

מבנה המחשב | 67200

הרצאות | אוהד פאליד ורונ גבר

כתביה | נמרוד רק

תשפ"ג סמסטר ב'

תוכן העניינים

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

52	מעבד V Single-Cycle
52	הרצאה
52	קידוד פקודות MIPS
53	יצוע פקודות ב-MIPS
54	שימוש השלבים ב-MIPS
58	תרגול
60	מעבד VI Multi-Cycle
60	הרצאה
62	יחידת השליטה
63	מעבד רב-מחזורי
64	שיטות השוואת יצואו מעבדים
66	איחודי ייחדות במעבד רב-מחזורי
66	מספר המוחזוריים לפי סוג פקודת
71	תרגול
71	דגלי ייחידת הקרה
74	מעבד VII pipelined
74	הרצאה
78	סכנות במעבד pipelined
78	פתרונות לסקנות
80	סקנות ב-branching
81	תרגול
81	שינויים מרכזיים ברכיבים תחת מעבד רב-מחזורי
81	אותות בקרה למრבבים במעבד רב-מחזורי
82	אותות בקרה חדשים במעבד רב-מחזורי
84	הערכת יצואים
85	הטמון VIII
85	הרצאה
86	טרמינולוגיית מטמון
87	מטמון Fully-Associative
88	מטמון Direct-Mapped
89	מטמון 2-Way אסוציאטיבי
90	תרגול

IX	עד על המטמון
91	הרצאה
91	סוגי פספסי מטמון
91	ביטויים של מטמון
92	שקלול תמורות גודל הבלוק
92	מטמון רב-שלבי
93	יעולים נוספים לגישות לזכור
94	מדיניות גירוש
94	מדיניות כתיבה למטמון
96	תרגול
X	זיכרון וירטואלי
97	הרצאה
97	TLB
98	יעולים לתרגומים כתובות וירטואליות
99	טבלת דפים היררכית
100	אינטראפטים
100	תרגול
101	מהלך קרייה מהזיכרון
102	
XI	חישוב במקביל
103	הרצאה
103	טקסונומיה של מקובל
103	heteksonomia של Flynn
104	סנכרון ת'רדים
104	מודלי זכרון משותף-ב- Multi-Processing
105	מה צריך בשbill לעבד עם ?Multi-Threading
105	הרצאה במקביל של ת'רדים במעבד
105	מודלים לגישה משותפת לזכור
106	קוורנטיות מטמון
106	עקרונות בימוש קוורנטיות
106	תרגול
107	פרימיט לולאות
107	OOOE
111	סקנות DATA חדשות ב-OOOE
111	шиיטת Register-Renaming
112	
XII	האצת ביצועי ת'רד יחיד
113	הרצאה
113	VLIW
115	מעבדים וקטוריים
116	
119	תרגול

123

OOOE XIII

123

הרצאה

124

האלגוריתם של Tomosulu

127

Branch-Prediction

127

סוגי ים-Predictor

129

תרגול

שבוע II | מבוא ומוליכים-למחצה

הרצאה

המחשבים הראשונים היו עצומים, כבדים ויקרות, ויחידת הבסיס של המעבד שלהם הייתה שפופרת קטודיות (אבי הטרנזיסטור) ואוז שפופרות 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.

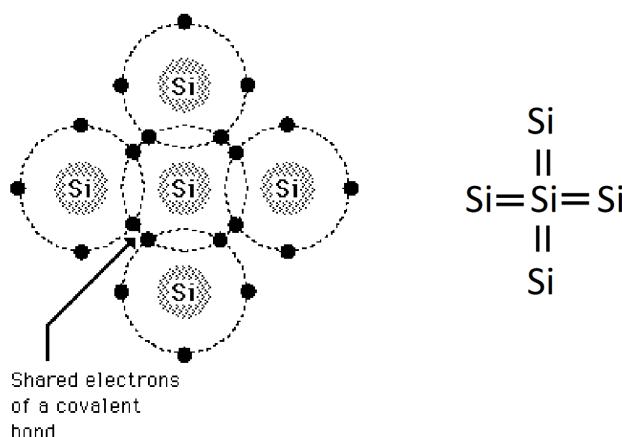
רקע כימי לטרנזיסטורים

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 חמישה 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

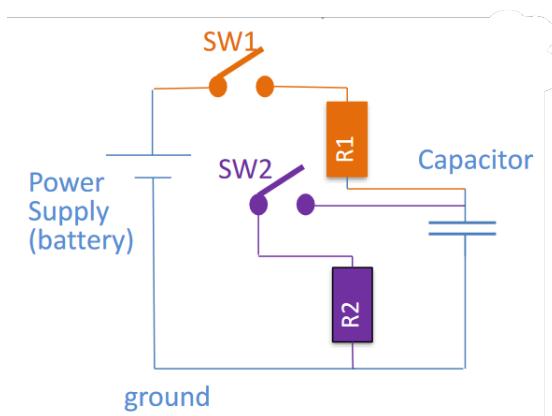
• זרם חשמלי הוא תנועה של אלקטرونים (או באנלוגיה תנועה של חורים, כאשר האלקטרונים השיליליים עוברים בין חורים אחרים חיוביים). בקונבנצייה הזרם נע מה-+--.

• מתח (פוטנציאל) נמדד בולט והוא היכולת להזורים, כאשר סוללה עצם מחזיקה מתח וכמו מגדל 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 סגור את המתג הראשון, סגור מעגל בין הסוללה לקבל כך שאלקטרונים יזרמו מהסוללה לקבל מלמעלה ומהקבל לסלולה מלמטה והקבל יקבל את אותו המתח של הסוללה. אם ננטק את המתג הקבל ימשיך להחזיק את המתח, ואם נדליך 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 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 במקודם לכלא אחד, ואז יקפוץ לאחר, וכו'). החורים שנוצרים בשאן את האלקטרון הנדרש ליציבות נעים (לשם התיאוריה, בכיוון ההפוך מהאלקטטרונים 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.

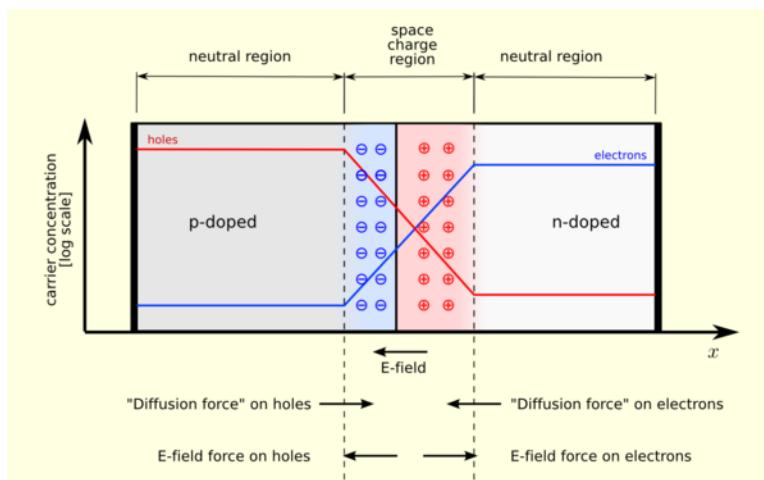
הצמדה של חומרים מסוג P ו-N יוצרת יונים - האלקטרונים מ-N שמחים לקפוץ ל-P שם "צרי" אותם. שני כוחות פועלים בעת הצמדת חומרים שכזו:

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

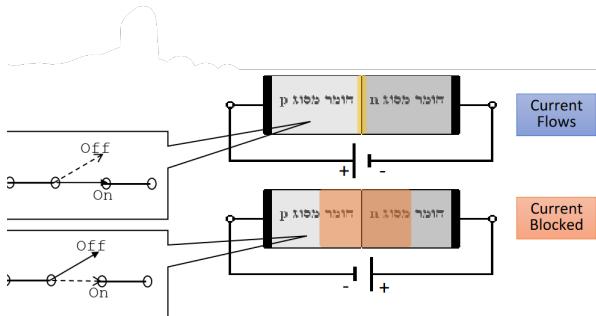
צומתpn

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.



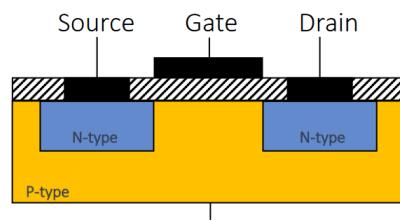
- חיבור סוללה עם + ל-P ו- – ל-N יגרום להרבה מאוד יוניים חיוביים לזרום מ-P ל-N ויוניים שליליים מ-N ל-P (בהתאם לכיוון הזרימה לפני שנחסמה) וכך יצטמצם המרווח באמצעותו עד ל-0 ותהייה לנו זרימה רגילה, כאילו סגרנו מתג.
- חיבור סוללה עם + ל-N ו- – ל-P יגרום ליוניים חיוביים ושליליים להגיע לחומרים ולהזקק את הioniים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

MOS-FET

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

N-Channel 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.

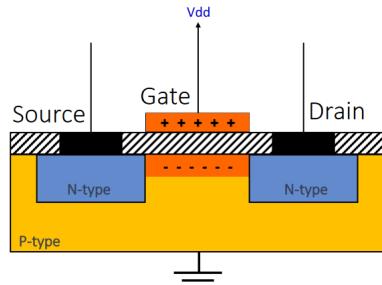


זכור כי זרימה ניתנת לקבל רק מ-P ל-N ולכן אי אפשר אף פעם להזרים שום דבר מהמקור (source) לשפך (drain) ולהפוך. יש לנו למעשה שתי דיזודות גב אל גב (שקבוצותיהן המקוור והשפך).

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 המתג 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		
P-MOS		

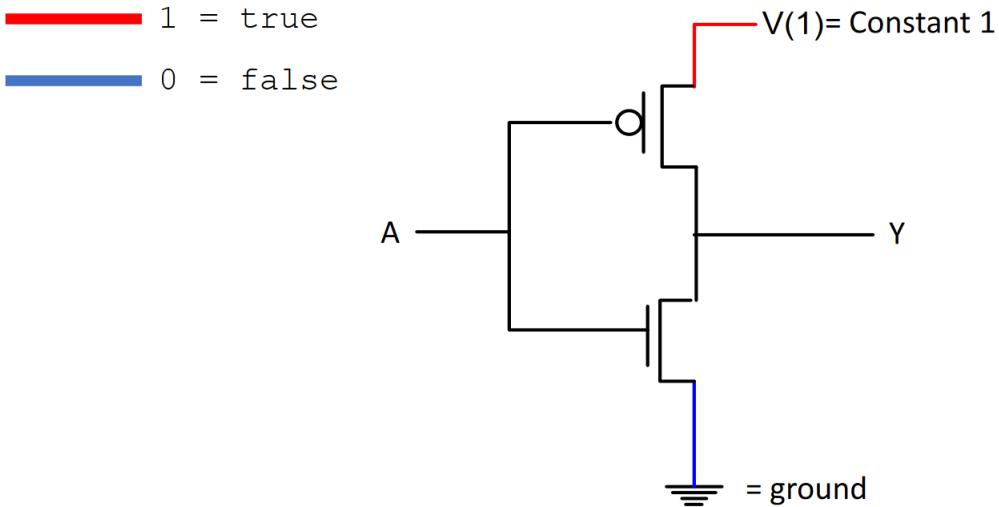
הערה בתחום הכלל, ○ מסמן שליליה, **ב-**PMOS למשל “שוללים” את המתח ומתנהגים הפוך ממנו. גם שערי NOT מסומנים עם עיגול.

בנייה של שערים 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) למתח ו-ים (N-Channel למתח), והיא כבאיור.



הוא הקלט ו- Y הוא הפלט. כשה- A הוא "1", יעבור זרם של 0 (שהוא זניח) דרך ה-N-Channel למתה (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 שאנוחנו 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) שערכו 0 או 1, בעשרוני 0 עד 9, ובhexadecimal 0 – F. $A = F$

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

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

פעולות חשבוניות אפשר לעשות באותו האופן כבסיס עשרוני (חיבור ארוך שבו עבררת 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 תחליך 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 משעמים לי.

במקרה הכללי נמיר שניי בסיסים כלשהם דרך בסיס 10 כאשר את הרכיב משמאלו ומימין לספרה העשרונית נטפל באופן נפרד זהה (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.

$$\text{דוגמה } 5 = 5 - 4 = (+9) + (-4) = 100100 + 11011 = 01001 + 11011 = 00100 \text{ זה הופך ל-00101.}$$

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 לכת מימין לשמאל עד ל-1 הראשון, לא לשנות אותו, וואז להפוך את כל מה שמשמאלו (ואז לא צריך להוסף 1).

דוגמה למה שווה 5 – בשיטת המשלים ל-2: $00101 = 5$, 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

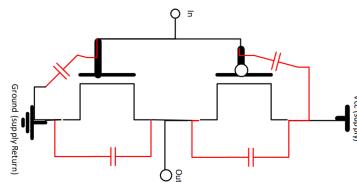
הרצאה

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

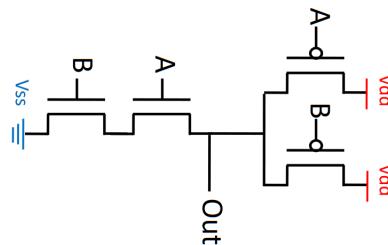
הערה לעיתים לא נרצה להזרים זרם אל תוך טרנזיסטור אחר דלوك (מהכיון הלא נכון) כי זה גורם לסתור.

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

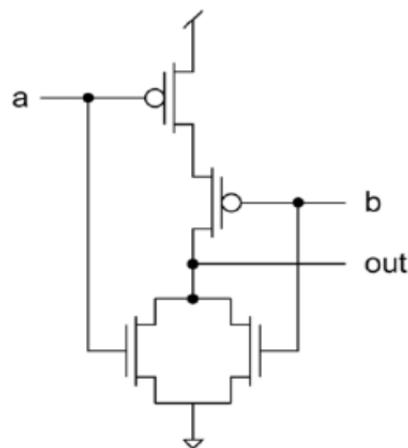
בין שער למקור, בין שער לשפק ובין שער לניצרים קבליים אפקטיביים (לא הוספנו שום דבר, פשוט יש רכיבים פיזיים שמוצאים עצמם מחזיקים מתח בין הצדדים). את הקבליים האלה לוקח זמן לטעון או לפרק מתח בעת שינוי מצב (שינוי הזרם מהשער, הזרם מהמקור). לכן במצבים רגילים יש זרימה שאנחנו לא בהכרח מצלפים לה במהלך המעבר ($10\text{--}15\text{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



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' שביחסו y, x , פולט 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 פעם אחת כליטרל או בתוך ליטרל מהופך.

דוגמה עבור שלושה משתנים וההשמה $x = 1, y = 1, z = 1$ המינטרם היחיד שערכו 1 יהיה $m_6 = xyz'$, עבור $x = 1, y = 1, z = 0$ יהיה $m_5 = xyz$ (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 $M_6 = x' + y' + z'$ וכוכ'.

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 שמקבלים 1 בפ' (בכחול).

לחולופין יוכל לייצג את הפ' עם PoS ע"י הכפלת כל ה-h-maxterm-ים שמקבלים ערך 0 (באדום) כלומר

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

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

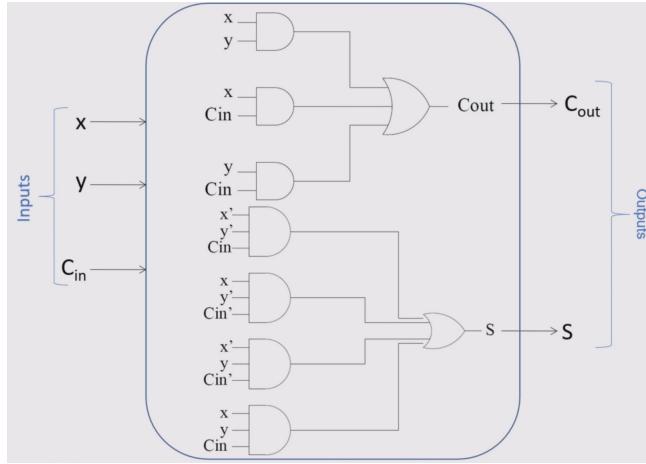
וניתן לרשום את הפ' כ- $C\text{-MUX}$, ואת זה ניתן למשם באמצעות שערים שכבר בנינו. עם זאת, ניתן למשם את המעגל עם פחות טרנזיסטורים מאשר בימוש נאיבי עם שני AND-ים, OR-ים ו-NOT.

דוגמא הוא מעגל שמקבל x, y, C_{in} כאשר S, C_{out} נשא מסכימה קודמת (כאשר 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), ואירועה הבא צמצמו כמה minterm-ים באמצעות מנות קרנו שנלמד בהמשך (בנוסף מופיע במעגל OR עם ארבע כניסה - הסתודנטית המשקיפה תראה כיצד ניתן לעשות זאת עם פי 2 טרנזיסטורים ממספר הכניסות).



הערה קל לשדרר כמה FA-ים כדי לקבל מעגל שסוכם מספרים עם מספר ביטים גדול יותר (לוקחים את C_{out} של ה-FA על זוג הביטים הראשונים, 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 $x = (s_{n-1} \dots s_0)_2$

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

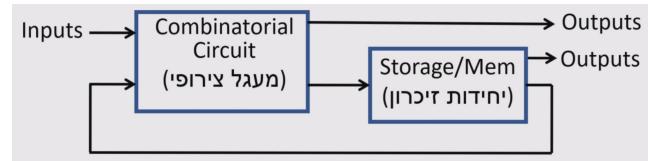
כלומר בהינתן בית, מחזיר מחרוזת שבה הכל אפסים חוץ מאולי הבית ה- $(s_{n-1} \dots s_0)_2$ שערכו x .

3. **מקודד (Encoder)**: הקלט הוא n ביטים x_0, \dots, x_{k-1} והוא $k = 2^n$ כאשר $e_i = 1$ עבור i אחר בדיק (וכל השאר אפסים) או $e_i = 0$, כלומר מחזיר את האינדקס (בבינארי) של הבית היחיד הדלק בקלט.

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

מעגלים סדרתיים

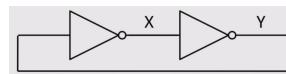
הגדרה מעגל סדרתי הוא מעגל קומבינטורי שחלק מהקלטים שלו הם פלטים של יחידות זיכרון שמחזיק השער (ראו איור)



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

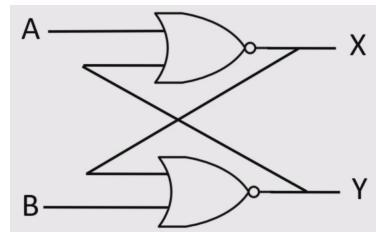
הגדרה שעון סיגナル בצורת גל מחזורי (0 או 1 ואז 0 ואז 1 וכו').

דוגמה כיצד השער הבא יתנהג?



אם הקלט בהתחלה הוא $X = 0, Y = 1$ אז $X = 1$ ו $Y = 0$. בדומה עבור $(X, Y) = (0, 1)$. נקבע מצב יציב של $(X, Y) = (0, 1)$. לעומת זאת, אם השער דואלי-יציב (עם שני מצבים יציבים), שיכולה לשמש אותנו לאחסון זכרון.

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



הפ' הזו מקיימת $X(t+1) = (A + Y(t))'$, $Y(t+1) = (B + X(t))'$ (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 נשLOW 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 לשולוט 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

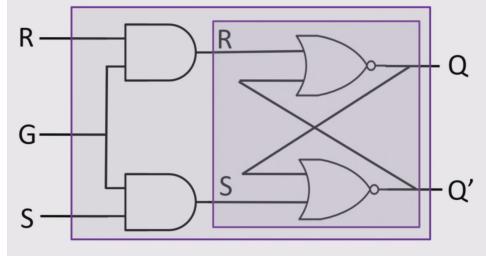
הערה מעגל סדרתיים ניתן לייצג באמצעות טבלת עירור (הטבלה הנ"ל) וטבלת מעברים, שעונה על השאלה "אילו קלטים נדרשים כדי לעבור בין מצב כלשהו לאחר" (Φ הוא ערך Don't care)

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

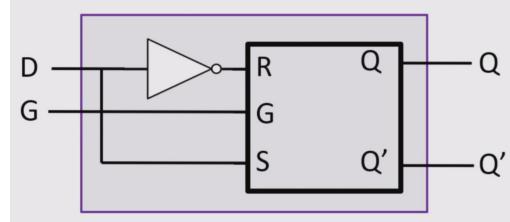
הערה יש היגיון בשמות הביטים S,R - אם רק (et) S דлок, קובעים את הפלט להיות 1, אם רק (reset) R דлок קובעים את הפלט להיות 0, ואם זה ולא זה זלקיים לא עושים כלום, ככלומר שומרים על המצב כמוות שהוא. כמוון שתחת סימולרים אלה, גם S וגם R זלקיים זה לא מוגדר היטב.

מעגלים סדרתייםים מורכבים על בסיס SR Latch

הוא שער שמנוע פליטת ערכאים לא חוקיים M-SR Latch, והוא מקבל קלטים S,R,G-ו-G' 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)

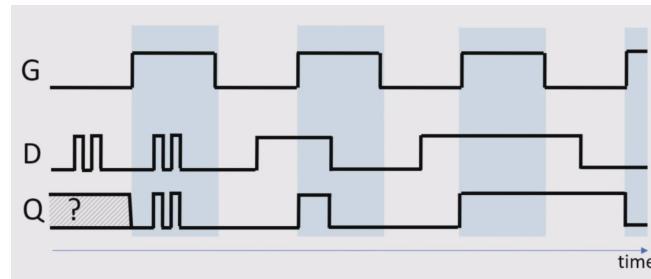


• **Gated D-Latch** הוא גרסה אפילו יותר בטוחה ל-SR Latch - במקומם לקלט קלטי R,S, נקלט D שייהיה מחובר ל-S ודרך מהפ' ל-R, וכן לא נקלט מצב של (1,0) (0,1), וכך ניתן פשוט לכבות את G (ראו איור)



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

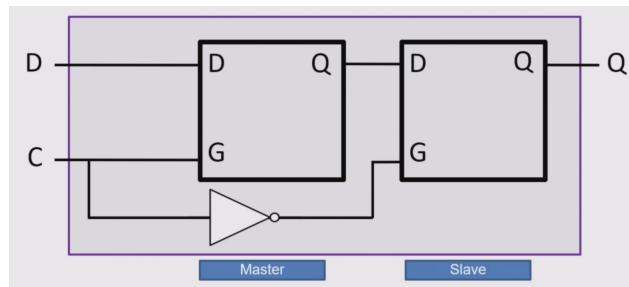
דוגמא בהינתן שעון שמחובר ל-G ובית D שערכו משתנה, נוכל לחשב מה יהיה הפלט Q, כפ' של הזמן. השטח המוקוקו בהתחלה פירשו שלא ידוע לנו מה הערך של Q שכן הוא לא מוגדר בשלב זהה.



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 מהפ' 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 הירידה 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 הזזה 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 משלים (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 הצמצום:** $x(x + y) = x, x + xy = x$.
- $(x')' = x$.

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 ליטרל 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 לכטוט את כל הריבועים. הטבלה היא מעגלית ולכן מלבן יכול לחצוץ 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 השבילינה 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 $w'z'$ והרביעייה 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 ונitin להשתמש בערך שיוטר נוח לנו אליו כך שיבוטלו ליטרלים רבים ככל האפשר. נסמן צירוף כזה ב- \emptyset .

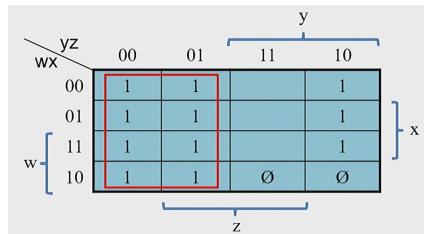
דוגמה $(1, 1, 1)$ הינה הפ' הבוליאנית והצירופים האדישים הם $d(x, y, z) = \sum(7)$ (כלומר רק 7).

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

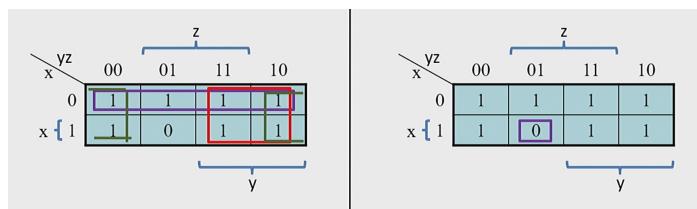
עתה נוכל לבחור מלבנים יותר נוחים (ראו איור) מאשר בהיעדר היצירוף האדיש כי אז לא היינו יכולים לבחור את השורה התחתונה כמלבן והוא לנו אמם עדין שני ביטויים אבל עם יותר ליטרלים, ככלומר יותר שערירים שזה פחות טוב.

הערה יצירוף אדיש מתקבל 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 מצטמים מכפלה של סכומים. נביט ב-(5) $F(x,y,z) = \sum (0,1,2,3,4,6,7) = \prod (5)$. כפי שניתן לראות, מצטום של סכום המכפלות דורש לפחות 3 נסכמים ואילו מכפלה סכומים דורש בדוק ליטרל אחד.



כשמצטמים פ' בוליאנית לפי מכפלה סכומים, מבצעים בדוק את התהילה של סכום מכפלות רק שסכום 0-ים במקומות 1-ים. לאחר הcisivo, ממירם את הcisivo למכפלה של סכומים, כאשר כל סכום מתאים למלבן אחד.

הערה ניתן להוכיח נכונות של מצטום לפי מפת קרנו למכפלה סכומים או באמצעות דה-מורן למצטום של סכום מכפלות, או פשוט באופן ישיר מטבלת האמת ונכונות ייצוג 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	00	0	Ø	0	0
	11	00	1	1	1	Ø
10	10	00	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	00	0	Ø	0	0
	11	00	1	1	1	Ø
10	10	00	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	00	0	Ø	0	0
	11	00	1	1	1	Ø
10	10	00	0	Ø	1	Ø

פונקציות שלמות

הגדירה קבוצה F של פ' בוליאניות נקראת שלמה אם ניתן למשתמש בכל פ' בוליאנית בעזרת פ' בקבוצה.

משמעות $\{+, \cdot, '\}$ היא שלמה.

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

מסקנה $\{+, \cdot, '\}$ הן שלמות.

■ **הוכחה:** עם דה-מורן אפשר לifyitz $+$ עם \cdot ו- $'$ עם $+$.

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

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 שכן יש לנו.

שבוע III | תזמון מעגלים

הרצאה

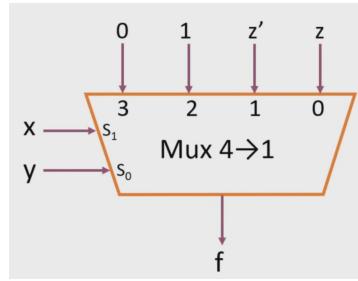
דוגמה נתונה הפ' עם טבלת האמת הבאה

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 ולהציב 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וק ערכי בטבלת האמת של f .
- עתה נמשח עם 4 MUX $\rightarrow 1$ יחיד ועוד שער אחד בלבד. כאן צריך יותר להתחמץ. 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 יחד 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.

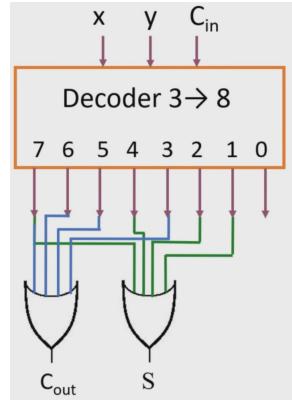
דוגמה נממש FA (שמקבל S, C_{out} ופלוט x, y, C_{in}) שיש לו את טבלת האמת הבאה (לצורך נוחות) עם :

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} , ואז 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)

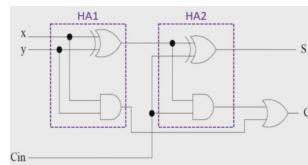


$\bullet .8 \rightarrow 1\text{-MUX}$

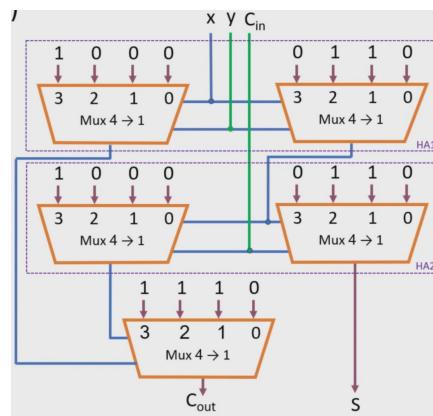
נבחר את הסלקטורים להיות x, y, C_{in} ונסבץ במסונת הקלטים של ה-MUX בטלת האמת לכל צירוף של הסלקטורים (הקלטים), ואותו הדבר שוב עם C_{out} .

$\bullet .4 \rightarrow 1\text{-MUX}$

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)

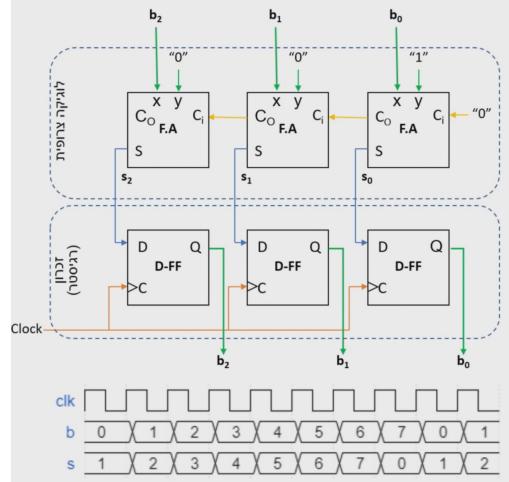


לאחר ראיינו DFF, שמקבל קלט D ושעון ומשנה את פלטו בעת עליית השעון לערך של D (זו וריאציית ה-Positive Edge) נוכל על בסיס יחידה זו לבנות יחידות יותר מורכבות.

דוגמאToggle FF מקבל T ושעון ומחשב בכל עלייה שעון $Q(t+1) = \text{XOR}(T, Q(t))$ יחד 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 -ה הפנימי.

דוגמה נממש ספרן, שסופר את מספר עליות השעון.



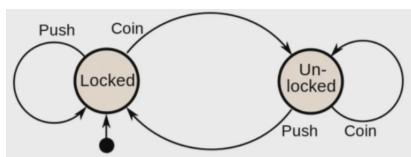
הweeneyו כאן הוא פשוט אבל המימוש לא כל כך: בכל עלייה שעון, ה-FA הימני ביותר מכניס עוד ערך 1 לסכום שמווחק ע"י כל המעלג. הסכום מופיע בכל עלייה שעון ל- DFF הבא (זה-FA המתאים לו), וכך ה- DFF-ים מחזקים שלושה ביטים שמיצגים מספר שערכו עולה באחד בכל עלייה שעון (הסתודנטית המשקיעה תרים את שלושת המחזוריים בראש/על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

ונכל לצייר את ה- FSM עם גראף

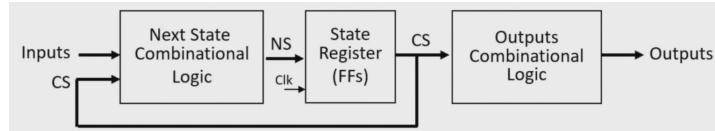


או טבלה (יש כמה דרכים)

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

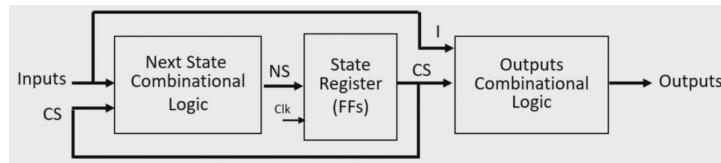
State	Output	Inputs	
		Coin	Push
Locked (Init)	Closed pass	Unlocked	Locked
Unlocked	Open pass	Unlocked	Locked

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



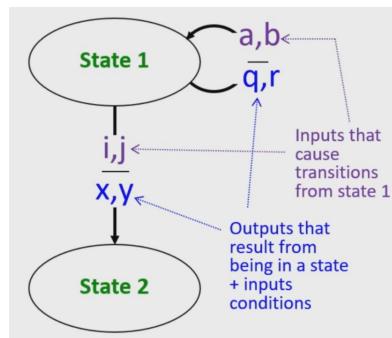
נשים לב שהרגיסטר שמורכב מ-FF-ים מחזיק את המצב הנוכחי, וערכו מחווט חוזרת פנימה לחישוב המצב הבא שיכנס לרגיסטר במחזור הבא.

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 משורה. את הייצוג הגרפי של מכונת Mealy מצירירים כמו אוטומט ריגיל, רק שעל החיצים (ה מעברים) נוסיף את הפלטים שהקלטים על החץ יחד עם המצב שעוברים אליו משורים (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 המשורה פלט.



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

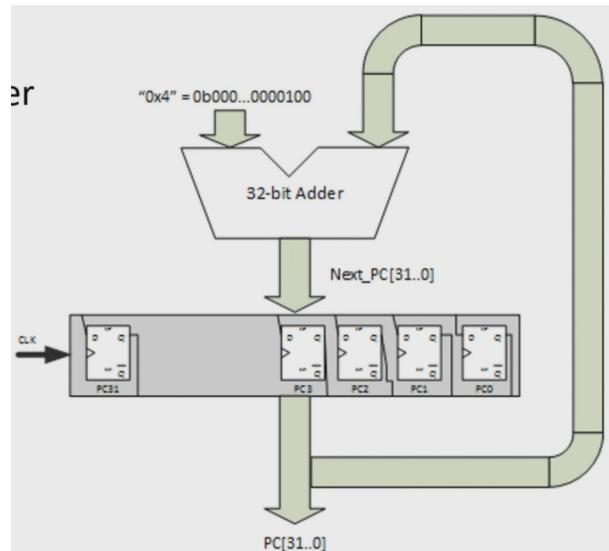
דוגמה נזכיר לדוגמת השער המסתובב. נגדיר 0 השער נעל ו-1 הוא פתוח. לכן המעגל של פ' הפלט הוא חוט ישיר מהמצב לפלט כי הפלט זהה למצב. את המעברים ניתן לראות בטבלת האמת הבאה

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 (בית הוא 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 מנגנון 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%.

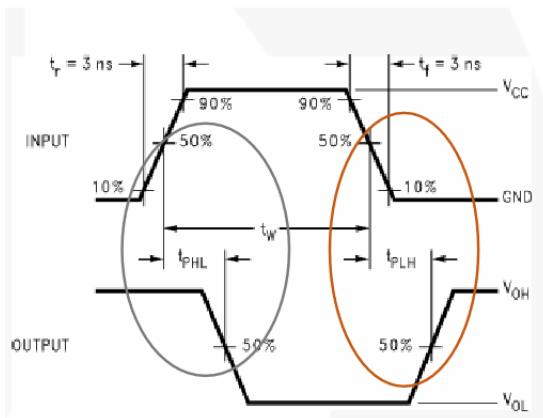
• **זמן עלייה** (t_r) : הזמן שלוקח לעלות מ-10% מתח ל-90% מתח.

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

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

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

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

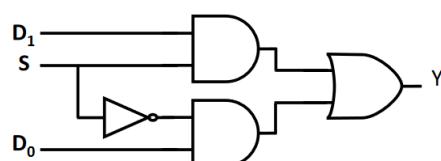


נשים לב שהשינוי הוא לא מיידי. במקרה הזה השינוי הניכר בקלט הוא חצייתו (מלמעלה או מלמטה) של הקולט את רף 50% המתח.

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}) שmbטיח 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?

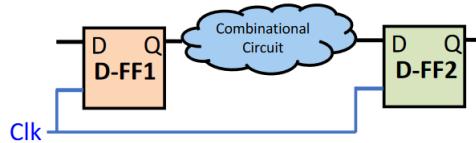
עתה נבחר את המסלול הקצר ביותר, שהוא כמובן $D_1 \rightarrow Y$ או $D_0 \rightarrow Y$ שדורש $4ns + 5ns = 9ns$ וזו

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 במשך 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

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

כדיizzare קודם לחוכות שהפלט של DFF1 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 DFF2 לאחר העליה.

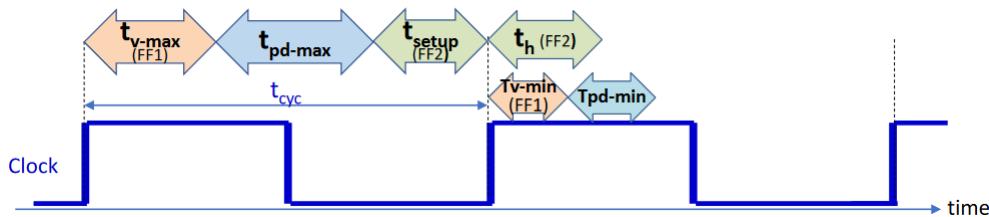
- מה צריך להיות t_h כדי שהשער יהיה תקין?

בפעם השנייה שבה ישנה הערך, נדרש ש- t_h של DFF2 יהיה פחות 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 לחתוך בחשבון שלפני תחילת המוחזר הבא נדרש ש- D יהיה יציב לפחות t_{setup} .

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

- 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 $t_h = 5 \leq t_h \leq t_{v(min)} + t_{pd(min)} = 2 + 9$.

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

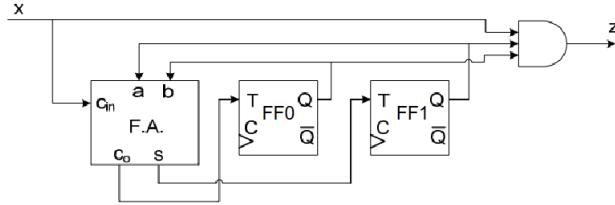
עד כה הטענו מהקלט האמייתי של המוגל שמחובר ל-S1. הוא עצמו גם חייב לקיים דרישת שינוי שלו ביחס לשעון וכן להתייחס אליו כסיגナル שמגיעה מפלט אחר והניתוח יהיה כנ"ל.

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 לפט 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 דרכו, 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 : 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

הגדלה מעגל צירופי (קומבינטורית) הוא מעגל שיש לו כניסה ויציאה כאשר האחרונות תלויות בכל ערך ורגע של הראשונות. מעגל סדרתי הוא מעגל שפלטו תלויים גם ביחסית זכרו שמחוברת לשעון.

דוגמה כיצד נמשח 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 לממש 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.

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

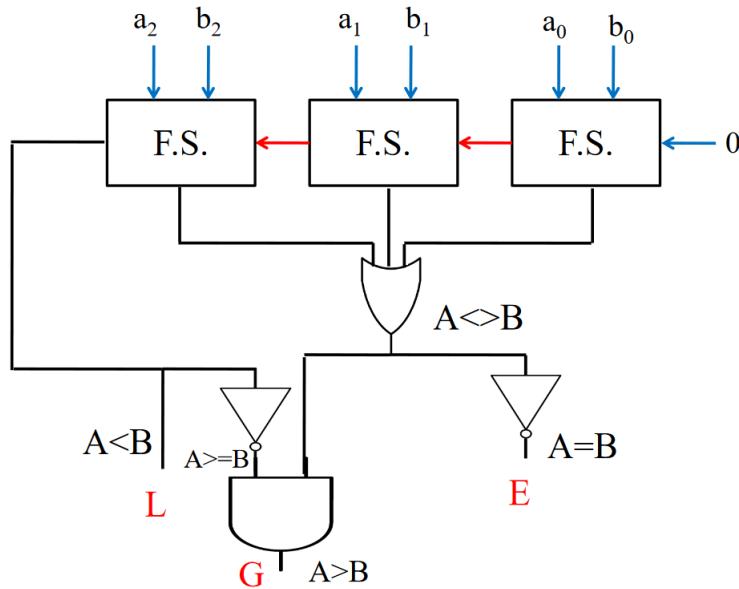
הגדלה משווים הם מעגלים שבודקים איזה קלט יותר גדול.

- דרך אחת למשם זאת היא באמצעות השוואה 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 נתונם 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 " $a > b$ " ו-0 במקרה $B > A$.

נמשח את המעגל כבאיור



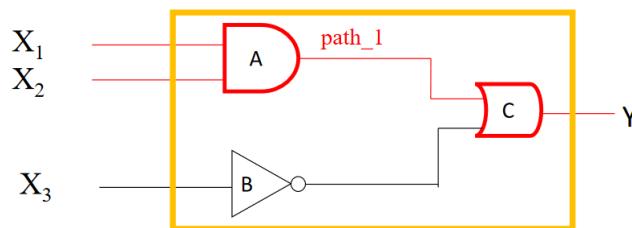
אם $a_2 < b_2$ בהכרח ישאר לנו Borrow Out כי אם BO מ לפני שימוש הלהה ואחרת $a_1a_0 < b_1b_0$ או יהיה $a_2 = b_2$ אז $a_1a_0 < b_1b_0$ ובעודאי שיהיה BO.

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 $A > B$ 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 . ובזומה עבר C . יש לנו שני מסלולים בمعالג, הראשון מ- X_1, X_2 ל- Y והשני מ- X_3 ל- Y .

לכן $t_{pd}(Y) = t_{pd}(A) + t_{pd}(C)$ (מסלול ראשון) ובהתאמה $t_{pd}(Y) = t_{pd}(B) + t_{pd}(C)$ (מסלול שני) כאשר הנתונים הבאים (Contamination מלשון $t_{cd} = t_{tp-min}$)

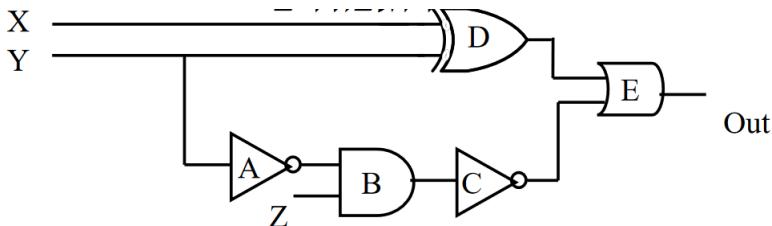
Data in ns	X_2	A	B	C
t_{PHL}	-	100	90	80
t_{PLH}	-	110	70	100
t_{cd}	-	12	8	10
t_r	14	20	12	18
t_f	15	17	13	19

נחשב את ה- t_{pd} של המעגל 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 מהקלטים לפלטיהם הם $B \rightarrow C \rightarrow E$, $A \rightarrow B \rightarrow C \rightarrow E$, $D \rightarrow E$ ו- Z בהתאם.

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

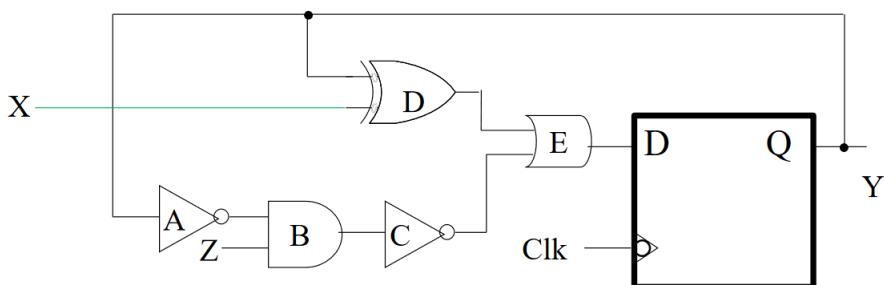
לכן $t_{pd(min)} = 15$ ו- $t_{pd} = 80$.

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

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

הערה הקלט למעגל חיב להתייצב 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 נצטרך את האילוצים על ה- DFF

$t_{hold} = 25ns$

- 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

הרצאה

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

דוגמה אנחנו נלמד על MIPS, אבל במצבות פופולריות 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 (שפתקמפלט), נתרגם את הקוד לסדרת פקודות אסמבלי באמצעות קומpileר. לאחר מכן נשימוש באסמבילר כדי לתרגם את קוד האסמביל לבינארי, שאותו המעבר כבר יודע להריצ'.

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 לקומפיילר 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 קומפיילרים עילים וכן נדרש המורכבות של פקודות האסמבלי, ואילו עם חלוף הזמן נהיה קל ויעיל יותר לכתוב קומפיילרים (בשפה עילית) שיבצעו את הפעולה המורכבת בעצמם.

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 קבוע 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 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 אותה.

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 וקובע והשמה ברגיסטר). כן $i = \$s + \t 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, שנגישים אליו באמצעות כתובות, כאשר כל כתובות מצביה לבית אחד של מידע, למרות שבפועל ב-MIPS נקרא ונכתב ביחידות של מילה (ארבעה בתים) מיישרות (כלומר כתובות שמתחלקות ב-4).

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

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

- 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 שמורות ב-\$s. נוכל לגשת אל האיברים במערך באמצעות `0($s3), 4($s3), 8($s3)`.

דוגמה נניח שיש לנו את הפקודות ב-C

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

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 מזיכרון, 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 לדג 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` (כפי שנכתב בקובץ ה-asm).

• קפיצה מותנת שווין: `bne $s==$t label` או `beq $s, $t, label` בדומה `bne` שkopצת אם $(\$s \neq \$t)$.

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- $\$t$.
- ואחרת 0.

מימוש פונקציות בא מסבל

הגדירה הקוראת היא הפ' שקוראת לפ' אחרת, הנקראת היא הפ' שנקוראת, הפרמטרים הם הערכים שמשוערים מהקוראת לנקוראת, התוצאות הם הערכים שהנקראת מחזירה לקוראת וכתובות החזרה היא הכתובת בזיכרון הקוד של הפוקודה שאחרי הקריאה לנקוראת בקוד של הקוראת.

המחסנית היא אוצר בזיכרון שתוכנה יכולה להשתמש בו, והיא משתמש שמיירה של מידע לוקאלית בעת הרצת פ' באופן שמאפשר קריאה 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 לכתובות 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 קריאה וחזרה מפונקציה

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

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

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

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

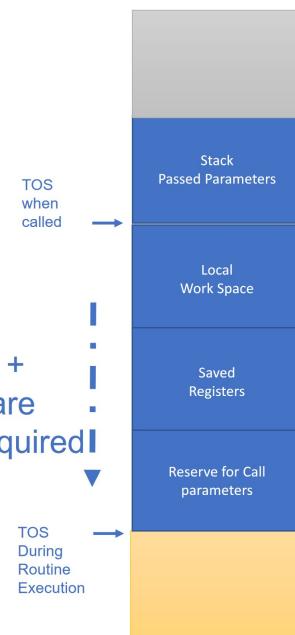
```
sw $ra, ($sp)
```

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



פקודות קריאה לפונקציה

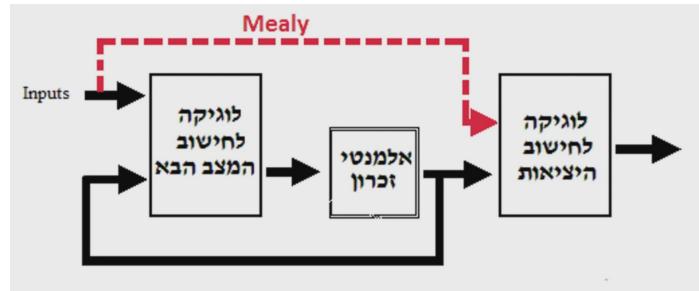
- label jal \$ra שם ב-\$ra את כתובות הפקודה הבאה (הערך הבא שיקבל PC) ואוז קופצת ל-label.

- \$.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

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

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

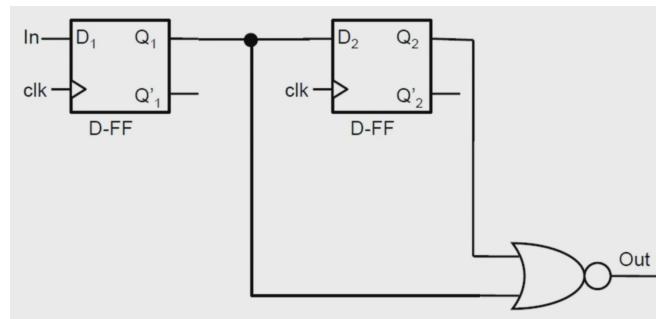
2. כתיבת טבלת מעברים.

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 הזכרון 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 הгинויות, אבל עדין נכתבות אותן.

3. ניתן שמות למצבים, כאשר מצב מוגדר ע"י קיבוע ערכאים של כל פלטי רכיבי הזכרון, 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

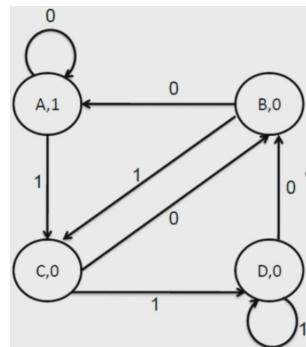
ועכשיו נוכל לחשב את הפלט של המעגל בהינתן הקלט וה המצב הנוכחי בטבלת המצבים הבאה (PS המצב הנוכחי ו-NS המצב הבא)

(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, כולם 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.

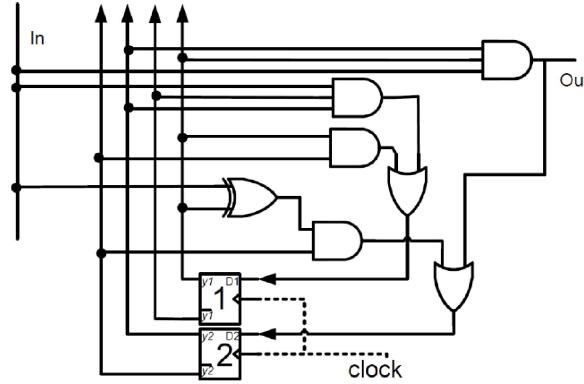
In	0	0	1	0	0	1	1	0	1	0	1	0	0	0
State	A	A	A	C	B	A	C	D	B	C	B	C	B	A
Out	φ	φ	1	0	0	1	0	0	0	0	0	0	0	1

כאשר שני הפלטים הראשונים הם Φ כי כל עוד לא עברו שתי ייחדות 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 במבנה 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 000 (ע"י היישאות ב-A). הכלל האופטימלי וה邏יגרטי 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 המცבים

	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 למცבים (קיובע פלטי הזיכרון) כרגע

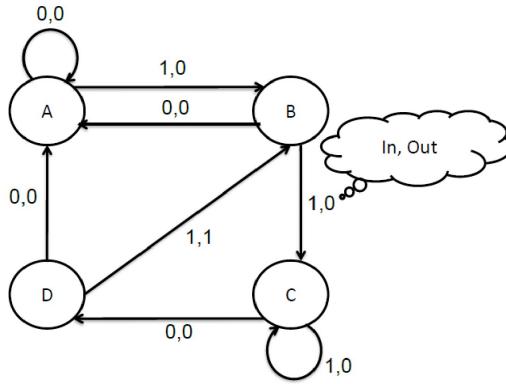
שמות המცבים	Q_1	Q_2
A	0	0
B	0	1
C	1	0
D	1	1

ועתה באמצעות הצבה של השמות בטבלת המცבים 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 לשימוש באוטומט Mealy, כלומר שעל הקשת כתוב איזה קלט מעביר אותו למצב

הבא ואיזה פלט הוא מספק לנו



5. עתה נרשום סדרת בוון שתראה אותו הדבר, רק שהפעם בחיפוש אחר כל השינוי נוצר לתחשב 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.

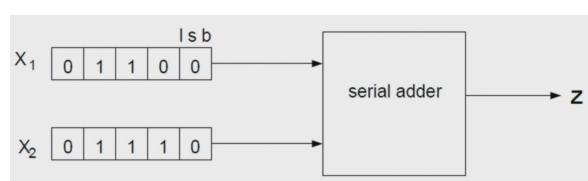
4. טבלת מעברים ופלט.

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

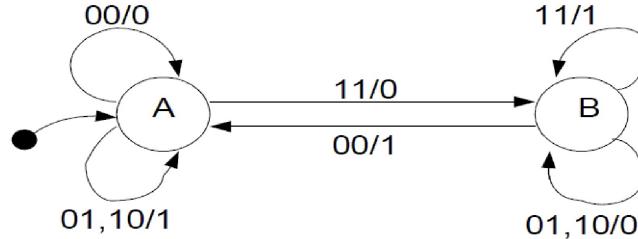
6. בניית המעגל.

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, שכךילו מתווספים כ-MSB במספרים שאנו חנו סוכמים).

ראו איור (כל פעם סוכמים את הביטים המתאים, 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!

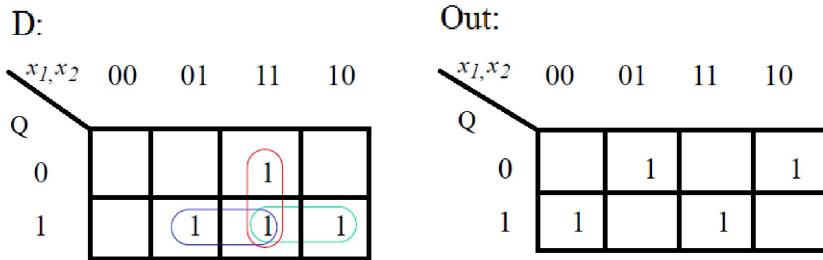
3. קידוד המ מצבים דורש בחירת ייצוג בינארי למצבים, כאשר יש שניים זה די קל - מספיק בית אחד שייצג את A כשהוא 0 ואת B כשהוא 1.

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

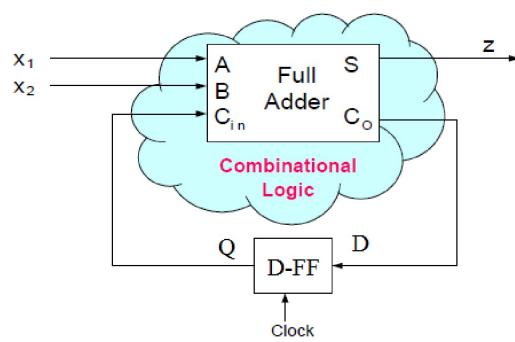
5. נפצל את הטעלה לשתי מפות קרנו (אחת ל- D , הקלט ל- DFF שיחזק את המצב הבא ואחת לפולט) ונכשה את הטעלה כמו שעשינו

בתרגול 2



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

כז



שבוע VII | 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 שימושן למשמעותם פקודה אחת שכן קיימת, לדוגמה $\$t = \d , $\$s = \t מימוש ע"י $\$t = \d 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} סח"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- 6 שגדיר אליו פעולה בדיק נבצע (לדוגמה ל-add ו-sub יהיה כאן ערכים שונים).топס 6 ביטים כך שיש לנו לכל היותר 64 פקודות מסווג R. נשים לב שככל פקודה תופסת הרבה מילימ (5 ביטים לכל אינדקס רגיסטר, כולל 32,000 מילימ שמייצגות פקודות סכימה כלשהיא).

- 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
--------	-----------

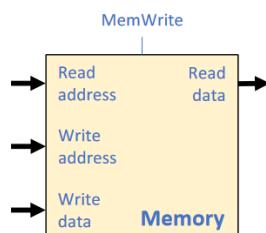
כדי שנוכל לפזר למגוון כתובות בזיכרון (2^{26} כתובות), כאשר רקjal ז-ו-ja מקודדות כך (שאר הקפיצות כוללות immediate)

הערה כתובות R-Type הן חci 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 קצת יותר מורכבת כי הפקודה רצה 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 בקפיצות 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.

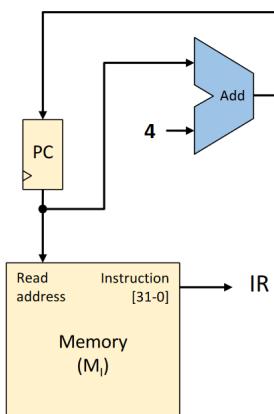
הערה כשנסמן חץ באופן הבא ← אל תוך רכיב ומתחתיו מספר, הכוונה שיש לנו סע שמורכב ממספר חוטים (מספר שכתבנו) ולא רק ביט אחד.

שלבי הרצת פקודה

- קראית הפקודה (fetch) : המעבד מספק כתובת (ששמורה ב-PC) לזכרון ומקבל את תוכנה, שהיא קידוד הפקודה שצורך להריץ.
- פענוח הפקודה (decode) : להבין איזו פקודה צריך לבצע ואילו רגיסטרים היא דורשת. בנוסף נזין 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` יכתוב.
- כתיבה חוזרת (write back) : לכתוב את תוצאות החישוב או ה-`load` ל-PC. השלב הזה למעשה משתנה בהתאם למצב המכונה לקרויה הפקודה הבאה.

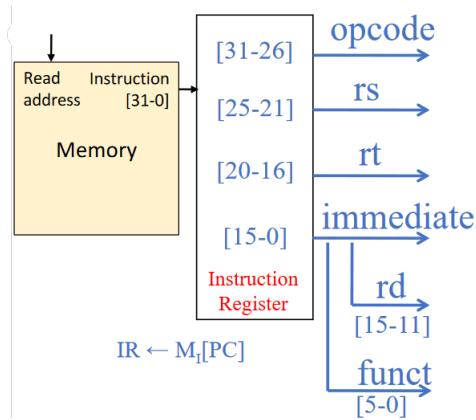
מימוש השלבים

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



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

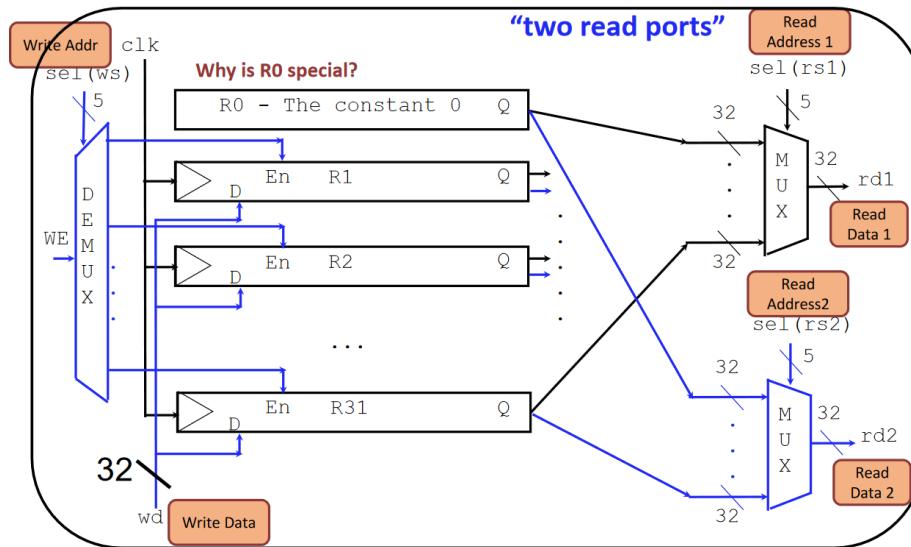
- פענוח: נפצל את הביטים ברегистר הפקודה (IR) למשמעויות הרגולונטיות שלם (ראו אייר)



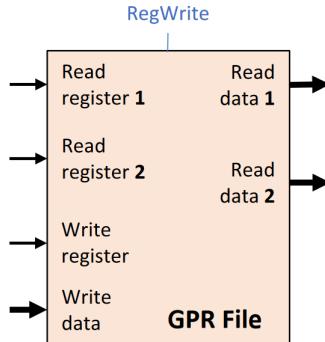
ובנוסף נביא את הרגיסטרים הנדרשים לפעולה, שממוקמים ב-FF, Register File. ים שבו הרגיסטר הראשון הוא בעצם קבוע 0 ולא רכיב זיכרון וה-31 האחרים הם רכיבי זכרון אמיטיים. כדי לקרוא מהם מידע, צריך להשתמש ב-Mux שבודח אחד מבין 32 רגיסטרים ומוציא את התוצאה החוצה. הכתובת שנקרה תלואה כמובן ב-rt, rs ו-rd לפי הצורך.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

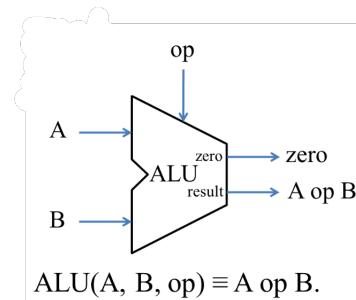
כדי לכתוב בהמשך לרוגיסטרים (בשלב ה-DeMux) נחבר את ה-FF-ים ל-DeMux שnbrור אליו את הרגיסטר הנוכחי אליו אנחנו רוצים לכתוב אליו באמצעות בית WriteEnable. האיסולטרציה הבאה מדגימה את כל האמור.



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

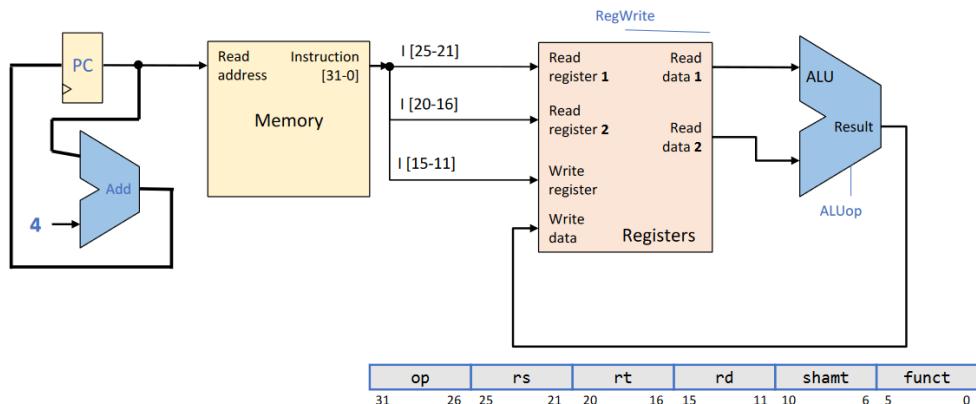


• ביצוע: הרכיב האחראי על החישוב האריתמטי נקרא Arithmetic Logic Unit (ALU), שמציג את הממשק הבא



כאשר הפלט zero הוא בית (דיל) שערכו 1 אם פلت החישוב הוא 0 (שימושי מאוד לкопיצות מותנות 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 בהתקדמות ה-PC, 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).

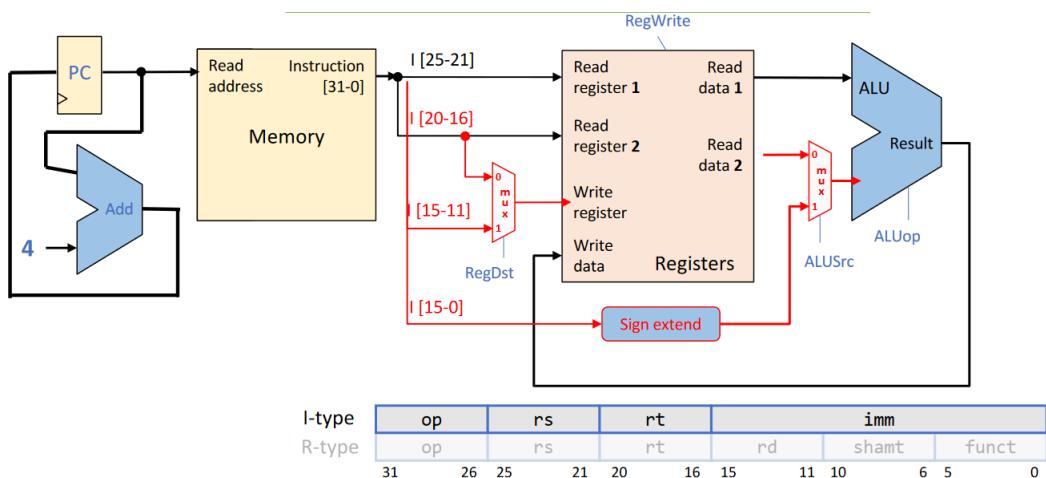
כדי לממש פקודת I-Type, נבצע את אותה הפעולה של R-Type עם כמה שינויים קטנים:

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 שבורר לפי סוג הפקודה.

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

הערה בגל של-sh-choffimm opcode, ה-mm funct מטרך להגיד לנו מה מדבר ב-A-Type-R-Type ו-I-Type כדי שנוכל לפרש את הביטים נכון.

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

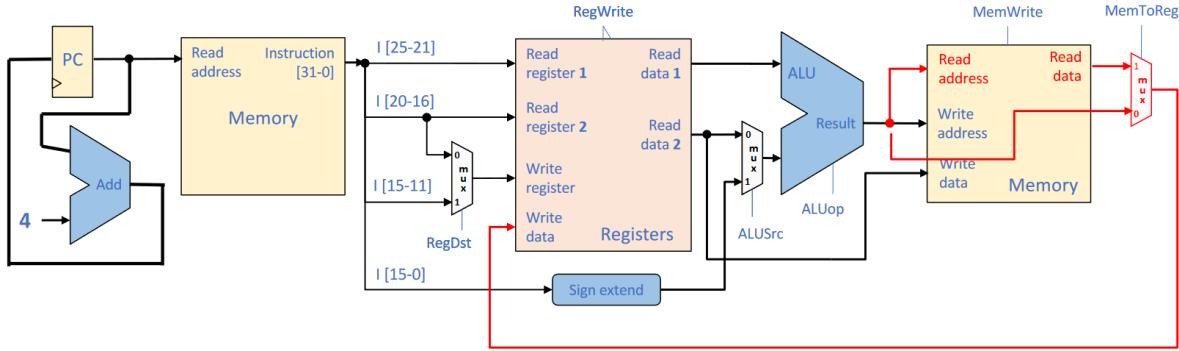


Reg [IR [rt]] : השלב הזה חיוני למימוש כתיבה וקריאה, כאשר בשתי הפקודות מעתיקים את הערך של memory access write back -]-[Reg [IR [rs]] + SignExtend (imm) WE=0 או להפוך, בהתאם.

כדי למש את הפעולה נשתמש ברכיב הזכרון Data Memory שמקבל קלטי Addr (32 ביט כל אחד) ו-WE (32 ביט כל אחד) ו-WE (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 (שיסכום 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 (\$5, 7(\$6) כאשר 35 הוא opcode של קריאה מהזיכרון, rt , כלומר מקודדים את אינדקס הרגיסטר ממנו לקרוא ואליו כתוב ולבסוף ה- mno שהוא היחסט מ- rt שנוטן את הכתובת.

הערה J-type מקודד את כתובות הפקודה ביחידות של מילה במקומות הביתי, ולכע אפע"פ שיש לנו 2^{26} ביטים, נוכל ליעץ 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 הבא

```
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 השוואה של הרגיסטרים המתאיםים 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, נוסיף:

- 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 נוסיף פקודת j לסוף ה-switch-case כולל, כך שלא נריץ בטעות הסגרים אחרים (בדומה לפקודת ה-break).

דוגמה את ההתניה if 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 ההיסטוריה אליו צריך לкопץ אם מתקיים התנאי.

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

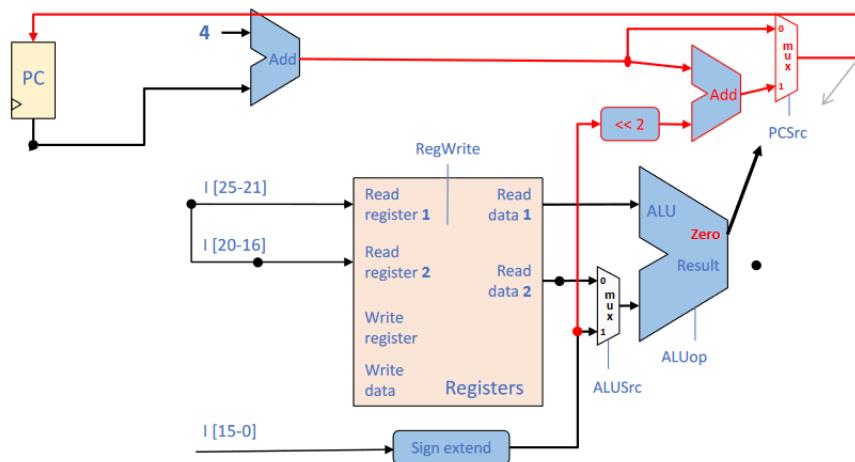
5. ה-ALU ישווה את הרגיסטרים לפי בהתאם לשוג ההשוואה.

6. PC יעדכן ל- $PC + 4$ או לכטובת במקרה הקפיצה, בהתאם לתוצאה ההשוואה.

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

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

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

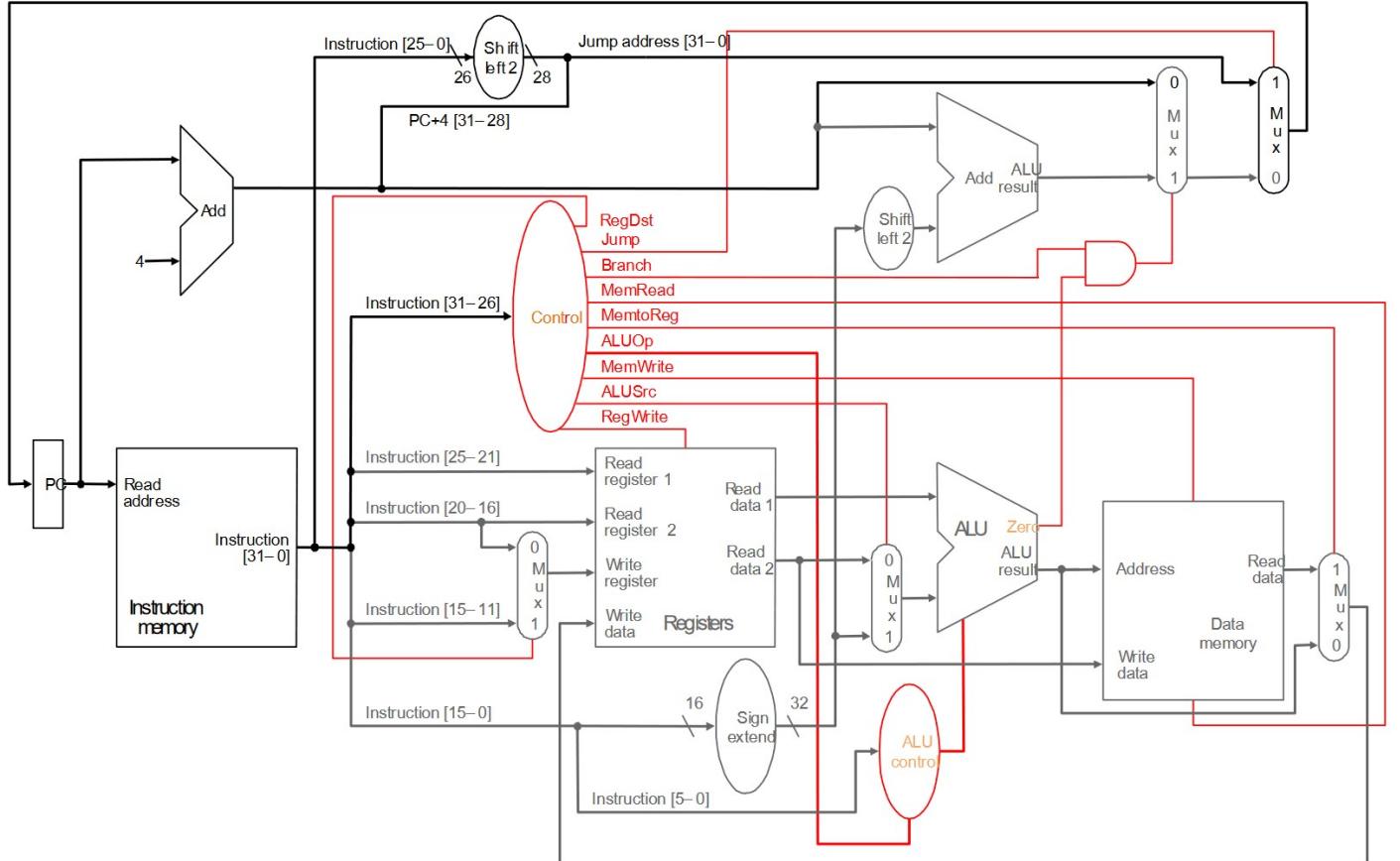


פקודות מסוג J-Type נמש בדומה, כאשר עתה נעדכן

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

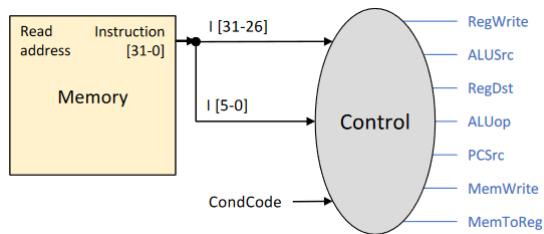
כאשר -. היא פועלות הצמדה (concatenation) של הפרש הכתובות בין כתובות הפקודה הנוכחיות לכתובת ה-label.

לxicoms המעבד השלים שלו נראה כך



יחידת השליטה

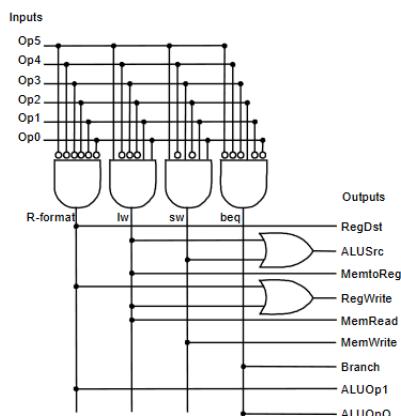
עד כה המעבד שלנו כולל כל מיני ביטים שימושיים כשלקוטורים ל-Mux-ים ורכיבים אחרים (מוסמנים בכחול ולא מגיעים משום מקום). את כולם אנחנו מחשבים על בסיס שלב הפענוח של הפקודה. היחידה שלוקחת את ה-opcode (ובמקרה של R-Type גם את ה-funct) ומחשבת את השודות השונות שלשולטים על הריצה נקרא **Control Unit**.



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



השיטה נקרא Logic Array.

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

מעבד רב-מחזורי

נמשך את המעבד בדרך אחרת תחת אותו פקדות.

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

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

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

דוגמה עבור Single Cycle MIPS CPI הוא 1 לכל פקודה.

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

- 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

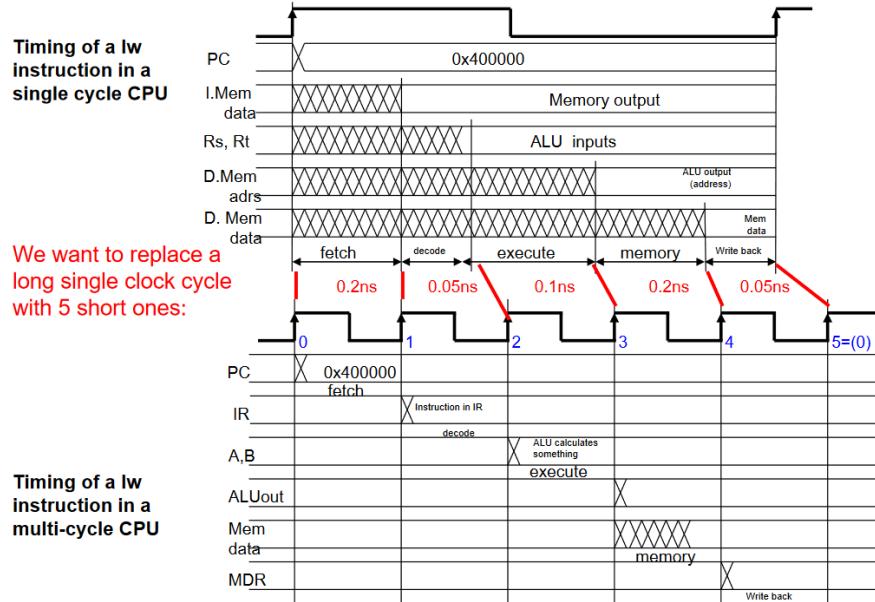
$$CPI = \frac{\#cycles}{IC} = \frac{\sum_i CPI_i \cdot IC_i}{IC} = \sum_i CPI_i \cdot \frac{IC_i}{IC} = \sum_i CPI_i \cdot F_i$$

שיטות השוואת ביצועי מעבדים

- הריצה על חומרה אמיתית: קל להריץ על פני הרבה זמן, קשה לנתח לעומק. מה נשווה בין מעבדים?
- 1. לרוב נ裏ץ-benchmark-ים סטנדרטיים של פעולות מקובלות כמו `compression`, `office`, תוכנות, קומפיילרים וинтерפרטרים.
- 2. אפשר לחשב ביצועים מקסימליים (MFLOPS ו-MIPS) אבל הם לרוב לא ריאליסטיים כי אי אפשר להגעה אליהם בריצה סטנדרטית.
- 3. 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 במציאות ו מגביל את אורך הריצה וארך המשאבים.
- אנליזה: חישוב תאורי 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 בעברה).

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

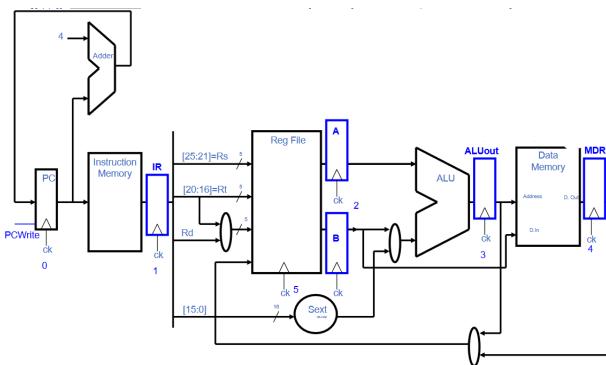


נשים לב שנכלל תדריות הרבה יותר גבוהה אבל מבחינה ביצועים עברו שלבים שאינם הארוכים ביותר, אנחנו משבזים זמן עכשו,

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)

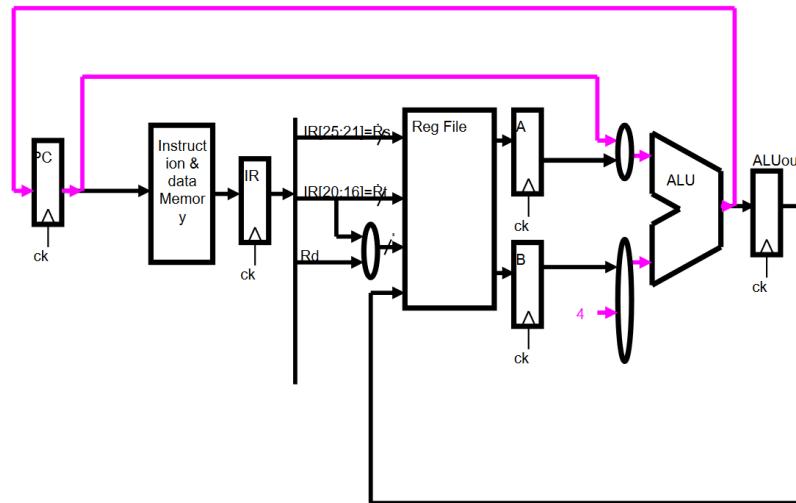


איך הגענו להפרדה זו? נביט במסלול של פקודת `lw`: מתחילה ב-`fetch` ואו קריאה מרגייסטרים, מעבר ב-`ALU` ולסיום כתיבה חוזרת לרגיסטר. לכן נפריד (1) בין ה-`fetch` לקריאה; (2) בין הקריאה לחישוב; (3) בין החישוב לכתיבה חוזרת לרגיסטרים; (4) ולפני כתיבה מהזיכרון לרגיסטרים.

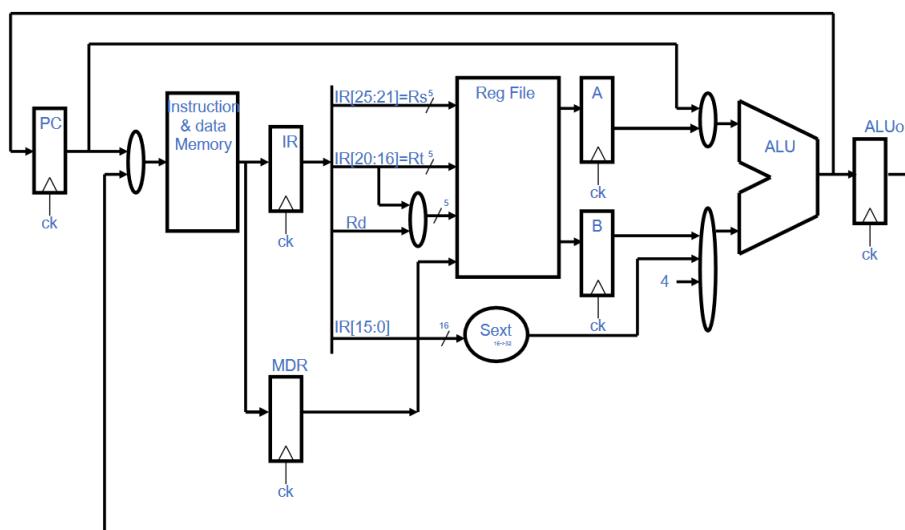
הערה כרגע חסירה לנו לוגיקה עדין שדואגת שהמעבד יבצע רק את השלב שצורך כרגע. כי אחרת ה-fetch קורה בכל מחזורי שעון ולא רק במחזוריים שימושיים ל-fetch.

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 והدادטא, 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 רוצים. עדין נפריד את התוצאה לרגיסטרים השונים כי נרצה לזכור מה המצביע שלנו כמו שצריך ואם הכתובת והدادטא היו באותו רגיסטר המצביע 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).

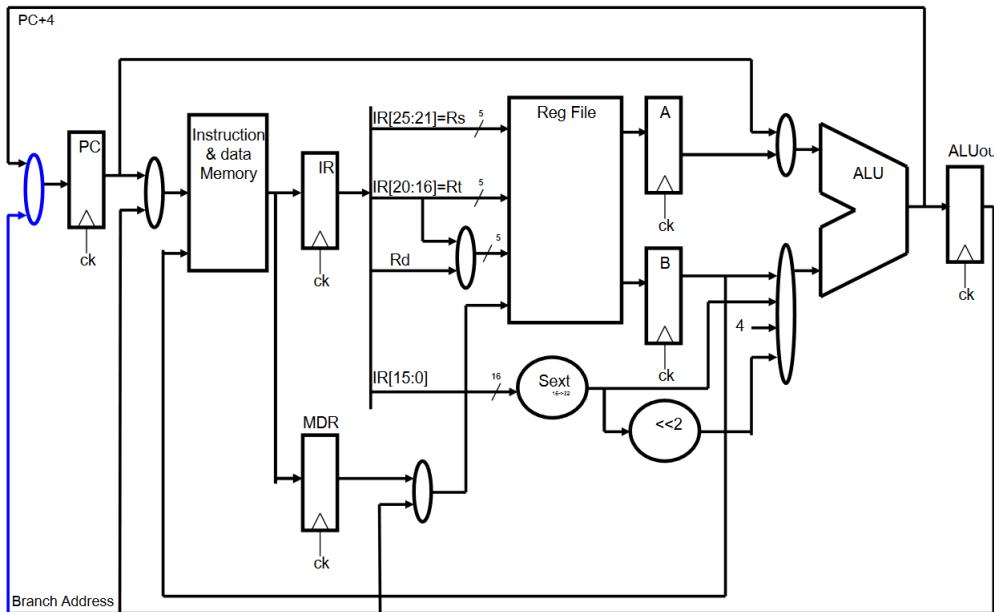
- sw : 4 מחזוריים (fetch, פענוח, קריאה מהרגיסטרים, חישוב ב-ALU וכתיבה לזכרונו), כאמור שוב במקומות חמיisha.

- 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.

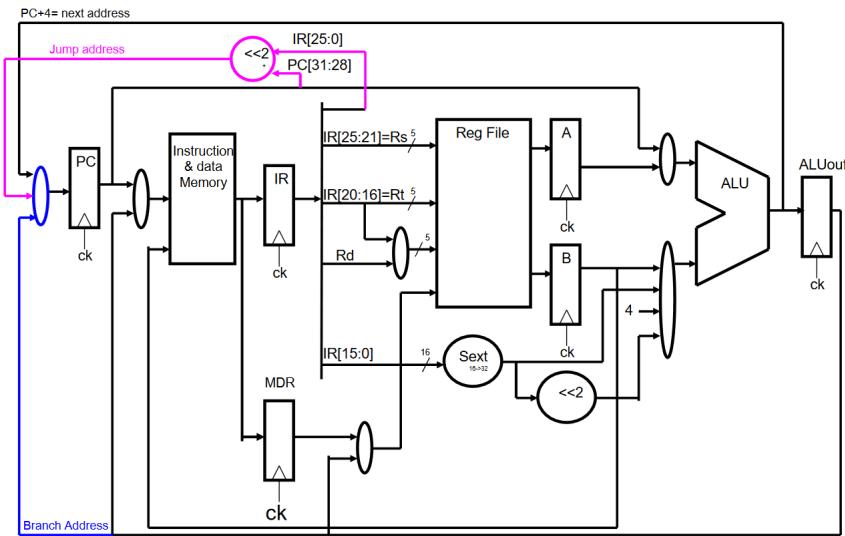
הבעיה כרגע היא שאם התנאי מתקיים זה לוקח שלב אחד יותר מאשר אם הוא לא, ונרצה שהו יקח מספר פעולות קבוע.

- 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).

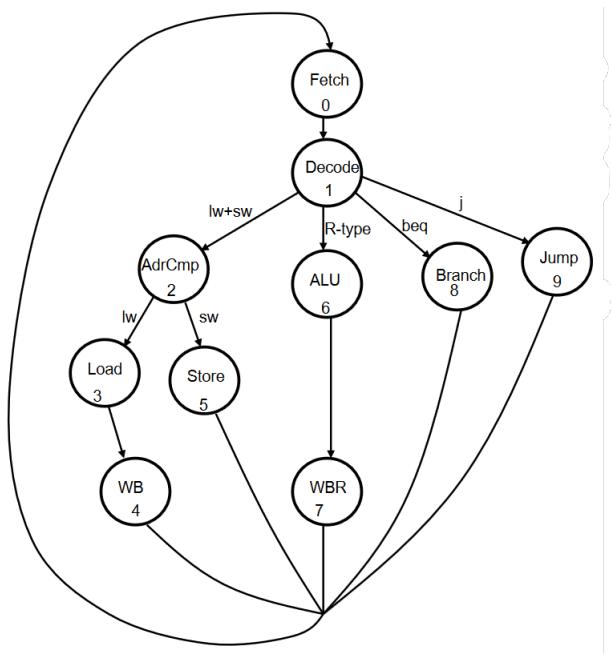
הסטודנטית המשקיפה תסתכל על המעבד החדש שנבנה (באיור) ותוווכח שאנו נדרשים רק שלושה מחזוריים כדי לבצע את beq.



- 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.



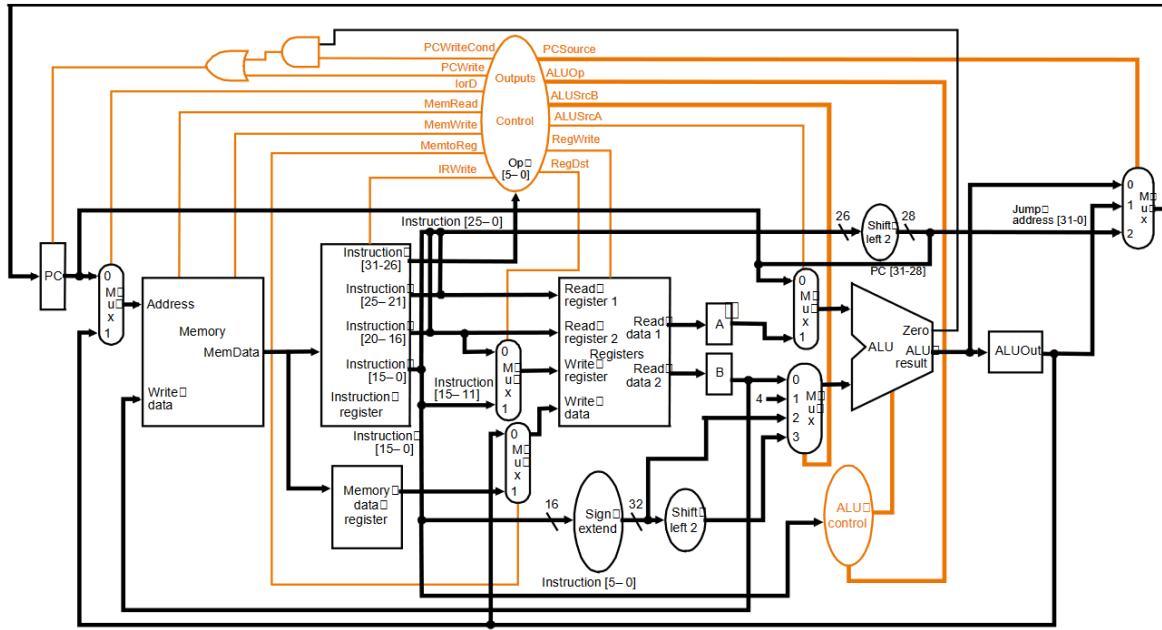
במהלך ניתוח כל אחת מהפקודות, בינוו בצד מכונת מצבים שמייצגת את המעבד, כי בסופו של דבר בעת מימוש המעבד באמצעות תהליכי הסינטזה של מודולנו צריך לבנות מכונת מצבים. להלן הטללה, שתואמת בדיק לניתוח הנ"ל של כל אחד מהשלבים.



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

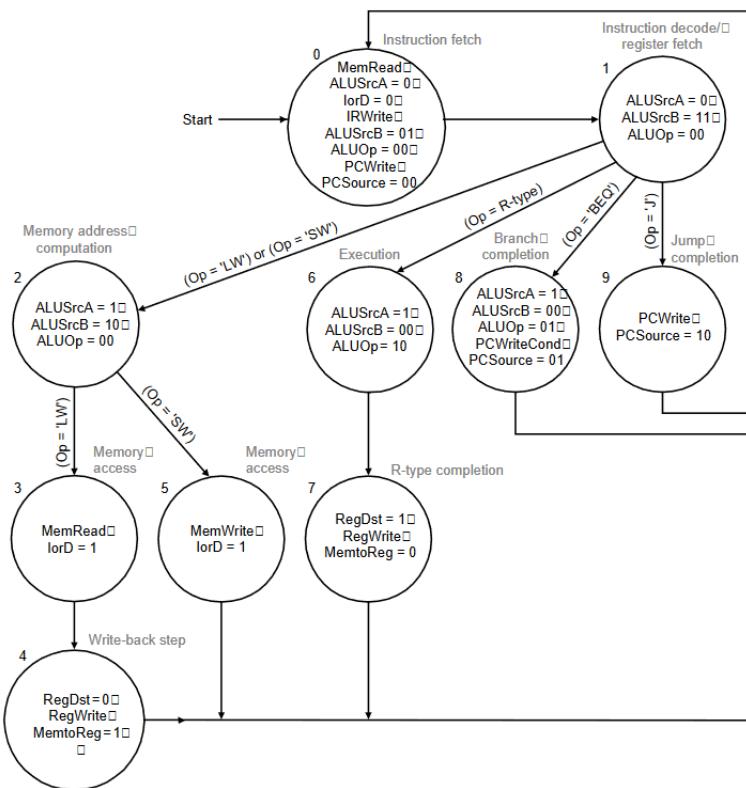
נשים לב שנוסף לנו בית חדש PCWriteCond שמחלית אם אנחנו הגיעו בשלב fetch ולכן צריך לקדם את המעבד. בנוסף יש לנו גם PCSOURCE ו-PCWRITE שקשורים למימוש של פקודות קפיצה מותנת ולא מותנת (jal). לסיום נוסף לנו IRWrite שקובע האם מעודכנים oczywiście את תוכן רגיסטר ה-IR, שמכיל את הפקודת שכותבה הופיע בזמן השעון הקודם 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 הינו, אבל אנחנו נתעלם ממנו.



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

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

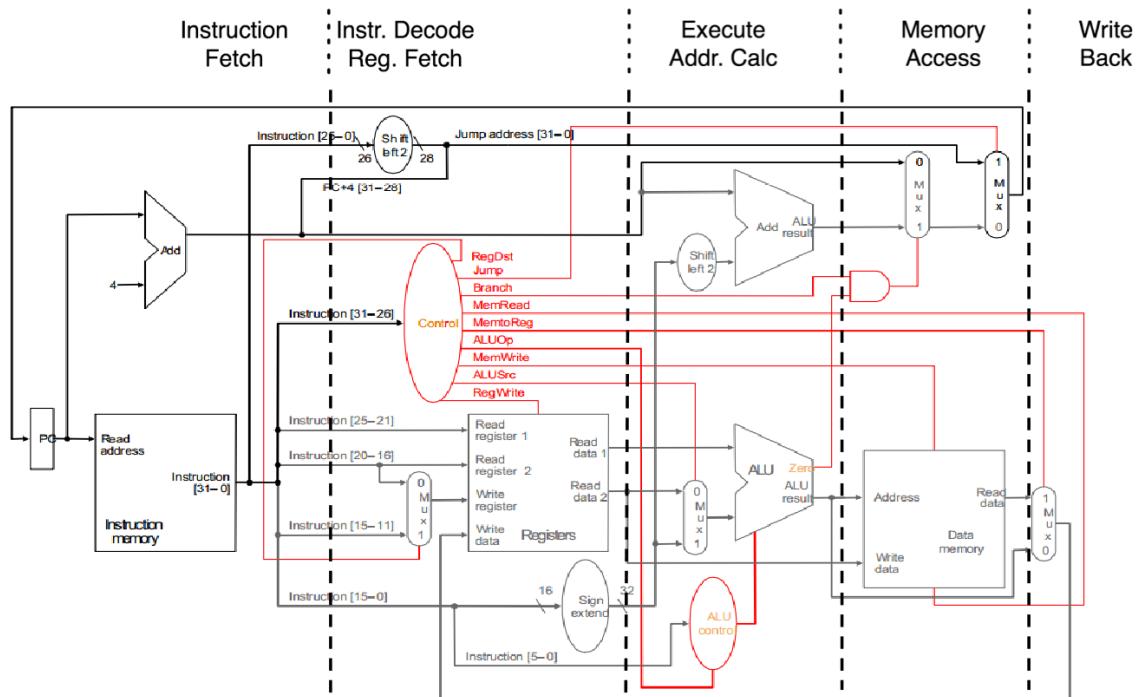


לסיכום כל שלב ומה הוא עושה, ללא מידע חדש

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	0	$IR = \text{Memory}[PC]$ $PC = PC + 4$		
Instruction decode/register fetch	1	$A = \text{Reg}[IR[25-21]]$ $B = \text{Reg}[IR[20-16]]$ $\text{ALUOut} = PC + (\text{sign-extend}(IR[15-0]) \ll 2)$		
Execution, address computation, branch/jump completion	$\text{ALUOut} = A \text{ op } B$	$\text{ALUOut} = A + \text{sign-extend}(IR[15-0])$	if ($A == B$) then $PC = \text{ALUOut}$ ($IR[25-0] \ll 2$)	8 9
Memory access or R-type completion	$\text{Reg}[IR[15-11]] = ALUOut$	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 האיוור המלא של המעבד, יחד עם יחידת הבקשה (והפרדה לוגית בלבד בין שלבי ביצוע (פקודה



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

דגלי יחידת הבקשה

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

- ביט שקובע האם במחוזר הנוכחי ניקח את המילה שנקרה מייחידת הזכרון (1) או את התוצאה (0) מה-ALU וכותב אותה ל-Register File (או שנותעלם ממנה).
- MemRead : ביט שקובע האם במחוזר הנוכחי קוראים מהזיכרון (וכנס לרכיב הזכרון).
- MemWrite : ביט שקובע האם כותבים לזכרון (וכנס 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 הנוכחית היא קפיצה לא מותנת.
- RegDst : האם הרגיסטר אליו כותבים (אם כותבים) 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. להלן טבלת ההמרה מ-ALUop ל-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 הריגיסטר השני שMASOPAK.
- $\text{Branch} = 0$ כי מדובר בפעולה אריתמטית ולא קפיצה.
- $\text{MemRead} = 1$ כי אנחנו רוצים לקרוא מהזיכרון ולכתוב לריגיסטר.
- $\text{MemToReg} = 1$ 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 הריגיסטר 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 הריגיסטר 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 הריגיסטרים.
- $\text{MemRead} = 1$ כי אנחנו כנ"ל.
- $\text{ALUop} = 01$ כי זה הקוד לפקודות מסווג I-Type ואז ה- ALU Control יחשב את הקוד שה-ALU מבין לשכימה (לכל ה--I Type זה מה השימושים) ונקבל 0010.
- $\text{MemWrite} = 0$ כי אנחנו לא קוראים ולא כתבים.
- $\text{ALUSrc} = 0$ כי אנחנו רוצים להשווות את הריגיסטר 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 הפוקודה מכילה opcode והיחסט אליו קופצים בלבד), לכן נוסיף עוד בית ל-RegDst וחיווט נדרש כך ש庆幸ה 00 פירשו כתיבה ל-\$rd (הריגיסטר השלישי ב-R-Type), 01 ל-\$rt (ב-I-Type) ו-10 לריגיסטר 31 (\$ra).

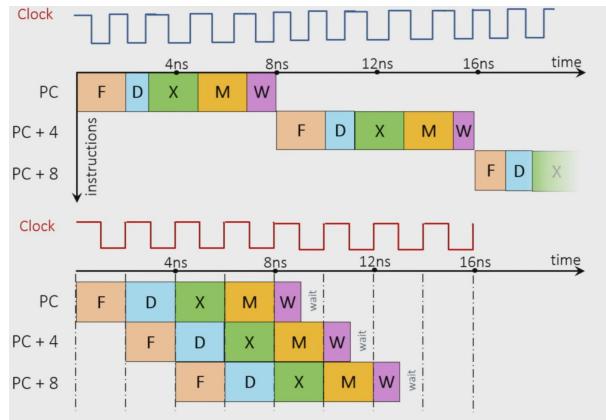
שבוע VII | מעבד pipelined

הרצאה

עד כה הצלחנו להוריד את זמן המחזור פי 4, אבל גם העלנו את מספר פקודות להוראה פי 4, כך שלא הרווחנו 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).

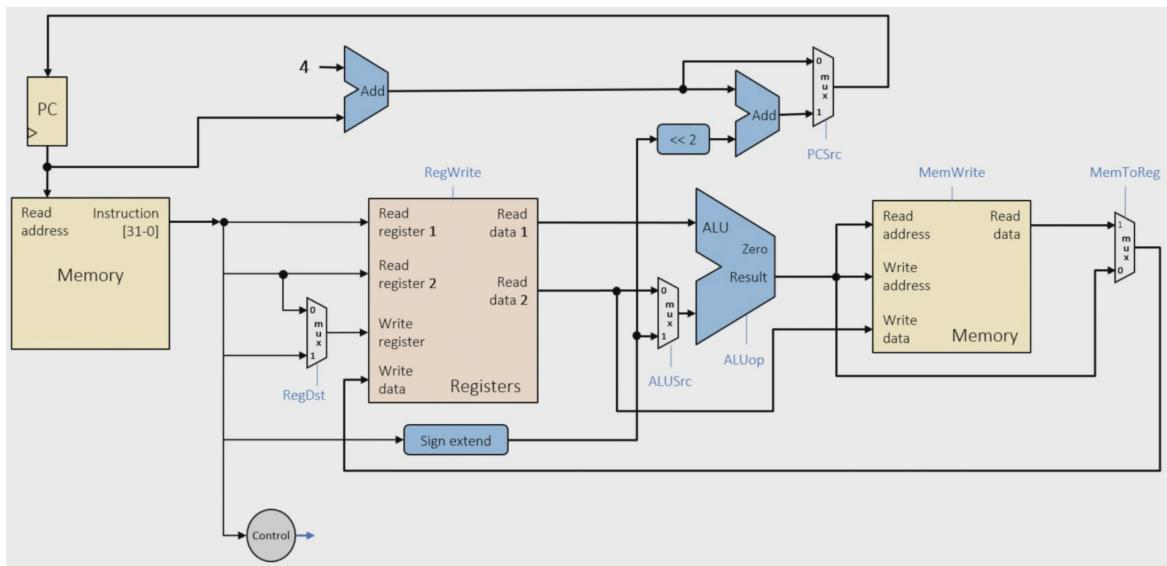
דוגמה במקרה של מעבדים, נשווה בין Multi Cycle (למעלה) ל-pipelined (למטה). ברור שעדיף Pipelined, ובפרט זמן החישוב שלו לינארי עם מקדם כמעט 1 (מגלמים את הפקודות בקצבות) ביחס למספר הפקודות, בעוד Multi-Cycle הוא לינארי עם מקדם יותר מ-4 ביחס למספר הפקודות.



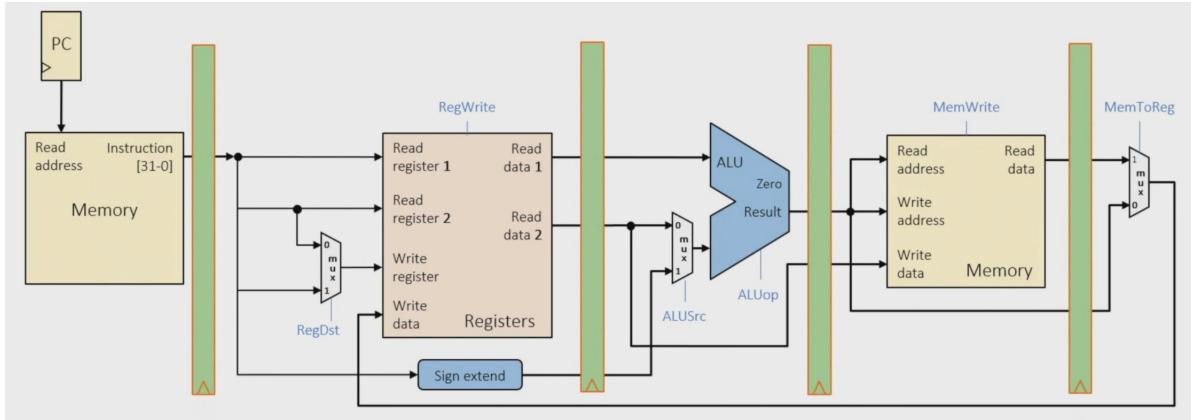
תחת הנתונים הנ"ל, ל-MC יש IPC של $\frac{1}{4}$ ותדריות של $500MHz$ ולכן $IPS = IPC * f = 0.25 * 500 = 125$ $\frac{1}{2ns} = 500MHz$ ואילו ל-IPC של 1 (בלי הקצוות) ואוthonה תדריות של $500MHz$ קלומר speedup של 4.

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:



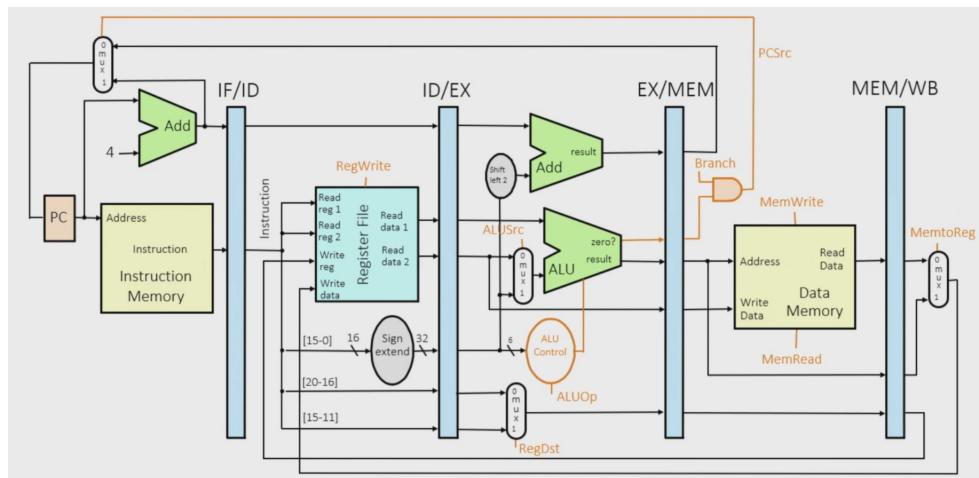
אם נתעלם ליעציו מהקונטROL ומה-branch-ים, יוכל לחלק את התהילה לשלבים השונים של ה-pipeline (הקיים הירוקים הם רגיסטרים ששומרים מצב לאחר חישוב השלב)



כלומר השלבים שלנו הם write back (5) ; memory access (4) ; ALUOp (3) ; decode (2) ; fetch (1) כולם לב שהחלוקת הזה זהה. נשים לב שמחולקה הזו זהה לזו במעבד הרב-מחזורי. הסיבה לדמיון הזה הוא שאלה שלוקחים אותן זמן ולכן חלוקתן לזמן מוחזר שוויים היא יחסית הוגנת.

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

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



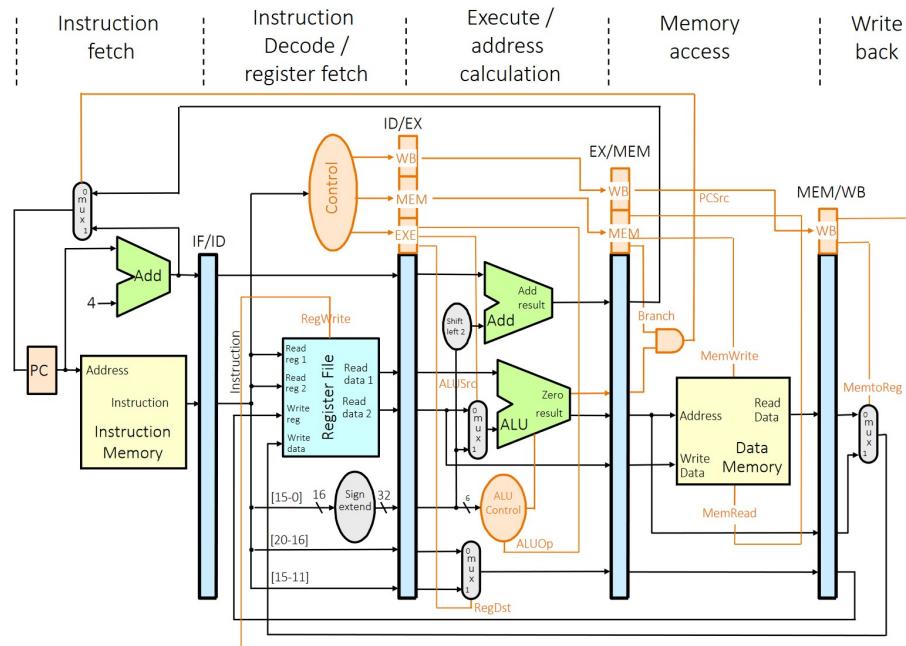
נפרט את השלבים שתעבור הפקודה במעבד :

- לאחר פקודת ה-`lw` יחזק את החוראה (כלומר מה שב吃过 נשמר ב-IR) ובנוסך את הכתובת ממנה נקראת (`PC+4`), זו נדרש בשביל branch-ים בהמשך.
- לאחר פקודת ה-`lw` יחזק את תוכן הרגיסטר `$2` בדומה לנ"ל, את תוכן הרגיסטר `$1`, את ה-`ALUOp`, את האינדקס של רגיסטר `1` ואת האינדקס `30`.
- לאחר פקודת ה-`lw` יחזק את תוכן `$2` סכום עם `30`, ואת אינדקס `1`.
- לאחר פקודת ה-`lw` יחזק את המידע מהכתובת שביבשנו יחד עם אינדקס `1`.

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

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

עתה יחד עם יחידת הבקרה, המעבד יראה כך



כאשר כמוון ה-control מעביר מידע רק אחד שלב decode-ה-ID/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 מילא. 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 ונגנים 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:

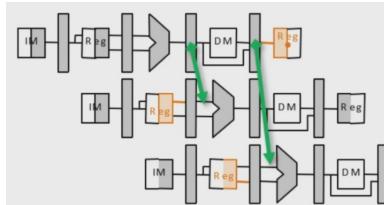
- **סכנות מבניות:** החומרה לא יכולה לתמוך בשילוב פקודות (לדוגמה נctruck זכרו שונה ל-PC והדאטא כי אחרת ניתקע במצב ש策יך לכתוב בו זמן לתשתיות כתובות מאותו זיכרו).
- **סכנות דאטה:** הוראות שמתבססות על פקודות שעוד לא חושבו pipeline (לדוגמה שימוש ברגיסטר שתוכנו אמרור היה להתמלא כבר בתוצאתה של פקודת סכימה שעוד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

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



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

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

דוגמה נציגים מקרה שבו נדרש לבחור בין פקודות חדשות לשונות

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

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

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

כאן בעת חישוב הפקודה השלישי, נרצה את התוצאה הטרייה יותר (הסדרה הראשונית, שכבר לא רלוונטית. אחרת היינו במצב של Out of Order Execution, שקרה רק ב-*Write After Write* ב-*Write After Write* After Write, שנעוק בשבות הבאים.

• העתיקת זכרון-לזכרון (סכנת DATA) : הבעיה עולה כxebaces{sw} לכתובות בזיכרונו שזה עתה כתבו אליה עם *sw*.

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

• load-to-use (סכנת DATA) : הבעיה עולה כxebaces{lw} פוליה שמתבססת על תוצאה קריאה מ-*lw* בפקודה הקודמת.

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

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

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

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

מניעת סכנות ב-branching

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

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

1. 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 מהקצotta) וזה קורה ל-20% מה-branch-ים, ככלומר פי 1.6 מוחזרים פר-הוראה, שזה פי $\frac{1}{1.6} = 0.63$ הוראות פר-מחזור ככלומר האטה של 37%.

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

2. תמיד לעשות כאילו לא קפצו: נמשיך את הפקודות כאילו הקפיצה לא קורתה, ואם היא קורתה אז נעשה לפקודות שלא היו אמורים לרוץ flush ונייקח את הפקודות שכן קפצו. 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. נשנה את ה-ISA כך שכל תוכנה, גם אם יש קפיצה, תזכה לפחות כמו פקודות לפני הקפיצה בכל מקרה, וכך יהיה לנו זמן לחשב את הקפיצה ולא תהיה ההתנגשות זו. ככלומר אנחנו מנסים את החוזה בין המפתח למעבד.

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

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

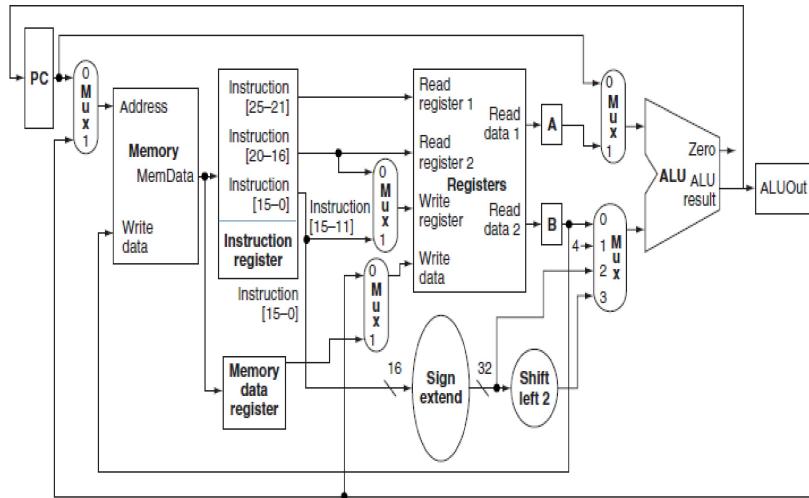
תרגול

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 למרבבים במעבד רב-מחזורי

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

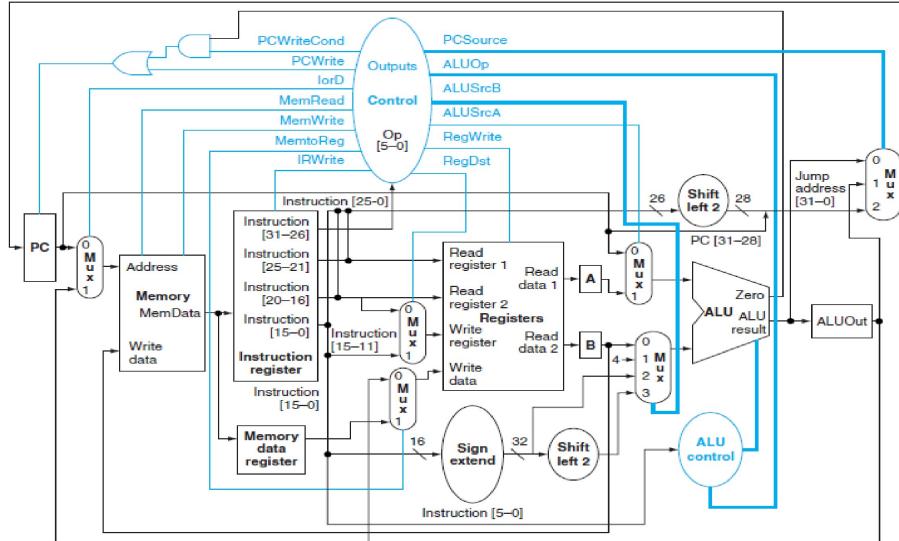


מעבר על כל ה-axut-ים ונסביר איזה בית מחובר אליהם ומה תפקידם, משמאל לימין (ושובר שווין לפי למעלה-למטה) :

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-RF (1) - רלוונטי לפקודת lw לעומת כל השאר (בחלקו לא נכתב כלום ל-RF ואז בית זה אינו רלוונטי).
4. מחובר ל-ALUSrcA, ופירשו האם יש להשתמש ב-PC כקלט הראשון ל-ALU (0) או בתוכנים של רגיסטרים מה-RF (1) - רלוונטי לפקודות שמחשובות היסט מה-PC הנוכחי (קפיצות מותנות ולא מותנות).
5. מחובר ל-ALUSrcB, ופירשו האם יש להשתמש ברגיסטר שנקרא מה-RF כקלט השני ל-ALU (00), ב-4 (01), ב-4*imm (10) או ב-4*imm (11) - רלוונטי לסוגי הפקודות השונות : PC ב-4, beq ו-R-Type, קידום PC ב-4, fetch, פקודות I-Type-arithmatic ו-sw/lw, וחישוב כתובות קפיצה, 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 שהיחסבו במקרה של פקודת j (jal).

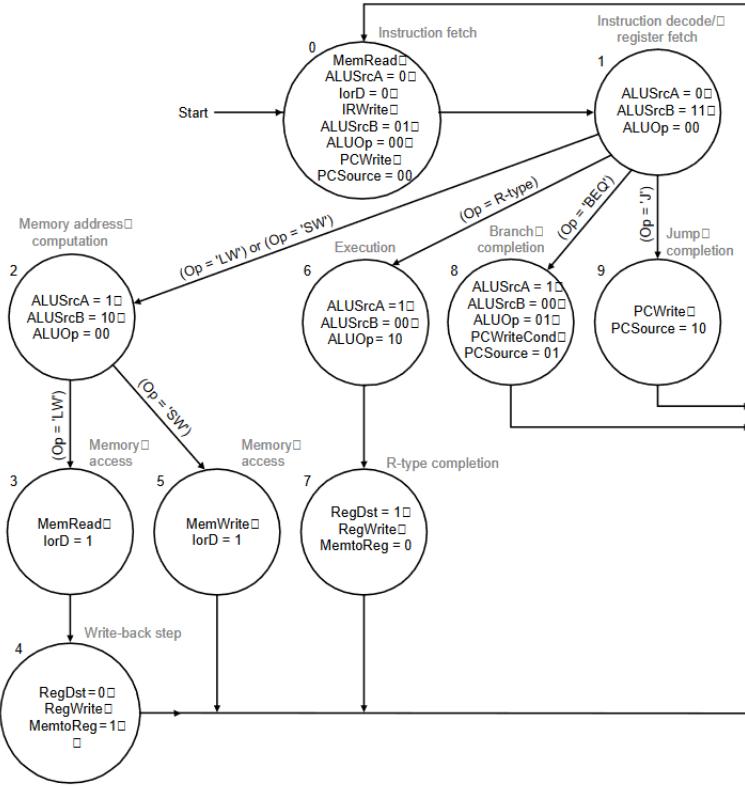
.2. האם לכתוב ל-IR את פלט הקריאה מהזיכרון (דлок בשלב ה-decode).

.3. תנאי מספיק לכתיבה ל-PC (דлок בשלב ה-fetch, או בעת קפיצה בלווית 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 נוסיף תמייה ל-addi. הוספה די דומה למקרה החד-מחזורי. עדיף לשנות כמה שיטור רק את ה-jzler, 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 שינוי את הדגלים כך שנסכום בין רגיסטר ל-immediate ולא נכתב לזכור (אבל כן ל-RF).

הערכת ביצועים

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

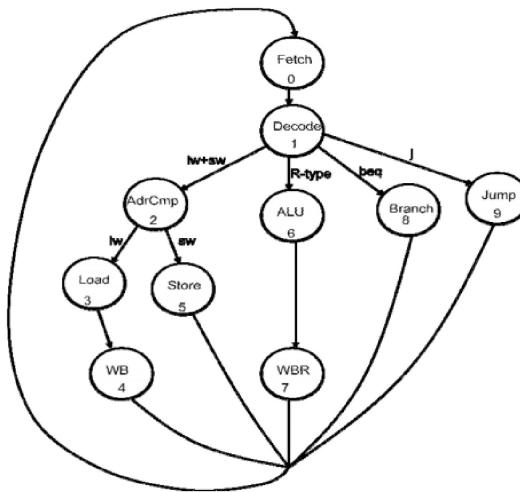
$$\text{Speedup}_{\text{overall}} = \frac{1}{1 - \text{Fraction}_{\text{enhanced}} + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

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

דוגמה שינו מעבד כך שעטה פקודות בנקודה צפה ירצו פי 2.5 יותר מה, גישה לזכור פי 3 יותר 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) שהיא מהירה יותר מהזיכרון הכללי - הלא הוא המטמון! במעבדים מודרניים, המטמון hei 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 משניים 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,

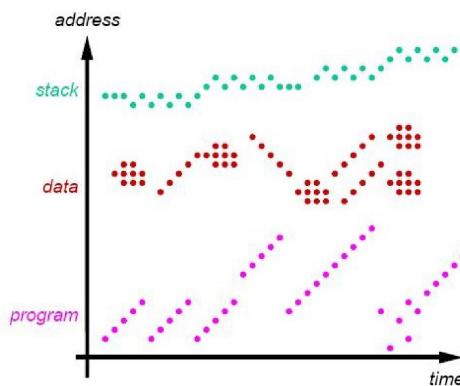
פעמיים :

- לוקליות בזמן :
 - בדואיה: 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.

• לוקליות למרחב :

- בדואיה: 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.
- בлок: 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 בлок במטמון בו.

הרענון הכללי הוא להחזיק חלק קטן מהזיכרון, ולצורך כך צריך למפות חלקים בזיכרון אל תוך המטמון. הזיכרון הרגיל מוחולק (וירטואלית) לשורות (בלוקים) בגודל טיפוסי בין 64 ל-128 בתים. המטמון יחזק 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 בית וולדירות.

בעת מענה לשאלתה, נשווה את כל התגים לכל הכניסות הוולדיות (בית וולדירות 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 חמיש פיספושים, שתי פגעות ועוד פיספוס.

החסרון ב-FA הוא שבגלל שיש לנו בלוק בסדרי גודל של C^{15} , קיבל שחחיפוש הופך למרכיב דומיננטי בתזמון והוא גורם להחמרה בביצועים (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 כMOV FFFF, ויש לה ST 00, لكن נכניס את השורה לכינסה הראשונה (הAPSIX).
- 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 מסויים, 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 אפשר עדין להימנע מקפיצה וכתייה ל-PC (כבר אז ידוע לנו האם אנחנו ב-branch והאם תוצאה ה-ALU היא 0 או לא). החישוב האם לקפוץ או לא נעשה עוד בשלב ה-execute (כבר אז ידוע לנו האם הוא השלב branch-ו והאם תוצאה ה-ALU היא 0 או לא). החסרונו בכך הוא שהוא מאריך עוד יותר את שלב ה-execute שבלאו הכל הוא השלב הארוך ביותר מבין השלבים השונים. בהתאם למימוש המעבד,

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

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

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.

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

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

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

ה-throughput לא משתנה כי אחורי שנבעע את חמישת השלבים הראשונים עבור ההוראה 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 האם 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/MEM 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 נדרס 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), ולכן 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, יוכל באמצעות pipeline שבו גישה לזכרון היא במחזור יחיד.

ביצועים של מטמון

הגדירה Miss Rate הוא חלקיות הפספוסים מתוך סה”כ הבקשות Hit Rate-Hit Rate-Miss Rate. Miss Rate (AMAT) Acess Time.

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. מהו CPI-*control* 20%-*i*st?

$$CPI = CPI_{ideal} + \text{זמן לפקודה hazards}$$

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

$$= 1.1 + 1.5 = 2.6$$

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

סקולול תמורה (טרריידוף) גודל הבלוק

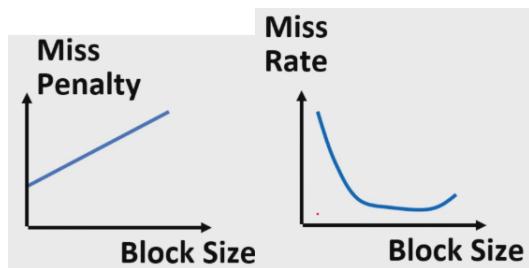
- יתרונות של בלוק גדול:

- 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.

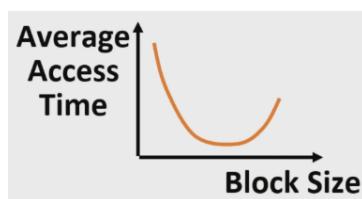
המטרה שלנו היא למזער את ה-AMAT.

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), שmbיאו אליו latency יותר גבוהה במקרה קרייה מהזיכרון.

דוגמה נניח שזמן הגישה ב-Hit הוא 3 מחזוריים ל-L1 ו-6 ל-L2, שה-Miss Rate ב-L1 ו-ב-L2 הוא 8% ו-3% 分別. Miss Penalty של L2 הוא 200 מחזוריים.

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

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

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 מילימט), כלומר העלנו את ה-throughput latency (בכפוף לזה ששני גודל הבלוק במטרמון 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 שמופות לסדרוגין. כך שתי גישות לזכרון שמייעות לרכיבים שונים יכולו לפעול 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 משותה היבט לוקאליות בזמן (במקרה של מחסנית בעת ריצה על מערך יצא שנזרוק לא מעט ערכים במחסנית שווה לא אידיאלי).
- LRU (Least Recently Used) : נגרש את הבלוק הכי פחות בשימוש. זה משותה היבט 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).
היתרונו 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).
היתרונו המרכזי הוא שזה מוריד את הניצול של הזכרון וה-latency בעת פעולה כתיבה. החסרונו המרכזי הוא שיש חשש לאובדן מידע (בעת כיבוי המחשב, לדוגמה), ובנוסח' שכחיש כמה תהליכיים שרוצים בו זמינות צריך לבצע בדיקות קוחהנות כל הזמן (snoops).

בדומה, יש שתי אפשרויות למימוש הכתיבה במטען במקרה של Miss.
 • Write Allocate : נקראת המילה מהזכרנו למטען (כאילו היה לנו Miss בקריאה) ואז נדרוס את הערך המטען עם מה שנרצה לכתוב. זה שימושי במקרה שאנו מוצפים לכתיות נוספות לאותו בлок (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 איוו)

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 בлок 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 פנימי)?

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

```

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

```

אנחנו ניגשים לתאים $(0, 0)$, $(1, 0)$ וכן' כאשר כל אחד מהם מקבל Compulsory Miss (כי הблок 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 (локח 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

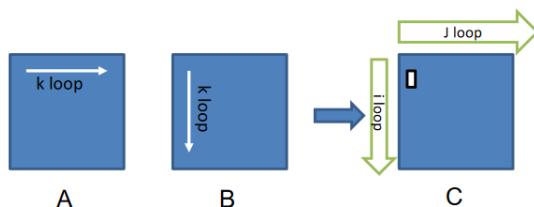
```

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

בבחירה הזוכרו, חייב להתקיים עקרון הכלכלה, כלומר שכל מה שモטמן בשכבה גבוהה בהכרח נמצא גם בשכבה נמוכה יותר.

הערה עקרון הכלכלה נדרש כדי שנוכל לחפש איטורטיבית מידע משכבה גבוהה לנמוכה ולא לנחש שהיא בשכבה אחת, לפחות שם נכתב אליו או הערך מהשכבה לפני לא יدرس 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 בלוק), 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 תתפוס 19 ביטים, כלומר מערך התגים תופס $2^{10} \cdot 19 + 2^{10} \cdot 2^2 + 2^5 = 2^{10} \cdot 19 + 2^{10} \cdot 4 + 2^5$ ביטים.

שבוע X | זכרון וירטואלי

הרצאה

כל תכנית מניחה שהזיכרון שלה מסודר לפי פורטט שכבר ראיינו בקונבנציות של פ' - סגננטי Text, BSS, Heap וכו'. אם יש לנו כמה תהליכיים, נctrיך שמערכת הפעלה תציג מציג שווה לתוכנית שאכן כך זה נראה, גם אם בפועל זה לא. בנוסף, נרצה שתהליכיים יכולים לגשת לזכרון גודל יותר מאשר זה שקיים פיזית, ולשם כך נctrיך לעשות לזכרון וירטואלי מספק לנו גם בידוד בין תהליכיים. לסום, זכרון וירטואלי נותן לנו אורתוגונליות בין השימוש בחומרה של הזיכרון לבין ריצה של תוכנית מעליו.

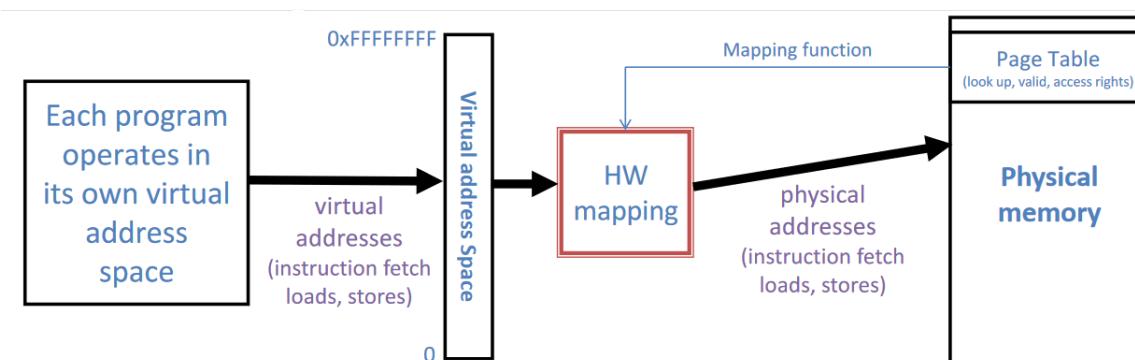
כל תהליך יש מרחב וירטואלי אחד. נפריד את מרחב הכתובות הווירטואלי לדפים. נפריד את מרחב הכתובות הפיזי למסגרות באותו הגודל כמו הדפים (לרוב 4KB).

ניפה לזכרון הפיזי רק דפים שבהם התהיליך משתמש בריצה שלו (זה תמיד יהיה משמעותית קטן ממרחב הכתובות הווירטואלי כולם). דפים שאנו לא משתמשים בהם נשמר בדיסק, כדי שנוכל בהמשך להביא אותם לדפים בזיכרון לשחזור לשימוש בהם.

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

כך תראה ייחידת התרגומים והקריאה 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 (וואולי גם ביטים לצורך מדיניות גירוש כמו LRU).

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



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

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

דוגמה תרגום הכתובות נעשה כמו שעשינו במתומו: נניח שיש לנו מרחב כתובות וירטואלי בגודל 2^{32} בית ומרחב כתובות פיזי בגודל 2^{30} בית וגודל דף $= 4KB = 2^{12}$. כתובות וירטואלי בת 32 בית תורגם כך: נשמר לצד את החיסט, שגודלו 12 בית וניקח את ה-20 ביטים הגבוהים של הכתובת, ונתרגם אותם לאינדקס של מסגרת בזיכרון הפיזי (Physical Page Number) באמצעות Page Table. לאחר מכן נצמיד את האינדקס הזה חזרה להיסט וזה תהיה הכתובת במרחב הפיזי.

כל כניסה ב-Page Table לוקחת 1 (20 – 12) ביטים (הוספנו 1 לוולדיות) אבל כדי שייהי נוח לגשת, לרוב נעגל ל-32 ביטים. הטבלה תתפוס סה"כ $2^{32-12} \cdot 2^2 = 2^{22}$ ביטים (כניסה לכל דף).

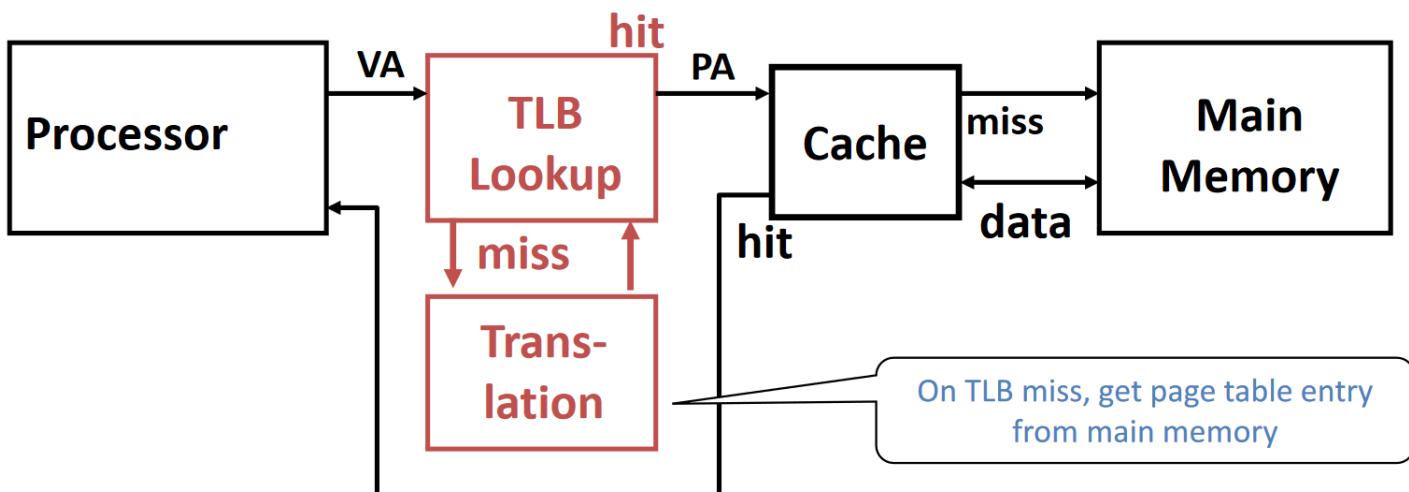
ה-Page Table Register מצביע לבסיס ה-Page Table, כך שאפשר באמצעות סכימה שלו יחד עם אינדקס הדף לתרגם (כפול 4) להגעה לכניסה שמכילה את ה-PPN המותאים.

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

הערה סוג השגיאה זה נקרא interrupt והוא מאפשר לעצור ריצה של תכנית גם אם לא מופיעה העזירה בקורס.

ה-TLB

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



הערה לרוב ה-TLB יהיה בתצורת FA.

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

פורמט הכתובות (של דאטה, לא כולל התג שנדרש למטמון) ב-TLB זהה לזו ב-Page Table (שכן הוא מטמון שלה), כלומר יהיו לנו גם ביטים של dirty ו-validity, הרשאות ומモן הכתובת המתווגמת.

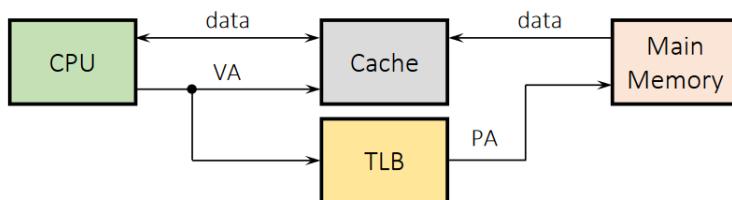
הערה כשבניבור בין תהליכיים ננקה את ה-Page Table (כלומר נקבע את כל הביטים של וולידיות להיות 0), כדי לא להתבלבל בין דפים של תהליכיים שונים.

איך נתרגם אם לא מצאנו ב-TLB

- **מנוהל ע"י תוכנה (MIPS)**: ברגע שיש miss ב-TLB, החומרה תשלח trap וה-OS יריץ את מה שקורה ב-Page Fault. נשים לב שהזיהוי נוסף מעבר למה שכבר יש, שקורה לצורך היביא את זה ל-Page Table. במהלך ה-OS תצטרכ לכתוב לדיסק את תוכן ה-page אם יש בייט dirty דלוק על הדף שגירשנו מה-TLB. תחת שיטה זו ה-Page Table לא עושה יותר מדי כי הטבלה האפקטיבית של דפים שאנו מחזיק היא ה-TLB.
- **מנוהל ע"י חומרה (רוב המעבדים)**: החומרה מביאה מידע מה-TLB (כמו במטמון הקלاسي). במקרה כזה חומרת המעבד צריכה להזכיר את פורמט ה-PTE, אבל ה-OS לא צריכה בכלל.

יעולים לתרגום כתובות וירטואליות

1. להשתמש בחלק מהביטים בהיסט כסטים ב-TLB שאינו FA. כך נדע את ה-way כבר לפני שנתרגם כי הוא נשאר קבוע.
2. להשתמש במטמון על כתובות וירטואליות. היתרונו הוא שכן נדרש לתרגום כתובות פיזיות רק במקרה של פספוס במטמון.



זה בעייתי כי לשני תהליכיים יכולם להיות את אותה הכתובת הווירטואלית וזה התנששות שאנו נא רוצים שתקרה, ולכן בין תהליכיים ננקה את המטמון כדי למנוע שימוש בדפים של תהליך אחר. לחופין, אפשר להוסיף שדה של PID ליד כל PTE וכך נדע לא להתיחס לפגיעות של המטמון על דפים לא שלנו.

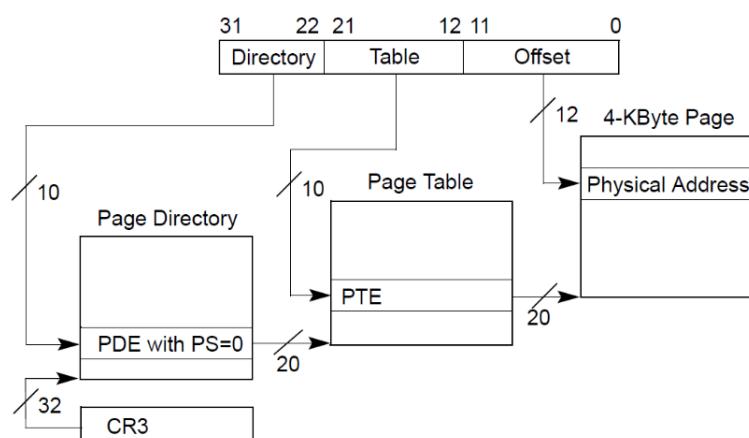
טבלת דפים היררכית

אם יש לנו מרחב כתובות וירטואלי של 64 ביטים וזכרו פיזי של 32 ביטים ודף בגודל 2^{12} בתים. במקרה כזה ה-Page Table עצמו יתפוז $2^{52} = 2^{64-12}$ כניסה שווה יותר מכל הזכרונו הפיזי עצמו. מה שמצויל אותנו היא העבודה שכמעט בכל הכניסות בטבלה אנחנו בכלל לא משתמשים.

כדי לפטור את הבעיה נשתמש בטבלת דפים היררכית. תהיה לנו טבלת דפים שהוא השורש, שכל הכניסות שלו מצביעות לטבלות שנייניות, שמצביעות לעוד טבלות עד שנגיע לעליים שבהם טבלאות שמצביעות באמת ל-PPN-ים. את האינדיקציה נעשה באמצעות חלוקת הכתובת הירטואלית שנותרה לאחר הסרט ההיסט בתוך הדף לכמה סגננטים (*x* הביטים הראשונים משמשים לטבלה הראשונה, *y* הבאים לטבלה השנייה וכן הלאה).

אמנם הטבלה במלואה יותר גדולה, אבל אם רק נשמר בזכרו הפיזי את הדפים שכולים את טבלאות לא ריקות, נידרש למשמעותית פחות זכרו לאחסון הטבלה ההיררכית.

דוגמה ב- 32×32 בית יש לנו טבלה היררכית עם רמות היררכיה אחת, כמו ה-Page Directory (קרויה קריונית) שמצביע לטבלאות רגילות (מצביעות לדפים), ראו איור



דוגמה נתון מעבד עם מרחב כתובות ב-64 ביט המוחולק לדפים בגודל 2^{21} בתים. המעבד מחובר לזכרו בגודל 2^{46} בתים ודפים ממופים עם טבלה לא היררכית. נרצה להריץ 4 תהליכיים שנמצאים בזכרו בו זמן, האם זה אפשרי? אם כן, כמה זכרו נדרש לטבלאות הזכרו של כל התהליכים יחד?

נצריך 21 – 46 ביטים ל-PPN (הורדנו את הביטים של ההיסט) ואם נעגל ל-32 (בשביל הביטים הנוספים ב-PTE) נקבל שככל כניסה דורשת 4 בתים. לכן כל טבלה דורשת $2^{64-21} = 2^{43}$ בתים לכל תהליך. כך נוכל להחזיק לכל היותר טבלאות של שני תהליכיים בו זמן, אבל לא 4, כי זה ידרוש 2^{47} בתים בזכרו הפיזי, שאין לנו.

אינטראפטים

נרצה לעיתים לעצור את ריצת התוכנית באופן לא מתוכנן, לדוגמה במקרה של Page Fault. נרצה שזה יקרה באופן שקויף כך שהתוכנית לא תשים לב שהיא נעצרה, ולכן החאנדרל של ה-*motion* (שהיא השגיאה שגורתה בעיצירת הריצה) צריכה לשמור את המצב שהתוכנית רואה כפי שהיא לפני השגיאה.

מקרה נוסף כזה קורה במקרה של Context Switch, כאשר נחנכו עוברים מהרצת תחילה אחד לאחר כי נגמר סלוט הזמן שהוקצה לתהילה.

בפועל זה ממומש עם רגיסטר Exception Program Counter כך שלא נדרש לגעת ב-PC של התכנית. הדבר הראשון שיקраה בקוד שאליו מצביע ה-EPC הוא שבירת כל הרегистרים שההאנדר מתכוון לגעת בהם למחסנית נפרדת מזו של פ' רגילה, ואז כשהוא יחוור הוא ישחרר את הערכים הללו חוזה לרегистרים.

לדוגמה, אפשר פשוט להוסיף עוד מצבים שנעבור אליהם אחרי ה-fetch או אחרי ה-execute, או אחרי ה-memory access exception (memory access overflow exception, ב-overflow exception, וב-exception, ב'-trap'). זה קורה במקרה של overflow וכיוצא' ב-, אם מתקיימים תנאים שוגרמים לכך (בעבור fetch ב-exception, או ב-access במקרה שקוראים או כתבים מכתובת לא חוקית).

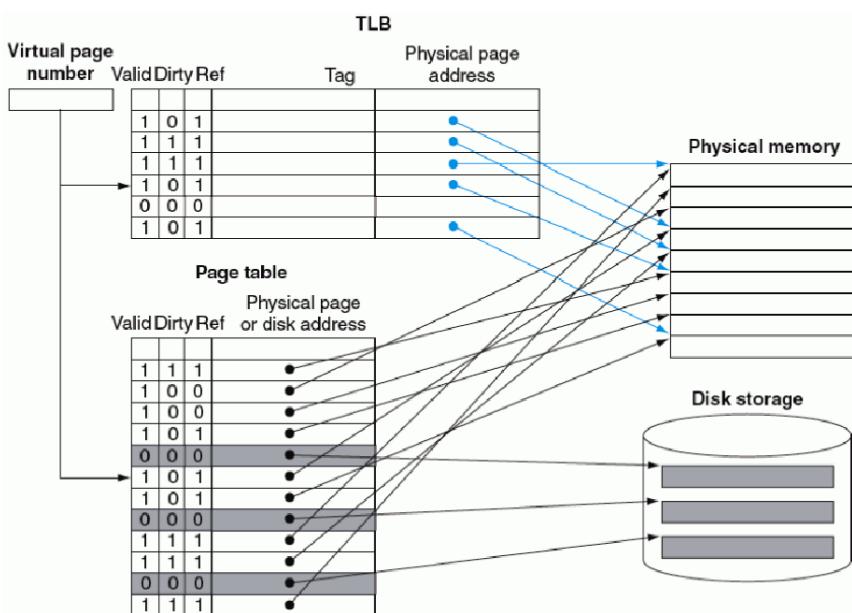
המעבד יחזק רגיסטר Cause שמחזיק ערך שמתאים לסיבת שגרמה לשגיאה, כך שנוכל לדעת מה ערך לו את הריצה והאם צריך לעשות משהו בעניין. נוסף על כך, ניתן לשלב ה-fetch מרובה שיאפשר קפיצה ל-EPC במקרה של Cause שאינו 0.

תרגול

הערה יש חשיבות רבה לרכיב האבטחה ב-indirection בין הזכרון הווירטואלי לזכרון הפיזי וشبירת הרציפות. כך וירוסים לא יכולים פשוט לגשת לזכרון שלו באמצעות buffer overflow ודומהם.

IMPLEMENTATION של LRU הוא מתוגר בתוכנה. אפשר לחלופין להשתמש ב-Not Recently Used (NRU), כלומר נסיר דפים שלא היו בשימוש לאחרונה ולא בחירה העתיקה ביותר. בנוסף ביט reference שערך יהיה האם ניגשנו לדף. אחת לכמה זמן, מערכת ההפעלה תבצע 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 מההרשאות).

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



מהלך קריאה מהזכרן

נתונה לנו כתובות וירטואלית.

- ניגשים ל-TLB :
 - יש פגיעה? קוראים מהמטמון ואם לא שם אז מהזכרן הפיזי וסימנו.
 - יש פספוס? נשים לטלת הדפים :
- * אם המידע בזיכרון (יש valid=1 בטלת הדפים), מכנים את ה-PTE ל-TLB וממשיכים ממש (קוראים מהמטמון ואם לא שם אז מהזכרן הפיזי).
- * אם הוא לא, יש Page Fault, שדורש טיפול של ה-OS וגישה לדיסק (ואז מפעעים לזכרון הפיזי ומשם למטרמון וכו').

דוגמה נתון מרחב כתובות וירטואלי ברוחב 32 בית המחולק לדפים בגודל 8KB ו-TLB עם 256 כניות.

- כמה שורות יהיו ב-Page Table ?
היחסט דורש 13 ביטים כך שיהיה 2^{19} דפים, כמספר השורות ב-PT.
- כמה שורות מה-PT יתמכו לכל TLB בהנחה שה-TLB הוא Direct Mapped.
יש 2^{11} כניות ב-TLB, כלומר 2 כניות לכל כניסה ב-TLB.
- איך נראה חלוקה של TLB הוא virtual address לשדות בהנחה שה-TLB הוא Direct Mapped ?
13 ביטים להיחסט, 8 לסט ושאר ה-11 להיסט.

דוגמה נתון מרחב כתובות וירטואלי של 32 בית המחולק לדפים בגודל 4KB עם 256 כניות שהוא Direct Mapped וזכרון הפיזי בגודל 256MB. בנוסף הנתונות הנקודות 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.

נשمرים 12 ביטים להיחסט בתוך הדף ולכון 20 הביטים הראשונים (שם 5 האותיות הראשונות) משמאלה הם ה-VPN. הסט של ה-VPN זהה ב-TLB הוא האות הרביעית והחמישית (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 בנתון של גודל הזיכרון הפיזי (モוטיב שיחזור על עצמו גם ב מבחנים).

דוגמה 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 הזיכרון הפיזי?

ה-PFN הוא 12 ביט וכאן יש לכל היוטר 2^{12} מסגרות פיזיות. כל מסגרת היא בגודל 2^{13} בתים וכאן סה"כ המרחב הפיזי הוא בגודל לכל היוטר 2^{25} בתים.

- מהו גודל ה-PT?

נספור את מספר ה-VPN שלו; מתוך 32 ביט, 13 מוקצחות להיסט בתוך הדף הווירטואלי וה-19 האחרות ל-VPN, כלומר יש 2^{19} VPN-ים וכמוהם כניסה ב-PTE. כל שורה ב-PTE תופסת 16 בתים וכאן סה"כ מדובר ב- 2^{23} בתים, שהם 2^{20} בתים.

שבוע II | חישוב במקביל

הרצאה

נרצה להציג את ביוצרי התכנית שלנו. נוכל או להריץ סריאלית (סדרתית) יותר מהר (לקצר את זמן המחזור, שימוש במתומו ועוד), או לפרק את התכנית להרבה חלקים קטנים שניצן להריץ במקביל.

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

טකסונומיה של מקבול

- מקבול מערכות : פירוק המשימות השונות מתוך המחשב כולם כמערכות (תהליכי לדומה) והרצתן במקביל.

דוגמה במשחק מחשב יש הרבה רכיבי תוכנה שונים: מעבד ללוגיקה, מעבד לרפיקה, מעבד למדיע שmagical מהאינטרנט (במשחק מוקו). נוכל 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 באזנה תוכנה והרצתן במשאבי עיבוד שונים.

דוגמה במעבד גרפי יש כמה שלבים pipeline עד לתצוגה, כאשר היחידה האוטומטית בגרפיקה היא פוליגונים, שהופכים לפרגמנטים, וכו'. כל הפעולות כאן אמנים תלויות אחת בשניה, אבל כל שלב דורש סוג חישוב/פקודות שונות מאוד, ולכן רכיבי עיבוד שונים עובדים בשלבים שונים ואת זה ניתן למקביל. מסיבה זו במחשבים רבים יש GPU שהוא ספציפי לגרפיקה.

- מקבול תת-משימות: פירוק תכנית להרבה משימות אחידות וקטנות הרצתן במקביל.

דוגמה בעת שאלתה למסד נתונים, כל מעבד יירץ את השאלה על חלק אחר במסד, כאשר המשימות (זמן הריצה שלהן) זהות לחלוטין.

אם המשימות לא איחודות אבל יש לנו מעבדים איחדים, אפשר לבנות תור משימות שכל מעבד יקח ממנה משימה ברגע שישים את הקודמת, ואז בתוחלת נקבל ביצועים שווים בין המעבדים.

- **מקובל פקודות**: הרצת פקודות אסמליל במקביל.

דוגמה במעבדים מודרניים מרכיבים כמה פקודות במקביל כי יש להם כמה ייחדות execution וכמה ייחדות issue (פענוח). ההבדל ביחס למקובלים האחרים היא שהמקובל הוא ברמת המעבד, והתכנית עצמה לא משתנה (זה שקווי למשתמש).

- **מקובל DATAה**: הרצה במקביל של משימה על חלקים שונים מיידיע.

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

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

הטקסונומיה של Flynn

- **(SISD)** Single Instruction Stream, Single Data Stream : מעבד יחיד.

(SIMD) Single Instruction Stream, Multiple Data Stream • (Vector Machines, CM-2) בלילה אחת מקובל ברמת הנתונים.

(MIMD) Multiple Instruction Stream, Multiple Data Stream • : חישוב בקלאסטרים - מקובל ברמת threads.

(MISD) Multiple Instruction Stream, Single Data Stream • : לא שימושי לשום דבר חוץ מאולאי אבטחה, אז נרצה להריץ כמה תוכנות שונות את אותו הדבר ונבדוק שהן קיבלו את אותן התוצאות, במקומות למשל.

سنכרון ות'רידים

בעת הרצת במקביל של ת'רידים, הבעיה המרכזית שצורך לפתור היא סנכרון (ניהול גישה למשאים מסוימים במקביל). פתרונות סנכרון מאייטים את הריצה של ת'ריד אחד, אבל שומרים על חוקיות הדאטה, ולכן אי אפשר למקבל כל דבר כי לפעמים הסנכרון יעלה את זמן הריצה ליותר זמן מאשר הריצה בת'ריד אחד. ת'רידים הם תת-יחידות חישוב שמוכלים יחד בתהליכיים, כך שימושות בהם מרחב זכרו.

הערה תהליכיים חדשים נוצרים ע"י קריאה ל-fork syscall שנקרא `fork` בזמן ריצת התהליך. יוצר תהליך זהה לחלוטין לו שיכר אותו, מבחרית המקום בתכנית שמננו ממשיכים והזכרו של התכנית, בלבד ה-pid, שבאמצעותו מזוהים האם אנחנו תהליך אב או בן. החל מהפיצול, שינויים במרחב הזכרו של תהליך אחד לא משפיעים על الآخر.

מודלי זכרון משותף ב-Multi-Processing

- Symmetric Multi-Processing : לכל המעבדים יש זכרון יחיד משותף. במודל כזה גישה היא מהירה ויעילה אבל בסKİיל זה יהיה איטי. בנוסף, כל שימוש לפתמה 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 מתי לגשת לזכרון שלו ומתי למשותף), אבל בסKİיל עובדת הרבה יותר טוב (רוב הזמן המעבדים יעבדו עם הזכרון שלהם, ורק חלק מהזמן יגשו לזכרון המשותף שלו throughput מוגבל).

הערה ה-PT מאפשר למחרב זכרון פרטני שונה לכל תהליך, כי לכל תהליך יש PT משל עצמו.

דוגמה תקשורת בין תהליכיים : ב-xInfiniBand ניתן לעשות זאת באמצעות管道ים, שהם אובייקטים שהתחליכים כותבים וקוראים להם, ויכולים לחכות למידע בהם וכוכ'.

מה צריך בשבייל לעבוד עם Multi-Threading ?

- פעולות אוטומיות בעת ריצת ת'רדים במקביל של אותו התהליך מאפשרות שמיירה על חוקיות המידע (פעולות שאינן אפשרו לעצור באמצעות).
- זכרון משותף לכמה ת'רדים, שהוא הזכרון המרכזי של התהליך, כך שוכולים יוכלו לגשת לאובייקטי הסנכרון.
- מערכת הפעלה שתנהל הכל, לרבות מימוש אינטראקטיבית להערה או הרדמה של תהליכים, שימושיים אותנו בהקשר הזה לעדכון תהליכיים שונים על סיום הכנת המידע לתהליך אחר.

הרצה במקביל של ת'רדים במעבד

- מעבד עם ת'רד יחיד מריצ' בכל מחזורי שעון את אותו הת'רד, אבל כמו פקודות שלו במקביל (זו ההגדשה של superscalar).
- מעבד עם מריצ' בכל מחזורי שעון פקודה אחת או יותר של ת'רד אחד, ועשה לת'רדים interleaving Fine-grained Multi-Threading ברמת המחזורי.
- מעבד עם Corase-grained Multi-Threading מריצ' תכנית עד שהיא נתקעת, ואז עובר לתוכנית אחרת עד שהיא נתקעת וחזר חלילה. כמו הנו'ל רק שהחלפה קורת ברמת הבלוקים של קוד ולא ברמת הפקודה.
- מעבד עם GPU General Purpose Coarse-Grain ומעבדים ב-Grain General Purpose עובדים ב-Grain.
- מעבד רב-liveti מריצ' בכל ליבה (מעבד בפני עצמו) תוכנה אחרת, ולא מערבב בין ת'רדים באותוה ליביה.
- מעבד עם Simultaneous Multi-Threading מנסה לשבע פקודות בLivetiות כמה שיותר צפוף, לעיתים אפילו פקודות של כמה ת'רדים באותו מחזורי. למנגנון זה נדרש להחזיק גיסטררים לכל אחד מהת'רדים שלנו, וארכיטקטורת זה מאוד מתאגר אבל עובד מאוד טוב.

מודלים לגישה משותפת לזכרו

1. לכל המעבדים יש זכרו אחד גדול ומשותף, והמקובל בזיכרון הוא בגישה SMP. לכל אחד יש מטמון שלו מעלה הזיכרון, אליו ניתן לגושים כולם דרך אוטו bus, שהוא צוואר הבקבוק כאן. במודול כזה קשה להרחיך תהליכי שניגשים לזכרו כי מהר מאוד מגיעים לרוחה ביביצועים.

היתרון בIMPLEMENTATION כזה של מקובל הוא שהגישה לזכרו היא אחידה, ולא נדרש פרט IMPLEMENTATION מימוש מתאגרים מעבר לפתרון בעיות סync'רו החסרו.

2. יהיה לנו מעבד אחד שומרץ הרבה ת'דים במקביל, כך שלכל הת'דים יש מטמון משותף ומשabi חישוב משותפים (לרבות זכרו משותף). היתרון כאן הוא שבזמן שת'ד אחד מחליף לתשובה מהזיכרון, נróżג ת'ד אחר.

3. יהיו לנו כמה מעבדים, כל אחד מרוזג תהליך עם מטמון שלו, והוא ניגשים לזכרו משותף (לוגית) ת'ד שמנומש כזכורות שונים (פיזית) שניגשים אליהם ת'ד רשות (במקום bus). נוכל לגשת לכמה זכרונות במקביל ת'ד הרשות. כדי להוכיח את התקורתה של הרשות, נctrax בלוקים מאוד גדולים ב-cache כך שה-latency יהיה קטן ביחס לזמן של העברת המידע עצמה.

הערה: נניח שה-sus יכול להעביר בכל פעם 32 ביטים של מידע מהזיכרון. כדי להביא שורה של 512 ביטים למטרונו, ה-sus ייחזר לנו את המידע בנתודות של 32 ביט, כשכל סט של נתודות נקרא Burst (פרץ). ה-sus מאוד יקר חומרתי כי הוא צריך להיות מאוד מהיר על פני מרחוקים מאוד גדולים, וכן הוא עדין ונחשב צוואר בקבוק.

קוהרנטיות מטמון

לאחר ביצוע פעולות על כתובות זכרו, יש במטרונו ובזיכרון ערכים שונים. כדי מעבד אחד עם מטרונו אחד וזכרנו אחד, זו לא בעיה. כאשר שני מעבדים, זה יוצר מצב של חוסר-קוהרנטיות - מעבד אחד קורא מידע לא מעודכן מהזיכרון שנמצא ברגע רק במטרונו. יש לנו שתי דרישות:

- **קוהרנטיות סדרתית:** כל רמות הזיכרות מחזיקות באותו ערך לאוותה כתובות זכרו (או לפחות המימוש חושף ממשך שימושיים זאת).
- **כתיבות לזכרו נעשות באופן סדרתי:** אם שני תהליכי כתובים לזכרו (קודם למטרון שלהם), A ולאחריו B, אז הערך של B הוא זה שיופיע בזיכרון.

עקרונות בIMPLEMENTATION קוהרנטיות

- **מיגרציה:** ברגע שתהליך כתוב, המידע עובר באופן שקווף לתהליכי אחרים.
- **רפליקציה:** ברגע שתהליך קורא, כל העותקים של המידע שהוא רוצה הם העדכנים ביותר.

דוגמה: נממש קוהרנטיות עם bus משותף בסגנון Invalidate-Invalidate. כתהליך יכתוב לזכרו, תשלח bus הודיע שמודיעת לכל (המטרונים של) התהליכים שהערך של הכתובת שאליה כתבו השתנה והעותק שלהם כבר לא ולידי. לאחר מכן, במקרה שתהליך אחר יקרא את הכתובת הוא יראה שהכניתה במטרון כבר לא ולידי ויקרא אותה מחדש מהזיכרון.

הנחה פה היא שהמטרון הוא Write-Through כדי שהעדכנים של הזיכרו יהיה מיד בגישה ולא רק בגירוש.

דוגמה פרוטוקולים מוכללים לזה שמאפשרים שימוש במתמוני Write-Back נקראים MESI, שמאפשרים לכטובת להיות באחד מארבעה מצבים הנראים באירא.

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

תרגול

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

פרימת לולאות

בסוף כל איטרציה של לולאה, המעבד צריך לבצע conditional branch, שהוא תליי בערך שחווש במהלך הלולאה - מקום מועד ל-stall. כדי למגר את מספר ההמתנות בסוף כל לולאה, יוכל למשם כמה חזורות בתוך כל איטרציה, כך ששה"כ תהיה כמות קטנה יותר (ביחס כפלי) מאשר מספר הלולאות המקורי.

דוגמה את השורות ממשמען לחץ ניתן לשורות מימין, כאשר עתה יש לנו חמישית ממספר האיטרציות המקורי, ללא פגיעה בנסיבות 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);
}
```

לחרון central מרכז מסpter הפקודות המוגדל בלולאה מעמיס על מיטמו הפקודות (וגם תופס יותר מקור בזיכרון הפקודות).
בנוסף, מסpter הרגיסטרים שישמרו ווצאות ביןיהם של הקריאות בלולאה יגדל (מ-1 למספר הפרימות).

הערה יש לשים לב במקרים הקצה בהם מספר האיטרציות הכללי לא מתחלק במסpter החזרות שנכול ידנית בתוך הלולאה.

דוגמה נתון הקוד הבא

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

קוד אסמלטי אפשרי לו (קליני, 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 ולכטوب לרגיסטר באותו מחזור וב-branch freeze עד להגעת התוצאה. אילו stall-ים המעבד יכניס?

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

- נשנה את סדר הפקודות; נעדכן את \$s0 כבר בהתחלה וنبצע branch לפני ה-sw - לא נוכיח נשמרות שיקילות אבל היא כן. עתה

ה-stall-ים (ושינוי הסדר) יראו כבאיור

```

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

```

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

בשימוש ב-unrolling ננתח פקודות לפי שתי מחלקות: עבודה אמיתית (קוד מהותי שמחשב דברים) וטיפול באיטרציה (הוזת אינדקסים והשוואות).

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 unrolling עם 4 פרימוט ונקבל את קוד האסמבלי הבא

```

Loop:    lw      $t0 , 0($$0) stall
          add    $t1, $t0, $$1 stall
          sw     $t1 , 0($$0) stall
          lw     $t2 , -4($$0) stall
          add    $t3, $t2, $$1
          sw     $t3 , -4($$0)
          lw     $t4 , -8($$0)
          add    $t5, $t4, $$1
          sw     $t5 , -8($$0)
          lw     $t6 , -12($$0)
          add   $t7, $t6, $$1
          sw    $t7 , -12($$0)
          addi  $$0, $$0, -16 stall
          bne   $$0, $$2, Loop stall

```

עתה קיבלנו $4 \cdot 4$ -stall + $2 + 2$ + 2 + 1 = 19 אירור מודגמים ה-stall-ים על הפרימה הראשונה) וסה"כ 33

פקודות, כולל 8.25 מהזרי שעון לאיטרציה (מקורית, לא פרומה), לעומת 7 בלי שום פרימה - זו הרעה ביצועים.

• נוכל לשנות את סדר הפעולות לאחר הפרימה, ולאחר מכן פחות stall-ים, באמצעות קיבוץ סוג הפקודות (כך שבזמן שначכה

לאינדקסים פרומים מוקדמים נחשב דברים נחוצים לאינדקסים פרומים מאוחרים יותר); ראו אירור

```

Loop:    lw      $t0 , 0($$0)
          lw      $t2 , -4($$0)
          lw      $t4 , -8($$0)
          lw      $t6 , -12($$0)
          add   $t1, $t0, $$1
          add   $t3, $t2, $$1
          add   $t5, $t4, $$1
          add   $t7, $t6, $$1
          sw    $t1 , 0($$0)
          sw    $t3 , -4($$0)
          sw    $t5 , -8($$0)
          sw    $t7 , -12($$0)
          addi  $$0, $$0, -16 stall
          bne   $$0, $$2, Loop stall

```

כך שעתה אנחנו עומדים על 3 stall-ים (cols קשורים לטיפול באיטרציה, ולא בעבודה אמיתית), כולל סה"כ 17

מהזרי שעון שהם 4.25 אירור לאיטרציה (מקורית) - שיפור משמעותי לעומת ה-baseLine

אפשר להוציא את מספר המהזרים לאיטרציה מקורית ל-3.5 ע"י העברת ה-addi לתחילת ושינוי ההסתדרים המוחשבים ב-,lw

.sw

נשים לב שמספר הרגיסטרים הנחוצים לחישוב גדול ממשמעותית ביחס ל-baseLine. אילו הקוד בתוך הלולאה היה דורש 3 פקודות לפחות, היו צריכים למצא רגיסטרים נוספים - רכיב נוסף בשקלול תמורה הביצועים בעת ביצוע unrolling.loop unrolling

הערה נזכר שמערכות דו ממדים נשמרים ב-major row, כלומר כל הערכים של השורה ראשונה ([...][A]) נשמרים באופן רציף, ומיד

לאחריהם הערכים של השורה השנייה וכן הלאה. תחת קונפיגורציה כזו, תמיד יותרiesel לעבור לפי שורות בלולאה החיצונית ועמודות

בפנימיות כשרוצים לקרוא את כל המערך, ולא להפסיק.

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];
}
```

- בנהנה שהמערכיים מיושרים בכתובותיהם לזכרון, מה ה miss rate בגישה לנוטונים?

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

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

ל-C אנחנו ניגשים בכל פעם לשורה אחרת, ויש 2^9 גישות לכל? לעומת זאת 2^9 כניסה במטמון ולכן לא נצליח לשמר בлокים במטמון על אותה שורה בין איטרציות, כך שככל קרייה היא פספוס. שני המערכיים אנחנו ניגשים 2048 פעמים בכל איטרציה, כולל miss rate של $\frac{\frac{2048}{16} + 2048}{2048 + 2048} \approx 53\%$.

חשוב לציין שרק בגלל שאחנו ב-LRU וגם רק גישה אחת ל-C בין כל שתי גישות ל-B, הרי שהבלוק האחרון מ-B לא מגורש מהמטמון (אילו היו 2^9 גישות ל-C לדוגמה, הבלוק של B היה מגורש עד לגישה הבאה אליו).

- מהו זמן הריצה של הקוד במעבד pipeline בהנחה שה miss penalty הוא 10 מחזורי, ה branch predictor הוא אידיאלי (כל branch קופץ נכון תוק מוחזר אחד ולא נדרש flush), יש 100% בhit rate בפעולת הפקודות, וכל לולאט זמוקודדת 7 פקודות אסמבלי (כולל stall-ים)?

זמן הריצה הוא

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

(זונחים את זמן הגישה לזכרון במקרה של פגיעה) ובהצבת הערכים קיבל

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

הדוגמה הנ'ל מראה כיצד חישוב פעולות בינהירות על מטריצות היא יקרה מאוד בשל ניצול גרווע של המטמון. ריצה לנצל מספר רב של מעבדים כדי למקבל את הפעולה הזו, ולנצל באופן מיטבי יותר את המטמון.

דוגמה נניח שיש לנו מערכת עם 100 והפ' מחושבת בצורה בלתי תלויה על כל ליביה. בנוסף נניח שה miss penalty גדול ל-100 מחזוריים ו-HAMR משותף בין הליביות.

השינוי היחיד בחישוב הנ"ל הוא ב-penalty miss, כאשר עתה בגל ה

- penalty
- high

 המוגדר כ- $\frac{2048^2 \cdot 113}{100}$ מ- $2048^2 \cdot 1.13 = 15.6$ times speedup.

OOOE

נרצה להציג ביצועים של תוכנה ששורות הקוד שלה קבועות, באמצעות הרצאה לא סדרתית של הפעולות בה. נניח שיש לנו מעבד pipeline עם כמה עותקים לכל שלב (לדוגמא כמה שלב פענוו שיכולים לרוץ במקביל). נוכל לסדר מחדש את השורות כך שכמה פקודות נדרשות פחות שלבים יריצו בזמן שפוקודות ארוכות רצות, וכך נשיג ניצול יותר גבוה.

דוגמה בהינתן שפעולות חלוקה דורשת הרבה זמן ביחס לחברו, את השורות הבאות ניתן למקבל (להתחיל את החלוקה, ובזמן זהה להריץ את שתי פעולות 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 לחכות ועדיין נריץ את השורה השלישית במהלך הזמן הזה.

דוגמה נביט בקוד הבא

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

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

סכנות דעתה חדשות ב-OOOE

- Cache-to-write-after-read - כתפקידו כתובת לרגיסטר לאחר שפוקודה אחריה קורת ממנה. בריצה לא חוקית של הקוד כשייש שימוש ב-OOOE.
- יתכן מקרה בו הפקודה המוקדמת תקרא את הערך החדש שנכתב ע"י הפקודה המאוחרת (סכנה זו כמובן לא יכולה לקרות אם לא משתמשים ב-OOOE).

- Cache-to-cache-after-write - כתפקידו כתובות לרגיסטר, ושל השימוש האפשרי בסדר הרצתן יתכן שהערך האחרון שיכתב לרגיסטר הוא של הפקודה המוקדמת (גם זו ייחדית לשימוש ב-OOOE).

דוגמה נביט בקוד הבא

```
div $t1, $t2, $t3
```

```
add $t5, $t6, $t1
```

```
add $t9, $t5, $t5
```

```
add $t5, $t7, $t8
```

נשים לב כי סכנת-h WAW שנוצרה אינה אינטראיניזית לקוד, משום שرك במקורה שתי הפקודות כתובות לאותו רגיסטר, ולא בגלל שיש בינהין תלות. התנשויות סתמיות כאלה נקראות Dependencies .False Dependencies .False Dependencies מ-OOOE שנובעת ב-TOUCH

Register Renaming

שיטת-h Register Renaming באה לפטור תלויות מדומות בלי אינסוף רגיסטרים. בשיטה זו, נציג למשתמש ממושך (וירטוואלי) של רגיסטרים ארכיטקטוניים \$R0,...,\$Rn, כבמציאות החומרה מספקת לנו רגיסטרים פיזיים \$Tm,...,\$T0. לכל רגיסטר ארכיטקטוני נוכל להתאים יותר מרגיסטר פיזי אחד :

- בכל פעם שפוקודה תבקש לבצע חישוב על בסיס רגיסטרים ארכיטקטוניים כלשהם - נעתיק את אנדקס ה-R בפוקודה לאנדקס ה-T כדי שהערכים מופיעים ברגיסטרים הפיזיים בפועל (קורה בשלב Fetch/Decode). כמובן, ניתן למעבד לחשב את הפעולה על בסיס רגיסטרים שמספריהם שונים מהמקור, אבל שמתנהגים באופן זהה.
- בכל פעם שפוקודה תבקש לכתוב לרגיסטר ארכיטקטוני כלשהו - נעדכן את העתקה כך שככל אנדקס T ידע לאיזה R הוא מתיחס (קורה בשלב Commit, הכתיבה בחזרה לרגיסטרים). בפועל מדובר באולו' שניי ה-R שמתמפה (בין היתר) ל-T הזה.

הערה היתרונו בשיטה זו הוא ש-T-ים שונים יכולים לשמר ערכים שונים של אותו R שאחרת יהיו בסכנת WAW או WAR בגלל תלות מדומה, אבל בזכות הפיזור ההגיוני יותר של הרגיסטרים שומרים על חוקיותם.

הערה החוקיות נשמרת ב-Register Renaming כי את ה-Commit, Fetch/Decode נעשה בסדר הנכוו, וכך גם את ה-Execute, כך שرك ה-h 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
```

בהרצתה 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 רגיסטרים פיזיים.

- את התוצאה של הפעולה הראשונה נשמר ב-T0 (נתעלם מהשאלת מאיפה רגיסטרי ה-R באים בפעם הראשונה).
- את התוצאה של הפעולה השנייה נשמר ב-T1, והיא ת恭ס על סכימה של R4 ו-T0, כדי שמייפינו את תוצאת הפעולה הראשונה.

- את התוצאה של הפעולה השלישי נשמר ב-T2.
- את התוצאה של הפעולה הרביעית נשמר ב-T3, והיא תהיה סכום של R3 ו-T2.

בזכות המיפוי הנפרד של ה-R2-ים בשורות 2 ו-3, השורה הרביעית תקבל בהכרח את הערך שהוא אמור לקבל (הלא הוא תוצאה החישוב בשורה הששית). בדומה, R4 יילקח מאייפשו בשורה 2, שהוא בהכרח לא T3, אליו תיכתב תוצאה החישוב בשורה 4. כך מנענו את שתי הסכנות שהיו כאמור תלויות מאוד.

דרך נפוצה למימוש Register Renaming הוא באמצעות Reorder Buffer (ROB), שמקבל הוראות Decode לפי הסדר ושומר אותן כך שהכניסה בה נמצאת פקודה היא הריגיסטר הפיזי אליו היא צריכה לכתוב. בשלב ה-Execute התוצאה תיכתב לבנייה המתאימה, אולי לא לפי סדר כרונולוגי. לסיום, ה-Commit-ים יבוצעו לפי הסדר, ככלומר שפקודה לא יכולה לעשות Commit לפני שקודמתה עשתה כן, וכך כאמור נשמר על נכונות.

שבוע III | האצת ביצועי ת'רד ייחיד

הרצאה

$$T_{cpu} = T_{cyc} \cdot CPI \cdot IC$$

- כדי למזער את זמן המחזור עברנו ל-multicycle.
- כדי למזער את ה-CPI השתמשנו ב-pipeline.
- כדי למזער את ה-IC, צריך לשנות את ה-ISA.

ב-pipeline-ה האידאלי הוא 1, ככלומר כשאין סכנות, וגודול ממש מ-1CSI. Super Pipelining, ולהציג CPI קטן מאחד באמצעות ביצוע כמה שלבים על פקודות שונות באותו מחזור שעון (להתחילה ALU באמצעות מחזור כשהתוצאה שהוא צריך זמינה). בפועל התדריות עולה וכן ה-CPI יורד.

החסרונות של Super Pipelining הם שהזמן שנחכח כדי למנוע סכנות בקרה (ביטול פקודות עתידיות בגלל קפיצה), שיש יותר סכנות ככלל וכן זיהוי סכנות ומימוש נהים הרבה יותר מרכיבים ברמת החומרה.

דוגמה נניח שיש לנו את הקוד הבא (raiino בתרגול הקודם)

```
for (int i = 0; i < 128; i++)
    A[i] = A[i] + s;
```

את הקוד הזה נתרגם לפקודות האסמבלי הבאות

```

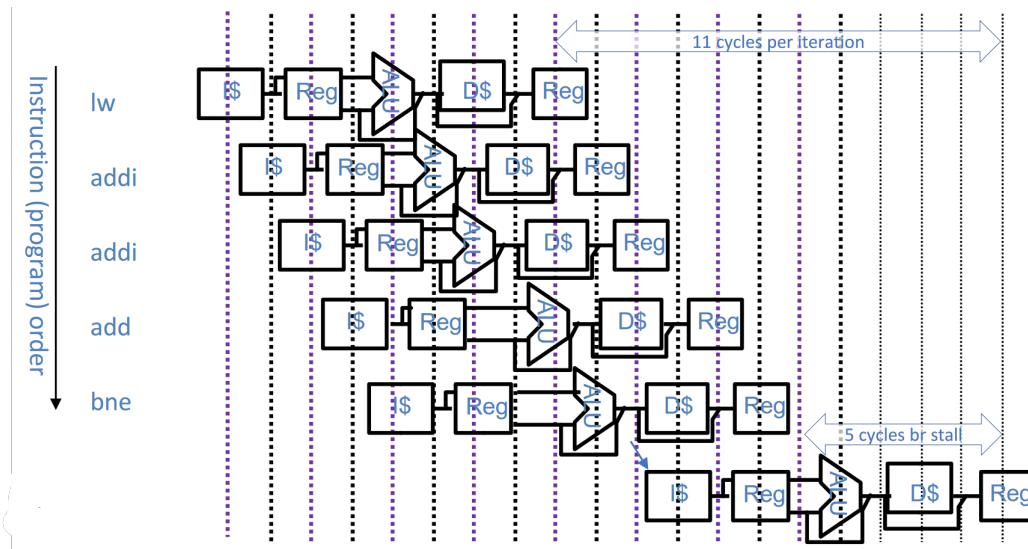
loop: lw $8, 0($5)      ; $8 ← A[i]
      addi $5, $5, 4    ; increment pointer for A[i]
      addi $7, $7, -1    ; decrement loop count
      add $10, $10, $8   ; $10 ← $10 + A[i]
      bne $7, $0, loop   ; Continue if loop count != 0

```

הקוד הזה יקח 7 מוחזוריים כי אחרי ה-`lw` נצטרך לחכות לתשובה ושאר הפקודות סודרו היטב כך שלא יהיו סכנות נוספות.
במציאות המספר המדויק של איטרציות הוא לא מכפלה של איטרציה אחת כי האיטרציה הראשונה לא מגיעה מ-`lw` איטרציה
שלפניה והאיטרציה האחורונה לא קופצת אחרת.

- עבור מעבד Super Pipelined : נתחיל עם פקודת ה-`lw` ונשਬץ את הפקודות על פני מוחזורי שיעו כאשר כל שלב אمنם או רך זמן
מוחזורי שלם אבל הוא יכול להתחילה באמצע מוחזור.
הרביעי, בኒוגד ל-pipeline הקלסי, חייב לחכות ל-`lw` עד שיקרה מהזכרנו (`D$` הכוונה גישה לזכרן ה-`D-I` ו-`I-D`)
ה-add (או ישתמש ב-`bypass` מ-`EX/MEM` לקלט ה-`ALU`). בסוף, ה-`bne` יצטרך לחכות מוחזור אחד כדי להריץ (Instruction Execute-
ExecuteWB) כרגע תפוס ע"י ה-`add`. לסיום ה-`bne` יגרום ל-`stall` של 5 מוחזורי שיעו עד שנגיש לשלב ה-`WB`.

באירוע הקווים השחורים הם מוחזוריים והסגולים הם חצאי מוחזור.

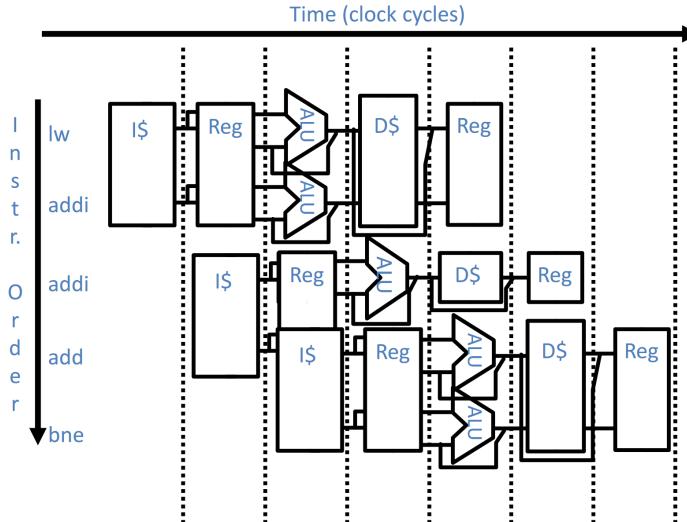


סה"כ קיבלנו 11 מוחזורי שיעו לאיטרציה כולל ה-`stall`, CPI = 2.2 שמשרים = 2.2 × 2 = 4.4 יוטר גובהה. ביחס ל-5.5 יוטר
מוחזוריים לאיטרציה בתדריות וגילה, מדובר בהאצה של רק $1.27 \times$ לمعالץ שרעץ פי 2 יותר מהר.

כדי להגדיל את ה-CPI אפשר להריץ כמה פקודות באותו המוחזור, ככלומר שתהיה לנו חומרה כפולה שתשתמש שתי יחידות עיבוד בمعالץ
במקביל. אם כל פקודה תלויות בזו שלפניה לא השגנו שיפור בכלל, ואם הפקודות ב"ת לחוטין השגנו שיפור של $2 \times$ (או כמספר המקבולים
שהוספנו), שכמובן בפועל לא נגיע לא לקיצון זהה ולא לזה.

דוגמה בפניטים של אינטל היו שני IF ו-ID ורכיב pairing שהחליט האם ניתן לבצע את שתי הפקודות במקביל או לא. שלב ה-Execute היה
נכns לאחד משני שלבים - U-Pipe ו-V-Pipe, כך שלא השגנו superscalar מושלם עוד אז (1993).

דוגמה נחזר לקוד הקודם שלנו, כשבציוו נשתמש ב-Superscalar עם יכול להריץ שני מעבדים בו זמנית (units-2). את הפקודה הראשונה והשנייה ניתן להריץ יחד כי עד ש-addi יזרוס את ה-w, הוא כבר סיים את השימוש בו. במחזור הבא יוכל לקרוא את שתי הפקודות הבאות. הפקודה הרביעית צריכה stall אחד עד שתוכל לשמש bypass שמכיל את תוצאת ה-w, لكن במחזור השלישי נקרא ביחידת הפקודה הרביעית והחמיישית (ואז נבטל את המשך ריצת הפקודה הרביעית כחלק מ-pairing המבוסס על תלות בפקודות, כבאיור).



עתה כל איטרציה דורשת 5 מחזוריים ועוד 2 stall-ים ל-bne, כלומר CPI כללי של $\frac{7}{5} = 1.4$.

הערה ברמת הקומפיילר אפשר להשתמש ב-loop unrolling כמו שראינו בתרגול, ולעrgb את כל השיטות לסייע זמן הריצה יחד. אם מקבצים יחד את הפקודות הקשורות בפרימית הלולאה יוכל לקבל אי תלות דיבוגה וכן להריץ רבות מהפקודות בלולאה הפרומה במקביל Super Scalar.

הערה מעל dual issue קשה מאוד למשוחה והביצועים כבר לא בהכרח שוויים את הcpu'יות בחומרה.

המגמה הכללית של שיפור ביצועים היא העברת הארכיטוט מהחומרה לקומפיילר; כך גם בהאצת ביצועים בת'רד ייחיד.

Very Long Instruction Word

ב-VLIW, החומרה מניהה שהתוכנה מספקת פקודות שאפשר למקבל תמיד. בדומה צו, את סידור החרצה אפשר כבר לחשב בזמן הקימפוף. המימוש יהיה באמצעות מילוט פקודה ארוכות שכוללת כמה פעולות בו זמנית שאין תלויות אחת בשניה. עתה החומרה לא ממשת לוגיקה של ניהול הפקודות המקבילות אלא מרים כל הזמן את מקבצי הפקודות שמוגדרים בפקודה הארוכה - כך נדרש גם לרבות רכיבי חומרה. החסרונות בסכמה זו הם שהקומפיילר פולט אSEMBLY שמותאים בדיקת מעבד אחד (הוא לא portable). ישיהו סЛОטים ריקים כתוצאה מאית התאמת של הקוד למטאורופיז שאנחנו מנסים לעשות לא ושחה-Instruction Cache. יתמלא הרבה יותר מהר וכן נדרש אחד יותר גדול (ירקר בחומרה). בסופו של דבר השיפור ביצועים מאוד תלוי באופי התוכנית.

VLIW מוגדרת ע"י מספר קבוע של פקודות פרימיטיביות. הקומפיילר אחראי על שיבוץ נכון של פקודות כך שייכנסו ל-VLIW.

דוגמה כל פקודה מכילה שני חישובי ALU על שלמים, אריתמטיקה של float-ים, שתי גישות כלשון לזכור ו-branch.

דוגמה נניח שיש לנו VLIW שעשויה bundle לפקודות ALU, ALU, ld/st, br. את הלולאה שמלואה אותו בכל הרצאה נוכל לפרום פעם אחת (שני עותקים בגוף כל ללולאה) ולקבץ לשתי פקודות VLIW. בראשונה נבצע את הסכימה בגוף הלולאה עם הקרייה מהאיטרציה הקודמת ואת שיוני האינדקס של הלולאה. בשנייה נזוז את אינדקס המערך ונוציא לסקום את הערך שקראונו בפקודה הקודמת (תמיד מתבססים על תוצאה ה-VLIW הקודמת). נשים לב שאין תלות בין הפקודות בתוך אותו VLIW, ושהפע"פ שאנו חזו כותבים וקוראים לאווטו הרגיסטר באותה VLIW, הם לא ישיינו אחד על השמי כי הקרייה תהיה מהערך הישן של הרגיסטר.

```

lw $8, 0($5)      ; $8 ← A[i]
lw $9, 4($5)      ; $9 ← A[i+1]
loop:
    add $10, $10, $8   ; $10 ← $10 + A[i]
    addi $7, $7, -2    ; decrement loop count (by 2)
    lw $8, 8($5)       ; $8 ← A[i+2] - prepare for next iteration
    NOP               ; (empty branch slot)
    addi $5, $5, 8     ; increment pointer for A[i] by 2
    add $10, $10, $9    ; $10 ← $10 + A[i+1]
    lw $9, 12($5)      ; $9 ← A[i+3] - prepare for next iteration
    bne $7, $0, loop    ; Continue if loop count != 0

```

אין צורך ב-stallים בין הפקודות ולכן יש לנו 4 מוחזרים (1 ל-VLIW הראשון בתחלת pipeline ו-3 ל-VLIW השני לפני שאפשר להתחילה את ה-VLIW של האיטרציה הפומרה הבאה) לכל שתי איטרציות, שהם 2 פקודות ללולאה.

הערה VLIW הוא מקרה פרטי של Superscalar, שופשט את הבעיה בכך שהוא מאפשר הריצה של פקודות מאוד ספציפיות במקביל.

אין ל-VLIW מנגנון לטיפול ב-cache misses, כמו כל שאר השיטות שהראנו עד עכשוו.

Vector Machines

כדי ליעל תוכנות שמתעסקות בהרבה>Data (באופן ב-“ת”), נוכל להשתמש בפקודות שעובdot על הרבה>Data בו זמנית.

הערה אפשר להשתמש ב-Vector Machines מעל כל שיטת אחרת ליעול המעבד (Superscalar וко'ו.).

דוגמה נניח שאנו כוכמים שני מערכים אל תוך מערך שליש. במקומות לסקום כל פעם שני מספרים שלמים בכל פעם (4 בתים), נוכל לסקום ביחידות של 64 בתים בו זמנית. משמאלו הקוד המקורי, באמצעות רגיל ומימין עם Vector Machines.

# C code	# Scalar Code	# Vector Code
<pre> for (i=0; i<64; i++) C[i] = A[i] + B[i]; </pre>	<pre> addi \$t0, \$0, 64 loop: lw \$t1, 0(\$a1) lw \$t2, 0(\$a2) add \$t3, \$t1, \$t2 sw \$t3, 0(\$a3) addi \$a1, \$a1, 4 addi \$a2, \$a2, 4 addi \$a3, \$a3, 4 addi \$t0, \$t0, -1 bne \$t0, \$0, loop </pre>	<pre> LI VLR, 64 LV \$vr1, 0(\$a1) LV \$vr2, 0(\$a2) ADDV \$vr3, \$vr1,\$vr2 SV \$vr3, 0(\$a3) </pre>

הערה גודל ה-Vector הכי גדול שימושים בו כיום הוא גודל של בלוק במתמNON, כי מעבר לזה ביצועי הגישה לזכרון גרעינים.

דוגמה בעת הכפלת מטריצות מכפילים שורה ועמודה, כשהאחרונה אינה שמורה בזיכרון בכלל השימוש ב-major row. לשם כך הומצאו פקודות LV שקוראות בקביצות (stride) קבועות, וכך ניתן לקרוא עמודה באמצעות פקודה אחת. פקודות gather מאפשרת לקרוא מהזיכרון בקביצות לא קבועות, עם כתובת בסיס ומערך היסטים. הפקודה המשלימה לה בכתיבת זיכרון נקראת scatter.

לצורך המימוש צריך רגיסטרים גדולים (רגיסטרי וקטורי) שיכילו את ייחidot הדאטה הגדולות שהן עוסקות. בנוסף נדרש להוסיף פקודות חדשות שיגדרו פעולה על וקטורים. מעבר לכך, נדרש Vector Functional Units, הכולר ייחdot חומרה שיודעות לבצע את החישובים הוקטוריים (ALU גדול).

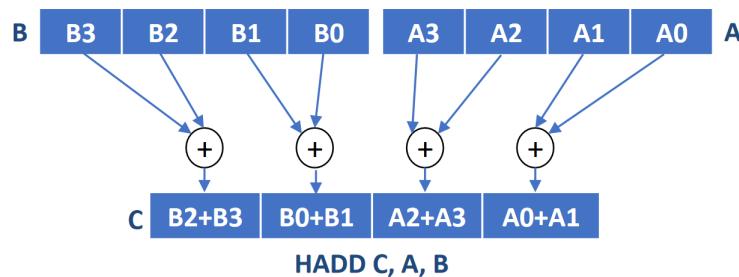
גם כאן אנו משתמשים בפרדיגמת העברת האחריות לקומפיילר, כשהעכשווי הוא יהיה אחראי על מציאת המקבילות, והחומרה רק על המימוש.

הערה כמו ה-WLIV, גם Super Scalar Vector Machines מאפשרות לפרק הפקודה בדיקת הרבה פעמים (על DATAה רבת).

בחלק מהמעבדים יש Vector Length Register שמנגנון גודל הווקטור שלו ממבצעים פעולה. באחרים יש שמאפשר לבצע את הפקודה רק על אלמנטים מסוימים.

דוגמה בעת חישוב על מערך שאינו מתוישר לגודל הרגיסטרים, נוכל לבצע פעולה וקטורית על הכל חוץ מהסיפה שלא מתוישרת, אז לבצע את הפעולה על מה שנשאר ולעשות אז רק לחלק מהבתים (כדי לא לדחוס ערכיהם מחוץ למערך).

פעולה שימושית נוספת בוקטוריים היא רדוקציה, הכולר ביצוע פעולה קבועה על כל האיברים לתוצאה אחת (לדוגמה סכימת כל האיברים ברגיסטר). מימוש יותר יעיל מסכימה אחד-אחד (בלולאה שתלויה באיטרציה הקודמת) הוא סכימה וקטורית בצורת עץ ביןאי, כך שנידרש למספר לוגריטמי של איטרציות לביצוע הרדוקציה (ראו אייר).



דוגמה נחזור לדוגמה האולטימטיבית - סכימת 128 מספרים ממערך. כתוב לולה שסכום 16 מספרים בכל פעם באופן לוג' (נסכם שני איברים סמוכים)

```

Initialize loop count ($7) to 128
Initialize $vr0 to 0
Initialize A[i] pointer ($5) to the base address of A.

1)  loop: lw    $vr1, 0($5)          ; $vr1 < A[i...i+15]
2)      addi $5, $5, 64            ; increment pointer A[i] by 16(*4)
3)      addi $7, $7, -16           ; decrement loop count by 16
4)      addv $vr0, $vr0, $vr1     ; $vr0 - accumulate sums
5)      bne  $7, $0, loop         ; Continue if loop count != 0
6) reduction { hadd $vr0,$vr0,$vr0   ; reduce the 16 elements vector into 8
7)             hadd $vr0,$vr0,$vr0   ; reduce the 8 elements vector into 4
8)             hadd $vr0,$vr0,$vr0   ; reduce the 4 elements vector into 2
9)             hadd $vr0,$vr0,$vr0   ; reduce the 2 elements vector into 1

```

כדי להעביר את הסכום לרגיסטר שאינו קטורי צריך לעשות איזשאו טרייק, כמו לסקום עם .mask.

דוגמה אפליו סכימה של 128 ערכים באופן לינארי (בהתעלם מהחישובים בזמנים) נותנת 7 מחזורי שעון ל-16 איברים, כלומר CPI של 0.44

דוגמה נתון הקוד הבא

```

int dot_product(int *a, int *b, int len) {
    int s = 0;
    for (int i=0; i< len; i++)
        // a[i] is the ith element in the array pointed by a
        s = s + a[i] * b[i];
    return s;
}

```

• תרגמו אותו לקוד אסמבלי.

נוצרך לעקוב אחר ה-movement, כלומר לקבל את הפרמטרים ב-**\$a0, \$a1, \$a2, \$a0**, להחזיר את התוצאה ב-**\$v0** (שם גם

ישמר). הקוד הבא יעבד על מעבד רגיל (ללא SIMD).

```

dot_product:
    add $v0, $0, $0
loop:
    lw $t2, 0($a0)
    lw $t3, 0($a1)
    mult $t0, $t2, $t3
    add $v0, $v0, $t0
    addi $a2, $a2, -1
    addi $a0, $a0, 4
    addi $a1, $a1, 4
    bne $a2, $0, loop
    jr $ra

```

- תרגמו את הפ' לקוד אסמבלי כשותון שיש רגייסטר וקטור $\vec{vec}_0, \dots, \vec{vec}_7$ ופעולות כפל, חיבור וכתיבה וקריאה על רהרגיסטרים האלה.

הकושי יהיה הרדוקציה לאחר המכפלת-סכום מעבר לרגיסטר רגיל. לשם כך נקצת נקרא ל- t_0, \dots, t_3 ערכים מתוך הוקטור וננסכו אותם ידנית

```

// $a0 = a; $a1=b, $a2=len
1. Dot_product:
2.     // assuming $vec0=0
3. loop:
4.     lv $vec1, 0($a0)
5.     lv $vec2, 0($a1)
6.     vmult $vec1, $vec1, $vec2
7.     vadd $vec0, $vec0, $vec1
8.     addi $a2, $a2, -4
9.     addi $a0, $a0, 16
10.    addi $a1, $a1, 16
11.    bne $a2, $0, loop
12.
13.    // horizontal reduction using memory
14.    //      done once at the end of the
15.    sv $vec0, -16($sp)
16.    lw $t0, -16($sp)
17.    lw $t1, -12($sp)
18.    lw $t2, -8($sp)
19.    lw $t3, -4($sp)
20.    add $v0, $t0, $t1
21.    add $v0, $v0, $t2
22.    add $v0, $v0, $t3
23.    jr $ra

```

נשים לב שכאן לא הקצנו מקום לשימוש ה- kd כמו בפ' רגילה, אבל זה בסדר כי אף אחד לא יכול להתערב לנו במהלך הרכיצה ולדרוס ערכים שם (או אנחנו לו) כי אפילו פסיקות כותבות לסטאך נפרד משלחן.

תרגול

דוגמה נניח שבתוכנית יש מיליון פקודות. התוכנית מבוצעת פעמיים :

1. על מעבד רגיל עם חמשה שלבים ומהזור שעון של 2ns.
2. על מעבד super-pipeline עם שמונה שלבים ומהזור שעון של 1.3ns.

בנהה שאין סיכון בתוכנה, איזה מעבד יירץ את התוכנית מהר יותר?

המעבד הראשון יקח $2 \cdot 1 \cdot 10^6$ והשני יקח $1 \cdot 1.3 \cdot 10^6$ וסה"כ נקבל שהמעבד השני יותר מהיר עם speedup של 1.53 $\cdot \frac{2}{1.3} = 1.53$

דוגמה נתון הקוד הבא (באדום וכחול מסומנים הרגיסטרים שגורמים לסכנות)

- (1) ADD R1, R2, R3
- (2) ADD R4, R5, R6
- (3) SUB R7, R1, R1
- (4) LW R7, 0(R1)
- (5) ADD R7, R7, R5
- (6) SUB R10, R7, R12

- נרים אותו ב-issue-by-issue. נניח שיש forwarding (בתוך כל pipe, ולא ביןיהם!).
- את ה-branch תמיד יוכל לעשות כי הוא תמיד ב"ת כל עוד אין execute של 3 לא צריך לחכות כי הוא משתמש ב-forwarding של EX/MEM.
- בגלל ש-4 דורשת גם את R1, נעביר אותה ל-pipe-u כך שהיא-forwarding בתוך ה-pipe יעביר את הערך החדש של R1.
- פקודה 5 דורשת את פלט פקודה 4 וכן נעביר אותה ל-pipe-u כדי שתוכל להשתמש בה השם forwarding ב-MEM מפלט ה-MEM כשהוא יהיה זמין. נאלץ להוסיף stall כדי שהיא תהיה זמין.
- בדומה, את 6 גם נשים ב-pipe-u כך שיוכל להשתמש ב-forwarding פקודה 5 כשייה אפשר ואז ההרצתה היא כמו ב-pipeline רגיל עד לשימוש הרצתת כל הפקודות.

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

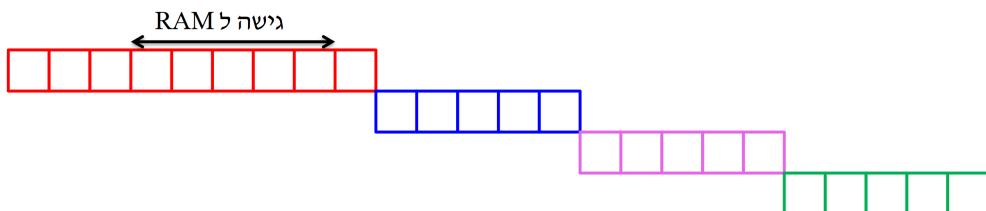
סה"כ נידרש ל-10 מחזורי שעון כדי להריץ את חמישת הפקודות.

דוגמה נדגים את הרצת הפקודות הבאות במעבדים שונים

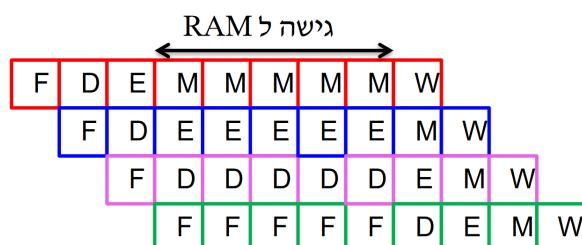
1. Lw \$t0, 4(\$t1)
2. Add \$t2, \$t3, \$t4
3. Add \$t5, \$t6, \$t2
4. Add \$t2, \$t4, \$t2

נניח שהיota miss בפקודה הראשונה. כל ריבוע מסמל מחזור שעון.

.1. מעבד Single Cycle :



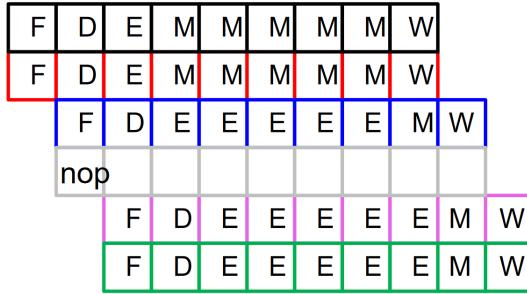
.2. מעבד stall : נבצע stall לכל השלבים עד שנתקבל תשובה (זה מעכבר את כל הפקודות בכל השלבים כי הן מחכות במצב צו או אחרת לתוצאות הפקודה הראשונה או פקודה שתלויה בה).



.3. מעבד 2-Way Superscalar : שתי הפקודות הראשונות ב"ת וילך נוכל להריץ אותן יחד. שתי הפקודות השניות גם ב"ת אבל stall על הזכרון תופס את ה-MEM לפקודות האחרונות ולילך הן ימתינו ב-Execute. נשים לב שיש כאן סכנה בין הפלט של ה-Execute לבין הפלט של 3 ו-4 וילך ב-pipe-forwarding עם Superscalar רק בתוך pipe-forwarding עם Superscalar ה-4 הינו צרכות להיות באוטו pipe כמו ה-3.



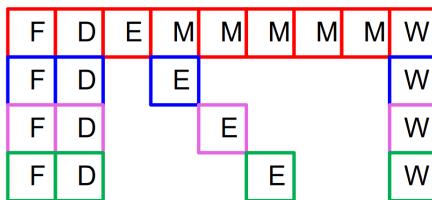
אם ה-w1 הוצמד לפקודה קודמת (שchorה), נאלץ להריץ הבא (כשיעור נניח שיש forwarding בין pipe-iים, בניגוד להנחה הרווחת ב מבחנים ותרגילים)



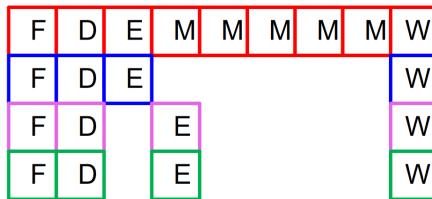
4. מעבד שיכול להריץ ב-single-issue, כלומר ניתן להריץ Execute של פקודה אחת בכל פעם, וכרגעיל Fetch/Commit לפי

הסדר ב-ROB. בזמן שיש stall על הזיכרון, שאר הפקודות יכולות להמשיך לזרוך כי אף אחת מהן לא תלויות בפלט הקריאה. שוב

נניח שיש forwarding בין ה-pipe-ים



אם היינו ב-dual-issue היינו מקבלים הרצה שנראית כך



דוגמה נתרגם את הקוד הבא (משמאלי) לאסמבלי (מימין)

```

struct node {
    int value;
    struct node *left, *right;
};

// a - points to the head of the tree
// v - the value we are counting

int *tree_scan(struct node *a, int v) {
    int count = 0;

    if (a == NULL) return count;
    if (a->value == v) count++;
    count += tree_scan(a->left, v);
    count += tree_scan(a->right, v);
    return count;
}

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

```

עתה נראה איך להריץ אותו ב-VLIW, כשלכל batch מכיל גישה אחת לזכרו, שתי פעולות אРИתמטיות ופעולות קפיצה אחת. הzbעים מפוזרים בין המקבצים (אין קשר בין שני צבעים זהים המופרדים בצבע שונה).

את beq חייבים להריץ בנפרד כי לא נרצה להריץ את שאר הפקודות אם אכן קופצים (אם יש קפיצה, עדין מרים את כל מה שיש ב-branch). את שתי הפקודות הבאות נריץ יחד אבל לא נטרף את ה-sw הנוסף כי אפשר גישה אחת לזכרו. בדומה ל-lw וה-sw הבא (6-1), רק של-7 נוכל להצמיד פעולה אРИתמטית ו-branch והוא ב"ת-ב-6 ולכן נריץ אותו ב-VLIW מוקדם מ-7!

את הקיבוצים וההפרדות בין גישות לזכרו וקופציות נבעו שוב ושוב. בנוסף, חלק מהפקודות שונות (לדוגמא שורה 4) כדי לשמר על נכונות במהלך הקיבוץ. מצורף משמאלי אותו התרגום כנ"ל כדי شيיה יותר נוח לבדוקו בשינוי הסדר של הפקודות.

# start of the function: \$a0=a, \$a1=v beq \$a0, \$0, end_null # if (a==NULL) addi \$sp, \$sp, -12 # it is allowed to defer the save after beq sw \$ra, 0(\$sp) sw \$so, 4(\$sp) sw \$si, 8(\$sp) lw \$t0, 0(\$a0) # \$t0 is a->value addi \$so, \$0, 0 # \$so is count, count = 0 bne \$t0, \$a1, not_equal # if (a->value != v) addi \$so, \$so, 1 # count++ lw \$si, 4(\$a0) # \$s1 = a->right lw \$a0, 8(\$a0) # \$a0 = a->left jal tree_scan # call tree_scan(a->left, v) add \$so, \$so, \$v0 # count += tree_scan(a->left, v) add \$so, \$s1, \$0 jal tree_scan # call tree_scan(a->right, v) add \$v0, \$so, \$v0 # setup return value: count += tree_scan(a->right, v) lw \$ra, 0(\$sp) lw \$so, 4(\$sp) lw \$si, 8(\$sp) addi \$sp, \$sp, 12 jr \$ra addi \$v0, \$0, 0 jr \$ra	1. tree_scan: 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. not_equal: 12. 13. 14. 15. 16. 17. 18. 19. 20. 21. 22. 23. end_null: 24.	# start of the function: \$a0=a, \$a1=v beq \$a0, \$0, end_null addi \$sp, \$sp, -12 sw \$ra, -12(\$sp) # the sw \$so, 4(\$sp) lw \$0, 0(\$a0) # ins! sw \$si, 8(\$sp) addi \$so, \$0, 0 # \$so bne \$t0, \$a1, not_equal addi \$so, \$so, 1 # col lw \$si, 4(\$a0) # \$s1 lw \$a0, 8(\$a0) # \$a0 jal tree_scan # call add \$so, \$so, \$v0 # col \$a0, \$s1, \$0 jal tree_scan # call add \$v0, \$so, \$v0 # se lw \$ra, 0(\$sp) lw \$so, 4(\$sp) lw \$si, 8(\$sp) addi \$sp, \$sp, 12 jr \$ra addi \$v0, \$0, 0 jr \$ra
---	---	--

שבוע | XIIII

הרצאה

פתרונותות שהציגו עד כה להאצת ביצועים בת' רד יחיד (VLIW, Loop Unrolling, Superscalar) לא מתיחסים בזמן ש策יך לחכות כדי לגשת לזכרו. גם אם 90% מהגישות הן hit למיטמון, גם זמן הגישה אליו דורשת כמה מחזורי שעון שבלי פתרון יודי נאלץ לעשות עליהם stall. זכרו.

דוגמה אם ידוע לנו שגישה למיטמון (ב-bit) לוקחת ארבעה מחזורי שעון, נוכל לבנות מעבד pipelined שגישה לזכרו אורכת ארבעה מחזוריים וכך נוכל לשמר על אותו throughput, זמן מחזורי קצר ו latency מינימלית יותר גרווע.

דוגמה נניח ש-50% מהפעולות הן אРИתמטיות, 30% גישות לזכרו ו-20% branch-ים. 10% מהגישות לזכרו הדאטה ו-2% הגישות לזכרו הפוקודות מפספסות עם penalty של 20 מחזוריים. ה-stall הממוצע לפוקודה הוא $1 = 30\% \cdot 20 + 2\% \cdot 20 + 10\% \cdot 20$. בטבלת ניתן לראות

ערכיהם טיפוסיים לשושה סוג מעבדים, וחישוב הזמן המתבקש על המתנה לזכרו בריצה עליהם

	Single-issue Pipelined	Superscalar Dual-issue	VLIW
Ideal CPI	1.1	0.7	0.5
CPI w/ cache misses	$1.1 + 1 = 2.1$	$0.7 + 1 = 1.7$	$0.5 + 1 = 1.5$
% מהזמן שמתבצע על המתנה לזכרון.	48%	59%	67%

כלומר המעבדים המהירים יותר ממחכים יותר זמן הריצה שלהם לזכרון.

הערה הפתרון שנראה לתלות בזיכרון הוא OOOE, אבל יש עוד פתרונות, לדוגמה פקודות prefetch שקוראות מהזיכרון למטרונו אבל לא חוסמות, שכן שימושות לקריאה מהזיכרון כשידוע לנו שנctrיך דאטה כלשהו בהמשך.

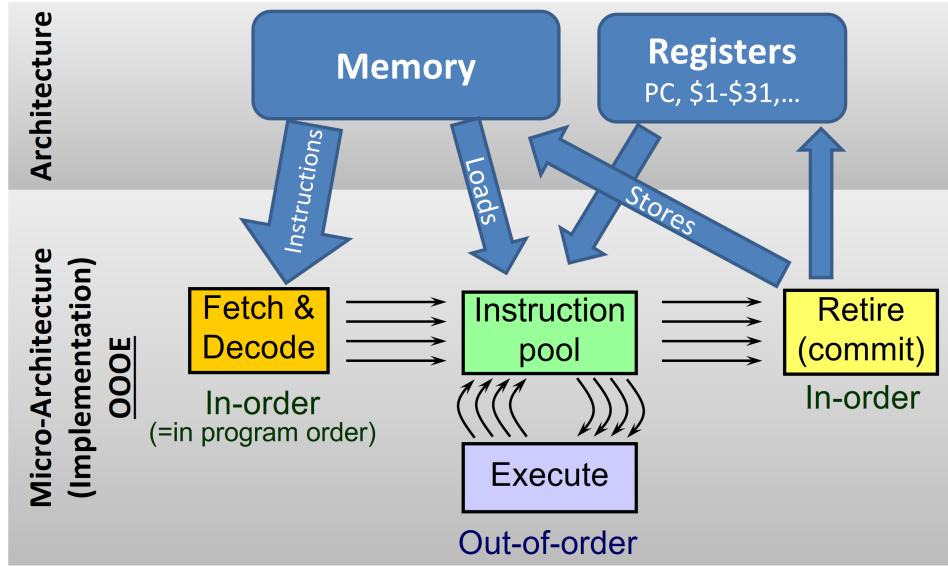
האלגוריתם של Tomosun

כדי לשפר ביצועים על אף העלות הגבוהה של גישה לזכרון, נctrיך לעקוּף פקודות שקוראות מהזיכרון ולהמשיך להריץ פקודות אחרות עד שנקבל תשובה. החסמים העיקריים במקובל בرمת הפקודה (ILP) הם התalianות בין פקודות שונות, וכן העבודה שמספר הרגיסטרים הנמור מגביל את יכולת הקומפיאילר לסדר מחדש את הפקודות כרצונו.

הפתרון לשתי הביעות האחרונות הוא OOOE בסגנון Tomosun (שהמציא את הראיון ב-1967), שביסדו לא משנה את הסמנטיקה של התוכנית המקורי, אבל מריצ פקודות בה לפי סדר זמינות הנתונים במקום סדר זמינות הפקודות (סדרתית, שזה המקורה הקלסי) על בסיס פקודות ב"ת אחת בשנייה".

הערה נבחן בין ארכיטקטורה (ISA), שהיא המשק שמעבד מציג למפתח, לבין המימוש מאחורי הקלעים של המשק הזה, שהוא המיקרו-אררכיטקטורה. WILW שינה את המיקרו-אררכיטקטורה וגם את הארכיטקטורה - ככלומר שבר תוכנות שאחרת היו יכולות לרווח באופן תקין ולהפוך. בקשר זה, הסמנטיקה היא התשובה לשאלת "מה הקוד עושה", וזה משתנה אם הארכיטקטורה משתנה. המטרה בתוכן OOOE היא לא לשנות את הארכיטקטורה (ולכן גם לא את הסמנטיקה של התוכנית), אבל כן את המיקרו-אררכיטקטורה, שהיא בלתי נראית למשתמש.

מימוש ה-OOOE יהיה בהשראת הסכמה הבאה ; Fetch/Decode, את ההריצה נמקבל על בסיס ה-Data Flow (תclf נראיה) ואת ה-WB (לרגיסטרים וזכרנו) נעשה שוב לפי הסדר.

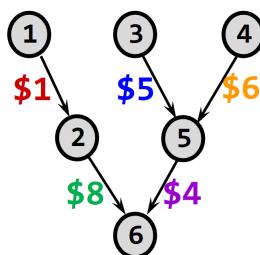


המעבד יבודק באופן תדריר אם יש רצף סדרתי של פקודות ב-pool שהוושלמו (שמתחילהות אחרי פקודה שכבר עשו לה) (Retire), ואם נמצא רצף כזה יעשה להם Retire לפי הסדר. בדומה, הוא יבודק תדריר האם ה-pool מלא ואם לא יקרה עוד פקודות מהזיכרון. במקרה, רכיב הביצוע יבודק שוב ושוב אילו פקודות ניתן לקדם ב-pipeline הביצוע וירץ אותן. כפי שניתנו לראות מספר החצים באירוע, מעבדי OOOE הם באופן טיפוסי גם Superscalar.

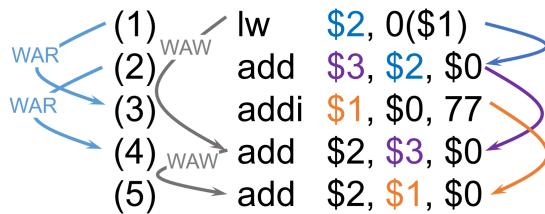
דוגמא נבייט בקוד הבא

- (1) lw \$1, 0(\$4)
- (2) add \$8, \$1, \$2
- (3) addi \$5, \$5, 1
- (4) sub \$6, \$6, \$3
- (5) add \$4, \$5, \$6
- (6) add \$9, \$8, \$4

נשים לב שאמנם הפקודות סודרו באופן מסוים מלכתחילה, הרצה בסדר אחר שלחן יכולה להציג ביצועים יותר טובים שלא דורשים המתנה כה אורך, וזאת על בסיס תלויות הנתונים שנitinן להסיק מהפקודות (ראו גוף)



דוגמא נבייט בקוד הבא



מיימן מסומנות סכנות אמיתיות (קלאסיוית, הופיו גם ב-In-Order) ומשמאלי סכנות חדשות שנוצרו כתוצאה מתלוויות מדוימות, ככלומר קשר בין פקודות שלא משפיע סמנטית על הנתונים אבל עלול לפגוע בחוקיות התוכנה בעת הריצה (לא בטוחה) ב-OOOE.

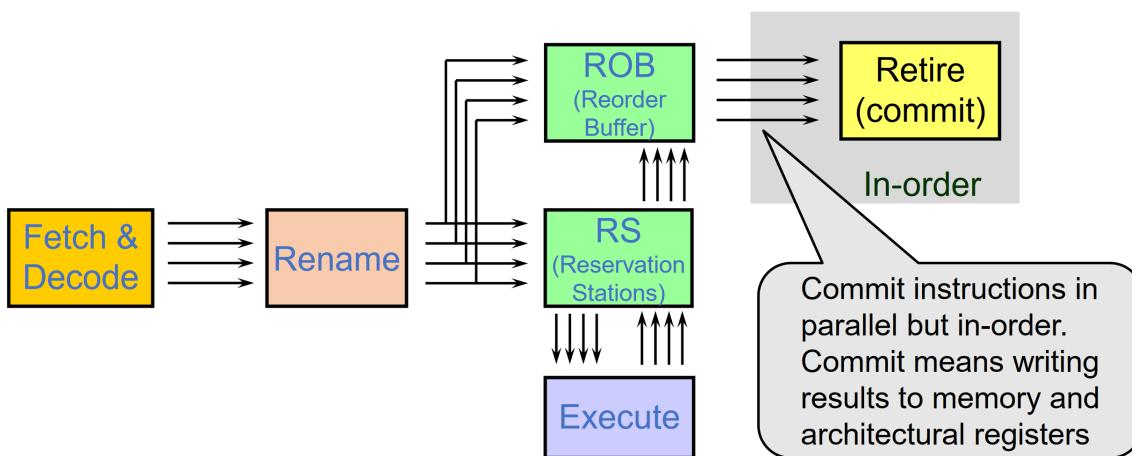
כפי שראינו בהרצתה, הפתרון ל-False Dependencies ; Register Renaming הוא ; נחזור על מה שראינו בתרגול בקורס. נשמר מייפוי של רגיסטרים ארכיטקטוריים לרגיסטרים פיזיים, ובכל פקודה נפה :

- נפה רגיסטרים ארכיטקטוניים שהם מקור (שימושים כאופרנדים) לרוגיסטרים פיזיים בהם נשמר המידע מלפני כן.
- נפה רגיסטרים ארכיטקטוניים שהם יעד (שימושים כיעד תוצאת הפעולה) לרוגיסטרים פיזיים יש לשמור את התוצאה, ונעדכן את המיפוי כך שפקודות שקוראות את הרוגיסטר הארכיטקטוני יופנו לרוגיסטר הפיזי הנכון.

שיטת ה-Register Renaming פותרת את כל ה-False Dependencies ומאפשרת יותר חופש תנוצה להריצה OOOE.

הערה במהלך Decode שמבצע לפיה הסדר ובמקביל לכמה פקודות נפה את הרוגיסטרים הפיזיים והארכיטקטוניים לפי סדר הפקודות המקורי.

בבית ב"מכונת OOOE קלאסי" (מעבד Tomosulu ביתר פירוט



אחרי ה-Decode, נבצע את ה-Renaming, כאמור לפי הסדר. משם, נקבע מקום לתוכנת הפקודה ב-ROB (מערך כניסה השמורות לתוכנות של פקודות, שבו דנו בתרגול). במקביל, נכתב למערך ה-RS את הפקודה והאופרנדים שהיא דורשת. אם כל האופרנדים כבר זמינים, הרצת הפקודה יכולה להתחיל. אחרת, ה-RS מאוזין ל-snooping שמכיל התוצאות ביןיהם ומאשר את התחלת ביצוע הפקודה ברגע שהאופרנדים זמינים.

רכיב הביצוע מכיל (אול) Functional Units כמה Shiccols למבצע פקודות במקביל (בסוגנו Superscalar). כשה-U-FU מסוימים לבצע פקודה RS-IC מבזבזת חזרה את התוצאה ל-ROB ויאפשר לפקודה חדשה להיכנס ל-RS. לסום, באותו האופן שתיארנו לעל, לבצע Retire לפקודות השלמנן, לפי הסדר כמוון.

דוגמה החשיבות של Retire לפי הסדר ניכרת בדוגמה די פשוטה: נניח שאנו מוצאים חלוקה ואחריה כמה פעולות נוספות שכבר השלמנו, ומווכנות לכתיבה לזכרו. אם החלוקה תהיה ב-0 במקרה, נחטוף גטוש Exception שיישתלט על הריצעה, ובמקרה כזה לא נרצה לכתוב תוצאה עתידית שיכל להיות שבכלל לא תרוץ.

פספוסי Branch Prediction

במעבדים עם pipeline יש רכיב חומרה שנקרא branch predictor, שאחראי על קביעה היוריסטית האם נקבע ב-branch או לא. במכונת OOOE, אם ה-branch predictor ניחש לא נכון האם נקבע נסמן את ה-branchmispredicted. בנוסף, כשההמעבד מזהה שפקובות Tomosulu שמסומנת ב-ROB, הוא יניקה (flush) את ה-branch, ומיפוי ה-RS, ה-branch Renaming ויתחיל מההתחלת באלאג' mispredicted flush. נשים לב שככל הפקודות דורשות flush כי יכול בהכרח צערות יותר מה-branch (כי Retire נעשה לפי מהכתבת המכונה בזכרנו הכתובות). איטרציה של המכונה בזכרנו הכתובות. נשים לב שככל הפקודות דורשות flush כי יכול בהכרח צערות יותר מה-branch (כי Retire נעשה לפי הסדר).

הערה בתוכנות טיפוסיות אחת ל-8 פקודות הן branch, ורוב מכירע שלןן כן מתבצעות בסופו של דבר (מספקה לולאה אחת גודלה שבכל איטרציה שלה חזק מהאהרונה קופצים חזרה להתחלה, וכל שאר ה-branchים מתגמדים ביחס אליה).

המחיר של branch misprediction הוא מאוד גבוהה מבחינת זמן ריצה.

דוגמה נניח שככל 8 פקודות יש branch בתוכנה שלנו, שרצה על מעבד flush 4-wide OOOE. במקרה של המעבד מתעכבר-5 מחזוריים (לינקויים הנ"ל ואთחול ה-pipeline). אם ה-branch predictor תמיד טועה אז כל 8 פקודות נעה stall של 5 מחזוריים כל 2 מחזוריים (שבמוצע יקראו 8 פקודות), וכך קיבל 27% ניצולות של המעבד בהרצת פקודות המכונות.

אם אנחנו צודקים בהסת' 50%, אחד לכל branch מנוחש לא נכון, כך שככל 4 מחזוריים יש stall של 5 מחזוריים, שזה 45% זמן עבודה מהותית.

אפילו בהסת' 90% של ניחושים נקבל רק 80% של זמן עבודה מהותית, כלומר נצטרך הסת' ניחוש מוצלחת הרבה יותר גבוהה.

דוגמה ה-branch predictor הפשט ביוטר יקבע אם מדובר בלולאה (קפיצה אחורה בקוד) ולא יקבע אם מדובר ב-if (קפיצה קטנה קדימה בקוד), אבל זה לא יספיק.

מדד שיעור לנו לקבוע את איקות ה-branch הוא predictor Misprediction-per-Instruction, כלומר היחס בין מספר הניחושים הטעילים לש"כ הפקודות.

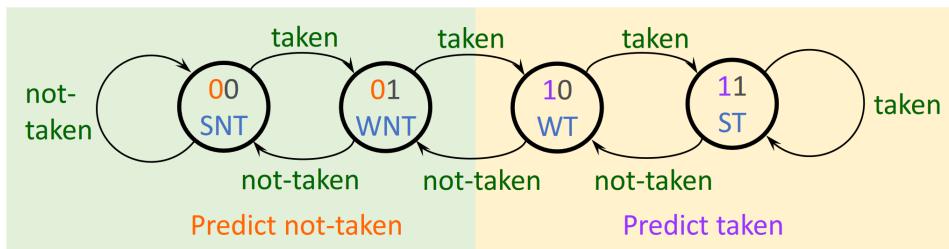
סוגי-Predictor

One-bit • נחזיק מערך ביטים עם כניסה למספר קבוע של branch-ים שמשודר באופן הבא: גדייר סט/איינדקס כמו שהגדכנו במתומו שمبוסס על הכתובת אליה פקודת branch-ים קופצת (שמזהה את ה-branch, אמם לא באופן ייחודי), וכך מסדר את ה-branch-ים

המחזקים במערך. בכל פעם שנגיעה לפקודת branch נקבע לפי הבית שופיע בכניסה המתאימה לסת שלה ונעדקן את המערך בהתאם למה שקרה בפועל (מימוש טריוויאלי של "זיכרון"). נשים לב שיכולת להיות התנשאות בין שני זיכרונות של קפיצה לכתובות שונות עם סט זהה (כמו שראינו במתומו).

החסרון בזיכרון בבייט 1 היא שאם אנחנו מרכיבים לולאה כמה פעמים (לדוגמא לולאה מקוננת) אז נטעה באיטרציה האחורונה שבנה (כפי לפניכן כל הזמן קפוץ) וגם באיטרציה הראשונה של החזרה הבאה של הלולאה, כי נחש מזכרן שלא נקבע, על אף שכן נקבע כי באיטרציות הראשונות כן נקבע.

- (2-bit Bimodal) : נשתמש בשני ביטים לזכרון, וכך נמנע את בעיית השגיאה הכפולה שראינו לעיל. נגיד מكونת מצבים עם ארבעה מצבים (כבאיור) כך שהחותם הדעת שלנו על כל branch מתחלקת ל"נלקח" ו"לא נלקח" בשתי רמות בטיחון שונות - "חזק" ו"חלש" ונחש בהתאם



ב-fetch נחש את הקפיצה בהתאם לערך הנוכחי של הספרן, וב-return נעדכן בהתאם לנוכנות הניחוש.

גם ל-predictor כזה לא מצליח לתפוס תבניות כמו 01010101 (שימושה ב-ifים שבבודקים זוגיות של אינדקס רץ).

הערה עד עכשיו לא התייחסנו לעובדה שכטיבה וקריאה לזכרון יכולה להיות לאותה הכתובת בלי שנדע זאת כי לשני רגיסטרים שונים יהיה את אותו ערך.

דוגמה נתון הקוד הבא

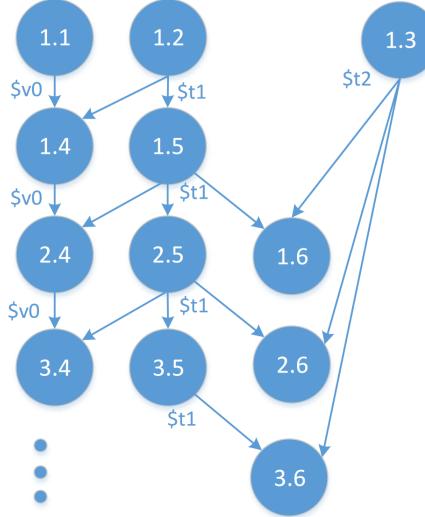
```

1.      add    $v0, $0, $0
2.      lw     $t1, 0($a0)
3.      lw     $t2, 4($a0)
4. loop: add   $v0, $v0, $t1
5.      addi  $t1, $t1, 1
6.      bne   $t1, $t2, loop

```

בנחלה שגיאות לזכרון עלולות לחתות זמן רב ומספר האיטרציות גדול, איזה רוחב מעבד 000 ירי את הקוד בצורה מהירה? יש להעזר בגרף התלויות של התוכנית.

גרף התלויות של התוכנית נראה כך (האינדקס השמאלי הוא מספר האיטרציה והימני הוא הפקודה, החץ מכיל את הרגיסטר שגורם לתלויות)



כאשר התבנית המסדרת נוצרה כי פקודות 4 ו-5 תלויות בפקודות 4 ו-5 של האיטרציה הקודמת (או 1 ו-2 אם אין צו) בהתאם ו-6 תמיד תלולה ב-3.

עתה, אם נריץ עם רוחב פס 1 או 2 נקבל תוצאות פחות אופטימליות מ-3 (כי יש 3 פקודות ב"ת שנייתן להריצ' בכל איטרציה), ואילו אם נריץ ב-4 ומעלה בהכרח נדרש **l-stall**-ים כי התוצאה לא תהיה מוכנה, لكن התשובה היא 3.

תרגול

הערה פקודה קריית וקטור עם stride לא נמצאת באף מחלקת פקודות שראינו עד כה (יש בה גם immediate וגם שני רגיסטרים שם אופרנדים).

דוגמה ברשותנו מחשב עם פעולות וקטוריות על וקטור בגודל 64 מיליון, בתדריות 8Khz. נתון הקוד הבא

```
void transpose (int** A, int** B) {
    for (int i = 0; i < 64; i++) {
        for (int j = 0; j < 64; j++)
            B[i][j] = A[j][i];
    }
}
```

- תרגמו את הקוד לאסmbli שתוכנן בפעולות וקטוריות כאמור.

להלן התרגומים, עם הערות בצד שմסבירות מה אנחנו עושים. פקודת `saws` מקבלת רגיסטר וקטורי לתוצאה, כתובות בסיס ו-stride וקוראות מהזכרנו מכתובת הבסיס עם קפיצות של stride עד כמספר הבטים בוקטור.

```

        addi $s0 , $zero , 64          // נתחל ספירה לולאה
Loop: lws $vec1 , 0($a0) , 256    // נטען עמודה במטריצה
        addi $a0 , $a0 , 4           // נקבע עמודה אחת ימינה
        sv $vec1 , 0($a1)           // נשמר את העמודה כשורה
        addi $a1 , $a1 , 256         // נרד שורה במטריצה השניה
        addi $s0, $s0, -1            // נקדם את הספירה
        bne $s0, $zero , Loop       // קפיצה

```

את העמודה נצטרק לקרוא באמצעות stride major row ואילו את השורה אין בעיה לקרוא כרגיל. אם לא הייתה לנו אפשרות להשתמש ב-`aws`, היינו נאלצים להשתמש בפעולות לא וקטוריות לצורך העמודות ולבן גם לשורות, ובמקרה זה עדיף כבר להריץ על מעבד רגיל שיש לו זמן מהזר הרבה יותר גובה מעבד וקטורי, כי אנחנו לא משתמשים ביכולות שלו בכלל.

- כמה שניות יידרשו לביצוע התוכנית, בהנחה שיש את כל המרכיבים?

כל איטרציה דורשת 8 פקודות והאחרונה 6 (ולכן סה"כ נידרש ל- $518 = 8 \cdot 8 + 6$ מהזרי שעון שעון ייעיל יותר (ביחס כפלי לגודל הוקטור) מעבד רגיל. הנתון על מהירות השעון גורר שמחזור שעון אורך $\frac{1}{8000}$ של שנייה, כלומר סה"כ $64 \approx \frac{518}{8}$ מילישניות.

- איזה שיפור אם בכלל, יהיה להריצה במעבד כזה על פני מעבד רגיל שבו ניתן לבצע 16 איטרציות-loop?
במעבד רגיל יש 64^2 איטרציות, ולאחר פרימה נקבל 256 איטרציות. מספר מהזרי השעון באיטרציה פרומה הוא $2 + 16 \cdot 4 = 66$ פעולות חזות אינדקסים, 2 פעולות גישה לזכרון שיחזרו על עצמןשוב ושוב, ו-2 פקודות לטיפול בלולאה, בהתאם מה-stall בקפיצה. סה"כ נקבל $66 \cdot 256 = 16896$.

דוגמה נביט בקוד הבא שסוכם ריקורסיבית אברים של רשימה מקוشرת

```

int sum_list(node* T) {
    if (T == null)
        return 0;
    else
        return T->data + sum_list(T->next);
}

```

בגלל אופן ההקצתה ב-C, אין לנו ידע על מיקום האובייקטים ביחס לאחרים. אין שום דרך למש את הפ' הזו באמצעות פעולות וקטוריות, כי אם ראשית נקרא את כל ה-`data` לוקטור או נסכם אותו כבר עדיף לסקום בזמן ההעתקה, וכל דרך אחרת משתמשת בוקטוריים לצורך פעולות לא וקטוריות.

הערה מעבדים וקטוריים הם יותר איטיים ממעבדים רגילים, והרבה מאוד דברים לא יעילים יותר איתם. מעבדים גרפיים הם דוגמה טובה לשימוש יעיל במעבדים וקטוריים.

ס. גו.