# Lab5


מגישים:

חן אילון - 201617032

גל קשי - 204572861


## שאלה 1

בHarvard architecture למעבד יש 2 זכרונות sram בגודל המקורי של sram, אחד להוראות: srami ואחד למידע: sramd.

יתרונות:

- מאפשר "לחסוך" בתלויות קריאה וכתיבה לזיכרון - מפני שכתיבה וקריאה של data מופרדת מקריאה של ההוראה מהזיכרון האחר. כלומר, חוסך זמן ריצה.
- יותר בטיחותי מבחינת ניהול זיכרון: כתיבת מידע לא יכולה לדרוס את התוכנית.

חסרונות:

- משתמש בפי 2 זיכרון מהארכיטקטורה הקודמת
- צריך לתמוך בכתיבות וקריאות ל2 זכרונות שונים - ללא דריסות זיכרון

זוהי החלטה טובה במקרה שלנו כיוון שהמטרה הייתה ביצועים טובים יותר ולא חומרה זולה יותר.

## שאלה 2

### - Structural Hazard

- פעולת כתיבה לזיכרון ואחריה קריאה מהזיכרון - נשים לב כי פעולת LD נגשת לזיכרון בשלב exec0 ופעולת ST נגשת בשלב exec1. כלומר, במימוש pipeline, אם תקרא פעולת ST ומיד אחריה פעולת LD, שתיהן ירצו להשתמש במשאב הזיכרון באותו זמן:

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| ST | FETCH0 | FETCH1 | DEC0 | DEC1 | EXEC0 | EXEC1 | |
| LD | | FETCH0 | FETCH1 | DEC0 | DEC1 | EXEC0 | EXEC1 |

ריצה מתוקנת תראה כך:

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| ST | FETCH0 | FETCH1 | DEC0 | DEC1 | EXEC0 | EXEC1 | | |
| LD | | FETCH0 | FETCH1 | DEC0 | DEC1 | ** | EXEC0 | EXEC1 |

### - Data Hazard

- RAW - כאשר קיימות 2 פעולות ברצף שהראשונה מכניסה מידע לרגיסטר מסויים (פקודת אריתמטיקה כלשהי או פקודת LD), והשניה משתמשת במידע שנכתב לרגיסטר המסויים.

לדוגמה -

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| ADD 2 3 4 | FETCH0 | FETCH1 | DEC0 | DEC1 | EXEC0 | EXEC1 | |
| ADD 5 6 2 | | FETCH0 | FETCH1 | DEC0 | DEC1 | EXEC0 | EXEC1 |

ריצה מתוקנת תראה כך:

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| ADD 2 3 4 | FETCH0 | FETCH1 | DEC0 | DEC1 | EXEC0 | EXEC1 | | | |
| ADD 5 6 2 | | FETCH0 | FETCH1 | DEC0 | ** | ** | DEC1 | EXEC0 | EXEC1 |

דוגמה נוספת -

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| LD 2 0 3 | FETCH0 | FETCH1 | DEC0 | DEC1 | EXEC0 | EXEC1 | | |
| ADD 5 0 0 | | FETCH0 | FETCH1 | DEC0 | DEC1 | EXEC0 | EXEC1 | |
| ST 0 2 4 | | | FETCH0 | FETCH1 | DEC0 | DEC1 | EXEC0 | EXEC1 |

ריצה מתוקנת תראה כך:

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| LD 2 0 3 | FETCH0 | FETCH1 | DEC0 | DEC1 | EXEC0 | EXEC1 | | | |
| ADD 5 0 0 | | FETCH0 | FETCH1 | DEC0 | DEC1 | EXEC0 | EXEC1 | | |
| ST 0 2 4 | | | FETCH0 | FETCH1 | DEC0 | ** | DEC1 | EXEC0 | EXEC1 |

- WAR - מצב בו הפעולה הראשונה, שקוראת מרגיסטר מסויים, מתעכבת. והפעולה אחריה, שכותבת לאותו רגיסטר, מסיימת קודם. ואז יכול לקרות מצב בו הרגיסטר שיקרא בפעולה הראשונה יכיל את המידע שנכתב אליו בפקודה השניה, ולא את המידע המקורי.

לדוגמה -

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| ADD 2 3 4 | FETCH0 | FETCH1 | DEC0 | ** | ** | ** | ** | DEC1 | EXEC0 | EXEC1 |
| ADD 4 5 6 | | FETCH0 | FETCH1 | DEC0 | DEC1 | EXEC0 | EXEC1 | | | |

ריצה מתוקנת תראה כך:

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD 2 3 4 | FETCH0 | FETCH1 | DEC0 | ** | ** | ** | ** | DEC1 | EXEC0 | EXEC1 | |
| ADD 4 5 6 | | FETCH0 | FETCH1 | ** | ** | ** | ** | DEC0 | DEC1 | EXEC0 | EXEC1 |

- WAW - מצב בו הפעולה הראשונה, שכותבת לרגיסטר מסויים, מתעכבת. והפעולה אחריה, שכותבת לאותו רגיסטר, מסיימת קודם. ואז יכול לקרות מצב בו הערך שישמר ברגיסטר אחרי 2 הפעולות יהיה של המידע בפעולה הראשונה, ולא בפעולה השניה, כפי שהיינו מצפים.

לדוגמה -

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| ADD 2 3 4 | FETCH0 | FETCH1 | DEC0 | DEC1 | EXEC0 | ** | EXEC1 |
| ADD 2 5 6 | | FETCH0 | FETCH1 | DEC0 | DEC1 | EXEC0 | EXEC1 |

ריצה מתוקנת תראה כך:

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| ADD 2 3 4 | FETCH0 | FETCH1 | DEC0 | DEC1 | EXEC0 | ** | EXEC1 | |
| ADD 2 5 6 | | FETCH0 | FETCH1 | DEC0 | DEC1 | ** | EXEC0 | EXEC1 |

<u>שאלה 3</u>

ההתמודדות עם קפיצות במימוש שלנו תתבצע באופן הבא:

- בשלב DEC0 אנו יודעים שמדובר בברנץ (באמצעות משוון) וחוזים אם צריך לקפוץ.
- עבור JMP DIRECT (16-19):
  - נחזיק חזאי 2bit של ברנצים. בכל פעם שתתבצע קפיצה נוסיף לו 1 (אלא אם הוא כבר במקסימום), ובכל פעם שלא תתקיים קפיצה נחסיר ממנו 1 (אלא אם הוא כבר אפס).
    כאשר נצטרך לחזות קפיצה, נסתכל על ערך המונה - אם הוא 2 ומעלה נקפוץ, אם הוא 1 ומטה, לא נקפוץ.
  - בשלב DEC1 אנו מיישמים את החיזוי. אם החיזוי הוא not-taken, ממשיכים בביצוע 2 הפעולות שנמשכו. אם החיזוי הוא taken, עושים ל2 הפעולות הללו flush ומביאים את הפעולה בכתובת שבimmediat.
  - בסוף שלב EXEC0 יודעים אם הברנץ באמת היה צריך להלקח. לכן, בשלב EXEC1 אם צדקנו בחיזוי - נמשיך בביצוע. אם טעינו בחיזוי - נעשה flush להוראות שהבאנו (3 הוראות במקרה שחזינו taken, 5 הוראות במקרה שחזינו not-taken), ונקפוץ להוראה בpc המתאים.
- עבור JMP INDIRECT (20):
  - לא נשתמש בחזאי, מפני שהקפיצה תמיד מתרחשת. כן נצטרך להשתמש בflush כאשר אנו מבינים שההוראה היא אכן קפיצה.

נסתכל על מספר דוגמאות עבורן החיזוי טועה. עבור התוכנית הבאה:

| ADDRESS | Instruction |
|---|---|
| 0 | JNE 0 2 3 8 |
| 1 | ADD 2 0 0 |
| 2 | ADD 3 0 0 |
| 3 | ADD 4 0 0 |
| 4 | ADD 5 0 0 |
| 5 | ADD 6 0 0 |
| 6 | |
| 7 | |
| 8 | SUB 2 0 0 |
| 9 | SUB 3 0 0 |
| 10 | SUB 4 0 0 |
| 11 | SUB 5 0 0 |
| 12 | SUB 6 0 0 |

עבור JMP DIRECT עם חיזוי taken, שלא נלקח:

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| JNE 0 2 3 8 | FETCH0 | FETCH1 | DEC0 | DEC1 | EXEC0 | EXEC1 | | | | | |
| ADD 2 0 0 | | FETCH0 | FETCH1 | | | | | | | | |
| ADD 3 0 0 | | | FETCH0 | | | | | | | | |
| SUB 2 0 0 | | | | FETCH0 | FETCH1 | | | | | | |
| SUB 3 0 0 | | | | | FETCH0 | | | | | | |
| ADD 2 0 0 | | | | | | FETCH0 | FETCH1 | DEC0 | DEC1 | EXEC0 | EXEC1 |

עבור JMP DIRECT עם חיזוי not-taken, שנלקח:

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| JNE 0 2 3 8 | FETCH0 | FETCH1 | DEC0 | DEC1 | EXEC0 | EXEC1 | | | | | |
| ADD 2 0 0 | | FETCH0 | FETCH1 | DEC0 | DEC1 | | | | | | |
| ADD 3 0 0 | | | FETCH0 | FETCH1 | DEC0 | | | | | | |
| ADD 4 0 0 | | | | FETCH0 | FETCH1 | | | | | | |
| ADD 5 0 0 | | | | | FETCH0 | | | | | | |
| SUB 2 0 0 | | | | | | FETCH0 | FETCH1 | DEC0 | DEC1 | EXEC0 | EXEC1 |

## שאלה 4

מצ"ב.

## שאלה 5

הלוגים מצורפים.

ההבדלים היחידים הם בין קבצי הinst_trace בספירה שונה של הpc (בhex או מול dec).

## שאלה 6

### חלק 1

- Mult -
  - מספר ההוראות בתוכנית - 32
  - מס' מחזורים להשלמה בLab2 ובLab4 - 138
  - מס' מחזורים להשלמה בLab5 - 53
  - חישובי Speedup: $\frac{138}{53} = 2.6$

- Fibo -
  - מספר ההוראות בתוכנית - 16
  - מס' מחזורים להשלמה בLab2 ובLab4 - 4758
  - מס' מחזורים להשלמה בLab5 - 1296
  - חישובי Speedup: $\frac{4758}{1296} = 3.67$

- Example -

- מספר ההוראות בתוכנית - 12
- מס' מחזורים להשלמה בLab2 ובLab4- 366
- מס' מחזורים להשלמה בLab5 - 142
- חישובי Speedup: $\frac{366}{142} = 2.57$

*הערה - מצאנו באג בתוכנית האסמבלי mul מהמעבדות הקודמות. תיקנו אותו והצאנו לוגים שוב.

<u>חלק 2 -</u>

נחשב את הCPI פר תוכנית:

- Mult -
  - מספר ההוראות בתוכנית - 32
  - CPI בLab2 ובLab4 - $\frac{138}{32} = 4.31$
  - CPI בLab5 - $\frac{53}{32} = 1.65$

- Fibo -
  - מספר ההוראות בתוכנית - 16
  - CPI בLab2 ובLab4 - $\frac{4758}{16} = 297.37$
  - CPI בLab5 - $\frac{1296}{16} = 18$

- Example -
  - מספר ההוראות בתוכנית - 12
  - CPI בLab2 ובLab4 - $\frac{366}{12} = 30.5$
  - CPI בLab5 - $\frac{142}{12} = 11.83$

למרות שקל לראות שהCPI החדש טוב יותר באופן משמעותי מהקודם, נשים לב כי הCPI החדש אינו 1 כמו שההנהלה ציפתה.
זה מתקבל בעקבות:
- stalls - תלויות של מבנה ושל מידע.
- חיזויים לא נכונים של קפיצות. מקרים כאלו יגרמו לflush של מספר הוראות, ולמעשה תעכב את התוכנית לפי מספר הflushים.

<u>חלק 3</u>

דרכים לשיפור הIPC בדור מתקדם יותר-

- שימוש בbranch prediction יותר מתוחכם (Two Level). לדוג' -
- שימוש בBTB ובהיסטוריית שליחות פר-ברנץ - ייתכן כי לכל branch בקוד קיימת התנהגות אופיינית לו. ועל כן יהיה יעיל יותר לאפיין התנהגות עבור קפיצה מסויימת בקוד.
- שמירת יותר ביטים בהיסטורייה (אנו השתמשנו ב2bit predictor). בתוכניות מסובכות יותר ייתכן שיהיה יעיל יותר לשמור היסטוריה יותר גדולה, עם יותר ביטים.

- שימוש בForwarding לטיפול ב-Data Hazard:
- RAW - ניתן לעשות שליחה מהירה של המידע בEXEC0, (לפי הwrite back בEXEC1) ישירות לDEC1 ולחסוך stall אחד. כלומר, במקרה של 2 stallים, העיכוב ירד לstall אחד, ובמקרה של stall אחד, לא יהיה אף stall.
- טיפול בstalls של תלויות מבנה:
- נרצה שיהיו לנו יותר רכיבים שכותבים לזיכרון. ואז לא נצטרך לחכות להתפנות משאב שכזה בפעולות קריאה וכתיבה.
- שיפור נוסף יכול להיות הכפלת החומרה והרצת תוכניות ,או חלקים שלהן, באופן מקבילי. למשל ע"י אלגוריתם טומסולו.

<u>שאלה 7</u>

על מנת להשתמש ביחידת ה-DMA הוספנו 2 הוראות חדשות ל-IS:

1. DMA (30) – מעתיק R[src0] מילים מהמקור החל מכתובת R[src1] ליעד החל מכתובת R[dst].
2. POL (31) – קובע את הערך של R[dst] להיות 1 אם קיימת פעולת DMA פעילה. אחרת, קובעת את הערך להיות 0. באחריות המפתח לבדוק שלא קיימת פעולת DMA פעילה לפני קריאה נוספת ל-DMA, גישה לזכרון שעליו רצה הפעולה, או סיום התוכנית.

נשים לב כי לפקודות לעיל יכולות להיות גם תלויות מבניות עם הזיכרון, וגם תלויות של דאטא (קוראים src0, src1, dst).

קוד הבדיקה ממעבדה 2 מספיק טוב גם בבדיקה הזו, כיוון שנבדקות בו תלויות:

<u>קוד הבדיקה:</u>

```
ADD, 2, 1, 0, 200   // 0: R2 = 200
ADD, 3, 1, 0, 500   // 1: R3 = 500
DMA, 3, 1, 2, 100   // 2: Copy MEM[R2:R2+100] to MEM[R3:R3+100]
ADD, 2, 1, 0, 30    // 3: R2 = 30
ADD, 3, 1, 0, 1     // 4: R3 = 1
ADD, 4, 1, 0, 8     // 5: R4 = 8
JEQ, 0, 3, 4, 14    // 6: PC = 14 if R3 == R4
LD,  5, 0, 2, 0     // 7: R5 = MEM[R2]
ADD, 2, 2, 1, 1     // 8: R2 = R2 + 1
LD,  6, 0, 2, 0     // 9: R6 = MEM[R2]
ADD, 6, 6, 5, 0     // 10: R6 = R6 + R5
ST,  0, 6, 2, 0     // 11: MEM[R2] = R6
ADD, 3, 3, 1, 1     // 12: R3 = R3 + 1
JEQ, 0, 0, 0, 6     // 13: PC = 6
POL, 2, 0, 0, 0     // 14: R2 = 1 if DMA is running, else 0
JNE, 0, 2, 0, 14    // 15: PC = 14 if R2 != 0
ADD, 2, 1, 0, 200   // 16: R2 = 200
ADD, 3, 1, 0, 500   // 17: R3 = 500
ADD, 4, 1, 0, 600   // 18: R4 = 600
JEQ, 0, 3, 4, 27    // 19: PC = 27 if R3 == R4
LD,  5, 0, 2, 0     // 20: R5 = MEM[R2]
LD,  6, 0, 3, 0     // 21: R6 = MEM[R3]
ADD, 2, 2, 1, 1     // 22: R2 = R2 + 1
ADD, 3, 3, 1, 1     // 23: R3 = R3 + 1
JEQ, 0, 5, 6, 19    // 24: PC = 19 if R5 == R6
ADD, 2, 1, 0, 0     // 25: R2 = 0
```

```
HLT, 0, 0, 0, 0      // 26: HALT
ADD, 2, 1, 0, 1      // 27: R2 = 1
HLT, 0, 0, 0, 0      // 28: HALT
```

## הסבר על כיסוי הבדיקה -

בפקודות הראשונות ניתן לראות תלויות דאטא:

```
ADD, 2, 1, 0, 200    // 0: R2 = 200
ADD, 3, 1, 0, 500    // 1: R3 = 500
DMA, 3, 1, 2, 100    // 2: Copy MEM[R2:R2+100] to MEM[R3:R3+100]
```

בפקודה DMA אנו קוראים את התוכן של R2 ושל R3 ולכן נתקל בData Hazard של RAW עבור R2 ושל WAW עבור R3.

בהמשך ניתן לראות שיש Structural Hazard:

```
DMA, 3, 1, 2, 100    // 2: Copy MEM[R2:R2+100] to MEM[R3:R3+100]
ADD, 2, 1, 0, 30     // 3: R2 = 30
ADD, 3, 1, 0, 1      // 4: R3 = 1
ADD, 4, 1, 0, 8      // 5: R4 = 8
JEQ, 0, 3, 4, 14     // 6: PC = 14 if R3 == R4
LD,  5, 0, 2, 0      // 7: R5 = MEM[R2]
```

במהלך שהפקודה DMA מעתיקה 100 תאים, מתרחשות פעולות זיכרון שיוצרות תלויות מבניות.

טיפול - לשאר פקודות הזיכרון יש עדיפות על הDMA, והוא יפנה להן מקום לרוץ וימשיך לרוץ כשיסיימו.

בהמשך ניתן לראות תלות דאטא נוספת:

```
POL, 2, 0, 0, 0      // 14: R2 = 1 if DMA is running, else 0
JNE, 0, 2, 0, 14     // 15: PC = 14 if R2 != 0
```

בפקודה JNE אנו צריכים להשתמש במידע שנכתב לR2 בPOL. ולכן יש לנו תלות מבנית מסוג RAW.

## הסבר שהטסט עובד עפ"י הלוגים -

הקוד מבוסס על תוכנית הדוגמה, מכיוון שהיא מכילה מספר קריאות לזכרון. ניתן לראות בקבצי הפלט כי התוכנית ביצעה את הפעולה המקורית שלה (חישוב סכום) במקביל להעתקת הבלוק, וכמו כן גם ההשוואה בסוף התוכנית הצליחה.