on the inputs.

MIPS ISA

רגיסטרים: השמות מתחילים ב־$. אפשר לסמן אותם או בתור $0, \ldots, $31 או לפי $s0 - $s7, $t0 - $t9, ... General Purpose Registers־ל בנוסף 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-Neaumann: 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: add $t0, $t1, $t2 ($t0 ← $t1 + $t2)

קידוד I-type:

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

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

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

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: ארגומנטים $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: lw $ra, ($sp) ; addi $sp, $sp, +4

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 ה־control מקבלת את 6 הביטים התחתונים (op) והעליונים (funct של R-type) ואת הפלט zero של ה־ALU. ומייצרת 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. control הוא מכונת מצבים.

Pipeline

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

סוגי hazardים:

• 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.

שערים לוגיים

	Gate	State	Input	Output
NMOS	1	On	**0**	**0**
			1	weak 1
	0 (any value other than 1)	Off	Any	Z

	Gate	State	Input	Output
PMOS	0	On	0	weak 0
			1	**1**
	1 (any value other than 0)	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 $x_1, \ldots, x_n$, אופרטורים: x' (not), $x + y$ (or), xy (and), $x \oplus y$ (xor), $x \downarrow y$ (nor), $x \uparrow y$ (nand).

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.

$$\text{HA}\left(x_0, y_0\right) \xrightarrow{C_0} \overset{S_1}{\text{FA}\left(x_1, y_1\right)} \xrightarrow{C_1} \ldots \overset{S_{n-1}}{\text{FA}\left(x_{n-1}, y_{n-1}\right)} \xrightarrow{C_{n-1}}$$

Ripple-Carry Subtractor: $(-y) = \left(y' + 1\right)$, so:

$$1 \rightarrow \overset{S_0}{\text{FA}\left(x_0, y_0'\right)} \xrightarrow{C_0} \overset{S_1}{\text{FA}\left(x_1, y_1'\right)} \xrightarrow{C_1} \ldots \overset{S_{n-1}}{\text{FA}\left(x_{n-1}, y_{n-1}'\right)} \xrightarrow{C_{n-1}}$$

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

תזמונים

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

t_{cd}: Contamination Delay. 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.

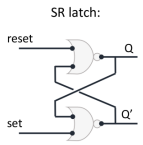
t_{ccq}: הזמן משינוי ב־Clk לשינוי ב־Q.

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

זמן ריצה: IC × CPI × Clock cycle.

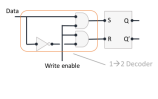
רגיסטרים

SR Latch



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 Latch (level triggered)



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

D Flip-Flop (edge triggered)

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

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.

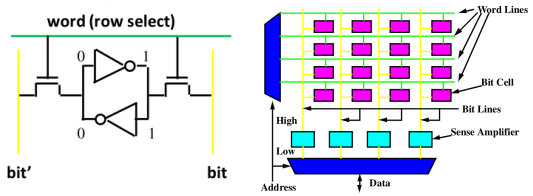
Moore and Mealy FSMs: In a Moore FSM, outputs only depend on current state, in Mealy FSM, outputs depend on the current state and

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 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־prediction. נשמור ב־Branch Target Buffer.
- Backward jump predictor: אם ה־branch קופץ אחורה נניח שלוקחים אותו. זה מייעל לולאות והרבה יותר פשוט מ־dynamic branch prediction.
- Bimodal predictor: מכונת מצבים עם 4 מצבים לכל BTB entry: strongly/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.



זיכרון SRAM: בכתיבה, נשים ערך x ב־bit וערך x' ב־bit' ונבחר את השורה. בקריאה, נטען גם את bit וגם את bit' עם ערך $V_{dd}/2$, נבחר את השורה, ונמדוד מה ההפרש בין bit ל־bit'. משמש בתור cache.

זיכרון DRAM: כל תא הוא capacitor עם transistor עליו. בכתיבה נשים ערך x ב־bit ונבחר בשורה ואז ה־capacitor נטען או מוציא את התוכן שלו. בקריאה נטען ערך $V_{dd}/2$, 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.

Cache

סוגי לוקאליות: • 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.

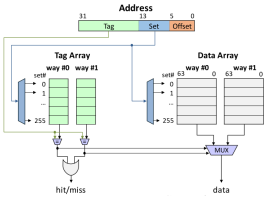
2-Way Associative:

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)

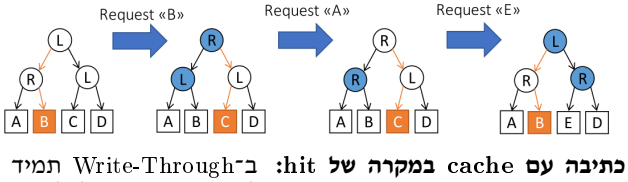


ב־tag array נצטרך לשמור עבור כל set ו־way את ה־tag. בנוסף ביט valid ובמקרה של Write-Back גם ביט Dirty. בנוסף ביטים של psuedo-LRU אם יש pseudo-LRU.

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

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.



כתיבה עם cache במקרה של hit: ב־Write-Through תמיד נעדכן גם את הזיכרון וגם את ה־cache. זה טוב ל־coherency אבל לוקח יותר memory bandwidth. ב־Write-Back נעדכן רק את ה־cache ונעדכן את הזיכרון ב־eviction (על פי ביט dirty). זה מוריד את ה־latency של כתיבות אבל עלול לשבור coherency.

כתיבה עם cache במקרה של miss: ב־Write allocate נטען את כל הבלוק ל־cache (בדומה ל־read miss) ורק אז נכתוב ל־cache. זה מתאים ל־Write-Back. 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 חוצה שני cache lineים. ב־MIPS יש דרישה שכל פעולת זיכרון חייבת להיות aligned ל־cache line.

Exceptions

לאחר שגיאה, שומרים את PC ב־EPC register, סוג השגיאה נשמר ב־Cause register, וקוראים לפונקציה של מערכת ההפעלה בכתובת קבועה. הרגיסטרים EPC ו־Cause הם לא רגיסטרים רגילים ואפשר לקרוא אותם רק עם פקודות מיוחדות. את הרגיסטרים $k0, $k1 לא צריך לשמור כי הם משמשים רק את מערכת ההפעלה ולא את מי שנעצר. דוגמאות לשגיאות: Unknown instruction (ID), Division by zero (EX), Unaligned address (MEM).

בעיות עם pipeline: • יכולים להיות כמה אקספשנים ב־pipeline ב־stageים שונים. נתן עדיפות ל־stage מאוחר יותר. • צריך לבטל את כל האינסטרקשנים הבאים, בדומה לbranch misprediction • אם הexception קרה בbranch שלא לקחנו אז הוא לא קרה.
הפתרון הכללי הוא להעביר את המידע על ה־exception לאורך ה־pipeline ולהתייחס אליו רק בשלב writeback.

Complex Pipeline

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

WAW hazard: אם פקודה קודמת לוקחת יותר זמן מפקודה שבאה אחריה 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	#cycles till writeback					In bypass (WB)	In RF
	5	4	3	2	1		
$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).

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 ומשומש 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 ב־RAM שממפה Virtual Page Number לכתובת פיזית. יש TLB שהוא cache של מיפויים של 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 ממופים 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 אז היא מוכנה לרוץ, אחרת נוסיף אותה לרשימת הפקודות המחכות של המקורות. כשפקודה מסיימת לרוץ היא מעדכנת את המשתמשים שלה והם יכולים להפוך למוכנים.

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.

False Dependencies: אם שתי פקודות משתמשות באותו מספר רגיסטר זה לא אומר בהכרח שיש תלות מידע ביניהן. לכן נוסיף 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 אחרי 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.

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.