

## הסבר הרצאות בקורס "מבנה מחשבים ספרתיים" 234267 כפי שנלמד בפקולטה למדעי המחשב, טכניון

מסמך זה נכתב באופן אישי כיזמה פרטית של סטודנט בפקולטה למדעי המחשב בטכניון בזמן שלמד את הקורס, ומהווה סיכום אישי של ההרצאות. הוא נכתב באופן עצמי ולא למטרת רווח, במטרה לעזור לעצמו ולסטודנטים אחרים להבין את החומר כפי שנלמד ב"מס' בסMASTER חורף 2016". המרצים האחרים באותו סMASTER הם ד"ר ליהוא רפפורט ומר עדי יועז.

הסבר הרצאות מובוסים על השקפים של ד"ר ליהוא רפפורט ומר עדי יועז.  
ההסבר של תרגול מס' 11 מובוס על התרגול שלימד מר פרנק סלה.

הסבר הרצאות נכתבו במטרה לתאר את ההסבירים בדיקן כפי שנלמדו בכתיבה - אך בפועל הם נערכו, שונים, הורחבו ו/או צומצמו במידה רבה, לפרשנות אישית ו/או שגوية, ע"י אלעד שטייגמן. הם תוצאה של מאיץ רב ומתרשם (!) לתמלו, להבין, ולהסביר בפיירוט את החומר הרלוונטי לקורס כפי שהועבר בסMASTER חורף 2016. יחד עם זאת, קשה עד בלתי אפשרי לעשות זאת במהלך סMASTER אחד.

אלעד שטייגמן, שכتب וערך, אינו נוטל כל אחריות על שימוש במסמך זה, בפרט על מידת הצלחה שלמידה מקורות אלה תספק, וממליץ בחום ללמידה באופן פעיל בקורס לפי הנחיות המפורשות ע"י סגל הקורס באותו סMASTER. אין התחייבות בגין **Rate Hit** של מושג זה **Misprediction** של החומר הנלמד בקורס!

המעורבים ביצירת מסמך זה ניסו בכל דרך לכבד את היוצרים, המורים ומוסד הטכניון שבזכותם סיכומים אלה יכולים להיקتب. אם בכלל זאת נעשתה פגיעה כלשהו אני להודיע על כך בהקדם.

כל זכויות על חומר رسمي שמופיע בכתב היד או מוצג בכתב היד תקפות וכל שימוש בקובץ זה נדרש לקיים אותו.

### הפרת הוראות אלה מהוות עבירה פלילית ועולה אזרחות!

אלעד שטייגמן  
[shtelad@yahoo.com](mailto:shtelad@yahoo.com)

תודה

לד"ר ליהוא רפפורט ולמר עדי יוזע

על שקי הרצאות

ועל שלימדו.

בצלחה

## תוכן עניינים

הרצאות (mboss על שקפים של ד"ר ליהוא רפפורט ומר עדי ירוז)

5 \_\_\_\_\_ הרצאה 1: הקדמה

19 \_\_\_\_\_ הרצאה 2: מעבד מצונר - Pipeline

32 \_\_\_\_\_ הרצאה 3: זיכרון מטמון - Cache

42 \_\_\_\_\_ הרצאה 4: ארגון זיכרון המטמון - Cache Organization

56 \_\_\_\_\_ הרצאה 5: שיתופי זיכרון המטמון, מערכת מחשב, זיכרון DRAM

70 \_\_\_\_\_ הרצאה 6: חיזוי קפיצות - Branch Prediction

87 \_\_\_\_\_ הרצאה 7: זיכרון וירטואלי - Virtual Memory

102 \_\_\_\_\_ הרצאה 8: זיכרון וירטואלי - Virtual Memory (המשך)

הרצאה 9\*: ביצוע שלא-על-פי הסדר - Out Of Order Execution

117 \_\_\_\_\_ הרצאה 10: ביצוע שלא-על-פי הסדר (המשך) - Out Of Order Execution 2

137 \_\_\_\_\_ הרצאה 11: מיקרו-ארQUITקטורה של מעבדי אינטל - Intel uArch

152 \_\_\_\_\_ הרצאה 12: הספק – Power

168 \_\_\_\_\_ הרצאה 13: ריבוי חוטים – Multithreading

נספח:

תרגול מס' 11 - ביצוע שלא על-פי הסדר של פקודות זיכרון - Memory Out Of Order

\*לא זמין בגרסה זו של הסיכום

# Computer Structure

## 234267

### הרצאה מס' 1: הקדמה

#### הקדמה

מטרים: עדי יועז וליהוא רפפורט. למעשה שניהם עושים את הקורס זהה כבר 12 שנה בפקולטה. באביב הקודם נתנו אותו גם בהנדסת חשמל. הקורס מבוסס על דברים שהמטרים עושים בחיי היום-יום: הם הארכיטקטנים הראשיים של ה-Core באינטל. בקורס מציגים את המבנה של ה-Core וקצת בסביבתו. כידוע מישראל יוצאים ה-Cores של אינטל - זה מרכז תכנון מאוד מוביל והם האחראים על קו המוצרים זהה, כך שאתה לומדים בקורס דברים אמיתיים שנמצאים בחזית הטכנולוגיה. המתרגלים בקורס גם הם בקבוצה של המטרים באינטל והם מאוד מסונכרים איתם.

הציון מורכב מרבעה תרגילי בית תקפים, ניתן להגיש אותן בזוגות ומשקלם 20% מהציון. הבחינה מהוות 80% מהציון הסופי. כל החומר שמופיע במחזור נלמד בהרצאות ובתרגולים.

יש לקורס אתר אליו יעלטו את כל הרצאותך שנייתן לפני הרצאה לעבור על השקפים ולראות מה אתה למד במסגרת השיעור. מומלץ מאוד להציג להרצאות ולשמור את ההסברים כי בהרצאה מבינים את הקשר של הנושאים השונים יותר טוב.

### Class Focus

#### ◆ CPU

- ❖ Introduction: performance, instruction set (RISC vs. CISC)
- ❖ Pipeline, hazards
- ❖ Branch prediction
- ❖ Out-of-order execution

#### ◆ Memory Hierarchy

- ❖ Cache and cache coherency
- ❖ Main memory
- ❖ Virtual Memory

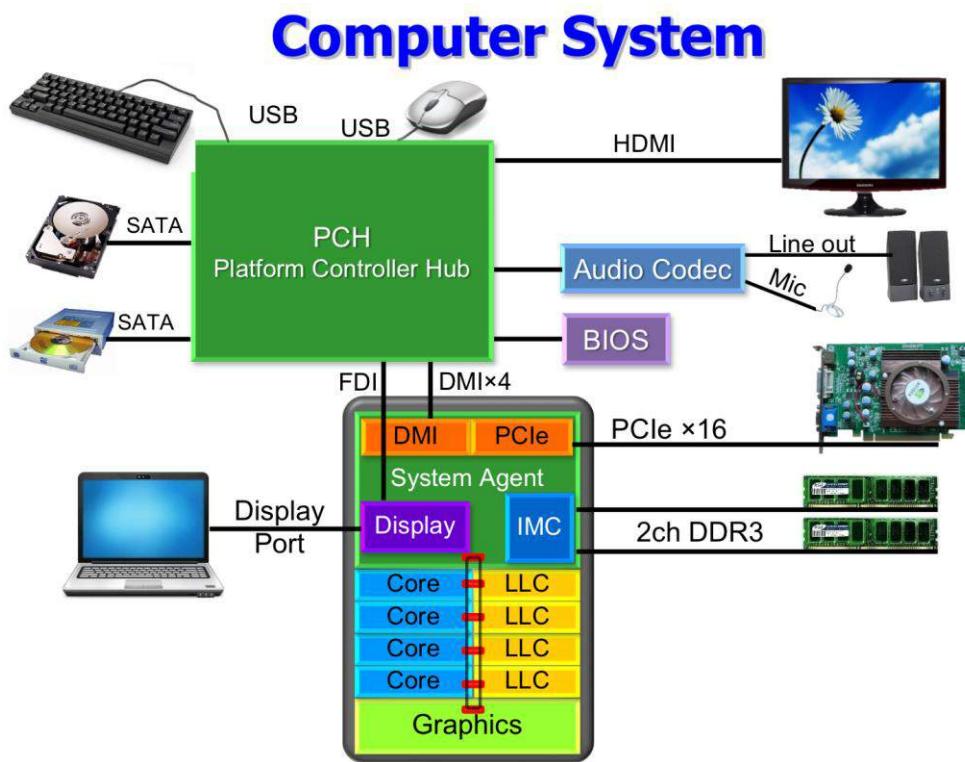
#### ◆ More topics

- ❖ Multi-threading
- ❖ System
- ❖ Power considerations

בקורס זה בעיקר נכנס לARBים של המעבד (CPU - Central Processing Unit). השיעור הראשון הוא שיעור מבוא בו נסביר איך מודדים ביצועים ומשווים בין מעבדים שונים וכן מה ההבדל בין מעבדי RISC למעבדי CISC. בשיעור הבא נדבר על צינור (Pipeline). אחרי למד על branch prediction: אלגוריתמים שמופיעים בכל מעבד מודרני ועוזרים להתקדם לביצועים מקסימליים. בהמשך הקורס נדבר על הזיכרון החיצוני/פנימי, איך הוא מורכב,

מה עושים כדי להאריך גישות לזכרון, מה זה זיכרון מטמון ואיך הוא בנווי. בנוסף נרחיב מעבר ל-**Core** ונדבר על המערכת כולה - אורך נראית מערכת מחשב,لوح אם, ועודון בשיקולי הספק (בניגוד לשיקולי ביצועים בלבד).

## מערכת מחשב



נתחיל עם תרשימים של המבנה הכללי של מערכת מחשב. בהמשך הקורס תהיה הרצאה شاملת על השקף הזה אז אל תנסו להבין את כלו בינוויים - המטרה כרגע היא לקבל רקע על מערכת מחשב לפני שונכנים לדון בלביה עצמה. בשקף רואים מודל של מערכת שבכללה ארבעה ליבות (המסגרת המרכזית למטה). מערכת כזו מתאימה למחשב נייד או שולחני (PC) אך לא לשרת (Server). המערכת מורכבת בעיקר סביב צ'יפ אחד העיקרי (יש שני צ'יפים על הלוח

אם, מודל זה נקרא **Two Chip Solution** (Two Chip Solution). הצ'יפ המרכזי עשו מפיסט סיליקון ועליה:

- ליבות: במקרה שלנו יש ארבע ליבות (היום בכל מחשב מודרני יש שתי ליבות לפחות, יש-Calala עם שישה. בעולם הסרברים יכולים להציג גם למש' הרבה יותר גודלים למשל 16 או 32 ליבות אבל כאמור כאן רואים צ'יפ-מרכזי של PC). הבהרה: כאותם "מעבד" מתכוונים לצ'יפ הראשי כולם, לא רק לライブות, אם כי נלמד יותר על הליבות עצמן מאשר על כל חלק אחר במערכת.

• על פיסת הסיליקון נמצא גם **First Level Cache** (FLC) שזה זיכרון-מטמון ("קاش") פרטן לכל ליבה. בהמשך נלמד על התעבורות של מידע בין הקаш לライブות - אזור המעבר נקרא **Ring**.

- מעבד **graphics** שנמצא על הצ'יפ (**GPU**) מסומן **Display** (מeosmon) - פתרון זה מתאים לכ-80% משתמשי המחשב בימינו.
- בקר זיכרון (**DRAM**) : מלבד הצ'יפ הראשי יש במחשב זיכרון ראשי (**DRAM**). זהו זיכרון חיצוני לצ'יפ המركזי. בקר הזיכרון, שבימינו גם נמצא על הצ'יפ הראשי, אחראי על הפניה ל-**DRAM** והבאת מידע אליו תוך המעבדים.
- על הצ'יפ יש באס של חיבור pci-express שנועד לחבר של יחידות חיצונית מהירות. דוג' לכך היה כרטיסי גראפי חיצוני - אם משתמש רוצה לשפר את הביצועים הגרפיים של המחשב מעבר לאלה של הcartesis המובנה שבצ'יפ המركזי.
- נוטן לנו אפשרות להתחבר למסך.

- כמו כן על לוח-האם יש צ'יפ נוסף שנקרא PCH (Platform Controller Hub) והוא מהווה את הממשק לשאר הhardware הקיימים יותר איטיים. אפשר להתחבר דרכו למסך חיצוני או למקרון, להתקני SATA כמו דיסק, למקלדת ועכבר וכו'..

כאמור בקורס רוב הדינונים יהיו על המבנה הפנימי של הקור וועל איך הוא עובד עם כל שאר המערכת. תריה הרצאה אחת שעוסקת בכל הצ'יפ המרכזי אבל בעיקר נדון בלביה עצמה.

## ארQUITקטורה ומיקרו-ארQUITקטורה

נדבר קצת על ההבדל בין ארQUITקטורה למיקרו-ארכ':

# Architecture & Microarchitecture

### ◆ **Architecture**

#### **The processor features seen by the "user"**

- ❖ Instruction set, addressing modes, data width, ...

### ◆ **Micro-architecture**

#### **The internal implementation of a processor**

- ❖ Caches size and structure, number of execution units, ...

### ◆ **Processors with different μArch can support the same architecture**

ארQUITקטורה היא כל מה שגלו למשתמש/קומפיילר שכותב תוכנית. הדוג' הנפוצה ביותר היא ה-instruction set, אוצר הפקודות של המעבד. דוג' נוספת היא מס' הרגיסטרים ושמותיהם.-arcl' לדוגמה הn x86 של אינטל וארcl' נוספות שמנוהלות ע"י חברת ARM.

ניתן לבנות ליבות שימושו-arcl' כלשהו בצורות שונות - למשל את x86 גם Intel וגם AMD מ mmapות וכל אחת מוכרת מעבד שתואם(arcl' זו. המימוש של-arcl' נקרא המיקרו-ארQUITקטורה: כל אחת מהחברות מ mmapת בפנים את האלגוריתמים השונים בצורה קצרה שונה על מנת שהמעבד יוכל לבצע את מה שהוגדר בתור אוצר הפקודות של הארcl'. למשל: בקורס נלמד על branch prediction, מנגן שמספר את ביצועי המעבד ע"י חיזוי לאן פקדת הסתעפות תלך (האם תקופץ או לא ואם כן - לאן תקופוץ) ומתייחס להביא פקדות לביצוע בהתאם לחיזוי. AMD וAINTEL יכולים להשתמש בשני אלג' שונים על מנת לבחור את החיזוי - זהו פרט של מיקרו-ארQUITקטורה. לעומת זאת כנסחאות בתכנית שכתובה באוצר הפקודות של x86 או התוצאה של התכנית חייבת להיות אותה על כל מעבד - אם כי כל מיקרו-ארcl' תשפר את התכנית הנ"ל באופן אחר. כמו לגביו הארcl' של ARM: יש כ-12 חברות שונות שמשמשות את הארcl' שלהם: Samsung, Qualcomm, ... כל אחת בונה את המעבד עם האלגוריתמים שלה - בסוף התוצאה של התכנית חייבת להיות אותו דבר על כל מיקרו-ארcl' שמשמשת הארcl' זהה כי תוצאת התכנית נובעת מהגדרת הארcl' - למשל מה המעבד ימצא בזיכרון אחורי פקדות Load / Store - אלה ציריכם בסוף להיות אותו דבר אבל מבפנים אפשר למשש את זהה באיזה אלג' וטريقים שונים וכל חברה ממששת את זהה כראות עיניה.

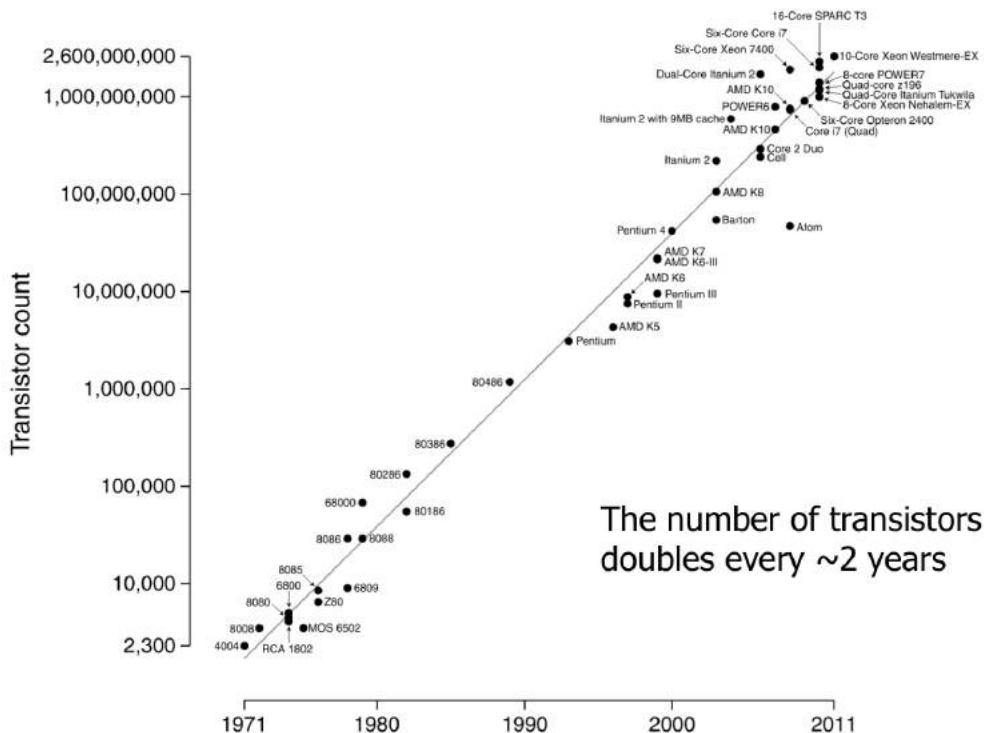
## תאמות לאחר וחוק מור

### ◆ Compatibility

- ❖ A new processor can run existing software
- ❖ The processor  *$\mu$ Arch* is new, but it supports the *Architecture* of past generations, possibly adding to it

תאמות, compatibility, היא היכולת להריץ תכנית ישנה על מעבד חדש בלי צורך בקמפול מחדש של התכנית. אם הצהרנו על קומpatibilitות, אז ניתן לחת את הבינהי היישן, להריץ אותו על המעבד החדש והתכנית תרוץ גם על המעבד החדש - בדר"כ יותר מהר. כשאנו מתחננים את הדור הבא של מעבד כלשהו מותר לנו להרחיב את הארכ', למשל להוסיף פקודות חדשות שלא היו בארכ' הישנה, כך שמי שירצה לкопל את האפליקציה שלו מחדש ישמש בפקודות החדשות ואולי יקבל האצה יותר גודלה (התכנית תתבצע יותר מהר), אבל יחד עם זאת חייבים לתמוך גם בתכנית המקורי שקומפלה לדור הקודם.

## Moore's Law



חוק מור הוא עקרון מנהה להמוני פיתוחים טכנולוגיים והוא מדבר על משהו שנקרא *process technology*: כמו שאנו יודעים כל מעבד שנחננו נדבר עליו מבוסס על סיליקון. הסיליקון נותן לנו אפשרות למשתתף בתיקן שנתקן טרנזיסטור שמסוגל לטעון/לפרוק וב歆ם בו משתמשים כדי לבנות את כל המבנה של השיא המעבד. הטרנזיסטורים הנ"ל נהיים קטנים יותר ככל שהזמן מתקדם ולפי חוק מור, התפתחות הטכנולוגיה מאפשרת גידול הטרנזיסטור לפחות במחצית בכל שנתיים. כל קפיצה כזו נקראת *process technology*, כלומר על אותה פיסת סיליקון בכל הסיליקון. התועלת של ההקטנה הזאת שונית לנצל את המקום על מנת לשפר את הביצועים, כלומר עם השיפורות הטכנולוגיות הבלתי, יש במעבד יותר מקום לדברים אחרים שתורמים לביצועים. המטרה שלנו, בתור ארכיטקטים של חומרה, היא לחת תכנית קיימת ולגרום לה להתבצע (להסתהים) יותר מהר.

למשל במעבד יש יחידות ביצוע וכן זיכרונות מטמון - ניתן לנצל את המקום שמרוחחים כדי להגדיל את מס' היחידות או להגדיל את הזיכרון. דוג' נוספת היא שבמעבדים יש predictors branch שחויזים כיוון של קפיצה וגורמים תכניות לרוץ מהר יותר - גם הם תופסים חלק ניכר משטח המעבד. כל האמור לעיל נעשה במגבלות הספק נתון: לכל מערכת יש יכולת קירור/פייזר-הספק מסוימת ווגם זה דבר מגביל, שמשתף עם טכנולוגיית התהילה, שנדרב עליו בפיירות לקראת סוף הקורס. בשקף רואים שאכן חוק מור מתקיים עד לאחרונה כל שתניות ניתן היה להכנס פי 2 טרנזיסטורים. היום הטכנולוגיה בשוק היא 30nm - זהו האורך של השער החשמלי (gauge).

## מושגי יסוד

### ◆ CPI – Cycles Per Instruction

- ❖ Average #cycles per Instruction (in a given program)

$$\text{CPI} = \frac{\text{\#cycles required to execute the program}}{\text{IC}}$$

- ❖ IPC (= 1/CPI) : Instructions per cycles

מדד חשוב נקרא CPI (cycles per instruction) והוא אומר כמה מחזורי-שעון המעבד צריך כדי לבצע פקודה כלשהי. למשל במעבד כלשהו יתכן שפקודת add אורך שעון אחד, פקודת load אוורכת ארבעה מחזורי שעון, branch מחזורי יחיד ופקודת כפל לוקחת שלושה מחזוריים. לעיתים רוצים לחשב את ה- CPI הממוצע של תכנית, ככלומר כמה פקודות בממוצע דרשה כל פקודה בתכנית. ה- CPI הממוצע של תכנית כלשהו הוא המנה בין מס' המחזוריים שנדרשו כדי לבצע את כל התוכנית לבין מס' הפקודות בתכנית (IC - Instruction Count).

מדד נוסף שימושו למדידת ביצועים בדומה ל-IPC הוא הרופכי לו, מס' פקודות שנעשות בכל מחזור שעון. בימינו מעבד מודרני מבצע בערך 2-2.5 פקודות בכל מחזור שעון - נלמד כיצד בהמשך הקורס.

דוגמה: נניח שבתכנית 100 פקודות והיא הتبוצעה ב-200 מחזוריים - אז ה- CPI = 200/100 = 2 cycles per instruction, וכן IPC = 1/2 Instruction per cycle (CPI), או לחלופין שבכל שני מחזורי שעון מסתיימת פקודה (IPC). בהמשך נבין מדוע לעיתים קרובות מתרחשים עיכובים בהשלמת פקודה.

הבהרה: CPI של פקודה/תכנית זה פרט הקשור למיקרו-ארQUITטורה - לימוש של המעבד.

בהתנחת ה- CPI לכל סוג פקודה והכמות של סוגי הפקודות השונים בתכנית אפשר לחשב את מס' המחזוריים הכלול לביצוע התכנית: עבור כל סוג פקודות :

$$\text{Cycles} = \text{average\_CPI} * \text{total\_IC} = \Sigma(\text{CPI}(i) * \text{IC}(i))$$

בפועל מדד הביצועים משתמשים בו להשוואה בין מעבדים הוא זמן-מעבד (cpu-time) : כמה זמן לוקח למעבד כלשהו לבצע תוכנית. הנוסחה לזמן המעבד היא:

$$\text{CPU Time} = \text{IC} * \text{CPI} * \text{ClockCycle} = \text{IC} * \text{CPI} * (1/\text{frequency})$$

המכפלה בין CPI (מספר הפקודות) ולתדר המעבד (זמן המחוור) אומරת כמה זמן ייקח לבצע את התוכנית. כשרוצים לשוק מחשב מפורטים את התדריות שלו אבל בפועל תדריות ריא רק רכיב אחד שלא משקי באופן מלא את הביצועים: אם מעבד כלשהו רץ בתדרות  $2\text{GHz}$  ומעבד אחר רץ ב- $\text{CPI}=4-4\text{GHz}$  ועבורו  $\text{CPI}=2$  אז בהתאם לנוסחה לזמן שלהם לביצוע תוכנית כלשהו יהיה זהה - لكن תדר המעבד אינו מספיק כדי להבין את הביצועים במלואם.

#### ◆ CPU Time - time required to execute a program

$$\text{CPU Time} = \text{IC} \times \text{CPI} \times \text{clock cycle}$$

#### ◆ Our goal: minimize CPU Time

- ❖ Minimize clock cycle: more GHz (process, circuit, uArch)
- ❖ Minimize CPI: uArch (e.g.: more execution units)
- ❖ Minimize IC: architecture (e.g.: AVX™)

arceritkutim אנחנו יכולים לשפר כל אחת מהרכיבים של נוסחת זמן המעבד (העלאת תדר/הורדת CPI/CPI/הורדת כמות הפקודות הנדרשת) ובכך לשפר את הזמן הקשור לביצוע תוכנית, למשל ככל שנעלה את התדריות (נויריד את זמן המחוור) נקבל זמן ביצוע קצר יותר לתכניות כלומר ביצועים יותר טובים. העבודה שלנו היא לטפל בשלושת הגורמים הנ"ל כדי לסייע את ה CPU Time. כמובן שבפועל יש מגבלות לכך לאפשרות מודד מסוימת, למשל שימושים תדר צורכים יותר הספק. לכן אי אפשר להעלות רכיב כלשהו בנוסחה כרצונו אלא יש balancing act שיש לבצע בין שלושתם כדי לקבל את המעבד הטוב ביותר.

כאמור ניתן לשפר את זמן הביצוע של תוכנית, CPU Time, ע"י שיפור של אחד משלושת הרכיבים:

- זמן המחוור/התדר מושפע מה process technology (הקטנת הגודל הפיזי של הטרנזיסטור מאפשרת להתקן לרזץ ב מהירות יותר גדולה, כי זמן הטעינה/פריקה של הקבלים ב- $\text{CIP}$  יותר קצר). התדר משתף גם עם שיפור בעיצוב ה-*circuit* של המעבד, וגם מיקרו-ארכ' יכולה להשפיע על התדר - בהמשך הקורס נראה כיצד (בגדול) Pipeline מאפשר להויריד את זמן המחוור).

- השפעה לטובה על הביצועים ע"י שיפור (הגדלת) ה-*CPI* - את זה נלמד בהמשך הקורס בהרחבה וזה בעיקר שייר למיקרו ארכ' - למשל הקטנת מס' המחוורים הממוצע שמתווסף כתוצאה מחיזוקים שגויים של קפיצות

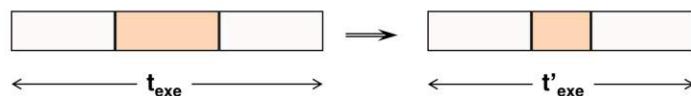
- הקטנת IC (Instruction Count), מס' הפקודות בתוכנית - זאת ע"י המצאת פקודות חדשות (פיתוח הארכ'), למשל שבפקודה אחת ניתן היה לבצע הרבה פעולות שדרשו מס' פקודות בדורותיים שונים של המעבד. לשם כך למשל הומצאה ב- $\text{x86}$  טכנולוגיה של פקודות וקטוריות שנקראת AVX למשל במקום ארבע פעולות חיבור מבצעת חיבור וקטורי בפעולה אחת. הערה חשובה: הקטנת IC על-ידי פקודות מורכבות יכולת להעלות את ה-*CPI* הממוצע בתוכנית, כי הפקודות מורכבות יותר ולכן דורשות יותר מוחזקים כדי לבצע אותן. יש לשקל האם מתתקבל שיפור כתוצאה מהרחבת אוצר הפקודות.

#### • חוק אמדל

חוק אmdl מאפשר לנו ליחס כמה האצת-כוללת (Speedup) נקבל בתוכנית ע"י האצת חלק מסוים מהתוכנית (למשל האצת הזמן המוקדש לביצוע פעולות כפל בתוכנית):

## Amdahl's Law

Suppose enhancement E accelerates a fraction F of the task by a factor S, and the remainder of the task is unaffected, then:



$$t'_{\text{exe}} = t_{\text{exe}} \times \left( (1 - F) + \frac{F}{S} \right)$$

$$\text{Speedup}_{\text{overall}} = \frac{t_{\text{exe}}}{t'_{\text{exe}}} = \frac{1}{(1 - F) + \frac{F}{S}}$$

נגידר את ה  $\text{Speedup}$  כ מנת הזמן המקורי הנדרש לביצוע התוכנית בזמן החדש (נשים לב ש  $1 > \text{Speedup} \geq 1$  משמעו שיפור בזמן ריצת התוכנית). את הזמן החדש הדורש ניתן לחשב ע"י סכימה של החלק משך הזמן עם הזמן שלא-שופר:

$$t'_{\text{exe}} = t_{\text{exe}} * [ (1 - F) + F/S ]$$

דוג': אם עשרית זמן הריצה של תוכנית כלשהי ביצענו פעולות כפל, ומהרנו את פעולות הכפל פי 2, אז 0.9 מהזמן המקורי שהתוכנית דרצה לא הואז ועשרה מהזמן המקורי פי 10 - لكن הזמן החדש מהווה 95% מהזמן המקורי - כלומר speedup הוא  $1.0 - 0.95 = 0.05$ . שיפור קטן מאד של פחות מ-10%. דוג' ז' של חוק אmdl מראה שגם אם נשפר מאוד את הביצועים של חלק קטן בתוכנית (בדוג' - פי 10) אז השיפור הכללי בביצועים לא יהיה כל כך גדול כי מראש זה לא היה החלק הארי של התוכנית ולכן שיפור שלו לא יהיה מאוד משמעותי. מסקנה עיקרית מחוק אmdl: האץ את השכיח: התמקד בשיפור הדברים בתוכנית שקוראים הרבה פעמים. זה מה שלוקח את עיקר הזמן וכן שיפור של זה יהיה ממשוני יותר.

### גישות למדדי ביצועים

כיצדعروשים השוואת ביצועים של מעבדים שונים:

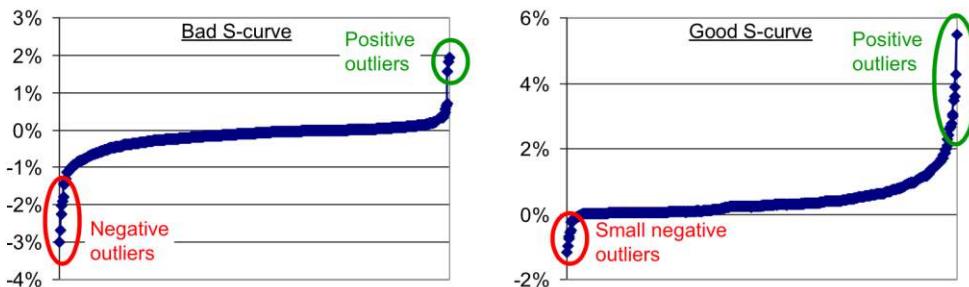
- מדדי peak performance - השוואת לפי הביצועים האידיאליים שמהמעבד יכול לספק - לא ההשוואה הכי טוביה/שימושית, כי בדר"כ בתוכנית אמתית לא ריאלי לצפות שמהמעבד יגיע לpeak performance, אבל משתמשים בה בפרסומות, בספרים ועוד. היא יכולה לתת אינדיקציה מסוימת אם כי פחות מהימנה מ CPU Time: דוג' למדד peak performance היא ה-**MIPS** (Million Instruction Per Second): כמה פקודות המעבד יכול לעשות בשניה, או למשל GFLOPS: מס' פעולות FP המקסימלי שנitin' לעשות בשניה. מדדים אלה נתונים מושג לגבי פס הרוחב שנייתן להגיע אליו תחת הנהזה שהמעבד זמין ואינו עיקובים "מחוץ למעבד", אבל הם לא משקפים מה יהיה ביצועי המעבד בעומס (workload) אמתית (תוכנית אמתית/אוסף תוכניות אמתית). ב-**workload** אמתית בדר"כ לא ניתן להביא את המעבד לביצועים אידיאליים, למשל בגלל תלויות בין פקודות או המתנה ל-IO

בזמן שمبرאים מידע מהדיסק ופקטורים נוספים שעולמים להגביל אותו - והם אכן מתרחשים לעיתים קרובות - למד עליהם במהלך הקורס. לכן תמיד עדיף למדוד ביצועים לפי תכנית כלשהי, עומס אמת.

- מדדי ביצועים אמתיים - אפליקציות אמתיות שמריצים על מעבדים ומודדים את זמן הביצוע שלהם עליהם.
- יש בנצח'מרקם סינטטיים שמדוימים עומס.
- המדדים המדויקים ביותר הם מדדים מפורטים ומקובלים בתעשייה שנקראים SPEC INT, SPEC FP, SPEC TPC ועוד. בהשוואה אמתית של מעבדים משתמשים בSPEC benchmarks אלה. רוב היצרנים מפרסמים את התוצאות של ביצועי המעבדים שלהם לפי מד זה.

## Evaluating Performance of future CPUs

- ◆ **Use a performance simulator to evaluate the performance of a new feature / algorithm**
  - ❖ Models the uarch to a great detail
  - ❖ Run 100's of representative applications
- ◆ **Produce the performance s-curve**
  - ❖ Sort the applications according to the IPC increase
  - ❖ Baseline (0) is the processor without the new feature



בפועל, כשרוצים לבנות אלג' חדש לא יש מממשים אותו בסיליקון אלא קודם מודדים את האלג' בסימולטור שכתובה בשפה עילית, כמו C) שמודל את העבודה המעבד. לאחר מכן משנים את המודל של המעבד בסימולטור ואז מרים כל מיני Trace-ים (סימולציות של עומס, נבחנות בקפידה ע"י הארכיטקטים) ורואים בכמה השיפור כל בנצ'ט. כאשר מתחננים לשפר מעבד, למשל כמו שיפור באלג' חיזוי הקפיצה שהזכרנו, אז לכל שיפור יש עומסים (תכניות) בהם הוא משפר את הביצועים וכן עומסים בהם הוא מרע את הביצועים. משתמשים בעוקמה שנקראת S-Curve שמתארת את השפעת האלג' החדש על כל אחד מהנצח'מרקם שמייצגים את מגוון העומסים הנפוץ - כך מעריכים האם השיפור כדאי או לא. אם, גוף אופייני של שיפור שנחשב טוב (צד ימין בשקף) הוא לפחות אם לרוב התכניות השיפור כמעט ולא شيئا, למעט הוא שיפור בכ-10% ובסה"כ הוסיף בממוצע אחוז אחד לביצועים. כמו שראים בשקף תמיד יש כמה מדדים שמספרדים כתוצאה שימוש באלג' חדש - צריך לוודא שביצועיהם משתפרים כתוצאה מאלג' אחרים שאנו חנו ממשים במעבד החדש באותו "דור". אם לחופין הכנסנו אלג' בו ניתן לראות שגובה המהדרים נופלים - נגד 50% מהמרקם נשארו אותו דבר, 20% השיפור ו-30% מהמרקם דפקנו את הביצועים, אז זה אלג' לא מצליח וכנראה שלא יקבלו אותו שיפור של המעבד. משמאלו רואים דוג' ל curve-s לא טובה שכנהה ולא יתקבל האלג' ממנו היא נגזרת. באופן כללי צריך לשים לב שה s-curve לא טובה שכנהה ולא יתקבל האלג' ממנה היא נגזרת.

הברחה: כמשמעותם מעבד מכניים הרבה שיפורים ויש לוודא שגם אם שינוי אחד מוריד את הביצועים של כמה מדדים, עדין בסה"כ לא יגעו הביצועים הכלולים של אף מדד (trace). לא סביר שבמעבד משופר תהיה איזושהי אפליקציה שנפגעת מכל השינויים כך שהביצועים שלו לא משתפרים.

## шиוקלים בתכנון ISA

כאמור ארכ' מגדרה את ה-ISA - רשימת פקודות אסמלר של המכונה שהקומפיילר/מתקנת יכול לעבוד איתן בתרו אבני בניין כדי לכתוב את התכניות שלו. את השימוש של הארכ', המיקרו-ארכ', כל חברה יכולה לעשות בצורה שונה (כל עוד התוצאה של התכנית היא אותה תוצאה - אותה ISA), למשל ע"י אלג' שונים של caching, branch prediction ועוד .. הארכ' הומצאה לפני כ-30 שנה והיא מתפתחת כל הזמן אבל כאמור חייבים לתמוך גם בbackward compatibility.

נלמד על כל מיני שיקולים של תכנון אוצר-פקודות:

### ◆ Reduce the IC to reduce execution time

- ❖ E.g., a single vector instruction performs the work of multiple scalar instructions

- שיקול אחד בתכנון ISA יכול להיות להקטין את ה-Instruction Count, שכי ראיינו משפיע על זמן הריצה של התכנית: כשאנחנו מתקנים את רשימת פקודות המכונה נהיה מעוניינים שתכניות יוכלו להיכתב במס' פקודות קטן (למשל ב-150 פקודות במקום ב-200 בדרך הקודם של הארכ') - אוסף הפקודות של הארכ' יקבע איזה וכמה פקודות צריך כדי למשתמש תכנית כלשהו. נניח, למשל, שהתכנית בשפה-העלית מבצעת מיון - באיזה פקודות מכונה משתמש כך? אולי יש טעם לספק פקודות ייעודיות למילון, למשל פקודה swap בין שני אורי זיכרון, או למשל נרצה להשתמש בפקודה של חישוב וקטורי שמדרשה ביצוע כמה פקודות סקלריות במקביל. בacr אנחנו מפשטים את העבודה של הקומפיילר - אם כי יתכן ואנחנו מסובכים את הארכיטקטורה ומספקים פקודות מורכבות בעלות latency גבוהה, ככלمر שדורשות הרבה מחזורי שעון (ה-CPI שלן גבוהה). צריך לחתה זאת בחשבון בעת ההחלה איזה פקודות להוסיף לארכ'.

### ◆ Simple instructions ⇒ simpler HW implementation

- ❖ Higher frequency, lower power, lower cost

- מוטיבציה נוספת, שלעתים מנוגדת לראשוונה, היא פישוט: יתכן ונרצה לעשות את אוסף הפקודות לא מורכב- מדי, כך שתכנית תוכל להתבצע מתוך אבני בניין פשוטות יחסית. זה עוזר למשל לבנות את המעבד כך שהוא ינצל שטח יותר קטן על הצ'יפ, מה שבobil להספק יותר קטן, ושאיר מקום בצ'יפ לדברים נוספים כמו זיכרון מתמון גדול יותר, או לחולופין בגל הקטנת הספק יתכן שהמעבד יוכל לעבוד בתדר גבוה יותר.

### ◆ Code size

- ❖ Long instructions take more time to fetch
- ❖ Longer instructions require a larger memory
  - Important in small devices, e.g., cell phones

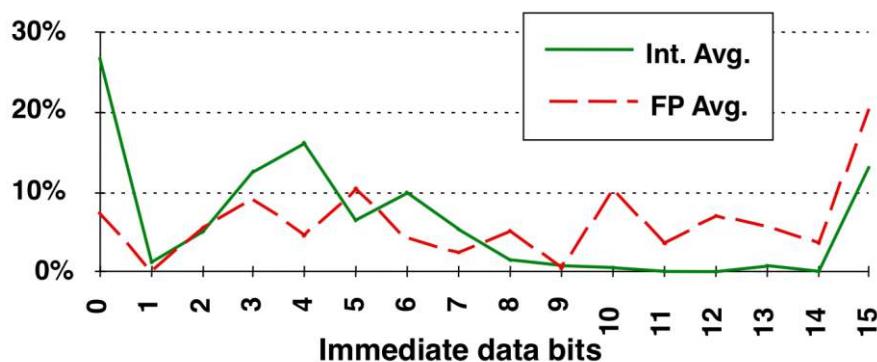
- שיקול נוסף, אם כי משנה בימינו, הוא גודל הקוד - בסופו של דבר הקוד נמצא בזכרון ונרצה שהתכנית תתפס פחות מקום בזכרון: בארכ' מסוימת כל פקודה תופסת ארבעה בתים (32 ביט) בזכרון וכתוצאה לכך אפשר לקבוע פורמט קבוע לכל פקודה. לחולופין אפשר לעשות אופטימיזציה על גודל הפקודה בזכרון ע"י שימוש באורך פקודה משתנה - למשל אפשר לקודד פקודות פשוטות ונפוצות ע"י בית-בודד ופקודות מורכבות ונדירות ב-4 או ב-6 בתים. במקרה של אוריך פקודה קבוע נבזבז 4 בתים על כל פקודה בעוד יתכן שהתכנית כלשהי 80% מהפקודות יכולים להיכנס לבית בודד - במקרה זה נראה שנדאי להשתמש באורך פקודה משתנה וכך לחסוך הרבה מקום. פעם הזיכרון היה מאד יקר ולכך היה כדאי במיוחד להשתמש באורך פקודה משתנה.

از אמרנו שנית להגדר ארכ' עם אוצר פקודות מורכב - יש לאלה מבחינות הקטנה מס' הפוקודות בתוכנית (IC) וכן מבחינות גודל הקוד בזיכרון - אבל הפעולות המורכבות יותר עלולות להשפיע לרעה על התדריות המקסימלית, הספק וכו' .. ב-8x בהתחלה היו הרבה פקודות מורכבות שעוזרו להפחית את IC אבל פגעו בפשטות ה-ISA ולכן גרמו לרעה בתדר / ב-PI. כמו כן פעם הזיכרון היה מצרך יקר יותר מהיום.

זכור חוק אמדל אומר: הסתכל על השכיח ולפי זה קבע את השינויים שיש לעשות. בפרט לא לתקן שינויים לפי מצבים הקיצוניים. נראה שিmorph במקורון זה:

## Architectural Consideration Example

### Immediate data size



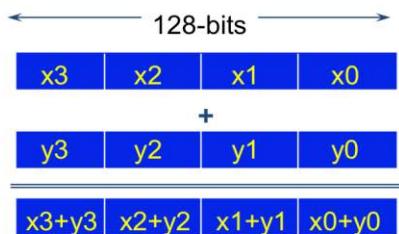
- ❖ 1% of data values > 16-bits
- ❖ 12 – 16 bits of needed

בשיקף רואים התפלגות של כמות הבתים שתופסים הערכים המפורשים (לייטרים, מידע מיידי, למשל בפקודה LD R2[offset] הערך של offset הוא מידע מיידי) בפקודות שmericות עומס אופייני. רואים שרק 1% מהערכים משתמשים בהם תופסים יותר מ-16 ביט (שני בתים). אם ב-99% מההmarker משותמשים בערכיהם "קטנים" אז כדי להקצות עד 2 בתים בלבד לאחסן הערך המיידי, קלומר: אם ה offset ניתן לקידוד כמעט תמיד ע"י 16 ביטים קטן מ  $2^{16}$  אז בתכנון ISA לא צריך להסתכל על המקרא הקיצוני בו שני בתים לא יספיקו אלא על המקרא השכיח בו בדר"כ יספיקו הקצתת שני בתים עבור הערך המיידי בפקודה. لكن כדי לבנות פקודה שימושת בשני בתים בשביל לאחסן את הערך המיידי שלה. כמובן חייבים לאפשר גם לספק ערכים גדולים יותר (המקרא הלא-שכיח), למשל ע"י הוספה פקודה נוספת שלוקחת את הערך המיידי שלה גם מהבתים הבאים בזיכרון, אבל זה מונחה את העיקרונות לפיו כדי לתקן את הפקודות הסטנדרטיות.

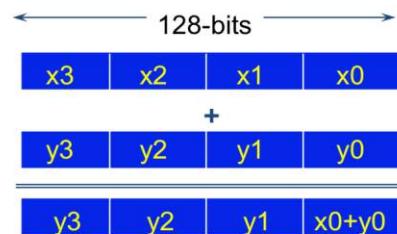
## Software Specific Extensions

- ◆ Extend arch to accelerate exec of specific apps
  - ❖ Reduce IC
- ◆ Example: SSE™ – Streaming SIMD Extensions
  - ❖ 128-bit packed (vector) / scalar single precision FP (4×32)
  - ❖ Introduced on Pentium® III on '99
  - ❖ 8 new 128 bit registers (XMM0 – XMM7)
  - ❖ Accelerates graphics, video, scientific calculations, ...

- ◆ Packed:



- Scalar:



דוג' לשיפור ה-ISA של הארכ' x86 עם הזמן היא התפתחות התמיכה בפעולות וקטוריות (צד שמאל של השקף). אלה פקודות אסמבילר חדשות זמינים לקומפיילר ומספקות את יכולת לחתן שני וקטורים שככל אחד מהם באורך 128 ביט (ואף 256 ביט במעבדים המודרניים) ולבצע פעולה חיבור וקטוריית של 32 ביט מכל וקטור - סה"כ ארבעה חיבורים. זה נקרא פקודת SIMD - simple instruction multiple data

## CISC VS RISC

CISC (Complete Instruction Set computer)

RISC (Reduced Instruction Set computer)

## CISC Processors

- ◆ CISC – Complex Instruction Set Computer

- ❖ The idea: a high level machine language
- ❖ Example: x86

- ◆ Characteristic

- ❖ Many instruction types, with many addressing modes
- ❖ Some of the instructions are complex
  - Execute complex tasks, and require many cycles
- ❖ Small number of registers, in many cases not orthogonal
  - E.g., some operations supported on specific registers
- ❖ ALU operations directly on memory
 

```
add eax, 7680[ecx] ;eax ← MEM[7680+ecx]+eax
```
- ❖ Variable length instructions
  - common instructions get short codes ⇒ save code length

## CISC Drawbacks

- ◆ Complex instructions and complex addressing modes

- ⇒ complicates the processor
- ⇒ slows down the simple, common instructions
- ⇒ contradicts Make The Common Case Fast

- ◆ Not compiler friendly

- ❖ Non orthogonal registers
- ❖ Unused complex addressing modes

- ◆ Variable length instructions are a pain

- ❖ Difficult to decode few instructions in parallel
  - As long as instruction is not decoded, its length is unknown
    - ⇒ Unknown where the inst. ends, and where the next inst. starts
- ❖ An instruction may cross a cache line or a page

## מעבדי CISCO

ההבדל הגדול בין שני סוגי הארכ' הוא שבמעבדי CISCO יש הרבה סוגים של פקודות (אבני בניין לתוכנית) בעוד ב-RISC מושך יותר מצומצם. חלק מהפקודות ב-CISC הן מאוד מורכבות למשל rep\_move\_stream שמעתיקה חוץ בזיכרון למיקום אחר בכרונ' והיא מתבצעת באופן פנימי ע"י ביצוע מס' פקודות במעבד. ב-RISC זה לא קיים - אם רוצים דבר כזה צריך לכתוב את הפקודות במפורש.

הארכ' x86 היה CISCO בעוד רוב הארכ' הצעירות יותר הן RISC - 10 או 20 שנה לאחר מכן נולדו מעבדי CISCO שנעודו לפתור את כל הבעיות שנובעות מהמורכבות של מחשבי CISCO, בפועל עד היום הפיתוח של CISCO פתר הרבה מהבעיות שלא.

כמה מהמאפיינים של מעבד CISCO הם:

- למעבד CISCO הרבה סוגים של פקודות, עם הרבה שיטות מייען - למשל הרבה סוגים של פקודות LD: פקודה שמקבלת כתובת מפורשת (מייען ישיר), אחרת שניגשת כתובת שנמצאת בכתבota (מייען עקיף), אחת מסויפה היסט לכתובת מפורשת/עקיפה ואז ניגשת לזכרון בכתבota המתקבלת וכו'.. דוגמה: [imm R1 R2 LD]. לעומת זאת RISC הפקודות פשוטות ואין יותר מדי סיבוכים - כל פקודות ה-Load נראות דומות, אין שיטות מייען מורכבות והשימוש של כל פקודה על הצ'יפ הוא הרבה יותר פשוט. למשל ב-RISC תחילה היינו מבצעים את הפקודה  $imm + R1 - R2 <- R1$  ורק אז מבצעים  $R1 <- R2$ .
- במעבד CISCO יש פחוות רגיסטרים כלליים (General-purpose) שהקומפайлר יכול לעבוד איתם ויתר רגיסטרים ייעודיים - הרגיסטרים אינם בלתי-תלוים (כלומר רגיסטרים שניתנים לשימוש בכל אחד מהם לכל שימוש) אלא פעולות שונות עובדות על רגיסטרים מיוחדים. למשל POP-POPU PUSH משנות תמיד את ה-ESP - אך ESP אינו רגיסטר General Purpose, כמו עברו IP או CLR3 (למד לעליו בהמשך הקורס).
- ב-CISC ניתן לבצע פעולות ALU ישירות על הזיכרון, למשל פקודה Add שכוללת בתוכה פקודה LD: ADD EAC, 7680[ECX]
- ב-DEC CISC כל פקודה יכולה להיות בעלת אורך משתנה - למשל כי 80-90% מהפקודות שעשוות לשימוש במידע קטן כפי שראינו. אורך פקודות משתנה עוזר מבחינת דרישות הקוד, אבל מאידך הוא סיווט לשלב פענוח הפקודות: זה מאד מסובך במערכות מודרניות שמכילות יותר מליבת אחת, כי כשכמה מעבדים מבצעים פקודות וקוראים במקביל פקודות מהזיכרון, המעבד צריך מראש להבין איזה בתים בזיכרון זהות עם כל פקודה, בניגוד למקורה ש-4 בתים הם אורך פקודה קבוע. ב-RISC אפשר לבצע במקביל הקלות פענוח (decoding) של הרבה פקודות בעוד את ה-decoding ב-CISC צריך לעשות לכל פקודה בנפרד כדי להבין את גבולות הזיכרון שלה. למשל אם יש פקודה בעלת 16 בתים צריך קודם לקרוא את הבית הראשון ולראות האם זו פקודה חוקית - אם כן לקרוא עוד שני בתים ולהבין מהם היכן הפקודה מסתימת.. זו בעיה סריאלית מטבעה וכשהיא נעשית במערכת מרובת-ליביות היא מקטינה את האפקטיביות של המערכת המקבילה: צריך לקודד כל פקודה בנפרד כדי להבין את הגודל שלה.

החסכנות של CISCO הן שההוראות המורכבות ומס' הרכ' של שיטות המייען גורמים לסיבור הלוגיקה ולכך לעליית תדר מוגבלת יותר עם הזמן - סותר את חוק אמדל שאומר "תń טיפול פשוט ויעיל למקרה שכח" (ולא לקרים אוטריים ונדרירים יחסית). בנוסף זה מנסה על קומפайлר לצוריך לעבוד קשה כדי להתאים את הרגיסטרים הלא-אורותוגונליים לסוגי הפקודות, כל פעם להבין איזה רגיסטר מתאים וgom פניו. כמו כן הבעיה של פענוח תחת אורך פקודות משתנה היא בעיה מאוד קשה.

בפועל הארכ' 8x, שהוא CISC, הצליחה להתמודד עם הקשיים הנ"ל ולשמור על העשור הרב של הארכ' בlij פגיעות בביצועים לעומת RISC. למשל הצלicho למצוא דרך לפטור את בעיית הקידוד בצורה מקבילית (או לפחות לא לסייע ממנה יותר מדי) וכן לקבל מכונה שתוכל במקביל לפענו פקודות באורך משתנה.

דוג' אמפירית לסבירו המיורר של מעבדי CISC: מתוך כל אוסף הפקודות שזמןנות עבד מעבד 8x ומדגם של כל התכניות שנכתבות עבورو, עולה כי 10% מגוון הפקודות האפשרות (אוסף הפקודות הסטטי) מהווים 96% מכל הפקודות שימוששות בתכנית מסוימת (אוסף הפקודות הדינמי בתכנית) לעומת 96% מהקוד של תכניות מתבצע ע"י 10% מהפקודות שימוששות ע"י ISA - זה מעיד על סיבוך מיותר בבחירה הפקודות בהן תומכת הארכ'.

## מעבדי RISC

### RISC Processors

#### ♦ RISC – Reduced Instruction Set Computer

- ❖ The idea: simple instructions enable fast hardware

#### ♦ Characteristics

- ❖ A small instruction set, with few instruction formats
- ❖ Simple instructions that execute simple tasks
  - Most of them require a single cycle (with pipeline)
- ❖ A few indexing methods
- ❖ ALU operations on registers only
  - Memory is accessed using Load and Store instructions only
- ❖ Many orthogonal registers – all instructions and all addressing modes available with any register
- ❖ Three address machine: Add dst, src1, src2
- ❖ Fixed length instructions

#### ♦ Complier friendly – easier to utilize the ISA

- ❖ Orthogonal, simple

#### ♦ Simple architecture ⇒ Simple micro-architecture

- ❖ Simple, small and fast control logic
- ❖ Simpler to design and validate
- ❖ Leave space for large on die caches
- ❖ Shorten time-to-market

#### ♦ Existing RISC processor are not "pure" RISC

- ❖ E.g., support division which takes many cycles
- ❖ Examples: MIPSTM, SparcTM, AlphaTM, PowerTM

#### ♦ Modern CISC processors use RISC μarch ideas

- ❖ Internally translate the CISC instructions into RISC-like operations
  - the inside core looks much like that of a RISC processor

בטכנולוגית RISC החיים הרבה יותר פשוטים - מעבדי RISC למדו מהבעיות של CISC והוציאו ISA הרבה יותר נקי. אוצר הפעולות הוא בסיסי, כך שבארך' זו העבודה של הקומפיילר קליה יותר - אין הרבה פקודות לבחור מהן ותמיד יודעים מהי הפקודה הנכונה:

• מעט פורמטים של פקודות (למשל ב-S-ISA, MIPS, עליו למדנו ב"תיכון לוגי", יש שלושה פורמטים בלבד).

• אוסף הפקודות הזמיןות לקומפיילר הוא הרבה יותר קטן - מעט הפקודות מתאימות לרוב המקרים הכלליים ורובן מתאימות לכל המקרים. בכך-RISC נמוש ביכולת פקודות את אותה בעיה/אלג' שallow במעבד CISC נמצא פקודה ספציפית יותר שמתאימה לבעה. עם זאת זמן הביצוע של כל פקודה פשוטה כנראה הרבה קצר יותר מאשר זמן הביצוע של פקודה CISC מורכבת. אם זאת גם מעבדי RISC מוצאים פקודות מורכבות ב-ISA, וגם שם יש פקודות שדורשות עבודה קשה - למשל פקודות החילוק, div, קיימת בכל ארץ' מודרנית והיא מורכבת מאוד ארוכה - גם במעבדי CISC.

• במעבדי RISC הארכ' הפוצה משרה עיצוב יותר פשוט של הצ'יף כלומר מיקרו-ארQUITקטורה יותר פשוטה. פקודות פשוטות, שביציאות טאסקים פשוטות, מאפשרות מימוש פשוט יותר ולכן מסיבות להקטין את זמן המחזיר (=לעבוד בתדר יותר גבוה). דוג' לפשטות הפקודות היא למשל שב-RISC פקודות לא עובדות ישירות על הזיכרון אלא רק על רגיסטרים וכך אם צריך לעבוד על הזיכרון עושים קודם Load, ואז מחשבים את הערך שהבנו.

• במעבדי RISC יש הרבה יותר רגיסטרים והם כולם אורתוגונליים - הפעולות מתבצעות עליהם באותו אופן וכך אפשר להשתמש בכל רגיסטר לכל צורך.

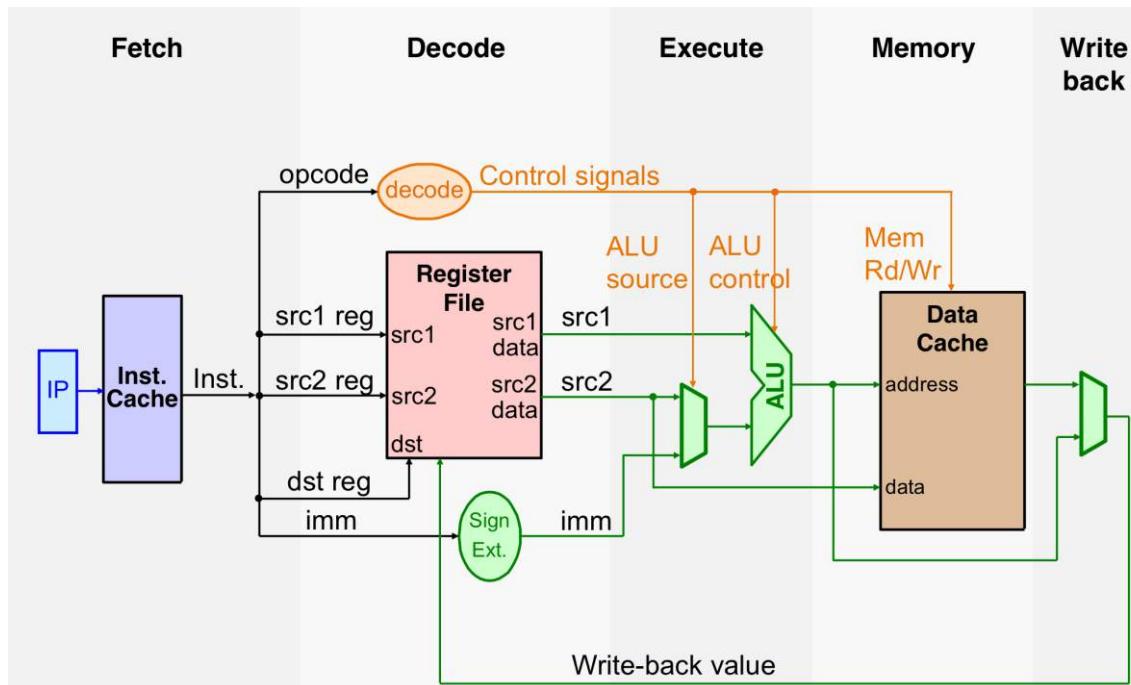
- סיכום קורס
- ב-RISC היעד של פקודה ניתן באופן בלתי-מפורט ע"י שני המקורות של הפקודה, ז"א היעד של הפקודה הוא תמיד גם הארגומנט הראשון שלה, בעוד ב-CISC פקודה מכילה רגיסטר-יעד במפורש וכן שני מקורות.
  - אורך הפקודות במעבדי RISC הוא קבוע, במעבד CISC בדרך כלל להשתנות (דוג' כך היא x86).

## הרצאה מס' 2: מעבד מצונר Pipeline

### מעבד ה-MIPS

נתחיל בסקירה של מעבד ה-MIPS שלמדו בקורס תיכון לוגי. כרגע אנחנו מביטים על המעבד בגרסתו הלא-מצונרת אבל עדין ניתן לראות בו חלוקה עקרונית של פקודה לשלבבי ביצוע שונים ובכל שלב ניתן להזות יחידות ביצוע משלו.

:Fetch, Decode, Execute, Memory, Write Back הם MIPS הם הפלט של ביצוע הפקודה במעבד ה-



1. בשלב ה-Fetch מביאים את תוכן הפקודה הנוכחית (כלומר מוצבעת ע"י ה-Instruction Pointer) מתוך ה-Instruction Cache, שם מוקצים בדיק אربעה בתים לכל פקודה (ב-MIPS) ובסוף ביצוע ה-Fetch זה ה-Instruction Pointer שכבר קודם ב-4, כלומר הוא כבר מצביע לפקודה הבאה שנעשתה לה. בוגדול, כרגע, כרך Cache (מטען), מכיל חלק מהזיכרון הרלוונטי לתכנית ומאפשר למעבד גישה מהירה לאותו חלק של הנתונים. יש נפרדדים לנוחונים (Data) ופקודות (Instructions) - ב-Fetch מביאים רצף בתים שמהווה את הפקודה מתוך ה-Instruction Cache-Pointer).

2. בשלב ה-Decode (פונוח) מזהים את הפקודה: מסתכלים על הביטים של הפקודה ומתוכם מייצרים את סיגנלי הבדיקה שישנו הוראות לכל חלקו המעבד. כל חלק צריך להבין מה הוא צריך לעשות ביצוע פקודה זו, בפרט בשלב זה מסתכלים לראשונה על ה-Opcode ומビינים איזה פקודה זו. סיגנלי הבדיקה אומרים למשל ל-ALU לבצע פקודת חיבור, או לזכור האם צריך לנקרא או לכתוב. ב-S-PIPE שלב זה גם קוראים את הרגיסטרים הרלוונטיים לפקודה מתוך ה-Register File, למשל אם פקודה משתמשת בערכיהם ששמורים ברגיסטרים, אז כולם נקוראים בשלב הפענוח. במעבדים מודרניים זה לא ככה והרגיסטרים נקוראים יותר מאוחר, בהתאם גם היביצוע (בפועל היצור, Pipe) מחולק להרבה יותר שלבי-ביצוע. במעבד מצונר (שנראה עוד מיד)

- הפקודה תשוחב איתה לכל שלב את סיגנלי הבדיקה שעוד נותרו כדי לתפעל את שאר השלבים שעלייה לעבור, ובכל שלב יקרו ערכיו הבדיקה המתאימים (והם לא ימייצגו לשלב הבא)
3. אחרי פענוח הפקודה וקריאת הרגיסטרים מגיע שלב הביצוע, Execute, בו מבצעים את פעולה ה-ALU הנדרשת. למשל מבצעים חיבור / חיסור של ערכיהם שנקרו מקובץ הרגיסטרים.
  4. בשלב הזיכרון (Memory) קוראים / כתובים לזכרון את הנתונים של הפקודה (אם הפקודה צריכה לבצע זאת, למשל store כותבת לזכרון בשלב זה). בדרך"כ הנתונים נקראים/כתבם מזיכרון ה- Data Cache שמכיל חלק קטן של נתונים מהזיכרון הרלוונטיים לתוכנית (לא נגישים ישירות לזכרון הראשי).
  5. בשלב האחרון, שלב Write Back, הפקודה כותבת את ערך החזרה אל רегистר היעד שלה (למשל את תוצאה של פעולה החיבור שחייבה).

## מעבד מצונר - Pipeline

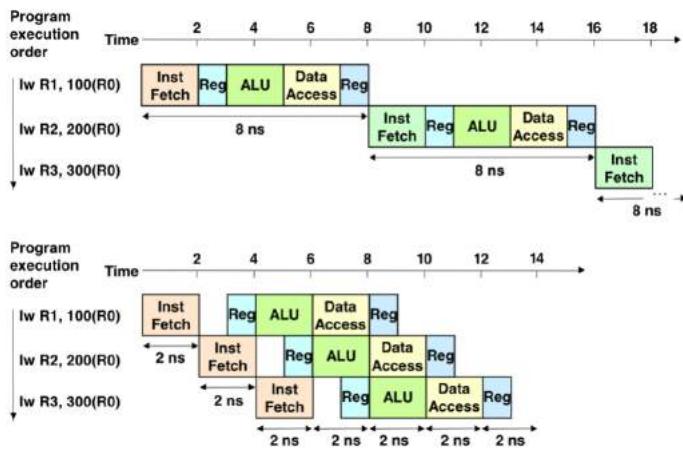
במימוש הבסיס שראינו למעבד MIPS אין צינור (Pipeline) - כל פקודה מתחילה רק אחרי שהפקודה לפניה הסתיימה. זה לא עיל - Pipeline משמשו עבורה בשיטת סרט-גע: הדוג' הקלאסית היא פס-ייצור של מכוניות. נניח שליצור מכונית שלושה שלבים: חיבור המרכיב, הוספת המנוע ולבסוף נעשות פעולות גימור. לכל שלב יש זמן ביצוע (Latency) שלו. כך יראה תהליך של ייצור מכוניות ב-Pipeline:

- המכונית הראשונה ב-Pipe-1 נמצאת שעה בחלק הראשון של חיבור המרכיב.
- לאחר שהמרכיב שלה מוכן היא עוברת לתchanת ההרכבת השנייה. נניח בשלב זה לוקח שעתיים. במקביל לכניתה שלב הרכבת המנוע נכנסת מכונית שנייה ל-Pipe והיא מתחילה את ביצוע השלב הראשון.
- הסרט הנע זו שב Rak אחרי שעתיים כי המכונית בשלב המנוע לוקח שעתיים. נשים לב שהוא אומר שהמכונית השנייה, שהייתה בשלב המרכיב, חיכתה ללא מעש במשך שעתיים אחרי סיום הרכבת המרכיב שלה - היא סיימה אבל לא יכולה להתקדם כי המכונית בשלב אחריה עוד לא סיימה. אחרי שעתיים המכונית הראשונה שנכנסה לצינור עוברת לשלב הגימור, השני לשלב המנוע ובמקביל מכונית השלישי שלישית מתחילה את שלב המרכיב.
- המכונית הראשונה שנכנסה לצינור יכולה לסיים את הרכבת המנוע, כמו השנייה, גם תהיה תוקעה שעעה לחינם בשלב המרכיב - בגלל שהוא החליף את המכונית הראשונה להתקדם רק בכל שעתיים.
- מכונית נוספת נכנסת ל-Pipe-2 בזמן שאחרת יוצא ממנה - וכך הלאה..

נשים לב:

- ה-*Latency*, הזמן שמכונית נכנסת ועד שהיא יוצאה, הינו ארבע שעות.
  - ה-*Throughput*, כל כמה זמן מופקת מכונית חדשה, הוא שעתיים.
- ב-Pipeline פחות מזמן אוטנו *Latency* יותר מאשר *Throughput*. יחד עם זאת תהליך ה-Pipe-2 מושפע במידה מה מה-*Latency*: למשל מכל אתחול של Pipe ריק ועד שיוצאת המכונית הראשונה עברו זמן שדרוש לייצור מלא של מכונית אחת, ככלומר *Latency* של ה-Pipe. גם במעבד, כמו הדוג' שראינו כתה, לעיתים תכוופות נאלצים לאותל את ה-Pipe ואז להשלים את הפקודה הבאה אחרי זמן ה-*Latency* שלו.

## Pipelining Instructions



- ◆ Pipelining does not reduce the **latency** of single task, it increases the **throughput** of entire workload
- ◆ Potential speedup = Number of pipe stages
  - ❖ Pipeline rate is limited by the slowest pipeline stage
    - ⇒ Partition the pipe to many pipe stages
    - ⇒ Make the longest pipe stage to be as short as possible
    - ⇒ Balance the work in the pipe stages
- ◆ Pipeline adds overhead (e.g., latches)
  - ❖ Time to "fill" pipeline and time to "drain" it reduces speedup
  - ❖ Stall for dependencies
    - ⇒ Too many pipe-stages start to loose performance
- ◆ IPC of an ideal pipelined machine is 1
  - ❖ Every clock one instruction finishes

כשמדובר במימוש מצונר של מעבד חיצוניים "לרווח" אותו, קלומר לקבוע את זמן המחזור שמייד בין שלבים, בהתאם לשלב האיטי ביותר. נניח שהשלב העיבוד הראשון במעבד הראשון לוקח 2ns והאחרים 1ns כך שסה"כ לוקח 6ns לסיום את הפקודה - אז השלב האיטי ביותר יקבע זמן המחזור שלנו להיות 2ns. בהשוואה למעבד לא-מצונר בימיוש המצונר - throughput. תחתפר פ' 3 - כי בימיוש המצונר בכל 2ns נסיים פקודה ובלא מצונר בכל 6ns. נשים לב שלעומת throughput בימיוש המצונר Latency גdag בהשוואה למימוש הלא-מצונר (אח 8 מול 6ns) בגין שחייבים להמתין זמן שווה לזמן השלב האיטי ביותר בכל אחד משלבי ה-Pipe. עם זאת, ה- throughput של המימוש המצונר טוב יותר מה- throughput של המימוש הלא-מצונר, כך שאנו מסיימים כל פקודה באח 2 במקומות באח 8 בלי-צינור. כאמור ה- throughput במעבד הרבה יותר חשוב מה- throughput כי הוא מבצע הרבה פקודות ולא אחת בלבד. לסייע מיפור את ה- throughput לא בהכרח מיפור את Latency אלא את ה- throughput.

נגידו את ה- Speedup (שיפור) בין שני מעבדים להיות היחס בין throughput שלהם. למשל היחס בין throughput של המעבד המצונר לבין throughput של המימוש הלא-מצונר. ככל אצביע חשוב הוא שה- Speedup הטוב ביותר ביזהר שנitin לקבל ע"י הוספה Pipe כמו' השלבים בו - אם למשל ניקח תהליך וחלוק אותו לעשרה שלבים אז ניתן להגיע עד לשיפור של פי 10 בהשוואה לתהליך הלא-מחלוק. ככל נוסף הוא שהשלב שקובע את throughput ב-Pipe הוא השלב האיטי ביותר! לכן כדי שהשלב האיטי ביותר יהיה גם השלב המהיר ביותר - ככל אצביע כדי לסייע הנדרשת כדי לבצע פקודה וחלוקת אותה לתחנות כך שמאז אחד יהיה הרבה תחנות אז מנסים לחת את כל העבודה הנדרשת כדי לבצע פקודה וחלוקת אותה לתחנות כך שמאז אחד יהיה הרבה תחנות כדי שה- Speedup יוכל להיות טוב יותר, וכך שני התוצאות יהיו כמה שייתר שות באורך כי אנחנו יודעים שמי שיכתיב את throughput תהיה התנה האיטית - ואין טעם לחלק לעד שלבים כשלב אחר מגביל אותו.

גם העבודה ב-Pipeline עצמה מוסיפה תקופה, למשל במקרה בפועל יש זמן בו מכוניות נעה מתחנה לתחנה - זמן זה לא היה קיים אם המכונית הייתה מורכבת במקום אחד (כלומר בימיוש הלא-מצונר). גם במעבד החלוקה של שלבים מוסיפה תקופה, למשל של חומרה המשמשת להעברת מידע בין שלבים שונים ב-Pipe שנדבר עליה מיד. כמו כן יש תקופה של אתחול ומילוי ה-Pipe. זו עוד סיבה לחלק ליותר ויותר שלבים ולצפות לקבל throughput הולך וגדיל לא יעבד - בגין תקופות שנראות אותן.

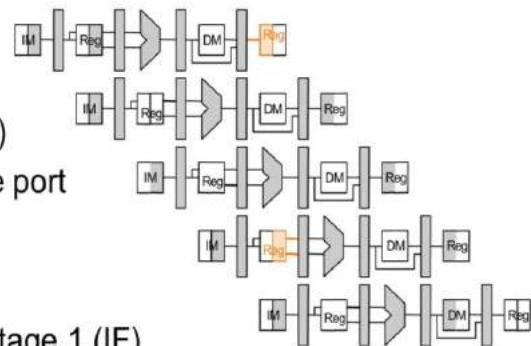
למשל ב-MIPS המצונר יש חמישה stages: Fetch, Decode, Execute, Write-Back. מותאמים לשלב הביצוע Memory, Write-Back כמה עשרות - לא מחלקים פקודה למאורות שלבים.

מי שמחזמן את התנועה של המידע בין השלבים ב-Pipeline זה השעון: כזכור בכל מערכת סינכרונית יש שעון שמחזמן את כל הרכיבים הסינכרוניים, למשל פלייפ-פלופים, לאצ'ים (Latches) וכו'. ב津ור משך הזמן אחריו עוברים מתחנה לתחנה הוא מחזיר שעון אחד. ככל שהמחזיר קצר יותר הה-Pipe מתקתק יותר מהר וב-Pipeline אופטימלי, בגלל שבכל מחזיר שעון המידע יתקדם תחנה אחת, אז בכל מחזיר שעון נסיים פקודה. لكن מס' המוחזרים לפקודה (IPC - Instructions Per Cycle) האופטימלי הוא 1. תזכורת: ה IPC לא נוגע לביצוע פקודה ספציפית כלשהו אלא לכל כמה זמן בממוצע מסוימים פקודה. אז במקרה האופטימלי  $IPC=1$  - בפועל Pipeline אמיתי לא מצליח להגיע ל-1 בגלל עיכובים שונים, שנראה מיד.

בשבייל לאפשר למעבד MIPS שראיינו בהתחלה את יכולת לעבוד בצורה Pipeline כל פקודה שעוברת לשלב הבא צריכה לגרום אתה את כל המידע שנדרש לה כדי לעשות את השלבים שנותרו. למשל כשפוקודה עוברת משלב Decode-Latch או הקלטים ב-Fetch כבר לא יציבים (התחלפו לקלטים של פקודה אחרת), לכן כל פקודה צריכה לנעול את הקלטים שלה באיזשהו Latch ולקחת את זה אתה. דוג' נוספת היא סיגנלי הבקרה לכל שאר השלבים שיש לבעור הפוקודה. שיקול זה נכון בין כל שני שלבים ב津ור ולכן יש להוסיף בין כל שני שלבים לאצ'ים (Latches): המטרה של האצ'ים היא שהלוגיקה של כל שלב תעבור על האינפורמציה של הפוקודה הרלוונטית לו. המידע בלאצ'ים מכיל מידע עבור השלב הנוכחי של הפוקודה והשלבים שעוד נותרו לה. כל האצ'ים הנ"ל אינם הכרחיים בימוש הלא-מצוור כי שם הקלטים יציבים לפחות כל ביצוע פקודה כלשהו.

## Structural hazards

- ◆ **Different instructions using the same resource at the same time**
- ◆ **Register File:**
  - ❖ Accessed in 2 stages:
    - Read during stage 2 (ID)
    - Write during stage 5 (WB)
  - ❖ Solution: 2 read ports, 1 write port
- ◆ **Memory**
  - ❖ Accessed in 2 stages:
    - Instruction Fetch during stage 1 (IF)
    - Data read/write during stage 4 (MEM)
  - ❖ Solution: separate instruction cache and data cache
- ◆ **Each functional unit can only be used once per instruction**
- ◆ **Each functional unit must be used at the same stage for all instructions**



הוא קונפליקט בביצוע שנוצר Ci פקודה אחת מנסה לגשת לחומרה שגם פקודה אחרת ניגשת אליה. נתخيل מדוג'ה: ה-Register File-RFCיב חומרה במעבד שמכיל את הרגיסטרים, משחק תפקיך בשני שלבים בפייפ: גם בשלב Decode,-CSKוראים ממנה ערכיהם של רגיסטרים, וגם בשלב Write Back,-CSKותבים תוצאה אל רגיסטר. נניח שיש לנו פקודה שמתבצעת בפייפין ופקודה שנייה שמתבצעת מחזיר אחד אחריה, כך שבמקביל הפוקודה הראשונה נמצאת בשלב השני והפקודה השנייה נמצאת בשלב ראשון, וכך הלאה בכל שלב בפייפ תמצא פקודה.

כלשהי. יתכן ופוקודת השגיעה לשלב ה-Write Back רוצח לכתוב ל-Register File ובאותו מוחזר פוקודת שהתחילה ארבעה מוחזוריים אחרים ומוצאת ב-Decide רוצח לקרווא מה RF, ז"א פוקודת אחת במוחזר האחרון שלה רוצח לבצע כתיבה ופוקודת אחרת במוחזר השני שלה רוצח לקרוא - כך נוצר קונפליקט על המשאב. Structural hazard שזכה, בו פוקודת אחת מנסה לגשת לחומרה שגם פוקודת אחרת ניגשת אליה, נקרא read-konfliket כדי להתמודד עם זה צריך לתכנן את קובץ הרגיסטרים כך שהוא ידע לקרווא ולכתוב נתונים במקביל. נשים לב שכשאין צינור הקונפליקט לא יתכן כי המעבד מוקדש לביצוע כל פוקודת בנפרד.

## סימולציה של הריצת תכנית פשוטה ע"י ה-Mips Pipelined

נדמה את **ביצוע התכנית** הבא:

- ```
#0: LOAD 100, r1  
#4: SUB r2, r3  
#8: AND r4, r5  
#12: OR r6, r7
```

- בהתחלה ה IP מצביע לכתובת 0 בזיכרון (הפקודה ראשונה). במהלך המחוור הראשון הInstruction Cache מספק לנו את הפקודה ע"י ה-IP, ובמקביל יש לנו מחבר, Adder, שמוסיף 4 ל-PC כך שהוא מצביע אל הכתובת הבאה. בסוף המחוור הראשון הכתובים שקרוינו מכתובות 0 ננעלים בלאי' שמקדים לשלב ה-Decode, הפלט של ה-Adder (4) נשמר ב IP ובעצם סיימנו את המחוור הראשון.
  - במהלך המחוור השני מתבצע Fetch נוספת בו ה Instruction Cache מביא לנו את פקודת החישור במצבה ע"י ה IP (בכתובת 4) וכן ה-IP מקודם ל-8. במקביל LOAD מתחילה את השלב השני שלו, Decode, ומتابסת על מה שנעול עבורה לפענוח הפקודה כך שהשאר השלבים נדע מה לעשות. כמו כן בשלב ה-Decode מסתכלים על שדות הרגיסטרים בפקודה כדי להבין איזה רגיסטרים לקרוא מה-Register File. ניתן לדעת איזה רגיסטרים צריכים לקרוא לפני פענוח סוג הפקודה כי לכל פקודה ב-MIPS מבנה קבוע מראש שמכתיב איפה בפקודה (באייה ביטים) מצוינים מס' הרגיסטרים - זה פשוט למדי ולא נעשו במעבדים מודרניים. בסוף המחוור השני מה שקרוינו מה פקודות הרגיסטרים מס' הרגיסטרים - זה נכון למדעי ולא נעשה במעבדים מודרניים. בסוף המחוור השני מה שקרוינו מה פקודה Fetch (הלאז' עבורו שלב ה-IP), Decode (הלאז' עבורו שלב ה-IP) יונל עם המידע יכיל את הכתובת הבאה, 8, והלאז' הבא אחרי ה Decode (הלאז' עבורו שלב ה-Execute) יונל עם המידע שדרוש לפקודות LOAD לשאר השלבים שלה - בין היתר נונל עבורה מס' רגיסטר וערך מיידי (ליטרל, מס' קבוע) שהוא הולכת לחבר לתוך הרגיסטר בשלב הבא כדי לחשב את הכתובת מנתה צריך לקרוא.
  - במהלך המחוור השלישי הבא יעבדו שלוש פקודות במקביל: הראשתה (המודמת בציור) היא AND שאנו חנו כרגע קוראים אותה מה RegFile,Instruction Cache, השניה היא החישור שכרגע מפענחים אותה וקוראים מה-RegFile הארגומנטים שלה, והפקודה הבאה היא ה-LOAD שכרגע מחשבים את הכתובת שלה - מבצעים חיבורו של הרגיסטר שלה עם הערך המיידי שלה.
  - בשלב הבא הערך ממנו עושם LOAD, כבר יהיה ידוע לנו והוא תגייע לשלב הטעינה מהזיכרון. שאר הפקודות יתקדמו גם הן.
  - בשלב הבא LOAD תכתוב בשלב WB שלה את הערך שתענה אל ה RF ברגיסטר 1ז ובזאת היא יכולה לצאת מהמעבד. פקודת החישור תגייע לשלב הרביעי בציור, שלב הזיכרון. למעשה אין ל-SUB מה לעשות בשלב הזיכרון ולכן היא סתם ממחכה.
  - במחוור הבא פקודה LOAD כבר לא נמצאת במעבד ובפרט המידע עבורה לא קיים אף אחד מהלאז'ים - וכך הלאה.

נשים לב שבכל שלב העבודה שהכל נועל בלאצ'ים ונשאר יחד עם הפקודה מאפשר לה לעשות את העבודה שלא למרות שהקלטים של המעבד כבר השתנו בשליל פקודות אחרות.

## Data Hazards

בתכנית הראשונה לא היו תלויות בין הפקודות, וכל פקודה יכולה להתקדם בציינור באופן עצמאי. עכשו נראהת תכנית עם תלויות בין פקודות.

וניה ויש לנו פקודת חיסור שמחשבת ערך ושומרת אותו ברגיסטר 2 ווניה ושהפקודות העוקבות אחריה, למשל SW ו- (Store Word) SWColon משתמשות ב-2 קלקט: OR, AND

SUB r1, r3, r2

AND r2, r4, r5

OR r2, r6, r7

SW r2, r8, r9

לכוארה יש בתכנית בעיה: פקודת החיסור כותבת את הערך שהוא מחשבת רק בשלב ה Write Back אבל AND קוראת את 2 לפניו כן, כשהיא בשלב Decode-Execute וכנ"ל עבר SUB בשלב AND-Execute. לכן אם AND-Execute קרא את ה register file זה יהיה לפני המחוור בו SUB כתבה אליו את התוצאה שלה, וכך מובן מה לא מה שהמתקנת/קומפילר התכוון אליו כשכתב מקובץ הרגיסטרים היא קיבל את הערך הישן - וזה מובן מה לא מה שהמתקנת/קומפילר התכוון אליו כשכתב את התכנית - ברור שהוא רוצה שהערך שיקרא ע"י הפקודות אחורי ה-SUB הוא זה שהוחשב ע"י SUB. אנחנו צריכים לפתור את הבעיה הזאת ולספק לתוכנת את הערך שהוא התכוון אליו - כלומר הערך שאמור להיות ב-2 לאחר פקודת SUB. לשם כך מותר לנו לעשות כל מיני טריקים אבל חשוב להבהיר שבסכל מקרה אסור לשנות את הסמנטיקה של התכנית - כלומר להחליט שהוא בסדר שהערך הישן נקרא והתכנית תמשיך את הביצוע עם הערך הזה. כלומר אחרי סיום הביצוע מצב האיך רצון חייב להראות כאלו התכנית חושבה באופן הסדרתי בו היא נכתבה. גם פקודת-OR קוראת את 2 לפניו שנעשה ה Write Back ע"י החיסור אבל שימוש לב שבמקרה של החיבור הוא גבולי כי זה קורה באותו שלב בו פקודת החיסור כותבת. לא ברור אם היא תוכל לקרו את הערך הנכון או לא בדבר בהמשך על הפתרון של בעיית הקריאה/כתיבה ל-RF באופן מוחזר כפי שהוא מומש במעבד MIPS. רק הפקודה الأخيرة, SW, קוראת את ה RF ממש אחורי שהחיסור סיממה, ולכן היא "באזור הבתווח" כלומר תקרא את הערך הנכון. תלויות כפי שראינו כאן, של פקודה שרצחה לקרוא ערך לפני שהפקודה שלפניה הספיקה לכתוב אותו, נקראת תלות RAW (Read After Write).

הפתרון הכיו פשטי לבעית התלויות הוא לצפתה מהקומפילר להזיר NOP בכל מקום בו יש צורך לחכות לערך. זה פתרון שם את האחריות על המתקנת, כי הוא דורש שהוא יודיע למיקרו ארכ' כך שידע לשוטל פקודות ריקות. אנחנו לא רוצים שהמתקנת יחשוף על המיקרו ארכ' ולכן כהתקומות הראשונה נציג פתרון דומה ל-sopch: אלא שהחומרה תכניס אותן במידת הצורך - "בוואות" ככל שהיא פוללה שיזמות ע"י המכונה נקראות stalls: AND-Execute תזהה את התלות ותעכיב את הפקודה AND מלקרא את הערך מה-RegFile וכמובן שיחד עם ה-OR-Execute תעציב גם את כל הפקודות שאחראית כי הפקודות מתבצעות עפ"י סדר הופעתן בתכנית. במקרה זה העיכוב נדרש עד שהחיסור יעשה WB - אח"כ אפשר להמשיך את ביצוע הפקודות ב-Pipeline. תכנית מזהים את הצורך בעיכוב ע"י השוואת רגיסטר היעד (dest) ורגיסטרי המקור (src) בין פקודות ורואים אם יש התנגשות. שימוש בא-stalls הוא פתרון שעובד, כאמור הוא מפיק תכניות נכונות, אבל הוא ממש לא טוב כי אם כל פעם נבצע stall זה מאוד יפגע ביציאות: תלויות כאלה הן דבר טבעי ונפוץ, ולא יוצא דופן, שմבאות איזשהו קשר סמנטי בין הפקודות בתכינה, ולכן הן קוראות המון..

נציע פתרון אחר לפתרון התלוויות בין פקודות בזיכרון: נשים לב שאمنם כתיבת הערך של החיסור ל-RF באמצעות רק בשלב WB אבל אותו ערך ימצא בזיכרון החל מסוף שלב EXE. אם נוכל לחת את ערך מסוף שלב EXE ולהעביר אותו לתחילת שלב EXE, בו נמצאת פעולה AND, אז היא תוכל "لتפס" את הערך והשתמש בו לצרכיה. הרעיון של זיהוי העובדה שהערך כבר זמין במכונה והעברתו לשלב אחר שצריך אותו לפני שהערך נכתב לרגיטרים/לזיכרון, קוראים Forwarding או Bypass.

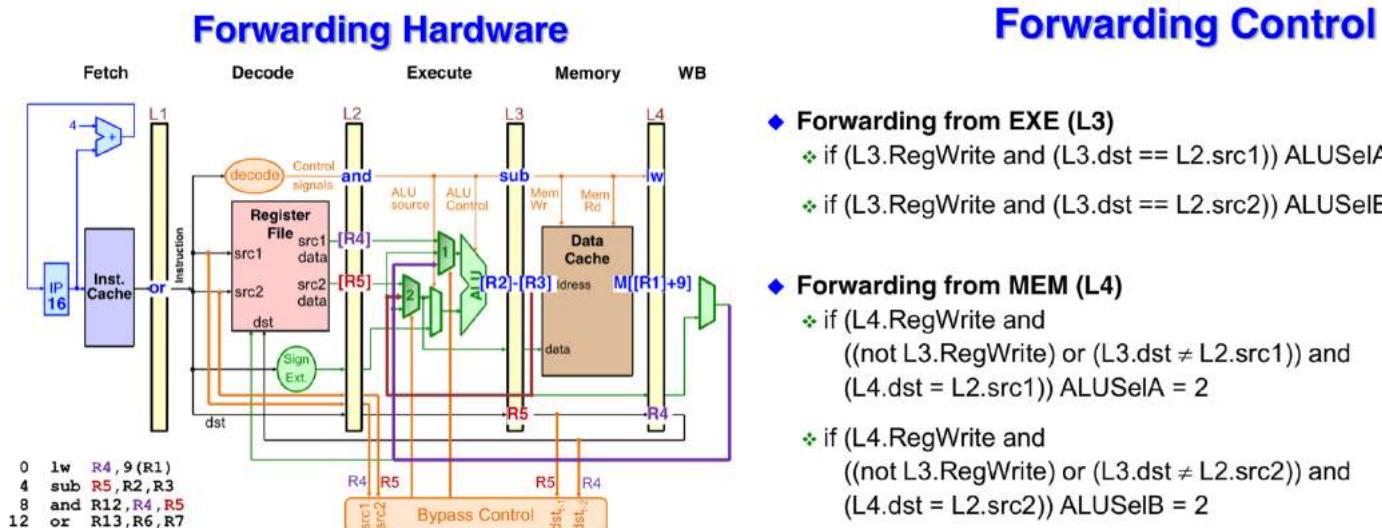
כאמור בשבייל לבצע Forwarding צריך קודם כל ליהות את ההתגשות, כלומר את זה שהערך שנקרה מהרגיטרים אינו ערך הנכון. לשם כך נתקין במעבד לוגיקה שתחזקה שמקורו של פקודה זהה ליעד של פקודה אחרת לפניה ובעצם תדע להעביר את מה שהולך להיכתב משלב WB או ה-MEM אל שלב EXE. נשים לב שהעברת הערך אינה בשלב הקירה Decode, אלא תמיד בשלב הביצוע - כמובן אנחנו יודעים כבר בשלב Decode שהפקודה בו תקרא ערך לא נכון מה-RF ולא מונעים את זה אלא פשוט מספקים לה בשלב EXE את הערך הנכון במקום הערך שנקרה ב-Decod.

החומרה שמחזקה את הקונפליקט תבודוק שוווין בין src ל-dest בשתי פקודות. בנוסף לחברים לאופרנדים של שלב ALU מוקסים (Mux) שמאפשרים לקרוא ערך שהועבר משלב אחר במקום לקרוא מוקבץ הרגיטרים. יש לוatu Mux שלוש אפשרויות:

1. להשתמש בערך שנקרה מהרגיטר פיאט בשלב Decode (הערך מתקדם עם הפקודה משלב Decode).
2. להשתמש בערך של הפקודה שהיא מהזר אחד לפני הנטווחית (העברה משלב Memory).
3. להשתמש בערך של הפקודה שהיא שני מוחזרים לפני הנטווחית (העברה משלב WB).

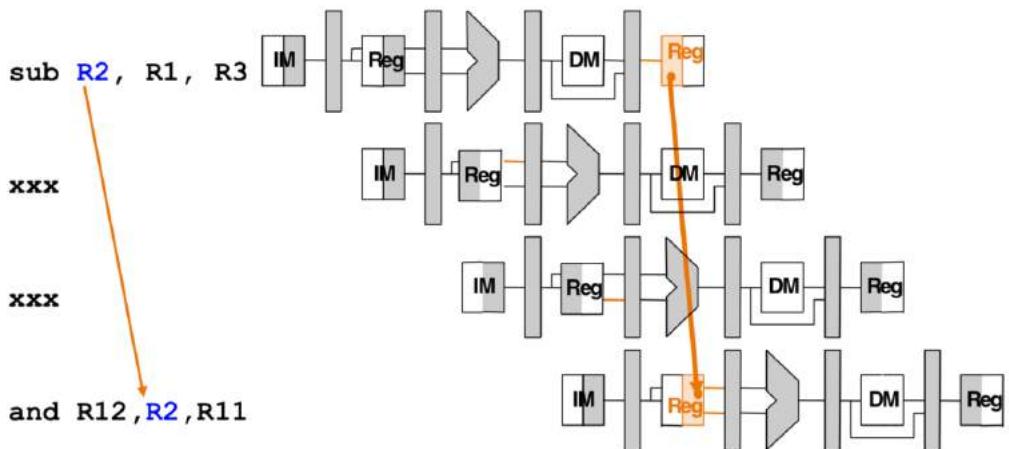
הערות:

- אם שתי פקודות כותבות ל-5, אחת משלב MEM וחתמת מ-WB, באותו מזמן בו פקודה אחרת צריכה אותו, אז צריך לחת את הערך של הפקודה הכי עדכנית (בשלב מוקדם יותר בפייפ, כי היא מופיעה מאוחר יותר בתוכנית). לכן כשחומרת Bypass רואה שמקורו קלשו שווה ליותר מחד היעדים היא בוחרת את הערך הכי עדכני, כלומר זה שחווש בפקודה הכי קרובה לפקודה הנטווחית.
- הערה חשובה: המוקסים עולים לנו, הם לא "בחינם", בעצם צריך באותו מזמן להפסיק לא רק את ה-ALU, אלא גם את החישוב של זיהוי התלוויות והעברתה, זה בהחלט יכול להשפיע לרעה על זמן המזמן.



בשיקף רואים את הלוגיקה של החומרה של bypass, בפרט רואים שני סיגנלי בקרה למוקדים שונים המוקם איזה רגל לבחורה: את הערך שקורא מקובץ הרגיסטרים, מה שהוחשב בפקודה לפני, או מה שהוחשב שניים לפני. רואים פה את לאציגים L1, L2, L3, L4 עבור השלבים DE, MEM, EXE, WB בתאמה. התנאי if(L3.RegWrite) בודק את אותן הפקודה Register File בלאז' השישי L3 ואם אותן דלוק זה אומר שהפקודה בלאז' 3 היא פוקודה שכותבת לRF. בעצם הקוד של התוכנה מלמד את החומרה שליטה על המוקם מתי הוא צריך לבחורה את רגל מס' 2 שלו כך: אם הפקודה שנעולה כרגע ב-L3 כותבת לRF וגם רגיסטר היעד שלו שווה לרגיסטר-מקור של פוקודה ב-L2 אז צריך לבחורה את רגל מס' 1. באופן דומה אם מה שיש ב-L1 שווה למה שיש ביעד של L3 אז צריך לבחורה את הרגל השנייה. אחרת קוראים כרגע מרגל 0. באופן סימטרי גם עבור המוקם השני אם המוקם ב-L2 שווה לעיד של L3 אז צריך לבחורה את רגל מס' 1 וכו'.

נשים לב שגם פוקודה לא כותבת לRF הערך של רגליים 1 ו-2 במוקם הוא זבל זהה בסדר כי לא צריך לקרוא אותו. נשים לב גם לתנאי שדווגע לכך שגם כתיבות נעשות במקביל לאותו רגיסטר אז בוחרים בערך מהשלב הקודם (מקורם) יותר - הרגל השנייה ולא הראשונה. עד כה דיברנו על העברת ערך מהלאציגים L-L2 או מ L4-L2.



אמרנו שהפקודה AND הבאה היא גבולית - היא קוראת מה RF באותו מחזור בו הפקודה שמספקת את הערך לרגיסטר כותבת אליו. האם הספיקה או לא הספיקה לקרוא את הערך הנוכחי? מה שעשו ב-MIPS היה לתוכן את המעבד כך שבמקרים כאלה הפקודה שקוראת תמיד תספק לקבל את הערך המעודכן. כיצד? ע"י "חציה" (splitting) של מחזור השעון: מחלקים את מחזור השעון לשני חלקים - חלק ראשון שנקרא HIGH, ובו מתבצעות כתיבות לרגיסטרים, וחלק שני שנקרא LOW, בו מתבצעות קריאות מרגיסטרים. לכן הכתיבה הנ"ל מבוצעת בשלב 5 HIGH שלה והקריאה של ה-AND נועשית בשלב 2 LOW שלה - וכך למרות שהכתיבה בוצעה באותו מחזור כמו הקריאה, מכיוון שהכתיבה מבוצעת בחצי ראשון של המחזור והקריאה בחצי השני של המחזור - בפועל הכתיבה קודמת לקריאה ולכן המצב הגבולי שהוא נכון.

**סיכום בניינימ:** אז עד כה ראיינו ציירנו וחילקנו אותו ל-FETCH, DECODE, EXECUTE, MEMORY, WRITE-BACK. ראיינו תלויות בין פקודות וディיברנו בפרט על RAW - מצב בו פוקודה אחת קוראת מרגיסטר-מקור אחריו שפקודה אחרת מחשבת ערך חדש לווטו מקור, כך שבאמצע פוקודה קוראת מה RF ערך שאינו עדכני. لكن דיברנו

על Bypass - זיהוי מצב שהיעד של פקודה שנמצאת בצינור שווה למקור של פקודה שנמצאת בשלב מוקדם יותר וביצוע מעבר של הערך שחוושב.

## Stall If Cannot Forward

### ◆ Load word can still causes a hazard:

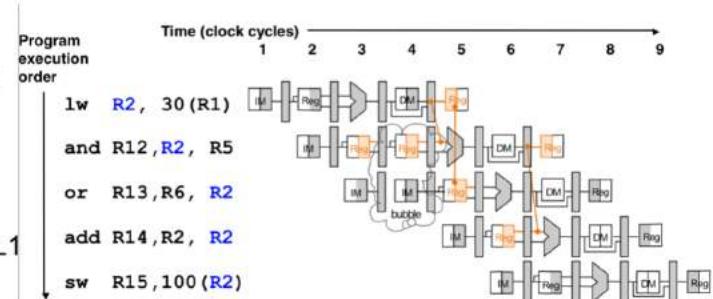
- ❖ an instruction tries to read a register following a load instruction that writes to the same register  
( $L2.dst == L1.src1$ ) or ( $L2.dst == L1.src2$ ) then stall

### ◆ De-assert the enable to the L1 latch, and to the IP

- ❖ The dependent instruction (and) stays another cycle in L1

### ◆ Issue a NOP into the L2 latch (instead of the stalled inst.)

### ◆ Allow the stalling instruction (lw) to move on



יש מצבים בהם לא ניתן להשתמש ב bypass וחייבים לבצע stall של הפיפי: למשל אם היוו מחליפים את הפקודה האריתמטית, שמחשבת את הערך לרגיסטר בשלב EXE שלו, בפקודה שטוענת אל אותו רגיסטר ערך מהזיכרון, למשל LW, עברוה רק בסוף שלב WB ידוע הערך שרוצים שיהיה ברגיסטר.פה אין מצב bypass: עם החישור זה עבד כי אין שהיא סימנה את ה EXE הערך היה ידוע, אבל במקרה של LW לא ניתן לבצע מעקר ופקודה שנמצאת בשלב EXE מייד אחרת וצריכה את הערך זהה לצורך לחכות באותו שלב - لكن אין ברירה אלא החומרה צריכה להוסיף stall.

כנ"ל במקרה שהחישוב של הערך לכתיבת EXE היה לוקח הרבה מוחזורי: יש פקודות בהן בשלב הביצוע, Execute' לוקח יותר מוחזור אחד, למשל בפקודה המורכבת div. גם במקרה זה נוצרה stall על מנת לבצע חישוב שצרכות את התוצאה כי היא עדין לא מוכנה והפקודה שהגיעה מוחזור אחרינו חייבת לחכות - כי לא סימנו לחשב בכלל וכןן אי אפשר להעביר ערך. לשם כך מוסיפים למעבד חומרה שמצויה את המצב בו המקור מגע מפקודה שעדיין לא מוכנה וועשה stall.

במקרה קריאה מהזיכרון stall יהיה במוחזור שנון אחד כדי לאפשר ל LW לתקדם עוד מוחזור, שם היא מקבלת את הערך ואז יהיה ניתן לבצע bypass בשלב Execute ולקבל את הערך. שורה תחתונה: לעיתים אנחנו רואים שהערך כבר מוכן ואז ניתן להעביר אותו ולפעמים הערך אינו מוכן ואז עושים stall.

## Software Scheduling to Avoid Load Hazards

Example: code for (assume all variables are in memory):

|       | <u>Slow code</u> | <u>Fast code</u> |
|-------|------------------|------------------|
|       | LW Rb,b          | LW Rb,b          |
|       | LW Rc,c          | LW Rc,c          |
| Stall | ADD Ra,Rb,Rc     | LW Re,e          |
|       | SW a,Ra          | ADD Ra,Rb,Rc     |
|       | LW Re,e          | LW Rf,f          |
|       | LW Rf,f          | SW a,Ra          |
| Stall | SUB Rd,Re,Rf     | SUB Rd,Re,Rf     |
|       | SW d,Rd          | SW d,Rd          |

Instruction order can be changed as long as the correctness is kept

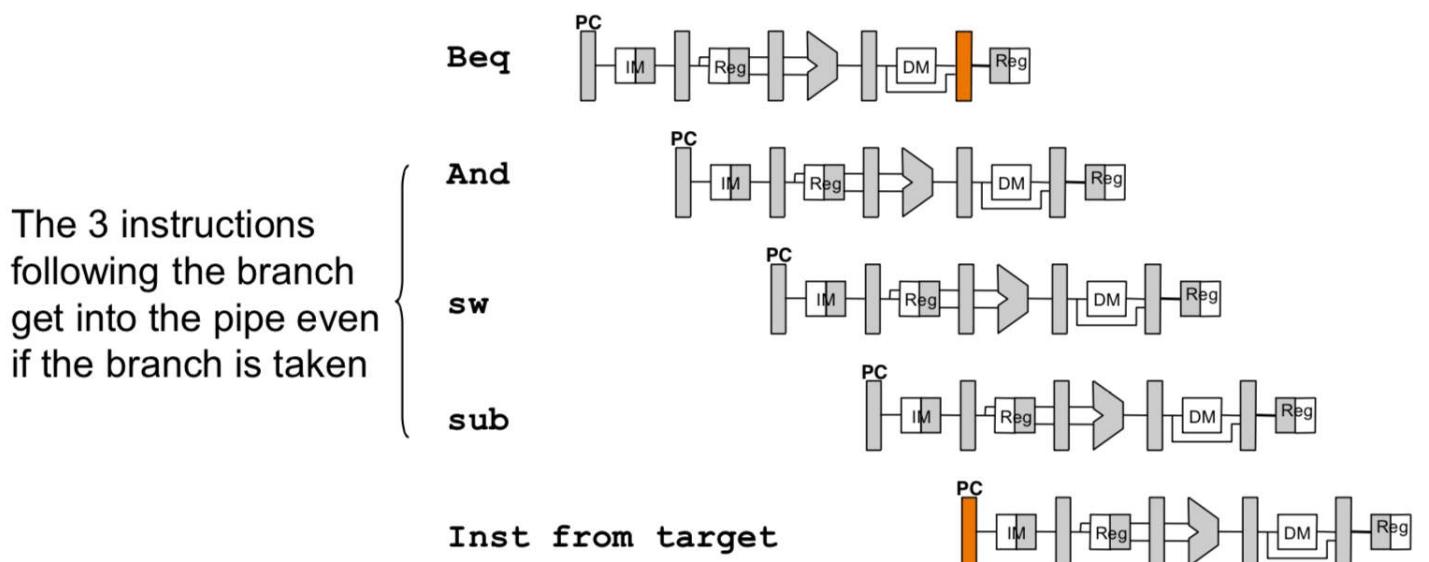
לעתים המתכנת יכול להביא בחשבון את המיקרו-ארכ' בשביל לשפר ביצועים. בהירה: הארכ' תמיד עובדת נכון, ולא מחייבת את המתכנת לעשות משהו בשビル לקבלת נכוןות. אם המתכנת לא מודיע איז החמרה חייבת לדעת לעשות את ה stall בעצמה, אבל אם המתכנת מודיע למיקרו-ארכ' איז הוא יכול לשפר את הביצועים. למשל בתכנית פשוטה:

$$a = b + c;$$

$$d = e - f;$$

הנחה היא שכל המשתנים מצבים קבועים בזיכרון ולכן עברו כל פקודה צריך לעשות לו אל שני גיסטרים, לעשות ביניהם חיבור ואז לעשות sw. כמובן לכל פקודה אריתמטית צריך לעשות stall פעמיים מראש מחזר אחד. לכן הצעה היא לשנות את סדר תכנית האסמלבי (בלי לשנות את הסמנטיקה שלה) כך שתני ה-w של הפקודה השנייה אפשר לבצע מיד אחרי שני ה-w של הראונה, במקום פקודת החיבור, וכך ליצור את הריווח הדרוש בין החיבור המקורי לטיענת הערכים m-b ו-c ע"י לו. כתע כשtagיע הפקודה ADD כבר יהיה אפשר לעשות bypass ובינתיים ביצנו פקודת אחרת ולא חיכינו בחוסר-מעשה. בימינו, בשビル לאפשרות למתכנת/מתכנן-הקומפונילר לכתוב תכניות שהביצועים שלهن יהיו גבויים, חברות של מעבדים מודרניים אופטימיזיות ואפשר לראות בהם פירוטים מאוד שימושיים על המיקרו-ארכ' של המעבד. זה מידע שלכורה היה ניתן לשמור בסוד, אבל הוא יכול לשפר את הביצועים ולכן היא חשופת אותו כהמלצות לכותבי קומפיילרים וכותבי אפליקציות Real-time. חשוב להבהיר שככל מקרה המיקרו-ארכ' ידע לשתול את ה stall ואין מצב שהתכנית תקרוס או תשמש בערכים לא נכוןים אבל אפשר לעזור לה.

## Control Hazards



cut נדבר על control hazard - נושא הקשור לкопיצות (פקודות jump ופקודות branch).

נניח לדוג' שיש תכנית עם פקודת קופיצה מותנית שבפועל לעיתים קופצת ולפעמים לא. אם יש קופיצה איז היא כתובות 48 בתכנית, אחרת ממשיכים להלאה לביוצו הפקודה בכתובות שאחרי פקודת הקופיצה. נשים לב שבמקרה זה לא ניתן לדעת מראש מהן הפקודות שנכנסות ל-Pipe אחרי הקופיצה.

ב-MIPS רק בסוף שלב exe יודעים אם פקודת ג'אמפ קופצת או לא. בשלב Fetch, בו הבנוו את הבטים של הפקודה, אפילו לא יודעים עדין שזויה פקודת קופיצה - لكن ברור שנמשיך לתקדם ושהפקודות העוקבות לה בזיכרון הפקודות הן אלה שיכנסו ל-Pipe. בשלב הפענוח נדע שזו jump אבל עדין לא נדע האם קופצים או לא.

נניח שנמשיך להביא פקודות לפי האפשר' שהוא לא קופץ ונניח שבפעם אחת גילינו אחרי EXE שצריך היה לkopoz כולם ביצענו חיזו-שגוי ולא היינו צריכים להביא את הפוקודות העוקבות לפוקודת הקפיצה, אלא לкопז. במקרה זה כל הפוקודות שהופיעו להיכנס אחורי פוקודת הקפיצה הן פוקודות לא טובות שלא היה צריך להתחילה את ביצוען. המצב הזה, שבו יש סכנה שיתבצעו פוקודות שבדייעבד יתברר שלא צרכות להתבצע, נקרא "control hazard". גם במצב של ג'אמפ שאינו מותנה יש בזבוז - הג'אמפ מזוהה בשלב Decode ולכן אחורי הוכנסה פוקודה אחת שלא הייתה צריכה - כלומר במקרה של קפיצה לא-מותנית יש לרוקן את הפוקודה בשלב Fetch. אז המצב בו מביאים פוקודות לא נכונות בעקבות פוקודות הסתעפות נקרא Control Hazard.

הפתרון הטריוויאלי Control Hazard הוא לעשות stall בכל פעם שרואים קפיצה: נחכה עד שנדע האם קופצים ורק אז נמשיך. זה פתרון לא טוב כי מסתבר שיש הרבה מאוד פוקודות בתכנית - בערך 1 ל-5. נניח שמעבדubo רוט=CPI=1 מבצע תכנית בה אחת מכל חמישה פוקודות היא פוקודת קפיצה ונניח שבכל קפיצה המעבד עוצר את הכנסת הפוקודות לפיפ' למישר לשולחה מחזוריים, עד שהקפיצה תסימן את exe וידוע מאיפה להביא את הפוקודות. במקרה זה כל חמישה פוקודות מבוצעים עוד לשולחה מחזורי שעון, מה שאומר שעכשו 8 מחזוריים לביצוע 5 פוקודות כלומר  $CPI = 8/5 = 1.6$ . זו הרעה של 60% בביצועים - וזה חמור!

פתרון אחר ל-hazard, שאינו מופיע בשקפים כי הוא כבר די מירשן, הוא לנסות לסדר את הפוקודות בתכנית כך שאחרי קפיצה יכנסו פוקודות שבוטוח צרכיות להתבצע - זה נקרא delayed jump. זה דורש מהקומפיילר להצליח למצוא את הפוקודות האלה ולהיות מודע למיקרו-ארכ': עומק החינוך, متى יודעים האם הוכנסו פוקודות לא נכונות ועוד - لكن שימוש ב-delayed jump היה פתרון רלוונטי בימים בהם קשר יותר הדוק בין הקומפיילר למיקרו-ארכ'.

פתרון נוסף זה taken/predict, הוא עדין נעשה במעבדים מאוד פשוטים והוא בעצם מתאר את מה שראינו בדוגמה הראשונה בה המשכנו להביא פוקודות עוקבות לקפיצה ורוקנו אותן במידת הצורך (כלומר אם הקפיצה התחבצה): נמשיך להביא את הפוקודות שאחרי הקפיצה, כאשרו שהוא לא תילך. בהסתברות מסוימת נניה צדקה, הקפיצה לא תילך ואז הפוקודות שהצליחו להיכנס ל-pipe. במקרה שקפיצה תילך נגלה זאת בשלב exe ואז נשלם את העונש של הכנסת פוקודות לא נכונות - ביצוע ריקון (Flush) ל-pipe: מחיקה של הפוקודות שנכנסו לאחר הג'אמפ. נשים לב ש מבחינות נכונות התכנית (נכונות מצב הזיכרון) זה בסדר למחוק אותן כשהקפיצה נמצאת בשלב EXE כי בודדות עד לא שייננו את מצב הזיכרון - אף פוקודה לא כתבה לרוגיסטר בשלב WB או כתבה לזכרון ב-MEM. אז מבחינת הזיכרון וה-RegFile זה-caillo הפוקודות אחורי הקפיצה עדין לא קרוא. במקרה של Predict Not Taken הבחירה בדוגמה הקודמת, בהנחה שהצ'מי מהפעמים הקפיצה לא תילך, ירדו ל-1.3 בלבד:

$$CPI = CPI_{ideal} + P(branch) * P(Taken) * Penalty = 1 + 0.2 * 0.5 * 3 = 1.3$$

בנוסף ביצוע ריקון ממשכו שתמיד בזבזו הספק של ביצוע השלבים המוקדמים בפוקודות שהוצענו.

טכנית flush נועשה ע"י ריסט לפלייפלופים שהופכים את כל הפוקודות הרלוונטיות לאפסים ואז המעבד לא עושה עבורן כלום.

דרך נוספת להתמודד עם Control Hazard שלא כ"כ משתמשים בה נקראת eager fetch בה מביאים פוקודות מושנים המסלולים: גם אלה שיכנסו אם הקפיצה תילך וגם אלה שלא. הבעיה היא שמסלול אחד תמיד מבוחץ ולכן תמיד יהיה בזבוז של כוח חישוב.

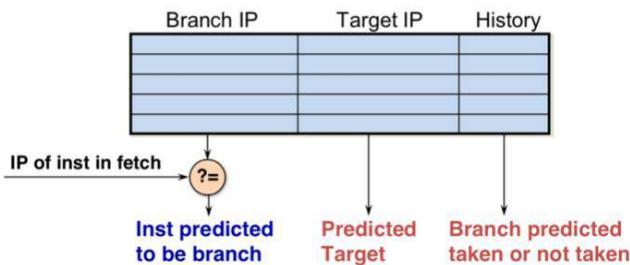
כעת נראה את הפתרון הנוכחי ל-control Hazards שבירינו מצליח לדאוג שכמה תמיד נביא פקודות מהמקום הנוכחי, בין אם הקפיצה נלקחת או לא.

## חיזוי קפיצות - Branch Prediction

### Dynamic Branch Prediction

- ◆ Add a **Branch Target Buffer (BTB)** that predicts (at fetch)

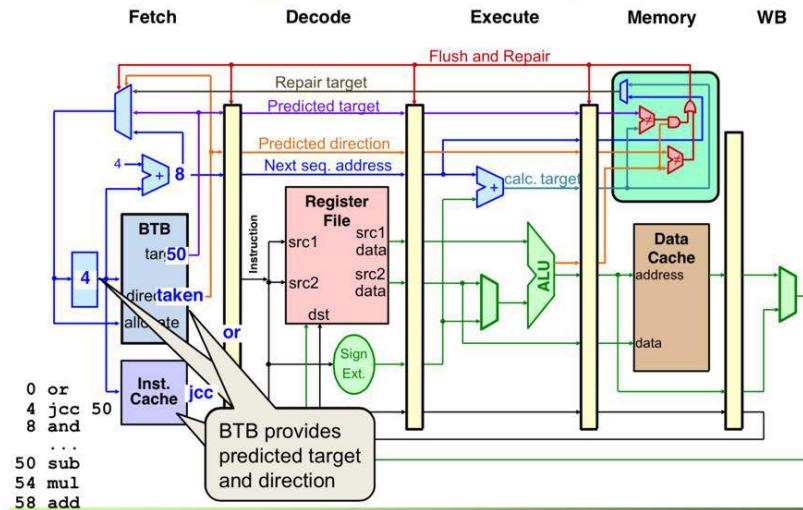
- ❖ Instruction is a branch
- ❖ Branch taken / not-taken
- ❖ Taken branch target



- ◆ BTB allocated at execute – after all branch info is known

- ◆ BTB is looked up at instruction fetch

### Adding a BTB to the Pipeline



הפתרון הנוכחי ל-control hazards הוא Branch prediction (Structural hazards)

ונסה לנחש על סמך העבר האם ולאן הבראנץ' י קופז. זה עובד כי בדרך כלל פקודה קפיצה תילקה/לא-תילקה כמה פעמים ברצף (תחשבו על לולאה ש קופצת 100). פעמים ורך בפעם האחרון לא קופצת לשם כך מוסיפים אלמנט חומרה שנקרא BTB (Branch Target Buffer). BTB במעבדים מודרניים מספק חיזוי נכון בהסתברות של לפחות 95% ובפועל אף יותר. בפועל BTB זו טבלה המכילה מידע לגבי פקודות הקפיצה בתכנית שכבר ראיינו.

בפרט כתוב בה את:

1. כתובות פקודות הקפיצה.
2. כתובות הקפיצה (היעד של הקפיצה).
3. היסטוריה כלשהי שמתארת האם הפקודה קופזה או לא.
4. בהתאם להיסטוריה כלשי BTB מוחזר חיזוי (prediction) האם הקפיצה תילקה או לא.

#### ◆ Allocation

- ❖ Allocate instructions identified as branches (after decode)
  - Both conditional and unconditional branches are allocated
- ❖ Not taken branches need not be allocated
  - BTB miss implicitly predicts not-taken

#### ◆ Prediction

- ❖ BTB lookup is done parallel to IC lookup
- ❖ BTB provides
  - Indication that the instruction is a branch (BTB hits)
  - Branch predicted target
  - Branch predicted direction
  - Branch predicted type (e.g., conditional, unconditional)

#### ◆ Update (when branch outcome is known)

- ❖ Branch target
- ❖ Branch history (taken / not-taken)

#### ◆ Wrong prediction

- ❖ Predict not-taken, actual taken
- ❖ Predict taken, actual not-taken, or actual taken but wrong target

#### ◆ In case of wrong prediction – flush the pipeline

- ❖ Reset latches (same as making all instructions to be NOPs)
- ❖ Select the PC source to be from the correct path
  - Need get the fall-through with the branch
- ❖ Start fetching instruction from correct path

#### ◆ Assuming P% correct prediction rate

- CPI new =  $1 + (0.2 \times (1-P)) \times 3$
- For example, if P=0.7
 
$$\text{CPI new} = 1 + (0.2 \times 0.3) \times 3 = 1.18$$

קריאה מהחזאי נעשית בשלב IF במקביל לקריאה מה-Instruction Cache שם כתובות הפקודה שמובאות מרשווית עם כתובות הפקודות שבטבלה ואם היא נמצאת שם אז זה אומר שהוא פקודת קפיצה - נשים לב שהחזאי מבין זאת עוד לפני שלב Decode ובירט יכול לשנות על הפקודה שתובא ל-Pipe מיד אחרי פקודת הקפיצה. כמובן כבר במחזור הראשון של הקפיצה החזאי יכול לספק לנו השערה האם להביא פקודות מייד הקפיצה או פקודות עוקבות לkapיצה.

העדרון של BTB עם היסטוריית הקפיצה (כלומר עדכון האם הפעם האחורונה נלקחה לא נלקחה) געשה רק כשהচכל ידוע לגבי הקפיצה - אחרי שידועים שהפקודה היא קפיצה, האם היא קופצת ולאן.

דוגמא: בפעם הראשונה בתוכנית שנראה קפיצה - בטוח לא נמצא אותה ב-BTB ואז געשה בפועל predict not taken - אבל אחרי שהפקודה הגיעו לשלב הביצוע מיד על גביה ייכתב BTB-IP והוא פעם הבא שנראה כתובות זאת נדע שהיא קפיצה ונדע לנחש יותר טוב איזה פקודות יש להכוnis בעקבותיה לצינור. כמובן, בתחילת ריצת התוכנית, לפני שפגשנו איזושרי פקודת קפיצה, החזאי היה ריק ולא יעזר לנו, וכל עד לא נכתב אליו מידע על הגאים פים אז לא יוכל להשתמש בו ונעשה predict not take.

אחרי זמן מה, כשנעשה בשלב Fetch-Instruction Cache במקביל ב-BTB, BTB ישווה את הכתובת ה-IP הנוכחית לכתובות בטבלה ואם הוא כבר נתקל בעבר בכתובת זו זה יהווה זיהוי שלה כפקודת קפיצה וה-BTB יכול להגיד לנו מהו עלייה. למשל אם ה-BTB ימצא את הכתובת בטבלה הוא יוכל להגיד לנו שככתובת זו באמת יש ג'אמפ, פעם קודמת הוא נלקח, ובעקבותיו עברנו לכתובת מסוימת, למשל 50. לכן יוכל, במידה והחזאי משער שהקפיצה תילקח שוב, להתחיל לחתך פקודות מכתובת 50. נשים לב שזה קורה לפני שפענחנו את הפקודה בכלל כך שאם החזוי הוא נכון אין צורך של מחזרי שעון כלל.

בשביל לתמוך בעדרון החזאי יחד עם פקודת הקפיצה נعتبر בלאצ'ים גם את המידע הבא על החזוי: האם חזינו קפיצה (not taken) או לא (taken) ומה הכתובת שצרכי להביא ממנה פקודות במקרה שהחזוי שלנו שגוי. כשהשיג'אמפ יסימם את ה-exe או לא (גם מבחינת הכוון וגם מבחינת יעד הקפיצה) אם החזוי היה נכון אז יש קומפרוטורים שימושיים ובודקים האם החזאי צדק או לא (גם מבחינת הכוון וגם מבחינת יעד הקפיצה) אם החזוי היה לא נכון אז ונעדכן את ההיסטוריה של הקפיצה ב-BTB בכך שהוא אכן קפיצה. אם לעומת זאת טעינו בחזוי (כוון או יעד) אז פקודות שנכנסו לא טובות ובנוסף לעדרון ההיסטוריה ואולי עדכון יעד הקפיצה - צריך לעשות ריקון ל-Pipeline לשולח את הכתובת המתוקנת אל ה-IP ולהתחל לחייב פקודות מהמקום הנוכחי.

החזוי עצמו נעשה בשלב fetch: ממש בהתחלה, במקביל לפניה ל-Instruction Cache, אפונים לחזאי כדי שיניחש עבורנו שזו קפיצה, האם תקין ולאן. שלב החספוץ verification-, שם משווים את מה שבאמת קרה לניחס שלנו, ושלב update, בו מעדכנים את החזאי במה שקרה, נעשה בסוף שלב exe, לאחר שידועים שהפקודה קפיצה ולאן. כאמור, במקרה של חזוי לא נכון נצטרך לעדכן את החזאי, לעשות פלאש ל-Pipeline ולהביא פקודות חדשות מהמקום הנוכחי.

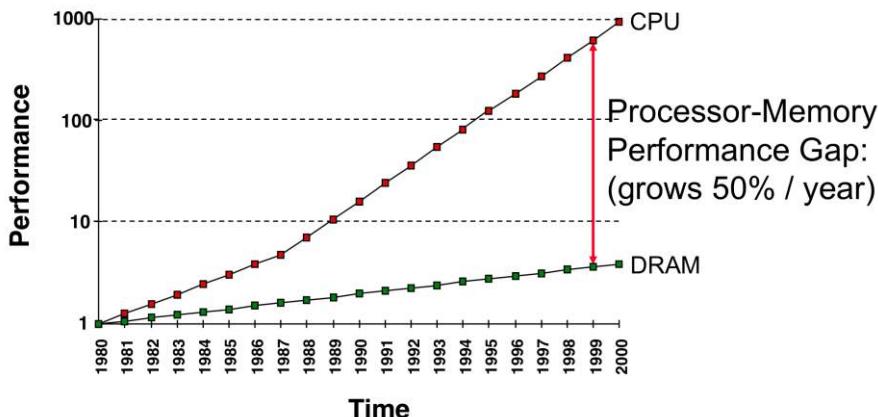
הערה: רוב הארכ' תומכות בJump Indirect: קפיצה לפי ערך שנקרא מרגיסטר או מהזיכרון, ככלומר כתובות הקפיצה לא מופיעה במפורש בפקודה. זה משמש למשל למימוש switch statement ב-C. זה גם מתאים באופן טבעי לקוד שהוא OOP - עבור קוד זהה חיברים ללמידה לא רק האם הוא קובץ אלא גם שם.

סיימנו עם מעבד הצעצוע MIPS. מעתה מהלאה נלמד איך עובדים דברים במעבדים מודרניים.

## הרצאה מס' 3: זיכרון מטמון (Cache)

היום נלמד על זיכרון המטמון (קash, Cache) ועל שיטות-ארגון שלו.

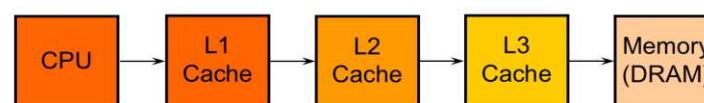
### מוטיבציה לשימוש בזכרון מטמון



הגרף מראה לנו את השיפור בביוצוי-המעבדים לאורך השנים לעומת זיכרון בביוצוי-הזיכרון: הזיכרון הוא התקן יחסית פשוט, שמורכב בדרך כלל ממערך של טרנזיסטורים לצור ערך מסוים - 0 או 1. כפי שראינו בהרצאה הראשונה כשדיברנו על חוק מור, הטכנולוגיה מצליחה ליצור טרנזיסטורים יותר מהירים בכל שניםתיים - קראנו לגדל הטרנזיסטור "טכנולוגיית התהילה". בעקבות התפתחותנו אנחנו יכולים לראות את התפתחות הזיכרונות במהלך 20 שנה ובעקבות העליינה את התפתחות המעבדים. אז הזיכרונות משתפרים - אבל רק קצת איטי. לעומת זאת, גם המעבד משתפר עם הזמן, והוא מושפע גם מטכנולוגיית התהילה (כמו הזיכרון) אבל גם משיפורים אחרים כמו ארכ' ומיקרו-ארכ': Branch Target Buffer, Out of Order Execution וכו'. נוצרת בעיה: יש לנו מערכת שרכיב אחד (המעבד) משתפר בה קצת גדול יותר מרכיב אחר (הזיכרון). כשהתכוון רצה יש בה פקודות כמו Add, Branch, Load, Load-Load. כמשמעותם ל-Load ניגשים לזכרון להביא שם נתון ואז משתמשים באותו ערך שהבנו מהזיכרון. כל פעם שהוא יש את ה Latency של הזיכרון - אבל אם המעבד נהיה מהיר והזיכרון לא כ"כ סימן שככל שהזמן מתקדם נחכח באופן יחסית יותר לזכרון, ייווצר צוואר-בקבוק כך שיתחיל למנוע מהביבוצים להשתפר.

## Memory Trade-Offs

- ◆ Large (dense) memories are slow
- ◆ Fast memories are small, expensive and consume high power
- ◆ Goal: give the processor a feeling that it has a memory which is large (dense), fast, consumes low power, and cheap
- ◆ Solution: a Hierarchy of memories



|        |          |         |
|--------|----------|---------|
| Speed: | Fastest  | Slowest |
| Size:  | Smallest | Biggest |
| Cost:  | Highest  | Lowest  |
| Power: | Highest  | Lowest  |

از הבעה שאנו רוצים לפתור היא היזכרות של **bottle neck** בגישה ל זיכרון. בשביל זה משתמשים ברגען של זיכרון-מטמון או **Cache Memory**, שהוא בפועל אומר לנו מארגנים זיכרונות שעובדים במבנהו שונה בזורה היררכית כך שננותים שאנו משתמשים בהם לעתים הקרובות ימצא בהיררכיה זיכרון מהירה, ונותן שימושים בו לעיתים רחוקות יוכל להיות בזיכרון האיטי. לכן בערך ה-CPU ישבת מערכת של קאשים בשם L2, LLC(L1), L3 (MLC). ככל שמתקרבים למעבד גודל הקאש וכן זמן הגישה אליו הולכים וקטנים - למשל L1 מכיל הכי מעט נתונים אבל הוא מאד מהיר.

לדוגמא: אם אנחנו רוצים לפנות לזכרון הראשי (RAM) צרכיים להמתין כ- 400 ממחוזרים עד שהנתון יגיע. לעומת זאת נדרשים 40 ממחוזרי המתנה, L2 בערך 12 ממחוזרים ול-L1 בין אחד לארכעה ממחוזרים. מבחינה כמה נתונים אפשר לשמר: בזיכרון הראשי אפשר לשמר 8GB, ב-L3 יש בסביבות 4MB, ב-L2 כ- 256k ובל-1 בערך 32KB. ככל רמה זמן הגישה קטנה ואותו גם הנפח - ככל שהזיכרון יותר גבוה בהיררכיה הוא יותר מהיר ויוטר קטן, המחיר שלו יותר גבוה, והוא צורך יותר הספק. אנחנו רוצים שברוב המקרים (בימינו זה קורה ביוטר מ- 95%) בהם המעבד יפנה לזכרון הוא ירגיש שהוא קיבל את מה שביקש במחוזר אחד, כלומר ימצא אותו ב-L1. זה יאפשר את הבעה שקצב ההתקדמות של הזיכרונות הרבה יותר איטי מקצב ההתקדמות של המעבדים. זה הרעיון של היררכיה זיכרון.

הערה: יש גישה לזכרון כדי להביא נתונים וגישה לזכרון לצורך הבאת קוד. לכן בדרך כלל זה יחולק לשני זיכרונות מטמון נפרדים, דאטא (L1 Data Cache) וקוד (L1 Instruction Cache) אולם היום שנדבר על קאש נתרכז באינטראקציות בין היררכיות שונות ככל מר\_ntiyics למטרות שנותן נתרכז בhirerccies שנותן ועובדים אחד מול השני - למשל L1 מול L2 או L3 מול זיכרון הראשי.

כאמור נרצה להגיע לנצח בו כל בקשה תמצא בקאש מהירות ביוטר, L1, בסביבות 95% ואף הרבה יותר מכ- 32KB בלבד בו בזמן שתכניות מודרניות צורכות הרבה יותר מקום. נוכל לקיים זאת בגלל שני עקרונות:

1. עקרון המוקומיות בזמן.
2. עקרון המוקומיות למרחב.

1. עקרון המוקומיות בזמן אומר שאם ניגשתי לנตอน בזמן כלשהו או סביר להניח שבזמן הקרוב ניגש אליו שוב. למשל אם עשינו `load` של ערך כלשהו קרוב לוודאי שנצטרכרשוב את אותה כתובות בקרוב. מה? כי נראה שהמשך נתקל בפקודת `branch` שחזרה באותו המיקום, למשל בימיוש הפקודה (`while (*ptr > 0)`). ניגשים שוב ושוב ל-`ptr`. גם בהקשר של פקודות: רוב הפקודות בתכנית מבוצעות בתוך לולה וכנראה שניגש כמה פעמים אותה הפקודה לאחר שקרה לנו אחת מהזיכרונות אל ה-`instruction cache`.

2. עקרון המוקומיות למרחב אומר שאם ניגשנו לאיזשהו אזור בזכרון, נגיד לכתובת 8100, אז נראה שבקarov ניגש שוב לכתובת שכנה לה, למשל אם ניגשנו לאינדקס כלשהו במערך נראה שבקarov ניגש לאינדקסים קרובים אליו. עד דוג' לגישה לכתובת שכנה היא למשל שהפקודה בכתובת 2004 נראה תבוצע בסמור לביצוע הפקודה בכתובת 2000.

עקרונות המוקומיות בזמן/מרחב מלמדים אותנו שגם למשל היהת לי גישה לכתובת 2000 אז כדאי להביא את כל הנתונים שיושבים בכתובת 2000 וסביבתה ולשים אותם בקאש כי נראה ששוב ניגש אליהם בקרוב. בדומה אם ניגשנו לכתובת מסוימת קרוב לוודאי שנצטרכר את הכתובות השכנות שלה וכך מארש נשמר אותו בקאש. עקרונות אלה מספקים מוטיבציה לזכרון היררכי באופן זה שסביר שרוב הפעמים שניגש לזכרון הקטן וההדור נמצוא את מה שהמעבד חיפש. נשים לב שעקרון המוקומיות מתקשר גם לחוק אמדל של מדנו בהרצאה הראשונה שאומר שcadai תמיד לשפר את מה ששכח בתכנית (במקרה שלנו - גישה למקומות זרים/קרובים).

הבהרה: נגיד שעכשו תכנית עובדת על מערך - כموון שפעם הראונה בתכנית שני ניגש למערך לא יהיה את הנתון בזיכרון המטען ואצטרך לגשת לזכרון הראשי, להביא אותו שם לקאש, ומרגע זה יוכל לקבל אותו במהירות.

## מושגי יסוד

נדון עכשו בכמה מושגי-יסוד בנושא של היררכיות זיכרון. תמיד נטרכז בשתי רמות: הרמה התחתונה והרמה שמעליה.

## Memory Hierarchy: Terminology

### ♦ For each memory level define the following

- ❖ Hit: data available in the memory level
- ❖ Hit Rate: the fraction of accesses found in that level
- ❖ Hit Time: time from data request till data received when hitting in the memory level; includes also the time to determine hit/miss
- ❖ Miss: data not present in memory level
- ❖ Miss Rate = 1 – (Hit Rate)
- ❖ Miss Penalty: Time to replace a block in the upper level + Time to deliver the block the processor

### ♦ Average memory-access time =

$$\begin{aligned} t_{\text{effective}} &= (\text{Hit time} \times \text{Hit Rate}) + (\text{Miss Time} \times \text{Miss rate}) \\ &= (\text{Hit time} \times \text{Hit Rate}) + (\text{Miss Time} \times (1 - \text{Hit rate})) \end{aligned}$$

- ❖ If hit rate is close to 1,  $t_{\text{effective}}$  is close to Hit time

- מושג Hit אומר שנגשתי לרמה כלשהי, ביצעתו בה lookup לכתובת מסוימת וממצאי את הנתון שchipsetי. למשל אם חיפשנו את הפקודה שנמצאת בכתבota 2000 וממציאנו את ארבעת הבתים שלה בזיכרון המטען - קיבלנו hit, שם עשינו fetch והבנו את הפקודה. אם לא מצאנו אותה בקאש אז נקרה לגישה ברמה זו Cache Miss. אז hit כורה כשהדאטא נמצא בתוכה הרמה שאנו פונה אליה ו- Miss זה כשהוא לא ברמה זו.
- Hit Rate הוא מס' הפעמים בהן קיבלנו Hit Cache מתוך סך הגישות לזכרון. בפועל Hit Rate של 95% זה משחו מאד סביר. hit Rate Miss הוא כמובן hit Rate - 1-Hit Rate.
- Hit Time זה הזמן שליקח למעבד מהרגע שהוא מבקש את הדאטא עד שהדאטא מגיע חזירה וניתן לשימוש בו. תכף נלמד איך קאש בניו ונראה שבצם כשמעבד עורשה lookup, אם הוא מוצא אז הוא מוחזר אותו ואם לא הוא עבר לחפש בהיררכיה הבאה. המסלול של lookup, hit/Miss indication, read data read יכול נקרה hit time. במקרה זה לוקח בדר"כ 3-4 סיקלים: מחזור ייחיד בשביל למצוא את הנתון ב-L1 ועוד כמה בשביל להביאו אותו. אז hit time כולל את זמן החיפוש.
- Miss penalty זה העונש ששילמנו על כך שהנתון לא היה בזיכרון המטען: אם חיפוש ב-L1 לוקח מחזור ייחיד וhit-time של L2 לוקח 10 מחזוריים אז hit-time Miss penalty של L1 הוא 11 מחזוריים. בשערשים lookup ונניח שמקבלים hitMiss צריך ללקת לרמה התחתונה וגם בה לעשות lookup. ניתן גם לבצע lookup בכל הרמות במקביל כדי למזער את זמן החיפוש אבל בדר"כ זה לא כדאי כי זה מבזבז הספק וכאמור בסבירות גבואה מLOAD הזכור ימצא ב-L1. בנוסף אפשר לשים חוץ בין רמות הזיכרון השונות וכשנמצא את המידע נכניס אותו לקאש העליון ואז למעבד,

או לחופין קודם נשתמש במידע ואח"כ נdagג שהוא ייכתב לכאש - כי המסלול הクリティ הוא מהזכרן בו המידע נמצא בין המעבד שצריך אותו. כדי לעשות את החישוב המדוקדק של Miss-Penalty צריך להבין בדיקת מבנה המערכת - למשל האם הלוקאפים מתחכימים במקביל. בהנחה: הזמן שלקח לי לגלוות את ה-Miss הוא חלק מהתהילה הבאת הדאטא. Miss penalty הוא העונש שקיבלנו מהרגע שהתחלה לחפש את הדאטא ועד שקיבלנו אותו. למשל אם נניח שיש לנו עשרה פקודות load, בתשע מהן יש hit ו-*load* העשيري יש Miss שגורם לנו לפחות 12 מחזוריים. אם היה לנו 100% hit זמן הגישה הזיכרון הממוצע בתכנית היה מחזור אחד אבל בגלל שהוא זמן זה ייקח בחשבון את ה-Miss penalty. נראה את הנוסחה לזמן הגישה לזכרון הממוצע:

חשוב להבין שהגדotta ה-Miss תלוי בהקונFIGורציה של המערכת - למשל אפשר לבנות מערכת שתකצר אותו על חישובו של הספק (אולי ע"י ביצוע lookup בכל הרמות במקביל).

## • EMAT •

## Effective Memory Access Time

### ◆ Cache – holds a subset of the memory

- ❖ Hopefully – the subset being used now

### ◆ Effective memory access time

- $t_{\text{effective}} = (t_{\text{cache}} \times \text{Hit Rate}) + (t_{\text{mem}} \times (1 - \text{Hit rate}))$
- $t_{\text{mem}}$  includes the time it takes to detect a cache miss

### ◆ Example

- ❖ Assume  $t_{\text{cache}} = 10 \text{ nsec}$  and  $t_{\text{mem}} = 100 \text{ nsec}$

| Hit Rate | $t_{\text{eff.}} (\text{nsec})$ |
|----------|---------------------------------|
| 0        | 100                             |
| 50       | 55                              |
| 90       | 20                              |
| 99       | 10.9                            |
| 99.9     | 10.1                            |

- ❖  $t_{\text{mem}}/t_{\text{cache}}$  goes up  $\Rightarrow$  more important that hit-rate closer to 1

רוצים זמן הגישה האפקטיבי לזכרון יהיה ברוב המקרים מינימלי כולם שמדובר שמדובר ירגיש שהוא פונה לזכרון ומתקבל את המידע בזמן שלוקח לגשת לכאש בrama הכי גבוהה. הקאש מחזיק רק תחת-קבוצה של זיכרון, כMOVED שלא את כלו, והמטרה היא שיחזיק כמה שיותר זיכרון בו משתמשים ברגע נתון. אז זמן הגישה האפקטיבי הוא המכפלה בין זמן הגישה לקאש ואוחז הצלחות-הגישה. נניח שאין בכלל hit - אז זמן הגישה האפקטיבי יהיה זמן הגישה למיטמן. אם לעומת זאת היה 100% hits אז זמן הגישה האפקטיבי יהיה רק זמן הגישה לקאש. ככל שהיחס בין הזמנים בהיררכיות זיכרון יהיה יותר גרוע כך יותר חשוב שHit-Rate יהיה קרוב ל-100%.

## – מבנה כללי Cache

בשביל גישה לזכרון הראשי יש bus address עם רגליים כמו'b' הביטים אותם ניתן ניגשים - למשל 32 ביטים. בגישה לזכרון שמים כתובת של בית ומוצאים את הדאטא ששמור בו, למשל אם הכתובת 00..00 אז ניגשים לbit הראשון במערך, אם הכתובת 1000 הולכים לbit הראשון וכן הלאה - אז בעצם בזיכרון רגיל מספיק לתת כתובת והוא מפה ישרות את המקום ממנו צריך להביא את הדאטא.

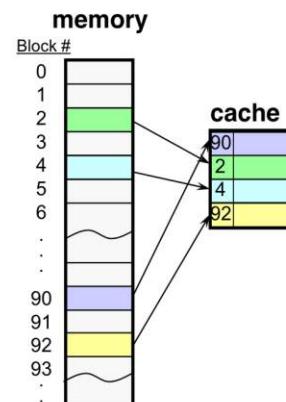
עכשו נישם רעיון חדש - נפרד כתובות, גישה לבית כלשהו בזיכרון, למס' בלוק (לעתים נקרא גם שורה) ולהיסט של הבית הרצוי בתוך הבלוק:



למשל אם נמיין את כל הזיכרון לשורות של 32 בתים או חמישת הביטים התחתיונים, ה-LSB, בכתבota שניאגים אליה הם אלה שיגדרו את המיקום של הבית המבוקש בתוך השורה. אז במקרה של בלוק בגודל B 32 ב' חמישת הביטים הראשונים אומרים איפה בית אני רוצה מתוך הבטים שמהווים את הבלוק האלה.  
חלוקת השמאלי בכתבota קוראים block number ולחקל הנטור הימני offset (היסט) בתוך הבלוק. אם למשל רוצים להביא את הבית הראשון מהשורה האמצעית בזיכרון נכוון אל הבלוק אמצעי ובהיסט נקבע את מס' הבית הראשון רצף (כלומר אפסים).

## Cache – Main Idea

- ◆ **The cache holds a small part of the entire memory**
  - ❖ Need to map parts of the memory into the cache
  
- ◆ **Main memory is (logically) partitioned into blocks**
  - ❖ Typical block size is 32 to 64 bytes
  - ❖ Blocks are aligned
  
- ◆ **Cache partitioned to cache lines**
  - ❖ Each cache line holds a block
  - ❖ Only a subset of the blocks is mapped to the cache at a given time
  - ❖ The cache views an address as



הकаш שומר שורות ברוחב זהה לאלה שהזיכרון מאורגן בו, אלא שבקاش יש מס' מוגבל של שורות, למשל 4 או 8 בלבד, ולכן כאמור יכול רק להוות סאבסט של הזיכרון. בקash לא מספיק להחזיק רק את הדאטא עצמו - צריך להקצות מקום לצד כל בלוק ולשמור עבورو מטא-דאטא. מקום זה נקרא tag array והוא שומר את מס' הבלוק של שורת המקור בזיכרון הראשי ויעזר לנו להזיהות האם השורה שאנו חוננו מחפסים. למשל עבור שורה 90 לא כתוב בשום מקום בזיכרון הראשי את המס' 90 - בקash שורה זו לא בהכרח תהיה בשורה 90 (ואולי אין 90 שורות בקash בכלל) - لكن מס' הבלוק בזיכרון לא מספיק כדי להזותו בקash. לשם כך נשמר את המס' 90 ב tag array בקash (לפחות בחלק) - וכך בקash יוכל��nde לאחסן את כל הבלוקים שאותם שומר tag array. בקash טיפוסי יש כ 32kb של שורה זו. בזיכרון לא צריך את זה כי עושים שם גישה ישירה לפי הכתובת שמחפסים. בקash טיפוסי יש כ 32 בתים, מה שאומר שיש בו 1024 שורות. נניח שעושים בקash lookup של שורת זיכרון - אז לפי הכתובת היא 32 בתים, אז בזיכרון הראשי נשמרים במתמון חייבים להחזיק תג כי הכתובת הישירה מביאה אותנו לשירות tag array נדע אם קיבלנו מיס או היט. אז בזיכרון לא צריך להחזיק תג כי זה סאבסט קטן זה רק 1024, למשל, "רחב הנכוון". אבל אם הרוחבות נשמרות במתמון חייבים להחזיק תג כי זה סאבסט קטן זה רק 4, למשל, שורות מתוך מיליון, אז חייבים לדעת איזה מס' שורה זאת. איך נדע איזה בתים בבלוק לקרוא בהנחה שמצאו את השורה? לפי האופסט! איך נדע כמה בתים רצינו לקרוא? תלוי בפקודה, למשל וא קוראת 4 בתים אבל שורק בית אחד.

נסכם: הקаш מחולק לשורת-קash שנקראת cache-line או **block**. גודלה בדר"כ כ- 32/64 בתים. כמובן שרק חלק מרבלוקים בזיכרון נמצאים בתחום הקash. כאשר מקבל כתובת לחיפוש איז הביטים התחתיו מוחווים את האופסט (מספרם נקבע בהתאם לגודל השורה) ושאר הביטים שייכים למש' הבלוק ז"א הקash צריך לפרק את הכתובת לאופסט וממש' בלוק ועם הлокאפ של מש' הבלוק בקash מצליה איז הוא יודע לפיה האופסט איזה בית צריך לקרוא.

## Cache Lookup

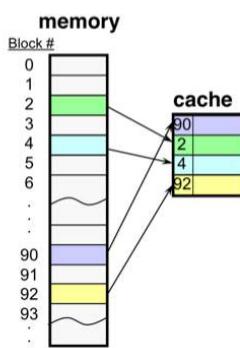
### Cache Lookup

#### ◆ Cache hit

- ❖ Block is mapped to the cache – return data according to block's offset

#### ◆ Cache miss

- ❖ Block is not mapped to the cache  
⇒ do a cache line fill
  - Fetch block into fill buffer
    - may require few bus cycle
  - Write fill buffer into cache
- ❖ May need to evict another block from the cache
  - Make room for the new block



### Cache Read Miss

#### On a read miss – perform a cache line fill

- ❖ Fetch entire block that contains the missing data from memory

#### Block is fetched into the cache line fill buffer

- ❖ May take a few bus cycles to complete the fetch
  - e.g., 64 bit (8 byte) data bus, 32 byte cache line ⇒ 4 bus cycles
  - Can stream the critical chunk into the core before the line fill ends

#### Once the entire block fetched into the fill buffer

- ❖ it is moved from the fill buffer into the cache

data-a Cache Lookup זו פעולה החיפוש של בלוק בקash. אם מצאנו את הנתון נקבל hit (ונעביר למעבד את ה-  
הרצוי), אחרת נקבל miss. Miss אומר שהבלוק שמכיל את הכתובת אליה - הילכנו לקash, עשינו lookup וקיבנו miss. במקרה זה צריך לעשות פעולה שנקראת cache line fill: הקash תמידעובד בגרנולריות של שורה, לכן בfill line מביאים מהזיכרון בלוק שלם בו מוכל הmissing data (לא רק את הדאטא שביקשנו, למשל load `load` מבקשת 4 בתים אבל עדין מביאים שורה שלמה). הבלוק מובא לתוך ה-line fill buffer והכתיבת אל הקash יכולה לחתות מהזרורי שעון: נניח שהdata bus הוא ברוחב 8 בתים, למשל. אם הוא צריך לשורה שלמה של 32 בתים, אז זה לוקח לו ארבע מהזרורי-באס (לא בהכרח ארבע מהזרורי שעון של מעבד). לעיתים (תלויה במימוש) קודם כל מעבירים את הכתובת הクリיטית למעבד ורק אז ממשיכים את תהליכי הכתיבה של שאר הבלוק. כשכל הבלוק הועבר לתוך ה-line fill buffer אז אפשר להעביר אותו לתוך הקash.

בנוסף ב-tag array יש לכל בלוק בקash בית נוסף שנקרוא valid. כל פעם שנשנים מידע בשורה כלשהו בקash נدلיק את הביט הזה עבור השורה הזאת. בעת חיפוש אם  $valid == 0$  לא נוכל לקבל cache hit (גם אם מצאנו את השורה כי המידע לא תקין. יש עוד ביטים בקash שנלמד עליהם בהמשך כמו dirty וכו').

## שיטות לארגון זיכרון ה Cache

עכשו נדברים על שיטות לארגון קash - בפרט נראה שלוש שיטות:

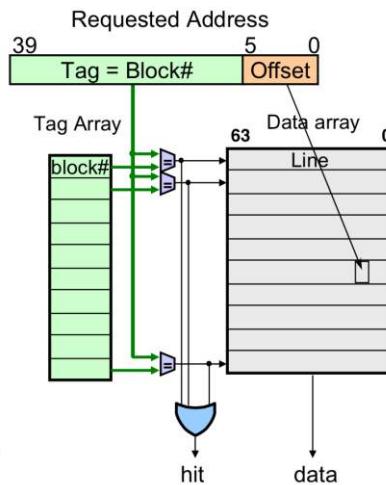
Fully associative .1

Direct map .2

n-way set associative .3

## Fully Associative Cache

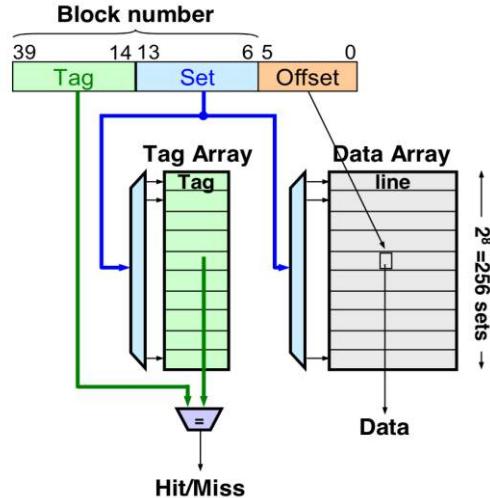
- ◆ An address is partitioned to
  - ❖ offset within block
  - ❖ block number
- ◆ Each block may be mapped to each of the cache lines
  - ❖ Lookup block in all lines
- ◆ Each cache line has a tag
  - ❖ All tags are compared to the block# in parallel
  - ❖ Need a comparator per line
  - ❖ If one of the tags matches the block#, we have a hit
    - Supply data according to offset



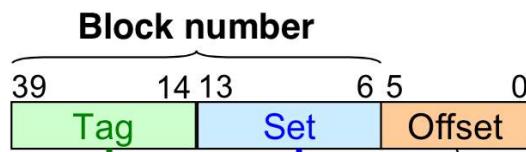
בשיטת fully associative אפשר לשים כל בלוק בזיכרון בכל אחד מהמקומות בקاش: עושים השוואה למס' הבלוק (כלומר לכל הסיביות מלבד סיביות ה offset) עם כל השורות ב- tag array וכך מזהים האם השורה הרצiosa נמצאת אייפשהו בקاش. במקרה של הדוג' בשקף כל שורה היא של 64 בתים (כי יש שיש סיביות offset) וגודל הקаш הוא 32kb ולכן 512 שורות ב-cache ולכן משתמשים ב- 512 קומפרטורים שימושיים במקביל את מס' הבלוק מתוך הכתובת למס' הבלוק ששמור ב tag array כדי לאלהות האם שורה כלשהי נמצאת בקاش. כאמור כל הtags משווים בצורה מקבילית ואם אחד מהם נוטן match זה אומר שיש לנו hit ואפשר לקרוא מתוך אותו בלוק את הנתון הרצוי לפי ה offset ולהעביר אותו למעבד. בעולם אמיתי לא באמת בונים קаш בצורה fully associative, כי כך נדרש הרבה הספק כדי לברר האם שורה נמצאת בקash - בפועל המערכת המקבילית הנ"ל מאוד כבדה ולכן לא נהוג לעשות את זה במערכות גדולות אבל כן במערכות קטניות כמו CN ICOLIM לספק בצורה מקבילית lookup אחד. והם היתרון של שיטתה של fully associative הוא שאפשר למפות כל בלוק לכל מקום. זה גם חיסרונו כי החיפוש נעשה כביד יותר.

## Direct Map Cache

- ◆ **#block l.s.bits map block to a specific cache line**
  - ❖ A given block is mapped to a specific cache line
    - Also called *set*
  - ❖ If two blocks are mapped to the same line, only one can reside in the cache
- ◆ **The rest of the #block bits are used as tag**
  - ❖ Compared to the tag stored in the cache for the appropriate set



עכשו נעבור לשיטות ארגון אחרות. היתרון של השיטות האלו יהיה שאפשר לדעת ב מהירות האם בלוק נמצא בזיכרון והחיסרונו הוא שלבלוק בזיכרון יהיה מס' מקומות מוגבל שהוא יכול להיות בהם. שיטת קיצון אחרת נקראת direct map. בשיטה זו לכל בלוק בזיכרון יש שורה אחת בלבד בה הוא יכול להיות בזיכרון, ז"א כל בלוק מתמפה עפ"י המס' שלו למקום אחד בזיכרון (כל שורה בזיכרון תקרא set). המיפוי הזה נועד לכך שכשנחפש את הדברים יהיה לנו קל לחפש - לפי הכתובת שקיבלו נדע באיזה שורה מחפשים את בלוק בזיכרון ואז באמצעות קומפקטור אחד אפשר לדעת אם הבלוק בזיכרון או לא.



איך נקבע את המיקום היחיד האפשרי בזיכרון ל כל שורה בזיכרון? נניח ויש לנו קаш עם 1024 שורות ורוצים לעשות מיפוי שיאפשר לכל שורה בזיכרון להיות רק במקום אחד. ניקח את 1024 השורות הראשונות בזיכרון ונמפה אותן כפי שהן בזיכרון, כך גם ל 1024 שורות הבאות וכך הלאה. באופן זה למשל שורות 0, 1024, 2048 וכן הלאה יכולות למכת רק לשורה 0 (הראשונה) בזיכרון. כעת יש לקבוע איך נדע פי מס' הבלוק באיזה סט בזיכרון הוא יכול להיות, ואז כשניגשים לאותו סט בזיכרון צריך את התג שמופיע ב tag array עם שאר הביטים של מס' הבלוק. נשים לב שם שמשותף למשל לשורות 0, 1024, 2048 זה שעשרה הביטים מביט 5 עד ביט 14 יהיו כולם 0 (מס' הבלוק מודולו 1024). אך בשיטת direct map מבנה הכתובת הוא קצת שונה: יש לנו של שט סיביות (כי גודל השורה הוא 64 בתים). בשביל הסט צריך מס' ביטים שיאפשר לפנות למס' השורות בזיכרון - למשל בדוגמה יש לנו 256 סטים בס"ה ולכן נדרש שמונה ביטים כדי להזיהות את הסט של פקודת מסויימת. לאחר הגישה לסט ונלק לשורה הזאת ונוסווה את שאר הביטים בכתובת למה שנשמר ב- tag array.

יתרון שיטת direct map:

- אפשר לדעת מהר אם יש לנו hit / Miss.
- אלג' החלפה שלנו הוא מאד טוב: אם מכנים שורה כלשהי ברור שמדוברים את השורה מאותו מקום שמכניםים אליו.

• הפתרון      הכי      טוב      לארגון      מבחינת      Kas

הבעיה של Kas בשיטת direct map היא שם התכנית מתעסקת עם שתי שורות שבמקרה נופלות על אותו סט, למשל שורות 0 ו- 1024 בזיכרון, אז יהיהuboן קונפליקט והן כל הזמן יזרכו אחת השניה מהkas לשארם שallow סטים אחרים נשארם פנויים: כשהגיע ל-0 נביא אותו לkas ואז כשגיע ל-1024 נביא אותו לkas במקום-block 0 ואז כשגיע ל-block 0 שבנו ציא את block 1024 וכן הלאה. אז החיסרון הוא שכל שורה בזיכרון מתחפה למקום אחד בלבד. לכן משתמשים בשיטה הזאת רק במקרים ספציפיים שהוא מתאים.

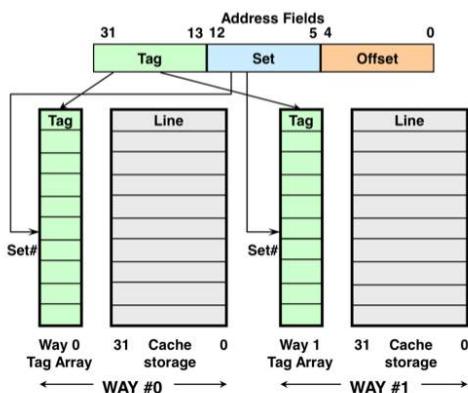
הערה: לא צריך לשמר ב array tag גם את סיביות ה-set: כל block יכול להתREFERENCE ל-set אחד בלבד ולכן אין טעם לשמור את הביטים האלה גם בתאג - עצם המיקום שניגשו אליו בkas לשם השוואת ה tag יחד עם מס' ה-set המשווה באופן ייחודי את הביטים של ה-set ב-block שמחפשים.

דוגמא: יש kas בגודל של 16kb ושורת kas היא בגודל 32 ביט. ע"י חלוקת גודל kas בגודל השורה מסיקים שיש לנו 512 סטים בkas - לכן כדי��ות את ה-set צריך תשעת ביטים. נגד שמחפשים כתובות - ניגשים בkas עפ"י 9 הסיביות המתאימות לשדה ה-set כתובות. שדה הhispt הוא בגודל 5 ביט ולכן שאר שמונה-עשרה הסיביות יהיו ה tag של block. לוקחים אותן ועושמים השוואת עם מה שיושב באותה שורה ב-tag array. הבירהה: בשיטת tag של block. lokchim אונן ועושים השוואת עם מה שכתוב שם זהה אז יש לנו היט.

.3

## 2-Way Set Associative Cache

- Each set holds two line (way 0 and way 1)
- Each block can be mapped into one of two lines in the appropriate set



| Example:                               |                       |
|----------------------------------------|-----------------------|
| Line Size:                             | 32 bytes              |
| Cache Size:                            | 16KB = $2^{14}$ Bytes |
| # of lines:                            | 512 lines             |
| #sets:                                 | 256                   |
| Offset bits:                           | 5 bits                |
| Set bits:                              | 8 bits                |
| Tag bits:                              | 19 bits               |
| Address 0x12345678                     |                       |
| 0001 0010 0011 0100                    |                       |
| 0101 0110 0111 1000                    |                       |
| Offset: 1 0000 = 0x18 = 24             |                       |
| Set: 1011 0011 = 0xB3 = 179            |                       |
| Tag: 000 1001 0001 1010 0010 = 0x091A2 |                       |

Line Size: 32 bytes  $\Rightarrow$  5 Offset bits

Cache Size: 16KB =  $2^{14}$  Bytes

$$\#lines = \text{cache size} / \text{line size} = 2^{14}/2^5 = 2^9 = 512$$

#sets = #lines

#set bits = 9 bits

#Tag bits

$$= 32 - (\#set bits + \#offset bits) = 32 - (9+5) = 18 \text{ bits}$$

Lookup Address: 0x12345678

|                                         |              |            |              |
|-----------------------------------------|--------------|------------|--------------|
| 0001 0010 0011 0100 0101 0110 0111 1000 | tag= 0x048B1 | set= 0x0B3 | offset= 0x18 |
|-----------------------------------------|--------------|------------|--------------|

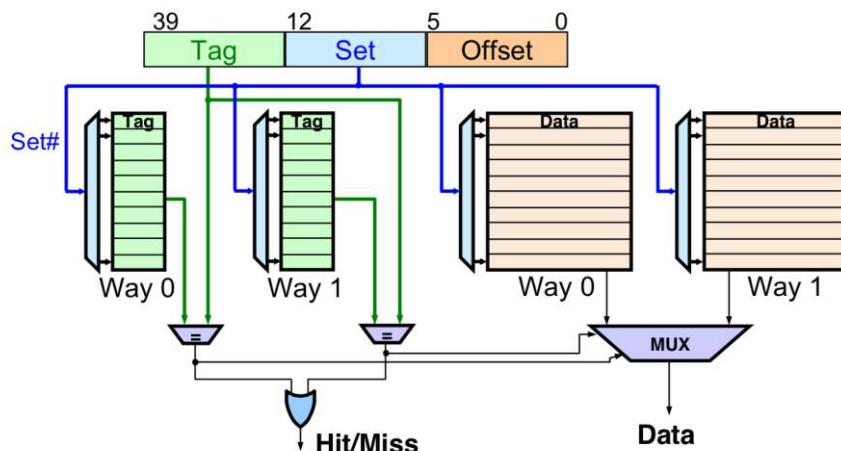
מעבר לשיטה שלישית שנראית set-way: בשיטה זו לכל שורה בזיכרון יש 2 מקומות אפשריים בkas (ways) בהם היא יכולה להיות. ב-direct map היה רק מקום אחד בkas שהינו יכולים למכת אליו. ב-2-way יהיו שני מקומות. ב-4-way נחלק את kas לאربע ways ואז יהיו לנו ארבע מקומות לכל block, וכן הלאה.

ນוחש בזאת: ב-2-way לוקחים את kas, מערך גדול של שורות, ומחלקים אותו לשני ways. כעת את המיפוי של שורות מהזיכרון לkas נעשה עפ"י צ'אנקיהם בגודל way, כלומר עפ"י מס' block מודולו גודל ה-way. לכן לכל שורה בזיכרון יהיו שני מקומות אפשריים בkas במקום מקום אפשרי אחד (שורה אחת אפשרית).

בכל way). באופן דומה אם זה 4-way אז לכל הקאש יחולק לארבעה תתי-מערכות ואז יהיה ארבעה מקומות אפשריים לכל שורה.

בדוג': קאש בעל 1024 שורות שמאורגן בשיטת associative 2-way set associative: מחלקים את הקאש למערך שנקרא-way 0 ו-way 1. בהירה: מס' השורות הכלול בקאש נשאר זהה, אלא שהארגון שלהם משתנה, עקרוני, לשני מערכיהם של 512, 1024, 2048 ו-5,12, ובכל שורה 0, 1, 2, 3... ניתן ללקת לאחד משני מקומות. ע"י שימוש ב-way-ה הורדנו דרמטית את מס' הקונפליקטים כי אם מתענסים עם שורות 0 ו-1024 הן לא יירקו את השניה את אלא יוכלו להישמר ביחד בקאש. בדוגמה' רואים שורה בגודל 32 בתים וקאש בגודל של 16KB - لكن יש 512 שורות סה"כ בקאש. הקאש הוא 2-way וכאן מחולק לשני מערכים של 256 שורות - כלומר שני ways נפרדים בעלי 256 סטים כ"א. אך בשביל לבחור סט צריכה להיות שמותה ביטים ממס' הבלוק, האופסט נשאר זהה ונקבע רק לפי גודל שורת הקאש, וכשעושים lookup משווים את התג לכל אחד מהקומפרטורים המתאימים לסט (אחד בכל way) וכרגע אם אחד מהם שווה אז קיבלנו hit וקוראים את הדטה מאיפה שמצאנו. אם התג לא שווה באף אחד מהם אז קיבלנו miss.

## 2-Way Cache – Hit Decision



בעבודה ב 2-way צריך שתי קומפרציות במקום אחת שהיא ב direct map. כמו direct replacement הוא פחות מהיר מאשר direct כי אם שני הסטים תפוסים ומביאים שורה שלישית לאוטו סט אז צריך לשקל הוצאה משני מקומות אפשריים.

סיכום שיטות ארגון ה-cache: ב fully associative cache יכול להתמפות בכל מקום, ב way ל ח מקומות וב direct map רק למקום אחד. בשיעור הבא נלמד על cache replacement - מנגנון הפינוי ואלגוריתמים שונים לבחירת השורה הפינוי.

## הרצאה מס' 4: ארגון זיכרון המטמון Cache Organization

נשיך לדבר על זיכרון המטמון (קאש). כזכור מטמון זה זיכרון קטן ומהיר, קרובה ל-core, שנועד להכיל את הנתונים שנרצה להשתמש בהם בעתיד הקרוב כדי שלא נצטרך לעשות יציאות יקרות לזכרון החיצוני. בغالל שהקاش קטן הוא מחזיק רק חלק מסוּר השורות בזיכרון - לא יכול להכיל את כלן.

למדנו על כמה שיטות לארגן cache:

### ◆ Direct map

- ❖ A new block is mapped to a single line in the cache
- ❖ Old line is evicted (re-written to memory if needed)

### ◆ N-way set associative cache

- ❖ Choose a victim from all ways in the appropriate set

### ◆ Replacement algorithms

- ❖ Optimal replacement algorithm
- ❖ FIFO – First In First Out
- ❖ Random
- ❖ LRU – Least Recently used

### ◆ LRU is the best

- ❖ but not that much better even than random

- קייזן אחד נקראת Direct Map בו לכל שורה בזיכרון הראשי יש בקاش מקום אחד בלבד שהוא יכול להתחמפות אליו (מקום צזה נקרא שורת-קאש או סט Set).
- הקצה השני, Fully Associative, ש商量שה לכל שורה בזיכרון להתחמפות לכל שורה בקאש.
- פתרון הביניים נקרא **Set Associative** בו המטמון מחולק לכמה חלקים שווים בגודלם שנקראים ways, בכל way אותו מס' סטים, וכל שורה בזיכרון יכולה להתחמפות לסת ספציפי בכל אחד מה-ways (ה-set של שורה נקבע לפי כתובת בזיכרון אבל ה-way יכול להשתנות וכדי לה辨ין באיזה way היא נמצאת היה צריך להשוות מול כל ways באותו סט ספציפי). הבהרה: במעבר ל-way-ה לא שינוינו את גודל הקאש אלא רק את איך שהוא מאורגן: נניח למשל שקאש שהוא way-2-way עם 512 שורות בכל way זהה בגודלו לקאש שמכיל way אחד ובו 1024 שורות (direct map).

בקאש יש **data array** שמחזיק את השורות מהזיכרון הראשי, הנתונים, ויש **tag array** שלכל סט בקאש מחזיק חלק מהכתובת המקורי בזיכרון של השורה שנמצאת בסט זה. ה-tag מאפשר לנו לבדוק בעת חיפוש בקאש האם שורה מסוימת היא זו שאנו מחפשים. כמו כן הוא מכיל שדות-בקרה כמו **dirty** (lookup) ביט.

## אלג' החלפת-שורה במתמונן - Cache Replacement Policies

היום נלמד שיטות להחלפת שורה בקash - Cache Replacement, כלומר שיטה שמאפשרת שורה חדשה לCACHE AIR במקומה.

הערה: הוכנסה לCACHE NKRAAT replacement וההוצאה של שורה ישנה נקראת eviction. בשיטת הארגון הcy פשטota, direct map, יש לכל שורה רק מקום אחד בCACHE שהוא יכול להתמכות אליו - זה מושה אלג' החלפה מאד פשוט: נוציא את השורה שהייתה בסט אליו מכנים את השורה החדשה. בway-way לכל שורה יש אפשרות להתמכות ל-*n* מקומות בCACHE (אותו מס' סט בכל way). במקרה זה אלג' ההחלפה יגיד לנו איזה שורה לזרוק מתוך 4-way associative cache. למשל ב- 4-way associative cache יש אפשרות בחירה מתוך ארבעה מקומות.

יש כל מיני אלג' החלפה, למשל:

- Random בו בוחרים סט אחד מבין כל המתאימים ושים במקומו את השורה החדשה.
  - אלג' FIFO בו מתבצעת החלפה כרך שהשורה הוותיקה ביותר להיכנס היא זו שיוצאת.
  - המדריך כל אלג' החלפה הוא כמה היה ה-Hit-Rate. לכן האלג' האופטימלי, מבחינה זו שהוא שייתן לנו ממד Hit-Rate��, הוא שבהינתן ידע מוקדם על כל הגישות בתכנית נסתכל קדימה ונראה אם מי צריך הcy רחוק, ככלור בעוד הcy הרבה זמן, ואוטו נזרוק מהCACHE. אלג' זה לא מתאים ניתנן לימוש כי לא יודעים את כל הגישות לפני שהן קורות אבל הוא ידוע בדיעבד (אחרי סיום ריצת התכנית) ולכן אפשר למדוד את הביצועים של אלג' אחרים על-פיו.
  - אלג' מאד יעיל שימושים בו נקרא (LRU: Least Recently Used): מוציאים את השורה שהשתמשנו בה לפני הcy הרבה זמן: כל פעם שקוראים/כותבים שורה "נווגים בה" וה-LRU מתחדכן בהתאם. LRU הוא האלג' הידוע הכי טוב שניתן למשרתו, ככלור ייתן את ה-Hit-Rate הcy הקרוב לאלג' האופטימלי.
  - Random: בוחרים סט באקראי מחד ה-ways וורקים את השורה שנמצאת בו.
- הערה: לכל האלג' הנ"ל יש ביצועים דומים. בפועל LRU לא הרבה יותר טוב מ Random.

מימוש אלג' LRU (בחומרה):

## LRU Implementation

### ◆ 2 ways

- ❖ 1 bit per set to mark latest way accessed in set
- ❖ Evict way not pointed by bit

### ◆ k-way set associative LRU

- ❖ Requires full ordering of way accesses
- ❖ Hold a  $\log_2 k$  bit counter per line
- ❖ When way *i* is accessed

$$X = \text{Counter}[i]$$

$$\text{Counter}[i] = k-1$$

$$\text{for } (j = 0 \text{ to } k-1)$$

$$\quad \text{if } ((j \neq i) \text{ AND } (\text{Counter}[j] > X)) \text{ Counter}[j]--;$$

- ❖ When replacement is needed

- evict way with counter = 0

- ❖ Expensive for even small *k*'s

| Initial State |   |   |   |
|---------------|---|---|---|
| Way           | 0 | 1 | 2 |
| Count         | 0 | 1 | 2 |

| Access way 2 |   |   |   |
|--------------|---|---|---|
| Way          | 0 | 1 | 2 |
| Count        | 0 | 1 | 3 |

| Access way 0 |   |   |   |
|--------------|---|---|---|
| Way          | 0 | 1 | 2 |
| Count        | 3 | 0 | 2 |

ב-2-way זה קל מאוד: צריך לבחור אחד מתוך שני סטים להוציאו, שכן שומרים בכל סט בית שואמר האם אליו ניגשנו אחרון, למשל בבית של סט 1 את הערך 1 אם' ממנו אחרון, ואז כשתגיע לשורה חדשה לכאש נדע שהסת עבורו ה-1 הוא LRU-bit, נפנה אותו ונחליפו בשורה חדשה (כמוון שיש לעדכן את הביטים של הסט השני ה-ways כך שהשורה בסט זה כבר לא תהיה ה-LRU אלא אחרת).

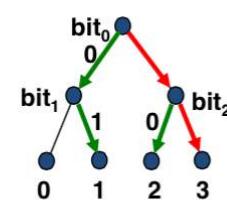
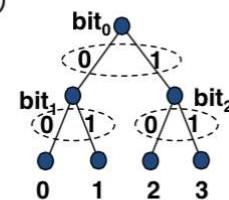
ב-4-way LRU נשמר בכל סט בכל way שני ביטים, למשל אפשר לקבוע שבסט שהשתמשנו בו לפני הכי הרבה זמן הביטים יהיו 00, כלומר הוא ה-LRU, בזה שהשתמשנו בו לפני היו 01, בבא 10 ובסט הטררי ביוטר נשמר את הערך 11. לצורך מישוחו להוציאו מסט כלשהו או נזרק את השורה מתוך ה-way שבסט שלו מופיע הערך 00, נכנסים במקומה שורה חדשה, נקבע לה את ה-LRU הגובה ביוטר, 11, וכל השאר ירדו באחד - זה יהיה 11 יהיה ל-10 וכן הלאה...

נחזיר על זה: זה שקראנו הופך להיות הכי עצובי והערך של כל מי שהיה גדול ממנו מונעם: ברגע שקראנו מישוח ערך השדה LRU שלו הופך ל-3 וכל מי שהיה גדול ממנו יורד באחד - מי שהיה קטן נשאר ללא כל שינוי כשרצהו לזרוק (להחליף) אז נוציא את השן ביוטר ואותו way קיבל את הערך הגבוה.

ה策עה לשיפור: לפי השיטה שהצענו כתעת עבור set צריך לשמור בכל סט שבסט way בדיק (h) log<sub>2</sub> ביטים - למשל בדוגמה' האחרונה 4-way צריך לשמר 2 ביטים לצד כל סט בכל way.策עה יותר חסכונית במקום: לא חייבים לשמר (h) log<sub>2</sub> ביטים לכל סט בכל way אלא לכל מס' סט נשמר מספיק ביטים כדי לייצג את כל העדיפויות (כל הסדרים) עבור אותו סט בכל ה-ways - למשל ב-4-way יש  $2^4 = 16$  אפשרויות לסדר אותם המהידיף ביוטר להכי פחות עדיף - לכך מספיקים חמישה ביטים במקום (שנויים לכל סט בכל אחד מארבעת ה-ways).

## Pseudo LRU (PLRU)

- ◆ **PLRU records a partial order using a tree structure**
  - ❖ The ways of a set are in the leaves; the PLRU bits are the internal nodes
    - For  $n$  leaves (ways) there are  $n-1$  internal nodes (PLRU bits)
- ◆ **Example: 4-ways, using 3 bits per-set:**
  - ❖ Bit<sub>0</sub> specifies if LRU in ways {0,1} or in ways {2,3}
  - ❖ Bit<sub>1</sub> specifies the LRU between {0,1}
  - ❖ Bit<sub>2</sub> specifies the LRU between {2,3}
- ◆ **On each access to a way**
  - ❖ Update PLRU bits from root to hit way to point to the way (0-right, 1-left)
- ◆ **Example: ways access order : 3,0,2,1**
  - ❖ bit<sub>0</sub>=0 points to the pair with way 1
  - ❖ bit<sub>1</sub>=1 points to way 1
  - ❖ bit<sub>2</sub>=0 way 2 was accessed after way 3
- ◆ **Victim selection**
  - ❖ Follow the opposite directions pointed by the PLRU bits



邏輯וון שקשה למש אלגי LRU בחומרה (כלומר כך שייעבוד מהר ולא ידרוש הרבה שטח/הספק) בפועל ממשמשים אלגי Pseudo-LRU שדורש 1-ח ביטים עבור כל סט בקש עם ways-ח, למשל עבור 4-way PLRU needs three bittes per-set. לכל סט ושלו שותם נתונים חיזוי טוב של איפה נמצא ה-LRU - זה לא אלגוריתם מדויק אבל קירוב טוב של ה-LRU. מממשים Pseudo-LRU ע"י סידור 1-ח ביטים כצמתים פנימיים בעץ שהעלים שלו הם הways האפשריים ובכל

רמה בעז הצמתים הפנימיים אומרים לנו את הכיוון שצורך לקחת עד שמגיעים לעלה והוא ה **set** הנבחר. זה מזכיר עץ חיפוש בו בכל שלב משתמשים בביט שהוא הצומת הפנימי כדי להחליט האם לוז ימינה/שמאלה. למשל בדוג' יש 4 ways ואם ה LRU היה ב 0-way או ב 1-way אז הביט הראשון יכיל 0, משמע שהסט שנבחר נמצא תחת העץ השמאלי שלו (שם נמצאים העלים המתאימים ל 0/1 ways) באופן דומה הצומת הפנימי יכיל 0 עבור 0-way ו-1 עבור 1-way.

עכשו כנדרצה לעשות replacement נ历 בדיק בכוון הփוך (כי רוצים את זה שהוא לא בשימוש לאחרונה): איפה שיש בית 0 נ历 ימינה וכן הלאה. האלג' הנ"ל הוא קירוב בכל שלב כי הסדר בצד שלא מעדכנים אותו יכול גם להשתנות (למשל אם הולכים ימינה אז כל הביטים ששיכים תחת העץ השמאלי לא ישנו את הסדר שלהם) וזה לא ישתקף באלג' ה P-LRU - ייחד עם זאת זה קירוב מצוין ל-LRU.

סביר שוב את הדוג': אם ה-LRU נמצא 3-way אז נשמר בביט הראשון בעז 1 ובשני גם 1. נניח שעכשו נגענו בסט כלשהו ב-0-way, אז הביט הראשון הופך ל 0 והביט שהוא שורש תחת העץ ימני גם הופך ל-0 - הביט השלישי נשאר אותו דבר (זה יהיה 1 ברמה השנייה של החיפוש). אח"כعشינו קרייה מ-2-way אז הביט השלישי הופך ל-1 זה שאחורי משתנה ל-0 - הביט של תחת העץ השמאלי נשאר אותו דבר. עכשו עושים ריד מ-1 - אז הביט המרכזי נהייה 0, ושורש תחת העץ השמאלי יהיה 1 - שורש תחת העץ ימני נשאר אותו דבר.

### מדידת ביצועים של זיכרון מטמן

## Effect of Cache on Performance

### ♦ MPI – miss per instruction

- ❖  $\text{MPI} = \# \text{cache misses} / \# \text{instructions}$
- $= \# \text{cache misses} / \# \text{mem access} \times$
- $\# \text{mem access} / \# \text{instructions}$
- ❖ More correlative to performance than cache miss rate
  - Takes into account also frequency of memory accesses

### ♦ Memory stall cycles

$$\begin{aligned} &= \# \text{memory accesses} \times \text{miss rate} \times \text{miss penalty} \\ &= \text{IC} \times \text{MPI} \times \text{miss penalty} \end{aligned}$$

### ♦ CPU time

$$\begin{aligned} &= (\text{CPU execution cycles} + \text{Memory stall cycles}) \times \text{cycle time} \\ &= \text{IC} \times (\text{CPI}_{\text{execution}} + \text{MPI} \times \text{Miss penalty}) \times \text{cycle time} \end{aligned}$$

אחד המדרדים החשובים לביצועים של זיכרון מטמן, ושל תכנית בכלל, הוא **Miss Per Instruction**: בתכנית כלשהי יש מס' מסוים של פקודות גישה לזיכרון, למשל Load-h-MPI. הוא המנה בין מס' הגישות עבורן לא ניתן את הנתון בקשר למס' הפקודות הכלול בתכנית.

MPI זה מודד טוב כדי לקרב **Performance** כי הליטנסי של פקודות רגילות ידוע (פחות או יותר) ובהתוואה ליציאה לזכרון ניתן להניח שהוא קרוב למוחזור שעון יחיד. לעומת זאת, ב-Miss צריך לצאת לזכרון ועל כל יציאה כזו משלמים عشرות ואף מאד מחזורים של חוסר-מעש במעבד - לכן IC מונע קירוב טוב ל-performance. בהינתן

ה CPI האידיאלי וה-MPI אפשר להבין את השפעת הגישות ל זיכרון על ביצועי התכנית (טכנית לחשב CPI מעודכן שיהיה גדול יותר מה-CPI האידיאלי - נראה דוג' מיד).

דוג' לחישוב MPI: תכנית במליצה הتبצעו - 100 פקודות סה"כ ומתוכן 30 היו גישות ל זיכרון (למשל פקודות Load) אז כל גישות הזיכרון מהווים 30% מהפקודות בתכנית. נניח שה-Miss rate של הגישות ל זיכרון הוא 10%, כלומר באחת מכל עשרה פקודות- זיכרון היה Miss - אז בסה"כ מתוך 30 גישות היו לנו 3 החטאות, וה-CPI בתכנית הוא  $\frac{3}{30} = 0.1$  - כלומר שבתכנית הזאת מכל X פקודות התקבלו  $X \cdot 0.1$  החטאות בממוצע.

דוג' לחישוב CPI: בהמשך לדוג' הקודמת אם ה-CPI האידיאלי הוא 1, כלומר כל פקודה ללא עיכובים לוקחת מזמן שעון בודד וזה MPI הוא 0.03, ונניח שבכל יציאה ל זיכרון מוחכים 100 מזמן, אז ה-CPI המעודכן שלנו יהיה:  
 $CPI = CPI(i) + MPI * Miss-Penalty = 1 + 0.03 * 100 = 4.33$

כמו שניתן לראות החטאות בזיכרון הן ממשמעותיות מאוד - בימינו במערכות מודרניותMPI הוא מאוד נמוך, הרבה פחות מ-0.03, וגם ה-Penalty נמוך יותר בגלל שיש עוד היררכיות בין L1 לבין הזיכרון.

הערה: גם בקריאה פקודה נגשים ל זיכרון וגם בשמירה ערך בזיכרון (Store). יחד עם זאת, האלמנט המשפיע הכי הרבה על הביצועים נקרא Load-To-Use Delay: כאשר יוצאים להביא ערך מהזיכרון יש פקודה שמחכה כדי להשתמש בערך זה. לעומת Load, Store צריך לשמור ערך בזיכרון וכך אחד לא מכחלה לו, לכן פקודות ה store משפיעות פחות מ Load על הביצועים. בכל זאת, צריך לחשב על המدد MPI בתור אחור הפעמים שהיינו צריכים לצאת ל זיכרון (לא משנה אם זה היה עבור Store, Load או משהו אחר).

### מדיניות כתיבה ל זיכרון המطمון

יש שתי דרכיים לביצוע הכתיבה (עדכו) של נתונים אשר נמצאים ב Cache אפשר:

## Memory Update Policy on Writes

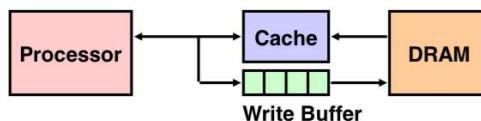
- ◆ **Write back – cheaper writes**
  - ❖ "Store" operations that hit the cache write only to the cache
    - Main memory is not accessed in case of write hit
    - Line is marked as *modified* or *dirty*
  - ❖ Modified cache line written to memory only when it is evicted
    - Saves memory accesses when a line is updated many times
    - On eviction, the entire line must be written to memory
      - There is no indication which bytes within the line were modified
- ◆ **Write through – cheaper evictions**
  - ❖ Stores that hit the cache write both to cache and to memory
    - Need to write only the bytes that were changed (not entire line)
  - ❖ The ratio between reads and write is  $\sim 4:1$ 
    - While writing to memory, the processor goes on reading from cache
  - ❖ No need to evict blocks from the cache (never dirty)
  - ❖ Use write buffers so that don't wait for lower level memory

- ב Write-Back כתובים ישירות רק לkashe ולא ל זיכרון הראשי - למשל אם יש store לכתובת כלשהי אז כתובים את הערך המעודכן בkashe, ובסוף, כנשנץרך לפונת השורה, רק אז נכתב את הערך המעודכן שלה בזיכרון. מדיניות WB מתאימה כאשר יהיו הרבה כתובות לאותה שורה ולא כדאי לבזבז זמן והספק ביציאה ל זיכרון כדי לכתוב שם נתון שישתנה בקרוב. בשביל למשתמש אותה צריך להחזיק לכל שורה במתמון בנוסף לביט ה LRU וbeit ה VALID גםbeit נוסף בשם Y - שיחווה האם הטענה עדכון לשורה מזאת שהיא הועלה לאש. אם הביט כבוי אז אפשר להוציא את השורה הנ"ל בלי לעדכן את הזיכרון אבל אם הוא דלוק נדע שיש בkashe את העותק העדכני היחיד בכל המערכת ולכן כשמוצאים את השורה לטובות אחרית צריך לכתוב את הערך המעודכן אל הזיכרון - בדר"כ הכתיבה מהkashe ל זיכרון געשית ע"י חוץ/באס נפרד. נשים לב שמכיוון שהkashe עובר בגרנוולריות של שורה אז בעדכון נתון מסוים צריך להזריך ל זיכרון את כל השורה שבה הוא מוכל - لكن Y DIRTY תקף לכל השורה.

## Write Buffer for Write Through

- ◆ A Write Buffer between the Cache and Memory

- ❖ Processor: writes data into the cache and the write buffer
- ❖ Memory controller: write contents of the buffer to memory



- ◆ Work ok if store frequency << 1/DRAM write cycle

- ❖ Otherwise store buffer overflows no matter how big it is

- ◆ Write combining

- ❖ combine writes in the write buffer

- ◆ On cache miss need to lookup write buffer

ב Write-Through בכל כתיבה מעדכנים גם את הקаш וגם את הזיכרון - גם את הכתיבה הזאת עושים דרך חוץ מיוחד אבל הפעונטה היא שבכל עדכון של נתון בkashe מעדכנים גם את הקаш וגם את הזיכרון. אם כתובים פעם אחת בלבד אז עדיף להשתמש בשיטה זו ולעדכן את הזיכרון באותו רגע - אבל כשיש כתיבות רבות לאותה שורה אז כל פעם צריך להשקי בעדכון הזיכרון בערכים זמינים שאולי אף אחד לא יקרא אותם, לעומת Write-Through בה כתובים ערך מעדכן רק פעם אחר, כשמננים את השורה מהkashe. בבדיקה במדיניות Write-Back לא יהיה שימוש לביט Y DIRTY כי הקash וה זיכרון תמיד מסונכר נים.

הכתיבה מחוץ-הכתיבה ל זיכרון נעשית בנפרד לפעולות המעבד כי הזיכרון הוא התקן איטי מאוד ולא נרצה ש-write ייעכב את המעבד במשך כל היציאה ל זיכרון. מכיוון שהבאפר נמצא במעבד החלק שמלא אותו יותר מאשר מהיר מזה שמדובר באותו ל זיכרון. לכן נרצה שהחוץ ידע לעדכן את הזיכרון בזמןו החופשי ויאפשר למעבד להיות פנו. בעצם החוץ מאפשר לנו לבצע את הכתובות במהלך מההפעלה לאט-לאט הן יועברו ל זיכרון. כמות ה-write בתכנית ממוצעת היא פחות מ 10% מסך גישות ל זיכרון ולכן אפשר לתוכנן את המערכת בהנחה שישיעור הכתיבה בה יהיה הרבה יותר קטן משיעור הקרייה. אם יהיה הרבה כתיבות אז החוץ יתמלא והמעבד יצטרך לחכות לו שיתרока כדי ל מלא אותו שוב. לכן WT נדרש את הבאפר יותר מאשר ב WB: ב WB כתובים תמיד גם ל זיכרון וגם ל kashe - אלא שאי אפשר לכתוב ל זיכרון באותו קצב של הקash, ההבדל הוא בערך בין 4gh ו 4mh.

## מדיניות הקצאה לזכרון המטען במקרה של גישה לכתיבה

ראינו מה קורה בכתיבה לנตอน שכך נמצא בקash, כתה נלמד מה עושים ב- Write Miss - המקרה שרצוים לכתוב לנตอน אך הוא לא נמצא בקash. נניח שרצינו לכתוב בכתובת כלשהי והוא לא נמצא בקash - השאלה היא האם מביאים את השורה מהזיכרון לפקש או שוכותם אותה בזיכרון בלבד. גם כאן נבחין בין שתי אפשרויות: Write Allocate או No-Allocate.

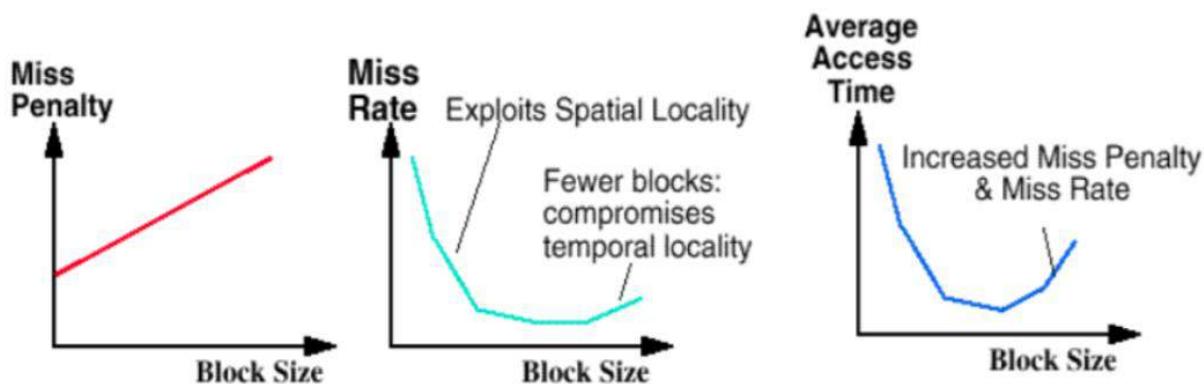
- בשיטה של Write-Allocate מביאים את השורה מהזיכרון לפקש. בדרך כלל הולך עם Write-Back עם Write-Allocate וWrite-No-Allocate הולך עם Write-Through, כי ב-WB משתמשים כשוחובים שיחיו הרבה כתיבות לאוותה שורה, שכן שווה גם להתאמץ ולהביא אותה ע"י Write-Allocate, לעומת זאת אם אנחנו מעדכנים את הזיכרון בכל כתיבה לפקש (Write-through) אז פחות הגיוני להביא שורה מהזיכרון לפקש בשביל כתיבה אלא פשוט לעדכן אותה בזיכרון ולכן יותר הגיוני להשתמש עבור כתיבות בעבור Write-No-Allocate (אלא אם natürlich גם לקריאות מסוימת שורה, במקרה זה שווה לעשות Write-Allocate).

## גודל שורת הקash

לגודל שורת הקash יש אפקט מכמה בחרינות:

## Cache Line Size

- ◆ **Larger line size takes advantage of spatial locality**
  - ❖ Too big blocks: may fetch unused data
  - ❖ Possibly evicting useful data  $\Rightarrow$  miss rate goes up
- ◆ **Larger line size means larger miss penalty**
  - ❖ Takes longer time to perform a cache line fill
    - Using critical chunk first reduces the issue
  - ❖ Takes longer to evict (when using write back update policy)



- עקרון הלוקליות: אנחנו יודעים שאם נגדיל את גודל השורה נצלח לתפוס יותר שורות שקרובות אליה ומעקرون המיקומיות למרחב נkirב Hit-Rate טוב כי נביא גם את הכתובות הקרובות לשורה שכתבנו אליה וכנראה ניגש אליה בעתיד. מצד שני אם מבאים מס' בתים גדול מדי אז נביא גם נתונים שאולי לא נשימוש בהם ובמקרים נירוק מהמטמון בלוקים שכן יהיו שימושיים - למשל במקרה של הבאת פקודות אולי פקודת branch תגרום לפיצעה עד שנגיע לביבוע כל הפקודות העוקבות שהבאנו יחד אתה - צריך למצוא איזון ויש אופטימיזציה לעשות

- שיקול שני הוא Latency: ככל שהורת קаш היא יותר ארוכה ה Miss-Penalty יותר ארוך ו לוקח לנו יותר זמן כדי לבצע את העברה.

• שני אלה מופיעים על זמן הגישה הממוצע לזכרון - AMAT (Average Memory Access Time)

בגרף: רואים את האפקט של גודל השורה על-h-Miss-Penalty, בהתחלה הוא קבוע כלשהו וככל שגודל השורה גדל כך עולה גם-h-MP (גרף לינארי). בgraf האמצעי רואים את השפעת גודל השורה על-h-Miss-Rate - בהתחלה עם הגדלת השורה-h-MR ירד כי נביא דברים ייחד אתה, ובאיישחו שלב-h-MR מתחילה לעלות בגל שהבלוק גדול מדי והוא מביא דברים לא רלוונטיים על פניו חוסר מקום לבlokים אחרים שייכלו להשתמש בהם. בgraf מימין לוקחים את האפקט של גודל הבלוק על-h-MR וה-h-MP ומחשבים אותו בסיסחה המציג את-h-Access Time Average Memory, זמן הגישה הממוצעת לזיכרון. הקומבינציה של שני ההיבטים הנ"ל נותנת לנו את אותה איישחו נק' איזון (נק' המינימום בזמן הגישה לזיכרון). בימינו גודל השורה האופטימלי הוא בדר"כ 32 או 64 בתים, כਮובן כתלות בצריכים (האפליקציות המורוצות).

## סוגי החטאות ב-cache

## Classifying Misses: 3 Cs

#### ◆ Compulsory

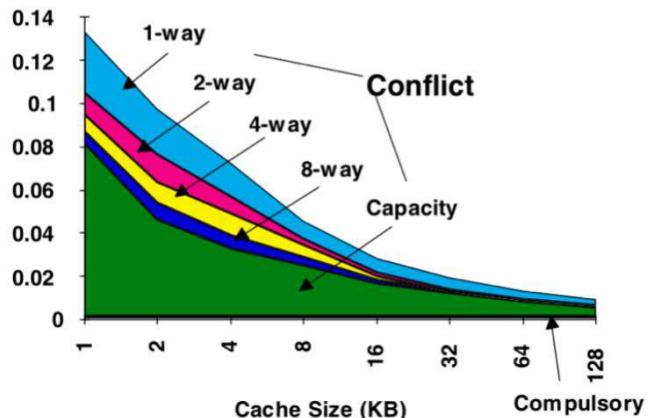
- First access to a block which is not in the cache
    - the block must be brought into the cache
  - Cache size does not matter
  - Solution: prefetching

#### ◆ Capacity

- Cache cannot contain all blocks needed during program execution
    - blocks are evicted and later retrieved
  - Solution: increase cache size, stream buffers

## ◆ Conflict

- Occurs in set associative or direct mapped caches when too many blocks are mapped to the same set
  - Solution: increase associativity, victim cache



ישנים שלוש סיבות לקלות בזיכרון: Cache-Miss, Conflict-Miss ו-Capacity-Miss.

- **Compulsory-Misses** הן החטאות "מחויבות-המצוות" - החטאות שנובעות מכך שנתקלנו בכתובה כלשהי בפעם הראשונה בתכנית. נשים לב שזה לא משנה איך נארגן את הקаш, מה יהיה גודלו והאסוציאטיביות שלו - בכל האפשר בפעם הראשונה המידע לא יהיה באש ונתקל בהחטאות אלה. לעומת זאת אפשר להגדיל את גודל שרשרת הזכור וכן להביא יותר מידע מראש, גם לפני שהיינו צריכים אותן. כל החטאות הקשורות בגישה ראשונה לבлок כלשהו הן **Compulsory-Misses**.

- Capacity-Miss נובעים מהמגבלה של גודל הקаш: אלה החטאות שהיו נפתרות אם הקаш שلنנו היה יותר גדול, למשל החטאות שם במקום 16KB הקаш היה בגודל היה 32KB היה מתאפשר עבור הgeshit שלhn hit. דוג' בקash שהוא Fully associative כל החטאה של מידע שלא השתמשנו בו בפעם הראשונה שהבאוו אותו היא capacity-Miss כי זרכנו אותם רק כשהכנסנו משהו אחר ולא היה לנו מספיק מקום בשבייל להשאיר אותם. כאמור, מה שמאפיין סוג זה של החטאות הוא שם הקash היה יותר גדול הן לא היו קורות.

- - בהחטאות אלה האסוציאטיביות של הקаш לא הייתה מספיק גבואה - למשל החטאות שאם היינו עובדים באסוציאטיביות 2-way או 4-way או לא הייתה החטא. כמובן שאם קASH הוא fully associative אז אין החטאות של Conflict ובמקרה יש החטאות של Capacity - לעומת זאת אם הקASH הוא direct mapped כנראה שהוא יהיה לנו Conflict-Miss דוג' קלאסית היא כאשר עובדים ב-Direct-Mapped עם שתי שורות שמתמחפות לאותו סט וכל פעם מכנים אותם יחד ומוציאים את השניה ואז בגישה הבאה (לשורה האחרת) תמיד נקבל Miss, אז נביא את השורה ונוציא את האחרת במקומה עד שנצטרך אותה שוב, נקבל Miss, וחזר חלילה. באופן כללי-ways היא החטאה כשייש כמה שורות-זיכרון רלוונטיות לתכנית שמתמחפות לאותו סט וממלאות אותו בכל הways כך שצורך להוציא שורות שנצטרך בעתיד ואז מפספסים אותן - כל זאת למקרה שעדיין יש בקASH מקום פניו בסיטים אחרים ואם היינו עושים שיטת מיפוי אחרת אז כל השורות יהיו נכנסות לkish בזמנית.

הבהרה: ההבדל בין Capacity-Miss ל-Miss-Conflict תלוי בשאלת האם ארגון אחר של הקASH היה הופך את ההחטאה לפגיעה.

בגרף (מיימן) רואים התפלגות נפוצה של סוגי החטאות בקASH עבור אסוציאטיביות שונה. בחתית הגרף (שכבה הדקה מודול) יש את ה compulsory misses שמקורם בתחלת ריצת תוכנית, כאמור אלה החטאות שייקרו בכל מקרה ולכן שייעורם נשאר זהה לכל גודל/אסוציאטיביות של קASH. הדבר הבולט ביותר שראויים הוא שבהתאם לגודל הקASH率 Miss-rate יורד (ככל שהקASH גדול יש פחות פספוסים). רואים שעבור גודל קבוע ככל שנגדיל את האסוציאטיביות של קASH נקבל פחות Conflict misses עד שגען ל-fully-associative cache. החלק העיקרי הוא המרחב capacity misses שנובעים מגודל הקASH ואינם תלויים באסוציאטיביות (כלומר מתרחשים עבור כל הזיכרונות).

### שיפור ביצועי ה-cache

עכשו נדבר על פתרונות לכל הדברים האלה: איך להעלות את ה-Average Memory Access Time ע"י הורדת שיעור ה-*Conflict-Misses*, איך לפתור בעיות ה-Capacity-*Misses* וכו'. אלה ישפרו לנו את ה-Hit-Rate /או יורידו את ה-Hit-Time ו/או ישפרו את ה-Miss-Penalty

:Victim Cache

## Victim Cache

- ◆ **Per-set pressure may be non-uniform**
  - ❖ Some sets may have more conflict misses than others
  - ❖ Solution: allocate ways to sets dynamically
- ◆ **Victim cache gives a 2<sup>nd</sup> chance to evicted lines**
  - ❖ A line evicted from L1 cache is placed in the victim cache
  - ❖ If victim cache is full  $\Rightarrow$  evict its LRU line
  - ❖ On L1 cache lookup, lookup victim cache in parallel
- ◆ **On victim cache hit**
  - ❖ Line is moved back to cache
  - ❖ Evicted line moved to the victim cache
  - ❖ Same access time as cache hit
- ◆ **Especially effective for direct mapped cache**
  - ❖ Enables to combine the fast hit time of a direct mapped cache and still reduce conflict misses

Victim Cache הוא מטען קטן נוספת, Shmoochyniyot ה החלפה שלו היא FIFO הוא יעיל במיוחד כשייש סט "חם" או כמה סטים "חמים", בהקשר זה שימושים בהם שוב ושוב על-פני סטים אחרים. זיכור מבחינה Hit Time המבנה של Direct Map הוא הכי אפקטיבי כי מציאת השורה בקash היא מהירה מאוד. הבעה אליו היא שהוא סובל מהמונן קונפליקטים. Victim Cache נועד לפחות אוטומ: אם יש לנו כמה סטים חמים אז נשים חוץ עם מעט cache lines בצד, גם קרוב ל-core, ועכשו כשייש לי קונפליקט ומוציאים שורה כלשהי ניתן לשורה הזו "הזרנות שנייה" - לא נזרק אותה למגררי אלא נשים אותה ב-cache victim. עכשו בעיה לוקלית של סט חם תיפטר באמצעות ה- cache victim למשל אם יש מקום לאربع שורות אז נוכל לאחסן בו עד ארבעה סטים חמים - בפועל זה יכולו הוספנו לסט כלשהו כמה ways באופן דינמי (עפ"י דרישות התכנית שבוצעת). נשים לב שב痴יפוש, lookup, צריך לחפש את ה-tag הרצוי גם ב-Cache הרגיל וגם ב-Victim-Cache. שימרנו ב-cache לא משפר את ה-MP ואף מעלה במקצת את ה-Hit Time אבל משפר את ה-Hit-Rate.

## Stream Buffers

- ◆ **Before inserting a new line into cache**
  - ❖ Put new line in a stream buffer
- ◆ **If the line is expected to be accessed again**
  - ❖ Move the line from the stream buffer into cache
  - ❖ E.g., if the line hits in the stream buffer
- ◆ **Example**
  - ❖ Scanning a very large array (much larger than the cache)
  - ❖ Each item in the array is accessed just once
  - ❖ If the array elements are inserted into the cache
    - The entire cache will be thrashed
  - ❖ If we detect that this is just a scan-once operation
    - E.g., using a hint from the software
    - Can avoid putting the array lines into the cache

הרענון הבא נקרא Stream Buffer - רוצים לשורה כלשהי תוכיח שימושים בה יותר מפעם אחת לפני שימושים אותה לקash: בפעם הראשונה שימושים בשורה בתכנית עושים חיפוש, מקבלים החטאה וכש מבאים את השורה מהזיכרון לא מכנים אותה ל-cache לא ל-stream buffer. אם יש עוד פעם hit על נתון מסוימת שורה - רק אז נעביר אותה מה-stream buffer לקash. באופן זה איז רוצים שהמידע "יוכיח שהוא ראוי" להיות בקash ע"י גישות נשנות וננסכים להכניס לCACHE רק מידע שימושים בו הרבה. כך אם קוראים את מידע פעם אחת בלבד איז לא ניתן לשורה שלו להיכנס לCACHE ואכן באמת לא צריך אותה שוב פעם - אך גם לא נוציא שום דבר במקומה.

דוגמא: מערכת ענק, יותר גדול מהCACHE, וקוראים כל נתון בו פעם אחת - אם נכנס את כל הנתונים שלו לCACHE זה ידריש הוצאה של כל המידע שהוא שם עד כה (כאמור המערכת הרבה הרבה יותר גדול מהCACHE) - כלומר נזהם את הקash. במקומות להגדיל את הקash הוספנו חוץ של כמה שורות ובכך מנענו זיהום של הקash, שמוביל לשיפור ב-Hit Rate.

## Prefetching

- ◆ **Hardware Data Prefetching – predict future data accesses**
  - ❖ Next sequential / Streaming
    - Triggered by an ascending access to very recently loaded data
    - Fetches the next line, assuming a streaming load
  - ❖ Stride prefetcher
    - Tracks individual load instructions, detecting a regular stride
    - Prefetch address = current address + stride
    - Detects strides of up to 2K bytes, both forward and backward
  - ❖ General pattern prefetcher
    - Identifies patterns of Load address distances
- ◆ **Data has to be prefetched before it is needed**
  - ❖ The prefetcher aggressiveness has to be tuned
  - ❖ How fast and how far ahead to issue the prefetch request, such that data arrives on time, before it is needed
- ◆ **Prefetching relies on extra memory bandwidth**
  - ❖ Too aggressive / inaccurate prefetching slows down demand fetches
    - Hurts performance
- ◆ **Software Prefetching**
  - ❖ Special prefetching instructions that cannot cause faults
- ◆ **Instruction Prefetching**
  - ❖ On a cache miss, prefetch sequential cache lines into stream buffers
  - ❖ Branch predictor directed prefetching
    - Let branch predictor run ahead

שיפור נוסף נקרא prefetching. הרעיון ב-prefetching הוא להתמודד עם ה-Compulsory Misses (מצור). החטאות שמתרכחות עבור שימוש במידע כלשהו בפעם הראשונה ע"י הבאת בלוקים ל-Cache עד לפני שהשתמשו בהם בפעם הראשונה. נזכיר שcompulsory misses הן החטאות בעיתיות מאוד כי גם אם נגדיל את גודל הקаш או האסוציאצייה שלו הן עדיין יקרו, והשאלה היא איך בכל זאת אפשר לטפל בהן. האלגוריתם הבנאי ביותר לימוש prefetch נקרא next-line prefetching: בהבאת שורה כלשהי מהזיכרון לקаш גם את השורה העוקבת לה. אם אנחנו יותר חכמים נחבר למנגנון זה את חזאי הקפיצות ואז לא תמיד נביא את השורה העוקבת - למשל אם יש לנו בראנץ'. אלגוריתם שלישי מסתכל על ההיסטוריה של התכנית, מזהה את הכתובות שהיא משתמשות בהן, ומנסה להזמין תבניות לגישות הבאות למשל אם יש גישה חזרת לאחר קפיצה לפקודה במרקח 128 ביטים מהקפיצה אז נדע להביא את הבלוק שמכיל את שורה זו. דוג' נוספת היא פקודת load כך שהמעבד מזוהה שאחריו הגישה אליה צטרך גם כתובת במרקח כלשהו ממנה, למשל סריקה של איברי מערך. לשם כך מנהלים טבלה במעבד שאומרת איזה נתון בעת הגישה כתובת כלשהי - זה נקרא אלגוריתם stride (גודל הקפיצה). נשים לב שחשיבות בלבד הכתובת שנצטרך גם את התזמון של הדברים, למשל כמה איטרציות לפני ההגעה לקפיצה ליזום את הבאת הנתון יש את הליטנסי של הבאים החיצוני, ואם נביא את השורה מוקדם מדי יכול להיות שנארוק אותה עד השימוש, אם מאוחר מדי אז נצטרך לחכות לה ואז כמובן זהה לא prefetch אופטימלי. לכן לאלגוריתם prefetch יש עבודה תכnon רובה. כאמור יש אלגוריתם שעובדים עפ"י תבנית פשוטה ויש-Calala שעובדים עפ"י תבנית מורכבת.

את ה-prefetching אפשר לעשות גם בתוכנה ע"י פקודת load שمبיאה את הנתונים וגם בחומרה ע"י זיהוי loads של הכתובות הבאות והבאתן מראש. דיברנו עד כה על prefetch data אבל אפשר להביא גם פקודות לפני הזמן. Hit-Rate Prefetching מוריד את ה-Hit-Rate.

הראינו שיפורים של ה-Hit-Rate באמצעות victim-cache וכן ע"י prefetching. עכשו רוצים לראות איך משפרים את ה-Miss penalty

## Critical Word First

- ◆ **Reduce Miss Penalty**
- ◆ **Don't wait for full block to be loaded before restarting CPU**
  - ❖ *Early restart*
    - As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - ❖ *Critical Word First*
    - Request the missed word first from memory and send it to the CPU as soon as it arrives
    - Let the CPU continue execution while filling the rest of the words in the block
    - Also called *wrapped fetch* and *requested word first*
- ◆ **Example: Pentium**
  - ❖ 64 bit = 8 byte bus, 32 byte cache line  $\Rightarrow$  4 bus cycles to fill line
  - ❖ Fetch date from 95H

|         |         |         |         |
|---------|---------|---------|---------|
| 80H-87H | 88H-8FH | 90H-97H | 98H-9FH |
| 3       | 2       | 1       | 4       |

זה אלג' שאומר שכשהמעבד מקבל cache-Miss, יצא החוצה לזכרון להביא מידע ובינתיים מהכח לשורה שמכילה את הנתון או קודם כל נבייא מהזיכרון את המילה חשובה לו מתחום אותה שורה, ניתן אותה למעבד שיוכל להמשיך לרווח ובקביל/לאחר מכן נבייא מהזיכרון את שאר השורה לקаш, ככלומר ניתן למעבד את המידע חשוב לו כדי שהוא יוכל לעבוד ורק אח"כ נביא את כל השאר. אפשר גם לנצל את עקרון המקבימות במרחוב ולעשות זאת זה בשיטת Wrap Around: להביא למעבד את המילה לה הוא ממחכה ואז את שתי המילים הסמוכות לה (משני הצדדים של המילה), ואז את הסמכות להן וכן הלאה, ורק בסוף לעדכן את הקаш עם כל השורה שהובאה. פתרון זה משפר את הזמן שהמעבד צריך לחכות אחרי Miss - קלומר משפר את ה- Miss-Penalty.

עד כה דיברנו על אופטימיזציות בחומרה - עכשו נעבור לדבר על אופטימיזציות ברמת הקומפיילר, בפרט הורדת ה- Hit-Rate מהטור הבנה של איך המטען בנוי.

## Code Optimizations: Merging Arrays

- ◆ Merge 2 arrays into a single array of compound elements

```
/* Before: two sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: One array of structures */
struct merge {
    int val;
    int key;
} merged_array[SIZE];
```

- ◆ Reduce conflicts between val and key
- ◆ Improves spatial locality

בדוג' רואים הגדרה של שני מערכים וקוראים באופן סדרתי את שניהם, אחד אחרי השני. בלי שום אופטימיזציה ישנה חילוק ל לקרוא את המערך השני נקבע החטאות בקש בזמן שנוציא ממנה את איברי המערך הראשון - פעולה אפשר למנוע את החטאות אם הקומפיילר יdag שמרת שני איברים באינדקס זהה במערך ישבו באותו מקום בזיכרון.

הדוג' הבאה נקרא **loop fusion**:

## Code Optimizations: Loop Fusion

- ◆ **Combine 2 independent loops that have same looping and some variables overlap**
- ◆ Assume each element in `a` is 4 bytes, 32KB cache, 32 B / line

```
for (i = 0; i < 10000; i++)
    a[i] = 1 / a[i];
for (i = 0; i < 10000; i++)
    sum = sum + a[i];
```

In the example only  
load accesses are  
taken into account for  
hit rate calculation

- ❖ First loop: hit 7/8 of iterations
- ❖ Second loop: array > cache ⇒ same hit rate as in 1<sup>st</sup> loop

- ◆ Fuse the loops

```
for (i = 0; i < 10000; i++) {
    a[i] = 1 / a[i];
    sum = sum + a[i];
}
```

- ❖ First line: hit 7/8 of iterations
- ❖ Second line: hit all

יש שתי לולאות בלתי-תלוויות שעוברות על מערך כלשהו. המערך מחזיק 10,000 אלמנטים, כל אחד בגודל ארבעה בתים, ולכן כל לולאה דורשת מעבר על 40,000 בתיםסה"כ. אם בקש 32KB אז הוא לא יוכל את כל הנתונים של המערך בבהת אחת - לכן במעבר הראשון אם כל אלמנט הוא 4 בתים ושורת הקаш היא בגודל 32 בתים אחרי כל אלמנט שהוא עבורו פגיעה נקלע שבע החטאות, ובמעבר השני נדרש להחזיר חלק מהאיברים שהוצאו חזרה לזכרון -סה"כ נסבול גם מגיעה אחת ואחריה שבע החטאות. אז אם נעשה את שתי הלולאות כפוי שונכנתבו במקור Hit-Rate בתכנית יהיה  $7/8$  בשתי האיטרציות. מצד שני אם נעשה merge לאחת כך שבכל איטרציה נבצע את שתי הגישות לאיבר כלשהו במערך אז בכל איטרציה נסbold את החטאה רק עבור הפוקודה של לולאה אחת ובשניה כבר יהיה לנו תמיד את המידע - לכן בפקודות שהיו שייכות לולאה השנייה נקלט Hit-Rate מושלם, 100%.

הרעיון שהנחנו אותו בדוג' של Loop-Fusion הוא שאם כבר עושם שימוש בנตอน עדיף בביטחון לאחסן בו את כל השימושים שהתקנית צריכה, למשל לקרוא אותו שוב לצורך פעולה אחרת. נשים לב שאם המערך היה נכנס בקש בשלהותו אז זה לא היה משנה והלולאה השנייה עדיין הייתה חווה Hit-Rate מושלם.

## Code Optimizations: Loop Interchange

- ♦ 2-dimension array in memory:

```
x[0][0] x[0][1] ... x[0][99] x[1][0] x[1][1] ...
```

```
/* Original program: access are 100 bytes apparat */
/* x[0][0] x[1][0] ... x[4999][0] x[0][1] x[1][1] ... */
for (j = 0; j < 100; j++)
    for (i = 0; i < 5000; i++)
        x[i][j] = 2 * x[i][j];
```

```
/* Reversing the loops order: access are adjacent */
/* Improved spatial locality */
for (i = 0; i < 5000; i++)
    for (j = 0; j < 100; j++)
        x[i][j] = 2 * x[j][i];
```



דוג' נוספת: מטריצה דו-ממדית שרוצים לעבור על כל האיברים שלה ולעשות משהו בלתי תלוי בכל אחד מהם, למשל להכפיל את ערכו. בהנחה שכל האיברים מסוודרים באופן רציף בזיכרון כנראה שאם נעשה את המעבר על האיברים לפי עמודות ואז לפי שורות, ולא להיפר, אז ה Hit-Rate יהיה לא-טוב. קומpileר שמבין איך הזיכרון מאורגן יכול לתכנן את אופן ביצוע הלולאה (כלומר לגשת לאיירם סמוכים קודם) כדי לשפר את ה Hit-Rate.

סיכום: ראיינו שיטות לשפר את ביצועי הקаш וחילקונו אותן לשושן קטגוריות:

- ♦ Reduce cache miss rate

- ❖ Larger cache
- ❖ Reduce compulsory misses
  - Larger Block Size
  - HW Prefetching (Instr, Data)
  - SW Prefetching (Data)
- ❖ Reduce conflict misses
  - Higher Associativity
  - Victim Cache
- ❖ Stream buffers
  - Reduce cache thrashing
- ❖ Compiler Optimizations

- ♦ Reduce the miss penalty

- ❖ Early Restart and Critical Word First on miss
- ❖ Non-blocking Caches (Hit under Miss, Miss under Miss)
- ❖ Second Level Cache

- ♦ Reduce cache hit time

- ❖ On-chip caches
- ❖ Smaller size cache (hit time increases with cache size)
- ❖ Direct map cache (hit time increases with associativity)

• שיפור ה hit-rate, למשל הגדלת הקash (Capacity misses או הגדלת גודל השורה ו/או Prefetching) או מביאים נתונים לפני ביקשו אותם (Conflict Misses). אפשר גם לשפר את ה Cache Penalty של ה Cache.

- שיפור ה Miss-Penalty, למשל באמצעות Critical Word First.
- שיפור ה Hit-Time, למשל ע"י אופטימיזציות של הקומPILEר.

## הרצאה מס' 5: חלוקה ושיתוף בזיכרון המטמון, מערכת מחשב, זיכרון DRAM

בסוף הפרק שעבר דיברנו על שלוש אופנים בהן ניתן לשפר זיכרון מטמון:

1. להוריד את ה hit time.
  2. להוריד את ה Miss penalty.
  3. להעלות את ה  $(\text{Miss rate}) = \text{hit rate}$ .
- ראינו כמה הצעות לשפר כל אחד מהדברים הנ"ל.

### הפרדת L1 Cache לנזונים ופקודות

## Separate Code / Data Caches

### ◆ Parallelize data access and instruction fetch

### ◆ Code cache is a read only cache

- ❖ No need to write back line into memory when evicted
- ❖ Simpler to manage

### ◆ Self modifying code

- ❖ Whenever executing a memory write  $\Rightarrow$  snoop code cache
  - Requires a dedicated snoop port: tag array read + tag match
  - Otherwise snoops would stall fetch
- ❖ If the code cache contains the written address
  - Invalidate the line in which the address is contained
  - Flush the pipeline – it main contain stale code

, L2 Cache-*inst. Cache* L1 (First Level Cache, L1) FLC מופרד לשניים: . אחראי יש את ה- L1 Data Cache-*instr. Cache* . שבדרכו משותף לכל המעבדים, ולעתים קיימים במערכת גם L3. בדרך כלל השניים הראשונים נמצאים בתוך המעבד, בעוד L3 נמצא מחוץ למעבד אך עדין על הципוף. אוח"כ יש את הזיכרון הראשי שעשויה מ-DRAM - נלמד עליו קצת לקרואת סוף ההרצאה.

הפרדה L1 לקוד ולפקודות נתנת לנו אפשרות' לגשת ל- Data Cache *instr. cache* ול- *instr. cache* במקביל, ככלומר לקרווא פקודות ונזונים באותו זמן. למשל המעבד יכול להביא את הפקודה הבאה לביצוע בתכנית בזמן שפקודת load או store תיגש ל- Data Cache - ע"י הפרדה ה- FLC נוכך לטפל בשתי הפעולות ביחד.

:self-modifying code הוא ה- read-only *instr. cache*, אם כי ב-*i*86x, הארכ' של אינטל, אפשר להרים self-modifying code ה- cache. קוד שמאפשר לכתוב לאזור הזיכרון שמכיל קוד של תוכנית, ככלומר לאפשר לפקודת store לכתוב ערך חדש שישנה את הפקודה המקורי בתכנית - למשל שינוי פקודת ADD ל SUB.

בעבודה עם self-modifying code צריך לבדוק כמה דברים:

- הקוד שמשנים נמצא גם בזיכרון ו咎 ב-*instruction cache* - אם כן, מלבד ביצוע store לזכרון צריך לעשות invalidate לשורה המכילה את הפקודה בקש כך שפעם הבאה שנבצע אותה נקרה את הפקודה המעודכנת.

כדי לא להפריע לתהיליך הקריאה התקין של הפקודות המתבצעות כתע בתכנית (כלומר להימנע מ structural hazard) משכפלים את ה- tag array בקash וועושים lookup על העותק החדש, למשל לאחר כל פעם שבוצעים store בודקים ב tag array האם 1000 נמצא 1000 inst. Cache הבדיקה האם פקודה מסוימת נמצאת במתמון הפקודות נקראת snoop (רחרוח). הערה: נשים לב שלא כותבים ישרות ל�ash הפקודות אלא ל�ash של המידע (כמו כל store) ואז קוראים את הפקודות ל�ash הפקודות.

- בנוסף לפועלות invalidation ציריך לעשות ריקון (flush) ל-Pipe ולהתחליל את ביצוע הפקודה הנוכחית מחדש. מרוקנים את ה-Pipe כי יכול להיות שהפקודה ששונתה הספיקה להיכנס וכעת צריך להתחליל אותה מחדש (לאחר השינוי).

### גישה לזכרון-מטמון במקביל

## Non-Blocking Cache

## Multi-ported Cache

### Hit Under Miss

- Allow cache hits while one miss is in progress
- Another miss has to wait
- Relevant for an out-of-order execution CPU

### Miss Under Miss, Hit Under Multiple Misses

- Allow hits and misses when other misses in progress
- Memory system must allow multiple pending requests
- Manage a list of outstanding cache misses
  - When miss is served and data gets back, update list

### N-ported cache enables $n$ accesses in parallel

- Parallelize cache access in different pipeline stages
- Parallelize cache access in a super-scalar processors

### About doubles the cache die size

### Possible solution: banked cache

- Each line is divided to  $n$  banks
- Can fetch data from  $k \leq n$  different banks in possibly different lines

Non-Blocking Cache זו תכונה אפשרית של קאשימים במעבדי (Out Of Order) OOO, עליהם נלמד בהמשך הקורס. במעבדים אלה יתכן שפקודות מסוימות "יעקפו" פקודות אחרות שלוקוח הרבה זמן לבצע אותן. מטמון מסוג Non-Blocking מאפשר להמשיך להריץ תכנית גם כאשר מקבלים cache Miss, כלומר המעבד יאפשר לנו לבקש לנו את ה-data של פקודה כלשהו בזמן שהוא מטפל במצב של miss מפקודה אחרת. בפרט לאחר שפניתי לבקש לנו את ה-data של פקודה כלשהו בזמן שהוא מטפל במצב של miss מפקודה אחרת. בפרט לאחר שפניתי לבקש וקיבلت ה-Miss אז מטמון שתומך ב hit under Miss יאפשר להמשיך לבצע את התכנית ולגשת אליו עד פעם גם בזמן שהמחשב עוזר מביא את ה-data מהרמות הבאות, אולי כדי לבצע חיפוש של כתובות אחרת במטמון (נשים לב שהוא רלוונטי למעבדי OOO כי במעבד סדרתי התכנית תלואה בפקודה שקרה עבורה ה-Miss). שכלל נוסף הוא מעבד שתומך ב hit under multiple misses יכול לקבל כמה החטאות מכמה פקודות זיכרון שונות ועוזר להמשיך לעבוד.

בדרכ' בזיכרון-מטמון יש יותר מ-snook, נקודת גישה, אחת - למשל אם נרצה לאפשר גישה ל�ash ע"י שתי פקודות במקביל. אפשרות אחת לימוש קASH שמאפשר גישה ע"י שתי פקודות במקביל (כלומר קASH עם שני פורטים) היא לשכפל את ה tag-array בלבד אך לא את ה-data array - זה פתרון כדי ויחסי-וזל. בוגרנו ל-*data-array*: מחלקים כל שורה ב�ash לכמה בנקים (banks), למשל אם גודל שורה הוא 64 בתים והחלינוו לחלק לארבעה בנקים אז יש בכל בנק 16 בתים. נשים בכל בנק port, נקודת גישה, וכך יהיה אפשר לספק לשתי הפקודות את המידע במקביל. עוזר בכל פעם שייהי קונפליקט על גישה לאותו מס' בנק באותו שורה נctrkr לחוכות - لكن חלוקה לכמה שיטות בנקים היא טוביה יותר בהקשר זה - מצד שני היא מוסיפה פורטים ועודalogika לזכרון המטמון מה שעשו

אותו יקר ואיטי יותר. נשים לב שנמנענו מלשכפל את ה **data array** - פעולה מאוד יקרה. יחד עם זאת אין ברירה אלא לשכפל את ה-**tag array** כדי לאפשר חיפושים במקביל של נתונים.

## L2 Cache

- ◆ **L2 cache is bigger, but with higher latency**

- ◆ Reduces L1 miss penalty – saves access to memory
- ◆ L2 cache is located on-chip, but may be shared by a few cores
- ◆ L2 is unified (code / data)

- ◆ **L2 inclusive of L1**

- ◆ All addresses in L1 are also contained in L2
  - Address evicted from L2  $\Rightarrow$  snoop invalidate it in L1
- ◆ Data in L1 may be more updated than in L2
  - When evicting a modified line from L1  $\Rightarrow$  write to L2
  - When evicting a line from L2 which is modified in L1
    - Snoop invalidate to L1 generates a write from L1 to L2
    - Line marked as modified in L2  $\Rightarrow$  Line written to memory
- ◆ Since L2 contains L1 it needs to be significantly larger
  - e.g., if L2 is only 2x L1, half of L2 is duplicated in L1

בימינו לרוב המערכות יש L2 מאוחד (Unified) קלומר מחזק פקודות data באוטו זיכרון. עושים זאת כי הרעיון של הפרדת נתונים ופקודות נועד לאפשר גישה מרובה במקביל לשני חלקים של זיכרון המסתמך L1 שניגשים אליו הרבה. בגלל שה-rate hit של L1 מאד גבוהה פונים ל-L2 בתדירות הרבה יותר נמוכה ובדרך"כ אין שתי גישות אלו באותו זמן - לכן אין טעם שהזיכרון אחרי L1 יופרד לפקודות ונתונים.

L2, כפי שהוא לומדים אותו, הוא **inclusive**, כלומר מתקיים ביןיהם עקרון הכללה - כל מה שנמצא ב-L1 נמצא גם ב-L2. בגלל זה היחס בין הגודלים שלהם חייב להיות לפחות פי 10 - אם למשל היתי עושה זיכרון L2 כפול בגודלו מ-L1 ומקיים את עקרון הכללה אז רק חצי מגודל הזיכרון L2 היה זמין לנתחים נוספים שלא ב-L1 - זה לא היה הגיוני כי L2 הייתה הרבה פחות אפקטיבי.

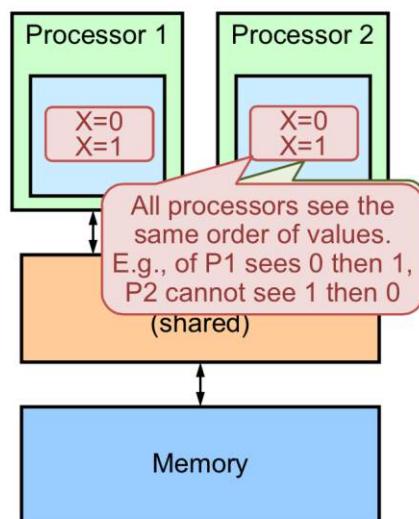
עקרון הכללה מחייב שכל פעם שעושים הוצאה מ-L2 צריך לעשות אותה גם ב-L1, כלומר בעת הוצאה מ-L2 עושים Lookup ב-Tag array של L1 לראות אם הכתובת נמצאת - אם כן עושם לה invalidation. בפרט, שעושים פינוי של שורה מ-L2 שהיא מעודכנת ב-L1 או אותו-time snoops גורר כתיבה ל-L2. רק אחרי הוצאה מ-L1 מוצאים את השורה מ-L2 - אך אין מצב ששורה נמצאת ב-L1 אך לא ב-L2 (שמירה על זיכרון ההקלטה). כמו כן שלפני הוצאה מ-L2 אם השורה מעודכנת גם ב-L2 אז נדרש לכתוב אותה לזכרון הבא כדי לא לדרוס אותה.

בנוסף כזכור L1 Write Back מעדכן את המידע בעת כתיבה ב-L1 בלבד ואז רק שם יהיה עותק המעודכן. לכן בפעולות **invalidate** snoops בודקים ב-L1 האם שורה היא dirty - אם כן מעדכנים אותה ב-L2 ורק אז ממשיכים את ה-update ב-L1. שעובדים ב-L1 Write-Through לבית dirty אין משמעות.

## פרוטוקול MESI: ניהול זיכרונות-מטמון במערכת מרובת מעבדים

◆ A memory system is **coherent** if

1. If P1 writes to address X, and later on P2 reads X, and there are no other writes to X in between  
⇒ P2's read returns the value written by P1's write
2. Writes to the same location are serialized:  
two writes to location X are seen in the same order by all processors



עכשו נדון ב- Caching במערכת מרובת-מעבדים, כולם שיתוף, sharing, של מידע בין כמה מעבדים שלכל אחד מהם יש L1 ולכלם L2 משותף - לצורך מינימום ש-L2 מקיים את עקרון הכלכלה. מערכת מרובת-מעבדים חייבת להיות קוהרנטית, כלומר כשמעבד כלשהו כותב לכתובת כלשהו ואות"כ מעבד אחר רוצה לקרוא את הערך הנ"ל אז צריך לשים לב שהמעבד יקרא את הערך המעודכן. המצב הנוכחי מורכב יותר כאשר יש הרבה מעבדים וגם במקרה זה צריך לשמר על עקביות במידע שקורא כל מעבד - הנושא נקרא גם Memory Ordering. לשם כך צריך לבנות מערכת בה שכאשר אחד יפנה לקאש המטמון שלו הוא ידע האם מעבד אחר כתב ערך אחר לאוთה כתובות במטמון שלו (מעקרון ההכלזה זה אומר שהשורה תהיה גם במטמון המשותף, אך לא בהכרח מעודכנת) וגם כן - קיבל את הערך המעודכן ולא מה שמופיע בסביבתו.

הפרוטוקול שמקיים את השיתוף הנ"ל נקרא MESI. הפרוטוקול מאפשר ארבעה מצבים לכל שורה קאש:

◆ Each cache line can be in one of 4 states

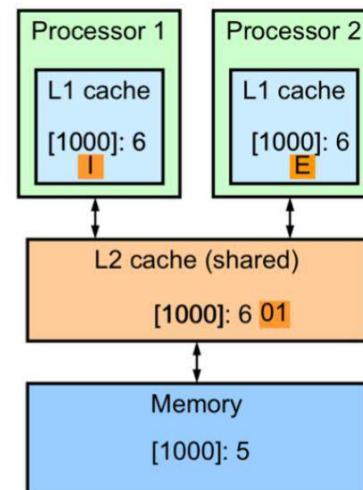
- ❖ Invalid – Line's data is not valid
- ❖ Shared – Line is valid and not dirty, copies may exist in other processors
- ❖ Exclusive – Line is valid and not dirty, other processors do not have the line in their local caches
- ❖ Modified – Line is valid and dirty, other processors do not have the line in their local caches

- מכן נובע השם של הפרוטוקול. המצב נשמר עבור כל שורה בכל L1 של כל אחד מהמעבדים. L2 מתחם את כל הפעולות של הפרוטוקול כפי שנראה בדוגמה מיד.
- המצב הכי פשוט הוא Invalid שהוא מציין שלא תקין אצל מעבד כלשהו.
  - אומר שהמידע בשורת הקאש תקין וקיים אצל אחד מעבדים. אם מעבד מבקש שורה כלשהו והוא כבר קיימת במעבד אחר, לא מעודכנת, אז L2 יספק אותה למעבד שביקש וגם אצלו יהיה במצב Shared.

- אומר שהמידע תקף, ומובטח של מעבדים אחרים אין את המידע זהה בקאים שלהם. אם מעבד רצה שורה שאין אצל אף מעבד אחר אז הוא מקבל אותה במצב Exclusive.
- אומר שהשורה שונתה ע"י מעבד מסוים כלשהו ולמעבדים אחרים אין את העותק העדכני של השורה הזאת בקاش הפרטוי שלהם.

נעsha דוג' וナルך שלב-שלב לראות איך ה프וטוקול עובד:

- ◆ **P1 reads 1000**
- ◆ **P1 writes 1000**
- ◆ **P2 reads 1000**
- ◆ **L2 snoops 1000**
- ◆ **P1 writes back 1000**
- ◆ **P2 gets 1000**
- ◆ **P2 requests for ownership with write intent**



- בתחילת מעבד 1 רוצה לקרוא נתון כתובות 1000 - נגד המס' 5. הוא הולך ל-L1 שלו, והוא שורהlookup לכתובות 1000 ומקבל Miss, בעקבותיו אותו מעבד עשו lookup ב-L2 (המשותף) ומקבל Miss, لكن L2 יוצא לזכורן וambil את התוכן של כתובות 1000 ל-L2. כשהנתון ישלח ל-L1 מעבד 1 יקבל אותו במצב Exclusive (כיום שהוא לא כתוב אליו, אז לא Modified, והוא גם לא חולק אותו, אז לא Shared).
- כשמעבד 1 רוצה ל כתוב ערך חדש לכתובות 1000, למשל 6, השורה תעבור במצב Modified: מצב זה אומר שהמעבד שינה את הנתונים בשורה ומובטח שיש לו את העותק העדכני היחיד במערכת - אם מישחו אי פעם יצטרך לקרוא את זה L2 צריך לגשת ל-L1 של המעבד הכתוב ולבקש את הערך המעודכן ממש.
- עציו מעבד 2 רוצה לקרוא את 1000 ומתקבל cache Miss ב-L1 שלו. הוא ניגש ל-L2 שם המידע קיים אך על L2 לטפל בו יש מעבד שני שפינה את הכתובות הזאת במתווך הפרטוי שלו (ואכן הכתובות נמצאות אצל מעבד 2). לכן מעבד 2 עשו noop לכתובות 1000 כדי לקבל את המידע מ-L1 של מעבד מס' 1, והוא מעביר לمعالב 2 את הנתון המעודכן. כשלשוני הקוראים יש את הנתון מסומנים אותו ב-L1 של כל אחד מהם כ-Shared.
- עצו מעבד 2 רוצה ל כתוב אל כתובות 1000 - בשביל זה הוא יצטרך לקבל בעלות (Ownership) כי רק הבעלים של שורה-קash יכול ל כתוב לה. בשביל לקבל בעלות מעבד 2 מודיע ל-L2 שהוא בוצע כתיבה ו-L2 צריך לעשות לمعالב 1 invalidation noop על הכתובות הזאת, ככלומר להודיע לمعالב 1 שהנתון שיש לו ב-L1 כבר לא מעודכן. אח"כ L2 מעביר את הנתון אל מעבד 2 במצב Modified (ולא Exclusive כי מעבד 2 רוצה לשנות אותו). באופן כללי ב-MESI - מי שרוצה לעדכן נתון צריך להיות owner שלו ולשם כך הוא צריך לעשות noop על הכתובות מהנתון ב-L2. כלומר לכלת לכל המעבדים שהמידע נמצא בהם, לעשות invalidate, להביא את הנתון אליו ורק אז לעדכן את הנתון. נשים לב שרק אם הוא רוצה ל כתוב הוא צריך להשיג בעלות ולבצע noop על הכתובות מהנתון.

invalidation בЛИבוט האחרות. אם מעבד רוצה לקרוא ערך הוא יכול להחזיק אותו במצב Shared יחד עם מעבדים אחרים.

◆ **L2 needs to keep track of the presence of each line in each of the processors**

- ❖ Determine if it needs to send a snoop to a processor
- ❖ Determine in what state to provide a requested line (S,E)
- Need to guarantee that the L1 caches in each processor are inclusive of the L2 cache

◆ **When L2 evicts a line**

- ❖ L2 sends a snoop invalidate to all processors that have it
- ❖ If the line is modified in the L1 cache of one of the processors (in which case it exists only in that processor)
  - The processor responds by sending the updated value to L2
  - When the line is evicted from L2, the updated value gets written to memory

◆ **A modified line must be exclusive**

- ❖ Otherwise, another processor which has the line will be using stale data
- ❖ Therefore, before modifying a line, a processor must request ownership of the line

ראינו ש-L2 בעצם עוקב אחריו המצב של כל שורה: לכל שורה ב-L2 נשמר בית אחד לכל מעבד באמצעותו L2 יודע באיזה מעבד יש את השורה, כולם באיזה מעבד היא באחד מה מצבים Shared, Modified או Exclusive, ובאיזה מעבד ריא לא قيمة כלל (או Invalid) - כך L2 יודע בתגובה לבקשת שורה באיזה מה מעבדים צריך לעשות snoop ובאיזה מצב להעביר את השורה למעבד ש牒ש אותה (Exclusive/Modified או Shared). תלוי אם היא נמצאת גם במעבד אחר. אחרי שהמעבד שעשה כל snoop עשה לו את השורה ועדכן אותה במתמונן הפרט של כותב את השינוי שעשה אל L2. ככל תרחיש אפשרי ה-unified L2 cache ידע לתת את השורה למעבד שביקש במצב הנוכחי.

פרוטוקול MESI מבטיח ש-L2 תמיד יוכל לדעת מה קורה עם נתון ב-L1, שכן חיבורים של L1 יהיה מוכן ב-L2 וכן כשייש פינוי שורה ב-L2 צריך לשולח הודעה invalidate snoop לכל המעבדים שמחזקים את השורה הזאת. אם השורה היא ב-L1 של אחד מהמעבדים אז צריכים לעדכן את L2 ורק אז לכתוב את ערך השורה לזכרון.

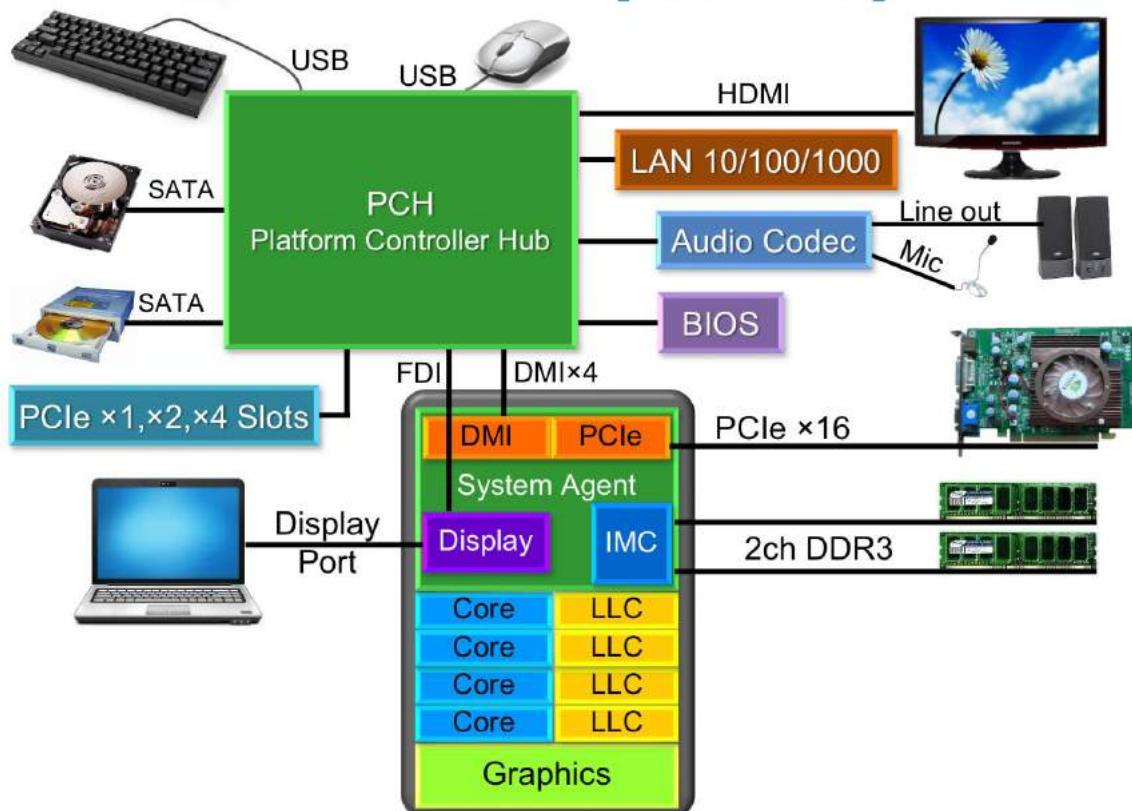
| State     | Valid | Modified | Copies may exist in other processors |
|-----------|-------|----------|--------------------------------------|
| Invalid   | No    | N.A.     | N.A.                                 |
| Shared    | Yes   | No       | Yes                                  |
| Exclusive | Yes   | No       | No                                   |
| Modified  | Yes   | Yes      | No                                   |

בטבלה רואים את ההבדל/דמיון בין המצבים השונים של MESI. כשמיידע לא תקף במעבד כלשהו הוא תמיד במצב Invalid - בכל שאר המצבים המידע תקף. במצבים shared ו-exclusive המידע תקף אך לא מעודכן ובו הוא מעודכן (כਮון במעבד אחד בלבד). אם במעבד כלשהו שורה נמצאת במצב Shared אז מעבדים אחרים יכולים להכיל את המידע שנמצא בה, במצב Exclusive או Modified היא פרטית למעבד.

כשמעבד רוצה לקרוא שורה שיש במעבד אחר אז היא תעבור למצב Shared בכל מקום שהיא נמצאת (כולל במעבד שמקבל אותה) שכן עתה יהיה קיים עותק ממשהם. כשמעבד רוצה לכתוב לשורה L2 צריך לתת לו עלייה בעלות ownership, ע"י snooop invalidate בשאר המעבדים שמכילים אותה כך שהיא תהפוך ל-Exclusive אצלו ולאחר הכתיבה השורה תעבור מ-Modified ל-Exclusive.

### מערכת מחשב – Computer System

## Personal Computer System



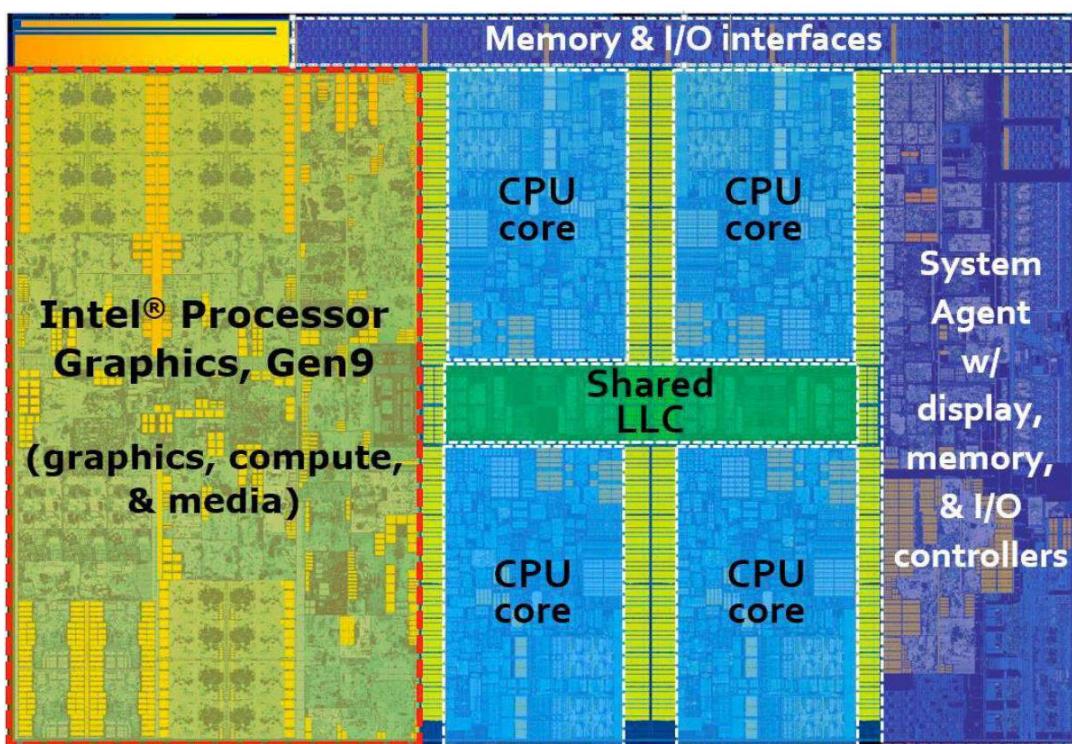
עכשו מתחילה לדבר על מערכת מחשב, קלומר על הצ'יפ הראשי של המחשב. בין היתר הצ'יפ הראשי מכיל:

- ליביות (cores). בימינו כל ליבת מגיעה עם L2,L1.
- LLC (הרמה האחורונה של הקאש - משותפת).
- מעבד גרפי.
- את ה System Agent - תכף נראה מה יש בו.
- לפניו כמה שנים עשו אינטגרציה לבודק הזיכרון לתוך הצ'יפ ויצרו את ה- DRAM שמדובר עם ה DRAM.

- כל השטח מחוץ לKERNA נקרא Uncore, זה בעצם המדיום שמחבר את כל הרכיבים במעבד ביחד - היום נלמד על פרוטוקול העבודה שלו ואיך הוא ממשת את רשת התקשרות בין הרכיבים השונים במעבד.
  - רכיב ה-PCI-Express מבקר ייחידות-קצת מהירות למצלם מסך חיצוני
  - ה port מתחבר לתצוגה LCD, למשל בנייד מדובר על המסך הפנימי.
- הערה: במחשב זה יש שני צ'יפים סה"כ, זה נקרא solution 2 chip. שני הצ'יפים יושבים על לוח האם. ה PCH הוא הצ'יף השני המהווה את הבקר של מכשירי הפלטפורמה במחשב - אפשר לחבר ל-PCH התקני קטן כמו דיסקים, וכן התקנים איטיים כמו מקלדת, כרטיס קול ועוד.

## 6<sup>th</sup> Generation Intel Core™ – Skylake

- 14nm process
- Quad core die, with Intel HD Graphics 530



במעבד Skylake של אינטל יש ארבע ליבות, מעבד גרפי ו-LLC משותף של 8 סה"כ. לכל ליבה יש זיכרון L2 פרטיז בגודל KB256 וזכרון L1 בגודל 32KB. הגישה ל-L2 נעשית באrbטwa מחזוריים והגישה ל-L1 בשני מחזוריים. עם כל ליבה שנוסףת מגיעה חתיכת זיכרון עבור ה-LLC (מה שמסומן בירוק הינו למעשה האיחור של ארבע חתיכות, אחת מכל מעבד). יחד עם זאת ה-LLC הוא משותף ולא שייך לליבה כלשהי. גודל טיפוסי לכל חתיכת LLC שmaguya עם הליבה הוא MB2, וכן בתמונה מתאפשרים MB8 מרבע חתיכות. בהרבה: ה-LLC הינו משותף לכל המעבדים ואף חתיכה אינה שומרת נתונים של מעבד ספציפי בלבד. מיד נראה כיצד מtbody השיתוף של ה-LLC

**סביבה המעבד - Uncore**

- **The uncore subsystem includes**
  - The System Agent (SA) handles I/O
    - Contains PCI Express, Memory Controller
  - The Graphics Unit (GT)
  - The Last Level Cache (LLC) Slices
  
- **A high bandwidth ring bus**
  - Connects between the Cores, LLC slices, Graphics, and System Agent

בשאר הרצאה נדון ב- Uncore וסביבתו, שבפועל מוגדר להיות הכל מלבד הליבוט והמעבד הגרפי. בפרט נדון באופן הפעולה של מה שנקרא Ring המחבר בין כל הישויות (Agents) שצרכיהם לקבל מידע ולעשות שימוש של מידע דרך ה-LLC. בפרט נדון ב-System Agent שמאפשר לנו לצורך חיבורם לעולם החיצון ומספק לנו את המידע דרכו. כמו כן ה-Uncore צריך לספק ערך מהיר ורחב (High-Bandwidth Bus) כדי להעביר את המידע IMC, Display ועוד. ה-Uncore בנוי בצורה של טבעת, ring, וכך לחבר בין ה-LLC לבין הסוכנים השונים במעבד. ה-uncore בנוי במבנה של טבעת, ring, וכך לחבר בין ה-LLC לבין כל המשתמשים בו. כל הליבוט וכמו ה-Graphics והמדיה - כולם משתמשים ב-LLC.

- **The LLC consists of multiple cache slices**
  - The number of slices is equal to the number of IA cores
  - Each slice can supply 32 bytes/cycle
  - Each slice may have 4/8/12/16 ways
    - Corresponding to 0.5M/1M/1.5M/2M block size
  
- **Physical addresses are uniformly distributed among cache slices using a hash function**
  - The LLC acts as one shared cache
    - With multiple ports and BW that scales with the number of cores
    - Ring/LLC not likely to be a BW limiter
  - Prevents hot spots
  
- **LLC is shared among all Cores, Graphics and Media**
  - GFX/media may compete with cores on LLC

כאמור ה-LLC מורכב ממשטחים (slices), אחת מכל ליבה - עם כל ליבה מגיעה חתיכת זיכרון של ה-LLC אבל החתיכה אינה פרטית - ה-LLC מהוות משאב אחד שמשותף לכלם. כל חתיכה תורמת לרוחב-הפס לקריאה/כתיבה אל ה-LLC עד 32 בתים בכל מחרוז, למשל במערכת עם שתי ליבות אפשר לקרוא/לכתוב לתוך ה-Ring עד 64 בתים בכל מחרוז. מכך אנחנו למדים שהרענון של LLC המורכב מפיסות שmagiuot עם כל מעבד זה פתרון שהוא

scalable כי ככל שימושים ליבוט מגבירים את רוחב הקריאה של הפס מול ה-LLC. לכל חתיכת LLC יכולים להיות כמה ways - כיום כל ליבת מגיעה עם חתיכת LLC בונח 2MB ו-16 ways.

ל-LLC, כמו כל קאש, יכול להיות direct map, fully associative או N-way (כלומר כתובות פיזיות יכולה להיות מפוזרות בכל מקום בקאש). כל אחת מהLIBOT יכולה לגשת אל כל ה-LLC כי כפי שהסבירנו הוא מתפרק כ-shared cache ייחד. הסבירנו כיצד הוא תמיד קורנטי וכי צד כל ליבת תקבל בעת הצורך מידע עדכני (עקרון ההכלה ובפרט-MESI), מדובר יש לו scalable multiple ports וכי צד הוא במס' הליבות.

- **LLC hit latency is 26-31 cycles**

- Depends on Core location relative to the LLC Slice  
(how far the request needs to travel on the ring)

- **Fetching data from LLC when another core has the data (in E/M states)**

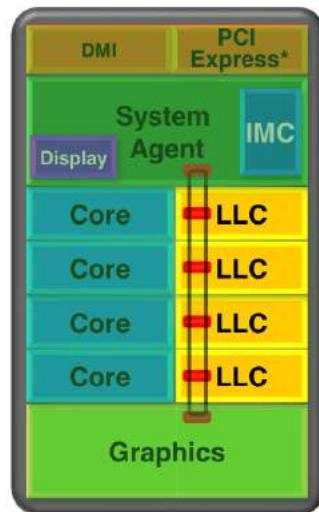
- Clean hit – data is not modified in the other core – 43 cycles
- Dirty hit – data is modified in the other core – 60 cycles

- **LLC is fully inclusive of all core internal caches**

- Eliminates unnecessary snoops to cores
- Per core “Core Valid bit” indicates if the internal core caches need to be snooped for a given cache line

- **Traffic that cannot be satisfied by the LLC**

- LLC misses, dirty line write-back,  
non-cacheable operations, MMIO/IO operations
- Travels through the ring to the IMC



זמן שלוקח לקרוא מה-LLC (hit time) תלוי במיקום הפיזי של הנתון ב-LLC ביחס לLIBOT הקוראת: קריאה של LIBOT מהסליל הסמור לה תהיה יותר מהירה מקריאה נתון שנמצא רחוק וצריך להעבירו על פני רוב ה-ring. لكن latency משתנה בהתאם למיקום הפיזי של הLIBOT והנתון.

במקרה שאנו רוצים לקבל data מהקאש שנמצא במצב Exclusive או Modified אז כמובן שהוא ייקח יותר מהזרורים כי לא מספיק רק לקבל את המידע אלא צריך להפעיל את ה프וטוקול MESI שלמדנו קודם שעשוže invalidation לכל המעבדים וכו' - הبات נתון תיקח כ-40 מהזרורים אם המידע לא מעודכן באך מעבד, אחרת הוא תיקח כ-60. כזכור ה-LLC מקיים את עקרון ההכללה - הוא inclusive רמות הקאש שמעליו ובכך הוא מונע snoops למעבדים שלא מחזקם את הנתונים כי כמו שראינו ה-LLC יודע לאיזה מהLIBOT יש את השורה. כמו כן ה-LLC במעבד Skylake זוכר בכל עת איפה דלוק ה-bit valid ולכן עושה invalidate snoop רק למי שצעריר. כמובן שהוא-LLC לא יכול לספק מידע נתון שנמצא רק בזכרון - במקרה זה מקבלים LLC Miss. בנוסף הפעולות של O-MIO Non-Cacheable לא עוברות דרך הקאש אלא שירות דרך בקר הזיכרון (the IMC) - אך בRing קיימת גם תחנה, system agent, כדי להכניס מידע שלא שמור ב-LLC.

הזכירנו בעבר את הנושא של Data Prefetchers - הרעיון הוא להביא דברים מהזיכרון ולשים אותם בקash לפני שהקוד צריך אותם. נזכיר את ה prefetcher השונים למעבד:

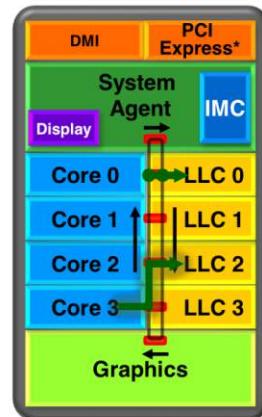
- **Two HW prefetchers fetch data from memory to MLC and LLC**
  - Prefetch the data to the LLC
  - Typically data is brought also to the MLC
    - Unless the MLC is heavily loaded with missing demand requests
- **Spatial Prefetcher**
  - For every 128B-aligned cache line fetched to the MLC, prefetch the next sequential cache line, to complete a 128B-aligned chunk
- **Streamer Prefetcher**
  - Monitors read requests from the L1 caches for ascending and descending sequences of addresses
    - L1 D\$ requests: loads, stores, and L1 D\$ HW prefetcher
    - L1 I\$ code fetch requests
  - When a forward or backward stream of requests is detected
    - Prefetch anticipated fwd/bwd sequential cache lines (no stride)
    - Cannot cross 4K page boundary (same physical page)

יש בחומרה שני Prefetchers שמביאים נתונים מהזיכרון ל-LLC ובדר"כ גם ל-MLC (כי ב-LLC אין הרבה החטאות). מנגנוןם אלה פועלם רק אם ה-MLC לא עמוס ע"י demand request, כלומר בבקשתות אמתיות שהמערכת כבר מacha להן (בניגוד לבקשתות של prefetching).

1. המנגנון הcy אלמנטרי נקרא Spatial prefetcher שמבייא את ה next line יחד עם השורה הנוכחית. למשל עבור כל גישה ל זיכרון להבאת שורה של 128 בתים הוא מביא גם את ה-128 הבאים.
  2. אל' יותר חכם נקרא streamer prefetcher שבעצם monitoring על כל בקשות הקרייה שmagiuot מה- L1 cache וכשהוא מגלה סדרה עולה/ירדת של בקשות אז הוא מייצר בלבד את הבקשות להבאה של הכתובות הבאות בסדרה. ניתן להביא שורות קדימה (סדרה עולה של גישות ל זיכרון) או אחורה (סדרה יורדת). בכל מקרה הוא לא חוצה גבולות של 4KB - תמיד מבאים אותו דף (nlmed על דפים בנושא של Virtual Memory).
- כשזההים סדרה כלשהי ואח"כ היא מתחילה שוב, למשל יש בקשה מהליבה לכתובת Ch ויהינו שאחריה יגיעו בקשות נוספות לכתובות עוקבות, אז ניתן להתחיל אוט', סדרת בקשות עולה כך שלמש מבאים שתיק בקשות קדימה וכשתהיה בקשה נוספת מבאים את השתיים הבאות וכן האלה. אפשר לפתח עד ל-20 סטרים כאלה לכל מיני שימושות prefetching שלローンטיות לתוכנית, כל אחד מהם יכול להביא פקודות מסדרות עולה/ירדנות בפתרונות שונים וכו'. נשים לב שיש כאן עניין של זמן: אנחנו לא רוצים להתקדם רחוק מדי ואז להביא דברים שנוצר בעוד הרבה זמן במקום דברים שנוצר בקרוב - לכן כשמבאים נתון שנמצא רחוק יחסית מאייפה שנמצאים שם אותו רק ב-LLC ולא ב-MLC, כדי למנוע הוצאה של שורות שלローンטיות לכל המעבדים.

## Scalable Ring Interconnect

- **High bandwidth scalable ring bus**
  - Interconnects Cores, Graphics, LLC and SA
  - Composed of 4 bidirectional rings
    - 32 Byte Data ring, Request ring, Acknowledge ring, and Snoop ring
  - Fully pipelined at core frequency/voltage
    - bandwidth, latency and power scale with cores
  - Ring wires run over LLC without area impact
  - Distributed arbitration, coherency and ordering
- **Each Core/LLC can get data from two stations**
  - One connected to the “down” direction,
  - and one connected to the “up”
- **Ring access always picks the shortest path**
  - E.g., Core3 to LLC2 data uses the “up” stream in 1 hop
    - Rather than from the “down” stream in 7 hops
  - Ave. trip from Core to LLC is  $(0+1+2+3)/4 = 1.5$  hops



ה Ring בפועל לחבר את כל ה agents ב-core. הוא חייב להיות High Bandwidth ולעולם לא להוות את צוואר ה bottleneck במערכת - לא רצימם לעשות stall בגללו. ה-ring למשה מרכיב מרובה מסלולים דו כיווניים לכל ליבת: אחד להעברת מידע בתוך המעבד והוא ברוחב 32 בתים, אחד לביקש מידע מבוחץ, השלישי הוא כדי לסמן קבלה, ורביעי כדי לבצע snoops - סה"כ 4 רינגים דו-כיווניים.

ה-ring הוא מצור בקשר זה שהוא מחולק לשלבים ומהידע מתקדם בהם בכל מהזור שעון לומר בתדרות המעבד. כפי שהזכירנו הרוחב שלו עולה עם מס' הליבות (כך שכל שנוטיף ליבות יוכל לכתוב יותר מידע לפחות במחזר שעון אחד). ה ring עשי מרובה חוטים - הם לא תופסים הרבה מקום כי הם עוביים מעל ה LLC. כל ליבה יכולה לקבל data משתי התחנות הקרובות לה ב-ring.

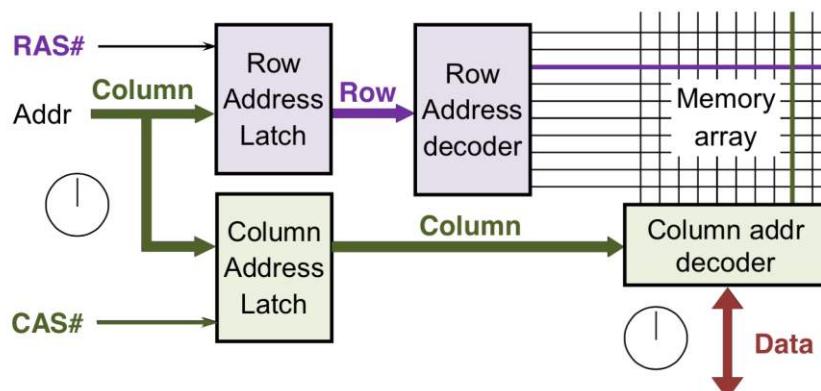
המידע נע באחד משני כיוונים על ה-ring, מעלה אומטה, וה-ring תמיד בוחרת ליבת את המידע דרין המסלול הקצר ביותר. בממוצע צריך לעבור 1.5 תחנות, hops, כדי לקבל את הנתון מקום כלשהו ב-LLC למאבד כלשהו.

- Each stop can pull one request off the ring per clock
  - Either from the "up" or from the "down"
- To avoid a case of data arriving to a given stop from both "up" and "down" directions at the same time
  - Each ring stop has a given polarity ("even" on "odd")
    - A ring stop can only pull data from the direction that matches that polarity
  - The ring changes polarity once per clock
  - The sender knows the receiver and its polarity is, and what the hop count is between them
  - By delaying sending the data for at most one cycle, a sender can assure the receiver can read it
    - and the case of two packets arriving at the same time never occurs

כדי להימנע מצב בו שני נתונים מגיעים לאותה ליבה בשתי התחנות שלה מוגדרת ל-ring זוגיות משתנה בכל מחזור שעון, וכן לכל תחנה מוגדרת זוגיות. ליבת יכולה למשוך data רק מתחנה שמתאימה לזוגיות של ה-ring שלה באותו מחזור. השולח מכיר את התחנה שמקבלת ואת הזוגיות של ה-ring וכן הוא יכול לתוכן בעצם את המסלול הקצר ביותר שנמצא ביניהם. אם עלולה להיווצר התנגשות הוא עושה delay one cycle, כלומר יכול להשאיר את ה-data למשך מחזור אחד כדי שנסתנכרן עם הזוגיות הנכונה.

## DRAM

### Basic DRAM chip



#### • DRAM access sequence

- Put Row on addr. bus
- Assert RAS# (Row Addr. Strobe) to latch Row
- Put Column on addr. bus
- Wait RAS# to CAS# delay and assert CAS# (Column Addr. Strobe) to latch Col
- Get data on address bus after CL (CAS latency)

DRAM זה התקן זיכרון זול וצפוף מאד. החיסרונו הגדול שלו הוא צורך צריך ריענון, refresh cycles, אחריו הוא עלול לאבד את המידע שהוא שומר. הוא DRAM הוא יותר איטי בהשוואה לזכרות SRAM שנמצאים בקאש. ב-DRAM כל תא מרכיב מטרניזטור אחד - טרנזיסטור זה התקן שיודע למלא קבל באופן מהזורי ("לרענן" את הקבל) ורק לשמר את המתח שלו. SRAM הוא סוג אחר של תא זיכרון בהם כל תא זיכרון מכיל שש טרנזיסטורים והוא יותר יקר ועמיד. בפועל DRAM הוא בעצם מטריצה שכל צומת בה הוא בית שמחזיק 0 או 1. בשביל לצמצם את מס' הפינים תכננו את-h-DRAM כך שלשם קריאה ראשית יש לציין מאייה שורה קוראים ואח"כ מאייה עמודה (בניגוד לציון המיקום המדוקיק של הביט בפעם אחת) - لكن גישה לוקחת יותר זמן ובזמן זה נדרש לעשות ל-DRAM ריענון. יש כל מיני מגבלות של זמן על העבודה מול התקן - למשל כמה זמן לחוכות בין הרגעים שמספקים לו את השורה לקריאה לזמן שמספקים לו את העמודה.

נסכם מה שצריך לדעת על זיכרון DRAM: זה זיכרון זול, צפוף, שיש פרוטוקול גישה כלשהו שיש לקיים מולו. זאת בניגוד ל-SRAM בו פשוט שמים כתובות ומקבלים את-h-data. כאמור הגישה ל-DRAM היא יותר איטית אבל המטריה העיקרית שלו היא שהוא זול.

השוואה בין זיכרות DRAM לזכרות SRAM:

## SRAM vs. DRAM

- Random Access: access time is the same for all locations

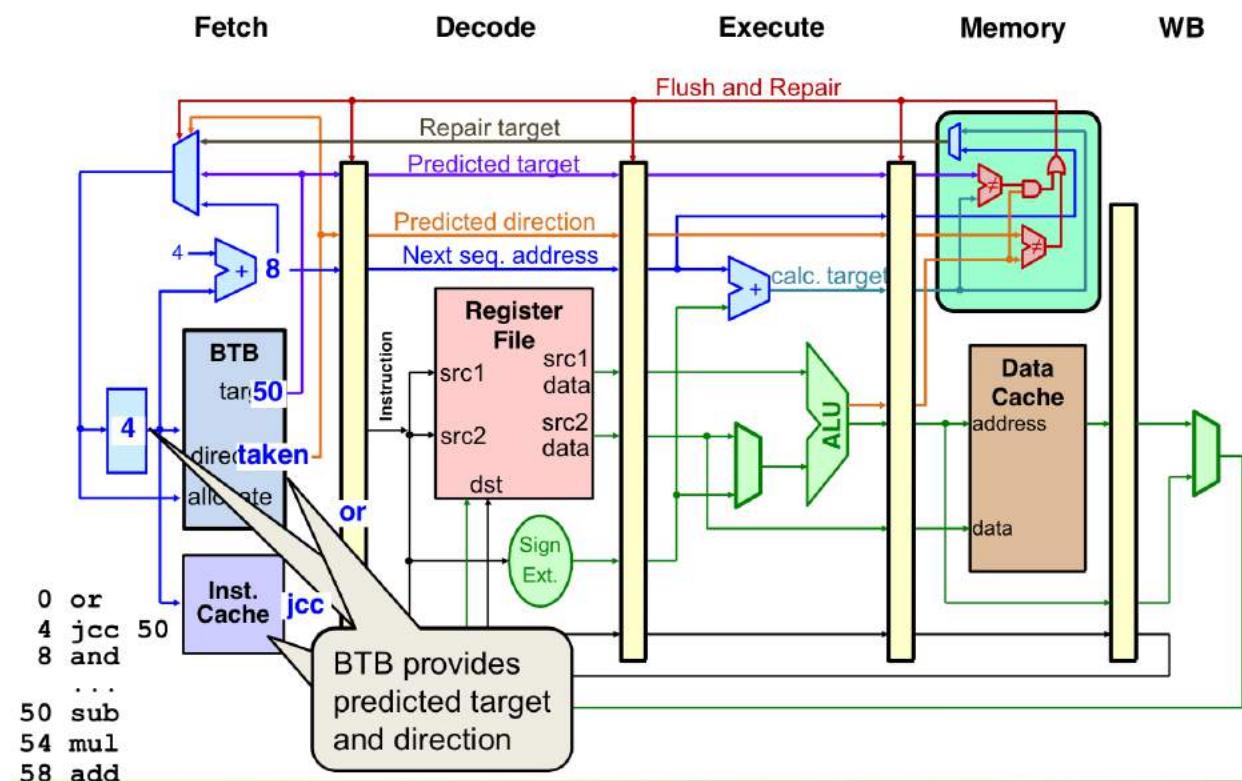
|                      | <b>DRAM – Dynamic RAM</b>  | <b>SRAM – Static RAM</b> |
|----------------------|----------------------------|--------------------------|
| <b>Refresh</b>       | Refresh needed             | No refresh needed        |
| <b>Address</b>       | Address muxed: row+ column | Address not multiplexed  |
| <b>Access</b>        | Not true "Random Access"   | True "Random Access"     |
| <b>density</b>       | High (1 Transistor/bit)    | Low (6 Transistor/bit)   |
| <b>Power</b>         | low                        | high                     |
| <b>Speed</b>         | slow                       | fast                     |
| <b>Price/bit</b>     | low                        | high                     |
| <b>Typical usage</b> | Main memory                | cache                    |

## הרצאה מס' 6: חיזוי קפיצות (Branch Prediction)

הערה: בהרצאה זו שנדרב על פקודות קפיצה הכוונה היא לפקודות של קפיצה-מוחתנית, branch, או קפיצה לא-מוחתנית, jump.

- תזכורת: בהרצאה בנושא Pipeline רأינו שלא חזאי-קפיצות נכונות לציור הפקודות שעוקבות אחריו פקודה הקפיצה ושה כמו כזו לחזות Not-Taken, כלומר רأינו שהחזות שהקפיצה לא תילך זה אותו דבר כמו לא לחזות בכלל כי בשני המקרים ממשיכים לבצע פקודות עוקבות אחריה הקפיצה. כמו כן למדנו ש כדי למנוע את הבעיה של הבאת פקודות לא-נכונות לאחר פקודה קפיצה, שמים (Branch Target Buffer) BTB (Branch Target Buffer) שמנסה לנחש את העתיד על סמך הערך, בפרט להגיד לנו האם תבוצע קפיצה בהתאם להיסטוריה. הקרייה ממנו היא בדרך"כ בשלב BTB במקביל לגישה ל הפקה, ככל עליים:
- שהוא אכן פקודה קפיצה (jump או branch כאמור).
  - לאן היא קופצת.
  - האם בפועל קופצת או לא (במקרה של פקודות jump תמיד תירשם שהיתה קפיצה).

## Adding a BTB to the Pipeline



בשוף רואים דוג' של תכנית שיש בה קפיצה (פקודה `jcc` בכתובת מס' 4), לפניו פקודה OR ואחריה מס' פקודות. לאחר ה-IP-OR מצביע לפקודה הקפיצה (אם כי בזיכרון עדין לא ידוע שזו קפיצה) ואז המעבד גם מביא את הפקודה מה-Instruction Cache וגם מחפש את כתובת 4 ב-BTB כדי לראות האם יש שם קפיצה. במידה ויש hit ב-BTB אז הפקודה היא אכן קפיצה והוא מוחזרת הכתובת שהייתה רלוונטית בפעם האחרונה שהיא הייתה במעבד, למשל אם הקפיצה קפזה ל-50 והחזאי חזק שהקפיצה אכן תילך אז BTB יקבע

אותנו מיד לכתובת 50 וכבר במחזור הבא נביא פקודות מכתובה זו. יחד עם הקפיצה סוחבים גם את הכתובה האלטרנטיבית, ככלומר אם חיזינו קפיצה אז סוחבים גם את הכתובה העוקבת לקפיצה (בדוג' כתובות מס' 8) וגם חיזינו אי-קפיצה אז את כתובת הקפיצה (בדוג' כתובות מס' 50). רק כשהפקודה מגיעה לשלבים מתקדמים יותר יודעים מה התוצאה האמתית של הקפיצה, מעדרנים את המערכות בהתאם ומשווים ביןיה לבין החיזוי כדי לראות אם הבאנו פקודות מהמקום הנכון. הבהרה: משווים גם את החיזוי וגם את היעד של הקפיצה. בדוג' שלנו חיזוי שהפקודה תילוך תמיד יהיה נכון (תמיד תהיה קפיצה והכתובה שלה קבועה) - בגלל שצדקו לא יקרה ב-*Decode* שבו דבר מיוחד. א' לעומת זאת אם החיזוי היה שגוי, (מצב זה נקרא *Miss-Prediction*) אז כל הפקודות שנכנטו לצינור אחרי הקפיצה לא היו הפקודות הנכונות ולא היה צריך להתחיל לבצע אותן - במקרה זה יש לבצע ריקון, *flush*, של הצינור של כל הפקודות עד *Fetch* ועד השלב בו זוהתה השגיאה (במקרה שלנו בשלב *the-h*-*Decode*, במעבדים מסוימים זה יכול להיות בשלב *Decode*) - ריקון משמעו מחיקה של כל הפקודות והבאת פקודות מחדש מהמקום הנכון.

נשים לב שבעוד החיזוי קורה בשלב *Fetch*-*BTB* העדכון של *Fetch*-*BTB* לגבי הקפיצה נעשה רק אחרי שבידינו כל הנתונים לגבי הקפיצה - במעבד *MIPS* זה קורה אחרי שלב *the-h*-*Execute*. זו הייתה הקדמה - עכשו נרחב על כל הנושא.

## BTB Overview

רוצים להבין איך ניתן לחזות את העתיד על סמך העבר וכמו כן מה בדיק ציריך לחזות. בשלב *Fetch* עדין לא ידוע לנו כלום על הפקודה, בפרט לא ידוע איזה פקודה הכנסנו למעבד, لكن פונים לחזאי ומקבלים ממנה חיזויים, *predictions*, כולם ניחושים האם הפקודה הנוכחית היא קפיצה, האם תקופוץ ולאן. החזאי נותן לנו את ההשערה שלו על סמך מה שעדכן בו בעבר.

מציר את הסוגים השונים של פקודות קפיצה:

- קפיצה מותנית (Conditional Branch) היא פקודה קפיצה שלעתים קופצת ולעתים לא כתלות בתנאי כלשהו.
- קפיצה לא-מותנית (Unconditional Branch) היא קפיצה שתמיד קופצת. גם קפיצות לא-מותניות מתחולקות לשניים:
  - Unconditional Direct - כתובת הקפיצה נתונה במפורש בפקודה, למשל בפקודת הקפיצה *call* שימושה בקריאה לפוני וכן פקודת *return* המשמשת לחזרה מפונ' (החזרה היא למקום האחרון ממנו בוצע *call*).
  - Unconditional Indirect - כתובת הקפיצה יכולה להשתנות, בפועל היא נמצאת ברגיסטר או בזיכרון. למשל קפיצה לערך שנמצא ברגיסטר 3 - כל פעם ניתן בו ערך אחר וכך יתכן שנkapופוץ למקומות שונים.

ב BTB יש את הרכיבים הבאים (נרחיב על כל אחד מהם בהמשך):

- רכיב שנראה *Target array* שתפקידו לספק לנו את סוג הקפיצה ואת יעדי הקפיצה - בהתאם לסוג הקפיצה אנחנו מחליטים באיזה רכיב של BTB להשתמש כדי לחזות את היעד (הכתובה שצריך לבצע לה). ה-*Target*-*Fetch*-*Array* שומר את כתובת הקפיצה גם עבור *Direct Branches*, ועבור קפיצות אלה-*the-h*-*Target array* כמעט תמיד צודק בזוגע ליעדי הקפיצה (כי הן קופצות לאותו מקום בכל פעם).
- התפקיד של הרכיב הבא לחזות את החלטות של פקודות ה-*Conditional Branch*. כאמור התוצאה שלhn תהיה ידועה רק בסוף ה-*Execute* והמעבד יחליט אותה על סמך דגל כלשהו כמו zero, carry ועוד שערכם נקבע עפ"י איזושהי השוואה. כמו כן נדרש לעשות חיזוי כבר בשלב *Fetch*. זה הרכיב החשוב ביותר ב BTB כי המחיר על טעות בחיזוי הוא גבוה ול-*Conditional Branches* נוכל לתקן את הטעות רק בסוף שלב ה-*Execute* בעוד בעוד ל-*Unconditional Branch* אפשר לדעת הכל בסוף שלב *Decode*, שכן הקנס על טעות עבר קפיצות

- モותנות הוא יותר גדול מקפיצות לא-מוותנות. יתרה מכך Conditional Branches אלה היג'אמפים הכי נפוצים - בתכנית ממוצעת 85% מהיג'אמפים הם Direct Jumps.
- הרכיב הבא בחזאי הוא ה-Return Stack Buffer שתפקידו לחזות את הכתובת אליה תוכון אותו פקודת Return Call ו-Return ובודת על המחסנית (הרגילה, זאת שמוצבעת ע"י הרגייסטר ESP): כל פעם שקובצים לשגרה דוחפים את כתובת החזרה על המחסנית ולכך כשוחזרים משגרה ווצים לקבל מהחסנית את כתובת החזרה אליה צריך לחזור. הבעיה היא שהחסנית נמצאת בזיכרון ובמעבד מודרני ניתן לזכור בשלב BTB-Memory לפחות עד לשלב BTB. למעשה, אצלנו במעבד זה נקרא שלב BTB-Memory.
- הרכיב האחרון הוא Fetch-ה-Register. Fetch-ה-Register אחראי המחסנית הגדולה מבחינת כתובות חזרה מפונקציות ומנסה לתת חזוי כבר בשלב Fetch-ה-Register.

במעבד שראינו עד כה Fetch-ה-Register הוא לחזות קפיצות לא-מוותנות - בראנטים שתלוים בערך ששמור בזיכרון או ברגיסטר. הפקיד של הרכיב האחרון הוא לחזות קפיצות לא-מוותנות - בשלבי Decode-ה-Register, Decode-ה-Execute ו-Execute-ה-Register.

הבהרה: השלב בו יודעים הכל על ה-branch הוא שלב Decode-ה-Register, יודעים בודאות שהפקודה היא קפיצה ואת הסוג שלה ואם היא קפיצה ישירה יודעים בשלב Decode-ה-Register את הכתובת וגם את החלטה (פקודה לא-מוותנית תמיד קופצת). בשלב Execute-ה-Register יודעים עבר קפיצה מוותנית האם תהיה קפיצה ומעבר קפיצה לא-ישירה יודעים בשלב זה את היעד - אם מגלים טעות בחיזוי באחד השלבים משלמים את העיכוב שנובע מריקון הצינור עד שלב הגילוי - ריקון עד Execute-ה-Register.

cut נרחב על כל אחד מחלקי החזאי שראינו:

## Target array

## The Target Array

- ◆ The TA is accessed using the branch address (branch IP)

- ❖ Implemented as an  $n$ -way set associative cache

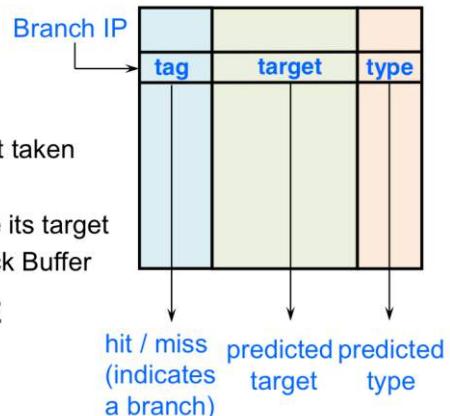
- ◆ The TA predicts the following

- ❖ Instruction is a branch
- ❖ Predicted target
- ❖ Branch type
  - Conditional: jump to target if predict taken
  - Unconditional direct: take target
  - Unconditional direct: if iTA hits, use its target
  - Return: get target from Return Stack Buffer

- ◆ The TA is allocated/updated at EXE

- ◆ Tags are usually partial

- ❖ Trade-off space, can get false hits
- ❖ Few branches aliased to same entry
- ❖ No correctness, only performance



ה-TA הוא 8-way associative cache, למשל 4-way, והוא חוצה את היעד של הקפיצה ואת כיוון הקפיצה (האם תילך/לא-תילך). נשים לב שעצם העובדה שיש ב-Target array בדר"כ אומרת שמדובר בפקודת קפיצה כי צ�ור האлокציה ל-BTB נועשית אחרי ה-Execute כשידעים הכל על הפוקודה, לכן אם מצאנו כתובת מסויימת סימן שהכתובות הוכנסה ל-BTB בשלב מוקדם יותר של הריצפה, זה קורה רק עבור קפיצות. יחד עם זאת יתכנו טעוויות וגם כישיש hit ב-Target array שהוא מושג בפקודת קפיצה: הסיבה העיקרית לטעוויות בזיהוי פוקודה כקפיצה היא שה-tag שה-Target array שומר הוא חלקי: במקרה של המעבד שמרנו tag מלא מכיוון שם הקash חייב להיות מדויק ולא יכולם לטעות ע"י הבאת הנtentן לתכנית מקום לא נכון, ב-DBTBT משמשים ב-tag חלקו, למשל שני בתים תחתוניים של כתובת הפוקודה, אז פקודות שונות יכולות לעשות hit לאותו tag ב-target array ובויהן פקודות שאין קפיצות. הסיבה שימושים ב-tag חלקו למרות הסיכון לטעוות היא חסכו במקומות - זה שיקול של עלות-תועלת, לא רוצים שהציף כלו יהיה מוקדש ל-Branch Predictor, צריך להשאיר מקום גם לשאר הדברים במעבד ולכך בעצם עושים פה trade-off. חשוב מאוד להבין שבניגוד לקash, התחום של Branch Predictor הוא לא פונקציונלי, כלומר יכול להיות לא-מדויק וטעויות כן באות בחשבון ובמקרה הגורע עושים ריקון וambilais פקודות מקום אחר - בכל מקרה נכוונות ה恬נית נשמרת (בניגוד להבאת נתון לא נכון מהמתמון ושימוש בו ע"י פקודות בתכנית - דבר שיותר קשה לתוך כשתמגלה הטעות). טעוויות של זיהוי פוקודה ב-Decode מעודכנות כבר בשלב Decode-ה. כי בו יודעים את סוג הפוקודה בוודאות. כדי שאמרנו טעוות שמתרגלה ב-Execute היא לא טעות יקרה כמו טעות בשלב שמתרגלה בשלב Decode, במיוחד במעבד מודרני בו Decode נמצא הרבה יותר מאוחר בזיכרון). בכל מקרה העובדה שניתנת לתקן את הטעות ע"י ריקון היא הסיבה לכך טעוויות בזיהוי ע"י ה-TA נאונות בחישוב Target array - וכך אפשר לחסוך מקום ע"י שמירת tag חלקו ב-Target array.

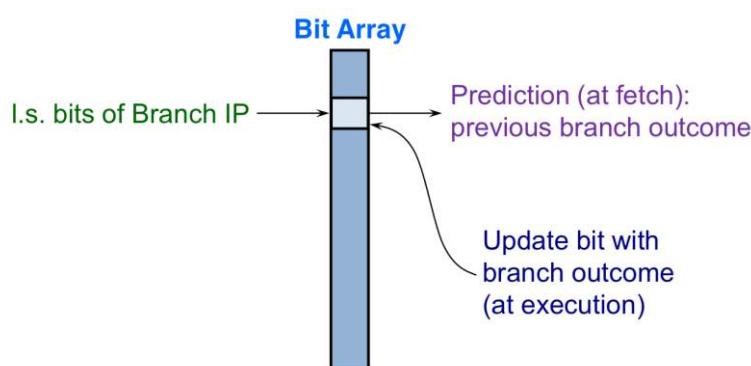
בפועל לא עורשים אлокציה לקפיצה כלשהו ב-BTB עד שהוא נלקחת (כלומר עד שהיא קופצת): יש לא מעט ג'אמפים בתכנית שאף פעם לא נלקחים ולכן לא עושים להם אлокציה - חבל להקצות מקום ב-BTB בשビルם כי גם

ככה תמיד צריך עבורם להביא פקודות עוקבות. פועל יוצא מכך הוא שבמקרה שבראנץ' נלקח רק בפעם העשירית שפוגשים אותו אז רק בפעם העשירית נשים אותו ב-BTB ומזה הפסדנו את היכולת למדוד את הבראנץ' עד שלב זה (נראה מיד מהכוונה ב"למוד את הבראנץ'"') אבל לפחות לא הקצנו משאבים ב-BTB שיכלו להיות לא בשימוש אף פעם - זו שאלה של רוחה והפסד, ממה נרווח יותר: מג'אמפיים שבסופה של דבר יהיו Taken ואז כבר תהיה לנו היסטוריה עבורם או מכך שבכלל לא עשו אлокציה של בראנץ'ים שהם אף פעם לא Taken והם לא יעיפו מה-BTB מידע על בראנץ'ים אחרים שכן יש צורך לטעוד אותם בחזאי. יש סבירות מאוד גבוהה שפקודת branch תהיה לנוכח לנצח והפסד בכך שדחיננו את הלמידה עליה הוא זנייה.

### Conditional Branch Predictor

זכור אמרנו שאנו רוצחים לחזות את העתיד על סמך הпроֹגְרָם. עכשו עוברים לרכיב החשוב ביותר בחזאי ותפקידו לחזות האם קפיצה-מוחנית נלקחת או לא. נתחיל מפתרונות פשוטים עבור חיזוי של קפיצה מוחנית ונשכלל אותן:

## One-Bit Predictor



חיזי מודול פשוט הוא מערכת שבכל כניסה שלו יש בית אחד ששומר את תוצאה הקפיצה הקודמת של הבראנץ' וחוצה על-פייה: אם הבראנץ' קופץ בפעם האחרון הביט היה 1, משמע שבפעם הבאה שנפגש את הבראנץ' נחזה קפיצה שוב, ואם הוא לא קופץ בפעם האחרון אז הביט יהיה 0 ובפעם הבאה נחזה אי-קפיצה. כאמור מזהים קפיצה כלשהי בחזאי באמצעות הסיבות התחתונות של כתובת הקפיצה, למשל אם ב Target array שומרים את עשר הסיבות התחתונות של ה-IP אז באמצעות פונטム ל-BTB - אך במקרה זה יש  $2^{10} = 1024$  כניסה בחזאי ולכל אחת מתאימה כניסה ב-1-Bit Predictor 1 שמכילה בית בודד שיגיד האם החיזי חזה שהבראנץ' י קופוץ או לא. נקודה חשובה לציין היא שלא בודקים את החלטת החיזי כל עוד ה-target array לא זיהה את הפקודה קפיצה ופרט קפיצה מוחנית (זה נעשה לפי השדה Type שקיים בכל שורה ב-target array). אז החיזי משתמש רק עבור דנים בו כתה מהו ערך הביט, זה החיזי, ובהתאם לחיזיו מחליטים האםLKופוץ או לא.

### Problem: 1-bit predictor has a double mistake in loops

|                |                                       |
|----------------|---------------------------------------|
| Branch Outcome | 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1   |
| Prediction     | ? 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 |

לחזות תמיד את מה שהיה בפעם הקודמת זו גישה מאוד בסיסית למראנט'ים שהם כל הזמן Taken (זכור במראנט'ים שאינם Taken אף פעם בכלל לא יכנסו לטבלה): נניח לדוגמה שבראנט' נלקח 10,000 פעמים ואז בפעם האחרונה הוא לא נלקח ובכך התכנית מסתתרת - אז מתווך 10,000 פעמים החזאי יהיה כודק 9,999 פעמים ובפעם ה-10,000 הוא יטעה (ואולי עוד פעם אם במקרה הראשונה חיזנו שלא תהיה קפיצה ובפועל הייתה). אז אחות החזוי של חזאי זה היא  $9999/10000$ , זה שיעור חזוי פנטסטי, וכך החזוי של Predictor 1-bit עברו בראנט'ים שתמיד יש להם את אותה תוצאה הוא מצוין, לעומת זאת ברגע שמגיבים לlolאות יותר קוצרות שמתבצעות שוב ושוב אז החזוי הוא כבר פחות טוב כי בכל פעם שהlolאה מתבצעת יש שגיאה אחת בסופה ותיתקן גם אחת בתחילתה וכאמור מס' האיטרציות בכל הריצה של lolאות הוא קטן. דוג':

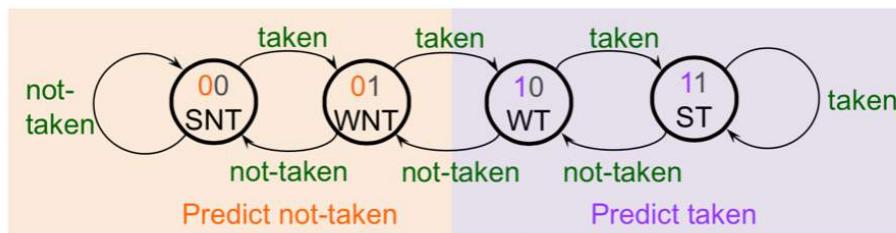
עבור lolאה שנלקחת חמיש פעמים ובפעם הששית לא נלקחת נוצרת תבנית של Chash Not-Taken אז ייחיד, נסמן זאת ע' 111110 111110 11110 11110 כרך lolאה... הפרדיקטור שדיברנו עליו עד כה פשטוט חוצה את התוצאה הקודמת של הקפיצה ולכן יראה נכון עד שמגיב Not-Taken באיטרציה האחרונה, בה הוא יטעה, וכשיתחיל סבב חדש של קפיצות הוא יטעה שוב כי במקרה שלנו ולא קפוץ ולא נקפוץ למחרות שכן נקפוץ כי התחילו סבב קפיצות חדש. אז בכל סבב החזאי טואה פעמיים: פעם אחת לא מוחש נכון את Ai-הקפיצה ששוברת את הסיבוב הנוכחי של lolאה ובפעם הבאה לא מזהה שנכנסנו לסיבוב נוסף וכן צריך לקפוץ.

cutת נראה חזאי משופר שטועה רק פעם אחת בכל הריצה של lolאה.

## Bimodal (2-bit) Predictor

### ♦ A 2-bit counter avoids the double mistake in glitches

- ❖ Need "more evidence" to change prediction



- ❖ Initial state: weakly-taken (most branches are taken)

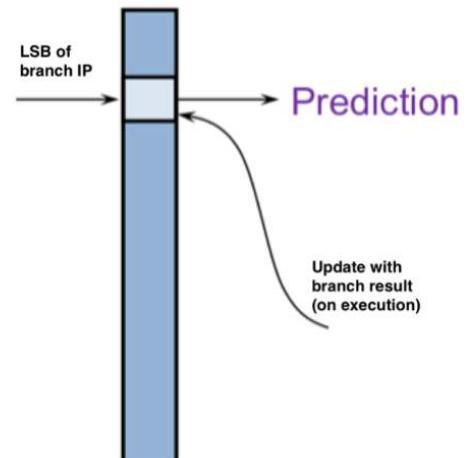
### ♦ Update (at execution)

- ❖ Branch was actually taken: increment counter (saturate at 11)
- ❖ Branch was actually not-taken: decrement counter (saturate at 00)

### ♦ Predict according to m.s.bit of counter (0 – NT, 1 – taken)

### ♦ Does not predict well branches with patterns like 010101...

### 2-bit-sat counter array



סוג החזאי הבא נקרא Bimodal Predictor או 2-Bit Predictor ("חזקאי דו מחייב"). הפעם בעזרת כתובת הקפיצה ניגשים למערך שכל תא בו מכיל מכונת מצבים עם ארבעה מצבים, כלומר בשבייה צרי 2 ביט. המצביעים הם:

- 00 - Strongly Not-Taken
- 01 - Weakly Not-Taken
- 10 - Weakly Taken
- 11 - Strongly Taken

החזקאי חוצה Not-Taken באחד משני המצביעים הראשוניים ו-Taken באחד משני האחרונים - במילים אחרות הוא חוצה לפי ה-MSB של המצביע הנוכחי. מכונת המצבים עצמה היא מונה שסופר למעלה אומטה בהתאם להאם

הבראנץ' נלקח אבל מגיעה לרוויה כשהוא בגבולות כלומר ניתן לעלות מ 10 ל-11 (במקרה זה לאחר שהבראנץ' נלקח) אך ב-11 אם הבראנץ' יילקה שוב מכונת המחשבים תישאר באותו מצב. כל פעם שרואים בזמן-h Execute-stronglyTaken מנקחים מעליים את המונה באחד וכל פעם שרואים שהוא לא נלקח מורידים את המונה באחד, لكن למשל שהג'אמפ נלקח מעליים את המונה באחד וכל פעם שרואים שהוא לא נלקח מורידים את המונה באחד, אך לעומת זאת מוגע Taken המכמה הגיעו למצב 11 ותישאר שם.

היתרון במונה כזה הוא שהוא דורש "שכנוע" כדי לשנות את החלטתו לגבי החזאי הבא, למשל לעבור מחייב של Taken Taken Taken Not-Taken. עכשו בדרכ' הקודמת של הלולאה החזאי יראה כל הזמן.. Taken Not-Taken Strongly Taken אז הוא לא ישתכנע כל כך ביכולות כלומר יעבור ל-Taken ובפעמים הבאות ימשיך לחזות - CUT החזאי יטעה רק פעמי אחת, ביציאה מהלולאה, אך לא יטעה גם בכניסה החזרה.

מסתבר שהשיפור של החזאי כך שהוא ידרש שכנוע כדי לעבור ממצב של קפיצה לא-קפיצה ולהיפך, ולא ישתכנע ע"י גלי"ץ' חד פעמי, משפר פרדיקציה בצורה מאוד שימושית בכל מיני מקרים נוספים - לאו דוקא במקרים של לולאה. דוג' נוספת היא שהרבה פעמים יש לנו if then else if שרוב הזמן לא קוראים, למשל האם הגובה של בן אדם הוא מעל 185 ס"מ - רוב הזמן זה לא נכון וmdi פעם כן - אך אם mdi פעם יש גלי"ץ' ומישהו גבוה יותר מגיע אז עברו החזאי יטעה אבל על המקרים הבאים לא. באופן כללי יש המון פעמים תבנית נפוצה של קפיצות וmdi פעם יש ממנה גלי"ץ' - עדיף לטעות על הגלי"צים' האלה רק פעם אחת.

## Bimodal Predictor - example

### ◆ Br1 prediction

- ❖ Pattern:
  - ❖ counter:
  - ❖ Prediction:
- |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 2 | 1 | 0 | 1 | 0 | 1 | 0 |
| ↓ | ↑ | ↑ | ↑ | ↑ | ↑ | ↑ |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Code:

int n = 6

Loop: ....

br1: if (n/2) { ... }

br2: if ((n+1)/2) { ... }

n--;

br3: JNZ n, Loop

### ◆ Br2 prediction

- ❖ Pattern:
  - ❖ counter:
  - ❖ Prediction:
- |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 2 | 0 | 1 | 0 | 1 | 0 | 1 |
| ↓ | ↑ | ↑ | ↑ | ↑ | ↑ | ↑ |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |

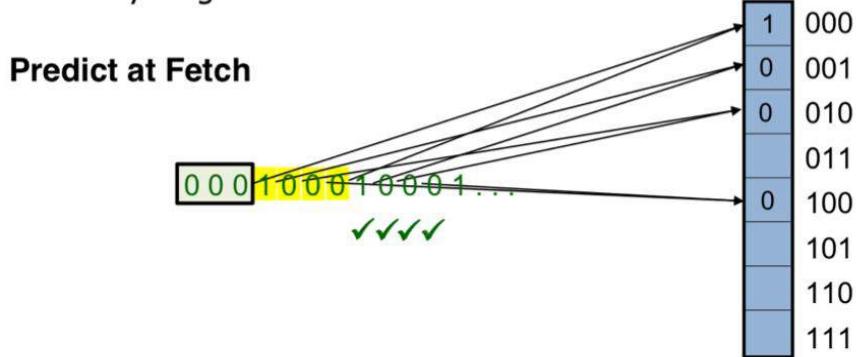
### ◆ Br3 prediction

- ❖ Pattern:
  - ❖ counter:
  - ❖ Prediction:
- |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 2 | 1 | 1 | 1 | 1 | 1 | 0 |
| ↓ | ↑ | ↑ | ↑ | ↑ | ↑ | ↑ |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |

עוד דוג' לדוגמה Bimodal Predictor - ביצועי הסדרה 010101 תלויים מאוד במצב ההתחלתי של החזאי: אם המצב ההתחלתי הוא Strongly Taken אז בפעם הראשונה שנראה 0 נעבור ל-Weakly Taken ואחריו כן ב-1 נחזור ל-Strongly Taken וכך נ חוזה כל הזמן Taken ואחוז החזאי שלנו יהיה 50%. זה אחוז-חזוי לא מוזהיר (אפילו גרווע - זה למשה מה random היה עשה). יכול להיות אפילו יותר גרווע: אם התחלנו במצב של Weakly Taken או חזינו 1 ואיז ב-0 עברנו ל-Weakly Not-Taken ואיז חזינו 0 וב-1 חזרנו ל-Weakly Taken וכך הלאה - וזה נ חוזה 100% מהזמן. סה"כ רואים שלסדרה מזגגת לחזאי bimodal יהיה 50% או 100% חייזרי שגווי ולכן הוא לא משווה בשביל תבניות כאלה.

## 2-Level Branch Prediction

- ◆ **Save branch direction history in a Branch History Register**
  - ❖ A shift-register that saves the last  $n$  outcomes of the branch
  - ❖ History points to an array of  $2^n$  bits
  - ❖ Bit pointed by a history specifies the predicted branch direction following that history
- ◆ **Example: predicting the pattern 0001 0001 0001 ...**
  - ❖ History length  $n=3$



מה שהיינו באמת רוצים זה חזאי שיעודו לחזות דפוס מסוים של בראנץ', למשל שעבור התבנית 00101 00101 00101 ידע לזהות אותה, ללמידה אותה ולקפוץ בה רק מתי צריך. לפניו כ-40 שנה התפרסם מאמר בו מצאו חזאי יעיל מאוד שיעודו לחזות מבנים. חזאי זה, שנקרא 2-level Predictor, משתמש ברגיסטר ששמור את ההיסטוריה של קפיצה - כל בית בהיסטוריה מסמן האם היא נלקחה או לא כך שרצף ביטים באורך  $n$  מלמד אותנו על התוצאות האחרונות שבראנץ' כלשהו עשה. אותה היסטוריה (בפועל הערך השמור ברגיסטר) משומשת Caindקס במערך שמחווה על החיזוי הבא עבורה. בשקף: ההיסטוריה זוכרת מה קפיצה כלשהי עשתה בשלוש הפעמים האחרונות והיא מצביעה למערך של ביטים שיגיד לנו את החיזוי בעקבות אותה היסטוריה - אותו מערך יכול להיות בפועל 2-bit או 1-bit. כדי למדנו עד כמה אך בסופו של דבר הוא מספק חיזוי של בית אחד (נלקח/לא-נלקח) על סמך ערך ההיסטוריה. בשקף רואים איך החזאי חוזה את התבנית 0001 בעקבות היסטוריה באורך שלוש נלקחות: לאחר שההיסטוריה היא 000 החזאי צריך לנחש שהג'אמפ יקפוץ, لكن Caindקס ה-000 במערך נזכיר 1. ההיסטוריה הבאה שהחזאי יראה עבורי בראנץ' זה ה-001, אחראי רוצים לזכור שלא תחיה קפיצה כלומר להיסטוריה מצטרף מצד ימין עוד 0. בפעם הבאה ההיסטוריה של הבראנץ' תהיה 010 ועבורה יש לזכור לא לחזות, אחורי כן ההיסטוריה תהיה 100 שגם עבורה צריך צריך לחזות קפיצה. מרגע זה והלאה ההיסטוריה מתחליה לחזור על עצמה, כלומר מייפינו את כל התבניות האפשריות שנראה עבורי הבראנץ' זהה ואם בכל פעם נשמר את הבית לפי הקפיצה הקודמת של אותה היסטוריה אז החזאי יספק חיזוי נכון בכל החיזויים שלו עבורי הבראנץ' זהה. אנחנו רואים שלא כל המערך שמספק חיזוי משומש כי יש הרבה רצפים שלא גונע אליהם, למשל התבנית 011 לא מופיעה בסדרה, בפרט אחרי התבנית 100 אנחנו חוזרים ל-000 שכבר ראיינו אותו ועכשו נזוז בمعالג סיבוב ריבועית הרצפים שראינו. לכן ככל שנשמר ההיסטוריה יותר רחבה המערך יהיה יותר גדול ואף יותר דليل, sparse - لكن פחות יעיל.

הבהרה: עד שלא נבעור פעם אחת על כל סדרת התבניות הרלוונטיות לפקודת קפיצה כלשהי יתכן ונקבל חיזויים לא נכוניים, אבל אחרי שנעביר עליה פעם אחת וכל פעם נעדכן את החיזוי שלנו לפי התוצאה של הבראנץ' עבורה (האם נלקח/לא-נלקח) אז בפעם הבאה שנראה את אותה התבנית כן נראה חיזויים נכוניים.

זה בעצם מקור השם Branch Predictor 2-Level: הrama הראשונה היא הרגיסטר שזוכר את ההיסטוריה והrama השנייה זה המערך שמאנדקסים באמצעות הרגיסטר, ככלומר ההיסטוריה מצביעה על מערך והמערך חוזה את הכוון עברו כל היסטוריה.

החזאי הכי ייעיל מסוג 2-Level Branch Predictor הוא זה שאורך מינימלי וחוזה ב 100% את הפטן. הוא הכי

יעיל משתי סיבות:

- זמן החימום שלו יהיה הכי קצר.
- הוא תופס הכי מעט מקום.

אנחנו רוצים לשמר היסטוריה ברגיסטר מהאורך הכי קצר שעדין חוזה נכון - בוואנו נראה מהו החזאי הכי קצר שיכול לחזות נכון את התבנית 100111:

◆ What is the shortest history needed to perfectly predict the pattern 100111 ... in steady state ?

❖ A history of length 3 predicts correctly:

- 100111 100111      100 → 1
- 100111 100111      001 → 1
- 100111 100111      011 → 1
- 100111 100111      111 → 0
- 100111 100111      110 → 0

❖ A history of length 2 is wrong twice per iteration

- 100111 100111      10 → 0
- 100111 100111      00 → 1
- 100111 100111      01 → 1
- 100111 100111      11 → 1
- 100111 100111      11 → 0

: ראשית נשים לב שההיסטוריה באורך חמישה סיביות בטוחה תפיק ואין טעם לשמר יותר מזה (כי אין בעיה לזכור את התבנית הזאת ע"י חישבה ביטים). כתע נראה שמספיק לזכור גם שלוש סיביות:

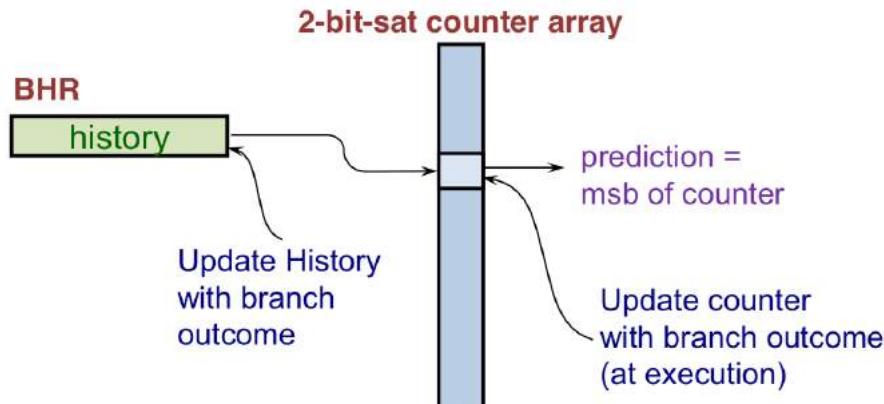
- אחרי 100 צריך 1
- אחרי 001 צריך 1
- אחרי 011 צריך 1
- אחרי 111 צריך 0
- אחרי 110 צריך 0
- אחרי 100 צריך 1
- וחזר חלילה..

נשים לב שבדוגמה זו עם היסטוריה באורך שתי-סיביות לא נוכל לספק חיזוי מושלם כי אמנם תמיד אחרי 10 בא 0 אבל עברו התבנית 11 לעיתים צריך לחזות 1 ולעתים צריך לחזות 0, לכן ברגע שני אני אראה את 11 כתוב לחזאי 0 וואז פעם הבאה שאראה 11 אטעה ואצטדרך לכתוב 1 וכך הלאה, ז"א ההיסטוריה באורך 2 לא מגדירה במדויק מה יבוא אחרי כל אחד מהרצפים. לעומת זאת בשמירת ההיסטוריה באורך 3 כל רצף בתבנית של הבראנץ' הזה הוא ייחודי ולכן תמיד נוכל לחזות אחרי את החיזוי הנוכחי. נשים לב שלא-הכרחי שככל רצף אפשרי של ההיסטורית הבראנץ' יהיה ייחודי אלא רק שהחיזוי הנוכחי עבורו יהיה תמיד אותו חיזוי - למשל יכול להיות ש-100 מופיע בשני מקומות שונים בתבנית של הקפיצה ואם בשנייהם אחורי תופיע אותה תוצאה, למשל 0, זה עדין יהיה בסדר. אז ראיינו התבנית שאורכה חמישה סיביות אבל היסטוריה באורך שלוש סיביות מספיקה כדי לחזות אותה ב 100%

עוד דוג': התבנית המאוד פשוטה של הרבה 0 ואחריה 1 בודד, למשל 1000000000, דורשת היסטוריה די ארכוכת - כארוך רצף האפסים (כי אחרי 0 יש לחזות 1 ואחריו שאר התבניות - 0).

#### ◆ There could be glitches in the pattern

- ❖ Example: 00001 00001 00001 01001 00001 00001
- ❖ Use 2-bit saturating counters instead of 1 bit to record outcome:



#### ◆ The longer the history

- ❖ Warm-Up is longer
- ❖ Counter array becomes very big (and sparse)

נשפר את החזאי שלנו כך שבמקרה שבו על החיזוי יהיה בית אחד נשים בו שני ביטים ומשתמשים בכל תא במכונת המ מצבים שראינו מקודם (ה Bimodal Predictor) ואז בעצם משלבים את החזוק של חיזוי על-פי ההיסטוריה עם החזוק של עמידות לגליץ' ואז במקומות לשלם בשגיאה כפולה על כל גליץ' בתבנית כלשהי נשלם רק על שגיאה בודדת. ייחד עם זאת החזאי הזה יקר מאוד כי עבור ההיסטוריה בגודל ח צריך לשמר  $2^{10} \times 2^6$  (ח<sup>10</sup> \* 2<sup>6</sup>) סיביות וכל זה כדי לטפל בבראנץ' ייחד (מכונה בגודל שני-בית עברו כל ערך של ההיסטוריה, עברו כל קפיצה), לכן עבור טיפול ב-1024 בראנץ'ים וההיסטוריה בגודל 6 החזאי אפילו עם שמונה ביטים של tag בלבד, יהיה גדול

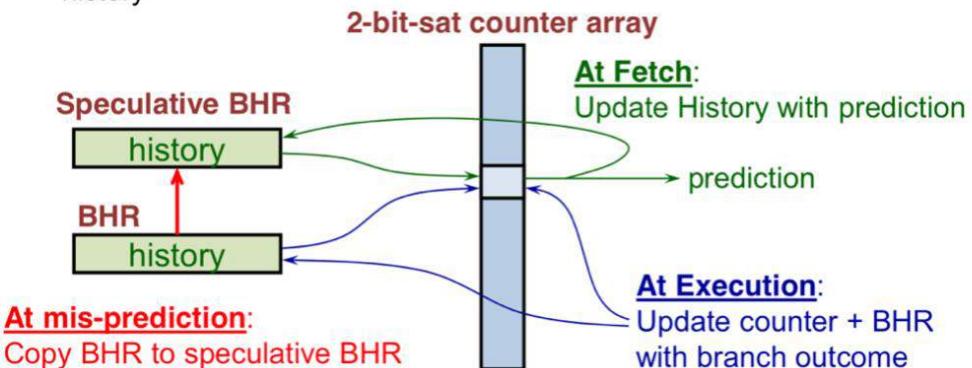
מאוד:

$$| BTB | = 2^{10} * [14 + 128] = 142KB$$

לכן בפועל בחזאי level-2 זהה אי אפשר לשמור היסטוריות ארכוכות. הבירה: במנגנון של חזאי level-2 בשילוב עם bimodal כל כניסה במערכת ההיסטוריה היא מכונה של ארבעה מצבים.

## Speculative History Updates

- ◆ Deep pipeline  $\Rightarrow$  many cycles between fetch and branch resolution
  - ❖ If history is updated only at resolution (branch execute)
    - Future occurrences of the same branch may be fetched before history is updated, and predicted with a stale history
  - ❖ Update history speculatively according to prediction
- ◆ If a branch is mispredicted
  - ❖ History continues to be updated until the bad branch gets to execution
  - ❖ Branches following it use a wrong history for their prediction
    - But this does not matter, as they will be anyhow flushed
  - ❖ Need to recover the branch history to its value before the bad branch
    - Maintain a non-speculative copy of the history, updated at execute
    - In case of misprediction, copy non-speculative history into speculative history



אמרנו שמעדכנים את החזאי בשלב Execute כי רק אז יודעים אם הוא באמת כפץ ולאן. הבעה היא שהעדכו קורה עמוק ב-Pipe והחיזוי בשלב מוקדם - במעבד מודרני יכולים לעבור عشرות מחזוריים ביניהם. יתרון שעד שаг'אמפים יגעו ל-Execute ונדען את ההיסטוריה שלהם אولي נעשה להם Fetch שוב ואז נזהה ג'אמפים לפי אותה היסטוריה שלא עדכנה, כלומר לפי תבנית לא-עדכנית - لكن אי אפשר לחכות ולעדכן את החזאי לגבי הג'אמף רק ב-Execute. שוב: הבעה היא שאם לא נעדכן ספקולטיבית את ההיסטוריה אז הגיעו עוד ג'אמף מאותה פקודה הוא לא ידע להסתכל על התבנית הנכונה של ההיסטוריה.

לכן נעדכן ספקולטיבית: מה שעשנים הוא שבפועל מעדכנים את ההיסטוריה באותו זמן שעשנים את החיזוי. נראה מודיע זה חוקי: ישנו שני מקרים לגבי תוכחת החיזוי: חיזוי נכון או שגוי. אם הוא נכון אז העבודה שהשתמשנו בפרדיקציה כדי לעדכן את ההיסטוריה היא בסדר גמור כי כך היינו מעדכנים את ההיסטוריה בכל מקרה. לחופין אם הפרדיקציה הייתה לא נכונה אז שנדרחו אותה להיסטוריה זה יוצר בלאן עם החזויים העוקבים שראינו עד בשלב Execute ואז נביא לפיעוף פקודות לא נכוןות, אבל גם במקרה זה עושים Flush לכל מה שבצינור אחרי העדכן השגוי הראשוני ובפרט לאותם בראנץ'ים שחזינו באמצעות היסטוריה לא נכוןה. לכן בשני המקרים אנחנו בסדר: אם החיזוי נכון אז טוב שעדכנו אותו מוקדם ואם לא אז עושים ריקון צינור ולכן לא משנה מה הפקודות שהגיעו לפיעולן.

לכן מה שעשנים בפועל הוא להסתכל על ההיסטוריה בזמן fetch, לפניות עם ההיסטוריה אל המערך, לקבל פרדייקציה ולעדכן את ההיסטוריה עם הפרדייקציה - כך ההיסטוריה שלנו תמיד מעודכנת ולא צריך לחכות עד שנגיעה לשלב Execute כדי לעדכן אותה. אם הגיענו לשלב Execute וקייםנו Miss-prediction, כלומר התברר שההיסטוריה העדכנתה ע"י פרדייקציות לא נכוןות או צריך לשחרר את ההיסטוריה לנוכח האخرונה שהיא היתה נכוןה.

נסביר שוב את הבעיה: באיזשהו שלב התחלו לדוחף חיזויים לא נכונים ומאז דחפנו עוד חיזויים, لكن ברגע שמקבליםים את ה Miss prediction צריך לשחרר את ההיסטוריה במצב שהיתה לפני העדכן השגוי הראשון ועלדען אותה עם התוצאה האמיתית של הבראנץ' - כדי לעשות זאת מעבירים בפייפ ייחד עם כל בראנץ' את ההיסטוריה שהיתה לפני העדכן עברו ואז ב-Execute אם הג'אמפ מתברר להיות Miss-predicted אז נחזיר את ההיסטוריה לקדמותה והפעם נוסיף לה את הבית שמתאים לחיזוי הנכון.

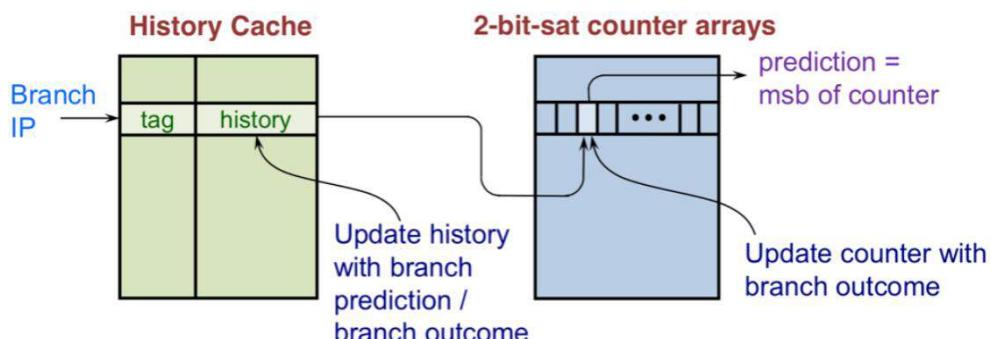
#### ◆ The counter array is not updated speculatively

- ❖ Prediction can change only on a misprediction
  - state 01→10 or 10→01
- ❖ Counter array is too big to be recovered following misprediction
  - Too expensive to maintain both speculative and non-speculative copies of the counter array

מה לגבי המערך של המונחים שמספק את החיזוי בהתאם להיסטוריה, האם גם אותו ניתן לעדכן בצורה ספקולטיבית ע"י החיזוי ואז לשחרר אותו? התשובה היא לא, כי אין דרך קלה לשחרר אותו, הוא ענק ומה לא שהוא שางר ללחוב יחד עם הפקודה בפייפ ואז לעדכן חזרה. לכן את ההיסטוריה מעדכנים בצורה ספקולטיבית אבל את המערך של המונחים אי אפשר לנctrar לשימוש במא שיש שם בתוקוה שתתבנית חזרת על עצמה. נבהיר למה זה בסדר: ברגע שלמדנו פעם אחת את הסדרה של ההיסטוריה או בעצם האינפורמציה שיש בהור המערך של המונחים היא הנכונה, וכל פעם שההיסטוריה תצביע לאיזושהי מצבים נקבל את מה שציריך עברו אותה היסטוריה בהנחה שאין איזשו גלי' ב התבנית - מה שקרה לעיתים רחוקות וגם ככה החזאי שלנו עמיד אליו אם משתמשים ב predictors

## Local Predictor: private counter arrays

Holding BHRs and counter arrays for many branches:



Predictor size: #BHRs × (tag\_size + history\_size + 2 × 2<sup>history\_size</sup>)

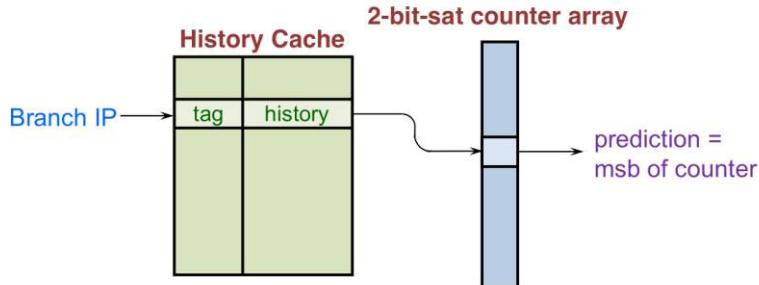
Example: #BHRs = 1024; tag\_size=8; history\_size=6 ⇒

$$\text{size} = 1024 \times (8 + 6 + 2 \times 2^6) = 142\text{Kbit}$$

בדוג' לחישוב הגודל של החזאי רואים שגם עבור היסטוריה קטנה יוצא שהוא גדול מאוד. כתעת נראה כל מיני אלטרנטיבות שהופכות אותו ליותר קומפקטי.

## Local Predictor: shared counter arrays

- ◆ Using a single counter array shared by all BHR's
  - ❖ All BHR's index the same array
  - ❖ Branches with similar patterns interfere with each other
    - Interference can be either constructive or destructive



Predictor size: #BHRs × (tag\_size + history\_size) + 2 × 2<sup>history\_size</sup>

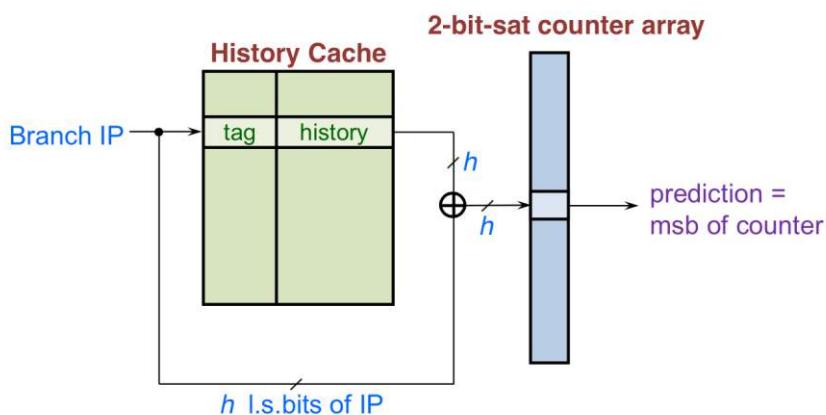
Example: #BHRs = 1024; tag\_size=8; history\_size=6 ⇒

$$\text{size} = 1024 \times (8 + 6) + 2 \times 2^6 = 14.1\text{Kbit}$$

בדוג' מועל רואים חזאי עבورو כאשר נקבע ג'אמפ נפנה אל-target array ואם זה בראנץ' אז נפנה עם ההיסטוריה זהו למערך של מכוניות המצביעים אבל הפעם כל הפעם כל ההיסטוריה מעדכנות את אותו מערך של מונויים. הדבר הזה יעבד בצורה סבירה כשייש לבראנץ'ים שונים תבניות דומות. במקרה זה כשמי בראנץ'ים ממופים לאוותה כניסה זו לא בהכרח יהיה בעייתי - יכול להיות שכולם רוצים חייזרים דומים עבור אותן תבניות. בנוסף ראיינו שעבור ההיסטוריה ארוכה המערך של מכוניות המצביעים, ככלומר המערך שמאנדקס עפ"י ההיסטוריה יהיה יותר דليل - אם הוא יהיה מספיק דليل או גם אם ההיסטוריה של בראנץ'ים שונים אז לא יהיה הרבה התנגשויות.

## Local Predictor: Ishare

**Ishare** reduces inter-branch-interference in the counter array:  
maps common patterns in different branches to different counters



Predictor size: #BHRs × (tag\_size + history\_size) + 2 × 2<sup>history\_size</sup>

כדי לוודא שייהיה כיסוי טוב של המערך משתמשים ב-hash function על ערך ההיסטוריה ובאמצעות התוצאה ניגשים למערך שמספק את החיזוי. נרצהשה-shah תשלב את ההיסטוריה והכתובת של הbraanç' (כדי שכתובות שונות ימודפו לתאים שונים), לכן נשתמש ב-XOR ביןיהן, שהוא hash טוב, ומגיעים עם התוצאה שלו לאינדקס במערך - זה יוצר פיזור טוב ולכן יהיו פחות קונפליקטים במערך מספיק גדול. הרעיון של לחת XOR דואג שגם

שני בראנץ'ים שונים שיש להם את אותו תת-רצף לא יעדכו את אותו אינדקסים במערך כי ה-XOR בין הכתובת ההיסטורית יהיה שונה בשני הבראנץ'ים. בנוסף לשימוש ב-XOR בפועל אין בתוכנית הרבה ג'אמפים שחוצים ביחס בלבד, כלומר לא הרבה מערכנים את המערך בו-זמן, למשל בביצוע תוכנית עם לולה פשוטה אז רק קפיצה אחת מעדכנת את המערך. גם אם בתוך הלולאה יש לולה נוספת נספה כנראה שהתובנות של שני הג'אמפים הפעילים.

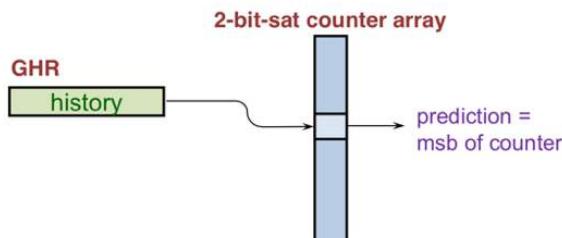
## Global Predictor

- The behavior of some branches is highly correlated

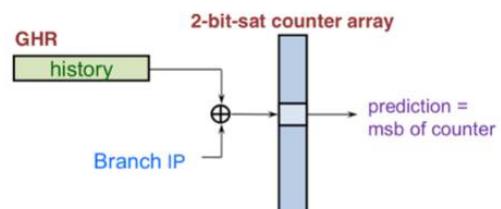
```
if (x < 1) ...
if (x > 1) ...
```

- Using a single Global History Register (GHR) for all branches

❖ The prediction of the 2<sup>nd</sup> *if* is influenced by the direction of the 1<sup>st</sup> *if*



- gshare* combines global history information with the branch IP



❖ The counter accessed is a function of the global history and of the specific branch being predicted / updated

❖ This turns out to be extremely accurate and space-efficient

- History interference between non-correlated might hurt prediction

- With only a single history, can afford a long history

❖ The predictor size:  $\text{history\_size} + 2^{\star}2^{\star}\text{history\_size}$

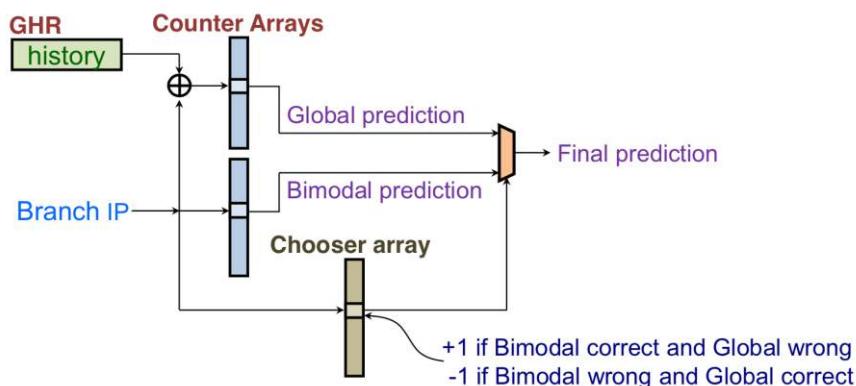
כדי לחסוך יותר מקום אפשר להשתמש ברגיסטר אחד עבור ההיסטוריה של כל הבראנץ'ים ככלומר לא לזכור היסטוריה נפרדת לכל בראנץ' אלא רגיסטר אחד שזוכר את ההיסטוריה האחורה של כל הקפיצות. חזאי כזה נקרא "חזאי גלובלי" וההצדקה לשימוש בו היא שההיסטוריה אוצרת בתוכה מידע על ריצת התוכנית עד ההגעה לשלב של בראנץ' כלשהו, ככלומר ההיסטוריה זוכרת את המסלול שדרכו הגיעו לג'אמפ זהה, מה שאומר שהחיזוי יתבסס על איזשהו חישוב שקרה בתוכנית עד ההגעה לבранץ'. מסתבר שזה נותן חיזוי חזק מאד. לחזאי הקודם שראינו, בו לכל פקודה יש מערך של מכונות מצבים משלה, קוראים local predictor בעוד לחזאי בו כל הבראנץ'ים מאנדקסים את אותו מערך של היסטוריות קוראים כאמור global predictor

.global predictor

## Chooser

- A chooser selects between 2 predictors for each branch

- Use the predictor that was more correct in the past
- The chooser array is an array of 2-bit saturating counters, indexed by the Branch IP (similar to the Bimodal array)
- Updated by which predictor is more correct



איך מושווים בין חזאים? לוקחים ותכנית, מרכיבים אותה בסימולטור של המעבד עם כמה חזאים שונים ורואים מי מהם נותן את החיזויים הכי טובים. באופן כללי מיקרו-ארכ' זה תחום של ניסוי וטעה אבל בגלל שמרחיב הפתרונאות הוא אינסופי כדי שתיהיא אינטואיציה טוב איזה ניסויים לעשות.

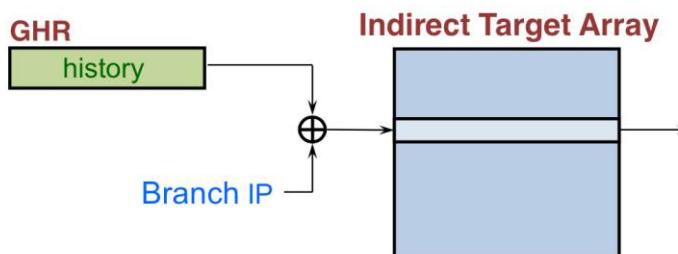
השלב הבא אחרי השיפורים שראינו לקחת שני חזאים שונים עבור כל קפיצה ולבחר בחיזוי של אחד מהם: למשל לחזות חלק מהבראנס'ים עפ"י חזאי לוקלי וחלק מהם עפ"י חזאי גלובלי. נעשה זאת ע"י מערכ של מונימ שמאונדקס עפ"י ההיסטוריה ונעדכן את המונימ במערך כך: כל פעם שהחזוי המקורי צדק והגלובלי טעה נעלם באחד, כאשר המקורי צדק והגלובלי טעו לא נעשה שינוי צדקו/טעו לא נעשה שינוי במונה. כך תוצאה המונה תהיה לנו באיזה בראנץ' עדיף לבחור בפעם הבאה שנגיעה לבראנץ'.

במעבד מודרני מחזיקים בזמנים הרבה חזאים, שפועלים עם היסטוריות אורך שונים, ותמיד מעדיפים לחזות עם החזאי בעל ההיסטוריה הקצרה ביותר (הכי פחות הספק) ואם לא מצליחים עוברים לחזות דרך ההיסטוריה יותר ארוכה וכך אפשר לחזות בראנצ'ים עם תבניות פשוטות ע"י חזאים זולים יותר ובראנצ'ים מורכבים משתמשים בחזאים הכבדים יותר.

עד כאן על ה-BTB - החלק ב-BTB ששמש לחיזוי קפיצות מותננות, שהוא כאמור הרכיב החשוב ביותר ב-BTB כי 85% מהקפיצות הן כאלה (conditional direct jumps) וכן שם הטעות על החזוי לא נכון היה הכל גדול לה.

## Indirect branch predictor

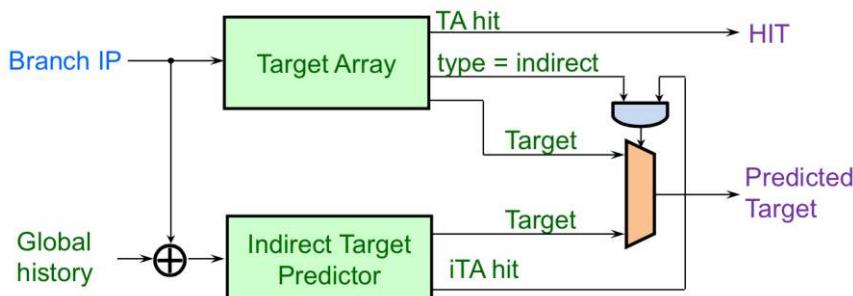
- ◆ **Indirect branch targets: target given in a register**
  - ❖ Can have many targets: e.g., a case statement
  - ❖ Resolved at execution  $\Rightarrow$  high misprediction penalty
  - ❖ Used in object-oriented code (C++, Java)
- ◆ **A history-based indirect branch target predictor**
  - ❖ Uses the same global history used for conditional jumps



Indirect Jumps הן קפיצות שkopatzot כל פעם לכתחבת אחרת - בקפיצות אלה השאלה הקשה היא לאן נkopoz. התשובה ידועה רק בסוף שלב Execute כי שם מחושב ערך הקפיצה (עפ"י ערך שנמצא ברגיסטר/זיכרון). לשם טיפול בקפיצות אלה נבנה חזאי שיחזה לאן יkopoz Jump Indirect  $\Rightarrow$  היסטוריה של הקפיצות הקודמות. נשים לב שהוא לא רצוי לשומר היסטוריה של כתובות, כלומר במקומות בהם של נלקח/לא-נקח לשומר רצפי כתובות שונות, כי כתובות הן רוחבות מאוד ומערך שייאונדקס ע"י רצפים כל כך ארוכים ייקח המון מקום. מה שאנו עושים הוא להשתמש בההיסטוריה הגלובלית שהשתמשנו בה בעבר ה-BTB global conditional predictor כדי לחזות את יעד הג'אמפים שם indirect - זה עובד כי מסתבר שהמקום אליו ה-Indirect Jump קופץ תלוי בצורה חזקה במסלול החישוב שהביא את התכנית אל הקפיצה, והמסלול הזה הוא בדיקת ההיסטוריה שזכרנו ואתה ניגשנו לחזאי הגלובלי. אז ע"י אותה ההיסטוריה פונים למערך שיכיל כתובות קפיצה - זה חזאי לא רע כדי לחזות unconditional jumps.

דוג' ל קומפקט Indirect Jump בתכנית: אם יש לנו switch case או כל פעם קופצים למקום אחר כתלות בערך עליו עושים switch. הערך זהה הוא תולדה של מה שקרה בתכנית עד ה switch, ומה שקרה בתכנית עד ה switch הוא פונ' של הבחירה NT/T שעשינו בבראנץ'ים שראינו עד כה.

- ◆ **Initially allocate indirect branch only in the Target Array (TA)**
  - ❖ Many indirect branches have only one target
- ◆ **If the TA mispredicts for an indirect jump**
  - ❖ Allocate an iTA entry with the current target
  - ❖ Indexed by IP  $\oplus$  Global History
- ◆ **Prediction from the iTA is used if**
  - ❖ TA indicates an indirect jump and iTA hits



את ה-Indirect Jump מזמנים בהתחלה ב-target array עם סימן (type) של direct jump או indirect jump. משמשים בו כדי לחזות את הקפיצה, כאשר היא תמיד יכולה לקפוץ רק לשם. כל עוד ה-target array צודק ואין בעיה אז הוא ימישר לחזות מידית ולא נקזה מקומות במערך חיזוי-הכתובות של ה-target array אבל ברגע שהה-target array יטעה כי קפצנו למקום אחר אז נשנה את ה-Type השמור ב-target array ונקזה ב-target array indirect array. גם זו דוג' שמראה את מרגע זה כל פעם שהtarget array נזהה את הכתובת הרצוייה מה-indirect array. גם זו דוג' שמראה את העיקנון לפיו תמיד מנסים לחזות בדרך פשוטה יותר, בדוג' זו ה-target array בלבד, וכל עוד היא עובדת ממשך להשתמש בה. רק ברגע שנטענו נתחיל להקצות מקומות בחזאים יותר מורכבים ורך אז בזמן הבדיקה הבאה נשתמש בהם.

**Return Stack Buffer**

# Return Stack Buffer

- ◆ **A return instruction is a special case of an indirect branch**
  - ❖ Each times it jumps to a different target
  - ❖ The target is determined by the location of the corresponding Call instruction
- ◆ **Maintain a small stack of targets at fetch time**
  - ❖ When the Target Array predicts a Call
    - Push the address of the instruction which follows the Call into the stack (this is the predicted Return address)
  - ❖ When the Target Array predicts a Return
    - Pop a target from the stack and use it as the Return address

הרכיב האחרון שנלמד עליו הוא ה- Return Stack Buffer שתפקידו לחזות פקודות מסווג return. באופן כללי בפקודות call המעבד דוחף למחסנית (בדומה לפעולת push) את הכתובת שאחרי פקודה ה-call, למשל אם בכתובת 1000 נמצאת הפקודה 2000 call אז המעבד דוחף למחסנית את הכתובת 1004 לפני הקפיצה לשגרה שנמצאת ב-2000 (כתובת החזרה היא זו שאחרי פקודה ה call), ולאחר ביצוע קוד הפונ' מגיעים לפקודת return שניגשת למחסנית, מוציאה ממנה את כתובות החזרה (בדומה לפעולת pop) ומשתמשת בהריך שקיבלה בתורו כתובות הקפיצה. הבעיה בכל הסיפור זהה היא שהמחסנית נמצאת בזיכרון ולכן ה-push וה-pop קורים בשלב מאוחר מאוד ב-Pipe, ועד שפקודת return מקבלת את הערך שלה עובר הרבה זמן ונחnano רוצים להכניס פקודת הביצוע בזמן זהה. לשם כך מנהלים בתחום המעבד עוד ממחסנית שככל פעם שיש call דוחפים אליו ערך וכל פעם שעורשים return מוצאים ממנה את הערך. העומק של הממחסנית הוא לא מאד عمוק כי בדר"כ אנחנו לא נמצאים בקינון של יותר מדי פונ' בזמן ריצה - לרוב גם ממחסנית שיש בה שלוש כתובות תהיה מספיק טוביה. לעומת זאת בתכנית בעלת קינון מאד גדול של שגרות נתחיל להזם את הממחסנית של כתובות החזרה ואז שנצטרכ לבעץ return לא נמצא בא יותר את ה return של ה call הרלוונטי לנו, נספק חיזוי שגורו ובסופה של דבר נסבול מביצוע יותר איטי. מתי העומק גדול במיוחד? ברקורסיה - שם אי אפשר לעשות חיזוי של כתובות החזרה וזה סיבה לכך שմבחן המעבד רקורסיה היא דבר לא טוב. אבל במקרה של תוכנית לא-רקורסיבית גם ממחסנית קטנה היא בעלת אחווי חיזוי-נכון טובים מאוד וכן משתמשים ב- Return Stack Buffer שהוא בפועל ממחסנית קטנה.

## הרצאה מס' 7: זיכרון וירטואלי

## Virtual Memory

היום נתחיל נושא חדש - זיכרון וירטואלי.

## Virtual Memory

- **Provide isolation: each process sees its own memory space**

- Many processes can run on a single machine
- Prevents a process from accessing the memory of other processes

- **Provides the illusion of a large memory for each process**

- Different machines have different amount of physical memory
  - Allows programs to run regardless of actual physical memory size
  - Sum of memory spaces of all process may be larger than physical memory

- **Provides illusion of contiguous memory**

- The amount of memory consumed by each process is dynamic
- Allows adding memory and keep it contiguous

המטרות של זיכרון וירטואלי:

• בידוד (isolation): כל תהליך צריך להיות מסוגל לגשת רק לזכרון שלו אלא אם ביקשו במפורש שיתוף (sharing) בין תהליכיים. בפרט בידוד בין כתובות הזיכרון של תהליכיים שונים שרצים על אותו מעבד. בכך מדובר גם על תהליכיים שונים של אותו משתמש, גם תהליכיים של משתמשים שונים וגם הפרדה בין תחילciי-משתמש לתהליכיים של מ"ה. למשל במחשב פרטי כל התהליכיים הם של אותו משתמש אבל עדין לא נרצה שתתהליך אחד יכול לגשת לזכרון של אחר, למשל לא הייתה רוצה שמשחק יוכל לגשת לקובץ Word וכו'. כמובן שבמיצבים כדוגמת שרת בהן מס' משתמשים רצים על אותו מעבד כל וחומר שלא נרצה לאפשר לתתהליך של משתמש אחד לגשת לזכרון של תהליך של משתמש אחר.

• דבר שני שרצים הוא שלכל תהליך שרצ על המעבד תהיה אשליה שיש לו מרחב-זיכרון גדול מאוד. אם בנושא הקודם, caching, רצינו>Create an alias for the memory area of each process that is large enough to hold all the data required by the process. בפרט התתהליך צריך לחזב שיש לו כמות זיכרון שלא תליה בגודל של הזיכרון הפיזי שモתקן במחשב גדול, ובפרט התתהליך צריך לחזב שיש לו כמות זיכרון שלא תליה בגודל של הזיכרון הפיזי שモתקן במחשב עליו הוא רצ - אם לקוח אחד קונה 8GB אחר 16GB אז כשתתהליך של אפליקציה כלשהי רצ הוא צריך לרוץ באופן זהה בלי תלות בכמה זיכרון פיזי יש במחשב. כמו כן יכול להיות שרצים במחשב עשרות ומאות תהליכיים של מ"ה, תכניות פתוחות וכו', וכל תתהליך רוצה להציג שיש לו זיכרון מספיק גדול - אם נעשה סכום של כל המרחבים שהם חושבים שיש להם בקளות נבעור את הזיכרון הפיזי שיש במחשב.

• כמו כן נרצה שכל תהליך יאמין שיש לו זיכרון רציף (contiguous memory) למרות שתהליכיים שונים יכולים לעבוד במקומות סמוכים בזיכרון הפיזי זו"א כל פעם שתתהליך יעשה malloc רוצים שמבחרינתו הוא יקבל תחום רציף של כתובות גם אם בפועל מ"ה נתנה לו לרוץ על מרחב לא רציף בזכרון הפיזי.

הweeney הבסיסי בזכרון וירטואלי (ז"ו): מרחב הכתובות שאנו מציינים בתכנית יקרא המרחב הווירטואלי (מ"ו). כל הכתובות שאנו/קומפיאר כתובים בתכנית הן וירטואליות - תתהליך אף פעם לא מייצר פניה ישירה לכתובת בזכרון הפיזי - לא בשביל פקודות ולא בשביל נתונים. אז המ"ו זה מרחב הכתובות שנמצא מתכוית, הוא זה

שהתהליך רואה, ומרחבי הכתובות הפיזי הוא הזיכרון שsspציפית קיים במחשב שלנו, למשל 4GB או 8GB. בסופו של דבר כדי שתוכנית תעבור למרחב הווירטואלי על גבי הזיכרון הפיזי צריך להגדיר תרגום שיאפשר להביא data מהזיכרון הפיזי, כלומר צריך לעתות תרגום לכל תחילה מהמרחב הווירטואלי למרחב הפיזי, למשל כתובת 1000 בתהיליך אחד וכתובת 1000 בתהיליך אחר בפועל יתורגם לכתובות פיזיות שונות - זו המשמעות של הבידוד (isolation) שדיברנו עליו. אז כל תחילה צריך למפות את המ"ז שלו לאיזושהי כתובות פיזיות.

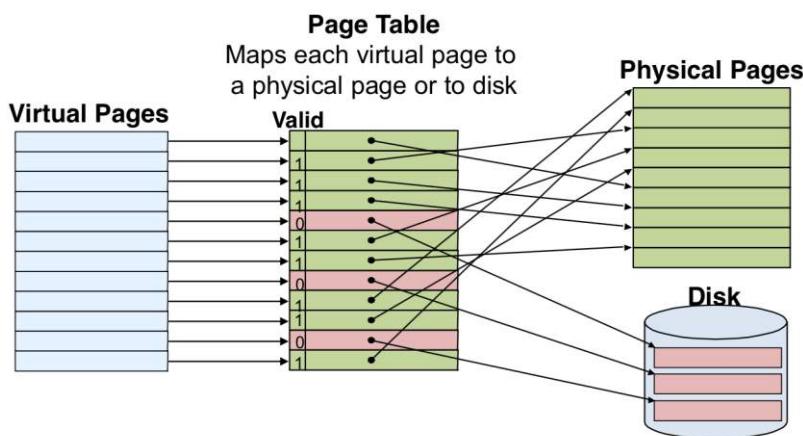
## טבלת דפים - Page Table

לשם הגדרת מיפוי למרחב וירטואלי של תחילה לזכרון פיזי, כמו שב-caching חילקו את הזיכרון לבlokים, פה מחלקים את המ"ז לדפים (Virtual Pages) ואת המרחב הפיזי מחלקים למסגרות (Frames) או Physical Pages - בפועל נקרא לשניהם pages ונבין מהקשר לאיזה מרחב הם שייכים. בעוד שבקаш היה מדובר בבלוקים קטנים יחסית, למשל 64B, גודל אופייני של דף (ווירטואלי ופיזי) הוא 4KB.

בנוסף נתונים לכל תחילה לחשו שיש לו מרחב וירטואלי ענק - למשל מרחב שנפרש ע"י  $40 \times 1TB = 2^{40}B$  ואומר מחלקים אותו לדפים של 4KB. במקרה של 1TB יהיו במ"ז  $2^{40}/2^{12} = 2^{28}$  דפים, כלומר M (כברע-מיליארד) דפים. באופן דומה נחלק גם את המרחב הפיזי לדפים באותו גודל - 4KB, למשל אם קנוינו 8GB אז מדובר ב  $2^{33}$  דפים. נשים לב שהמרחב הפיזי קטן יותר מהמרחב הווירטואלי, שמשרת את המטרה של לתת לתהיליך לחשוב שיש לו מרחב זיכרון גדול.

המייפוי בין כתובות פיזיות לכתובות וירטואלית נעשה ע"י טבלאות דפים:

## Page Tables



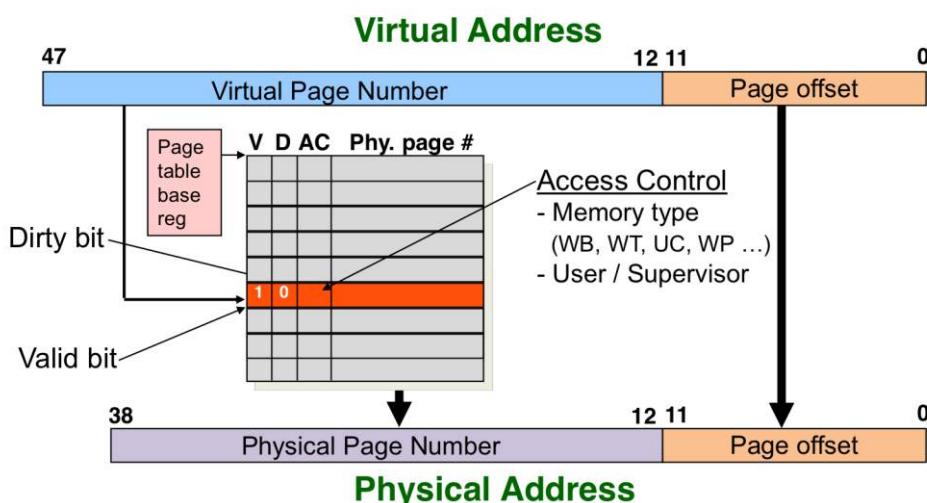
- **The OS manages a page table for each process**
  - The OS maps the process virtual pages to physical pages or to disk
  - Only the OS writes into the page tables
  - The page tables reside in the physical memory (DRAM)

כל תחילה יש טבלת דפים משלו וזה מה שבעצם מבידיל את המרחב של תחילה אחד מהמרחב של אחר. טבלת הדפים היא גדולה מאוד - יש בה כניסה לכל אחד מהדפים הווירטואליים במ"ז של התהיליך. אם למשל יש  $2^{28}$  דפים וירטואליים אז יהיו בטבלה  $2^{28}$  כניסות. והיא מבצעת את המיפוי מכתובת וירטואלית לפיזית כך: בהינתן מס' דף וירטואלי פונים (VPN - Virtual Page Number) לכינסה המתאימה לו בטבלת הדפים וממנה מקבלים את המיפוי לכתובת הפיזית שמהווה את תחילת הדף הפיזי שכרגע מומפה לאותו דף וירטואלי שרצויים לתרגם. לכל תהיליך יש טבלת דפים משלו וזה בדיקת הדרך של מ"ה להבטיח בידוד - מ"ה לא תקצה לשני תהליכיים שונים את

אותו מיפוי לדף פיזי (אלא אם ביקשו חלוקה במפורש - נראה דוג' לכך בהמשך) אלא שבתיח שהמיפויים זרים למיפויים פיזיים שתהיליך אחר קיבול. טבלת הדפים נמצאת בזכרון הפיזי ובהמשך נראה כיצד בדיק פונימי אליה לקבלת התרגום.

מי אחראי על מיilo טבלת הדפים זו מ"ה, ולמעשה מאוד חשוב לשים לב בכך כל הנושא של זיכרון וירטואלי איפה עבר הגבול - מה עורשה החומרה ומה עורשה מ"ה. כלל פשוט אומר: מ"ה היא זו שכותבת לטבלת הדפים והחומרה היא זו שקוראת ממנו, ככלומר מ"ה קובעת את המיפויים והחומרה רק קוראת אותם - היא לא מייצרת אותם. כיוון שהזיכרון הפיזי הרבה יותר קטן ממה"ו של תהיליך כלשהו, וכמוון שהזיכרון הפיזי קטן מסכום המ"ו של כל התהיליכים במערכת, אז לא יתכן שייהי מיפויי "חח" כלומר לא יתכן שייהי מקום בזכרון הפיזי לכל המרחבים הוירטואליים של כל התהיליכים (וכנראה אפילו לא של אחד כאמור). אז ברור שאין מיפויי שיאפשר לכל הדפים הוירטואליים של כל התהיליכים להימצא בזכרון הפיזי בו זמןית - שכן מ"ה משתמש במנגנון של דפודף (Paging) כלומר העברת דפים מהזיכרון הפיזי להارد דיסק: מ"ה, בהתאם לשיקולים שלה, מחליטה איזה דף פיזי היא מפה לדיסק הקשייה ואיזה דפים וירטואליים לזכרון הפיזי ואיז רק הדפים הוירטואליים בשימוש כרגע יהיו ממופים ע"י מ"ה לזכרון הראשי.

מבנה הכתובות במרחב הוירטואלי/פיזי:



PTE – Page Table Entry

Page size:  $2^{12}$  byte = 4K byte

בדוג' כתובת המרחב הוירטואלי היא בעלת 48 סיביות - כלומר כל כתובת בתכנית שהקומפיילר מייצר היא בעלת 48 סיביות. זה בפועל הגדול המקסימלי שנתרם כרגע ב-Intel: כשהגע לדבר על מימוש נראה שהמקרו-ארכ' בפועל תומכת רק עד 48 סיביות - עם הזמן מגדים את זה יותר ויתר לקרה 64 וכרגע זה על 48. נשים לב שבניגוד לכתובת הוירטואלית הכתובת הפיזית תלויות לא רק במעבד אלא חייבת להיות מגובה ע"י כמות הזיכרון פיזי יש לנו במחשב - ז"א אם אני במחשב ביתי 8GB או 16GB זה יקבע את גודל המרחב הפיזי - אם מחר נקנה יותר זיכרון אז גודל המרחב הפיזי יגדל.

בשביל לבצע מיפוי מכתובות וירטואלית של תהיליך לכתובת פיזית עושים שהוא יוכל להזיכר קצר ארגון של cache - מחלקים את הכתובת של התהיליך נתן לשני חלקים: החלק העליון הוא מס' הדף הוירטואלי והתחתון הוא ההיסט (offset) בטור הדף. כשאנחנו עובדים עם דפים בגודל 4KB אז הריסט הוא בגודל 12 ביטים, כדי שייהי אפשר לגשת לכל אחד מבין  $2^{12}$  הבטים בדף. כל הביטים בכתובת הוירטואלית החל מביט מס' 12 מהווים את מס' הדף הוירטואלי - ה-Virtual Page Number (VPN).

בעת תרגום לוקחים את מס' הדף הוירטואלי, החלק העליון של הכתובת כאמור, ומשתמשים בו כאינדקס בטבלת הדפים - כלומר פונמים לכינוסה המתאימה למס' הדף

בטבלה ושם מוצאים את התרגום של הדף הווירטואלי לדף הפיזי בזיכרון יחד עם עוד כמה ביטים - בין היתר בכל כניסה בטבלת הדפים יש בית שנקרא **Valid** שאומר האם הדף המתאים לכניסה זו בכלל מוגדר לזכרון הפיזי או לא - אם הביט דלוק אז הכניסה מכילה את מס' הדף הפיזי, אם הביט כבוי אז היא מכילה זבל.

שאלה: מי מעדכן את bit **Valid**? מ"ה - כאמור היא היחידה שכותבת לטבלת הדפים! כשם"ה החלטה למפות את הדף היא הדלקה את bit **Valid** כדי שהחומרה תדע שהיא יכולה להשתמש בכניסה זו עבור התרגום של הדף הווירטואלי המתאים לה.

הערה: יש בית אחד יוצא דופן מבחן זה שהוא החומרה מדילקה - **Dirty bit**, שאומר האם בוצעה כתיבה לדף מסוים שהוא הובא מהדיסק אל הזיכרון הפיזי - ז"א **dirty bit** אומר האם מהרגע שם"ה החלטה להביא את הדף מההارد דיסק לזכרון והפיזי ועד לבדיקה הביט העותק שבהارد דיסק נותר זהה לעותק שבזיכרון הפיזי - לשם כך ברגע שבוצעה כתיבה לדף בזכרון הפיזי החומרה מדילקה את **dirty** bit ואז ניתן לדעת שבוצעה כתיבה אל איששו בית בדף. צריך לדעת את זה כי כשם"ה מוציאה דף מהזיכרון לצורך הכנסת דף אחר היא צריכה לדעת האם יש צורך לעדכן אותו בדיסק. נחזור על זה: כאשר מ"ה תבחר את הדף שייהה קורבן להוצאה אז היא תקרה את **dirty**: אם הביט כבוי אז הקרבון של הדף הוא קל - פשוט זורקים אותו מהזיכרון הפיזי כי יודעים שמילא העותק שבדיסק זהה אז לא צריך להעתיק אותו חזרה מהזיכרון הפיזי אל הדיסק, ואז זול לפנות אותו. אם הביט דלוק צריך לנקחת את הדף ולהעתיק אותו אל הדיסק.

**Access control bits** הם ביטים שקיימים בכל כניסה בטבלת הדפים, גם הם מעודכנים ע"י מ"ה, והם מגדרים מהן זכויות הגישה לדף - ופה בעצם אנחנו מוסיפים יתרון נוסף - סיבת ריבועית לזכרון וירטואלי: מלבד בידוד בין תהליכיים, זיכרון גדול וזכרון רציף מ"ה יכולה להשתמש במנגנון הזה כדי להגדיר ברמות גישה לכל דף, למשל שהדף הוא לקריאה בלבד, ואז מ"ה לא מרשה לתוכר לכתוב אליו אלא רק לקרוא ממנו, או למשל מדילקה דגלים של **WR** שאומר שאפשר גם וגם, או שמדילקה בית שנקרא **non-executable** כלומר שמה"ה לא מרשה לעשות מדף זה **Fetch** של פקודות - הביט הזה נוסך בשנים האחרונות והוא נדרש כדי להגן מפני וירוסים: אחת ההתקפות הבסיסיות של וירוסים היא לשנות איזור של **data** קר שיכיל קוד של וירוס ואז להתחילה לבצע מאזור זה את הפקודות **- non-executable** - יכול למנוע מהחומרה לאפשר זאת.

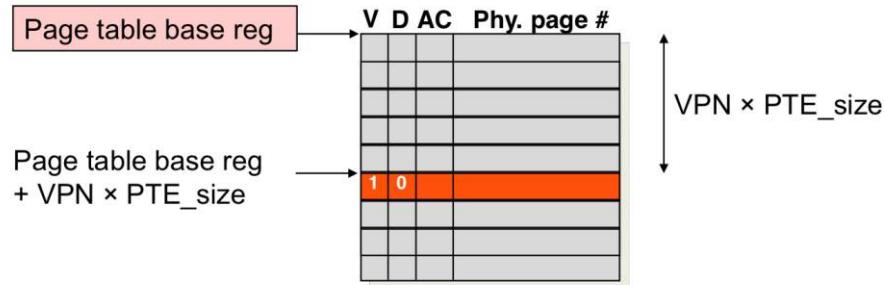
הבהרה: אינדקסים עוקבים בטבלת הדפים משרים מ"ו רציף - אבל הטבלה לא ממפה באופן רציף. אם נעשה malloc לשני דפים אז הכתובות הווירטואליות של האיזור שנקלבל יראו כרציפות, לעומת זאת המיפויים הם לא ידועים ואף אקרים - בשנים האחרונות מ"ה הוסיף פון' שמנסה לערבות את המיפויים האלה כי זה מקשה על וירוסים.

בשדה העיקרי והגדול ביותר בכניסה של טבלת הדפים כתוב לאיזה דף פיזי הדף הווירטואלי שמתאים אותה כניסה מוגדר: כאמור השדה זהה תקף רק אם bit **Valid**. Dolok, כלומר רק אז יש לשדה הזה המשמעות של מס' הדף הפיזי אליו המפנה הדף הווירטואלי. נשים לב כי מס' הדף הפיזי, כלומר כמות הביטים בשדה זה, יכול להיות קצר יותר מה-**VPN**, כי מס' הדפים הפיזיים יכול להיות קטן ממש הדפים הווירטואליים.

از נניח שהחומרה פנתה לטבלת הדפים, עברה את בקרת הגישה ווידה שהדף **Valid** - עכשו היא מקבלת את מס' הדף הפיזי אליו הדף הווירטואלי מוגדר. בשלב זה מ"ה לוקחת את ה-**offset** שבסכמתה המקורית ומשרשת אותו מלמטה למס' הדף הפיזי (ה-**offset** בדף הווירטואלי זהה ל-**offset** בדף הפיזי כי שניהם באותו גודל - 4KB) וכך קיבלו כתובות פיזיות שעבורה פונים לזכרון.

- **The page table of each process resides in main memory**

- When a process is running, the start address of its page table is pointed by a special register in the CPU: the *page table base register*
- The *page table base register* holds the physical address of entry 0



- **The physical address of the PTE of virtual page #VPN**

- PTE address = Page table base reg + VPN × PTE\_size

טבלת הדפים יושבת בזיכרון הראשי (ולכן בטוח שלא כל הזיכרון הראשי מוקצה לשימוש ע"י תכניות). אמוננו שהקל מהמ"ז של תחיליך ממופה לזכרון הפיזי (כל דף שעבורו Valid bit Dolik), אבל מ"ה צריכה לחלק את הזיכרון הפיזי בין כל התהליכים, כולל לשמור מקום בשבייל טבלת הדפים שלהם וכן לשומר חלק ממנו לעצמה. בכל ארכ' ישנו רגיסטר-ארQUITקטוני (כלומר רגיסטר שמוגדר כחלק מה-ISA של הארכ' - קומפיילרים יכולים לפנות אליו ולהשתמש בו) שנקרא (באיינט) CR3 (Page Table Base Register) ו-PTBR (Page Table Base Register). נלמד עליו כשנדבר על ז"ו בארכ' 68x (והוא תמיד מכיל את הכתובת הפיזית של תחילית טבלת הדפים של תחיליך הנוכחי שרץ במעבד. כל פעם שמעבד מחליף תחיליך ועובד להריץ תחיליך אחר, כלומר עשויה החלפת הקשר (Context Switch) מחליפים את הערך של ה-PTBR וגורמים לו להציג על טבלת הדפים של תחיליך החדש. חשוב להבין (ולזכור) שטוענים אלו ערך חדש כל פעם שיש החלפת הקשר!

הבהרה: מי שמשתמש בטבלת הדפים היא החומרה: כל פעם שקוד-מכונה של תחיליך מבקש כתובות, בין אם של פקודה או של נתונים, הכתובת היא ורטואלית - ולפניהם שהחומרה יכולה לגשת לזכרון בכתובת הפיזית המתאימה היא צריכה לגשת לזכרון בשבייל טבלת הדפים, לקבל את המיפוי, ולאחר מכן באמת הכתובת הפיזית ורק אז אפשר לגשת למידע עצמו בזיכרון - אז כל גישה לזכרון (למשל load) עליה שתי גישות לזכרון: אחת בשבייל להביא את התרגום ואחת בשבייל המידע הנוכחי. לפני שונילחץ מזה נזכיר שהגישה השנייה לזכרון, זו שתפקידה להביא את הדאטא בכל פעם, לא אמורה להציג כללי כי בשבייל זה יש קאשיים, למשל L1 ו-L2. עוד מעט נראה שגם תרגומים של טבלת הדפים יש במעבד קאש והוא נקרא TLB.

## Page Fault

# Address Mapping Algorithm

**If  $V = 1$  then**

page is in main memory at frame address stored in table  
 ⇒ Fetch data

**else (page fault)**

need to fetch page from disk  
 ⇒ causes a trap, usually accompanied by a *context switch*:  
 current process suspended while page is fetched from disk

**Access Control (R = Read-only, R/W = read/write, X = execute only)**

If kind of access not compatible with specified access rights then  
**protection violation fault**  
 ⇒ causes trap to hardware, or software fault handler

• Missing item fetched from secondary memory only on the occurrence of a fault ⇒ *demand load policy*

- **Page fault**

- Data is not in memory ⇒ retrieve it from disk
- The CPU **detects** the situation, but cannot **remedy** the situation  
 ⇒ the CPU traps to the OS to remedy the situation
- **The OS**
  - If there is no free space in physical memory
    - ▲ Picks a physical page to discard (based on some replacement policy)
    - ▲ Marks the page as not present in the page table
    - ▲ Copies the page to the disk swap area
  - Loads the new page from disk into the selected physical page
  - Updates the page table entry of the new page
  - Resume to the program so HW retries and succeeds

- **A page fault usually causes a context switch**

- Current process suspended while page is fetched from disk

כאשר החומרה מבקשת לתרגם כתובות מווירט' לפיזית היא ניגשת לטבלת הדפים ובודם כל קוראת את ה- **valid bit**. אם הוא 1 אז החומרה קוראת את התרגום מטבלת הדפים, אם הוא 0 זה אומר שהדף המתאים לא ממופה כרגע ל זיכרון הפיזי - מצב של גישה לדף שאינו ממופה נקרא **page fault**. קודם כל חשוב להבין שהחומרה לא יודעת מה לעשות במקרה זה: הדף נמצא בהארד-ディסק ולמעבד אין מושג מה זה ההארד-ディסק ולא כל שכן איך לפנות לנוטונים שם. לכן כשייש **page fault** המעבד מיד מפסיק את ריצת התהיליך הנוכחי (interrupt) וקורא לשגרה של מ"ה - קריית מערכת שתפקידה לטפל ב **page fault**. איז התהיליך הרגיל מפסיק לרגע, מתќבלת חריגה ומתחילה לרגע שתהריך שתהריך והיא בתורה קוראת לשגרה של מ"ה שמטפלת ב-**page faults**. מ"ה היא זו שתצטרכן ללקת להביא את הדף החסר מההארד-ディסק, לעדכן את המיפוי שלו בטבלת הדפים ואם נגמר לה המקום בזיכרון הפיזי, או שהיא חושבת שנשאר מעט מדי, איז היא צריכה במקומם הדף החדש לזרוק דף אחר מהזיכרון הפיזי לדיסק ועוד - בשורה התחתונה מ"ה תdag לכך שתהיליך שפנה לדף שגורם ל-**page fault** יהיה מיפוי בזיכרון הפיזי ואז נחזיר אל הפקודה שקיבלה **fault**, נבצע אותה שוב ועכשו כבר לא יהיה **page fault** והחומרה תמצא את ה **valid bit** דולק עם המיפוי החדש שהחומרה שמה שם.

אפשר לקבל **fc** לא רק במקרה של **valid bit** כבוי אלא גם במקרה שהתהיליך ביצע פקודה לא חוקית לדף - כלומר במקרה של **Access Violation**. למשל התהיליך ניסה לבצע כתיבה ל זיכרון (**store**) לדף ש"מ"ה הגדרה אותו כ **Read Only**. מ"ה תצטרכן להבין מה לעשות עם התהיליך זה - בדר"כ הורגמים אותו, אבל זה כאמור החלטה של מ"ה, אפשר גם לשלוח לו סיגナル כך שיישנה את פקודת הגישה וعود. הערה: לעיתים מ"ה מביאה מראש דפים מהדיסק ל זיכרון הפיזי, זה נקרא **demand-paging**, אבל זה נוגע לקורס במ"ה ולא חלק של הקורס שלנו.

עוד הערה: מצב של **page fault** במוסאי המעבד זה נצח - אם אמרנו שגיישה ל זיכרון הפיזי אורכת מאות מחזוריים אז גישה להארד-ディסק אורכת עשרות-אלפי ואף מאות-אלפי מחזוריים, לכן כשמקבלים **page fault** מ"ה בדרך כלל תייצר context switch כך שתהיליך אחר יוכל לרטוק בזמן שברקע נdag את הדף מההארד-ディסק ל זיכרון הפיזי.

## Optimal Page Size

- **Minimize wasted storage**
  - Small page minimizes internal fragmentation
  - Small page increase size of page table
- **Minimize transfer time – use large pages (multiple disk sectors)**
  - Amortize access cost and prefetch useful data
  - But, might transfer unnecessary info and discard useful data early
- **General trend toward larger pages because**
  - Big cheap RAM
  - Increasing memory / disk performance gap
  - Larger address spaces

דיון זה קצר דומה לדיוון על הגודל האופטימלי לבLOCK בקאס: מצד אחד אם יש דפים קטנים אז הם מקטינים את הפרגמננטציה (fragmentation, שבירור) בזיכרון: פרגמננטציה אומרת שלא כל הדף בשימוש ואז יש בזבוז של זיכרון, מקום שהוא יהיה מנווצל ביעילות. דוג': אם תחליך רוחה 2KB של זיכרון מסוים והחומרה עובדת בדפים של 2MB, אז למ"ה אין ברירה אלא להקצתות לתחליך 2MB בזיכרון הפיזי למרות שהוא ציריך רק 2KB - זה בזבוז. במובן זה עדיף דפים קטנים כדי שלא תהיה פרגמננטציה. בנוסף לפרגמננטציה התחליך ציריך רק KB2 אבל עדין צריך להביא את כל ה-MB2-ים בזמן לטיפול בpage fault היה אורך כי להביא MB2 מהdisk לוקח יותר זמן. מצד שני כשהדפים קטנים או הז"ו מתחולק ליותר דפים וואז טבלת הדפים הופך מגודלה לענקית: אמרנו שהטבלה בזיכרון הפיזי, ואם הדפים קטנים או מספרם גדול והוא ביכולות יכולת לחזור, מבחינת הזיכרון הדורש כדי לאחסן אותה, מהזיכרון הפיזי במחשב - נפתר את זה בהמשך אבל עדין מבחינה זו עדיף לנו דפים כמה שיותר גדולים.

הערה לגבי הקצת זיכרון דינמית: ברגע שעשינו `malloc` מ"ה תקצה לתחליך דף בזיכרון הווירטואלי - ברגע זה עדין נעשתה הקצתה הדף/דפים בזיכרון הפיזי. ברגע שתחליך יבצע פניה ראשונה לכתיבתה/קריאה מדף שהוקצתה רק ציריך ממש דף בזיכרון הפיזי, لكن תיווצר `page fault`, מ"ה תLER אל ההארד-ディסק ותעתיק ממנו מידע אל דף פיזי שתקצתה בזיכרון. גם אלה יש אופטימיזציה: בדרך"כ כשם"ה מקצתה דף בפעם הראשונה היא מניחה שהדף כולל אפסים. כל עוד התחליך רק יקרה או יכתוב רק אפסים אז לא יוקצתה דף פיזי עבור הדף אלא הערכיהם יקראו מדף אפסים ששמור במיוחד לצורך זה - רק כאשר מבוצע כתיבה לראשונה של ערך שאינו 0 אז מ"ה תקצתה את הדף הזה בזיכרון הפיזי - זה נקרא (Copy On Write) COW.

אז כאמור בבחירה הגודל של הדף השיקולים بعد דפים קטנים הם פרגמננטציה ו-transfer time (אם צריך להעביר דף גדול מהזיכרון הפיזי לדיסק או להיפול או הטיפול ב-fk הוא יותר גדול). שיקול بعد דפים גדולים הוא שהאחסון של טבלת-התרגום יהיה יותר קומפקטי. באופן כללי הטרנד הוא שהגדלים של הדפים גדל עם הזמן - אם לפני 50 שנה לאנשים היו הארד דיסקים קטנים וDRAM קטנים וקושים קטנים ועסקו בDATA קטן אז היום כל זה הולך ונוהיה יותר גדול - יותר תכניות עובדות על מסדי נתונים עוקיים וגם זיכרונות פיזיים ודיםקים גדולים ומהירים, אז זה הגיוני שבאופן הדרגי הדפים שעובדים אתכם נהנים יותר ויותר גדולים.

בדרך"כ החומרה תומכת בכמה גדלים מסוימים של דפים, למשל x86 תומכת ב-4KB, 2MB, 1GB, ... ומ"ה היא זו שבוחרת האם להקצות דפים בגרנולריות צזו או אחרת. כשאיןטלו רוחה לשנות משחו בתחום זהה היא חייבת לסכם

עם יצרניות מ"ה כמו Google (Android), Apple (MacOS), Microsoft (Windows) ועוד. נראה בהמשך איך בדיק 64x תומכת בשלושת הגדלים הנ"ל.

## TLB (Translation Lookaside Buffer)

### Translation Look aside Buffer (TLB)

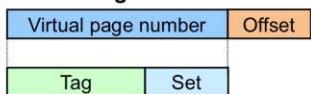
- Page table resides in memory**

⇒ כל תרגום נדרש גישה נוספת לメモリ.

- The TLB caches recently used PTEs**

- בדרך כלל 128 עד 256 אינטראקציות, 4 עד 8 אונטיאטיביות.

- TLB Indexing**



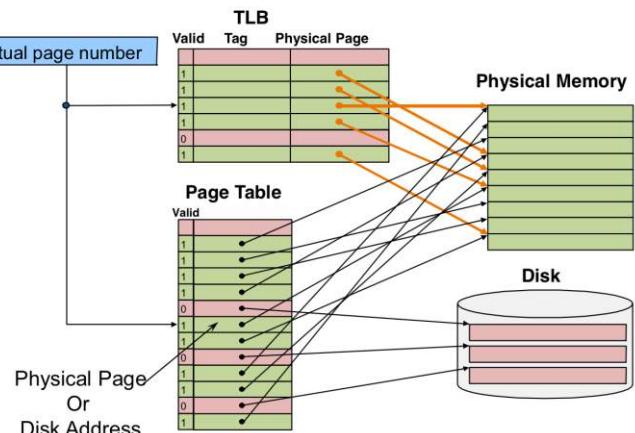
- On A TLB miss**

- הידול בפער TLB (HW PMH) מקבל PTE מהזיכרון.

- OS is responsible to maintain coherency between page table and TLB**

- כל פעם שcribes לpte בזיכרון, TLB must be invalidated (אם קיים).

The TLB is a cache for recent address translations:



אמרנו שעכשו כל גישה לזכרון לכאורה דורשת שתי גישות לזכרון: אחת לתרגום ואחת למידע.

הינו רוצים לגשת לחסוך את הגישה לזכרון בשבייל התרגומים, לכן מגדרים במעבד cache עבור טבלאות הדפים: כמו שהוא L1,L2 עבור נתוני/פקודות יש גם כאשר הם מוקדשים אך ורק להקל תרגומים מטבלאות דפים והם נקראים Look aside Buffer (TLB) (Translation Lookaside Buffer). בכל מעבד בימינו יש TLB וכל פעם שעושים החלפת-הקשר מחליפים אותו - ככל התרגומים שהוא מכיל הם תמיד של התהילה הנוכחי שרצ.

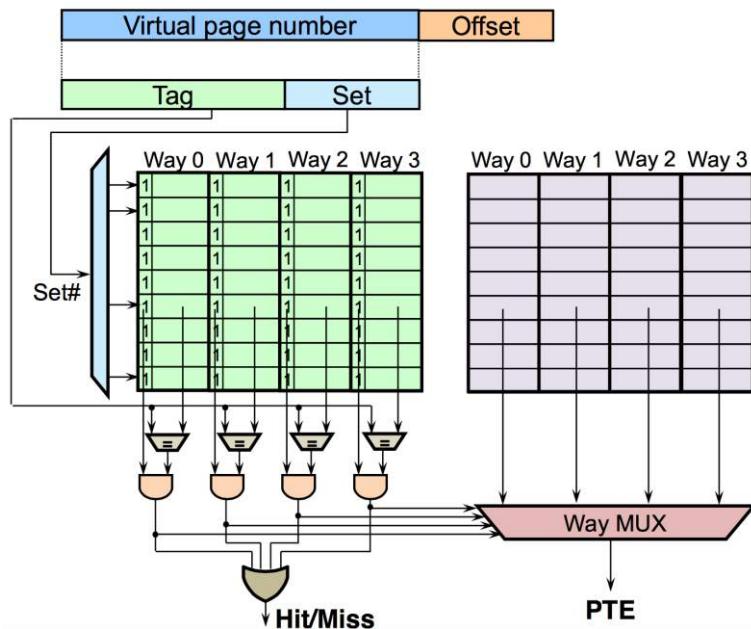
כאמור ה-TLB הוא קאש של טבלת הדפים וכל פעם שיש גישה לכתובת (למשל ע"י היבאת פקודה לביצוע או, load/store) התהילה הנוכחי, לפני שניגש הראשי כדי לבקש את התרגום, קודם כל ניגש ל-TLB ושאל האם יש לנו את התרגום של הדף הוירטואלי זהה לדף פיזי - אם כן זה נקרא hit ו-TLB נותן מיד את התרגום לכתובת פיזית. אם לעומת זאת TLB אין את התרגום אז זה נקרא Miss TLB ואז באמת צריך ללקת להיררכיות הזיכרון (אשימים, זיכרון ראשי וכו') ולקבל שם את התרגום - בהתאם לאזמנות שמיים את התרגום גם ב-TLB בשבייל הגישות הבאות לו דף (עקרון הלוקליות בזמן). נשים לב שככל תרגום שמופיע ב-TLB תמיד תואם לתרגום בטבלת הדפים (נרחיב על כך מיד) וכן בהוצאה ממנו אפשר פשוט לדרום את הכניסה שרצוים להוציא.

כמו בכל תרגום ממרחב וירטואלי למרחב פיזי גם ה-TLB לא מתרגם את הכתובת הוירטואלית כולה אלא רק את מס' הדף הוירטואלי - שאר הכתובת מושלמת בעזרת offset בתוך הדף שזהה גם עבור דפים פיזיים וגם עבור וירטואליים. לכן ה-TLB מקבל את מס' הדף הוירטואלי (הכתובת הוירטואלית ללא היחס בתוך הדף) ומחלק אותו ל-set ו-tag: הביטים הנמוכים יבחרו את-set של VPN ב-TLB והשאר את-h-tags. לפי זה נעשה lookup שקובע האם יש hit/miss TLB - אם יש hit ניגשים לכתובת הפיזית המתאימה בזיכרון ואחרת ניגשים ל-TLB, רכיב חומרה שתפקידו ללקת לטבלה בזיכרון הראשי, להביא את התרגום המתאים PMH - Page Miss Handler ולשים אותו ב-TLB. כיוון שה-TLB הוא קאש עבור טבלת הדפים של התהילה שרצ כרגע אז כל פעם שיש החלפת-הקשר צריך לעשות flush כלומר אחרי החלפת הקשר הוא תמיד ריק. בנוסף כל פעם שמשהה משתה בטבלת הדפים היא צריכה ללקת ולוחז אב-טבלת הדפים ב-TLB במעבד לא יהיה עותק לא-מעודכן של התרגום. זה הבדל מעניין ממה שלמדנו עד כה: אשימים נכתבים ע"י חומרה בלבד, למשל אם מישו כתוב לכתובת כלשהי במתומו הפרטיה שלו אז

פרוטוקול MESI דאג לשומר על קוהרנטיות ושלא יהיה ערכים שונים אותה כתובת בזכרון המטען של מעבדים שונים וכו' - לעומת זאת מ"ה היא זאת שאחראית לשמירה על קוהרנטיות בין ה-TLB לבין טבלת הדפים (של התהילך הנוכחי), בפרט מ"ה יכולה לבטל ב-TLB את המיפוי של דף כלשהו או לעדכן ב-TLB את ה-Access Rights של דף - למשל מ W/R/W ל Read-Only. זה חשוב כי אם לא יבוצע סנכרון במפורש אז כשהתהילך ייגש ל-UBR עבור הדף הספציפי בו בוצע השינוי הוא לא ידע בכלל שבטבלת הדפים נעשה שינוי ע"י מ"ה. לכן כל פעם שמ"ה עrsa השינוי בטבלת הדפים היא מרים פקודה `invalidate page` שמקבלת את מס' הדף הוירטואלי ועושה `invalidate` ב-TLB של המעבד. נשים לב שם"ה לא טרחת לעדכן את התרגום ב-TLB ובאופן כללי מ"ה לא מכינהו או מודעת למה שנמצא ב-TLB אלא רק מבקשת למחוק את התרגום היישן אם הוא שם. אז לפני שמ"ה מבצעת שינוי בטבלת הדפים לגבי ד"ו מסוים היא מבצעת את ה `invalidate page` שעוז פקודה ארכ' (חלק משפט המעבד, ה-ISA) שתגרום לכך שהמעבד יעשה `snoop` ואחריה הדף כבר לא נמצא ב-TLB. לכן אם מקבלים תרגום מה-TLB אז לא יכול להיות שיש עליו `page fault` כלומר תרגום שנמצא ב-TLB תמיד זהה לתרגום שבטבלת הדפים בזכרון הפיזי.

נפרט איך ניתן לסת ול-way הנכונים:

## TLB Access

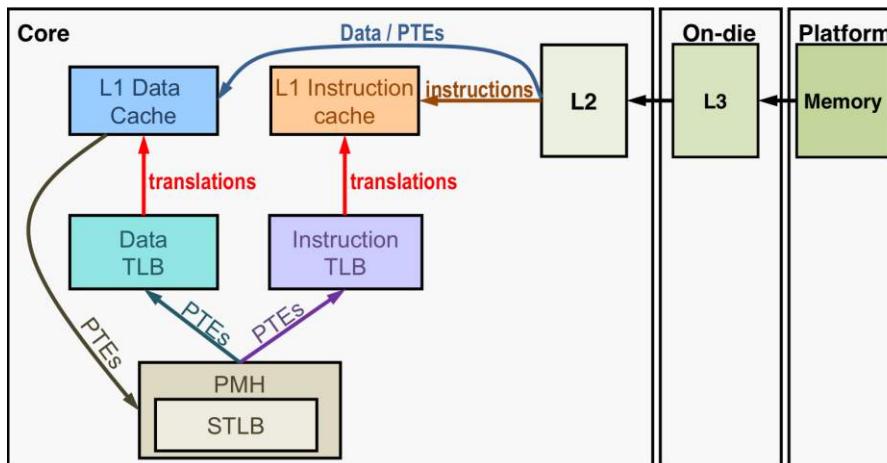


כאמור בהינתן מס' דף וירטואלי מחלקים אותו לשדה **set**, הולכים לסת המתאים ב-TLB, משווים את ה-**tags** בכל ה **ways** באותו סט ואם אחד מהם נותן **match** אז יש בעצם TLB **hit** ובעצם מחזירים את תוכן ה **page** (page table entry) המתאים לדף וכן בעצם חסכנו את הגישה ל זיכרון עבור תרגול הדף. הבהרה: **PTE** (page table entry) היא כניסה כלשהי בטבלת הדפים.

בشكل הבא רואים את ה-TLB והקאים שיש במעבד:

## Processor Caches

- L2 and L3 are unified, as the memory – hold data and instructions
- In case of STLB miss, PMH accesses the data cache for page walk



נתחל מהיררכיית הזיכרון המוכרת לנו:

|                             |                               |
|-----------------------------|-------------------------------|
| L1 (Data & L1 Instructions) |                               |
| L2 (Unified)                | כל זה עדין בתוך ליבת אחת //   |
| L3                          | משרת כמה קוררים //            |
| Memory                      | משרת מעבדים (צ'יפים) שונים // |

הבהרה חשובה: זיכרונות קаш עובדים עם כתובות פיזיות ולא וירטואליות (נרחיב על כל בשבוע הבא). בדומה ל-L1 יש שני TLB, אחד לכתובות של נתונים ואחד לפקודות, כאשר כל אחד מהם מכיל חלק מהKENISOT בטלת הדפים של התהlixir הנוכחי. שימוש לדוגמא: כשהעושים fetch ניגשים L-Instruction TLB עם הכתובת ב-Instruction Pointer ובנהנה שיש hit TLB ניגשים לקאש-הפקודות עם התרגום הפיזי שלה כדי להביא את הכתובת הבאה. אותו דבר לנתחנים: ברגע שיש load/store מחשבים את הכתובת הווירטואלית אליה ניגשים ופונים עמה אל ה-TLB, מקבלים את הכתובת הפיזית המתאימה ואתה הולכים ל-Data Cache.

כמו שיש היררכיה של L3 אז יש גם Second-level TLB אוhitTLB אם יש TLB Miss: אם יש לפני השולדים ל-PMH, שניגש לטבלת הדפים בזיכרון, יש לשמשת גם את ה-TLB instr. בדומה לארגון/שיטוף/מס' הקאים גם ארגון של TLB זה שיקול מיקרו-ארכ' של ביצועים, בפרט לא לכל מעבד חייב להיות S-TLB.

כמה העורות ודגשים לגבי ה-TLB:

- כאמור כל הKENISOT ב-TLB בשקל נמחקوت בעת החלפת הקשר.
- ה-Second level TLB הוא יותר גדול מה-TLB - זה הגיוני בדומה למה שנעשה בקאים.
- לא בהכרח מתקיים עקרון ההכללה בין S-TLB וה-TLB.

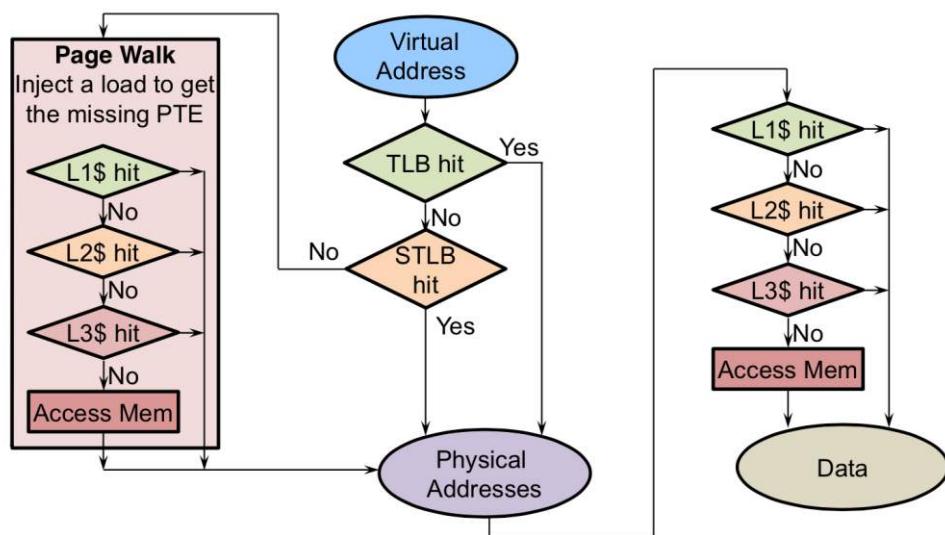
נסכם: כאשר יש load הכתובת ממנה טענים היא וירטואלית, לכן נגשים קודם כל ל-TLB data, שואלים האם יש לו את התרגום, אם יש hit אז הוא נותן את התרגום ואפשר לגשת ל Data Cache ואם אין אז נגשים ל-STLB.

ושואלים האם שם יש את התרגום, בהנחה שיש מבאים אותו ל-TLB וואחריו זה נגשים ל-Data Cache, אם אין נגשים ל-PMH ש מביא את התרגום ל-S-TLB או ל-B-TLB (או לשנייהם).

אמרנו שכשיש TLB Miss ה PMH צריך ללקת להביא את הכתובת של הדף מהזיכרון, לשם כך ה PMH מזיריק בעצמו load לאזרם ההוראות של המעבד - זה נקרא load stuffed והוא ה load היחיד שנעשה לכתובת פיזית - מלבד זאת ה load הזה רגיל לחלווטין - קודם כל הולך ל-1 וואז לשאר..

נביט בשקף על המסלול בו מבאים מהזיכרון את המידע שמתאים לכתובת וירטואלית כלשהי:

## Virtual Memory And Cache



- ❖ TLB access is serial with cache access
- ❖ Page table entries are cached in L1 D\$, L2\$ and L3\$ as data

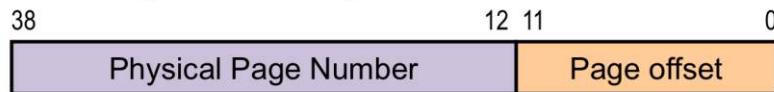
агף שמאל מתאר את הפניה להבאת התרגום של הכתובת הווירטואלית ואגר' ימין מתאר את הפניה להבאת ה-Data (אחרי קבלת התרגום): ברגע שמקבלים כתובת וירטואלית כלשהו בשלב הראשון הוא להשיג את התרגום שלה לכתובת פיזית: ניגשים ל-TLB לבקש תרגום, אם יש hit פונים לקאש עם הכתובת הפיזית שהוא נותן לנו ומקבלים את ה-data. לחופין אם קיבל TLB Miss ננסה ב-2nd-level TLB ואז אפשר ללקת שוב להביא את ה-data. אם קיבל S-TLB Miss אז ה-PMH נדרש load כדי להביא את התרגום מהקאשיהם ואם לא בקאשים צריך להביא את החלק הזה של טבלת הדפים מהזיכרון. נשים לב שגם כל PTE היא 8 בתים ושורת הקאש היא של B64 אז בכל שורה יהיו שמונה כניסה, לכן יחד עם ה-PTE הרלוונטי נביא לקאש גם עוד שבע PTEs ואז בסבירות גבוהה כשייה miss לדף הבא שנוצר לתרגום מעקרון המיקומיות במרחב הוא יתאים לאחת הכניסות שהובאו וה-PTE הרצוי תהיה בקאש. נחזיר על זה, יותר ברורו: כל כניסה של PTE היא בגודל 8 בתים, לכן כשambilאים PTE אחת מהזיכרון לקאש בפועל מבאים עוד שבע, ואז יהיו שמונה PTEs סמוכות בקאש וסביר מאד שנמצא את התרגומים הבאים שם. לעומת זאת לא מבאים את כל השמונה ל-TLB.

הכתובת (הפיזית) של ה-PTE הרצוי, אתה ניגש לזכרון המתמוך ול זיכרון, במידה ונתקבל בהם החטאות, שווה ל- $PTBR + VPN * PTE\_SIZE$ .

נשים לב שגם אם נניח שהגישות ל-TLB ול-L1 אורך מוחזר שעון אחד אום עד היום חשבנו שהכי מהר שאפשר לקבל נתון כלשהו זה במחזור יחיד הנה מבנים שזה לוקח לפחות לפחות שנים. בעת נverbן לדבר על דרך בה אפשר לחופף חלק מהגישות ל-TLB ולקأش.

## גישה במקביל ל Cache ו-TLB

### Virtual Memory view of a Physical Address



### Cache view of a Physical Address



#### In the above example #Set is contained within the Page Offset

- ⇒ The #Set is known immediately
- ⇒ Cache can be accessed in parallel with address translation
- ⇒ Once translation is done, match upper bits with tags

#### Limitation: Cache $\leq$ (page size $\times$ associativity)

- מבחינת ז'ו - המבנה של כתובת פיזית מורכב ממש' דף פיזי ו-offset בטור הדף.זיכרון ה-*page* נשאר זהה, רק ממש' הדף הוירטואלי מתרגם לממש' דף פיזי.
- מבחינת הקאש - כתובת (פיזית) מחולקת ל-offset בטור הבלוק, ל-tag ולמש' הסט בקאש.

כלומר ה-TLB והקאש מחלקים את הכתובות באופן שונה.

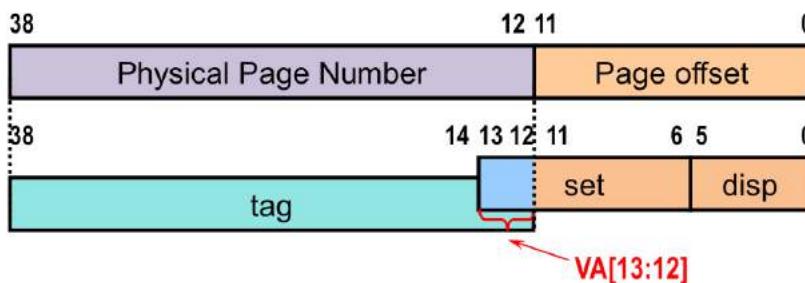
נשים לב שאם הביטים של page offset + block offset + הביטים של הסט בקאש לא יעברו את הביטים של ה-*tag* בכתובת הוירטואלית אז הגישה לסט המתאים בקאש לא תלויות בתרגום שיתקבל מה-TLB - ככלומר אף אחד מהביטים בכתובת הפיזית שמהווים את שדות ה-set offset לא ישתו במהלך התרגום לכתובת פיזית ולכן אפשר לדעת לאיזה סט לגשת ולטוען את ה-tags להשוואה במקביל להבאת ה-*tag* של הכתובת הפיזית ע"ז תהליך התרגום. אז אם ההיסט וממש' הסט מוכלים בטור הביטים של ה-*page offset*, שלא מתרגם, בעצם אפשר להשתמש בהם בלי לחכות ל-TLB. נזכיר על זה: אם נארגן את הקאש כך שגודלו הבלוק וממש' הסטים קטנים מספיק כדי להיות מוכלים בביטים של ה-*page offset* בטור הדף, אז במקביל לגישה ל-TLB אפשר גם לגשת לקאש, לקרווא את כל התגים מהסט המתאים בקאש וברגע שmagiu התרגום אל הדף הפיזי tag אפשר מיד לבצע את ההשוואה מול ה-*tag* בכל סט ולהוכיח רק לתוצאת ה-*tag match*.

היתרון הזה מכתיב את הגודל/אסוציאטיביות של הקאש (כי גודל הבלוק וממש' הסטים קבועים את גודלו). הגודל של הקאש במעבדי Intel הוא 32KB כבר הרבה זמן מהסבירה שנרצה במקביל לגשת גם ל-TLB וגם לקאש. אם נרצה להגדיל את הקאש ליותר מ 32 ק"ב עדין נרצה את הגישה במקביל וזה אומר שצריך להגדיל האסוציאטיביות - זה לא כל כך טוב מבחינת ביצועים והספק אבל זה לא הכרה המציגות ואולי אפשר לפתור את זה - בכל מקרה זה שיקול.

از אמרנו שאנו רוצים שהקash יהיה אחד גדול, מצד שני שלא יהיה בו יותר מדי ways ומצד שלישי שניהה מסוגלים לגשת ל-set הנכון בכל way במקביל ל-TLB כדי שלא נחכה הרבה מוחזרים בשבייל ביצוע תרגום של כתובות וירטואלית לכתובת פיזית. למשל אם נוכל לגשת במקביל לקash ול-TLB ובשניהם נקבל hit (זה המצביע) אז את שניהם נעשה באותו מוחזר ובמהדור הבא נבצע את החיפוש של ה-tag שהתקבל ע"י התרגום עם הרווח (זה שומר באותו סט בכל אחד מהways בקash, כך שבמוקם שלושה מוחזרים לכל גישה (תרגומים, גישה לקash, השוואת תגים) דרישים שני מוחזרים וההבדל בביטויים בין גישה של שלושה מוחזרים לשניים זה ממשמעותי מאוד - סדר גודל של 10% שיופיעו בתכניות אמיתיות.

ה"טריק" הבא שנלמד נמצא בשימוש ע"י AMD ולעתים קרובות מופיע בבחינות:

- **Assume a cache is 32K Byte, 2 way set-associative, 64 byte/line**
  - $(2^{15}/2 \text{ ways}) / (2^6 \text{ bytes/line}) = 2^{15-1-6} = 2^8 = 256 \text{ sets}$
  - Cache indexing: offset: PA[5:0], set: PA[13:6], tag: PA[38:14]
- **In order to still allow overlap between set access and TLB access**
  - Instead of using PA[13:6] for the set, use VA[13:6] ([11:6] from page offset)



- **Cache offset+set+tag must cover the entire Physical Address**
  - Since bits PA[13:12] are not part of the set, they are covered in the tag
  - The tag is comprised of PA[38:12]

איך אפשר לצור חפיפה בין הגישה ל-TLB לבין הגישה לקash למוראות שהביטים של היסט בבלוק + מס' ה-set חורגים מהביטים של ה-offset בדף הזיכרון?

רוצים שייהי קASH יותר גדול בפרט לאפשר לביטים של הסט לחרוג מביט-11 (זכור ברוב הארכ' היום גודל דף 4KB, لكن ה-page offset הוא בדר"כ בביטים 11-0), ובכל זאת תתאפשר גישה במקביל, ככלומר שלא נצטרך לחכום לתרגם של מס' הדף מה-TLB כדי לקבל מהתרגום את שאר הביטים של הסט ואז נוכל לגשת אותם. למשל בדוגמה מס' הדף החרוג מה-page offset בשני ביטים, כלומר ביטים 12 ו-13 הם הביטים הבלתיים ולא נוכל לגשתם לסט המתאים בקash לפני שהתרגם של הכתובת הפיזית מה-TLB יגיע. הטריק הוא שבסמוך להשתמש בביטויים 12,13 הפיזיים בגישה לקash (כלומר בבחירה הסט אליו ניתן בקash) נשתחם בביטויים 12,13 הווירטואליים כך שהקash הופך ל"שעטנע" זהה: חלק מהביטים שקובעים את הסט בקash הם פיזיים, אלה הביטים שהיו בתוך האזורי של ה-tag, page offset, וחלק מהם וירטואליים. נשים לב שגם זה המצביע איז איז אפשר פשוט להשוות את ה-tag המתתקבל ע"י ה-TLB ולדעת אם יש hit או לא - יכול להיות שהtag יהיה זהה אבל עדין יהיו ביטים במס' הדף הפיזי (הBITSים התחתונים) שלא השתמשו בהם בתהיליך ה-cache lookup כלל, כלומר כל כתובות שנבדלת רק בביטויים אלה יכולה להיות במקום זה בקash וצריך להבדיל ביניהן. נחזור על זה: הבעירה היא שינוי הביטים 12,13 הפיזיים, עדין לא נלקחו בחשבון ביחס לTAG - השתחם בביטויים 12,13 הווירטואליים, וכך בביטויים 12,13 בכתובת הפיזית יצטרכו להופיע איפשהו בתהיליך ה-cache lookup. איפה? ב-tag! נצטרך להשתמש בביטויים 12,13 כחלק מה-tag!

כדי לזהות את הכתובת, כזכור ל-tag חייבים לחכות כי הוא מהוות חלק מהכתובת פיזית, ורק אחרי שקיבלי אותו, בפרט את ביטים 12,13, משווים את התג ייחד אותם ורק אז אני בטוחה ידוע שמה שיש במקום מסוים בזיכרון הוא אכן הכתובת המבוקשת ולא כתובת קצרה אחרת, במובן זה שני הביטים האלה מתאימים למה שאנו חפשים.

הבהרה: ביטים 12,13 גולשים מעבר ל-offset של הדף ואני אדע אותם רק אחרי שה-TLB יחזיר את התרגומים, אבל אני רוצה לגשת לסת הנכון לפני כן.. לכן אני משתמש בביטים הוירטואליים כשאני מכניס כתובת לקаш וכן כשאני ממחפש כתובת בקاش.

נמיהש את זה בדוגמה: נניח וקיבלי כתובת וירטואלית מסוימת לתרגום. בהמשך קיבל תרגום של מס' הדף שלו ונניח שביטים 12,13 בכתובת הפיזית יתבררו שונים מהוירטואליים. אמרנו שכאשר אני עושה cache lookup אז אני מכניס לקаш אני הולך לסת שנקבע לפי הביטים הוירטואליים. בפעם הבאה שאלך שוב לאוותה כתובת אפנה עם אותם ביטים וכן אגע לאותו מקום - אבל אני עדיין צריך לוודא שמדובר באותה כתובת פיזית שאני ממחפש. דוג'': נניח שני דפים וירטואליים והמיופיעים שלהם הם:

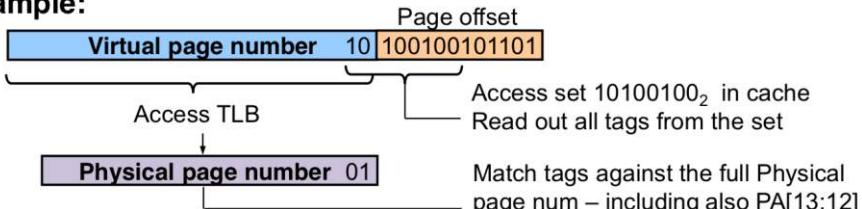
**111101110 10 -> 1010111 00 (000000000000)**

**00100000 10 -> 1010111 01 (000000000000)**

נשים לב שהכתובות של שני הדפים הפיזיים נבדלות רק בביטים 12,13. נניח שהחנסת נתון אל אחד משני הדפים הוירטואליים הנ"ל אני פונה לקаш בסט 10 ושם שם בлок שיופיע לדף הפיזי המתאים. אח"כ אני עושה cache lookup לנตอน מהדף השני - נשים לב שה-tag (לא הביטים 12 ו-13 הפיזיים) הוא אותו tag וכן שני הביטים הוירטואליים, שאני משתמש בהם עבור הסת, הם אותם ביטים - אך הגישה תיתן לי את אותו נตอน בדיק ששמנו מהדף הראשון - אלא שהוא לא נתון מאותו דף. איך נdag שזה לא יהיה המצביע? איך נבדיל ביניהם? התג חייב לכלול גם את שתי הסיבות החסרות שלא נלקחו בחשבון, ואז אין בעיה - התג לא יהיה אותו tag, לא נקבל tag match וננדע שהנתון שיש בקاش הוא לא הנตอน שchipשנו.

אם משתמשים בשיטה זו העניין יכול להסתבר בשני אופנים:

- **Example:**



- **OS may map two virtual pages to the same physical page**

- Each one of these virtual pages may have different VA[13:12] bits
  - The same physical address is mapped into 2 different sets in the cache
  - Data in one copy may be different than the other
- Solution: allow only one virtual alias in the cache at any given time
  - On a cache miss, before writing the missed entry to the cache, search for virtual aliases already in the cache and evict them first
  - No special work is necessary during a cache hit

- **An external Snoop supplies only the physical address**

- With virtual indexing, all the possible sets must be snooped

1. מ"ה יכולה למפות מס' דפים וירטואליים שונים לאותו דף פיזי ואם נשתמש בבייטים הווירטואליים אז עלולים להיות מס' עותקים של אותו נתן פיזי במקומות שונים בקاش - בכל אחד משני הדפים הווירטואליים ביטים 12,13 יהיו שונים ואז אותו דף פיזי יופיע בכמה סטים שונים בקاش - זה לא טוב כי יכול להיות שאינה כתיבה לכתובת בדף כלשהו דרך כתובת וירטואלית אחת ואז אקרה את הנגנון מכתובת וירטואלית אחרת (ולכן מסט אחר בקash) שאמורה להתמפהות לאותו דף אך שבפועל נקרה ערך ישן - כלומר יהיו כמה עותקים של אותו דף פיזי בקash. לכן שימושים בבייטים הווירטואליים לצורך בחירת הסט בכל פעם שיש cache Miss מוחפשם בכל הסטים המתאפשרים ע"י שינוי של אותם בייטים וירטואליים האם במקרה המיפוי של אותה כתובת הפיזית נעשה גם באמצעות בייטים וירטואליים אחרים. זה מזכיר מאד את תהליך הטיפול ב cache Miss - כל פעם נדרש למצוא את הכתובת בכל המיקומות (למשל אם משתמש בשני בייטים וירטואליים נדרש לחפש ארבעה סטים ואם משתמשים בחמשה בייטים וירטואליים נדרש לחפש ב-32 סטים, בכל סט בכל אחד מה-ח ways) ואם מצאנו את הכתובת אפשרו צורך כדי לוודא שלא תהיה במקומות שונים ואז לבצע את ההכנסה.
  2. ראיינו שלעתים מבצעים `invalidate` כדי לשמור על הכללה, למשל אם זורקים משם מ `L2 cache` נדרש לוודא שהוא לא מופיע ב-`L1` - אבל כשמבצעים `invalidate` עושים זאת עבור כתובת פיזית, כלומר כשעושים זאת `L2` בכלל לא מודע לכתובת הווירטואלית שבמעבד, לכן `L2` נדרש לחפש בכל הסטים שמתאפשרים ע"י כל האפשר עבור הבייטים הווירטואליים שימושים בהם בקash.
- از כאמור שני הדברים הרעים בשיטה זו הם שהייבים לוודא שהכתובת לא מופיעה בכמה מקומות שונים בקash בכלל מיפויים של כמה דפים וירטואליים לאותו דף פיזי וشب-קoodoks חיצוני נדרש לחפש בכל הסטים האפשריים כתוצאה **משינוי** היביטים הווירטואליים.

הרצתה מס' 8: זיכרון וירטואלי – המשך.

## Virtual Memory

בברצאה שעברה התחלנו ללמידה על זיכרון וירטואלי - נעשה חזרה קצרה: ז"ו מאפשר לכל תהליך לחשב שיש לו מרחב זיכרון גדול, רציף ופרטוי - ככלומר שאף תהליכי אחר לא יכול לגשת אליו. עשינו את זה ע"י חלוקת מרחב הכתובות הוירטואלי של תהליכי לדפים, וכן"ל חילקנו את מרחב הזיכרון הפיזי (מרחב הזיכרון הראשי, DRAM) למסגרות שגם להן קראנו דפים. בהינתן החלוקה העקרונית זו יצרנו באמצעות טבלת הדפים מיפוי בין המרחב הוירטואלי של כל תהליך למרחב הפיזי.

המיופיע עבד כר: בהינתן כתובת וירטואלית (אליה שעמן עובדים בתכנית) הביטים העליונים של הכתובת יהיו את מס' הדף הווירטואלי (VPN - Virtual Page Number) ומשתמשים בהם כאינדקס בטבלת הדפים.

אם היה `hit` בטבלת הדפים, כלומר אם ה-`bit valid`, שקיים בכל כניסה, דлок, אז בכניסה המתאימה מופיע מס' הדף הפיזי אליו הד"ו מופיע - הדפים מיושרים בזיכרונו בкопיות של דף (כלומר כתובות החתולות של הדף מהוות את מס' הדף משורשר לרצף של אפסים). לאחר קבלת מס' הדף הווירטואלי משרשרים אליו את שאר הביטים התאימים בכתובת, שנקבעים היסט - `offset`, בדרכו 12 ביטים (כלומר בדרכו 12 עמודים עם דפים בגודל 4KB). נראה היום כמה אפשרויות אחרות לגודל הדף ואז שדה היסט אינו בגודל 12 ביט אלא גודל מסוים כדי להתייחס לכל הכתובות בדף, למשל עבור דף של GB 1 נצטרר היסט בגודל 30 ביט.

המצביע בו ניגשים לכינסה בביטול הדפים בה בית-*Valid* נקרא *page fault* - זה מצב שמצווה ע"י החומרה אך מטופל ע"י מ"ה: بغدادו מה ש"מ"ה צריכה לעשות במצב זה הוא לדאוג לכך שהמיופי של הכתובת הווירטואלית תריה תקפה, כלומר לדאוג שהדף הווירטואלי בו היא נמצאת תומפה לדף פיזי כלשהו בזיכרון הראשי. אם נגמר מקום בזיכרון הפיזי, יכולם כבר מוקצה לדפים אחרים (של התהיליך/מ"ה/תהליכיים אחרים) אז היא צריכה להמוציא דף קורבן, לכתוב אותו חזרה לדיסק במידה ובכינסה שלו דлок דגל *Dirty*, שאומר שתוכן הדף שונה מזאת שהובא מהדיסק, להביא את הדף המתאים לגישה שלו מהדיסק ולמפות את הדף הווירטואלי למקום של הדף החדש בזיכרון הפיזי. א"כ הפקודה שגרמה ל-*page-fault* תורץ שוב והפעם תצליח כי קבענו מיופי מתאים עבור רכਮות הווירטואליות שרשימים רפקודה זו.

הזכרנו שבראכ' מגדר רגיסטר ייעודי שתמיד מציבע לתחילה טבלת הדפים, שנמצאת בזיכרון הפיזי, ולכן הגישה הראשונה, לצורך הבאת התרגומים, היא ישירות לזכרון הפיזי ולא דורשת תרגום כלשהו - פשוט לוקחים את הרגיסטר שציבע לתחילת ט"ד ומוסיפים לו את מכפלת מס' הדף הווריאוֹלִי בגודל כניסה בטבלה וכן מקבלים את התרגומים (למשל אם כל כניסה ביא בגודל 8 בתים וורוצאים את דפ' 1000 מוסיפים 8000 לרגיסטר המצביע לטבלה).

דיברנו על כך שלכארה כשבודים עם ז"ו איז כל גישה לזיכרון עצמו דורשת לפחות שתי גישות: גישה אחת לזיכרון לטבלת הדפים כדי לקבל את התרגום של הדף הוירטואלי לדף פיזי ורק אח"כ גישה ל-data, למשל דרך ה-L1 Data Cache. אמרנו ששתי גישות לזכרון זה יקר מדי ולכן כמו שיש cache יש גם עבור תרגומי TLB והוא מתרגם ממ' דף וירטואלי למ' דף פיזי במהירות: כמו cache לכל דבר יכולה להיות לו דפים שנקראו TLB ומס' הדרישה מthem. וכך יופיע הדרישה לTLB.TLB מחלק אותו ל-set ו-tag אסוציאטיבי, למשל set associative 2-way, ולכן בפועלתו הוא לוקח את מס' הדף, מחלק אותו ל-set ו-tag ומסתכל האם הדף ב-set בו הוא מצפה למצאו אותו (בקיים), והוא עובד כמו cache לכל דבר. ראיינו שיש הפרדה בין TLB שמכיל תרגומים של נתונים (data) ו-TLB שמכיל תרגומים של פקודות (instructions). אז למשל כל פעם שרוצים לקבל מידע כלשהו קודם כל פונים ל-TLB data/instr.TLB ו רק אז יש לנו את התרגום לכתובת פיזי שאתה פונים ל-data/instr cache-L1.

במידה ויש לנו חטאה באחד ה-TLBs אז יש לנו ברוב המעבדים בימינו גם 2nd level TLB, שבדרכו משתמש גם עבורו נתונים וגם עבור פקודות, במידה והוא מוצא את התרגום הוא מעביר אותו ואם גם ב-2nd level TLB יש החטאה אז אין ברירה אלא באמצעות לטבלת הדפים בזיכרון - מי שערשה את זה הוא ה-PMS (Page Miss Handler) שידוע לגשת לטבלת הדפים בזיכרון הפיזי ולשים את התרגום שמצא שם ב-2nd level TLB וב-2nd level TLB Handler.

הנתוניות/פקודות. לצורך זה ה-PME מזրיך **load**, זה נקרא **stuffed load**, לכתובות הפיזיות בה נמצאת ה-PTE, אותה הוא חישב על סמך מס' הדף וערכו של הרגיסטר שמצויב לטבלת הדפים כפי שהסביר ומקבל את תוכן הכניסה המתאימה בטבלת הדפים (זהו המקרה היחיד של **load** לכתובות פיזית, כל שאר ה-**loads** הם תמיד לכתובות וירטואלית). כמו כל גישה לזכורן ה-**stuffed load** קודם כל פונה ל-**L1 cache** כי יכול להיות שתוכן הכניסה בטבלת הדפים נמצא שם וכך הוא הולך ל-**L2**, **L3** ועוד גישה לזכורן הפיזי עצמו.

אמרנו שהמסלול הכי מהיר של גישה לכתובות כלשהו קורה כאשר יש **hit**-ב-TLB וגם **hit** בעור הכתובות המתורגם ב-**L1 cache** ולש machtanu זה גם המקרה הנפוץ. אבל אנחנו גם יכולים לקבל **TLB Miss**-ב-**TLB** ואז להצטרך לעשות מסלולים שונים וארכוכים עד שנגיע לתרגום של כתובות ה-**data** הווירטואלית שהתייחסנו אליה בתכנית - מסלולים אלה, להבאת התרגומים, גם עוברים דרך זיכרונות המטען כמו כל גישה אחרת כolumn אחריו **TLB Miss** ו-**STLB** **Miss** פונים בשביל מיציאת ה-PTE ל-**L1, L2** וכן הלאה עד לזכורן. אז לעיתים התרגום ייקח יותר זמן ולפעמים פחות.

הנושא האחרון שדנו בו הוא כיצד לבצע גישה ל-TLB כדי לקבל את התרגום הווירטואלית לפיזית ובמקביל להתחיל לבצע את הגישה לבקש להבאת הנתון מכוחבו המתורגם (הפיזית): במקרה שהבאים בכתובות שמצוינים מס' סט בקשר לאchorים מהbiteים של שדה הhispt איז עצם אפשר לדעת מה יהיה הסט של הנתון בקשר גם לפני שעשינו את התרגום (כי הביטים של **offset**, ובקרה זה של הסט, לא מתורגים) ואז אפשר במקביל לגשת ל-TLB ולטוען את ה-**tags** של הסטים המתאים בקשר (סט אחד בכל **way**) ואחרי טיענת ה-**tags** נשאר רק לחכות ל-TLB בשביל שנוכל להשוות את הכתובות הפיזיות של הנתון עם ה-**tag** בכל סט ולדעתי אם יש לנו רק יוצרים חיפפה בין התרגומים ובין הגישה לקשר. בהקשר זה ראיינו דוג' (חוובה למחה!) שمرة איך אפשר לבצע פניה ל-TLB ולקשר במקביל גם כשלדה הסט בכתובות חורג מההיסטוריה של הדף הווירטואלי.

## פינוי דף מהזיכרון

# Evicting a Page to Disk

- **OS updates the Page Table**

- OS marks the swapped-out page as *not present* (valid=0) in its PTE
- OS writes the page disk location into the PTE
- OS invalidate the page from the TLBs
  - It is OS responsibility to maintain coherency between page table and TLBs for any change in the page table

- **OS code copies the page to the disk**

- Read each block within the page
  - Snoop-invalidate each byte in the CPU caches (L1, L2, ...), and if hits
    - ▲ If it is modified read its line from the cache to get updated data
    - ▲ Invalidate the line
- Write the block to the disk controller Memory-Mapped I/O area
- This means that when a page is swapped-out of memory
  - All data in the caches which belongs to that page is invalidated
  - The page in the disk is up-to-date

כשיש page fault החומרה מבקשת ממ"ה את הדף שחרר לה - אבל יתכן שם"ה רואה שאין יותר מקום בזיכרון הפיזי. במקרה זה מ"ה צריכה להחליט על ידי אלג' כלשהו מי הקורבן (דף הווירטואלי שתרגומו יוצא מהזיכרון הפיזי) ואז להחליט מה היא עשויה כדי להוציאו אותו בבטחה. להלן סכמה של תהליך ההוצאה:

- קודם כל צריך לסמך את הקורבן כ-**invalid** בטבלת הדפים (טכנית, לכבות את בית ה-**Valid** בכניסה המתאימה לדף הווירטואלי בטבלת הדפים) כדי שמי שייגש אליו ידע שבפועל בכניסה רשומה בטבלת הדפים כבר יש תרגום אחר.

• מלבד זאת יש לשחרר את התרגומים שאולי קיימים ב-TLB עבור דף וירטואלי זה (ומיכילים את התרגום שיש בטבלת הדפים עבור אותו דף). לשם כך מ"ה חייבת להריץ פקודה מיוחדת, **invalid page**, שבפועל עשויה לטבלת הדפים.

- oczywiście מ"ה צריכה להעתיק את הקורבן אל הדיסק - לא בהכרח מספיק להעתיק את תוכן הזיכרון הפיזי כי יכול להיות שחלק מהבתים שלו אותו דף נמצאים בקאים אחד המעבדים ובפרט שהערך של בית כלשהו בדף אחד הקאים אלה הוא **Modified** ככלומר שב�אש יש ערך שונה מזה שקיים בזכרון הפיזי - אז לא מספיק שמ"ה תעתק את הדף מהזיכרון לדיסק, יכול להיות במעבד כלשהו יש עותקים מעודכנים יותר. לכן מ"ה צריכה לעבור על כל הכתובות בדף ולשאול את כל אחד המעבדים אם יש לו את הכתובת ובמצב **Modified**, אם כן המעבדעונה שיש לו ומעביר את המידע מה�אש שלו עד ל זיכרון הראשי ואז עשויה לנתק בקאש שלו **invalidate**. רק אחרי שכל הכתובות בדף מכילות בזכרון את הערך העדכני ביותר שלן ניתן להעביר את הדף מהזיכרון לדיסק. הבהרה: מ"ה היא זו שעוברת על כל כתובות בדף וושאלת כל מעבד האם יש את הכתובת הזאת בזכרון המטען שלו.
- בקיצור: כשהמ"ה רוצה לנחות קורבן מהזיכרון להارد דיסק בזכרון לא בהכרח יש את המידע העדכני של כל כתובות - אז צריך קודם כל לעשות **invalidate** לקאש של המעבד ובסוף להעתיק את הדף ל זיכרון.

## החלפת הקשר - Context Switch

נלמד קורה בהקשר של זיכרון וירטואלי בהחלפת הקשר בין תהליכיים - Context Switch

# Context Switch

- **Each process has its own address space**
  - Each process has its own page table
  - When the OS allocates to a process pages in physical memory, it updates the process's page table
  - A process cannot access physical memory allocated to another process
    - Unless the OS deliberately allocates the same physical page to two processes
- **On a context switch**
  - Save the current architectural state to a dedicated memory location
    - Architectural registers
    - Register that holds the page table base address in memory
  - Flush the TLB
  - Load the new architectural state from memory
    - Architectural registers
    - Register that holds the page table base address in memory

אמרנו שטבלת הדפים של תהליך מגדרה את המיפוי בין המרחב הווירטואלי שלו (יש אחד לכל תהליך) למרחב הפיזי (היחיד במחשב). כמו כן אמרנו שמה שמגדיר את טבלת הדפים בפועל הוא הרגיסטר שמצויב אל כתובتها בזיכרון הפיזי - זאת מכיוון שברגע שטרוונים את הרגיסטר זהה כך שמצויב לטבלת הדפים של התהליך הנוכחי אז אפשר להתחיל לשתחם בטבלת הדפים של התהליך ולכן כל מה שצריך לעשות בהחלפת הקשר מבחינות טעינה מרחב הזיכרון הווירטואלי של התהליך הנוכחי זה לטוען אליו את הערך הנוכחי. בשבילו שהוא אפשר לטוען את הערך הנוכחי בכניסה של תהליך, כל פעם שיש החלפת הקשר מעתיקים את כל הרגיסטרים הארכ' של התהליך היוצא לאיזשהו אזור ייעודי בזיכרון שמכיל גיבוי של הרגיסטרים של תהליך ספציפי (בדר' זהו אזור זיכרון שיופיע למא) ובזמן זה מגבים גם את הרגיסטר שמצויב לטבלת הדפים (באנטול הוא נקרא CR3, כפי שנראה בהמשך השיעור) - אז מ"ה לוקחת את כל הרגיסטרים הארכיטקטוניים ומעתיקה אותם לזכרון הראשי. אח"כ, כשהתהליך חוזר ל clues, טוונים מאוזר הזיכרון המתאים לו את הרגיסטרים האלה ובין היתר טוונים את הכתובת של תחילת טבלת הדפים, שנשמרה בעת היציאה, חוזרת אל הרגיסטר המצויב לטבלת הדפים.

מלבד טעינת כתובות בטבלת הדפים אל הרגיסטר צריך בהחלפת הקשר לעשות ריקון, flush, לכל ה TLBs - נשים לב שלא צריך לבצע פלאש לאיזכרונות המטען: הקאשיים הם פיזיים ולכן לא מכילים שם דבר לפני תרגום! אם יש בעקב מטמוני וירטואליים (NELMD עליהם קצר מיד) אז צריך לרוקן גם אותם.

הערה: כזכור שבhalbת הקשר צריך גם לרוקן הציגור (the Pipeline) אבל זה לא קשר רק לזכרון וירטואלי: באיזשהו שלב הפעולות של האפליקציות של התהליך היוצא מפסיקות ל clues, אחריהן רצות פקודות של מ"ה, שמטפלות בהחלפת הקשר, וرك כזה נגמר עושם פלאש ואתחול לצינור כדי להתחיל להריץ את הפקודות של התהליך החדש.

## שיתוף - Sharing

- **OS may map two different virtual pages to the same phy. page**

- **DLLs in Linux/Windows**

- DLL pages are used by all processes, marked as read only  
⇒ one process cannot change the code of another process
- Same DLL may be loaded by a process twice
  - Two virtual pages of the same process may map to the same phy. page

- **Large malloc**

- OS maps all allocated pages to the same all-zero read only physical page
- In case of a write to one of these virtual pages
  - Access violation page fault
  - OS identifies the page mapped to the special 4K page
  - Allocates a "real" physical page (copy-on-write)

- **Shared Memory**

- OS maps virtual pages of two processes to the same physical memory

מ"ה יכול לבחור למפות כמה דפים-ווירטואליים לאותה כתובות פיזית. באופן כללי מצב בו שני תהליכי חולקים מביצעים לאותם דפים פיזיים נקרא שיתוף (sharing) בין תהליכי. נראה שני מקרים נפוצים לכך:

1. כאשר מmaps דפים אל השגרות שמסופקota ע"י מ"ה: למשל שני תהליכי קוראים לprintf - אין סיבה שהדפים הפיזיים שמכילים את הקוד של printf יכולים עותקים שונים שלה עברו כל תהליך, כלומר כל תהליך יכול לראות את אותו דף פיזי של מ"ה. מה שמש"ה עושה במקרה זה הוא למפות דפים וירטואליים (אולי שונים, כאמור בעלי VPN שונה) בתהליכיים שונים (ובפרט עם טבלאות דפים שונות) אל אותם דפים פיזיים בזיכרון.

2. דוג' נouceת נוספת לשיתוף היא ב-malloc: כאשר תהליך מקבל מקצה זיכרון דינמי ברוב המימושים מה שהוא מקבל זה דף שכלו אפסים - ומה שמש"ה עושה הוא בעצם לתת לכל התהליכי שהקצו זיכרון שמצויב לאותו דף

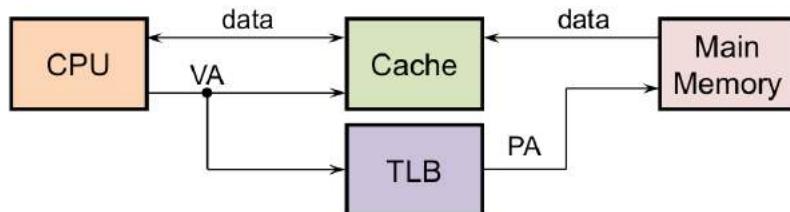
אפסים פיזי (בפועל זה גם לא חייב להיות דף פיזי, אלא פונקציה כלשהי, אבל נניח שהוא דף אפסים) והדף הזה מסומן בט"ד שליהם C-only Read Only ביט R דלוק ו-X כבויים ובפרט לא מושרים לכתיבה אליו. כל עוד שהתהליך לא השתמש בדף או שהוא רק קורא מהדף, ככלומר מכתובת הזיכרון שהקצתה דינמית, הוא מקבל אפסים - אבל ברגע שהתהליך מנסה לכתוב אליו הוא מקבל page fault, מ"ה בעת הטיפול בpage fault רואה שמדובר בדף האפסים המיעוד, מקצה לתהילך דף אמיתי, משנה את הכניסה בטבלת הדפים להצביע על הדף אמיתי וחזרה מה-page fault כך שהתהליך מנסה שוב לבצע את אותה כתיבה - וכרגע בחזרה מ-page fault הפעם הגישה לדף מצלילה. הבירהה: אם הקצנו זיכרון שמכיל הרבה דפים אך פונים רק לדף אחד מתוכם אז רק לדף זהה תתבצע החלפה בין דף האפסים המשותף לדף חדש והשאר עדין יצבעו לדף האפסים. המנגנון הנ"ל נקרא C.O.W: Copy On Write ניגשנות אליו ולכן הרבה פעמים אין צורך להחליף את דף האפסים בדף אמיתי בזיכרון.

נשים לב שהשיטוף קורה בא-ידיית החומרה אלא זהה החלטה של מ"ה למפות את הדפים שמקצים לתהילך אל דף האפסים זהה ולסמן את הביטים בכניסה המתאימה עבורו בטבלת הדפים כך שיגרמו ל.page fault. באופן דומה מ"ה היא זו שמשנה את התרגום המוצבע מכניסה זו בתגובה לאותה.page fault.

### זיכרון מטמן וירטואליים

## Virtually-Addressed Cache

- Cache uses virtual addresses (tags are virtual)
- Address translation only done on a cache miss
  - TLB is not in the path to cache hit



- Two virtual pages mapped to the same physical page
  - Must not reside in the cache together
  - On a cache miss, use a reverse TLB to verify that no other cache line already in the cache mapped to the missed physical address
- Flush cache on context switch
  - Alternatively: add process ID as part of the tag

בארכ' שיש לנו זיכרונות מטמן וירטואליים (Virtual-Addressed Cache) ה-cache שומר נתונים על-פי כתובותם הוירטואלית. התמരץ לשימוש במטמן וירטואלי הוא שזה חוסך את הגישה ל-TLB, ככלומר ה-tag-יכיל את הכתובת הוירטואלית ולא ידרש את התרגום שלה ב-TLB בשביל השוואה. למרות שלמדנו שבמקרה שיש TLB hit התרגום לוקח רק מחזיר אחד, גם אז שווה להתאים בשביל לחסוך אותו (בהתבה שヒיה hit - cache hit - cache miss)oen תיבא את הנתון - אבל במקרה זה התרגום CAN BE REQUIRED (בהתבה שיבא את הנתון - אבל במקרה זה התרגום CAN BE REQUIRED). החיסרונו באש וירטואלי הוא שחייבים לעשות לו ריקון בכל חילוף הקשר, של ה-TLB אינו העיכוב המשמעותי).

כלומר רק תהליך ייחד רואה אותו ואי אפשר לשתף בו מידע - זאת כי בכל תהליך כתובות וירטואלית כלשיי מחזיקה מידע שונה.

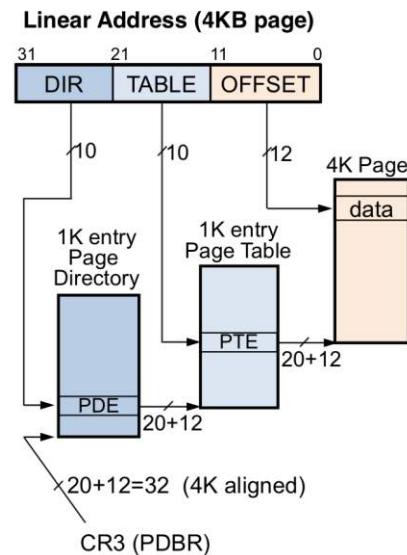
גם אם לא רוצים שהייה שיתוף בין ההלכים השיטה של מטמון וירטואלי מתאימה רק בזיכרונות מטמון קטנים: בשיזכרון המטמון גדול נרצה שכשתהlixir יוציא איז פעם הבאה כشيخזור הוא ימצא שם את המידע שהוא השאיר. במטמוניים קטנים גם כקה כנראה שהקash יתמלא בדברים אחרים לפני שתהליך ייחזר ואיז הגינוי לעשות אותו וירטואלי כי ממש לא יהיה שם זכר מההלכים שרצוי לפני כן.

עד חישרנו של מטמון וירטואלי בא לידי ביטוי כשרוצים לעשות לו *oopooch* חיצוני: *oopooch* הוא תמיד פיזי - הוא אפילו לא יודע איזה תהליך רץ כרגע - איז איך אפנה לקаш וירטואלי ואתאים אליו את הכתובת לה אני רוצה לעשות *oopooch* - זה פשוט.

## זיכרון וירטואלי ב-86x

### 32bit Mode: 4KB Page Mapping

- **2-level hierarchical mapping**
  - Page directory and page tables
  - Pages / page tables are 4KB aligned
- **CR3 points to the current Page Directory**
- **Upper 10 Linear addr. bits point to a PDE**
  - PDE (Page Directory Entry) provides a 20 bit, 4KB aligned base physical address of a page table
- **Next 10 Linear addr. bits point to a PTE within the given Page Table**
  - PTE (Page Table Entry) provides a 20 bit, 4KB aligned, base physical address of a 4KB page
- **Lowest 12 Linear Addr. bits provide offset within the selected 4KB page**
  - Point to the data bytes



עכשו עוברים לדבר על דוג' ספציפית: מימוש הזיכרון הוירטואלי בארכיטקטורה 86x של אינטל. הארכ' 86 תומכת בכמה מיני מצבים (modes) של זיכרון וירטואלי, ביניהם מצב שנקרא 32bit ומצב שנקרא 64bit. המצב הראשון עליו נלמד 32bit mode והוא היה טוב כל עוד הספיק לתהליכיים מרחב הכתובות שנייתן להתייחס אליו ב-32 ביט, ככלומר מרחב בגודל  $2^{32}$ =4GB. היום זה לעיתים קרובות לא מספיק אך כМОון שמעבדים של אינטל עדין ממשיכים לתמוך בו בשבייל אפליקציות ישנות, מ"ה ישנות וכו' - חשוב שמעבד ידע להמשיך להריץ קוד של תוכניות ישנות.

32bit mode הוא בעצם מודל היררכי: יש בו שתי רמות של טבלאות דפים. בשבייל לקבל את התרגום מדף וירטואלי לדף פיזי, בהינתן כתובות וירטואלית כלשהי, לוקחים את החלק העליון של הכתובת, מס' הדף הוירטואלי שהוא בגודל 20 ביט (כי הכתובת היא בגודל 32 ביט ו-12 ביט התחתיונים שייכים להיסט - צכור הדפים בגודל של 4KB), ומחלקים אותו לשני חלקים שונים: בעזרת שורה הביטים העליונים (битים 22 עד 31) פונים לטבלת תרגום ראשונה שנקראת (PDT) (Page Directory Table) - לכל תהליך יש רק טבלת PDT אחת ותמיד פונים אליה עם עשרה הביטים העליונים של הכתובת הוירטואלית.

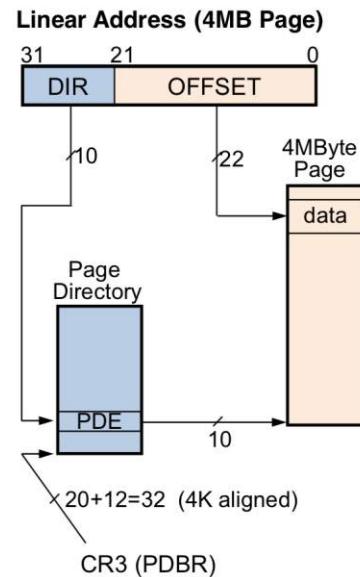
از לוקחים את עשרה הביטים העליונים של ה'כ'ו' ופונים אתה לכינסה ספציפית - הערך שרשום בciniseה זו ב-PDT לא מצביע לכתובת פיזית של הדף (בנוגוד לטבלת הדפים שלמדנו לעלה עד כה) אלא מצביע לעוד טבלת דפים שאליה ניגשים באמצעות עשרת הביטים הבאים במס' הדף, כולל גם בה  $2^{10}$  כניסות. הבהרה: כל(ciniseה ב-PDT אליה ניגשים באמצעות עשרת הביטים העליונים מצביעה לטבלה אחת מתוך 1024 אפשרויות, ואליה פונים באמצעות עשרה הביטים הנותרים - מה שנותר כדי להשלים את הכתובת של הדף הפיזי המתאים לדף הווירטואלי. "א גם עשרת הביטים העליונים וגם עשרת התħħħolot במס' הדף משמשים גם כאינדקס. מהטבלה האחורה מקבלים באמצעות את התרגום הפיזי של מס' הדף שלו - ואז בעורת offset פונים לבית ספציפי כלשהו באותו דף (פיזי) בזיכרון הפיזי.

היתרון של המבנה היררכי זה הוא חסכו מקום הנדרש עבור טבלת הדפים: אם תעשו חשבון תראו שטבלת הדפים הסטנדרטית היא ענקית, קל וחומר כשי המונן תħħolot והמנן טבלאות-דפים (נעsha את החישוב: יש  $2^{20} = 1M$  דפים, כל(ciniseה דורשת למשל ארבעה בתים, אז הטבלה בגודל  $2^{20} * 2^{22} = 4MB$  - ובמ"ה מודרנית יש מאות ואף אלפי תħħolot שיכולים להיות ביחד - נדרש המונן זיכרון). המבנה היררכי מנצל את העבודה שתħħolir בדרכ' משמש בחלק קטן מהמרחב הווירטואלי שלו - באמצעות המבנה הזה לא צריך לשמר תרגומים שלאזורים במרחב הווירטואלי שהטהליק לא משתמש בהם אלא מייצרים את טבלאות הדפים On Demand בהתאם לאזור הזיכרון הווירטואלי אליו הטהליק ניגש.

טבלת PDT תמיד קיימת, עבור כל הטהליק, אבל לא בהכרח כל 1024 הטבלאות שמוצבעות ממנה קיימות: כל עד הטהליק לא פונה לכינויים זיכרון המתאימים לטבלאות אלה (הטווח נקבע ע"י עשרת הביטים העליונים, כוללם על סמרק הciniseה-PDT) אז לא צריך לייצר לו עוד ועוד טבלאות דפים. בפועל כל טבלה דפים ברמה השניה מכילה 4MB (כי יש 1024 כניסות לדפים רציפים בגודל 4KB כל אחד). לכן רק כשהטהליק ירצה יותר זיכרון מקצים לו עוד טבלאות דפים ברמה השניה - אז הרעיון הוא לחסוך בזכרון שנדרש עבור המיפוי - כוללם עבור טבלאות הדפים.

## 32bit Mode: 4MB Page Mapping

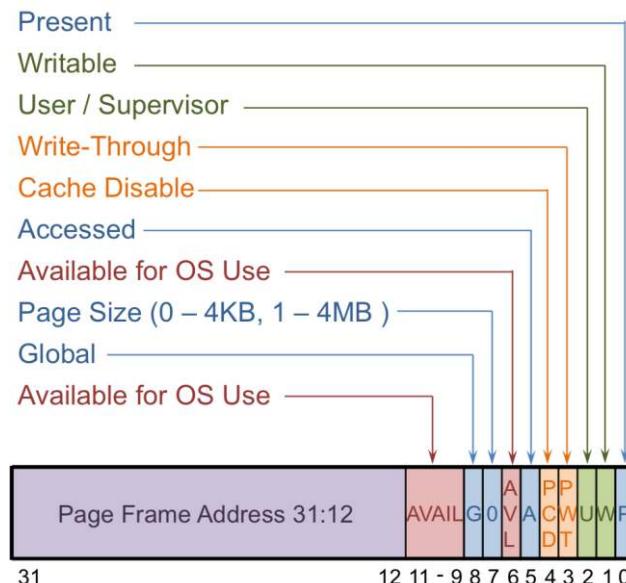
- PDE directly maps up to 1024 4MB pages
- Upper 10 Linear addr. bits point to a PDE
  - PS in the PDE = 1 ⇒  
PDE provides a 10 bit, 4MB aligned,  
base physical address of a 4MB page
- Lowest 22 Linear addr. bits provide  
offset within selected 4MB page
  - Point to the data bytes
- Mixing 4KByte and 4MByte Pages
  - Separate TLBs for 4MB pages and 4KB pages
    - Often used code (e.g., kernels) is placed  
in a 4MB page ⇒ frees up 4KB TLB entries
    - Reduces TLB misses and improves overall  
system performance



ב-32bit mode תומך גם בדפים גדולים יותר, דפים של 4MB, בניגוד לדפים של 4KB: כל(ciniseה ב-PDT יכולה להצביע או לטבלה שמצויבע ל-1024 דפים כפי שהסביר או להצביע שירות לדף גדול של 4MB - ועבור כל(ciniseה ב-PDT המערכת מבדילה בין שתי התצורות ע"י בדיקה האם המונן ה'page-size' ב-disk: אם כן זה אומר שהciniseה מצביע שירות לדף גדול, ואז לא עוברים דרך רמת התרגום הנוספת אלא לוקחים את כל 22 הביטים

הנותרים ומתייחסים אליהם offset בתחום דף של 4MB. הבהרה: מי שמחלית על זה המיפוי, ובפרט האם למפות כניסה ב PTD לדף-גדול או לדף רגיל, היא מ"ה.

## מבנה של כניסה בטבלת הדפים - PTE (Page Table Entry)



نبיט על הדגמים השונים שקיים בכניסה כלשדי בטבלת הדפים

- הביט Present זה ה-bit Valid שהוא האם בכניסה יש מיפוי תקין ל זיכרון הפיזי, או שיש בגישה אליה page fault (יתכנו page faults בעוד מצבים - למשל בא-ציותות להרשאות גישה).
- הביט Writable אומר האם מותר לבצע כתיבה לדף.
- הביט User/User/Superuser האם דף שייר לאפליקציה של המשתמש או שהוא שייך למ"ה - אם הביט דлок ואפליקציה תיגש אליו היא תקבל page fault, לעומת זאת למ"ה מותר לגשת לכל כתובות - זו דוג' להפרדה נוספת שנתקמת בחומרה בין מ"ה לבין קוד של המשתמש.
- הביט WB (Write-Back) מגדר האם הזיכרון בדף הפיזי זה הוא WT (Write-Through) או (WT) (cache).
- הביט Cache Disable אומר שאסור להכנס אך נתון מדף זה אל אף זיכרון מטמון כך שבפועל כל גישה אליו תהייב גישה ל זיכרון החיצוני. בדר"כ מ"ה תרצה להדilk את הדגל הזה כשהיא משתמש באזרז זיכרון כלשהו I/O (Memory Mapped I/O), שם גישה לתא זיכרון היא מתורגם לגישה להתקן פיזי: מtag הפעלה כלשהו או חוצץ של מכשיר קלט-פלט פיזי לדוגמת מדפסת. במקרה זה אם נשמר את מיקום הזיכרון הנ"ל בקוש איז עדכון הזיכרון ישנה ישנה במידע בזיכרון הפיזי (לפוחות בזכרון שהוא במצב WB) ואז במקומם להפעיל את התקן המצביע לו לא באמצעות ישנה - כי התקן הפיזי צריך שהכתיבה תועבר על ה-S-BUS של הזיכרון הפיזי שבפועל מגיע ל זיכרון I/O בהתקן החיצוני. איז במקרה של MMIO, למשל, מ"ה יכולה להגדיר דף שאינו Cacheable ובכך להכריח כל קריאה/כתבה לגשת פיזית ל זיכרון.
- הביטים Accessed ו Dirty Accessed עארים למ"ה בבחירה דף-קורבן - על Dirty דיברנו רבות והוא אומר האם צריך להעתיק את הדף מהמטמון ל זיכרון לפני פינוי מהזיכרון ו Accessed אומר האם ניגשו אל הדף מאז שהוא הובא למטרון/ל זיכרון (זה יכול להויל למ"ה בבחירה הדף להוצאה, למשל אם דף לא היה בשימוש אז פחות צריך לשמר אותו).

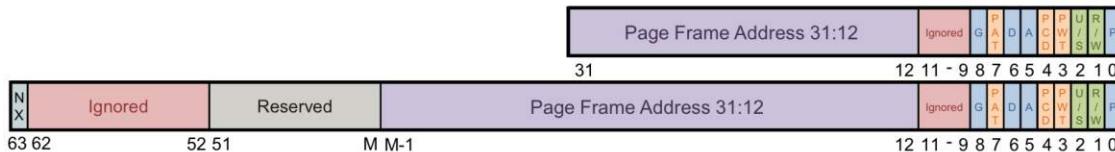
- הבית Global אומר שלא עושים flush לכניסה הצעת בטבלת הדפים בעת החלפת הקשר. מ"ה יכולה לסמך דפים Global ועל-ידי כך בעצם יוצרת מיפויים זרים בין תהליכיים שונים - למשל בדוג' שדיברנו עליה עם printf שנתקראת משני תהליכיים שונים ורוצחים שכל התהליכיים ימפו את הכתובת הוורטואלית של printf לאוטו דף פיזי - אין טעם להסיר את הניסה בעת החלפת הקשר.
- הביתים LVAIIL לא משפיעים על החומרה כלל והם לרשות מ"ה לכתיבת/קריאה לצרכיה וללאן' שונים שהוא עשויו לרצות למשש שנוגעים לטבלאות התרגומים (למשל בשבייל מגנון Copy On Write).
- שאר הביטים בכניםה מהווים את מס' הדף הפיזי שמהווה את התרגום מדף וירטואלי לדף פיזי.

המבנה של (PDE, כניסה ב-PDT), הוא מאד דומה ל PTE עם הבדל חשוב אחד: נוסף page size שאומר האם ה-PDE מצביע שירות על דף גדול של MB4 או שהוא פונה לטבלת דפים. במקרה של דף-גדול אין צורך להמשיך את התרגום אלא יש לגשת שירות לזכרון לפי שאר הביטים בכתובת כהיסט מתחילה הדף.

## Paging in 64 bit Mode

- **Physical address size grows to M>32 bits**

- Grow PTE from 4 bytes to 8 bytes to support >32 bits Physical address
- To keep each table 4KB  $\Rightarrow$  each table holds 512 entries (instead of 1K)  
 $\Rightarrow$  each table is indexed by 9 linear-address bits (instead of 10)  
 $\Rightarrow$  Large pages become  $512 \times 4KB = 2MB$  (instead of 4MB)



ניבור לדבר על מצב נוסף ב-x86: 64-bit mode. מצב זה בא לפטור שתי בעיות:

1. רוצחים מרחב וירטואלי גדול יותר מ 4GB לכל תהליך - בפועל ע"י שימוש ב 32bit אפשר למפות מרחב וירטואלי בגודל 4GB. במצב 32bit משתמשים בבייטים 12 עד 32 בשבייל ציון מס' הדף ובמצב 64bit נשימוש ביותר ביטים, ככלומר נמפה יותר דפים כלומר מרחב הווירטואלי יהיה גדול יותר.
2. רוצחים לתמוך במרחב פיזי גדול יותר מ 4GB במחשב: 4GB הם לא מספיקים לזכרון-פיזי בימינו (ולכן ודאי לא לזכרון וירטואלי): 64bit mode תומך בזכרון פיזי גדול יותר כי באופן דומה לכתובת וירטואלית ב-32bit פשטוט לא היה מספיק מקום לכתובת גדולה יותר, עכשו נוכל להשתמש ביחסים כדי לציין את מס' הדף הפיזי.

כשעברו ל-64bit mode היו צריכים לעبور מכינסה בטבלה שהיא בגודל ארבעה בתים (32 ביט) לכינוי בגודל שמונה בתים (64 ביט) בשבייל להחזיק את התרגומים. מכיוון שבטבלת דפים עד כה היו 1K מכינסה אז כשכל מכינסה היא בגודל ארבעה בתים צרייכים 4KB בשבייל כל טבלה - בדיקות הגדול של דף אחד, כלומר הטבלה (גם ה-Page Directory וגם כל Table) הייתה בגודל דף זה מאוד נוח. ברגע שכל מכינסה תופסת שמונה בתים אז הגדול של K1 מכינסות יהיה 8KB, שהם יותר מדף וירטואלי, וזה בעיתוי. מכיוון שרצוי להשאיר את גודל הטבלה להיות של 4KB כדי שתיכנס בדף אחד גם לאחר הגדלת המכינסות מ-4B ל-8B החלטתו להקטין את מס' המכינסאות בה כך שהיא תוכל רק 512 מכינסות. כתע הטבלה שוב בגודל דף אבל האינדקס עבורה, אחד מ-512 אינדקסים, הוא בגודל 9 בייטים - ראיינו שדיברנו על 32bit mode שימושים בעשרה בייטים כאינדקס לכל טבלה בהיררכית התרגומים -

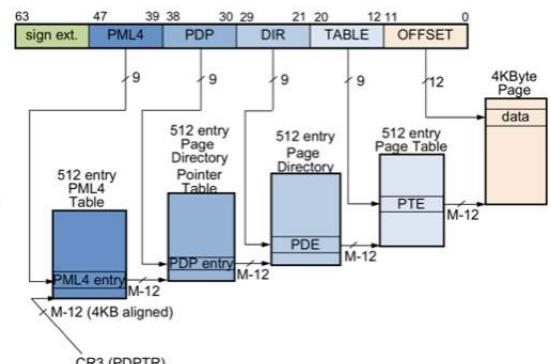
ב-mode 64bit נדרש לשמש רק בתשעת ביטים. אז שינוי גודל הכניסה ל-8B קרה קודם, כתוצאה מכך בכל טבלת דפים היו 512 כניסה וכתוצאה מכך פניה לכל טבלת דפים נעשית עם תשעת ביטים ולא עשרה. כתוצאה מהפניה עם תשעת ביטים, כל דף-גודל, שמחלייף טבלה שלמה, מאונדקס כרגע  $2^{32-9}=2^{21}$  סיביות, כלומר הוא בגודל  $2^{21} = 2\text{MB}$  ולא  $.MB$ .

עכשו רואים להגדיר את המרחב הווירטואלי ב-mode 64bit

## 4KB Page Mapping in 64 bit Mode

- Linear address size becomes 48 bits**

- Two new levels page tables are added
  - *Page Directory Pointer Table (PDP)* – indexed by LA[38:30]
  - *Page map level 4 table (PML4)* – indexed by LA[47:39]
- Each entry in PML4, PDP, DIR provides base address of next level table
  - maxphyaddr – 12 bits, 4KB aligned (for maxphyaddr = 40  $\Rightarrow$  28 bits)

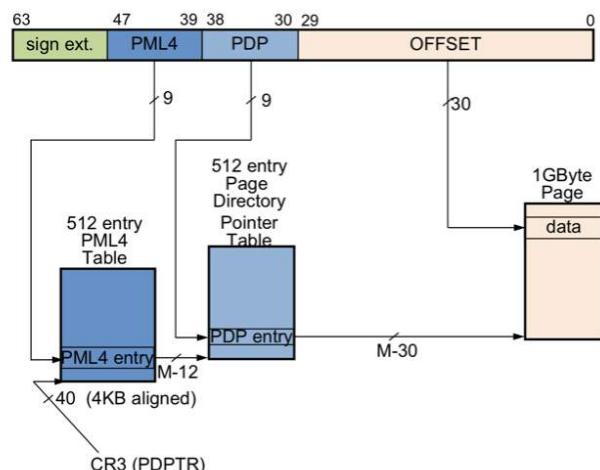
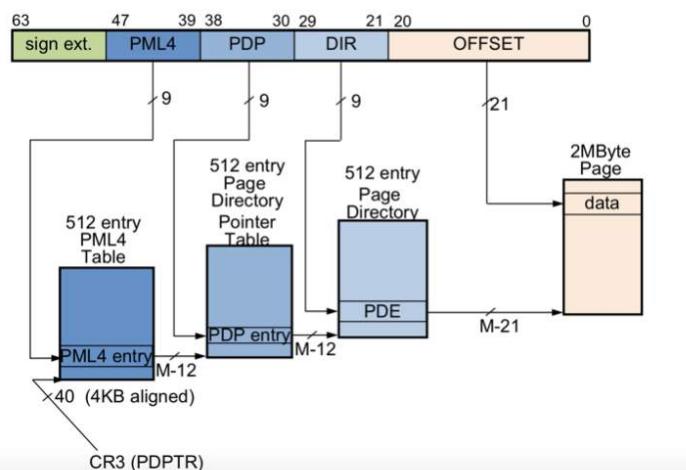


בפועל למרות שעברנו לעבד תחת השם "64 ביט" עדין לא תומכים היום בכל ה-64 ביט עבור ציון כתובות אלא הכתובת הווירטואלית במערכות של 64 ביט היא בפועל בגודל 48 ביט (מתעלמים מ-16 הביטים העליוניים) - עדין כל תהליך רואה מ"ז בגודל  $2^{48} = 256\text{TB}$  שזה המן.

במצב זה מוסיפים עוד רמות להיררכיה שראינו כshedיברנו על 32-bit-mode 64 bit-mode 32 כולם יש יותר טבלאות ביןיהם: עכשו שמקבלים כתובת וירטואלית, בת 48 סיביות כאמור, קודם כל לוקחים את תשעת הביטים העליוניים ואיתם פונים לטבלה בשם PML4, מהם מקבלים מצביע לטבלה אחרת בשם PDP (אחת מ-512 אפשרויות), ע"י גישה אליה עם תשעת הסיביות הבאות מקבלים טבלת DIR (מ-512 אפשרויות להצבעה אותה טבלת PDP), אחרת טבלת PT (באופן דומה טבלת ה-PDP מצביעה ל-512 טבלאות PT שונות ואנחנו מקבלים את הכתובת של אחת מהן) ורק ממנה מקבלים מס' דף פיזי אליו משרשרים את ה-offset ומקבלים data.

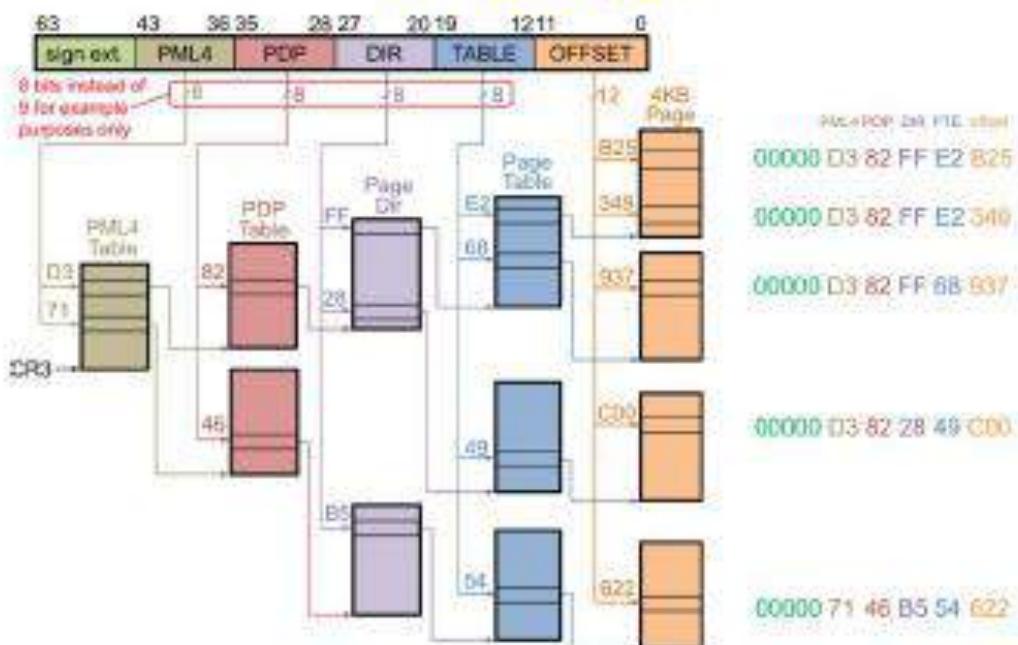
## 2MB Page Mapping in 64 bit Mode

## 1GB Page Mapping in 64 bit Mode



כמוון שגם מצב **64bit mode** תומך בגישה ישירה לדפים גדולים שכאמר עכשו הם בגודל של  $2^{21}$  ביטים, כלומר הם בגודל 2MB ולא 4MB כמו שהוא ב-**32bit mode**. בנוסף היוםאפשרות לחייב שירות מרמת-PDP (השנייה בהיררכיה) אל דף-ענף בגודל 1GB - ובמקרה זה היחסט בדף הוא בגודל 30bit.

## VM Example



דוגמא' לגישה לזיכרון ב-64bit-mode, לא נועבור על כולה אבל היא בא להראות שהמבנה ההיררכי הוא של עץ ובפרט עצ דיל: שינוינו בשביב הדוג' את כל השdotות להיות בגודל שמונה סיביות (ולא תשעה כפי שנעשה בפועל) כי אז אפשר להביע ערך של כניסה כלשהי בклות ע"י שטי ספורות הקסדצימליות (בסיס 16) ובאופן דומה להביע ערך של כ"ו שלמה - זה נוח לצורך הדוג'. במקרה זה עם ארבע רמות היררכיה ו-12 ביטים להיסט בתוך דף הכתובת נגמרה בבייט 43 (כלומר הכתובת היא מביט 0 עד בייט 43, סה"כ 44 סיביות).

- בדוג' מקבלים כתובת כלשהו, לוקחים את שמוות הביטים העליונים שלה, בדוג' D3, ופונים לכניסה זו בטבלה שנתקראת PML4, ממש לוקחים עוד שמוות ביטים, בדוג' 82, ופונים לרמה הבאה PDP וכן הלאה עד שמקבלים את התרגום.
- עכשו מסתכלים על כ"ו אחרית שכל הביטים שלא זהים לאלה של הראשונה בלבד בשדה היחסט - בתרגום הכל"ו משתמשים בדיק באותן טבלאות כמו של הכתובת הראשונה כי היא מתחפה לאותו דף וירטואלי וכן גם לאותו דף פיזי.
- עכשו ניגשים לכתובת שפונה לדף אחר אבל עדיין נמצא מתחת אותה רמה של PT (בשוף שדה 68 בסה"כ מייצרים עוד דף אחד בלבד במהלך התרגום - ברמה אחרי הרמה המשותפת, הפעם מוצבע מכניסה ב-PT).
- עכשו ניגשים לכתובת שפונה לדף חדש לגמרי בכתובת שונה בכניסה לשונה ב-Directory Page - במקרה זה נדרש להקצת טבלת PT חדשה, בתוכה דף חדש וכו'.
- גישה נוספת לכינסה שונה לגמרי ב PML4 (הכניסה המתאימה למס' הקסדצימלי 46) - עבורה נקצת טבלה אחת חדשה בכל הכניסה דרכן נועבור בהיררכית התרגום.

בזה"כ הדוג' מראה שפרשו רק חלק קטן מהעץ ובכל טבלה שモוקזית מתוך טבלת ה-PML4 יש הצבה ל-512 טבלאות PDP שכל אחת מצביעת ל-512 טבלאות DIR בrama הבאה וכך הלאה ל-512 טבלאות PT ש모וקזות לדפים בזיכרון וכן פורשים את כל העץ הרחב זהה - נוצר עז דليل מאוד שמייצג את מיפוי הזיכרון.

- **If the execute disable bit of a memory page is set**

- The page can be used only as data
- An attempt to execute code from a memory page with the execute-disable bit set causes a page-fault exception
- Setting the execute-disable bit at any level of the paging structure, protects all pages pointed from this entry

חידוש נוסף שהתווסף ב-64bit-mode הוא ה-bit Execute-Disable שמנוע גישה לכך כלשדי לשם הרצת קוד: כפי שהזכירנו בעבר הרבה פעמים וירוסים מנסים לתקוף ע"י גישה לאזרוי data, שם אפליקציות מורשות לכתוב מה שهن רוצות, כתיבת קוד זדוני של הוירוס לאזורי זה - ואז הוירוס רק צריך לשנות את ה-Instruction Pointer מה מעבד כדי לקפוץ לשם ולהתחליל לבצע פקודות של הוירוס אליו זה קוד לגיטימי - זה הרבה פעמים השלב הראשון שבוirus מתוכנן לעשות כדי לתקוף מחשב. כדי למנוע את זה מ"ה מגדרה דפים של data שאפשר להריץ מהם קוד כך שהוא מסמנת עבורם Execute-Disable ואז אם ה-IP עבר להצביע לדף זהה ומנסה להביא משם פקודות הוא יקבל page-fault עקב ניסיון לעשות Execute-Disable שהוא עוד דבר של מ"ה במאזן שלהם להקשות על הפורצים.

- **The most recently used PDEs and PTEs are cached in TLBs**

- Separate TLB for data and instruction caches
- Separate TLBs for 4KB, 2/4MB and 1GB page sizes
- TLB sizes in 4<sup>th</sup> Generation Intel® Core™ Processors:

|                  | <b>4KB<br/>pages</b>   | <b>2MB/4MB<br/>Pages</b> | <b>1GB<br/>Pages</b> |
|------------------|------------------------|--------------------------|----------------------|
| Instruction TLBs | 128 entries,<br>4 ways | 8 entries                | none                 |
| Data TLBs        | 64 entries,<br>4 ways  | 32 entries,<br>4 ways    | 4 entries,<br>4 ways |

- **In case of a hit in multiple TLBs**

- The largest page that hits is used

ראינו שבמעבד ישנו TLB נפרדים לפקודות (instructions) ולמידע (data) - בנוסף במעבד יש TLB עבור כל אחד מסוגי הדפים (רגיל-4KB, גודל-2/4MB, ענק-1GB). כמובן שככל שגודל הדף יותר קטן אז TLB שלו יהיה יותר כניסה - צריך הרבה דפים קטנים כדי לכטוט קטע גדול בזיכרון או יותר סביר שנעבוד עם הרבה דפים בגודל זה. למשל במקרים לפקודות לדפים גדולים יש רק שמונה כניסה TLB ולדפים גדולים מאוין בכלל. כמובן שככל הנתחנים הם מיקרו-ארכ' וכן אפשר להחליט מה שרוצים אבל השקף מסביר מה הגיוני לשימוש במעבד: ככל שהדף יותר גדול יהיו פחות entries ב-TLB שמתאימים לגודלו. הערה: ה-TLB לא מודע לחלוקת ההיררכית כי אליו מכניסים את התוצאה של כל החיפוש זהה, והוא בעצם נדרש לחסוך אותה - ה-tag שלו זה כל התרגום הארוך והוא בעצם נותן לנו את תוכן ה-PTE הסופית שהיינו מקבלים אם היינו עושים את כל ההליכה המיגעת, ז"א ברגע שיש hit TLB מקבלים את התוצר הסופי.

## PMH (Page Miss Handler)

# PMH – Page Miss Handler

- **PDP Cache is accessed with VA[47:30]**

- Includes both the bits used for accessing PML4 table and the PDP table
- PDP Cache hit  $\Rightarrow$  returns directly the PDP entry (still need PDE table access)
  - Saving both the PML4 table access and the PDP table access
    - ▲ Each of R/W flag and U/S flag are logical AND of their values in PML4 and PDP
    - ▲ XD flag is logical OR of PML4 and PDP values
    - ▲ The values of the PCD and PWT flags are taken for the PDP entry

- **PDE Cache is accessed with VA[47:21]**

- Includes the bits used for accessing PML4 table, PDP table, and PDE table
- PDP Cache hit  $\Rightarrow$  returns directly the PDE entry
  - Saving the accesses to PML4 table, PDP table, and PDE table

- **All 3 caches are accessed in parallel**

- Use hit from the lowest table that hits (which saves the most)

- **When the PMH accesses a page table during the page walk**

- It issues a "load" to the page table entry address, which as other loads
  - It first goes to the D\$, then to the L2\$, then the L3\$, and then to memory

- **In case of an iTLB or a dTLB miss**

- Request PTE from PMH
- PMH first tries the STLB (2<sup>nd</sup> level TLB); in case of an STLB hit
  - STLB returns the PTE, and saves the page walk
  - Starting at 4<sup>th</sup> Gen. Core™, STLB also caches PDEs for 2M/4M pages

- **In case of an STLB miss the PMH performs a page walk**

- It traverses the page table tree, starting at the root

- **The PMH includes caches to shorten the page walk time**

| cache      | Accessed with VA bits | If hits, returns |
|------------|-----------------------|------------------|
| PDE cache  | [47:21]               | PDE              |
| PDP cache  | [47:30]               | PDP entry        |
| PML4 cache | [47:39]               | PML4 entry       |

- **PML4 Cache is accessed with VA[47:39], and in case of a hit**

- Returns PML4 entry, saves accessing PML4 table
- Still need to access the PDP table and the PDE table

אמרנו שמשמש לנו PMH 2nd-TLB Miss ו-TLB Miss פונים ל-PMH ואו ציריך ללקת להביא את התרגום מהזיכרון -

אבל כפי שראינו ב-86x זו לא טבלת דפים אחת שהוא אליה אלא כמה, אחת בכל רמה. אז לכוראה ה-PMH

צריך לעשות ארבע גישות לזכרון רק כדי לקבל את התרגום - כלומר ע"י החלוקה לרמות של טבלאות חסכוño במקום אך גרמו לעצמונו לבצע יותר יותר גישות לזכרון - בפועל לא נסכים לכך שצריך ארבע גישות ולכן בכל רמת התרגום, צריך להיות לנו cache לטבלאות הדפים באוטה רמה. זה נקרא Translation Caches וחשוב להבדיל ביןיהן לבין ה-TLB, שנוטן תוצאה סופית, בעוד הם נתונים תוצאות ביניהם. נבהיר שככל עז התרגום וגם ה-PMH עוברים ריקון בכל החלפת תħallir.

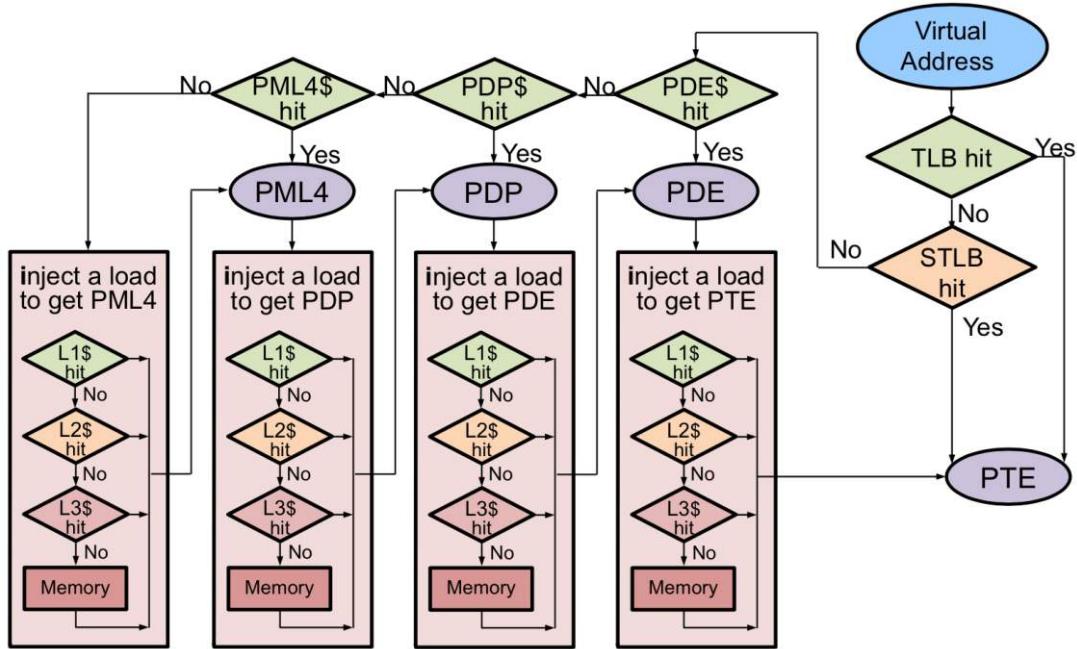
از כאשר אנחנו נתונים ל-PMH את הכתובת הוא מחלק אותה לפי המבנה ההיררכית ופונה במקביל לכל אחד שלושת ה-caches ובודק את הערך מה-Translation Cache שהיה לו hit בשלב ה-עומק בהיררכיה. הבהיר:

ה-Translation Caches הם חלק מה-PMH והוא תמיד מתחילה את התרגום באמצעותם.

לדוגמא ב-PML4 של כל תħallir יש 512 כניסה וברוב התħallicos כניסה אחת של PML4 פורסמת מרחב מספק (GB512=9/2^9=2^382^4) עבור כל תקנית שהטהħħilir מרייצ' - לכן בפועל הקאש של ה-PML4 ב-PMH יכול להיות מאוד קטן ואם יש לנו hit אז הוא מספק לנו יישור את הערך שכתוב ב-PML4 entry - אם הצלחנו זה חוסך לנו את השלב הראשון בתרגום ונוטן את ה-PML4 entry שאחננו צריכים.

שאלה מכם: עם איזה ביטים פונים ל-PDP cache, המטמון ברמה הבאה? לא פונים אליו עם ביטים 30 עד 38 אלא עם ביטים 30 עד 47 - בפרט ה-47-PDP cache יכול לשמר את הערך של PDP entry שמאופיינה באותו אינדקס בשתי טבלאות שונות שמוצבעות מכניות שונות ברמה מעל ולכן צריך את כל הביטים מהחלק הרלוונטי לכניסה ועד סוף הכתובת הוירטואלית כדי ליהות את הكنيיה הנכונה. אז הפניה ל-PDP cache עם כל ה-18 ביט הנ"ל כדי שהמטמון ברמה זו ידע לבדוק לאיזה PDP entry רוצים פנות. באופן דומה ניתן ל-**PDP Cache** עם ביטים 21 עד 47 כי הוא נותן את ה-PDE Entry הספציפית שמוצבעת מתוך טבלת PD ספציפית שמוצבעת מתוך טבלת PML4 ספציפית.

בבהרה: אם יש PDP hit זה לא משנה האם היה גם entry PML4 כי כבר יש לנו את ה-*entry* שRELONVENTIAL לשלב מתקדם יותר של התרגום - לכן בפועל hit ב PDP Cache חוטף לנו שתי פניות ל זיכרון: גם את הפניה לכינסה המתאימה ב-PML4 וגם לפניה לכינסה המתאימה ב-PDP.



בשוף רואים את העבודה של ה-PMH בתרגום מ כתובת וירטואלית לפיזית - קודם כל מנסים את מולנו ב-TLB, אם יש TLB mid מקבלים את תוכן ה-PTE הרלוונטי, אחרת אם יש לנו TLB 2nd-TLB Miss או הולכים ל-H-PMH, מוחפשים בכל שלושת הקאשימים שלו במקביל וכל מה שנותר אם לא מצאנו את הכתובת באף אחד מהם הוא לרכת ל זיכרון ולהביא את ה-PTE החסרה. איך עושים את זה? כאמור כל גישה ל זיכרון זה *load injection* של load מכתובת פיזית ע"י ה-H-PMH וכל אחת סוגיה מהכニיסות יכולה להימצא באחד הקאשימים - אחרת החיפוש מגיע עד ל זיכרון.

כתבו פה פרט נוסף אבל אני כבר מתייחסות אז תקרוו לבד:

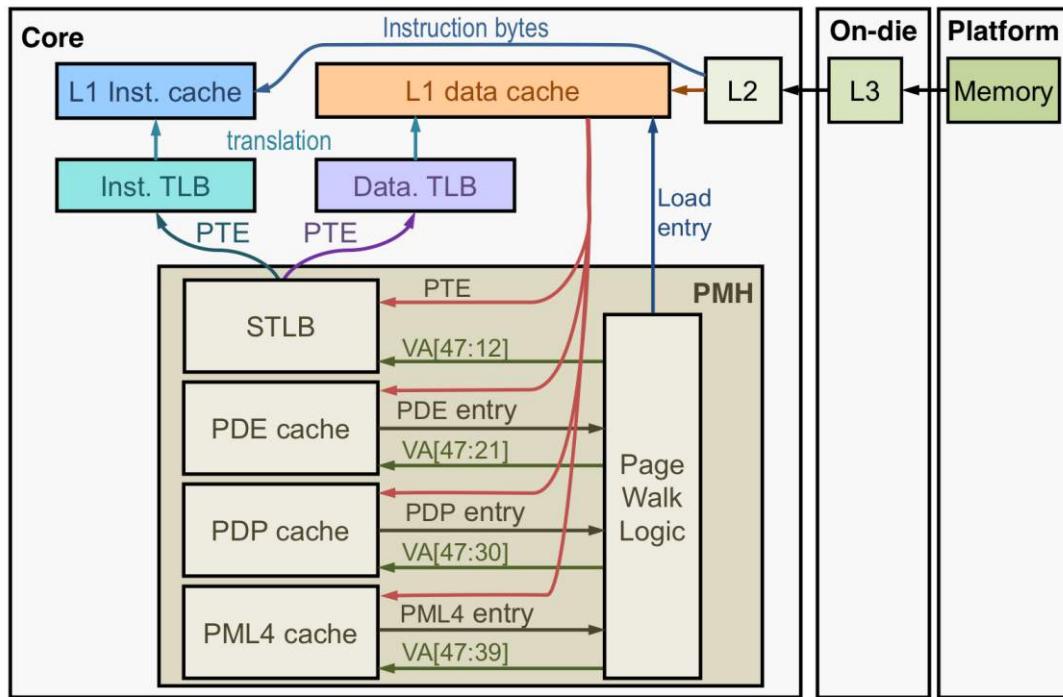
## PMH – Page Miss Handler

- Flags values

- R/W flag: logical AND of R/W flag in all levels
  - Can only write to a page if ALL levels allow it
- U/S flag: logical AND of U/S flag in all levels
  - Set as Supervisor only if ALL levels indicate Supervisor
- XD flag: logical OR of XD flag in all levels
  - Execution is disabled if it is disabled in any of the levels
- The values of the PCD and PWT flags are taken for the PDP entry

בש侃: סיכום של כל הקאשים ושלבי התרגום, זה דומה לצירור שראינו מוקדם רק שעכשיו הוספנו גם את ה PMH: PDE/PDP/PML4 cache שנמצאים בתוך ה-PMH (גם הגישה ל-STLB היא חלק מעבודת ה-PMH). החומרה יודעת לעשות את כל התהליך שראינו.

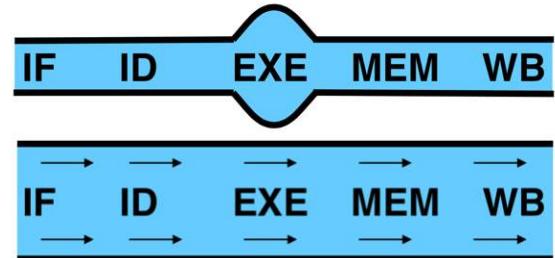
## Cache and Translation Structures



## הרצאה מס' 10: המשר Out Of Order Execution

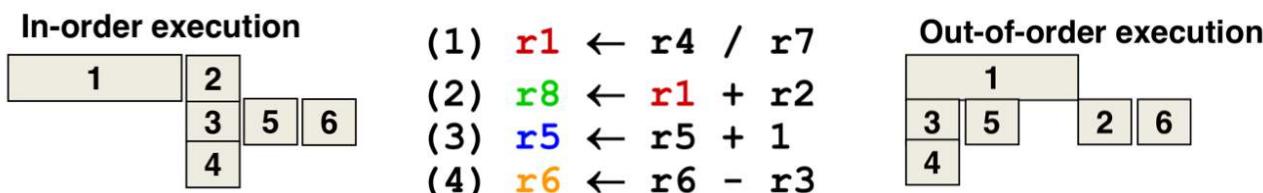
בשיעור שעבר התחלנו לדבר על מעבדים שפועלים ב-*Out Of Order*

- ◆ **Duplicating HW in one pipe stage won't help**
  - ❖ e.g., have 2 ALUs
  - ❖ the bottleneck moves to other stages
- ◆ **Getting IPC > 1 requires to fetch, decode, exe, and retire >1 instruction per clock:**



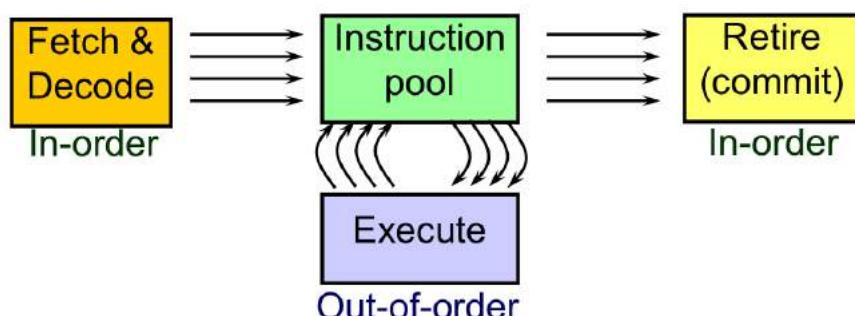
התחלנו מזה שאמרנו שבהרחבת הציגור (Pipeline) צריך להרחיב לכל האורך. כל התהנותות צריכה להיות באותו רוחב אחרית יהיה לנו צוואר-בקבוק בתחנה הצרה-ביותר. ראיינו צינור בו כל שלב יכול לטפל בשתי פקודות עוקבות בתכנית - זה יפה אבל מה אם רוצים צינור יותר רחב? אח"כ ראיינו את הבעה המרכזית ב-Pipeline, גם אם הוא רחב: פקודות שיש ביניין תלויות לא יכולות להישלח בו זמנית, לעומת זאת פקודה מאוחרת לא יכולה להתחליל לפני שהמקורות שלה (הרגיסטרים שהוא צורך) יסופקו ע"י פקודה מוקדמת יותר (אם אכן יש תלות - יש פקודות שאין תלויות באף פקודה ואז אין בעיה להתחליל את הביצוע שלהן). לכן אמרנו שבשביל לתכנן מכונה רוחבה לא מספיק להסתכל על פקודות עוקבות בתכנית ולקוטות שכולן יוכל לזרע בראץ (הן לא) אלא צריך לנצל "חלון פקודות" בו נחשף פקודות שאפשר לשולח לביצוע בלבד - החיפוש אחר פקודות לביצוע געשה ע"י ניתוח התלויות בין הפקודות ושלילה במקביל פקודות שאין תלויות בפקודות שקדומות להן.

ראיינו דוג' לתלויות בין פקודות:



בדוג': פקודה מס' 1 מבצעת חילוק ולכן לוקחת הרבה זמן. מבחינת התלויות בתכנית, יחד עם שליחת פקודה מס' 1 אפשר כבר לשולח את פקודות 3,4 ו-5 ואחריו שיסתיימו גם את פקודה 5 - כל זאת לפני שפוקודה מס' 1 מסתיימת. יחד עם זאת בביצוע פקודות על-פי הסדר אי אפשר להתקדם בכלל כי פקודה 2 תלואה בפקודה 1 בשביל הערך של r1. זהו התמרץ לביצוע (Out Of Order).

## OOOE – General Scheme



הראינו שהסכמה הכללית של מכונת OOO מתחילה ומסתיימת עפ"י סדר הופעת הפקודות בתכנית, כלומר זו: **order**, بعد שלב הביצוע נעשה **out of order**. בהתחלה עושים **In-Order Fetch** ו-**In-Order Decode** וממלאים **Instruction Pool**, זהו אותו "חולן פקודות" בו מוחפשים פקודות בלתי תלויות שאפשר לשלוח לביצוע שלא בהכרח בסדר הופעתן התכניתית. כמו כן מסיימים ב-**In-Order Retirement** כלומר פקודות מעשה משנות את מצב המכונה (ביצוע והופך לרשמי) לפי סדר הופעתן בתכנית. רק ביצוע הפקודות נעשה **Order Of Out** - חשוב לזכור זאת.

(Write WAW - Write After Read) WAR (Write After Read) Execution OOO שנוצרות תלויות מודומות כמו :After Write)

|     |                       |                                                                           |
|-----|-----------------------|---------------------------------------------------------------------------|
| (1) | $r1 \leftarrow R9/17$ | If inst (3) is executed before inst (1), r1 ends up having a wrong value. |
| (2) | $r2 \leftarrow r2+r1$ |                                                                           |
| (3) | $r1 \leftarrow 23$    | Called <b>write-after-write</b> false dependency.                         |

|     |                       |                                                                             |
|-----|-----------------------|-----------------------------------------------------------------------------|
| (7) | $r4 \leftarrow r3+r1$ | If inst (8) is executed before inst (7), inst (7) gets a wrong value of r3. |
| (8) | $r3 \leftarrow 2$     | Called <b>write-after-read</b> false dependency.                            |

תלוות נוספת נוצרת כאשר נשלח לביצוע פקודות שנמצאות אחרי קפיצה - כלומר יתכן שבפועל עד שהיינו מגיעים אליהן ביצוע In-Order תקוף והן כלל לא יבוצעו. גם את פתרון התלוות המודומות וגם את בעיית הביצוע הספקולטיבי פתרנו ע"י Register Renaming שבעצם אמר שבסמוך שפקודה כתובה לרגיסטר הארכיטקטוני של המכונה היא כתובה לרגיסטר זמני, פיזיקלי.

ביצוע renaming נעשה לפי סדר הופעת הפקודות בתכנית:

| <u>Renaming</u> |                       |          |                          |
|-----------------|-----------------------|----------|--------------------------|
| (1)             | $r1 \leftarrow 17$    | $r1:pr1$ | $pr1 \leftarrow 17$      |
| (2)             | $r2 \leftarrow r2+r1$ | $r2:pr2$ | $pr2 \leftarrow r2+pr1$  |
| (3)             | $r1 \leftarrow 23$    | $r1:pr3$ | $pr3 \leftarrow 23$      |
| (4)             | $r3 \leftarrow r3+r1$ | $r3:pr4$ | $pr4 \leftarrow r3+pr3$  |
| (5)             | jcc L2                |          |                          |
| (6)             | L2 $r1 \leftarrow 35$ | $r1:pr5$ | $pr5 \leftarrow 35$      |
| (7)             | $r4 \leftarrow r3+r1$ | $r4:pr6$ | $pr6 \leftarrow pr4+pr5$ |
| (8)             | $r3 \leftarrow 2$     | $r3:pr7$ | $pr7 \leftarrow 2$       |

|                  | $r1$ | $r2$ | $r3$ | $r4$ |
|------------------|------|------|------|------|
| Register mapping | pr5  | pr2  | pr7  | pr6  |

לפי סדר הופעת הפקודות בתכנית ממפים מהו הרגיסטר הפיזי האחרון שמחזק את הערך של רגיסטר ארכ' מסוים ואז כשקודם רוצה לקרוא רגיסטר ארכ' כלשהו יודעים לבדוק איזה רגיסטר פיזי צריך לספק לה את הנתון, למשל בסוף הקוד בשים ניתן לדעת שהערך העדכני ביותר של הרגיסטר  $r1$  נמצא ברגיסטר הפיזי  $pr5$ , ואז פקודה שחרrica את הערך הכי עדכני קוראת מהרגיסטר הפיזי המתאים. הבהרה: הערך של כל רגיסטר פיזי נכתב ע"י פקודה אחת בלבד. כך פקודה כלשהי תקרה את הערך הנוכחי בלי אפשרות להחלפת הערך של הרגיסטר הארכ' ע"י פקודה אחרת שהופיעה בתכנית מאוחר יותר אבל מתבצעת מוקדם יותר (תלוות WAR), כלומר כך השינוי בסדר

ביצוע הפקודות לא יוצר תלויות המדומות. תהליך של Register Renaming מסיע גם לגבי הקפיצה: אם פקודות בוצעו ללא צורך אז הם לא כתבו לרגיסטרים ארכל' אלא רק לרגיסטרים זמינים ואפשר לארוך את הפקודות בלי צורך לשחרר את מצב המכונה - יחד עם זאת במקרה זהה התרגום בין הרגיסטרים הפיזיים לארכל' יתפרק: אם פקודה כותבת ערך ב-7ק וזה המיפוי של 3 אז ברור שם בסוף היא לא מתבצעת המיפוי ל-3 אז לא תיקין כלומר הערך הרצוי של 3 אז לא ימצא בפועל ב-7ק וצריך לתקן את המיפוי שעודכן עקב פקודות שלא היו צריכות להתבצע. בהמשך השיעור נראה את האפשרויות לתקן המיפוי במקרה זהה.

אחרי שלבי ה- ROB (Reorder Buffer) וואז מקבלים את הפקודות בשני מבנים, Fetch, Decode :out of order RS (Reservation Stations) מה-RS היא *in order*. הказאת פקודות אליהן נעשית *in order*. הוצאה מה-RS היא *out of order*

## Reservation station

- Pool of all "not yet executed" instructions

- Holds the instruction's attributes and source data until it is executed

| RS  |   |      |   |      |      |  |
|-----|---|------|---|------|------|--|
|     | v | src1 | v | src2 | Pdst |  |
| add | 1 | 97H  | 1 | 12H  | 37   |  |
|     |   |      |   |      |      |  |

- When instruction is allocated in RS, operand values are updated

| Operand from           | Data valid | Get value from |
|------------------------|------------|----------------|
| architectural register | 1          | RRF            |
| physical register      | 0          | ROB            |
|                        | 1          | Wait for value |

- The RS maintains operands status "ready/not-ready"

- Each cycle, executed instructions make more operands "ready"
  - The RS arbitrate the WB busses between the units
  - The RS monitors the WB bus to capture data needed by awaiting instructions
  - Data can be bypassed directly from WB bus to execution unit
- Instructions whose all operands are ready can be *dispatched* for execution
  - Dispatcher chooses which of the *ready* instructions to execute next
  - Dispatches chosen instructions to functional units

בכל מחזור שעון בוורדים מס' פקודות מתוך RS לשילוח לביצוע - הבחירה של פקודות שיוצאות מה-RS אינה לפי סדר הופעתן בתכנית אלא לפי המוכנות של הנתונים שהן אמינות יחידות הביצוע השונות הנדרשות כדי לבצע אותן. מכיוון שממלאים את ה-RS לפי סדר אבל מוצאים שלא עפ"י הסדר אז מהר מאוד נוצרם בו "חוררים" - מי שומר על הסדר זה ROB שהוא FIFO כאמור פקודות וכוכחות *in order* וויצאות *out of order*. הסיום של פקודות גורם לעוד נתונים להיות מוכנים ולעוד יחידות פנויות וכך בסופו של דבר מתאפשר להוציא עוד פקודות לביצוע מה-RS ועוד פקודות לסיום (Retire) מה-ROB. כשפוקודה יוצאה מה-RS ל-*Retire* אז אפשר להתחייב על שינוי מצב המכונה בעקבותיה וכך נעשה כתיבה בפועל ל זיכרון או העתקת רגיסטרים פיזיים לרגיסטרים ארכל'.

ראיינו שב-RS לכל פקודה נרשמים המקורות/ארגוני שלה - אם הם מוכנים כשהפקודה נכנסת ל-RS אז שומרים את הערך שלהם ואם לא אז את שם הרגיסטר הפיזי לו מוכחים. כמו כן ראיינו שלכל פקודה ב-ROB מוצמד רגיסטר פיזי, מספרו הוא מס' הcnisa ב-ROB בו הפקודה מוקצת. לרגיסטר פיזי זה הפקודה כותבת את החישוב שלה והוא יתפנה כשהיא תסימם את הביצוע.  
ראיינו את המסלולים מה-RS אל הdzits Units Execution והזכירנו שגם הפורטים, דרכם עוברות הפקודות לביצוע הם משאב שפקודות מתחרות עליו כי ברגע שפורט התפנה זה אומר שהExecution Unit הרלוונטיות לו פניות.

אמרנו שיש מעקפים של ערכים שמחושבים ע"י פקודות כר שניתן להעביר אותן גם לפני שלב ה-Retirement RS יכול לשלוח פקודה לביצוע ואח"כ ולהעביר את הערך שהיא חישבה לפקודה אחרת ב-RS ברגע שהוא יהיה מוכן. ישנים כמה סוגים של מעקפים:

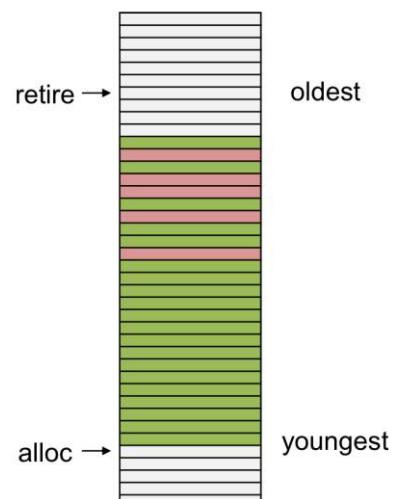
- מעוף בטור יחידת-ביצוע כלשהו (זה נקרא מעוף ברמה 0).
- מעוף בין יחידות ביצוע שונות באוטו פורט (רמה 1).

מעוף בין יחידות שונות בפורטים שונים: לכל פורט יש את שני המקורות שלו שנשלחים ליחידה ביצוע ואת הערך שהפקודה בפורט זה חישבה - למשל עבור יחידת חיבור הערך יהיה תוצאת החיבור שהפקודה ביחידה זו חישבה. כאמור הערך יכול להיות מועבר במעוף לפקודה מסוימת וכל פעם שמקור ערך חדש הוא מיחידה כלשהו (כלומר תוצאה של פקודה כלשהו) הערך נשלח ב-RS כדי להעיר את כל מי שתלו בו.

ראינו בפירוט את ה-ROB שבנוי FIFO:

#### ◆ The ROB

- ❖ Instructions are allocated in-order
- ❖ After an instruction is executed
  - marked as *executed* ⇒ ready for retirement
- ❖ An executed instruction can be retired once all prior instructions have retired
  - Once oldest instructions in the ROB are ready to retire, they are retired
  - Instructions are retired in-order
- ❖ Upon instruction retirement
  - Copy the value from its physical register to the architectural register
  - Its ROB entry is released



כותבים אליו In-Order ומסמנים בו מתי אפשר לעשות Retirement לפקודה - בשקף כל הפקודות שהסתינו מסומנות באדום ורק ברגע שנוצר רצף של פקודות שישיהם מתחילה ה-ROB אפשר לעשות Retire עבורן - ז"א אפשר להתחייב על פקודה רק כאשר להתחייב גם על כל הפקודות שהקדימו אותה. כאמור אפשר לעשות את זה לכמה פקודות במקביל, למשל במכונה ברוחב 4 אפשר לעשות Retirement לפחות ארבע פקודות ביחיד.

#### ◆ Faults and exceptions are served in program order

- ❖ At EXE: mark an instruction that takes a fault/exception
  - Divide by 0, page fault, ...
- ❖ Instructions older than the faulting instruction are retired
- ❖ When the faulting instruction retires – handle the fault
  - Flush the ROB
  - Initiate the fault handling code according to the fault type
  - Re-fetch the faulting instruction and the subsequent instructions

#### ◆ Interrupts are served when the next instruction retires

- ❖ Let the instruction in the current cycle retire
- ❖ Flush subsequent instructions
- ❖ Initiate the interrupt service code
- ❖ Fetch the subsequent instructions

דיברנו על חריגות/פסיקות במכונת OOO:

اذכור: חריגה (Fault/Exception) זה אירוע שבב מפוקהה מסוימת, למשל 0-div, ואוותה פוקהה היא זו שצריכה לקבל את הטיפול, לעומת זאת פסיקה (Interrupt) זה אירוע חיצוני וצריך לטפל בו ברגע שהוא מגע - לכן ניתן להלביש אותו על כל פוקהה.

אמרנו בנוגע לחריגות שכשפוקהה ביצה חלקה ב-0, למשל, אז בזמן הביצוע עצמו, שלב Execution, רק מסמנים את הפוקהה עם הסימון fault אבל עד לא מטפלים בשגיאה כי החישוב כולם עדיין ספקולטיבי וננדע בוודאות האם צריך לטפל בחיריה שנוצרה רק כאשר הפוקהה תהיה מוכנה ל-Retirement, ואכן רק בשלב ה-Retirement של פוקהה חורגת נטפל בחיריה שלה, אז כל הטיפול בחיריה נעשה in-order ורק כשפוקהה הגיעה לאם היא קיבלה fault אז נטפל. עושים זאת מכיוון שאם למשל יתברר שביצעוו את הפוקהה שחרגה בעקבות שגיאת-חיזוי בקפייה שקדמתה לה בתכנית אז לא היינו אמורים לבצע את הפוקהה שחרגה כלל (ולכן גם לא אמורים לטפל בחיריה שלה), לעומת זאת אם אותה פוקהה שחרגה הגעה ל-Retire אז בהכרח עשינו Retire גם לכל הפוקהות שהיו לפניה (ובפרט לכל פוקהה קפייה שהייתה) - אם הגענו לשלב זה בשלפ זהה של הפוקהה שחרגה עברה ריקון מהצינור, אז פוקהה זו באמת הייתה צריכה להתבצע ובתווך צריך לטפל בפסיקה שלה.

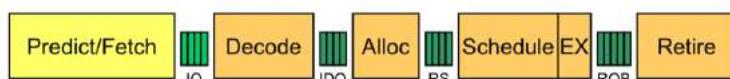
לABI פסיקה, שגיאת שדורשת טיפול מיידי, למשל נפילת מתח או השחתת זיכרון: במכונת in-order הפוקהה שנמצאת בסוף ה-Pipe היא זו שמקבלת את הסימון ב-fault ולפוקהות אחרת עושים flush ומתחילה לבצע את הקוד שמטפל בפסיקה - רק אחרי שה-handler יגמר נביא פוקהות חדשות. באופן דומה ב-OOO בשביל טיפול בפסיקה מסמנים אותה כשייכת לפוקהה שעושה Retire כתע - ואז מטפלים בה באותו רגע.

עד כאן חזרנו על מה שלמדנו בתחילת הנושא של OOO.

## שלבי הצינור (Pipeline) במכונת OOO

نبיט על השלבים השונים במעבד OOO מצורן:

### Pipeline: Fetch



- ◆ **Fetch multiple instruction bytes from the I\$**
- ◆ **In case of a variable instruction length architecture (like x86)**
  - ❖ Length-decode instructions within the fetched instruction bytes
  - ❖ Write the instructions into an Instruction Queue

השלב הראשון הוא שלב ה-Fetch והוא גם בשלב בו נעשה החיזוי של פוקהות קפייה (ה-Branch Predictor) להגיד לנו מאיפה להביא פוקהות: פונים ל inst. cache וambilאים בבת אחת מס' פוקהות - זה קורה.in-order. דבר אחד שהוא ספציפי ל-x86 של אינטל: ב-x86 הפוקהות הן באורך משתנה - אורך פוקהה יכול להיות אחד ועד 15 בתים. זה אומר שכשקוראים שורה מה-instr. cache, למשל 16 בתים, קודם כל חייבים להבין איפה מתחילה וגמרה כל פוקהה - כי הפוקהות באורך משתנה. הדבר על זה יותר בשבע הבא בהרצאה שМОקדשת לארכל' של אינטל אבל בגודל הסיבה שיש לנו את המבנה Instruction Queue היא כדי לצור חציצה בין המצביע של הבאת 16

בתיים של פקודות לבין הפרדת בתים אלה לפוקודות - ב IQ בכל כניסה כותבים פקודה אחת שלמה אחורי שכבר זיהינו איפה היא מתחילה ואיפה היא נגמרה. ההפרדה הזאת אופיינית למכונות שמטפלות ב Instruction Set IQ.

## Pipeline: Decode

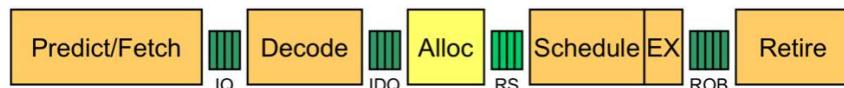


- ◆ **Read multiple instructions from the IQ**
  - ◆ **Decode the instructions**
    - ❖ For x86 processors, each instruction is decoded into  $\geq 1$  µops
    - ❖ µops are RISC-like instructions
  - ◆ **Write the resulting µops into the Instruction Decoder Queue (IDQ)**

הערה חשובה: למה צריך את התווים IQ ו-IDQ ותורמים באופן כללי? נניח שמתוך ה-`instr.cache` יש 16 בתים - כמה הפוקודות שנכונות בבתים אלה יכולה להשנות. ברגע שעשויים חוץ איז בכל מוחזור שעון מכניים אליו מס' פוקודות שונה אבל ההוצאה ממנו נעשית במס' פוקודות קבועה שמהווה מיצוע של רוחב הפס: אישרו BW sustain שנוול לצורך באופן קבוע. כנ"ל עבור העברה `-lsoops`: המעבר של ארבע פוקודות ארכל' לפוקודו מיקרו-ארכל' מניב מס' סoops שונה וכך גם ב-IDQ ההכנסה לא בהכרח נעשית בקצב אחד - אבל הקריאה ממנו היא כן בקצב אחד. שאלת חשובה בהקשר זה היא כמה IPC מכונה צו באמת מוציאה (*sustained IPC*) - בימינו מדובר בגודל 4 ה *sustained IPC* הוא בערך 2 - בעצם אם בזרה עקבית אפשר לדוחוף יותר פוקודות מההמכונה מאשר תמייד החוץים תמיד ישרמו מספיק פוקודות ותמיד יהיה מה לדוחוף קדימה. לעומת זאת אם נימצא במצב שבו מרווחים יותר פוקודות מאשר מושגניים אז בפועל לא יוכל להמשיך לבצע את התוכנית אלא מהר מאוד נתקע כשמחייבים לפוקודות אחרות שיוכנסו. בפועל אכן ממלאים בקצב מהיר יותר מהקצב בו מרווחים וכותזאה מזה כן ניתן לróż קדימה עם ביצוע התוכנית. בהקשר זה תמייד כדי לזכור שרוחב המכונה מגדר את ביצוע השיא (*peak performance*) ולא את הביצוע הממוצע - למשל אם פיזית אפשר לעשות בכל מוחזור Retire

לאربع פקודות עדין בפועל נראית הרבה תלויות בין פקודות ולכן המכוונה לא באמת תגיע לביצועים אופטימליים אלא יותר קרוב לביצוע של Retire לשתי פקודות בכל מחרוז.

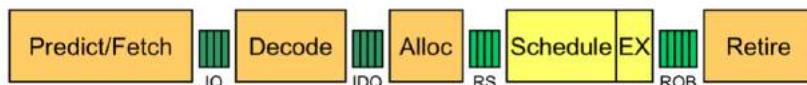
## Pipeline: Allocate



- ◆ **Allocate, port bind and rename multiple mops**
- ◆ **Allocate ROB/RS entry per mop**
  - ❖ If source data is available from ROB or RRF, write data to RS
  - ❖ Otherwise, mark data not ready in RS

השלב הבא הוא שלב ה-Allocate: הקצאת הפקודות והמקום עבורן בכל המיקומות השונים בציינור, וכן בשלב זה מתבצע גם ה-Renaming. נשים לב שככל מה שראינו מתחילה הציינור ועד כה עדין נעשה *in order*.

## Pipeline: EXE

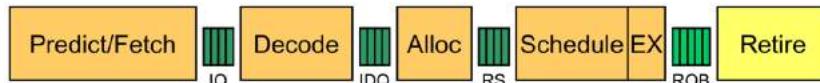


- ◆ **Ready/Schedule**
  - ❖ Check for data-ready mops if needed functional unit available
  - ❖ Select and dispatch multiple ready mops/clock to EXE
- ◆ **Write back results into RS/ROB**
  - ❖ Wake up mops in the RS that are waiting for the results as a sources
    - Update data-ready status of these mops in the RS
  - ❖ Write back results into the Physical Registers
  - ❖ Reclaim RS entries

בשלב ה-Schedule/Execute מסתכליםüberים לביוץ שלא עפ"י סדר התכנית - כאשר מבצעים scheduling על כל הפקודות ב-RS במקביל, מビינים איזה מהן מוכנות לביוץ (מבחןeo זו שהמקורות שלהם מוכנים וכן ה-Execution Units שהן דורשות פניוות), ווז מתחילה שלב ה-OOO - ככלומר עד כה רק השילוח לביוץ היה לא לפני הסדר.

כאמור בשלב זה בוחרים את הפקודות לשЛОח לביוץ - איזה פקודה כדאי לבחור? התשובה היא לא את הפקודה הכי ישנות (כפי שנעשה במכונת *in order*) אלא את הפקודות שיש הכי הרבה פקודות שתלויות בהן או את אלה שלهن יש את מסלול הביצוע הכי ארוך. האינטואיציה לכך היא שפקודה שאין אחראית שתלויות בה לא תעכיב את התכנית - הבעייה היא שלעתים קשה לדעת מראש איזה תלויות יש לפקודה אם כי זהו תחום מאוד רחב ויש הרבה אלג' לזכור כך שמאוד משפרים את הבעייה. לא נרحب מדי על הנושא זהה.

## Pipeline: Retire



### ◆ Retire oldest μops in the ROB

- ❖ A μop may retire if
  - Its “executed” bit is set
  - It is not marked with an exception / fault
  - All preceding μop are eligible for retirement
- ❖ Commit results from the Physical Register to the Arch Register
- ❖ Reclaim the ROB entry

### ◆ In case of exception / fault

- ❖ Flush the pipeline and handle the exception / fault

השלב האחרון בצינור של מכונת OOO הוא שלב ה-Retire. בשלב זה משנים את מצב המכונה באמצעות הפקודות המוקדמתות ביותר ב-ROB ברגע שהן בוצעו בהצלחה - למשל מעתיקים את הרגיסטר הפיזי לרגיסטר הארכ' ובכך ביצוע הפקודה נעשה רשמי - שינויו את מצב המכונה, זה משוקף למשל ע"י מצב הרגיסטרים הארכ'. דוג' נוספת לשינוי מצב המכונה ב-Retirement ערך לכתיבת זיכרון/מטען בפקודת store.(stage 4) לצורך ה-Retire נדרש לפחות ברוחב המכונה כי הוא מגביל מלמעלה את מס' הפקודות אותן ניתן לשים בכל מחזור. אז למשל במכונה ברוחב 4 אפשר להניח בוודאות גבואה שהשלב ה-Retire המעבד יכול להתחייב על ארבע פקודות בכל מחזור.

## טיפול בחיזוי שגוי (misprediction) במכונות OOO

הנושא הבא שנדון בו הוא איך לטפל ב-misprediction וספציפית איך לפתור את בעיית המיפוי בין רגיסטרים ארכ' לרגיסטרים פיזיים שהתקלקל במהלך הכנסת הפקודות השגויות: כזכור ביצנו renaming ובכך הגדרנו מיפוי מרגיסטר ארכיטקטוני לרגיסטר הפיזי שמחליף אותו (כלומר ניגשים לרגיסטר הפיזי במקום לגשת אל הרגיסטר הארכ'), למשל אם 1<sup>st</sup> מומפה ל-5<sup>th</sup> או פקודות שיבקשו את 1<sup>st</sup> יקרו בפועל את 5<sup>th</sup>. ברגע שראינו שהפקודות המבוצעות לא נכונות עקב MP (בין אם בוצעו/לא) המיפוי שלהם כבר קרה - כלומר המיפוי כבר התקלקל וצריך אייכשהו לחזור למצבם של מיפויים תקינים. יש כמה אפשרויות לעשות זאת.

אפשר 1 לטיפול בחיזוי שגוי במכונת OOO - טיפול בשלב ה-Retire:

## Jump Misprediction – Flush at Retire

### ◆ When a mispredicted jump retires

- ❖ Flush the pipeline
  - When the jump commits, all the instructions remaining in the pipe are younger than the jump  $\Rightarrow$  from the wrong path
- ❖ Reset the renaming map
  - So all the registers are mapped to the architectural registers
  - This is ok since there are no consumers of physical registers (pipe is flushed)
- ❖ Start fetching instructions from the correct path

### ◆ Disadvantage

- ❖ Very high misprediction penalty
- ❖ Misprediction is already known after the jump was executed
- ❖ We will see ways to recover a misprediction at execution

הדרך הכי פשוטה לטפל בחיזוי שגוי זאת היא שברגע ש- jump מתגללה כ-Miss predicted לסמן אותו ולחכotta לרוגע בו יהיה מוקן ל-Retirement. מכיוון שפרישה נעשית In-Order אז ברגע הפרישה של ה jump Miss predicted פשטוט למחוק את כל הפקודות במכונה יהיו צעירות ממנה ולכן ניתן היה לזרוק את כולן ע"י ריקון הצינור - פשוט למחוק את כל הפקודות אלה מהמכונה (כולל מה-ROB,RS...) ולהגיד ל-Fetcher להביא פקודות מהמקום הנכון. ברגע שהצינור ריק אנחנו יכולים לאתחל את המיפוי עבור כל הרגיסטרים שהיו בשימוש ע"י פקודות שנמחקו, ולאפשר לפקודות הבאות שייכנסו "لتפוס" את הרגיסטרים האלה במיפוי חדש.

הvisorון בשיטה זו הוא שאנו יודעים שפקודת הקפיצה היא MP כבר בשלב ה-Execute ולמרות זאת מוחכים עד שה-branch יהיה ב-Retire - הזמן מרגע הגילוי ועד שלב הפרישה של הפקודה הוא זמן מבואז שבמהלכו מביאים הרבה פקודות לא-טובות.

רוצים לשפר את הטיפול ב-Miss-prediction כך שיעשה לפני שלב פרישת פקודת הקפיצה, בפרט בשלב ה-Execute בו החיזוי השגוי מתגללה.

## אפש' 2 לטיפול בחיזוי שגוי במכונה OOO - טיפול בשלב ה-Execute :

### Jump Misprediction – Flush at Execute

- ◆ **When a jump misprediction is detected (at jump execution)**

- ❖ Flush the in-order front-end
- ❖ Instructions already in the OOO part continue to execute
  - Including wrong-path instructions (waist of execution resource and power)
- ❖ Start fetching and decoding instruction from the correct path
  - Note that the "correct" path may still be wrong ...
    - An older instruction may cause an exception when it retires
    - A older jump executed out-of-order can also mispredict
- ❖ Block younger jumps (executed OOO) from clearing
- ❖ The correct instruction stream is stalled at the RAT
  - The RAT was wrongly updated also by wrong path instruction

- ◆ **When the mispredicted jump retires**

- ❖ All instructions in the RS/ROB are from the wrong path
  - ⇒ Flush all instructions from the RS/ROB
- ❖ Reset the RAT to point only to architectural registers
- ❖ Un-stall RAT, and allow instructions from correct path to rename/alloc

כשהקפיצה מגיעה לשלב ה-Execute וידועים האם החיזוי היה שגוי יתכן שהפקודות שהיו בעקבותיה כבר יצאו מה-RS ושינו את המיפוי של הרגיסטרים ונמצאות בביוזע יחד עם פקודות שהיו לפני הקפיצה - אך אי אפשר לעשות flush לשלב ה-Execute. לעומת זאת יש חלקים בציינור בהם אפשר לבצע ריקון, למשל בשלב ה-Decode מתרבצע In-Order ולכן אפשר למחוק את כל הפקודות שנמצאות בשלב זה, כי הן הגיעו אחרי הקפיצה. כמו כן אפשר להגיד לשלב ה-Fetch להתחיל להביא פקודות מהמקום הנכון. لكن נרוקן את ה-Fetch/Decode-Stage ובביא פקודות חדשות אך נשים חסימה על שלב המיפוי, ככלומר לא ניתן לפקודות חדשות להשתמש במיפוי או לעשות Renaming למקורות שלhn כי המיפוי כבר מוקלקל. בכך מפרידים בין המתנה שהקפיצה הגיע ל-Retire לבין הבאת הפקודות מהמקום הנכון.

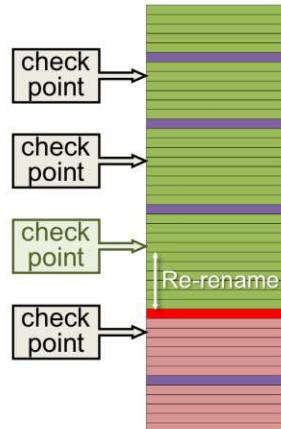
כאשר הקפיצה הגיעו לפרקיה אז בודאות היא הפקודה הכי ותיקה במכונה (פרקיה נעשית Order 0) ולכן אפשר למחוק את כל מי שנשאר בחלק הביצוע (Execute) של הציגור ולא תחול את טבלת המיפוי בין הרגיסטרים הארכ' לפיזיים (זה חוקי כי אין שום פקודה שכרגע בביצוע - כל הפקודות שנכנסו מהמסלול הנוכחי עוכבו בשלב ה-Renaming). השלב האחרון יהיה הסרת המחסום על המיפוי ואפשר לפקדות שהובאו מהמסלול הנוכחי להחיל בעשותה branch. כאמור באופן זה יצרנו חפיפה בין תחילת הטיפול בפקודות נכונות לבין המתנה שה-Retire.

גם הפתרון שראינו עתה לא מספיק טוב - רוצים שהוא עוד יותר מהיר ובפרט ממש להסיר את הפקודות הללו-טובות במכונה כבר בשלב exe של הקפיצה, לעומת זאת רוצים לבחوت שהקפיצה הגיעו לפרקיה וגם לא רוצים לעכב פקודות חדשות שנלקחו מהמקום הנוכחי מביצוע.

**אפש' 3 לטיפול בחיזוי שגוי במכונה 000 - טיפול בשלב Execute ע"י snapshots:**

## Periodic Checkpoints

- ◆ **Allow allocation and execution of instructions from the correct path before the mispredicted jump retires**
- ◆ **Every few instructions take a checkpoint of the RAT**
  - ❖ A snapshot of the current renaming map
- ◆ **In case of misprediction**
  - ❖ Flush the frontend and start fetching instructions from the correct path
  - ❖ Selectively flush younger instructions from the ROB/RS
  - ❖ Recover RAT to latest checkpoint taken prior to the mispredicted jump
  - ❖ Recover RAT to its state at the jump
    - Rename instructions from the checkpoint and until the jump
  - ❖ Allow instructions from the correct path to allocate



마다 פעם לוקחים צילום של טבלת המיפוי שיאפשר לשחזר את טבלת המיפוי במצב של לפני הקפיצה - טכנית לקחת צילום של מצב המיפוי לפני הקפיצה ובעת גילוי הטעות לשחזר ממנה את המיפוי. שחזר המיפוי יכול להיות יקר: יש הרבה מאוד רגיסטרים ארכ', צריך לזכור בכל snapshot. לכן לא רוצים לשחזר יותר מדי רגיסטרים וועושים זאת ע"י כך שקובעים מדי פקודות נקודת-שחזור בה נלקח snapshot, ואז כשמבצעים ביחסו操作ים ל-snapshot שלה וזה כל מה צריך לעשות זה לסגור את הפער בין המצב לפני הקפיצה שהגינו ביחסו עבורה לבין המצב הנוכחי, כלומר צריך לעשות Re-Renaming רק של כל הפקודות שבוצעו להן Renaming בין אותה נק' שחזור לקפיצה שהגינו ביחסו שלה - כך עושים שחזור של כל המיפוי שהוא משתמש לפני הקומן. הבדיקה: snapshot הוא מיפוי שאומר באיזה רגיסטר פיזי ממוקם כל רגיסטר ארכ' - לוקחים snapshot לפני כל הקפיצה וזה מאפשר לנו לסגור את הפער מהקפיצה ועד הזמן בו נתקן את המיפוי.

כאמור כשזהם חיזוי שגוי בפקודת קפיצה מיד עושם ריקון לכל החלקים בציגור שפועלים out-of-order וכעת אפשר להביא אליהם פקודות מהמקום הנוכחי (זה היה נכון גם בפתרון הקודם). מה שחדש כאן הוא שעושים ריקון סלקטיבי

כלומר ריקון של הפקודות שהן בביוץ OOO בתוך המכונה שהגינו אחורי הבראנץ' - ה-hot snapshot מאפשר לעשות זאת כי אפשר לתקן את המיפוי של הרегистרים.

## RAT Recovery

- ◆ **Restore RAT from latest check-point before jump**
- ◆ **Recover RAT to its states just after the jump**
  - ❖ Before any instruction on the wrong path
- ◆ **Meanwhile front-end starts fetching and decoding instructions from the correct path**
- ◆ **Once done restoring the RAT**
  - ❖ allow allocation of instructions from correct path

הערה: אם בכל מחזור שעון יודיעים לעשות פקודות אז שוחזר המיפוי יכול לחתך קצת זמן - אבל גם אם נחכה קצת זה לא נורא - ביצוע הפקודות יהיה עוד מחזור או שניים. כזכור שהמצב האופטימלי הוא שברגע שזיהינו misprediction flush לכל הפקודות השגויה, פקודות חדשות יכנסו ל-fetch / Decode ופקודות ותיקות (לפני הקפיצה) ישמרו על המיפויים הקודמים שלהם. מרגע שזיהינו misprediction flush עד שמשוחררים את המיפוי שלנו ובמקביל עושים איז מורוקנים את חלק In-Ordereds וחותמים את ה-Renaming. כלומר מאפשרים פקודות fetch מהמקום הנכון - ברגע שהשחזור מסתיים מסירים את המיחסום על ה-Renaming, כלומר מאפשרים פקודות חדשות לשזהר, Renaming, וכך באמת עושים ריקון אחורי חיזוי-שגוי במכונת OOO.

נניח שיש פקודה קפיצה שהתבררה C-MP וכתווצה מהחיזוי השגוי עשוינו גם Fetch, Decode וכך הלאה לעד פקודה קפיצה שגם היא "בצל" של השניה. Miss predicted

## חשיבות הגדל של ה-ROB וה-RS

### Large ROB and RS are Important

- ◆ **A Large RS**
  - ❖ Increase the window in which looking for independent instructions
  - Exposes more parallelism potential
- ◆ **Large ROB**
  - ❖ The ROB is a superset of the RS  $\Rightarrow$  ROB size  $\geq$  RS size
  - ❖ Allows for covering long latency operations (cache miss, divide)
- ◆ **Example**
  - ❖ Assume there is a Load that misses the L1 cache
    - Data takes  $\sim$ 10 cycles to return  $\Rightarrow$   $\sim$ 30 new instrs get into pipeline
  - ❖ Instructions following the Load cannot commit  $\Rightarrow$  pile up in the ROB
  - ❖ Instructions independent of the load are executed, and leave the RS
    - As long as the ROB is not full, we can keep executing instructions

בכל מחזור צריך להיות מוכנים לשלווח פקודות לביצוע - אי אפשר להרשות ברווח בציור כי ההשפעה שלהן על הביצועים מאד גדולה. טכנית RS גדול משמעו חלון פקודות גדול זהה חשוב - لكن בכל דור של מעבד מנסים

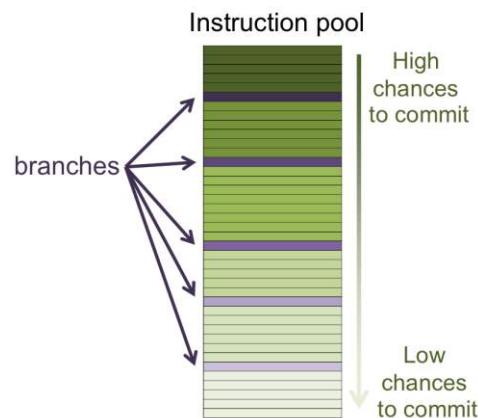
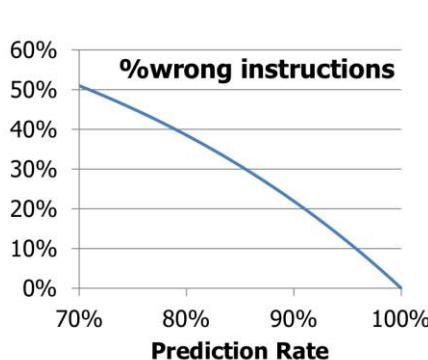
להגדיל את הגודל של ה-RS. לפועלה של ה-RS יש סיבוכיות ריבועית כי בבדיקה התלוויות של פקודה כלשהי צריך להשווות כל פקודה ב-RS לכל שאר הפקודות בו כדי להחליט מי מהפקודות מוכנות ואת מי הכי כדאי לשולח לביצוע - لكن לבנות אותו זהאתגר לא פשוט ואחד האתגרים המרכזיים בתכנון של מיקרו-ארך' הוא להצליח לבנות RS גדול.

צריך גם ROB מאד גדול: נניח שה-RS עובד במקבילות גבוהה אבל יש במעבד פקודה אחת שאחרות לא תלויות בה שהשווינו את ביצועה, למשל פקודת load ללא תלויות שקיבלה cache Miss ולכן ממתינה לכך שהיא תוכל לקבל את הנתון. בעצם ייקח מאות מוחזרים עד שהנתון יגיע אליה מהזיכרון וסביר מאד שככל הפקודות שנכנסו לאחריה כבר יבוצעו. במצב זה מה שמגביל אותנו זה הגודל של ה-ROB כי פקודה זו הופכת להיות הזקנה ביוטר ומכיון שהיא לא בוצעה אז למרות שהרבה פקודות שנכנסו אחריה כבר סיימו אי אפשר להוציא אותה כי היא יכולה לסיים את הביצוע וה-RS יכול להיות ריק בזמן שה-ROB תקוע - וברגע שה-ROB יתמלא אי אפשר להמשיך לעבוד, למשל אי אפשר לעשות Renaming חדש לאף פקודה כי אז צריך לכתוב את הערכים שלה גם ב-RS וגם ב-ROB שהוא מלא כאמור. אז ברגע שה-ROB מתמלא עושים Allocation stall וזה מאט את הביצועים של המעבד - לכן כדי שה-ROB לא יהיה צוואר הקבוק בונים אותו לפחות גודל פי שניים מה-RS. הוא בסך הכל תור FIFO ציקלי ואין בו צורך בסיבוכיות ריבועית או משוה כזו - אך יחסית קל להגדיל אותו כך שהיא בערך כפול מה-RS.

בشكل הבא תיאור של מצב אפשרי של ה-RS:

## 000 Requires Accurate Branch Predictor

- ◆ **Accurate branch predictor increases the effective scheduling window size**
  - ❖ Speculate across multiple branches (a branch every 5 – 10 instructions)



- בממוצע בין כל חמש עד עשר פקודות בתכנית יש פקודה קפיצה ולבן ב-RS, אליו פקודות נוכנות in-order, נראה פקודות/קפיצה/פקודות/קפיצה וכך הלאה (בפועל יתכו גם חורמים אבל לצורך פשוטות הם לא מופיעים בשקוף).
- נניח שהסתברות לחיזוי נכון נכון בודד היא כמעט 100%, ככלומר ההסתברות שהפקודות לפני הקפיצה הראשונה יגיעו ל-Retire היא כ-100%.
  - ההסתברות שהפקודות בין הקפיצה הראשונה לשניה יהיו Retired (כלומר שהקפיצה הראשונה והשנייה לא יתבגרו כ-mispredicted) היא קטנה כמעט ושוואה להסתברות לחיזוי נכון של שתי פקודות.
  - ההסתברות שהפקודות אחרי הקפיצה הראשונה והשנייה אבל לפני השלים יגיעו ל-Retirement היא כבר יותר קטנה כי צריך שהחזאי יהיה צדק לגבי שלוש הקפיצות.

- וכך הלאה.

ההסתברות שנעשה לפקודות **Retire** הינה חלון הפקודות, יורד מהר מאוד, שכן אם נגדיל מאוד את ה-RS ההסתברות שנעשה R אחרי הרבה מאוד קפיצות תהיה אפסית ואין סיבה לעשות RS עד כדי כך גדול כאשר חזאי הקפיצות לא משרה אוחזים גבוהים של Retirement לפקודות שבמורד החלון - שכן במכונת OOO כל שגדילים את ה-RS חייבים לשפר גם את חזאי הקפיצות (Branch Predictor) כדי לא להיכנס למצב שההסתברות לעשות Retire תהיה מדי וגודל ה-RS לא יהיה אפקטיבי.

לסיכום: עד עכשוו למדנו שהמטרה של מכונת OOO היא לאפשר ביצוע פקודות בלתי תלויות במקביל ע"י הסתכלות קדימה לצורך חיפוש פקודות שאינן תלויות אחת בשניה. ראיינו שהמכונה היא די מסובכת ויש דברים שצריך לטפל בהם כמו Register Renaming , Snapshots Register Renaming וכו' וכל אלה עושים הרבה יותר מסובכת - זו גם הסיבה שהמיקרו-מעבדים הראשונים שתוכלו ב-Execution OOO הופיעו רק ב-1995: רק אז הצלחו לשים על המעבד מספיק טרנזיסטורים כדי לתמוך בלוגיקה מורכבת כ"כ.

### ביצוע OOO של פקודות גישה לזיכרון (load/store OOO Execution)

## OOO Execution of Memory Operations

### ◆ The RS dispatches instructions based on register dependencies

- ❖ The RS cannot detect memory dependencies
 

```
store Mem[r1+r3*2] ← r9
load   r2 ← Mem[r10+r7*2]
```

  - Does not know the load/store memory addresses
- ❖ The RS dispatches load/store instructions to the Address Generation Unit (AGU) when the sources for the address calculation are ready
- ❖ The AGU calculates the linear (virtual) memory address
 

```
Segment-Base + Base Register + (Scale × Index Register) + Displacement
```

### ◆ The AGU sends linear address to the Memory Order Buffer (MOB)

- ❖ The MOB resolves memory dependencies and enforces memory ordering

עד כה הتمקדמו ביצועו של פיקוד רגילה. ניבור לדון ביצוע OOO של פקודות זיכרון, ככלומר פקודות **load/store**. נזכיר בסמנטיקה של פקודות **load/store**:

- בפקודת **store** המעבד שם ערך שנמצא בריגיסטר כלשהו, למשל 9, כתובות כלשהי בזיכרון, למשל 1000 או הכתובת **שמצוינית ע"י תוצאת הביטוי  $r3*2 + r2$** .
- בפקודת **load** המעבד טוען ערך מכתובת, גם היא יכולה להיות נתונה **ע"י ביטוי אריתמטי**, אל ריגיסטר כלשהו, למשל  **$r2$** .

נשים לב שתלוות אפשרית בין **load**-**store** היא כשה-**load** קורא ערך מכתובת כלשהי אחריו ש-**store** כותב אליה. ה-RS, שאחראי על פתרת תלויות בין רגיסטרים בלבד, לא יכול לדעת על התלוות בין גישות לזכרון, למשל האם **load** קורא ערך מכתובת ש-**store** שקדם לו כותב אליה, כי בזמן שה-RS מחשב את התלוות הערך של הכתובת של **store/load** אינו בהכרח ידוע (הוא עוד לא חושב). למשל ה-RS לא זיהה תלות בין **store** של ערך כלשהו לכתובת שמורה ב- **$r1$**  לבין **load** אל ריגיסטר כלשהו מהכתובת שמורה ב- **$r2$**  - גם אם  **$r1 = r2$**  מכילים

כתובות זהות. אך ה-RS הוא לא זה שמטפל בתלויות בין load/store - באופן כללי תפקידו הוא רק להחליט מתי הרגיסטרים מוכנים, בפרט במקרה של פקודת זיכרון תפkid-h-RS הוא להחליט מתי הרגיסטרים הדרושים לחישוב הכתובת מוכנים - רק ניתן להתחיל לחשב את הכתובת של load/store. לצורך חישוב הכתובת עצמה ה-RS שולח את פקודת הזיכרון אל ה-(Address Generating Unit) AGU (Address Generating Unit) שתפקידה הוא לחשב את הכתובת של גישת-הזיכרון. אחורי חישוב הכתובת הטיפול בגישת-הזיכרון עובר לרכיב בשם MOB (Memory Ordering Buffer) והוא הרכיב שمبין האם הכתובות של שתי פקודות loads/store זהות ודואג לסדר נכון בין פקודות אלה - למשל אם מתברר ש-store כותבת אל כתובת 1000 ואחריה בתכנית יש load שקורא כתובת 1000 אז אסור לתת ל-load לקרוא בפועל את הערך מ-1000 כל עוד ה-store לא כתב אליה כדי שה-load לא יקרא את הערך הישן (זה שהיה בכתובת 1000 לפני ה-store). אז ה-MOB י策וך לדאוג לכך שה-load יקרא את הכתובת מהמטען/זיכרון רק אחרי שה-store אליה.

## Load and Store Ordering

- ◆ **x86 has small register set ⇒ uses memory often**
  - ❖ Preventing Stores from passing Stores/Loads: 3%~5% perf. loss
    - P6 chooses not allow Stores to pass Stores/Loads
  - ❖ Preventing Loads from passing Loads/Stores: big perf. loss
    - P6 allows Loads to pass Stores, and Loads to pass Loads
- ◆ **Stores are not executed OOO**
  - ❖ Stores are never performed speculatively
    - There is no transparent way to undo them
  - ❖ Stores are also never re-ordered among themselves
    - The Store Buffer dispatches a store only when
      - The store has both its address and its data, and
      - There are no older stores awaiting dispatch
  - ❖ Store commits its value to memory (DCU) post retirement

store חיב להתבצע In-Order והכתובת שלו לזכור יכולת להתבצע אך ורק ב Retirement כי עד אז אנחנו לא יכולים להיות בטוחים ש-store צריך להתבצע, למשל רק כשהוא ב-Retire אפשר לדעת שאם פקודה לפניו לא הצליחה חריגה או שלא הייתה שגיאת חיזוי או שהוא כזה שימנע מה-store להתבצע. כמו כן אין אפשרות לבצע store בOPEN ספוקלטיבי ללא עדכון מצב המכונה: כשדיברנו על פקודות שמעדכנות רגיסטרים הענו פתרון של Register Renaming שיעזר לנו במקרה שצריך לבטל פקודות: הפקודות יכתבו לרגיסטרים זמינים ואז פשוט עשו להם ריקון במידת הצורך ונמחק את המיפויים וכך הלאה. במקרה של גישות לזכור אין אפשרות לשומר עותק זמני של הזיכרון בו כל פקודה כתובה לתא זיכרון זמני ורק כשהיא הגיע ל-Retirement נעתיק אותה, ככלומר באפשר לבצע undo ל-store. כיוון שאין דרך לעשות undo כנ"ל נותר לבצע את ה-store רק בסופו, ככלומר באפשרות לcancel הפקודה במאוחר יותר. סיבה נוספת לכך היא ששינוי בזיכרון יכול להוביל לשינויים נוספים במצב המכונה, למשל היום תקשורת של מעבד מול התקן פיזי מבוצעת ע"י MMIO, תחום כתובות שכתיבה אליה מתוגמת לפנויות קלט-פלט. אז איז אפשר להציג משהו ספוקלטיבית, לכתוב לכונן קשייה ספוקלטיבית וכו' - אך פקודות store חייבות להיות לא-ספקולטיביות ולכן stores תמיד מתבצעים in-order - אז מפהណון רק בביצוע loads שלא עפ"י סדר הופיעו בתכנית ו-stores תמיד יתבצעו In-Order.

לעומת `store`, פקודות `load` לא חייבות להתבצע ב-`Retirement` אלא אפשר לבצע אותן ספוקולטיבית החל מהשלב בו הכתובת מחושבת, במקרה שלנו שלב `Execute` בתנאי שוויידאנו שאין `store` שורצוה לכתוב לאוותה כתובה. תMRIIZ נוסף לכך שניצב אוטם OOO הוא שיש בתכנית המון `loads`: בתכנית אופיינית כל פקודה שלישית היא `load`, למשל מצב אופייני הוא `load` ומיד אחרי זה קריאה של הערך שנטען, וכך אם הינו מבצעים אותו - `load` `order` מעשה מכתיבים שכל המכונה היא מכונת `order-in-order` - סיכום לא בא בחשבון לבצע בפועל פקודות טעונה `.in-order`.

רוצים גם לאפשר ביצוע OOO של `load` על-פני פקודות `store`: ברגע שיעדים `sh-load` הוא לכתוב 1000 ו-`store` שחי אליו במקביל איןו לכתוב זו איז אפשר לעשות את ה-`load` - כאמור זו אחראיות ה-`MOB`.

## Store Implemented as 2 mops

- ◆ **Store decoded as two independent mops**
  - ❖ STA (store-address): calculates the address of the store
  - ❖ STD (store-data): stores the data into the Store Data buffer
    - The actual write to memory is done when the store retires
- ◆ **Separating STA & STD is important for memory OOO**
  - ❖ Allows STA to dispatch earlier, even before the data is known
  - ❖ Address conflicts resolved earlier ⇒ opens memory pipeline for other loads
- ◆ **STA and STD can be issued to execution units in parallel**
  - ❖ STA dispatched to AGU when its sources (base + index) are ready
  - ❖ STD dispatched to SDB when its source operand is available

פקודת אחסון ממומשת כשתי מיקרו-פקודות: `(STD)` ו-`(STA)` :store-data (STD) ו-store-address (STA)

• התפקיד של `store-address` הוא לחשב את הכתובת אליה שומרים - אפשר לעשות זאת ברגע שהרגיסטרים שדרושים לחישוב הכתובת מוכנים.

• התפקיד של `store-data` זה לחשב את הביטוי האריתמטי של הערך שנשמר בכתובת - גם הוא מחייב רגיסטרים כלשהם שיכולים להיות שונים מלהם מחייב ה `store-address`.

פקודת `store` היא פקודה יותר דחופה בשבייל `Out Of Order Execution` כי ברגע שנណע אותה נדע אם `loads` אחרים תלויים בה או לא, כי ברגע שיעדים מה הכתובת אליה כותבים איז ידיעים שטעינות מכתובות אחרות אין תלויות בה.

פקודת `load` יכולה להישלח לביצוע רק כשכל פקודות `store` שלפניה ידועות ואין לפניה פקודת `store` עם אותה כתובות.

בימינו קומפיילרים, שמיצרים את התלוויות בין רגיסטרים, מצליחים להימנע מטלויות לא הכרחיות בין `stores` ל-`loads`. ההפרדה בין `STA` ל `STD` היא כדי שלא יהיה צריך לחכות שה-`data` יהיה מוכן בשבייל חישוב התלוויות. אם לא הייתה את ההפרדה הנ"ל איז הפקודה היחידה של `store` הייתה נשלחת לביצוע רק כשה-`data` היה מוכן ועד אז `load` היה צריך לחכות.

**בדוג' רואים את התלותות האפשרות בין פקודות זיכרון:**

◆ **The MOB tracks dependencies between loads and stores**

- ❖ An older STA has an unresolved address  $\Rightarrow$  block load



- ❖ An older STA to same address, but Store's data is not ready  $\Rightarrow$  block load



**בדוג' הראשונה:**

- store שכותב את הערך 7 לכתובת 2000.
- store שכותב את הערך 8 לכתובת לא ידועה.
- load שקורא מכיוון 1000.

בדוג' זו load חייב לחכות עד שהכתובת אליה store כותבת תהיה ידועה.

**בדוג' השנייה:**

- store כותב לכתובת 2000.
- עוד store שכותב ערך לא ידוע לכתובת 1000.
- load שקורא מכיוון 1000.

. במקרה זה אנחנו יודעים בוודאות שיש ל-load תלות: הוא חייב לחכות ל-data-store.

از בעצם ה-load בדוג' הראשונה חסום על כתובות לא ידועה, ובדוג' השנייה חסום ע"י ערך לא ידוע - אלה שתि הסיבות בגיןן פקודה load יכולה להיות חסומה.

### MOB (Memory Order Buffer)- מבנה/פעולות ה-

# Memory Order Buffer (MOB)

- ◆ **Store Coloring**
    - ❖ Each Store is allocated in-order in the ***Store Buffer***, and gets a SBID
    - ❖ Each load is allocated in-order in the ***Load Buffer***, and gets LBID + current SBID
  - ◆ **Load is checked against all previous stores**
    - ❖ Stores with SBID  $\leq$  load's SBID
  - ◆ **Load blocked if**
    - ❖ Unresolved address of a relevant STAs
    - ❖ STA to same address, but data not ready
  - ◆ **MOB writes blocking info into load buffer**
    - ❖ Re-dispatches load when wake-up signal received
  - ◆ **If Load is not blocked  $\Rightarrow$  executed (bypassed)**

|       | LBID |
|-------|------|
| Store | -    |
| Store | -    |
| Load  | 0    |
| Store | -    |
| Load  | 1    |
| Load  | 2    |
| Load  | 3    |
| Store | -    |
| Load  | 4    |

|              | <b>LBID</b> | <b>SBID</b> |
|--------------|-------------|-------------|
| <b>Store</b> | -           | 0           |
| <b>Store</b> | -           | 1           |
| <b>Load</b>  | 0           | 1           |
| <b>Store</b> | -           | 2           |
| <b>Load</b>  | 1           | 2           |
| <b>Load</b>  | 2           | 2           |
| <b>Load</b>  | 3           | 2           |
| <b>Store</b> | -           | 3           |
| <b>Load</b>  | 4           | 3           |

ה-MOB אחראי על חסימת loads וSchedulerם לביצוע: לכל load מוקצתה ב-MOB באפר שנקרא load-buffer. load-buffer מוקצתה store-buffer ושניהם מוקצים מוקצים-in (בדומה להקצאת פקודות ב-ROB/RS). בנוסף לכל store/store-buffer id (LBID/SBID) load/store buffer id (LBID/SBID). בנוסף load/store SBID שומרים גם את ה-SBID הנוכחי, כלומר ה-id שנitin ל-store האחרון שהגיע לפני ה-load. עושים זאת כי load יכול להיות תלוי בכל stores-sha SBID שלהם קטן או שווה ל-SBID של אותו load - כלומר ע"י שימוש ב-SBID לכל load ידוע מי stores-lפניהם וככל עוד לא ידועים שאין לו-load כלשהו התנגשות עם אף store לפניו אי אפשר לבצע אותו. לכן stores-lפניהם וככל פעם שיודיעים את הכתובת של load-h MOD בודק תלויות של load-h כנגד כל stores-sha להם SBID יזomer קטן מה-SBID של load-h

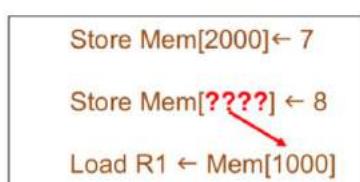
כפי שראינו בדוג' מקודם, ה-`load` נחסמ בשני מקרים:

- יש לפניו store שלא יודעים את הכתובת שלו.
  - יש לפניו store שהכתובת שלו זהה לכתוות mana ערים load אבל לא יודעים את הערך שהוא כותב.

load חסום נשאר להמתין ב load-buffer ובכל מחזור שעון-h MOB מנסה לבדוק האם הוסר תנאי החסימה של ה load - כלומר הוא משערת את החסימה של כל-h loads וברגע שיש כאלה שאין חסומים הוא מאפשר להם להתבצע (להביא את נתונים מה-Data Cache). נשים לב שיכולים להיות sh-load עובי מלהיות חסום על unresolved address

- ◆ The MOB predicts if a load can proceed despite unknown STAs

- Predict colliding → block Load if there is unknown STA (as usual)
  - Predict non colliding → execute even if there are unknown STAs



- ◆ In case of wrong prediction

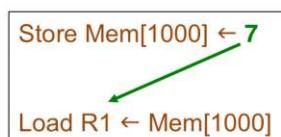
- The entire pipeline is flushed when the load retires

עוד לפני שיוודעים בודאות האם load תלוי ב-store לפניו או לא, למה שלא להשתמש בחזאי? אפשר ללמוד אותו: נניח שהייתה איזשהו load שהיכלה ל-store עם כתובת לא ידועה והתברר בדייבד שלא הייתה התנגשות בינוים - ה-load היה חסום לחינם. רוצים למנוע את החסימה בפעם הבאה - וכך מוסיפים חזאי שמהוה כל load לפי הכתובת שלו וכל פעם שמתברר שה-load לא היה תלוי (בפועל) מגדילים איזשהו מונה - ברגע שהמונה מגיעה לרף מסוים לוקחים סיכון ומציעים את-h-load בכל מקרה, למשל אם עשר פעמים גילינו-sh-load לא היה תלוי באפקודת אז בפעם הבאה לוקחים סיכון ושולחים אותו למטרון לקריאת הערך מהכתובת שלו - זה מאפשר לנו להמשיך להריצ' פקודות load למשך זמן מסוים stores שהכתובת שלהם לא ידועה. אם יתריר שטעינו ואיזשהו store כותב לכתובת של-h-load לו אפשרנו להתבצע איז ציריך לעשות flush לכל המכונה (כי הערך נקרא לתוכו איזשהו גיסטר ומרגע זה אולי שימוש קקלט לפקודות אחרות), אך כשה-load מגיע ל-h-Retirement ציריך לעשות flush של כל הפקודות אחרי-h-load ולהביא אותן מחדש.

הערה: ניתן לשככל את אותו חזאי כך שיגיד בדיק באיזה פקודות load תלוי ובאופן כללי ניתן לנו עד מידי - זה נקרא selective flush וזה מסתבר מאד - יש הרבה מאמרם בנושא ולא נרחיב על זה כאן.

## Store → Load Forwarding

- ◆ An older STA to same address, and Store's data is ready
  - ⇒ Store → Load Forwarding: Load gets data directly from SDB
  - ❖ Does not need to wait for the data to be written to the DCU



נניח ש-store כותב לכתובת 1000 ואחריו load קורא ממנה - בפועל ה-store יכתוב את הערך לקаш רק ב-  
ולכארה ה-h-load יצטרכ ללחכות עד אז - זה בעצם גדול כי ה-store יכול להגיע לפירישה אחרי הרבה זמן - לכן נעשה bypass בין הערך הנכתב לערך הנקרא: ה-store אמנם ישמור את הערך שלו לקаш רק בפירישה אבל ערך זה יועבר ל-load בשלב הרבה יותר מוקדם - זה נעשה ע"י כך שה-h-MOB מזהה את התלות ועושה Store-To-Load forward כלומר מעביר את הערך ישירות אל-h-Load.

אם יש store שכותב לכתובת 1000 ואחריו אחד שכותב לכתובת לא ידועה אז במקרה זה לא ניתן לפתור את המצב, ככלומר להתייר את-h-memory disambiguation, ונצטרך לחכotta עם המעקב כי לא בטוח מהו הערך הנוכחי שיש להעביר.

הנושא הבא שנדון בו הוא המקרה של פקודה load יש cache Miss

## DCU Miss

- ◆ **Blocking caches severely hurt OOO**
  - ❖ A cache miss prevents from other cache requests (which could possibly be hits) to be served
  - ❖ Hurts one of the main gains from OOO – hiding caches misses
  - ❖ Cache in OOO machine are **non-blocking**
- ◆ **If a Load misses in the DCU**
  - ❖ The DCU marks the write-back data as invalid
  - ❖ Assigns a fill buffer to the load, and issues an L2 request
    - As long as there are still free fill buffer more loads can be dispatched
  - ❖ When the critical chunk returns, wakeup and re-dispatch the load
- ◆ **Squash subsequent requests for the same missed cache line**
  - ❖ Use the same fill buffer

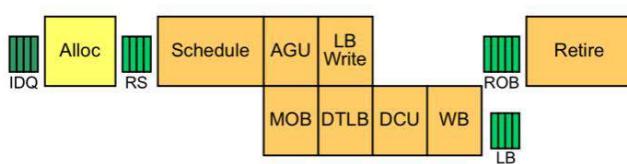
נניח שה-load עבר את כל הטיפול של השקפים הקודמים ונשלח למטרון להביא את הנתון שלו אך קיבל החטאה - במקורה זה ה-load לא מקבל את ה-data שלו וכן כל הפקודות שתלויות באותו load לא יכולות המשיך. כזכור בתכניות יש כל הזמן loads וזה אומר שהמתמונן במכונה OOO חייב להיות non-blocking cache, כלומר הוא חייב להיות מוכן לקבל בקשות מעוד loads גם אם יש החטאה - לאחרת זה יתקע במהירות כל מכונה OOO מלבד פקודות נוספות. בשבייל למש use-b-OOOE שמיים במתמון מההfers כמחושב, fill buffers, עשרה, וכל פעם שיש load עם החטאה מקצים עבورو חוץ נ"ל ושולחים בקשה ל-L2 cache לספק את הערך הדורש בחוץ זה - וממשיכים הלאה. אם יש עוד load ויש לו hit או cache או נתונים לו את הנתון, אם יש לו Miss אז באוטו אופן מקצים לו עוד fill buffer וכן הלאה. אם כל ה-load-buffer-h-L1 ואיז ישוחרר ה-load בשבייל load-buffer אחר ונוכל המשיך. ככלו יחוור מה L2 אל ה-buffer שלו, ייכתב ל-L1 ואיז ישוחרר ה-load בשבייל load-buffer אחר ונוכל המשיך.

از חשוב לזכור שמתמונן במכונה OOO חייב להיות מסוגל לשרת load גם תחת Miss וגם תחת hit.

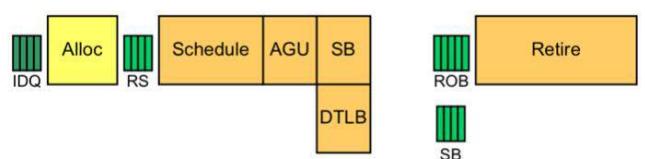
### מצב הצינור של load/store OOO Machine

:Allocation •

#### Pipeline: Load: Allocate



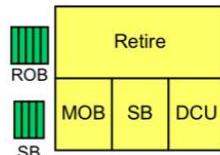
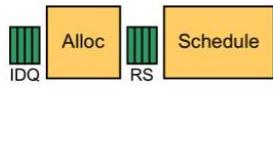
#### Pipeline: Store: Allocate



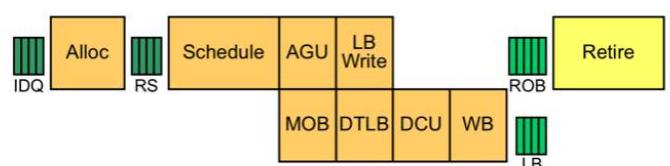
- ◆ Allocate RS entry, ROB entry, and Load Buffer entry for the Load
- ◆ Assign Store Buffer ID (SBID) to enable ordering
- ◆ Allocate ROB/RS
- ◆ Allocate Store Buffer entry

- ב-**RS Scheduling** עושה רק ברגע שהօפרנדים של הפקודה מוכנים לחישוב (ראינו שלא אכפת לו ממשו אחר), ברגע זהה קורה הוא מעביר ל-AGU את הפקודה שיחשב את הכתובת (הוירטואלית) וואז שולח אותה אל ה-MOB.
- במחוזר אחרי ה-AGU בו ה-MOB קבוע יש load המשיך - אז אפשר לשולח המرة לכתובת פיזית ולכתוב את התוצאה ל-DCU. עבור load שה-AGU חישב את הכתובת אם ה-MOB אמר שהוא חסום אז load ממתין בבאפר ובכל מחוזר load מחשב את תנאי החסימה ובלב מאוחר יותר כשה-MOB מגלה שתנאי החסימה הוסרו הוא שולח את ה-load למטרון.

## Pipeline: Senior Store Retirement



## Pipeline: Load: Retire



- When STA (and thus STD) retire
  - Store Buffer entry marked as senior
- When the DCU is idle  $\Rightarrow$  MOB dispatches a senior store
- Read senior store entry from the Store Buffer
  - Store Buffer sends the data and the physical address
- DCU writes the data into the specified physical address
- Reclaim Store Buffer entry

- Reclaim ROB, Load Buffer entries
- Commit results to RRF

- בסוף תהליך הפרישה של ה-load מעתיק את הערך שנטען מהרגיסטר הפיזי לארכ'.

עבור store זכרוains את הפיצול לשתי types:

- פקודת store-address שנשלחת לחישוב ב-AGU ברגע שהמקורות שלה מוכנים.
- פקודת .store-data

ברגע שתיהן מסויימות ה-MOB כותב את ערך ה-store buffer ל-store forwarding אם יש load שמחכה לערך זה. הכתיבה מה-store buffer אל הקאש מתבצעת רק ב-Retirement של ה-store - רק אז ידועים שהוא לא ספקולטיבי ואפשר לשולח אותו לקאש.

ברגע שפקודת store היא לא מפונה מה-MOB אלא מסומנת בו כ senior store, בשלב זה הכתיבה לזכרון נעשית דרך חוץ כדי לא לתפוס את המעבד לביצוע פקודות אחרות (גישה לזכרון היא ארוכה). כשהנתנו נכתב לזכרון ה-store נמחק גם מה-MOB ובכך הוא יוצא סופית מהמעבד.

## הרצאה מס' 11: מיקרו-ארQUITקטורה של מעבדי אינטל

Intel uArch

שבוע ש עבר דיברנו על כך שהצינור במעבד צריך להיות רחב ככל אורך כדי שלא יהיה מקום בו ייוצר צוואר-בקבוק. אח"כ המשכנו לדון במכונת OOO:

- התחילו עם Fetch/Decode רחוב שיתן לנו מס' הוראות בכל פעם והראינו צינור אופייני למעבדי אינטל בכך שיש שם IQ (Instruction Queue) שמחזק פקודות באורך משתנה.
- אחרי Decode עוזרים Allocation&Renaming - כל השלבים עד כה נעשים in-order.
- בכל מחזור בוחרים מבין כל הפקודות שנמצאות ב-RS את הפקודות שאפשר לשלוח לביצוע (זהו החלק שנעשה Out Of Order).
- אח"כ יש את שלב ה-Retire שגם הוא in-order.

דיברנו על כמה סכנות שונות להתמודד עם טעות בחיזוי:

- לחכות שהקפיצה תגיע ל-Retire, בשלב זה כל מי שאחריה בוודאי מופיע בתכנית המקורית אחראי הקפיצה ולכן עושים פלאש לכל הצינור.
- רוצים לעשות את זה יותר מהר ובפרט לא לחכות לסוף הטיפול בקפיצה, אך ראינו עד שני סכנות - אחת הרואعشות רק לחלק שהוא עדין In-Order ואיפה שזה מעורבב לא לרוקן - מביאים פקודות מהמסלול הנוכחי עד שהקפיצה מגיעה לפירשה, ואז עושים ריקון לכל החלק המאוחר יותר של המכונה והפקודות חדשות יכולות להמשך.
- בדרך השלישי והמתוחכמת ביותר אומרת שנעשה מיד ריקון לכל הפקודות, גם אלה הצעירות יותר מהקפיצה, אלא שזו צריכה דרך לשוב את טבלת המיפוי מהרגיסטרים הפיזיים לארQUITקטוניים שהתכלכה ע"י הפקודות הלא נוכנות אחראי הקפיצה, לשם כך שמרנו checkpoints - חזרנו אל האחזרנה ושבורנו את המיפוי שהוא לפניה ע"י Re-Renaming.

אחריו זה דיברנו על ביצוע שלא על-פי הסדר (Out Of Order) של פקודות זיכרון ואמנו שה-RS לא יודע לסדר פקודות של זיכרון אלא שכל פעם שהוא רואה פקודות load/store הוא רק מכחכת שהמקורות של הכתובות שלן יהיו מוכנים וכשה קורה אפשר לחשב את הכתובת ולכך הוא שולח את ה load/store החלט מה-AGU ומפסיק להיות מעורב - מי שמעורב במקומו זה ה-MOB.

אמנו שאפשר לנתח את התלוויות של load כלשהו רק כאשר כל הכתובות של כל ה-store לפניה כבר ידועות, כי כשה לא כך יכול להיות store-write כותב אותה כתובות כמו ה-load ואז צריך לחכות. אמנו גם שכאשר ה-store רוצה לכתוב לכתובת של load אבל הערך לכתיבה שלו מוכן גם צריך לחכות - לכן load יכול להיות חסום על address/data.

דיברנו על כך ש-store מפוץ לשתי מיקרו-פקודות, אחת-Store-Address (STA) והשנייה-Store-Data (STD). עוסים זאת כדי לא לצורך תלות מיותרת בין המקורות שצרכיהם את הכתובות ללא שצרכיהם את הנתון נשמר - וכן מספיק שנדע את הכתובת בלבד כדי לגלו את התלוויות של loads אחירות בפקודה זו (טוב בשביב ביצוע של loads לא עפ"י הסדר כי load תלוי באך store יכולת להתבצע).

דיברנו על load-to-store forwarding: ברגע ש-load תלוי ב-store וידוע שהכתובת של שתיהן זהה ה-load לא חייב לחכות עד שהມידע ייכתב לפחות כדי לחתול להתבצע אלא ה-load יכול לקבל אותו ע"י קידום (forwarding).

אמרנו גם שאפשר לשים חזאי שמנחש את התלותות בין גישות זיכרון לפני שהן מוגנות כך שלעתים לא חייבים לעצור load כלשהו ואפשר לקרוא את המידע מה-Data Cache גם לפני שההנובת של פקודה store אחרת ידועה. דיברנו על זה שהמתמון במכונת OOO צריך להיות non-blocking למשל cache-load load->cache Miss אז לא עוצרים את הביצוע אלאאפשרים עד פקודות load לגשת אליו עד שנגמרם לנו ה-fills buffers (חוצים שמקצים לכל load שמתחין למידע מהמתמון).

השיעור של היום מביא דוג' לمعالג של Intel, מופיע בו הרבה מידע שנאוסף מכל מיני מצגות/מדריכי-אופטימיזציות שונות.

### רקע לפיתוח מעבדים באינטל

## Tick/Tock Development Model

| 65nm                                                                        | 45nm                                                                         | 32nm                                                                        | 22nm                                                                     | 14nm                                                                             |
|-----------------------------------------------------------------------------|------------------------------------------------------------------------------|-----------------------------------------------------------------------------|--------------------------------------------------------------------------|----------------------------------------------------------------------------------|
| Merom                                                                       | Penryn                                                                       | Nehalem                                                                     | Westmere                                                                 | Sandy Bridge                                                                     |
| Intel® Core2™ Duo<br>New Intel® uarch<br>(Merom)                            | Intel® Core2™ Duo<br>Intel® uarch<br>(Merom)                                 | 1 <sup>st</sup> Generation<br>Intel® Core™<br>New Intel® uarch<br>(Nehalem) | 1 <sup>st</sup> Generation<br>Intel® Core™<br>Intel® uarch<br>(Westmere) | 2 <sup>nd</sup> Generation<br>Intel® Core™<br>New Intel® uarch<br>(Sandy Bridge) |
| <b>TOCK</b>                                                                 | <b>TICK</b>                                                                  | <b>TOCK</b>                                                                 | <b>TICK</b>                                                              | <b>TOCK</b>                                                                      |
| 2006                                                                        | 2007                                                                         | 2008                                                                        | 2010                                                                     | 2011                                                                             |
| Haswell                                                                     | Ivy Bridge                                                                   |                                                                             |                                                                          |                                                                                  |
| 5 <sup>th</sup> Generation<br>Intel® Core™<br>Intel® uarch<br>(Haswell)     | 4 <sup>th</sup> Generation<br>Intel® Core™<br>Intel® uarch<br>(Sandy Bridge) |                                                                             |                                                                          |                                                                                  |
| <b>TOCK</b>                                                                 | <b>TICK</b>                                                                  |                                                                             |                                                                          |                                                                                  |
| 2013                                                                        | 2012                                                                         |                                                                             |                                                                          |                                                                                  |
| Broadwell                                                                   |                                                                              |                                                                             |                                                                          |                                                                                  |
| 6 <sup>th</sup> Generation<br>Intel® Core™<br>New Intel® uarch<br>(Skylake) |                                                                              |                                                                             |                                                                          |                                                                                  |
| <b>TOCK</b>                                                                 |                                                                              |                                                                             |                                                                          |                                                                                  |
| 2014                                                                        |                                                                              |                                                                             |                                                                          |                                                                                  |
| Skylake                                                                     |                                                                              |                                                                             |                                                                          |                                                                                  |
| 2015                                                                        |                                                                              |                                                                             |                                                                          |                                                                                  |

איןTEL מפתחת מעבדים במודל-עבודה שנקרא Tick/Tock: בכל שנה מוצאים מעבד חדש - שנה אחת זהו מודל חדש לגמרי (Tick) ו שנה אחריה צהו שיפור של אותו מודל עם טרנזיסטורים חדשים (Tock). כל שנתיים יצא Process Technology חדש (זה נכון בכל התעשייה, לא רק באינTEL), כלומר כל שנתיים בקרוב מפתחים דור חדש של טרנזיסטורים שהם קטנים יותר (נמדד בגודל של השער - gauge) מאשר הדור הקודם, מהירים יותר ופועלים בהספק נמוך יותר. זה הכוח העיקרי שמניע את תעשיית המוליכים למחצה והואיצר התקדמות מודרנית בתעשייה זאת בהשוואה למשל לתעשייה הרכבת - אם הייתה לנו מכונית חסכוונית פי 2 בכל שנתיים אז מכוניות כבר היו מאד חסכוניות עד היום. אז זה הניע עד לאחרונה את כל תעשיית השבבים והרעיון היה לעשות דבר זהה: כל פעם שיוצא דור חדש של שבבים עושים באינTEL תהליך שנקרא Shrink, לוקחים את הדור הקודם (העדכני ביזטר) ופשוט מעבירים אותו לעבוד עם הטרנזיסטורים החדש. בשלב זה רוב הרוח הוא מהטרנזיסטורים החדש ולא משיפורים בארכ'/מיקו-ארכ'. כמובן שהלקחות רואים שיפור ממשוני גם במקרה זה אבל הוא לא מגיע משיפורים ארכ' אלא משיפורים בתשתיית, בטרנזיסטורים עצם. גם חברות אחרות עובדות בצורה דומה ומיצלת את שני הדברים אלה - שיפורים ארכ' ושיפורים בטכני התהליך.

בקשר באפור רואים את המעבדים החדש (מיקו-ארכ' חדש) ובכחול את ה-Shrink - המעבר של אותו מעבד לטרנזיסטורים החדש.

מרכז הפיתוח בחיפה היה מרכז צדי ייחודי ובשנת 2000 בערך נתנו לו לפתח מעבד שהיה מוקדש לשוק המעבדים הנידיים, שהיה שוק לא כ"כ חשוב בשנת 2000. פיתחו שם את המעבד "בניאס", שמיועד למעבדים נידיים, ואז היה את הבום הגדול של מחשבים נידיים ותזואה גדולה של השוק אליהם - זה כמובן קידם מאוד את מרכז הפיתוח בחיפה שבבקבוקה זו הפק למרכז פיתוח מרכזי.

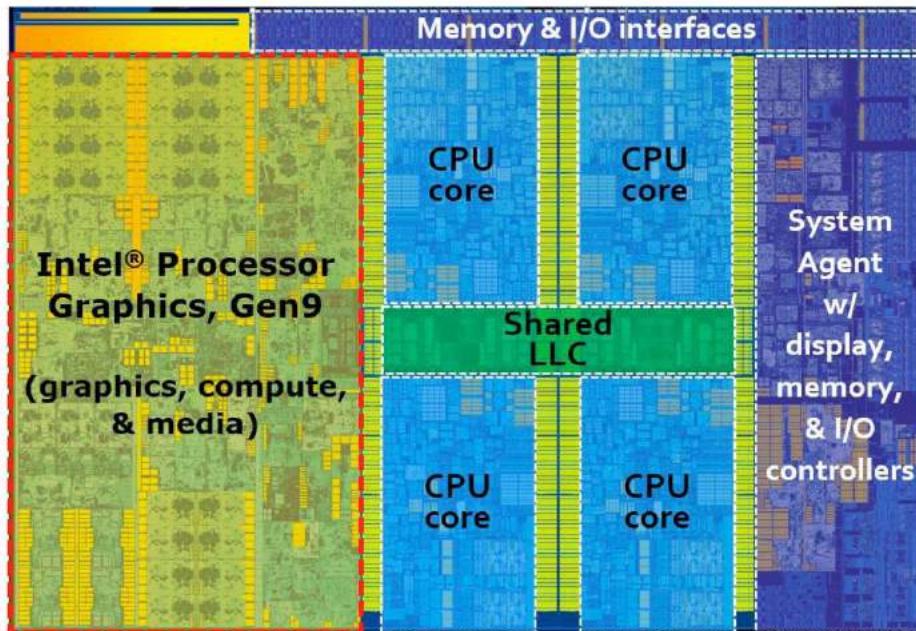
כעבור כמה שנים, בשנת 2006, הגיעו מהריה המעבד Merom שגם הוא היה מיועד למעבדים נידיים - אמנם בשנים אלה עדין מכיוון יותר מעבדים שימושיים למחשבים נייחים אף גם בנייחים וגם בשרתים התחללו לבנות קופסאות קטנות יותר כי ההספק התחליל לציבור חשיבות גדולה - עד אז הרספק היה פחות חשוב וקו המוצרים הקודם, פנטום 4, עבר בהספק מאד גובה. אלא שבדיקות בשנים אלה התברר שההספק הגבוה לא מתאים עבור מחשבים נייחים/שרתים, כי ברגע שיש הספק גבוהה המעבד מפיז הרבה חום, קשה לクリר אותו (למשל בטאבלט/פלאפון אין מואורר בכלל, ובנוייד יש מאורר שלישי), ואז המעבד צריך לדודת בתדר וכן בביטויים - אז הספק משפיע על הביצועים מואורר והניזוק מושפע ממעבדים הנידיים, הביצועים שלו היו פחות טובים בגלל מגבלות קו המוצרים של פנטום 4, שהייתה מוגבל לשרתים ולמעבדים הנידיים, האמור מושפע על הביצועיםagal המספק שהגיעו לגבול. אז אינטל הבינו שהמעבד Merom הוא טוב לא רק עבור נידיים אלא גם עבור נייחים ושרתים ובעצם המעבד הזה, שהיה מוגבל מריאש רק לנידיים, הפק למעבד שימוש את אינטל בכל הפלטפורמות שלה - ז"א הפק להיות המעבד של אינטל, מה שכמובן הקפיץ מאד את מרכז הפיתוח בחיפה, שניה מאותו רגע אחד משנה מרכז הפיתוח העיקריים (השני ב-חוסון, Oregon, ארחה"ב) שככל פעם מתחלפים במאי אחראי על הוצאה הדור הבא. זה עובד טוב כי תהליך הפיתוח של מעבד לוח כ-4 שנים ועם שני מרכזים פיתוח אפשר להוציא דור חדש כל שנתיים - אז הדורות החדשניים של המעבדים הם אלה שצובעים באפור והם עושים שינויים מיקרו-ארק' משמעותיים בהשוואה לקודמים למשל בנושאים של OOO, IPC ושאר הדברים שלמדנו עליהם בקורס. בדורות הביניים מה שurosים בעיקר זה לקחת את המעבד הקיים ולהעביר אותו לטרנזיסטוריים החדשניים. המעבד החדש ביותר שהוא נקרא Skylake והוא מפותח ע"י מרכז הפיתוח בחיפה.

ההפרשים בין הטכנולוגיות של התהיליך הם די לינאריים - היחס בין כל גודל לזה שלפניו צריך להיות בערך 0.7 (חוק Moore). עד איזשהו שלב התהיליך היה באמת אותו והשיפורים בטכני התהיליך קרו באופן טבעי - באיזשהו שלב נהיה יותר קשה לשפר את הטרנזיסטור ואז היוזם צריכים למצוא טריקים אחרים כדי להצליח לשמור את המומנטום האלה, למשל שימור בטרנזיסטורים תלת-מדדים ועוד. היום התהיליך כבר לא משתפר כל שנתיים אלא כל שנתיים וחצי וזה אכן סיבה לדאגה בתעשייה זו - ברגע שশמוכרים לאנשים מושצר שהוא יותר טוב מאשר מוכנים לשלם כסף ולקנות חדש, כשהשיפור יותר קטן לא כ"כ - רואים את זה גם בפלאפונים למורות שם יש גם אלמנטים של אופנה, בלאי וכו'. בKİצ'ור כל תעשייה צריכה למצוא מה גורם לאנשים להחליף מוצר ובתעשיית המעבדים זה הולך ונראה יותר קשה.

מי מפתח טרנזיסטוריים חדשים? אינטל, סמסונג, TSMC (בטאיוואן) - וזה! המכוניות שמייצרות את הטרנס' הולכות ונחיות יותר יקרים וטכני המחבר גם כן נהיית יותר יקרה: ככל שהטרנזיסטור קטן יותר הוא יותר רגש לאבק, הבניין צריך לשבת על קפיצים כדי למנוע רעידות ועוד. אז אם בעבר (Fabrication Center) FAB, מפעל ייצור FAB עליה כמילייארד דולר, היום הוא עולה חמישה מיליארד דולר - מאוד יקר, וההמודון של חברות שמחזיקות FAB הולך ומצטמצם והיום יש מס' קטן של חברות שעשו זאת (למרות שברגע ש-FAB נהייה מישן אפשר למוכר אותו לחברת אחרת שתיצור בו צ'יפים למכוונות חלשות יותר כמו מכוניות כביסה וכו').

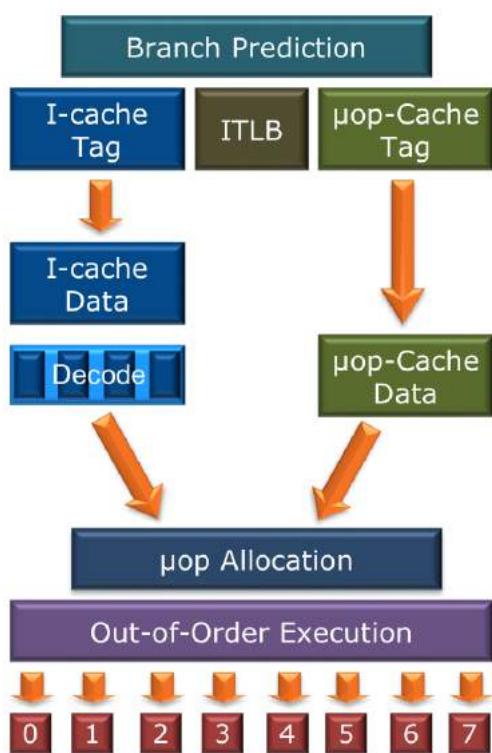
הבהרה: בכל עת לאינטל יש מעבד עיקרי יחיד והוא צריך לשמש את כל הוויריאנטים/סדרות שלו. למשל הסדרות 7, 15, 13, כל אלה וריאנטים על אותו מעבד. איך מייצרים את הוויריאנטים השונים? כמשמעותם יוצאים מה-FAB אין להם את אותם מאפיינים, יש כאלה חזקים יותר ופחות וריש שונות בכוח שלהם גם כאשר יוצרו על אותה פיסת סיליקון. אז לוקחים את המעבדים שנוצרו באותו תהליך אבל עדין יכולים לעמוד בתדרים שונים וושים בדיקה לכל אחד מהו באמת התדר שלו וזה מוביל לפוי מקטלגים מעבד כ-15, 13, 7 וכו'. בנוסף יש דברים על הצ'יפ שעושים להם disable כר שב-3 לא יהיה אותו וב-5 כן.

## 6th Generation Intel CoreTM – Skylake



המעבד מיוצר בטכ' תהליך של nm 14, יש בו ארבע ליביות, כרטיס-גרפי מובנה שמהווה חלק גדול מהsilicon, רואים גם את ה-LLC (Last Level Cache) המשותף לליביות ויכול לשמש גם את הכרטיס הגרפי וכן רואים את ה-System Agent שמשרת את כל מה שמתחבר אל הליביות.

## Core at a Glance



### Next generation branch prediction

- Improves performance *and* saves wasted work

### Improved front-end

- Initiate TLB and cache misses speculatively
- Handle cache misses in parallel to hide latency
- Leverages improved branch prediction

### Deeper buffers

- Extract more instruction parallelism
- More resources when running a single thread

### More execution units, shorter latencies

- Power down when not in use

### More load/store bandwidth

- Better prefetching, better cache line split latency & throughput, double L2 bandwidth
- New modes save power without losing performance

### No pipeline growth

- Same branch misprediction latency
- Same L1/L2 cache latency

## סקירה המיקרו-ארכ' של אינטל:

משמאל רואים את הציגור של המעבד, כМОן שהוא מעבד OOO, בגודל בכל דור של מעבדים הרבה דברים פשוט מוגדים - למשל ה-RS, ROB, BTB, 2nd-TLB ועוד - עוד מעט נראה טבלה שמתארת את הגדילה של כל אלה.

## Branch Prediction Unit

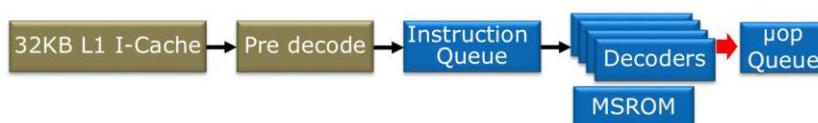
- **Predict branch targets**
  - Direct Calls and Jumps – target provided by a Target Array
  - Indirect Calls and Jumps – predicted either as having a fixed target or as having targets that vary based on execution path
  - Returns – predicted by a 16 entry Return Stack Buffer (RSB)

- **For conditional branches**
  - predict if taken or not
- **The BPU makes predictions for 32 bytes at a time**
  - Twice the width of the fetch engine
  - Enables taken branches to be predicted with no penalty

נתחיל מחזאי-הקייזות, נזכיר הוא חייב להיות מאד מדויק גם כי הוא קובע את הגודל האפקטיבי של ה-RS וכפי שראינו בפרק שעבר כל פעם שיש קפיצה או הסתברות שנעשה לפקוודה כלשהי Retire יורדת אחורי כל קפיצה וכן כדי להגדיל את ה-RS חייבם לשפר גם את ה-BTB.

כמו כן ה-BTB רץ בקצב יותר מהיר מאשר המכוונה - אם למשל אנחנו מבאים 16 בתים בכל מחזור שעון (כ-4 פקודות) אז החזאי עובד בקצב כפול - הוא עורך חייזוי ל-32 בתים וכתוכאה מזה החזאי יכול לראות את עתיד המשך הביצוע למשל הוא עושה חיפוש (lookup) ב-instruction cache, מגלה חייזוי שקפיצה נלקחת לפני הזמן ומספיק לחפש ולעשות ל-fetch לפקודות המתאימות - אך נחסר לנו ה-penalty שנובע מביצוע חייזוי של taken כאשר החיזוי נעשה אחורי שלב ה-fetch.

## Instruction Fetch and Pre-decode



- **32KB 8-way I-Cache**
  - Fetches aligned 16 bytes per cycle
    - Typical programs average ~4 bytes per instruction
  - A misaligned target into the line or a taken branch out of the line reduces the effective number of instruction bytes fetched
    - In typical integer code: a taken branches every ~10 instructions, which translates into a partial fetch every 3~4 cycles

וז דיברנו על החזאי ואחריו עוברים לעשות instruction cache של Fetch/Decode - נתחיל ב-Fetch/Decode של פקודות - בוגודל 32KB ובכל מחזור קוראים ממנה 16 בתים - למעשה בפועל נשארות מכך כשלוש פקודות כי עושים תהליך שנקרא pre-Decode: ב-8x64 הפקודות הן באורך משתנה. כאשר קוראים 16 בתים של פקודות ראשית למצוא את האורך של כל הפקודות - איפה כל פקוודה מתחילה, למשל נניח שבבית השלישי מתחילה פקוודה חדשה, וכך' בבייה

7 ואז בבית 12. הבעה היא שכשמתחילה לפענה פקודה כלשהי בתחילת השורה עד שלא נגלה איפה היא מסתנית לא נגלה איפה מתחילה הבהא - זה תהליך סדרתי מיסודו, אלא שאי אפשר להרשות לעצמו לעשות את זה באופן סדרתי וצריך לפענה את כל הפוקודות במקביל - لكن בצורה ספקולטיבית מניחים עבור כל בית שבו מתחילה פקודה זהה וזה עוזר לנחש את האורך שלה, אם מתברר שפקודה באמת מתחילה בבית זה אז הפוקודה שם מוכנה וכן כמו פוקודות יכולות להיות הן מוכנות במקביל - וזה תשלום יקר בחומרה. אז ה-Pre-Decoder מחלק את השורה לפוקודות.

לאחר פענוח הפוקודות אפשר לכתוב אותן לטור ה-Instruction Queue, זה תור שבכל כניסה מכיל פקודה ובכל מחזור שעון אפשר לכתוב לו תוכו עד שיש פוקודות מתוכו אותן 16 בתים, כאמור בדרכ' יכתבו שלוש אבל אפשר שגם יפוענו 16 פוקודות או אף פוקודות (יתכונו פוקודות ריקות). כמו כן יכול להיות שהייתה לי פוקודת קפיצה שקפיצה כך שהבאונו 16 בתים שמתחללים לקרווא רק מהוסף שלהם (תמיד מבאים 16 בתים מיזוחרים בזיכרון) ואז קראננו הרבה בתים למרות שלא הוצאנו מהם פוקודות ובפרט עם הפוקודה שקפצנו אליה לא מסתנית באוטם 16 בתים אז נקרא פוקודה שלמה (אחד לפחות) רק במחזור הבא. אם יש יותר משש פוקודות ב-16 בתים אז ייקח כמה מוחזרי שעון לכתוב אותן ל-Instruction Queue אבל זה מצב טוב - כל עוד מצלחים לכתוב לפחות ארבע פוקודות אנחנו במצב מעולה. בשביל למנוע את המצב שאחורי קפיצה עושים Fetch למס' פוקודות קטן מדי כדי שה-Jump Target יהיה לכתובת שמיישרת ל-16 בתים בזיכרון וקומפונילרים אכן מנסים לעשות זאת.

## Micro Operations (Uops)

- **Each X86 inst. is broken into one or more RISC μ-ops**

- Each μop is (relatively) simple

- **Simple instructions translate to a few μops**

- Typical μop count (it is not necessarily cycle count!)

|                            |                           |
|----------------------------|---------------------------|
| Reg-Reg ALU/Mov inst:      | 1 μop                     |
| Mem-Reg Mov (load)         | 1 μop                     |
| Mem-Reg ALU (load + op)    | 2 μops                    |
| Reg-Mem Mov (store)        | 2 μops (st addr, st data) |
| Reg-Mem ALU (ld + op + st) | 4 μops                    |

- **4 Decoders, which decode instructions into μops**

- 1<sup>st</sup> decoder can decode all instructions of up to 4 μops
- The other 3 decoders handle common single μop instructions

- **Instructions with >4 μops generate μops from the MSROM**

- **Up to 4 μops/cycle are delivered into the μop Queue**

- Buffers 56 μops (28 μops / thread when running 2 threads)
- Decouples the front end and the out-of order engine
- Helps hiding bubbles introduced between the various sources of μops in the front end

לאחר שכותבים instruction queue (עד שיש פוקודות בכל מחזור שעון) בשלב ה-Pre-Decode שלוחים את הפוקודות ל-Decoders: התפקיד שלהם הוא "לפרק" את הפוקודות הרגילות למיקרו-פקודות שנקבעו Ops-u - הסיבה לכך היא שרוצים מעבד שהוא "RISC-like": כ-Sh-86 נולד היו לו הרבה פוקודות מסוימות וזה דרש חומרה מורכבת מדי שהכhibaיה מאד על הארכ' ועל המיקרו-ארכ'. הרבה יותר קל, ויותר נכון, לבנות חומרה שיזדעת לשעות רק פוקודות פשוטות, לתת לפוקודות מסוימות להופיע ב-ISA, ככלומר בשפה שמנוגרת ע"י הארכ', ושה-Decoders יפרקו פוקודות מסוימות לפוקודות פשוטות יותר. דוג'ה:

- **Reg-Reg ALU/MOV: פוקודה שקוראת את הערכים משני רגיסטרים, לחברת אותם ושמוררת את התוצאה ברגיסטר שלישי תתרגם ל-kon בוודدت. כנ"ל עבור פוקודה שמצויה ערך מרגיסטר אחד לאחר.**

- **Mem-Reg MOV: גם פוקודת load שקוראת ערך מהזיכרון ושם אותו ברגיסטר תוממש ע"י kon בסachat.**

- **Mem-Reg ALU: פוקודה שקוראת ערך מרגיסטר וערך אחר מהזיכרון, עושה פעולה אריתמטית כלשהי עליהם וכותבת את התוצאה לרגיסטר - תפורק ל-kon שעושה את ה-load, קבלת הערך מהזיכרון, ואז kon שיודיע ערך לשעות את החיבור.**

- **פקודת store כפי שהסבירנו בנושא של Out-Of-Order Execution מתחולקת לשתי-מיקרו פוקודות, A-STO, STD, כי לא רוצים לצור תלות בין חישוב הכתובה וחישוב ה-data.**

از כאמור תפקיד המפענחים, Decoders, הוא לפענה את הפקודות ולהפוך אותן ל-sops. במעבד החדש יש ארבעה מפענחים ואלמנט-חומרה נוספת שנקרא MSROM - התפקיד שלו זה לפענה פקודות שמתפרקות יותר ROI (Return Over Investment) tradeoff (רווח) בין השקעה (מחיר) לבין תועלת (רווח): רוב הפקודות מתפענחות ל-ms' מרובה של sops (עד ארבע) ושלושת האחרים Decoders כך שرك הראשון יודע לפענה פקודות שמתפרקות ל-ms' מרובה של sops (עד ארבע) ושלושת האחרים יודעים לפענה רק פקודות שמתפענחות ל-ms' בלבד - וכך כל פעם שיש פקודה שהיא מסובכת צריך להביא אותה למפענח הראשון שرك הוא יודע לפענה אותה. בנוסף יש פקודות שמתפרקות יותר ארבע מיקרו-פקודות, אולי יש פקודות איטרטיביות כמו repeat move string שודיעת לBLOCK בזיכרון ולהעביר אותו למקום אחר ומקבלות את גודל הבלוק כารוגמנט, ככלומר ms' sops עבורן משתנה. בשביל זה יש את ה- ROM והוא אחראי על יצירת מיקרו-פקודות עבור פקודה שדורשת יותר ארבע מיקרו-פקודות.

אז כאמור יש ארבעה מפענחים, הראשון יודע להוציא ארבע מיקרו-פקודות והאחרים אחד בלבד - וכך לכואורה מכל Decoders יכולים לצאת בכל מחזור שבעה sops. בפועל משיקולי עלות-תועלת יתרונם עד חמשה sops בסך הכל - למשל אם המפענח הראשון הוציא ארבע מיקרו-פקודות אז רק עוד מפענה אחד אחר יכול להוציא מיקרו-פקודה אחת. באופן כללי כמשמעותו נדרש כל הזמן לעשות את השיקול של האם באמת משתלם לתמוך בכל מה שאפשר או שעדייף להשקיע את הטרנסיסטורים במקום אחר כי ms' הטרנסיסטורים מוגבלים - בדר"כ עדיף לשימושם במקום שמרוחחים יותר.

## Macro-Fusion

- **The IQ sends up to 5 inst. / cycle to the decoders**
- **Merge two adjacent instructions into a single pop**
  - A macro-fused instruction executes with a single dispatch
    - Reduces latency and frees execution resources
  - Increased decode, rename and retire bandwidth
  - Power savings from representing more work in fewer bits
- **The 1<sup>st</sup> instruction modifies flags**
  - CMP, TEST, ADD, SUB, AND, INC, DEC
- **The 2<sup>nd</sup> instruction pair is a conditional branch**
- **These pairs are common in many types of applications**

דבר על Macro-Fusion - איחוד של שתי פקודות מכונה למיקרו-פקודה אחת: יש צמד פקודות הקשורות מאוד נפוץ של השוואה שמעדכנת דגלים כלשהם במעבד ואחריה קפיצה בהתאם לערך אחד הדגלים, למשל כמשווים שני רגיסטרים וועושים equal Jump,Jump if less than .. הפקודות הנ"ל מאוד נפוצות כי כל if שעושים בקוד מתרגם לצמד nn".lv. אז Macro-Fusion מŁוקח שתי פקודות אלה ומתרגם אותן למיקרו-פקודה אחת - בעצם הבינו שככל מהן כל כך פשוטה שאפשר לאחד אותן ל-ms' בלבד: שולחים אותן ל-Decoders כשתי פקודות ושם הן הופכות ל-ms' אחת - זו דוג' לرك שהמיקרו-ארכ' מקופה על ביצוע התכנית באופן שהארכ' לא צריכה להיות מודעת אליו כלל.

כפי שהסבירנו יש תורים/באים שמטרתם להחליק את הפס: מצד אחד מלאים כמוות לא-קבועה של פקודות לתורים ומצד שני רוצים שתיהיה לנו אפשרות לצורר תמיד מס' פקודות קבוע: באמצעות תורים יוצא שלעתים יכנסו לתורים קצת יותר ולעתים קצת פחות פקודות ומצד שני אפשר להוציא אותן מהטור בקצב קבוע.

## Stack Pointer Tracker

- **PUSH, POP, CALL, RET implicitly update ESP**
  - Add or sub an offset, which would require a dedicated `pop`
  - The Stack Pointer Tracker performs implicit ESP updates

|          |                  |                       |
|----------|------------------|-----------------------|
|          |                  | $\Delta = 0$          |
| PUSH EAX |                  | $\Delta = \Delta - 4$ |
|          | STORE [ESP], EAX | STORE [ESP-4], EAX    |
| PUSH EBX |                  | $\Delta = \Delta - 4$ |
|          | STORE [ESP], EBX | STORE [ESP-8], EBX    |
| INC ESP  | ESP = ADD ESP, 1 | Need to sync ESP !    |
|          |                  | $\Delta = 0$          |
|          |                  | ESP = ADD ESP, 1      |

- **Provides the following benefits**

- Improves decode BW: PUSH, POP and RET become single `pop` instructions
- Conserves rename, execution and retire resources
- Remove dependencies on ESP – can execute stack operations in parallel
- Saves power

במקביל לפירוק פקודה לכמה מיקרו-פקודות אנחנו תמיד מוחזקות את כמות הפקודות / `ops` – זה גם תורם לביצועים וגם חוסך הספק. שיטה שמאפשרת זאת נקראת Stack Pointer Tracker Push/Pop/Call/Return (מצביע המחסנית) שעובdot על המחסנית - `push`, `push/pop/call/return`. באופן דומה `call` מתרגמות לשתי פעולות: הקטנה/הגדלה של מצביע המחסנית ופעולה `load/store` כלומר אחרי שהגדלו את מצביע המחסנית קוראים או כתובים מראש המחסנית. כדי לעקוב במדויקות אחריו שינוויים ב-ESP מוחזקים משתנה בחומרה שנקרא `Delta` והוא שומר היסט מוקומי: בתחילת הריצה, כלומר לאחר אתחול הצינור,  $\Delta = 0$ . נניח שיש לנו פקודה `push`, אז היא תפוצל לשולש `ops` – אחת עשויה  $\Delta = 4$  – ועוד שני `ops` של `store data-to-store address`. כתע מעליים את ה-Op  $\Delta$  הראשו ובקומו מעדכנים את המשתנה הגלובלי `delta` להיות  $-4$  – ואז משנים את ה-`store address` כך שבמקום שהוא תשנה את הערך של ESP ואז תשמש בו היא עשויה `store` שירות לכתובות ESP-delta בלי לשנות את `store address` נניח שאחריה הגיעה עוד `push`, אז שוב מפעלים אותה, מקטינים `delta` כך שהוא הופך ל- $-8$  ושוב ה-`store` הופך להיות שוב לכתובות המפורשת במקומות להתחיל משינויו ערכו של ESP ואז שימוש בערך זה לכתיבת הוא כותב שירות ל-`ESP-delta`. אז כל עוד יש לנו `push/pop/call/return` רק משנים את הערך של המשתנה המקומי `delta` ומשתמשים בו בהתאם. בכך חסכנו כמה דברים:

- הורדנו את מס' המיקרו-פקודות אליה מתפענת כל פקודה וכן המפענחים פשוטים יותר יכולים להתמודד אתם – משפר את רוחב הפס במכונה.

- בנוסף מכיוון שלא צריך לשנות את ערכו של ESP אז לא שולחים את הפקודה `Renaming` או `l-ops` אלא ישר להנפיק את הכתיבה ל זיכרון – זה חוסך הספק בכך שזה מפנה את משאבי המכונה.
- יתרון שלישי הוא הסרת תלויות – במקור יש כאן כמה חישובים על ערכו ESP שתלויים אחד בשני כי כל חישוב צריך לחcout שהחישוב לפניו יסיים את הכתיבה – אחרי המעבר לשימוש בכתובות מפורשת באמצעות `delta`-`store` אין יותר תלות בין הכתיבה, כל החישובים מפורשים. לדוג' במקירו-ארק' שלא עושות את זה אפשר לשולח רק-Push אחד בכל פעם כי צריך לעדכן את ESP ואז פקודה הבאה צריכה את ערכו צריכה לחcout לה.

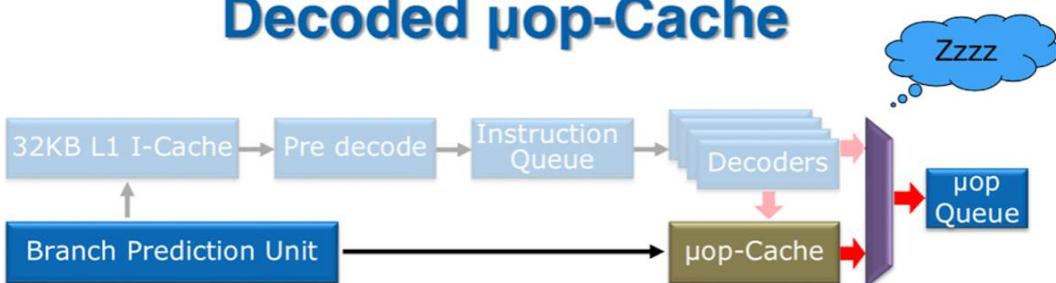
נשים לב שברגע שיש פקודה שרוצה את ערכו של ESP (למשל בשביל לשנות אותו במפורש) אנחנו משלמים מחיר - הערך שלו לא נכון! כזה קורה צרכים לנכון את ESP ולשים בו את הערך האמתי, אז מייצרים דוס שעשה ESP = ESP+delta, מופיעים את ה-delta וכעת ל-ESP יש את הערך האמתי שלו ואפשר להשתמש בו - אז מדי פעם משלמים את הדוס הנוסף הזה.

## Micro-Fusion

- **Fuse multiple pops from same instruction into a single pop**
  - Instruction which decode into a single micro-fused pop can be handled by all decoders
  - Improves instruction bandwidth delivered from decode to retirement and saves power
  - A micro-fused pop is dispatched multiple times in the OOO
    - As it would if it were not micro-fused
- **Micro-fused instructions**
  - Stores are comprised of 2 pops: store-address and store-data
    - Fused into a single pop
  - Load + op instruction
    - e.g., FADD DOUBLE PTR [RDI+RSI\*8]
  - Load + jump
    - e.g., JMP [RDI+200]

עוד דבר שעושים נקרא Micro-Fusion: איחוד שתי pops השויות לפקודה ל-**pop אחד** כדי שככל המפענים ידעו לפענה אותה, אחרת כל הפוקודות היו צרכות להישלח ל Decoder 0, זה לא יעיל. אחרי הפענה המיקרו-פקודה המאוחדת נשלחת לביצוע בכמה מקומות (כאיו לא נעשה איחוד).

## Decoded pop-Cache

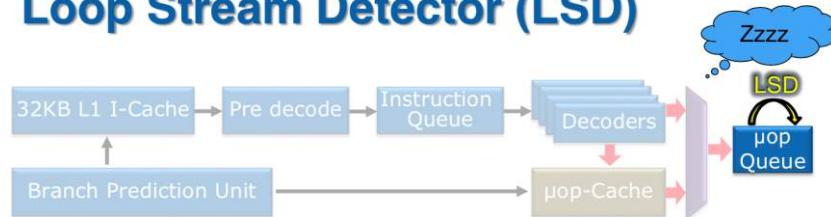


- **Caches the pops coming out of the decoders**
    - Up to 1.5K pops (32 sets × 8 ways × 6 pops/way)
    - Next time pops are taken from the pop Cache
    - ~80% hit rate for most applications
    - Included in the IC and iTLB, flushed on a context switch
  - **Higher Bandwidth and Lower Latency**
    - More cycles sustaining 4 instruction/cycle
      - In each cycle provide pops for instructions mapped to 32 bytes
      - Able to 'stitch' across taken branches in the control flow
- Save Power while Increasing Performance

דבר נוסף שנעשה הוא שככל הדוס ישלחו למטען של מיקרו-פקודות שנקרה pop ואז בשלב ה-Fetch Front-End יש לו hit ב-pop-Cache או ב-instruction Cache TLB (היזוי-קפיצה, וכן ה-)

שדייברנו עליו הולך לישון וה-cache dop מספק את המיקרו-פקודות במקום ה-Decoder. זה טוב גם כי מרוחחים הספק וגם ביצועים מבחינות זמן הפענוח שונחן. הערה: ה-cache dop הוא וירטואלי כי רוצים לחסוך את זמן/הספק התרגום - לכן צריך לעשות לו flush בכל החלטת - הקשר של תהליכי במעבד. הגודל שלו הוא 1.5k uops.

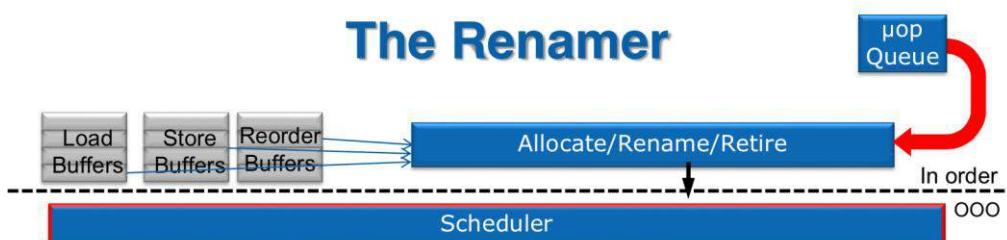
## Loop Stream Detector (LSD)



- **LSD detects small loops that fit in the μop queue**
  - The μop queue streams the loop, allowing the front-end to sleep
  - Until a branch miss-prediction inevitably ends it
- **Loops qualify for LSD replay if all following conditions are met**
  - Up to 28 μops, with ≤8 taken branches, ≤ 8 32-byte chunks
  - All μops are also resident in the μop-Cache
  - No CALL or RET
  - No mismatched stack operations (e.g., more PUSH than POP)

הדבר הבא שנראה הוא פיצ'ר מיקרו-ארQUITטוני שנקרא LSD: כשהם זמינים לוולה שהוא כל כך קטעה שהוא נכנסת כולה בתוך ה-queue dop או עוברים לביצוע הלולאה ישורות מתוך ה-queue dop עד שהוא נפסקת. מלבד שיפור ביצועים זה מאפשר לנו לחסוך את ההספק של ה-Front-End כולל משך זמן, קלומר ממש לכבות אותו - פיצ'ר מאד טוב שיחסם הספק ומשפר ביצועים.

מפה אנחנו עוברים לדבר על ה-Renamer:



- **Moves ≤4μops/cycle from the μop-queue to the OOO**
  - Renames architectural sources and destinations of the μops to micro-architectural sources and destinations
  - Allocates resources to the μops, e.g., load or store buffers
  - Binds the μop to an appropriate dispatch port
  - Up to 2 branches each cycle
- **Some μops are executed to completion during rename, effectively costing no execution bandwidth**
  - A subset of register-to-register MOV and FXCHG
  - Zero-Idioms
  - NOP

זכור ה-Renaming מתבצע in-order ועושים 4 ops בכל עת. כמו כן ה-renamer יודע לבצע חלק מה-ops לגמרי עצמו, בלי לשלוח אותו לביצוע, למשל את הפקודה MOV לא צריך לשלוח לביצוע: אם רואים שיש לנו mov r2<1z r5, אז נניח שהרגיסטר הפיזי שמחזיק את 2r זה pr5, ה-Renamer יעשה את הפקודה בלי לשלוח אותה לביצוע כך: יגדיר את 1z בפקודה הנ"ל להיות 5ck ובעצם על ידי שינוי זה הוא מעביר את הערך של 2r אל 1z כי מכשיו כל מי שירצה להשתמש בערך של 1z יוכל ישמש ב-5ck שמחזיק גם את הערך של 2r. הדבר הזה מאד יעיל - כל פקודות הMOV מתחבצות ע"י הטריק הפשטן הזה ולכן אין נשלחות לביצוע כלל. נבהיר את זה בדוג':

- נניח ש-1z הוא רגיסטר שבפועל מוצבע מ-2r ו-2r מוצבע מ-5ck.
- פקודה כלשהי מייצרת ערך ל-2r, למשל המס' 37 ולכן כתוב אותו ל-5ck.
- פקודה אחרת מבצעת  $r2 > r1$
- ואז פקודה שקוראת את 1z - אז ה-1z שהוא הגיע מ-2r, זה אותו 37. אז ע"י כך שנגיד לפקודה MOV לקחת את הערך של 5ck בשבייל 1z גם הפקודה שקוראת את 1z קיבל את אותו 37.

זה תרגיל טוב לבחינה!

## Dependency Breaking Idioms

- **Zero-Idiom – an instruction that zeroes a register**
  - Regardless of the input data, the output data is always zero
  - E.g.: XOR REG, REG and SUB REG, REG
  - No μop dependency on its sources
- **Zero-Idioms are detected and removed by the Renamer**
  - Do not consume execution resource, have zero exe latency
- **Zero-Idioms remove partial register dependencies**
  - Improve instruction parallelism
- **Ones-Idiom – an instruction that sets a register to "all 1s"**
  - Regardless of the input data the output is always "all 1s"
  - E.g., CMPEQ XMM1, XMM1;
    - No μop dependency on its sources, as with the zero idiom
    - Can execute as soon as it finds a free execution port
  - As opposed to Zero-Idiom, the Ones-Idiom μop must execute

עוד דבר שה-RAT יודע לעשות בעצמו נקרא **0-idiom** – זהה בעצם להחליפ חישובים מסוימים קבועו 0. דוג': XOR של ערך עם עצמו זה תמיד 0, אז אם למשל יש בתכנית כלשהי את רצף הפקודות:

$$(1) R1 = R1+R3$$

$$(2) R1 = R1\text{xor}R1$$

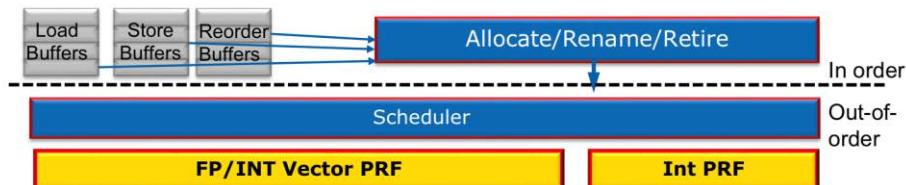
$$(3) R5 = R1+9$$

לכוארה הפקודה הראשונה תלולה בשנייה והשנייה בשלישית, אבל בפועל השנייה לא באמת צריכה להיות בראשונה כי התוצאה שלה בכל מקרה תהיה 0. זה נקרא **0-idiom**: בפועל ה-RS לא צריך לחכות עם פקודה 2 בגלל פקודה 1 איז ה-RAT בעצמו מייצר את הערך 0 (בדוג' - במקום האגרף הימני של פקודה 2) והוא גם לא באמת שולח את זה ל-Execution. כתוצאה מהשינוי הנ"ל גם הפקודה השלישית לא רואה את התלוות כי הערך של R1 בשבייל 0 והוא רואה את זה מיד ולכן כבר ב-Rename הפקודה השנייה והשלישית מוכנות לביצוע (והראשונה הייתה מוכנה בכל מקרה).

באופן דומה יש גם מהו שנקרא **1-idiom** – של קביעת רצף של 1 ברגיסטר כלשהו, גם בשבייל לשבור תלויות וכו'.

ה-0/1-idioms אומר שגם אם הקומפיילר השאיר את הפקודות האלה (ולא החליף את הקבועים בעצמו) עדין מעבד לא יצטרך לבצע אותן.

## Out-of-Order Cluster

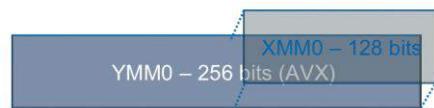


- **The Scheduler queues mops until all source operands are ready**
  - Schedules and dispatches ready mops to the available execution units in as close to a first in first out (FIFO) order as possible
- **Physical Register File (PRF)**
  - Instead of centralized Retirement Register File
    - Single copy of every data with no movement after calculation
  - Allows significant increase in buffer sizes
    - Dataflow window ~33% larger
- **Retirement**
  - Retires mops in order and handles faults and exceptions

הvisorון לבנייה של מדנו בו רגיסטרים מוצמדיםUrנים לפקודות ב-ROB הוא שכטיבת הערך חוזרת לו ROB ואז העברתו לפקודה ב-RS עולה הרבה אנרגיה - רצים לחסוך את זה ולכן משתמשים בקובץ רגיסטרים פיזי - אחרי שחישבונו את הערך כתובים אותו ב-ROB ויזהר מאוחר שימושו רוצה להשתמש בו -Cs קוראים אותו ישרות מה-ROB - אז הערך נכתב בשלב ה-Execute וכל מי שציריך קורא ישרות מאותו נקודה. יתרה מכך אין יותר רגיסטרים ארכ', Register File ואין יותר מצב שרגיסטר לפעמים הוא x ולפעמים 5 rk - בכל רגע יש קובץ-רגיסטרים שמחזיק את הרגיסטרים הארכ' - צצור ב-Retirement אמerno שמעתקים מפיזי לארכ' במקרה שלא היו חריגות וכו' - פה אין זאת ויש רק סט רגיסטרים אחד שמשמש להכל.

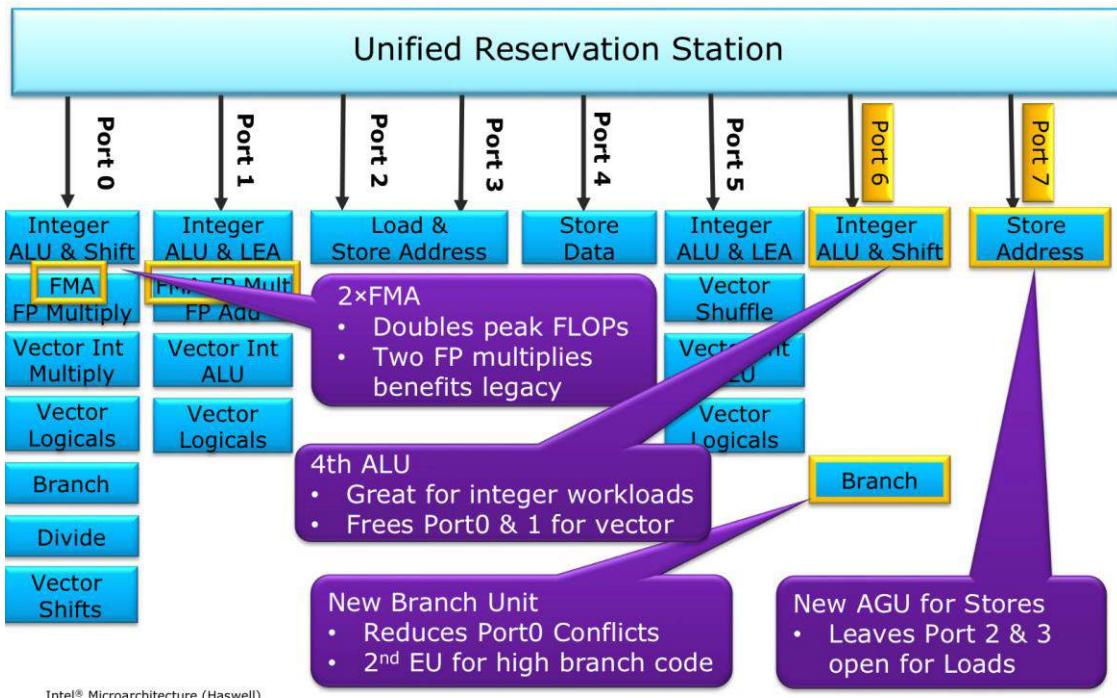
## Intel® Advanced Vector Extensions

- **Vectors are a natural data-type for many apps**
  - Extend SSE FP instruction set to 256 bits operand size
  - Extend all 16 XMM registers to 256bits



כל מיני פקודות ארכ' נוספו - למשל פקודות וקטוריות - פקודה צזו היא פקודה שיודעת בבת אחד לקחת שני וקטוריים, רצפי מידע בזיכרון, ולבצע פעולה כלשהי על האלמנטים שמרכיבים אותם - וקטור הוא 128 ביט בזיכרון שמורכב מרבעה איברים של 32 ביט, מוחזק ברגיסטר מיוחד לוקטור שנקרא (i)XMM, ולמשל יש פקודה שבמבחן שעון אחד מחברת כל רכיב המקביל לו בוקטור אחר ושםה את התוצאה ברגיסטר-וקטורי שלישי. השלב הבא הוא לעבור לוקטורים באורך 256 ביט - כך יש הרבה יותר Data שנitinן לעבד בצורה וקטוריית, למשל מקום לעשות add בין שמונה זוגות של מס' מהזיכרון היום אפשר לעשות את זה בשתי פקודות חיבור-וקטורי ובדור הבא יהיה אפשר לעשות את זה בפקודה אחת.

# Haswell Execution Units



במעבד יש יותר פורטים ויותר Execution Units וכל הזמן מוסיפים עוד כדי שהמכונה תהיה יותר חזקה. למשל מבחינת גישות ל>Zיכרון דיברנו על load-buffers ו-store-buffers שימושים בהם כדי לדאוג ל-memory ordering. כמו כן יש  $L_1$ ,  $L_1, L_2$  cache וה-  $L_2$  הוא non-blocking ומאפשר לעד עשר פקודות load לסייע  $load$   $Miss$ . במקביל ולהוציאו ביחד בקשה ל- $L_2$ . ניתן גם לבצע  $load/store$  forwarding וגם memory disambiguation  $load$ / $store$  forwarding. ייחד עם זאת לפקודה  $load$  שלא נכנסה ב- $fetch$  שמאפשר לנחש שפקודה  $load$  תלויה/לא-תלויה ב- $store$  כלשהו. ייחד עם זאת לפקודה  $load$  שלא נכנסה ב- $line$  אחת, ככלומר ב-16 בתים מיושרים של פקודות, אלא מתחילה מהאמצע של שורת בתים אחת וחורגת לבאה, או אפשר לעשות  $forwarding$  - הסיבה שמספרטים את זה היא לעודד מתכנתיו קומפונילרים לא ליצור loads שהם מופצלים בשתי שורות וכך לאפשר למעבד לבצע יותר טוב.

ב-Data Cache יש שני Prefetchers שמנסים לחזות גישות עתידיות לCACHE ולהביא אותן מבעוד-מועד. ישים שני פריפצ'רים: אחד מסתכל על גישות כתובות עולות, למשל ברגע שسورקים מערך הוא מבין את הטרנד ויוזם בקשנות הלאה לפני שהוא יתאפשר יישום. והשני נקרא stride-prefetcher והוא לומד על כל load בנפרד את הכתובות שלאו ומנסה להזמין טעינה מרצף כתובות בעל stride קבוע למשל אם פניו לכתובת+#100 אז שוב+#100 הוא לומד זאת ומיציר מראש את load הבא.

## Core Cache Size/Latency/Bandwidth

| Metric                       | Nehalem                                            | Sandy Bridge                                      | Haswell                                           |
|------------------------------|----------------------------------------------------|---------------------------------------------------|---------------------------------------------------|
| L1 Instruction Cache         | 32K, 4-way                                         | 32K, 8-way                                        | 32K, 8-way                                        |
| L1 Data Cache                | 32K, 8-way                                         | 32K, 8-way                                        | 32K, 8-way                                        |
| Fastest Load-to-use          | 4 cycles                                           | 4 cycles                                          | 4 cycles                                          |
| Load bandwidth               | 16 Bytes/cycle                                     | 32 Bytes/cycle (banked)                           | 64 Bytes/cycle                                    |
| Store bandwidth              | 16 Bytes/cycle                                     | 16 Bytes/cycle                                    | 32 Bytes/cycle                                    |
| L2 Unified Cache             | 256K, 8-way                                        | 256K, 8-way                                       | 256K, 8-way                                       |
| Fastest load-to-use          | 10 cycles                                          | 11 cycles                                         | 11 cycles                                         |
| Bandwidth to L1              | 32 Bytes/cycle                                     | 32 Bytes/cycle                                    | 64 Bytes/cycle                                    |
| L1 Instruction TLB           | 4K: 128, 4-way<br>2M/4M: 7/thread                  | 4K: 128, 4-way<br>2M/4M: 8/thread                 | 4K: 128, 4-way<br>2M/4M: 8/thread                 |
| L1 Data TLB                  | 4K: 64, 4-way<br>2M/4M: 32, 4-way<br>1G: fractured | 4K: 64, 4-way<br>2M/4M: 32, 4-way<br>1G: 4, 4-way | 4K: 64, 4-way<br>2M/4M: 32, 4-way<br>1G: 4, 4-way |
| L2 Unified TLB               | 4K: 512, 4-way                                     | 4K: 512, 4-way                                    | 4K+2M shared: 1024, 8-way                         |
| All caches use 64-byte lines |                                                    |                                                   |                                                   |

## Buffer Sizes

| Extract more parallelism in every generation |           |              |         |           |
|----------------------------------------------|-----------|--------------|---------|-----------|
|                                              | Nehalem   | Sandy Bridge | Haswell | Skylake   |
| Out-of-order Window                          | 128       | 168          | 192     | 224       |
| In-flight Loads                              | 48        | 64           | 72      | 72        |
| In-flight Stores                             | 32        | 36           | 42      | 56        |
| Scheduler Entries                            | 36        | 54           | 60      | 97        |
| Integer Register File                        | N/A       | 160          | 168     | 180       |
| FP Register File                             | N/A       | 144          | 168     | 168       |
| Allocation Queue                             | 28/thread | 28/thread    | 56      | 64/thread |

משמאלי רואים הרבה מגדילים נטען על המעבד Haswell (הדור הקודם): בפרט רואים שהגדילו את ה-STLB. אחד השיפורים הנפוצים בין דורות של מעבדים זה פשוט להגדיל את מבני הנתונים במעבד - מזה שיש לנו יותר טרנסיסטורים אפשר בקלות לשפר ביצועים, למשל כש-Apple נכנסו לשוק המעבדים היה להם יתרון - ברגע שאתה מפגר מאוחר כל יותר להדיבק את הפער כי כל הטכנולוגיה כבר זמינה - למשל לפני 4 שנים המעבד שלהם השתמש בטרנסיסטורים שהיו יישנים בשלושה דורות או כל שנה הם יכולים לעבור לטרנסיסטורים בטכנולוגיה משופרת בעודם באינטל צריך לחייב לטכנולוגיות חדשות ביותר שיוצאות כל שנתיים. באופן דומה עוד דבר ש-Apple עשו באופן עקיبي הוא להגדיל הרבה מאוד חוצצים ובני נטען במעבד. עכשו הם "ישראל-קו" ויש להם מעבד חדש שנקרא A9 והוא מעבד מאוד מוצלח ברוחב 6 (כלומר יכול לטפל בעד שש פקודות בכל מחזור). (אגב ל-APPLE אין FAB, הם משתמשים בשל סמסונג, אבל יש להם Design Centers של חומרה).

הערה: מי שמנהל את כל פעילות החומרה של Apple זה ג'וני סרווגי (Johny Srouji) שמדועו ישירות ל-[Tim Cook](#) (Tim Cook) שמדובר ישירות ל-[IBM](#), והוא עשה תפקיד שני פה בטכניון - עבד באינטל וב-IBM, ואז עבר ל-Apple וקדם - הוא נראה הישראלי הבכיר ביותר בתעשייה ההיברידית.

בטבלה מימין רשומים כל מיני גדלים של אפרים במעבד, למשל הגדול של ה-ROB ב-Skylake הוא 224 כניסות, הגדול של RS הוא בגודל 97 כניסות, הגדול של-h-Store Buffers הוא 56 ועוד. קבועים כמה להגדיל ע"י ניסוי וטעיה משיקולים של ROI ועמידה בזמן. אי אפשר להגדיל יותר מדי בלי שנוצרת להגדיל את מס' המוחזרים הדרושים לעבוד עם המבנה ולכן גם זה שיקול.

בשיקוף רואים את כל המעבד Haswell שמכיל את:

- Cache •

- Pre-Decoders •

- Instruction Queue •

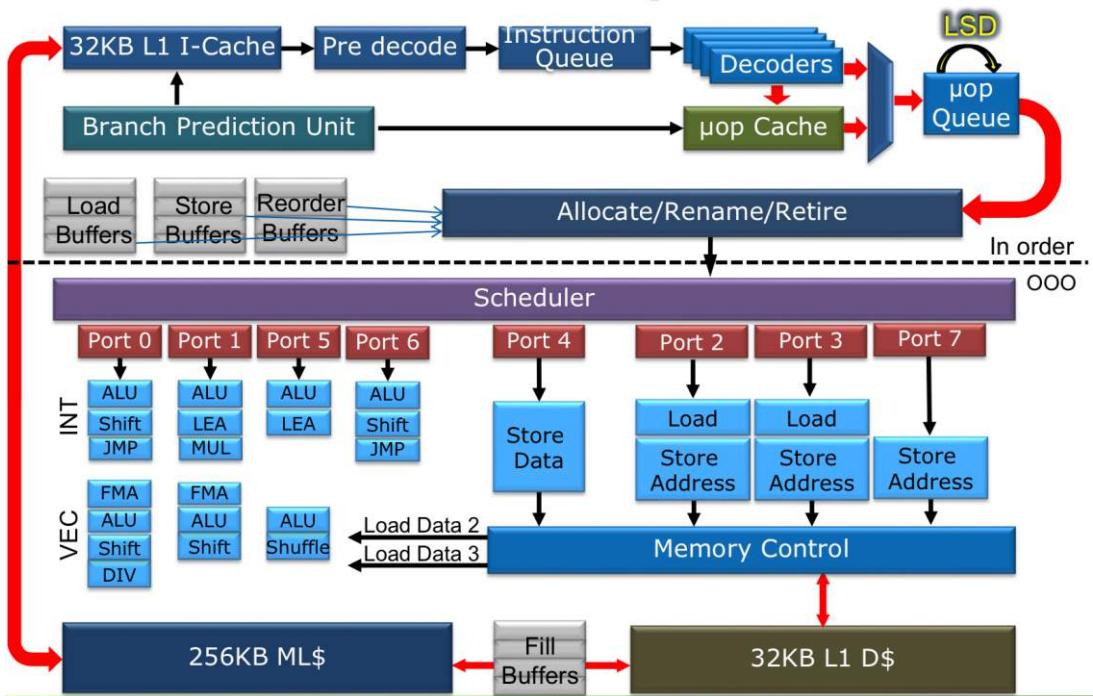
- Decoders •

- uOp-Cache •

- uOp-Queue •

- LSD
- Renamer
- RS-Store/load buffer Allocation, אחרי זה ל-ROB
- Scheduler
- Execution Ports/Units
- כל ה-*s*-units
- L2 Cache ו-L1 cache

## Haswell Core μArch



סיימנו.

## הרצאה מס' 12: הספק (Power)

היום נדבר על הספק (Power) והקשר שלו לביצועים (Performance). הנושא של הספק הוא מאוד חשוב, בשנים האחרונות במיוחד. תהיה במחן שאלת שעוסקת בנושא - גרא דוג', בסוף הרצאה.

עד כה בקורס למדנו איך לשפר את הביצועים של המעבד בלי להתחשב בשיקולי הספק: OOO, Caching, BTB .. בפועל מעבד מפיק חום כשהוא מרים אפליקציות ויש לפזר את החום הזה. כשהמעבד מתחמם הוא מתקרב לאבול ההספק שלו - גבול זה נקבע עפ"י הקופסה שהוא לננס אליו (למשל צלעות קירור ומעליות מאוורר) והוא בעצם מגביל את הביצועים המקוריים. אז היום נלמד כיצד מחשבים הספק, מהם המדרידים של הספק וכייז מתקנים שייפור כלשהו ביצועי המעבד כך שהשיפור יהיה יעיל מבחינת הספק (Power-Efficient).

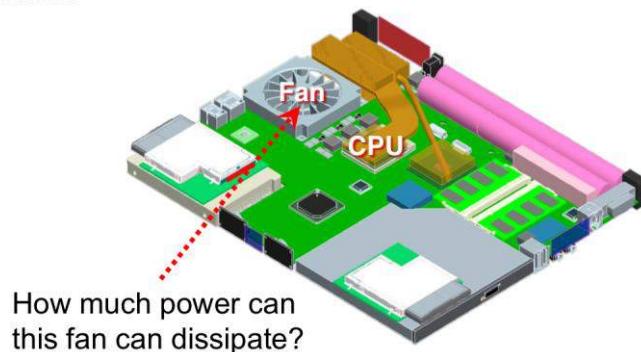
### נק' ההספק המקורי - Thermal Design Point (TDP) Power

- **TDP power**

- Maximum amount of power the thermal solution of the platform is required to dissipate
- Calculated as the rolling average of 5sec power of the highest real existing applications
- Not including Viruses
  - Some virus like application are ignored as well

- **TDP power impacts**

- Guaranteed frequency
- Affects form factor
- cooling solution cost
- acoustic noise



כל מעבד לננס לקופסה כלשהי בה החום שלו מctrבר וכן מתחפר - למשל יש קופסאות שונות ל- laptop, desktop, mobile. לכל קופסה יש כושר קירור שונה: למשל לקופסה של מחשב אישי יש יכולת קירור של 65-95 וואט (mobile). לכל קופסה יש כושר קירור שונה: לאלה מס' טיפוסיים עבור ההספק של מעבד במחשב שולחני בימינו. אז כאמור במעבד וב קופסה בה הוא נמצא) - אלה גם קובע את המעבד ומעליו יחידת קירור (הנוחות והמאוורר) שקובעת את גבול ההספק של המעבד בכל מחשב שולחני יש את המעבד ומעליו יחידת קירור (הנוחות והמאוורר) שקובעת את גבול ההספק של המעבד ונראה עד מעט שהוא גם קובע את גבול הביצועים. גבול ההספק המקורי נקרא ה- Thermal Design Point (TDP), למשל במחשב שולחני נפוץ ה-TDP הוא 65 וואט. במחשב נייד מדובר על TDP שנע בין 15 ל-28 וואט - בניידים הקופסה יודעת לקרוא פחות טוב וצריך להיכנס למוגבלות הספק יותר קטנה. טלפון סלולרי הוא בלי מאוורר בכלל וכושר הקירור שלו מתאים ל-TDP של 4 וואט.

הבהרה: כשמדובר על ההספק של המחשב מתכוונים להספק של ה-*czif* המרכזי, ה-(*SOC*) שמכיל את הליבות ואת קרטיס המסר המבנה וכו'.

מיד נסביר איך כושר הקירור משפייע על הביצועים שאפשר לקבל מה קופסה אבל חשוב להבין שכל Feature שימושיים למעבד דורש הספק נוסף - למשל בהרחבת מכונת OOO משלוש פקודות לארבעה הוסףנו עוד לוגיקה שנوتנת ביצועים וצורכת הספק - תמיד יש לתכנן את השיפור בהתאם למוגבלות ההספק של ה-SOC.

הגדירה: אפליקציית TDP, או אפליקציה חמה, היא אפליקציה שפעילה את כל מעגלי המעבד בצורה משמעותית, למשל אין בה יותר מדי Cache Misses ויציאות לזכרון כך שהמעבד נח בזמן הביצוע שלו, אלא היא מתחמת את המעבד בצורה אחידה ומשמעותית. למשל אפליקציית TDP מצליחה לעשות Dispatch לארבע פקודות בכל מחזור, מפעילה את יחידות הביצוע השונות במעבד ומגיעה ל-Power TDP, כלומר היא מפעילה את כל הטרנזיסטורים במעבד בצורה משמעותית. אפליקציית TDP היא תמיד המدد כנגדו נבדקים פתרונות קירור ושיקולי הספק. בפרט התדר שמסומן על-גביו המעבד מבטיח להיות מסוגל להריץ אפליקציית TDP.

דוגמה נגדית לאפליקציה חמה הן אפליקציות שימושísticas שוב ושוב באותו זמן טרנזיסטורים אר לא בccoli - זה נקרא אפליקציות-וירוס (Power-Virus).

התדר שמשפיע על כמה דברים, ביניהם Guaranteed Frequency מהירות התדרות שהיצרך מתחייב שהמעבד יוכל לעמוד בה - יש קשר בין תדר העבודה של המעבד ולכל מעבד יש אפשרות לשפר בכמה מהתדרים ותדריות שמתאימות להם - ככל שימושים את המתח אפשר (אם כי לא תמיד כדאי) להעלות את התדרות ובפרט לכל מעבד יש נק'  $V_{min}$  וכן  $V_{max}$  (לה מתאימה תדרות מקסימלית אפשרית). המתח המקסימלי של מעבד,  $V_{max}$ , נקבע ע"י TDP Power שלו ובאופן כללי המתח של המעבד יכול לנوع בגבול מסוים, למשל 7.6V. כל נקודה בגרף התדרות/ספק מואפיינת ע"י תדרות (ציר x) והספק המעבד בתדרות זו (ציר y). הגראף תלוי באפליקציה שמרוצת על המעבד - אפליקציית TDP שרצה בתדר כלשהו צורכת הספק מסוים ואם ניקח אפליקציה אחרת שאינה TDP אז יכול להיות שבאותו תדר היא תצרוך פחות ספק (למשל אם נריץ אפליקציה TDP מעבד ניד יצרוך 30W ועל אפליקציה אחרת ב-fmax יצרוך רק 20W). אנחנו תמיד נניח שימושים באפליקציה TDP בגרף התדר/ספק של המעבד. בהקשר זה אפליקציית TDP קובעת את ה guaranteed frequency, שהיא תדרות מסוימת על העוקמה שנמצאת באמצעות מבחינת הספק שהיא דרושת. זו גם התדרות שיצרך המעבד מתחייב עליה והוא מסמנת את מהירות המעבד כפי שנכתבת "על הקופסה". הבהרה: כשהחברה מכירה על מעבד מסוים כrz בתדרות כלשהי הכוונה היא שככל אפליקציה יכולה לרוץ בתדרות זו, בפרט אפליקציית TDP שימושת בכל מעלי המעבד באופן אינטנסיבי.

## ספק במעבד

הספק שמתאפשר ממעבד על עומס/תכנית כלשהי ברגע נתון נקרא "ספק דינמי" (לעתים נקרא גם הספק activity): בדרך כלל שמריצים אפליקציה יש תקופה בה היא צריכה הספק גבוהה, כי היא רצתה בוגרום פעילות (factor) גבוהה, אז עוברת לפעולה ב-idle (למשל כי מתבצעת יציאה לזכרון), אז חוזרת לפעולות גבוהה וכך הלאה. במחשבים אישיים (שולחניים/ניידים, אל לא שרתים) בדרך כלל 90%-95% הזמן האפליקציה שרצה במעבד תהיה ב-IDLE. ב-10% הנותרים, כשהיא רצתה בוגרום פעילות גבוהה, אז הטרנזיסטורים בתוך המעבד טוענים ופורקים כבילים - כל פעם שכבל נטען ל-1 / נפרק ל-0 זה עולה לנו בהספק ולמשל כשמריצים אפליקציה שעשויה פענוח של ארבע פקודות במחזור אז היא מפעילה יותר לוגיקה שעולה ביותר בהספק דינמי.

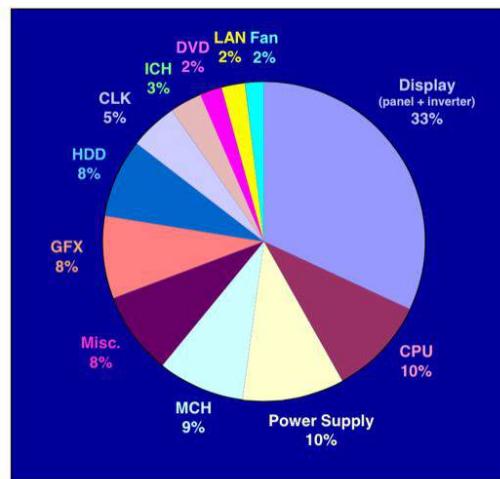
### • Average Power

- Average power = Total Energy / Total time
- Including low-activity and idle-time (~90% idle time for client)

### • Average Power determines

- Battery life – for mobile devices
  - Electricity bill
  - Air-condition bill
  - Air pollution
- } For servers

| Battery Life                             |       |
|------------------------------------------|-------|
| Continuous Web surfing over wire         |       |
| Lenovo ThinkPad T450s (extended battery) | 12    |
| Lenovo ThinkPad X250 (extended battery)  | 15    |
| Acer Aspire One Cloudbook (14-inch)      | 14:43 |



חוץ מההספק הדינמי, שנובע מהמעבד כשהוא תחת עומס כלשהו, במערכת מחשב יש גם מושג של "הספק ממוצע" בו לא מסתכלים על הביצועים המקוריים של הרצת עומס TDP במתוך/תדריות מסוימות אלא על ההספק הכלול שמתבצע בכל זמן פעולה המחשב. הספק ממוצע אינו משומש לצורך תכנון פיזור החום והוא לא מגביל את הביצועים אך הוא משפייע על חיי הסוללה (במחשבים ניידים) ובשרותים משפייע מאוד על חישוב החשמל של חוות-שרותים, בנוסף על הכוח שדרוש כדי לクリר את החווה, כמו זיהום-אוויר יגרם ממנה ועד. אז חשוב לזכור שההספק הממוצע לא מגביל את הביצועים אם כי הוא כן משפייע על דברים אחרים כמו חיי סוללה ומחריר החשמל. המעבד הוא לא הרכיב הכי משמעותי במחשב מבחינת הספק ממוצע, אלא המסר - ניתן לראות בשקף את החלוקה של ההספק הממוצע על הצלכניים השונים במחשב: המעבד צורך רק 10% מכל הספק המחשב בעוד המסר צריך שלישי, 33%, והוא כאמור ה张ן העיקרי של הספק ממוצע. אז שיקולים של הספק ממוצע הם חשובים, למשל כשמתכוונים את חיי הסוללה, אבל ההספק הדינמי של המעבד הוא ממשותי בקביעת התדריות, כמו ביצועים, performance, של המעבד, ועליו נדבר היום. אז דיברנו על שני דברים: הספק דינמי, שנמדד ע"י הרצת אפליקציה חמה במעבד וקשר מאוד לביצועים, והספק ממוצע שמשפייע בעיקר על חיי-סוללה/חישוב החשמל שם המעבד הוא גורם פחוות ממשותי. בשאר הרצאה נדבר רק על הספק דינמי.

cutת נעבר ללמידה את השיקולים של הספק דינמי ובפרט איך הוא יגביל לנו את הביצועים:

### • Dynamic power: $P_{dyn} = CV^2f$

- C – total electrical capacitance charged/discharged per cycle
  - The sum of all the capacities of all transistors and wires which are charged/discharged (toggle from 0→1 or from 1→0) per cycle
    - ▲ Transistors which maintain their value do not spend dynamic power
  - Application dependent
    - ▲ E.g., an application with no floating point calculation will not toggle the transistors in the floating point execution unit
    - Typically, a bigger CPU has a bigger  $C_{dyn}$
- V – voltage
- f – frequency: increasing f also requires increasing V ~linearly
  - $CV^2f \sim f^3 \Rightarrow X\% \text{ frequency costs } \sim 3X\% \text{ power}$

ההספק הדינמי הוא מתקובל ע"י הכפלת הקיבול, ריבוע המתח, והתדרות:  $f$

בכל רגע ההספק הדינמי משתנה כתלות בקיובל המעבד:  $C$

C - קיבול המעבד ברגע כלשהו: לכל מעבד יש קיבול דינמי, C, שנקבע כאמור עפ"י כמות הטרנזיסטורים אותו טוענים/פורקים, ככלומר עפ"י המعالגים הפעילים במעבד. אם הרחכנו את המכונה אז נראה עשוינו את זה ע"י הוספה לוגיקה, ככלומר במעבד החדש יש יותר טרנזיסטורים ובפרט יותר טעינה/פריקה של קבלים - הקיבול הדינמי Cdyn עולה. אז בהתאם למעבד ולאפליקציה שפעילה אותו (שתמיד נניח שהוא אפליקציית TDP, כאמור) מקבלים קיבול שונה.

### • Increasing f also requires increasing V (~linearly)

- Dynamic Power =  $aCV^2f = Kf^3 \Rightarrow X\% \text{ performance costs } \sim 3X\% \text{ power}$
- A power efficient feature – better than 1:3 performance : power
  - Otherwise it is better to just increase frequency (and voltage)

הקיבול, בשילוב עם נקודת המתח והתדר,קובעים את ההספק של המכונה, וכשאנחנו עובדים בקורסא כלשהו גודלה אומר לנו מה תקציב ההספק שיש לנו. כאמור הקיבול נקבע מתכונות של המעבד/אפליקציה. מיד נראה שנוסחת ההספק הדינמי  $P_{dyn} = C * V^2 * f$ , קובעת את התדר בו אפשר/כדי לזרוץ והוא הקשר העיקרי בין ביצועים (נקבעים ע"י התדר) להספק.

### • Leakage power

- Leakage of transistors under voltage, which is a function of
  - Z – total size of all transistors, V – voltage, t – temperature

חוץ מהספק אקטיבי יש למעבד "הספק נדייף", Leakage Power: כטרנזיסטור טוען יש פריקת מטען טבעי שתרחשת כל הזמן וצריך בכל מחזורי/בכל כמה מחזוריים לטוען אותו מחדש במאם שנפרק ממנו באופן טבעי - ההספק הנדייף הוא זה שמשמש כדי לבצע את הטעינה החוזרת הזאת. ככלומר גם כשקל לא משנה את ערכו יש בזבוז של הספק. ההספק המבוחני, Plkg, יחד עם ההספק הדינמי, Pdyn, קובעים את ההספק הכלול של המערכת:

$$P_{total} = P_{dyn} + Plkg$$

מה שמופיע על Plkg זה בעיקר אורך כל הטרנזיסטורים במעבד - בדר"כ גודל זה מסומן ב-Z, נמדד בマイקرونיהם שווה למכפלת אורך טרנזיסטור בודד במס' הטרנזיסטורים הכלול במעבד. אורך אופייני של Z הוא  $\sim 10\text{nm}$ . הערכה: גם לטמפרטורה גבוהה על Plkg אבל החשיבות של Z היא הרבה יותר גדולה.

## בחירה התדר בו יזרוץ המעבד: יעילות (ביצועים ביחס לאנרגיה).

עכשו שנחנו מבינים שיש קשר בין הספק ווצים לבין הבחירה בין יותר טוב אויר מתנהגת הנוסחה  $P_{dyn} = C * V^2 * f$  ואיך היא עארת לנו לבחור את תדריות המעבד - צצור בהקשר הספק הדורך שלנו לשפר את ביצועי המעבד היא להעלות את התדר. הבהרה: מעבד יכול לשנות את התדריות בה הוא עובד ולקבל הספק בהתאם. מתברר שככל שמעלים את התדריות נדרשים להעלות את המתח ביחס ישיר, ככלומר עבור העלאה של  $\Delta X\%$  בתדריות מתקבל העלאה של  $\Delta X\%$  בתמח (C קבוע כלשהו) ובפרט התדר בו רץ המעבד קבוע את מתח-העבודה ולהיפך. נשים לב שכשעלולים מנק' כלשהו על עקומת-hfreq, פוטומטר עלולים בתדריות (ציר x), אז מקבלים שיפור מסוים ביצועים, ובתמורה יש גם תוספת להספק (עליה בכיוון ציר y). ע"י פישוט של נוסחת ההספק, מתוור ההבנה שהמתח והתדריות עלולים ביחס ישיר, מתקבל שככל עלייה של אחוז בתדר מביאה לעלייה של שלושה אחוז בהספק, ככלומר עלייה של  $\Delta X\%$  בתמח עליה עולה לנו בתוספת של  $\Delta X\%$  בהספק.

תשולם של פי 3 בתמורה לשיפור ביצועים הוא גבוה, לכן ככל שנוריד את התדר נהייה במצב-עבודה שהוא יותר יעיל. הנקודה הכו עיליה (efficient) מבחינה זו שבה מתקבל היחס הטוב ביותר בין הביצועים להספק והוא הנק' בה התדריות מתאימה ל- $V_{min}$ : בנק' הזו קיבל את היחס הכי טוב בין הביצועים לבין ההספק כי ברגע

שמורידים את התדריות מתחת לתדריות של  $V_{min}$  אז המתח עדין נשאר  $V_{min}$ , כלומר זה המתח המינימלי להרצת המעבד, והורדת התדריות מעבר למתח זה לא חוסכת במתח/הספק כלל - כלומר גם אם נשיר בהורדת התדר לא ניתן לרדת מתחת ל- $V_{min}$  - אך בהורדת מעבר ל- $V_{min}$  מפסדים ביצועים ולא חוסכים הספק.

از בכל נק' בה התדריות גבוהה מהתדריות המתאימה ל- $V_{min}$  קיבל ביצועים טובים יותר, כי התדר עלה, אבל נשלם הרבה יותר בהספק, ולכן כשהמעבד רץ בנק'  $V_{min}$  הוא הכי עיל, מבחינה זו שהיחס בין הביצועים להספק הוא הכי טוב, ואם אנחנו מונינים ביעילות אז עדיף לנו לזרז בנק'  $V_{min}$  אבל אם רוצים לקבל ביצועים אז אפשר לשנות את התדר בכיוון  $V_{max}$  לומר בכיוון ההספק המקורי האפשרי.

### • **$V_{min}$ is the minimal operation voltage**

- Once at  $V_{min}$ , reducing frequency no longer reduces voltage
- At this point a feature is power efficient only if it is 1:1 performance : power

מתי כדאי לנו להעלות את התדריות ומתי לעבד ב- $V_{min}$  (قولמר ביעילות מקסימלית)? במקרה שרוצים לקבל הרבה ביצועים כדאי להעלות ובמקרה שמדובר אפליקציה שלא מספיק לנו מתי היא תסתיימן אז כדאי להרים אותה בנק' הכי עיל (قولמר ב- $V_{min}$ ). כמו כן במערכות מרובות מעבדים כשל המעבדים תמיד עוסקים כדאי לעבד עם כל מעבד ב- $V_{min}$  אחרת נבזבז הספק שיכלנו לנצל כדי להכניס עוד מעבדים למערכת (שגם הם יעבדו ב- $V_{min}$ )  
- נראה דוג' לך בסוף השיעור.

כפי שהזכרנו השימוש במעבד משתנה והוא תכונה ההספק: יש זמנים של פעילות גבוהה ויש זמנים של פעילות נמוכה, ותclf נראה תכונת המעבד בזרה צאת שוכן לחת *bursts*, פרצוי ביצועים/הספק. זה מתאים כי באמצעות בדר"כ שיכלוחצים על כפטור כלשהו רוצים תגובה מהירה.

از אנחנו מבינים שלביצועים יש מחיר בהספק ומעבר ל- $V_{min}$  המחיר הוא פי שלוש מהתמורה,قولמר פי שלוש מהחוז העלייה בתדר. נניח שאנחנו מתכוונים את הדור הבא של מעבד ושוקלים להוסיף לו פיצ'ר מסוים, שמכובן מכנים עד מעגלים, מגדיל את הקיבול הדינמי והסטטי ואת ההספק - ורוצים להחליט האם להכניס את השיפור למעבד או לא. כלל אכבע טוב הוא שרטוט שיריה גידול זהה של ביצועים ושל הספק - אם עומדים בזה זה או פיצ'ר טוב, אחרת הוא נראה לא כ"כ מוצלח. האמת היא שה-*cutoff*, قولמר הסף האמתי בין פיצ'ר כדאי לא-כדי הוא אכן פי שלוש אך פיצ'ר באמת מוצלח נעשה ביחס 1 ל-1 בין גידולה ביצועים לגידולה בהספק. למשל נניח שהמעבד הקיימ יכול לזרז בתדר מסוים - אם אני מוסיף עוד אחוז ביצועים וזה יעלה לי בערך 4% - עדיף היה להעלות את התדר (שגם מזה מדיפים להימנע) אז אין טעם להביא פיצ'ר כזה. גם עלות של 2% היא לא גורואה כמו העלתת התדר אבל נראה לא מספיק מושכת כדי להכניס את השינוי במעבד.

## ניהול הספק: מצב ביצועים (C-States) ומצבי שינוי (P-States)

ניהול הספק הוא האופן בו המעבד מתנהל לאור העומס שהוא צריך להריץ - בפרט בבחירה התדר בו המעבד רץ ובחלוקת המתח לרכיבים השונים במעבד. נשים לב שמערכת הפעלה היא זו שיודעת את הזרים של האפליקציה ובפרט האם ריא צריכה להסתiens מהר או שזה לא משנה כ"כ - אז מה' היא זו שתבקש מהמעבד להריץ מהר או תנסה אותו שזה לא חשוב, قولמר ניהול הספק משפייע על ביצועים ועל צריך ההספק כשהמתירה היא

לחת לאפליקציות חשובות/региשות יותר ביצועים טובים ולאפליקציות פחות-חשובות פחות ביצועים כדי לצרור פחות הספק. תוצאות שונות של ניהול הספק וקבועות באמצעות P-States ו-C-States.

- **Optimize power and energy consumption**

- High power when high performance is needed
- Low power at low activity or idle

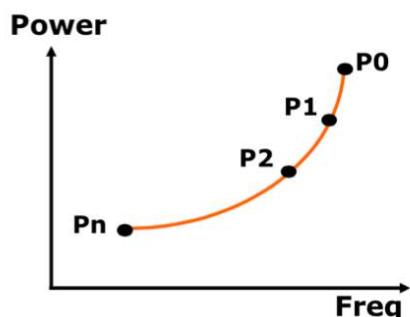
- **Enhanced Intel SpeedStep® Technology**

- Multi voltage/frequency operating points
- OS changes frequency to meet performance needs and minimize power
- Referred to as processor Performance states = P-States

- **Operation frequencies are called P-states = Performance states**

- P0 is the highest frequency
- P1,2,3... are lower frequencies
- Pn is the min Vcc point = Energy efficient point

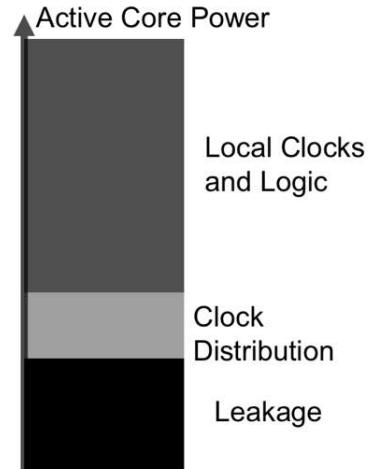
⇒ Pn is the most energy efficient point



- Going up/down the cubic curve of power
  - High cost to achieve frequency
  - large power savings for some small frequency reduction

יש כמה מצבים עובדה שונים שנקראים P-States (Performance-States). במצב P0 המעבד מפיק את הביצועים והספק המקסימליים. יש גם מצבים P1,P2 וכך הלאה עד Ch ובקב עוקב ביצועי המעבד והספק שלו יוזדים. כאמור מ"ה והחומרה מנהלת ביחיד את התזוזה של פעילות המעבד על פני הציר של כל הstates P-States DVFS (Dynamic Voltage & Frequency Scaling) המנגנון זה נקרא באינטל (Dynamic Voltage & Frequency Scaling). איז אלה הם מצבים הביצוע השוניםnP . P-states המטרה שלו היא לאזן בין ביצועים לצריכת הספק של המעבד. איז אלה הם מצבים הביצוע השוניםnP . P-states עובד באותו תדר עבורו מתקיים  $V_{min}$  וכפי שראינו זו הנקודה הכי גבוהה ז"א התדר שייתן הכי הרבה ביצועים ביחס להספק שהמעבד צריך בתדר זה.

- **OS notifies CPU when no tasks are ready for execution**
  - CPU enters sleep state, called C-state
  - Using MWAIT instruction, with C-state level as an argument
  - Tradeoff between power and latency
    - Deeper sleep → more power savings → longer to wake
- **C0: CPU active state**
- **C1: Halt state:**
  - Stop core pipeline
  - Stop most core clocks
  - No instructions are executed
  - Caches respond to external snoops
- **C3 state:**
  - Stop remaining core clocks
  - Flush internal core caches
- **C6 state:**
  - Processor saves architectural state
  - Turn off power gate, eliminating leakage

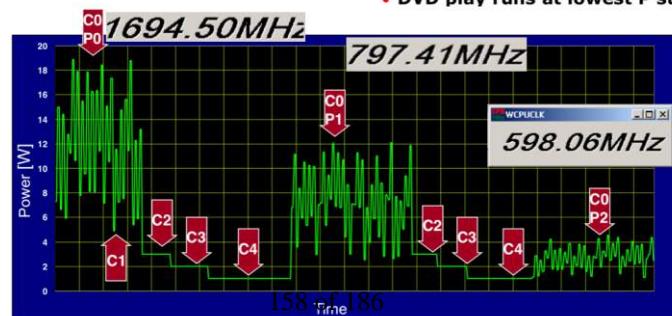


לכל P-state יש כמה C-states, מצבים שונים, שהם מצבים ביןיהם מ"ה בוורתה בהתאם למידת האקטיבציה של העומס הנווכחי: כשהל המעבד עובד הוא יהיה במצב C0 - זה המצב שהמעבד בו הכי אקטיבי ולמעשה זה מצב של חוסר-שינוי. אח"כ יש את המצב C1 שהוא מצב השינה הכי עրני - בכל מצב עוקב שומרים פחות ופחות משאים פעילים כך שאפשר להיכנס לשינה יותר עמוקה מבחינת הספק. ככל שהשינה יותר عمוקה, ככל שאנחנו במצב C-state גובה יותר, יהיה קנס יותר גדול בהעתה המעבד, אך תמיד בכניסה למצב פעילות, P-state כלשהו מתחילה מ-C0, ואם רואים שאין פעילות גבוהה אז עוברים ל-C1 וכך הלאה. מצב C0 עוצר את התקדמות הצינור עם מהזורי השעון אבל הוא עדין מתחזק את תוכן הזיכרונות ולא מכבה לגמרי את המעבד, למשל בשביל לקבל פסיקות. אח"כ ב-3C עושים flush לכל הצינור אבל לא לזכרון פנימיים, וב-6C המעבד מגבה את הריגistros של המעבד ומהש מנתק אותו מהמתה - ככה נחסך גם ההספק הנדייר, אבל כמובן שהעיר את המעבד ייקח יותר זמן והספק כי נדרש לשחרר את מצב המעבד כולל מצב הקאשים וכו'.

נראה איך זה מתבטא במערכות אמתיות: ב-P0 ו-C0 יש את הביצועים הכי טובים והמעבד נגיש לחוטאים. אם המעבד במצב של חוסר מעש (idle) או נרדים חלק ממנו ע"י כניסה ל-C-state מתאים וכך הלאה נרד למצבים C עוקבים ככל שניה פחות פעילים, אח"כ מ"ה תבקש לעבור למצב P1 בו התדר נמוך יותר - מתחילה אותו מ-C0 עד שאנחנושוב במצב idle ואז מתחילה לרדת. מצב P0 שרוואים בשקף הוא המצב בו מתבצעת ניגון של סרט מכוון DVD.

### Putting it all together

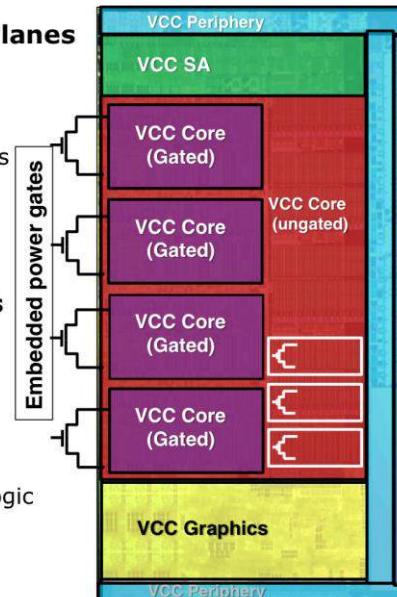
- CPU running at max power and frequency
- Periodically enters C1
- Going into idle period
  - Gradually enters deeper C states
  - Controlled by OS
- Tracking CPU utilization history
  - OS identifies low activity
  - Switches CPU to lower P state
- CPU enters Idle state again
- Further lowering the P state
- DVD play runs at lowest P state



## ניהול הספק: חלוקת-מתח דינמית על-פני הצל'יף

# Voltage and Frequency Domains

- **Two Independent Variable Power Planes**
  - CPU cores, ring and LLC
    - Embedded power gates – each core can be turned off individually
    - Cache power gating – turn off portions or all cache at deeper sleep states
  - Graphics processor
    - Can be varied or turned off when not active
- **Shared frequency for all IA32 cores and ring**
- **Independent frequency for PG**
- **Fixed Programmable power plane for System Agent**
  - Optimize SA power consumption
  - System On Chip functionality and PCU logic
  - Periphery: DDR, PCIe, Display



דבר על חלוקת הספק (חלוקת-מתח) על-פני הצל'יף: הרעיון הוא לתת הספק גביה יותר למי שציריך אותו: יש על הצל'יף כל מיני יישויות שכוללות את הליביות, הkartיס הגראפי, הזיכרון וכו', ורוצhim לחת את ה-Power Budget למי שעבוד כרגע כי יש אפליקציות שעובdot על ליבת אחת ולא על אחרות או בעיקר על הkartis הגראפי ולא על הליביות - לכן צריך יכולת לשנות את ה-frequency/voltage שלקן את ההספק לחלוtin מחלקים בצל'יף שלא עובדים כלל. האיזון הזה נקרא Power Balance ובשבילו יש כמה דברים:

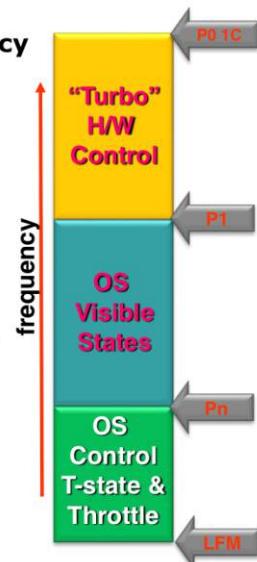
- שני מישורי-הספק (Power Planes) עצמאיים בהם הצל'יף פועל באופן בלתי-תלוי, מישור אחד הוא לכל הליביות ומישור שני לכרטיס הגראפי - מה שאומר kartis הגראפי יכול לעבוד במתח/תדרות שונות מהליביות ובלי תלויה בהן.

- בנוסף להיכולת לנתק ליבות לא-פעילות מהמתח וכן לחסוך את ההספק שנדרש כדי להפעיל אותן (динמי וסטטי) - אז מכל ליבת אפשר לנתק את ההספק בנפרד לתקופה מסוימת, כשפחחות נדרש אותה, ולהחבר אותה למתח שוב כשכן צריך.

התועלת של embedded power gates על-פני מישורי-הספק הוא שהם קיימים לכל ליבת בנפרד ולא חוסכים רק את ה-power leakage כי הוא מנתק את היחידה מהספק לגמרי.

## Turbo Mode

- **P1 is guaranteed frequency**
  - CPU and GFX simultaneous heavy load at worst case conditions
  - Actual power has high dynamic range
- **P0 is max possible frequency – the Turbo frequency**
  - P1-P0 has significant frequency range (GHz)
    - Single thread or lightly loaded applications
    - GFX <>CPU balancing
  - OS treats P0 as any other P-state
    - Requesting is when it needs more performance
  - P1 to P0 range is fully H/W controlled
    - Frequency transitions handled completely in HW
    - PCU keeps silicon within existing operating limits
  - Systems designed to same specs, with or without Turbo Mode
- **Pn is the energy efficient state**
  - Lower than Pn is controlled by Thermal-State



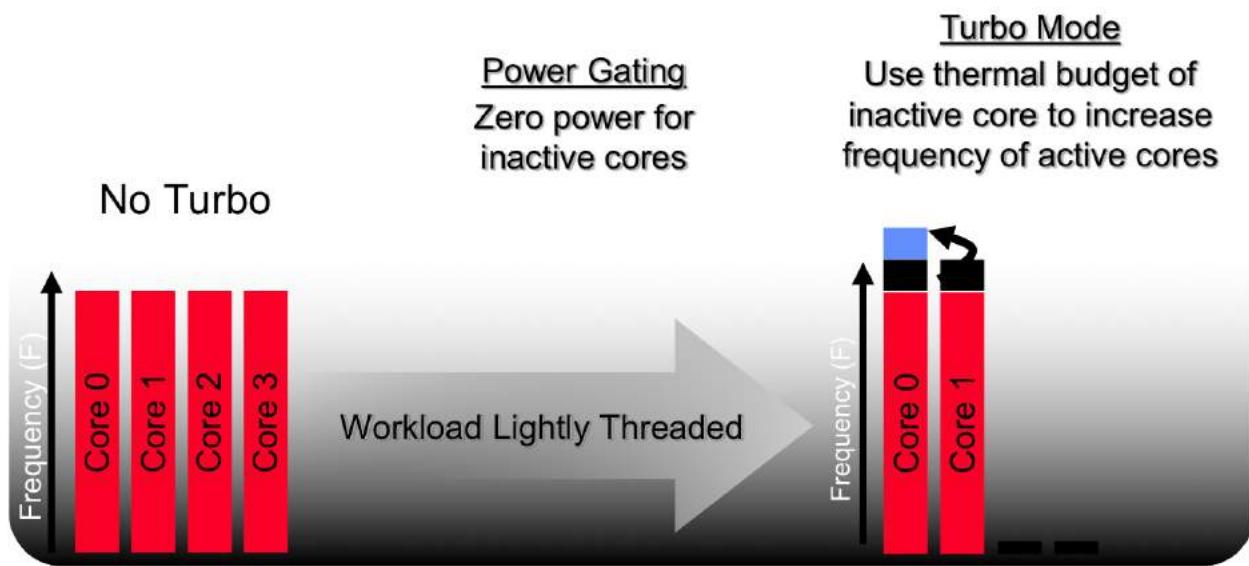
נראה איך השיטה במתוח של הרכיבים השונים יכולה לעזור למקסם את ה Power Budget לטובות הביצועים: התדרות שモבנתה שהמעבד יעבוד בה, ככלmor זו שモבנתה ע"י היצרך היא זו שבמצב P1 - אפליקציה חמה יכולה לרווח בתדר שמתאים למצב זה. אבל כפי שלמדנו ב-P1 המעבד לא רץ בתדר המקסימלי אלא ב-P0 - תדר זה נקרא התדר של מצב-טורבו.

אפשר לרווח במצב טורבו בכל מיני תרחישים שתclf נראת ומה שמשותף להם הוא שהם לא מפעילים באופן מלא את כל המערכת אלא בעיקר את הליבות - המערכת מורכבת מהרבה רכיבים אך לא תמיד היא מועמסה עם TDP application שרצה בכל הליבות ובכרטיסי הגרפי ובזיכרון וכו' וכו'. כשה恂רים את המערכת מפרסמים את הה-guaranteed frequency במצב עבודה מלאה, מצב של עומס, כגון כשמיירם את התדרות מפרסמים תדר שמתאים למצב P1 למרות שבפועל בין P0 לביון P1 יש טווח גדול בתדר של לפחות כ-500Mhz. אז מצב-טורבו הוא ההספק המקסימלי שאפשר לרווח בו ואז מקבל ביצועים יותר גבוהים. נחזר על זה: P0 זה ה-Max possible frequency P1 זה ה-guaranteed frequency והמרווח בינויהם הוא טווח מאוד משמעותי. מתי אפשר לרווח בתדר המקסימלי? כדי מעט חוטים ולכן לא כל הליבות רצות, או שMRIIZIM אפליקציה שלא כורכת הרבה חישובים גרפיים ואפשר להקצות את ה-Power Budget ל-CPU.

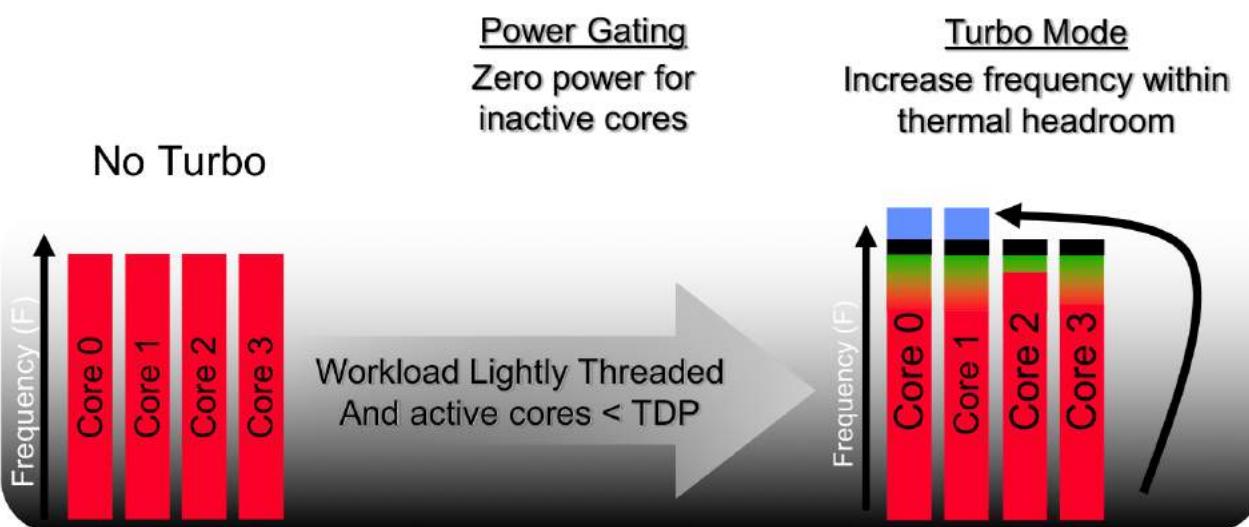
מה יכולה לבקש מהמעבד לרווח Pstate כלשהו אבל זה מנוהל ע"י החומרה כלומר התוכנה תבקש לרווח ב-P0 והחומרה תודע שהתנאים הם כאלה שבאמת אפשר לרווח שם ולא מתחממים מעלה המידה - לשם כך יש יחידה בשם Power Control Unit (PCU) שנמצאת בתוך המעבד והיא מבטיחה שלא נעבור את הגבול מבחינת ריצה בתדר גבוה מדי עד כדי נזק למערכת. כל התחרום בין P0 לבין P1 נשלט ע"י החומרה ומבודק ע"י מ"ה - מעבר לזה החומרה מחליטה באיזה מצב לרווח. Pn הוא המצב המינימלי שמה'ה יכולה לבקש.

ה-PCU היא יחידת החומרה שמנוהלת את הבקשות לשינוי מצב העבודה: מקבלת בקשה ממ"ה לעבור למצב של ביצועים גבוהים ונותנת לה את זה רק אם זה אפשרי.

דוג' :



בהתחלתו יש לנו ארבע ליבות שמספקות ביצועים רגילים - ללא טורבו. ברגע כלשהו המחשב מתחילה לבצע אפליקציה על שני חוטים בלבד, ולא צריך ארבע ליבות, אפשר להעביר שתי ליבות למצב של power gated כך שהן לא יבזזו הספק (נקרא גם מצב 0-power), ואז יהיו רק שתי ליבות פעילות, ככלומר אפשר להעביר את התקציב לשתי ליבות שרצות ולבקש מהמעבד לחתה להן תדר יותר גבוה - ה-PCU צריך לנוהל את זה ולהבטיח שם נרוץ בתדר יותר גבוה עדין נהייה בגבולות ה-Power Budget הכלול.

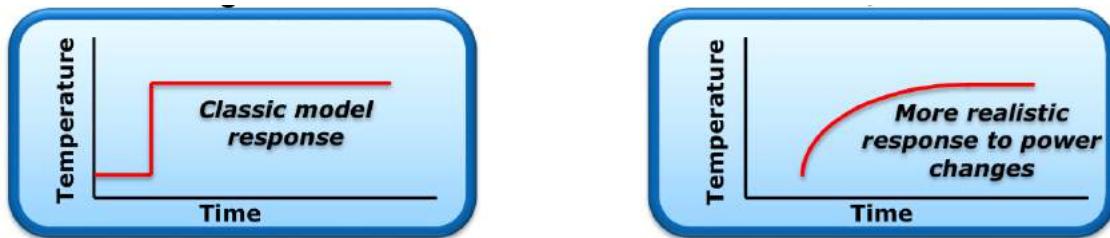


באופן כללי תרחישים בהם ניתן ליהנות ממצב-טורבו המ בהרצת אפליקציות שאין חמות: צריכת ההספק שלתן קטנה מה-TDP, וכן ניתן להריץ אותן בתדר גובה ולקבל יותר ביצועים ועודין להיות בתחום מגבלת ההספק - ה-Power Envelope. למשל אם עברתי לאפליקציה בהן יש ליבות שרצות בעומס קל יותר מאשר או עכשו איפשר להעביר ליבות הפעולות עוד יותר תקציב על-חשבון הליבות שאין עמוסות. כל ניהול הדינמי של הטורבו נעשה באישור המעבד שלו לקח את תקציב ההספק הנוכחי לו ומשנה את הביצועים שלו בהתאם ל-Power Budget זמין באותה נק'.

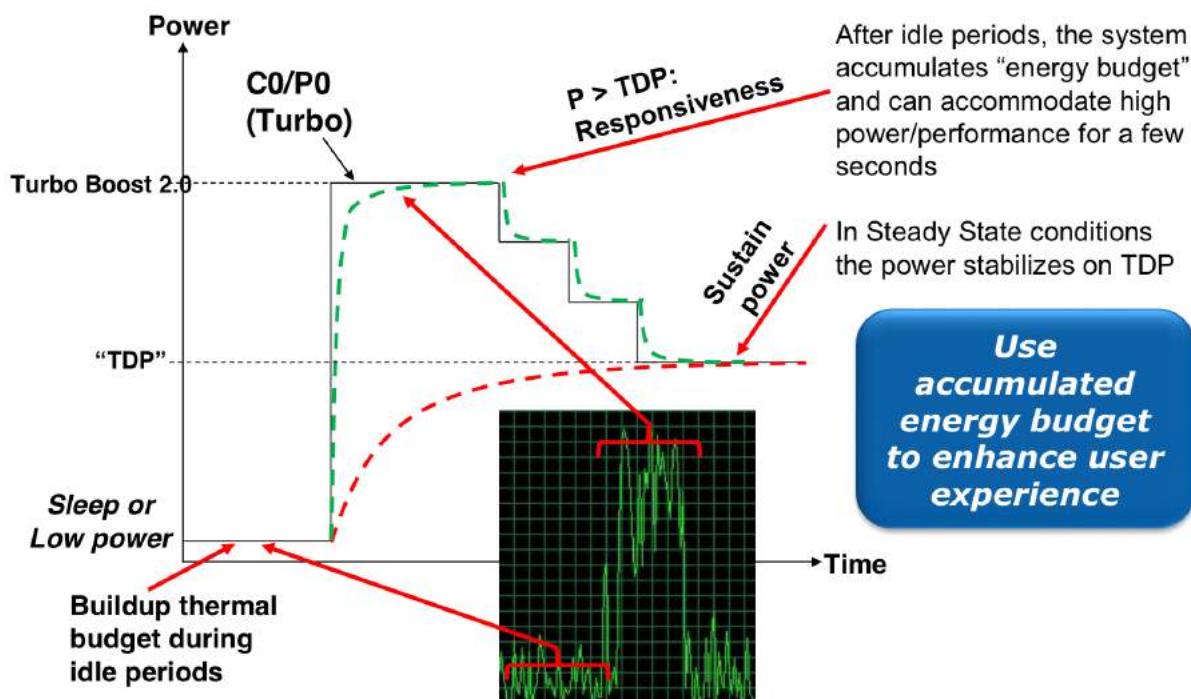
הבהרה: יש טווח גדול של תדרים בין P0 ל-P1, מ"ה מכירה רק את שני המ מצבים האלה אבל החומרה יכולה לעבוד בפועל באמצעות.

## Performance-Boosts (Intel Turbo Boost Technology)

כעת נדבר על פרצי-ההספק (Boosts) שהזכירנו:



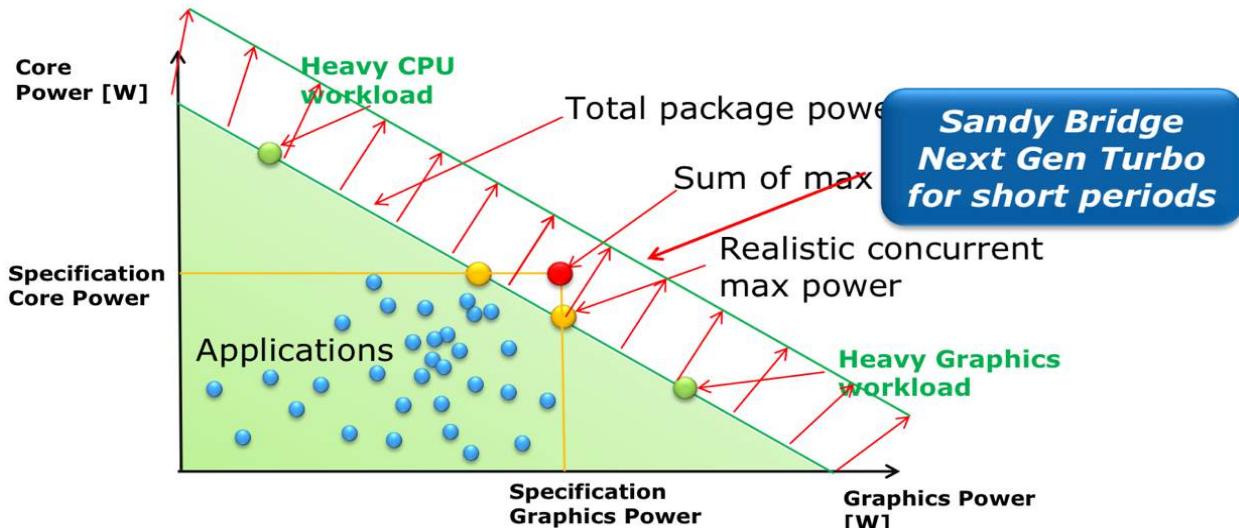
נניח שהיינו במצב של idle ועברנו למצב אקטיבי - אז ההספק לא מתפתח מידית אלא החימוש במערכת נעשה על-פni עקומה כלשהי.



בזמן החימוש רוצים לאפשר תדר יותר גבוה מהמקסימלי כלומר רוצים לנצל את הזמן עד שmaguiim לחום המרבי המותר (TDP) כדי לחת ביצועים יותר טובים. אז ננצל שהוא שנקרא burst mode. למשל אם לחצתי על כפתור באפליקציה כלשהי אני רוצה לקבל את הביצועים המקסימליים האפשריים - זה יכול להיות גם למשר דקה למשל בה אפשר לרצ בتردد יותר גבוה מהתדר שמשמעותם "על הקופסה" ולתת הרגשה של ביצועים גבוהים ולאט לאט כשמהערכות תחיל תחילה להתחמם לדת בתדר כדי לעמוד בדרישות ההספק.

## Core and Graphic Power Budgeting

- Cores and Graphics integrated on the same die with separate voltage/frequency controls; tight HW control**
- Full package power specifications available for sharing**
- Power budget can shift between Cores and Graphics**



איך זה נראה מבחינה ביצועים של אפליקציות: כל נק' בתרשים מייצגת אפליקציה כלשהי, בציר x וראים כמה מההספק היא צורכת עבור גרפיקה ובציר ה-y רואים כמה הספק ריא צריכה עבור המעבד - למשל יש אפליקציות שלא צורכות כמעט את הרכיטיס הגרפי אבל צורכות הרבה כוח-מעבד (בشكף מסומן בHeavy CPU ב-Sandy Bridge Next Gen Turbo) ולהיפך ויש גם אפליקציות שאין יותר מזמן מבחןיה זו שהן צורכות מידה מסוימת של כוח-מעבד וגם של כוח-חישוב גרפי. הקו הוא ה-TDP, כלומר ההספק המקורי האפשרי. אז בזכות התקופות שבהמעבד קר אפשר לתקופות קצרות לזרע בתדר יותר גבוה ככלarlo היה לנו את תקציב-הספק יותר גדול, מה שאומר שאפשר לקבל ביצועים יותר גבוהים, בדר'כ' לכמה שניות או דקה, ואנו לחזור לו של ההספק בו תוכנה המערכת יכולה לעבוד - הקו המקורי של ה-TDP.

בבהרה: הקו "מתרחק" כשעוביים ממצב של idle למצב של עבודה ואז יש בערך דקה בה המעבד יכול לזרע בתדר יותר גבוה ככללו שהוא-TDP מתאים לו רק יותר - למשל נניח שהמערכת יודעת לדרוש 8W אבל לשך דקה או שתים אפשר לזרע ככללו המעבד במעטפת שיכולה לדרוש 15W.

סיכום: ניהול נכון של ה-Power Budget הוא מאד חשוב כי באמצעותו אפשר לקבל בפועל הרבה ביצועים וכל האלגוריתמים שהוכנסו למעבד בשיל זה בשנים האחרונות שיפורו את חווית המשתמש - למשל זה גורם לכך שאפשר להכניס מעבדים חזקים לקופסאות יותר ויותר קטנות - דוג' לכך היא שמחשבים ניידים היום הם הרבה יותר קלים ודקים וудין מספקים ביצועים טובים - עד לפני 5 שנים רוב הניידים היו באיזור ה-35W והיום אפשר بكلות להכניס מחשבים ניידים لكופסאות של 8W.

### שאלת מבחן

cutet נפתרו שאלה מבחן כדי לראות איך מתרגמים את החומר שלנו היום לשאלות:

השאלה עוסקת בבחירה ובבחירה בין שתי אלטרנטיבות לבניית מערכת לשרת (מערכת בעלת הרבה מאוד מעבדים שנועד להיות עסוקים בכל-עת): אופציה אחת היא לבנות את המערכת מליבות גודלות ואופציה אחרת היא להשתמש בלבבות קטנות - לכל ליבת יש את המאפיינים שלה ורוצים להחליט באיזה משתי היבשות עדיף להשתמש.

## Question

דרושים לתכנן מערכת Micro Server Multi Processor ויש להשוות בין שני סוגי מעבדים כך שניתן יהיה להריץ אפליקציות חממות (TDP) ביעילות תור קבלת ביצועים (MP performance) הגבוהים ביותר ביחס לתקציב ההספק.

מעטפת הספק של המערכת היא 60Watt כאשר שני שליש מתקציב ההספק הינו עבור המעבדים – Core's (והשאר עבור ה – Uncore).

יש להשוות בין שני סוגי מעבדים אפשריים לצורך בניית המערכת, מעבד גדול ומעבד קטן: הגדל גודלו ארבע מילימטר רבוע<sup>2</sup> 4mm<sup>2</sup> והקטן גודלו שני מילימטר רבוע<sup>2</sup>.2mm<sup>2</sup>. המעבד הגדל הואIPC=3 4wide עם IPC=3 והקטן הוא מעבד 3wide עם IPC=2. IPC זה מתקבל כאשר מרכיבים אפליקציות חממות (TDP) ללא טעויות בחיזוי הקפיצה.

ההספק הסטטי Leakage Power הוא 0.25Watt למילימטר רבוע (ההספק הסטטי קבוע ולא משתנה עם המתח/תדר).

הקיבול הדינامي של כל מעבד נתון כפונקציה של הIPC של האפליקציה אותה הוא מרים וערכו הוא:  $Cdyn = IPC \times 750pF$

להלן נתונה טבלה המראה את נקודות מתח ותדר אפשריות לעבודת המעבדים הגדל והקטן.

| Volt's   | מתח ב Volt's | תדר ב Ghz מעבד קטן | תדר ב Ghz מעבד גדול |
|----------|--------------|--------------------|---------------------|
| Vmin=0.7 | 1            | 0.75               |                     |
| Vmin=0.7 | 1.25         | 1                  |                     |
| Vmin=0.7 | 1.45         | 1.35               |                     |
| 0.8      | 1.75         | 1.75               |                     |
| 0.85     | 2            | 2.25               |                     |
| 0.9      | 2.25         | 2.5                |                     |
| 0.95     | 2.5          | 3                  |                     |
| Vmax=1   | 2.75         | 3.5                |                     |

יש לתכנן שתי מערכות אחת עם מעבדים גדולים ואחת עם מעבדים קטנים כך שניתן יהיה להריץ אפליקציות חממות (TDP) ביעילות תור קבלת ביצועים הגבוהים ביותר ביחס לתקציב ההספק. עבור כל מערכת יש למצוא את מספר המעבדים הדרושים.

עבור כל מערכת חשבו את הביצועים (MP performance instructions per second) אפליקציה חמה בנקודות  $V_{min}$ . לאור התוצאות איזו מערכת יהיה ממליצים ליצור? זו עם המעבד הגדל או זו עם המעבד הקטן.

$$MP\ Performance = \#of\ cores \times IPC \times f$$

# Solution

רוצים להשוות בין שני סוגי מעבדים כך שנitin להריצ אפליקציות TPD ביעילות תוך קבלת ביצועי-MP (Multi-Processor). מה שמאפיין מערכת של שרת היא שככל המעבדים בה כל הזמן עומסים - זה לא כמו שלמדנו קודם שאפשר לעבור מצב של idle-active כדי צבירת power credit - פה דברים כל הזמן רצים במצב קבועה, אפליקציות חממות - מערכת של שרת בנזיה לכך שהיא כל הזמן נותנת Throughput ולפי זה צריך לתקן את ה- MP Performance.

אם נרצה לקבל את היעילות הכי גבוהה במערכת זאת אז יהיה כדאי כל מעבד בנק' ה-Vmin שלו כך שהיעילות (היחס בין הביצועים להספק) תהיה הגבוהה ביותר וניתן יהיה לנצל את שאר הרесפוק כדי להשתמש בעוד מעבדים - מה שקובע כמה מעבדים אפשר לשימוש זה להספק, ה-Power-Budget, שיש לקופסה ואם נróż בנק' של יעילות גבוהה מאיידיאלית אז במקום להכניס X מעבדים ב-100W יוכל להכניס פחות מ-X מעבדים - אז חשוב שהסטודנט שפותר את השאלה יבין שככל מעבד המערכת זו צריך לרוץ בנק' ה-Vmin שלו כדי לקבל את היחס power/perf הכי יעיל זה תמיד נכון במערכת מרובת מעבדים (שנوعדה להריצ אפליקציות חממות).

$$\text{Power} = \text{leakage} + C \times f \times V^2$$

$$\text{Leakage of the core big Core} = 4 \times 0.25W = 1W$$

$$\text{Leakage of the core small Core} = 2 \times 0.25W = 0.5W$$

To maximize Total MP performance need to work at Vmin for TDP app

Big Core:  $V = 0.7V$   $f = 1.35Ghz$

Small Core  $V = 0.7V$   $f = 1.45Ghz$

For each system need to calculate the per Core Power budget for most efficient operating point which is Vmin when Hot App is running, this will allow maximizing number of cores and overall performance:

$$\text{For Big Core System: } P = C \times V^2 \times f + 1 \Rightarrow 3 \times 0.75 \times 0.7^2 \times 1.35 + 1 = 2.49W$$

$$\text{For Small Core System: } P = C \times V^2 \times f + 0.5 \Rightarrow 2 \times 0.75 \times 0.7^2 \times 1.45 + 0.5 = 1.57W$$

$$\text{Power budget for all cores} = 60 \times 2/3 = 40W$$

$$\text{Number of Cores: Big } 40/2.49 = 16 \quad \text{Small } 40/1.57 = 25$$

When building a system with big Cores putting 16 big Cores will provide highest MP performance in the system power envelop

When building a system with small Cores putting 25 small Cores will provide highest MP performance in the system power envelop

מעטפת ההספק בשאלת היא 60W כאשר נתון שני שליש ממנו נועד עבור הליבוט והשאר עבור הציף, ה-Uncore. אז סה"כ הליבוט מקבלות 40W, זה נכון בשתי האלטרנטיבות שנגבות. כאמור יש להשוות בין שני סוגי מעבדים לצורך בניית המערכת - מעבד גדול ומעבד קטן:

- המעבד הגדל הוא בגודל  $4[mm^2]$  והקטן בגודל  $2[mm^2]$  - הגודל של המעבד הולך להשפיע על ה- $P_{lkg}$  כאמור הוא פו', של  $Z$  כלומר של שטח המעבד.
- המעבד הגדל הוא מעבד ברוחב 4 (בפועל עם  $3[IPC]$ ) והקטן ברוחב 3 (בפועל עם  $2[IPC]$ ).
- ההספק הסטטי קבוע ולא משתנה והוא  $W_{25} = 0.25$  לכל מ"מ-רבוע - لكن  $P_{lkg}$  של המעבדים הקטנים יהיה  $W_{0.5}$  ושל הגודלים הוא  $W_1$  (כי גודל כל מעבד גדול הוא  $[2mm^2]$ ).
- הקיבול הדינמי של כל מעבד נתון כפוי של ה- $IPC = 0.75nF$  ורכזו  $F_{0.75nF}$ , אך במעבד הקטן הקיבול הדינמי הוא  $Cdyn = 1.5nF$  ובגודל הוא  $0.75nF \cdot 3$ .
- הטבלה בשאלת מראה את נק' העבודה השונות של מתח ותדר-אפשרי לכל אחד מהמעבדים - המתח המינימלי  $V_{min}$  הוא  $7V$  והמקסימלי  $17V$ , ויש תדר שמתאים לכל נק' מתח. נשים לב שכי שהסברנו כשאני מושיר להקטין את התדר המתח לא יורד מתחת ל- $V_{min}$  וכן אין טעם להמשיך להוריד מתחת ל- $V_{min}$ , מצד שני תדר שדורש מתח גבוה מ- $V_{min}$  אינו יעיל וכן התדר בשורה השלישית, ככלומר התדר המקסימלי שפועל בנק' ה- $V_{min}$ , הוא יהיה התדר הכי ייעיל וכן כדאי להשתמש בו. סטודנט שפותר את זה צריך להבין שכדי לקבל את הייעילות הכי גבוהה צריך למקסם את מס' המעבדים שנitin להכניס בתקציב ההספק ולכן למקסם את הביצועים, ובשביל זה כל מעבד צריך לרווח בנק' ה- $V_{min}$  ככלומר שהמעבד הגדל ירווח ב- $1.35GHz$  והקטן ב- $1.45GHz$ .

כאמור יש לתקן שתי מערכות, אחד עם מעבדים גדולים ואחת עם מעבדים קטנים, כך שנitin יהיה להריץ אפליקציות חמורות ביעילות תור קבלת ביצועים גבוהים-bijouter במסגרת התקציב ההספק. בנוסף עברו כל מערכת יש למצוא את מס' המעבדים הדרוש.

ההספק הכלול מרכיב מההספק הדינמי וההספק הנדייף:

$$P_{tot} = P_{lkg} + P_{dyn}$$

את  $P_{lkg}$  מצאנו:  $W_1$  עבור הליבה הגדולה ו- $W_{0.5}$  עבור הקטנה.

זכור  $f * V^2 * P_{dyn} = Cdyn$ . כדי למקסם את ה- Overall MP Performance צריך לעבוד ב- $V_{min}$ , ככלומר התדר של המעבד הגדל צריך להיות  $1.35Ghz$  ושל הקטן  $1.45Ghz$ . עכשו צריך לחשב את ההספק הדינמי של כל ליבת לפיה הנוסחה שנitinנו לנו במעבד הקטן הקיבול הדינמי הוא  $2 = 0.75nF \cdot 1.5nF = 2.25nF$ . לכן ההספק הדינמי של המעבד הגדל הוא:

$$P_{tot}(Large) = Cdyn * V^2 * f = P_{lkg} + P_{dyn} = 1 + 1.35 * 0.7^2 * (3 * 0.75) = 2.49W$$

באופן דומה עבור המעבד הקטן יוצא שההספק הוא  $W_{1.57}$

עכשו מחלקים את סך ההספק הזמין למעבדים,  $W_{40}$ , בהספק של כל מעבד ומビינים כמה מעבדים מכל סוג נכנסים במערכת שלנו אז יתכונו  $40/2.49 = 16$  מעבדים גדולים ו- $25 = 40/1.57 = 16$  מעבדים קטנים - נשים לב שייתכן צורך לעגל כלפי מטה (כי אי אפשר למשל להכניס חצי-מעבד).

הערה: במערכת זאת אין שינוי דינמי של העומס ולכן תמיד כדי לרווח ב- $V_{min}$  אבל במצבים לא תמיד כל הליבות יהיו עמוסות בכל רגע ואם ברגע נתון רק 20 מתוך 25 ליבות הן עמוסות אז אפשר להריץ אותן בתדר יותר גבוה כדי למקסם את ההספק האפשרי תוך כדי העלאת התדר - אבל כשהקל עמוס באופן עקבי צריך להריץ תמיד את כל 25 הליבות אז MP Performance הגובה היותר תתקבל בנק' שווי המשקל, ככלומר ב- $V_{min}$ .

עכשו צריך להחליט איזה מהמערכות יותר טובה - לשם כך מכפילים את מס' המעבדים, ה- $IPC$  והתדר ומוצאים את ה- MP Performance של כל מערכת:

MP Performance = #of cores×IPC×f

For Big Core: f=1.35Ghz; IPC\_tdp=3; #of cores=16

TDP MP Performance=  $16 \times 3 \times 1.35 = 64.8 \times 10^9$  Instructions per second

For small Core: f=1.45Ghz; IPC\_tdp=2; #of cores=25

TDP MP Performance=  $25 \times 2 \times 1.45 = 72.5 \times 10^9$  Instructions per second

The system with the small cores provides the highest MP performance and takes less area ( $25 \times 2 = 50 \text{mm}^2$  vs  $16 \times 4 = 64 \text{mm}^2$ ) so we recommend to build the system with the small cores provide highest MP performance in the system power envelop

ראויים שהמערכת עם המעבדים הקטנים נותנת ביצועים יותר טובים (72.5 מיליארד פקודות בשניה לעומת 64.8 מיליארד פקודות במערכת עם המעבדים הגדולים). כמו כן נביט על השיטה של שתי המערכות: המערכת עם

המעבדים הקטנים תופסת =  $644^2 \times 16 \text{[mm}^2]$  והגדולה =  $502^2 \times 2 \text{[mm}^2]$ .

מכיוון שהמערכת של המעבדים הקטנים גם נותנת ביצועים גבוהים יותר אז היא יותר משתלמת.

דוג' זו מראה לנו שלא כדאי ל选取 מעבד קטן ו"לשפר" אותו בעלות גדולה מדי (הגדלה של השטח וההספק בעלי הגדלה תואמת ב-IPC) ומדד לכך הוא האם הביצועים הכלולים שלו השתרעו בצורה משמעותית כמו התוספת בצריכת ההספק - במקרה שלנוIPC עליה מ-2 ל-3 (פי 1.5) וההספק עליה מ-1.57 W ל-2.5 W (פי 1.6) ואכן השיפור התרברר כמעט-כדי.

## Multithreading מס' 13

היום נדבר על מערכת מקבiliarת שנקראת מערכת MT (Multithreading) או מערכת Multi- MT (Simultaneous Multi-). מערכת זו מאפשרת לקבל מקבiliarות יותר גודלה ע"י הרצת תכניות שונות על אותה ליבת Threading).

- **Multiprocessor systems have been used for many years**
- There are known techniques to exploit multiprocessors
- Chip Multi-Processing (CMP): multiple Cores on the same die
- Applications consist of multiple threads or processes that can be executed in parallel on multiple processors
- **Thread-level parallelism (TLP) – threads can be from**
- The same application
- Different applications running simultaneously
- Operating system services

סוגי מקבiliarות שראינו בקורס עד כה:

- מערכת בעלת מס' ליבות, למשל ארבע, ולה קרנו מערכת (MP) Multi-Processor. במערכות MP ניצלו את העבודה שיש מס' מעבדים - ולצורך כך נכתבו תכניות מקבiliarות שמודעת לנצל מס' ליבות וע"י כך להאיץ את העבודה.
- לפניהם כתנית נכתבה בהמה שנקרא Single-Thread (ST) כלומר כרצף של פקודות שהמעבד יודע להריץ. מקבiliarות בהקשר זה היא היכולת של מעבד להריץ תכנית ST ב-OOO - קרנו לה ILP (Instruction Level Parallelism).
- הזכנו בקורס גם פקודות חדשות שייצרו מקבiliarות, קרנו לה SIMD (Data-Level Parallelism): דוג' לכך היא פקודה SIMD שביצעה פעולה כלשהי באופן וקטורי (פעולה וקטורית בקיצור).

היום נדבר על מקבiliarות שנקראת Thread Level Parallelism (TLP) שזה ההפרש בין Thread Level Parallelism (TLP) ובין Thread Level Parallelism (TLP). Thread Level Parallelism (TLP) שeroxם זה שרצו לשבור את תכניות כמה חוטים נפרדים לגמרי שירצטו במקביל. Thread Level Parallelism (TLP) יכול להיות באותה אפליקציה - למשל Word כתוב כאוסף חוטים שרצו ביחיד, למשל יש את speLLCheck שרצ במקביל לשאר העבודה ולא תלוי בה, כמו ייחידת איסוף האשפה של התכנית. החוטים קיימים כדי לחתם מקבiliarות בתחום אפליקציה מסוימת וכן כדי לחלק את העבודה לחתם-משימות שונות. המטרת שלנו היום היא לתכנן חומרה שמסוגלת להריץ כמה חוטים במקביל. הבירה: ב-Thread Level Parallelism החוטים לא בהכרח של אותה אפליקציה, יכולים להיות משתי אפליקציות שונות לחלוטין.

- **Increasing single thread performance becomes harder**
- And is less and less power efficient

אחד הסיבות לכך ש"כ חשוב היה העבודה שלהוציא מקבiliarות מתכניות Single Thread הופך להיות יותר קשה - דיברנו על כך שבמכונות OOO צריך חלונות גדולים מthoseם אפשר למצוא מקבiliarות וגם בדרכים נוספות. אפליקציות ST הן מוגבלות מבחינה מקובל - אז רוצים לפרט את הרף הזה ואחת הטכניות היא לעבור ל-MT. גם בהקשר של הספק כדי לבנות מכונה יותר טובה ב-ST עליה בדר"כ הרבה הספק ולכן מבחןת הספק ו-Efficient - אז מփשים דרכיהם להגדיל את המקבiliarות בצורה שתיהיה ידידותית ויעילה מבחינת הספק ו-Multithreading היא פתרון טוב ויעיל לכך.

עכשו נדבר קצר על הסכנות השונות של מכונת Multithreading - יש כמה אפשרויות למש את זה:

.1

### • Switch-on-event multithreading

- Switch threads on long latency events such as last level cache misses
- Works well for server applications that have many cache misses
- Does not cover for branch mispredictions and long dependencies

.Miss: בשיטה זו חוט כלשהו רץ על הליבה עד שהוא מגיע לגישת זיכרון שיש עליהם Miss. בשלב זה החוט תקוע מבחינת המעבד כי הוא צריך לצאת לזכרון החיצוני (ל-DRAM) וזה לוקח 300-400 מיליאדים לפחות אחד יוצא לזכרון וושים החלפת חוטים במעבד ועובדים לבצע חוט אחר. כשחוטים מוגעים לגישת זיכרון עליה הוא מקבל Miss, למשל load בעיתוי, אז שוב במקומם שהמכונה תחכה אナンנו חוטים לחוט הראשון (בתקופה שהוא חזר) או עוברים לחוט שלישי שממתין (באופן תאורטי - בפועל עובדים כאמור).

חוטים

שני

רק

עם

.2

### • Simultaneous multi-threading (SMT)

- Multiple threads execute on a single processor simultaneously w/o switching
- When one thread is stalled / slowed down, other thread gets more resources
- Makes the most effective use of processor resources

### • Running ≥2 threads increases the utilization of core resources

- When one thread is stalled (on a cache miss, branch misprediction, or a long dependency), the other thread gets to use the free resources
- The RS can find more independent mops

במציאות לא עושים SOE אלא סכמתה שנקראת SMT ובזה רוצים באמת להריץ שני חוטים בו-זמנית על מכונה אחת ולא רק להחליף ביניהם ובכל רגע רק אחד יהיה פעיל - בשביל זה חלק מהרכיבים במעבד ישוכפלו כדי לחתה תמייהה בינהו שנקרא הקשר, context, למשל הרגיסטרים הארכ' ישוכפלו ולכל חוט יהיה הקשר משלהן, ערכי וגייסטרים שלו. הרעיון הוא בעצם להיות מסוגלים להריץ את שני החוטים האלה ולחתה לשנייהם להתקדם בו-זמנית, שני העקרונות שרצוים להבטיח ע"י כר' נקראיים Forward-Progress ו-Fairness: מצד אחד רוצים להבטיח שלא תהיה הרעה, Starvation, ככלומר שלא יהיה חוט אחד תקוע בלי לקבל זמן מעבד בכלל, ומצד שני שהמערכת יכולה תמיד לתקדם.

למרות שימושם במילה Multi כדי לرمז על ריבוי גدول (באיינטלי קוראים לזה Hyper-Threading) מכונת שתומכות ב-HT תומכות בכך עד שני חוטים שרצים בו-זמנית על אותה ליבה. אז כאמור היום נתמקד בה מה שצריך לעשות בחומרה כדי שהמעבד ידע להריץ שני חוטים במקביל על אותה ליבה. שאלה: לא עדיף פשוט להפריד את העומס לשתי ליבות? באופן עקרוני כן אבל אי אפשר לשים כמה ליבות שנרצה משיקולים של הספק וגם של העומס על המערכת - ברור שאם המערכת יש שתי ליבות פנויות או היא תשלח כל חוט לLiverה שונה אבל כשאין ליבות פנויות ויש כמה חוטים אז הם צריכים להתחולק על Liverה פיזית ובשביל כר' עוברים ל-SMT שאנונו לומדים היום.

از שעובדים עם חוט אחד כל המשאים של המכונה משרתים את החוט הזה, אבל לעיתים נוצרים "חורים" - המכונה לא מנוצלת ב-100%, ולהקניות את החוט השני לפועלה גורם לה להיות מופעלת יותר טוב. נשים לב ש

Execution Unit SOE זה גרעיניות מאוד גדולה, בעוד ב-SMT ההחלה היא בכל התפנות משאב כלשהו (למשל Execution Unit לשאהי) של המcona (ברענינות של מהזור שעון).

זכור במערכת OOO ה-RS מחייב פקודות בלתי-תלוויות לשולח ליחידות הביצוע - ה-RS הוא תור גדול שזכור הרבה פקודות ויכול לשולח לביצוע פקודה ברגע שככל הקלטים שלו מוכנים, כל התלוויות שלו resolved ויש יחידת-ביצוע מוכנה לפעולה. זה עדין יכול להשאיר מצבים בהם יחידות ביצוע מסויימות לא עובדות כלל (למשל כי מחדים למקורות של פקודה שיחזרו) - הרעיון של SMT הוא לנצל את המשאים הפנוים במcona כדי להשתמש בה בשביל שני חוטים.

## Hyper-Threading Technology

- **Two logical processors within one physical core**
  - Sharing most execution/cache resources using SMT
  - Look like two processors to the SW (OS and apps)
- **Each logical processor executes a software thread**
  - Threads execute simultaneously on one physical core
- **Each logical processor maintains its own arch. state**
  - Complete set of architectural registers
    - General-purpose registers, Control registers, Machine state registers, Debug registers
  - Instruction pointers
- **Each logical processor has its own interrupt controller**
  - Handles the interrupts sent to the specific logical processor

אי-ב-HT כל מעבד פיזי יתחלק לכמה ליבות לוגיות - כאמור בימינו ב-SMT יש רק שני חוטים שרצים ביחד באופן שבאותו מהזור אפשר לעשות dispatch לפקודות מהזוט-0 וכן לפקודות מהזוט-1 כל עוד כל המשאים פנוים. אז המעבד יתחלק לשתי ליבות לוגיות. נשים לב שהזוט-0 והזוט-1 רצים במקביל לא אומר שהם יכולים לקרו את ה-data אחד של השני - ההקשרים שלהם, contexts, עדין מבודדים אחד מהשני.

אי יש שני מעבדים לוגיים בתוך ליבה פיזית אחת ולמרות שהם חולקים את משאבי המcona במובנים מסוימים כמו שני מעבדים נפרדים (הברורה: מ"ה.cn מודעת לקיום של ליבות פיזיות וליבות לוגיות) שלכל מעבד (לוגי) יש לו את המצביע הארכיטקטוני שלו, כולם ה-LCR3 משלו, רגיסטרים ארכ' כמו EAX,EBX,EIP וכמו כן מבון Instruction-Pointer משלו שאומר איפה החוט נמצא בתוכנית. לעומת זאת הקאשים לא נפרדים, לא משכפלים אותם - עדין יש רק L1,L2 אבל בכל שורה במטמון יהיה בית שאומר האם השורה שייכת ל-thread 0 או ל-thread 1.

התוצאות של הרצת שני חוטים במקביל הוא בערך 7%, לעומת 7% בתוספת של חומרה למעבד מאפשרת לנו לתמוך ביכולת להריץ שני חוטים. בתמורה זה נותן מהו כ-30% תוספת לביצועים - זה אומר שאם היינו משתמשים על המcona היחידה ומודדים לכמה פקודות עושים Retire בכל מהזור שעון אז במכונה המשולבת עושים פי 1.3 ממה שעושים בחוט יחיד בלבד. למשל אם בחוט אחד היינו מסיימים שלוש פקודות בכל מהזור אז עם SMT נרץ שניים ביחס כך שהם ביחס יסימו 4 פקודות בכל מהזור. לא בהכרח כל חוט יסיים שתי פקודות: יכול להיות שאחד יתקדם יותר טוב מהשני (אם כי מבטחים Fairness כלומר כל החוטים יתקדמו בקצב כלשהו). חשוב להבין שעדיין חוט אחד בלבד על המעבד יתקדם יותר מהר מאשר אם הוא חולק אותו גם עוד חוט, למשל כי יש לו פחות Cache Misses. נסביר את זה במספרים: אם הביצועים של כל חוט בלבד הם 100% אז בהרצת שניים במקביל כל חוט יירוץ ב-65% מהביצועים המקוריים שלו, או שחותן אחד יירוץ ב-80% וחוט אחר ב-50% - וביחד אנחנו נראה שישר של 30% בביצועים שהיו אם היינו מרכיבים את החוט בנפרד - אז הטכנולוגיה משפרת את throughput של מערכת-חוטים אבל ברמת החוט הבודד הביצועים יורדים בהשוואה לרכיבתם על המעבד.

- **Pipeline arbitration points select the thread gets to use a resource in a given cycle**

- If both threads are active and require the resource
  - Use a “ping-pong” scheme to switch between the two threads on a cycle-by-cycle basis
- If only one thread has work to do and requires the resource
  - Allow that thread to get the full resource bandwidth

- **The thread select scheme**

- Achieves fairness between the threads
- Gives full bandwidth to one thread in case the other thread does not require the given resource

עכשו רוצים לראות איך בונים מערכת SMT - מה צריך להוסיף בחומרה כדי להיות מסוגלים להריץ את שני החוטים האלה. אז למשל הלוגיקה של שלבי Fetch/Decode: במכונת 4Wide Fetch/Decode יש ארבעה מפענחים זה אומרים שבכל מחזור היא יכולה לקבל ארבע פקודות ולעשות להן פענו - המפענחים הם כבדים מאוד ולא רוצים לשכפל אותם, אך יש תור שמחזיק בתים מתוך ה-CI, ובשיטה של פינג-פונג עושם fetch משני החוטים השונים לתוךthread לシリוגן כלומר במחזור אחד ארבעת המפענחים מוקצים ל-thread 0-0 ובסחרור הבא כל הארבעה מוקצים ל-thread 1. זה מבטיח הגינות מבחינה זו שאם למשל רק חוט 1 הוא אקטיבי, כי ה-queue של الآخر התרוקן ונדריך להביא את הפקודות שלו מה-DRAM או מה-L3 או מה-Instruction Cache-Decoders וה-ה-Renamer כי לכל חוט יש מרחב שמות שונה.

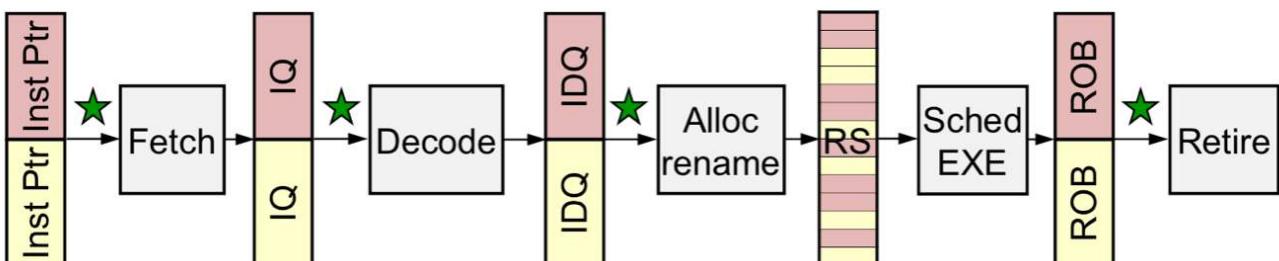
از ה-**Decoders** ו**Instruction Cache** עובדים ב”פינג-פונג”, כנ”ל ה-**Renamer**, כי לכל חוט יש מרחב שמות שונה.

### The RS is not a thread arbitration point

- The RS schedules pops to the execution units based on readiness and based on age (allocation time)
- Regardless of which thread they belong to

### The key arbitration points in the pipeline select in a given cycle which thread gets to

- Read the I\$ to Fetch instruction
- Use the Decodes for decoding instructions into pops
- Allocate pops to the ROB, store/load buffer, and the RS
- Retire pops and reclaim their resources



ה-RS, כמו scheduler, כבר לא עובד כך - בעקנון, לא אכפת לו מאייזה חוט מגיעה פקודה - הוא יכול להסתכל על כל הפקודות המוכנות לפי גילן, כמו המבוגרת מקבלת עדיפות יותר גבוהה, ופושט לחתת לביצוע פקודות שהמקורות שלהן מוכנים ויחידות הביצוע עבורן פנוiot. בפועל בגלל שלא רוצים שhot אחד ישתלט על כל ה-RS אז ימנע יש כמה כניסה ב-RS שהן ייעודו לכל חוט ועוד כמה שהן - shared fairness - אז כך מתחבطة ה-

SMT במנגנון ה-OOO. ייחדות הביצוע ה- Thread-Obliviousnas כלומר לא יודעת לאיזה חוט שייכת פקודה ורק מבצעות פעולות מוכנה - לא קשורות לשירות להחלטה של איזה מהחוטים לחתת וכו'.

כפי שכבר רأינו, לאורך הזמן יש נקודות בהן צריך לבחור בין אחד מהחוטים, ההחלטה בנק' הבחירה צריכה להיות בכל מחרוז - אם שני החוטים אקטיביים, כאמור יש פקודות והן מוכנות לביצוע (אך חוט לא-b-Miss) אז משתמשים באלג פינג-פונג' ובכל מחרוז עוברים מ-0-thread ל-1-thread ולהיפך. אם רק ב-thread יחיד יש עבודה אז מובן שהוא מקבל את כל המשאבם. הרעיון הוא להבטיח הגינות ופינג-פונג מבטיח לנו את זה. מצד שני לא רוצים לעשות חלוקה קשיה מדי עד כדי לא לתת משאבם כשייש רק חוט אחד אקטיבי.

נראה קצת יותר עמוק את ה-Thread Selection Points:

- דיברנו על ה-Instruction Cache: יש לנו שני pointers ל-Instruction Cache ולכן פעם אחת ערשים lookup באמצעות אחד מהם ופעם אחרת באמצעות השני.
- כנ"ל ב- Decode שמתרגם הוראות למיקרו-פקודות - אם אחד מה-IC ריק או לוקחים מהשני במובן זה שנותנים רק לו להשתמש ב-Allocator/Renamer.
- ה-RS הוא כבר משאב משותף שאינו מהו נקודת-בחירה. בשקף מסוימים בכוכבית המקומות בהם עובדים בשיטה של פינג-פונג.
- הפקודות שמוכנות לפרישה ב-ROB גם מופרדות לפי חוטים.

از אמרנו ש RS שלוח מיקרו-פקודות לביצוע על סמך המוכנות שלו (מקורות ייחדות ביצוע) ועל סמך גילן שנקבע ע"י הזמן בו הפקודות עושות Allocate. בפרט ה-RS יכול לשולח פקודות שני חוטים בלבד (בנוכח שהמקורות מוכנים ויש ייחדות ביצוע). דוג': אם פקודות כפל שניים יהיו פנויות וכן ייחידת הכפל תהיה פנווי אז תיבחר הפקודה הותיקה יותר ב-RS, ככלומר זאת זמן ה-Allocation שלה גדול יותר.

כמה עקרונות של SMT:

## SMT Principles

- When one thread is stalled, the other thread can continue to make progress**
  - Independent progress ensured by either
    - Partitioning buffering queues and limiting the number of entries each thread can use
    - Duplicating buffering queues
- A single active thread running on a processor with HT runs at the same speed as without HT**
  - Partitioned resources are recombined when only one thread is active

1. כשחוט אחד תקוע, למשל בגל cache Miss, אז החוט השני יכול להמשיך ולהתקדם. لكن למשל ב-RS יש מס' מקומות שמקצים לכל חוט בנפרד: לאחרת היה יכול להיזכר מצב שכל-hs-RS מלא בפקודות של חוט אחד שתקוע בעקבות אירוע מאד ארוך (כמו Cache-Miss interrupt) ובעקבות זאת החוט השני לא יוכל להתקדם - לא רוצים לאפשר מצב כזה, ככלומר תמיד צריך להבטיח Forward-Progress - שלא יהיה מצב שחווט כלשהו שתקוע משתכל על המכוונה ותוקע את החוט השני.

2. יש גם Independent-Progress, עקרון לפיו כל חוט יכול להתקדם בלי תלות בשני - השני יכול לתפוס לו משאבים ולכן להאט אותו אבל הוא לא יכול לעצור אותו. עושים את זה ע"י חלוקת חלק מהחוטים במעבד חלוקה קשיה, נגיד חצי-חצי ואז חוט יכול להשתכל רק על מס' כניסה מוגבל. שתי הדרכים לחלוקת משאב

הן ה-**RS** הולך לפי העיקרון של **Limit Hard-Partition** וונראה גם באפרים אחרים שהם משוכפלים ממש (כלומר **Hard-Partitioned**). בכל מקרה, אם יש לנו רק חוט אחד פעיל שרצ על המכונה אז גם אם משאב הוא ב-**Hard Partition** עדין חוט בודד יוכל את מלא הביצועים שלו.

בஹש נראה איך המכונה עוברת מ מצב של **Single-Thread** ל-**Multi-Thread** ולהיפך: יכול להיות שלמכונה בעלת ארבע ליבות יש רק ארבעה חוטים בכל התכניות שלה - במצב זה כל חוט יכול לרווח על מכונה בלבד, לקבל את כל הביצועים שלה ולרווח בביטויים המקסימליים שלו.

## Physical Resource Sharing Schemes

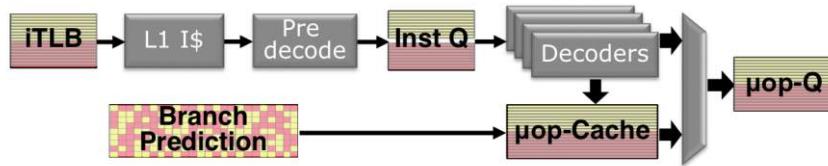
- **Replicated Resources:** e.g. CR3
  - Each thread has its own resource
- **Partitioned Resources:** e.g. uop-Cache
  - Resource is partitioned in MT mode, and combined in ST mode
- **Competitively-shared resource:** e.g. RS entries
  - Both threads compete for the entire resource
- **HT unaware resources:** e.g. execution unit
  - Both threads use the resource
- **Alternating Resources**
  - Alternate between active threads
  - if one thread idle
    - the active thread uses the resource continuously



- עכשו נסתכל יותר לעומק על כל משאב במעבד OOO ונראה איך הם מתחלקים בין שני החוטים אבל לפני כן נדון באופן עקרוני בסוגי החלוקת השונים של משאב:
- **שכפול (Replicated Resources):** משאים משוכפלים מהווים חלק מההקשר של כל חוט, למשל **Instruction Pointer** או הרגיסטר **LCR3** (מצבייע לטבלאות הדפים) או כל אחד מהרגיסטרים הארכיטקטוניים - הם משוכפלים לגמרי וכמובן שהוא תורם ל-7% תקורה שדיברנו עליה.
  - **חלוקת (Partitioned Resources):** אופצייה שנייה היא חלוקה קשיחה של משאב יחיד, לדוגמה ה-**uop-Cache** של כל ליבה הוא בגודל 1024 ו-512 יוקטו לכל אחד משני החוטים (כמובן שבמצב **Single-Thread** הכל יוקצה לחוט היחיד). כמו ה-**ROB** גם ה-**uop-Cache** הוא בחלוקת קשיחה והכן ה-**load/store buffers** מתחלקים חצי-חצי כדי להבטיח Fairness.
  - **סוג שלישי של שיתוף של משאב נקרא Competitive-Sharing** שאומר שני החוטים מתחברים אל המשאב ביחד - נתנו דוג' לcker את **RS**: כל חוט שיכל להיכנס לשם רשייא להיכנס - עד כדי כמה מקומות בודדים שמתביחסים Forward-Progress.
  - **יש משאים שהם Thread-Oblivious** כלומר אינם מודעים כלל לכך שהמעבד במצב של Multithreading. דוג' לcker היא ייחidot הביצוע: לא נדרש ליחידת ביצוע כלשהו אם פקדה נשלחת לחוט-0 או לחוט-1.
  - **חלוקת משאים נספפת היא בשיטת Alternating-Resources**: Alternating-Resources, Decoder, המפענה, הוא משאב שהוא מוחולק בצורה משתנה (Alternating) כלומר מוחזר אחד לחוט-0 ומוחזר הבא לחוט-1. כמובן שגם חוט אחד הוא idle אז החוט האקטיבי מקבל את המשאב בכל מוחזר שעון.

דוג': ה-*h*-cache הוא Partitioned Cache כי זה משאב יחסית-קטן: משאב גדול, כמו ה-*Instruction Cache*, להכיל הרבה שורות ולא רוצים לגרום לכך חלק אווות בזיכרון קשירה כי אז כל חוט יוכל לקבל לכל היותר חצי וזה סותר את המקובל אם למשל לחוט 2 יש המונע פקדות לביצוע ולהוט 1 יש תכנית קצרה עם מעט פקדות, למשל לולאה קטנה שתופסת רק קצת מה-*Instruction Cache*. לעומת זאת רוצים ליצור חלוקה קשירה על משאב קטן כדי למנוע מצב של חוט אחד לא יהיה מספיק ולשני כן - עושים שם איזון, balancing, כדי לשמר על הוגנות. אז לא כל משאב רוצים לעשות כמו ה-*instruction cache*, למשל ה-*mop cache*, יש בו מקום רק לכ-750 Ops וסביר שנחלייט שכל חוט יקבל בו 375 מקומות כי אין הרבה מוצבים בהם חוט צריך פחות מזה לעומת זאת במשאב גדול אפשר לאפשר חלוקה בגודל משתנה וудין כל חוט יכול מספיק מהמשאב.

## Front End Resource Sharing



- The Instruction Pointer is replicated
- The small page iTLB is partitioned
- Branch prediction resources are mostly shared
- The I\$ is thread unaware (thus competitively-shared)
- The pre-decode logic and the instruction decoder logic are used by one thread at a time
- The uop-Cache is partitioned
- The uop Queue is partitioned

אפשר לראות איזה משאבים ב-*Partitioned Front End*, איזה Shared וכן הלאה:

- Shared L1 הוא L1
- ה-iTLB הוא Hard-Partitioned iTLB חצי-חצי וכן ה-*Instruction Queue*, *uop-Cache*, *uop-Queue* וה-*Instruction Pointer*.
- ה-*Instruction Pointer* משוכפל וה-iTLB הוא חצי-חצי.
- נשים לב שהזאי הקפיצות הוא משותף (Shared) וכל חוט בו יכול להשלט על כמה כניסה שהוא צריך - יכול להיות שבגלל זה חוט אחד יפריע לחוט שני אבל יכול להיות שני חוטים גם יתרמו אחד לשני: חלק מההיסטוריה ובנחתה ב-BP יכולות להיות משותפות וחוט כלשהו יכול להרוויח מזה שחוט שני שכבר יצר אוiso היסטוריה ובלחץ שהקפיצות שלו מתנהגות דומה לקפיצות של השני. אז בהזאי הקפיצות לחלק מהדברים יש Thread-bit ולחלקם אין - מכור החזאי לא מסכן את הנכונות כי הכל בו נעשה באופן ספוקטיבי, אלא יכול במקרה הגרוע לא לקדם את הביצועים. נחזור על זה: החזאי הוא משותף, וזה יכול לצור כמה מצבים:
  1. כניסה משותפת כר שחוט אחד יתפס לשני את הכניסות ואז למשל יזרוק לשני את ההיסטוריה ולכן יבצע בביטויים טובים יותר ממנו - אז להזאי הקפיצות יכולות להיות תופעות שאחד פוגע בשני
  2. מצד שני יכול להיות שחוט אחד יבנה ההיסטוריה לשני וישתמש בהיסטוריה שלו.

ב-BP עושים flush לכל חוט עצמו - שתי המוטיבציות הći גדולות ל-*Multithreading* זה ה-*Branch Miss* predictions וה-*Instruction/Data Cache misses*: כתעת כשחוט אחד חווה flush השני יכול להתקדם ולתפוס את החורדים שנוצרו. נשים לב שב-*Event Switch On* הרוח הזה קטן רק בזמןם בהם חוט אחד נתקע לחולוטין

ובגלל זה הוא פחות טוב - לעומת זאת ב-SMT מוצלים כל מחזר שימושו תקוע וכל משאב פנוי - אם אין מה לעשות ביחיד כלשהו במעבד עבור חוט אחד נשתמש בה עבור החוט השני, ב-SOE אין דבר כזה כי מוחכים לקטסטרופה גדולה ורק אז מעבירים את הזכות על כל המשאים אחד לחוט השני - פה אפשר ליהנות מכל המשאים בכל עת ע"י חלוקתם בכל מחזר בין שני החוטים.

## Back End Resource Sharing

- **Out-Of-Order Execution**

- Register renaming tables are replicated
- The reservation stations are competitively-shared
- Execution units are HT unaware
- The re-order buffers are partitioned
- Retirement logic alternates between threads

- **Memory**

- The Load buffers and store buffers are partitioned
- The DTLB and STLB are competitively-shared
- The cache hierarchy and fill buffers are competitively-shared

עד עכשיו דיברנו על ה-Front-End. ועכשיו נדון במשאבי ה-Back-End.

• נתחיל מה-Units OOO: כל מה הקשור ל-Renaming חייב להיות משוכפל כי יש לכל חוט רגיסטרים משלו - זה חלק מה-7% RS. כאמור הוא Shared ויחידות-הביצוע הן לא-מודעות להבדיל בין החוטים. ה-ROB מחולק חצי-חצי וה-Retirement באתרי פינג-פונג - ככלומר בכל מחזר אורחים Retire לפקדות מחוט אחר.

• מבחינת הזיכרון: ה-STLB וה-dTLB הם Hardly-Partitioned Load/Store Buffers. Shared.

עכשו עומרים לדבר על מעורבת מ"ה ואיך היא מעבירה את המעבד ל/מצב של ריבוי-חוטים:

## Single-task And Multi-task Modes

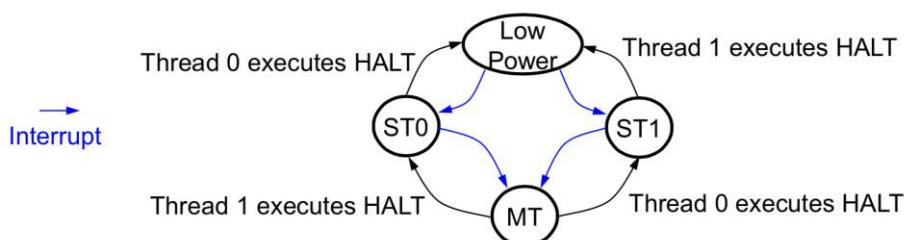
- **MT-mode (Multi-task mode)**

- Two active threads, with some resources partitioned as described earlier

- **ST-mode (Single-task mode)**

- ST0 / ST1 – only thread 0 / 1 is active
- Resources that are partitioned in MT-mode are re-combined to give the single active logical processor use of all of the resources

- **Moving the processor from between modes**



אפשר לעבור בין Hardly Multithread Mode ל Single-Thread Mode : מבחינת החומרה המשאבים שהם-Partitioned ב-MT הופכים זמינים למאריך לחוט היחיד ב-ST. איך זה קורה? מ"ה שלוחת למעבד Interrupt ככל שחייב רצחה לעבור בין המצבים, וה-Power Gauge מחזואה את mode הנוכחי - נניח שפעם היו שלושה חוטים, ככלומר ברגע כלשהו יש לשולח ליבוט פנווות, ואז הם התפצלו לשישה. איך מ"ה תחלק את העומס בין הליבות? ראשית היא תשלח חוט רביעים לביצוע על הליבה הפיזית הפנווות ואז תשלח למעבד זה פסיקה כרשה המעבד יתחל לזרע ב-Single-Thread. כשהיא תרצה לשלוח עוד חוט, החמישי, כל הליבות כבר יהיו עסוקות וכך יוכל היהת לשלוח אותו לLIBA-LOGIT, ואז שוב תשלח מ"ה מעבד כר שהוא יעבור ל-MT וירוץ בין חוטיהם. נניח עכשו שחווט כלשהו סיים (והשני נשאר לרוץ על הליבה) - אז מ"ה תעביר את המכוונה שלושוב ל-ST ואם גם הוא יעשה אז חזרה low-power mode שם אפשר לעשות gating-power 0-0 ולבכבות את הליבה למאריך.

## Thread Optimization

**The OS should implement two optimizations:**

- **Use HALT if only one logical processor is active**

- Allows the processor to transition to either the ST0 or ST1 mode
- Otherwise the OS would execute on the idle logical processor a sequence of instructions that repeatedly checks for work to do
- This so-called “idle loop” can consume significant execution resources that could otherwise be used by the other active logical processor

- **On a multi-processor system**

- OS views logical processors similar to physical processors
  - But can still differentiate and prefer to schedule a thread on a new physical processor rather than on a 2nd logical processor in the same phy processor
- Schedule threads to logical processors on different physical processors before scheduling multiple threads to the same physical processor
- Allows SW threads to use different physical resources when possible

יש שני אופטימיזציות חשובות שכל מ"ה צריכה לעשות כשהיא מריצה תוכניות במקבץ SMT:

1. השימוש בפקודת halt בתכנית הוא חשוב בשביב להבahir שרק מעבד-לוגי אחד צריך להיות אקטיבי, אחרת,

אם מ"ה צריכה לבצע כל הזמן לבדוק האם יש לחוט פקדת מוכנה, ככלומר חלק מהמעבד מוקצת idle-loop זהה סתם בזבוז. לכן צריך מערכת הפעלה צריכה למשתמש בפקודת halt כאשר לה משזו להריץ.

2. במערכות עם מס' מעבדים שונים כל חוט לLIBA-LOGIT שונה לפני שלוחים חוט-נוסך לLIBA-LOGIT, ככלומר

לפני שני חוטים ירצו ביחד כל LIBA-LOGIT חוט אחד. במערכת מרובת-מעבדים מ"ה יודעת על הקיום של LIBA-LOGIT וגם יודעת האם הליבות תומכות ב-threading, זה שהן תומכות אומר שכל LIBA-LOGIT צואת

מבחינת מ"ה היא שתி LIBA-LOGIT ולמ"ה בתורה יש חוטים שונים לביצוע, אם יש לה חוטים כמו' הLIBOT ה-LOGIT או ה-LOGIT-Aligned, למשל אם יש על המעבד ארבע LIBOT ויש פחות ארבע חוטים שצרכיהם ביצוע אז היא תפעיל רק LIBOT פיזיות ברגע שייהיו לה יותר חוטים היא תתחל ל-הפעיל גם LIBA-LOGIT.

הערה: ככליבות פיזיות מתפנות אם יש LIBA-LOGIT שני חוטים או תיתכן העברת של אחד מהשניים אל LIBA-LOGIT, זה נקרא Migration וזה יקרה רק שיגמר הזמן שמ"ה הקצתה לחוטים לרוץ (זמן אופייני הוא

ירוץ LIBA-LOGIT ms50) ואז השיטה חזרת-L-Scheduler של מ"ה (לא קשור ל-Scheduler של ה-LOGIT) ופעם הבאה שהוא

ירוץ LIBA-LOGIT אותו תשלח LIBA-LOGIT פיזיות פנווות פנווות.

הערה נוספת: בהחלט יכול לקרות בימיינו מצב שאין מה להריץ, הראיינו את זה בהרצאה הקודמת כשדיברנו על

חלוקת ההספק על פני המעבד ובפרט על ה-Power Gate שמנתק ליבوت ריקות מהמתח ומריצ ST על ליבות אחרות. זה כמעט ולא קורה בשרתים ולכן שם שמיים 32 ליבות ולא ארבע כי תמיד יש עבודה - במקרה של שרתיים מה שחשוב זה throughput ורק אם מתבצע רק ב-0.70% בהשוואה ל-ST זה פחות קרייתי אלא נהנים מ-130% ביצועים בהשוואה לחוט יחיד

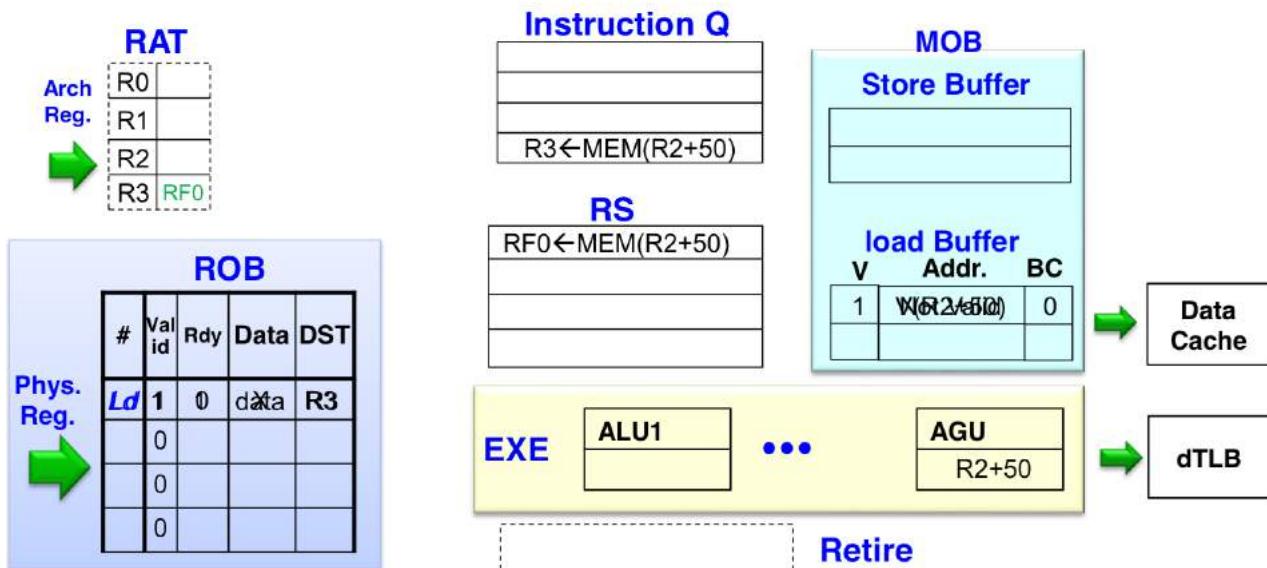
סיימנו את החומר של הקורס!

## תרגול מס' 11: ביצוע גישות-זיכרון שלא על-פי הסדר.

Out Of Order Memory Execution

שלבי הביצוע של פקודה load במכונת OOO

## The life of a Load...

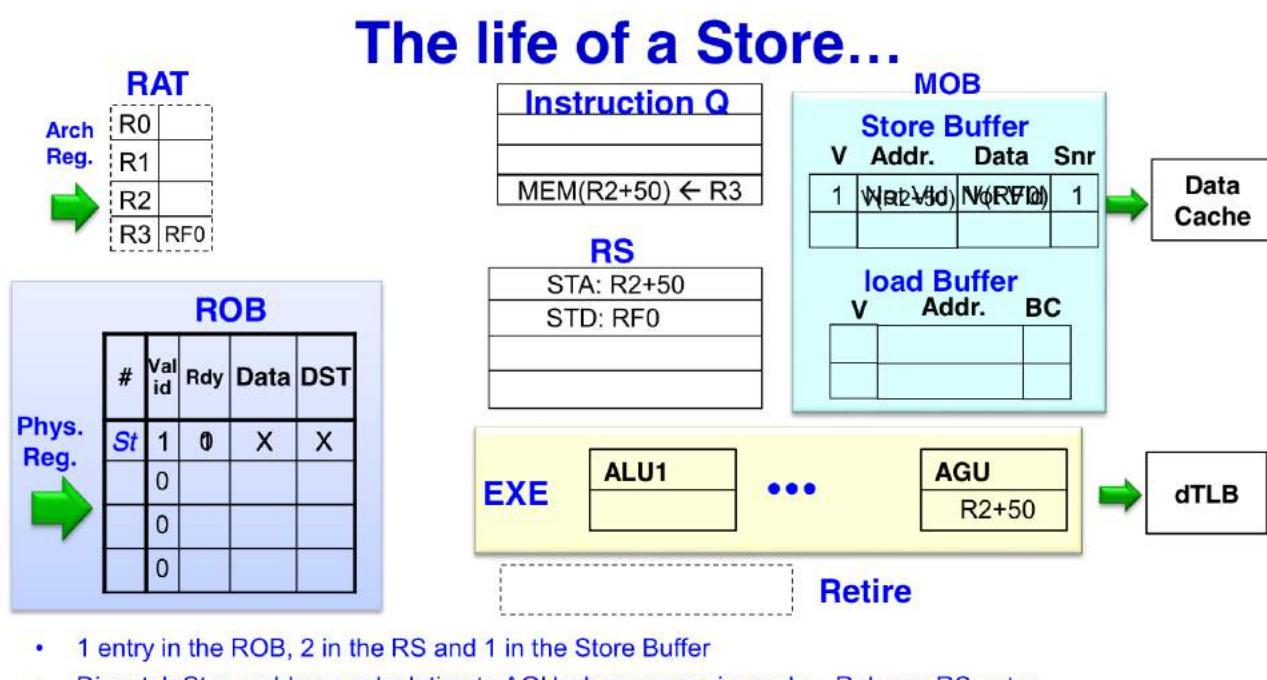


- 1 entry in the ROB, RS and Load Buffer + rename in RAT
- Dispatch Load address calculation to AGU when source is ready – Release RS entry
- AGU updates the address in the Load buffer. Pipeline proceeds to dTLB
- Load Buffer checks for blocking conditions and dispatches the Load to the DCU
- DCU sends the result to RS and updates the ROB with the load result
- Load will retire as any other instruction (when all previous instructions have retired) – RAT updated
- LB and ROB entry are released

- אלוקציה ל-load נעשית כפי שנעשה לכל פקודה (מקום ייחד ב ROB,RS וביצוע renaming) עם התוספת שמקצים ל-load ב-MOB גם LBID ו-load-buffer (מס' מזהה ל-load).
- ברגע שהמקורות מוכנים שולחים את ה-load ל-AGU, שם מחשבים את כתובת הטיענה.
- לאחר חישוב הכתובת ע"י ה-AGU מעדכנים את הכתובת ב-load buffer ב-MOB. עד כה קיבלנו את הכתובת הווירטואלית, אך הולכים ל-TLB data והוא מתרגם את הכתובת מווירטואלית לפיזית - קלומר מספק לנו את כתובת הטיענה ממש.
- ברגע שיש לנו את הכתובת ה-MOB יכול לבדוק האם אפשר כבר לבצע את ה-load load מבחינת תלויות: התפקיד של ה-MOB זה להסתכל על כל store לפני אותו load (כלומר על כל store-buffer עם SBID נמוך/שווה ל-LBID) של ה-MOB load-shoka (load-buffer) ואם יש תנאי חסימה אז ה-loadחסום ואי אפשר להכניס אותו לביצוע OOO. ישנו שני סוגי של תנאי חסימה: תנאי-חסימה מס' 1: קיימים storestore לשועוד לא יודעים את הכתובת שלו - במצב זה אומרים שה-load load חסום על כתובת". תנאי-חסימה מס' 2: קיימים store לפני ה-load load שידוע שהוא כותב אותה כתובת ממנו load load רוצה לקרוא, אלא שה-data data כותב עדין לא ידוע - במקרה זה ה-load load load-forwarding על data".

- לעתים זה נעשה באותו מוחזור בו הם מוכנים ולעתים במחזור העוקב - תלוי בארכ'/בתרגיל.
- ברגע שהבאנו את הנתון מה-cache מדכנים את ה-ROB עם המידע שהולך להיכתב לרגיסטר (כמו עבור כל פקודה), כמו כן מדכנים את המקורות של כל מי שמחכה לרגיסטר זה ב-RS).
  - משחררים את ה-load כמו כל פקודה - קלומר משנים את מצב המכונה לפי הסדר (in-order). כמו כן מדכנים את ה-RAT שהמיופיע כבר לא רלוונטי.
  - מוציאים את ה-load מה-MOB, קלומר משחררים את ה-load-buffer עבورو, וכן הכניסה שלו משוחררת מה-ROB.

### שלבי הביצוע של פקודה store במכונת OOO



- 1 entry in the ROB, 2 in the RS and 1 in the Store Buffer
- Dispatch Store address calculation to AGU when source is ready – Release RS entry
- AGU updates the address in the Store buffer → update the Store Buffer & provide addr. to depending loads
- Store pipeline proceeds to dTLB. Physical address will be updated in the SB
- Dispatch Store Data when Data is ready → update the Store Buffer & provide data to depending loads
- The Store Buffer updates the ROB entry
- The Store will retire from the ROB as any other instruction (when all previous instructions have retired)
- After this, the Store is marked as **Senior Store** in the Store Buffer
- The Store buffer will initiate a DCU write. When the write is done, the SB reclaims the entry

- אלוקציה: פקודה store דורשת כניסה אחת ב-ROB ושתי כניסה ב-RS: כבר בשלב ההקצאה ה **store-mתפצל** לשתי פקודות: STA לחישוב כתובות הטיעינה ו-STD לחישוב הנתון, שקל אחת יכולה להיכנס לביצוע מתי שתהיה מוכנה - لكن שתי פקודות מוקצחות ב-RS. למרות ש-store מפוצלת לשתי mikro-פקודות, ב-ROB מוקזית פקודה עבור store שורה אחת בלבד (כי הן עוסות commit ביחד) - קלומר store ונכנס למוכנה כפקודה אחת ויוצאת ממנה כפקודה אחת. כמו כן לכל store מוקצת store Buffer ב-MOB.
- פקודה ה-STA עוברת ל-AGU שמחשב את הכתובת אליה יבוצע ה-store.

- מעדכנים את ה-Store Buffer ב-MOB עם כתובת השמירה - והוא בודק אם load כלשהו חיכה לה (ולכן השחרר) - אם כן מעיר אותו. כמו כן load שיוכנו משלב זה יתחשב ב-store זה בבדיקה התלויות שלהם.
- שולחים את ה-store ל-TLB לקבالت הכתובת הפיזית אליה שומרים.
- ברגע שהוא-STD יכול להיכנס לביצוע שלוחים אותו ישירות ל-buffer store ומעתיקים לשם את ה-data שחויב.
- גם זה משמש לבדיקת התלויות של loads ב-MOB באותו store.
- מעדכנים את ה-ROB עם הערכיהם שה-store שומר (כלומר מה שworshפ ע"י פקודת ה-STD).
- בסוף עושים Retire כמו לכל פקודה.
- אחרי Retire מסמנים ב-MOB את ה-store כ-Senior Store-Buffer ובפרט עד לא מוצאים אותו מה-Store-Buffer ב-MOB.
- הסיבה לשימונם ה-Store-Buffer ב-MOB היא Data Cache Senior : במעבד כל הזמן בשימוש - יש עליו הרבה מאוד לחץ ובדרכ"כ יש לו רק פורט אחד לכטיבה - لكن ה-store ייחה עד שהמתਮן יהיה פניו לכטיבה (יתפוס את פורט הכתובת), יעדכן את המתמן ורק אז יהיה אפשר לפנות את ה-store buffer שלו ב-MOB.

### שאלת מבחן

עכשו נפתרו שאלה שניית בבחן כבר שלוש או ארבע שנים ברצף - להלן הקוד של תכנית כלשהי:

|      |       |             |                       |
|------|-------|-------------|-----------------------|
| 1000 | load  | R2,R1,30    | ; R2=m[R1+30]         |
| 1004 | store | R2,20,R1    | ; m[R2+20]=R1         |
| 1008 | load  | R3,R1,100   | ; R3=m[R1+100]        |
| 100C | store | R1,40,R3    | ; m[R1+40]=R3         |
| 1010 | add   | R1,R1,10    | ; R1=R1+10            |
| 1014 | blt   | R1,100,1000 | ; if (R1<100) PC=1000 |

כל המס' בתכנית ניתנים בסיסי הקסדצימלי. לשם פשוטות נניח שככל הכתובות הן פיזיות.

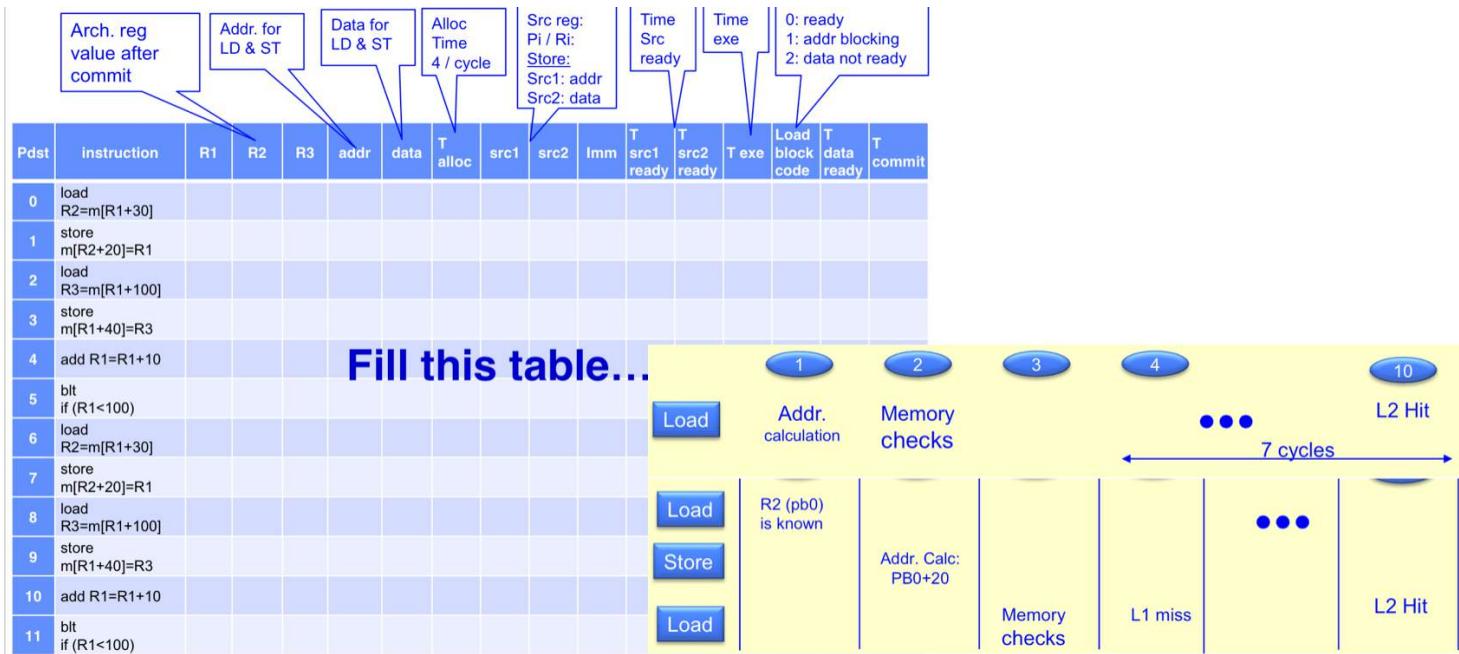
הנחיות:

- כל הרגיסטרים, בפרט R1,R2,R3, מאותחלים ל-10.
- בכל כתובת זיכרון N שמור הערך N.
- יש במעבד L1 Data Cache שנוטן את המידע תוך שבעה מהזורים החל מרגע הגישה ל-L1 (כלומר שבעה מהזורים כולל זמן ההחטאה ב-L1). הערה: תמיד כדאי לשים לב מהי גודל שורת הקאש ב-L1 כי גודל זה יעזר לנו לדעת האם יש לנו hit או miss בגישות הבאות לזכרון.
- עשויים אלוקציה לארבע פקודות בכל מהזור-שעון.
- אופן פעולה load:
  - במחזור הראשון ה-load מחשב את הכתובת.
  - במחזור השני עושים בדיקת זיכרון על תנאי החסימה של ה-load.
- אם אין תנאי חסימה: ניגשים ל-L1 - אם יש בו hit קיבל מיד את המידע ואם נחתיא אין נספר שבעה מהזורים L1-.

הגיישה

כשיש תנאי חסימה: בודקים בכל מחזור את התנאי (מחכים עד שהוא ישחרר) ובמחזור אחריו כן אפשר לגשת ל-cache.

- הינה יותר פשוט load-m-store באופן זה שאף פעם אין לו תנאי חסימה: במחזור הראשון מחשבים את כתובות הטעינה באמצעות STA ובמחזור הבא בודקים האם המידע שלו מוכן לשמירה - אם כן אז במחזור הבא המידע כתיבת מועתק ל MOB-store-buffer ב-in-order stores. כמו כן stores תמיד מבוצעים תחילה.



בשאלה צריך למלא את הטבלה לעיל בעזרת הנתונים על התכנית ועל המערכת.

השלב הראשון למילוי הטבלה הוא למלא רק את השדות עד העמודה T-alloc. חבל לא קיבל את הנקודות כאן - פשוט צריך להריץ את הקוד ולראות מהם ערכי הרגיסטרים/כתובות בסוף כל פקודה. התוצאה (МОך במסגרתabadom):

The table below provides the detailed timing and register values for the assembly code. The first 11 rows are highlighted with a red border.

| Pdst | instruction       | R1 | R2  | R3  | addr | data | T alloc | src1 | src2 | imm | T src1 ready | T src2 ready | T exe             | Load block code | T data ready | T commit |    |
|------|-------------------|----|-----|-----|------|------|---------|------|------|-----|--------------|--------------|-------------------|-----------------|--------------|----------|----|
| 0    | load R2=m[R1+30]  | 10 | 40  | 10  | 40   | 40   | 1       | R1   |      | 30  | 1            |              | 2                 | 0               | 11           | 12       |    |
| 1    | store m[R2+20]=R1 |    |     |     | 60   | 10   | 1       | P0   | R1   | 20  | 11           | 1            | Std: 2<br>Sta: 12 |                 |              | 12       | 13 |
| 2    | load R3=m[R1+100] |    |     | 110 | 110  | 110  | 1       | R1   |      | 100 | 1            |              | 2                 | 1               | 21           | 22       |    |
| 3    | store m[R1+40]=R3 |    |     |     | 50   | 110  | 1       | R1   | P2   | 40  | 1            | 21           | Std: 22<br>Sta: 2 |                 |              | 22       | 23 |
| 4    | add R1=R1+10      | 20 |     |     |      |      | 2       | R1   |      | 10  | 2            |              | 3                 |                 |              |          | 23 |
| 5    | bit if (R1<100)   | 20 | 40  | 110 |      |      | 2       | P4   |      | 100 | 3            |              | 4                 |                 |              |          | 23 |
| 6    | load R2=m[R1+30]  |    | 110 |     | 50   | 110  | 2       | P4   |      | 30  | 3            |              | 4                 | 1, 2            | 24           | 25       |    |
| 7    | store m[R2+20]=R1 |    |     |     | 130  | 20   | 2       | P6   | P4   | 20  | 24           | 3            | Std: 4<br>Sta: 25 |                 |              | 25       | 26 |
| 8    | load R3=m[R1+100] |    |     |     | 120  | 120  |         |      |      |     |              |              |                   |                 |              |          |    |
| 9    | store m[R1+40]=R3 |    |     |     |      | 60   |         |      |      |     |              |              |                   |                 |              |          |    |
| 10   | add R1=R1+10      | 30 |     |     |      |      |         |      |      |     |              |              |                   |                 |              |          |    |
| 11   | bit if (R1<100)   | 30 | 110 | 120 |      |      |         |      |      |     |              |              |                   |                 |              |          |    |

כשיש קור-תחthon מתחת למס' כלשהו זה אומר שבמחוזר זה משנים את ערך הרגיסטר המתאים - למשל בשורה הראשונה (מחוזר 0) הרגיסטרים colum 10 ולכן כתובותם אל R2 ערך מכתובות R1+30 בפועל טוונים מכתובות 40, ומכיון שנותן בשאלת שבקותה N שמור הערך N אז  $R2=40$  בסוף המוחזר-0.

נעביר למלא את שאר השדות לפי סדר הפקודות, ככלומר החל מהפקודה הראשונה בשורה 0:

### פקודה 0 – load R2=m[R1+30]

| Pdst | instruction         | R1 | R2        | R3 | addr | data | T alloc | src1 | ✓ src2 | Imm | T src1 ready | T src2 ready | T exe | ✓ Load block code | T data ready | T commit |
|------|---------------------|----|-----------|----|------|------|---------|------|--------|-----|--------------|--------------|-------|-------------------|--------------|----------|
| 0    | load<br>R2=m[R1+30] | 10 | <u>40</u> | 10 | 40   | 40   | 1       | R1   |        | 30  | 1            |              | 2     | 0                 | 11           | 12       |

- T alloc: אלוקציה של פקודה 0: מתחילה לשות אלוקציה כבר במחוזר הראשון, ככלומר מוחזר מס' 1 (שם מבאים בסה"כ ארבע פקודות).
- src1: המקור היחיד בפקודה זו הוא R1.
- imm: הערך המיידי שלה הוא 30.
- T src1 ready: מתי המקור הראשון מוכן? מכיוון שגם הפקודה הראשונה היעד הוא רגיסטר AR[כ' ולא מחכים לאף פקודה אחרת שבילו, הוא מוכן מיד, ככלומר במחוזר אחרי האлокציה: 1.
- T exe: הפקודה נשלחת לביצוע במחוזר אחורי שהמקורות מוכנים - מוחזר 2.
- Load block code: אין לפניה שום store ולכן אין חסימה - כותבים 0.
- T data ready: מתי הגיע אליו ה-data? בפקודת load ניגשים אל המטען. ב-L1 יהיה לנוטיס (כי הוא ריק בתחילת הריצה) ונתון שב-L2 תמיד יש hit - חשוב מאד להבין בדיק באיזה מוחזור הגיע המידע: מחשבים את הכתובת במחוזר 2, במחוזר 3 רואים שאון תנאי חסימה, במחוזר 4 מקבלים החטהה במתמון L1 וממחזר 4 (כולל) לוקחים שבעה מוחזרים להביא את המידע - אך הוא מגיע במחוזר 10 (ready) וזמן החל ממחוזר 11.
- T commit: הפקודה עושה מוחזרים להביא את המידע - אוטומטית מוחזר 12. כלומר במחוזר-12.

### פקודה 1 – store m[R2+20]=R1

| Pdst | instruction          | R1 | R2 | R3 | addr | data | T alloc | src1 | ✓ src2 | Imm | T src1 ready | T src2 ready | T exe | ✓ Load block code | T data ready | T commit |    |
|------|----------------------|----|----|----|------|------|---------|------|--------|-----|--------------|--------------|-------|-------------------|--------------|----------|----|
| 1    | store<br>m[R2+20]=R1 |    |    |    |      | 60   | 10      | 1    | P0     | R1  | 20           | 11           | 1     | Std: 2<br>Sta: 12 |              | 12       | 13 |

- הפקודה הבאה היא פקודת store ויש לה שני מקורות - הראשון (src1) משמש לחישוב הכתובת והשני (src2) לחישוב המידע.
- האлокציה נעשית במחוזר 1 - אותו זמן של הקצאת הפקודה הראשונה כי עושים אלוקציה לארבע פקודות בכל מוחזר.
  - המקור הראשון הוא הרגיסטר הפיזי שאליו ממוקפה 2 ע"י הפקודה הקודמת - בשאלת מיניכים שכל פקודה שמורת ערך אל הרגיסטר הפיזי שמספרו כמו' השורה שלה - כך זה גם נעשה ב-ROB אמיתי. ככלומר הפקודה בשורה 1 מחכה לרגיסטר הפיזי P0 (זה שהפקודה בשורה 0 כותבת אליו את התוצאה שלה).
  - המקור השני הוא R1, שביל חישוב ה-data שנכתב.

- המקור הראשון מוכן במחזור 0 לאחר שפקודה 0 תחשב את הערך שלה - היא מחשבת אותו במחזור 11 ולכן המkor הראשון יהיה מוכן במחזור 12 (נשים לב שיש כאן **forwarding**, פקודת 0 לא עשתה commit לפני מחזור 12).
- המקור השני מוכן עם הקצאת הפקודת - ככלומר במחזור 1.
- הערך המיידי שמופיע בפקודה הוא 20.
- זמן הביצוע: מכיוון שכל store מפוץל לשתי מיקרו פקודות יהיו שני זמני-ביצוע: ל- STA ו-STD. לכן תמיד נציין שני מס' בתא exe T של פקודת store. STA צריך את הערך של R1 ולכן יכול להיכנס לביצוע כבר במחזור 2. STA צריך את הערך שגיעה במחזור 11 - לכן מתחליל את ביצועו במחזור 12 (כאמור יש כאן forwarding).
- מי זהו אותו מחזור בו המוקר מוכן).
- אם יש תנאי-חסימה? זהו store ואף פעם אין עבורו תנאי חסימה! לכן לא מלאים כלום.
- ה-data של הפקודת, בעצם הערך שיישמר ב-R1, מוכן במחזור בו מסתיים החישוב, ככלומר במחזור 12. כמו כן במחזור 2 לוקחים את המידע שמקורן ומעבירים אותו ל-store buffer - זה לא היה אפשרי במחזור 1 כי במחזור זה מעתיקים את המידע על ה-store ל-RS (שלב ההקצאה).
- אפשר לעשות commit לפקודת store במחזור מס' 13 - מחזור אחריו סיום החישוב של STA (STD סיום מוקדם יותר).

### פקודה 2 load: R3=m[R1+100]

| Pdst | instruction          | R1 | R2 | R3         | addr | data       | T alloc | src1 | src2 | Imm | T src1 ready | T src2 ready | T exe | Load block code | T data ready | T commit |
|------|----------------------|----|----|------------|------|------------|---------|------|------|-----|--------------|--------------|-------|-----------------|--------------|----------|
| 2    | load<br>R3=m[R1+100] |    |    | <u>110</u> | 110  | <u>110</u> | 1       | R1   |      | 100 | 1            |              | 2     | 1               | 21           | 22       |

- מכיוון שמקצים ארבע פקודות בכל מחזור גם כאן T alloc הוא 1.
  - לפקודה יש רק מקור אחד - רגיסטר R1 שנדרש בשבייל חישוב הכתובת.
  - המקור מוכן כבר מהמחזור הראשון ולכן זמן תחילת הביצוע הוא 2.
  - הפעם יש תנאי חסימה על ה-load: נסתכל על כל ה-store שלפנינו - במחזור מס' 1 יש store שלא יודעים את הכתובת שלו - לכן זהו תנאי חסימה מסווג 1 (תקועים על הכתובת).
  - מתי ה-data יהיה מוכן? תנאי החסימה הוא עד מחזור מס' 12 - רק אז מסתיימים פקודות STA. עד אז בכל מחזור בודקים את תנאי החסימה ורואים שהוא-loadeadily חסום - כולל במחזור מס' 12. רק במחזור 13 המעבד בודק שוב ומבחן WHETHER החסימה התבטלה - ולכן במחזור הבא הוא ניגש ל-L1. אם קיבל Miss או hit ב-L1? נשים לב:
    - גודל הבלוק בקאס הוא 80 (הקסדצימלי).
    - עשינו load לכתובת 40
  - ה-store הוא Write no-allocate ולכן לא הבנו כלום למטען.
  - אנחנו רוצים לקרוא מכתובת 110.
  - לכן קיבל cache Miss (מכיוון ש cache Miss (מכיוון ש  $C0 < 110$ ) Hexadecimal =  $(40+80)$  Hexadecimal).
- از כאמור ה-store מחשב את הכתובת ("יעי") STA במחזור 12, במחזור 13 אנחנו מתעדכנים שהכתובת מוכנה, במחזור 14 ניגשים ב-L1 ומקבלים Miss והחל ממחזור זה (כולל) סופרים שבעה מחזוריים ומקבלים את המידע בסוף מחזור 20 - המידע יהיה מוכן במחזור 21.
- אפשר לסיים את הפקודה במחזור העוקב - ככלומר במחזור 22.

פקודה 3 - store: m[R1+40]=R3

| Pdst | instruction          | R1 | R2 | R3 | addr | data | T alloc | src1 | src2 | Imm | T src1 ready | T src2 ready | T exe            | Load block code | T data ready | T commit |
|------|----------------------|----|----|----|------|------|---------|------|------|-----|--------------|--------------|------------------|-----------------|--------------|----------|
| 3    | store<br>m[R1+40]=R3 |    |    |    | 50   | 110  | 1       | R1   | P2   | 40  | 1            | 21           | Std:22<br>Sta: 2 |                 | 22           | 23       |

- זמן האлокציה הוא כרגע 1 (מקצים ארבע פקודות בכל מחזור).
- המקור הראשון של הפקודה הוא R1 (לא עשינו אליו store لكن לא מחכים לאפ' רגיסטר פיזי).
- המקור השני הוא ערכו של R3 כפי שנקבע ע"י ה-store בשורה מס' 2 - לכן הממקור הוא הרגיסטר הפיזי P2.
- המקור השלישי, R1, מוכן החל מתחלת הריצה והמקור השני מרגע קבלת המידע בפקודה 2 - 21. נשים לב שמכיוון שעשינו forwarding לא מחכים עד ל-commit ואפשר להעביר את הערך כבר בסוף מחזור 21.
- זמן הביצוע של הפקודה: כאמור זה store ולכן מצפים לראות כאן שני מספרים: - אחד עבור תחילת ה-STD ואחד עבור תחילת ה-STA, כל אחד במחזור העוקב למחזור בו המידע שהפקודה מוחכה לו מוכן. לכן STA יכנס לחישוב כבר במחזור 2 ו-STD, שחיכה ל-P2 שהתקבל במחזור 21, נכנס לביוץ במחזור 22.
- ה-data ready קורה במחזור בו src2 מוחש (כי לא מתבצע עליו שום חישוב) - 22.
- זמן ה-commit הוא במחזור אחריו זמן קבלת ה-data קלומר במחזור מס' 23.

פקודה 4 - add: R1=R1+10

| Pdst | instruction  | R1 | R2 | R3 | addr | data | T alloc | src1 | src2 | Imm | T src1 ready | T src2 ready | T exe | Load block code | T data ready | T commit |
|------|--------------|----|----|----|------|------|---------|------|------|-----|--------------|--------------|-------|-----------------|--------------|----------|
| 4    | add R1=R1+10 | 20 |    |    |      |      | 2       | R1   |      | 10  | 2            |              | 3     |                 |              | 23       |

- בפקודה זו מתחילה את הסבב הבא של האлокציות.
- פקודה 4 היא החמישית לעבר הקזאה ולכן מוקזית במחזור מס' 2.
  - המקור היחיד שלו הוא R1.
  - כמו כן לפקודה יש ערך מיידי של 10.
  - המקור, R1, מוכן במחזור מס' 2 קלומר מיד כשנכנסה.
  - במחזור אחריו קיבלת הממקור היא נוכנת לביצוע, קלומר כבר במחזור 3. נשים לב שהיא עוקפת את שני פקודות store ולמעשה מתבצעת Out Of Order.
  - מכיוון ש-commit נעשה in-order הפקודה עשויה רק במחזור 23 - בתרגיל זה אפשר לעשות commit לאחר פקודות שנרכזה במקביל בכל מחזור.

הערה: נניח ויש בטבלה פקודת קפיצה עבורה מתקבל חייזי-שגווי (אם כי זה לא המצב בתרגיל) - אם נתון שלוקח חמישה מחזוריים עד שמගלים את השגיאה בחיזוי אז אחרי הקפיצה נוכנות פקודות לא נוכנות מראש חמישה מחזוריים ואנו צריכים צרכיים למלא את הטבלה גם עבורן, אבל הן לא עשוות commit - אף פעם לא עשוים לפקודה לא נוכנה! חשוב לזכור את זה אם ניתקל בזיה במחבן.

שאלה: פקודת branch צריכה לעשות commit - מדוע? תשובה: כי היא מעדכנת את ה-data pointer !instruction pointer. כללי כל פקודה צריכה לעשות commit בין אם היא משנה את מצב המכונה או אפילו אם היא dop.

**פקודה 5 - blt: branch if(R1<100)**

| Pdst | instruction        | R1 | R2 | R3  | addr | data | T alloc | src1 | src2 | Imm | T src1 ready | T src2 ready | T exe | Load block code | T data ready | T commit |
|------|--------------------|----|----|-----|------|------|---------|------|------|-----|--------------|--------------|-------|-----------------|--------------|----------|
| 5    | blt<br>if (R1<100) | 20 | 40 | 110 |      |      | 2       | P4   |      | 100 | 3            |              | 4     |                 |              | 23       |

- הקפיצה עושה הקצאה במחזור 2 והמקור שלה הוא הערך שיהיה ב-R1 לאחר שייחסב ע"י פקודה מס' 4, כלומר P4.
- המקור P4 מוכן כבר במחזור 3 (מחושב במחזור 3 בשורה מס' 4).
- מכיוון שהמקור מוכן כבר במחזור 3 ניתן לבצע הפקודה יכול להתחליל כבר במחזור 4.

**פקודה 6 - load: R2=m[R1+30]**

| Pdst | instruction         | R1 | R2  | R3 | addr | data | T alloc | src1 | src2 | Imm | T src1 ready | T src2 ready | T exe | Load block code | T data ready | T commit |
|------|---------------------|----|-----|----|------|------|---------|------|------|-----|--------------|--------------|-------|-----------------|--------------|----------|
| 6    | load<br>R2=m[R1+30] |    | 110 |    | 50   | 110  | 2       | P4   |      | 30  | 3            |              | 4     | 1, 2            | 24           | 25       |

- הפקודה עושה הקצאה במחזור 2.
- המקור שלה, P4, מוכן במחזור 3 - אותו דבר כמו עברו פקודה מס' 5.
- נשלח את הפקודה לביצוע, ככלור לחישוב כתובת הטיענה, במחזור 4 - המחזור אחריו שהמקור מוכן.
- אם יש ל-load זה תנאי חסימה? צריך להסתכל על כל ה-stores שלפנויו - יש שניים (פקודות 3 ו-1):
- נשים לב שה-load קורא מכתובת 50 ופקודה מס' 3 היא store שכותב לשם - אך ה-load חסום על תנאי מס' 2 עברו (תנאי חסימה על ה-data).
- כמו כן יש תנאי חסימה מה-store בשורה מס' 1 בגלל שבמחזור 4 עוד לא סיימנו לחשב את הכתובת של פקודה מס' 1 (store 1) ולכן עד מחזור 12 לא ידוע האם היא כתובת לכתחבת 50 או לא. זהו חסימה על הכתובת - תנאי-חסימה מס' 1. בפועל במקרה זה תנאי מס' 1 הוא חלש יותר כי הוא ישחרר מוקדם יותר אבל זהו אכן חסימה.
- במחזור 22 מתבטל תנאי 1, ב-23 ה-load נבדק שוב, ובמחזור 24 מתבצע forwarding אליו - המידע מוכן בסוף המחזור בו נעשה forwarding - אך במחזור 24.
- ניתן יהיה להתחייב (commit) על הפקודה במחזור 25.

את שאר הטרבלה נוכל למלא בעצמנו, הנה התוצאה הסופית (מה שכבר עשינו מסווג במוגרת הצהובה):

| Pdst | instruction          | R1 | R2  | R3  | addr | data | T alloc | src1 | ✓ src2 | Imm | T src1 ready | T src2 ready | T exe             | Load block code | T data ready | T commit |
|------|----------------------|----|-----|-----|------|------|---------|------|--------|-----|--------------|--------------|-------------------|-----------------|--------------|----------|
| 0    | load<br>R2=m[R1+30]  | 10 | 40  | 10  | 40   | 40   | 1       | R1   |        | 30  | 1            |              | 2                 | 0               | 11           | 12       |
| 1    | store<br>m[R2+20]=R1 |    |     |     | 60   | 10   | 1       | P0   | R1     | 20  | 11           | 1            | Std: 2<br>Sta: 12 |                 | 12           | 13       |
| 2    | load<br>R3=m[R1+100] |    |     | 110 | 110  | 110  | 1       | R1   |        | 100 | 1            |              | 2                 | 1               | 21           | 22       |
| 3    | store<br>m[R1+40]=R3 |    |     |     | 50   | 110  | 1       | R1   | P2     | 40  | 1            | 21           | Std: 22<br>Sta: 2 |                 | 22           | 23       |
| 4    | add R1=R1+10         | 20 |     |     |      |      | 2       | R1   |        | 10  | 2            |              | 3                 |                 |              | 23       |
| 5    | blt<br>if (R1<100)   | 20 | 40  | 110 |      |      | 2       | P4   |        | 100 | 3            |              | 4                 |                 |              | 23       |
| 6    | load<br>R2=m[R1+30]  |    | 110 |     | 50   | 110  | 2       | P4   |        | 30  | 3            |              | 4                 | 1, 2            | 24           | 25       |
| 7    | store<br>m[R2+20]=R1 |    |     |     | 130  | 20   | 2       | P6   | P4     | 20  | 24           | 3            | Std: 4<br>Sta: 25 |                 | 25           | 26       |
| 8    | load<br>R3=m[R1+100] |    |     | 120 | 120  | 120  | 3       | P4   |        | 100 | 3            |              | 4                 | 1               | 27           | 28       |
| 9    | store<br>m[R1+40]=R3 |    |     |     | 60   | 120  | 3       | P8   | P4     | 40  | 27           | 3            | Std: 4<br>Sta: 28 |                 | 28           | 29       |
| 10   | add R1=R1+10         | 30 |     |     |      |      | 3       | P4   |        | 10  | 3            |              | 4                 |                 |              | 29       |
| 11   | blt<br>if (R1<100)   | 30 | 110 | 120 |      |      | 3       | P10  |        | 100 | 4            |              | 5                 |                 |              | 29       |