פרק 4 בספר – ביצועי מעבד והערכת ביצועים

CPU = זמן הפעולה של תוכנית.

CPU Time = IC × CPI × CCT

10 nsec clock cycle => 100 MHz clock rate
5 nsec clock cycle => 200 MHz clock rate
2 nsec clock cycle => 500 MHz clock rate
1 nsec clock cycle => 1 GHz clock rate
500 psec clock cycle => 2 GHz clock rate
250 psec clock cycle => 4 GHz clock rate
200 psec clock cycle => 5 GHz clock rate

IC = מס פקודות בתוכנית **(ללא יח')**

CPI = מס מחזורים לפעולה בממוצע **(ללא יח')**

CCT = הזמן שנמשך מחזור אחד בשניות. $\frac{1}{CR}$ = CCT

CPU clock cycles = IC x CPI

Hardware or software component	Affects what?	How?
Algorithm	Instruction count, possibly CPI	The algorithm determines the number of source program instructions executed and hence the number of processor instructions executed. The algorithm may also affect the CPI, by favoring slower or faster instructions. For example, if the algorithm uses more floating-point operations, it will tend to have a higher CPI.
Programming language	Instruction count, CPI	The programming language certainly affects the instruction count, since statements in the language are translated to processor instructions, which determine instruction count. The language may also affect the CPI because of its features; for example, a language with heavy support for data abstraction (e.g., Java) will require indirect calls, which will use higher-CPI instructions.
Compiler	Instruction count, CPI	The efficiency of the compiler affects both the instruction count and average cycles per instruction, since the compiler determines the translation of the source language instructions into computer instructions. The compiler's role can be very complex and affect the CPI in complex ways.
Instruction set architecture	Instruction count, clock rate, CPI	The instruction set architecture affects all three aspects of CPU performance, since it affects the instructions needed for a function, the cost in cycles of each instruction, and the overall clock rate of the processor.

איטי חלקי המהיר

$$\text{Speedup} = \frac{\text{CPU\_time}(S)}{\text{CPU\_time}(F)}$$

צריך לזכור ש- CPU מושפע מהרבה גורמים, ולכן לא תמיד קוד קטן מהיר יותר. הקוד תלוי גם בפקודות עצמן ובזמן הביצוע שלהן.

חוק אמדל

הכוונה בחוק זה היא להפוך את הפעולות הנפוצות ביותר בקוד ליעילות יותר ומהירות יותר.

$$ExTime_{new} = ExTime_{old} \times \left[(1 - Fraction) + \frac{Fraction}{Speedup} \right]$$

- $ExTime_{new}$: זמן ריצה **חדש** של תוכנית כלשהי
- $ExTime_{old}$: זמן ריצה **ישן** של תוכנית כלשהי
- $Fraction$: אחוז שורות הקוד הכולל שהושפעו מהשינוי
- $Speedup$: כל אחוז שורות קוד שהושפעו מהשינוי חלקי פי כמה הושפעו.

  במידה ויש כמה לחבר את כל השברים בסוף.

את ה SPEEDUP ניתן לחשב:

$$Speedup_{overall} = \frac{ExTime_{old}}{ExTime_{new}} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

ניתן להכליל גם לחישוב CPI:

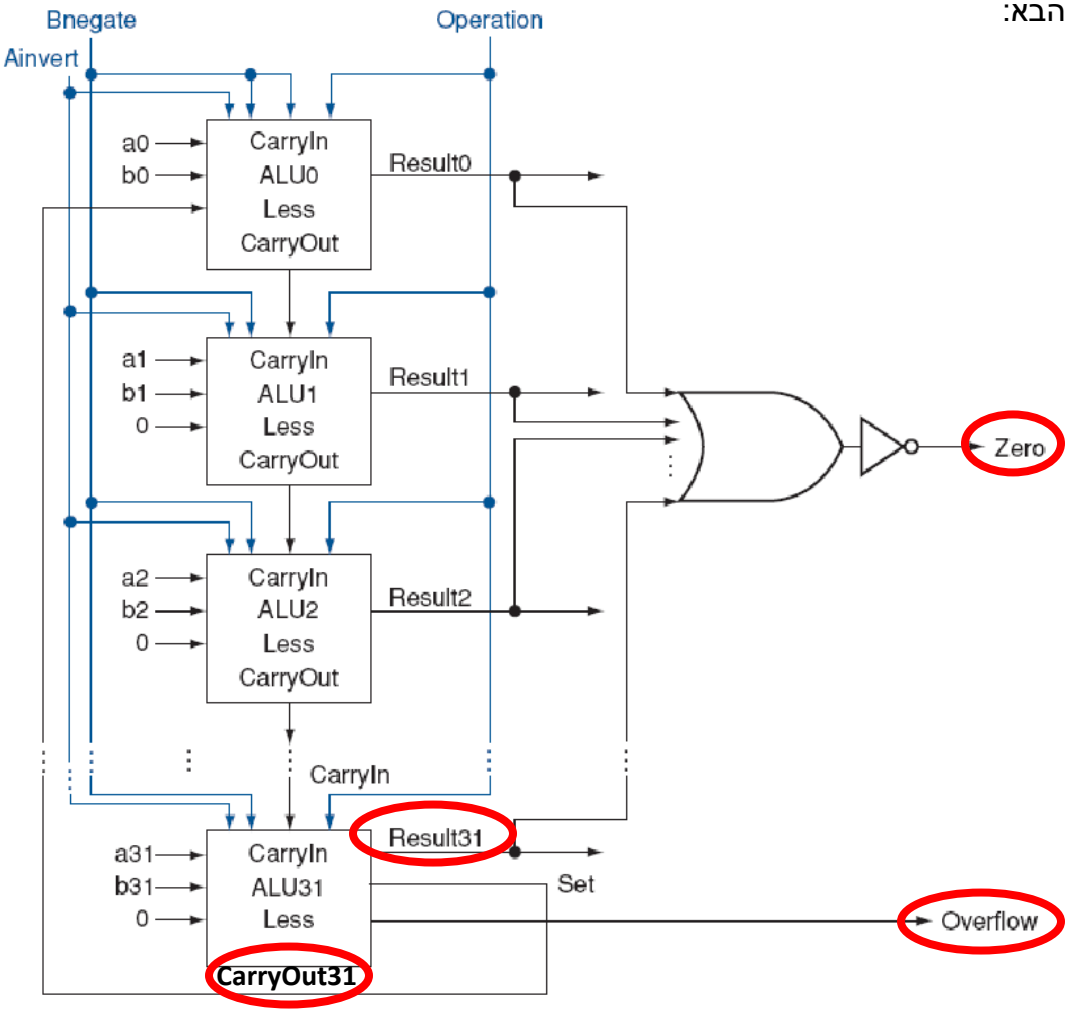
$$CPI_{new} = CPI_{old} \times \left[(1 - Fraction) + \frac{Fraction}{Speedup(in\ cycles)} \right]$$

פרק 5 בספר – מעבד חד מחזורי

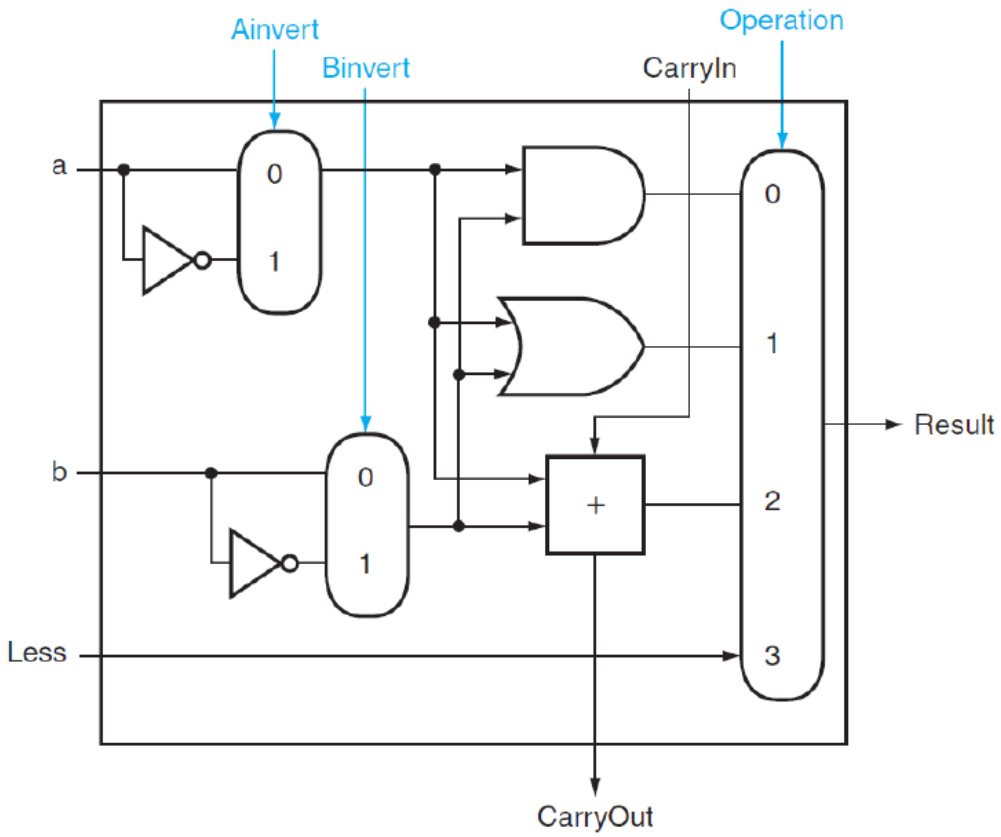
ALU – בקרה משנית "הביצה הקטנה"

כיצד בנוי ה- ALU?

ה- ALU היא יחידת החישוב הפנימית של המעבד.
בנויה באופן הבא:



כאשר כל ריבוע בנוי כך:



ארבעת העיגולים האדומים בתרשים הקודם מסמלים את הדגלים המושפכים מפעולות ה ALU.

- **ZERO** = מקבל 1 במידה והתוצאה של פעולת ה ALU הינה 0. עובד עם BRANCH
- **CARRY** = אינדיקציה לגלישה לפי שיטת ללא סימן. זהו ה carryout31
- **SIGN** = סיבית הסימן MSB. זהו Result31
- **OVERFLOW** = גלישה לפי משלים ל-2. האינדיקציה: Carryout31 ⊕ Carryin31

צריך לזכור כי הפעולה הארוכה ביותר קובעת זמן התייצבות. במקרה זה זוהי הפעולה SLT. היא בודקת האם A > B ע"י SET = SIGN ⊕ OVERFLOW

ההבדל בין ADD ל – ADDU הוא ש- ADD מתייחס לדגל OVERFLOW ו- ADDU לא.
ולכן במידה ויש גלישה, התוכנית תקרוס.

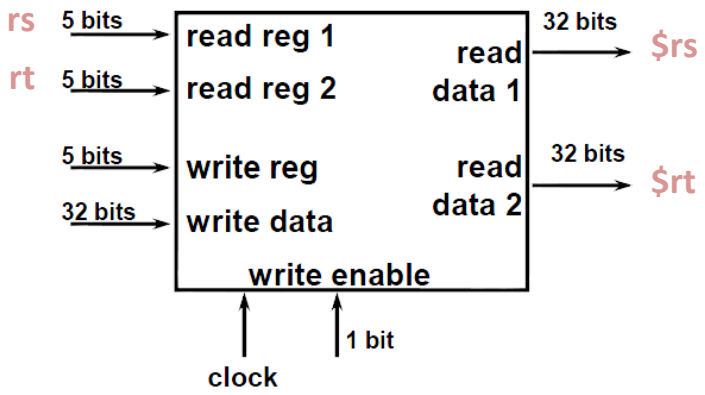
ה- ALU הוא רכיב שמקבל 4 ביטים ובעזרתם נקבעת הפעולה שיש לבצע :

הביטים שהתקבלו				הפעולה לבצע	
A	B				
0	0	0	0	AND	
0	0	0	1	OR	
0	0	1	0	חיבור	פעולה אריתמתית
0	1	1	0	חיסור a - b	פעולה אריתמתית
0	1	1	1	SLT	פעולה אריתמתית
1	1	0	0	NOR	
1	1	0	1	NAND	

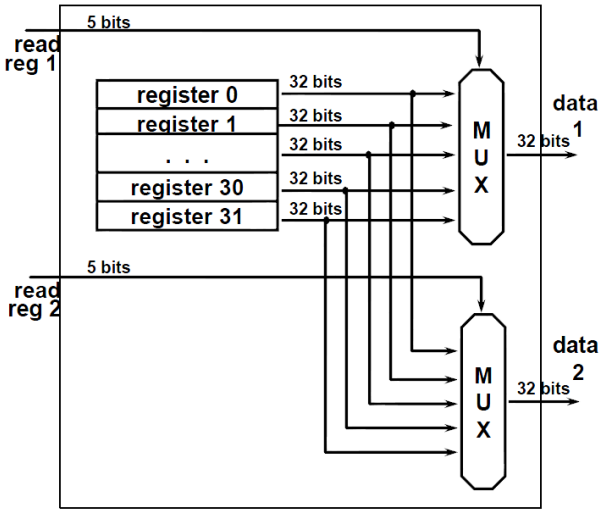
R-type	Op	Rs	Rt	Rd	Shamt	Funct
I-type	Op	Rs	Rt	Address offset		

Instruction	Op	Funct	(ALUOp)	ALU ctrl	Function
lw	35	-	00	010	ADD
sw	43	-	00	010	ADD
beq	4	-	01	110	SUB
addu	0	33	10	010	ADD
sub	0	34	10	110	SUB
and	0	36	10	000	AND
or	0	37	10	001	OR
slt	0	42	10	111	SLT

מקבץ האוגרים



כל האוגרים נמצאים בזיכרון (0 עד 31) וכדי לקרוא מידע מאחד מהם, קיים MUX אשר בעזרתו מגיעים לאוגר הרצוי:



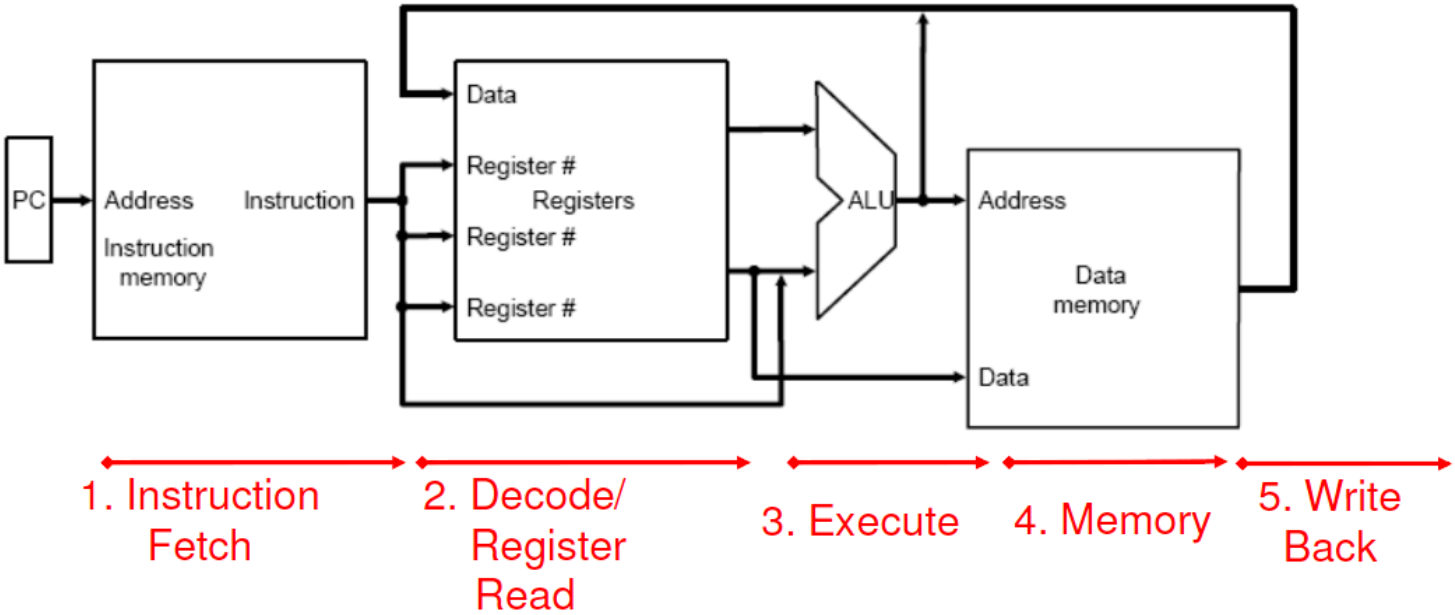
כתיבה לאוגר:

ברגע שמסנכרנים את מקבץ האוגרים לפעימת שעון, המידע נכתב בעת ירידת שעון. כמו כן צריך שהכניסה WRITE ENABLE תהיה ENABLE.

כתיבה לאוגר 0 אינה אפשרית.
כל פקודה יכולה לעדכן ערך אחד.

מעבד חד מחזורי

שלבי ביצוע הוראות מכונה:

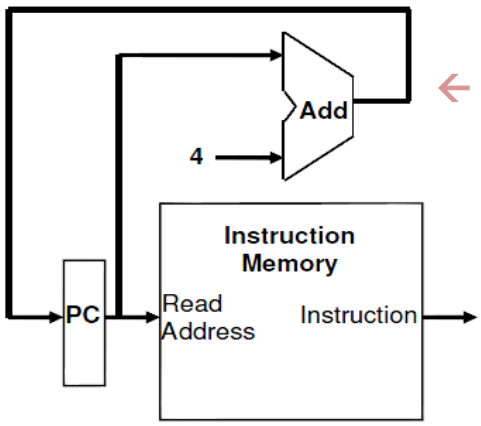


1. **FETCH** = הבאת פקודה מהזיכרון עפ"י כתובת PC
2. **DECODE** = פיענוח הפקודה וקריאת האוגרים הנחוצים
3. **EXECUTE** = חישוב התוצאה או כתובת רצויה בעזרת ה-ALU
4. **MEMORY** = השתמש בתוצאה לבצע קריאה או טעינה לזיכרון
5. **WRITE BACK** = כתיבה חזרה למקבץ האוגרים

שני השלבים הראשונים זהים בכל הפקודות. אך לא כל פקודה זקוקה לשאר שלושת השלבים.

- שלב 1 - FETCH

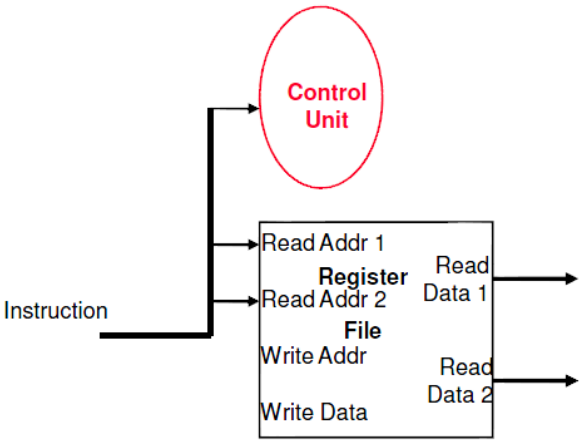
קריאת פקודה מזיכרון הפקודות
עדכון PC לכתובת הפקודה הבאה



תפקיד ה-ADD הוא חישוב הכתובת הבאה ←

- שלב 2 - DECODE

העברת שדה ה opcode ושדה ה function ליחידות הבקרה.



קידוד הפקודות

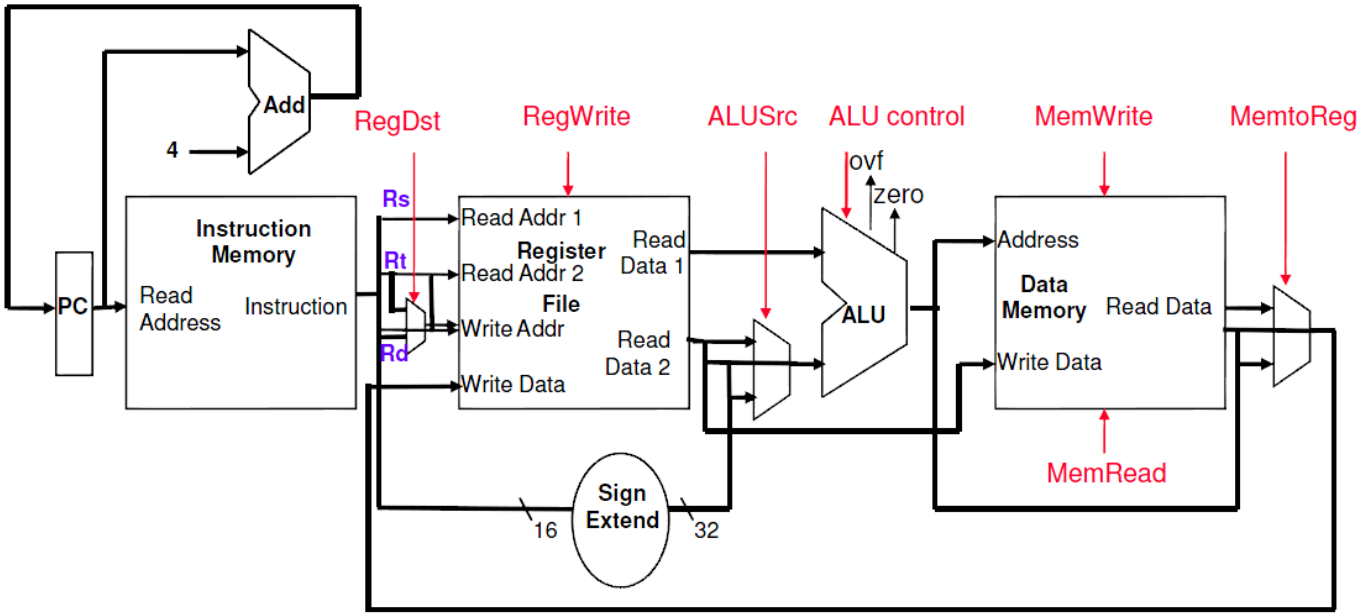
פקודות מסוג R

Opcode	Rs	Rt	Rd	Shift amount	func
6bit	5bit	5bit	5bit	5bit	6bit

פקודות מסוג I

Opcode	Rs	Rt	Address \ Immediate
6bit	5bit	5bit	16 bit

נתיב נתונים משותף:



קווי בקרה של איפשור – מקבלים ביט בודד (0 או 1)

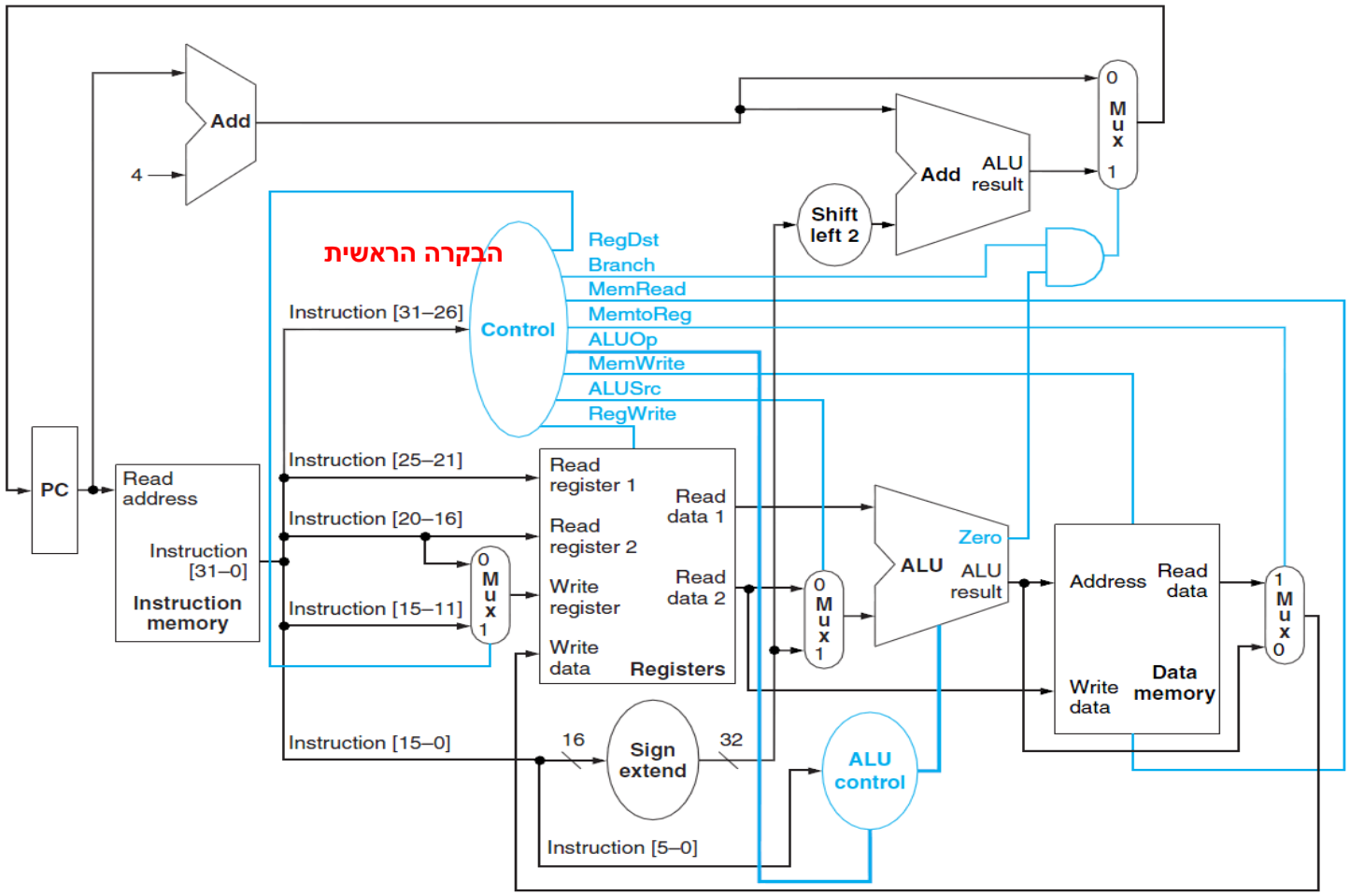
- RegWrite = האם תהיה כתיבה למקבץ האוגרים פעיל ב R-TYPE וכו'
- MemWrite = האם תהיה כתיבה לזיכרון נתונים פעיל ב- SW
- MemRead = האם תתבצע קריאה מזיכרון הנתונים פעיל ב- LW

קווי בקרה בוררים - במרבב

- RegDst = בורר לאיזה אוגר נכתוב במקבץ האוגרים. ALU Result או ReadData
- MemToReg = בורר איזה מידע יכתב למקבץ האוגרים. Rt (lw) או Rd (R-Type)
- ALUSrc = בורר איזה ערך יכנס ל- ALU כאופרנד שני. sign-extended או $rt (R-Type) lower 16 bits (lw,sw)

ללא REDWRITE אין MEMTOREG או REGDST

בקרה ראשית – "הביצה הגדולה"



הבקרה הראשית מוציאה את קווי הבוררים (לפי התמונה).
היא מקבלת 6 ביטים שהם ה OPCODE ובהתאם לכך מדליקה (שולחת 1) את קו הבורר הרלוונטי.

הבקרה הראשית מחוברת לבקרה המשנית. תפקידה של המשנית הוא לשלוט על ה-ALU. היא מקבלת מהבקרה הראשית ALUop0 , ALUop1 והיא מחוברת גם ל-FUNC.
הביצה הקטנה מתחברת ל- ALU דרך :

- 2 הביטים המחליטים על הפעולה
- Ainvert
- B-nagate

אלו הם 4 קווי הבקרה הנקראים ALU Control

חלק מקווי הבקרה ומשמעותם בהינתן ביט כלשהו :

Signal Name	Effect when 0	Effect when 1
MemRead	None.	Data Memory contents at the *read address* are put on *read data* output.
MemWrite	None.	Data memory contents at address given by write address are replaced by value on *write data* input
ALUSrc	The second ALU operand comes from the second register file output.	The second ALU operand is the sign-extended lower 16 bits of the instruction.
RegDst	The register destination for the *register write* comes from the **rt** field.	The register destination number for the *register write* comes from the **rd** field.
RegWrite	None.	The register given by *write register* number input is written into with the value on *the write data* input.
PCSrc (== Branch & cond)	The PC is replaced by the output of adder that computes the value of PC+4.	The PC is replaced by the output of the adder that computes the branch target.
MemtoReg	The value fed to the *register write* data input comes from the ALU.	The value fed to the *register write* data input comes from the data memory.

סיכום :

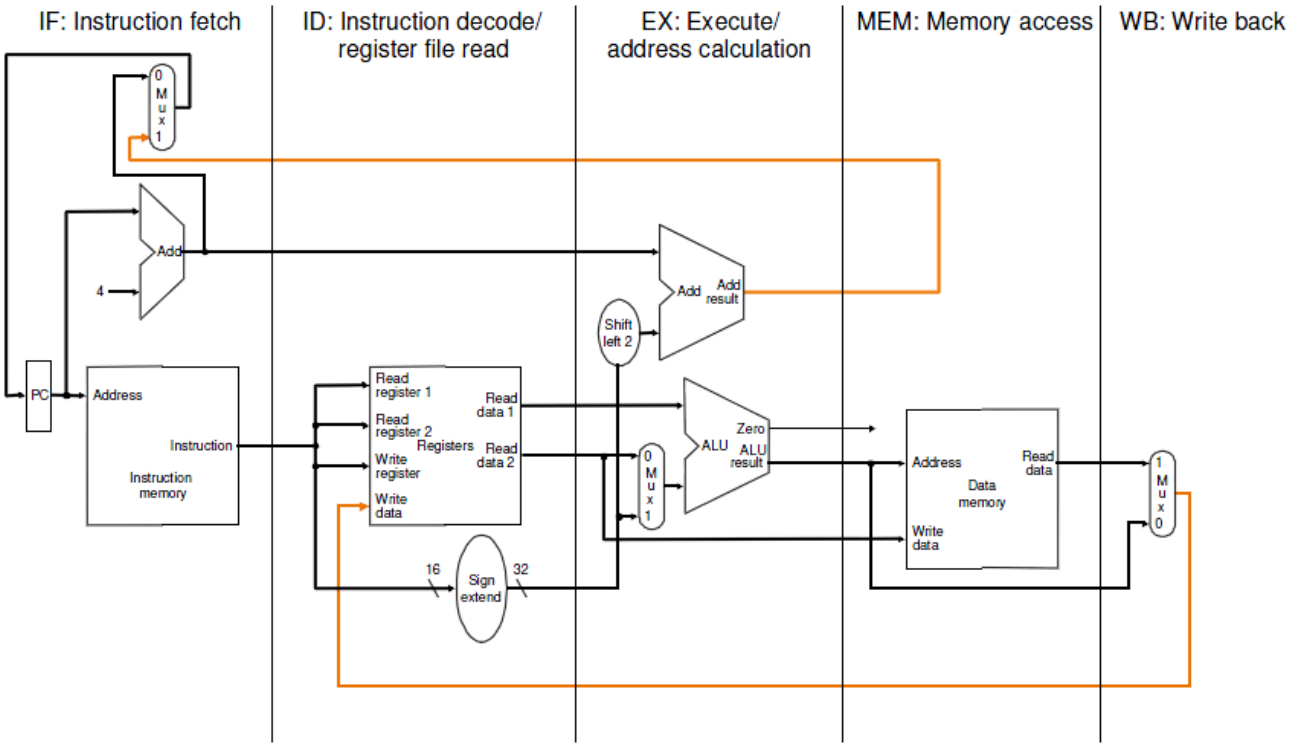
Inst.	OpCode	RegDst	ALUSrc	MemTo -Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUop	Jump
R-Type	0	1	0	0	1	0	0	0	10	0
LW	35	0	1	1	1	1	0	0	00	0
SW	43	X	1	X	0	0	1	0	00	0
Beq	4	X	0	X	0	0	0	1	01	0
J	0	X	X	0	0	0	0	X	XX	1
addi	8	0	1	0	1	0	0	0	00	0

פרק 6 בספר – הצנרה PIPLINING

שיפור ביצועים באמצעות הצנרה:

חלוקת מסלול הנתונים ל-5 שלבים הופכת אותו ל"רב מחזורי" :

- IF
- ID
- EX
- MEM
- WB



- **IF** – בו נקראת הפקודה מהזיכרון אל האוגר IR
- **ID** – בו מפוענח ה-OPCODE של הפקודה, נקרא מידע ממקבץ האוגרים ומחושבת כתובת צפויה לקפיצה.
- **EX** – בו ה-ALU מבצע את הפקודה הספציפית.
- **MEM** – בו נעשית הפניה לזיכרון בפקודות LW ו- SW.
- **WB** – בו נכתב ערך לאחד האוגרים.

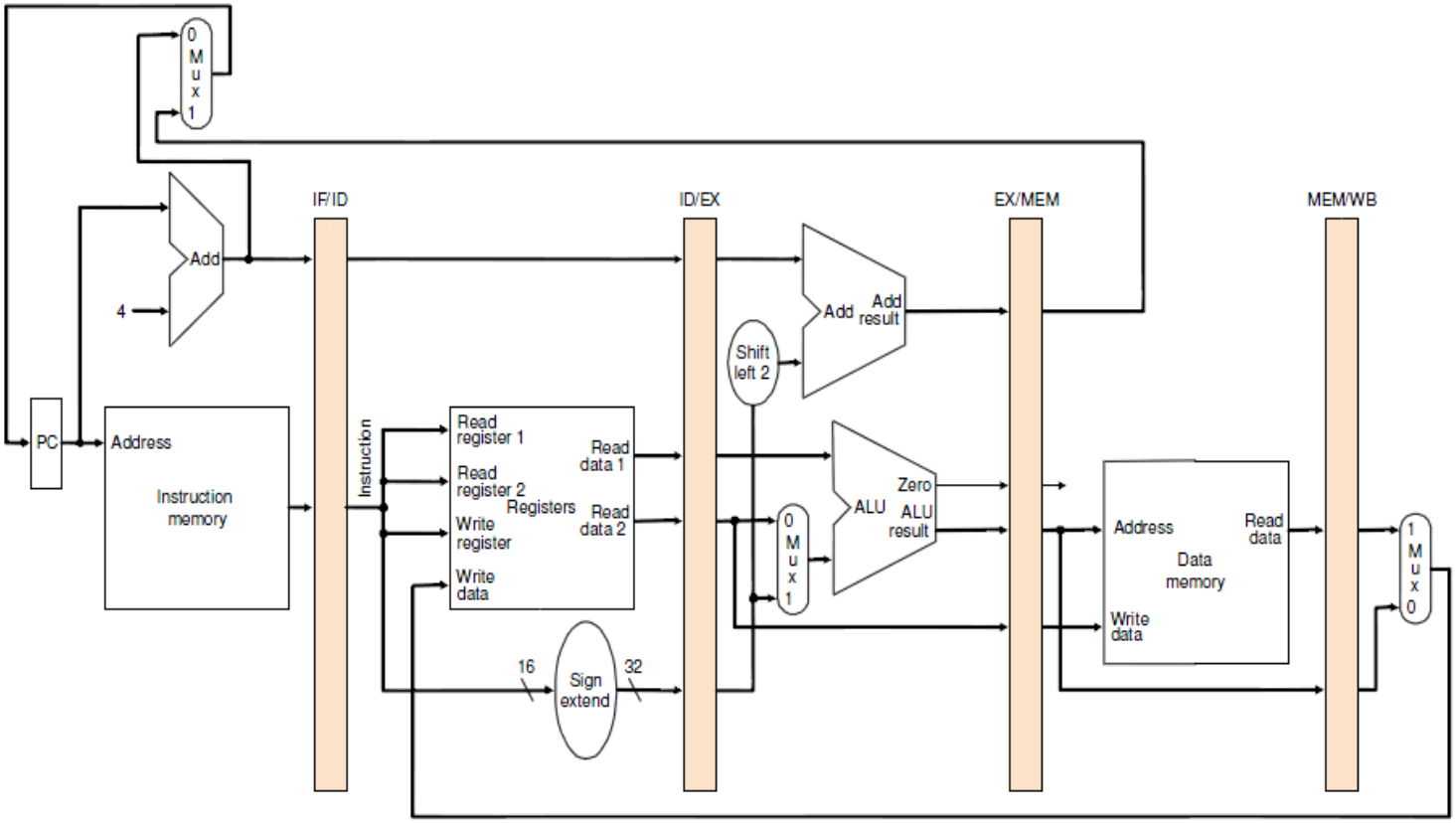
על מנת שהמידע יזרום באופן מבוקר ותהליכים לא יתערבבו, מוסיפים אוגרי צנרת שמונעים מעבר מידע. כל האוגרים הללו מחוברים לאותו השעון ובסוף פעימת שעון(נפילת שעון) המידע נכתב עליהם.

באוגרים הללו יש את כל המידע שעלול להיות הכרחי להמשך הפקודה. האוגר IF/ID (האוגר בין השלבים IF ו- ID) הוא של 64 ביט.

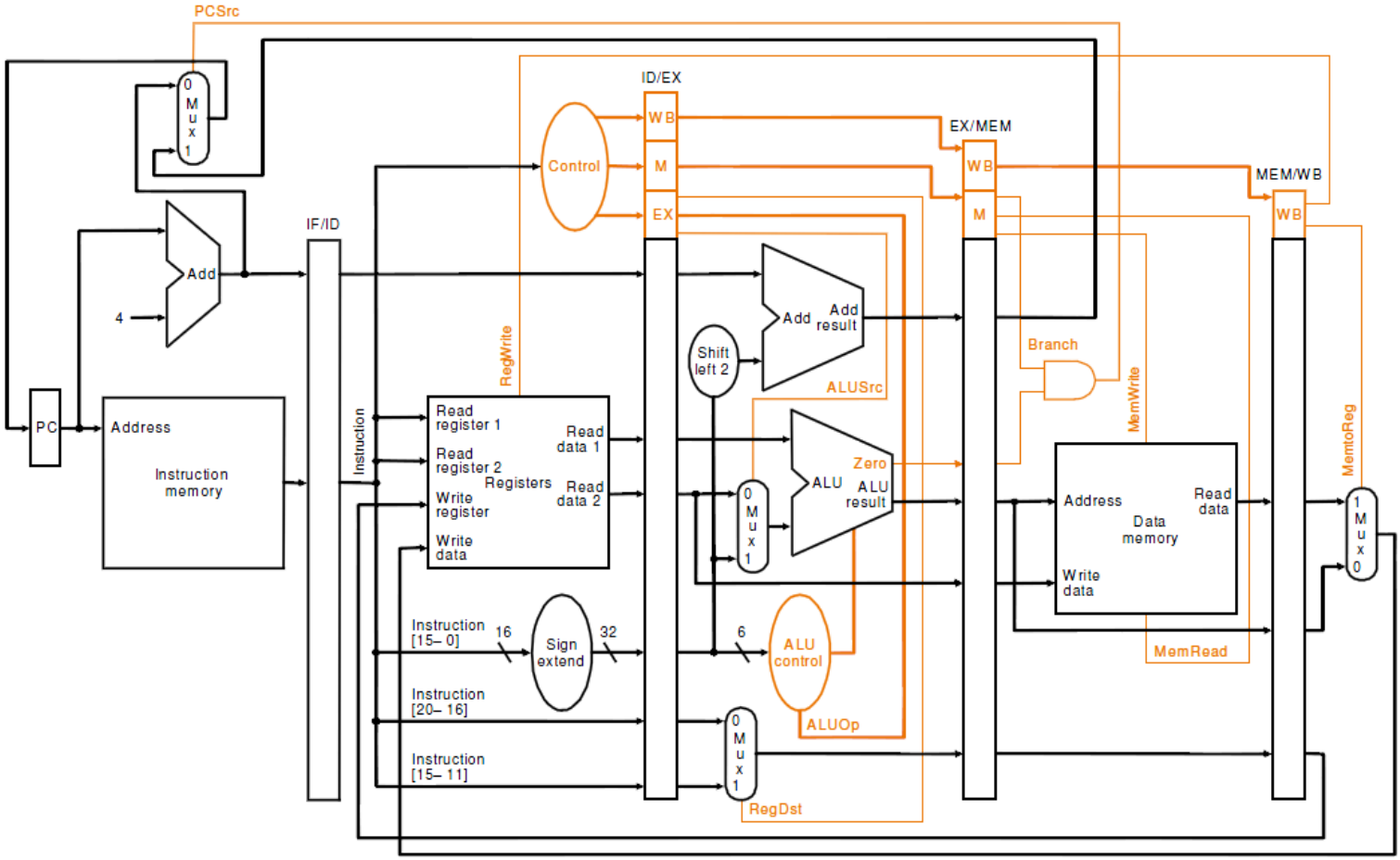
מידע שיכול להיות על האוגרים הוא PC+4 במידה והפקודה היא J

במידה ויש צורך במידע בשלבים מאוחרים של התהליך, המידע יעבור מאוגר לאוגר עד השימוש בו.

עד פעימת השעון השנייה המעבד לא יודע באיזו פקודה מדובר, לכן לא ניתן לקבוע עדיין שום קו בקרה.



נתיב הנתונים כולל ההצנרה:

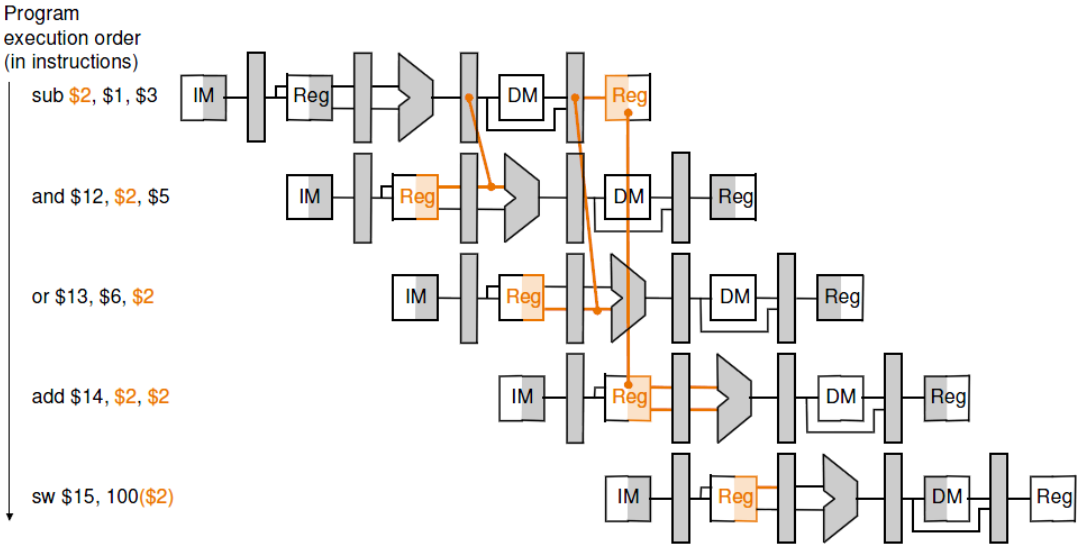


סיכונים בהצנרה:

- **סיכון מבני –** 2 פעולות מנסות להשתמש באותו המשאב. (אוגר וכו')
- **סיכון נתונים –** ניסיון להשתמש במידע לפני שהמידע מוכן. במקרה זה ניצור FORWARDING או במידה ולא ניתן, נעשה השהייה STALL.
- **סיכון בקרה –** ניסיון להחליט החלטה לגבי המשך התוכנית לפני שהתנאי נבחן וה- PC הבא חושב.

כדי לתקן סיכון נתונים, לאחר פקודה הכותבת לאוגר כלשהו שצריכים בפקודה הבאה נרשום 3 פעמים NOP – מספיק כדי שהמידע יתייצב באוגר.

כדי שנוכל לעקוף את ה NOP נשתמש בעיקרון FORWARDING (גניבת ערכים). נשמור את ערך האוגר הרצוי או המידע שצריך באוגרי הצנרת. כך במידה ונקרא מידע לא מעודכן או לא מוכן, נחפש בכל סוף שלב באוגרי הצנרת האם יש מידע מעודכן יותר.



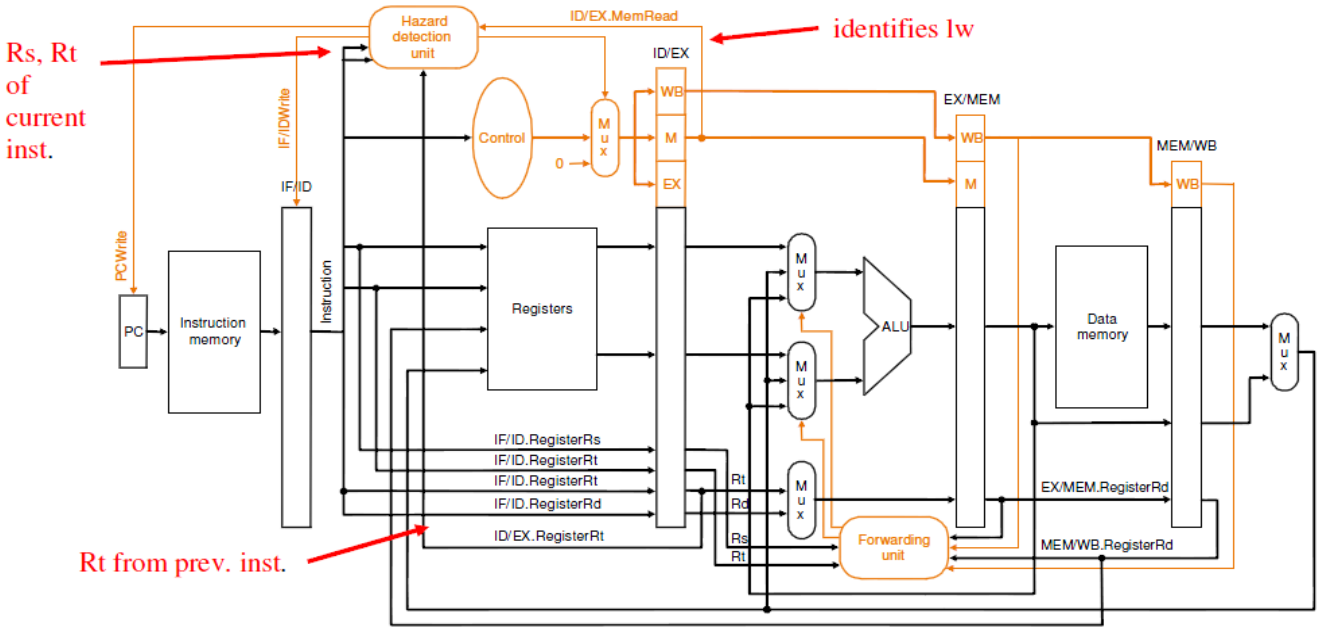
יחידת איתור סיכונים HDU :

ההשהיה בחומרה נעשית במעבד ע"י יחידת איתור הסיכונים.

לאחר שה- HDU מזהה **סיכון נתונים** היא מבצעת את הפעולות הבאות:

1) נתינת ערך 0 לכל קווי הבקרה שיוכנסו לאוגר ID/EX משמע הופכת את הפקודה ל NOP
2) שמירת הערך באוגר IF /ID כדי שהפקודה שיושבת שם תבוצע עוד פעם (נעשה ע"י הקו IF/IDwrite אשר לא מאפשר כתיבה לאוגר IF/ID)
3) ע"י ההקפאה של IF/ID ביצענו שוב את הפקודה – כלומר גם ערך PC ישמר. זה יעשה ע"י הקו PCwrite.

במקרה זה המעבד מפסיד מחזור שעון אחד. במקרה כזה אומרים שיש "בועה" בצנרת.
התנאי להשהייה הוא: ((ID/EX.Rt= =IF/ID.Rt) || (ID/EX.Rt= =IF/ID.Rs)) &&(ID/EX.MemRd)



במקרה של **סיכון בקרה** (מתרחשים כאשר במקום מסויים בתוכנית יש קפיצה לכתובת אחרת) מכיוון שהקפיצה עצמה תעשה רק במחזור השעון הרביעי, יספיקו להיכנס לצנרת עוד 3 פקודות שכולן מיותרות ואין לבצע אותו במקרה של קפיצה.

במקרה של סיכון בקרה, לא מספיק רק עיכוב בביצוע הפקודות (כמו בסיכוני נתונים) אלא יש לבטל את ביצוע פקודות המכונה המיותרות שהמעבד התחיל לבצע.

FLUSH :

תהליך זה נקרא FLUSH. כלומר, צריך לדאוג שה- PC ימשיך להתקדם אך הפקודות שהחלו להתבצע לא יכתבו לשום אוגר או לזיכרון. הפקודות המיותרות ימשיכו להתקדם בצנרת אך לא יגרמו לשום שינוי, והם למעשה "ישטפו" החוצה.

כל פקודה מבוטלת מעכבת את הצנרת במחזור שעון אחד. אם הקפיצה הייתה מתבצעת במחזור שעון מוקדם יותר, נוכל לבטל פחות פקודות ולשפר את ביצועי המעבד.

פסיקות :

פסיקה היא אות המתקבל במעבד מרכיב חומרה או תוכנה ומאפשר לשנות את סדר ביצוע הפקודות בתוכנית מחשב שלא על ידי בקרה מותנית .בעת קבלת הפסיקה משהה המחשב את ביצועה הסדרתי של התוכנית, כדי להפעיל שגרת טיפול בפסיקה. לאחר הטיפול, ממשיך המחשב בביצוע הסדרתי של התוכנית. פסיקות משמשות כאמצעי תקשורת בין תהליכים במחשב ופסיקות תוכנה נמצאות בשימוש נרחב במחשבים הפועלים בריבוי משימות .

בעת הפעלת תוכנית טוענת מערכת ההפעלה את התוכנית לזיכרון. לאחר הטענת התוכנית לזיכרון מבוצעות הפקודות באופן סדרתי. ביצוע כל פקודה נקרא "מחזור עבודה", והוא מורכב משלושה חלקים עיקריים:

- **מחזור פסיקה -** במחזור הפסיקה בודק המעבד האם קיימות פסיקות הממתינות לטיפולו, ובמקרה שכן, הוא מטפל בפסיקה וממשיך במחזור העבודה.
- **מחזור הבאה -** המעבד קורא לתוך האוגר MBR את הפקודה הנמצאת בכתובת הזיכרון שבמונה הפקודות ומקדם את מונה הפקודות לכתובת הבאה.
- **מחזור ביצוע -** הפקודה לביצוע מנותחת ומבוצעת על ידי המעבד.

המחשב מבצע סדרת פעולות זו בלולאה. כאשר ישנה הפרעה לביצוע התוכנית הרגיל, הנגרמת על ידי אירוע לא-צפוי המעורר פסיקה (כגון הקשה על תו במקלדת או חלוקה באפס) עובר המחשב לביצוע מחזור פסיקה לאחר מחזור הביצוע (ולפני מחזור ההבאה) ומטפל בפסיקה. מטרת הפסיקות היא לאפשר טיפול באירועים, תקינים או בלתי תקינים, שאינם חלק מהביצוע הסדרתי של התוכנית.

בMIPS מערכת הפסיקות מגיבה גם ל-ERRORS פנימיות כתוצאה מהרצה וגם מפלט/קלט. יש חלק ב-CPU (בMIPS) הנקרא coprocessor0 שתפקידו לשמור את כל המידע הנחוץ לטיפול בפסיקה או בחריגה.

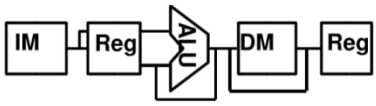
האוגרים של coprocessor0:

Register name	Register number	Usage
BadVAddr	8	Memory address at which an offending memory reference occurred
Status	12	Interrupt mask and enable bits
Cause	13	exception type and pending interrupt bits.
EPC	14	Address of instruction that caused exception

ברגע שמזוהה שגיאה, אנחנו קופצים לחלק קוד בקרנל שאחראי על טיפול בשגיאות. שם או שהתוכנית מפסיקה לפעול או שמתבצעת פעולה.

בצנרת, ברגע שמתגלה שגיאה, הפעולה שגרמה לשגיאה מופסקת באמצע. כל הפעולות שלפניה מתבצעות כרגיל, וכל הפעולות שנכנסו לצנרת אחריה ישטפו החוצה (flush). אחד האוגרים יכיל את הסיבה לשגיאה, שומרים את הכתובת של הפעולה שגרמה לשגיאה וקופצים לכתובת קבועה מראש (כתובת ה - exception handler בקוד)
התוכנה בודקת מה הסיבה לשגיאה ומטפלת בה בהתאם.

היכן מתגלות שגיאות בצנרת?



	Stage(s)?	Synchronous?
❑ Arithmetic overflow	EX	yes
❑ Undefined instruction	ID	yes
❑ TLB or page fault	IF, MEM	yes
❑ I/O service request	any	no
❑ Hardware malfunction	any	no

❑ Beware that multiple exceptions can occur simultaneously in a *single* clock cycle

מכיוון שחריגות יכולות להיווצר במחזורי שעון שונים , לעיתים יכול להיווצר מצב בו שתי חריגות מגיעות בעת ובעונה אחת משתי פקודות מכונה שנמצאות כעת בצנרת . במקרה כזה צריך ליצור מנגנון עדיפויות בין החריגות. הב-MIPS המעבד מעדיף במקרה זה את החריגה שנגרמה על ידי הפקודה שנכנסה מוקדם יותר אל הצנרת.

זיכרון מטמון

יש 2 זכרונות נדיפים (נמחקים כשהמחשב נכבה) : DRAM ו- SRAM (דינאמי וסטטי בהתאמה).
השוני בטכנולוגיות אלו הוא ש- SRAM יותר יקר,מהיר וצורך אנרגיה מ-DRAM.

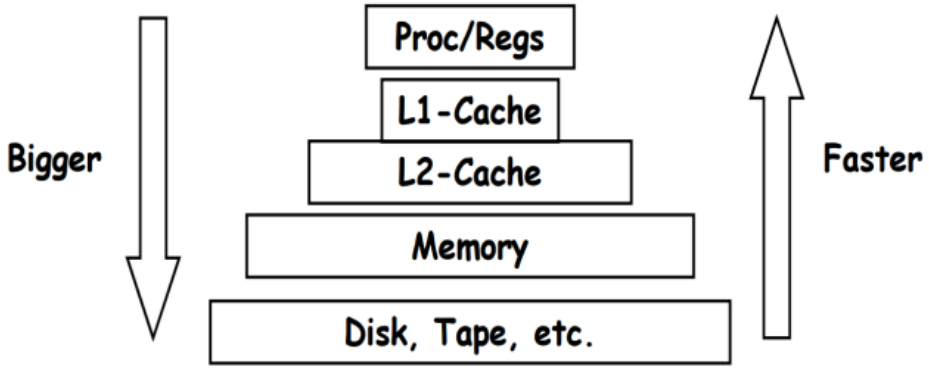
עקרונות של לוקאליות בזמן ובמרחב

- **לוקאליות בזמן** – אם ניגשים לנק' בזיכרון יש סיכוי גבוה לגשת לנק' זו שוב. **למשל לולאות בקוד**
- **לוקאליות במרחב** – אם ניגשים לנק' מסוימת בזיכרון יש סיכוי גבוה לגשת לנק' הצמודה אליה. **למשל מערכים**

עקרונות אלו מתבססים על כך שהתוכניות צפויות בדרכ. על סמך עקרונות אלו רוצים לייצר אשליה שהזיכרון מהיר וגדול.
לכן נשתמש בהיררכיה של זיכרון.
ההיררכיה בנויה כפירמידה, כל ה"משחק" נעשה ב-3 רמות, כאשר כל רמה מוכלת ברמה שמתחתיה.
רמה 1 מוכלת ברמה 2, רמה 2 מוכלת ברמה 3 וכו'.

עפ"י לוקאליות בזמן ובמרחב נעלה את המידע הכי רלוונטי לרמה הכי גבוהה, מכיוון שככל שעולים למעלה (יורדים ברמות) הגישה יותר מהירה. אך מצד שני ככל שיורדים למטה (עלייה ברמות) יש יותר קיבולת זיכרון.

הזיכרון מטמון (CACHE) נוצר מכיוון שה-CPU מהיר בהרבה מ- DRAM והגישה הישירה ל-DRAM מבזבזת זמן (פעימות שעון) ולכן הזיכרון מטמון מהווה זיכרון קטן אך מהיר יותר. כך נוצרת האשליה שהגישה מהירה יותר.



מונחים מרכזיים בזיכרון מטמון:

- **HIT** – המידע נמצא בזיכרון המטמון.
- **MISS** – המידע לא נמצא בזיכרון מטמון. (נקבע עבור כל רמה באופן ספציפי)
- **BLOCK** – שורה בזיכרון מטמון. הגודל המינימלי הוא מילה, אך לפי לוקאליות במרחב, בלוק גדול ממילה.
- **MISS RATE** – אחוז הגישות לזיכרון שהניבו MISS
- **HIT RATE** – אחוז הגישות לזיכרון שהניבו HIT
- **HIT TIME** – הזמן הנדרש לגשת לרמת זיכרון כלשהי כולל הזמן שנדרש לקבע האם יש HIT או MISS.
- **MISS PENALTY** – כמה זמן ביזבזנו כדי למצוא בלוק מידע, כולל הזמן שלוקח להעלות את המידע לרמה נמוכה יותר שבה היה MISS עבור הבלוק הרצוי.

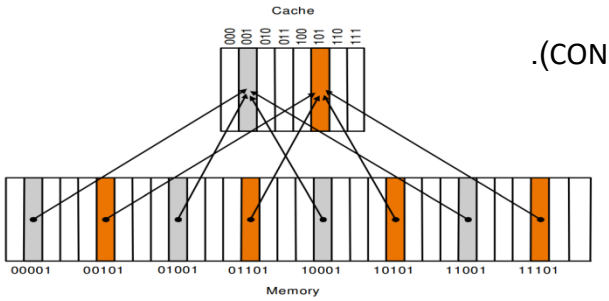
סיבות אפשריות ל- MISS:

- **COMPULSORY** – גישה ראשונה לבלוק.
- **CONFLICT** – כמה נק' בזיכרון ממופים לאותו המקום בזיכרון המטמון. MISS TAG
- **Capacity** - הזיכרון מטמון קטן ולא יכול להכיל את כל המידע שיש בזיכרון הראשי.
- **Coherence** – תהליכים אחרים מעדכנים את המידע אליו רוצים לגשת. Invalidation

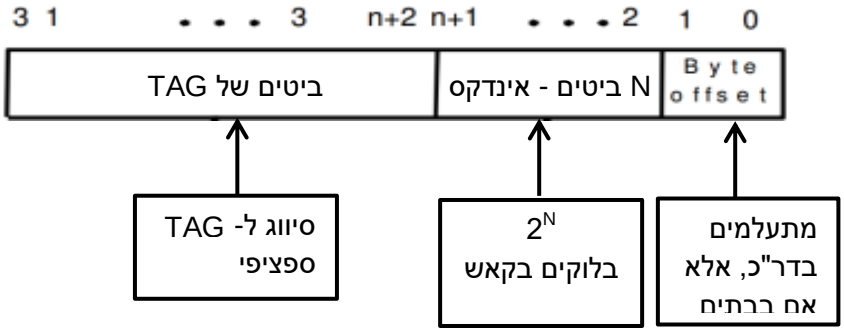
כשמגדילים את גודל הבלוק, ה- MISS RATE קטן. אך אם לא מגדילים גם את גודל זיכרון המטמון בהתאם, נגיע למצב שבו יש פחות מידי בלוקים ונאלץ לשנות אותם לפני שיכולנו לנצל את עקרו הלוקאליות (כלומר לפני שהשתמשנו בכל המידע בבלוק) ובעצם כך נגדיל את ה- MISS RATE.

מיפוי ישיר של זיכרון המטמון DMC :

מיפוי הקאש לפי הסיביות הנמוכות. (גורם ל-CONFLICT).

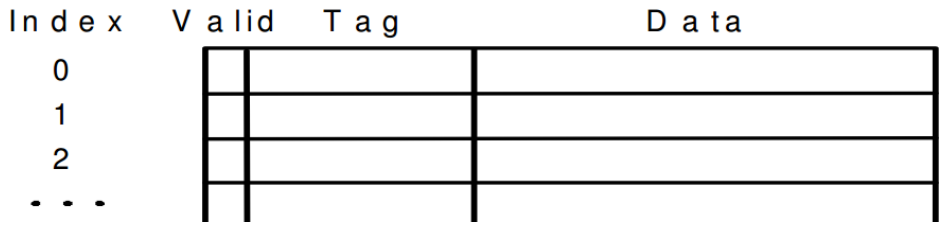


כתובת בזיכרון הראשי:



בכל בלוק/שורה יש VALID. ה- VALIDמציין האם הגיעו לשורה הספציפית. כשהקאש מתמלא, ה-VALID יהיה 0 עבור כל בלוק. ברגע שמגיעים לבלוק כלשהו ה- VALID משתנה ל-1 ונשאר כך עד שמרוקנים את הקאש.

כיצד נראה בלוק?

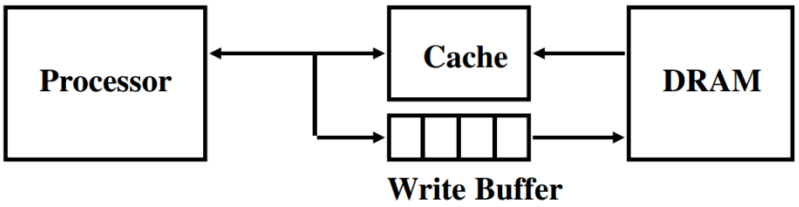


האינדקס לא נשמר בקאש.
זהו רק מיספור השורות.

טיפול בכתיבה:

- **Write-Through** – כל מה שכותבים נכתב ישירות לקאש. הקאש משתמש ב-BUFFER שב"זמנו החופשי" מחלחל את המידע למטה לזיכרון הראשי.

ה – BUFFER עובד בשיטת FIFO



- **Write-Back** – בשיטה זו, מעבירים את המידע מהקאש לזיכרון הראשי רק כאשר בלוק "נדרס".

גודל בלוק וכתיבה:

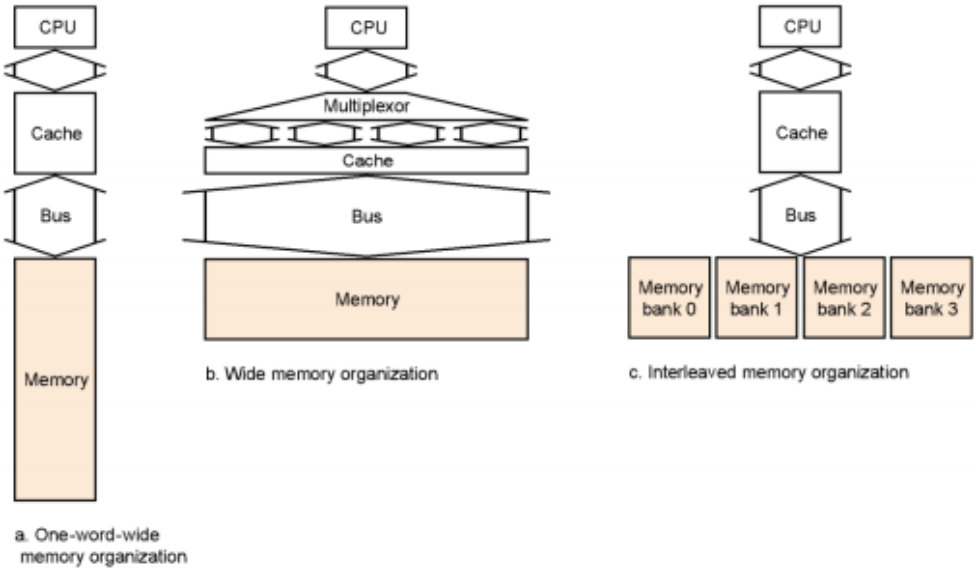
במידה וגודל הבלוק גדול ממילה, ורוצים לכתוב מילה לבלוק:

- **MISS CASE** – קודם קוראים את כל הבלוק מהזיכרון ואז כותבים אותו לזיכרון המטמון
- **HIT CASE** – אם הבלוק כבר היה בזיכרון מטמון, התהליך זהה לתהליך כשהבלוק בגודל מילה.

לפי Write-Back כאשר כותבים לבלוק, נכתוב אותו ישירות לזיכרון המטמון ונסמן אותו. כאשר רוצים לדרוס את הבלוק הזה אז נכתוב אותו לזיכרון. (יכול להשתמש ב-BUFFER כדי לאפשר קודם כתיבה של המידע הישן ואח"כ דריסה).

גודל בלוק וקריאה:

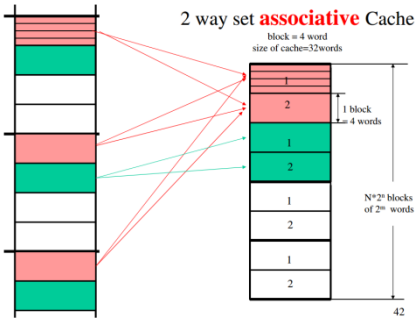
כשיש לנו בלוק בגודל גדול ממילה, הזמן שלוקח לקרוא את הבלוק גדל (כי צריך לקרוא יותר מידע). יש עיצוב של הזיכרון שמיועד לקריאה מהירה, בו קוראים כמה מילים במקביל :



2 way set associative Cache :

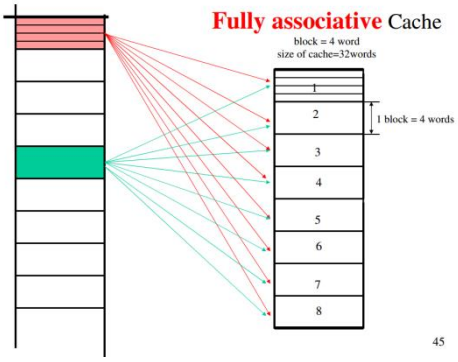
כדי לפתור את ה- CONFLICT ניתן לזיכרון מטמון יותר גמישות בנוגע לשמירת הבלוקים. לכל TAG יהיו 2 אופציות לשמירה. לכל אינדקס יש כמה TAG.
בודקים במקביל את כל ה-TAGS של אינדקס ספציפי
וברגע שיש HIT מפסיקים.

ניתן לעשות זאת עם N רמות אסוציאטיביות.

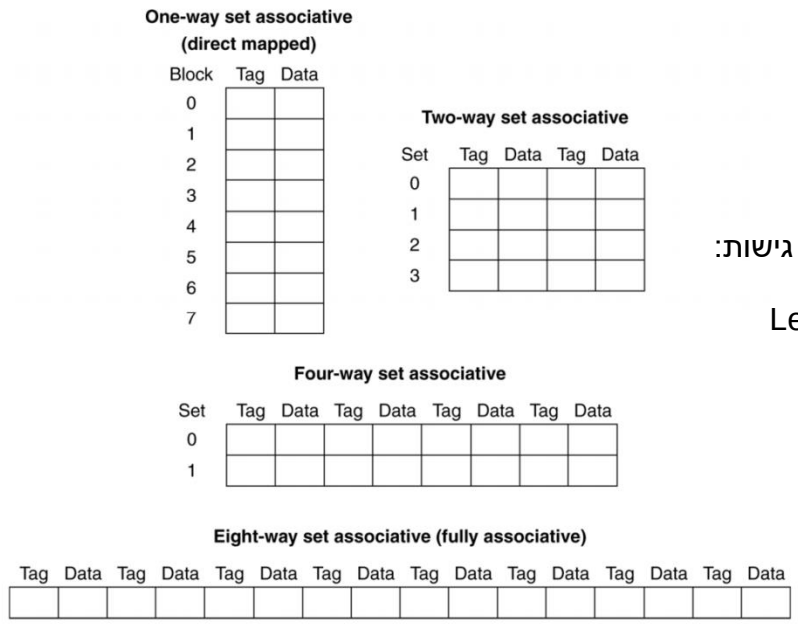


Fully associative Cache :

במקרה זה אין אינדקס.
כשרוצים להגיע לבלוק מסויים רצים על כל ה-TAG.



במימוש אסוציאטיבי ההשוואות נעשות במקביל והעלאת דרגת האסוציאטיביות מקטינה בדרך כלל את שיעור ההחטאה אך עלולה להגדיל את זמן הפגיעה עקב הוספה וסיבוך חומרה.



סיכום:

במידה וכל ה- SET מלא
וצריך להחליף בלוק כלשהו ב-SET, יש 2 גישות:

- באופן רנדומלי
- Least Recently Used = LRU

מספר בלוק Div 2^n = TAG
מספר בלוק Mod 2^n = INDEX

מספר בלוק = גודל בלוק $\overline{\lfloor X \rfloor}$

בדיקת ביצועי זיכרון מטמון:

ה- CPU TIME מורכב ממספר פעימות השעון שהתוכנית צריכה כדי לרוץ, וכמו כן גם פעימות שעון שעוכבו מהגישה לזיכרון.

$$\text{Memory stall cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

בביצועי הזיכרון מטמון חשוב מאוד גם ה- HIT TIME. לכן נחשב Average memory access time (AMAT)

$$\text{AMAT} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

סיכום ביצועים:

- מגדילים את ביצועי ה- CPU - Miss Penalty נעשה יותר משמעותי
- מקטינים CPI - חלק גדול יותר מהזמן מתבזבז על Memory Stall
- מגדילים Clock Rate - Memory Stall לוקח יותר CPU Cycles
- צריך לזכור את התנהגות ה- Cache בחישוב ביצועים

פקודות ב-MIPS:

Syscalls:

Service	Code in $v0	Arguments	Result
print integer	1	$a0 = integer to print	
print float	2	$f12 = float to print	
print double	3	$f12 = double to print	
print string	4	$a0 = address of null-terminated string to print	
read integer	5		$v0 contains integer read
read float	6		$f0 contains float read
read double	7		$f0 contains double read
read string	8	$a0 = address of input buffer $a1 = maximum number of characters to read	*See note below table*
sbrk (allocate heap memory)	9	$a0 = number of bytes to allocate	$v0 contains address of allocated memory
exit (terminate execution)	10		
print character	11	$a0 = character to print	*See note below table*
read character	12		$v0 contains character read