

044252 – מערכות ספרתיות

ומבנה המחשב

סיכום החומר



עריכה : אורי צנגוט

תוכן עניינים

2	ייצוג מספרים וקודים לקידוד ספרות
9	קודים לגילוי ותיקון שגיאות
12	Assembly
15	קריאה לפונקציות
16	אלגברה בוליאנית ופונקציות מיתוג
22	זמני השהייה של שערים לוגיים
26	תכן לוגי
32	רכיבי זיכרון וחישוב זמנים
37	מערכות סינכרוניות
41	Pipeline
43	תקשורת
45	פורמט פקודה ב-RISC-V
48	RISC-V Single Cycle
51	RISC-V Multi Cycle
56	חריגות ופסיקות
58	Pipelined RISC-V

הקדמה

זהו סיכום המכסה את רוב החומר הנלמד בקורס 044252 – מערכות ספרתיות ומבנה המחשב. הסיכום נכתב לפי מצגות ההרצאות והתרגולים המפורסמות באתר הקורס (והזכויות שמורות לעורכי מצגות אלה) ולפי המחברת האישית שלי. ייתכנו טעויות ויש לקחת את כל הכתוב בערבון מוגבל.

בהצלחה במבחן! 😊

ייצוג מספרים וקודים לקידוד ספרות

מושגים בסיסיים

- **בסיס** – מספר הספרות המייצגות מספר.
- **בסיס עשרוני** – בסיס בו יש 10 ספרות (0-9). מיקום הספרה (i) קובע את המשקל שתקבל.
- **משקל** – הספרה במקום i בבסיס b כלשהו, משקלה הוא $w = b^i$.
- **LSB** – least significant bit - הספרה בעלת החשיבות הנמוכה ביותר
- **MSB** – most significant bit - הספרה בעלת החשיבות הגבוהה ביותר

מעבר מבסיס b כלשהו לבסיס 10

מעבר זה נעשה ע"י הכפלת כל ספרה במספר, במשקלה המתאים.

$$N = \sum_i d_i * w_i$$

- N - ערך המספר במספרים עשרוניים
- d_i - הספרה ה־i
- $w_i = b^i$ - משקלה של הספרה ה־i, כאשר b -

לדוגמה:

$$(A31)_{16} = 10 * 16^2 + 3 * 16^1 + 1 * 16^0 = (2609)_{10}$$

$$(7135)_8 = 7 * 8^3 + 1 * 8^2 + 3 * 8^1 + 5 * 8^0 = (3677)_{10}$$

מעבר מבסיס 10 לבסיס b כלשהו

נפעל ע"פ האלגוריתם הבא: נחלק את המספר כל פעם ב־b, נכתוב כל פעם את שארית החלוקה, והתוצאה היא כל שאריות החלוקה כאשר הראשונה היא ה־LSB והאחרונה היא ה־MSB.

דוגמה 1: חשב את המספר 500 בבסיס הקסהדצימלי (בסיס 16)

ערך	שארית החלוקה ב־16	
500		
31	4	LSB
1	15 (F)	
0	1	MSB

תשובה: $(1F4)_{16} = (500)_{10}$

דוגמה 2 : חשב את המספר 13 בבסיס בינארי (בסיס 2)

ערך	שארית החלוקה ב-2	
13		
6	1	LSB
3	0	
1	1	
0	1	MSB

תשובה: $(1101)_2 = (13)_{10}$

מעבר בין בסיסים שהם חזקות אחד של האחר

נחלק את הספרות לקבוצות בגודל המעריך. לדוגמה, אם נרצה לעבור מבסיס בינארי לבסיס

הקסהדצימלי, נחלק את הספרות לקבוצות של 4, כיוון ש $2^4 = 16$.

כעת ניתן להמיר כל קבוצה לספרה ולשרשרן באותו הסדר.

לדוגמה, נחשב בבסיס 16 את המספר $(1101011100001111)_2$

$1101\{13=D\} 0111\{7\} 0000\{0\} 1111\{15=F\}$

ומכאן ש $(1101011100001111)_2 = (D70F)_{16}$

שיטת המשלים ל-2nd complement

שיטת המשלים ל-2 היא שיטה לייצוג מספרים עם סימן בבסיס בינארי. בשיטה זאת הסיבית הגבוהה ביותר (MSB - Most Significant Bit) מייצגת את הסימן של המספר (חיובי או שלילי) ושאר הספרות מייצגות את המספר. שיטה זו מקובלת במחשבים לייצוג בינארי של מספרים שעשויים להיות שליליים,



כיוון שקל יחסית לחשב את ערך המספר וכן לבצע עליו פעולות בסיסיות.

טווח הייצוג הוא

$$-2^{N-1} \leq A \leq 2^{N-1} - 1$$

כיוון שיש N-1 ביטים לייצוג ספרות, ועוד ביט אחד לייצוג סימן.

ייצוג המספר נעשה באופן הבא :

$$sign = \begin{cases} 0 - positive \\ 1 - negative \end{cases}$$

- קובעים מה הסימן, אם הוא חיובי, ה-MSB תהיה 0, ובשאר הספרות יהיה הערך הבינארי של המספר.
- אם המספר שלילי, מתחילים עם המספר בערך מוחלט (כבשלב הקודם)
- כל ספרה הופכת לספרה הנגדית (1 ל-0 ו-0 ל-1)
- מוסיפים 1.

לדוגמה, נייצג את המספר -10 בשיטת המשלים ל-2 :

נתחיל עם המספר בערך מוחלט +10 : 01010

נבצע היפוך של הספרות : 10101

נבצע +1 : 10110

וזוהי התוצאה הסופית.

(אם יש carry פשוט מתעלמים ממנו)

גלישה :

יש לשים לב כי חיבור שני מספרים שליליים עלול להביא לתוצאה חיובית (וההפך), כלומר לגלישה.

לדוגמה : (4-) + (3-) = (נבצע את שלושת השלבים ממקודם :)

011 100

100 011

101 100

$$7- \neq 1 = 001 = 101 + 100$$

לעומת זאת, חיבור מספר שלילי וחיובי (תקניים) תמיד ייתן תוצאה תקינה.

מספר בינארי	ערך (ללא סימן)	2's complement
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

נקודה צפה – floating point

נקודה צפה (floating-point) היא שיטת ייצוג למספרים ממשיים. הנקודה משתמשת בייצוג ספרתי על מנת לשמור ערך המהווה קירוב של מספר ממשי. השיטה יעילה גם לרישום מספרים ארוכים. שיטת הנקודה הצפה היא שיטה מקובלת לייצוג מספרים במחשב. השם "נקודה צפה" מציין את העובדה שבשיטת ייצוג זו, מספר הספרות משמאל לנקודה העשרונית ומימין לה גמיש, בהתאם למספר המיוצג (להבדיל מייצוג בשיטת נקודה קבועה, שבו מספר הספרות של המספר המיוצג קבוע). נשתמש בשיטה זו ע"מ לייצג מספרים מאוד גדולים או שברים.

נחלק את הביטים באופן הבא :



ערך המספר יקבע ע"פ הנוסחה הבאה :

- מצב מנורמל ($1 \leq E \leq 254$) :

$$\text{Value} = (-1)^S * 2^{E-B} * (1 + F)$$

- מצב לא מנורמל ($E = 0$) :

$$\text{Value} = (-1)^S * 2^{E-B+1} * F$$

- אם $E=255$:

○ אם $F=0$, אז $\text{Value} = \infty$

○ אם $F \neq 0$, אז $\text{Value} = \text{NaN}$, מייצג מספר לא קיים

* $B = \text{Bias}$, זהו גודל קבוע. ב-RiscV הוא שווה ל-127 (נשים לב שזה בדיוק חצי מ-254)

איך נקבע מה ערך Mantissa?

2^{-1}	2^{-2}	2^{-3}	...	2^{-23}
----------	----------	----------	-----	-----------

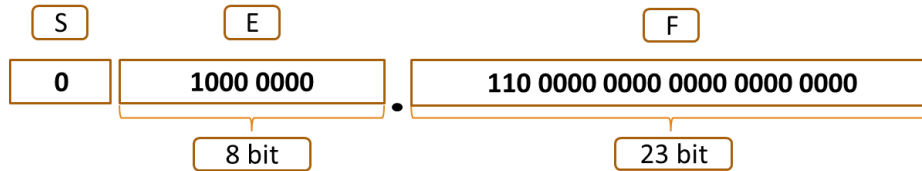
כל ביט במקום i מייצג את הערך 2^{-i} , כלומר :

לדוג' אם הביטים הם : 101 0100 0000 0000 0000 0000

אז נקבע כי

$$F = 2^{-1} + 2^{-3} + 2^{-5} = \frac{1}{2} + \frac{1}{8} + \frac{1}{32} = \frac{21}{32} = 0.65625$$

• (1) מהו ערכו של הייצוג הבא?



$$F = 2^{-1} + 2^{-2} = 0.75$$

$$\text{Value} = (-1)^0 * 2^{128-127} * 1.75 = 3.5$$

• כיצד יראה מספר זה בייצוג הקסדסימלי (בסיס 16)?

0x40600000

• (2) מהו ערכו של הייצוג הבא?



$$F = 2^{-3} + 2^{-4} + 2^{-5} + 2^{-6} + 2^{-9} = 0.236328125$$

$$\begin{aligned} \text{Value} &= (-1)^0 * 2^{132-127} * 1.236328125 \\ &= 1.236328125 * 2^5 \end{aligned}$$

• כיצד יראה מספר זה בייצוג הקסדסימלי (בסיס 16)?

0x421E4000

קודים בינאריים לייצוג ספרות בבסיס עשרוני

יש מספר דרכים לקודד ספרות עשרוניות לקודים המכילים 1 ו-0 (ייצוג בינארי). לעיתים פשוט נקבע נוסחה/טבלת מעבר (ללא חוקיות מסוימת) למשל $1001=1$, $0110=2$, $1111=3$, וכו'... ובפעמים אחרות נגדיר קוד ע"י חוקיות מסוימת, כמו בדוגמות הבאות:

קוד משוקלל

בקוד זה כל ספרה מיוצגת ע"י 4 סיביות, ולכל סיבית יש משקל מוגדר. ערך הספרה יהיה סכום המכפלה של כל סיבית במשקלה, כלומר:

$$value = \sum_{i=1}^4 w_i x_i = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4$$

קוד BCD

ספרה / משקלים	8 4 2 1	2 4 2 1	6 4 2 -3
0	0 0 0 0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1	0 1 0 1
2	0 0 1 0	0 0 1 0	0 0 1 0
3	0 0 1 1	0 0 1 1	1 0 0 1
4	0 1 0 0	0 1 0 0	0 1 0 0
5	0 1 0 1	1 0 1 1	1 0 1 1
6	0 1 1 0	1 1 0 0	0 1 1 0
7	0 1 1 1	1 1 0 1	1 1 0 1
8	1 0 0 0	1 1 1 0	1 0 1 0
9	1 0 0 1	1 1 1 1	1 1 1 1

דוגמות לקודים משוקללים:

- נשים לב כי אין מניעה שנוכל לתאר ספרה מסוימת בכמה דרכים: נסתכל למשל על הקוד המתואר בעמודה האמצעית, הספרה 6 יכולה להיות מיוצגת ע"י 1100 וגם ע"י 0110.
- יש להבחין בהבדל בין קוד BCD ובין מספר בבסיס בינארי! BCD הוא קוד משוקלל כאשר משקל כל ספרה הוא 8,4,2,1 – אך קוד זה הוא לייצוג ספרות! (לא מספרים!). לדוג' המספר 16 יקודד ב-BCD כך: 0001 0110 (צירוף של הספרה 1 והספרה 6), בעוד ש-16 בבסיס בינארי: 10000.

קוד ציקלי

בקוד ציקלי מילת קוד של ספרה כלשהי שונה ממילת הקוד של הספרה שלפניה ואחריה בסיבית אחת בלבד. כך גם מילת הקוד האחרונה שונה בספרה אחת בלבד ממילת הקוד הראשונה (ולכן קיבל את שמו – ציקלי). דוגמה שימושית היא קוד גריי, עליו נרחיב בעמוד הבא.

קוד Gray

קוד גריי הוא שיטה לקידוד מספרים באופן בינארי שבה כל מספר-עוקב שונה מקודמו בספרה בינארית אחת. קוד גריי הוא קוד מראה שנהגה על ידי פרנק גריי במעבדות בל בשנת 1947 כדי למנוע פלט שגוי ממתגים אלקטרומכניים. היום שימושו העיקרי של קוד גריי הוא בתיקוני טעויות בתקשורת ספרתית כגון טלוויזיה בכבלים.

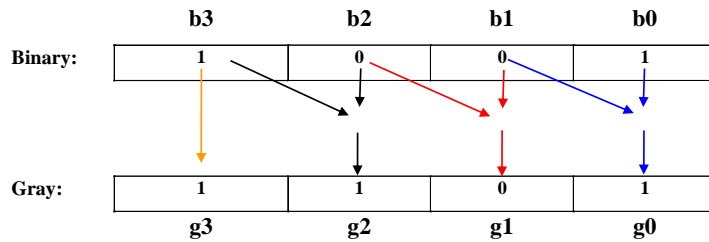
המרה מבינארי ל-Gray

את הביט הראשון מעתיקים כמו שהוא

עושים XOR בין כל זוג ביטים באופן

הבא:

$$\begin{cases} g_n = b_n \\ g_i = b_i \oplus b_{i+1} \end{cases}$$



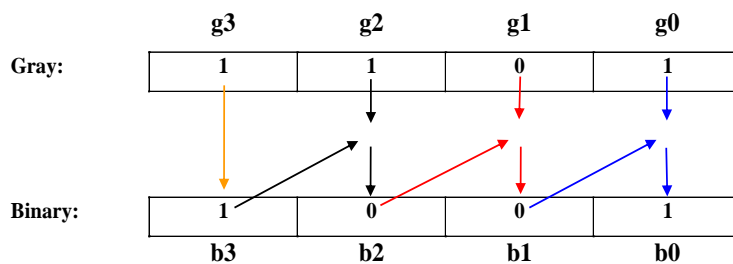
המרה מ-Gray לבינארי

את הביט הראשון מעתיקים כמו שהוא

עושים XOR בין כל זוג ביטים באופן

הבא:

$$\begin{cases} b_n = g_n \\ b_i = b_{i+1} \oplus g_i \end{cases}$$



(שיטה לזכור: רוצים כל הזמן לקחת כמה שיותר מידע מהבינארי, לכן החיצים בשני המקרים, יצאו ממנו)

קודים לגילוי ותיקון שגיאות

כשמעבירים אינפורמציה בינארית על קו תקשורת, עלולה לקרות תקלה, כך שהצד השני יקבל אינפורמציה שונה מזו שנשלחה.

לכן, תוכננו קודים לגילוי של שגיאות ובמקרים מתאימים אף תיקון שגיאות כאלה. המשדר שולח קוד במקום את המידע המקורי, והמקלט בודק אם הקוד תקין או לא.

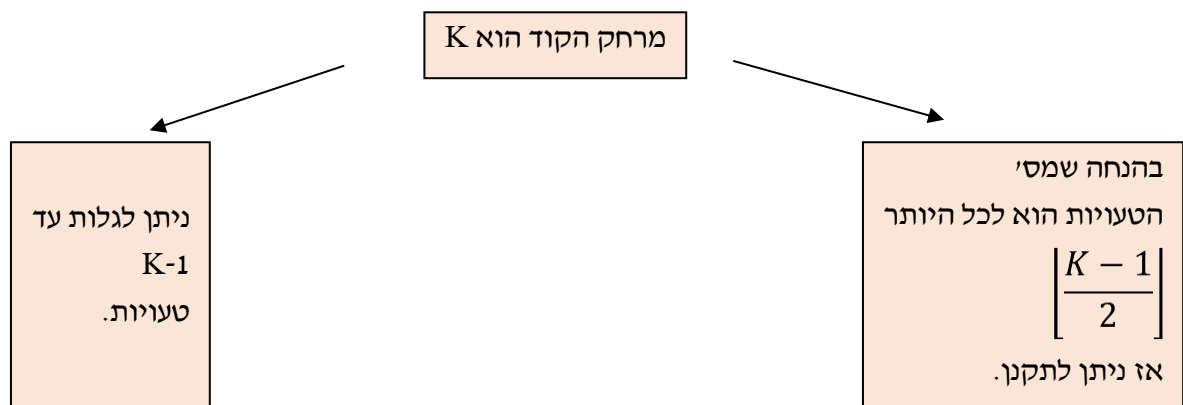
הגדרות

- קוד – אוסף מסוים של מילים מתוך כלל המילים האפשריות
- מרחק בין מילים – מס' הסיביות השונות בין המילים
- מרחק הקוד – מספר הסיביות המינימלי השונה בין מילה למילה

תיקון וגילוי שגיאות

כאשר מנתחים יכולת גילוי שגיאות של קוד, עלינו להניח מה מספר השגיאות המקסימלי שעלולות להתרחש בזמן שידור.

ע"פ מרחק הקוד נוכל לדעת כמה טעויות נוכל לגלות וכמה נוכל לתקן באופן הבא :



בדיקת זוגיות – Parity check

למילת הקוד המקורית מוסיפים ביט נוסף. את ביט זה נקבע ע"פ מספר ה1ים במילה – אם מס' ה1ים הוא זוגי הביט יהיה 0, ואם הוא אי-זוגי הביט הנוסף יהיה 1 (נשים לב שבשני המקרים התוספת של ביט זה הופכת את מס' ה1ים במילה החדשה להיות זוגי).

עוד דרך לקבוע את ערך ביט הזוגיות : לבצע XOR של כל הסיביות.

מרחק הקוד החדש יהיה 2, ולפי הנאמר קודם לכן – יש יכולת גילוי של שגיאה בודדת.

(הכוונה היא שאם המרחק הישן, לפני הוספת הביט, היה 1 אז החדש יהיה 2. אם הישן היה 2 אז החדש

יישאר 2. ואם הישן היה גדול מ2 הוא לא ישתנה כי הוספת הביט לא מקטינה את מרחק הקוד).

קוד חזרות

בקוד זה כל ביט משודר r פעמים.

לדוגמה עבור $r=4$, במקום הביט 1 ישודר 1111.

במקרה זה מרחק הקוד הוא r .

שאלה ממבחן – שאלה 3 מתוך בוחן אמצע 1 אביב 2020

לפיצרייה "פיצה-קוד" ישנה מערכת הזמנת משלוחים מיוחדת. לכל לקוח יש משדר שממנו הוא משדר את ההזמנה, ובפיצרייה יש מקלט אשר מקבל את כל ההזמנות ומפענח אותן. לכל סוג פיצה ישנו קידוד ייחודי

מהצורה $a_4 a_3 a_2 a_1 a_0$. המערכת עברה עדכון גרסה וכעת עבור כל סוג פיצה הקידוד המתאים נתון

בטבלה הבאה:

קוד	פיצה
00110	זיתים
01001	פטריות
00011	פפרוני
11000	מוצרלה
10001	אננס
01100	בולגרית

על מנת לשפר את השירות, החליטו להוסיף לכל מילת קוד את הסיביות הבאות:

$$c_0 = a_0$$

$$c_1 = a_1$$

$$c_2 = a_2$$

$$c_3 = a_3$$

$$c_4 = a_4$$

$$c_5 = a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4$$

מה מרחק הקוד של המילים החשדות: $a_0 a_1 a_2 a_3 a_4 c_0 c_1 c_2 c_3 c_4 c_5$?

פתרון: נשים לי כי מרחק הקוד המקורי הוא 2 (למשל בין פטריות ופפרוני). שכפול כל ביט תהפוך את מרחק הקוד החדש ל-4, ו-5 היא סיבית זוגיות (הפוכה) על הקוד המקורי – נשים לב שבכל מילה בקוד המקורי יש בדיוק שני 1ים, ולכן 5 תהיה תמיד 0 (או 1 לאחר שנבצע עליה NOT) ולא תשפיע על מרחק הקוד. ולכן מרחק הקוד החדש יהיה 4.

שאלה ממבחן – שאלה 2 מתוך מועד א' אביב 2020

נתון אוסף מילים באורך 4 ביט מהצורה abcd. אוסף המילים מהווה קוד, כאשר a הוא MSB ו-d הוא LSB. בכדי לשפר את יכולת גילוי השגיאות, הוחלט להרחיב את מילות הקוד המקורי כך שכל מילה תהיה מהצורה abcdxyz כאשר מתקיים:

- x – סיבית הזוגיות של המילה abcd אשר שייכת לקוד המקורי
 - y – סיבית אי-הזוגיות של המילה abcd אשר שייכת לקוד המקורי, כלומר סיבית אשר גורמת למספר ה-1-ים במילה להיות אי זוגי.
 - z – סיביות הזוגיות של המילה abcdxyz.
- הניחו כי הקוד מכיל לפחות שתי מילים, ובחרו את הטענה הנכונה:
- א. עבור כל קוד מהצורה abcd, הקוד המורחב, abcdxyz, מגדיל את מרחק הקוד ב-3.
 - ב. בהינתן כי מרחק הקוד של הקוד המקורי abcd הוא 2, מרחק הקוד של הקוד החדש, abcdxyz, הינו בהכרח 5.
 - ג. קיים קוד מקורי, abcd, בעל מרחק קוד השווה ל-1, אשר הקוד המורחב שנוצר על בסיסו, abcdxyz, הוא בעל מרחק קוד השווה ל-4.
 - ד. קיים קוד מקורי, abcd, בעל מרחק קוד השווה ל-1, אשר הקוד המורחב שנוצר על בסיסו, abcdxyz, הוא בעל מרחק קוד השווה ל-1.
 - ה. תשובות ב' ו-ג' נכונות.

פתרון:

- א. לא נכון. עבור קוד מקורי abcd שמרחק הקוד שלו הוא גדול מ-2, הוספת ביט זוגיות לא תגדיל את מרחק הקוד.
- ב. לא נכון. אותו נימוק כמו בסעיף א'.
- ג. נכון. נקח למשל את הקוד: {0001, 0000} – מרחק הרוד הוא 1, והקוד המורחב הוא {0001100, 0000011} ומרחק הקוד שלו הוא 4.
- ד. לא נכון. אם בקוד המקורי מרחק הקוד הוא 1, אז בוודאות יש 2 מילות קוד בהן מס' ה-1ים הוא זוגי באחת ואי זוגי בשנייה וזה גורר x ו-y שונים (מרחק קוד 2 לפחות).

Assembly

אסמבלי זו שפת התכנות הבסיסית ביותר והקרובה ביותר לשפת מכונה. השפות העיליות איתן נכתוב (C, Java) יעברו תרגום לאסמבלי ומשם לשפת מכונה.

רגיסטרים

שלא כמו בשפות עליות, באסמבלי אין משתנים, והפעולות נעשות על אובייקטים הנקראים **רגיסטרים**.

- הרגיסטרים הם חלק מהחומרה ולכן הגישה אליהן מאוד מהירה.
- יש מספר מצומצם של רגיסטרים.
- פעולות אריתמטיות יכולות להתבצע רק על הרגיסטרים (ולא על מידע בזיכרון).

במעבד RISC-V, המעבד איתו נעבוד לאורך הקורס, יש 32 רגיסטרים הממוספרים X0-X31. רגיסטר x0 הוא מיוחד והוא מחזיק תמיד את הערך 0 (כך שבפועל יש לנו 31 רגיסטרים פנויים).

בנוסף למספור, לכל רגיסטר ניתן שם ובד"כ גם תפקיד ייעודי, כמו שניתן לראות בטבלה הבאה:

Name	Reg #	Use	Saver
\$zero	0	Constant 0	-
ra	1	Return Address	Caller
sp	2	Stack Pointer	Callee
gp	3	Global Pointer	-
tp	4	Thread Pointer	-
t0-t2	5-7	Temp	Caller
s0-s1	8-9	Saved Registers	Callee
a0-a7	10-17	Function arguments/ return values	Caller
s2-s11	18-27	Saved Registers	Callee
t3-t6	28-31	Temp	Caller

פקודות נפוצות בקורס

פקודת lw

הפעולה ניגשת אל הכתובת $imm+rs1$, כלומר לכתובת השמורה ברגיסטר rs1 ועוד הקבוע הנמצא בimm, וטוענת את הערך שנמצא שם לתוך הרגיסטר rd. נשים לב שנטענים 4 בתים (מילה = 4 בתים = 32 ביטים). למשל, אם אחנו במערך מספרים מטיפוס int נקפץ בקפיצות של 4 (כלומר אם S0 הוא הרגיסטר ובו כתובת האיבר הראשון של המערך, אז האיבר השני של המערך יהיה בכתובת S0), האיבר השני יהיה בכתובת S0 (וכן הלאה...)

פקודת Add

פעולה המחברת את rs1 ואת rs2 ושומרת את תוצאת החיבור בrd.

פקודת Addi

פעולה המחברת את rs1 ואת הקבוע imm ושומרת את תוצאת החיבור בrd.

פקודת slli

הפקודה מזיזה מספר ביטים שמאלה, ושמה אפסים בביטים הריקים ש"התפנו". פעולה זו שקולה להכפלת המספר ב 2^k , כאשר k הוא מספר הביטים בו זזנו שמאלה. אמנם, קיימת פעולת mul, להכפלה, אך אם מדובר בחזקות של 2 יותר יעיל להשתמש בslli.

פקודת jal

Jump and Link – פקודה זו קופצת לרגיסטר הנמצא imm מקומות לאחר הפקודה הנוכחית, כלומר מעדכנת את PC להיות PC+imm.

בנוסף, ברגיסטר rd נשמרת כתובת הפקודה הבאה, כלומר $rd = PC + 4$. נשים לב כי הפקודה 'j' היא פסואדו-פקודה, כלומר, היא אינה פקודה אמיתית. וכאשר האסמבלר נתקל בן הוא למעשה משתמש בJAL וברגיסר היעד rd מכניס את רגיסטר האפס X0 ובכך למעשה לא נשמרת באף רגיסטר כתובת החזרה.

פקודת jalr

Jump and Link register – בדומה לפקודת jal, רק כשהפעם כתובת הקפיצה אינה הכתובת הנוכחית ועוד imm, אלא הכתובת המצויה ברגיסטר rs ועוד imm. נועד לטובת קפיצה למקומות רחוקים בתכנית, שהimm של jal (20 ביטים בלבד) קטן מהמרווח בין הפקודה הנוכחית לכתובת אליה נרצה לקפוץ.

פקודות נוספות שחשוב להכיר:

MNEMONIC	FMT	NAME	DESCRIPTION
add rd, rs1, rs2	R	ADD (Word)	$R[rd] = R[rs1] + R[rs2]$
addi rd, rs1, imm	I	ADD Immediate (Word)	$R[rd] = R[rs1] + imm$
beq rs1, rs2, imm	SB	Branch Equal	if ($R[rs1] == R[rs2]$) $PC = PC + \{imm, 1b'0\}$
blt rs1, rs2, imm	SB	Branch Less Than	if ($R[rs1] < R[rs2]$) $PC = PC + \{imm, 1b'0\}$
bne rs1, rs2, imm	SB	Branch Not Equal	if ($R[rs1] != R[rs2]$) $PC = PC + \{imm, 1b'0\}$
j imm	UJ	Jump	$PC = PC + \{imm, 1b'0\}$ pseudo uses jal
jal	UJ	Jump & Link	$R[rd] = PC + 4$; $PC = PC + \{Imm, 1b'0\}$
jalr	I	Jump & Link Register	$R[rd] = PC + 4$; $PC = R[rs1] + imm$
jr rd	I	Jump Register	$PC = R[rs1]$ pseudo uses jalr
lui rd, Imm	U	Load Upper Immediate	$R[rd] = \{32b'imm < 31>, imm, 12b'0\}$
lw rd, imm(rs1)	I	Load Word	$R[rd] = \{32'bM[(31), M[R[rs1] + imm](31:0)]\}$
ori rd, rs1, imm	I	OR Immediate (Word)	$R[rd] = R[rs1] \mid imm$
slli rd, imm(rs1)	I	Shift Left Immediate	$R[rd] = R[rs1] \ll imm$
srli rd, imm(rs1)	I	Shift Right Immediate	$R[rd] = R[rs1] \gg imm$
subi rd, rs1, imm	I	SUBtract Immediate (Word)	$R[rd] = R[rs1] - imm$
sw rs2, imm(rs1)	S	Store Word	$M[R[rs1] + imm](31:0) = R[rs2](31:0)$

שאלה ממבחן – שאלה 14 מתוך מועד א' אביב 2020

סטודנט חרוץ החליט לממש את אלגוריתם המיון "bubble sort" בעזרת קוד אסמבלי. האלגוריתם יבצע מיון על מערך של מספרים, כאשר כל מספר הוא בגודל של 4 בתים. עקב תקלה, חלקים מן המימוש נמחקו. עליכם להשלים את חלקי הקוד החסרים (מסומנים בקו תחתון) בכדי שהמימוש יפעל כנדרש. לנחיותכם מצורף מימוש אלגוריתם המיון בקוד C:

```
void bubbleSort(int arr[], int N)
{
    int i, j;
    for (i = 0; i < N-1; i++)

        // Last i elements are already in place
        for (j = 0; j < N-i-1; j++)
            if (arr[j] > arr[j+1])
                swap(&arr[j], &arr[j+1]);
}
```

הפונקציה swap מבצעת החלפה במיקומם של שני איברים במערך.

במהלך המימוש הניחו כי המיפוי בין רגיסטרים למשתנים הוא:

$s0 \rightarrow i, s1 \rightarrow N, s2 \rightarrow j, s10 \rightarrow arr$

שימו לב כי רגיסטר s1 מכיל את גודל המערך (N).

פתרון:

0x1AA0 0000	Main:	addi s0, x0, 0	// s0 = i = 0
0x1AA0 0004		addi s7, s1, -1	// s7 = N-1
0x1AA0 0008		addi s2, x0, 0	// s2 = j = 0
0x1AA0 000C	OuterLoop:	sub s3, s1, s0	// s3 = N-i
0x1AA0 0010		addi s3, s3, -1	// s3 = N-i-1
0x1AA0 0014	InternalLoop:	slli t0, s2, <u>2</u>	// t0 = 4j
0x1AA0 0018		add t0, t0, <u>s10</u>	// t0 = arr + 4j = a[j]
0x1AA0 001C		addi t1, <u>t0</u> , 4	// t1 = arr + 4j + 4 = a[j+1]
0x1AA0 0020		lw a0, 0(t0)	// access a[j]
0x1AA0 0024		lw a1, 0(t1)	// access a[j+1]
0x1AA0 0028		bge <u>a1</u> , <u>a0</u> , <u>AfterSwap</u>	// a1 ≥ a0 אם a0 > a1, אף -1, אם a0 = a1
0x1AA0 002C		<u>sw</u> a1, 0(<u>t0</u>)	// swap cells
0x1AA0 0030		<u>sw</u> a0, 0(<u>t1</u>)	
0x1AA0 0034	AfterSwap:	addi s2, s2, 1	
0x1AA0 0038		bne s3, s2, InternalLoop	
0x1AA0 003C		addi <u>s2</u> , x0, <u>0</u>	// j = 0
0x1AA0 0040		addi s0, s0, 1	// i++
0x1AA0 0044		bne s0, s7, OuterLoop	
0x1AA0 0048	Exit:		// done

קריאה לפונקציות

להלן קונבנציית הקריאה לפונקציות המקובלת:

הפונקציה הקוראת - caller

1. גיבוי הרגיסטרים בהם הפונקציה הנקראת יכולה לעשות שינויים {a0-a7, ra, t0-t6}
2. להכין את הארגומנטים של הפונקציה a0-a7 (הפרמטרים שהפונקציה מקבלת)
3. לשמור את כתובת החזרה לאחר הפונקציה ב-ra

הפונקציה הנקראת - callee

פרולוג

1. הקצאת מחסנית ע"י שינוי ערך sp – בפועל נעשה ע"י הקטנת הכתובת ב-sp
2. גיבוי הערכים שהיא הולכת לשנות {s0-s11}

– ביצוע חישובים –

אפילוג

1. שמירת תוצאות החישוב ב-a0/a1
2. שחזור הרגיסטרים שגובו קודם לכן (שלב 2 בפרולוג) ושונו ע"י הפונקציה
3. לשחרר את המחסנית ע"י הגדלת ערך sp
4. לחזור לכתובת החזרה השמורה בכתובת ra (ע"י שימוש ב-jr).

הפונקציה הקוראת - caller

שחזור הערכים שגובו לפני הקריאה לפונקציה {a0-a7, ra, t0-t6}.

שיטה לזכור – מי צריך לגבות את מי: הרגיסטרים שמתחילים ב-s הם "saved" כלומר, הם לא אמורים להשתנות במהלך הריצה, ואם בכל זאת callee מחליט להשתמש בהם – באחריותו לגבות אותם ולשחזר אותם לערכים המקוריים כשיסיים.

באופן דומה הרגיסטרים המתחילים ב-a, t הם "temp" זמניים, ולכן אם לפונקציה הקוראת caller חשוב המידע הנמצא בהם – באחריותה לגבות אותם לפני הקריאה לפונק' אחרת.

האם תמיד צריכה הפונקציה הקוראת לגבות ערכים?

את a0-a7, t0-t6 לא תמיד צריך לגבות, רק אם עושים בהם שימוש לאחר מכן. לעומד זאת, את ra תמיד נצטרך לגבות! גם הפונקציה הקוראת היא פונק' שנקראה ע"י פונקציה אחרת – גם מע' ההפעלה צריכה לדעת לאן לחזור ולכן גם בקריאה ל Main נגבה את ra.

פונקציות הקוראות לפונקציות אחרות, פונקציות רקורסיביות

בקריאה רקורסיבית לפונקציות נגבה ב-callee גם את a0-a7 וגם את ra על מנת שנוכל לדעת לאן לחזור (הגיבוי יעשה כרגיל במחסנית בדומה ל-s0-s11 בשלב 2)

אלגברה בוליאנית ופונקציות מיתוג

אלגברה בוליאנית היא התחום המתמטי העוסק במבנים האלגבריים הקרויים "אלגברה בוליאנית", ובנושאים הקשורים לכך. אחד היישומים המוכרים של התחום הוא בלוגיקה בוליאנית. אלגברה בוליאנית מיושמת גם בתורת הקבוצות ואף באלקטרוניקה.

זהויות – לוגיקה צירופית (מופיע גם בדף נוסחאות)

$x+x=x$	$x \cdot x=x$	1. אדישות
$x+0=x$ $x+1=1$	$x \cdot 0=0$ $x \cdot 1=x$	2. ערכים אדישים ושולטים
$x+y=y+x$	$x \cdot y=y \cdot x$	3. חילוף
$(x+y)+z=x+(y+z)$	$(x \cdot y) \cdot z=x \cdot (y \cdot z)$	4. קיבוץ
$x+x'=1$	$x \cdot x'=0$	5. השלמה
$x(y+z)=xy+xz$	$x+(y \cdot z)=(x+y) \cdot (x+z)$	6. פילוג
$x(x+y)=x$	$x+xy=x$	7. בליעה (ראשון)
$x(x'+y)=xy$	$x+x'y=x+y$	8. בליעה (שני)
	$xy+x'z+yz=xy+x'z$	9. קונצנוס
	$(x')'=x$	10. היפוך עצמי
$(x+y)'=x' \cdot y'$	$(xy)'=x'+y'$	11. דה-מורגן
	$(x+y)(x+z)=x+yz$	12.

השער XOR

- מקבל 1 אם שתי סיביות הכניסה שונות אחת מהשניה
- מקבל 0 עבור מס' זוגי של 1ים ו1 עבור מספר איזוגי של 1ים
- מבצע חיבור רגיל (ללא סיבית הcarry) כלומר, חיבור מודולו 2.
- זהויות:

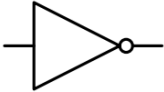


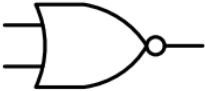


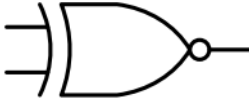
$$A \oplus B = B \oplus A \quad \circ$$

$$A \oplus 1 = \bar{A} \quad A \oplus 0 = A \quad \circ$$

$$A \oplus A = 0 \quad A \oplus \bar{A} = 1 \quad \circ$$

$$A=B \text{ אז } A \oplus B = B \oplus C \quad \circ$$

סיכום שערים לוגיים

רישום	טבלת אמת			כתיב מתמטי	שם השער														
		<table><tr><td>y</td><td>\bar{x}</td></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	y	\bar{x}	0	1	1	0		$NOT(x) = \bar{x}$	NOT								
y	\bar{x}																		
0	1																		
1	0																		
	<table><tr><td>x</td><td>y</td><td>$x + y$</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	$x + y$	0	0	0	0	1	1	1	0	1	1	1	1		$OR(x, y) = x + y$	OR
x	y	$x + y$																	
0	0	0																	
0	1	1																	
1	0	1																	
1	1	1																	
	<table><tr><td>x</td><td>y</td><td>$x \cdot y$</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	$x \cdot y$	0	0	0	0	1	0	1	0	0	1	1	1		$AND(x, y) = x \cdot y$	AND
x	y	$x \cdot y$																	
0	0	0																	
0	1	0																	
1	0	0																	
1	1	1																	
	<table><tr><td>x</td><td>y</td><td>$\overline{x + y}$</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	$\overline{x + y}$	0	0	1	0	1	0	1	0	0	1	1	0		$NOR(x, y) = \overline{x + y}$	NOR
x	y	$\overline{x + y}$																	
0	0	1																	
0	1	0																	
1	0	0																	
1	1	0																	
	<table><tr><td>x</td><td>y</td><td>$\overline{x \cdot y}$</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	$\overline{x \cdot y}$	0	0	1	0	1	1	1	0	1	1	1	0		$NAND(x, y) = \overline{x \cdot y}$	NAND
x	y	$\overline{x \cdot y}$																	
0	0	1																	
0	1	1																	
1	0	1																	
1	1	0																	
	<table><tr><td>x</td><td>y</td><td>$x \oplus y$</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	$x \oplus y$	0	0	0	0	1	1	1	0	1	1	1	0		$XOR(x, y) = x \oplus y$ $= \bar{x}y + x\bar{y}$	XOR
x	y	$x \oplus y$																	
0	0	0																	
0	1	1																	
1	0	1																	
1	1	0																	
	<table><tr><td>x</td><td>y</td><td>$\overline{x \oplus y}$</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	$\overline{x \oplus y}$	0	0	1	0	1	0	1	0	0	1	1	1		$XNOR(x, y) = \overline{x \oplus y}$ $= \bar{x}\bar{y} + xy$	XNOR
x	y	$\overline{x \oplus y}$																	
0	0	1																	
0	1	0																	
1	0	0																	
1	1	1																	

פונקציות מיתוג הגדרה – פונקציית מיתוג ב- n משתנים $f(x_1, \dots, x_n)$ הינה כל התאמה בין 2^n הצירופים השונים של המשתנים x_1, \dots, x_n לבין הקבוצה $\{0,1\}$.

צורות קנוניות של פונקציות

פונקציה אחת ניתנת לתיאור באמצעות ביטויי מיתוג שונים. נגדיר שתי צורות סטנדרטיות לתיאור פונקציות:

צורה קנונית של סכום מכפלות (Sum of Products, SoP)

$$\text{כגון: } f(x,y,z)=x'y'z+x'yz'+xyz$$

מכפלה המכילה את כל n המשתנים (ליטרלים) של הפונק' תקרא 'מינטרם' - minterm. כל מכפלה מקבלת ערך '1' רק עבור צירוף אחד של ערכי המשתנים, וערך הביטוי כולו יהיה 1 אם לפחות אחת המכפלות ערכה '1'. בצורה הקנונית SoP יש לרשום את סכום המכפלות (minterms) המתאימות לשורות בטבלת האמת בהן הפונקציה מקבלת ערך '1'.

רישום מקוצר: $f(x,y,z) = \sum(0,2,3,6,7)$, כאשר בדוגמה זו שורות 0,2,3,6,7 בטבלת האמת של הפונק' יקבלו את הערך '1'.

צורה קנונית של מכפלת סכומים (Product of Sums, PoS)

$$\text{כגון: } f(x,y,z)=(x+y+z)(x+y'+z)(x'+y+z')$$

סכום המכיל את כל n הליטרלים יקרא 'מקסטרם' - maxterm. כל סכום מקבל את הערך '0' רק עבור צירוף אחד של ערכי המשתנים, וערך הביטוי כולו יהיה '0' אם לפחות אחד הסכומים במכפלת הסכומים יהיה שווה ל'0'. בצורה הקנונית PoS יש לרשום את מכפלת כל הסכומים המתאימים לשורות בטבלת האמת של הפונק' בהן הפונקציה מקבל ערך 0.

כאשר נרשום את הליטרלים בסכומים, נקח את הנגדי של כל אחד מכפי שהוא מופיע בטבלת האמת. לדוגמה, אם עבור שורה 5 ($x,y,z = 1,0,1$) הפונק' מקבלת 0, המקסטרם המתאים יהיה $(x'+y^0+z')$. רישום מקוצר: $f(x,y,z) = \prod(1,4,5)$, כאשר בדוגמה זו שורות 1,4,5 בטבלת האמת של הפונק' יקבלו את הערך '0' (זוהי אותה פונק' מהדוגמה של SoP).

הצורה הקנונית הינה יחידה (עד כדי חילוף), לפונקציות בעלות אותה טבלת-אמת, אותה צורה קנונית - אלו פונקציות שוות/זהות/שקולות

משפט הפיתוח של שאנון

כל פונקציית מיתוג $f(x_1, \dots, x_n)$ ניתנת לרישום בתור:

$$f(x_1, x_2, \dots, x_n) = x_1 \cdot f(1, x_2, \dots, x_n) + x_1' \cdot f(0, x_2, \dots, x_n)$$

לדוג' עבור פונקציה של משתנה יחיד:

$$f(x) = f(1)x + f(0)\bar{x}$$

מערכת פעולות שלמה

- הגדרה: קבוצת פעולות נקראת שלמה אם ניתן להציג בעזרתה את כל הפונקציות הלוגיות $\{', +, \cdot\}$.
- בעזרת דה-מורגן ניתן להראות שכל קבוצת פעולות היא שלמה אם ניתן להציג בעזרתה את אחד מהזוגות: $\{', +\}$ או $\{', \cdot\}$.
- אופרטור יחיד המהווה מערכת פעולות שלמה נקרא **אוניברסלי**, דוגמות לאופרטור כזה: NAND, NOR.

משפט: תנאי הכרחי לכך שמערכת תהיה מע' פעולות שלמה הוא $f(a, \dots, a) = \bar{a}$.

נשים לב שזהו תנאי הכרחי ולא תנאי מספיק (שימוש עיקרי: אם התנאי לא מתקיים זה מוכיח שהמע' אינה שלמה). משפט זה נכון רק לגבי מערכת שבה יש פונקציה יחידה!

מערכת פעולות חצי שלמה - מערכת פעולות תיקרא חצי שלמה אם הינה שלמה רק בתוספת קבועים (0 או 1 או שניהם).

שאלה ממבחן – שאלה 8 מתוך מועד ב' חורף 2017-2018

נתונה קבוצת האופרטורים Δ שבעזרתה ניתן לממש:

1. את כל הפונקציות הלוגיות בעלות משתנה אחד בלבד

2. את הפונקציה $f(x, y, z) = x'y + y'z$

האם הקבוצה Δ היא שלמה / חצי שלמה / אף אחת מהן?

פתרון:

לפי 1, נתון כי היא יכולה לממש את כל הפונק' בעלות משתנה אחד – לכן היא יכולה לממש NOT.

לפי 2, היא יכולה לממש את f הנתונה. נכניס בארגומנטים של f את המשתנים הבאים (כזכור יש לנו NOT):

$$f(x', y, y) = x''y + y'y = xy$$

ומכאן קיבלנו גם שער AND, ולכן מע' זו היא מערכת פעולות שלמה.

צמצום פונקציות מיתוג בעזרת מפת קרנו

משבצות שכנות: שתי משבצות בפת קרנו תיקראנה שכנות, אם ייצוגן (הבינארי) נבדל בסיבית אחת בלבד. כאשר המפה מייצגת טבלת אמת של n משתנים, לכל משבצת יש n שכנים.

קוביה m מימדית: אוסף מירבי של 2^m משבצות שכולן זהות בייצוגן הבינארי ב- m מקומות.

לדוג': במפת קרנו של $n=4$ משתנים (4^*4) , קוביה חד-מימדית ($m=1$) תהיה מורכבת מזוגות של מינטרמים, כאשר כל זוג בעל $4-1=3$ סיביות זהות. קוביה דו-מימדית ($m=2$) תהיה מורכבת מרביעיות של מינטרמים, כאשר כל רביעיה בעלת $4-2=2$ סיביות זהות. וכן הלאה.

משפט הקוביה: קוביה m מימדית במפת קרנו של n משתנים מתאימה למכפלה של $n-m$ ליטרלים.

צמצום בעזרת מפת קרנו

המטרה היא כיסוי ה-1ים במפה ע"י מינימום קוביות, גדולות ככל האפשר.

גורר – נאמר f מכסה את g (ונרשום $f \supseteq g$) אם לכל כניסה עבורה g תוציא 1, גם f תוציא 1, כלומר מתקיים: $1: f = 1 \Rightarrow g = 1$. במקרה זה (אם g היא מכפלת ליטרלים) נאמר ש g גורר של f .

דוגמה: אם $f=yz$, אז $g=yz'w$ הוא גורר של f .

גורר ראשוני (PI) – גורר שכל השמטה של ליטרל (סיבית) ממנו, יוצרת מכפלה שאינה מכוסה ע"י הפונקציה.

גורר ראשוני הכרחי (EPI) – גורר ראשוני המכסה מינטרם שאינו מכוסה ע"י אף גורר ראשוני אחר, כלומר – אם לא נסמן אותו, יהיה 1 כלשהו לפחות שלא יהיה מכוסה ע"י אף גורר ראשוני אחר.

תהליך הצמצום:

1. סימון כל הגוררים הראשוניים
2. מתוכם, בחירת כל הגוררים הראשוניים ההכרחיים.
3. אם כל הגוררים הראשוניים ההכרחיים מכסים את הפונק' סיימנו
4. אם לא, נוסיף מספר מינימאלי של גוררים ראשוניים (לא הכרחיים) כך שנכסה את כל f .

Don't Care – צירופי ברירה

אלו צירופים שבהם הפונקציה אינה מוגדרת ונוכל להגדירם כ0 או 1 כרצוננו.

דוגמות לשימוש: המשבצות מייצגות שעות ביום (15-12 יסומנו ב \emptyset , כיוון שאין שעה כזו), המשבצות מייצגות ספרות עשרוניות (15-10 יסומנו ב \emptyset), מייצגות מספרים טבעיים (0 יסומן ב \emptyset) וכו'.

נחליט אם לסמן את צירוף הברירה ב0 או 1, לפי הבחירה התביא לנו משבצות גדולות ככל האפשר.

שאלה ממבחן – שאלה 18 מתוך מועד א' חורף 2017-2018

נתונה הפונקציה הבאה :

$$f(w,x,y,z)=\Sigma(1,2,5,6,8,11,12,15)+\Sigma_{\emptyset}(4,7,13,14)$$

עבור הפונקציה הנתונה, מהו מספר הגוררים הראשוניים (PI ,) ומתוכם מהו מספר הגוררים הראשוניים
ההכרחיים (EPI ,) לפי צמצום כסכום מכפלות?

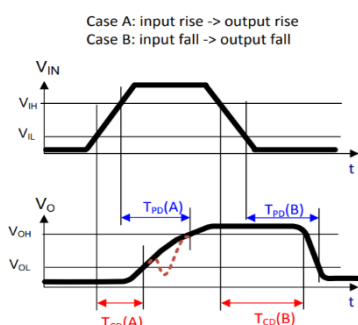
פתרון : יש 4 הכרחיים ו5 לא הכרחיים, באופן הבא :

wx \ yz	00	01	11	10
00		∅	1	1
01	1	1	∅	
11		∅	1	1
10	1	1	∅	

נסמן כל זוג 1ים כגוררים, וגם את הקובייה הכחולה. נשים לב שהקובייה בכחול X היא גורר ראשוני, הרי כל השמטה של ליטרל ממנה לא תכוסה ע"י הפונקציה. אולם, זהו אינו גורר ראשוני הכרחי, כיוון שניתן לא לסמנו ועדיין שאר הפונקציה תכוסה.

זמני השהייה של שערים לוגיים

מושגים:



- t_{cd} – הזמן מתחילת ביצוע שינוי בכניסה במהלכו מובטח כי היציאה אינה משתנה את ערכה הלוגי. נמדד מהרגע בו יש שינוי ראשוני בערך הלוגי הישן של הכניסה, עד הרגע בו הערך הלוגי במוצא מתחיל להשתנות.
- t_{pd} – הזמן מרגע סיום ביצוע השינוי בכניסה שלאחריו מובטח כי היציאה התייצבה על ערכה הלוגי הסופי. נמדד מהרגע בו הכניסה התייצבה על ערכה החדש, עד הרגע בו היציאה התייצבה על ערכה החדש.
- $t_{pdLH} - t_{pd}$ של שינוי ביציאה מ'0' ל'1'.
- $t_{pdHL} - t_{pd}$ של שינוי ביציאה מ'1' ל'0'.
- t_{pdHH}, t_{pdLL} – כאשר אין שינוי במוצא, אך לפרק זמן מסוים המוצא משתנה ולאחר מכן חוזר לערכו הקודם (glitch או hazard).

מתקיים:

$$\begin{aligned}
 t_{pd,H} &= \max\{t_{pd,LH}, t_{pd,HH}\} & t_{cd,H} &= \min\{t_{cd,LH}, t_{cd,HH}\} \\
 t_{pd,L} &= \max\{t_{pd,HL}, t_{pd,LL}\} & t_{cd,L} &= \min\{t_{cd,HL}, t_{cd,LL}\} \\
 t_{pd} &= \max\{t_{pd,L}, t_{pd,H}\} & t_{cd} &= \min\{t_{cd,L}, t_{cd,H}\} \\
 t_{pd} &\geq t_{cd}
 \end{aligned}$$

זמני השהייה לאורך מסלול - כאשר נרצה לחשב זמן השהייה של שערים לוגיים המחוברים אחד לשני, נעקוב אחרי מסלול מסויים, ובדוק כיצד כל שינוי בכניסה משפיע על מוצא מסוים. אם שתי כניסות מחוברות לשער לוגי מסוים, נקבע את הכניסה השניה (זו שלא שייכת למסלול אותו אנו בוחנים) להיות הערך האדיש לאותו שער (לדוג' '0' ב-OR ו'1' ב-AND), כך שאנו מבטיחים כי כניסה זו לא תשפיע על ההשהיה לאורך המסלול.

השהייה של פונקציות צירופיות - זמן השהיה של כניסה x_0 כלשהי הוא הזמן שנדרש מרגע שכניסה זו משתנה ועד שמוצא המערכת משתנה, כאשר שאר הכניסות קבועות. זמן ההשהיה תלוי במימוש ולא בלוגיקה (כלומר, לפונקציות זהות הממומשות באופן שונה, יתכנו זמני השהייה שונים). יתכן זמן השהיה שונה מכניסה מסוימת למוצא בהתאם לערך ביתר הכניסות. יתכן גם שעבור ערכים בכניסות האחרות, כניסה x_0 לא תשנה את המוצא. זמני ההשהיה מכניסה מסוימת נקבעים לפי המקרה הגרוע ביותר.

כיצד נגלה אילו שינוי בערכי הכניסה גוררים שינוי במוצא המערכת כולה? ניעזר במפת קרנו:

$z \backslash xy$	00	01	11	10
0	1	1	0	1
1	0	1	1	1

נעקוב ונראה בין איזה משבצות שכנות יש שוני במוצא הפונקציה. כיוון שמפת קרנו מבוססת על קוד gray, בין כל 2 משבצות שכנות יש שוני רק בביט אחד, לכן נוכל לקבוע מה הערך הקבוע של שאר הביטים.

הבהוב סטטי (Static Hazard) 1-0-1

כאשר יש שוני בכניסה מסוימת שלא אמור להשפיע על המוצא, אך בכל זאת לזמן רגעי הכניסה משתנה ולאחר מכן חוזרת לערך המקורי – תופעה זו נקראת הבהוב סטטי. תופעה זו נגרמת ע"י הבדלים בזמני ההתפשטות ברכיבים שונים, והיא אופיינית למעבר במפת קרנו מגורר אחד לגורר אחר, לדוג':

$Y_3 \backslash Y_1 Y_2$	00	01	11	10
0				1
1		1	1	1

במעבר זה מוצא המע' עלול להיראות כך: 1-0-1

$Y_3 \backslash Y_1 Y_2$	00	01	11	10
0				1
1		1	1	1

ניתן למנוע הבהוב סטטי ע"י הוספת גורר נוסף המכסה את החץ שהופיע במפת קרנו. ערכו של גורר זה לא ישתנה בעת המעבר. אם נחזור לדוג' הגורר הנוסף יראה כך:

ומבחינת המימוש הדבר כרוך בהוספת שערים לוגיים.

עם זאת, פתרון זה מבוסס על ההנחה כי בו זמנית לא משתנה יותר

מכניסה אחת למעגל, ושינויים נוספים בכניסות לא יקרו עד אשר יסתיימו כל השינויים בתוך המעגל הנבועים משנוי הכניסה האחרון.

נשים לב כי לאחר פתרון זה תתקבל צורה שאינה הצורה המינימלית של הפונקציה!

הבהוב דינמי (Dynamic Hazard) 1-0-1-0

כאשר יש שוני בכניסה מסוימת אשר אמור להשפיע על המוצא, אך השינוי נעשה תוך שלושה מעברים לפחות – תופעה זו נקראת הבהוב דינמי.

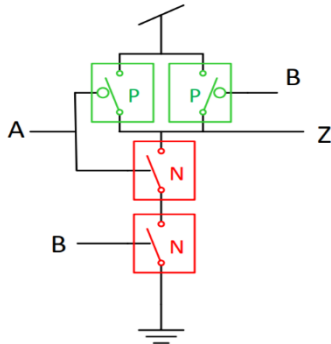
במעבר זה מוצא המע' עלול להיראות כך: 1-0-1-0

פתרונות לבעיה זו דומים לפתרון הבהוב סטטי, אולם בעיה זו מורכבת יותר ואין פתרון כללי (יש מקרים שאינם ניתנים לפתרון).

בניית שערים לוגיים באמצעות מתגים

המימוש הטכנולוגי של שערים לוגיים נעשה באמצעות טרנזיסטורים המשמשים כמתגים. לכל מתג 3 קצוות: כניסת בקרה (C) ושני קצוות (A,B) שהמתג יכול לחבר ביניהם.

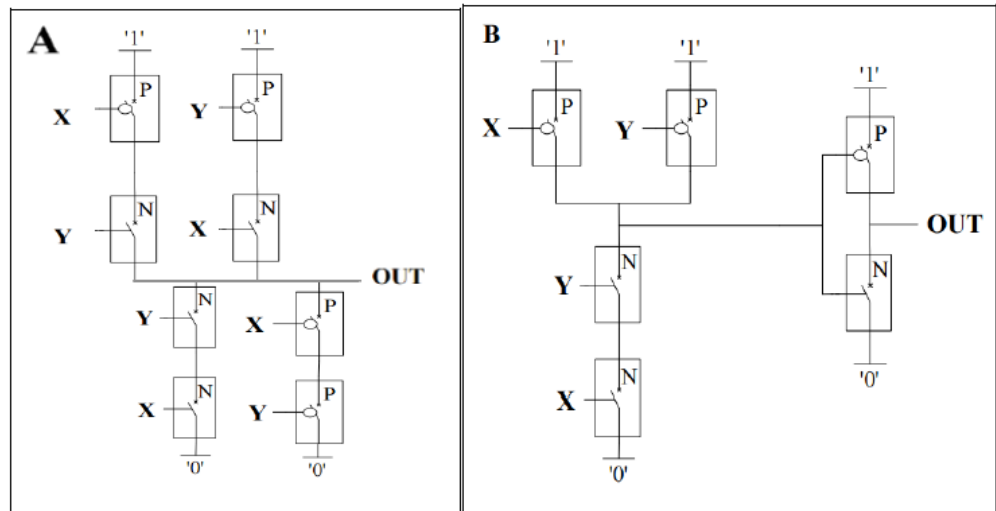
נגדיר שני סוגי מתגים: **מתג P** – כאשר $C=0$ המתג מחובר (עובר זרם), וכאשר $C=1$ המתג מנותק (לא עובר זרם), ו**מתג N** – בו המצב הפוך ($C=1$ – מתג מחובר, $C=0$ – מתג מנותק).



נראה לדוגמה מימוש של שער NAND בעזרת מתגים אלו, כאשר מתקיים: $Z = \text{NAND}(A,B)$

שאלה ממבחן – שאלה 7 מתוך מועד א' אביב 2017

נתונים שני הרכיבים הבאים:



נתונה הפונקציה $f(x,y) = x \cdot \bar{y}$

בחרו את התשובה הנכונה ביותר:

- ניתן לממש את f בעזרת רכיב A בלבד.
- ניתן לממש את f בעזרת רכיב B בלבד.
- ניתן לממש את f בעזרת הרכיבים A ו-B ביחד.
- שני הרכיבים $\{A, B\}$ **לא מהווים** מערכת פעולות שלמה ולכן לא ניתן לממש את f .
- שני הרכיבים $\{A, B\}$ **מהווים** מערכת פעולות שלמה ולכן ניתן לממש כל פונקציה בפרט f .

פתרון:

תחילה ננסה להבין מה כל אחד מהרכיבים ממש. נבנה טבלת אמת עבור כל אחד מהרכיבים:

רכיב A			רכיב B		
X	Y	Out	X	Y	Out
0	0	0	0	0	0
0	1	1	0	1	0
1	0	1	1	0	0
1	1	0	1	1	1

נשים לב כי רכיב A מייצג שער XOR ורכיב B מייצג שער AND.

כעת נעבור על האפשרויות:

א+ב – לא נכון, לא ניתן באמצעות XOR או AND בלבד לממש את הפונקציה הנתונה.

ג - ננסה לממש את f , כידוע $XOR(X, Y) = \bar{X}Y + X\bar{Y}$, ננסה כעת להגיע באמצעות AND לפונקציה

$$\text{הנתונה: } AND(\bar{X}Y + X\bar{Y}, X) = X\bar{X}Y + XX\bar{Y} = 0 + XX\bar{Y} = X\bar{Y} = f$$

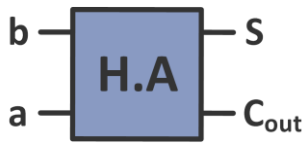
ד – התשובה לא נכונה כיוון שראינו בג' שניתן לממש באמצעותן את f .

ה – באמצעות XOR ו AND לא ניתן ליצור NOT, ולכן הללו לא מע' פעולות שלמה.

תשובה נכונה: ג'

תכנ לוגי

Half Adder



מחבר שני ביטים בודדים :

$$S = a + b$$

• Carry – נשא תוצאת החיבור

טבלת אמת :

a	b	C _{out}	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

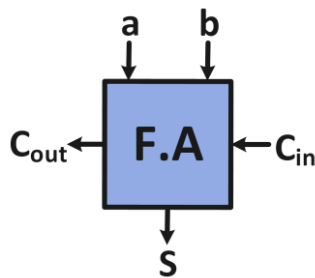
ביטוי :

$$S = a \oplus b$$

$$C_{out} = ab$$

Full Adder

מחבר שני ביטים בודדים, ולוקח בחשבון carry של פעולת חיבור קודמת :



$$S = a + b + C_{in}$$

• Cout – נשא תוצאת החיבור

טבלת אמת :

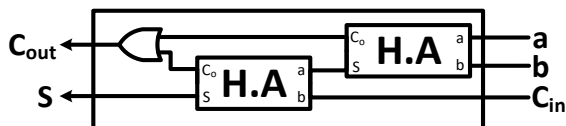
a	b	C _{in}	C _{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

ביטוי :

$$S = a \oplus b \oplus C_{in}$$

$$C_{out} = ab + ac_{in} + bc_{in}$$

$$C_{out} = ab + (a \oplus b)C_{in} \text{ מימוש נוסף}$$



ניתן גם לממש F.A ע"י שימוש ב-H.Aים, באופן הבא :

הוכחה כי Full Adder מהווה מערכת פעולות חצי שלמה :

$$S(a, 0, 1) = a \oplus 0 \oplus 1 = a \oplus 1 = \bar{a} \text{ ניתן לקבל NOT ע"י}$$

$$C_{out}(a, b, 0) = ab + 0 + 0 = ab \text{ ניתן לקבל AND ע"י}$$

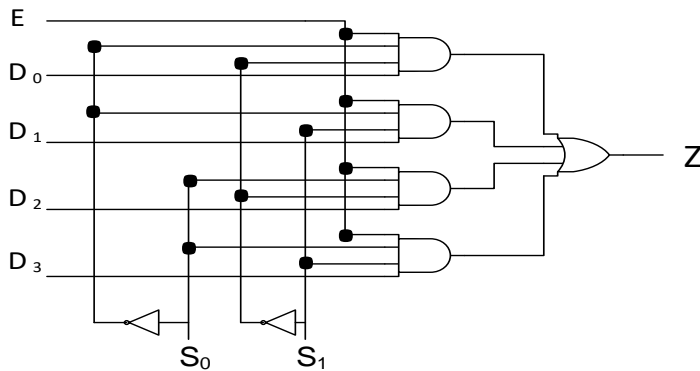
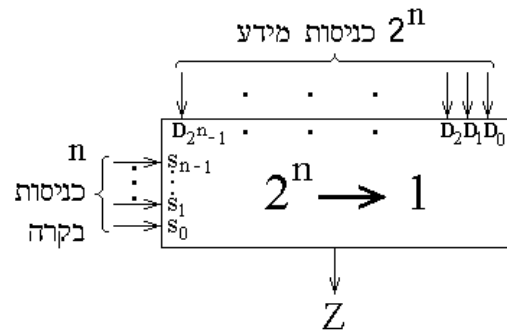
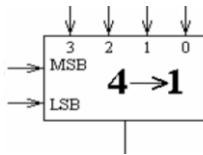
ע"י שרשור של F.Aים נוכל לבצע פעולות חיבור של מספר רב של סיביות. נעשה זאת ע"י חיבור C_{out} של

F.A אחד, ל-C_{in} של F.A ההבא בתור.

בורר – MUX

מעביר אחת מכניסותיו למוצא. בחירת הכניסה הרצויה תעשה ע"י כניסות בקרה:

למשל כאשר $n=2$:



מימוש אפשרי עבור בורר 4 -> 1 :
enable = E

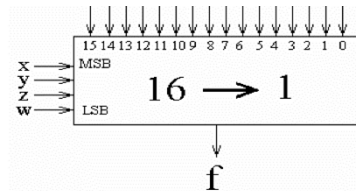
כאשר הביטוי הלוגי הוא:

(נשים לב שזהו למעשה סכום של מכפלת
ערך כל כניסה בבינארי, בכניסה עצמה)

$$z = E\bar{S}_0\bar{S}_1D_0 + E\bar{S}_0S_1D_1 + ES_0\bar{S}_1D_2 + ES_0S_1D_3$$

מימוש פונקציה של 4 משתנים בעזרת בוררים וקבועים

MSB	x	y	z	LSB	w	f
0	0	0	0	0	0	1
1	0	0	0	1	1	1
2	0	0	1	0	0	0
3	0	0	1	1	1	0
4	0	1	0	0	0	0
5	0	1	0	1	1	1
6	0	1	1	0	0	0
7	0	1	1	1	1	1
8	1	0	0	0	0	0
9	1	0	0	1	1	0
10	1	0	1	0	0	0
11	1	0	1	1	1	0
12	1	1	0	0	0	0
13	1	1	0	1	1	1
14	1	1	1	0	1	1
15	1	1	1	1	0	0



א. מימוש בעזרת בורר $16 \rightarrow 1$

ארבעת הביטים יחוברו לכניסות
הבקרה של הבורר, ובכל אחת
מכניסותיו יהיה קבוע '1' או '0' ע"פ
טבלת האמת של הפונקציה.
[כל פונקציה ניתן לממש בדרך זו].

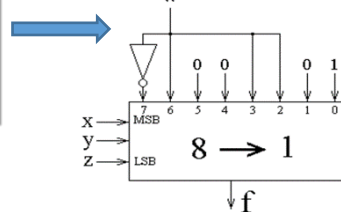
ב. מימוש בעזרת בורר $8 \rightarrow 1$ ושערים לוגיים

מצמידים כל 2 שורות אחת לשנייה, כך שבכל זוג יש 3 ביטים (שערכן גבוה יותר) זהים. ביטים אלו
יישארו כמו קודם ככניסות הבקרה של הבורר, והביט
הרביעי (השונה), נסמנו w, יכנס לכניסות הבקרה לפי
ערך הפונקציה, כמו
בדוגמה הבאה:

MSB	x	y	z	LSB	w	f
0	0	0	0	0	1	1
1	0	0	0	1	1	1
2	0	0	1	0	0	0
3	0	0	1	1	0	0
4	0	1	0	0	0	0
5	0	1	0	1	1	1
6	0	1	1	0	0	0
7	0	1	1	1	1	1
8	1	0	0	0	0	0
9	1	0	0	1	1	0
10	1	0	1	0	0	0
11	1	0	1	1	1	0
12	1	1	0	0	0	0
13	1	1	0	1	1	1
14	1	1	1	0	1	1
15	1	1	1	1	0	0

MSB	x	y	z	LSB	w	f
0	0	0	0	0	1	1
1	0	0	0	1	1	1
2	0	0	1	0	0	0
3	0	0	1	1	0	0
4	0	1	0	0	0	0
5	0	1	0	1	1	1
6	0	1	1	0	0	0
7	0	1	1	1	1	1
8	1	0	0	0	0	0
9	1	0	0	1	1	0
10	1	0	1	0	0	0
11	1	0	1	1	1	0
12	1	1	0	0	0	0
13	1	1	0	1	1	1
14	1	1	1	0	1	1
15	1	1	1	1	0	0

MSB	x	y	z	LSB	f
0	0	0	0	0	1
1	0	0	1	0	0
2	0	1	0	0	0
3	0	1	1	0	0
4	1	0	0	0	0
5	1	0	1	0	0
6	1	1	0	0	0
7	1	1	1	0	0



[במידה וברשותנו שער
NOT, ניתן לממש כל
פונקציה בדרך זו].

ג. מימוש בעזרת בורר $1 \rightarrow 4$ (מספר בוררים, או בורר ושערים לוגיים).

MSB	x	y	z	w	f
0	0	0	0	0	1
1	0	0	0	1	1
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	0
9	1	0	0	1	0
10	1	0	1	0	0
11	1	0	1	1	0
12	1	1	0	0	0
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	0

בדומה למקרה הקודם, כעת נצמיד כל 4 שורות בטבלת האמת, כך ששני הביטים שערכן גבוה יותר יהיו זהים. שני הביטים הללו יחוברו לכניסות הבקרה של הבורר התחתון, כך שכל רביעיית שורות תייצג כניסה אחת של בורר זה.

כעת יש שתי אפשרויות:

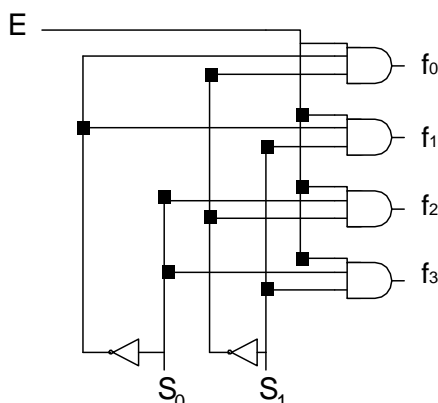
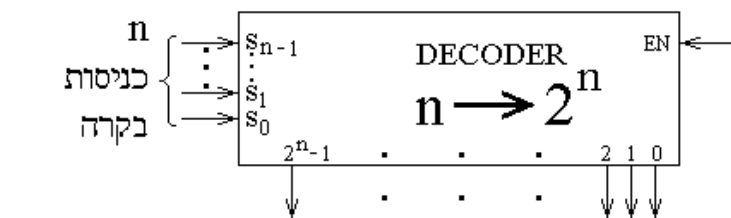
i. מימוש כל כניסה לבורר ע"י שער לוגי כלשהו או קבוע. כלומר, סה"כ שימוש בבורר אחד ושערים לוגיים וקבועים (במידה וברשותנו את כל השערים הלוגיים).

ii. מימוש כל כניסה לבורר ע"י בורר נוסף. כלומר עד 4 בוררים שכל אחד מחובר לכניסה של הבורר הנוכחי (כלומר סה"כ עד 5 בוררים). נשים לב שלמעשה כל שער לוגי ניתן לייצג ע"י בורר של $1 \rightarrow 4$, פשוט מעתיקים את טבלת האמת שלו לכניסותיו.

[כל פונקציה ניתן לממש בהינתן 5 בוררים, או בורר אחד וכל הקבועים והשערים הלוגיים. אולם, לא מעט פונקציות ניתן לממש גם בפחות מכך, וכאן צריך לבצע ניסוי ותהיה ולהבין איזה סידור של הביטים בטבלה יניב מס' מינימלי של בוררים/שערים לפי המלאי הנתון].

מפענח – Decoder

במפענח יש n כניסות בקרה ו- 2^n יציאות. כל היציאות יראו 0 חוץ מהיציאה ה- i , כאשר i הוא המספר שערכו מיוצג בבסיס בינארי ע"י כניסות הבקרה. גם כאן יכולה להיות כניסת $enable$, וכאשר $E=0$, כל היציאות המפענח יראו 0.



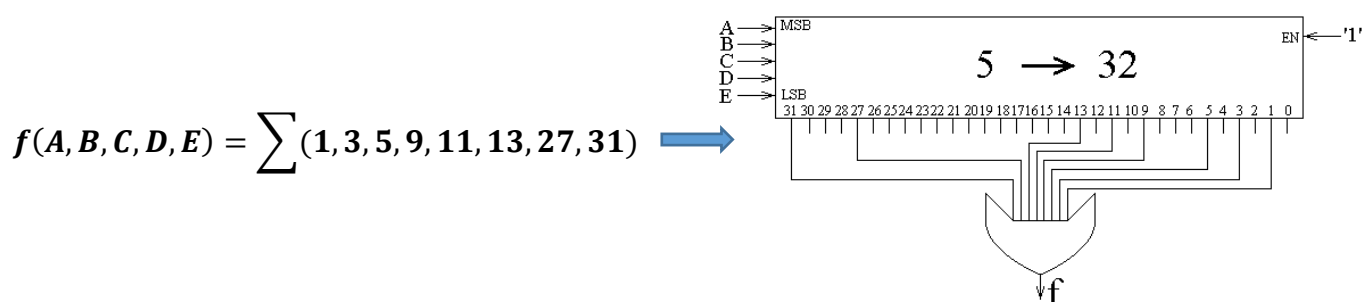
מימוש אפשרי של מפענח $2 \rightarrow 4$:

מימוש פונקציה בעזרת מפענחים

א. מימוש ישיר בעזרת מפענח בגודל טבלת האמת:

מחברים את כניסות הפונקציה לכניסות הבקרה של במפענח. את היציאות במספרן של כל השורות בהן הפונקציה מקבלת '1' בטבלת האמת, מחברים לשער OR אחד (את שאר היציאות לא מחברים לכלום). בכניסת enable שמם '1' קבוע.

לדוג': מימוש פונק' של 5 משתנים עם מפענח $32 \rightarrow 5$:



ב. מימוש באמצעות מפענח/ים קטנים יותר:

נסתכל על השורות בטלת האמת בהן הפונקציה מקבלת '1'. כעת נפצל לפי משתנה מסוים את הטבלה, כלומר חלק אחד יהיה כאשר אותו משתנה הוא '0', והחלק השני יהיה כאשר הוא '1'. קיבלנו למעשה 2 טבלאות אמת של $n-1$ משתנים, כעת נבנה שתי פונקציות חדשות לפי טבלאות האמת החדשות.

את המשתנה שבחרנו נכניס לכניסות enablen של כל אחד מהמפענחים, כאשר שלפונקציה שנבנתה על סמך טבלת האמת בו הוא י'ס' נחבר את את NOT של אותו משתנה, ולפונקציה האחרת נחבר את המשתנה עצמו. לדוג': מימוש אותה הפונקציה ממקודם עם 2 מפענחי $16 \rightarrow 4$:

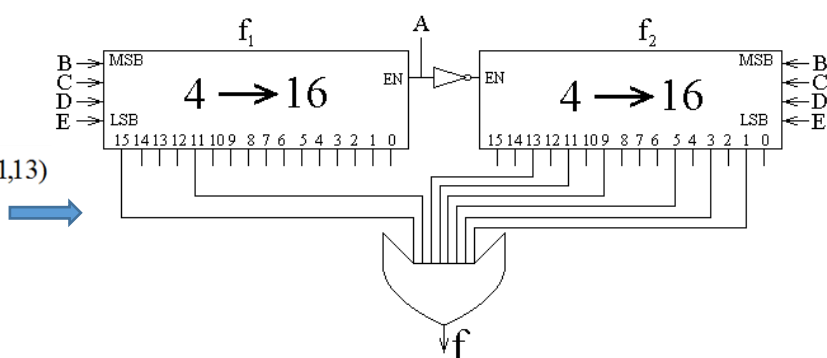
$$f(A,B,C,D,E) = A \cdot f_1(B,C,D,E) + A' \cdot f_2(B,C,D,E)$$

ABCDE	F
00001	1
00011	1
00101	1
01001	1
01011	1
01101	1
11011	1
11111	1

BCDE	F
0001	1
0011	1
0101	1
1001	1
1011	1
1101	1
1011	1
1111	1

$$f_2(B,C,D,E) = \sum (1,3,5,9,11,13)$$

$$f_1(B,C,D,E) = \sum (11,15)$$



לעיתים, אם נבחר משתנה אחר, נוכל להמעיט במספר המפענחים. למשל, אם ניקח את אותה הדוגמה אך

$$f(A,B,C,D,E) = E \cdot f_1(A,B,C,D) + E' \cdot f_2(A,B,C,D)$$

ABCDE	F
00001	1
00011	1
00101	1
01001	1
01011	1
01101	1
11011	1
11111	1

ABCD	F
0000	1
0001	1
0010	1
0100	1
0101	1
0110	1
1101	1
1111	1

$$= \sum (0,1,2,4,5,6,13,15)$$

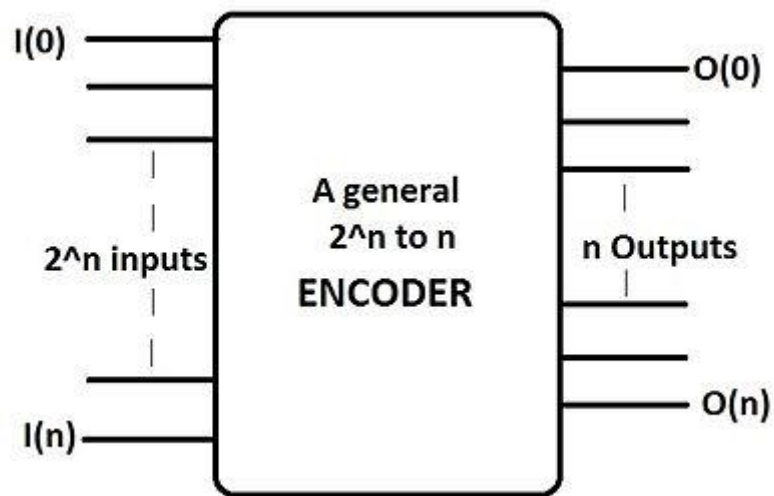
$$f_2 = 0$$

נבחר לפצל לפי משתנה E, נוכל
לממש את הפונקציה בעזרת מפתח
 $16 \rightarrow 4$ אחד בלבד, באופן הבא:

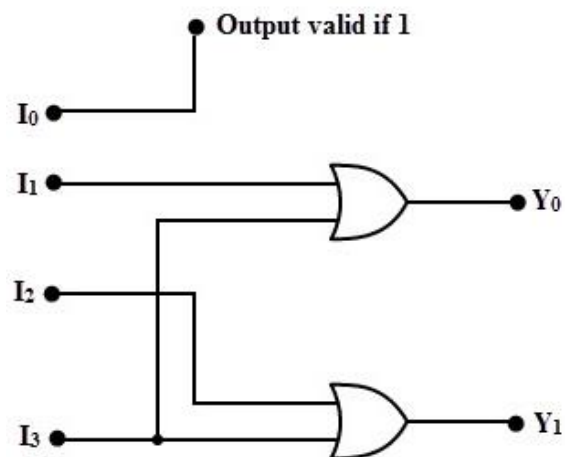
מקודד – Encoder

במקודד יש n כניסות ו $\log_2 n$ יציאות.

היציאה תהיה הקוד הבינארי של מספר הכניסה (היחידה) שערכה הוא '1'.



מימוש אפשרי למקודד $4 \rightarrow 2$:



מצבים בהם אין בדיוק כניסה אחת שערכה '1' יוגדרו כDon't Cares. יש מקודדים מסוימים, בהם למצבים כאלו מוגדר כי היציאה היא הקוד הבינארי של הכניסה הראשונה שערכה הוא '1'.

ניסוי לגילוי תקלות צירופיות

תהליך הייצור של מעגלים ספרתיים איננו אידיאלי – מידי פעם נופלות תקלות במעגלים. נרצה לבצע ניסוי שיגלה אם יש תקלות בדרך הקצרה והיעילה ביותר.

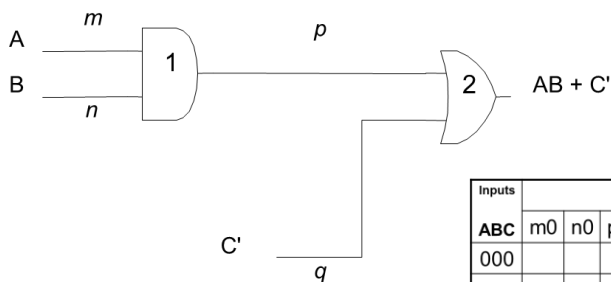
במקרה הפשוט של מעגל צירופי בעל n כניסות, נצטרך לייצר את כל 2^n המצבים האפשריים ואם צירוף כלשהו יהיה שונה מזה שמאור להתקבל (בטבלת האמת) נדע כי הרכיב תקול.

אולם, ניתן לעשות זאת בדרכים יעילות יותר.

במסגרת הקורס נגביל את הדיון לתקלה בודדת ברכיב מסוג של "צומת תקוע" (תקלה לפיה חוט מסוים במעגל תקוע במצב של '1' או '0').

1. נערוך טבלה ובה נעבור על כל המקרים האפשריים של צומת תקועה ונבחן איזה מקרה כזה יגרור

מוצא שגוי של הרכיב, לדוג':



נשים לב שכאשר בוחנים את הצירוף '000', אם

החוט q תקוע על '0' נקבל שגיאה כיוון שמוצא

הפונקציה הצפוי הוא '1', אך

בפועל נקבל '0', לכן נסמן את

המשבצת של q0 (כלומר החוט q

תקוע ב0) והצירוף 000.

Inputs ABC	Possible Faults							
	m0	n0	p0	q0	m1	n1	p1	q1
000				x				
001							x	x
010				x				
011					x		x	x
100				x				
101						x	x	x
110								
111	x	x	x					

2. נבדוק איזה מהבדיקות הן הכרחיות – נעבור על העמודות ונראה באיזו עמודה מסומן רק X אחד,

כלומר היכן הבדיקה הנוכחית היא היחידה שנוכל בעזרתה לעלות התקלה.

בדוגמה שלנו '011', '101', '111'.

3. נסמן גם שאר העמודות שנבדקות ע"י העמודות שסימנו. לדוג' במקרה שלנו סימנו את 011 כי זה

הצירוף היחיד שבודק את m1, אך הוא בודק "על הדרך" גם את p1 וq1.

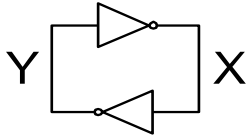
4. נעבור כעת על השורות ונבדוק איזה עמודות לא מסומנות ע"י אף שורה, ונסמן מינימום שורות כך

שכל העמודות יכוסו, לדוג' במקרה שלנו q0 לא כוסתה עם אף שורה שסימנו בשלבים הקודמים,

ולכן נסמן שורה כלשהי המכסה גם אותה (למשל 000/010/100).

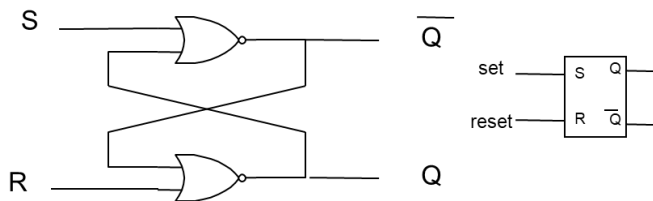
נשים לב כי ע"י שימוש בשיטה זו, בדוגמה שלנו מצאנו כי מספיק לבצע 4 בדיקות במקום 8.

רכיבי זיכרון וחישוב זמנים



נשאלת השאלה כיצד ניתן לשמור ערך לוגי מסוים לאורך זמן. נשים לב שע"י חיבור שני מהפכים אחד לשני באופן הבא :
נוכל לקבוע מצב יציב, למשל $x=1$ ו $y=0$ (או ההפך). הבעיה היא שלא נוכל לקרוא ערך זה.

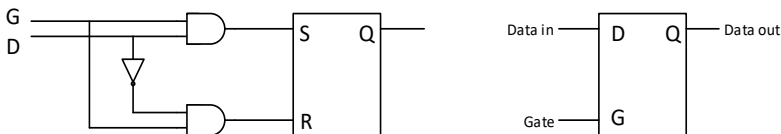
SR Latch



פתרון אפשרי לבעיה שהוצגה מקודם היא להשתמש במקום בשערי NOT בשערי NOR באופן הבא :
כאשר $S=R=0$ המעגל שקול לשני מהפכים.
רכיב זה קרוי Latch ("מנעול").

הכנסת הערכים $S=1, R=0$, מכונה מצב Set והיא תשמור את הערך '1' ברכיב. כל עוד $R=0$, הערך ישמר.
הכנסת הערכים $S=0, R=1$, מכונה מצב Reset והיא תשמור את הערך '0' ברכיב.
הכנסת הערכים $0,0$ תגרום לבלבול היציאות (שתייהן יראו 0) והכנסת $1,1$ תגרום למרוץ. נדאג לא להכניס ערכים אלו.

D Latch

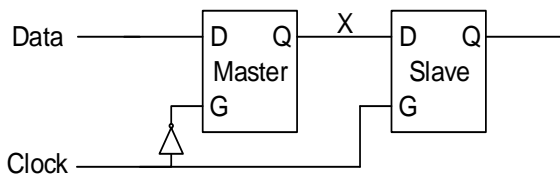


ע"י שינוי קטן ברכיב שתארנו (ראה איור) ניתן להשתמש בו לאחסון סיבית מידע (נסמן D) בודדת.
מקובל להכניס בכניסת Gaten, אות

מחזורי הנקרא שעון (Clock). נדרוש שכאשר כניסה אחת משתנה, השנייה תישאר יציבה : כאשר D משתנה, $G=0$. תנאים אלו מבטיחים שמצב $S=R=1$ לא יהיה אפשרי.

DFF או D Flip Flop

במקום שהשינוי יתאפשר רק כשהשעון ברמה גבוהה, נאפשר את שינוי היציאה רק כתוצאה מעליית השעון.



נעשה זאת ע"י חיבור שני רכיבי Latch. כאשר השעון $Clk=0$, ה-latch הראשון 'שקוף' וכניסת סיבית המידע עוברת לצומת הפנימית (המסומנת ב-X). כאשר השעון עולה ל-1, ה-latch הראשון נסגר ומכאן והלאה הוא "זוכר"

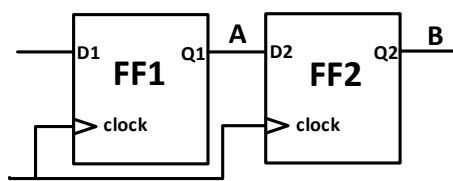
את ערך סיבית המידע האחרונה, וה-latch השני נפתח והערך שב-X עובר ליציאה.

מאוחר יותר (לאחר עוד מחצית מחזור השעון) יורד השעון שוב ל-0, ואז קורים שני הדברים הבאים :
ה-latch השני נסגר ומכאן והלאה הוא "זוכר" את ערכו האחרון של X וה-latch הראשון נפתח ומאפשר לכניסה Data לקבוע את ערכו הבא של X.

אפיון הזמנים של DFF

- t_{set-up} – משך הזמן לפני עליית השעון שבו המידע חייב להיות יציב.
- t_{hold} – משך הזמן לאחר עליית השעון שבו המידע חייב להמשיך להיות יציב.
- t_{cc-Q} – זהו למעשה ה" t_{cd} " של DFF. זהו הזמן מרגע עליית השעון עד לזמן בו בוודאות לא ישתנה המתח ביציאה Q (החסם התחתון).
- t_{pc-Q} – זהו למעשה ה" t_{pd} " של DFF. זהו הזמן מרגע עליית השעון עד לזמן בו בוודאות יתייצב המתח ביציאה Q על ערכו החדש (החסם העליון).

הערה: לשים לב ששני הזמנים האחרונים נמדדים החל מעליית השעון ולא החל משינוי המידע.



חיבור DFFים בטור - תזמון

תנאי Hold – בחיבור DFFים בטור, נדרוש שהזמן המינימלי בו הפליפ-פלופ הראשון משנה את ערכו (כלומר ה t_{cc-Q} שלו)

יהיה גדול מזמן ההשהיה (t_{hold}) של הפליפ-פלופ השני, על מנת שהוא לא ידגום בטעות את המידע החדש.

$$\text{כלומר, נדרוש: } t_{cc-Q}(FF1) \geq t_h(FF2)$$

תנאי Set-Up – בחיבור DFFים בטור, נדרוש שהמוצא של הפליפ-פלופ הראשון ייטיב לכל המאוחר לפני תחילת זמן ההמתנה הדרוש של הפליפ-פלופ השני. כלומר נדרוש שה t_{pc-Q} (מקסימום זמן להתייצבות

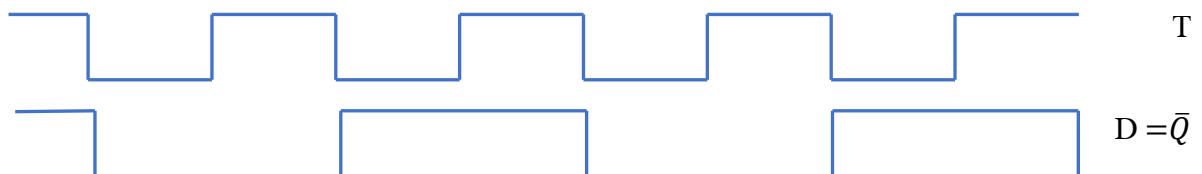
הערך החדש) של הראשון וה t_{set-up} של השני, יכנסו שניהם בזמן מחזור אחד.

$$\text{כלומר, נדרוש: } t_{pc-Q}(FF1) + t_{su}(FF2) \leq T$$

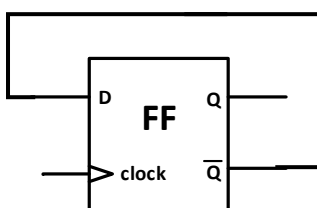
(כאשר T מסמן את זמן המחזור של השעון, לעיתים גם מסומן כ T_{clk} או T_{cy}).

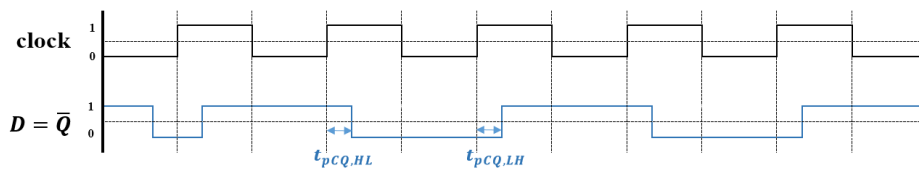
שימושים של DFF בטור

מחלק תדר – כדי לבנות מחלק תדר נחבר את היציאה \bar{Q} לכניסת המידע D, נשים לב שהאות שנקבל הוא האות הבא:



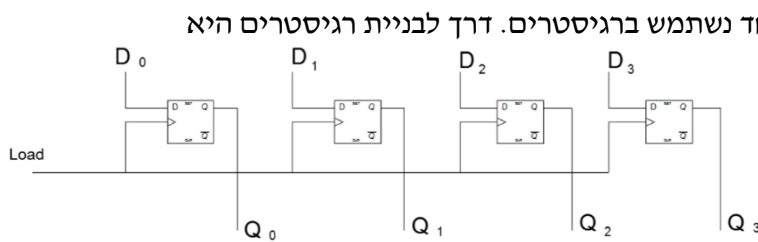
כלומר, קיבלנו אות מחזורי בעל מחזור הארוך פי 2 ממחזור אות השעון המקורי – לכן התדר החדש הוא חצי מהתדר הישן.





בפועל יש להתחשב גם כאן במשטרי התזמון - חיבור DFF לעצמו שקול לחיבור שני DFFים בטור, והתדר האמיתי יראה כך:

אם נשים לב כי הדבר לא עומד במשטר התזמון נוכח להוסיף מהפכים בחוט המחובר את \bar{Q} לכניסת המידע D. נקפיד לשים מספר זוגי של שערי NOT כדי לא לשנות את הלוגיקה. (הוספת שני שערי NOT שקולה להוספת Buffer – לא משפיע על הלוגיקה, אך יש לו t_{pd}).



רגיסטרים – כאשר רוצים לזכור יותר מביט אחד נשתמש ברגיסטרים. דרך לבניית רגיסטרים היא באמצעות DFFים. נחבר את כל הDFFים לאותו אות שעון כך שידגמו באותו הזמן את הערכים החדשים, וכך נוכל לשמור מספר בינארי בעל מספר רב של ביטים.



חיבור DFFים ולוגיקה צירופית

במקרה כזה נצטרך להתחשב גם בזמני ההשהיה של הלוגיקה הצירופית.

לכן, נצטרך "לתקן" את תנאי התזמון שניסחנו מקודם על מנת שיתאימו למקרה זה:

תנאי Hold – כעת למעשה נוסף עוד זמן השהיה בו עדיין FF2 מקבל את הערך הישן, ניתן אינטואיטיבית לחשוב כי "עזרנו" לתנאי Hold להתקיים והפכנו אותו ל"קל" יותר.

$$t_{cc-Q}(FF1) + t_{cd}(logic) \geq t_h(FF2) \quad \text{התנאי המתוקן:}$$

תנאי Set-Up – כעת הוספנו עוד זמן שצריך להיכנס בתוך זמן מחזור אחד. ניתן אינטואיטיבית לחשוב כי הפכנו את תנאי Setup ל"קשה" יותר.

$$t_{pc-Q}(FF1) + t_{pd}(logic) + t_{su}(FF2) \leq T \quad \text{התנאי המתוקן:}$$

הערה: ההפרש בין האגפים באי השוויוניים (כלומר, בתנאי הולד: הזמן מסיום t_h עד לסיום $t_{cc-Q} + t_{cd}$ ובתנאי סט-אפ: הזמן מסוף t_{pd} של הלוגיקה הצירופית עד לתחילת t_{su} של הפליפ-פלופ השני) מכונה t_{slack} . אם t_{slack} חיובי, אז המעגל תקין. לעומת זאת, אם t_{slack} שלילי, התנאי לא מתקיים וצריך לפעול בדרך כלשהי כדי לתקן את הבעייתיות. (למשל הוספת buffer ע"מ להגדיל זמן שיהוי של לוגיקה צירופית, הגדלת זמן מחזור אם אפשרי, הוספת skew לאחד הDFFים).

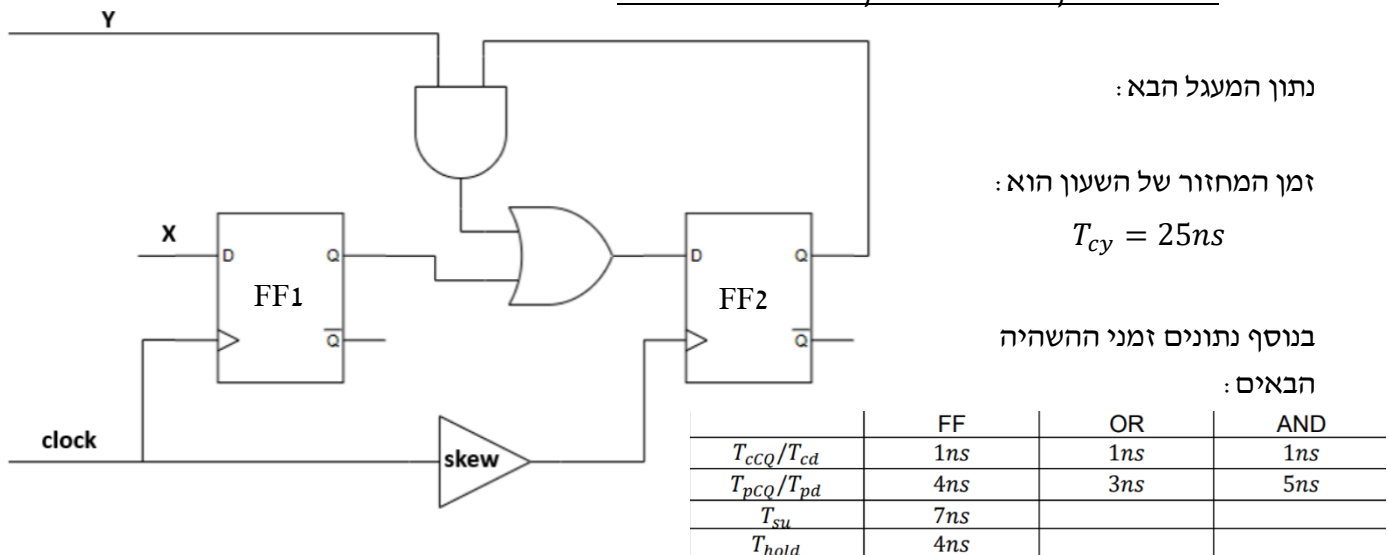
Clock Skew

תופעה (הנגרמת באופן רצוי, או שלא) בה אות השעון מגיע בזמן שונה לDFFים שונים באותו מעגל. נסמן את ההפרש בהגעת השעון ב t_{skew} או ב t_k . זמן זה יכול להיות חיובי או שלילי (תלוי בנתוני השאלה).

איך נפתור שאלות בתזמון?

נאתר את כל המסלולים שמתחילים ונגמרים ב DFF ונבדוק את תקינות תנאי $Hold$ ו $Setup$. במידה ויש $Skew$ כשלהו, מומלץ לצייר את אות השעון השונים, ולצייר מעין תרשים של כל הזמנים המופיעים באי השוויונים – באופן זה ניתן לקבוע בקלות לאיזה אגף באי השוויון צריך להוסיף את ה t_{skew} , מבלי להתעסק בסימני +/- (דבר שיכול להיות מבלבל).

שאלה ממבחן – שאלה 2 מתוך מועד א' אביב 2020



נתון כי הכניסות עומדות בתנאי $hold$ ו su . בין שני ה DFFים קיים skew בשעון שערכו הוא t_{skew} .

א. עבור $t_{skew} = 0$, האם המעגל עומד במשטר הזמנים הדינאמי?

ב. מבין הערכים הבאים, מהו הערך של t_{skew} עבורו המעגל עומד במשטר הזמנים הדינאמי?

1. -2ns

2. 3ns

3. 0ns

4. אף ערך

5. 1ns

ג. כעת נתון כי $t_{skew} = 1ns$. על מנת לאפשר פעילות תקינה של המעגל, הוחלט לבצע שימוש ב $buffer$

עבורו מתקיים כי $t_{cd} = t_{pd}$. הוסיפו את $buffer$ במקום המתאים בשרטוט וקבעו את זמן ההשהיה

המינימלי והמקסימלי של החוצץ המאפשרים עמידה במשטר הזמנים הדינאמי.

פתרון:

א. נסתכל על המסלול שבין FF2 לעצמו. נבדוק אם תנאי Hold מתקיים:

$$t_{cc-q}(FF2) + t_{cd}(AND) + t_{cd}(OR) \geq t_h(FF2)$$

$$1 + 1 + 1 \geq 4$$

קיבלנו t_{slack} שלילי ולכן תנאי hold לא מתקיים. המעגל לא עומד במשטר הזמנים הדינאמי.

ב. אף ערך לא יהיה מתאים כיוון שבמסלול $DFF2 \rightarrow DFF2$ ה skew לא משפיע על תנאי hold.

ג. נוסיף את ה buffer לפני הכניסה ל FF2.

נבדוק כל אחד מהמסלולים המתחילים והנגמרים ב FF2 ונבדוק תקינות של תנאי su ותנאי hold:

FF1 → FF2:

תנאי hold:

$$t_{cc-q}(FF1) + t_{cd}(OR) + t_{buffer} \geq t_h(FF2) + t_{skew}$$

$$1 + 1 + t_{buffer} \geq 4 + 1$$

$$t_{buffer} \geq 3$$

תנאי setup:

$$t_{pc-q}(FF1) + t_{pd}(OR) + t_{buffer} + t_{su}(FF2) \leq T + t_{skew}$$

$$4 + 3 + t_{buffer} + 7 \leq 25 + 1$$

$$t_{buffer} \leq 12$$

FF2 → FF2:

תנאי hold:

$$t_{cc-q}(FF2) + t_{cd}(AND) + t_{cd}(OR) + t_{buffer} \geq t_h(FF2)$$

$$1 + 1 + 1 + t_{buffer} \geq 4$$

$$t_{buffer} \geq 1$$

תנאי setup:

$$t_{pc-q}(FF2) + t_{pd}(AND) + t_{pd}(OR) + t_{buffer} + t_{su}(FF2) \leq T$$

$$4 + 5 + 3 + t_{buffer} + 7 \leq 25$$

$$t_{buffer} \leq 6$$

ניקח את מקרי הקיצון ונקבל לסיכום:

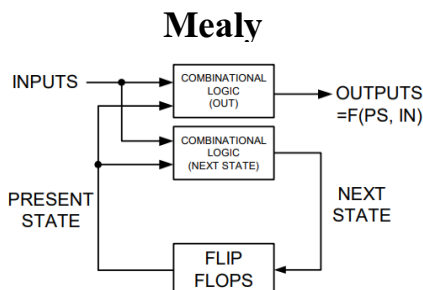
$$3 \leq t_{buffer} \leq 6$$

מערכות סינכרוניות

עד כה עסקנו במערכות צירופיות, בהן ערכי הפלט תלויים אך ורק בערכים הנוכחיים של משתני הקלט. כעת נעסוק במכונות מצבים, המורכבות ממערכת צירופית + זיכרון הזוכר "מצב". ערכי הפלט יהיו תלויים בערכי הקלט ובמצב הקיים.

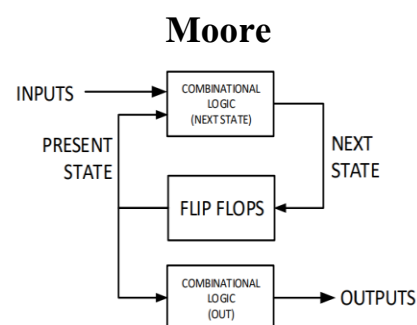
ערכי המצב החדש גם הם תלויים בערכי בקלט ובמצב הקיים.

נבדיל בין שני סוגים של מכונות מצבים:



• מכונת מילי Mealy

הפלט, והמצב הבא, תלויים גם בקלט וגם במצב הנוכחי.



• מכונת מור Moore

הפלט תלוי במצב הנוכחי בלבד.
המצב הבא תלוי בקלט ובמצב הנוכחי

השלבים לביצוע סינתזה של מערכות סינכרוניות

1. דיאגרמת מצבים – נתרגם את הבעיה המילולית שברשותנו לדיאגרמת מצבים, כך שמכל מצב יוצאים חיצים עליהן כל כניסות הקלט האפשריות (לדוג' במערכת עם כניסה ברוחב 2 ביטים, מכל מצב יצאו $2^2 = 4$ חצים).
על כל חץ יופיע הקלט אליו הוא מתייחס והפלט של המערכת במקרה של מכונת מילי.
במקרה של מכונת מור נכתוב את הפלט בתוך המצב.
2. טבלת מצבים (+צמצום) – נתרגם את הדיאגרמה לטבלה ששורותיה הן המצבים ועמודותיה הן כל הקומבינציות של קלטים אפשריים. במפגש כל שורה ועמודה נכתוב מהו המצב הבא הרצוי בהינתן קלט+מצב נוכחי זה.
3. הקצאת מצבים – מתן קוד לכל מצב – ניתן לכל קוד מספר המייצג אותו. בשביל n מצבים נצטרך $\lceil \log_2 n \rceil$ ביטים לייצוג הקוד (וגם מספר זה של FFים).
4. טבלת מעברים ופלט (הצבת הקודים בטבלת המצבים) – נחזור לטבלת המצבים ובמקום כל שם של מצב נכתוב את הקוד המתאים לו (נקפיד להיות קונסיסטנטיים).
5. מציאת הפונקציות המתארות את כניסות ה FF והיציאה (או היציאות) Z של המערכת.
נעשה זאת כמו שמצאנו פונקציות צירופיות רגילות. נצטרך למצוא ביטוי כל אחת מהביטים של המצב הבא.
6. שרטוט המעגל
לדוגמה, אם יש לנו מכונת מצבים עם כניסה ברוחב ביט אחד, נסמנה X , 71 מצבים. נצטרך לטובת המימוש $\lceil \log_2 71 \rceil = 3$ ביטים שיתארו את המצבים (וגם 3 FFים). נסמן את קוד המצב הבא ב: $D_0 D_1 D_2$ ואת המצב הנוכחי ב $Q_0 Q_1 Q_2$ ואת מוצא המערכת ב Z . נצטרך למצוא את 4 הפונקציות הבאות: $D_0(Q_0, Q_1, Q_2, X)$, $D_1(Q_0, Q_1, Q_2, X)$, $D_2(Q_0, Q_1, Q_2, X)$, $Z(Q_0, Q_1, Q_2, X)$.

ארבע דרכים להמיר מכונת Mealy למכונת Moore

1. תכנון מחדש כמכונת Moore
2. הוספת רגיסטרים בכניסה
3. הוספת רגיסטרים ביציאה
4. הוספת מצבים לטבלת המצבים היכן שהמוצא אינו זהה לכל הכניסות

צמצום מצבים – הגדרות:

- מצבים בני הפרדה – נסתכל על מצבים מסוימים, נתחיל כל פעם ממצב אחר, ונכניס סדרה של קלט.
אם לאחר הכנסת סדרת הקלט, נקבל סדרת פלט שונה (כאשר התחלנו מהמצב האחר) נאמר שהמצבים הם בני הפרדה.
אם נדרשה סדרה של k ביטים ע"מ לקבל מוצא שונה, נאמר שהמצבים הם k -בני הפרדה.
- מצבים שקולים – אם לכל סדרה שנכניס (כאשר בכל פעם מתחילים ממצב אחר) נקבל בדיוק את אותו המוצא, נאמר שהמצבים (מהם התחלנו) הם מצבים שקולים.
נגיד גם ששני מצבים יהיו שקולים אם הם אינם בני הפרדה.
מצבים מכונים k שקולים אן הם אינם k בני הפרדה (כלומר אין אף סדרה באורך k היכולה להפריד ביניהם).
- מחלקות שקילות – תתי קבוצות של כל המצבים כך שכל חברי מחלקה שקולים זה לזה ואינם שקולים לאף חבר של קבוצה אחרת.
- מצב עוקב –
 - **מצב 0-עוקב** של מצב כלשהו זה המצב שעוברים אליו ממצב זה בגין כניסה 0
 - **מצב 1-עוקב** של מצב כלשהו זה המצב שעוברים אליו ממצב זה בגין כניסה 1
 - **מצב X-עוקב** של מצב כלשהו זה המצב שעוברים אליו ממצב זה בגין כניסה של הסדרה X

האלגוריתם של Moore לצמצום מכונה

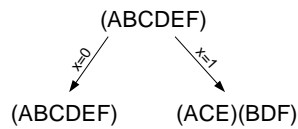
- נפתח בקבוצת כל המצבים (ניתן למעשה לומר שכל המצבים הם 0-שקולים).

- בשלב הראשון נבחין בין מצבים שונים לפי הפלט – נבדוק עבור אילו כניסות נקבל פלט שונה.

PS	x=0	x=1
A	E,0	D,1
B	F,0	D,0
C	E,0	B,1
D	F,0	B,0
E	C,0	F,1
F	B,0	C,0

לדוגמה, בטבלה הבאה נשים לב שעבור $X=1$, המצבים ACE יתנו ביציאה 1, והמצבים BDF יתנו 0.

לכן נוכל לפצל את ACE ו-BDF למחלקות שקילות שונות, ונאמר שהם 1-בני הפרדה, כלומר סדרה באורך 1 הצליחה להפריד ביניהם (הצלחנו לקבל מוצא שונה במערכת לאחר שהכנסנו ביט אחד).



מספר מחלקות השקילות שניתן לקבל בשלב זה

תלוי ברוחב הכניסה ורוחב היציאה – במידה ומדובר למשל בכניסה ברוחב 1

(ניתן בכל פעם להכניס '0' או '1') ויציאה ברוחב 1 (המכונה מוציאה '0' או '1'), אז נוכל לאחר שלב

זה לקבל 4 מחלקות שקילות שונות. או באופן כללי, אם הכניסה ברוחב m והיציאה ברוחב n,

נוכל לקבל בשלב זה $2^{m \cdot 2^n}$ מחלקות שקילות לכל היותר.

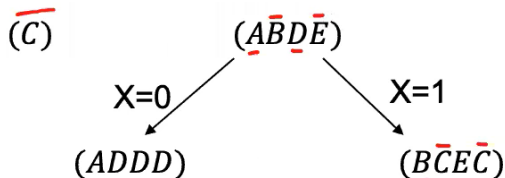
במידה ומדובר במכונת Moore, יש לזכור כי המצב הבא אינו מושפע מהכניסה, אלא רק מהמצב הנוכחי, לכן פשוט נסתכל על היציאה נראה עבור איזה מצבים נקבל יציאה שונה – ואותם נשים במחלקות נפרדות.

- כעת, נסתכל לאן מוביל אותנו כל מצב (במקרה של מכונת Mealy, נבדוק עבור כל קלט). אם

"סט" מסוים של מצבים, מוביל אותנו למצבים הנמצאים במצבים הנמצאים ב"סטים" שונים

ברמה הנוכחית – נפריד אותם למחלקות שקילות שונות.

למשל בדוגמה הבאה:



נשים לב כי AD הובילו אותנו לBE (הנמצאים באותו סט)

אך BE הובילו אותנו למצב C הנמצא בסט נפרד ברמה

הנוכחית, ולכן נפריד את AD מBE (כי למעשה באמצעות

הכנסת $1=X$ נוכל להביא אותם למצב הנמצא במחלקת שקילות אחרת).

- ממשיכים בביצוע השלב האחרון עד שמגיעים למצב בו מחלקת שקילות זהה לזו שלפניה, ואז

סיימנו.

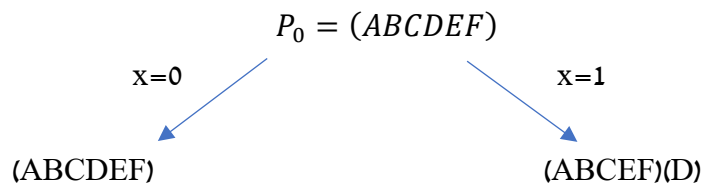
שאלה ממבחן – שאלה 8 מתוך מועד א' אביב 2020

נתונה טבלת המעברים של מערכת עקיבה בעלת כניסה אחת X ויציאה אחת Z :

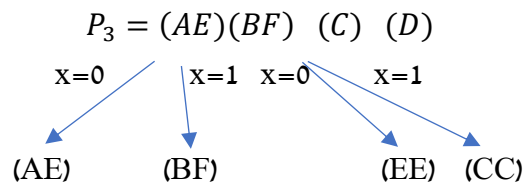
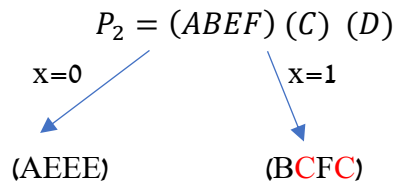
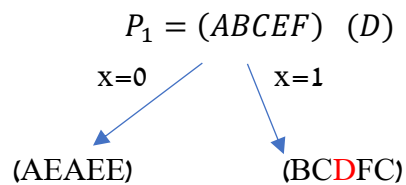
Present State	X=0		X=1	
	Next state	Z	Next state	Z
A	A	0	B	0
B	E	0	C	0
C	A	0	D	0
D	A	0	D	1
E	E	0	F	0
F	E	0	C	0

מהן מחלקות השקילות אשר מתקבלות מצמצום מכונת המצבים הנתונה?

פתרון: זוהי מכונת מילי, נתחיל בשלב הראשון לצמצם לפי הקלט :



נמשיך לפי המצב הבא :



$$P_4 = (AE)(BF) (C)(D)$$

$$P_3 = P_4$$

ולכן התשובה היא : $(AE)(BF) (C)(D)$

Pipeline

מדדים לביצועי מערכת ספרתית

- Latency – עבור פעולה אחת, זהו משך הזמן מכניסת הקלט ועד למוצא הפלט
- Throughput – מספר התפוקות שהמערכת מייצרת ביחידת זמן.

בכל המערכות שראינו עד כה, מתקיים: $Throughput = \frac{1}{Latency}$. בעזרת שיטת ה pipeline ('צינור') ניתן להגדיל את ה Throughput.

במערכת מצונרת, ה Throughput מייצג כל כמה זמן נקבל תוצאה חדשה, ומתקיים $Throughput = \frac{1}{T}$, כאשר T הוא זמן המחזור של המערכת.

צינור מדרגה K – K-pipeline – מעגל הבנוי מלוגיקה צירופית ורגיסטרים, כך שכל מסלול מכל כניסה ליציאה עובר דרך K רגיסטרים בדיוק. הערה: תמיד נמקם רגיסטרים בכניסה או ביציאה לצורך סנכרון עם מערכות אחרות (בד"כ ביציאה).

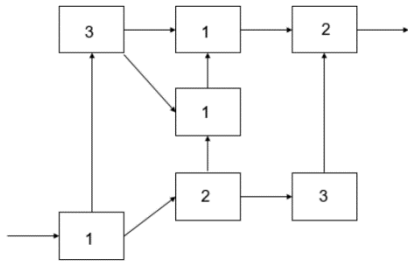
צינור וחישוב ביצועים עבור המערכת מצונרת

1. זיהוי היחידה הצירופית בעלה ה t_{pd} המקסימלי
2. לפי יחידה זו נקבע את זמן המחזור המינימלי (זהו למעשה ה t_{pd} המקסימלי).
כאשר ה FFים לא אידאליים יש כמובן גם להוסיף $t_{su}(FF2)$ ו $t_{pc-q}(FF1)$, כלומר:
3. נקבע את K – דרגת הצינור (דרך כמה רגיסטרים יעבור כל מסלול מהכניסה אל היציאה).
4. חישוב ביצועים ע"י:

$$Throughput = \frac{1}{T} \quad Latency = k \cdot T$$

עקרונות לצינור נכון:

- נצייר תמיד את המערכת מחדש כך שכל החיצים פונים לאותו כיוון – אסור להעביר קו רגיסטרים שהחיצים חוצים אותו בכיוונים מנוגדים!
 - נעביר תמיד קשת (קו רגיסטרים) ביציאה מהמערכת
 - נבודד את היחידה הצירופית בעלת ה t_{pd} המקסימלי בעזרת רגיסטרים
 - "נשבור" רצפים של יחידות עם t_{pd} ארוך יותר מזה של היחידה שבודדנו בשלב הקודם
- # שיטת הגומיות: נקבע 2 נקודות מעל ומתחת המערכת, ו"נמתח" קווי רגיסטרים ביניהם.
- # לאחר הסיום נוודא כי אכן בכל מסלול שנעביר בין כל כניסה לכל יציאה, נעבור בדיוק דרך K רגיסטרים. אם לא, אז יש טעות בצינור.



שאלה ממבחן – שאלה 11 מתוך מועד א' אביב 2020

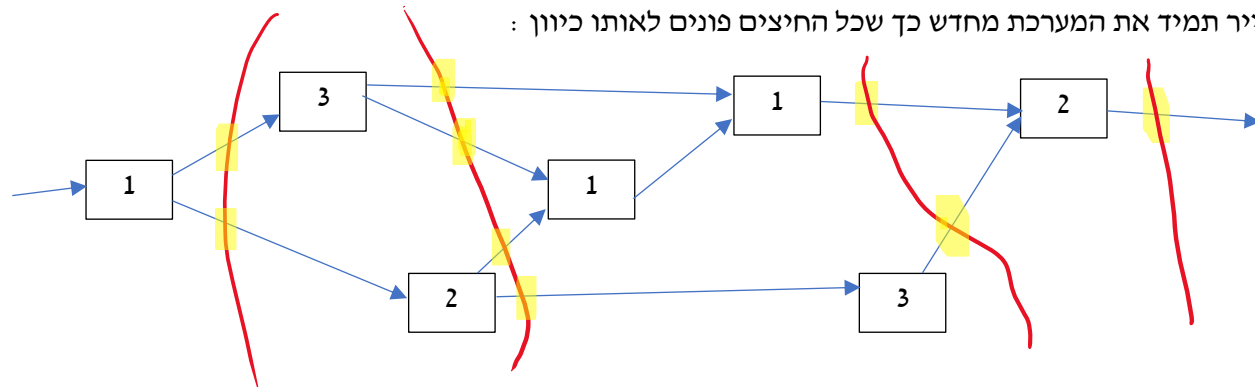
נתונה המערכת הבאה:

(זמן ההשהיה של כל רכיב כתוב בתוכו ונתון ns).

נרצה לצנר את המערכת בעזרת רגיסטרים אידיאליים על מנת לקבל Throughput מקסימלי בעדיפות ראשונה, ומספר רגיסטרים מינימלי בעדיפות שניה. מהו מספר הרגיסטרים המינימלי אשר דרוש לצורך צינור המערכת על פי סדר עדיפויות זה?

פתרון

- נצייר תמיד את המערכת מחדש כך שכל החיצים פונים לאותו כיוון:



- נעביר תמיד קשת (קו רגיסטרים) ביציאה מהמערכת
- נבודד את היחידה הצירופית בעלת ה t_{pd} המקסימלי בעזרת רגיסטרים
- "נשבור" רצפים של יחידות עם t_{pd} ארוך יותר מזה של היחידה שבודדנו בשלב הקודם

קיבלנו כי צריך בסה"כ 9 רגיסטרים.

תקשורת

פרוטוקול UART

זהו פרוטוקול תקשורת על חוט יחיד.

- במצב בו לא רוצים לשלוח מידע החוט נמצא ב **IDLE state** - על '1' לוגי קבוע.
- כשרוצים להתחיל לשלוח מידע שולחים **start-bit** שערכו מוגדר להיות '0' לוגי.
- לאחר מכן שולחים את סיביות המידע.
- כשמסיימים לשלוח את המידע שולחים **stop-bit** שערכו מוגדר להיות '0' לוגי.

זמן השידור של ביט אחד נקרא t_{bit} .

ה Transmitter וה Receiver צריכים "להסכים" על משך ה t_{bit} כך שיתאים לקצב השידור של כל אחד מהם:

$$N_T \cdot Tcycle(T_X) = N_R \cdot Tcycle(R_X) = T_{bit}$$

- N_T - מספר מחזורי השעון שה Transmitter צריך לשדר את אותו הביט $Tcycle(T_X)$ - זמן המחזור של Transmitter
- N_R - מספר מחזורי השעון שה Receiver דוגם את אותו הביט $Tcycle(T_X)$ - זמן המחזור של ה Receiver

סטייה בין המקלט והמשדר

כאשר ה t_{bit} של המשדר והמקלט שונים, נוצרת סטייה הנגררת לאורך שידור המידע. מספר הביטים המקסימלי שניתן לשדר יהיה כזה שדגימת המקלט, (לאחר כל הסטייה הנגררת) תהיה עדיין לפני סיום שידור הביט האחרון.

לדוגמה, אם עבור המשדר $t_{bit} = 400ns$ ועבור המקלט $t_{bit} = 408ns$, נוצרת סטייה של 8ns בשידור של כל ביט. אם נרצה לבדוק מהו מספר הביטים המקסימלי שניתן לשדר, נחשב מתי הסטייה הנגררת תעבור זמן של מחצית t_{bit} של המשדר, כלומר תעבור את 200ns. נסמן את מספר הביטים המקסימלי ב X . יש לזכור כי המקלט מתחיל לדגום את הביט הראשון רק לאחר $1.5t_{bit}$, ולכן נוסיף לסטייה הנגררת המחושבת סטייה של חצי ביט נוסף, ובמקרה זה 4ns:

$$4 + 8x \leq 200$$

$$8x \leq 196$$

$$x \leq 24.5$$

כמובן שנעגל כלפי מטה (אם נעגל כלפי מעלה כבר נעבור את הסטייה המותרת) ונקבל שבמקרה זה מספר הביטים המקסימלי המותר לשידור הוא 24. אם לא נספור את ה bit-stop כאחד מהם, אז התשובה תהיה כי מספר הביטים המקסימלי הוא 23 ביטים.

שאלה ממבחן – שאלה 9 מתוך מועד א' אביב 2020

התקשורת בין חיפה לתל אביב מתבססת על פרוטוקול ה-UART הבסיסי כפי שנלמד בכיתה (בכל שידור נשלחות 8 סיביות מידע, סיבית $start$ וסיבית $stop$).

על הקו נשלחות מילים באורך 64 סיביות. קצב שידור המילים הוא $f_{words} = 2000 \frac{words}{sec}$.

לאור השיבושים הרבים בקו התקשורת, החליטו מתכנני הקו לשלוח, בנוסף למילה המקורית, סיבית זוגיות עבור כל בית במילה, ללא ביצוע שינויים בפרוטוקול ה-UART הבסיסי כפי שנלמד בכיתה. השינוי היחיד אשר ניתן לבצע הוא שינוי משך השידור של ביט בודד (t_{bit}).

מהו משך השידור $t_{bit-new}$ אשר יאפשר שמירה על קצב שידור המילים המקורי?

פתרון

קצב שידור המילים המקורי הוא 2000 מילים בשנייה, כלומר משך שידור של כל מילה הוא $500\mu sec$. נרצה להוסיף סיבית זוגיות לכל בית במילה, כלומר להוסיף סה"כ 8 סיביות זוגיות, ונקבל כעת כי כל מילה היא באורך $72 = 64 + 8$ סיביות.

ע"מ לשדר 72 סיביות נצטרך לשדר $(72:8) = 9$ פעמים, ולכל שידור נוסיף $start-bit$ ו- $stop-bit$. כלומר נוסיף סה"כ עוד $(9 \cdot 2) = 18$ סיביות לכל מילה.

לאחר השינוי, ע"מ לשדר מילה שלמה נצטרך לשדר 90 סיביות.

כאמור, הקצב המקורי דרש שמשך שידור של כל מילה יהיה $500\mu sec$, ולכן ע"מ לקבל משך שידור של ביט בודד נחלק זמן זה ב-90 ונקבל:

$$t_{bit-new} = \frac{500}{90} \mu sec = 5.56 \mu sec$$

פורמט פקודה ב-RISC-V

פקודות ומידע שמורים בזיכרון, לכל אחד מהם יש כתובת בזיכרון.
יש רגיסטר ייעודי המחזיק את כתובת הפקודה המתבצעת כעת: "Program Counter" (PC).
רוב המידע שאנחנו עובדים איתו מגיע בתצורה של 'מילים' (32 ביטים). כך גם הרגיסטרים במעבד.
כל הפקודות גם הן מכילות 32 ביטים, המסודרים בפורמטים קבועים לפי סוג הפקודה.

R-Type

פקודות אלה מסודרות באופן הבא:

31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rd	opcode	
7	5	5	3	5	7	

opcode ביחד עם **funct3** ו**funct7** אחראים על זיהוי הפקודה.
השדות **rs1** ו**rs2** מכילים כתובת רגיסטרים לקריאה והשדה **rd** מכיל כתובת רגיסטר לכתובה (רגיסטר יעד). כל אחד מאלה מכיל 5 ביטים (צריך להכיל מספר בין 0-31 ולכך מספיקים 5 ביטים).

I-Type

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	funct3	rd	opcode	
12	5	3	5	7	

בפורמט פקודה זה השדה **imm[11:0]** מכיל מקום לערך של קבוע בגודל 12 ביט.
קבוע זה במעבד יעבור sign-extension לגודל של 32 ביט על מנת שנוכל לעשות פעולות אריתמטיות איתו ועם שאר הגדלים שיש לנו ברגיסטרים שאנחנו עובדים איתם הם בגודל זה).

כל פקודות shift (slli, srli, וכו') הם I-Type.
כמו כן גם פעולות Load הם I-Type: בפקודות אלה שדה **rs1** ישמש ב**base** – הרגיסטר בו מצויה הכתובת הרצויה רוצים לטעון, והשדה **imm[11:0]** ישמש כ**offset[0:11]** – ערך קבוע, כאשר הכתובת אליה נקפוץ מחושבת ע"י $base + offset$.

S-Type

31	25 24	20 19	15 14	12 11	7 6	0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	
7	5	5	3	5	7	
offset[11:5]	src	base	width	offset[4:0]	STORE	

פורמט זה כולל את כל פעולות Store. באופן דומה ל**Load** גם כאן הכתובת (בה נטען) מחושבת ע"י $base + offset$. נשים לב כי ה**imm** פוצל ע"י ש **rs1** ו**rs2** ישארו באותו מקום.
שדה **rs2** מכיל את כתובת הרגיסטר בו המידע אשר נרצה לטעון.

B-Type

פורמט זה מכיל את פקודות הbranch. כאשר אנחנו בוחנים פקודות של קפיצה, נרצה לקפוץ מהפקודה הנוכחית הנמצאת בPC לפקודה אחרת. בד"כ הפקודה האחרת תהיה יחסית "קרובה" לפקודה הנוכחית (אין 'חורים' בקוד והוא כתוב ברצף). נרצה להשתמש בimm כערך offset שאותו נוסיף/נחסיר מהPC. נראה שגם בפקודות קפיצה לאחר שממקמים את כל השדות הנחוצים (opcode, רגיסטרים, וכו') נשאר לנו imm בגודל של 12 ביטים. אם אחד מהם משמש עבור הסימן (נרצה להיות מסוגלים גם להחסיר) אז נוכל לקפוץ מהכתובת הנוכחית המצויה בPC עד $\pm 2^{11}$ כתובות.

אולם, כיוון שהפקודות מסודרות תמיד כ-32 ביט כל אחת, כתובת של כל פקודה תמיד תסתיים ב00, כלומר, תתחלק ב4 (בכל פקודה יש 4 בתים). לכן, תיאורטית נוכל להתעלם מ2 הביטים האחרונים (להוסיף אח"כ 00 באופן "ידני") ובכך להרחיב פי 4 את טווח הכתובות לקפיצה, כלומר עד $\pm 2^{13}$ כתובות, או עד $\pm 2^{11}$ פקודות. הנ"ל לא קורה במעבד שלנו (מסיבות של התאמה גם לפקודות של 16 ביט) ובמקום שני אפסים, נשמיט רק 0 אחד, ולכן נרחיב את טווח הכתובת לקפיצה רק פי 2, כלומר עד $\pm 2^{10}$ פקודות. הפורמט עצמו נראה כך:

31	30	25	24	20	19	15	14	12	11	8	7	6	0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode						
1	6	5	5	3	4	1	7						

נשים לב ש12 הביטים של imm שוב מפוצלים. נשים לב גם שלא מופיע imm[0] מהסיבה שהוזכרה קודם לכן – ביט זה הוא תמיד 0 והוא מגיע "מבחוץ".

U-Type

פקודות התומכות בimm בעל מספר רב יותר של ביטים. הפורמט יראה כך:

31	12	11	7	6	0
imm[31:12]	rd	opcode			
20	5	7			
U-immediate[31:12]	dest	LUI			
U-immediate[31:12]	dest	AUIPC			

פקודה לדוגמה: LUI – לוקח את ה20 ביטים מהשדה imm[31:12], מכניס אותם ל20 ביטים העליונים ברגיסטר שכתובתו בrd, ומוחק את הביטים התחתונים בו. פקודה זו שימושית אם רוצים להכניס ערך של 32 ביטים מimm לתוך רגיסטר – נעשה זאת ע"י שימוש בLUI ואח"כ ADDI על מנת להשלים את 12 הביטים התחתונים. קיימת פסאדו-פקודה הטוענת ערך imm של 32 ביטים לרגיסטר (והיא למעשה עושה את מה שתואר) בשם liw.

J-Type

פורמט זה מכיל את פקודות הjump, כמו JAL למשל.
 כאן imm בגודל של 20 ביטים והוא משמש את offset לחישוב כתובת הקפיצה (כזכור נרצה לקפוץ לכתובת PC+offset).
 נבדוק מה טווח הקפיצה: 20 ביטים, אחד משמש לסימן, לכן $\pm 2^{19}$ כתובות, או $\pm 2^{17}$ פקודות (הכתובות כזכור בקפיצות של 4). אך גם כאן נעשה את אותו "טריק" עם ה0 הנוסף ובכך נכפיל את הטווח, ולכן טווח הקפיצה יהיה $\pm 2^{18}$ פקודות מהפקודה הנוכחית.
 הפורמט נראה כך:

31	30	21	20	19	12	11	7	6	0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode				
1	10	1	8	5	7				
offset[20:1]				dest	JAL				

נשים לב שהפקודה JALR היא אמנם גם פקודת קפיצה, אך כיוון שהיא צריכה מקום נוסף לרגיסטר (תזכורת: פקודה זו דומה לJAL, רק שהפעם כתובת הקפיצה אינה הכתובת הנוכחית ועוד imm, אלא הכתובת המצויה ברגיסטר rs1 ועוד imm) פורמט J אינו מתאים, והיא ממומשת באמצעות פורמט I (ניתן לראות שיש שם בדיוק את השדות להם היא זקוקה).
 פסאודו-פקודות הממומשות ע"י JALR:
 $ret = jr\ ra = jalr\ x0, ra, 0$ (לא ניתן לשמור לרגיסטר x0, לכן למעשה לא נשמר כלום ורק מתבצעת הקפיצה לרגיסטר ra).

שאלה ממבחן – שאלה 11 מתוך מועד א' חורף 2019

להלן קוד שאמור להתבצע על אחד ממעבדי V-RISC:
`00000010000011100010100000000011`
`00000001000011000000011000110011`
 נתון כי לפני ביצוע שתי הפקודות הנ"ל:
 • התוכן של כל רגיסטר ב-file register שווה למספר הרגיסטר, לדוגמה רגיסטר x4 מכיל את הערך 4.
 • התוכן של כל בית בזיכרון הנתונים שווה לבית התחתון של כתובתו, לדוגמה בכתובת 0x12345678 נמצא הערך 0x78.
 מהו הערך ברגיסטר X12 לאחר ריצת 2 הפקודות? התשובות נתונות בבסיס עשרוני.

פתרון: נפענח מהן כל אחד מהפקודות:
 בפקודה הראשונה opcode הוא 0000011 func3 הוא 010 ולכן מדובר בlw. נפענח את הפקודה המלאה:
 $lw\ x16\ x28\ 32$: לסיכום: $imm[0:11]=00000100000=32$, $rs1=11100=x28$, $rd=10000=x16$
 בפקודה הראשונה opcode הוא 0110011 func3 הוא 000 לכן מדובר בadd. נפענח את הפקודה המלאה:
 $add\ x12\ x24\ x16$: לסיכום: $func7$ מתאים לADD, $rs2=10000=x16$, $rs1=11000=x24$, $rd=01100=x12$
 $lw\ x16\ x28\ 32$
 $16 + add\ x12\ x24\ x16$
 ולכן בx16 לאחר הפקודה הראשונה יהיה הערך בכתובת $60=28+32$ והוא 60.
 לאחר הפקודה השנייה בx12 יהיה $84 = 60+24$. תשובה: 84

RISC-V Single Cycle

מרכיבי המעבד :

PC – Program Counter – מצביע על הכתובת של הפקודה הנוכחית אותה אנחנו רוצים לבצע במחזור השעון הנוכחי.

IMEM - חלק מהזיכרון בו שמורות ההוראות.

4+ – יחידה אריתמטית שמטרתה להעלות את PC ב4, על מנת להתקדם לפקודה הבאה.

Register File – בו נמצאים כל הרגיסטרים (32 במעבד שלנו).
המספר המקסימלי של רגיסטרים בפעולה הוא 3 – שני רגיסטרי מקור ורגיסטר יעד.
כניסות:

- dataD – כניסת מידע [32 ביט]
- AddrD - מספר רגיסטר היעד (הרגיסטר בו שומרים את פעולות החישוב) [5 ביט]
- AddrA – מספר רגיסטר המקור הראשון [5 ביט]
- AddrB – מספר רגיסטר המקור השני (הרגיסטרים עליהם עושים את הפעולות) [5 ביט]

יציאות:

- DataA – יציאת מידע של הרגיסטר הראשון [32 ביט]
- DataB – יציאת מידע של הרגיסטר השני [32 ביט]

Imm Gen – תפקידו להכין את ערך imm. מקבל מספר ביטים של imm (תלוי בפורמט הפקודה) ואת סוג הפקודה ועושה sing extension עד לקבלת ערך מספרי באורך 32 ביט – איתו אפשר לבצע פעולות אריתמטיות.

ALU – יחידה אריתמטית, מבצעת חישובים.

DMEM - חלק מהזיכרון בו שמור המידע. כניסות/יציאות :

קו בקרה מהבקר – Read/Write – בהתאם לפקודה (אם צריך לכתוב או לקרוא מהזיכרון),
Addr – הכתובת אליה/ממנה נכתוב/נקרא , DataW – המידע אשר נרצה לטעון לזכרון ,
DataR – המידע אשר חולץ מהזכרון.

Branch Comparator – הרכיב האחראי על פעולות branch.

כניסות : A,B – אלו למעשה היציאות DataA וDataB מה RegFile ומכילות את המידע שהיה על הרגיסטרים. במידה ו $A=B$, נקבל ביציאה $BrEq=1$. אם $A < B$ או $BrLT=1$.
כניסת הבקרה BrUn מציינת אם המספר הוא מסוג sing או לא (אם אחד שלילי ואחד חיובי זה רלוונטי להשוואת הגודל).
נשים לב שתוצאת Branch comp משפיעה באופן ישיר על כניסת הבקרה PCSel בפקודות branch כיוון שזו קובעת אם תקרה הקפיצה או שלא.

נשים לב שבמעבדת בארכיטקטורת Single-Cycle, זמן המחזור נקבע ע"פ הפעולה הארוכה ביותר – גם אם שאר הפעולות לא זקוקות לכל חמשת השלבים.
כיוון שפעולה כמו lw זקוקה לחמשת השלבים זמן המחזור ייקבע להיות הזמן הטוטאלי של כולם, ובמקרה של הטבלה הנתונה – $T=800ps$.

שאלה ממבחן – שאלה 7 מתוך מועד ב' קיץ 2019

הוחלט להוסיף 2 פקודות קפיצה חדשות לארכיטקטורת RISC-V Cycle Single.
הפקודה הראשונה מקבלת מספר של רגיסטר יחיד שבו מאוחסנת כתובת לקפיצה.
הפקודה הבאה תגיע מכתובת זו. בנוסף ה-PC+4 ישמר ברגיסטר מספר 1 (אין צורך לציין את מספר הרגיסטר בפקודה).
הפקודה השנייה מקבלת מספרים של שני רגיסטרים. באחד מהם מאוחסנת כתובת לקפיצה- הפקודה הבאה תגיע מכתובת זו. ברגיסטר השני ישמר ה-PC+4.

מותר להוסיף כניסות MUX וחיבורים במסלול הנתונים, ולשנות את הבקר.
אסור לשנות את מבנה ה-File Register והזיכרון.

מבין התשובות הבאות, בחרו את התשובה הנכונה ביותר:

- א- אפשר לממש רק את האפשרות הראשונה.
- ב- אפשר לממש רק את האפשרות השנייה.
- ג- אפשר לממש את שתי האפשרויות.
- ד- אי אפשר לממש אף אחת מהאפשרויות הללו.

פתרון:

ג' –

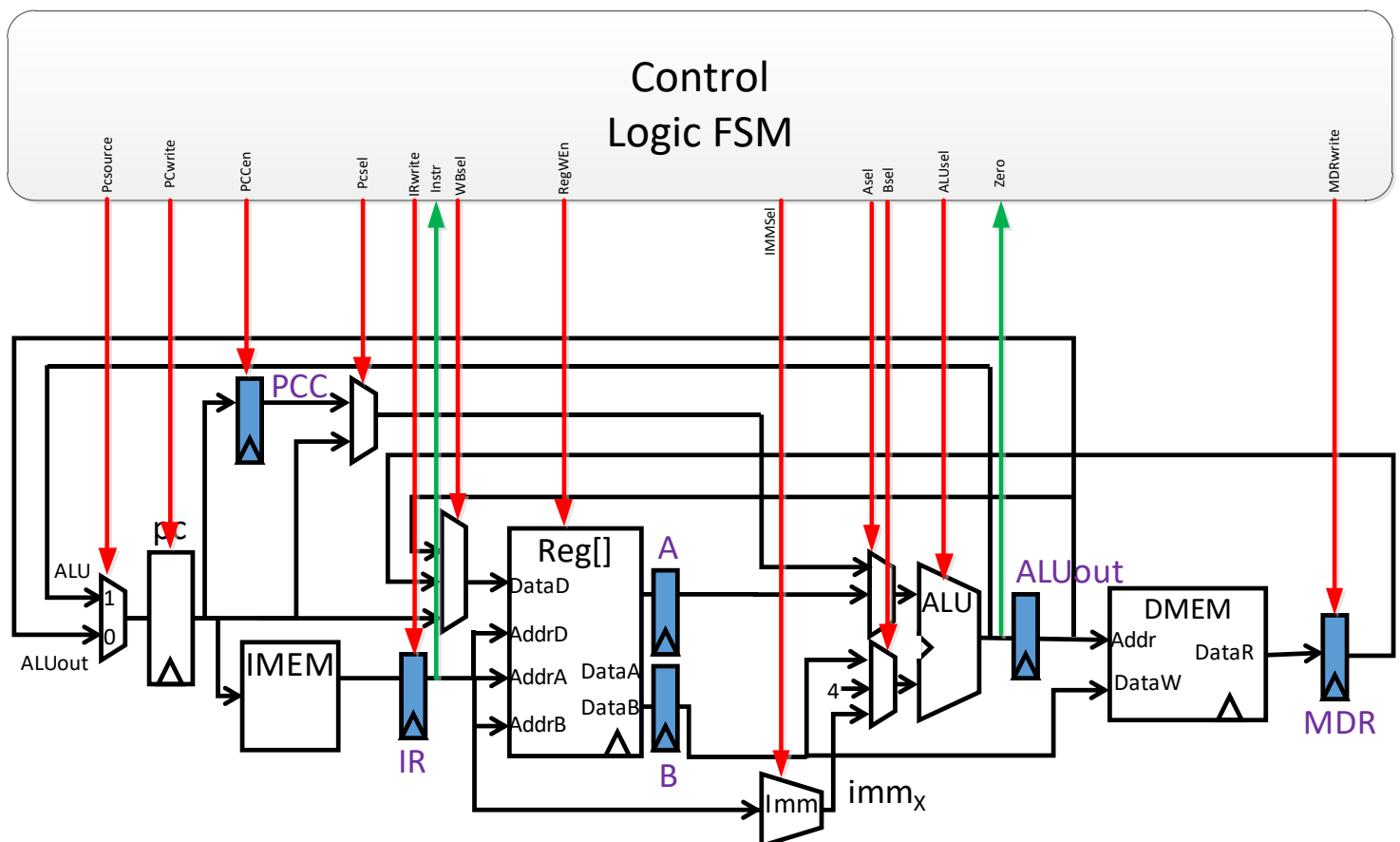
את שתי האפשרויות ניתן לממש, כיוון שניתן גם לשמור וגם לקרוא מקובץ הרגיסטרים ובשתי האפשרויות צריך לקרוא מהרגיסטר שבו מאוחסנת כתובת הקפיצה כדי לדעת לאן לקפוץ, ולכתוב אל הרגיסטר שאליו רוצים לכתוב את ערך ה-PC+4.

RISC-V Multi Cycle

החיסרון העיקרי בsingle-cycles היה שגם פעולות קצרות, נמשכות אותו דבר כמו הפעולה הארוכה ביותר. הרעיון בmulti-cycles הוא לפצל כל פקודה לכמה מחזורי שעון, כך שמחזור השעון קצר משמעותית, וכל פקודה תמשך מספר שונה של מחזורי שעון ולא תארך יותר זמן ממה שהיא צריכה.

השינויים העיקריים בDatapath:

- הרכיבים Branch Comp. ו' +4' לא קיימים בארכיטקטורה זו – במקומם החישוב מתבצע בALU.
- הוספת רגיסטרים זמניים:
 - **PCC** – שומר את הכתובת של הפקודה הנוכחית (כיוון שבPC "דורסים" את הערך השמור עם PC+4).
 - **IR** – שומר את הפקודה הנוכחית ע"מ שנוכל לטפל בפקודה זו בכל השלבים מבלי להגיע לפקודה הבאה.
 - **A, B** – שומרים את המידע שחולץ מהרגיסטרים
 - **ALUOut** – שומר את מוצא הALU
 - **MDR** – שומר את המידע שחולץ מהזיכרון



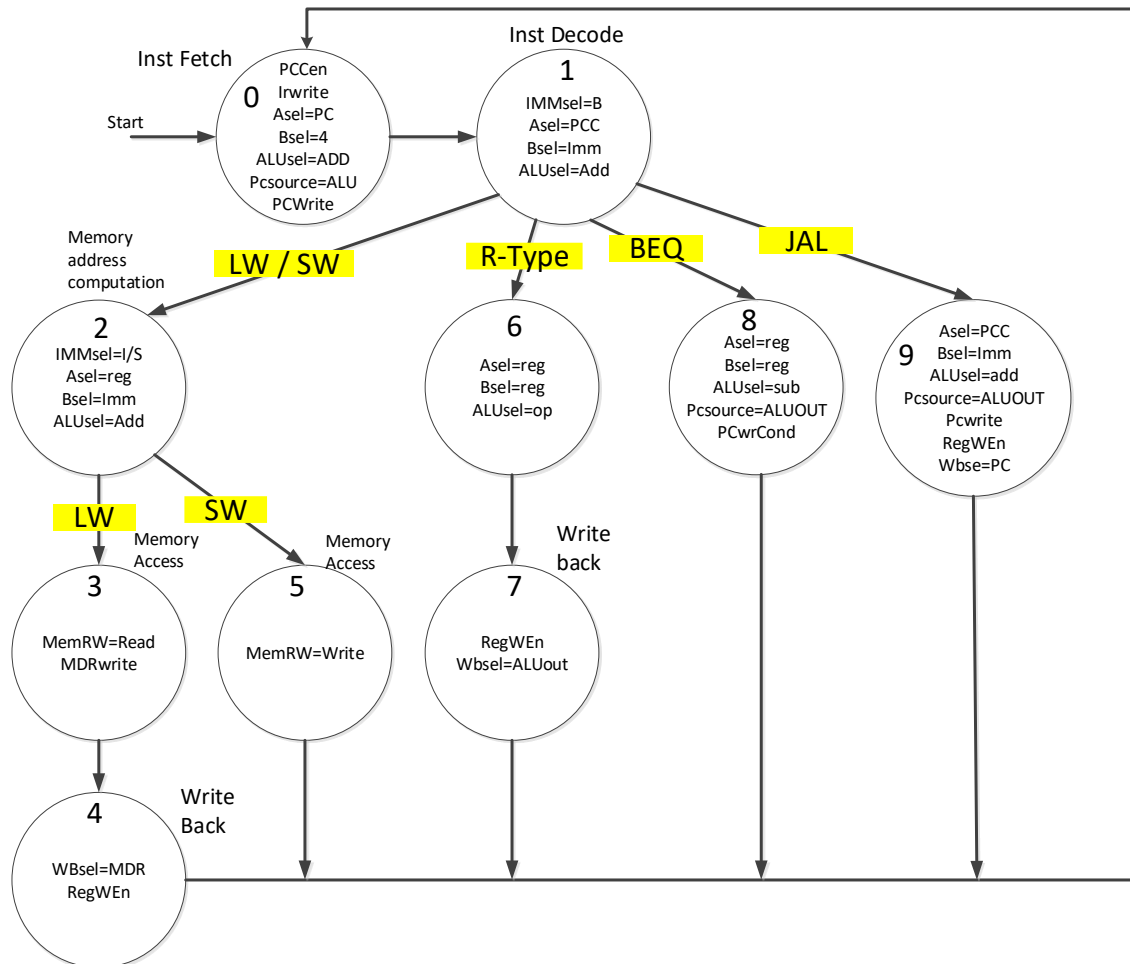
PCC=PC current

IR=Instruction Register

MDR=Memory Data Register

הבקר – Controller

הבקר צריך להוציא בכל מחזור קווי בקרה שונים, לכן הוא ממומש ע"י מכונת מצבים. המחזור הראשון (Inst Fetch) והשני (Inst Decode) משותפים לכל הפקודות. שאר המצבים שונים ע"פ סוג הפקודה.



נעבור על מצבי הבקר השונים :

- **Inst Fetch [0]** – זהה לכל הפקודות.
לרגיסטר IR נכנסת הפקודה שנמצאת בזכרון (בכתובת המצויה בPC), לרגיסטר PCC נכנסת הכתובת של הפקודה הנוכחית שנמצאת בPC, והכתובת בPC מעודכנת להיות PC+4 (החישוב נעשה בALU)
- **Inst Decode [1]** – זהה לכל הפקודות. ל A,B מוכנסים המידע מהרגיסטרים שיצא מהRegFile, ובנוסף, מחושבת כתובת הקפיצה המשוערת למקרה ומדובר בפקודת branch : $ALUOut = PCC + branchoffset$

השלבים הבאים שונים בפורמטים שונים :

R-Type

- **Execute[6]** – ברגיסטר ALUOut נשמרת תוצאת החישוב של הALU
- **Write Back[7]** – המידע ALUOut מגיע לכניסת DataD בRegFile וייכתב לרגיסטר היעד עם עליית השעון.

BEQ

- **Execute[8]** – תנאי הקפיצה נבדק בALU. במוצא הALU יש חיבור (לא משורטט בציור) בשם Zero שיוצא מהALU לבקר ומתקיים $Zero=1$ כאשר תוצאת החיסור בALU היא אפס (כלומר הערכים A ו B שווים).
בפעולות branch יציאת הבקרה PCwrCond הופכת ל'1' ובאמצעות הלוגיקה הבאה : (הנכנסת לרגיסטר PC) מתעדכנת כתובת הקפיצה.



LW/SW

- **Execute[2]** – מחשבים בALU את הכתובת הרצויה :
 $ALUOut = A(\text{base Address}) + \text{imm}(\text{offset})$

SW

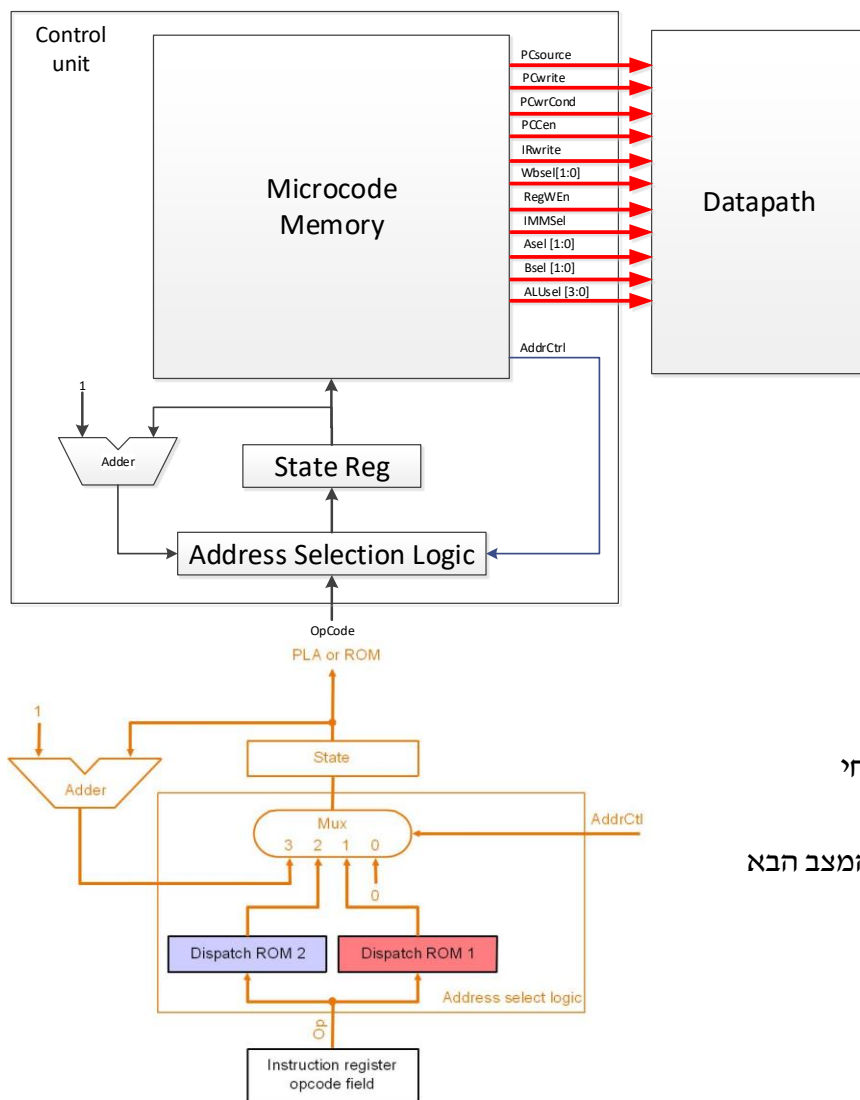
- **Memory[5]** – מכניסים לכתובת השמורה בALUOut (שחושבה בשלב הקודם) את המידע ברגיסטר B (המידע שחולץ מrs2).

SW

- **Memory[3]** – מכניסים את המידע השמור בכתובת שבALUOut (שחושבה בשלב הקודם) לרגיסטר MDR.
- **Write Back[4]** – המידע מMDR מגיע לכניסת DataD בRegFile וייכתב לרגיסטר היעד עם עליית השעון.

JAL

- **Execute[9]** –
לתוך רגיסטר היעד מגיעה הכתובת לחזרה $PC+4$. יש לזכור שכתובת זו כבר נמצאת ברגיסטר PC – הוכנסה לשם בשלב הIF.
לתוך הרגיסטר PC מגיעה הכתובת $PCC + \text{offset}$, כאשר החיבור מחושב בALU.



מבנה הבקר

מורכב מחלק של ROM (Microcode Memory) בעלת 10 שורות, בו כל שורה מייצגת מצב בדיאגרמת המצבים. העמודות מייצגות את יציאות הבקר. (מקבלים פה טבלה הרבה יותר קטנה מהבקר ב-single cycle).

העמודה האחרונה קובעת מה יהיה המצב הבא, והוא אחד מהבאים:

- **Fetch** – לחזור למצב 0
- **Dispatch1** – טבלת קפיצה ממצב 1
- **Dispatch2** – טבלת קפיצה ממצב 1
- **Seq** – המצב הוא 1+ המצב הנוכחי

בState Reg מופיע תמיד המצב הנוכחי, והמצב הבא ייבחר באמצעות MUX באופן הבא:

שאלה ממבחן – שאלה 15 מתוך מועד א' אביב 2020

מעוניינים להוסיף מימוש של הפקודה `dlw` כפסאודו-פקודה תוך שימוש בפקודות קיימות.

פקודה זו מביאה מילה מהזיכרון לפי כתובת המחושבת באופן הבא :

כתובת המילה מובאת מהזיכרון מהכתובת ששמורה ברגיסטר `rs` ועוד ערך ה-`imm`, ושומרת את המילה

שהובאה מהזיכרון ברגיסטר `rd`.

פקודה זו בעלת הפורמט :

`dlw rd, rs, imm`

המבצעת את הפעולה הבאה :

$reg[rd] \leftarrow Mem[Mem[reg[rs]+imm]]$

א. כתבו את המימוש המינימלי של הפקודה כרצף של פקודות אמיתיות (ניתן להשתמש

ברגיסטרים `t1, t0` במידת הצורך)

ב. מה מספר המחזורים המינימלי הנדרש לביצוע פסאודו פקודה זו במעבד

`single cycle RISC-V`?

ג. מה מספר המחזורים המינימלי הנדרש לביצוע פסאודו פקודה זו במעבד

`Multi-cycle RISC-V`?

ד. כעת ניתן לבצע שינויים במעבד הכוללים הוספת/הרחבת בוררים, והוספת חיוטים.

מה מספר המחזורים המינימלי הנדרש לביצוע פקודה זו כפקודה אמיתית במעבד

`Multi-cycle RISC-V` ?

פתרון

א. מימוש באמצעות שתי פקודות :

`lw t0 imm(rs)`

`lw rd 0(t0)`

ב. מספר המחזורים המינימלי הוא 2 (ב-`single-cycle` כל פקודה אורכת מחזור)

ג. מספר המחזורים המינימלי הוא 8 (ב-`multi-cycle` כל פקודת `lw` אורכת 4 מחזורים)

ד. נבדוק אילו שלבים נצטרך ע"מ לבצע פקודה זו כפקודה אמיתית במעבד :

1+2 – `D+IF` – בדומה לשאר הפקודות ב-`multi-cycle`

3 – `EXE` – חישוב `rs+imm` ושמירתו ב-`ALUOut`

4 – `MEM` – גישה לזכרון וחילוץ המידע השמור בכתובת `rs+imm`

5 – `MEM` – גישה שנייה לזכרון. נעשה זאת ע"י הוספת חיבור נוסף מ-`MDR` ל-`DMEM-Addr`, ובכך נחלץ

את המידע השמור ב-`MEM[rs+imm]`.

6 – `WB` – שמירת המידע ב-`rd`

תשובה : נצטרך בסה"כ 6 מחזורי שעון.

חריגות ופסיקות

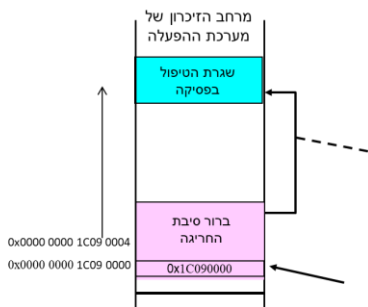
חריגות (Exceptions) הן אירועים הגורמים להפסקת רצף הפעולות ולקפיצה לתוכנית אחרת. סיבות אפשריות לחריגות:

- **התוכנית עצמה** – לדוגמה, כאשר התוכנית צריכה לקבל משהו ממערכת ההפעלה (למשל קלט מהמקלדת)
- **בעיות בביצוע התוכנית** – דוגמות: overflow, חלוקה באפס, גישה לכתובת לא חוקית...
- **גורם חיצוני** – קלט מהמקלדת/שעון/אינטרנט...

לחריגה הנגרמת ע"י גורם חיצוני נקרא **פסיקה** (Interrupt). ההבדל העיקרי בין חריגה ופסיקה הוא שבפסיקה נסיים את הפעולה הנוכחית ורק אז נטפל בגורם החיצוני בעוד שבחריגה נעצור את הפקודה הנוכחית.

טיפול בחריגות

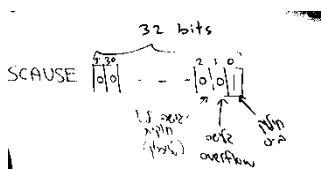
תחילה נצטרך לשמור את הכתובת של הפקודה הנוכחית, ע"מ שנוכל לדעת לאן לחזור לאחר שנסיים לטפל בגורם הגרם לחריגה. ב-RISC-V קיים רגיסטר מיוחד השומר את הכתובת הנוכחית הנקרא SEPC. (הכתובת הנוכחית תגיע מהרגיסטר PCC, ובמעבדים בהם לא קיים נחסיר 4 מ-PC). נתעסק בקורס בשתי שיטות עיקריות לטיפול בחריגות:



שיטה א' – קוד הגורם לחריגה

בשיטה זו בכל פעם שתקרה חריגה נקפוץ לכתובת קבועה (0x1c090000) וממנה יתחיל קטע קוד שמטרתו היא בירור סיבת החריגה, באמצעות התבוננות ברגיסטר הנקרא SCAUSE. המעבד טוען קוד לרגיסטר זה (עוד מלפני הקפיצה) קיימות שתי שיטות שנלמד לקידוד הקוד ברגיסטר זה:

- **חזקות של 2** – תחילה כל הביטים ברגיסטר שווים לאפס. מגדירים כי כל ביט ברגיסטר אחראי על



סוג מסוים של חריגה, למשל: ביט 0 אחראי על חילוק באפס, ביט 1 אחראי על overflow, ביט 2 אחראי על גישה לכתובת לא חוקית, וכן הלאה... המעבד טוען 1 במיקום הביט (חזקה של 2) המתאים לגורם הנוכחי לחריגה. יתרון לשיטה זו שאפשר לטפל בכמה חריגות במקביל (לטעון כמה ביטים ב-1).

חסרון לשיטה זו שהרגיסטר מכיל 32 ביטים ולכן נוכל לטפל לכל היותר ב-32 חריגות שונות.

- **מספר רץ** – בשיטה זו כל חריגה מקודדת לקוד מסוים המייצג את הסיבה הנוכחית לחריגה.

היתרון בשיטה זו הוא שניתן כעת לטפל ב- 2^{32} חריגות שונות (לעומת 32 בשיטה הקודמת), אולם החיסרון הוא שניתן לטפל כל פעם רק בחריגה אחת בו זמנית.

לאחר שהסיבה אותרה, תתבצע קפיצה לפונקציה ייעודית שמטרתה טיפול בחריגה הספציפית שהתרחשה. לבסוף הפונקציה תקפוץ חזרה לפקודה הנוכחית הנמצאת ב-SEPC ותרץ את הפקודה לאחר שטופלה בשנית.

שיטה ב' – וקטור הפסיקות

זהו בלוק במרחב הזיכרון של מערכת ההפעלה. בשיטה זו לא נקפוץ לכתובת קבועה, אלא נקפוץ לכתובת מסוימת לפי סוג החריגה. המעבד מחשב את הכתובת אליה נצטרך לקפוץ (לפי סוג החריגה), לכן בשיטה זו

אין צורך ברגיסטר SACUSE כיוון שכל שורה ייעודית לסיבת חריגה מסוימת.

בכתובות אלה יכולות להופיע פקודות קפיצה לפונקציה שתמשיך את הטיפול

בחריגה.

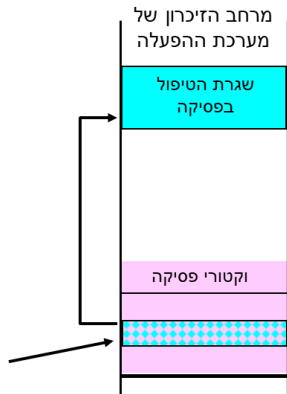
נשאלת השאלה מדוע לקפוץ לשורה בווקטור הפסיקות ולא לקפוץ ישר לתוכנית

handler המטפלת בחריגה, והסיבות לכך הן שכך קל יותר לבצע שינויים בתוך

מערכת ההפעלה (צריך לשנות את תכנית handler אך אין צורך לשנות את כתובת

הקפיצה שהמעבד מחשב) ושקל יותר בצורה זו לחשב את כתובת הקפיצה (בדומה

לקפיצה למערך – קופצים לכתובת הבסיס ומשנים רק את offset).



שאלה ממבחן – שאלה 15 מתוך מועד א' אביב 2020

נתון מעבד RISC-V Multicycle התומך בטיפול בחריגות, כך שהוא מפסיק את ריצת התוכנית במידה והתקבלה חריגה. אחד הסטודנטים בקורס "מערכות ספרתיות ומבנה המחשב" כתב את הקוד הבא:

```
0x1AA0 0000    Main: addi x2, x0, 4
0x1AA0 0004      addi x4, x0, 1
0x1AA0 0008      mult x1, x2, x2
0x1AA0 000C      add x1, x1, x1
0x1AA0 0010      add x0, x1, x2
```

שימו לב: הפקודה `mult rd, rs1, rs2` מבצעת כפל בין שני

הרגיסטרים `rs1` ו-`rs2` כך שמתקיים: $rd = rs1 \cdot rs2$. באופן דומה

הפקודה `div rd, rs1, rs2` מבצעת חלוקה כך שמתקיים: $rd = rs1/rs2$

הסטודנט גילה כי נפח הזכרון הוא $1GB (2^{30}B)$. שותפו של הסטודנט

בחן את הקוד וקבע בהחלטיות כי הרצת הקוד תגרום לחריגה.

איזה סוג חריגה תתרחש?

```
0x1AA0 0014    Loop: addi x1, x1, -1
0x1AA0 0018      beq x1, x0, EXIT
0x1AA0 001C      add x4, x4, x4
0x1AA0 0020      div x0, x1, x1
0x1AA0 0024      j Loop
0x1AA0 0028    EXIT:
0x1AA0 002C      sw x4, 0(x4)
```

פתרון:

נבצע מעקב אחרי התוכנית:

MAIN מאתחלת את הרגיסטרים לערכים:

$$x2=4$$

$$x4=1$$

$$x1=x2*x2=4*4=16$$

$$x1=x1+x1=16+16=32$$

הלולאה, מבצעת $x1-1$ פעמים (31 פעמים) $2*x4$, כלומר בסוף הלולאה הערך השמור ב- $x4$ יהיה 2^{31} .

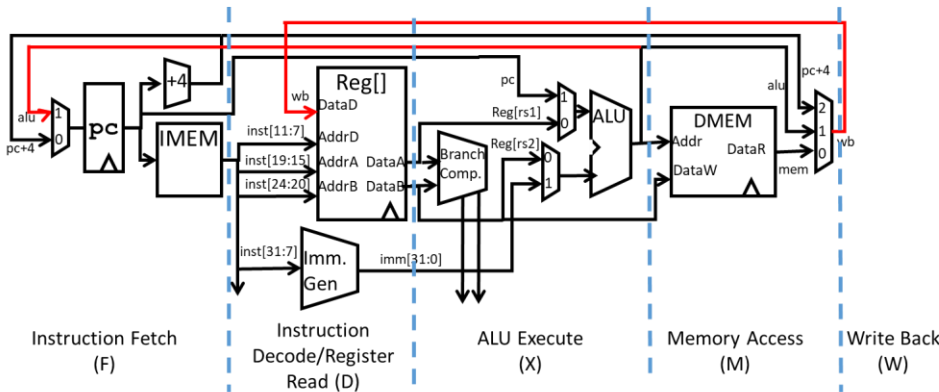
לאחר מכן מתבצע `sw` לכתובת השמורה ב- $x4$, אולם, נתון כי הזיכרון בגודל 2^{30} לכן אין תא בזיכרון היכול

להיות מיוצג ע"י מספר גדול יותר מגודל הזיכרון – כלומר, תתרחש חריגה מסוג **גישה לכתובת לא חוקית**.

Pipelined RISC-V

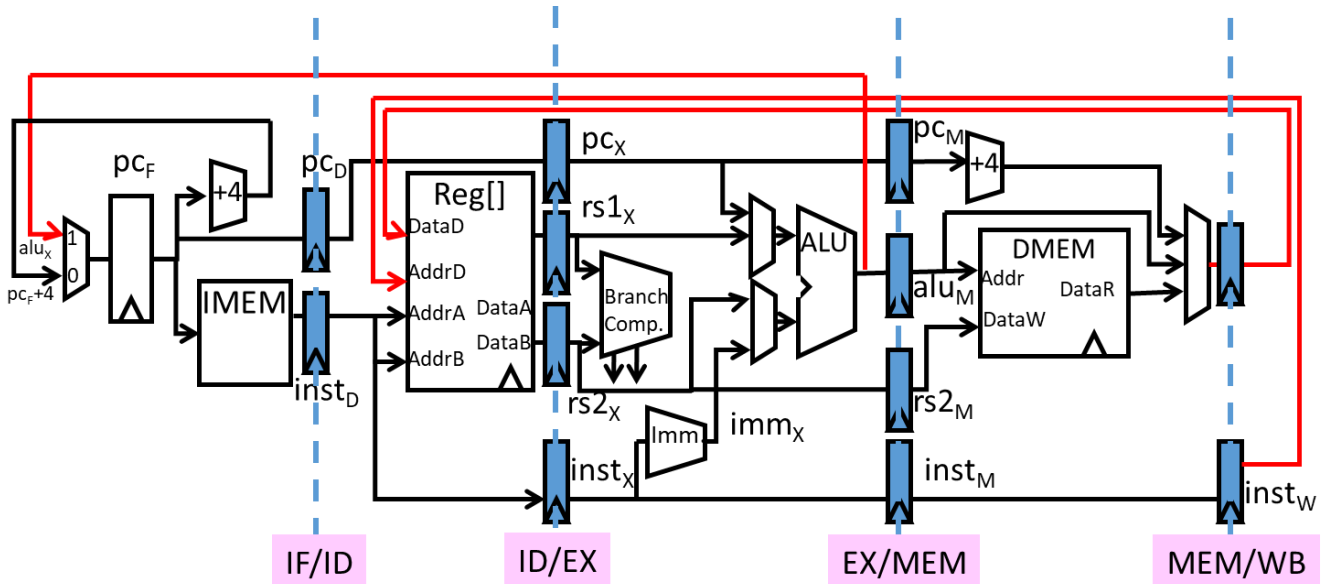
בתצורה זו נחלק את DataPath ל-5 שלבים: IF, D, EXE, MEM, WB

ובכך ניצור מערכת מצוננת כך שכל שלב מהווה יחידה צירופית ורגיסטרים מפרידים בין השלבים.



בכל שלב ניתן להשתמש רק במשאבים הנמצאים אצלו (למשל לחישוב PC+4 המחושב בשלב ה-IF לא נוכל להשתמש ב-ALU שנמצא בשלב ה-EXE) ולכן נוסיף בחזרה את יחידת ה-PC+4 ואת ה-Branch comp גם נוספת של '+4'.

בנוסף נוסיף רגיסטרים שיפרידו בין כל שלב:



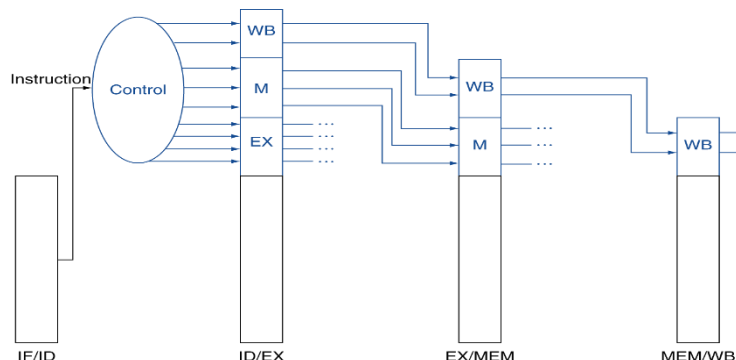
כיוון שיש לנו 5 שלבים ניתן לטפל בעד 5 פקודות במקביל ברגע נתון.

נשים לב שצריך להעביר את הפקודה לאורך כל הצינור, ולכן הפקודה עוברת ברגיסטרים ייעודיים.

יחידת ה-PC+4 נוספת קיימת בשלב ה-WB לטובת פקודות JAL.

הבקר – Controller

נשאלת השאלה איך ימומש הבקר אם בכל שלב נמצאת פקודה אחרת, והתשובה לכך היא שיציאות הבקרה מחושבות מראש עבור כל השלבים, ועוברות ביחד עם הפקודה ברגיסטרים של הצינור. אולם, אין צורך להעביר תמיד את כל היציאות – לאחר שמסיימים עם שלב מסוים אין שימוש ביציאות שלו ולכן נעביר רק את היציאות של השלבים הנוותרים:



Hazards

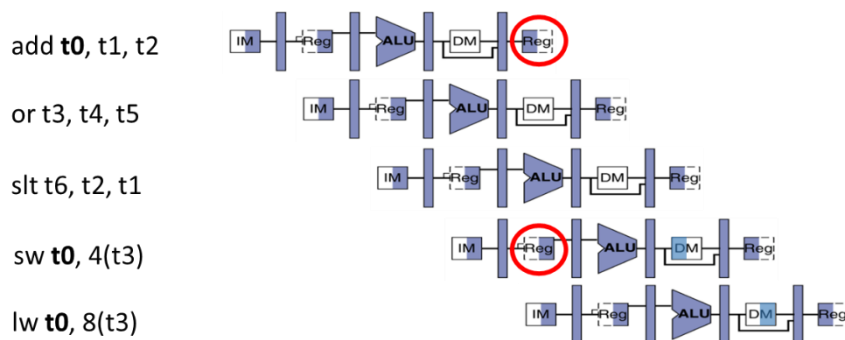
Structural Hazard

כאשר שתי פקודות שונות צריכות להשתמש באותו המשאב. למשל כשפקודה אחת צריכה לחשב PC+4 ופקודה אחרת צריכה לחשב כתובת לקפיצה – שתייהן צריכות להשתמש בALU לטובת חישוב. הפתרון הוא הוספת חומרה, ולכן הוספנו למשל את היחידה '+4' שלא קיימת בmulti-cycle.

Data Hazard

קורה למשל כאשר פקודה מסוימת רוצה לקרוא ערך מרגיסטר מסוים שפקודה שלפניה עדיין לא הספיקה לעדכן (תזכורת: קריאה מרגיסטר היא א-סינכרונית, וכתובה מתבצעת בסוף מחזור עם עליית השעון).

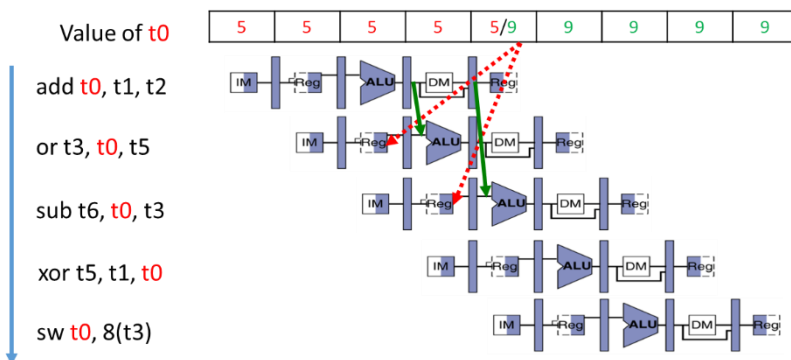
WB → DEC – רלוונטי לפקודות במרווח של 3 פקודות.



כאשר פקודה מסוימת קוראת מרגיסטר (בשלב DEC), אך פקודה שלפניה נמצאת בשלב WB וכותבת לאותו הרגיסטר. הפתרון הוא forwarding בתוך הRegFile – במקום לקרוא את הערך מהרגיסטר, נוסף חיבור מכניסת המידע של RegFile ישירות

לרגיסטרים שבמוצא RegFile, ויחידת בקרה RegFile Forward Logic וMUXים – וכאשר היח' בקרה מזוהה מצב כזה, הערך במוצא מגיע ישירות מכניסת המידע (ובמקביל גם מתעדכן ברגיסטר המדובר).

WB → EXE – רלוונטי לפקודות במרווח של 2 פקודות.



כאשר פקודה רוצה לבצע חישוב על ערך של רגיסטר מסוים, אך פקודה שלפניה נמצאת בשלב WB וכותבת לאותו רגיסטר. הפתרון הוא forwarding במעבד משלב WB לשלב EXE.

MEM → EXE – רלוונטי לפקודות במרווח של 1 פקודות (פקודות סמוכות).

כאשר פקודה רוצה לבצע חישוב על ערך של רגיסטר מסוים, אך פקודה שלפניה ביצעה חישוב כלשהו עם אותו הרגיסטר וכרגע היא בשלב MEM. הפתרון הוא forwarding במעבד משלב MEM לשלב EXE.

על מנת שהמעבד ידע מתי צריך להפעיל forwarding בין שלב לשלב, נוסף יחידה לוגית למעבד – **Forwarding control logic** – הקלט שלה יהיה rdn של הפקודות שבשלב MEM וWB, והrsn של הפקודה הנמצאת בשלב EXE.

Load Data Hazard

סוג נוסף של forwarding במעבד הוא בפקודות lw. כאשר טוענים ערך מהזיכרון (lw) הוא יהיה זמין רק בשלב WB (למעשה, לאחר עליית השעון של שלב MEMn). לכן, פקודות לאחר פקודת lw שצריכות לעשות שימוש במידע שצריך להיטען, חייבות לחכות שהמידע יהיה לפחות בשלב WB של פקודת הlw, אחרת הוא אפילו לא נמצא בצינור:

פקודות במרווח של 2 פקודות – במקרה זה ניתן לעשות forwarding WB→EX שכבר ראינו שקיים.

פקודות במרווח של 1 פקודות (פקודות סמוכות): במקרה כזה (של פקודה שעושה שימוש בערך הבאה ברצף לאחר פקודת lw הטוענת את אותו הערך) אין דרך להתגבר על הבעיה, והפתרון האפשרי היחיד הוא לחכות שפקודת הlw תסתיים ע"י הכנסת NOP (פקודה שלא עושה שום דבר, ולמעשה שקולה להמתנה של פקודה אחת).
איך זה יתבצע?

נוסיף יחידה לוגית בשם **Hazard detection unit** שתפקידה לבדוק האם הפקודה שנמצאת בשלב EXn היא פקודת lw, ובנוסף – האם rdn שפקודת הlw רוצה לעדכן, זה אחד מהrs שהפקודה שנמצאת בשלב הDEC רוצה לקרוא. במידה וכן, היא תכניס NOP לצינור ע"י הפיכת סיגנלי הבקרה של הבקר לאפסים ובכך "נתקע" למשך מחזור אחד את הפקודות שלאחר הlw.

פתרון נוסף, במידה והדבר מתאפשר – לשנות את סדר הפקודות כך שיהיו מרווח של 2 לפחות בין פקודת הlw ובין הפקודה שרוצה לקרוא את הערך שהיא טוענת. (זוהי אופטימיזציה שנעשית בד"כ באופן אוטומטי ע"י הקומפיילר).

Control Hazard

בפקודות branch, נוכל לדעת האם תתבצע קפיצה (או שלא) רק בשלב EXn, כלומר במחזור השעון השלישי. אם כן, איך נדע בשני המחזורים העוקבים מה להכניס לpipe? נניח ("נהמר") שהקפיצה לא מתבצעת ונכניס כרגיל את הפקודות הבאות. אם נגלה בשלב EXn שהbranch אכן צריך לבצע קפיצה, נקפוץ לפקודה הנכונה ונבצע flush לשתי הפקודות שכבר הכנסנו לpipe (flush = מחיקת הפקודות ע"י הפיכת סיגנלי הבקרה לאפסים).
הערה: ביצענו flush ל 2 פקודות כי ידוע לנו שהחלטה על הקפיצה מתבצעת בשלב EXn (המחזור השלישי). אם לדוג' בתרגיל נתון לנו כי ההחלטה על הקפיצה מתבצעת בשלב MEMn (המחזור הרביעי) אז נצטרך למחוק 1-4 = 3 פקודות.

Branch prediction

ע"מ להפחית את ה"הימורים" הלא נכונים, נערוך מעין סטטיסטיקה ובכך ננסה להפחית את ההנחות הלא נכונות של קפיצה בפקודות branch (למשל בלולאה עם 100 איטרציות, נבין כי לאורך זמן מתבצעת קפיצה ובכך נוכל להפחית את ההנחות הלא נכונות ובכך לשפר ביצועים) – מעין "ניחוש מושכל".
עקרון הפעולה לא נלמד במסגרת קורס זה.

סיכום ביניים: מתי נצטרך להוסיף NOPים?

1. **פקודות lw** – נבדוק אם בפקודה לאחר מכן, אחד rsn הוא הרגיסטר שאליו lwn טוען.
אם כן מוסיפים NOP אחד (במידה וקיים forwarding מלא, אם לא נצטרך להוסיף עוד)
2. **פקודות branch** – כאשר עושים "ניחוש" לא נכון בנוגע לקפיצה, נצטרך להוסיף NOPים לפי השלב בו המעבד מחליט על הקפיצה. למשל אם זה בEXE (המחזור השלישי) נצטרך להוסיף $2 = 3 - 1$ NOPים

חישוב משך זמן ריצת תכנית במעבד pipeline

דרך נוחה לחישוב היא לספור את מספר הפקודות, כאשר יש לקחת בחשבון NOPים, השקולים מבחינת זמן הריצה לפקודה נוספת. זה למעשה ייתן את מספר המחזור בו כל פקודה נכנסת לpipe, ולמספר זה להוסיף 4 (הזמן בו לוקח לפקודה האחרונה לצאת מהpipe).

שאלה ממבחן – שאלות 3-4 מתוך מועד א' אביב 2020

בשאלה זו התעלמו מקיום תנאי ה-hold במערכת.

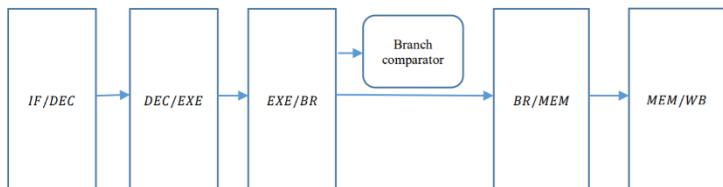
במהלך התכנון של מעבד Pipelined RISC-V, בעל מנגנון forwarding מלא-או-unit detection hazard כפי שנלמדו בכיתה. קבלת ההחלטה על branch מתקבלת בשלב ה-execute.
זמני ה- t_{su} וה- t_{pc-q} של רגיסטר PC זהים לאלו של הרגיסטרים אשר מפרידים בין השלבים.
במעבד נפלה תקלה אשר גרמה להפרה של תנאי ה-setup של משטר הזמנים הדינאמי בשלב ה-execute.
התקלה נגרמת מכיוון שזמן ה- t_{pd} של ה-Branch comp. הוא גדול מידי.

בנוסף, התגלה כי תהליך הייצור גורם להיווצרותו של skew ברגיסטר mem/wb (ביחס לרגיסטרים

Timing	
5ns	Memory access (data or instruction)
4ns	Read/write a value from/to the register file
5ns	ALU operation
7ns	branch comparator
10ns	T (זמן המחזור של המעבד המקורי)
3ns	Skew
3ns	$t_{pcq}(IF/DEC), (DEC/EXE), (EXE/MEM), (MEM/WB), (BR/MEM)$
2ns	$t_{su}(IF/DEC), (DEC/EXE), (EXE/MEM), (MEM/WB), (BR/MEM)$
1ns	$t_{pcq}(EXE/BR)$
1ns	$t_{su}(EXE/BR)$

האחרים אשר מפרידים בין השלבים השונים), וכי לא ניתן למנוע סטייה זו בשום צורה.
בטבלה מתוארים פרמטרי המערכת.
רכיבים אשר זמן ה- t_{pd} שלהם לא צוין הם בעלי זמן t_{pd} זניח.

הצעה 2:



בכדי לפתור את הבעיה הוצעו שלושה רעיונות:

1. הגדלת זמן המחזור של המעבד: $T_{new} = 12ns$
2. הוספת שלב חדש ל-Pipeline בין שלב ה-execute לשלב ה-memory, אשר יקרא נר. רכיב ה-branch comp. יעבור לשלב זה. במידה וישנו צורך ב-forwarding, הוא מבוצע בשלב ה-execute והשלב החדש מקבל את הערכים העדכניים ביותר. התמיכה הרלוונטית לצורך תפקוד תקין של מנגנון ה-forwarding הכללי נוספת גם היא. הרגיסטר exe/br, יפריד בין שלב ה-exe לשלב ה-br ויהיה בעל הפרמטרים אשר צוינו בטבלה. פרמטרי הרגיסטר אשר יפריד בין שלב ה-br לשלב ה-mem, br/mem, זהים לרגיסטרים המקוריים ומופיעים בטבלה. זמן המחזור של המעבד נקבע על פי השינויים. מצורף איור של הצעה זו.
3. השארת המעבד המקורי ללא שינוי, למעט העברה של רכיב ה-branch comp. לשלב ה-memory.

העדיפות העיקרית של מתכנני המערכת הוא latency קצר ורק לאחר מכן throughput גבוה (מקסימלי).
דרגו את ההצעות מהטובה ביותר לטובה פחות לפי עדיפויות אלו.

פתרון

נחשב latency ו throughput של הצעה 1 :

$$\text{latency} = N_{cyc} \cdot T_{cyc} = 5 \cdot 12 = 60$$

$$\text{throughput} = \frac{1}{T} = \frac{1}{12} = 83.33 \text{MHz}$$

נבדוק את הצעה 2 : במערכת מצוננת זמן המחזור נקבע לפי היחידה הצירופית בעלת ה t_{pd} המקסימלי.
נסתכל על הטבלה ונראה כי היחידה בעלת ה t_{pd} המירבי היא ה ALU והגישה לזכרון, בשתייהן : $t_{pd} = 5$,
אולם בעקבות skew של EXE הוא "קריטי" יותר, נקבע מה יהיה זמן המחזור לפי המשטר הדימני :

$$t_{pc-q}(dec/exe) + t_{pd}(ALU) + t_{su}(exe/br) \leq T$$

$$T \leq 3 + 5 + 1 = 9ns$$

נבדוק כעת גם בשלב ה brn החדש :

$$t_{pc-q}(exe/br) + t_{pd}(BrComp) + t_{su}(br/mem) \leq T$$

$$T \leq 1 + 7 + 2 = 10ns$$

ולכן זמן המחזור המינימלי בהצעה זו הוא 10ns.

נחשב latency ו throughput :

$$\text{latency} = N_{cyc} \cdot T_{cyc} = 6 \cdot 10 = 60$$

$$\text{throughput} = \frac{1}{T} = \frac{1}{10} = 100 \text{MHz}$$

נבדוק את הצעה 3, כעת השלב ה"קריטי" הוא שלב ה MEM בוא נמצא ה brn שזמן ההשהיה שלו הגדול ביותר :

$$t_{pc-q}(exe/mem) + t_{pd}(Branch\ Comp.) + t_{su}(mem/wb) \leq T + skew$$

$$T \leq 3 + 7 + 2 - 3 = 9ns \leq 10ns$$

כלומר, ניתן להשאיר את זמן המחזור של המעבד המקורי ללא שינוי.

נחשב latency ו throughput :

$$\text{latency} = N_{cyc} \cdot T_{cyc} = 5 \cdot 10 = 50$$

$$\text{throughput} = \frac{1}{T} = \frac{1}{10} = 100 \text{MHz}$$

לכן לפי סדר העדיפויות שהוצג בשאלה, הצעה 3 היא הטובה ביותר, לאחר מכן הצעה 2 ולבסוף הצעה 1.
התשובה הנכונה היא ג'.

המשך – שאלה 4 :

נתוני שאלה זו זהים לאלו של השאלה הקודמת. כל שלוש ההצעות עובדות תחת ההנחה שפקודות קפיצה אינן מתבצעות, ובמידה ומתגלה כי קפיצה כן צריכה להתבצע מבוצע שימוש במנגנון flush בדומה לנלמד בכיתה.

מהו מספר הפקודות אשר עליהן מתבצע ה flush-במידה ויש בו צורך?

פתרון :

לפי הנלמד בכיתה, מספר הפקודות עליהן מתבצע flush תלוי בשלב בו המעבד מחליט על ביצוע הקפיצה. בהצעה 1 : ההחלטה מתבצעת בשלב ה EXE (המחזור השלישי) לכן צריך לנקות : $3 - 1 = 2$ פקודות. בהצעה 2 : ההחלטה מתבצעת בשלב החדש, Br (המחזור הרביעי) לכן צריך לנקות : $4 - 1 = 3$ פקודות. בהצעה 3 : ההחלטה מתבצעת בשלב ה MEM (המחזור הרביעי) לכן צריך לנקות : $4 - 1 = 3$ פקודות.

טבלת עזר – מעבד pipeline

בלא מעט שאלות יהיה נתון כי אין מנגנון forwarding מלא, ונצטרך להוסיף פקודות NOPs לקוד (באופן "ידני") על מנת לגרום לקוד לעבוד באופן תקין :

<u>מרחק בין הפקודות</u>	<u>מס' הפקודות המפרידות</u>	<u>איור</u>	<u>איזה forwarding מתאים</u>	<u>כמה NOPs צריך אם אין forwarding</u>	<u>חלופות אחרות</u>
Data Hazards – כשרוצים להשתמש במידע שעוד לא התעדכן (לחפש רגיסטר rdx ולאחר מכן אותו אחד בrs)					
1	0		MEM → EXE	3	WB → DEC 1 NOPs 2 - WB → EXE 1 NOP 1 -
2	1		WB → EXE	2	WB → DEC 1 NOP 1 -
3	2		WB → DEC	1	-
4	3	פקודה תרוץ באופן תקין	-	-	-
Load Data Hazard – כשרוצים להשתמש במידע לאחר שנטען lw (לחפש lw בקוד)					
1	0		flush + WB → EXE	3	WB → DEC 1 NOPs 2 - WB → EXE 1 NOP 1 -
2	1		WB → EXE	2	WB → DEC 1 NOP 1 -
3	2		WB → DEC	1	-
4	3		פקודה תרוץ באופן תקין	-	-
Control Hazard – כש"מהמרים" לא נכון אם תקרה הקפיצה (לחפש פקודות branch בקוד)					
נסמן בו את המחזור בו מתבצעת ההחלטה על פקודת branch, למשל n=3 אם ההחלטה נלקחה במחזור השלישי, כלומר בשלב EXE (זהו גם שלב ברירת המחדל אם לא צוין אחרת). נצטרך לבצע flush (או לחילופין להוסיף NOPs) n-1 פעמים. לדוג' עבור מעבד בו ההחלטה על הקפיצה היא בשלב EXE נבצע 2 flush (או נוסיף 2 NOPs).					