

MIPS ISA
רגיסטרים: השמות מתחילים ב-\$. אפשר לסמן אותם או בתור \$0, ..., \$31 או לפי \$s0 – \$s7, \$t0 – \$t9, ... בנוסף ל-General Purpose Registers יש גם את ה-Program Counter שבו נמצאת הכתובת של הפקודה הנוכחית.
זכרון: byte-addressable כלומר אפשר להעביר בייטים, half-word (2 בייטים) ו-word (4 בייטים).
מודל Von-Neumann: גם הקוד וגם המידע נמצאים באותו הזיכרון. קשה להתמודד עם self-modifying code. **מודל Harvard:** הקוד נפרד מהמידע. יותר קשה לעדכן קוד.

קידוד R-type:

op (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
--------	--------	--------	--------	-----------	-----------

- op = 000000
- rs, rt הם רגיסטרים של המקור הראשון והשני
- rd הוא destination register
- shamt משמש רק בשביל shift
- funct בוחר פעולה אריתמטית או לוגית.

דוגמה: (\$t0 ← \$t1 + \$t2) add \$t0, \$t1, \$t2

קידוד I-type:

op (6)	rs (5)	rt (5)	imm (16)
--------	--------	--------	----------

- op בוחר פעולה
- rs source register
- rt destination register
- imm signed בין ל-32768 ל-32767

דוגמה: (\$t0 ← \$t1 + 4) addi \$t0, \$t1, +4
(\$t1 ← Mem[\$t2 + 8]) lw \$t1, +8(\$t2)
(Mem[\$t2 - 4] ← \$t1) sw \$t1, -4(\$t2)
(\$t1 = \$t2 אם target = (PC + 4) + (offset << 2)) beq \$t1, \$t2, +7

קידוד J-type:

op (6)	offset (26)
--------	-------------

target = ((PC + 4) & 0xf0000000) | (offset << 2)

פסאודו-אינסטרקשן: הרגיסטר \$1 או \$at משמש את האסמבלר כדי לבטא פקודות מסובכות.

blt \$s0, \$s1, x	slt \$at, \$s0, \$s1 ; bne \$at, \$zero, x
nop	sll \$zero, \$zero, 0
li \$s0, 0x12345678	lui \$at, 0x1234 ; ori \$s0, \$at, 0x5678

פונקציות: \$v0, \$v1 החזרה \$a0, ..., \$a3 וערכי החזרה \$v0, \$v1. כתובת חזרה \$ra. קריאה לפונקציה: jal foo. חזרה מפונקציה: jr \$ra.

רגיסטרים Callee-saved ו-Caller-saved: הרגיסטרים \$t*, \$a*, \$ra הם caller-saved. הרגיסטרים \$s*, \$sp הם callee-saved.

מחסנית: לדחוף את \$ra: addi \$sp, \$sp, -4 ; sw \$ra, (\$sp)
לשחזר את \$ra: addi \$sp, \$sp, +4 ; lw \$ra, (\$sp)

דוגמה לפונקציה שמשתמשת ב-\$s0 וקוראת לפונקציות:

Epilogue: addi \$sp, \$sp, -8 ; sw \$ra, +4(\$sp) ; sw \$s0, (\$sp)
Prologue: lw \$s0, (\$sp) ; lw \$ra, +4(\$sp) ; addi \$sp, \$sp, +8 ; jr \$ra

MIPS Single Cycle

Fetch Decode eXecute Memory Writeback

יחידת control-ת מקבלת את 6 הביטים התחתונים (op) והעליונים (funct) של (R-type) ואת הפלט zero של ALU. ומיצרת 7 פלטים:

- MemWrite
- PCSrc
- ALUOp
- RegDst
- ALUSrc
- RegWrite
- MemToReg

Instr.	RegDst	RegWrite	ALUSrc	ALUOp
add	1	1	0	010
sub	1	1	0	110
or	1	1	0	001
addi	0	1	1	010
lw	0	1	1	010
sw	x	0	1	010
beq	x	0	0	110
Instr.	MemWrite	MemToReg	PCSrc	
add	0	0	0	
sub	0	0	0	
or	0	0	0	
addi	0	0	0	
lw	0	1	0	
sw	1	x	0	
beq	0	x	0/1	

Multi Cycle

נוסיף רגיסטרים בין השלבים ב-single cycle. הזכרון instruction ו-data הוא אותו זיכרון. ואת control הוא מכונת מצבים.

Pipeline

נוסיף רגיסטרים בין השלבים: IF/ID, ID/EX, EX/MEM, MEM/WB.

סוגי hazards:

- Structural hazards:** hardware cannot support a combination of instructions
- Data hazards:** Instruction depends on the result of prior instruction which is still in the pipeline.
- Control hazards:** branch resolution depends on the result of a previous operation

פתרונות ל-data hazards:

- Bubble (interlock): לעצור את כל הפקודות חוץ מזאת שיש תלות בפלט שלה.
- Forwarding: להשתמש בתוצאה מהרגע שהיא חושבה ולא רק ממתי שהיא נכנסת לרגיסטרים.
- Transparent register file: אפשר לכתוב ולקרוא מאותו רגיסטר באותו מחזור שעון.

פתרונות ל-control hazards:

- Always fetch not-taken: להריץ פקודות עם ההנחה שלא נקח אתה-branch. אם ה-branch נלקח אז נפטר מהפקודות שנמצאות כרגע ב-pipeline לפני שהן משנות את הזיכרון או את הרגיסטרים.
- Delayed branches: ה-branch מתבצע רק כמה פקודות לאחר המיקום

שערים לוגיים			
	Gate	State	Input Output
NMOS	1	On	0 0
			1 weak 1
		Off	Any Z
PMOS	0	On	0 weak 0
			1 1
		Off	Any Z

כדי ליצור שערים חלק אחד בנוי מ-PMOS ומחובר לחשמל ומוציא פלטים של 1, וחלק שני בנוי מ-NMOS ומוציא פלטים של 0. אם חלק אחד דלוק החלק השני כבוי (מוציא Z).

ביטויים בוליאניים: קבועים 0/1, משתנים x_1, \dots, x_n , אופרטורים: x' (not), $x \uparrow y$ (nand), $x \downarrow y$ (nor), $x \oplus y$ (xor), $x \oplus y$ (and), xy (or), $x + y$

מפת קרנו: נסדר את השורות והעמודות כך ששורות ועמודות עוקבות שונות זו מזו בביט אחד. נרצה לכסות את הביטים עם ערך 1 בטבלת האמת עם כמה שפחות מלבנים שאורכם ורוחבם הם חזקות 2, וכל אחד מהם כמה שיותר גדול. כל מלבן הוא מכפלה של כמה משתנים.

חלקים לוגיים

Decoder: Input: n bits, a number between 0 and $2^n - 1$.
Output: 2^n bits, output j is $1 \iff$ input number = j .
We can create any function using a decoder and OR gates.

Encoder: Input: 2^n bits, “one-hot” - exactly one bit is set to 1.
Output: n bits, representing (in binary) the location of the 1. If not unary, output is not defined.

Multiplexer: Input: 2^n input bits and n selector bits.
Output: one bit which gets the value of the input indexed by selector.

Half Adder: Input: x, y .
Output: sum = $x \oplus y = x'y + xy'$, carry = xy

Full Adder: Input: x, y, c_{i-1}
Output: x, y, s_i, c_i . Built by chaining HA(x, y)'s sum to HA(c_{i-1} , sum), which is s_i , and c_i is OR of both carry outputs.

Ripple-Carry Adder: If $C_{n-1} = 1$ then there's overflow.

HA $(x_0, y_0) \xrightarrow{S_0} \text{FA} (x_1, y_1) \xrightarrow{C_1} \dots \text{FA} (x_{n-1}, y_{n-1}) \xrightarrow{C_{n-1}}$

Ripple-Carry Subtractor: $(-y) = (y' + 1)$, so:
 $S_0 \quad S_1 \quad S_{n-1}$
 $1 \rightarrow \text{FA} (x_0, y'_0) \xrightarrow{C_0} \text{FA} (x_1, y'_1) \xrightarrow{C_1} \dots \text{FA} (x_{n-1}, y'_{n-1}) \xrightarrow{C_{n-1}}$

Signed overflow detection: carry into MSB \neq carry out of MSB.

Carry-Lookahead Adder: $g_i = x_i y_i, p_i = x_i + y_i, c_{i+1} = g_i + p_i c_i$ (open the definition for every i)

Comparator: $x = y \iff x - y = 0 / x \oplus y = 0$, unsigned $x \geq y \iff$ computing $x - y$ yields no unsigned overflow, signed $x \geq y \iff x \geq 0 > y$ or same sign and $x - y$ is non-negative by MSB.

Unsigned Multiplication: $p_i[j] = a[j] b[i], P = \sum 2^i p_i$

תזמונים

Propagation Delay t_{pd} : זמן מקסימלי עד שהפלט מפסיק להשתנות ברגע שהקלט מפסיק להשתנות.

Contamination Delay t_{cd} : זמן מינימלי עד שהפלט מתחיל להשתנות ברגע שהקלט מתחיל להשתנות.

t_{setup} : זמן לפני edge ש- D צריך להיות יציב.

t_{hold} : זמן אחרי edge ש- D צריך להיות יציב.

t_{pcq} : הזמן שלוקח ל-Q להתייצב אחרי ש-Clk התייצב.

t_{ccq} : הזמן משינוי ב-Clk לשינוי ב-Q.

דרישה: $t_{hold} \leq t_{ccq} + t_{cd}$ **דרישה:** $t_{cycle} \geq t_{pcq} + t_{pd} + t_{setup}$

זמן ריצה: $IC \times CPI \times \text{Clock cycle}$

רגיסטרים

SR Latch

D Latch (level triggered)

D Flip-Flop (edge triggered)

S	R	Q_{t-1}	Q'_{t-1}	Q_t	Q'_t
0	0	Q	Q'	Q	Q'
0	1	Q	Q'	0	1
1	0	Q	Q'	1	0
1	1	Q	Q'	0	0

D	WE	Q_{t-1}	Q_t
D	0	Q	Q
0	1	Q	0
1	1	Q	1

D	WE	Q_{t-1}	Q_t
D	0	Q	Q
D	1	Q	Q
D	1→0	Q	0
D	0→1	Q	1

MIPS Register File: Input: Read Reg 1 (5), Read Reg 2 (5), Write Reg (5), Write Data (32)
Output: Read Data 1 (32), Read Data 2 (32)
Register \$0 is constant 0.

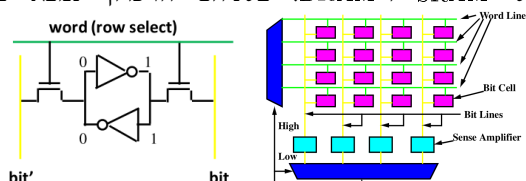
Moore and Mealy FSMs: In a Moore FSM, outputs only depend on current state, in Mealy FSM, outputs depend on the current state and on the inputs.

שלו. הפקודות שנמצאות ישר אחרי ה-branch מתבצעות בכל מקרה. יתכן שלקומפילר יהיו פקודות שימושיות לשים שמה - אם לא אפשר לשים nop.

- Dynamic branch prediction: בשלב ה-fetch נבחר כתובת לפי ה-prediction. נשמור ב-Branch Target Buffer.
- Backward jump predictor: אם ה-branch קופץ אחורה נניח שלוקחים אותו. זה מייצל לולאות והרבה יותר פשוט מ-dynamic branch prediction.
- Bimodal predictor: מכונה מצבים עם 4 מצבים לכל BTB entry: strong-ly/weakly not-taken, weakly/strongly taken.
- Software hints: ב-MIPS II יש פקודה של Likely taken branch.
- Predication: להמיר control hazard ל-data hazard: ב-MIPS IV יש פקודות movn, movz שמעתיקות ערך לפי האם ערך אחר שווה או שונה מ-0.

RAM

יש SRAM ו-DRAM. בשניהם הזיכרון נמצא במטריצה דו מימדית.



Register	#cycles till writeback					In bypass (WB)	In RF
	5	4	3	2	1		
\$1						1	
\$2							1
\$31			1				

בכל סייקל נעדכן אותה בכך שניזי את כל הביטים ימינה, ואם יש בעיה אז נעשה stall. נשים לב שזה בעייתי גם לכתוב לשני רגיסטרים באותו ה-cycle (בגלל מבנה ה-register file).

Superscalar Pipeline: יכולות לרוץ מספר פקודות במקביל. Instruction-level parallelism. לדוגמה, ב-U-V Pipeline, בשלב ה-fetch וה-decode נוציא בכל פעם 2 פקודות. ואז שאר השלבים נפרדים ל-U-pipe ול-V-pipe כאשר V-pipe מוגבל יחסית ולא יכול להריץ הכל ומשמש רק בשביל פקודות בלתי תלויות. פקודות מסובכות יכולות להשתמש גם ב-U וגם ב-V. ההחלטה האם להשתמש ב-V-pipe היא ב-decoder.

זיכרון וירטואלי

כתובת בזיכרון מפוצלת ל-Virtual Page Number ול-Page Offset ויש טבלה RAM שממפה Virtual Page Number לכתובת פיזית. יש TLB שהוא cache של מפינים עם פייג'ים.

overlapped TLB and cache access: אם מניחים ש-set# מוכל ב-page offset אפשר להתחיל לגשת ל-cache במקביל לתרגום הכתובת.

virtually-addressed cache: תרגום הכתובת מתבצע רק במקרה של cache miss. אבל, יתכן ששני פייג'ים שונים ממפים לאותה כתובת פיזית. אז, אסור ששניהם יופיעו ביחד ב-cache, וב-cache miss צריך reverse TLB כדי לוודא שאף שורה אחרת לא ממופה לאותה כתובת.

OOOE

Reorder Buffer (ROB): פקודות נכתבות ונשמרות לפי הסדר שלהן ל-ROB. פקודות יוצאות מה-ROB (retire) ומעדכנות את המצב הארכיטקטוני. אם הן הכי ישנות וסיימו לרוץ. לכל פקודה, אם שני המקורות שלה רצו כבר היא מוכנה לרוץ, אחרת נוסף אותה לרשימת הפקודות המכונות של המקורות. כשפקודה מסיימת לרוץ היא מעדכנת את המשתמשים שלה והם יכולים להפוך למוכנים.

במקרה branch, על פי הפרדיקציה נמשך לקרוא עוד ועוד פקודות. אם זו הייתה פרדיקציה לא נכונה אז נאלץ למחוק את כל ה-ROB אחריו. ובמקרה של exception הוא יקרה רק כשהוא מגיע ל-retire ונרוקן את כל ה-ROB אז.

False Dependencies: אם שתי פקודות משתמשות באותו מספר רגיסטר זה לא אומר בהכרח שיש תלות מידע ביניהן. לכן נוסף register renaming. נשמור טבלה Register Aliases Table שממפה כל רגיסטר \$r לרגיסטר פיזי pr*. בשלב ה-retire נכתוב לרגיסטר הארכיטקטוני. בנוסף כדי לשחזר את ה-RAT אחריו flush, בכל ROB entry נשמור היסטוריה שאומרת איזה שינוי ROB קרה בפקודה הזו.

Scheduling Queue: הרבה מהפקודות ב-ROB סיימו לרוץ כבר, ומקום ב-ROB הוא יקר כי צריך בכל סייקל למצוא איזה פקודות בו מוכנות ועבור כל פקודה שנכנסת למצוא באיזה פקודה ב-ROB היא תלויה (כדי לרוץ אחריה). SQ הוא תור יותר קטן ששומרים בו רק את הפקודות שעדיין לא רצו. המימוש של התלויות הוא באמצעות מטריצה שבה כל ביט אומר שהפקודה ה-i מחכה לפקודה ה-j כדי להיות מוכנה.

Memory Disambiguation: פתרנו את הבעיה של רגיסטרים אבל לא של זיכרון. יכולה להיות תלות בין שתי גישות של זיכרון ולא נדע את זה בזמן האלוקציה כי צריך לחשב את הכתובת.

עבור כתיבות: לא מריצים כתיבות ספקולטיבית כי אין דרך לעשות להן undo. אף פעם לא מסדרים מחדש כתיבות - מריצים כתיבה רק כשהכתובת והמידע מוכנים וגם אין כתיבות קודמות שמכונות לרוץ. מכניסים את המידע ל-cache רק אחרי ה-retire.

עבור קריאות: קריאות צריכות לחכות עד שכל הכתיבות הקודמות מסיימות (אחרי ה-retire שלהן). אבל זה איטי. אפשר להגיד שקריאות מחכות רק אם יש חיתוך, ואז במקרה שאין חיתוך צריך רק לחכות שכל הכתיבות הקודמות יחשבו את הכתובת שלהן. בדומה ל-SQ נבנה Store Buffer ו-Load Buffer. ב-SB נחזיק את כל הכתובות של הכתיבות שנמצאות בתור וב-LB נחזיק את כל הכתובות של הקריאות בתור ואינדקס של כתיבה שמחכים שתסיימו. זה לא דורש לחכות שה-data של הכתיבה יהיה מוכן ולכן נפצל כתיבות לפקודה שמחשבת את הכתובת ושומרת את המידע.

TLP

עד עכשיו עבדנו על instruction-level parallelism אבל אם יש כמה חוטי ריצה נפרדים מעניין אותנו thread-level parallelism. SMP - Symmetric Multi-Processing - אומר שיש מספר מעבדים זהים (ליבות) עם גישה שווה לזיכרון משותף. SMT - Simultaneous Multi-Threading - שומר על אותה כמות ליבות אבל לוגית יש פי 2 יותר ליבות. לכל מעבד לוגי יש מצב ארכיטקטוני נפרד, אבל חלק מהמשאבים משותפים. למשל ה-SQ הוא משותף לשניהם. ה-ROB הוא מפוצל ובכל חצי יש פקודות של thread אחר ו-retirement בנפרד.

- Temporal Locality: אחרי שניגשים לכתובת לרוב ניגש אליה שוב בקרוב. למשל קוד ומשתנים.
- Spatial Locality: אחרי שניגשים לכתובת לרוב ניגש לכתובות סמוכות בקרוב. למשל סריקה של מערך.
- Fully Associative: (שורות של 64) tag = line# (31-5) | offset (5-0). כל שורה יכולה להיות בכל מקום ב-cache.
- Direct Mapped: tag = line# (31-14) | set (14-5) | offset (5-0). כל שורה יכולה להיות רק במקום אחד ב-cache שמתאים למספר ה-set.
- 2-Way Associative: tag = line# (31-14) | set (13-5) | offset (5-0). לכל שורה יש שני מקומות שהיא יכולה להיות בהם.

חישוב כמות הביטים: נסמן ב-N את מספר ה-ways, ב-b את גודל הבלוק וב-c את גודל הקאש. נגדיר $X = \log_2(b)$, $Y = \log_2(c/N)$.

tag = line#	(31-Y)	set	(Y-X)	offset	(X-0)
-------------	--------	-----	-------	--------	-------

ב-array tag נצטרך לשמור עבור כל set ו-way את ה-tag. בנוסף ביט valid ובמקרה של Write-Back גם ביט Dirty. בנוסף ביטים של pseudo-LRU או psuedo-LRU.

מהירות זיכרון: $AMAT = Hit Rate \cdot Hit Time + Miss Rate \cdot Miss Penalty$. אם יש L1 ו-L2, אז $Miss Penalty L1 = Hit Time L2 + Miss Rate L2 \cdot Miss Penalty L2$.

סוגי cache miss: • Compulsory miss: קורים גם אם ה-cache אינסופי. גישה ראשונה לזיכרון. • Capacity miss: קורים גם ב-fully associative - עובדים עם יותר זיכרון ממה שיש מקום ב-cache. • Conflict miss: הגודל של ה-cache מספיק אבל יש בעיה במיפוי של שורות לקבוצות.

החלפת שורות ב-cache: צריך לבחור את מי להוציא מבין האיברים בקבוצה. FIFO: קל לממש אבל מנוגד ל-temporal locality. LRU: נוציא את השורה שלא נגעו בה הכי הרבה זמן. חומרה מסובכת ולכן נשתמש ב-pseudo-LRU: עץ בינארי שבו כל צומת שומרת איזה צד היה האחרון בשימוש.

סוגי cache miss: • Compulsory miss: קורים גם אם ה-cache אינסופי. גישה ראשונה לזיכרון. • Capacity miss: קורים גם ב-fully associative - עובדים עם יותר זיכרון ממה שיש מקום ב-cache. • Conflict miss: הגודל של ה-cache מספיק אבל יש בעיה במיפוי של שורות לקבוצות.

החלפת שורות ב-cache: צריך לבחור את מי להוציא מבין האיברים בקבוצה. FIFO: קל לממש אבל מנוגד ל-temporal locality. LRU: נוציא את השורה שלא נגעו בה הכי הרבה זמן. חומרה מסובכת ולכן נשתמש ב-pseudo-LRU: עץ בינארי שבו כל צומת שומרת איזה צד היה האחרון בשימוש.

סוגי cache miss: • Compulsory miss: קורים גם אם ה-cache אינסופי. גישה ראשונה לזיכרון. • Capacity miss: קורים גם ב-fully associative - עובדים עם יותר זיכרון ממה שיש מקום ב-cache. • Conflict miss: הגודל של ה-cache מספיק אבל יש בעיה במיפוי של שורות לקבוצות.

החלפת שורות ב-cache: צריך לבחור את מי להוציא מבין האיברים בקבוצה. FIFO: קל לממש אבל מנוגד ל-temporal locality. LRU: נוציא את השורה שלא נגעו בה הכי הרבה זמן. חומרה מסובכת ולכן נשתמש ב-pseudo-LRU: עץ בינארי שבו כל צומת שומרת איזה צד היה האחרון בשימוש.

סוגי cache miss: • Compulsory miss: קורים גם אם ה-cache אינסופי. גישה ראשונה לזיכרון. • Capacity miss: קורים גם ב-fully associative - עובדים עם יותר זיכרון ממה שיש מקום ב-cache. • Conflict miss: הגודל של ה-cache מספיק אבל יש בעיה במיפוי של שורות לקבוצות.

החלפת שורות ב-cache: צריך לבחור את מי להוציא מבין האיברים בקבוצה. FIFO: קל לממש אבל מנוגד ל-temporal locality. LRU: נוציא את השורה שלא נגעו בה הכי הרבה זמן. חומרה מסובכת ולכן נשתמש ב-pseudo-LRU: עץ בינארי שבו כל צומת שומרת איזה צד היה האחרון בשימוש.

סוגי cache miss: • Compulsory miss: קורים גם אם ה-cache אינסופי. גישה ראשונה לזיכרון. • Capacity miss: קורים גם ב-fully associative - עובדים עם יותר זיכרון ממה שיש מקום ב-cache. • Conflict miss: הגודל של ה-cache מספיק אבל יש בעיה במיפוי של שורות לקבוצות.

החלפת שורות ב-cache: צריך לבחור את מי להוציא מבין האיברים בקבוצה. FIFO: קל לממש אבל מנוגד ל-temporal locality. LRU: נוציא את השורה שלא נגעו בה הכי הרבה זמן. חומרה מסובכת ולכן נשתמש ב-pseudo-LRU: עץ בינארי שבו כל צומת שומרת איזה צד היה האחרון בשימוש.

סוגי cache miss: • Compulsory miss: קורים גם אם ה-cache אינסופי. גישה ראשונה לזיכרון. • Capacity miss: קורים גם ב-fully associative - עובדים עם יותר זיכרון ממה שיש מקום ב-cache. • Conflict miss: הגודל של ה-cache מספיק אבל יש בעיה במיפוי של שורות לקבוצות.

החלפת שורות ב-cache: צריך לבחור את מי להוציא מבין האיברים בקבוצה. FIFO: קל לממש אבל מנוגד ל-temporal locality. LRU: נוציא את השורה שלא נגעו בה הכי הרבה זמן. חומרה מסובכת ולכן נשתמש ב-pseudo-LRU: עץ בינארי שבו כל צומת שומרת איזה צד היה האחרון בשימוש.

סוגי cache miss: • Compulsory miss: קורים גם אם ה-cache אינסופי. גישה ראשונה לזיכרון. • Capacity miss: קורים גם ב-fully associative - עובדים עם יותר זיכרון ממה שיש מקום ב-cache. • Conflict miss: הגודל של ה-cache מספיק אבל יש בעיה במיפוי של שורות לקבוצות.

החלפת שורות ב-cache: צריך לבחור את מי להוציא מבין האיברים בקבוצה. FIFO: קל לממש אבל מנוגד ל-temporal locality. LRU: נוציא את השורה שלא נגעו בה הכי הרבה זמן. חומרה מסובכת ולכן נשתמש ב-pseudo-LRU: עץ בינארי שבו כל צומת שומרת איזה צד היה האחרון בשימוש.

סוגי cache miss: • Compulsory miss: קורים גם אם ה-cache אינסופי. גישה ראשונה לזיכרון. • Capacity miss: קורים גם ב-fully associative - עובדים עם יותר זיכרון ממה שיש מקום ב-cache. • Conflict miss: הגודל של ה-cache מספיק אבל יש בעיה במיפוי של שורות לקבוצות.

החלפת שורות ב-cache: צריך לבחור את מי להוציא מבין האיברים בקבוצה. FIFO: קל לממש אבל מנוגד ל-temporal locality. LRU: נוציא את השורה שלא נגעו בה הכי הרבה זמן. חומרה מסובכת ולכן נשתמש ב-pseudo-LRU: עץ בינארי שבו כל צומת שומרת איזה צד היה האחרון בשימוש.

סוגי cache miss: • Compulsory miss: קורים גם אם ה-cache אינסופי. גישה ראשונה לזיכרון. • Capacity miss: קורים גם ב-fully associative - עובדים עם יותר זיכרון ממה שיש מקום ב-cache. • Conflict miss: הגודל של ה-cache מספיק אבל יש בעיה במיפוי של שורות לקבוצות.

החלפת שורות ב-cache: צריך לבחור את מי להוציא מבין האיברים בקבוצה. FIFO: קל לממש אבל מנוגד ל-temporal locality. LRU: נוציא את השורה שלא נגעו בה הכי הרבה זמן. חומרה מסובכת ולכן נשתמש ב-pseudo-LRU: עץ בינארי שבו כל צומת שומרת איזה צד היה האחרון בשימוש.

סוגי cache miss: • Compulsory miss: קורים גם אם ה-cache אינסופי. גישה ראשונה לזיכרון. • Capacity miss: קורים גם ב-fully associative - עובדים עם יותר זיכרון ממה שיש מקום ב-cache. • Conflict miss: הגודל של ה-cache מספיק אבל יש בעיה במיפוי של שורות לקבוצות.

החלפת שורות ב-cache: צריך לבחור את מי להוציא מבין האיברים בקבוצה. FIFO: קל לממש אבל מנוגד ל-temporal locality. LRU: נוציא את השורה שלא נגעו בה הכי הרבה זמן. חומרה מסובכת ולכן נשתמש ב-pseudo-LRU: עץ בינארי שבו כל צומת שומרת איזה צד היה האחרון בשימוש.

סוגי cache miss: • Compulsory miss: קורים גם אם ה-cache אינסופי. גישה ראשונה לזיכרון. • Capacity miss: קורים גם ב-fully associative - עובדים עם יותר זיכרון ממה שיש מקום ב-cache. • Conflict miss: הגודל של ה-cache מספיק אבל יש בעיה במיפוי של שורות לקבוצות.

החלפת שורות ב-cache: צריך לבחור את מי להוציא מבין האיברים בקבוצה. FIFO: קל לממש אבל מנוגד ל-temporal locality. LRU: נוציא את השורה שלא נגעו בה הכי הרבה זמן. חומרה מסובכת ולכן נשתמש ב-pseudo-LRU: עץ בינארי שבו כל צומת שומרת איזה צד היה האחרון בשימוש.

סוגי cache miss: • Compulsory miss: קורים גם אם ה-cache אינסופי. גישה ראשונה לזיכרון. • Capacity miss: קורים גם ב-fully associative - עובדים עם יותר זיכרון ממה שיש מקום ב-cache. • Conflict miss: הגודל של ה-cache מספיק אבל יש בעיה במיפוי של שורות לקבוצות.

החלפת שורות ב-cache: צריך לבחור את מי להוציא מבין האיברים בקבוצה. FIFO: קל לממש אבל מנוגד ל-temporal locality. LRU: נוציא את השורה שלא נגעו בה הכי הרבה זמן. חומרה מסובכת ולכן נשתמש ב-pseudo-LRU: עץ בינארי שבו כל צומת שומרת איזה צד היה האחרון בשימוש.

סוגי cache miss: • Compulsory miss: קורים גם אם ה-cache אינסופי. גישה ראשונה לזיכרון. • Capacity miss: קורים גם ב-fully associative - עובדים עם יותר זיכרון ממה שיש מקום ב-cache. • Conflict miss: הגודל של ה-cache מספיק אבל יש בעיה במיפוי של שורות לקבוצות.

החלפת שורות ב-cache: צריך לבחור את מי להוציא מבין האיברים בקבוצה. FIFO: קל לממש אבל מנוגד ל-temporal locality. LRU: נוציא את השורה שלא נגעו בה הכי הרבה זמן. חומרה מסובכת ולכן נשתמש ב-pseudo-LRU: עץ בינארי שבו כל צומת שומרת איזה צד היה האחרון בשימוש.