**Lab5**

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:



ריצה מתוקנת תראה כך:



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 -

C:\Users\DELL1\AppData\Local\Microsoft\Windows\INetCache\Content.Word\WhatsApp Image 2019-01-15 at 11.07.19.jpeg

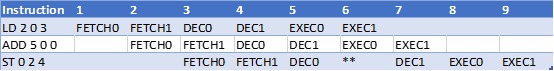
ריצה מתוקנת תראה כך:

C:\Users\DELL1\AppData\Local\Microsoft\Windows\INetCache\Content.Word\WhatsApp Image 2019-01-15 at 11.15.26.jpeg

דוגמה נוספת -



ריצה מתוקנת תראה כך:



* 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 -



ריצה מתוקנת תראה כך:

C:\Users\DELL1\AppData\Local\Microsoft\Windows\INetCache\Content.Word\WhatsApp Image 2019-01-15 at 11.33.07.jpeg

* 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 -



ריצה מתוקנת תראה כך:



שאלה 3

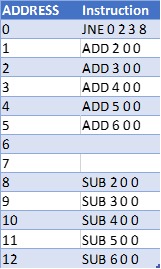
ההתמודדות עם קפיצות במימוש שלנו תתבצע באופן הבא:

* 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).

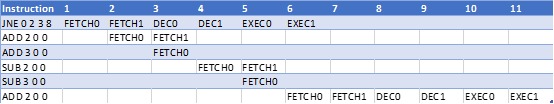
כאשר נצטרך לחזות קפיצה, נסתכל על ערך המונה - אם הוא 2 ומעלה נקפוץ, אם הוא 1 ומטה, לא נקפוץ.

* 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):
* נשתמש בחזאי סטטי not-taken.
* בגלל שרק בסוף שלב DEC1 אנו יודעים לאיפה אולי צריך לקפוץ. החיסכון שלנחש taken הוא של הוראה אחת. ואילו ההפסד במקרה שטעינו בחיזוי של taken יוצא יקר (4 הוראות מתבזבזות). לכן, החלטנו לבחור בחזאי סטטי ולא בחזאי 2 ביט.

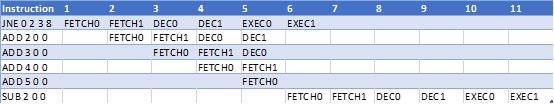
נסתכל על מספר דוגמאות עבורן החיזוי טועה. עבור התוכנית הבאה:



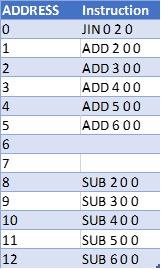
עבור JMP DIRECT עם חיזוי taken, שלא נלקח:



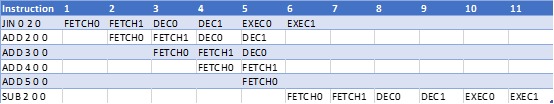
עבור JMP DIRECT עם חיזוי not-taken, שנלקח:



עבור התוכנית הבאה:



דוגמה לJMP INDIRECT עם חיזוי not-taken שנלקח:



שאלה 4

שאלה 5

שאלה 6

שאלה 7