

מבנה המחשב | 67200

הרצאות | אוהד פאליד ורונ גבר

כתביה | נמרוד רק

תשפ"ג סמסטר ב'

תוכן העניינים

5	I מבוא ומוליכים-למחאה
5	הרצאה
5	רקע כימי לרנזיסיטורים
6	רקע פיזיקלי לטרנזיסיטורים
7	מוליכים למחאה
8	צומת ח-ק
9	MOS-FET
10	בנייה של שערים מטרנזיסיטורים
11	תרגול
12	II שערים
12	הרצאה
14	סכום מכפלות ומכפלת סכומים
16	יחידות סטנדרטיות בمعالגים לוגיים
17	معالגים סדרתיים
18	معالגים סדרתיים מורכבים על בסיס SR-Latch
19	DFF
20	תרגול
21	מפות קרנו
25	פונקציות שלמות
26	III תזמון מעגלים
26	הרצאה
29	FSM
31	توزון מעבד והגדרות זמני פעולה
36	תרגול
39	MIPS IV
39	הרצאה
39	RISC vs CISC
40	ריגיסטרים-ב-MIPS
40	פקודות אריתמטיות
41	פעולות לוגיות
42	פעולות זכרון
42	פילוסופיות ארגון זכרון
43	פקודות קפיצה והתניות
44	שימוש פונקציות באסבלי
45	פקודות קראיה לפונקציה
45	תרגול
46	אנליזה של מעגלים סינכרוניים
49	синטזה של מעגלים סינכרוניים

51	מעבד V	Single-Cycle
51	הרצאה	
51	קידוד פקודות MIPS	
52	יצוע פקודות ב-MIPS	
53	שימוש השלבים ב-MIPS	
57	תרגול	
59	מעבד VI	Multi-Cycle
59	הרצאה	
61	יחידת השליטה	
62	מעבד רב-מחזורי	
63	שיטות השוואת יצואו מעבדים	
65	איחודי ייחידות במעבד רב-מחזורי	
65	מספר המוחזוריים לפי סוג פקודות	
70	תרגול	
70	דגלי ייחידת הבראה	
73	מעבד VII	pipelined
73	הרצאה	
77	סכנות במעבד pipelined	
77	פתרונות לסקנות	
79	סקנות ב-branching	
80	תרגול	
80	שינויים מרכזיים ברכיבים תחת מעבד רב-מחזורי	
80	אותות בקרה למრבבים במעבד רב-מחזורי	
81	אותות בקרה חדשים במעבד רב-מחזורי	
83	הערכת יצואים	
84	הטמון VIII	the
84	הרצאה	
85	טרמינולוגיית מטמון	
86	מטמון Fully-Associative	
87	מטמון Direct-Mapped	
88	מטמון 2-Way אסוציאטיבי	
89	תרגול	
90	עוד על המטמון IX	on cache
90	הרצאה	
95	תרגול	

X זיכרון וירטואלי

96	הרצאה
96	תרגול
100	
102	
102	חישוב במקביל
106	תרגול

XI חישוב במקביל

102	הרצאה
102	תרגול
106	

שבוע II | מבוא ומוליכים-למחצה

הרצאה

המחשבים הראשונים היו עצומים, כבדים ויקרות, ויחידת הבסיס של המעבד שלהם הייתה שפופורת קטודיות (אבי הטרנזיסטור) ואוז שפופורת ריק, וכיום משתמשים 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 נדי' 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.

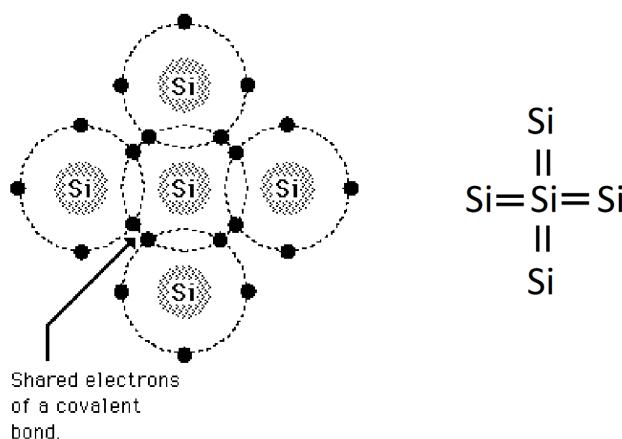
רקע כימי לטרנזיסטורים

המודל של בוחר ניסה להסביר את תופעת התכוונות המשותפות ליסודות באוטה עמודה של הטלחה המחזוריית אותה בנה מנדليب 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 המחזورية, כלומר יש לו 14 פרוטונים (ובמקרה הזה גם 14 ניטרונים) וכן אלקטرونים בשכבות שונות סביבה הגרעין עם מספר שונה של אלקטرونים בכל שכבה.

בזה גילה בוסף 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 חמשה אטומים)



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

יונ הוא אטום או מולוקולה עם מספר לא שווה של פרוטונים (+) ואלקטרונים (-) והמטען של החלקיק הוא הפרש הערכיים האלו.

קשר יוני הוא קשר בין אטומים שנוצר כאשר אחד מהמסלול האחרון של אטום אחר (כדי להשלים מסלול) אבל מספר הפרוטונים באטומים שווה למספר האלקטרונים בכל אטום למעבר האלקטרון. כך לשניים כתה יש מטען מנוגד לשני ומושם ניגודיות המטענים מקרבת (באמצעות כוח משיכה אלקטرون-סטטי) בין האטומים לכדי קשר.

רקע פיזיקלי לטרנזיסטורים

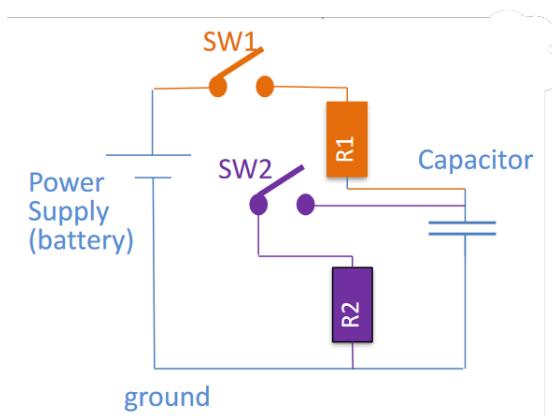
• זרם חשמלי הוא תנועה של אלקטرونים (או באנלוגיה תנועה של חורים, כאשר האלקטרונים השיליליים עוברים בין חורים אחרים חיוביים). בקונבנצייה הזרם נע מה-+---.

• מתח (פוטנציאל) נמדד בולט והוא יכולת להזור, כאשר סוללה עצם מחזיקה מתח וכמו מגדים מים באנלוגיה מים, אפשר לעשות בו חור וכך לגרום לו זורמה אבל כל עוד לא מחברים/מחוררים את הכליל לא יקרה שום דבר.

• מטען הוא מספר האלקטרונים שמאוחסנים בחומר כלשהו.

• קיבול זו היכולת להחזיק מטען, כאשר בעת אחסון מטען נוצר מתח בקבל (מקביל למיכל מים ולחץ).

דוגמא נתבב במעגל הבא. R1 הוא נגד (סקול לצינור צר יותר, שיוצר התנגדות). אם נסגור את המתג הראשון, נסגור מעגל בין הסוללה לקבל כך שאלקטרונים יזרמו מהסוללה לקבל מלמעלה ומהקבל לסלולה מלמטה והקבל מקבל 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 (בחילקו העליון של הקבל היונים החיוביים ומתחתיו השיליליים, והאלקטرونים הם אלו שענים).



בבדיקה של מחשבים נקבע מתח נמוך (עד 1.5V) להיות 0 ומתוך גבוה (פחות 3.5V) להיות 1 - "כיך". בין 1.5V ו-3.5V וולט לא אמררים להיות במצב יציב, רק במעבר בין המצבים. האנלוגיה לכך היא אם מילוי הוא מלא או ריק.

מוליכים למחצה

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

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

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

- **P-type :** נוסיף לסיליקון קצת אלומיניום (לו 3 אלקטטרונים מתק 4 במסלול האחרון) כך שייקשר לאטומי הסיליקון ועתה לחומר יהיה חסר אלקטרון והוא ישmach לקבל אותו, ואז אלקטטרונים יעברו בקלות בין האי-שלמות הallele (פעם ישלים את המחשבור במקודל אחד, ואז יקפוץ לאחר, וכו'). החורים שנוצרים בשאן את האלקטרון הנדרש ליציבות נעים (לשם התיאוריה, בכיוון ההפוך מהאלקטرونים וכן נוצר זרם חיובי בכיוון ההפוך מתנועת האלקטרונים).

- **N-type :** העיקרון הנ"ל רק עם זרחן (לו 5 אלקטטרונים כלומר 1 במסלול האחרון) כך שייצור עודף אלקטטרונים והאלקטטרונים העודפים חופשיים לנوع לאן שירצטו ויצרו זרם.

لمוליך למחצה מסווג P ו-N אין מטען חיובי או שלילי אלא רק סיבובות שונות לתנועת האלקטרונים כתגובה מהרצון להשלים מסלולים אחרים אטומים.

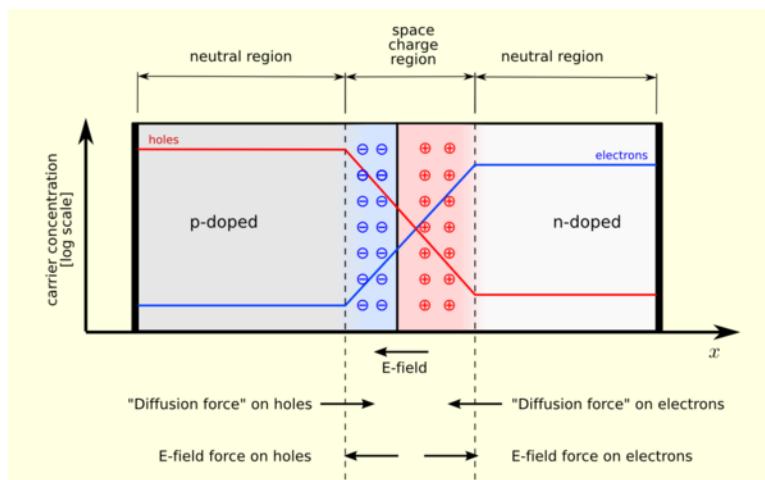
הצמדה של חומרים מסוג P ו-N יוצרת יונים - האלקטרונים מ-N שמחים לקפוץ ל-P שם "צרי" אותם. שני כוחות פועלים בעת הצמדת חומרים שכזו:

- הרצון של המסלולים להתמלא - מה שגורם לאלקטרונים לקפוץ מ-N ל-P.
- הכוח החשמלי שיוצרים האלקטרונים - כוח דחיה בין מטענים שליליים שמנעו קפיצת אלקטרונים מ-P ל-N (N אומר "יש לי מספיק שליליים כבר").

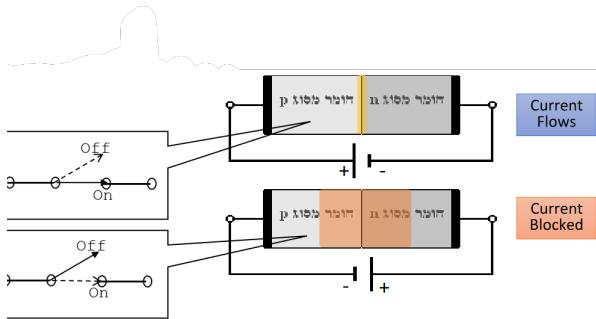
צומתpn

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

בשל תזוזה מקראית של אלקטرونים ופロוטוניים בסמוך לצומת, מצלחים לעبور אטומים יוניים שליליים מ-N ל-P וחוביים להפוך (בניגוד לכיוון הזורימה). כמשמעות התחלפויות אלה קורות, ישנה מסה של אטומים יוניים חיוביים ב-N ומסה של שליליים ב-P שלא יעברו לצד השני בغالל שהם חלק מהגביש כבר. בaczורה זו נוצר מחסום מכוח כוח הדחיה החשמלי שגורם להפסקת הזורימה דרך הצומה והחזקת מתח בו (ראו איור)



עתה חיבור סוללה לדiode נותן לנו תכונות מעניינות.



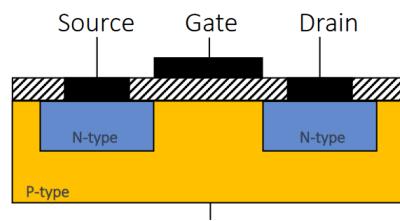
- חיבור סוללה עם + ל-P ו- – ל-N יגרום להרבה מאוד יוניים חיוביים לזרום מ-P ל-N ויוניים שליליים מ-N ל-P (בהתאם לכיוון הזרימה לפני שנחסמה) וכך יצטמצם המרווח באמצעותו עד ל-0 ותהייה לנו זרימה רגילה, כאילו סגרנו מתג.
- חיבור סוללה עם + ל-N ו- – ל-P יגרום ליוניים חיוביים ושליליים להגיע לחומרים ולהזק את הioniים במרווח שכרגע מונעים מעבר ורק יחזקו את החסימה, כך שלמעשה דימיינו התנתקות של פתיחת מתג.

MOS-FET

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

N-Channel ה-ORINANT

באיר ניתן לראות NMOS, וריאנט מבן שניים של MOS-FET (השני משללים לו רק Sh-N ו-P במיקומים הפוכים). החומר המכווקו הוא המבודד והשחור הוא המתכת.

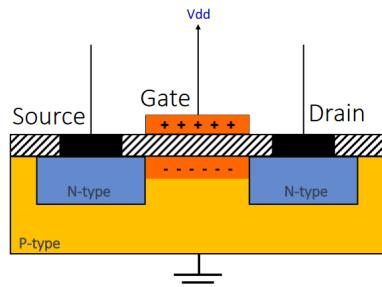


זכור כי זרימה ניתן לקבל רק מ-P ל-N ולכן אי אפשר אף פעם להזרים שום דבר מהמקור (source) לשפך (drain) ולהפך. יש לנו למעשה שתי דיזודות גב אל גב (שקבוצותיהן המקוור והשפך).

אם המתכת של השער מקבלת מטען, האזורי בין השער למוליך-למחצה נהיה קובל כי הוא מחזיק הפרשי מטען!

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

- אם נחבר את השער לאדמה (הארקה), יהיה לנו שני צמתים זה-ק שלא ניתן יהיה להזירם דרךן (ובפרט בין S ל-D דבר).
- אם נחבר את השער למתח, הקבל שהזכרנו לעיל מקבל מתח ועכשו יש מטען חיובי מעליו ושלילי מתחתיו, ככלומר ישנים אלקטטרוניים חופשיים ב-type-p, וכשאלו נוגעים בחומר type-n משני הצדדים יוצרם ערז אלקטטרוניים חופשיים דרךן יכול לעבור זרם, ומכאן השם **n-channel**.



מה שקיבלו בסופו של דבר הוא סוייז' שאנחנו יכולים לשולט בו באמצעות זרם לשער (0 או 1). את הטרנזיסטור נסמן בסימול הבא -
עובד אותו הדבר רק הפוך - הזרמת "0" לשער מסגור את המtag ו-"1" תפתח אותו. ההבדל המהותי היחיד הוא מהחומרים הוא
יש מתח שמחובר מלמטה במקומות הארקה. להלן טבלת סיכום של דרך הפעולה של הטרנזיסטורים.

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

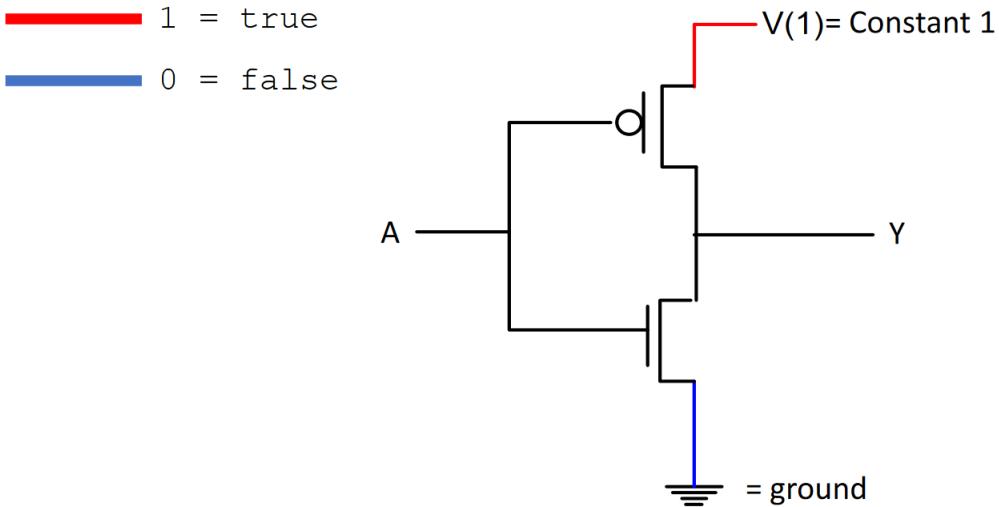
הערה בתחום הכלל, ○ מסמן שליליה, **ב-**PMOS למשל "שוללים" את המתח ומתנהגים הפוך ממנו. גם שערי NOT מסומנים עם עיגול.

בנייה של ריצוף מטרנזיסטורים

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

דוגמה בניית שער NOT (Inverter) מסומן ב-

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



הוא הקלט ו- Y הוא הפלט. כשה- A הוא "1", יעבור זרם של 0 (שהוא זניח) דרך ה-N-Channel למיטה (השער דלוק) ולא יזרום שום דבר דרך ה-P-Channel למעלה (השער דלוק لكن אין זרימה). סה"כ אין שום זרימה עם מתח לא זניח שכן Y יהיה "0".

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

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

תרגול

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

$$(a_4a_3a_2a_1a_0.a_{-1}a_{-2}a_{-3})_{10} = a_410^4 + \dots a_010^0 + a_{-1}10^{-1} + a_{-2}10^{-2} + a_{-3}10^{-3}$$

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

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

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

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

מעבר לבסיס 10 הוא די פשוט (פרישת הייצוג וחישוב מכפלות וחיבור בסיס עשרוני). מעבר מבסיס 10 לבסיס אחר הוא פשוט תחילי איטרטיבי של פעולות מודולו (ν הבסיס) כאשר מה שהוא לא השארית יהיה הספרה הראשונה (משמאלו), השנייה, וכו' עד שנגמרות הספרות. לא כלתי כאן אינסוף דוגמאות להמרות בין בסיסים כי לא מספיק משעם ל.'

במקרה הכללי נמיר שני בסיסים כלשהם דרך בסיס 10 כאשר את הרכיב משמאלו ומימין לספרה העשrownית נטפל באופן נפרד וזהה (עד כדי חלוקה איטרטיבית בשמאלו וככפלת איטרטיבית בימין).

דוגמה נמיר את 0.79272_{10} לבסיס 4. נכפיל את המספר ב-4 ונקבל 3.17088 . לכן הספרה הראשונה היא 3 והשארית היא 0.17088. נבצע זאת שוב ועתה נקבל 0 ושארית 0.68352, וחזור חלילה עד שהשארית תהיה 0, כאשר עתה הספרות ה-0 מלמעלה למטה במקום למטה מלמעלה בספרות הרגילים.

שיטות לייצוג מספרים

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

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

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

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

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

$$\text{דוגמה } 5 = 5 - 4 = (+9) + (-4) = 100100 - 01001 = 11011 \text{ והוא הופך ל-00100.}$$

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

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

$$\text{דוגמה } -4 = -00100 = 11100 - (00011 + 1)$$

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

דוגמה למה שווה 5 – בשיטת המשלים ל-2: $5 = 00101$, נוסיף אחד ונשנה את בית הסימן ונקבל 10110 .

כדי לחסר מספרים, נסכום אותם ונמחק carry אם יש.

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

דוגמה נשתמש בשיטת המשלים ל-1 עם 5 ביטים, $01011 + 01101 = 11000$ קלומר סכמנו חיוביים וקיבלו תוצאה שלילית! הערה הפתרון overflow הוא הוספת ביטים כך שיגדל טווח הייצוג.

שבוע III | שערים

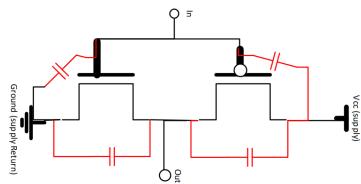
הרצאה

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

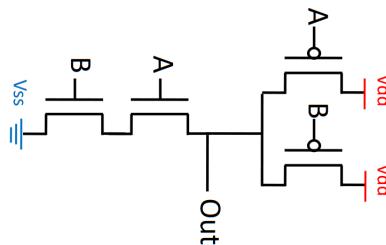
הערה לעיתים לא נרצה להזרים זרם אל תוך טרנזיסטור אחר דלוק (מהכיון הלא נכון) כי זה גורם לסתור.

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

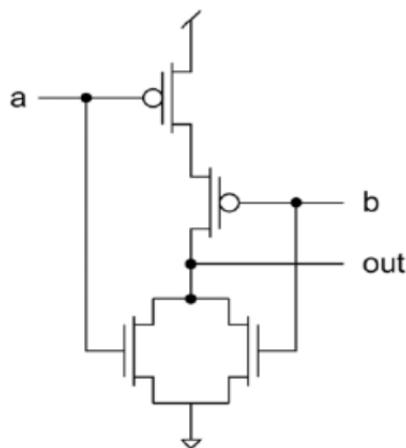
בין שער למקור, בין שער לשפק ובין שער לניצרים קבליים אפקטיביים (לא הוספנו שום דבר, פשוט יש רכיבים פיזיים שמוצאים עצמם מחזיקים מתח בין הצדדים). את הקבליים האלה לוקחים זמן לטעון או לפרק מתח בעת שינוי מצב (שינוי הזרם מהשער, הזרם מהמקור). לכן במצבים רגילים יש זרימה שאנחנו לא בהכרח מצלפים לה במהלך המעבר ($10\text{--}15\text{ ns}$ של שנייה). ראו באյור אדום את הקבליים שנוצרו.



השער היסודי שמאפשר לבנות את כל שער השערים הוא NAND (Not And) - שנותן 0 אם "ס" שני הפלטים 1 (ואחרת 1), ומסומן כך ניתן לבנות את NAND עם CMOS באופן הבא (הריצו פלטים כדי לראותו שזה עובד).



בדומה ניתן לבנות שער NOR באופן הבא (קו אלכסוני הוא (1) V קלומר מתח קבוע דלוק וחץ הוא (0) GND קלומר הארכף)



הגדירה פ' בוליאנית היא פ' שמקבלת קלטיהם בוליאניים (משתנה שיכول לקבל אחד מבין שני ערכיים, ביטים לדוגמה) ופולטת פלט בוליאני אחד בדיק.

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

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

סכום מכפלות ומכפלת סכומיים

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

דוגמה עבור שלושה משתנים וההשמה $x = 1, y = 1, z = 1$ המינטרם היחיד שערכו 1 יהיה $m_6 = xyz'$, ועבור $x = 1, y = 1, z = 0$ יהיה $m_7 = xyz$ (אינדקס המינטרם עם ערך 1 מתקובל ע"י הערך הדצימלי של הסטרינג הבינארי המתקבל מהצמתה 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 בבדיקה.

דוגמה עבור שלושה משתנים עם ההשמה $x = 1, y = 1, z = 0$ המקסטרם היחיד שמקבל ערך 0 הוא $M_6 = x' + y' + z'$ וכו'.

הערה m_i משלים ל- M_i .

הגדירה Sum of Products הוא סכום מכפלות maxterm-minterm产品的。

דוגמה 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	$m_3 = x'yz$	$M_3 = x+y'+z'$
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 הייתה לסקום את כל ה-h-minterm-ים שמקבילים 1 בפ' (בכחול).

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

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

כלומר הצנו בצורה קנונית פ' לכורה מורכבת!

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

דוגמא מולטיפלסקר (MUX) מקבל שלושה קלטים : $S = 0$ ו- D_0, D_1 . 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

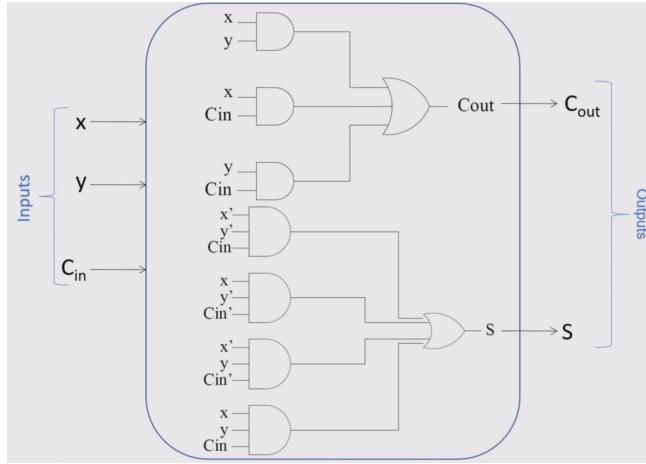
וניתן לרשום את הפ' כ- $C\text{-MUX}$, ואת זה ניתן למשם באמצעות שערים שכבר בנינו. עם זאת, ניתן למשם את המעגל עם פחות טרנזיסטורים מאשר בימוש נאיבי עם שני AND-ים, OR-ים ו-NOT.

דוגמא הוא מעגל שמקבל x, y, C_{in} כאשר S, C_{out} נשא מסכימה קודמת (כasher C_{in} נשא מסכימה קודמת) ופולט C_{out} כאשר S הוא הסכום ו- C_{out} הוא הנשא מתוך הסכום (ראו טבלת אמת) (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 $.S = 0, C_{out} = 1$ $x + y + C_{in} = 0 + 1 + 0 = (10)_2$ אז $x = 0, y = 1, C_{in} = 1$

למעשה, בגלל שיש למעגל שני פלטים, הרי שהוא מורכב משתי פ' בוליאניות. כרגע אפשר למשם את המעגל באמצעות SoP (סכום המינרמלים), ובשימוש הבא צמצמו כמה minterm-ים באמצעות מנות קרנו שנלמד בהמשך (בנוסף מופיע במעגל OR עם שלושה minterm-ים), ואנו נזכיר את המינימיזציה הדרישה ממספר הטרנזיסטורים.



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, מכניסים אותו ל-FA ווחזר חיללה).

יחידות סטנדרטיות במעגלים לוגיים

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

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

כלומר פורש וקטור של n ביטים על 2^n ביטים שככל אחד מייצג מספר מתוך $\{0, \dots, 2^n - 1\}$.
ברגע שיש לנו בלוק של מפענה, אפשר להשתמש בו כדי למשוך פ' בוליאנית באופן טריוייאלי, 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 x, s_0, \dots, s_{n-1} והוא $k = 2^n$ כאשר $x = (s_{n-1} \dots s_0)_2$

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

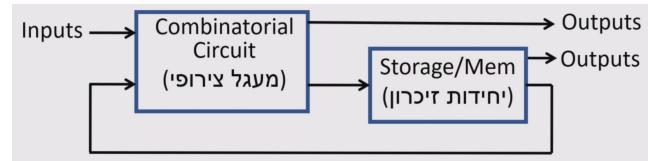
כלומר בהינתן בית, מחזיר מחרוזת שבה הכל אפסים חוץ מאולי הבית ה- $(s_{n-1} \dots s_0)_2$ שערכו x .

3. **מקודד (Encoder)**: הקלט הוא n ביטים x_0, \dots, x_{k-1} והוא $k = 2^n$ כאשר e_0, \dots, e_n כנדרש אם $x_i = 1$ עבור i אחר בדיק (וכל השאר אפסים) או $(e_{n-1} \dots e_0)_2 = (i)_{10}$, כלומר מחזיר את האינדקס (בבינארי) של הבית היחיד הדלק בקלט.

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

מעגלים סדרתיים

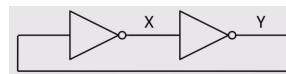
הגדרה מעגל סדרתי הוא מעגל קומבינטורי שחלק מהקלטים שלו הם פלטים של יחידות זיכרון שמחזיק השער (ראו איור)



מעגלים סדרתיים אסינכרוניים יכולים לשנות מצב בכל זמן, ואילו מעגלים סינכרוניים מושנים מצב בהתאם לשעון.

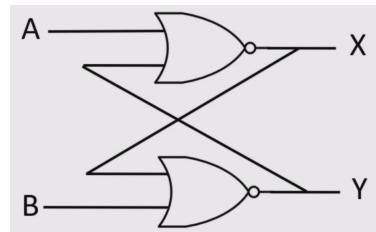
הגדרה שעון סיגナル בצורת גל מחזורי (0 או 1 ואז 0 ואז 1 וכו').

דוגמה כיצד השער הבא יתנהג?



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

דוגמה נביט בשער הבא, שנקרא SR Latch (נזכיר שהשערים בمعالג הם NOR-ים)



הפ' הזו מקיימת $X(t+1) = (A + Y(t))'$, $Y(t+1) = (B + X(t))'$ (כאשר $X(t)$ הוא ערכו של X לאחר שינוי קלטיהם מסונכראן עם שעון שביצעו 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 נשלו רק את הערכים המעניינים, נוכל להביט בתופעה מעניינת (חצאים מעגליים ממשען לאחר טיק נשאר באותו רצף הערכים וחצאים 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								

ונשים לב שם $X = Y$ נקבע מצב לא יציב (לא משנה מה ערכי A, B תמיד יש חץ שייזז אוותנו למצב אחר) ולכן תמיד נניח שאחננו משתמשים ב-SR Latch כאשר $Y' = X = Y = 0$ (זה דומה למצב בדוגמה הקודמת שבו $X = Y = 0$ שזה לא מוגדר בכלל כי יש לנו מהפך עם אותו הערך משני הצדדים).

אם כן, עבור $(X, Y) = (0, 0)$, נקבל ש- $(A, B) = (0, 1)$ שומר על ערכו (כזכור אנחנו מתעלמים מ- Y'), ואם $(A, B) = (1, 1)$ אז אנחנו מופיעים את Y , ואם $(A, B) = (1, 0)$ נקבל מצב לא מוגדר שנתעלם ממנו.

לכן, באמצעות $(S, R) = (A, B) = (A, B)$ נוכל לשלוט בערכו של Y כשייש לנו זכרון של המצב הקודם, עם טבלת אמת מצומצמת נוחה ביותר,

$$\text{כאשר } Q(t) = Y(t) = X(t)' \quad (\text{בהתעלם מהמקרים הלא חוקיים})$$

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

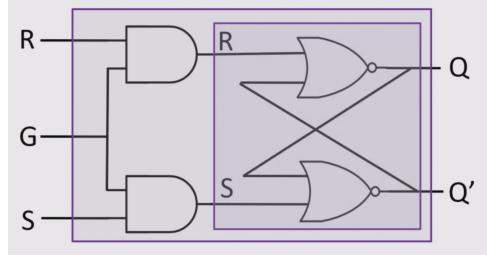
הערה מעגל סדרתיים ניתן לייצג באמצעות טבלת עירור (הטבלה הנ"ל) וטבלת מעברים, שעונה על השאלה "אילו קלטים נדרשים כדי לעבור בין מצב כלשהו לאחר" (Φ הוא ערך Don't care)

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 דлок, קובעים את הפלט להיות 1, אם רק (reset) R דлок קובעים את הפלט להיות 0, ואם זה ולא זה זלקיים לא עושים כלום, ככלומר שומרים על המצב כמוות שהוא. כמוון שתחת סימולרים אלה, גם S וגם R זלקיים זה לא מוגדר היטב.

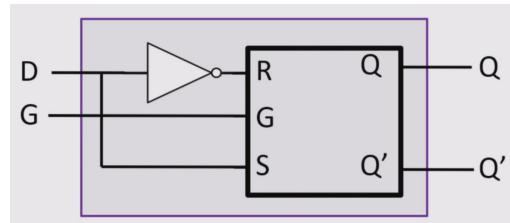
מעגלים סדרתייםים מורכבים על בסיס SR Latch

הוא שער שמנוע פליטתית ערכיים לא חוקיים M-SR Latch, והוא מקבל קלטים S,R,G-AND כאשר G הוא ביט שער שאם הוא דлок אז המעגל מתפרק כ-SR Latch רגיל, ואם כבוי אז הפלטים לא משתנים לא משנה מה. המימוש הוא די פשוט, כולל AND של S,R עם G לפני הכניסה למעגל הפנימי (ראו איור)



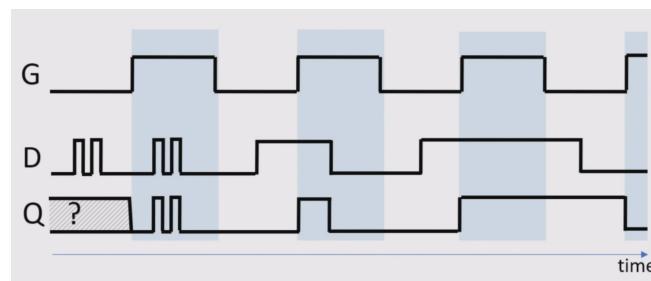
• **Gated D-Latch** הוא גרסה אפילו יותר בטוחה ל-SR Latch - במקומם לקלט קלטי R,S, נקלט D שייהה מחובר ל-S ודרך מהפץ

ל-R, וכך לא נקלט מצב של (1,0), וכך ניתן פשוט לכבות את G (ראו איור)



הערה לרוב לחבר את השעון ל-G, ככלمر אפשר לשנות את הערך רק כשהשעון בטיק שערכו 1.

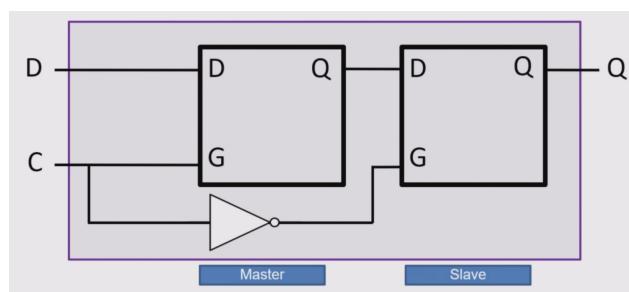
דוגמא בהינתן שעון שמחובר ל-G ובית D שערכו משתנה, נוכל לחשב מה יהיה הפלט Q, כפ' של הזמן. השטח המוקוקו בהתחלה פירשו שלא ידוע לנו מה הערך של Q שכן הוא לא מוגדר בשלב זהה.



D Flip Flop

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

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



לכן רק בעת ירידת השעון נקלט שינוי של הפלט Q, כי לאחר העליה ה-Master ישנה את הפלט ולאחר הירידת ה-Slave ישנה את הערך, הלא הוא פלט המעל כלו. כל עוד השעון לא ירד, הערך ישאר זהה לערך שקיבל בירידת השעון האחרון.

הערה ניתן לבנות מעגל שישתנה רק בעלייה באמצעות הזזת המהפק לפני השער של המאסטר ולא העבד.

תרגול

הגדרה אלגברה בولיאנית היא מבנה אלגברי המוגדר על קבוצת איברים B עם שני אופרטורים בינאריים $\cdot, +$, $x, y, z \in B$ מתקיימות אקסיום Huntington: סגירות לכפל וחיבור, קיום איברי ייחידה לכפל וחיבור, קומוטטיביות בחיבור וכפל, דיסטריביטיביות (שני הנוסחים), קיום משלימים (כך $x \cdot x' = 0, x + x' = 1$) ו-

הגדרה אלגברה בוליאנית דו-ערכית מוגדרת על קבוצה בת שני איברים $B = \{0, 1\}$ עם האופרטורים $x \cdot y = \text{AND}(x, y), x + y = \text{OR}(x, y)$

טענה באלגברה בוליאנית דו-ערכית מתקיימות התכונות הבאות:

- **אידמאונטיות:** $x \cdot x = x + x = x$ לכל x .
- $x \cdot 0 = 0, x + 1 = 1$.
- **אסוציאטיביות לחיבור וכפל.**
- **חוק הצטום:** $x(x + y) = x, x + xy = x$.
- $(x')' = x$.

הערה סדר האופרטורים ביחסוב ביטוי בוליאני הוא קודם סוגרים, או NOT, או AND או OR.

דרכים לבטא פונקציה בוליאנית

- **ביטוי בוליאני** (ביטוי על המשתנים עם שני אופרטורים ושליליה).
- **טבלתאמת**: לטבלה יהיו 2^n שורות כאשר יש n קלטים.
- **סכום מכפלות**: מספיק שמכפלה אחת תהיה 1 כדי שהסכום יהיה 1. כדי להגיע לייצוג סכום מכפלות, סוכמים את כל המכפלות הסטנדרטיות (minterm) שמקבלות 1 בטבלת האמת.
- המשתנה ה- j מקבל שליליה ב-minterm ה- i אם "ם הביט ה- j -ב- $_{(2)}^i$ הוא 0.
- **מכפלת סכומים**: מספיק שסכום אחד יהיה 0 ואז כל המכפלה היא 0. כדי להגיע לייצוג, מכפילים את כל הסכומים הסטנדרטיים שמקבלים 0 בטבלת האמת. ניתן להציג לשקלות לסכום מכפלות עם שליליה על הביטוי ושימוש בשני כלילי דה-מורגן.
- המשתנה ה- j מקבל שליליה ב-maxterm ה- i אם "ם הביט ה- j -ב- $_{(2)}^i$ הוא 1.

נרצה לצמצם את מספר הליטרלים שלנו (מספר השערים).

דוגמה נביט ב- $F(x, y, z) = xy' + x'z - F1(x, y, z) = x'y'z + x'yz + xy'$. את הפ' השנייה ניתן למש עם משמעותית פחות שעריםalogisms (יש הרבה פעולות אבל הפ' שקולות ולכן את השנייה תמיד!).
את השקלות אפשר להראות עם טבלת אמות או באמצעות אלגברה בولיאנית:

$$\begin{aligned} xy'z + x'yz + x' &= x'zy' + x'zy + xy' \\ &= \underline{x'z(y' + y)}_1 + xy' \\ &= x'z + xy' \\ &= F2(x, y, z) \end{aligned}$$

דוגמה נפשט עוד פ'

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

מפות קרנו

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

ראשית נבנה טבלת אמות לפ' הבוליאנית, ואז נציב את המinterm-im בשבלונה של מפת הקרנו אם נצליח לשנן 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 שבסיסים הכהולים למשתנים, באמצעות ערכי x ו- yz ניתן להסיק בקלות את המinterms (יש לשים לב שערכי yz הם לא לפי סדר לקסיקוגרפי)

\bar{x}	yz	00	01	11	10
0	$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$	
1	$xy'z'$	$xy'z$	xyz	xyz'	
					y

הערה ארבעת המשבצות ש- z מסומן עליהם נסכום לליטרל יחיד 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 נמצאת קבוצות של ריבועים סמוכים שערכם בטבלת האמת 1 עבור הפ', כשתוגדל 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 מלבן יכול לחזות את הקצה מימין ולהמשיך משמאלו.

דוגמא עבור $F(x, y, z) = \sum(2, 3, 4, 5)$ (סכום מכפלות עם אינדקסים). מפת הקרןו שלו היא הבאה, כאשר הגענו אליה או באמצעות השבילה או באופן הבא: הסימונים של y , x - 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 מדובר במקרה של $(0, 1, 1)$ שבמקרה שלו זה 1 (כי זהו ערכו של המinterm השלישי שמהגדרת הפ' הוא 1).

\bar{x}	yz	00	01	11	10
0	0	0	1	1	1
1	1	1	0	0	0
					y

כדי לצמצם את הטבלה עכשו נcosa את הטבלה עם שני מלבנים, אחד משמאלו למיטה ואחד מימין למיטה וכן נקבל ייצוג מינימלי של הפ' הבוליאנית שהוא $F(x, y, z) = x'y + xy'$ (במלבן משמאלו משותף הכל חוץ מ- z וכך גם במלבן מימין).

דוגמא $F(x, y, z) = \sum(0, 2, 4, 5, 6)$. הביטוי הלא מצומצם מכיל 5 ביטויים. נמלא איכשהו את הטבלה ונקבל

\bar{x}	yz	00	01	11	10
0	1	0	0	1	
1	1	1	0	1	
					y

נשתמש בחפיפות וגם במעגליות ונקבל מלבן אחד משמאלי למטה ועוד מלבן שכולל את העמודה השמאלית והעמודה הימנית יחד, אז הביטוי המינימלי הוא $F(x, y, z) = xy' + z'$ (בשמאלי למטה נפל z ובמעגלי נופלים x ו- y , פשוט עוברים על זוג ריבועים אופקי וזוג אנכי ורואים מה משתנה בהם, ומה משותף בהם).

מפת קרנו באربعة משתנים נראה כך, כאשר אפשר לזכור אותה באמצעות העורכה שערכי היצירם שלה (גם אופקי וגם אנכי) הם 2, 3, 1, 0 (היצוג הבינארי של המספרים).

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

דוגמה $F(x, y, z) = (0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$

$wx \backslash yz$	00	01	11	10
00	1	1		1
01	1	1		1
11	1	1		1
10	1	1		

נבחר את חצי המפה השמאלי כי זו שמייניה והמלבן הגדל ביוטר שיש, ובשביל הערכים מימין נבחר שתי רביעיות מעגליות (אי אפשר את כולם ביחד כי זה נותן לא חזקה של 2).

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

הריבועיה המעגלית העליונה נבדלת ב- x ו- y ו- w , z מקבלים ערכים 0 שניהם, ככלומר יש לנו $w'z'$ והרביעיה המעגלית התחתונה שונה ב- w ו- y ו- x ו- z מקבלים ערכים 1 ו-0 בהתאם, ככלומר הביטוי הוא xz' .

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

הגדרה לפעמים עברו צירופים מסוימים, לא יהיה מספיק לנו מה הוא פلت הפ', צירופים כאלה נקראים צירופים אדישים ונינן להשתמש בערך שיוטר נוח לנו אליו כך שיבוטלו ליטרלים רבים ככל האפשר. נסמן צירוף כזה ב- \emptyset .

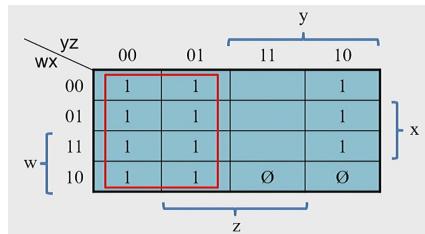
דוגמה $(1, 1, 1)$ הינה הפ' הבוליאנית והצירופים האדישים הם $d(x, y, z) = \sum(7)$ (כלומר רק 7).

$x \backslash yz$	00	01	11	10
0	1	0	0	1
1	1	1	Ø	1

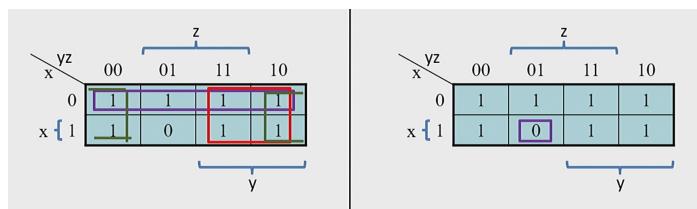
עתה נוכל לבחור מלבנים יותר נוחים (ראו איור) מאשר בהיעדר היצירוף האדיש כי אז לא היינו יכולים לבחור את השורה התחתונה כמלבן והוא לנו אמם עדין שני ביטויים אבל עם יותר ליטרלים, ככלומר יותר שערירים שזה פחות טוב.

הערה יצירוף אדיש מתקיים לדוגמה כשברכיב אלקטרוני איזשהו מעגל בכל מקרה מחובר להארקה כך שלא משנה ערכו עברו יצירוף מסוים כלשהו.

דוגמה נביט במפת קרנו הבאה עם שני יצירופים אדישים. הבחירה הכי נוחה היא 0 לשמאלי ו-1 לימני כי כל קומבינציה אחרת הייתה דורשת מאייתנו יותר מושни מלבנים או מלבנים קטנים יותר (יותר ליטרלים)* שזה פחות אידיאלי.



דוגמה אפשר להשתמש במפת קרנו גם כדי来找出公因子。 נביט ב-(5) $F(x,y,z) = \sum (0,1,2,3,4,6,7) = \prod (5)$. כפי שניתן לראות, מצומצם של סכום המכפלות דורש לפחות 3 נסכמים ואילו מכפלת סכומים דורש בדוק ליטרל אחד.



כמצמצמים פ' בוליאנית לפי מכפלת סכומים, מבצעים בדוק את התהילה של סכום מכפלות רק שסכום 0-ים במקומות 1-ים. לאחר הcisivo, ממירים את הcisivo למכפלה של סכומים, כאשר כל סכום מותאים למלבן אחד.

הערה ניתן להוכיח נכונות של למצטם לפי מפת קרנו למכפלת סכומים או באמצעות דה-מורן למצטם של סכום מכפלות, או פשוט באופן ישיר מטבלת האמת ונכונות יי'זג ה-maxterm-ים.

דוגמה נתונה הפ' הבוליאנית

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

- ציירו את מפת קרנו עבור הפ' F .

המפה תראה כך

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

- כמה פ' שוניות מיוצגות ע"י המפה?

$2^4 = 16$ כי אפשר לבחור ערכים שרירוטיים עבור כל אחד מהצירופים האדישים 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 הזוגות. כך קיבל סה"כ את הכספי הבא

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

שנותן לנו את הביטוי $z'y + x'y + x'w$. זה לא ביוטי מינימלי כי אפשר לבחור בכל הצירופים האדישים יהו 1 חוץ מ-(0, 1, 1, 1).

ואז קיבל הכספי הבא עם אותו מספר ליטרלים

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

פונקציות שלמות

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

משמעות $\{+, \cdot, '\}$ היא שלמה.

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

מסקנה $\{+, \cdot, '\}$ הן שלמות.

■ **הוכחה:** עם דה-מורן אפשר לifycr + עם $' \cdot$ ול- $' +$.

דוגמה NAND, קלומר' $(x \cdot y)' = x' + y'$ הינו פ' שלמה. ראשית ניתן להשיג NOT () באמצעות AND-ו- $x \cdot x' = 0$. אפשר להשיג באמצעות NOT על NAND, שכן אם כבר יש לנו NOT.

דוגמה הוכיחו כי $f(x, y, z) = x' + yz$ והפ' הקבועות 0, 1 הן קבוצה שלמה. ראשית ל-NOT לנitin להגעה באמצעות $x' = f(1, x, y)$. ל-AND אפשר להגעה ע"י $f(x, 0, 0) = x' + 0 \cdot 0 = x'$. לכן ניתן לייצר קבוצה שלמה כלומר הקבוצה המקורי היא שלמה. אפשר גם להגעה 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 שכנן יש לנו.

שבוע III | תזמון מעגלים

הרצאה

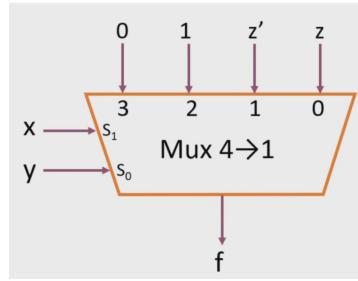
דוגמה נתונה הפ' עם טבלת האמת הבאה

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

- נמשח את הפ' עם 8 MUX $\rightarrow 1$ יחיד. כל מה שצרכי לעשות זה לחבר את z ל- s_0, s_1, s_2 , x, y בהתאם ולהציג בקלטים x_0, \dots, x_7 של ה-MUX את ערכי f בטבלת האמת לפי הסדר. כך נבחר עבור כל צירוף (x, y, z) בדיקות התוצאה מトוקע ערכי בטבלת האמת של f .
- עתה נמשח עם 4 MUX $\rightarrow 1$ יחיד ועוד שער אחד בלבד. כאן צריך יותר להתאמץ. ראשית נקבע 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 X MUX יחיד עם סלקטורים y , x ובקלט ה- i -נשימים או קבועים אם שני היצירופים עם z נותנים את אותו ערך, או z' בהתאם לערך שהערך משתנה (שכנעו עצמכם שאכן אלו כל האפשרויות). כך נקבל את השער הבא



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

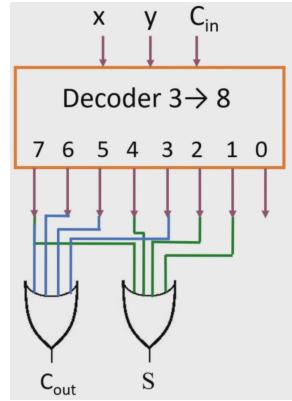
דוגמה נממש FA (שמקבל S, C_{out} ופלט x, y, C_{in}) שיש לו את טבלת האמת הבאה (לצורך נוחות) עם :

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} , ואו על הפלטים נצמיד שער OR אחד לפט S ואחד לפט C_{out} . סה"כ זה יראה כך (רק מוציא חוט לשני ה-OR-ים, בהתאם לטבלת האמת)

(1, 1, 1)

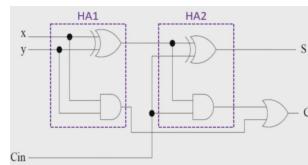


$\bullet .8 \rightarrow 1$ -MUX •

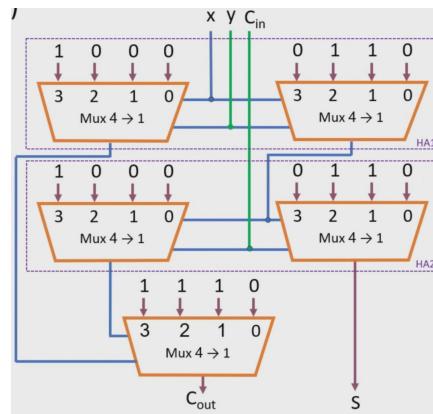
נבחר את הסלקטורים להיות x, y, C_{in} ונסבץ במסונת הקלטים של ה-MUX בטלת האמת לכל צורך של הסלקטורים (הקלטים), והוא זה הדבר שוב עם C_{out} .

$\bullet .4 \rightarrow 1$ -MUX •

זה כבר יותר מרכיב, ודורש פירוק של שני FA לשני Half Adderים שלא הזכרנו כאן, אבל הם שעריים שמקבלים y, x ופולטים S, C_{out} . מימוש די פשוט ניתן לראות בתוך המלון 1, HA1, XOR הקלטים והנשא הוא AND על הקלטים.



כך נוכל לחבר יחד שניים כאלו כדי לחשב $HA.x + y + C_{in} = (x + y) + C_{in}$ עם 4 → 1 MUX (משבצים את ערכי טבלת האמת כאמור) וכל שנותר הוא להרכיב שני HA לאחד יותר גדול (ראו איור)

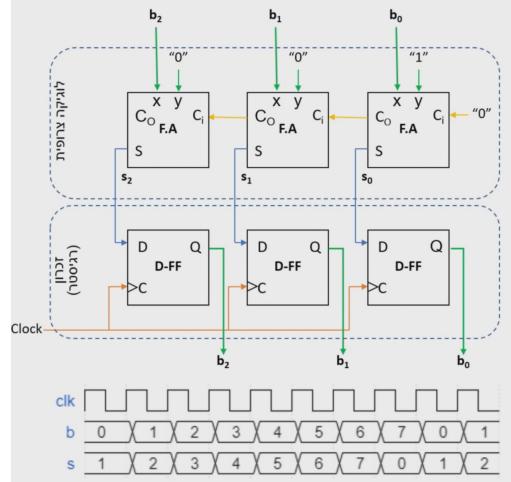


לאחר ראיינו DFF, מקבל קלט D ושעון ומשנה את פלטו בעת עליית השעון לערך של D (זו וריאצית ה-Positive Edge) נוכל על בסיס יחידה זו לבנות יחידות יותר מורכבות.

דוגמה Toggle FF מקבל T ושעון ומחשב בכל עלייה שעון $Q(t+1) = \text{XOR}(T, Q(t))$ ייחד עם T אל תוך XOR שמזון ל-D (קלט ה-DFF הפנימי). כלומר $Q(t) = 0$ יהיה $Q(t+1) = Q(t)'$ אחרת.

דוגמה מקלט JK-FF ושעון ומחשב בכל עלייה שעון $Q(t+1) = JQ'(t) + K'Q(t)$ הרכבת J, K ומחפל Q בשער לוגי שתוצאתו מזנת D -FF של h -הפנימי.

דוגמה נממש ספרן, שסופר את מספר עליות השעון.



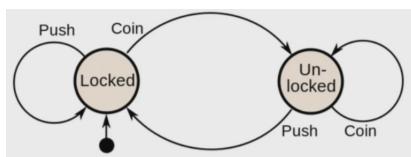
הweeney כאן הוא פשוט אבל המימוש לא כל כך: בכל עלייה שעון, ה-FA הימני ביותר מכניס עוד ערך 1 לסכום שמווחק ע"י כל המעלג. הסכום מופיע בכל עלייה שעון ל-DFF הבא (זה-FA המתאים לו), וכך ה-FA-ים מחזקים שלושה ביטים שמייצגים מספר שערכו עליה באחד בכל עלייה שעון (הסתודנטית המשקיעה תרים את שלושת המחזוריים בראש/על 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 $\rightarrow \{ \text{מצבים} \times \{ \text{מצבים} \} \rightarrow \{ \text{קלטים} \}$ ופ' פלט.

דוגמה שער מסטובב (Turnstile) יכול להיות פתוח או סגור, אם הוא מקבל מטבע והוא נועל, הוא נפתח, אם הוא לא נועל ונדחף, הוא נועל וכו'. כן המצבים הם "פתוח" ו"נעול", הקלטים הם מטבע ודחיפה, הפלט עברור "פתוח" הוא "אפשר לעבר" ובהתאמה עבורו "נעול".

ונכל לצייר את ה-FSM עם גוף

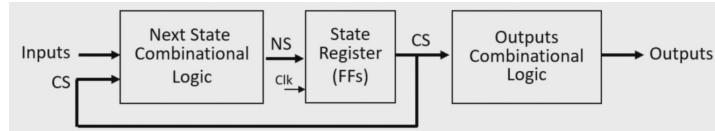


או טבלה (יש כמה דרכים)

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

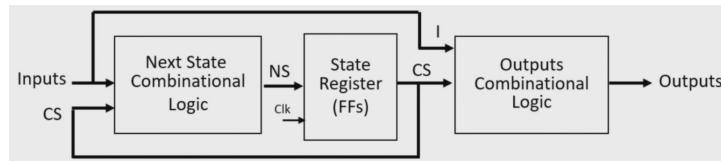
State	Output	Inputs	
		Coin	Push
Locked (Init)	Closed pass	Unlocked	Locked
Unlocked	Open pass	Unlocked	Locked

דוגמה מכונת Moore היא מכונה כבאיור (NS-ים) המציב הבא והנוכחי בהתאם, שמה שמייחד אותה הוא שהפלטים תלויים אך ורק במצב הנוכחי.



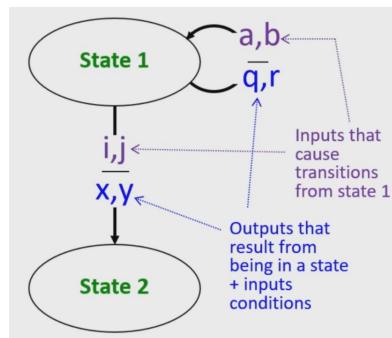
נשים לב שהרגיסטר שמורכב מ-FF-ים מחזיק את המצב הנוכחי, וערכו מחווט חזרה פנימה לחישוב המצב הבא שיכנס לרגיסטר במחזור הבא.

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



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

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



הערה אפשר לכתוב מכונות Mealy שקוולות למוגנות Moore עם פחות מצבים, אבל החסרונו הוא ש-Mealy גורם לביעיות עם תזומנים (שנלמד בהמשך).

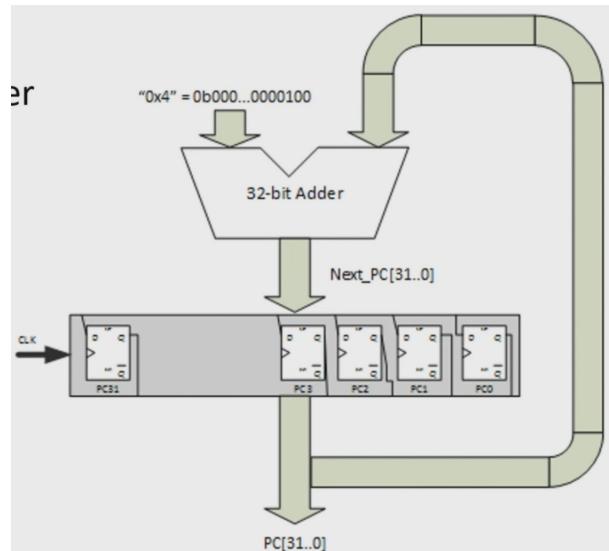
דוגמה נזכיר לדוגמת השער המסתובב. נגדיר 0 השער נעל ו-1 הוא פתוח. לכן המנגנון של פ' הפלט הוא חוט ישיר מהמצב לפלט כי הפלט זהה למצב. את המעברים ניתן לראות בטבלת האמת הבאה

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

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

דוגמה Program Counter נתון למעבד בכל פעם את הכתובת בזיכרון ממנה צריך לקרוא את הפוקודה הבאה. הספרן פולט כתובות באורך 32 ביטים שקובצת בקפיצות של 4 (אלא אם הייתה קפיצה למקומות אחרים) בגלל של מילה היא באורך 32 ביטים (ביה 8 ביט).

המיושן הוא די פשוט: נזכיר 32-DFF-ים שיחזיקו את הכתובת, נחווט ישירות את ה-32 הפלטים והמעגל לחישוב המצב הבא הוא $x = 4 \cdot Q + y = Q$ (המצב הנוכחי), או באյור ברוזולוציה נמוכה משום מה זה יראה כך



זהו מכונת Moore, שיטה מתעדכן פעם אחת בכל מחזור.

זמן מעבד

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

טעינה ופריקה של מטען לוקחים זמן ולכון מעבר או חסימת מעבר של זרם בתוך טרנזיסטור אינם מיידיים (אלא מאוד מהירים). פרק הזמן הזה נקרא **Propogation Delay**.

פרמטרים של זמן פעולה

- 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 נחשב גובה (ומתפרש כ-1) רק כשהוא 90% ומעלה מערכו המקסימלי, וכך יש הנחה סטטיסטית שהעליה והירידה של המתח הם מאוד 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%.

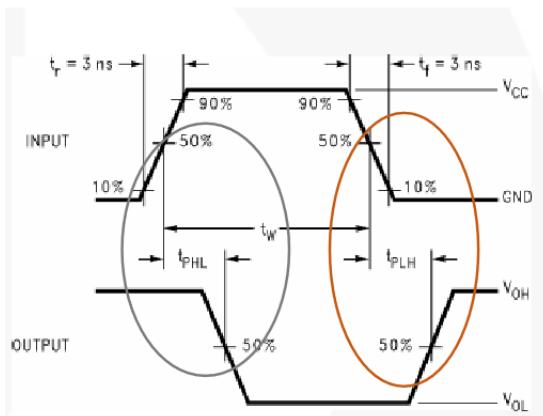
• **זמן עלייה** (t_r , Rise) : הזמן שלוקח לעלות מ-10% מתח ל-90% מתח.

• **זמן ירידת** (t_f , Fall) : הזמן שלוקח לרדת מ-90% ל-10% מתח.

• **זמן פעוף** (t_{pd}) : כ- $t_{pd} = t_{PHL}$, נקרא גם t_{PLH} .

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

דוגמא בשער NOT כלשהו מקבלים את הגרף הבא של הפלט כתלות בקלט.

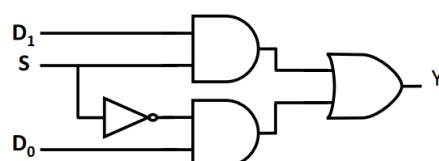


נשים לב שהשינוי הוא לא מיידי. במקרה הזה השינוי הניכר בקלט הוא חצייתו (מלמעלה או מלמטה) של הקולט את רף 50% המתח.

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 לאיזזה רף בטוח הערכים לא ישתנו.

הערה t_{pd_min} יכול להיות שונים עבור שינויים בקלטים שונים (קלט x משפיע יותר מהר מאשר y).

דוגמא נתון השער MUX שמשומש 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 $D_1 \rightarrow Y$ (כמו זמן לאחר שינוי D_1 ישנה) נדרש 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 המקיימים של כל המסלולים, כלומר $23ns$.

- מהו t_{pd-min} של השער כולם?

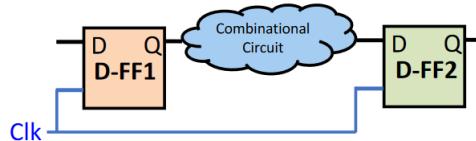
עתה נבחר את המסלול הקצר ביותר, שהוא כMOVED $(D_1 \rightarrow Y)$ או $(D_0 \rightarrow Y)$ שדורש $4ns + 5ns = 9ns$ וזו

הגדולה זמן הפעוף של $D \rightarrow Q$ מחושב (מודגר) החל מעלייה של השעון ל-50% ועד לשינוי Q והוא נקרא t_v (או t_{PCQ}) והוא למעשה מעשה מוגדר אחרי כמה זמן לאחר עליית השעון נוכל להחשב את הקלט כחוקי. t_{v-min} מבטיח עדמתי Q ישר בערכו הקודם ו- t_{v-max} מבטיח החל ממתי יהיה הערך החדש.

הגדולה כדי Q יהיה תקין, צריך להיות יציב ולא להשתנות לפחות פרק זמן לפני עליית השעון, זהו t_s (Setup), וגם לאחר עליית השעון, זהו t_h (Hold).

הערה $t_s > 0$ כי בתוך ה-Master-DFF משנה את ערכו קצר לפני ה-Slave-DFF ולכן הערך שם צריך לא להשתנות כדי של-Slave יהיה את הערך הנוכחי.

דוגמה נבייט 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 המוחזר המינימלי כדי שהמעגל יהיה תקין, כלומר כדי שהתוצאה תגיע מהקלט ל-1-FF עד לפולט של 2 FF תוק שני מוחזרים (בראשו הקלטים עוברים את השער ובשני הם כבר מופיעים הצד השני)?

- זמן המוחזר צריך לקיים את הדרישה

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

כדי שיהיה כוחות שהפלט של FF1 יהיה חוקי (t_{pd-max}), אז למת לו לעבור את כל השער (t_{pd-max}) ואז שהפלט יהיה יציב מספיק זמן לפני עליית השעון הבאה כדי שייעבור בהצלחה ל-Q של FF2 לאחר העליה.

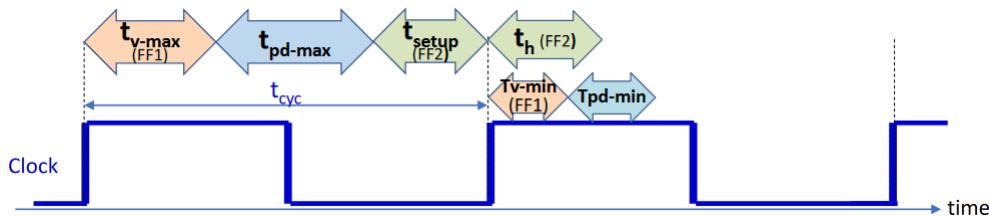
- מה צריך להיות t_h כדי שהשער יהיה תקין?

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

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

כי הזמן שלוקח לקלט לעבור מ- D חסום מלמטה ע"י הזמן המינימלי שעבורו ($DFF1$) (Q) לא ישנה לאחר עליית השעון (t_{v-min}) ועוד הזמן המינימלי שעבורו (D) ($DFF2$) לא יוכל ערך חדש כפלט של המוגל (t_{pd-min}).

סה"כ מחלק התזומנים כפ' של השעון הוא באיר



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

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

- מהו זמן המוחזר המינימלי האפשרי?

כדי שהמוגל יהיה תקין, צריך שמסלול הפעוף הארוך ביותר במבנה יהיה כולל מוחזר אחד. המסלול הארוך ביותר הוא משינויי ב- $(S0/Q)$, דרך חישוב ה-MUX ועד לשינוי הערך ב- D ,ऋיך לחתות בחשבון שלפני תחילת המוחזר הבא נדרש ש- D יהיה יציב לפחות t_{setup} שניות. סה"כ נדרש שיתקיים (בזומה 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?

נדרש שהערך של D לא ישנה מוקדם מדי לאחר עליית השעון, בפרט שזמן שינוי הערך Q וחישוב ה-MUX יקחו יותר מאשר $t_h = 5 \leq t_h \leq t_{v(min)} + t_{pd(min)} = 2 + 9$.

$$\text{סה"כ } F_{max} = \frac{1}{33ns} = 30MHz \text{ כלומר } T_{cyc(min)} = 33ns$$

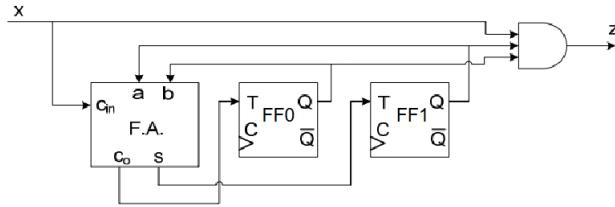
עד כה הטענו מהקלט האמייתי של המוגל שמחובר ל-S1. הוא עצמו גם חייב לקיים דרישת שינוי שלו ביחס לשעון וכן להתייחס אליו כסיגナル שמגיעה מפלט אחר והניתוח יהיה כנ"ל.

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

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

הערה כשנתח תזמון של מעגל, נסתכ אל כל המסלולים שמתחלים בכניסה ל-DFF ונגם

דוגמה נסתכ אל המבנה הבא, כשהנתנו $0 \leq t_{setup} = 9, t_{hold} = 0$ ודרישות על הפלט $t_{v(max)}^x = 15, t_{v(min)}^x = 4$ נדרש להיות יכיב לפחות $9ns$ לפני העלייה 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 $\{a,b,Cin\} \rightarrow S$	12	3				
FA $\{a,b,Cin\} \rightarrow Cout$	8	3				
T-FF			7	2	4	0

• מהו $T_{cyc-min}$? נתח את כל המסלולים שנגמרים בפלט (כי לפלט יש דרישות ביחס לשעון, בפרט ל-DFF יש טכני ולפלט

יש כזה שנדרש מאייתנו)

– מסלולים שנגמרים ב-z: המסלול המקסימלי הוא באורך

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

כי z נדרש להיות יכיב זמן לפני העלייה השעון, ערכו מחושב ע"י AND או מוסיפים את זמן הפעוף דרךו, וקלט

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

מסמנים $.(t_v = t_{v(max)})$

– מסלולים שנגמרים ב-T (FF1): המסלול המקסימלי הוא באורך

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

– מסלולים שנגמרים ב-(FF0):

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

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

- האם מתקיימת הדרישה על ה-Min Delay ?

כן ! עבור z מתקיים $t_h^z = 0ns$ והזמן המינימלי לפעוף הוא

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

כלומר מתקיימת הדרישה ועבור $t_h^{FF} = 2ns$ יש FF0, FF1 והוא מינימלי לפעוף הוא

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

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

תרגול

הגדירה מעגל צירופי (קומבינטורית) הוא מעגל שיש לו כניסה ויציאות כאשר האחרונות תלויות בכל ערך ורגע של הראשונות. מעגל סדרתי הוא מעגל שפלטי תלויים גם ביחסית זכרון שמחוברת לשעון.

דוגמה כיצד נממש Full Adder (כזכור קלטים C_{out} , S , C_{in} ופלטים x, y, C_{in} ו- y) ? ראשית נמלא את טבלת הקרנו לפלט S (משמאלי ו-מימין)

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

ולכן אפשר למשם את S עם ארבעה פלטי AND (שכוללים קלטים שעוברים דרך מהפץ) ואת C_{out} אפשר קצת יותר יעיל. ראיינו בהרצאה שהIMPLEMENTATION של FA כולל בתוכו שני HA.

הגדירה מחסרים מחשבים חישור ספורות בינאריות. עטן נפלוט את ההפרש (ערך מוחלט) ו- $Borrow$ שיגיד לנו כמה יותר לחסר מעבר להפרש. הסטודנטית המשקיפה תמשח חצי-מחסר וממחסר מלא.

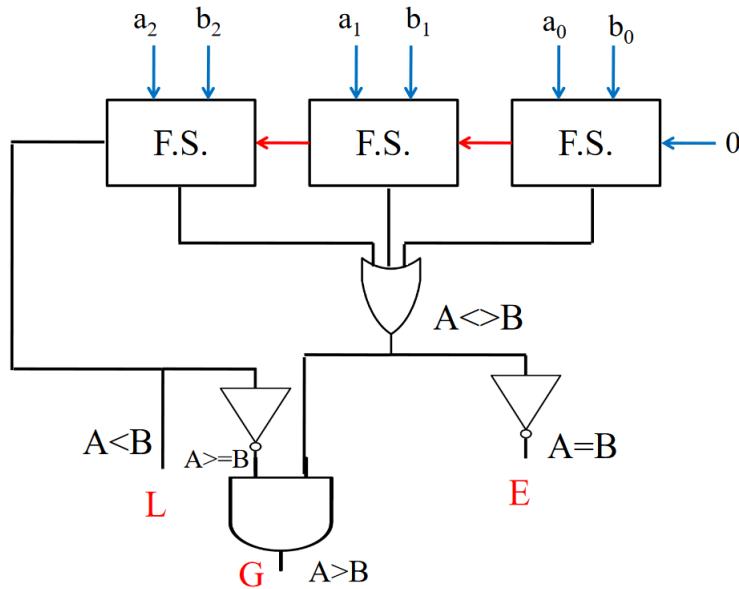
הגדירה משווים הם מעגלים שבודקים אם איזה קלט יותר גדול.

- דרך אחת למשם זאת היא באמצעות השוואת מה-LSB ל-MSB כאשר בפעם הראשונה שיש אי-שוויון בביטים נבחר את האחד שקלטו גדול יותר (1 לעממת 0).

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

דוגמה נתונים הקלטים $A = a_2a_1a_0$, $B = b_2b_1b_0$ ממשו עם מחסרים מעגל שפלט ביטים G, E, L שערך כל אחד מהם 1 אם "מ" $A > B$, $A = B$, $A < B$ בהתאם.

נמשם את המעגל כבאיור



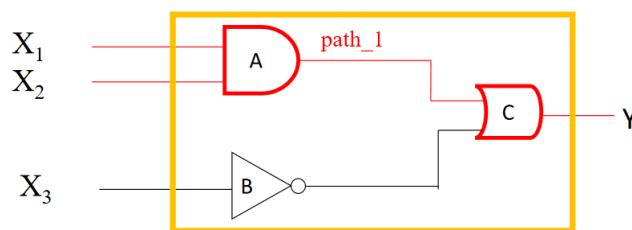
אם $a_2 < b_2$ בהכרח ישאר לנו Borrow Out כי אם BO מ לפני שימוש הלהה ואחרת וכן בודאי שהיא BO.

אם כל הביטים זרים נוצר NOR על כל הפרשים (כל החיסורים פולטים 0) וכן זה מה שבנוינו.

בשיטת האלימנציה, G הוא 1 אם $A \neq B$, ואם $A \geq B$ הוא 0 וכן אכן מופיע בمعالג.

הערה השימוש ב-10%-90% כרף לחישובי תזמון נובע מכך שקשה לאפיין את פריקת הקבל כתהיליך לינארי בקצבות, ולכן מעריכים ממהם.

דוגמה נתיב בפ' $X_1 * X_2 + X'_3$ שטמומשת באופן הבא



מתקיים $t_{pd}(Y) = \max\{t_{PLH}(A), t_{PHL}(A)\}$ ל- X_1, X_2 ו $t_{pd}(Y) = t_{cd}$ ל- X_3 . יש שני מסלולים בمعالג, הראשון מ- X_1, X_2 ל- C והשני מ- X_3 ובהזמנה עבור C .

לכן $t_{pd}(Y) = t_{pd}(B) + t_{pd}(C)$ תחת הנ吐נים הבאים (כאשר $t_{pd}(B) = t_{pd}(A)$ ובהזמנה $t_{pd}(C) = t_{pd}(A) + t_{pd}(C)$)
 $t_{pd}(Y) = t_{pd}(A) + t_{pd}(C) + t_{cd}$ (Contamination time $t_{cd} = t_{tp-min}$)

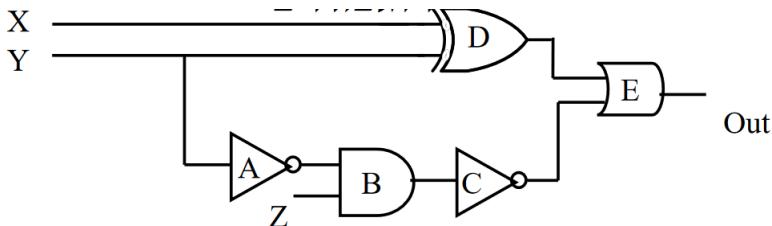
Data in ns	X_2	A	B	C
t_{PHL}	-	100	90	80
t_{PLH}	-	110	70	100
t_{cd}	-	12	8	10
t_r	14	20	12	18
t_f	15	17	13	19

נחשב את ה- t_{pd} של המעגל כולם. עבור כל שער, וולכן $t_{pd} = \max \{t_{PHL}, t_{PLH}\}$

$$t_{pd}^{p2} = 90 + 100 = 190ns$$

נחשב את ה- $t_{pd(min)}$ של המעגל. $t_{pd(min)} = 12 + 10 = 22ns$

דוגמה נתונים המעגל והנתונים 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 מתקלטים לפטיטים הם $B \rightarrow C \rightarrow E$, $A \rightarrow B \rightarrow C \rightarrow E$, $D \rightarrow E$ ו- $60 - 75 - 80 - t_{pd}$ בהתאם.

ו- $15 - 20 - 15 - t_{pd(min)}$ בהתאם.

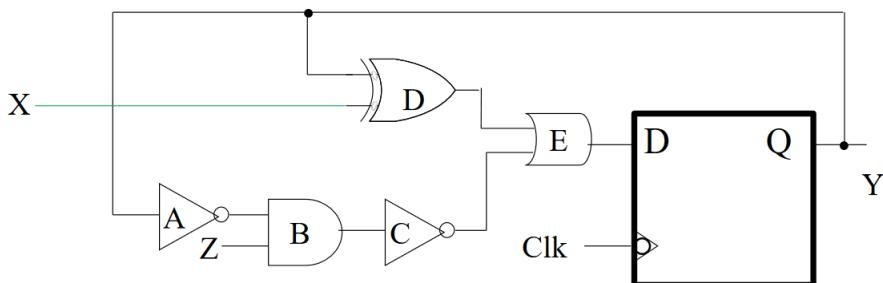
לכן $t_{pd(min)} = 15 - t_{pd} = 80$.

הערה כל עוד אין מעגלים סדרתיים, חישוב t_{pd} נעשה ע"י מינימום הזמנים על כל המסלולים מפלטיטים לקליטים ו- ע"י מינימום.

הערה עבור כל מעגל סדרתי חיבר להתקנים $t_{hold} \leq t_{v(min)}^{FF} + t_{pd(min)}^{לייק}$ כדי שהשינוי הכוי מהיר במעגל יקח יותר מאשר הזמן שהפלט צריך להישאר זהה אחריו עלילית השעון.

הערה הקלט למעגל חיבר להטייך לאחר כל היותר $t_{pd} + t_{setup}$ זמן כדי שה-FF יוכל לחשב באופן תקין את Q .

דוגמה נתון המעגל הסדרתי הבא עם הנתונים 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 תזמון המעגל נctrarק את האילוצים על ה- DFF $t_{v(min)} = 5ns$, $t_v = 10ns$

$.t_{hold} = 25ns$

- האם המעגל תקין?

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

- כיצד נפתרת הבעיה?

נוסיף 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 $25 = t_{hold} \leq 5 + 25 = 30$ ועכשו נקבע $t_{pd(min)} = 25$ זה כן תקין. החסרונו הוא שהתדר המקסימלי האפשרי ירד כי זמן המחווזור המינימלי עלה כתוצאה נוספת שני ה-NOT-ים.

MIPS | IV שבוע

הרצאה

הגדרה Instruction Set Architecture היא אוסף כללים שמתקנת צריך לעמוד להם כשהוא מפתח למעבד.

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

הגדרה מיקרו-ארQUITטורה היא מיומש של ISA.

דוגמה המיומש של אינטל ו-AMD ל-x86 הוא מיקרו-ארQUITטורה.

לצורך האבstraction שלנו, מעבד הוא מכונת מצבים המוחוברת לזכרון, כאשר המצב כולל רגיסטרים שערכם משתנה על ידי פקודות. הזיכרונו 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 לפקודה הבאה).

3. בצע את הפקודה (ושנה את מצב המעבד).

4. חוזר לשלב 1.

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

הערה לעיתים הקוד שלנו ישמש בספריות חיצונית או בקבצי קוד אחרים, ועל איחוי כל הפקודות לקובץ אובייקט יחיד.

הפקודות באסמביל הן פקודות שהמעבד יודע לבצע (ה-ISA של המעבד), אבל לפני שהן מקודדות ל-1-ים ו-0-ים עברו המעבד.

CISC vs RISC

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

x86 הולכים לפי גישת CISC

Reduced Instruction Set Computer • גישה לפיה יש לשמור את מספר הפקודות מצומצם וכך לפחות את פעולות החומרה, ולתת לקומפיאילר לעשות את העבודה הקשה של בחרת הפעולות ואופטימיזציה. הפקודות הן די פשוטות והוא רק בין רגיסטרים (ולא על הזיכרון), וצריך באופן מפורש לטעון ולשמור נתונים בזיכרון. אורך הפקודות הוא קבוע. RISC-V ו-RISC הולכים לפי גישת MIPS, ARM

הערה המגמה עם הזמן היא לעבור מ-CISC ל-RISC משום שבמעבר קשה היה כתוב קומפיאילרים עילים וכן נדרש המורכבות של פקודות האסמבלי, ואילו עם חלוף הזמן יהיה קל ויעיל יותר לכתוב קומפיאילרים (בשפה עילית) שיבצעו את הפעולה המורכבת בעצם.

דוגמה עבור העתקה של 100 ערכים בין מערך אחד לאחד ב-c, יש ב-86x פקודה אחת שמקבלת את מספר הבטים להעתקה והפונקציות והחומרה כבר תמשח את העתקה, לעומת זאת RISC שם צריך למשולש שמעתקה לריגיסטר ואז לזכור מילה מילה.

MIPS

במעבד MIPS יש 32 רגיסטרים, \$0, ..., \$31, שכל אחד מהם בגודל 32 ביט, המכונה "מילה" (4 בתים). מספר הרגיסטרים נקבע כך לאחר ניתוח של מספר המשתנים בתוכנות מוגניות, כאשר אם יש יותר משתנים מריגיסטרים השתמש בזיכרון הראשי כדי להחזיק את ערכם. חלק מהרגיסטרים יש יעודים 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

נשים לב שרבים מהרגיסטרים משמשים פעילות תקינה של המחשנית (SP, FP, RA), חלק שומרים ארגומנטים וחלק משומשים לצרכים אחרים.

כל פקודה היא בגודל מילה (32 ביט), וכל פקודה מבצעת פעולה פשוטה, בין היתר פעולות אРИתמטיות ולוגיות, גישה לזכרון (load, store) ופקודות מותנות. הפקודות שמורות בזיכרון ובכל פעם נקרא מהזכרן את הפקודה ונrai איתה.

פקודות אРИתמטיות

- **חיבור/חיסור :** נבייט בפקודה $\$t = \$d + \$s$. היא מ לחברת את התוצאה $\$t$ ושם אותה $\$s$, $\$d$ (כאשר שלושת הפרמטרים הם רегистרים), ובדומה $\$t = \$d - \$s$.

נשים לב שלא הודיעו למבצע בשום מקום שאנו ממשמשים בשיטת המשלים ל-2, כי אנחנו מניחים שמי שקימפל את הפקודה יודע מהקשר שהוא סוכם רגיסטרים שיש בהם כבר ערכיהם מיוצגים במשלים ל-2, ואם לא אז זה באג.

$$\text{דוגמה נקמפל את הפקודה } f = (g + h) - (i + j)$$

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

`add $t0, $s1, $s2`

`add $t1, $s3, $s4`

`sub $s0, $t0, $t1`

אבל יש דרכים אחרות (לדוגמה לפתוח סוגרים). במתמטיקה אמנס התוצאות שקולות, אבל כאן יכול להיות שנקלט תוצאה אחרת בגלגול-overflow-ים.

- **חיבור מיידי :** הפקודה $i = \$s + \t מבצע חיבור בין רגיסטר וקבוע והשמה ברגיסטר). כן i נתון לנו במשלים ל-2 אבל בגודל 16 בית כדי שנוכל לכלול אותו בתוך קידוד הפקודה בגודל מילה אחת, ולכן נדרש להרחיב אותו למשלים ל-2 בגודל 32 ביטים, פולה זו נקראת Sign Extend, וניתן למימוש בклות ע"י ריפוד בצד של ה-MSB עם ערך קבוע של 0 לחובבים ו-1 לשיליליים (למעשה ערך ה-MSB לפני הריפוד).

הערה לא צריך `sub` כי אפשר למש בклות עם `addi` כאשר ה- i הוא מספר שלילי.

פעולות לוגיות

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

Instruction	Operation	Description
sll \$d, \$t, a	Shift Left Logical \$d = \$t << a	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.
sllv \$d, \$t, \$s	Shift Left Logical \$d = \$t << \$s	Shifts a register value left by the value in a second register and places the result in a third register. Zeroes are shifted in.
sra \$d, \$t, a	Shift Right Arithmetic \$d = \$t >> a	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. The sign bit is shifted in.
sraw \$d, \$t, \$s	Shift Right Arithmetic \$d = \$t >> \$s	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.
srl \$d, \$t, a	Shift Right Logical \$d = \$t >>> a	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. Zeroes are shifted in.
srlv \$d, \$t, \$s	Shift Right Logical \$d = \$t >>> \$s	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 שמאלה פרושה הכפלה ב-2 (עד כדי overflow) וימינה פרושה חלוקה ב-2 (עם עיגול כלפי מטה).

פעולות זכרון

הזכרון הוא מערך, שניגשים אליו באמצעות כתובות, כאשר כל כתובת מצביעה לבית אחד של מידע, למרות שבפועל ב-MIPS נקרא ונכתוב ביחידות של מילה (ארבעה בתים) מיושרות (כלומר כתובות שמתחלקות ב-4).

- קריית מילה : `lw $t, i($s + imm)` הוא ה-immediate i קוראת מילה מהזיכרון בכתובת `$s + imm` ושומרת אותה ב-\$t.

דוגמה `lw $t1, 0($a0)` קוראת מילה מהכתובת שמורה ב-\$a0 וכתוות אותו לרגיסטר `$t1`.

- כתיבת מילה `sw $t, i($s + imm)` כתובת את ארבעת הבטים ברегистר `$t` לכתובת `$s + imm` בזיכרון.

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

דוגמה נניח שיש לנו את הפקודות ב-C

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

הקוד הזה יתורגם באסמבלי ל-

```
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 וא-ניומן : זכרון אחד לפקודות התוכנה וגם לששתני התוכנה. כש庫ראים מהזיכרון, משמעות התוכן (פקודה או מידע) תלויה בפעולה שהובילה לקריאה. אפ"פ שתיתכן הפרדה פיזית בין החלקים, לוגית הם ממופים לאותו המקום.

יתרונות אזור זכרון מאוחד, וכל לדבג תוכנות ולשנות את אופן הטעינה.

חרוגות קריאת פקודות וקריאת מידע קוראות על אותו המשאב.

ממומש ב-**x86, MIPS, ARM**.

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

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

חרוגות חוסר גמישות בגודל הסגמנטים של קוד לעומת DATA וקשה יותר לעורך את הקוד לדיבוג.

ממומש בעיקר ב-**DSP**.

הערה גם בוואן-ניומן המימוש בפועל יכול להיות באמצעות רכיבים פיזיים שונים, הגישה תהיה באמצעות אותו מרכיב כתובות (אם כי כתובות אחרות, אבל באותו מיפוי).

פקודות קפיצה והתנויות

- קפיצה בלתי-מוותנת: `label j` קופץ לאחר סיום הפקודה לשורה הקוד שמופיע לאחר ה-label (`label` כתוב בקובץ asm).

• קפיצה מותנת שווין: `label $s==$t` קופץ ל-label `label` אם `$s == $t` ואחרת ממשיכה לפקודה הבאה (ובדומה `bne` ש קופצת אם `.$s != $t`).

דוגמה נתון הקוד הבא ב-**C**:

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

הוא יתרגם לקוד אסטטטי 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: $\$s < imm$ $\$t = slt$ $\$d, \$s, \$t$ אם $\$t < \s ואחרת, בדומה i , שם ב- $\$t$.
- ואחרות.

מימוש פונקציות בא מסבל

הגדולה הקוראת היא הפ' שקוראת לפ' אחרת, הנקראת היא הפ' שנקוראת, הפרמטרים הם הערכים שמשוערים מהקוראת לנקוראת, המוצאות הם הערכים שהנקראת מחזירה לקוראת וכתובות החזורה היא הכתובת בזיכרון הקוד של הפוקודה שאחורי הקראת לנקוראת בקוד של הקראת.

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

הערה לעיתים אפשר לנשთ גם לערכים אקריםם במחסנית, ובכל מקרה ניגשים לכתובות ביחס לכתובת ראש המחסנית - Top of Stack, כאשר כל דבר מתחת איינו ואלידי. זאת משום שהמחסנית גדולה נגד כיוון הכתובות, כלומר הוספה ערך תיזה את TOS ארבעה בתים למטה.

שמירת רגיסטרים בעת קראת וחזרה מפונקציה

הנקראת לא יודעת מה הקראת עשויה, ורק מצפה לפרמטרים נכונים ולהוציא תוצאות נכונות. לכן אם הקראת משתמשת ברגיסטרים בלבד, הנקראת תזרוס אותם בלי לדעת שהקוראת צריכה אותם. כדי לשמר את המצב לפני ואחרי קראת לפ' יש *ישotton*; נניח ש-*f* (הקוראת) הייתה באמצעות חישוב שකראה ל-*g* (הנקראת). רק חלק מהרגיסטרים נשמרים, וע"י גורמים שונים:

- $\$f-\$0-\$1$ הם רגיסטרים זמינים ולבן לאחריות *f* (הקוראת) לשמור אותם במקום אחר במהלך הקראת ל-*g*.
- $\$7-\$8-\$9$ הם רגיסטרים סטטיים ולבן *f* מצפה שהם לא ישתנו, כך שגם *g* משתמש בהם היא תצטרכן לשחזר אותם לערך בטוטם קראתה כדי ש-*f* תתפרק כמו שצሪך.
- $\$a0-\$a1-\$a2$ משמשים העברת והחזרת ארגומנטים ולבן הקראת צריכה לשמר אותם אם היא רוצה להשתמש בערכים שהוא קיבלה (או תחזיר) בתור נקוראת.
- $\$ra$ הוא הרגיסטר שמחזיק את כתובות החזרה מהפ', והוא נדרשת ע"י הפוקודה *jal* בעת הקראת לפ' הנקראת, כך שאחריות השימור היא על הקראת.
- $\$sp$, המצביע לראש המחסנית, שנדרש לכל הפ' על מנת לפעול באופן תקין, ולאחר מכן נדרש לשחזר אותו בעת סיום הקראת לפ'.
- את הערכים נשמר תמיד במחסנית.

- כדי לדוחף (ולשמור) את \$ra למחסנית נשתמש בפקודות

```
addi $sp, $sp, -4
```

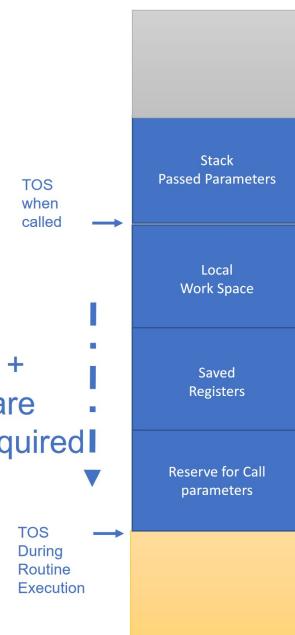
```
sw $ra, ($sp)
```

כלומר קודם מזיזים את המצביע לראש המחסנית מילה אחת למטה בזיכרון (מעליהם את גובה המחסנית) ואוז כתובים לכתובות הבסיס של המחסנית את הערך של \$ra, כך שהוא עכשו בראש הערימה.

- כדי לגשת לערכיהם אפשר פשוט לבצע lw על כל היסט (חיבוי) ביחס ל-\$sp.

- כדי לעשות pop קודם נקרא את המילה ואוז נזיז כלפי מעלה את \$sp.

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



פקודות קריאה לפונקציה

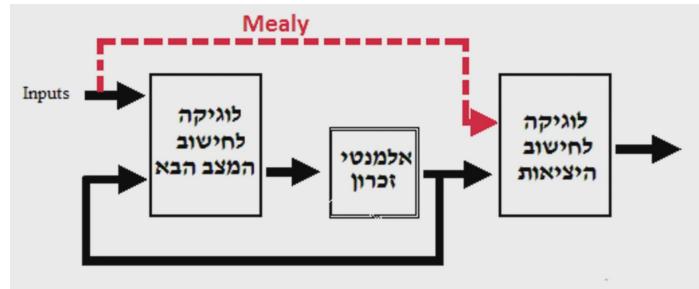
- label jal \$ra שם ב-\$ra את כתובות הפקודה הבאה (הערך הבא שיקבל PC) ואוז קופצת ל-label.

- \$.jr \$s קופצת לכתובות שברגייסטר \$.s.

כשנקרא לפ', נרים jal, וכשנחזיר מפ', נרים \$ra jr.

תרגול

להדגמת ההבדל בין מכונות Moore ל-Mealy, ראו האיור הבא כאשר בשחור מכונה Moore ובאדום התווסף שהופכת אותה למוכנת Mealy.



אנליזה של מעגלים סינכרוניים

בහינתן מעגל, נרצה להבין מה הוא עושה (איזו מכונה הוא מייצג, איך הוא מתנהג). השלבים לאנליזה הם :

1. הצגת הקלטים לזיכרון ופלטי המעגל באמצעות הפלטיהם מהזיכרון והקלטים למעגל.

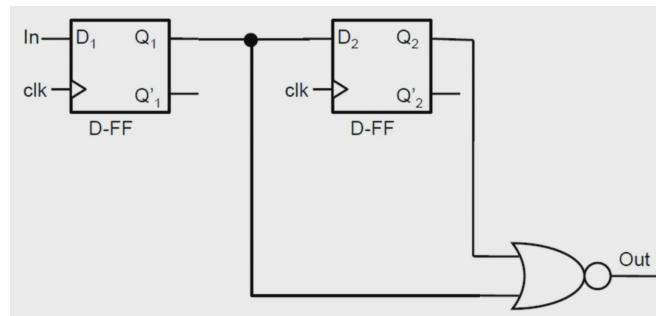
2. כתיבת טבלת מעברים.

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

4. רישום אוטומט מצבים.

5. ניתוח האוטומט באמצעות סדרת בוחן.

דוגמה נתון המעגל הבא



1. בклטי הזיכרון מתקיים $D_1 = In$, $D_2 = Q_1$ ובפלט המעגל מתקיים $'Out = (Q_1 + Q_2)'$

2. טבלת המעברים תראה כך, כאשר אנחנו עוברים על כל אפשרות ל- In, Q_1, Q_2, D_1, D_2

	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 מהקונפיגורציות לא הגינניות, אבל עדיין נכתבות אותן.

3. ניתן שמות למצבים, כאשר מצב מוגדר ע"י קיבועם ערכיים של כל פלטי רכיבי הזיכרון, 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

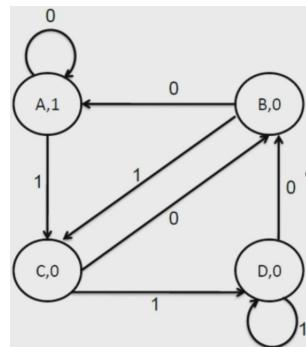
ועכשיו נוכל לחשב את הפלט של המעגל בהינתן הקלט וה המצב הנוכחי בטבלת המצבים הבאה (PS המצב הנוכחי ו-NS המצב הבא)

(הבא)

PS	NS		
	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 להחיליט האם לבנות אוטומט Moore או Mealy, ובשלב הראשון עוד יוכלו לשים לב שהמעגל

הЛОגי שמנדריך את הפלט תלוי רק בזיכרון ולא בקלט וכן נסתפק באוטומט Moore



5. נתח אוטומט, כשם שמשמעותו בסופו של דבר היא אליו מקרים המכונה פולטה 1. השתמש בסדרת בוחן, ככלומר טבלה עם

שלוש שורות: מצב, פלט, והקלט שאיתו נבעור במצב הבא בשורה. נdag שהמעברים בין כל שני תאים סמוכים בשורת המצבים יכסו את כל המעברים הקיימים באוטומט (כל הקשתות) לפחות פעם אחת. אין חשיבות לסדר המעבר.

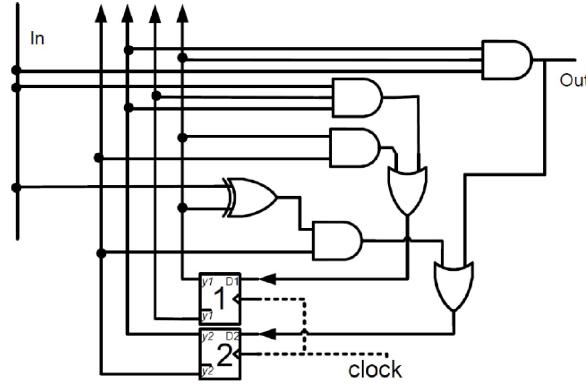
In	0	0	1	0	0	1	1	0	1	0	1	0	0	0
State	A	A	A	C	B	A	C	D	B	C	B	C	B	A
Out	φ	φ	1	0	0	1	0	0	0	0	0	0	0	1

כאשר שני הפלטים הראשונים הם Φ כי כל עוד לא עברו שתי ייחידות זמן, לא נוכל לדעת מה הפלט של רכיב הזיכרון השני כי הקלט שלו מוגדר רק אחרי המחוור הראשון) שMOVIL פלט 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 כלל השינוי! נשים לב שהקלט בזמן המחוור הנוכחי ולא משנה כי מדובר במכונת Moore ולא Mealy.

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

דוגמא נתון המעגל הבא



כאשר הפלט היחיד הוא Out (והחיצים לעלה 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 נקבל את טבלת המצבים

		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

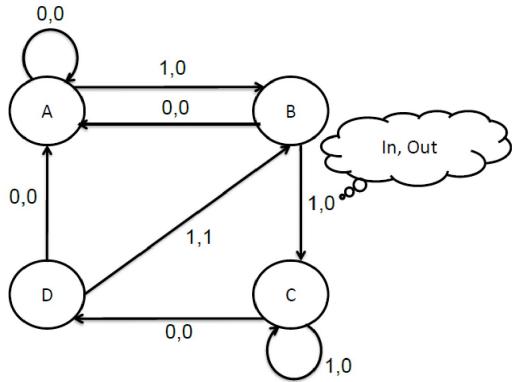
3. נבחר שמות למצבים (קיובע פלטי הזיכרון) כרגיל

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

ועתה באמצעות הצבה של השמות בטבלת המצבים נקבל כאשר הפעם הקלט 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 נctrוך עכשו להשתמש באוטומט Mealy, כלומר שעל הקשת כתוב איזה קלט מעביר אותו למצב הבא ואיזה פלט הוא מספק לנו



5. עתה נרשום סדרת בוון שטראה אותו הדבר, רק שהפעם בחיפוש אחר כל השינויים נוצרק 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 מתקיים רק כאשר הקלט בזמןים $t, \dots, t+3$ היה .1, 1, 0, 1

סינטזה של מעגלים סינכרוניים

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

1. בניית אוטומט בהתאם לדרישות.

2. טבלת מצבים.

3. קידוד מצבים ובחירה סוג FF.

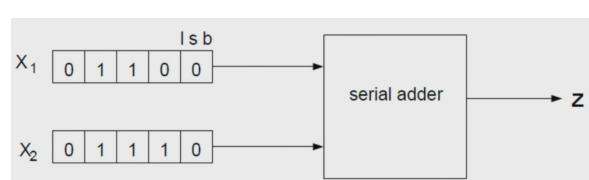
4. טבלת מעברים ופלט.

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

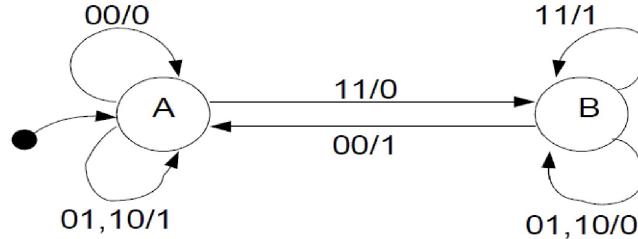
6. בניית המעגל.

דוגמה נרצה לבנות מסכם ביןاري טורי, כלומר מעגל שמקבל קלטיהם x_1, x_2 וזמן t_i מחשב את הבית ה- i (מכיוון ה-LSB) בסכימת המספרים המתואימים על הסרט הנע של x_2 , x_1 (כל מהזור מקבלים שני ביטים חדשים, שכאילו מתואספים כ-MSB במספרים שאנו חנו סוכמים).

ראו איור (כל פעם סוכמים את הביטים המתאימים, כמוון עם נושא מהסכימות הקודומות)



1. ראשית נבנה אוטומט שמקיים את הדרישות, כאשר הקלט (מיון ל- $/$) הוא שני הביטים m_1x_1 ו- x_2 בהתאם. נבחר שה מצבים שלו יציגו האם יש לנו נשא מה חישוב הקודם, וכך נשמר את המידע הזה בזיכרון למשעה. A מייצג מצב שאין בו נשא מה חישוב הקודם ו- B מייצג מצב שבו יש נשא מה חישוב הקודם. בכל פעם נחשב את הסכימה יחד עם הנשא (אם יש כזה), ונבחר מה הפלט של התוצאה ובנוסף האם יש לנו נשא ונויבור מצב בהתאם.



2. נבנה טבלת מצבים, שמכילה 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 בטבלת קרנו!

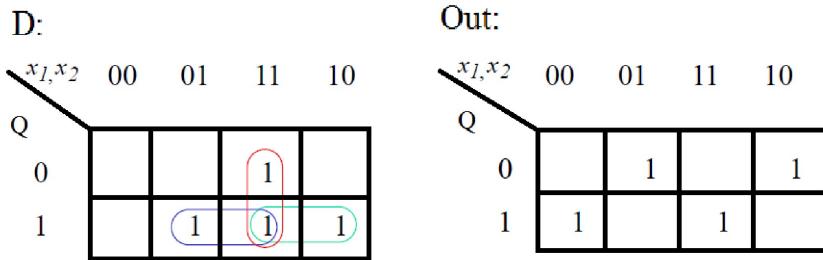
3. קידוד המ מצבים דורש בחירת ייצוג בינארי למצבים, כאשר יש שניים זה די קל - מספיק בית אחד שייצג את A כשהוא 0 ואת B כשהוא 1.

4. נחליף את A ו- B בטבלה בביטויים המתאימים להם ונקבלת טבלה מעברים ופלט שמכילה 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

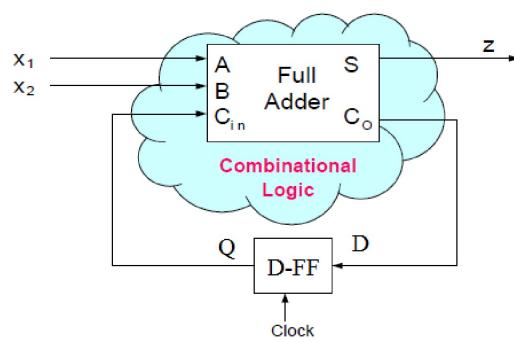
5. נפצל את הטעלה לשתי מפות קרנו (אחת ל- D , הקלט ל- DFF שיחזק את המצב הבא ואחת לפולט) ונכשה את הטעלה כמו שעשינו

בתרגול 2



כך שנתקבל בסופו של דבר דבר זה בדיקת הפלטים של FA (באופן שדי מתאים לאפיון) נקבל שימוש את המעגל

כז



שבוע 7 | מעבד

הרצאה

הערה למה שלא נשמר את כל הרגיסטרים לצד הקוראת או הנקראת? כי זה מאד בזבזני בין היתר כי לא תמיד נדרש את כל הרגיסטרים שמורים. מה שעדייף הוא שככל צד ישמר חלק מהרגיסטרים שלו אצלנו וישמר רגיסטרים אחרים בשביל الآخر. בדומה כזו לא נדרש תמיד לשמר הכל, אלא רק מה שאנו צריכים לשמר (לדוגמה אם הקוראת השתמשה ב- $\$z$ ולא צריכה אותה יותר אחרי הקריאה לפ', היה לא נדרש לשמר אותה עצמה).

הערה ישנן פקודות שלא קיימות ב-MIPS אבל שנויות למימוש באמצעות פקודה אחת שכן קיימת, לדוגמה `not $t, $d`, `not $t, $s` ו-`not $t, $0` והאסמבלר יוכל לתרגם את המקרים הפרטיים האלה.

קידוד פקודות MIPS

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

- פקודות עם שני רגיסטרים כאופrndים מקודדת 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 אינדקסי הרגיסטרים של האופרנדים ו-rd אינדקס רגיסטר היעד (5 ביטים כל אחד) ; shamt מספר הביטים ל-left shift כשמדבר בפקודה shift (5 ביטים) ; funct ו- 64 שגדיר איזו פעולה בדיק נבע (לדוגמה ל-add ו-sub יהיה כאן ערכים שונים).топס 6 ביטים כך שיש לנו לכל היותר פקודות מסווג R. נשים לב שככל פקודה תופסת הרבה מילימ (5 ביטים לכל אינדקס רגיסטר, כולל 32,000 מילימ שמייצגות פקודות סכימה כלשהי).

- פקודות עם רגיסטרים ו-immediates מקודדות ב-I-Type

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

כאשר כאן rs הוא אינדקס רגיסטר ה-operand והו ייחד עם ה-immediate שהוא 16 ביטים ו-rt הוא רגיסטר היעד. הרוב המכريع של opcodes משמש פקודות מסווג זה.

- פקודות קפיצה מקודדת ב-J-Type

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

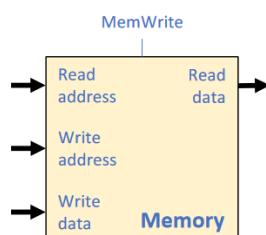
כדי שנוכל לפזר למגוון כתובות בזיכרון (2^{26} כתובות), כאשר רקjal מקודדות כך (שאר הקפיצות כוללות immediate)

הערה כתובות R-Type הן כי זולות, כי אפשר להכניס הרבה סוגים שונים של פקודות באמצעות שדות ה-funct ו-h-funct. אם נרצה עוד פקודה מסווג R-Type, נפנה מיד לפקודה מיותרת מסווג I-Type, שתופסת משמעותית יותר מקום בעבור פקודות אחרות, ונסב את ה-R-Type opcode שלא לצורך פקודות R-Type.

ביצוע פקודות ב-MIPS

מעבד הוא בסה"כ מכונת מצבים ענקית, כאשר ההבחנה הבורורה בין הלוגיקה למצב/זיכרון היא קצר יותר מרכיבת כי הפקודה רצתה איפשהו בפועל (לפני ואחרי מצבים). עם זאת, ביצוע פקודה בסופו של דבר מעביר אותנו מצב (משנה את ה-PC וכו'). זוכור hvordan אמנס מאורגן בבתים, אבל הקריאה ממנו היא ביחידות של ארבעה בתים (מילה), וכך גם PC גדול בקצבים של 4.

ברמת הארכיטקטורה הנוכחית, רכיב הזכרון מציג את הממשק הבא



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

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

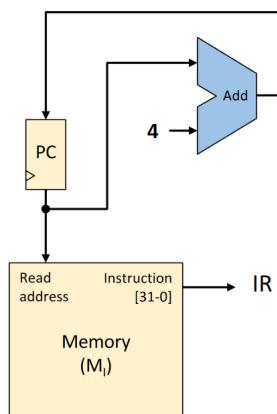
הערה כשנסמן חץ באופן הבא ← אל תוך רכיב ומתחתיו מספר, הכוונה שיש לנו סע שמורכב ממספר חוטים (מספר שכנו) ולא רק ביט אחד.

שלבי הרצת פקודה

- קראית הפקודה (fetch) : המעבד מספק כתובת (ששמורה ב-PC) לזכרון ומתקבלת תוכנה, שהיא קידוד הפקודה שצורך להריץ.
- 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 ואחרות אינקרמנט).
- ביצוע הפקודה (execute) : לבצע את הפעולה המתמטית.
- גישה לזכרון (memory access) : נדרש כשהפעולה ניגשת לזכרון `load` יקרה ו-`store` יכתב.
- כתיבה חוזרת (write back) : לכתוב את תוכנות החישוב או ה-`load` או ה-`branch`. השלב הזה למעשה משתנה בהתאם למצב המכונה לקרויה הפקודה הבאה.

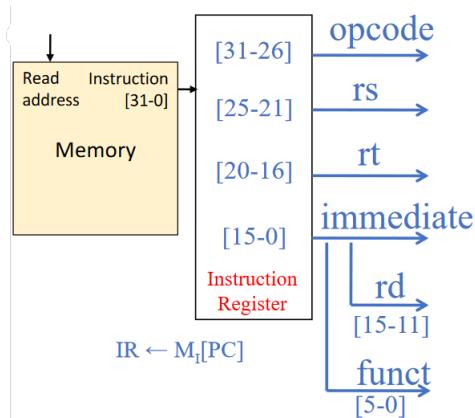
מימוש השלבים

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



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

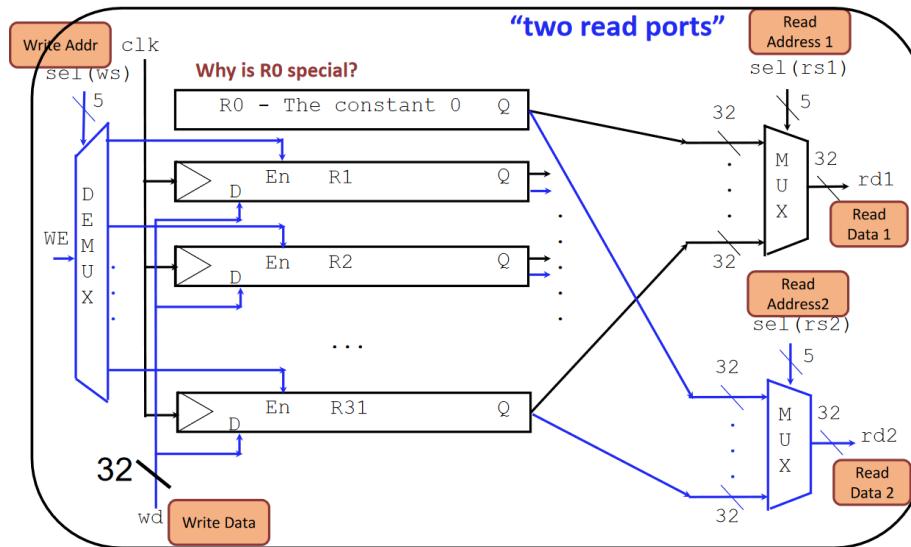
- פענוח: נפצל את הביטים ברегистר הפקודה (IR) לשימושיות הרגולונטיות שלם (ראו אייר)



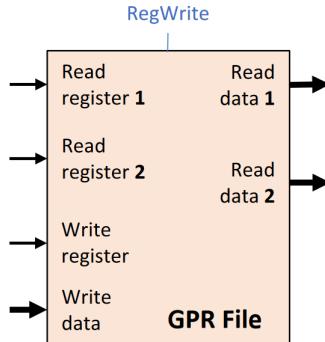
ובנוסף נביא את הרגיסטרים הנדרשים לפעולה, שממוקמים ב-FF, Register File. ים שבו הרגיסטר הראשון הוא בעצם קבוע 0 ולא רכיב זיכרון וה-31 האחרים הם רכיבי זכרון אמיטיים. כדי לקרוא מהם מידע, צריך להשתמש ב-Mux שבודח אחד מבין 32 רגיסטרים ומוציא את התוצאה החוצה. הכתובת שנקרה תלואה כמובן ב-rt, rs ו-rd לפי הצורך. ה-Mux הוא בעצם מערך של 32 Mux-ים שככל אחד בורר בין 32 אפשרויות.

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

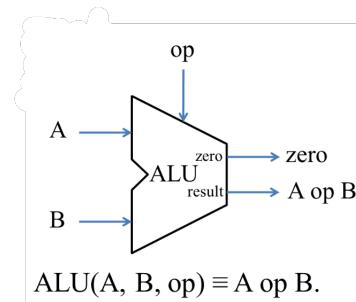
כדי לכתחזק בזיכרון לרוגיסטרים (בשלב ה-DeMux) נחבר את ה-FF-ים ל-DeMux שnbrור אליו את הרגיסטר הנוכחי אליו אנחנו רוצים לכתחזק אליו באמצעות בית WriteEnable. האיסולטרציה הבאה מדגימה את כל האמור.



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

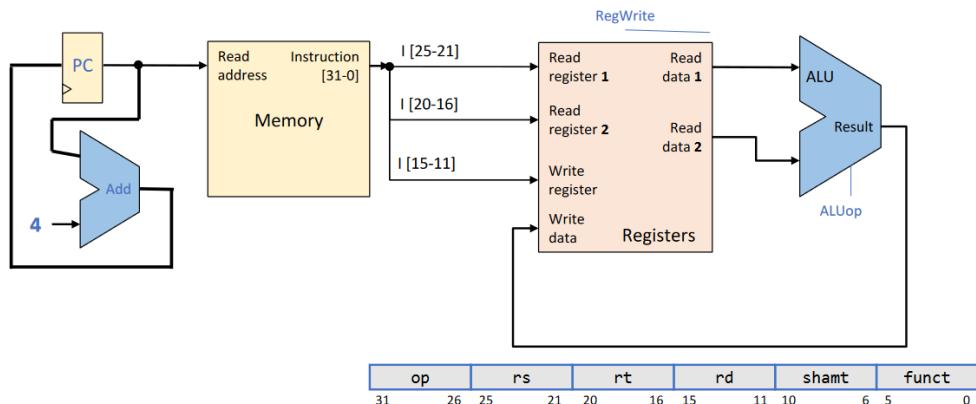


• ביצוע: הרכיב האחראי על החישוב האריתמטי נקרא Arithmetic Logic Unit (ALU), שמציג את הממשק הבא



כאשר הפלט zero הוא בית (דיל) שערכו 1 אם פلت החישוב הוא 0 (שימושי מאד לкопיצות מוגנות שוויון/אי-שוויון).

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



הערה המוקם היחיד שבו נדרש לעליית שעון כדי להתקדם הוא ב-ALU, שכן עד לחישוב התוצאה ב-ALU אין שום לוגיקה שאינה צירופית, וכי לכתוב את המידע לרגיסטרים צריך עוד עליית שעון. לעומת זאת, מחזור אחד לכל פקודה (עד כדי setup time ו-hold time), מכאן בא השם Single Cycle (time).

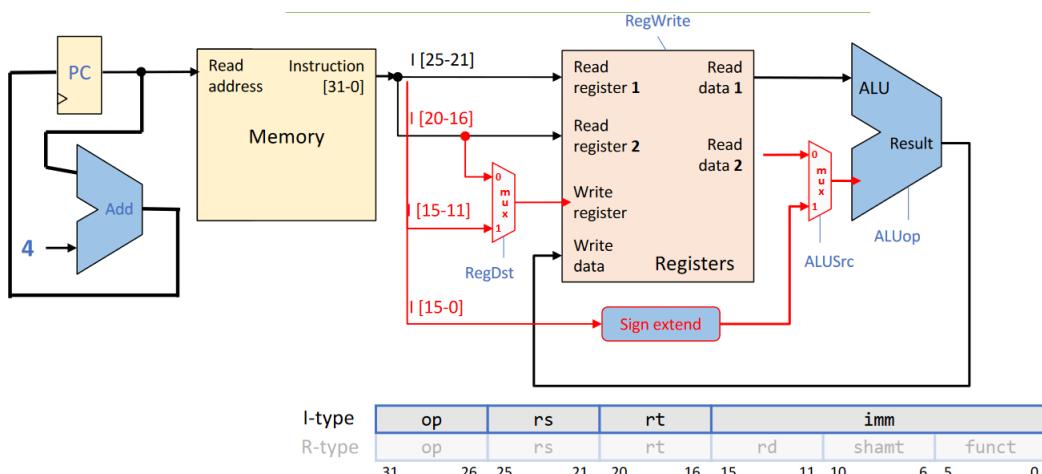
כדי לממש פקודת I-Type, נבצע את אותה הפעולה של R-Type עם כמה שינויים קטנים:

1. ניקח את תוכן ה-mm imm מהפקודה ונעשה לה sign extend (הרחבת המספר מ-16 ביטים ל-32 ביטים כפי שראינו בעבר) וכן נctrיך לקרוא רק מכתובת אחת ב-bus Register File (ספק זבל ל-bus כתובת הקריאה השניה). את הבחירה במה לשימוש ממש באמצעות Mux שבורר לפי סוג הפקודה.

2. כתובות הכתיבה צריכה AxM לפניה בדומה לנ"ל כי ב-A-Type-R-Type כותבים לרגיסטר rd (האינדקס השלישי בקידוד) ואילו ב-I-Type-I מדובר ברגיסטר rt.

הערה בגל של-sh-choffimm funct opcode יctrיך להגיד לנו חח"ע האם מדובר ב-A-Type-R-Type ו-I-Type-I כדי שנוכל לפרש את הביטים נכון.

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

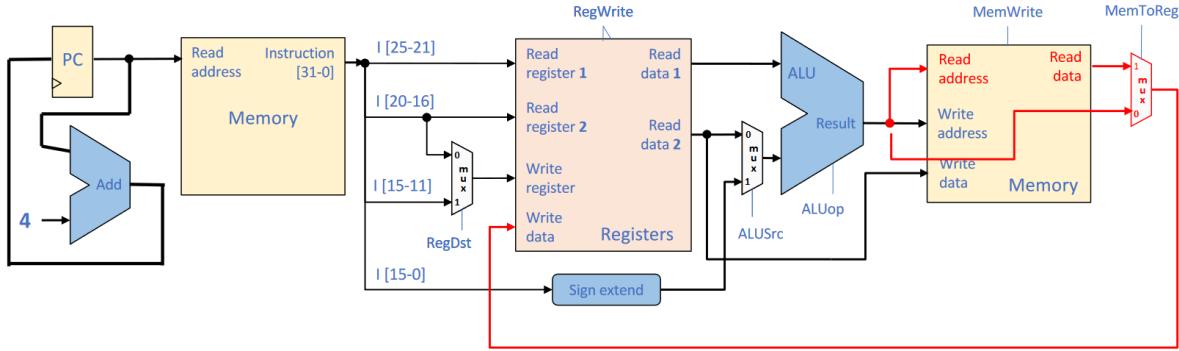


Reg [IR [rt]] : השלב הזה חיוני למימוש כתיבה וקריאה, כאשר בשתי הפקודות מעתיקים את הערך של memory access write back -]-[Reg [IR [rs]] + SignExtend (imm) WE=0 או להפוך, בהתאם.

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

בשביל הקריאה וגם הכתיבה ניתן ל-Data Memory את כתובת הקריאה והכתיבה שהיא פلت ה-ALU (ישיכום את הרגיסטר יחד עם ה-mm imm לכדי כתובת אחת). תוכן הכתיבה יהיה פשוט תוכן הרגיסטר השני שקראונו מהרגיסטרים. פلت הזכרון יבורר יחד עם פلت ה-ALU לפי האם אנחנו קוראים מהזיכרון (פלט הזכרון) או מבצעים חישוב מתמטי נתו (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 (35) כאשר \$w(\$5, 7(\$6) הוא opcode של קריאה מהזיכרון, rt , כלומר מקודדים את אינדקס הרגיסטר ממנו לקרוא ואליו כתוב ולבסוף ה- m וּמוּ שהוא היחסט מ- rs שנוטן את הכתובת.

הערה J-type מקודד את כתובות הפקודה ביחידות של מילה במקומות הביתי, ולכע אפע"פ שיש לנו 2^{26} ביטים, נוכל ליציג מרחב זיכרון בגודל $.2^{28}$.

דוגמה את השורה $A[6] = b - a$ נמשב באמצעות שתי פקודות: קריאה של $A[6]$ לרגיסטר זמני והשמדת ההפרש ברגיסטר נוסף.

דוגמה נביט בקוד הלולאה הבאה

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

אותו נוכל לכתוב כפסאודו-קוד אסמבלי

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

מכאן נקצת רגיסטר לכל אחד מה משתנים, לדוגמה $g = \$s1, h = \$s2, i = \$s3, j = \$s4$ וכתוות הבסיס של A נשמר ב- $\$s5$. לסיום נוכל לתרגם זאת לשפת אסמבלי אמיתי

```

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 $s1,$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 מראשייתו של המערך באמצעות הכפלת הרגיסטר ב-4 (כי הכתובות כשחן לא ב- imm הן ביחידות של בתים), נוסף אותו לבסיס של A ונטען את המילה מהזכרונו. משם סתם עושים אריתמטיקה עד לתנאי הלולאה שkopץ חזרה להתחלה רק אם i שונה מ- h לאחר השוואת הרגיסטרים המתאים להם (ne המשמעות $(Not Equal)$.

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

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

- פקודת addi שתחשב את ההפרש בין המשתנה למועד ההשוואה בסהగ
- פקודת bne שתבדוק האם ההפרש שונה מאפס ואם כן תקופץ לתוויות של הרגע הבא (אחרת לא נקפוץ ונשאר להריץ את תוכן הרגע הנוכחי).

לאחר סיום תוכן הרגע נוסף פקודת j לסוף ה-switch-case כולל, כך שלא נריץ בטעות הסגרים אחרים (בדומה לפקודת ה-break).

דוגמה את ההתניה if ($g < h$) goto label נוכל למש בקלות באופן הבא (בנחתה שהמשתנים הם ב- $\$s0, \$s1$ בהתאמה)

```

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

```

ההערות ליד חן הפענוח (מעבר לפסאודו-אסמבלאי)

```

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 עושה באמצעות טבלת מעקב לדוגמה, ונקבל שזה סוכם את כל המספרים האיזוגיים בין 1 ל-n.

דוגמה נמיר את הקוד הבא מ-C לאסמבלי MIPS כאשר הערכיהם 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 variable נשים ב-\$t0, ונתיק את ערכו של a ל-\$t1 ושם נבצע לו דיקרמןט. בולאליה נקבע לסוף אם \$t1 הוא 0 ואחרת נריץ את תוכנה.

לבסוף נכתוב את תוכנו של \$t0 = value למילה החמישית במערך. הקוד כולם 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

הרצאה

עד כה בנו מעבד שיכל להריץ כל פקודת I או R, כך שכל שנותר הוא קפיצות והתנויות. נמשב אופן הבא:

1. נקרא את הפקודה מזכרו הפקודות.
2. נקרא את הרגיסטרים t, rs מה-.register file
3. נחשב sign extend ונכפיל ב-4 (shift ב-2 שמאלה) את ההיסט אליו צריך לкопץ אם מתקיים התנאי.

4. נחשב את הכתובת הבא אם לא נקבע, ונקבע את הכתובת אליה אולי נקבע להיות $(PC + 4) + (offset \ll 2)$

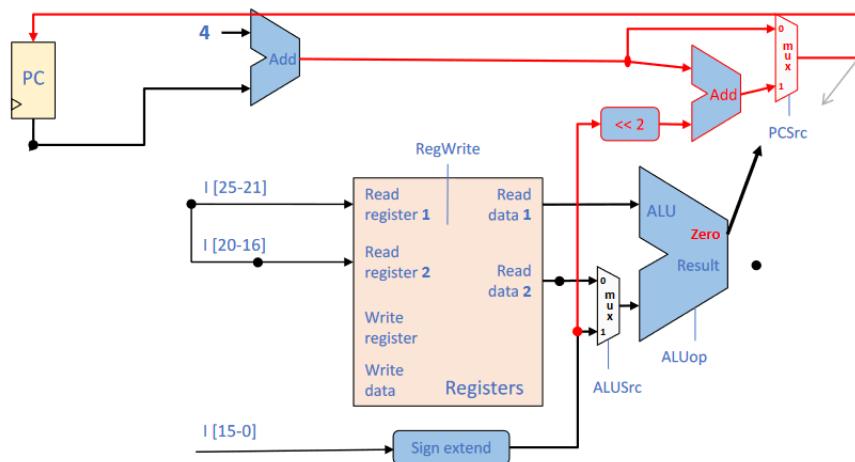
5. ה-ALU ישווה את הרגיסטרים לפי בהתאם לסוג ההשוואה.

6. PC יעדכן 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 החיוות החדש שנדרש, באדום ניתן לראות את הרכיבים שנוספו לנו כדי לספק את התיאור הנ"ל.

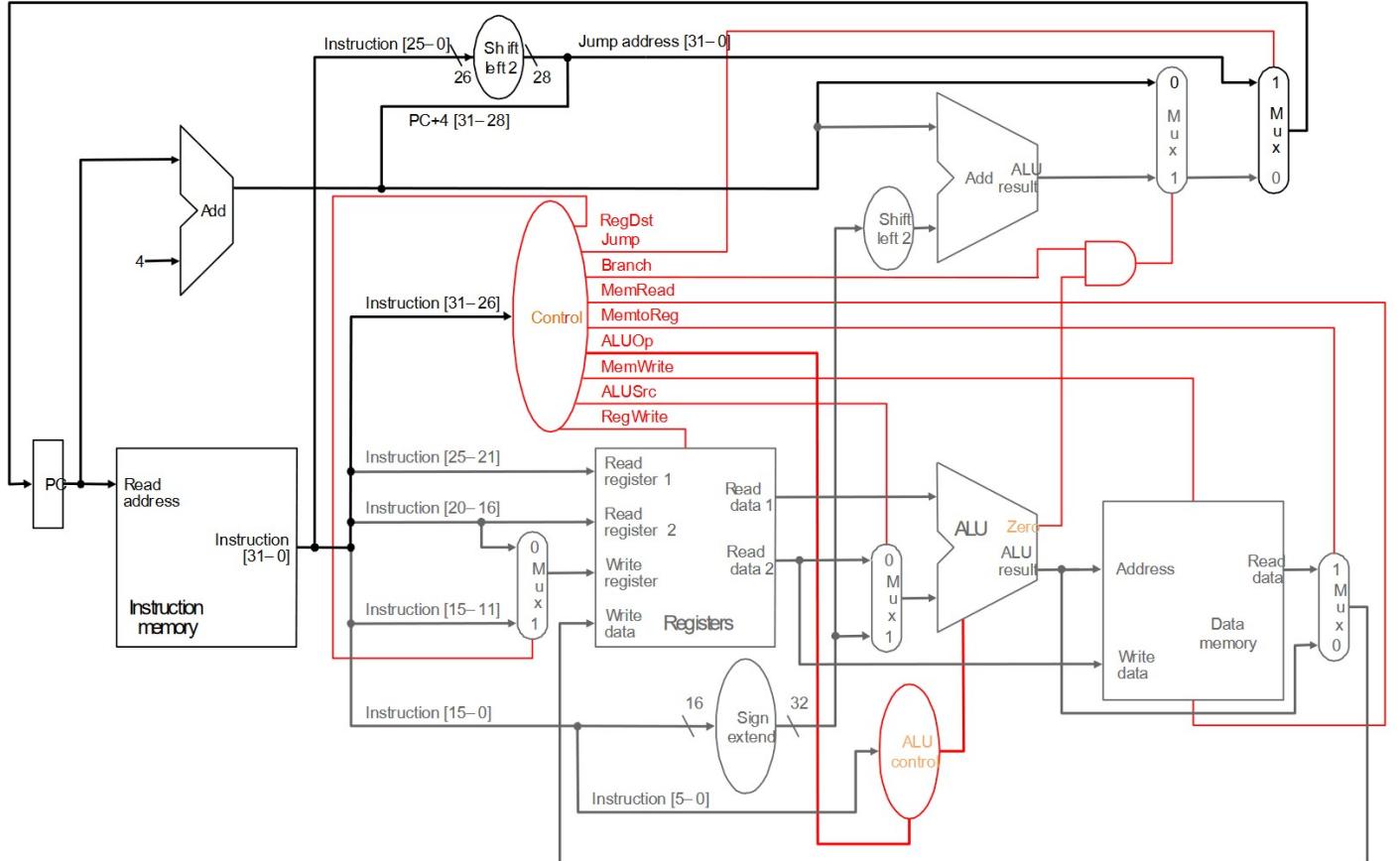


פקודות מסוג J-Type נמש בדומה, כאשר עתה נעדכן

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

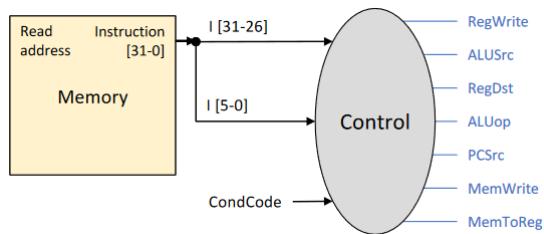
כasher ה-. היא פולת הצמדה (concatenation) של הפרש הכתובות בין כתובות הפקודה הנוכחיות לכתובת ה-label.

לסיכום המעבד השלים שלו נראה כך



יחידת השליטה

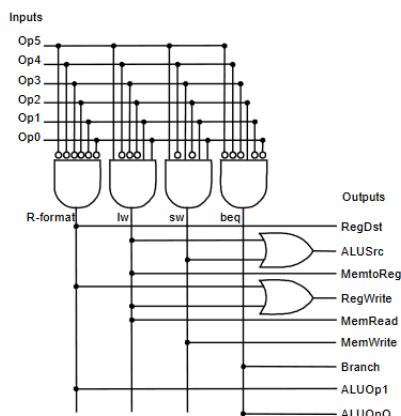
עד כה המעבד שלנו כולל כל מיני ביטים שימושיים כשלקוטורים ל-Mux-ים ורכיבים אחרים (מוסמנים בכחול ולא מגיעים משום מקום). את כולם אנחנו מחשבים על בסיס שלב הפענוח של הפקודה. היחידה שלוקחת את ה-opcode (ובמקרה של R-Type גם את ה-funct) ומחשבת את השודות השונות שלוליטים על הריצה נקרא **Control Unit**.



דוגמה להלן טבלת אמת שכוללת את ערכי השודות שמחשב ה-Control Unit עבור 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 Type-I, 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 per Instruction, שסופר את מספר המחזורים שנדרשים לביצוע פקודה.

דוגמה עבור 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

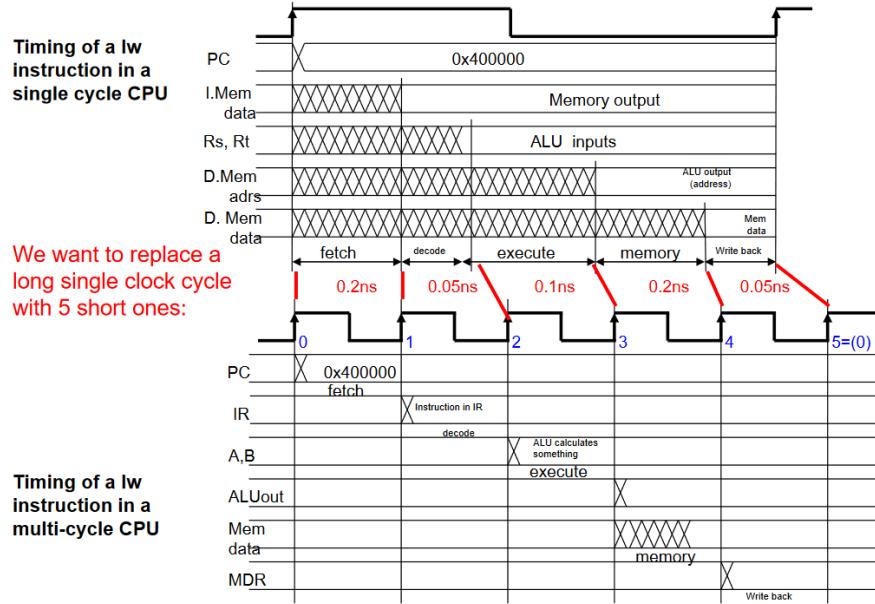
$$CPI = \frac{\#cycles}{IC} = \frac{\sum_i CPI_i \cdot IC_i}{IC} = \sum_i CPI_i \cdot \frac{IC_i}{IC} = \sum_i CPI_i \cdot F_i$$

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.

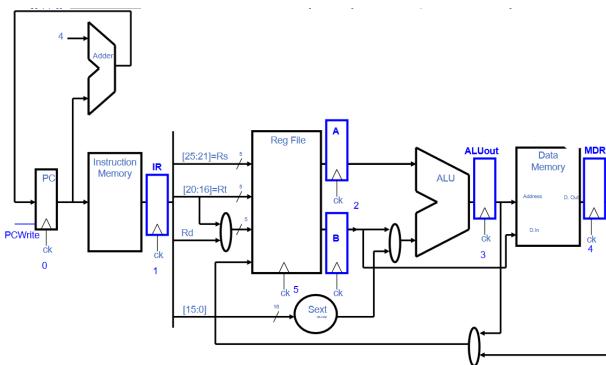


נשים לב שנכלל תדריות הרבה יותר גבוהה אבל מבחינה ביצועים עברו שלבים שאינם הארוכים ביותר, אנחנו משבזים זמן עכשו,

כלומר הרצת `lw` לוקחת יותר זמן. לעומת זאת, פקודות אחרות לא ידרשו את כל השלבים ולכן כן יהיה יותר קצרים.

כדי לאפשר את הרכיצה עם מספר שלבים, נצטרך לזכור מה קרה בשלב הקודם, וنعשה זאת באמצעות רגיסטרים שיחזיקו את תוכנת הפקודות

בכל שלב (ראו איור)

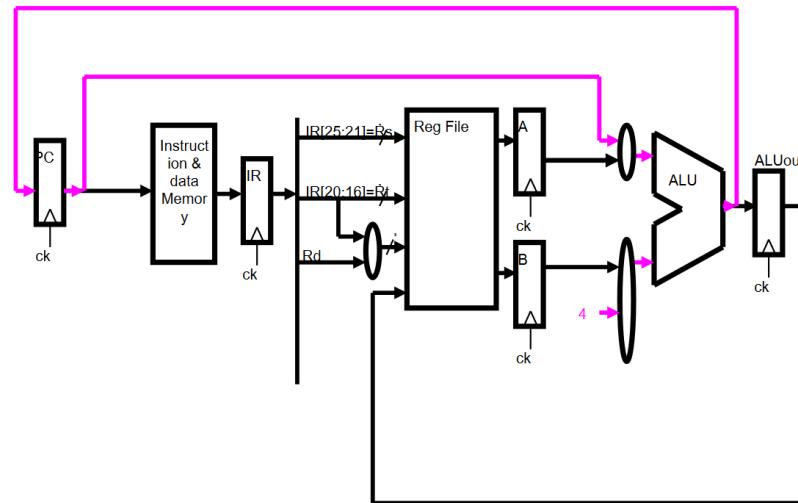


איך הגענו להפרדה זו? נביט במסלול של פקודת `lw`: מתחילה ב-`fetch` ואו קריאה מרגייסטרים, מעבר ב-`ALU` ולסיום כתיבה חוזרת לרגיסטר. לכן נפריד (1) בין ה-`fetch` לקריאה; (2) בין הקריאה לחישוב; (3) בין החישוב לכתיבה חוזרת לרגיסטרים; (4) ולפני כתיבה מהזיכרון לרגיסטרים.

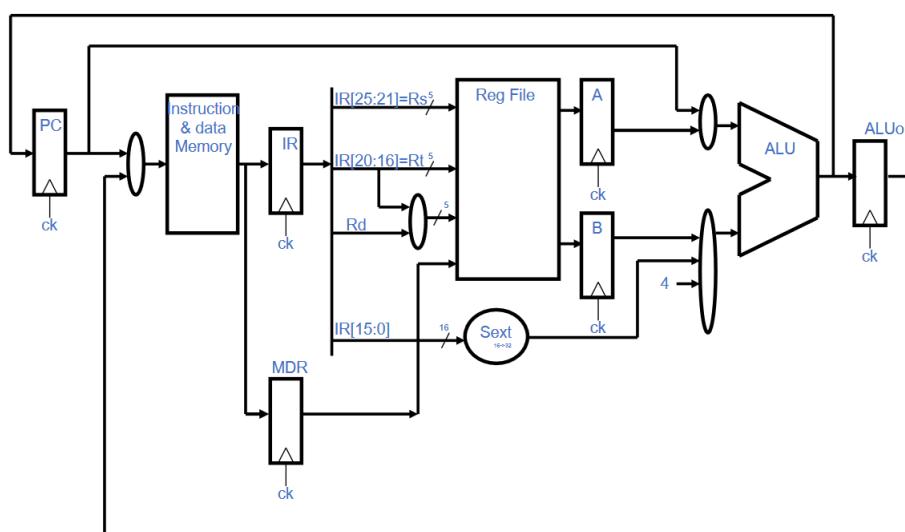
הערה כרגע חסירה לנו לוגיקה עדין שדואגת שהמעבד יבצע רק את השלב שצורך כרגע. כי אחרת ה-fetch קורה בכל מחזורי שעון ולא רק במחזוריים שימושיים ל-fetch.

איחוד ייחודי

- ה-fetch קורא מהזיכרון אבל גם עושה אינקרמנט ב-4 בכל פעם. במקום שהוא נפרד שדורש טרנזיסטורים, משתמש ב-ALU שבכל מקרה כרגע לא פעיל (ראו חוטים חדשים בסגול), זה יעזר לנו לקיצרו פעולות branch בעתיד.



- בשביל הבנת המעבד, הפרדנו את הזיכרון של הכתובות והدادטאות, אבל אין באמת סיבה לעשות זאת זה וזה מאוד יקר. נשים לב שברגע שאנחנו ב-multicycle, יכולים לא להשתמש בשתי ייחודות הזיכרון באותו זמן מחזורי, ולכן אפשר פשוט לאחד אותן עם אונט שמבעטת איזה מידע אנחנו רוצחים. עדין נפריד את התוצאה לרגיסטרים השונים כי נרצה לזכור מה המצב שלנו כמו שצריך ואם הכתובת והدادטאות היו באותו רגיסטר המצב לא היה מוגדר היטב.



מספר המחזוריים לפי סוגי פקודות

- R-Type : 4 מחזוריים (fetch, פענוח וקריאה מהרגיסטרים, חישוב ב-ALU וכתיבה חוזרת לרגיסטרים), במקומות חמיisha כי אנחנו לא קוראים מהזכרונו - חסכנו זמן!

- lw : כל 5 המוחזוריים (fetch, פענוח, קריאה מהרגיסטרים, חישוב ב-ALU, קריאה מהזכרונו וכתיבה חוזרת לרגיסטרים).

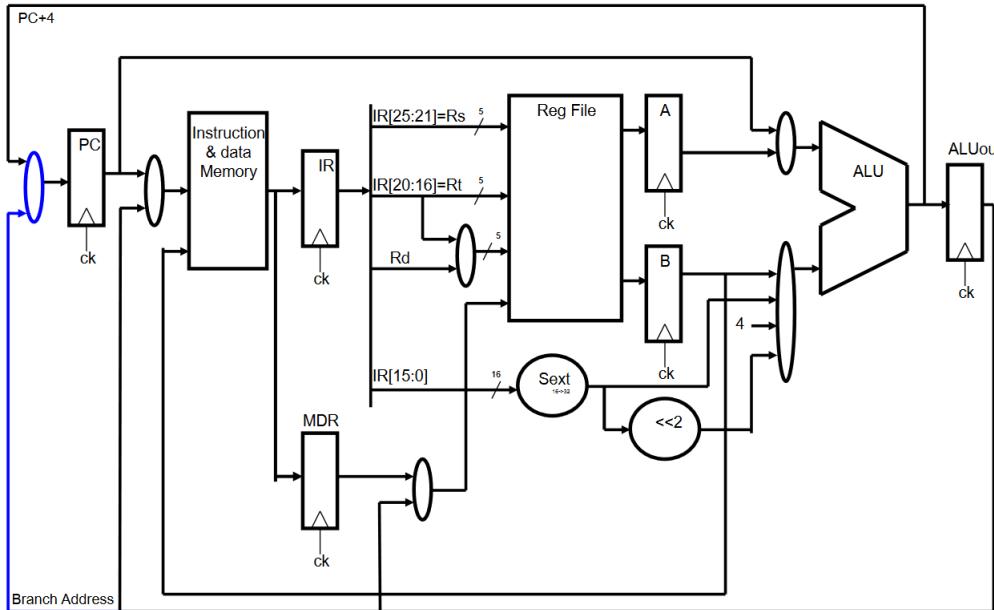
- sw : 4 מחזוריים (fetch, פענוח, קריאה מהרגיסטרים, חישוב ב-ALU וכתיבה לזכרונו), כאמור שוב במקומות חמיisha.

- beq : 4 מחזוריים (fetch, פענוח, קריאה מהרגיסטרים, חישוב התנאי ב-ALU באמצעות דגל ה-0, אולי חישוב הכתובת עם ההיסט במקורה של קפיצה וכתיבה ל-PC), כאשר חיבור ה-PC ל-ALU, כפי שראינו באיחוד היחידות, מאפשר לנו לחשב את ההיסט מ-PC.

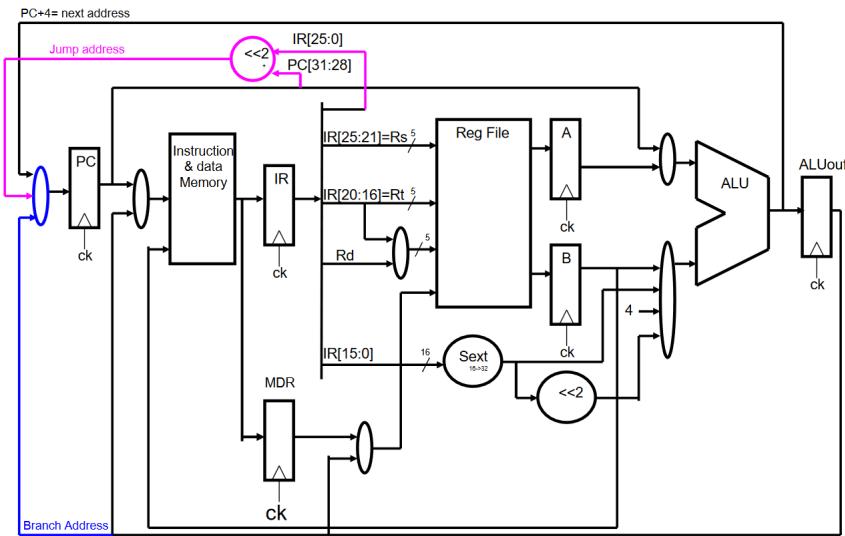
הבעיה כרגע היא שאם התנאי מתקיים זה לוקח שלב אחד יותר מאשר אם הוא לא, ונרצה שהו יקח מספר פעולות קבוע.

- לכן נוכל לחשב את ההיסט (שודרש רק את ה-mmמ של הפקודה) בזמן הפענוח ונשמר את התוצאה ב-ALUout (שבו כרגע אף אחד לא משתמש), ואנו 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 ירדנו ל-3 מחזוריים (fetch, פענוח וחישוב ההיסט וחישוב ב-ALU וכתיבה ל-PC).

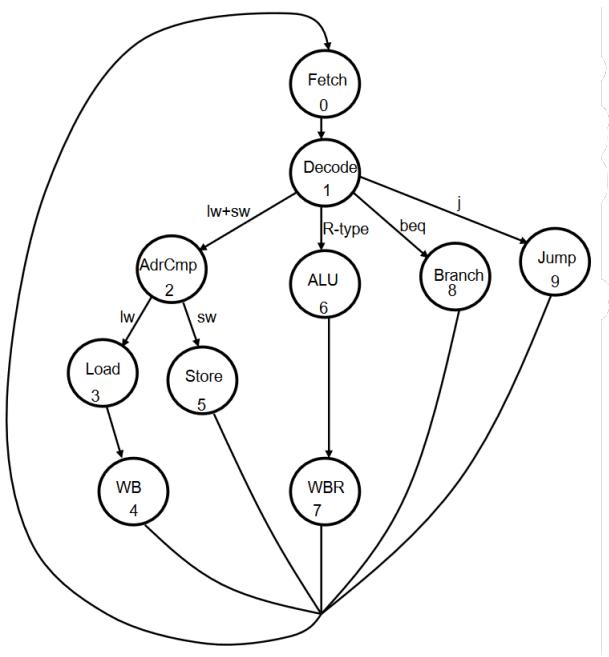
הסטודנטית המשקיפה תסתכל על המעבד החדש שנבנה (באיור) ותוווכח שאנו נדרשים רק שלושה מחזוריים כדי לבצע את beq.



- J-Type : 2 מחזוריים (fetch ופענוח וקפיצה) כאשר בין הפענוח לחישוב הערך שיכנס ל-PC במחזור הבא אין אף Flip-Flop ולבן זה קורה באותו המוחזור.



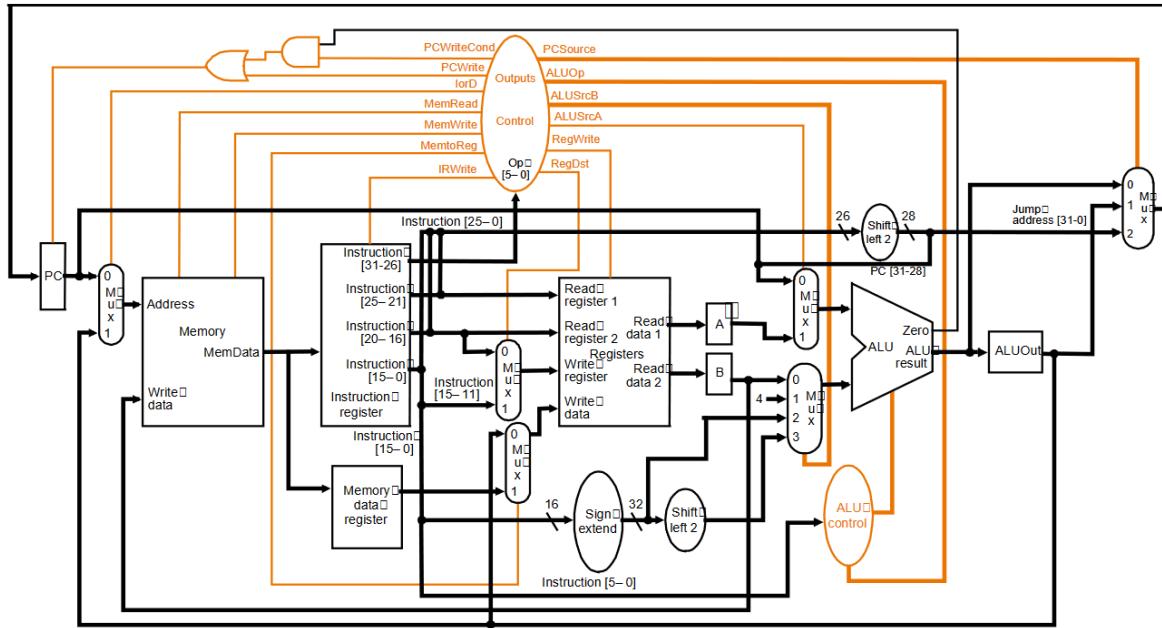
במהלך ניתוח כל אחת מהפקודות, בינוו בצד מכונת מצבים שמייצגת את המעבד, כי בסופו של דבר בעת מימוש המעבד באמצעות תהליכי הסינטזה של מודולנו צריך לבנות מכונת מצבים. להלן הטללה, שתואמת בדיק לניתוח הנ"ל של כל אחד מהשלבים.



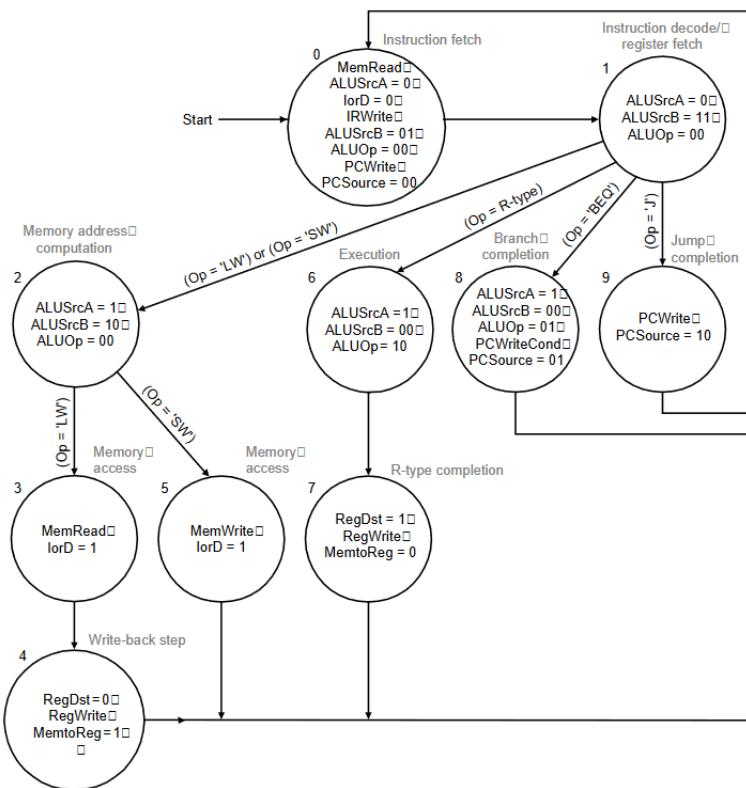
הרכיב היחיד שעוד לא התייחסנו אליו זה הקונט롤, שדווגע לכל המוקדים ולשליטה על השלבים השונים, והוא מוטמע במעבד הקיים (כמוון בלי הפרטים המלאים בתחום הרכיב) באופן הבא.

נשים לב שנוסף לנו בית חדש PCWriteCond שמחליט האם אנחנו כרגע 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 שקובע האם מעודכנים oczywiście את תוכן רגיסטר ה-IR, שמכיל את הפקודת שכותבתה הופיע בזמן השעון הקודם ב-PC. אם הביט הזה דлок, פירוש הדבר שאנחנו צריכים לעשות בשלב decode ה-PC.

כדי שיחידת הבדיקה תדע באיזה שלב היא נמצאת כרגע ומה השלב הבא, היא צריכה שייהיה לה איזשחו זיכרון פנימי (כמה DFF-ים) שמשמשים את הלוגיקה של אוטומט המצביע ה-IR, אבל אנחנו נתעלם ממנו.



לדגלים אפשריים ולכן אם המושגים הם דלוקים ; מה שלא מסומן לא השתנה מהשלב הקודם ; הריבועים הלבנים חסרי משענות

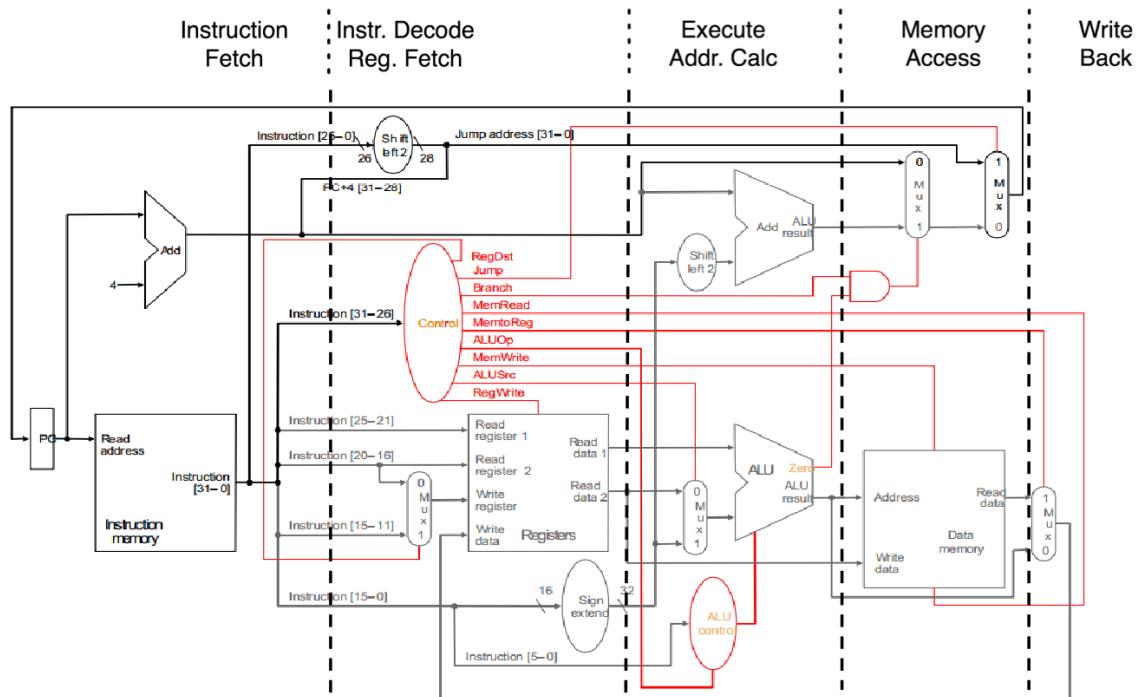


לסייעם כל שלב ומה הוא עושה, ללא מידע חדש

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	$\text{Reg}[IR[15-11]] = ALUOut$	Load: $MDR = \text{Memory}[\text{ALUOut}]$ or 5 Store: $\text{Memory}[\text{ALUOut}] = B$		
Memory read completion		4 Load: $\text{Reg}[IR[20-16]] = MDR$		

תרגול

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



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

דגלי יחידת הבקרה

ALUop : 2 ביטים המצביעים את קוד הפעולה שכרגע נבצע (משמשים את ה-ALU Control להחלה 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 MemRead 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 שנ;br/>שנצרך לעשות הוא באינדקס הרגיסטר אליו אנחנו כותבים. כרגע אפשר לכתוב רק לרגיסטר שהאינדקס שלו מופיע בפקודה (חמשת הביטים ב-R-Type ו-I-Type). עם זאת עצה נרצה במקרה שמדובר ב-opcodejal, לקובע את אינדקס הרגיסטר אליו נכתב (\$ra) בלי שיופיע בפקודה (כי הפקודה מכילה opcode וההיסט 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).

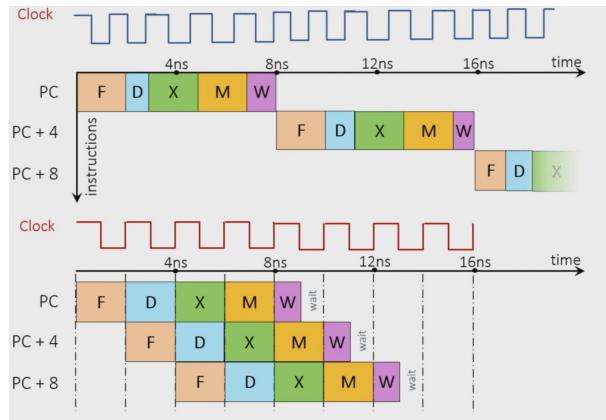
שבוע VII | מעבד pipelined

הרצאה

עד כה הצליחנו להוריד את זמן המחזורי פי 4, אבל גם העלנו את מספר פקודות להוראה פי 4, כך שלא הרוויחנו יותר מדי. לשם כך נהפוך את המעבד ל-pipelined, כלומר במקום שנרים פקודה אחת דרך המעבד בכל רגע נתון, נרים 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).

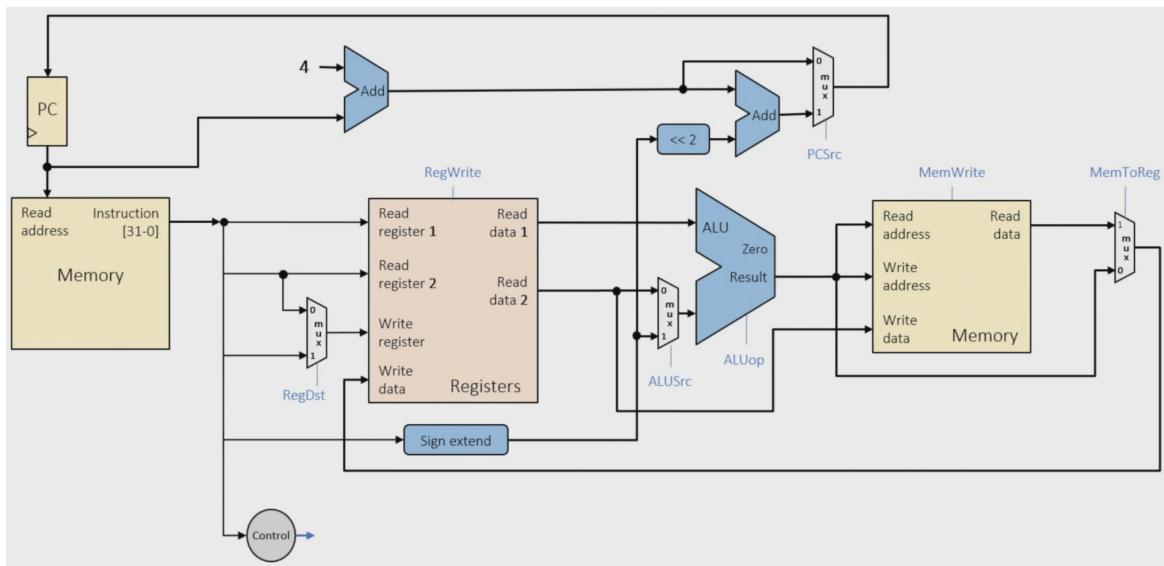
דוגמה במקרה של מעבדים, נשווה בין Multi Cycle (למעלה) ל-pipelined (למטה). ברור שעדיף Pipelined, ובפרט זמן החישוב שלו לינארי עם מקדם כמעט 1 (מגלמים את הפוקודות בקצבות) ביחס למספר הפוקודות, בעוד Multi-Cycle הוא לינארי עם מקדם יותר מ-4 ביחס למספר הפוקודות.



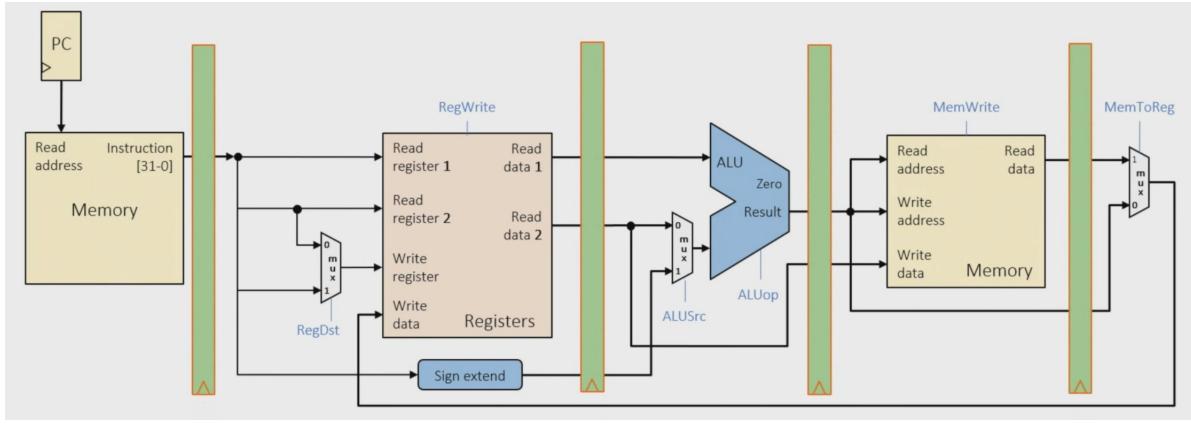
תחת הנתונים הנ"ל, ל-MC יש IPC של $\frac{1}{4}$ ותדריות של $500MHz$ ולכן $IPS = IPC * f = 0.25 * 500 = 125$ $\frac{1}{2ns} = 500MHz$ ואילוIPC של 1 (בלי הקצוות) ואותה תדריות של $500MHz$ pipelinedspeedup של 4.

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

מעבד ה-MIPS במחזור אחד נראה如下:



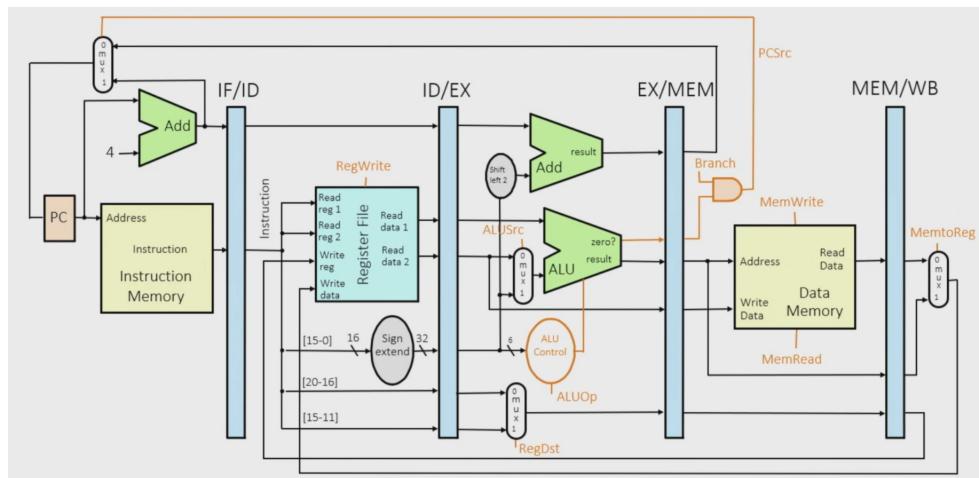
אם נתעלם ליעשו מהקונטROL ומה-branch-ים, יוכל לחלק את התהילה לשלבים השונים של ה-pipeline (הקיים הירוקים הם רגיסטרים ששומרים מצב לאחר חישוב השלב)



כלומר השלבים שלנו הם write back (5) ; memory access (4) ; ALU (3) ; decode (2) ; fetch (1). נשים לב שההבדל בין השלבים הוא רק בזמן הפעולה. השלב fetch מזמן הפעולה הראשון ועד השני מזמן הפעולה השני ועד השלישי. השלב write back מזמן הפעולה השלישי ועד השלב הראשון של הפעולה הבאה.

לרגיסטרי המצביע יש שמות מקובלים (משמעותם לימי) : (1) Fetch latch (IF/ID) ; (2) PC ושר המידע נשמר שם ; (3) EX/MEM (בהתאמה) ; (4) MEM/WB ; (5) ID/EX (בין ה- decode וה- fetch).

דוגמה להלן אירור בסיסי של מעבד ה-**MIPS** עם pipeline, שימושו בהזגמת הרצף הפוקודה (**lw \$1, 30(\$2)**)



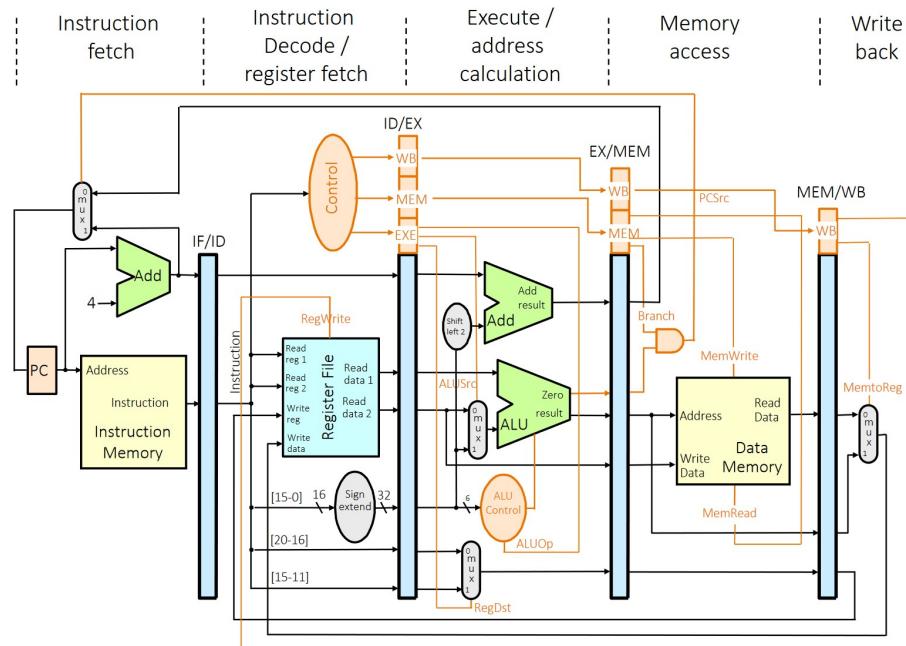
נפרט את השלבים שתעביר הפקודה במעבד:

- לאחר פקודת-h-ID, fetch-ID, IF-ID יחזיק את ההוראה (כלומר מה שבעבר נשמר ב-IR) ובנוסף את הכתובת ממנה נקראת (PC+4), זו נדרשת בשביל branch-ים בהמשך.
 - לאחר פקודת-h-ID/EX-ID, decode-ID, sign extend-ID יחזיק את PC+4 בדומה לנ"ל, את תוכן הרегистר \$2, את ה-immediate-\$2 ואת ה-30 לאחר פקודת-h-ID/EX-ID.
 - לאחר פקודת-h-EX/MEM-ID יחזיק את תוכן \$2 סכום עם 30, ואת אינדקס \$.1 ואת האינדקס של רגיסטר \$.1.
 - לאחר פקודת-h-MEM/WB-ID, memory access-ID יחזיק את המידע מהכתובת שבקישנו 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 ניצלנו 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-ב-ID/ex 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 מהלאה. 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:

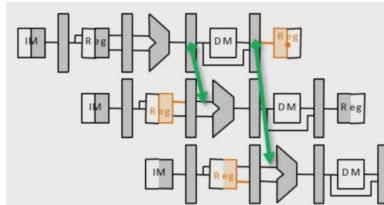
- **סכנות מבניות:** החומרה לא יכולה לתמוך בשילוב פקודות (לדוגמה נctruck זכרו שונה ל-PC והדאטא כי אחרת ניתקע במצב ש策יך לכתוב בו זמן לתשתיות כתובות מאותו זיכרו).
- **סכנות דאטה:** הוראות שמתבססות על פקודות שעוד לא חושבו pipeline (לדוגמה שימוש ברגיסטר שתוכנו אמרור היה להתמלא כבר בתוצאתה של פקודת סכימה שעוד לא הסתיימה).
- **סכנות בקרה:** צריך לבצע קפיצה אבל תנאי הקפיצה עוד לא חשוב (לדוגמה בקפיצה מותנת שווין 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 (שווין אינדקסים) באותו זמן מחוזר כך שה-DFF-ים יעדכו את יציאותיהם רק במחוזר הבא, ניקח את הביטים מחוץ 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 שכבר מאמץ מחוזר 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 מחזוריים ביןיהם כדי שההוראה הקודמת תסתיים לפני שהבאמה מתחילה, וכך ניצור את המרווח הדרוש. הפתרון הזה די בזבוני (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 שההווכן יכתוב לרגיסטר.

פתרונות עדיף נקרא Forwarding (עוד סוג של bypass) כלומר שההוראה הבאה תקרה שירות מה-ALU את תוצאת החישוב ולא תחכה לשני השלבים הנוספים של ההוראה הקודמת (MA ו-WB). ראו אייר המדגים זאת (החזים הירוקים הם bypass, שמעביר את תוצאת ה-ALU למקומות הנדרשים בשלב decode).



המשמעות בפועל הוא דיאלוג control-stateful: זכרו את שתי הפקודות הקודמות ולאן הן כתובות, וכך ידע האם יש צורך ב-forwarding. אם יש צורך, שלב ה-execute יקח החלטה באמצעות mux) ערך של אחד או שניים מהרגיסטרים מפלט ה-EX/MEM-ב-RegWrite. האחרונה תפלוט את הערך הרלוונטי לפיה קולטם שהוא מתקבל מתוך חישוב והביט forwarding unit (הפקודה הקודמת) ותוציאת החישוב והביט RegWrite ב-WB/MEM (הפקודה שלפני הקודמת).

ברמת הולוגיקה, אם `RegWrite=1` וגם אינדקס הרגיסטר אליו כותבים בפקודה קדומה זהה לאינדקס רегистר שקוראים ממנו, נבחר את תוצאת ה-ALU על פניהם ה-`RF`, עם עדיפות לפקודות חדשות יותר (קודם מסתכלים על EX/MEM ורך אז על MEM/WB).

דוגמא נדגים מקרה שבו נדרש לבחור בין פקודות חדשות לישנות

add \$2, \$1, \$3
add \$2, \$2, \$4
add \$2, \$2, \$5

אחרת היו במצב של Write After Write, שקרה רק ב-**Out of Order Execution** בעקבות הוראות הכתובת המאוחרת. וכך נוצרה הטעיה.

- העתקת זכרו-לזכרן (סקנת דעתה): הבעייה עולה כshmatzim או לכתובות בזיכרין שזה עתה כתובנו אליה עם sw.

אפשר לפטור את הבעיה עם forwarding שלוקח את המילה שנכתבה בפקודה הקודמת (שנשמרת ב-WB/MEM) ומשתמש בה לצורכי קבלת של הקריאה מה-data memory. כך גם נמנעים מ-stall יקר.

- load-to-use (סכנת דאטה) : הבעה עולה כshmבעצים פעולה שמתבססת על תוצאת קרייה מ-w בפקודה קודמת.

את הפעולה הזו נפתח עם `stall` (או NOP), ככלומר נבדוק לפני ה-`execute` האם אין דקסק של רגיסטר שאנחנו צריכים בפקודה הבאה (ידע שיש לנו מ-ID/IF) זהה לאינדקס היעד של הפקודה הנוכחית, שהוא `Iw` (ידע שיש לנו מ-ID/EX). אם כן, נעכבר את ה-`pipeline`.

הערה את כל הסכנות נctruck לזהות כבר בשלב ה-*decode* וזהו השלב המוקדם ביותר שניתן לעשות בו את זה, כי רק אז אנחנו יודעים בכלל מה הפוקודה עשוה והאם נדרשים אמצעים מיוחדים כדי להבטיח נכונות. את הלוגיקה של בחירת האמצעי הנדרש למניעת סכנות .Hazard Detection Unit (bypass, stall)

הערה פתרון אלטרנטיבי לפני מנגנוני מניעת הסכנות היא כתיבת קוד יותר חכם ברמת הקומפיילר/בן אדם שכותב אסמבלי, כך שנכחיק פקודות מסווגות אחת מהשניה ונשים בכךין פקודות אחרות לצורכיהן לקרות 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 שבוណע לאן 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"ג שהן התחילו את מהלכן ב-pipeline. הכוונה בביטול היא לא בהכרח שלא מחשבים משחו ב-ALU אלא שלא כתבים שום דבר חוזרת (לא ל-PC, לא ל-DM ולא ל-RF).

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

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

השימוש של זה בפועל כموון 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 עושה) לא תשתנה. פקודות אלו יכולות להגיד לפני הקפיצה (כאלו שלא משפיעות על הת寧יות הקפיצה) או ממסלול ההתקנסות לאחר הפיצול. אם כל הפקודות תלויות, אפשר להוציא *copy*-ים במרקחה הכיגרוע.

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 מיותרים 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 התקיימים 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.

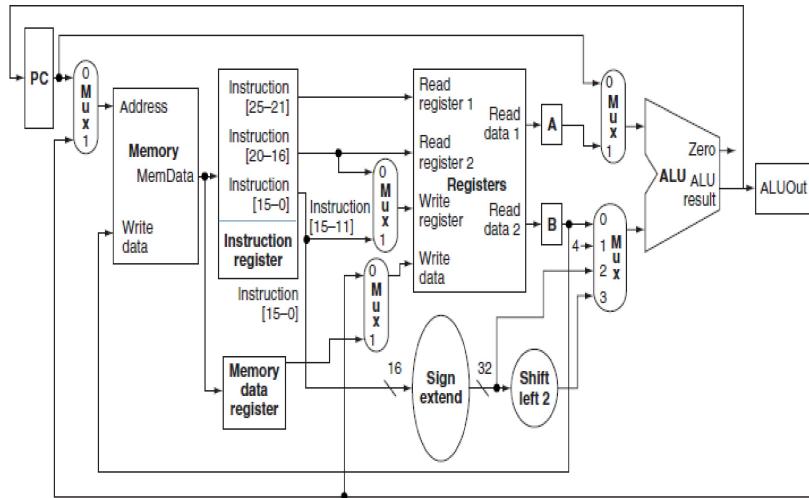
תרגול

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 תוצאתו היא קידום ה-PC ב-4 או חישוב בשלב ה-*branch* (ובמקרה של זומנים 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 יוציאו רוגיסטרים שיזכרו נתונים רלוונטיים מהשלב הקודם.
- שמירת מצב ביחידת הבקרה: נוסף DFF-ים בתוך יחידת הבקרה שישמרו באיזה מצב אנחנו כרגע.

אותות בקרה למרכיבים במעבד רב-מחזורי

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

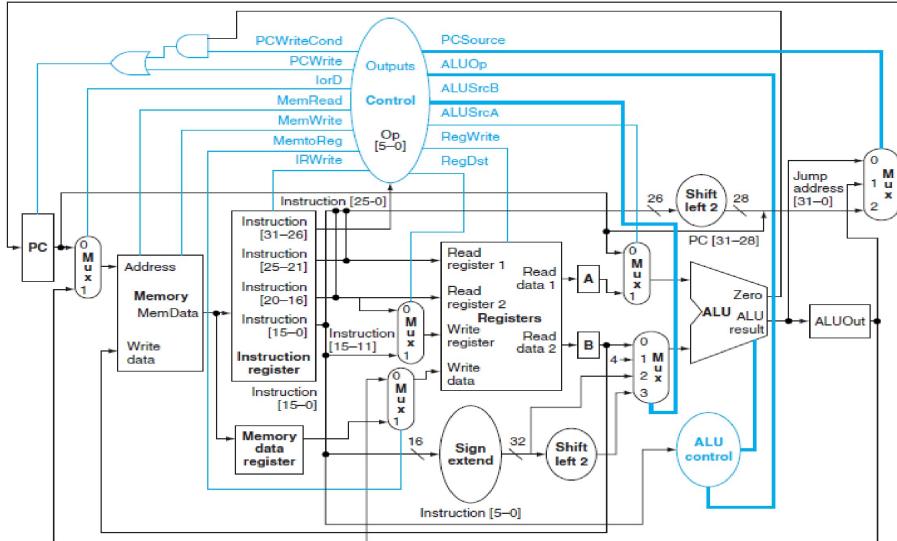


מעבר על כל ה-axut-ים ונסביר איזה בית מחובר אליהם ומה תפקידם, משמאל לימין (ושובר שווין לפי למעלה-למטה) :

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-(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 fetch, beq ו-R-Type, פקודות I-Type-arithmatic ו-sw/lw, וחישוב כתובות קפיצה, בהתאמה.

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 מהמחוזר הקודם במקרה).

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

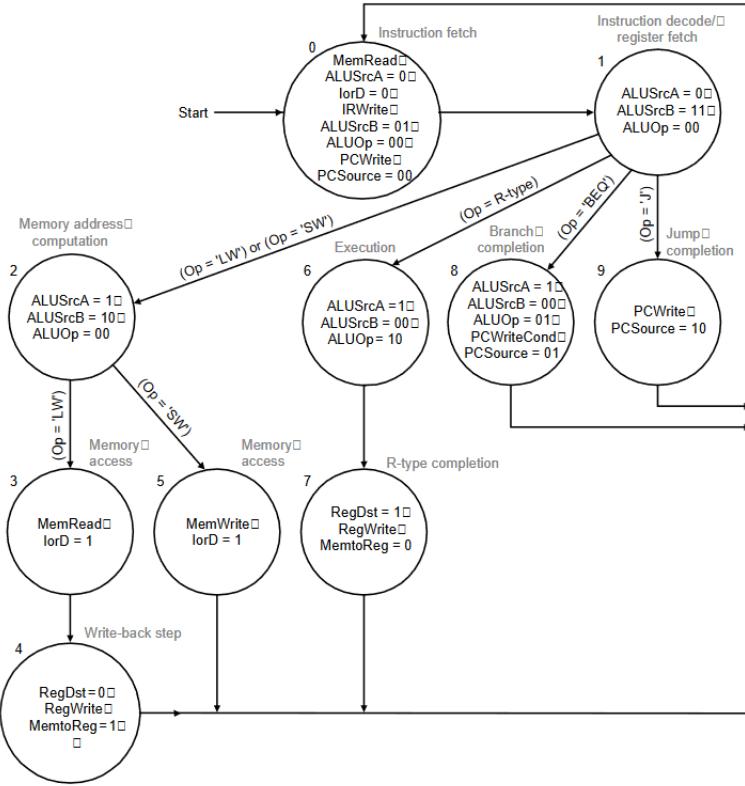
.2 . האם לכתוב ל-IR את פלט הקריאה מהזיכרון (דлок בשלב ה-decode).

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

.4 . האם לאפשר כתיבה ל-PC בהנחה שדגל zero של ה-ALU (דлок אם 'מ' מדובר בפקודת branch, או נקבע אם 'מ' הרגיסטרים שוויים).

דוגמה נסיף תמייה ל-addi. הוספה די דומה למקרה החד-מחזורי. עדיף לשנות כמה שיטור 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

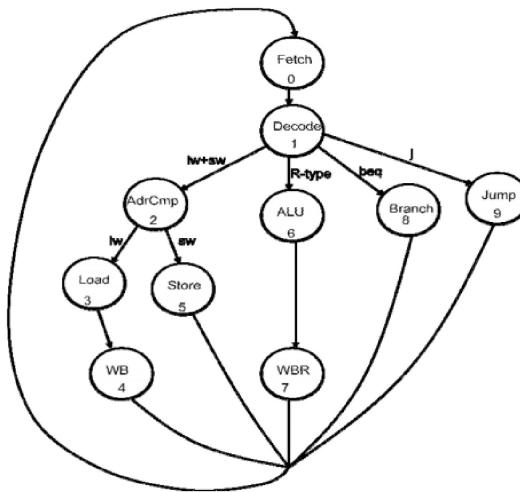
$$\text{Speedup}_{\text{overall}} = \frac{1}{1 - \text{Fraction}_{\text{enhanced}} + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

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

דוגמה שינו מעבד כך שעטה פקודות בנקודה צפה ירצו פי 2.5 יותר מה, גישה לזכור פי 3 יותר מהר 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 מאד 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 מנצח 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,

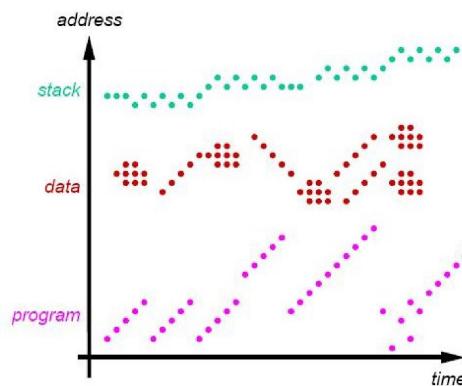
פעמיים :

- **локליות בזמן :**
- בדआא: 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.

• **локалиות במרחב :**

- בדआא: 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 למרחב של קריאת נתונים מהזיכרון, והירוק מייצג לוקליות 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 למטען 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 במטען בו.

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

- נקלט hit, כלומר שהשורה אכן נמצאת במטמון ונוכל ב מהירות רבה לחתה למעבד.
- נקלט miss, cache miss, כלומר לא נמצאת במטמון, ולכן נצטרך להביא מהזיכרון הרגיל אותה ולשים אותה במטמון, דבר שיקח זמן רב. בסוף נctrck אולץ לעשות evict (לגרש) שורה אחרת כדי לפנות מקום לשורה החדשה.

הערה אופן הגירוש נעשה באמצעות היררכיות בהן עוסקת בהמשך.

Fully Associative

מטמון מסווג זה יכול למפות כל שורה בזיכרון לכל כניסה במטמון. הכתובות המסופק בשאלתה למטמון מוחולקת (מגבוה לנמוך) למספר השורה, התג (26 בתים), וההיסטוריה (6 בתים). לכל כניסה במטמון יש-tag ו-bit וולדיות.

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

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

Memory:

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 6f 6b 20 68 65 72 20 73 69 73 74 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

נניח שיש לנו 64 בתיים במטמון FA, כלומר 4 שורות של 16 בתים כל אחת, ושאנו משתמשים במדיניות גירוש של LRU (קרי נגרש את השורה שבה השתמשנו לפני יותר זמן מכל השאר). בנוסף נניח שאנו עושים lw אחד אחרי השני לכתובות לפי דפוס הגישה הבא

008100E0:	77	68	61	74	20	69	73	20	74	68
008100F0:	6f	66	20	61	20	62	6f	6b	2c	
00810100:	67	68	74	20	41	6c	69	63	65	20
00810110:	75	74	20	70	69	63	74	75	72	65
00810120:	6f	6e	76	65	72	73	61	74	69	6f

LW (program order):

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

נשים לב קודם כל כי יש לנו 5 פיספושים הכרחיים (compulsory misses), כי הנתונים שלנו מופרסים על פני חמיש שורות ולכן כשייגש לפחות בית אחד מכל השורות הפעם הראשונה תמיד תהיה miss כי לא ראיינו את השורה לפני.

- בקראיה הראשונה קיבל miss, שכן נקרא את השורה הראשונה אל תוך המטמון ונחזיר למעבד.
- לאחר מכן בקראיה השנייה, קיבל גם פספוס, שכן נקרא את השורה החמישית ונחזיר למעבד.
- גם בקראיה השלישי והרביעית קיבל פספוסים. הקראיה החמישית היא מהשורה החמישית, אותה יש לנו כבר במטמון שכן להחזיר אותה, זה **hit**!
- הקראיה הששית היא לשורה הראשונה שעדיין במטמון, שכן זה עוד **hit**.
- הקראיה השביעית היא לשורה הרביעית, שעוד לא ראיינו, וכן נדרש להכנס אותה למטמון על חשבונו של השורה אחרת. השורה אותה נגרש היא העתקה נוספת שהשתמשנו בה, שהוא הבלוק (שורה) השני.

סה"כ היו לנו חמישה פיספושים, שתי פגעות ועוד פיספוס.

החסרון ב-FA הוא שבגלל שיש לנו בלוק בסדרי גודל של C^2 ,¹⁵ קיבל שחחיפוש הופך למרכיב דומיננטי בתזמון והוא גורם להחמרה בBITS (אפילו אם ממשים את חישוב $hit/miss$ עצ-OR-ים, עדין מדובר בערך באותה חזקה).

מטמון Direct Mapped

נרצה לחזוק את עצ-OR האורך שמחפש את הכתובת. עתה נסתכל על כתובות שאליתה באופן הבא: תג השורה (26 ביטים), אינדקס/סט (2 ביטים) והיסט (4 ביטים).

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

- **מערך הדאטה :** מערך עם כניסה כמספר הסטם ($^{2^2}$ במקרה שלנו), שבכל כניסה יש לו את השורה האחרונות ששמרה שיש לה אינדקס Caindex השורה.
- **מערך התגים :** מערך עם כניסה כמספר הסטם, שבכל כניסה יש את התג המתאים לשורה השמורה באינדקס המתאים במערך הדאטה, ובית וולדית.

דוגמה אם קיבלנו פספוס על הכתובת 10 0010 . . . 00, נקרא את השורה 0010 . . . 00 ונכתוב אותה לכינסה ה-10-ית של מערך הדאטה. באותו הזמן נכתוב לכינסה ה-10-ית של מערך התגים את התג של השורה הזו, 00 . . . 00.

הערה עתה נגרש שורות לא בגל שהטבלה כולה מלאה, אלא רק אם בכינסה המתאימה לסט שלנו יש כרגע ערך (ולידי) אחר. **דוגמה** נניח שאנו רוצים לקרוא באותו דפוס גישה כבוגמה על FA. הSTDVENTIT המשקיעת תפרק כל כתובות לבינאי ותאמת את נכונות הפירוט הבא :

- הקריאה הראשונה היא כMOVN פספוס, ויש לה ST 00, لكن נכניס את השורה לכינסה הראשונה (הAPSIT).
- הקריאה השנייה גם פספוס, וגם לה ST 00, لكن נדרוס את הערך הקודם ונחליף אותו בבלוק החדש שלנו.
- הקריאה השלישית גם היא פספוס, רק שעתה היא עם ST 01, אז לא נדרוס את הבלוק הקודם כי הכינסה שלנו ל-01 כרגע פנואה.
- הקריאה הרביעית, החמישית, והשישית הן גם פספוס כי דרשו כבר את הערכים שלhn משומשគולן עם ST 00.
- הקריאה השביעית גם היא פספוס אבל עם ST 11 (או לפחות לא נדרוס בлок קודם).

מתמון 2-Way Set Associative

ראינו ב-*direct mapped* שההנשויות בין סטם מאד מזיקות, לכן נרצה שלכל כתובות יהיו שני סטם (ways) אפשריים שבהם הם יכולים להיות. את הגירוש בין שתי ה-ways, ננהל לפי היוריסטיקת גירוש (כגון LRU). בגלל שסוג השאלות לא משתנה, אלא רק אופן המענה להן, פירוש הכתובות נשאר אותו הדבר זהה ב-*direct mapped*.

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

דוגמה נרים שוב את אותה הדוגמה עם מתמון 2-Way :

- הקריאה הראשונה מפספסת ונכנסת לסט 0 (way 0).
 - הקריאה השנייה מפספסת ונכנסת לסט 0 (way 1).
 - הקריאה השלישית מפספס ונכנסת לסט 1 (way 0).
 - הקריאה הרביעית מפספסת ונכנסת לסט 0 (way 1).
- בכינסת 1 way.

- הקריאה החמישית פוגעת כי היא מבקשת כתובת שמצויה בשורה שהשארנו בסט 0 (way 1).

הערה אין חשיבות לדוקא 2 ways, יש גם מטמוניים שהם way-4 וכיוצא-ב. כמובן שככל שיש לנו יותר ways, כך הסיכוי להתגשות יותר נזק.

תרגול

הערה בלי להכניס את ה-branch, decode, PC רק בשלב ה-branch וכתיבת PC לא יתאפשר memory access. אפשר עדין להימנע מקפיצה וכתייה ל-PC רק בשלב ה-branch (כבר אז ידוע לנו האם אנחנו ב-branch והאם תוצאה ה-ALU היא 0 או לא). החסרון בכך הוא שהוא מאריך עוד יותר את שלב ה-execute שבלאו היכי השלב הארוך ביותר מבין השלבים השונים. בהתאם למימוש המעבד,

דוגמא נניח $CPI_{new} = ?$ בפועל אידיאלי, שבתכנות יש 20% פקודות branch, stall ו-branch. מהו CPI בהתחשב ב-branch? נניח שתוכנינו שלושה פיצויים (הוספה שלושה stalls).

$$CPI_{new} = \frac{1}{1 + 0.2 \times 3} = 1.6$$

דוגמא רוצים להוסיף לסט הפקודות של MIPS עם pipeline פעולה כפל. נניח שתזומני השלבים הם הבאים

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

עומדת בפנינו שתי אפשרויות:

- להוסיף לשלב ה-execute לוגיקה שתחשב את פעולה הכפל. נוספת זו תעלה את זמן הריצה של שלב ה-execute ב-20%.
 - להוסיף שלב ל-pipeline שבו תהיה לוגיקה שמחשבת את פעולה הכפל באותו האורך כמו ה-execute (כך יהיו 6 שלבים).
- מה תהיה ההשפעה על הביצועים (latency ו-throughput) בהנחה שה-pipeline הוא נטול סכנות?

1. latency יעלה $2.4 \times 5 = 12ns$ כי הפעולה הקритית, הלא היא ה-execute, ארוכה ב-20% מלפני, כלומר אורכת 2.4 ns.

שניות, והוא קובעת את זמן המוחזר המיניימי עבור המעבד.

2. throughput עתה מחייב כי זמן המוחזר עולה ל-2.4 ns ויש לנו תוצאה (חוץ מהקצתה) אחורי כל מוחזר שעון, כלומר הוא

$$\text{עומד עכשו על } \frac{1}{2.4ns}.$$

3. latency יעלה $2 \times 6 = 12ns$ כי יש לנו עכשו שישה שלבים, שהארוך מביניהם לוקח 2ns וכל הוראה עוברת את כל ששת השלבים.

throughput לא משתנה כי אחורי שנבעע את חמישת השלבים הראשוניים עבור הוראה הראשונה, לאחר כל מוחזר שעון

$$\text{נקבל תוצאה, כלומר מספר תוצאות חישוב ליחידה זמן נשאר זהה - } \frac{1}{2ns}.$$

דוגמא עבור קטע הקוד הבאים, נזכיר אם יחייב 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 ישפה בעיה, מסוג(read-to-use) (קרוי גם interlock), ולכן נדרש להוסיף nop (הלא הוא .stalling).

.2

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

כאן יש לנו סכנת דאטה בין פקודה 1 ו-2, שנייתן לפטור בנסיבות עם forwarding (בין הערכים ב-EX/MEM EX שמחזיקים מידע על הפוקודה הראשונה לקלטים ל-ALU בשלב ה-execute של הפוקודה השנייה).

.3

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

נשים לב שכאן אין צורך מגנון למניעת סכנות, כי גם הערך של t_0 לא משתנה וגם t_3 נדרש ולכן לא צריך לשמור או לדוחות את הכתובת אליו מחדש.

שבוע XII | עוד על המטמון

הרצאה

סוגי פספוסי מטמון

- Compulsory : בפעם הראשונה שניגשים למידע בהכרח יהיה לנו פספוס, גם אם המטמון היה אינסובי.
- Niutn להפחית פספוסים מסווג זה ע"י הגדלת הבלוק במטמון.
- Capacity : פספוסים שנובעים מכך שגרשנו מטמון FA שורה שהייתה חלק מה-Working Set כי המטמון התמלא, כאשר ברמה העקרונית זה נובע מכך שהמטמון קטן מה-Working Set.
- Niutn להפחית פספוסים מסווג זה ע"י הגדלת המטמון.

- **Conflict** : פספוסים שנובעים מ“Asociational” נמוכה מדי (מספר הביטים לסט לא מספיק גדול) או מיפוי ישיר, לדוגמה כאשר שורה דורשת שורה אחרת עם אותה הסט.
- ניתן להפחית פספוסים מסווג זה ע”י הגדלת האסוציאטיביות.
- **Coherency** : פספוסים שנובעים משינויו הערך בזיכרון ע”י תהליך אחר.

דוגמה נניח שאנו עוברים על מערך מילים בגודל 256 מילים (שהוא מיושר ביחס לזכרו) ויש לנו מטמון כלשהו. אם גודל הבלוק במטמון הוא compulsory 4 בתים, כל גישה לכל מילה תהיה miss compulsory, לעומת זאת אם גודל הבלוק הוא 64 בתים, הגישה הראשונה תהיה compulsory אבל השנייה עד ה-16 יהיה hit כי קראוינו 16 מילים בפספוס הראשון.

נשים לב שהמשק שהמטמון מציג למעבד זהה לזה של הזכרון המרכזי (להוציא מה-miss), וכך נוכל להחליף את האחרון בראשון, כאשר המטמוניים הן של הדאטה והן של הפקודות מחוברים ל-*bus* שהמחובר לזכרון המרכזי שקורא או כותב לסוג הזכרון המתאים (שאוחדר יחד עם שאר הסוגים בזיכרון הראשי).

הערה עכשו שהמעבד מתקשר עם מטמון מהיר ולא הזכרון הראשי האיטי מאוד ביחס למעבד, נוכל באמצעות pipeline שבו גישה לזכרון היא במחזור יחיד.

ביצועים של מטמון

הגדירה Miss Rate הוא חלקיות הפספוסים מתוך סה”כ הבקשותHit Rate-Hit Rate-Miss Rate (AMAT) Acess Time .Miss Rate-miss כפול ה-

הערה לעיתים נדרש להוציא ל-*AMAT* זמן lookup.

דוגמה נניח שיש לנו מעבד עם CPI_{ideal} = 1.1 (אידיאלי, כלומר אם אין hazards ו-i-stall-iים) ותוכנה עם 50% פעולות אРИתמטיות, 30% control ו-10% מפעולות הזכרון הנגרמות ב-*miss* עם CPI_{control} = 2.0 ו-*i*ld/st האפקטיבי?

$$\text{CPI} = \text{CPI}_{ideal} + \text{זמן לפקודה -stall}$$

$$= 1.1 + (0.3 \cdot 0.1 * 0.5)$$

$$= 1.1 + 1.5 = 2.6$$

כלומר 58% מהזמן ($\frac{1.5}{2.6}$) המעבד מכחך לזכרון.

סקולול תמורה (טרריידוו) גודל הבלוק

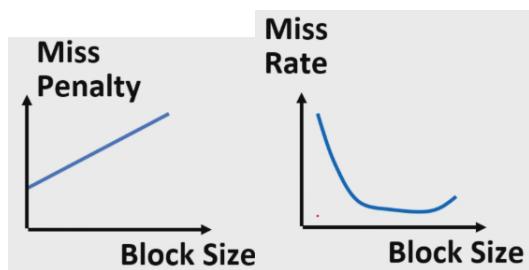
- יתרונות של בלוק גדול:

- משרת לוקאליות למרחב (אם ניגשנו למילה כלשהי, נוכל לגשת ליותר מיללים סביבה בלי לפספס).
 - משרת הרצה סדרתית של פקודות (אם נריץ פקודה, סביר שנריץ את כל האלו שאחריה, כMOVED להוציא מהמקרה שנקפוץ).
 - משרת גישה סדרתית למערכות.
- חסרונות של בлок גדול :

- זכרון גודל מעלה את ה-Miss Penalty.
- אם הבlok גודל מדי ביחס למטרונו, יהיו לנו מעט מאוד בלוקים.

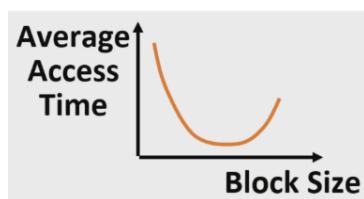
המטרה שלנו היא למזער את ה-AMAT.

הערה בפועל נקבל את הגראפים הבאים של ה-Miss Rate ו-**Miss Penalty** כפ' של גודל הבלוק



הסיבה לעלייה ב-Miss Penalty היא פשוט כי יותר זכרון נדרש (בנוסח latency) בסיס שוגר מהמරחק בין הטענו לזכרו). הסיבה לירידה ב-Miss Rate בהתחלה היא בזכות מענה יותר טוב לлокליות מרוחבית (שלשות הירונוט שהזכרנו) והעליה בהמשך נובעת מירידה משמעותית ב-capacity כתוצאה מבלוקים גדולים מדי ביחס לגודל המטרונו כולו.

בגלל שה-AMAT מורכב משני הפרמטרים הנ"ל, נקבל סה"כ את הגרף הבא



כאשר הנקודה האופטימלית משתנה בהתאם לתכנית (כך גם הגרף כולו).

מטרוֹן רב-שלבי

אם יש לנו שלב אחד של מטרונו, כל פספוס יהיה מאד יקר. אם נוסיף שלב ביןיהם, המקרה הכى טוב לא יפגע אבל המקרה הממוצע של פספוס יהיה מהיר יותר כי נוכל למצוא את הבלוק בשלב הביניים במקום בזיכרון הראשי. העלות של זה היא מטרונו נוספת (שהוא איטי יותר אבל גדול יותר), שmbיא איטו latency יותר גבוהה במקרה קרייה מהזיכרון.

דוגמה נניח שזמן הגישה ב-Hit הוא 3 מחזוריים-L1 ו-6 ל-L2, שה-Miss Rate L1 ו-3% ב-L2 ושה-Miss Penalty של L2 הוא 200 ממחזוריים.

$$AMAT = \text{Time Hit L1} + \text{Rate Miss L1} \cdot \text{Penalty Miss L1}$$

$$= \text{Time Hit L1} + \text{Rate Miss L1} \cdot (\text{Time Hit L2} + \text{Rate Miss L2} \cdot \text{Penalty Miss L2})$$

והצגה תחשוף שעם L2 נקלט AMAT יותר טוב מ-L1 (עבור 200 Über 1).

יעולים נוספים לגישות לזכרון

- הרחבה ה-bus : נוכל בעלות של רכיב יוטר יקר להעתיק יותר בתים מהזכרון למטרמון (לא למעבד, שבכל מקרה עובד עם מילאים), כלומר העלנו את ה-throughput latency (בכפוף לזה ששני גודל הבלוק למטרמון לא פוגעת בBITS).
- קיצור ה-bus : הקטנת המרחק הפיזי בין המטרמון לזכרון מקטינה את ה-latency.
- Memory Interleave : שימוש חומרת הזכרון באמצעות שני רכיבי זכרון עם כתובות שטניות לסדרוגין. כך שתי גישות לזכרון ש מגיעות לרכיבים שונים יכולו לקרות במקביל, או לפחות לא נדרש להוכיח קבילים בזיכרון בין הגישות.

מדיניות גירוש

במטרמון Direct Mapped אין לנו בחירה את מי לגרש - זה חייב להיות הבלוק שתופס לנו את הסט. ב-FA ואסוציאטיבי N-way צריך לקבוע את מי לגורש.

- גירוש פסאודו-אקראי : עובד לא רע, כי הסיכוי שנזרוק בבדיקה אחד שנctruck בקרוב הוא נמוך.
- FIFO (First In First Out) : נגרש את הבלוק היישן ביותר. מאוד קל למימוש, אבל לא משרת היטב לוקאליות בזמן (במקרה של מחסנית בעת ריצה על מערך יצא שנזרוק לא מעט ערכים במחסנית שזה לא אידיאלי).
- LRU (Least Recently Used) : נגרש את הבלוק הכי פחות בשימוש. זה משרת היטב לוקאליות בזמן, אבל קשה למימוש בחומרה כהאסוציאטיבית גבוהה. לעיתים נשערך LRU באופן יותר קל למימוש בחומרה (PLRU).

מדיניות כתיבה למטרמון

בעת פעולות store, יש שתי אפשרויות למימוש הכתיבה במטרמון במקרה של Hit.

• Write Through : תמיד נכתב גם לזכרו וגם למטען (לרוב עם buffer כתיבה כדי לצמצם את זמן ההמתנה של ה-CPU).
היתרון המרכזי הוא שומרם על Coherence בклות (הטען והזיכרון באותו מצב) והחסרון המרכזי הוא שזה מעלה את הניצול של רוחב הפס של הזכרון.

• Write Back : נכתב רק למטען, ובעת גירוש נעדכן בזכרו (נמשך עם בית modified נוסף בכניסת המטען).
היתרון המרכזי הוא שזה מוריד את הניצול של הזכרון וה-latency בעת פעולה כתיבה. החסרון המרכזי הוא שיש צורך לאובדן מידע (בעת כיבוי המחשב, לדוגמה), ובנוסח שכחיש כמה תהליכי שרוצים בו זמינות צריך לבצע בדיקות קוהרנטיות כל הזמן (snoops).

בדומה, יש שתי אפשרות למימוש הכתיבה במטען במקרה של Miss.
• Write Allocate : נקראת המילה מהזכרן למטען (כאיו היה לנו Miss בקריאה) ואז נדרוס את הערך המטען עם מה שנרצה לכתוב. זה שימושי במקרה שאנו מצפים לכתובות נוספות לאותו בלוק (מתכתב ריעונית עם Write Back).
• Write Through : נכתב ישרות לזכרו המרכזי בלי להביא את המילה למטען (מתכתב עם Write No-allocate).

דוגמא נניח שיש לנו מערך ששמור בזכרו בצורה מטריצה דו-ממדית ($m \times n$) של מילימ (4 בתים) עבור $n = m = 1024$ כאשר השורות שמורות סדרתית ועמודות של אותה שורה מופרדות אחד מהשני (ראו או איוור)

N columns					
0,0 base+0	0,1 base+4	0,2 base+8	...	0,n base+4n-4	
1,0 base+4n+0	1,1 base+4n+4	1,2	...	1,n base+8n-4	
2,0 base+8n+0					
3,0 base+12n+0					
...					
m,0 base+(m-1)n+0				M,n base+mn-4	

M rows

כלומר גישה סדרתית במערך לפי שורה תרוויח מניצול טוב של המטען את הлокאליות המרחבית, ואיילו גישה לפי עמודות מסבב thrashing. נראה זאת. נניח שיש לנו מטען בגודל 32KB, כאשר כל בלוק הוא בגודל 64 בתים (סה'כ 512 בלוקים במטען).
הגישה למערך היא לפי array[col][row] (זה המשיק שהזיכרון מספק). מה יקרה כשנ裏ץ את הקוד הבא (リスト אינדקסים לפי שורות בחוץ ועמודות פנימה)?

```
int sum = 0;
for (int i = 0; i < 1024; i++)
```

```

for (int j = 0; j < 1024; j++)
    sum += A[j][i];

```

אנחנו ניגשים לתאים $(0, 0)$, $(0, 1)$ וכן כאשר כל אחד מהם מקבל Compulsory Miss (כי הבלוק שנקרא מכיל תאים נוספים מאותה שורה), קלומר ה-Hit Rate הוא 0.

אם נחליף את סדר הרצת האינדקסים (עמודות בחוץ ושורות בפנים) אז נקבל ניצול מיטבי של המטמון ו-Hit Rate של $\frac{15}{16}$ (בלוק הוא 16 מיילים).

בלוק קטן יותר יגרום לפגיעה במקרה הראשון להיות יותר מצומצמת (ЛОקח פחות זמן לקרה כל בלוק וגם יש יותר בלוקים במטמון לפני גירוש). אם עולה את האסוציאטיביות לא נקלט שניי משמעותי כי בכל מקרה אם אנחנו N -אסוציאטיביים אז בכל מקרה תמיד נגרש מחזוריית את הבלוק ה- N לפניו במערך (כי הסט חוזר על עצמו).

דוגמא מה אם נעשה כפל מטריצות? השתמש בקוד הבא

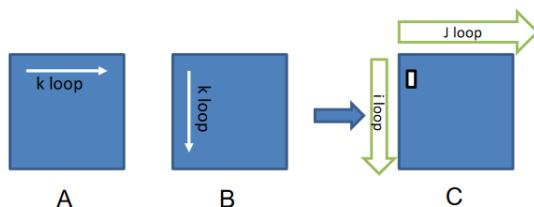
```

int sum = 0;

for (int i = 0; i < 1024; i++)
    for (int j = 0; j < 1024; j++) {
        int tmp = 0;
        for (int k = 0; k < 1024; k++)
            tmp += A[i][k]*B[k][j];
        c[i][j] = tmp;
    }
}

```

נשים לב שאין לנו מנוס מלעבור גם על עמודות וגם על שורות (מטריצה אחת נורץ לאורך ואחרת לרוחב, לא משנה מי זו מי) ולכן בהכרח ננצל גרוע את המטמון. ריצת האינדקסים היא כזו



תרגול

בahirkeiyt hozoruno, chayib lehatkiyim ukruon hahechla, kolomer shekl maha shmotman b'shchba gibohah behcherach nme'az gam b'shchba nmoocha yotra.

הערה עקרון ההכללה נדרש כדי שונוכל לחפש איטורטיבית מידע משכבה gibohah לנמו'חה ולא לנחש שהיא בשכבה אחת, לקוות שאם נכתב אליה זו הערך מהשכבה לפני לא ידרס וכו'.

דוגמה כמה ביטים סה"כ צריך בשבייל מטמון direct mapped עם 16KB של נתונים ובלוקים בגודל 4 מילימ"מ (בארQUITטורת 32 ביטים)?
 בלוקים דורשים 4 ביטים להיסט בתוכם $4 \cdot 4$ ביטים בכל לוק), ולכן יש לנו $2^{10} = \frac{2^{14}}{2^4}$ בלוקים במטמון. נותרו לנו $18 - 4 = 14$ ביטים לתג (להוציא מההיסטוריה והסטט) ויחד עם בית וולידיות נוסף, כל כניסה תתפוץ 19 ביטים, כלומר מערך התגים תופס $2^{10} \cdot 19 + 2^{10} \cdot 2^2 + 2^5$ ביטים, ויחד עם מערך הדאטה זה יתפוץ $2^2 \cdot 2^5 \cdot 2^{10}$ ביטים.

שבוע X | זכרון וירטואלי

כל תכנית מניחה שהזיכרון שלה מסודר לפי פורמט שכבר ראיינו בקורס נציגות של פ' - סגננטי Text, BSS, Heap וכו'. אם יש לנו כמו תחilibים, נדרש שמערכת הפעלה תציג מציג שווה לתוכנית שאכן כך זה נראה, גם אם פעולה זה לא. בנוסף, נרצה שתהליכיים יכולים לגשת לזכרון גדול יותר מאשר זה שקיים פיזית, ולשם כך נדרש לעשות לזכרון וירטואלי מספק לנו גם במידוד בין תהליכיים. לסום, זכרון וירטואלי נותן לנו אורתוגונליות בין השימוש בחומרה של הזיכרון לבין ריצעה של תוכנית מעליו.

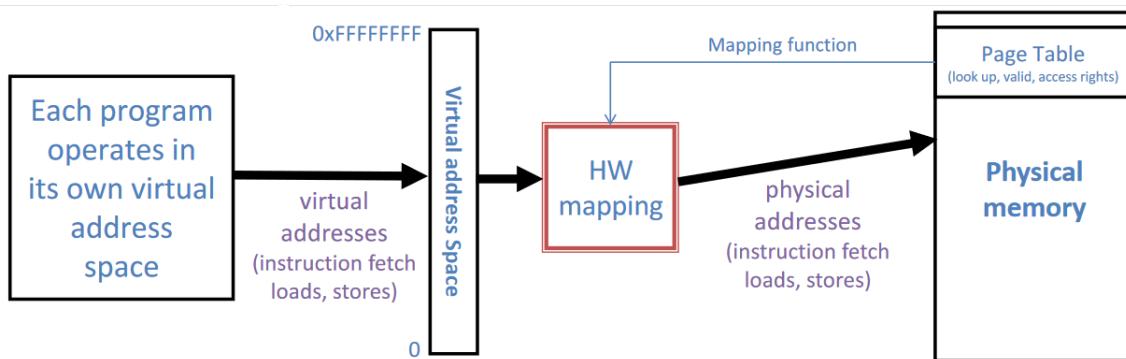
כל תחילה יש מרחב וירטואלי אחד. נפריד את מרחב הכתובות הפיזי למסגרות באותו הגודל כמו הדפים (לרוב 4KB).

ນפה לזכרון הפיזי רק דפים שבם התהיליך משתמש בריצעה שלו (זה תמיד יהיה משמעותית קטן ממרחב הכתובות הווירטואלי כולו). דפים שאנו לא משתמשים בהם נשמרם בדיסק, כדי שנוכל בהמשך להביא אותם לדפים בזיכרון לשחזור לשימוש בהם.

הערה בסופו של דבר הזיכרון הראשי מהו זה מטמון לדיסק, ואוטם עקרונות לוקאליות ושיפור ביצועים שראינו במטמון חלים גם כאן.

כך תראה ייחידת התרגומים והקרירה מהזיכרון הראשי. ה-Page Table היא הטבלה ששומרת את המיפוי מדפים למסגרות (כניסה לכל דף), יחד עם שדות נוספים כמו בית וולידיות, זכויות גישה ובית dirty (אוול גם ביטים לצורך מדיניות גירוש כמו LRU).

אם בית הולידיות כבוי, איןדקט הפריים יקבע לכתובת בדיסק בה שמור הדף. הביט dirty קובע האם כתבנו לדף זה מאז שקראננו אותו מהdisk, כלומר שבגירוש צריך לכתוב אותו לדיסק ולא מספיק לדروس אותו.



הערה המיפוי נעשה בחומרה (אבל ניהול הטבלה בתוכנת הkernel!) כך שהוא שוקף לכל התוכנות (של משתמשים).

הערה לכל תחילה יהיה Page Table נפרד, שמערכת הפעלה תנהל בעצמה, והוא זו שתאפשר לשתי טבלאות שונות להתמפות לאותו זיכרון פיזי.

דוגמה תרגום הכתובות נעשה כמו שעשינו במתמטו: נניח שיש לנו מרחב כתובות וירטואלי בגודל 2^{32} ביט ומרחב כתובות פיזי בגודל 2^{30} ביט וגודל דף $= 4KB = 2^{12}$. כתובות וירטואלי בת 32 ביט תורגם כך: נשמר בצד את ההיסטוריה, שנדרשו 12 ביט וניקח את ה-20 ביטים הגבויים של הכתובת, ונתרגם אותם לאינדקס של מסגרת בזכרון הפיזי (Physical Page Number) באמצעות ה-Page Table. לאחר מכן נצמיד את האינדקס הזה חזרה להיסטוריה וזו תהיה הכתובת במרחב הפיזי.

כל כניסה ב-Page Table לוקחת $1 + 20 - 12 = 8$ ביטים (הוספנו 1 לוולידיות) אבל כדי שהיה נוח לגשת, לרוב נעגל ל-32 ביטים. הטבלה בתפוקס סה"כ $2^{22} \cdot 2^{32-12} = 2^{22} \cdot 4$ ביטים (כניסה לכל דף).

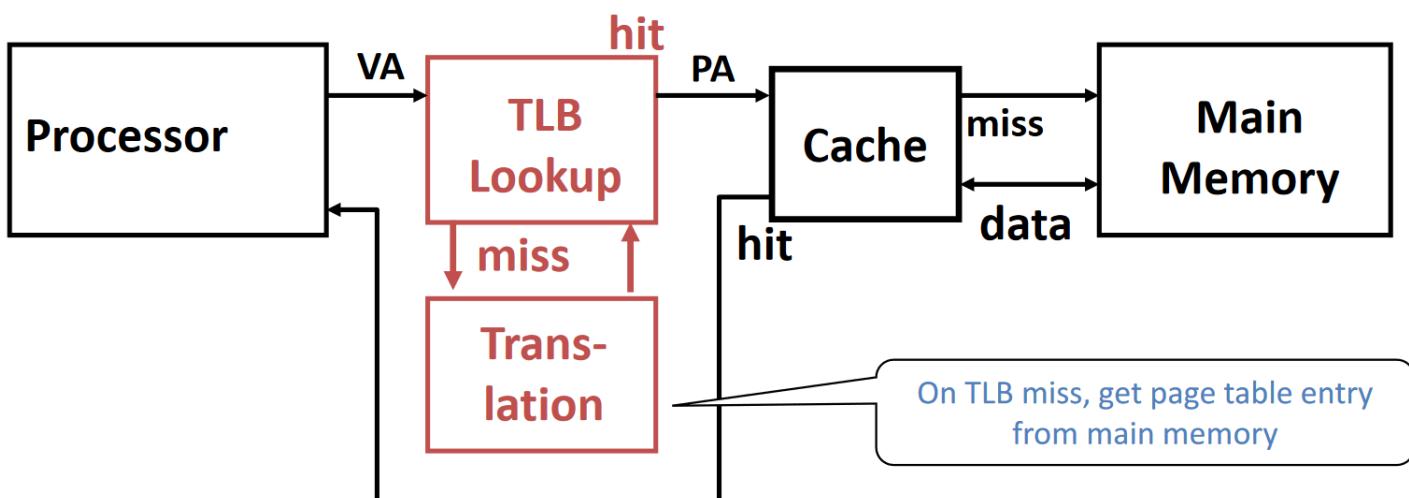
המצביע בבסיס ה-Page Table Register, כך שאפשר באמצעות סכימה שלו ייחד עם אינדקס הדף לתרגם (כפול 4) להגעה לכניסה שמכילה את ה-PPN המתאים.

אם המעבד ינסה לגשת לכתובת בזכרון שלא מופת (כלומר הביט וולידיות כבוי), המעבד יקבל שגיאה שנקראת **Page Fault**. לאחריו, החומרה נותנת למערכת הפעלה את השיטה להביא את הדף מהדיסק, לגרש דף מהזיכרון הפיזי אם אין מקום, ולעדכן את ה-PTE (הכניסה הרלוונטית ב-Page Table). לאחר סיום פעולות ה-OS, היא תחזיר את השיטה לחומרה שתמשיך את ריצת המעבד כרואיל מפקודת הגישה לזכרון שנכשלה.

הערה סוג ההשגיאה הזה נקרא interrupt והוא מאפשר לעצור ריצה של תוכנית גם אם לא מופיעה העזירה בקוד.

TLB

ביצוע התרגומים לוקח הרבה זמן (דורש גישה ל-Page Table), ולכן נרצה ליעיל אותו. לשם כך משתמש במתמטון קטן שומר את תוכנות התרגומים, שיפעל באופן דומה לזה שראינו בהרצאות הקודמות. כך במקרה הטיפוסי תרגום יהיה מאוד מהיר, ובמקרה של פטפוס ניגש ל-Page Table בעצמו (ראו איור).



הערה לרוב ה-TLB יהיה בתצורת FA.

הערה המטמון שומר אצלו בלוקים של זכרון פיזי, לא וירטואלי, لكن ניגש אליו רק אחרי תרגום הכתובת!

פורמט הכתובת (של DATAה, לא כולל התג שנדרש למטרמו) ב-TLB זהה לזו ב-Page Table (שכן הוא מטמון שלה), כלומר יהיו לנו גם ביטים של dirty, validity, הרשות וכמוון הכתובת המתוגמת.

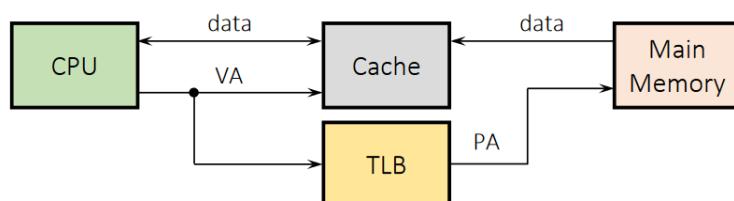
הערה כשבועור בין תהליכי ננקה את ה-Page Table (כלומר נקבע את כל הביטים של וolidiyות להיות 0), כדי לא להתבלבל בין דפים של תהליכי שונים.

איך נתרגם אם לא מצאנו ב-TLB

- **מנוהל ע"י תוכנה (MIPS)**: ברגע שיש miss ב-TLB, החומרה תשלח trap וה-OS יירץ את מה שקרה ב-TLB miss. נשים לב שהוא Page Fault נושא מעבר למה שכבר יש, שקרה כנדרך להביא את זה ל-Page Table. במהלך ה-OS תצטרכן לכתוב לדיסק את תוכן ה-page אם יש ביט dirty דלוק על הדף שגירשנו מה-TLB. תחת שיטה זו ה-Page Table לא עשויה יותר מדי כי הבלה האפקטיבית של דפים שאנחנו מחזקים היא ה-TLB.
- **ה-OS תצטרך להכיר את פורמט ה-PTE**, אבל המעבד לא.
- **מנוהל ע"י חומרה (רוב המעבדים)**: החומרה מביאה מידע מה-Page Table אל ה-TLB (כמו במטרמו הקלאסי). במקרה כזה חומרת המעבד צריכה להכיר את פורמט ה-PTE, אבל ה-OS לא צריכה בכלל.

יעולים

1. להשתמש בחלק מהביטים בהיליט כסטים ב-TLB שאינם FA. כך נדע את ה-way כבר לפני שנתרגם כי הוא נשאר קבוע.
2. להשתמש במטרמו על כתובות וירטואליות. היתרון הוא שכן נדרש לתרגם לכתובת פיזית רק במקרה של פספוס במטרמו.



זה בעייתי כי לשני תהליכי יכולם להיות את אותן הכתובות הווירטואליות וזה התנששות שאנחנו לא רוצים שתקרה, ולכן בין תהליכי ננקה את המטרמו כדי למנוע שימוש בדפים של תהליך אחר. לחולפן, אפשר להוסיף שדה של PID ליד כל PTE וכך נדע לא להתייחס לפגיעות של המטרמו על דפים לא שלנו.

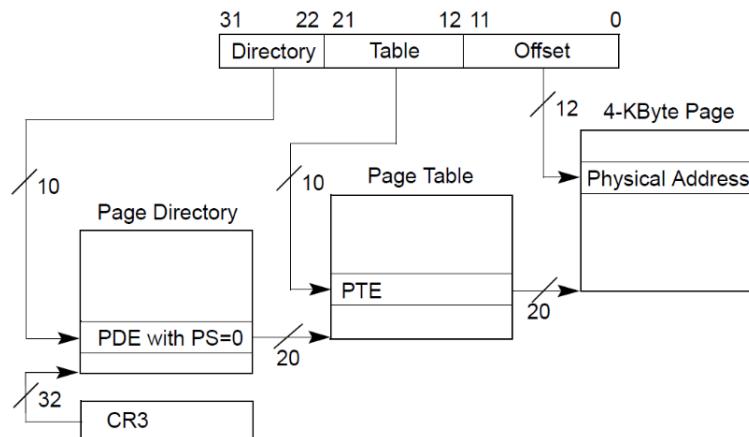
טבלת דפים היררכית

אם יש לנו מרחב כתובות וירטואלי של 64 ביטים וזכרון פיזי של 32 ביטים ודף בגודל 2^{12} בתים. במקרה כזה ה-Page Table עצמה יתפוז $= 2^{64-12} = 2^{52}$ כניסה שזו יותר מכל הזכרון הפיזי עצמו. מה שמציל אותנו היא העובדה שכמעט בכל הכתובות בטבלה אנחנו בכלל לא משתמשים.

כדי לפחות את הבעיה השתמש בטבלה דפים היררכית. תהיה לנו טבלה דפים שהוא השורש, שכל הכניסות שלו מצביעות לטבלות שנייניות, שמצוינות לפחות עד שנגיע לעליים שהם טבלאות שמצוינות באמה ל-PPN-ים. את האינדיקציה נעשה באמצעות חלוקת הכתובת היררכואלית שנותרה לאחר הסרט ההיסט בתוך הדף לכמה סגמנטים (*x* הביטים הראשונים משמשים לטבלה הראשונה, *y* הבאים לטבלה השנייה וכן הלאה).

אם נסמן הטבלה במלואה יותר גדולה, אבל אם רק נשמר בזיכרון הפיזי את הדפים שכוללים את טבלאות לא ריקות, נידרש למשמעותית פחות זכרון לאחסן הטבלה היררכית.

דוגמא ב-32Bit יש לנו טבלה היררכית עם רמת היררכיה אחת, כלומר השורש (Page Directory) שמצוין לטבלאות רגילות (מצוינות לדפים), ראו איור



דוגמא נתון מעבד עם מוחב כתובות ב-64Bit המוחולק לדפים בגודל 2^{21} בתים. המעבד מחובר לזכרון בגודל 2^{46} בתים ודפים ממופים עם טבלה לא היררכית. נרצה להציג 4 תהליכיים שנמצאים בזיכרון בו זמנית, האם זה אפשרי? אם כן, כמה זכרון נדרש לטבלאות הזכרון של כל התהליכים יחד?

נctract 21 – 46 Bits ל-PPN (הורדנו את הביטים של ההיסט) ואמצע ל-32 (בשביל הביטים הנוספתיים ב-PTE) נקבל שכל כניסה דורשת 4 בתים. לכן כל טבלה דורשת $2^{43} = 2^{64-21}$ בתים לכל תהליך. כך נוכל להחזיק בכל היותר טבלאות של שני תהליכיים בו זמנית, אבל לא 4, כי זה ידרשו 2^{47} בתים בזיכרון הפיזי, שאין לנו.

אינטראפטים

נרצה לעזר את ריצת התוכנית באופן לא מתוכנן, לדוגמה במקרה של Page Fault. נרצה שזה יקרה באופן שקויף כך שהתוכנית לא תשים לב שהיא נעצרה, ולכן ההאנדרל של ה-*exception* (שהיא השגיאה שקרתה שגרמה לעצירת הריצה) צריכה לשמור את המצב שהתוכנית רואה כפי שהיא לפני השגיאה.

מקרה נוסף כזה קורה במקרה של Context Switch, כלומר כאשר אנחנו עוברים מהרצת תהליך אחד לאחר כי נגמר סלוט הזמן שהוקצתה לתהליך. בפועל זה ממומש עם רגיסטר Exception Program Counter כז שלא נctract לגעת ב-PC של התוכנית. הדבר הראשון שקרה בקוד שאליו מצביע ה-EPC הוא שמירת כל הרגיסטרים שההאנדרל מתוכנן לגעת בהם למחסנית נפרדת מזו של פ' רגילה, ואז כשהוא יחזור הוא ישחרר את הערכים הללו חוזה לרגיסטרים.

ב-*Multi Cycle* לדוגמה, אפשר פשטוט להוסיף עוד מצבים שעובהו אליהם אחרי ה-*fetch* או אחרי ה-*execute*, או אף לפני ה-*fetch*. אם מתקיימים תנאים שגורמים למינון exception (בעבר ב-*trap*, ב-*execute* זה קורה ב-*fetch* ו-*overflow* ו-*coincident*), וב-*memory access* זה בנסיבות שקוראים או כתובים מכתובות לא חוקית).

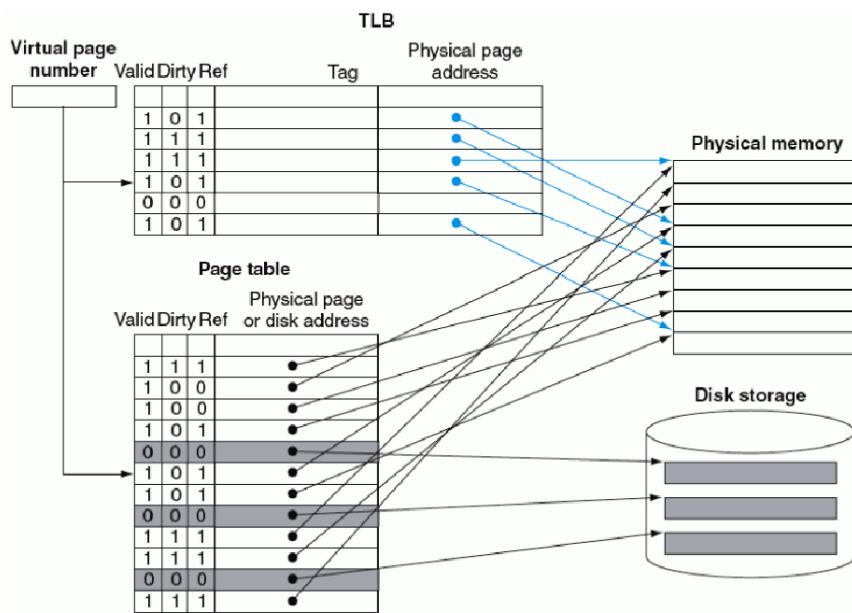
המעבד יחזק רגיסטר Cause שמחזק ערך שמתאים לסיבה שגרמה לשגיאה, כך שנוכל לדעת מה עצר לנו את הריצה והאם צריך לעשות משהו בעניין. נוסף על כך, נוסיף לשלב ה-*fetch* מרובב שיאפשר קפיצה ל-*EPC* במקרה של Cause שאינו 0.

תרגול

הערה יש חשיבות גבואה לרכיב האבטחה ב-*indirection* בין הזכרון הווירטואלי לבין הפיזי ושבירת הרציפות. כך וירוסים לא יכולים פשטוט לגשת לזכרון שלנו באמצעות buffer overflow ודומיהם.

מימוש של LRU הוא מתגר בתוכנה. אפשר לחלופין לשימוש ב-*Not Recently Used*, כלומר נסיר דפים שלא היו בשימוש לאחרונה (לא בהכרח העתיק ביותר). נוסיף עוד ביט reference שערכו יהיה האם ניגשנו לדף. אחת לכך, מערכת ההפעלה תקבע את הביטים האלה, ואז נגרש דפים עם reference bit כבוי, שהוא אומר שלא השתמשו בהם מאז האיפוס האחרון, כלומר שערכנו את LRU. עתה ה-*PTE* שלנו יכיל ביטים של validity, reference, modified וכן את ה-*PPN* (נתעלם עכשו מלהרשאות).

דוגמה להלן הדוגמה של TLB (שהוא FA) וטבלת דפים.



מהלך קריאה מהזיכרון

יש לנו כתובות וירטואליות

• ניגשים ל-TLB :

- יש פגיעה? קוראים מהמטמון ואם לא שם אז מהזכרונו הפיזי וסיימנו.
 - יש פספוס? נגים לטלת הדפים:
- * אם המידע בזכרונו (יש `valid=1` בטלת הדפים), מכנים את ה-PTE ל-TLB וממשיכים משם (קוראים מהמטמון ואם לא שם אז מהזכרונו הפיזי).
- * אם הוא לא, יש `Page Fault`, שודרש טיפול של ה-OS וגישה לדיסק (ואז מפעעים לזכרו הפיזי ומשם למטרמון וכו').

דוגמא נתון מרחב כתובות וירטואלי ברוחב 32 ביט המחולק לדפים בגודל 8KB ו-TLB עם 256 כניסות.

- כמה שורות יהיו ב-Page Table?

היחסט דורש 13 ביטים כך שיהיה 2^{13} דפים, מספר השורות ב-PT.

- כמה שורות מה-PT יתמכו לכל TLB בהנחה שה-TLB הוא ?Direct Mapped TLB.

יש 2^8 כניסות ב-TLB, כלומר 256 כניסות לכל כניסה ב-TLB.

- איך נראה חלוקה של ה-TLB לשורות בהנחה שה-TLB הוא ?Direct Mapped virtual address הואTLB 13 ביטים להיחסט, 8 לסט ושאר ה-11 להיחסט.

דוגמא נתון מרחב כתובות וירטואלי של 32 ביט המחולק לדפים בגודל 4KB, TLB עם 256 כניסות שהוא זיכרו הפיזיTLB בגודל 256MB. בנוסף נתונים הכנימות הבאות ב-TLB

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

האם הכתובת 0x0FB00ABC ממופת ל-TLB? אם כן, מצאו את הכתובת הפיזית המתאימה לה.

נשמרים 12 ביטים להיחסט בתוך הדף ולכון 20 הביטים הראשונים (שחמשים 5 האותיות הראשונות) משמאלו הם ה-VPN. הסט של ה-TLB הוא האות הרביעית וה חמישית (8 ביטים) ושלושת האותיות הראשונות (12 ביטים) הם התג. כלומר נבדוק האם בסט 0x00 יש את התג 0xFB0, ואכן יש!

נותר להצמיד את ה-VPN להיחסט בדף ולקבל את הכתובת הפיזית, הלא היא 0x00F00ABC.

נשים לב שלא השתמשנו בכלל בתנתן של גודל הזיכרון הפיזי (牟וטיב שיחזור על עצמו גם ב מבחנים).

דוגמא נתון מרחב וירטואלי ב-32 ביט וגודל דף של 2^{13} בו. מבנה הכניסה ב-PT הוא כבאיור

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

- מהו גודל הזכרון הפיזי?
- ה-PFN הוא 12 בית וכאן יש לכל היותר 2^{12} מסגרות פיזיות. כל מסגרת היא בגודל 2^{13} בתים ולכן סה"כ המרחב הפיזי הוא בגודל לכל היותר 2^{25} בתים.
- מהו גודל ה-PT?
- נספור את מספר ה-VPN'ים; מתוך 32 בית, 13 מוקצחות להיסט בתוך הדף הווירטואלי וה-19 האחרות ל-VPN, כלומר יש 2^{19} VPN'ים וכמוותם כניסה ב-PTE. כל שורה ב-PTE תופסת 16 ביתים ולכן סה"כ מדובר ב- 2^{23} ביתים, שהם 2^{20} בתים.

שבוע II | חישוב במקביל

נרצה להציג את ביצועי התכננית שלנו. נוכל או להריץ סריאלית (סדרתית) יותר מהר (לקצר את זמן המחזור, שימוש במטמון ועוד), או לפרק את התכננית להרבה חלקים קטנים שנייתן להריץ במקביל.

הערה העניין שלנו במקבול עולה כי הגענו לרווחה מבחינת שיפור הביצועים של ליבה אחת (ביקורת חשמל לביצועים, שיטה לביצועים ועוד).

טකסונומיה של מקבול

- **מקבול מערכת**: פירוק המשימות השונות מתוך המחשב כולם למערכת (תהליכיים לדומה) והרצתן במקביל.
- דוגמה** במשחק מחשב יש הרבה רכיבי תוכנה שונים: מעבד לוגיקה, מעבד לארטיק, מעבד שmagיע מהאינטרנט (במשחק מקוון).
- nocל להריץ את התכניות האלה במקביל ויתקשרו אחת עם השנייה בעת הצורך - רוב הזמן, אין סיבה שייחכו אחת לשניה לזמן עיבוד.
- **מקבול Task Pipeline**: פירוק תכניות למשימות באוטה תוכנה והרצתן במסאבי עיבוד שונים.
- דוגמה** בעבד גרפי יש כמה שלבים pipeline עד לתצוגה, כאשר היחידה האוטומטית בגרפיקה היא פוליגונים, שהופכים לפרגמנטים, וכו'. כל הפעולות כאן אמנים תלויות אחת בשנית, אבל כל שלב דורש סוג חישוב/פקודות שונות מאוד, ולכן רכיבי עיבוד שונים עובדים בשלבים שונים ואת זה ניתן למקבול. סיבה זו במחשבים רבים יש GPU שהוא מעבד ספציפי לגרפיקה.
- **מקבול תת-משימות**: פירוק תכנית להרבה משימות איחודות וקטנות הרצתן במקביל.
- דוגמה** בעת שאלתה למסד נתונים, כל מעבד יריד את השאלה על חלק אחר במסד, כאשר המשימות (זמן הריצה שלhn) זהות לחלווטין.
- אם המשימות לא איחודות אבל יש לנו מעבדים אחדים, אפשר לבנות תור משימות שכל מעבד יקח ממנו משימה ברגע שישים את הקודמת, ואז בתוחלת נקבל ביצועים שווים בין המעבדים.
- **מקבול פקודות**: הרצת פקודות אסמבלי במקביל.
- דוגמה** במעבדים מודרניים מרים כמה פקודות במקביל כי יש להם כמה יחידות execution וכמה יחידות issue (פענוח).

ההבדל ביחס למקבילים האחרים היא שהמקובל הוא ברמת המעבד, והתכנית עצמה לא משתנה (זה שקווי למשתמש).

- מקובל דатаה : הרצת במקביל של משימה על חלקים שונים במידע.

דוגמה סכימת שני וקטורים ניתנת לביצוע במקובל מלא (כל שני איברים נסכימים בנפרד ובאופן ב"ת מאחרים).

במציאות מעורבים את כל הפתרונות האלה : יש לנו הרצת במקביל בכל ליבה, הרצת OOO שיכולה להריץ עד 4 בפקודות בכל זמן מוחזר, יש מעבדים שונים לגרפיקה וכו'.

הטקסטונומיה של Flynn

- (SISD) Single Instruction Stream, Single Data Stream : מעבד יחיד.
- (SIMD) Single Instruction Stream, Multiple Data Stream : יכולת עיבוד של וקטורים (Vector Machines, CM-2) בליבה אחת
- (MIMD) Multiple Instruction Stream, Multiple Data Stream : חישוב בקלאסטרים - מקובל ברמת threads.
- (MISD) Multiple Instruction Stream, Single Data Stream : לא שימושי לשום דבר חז' מאולי אבטחה, אז נרצה להריץ כמה תוכנות שונות את אותו הדבר ונבדוק שהן קיבלו את אותן התוצאות, במתוסים למשל.

סנכרון ות'רדים

בעת הרצת במקביל של ת'רדים, הבעה המרכזית צריכה לפטור היא סנכרון (ניהול גישה למשאבים משותפים במקביל). פתרונות סנכרון מאייטים את הריצה של ת'רד אחד, אבל שומרים על חוקיות הדטה, וכך אי אפשר למקבל כל דבר כי לפעמים הסנכרון יעלה את זמן הריצה ליותר זמן מאשר הריצה בת'רד אחד. ת'רדים הם תת-יחידות חישוב שמוכלים יחד בתהליכיים, כך שימושם להם מרחב זכרון.

הערה תהליכיים חדשים נוצרים ע"י קריאה ל-`fork` שנקרא `fork` בזמן ריצת התהליך. יוצר תהליך זהה לחלווטין לו שיצר אותו, מבחינת המיקום בתכנית שמננו ממשיכים והזכرون של התכנית, בלבד ה-`pid`, שבאמצעותו מזוהים האם אנחנו תהליך אב או בן. החל מהפיצול, שינויים במרחב הזכרון של תהליך אחד לא משפיעים על الآخر.

מודלי זכרון משותף ב-Multi-Processing

- Symmetric Multi-Processing : לכל המעבדים יש זכרון ייחיד משותף.
- במודל כזה גישה היא מהירה ויעילה אבל בסKİיל זה נהיה איטי. בנוסף, כל משתמש לפתח תחת מודל זה.
- Non-Uniform Memory Access : לכל מעבד יש זכרון משל עצמו וגם גישה לזכרון משותף.
- במודל כזה גישה היא יותר מורכבת (המפתח צריך לדעת متى לגשת לזכרון שלו ומתי למשותף), אבל בסKİיל עובדת הרבה יותר טוב (רובה הזמן המעבדים יעבדו עם הזכרון שלהם, ורק חלק מהזמן יגשו לזכרון המשותף שלו throughput מוגבל).

הערה ה-PT מאפשר למחבב זכרון פרטי שונה לכל תהליך, כי לכל תהליך יש PT משל עצמו.

דוגמה תקשורת בין תהליכיים: ב-**AxonU** ניתן לעשות זאת באמצעותם של אובייקטים שהתהליכים כתובים וקוראים להם, ויכולים לחכות למידע בהם וכוכו.

מה צריך בשבייל לעבוד עם ?Multi-Threading

- פעולות אוטומטיות בעת ריצת ת'רדים במקביל של אותו התהליך מאפשרות שמירה על חוקיות המידע (פעולות שאינן אפשרו לעצור במקרה).
- זכרון משותף לכמה ת'רדים, שהוא הזכרון המרכזי של התהליך, כך שכולם יכולים לגשת לאובייקטי הסנכרון.
- מערכת הפעלה שתנהל הכל, לרבות מימוש אינטראפטיסל-הערה או הרדמה של תהליכיים, שימושיים אותנו בהקשר זה לעדכו תהליכיים שונים על סיום הכנת המידע לתהליך אחר.

הרצה במקביל של ת'רדים במעבד

- מעבד עם ת'רד ייחיד מריצ' בכל מחזור שעון את אותו הת'רד, אבל כמו פקודות שלו במקביל (זו ההגדרה של **Superscalar**).
 - מעבד עם interleaving Fine-grained Multi-Threading מריצ' בכל מחזור שעון פקודה אחת או יותר של ת'רד אחד, ועשה לת'רדים ברמת המחזור.
 - מעבד עם Corase-grained Multi-Threading מריצ' תכנית עד שהוא נתקעת, והוא עובר לתכנית אחרת עד שהוא נתקעת וחזור חלילה. כמו הנו, רק שההחלפה קורת ברמת הבלוקים של קוד ולא ברמת הפקודה.
- .Fine-Grain GPU General Purpose Coarse-Grain מעבדים ב- ומעבדים ב-.
• מעבד רב-liveti מריצ' בכל ליבת (מעבד בפני עצמו) תוכנה אחרת, ולא מעורבב בין ת'רדים באותה ליבת.
- מעבד עם Simultaneous Multi-Threading מנסה לשבץ פקודות בלביות כמה שייותר צפוף, לעיתים אפילו פקודות של כמה ת'רדים באותו מחזור. למנגנון זה נדרש להחזיק וersistרים לכל אחד מהת'רדים שלו, וארכיטקטורת זה מאוד מתאגר אבל עובד מאוד טוב.

מודלים לגישה משותפת לזכרון

.1

(א) לכל המעבדים יש זכרון אחד גדול ומשותף, והמקובל בזיכרון הוא בגישת SMP. לכל אחד יש מטמון משלו מעל הזכרון, אליו ניתן כולם דרך אוטו bus, שהוא צוואר הבקבוק כאן. במודל זה קשה להריץ הרבה תהליכיים שניגשים לזכרון כי מהר מאוד מגיעים להוויה בביבוצים.

היתרון במימוש זה של מקובל הוא שהגישה לזכרון היא איחודית, ולא נדרשים פרטי מימוש מתאגרים מעבר לפתרון בעיות סנכרון. החסרו

(ב) יהיה לנו מעבד אחד שMRI'ץ הרבה ת'רדים במקביל, כך שלכל התרדים יש מטמון משותף ומשאי חישוב משותפים (לרובות זכרו משותף). היתרונו כאן הוא שבזמן שת'רד אחד ממחכה לשובה מהזכרונו, MRI'ץ ת'רד אחר.

(ג) יהיו לנו כמה מעבדים, כל אחד MRI'ץ תהליך עם מטמון שלו, והם נגישים לזכרו משותף (לוגית) דרך שטמאמש כזכרוןות שונים (פיזית) שנגישים אליהם דרך רשות (במקום bus). נוכל לגשת לכמה זכרונות במקביל דרך הרשות. כדי להוכיח את התקורתה של הרשות, נדרש בлокים מאוד גדולים ב-cache כך שה-latency בכל גישה יהיה קטן ביחס לזמן של העברת המידע עצמה.

הערה נניח שה-bus יכול להעביר בכל פעם 32 בייטים של מידע מהזכרונו. כדי להביא שורה של 512 בייטים למטרונו, ה-*bus* ייחזר לנו את המידע ב.ngנות של 32 בית, כשלכל סט של נגנות נקרא Burst (פרץ). ה-*bus* מאוד יקר חומרתי כי הוא צריך להיות מאוד מהיר על פני מרחוקים מאוד גדולים, וכן הוא עדין נחשב צוואר בקבוק.

קוורנטיות מטמון

לאחר ביצוע פעולות על כתובות בזכרו, יש במטרונו ובזכרו ערכים שונים. כשיש מעבד אחד עם מטמון אחד וזכרו אחד, זו לא בעיה. כשהיש שני מעבדים, זה יוצר מצב של חוסר-קוורנטיות - מעבד אחד קורא מידע לא מעודכן מהזכרונו שנמצא כרגע רק במטרונו. יש לנו שתי דרישות:

- **קוורנטיות סדרתית**: כל רמות הזיכרות מחזיקות באותו ערך לאוთה כתובות בזכרו (או לפחות המימוש החושך ממשק שימושיים זאת).
- **כתיבות לזכרו נעשות באופן סדרתי**: אם שני תהליכי כתובים לזכרו (קדם למטרונו שלהם), A ולאחריו B, אז הערך של B הוא זה שיוופיע בזכרו.

עקרונות במימוש קוורנטיות

- **מיגרציה**: ברגע שתהליך כותב, המידע עובר באופן שקווי לתהליכי אחרים.

- **רפליקציה**: ברגע שתהליך קורא, כל העותקים של המידע שהוא רוצה הם העדכניים ביותר.

דוגמה נմמש קוורנטיות עם bus משותף בסגנון Snoop-Invalidate. כשהتلיך יכתוב לזכרו, תשלח bus הודעת snoop שמודיעה לכל (מטמוניים של) התהליכים שהערך של הכתובת שאליה כתבו השתנה והעוטק שלהם כבר לאolid. לאחר מכן, במקרה הבאה שתהליך אחר יקרא את הכתובת הוא יראה שהכניסה במטרונו כבר לאolidית ויקרא אותה מחדש מהזכרונו.

הנחה פה היא שהמטרון הוא Write-Thru Write-Back כדי שהעדכנים של הזכרונו יהיה מיד בגישה ולא רק בגירוש.

דוגמה פרוטוקולים מוכלים לזה שמיושם במטרוני Write-Back נקראים MESI, שמאפשרים לכתובת להיות באחד מארבעה מצבים הנראים באior.

	Valid?	Owned?	Modified?
Invalid	NO	NA	NA
Shared	YES	NO	NO
Exclusive	YES	YES	NO
Modified	YES	YES	YES

תרגול

נדון בשינוי סדר פקודות בرمת הקומפיילר, כשהמטרה הסופית שלנו היא למזער את ה-nop/stall-ים שהמעבד נאלץ להכניס כדי למנוע סכנות.

פרימית לולאות

בסוף כל איטרציה של לולאה, המעבד צריך לבצע בערך שחוושב במהלך הלולאה - מקום מועד ל-stall. כדי למגר את מספר ההמתנות בסוף כל לולאה, נוכל למשם כמה חזרות בתוך כל איטרציה, כך ששה"כ תהיה כמות קתנה יותר (ביחס כפלי) מאשר מספר הלולאות המקורי.

דוגמא את השורות שמאל לחץ ניתן להמיר לשורות מימין, כאשר עתה יש לנו חמישית ממספר האיטרציות המקורי, ללא פגיעה בנכונות הקוד (אלו שורות שקולות לחלווטין).

```
for (int x = 0; x < 100; x++)
{
    func(x);
}                                →
for (int x = 0; x < 100; x += 5)
{
    func(x);
    func(x+1);
    func(x+2);
    func(x+3);
    func(x+4);
}
```

ל-loop unrolling יש חסרון מרכזי במספר הפקודות המוגדל בלולאה מעmis על מטמון הפקודות (וגם תופס יותר מקור בזכרון הפקודות). בנוסף, מספר הרגיסטרים שישמרו תוצאות ביןיהם של הקראות בלולאה יגדל (מ-1 למספר הפרימות).

הערה יש לשים לב במקרה הקצה בהם מספר האיטרציות הכללי לא מתחלק במספר החזרות שכלל ידנית בתוך הלולאה.

דוגמא נתון הקוד הבא

```
for (int i = 1000; i > 0; i→)
```

```
    A[i] = A[i] + s;
```

קוד אסמלבי אפשרי לו (קלאסי, ללא פרימה) הוא

```
Loop: lw    $t0 , 0($s0)    # $t0 = A[i]
      add   $t4, $t0, $s1    # $t4 = A[i] + s
      sw    $t4 , 0($s0)    # A[i] = A[i] + s
      addi  $s0, $s0, -4
      bne   $s0, $s2, Loop
```

נניח שאין bypass-ים, שנitinן לקרוא ולכתוב לרגייסטר באותו מחזור וב-branch freeze עד להגעת התוצאה. אילו stall-ים

המעבד יכנס?

בין השורה הראשונה לשנייה יהיו שני stall-ים, בין השניה לשלישית גם, בין הרביעית לחמישית גם וכן ולסיוום עוד אחת לאחר ה- branch - סה"כ 12 מחזורי שעון (5 במקור ועוד 7 stall-ים), כמעט פי 2 מריצה אופטימלית!

- ננסה את סדר הפקודות; נעדכו את \$s0 כבר בהתחלה ובמצע branch לפני ה-sw - לא כדאי ששנשמרת שקיילות אבל היא כן. עתה ה-all-stall-ים (ושינוי הסדר) יראו כמו:

```

Loop: lw      $t0 , 0($$0)
       addi   $$s0, $$s0, -4
       stall
       add    $t4, $t0, $$s1
       bne   $$s0, $$s2, Loop
       stall
       sw    $t4 , 4($$0)

```

עכשו ירדנו ל-7 מחזורי שעון, עם 2 עיכובים שננטה להיפטר גם מהם.

בשימוש ב-loop unrolling ניתן פקודות לפי שתי מחלקות: עובדה אמיתית (קוד מהותי שמחשב דבריהם) וטיפול באיטרציה (הזות אינדקסים והשווותם).

נרצה להמעיט כמה שאפשר בפקודות שמתפלות באיטרציה (בין היתר ע"י הטייה היחס בין המחלקות, וכשאין תלות בין שתי המחלקות זה קל מאד לעשות (כמו כאן).

- נבצע all-stall עם 4 פרימות ונקבל את קוד האSEMBLY הבא

```

Loop: lw      $t0 , 0($$0) stall
       add    $t1, $t0, $$s1 stall
       sw    $t1 , 0($$0) stall
       lw    $t2 , -4($$0) stall
       add    $t3, $t2, $$s1
       sw    $t3 , -4($$0)
       lw    $t4 , -8($$0)
       add    $t5, $t4, $$s1
       sw    $t5 , -8($$0)
       lw    $t6 , -12($$0)
       add   $t7, $t6, $$s1
       sw    $t7 , -12($$0) stall
       addi  $$s0, $$s0, -16 stall
       bne   $$s0, $$s2, Loop stall

```

עתה קיבלנו $19 = 4 \cdot (2 + 2) + 2 + 1 = 19$ all-stall-ים בכל איטרציה (באյור מוגדים all-stall-ים על הפרימה הראשונה) וסה"כ 33 פקודות, לעומת 8.25 מחזורי שעון לאיטרציה (מקורית, לא פרומה), לעומת 7 בלי שום פרימה - זו הרעה בביטויים.

- נוכל לשנות את סדר הפעולות לאחר הפרימה, ולקבל הרבה פחות all-stall-ים, באמצעות קיבוץ סוגים של פקודות (כך שבזמן שונכה לאינדקסים פרומיים מוקדמים נחשב דברים נחוצים לאינדקסים פרומיים מאוחרים יותר); ראו אייר

```

Loop: lw      $t0 , 0($s0)
      lw      $t2 , -4($s0)
      lw      $t4 , -8($s0)
      lw      $t6 , -12($s0)
      add   $t1, $t0, $s1
      add   $t3, $t2, $s1
      add   $t5, $t4, $s1
      add   $t7, $t6, $s1
      sw    $t1 , 0($s0)
      sw    $t3 , -4($s0)
      sw    $t5 , -8($s0)
      sw    $t7 , -12($s0) stall
      addi  $s0, $s0, -16 stall
      bne   $s0, $s2, Loop stall

```

כך שעתה אנחנו עומדים על 3 all-stallים (כולם קשורים לטיפול באיטרציה, ולא בעבודה אמיתית), כולם סה"כ $17 = 14 + 2 + 1$

מחזורי שעון שהם 4.25 מהזורי לאיטרציה (מקורית) - שיפור משמעותי לעומת baseline!

אפשר להוריד את מספר המוחזרים לאיטרציה מקורית ל-5.5 ע"י העברת ה-addi להתחלת שינוי היחסטרים המחשבים ב-,lw

.sw

נשים לב שמספר הרגיסטרים הנחוצים לחישוב גדול משמעותית ביחס ל-baseline. אילו הקוד בתוך הלולאה היה דורש 3 פקודות לפחות, היו צריכים למצוא רגיסטרים נוספים או להשתמש בזכרון - ריבב נוספת בחלוקת הביצועים בעת ביצוע unrolling loop.

הערה נזכיר שמערכות דואות מדדיים נשמרים ב-major row, ככלمر כל הערכים של השורה ראשונה [...]A [...]B שמורים באופן רציף, ומיד לאחריהם הערכים של השורה השנייה וכן הלאה. תחת קונפיגורציה כזו, תמיד יותר יעיל לעבור לפישורות בלולאה החיצונית ועמודות פנימית כאשרוצים לקרוא את כל המערך, ולא להפוך.

דוגמה נתונה הפ' הבאה שרצה על חומרה עם מטמון בגודל 32KB בסכמת FA עם אלג' גירוש LRU ובлок בגודל 64 בתים

```

int A[2048];
int, B[2048,2048 ], C[2048,2048 ],

For (i=0, i<2048, i++)
{
    A[i] = 0
    For (j=0; j<2048. j++) //loop1
        A[i] += B[i,j]*C[j,i];
}

```

- בהנחה שהמערכות מיושרים בכתובותיהם לזכרון, מה הה-miss rate בגישה לננתונים?

נתעלם מהഫסוסים בכתובות ל-A (מדובר בסדר גודל של $\frac{1}{2048}$ ביחס למספר הפגיעה סה"כ). על B אנחנו עוברים בשורות רציפות,

ולכן יש פספוס פעמי 16 גישות (כל int טופס 4 בתים, סה"כ 16 int-ים בבלוק של המטמון).

ל-C אנחנו ניגשים בכל פעם לשורה אחרת, ויש 2^{10} גישות לכל? לעומת זאת 2^9 כניסה במטמון ולכן לא נצליח לשמור בלוקים במטמון

על אותה שורה בין איטרציות, כך שכל קריאה היא פספוס. לשני המרכיבים אנחנו ניגשים 2048 פעמים בכל איטרציה, כולם miss

$$\cdot \frac{\frac{2048}{16} + 2048}{2048 + 2048} \approx 53\% \text{ של rate}$$

חשיבותו של תוצאת ה-branch predictor ב-2048 המהמיטמוון (אילו היו 2 גישות ל-C לעומת 10 גישות ל-B). הרעיון הוא שבלוק ה-branch predictor יתבצע עד לגישה הבאה אליו.

- זמן הריצה של הקוד במעבד pipeline ביחס ל-branch predictor הוא אידיאלי (כל קופץ נכוון תוק מחזורי אחד ולא נדרש flush), יש hit rate של 100% במתומו הפקודות, וכל לולאט זמוקודדת כ-7 פקודות branch אסמלבי (corollary-stall?)?

זמן הריצה הוא

$$(\#i - \text{iterations}) \cdot (\#j - \text{iterations}) \cdot ((\#instructions \text{ per } j - \text{iteration}) + (\#memory \text{ accesses per } j - \text{iteration}) \cdot (\text{miss rate}) \cdot (\text{miss penalty}))$$

(מצ Nichols את זמן הגישה לזכרון במקרה של פגיעה) ובהתאם הערכים נקבל

$$2048 \cdot 2048 \cdot (7 + 2 \cdot 0.53 \cdot 10) = 2048^2 \cdot 17.7$$

הדוגמה הנ"ל מראה כיצד חישוב פעולות ביןאריות על מטריצות היא יקרה מאוד בשל ניצול גרווע של המטמוון. ריצה לנצל מספר רב של מעבדים כדי למקבל את הפעולה הרצויה, ולנצל באופן מיטבי יותר את המטמוון.

דוגמה נוספת שמשתמשה ב- miss penalty של 100 מחזוריים ו- RAM latency של 100 מחזוריים.

השינוי היחיד בחישוב הנ"ל הוא ב- miss penalty , כאשר עתה בפועל העדשה מ-10 ל-100 נקבעה כ- $2048^2 \cdot 113$. עבור ליבת אחות. בнерמול למספר הליבות, נקבעה כ- $2048^2 \cdot 1.13 \cdot \frac{2048^2 \cdot 113}{100} = 15.6$.

OOOE

ריצה להאיץ ביצועים של תוכנה שישורות הקוד שלו קבועות, באמצעות הריצה לא סדרתית של הפעולות בה. נניח שיש לנו מעבד pipeline עם כמה עותקים לכל שלב (לדוגמה כמה שלב פענו שיכולים לרוץ במקביל). נוכל לסדר מחדש את השורות כך שכמה פקודות נדרשות פחות שלבים יריצו בזמן שפוקודות ארוכות רצוטות, וכך נשיג ניצול יותר גבוה.

דוגמה בהינתן שפעולות חלוקה דורשת הרבה זמן לחברו, את השורות הבאות ניתן למקבל (להתחיל את החלוקה, ובזמן זה להריץ את שתי הפעולות החיבור)

```
div $t1, $t2, $t3
add $t4, $t5, $t6
add $t7, $t8, $t9
```

ואם השורה השנייה מוחלפת ב-\$t1, \$t5, add \$t4, \$t5 (כלומר משתמש בתוצאת החישוב של פעולה שעדיין לא הסתיימה), ניתן לה לחייב שעדיין נrix את השורה השלישית במהלך הזמן הזה.

דוגמה נבייט בקוד הבא

```
div $t1, $t2, $t3
add $t4, $t5, $t1
add $t5, $t8, $t9
```

לפי התיאור הנ"ל, הפעולה השלישית תסתמיכים לפני השניה (כי זו עדין מחייבת תוצאת החילוק שאורכת זמן רב) ועתה האופרנד הראשון בשורה השנייה מושפע מהשורה השלישית - זו סכנה חדשה!

סכנות דאטה חדשות ב-OOOE

- Write After Read - כשלעצמה כותבת לריגיסטר לאחר שפקודה אחרה קורת ממנה. בריצה לא חוקית של הקוד בשימוש ב-OOOE. יתכן מקרה בו הפקודה המוקדמת תקרא את הערך החדש שנכתב ע"י הפקודה המאוחרת (סקנה זו יכולה לקרות אם לא משתמשים ב-OOOE).
- Write After Write - כשתwei פקודות כותבות לריגיסטר, ובשל השימוש האפשרי בסדר הרצטן ייתכן שהערך האחרון שיכתב לריגיסטר הוא של הפקודה המוקדמת (גם זו ייחודית לשימוש ב-OOOE).

דוגמה נבייט בקוד הבא

```
div $t1, $t2, $t3
add $t5, $t6, $t1
add $t9, $t5, $t5
add $t5, $t7, $t8
```

נשים לב כי סכנת-WAW שנוצרה אינה אינטראינית לקוד, משום שرك במקרה שתי הפקודות כותבות לאותו ריגיסטר, ולא בגלל שיש ביניהן תלות. התנשויות סתמיות כאלה נקראות False Dependencies. בפרט, עם אינסוף רגיסטרים ניתן היה להימנע מכל הסכנות הדאטה תוך שימוש ב-OOOE שנובעת מ-False Dependencies.

Register Renaming

שיטת Register Renaming בא להפטור תלויות מדומות בלי אינסוף רגיסטרים. בשיטה זו, נציג למשתמש מממשק (וירטוואלי) של רגיסטרים ארכיטקטוניים \$R0,...,\$Rn, כאשרמציאות החומרה מספקת לנו רגיסטרים פיזיים \$T0,...,\$Tm. לכל רגיסטר ארכיטקטוני נוכל להתאים יותר מרגיסטר פיזי אחד:

- בכל פעם שפוקודת תבקש לבצע חישוב על בסיס רегистרים ארכיטקטוניים כלשהם - נתיק את אנדקס ה-R בפוקודת לאנדקס T כדי שהערכים מפוזרים ברגיסטרים הפיזיים בפועל (קורה בשלב Fetch/Decode). כמובן, ניתן למעבד לחשב את הפעולה על בסיס רегистרים שמספריהם שונים מהמקור, אבל שמתנהגים באופן זהה.

- בכל פעם שפוקודת תבקש לכתוב לרגיסטר ארכיטקטוני כלשהו - נעדכן את העתקה כך שכל אינדקס T ידע לאיזה R הוא מתייחס (קורה בשלב Commit, הכתיבה חוזרת לרגיסטרים). בפועל מדובר באולו' שינוי ה-R שמתמפה (בין היתר) ל-T הזה.

הערה היתרון בשיטה זו הוא ש-T-ים שונים יכולים לשמר ערכים שונים של אותו R לאחר מכן ה-WAW או WAR בغالל תלות מדומה, אבל בזכות הפיזור ההגיוני יותר של הרגיסטרים שומרם על חוקיותם.

הערה החוקיות נשמרת ב-Register Renaming Execute Commit Fetch/Decode כך גם את ה-T, וכך בסדר הנכון, וכך גם את ה-R, יבוצע (אולו') בשינוי סדר אבל שמירת הנתונים תהיה נכונה.

דוגמה נניח שאנו מרכיבים את השורות הבאות

```
div R1, R2, R3
add R2, R4, R1
add R2, R3, R3
add R4, R3, R2
```

בהרצאה רגילה של הקוד עם OOOE, אנחנו חשופים לסכנת WAW בין שורות 2 ו-3 וסכנת WAR בין שורות 2 ו-4.

עתה נניח שאנו משתמשים בשיטת Register Renaming עם ארבעה רегистרים פיזיים.

- את התוצאה של הפעולה הראשונה נשמר ב-T0 (נתעלם מהשאלת מאיפה ורגיסטרי ה-R באים בפעם הראשונה).
- את התוצאה של הפעולה השנייה נשמר ב-T1, והיא תתבסס על סכמה של R4 ו-T0, כי לשם מיפויו את תוכנת הפעולה הראשונה.
- את התוצאה של הפעולה השלישי נשמר ב-T2, והיא תהיה סכום של R3 ו-T2.
- את התוצאה של הפעולה הרביעית נשמר ב-T3, והיא תהיה סכום של R2 ו-T3.

בזכות המיפוי הנפרד של ה-R-ים בשורות 2 ו-3, השורה הרביעית תקבל בהכרח את הערך שהוא אמור לקבל (הלא הוא תוצאת החישוב בשורה השלישית). בדומה, R4 יילקח מיפויו בשורה 2, שהוא בהכרח לא T3, אליו תיכתב תוצאה החישוב בשורה 4. כך מנענו את שתי הסכנות שהיו כאמור תלויות מdomo.

דרך נוספת למימוש Register Renaming הוא באמצעות Reorder Buffer (ROB), שמקבל הוראות Decode לפי הסדר ושומר אותן כך שהכנסה בה נמצאת פוקודת היא הרגיסטר הפיזי אליו היא צריכה לכתוב. בשלב Execute-T ההתוצאה תיכתב לכניסה המתאימה, אולי לא לפי סדר כרונולוגי. לסיום, ה-T-ים יבוצעו לפי הסדר, ככל מר שפוקודת לא יכולה לעשות Commit לפני שקודמתה עשתה כן, וכך כאמור נשמר על נכונות.