לקבוצת (set) בבלוקים ב-cache. בתוך כל קבוצה יש n בלוקים ונitin למפות לכל אחד מ- n הבלוקים. פונקציית המיפוי של בלוק $-set$: $set = \text{block mod } len(\text{set})$ יהיה.(block mod len)(set).
בשנחות כתובות נצטרך להשווות את כל ה-tag-im של כל הבלוקים ב-set.

הרכבת ביצועי זיכרון היררכי:

Average Memory Access Time (AMAT) =

$$\text{hit time} + \text{miss rate} \times \text{miss penalty}$$

מחשב את ה-AMAT (זמן גישה ממוצע) באופן הבא:

- hit time – זמן גישה למידע ב-cache כאשר מצאנו אותו.
- miss rate – החלק היחסני של הגישות ליזכרון שבהן הבלוק שנגענו בו אינו נמצא ב-cache.
- miss penalty – הזמן שלוקח להביא בלוק שלא נמצא ב-cache.

איך נமצער את ה-AMAT?

- הקטנת miss rate – על ידי שימוש במדיניות מיפוי מתאימה ובמדיניות הכנסה ופינוי מה-cache.
- Multilevel Cache – miss penalty

Multi-Level Cache

Average Memory Access Time (AMAT) =

$$L_1 \text{hit time} + L_1 \text{miss rate} \times L_1 \text{miss penalty}$$

איך מחשבים את L_1 miss penalty?

L_1 miss penalty =

$$L_2 \text{hit time} + L_2 \text{miss rate} \times L_2 \text{miss penalty}$$

- תמיד ניגשים קודם כל לרמת ה-cache היב עליונה.
- אם לא מצויים, ממשיכים ברמות הזיכרון עד שmagiumים לבתון (יכול להיות ב-L2, ב-RAM וכו').
- אם הגישה היא לקריאה/כתיבה – מבאים את הנתון ל-L1 (ולכל הרמות שמעלוי, אם צרי, כדי לשמור על עקרון ההכללה).
- נניח כי $L1_{ht} = 1 \text{ cycle}$, $L1_{mr} = 5\%$, $L2_{ht} = .5 \text{ cycles}$, $L2_{mr} = 15\%$, $L2_{mp} = 200 \text{ cycles}$
- אם L2: קיבל כי $L1_{mp} = 5 + 0.15 \cdot 200 = 35$, $AMAT = 1 + 0.05 \cdot 35 = 2.75$
- בלי L2: מתקיים $L1_{mp} = 200$ ולכן $AMAT = 1 + 0.05 \cdot 200 = 11$

דוגמאות:

תרגיל 1 – נניח שיש לנו data cache בגודל 8KB, גודל הבלוק הינו 32B וגודל הכתובת בזיכרון הראשי הינה 32bit. שרטטו את חלוקת הכתובת (32 סיביות) לשדות השונים (tag, index, offset).

ראשית נשים לב שמספר הסיביות עבור offset זהות עבור כל המקרים כי נקבעות על סמך גודל הבלוק בלבד. גודל הבלוק הוא⁵ בתים, לכן ציריך 5 ביטים לשדה offset.

- :direct map
- index – עליינו לחלק KB 8 בגודל כל בלוק שהוא 32B: נקבל $\frac{2^{13}}{2^5} = 2^8$ בלוקים, נציריך 8 ביטים לשדה index.
- – שארית הביטים היא $19 = 8 - 5 - 32$.
- :fully associative
- index – מספר הביטים הוא 0 – ניתן להתאמות לכל בלוק/cache.
- – נישאר עם $27 = 32 - 5$.
- :8-way associative
- index – במתו ה-set-im הוא מספר הבלוקים (чисלנו קודם) חלקו: $8 = \frac{2^8}{2^3}$. לכן נציריך 5 ביטים.
- – נישאר עם $22 = 32 - 5 - 5$.

תרגיל 2 – כמה ביטים בסה"כ נדרש בשביל direct map עם 16KB של נתונים ובלוקים בגודל 4 מילימ'ם? בהנחה שמדובר בארכיטקטורה עם כתובות בגודל 32 סיביות.

- עبور כל בלוק-cache יש לנו את הבלוק עצמו, ביט valid, ביט tag, ואת שדה ה-tag של אותו הבלוק הלוקה מהזיכרון.
- לבן, מספר הביטים הכלול הנחוץ הוא: $(1 + \text{len(blocks)} \cdot (\text{block}_\text{bits} + \text{tag}_\text{bits}) \cdot \dots)$
- מספר הבלוקים:

$$\text{len(blocks)} = \frac{\text{cache}_\text{size}}{\text{block}_\text{size}} = \frac{16\text{KB}}{4\text{W}} = \frac{16 \cdot 2^{10} \text{bytes}}{16 \text{bytes}} = 2^{10} \text{ bytes}$$

$$.4\text{W} = 16\text{Bytes} = 2^4 \cdot 2^3 \text{bits} = 2^7 \text{bits}$$

מספר הביטים בעדרה ה-tag:

- נחשב תחילה את מספר הביטים הנחוצים לשדה ה-index: 10 ביטים כי יש לנו 2^2 בלוקים.
- עبور שדה ה-offset: כל בלוק הוא בגודל 16 = 2^4 בתים, לכן ציריך 4 ביטים.
- נישאר עbor ה-tag עם $18 = 4 - 10 - 32$.
- קיבלנו: $2^{17} = (2^7 + 19) \cdot 2^{10} = 2^{10} \cdot 19 + 2^{10} \cdot 2^7$ ביטים.

Virtual Memoryהע:

בין הזיכרון הראשי לדיסק, יושב עוד מנגנון של caching בשם virtual memory (ה-RAM הוא בסיס הcache של ה-HD). מה אנחנו רוצים מה-virtual memory, מה הוא בא לפטור?

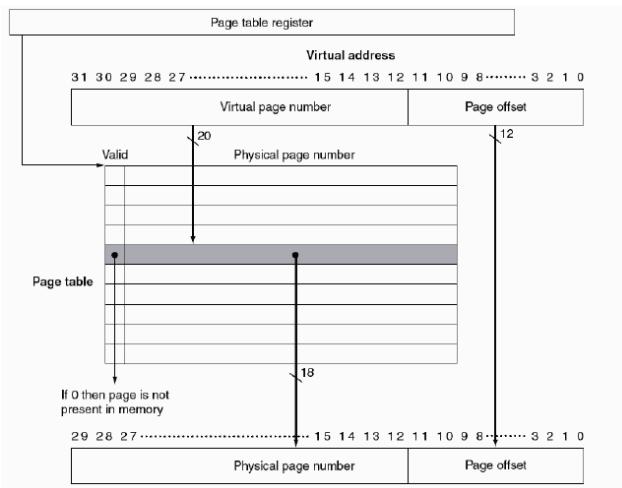
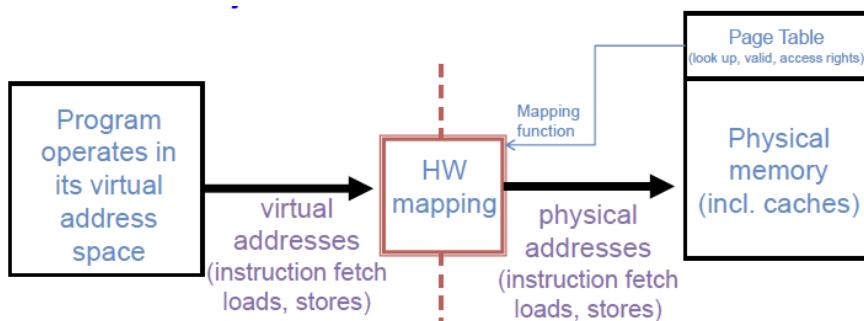
1. **אבלוחה** – לתוכנה אחת לא צריכה להיות גישה לנ נתונים של תוכנה אחרת על אותה מכונה. אנחנו רוצים הפרדה מוחלטת.
2. **אורותוגונליות** – לכל תוכנה יש מרחב כתובות קבוע, לא משנה על איזה מחשב היא רצתה ומה רץ במקביל עליה. לא משנה כמה זיכרון יש לנו בפועל, מבחינת התוכנה יש לה מרחב כתובות קבוע (4GB למשל, לא באמת מוקזית כמות כזאת של זיכרון לכל process). עלינו לתרגם את הכתובת הווירטואלית לכתובת פיזית אמיתית.
3. **סקלביליות** – כל תוכנית מקבל מרחב זיכרון מסוים ללא תלות בסך הזיכרון הזמין הקיים (מרחיב הזיכרון המוצג לה אין בהכרח הזיכרון הפיזי הנגיש לה). לעומת, יש יותר זיכרון שנitin לעבד איתו מהר יותר.

טרמינולוגיה:

- – ה-RAM עצמו, זיכרון אמייתי.
- – אותו מרחב כתובות שהתוכנית רואה, לא באמת קיים.
- **page** – יחידת זיכרון בסיסית שעוברת בין ה-HD לבין ה-RAM, זה מתקבל לבלוק cache-page.
- (רצית) לקרוא בכתובת מסוימת, היא לא נמצאת ב-RAM, אז ציריך להביא את הכתובת ה затה מה-HD).
- **page table** – טבלה שמחפה בין הזיכרון הווירטואלי לפיזי. זה מקביל ל-**tag** cache-page-table (איפה המ庫ר ב-HD).
- – **TLB** (Translation Look-aside Buffer) – page table מגע לו cache-large.



איך זה עובד? יש חומרה שmaps את הכתובת הווירטואלית לכתובת פיזית. כאן סימנו עם הכתובת הווירטואלית. בעת, כתובות פיזיות היא כתובות של ה-RAM, ומשם נוכל לבדוק האם היא לא ב-RAM, או לא. אם היא לא ב-RAM, יש page fault ואז מבאים את זה מה-
הו (ב-**cache** RAM הוא cache בעצמו).



:Page Table

יש לנו את היחידה הבסיסית שקוראים לה **page**, אנו מחלקים את הזיכרון ל-pages. זה לא מנהל בחומרה, אלא בתוכנה, لكن כל chunk של זיכרון וירטואלי יכול להוות מיפוי לכל chunk של זיכרון פיזי. נתחזק טבלת מיפוי – **page table**. בטבלה זו:

- יש לנו את היחידה הבסיסית שקוראים לה **page offset**
- **virtual page number** צריך לעבור תרגום

ב-**virtual** = **PageTable[virtual]**. החלק הזה של התרגום צריך לזכור מהר: זה כן קורה בחומרה, אחרת זה לא יעיל.

ב-**page table** אנו שומרים את המיפוי של כתובות וירטואליות (VPN) לכבות פיזיות (PPN). בכל process במערכת הפעלה יש **page table** מסויל. מה קורה בשיש miss? **page fault**? **page fault** לא נמצא ב-**VPN** לא נכון. אז מגיעה מערכת הפעלה וمعدכנת את הטבלה. כל רשותה בטבלה נקראת **PTE** (page table entry): מכיל שדה **Physical Page #**, **Access Rights**, **Valid**, **Ref**, **Dirty**, **Tag**.

:TLB

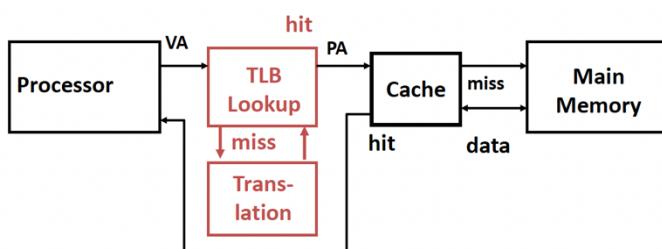
הבעיה: כל גישה ל זיכרון היא גישה כפולה. איך נפתרת את זה? – במאיצעות cache. גם ל-PTE אין סיבה שלא יהיה לו cache משול. ל-cache נקרא **TLB**. הגישה אליו היא מאוד מהירה, והוא מחייב את התרגומים מהכתובות הווירטואליות לפיזיות, כדי להקל על התרגומים: מיפוי בין VPN ל-PPN. **אם יש miss ב-TLB הולכים ל זיכרון ומוציאים את-PTE הנכון**, בולמר מדובר ב-cache בלבד, לא של data. ה-ref counter הינה מתחזק את במות הגישות ל-page זהה מפעם גישה. מאפשר לחשב LRU במקרה של החלפה.

לסיכום, יש לנו cache-ים שונים:

1. **TLB** – מתרגם VPN ל-PPN. זה רק המיפוי של הכתובות. אם יש miss מבאים את ה-PTE מהזיכרון.
2. **RAM Cache** – האם ה-page עצמו (data) נמצא ב-RAM או שציר להביא אותו במיוחד מה-HD.

Comparing the 2 levels of hierarchy

<u>Cache version</u>	<u>Virtual Memory vers.</u>
Block or Line	Page
Miss	Page Fault
Block Size: 32-64B	Page Size: 4K-1GB
Placement:	Fully Associative
Direct Mapped, N-way Set Associative	(in most cases)
Replacement:	Least Recently Used (LRU)
LRU or Random	
Write Through or Write Back	Write Back



On TLB miss, get page table entry from main memory

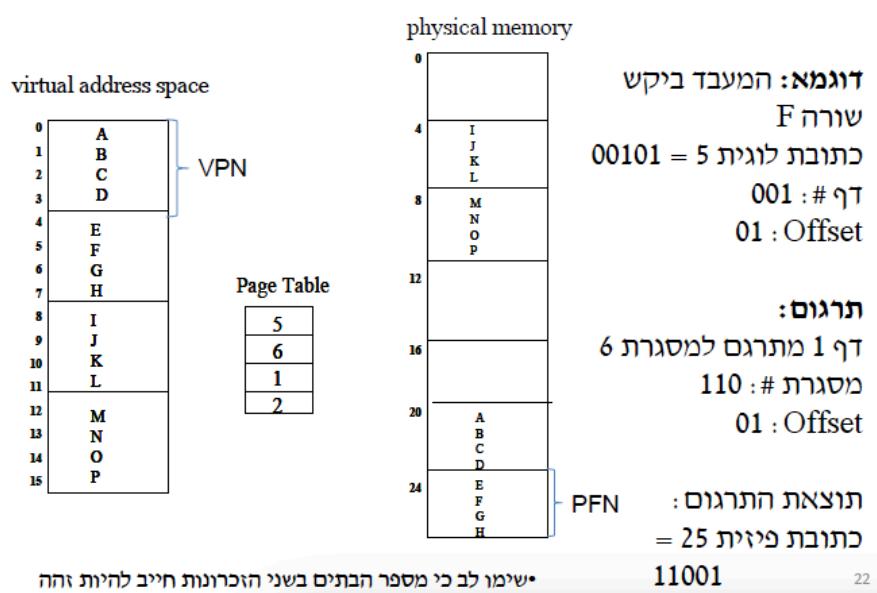
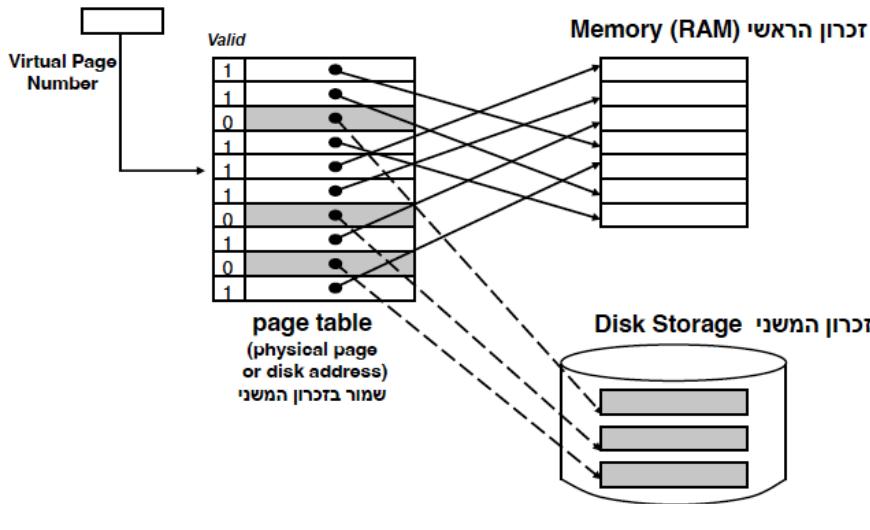


(תרגול 11) Virtual Memory

זיכרון וירטואלי:

מושיביצה – יש בנותה ומוגבלת של זיכרון פיזי, וצריך לחלק אותו בין התהליכים השונים. ה-*virtual memory* הוא טכניקה לניצול והקצאה של הזיכרון, המסתירה את הזיכרון הפיזי של המחשב ומדמה זיכרון רציף וגדול, מפרידה בין ניהול הזיכרון של התהליכים השונים.

תהליך המנסה לגשת לתובות X (ווירטואלית), בפועל יגיש לתובות מתאימה Y (פיזית) בזיכרון הראשי (RAM), או לתובות פיזית אחרת בזיכרון משני (HD) – **במקרה זה המידע המבוקש יישלח מה-HD ווועתק ל-RAM, מהיר יותר, דומה להתנהגות של Cache**. כמובן, לא יקרה מצב שתוכנית תיגש ישירות ל-HD.



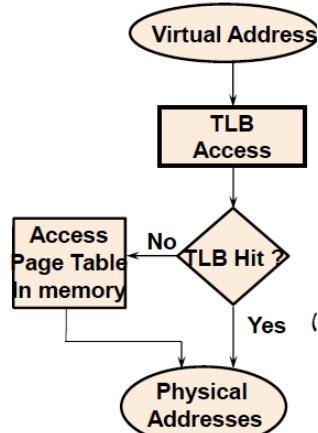
שימוש לב Ci מספר הבטים בשני הזכרונות חייב להיות זהה

- דוגמא: המעבד בิกש**
 - כתובת לוגית 5 = 00101
 - דף # : 001
 - 01 : Offset
 - תרגום :**
 - דף 1 מתרגסם למסגרת 6
 - מסגרת # : 110
 - 01 : Offset
 - תוצאת התרגום :**
 - כתובת פיזית = 25
 - 11001
 - 22
- בזוגה (ספרים החל מ-0) הכתובת הירטואלית היא 5, VPN הוא 1 וה-offset הוא 1. ניגשים ל-**page table** בזורה 1 ומתקבלים את הדף הפיזי 6: 110. מצמידים את ה-offset והוא 11001 וקיים 11001 שזו כתובת פיזית 25.
- פרטי מימוש:**
- **ביט valid** – מודם לנו עברו PTE האם לחתת מידע מה-RAM או מה-HD.
 - **מдинיות write back** – בשרוצים לכתוב מידע ושבירות גבוהה שנדרча לכתוב כמה פעמים לאוטו דף (בניגוד ל-cache, שם הוא מה-RAM), כי הדפים מאד גדולים ושבירות גבוהה שנדרча לכתוב כמה פעמים לאוטו דף (בניגוד ל-cache, שם הוא מה-RAM, גם ל-RAM, וגם ל-HD).
 - **מתי כתובים מ-RAM ל-HD?**
 - כאשר המעבד רוצה לכתוב לכתוב מסויימת ב-RAM אבל המידע נמצא ב-HD (למשל פקודות SW).
 - כאשר הוחלט לפונת את הדף המזכיר **וגם סיבית modified** דלוכה (המידע בדף עודכן מאז שנטען) **וגם** התרחש **page fault**, נצטרך לקחת את המידע מה-RAM ולהעבירו ל-HD, ואז לדרכם את המידע.
 - **זיכרון הפיזי cache** – משתמש ב-*fully associative mapping*, המאפשר גמישות בבחירה הדפים שיוסרו מהזיכרון. כל כתובת בזיכרון הפיזי יכולה להיות שומרה בכל מקום בזיכרון הירטואלי.

עקרון ה-NRU (Not Recently Used): במידה וה-RAM מלא, אילו דפים יוסרו מנתנו בשבייל דפים חדשים? ווסרו דפים שלא היו בשימוש לאחרונה. כדי למשמש אותו, מוסיפים לכל PTE ביט reference – מודלק ע"י המעבד כאשר ניגשים לדף. סה"כ ב-PTE יש לנו 4 דברים: **תרגום בין VPN ל-PPN**, ביט **valid** (המידע ב-RAM או ב-HD), ביט **modified/dirty** (האם המידע ב-HD מעודכן למה שיש ב-RAM), ביט **reference/used** (אייזה דף לא היה בשימוש לאורך זמן).

TLB

לכל בקשת גישה, ניגש ל-PT כדי לבצע תרגום לכתובת הפיזית, ואז ניגש לכתובת הפיזית עצמה. כמובן, כל בקשת גישה דורשת **שתי קוריאות**. לשם כך אנחנו מתחזקים cache ייעודי לשורות מה-PT על בסיס לקאליות בזמן: אם אנחנו מתרגמים כתובות יירטואיות לפיזית, סביר שנצטרך לעשות את זה שוב בעתיד הקרוב. TLB מחולק לשדות tag, offset, index ב-PT. TLB ייחד משותף לכל התהליכיים (חומרתי) ושדה RID.



כאשר רצים לגשת לכתובת יירטואלית מסוימת, ניגשים לתא המתאים ב-TLB:

- אם יש hit, מקבלים את ה-PPN וביחסו ה-offset מקבלים את הכתובת הפיזית המלאה וממשיכים.
- אם יש miss, ניגשים לתא המתאים ב-PT:
 - אם המידע ב-RAM (valid=1), מעלים ל-TLB את התא המתאים ב-PT. ניגשים ל-cache.
 - אם יש hit, סיימנו.
 - אם יש miss, צריך לגשת ל-RAM ולהביא את המידע.
 - אם המידע ב-HD (valid=0), יש fault, מביאים את הדף מה-HD ל-RAM, ומה-cache ל-RAM.

תרגילים:

תרגיל 1: נתון מרחב כתובות וירטואלי ברוחב 32 ביט החולק לדפים בגודל 8KB ו-TLB עם 256 כניסה.

(1) כמה שורות יהיו ב-page table? על פי הנתונים מרחב הכתובות הירטואלי הוא ברוחב 32 ביט, כלומר גודל הזיכרון הירטואלי הוא $B = 2^{32}$. גודל כל דף הוא $8KB = 2^3 \cdot 2^{10}B = 2^{13}B$. מנת הединות ב-TLB הוא $2^8 = 256$. לכן מספר השורות זה מסpter הדפים שיש בזיכרון הירטואלי: $\frac{2^{32}}{2^{13}} = 2^{19}$.

(2) כמה שורות page table יתמפו לכל שורה TLB בהנחה שהיא direct mapped cache? זה מספר השורות ב-page table? direct mapped cache של TLB הוא 2^{19} .

(3) איך נראה חלוקה של虚拟 address של TLB לשדות בהנחה שהיא direct mapped cache?

- offset: בכל דף יש 2^{13} ולכן צריכים 13 ביטים לייצוג ה-offset.
- index: לפי מספר "הקבוצות" שיש ב-TLB בשיטת המיפוי. מספר הקבוצות הנ"ל הוא מספר השורות ב-TLB, שהוא 2^8 ולכן צריכים 8 ביטים לייצוג ה-index.
- tag: נחישב את מה שנשאר $32 - 13 - 8 = 11$.

תרגיל 2: נתון מרחב כתובות וירטואלי של 32 ביט, המחולק לדפים בגודל 4KB,TLB בצורת direct mapped cache בגודל 256. האם הכתובת 0x0FB00ABC מומפה ל-TLB? אם כן, מצאו את הכתובת הפיזית המתאימה לה.

#row	valid	dirty	ref	Tag	Physical page #
0	1	0	1	0xFB	0xF
1	1	1	1	0x1F2	0xF0
2	1	0	0	0x7C6	0xFFFF
...31	1	0	0	0x5	0xF000

- ראשית נחלק את הכתובת לשדות tag, index, offset. גודל tag, index, offset של כל דף הוא $2^{12}B = 2^{12} \cdot 2^{10}B = 2^2 \cdot 2^{10}B = 4KB$ ולכן שדה ה-index מוצג על ידי 12 ביטים. מספר הקבוצות ב-cache הוא 32 – 12 – 8 = 12. נשארנו עם 2^8 . נשים 2^8 ביטים לשדה ה-tag.

0x 0FB | 00 ABC
tag index offset

ניגש לשורה המתאימה באמצעות ה-index ונשווה בין ה-tag-im. השורה היא 0, מצאנו את ה-0xFB tag, המתרגם לכתובת הפיזית F. נוסיף את ה-offset ונקבל 0xFABC.

1 bit	1 bit	2 bit	12 bit
M	V	Unused	PFN

תרגיל 3: במחשב מסוים יש זיכרון וירטואלי של 2^{32} כתובות. גודל דף הוא $2^{13}B$. מבנה PTE ב-PT הוא:

(1) מה הגודל המקסימלי של הזיכרון הפיזי? גודל שדה ה-PFN הוא 12 ביטים, ולכן גודל ה-PT הוא $2^{12} \cdot 2^{13} = 2^{25}$.

(2) מה גודל ה-PT? ישנים $2^{19} = \frac{2^{32}}{2^{13}}$ דפים וירטואליים ולכן אותה כמות בinskyot ב-PT. למציאת גודל PT בודד נסכום את השדות ונקבל 16 ביטים לפחות 2 בטים. סה"כ נקבל $2^{20} = 2^{19} \cdot 2$.

3 – שיפור ביצועים

Single Thread

שיפור ביצועים

ראינו כי אנו מחשבים את זמן הריצה באופן הבא:

◆ Goal: minimize CPU Time

$$\text{CPU Time} = \text{clock cycle} \times \text{CPI} \times \text{IC}$$

Average Cycles Per Instruction

Instruction Count

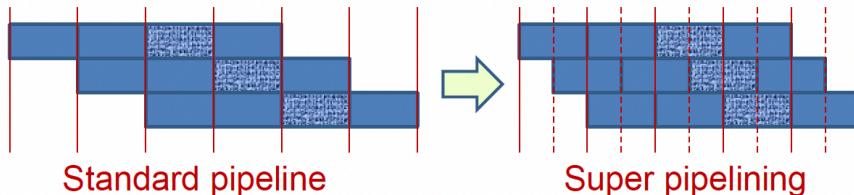
◆ Examples:

- Minimize **clock cycle** \Rightarrow add more pipe stages
- Minimize **CPI** \Rightarrow use pipeline
- Minimize **IC** \Rightarrow architecture

נרצה ליעיל ולצמצם את ה-CPU Time.

:(clock cycle Superpipelining)

אפשר לקחת את ה-pipeline שאנו מכירים, ולחולק ליותר שלבים. זה יוריד את זמן המחוור: הוא נקבע לפי המסלול הארוך ביותר M-LFF לאורך כל המעבד. אם בכל stage עושים חצי מהדברים, אז ה-cycle time קטן בחצי.



הו לנו חמישה שלבים pipeline, ובכשו אנו מחלקים כל שלב ל-2 שלבים, נדחוף באמצעות של כל שלב עוד FF. אפשר לבצע את זה. יש לנו פי 2 cycles, אבל בשહכול מואון ועובד בכל cycle פוקודה 1 נגמרה: ה-IPC הוא עדין 1, וה-cycle time קטן פי 2. חישוב יותר עדין נמצא במצגת, ה-IPC גם נפגע קצת, לא נשאר לבדוק אותו דבר.

:(CPI Superscalar)

Ex: Superscalar (2-issue)

Single-issue	7 cycles/iter
Superscalar (2-issue)	5 cycles/iter

```
loop:
    lw $8, 0($5)      ; $8 - A[i]
    addi $5, $5, 4     ; increment pointer for A[i]
    addi $7, $7, -1    ; decrement loop count
    add $10, $10, $8   ; $10 - $10 + A[i]
    bne $7, $0, loop    ; Continue if loop count != 0
```

יש לנו כמה מסלולים בתוך ה-pipeline, אפשר להריץ כמה פוקודות בו זמנית, כל עוד אין תלויות ביןיהן (הגדלת כל הריבבים במעבד כך שככל אחד יוכל לעבוד יותר מפוקודה אחת). למשל, נוכל לעשות fetch לשתי

פקודות בו זמנית באותו cycle, וזה IPC הוא 2 **אם הכל עובד חילך**.

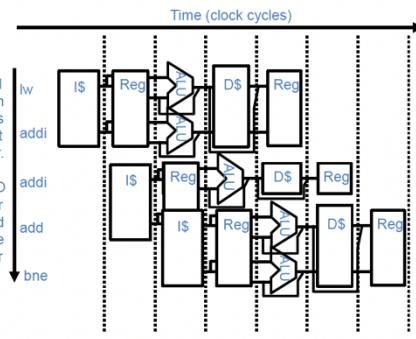
◆ Dependencies based pairing

- bubble (stall) due to the lw (1st instr) delays the execution of the 4th instr
- Dependency detected at decode stage

◆ 2 cycles branch stall

◆ 5 cycles per iteration

◆ CPI (of loop): 5/5 = 1

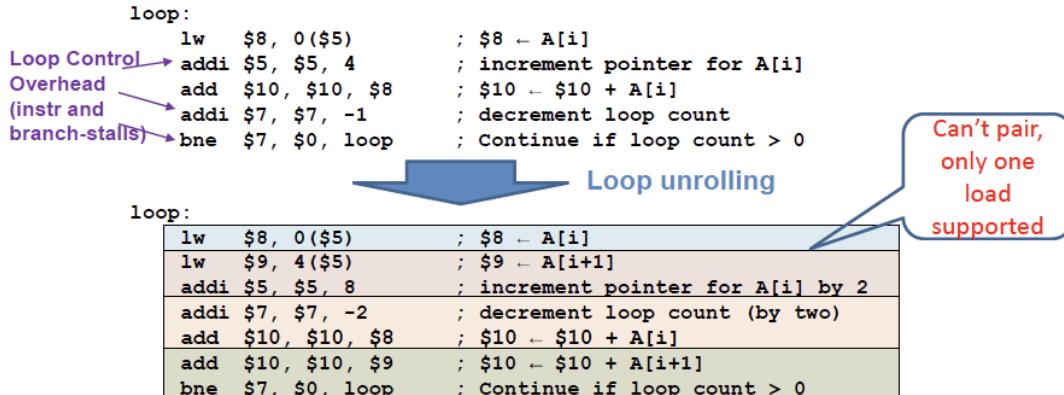


למשל, ב-**2-issue** (שיכי מסלולים), נוכל לבצע את שתי הפקודות הראשונות ביחד. לאחר מכן, נוכל להכניס רק את ה-add-bubble (ביוון שה-add ציריך את \$8 מהסת הראISON של הפקודות שביצעון (פקודת wa)). לכן נבצע את add בforward, ואז את add-i-and-bne ביחד (כasher נבצע forward כפי שכבר ראיינו).

השיפור הוא לא ממש פי 2 ... האם אפשר לעוזר לתהיליך זהה? אין אפשרות לבצע שתי הפקודות בו זמנית ולמצות את הפוטנציאלי של superscalar יותר? למשל באמצעות הקומפיאילר.

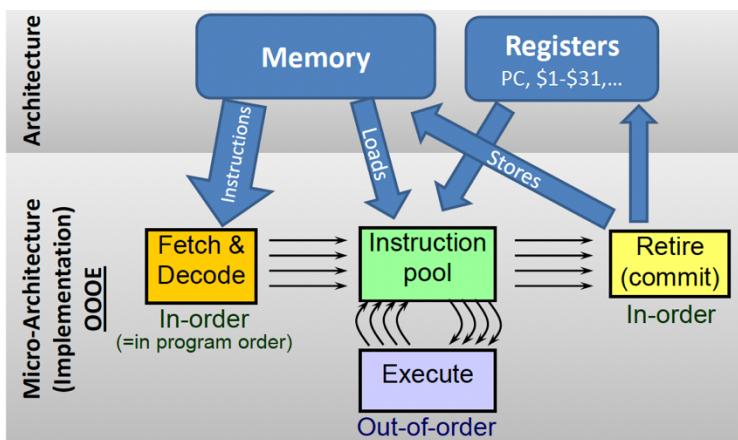


- הkompiiler יכול ללחוץ שהולך להיכנס stall (כיו יש פקודה branch/hazard) ולקחת פקודה נוספת יותר שלא מפרעה, ובמוקם לשימ stall נבצע את הפקודה ההיא. הבעה היא שלא תמיד יודעים متى הולך להיות stall.
- תיקן נוסף הוא **loop unrolling**: השיליטה ב-loop עצמה (הבקורה – עדכן ה-counter, ובדיקה התנאי) הוא overhead overhead (תקורה) שאנו מושלים על לולאה חסית קעינה בתוכן עצמו. על כל תא שאנו מוציאים מהמערך אנחנו מוצאים 2 פקודות הקשורות רק ל-*for* עצמו, "אדמיניסטרציה" נטו. אחרי unrolling, נשלף שני איברים מהמערך באותו מקום אחד, נקדם את ה-pointer ב-2 בתובות מקום אחד, והרখנו את addi מ-2 ל-4. אפשר להרחיב את ה-unrolling מ-2 ל-4.
- אמנם, לא יוכל להרחיב לנצח: העלות היא הוספת הרבה רגיסטרים.

:VLIW

הkompiiler מאנדר N פקודות יחד והופך אותן לפקודה אחת ארוכה, זו גישת ה-VLIW: Very Long Instruction Word. בכל הכתובת פקודות צדו אין תלויות בין הפקודות.

- היחסון – כדי לדעת אילו פקודות יכולות לרוץ עם אילו פקודות אחרות, צריך להכיר את האנטומיה של המחשב, אם נרצה להריץ על מחשב אחר אי אפשר. במקרים מסוימים (כמו router) נראה kompiiler מאנדר ספציפי עם VLIW. אם נרצה router אחר גם החומרה תהיה אחרת.
- זה יכול לעבוד רק מעל חומרה שהיא superscalar.

000E

היעון: נרצה שיהיה לנו זook של פקודות בצורה תור. מהתור זה נוכל לשולף פקודות להריצה. את ה-decode וה-fetch->in, ועושים לפיו הסדר אחד-אחד, order-order, ואז את ה-out-of-order, order-order, מה שאפשר לנו נרץ. התהילה זהה צריך להיות שקוף לחולון למשתמש. עליין לדאוג שהקירה והכתיבה של הפקודות יהיו לפי הסדר, וכן לצמצם את במותה stall נרץ בסדר אחר.

- fetch & decode** – במקביל (תלוי במה אנחנו בסדר המקורי, ממלאים את ה-pool instruction).
- execute** – בודקים אם פקודה מוכנה ליביצוע? במובן זה שאי לה תלויות יותר: כל ה-sources שלה מעודכנים.
- write back** – ה-commit של סיום הפקודה (write back) ליחורן/ל긱יסטים) מתבצע במקביל אבל בסדר המקורי.

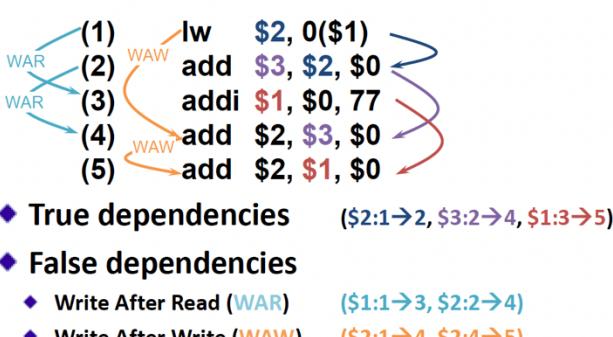
בעיה – data dependency

כאשר יש לנו רצף של פקודות, יש dependencies שונים:

- False dependencies – בעיות של WAR ו-WAW שMOVEDOUT ברגע ריצת OOO לא היו שם קודם.

- True dependencies – בעיות שנובעות מהרצף של הפקודות עצמן – data hazards.

איך נוכל להתמודד עם ה-False dependencies? באמצעות registers naming



פתרון – registers renaming

יש לנו את הרגיסטרים **הארכיטקטוניים** (שהתוכנה ממש רואה, אלו המוכרים לנו). בנוסף, יש לנו יותר ענק של וגייסטרים שנקרוים הרגיסטרים **פיזיים**. התוכנה לא יודעת מהם קיימים, אבל כל החישובים בעצם מותבצעים מעליהם. ככל פקודה תוכל להקצות רגיסטרים חדשים, ורק בשלהול נגמר (בשלב-commit) מ קופלים את הכל ומעודכנים את הרגיסטרים המקוריים.

Register Renaming

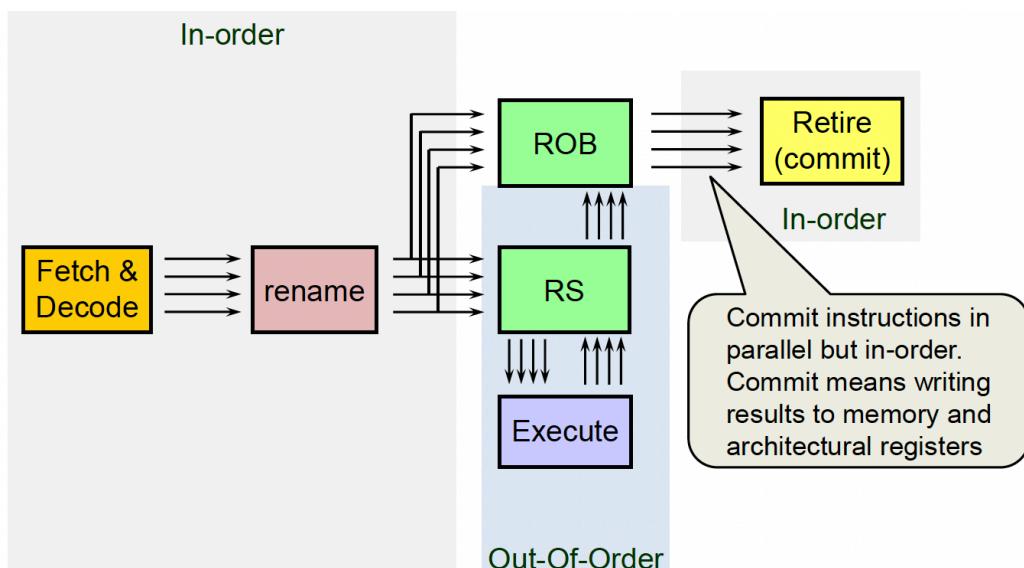
before	mapping	after
(1) <code>Iw \$2, 0(\$1)</code>	<code>[\$2→p1]</code>	<code>Iw p1, 0(\$1)</code>
(2) <code>add \$3, \$2, \$0</code>	<code>[\$3→p2]</code>	<code>add p2, p1, \$0</code>
(3) <code>addi \$1, \$0, 77</code>	<code>[\$1→p3]</code>	<code>addi p3, \$0, 77</code>
(4) <code>add \$2, \$3, \$0</code>	<code>[\$2→p4]</code>	<code>add p4, p2, \$0</code>
(5) <code>add \$2, \$1, \$2</code>	<code>[\$2→p5]</code>	<code>add p5, p3, p4</code>

אנו יכולים לבצעו להריץ `out-of-order` וללא `WAW` או `WAR`. בעת שהנגיש לשלב execute, **כל פקודה שהאופrndים שלה מוכנים – יכולה לחוץ**. כלומר, פתרנו את ה-`False dependencies` אבל נשארנו עם ה-

.True

איך זה עובד ברמת החומרה? אילו רכיבים משתתפים בתהליך?

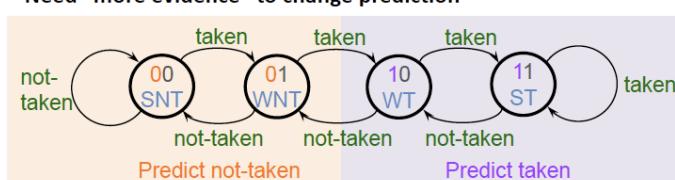
- מביצעים fetch and decode במקביל לפי הסדר, ואת **renaming** של הרגיסטרים.
- ה-RS וה-ROB מנהלים את ה-`RS.execution` (FIFO) שבו כל ה-`rs-renamed instructions` נוכנסו אליו ומחכות להבצע. ROB: כל הפקודות שאחרי שהותבצעו, מחכות לביצוע `commit`.
- ה-`execute` מותבצע שוב במקביל לפי הסדר, כאשר **ה悬念ות האמיתיות לרוגיסטרים הארכיטקטוניים**.

branch prediction

כל מה שדיברנו עליו עד עכשיו מニアית. מה קורה במקרה ריצה לנארית. מה קורה במקרה של branch prediction? נבצע `branch prediction`. יש לנו ייחידה בשם BPU שתפקידיה לחזות האם הולך קרות או לא, וזה בא עם cache משלה.

- בהינתן פקודת `branch`, הוא מיד מוחזר לנו האם אנחנו הולכים לקפוץ או לא ולאיזה בתובת. במדיניות ה-`CPI` פשוטה, עבר בתובת יש ביט אחד שאומר האם פעם אחרונה קפכנו או לא.

- אם טעינו ב-`branch prediction` צריך לזרוק את כל ה-`pipeline` ולעשות `flush`. ביען שאט ה-`commit` אנחנו עושים flush.
- Need “more evidence” to change prediction



- Initial state: weakly-taken (most branches are taken)



אופטימיזציה ושיפור ביצועים (תרגול 12)

[אופטימיזציה קומפайлרים:](#)

בקוד הזה יש RAW בין פקודה 1 ל-2. אף פקודה 5 לא תלולה באף אחד מifikasiות שלפנייה (חוץ מבפקודה 1) אך ניתן להזיז אותה כדי ליצור רווח בין פקודה 1 ל-2 (או נציגך רק stall אחד).

```
add $s0, $s1, $s2
add $s3, $s0, $t0
add $t0, $t1, $t2
add $s3, $t3, $t4
add $s4, $s1, $s2
```

WAR (Write after read)
WAW (Write after write)

ניתן במקום בו אמור להיות stall להכניס פקודה בלבד מהקדם שלא תלולה בגין היסכון ולא משפיע על הפוקודות האחרות. במילים אחרות, ניתן "לרווח" בין פקודות שתלוויות אחת בשנייה ע"י הקדמת פקודה שאינה תלולה בפקודות שלפנייה.

בסוף איטרציה של LOLAH conditional branch, רק אחרי תוצאת ההשוואה נדע האם קופצים או לא. ניתן להקטין את מספר ההפוקודות על ידי unrolling loop. הרעיון הוא ברגע הlolah בכל איטרציה לבצע יותר עבודה (לא רק על ? אלא גם על האיברים הבאים). חסרונות: יותר גיסטים, יותר instructions ברגע הlolah, יש מקרי קצה (אם ב모ת האיטרציות לא מתחלקת ב-2, צריך לטפל במקרה נבנה נבנה...).

דוגמאות:

• קוד מקורי:

```
for (int i=1000; i>0; i--)
    A[i] = A[i] + s;
```

האיבר האחרון:	\$s0 ← &A[1000]
הערך s:	\$s1 ← s
בסיס המערך:	\$s2 ← &A[0]

במה stalls יש בתחום (אין forwarding)? יש תלות בין lw ל-add (בגלל \$t0=\$s0, בין add ל-sw (בגלל \$t4=\$s1) ובין addi ל-bne (בגלל \$s0=\$s1). יש 6 סה"ב + 1 בסוף אחרי ה-bne. סה"ב 12 מחזורי שעון.

אפשר לשנות את סדר הפוקודות, את i ללחוף בין lw ל-add. עבשו נציגך רק stall אחד לפני add, ורק אחד לפני sw. 2 עיכובים לכל איטרציה, 7 מחזורי שעון. נשים לב שרך 3 מחזורי שעון מתוך ה-7 באמת מהווים את זמן העבודה של האיטרציה (התוכן עצמו). שאר מחזורי השעון הם פקודות בקרה בלבד של lolah.

נפריד בין **עבודה אמיתיית לטיפול באיטרציה**. נמיצט כמה שיותר בפקודות המטפלות באיטרציה. אין תלות בין איטרציות ולכן ניתן לפשטם באמצעות unrolling loop. נפרוש 4 איטרציות. בוצע את הפוקודות של העבודה האמיתית 4 פעמים, ולבסוף נעדכן את lolah תוך קידום ב-16 ולא ב-4.

אחרי כל swo יש 2 עיכובים, גם אחרי add 2 עיכובים: ככלומר 4 עיכובים לכל רצף פקודות אמיתיות, 16 לכל האיטרציה. בנוסף עוד 3 עיכובים לפקודות הבקרה. נסיף את 14 הפקודות ונקבל 33 מחזורי שעון ל-4 איטרציות, בכלומר 8.25 מחזורי שעון לאיטרציה.

שיפור נסיף – שינוי סדר הפוקודות. נקבע את כל ה-swo, את כל ה-add, ואת כל ה-sw (נשתמש בעוד גיסטים). עבשו ביטלנו את כל העיכובים של הפוקודות האמיתיות! רק 3 של הבקרה נשארו. סה"ב 14 פקודות ועוד 3 עיכובים = 17 מחזורי שעון ל-4 איטרציות. **3.25 מחזורי שעון לאיטרציה**.

[שיפור ביצועי המעבד:](#)

נרצה להקטין את זמן ה-CPU, נזכיר כי אנחנו מחשבים אותו באופן הבא:

$$cpu_{time} = IC \cdot CPI \cdot clock\ cycle$$

ב-SC כל פקודה רצה במחזור שעון אחד אורך. CPI = 1. ב-MC כל פקודה רצה בכמה מחזורי שעון קרירים יותר (מספר מחזורי השעון שהוא עברו כל פקודה), קיבלנו $CPI > 1$ אך מחזור שעון שהתקצר משמעותית. ב-SC קיבלנו $CPI = 1$ ומחזור שעון קצר. אילו עוד שיפורים ניתן לבצע?

:Super pipelining

על מעבד pipeline עם חמישה שלבים בעל מחזור שעון של 2ns :

$$\text{CPU}_{\text{pipeline}} =$$

$$\text{IC} \times \text{CPI} \times \text{clock cycle} = 1,000,000 \times 1 \times 2\text{ns} = 2\text{ms}$$

על מעבד super-pipeline עם שמונה שלבים בעל מחזור שעון של 1.3ns :

$$\text{CPU}_{\text{super-pipeline}} =$$

$$\text{IC} \times \text{CPI} \times \text{clock cycle} = 1,000,000 \times 1 \times 1.3\text{ns} = 1.3\text{ms}$$

ובש"כ נקבל:

$$\text{Speedup} = \frac{\text{CPU}_{\text{pipeline}}}{\text{CPU}_{\text{super-pipeline}}} = \frac{2}{1.3} = 1.53$$

חלוקת של שלבי pipeline לשלבים קטנים יותר: במצב אידיאלי עדין $\text{CPI} = 1$, אבל מוחור השעון מתפרק עוד. חסרונות: חלוקה לשלבים יותר קטנים לא פשוטה מבחינה החומרה, מכפיל את כמות הרגיסטרים, מספר הפקודות שmpsפפסים בחיזוי branch יותר גדול.

דוגמה: נניח כי בתכנית ישן מיליון פקודות. התכנית מבוצעת פעמיים: (1) pipeline, חמישה שלבים, = 2ns (2) superpipeline, שמונה שלבים, = 1.3ns . על איזה מעבד התכנית מבוצעת מהר יותר (בנחתה שאין סיכון)?

נתון לנו מספר שלבים, ואנחנו לא מתייחסים לנathan זהה – הוא לא משנה שום דבר במקרה זהה.

:Superscalar pipeline

מבצע יותר מפקודה אחת במחזור שעון, בצע במאה פקודות יחד (מס' pipelines במקביל).

אפשר להגיע ל-CPI של פחות מ-1. תיאורית
ניתן להגיע ל-CPI של 0.5, במקדים שבהם אין תלות בין פקודות ואז אפשר להכניס את שתיהן במקביל.

תרגיל 7

Clock	IF	ID	EX	MEM / ALUWB	WB	Pipeline	Remarks
1	1 2					U pipe V pipe	
2	3 4	1 2				U pipe V pipe	
3	5 6	3 4	1 2			U pipe V pipe	Issue 1,2
4	6 -	4 5	3 -	1 2		U pipe V pipe	Issue 3 r1:1⇒3
5	- -	5 6	4 -	3 2	1	U pipe V pipe	Issue 4 r1:1⇒4
6	- -	5 6	- -	4 -	3	U pipe V pipe	Stall issuing
7	- -	6 -	5 -	- -	4	U pipe V pipe	Issue 5 r7:4⇒5
8	- -	- -	6 -	5 -	-	U pipe V pipe	Issue 6 r7:5⇒6
9	- -	- -	- -	6 -	5	U pipe V pipe	
10	- -	- -	- -	- -	6	U pipe V pipe	

סה"כ קיבלנו 10 מחזורי שעון

:000E

ביצוע הפקודות במעבד הוא לא **לפי סדר הפקודות בקוד – אלא לפי זמינות**. הריצה זו במקביל יוצרת בעיות חדשות שלא היו קודם בנוסף ל-RW המוכר (לא קיימות ב-pipeline כי ההריצה היא **לפי הסדר**):

- WAR – אנחנו קוראים מ-6R בפקודה 2, וכותבים אליו בפקודה 3. אמנם, פקודה 2 תליה בפקודה 1 ולכן היא ממחכה. נריץ את פקודות 1 ו-3 במקביל, אז פקודה 3 תנסה את הערך של R6 שפקודה 2 קוראת.
- WAW – אנחנו כותבים ל-5R בפקודה 2, וגם בפקודה 4. אם פקודה 4 תרוץ לפני 2 אז הערך האחרון של R5 יהיה לפי פקודה 2 וזה לא מה שאנחנו רצימים.

DIV R1,R2,R3
ADD R5,R6,R1
ADD R9,R5,R5
ADD R5,R7,R8

DIV R1,R2,R3
ADD R5,R6,R1
ADD R6,R7,R8

כדי למנוע את הבעיה החדשות (false dependencies) ניעזר ב-renaming (false dependencies) – לשמור שתי מערכות רגיסטרים, הארכיטקטוניים שගלויים למשתמש, ורגיסטרים פיזיים שהוא לא רואה.

- המיפוי הראשון: מהארכיטקטוניים לפיזיים מתבצע בין שלב decode לשלב execute .commit
- המיפוי השני: מהфизיים לארכיטקטוניים מבוצע בחלק מפעולת-h-commit

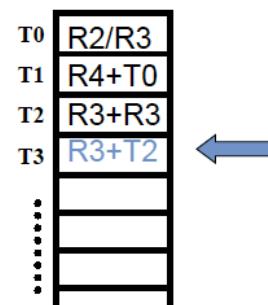


- ROB: מקבל את ההוראות משלב decode על פי הסדר, הבנינה המתאימה ב-ROB מראה את מספר הרגיסטר הנוכחי. בשלב execute נכתבת התוצאה לבנייה המתאימה ב-ROB וההוראות מבצעות את שלב commit על פי הסדר שלו ב-ROB. הוראה יכולה לבצע commit רק אם זו שלפניה עשתה זאת.

RAT	
R1	T0
R2	T2
R3	
R4	T3

הנקודה החשובה:
Commit ו Fetch Decode מתבצעים לפי הסדר לנכון. המיפוי תמיד יהיה תקין. וגם הכתיבה בפועל לרוגיסטרים תהיה בסדר הנכון.

עוד נשים לי כי כתע גם אם פקודה 2 כתוב לרוגיסטר R2 היא למעשה כתוב ל- T1 אבל מה שייכתב בפועל ל-R2 יהיה מה שנמצא ברגע ב-T2.
כניל אל אם פקודה 4 מותבצעת לפני פקודה 2 או R4 כתוב אונומס, אבל הוא נכתב ל-T3, ואילו פקודה 2 קוראת את T0 ו עוד R4 (ולכן קוראת את הערך המקורי).



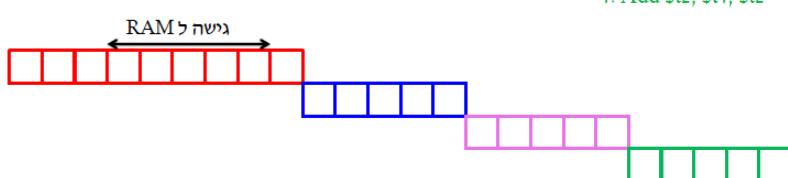
שימו לב!
כעת אין תלויות מסוג False Dependencies
אבל סיכון מסוג RAW עדין קיימים.

שיפור ביצועים (תרגול 13)

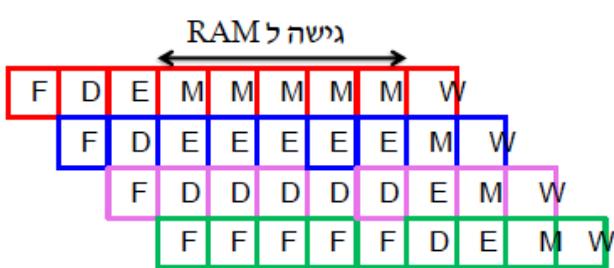
המחשה ויזואלית:

נתון הקוד הבא, כאשר יש cache miss בגישה ל זיכרון :

1. Lw \$t0, 4(\$t1)
2. Add \$t2, \$t3, \$t4
3. Add \$t5, \$t6, \$t2
4. Add \$t2, \$t4, \$t2



: אם יש cache miss זה אומר שאנו כבר בדקנו את ה-cache, זה אומר שאנו צריכים לזכור לקחת(cache) את הזמן של גישה ל-cache.

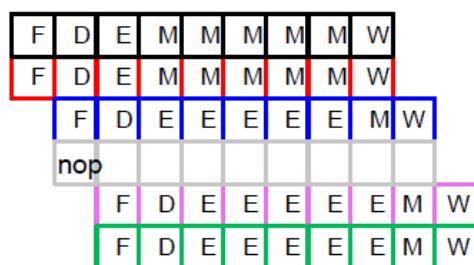


- Pipeline: שלב ה-memoryлокוח 5 מחזורי שעון (שרירותית) בגלל ה-cache miss. לכן גם שלב ה-execute של הפקודה הבאה מתעכב, כי הרכיב של ה-memory עדין לא סיים לבצע את הפקודה הראשונה.

:Superscalar 2-way

אפשרות 1: זמן ביצוע פקודה במצב רגיל לוקח 5 שלבים. כחול תקוע ב-M כי אדום תקוע ב-M והפקודות צריכות להתקדם במקביל. סגול יורך תקעים בשלב ה-E לאחר ה-M תפוס.

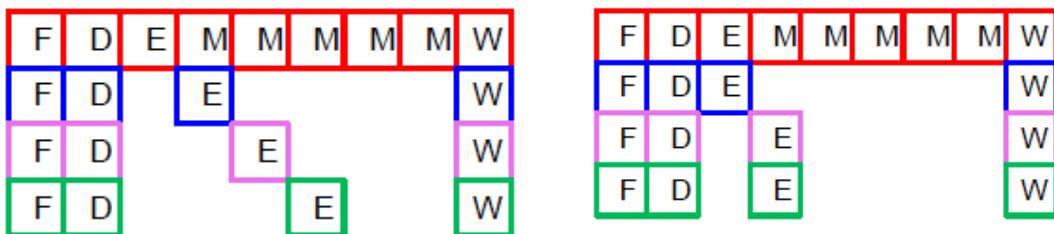
אפשרות 2: ה-W הוזמד לפקודה קודמת (בצבע שחור), פקודות 2 ו-3 לא יכולות להתבצע במקביל בגלל תלות, RAW בغال הרגיסטר. לכן ה-slot השני ליד הכתול נותר ריק.





:OOOE

F ו-D מתרחשים במקביל אבל order in. כל פקודה בתורה עשויה E, האודם מבללה זמן מה ב-M. כתיבת התוצאות לריגיסטרים במקביל אבל order.in. בנסיבות הפקודות הזאת, גם superscalar 2-way לא חוסך למ, כי האודם תקוע ב-M. אם היינו מוסיפים עד פקודות, יכול להיות שאז 2-way היה יעיל יותר מבחינת זמן ריצה, פחות מוחזרי שעון.



VLIW

מעבד שמאפשר הריצה של מספר פקודות בלתי תלויות במקביל (קובצת פקודות במקביל) אין מבנה של pipeline, ככלומר מס' מים לבצע אחד ועוביים לבא אחריו. למעבד זה מספר slots למטרות שונות, ובכל batch מלאו אותם בפקודות מתאימות (יתכן ויישארו slots ריקים).

תרגיל: נתונה ארכיטקטורת WLIW (ممומש ב-pipeline) שבו קיבוץ הפקודות הינו:

- 1 פעולות גישה ליברון.
- 2 פעולות אריתמטיות.
- 1 פעולה קפיצה (פעולות קפיצה תמיד יתבצעו, קפיצות שנלקחות גורמות ל-stall של 2 מוחזרי שעון).

נחלק את הקוד הבא ל-slots instructions WLIW כדי לצמצם את זמן הריצה. צריך להראות קיבוץ כזה בצורה אופטימלית: נשים לב שהפקודה **jal** היא פקודת קפיצה לכל דבר.

```

1. tree_scan:      # start of the function: $a0=a, $a1=v
2.      beq    $a0, $0, end_null          # if (a==NULL)
3.      addi   $sp, $sp, -12
4.      sw     $ra, -12($sp) # the offset was changed to support grouping
5.      sw     $s0, 4($sp)
7.      lw     $t0, 0($a0) # instruction order changed to improve grouping
6.      sw     $s1, 8($sp)
8.      addi   $s0, $0, 0 # $s0 is count, count = 0
9.      bne   $t0, $a1, not_equal # if (a->value != v)
10.     addi   $s0, $s0, 1 # count++
11. not_equal:
12.     lw     $s1, 4($a0) # $s1 = a->right
13.     lw     $a0, 8($a0) # $a0 = a->left
14.     jal   tree_scan # call tree_scan(a->left, v)
15.     add   $s0, $s0, $v0# count += tree_scan(a->left, v)
16.     add   $a0, $s1, $0
17.     jal   tree_scan # call tree_scan(a->right, v)
18.     add   $v0, $s0, $v0 # setup return value: count += tree_scan(a->right, v)
19.     lw     $ra, 0($sp)
20.     sw     $s0, 4($sp)
21.     lw     $s1, 8($sp)
22.     addi   $sp, $sp, 12
23. end_null:
24.     jr     $ra
           addi   $v0, $0, 0
           jr     $ra

```

החלפנו את 6 ו-7: אם היה 6 ואז 7, אך לא היינו יכולים לשימוש בא-**beq** שמשתמש ב-0-\$t0 ותלו依 בפקודה 7, יחד עם פקודה 7 באותו .batch



שאלות מסכימות

Execution Analysis for In-Order and OOOE

Program:

```

\$a0 = pointer to int array
\$t2 = 16 (ints in array)

add \$t0, \$0, \$0
LOOP:
    lw \$t1, 0($a0)
    addi \$t2, \$t2, -1
    addi \$a0, \$a0, 4
    add \$t0, \$t0, \$t1
    bne \$t2, \$0, LOOP

```

שאלה 1: יש לנו תוכנה קטנה נתונה, אנחנו רוצים לנתח אותה בכל מיני וריאציות של ארכיטקטורות:

.1. **In-Order Pipeline**: עם כל ה-branch stall, bypasses 2 cycles

.2. **In-Order 3-wide Superscalar**: עם כל ה-branch stall, bypasses 2 cycles

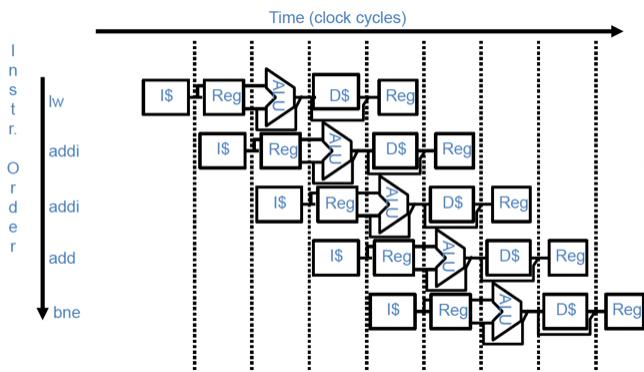
.3. **OOOE 5-wide**: 5 פקודות במקביל במחזור שעון, 256 רשומות ROB, RS, RS, RS, RS, CPI=0.25

נתון לנו כי:

- מבנה ה-cache: כל בלוק בגודל 64B, עונש על miss הוא .20 cycles
- hit – תמיד מבצע .hit

פתרון 1:

ראשית נשים לב מה קטע הקוד מבוצע: יש כאן לפחות 16 איטרציות העוברת על מערך, וצוברת את סכום האיברים שלו. ה- \$t0 הוא המשתנה הצובר שלנו והוא .\$t0. ננתח את הקוד:

1. Pipeline:


נשים לב שביןlw ל-addi אין תלוות. המידע ב-w\$
מוקן רק אחרי שלב h-mem, אז אם ב-addi היו
משתמשים בmidع שמוסכאים ב-w\$ היה חייב להיות
פה stall. אין תלות בכך אין צורך.

בכל איטרציה של הלולאה (5 פקודות) ללא miss עליה
סה"ב קיבלנו 7 cycles פר איטרציה.

c. 16 איטרציות של הלולאה (80 פקודות) ללא miss
עלות 112 = 16 · 7, קיבלנו 112 cycles ללולאה.

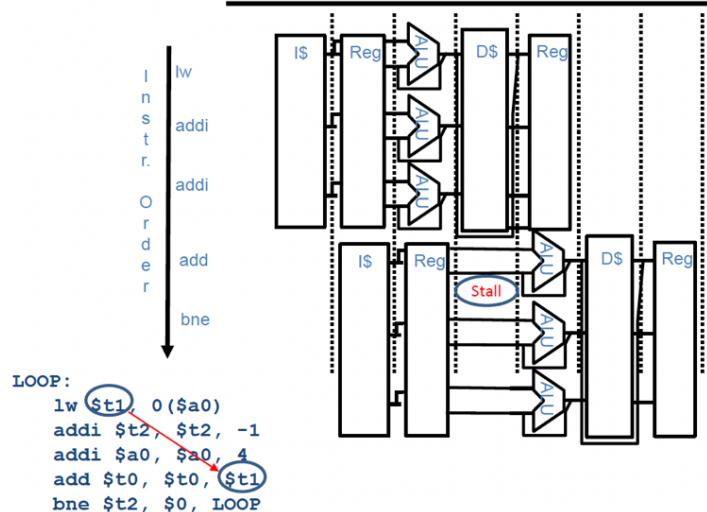
d. מה קורה אם יש miss? כל בלוק הוא 64B, וכל

cycle penalty הוא 20 cycles. אנחנו לא צריכים עוד מידע

על ה-cache (מדיניות החלפה, דרך מימוש direct/associative) כי כל תא נקרא רק פעם אחת, ואיתו ביחס
אנו שולפים נתונים נוספים וביהם משתמשים (локאליות במקומם). כמה נתונים אנחנו שולפים מהזיכרון? 64B
כדי לספרור את כמות המילים נחלק ב-4 ונקבל W16. אנחנו טוענים בדיקת כל המערך לזכרון, لكن מדובר על

cache block אחד, 1 miss cache block (לכל 80 הפקודות).

$$CPI = \frac{132}{80} = 1.65$$

2. Superscalar:


a. שלושת הפקודות הראשונות רצות יחד במחזור הראשון, כי אין ביניהן איזושהן dependency. לא
חייבים שהפקודה של lw תיגמר כדי לבצע אתosci-hi.

b. נשים לב כי יש WAR בכתיבת \\$t0, אבל זה לא
In-Order hazard.

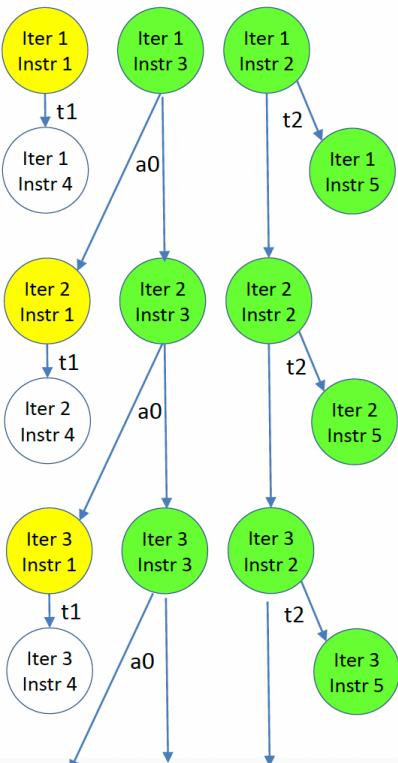
c. במחזור השני, ה-add צריך לקרוא את \\$t1, אז לו
יש מחזור אחר מחזור lw ואז שימוש בו. לכן
נចטרך להכניס stall אחד. לאחר מכן יש כמובן
shov stalls של b-ne.

d. כל איטרציה (5 הפקודות) ללא miss עליה
branch stall 2 cycles ועוד 2 cycles
סה"ב 5 cycles פר איטרציה.

e. 16 איטרציות של הלולאה (80 הפקודות) ללא miss
עלות 80 = 16 · 5, קיבלנו 80 cycles ללולאה.

f. גם כאן יש miss אחד בעלות 20 cycles (לכל 80 הפקודות).

$$CPI = \frac{100}{80} = 1.25$$

.3. הערות על E000:

- a. חיבים לשאול מה גודל הבאר? יש 256 רשומות ב-ROB, ו-128 רשומות ב-RS: אפשר לחכות עם עד 128 פקודות בו-זמןית שעדיין לא יכולות לזרע. סך כל הפוקודות הוא 80 **ואין כאן בעיה מההיבינה זו.**
- b. את פקודה 1 אפשר להריץ, אבל את 4 אי אפשר להריץ עד ש-1 לא מסתויימת. את פקודות 2, 3 אפשר להריץ במקביל, ובש-2 מסיימת אפשרות להריץ את 5.
- c. מה קורה **בזמן שימושים ב-20 cycles של miss?** אפשר לבצע את פקודות 2 ו-3 לכל האיטרציות בו-זמןית עם ה-renaming (יש תלות בין איטרציות, אבל בכל איטרציה אפשר לבצע בו-זמןית את 2 ו-3, ובאייטרציה הבאה את 2,3 ו-5 בו-זמןית וכו'...).
- .n. **בכל מחזור מבצעים 3 פעולות יירוקות:** את פקודה 5 של איטרציה אחת ואת 2 ו-3 של איטרציה שתיים. בכל 16 האיטרציות מדובר ב- $16 \cdot 3 = 48$ פקודות סה"כ. נתנו לנו CPI=0.25 ולכן ב- $48 \cdot 0.25 = 12$ מחזורי שעון, זה פחות מ-20 מחזורי miss! לעומת זאת, אפשר לבצע את כל הפוקודות הירוקות **"בצל miss" (in the shadow).**
- ii. אחרישה miss נגמר: נשארו עם 32 פקודות צהובות (שתי פקודות add, wo) ב-16 איטרציות. נשים לב כי יש תלות בין פקודות 4 בין איטרציות. לכן יש לנו עוד 16 מחזורי שעון שבהם מרים זוגות של פקודות.
- d. סה"כ יש לנו 20 cycles miss ועוד 16 cycles לאחר miss: $CPI = \frac{36}{80} = 0.45$. קיבלנו **36 cycles לכל התוכנה**, נקבל $20 + 16 = 36$

הערות על E000:

- מתעלמים מכמות הזמן שהוקח לעשות fetch-retire או לפקודות לתוך ה-ROB? אידיאלית,
- $\frac{80}{5} = 16$ מחזורי שעון, או לפחות 20 וזה עדין יצא **פחות מ-36 מחזורים, لكن זה לא מגביל אותנו.**
- למה 1 ו-3 לא מהווים hazard של WAR? זה התפקיד של ה-renaming, והוא פוטר את זה (false dependency).
- branch prediction ב-E-BE זה מורכב (ולא דיברנו לעומק על איך מטפלים בה): אפשר פשוט לקחת את המידע הנוכחי בשאלת השה-CPI (זה מגולם פנימה בפנים, יחד עם שיקולים אחרים). לעומת זאת הארכיטקטורה מביניהם כי במצב אידיאלי נריץ 5 פקודות כל מחזור, לעומת זאת השה-CPI=0.2. עברו הקוד הספציפי, מגלמים את השה-CPI ואומרים שאפשר בפועל להריץ 4 פקודות כל מחזור, השה-CPI=0.25.

Coding, Stalls and Loop unrolling

שאלה 2: נרצה לחשב X איברים בסדרת פיבונacci: סדרה שבה כל איבר שווה לסכום שני קודמייו. שני האיברים הראשונים בסדרה הם 1, 1. יש לענות על הפעולות הבאים:

```
int FibSeq[47]; // pre allocated by the program (passed as parameter)

// function call example
CalcFibonacciSequence(FibSeq, 47);
...
```

```
void CalcFibonacciSequence(int *Array, int NumOfElements)
{
    int i;
    Array[0] = 1;
    Array[1] = 1;

    for(i=2, i < NumOfElements; i++)
        Array[i] = Array[i-1] + Array[i-2];
}
```

1. נתון קוד ב-C, צריך לתרגם אותו ל-MIPS.
2. לנתח את stalls-forwarding (בלוי stalls) (forwarding עםLoop unrolling)
3. לבצע stalls-forwarding (forwarding עםLoop unrolling)

פתרון 2:

1. נתון לנו הקוד הבא ב-C. נרצה לתרגם לאסמבלי את הפונקציה CalcFibonacciSequence תוך שמירה על calling convention: מעבירים \$a, \$t ו-\$sp על פריגיסטרים. ניתן להשתמש ברגיסטרים \$s, \$r, \$t והמצביע למחסנית הוא \$ra. וכותבת החדרה היא \$ra.



```

1. CalcFibonacciSequence:
2.         addi    $t0, $0, 1
3.         sw     $t0, 0($a0)
4.         addi    $t1, $0, 1
5.         sw     $t1, 4($a0)
6.         addi    $a0, $a0, 8
7.         addi    $t2, $0, 2
8. loop:   add    $t4, $t0, $t1
9.         add    $t0, $t1, $0
10.        add   $t1, $t4, $0
11.        sw    $t4, 0($a0)
12.        addi   $t2, $t2, 1
13.        addi   $a0, $a0, 4
14.        bne   $t2, $a1, loop
15.        jr    $ra

```

- a. תחילת נגידר את הפרמטרים: Array Num מועבר ב-1,\$a0-\$t4, \$t0 מועבר ב-1,i.
- b. תחילת ניעזר ב-\$t0 עבור המספר הראשון (באינדקס 2 - i), וב-\$t1 עבור המספר השני (באינדקס 1 - i), הם שומרים את שני הערבים האחרונים בסדרה. נבצע sw לשני הערכים הללו ב-offsetüber האיבר הראשון (0) ועבור האיבר השני (4) במערך. נקדם את המצביע למערך \$a0 ב-8 (כדי להגיע לאיבר השלישי), וניעזר ב-\$t2 בתור ה-2 שלמו בלולאה.

- c. בעת בלולאה: \$t4 יהיה סכום שני האיברים האחרונים בסדרה. נבצע את העדכון של שני האיברים האחרונים, נשמר במערך \$t4, נקדם שוב את \$a0 ב-4 (להציגו לאיבר הבא), ונבדוק באמצעות bne האם הגענו לכמה האיברים שניתנה ב-1.\$a1. אם לא, נקופץ בחזרה ל-loop וממשר את הלולאה.

2. נרצה לחשב את stalls-pipeline:

- a. בבחור ה-registers file-no forwarding except writeback – ניתן register file אפשר לבתוב/לקראא מאותה בתובת בו-זמןית, ולבצע forward לשלבים אחרים, כל שאר ה-forwardings לא קיימים! אם למשל בוצע writeback ונצטרך את התוצאה מה-ALU, לא יוכל להזין אותה לפקודה הבאה, נצטרך לבצע 2 stalls.

- b. לפני פקודה 3 יש לנו RAW ולקמן stalls 2, וכך גם לפני פקודה 5 באותו אופן. מפקודה 8 לפני פקודה 10 יש לנו שוב RAW אבל יש לנו את פקודה 9 במאוץ לנו נצטרך רק 1 stall. וכך גם בין פקודה 12 לפני פקודה 14 באותו אופן.

- c. חוץ מזה, יש לנו 2 stalls של ה-branch. seh"c יש לנו 8 stalls בקוד הזה.

3. נוסיף Loop unrolling:

- a. הפעם יש stall, היחיד שעדיין נשאר הוא אחריו אין ציריך ואין פה \$t1. לכן איטרציה בודדת תיקח 7 cycles ועוד 2 branch stalls ב-column 9 מחזורי שיען לאיטרציה.
- b. נבצע 2 פעולות בכל איטרציה, נטפל ב-2 איברים. אנחנו צריכים 45 איטרציות, שהה מסpter איזוגי. בסוף נצטרך לעשות עוד איטרציה אחת בcli unrolling. נוסיף פקודות שמחישות את האיבר הבא, רושמות אותו ל-0 עם offset של 4 (האיבר הבא), ואז נקדם את ה-loop counter שלו ב-2, ונעדכן את המצביע למערך \$a0 ב-8 (להציגו שני איברים קדימה).

```

1. loop:   add    $t4, $t0,$t1      // A[i] = A[i-1] + A[i-2]
2.         add    $t0, $t1, $0      // move A[i-1] to t0
3.         add    $t1, $t4, $0      // move A[i] to t1
4.         sw    $t4, 0($a0)      // store to A[i]
5.         add    $t4, $t0,$t1      // A[i+1] = A[i] + A[i-1]
6.         add    $t0, $t1, $0      // move A[i] to t0
7.         add    $t1, $t4, $0      // move A[i+1] to t1
8.         sw    $t4, 4($a0)      // store to A[i+1]
9.         addi   $t2, $t2, 2      // increment loop counter
10.        addi   $a0, $a0, 8      // point to next Array[i]
11.        bne   $t2, $a1, loop    // branch to loop until i=NumberOfElements
12.        jr    $ra

```

- c. בקוד הנ"ל הייתה הינהה של מספר זוגי. אם נרצה לטפל במספר אי-זוגי: לא נעשה bne אלא \$t2, ונוסיף את הקוד המזכיר בcli unrolling והוא יירוץ איטרציות בודדות בסוף מה שנשאר לו.

- d. יש לנו 11 פקודות בתוספת 2 עבור branch: seh"c 13 מחזורי שיען ל-2 איטרציות. מה קורה אם נעשה X times unrolled by X times control של loop, 3 פקודות של control,濂ן נקבל $2 + 3 + 4X + 3 + 4X$ מחזורי שיען ל-2 איטרציות.

$$\text{濂ן נקבל } 2 + 3 + 4X + 3 + 4X \text{ מחזורי שיען ל-2 איטרציות: נקבל } \frac{5}{X} + 4 \text{ מחזורי שיען לאיטרציה.}$$



```

struct Node {
    int X, Y;
    struct Node *Left, *Right;
};

Node *Find_X_In_SortedTree(int X, struct Node *T)
{
    while(T && (T->X != X)) {
        if (T->X < X)
            T = T->Left;
        else
            T = T->Right;
    }
    return T;
}

```

Find X in sorted BST

שאלה 3: נתונה תוכנית שמחזאת את הערך X בתוך עץ בינארי, ומוחירה את האיבר מהעץ. כל איבר בעץ מכיל שני ערכים X, Y וכי מצביים: ימין לערכי X הגדולים ממנו, ושמאל לערכי X ממנה.

1. תרגום קוד C ל-MIPS.
2. ניתוח hit rate.
3. ניתוח ריצה.
4. רקורסיה.

פתרון 3:

1. תרגם את הקוד לאסמביל:
 - a. בולאה Loop נובן את \$a0 להיות ה-node הנכובי של העץ, ונבדוק האם הערך הוא null (0). אם כן, נסיום.
 - b. אחרת, נגיע ל-XCmp שמשווה בין ערך ה-X של node הנכובי שנבען לערך \$t1 (נמצא ב-0 offset של \$a1). בין ערך ה-X המבוקש שנמצא ב-0.\$a0 לשווים, נסיום.
 - c. אחרת, נקבע ל-XNeq, נרצה לדעת האם המשיר את החיפוש בצד ימין או שמאל. בצעט \$t2 כדי לבדוק האם הערך של node הנכובי קטן מ-X המבוקש. נבצע beq, ואם \$t2 הוא 0 (כלומר הערך גדול או שווה), נקבע ל-XGrt: נתקדם למצביים הימני (ב-12 offset ב-node) נקבע חזרה ל-.Loop.
 - d. אחרת, נתקדם למצביים השמאלי (ב-8 offset ב-node). נקבע חזרה ל-.Loop.

```

// $a0: X, $a1: T
Find_X_In_SortedTree:
Loop: add $v0, $a1, $0
      bne $a1, $0, CmpX
      jr $ra
CmpX: lw $t1, 0($a1)
      bne $t1, $a0, XNeq
      jr $ra
XNeq: slt $t2, $t1, $a0 // $t2 = ($t1 < $a0) ? 1 : 0
      beq $t2, $0, XGrt
      lw $a1, 8($a1)
      beq $0,$0, Loop
XGrt: lw $a1, 12($a1)
      beq $0,$0, Loop

```

```

Node *Find_X_In_SortedTree(int X,
                           struct Node *T)
{
    while(T && (T->X != X)) {
        if (T->X < X)
            T = T->Left;
        else
            T = T->Right;
    }
    return T;
}

```

ניתוח cache:

נתון לנו כי גודל ה-cache הוא 1KB, מסוג direct map, גודל כל בלוק W=4B=16B. בערך יש 2^{20} איברים, עומק ממוצע 21. כל הקוד נמצא ב-cache misses של instruction cache (אפשר להעתיק מ-cache hits) נרצה לחשב את hit rate-cache של ה-data cache?

- a. נשים לב כי גודל ה-node struct הוא בדיק בגודל הבלוק (4 שדות, W).
- b. לכל node שאנו חונכו ניגשים אליו **יהיה compulsory miss** ראשוני, כאשר הגישה השנייה היא כבר hit. לא נחרוזאותו node שב, לכל בלוק ניתן פעמי אחד, אז לא משנה מה היה קודם.
- c. **יש שתי גישות לזכור:** פעמי אחת נשווה את הערך של X, ופעמי שנייה נשלוף מצביע left או right. לכן hit rate היה **50%**.

ניתוח ריצה: יש לנו כמה ארכיטקטורות שונות, ונרצה להשוות את ה-
taken performance שלתן. יש לנו 8 פקודות בכל איטרציה, 3.5 branches taken. ב-branches dependency. **ב-branches stall אחד** בשיל In-Order. branches between lw ל-bne. יש 21 איטרציות (עומק ממוצע 21). לכל branch יש 3 cycles penalty. **In-Order pipeline f** .a $21 \cdot (8 + 1 + 3.5 \cdot 3) = 1410$. $cpu_{time} = 410 \cdot T = \frac{410}{f}$ אז **410** .b **In-Order pipeline 1.5f** . $21 \cdot (8 + 1 + 3.5 \cdot 3) = 273$, אבל זה זהה ל-273 cycles. **f**. החישוב שמניב לנו את זה הוא $\frac{410}{1.5f} = \frac{273}{f}$

c. In-Order superpipeline 2f: הכל הולך לרווח פי 2 יותר מהר חוץ מ-hazards-and-branches. הם מעכבים יותר

(פי 2), אך עכשו הוא 6 stalls (אחרי ה-ALU אנחנו יודעים אם מתבצעת קפיצה או לא, ואז כל מה שהבננו ל-pipeline עד לנקודה הזאת צריך לעוף, עברו 6 מחזרים עד שהגענו לנקודה הזאת), ו-hazard 2

$$. cpus_{time} = \frac{326}{f} \cdot 21. \text{ נחלק ב-2 כדי להשווות stalls}$$

עכשו קיבל 651 stalls = $\frac{8}{4} \cdot (8 + 2 + 3.5 \cdot 6) = 651$. עכשו נקבע 000E 4-wide, no prediction (taken=3 cycles) .d

$$\cdot 21 + 3.5 \cdot 3 = 263 \text{ dependency של wl. נקבע 263}$$

e. 000E 2-wide, 99% accurate prediction: יהו לנו 4 מחזרי שעון בכל איטרציה. יש לנו stall רק

באחוז אחד מהמרקם. נקבע $86 = (4 + 3.5 \cdot 3 \cdot 0.01) \cdot 21$. בчисוב זהה יותר מדוקן לחשב עם 4 stalls בבל .branches, branch

4. נמש גישה רקורסיבית:

a. במקום לעשות while, נקרא רקורסיבית עם המצביע המתאים (שמאל/ימין):

Recursive

```
struct Node {
    int X, Y;
    struct Node *Left, *Right;
};

Node *Find_X_In_SortedTree(int X,
                           struct Node *T)
{
    while(T && (T->X != X)) {
        if (T->X < X)
            T = T->Left;
        else
            T = T->Right;
    }
    return T;
}
```

```
struct Node {
    int X, Y;
    struct Node *Left, *Right;
};

Node *Find_X_In_Tree(int X,
                     struct Node *T)
{
    if (T && (T->X != X)) {
        if (T->X < X)
            T = Find_X_In_Tree(X, T->Left);
        else
            T = Find_X_In_Tree(X, T->Right);
    }
    return T;
}
```

b. בيون שהוספנו קריאה לפונקציה, אנחנו שומרים על המחסנית את \$ra, מבצעים **jal** ובסוף משחזרים את \$ra מהמחסנית ו קופצים אליו (בתובת החזרה המקורי).

```
// $a0: X, $a1: T
Find_X_In_Tree:
    add $v0, $a1, $0
    bne $a1, $0, CmpX
    jr $ra
CmpX: lw $t1, 0($a1)
    bne $t1, $a0, XNeq
    jr $ra
① XNeq: addi $sp, $sp, -4
    sw $ra, 0($sp)
    slt $t2, $t1, $a0 // $t2 = ($t1 < $a0) ? 1 : 0
    beq $t2, $0, XGrt
    lw $a1, 8($a1)
② jal Find_X_In_SortedTree
    j done
XGrt: lw $a1, 12($a1)
② jal Find_X_In_SortedTree
③ Done: lw $ra, 0($sp)
    addi $sp, $sp, 4
    jr $ra
```

```
Node *Find_X_In_Tree(int X,
                     struct Node *T)
{
    if (T && (T->X != X)) {
        if (T->X < X)
            T = Find_X_In_Tree(X, T->Left);
        else
            T = Find_X_In_Tree(X, T->Right);
    }
    return T;
}
```

① Save \$ra on stack

② Do recursion (jump recursively)

③ Restore \$ra from stack

Q: how does the stack look during the execution?