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 N current state, in Mealy FSM, outputs depend on the current state and on the inputs.

MIPS ISA

רגיסטרים: השמות מתחילים ב־\$. אפשר לסמן אותם או בתור $\$0,\ldots,\31 או ע גם General Purpose Registers- בנוסף ל-\$ $so-\$s7,\$t0-\$t9,\ldots$ לפי את הרסקודה הנוכחית. Program Counter שבו נמצאת הכתובת של הפקודה

(בייטים) half-word כלומר אפשר להעביר בייטים, byte-addressable זכרון: ו־word (4 בייטים).

מודל Von-Neaumann: גם הקוד וגם המידע נמצאים באותו הזיכרון. קשה להתמודד עם self-modifying code. מודל Self-modifying code להתמודד עם יותר קשה לעדכן קוד.

:R-type קידוד

op (\mathbf{o} \mathbf{r}	$\mathbf{s}(\mathfrak{d})$	rt (ວ)	ra (o)	snamt ((5)	funct (b	Ш
			—						000000	

הוא $\operatorname{rd} ullet$ הם רגיסטרים של המקור הראשון והשני $\operatorname{rs, rt} ullet$ הוא rs, $\operatorname{rt} ullet$ הוא shamt • destination register • shift משמש רק בשביל אריתמטית או לוגית.

 $(\$t0 \leftarrow \$t1 + \$t2) \text{ add } \$t0, \$t1, \$t2$ דוגמה:

:I-type קידוד

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

imm • destination register ⁻ rt ,source register ⁻ rs • בוחר פעולה op • 32767ל־-32768 בין signed הוא ערך

 $(\$t0 \leftarrow \$t1 + 4) \text{ addi } \$t0, \$t1, +4$ דוגמה:

 $(\$t1 \leftarrow \text{Mem} [\$t2 + 8]) \text{ lw } \$t1, +8 (\$t2)$ $(\text{Mem} [\$t2 - 4] \leftarrow \$t1) \text{ sw } \$t1, -4 (\$t2)$

(t1 = t2 אם ,target = (PC + 4) + (offset << 2)) beq t1, t2, +7

:J-type קידוד

op (6) offset (26)

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

פסאודו־אינסטרקשן: הרגיסטר \$1 או \$1 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

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

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

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

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

ה־Control מקבל רק את ה־opcode שהוא ה־6 ביטים הראשונים בשביל ליצור את הערכים שלו. יש בנוסף ALU control שמקבל את ה־6 ביטים העליונים (עבור ALUop הם מכילים את הפקודה) ושני ביטים של R-type למקרה שזה . שאומר להסתמך על ה־6 ביטים העליונים). $^{\circ}$ R-type

Multi Cycle

נוסיף רגיסטרים בין השלבים ב־single cycle: שניים ששומרים את המידע שנקרא מהרגיסטרים, ואחד ששומר את תוצאת ה־ALU. הזכרון instruction מהרגיסטרים הוא אותו זיכרון. control הוא מכונת מצבים.

Pipeline

 ${
m LIF/ID,\ ID/EX,\ EX/MEM,\ MEM/WB}$ נוסיף רגיסטרים בין השלבים:

ים:hazard ים:

- Structural hazards: hardware cannot support a combination of in-
- 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

:data hazards-פתרונות ל

- שיש תלות בפלט: Bubble (interlock) לעצור את כל הפקודות חוץ מזאת שיש
- י להשתמש בתוצאה מהרגע שהיא חושבה ולא רק ממתי שהיא:Forwarding נכנסת לרגיסטרים. למשל עבור שתי פעולות אריתמטיות עוקבות נצטרך . Forwarding EX/MEM \rightarrow ID/EX
- שפר לכתוב ולקרוא מאותו רגיסטר באותו מחזור :Transparent register file שעון.

:control hazards

- .branch להריץ פקודות עם ההנחה שלא נקח אתה־Always fetch not-taken אם ה־branch נלקח אז נפטר מהפקודות שנמצאות כרגע ב־pipeline לפני שהן משנות את הזיכרון או את הרגיסטרים.
- ה־Delayed branches מתבצע רק כמה פקודות לאחר המיקום שלו. הפקודות שנמצאות ישר אחרי ה־branch מתבצעות בכל מקרה. יתכן שלקומפיילר יהיו פקודות שימושיות לשים שמה ⁻ אם לא אפשר לשים nop.
- Dynamic branch prediction: בשלב ה־fetch נבחר כתובת לפי ה־

שערים לוגיים

	Gate	State	Input	Output
NMOS	1	On	0	0
VIVIOS	_	On	1	weak 1
	0 (any value other than 1)	Off	Any	Z
	Gate	State	Input	Output
PMOS	0	On	0	weak 0
1,1100	V	On	1	1
	1 (any value other than 0)	Off	Any	Z

כדי ליצור שערים חלק אחד בנוי מ־PMOS ומחובר לחשמל ומוציא פלטים של החלק אחד דלוק של 0. אם מוציא אוו ${
m NMOS}$, וחלק שני בנוי מ־ השני כבוי (מוציא Z).

 (not) x' אופרטורים: x_1,\ldots,x_n , משתנים אופרטורים: x'.(nand) $x \uparrow y$, (nor) $x \downarrow y$, (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 = $\hat{x} \oplus y = x'y + xy'$, carry = xy

Full Adder: Input: x, y, c_{i-1}

Built by chaining HA(x,y)'s sum to x, y, s_i, c_i . $\operatorname{HA}(c_{i-1}, \operatorname{sum})$, 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.

 $\operatorname{HA}(x_0, y_0) \xrightarrow{C_0} \operatorname{FA}(x_1, y_1) \xrightarrow{C_1} \dots \operatorname{FA}(x_{n-1}, y_{n-1}) \xrightarrow{C_{n-1}}$

Ripple-Carry Subtractor: (-y) = (y'+1), so:

$$1 \to \operatorname{FA}(x_0, y_0') \xrightarrow{C_0} \operatorname{FA}(x_1, y_1') \xrightarrow{C_1} \dots \operatorname{FA}(x_{n-1}, y_{n-1}') \xrightarrow{C_{n-1}}$$

Signed overflow detection: carry into MSB ≠ 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 $x \ge y$ \iff computing x-y yields no unsigned overflow, signed $x \geq y \iff$ $x \ge 0 > y$ 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

אמן מקסימלי עד שהפלט מפסיק להשתנות ברגע . $ext{Propagation Delay}: t_{pd}$ שהקלט מפסיק להשתנות.

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

עיב. צריך להיות אריך פdge אמן יציב: t_{setup}

שים צריך להיות יציב. d פdge אחרי: t_{hold}

התייצב הזמן שלוקח ל־Q להתייצב אחרי ש־ t_{pcq}

Qבינוי ב־Clk הזמן משינוי : t_{ccq}

 $t_{cycle} \geq t_{pcq} + t_{pd} + t_{setup}$ דרישה: $t_{hold} \leq t_{ccq} + t_{cd}$.IC \times CPI \times Clock cycle :זמן ריצה

חוק אמדל: נסמן Fraction Enhanced החלק מהזמן בו משתמשים בשיפור. . פי כמה שופר אותו חלק שעבר שיפור Speedup Enhanced

 $\frac{\text{ExecTime}_{\text{old}}}{\text{ExecTime}_{\text{new}}} = \frac{1}{(1 - \text{Fraction Enhanced}) + \frac{\text{Fraction Enhanced}}{\text{Speedup Enhanced}}}$ $Speedup_{Overall} = \frac{ExecTime_{new}}{ExecTime_{new}}$

nanosecond = 10^{-9} seconds, microsecond = 10^{-6} seconds



SR Latch

D Latch

(level

triggered)

D Flip-Flop

(edge

·		- Q	\Q		
1	0	Q	Q'	1	
1	1	Q	Q'	0	
	D	WE	Q_{t-1}	Q_t	
	D	0	Q	Q	1
	0	1	Q	0	
	1	1	Q	1]

 $\overline{\mathbf{D}}$

triggered) MIPS Register File: Input: Read Reg 1 (5), Read Reg 2 (5), Write

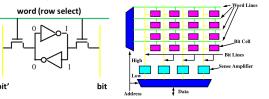
Reg (5), Write Data (32)

Branch Target Buffer נשמור ב-prediction.

- שלוקחים אותו. Backward jump predictor פופץ אחורה נניח שלוקחים אותו.
 מייעל לולאות והרבה יותר פשוט מ־dynamic branch prediction.
- strong- :BTB entry מכונת מצבים עם 4 מצבים :Bimodal predictor .ly/weakly not-taken, weakly/strongly taken
 - .Likely taken branch יש פקודה של MIPS II־ב :Software hints ●
- Predication: להמיר control hazard ל-data hazard: ב־MIPS IV: ש פקודות movn, movz שמעתיקות ערך לפי האם ערך אחר שווה או שונה מ־0.

$\underline{\mathbf{RAM}}$

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



ייכרון לרמה עליו. בכתיבה נשים capacitor עליו. בכתיבה נשים מלרו בכתיבה נאים לבחר בשורה ואז ה־capacitor נטען או מוציא את התוכן שלו. ערך x ב־bit ונבחר בשורה ואז ה־capacitor נטען או מוציא את התוכן שלו בקריאה נטען ערך $V_{\rm dd}/2$, נבחר בשורה, ונמדוד את השינויים ב־bit ואז נשחזר את הערך באמצעות כתיבה. בנוסף צריך לעשות Refresh בכך שנבצע קריאה בכל התאים כל כמה זמן כי הטעינה ב־capacitor יורדת באופן טבעי. יותר מהר לקרוא עמודות באותה שורה מאשר שורות שונות. משמש בתור זיכרון ראשי.

<u>Cache</u>

סוגי לוקאליות: ● Temporal Locality י אחרי שניגשים לכתובת לרוב ניגש אליה שוב בקרוב. למשל קוד ומשתנים.

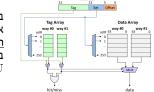
אחרי שניגשים לכתובת לרוב ניגש לכתובות המוכות האחרי Spatial Locality בקרוב. למשל סריקה של מערך.

tag = line# (31-5) | offset (5-0) | (64 שורות של Fully Associative מל שורה יכולה להיות בכל מקום ב־מרבה (21.14) שמורה יכולה להיות בל מקום ב-מרבה (21.14) שמורה יכולה (21.14) שמורה (21.

tag = line# (31-14) | set (14-5) | offset (5-0) : Direct Mapped .set מל שורה יכולה להיות רק במקום אחד ב־cache שמתאים למספר ה־set מינולה להיות רק במקום אחד ב־(10.5) שמתאים למספר ה־set מינולה להיות רק במקום אחד ב־(10.5) שמתאים למספר ה־set מינולה להיות רק במקום אחד ב־(10.5) שמתאים למספר ה־set מינולה מינולה למספר ה־set מינולה מינולה

 ${
m tag = line \# (31-14) \mid set \ (13-5) \mid offset \ (5-0)}$:2-Way Associative לכל שורה יש שני מקומות שהיא יכולה להיות בהם.

c-ביל את גודל הבלוק וב־N את מספר ה־ways המטוב נסמן ב־N את גודל הבלוק וב־ $X = \log_2\left(b\right), Y = \log_2\left(c/N\right)$ את גודל הקאש. נגדיר $\log_2\left(c/N\right)$ או $\log_2\left(\sin\#\left(31\text{-}Y\right)\right)$ set (Y-X) offset (X-0)

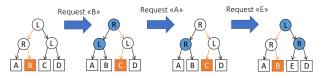


m wayו set יצטרך לשמור עבור כל tag array ב' tag בכל שיטות את ה־tag בנוסף ביט valid (קיים בכל שיטות ביט Dirty (במקרה של Write-Back ביטים של psuedo-LRU בנוסף ביטים של LRU בנוסף ביטים של LRU

AMAT = Hit Rate·Hit Time + Miss Rate·Miss Penalty מהירות זיכרון: Miss Penalty L1 = Hit Time L2 + Miss Rate L2 אס יש L1 ו-L1 אז גער אווי בא או

סוגי cache היס אינסופי. קורים גם אם ה־Compulsory miss ullet cache miss יקורים גם ב־fully associative גישה ראשונה לזיכרון. Capacity miss יקורים גם ב־Conflict miss יקורים אובדים עם יותר זיכרון ממה שיש מקום ב-cache מספיק אבל יש בעיה במיפוי של שורות לקבוצות.

החלפת שורות ב־cache: צריך לבחור את מי להוציא מבין האיברים בקבוצה. במהלפת שורות ב־cache: נוציא את השורה :FIFO לממש אבל מנוגד ל־tRU .temporal locality: 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.



כתיבה עם cache במקרה של hit: ב־thit תמיד נעדכן גם את במריבה עם cache במקרה של cache. זה טוב ל־cherency אבל לוקח יותר cacher. זה טוב ל־cherency אבל לוקח יותר write-Back. ב־bandwidth (עדכן רק את ה־cacher) נעדכן את הזיכרון ב־tacher). זה מוריד את ה־latency של כתיבות אבל עלול (coherency). לשבור coherency.

כתיבה עם cache במקרה של miss: ב־שrite מען את כל הבלוק write- (read miss ל-cache). זה מתאים ל-cache ל-cache (בדומה ל-read miss) ורק אז נכתוב ל-write no-allocate ב-Write no-allocate נכתוב ישירות לזיכרון הראשי. זה מתאים ל-write-Through. המעבד לא צריך לחכות שהכתיבה תסיים.

Data alignment: אם קוראים מכתובות לא עגולות יכול להיות שקריאה אחת במכתוב שני מכתובות של MIPS יש דרישה שכל פעולת זיכרון חייבת להיות cache line ל-aligned

סנכרון זכרון: קוהרנטיות: כל הליבות צריכות לראות את אותה תמונת זיכרון ולראות את המצב העדכני. קונסיסטנטיות: אם כתבנו לכתובת ואף אחד אחר לא שינה אותה, נרצה לקרוא את אותו ערך שכתבנו. אם כתבנו לכתובת ומישהו אחר קורא ממנה (לאחר מספיק זמן), הוא צריך לקרוא את מה שכתבנו. אם

שני מעבדים כותבים לאותה כתובת, כולם יסכימו על סדר הכתיבה. Exceptions

Cause סוג השגיאה, שומרים את PC ב-EPC הומרים את הארה נשמר ב-EPC הוקראים לפונקציה של מערכת ההפעלה בכתובת קבועה. <u>הרגיסטרים ועופר לקרוא אותם רק עם פקודות EPC הם לא רגיסטרים רגילים ואפשר לקרוא אותם רק עם פקודות EPC מיוחדות</u>. את הרגיסטרים $\$k0,\,\$k1$ לא צריך לשמור כי הם משמשים רק את מי שנעצר. דוגמאות לשגיאות: Unknown instruction מערכת ההפעלה ולא את מי שנעצר. דוגמאות לשגיאות: (ID), Division by zero (EX), Unaligned address (MEM)

בעיות עם pipeline: • יכולים להיות כמה אקספשנים ב־stage ב־stage ב־stage ב־stage שונים. נתן עדיפות ל־stage מאוחר יותר. • צריך לבטל את כל האינסטרקשנים שונים. נתן עדיפות ל־branch misprediction שם הבאים, בדומה ל-branch misprediction • אם הרבאים, בדומה לקרונ.

pipeline לאורך exception הפתרון הכללי הוא להעביר את המידע על ה-writeback ולהתייחס אליו רק בשלב

Complex Pipeline

ב־Unified pipeline לכל פקודה לוקח אותה כמות סייקלים לסיים. אבל זה Non-unified ליותר מסובך ומעלה את כמות הstall ליותר מסובך ומעלה את כמות הstall ליותר מסובר pipeline, פקודות שונות יכולות לסיים בשלבים שונים.

אם פקודה קודמת לוקחת יותר זמן מפקודה שבאה אחריה WAW hazard אז יכול להיות שהפקודה הקצרה יותר כתבה לפני שהפקודה הארוכה הספיקה לעשות זאת. עד עכשיו היה בעיה רק עם RAW. כדי למנוע WAW נצטרך לדעת על כל רגיסטר מתי הוא יהיה מוכן לשימוש באמצעות טבלה:

Register	#0	ycl∈ 4	s til	l wr	iteback 1	In bypass (WB)	In RF
\$1						1	
\$2							1
\$31			1				

בכל סייקל נעדכן אותה בכך שנזיז 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).

Instruction- יכולות לרוץ מספר פקודות במקביל. Superscalar Pipeline נוציא: לדוגמה, ב־U-V Pipeline, לדוגמה, ב-level parallelism נוציא level parallelism. לדוגמה, ואז שאר השלבים נפרדים ל־U-pipe ול־U-pipe כאשר ער-pipe מוגבל יחסית ולא יכול להריץ הכל ומשומש רק בשביל פקודות בלתי עליות. פקודות מסובכות יכולות להשתמש גם ב־U וגם ב־V. ההחלטה האם להשתמש ב־U-pipe היא ב־U-pipe.

זיכרון וירטואלי

כתובת בזיכרון מפוצלת ל־Virtual Page Number ולי ויש טבלה Page Offset מכמבת בזיכרון מפוצלת ל־Virtual Page Number שממפה ב־RAM שממפה של פייג'ים. של מיפויים של פייג'ים.

m pageים מוכל ב־soverlapped TLB and cache access: אם מניחים ש־cache לגשת ל-cache אפשר להתחיל לגשת ל-offset

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

OOOE

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

במקרה של branch, על פי הפרדיקציה נמשיך 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.

המשתמשות באותו מספר רגיסטר זה False Dependencies: אם שתי פקודות משתמשות באותו מספר רגיסטר זה לא אומר בהכרח שיש תלות מידע ביניהן. לכן נוסיף register renaming: נשמור re שמפה כל רגיסטר פיזי re לרגיסטר פיזי re RAT שמפה כל רגיסטר פיזי זיינופר בשלב ה־retire נכתוב לרגיסטר הארכיטקטוני. בנוסף כדי לשחזר את ה־ROB בשלב ה־ROB, בכל ROB entry נשמור היסטוריה שאומרת איזה שינוי ROB פסקודה הזו.

הרבה מהפקודות ב־ROB סיימו לרוץ כבר, ומקום ב־Scheduling Queue הוא יקר כי צריך בכל סייקל למצוא איזה פקודות בו מוכנות ועבור כל 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 i מחכה לפקודה i כדי להיות מוכנה.

\mathbf{TLP}

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

