

מבנה המחשב | 67200

הרצאות | אוהד פאליד ורונ גבר

כתביה | נמרוד רק

תשפ"ג סמסטר ב'

תוכן העניינים

4	I מבוא ומוליכים-למחאה
4	הרצאה
4	רקע כימי לרנזיסיטורים
5	רקע פיזיקלי לטרנזיסיטורים
5	מוליכים למחאה
6	צומת ח-ק
7	MOS-FET
8	בנייה של שערים מטרנזיסיטורים
9	תרגול
11	II שערים
11	הרצאה
12	סכום מכפלות ומכפלת סכומים
14	יחידות סטנדרטיות בمعالגים לוגיים
15	معالגים סדרתיים
17	معالגים סדרתיים מורכבים על בסיס SR-Latch
18	DFF
18	תרגול
20	מפות קרנו
24	פונקציות שלמות
24	III תזמון מעגלים
24	הרצאה
27	FSM
30	توزון מעבד והגדרות זמני פעולה
34	תרגול
37	MIPS IV
37	הרצאה
38	RISC vs CISC
38	ריגיסטרים ב-MIPS
39	פקודות אРИטמטיות
39	פעולות לוגיות
40	פעולות זכרון
41	פילוסופיות ארגון זכרון
41	פקודות קפיצה והחנויות
42	שימוש פונקציות באמסבל
44	פקודות קראיה לפונקציה
44	תרגול
44	אנליזה של מעגלים סינכרוניים
47	סינטזה של מעגלים סינכרוניים

49	מעבד V Single-Cycle
49	הרצאה
50	קידוד פקודות MIPS
50	יצוע פקודות ב-MIPS
51	שימוש השלבים ב-MIPS
55	תרגול
57	מעבד VI Multi-Cycle
57	הרצאה
59	יחידת השליטה
60	מעבד רב-מחזורי
61	שיטות השוואת יצואו מעבדים
62	איחודי ייחדות במעבד רב-מחזורי
63	מספר המוחזוריים לפי סוג פקודות
67	תרגול
67	דגלי ייחדת הקרה
70	מעבד VII pipelined
70	הרצאה
74	סכנות במעבד pipelined
74	פתרונות לסקנות
76	סקנות ב-branching
77	תרגול
77	שינויים מרכזיים ברכיבים תחת מעבד רב-מחזורי
77	אותות בקרה למრבבים במעבד רב-מחזורי
78	אותות בקרה חדשים במעבד רב-מחזורי
80	הערכת יצואים
81	הטמון VIII
81	הרצאה
82	טרמינולוגיית מטמון
83	מטמון Fully-Associative
84	מטמון Direct-Mapped
85	מטמון 2-Way אסוציאטיבי
85	תרגול

שבוע II | מבוא ומוליכים-למחצה

הרצאה

המחשבים הראשונים היו עצומים, כבדים ויקרות, ויחידת הבסיס של המעבד שלහן היה שפופורת קטודיות (אבי הטרנזיסטור) ואז שפופורת ריק, וכיום משתמש בטכנולוגיית CMOS שמאפשרת גדילה בעשרות סדרי גודל בזמן המוחזר, מהירות השעון, מספר הטרנזיסטורים ועוד. הקורס עוסק במבנה המעבד בעיקר.

בתוך כל מחשב יש אביזרי קלט ופלט (חישוני סונאר, מסך, עכבר), אמצעי אחסון (נדף RAM ולא נדיף כמו דיסק קשיח), מעבד, מערכת הפעלה, דרייברים ותוכנה. בקורס עוסק במעבדים ותוכנה ונזכיר מערכות הפעלה ואמצעי אחסון.

קצב ההתקדמות עד לשנים האחרונות התנהג לפי חוק מור (1965) - כל שנה مضליים להכפיל את מספר הטרנזיסטורים כל שנתיים. העליה במספר הטרנזיסטורים מספקת גם עלייה אקספ' בביטויים, ביעילות אנרגיה וגם בגודל האחסון שאפשר לייצר.

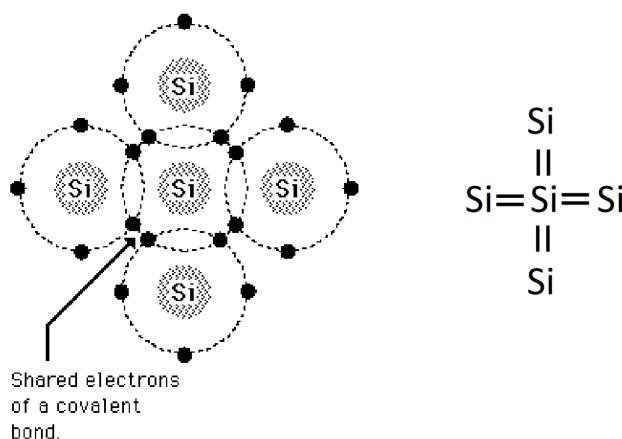
רקע כימי לטרנזיסטורים

המודל של בוחר ניסה להסביר את תופעת התכוונות המשותפות ליסודות באוטה עמודה של הטבלה המוחזרית אותה בנה מנדלייב כמה עשרות שנים קודם לכן. המודל קובע שככל חומר מורכב מאטומים, שלהם יש גרעין עם פרוטונים (חלקיים עם מטען חיובי חיובי), ניטרונים (חלקיים ללא מטען) ואלקטרונים (עם מטען שלילי) ששובבים את הגרעין.

דוגמה סיליקון (צורן) הוא מספר 14 בטבלה המוחזרית, כלומר יש לו 14 פרוטונים (ובמקרה זה גם 14 ניטרונים) וכן אלקטرونים בשכבות שונות סביבה הגרעין עם מספר שונה של אלקטرونים בכל שכבה.

בזה גילה בנוספ' שהמסלול (המעגל, השכבה של אלקטرونים) האחרון של כל אטום במצב יציב הוא מלא, אך אטום ירצה לחת או לחת אלקטرونים מאטומים אחרים כדי לקבל מסלול מלא.

דוגמה הרבה אטומים של סיליקון יוצרים יחד במצב יציב שmorכב משציג אטומי סיליקון עם מסלול אחד מלא לכולם כך שהם חולקים אלקטرونים (ראו איור במקורה של חמישה אטומים)



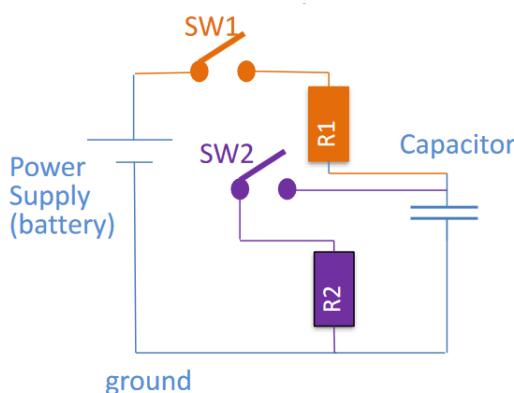
קשר בו אטומים חולקים (sharing) אלקטרונים נקרא קשר קוולנטי (covalent bond).

יונ הוא אטום או מולוקולה עם מספר לא שווה של פרוטונים (+) ואלקטרונים (-) והמשמעות של היחסיק הוא הפרש הערכאים האלו. קשר יוני הוא קשר בין אטומים שנוצר כאשר ממסלול האחרון של אטום אחד לאחר (כדי להשלים מסלול) אבל מספר הפרוטונים באטומים שווה במספר האלקטרונים בכל אטום למעבר האלקטרון. כך לשניים כתה יש מטען מנוגד לשני ומושג ניגודיות המטענים מקרבת (באמצעות כוח משיכה אלקטرون-סטטי) בין האטומים לכדי קשר.

רקע פיזיקלי לטרנזיסטורים

- זרם חשמלי הוא תנועה של אלקטרונים (או באנלוגיה תנועה של חורים, כאשר האלקטרונים השיליליים עוברים בין חורים שהם אחרים חיוביים). בקונבנצייה הזרם נע מה-+ ל-.
- מתוך (פוטנציאל) נמדד בולט והוא היכולת להזירים, כאשר סוללה עצמה מחזיקה מתח וכמו מגדל מים באנלוגיה מים, אפשר לעשות בו חור וכך לגרום לו זרימה אבל כל עוד לא מחברים/מחוררים את הכליל לא יקרה שום דבר.
- מטרון הוא מספר האלקטרונים שמאוחסנים בחומר כלשהו.
- קיבול זו היכולת להחזיק מטען, כאשר בעת אחסון מטען נוצר מתח בקבל (מקביל למיכל מים ולחץ).

דוגמא נביט בمعالג הבא. R1 הוא נגד (שקלול לצינור צר יותר, שיוצר התנדות). אם נסגור את המתג הראשון, נסגור מעגל בין הסוללה לקבל כך שאלקטרונים יזרמו מהסוללה לקבל מלמעלה ומהקבל למים והקבל יקבל את אותו המתח של הסוללה. אם ננטק את המתג הקבל ימשיך להחזיק את המתח, ואם נדליק את המתג השני יהיה לנו זרם קצר מהקבל אליו עם כיוון השעון (בחילוקו העליון של הקבל היונים החיוביים ומתחתיו השיליליים, והאלקטרונים הם אלו שנעים).



בהקשר של מחשבים נקבע מתח נמוך (עד 1.5V) להיות 0 ומתוך גובה (פחות 3.5V) להיות 1 - "כי ככה". בין 1.5V ו-3.5V וולט לא אמרורים להיות במצב יציב, רק במעבר בין המצבים. האנלוגיה כאן היא האם מיכל מים הוא מלא או ריק.

מוליצים למחצה

מוליך למחצה הוא חומר שלא מוליך חשמל מאוד טוב (מוליצותו היא בין חומר לא מוליך לחומר מוליך).

דוגמה סיליקון טהור הוא מוליך למחצה כי בצורת הגביש עליה דיברנו יש מעט מאוד אלקטرونים חופשיים (הרבה מהם תפוסים בין כמה אטומים בקשר קוולנטי) ולכן קשה להזרים דרכו זרם.

אלוח (doping) הוא תהליך שבו "מלככים" את הסיליקון.

• P-type : נוסיף לסיליקון קצת אלומיניום (לו 3 אלקטرونים מ��וק 4 במסלול האחורי) כך שייקשר לאטומי הסיליקון ועתה לחומר יהיה חסר אלקטרון והוא ישmach לקבלו, ואז אלקטرونים יעברו בקבלה בין האי-שלМОיות האלה (פעם ישלים את המחשור במוקד לככלוך אחד, ואז יקפוץ לאחר, וכו'). החורדים שנוצרים כאשר אין את האלקטרון הנדרש ליציבותם נעים (לשם התיאוריה), בכיוון ההופך מהאלקטرونים וכך נוצר זרם חיובי בכיוון ההופך מתנוועת האלקטרונים.

• N-type : העיקרון הנ"ל רק עם זרחן (לו 5 אלקטرونים לעומת 1 במסלול האחורי) כך שייצור עודף אלקטرونים והאלקטرونים העודפים חוזרים לנوع لأن שירצטו וייצרו זרם.

למוליך ממחצה מסוג P ו-N אין מטען חיובי או שלילי אלא רק סיבות שונות לתנוועת האלקטרונים כתוצאה מהרצון להשלים מסלולים אחרים מינים באטומים.

הצמדה של חומרים מסוג P ו-N יוצרת יוניים - האלקטרונים מ-N שמחים לקפוץ ל-P שם "צרי" אותם. שני כוחות פועלים בעת הצמדת חומרים שכזו :

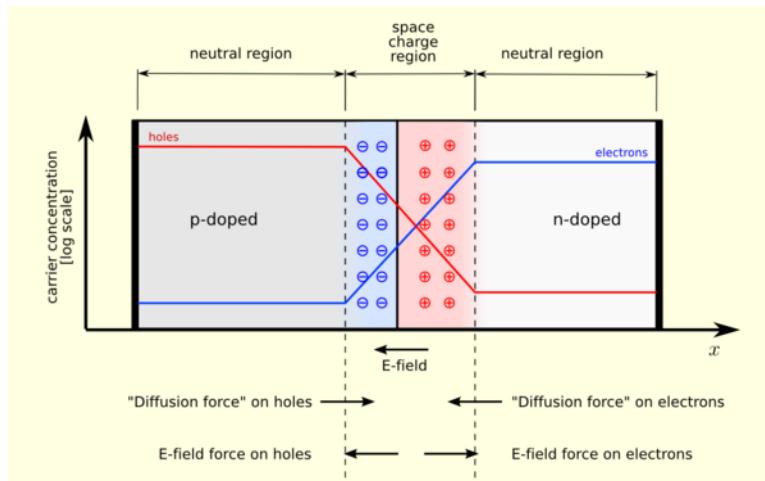
- הרצון של המסלולים להתملא - מה שגורם לאלקטרונים לקפוץ מ-N ל-P.
- הכוח החשמלי שיוצרים האלקטרונים - כוח דחיה בין מטענים שליליים שמונע קפיצה אלקטرونים מ-P ל-N (N אומר "יש לי מספיק שליליים כבר").

צומתpn

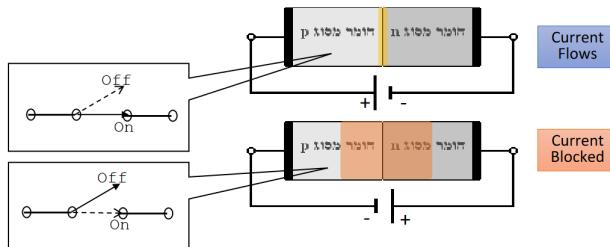
הצמדה זו נקרא צומת PN (PN junction) והוא מוליכה מכיוון אחד וחוסמת זרם מכיוון אחר (כמו שסתום!). המנגנון שהצומת מספק נקרא דיודה (diode) ומסומן באירועים הבאים



בשל תזוזה מקרית של אלקטرونים ופוטוונים בסמוך לצומת, מצלחים לעבר אטומים יוניים שליליים מ-N ל-P וחובבים להפק (בניגוד לכיוון הזרימה). כמשמעותה התחלפו כליה קורות, ישנה מאסה של אטומים יוניים חיוביים ב-N ומאהה של שליליים ב-P שלא יעברו לצד השני בغالל שהם חלק מהגביש כבר. במצב זה נוצר מחסום מכוח כוח הדחיה החשמלי שגורם להפסקת הזרימה דרך הצומת והחזקת מתח בו (ראו איור)



עתה חיבור סוללה לדiode נותן לנו תכונות מעניינות.



- חיבור סוללה עם + ל-P ו- ל-N יגרום להרבה מאוד יוניים חיוביים לזרום מ-P ל-N ויוניים שליליים מ-N ל-P (בהתאם לכיוון הזרימה לפני שנחסמה) וכך יצטמצם המרווח במרכז עד ל-0 ותהייה לו זרימה רגילה, כאילו סגרנו מתג.

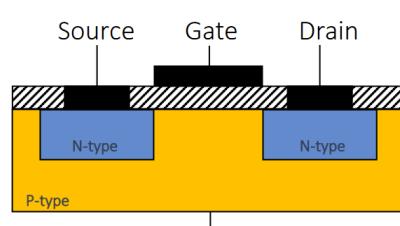
- חיבור סוללה עם + ל-N ו- ל-P יגרום ליוניים חיוביים ושליליים להגיע לחומרים ולהזק את היוניים במרווח שכרען מונעים מעבר ורק יחזקו את החסימה, כך שלמעשה דימינו התנהגות של פתיחת מתג.

MOS-FET

טרנזיסטור MOS-FET עשוי שלושה חומרים: מוליך למחצה; תחמושת מבודדת; ומתקת. יש לו בנוסף שלוש רגליים: source ; drain ; gate (ורgel הארקה שמחוברת תמיד).

N-Channel וורייאנט ה-

באир ניתן לראות NMOS, וריאנט מבין שניים של MOS-FET (השני משללים לו רק Sh-N ו-P במקומות הפוכים). החומר המוקווקו הוא המבודד והשחור הוא המתקת.

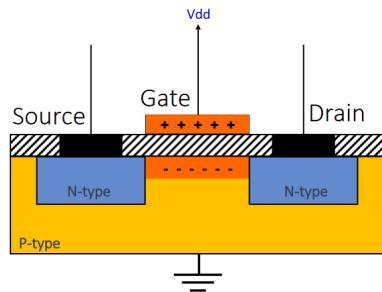


זכור כי זרימה ניתנת לקבל רק מ-P ל-N וכן אי אפשר אף פעם להזרים שום דבר מהמקור (source) לשפק (drain) ולהפך. יש לנו למעשה שתי דיזודות גב אל גב (שקבוצותיהן המקור והשפך).

אם המתכת של השער מקבלת מטען, האзор בין השער למוליך-למחצה נהייה קובל כי הוא מחזיק הפרשי מטעןים!

הערה את כל ארבעת הרגליים תמיד לחבר לאנשאנו - או לאדמה או לסוללה או לטרנזיסטור אחר.

- אם לחבר את השער לאדמה (הארקה), יהיה לנו שני צמחיי-Q-k שלא ניתן יהיה להזרים דרךן (ובפרט בין S ל-D דבר).
- אם לחבר את השער למתח, הקובל שהזכירנו מקבל מתח ועכשו יש מטען חיובי מעליו ושלילי מתחתיו, ככלומר ישנים אלקטرونים חופשיים ב-p-type, וכשהלו נוגעים בחומר type-n משני הצדדים יוצרם ערוץ אלקטронים חופשיים דרךו כן יוכל לעبور זרם, ומכאן השם n-channel.



מה שקיבלו בסופו של דבר הוא סוייז' שאנו יכולים לשלוט בו באמצעות זרם לשער (0 או 1). את הטרנזיסטור נסמן בסימול הבא -
עובד אותו הדבר רק הפוך - הזרמת "0" לשער תסגור את המtag ו-"1" תפתח אותו. ההבדל המהותי היחיד חוץ מהחומרים הוא שיש מתח שמחובר מלמטה במקום הארץקה. להלן טבלת סיכום של דרך הפעולה של הטרנזיסטורים.

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

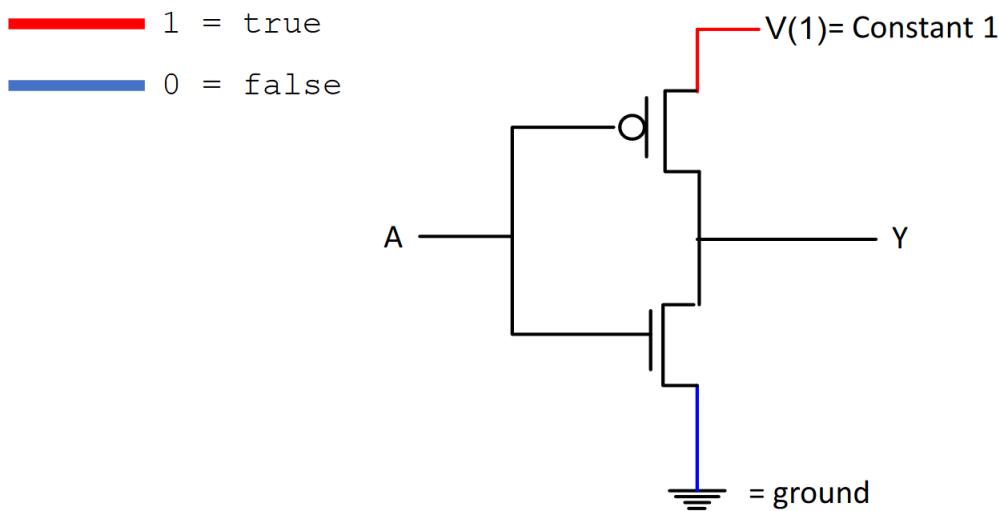
הערה בתחומי הכלל, מסמן שליליה, ב-PMOS למשל "שוללים" את המתח ומתנהגים הפוך ממנו. גם שערי NOT מסומנים עם עיגול.

בנייה שערים מטרנזיסטורים

טרנזיסטורים הם שימושיים לנו כי אנחנו יכולים לבנות מהם שערים לוגיים.

דוגמה נבנה שער NOT (Inverter) שימושו ב-

הבנייה דורשת שני MOF-SET (P-Channel ו-N-Channel) למיטה (למטה), והיא כבאיור.



A הוא הקלט ו- Y הוא הפלט. כשה- A הוא "1", עבר זרם של 0 (שהוא זניח) דרך ה-N-Channel למיטה (השער דלוק) ולא יזרום שום

דבר דרך ה-P-Channel למיטה (השער דלוק لكن אין זרימה). seh"כ אין שום זרימה עם מתח לא זניח لكن Y יהיה "0".

בהתאם אם A הוא "0" אז ה-PMOS יזרם זרם קבוע של "1" ו- Y יהיה "1".

הערה צריך את ה-NMOS התיכון כי אחרת ב-" 1 " $A = 1$ יש לנו מעגל פתוח שיכול להיות לו כל זרם ולא בהכרח "0" כמו שאנו רוצים.

תרגול

בסיס ספירה הוא דרך לייצג מספרים ממשיים. בסיס עשרוני נבעז את ההמרה הבאה

$$(a_4 a_3 a_2 a_1 a_0 . a_{-1} a_{-2} a_{-3})_{10} = a_4 10^4 + \dots a_0 10^0 + a_{-1} 10^{-1} + a_{-2} 10^{-2} + a_{-3} 10^{-3}$$

בסיס בינארי משתמש בסיביות (ביטים) שערך 0 או 1, בעשרוני 0 עד 9, ובhexadecimal 0 – F.

טוחם המספרים בעל n ספרות בסיס r הוא $\{0, \dots, r^n - 1\}$

במקרה הכללי $a_m \dots a_0 . a_{-1} \dots a_{-n}$ בסיס r מייצג את המספר $\sum_{i=-n}^m a_i r^i$

פעולות חישובניות אפשר לשותו באותו האופן כבסיס עשרוני (חיבור ארוך שבו עברת ספרה הלאה, כאשר הספרות נגמרות לא בהכרח אחריו). (9)

מעבר לבסיס 10 הוא די פשוט (פריסת הייצוג וחישוב מכפלות וחיבור בסיס עשרוני). מעבר מבסיס 10 לבסיס אחר הוא פשוט תהליך איטרטיבי של פעולות מודולו (r הבסיס) כאשר מה שהוא לא השארית יהיה הספרה הראשונה (משמאלה), השנייה, וכו' עד שנגמרות הספרות.

לא כלתי כאן אינסוף דוגמאות להמרוות בין בסיסים כי לא מספיק משעם ל.>.

במקרה הכללי נמיר מני בסיסים כלשהם דרך בסיס 10 כאשר את הרכיב משמאלו ומיימן בספרה העשרונית נטפל באופן נפרד וזהה (עד כדי חלוקה איטרטיבית בשמאלו וככפלה איטרטיבית בימין).

דוגמה נמיר את $(0.79272)_{10}$ לבסיס 4. נכפיל את המספר ב-4 ונקבל 3.17088. لكن הספרה הראשונה היא 3 והשארית היא 0.17088. נבצע זאת שוב ועתה נקבל 0.68352 ושארית 0, וחזר חיליה עד שהשארית תהיה 0, כאשר עתה הספרות ה-0 מלמעלה למיטה במקום למיטה למעלה בספרות הרגילים.

שיטות לייצוג מספרים

- **שיטת גודל וסימן**: מספר יתחל בביט סימן (0 חיובי 1 שלילי) ושאר הביטים יהיו ייצוג בינארי של המספר.

$$\text{דוגמה } 9 = (-1)^0 (2^3 + 2^0)$$

טוחה הייצוג בשיטה זו הוא $(2^{n-1} - 1), \dots, 0, \dots, (2^{n-1} - 1)$

• **שיטת המשלים אחד**: בית סימן, שלפי ערכו נדע אם שאר הספרות הן ערך הבינארי של המספר זהה, או שזו הערך הבינארי של המשלים של המספר $-1 - (2^{n-1} - 1)$, ובנוסף הפיכת כל בית תספק לנו את השילילה של המספר. לדוגמה, $4 = 00100$, ואם נהפוך כל בית נקבל $4 = 11011$.

חישור מספרים הוא די פשוט: צריך לחבר את המספרים, ואם יש carry לאחר החישוב נמתק אותו ונוסיף אחד לתוצאה.

$$\text{דוגמה } 5 = 9 - 4 = (+9) + (-4) = 01001 + 11011 = 100100$$

טוחה הייצוג הוא כמו בשיטת גודל וסימן ויש בו, כמו בשיטה הקודמת 0 חיובי ו-0 שלילי.

- **שיטת המשלים לשתיים**: בית סימן, שעבור מספרים שליליים ערכו הנגדי למספר שמתתקבל מהיפוך הביטים והוספה אחד.

$$\text{דוגמה } -4 = -(00011 + 1) = -00100$$

לחולופין לחישוב המשלים אפשר ללקת מימין לשמאל עד ל-1 הראשון, לא לשנות אותו, ואז להפוך את כל מה שימושאלו (ואז לא צריך להוסיף 1).

$$\text{דוגמה } 5 - 5 = 00101 - 00101 = 10110$$

כדי לחסר מספרים, נסכום אותם ונמתק carry אם יש.

הגדירה overflow הוא מצב שבו אנחנו סוכמים שני מספרים באותו הסימן ומקבלים מספר בסימן הפוך, במקרה זה התוצאה כמובן שגויה.

דוגמה נשתמש בשיטת המשלים ל-1 עם 5 ביטים, $01011 + 01101 = 11000$ קלומר סכמנו חיוביים וקיבלו תוצאה שלילית!

הערה הפתרון overflow הוא הוספה ביטים כך שיגדל טוחה הייצוג.

שבוע III | שערים

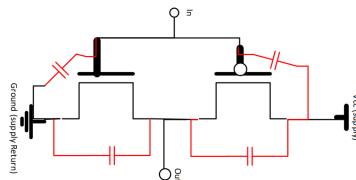
הרצאה

הערה כל שער שנבנה נבנה סימטרית מבחינת השימוש ב-PMOS ו-NMOS, אך השערים עם טרנזיסטורים אלה נקראים .MOS

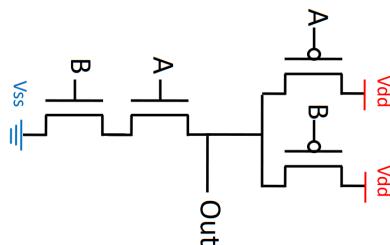
הערה לעיתים לא נרצה להזרים זרם אל תוך טרנזיסטור אחר דלوك (מהכיון הלא נכון) כי זה גורם לסתור.

שער אחד בפני עצמו זה נחמד, אבל מעבדים בונים ע"י חיבור שערים. את השערים נחבר עם חומר מוליך בין הקלט של שער אחד לפולט של השער שמננו אנחנו לוקחים את התוצאה.

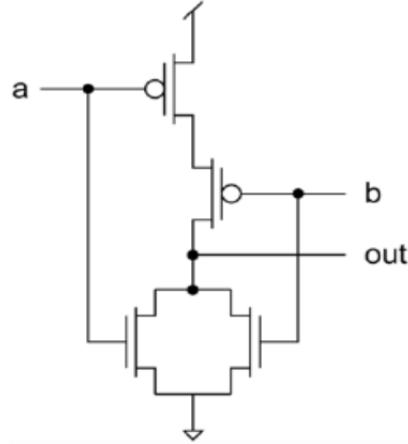
בין שער למקור, בין מקור לשפק ובין שער לשפק נדרשים קבלים אפקטיביים (לא הוספנו שום דבר, פשוט יש רכיבים פיזיים שמוצאים עצמם מחזיקים מתח בין הצדדים). את הקבלים האלה לוקחים מפרק מתח בעת שנייה מצב (שינוי הזרם מהשער, הזרם מהמקור). לכן במצבים רגילים יש זרימה שאנחנו לא בהכרח מצלפים לה במהלך המעבר ($10\text{--}15\text{ }\mu\text{s}$ שנייה). ראו באירור באודם את הקבלים שנדרדים.



השער היסודי שמאפשר לבנות את כל שער השערים הוא NAND (Not And) - שנותן 0 אם "ס" שני הפלטים 1 (ואחרת 1), ומסומן כך \overline{AB} . ניתן לבנות את NAND עם CMOS באמצעות הבאה (הריצו פלטים כדי לראות שזה עובד).



בדומה ניתן לבנות שער NOR באמצעות הבאה (קו אלכטוני הוא 1) V GND (0) קלומר הארקה (0)



הגדירה פ' בוליאנית היא פ' שמקבלת קלטיים בוליאניים (משתנה שיכول לקבל אחד מבין שני ערכים, ביטים לדוגמה) ופולטת פלט בוליאני אחד בדיקוק.

הערה מעתה נסמן x' להיות החפה ל- x (כלומר שהפעלו עליו שער NOT).

דוגמא $F = xy'$ היא פ' שבהינתן y, x , פולטת פלט שערכו x וגם לא y .

סכום מכפלות ומכפלת סכומים

הגדירה בהינתן משתנים בוליאניים לפ' בוליאנית כלשהי, minterm הוא מכפלה (AND) של ליטרלים, כאשר ליטרל הוא משתנה או היפוכו וכל משתנה מופיע בדיקוק פעם אחת כליטרל או בתוך ליטרל מהופך.

דוגמא עבור שלושה משתנים וההשמה $x = 1, y = 1, z = 1$ המינטרם היחיד שערךו 1 יהיה $m_6 = xyz'$, ועבור $x = 1, y = 1, z = 0$ המינטרם היחיד שערך 1 מתקבל ע"י הערך הדצימלי של הסטריאג הבינארי המתקבל מהצמדת ההשומות (במשתנים).

נשים לב שש-minterm מקבל ערך 1 בבדיקה שורה אחת של טבלת אמת מלאה על המשתנים.

הגדירה בהינתן השמה במשתנים, maxterm הוא סכום (OR) של המשתנים או היפוכיהם כך שכל משתנה מופיע פעם אחת בדיקוק.

דוגמא עבור שלושה משתנים עם ההשמה $x = 1, y = 1, z = 0$ המינטרם היחיד שערך 0 הוא $M_6 = x' + y' + z$ וכן.

הערה משלים m_i ל- M_i .

הגדירה Sum of Products הוא סכום מכפלות minterm-ים ו-Product of Sums הוא מכפלת maxterm-ים.

דוגמא הפ' F_1 עם טבלת האמת הבאה

x	y	z	F ₁	Minterm	Maxterm
0	0	0	0	m ₀ =x'y'z'	M ₀ =x+y+z
0	0	1	1	m ₁ =x'y'z	M ₁ =x+y+z'
0	1	0	0	m ₂ =x'yz'	M ₂ =x+y'+z
0	1	1	1	m ₃ =xyz	M ₃ =x+y'+z'
1	0	0	1	m ₄ =xy'z'	M ₄ =x'+y+z
1	0	1	0	m ₅ =xy'z	M ₅ =x'+y+z'
1	1	0	1	m ₆ =xyz'	M ₆ =x'+y'+z
1	1	1	0	m ₇ =xyz	M ₇ =x'+y'+z'

ניתנת לייצור ע"י

$$F_1(x, y, z) = m_1 + m_3 + m_4 + m_6 = x'y'z + x'yz + xy'z' + xyz'$$

הטכנית היהתה לסקום את כל ה-minterm-ים שמקבילים 1 בפ' (בכלוח).

לחולופין נוכל לייצג את הפ' עם PoS ע"י הכפלת כל ה-maxterm-ים שמקבילים ערך 0 (באדום) כולם

$$F_1(x, y, z) = M_0 \cdot M_2 \cdot M_5 \cdot M_7$$

כולם הצנו בצורה קוננית פ' לכאהורה מורכבת!

הערה למה משתמשים כל הייצוגים השונים (טבלת אמת, PoS, SOP ופתח קרנו שנלמד בתרגול)? נרצה בסופו של דבר את הייצוג המינימלי, כך שנדרש למספר הקטן ביותר של שערים כדי למשמש אותן.

דוגמה מולטייפלסקר (MUX) מקבל שלושה קלטים : S(*elect*), D0, D1 . בטבלה X משמעו "לא משנה" מה הערך"

S	D1	D0	Y
0	X	0	0
0	X	1	1
1	0	X	0
1	1	X	1

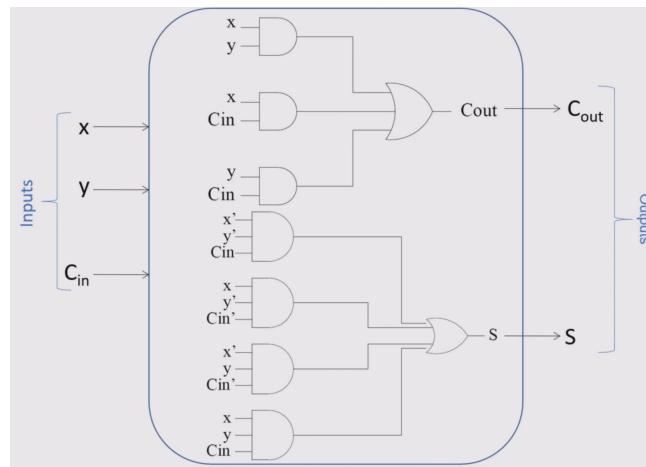
וניתן לרשום את הפ' כ- $MUX(S, D_0, D_1) = S \cdot D_1 + S' \cdot D_0$, ואת זה ניתן למשם באמצעות שערים שכבר בנינו. עם זאת, ניתן למשם את המעגל עם פחות טרנזיסטורים מאשר בIMPLEMENTATION-USING-NOT-AND-OR-NOT.

דוגמה הוא מעגל שמקבל C_{out} (כאשר S, C_{in} נשא מסכימה קודמת) ופולט x, y, C_{out} כאשר S הוא הסכום והוא הנשא (overflow)

Truth Table					
x	y	C _{in}	S	C _{out}	
0	0	0	0	0	
0	0	1	1	0	
0	1	0	1	0	
0	1	1	0	1	
1	0	0	1	0	
1	0	1	0	1	
1	1	0	0	1	
1	1	1	1	1	

ניקח את $S = 0, C_{out} = 1$ $x + y + C_{in} = 0 + 1 + 0 = (10)_2$, או $x = 0, y = 1, C_{in} = 1$

למעשה, בגלל שיש למעגל שני פלטים, הרי שהוא מורכב משתי פ' בוליאניות. כרגע אפשר למשת את המעגל באמצעות SoP (סכימות minterm-ים), ובמימוש הבא צמצמו כמה minterm-ים באמצעות מנות קרנו שלמדו בהמשך (בנוסף מופיע במעגל OR עם שלישה וארבעה כניסה - הסתודנטית המשקיעה תראה כיצד ניתן לעשות זאת עם פי 2 טרנזיסטורים ממספר הכניסות).



הערה קל לשדרר כמה FA-ים כדי לקבל מעגל שסוכם מספרים עם מספר ביטים גדול יותר (לוקחים את C_{out} של ה-FA על זוג הביטים הראשונים, מכנים אותו ל-FA ווחזר חלילה).

יחידות סטנדרטיות במעגלים לוגיים

1. **מפענה**: הקלט הוא n ביטים d_0, \dots, d_{k-1} כאשר $k = 2^n$ והפלט הוא x_0, \dots, x_{n-1} (Decoder).

$$d_j = \begin{cases} 1 & (j)_{10} = (x_{n-1} \dots x_0)_2 \\ 0 & \text{אחרת} \end{cases}$$

כלומר פורש וקטור של n ביטים על 2^n ביטים שכל אחד מייצג מספר מתוך $\{0, \dots, 2^n - 1\}$.

ברגע שיש לנו בлок של מפענה, אפשר להשתמש בו כדי למשוך פ' בוליאנית באופן טריוויאלי, כי כל מה שצריך לעשות זה לעשות OR על כל d_i -ים שמייצגים x_0, \dots, x_{n-1} שמקבל ערך 1 בטבלת האמת של הפ' הבוליאנית.

2. מפלג (DeMultiplexer) : הקלט הוא f_0, \dots, f_{k-1} כל אחד בגודל בית והפלט הוא x, s_0, \dots, s_{n-1} כאשר $k = 2^n$ ו $x = (s_{n-1} \dots s_0)_2$.

$$f_j = \begin{cases} x & (j)_{10} = (s_{n-1} \dots s_0)_2 \\ 0 & \text{אחרת} \end{cases}$$

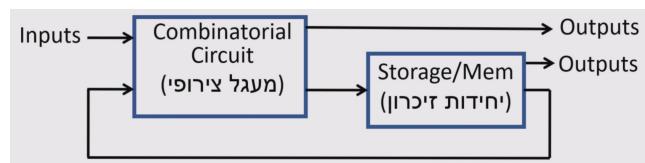
כלומר בהינתן בית, מחזיר מחרוזת שבה הכל אפסים חוץ מאולי הבית ה- $(s_{n-1} \dots s_0)_2$ שערכו x .

3. מקודד (Encoder) : הקלט הוא ביטים x_0, \dots, x_{k-1} כאשר $k = 2^n$ והפלט הוא ביטים e_0, \dots, e_n כאשר אם $x_i = 1$ עבור i אחר בדיקות (וכל השאר אפסים) אז $(e_{n-1} \dots e_0)_2 = (i)_{10}$, כלומר מחזיר את האינדקס (בבינה-ארית) של הבית היחיד הדלוק בקלט.

4. מרובב (Multiplexer) : הקלט הוא $k = 2^n$ ביטים s_0, \dots, s_{n-1} וביטים x_0, \dots, x_{k-1} והפלט הוא f שהוא ערך הבית עם אינדקס $.x \cdot (s_{n-1} \dots s_0)_2$.

מעגלים סדרתיים

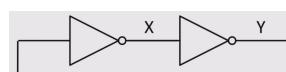
הגדרה מעגל סדרתי הוא מעגל קומבינטוררי שהקלטים שלו הם פלטיהם של יחידת זיכרון שמחזיק השער (ראו איור)



מעגלים סדרתיים אסינכرونנים יכולים לשנות מצב בכל זמן, ואילו מעגלים סינכרוניים משנים מצב בהתאם לשעון.

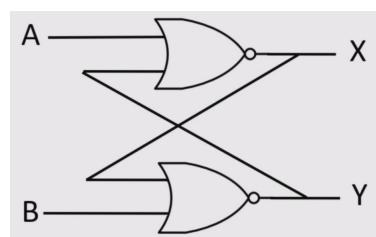
הגדרה שעון סיגナル בצורה גל מחזורי (או 1 ואו 0 ואו 1 וכו').

דוגמה כיצד השער הבא יתנהג?



אם הקלט בהתחלה הוא $X = 0, Y = 1$ או $X = 1, Y = 0$ ונקבל מצב יציב של $(X, Y) = (0, 1)$. בדומה לעבורנו. ככלומר, יש לנו מערכת דו-יציבה (עם שני מצבים יציבים), שיכולה לשמש אותנו לאחסון זכרו.

דוגמה נבית בשער הבא, שנקרא SR Latch (נזכיר שהשערים במעגל הם NOR-ים)



הפ' הזו מקיימת $X(t+1) = (A + Y(t))'$, $Y(t+1) = (B + X(t))'$ לאחר שינוי קלטים מסווגים עם שעון שביצע t טיקים), או בטבלת אמת עמוסה

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

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

ואם נשלוף רק את הערכים המעניינים, נוכל להביט בתופעה מעניינת (חצים מעגליים משמע לאחר טיק נשאר באותו מצבים וחיצים אומרים שנעבור לזוג ערכים אחר)

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

Stable Clear (Y) Set (Y) Undefined

ונשים לב שאם $X = Y$ נקבל מצב לא יציב (לא משנה מה ערכי A, B תמיד יש חץ שיוציא אותנו למצב אחר) ולכן תמיד נניח שאנוחנו משתמשים ב-SR Latch כאשר $Y = X'$ (זה דומה למצב בדוגמה הקודמת שבו $Y = 0$ שווה לא מוגדר בכלל כי יש לנו מהפץ עם אותו הערך משני הצדדים).

אם כן, עבור $(X, Y) = (0, 0)$, נקבל ש-(X, Y) שומר על ערכו (כזכור אנחנו מתעלמים מ- $M-Y$), ואם $(X, Y) = (0, 1)$ או $(A, B) = (0, 1)$ מפעילים את Y , ואם $(A, B) = (1, 1)$ או מפעילים את Y ואם $(A, B) = (1, 0)$.

לכן, באמצעות $(S, R) = (A, B)$ נוכל לשולוט בערכו של Y כישיש לנו זכרוں של המצב הקודם, עם טבלת אמת מצומצפת נוחה ביותר, כאשר $Q(t) = Y(t) = X(t)'$ (בהתעלם מהמקרים הלא חוקיים)

S	R	$Q(t+1)$	$Q'(t+1)$
0	0	$Q(t)$	$Q'(t)$
0	1	0	1
1	0	1	0
1	1	0	0

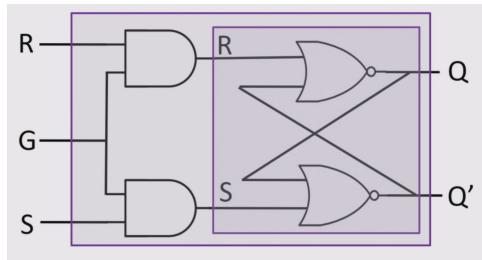
הערה מעגל סדרתיים ניתן לייצג באמצעות טבלת עירור (הטבלה הנ"ל) וטבלת מעברים, שועונה על השאלה "אילו קלטים נדרשים כדי לעבור בין מצב כלשהו לאחר" (Φ הוא ערך Don't care)

From Q(t)	To Q(t+1)	S	R
0	0	0	Φ^+
0	1	1	0
1	0	0	1
1	1	Φ	0

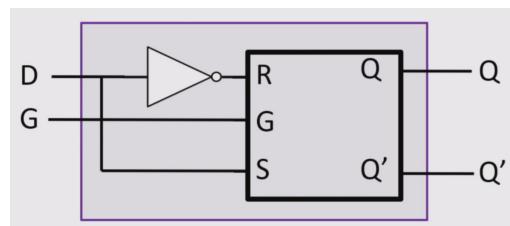
הערה יש היגיון בשמות הביטים S,R - אם רק (et) S דלוק, קובעים את הפלט להיות 1, אם רק (eset) R דלוק קובעים את הפלט להיות 0, ואם לא זה ולא זה דלוקים לא עושים כלום, כלומר שומרים על המצב כמוות שהיה. כמובן שתחת סימולדים אלה, גם S וגם R דלוקים זה לא מוגדר היטב.

מעגלים סדרתיים מורכבים על בסיס SR Latch

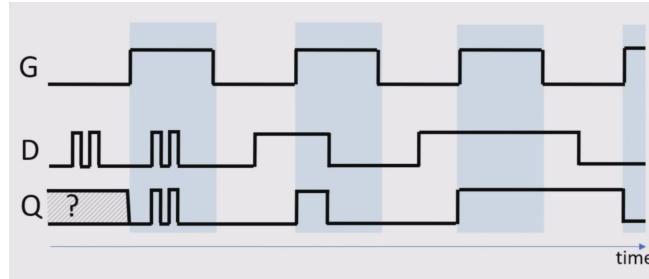
- **Gated SR Latch** הוא שער שמנוע פליטתית ערכיים לא חוקיים מ-S,R, והוא מקבל קלטיים S,R כאשר G הוא בית שער שאם הוא דלוק אז המעגל מתפרק כ-Reg, ואם כבוי אז הפלטים לא משתנים לא משנה מהו. המימוש הוא די פשוט, כולל AND של G עם S,R לפני הכניסה למעגל הפנימי (ראו איור)



- **Gated D-Latch** הוא גרטה אפילו יותר בטוחה ל-Gated SR Latch - במקומם לקבל קלטי S,R, מקבל D שייהיה מחובר ל-S ודרך מהפץ ל-R, וכן לא מקבל מצב של (1,0) וכדי לקבל (0,1) אפשר פשוט לכבות את G (ראו איור)



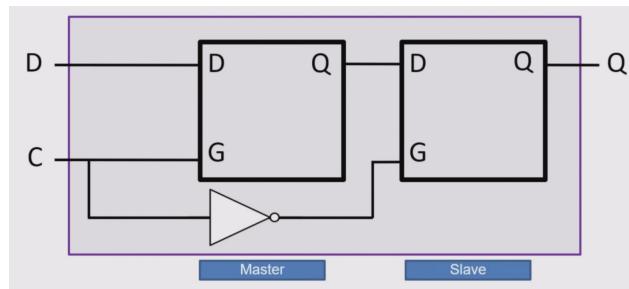
הערה לרוב נחבר את השעון ל-G, כלומר אפשר לשנות את הערך רק כשהשעון בטיק שערכו 1. דוגמה בהינתן שעון שמחובר ל-G ובית D שערכו משתנה, יוכל לחשב מה יהיה הפלט Q, כפ' של הזמן. השיטה המקורית בהתחלה פירשו שלא ידוע לנו מה הערך של Q שכן הוא לא מוגדר בשלב הזה.



D Flip Flop

בנה מעגל Shifter, שהוא מעגל שבו בית הקלט Z צעד אחד ימינה בכל מחזור. אם נשרשר D-Latch-G-ים שכלה-G-ים שלהם מחוברים לשעון, הקלט יופיע מיד בסוף השרשור (למעט זמן פגיעה של החישמל שהוא זניח).

הפתרון זהה הוא להוסיף מהפץ בין השעון ל-Latch השני כך שכשהשעון ב-1 רק ה-Latch הראשון יוכל לשנות את ערכו וכשהשעון ב-0 רק ה-Latch השני ישנה את ערכו והראשון לא ישנה. שער כזה נקרא .D Flip Flop



לכן רק בעת ירידת השעון מקבל שינוי של הפלט Q , כי לאחר העליה ה-Master ישנה את הפלט ולאחר הירידה ה-Slave ישנה את הערך, הלא הוא פلت המנגנון כולו. כל עוד השעון לא יריד, הערך ישאר זהה לערך שקיבל בירידת השעון האחורונה.

הערה ניתן לבנות מעגל שישתנה רק בעליה באמצעות הזזת המהפק לפניו השער של המאסטר ולא העבודה.

תרגול

הגדרה אלגברה בוליאנית היא מבנה אלגברי המוגדר על קבוצת איברים B עם שני אופרטורים בינאריים $\cdot, +$, $x, y, z \in B$ מתקיימות אקסiomות Huntington: סגירות לכפל וחיבור, קיום איברי ייחידה לכפל וחיבור, קומוטטיביות בחיבור וכפל, דיסטרויטיביות (שני הנוסחים), קיום משלים (כך $x \cdot x' = 0, x + x' = 1$) ו- $|B| \geq 2$.

הגדרה אלגברה בוליאנית דו-ערכית מוגדרת על קבוצה בת שני איברים $B = \{0, 1\}$ עם האופרטורים $x \cdot y = \text{AND}(x, y), x + y = \text{OR}(x, y)$ ו- $x' = \text{Not}(x)$

טענה באלגברה בוליאנית דו-ערכי מתקיימות התכונות הבאות:

- אידempotentיות: $x \cdot x = x + x = x$ לכל x .

- $x \cdot 0 = 0, x + 1 = 1$

- אסוציאטיביות לחיבור וכפל.

- חוק הצמצום : $x(x+y) = x, x+xy = x$

- $(x')' = x$

הערה סדר האופטורים בחישוב ביטויי בוליאני הוא קודם סוגרים, אז NOT, אז AND ואז OR.

דרכים לבטא פונקציה בוליאנית

- ביטוי בוליאני (ביטוי על המשתנים עם שני האופרטורים ושלילה).

- טבלת אמת : לטבלה יהיו 2^n שורות כאשר יש n קלטים.

- סכום מכפלות : מספיק שמכפלה אחת תהיה 1 כדי שהסכום יהיה 1. כדי להציג סכום מכפלות, סוכמים את כל המכפלות הסטנדרטיות (minterm) שמקבלות 1 בטבלת האמת.

המשתנה ה- j מקבל שליליה ב-minterm ה- i אם "ס הבית ה- j -ב- $_{(2)}^i$ הוא 0.

- מכפלת סכומים : מספיק שסכום אחד יהיה 0 ואז כל המכפלה היא 0. כדי להציג סכום מכפלות עם שליליה על הביטוי ושימוש בשני כללי דה-מורגן.

המשתנה ה- j מקבל שליליה ב-maxterm ה- i אם "ס הבית ה- j -ב- $_{(2)}^i$ הוא 1.

נרצה לצמצם את מספר הליטרלים שלנו (מספר השערים).

דוגמה נביט ב- $F(x, y, z) = xy' + x'z - F_2(x, y, z) = x'y'z + x'yz + xy'$. את הפ' השנייה ניתן למש עם משמעותית פחות שעריםalogisms (יש הרבה פחות פעולות) אבל הפ' שקוילות ולכן את השנייה תמיד!

את השקוילות אפשר להראות עם טבלת אמת או באמצעות אלגברה בוליאנית :

$$\begin{aligned} xy'z + x'yz + x' &= x'zy' + x'zy + xy' \\ &= x'z(y' + y) + xy' \\ &= x'z + xy' \\ &= F_2(x, y, z) \end{aligned}$$

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

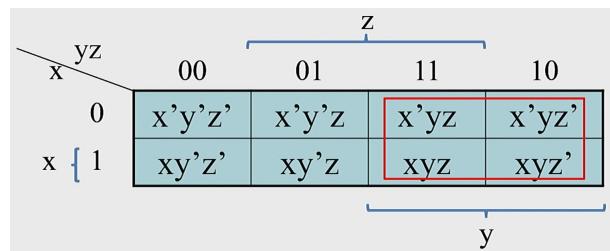
מפות קרנו

מפת קרנו בונים פעם אחת לכל הפ' הبولיאניות ב- n משתנים. ל- n משתנים יש מפת קרנו עם 2^n משבצות.

ראשית נבנה טבלת אמת לפ' הפוליאנית, ואז נציב את ה-minterm-ים בשבלונה של מפת הקרנו אם נצליח לשன אותה.

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

לחולופין נוכל לבנות מחדש את הטבלה עם האינטואיציה שבסיסנים הכהולים למשתנים, באמצעות ערכי x ו- yz ניתן להסיק בקלות את המinterm-ים (יש לשים לב שערכי yz הם לא לפי סדר לקסיקוגרפי)



הערה ארבעת המשבצות ש- z מסומן עליהם נסכום לליטרל ייחיד שהוא z , כך גם על y , וכך גם השורה התחתונה נסכמת ל- x . נשים לב גם כי כל שני ריבועים סמוכים נבדלים בליטרל אחד בלבד.

כדי לבצע צמצומים נמצא קבוצות של ריבועים סמוכים שערכם בטבלה האמת 1 עבור הפ', כאשר הקבוצה חייב להיות חזקה של 2 (כולל 1) ועלינו לבחור קבוצות גדולות ככל האפשר. קבוצות יכולות לחפות וצריך לכסות את כל הריבועים. הטבלה היא מעגלית ולכן ניתן לבצע חצחות את הקצה מימין ולהמשיך ממשאל.

דוגמה עבור $F(x, y, z) = \sum(2, 3, 4, 5)$ (סכום מכפלות עם אינדקסים). מפת קרנו שלנו היא הבאה, כאשר הגענו אליה או באמצעות השבלונה או באופן הבא: הסימונים של x, y, z אומרים לנו Aiife המשטנה מקבל ערך 1, ולכן במשבצת השנייה מימין בשורה העליונה לדוגמה, גם y וגם z הם 1 אבל x הוא 0, כלומר מדבר במקרה של (0, 1, 1) שבקורה שלו זה 1 (כי זהו ערכו של המinterm השלישי). מהגדרת הפ' הוא 1).

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

כדי ליצנס את הtablulation עכשו נcosa את הtablulation עם שני מלבים, אחד משמאלי למיטה ואחד מימין למעלה וכן קיבל ייצוג מינימלי של הפ' הבוליאנית שהוא $F(x, y, z) = x'y + xy'$ (בלבן משמאלי משותף הכל חוץ מ- z וכן גם בסמלון מימין).

דוגמה ($F(x, y, z) = \sum(0, 2, 4, 5, 6)$). הביטוי הלא מצומצם מכיל 5 ביטויים. מלאו אייכשו את הtablulation ונקבל

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

נשתמש בחפיפות וגם במעגליות וכן מקבל מלבן אחד משמאלי למיטה ועוד מלבן שככלו את העמודה השמאלית והעמודה הימנית יחד, ואז הביטוי המינימלי הוא $F(x, y, z) = xy' + z' + y$ (בשמאלי למיטה נופל z ובמעגלי נופלים x ו- y , פשוט עוברים על זוג ריבועים אופקיים זוגiani ורואים מה משתנה בהם, ומה משותף בהם).

מפת קרנו באربעה משתנים נראית כך, כאשר אפשר לזכור אותה באמצעות העובדה שערכי הציריים שלה (גם אופקיים וגם אנכיים) הם 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 (היצוג הבינארי של המספרים).

		00	01	11	10
wx	00	0000	0001	0011	0010
	01	0100	0101	0111	0110
	11	1100	1101	1111	1110
	10	1000	1001	1011	1010

דוגמה ($F(x, y, z) = (0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$) מקבלת מפת קרנו עם הערכים הבאים (לא משנה איך מגיעים לזה)

נבחר את חצי המפה השמאלית כי זו שמנינה והמלבן הגדול ביותר שיש, ובשביל הערכים מימין נבחר שתי רכיביות מעגליות (אי אפשר את כולם ביחד כי זה נותן לא חזקה של 2).

החצי השמאלי נבדל ב- w , z , x , ו- y מקבל 0 כלומר הביטוי הראשון הוא y' .

הרכיביה המעגלית העליונה נבדلت ב- x ו- y ו- w , z מקבלים ערכים 0 שניהם, כלומר יש לנו $w'z'$ והרכיביה המעגלית התחתונה שונה ב- w ו- y ו- x ו- z מקבלים ערכים 1 ו-0 בהתאם, כלומר הביטוי השני הוא xz' .

$$\text{סה"כ קיבלנו } F(x, y, z, w) = y' + z'w' + xz'$$

הأدירה לפעמים עבור צירופים מסוימים, לא יהיה אכפת לנו מה הוא פלט הפ', צירופים כאלה נקראים **צירופים אדישים** ונitin להשתמש בערך שיותר נוח לו איתו כך שיבוטלו ליטרלים רבים ככל האפשר. נסמן צירוף כזה ב- \emptyset .

דוגמה ((1,1,1)) ($F(x, y, z) = \sum (0, 2, 4, 5, 6)$) (כלומר רק).

עתה נוכל לבחור מלבנים יותר נוחים (ראו או איור) מאשר בהיעדר הצירוף האדיש כי אז לא היינו יכולים לבחור את השורה התחתונה כמעט והיו לנו אמנים עדין שני ביוטוים אבל עם יותר ליטרלים, כלומר יותר שערירים שזה פחות טוב.

הערה צירוף אדיש מתקיים לדוגמה כשברכיב אלקטронאי איזשהו מעגל בכל מקרה מחובר להארקה כך שלא משנה ערכו עבור צירוף מסוים כלשהו.

דוגמה נביט במפת הקרן הbhא עם שני צירופים אדישים. הבחירה הכח נוחה היא 0 לשמאלי ו-1 לימני כי כל קומבינציה אחרת הייתה דורשת מאייתנו יותר משני מלבנים או מלבנים קטנים יותר (יותר ליטרלים)* שזה פחות אידיאלי.

דוגמה אפשר להשתמש בມפה קרנו גם כדי לציגם מכפלה של סכומים. נביט ב-(5) $F(x,y,z) = \sum (0,1,2,3,4,6,7) = \Pi(5)$. כפי שניתן למטה, נמצא של סכום המכפלות נדרש לפחות 3 נסכומים ואילו מכפלה סכומים דרוש לבדוק ליטרל אחד.

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

בשאלה פ' בוליאנית לפי מכפלה סכומים, מבצעים בדיקת התהיליך של סכום מכפלות רק שטחים 0-ים במקומות 1-ים. לאחר מכן, ממירמים את הכיסויים למכפלה של סכומים, כאשר כל סכום מתאים למבחן אחד.

הערה ניתן להוכיח נכונות של צמצום לפי מפה קרנו למכפלה סכומים או באמצעות דה-מורגן לצמצום של סכום מכפלות, או פשוט באופן ישיר מطلب האמת ונכונות ייצוג המינימום-טרמינום.

דוגמה נתונה הפ' הבוליאנית

$$F(w,x,y,z) = \sum (1, 2, 3, 11, 12, 13, 15) + d(w,x,y,z), \quad d(w,x,y,z) = \sum (5, 9, 10, 14)$$

- ציררו את מפה קרנו עבור הפ' F .

המפה וראה כך

w\x\y\z	00	01	11	10
00	0	1	1	1
01	0	∅	0	0
11	1	1	1	∅
10	0	∅	1	∅

- כמה פ' שוניות מיוצגות עליה המפה?

$2^4 = 16$ כי אפשר לבחור ערכי שרים שונים עבור כל אחד מהצירופים האפשריים שהוא ב"ת באחרים.

- רשמו סכום מכפלות מינימאלי עבור F , האם הסכום ייחיד?

הכי נוח יהיה שהשורה השנייה תהיה כולה 0 והשלישית כולה 1, ובשורה הרביעית כמה שיותר 1-ים כדי שנקבלים מבנים של רביעיות במקומות זוגות. כך נקבל סה"כ את הכיסוי הבא

w\x\y\z	00	01	11	10
00	0	1	1	1
01	0	∅	0	0
11	1	1	1	∅
10	0	∅	1	∅

שנותנו לנו את הביטוי $z'x'y + x'y + x'w + w$. זה לא ביוטי מינימלי כי אפשר לבחור שכל הצירופים האדיישים יהיו 1 חוץ מ-(0, 0, 1, 1).

ואז קיבל את הכיסוי הבא עם אותו מספר ליטרלים

	wx		yz	
	00	01	11	10
00	0	1	1	1
01	0	Ø	0	0
w	11	1	1	Ø
10	0	Ø	1	Ø

פונקציות שלמות

הגדירה קבוצה F של פ' בוליאניות נקראת שלמה אם ניתן למשת כל פ' בוליאנית בעזרת פ' בקבוצה.

משפט $\{ \cdot, \cdot', +, \cdot', \cdot'' \}$ היא שלמה.

■ **הוכחה:** כל פ' בוליאנית ניתנת להציג כסכום מכפלות שדורש רק את שלושת הפעולות הללו.

מסקנה $\{ \cdot, \cdot', +, \cdot', \cdot'' \}$ הן שלמות.

■ **הוכחה:** עם דה-מורגן אפשר ליצר $+$ עם \cdot' , ו- \cdot עם \cdot' .

דוגמה NAND, קלומר $(y \cdot x)'$ היא פ' שלמה. ראשית ניתן להשיג NOT (\cdot') באמצעות AND-ו- \cdot ($x \cdot x' = x'$). לכן ניתן להשיג באמצעות NOT על NAND, שניהם כבר ייש לנו.

דוגמה הוכיחו כי $f(x, y, z) = x' + yz$ הוא קבוצה שלמה.

ראשית ל-NOT ניתן להגיע באמצעות $x' = f(x, 0, 0)$. f(x, 0, 0) = $x' + 0 \cdot 0 = x' + 0 = x'$. f(x, 0, 0) אפשר להגיע ע"י $f(1, x, y)$. לכן ניתן ליצר קבוצה שלמה כלומר הקבוצה המקורית היא שלמה. אפשר גם להגיע ל-OR ע"י $f(f(x, 0, 0), y, 1) = (x')' + y \cdot 1 = x + y$.

למעשה לא צריך את שני הקבועים כי ברגע שיש NOT עם אחד הקבועים אפשר להגיע לקבוע אחר עם NOT על הקבוע שכן ייש לנו.

שבוע 3 | תזמון מעגלים

הרצאה

דוגמה נתונה הפ' עם טבלת האמת הבאה

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

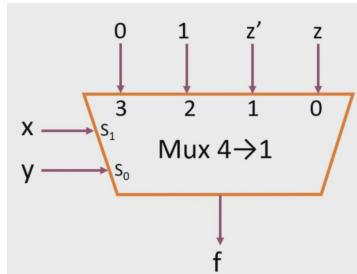
- נממש את הפ' עם 8 MUX → 1 יחיד.

כל מה שצרכי לעשות זה לחבר את z ל- s_2, s_1, s_0 , להציב בקלטים x_0, \dots, x_7 של ה-MUX את ערכי f בטבלת האמת לפיה הסדר. כך נבחר עבור כל צירוף (x, y, z) בדיקות התוצאה מתוך ערכי טבלת האמת של f .

- עתה נממש עם 4 MUX → 1 יחיד ועוד שער אחד בלבד. כאן צריך יותר להתאמץ. ראשית נביט בטבלה כאשר (x, y) מופרדים מ- z , כך שלכל (x, y) יש שני צירופים עם z עם ערכים אפשריים.

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

בנייה MUX יחיד עם סלקטורים y, x ובקלט ה- z - נשים או קבוע אם שני הצירופים עם z נותנים את אותו ערך, או z או z' בהתאם לערך השער משטנה (שכנוו עצמכם שאכן אללו כל האפשרויות). כך קיבל את השער הבא



אנחנו מקיימים את הדרישות כי יש MUX אחד ושער אחד שהוא NOT על הקלט השני ל-MUX.

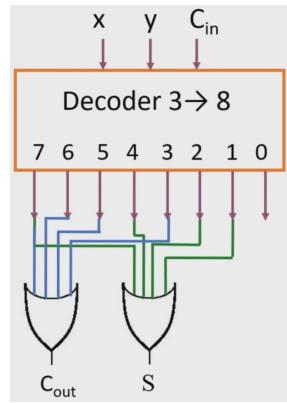
דוגמה נממש FA (שמקבל S, C_{out}, x, y, C_{in} ופולט C_{out} כזכור) שיש לו את טבלת האמת הבאה (לצורך נוחות) עם :

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

• מפענה $8 \rightarrow 3$ ושערי OR •

נבחר קלטים למפענה x, y, C_{in} , ואז על הפלטים נצמיד שער OR אחד לפט S ואחד לפט C_{out} . מה"כ זה יראה כך (יראה ברכ

(1, 1, 1) מוציאה חוט לשני ה-OR-ים, בהתאם לטבלת האמת)

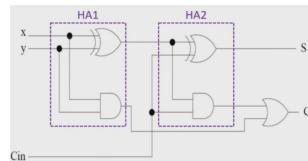


• $8 \rightarrow 1$ -MUX •

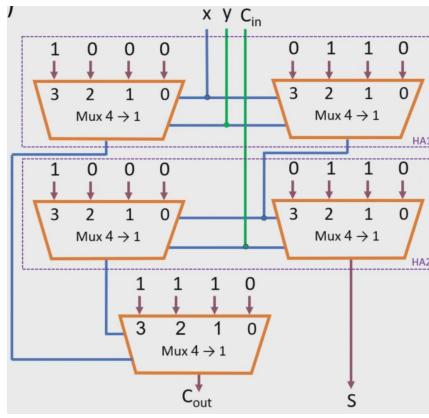
נבחר את הסלקטורים להיות x, y, C_{in} ונשבץ בשמות הקלטים של ה-MUX. מUX את ערכי S בטבלת האמת לכל צירוף של הסלקטורים (הקלטים), והואתו הדבר שוב עם C_{out} .

• $4 \rightarrow 1$ -MUX •

זה כבר יותר מורכב, והורש פירוק של FA לשני Half Adder-ים שלא הזכרנו כאן, אבל הם שעריים שמקבלים x, y ופלטים S, C_{out} . מימוש די פשוט ניתן לראות בתוך המלבן XOR הטעום הוא XOR הקלטים והנשא הוא AND על הקלטים.



כך נוכל לחבר יחד שניים כאלה כדי לחשב $C_{in} = (x + y) + C_{in}$. HA הוא מעגל עם שני קלטים ולכן קל לממש אותו עם $1 \rightarrow 4$ MUX (משבצים את ערכי טבלת האמת כאמור) וכל שנותר הוא להרכיב שני HA לאחד יותר גדול (ראו איור)

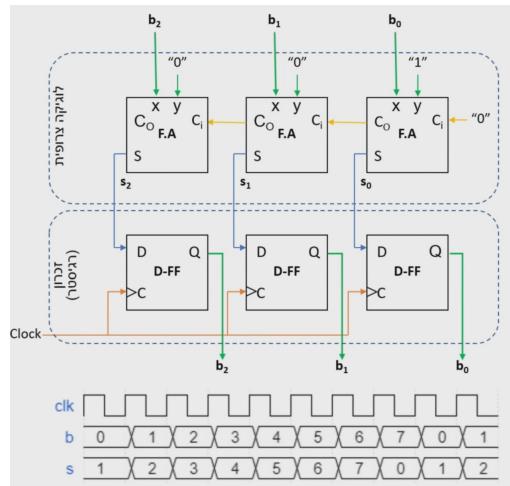


לאחר ראיינו DFF, שמקבל קלט D ושעון ומשנה את פלטו בעת עליית השעון לערך של D (זו וריאצית ה-Edge) נוכל על בסיס יחידה זו לבנות יחידות יותר מורכבות.

דוגמה Toggle FF מקבל T ושעון ומחשב בכל עלייה שעון $Q(t+1) = \text{XOR}(T, Q(t))$ יחד עם T אל תוך XOR $Q(t+1)' = Q(t)$ אם $T = 0$ ו- $T' = 1$ אחרת.

דוגמה JK-FF מקבל J, K ושעון ומחשב בכל עלייה שעון $Q(t+1) = JQ'(t) + K'Q(t)$ והפלט Q בשער לוגי שתוצאתו מזונת- D של ה-DFF הפנימי.

דוגמה נממש ספונ, שסופר את מספר עליות השעון.



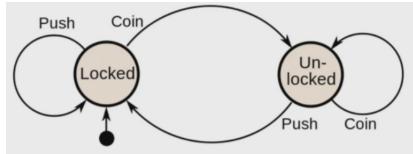
הרעיוון כאן הוא פשוט אבל המימוש לא כל כך: בכל עלייה שעון, ה-FA הימני ביותר מכניס עוד ערך 1 לסכום שמוחזק ע"י כל המעלג. הסכום מופיע בכל עלייה שעון לדFF הבא (זה-FA המתאים לו), וכך ה-DFFים מחזקים שלושה ביטים שמייצגים מספר שערכו עליה באחד בכל עלייה שעון (הסתודנטית המשקיעה תרים את שלושת המוחזרים בראש/על נייר ותראה שהוא אכן עובד).

Finite State Machine

הגדרה FSM הוא מודל חישובי עם מספר סופי של מצבים, מצב התחלתי, מספר סופי של קלטים ופלטים, פ' מעברים $\rightarrow \{\text{קלטים}\} \times \{\text{מצבים}\}$ ופ' פלט.

דוגמה שער מסתובב (Turnstile) יכול להיות פתוח או סגור, אם הוא מקבל מטבע והוא נועל, הוא נפתח, אם הוא לא נועל ונדחף, הוא נועל. כן המצבים הם "פתוח" ו"נועל", הקלטים הם מטבע ודחיפה, הפלט עבורו "פתוח" הוא "אפשר לעبور" ובהתקדמות עבורו "נועל".

ונוכל לצייר את ה-**FSM** עם גראף

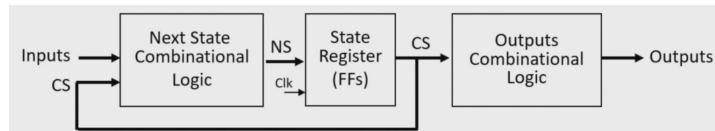


או טבלה (יש כמה דרכים)

State	Output	Input	Next State
Locked (init)	Closed-pass	Coin	Unlocked
	Push		Locked
Unlocked	Open-pass	Coin	Unlocked
	Push		Locked

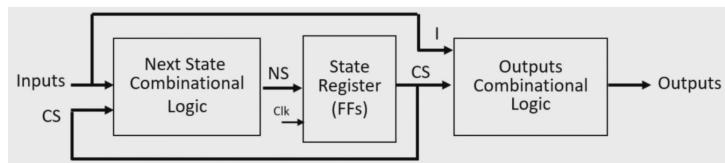
State	Output	Inputs	
		Coin	Push
Locked (Init)	Closed pass	Unlocked	Locked
Unlocked	Open pass	Unlocked	Locked

דוגמה מכונה כבאיור (CS-NS Moore) היא מכונה שבה המצב הבא והווכחי בהתאם, שמה שמייחד אותה הוא שהפלטים תלויים אך ורק במצב הנוכחי.



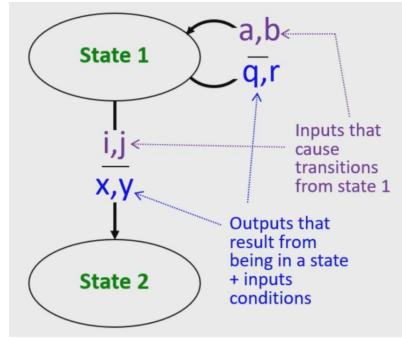
נשים לב שהרגיסטר שמורכב מ-FF-ים מחזיק את המצב הנוכחי, וערכו מהווות חזרה פנימית לחישוב המצב הבא שיכנס לרגיסטר במחזור הבא.

דוגמה מכונה Mealy היא מכונה שבה הפלטים תלויים גם במצב וגם בקלטים, והארQUITטורה שלו היא כבאיור



הערה את הייצוג של הגרפי של מכונה Moore מצירירים כמו אוטומט רגיל, רק שם כל מצב (מעגל) מכיל גם את הפלטים שהוא משרת. את הייצוג הגרפי של מכונה Mealy מצירירים כמו אוטומט רגיל, רק שעל החצים (המעברים) נוסיף את הפלטים שהקלטים על החץ יחד עם המצב שעוברים אליו משרים (ראו איור).

הערה הפלטים יושפעו מהקלט מהר יותר ב-**Mealy** כי הם מחוברים ישירות לפ' הפלטים, בעוד ב-**Moore** נדרש עלייה שעון כדי שישתנה המצב המשרת פלאט.



הערה אפשר לכתוב מכונות Mealy ששוולות למכונות Moore עם פחרות מלבנים, אבל החסרונו הוא ש-Mealy גורם לביעיות עם תזמנונים (שנלמד בהמשך).

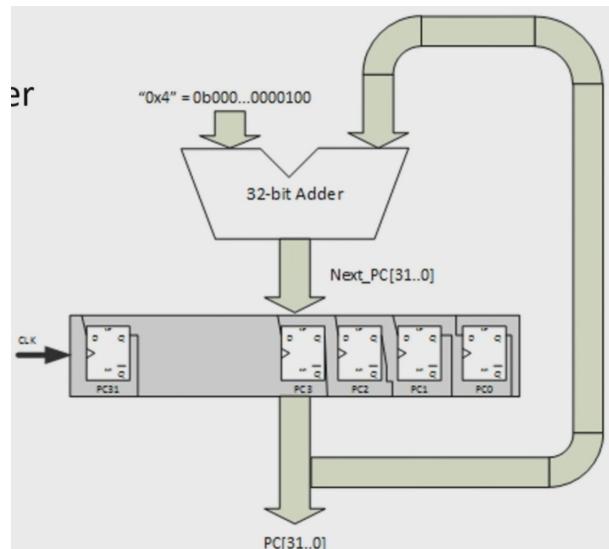
דוגמה נזכיר לדוגמת השער המסתובב. נגידר 0 השער נעל-ו-1 הוא פתוח. לכן המעגל שלפ' הפלט הוא חוט ישיר מהמצב לפלט כי הפלט זהה למצב. את המעברים ניתן לראות בטבלת האמת הבאה

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

והמיוש של המעגל מעבר למצב הבא הוא די פשוט: אם Q הוא המצב הנוכחי ו- D המצב הבא (הקלט ל-DFF) אז $D = \text{Coin} \cdot Q + \text{Push}' \cdot Q'$ שזה ניתן למימוש עם שערים מאוד פשוטים.

דוגמה Program Counter נתונים למעבד בכל פעם את הכתובת בזיכרון ממנה צריך לקרוא את הפקודה הבאה. הספרן פולט כתובות באורך 32 ביטים שkopatzת בקייזות של 4 (אלא אם הייתה קפיצה למקום אחר) בגלל שככל מילה היא באורך 32 ביטים (בית הוא 8 ביט).

המיוש הוא די פשוט: נחזק 32-DFF-ים שייחזקו את הכתובת, נחוות יישירות את ה-DFF ל-32 הפלטים והמעגל לחישוב המצב הבא הוא FA עם FA $y = Q \cdot x = Q \cdot 4$ (Q המצב הנוכחי), או באյור ברוזולוציה נמוכה משום מה זה יראה כך



זהה מכונת Moore, שערכה מתעדכן פעם אחת בכל מחזור.

תזמון מעבד

הגדרה התדר של מעבד הוא מספר המוחזורים של השעון בשנייה (ביחידות Hz).

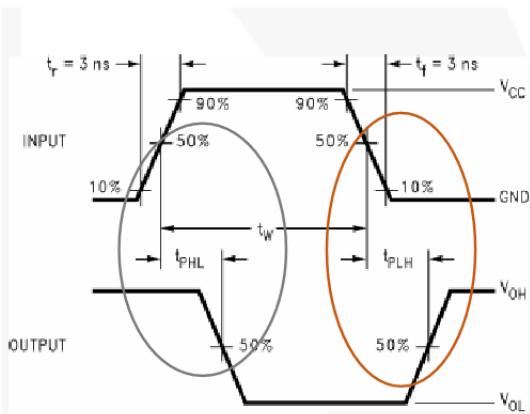
טעינה ופריקה של מטען לוקחים זמן ולכון מעבר של זרם בתוך טרנזיסטור אינם מיידיים (אלא מאוד מהירים). פרק הזמן זהה נקרא Propagation Delay.

פרמטרים של זמן פעוף

- זמן הפעוף מנמוך לגובה (t_{PLH}) : זמן הפעוף כשהפלט עובר מנמוך (0) לגובה (1). מודדים אותו החל משינוי ניכר בקלט ועד לעליית הפלט ל-50% מתח. בפועל מתח נחسب גובה (ומתפרש כ-1) רק כשהוא 90% ומעלה מערכו המקסימלי, וכך יש הנחה סבואה שהעליה והירידה של המתח הם מאד מהירים ולכון מ-50% ל-90% אין הבדל משמעותי.
- זמן הפעוף מגובה לנמוך (t_{PHL}) : זמן הפעוף כשהפלט עובר מגובה לנמוך, מחושב ע"י הפרש הזמנים בין שינוי ניכר בקלט ועד לירידת הפלט למתחת ל-50%.
- זמן עלייה (t_r , Rise) : הזמן שלוקח לעלות מ-10% מתח ל-90% מתח.
- זמן ירידת (t_f , Fall) : הזמן שלוקח לרדת מ-90% ל-10% מתח.
- זמן פעוף (t_{pd}) : כש- $t_{pd} = t_{PHL}$, נקרא להם.

הערה t_r, t_f הם מאוד קטנים (ננו-שניות).

דוגמה בשער NOT כלשהו מקבלים את הגירף הבא של הפלט כתלות בקלט.

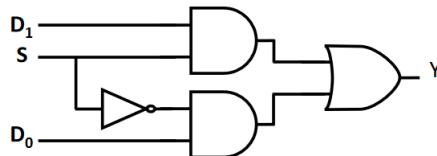


נשים לב שהשינוי הוא לא מיידי. במקרה הזה השינוי הניכר בקלט הוא חצייתו (מלמעלה או מלמטה) של הקלט את רף 50% המתח.

הגדרה עבור t_{pd} יש ערך טיפוסי, ערך מקסימלי (t_{pd_max}) שאומר אחרי כמה זמן בטוח הפלטים כבר ישתנו וערך מינימלי (t_{pd_min}) שمبטיח מתחת לאיזה רף בטוח הערכים לא ישתנו.

הערה $t_{pd_min/max}$ יכולים להיות שונים עבור שינוי בקלטים שונים (קלט x משפיע יותר מהר מאשר y).

דוגמא נתון השער X שמשמש באופן הבא



וחסמי זמן פעוף לשערים

t_{pd_max}	t_{pd_min}	שער
5ns	2ns	NOT
8ns	4ns	AND
10ns	5ns	OR

- מהו t_{pd_max} של השער כולם?

עבור ההשפעה $Y \rightarrow D_1 \rightarrow Y$ (כמה זמן לאחר שינוי D_1 ישתנה) נctrיך לעבור דרך AND ו-OR כלומר סה"כ 18ns, וכך גם עבור

. D_0

עבור $Y \rightarrow S$ (זמן הפעוף המקסימלי הוא $\max(AND + OR, NOT + AND + OR) = \max(18, 23) = 23$ ns)

זמן הפעוף של השער כולם הוא המינימום של כל המסלולים, כלומר 23ns.

- מהו t_{pd_min} של השער כולם?

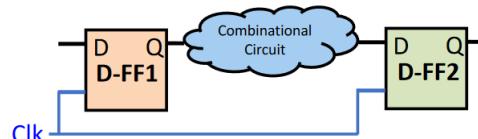
עתה נבחר את המסלול הקצר ביותר, שהוא כמובן $D_0 \rightarrow Y$ ($D_1 \rightarrow Y$ שדורש 4ns ועוד).

הגדרה זמן הפעוף של $D \rightarrow Q$ מחושב (מודגדר) החל מעלייה של השעון ל-50% ועד לשינוי Q והוא נקרא t_v (או t_{PCQ}) והוא למעשה מגדי אחרי כמה זמן לאחר עליית השעון נוכל לה חשב את הקלט כחוקי. t_{v_min} מבטיח עד מתי Q ישאר בערכו הקודם ו- t_{v_max} מבטיח החל ממתי יהיה הערך החדש.

הגדרה כדי Q יהיה תקין, D צריך להיות יציב ולא להשתנות במשך פרק זמן לפני עליית השעון, זהו (*Setup*), וגם לאחר עליית השעון, זהו (*Hold*) t_h .

הערה $t_s > 0$ כי בתוך ה- DFF- Master-slave ישנה את ערכו קצר לפני ה-Slave וכאן הערך שם צריך לא להשתנות כדי של-Slave יהיה את הערך הנכון.

דוגמא נביט בكونסטרוקציה הבאה



נניח שזמן מחזור השעון הוא t_{cyc} (זמן בין עליית שעון אחת לשניה), לכן קצב השעון הוא $f = \frac{1}{t_{cyc}}$. נניח כי זמן הפעוף המקסימלי של השער הוא t_{pd-max} . מהו זמן המחזור המינימלי כדי שהמעגל יהיה תקין, כלומר כדי שההתוצאה תגיע ממעגל-L-1 עד לפולט של 2 תוך שני מחזורים (בראשו הקלטים עוברים את השער ובשניים הם כבר מופיעים בצד השני)?

- זמן המחזור צריך לפחות t_{cyc} .

$$t_{cyc} \geq t_{v-max} + t_{pd-max} + t_{setup}$$

כדי>Create קודם לחייב שփולט של DFF1 יהיה חוקי (t_{pd-max}), אז למתן לעבור את כל השער (t_{v-max}) ואז שהפלט יהיה ייציב מספיק זמן לפני עליית השעון הבאה כדי שייעבור בהצלחה ל-Q של DFF2 לאחר העליה.

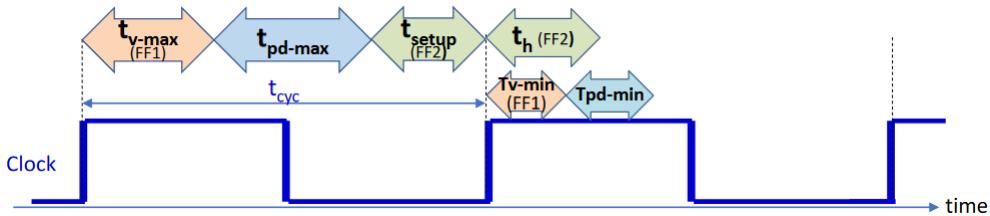
- מה צריך להיות t_h כדי שהשער יהיה תקין?

בפעם השנייה שבה ישנה הערך, נדרש ש- t_h של DFF2 יהיה פחות מזמן שלוקח ל-D של DFF2 להשתנות בהשפעת ה-Q. החדש של DFF1. כלומר, חייב להתקיים

$$t_h \leq t_{v-min} + t_{pd-min}$$

כדי>Create קודם לעבור מ- t_{pd-min} (DFF1) ו- t_{v-min} (DFF2) ייזכר זמן המינימלי שעבורו (DFF1) Q לא ישנה לאחר עליית השעון (t_{pd-min}) ועוד זמן המינימלי שעבורו (DFF2) לא יקבל ערך חדש כפלט של המעגל (t_{v-min}).

סה"כ מחלק התזומנים כפ' של השעון הוא באירוע



הערה אם אין לנו דרך לשלוט ב- t_h ונרצה עדין מעגל חוקי, אפשר לחיבר את t_{pd-min} להיות יותר גדול ע"י הוספה שני NOT-ים למסלול הקצר ביותר במעגל (הוא לרוב לא הארוך ביותר) וכך לא להשפיע על התוצאות אבל כן על התזמון המינימלי.

דוגמא נתונה הקונסטרוקציה הבאה עם MUX, שלו (הכלביות ns , $t_{pd-min} = 9$, $t_{pd-max} = 23$, $t_{v-min} = 2$, $t_{v-max} = 7$, $t_s = 3$, $t_h = 5$)

- מהו זמן המחזור המינימלי האפשרי?

כדי שהמעגל יהיה תקין, נדרש שמסלול הפעוף הארוך ביותר במבנה יהיה כולל מוכל במחזור אחד. המסלול הארוך ביותר הוא משינויי ב- $(S0)/Q$, דרך חישוב ה-X-MUX ועד לשינוי הערך ב- D ,ऋצרייך לחתות בחשבון שלפני תחילת המחזור הבא נוצר ש- D יהיה יציב ל- t_{setup} -שניות. סה"כ נדרש שיתקיים (בדומה לדוגמה הקודמת)

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

- מהי הדרישה על t_h כדי לקבל מבנה תקין?

נוצר שערך של D לא משתנה מוקדם מדי לאחר עליית השעון, בפרט שזמן שינוי הערך Q וחישוב ה-MUX יקחו יותר מאשר

$$(t_h = 5 \leq 9 \text{ ו } t_h \leq t_{v(min)} + t_{pd(min)} = 2 + 9) \text{ כלומר } t_h = 9$$

$$F_{max} = \frac{1}{33ns} = 30MHz \text{ ולכן } T_{cyc(min)} = 33ns$$

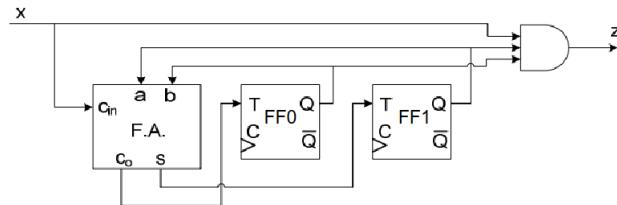
עד כה הטענו מהקלט האמיתי של המודול שמחובר ל-S1. הוא עצמו גם חייב לקיים דרישת זמן לשעון וכן נוכל להתייחס אליו כסיגנל שמסוגל מפלט אחר והניתוח יהיה כן.

כל קלט למעגל אחד הוא פלט של מעגל אחר, וכך יכולם להיות להם ערכי t_v, t_{v-min} שונים ממודול אחר.

כל פלט למעגל כלשהו הוא קלט למעגל אחר וכן הפלטים צריכים לעמוד בדרישות t_h, t_s מסוימות גם כן.

הערה כשבנתח זמן של מודול, נסתכל על כל המסלולים שמתחלים בכניסה לדFF ונגם

דוגמה נסתכל על המבנה הבא, כשהנתון $t_{setup} = 9, t_{hold} = 0$ (ז' נדרש להיות יציב לפחות $9ns$ לפני עליית השעון, ו- $0ns$ אחריו)



עם הנתונים

Parameter	t_{pd-max}	t_{pd-min}	t_{setup}	t_{hold}	t_v	t_{v-min}
AND 3 inputs	3	1				
FA [a,b,Cin]->S	12	3				
FA [a,b,Cin]->Co	8	3				
T-FF			7	2	4	0

• מהו $T_{cyc-min}$? נתח את כל המסלולים שנגמרים בפלט (כפי לפלט יש דרישות ביחס לשעון, בפרט לדFF יש

יש כזה שנדרש מאיתו)

– מסלולים שנגמרים ב-z : המסלול המקסימלי הוא באורך

$$t_{setup} + t_{pd}^{AND} + \max \{t_v^{FF1}, t_v^{FF0}, t_v^x\} = 9 + 3 + \max \{4, 4, 15\} = 27ns$$

כדי שהזמן יהיה יציב זמן לפני עליית השעון, ערכו מחושב ע"י AND או מוסיפים את זמן הפעוע דרכו, וקלט

ה-AND הם פלטי FF1, FF0 ו-x בתאמה, שערכם מתעדכן למוחזר הנוכחי לאחר ה- t_v של כל אחד מהם (כאן אנחנו

$$(t_v = t_{v(max)}) \text{ מיסמנים}$$

– מסלולים שנגמרים ב- T : המסלול המקסימלי הוא באורך

$$t_{\text{setup}}^{\text{FF1}} + t_{pd(\rightarrow S)}^{\text{FA}} + \max \{t_v^{\text{FF1}}, t_v^{\text{FF0}}, x_{\text{valid}}\} = 7 + 12 + \max \{4, 4, 15\} = 34ns$$

– מסלולים שנגמורים ב- T : המסלול המינימלי הוא באורך

$$t_{\text{setup}}^{\text{FF0}} + t_{pd(\rightarrow C_0)}^{\text{FA}} + \max \{t_v^{\text{FF1}}, t_v^{\text{FF0}}, x_{\text{valid}}\} = 7 + 8 + \max \{4, 4, 15\} = 30ns$$

$$\text{ולכן סה"כ נצרך } T_{\text{cyc-min}} \geq \max \{27, 34, 30\} = 34ns$$

- האם מתקיימת הדרישה על ה- Min Delay .

$$\text{כן! עבור } z \text{ מתקיים } t_h^z = 0ns \text{ והזמן המינימלי לפעוף הוא}$$

$$t_{pd(\min)}^{\text{AND}} + \max \{t_{v(\min)}^x, t_{v(\min)}^{\text{FF}}\} = 1 + \min \{0, 0\} = 1ns$$

כלומר מתקיימת הדרישה ועבור $t_h^{\text{FF}} = 2ns$ FF0, FF1 יש $t_h^{\text{FF}} = 2ns$

$$t_{fpd(\min)}^{\text{FA}} + \min \{t_{v(\min)}^{\text{FF}}, t_{v(\min)}^x\} = 3ns$$

(כי הערך החדש צריך לעבור דרך FA ולהגיע או משינוי ב- x או משינוי ב- Q של אחד ה-FF-ים) ולכן מתקיימת הדרישה גם כן.

תרגול

הגדירה מעגל צירופי (קומביינטורי) הוא מעגל שיש לו כניסה ויציאות כאשר האחרונות תלויות בכל ערך רגעי של הראשונות. מעגל סדרתי הוא מעגל שפלטיו תלויים גם ביחידת זכרון שמחוברת לשעון.

דוגמה כיצד נממש **Full Adder** (כזכור קלטים S, C_{out} ופלטים x, y, C_{in} (מיימן))?

		C _{in}			
		00	01	11	10
x	0	0	1	0	1
	1	1	0	1	0
		y			

		C _{in}			
		00	01	11	10
x	0			1	
	1		1	1	1
		y			

ולכן אפשר למש את S עם OR על ארבעה פלטי AND (שכוללים קלטים שעוברים דרך מהפץ) ואת C_{out} אפשר קצת יותר עיל. ראיינו בהרצאה שהמיימוש של FA כולל בתוכו שני HA.

הגדרה מוחסרים מחשבים חישור ספורות בינאריות. עתנ נפלוט את ההפרש (בערך מוחלט) ו- Out Borrow (בערך מוחלט) שיגיד לנו כמה יותר לחסר מעבר להפרש. הסטודנטית המשקיעה תמשח חצי-מחסר ומוחסר מלא.

הגדרה משווים הם מעגלים שבודקים איזה קלט יותר גדול.

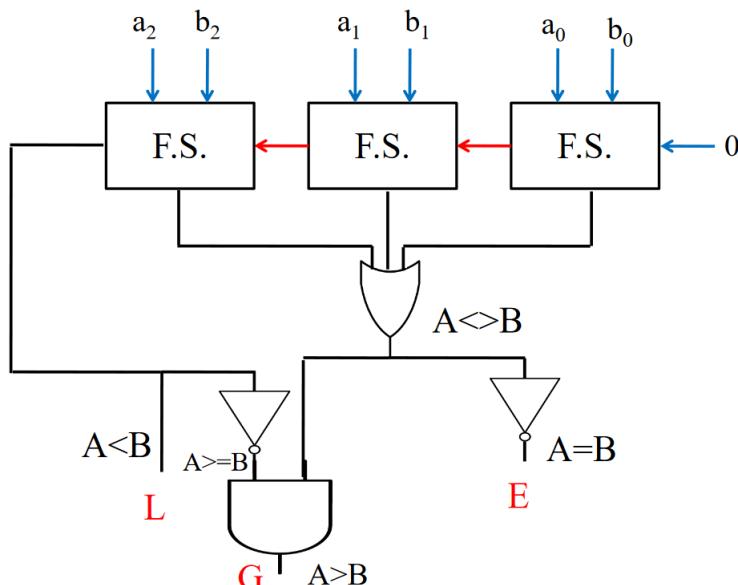
- דרך אחת למשוך זאת היא באמצעות השוואת מה-LSB ל-MSB כאשר בפעם הראשונה שיש אי-שוויון בביטים נבחר את האחד

שקלטו גדול יותר (1 לעומת 0).

- דרך אחרת היא באמצעות מוחסרים : $A > B \iff A - B > 0$ וכו'.

דוגמא נתונים הקלטים G, E, L ממשו עם מוחסרים מעגל שפלט ביטים $A = a_2a_1a_0, B = b_2b_1b_0$ שערך כל אחד מהם אם "1" אם "0" בהתאם $B, A = B, A < E$.

נמשח את המעגל כבאיור



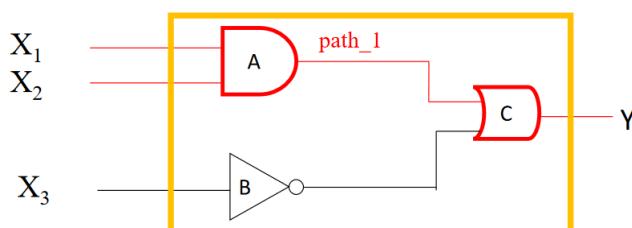
אם $a_2 < b_2$ בהכרח ישאר לנו Borrow Out או ייה BO מלפני שימוש הלהה ואחרות $a_2 = b_2$ או יהי אם $a_1a_0 < b_1b_0$ וגם $a_2 = b_2$ כי אם BO מילפני שימוש הלהה ואחרות $a_1a_0 < b_1b_0$ אז יהי BO. ולכן שיהיה BO.

אם כל הביטים זהים נצטרך NOR על כל הפרשיים (כל החיסורים פולטים 0) ואכן זה מה שנבנו.

בשיטת האלימנציה, G הוא 1 אם $A \geq B$ וגם $A \neq B$, כלומר $A \geq B$ הוא 1 ו- L הוא 0 וכן אכן מופיע במעגל.

הערה השימוש ב-10%-ו-90% כרף לחישובי זמן נובע מכך שקשה לאפיין את פריקת הקבל כתהיליך לינארי בקטנות, ולכן מתעלמים מהם.

דוגמא נביט בפ' $X_1 * X_2 + X'_3$ שמשמעותו באוף הבא



מתתקיים $t_{pd}(A) = \max\{t_{PLH}(A), t_{PHL}(A)\}$ ו- X_3 מ- X_1, X_2 והשני מ- Y .

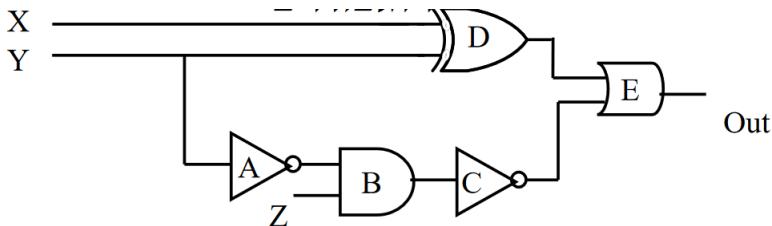
לכן $t_{pd}(B) + t_{pd}(C) = t_{pd}(A) + t_{pd}(C)$ (מסלול ראשון) ובהתאמה (מסלול שני) תחת הנ吐ונים הבאים (כאשר t_{cd} מלשון $t_{cd} = t_{tp-min}$)

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

נחשב את ה- t_{pd} של המעגל כולו. עברו כל שער, וכן $t_{pd} = \max\{t_{PHL}, t_{PLH}\}$ ו- $t_{pd} = 190ns$.

נחשב את ה- $t_{pd(min)}$ של המעגל. $t_{pd(min)} = 18ns$ ו- $t_{pd(min)} = 22ns$.

דוגמה נתוני המעגל והנת吐ונים הבאים



Data in ns	A	B	C	D	E
t_{pd}	15	25	15	60	20
t_{pd-min}	5	5	5	10	5

המסלולים של שערים מהקלטים לפולטים הם $B \rightarrow C \rightarrow E$, $A \rightarrow B \rightarrow C \rightarrow E$, $D \rightarrow E$ ו- $Z \rightarrow D$ בהתאמה $t_{pd} = 80ns$ ו- $t_{pd(min)} = 15ns$.

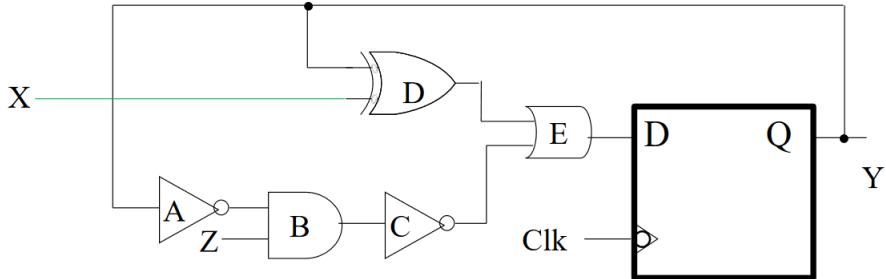
לכן $t_{pd(min)} = 15$ ו- $t_{pd} = 80$.

הערה כל עוד אין מעגלים סדרתיים, חישוב t_{pd} נעשה ע"י מקסימום הזמנים על כל המסלולים מפלטים לקלטדים ו- $t_{pd(min)}$ ע"י מינימום.

הערה עברו כל מעגל סדרתי חייב להתקיים $t_{hold} \leq t_{v(min)}^{FF} + t_{pd(min)}^{לגיון}$ כדי שהשינוי הכוי מהיר במעגל יקח יותר מאשר הזמן שהפלט צריך להישאר זהה אחרי עליית השעון.

הערה הקלטו למעגל חייב להתייצב לאחר לכל היוטר $+ t_{setup}$ זמן כדי שה-FF יוכל לחשב באופן תקין את Q .

דוגמה נתון המעגל הסדרתי הבא עם הנת吐ונים תחתיו



Data in ns	A	B	C	D	E
t_{pd}	15	25	15	60	20
$t_{pd(min)}$	5	5	5	10	5

כדי להשלים את הניתוח של הזמן המוגדר נצטרך את האילוצים על ה-DFF
 $t_{setup} = 10ns$, $t_{v(min)} = 5ns$, $t_v = 10ns$ DFF $t_{hold} = 25ns$

• האם המוגל תקין?

לא! חיבר להתקיים $25 = t_{hold} \leq 5 + t_{pd(min)} = 5 + t_{pd(min)}^{D \rightarrow E} = 5 + 10 + 5 = 20$ סתייה.

• כיצד ניתן את הבעיה?

ונסיף דילוי על המסלול הקצר ביותר, בפרט נוסיף שני NOT-ים על החוט בין Q לקלט העליון של D (ושל A). עכשו המסלול הקצר ביותר יש לו $25 = t_{pd(min)} = 5 + 25 = 30$ ועכשו קיבל $25 = t_{hold} \leq 5 + 25 = 30$ כלומר זה כן תקין. החסרונו הוא שהתדר המקסימלי האפשרי ירד כי זמן המוחזר המינימלי עלה כתוצאה מהוספה שני NOT-ים.

MIPS | VII שבוע

הרצאה

הגדרה Instruction Set Architecture היא אוסף כללים שמתכונת צריך לענות להם כשהוא מפתח למעבד.

דוגמה אנחנו נלמד על MIPS, אבל במציאות פופולריים מאוד x86 ו-ARM, וגם V-RISC-Sh הוא פרויקט קוד פתוח.

הגדרה מיקרו-ארQUITקטורה היא מימוש של ISA.

דוגמה המימוש של אינטל ו-AMD ל-x86 הוא מיקרו-ארQUITקטורה.

לצורך האבstrקטציה שלנו, מעבד הוא מכונת מצבים המוחברת ל זיכרון, כאשר המცב כולל רגיסטרים שערכם משתנה על ידי פקודות. הזיכרון הוא מערך שניגשים אליו לפי אינדקס (בית אחד בכל פעם). כל מעבד מרים את התכנית הבאה:

1. קרא את הפקודה הבאה מהזיכרון בכתב שברגיסטר PC (Program Counter).

2. הוסף לרגיסטר PC את מספר הבטים שתופסת פקודה (התקדמות לפוקודה הבאה).
3. בצע את הפוקודה (וישנה את מצב המעבד).
4. חוזר לשלב 1.

בහינתן תוכנה בשפה עילית (שפתקמפלט), נתרגם את הקוד לסדרת פקודות אסמבלי באמצעות קומpileר. לאחר מכן נשתמש באסמבילר כדי לתרגם את קוד האסמביל לבינארי, שאותו המעבר כבר יודע להרץ.

הערה לעיתים הקוד שלנו ישמש בספריות חיצונית או בקבצי קוד אחרים, ועל אייחוי כל הפקודות לקובץ אובייקט יחיד. הפקודות באסמביל הן פקודות שהמעבד יודע לבצע (ח-ISA של המעבד), אבל לפני שהן מקודדות ל-1-ים ו-0-ים עבר המעבד.

CISC vs RISC

- Complex Instruction Set Computer זו גישה לפיה פקודות האסמביל יהיו קרובות ככל הניתן לשפה עילית, וכך להוריד את מספר הפקודות בתוכנה. בפועל זה מומッシュ ע"י פקודות שפותחות למיקר-פעולות ע"י החומרה. ל-ISA הזה יש הרבה מאוד פקודות, בפורמטים שונים והרכבה של פקודות שונות אחת על השניה, ואפשר אפילו להריץ פקודות ישירות על הזיכרון שמסתירות את המעבד בריגיסטרים. אורך הפקודות יכול להשנות, כאשר נקודד פקודות שכיחות באמצעות בית אחד, עד לפקודות הנדירות ביותר שהן באורך 15 בתים.

8x DSP הולכים לפי גישת CISC.

- Reduced Instruction Set Computer היא גישה לפיה יש לשמור את מספר הפקודות מצומצם וכך לפחות את פעולות החומרה, ולתת לקומpileר לעשות את העבודה הקשה של בחירת הפעולות ואופטימיזציה. הפקודות הם די פשוטות והוא רק בין רגיסטרים (ולא על הזיכרון), ורק באופן מפורש לטעון ולשמור נתונים בזיכרון. אורך הפקודות הוא קבוע.

RISC-V MIPS, ARM RISC הולכים לפי גישת

הערה המגמה עם הזמן היא לעבור מ-CISC ל-RISC משום שבמעבר קשה היה כתוב קומpileרים עילאים ולכן נדרשה המורכבות של פקודות האסמביל, ואילו עם חלוף הזמן נהיה קל ויעיל יותר לכתוב קומpileרים (בשפה עילית) שיבצעו את הפעולה המורכבת עצמאם.

דוגמה עבור העתקה של 100 ערכים בין מערך אחד לאחד ב-c, יש ב-8x פקודה אחת שמקבלת את מספר הבטים להעתקה והפוניטרים והחומרה כבר תממש את ההעתקה, לעומת זאת RISC שם צריך למשולאה שמעיטה לרוגיסטר ואז לזכור מילה מילה.

MIPS

במעבד MIPS יש 32 רגיסטרים, \$31, ..., \$0, שכל אחד מהם בגודל 32 ביט, המכונה "מילה" (4 בתים). מספר הרגיסטרים קבוע כך לאחר ניתוח של מספר המשתנים בתוכנות מדגמיות, כאשר אם יש יותר משתנים מרוגיסטרים נשתמש בזיכרון הראשי כדי להחזיק את ערכם. לחלק מהרגיסטרים יש יעודיים ספציפיים, כפי שניתן לראות בטבלה הבאה

Name	Register Number	Usage	Preserve on call?
\$zero	0	constant 0 (hardware)	n.a.
\$at	1	reserved for assembler	n.a.
\$v0 - \$v1	2-3	returned values	no
\$a0 - \$a3	4-7	arguments	yes
\$t0 - \$t7	8-15	temporaries	no
\$s0 - \$s7	16-23	saved values	yes
\$t8 - \$t9	24-25	temporaries	no
\$gp	28	global pointer	yes
\$sp	29	stack pointer	yes
\$fp	30	frame pointer	yes
\$ra	31	return addr (hardware)	yes

ניסיונות רבים מהרגיסטרים משמשים פעילות תקינה של המחשבנית (SP, FP, RA), חלק שומרם ארגומנטים וחלק משומשים לצרכים אחרים.

כל פקודה היא בגודל מילה (32 ביט), וכל פקודה מבצעת פעולה פשוטה, בין היתר פעולות אРИתמטיות ולוגיות, גישה ל זיכרון (load, store) ופקודות ופוקודות מותניות. הפוקודות שמורות בזיכרון ובכל פעם נקרא מהזיכרון את הפוקודה ונರץ אותה.

פקודות אРИתמטיות

- חיבור/חיסור : נביט בפקודה `add $d, $s, $t` (כאשר שלושת הפרמטרים הם רגיסטרים), ובדומה `sub $d, $s, $t`

ניסיונות לא הודיעו לprocessor שמדובר במספרים המשמשים בשיטת המשלים ל-2, כי אנחנו מניחים שמי שקיים פוקודה יודע מהקשר שהוא סוכם רגיסטרים שיש בהם כבר ערכים מיוצגים במסללים ל-2, ואם לא אז זה באג.

$$f = (g + h) - (i + j)$$

הקומפיילר יקצח רגיסטרים למשתנים ונקבע $\$s0 = (\$s1 + \$s2) - (\$s3 + \$s4)$, ואז לתרגם השורה לשפת אסמבלי יש כמה אפשרויות, הנאייה ביותר מתוכן היא

`add $t0, $s1, $s2`

`add $t1, $s3, $s4`

`sub $s0, $t0, $t1`

אבל יש דרכים אחרות (לדוגמה לפתח סוגרים). במתמטיקה אמנים התוצאות שקולות, אבל כאן יכול להיות שנתקבל תוצאה אחרת בגלגול-overflow.

- חיבור מיידי: הפוקודה `i imm $t = $s + addi $s, $t, imm` מחשבת i (כלומר חיבור בין רגיסטר וקבוע והשמה ברגיסטר).
- כאן וטמו נתנו לנו במסללים ל-2 אבל בגודל 16 ביט כדי שנוכל לכלול אותו בתוך קידוד הפוקודה בגודל מילה אחת, ולכן נדרש להרחיב אותו במסללים ל-2 בגודל 32 ביטים, פועלה זו נקראת [Sign Extend](#), וניתן למשה בקבוצות ע"י ריפוד בצד של ה-MSB עם ערך קבוע של 0 לחוביים ו-1 לשיליינים (למעשה ערך ה-MSB לפני הריפוד).

הערה לא צריך `sub` כי אפשר למשה בקבוצות עם `addi` כאשר ה-`imm` הוא מספר שלילי.

פעולות לוגיות

- הפקודה היא $t = s \text{ and } d$ והוא מחשבת שער AND על כל שני ביטים מתאימים $m-s$ ו- t (בו זמנית על כל הביטים) ושומרת את התוצאה ב- d .
- הזהה לוגית: הפקודה $a = sll d, t$ (מלשון sll Shift Left Logical ביטים שמאל (לכיוון ה-MSB)) את הביטים של t מימין מכנים אפסים ובנותיים מאבדים ביטים משמאלי, כאשר במקרה שמספרים מיוצגים במשלים ל-2 זה יכול לאבד את בית הסימן, ובכל מקרה נוכל לקבל overflow גם במקרה של טבעים.
- הזהה אריתמטית: נשמר על בית הסימן גם לאחר ההזהה במקום להתעלם ממנו. ראו טבלה שמשווה את כל ההוצאות הקיימות ב- MIPS.

Instruction	Operation	Description
<code>sll \$d, \$t, a</code>	<code>Shift Left Logical \$d = \$t << a</code>	Shifts a register value left by the shift amount listed in the instruction and places the result in a third register. Zeroes are shifted in.
<code>sllv \$d, \$t, \$s</code>	<code>Shift Left Logical \$d = \$t << \$s</code>	Shifts a register value left by the value in a second register and places the result in a third register. Zeroes are shifted in.
<code>sra \$d, \$t, a</code>	<code>Shift Right Arithmetic \$d = \$t >> a</code>	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. The sign bit is shifted in.
<code>sraw \$d, \$t, \$s</code>	<code>Shift Right Arithmetic \$d = \$t >> \$s</code>	Shifts a register value right by the value in a second register and places the value in the destination register. The sign bit is shifted in.
<code>srl \$d, \$t, a</code>	<code>Shift Right Logical \$d = \$t >>> a</code>	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. Zeroes are shifted in.
<code>srlv \$d, \$t, \$s</code>	<code>Shift Right Logical \$d = \$t >>> \$s</code>	Shifts a register value right by the amount specified in \$s and places the value in the destination register. Zeroes are shifted in.

הזהה אריתמטית של בית אחד שמאל פירושה חילוקה ב-2 (עד כדי overflow) וימינה פירושה חילוקה ב-2 (עם עיגול כלפי מטה).

פעולות זכרון

הזכרון הוא מערך, שנגישים אליו באמצעות כתובות, כאשר כל כתובות מצביעה לבית אחד של מידע, למרות שבפועל ב-MIPS נקרא וכנתוב בichiroot של מילה (ארבעה בתים) מיושרת (כלומר כתובות שמתחלקות ב-4).

- קריאה מילה: $t = s$ קוראת מילה מהזיכרון בכתובת $s + imm$ (immediate) והומרת אותה ב- t .
- דוגמה (0(\$a0), 1(\$t1)): קוראת מילה מהכתובת שמורה ב- $a0$ וככתובת אותו לרגיסטר t .
- כתיבה מילה: $s = t$, $i(s) = t$, $w(s) = t$, $l(s) = t$, $sw(s) = t$, $sl(s) = t$ לכתובת $s + imm$ בזיכרון.

דוגמה יש לנו מערך A עם שלושה ערכים (מספרים, בגודל ארבעה בתים), שכתובת הבסיס שלו שמורה ב- s . נוכל לגשת אל האיברים במערך באמצעות $0($s3)$, $4($s3)$, $8($s3)$.

דוגמה נניח שיש לנו את הפקודות ב-C

```
int A[100];
A[12] = h + A[8]
```

```
lw $t0, 32($s3) # load A[8] into a temporary register
add $t0, $s2, $t0 # add h to the temporary register
sw $t0, 48($s3) # save the result in A[12]
```

פילוסופיות ארגון זכרון

- ארכיטקטורת אוניברסליות : זכרון אחד לפקודות התוכנה וגם לשמשני התוכנה. כש庫ראים מהזיכרון, משמעות התוכן (פקודה או מידע) תלוייה בפעולת שהובילה לקריאה. אפ"פ שתיתכן הפרדה פיזית בין החלקים, לוגית הם ממופים לאותו המקום.

יתרונות אזור זכרון מאוחד, וכל לדג תוכנות ולשנות את אופן הפעינה.

חסרונות קריאת פקודות וקריאת מידע קוראות על אותו המשאב.

ממומש ב-**x86, MIPS, ARM**.

- ארכיטקטורת הארוורד : הזכרון של הקוד מופרד מהזיכרון של המידע, כך ש-wa קורא רק מידע וfetch לפקודות קורא רק פקודות.

יתרונות אין גישה במקביל למשאים שונים על אותו הפס - ביצועים יותר טובים.

חסרונות חוסר גמישות בגודל הסגמנטים של קוד לעומת דאטא וקשה יותר לעורוך את הקוד לדיבוג.

ממומש בעיקר ב-**DSP**.

הערה גם בוואנו-ניומן המימוש בפועל יכול להיות באמצעות רכיבים פיזיים שונים, הגישה תהיה באמצעות אוטומרחב כתובות (אמנם כתובות אחרות, אבל באותו מיפוי).

פקודות קפיצה והתנויות

- קפיצה בלתי-מוותנת : **label j** קופץ לאחר סיום הפקודה לשורה הקוד שמופיע לאחר ה-label **label (copy שנכתב בקובץ asm).**
- קפיצה מוותנת שוויזן : **label s==t beq \$s, \$t, label** אם \$s==\$t קופץ ל-label והוא לפקודה הבאה (ובדומה bne שkopatzet אם (\$s!=t).

דוגמה נתון הקוד הבא ב-C

```
if (i == j)
    f = g + h;
else
```

f = g - h;

הוא יתורגם לקוד אסםבייל באופן הבא

```
bne $s0, $s1, not_eq # s0=i, s1=j  
add $v0, $s2, $s4 # f = g+ h  
  
j cont  
  
not_eq:  
  
sub $v0, $s2, $s4 # f = g - h  
  
cont:
```

- השמה מותנת: \$s < imm : \$t slt \$t, \$s משימה ב-\$t 1 אם \$s < \$t 0 ואחרת, בדומה ל-*i* אם \$s < \$t 1 ואם \$s > \$t 0.

IMPLEMENTATION OF CALLS

הגדולה הקוראת היא הפ' שקוראת לפ' אחרת, הנקראת היא הפ' שנקראת, הפרמטרים הם הערכים שמיוערים מהקוראת לנקראת, התוצאות הם הערכים שהנקראת מחזירה לקוראת וכתובות החזרה היא הכתובת בזיכרון הקוד של הפקודה שאחרי הקוראה לנקראת בקוד של הקוראת.

המחסנית היא אוצר בזיכרון שתוכנה יכולה להשתמש בו, והיא משתמש שמיירה של מידע לוקאלית בעת הרצת פ' באופן שמאפשר קרייה מקונגננת וחזרה מקרים של פ'. מבנה הנתונים עובד בשיטת LIFO ואפשר או לדחוף (push) אלמנט בראש המחסנית, או להוציא (pop) את הערך העליון במחסנית.

הערה לעתים אפשר לישת גם לערכים אקראים במחסנית, ובכל מקרה ניגשים לכתובות ביחס לכתובת ראש המחסנית - Top of Stack - כאשר כל דבר מתחת איינו ואליזי. זאת מושם שהמחסנית גדלה נגד כיוון הכתובות, כלומר הוספה ערך תיזו את TOS ארבעה בתים למטה.

SAVING REGISTERS DURING CALL AND RETURN

הנקראת לא יודעת מה הקוראת עשויה, ורק מצפה לפרמטרים נוספים ולהוציא תוצאות נכונות. לכן אם הקוראת משתמשת ברגיסטרים בלבד, הנקראת תזרוס אותם בלי לדעת שהקוראת צריכה אותם. כדי לשמור את המצביע לפני ואחרי קרייה לפ' יש מנגנון ; calling convention ; נניח ש-f (הקוראת) הייתה באמצעות חישוב כשלקרה -g (הנקראת). רק חלק מהרגיסטרים נשמרים, וקיימים גורמים שונים :

- \$f-\$0-\$1 הם רגיסטרים זמינים ולכון לאחריות f (הקוראת) לשמר אותם במקום אחר במהלך הקרייה ל-g.

- \$a7-\$s7 הם רגיסטרים סטטיים ולכן f מזכה אותם לא ישתנו, כך שגם לשובם לערך בטרם קראתה כדי ש- f תתפקד כמו שצריך.
- \$a3-\$v0 ו-\$v1 משמשים העברת והחזרת ארגומנטים ולכן הקוראת צריכה לשמור אותם אם היא רוצה להשתמש בהם בהמשך. עצמה קיבלה (או תחזיר) בתור נקראת.
- \$ra הוא הרגיסטר שמחזיק את כתובות החזרה מהפ', והוא נדרסט ע"י הפקודה `jal` בעת הקראה לפ' הנקראת, כך שאחריות השימור היא על הקוראת.
- \$\$sp, המצביע לראש המחסנית, שנדרש לכל הפ' על מנת לפעול באופן תקין, וכן הנקראת נדרשת לשזור אותו בעת סיום הקראה לפ'.
- את הערכים נשמר תמיד במחסנית.
- כדי לדוחף (ולשמור) את \$ra למחסנית השתמש בפקודות

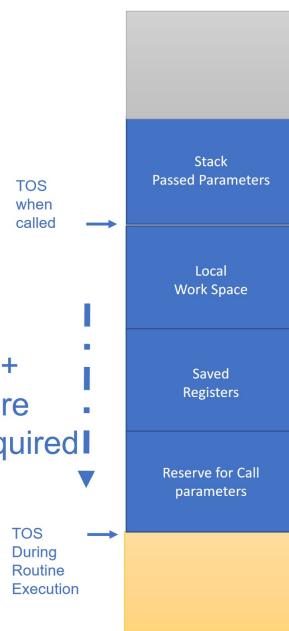
`addi $sp, $sp, -4`

`sw $ra, ($sp)`

כלומר קודם מזוהים את המצביע לראש המחסנית מילה אחת למילה בזיכרון (מעלים את גובה המחסנית) ואז כותבים לכתובות הבסיס של המחסנית את הערך של \$ra, כך שהוא עכשו בראש הערימה.

- כדי לגשת לערכים אפשר פשוט לבצע `lw` על כל היסט (חיובי) ביחס ל-\$sp.
- כדי לעשות `pop` קודם נקרא את המילה ואז נזיז כלפי מעלה את \$sp.

בעת קימפול נוכל לדעת כמה מקום פ' צריכה במחסנית (מבחןת משתנים מקומיים, שמיירת רגיסטרים ומקום לקרוא לפ' אחריות). מבחינה סידור הזיכרון בעת קראה לפ', המחסנית תראה כך



פקודות קרייה לפונקציה

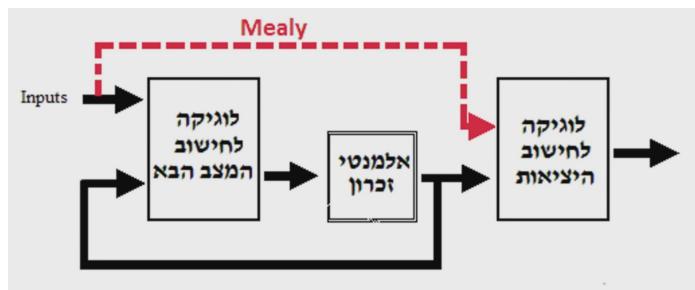
• `jal $ra` שמה ב-\$a את כתובות הפקודה הבאה (הערך הבא שיקבל PC) ואז קופצת ל-label.

• `$s $r $j` קופצת לכתובת שברגייסטר \$s.

כשנקרה לפ', נרץ `jal`, וכשנচזור מפ', נרץ `$ra $jr`.

תרגול

להדגמת ההבדל בין מכונות Moore ל-Mealy, ראו האירור הבא כאשר בשוחר Moore מכונה ובדום התוספת שהופכת אותה למכונית Mealy.



אנליזה של מעגלים סינכרוניים

בבינה מוגל, נרצה להבין מה הוא עשו (איזה מכונה הוא מייצג, איך הוא מתנהג). השלבים לאנליזה הם:

1. הצגת הקלטים לזיכרון ופלטי המוגל באמצעות הפלטים מהזיכרון והקלטים למעגל.

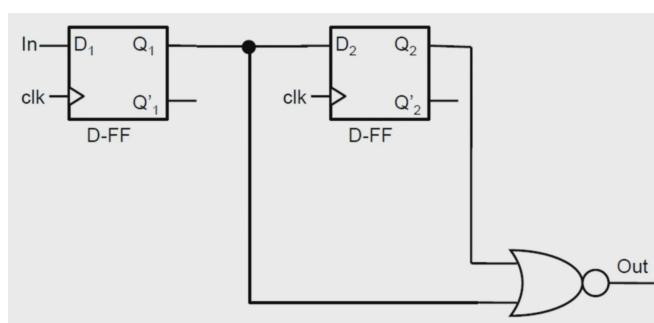
2. כתיבת טבלה מעברים.

3. כתיבת טבלה מעברים סימבולית (טבלה עם שמות למצבים).

4. רישום אוטומט מצבים.

5. ניתוח האוטומט באמצעות סדרת בוחן.

דוגמה נתון המוגל הבא



1. בקלטי הזכרו מתקיים $.Out = (Q_1 + Q_2)'$ ובפלט המعال מתקיים $D_1 = In, D_2 = Q_1$

2. טבלת המעברים תראה כך, כאשר אנחנו עוברים על כל אפשרות ל-

		In=0		In=1		
Q_1	Q_2	D_1	D_2	D_1	D_2	Out
0	0	0	0	1	0	1
0	1	0	0	1	0	0
1	0	0	1	1	1	0
1	1	0	1	1	1	0

כאשר חלק מהקומבינציות לא הgioיניות, אבל עדיין נכתבות אותן.

3. ניתן שמות למצבים, כאשר מצב מוגדר ע"י קיבוע ערכיהם של כל פלט רכיבי הזכרו, במקרה הזה יש לנו רק שני DFF-ים

	שמות המצבים	Q_1	Q_2
A		0	0
B		0	1
C		1	0
D		1	1

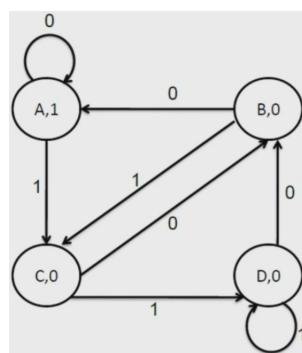
ועכשיו נוכל לחשב את הפלט של המعال בהינתן הקלט וה המצב הנוכחי בטבלת המצבים הבאה (PS המצב הנוכחי ו-NS המצב

הבא)

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

4. עתה נבנה אוטומט, כאשר נctrיך להחליט האם לבנות אוטומט Moore או Mealy, ובשלב הראשון עוד יכולנו לשים לב שהمعالג

Moore הלוגי שמנדר את הפלט תלוי רק בזיכרון ולא בקלט וכן נסתפק באוטומט Moore



5. נתח אוטומט, כשם שמענינו אותו בסופו של דבר היא באילו מקירים המכונה פולטה 1. נשתמש בסדרת בוחן, כולם טבלה עם

שלוש שורות: מצב, פלט, והקלט שאיתו נעבור במצב הבא בשורה. נdag שהמעברים בין כל שני STATES סמכים בשורת המצבים

יכסו את כל המעברים הקיימים באוטומט (כל הקשתות) לפחות פעם אחת. אין חשיבות לדזר המעבר.

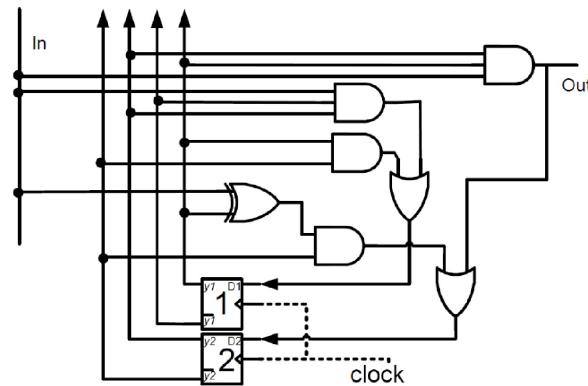
In	0	0	1	0	0	1	1	0	1	0	1	0	0	0
State	A	A	A	C	B	A	C	D	B	C	B	C	B	A
Out	ϕ	ϕ	1	0	0	1	0	0	0	0	0	0	0	1

כאשר שני הפלטים הראשוניים הם Φ כי כל עוד לא עברו שתי ייחידות זמן, לא יוכל לדעת מה הפלט של רכיב הזכרון השני (כי הקלט שלו מוגדר רק אחרי המחוור הראשון) שMOVABLE לפולט ולכן הפלט לא מוגדר.

עתה נחפש את ה-1-ים בפלטים, ונשים לב שכדי לקבל 1 בפלט, צריך שהקלטים בשני מחזורי השעון הקודמים יהיו 0, וזהו כולל השינוי! נשים לב שהקלט בזמן המחוור הנוכחי לא משנה כי מדובר במכונת Moore ולא Mealy.

איך נדע שצריך רק את שני הביטים שלפני המחוור הנוכחי ולא יותר או פחות? אפשר לבדוק כללי שינויי מרכיבים יותר (שמורכבים משלושה ביטים לדוגמה) ולשים לב שהם לא מתკיימים, כי אפשר לקבל 1 גם עם 100 (זה קורה בסדרת הבדיקה) וגם באמצעות (ע"י הישארות ב- A שוב ושוב). הכלל האופטימלי וה邏輯י הוא זה שמשמעותו אונטו.

דוגמה נתון המעגל הבא



כאשר הפלט היחיד הוא Out (והחצאים למעלה חסרי משמעות).

1. ניצג את הפלטים והקלטים לזכרון. ובנוסף $D_2 = y_2 \cdot y_1 \cdot In + y'_2 \cdot (y_1 \oplus In)$ ו- $D_1 = y_1 \cdot y'_2 + y_2 y'_1 In$.

2. מהצבת הערכים קיבל את בטבת המצביעים

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

3. נבחר שמות למצביעים (קייבוע פלטי הזכרון) כרגע

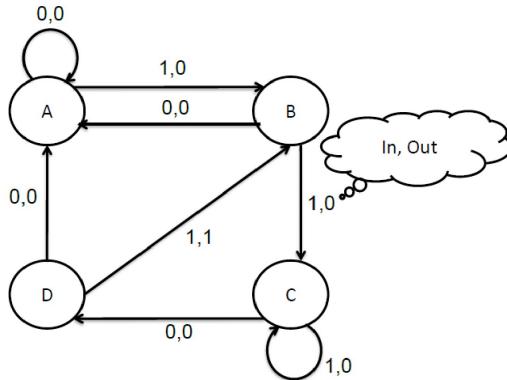
שמות המצביעים	Q_1	Q_2
A	0	0
B	0	1
C	1	0
D	1	1

ועתה באמצעות הצבה של השמות בטבת המצביעים קיבל כאשר הפלט כן משפיע על הפלט, כלומר מדובר במכונת Mealy

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

4. כדי לבנות את האוטומט נצטרך עכשו לשימוש באוטומט Mealy, כזכור של הקשת נכתוב איזה קלט מעביר אותו לUMB

הבא ואיזה פלט הוא מספק לנו



5. עתה נרשום סדרת בוון שתראה אותנו הדבר, רק שהפעם בחריפוש אחר כל השינוי נצטרך להתחשב גם בערך הנוכחי של הקלט

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

כאשר 1 מתקבל רק כאשר הקלט בזמןים $t+3, t+2, \dots, t+1$ היה .1,1,0,1

סינטזה של מעגלים סינכרוניים

בහינתן אפין, נרצה לממש מעגל סינכרוני שמקיים את הדרישות. נשתמש בסכמה הבאה :

1. בניית אוטומט בהתאם לדרישות.

2. טבלת מצבים.

3. קידוד מצבים ובחירה סוג FF.

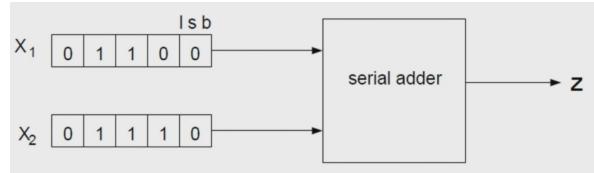
4. טבלת מעברים ופלט.

5. הגדרת פ' הכניסות של רכיבי הזיכרון ויציאת המעגל.

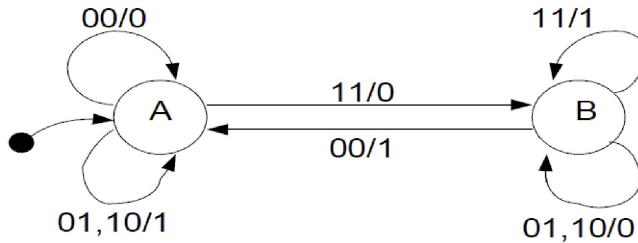
6. בניית המעגל.

דוגמה נרצה לבנות מסכם בינארי טורי, כולם מוגדר שמקבל קלטים x_1, x_2 וזמן t_i מחשב את הביט $h-i$ (מכיוון ה-LSB) בסכימת המספרים המתואימים על הסרט הנע של x_1, x_2 (כל מהזור מקבלים שני ביטים חדשים, שכайлו מותוספים כ-MSB למספרים שאנו סוכמים).

ראו איור (כל פעם סוכמים את הביטים המתואימים, כמוגן עם נשא מהסכימות הקודמות)



- ראשית נבנה אוטומט שמקיים את הדרישות, כאשר הקלט (מיון ל-/-) הוא שני הביטים $m-1-x_1$ ו- x_2 בהתאם. נבחר שהמצבים שלנו ייצגו האם יש לנו נשא מהחישוב הקודם, וכן לשמור את המידע הזה בזיכרון למשה. A מייצג מצב שאינו בו נשא מהחישוב הקודם ו- B מייצג מצב שבו יש נשא מהחישוב הקודם. בכל פעם נחשב את הסכימה יחד עם הנשא (אם יש כזה), ובחר מה הפלט של התוצאה ובנוסף האם יש לנו נשא ונעביר מצב בהתאם.



- نبנה טבלה מצבים, ש מכילה גם את הפלט במצב אליו עוברים כי הפלט תלוי בקלט הנוכחי כי זו מכונת Mealy

		NS (Next State), z (Output)			
		Input			
		00	01	11	10
PS (Present State)	A	A,0	A,1	B,0	A,1
	B	A,1	B,0	B,1	B,0

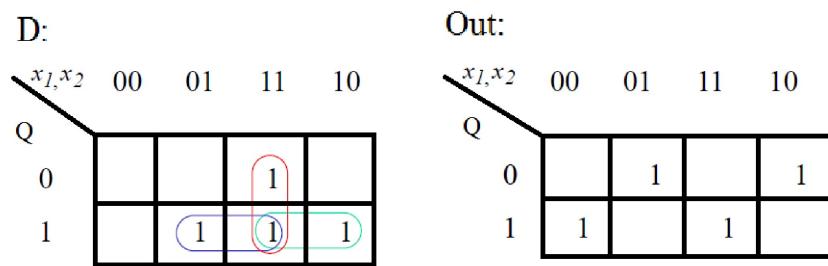
כאשר נשים לב שהקלטים לא מסודרים לקסיקוגרפיה אלא כמו בטבלה קרנו!

- קידוד המצבים דורש בחירות ייצוג בינארי למצבים, וכך אשר יש שניים זה די קל - מספיק ביט אחד שייצג את A כשהוא 0 ואת B כשהוא 1.
- נחליף את A ו- B בטבלה בביטויים המתואימים להם ונקבלת טבלה מעברים ופלט ש מכילה רק ביטים

		NS (Next State), z (Output)			
		Input			
		00	01	11	10
PS (Present State)	0	0,0	0,1	1,0	0,1
	1	0,1	1,0	1,1	1,0

5. נפצל את הטבלה לשתי מפות קרנו (אחת ל-D, הקלט לשילוב המצב הבא ואחת לפולט) וככשה את הטבלה כמו שעשינו

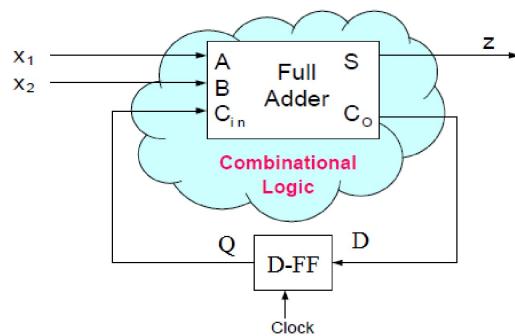
בתרגול 2



$$\text{Out} = Qx'_1x'_2 + Q'x'_1x_2 + Qx_1x_2 + Q'x_1x'_2 \quad \text{D} = x_1x_2 + Qx_1 + Qx_2$$

6. הסטודנטית הערנית תשים לב שהחישובים האלה הם בדיק הפלטים של FA ולכן (באופן שדי מותאים לאפיון) נקבל שימושו את המעגל

כך



שבוע VII | מעבד Single-Cycle

הרצאה

הערה למה שלא נשמר את כל הרגיסטרים לצד של הקוראת או הנקראת? כיזה מאד בזבזני בין היתר כי לא תמיד נדרש את כל הרגיסטרים שמורים. מה שענייפ הוא שכל צד ישמור חלק מהרגיסטרים שלו אצלנו ויישמר רגיסטרים אחרים בשילוב אחר. בצורה כזו לא נדרש

תמיד לשמר הכל, אלא רק מה שאנו צריכים לשמר (לדוגמה אם הקוראת השתמשה ב-\$3\$ ולא צריכה אותה יותר אחרי הקריאה לפ', היא לא צריכה לשמר אותה עצמה).

הערה ישן פקודות שלא קיימות ב-MIPS אבל שניות למימוש באמצעות פקודת אחת שכן קיימת, לדוגמה \$s \$d, \$s not נתן למימוש ע"י nor והאסטמבלר יוכל לתרגם את המקרים הפרטיים האלה.

קידוד פקודות MIPS

כל פקודה מוקדדת באמצעות 32 ביטים, ולא יכולה להיות בגודל דינامي, ולכן נדרש לנחל את תקציב הפקודות שלו (2^{32} סה"כ) באופן יעיל.

- פקודות עם שני רגיסטרים אופרנדים מוקדדת ב-Type R.

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

כאשר opcode הוא מזהה הפעולה (כל פקודות ה-R הן 0) שגודלו 6 ביטים; rs ו-rt הם אינדקסים הרגיסטרים של האופרנדים ו-rd אינדקס רגיסטר היעד (5 ביטים כל אחד); shamt מספר הביטים ל-shift כמשמעותו בפקודה shift (5 ביטים); ו- funct שגדיר איזו פעולה בדיק נבצע (לדוגמה ל-add ו-sub יהיו כאן ערכים שונים). functuous 6 ביטים לכך שיש לנו לכל היותר 64 פקודות מסווג R. נשים לב שככל פקודה תופסת הרבה מילימ (5 ביטים לכל אינדקס רגיסטר, כולל 32,000 כולם שמייצגות פקודות סכימה שלשה).

- פקודות עם רגיסטרים ו-immediates מוקדדת ב-Type I.

opcode	rs	rt	immediate
--------	----	----	-----------

כאשר rt הוא אינדקס רגיסטר ה-operand השני יחד עם ה-immediate (שהוא 16 ביטים) ו-rs הוא רגיסטר היעד. הרוב המכריע של opcodes משמש פקודות מסווג זה.

- פקודות קפיצה מוקדדת ב-Type J.

opcode	immediate
--------	-----------

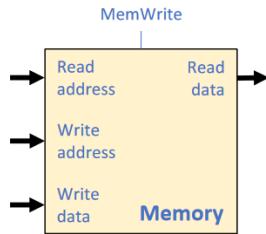
כדי שנוכל Kapoor למגוון כתובות בזיכרון (2^{26} כתובות), כאשר רק j-ojal מוקדות כך (שאר הקפיצות כוללות immediate).

הערה כתובות R-Type הן חci זולות, כי אפשר להכניס הרבה סוגים שונים של פקודות באמצעות שדות ה-opcode וה-funct. אם נרצה עוד פקודה מסווג R-Type, נפנה מיד לפקודה מיותרת מסווג I-Type, שתופסת שימושית יותר מקום בעבר פחות פקודות, ונשאבת ה-R-Type opcode שלה לצורך פקודות R-Type.

ביצוע פקודות MIPS

מעבד הוא בסה"כ מכונת מצבים ענקית, כאשר הבדיקה הברורה בין הלוגיקה למצב/זיכרון היא קצר יותר מרכיבת כי הפקודה רצתה איפשהו באמצעות (לפni ואחרי מצבים). עם זאת, ביצוע פקודה בסופו של דבר מעביר אותנו מצב (משנה את ה-PC וכו'). כזכור הזיכרון אמין מאורגן בביטחון, אבל הקריאה ממנו היא ביחידות של ארבעה בתים (מיליה), וכך גם PC גדול בקטגוריות של 4.

ברמת הארכיטקטורה הנוכחית, רכיב הזיכרון מציג את הממשק הבא



כאשר כתובות הקריאה והכתיבה הן 32 ביט, הוא המילה שנרצה לכתוב לכתובת הכתיבה (אם נכתב), ו-`MemWrite` הוא ביט שקבע האם נכתב או לא. הפלט הוא `Read data` שהוא המילה שנמצאת בכתובת הקריאה. משיק זה מאפשר לנו לספק תמיד את כל הקלטים, אבל לכתוב לזכרו רק אם אנחנו צריכים, וזה יקל علينا בהמשך במימוש המעבד.

בדומה, רגיסטרים הם אוסף `Flip-Flop`-ים.

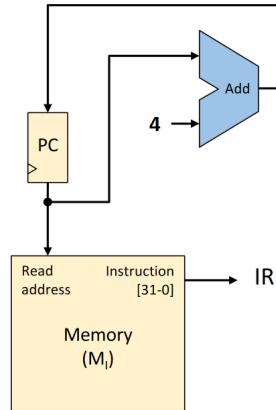
הערה כנסמן חץ באופן הבא ↔ אל תוך רכיב ומתוחתיו מספר, הכוונה שיש לנו `bus` שמורכב ממספר חוטים (מספר שכטבנו) ולא רק ביט אחד.

שלבי הרצת פקודה

- קראית הפקודה (fetch) : המעבד מספק כתובות (ששמורה ב-PC) לזכרו ומקבל את תוכנה, שהיא קידוד הפקודה שצורך להרץ.
- פענוח הפקודה (decode) : להבין איזו פקודה צריך לבצע ואילו רגיסטרים היא דורשת. בנוסף נזיה את ה-PC בהתאם לפקודה (קפיצה לכתובת אם מדובר ב-branch או jump ואחרת אינקרמנט).
- ביצוע הפקודה (execute) : לבצע את הפעולה המתמטית.
- גישה לזכרו (memory access) : נדרש כשהפעולה ניגשת לזכרו `load` יקרא ו-`store` יכתוב.
- כתיבה חוזרת (write back) : לכתוב את תוצאות החישוב או ה-`load` או ה-`branch`. השלב הזה למעשה משתנה בהתאם למצב המכונה לקרהת הפקודה הבאה.

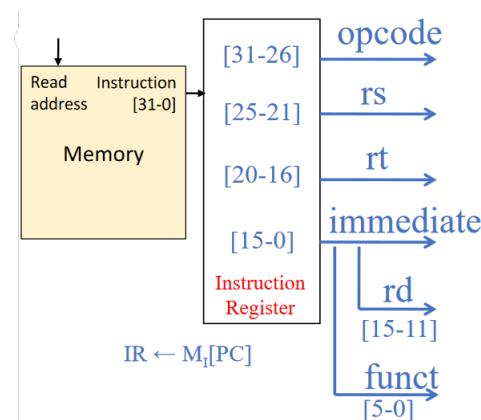
מימוש השלבים

- המימוש דורש חיבור בין רגיסטר ה-PC לזכרו וגם אינקרמנטציה, כלומר יראה כך



כאשר IR הוא רגיסטר שמכיל את הפקודה הנוכחי, כי $IR = M_I[PC]$ (עבור M_I זיכרונו הפקודות).

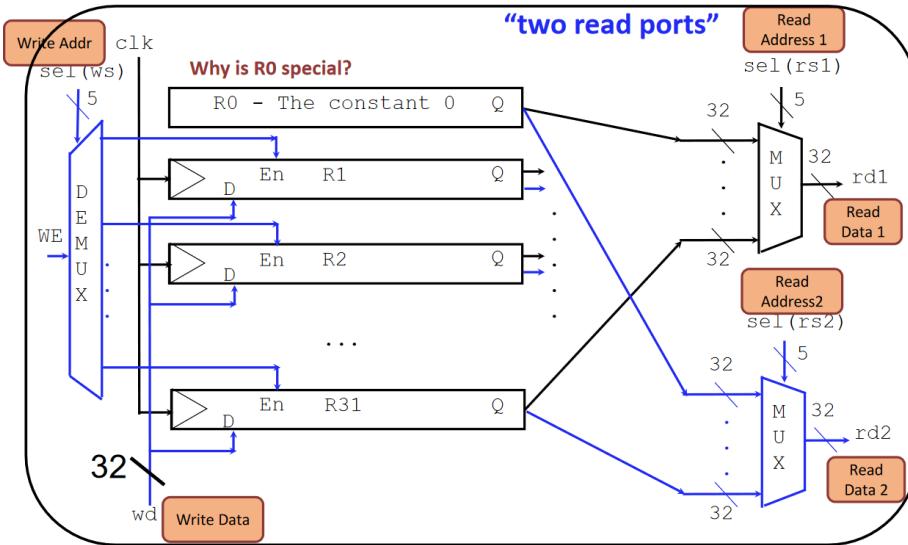
- פענוח: נפצל את הביטים ברגיסטר הפקודה (IR) למשמעותו הרגולונטיו שליהם (ראו איור)



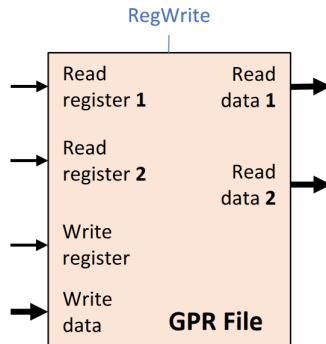
ובנוסף נביא את הריגיסטרים הנדרשים לפעולה, שמוקמים ב-Register File, Register File-ים שבו הריגיסטר הראשון הוא בעצם קבוע 0 ולא רכיב זיכרונו וה-31 האחרים הם רכיבי זיכרונו אמיטיים. כדי לקרוא מהם מידע, צריך להשתמש ב-Mux אחד מבין 32 רגיסטרים ומוציאה את התוצאה החוצה. הכתובת שנקרה תלואה כמובן ב- rs , rt ו- rd לפי הצורך. ה-Mux הוא בעצם מערך של Mux 32-ים שכל אחד בורר בין 32 אפשרויות.

כל אחד ממה-Mux32-ים ניתן למימוש עם 32 NAND-ים ועוד NAND עם 32 כניסה, כלומר למעלה מ-1000 שערים לכל לוגיקת הקראיה מה-File.

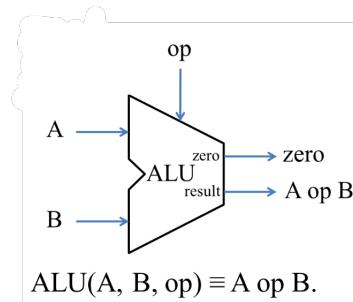
כדי לכתוב בהמשך לריגיסטרים (בשלב ה-DeMux) נחבר את ה-DFF-ים ל-Mux שnbrור אליו את הריגיסטר הנוכחי אליו אנחנו רוצחים לכתוב אליו באמצעות בית WriteEnable. האיסולטורציה הבאה מדגימה את כל האמור.



כל הרכיב באיור נקרא יחד ה-Register File, והוא מציג את הממשק הבא

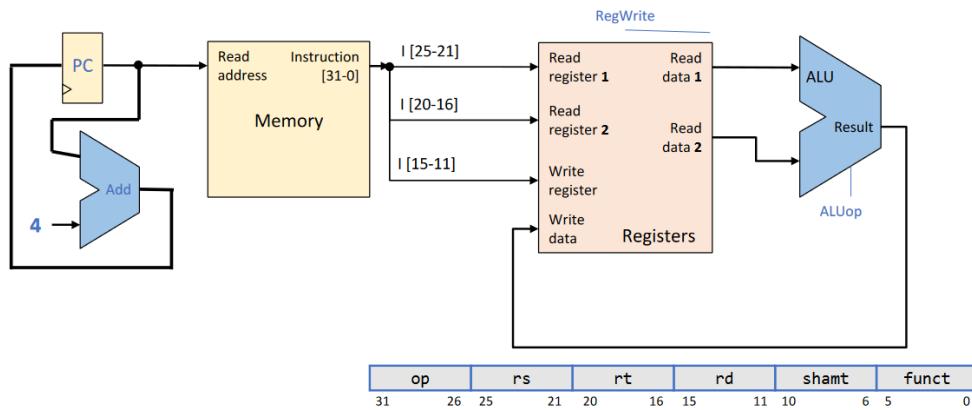


• ביצוע: הרכיב שהחראי על החישוב האריתמטי נקרא Arithmetic Logic Unit (ALU), שמציג את הממשק הבא



כאשר הפלט zero הוא ביט (דיל) שערכו 1 אמ"ם פלט החישוב הוא 0 (שימושי מאד לקפיצות מותנות שוויה/אי-שוויה).

עבור הרצת פקודת R-Type, כבר יש לנו את כל הרכיבים הנדרשים והתהליך יראה כך (cronologית משמאלי לימין).



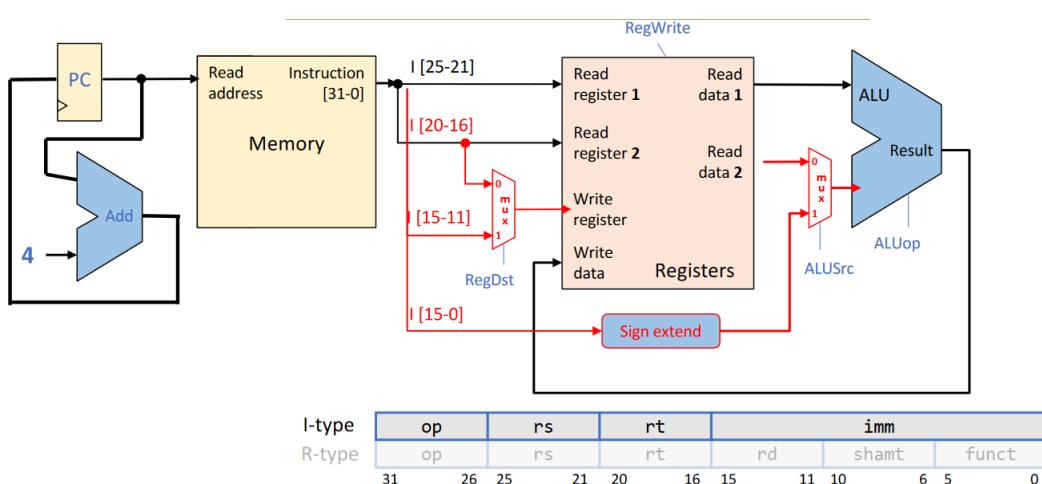
הערה המוקם היחיד שבו נדרש לעליית שעון כדי להתקדם הוא בהתקדמות ה-PC, שכן עד לחישוב ההתוצאה ב-ALU אין שום לוגיקה שאינה כירופית, וכיילו לכתוב את המידע לרגיסטרים צריך עוד עליית שעון. לעומת זאת, מוחזר אחד לכל פקודה (עד כדי setup time ו-hold time), מכיוון בא הזמן (time).

כדי לממש פקודות I-Type, נבצע את אותה הפעולה של R-Type עם כמה שינויים קטנים:

1. ניקח את תוכן ה-imm מהפקודה ונעשה לה sign extend (הרחבת המספר מ-16 ביטים ל-32 ביטים כפי שראינו בעבר) ולכן נctrיך לקרוא רק מכתובת אחת ב-RegFile (נספק זבול ל-sel כתובות הקריאה השנייה). את הבדיקה במה להשתמש ממש באמצעות Mux שבורר לפי סוג הפקודה.
2. כתובת הכתיבה צריכה mux לפניה בדומה לנ"ל כי ב-Type R כתובים לרגיסטר rd (האינדקס השלישי בקידוד) ואילו ב-Type I מדווח ברגיסטר rt.

הערה בגלל ש-Imm ו-funct הם opcode, הם יctrיך להגיד לנו מה אם מדובר ב-Type R-Type ו- I-Type I-Type. כדי שנוכל לפרש את הביטים נכון.

האיור הבא מציג מעבד שיכל להריץ פקודות I ו-R מתמטיות, עם אינדקסי הביטים בקידודים למטה

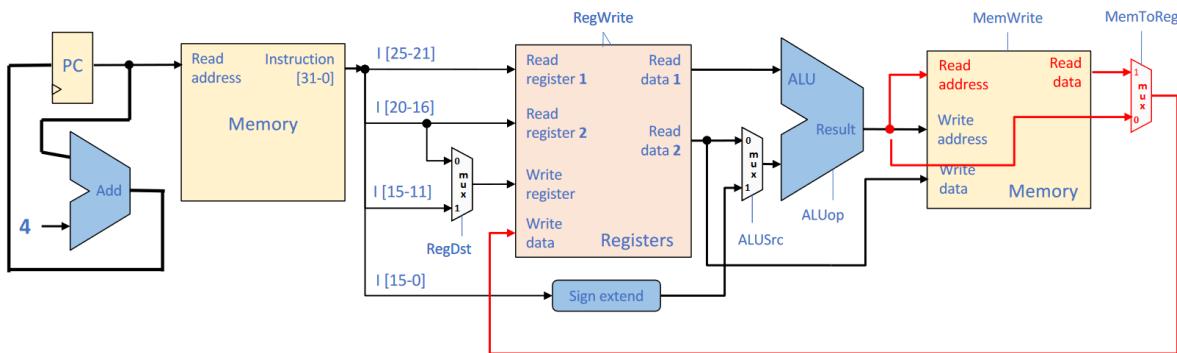


Reg [IR [rt]] : השלב הזה חיוני למימוש כתיבה וקריאה, כאשר בשתי הפקודות מעתיקים את הערך של write back memory access •
 $M_D [Reg [IR [rs]] + SignExtend (imm)]$ או להפוך, בהתאם.

כדי למש את הפעולה נשתמש ברכיב הזכרון Data Memory שמקבל קלטי Addr ו-Din (32 ביט כל אחד) ו-WE (ושעון כי זה זכרו)
 ופלט Dout (32 ביט) כאשר אם WE=0 אז Dout מכיל את תוכן הקריאה ואחרית הפלט לא מעניין אותו.

בשביל הקריאה וגם הכתיבה ניתן ל-Data Memory את כתובות הקריאה והכתיבה שהיא פلت ה-ALU (שיסכם את הרגיסטר יחד עם ה- imm לכדי כתובות אחת). תוכן הכתיבה יהיה פשוט תוכן הרגיסטר השני שקרהנו מהרגיסטרים. פلت הזכרון יבורר יחד עם פلت ה-ALU לפי האם אנחנו קוראים מהזיכרון (פלט הזכרון) או מבצעים חישוב מתמטי נטו (פלט ה-ALU).

האיור הבא מדגים את המעבד שבנו עד כה, שיכל להריץ כל פקודה מסוג R או I



תרגול

דוגמה הפקודה

100011 00110 00101 0000 0000 0000 0000

מבצעת (lw \$5, 7(\$6) כאשר ה- opcode הוא 35 קריאה מהזיכרון, rt, rs כrangle מקודדים את אינדקס הרגיסטר ממנו נקרא ואליו כתוב ולבסוף ה- imm שהוא ההיסט מ- rs שנוטן את הכתובת.

הערה J-type מקודד את כתובות הפקודה ביחידות של מילה במקומות הביתי, וכך אפע"פ שיש לנו 2^{28} ביטים, נוכל ליציג מרחב זיכרון בגודל $.2^{28}$

דוגמה את השורה $a = A[6] - b$ נממש באמצעות שתי פקודות: קריאה של $A[6]$ לרגיסטר זמני ושם הפרש ברגיסטר נוסף.

דוגמה נביט בקוד הלולאה הבא

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

אנו יכולים לכתוב כפסאודו-קוד אסמבלי

```

Loop: g = g + A[i];
i = i + j;
if (i != h) goto Loop;

```

מכאן נקצת רגיסטר לכל אחד מהמשתנים, לדוגמה $g = \$s1$, $h = \$s2$, $i = \$s3$, $j = \$s4$ וכתובת הבסיס של A נשמר ב- $\$s5$. סיום
ונכל לתרגם זאת לשפת אסםביי אמיתית

```

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

```

כאשר שלושת השורות הראשונות הן עצם העניין: כדי לקרוא מהמערך, נחשב את היחסט i מראשיתו של המערך באמצעות הכפלת הרגיסטר ב-4 (כי הכתובות כשහן לא ב- $mimm$ הן ביחידות של בתים), נוסף אליו לבסיס של A ונטען את המילה מהזכרונו. משם סתם עושים אריתמטיקה עד לתנאי הלולאה ש קופץ חזרה להתחלה רק אם i שונה מ- h לאחר השוואת היחסטים המתאימים להם (ne משמעו **Not Equal**).

דוגמה switch-case אפשר למש באופן הבא, והימוש כمو奔 שקול לשרשור ארוך של if-else-ים.

לכל הסגר ב-switch-case : לפני פקודות התוכן של ההסגר, נוסף :

- פקודת addi שת חשב את ההפרש בין המשתנה למועדד ההשוואה בסתגר |

- פקודת bne שת בדוק האם ההפרש שונה מאפס ואם כן תקפוץ לתוויות של ההסגר הבא (אחרת לא נקפוץ ונשאר להריץ את תוכן |

הסגר הנוכחי).

לאחר סיום תוכן הסגר נוסף פקודת j לסוף switch-case כלו, כך שלא נרץ בטעות הסגרים אחרים (בדומה לפקודת the-break).

דוגמה את ההתנית if (g < h) goto label אפשר בклות באופן הבא (בנחתה שהמשתנים הם ב- $\$s0$, $\$s1$ בהתחאמה)

```
slt $t0, $s0, $s1 # g<h ? 1 : 0
```

```
bne $t0, $0, label
```

דוגמה נפענח את הקוד הבא

```

begin: addi $t0,$zero, 0
        addi $t1,$zero, 1
loop:   slt $t2,$a0,$t1
        bne $t2,$zero, finish
        add $t0,$t0,$t1
        addi $t1,$t1,2
        j loop
finish: add $v0,$t0,$zero

```

```

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

```

ועתה נבדוק מה הקוד עושים באמצעות טבלת מעקב לדוגמה, ונקבל שזה סוכם את כל המספרים האי-זוגיים בין 1 ל-n.

דוגמה נמיר את הקוד הבא מ-C לאסמבלי MIPS כאשר העריכים ההתחלתיים של a ו-b נמצאים ב-\$a0 ו-\$a1 בהתאם, וכתוות הבסיס של

.\$s0 A B-\$

```

int mult(int a, int b) {
    int value;
    value = 0;
    if (a < 0) {
        a = -a;
        b = -b;
    }
    while (a != 0) {
        value += b;
        a--;
    }
    A[4] = value;
}

```

את value נשים ב-\$t0, ונטיק את ערכו של a ל-\$t1 ושם נבצע לו דיקרמן. בולאה נקבע לסיום אם \$t1 הוא 0 ואחרת נרים את תוכנה. לבסוף נכתוב את תוכנו של \$t0 = value לミלה החמשית במערך. הקוד כולו הוא

```

Begin: add $t0, $zero, $zero
        slt $t1, $a0, $zero
        beq $t1, $zero, Loop
        sub $a0, $zero $a0
        sub $a1, $zero $a1
Loop:   beq $a0, $zero, Finish
        add $t0, $t0, $a1
        sub $a0, $a0, 1
        j Loop
Finish: sw $t0, 16($s0)

```

שבוע VII | מעבד I Multi-Cycle

הרצאה

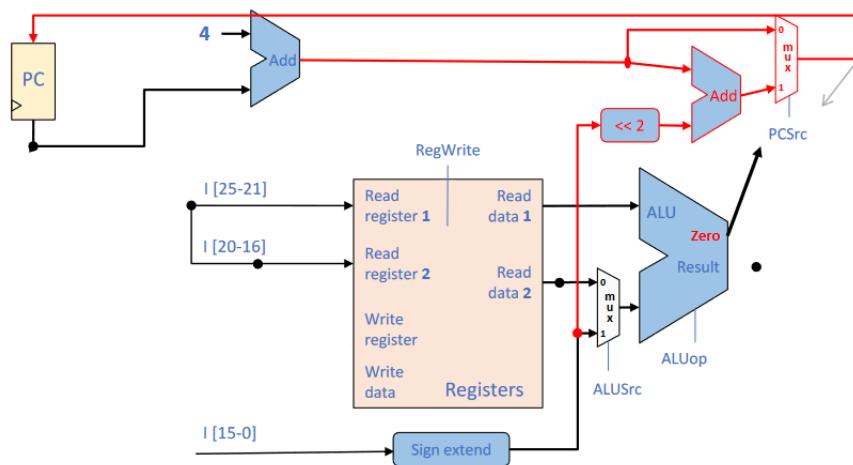
עד כה בנו מעבד שיכל להריץ כל פקודות I או R, כך שכל שנוצר הוא קפיצות וחתניות. נמשח באופן הבא:

1. נקרא את הפקודה מזכרון הפקודות.
2. נקרא את הרגיסטרים t_1 , t_2 מה-.register file
3. נחשב sign extend (shift b-2 שמאליה) את ההיסטוריה אליו נדרש אם מתקיים התנאי.
4. נחשב את הכתובת הבא אם לא נקפוץ, ונקבע את הכתובת אליה أولי נקפוץ לחוות $2 \ll 2$.
5. ה-ALU ישווה את הרגיסטרים לפי בהתאם לסוג ההשוואה.
6. PC יעודכן ל- $PC + 4$ או לכטובת במקרה הקפיצה, בהתאם לתוצאה ההשוואה.

דוגמה הפקודה `beq $t1, $t2, -0x0040` תבצע

$$PC = ((Reg[t1] - Reg[t2] == 0)) ? ((PC + 4) + (-0x0040) \ll 2) : (PC + 4)$$

מבחינת החיווט החדש שנדרש, באדום ניתן לראות את הרכיבים שנוספו לנו כדי לספק את התיאור הנ"ל.

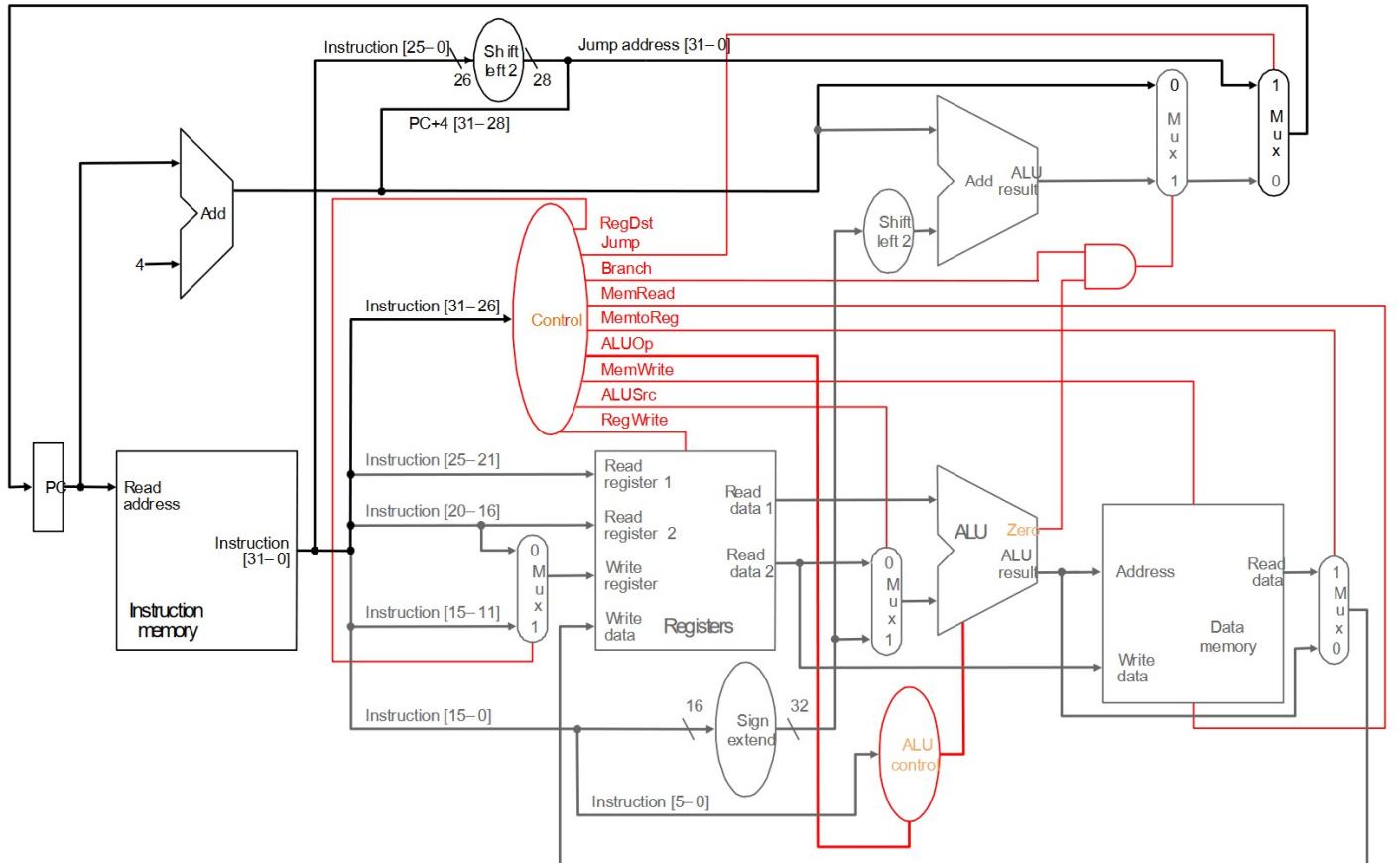


פקודות מסוג J-Type נמשח בדומה, כאשר עתה נעדכן

$$PC = (PC + 4)[31 - 28] . (offset \ll 2)$$

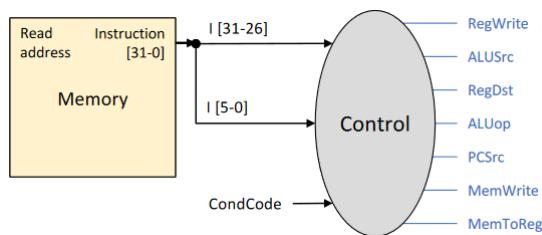
כasher ה-. היא פוללת הצמדה (concatenation) offset וה-ה תרגום של הפרש הכתובות בין כתובת הפקודה הנוכחי לכתובת ה-.label

לxicom המעבד השלים שלו נראה כך



יחידת השליטה

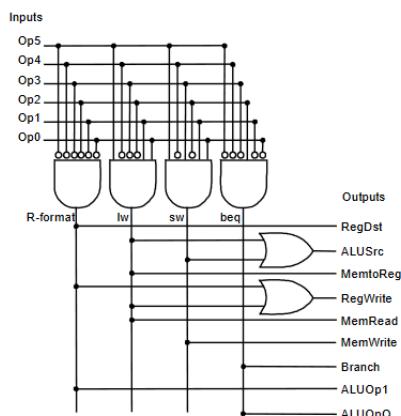
עד כה המעבד שלנו כולל כל מיני ביטים שימושיים כשלקוטורים ל-Mux-ים ורכיבים אחרים (מסומנים בכחול ולא מגיעים למקום מסוים). את כולם אנחנו מחשבים על בסיס שלב הפענוח של הפקודה. היחידה שלוקחת את ה-opcode (funct) של R-Type (גמ את ה-RegWrite) ואת השדות השונים שלוחטים על הריצה נקרא ה-Control Unit.



דוגמה להלן טבלת אמת שכוללת את ערכי השדות שמחשב ה-Control Unit עברו מספר פקודות

Instruction	RegDst	RegWrite	ALUSrc	ALUOp (alu_ctl)	MemWrit e	MemToReg	PCSrc
add	1	1	0	10 (010)	0	0	0
sub	1	1	0	10 (110)	0	0	0
or	1	1	0	10 (001)	0	0	0
addi	0	1	1	10 (010)	0	0	0
lw	0	1	1	00 (010)	0	1	0
sw	x	0	1	00 (010)	1	x	0
beq	x	0	0	01 (110)	0	x	0/1

דוגמה בהתעלם מפקודות אРИטמטיות מסוג I-Type, ניתן למשם ביעילות את המולטי-פונקציה הזו באמצעות שער AND שיזהו את סוג הפקודה ו-OR שיחשבו את ערך השדה על בסיס סוג הפקודה



השיטה נקרא Logic Array.

הערה גם את ה-control Control בתוכה ALU (חאם התוצאה היא אפס, שלילית וכו') ניתן למשם באופן דומה ולא מעוניין.

מעבד רב-מחזורי

נמשך את המעבד בדרך אחרת תחת אותו פקדות.

הבעיה ב-Single Cycle MIPS היא שהכל קורה בזמן מחזור אחד, כלומר זמן המחזור חסום מלמטה ע"י t_{pd} של המסלול הקרייטי (שהוא פקודת SW) כלומר

$$t_{cycle} \geq t_{fetch} + t_{decode} + t_{execute} + t_{memory} + t_{writeback}$$

הגדרה לכל פקודה ב-ISA נגידר CPI, שסופר את מספר המחזורים שנדרשים לביצוע פקודה.

דוגמה עבור Single Cycle MIPS CPI הוא 1 לכל פקודה.

זמן שלוקח להריץ תוכנית היא $IC \times CPI \times t_{cycle}$, כלומר מספר הפקודות בתוכנה. המטרה שלנו היא למזער את זמן הריצה של תוכנה, גם אם מספר המוחזורים לפקודה עלה (כי זה יכול לאפשר לנו להוריד את זמן המוחזור שימושית).

- כדי למזער את t_{cycle} צריך למש את המעבד באופן יותרiesel, או להפריד כל פקודה ליותר מוחזורים.
- כדי למזער את CPI צריך למש פקודות בפחות מוחזורים.
- כדי למזער IC צריך למש קומפיילר יותר חכם שמצוצם פעולות נוספות.

איך נחשב CPI של תוכנה מסוימת? נחשב כך

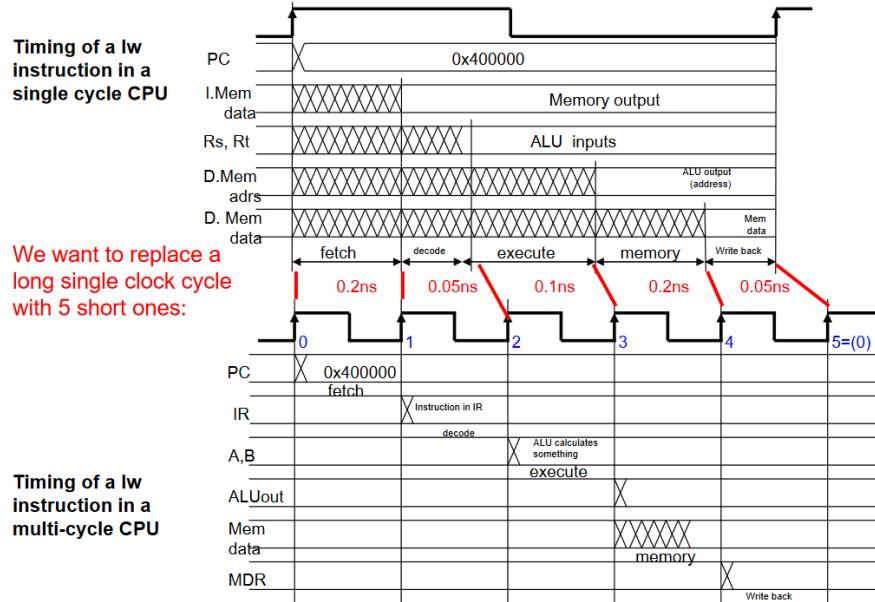
$$CPI = \frac{\#cycles}{IC} = \frac{\sum_i CPI_i \cdot IC_i}{IC} = \sum_i CPI_i \cdot \frac{IC_i}{IC} = \sum_i CPI_i \cdot F_i$$

שיטות השוואת ביצועי מעבדים

- הריצה על חומרה אמיתית: קל להריץ על פני הרבה זמן, קשה לנתח לעומק. מה נשווה בין מעבדים?
- לרוב נרץ benchmarks סטנדרטיים של פעולות מקובלות כמו `compression`, `office`, תוכנות, קומפיילרים וинтерפרטרים.
- אפשר לחשב ביצועים מקסימליים (MFLOPS ו-MIPS) אבל הם לרוב לא ריאליסטיים כי אי אפשר להגיעה אליהם בრיצה סטנדרטית.
- אפשר, אבל בפועל קשה, להשוות את ה-IC וה-CPI אחד-לאחד עם מעבדים אחרים כי אם ה-ISAs משתנה אז ההשוואה חסרת תוכן.
- סימולטוריים: יותר איטי מביצועים למציאות ו מגביל את אורך הריצה ושאר המשאבים.
- אנליזה: חישוב תאורטי של ביצועים על פני תבניות שימוש שונות.

אם נפריד סוג פקודות שונות ונבדוק את זמן הריצה שלהם (t_{pd} , נгла שחלק מהפקודות אפשר למש ביעילות כי זמן נשרף (לדוגמה פקודות שלא כותבות לזכרו). נחלק כל פקודה לחמשה חלקים (חלוקת הקונוניים של ביצוע פקודה שראינו בהרצאה בעבר).

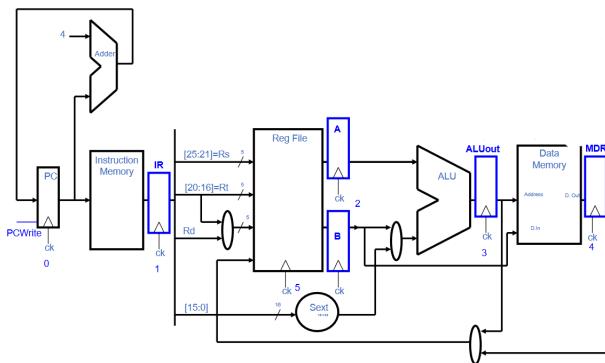
דוגמא נפצל את הפקודה `lw` למוחזר שונה לכל שלב, כלומר במקום מוחזר אחד לחמשה שלבים, מוחזר אחד לכל שלב.



נשים לב שנכלל תדריות הרבה יותר גבואה אבל מבחינה ביצועים עבור השלבים שאינם הארוכים ביותר, אנחנו מזבזים זמן עכשו, ככלומר הרצת לנו לוקחת יותר זמן. לעומת זאת, פקודות אחרות לא ידרשו את כל השלבים ולכן כן יהיה יותר קצריים.

כדי לאפשר את הרכזה עם מספר שלבים, נctrיך לזכור מה קרה בשלב הקודם, ונהשה זאת באמצעות רגיסטרים שייזיקו את תוצאת הפקודות

בכל שלב (ראו אייר)

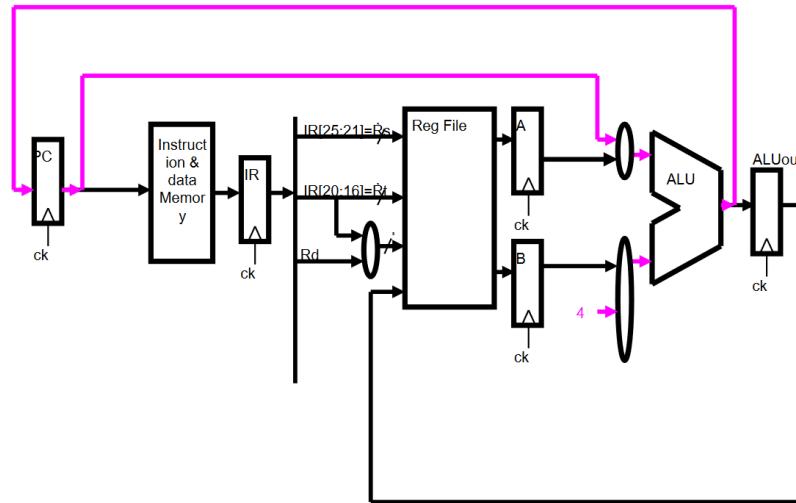


איך הגיעו להפרדה זו? נביט במסלול של פקודת lw : מתחילה fetch ואז קריאה מריגיסטרים, מעבר ב-ALU ולסיום כתיבה חוזרת לרגיסטר. לכן נפריד (1) בין fetch לקריה; (2) בין הקריאה לחישוב; (3) בין החישוב לכתיבה חוזרת לרגיסטרים; (4) ולפני כתיבה מהזיכרון לרגיסטרים.

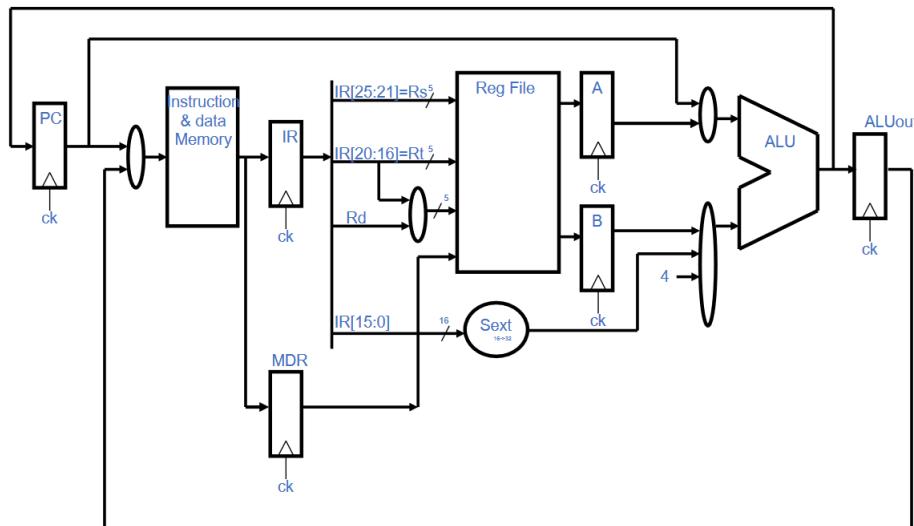
הערה כרגע חסירה לנו לוגיקה עדין שמודאגת שהמעבד יבצע רק את השלב שצרכי כרגע. כי אחרת הlw קורה בכל מחזור שעון ולא רק במקרים מסוימים לfetch.

איחוד יחידות

- ה-fetch קורא מהזיכרון אבל גם עושה אינקרמנט ב-4 בכל פעם. במקומות שבהם מוחבר נפרד שדורש טרנזיסטורים, משתמש ב-ALU שבסך כל מקרה כרגע לא פעיל (ראו חוטים חדשים בסגול), וזה יעזור לנו לקיצרו פעולות branch בעתיד.



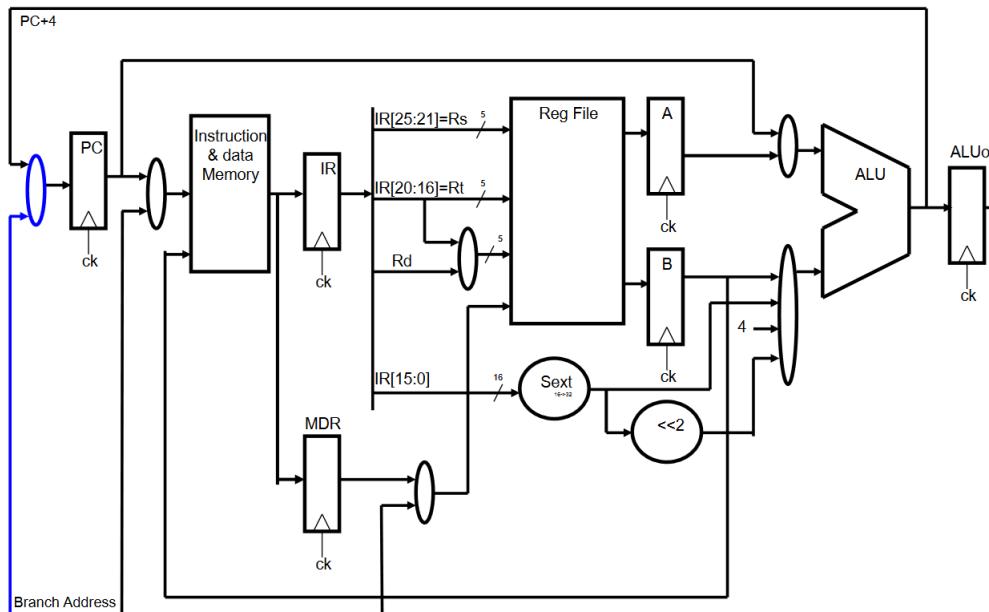
- בשליל הבנת המעבד, הפרדנו את הזיכרון של הכתובות והدادטא, אבל אין באמת סיבה לעשות זאת זה וזה מאד יקר. נשים לב שברגע שאנחנו ב-multicycle, יכולים לא להשתמש בשתי יחידות הזיכרון באותו זמן מזמן, ולכן אפשר פשוט לאחד אותן עם אונט שմבטיאizia מדע אנחנו רוצים. עדין נפריד את התוצאה לרוגיסטרים השונים כי נרצה לזכור מה המצב שלנו כמו שצורך ואם הכתובת והدادטא היו באותו רוגיסטר המצב לא היה מוגדר היטב.



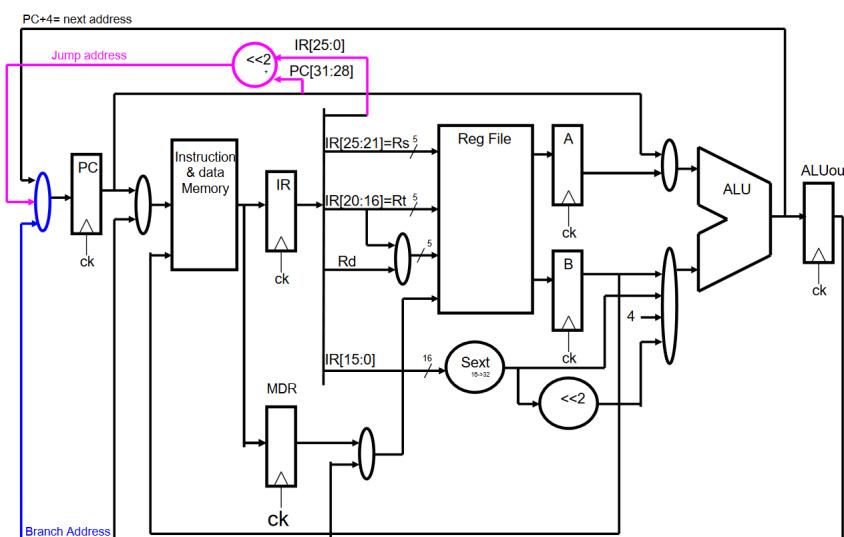
מספר המוחזרים לפי סוגי פקודות

- R-Type : 4 מוחזרים (fetch, פענוח וקריאה מהרוגיסטרים, חישוב ב-ALU וכ כתיבה חוזרת לרוגיסטרים), במקומות חמישה כי אנחנו לא קוראים מהזיכרון - חסכנו זמן!
- Iw : כל 5 המוחזרים (fetch, פענוח, קריאה מהרוגיסטרים, חישוב ב-ALU, קריאה מהזיכרון וכ כתיבה חוזרת לרוגיסטרים).

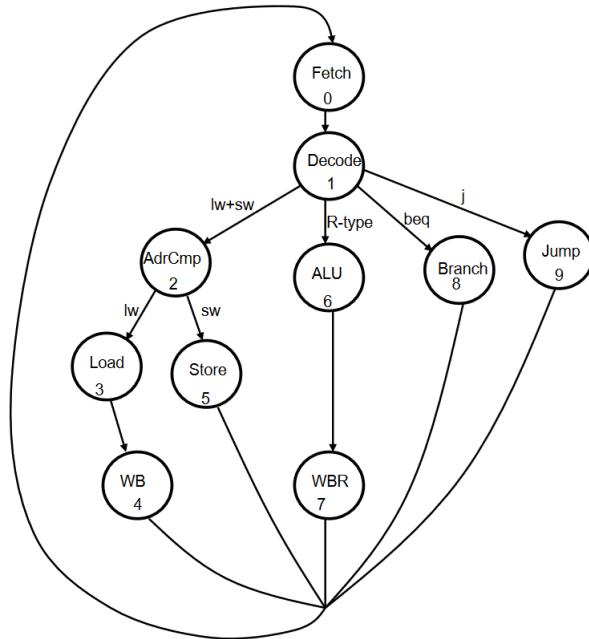
- sw : fetch, פענוח, קריאה מהרגיסטרים, חישוב ב-ALU וכתיבה לזכרון, כאמור שוב במקום חמשה.
- beq : 4 מחזוריים (fetch, פענוח, קריאה מהרגיסטרים, חישוב התנאי ב-ALU וכתיבה לזכרון), אולם חישוב הכתובת עם היחסט במקורה של קפיצה וכתיבה ל-PC, כאשר חיבור ה-PC ל-ALU, כפי שראינו באיחוד היחידות, מאפשר לנו לחשב את היחסט מ-PC.
- הבעיה כרגע היא שאם התנאי מתקיים זה לוקח שלב אחד יותר מאשר אם הוא לא, ונרצה שזה יקח מספר פעולות קבוע.
- לכן נוכל לחשב את היחסט (שודרש רק את ה-mmimm של הפוקודה) בזמן הפענוח ונסמוך את התוצאה ב-ALUout (שבו כרגע אף אחד לא משתמש), ואז נחשב את התנאי, ורק אז נכתב ל-PC או את הכתובת המוחשבת או סתם אינקרמנט ב-4. חשוב לשים לב שאין לנו דרך לשים את החישוב באותו זמן מחזורי ב-PC. כך יידנו ל-3 מחזוריים (fetch, פענוח וחישוב היחסט וחישוב ב-ALU וכתיבה ל-PC).
- הסטודנטית המשקיפה תסתכל על המעבד החדש שנבנה (באירור) ותוכח שאכן נדרשים רק שלושה מחזוריים כדי לבצע את beq.



- J-Type : 2 מחזוריים (fetch, פענוח וקפיצה) כאשר בין הפענוח לחישוב הערך שייכנס ל-PC במחזור הבא אין אף אחר כך זיהור באותו המחזורי.



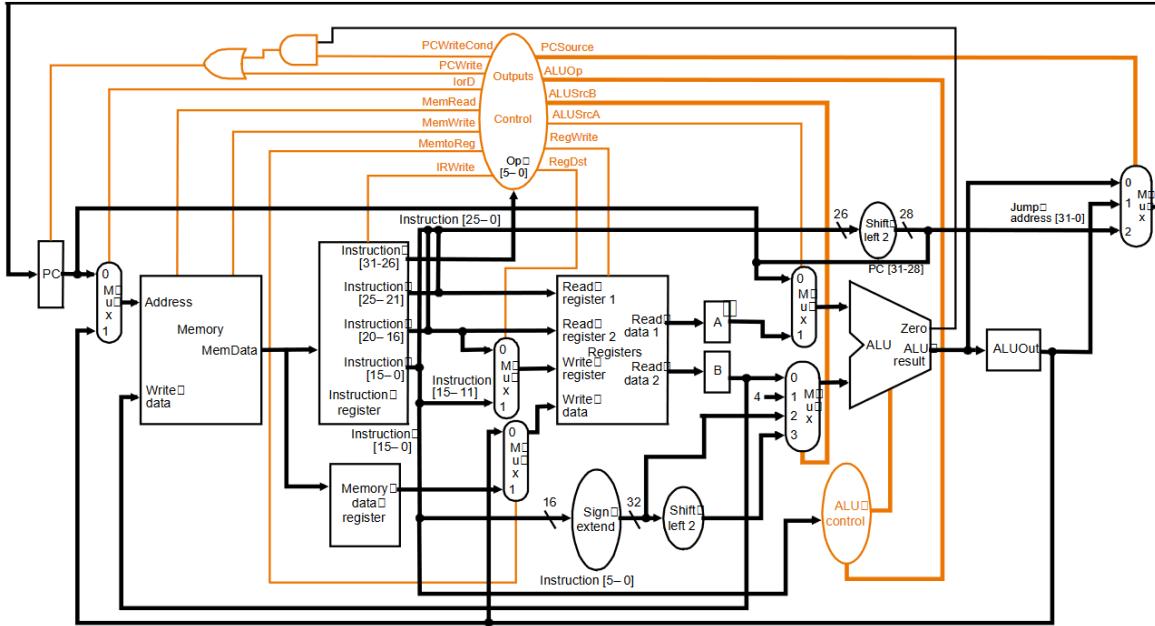
במהלך ניתוח כל אחת מהפקודות, בינויו בצד מכונות מוצבים שמייצגת את המעבד, כי בסופו של דבר בעת מימוש המעבד באמצעות תהליכי הפעלה. הסינטזה של מדונו נדרש לבנות מכונות מוצבים. להלן הדוגמה, שתואמת בדיקת ניתוח ה"ל של כל אחד מהשלבים.



הרכיב היחיד שעוד לא תתייחסנו אליו זה הקונטרול, שדוגג לכל המוקדים ולשליטה על השלבים השונים, והוא מוטמע במעבד המקורי (כמווב בlij הפרטים המלאים בתחום הרכיב) באופן הבא.

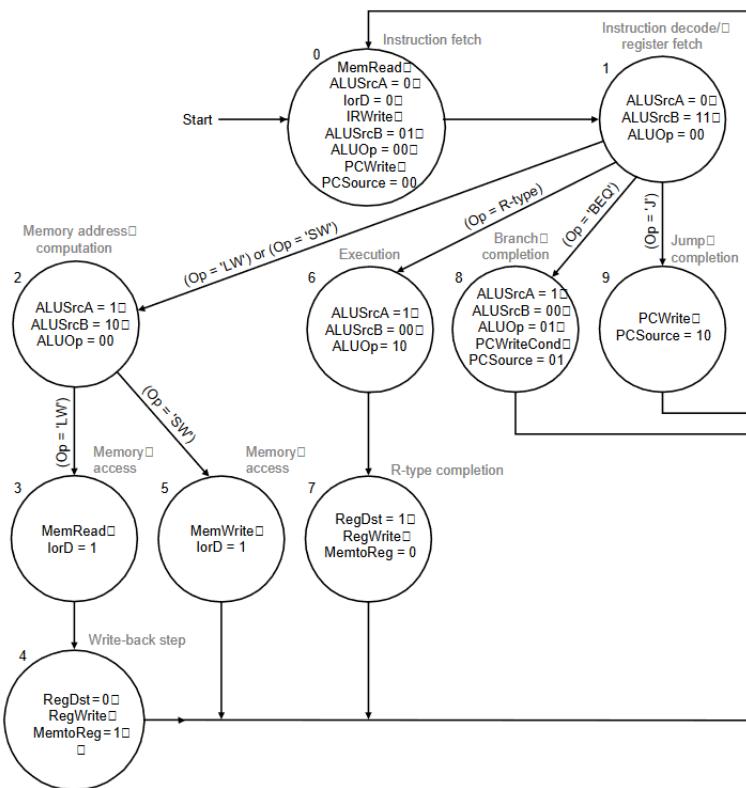
נשים לב שנוסף לנו בית חדש PCWriteCond שמחיליט האם אנחנו הגיעו בשלב ה-fetch ולכן צריך לקדם את המעבד. בנוסף יש לנו גם PCSOURCE ו-PCWRITE שקשורים למימוש של פקודות קפיצה מותנת ולא מותנת (jal). לסיום נוסף לנו IRWrite האם מעודכנים העשויו את תוכן וגייסתו ה-IR, שמכיל את הפקודה שכותבתה הופיע בזמן השעון הקודם ב-PC. אם הביט הזה דлок, פירוש הדבר שאנו העשויו_decode בשלב ה-decode.

כדי שיחידת הבדיקה תדע באיזה שלב היא נמצאת כרגע ומה שלב הבא, היא צריכה שיהיה לה איזושו זיכרון פנימי (כמה DFF-ים) שמשמשים את הלוגיקה של אוטומט המוצבים ה"ל, אבל אנחנו נתעלם ממנו.



מכונת המצלבים יחד עם הקונטROL שמנדר באיזה שלב אנחנו בכל פעם נראת כך (מקרה : דוגלים שלא כתוב אם הם 0 או 1 הם 1, הכוונה

לדגלים מאפשרים ולכן אם הם מוזכרים הם דלוקים ; מה שלא מסומן לא השתנה מהשלב הקודם ; הריבועים הלבנים חסרי שימוש)

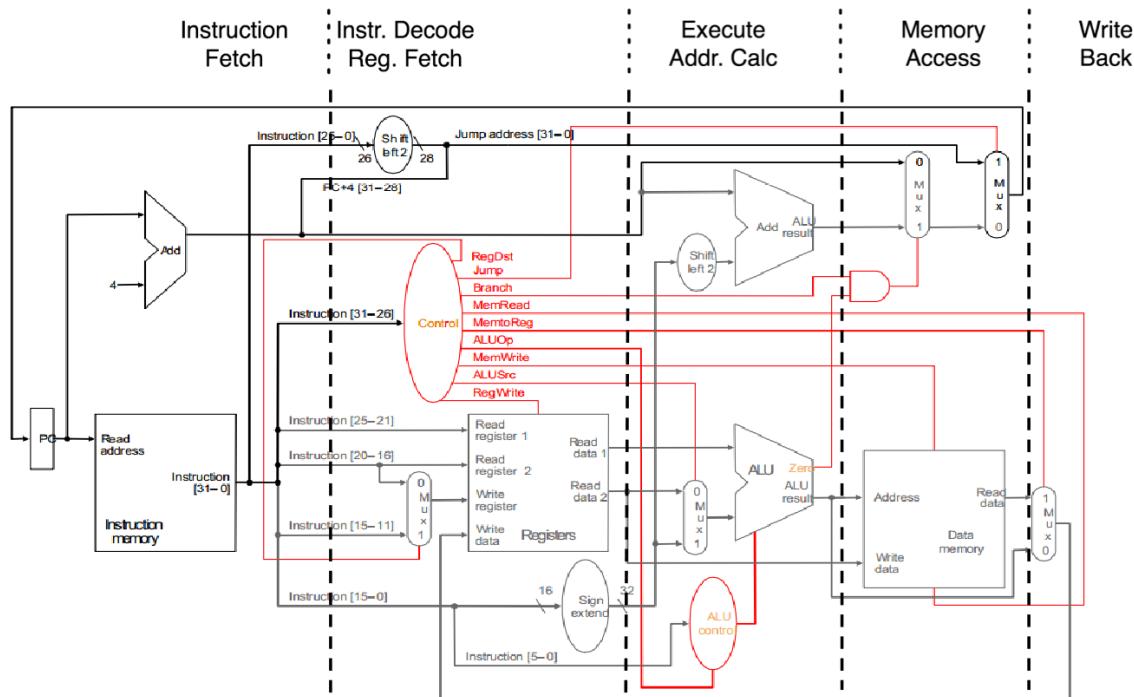


לסיכום כל שלב ומה הוא עושה, ללא מידע חדש

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	0	$IR = \text{Memory}[PC]$ $PC = PC + 4$		
Instruction decode/register fetch	1	$A = \text{Reg}[IR[25-21]]$ $B = \text{Reg}[IR[20-16]]$ $\text{ALUOut} = PC + (\text{sign-extend}(IR[15-0]) \ll 2)$		
Execution, address computation, branch/jump completion	$\text{ALUOut} = A \text{ op } B$	$\text{ALUOut} = A + \text{sign-extend}(IR[15-0])$	if ($A == B$) then $PC = \text{ALUOut}$ ($IR[25-0] \ll 2$)	8 9
Memory access or R-type completion	$\text{Reg}[IR[15-11]] = \text{ALUOut}$	Load: $\text{MDR} = \text{Memory}[\text{ALUOut}]$ or 5 Store: $\text{Memory}[\text{ALUOut}] = B$		
Memory read completion		4 Load: $\text{Reg}[IR[20-16]] = \text{MDR}$		

תרגול

נסקרו מחדש את מעבד MIPS במחזור אחד. להלן האיוור המלא של המעבד, יחד עם יחידת הבקשה (והפרדה לוגית בלבד בין שלבי ביצוע (פקודה



הערה המעבד שהגדכנו בהרצאה (זהו שרואים כאן באיוור) תומך רק בפעולות j, bne, lw, sw, add, sub, and, or, slt, beq,jal ועוד .

דגלי יחידת הבקשה

ALUop : 2 ביטים המחזיקים את קוד הפעולה שכרגע נבצע (משמשים את ה-ALU Control להחלטה לאיזה אופרטור ב-ALU נקרא).

- ביט שקובע האם במחוזר הנוכחי ניקח את המילה שנקרה מייחידת הזכרון (1) או את התוצאה (0) מה-ALU ונכתב אותה ל-Register File (או שנuttleם ממנה).
- MemRead : ביט שקובע האם במחוזר הנוכחי קוראים מהזיכרון (ונכנס לרכיב הזכרון).
- MemWrite : ביט שקובע האם כתובים לזכרון (ונכנס לרכיב הזכרון).
- Branch : האם הפוקודה היא פועלות branch (ולכן יש לשקלל קפיצה להיסט).
- ALUSrc : האם הקלט השני ל-ALU הוא רגיסטר (1) או immediate (0) Sign Extend שעשו לו.
- RegWrite : ביט שקובע האם לכתוב את המילה שנכנסת לחוט הדאטה לכתיבה ל-Register File לאחד הרגיסטרים בו (או להתעלם ממנה).
- Jump : האם הפוקודה הנוכחי היא קפיצה לא מותנת.
- RegDst : האם הרגיסטר אליו כתובים (אם כתובים) את תוצאה הפעולה הוא השני שמוגדר בפקודה (0) או השלישי (1).

ה-ALU מקבל קידוד לאופרטור אותו הוא אמור להריץ. אנחנו משתמש באופרטורים הבאים

האופרטור	קידוד האופרטור
AND	0000
OR	0001
add	0010
sub	0110
slt	0111
NOR	1100

אבל קלט האופרטור ל-ALU לא מגיע ישיר מהפקודה, אלא עובר קודם דרך ALU Control, שמקבל קלט את הדגל ALUop. הוא נקבע לפי ה-opcode ובמקרה שמדובר ב-R-Type funct, גם ה-ALUop יקבע קלט ה-ALU Control.

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

דוגמה נחשב את הדגלים בפקודת add.

- $\text{RegDst} = 1$ כי השתמש ברגיסטר השלישי בפקודה כיעד החישוב.
- $\text{Branch} = 0$ כי מדובר בפעולה אריתמטית ולא קפיצה.
- $\text{MemRead} = 0$ כי אנחנו לא מתקשרים עם הזכרון בסכימה.
- $\text{MemToReg} = 0$ כי כאמור לא נרצה השפעה של הזכרון ונרצה את תוצאת ה-ALU לרגיסטר היעד.
- $\text{ALU Control} = 10$ כי זה הקוד לפקודות מסווג R-Type והוא 100000 (לא אמורים זכור), אז ה- ALUop יחשב את הקוד שה-ALU מבין ונקבל 0010.

- $\text{MemWrite} = 0$ כי כאמור אין לנו עיסוק בזיכרון.
- $\text{ALUSrc} = 0$ כי אנחנו רוצים לחשב סכום על שני רגיסטרים ולא immediate.
- $\text{Register File Write} = 1$ כי אנחנו כן רוצים לכתוב את תוצאת החישוב חזרה ל-Register File.

דוגמה נחשב את הגדלים בפקודת `lw`.

- $\text{RegDst} = 0$ כי `lw` היא פקודה מסווג I-Type וכאן היעד הוא הרגיסטר השני שMASOPAK.
- $\text{Branch} = 0$ כי מדובר בפעולה אריתמטית ולא קפיצה.
- $\text{MemRead} = 1$ כי אנחנו רוצים לקרוא מהזיכרון ולכתוב לרגיסטר.
- $\text{MemToReg} = 1$ כי כאמור אנחנו רוצים לקרוא מהזיכרון ולהשתמש במידע הזה.
- $\text{ALUop} = 01$ כי זה הקוד לפקודות מסווג I-Type ואז ה- ALU Control יחשב את הקוד שה-ALU מבין לשכימה (לכל ה--I Type זה מה שימושים) ונקבל 0010.
- $\text{MemWrite} = 0$ כי אנחנו קוראים ולא כתבים.

- $\text{ALUSrc} = 1$ כי אנחנו רוצים לחשב את כתובות הקריאה לפי סכימה של הרגיסטר הראשון עם ה-immediate.
- $\text{Register File Write} = 1$ כי אנחנו כן רוצים לכתוב את תוצאת הקריאה מהזיכרון חזרה ל-Register File.

דוגמה נחשב את הגדלים בפקודת `beq`.

- $\text{RegDst} = 0$ כי `lw` היא פקודה מסווג I-Type וכאן היעד הוא הרגיסטר השני שMASOPAK.
- $\text{Branch} = 1$ כי מדובר בפעולות קפיצה.
- $\text{MemToReg} = 0$ לא משנה כי כל עוד $\text{RegWrite} = 0$ כל ערך שלו לא ישפיע על מצב הרגיסטרים.
- $\text{MemRead} = 1$ כי אנחנו כנ"ל.
- $\text{ALUop} = 01$ כי זה הקוד לפקודות מסווג I-Type ואז ה- ALU Control יחשב את הקוד שה-ALU מבין לשכימה (לכל ה--I Type זה מה השימושים) ונקבל 0010.
- $\text{MemWrite} = 0$ כי אנחנו לא קוראים ולא כתבים.
- $\text{ALUSrc} = 0$ כי אנחנו רוצים להשווות את הרגיסטר עם ה-immediate לחישוב התנאי.
- $\text{Register File Write} = 0$ כי אנחנו לא צריכים לכתוב שום דבר חזרה ל-Register File.

דוגמה נניח שנרצה להוסיף תומכה לפקודת addi. לצורך כך, נדרש להוסיף שערי AND מותאימים לopcode שיתאים לפקודה ביחידת הבקרה עם הדגמים המותאימים (נדליק רק את ALUop ו-RegWrite ונשתמש ב-ALUsrc שערךו 01). מעבר לכך לא נדרש להוסיף לחסוך חוטים וכו'.

דוגמה נרצה להוסיף תומכה לפקודת jal. לשם כך נדרש להוסיף opcode חדש עם חוטים חדשים ביחידת הבקרה כנ"ל, אבל הפעם נדרש לבצע שני שינויים למימוש המעבד:

הראשון הוא חיבור ה-PC (עם aux) לקלט הכתיבה ל-Register File כך שנוכל לכתוב את כתובת 4 PC+4 ל-\$ra. נוסיף בית נוסף ל-RegToMem כך שעתה 00 פירשו כתיבה מה-ALU, 01 מהזיכרון ו-10 מ-PC (+4).

השניינו השני שנוצר על מנת שהוא יוכל לרשום את הכתובת באליאו אונחנו כתובים. כרגע אפשר לכתוב רק לרגיסטר שהאינדקס שלו מופיע בפקודה (חמשת הביטים ב-R-Type ו-I-Type). עם זאת עצת נרצה במקרה שמדובר ב-opcodejal, לקובע את אינדקס הריגיסטר אליו נכתב (\$ra) בלי שיופיע בפקודה (כי הפקודה מכילה opcode והיחסט אליו קופצים בלבד), לכן נוסיף עוד בית ל-RegDst וחיווט נדרש כך ש庆幸ה 00 פירשו כתיבה ל-rd (הריגיסטר השלישי ב-R-Type), 01 ל-rt (ב-I-Type) ו-10 לריגיסטר 31 (\$ra).

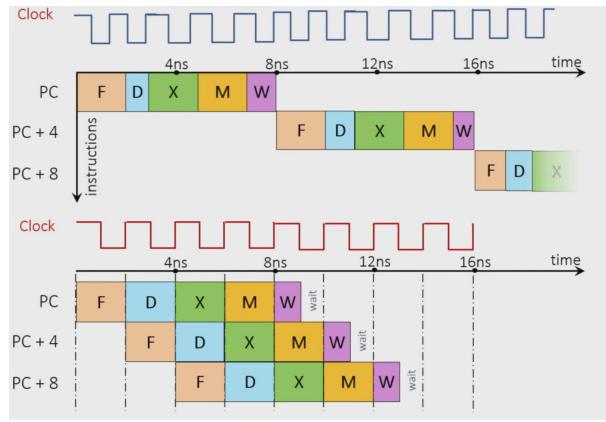
שבוע VII | מעבד pipelined

הרצאה

עד כה הצליחנו להוריד את זמן המחזורי פי 4, אבל גם העלנו את מספר הפקודות להוראה פי 4, כך שלא הרווחנו יותר מדי. לשם כך נהפוך את המעבד ל-pipelined, כלומר במקום שנריץ פקודה אחת דרך המעבד בכל רגע נתון, נריץ כמה פקודות בחלקים שונים של המעבד, ששייכות לכמה הוראות שונות, בו זמנית.

דוגמה נרצה לעשות כביסה: להכנס למכונית כביסה, ליבש, לקפול ואכسن. במקום לכל עירימת כביסה לעשות את ארבעת השלבים ואז לעבור לעירימה הבאה, אפשר אחרי סיוםו את המכונית הראשונה, להעביר את העירימה למיבש ומיד להכנס את העירימה השנייה למכונית הכביסה, וכך נוכל לחסוך הרבה זמן ריצה מצטבר (במקום 16 שלבים, רק 7).

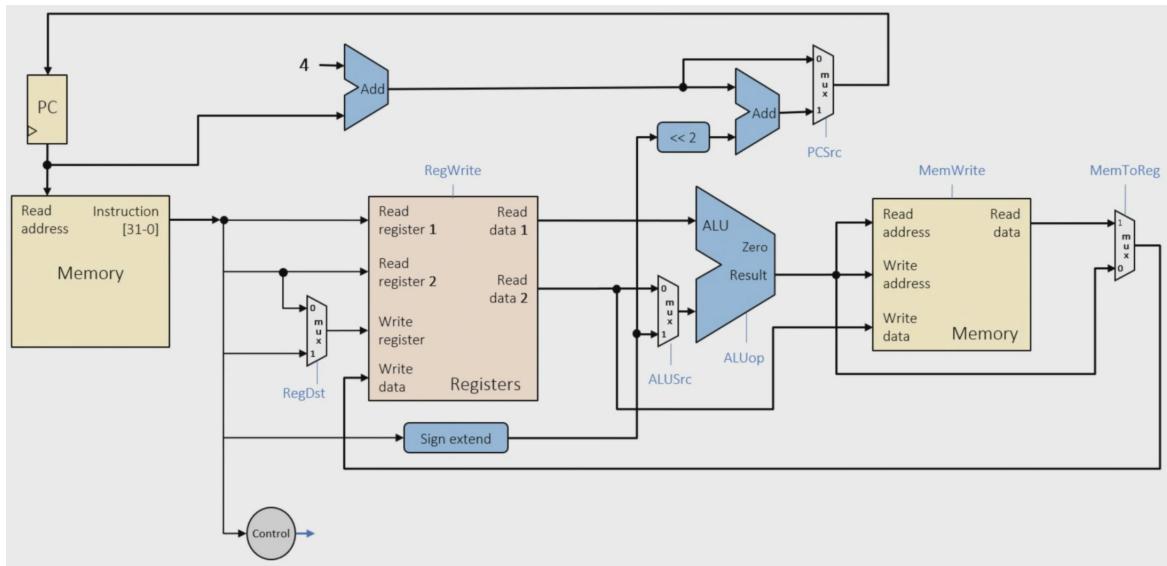
דוגמה במקרה של מעבדים, נשווה בין Multi Cycle (למעלה) ל-pipelined. ברור שעדיף Pipelined, ובפרט זמן החישוב שלו לינארי עם מקדם כמעט 1 (מגלמים את הפקודות בקצבות) ביחס למספר הפקודות, בעוד Multi-Cycle הוא לינארי עם מקדם יותר מ-4 ביחס למספר הפקודות.



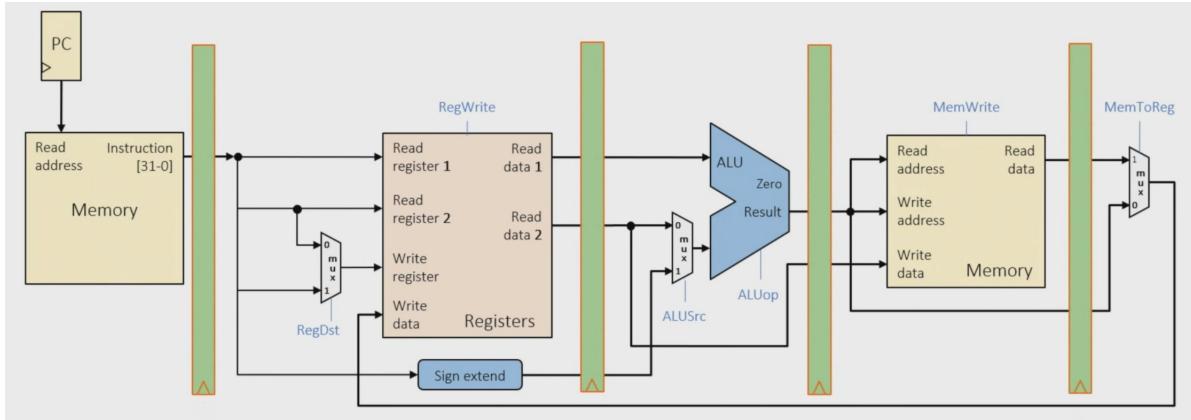
תחת הנתונים הנ"ל, ל-IPC יש $0.25 * 500 = 125 \frac{1}{2ns} = 500MHz$ ותדרות של $\frac{1}{4}$ IPC וαιילוIPC יש של 1 (בלי הקצוות) ואותה תדרות של $500MHz$ pipelined כולם speedup של 4.

הערה ה-latency throughput של כל פקודה לא השתנה (אולי נגראן, כי כל הפקודות עוברות 5 שלבים גם אם דורשות 2) אבל ה-latency המשמעותית.

מעבד ה-MIPS במחזור אחד נראה如下:



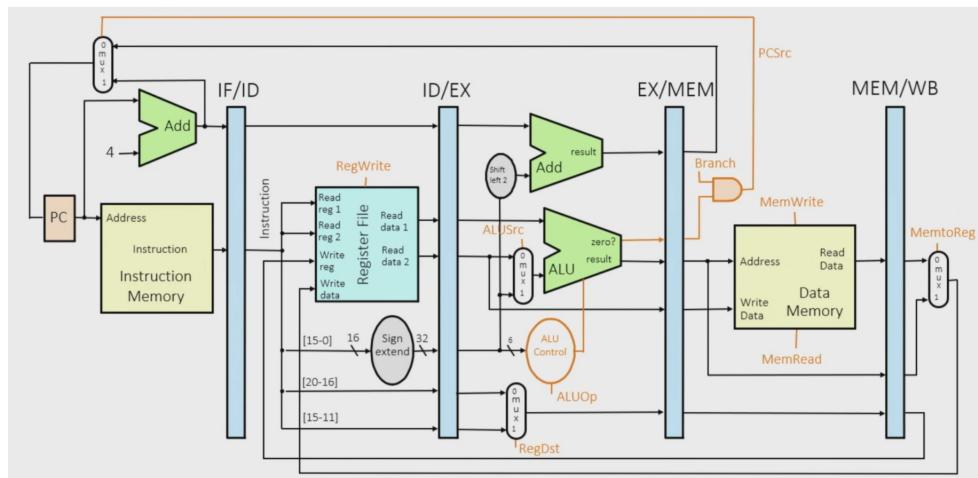
אם נתעלם ליעשו מהקונטROL ומה-branch-ים, יוכל לחלק את התהילה לשלבים השונים של ה-pipeline (הקיים הירוקים הם רגיסטרים ששומרים מצב לאחר חישוב השלב)



כלומר השלבים שלנו הם write back (5) ; memory access (4) ; ALUOp (3) ; decode (2) ; fetch (1) . נשים לב שהחלוקת הזה זהה לזו במעבד הרב-מחזורי. הסיבה לדמיון הזה הוא שאליה שלוקחים את הזמן זמן ולכן חלוקת זמן מוחזר שוויים היא יחסית הוגנת.

לרגיטורי המצב יש שמות מקובלים (משמאלי לימי) : Fetch latch (1) (רегистר ה-PC ושר המידע שנשמר שם) ; IF/ID (2) (כשם שהוא בין EX/MEM (4) ; (בהתאם) ; (5) ו- decode (3) ; (decode fetch- וה- execute-ID/EX (3) ;) .

דוגמא להלן איזור בסיסי של מעבד MIPS עם pipeline, שישמש אותנו בהדגמת הרצת הפקודה lw \$1, 30(\$2,



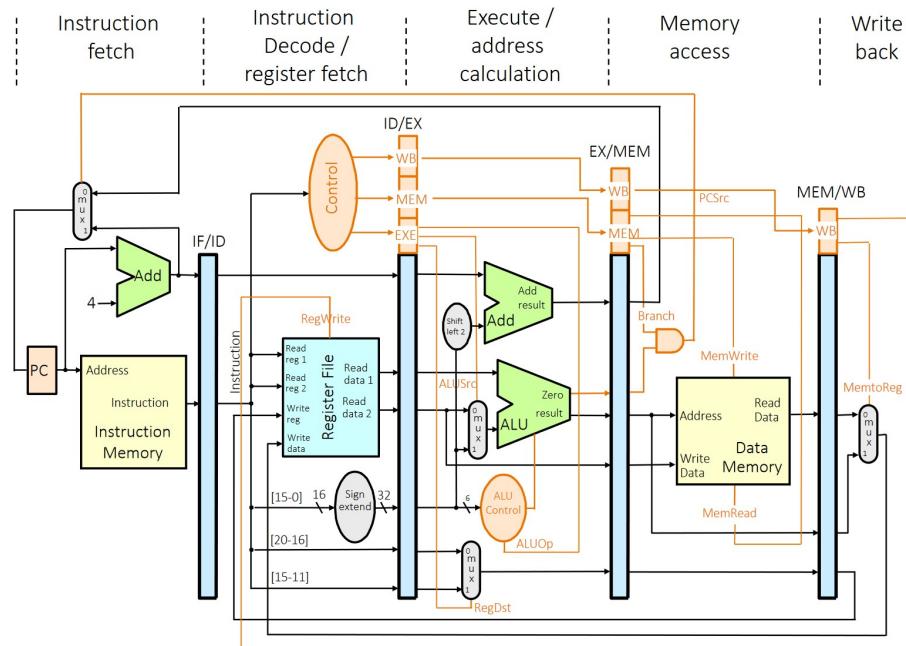
נפרט את השלבים שתעבור הפקודה במעבד :

- לאחר פקודת ה-PC+4, fetch-ID, יחזק את החוראה (כלומר מה שב吃过 נשמר ב-IR) ובנוסך את הכתובת ממנה נקראת (\$2), וזו נדרש בשביל branch-ים בהמשך.
- לאחר פקודת ה-ID/EX, decode-ID/ex, יחזק את תוכן הרגיסטר \$2 בדומה לנ"ל, את תוכן הרגיסטר PC+\$4, את ה-ALUOp ואות האינדקס של רגיסטר 1.
- לאחר פקודת ה-ID/EX, decode-ID/ex, יחזק את תוכן \$2 סכום עם 30, ואת אינדקס 1. לאחר פקודת ה-EX/MEM, execute-ID/ex, יחזק את תוכן \$2 סכום עם 30, ואת אינדקס 1.
- לאחר פקודת ה-MEM/WB, memory access, יחזק את המידע מהכתובת שביבשנו יחד עם אינדקס 1.

- במחזור השעון הבא, שני הנתונים הנ"ל יגיעו חוזרת ל-Register File ויבילו לכטיבה אליו כמצופה מהפקודה.

הערה באופן מוהטי, היחידות של מעבד pipelined זה יותר דומה למעבד חד-מחזורי מרוב-מחזורי כי ברב-מחזורי ניצלו את העבודה שיש ייחדות שלא פועלות בשלבים מסוימים כדי להשתמש בהן לצורך שלבים אחרים, אבל ב-pipelined אנחנו מרכיבים את כל השלבים כל הזמן (עבור הוראות שונות כמובן).

עתה יחד עם יחידת הבקרה, המעבד יראה כך



כאשר כמוון ה-control מעביר מידע רק אחד שלב decode-ה-ID נחזק את הקונטロלים הנדרשים לשלב הבא, שהוא execute, אבל גם לשלבים שאחרי (קרי write back ו-memory access) כי רק כך אפשר להעביר מידע מידי הלאה. בדומה EX/MEM יחזיק את הקונטロלים של השלב שלו וזה שאחריו וכו'.

בטבלה הבאה ניתן לראות את ההפרדה לשלב בו נדרש כל קונטROL שיווצר מיחידת הבקרה, יחד עם רשיימה (לא ממצה) של ערכיהם עבור פקודות

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

דוגמה נרץ כמה פקודות (6 מחזוריים סה"כ) בו זמנית pipeline ונציג כיצד המידע עוצר בין השלבים (בכל תא רשום באיזה שלב אנחנו נמצא, ומה שמՐנו מהשלב הקודם)

מחוזר שעון/פקודה	t	$t + 1$	$t + 2$	$t + 3$	$t + 4$	$t + 5$
0: lw \$10, 9(\$1)	IF (0)	ID (4, lw ...)	EX (4, [\$1], 9, \$10)	MEM ([\\$1] + 9, \$10)	WB (Mem[[\\$1]+9], \$10)	
4: sub \$11, \$2, \$3		IF (4)	ID (8, sub ...)	EX (8, [\$3], [\$4], \$11)	MEM ([\\$3]-[\$2], \$11)	WB ([\\$3]-[\$2], \$11)
8: and \$12, \$4, \$5			IF (8)	ID (12, and ...)	EX ([\\$4], [\$5], \$12)	MEM ([\\$4]&[\$5], \$12)
12: or \$13, \$6, \$7				IF (12)	ID (16, or ...)	EX ([\\$6] [\$7], \$13)

סכנות במעבד pipelined

נשים לב שלא נוכל למקבל את כל הפעולות, משום שנוכל להיתקל ב-Pipeline Hazards, ככלומר, הוראות שלא נוכל לבצע בזמן המוחזר המקורי שהוקצתה להן. ישנו שלושה סוגים hazards:

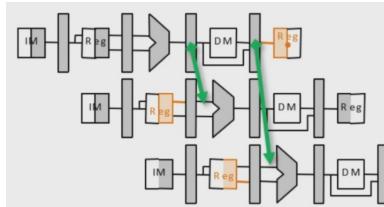
- **סכנות מבניות:** החומרה לא יכולה לתמוך בשילוב פקודות (לדוגמה נctruck זכרו שונה ל-PC והדאטא כי אחרת ניתקע במצב ש策יך לכתוב בו זמן לתשתיות כתובות מאותו זיכרו).
- **סכנות דאטה:** הוראות שמתבססות על פקודות שעוד לא חושבו pipeline (לדוגמה שימוש ברגיסטר שתוכנו אמרור היה להתמלא כבר בתוצאתה של פקודת סכימה שעוד לא הסתיימה).
- **סכנות בקרה:** צריך לבצע קפיצה אבל תנאי הקפיצה עוד לא חשוב (לדוגמה בקפיצה מותנת שווינו בין רגיסטרים שתוכנם עתה מחושב בשלבים מאוחרים יותר של ה-pipeline).

פתרונות לסכנות

- **קריאה וכתיבה במקביל ב-RF** (סכנה מבנית): אם אנחנו עושים קוראים רגיסטר שאחנו גם כותבים לו (שווינו אינדקסים) באותו זמן מחזיר כךשה-DFF-ים יעדכו את יציאותיהם רק במחזר הבא, ניקח את הביטים מוחוט הכתיבה במקומות מפלט ה-DFF-ים. אנחנו מבצעים bypass לרגיסטרים.
- אפשר לפטור זאת באופן דומה באמצעות כתיבה בירידת שעון וקריאה בכתיבת שעון (בהנחה שהתזומנים מסתדרים כמו שלמדנו) ואז ה-DFF יציג תוכן כבר מאמצע מחזיר השעון (נספר לפי עליות) והקריאה תעשה רק בסוף המוחזר הנוכחי, ככלומר העלייה הבאה, ותוכנה כבר יהיה תקין.
- **קריאה אחרי כתיבה שטרם הסתיימה** (סכנה דאטה): הבעייה היא שאופרנד של הוראה יכול להיות התוצאה של הוראה קודמת שעוד לא הגיע WB.

פתרון אפשרי לבועיה זו הוא החוספת NOP-ים בשלב הקימפול בין פקודות שצרכו הפרש מחזירים ביןיהם כדי שההוראה הקודמת תסתיים לפני שהבאמה מתחילה, וכך ניצור את המרווח הדרוש. הפתרון הזה די בזמני (30% הגרעה בבייצועים). פתרו מקביל לכך הוא stall, ככלומר שלא נוסיף במפורש פקודות NOP, אלא שהחומרה תזהה שיש צורך ביצירת מרווה ותמנע התחלת של ה-pipeline להוראה הבאה עד שהתוכן יכתב לרגיסטר.

פתרון עדיף נקרא Forwarding (עוד סוג של bypass) כলומר שההוראה הבאה תקרא ישירות מה-ALU את תוצאה החישוב ולא תחכה לשני השלבים הנוספים של ההוראה הקודמת (MA ו-WB). ראו איור המדגים זאת (החציים הירוקים הם bypass, שמעביר את תוצאה ה-ALU למקומות הנדרשים בשלב ה-decode)



הימוש בפועל הוא די פשוט : ה-*control*, stateful, יזכור את שתי הפקודות הקודמות ולאן הן כתובות, וכך ידע האם יש צורך ב-forwarding. אם יש צורך, שלב ה-execute יקח (ה指挥ה באמצעות אונט) ערך של אחד או שניים מהגיסטרים מפלט ה-EX/MEM. האחרונה תפלוט את הערך הרלוונטי לפי קלטים שהוא מקבלת מתוצאת החישוב והביט RegWrite unit ב-Forwarding unit (הפקודה הקודמת) ותוצאת החישוב והביט RegWrite ב-MEM/WB (הפקודה שלפני הפקודה הקודמת).

ברמת הלוגיקה, אם $\text{RegWrite} = 1$ וגם אינדקס הרגיסטר אליו כתובים בפקודה קודמה זהה לאינדקס רגיסטר שקוראים ממנו, נבחר את תוצאה ה-ALU על פני ה-RF, עם עדיפות לפקודות חדשות יותר (קודם מסתכלים על EX/MEM ורak אז על WB/MEM).

דוגמה נציגים מקרים שבו נדרש לבחור בין פקודות חדשות לשונות

add \$2, \$1, \$3

add \$2, \$2, \$4

add \$2, \$2, \$5

כאן בעית חישוב הפקודה השלישית, נרצה את התוצאה הטרייה יותר (הסכמה השנייה), שכבר לא רלוונטית. אחרת היינו במצב של Out of Order Execution, שקרה רק ב-Write After Write (Write After Write After Write). הבאים.

• העתיקת זכרון-לזכרון (סכתן DATA) : הבעיה עולה כxebacesim Iw לכתובת בזיכרונו שזה עתה כתבנו אליה עם sw.

אפשר לפטור את הבעיה עם forwarding שלוקח את המילה שנכתבה בפקודה הקודמת (שנשמרת ב-WB/MEM) ומשתמש בה לצורך קלט של הקריאה מה-.data memory stall יקר.

• load-to-use (סכתן DATA) : הבעיה עולה כxebacesim Iw לכתובת שמתבססת על תוצאה קריאה מ-Iw בפקודה הקודמת.

את הבעיה זו נפטרו עם stall (או NOP), כלומר נבדוק לפני ה-execute האם אינדקס של רגיסטר שאנו חנו צריכים בפקודה הבאה (ידע שיש לנו M-ID/IF) וזה לאינדקס היעד של הפקודה הנוכחית, שהוא Iw (ידע שיש לנו M-EX/ID). אם כן, מעכבות את ה-pipeline.

הערה את כל הסכנות נצטרך לוחות כבר בשלב ה-decode והוא השלב המוקדם ביותר שניתן לעשות בו את זה, כי רק אז אנחנו יודעים בכלל מה הפקודה עשויה והאם נדרשים אמצעים מיוחדים כדי להבטיח נכונות. את הלוגיקה של בחירת האמצעי הנדרש למניעת סכנות Hazard Detection Unit (bypass, stall) מושם בתוך ה-unit.

הערה פתרון אלטרנטיבי לפני מנגנוני מניעת הסכנות היא כתיבת קוד יותר חכם ברמת הקומפיילר/בן אדם שכותב אסמבלי, כך שנrichik פקודות מסווגות אחת מהשניה ונשים בכךן פקודות אחרות לצריכות לkerות שלא דורשות תלות בעיתית שגורמת לטכנה.

הערה השלוטים של הוראות אחרי הוראות load delay slot שבהם צריכים למנוע התנגשות עם פקודה עתידית נקראים load delay slot והקומפיילר אידיאלית ישם בהם כאמור פקודות אחרות שלא מתנגשות. אפשר לשנות את דרך הפקודות כל עוד שומרים על תקינות התוכנה ביחס להוראות המקורית.

מניעת סכנות ב-branching

השלב הכי מוקדם שבוណו לנו הולכים לקפוץ הוא בסוף ה-ALU (כשמחשבים את התנאי לקפוצה מותנה). הבעה היא שעד שפקודת ה-branch הגיעו לשם לא נדע האם علينا להריץ את הפקודות הבאות בזיכרון הכתובות או לקפוץ לכתובת אחרת ולהריץ ממש. כך שהסכנה היא שאם נרוץ כרגע, חלק מהפקודות ב-pipeline לא אמורים בכלל לא לרוץ כי הינו אמורים לקפוץ. נציג כמה פתרונות לבעה זו.

הערה forwarding לא עובד כי צריך לקרוא בכל פקודות אחרות במקרה של קפיצה. ככלומר אם קופצים, שלושת הפקודות הבאות מותבלות, אפ"פ שהן התחילו את מהלכן ב-pipeline. הכוונה בביטול היא לא בהכרח שלא מחשבים משחו ב-ALU אלא שלא כתבים שום דבר חוזרת (לא ל-DM, לא ל-PC ולא ל-RF).

1. stall עד שמכריעים: נכח אחריו כל branch שלושה מוחזרים ורק אז נמשיך את ה-finish של הפקודה הבאה. זה ייתן לנו החמורה של פי 3 זמן ביצוע (כל פקודה לוקחת זמן מוחזר 1 בתעלם מהקצotta) וזה קורה ל-20% מה-branch-ים, ככלומר פי 1.6 מוחזרים פר-הוראה, שזה פי $\frac{1}{1.6} = 0.63$ הוראות פר-מחזור ככלומר האטה של 37%.

ונוכל לשפר את החמורה בביוצעים קצר באמצעות branch-decode כבר בשלב ה-decode (באמצעות משווה מיוחד שייחסב האם קופצים). כך רק פקודה אחת דינה להתבטל (או שבנתאים מוחזקת ב-ID/IF). ברגע שאנחנו מאתרים branch, נבטל את הפקודה הקודמת באמצעות ביצוע flush ל-ID/IF, ככלומר נחליף את תוכנו בתוכן שמתאים לפקודת NOP כך שהפקודות שביטלו לא תבצעו.

2. תמיד לעשותcaiilo לא kpczno: נמשיך את הפקודותcaiilo הקפיצה לא קורתה, ואם היא קורתה אז נעשה לפקודות שלא היו אמורים לרוץ flush וניתך את הפקודות שכן kpczno. זה קצר משפר את המצב כי אם נניח ש-50% מה-branch-ים קופצים, פתאום יש CPI גבוה ב-1.3 ככלומר פי 0.77.

3. delayed branches: נשנה את ה-ISA כך שכל תוכנה, גם אם יש קפיצה, תזכה לפחות כמה פקודות לפני הקפיצה בכל מקרה, וכך יהיה לנו זמן לחשב את הקפיצה ולא תהיה ההתנגשות הזו. ככלומר אנחנו מנסים את החוזה בין המפתח למעבד.

המשמעות של זה בפועל כמוון לא כולל את המפתח, אלא רק את הקומפיילר; בהנחה שנמדד את הקפיצה כפקודות לפני, שני מסלולים אפשריים ואז התוכנסות למסלול פקודות אחרי הפיצול, הקומפיילר יהיה זה שיסופף *a* פקודות אחריו branch-ה-branch, כך שהסמנטיקה (מה שהקוד עושים) לא תשתנה. פקודות אלו יכולות להגיד לפני הקפיצה (כאלו שלא משפיעות על התנייניות הקפיצה) או מסלול התוכנסות לאחר הפיצול. אם כל הפקודות תלויות, אפשר להוציא *copy*-ים במרקחה הכיר�ע.

בפועל פקודה אחת שלא קשורה לקפיצה די קל למצוא, שתי פקודות לעתים אפשר ושלוש כבר כמעט בלתי אפשרי (אחרת התוכנה הייתה עשויה כל מיני דברים מיוחדים כנראה).

.4 ננסה לחזות האם נקופץ הפעם או לא: כבר בשלב ה-*fetch* של הקפיצה, נשנה את ה-PC לפי החיזוי שלנו. בשלב ה-*execute* אם אכן branch התקיים נוסיף אותו ל-*Branch Target Buffer*, שהוא סוג של מטמון שזוכר את ה-*branch*-ים האחרונים שקרו. פעם הבאה שנגיעה אליו ה-*branch* (מאונדקס לפי PC היעד של הקפיצה), נענה כמו שענינו פעם קודמת. אם טעינו, נעשה flush קריגיל.

דוגמה בולולאות של 1,000,000 פעמים, נתעה לכל היותר פעמיים, ככלומר ה-CPI לא הושפע!

מושווה את הביצועים של המעבד החדש-מחזורי, הרב-מחזורי, וה-*branch*-*pipelined* (*branch*-*pipelined* אנקחו מניחים ש-30% מפקודות ה-*lw* דורשות *lw*-*stall* בغالל סכנת קראיה-שימוש ו-90% מה-*branch*-*stall*, שהוא באורך 2 סלוטים לצורך החישוב)

פקודה	תדירות הפקודה	CPI			<i>pipelined</i>
		חד-מחזורי	רב-מחזורי	pipelined	
R-Type	50%	1	4	1	
lw	30%	1	5	$1 + 30\% * 1$	
sw	10%	1	4	1	
branch/jump	10%	1	3	$1 + 2 * 90\%$	
CPI ממוצע		1	4.2	1.279	
זמן מחזור		(baseline) 1	$\times 0.25$	$\times 0.25$	

nicer שהמעבד ה-*pipelined* הוא כבר פי 3 יותר מהחיד-מחזורי.

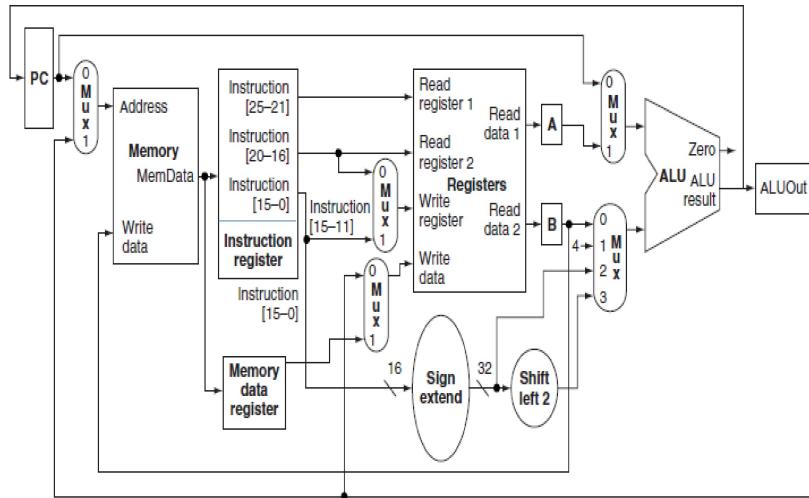
תרגול

שינויים מרכזיים ברכיבים תחת מעבד רב-מחזורי

- הזכירו: נצרך לזכור האם המילה שקרהנו מוחזרה היא הפקודה הבאה אותה נרץ או מידע לרוגיסטר.
- ALU - כל החישובים בשלבים השונים מאוחדים אל תוך ALU יחיד ולכל השינויים הבאים יקרו:
 - נצרך לזכור האם תוצאהו היא קידום ה-PC ב-4 או חישוב בשלב ה-*branch* (ובמקרה של *branch* בזמנים שונים ישמש בשני התפקידים האלה).
 - נצרך לזכור האם האופrndים המוכנסים ל-ALU מגיעים מרוגיסטרים או מ-immediate-ים.
- שמירת נתונים בין-שלביות: בזע כל שני שלבים נוספים יוסיפו רוגיסטרים שיזכרו נתונים רלוונטיים מהשלב הקודם.
- שמירת מצב ביחידת הבקרה: נוסף DFF-ים בתוך יחידת הבקרה שישמרו באיזה מצב אנחנו כרגע.

אותות בקרה למרבבים במעבד רב-מחזורי

ນביט באירור הבא של מעבד רב-מחזורי ללא רישום מלא של חיוטי הבקרה

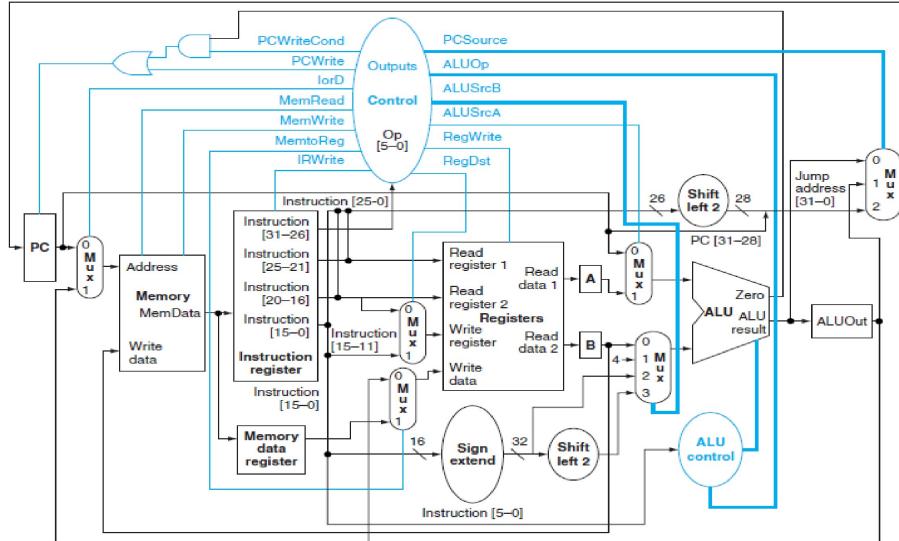


מעבר על כל ה-*aux*-ים וסבירו איזה בית מחובר אליהם ומה תפקידם, משמאלי לימין (ושובר שווין לפי למעלה-למטה) :

1. מחובר ל-*IorD*, ופירשו האם יש לקרוא מהזכרונו פקודת במאזעות כתובות מה-PC (0), או דатаה, באמצעות כתובות שחושבה ב-ALU.
.(1)
2. מחובר ל-*RegDst*, ופירשו האם רגיסטר היעד הוא השני (0) או השלישי (1) - רלוונטי להבדל בין פקודות מסוג R-Type וכל השאר.
3. מחובר ל-*MemToReg*, ופירשו האם רגיסטר היעד יש לכתוב את תוצאה ה-ALU (0) או מילא שזה עתה נקרא מהזכרונו ל-*RF* (1) - רלוונטי לפקודת *lw* לעומת כל השאר (בחלקו לא נכתב כלום ל-*RF* ואז בית זה אינו רלוונטי).
4. מחובר ל-*ALUSrcA*, ופירשו האם יש להשתמש ב-PC כקלט הראשון ל-ALU (0) או בתוכנים של רגיסטרים מה-*RF* (1) - רלוונטי לפקודות שמחשובות היסט מה-PC הנוכחי (קפיצות מותנות ולא מותנות).
5. מחובר ל-*ALUSrcB*, ופירשו האם יש להשתמש ברגיסטר שנקרא מה-*RF* כקלט השני ל-ALU (00), ב-4 (01), ב-4*imm (10) או ב-4*imm (11) - רלוונטי לסוגי הפקודות השונות : PC ב-4, beq ו-R-Type, קידום PC ב-4, fetch, פקודות I-Type-arithmatic ו-sw/lw, וחישוב כתובות קפיצה, בהतאמה.

אותות בקרה חדשים במעבד רב-מחזורי

להלן המעבד הרב-מחזורי המלא של MIPS, כולל ייחידת הבקרה. עתה נפרט את שאר החוטים שטרם הזכירנו שנוסףו לנו במעבד הרב-מחזורי



.1. קובע האם נכתוב ל-PC (אם נכתבו את פלט ה-ALU במקורה של קידום ב-4 (00), פלט ה-ALU מהמחזור הקודם במקרה של PCWriteCond).

.2. קפיצה מותנת שבה ההתנייה מתקימת (01), או את הכתובת שהישבנו במקרה של פקודת j (jal).

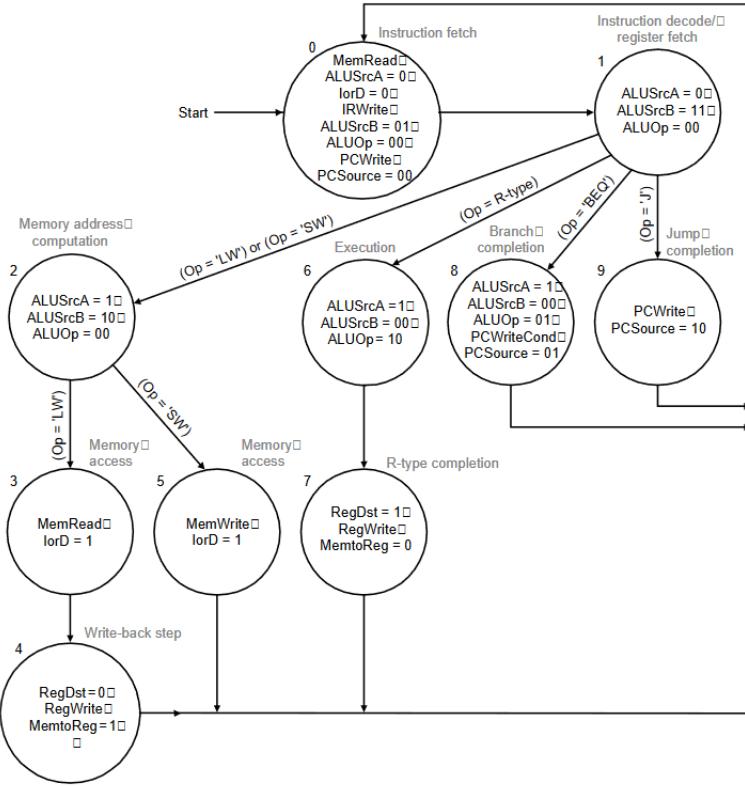
.3. האם לכתוב ל-IR את פלט הקריאה מהזיכרון (דлок בשלב ה-decode).

.4. תנאי מספיק לכתיבה ל-PC (דлок בשלב ה-fetch, או בעת קפיצה בלווית מותנת).

.5. האם לאפשר כתיבה ל-PC בהנחה שדגל zero של ה-ALU (דлок אם "מ" מדבר בפקודת branch, או נקפו).

דוגמה נוסיף תמייה ל-addi. הוספה די דומה למקרה החד-מחזרי. עדיף לשנות כמה שיטות רק את ה-Controller, ולא דברים אחרים.

נשים לב שהפעם אנחנו משנים את יחידת הבקרה פר-מכב. נביט במכונת המცבים הנוכחית של הקונטROLLER



נוסיף מצב חדש 10, שמගיעים אליו מ-2 ויוצאים ממנו ל-0 שישנה את הדגלים כך
שנסכום בין רегистר ל-immediate ולא נכתב לזכור (אבל כן ל-RF).

הערכת ביצועים

טענה (חוק אמדל) עבור Fraction_{enhanced}, Speedup_{enhanced} נקבל שיפור כללי ביצועים של

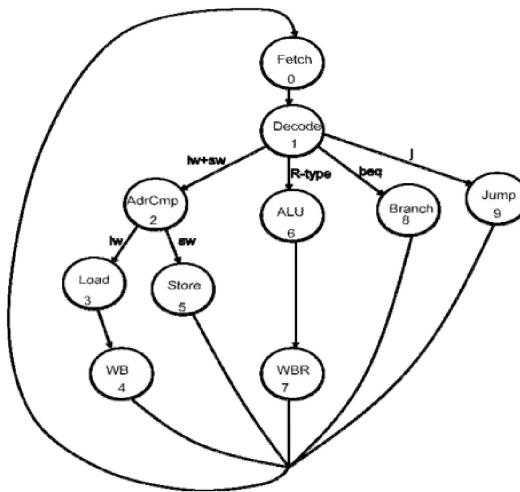
$$\text{Speedup}_{\text{overall}} = \frac{1}{1 - \text{Fraction}_{\text{enhanced}} + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

דוגמה הצלחנו לשפר פקודות R-Type פי 2, כאשר אלו הן רק 10% מכל התוכנית, אך השיפור הכללי הוא $\frac{1}{1 - \frac{1}{10} + \frac{1}{20}} = \frac{1}{\frac{19}{20}} = \frac{20}{19}$.

דוגמה שינו מעבד כך שעטה פקודות בנקודה צפה ירצו פי 2.5 יותר מה, גישה לזכור פי 3 יותר מה ופעולות חיבור היוצרים בשלמים פי 1.5 יותר לפחות. זה בשימוש 15%, 20% ו-40% מהתוכנית בהתאם. נחשב את השיפור הכללי

$$\frac{1}{1 - (0.15 + 0.2 + 0.4) + \frac{0.15}{2.5} + \frac{0.2}{3} + \frac{0.4}{\frac{1}{3}}} \approx 1.024$$

כלומר חוק אמדל עבר הכללה טרויאלית למספר שינויים.



וכן שהוא מבצע 25% פעולות קרייה מהזיכרון, 8% כתיבה לזכרון, 20% קפיצות מסווגים שונים והשאר R-Type. מספרים את פקודת ה-ALU במחזור שערן אחד, מה יהיה ה-speedup של המעבד?

ההפרש היחיד בין המעבדים הוא ב-CPI ולכן היחס בין הישן לחידש הוא speedup. הפקודה היחידה שמושפעת מהשינוי היא Type, כלומר נקלט speedup של

$$\frac{5 * 0.25 + 4 * 0.08 + 3 * 0.2 + 4 * (1 - 0.25 - 0.08 - 0.2)}{5 * 0.25 + 4 * 0.08 + 3 * 0.2 + 3 * (1 - 0.25 - 0.08 - 0.2)} \approx 1.13$$

כלומר שיפור בביצועים של 13%.

שבוע VIII | המטמון

הרצאה

קצב המעבד משתפר פי 2 כל שנתיים (חוק מור), לעומת זאת הזיכרון שניהה פי 2 יותר מהיר כל 10 שנים - הפער רק גדול יותר ויותר. לכן נדרש דרך לגשת לתאים בזיכרון (לשמור דברים שאין מקום ברגיסטרים) שהוא מהיר יותר מהזיכרון הכללי - הלא הוא המטמון! במעבדים מודרניים, המטמוןściי הינו מהיר (וקטן) הוא L1 (יש L1 ID ו-L1 לדאטה פקודות בהתאם), לאחריו L2 ולאחריו L3, ולאחריו הזיכרון הרגיל. ככל שיורדים בהיררכיית הזיכרון, נדרשים פחות טרנזיסטורים לביט למימוש הזיכרון, אבל גם הגרנולריות בה מאפשר לגשת לזכרו יורדת (רגיסטר אפשר פר-ביט, ב-DRAM כבר צרייך פר-שורה), וכך גם עולה זמן הקריאה/כתיבה.

הערה למה שלא נעשה פשוט המון זיכרון מאוד מהיר? כי ככל שהזיכרון יותר מהיר הוא דורש יותר טרנזיסטורים, ולכן במערכות גדולים שלו החוט שקורא/כותב נהייה צואර הבקבוק ומונע האצה בביצועים. مكان המנטרה: זיכרון מהיר לא יכול להיות גדול וזיכרון גדול לא יכול להיות מהיר.

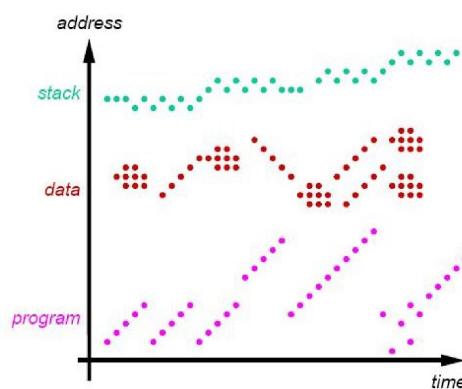
זכרו מטמון מונע שני עקרונות לוקליות: לוקליות טופורלית (אחרי שניגשנו לבית כלשהו, סביר שניגש אליו שוב ושוב, לדוגמה משתנים בפ') ולוקליות מרחבית (אם ניגשתי לבית כלשהו, סביר שאני אגש לבית מימינו, לדוגמה בערכיהם).

דוגמה נניח שיש לנו פ' שסוכמת אל תוך משתנה של הפ' את ערכיו (הסדרתיים) של מערך בלולאה. שני עקרונות הלוקליות יופיעו כאן,

פעמיים :

- **локליות בזמן :**
- בדואו: אנחנו צריכים את ערך משתנה הסכום שוב ושוב כדי לחשב את הסכימה עם כל ערך נוסף של המערך.
- בפקודות: אנחנו ניגשים לאותן פקודות בתוך הלולאה שוב ושוב.
- **локליות במרחב :**
- בדואו: אנחנו צריכים ערכים סמוכים של המערך כדי לחשב את הסכימה כל פעם - זו לוקליות מרחבית.
- בפקודות: אנחנו כל פעם לוקחים את הפקודה הבאה בתור (חוץ מבקפיות).

דוגמה להלן דigramת הלוקליות של תוכנה מסוימת



הורד מייצג הריצה סדרתית של פקודות, עם קפיצות מדי פעם (לולאות וכו'), האדום מייצג לוקליות במרחב של קריאת נתונים מהזיכרון, והירוק מייצג לוקליות במרחב במחסנית, שניגשת לערכים סמוכים אחד לשני בזיכרון (מכניסה ערך, מוציאה ערך וחוזר חלילה).

טרמינולוגיה מטמון

- **hit (פגיעה)**: חיפוש מוצלח של נתון במטמון.
- **miss (פיסוף)**: חיפוש שלא נמצא את הנתון הנדרש במטמון.
- **בלוק**: יחידת המידה הבסיסית שנuttleת למטמון לאחר פספוס, הגודל המינימלי של בלוק הוא בית אחד.
- **miss penalty (עונש פספוס)**: כמה זמן לוקח להביא את הבלוק הנדרש מהשלב הבא בהיררכיית הזיכרון (מתחת) ולהחליפ בבלוק במטמון בו.

הרענון הכללי הוא להחזיק חלק קטן מהזיכרון, ולצורך כך צריך למפות חלקים בזיכרון אל תוך המטמון. הזיכרון הרגיל מוחולק (וירטואלית) לשורות (בלוקים) בגודל טיפוסי בין 64 ל-128 בתים. המטמון יחזק נתונים בשורות, וכך נשתמש בכתבוב השורה כמפתח במטמון (נענה על שאלות מהצורה "האם יש לך את שורה X"). בעת ביצוע שאלתה למטמון, או ש:

- נקלט hit, cache hit, כלומר שהשורה אכן נמצאת במטמון ונוכל במהירות הרבה לתת אותה למעבד.
- נקלט miss, cache miss, כלומר לא נמצאת במטמון, ולכן נדרש להביא מהזיכרון השורה ולחזור אותה במטמון, דבר שייקח זמן רב. בנוסף נדרש לערוך evict (לגרש) שורה אחרת כדי לפנות מקום לשורה החדשה.

הערה אופן הגירוש נעשה באמצעות היררכיות בחון נסוק בהמשך.

Fully Associative

מטמון מסווג זה יכול למפות כל שורה בזיכרון לכל כניסה במטמון. הכתובת המסופק בשאלתה למטמון מוחולקת (מגבוה לנמוך) למספר השורה, התג (26 בתים), וההיסטוריה (6 בתים). לכל כניסה במטמון יש תג ובית וולדיות.

בעת מענה לשאלתה, נשווה את כל התגים לכל הכתובות הוולדיות (בבית וולדיות דלוק) בו זמנית, ואם תהיה התאמה נענה עם הדאטה בהיסטוריה המתאים בשורה.

דוגמה נביט בדאטה הבא בזיכרונו (מסודר לפי שורות בגודל 16 בתים), עם דפוס הגישה כפי שמופיע למטה (לפי הסדר)

00810000:	41 6c 69 63 65 20 77 61 73 20 62 65 67 69 6e 6e Alice was beginn
00810010:	69 6e 67 20 74 6f 20 67 65 74 20 76 65 72 79 20 ing to get very
00810020:	74 69 72 65 64 20 6f 66 20 73 69 74 69 6e 67 tired of sitting
00810030:	20 62 79 20 68 65 72 20 73 69 73 74 65 72 20 6f by her sister o
00810040:	6e 20 74 68 65 20 62 61 6e 6b 2c 20 61 6e 64 20 n the bank, and
00810050:	6f 66 20 68 61 76 69 6e 67 20 6e 6f 74 68 69 6e of having nothin
00810060:	67 20 74 6f 20 64 6f 3a 20 6f 6e 63 65 20 6f 72 g to do: once or
00810070:	20 74 77 69 63 65 20 73 68 65 20 68 61 64 20 70 twice she had p
00810080:	65 65 70 65 64 20 69 6e 74 6f 20 74 68 65 20 62 eeped into the b
00810090:	6f 6e 6b 20 68 65 72 20 73 69 73 65 72 20 77 ook her sister w
008100A0:	61 73 20 72 65 61 64 69 6e 67 2c 20 62 75 74 20 as reading, but
008100B0:	69 74 20 68 61 64 20 6e 6f 20 70 69 63 74 75 72 it had no pictur
008100C0:	65 73 20 6f 72 20 63 6f 6e 76 65 72 73 61 74 69 es or conversati
008100D0:	6f 6e 73 20 69 6e 20 69 74 2c 20 27 61 6e 64 20 ons in it, 'and
008100E0:	77 68 61 74 20 69 73 20 74 68 65 20 75 73 65 20 what is the use
008100F0:	6f 66 20 61 20 62 6f 6b 2c 27 20 74 68 6f 75 of a book,' thou
00810100:	67 68 74 20 41 6c 69 63 65 20 27 77 69 74 68 6f ght Alice 'witho
00810110:	75 74 20 70 69 63 74 75 72 65 73 20 6f 72 20 63 ut pictures or c
00810120:	6f 6e 76 65 72 73 61 74 69 6f 6e 3f 27 00 00 00 onversation?'

נניח שיש לנו 64 בתים במטמון FA, כלומר 4 שורות של 16 בתים כל אחת, ושאנחנו משתמשים במידיניות גירוש של LRU (קרי נגרש את השורה שבה השתמשנו לפני יותר זמן מכל השאר). בנוסף נניח שאנחנו עושים וא אחד אחראי השני לכתובות לפי דפוס הגישה הבא

- | |
|---------------|
| 1) 0x00810000 |
| 2) 0x00810040 |
| 3) 0x00810010 |
| 4) 0x00810084 |
| 5) 0x00810048 |
| 6) 0x00810000 |
| 7) 0x00810030 |

נשים לב קודם כל כי יש לנו 5 פיספוסים הכרחיים (compulsory misses), כי הנ吐נים שלנו מתרפרפים על פני חמש שורות ולבן בשיניגש לפחות בית אחד מכל אחת מהשורות הפעם הראשונה תמיד תהיה miss כי לא ראיינו את השורה לפני.

- בקריאה הראשונה נקלט miss, שכן נקרא את השורה הראשונה אל תוך המטמון ונחזיר למאבד.
- לאחר מכן בקריאה השנייה, נקלט גם פספוס, שכן נקרא את השורה החמישית ונחזיר למאבד.
- גם בקריאה השלישית והרביעית נקלט פספוסים. הקריאה החמישית היא מהשורה החמישית, שאוותה יש לנו כבר במטמון שכן נוכל להחזיר אותה, זה hit!
- הקריאה הששית היא לשורה הראשונה שעדיין במטמון, שכן זה עוד hit.
- הקריאה השביעית היא לשורה הרביעית, שעוד לא ראיינו, וכן נctrיך להכניס אותה למטמון על חשבון שורה אחרת. השורה אותה נגרש היא העתיקה ביותר שהשתמשנו בה, שהוא הבלוק (שורה) השני.
- סה"כ היו לנו חמישה פיספוסים, שתי פגיעות ועוד פיספוס.

החסרון ב-FA הוא שבגלל שיש לנו בלוק בסדרי גודל של 2^6 בתים וגודל מטמון של כ-¹⁵, נקלט שהחיפוש הופך למרכיב דומיננטי בתזמון והוא גורם להחמרה בביטויים (אפילו אם ממשים את חישוב ה- $hit/miss$ עצום OR-ים, עדיין מדובר בערך באותה חזקה).

מטמון Direct Mapped

נרצה לחסוך את עצם ה-OR הארוך שמחפש את הכתובת. עתה נסתכל על כתובות שאליה באוף הבא : תג השורה (26 ביטים), אינדקס/סט (2 ביטים) והיסט (4 ביטים).

הערה המספרים הם להדגמה בלבד, אבל היחסים ביניהם נשמרים לרוב (כלומר התג ארוך ביחס להיסט והסט).

נשמר שני מערכים במטמון :

- מערך הדטה : מערך עם כניסה כמספר הסטים (2^2 במקרה שלנו), שבו כל כניסה יש לו את השורה האחורונה שנשמרה שיש לה אינדקס כאינדקס השורה.
- מערך התגים : מערך עם כניסה כמספר הסטים, שבו כל כניסה יש את התג המתאים לשורה השמורה באינדקס הכניסה, ובית וולידיות.

דוגמא אם קיבלנו פספוס על הכתובת 10 0010 10 00 . . . 00, נקרא את השורה 00010 . . . 00 ונכתוב אותה לכינסה ה-10-ית של מערך הדטה. באותו הזמן נכתב לכינסה ה-10-ית של מערך התגים את התג של השורה הזו, 00 . . . 00.

הערה עתה נגרש שורות לא בגלל שהטבלה כוללה מלאה, אלא רק אם בכניסה המתאימה לטט שלנו יש כרגע ערך (ולידי) אחר. **דוגמא** נניח שאנו רוצים לקרוא באותו דפוס גישה כבוגמה על FA. הסטודנטית המשקיפה תפרק כל כתובת לבינארי ותאמת את נכונות הפירוט הבא :

- הקריאה הראשונה היא כMOVFN פספוס, ויש לה סט 00, שכן נכנס את השורה לכינסה הראשונה (האפסית).

- הקריאה השנייה גם פספוס, וגם לה סט 00, لكن נדרוס את הערך הקודם ונחליף אותו בבלוק החדש שלו.
- הקריאה השלישייה גם היא פספוס, רק שעתה היא עם סט 01, אז לא נדרוס את הבלוק הקודם כי הכניסה שלנו ל-01 כרגע פנואה.
- הקריאה הרביעית, החמישית, והששית הן גם פספוס כי דרשו כבר את הערכים שלחן משוכן עם סט 00.
- הקריאה השביעית גם היא פספוס אבל עם סט 11 (או לפחות לא נדרוס בבלוק קודם).

מטמון-2 Way Set Associative-2

ראינו ב-direct mapped שהתנגשויות בין סטים מואוד מזיקות, אך נרצה שלכל כתובות יהיו שני סטים (ways) אפשריים שבהם הם יכולים להיות. את הגירוש בין שתי ה-ways, ננהל לפי היררכית גירוש (כגון LRU). בגלל שסוג השאלה לא משתנה, אלא רק אופן המענה להן. פירוש הכתובות נשאר אותו הדבר להז ב-direct mapped.

דוגמה כך אם הכנסנו שורה אחת ל-way 0 של סט מסוים, ולאחריו נדרש להכנס עוד שורה עם אותו סט, יוכל לשים אותה ב-way מס' 1 בלי לדروس את השורה הקודמת.

דוגמה נrix שוב את אותה הדוגמה עם מטמון 2-Way :

- הקריאה הראשונה מפספסת ונכנסת לסט 0 (way 0).
- הקריאה השנייה מפספסת ונכנסת לסט 0 (way 1).
- הקריאה השלישייה מפספסת ונכנסת לסט 1 (way 0).
- הקריאה הרביעית מפספסת ונכנסת לסט 0, כאשר עכשו נחליט לגרש את 0 way כי השתמשנו בו פעם אחרונה לפני שהשתמשנו בכניסת 1 way.
- הקריאה החמישית פוגעת כי היא מבקשת כתובת שנמצאת בשורה שהשארנו בסט 0 (way 1).

הערה אין חשיבות לדוקא 2 ways, יש גם מטמוניים שהם 4-way ו-ways'ב. כמוון שכלל שיש לנו יותר ways, כך הסיכוי להתנגשות יותר נמוך.

תרגול

הערה בלי להכנס את ה-branch לשלב decode, אפשר עדין להימנע מקפיצה וכ蒂יה ל-PC רק בשלב ה-branch memory access ע"י כך שאט החישוב האם לקפוץ או לא נעשה עוד בשלב execute (כבר אז ידוע לנו האם אנחנו ב-branch והאם תוצאה ה-ALU היא 0 או לא). החסרונו בכך הוא שהוא מאריך עוד יותר את שלב execute שבלאו הכי הוא השלב הארוך ביותר מבין השלבים השונים. בהתאם למימוש המעבד,

דוגמה נניח $CPI_{new} = ?stalling$ CPI באופן אידיאלי, שבתכנית יש 20% פקודות branch, stall על branch. מהו ה-CPI בהתחשב ב-branch-stall שלושה (הוסףנו שלושה stalls-stall-ip).

$$1 + 0.2 \times 3 = 1.6$$

דוגמה רוצים להוסיף לסט הפקודות של MIPS עם pipeline פעולה כפלה. נניח שתזומוני השלבים הם הבאים

IF	ID	Exec	Mem	WB
2ns	1ns	2ns	2ns	1ns

עומדת בפנינו שתי אפשרויות:

1. להוסיף לשלב execute לוגיקה שתחשב את פעולה הכפל. תוספת זו תעלה את זמן הריצה של שלב execute ב-20%.
 2. להוסיף שלב pipeline שבו תהיה לוגיקה שמחשבת את פעולה הכפל באותו האורך כמו ה-execute (כך שיהיה 6 שלבים).
- מה תהיה ההשפעה על הביצועים (throughput ו-latency לאורך כל התכנית) בהנחה שה-pipeline הוא נטול סכנות?
1. ה-latency יעלה $2.4 \times 5 = 12ns$ כי הפעולה הקritisית, הלא היא ה-execute, ארוכה ב-20% מלפני, כלומר אורך 2.4 ns מחייב ערך קבוע את זמן המוחזר המינימלי עבור המעבד.
 2. ה-throughput עתה מחייב כי זמן המוחזר עולה ל-2.4. ויש לנו תוצאה (חוץ מהקצתה) אחרת כל מוחזר שעון, כלומר אורך 2 ns יוביל ל- $12ns \cdot 2 = 24ns$ כי יש לנו עכשו שישה שלבים, שהארוך מביניהם לוקח 2ns וכל הוראה עוברת את כל שלושת השלבים.
- ה-latency לא משתנה כי אחרי שנבצע את חמישת השלבים הראשונים עברו הוראה הראשונה, לאחר כל מוחזר שעון קיבל תוצאה, כלומר מספר תוצאות חישוב ליחידת זמן נשאר זהה - $\frac{1}{2ns}$.

דוגמה עבור קטע הקוד הבאים, נזכיר האם יחייבו stall, או שאפשר לפטור אותם עם forwarding או שהם לא דורשים שם מנגנון מניעת סכנות.

.1

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

אכן יש פה בעיה, מסווג(read-to-use) (קרוי גם interlock), ולכן נדרש nop (הלא הוא (stalling)).

.2

```
add $t1, $t0, $t0
addi $t2, $t0, 5
addi $t4, $t1, 5
```

כאן יש לנו סכנת DATA בין פקודת add 1 ו-2, שכן לפטור בנסיבות עם forwarding (בין הערכים ב-EX/MEM) שמחזיקים מידע על הפקודה הראשונה לקלטים ל-ALU בשלב execute של הפקודה השנייה).

.3

```
addi $t1, $t0, 1  
addi $t2, $t0, 2  
addi $t3, $t0, 2  
addi $t3, $t0, 4  
addi $t5, $t0, 5
```

נשים לב שכאנו אין צורך בשום מגנון למניעת סכנות, כי גם הערך של $\$t0$ לא משתנה וגם $\$t3$ נدرس וכך לא צריך לשמור או לדוחות את הכתיבה אליו מחדש.