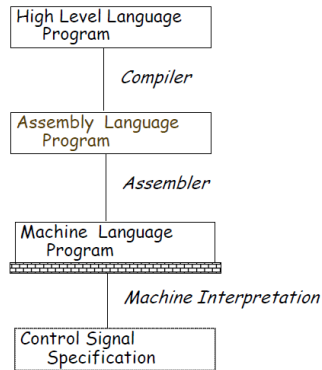


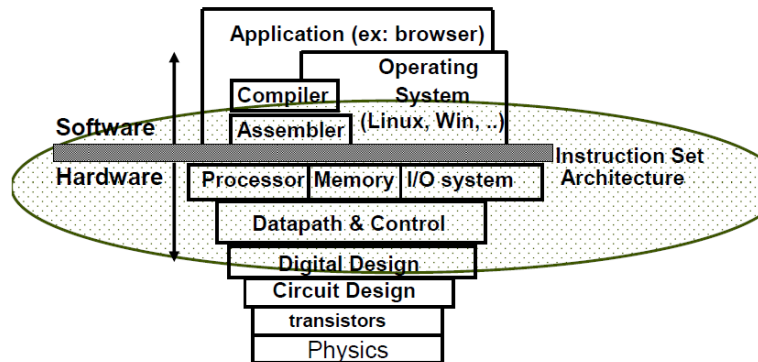
הקדמה:

רמות הפשטה בביצוע תוכנית מחשב פשוטה:



- (1) תכנות מונחה עצמים
- (2) שפת על (כמו C)
- (3) המהדר (Compiler)
- (4) מערכת ההפעלה
- (5) שפת סף / אסמבלי
- (6) שפת מכונה
- (7) מיקרו פקודות
- (8) מעגלים לוגיים
- (9) מיקרו אלקטרוניקה
- (10) פיזיקת מצב מוצק

ארגון המחשב:

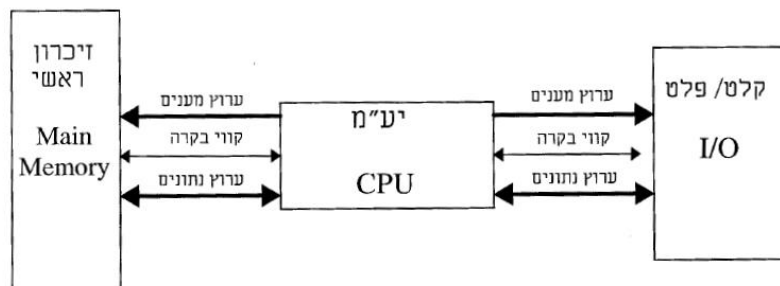


מרכיבי המחשב המרכזיים:

- (1) יחידת עיבוד (Processor):
 - א. בקר (Control – "מוח")
 - ב. נתיב מידע (Datapath – "כוח")
- (2) זיכרון – המיקום בו המידע והתוכניות שמורות ורצות.
- (3) ציוד קצה / מכשירים – ציוד קלט וציוד פלט

מודל וון-ניימן (Von Neumann Machine):

מחשב סידרתי. עובד בטור.



רמות תכנות:

- (1) שפה עלית
 - א. רמת ההפשטה הקרובה ביותר לתחום הבעיה.
 - ב. מאפשרת פריון עבודה וניידות.
- (2) שפת אסמבלי
 - א. ייצוג מידע וסט הנחיות בפורמט טקסט.
- (3) שפת מכונה
 - א. ייצוג חומרתי של ההנחיות והמידע מקודד בביטים בינאריים.

תאימות:

- 1) תאימות לאחר – חשוב שחומרה חדשה תוכל לתמוך בתוכנות חדשות.
- 2) תאימות קדימה – חשוב שחומרה ישנה תוכל לתמוך בתוכנות חדשות.
- 3) תוכנה תלויה ארכיטקטורה.

ארכיטקטורה ומיקרו ארכיטקטורה:

ארכיטקטורה:

תכונות המעבד כפי "שנראות" ע"י המשתמש. כולל בתוכה סט פקודות (הנחיות), צורות פנייה, רוחב מידע וכו'.

מיקרו ארכיטקטורה (uArch):

הדרך בה מיישמים את המעבד. כולל בתוכה גודל זיכרון ומבנה זיכרון, תזמון וכו'. מעבדים בעלי מיקרו ארכיטקטורה שונה תומכים באותה ארכיטקטורת אב.

ISA (Industry Standard Architecture):

נקרא גם סט הפקודות (Instruction Set) אותו המעבד מכיר. סט פקודות זה נוצר ע"י היצרן. סט פקודות זה הוא מה שהמשתמש והקומפיילר (המהדר) רואים ומה שעל החומרה ליישם.

סוגי מעבדים – RISC vs CISC:

CISC (Complex Instruction Set):

הרעיון העיקרי הוא להשתמש בשפת מכונה עילית.

תכונות:

- 1) הרבה סוגי פקודות עם הרבה מצבי פניה.
- 2) חלק מהפקודות מורכבות
א. מבצעות פעולות מורכבות
ב. דורשות הרבה זמני מחזור של שעון לביצוע
- 3) פעולות ALU מבוצעות ישירות בזיכרון. אין כמעט אוגרים (Registers). לרוב הפקודות אינן אורתוגונליות. כלומר לפקודות מסוימות יש מצב פניה מוגדר להן. מצב הפניה תלוי בפקודה ולהפך.
- 4) אורך פקודות משתנה. פקודות נפוצות מקבלות קודים קצרים כדי לצמצם באורך פקודה.

חסרונות:

- 1) הרבה סוגי פקודות עם הרבה מצבי פניה.
א. מסבכות את מבנה המעבד.
ב. מאטות את ביצוע המעבד אפילו בפקודות הנפוצות והפשוטות ביותר.
ג. סותרות את הכלל של להפוך את המקרה הנפוץ ביותר למהיר ביותר (Make The Common Case Fast).
2) לא ידידותי לקומפיילר.
א. אוגרים לא אורתוגונליים.
ב. מצבי פניה מורכבים שלא בשימוש.
3) אורכי פקודות משתנה גורם לבעיות.
א. מקשה על פיענוח מסי' פקודות במקביל. כל עוד הפקודה אינה מפוענחת אורכה לא ידוע.
כלומר לא ידוע איפה הפקודה מתחילה ואיפה היא נגמרת.
ב. פקודה יכולה לחרוג מהזיכרון או מהדף.

מידע נוסף:

- 1) ארכיטקטורת x86 שהיא מסוג CISC היא הדומיננטית יותר בשוק המעבדים.
- 2) מעבדי CISC משתמשים ברעיונות מעולם מעבדי ה-RISC.
- 3) כיום מעבדי CISC מתרגמים חלק מהפקודות שלהם לפעולות הדומות למעבדי RISC.

RISC (Reduced Instruction Set Computer):

הרעיון העיקרי הוא להשתמש בסט פקודות פשוט ומצומצם כדי לאפשר חומרה מהירה יותר.

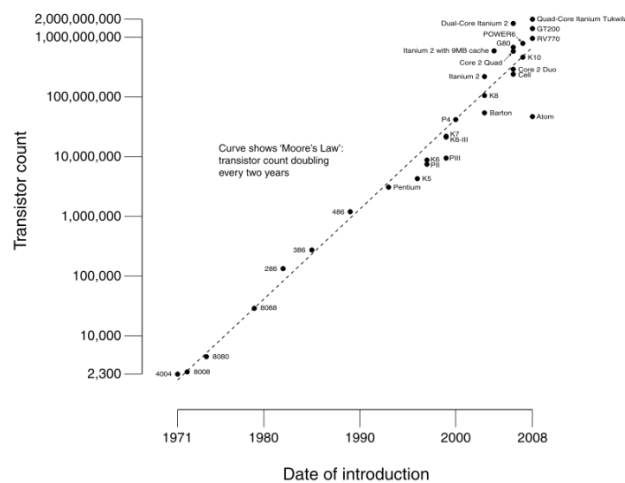
תכונות:

- (1) סט פקודות פשוט ומצומצם, באורך קבוע ומעט מאוד סוגי פקודות.
 - א. מבצע פעולות פשוטות
 - ב. מצריך מחזור שעות אחד (עם Pipeline)
- (2) פעולות ALU מבוצעות על אוגרים בלבד.
- (3) מעט שיטות מיון.
- (4) ארכיטקטורה ומיקרו ארכיטקטורה פשוטה
 - א. בקר לוגי מהיר פשוט וקטן
 - ב. קל לתכנון ולאבחון (לתת תוקף)
 - ג. מקום עבור זיכרון מטמון עבור פקודות וזיכרון מטמון עבור מידע
 - ד. זמן קצר יותר לייצור
 - ה. קל להטמיע מעבד מרובה ליבות
- (5) קומפילר חכם יותר
 - א. שימוש נבון יותר ב-Pipeline
 - ב. ניהול והקצאה נבונה יותר של אוגרים

חוק מור (Moore's Law):

חוק מור הוא תחזית או ניבוי משנת 1965 של גורדון מור לפיה צפיפות הטרנזיסטורים במעגלים משולבים במחיר מינימלי, תוכפל כל שמונה עשרה עד עשרים וארבעה חודשים. התחזית של מור שונתה כך שעוצמת המחשוב באופן כללי תוכפל כל שנתיים בערך.

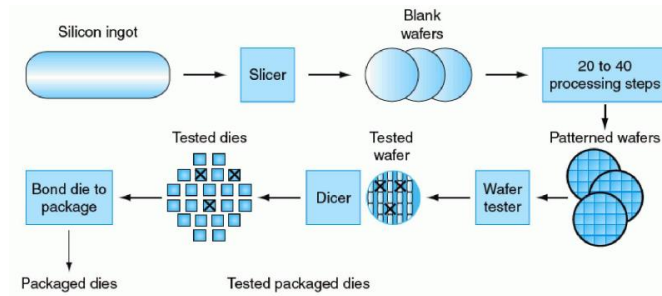
CPU Transistor Counts 1971-2008 & Moore's Law



חומת המתח (Power Wall):

- (1) $\text{Power (dynamic)} = \text{Capacitive load} * \text{voltage}^2 * \text{Frequency switched} - (CV^2f)$
- (2) הטרנזיסטורים מיוצרים בטכנולוגיית CMOS בה צריכת ההספק בעיקרה הינה ע"י מיתוג כאשר תדירות המיתוג נקבעת ע"י קצב שעות המעבד.
- (3) עומס הקיבול לטרנזיסטור נקבע ע"י הטכנולוגיה וע"י ה-Fanout (מס' הטרנזיסטורים ביציאה).
- (4) להספק יש תלות ריבועית במתח הנופל כל טרנזיסטור ולכן יש לייצר טרנזיסטורים הפועלים במתח נמוך.
- (5) הבעיה הנוצרת במתחים נמוכים היא של זליגה מהרכיב במעבדים החדשים. אובדן ההספק הנובע מכך יכול להגיע ל-40%.
- (6) היות וכיום לא ניתן להוריד משמעותית את המתח בטכנולוגיות הקיימות קיים ניסיון לפתח טכניקות קירור למעבד.
- (7) על מנת להמשיך ושלפר את ביצועי המעבד מגבלת ההספק הייתה בין הגורמים המרכזיים למעבדים מרובי ליבות.
- (8) ההספק משמעותו גם חום. הספק גבוהה מדי גורם להתחממות רבה של המעבד ולשריפתו.

ייצור מעבד וטרנזיסטורים:



ארכיטקטורת מחשבים בשנים הבאות:

היום	בעבר
Power Wall. חשמל יקר לעומת טרנזיסטורים בחינם.	אנרגיה וצריכת חשמל לא הייתה בעיה
היום חומת (IPL (Instruction Level Parallelism. שיפורי חומרה לטובת שיפורי ביצועים לא משתם.	ביצועים משתפרים על מקבול ברמת פקודות מכונה, קומפילרים חכמים וארכיטקטורת CPU יחיד.
חומת זיכרון. כפל מהיר וגישות לזיכרון איטיות.	כפל איטי, גישה לזיכרון מהירה.
ביצוע ענבד מהיר פי 2 אולי כל 5 שנים אך פי 2 ליבות כל שנתיים.	ביצועי מעבד יחיד מהיר פי 2 כל 1.5 שנים

ייצוג מידע במחשב:

קוד ASCII:

ASCII זה ראשי תיבות של (American Standard Code for Information Interchange). זהו קוד לייצוגם של תווים (ספרות, אותיות, סימני פיסוק ועוד) בזיכרון מחשב ובקובצי מחשב. קוד זה משמש להצגת אותיות האלפבית הלטיני הפשוט. ASCII הוא קידוד תווים של 7 סיביות המכיל 128 תווים, בניהם 32 תווי בקרה (כמו ירידת שורה), 52 אותיות, 10 ספרות וסימנים מיוחדים (כמו פיסוק ורווח).

כיום תקני ISO מרחיבים תקן זה כולל אותיות לטיניות אם אקצנטים. ISO 8859-8 הוא התקן לעברית המשמש ברוב מערכות המחשב הקיימות.

בעתיד ישלוט Unicode הכולל אוסף רחב בהרבה של סימנים ומאפשר שפות אחדות על דף אחד.

קוד נוסף למטרה דומה הוא קוד EBCDIC שפיתחה חברת IBM.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	\$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(72	48	H	104	68	h
9	09	Horizontal tab	41	29)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	
29	1D	Group separator	61	3D	=	93	5D]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

טבלת ASCII:

ייצוג מספרים:

בסיסים עיקריים:

Number System מערכת הצגה		Symbols (Digits) ספרות
Decimal	עשרונית (בסיס 10)	0,1,2,3,4,5,6,7,8,9
Binary	בינארית (בסיס 2)	0,1
Octal	אוקטאלית (בסיס 8)	0,1,2,3,4,5,6,7
Hexadecimal	הקסדצימלית (בסיס 16)	0,1,2,3,4,5,6,7,8,9,A(10),B(11),C(12),D(13),E(14) F(15)

ייצוג עשרוני:

10^4	10^3	10^2	10^1	10^0
10000	1000	100	10	1

דוגמא: ייצוג המס' 243				
10^4	10^3	10^2	10^1	10^0
0	0	$2 * 100$	$4 * 10$	$3 * 1$

מחברים את התאים ומקבלים 243. כלומר הדוגמא הנ"ל היא מעבר בין בינארי לעשרוני.

ייצוג בינארי:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

דוגמא: ייצוג המס' הבינארי 11110011							
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
$1 * 128$	$1 * 64$	$1 * 32$	$1 * 16$	$0 * 8$	$0 * 4$	$1 * 2$	$1 * 1$

מחברים את כל התאים ומקבלים 243.

ייצוג אוקטאלי:

ייצוג זה כולל את הספרות 0-7.

מעבר מבינארי לאוקטאלי ובחזרה:

בינארי	אוקטאלי
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

כלומר מייצגים כל ספרה בנפרד בבינארי כאילו היא עשרונית ב-3 סיביות.

המעבר מבינארי לאוקטאלי מתבצע ע"י קיבוץ של 3 ביטים מימין לשמאל. לדוגמא:

001	111	110	011	100	100
1	7	6	3	4	4

*השני אפסים בכחול זה תוספת כדי להגיע ל-3 סיביות.

ייצוג הקסדצימלי:

ייצוג זה כולל את הספרות 0-9 והאותיות A-F.

מעבר מבינארי להקסדצימלי ובחזרה:

הקסדצימלי	בינארי
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

כלומר מייצגים כל ספרה בנפרד בבינארי כאילו היא עשרונית ב-4 סיביות.

המעבר מבינארי לאוקטאלי מתבצע ע"י קיבוץ של 4 ביטים מימין לשמאל. לדוגמא:

1111	1100	1110	0100
F	C	E	4

המרת בסיסים:

ניתן להציג כל מס' הנתון בבסיס כלשהו בצורה של בסיס אחר. קיימים שלושה סוגי מעברים:

- מעבר בין מס' בבסיס כלשהו למס' בבסיס עשרוני.
- מעבר בין מס' עשרוני למס' בבסיס כלשהו.
- מעבר בין בסיסים שהינם חזקות אחד של השני.

מעבר מס' בבסיס כלשהו לבסיס עשרוני:

מס' בבסיס r המיוצג כאוסף הספרות הבא: $r_{n-1}, r_{n-2}, \dots, r_0$ שקול למס' העשרוני המחשוב כך:

$$\sum_{i=0}^{n-1} r_i * r^i$$

כלומר התהליך הוא כזה:

- כל ספרה מייצגת סיבית (אחדות, עשרות וכו') - r^i .
- כופלים את הספרה הראשונה המופיעה בספרה המייצגת של אותה סיבית.
- מחברים את כל התוצאות ומתקבל הייצוג העשרוני. של הספרה.

מעבר בין מס' בבסיס עשרוני לבסיס כלשהו:

במעבר מבסיס עשרוני לבסיס r מבצעים את האלגוריתם הבא:

- מחלקים את N (המס' בבסיס עשרוני) ב- r , חילוקים עם שארית.
- לשארית הראשונה נקרא r_0 , לשנייה r_1 וכן הלאה.
- למס' החדש שיתקבל מהחלוקה נקרא N .
- נבצע את שלבים 1,2 עד אשר $N = 0$.
- המס' בבסיס r יהיה $r_{n-1}, r_{n-2}, \dots, r_0$

מעבר בין בסיסים שהם חזקות אחד של השני:

במעבר מבסיס r לבסיס r^n נאגד קבוצה של n ביטים החל מה-LSB ונתרגם כל קבוצה לספרה בבסיס החדש. קבוצת ה- n ביטים היא בהתאם לבסיס אליו אנו עוברים.

ייצוג מס רציונליים (שברים) – Floating Point:

כללי:

לפעמים יותר קל ונוח לייצג ערך מס' באמצעות נקודה צפה (מס' עשרוני). לדוגמא הצגת המס' $1,200,000,000 = 1.2 * 10^9$.

מנטיסה (Mantissa) – מוצגת בדרך כלל באמצעות שבר בשיטת נקודה קבועה אך היא יכולה להיות גם שלם (בדוגמא שלנו המנטיסה היא 1.2).

בסיס (Base) – זהו מס' שלם קבוע ואינו חלק מייצוג המנטיסה. במחשבים מקובל להשתמש בבסיס 2 ובבסיס 10 עבור הצגה רגילה (בדוגמא שלנו זה 10).

אקספוננט (Exponent) – זהו המעריך אשר מיוצג במס' שלם (בדוגמא שלנו 9).

נשתמש באחד מהפורמטים הבאים:

$$d_0.d_1d_2 \dots d_{p-1} * B^e \quad (d_0 \neq 0) \quad (1)$$

$$(d_0 + \sum_{j=1}^{p-1} d_j B^{-j}) B^e \quad (2)$$

דוגמאות:

$$123 = 1 * 10^2 + 2 * 10^1 + 3 * 10^0 = 1.23 * 10^2 \quad (1)$$

$$123.456 = 1 * 10^2 + 2 * 10^1 + 3 * 10^0 + 4 * 10^{-1} + 5 * 10^{-2} + 6 * 10^{-3} = \quad (2)$$

$$10^2(1 * 10^0 + 2 * 10^{-1} + 3 * 10^{-2} + 4 * 10^{-3} + 5 * 10^{-4} + 6 * 10^{-5}) = 1.23456 * 10^2$$

המרת שברים בבסיס כלשהו לבסיס עשרוני:

המרת השבר בבסיס כלשהו לשבר בבסיס עשרוני תהיה זהה לשיטה שראינו קודם אולם המס' אחרי הנקודה יהיו בחזקות שליליות.

המרה משבר עשרוני לבסיס כלשהו:

(1) מכפילים את N ב- r .

(2) מסירים מ- N את הספרה שמשמאל לנקודה (החלק השלם).

(3) לחלק השלם הראשון נקרא r_{-1} , לשני נקרא r_{-2} וכן הלאה.

(4) נבצע את שלבים 1,2 עד אשר $N = 0$ (או עד אשר נבחר להפסיק).

(5) המס' בבסיס r יהיה r_{-1}, r_{-2}, \dots

המרה מעשרוני לנקודה צפה:

(1) נמיר את המס' לבסיסים של 2 בחזקות חיוביות לשלמים וחזקות שליליות לשברים.

(2) נוציא את החזקה הגבוהה ביותר הקיימת מכל איבר ואיבר.

(3) החזקה הנותרת לכל איבר היא מיקום הערך 1 בינארי ו-2 בבסיס החזקה הגבוהה ביותר הוא הבסיס והאקספוננט.

(4) דוגמא:

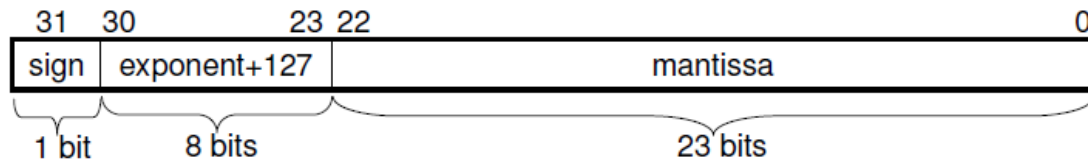
$$20.75 = 2^4 + 2^2 + 2^{-1} + 2^{-2} = 2^4(1 + 2^{-2} + 2^{-5} + 2^{-6})$$

$$= 2^4(1 * 2^0 + 1 * 2^{-2} + 1 * 2^{-5} + 1 * 2^{-6}) = 1.010011 * 2^4$$

תקן IEEE 754:

ישנם שתי דרכים לייצוג נקודה צפה (4 בייט float – או 8 בייט double) אשר תלויות ברמת הדיוק שאנו צריכים. לפי התקן נשתמש בייצוג $d_0.d_1d_2 \dots d_{p-1} * B^e \quad (d_0 \neq 0)$. בבינארי d_0 יכול להיות בעל ערך 1 בלבד והבסיס אנחנו יודעים שהוא 2. אנו נתעלם מ- d_0 ולכן נקרא לו הביט הנסתר (Hidden Bit).

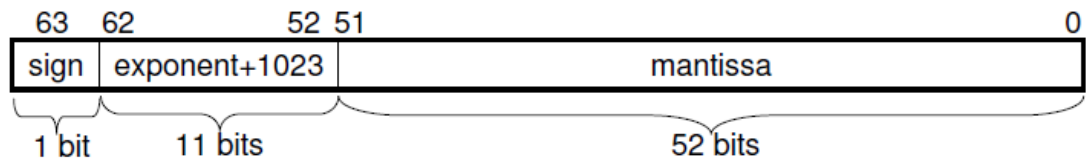
מבנה float :



ייצוג של 32 סיביות : 1 לסימן, 23 עבור מנטיסה ו-8 עבור האקספוננט. חשוב לזכור כי הערך האמיתי של האקספוננט הוא ייצוג ללא סימן של -127. הערות :

- (1) 0 מיוצג ע"י מנטיסה 0 ואקספוננט 0.
- (2) הערך הגדול ביותר הוא $2^{128} - \epsilon = 1.111 \dots 1 * 2^{127}$ (כל סיביות המנטיסה 1 ובאקספוננט כל הסיביות הן 1 פרט לסיבית הנמוכה שהיא 0).
- (3) הערך המינימאלי ביותר הוא $2^{-126} = 1.000 \dots 0 * 2^{-126}$ (כל סיביות המנטיסה 0 ובאקספוננט כל הסיביות הן 0 פרט לסיבית הנמוכה שהיא 1).
- (4) כאשר האקספוננט הוא 255 (כל הסיביות 1 או מייצגים $\pm\infty$).

מבנה double :



ייצוג של 64 סיביות : 1 לסימן, 52 עבור מנטיסה ו-11 עבור האקספוננט. חשוב לזכור כי הערך האמיתי של האקספוננט הוא ייצוג ללא סימן של -1023. הערות :

- (1) 0 מיוצג ע"י מנטיסה 0 ואקספוננט 0.
- (2) הערך הגדול ביותר הוא $2^{1023} - \epsilon = 1.111 \dots 1 * 2^{1023}$ (כל סיביות המנטיסה 1 ובאקספוננט כל הסיביות הן 1 פרט לסיבית הנמוכה שהיא 0).
- (3) הערך המינימאלי ביותר הוא $2^{-1022} = 1.000 \dots 0 * 2^{-1022}$ (כל סיביות המנטיסה 0 ובאקספוננט כל הסיביות הן 0 פרט לסיבית הנמוכה שהיא 1).
- (4) כאשר האקספוננט הוא 2047 (כל הסיביות 1 או מייצגים $\pm\infty$).

המרה מנקודה צפה לעשרוני :

- (1) במידה ונתון הקסדצימלי נמיר לבינארי.
- (2) נחלק את הסיביות לפי מבנה של נק' צפה.
- (3) נמיר את חלק האקספוננט לדצימאלי ונחסיר ממנו 127 כדי לקבל את ערך האקספוננט האמיתי.
- (4) נסתכל רק על המנטיסה (כולל הביט הנסתר). המיקום של המס' הבינארי הוא החזקה השלילית של בסיס 2.
- (5) נחבר בין כל הבסיסים והחזקות (הערה : הביט הנסתר הוא תמיד 1).
- (6) נכפיל ב-2 בחזקת האקספוננט ונקבל את המס' בעשרוני. לא נשכח להוסיף לו סימן ע"פ סיבית הסימן שהיא הסיבית הגבוהה ביותר.

דוגמא :

נתון לנו $0xC3528000 = 11000011010100101000000000000000$

מנטיסה – 23 סיביות	אקפוננט – 8 סיביות	סימן – סיבית
101001010000000000000000	10000110	1

נסתכל על המנטיסה על החלק שמעניין אותנו ונוסיף את הביט המוסתר : $1 + 2^{-1} + 2^{-3} + 2^{-6} + 2^{-8}$

נמצא את האקספוננט : $134 - 127 = 7$

נסכם : $(-1) * 2^7(1 + 2^{-1} + 2^{-3} + 2^{-6} + 2^{-8}) = -210.5$

פעולות חשבון בינאריות:

חוקי חיבור:

$$0 + 0 = 0 \quad (1)$$

$$0 + 1 = 1 \quad (2)$$

$$1 + 0 = 1 \quad (3)$$

$$1 + 1 = 10_2 = 2_{10} = 0 + 1_{carry} \quad (4)$$

$$1 + 1 + 1 = 11_2 = 3_{10} = 1 + 1_{carry} \quad (5)$$

חיבור מס' בבסיס r:

$$a_i \in \{0, 1, \dots, r-1\}$$

$$b_i \in \{0, 1, \dots, r-1\}$$

$$o_i = (a_i + b_i + c_{i-1})$$

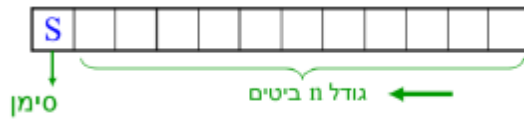
$$c_i = [(a_i + b_i + c_{i-1}) \geq r] = 1 \text{ (0 otherwise)}$$

$$\begin{array}{r} \dots a_2 a_1 a_0 \\ + \dots b_2 b_1 b_0 \\ \hline \dots o_2 o_1 o_0 \end{array}$$

(Carry bits \$c_1, c_2\$ are shown as red arrows pointing right)

חיסור:

ייצוג של מס' שליליים יהיה ע"י גודל ביטים וסימן.



הביט השמאלי ביותר (MSB) מייצג את הסימן:

$$S=0 \text{ המס' חיובי} \quad (1)$$

$$S=1 \text{ המס' שלילי} \quad (2)$$

שיטת המשלימים:

לכל מערכת לפי בסיס r קיימים שני סוגי משלימים:

$$(1) \text{ משלים ל-} r$$

$$(2) \text{ משלים ל-} r-1$$

המשלים ל-r:

בהינתן מס' חיובי N לפי בסיס r בעל חלק שלם הכולל n ספרות, המשלים ל-r מוגדר כ:

$$r^n - N \mid N \neq 0 \quad (1)$$

$$0 \mid N = 0 \quad (2)$$

המשלים ל-r-1:

בהינתן מס' חיובי N לפי בסיס r בעל חלק שלם הכולל n ספרות ומשבר בעל m ספרות המשלים ל-r-1 של N מוגדר כ:

$$r^n - r^m - N \quad (1)$$

מיקוד בשיטת המשלים ל-1 והמשלים ל-2 בחיסור בינארי:

נתון לנו שני מס' X, Y . כדי לחסר $X - Y$ נשתמש באחת מהשיטות הנ"ל (עובדים מימין לשמאל):

המשלים ל-2	המשלים ל-1
נעתיק את כל האפסים עד לאחד הראשון כולל.	נהפוך את כל הביטים של Y
נהפוך את כל שאר הביטים.	נבצע חיבור בין X ל- Y
נבצע חיבור בין X ל- Y	אם קיים נשא בסיבית האחרונה נבצע חיבור נוסף בינה למס' שקיבלנו עד הנשא.
אם קיים נשא בסיבית האחרונה נתעלם ממנה.	אם המס' שלילי נבצע שוב פעם משלים ל-1 של המס' הנ"ל ונקבל את התוצאה ונוסיף לה מינוס.

הערות:

- שיטה נוספת להגיע למשלים ל-2 זה לבצע את המשלים ל-1 ולחבר לו 1.
- המס' החיוביים זהים בשתי השיטות.

דוגמא:

Decimal	1's complement	2's complement
10	00001010	00001010
+ (-3)	11111100	11111101
+		
7	1 0000110	1 0000111
	carry 1	discarded
	00000111	

הערכת ביצועי מעבד:

הגדרות:

1. CPU TIME [sec] (זמן ריצת מעבד) – זהו הזמן הלוקח להריץ תוכנית וכולל בתוכו:
 - א. זמן תגובה (Response Time)
 - ב. זמן ביצוע (Execution Time)
 - ג. זמן חולף (Elapsed Time)
 - ד. זמן אמיתי (Real Time)
 - ה. זמן ריצה (Run Time)
 - ו. זמן השהייה (Latency). כלומר הזמן בין סיבה לתוצאה.
2. IC [instructions] (Instruction Count) – זהו מס' הפקודות בתוכנית.
3. CPI (Clock Per Instruction) $\left[\frac{\text{Cycle}}{\text{instruction}} \right]$ – ממוצע מס' מחזורי השעון שנדרש לפקודה אחת.
4. CCT (Clock Cycle Time) [sec] – הזמן שנמשך מחזור שעון אחד בשניות.
5. CR (Clock Rate) $\left[\frac{\text{cycle}}{\text{sec}} \right]$ – זהו קצב השעון והוא שווה ל- $CR = \frac{1}{CCT}$. ניתן גם להגיד כי $CCT = \frac{1}{CR}$.
6. Throughput (תפוקה) – זה סך כל העבודה המבוצעת בזמן מסוים. הקטנת זמן התגובה כמעט תמיד תשפר את התפוקה.

חישוב CPU TIME:

$$CPU\ TIME = IC * CPI * CCT$$

$$CPU\ TIME = \frac{IC * CPI}{CR}$$

שיפור ביצוע מעבד:

כלומר מטרתנו היא לשפר את זמן ריצת המעבד (CPU TIME). כדי לבצע זאת ניתן לשפר אחד מהרכיבים המרכיבים את זמן ריצת המעבד או לחילופין לשפר את כל הרכיבים. הדבר תלוי בכדאיות כספית.

- (1) צמצום CCT ע"י העלאת קצב השעון.
- (2) צמצום CPI ע"י התייעלות פקודות וארכיטקטורה.
- (3) צמצום IC ע"י שינוי ארכיטקטורה.

יעילות מעבד ע"י CPI:

CPI מאפשר לבדוק את יעילות המעבד רק כאשר ה-ISA זהה לכל מעבד שנבדק. מה שישתנה הוא ההטמעה של כל ISA.

מדד זה לא תמיד יעיל: ISA בנוי ממס' סוגים של פקודות (אריתמטיות, לוגיות, נק' צפה וכו') כאשר לכל קבוצה יש מס' שונה של CPI. על כן כדי להשתמש במדד זה יש צורך בפילוח סוגי הפקודות לשכיחות השימוש בהם ולחשב CPI ממוצע. כלומר CPI יהיה ממוצע יחסי של הפקודות ע"פ משקלן בקוד התוכנית. באמצעות CPI ממוצע ניתן יהיה לחשב מדידה מדויקת יותר של זמן ריצת המעבד.

גורם ההאצה (Speedup):

F הוא n פעמים מהיר יותר מ-S. כלומר:

$$n = speedup = \frac{CPU\ TIME\ (S)}{CPU\ TIME\ (F)}$$

מדד Peak Performance (ביצועי שיא):

מוגדרים כביצועים הטובים ביותר שמחשב יכול להפגין. כלומר כדי למצוא את מדד ביצועי השיא נבחר את ה-CPI הקטן ביותר ולפיו נמדוד את זמן הריצה. מדד זה אינו אמין שכן יכול להיות ששכיחות ה-CPI הקטן ביותר בתוכנית מסוימת הוא אפסי ועל כן המחשב כמעט ולא יגיע לביצועי השיא שלו בעוד מחשב אחר שיש לו CPI גדול יותר בסייכ יהיה מחשב מהיר וטוב יותר.

טבלה המסכמת השפעות על ביצועי מעבד:

	IC	CPI	CCT
אלגוריתם (Algorithm)	V	V	
שפת תכנות (Programming Language)	V	V	
מהדר (Compiler)	V	V	
סט הפקודות (ISA)	V	V	V
ארכיטקטורת מעבד (Processor Organization)		V	V
טכנולוגיה (Technology)			V

מדד MIPS (Millions of Instructions Per Second):

זהו בדד הבודק כמה מילוני פעולות מבצע המעבד בשנייה.

$$MIPS = \frac{IC}{CPU\ TIME * 10^6} = \frac{CR}{CPI * 10^6} \left[\frac{instructions}{sec} \right]$$

הבעייתיות במדד זה שהוא תלוי בסט הפקודות של המכונה ובהרכב תכנית הבדיקה. אם במעבד מסוים יש פקודות אסמבלר מורכבות יותר, שלוקח הרבה זמן לבצען אבל עקב כך משתמשים בפחות פקודות מאשר במעבדים אחרים (לצורך אותם פעולות) אז מדד ה-MIPS שיתקבל עבור אותו מחשב יהיה נמוך ולא בהכרח בצדק.

חוק אמדל (Amdahl's Law):

חוק אמדל אומר שיש לשפר את המקרה השכיח ביותר (Make the common case fast).
ה-fraction הוא החלק אותו אנו משפרים.

$$ExTime_{new} = ExTime_{old} \left[(1 - fraction) + \frac{fraction}{speedup} \right]$$

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

חישוב CPI באמצעות חוק זה:

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

השוואה באמצעות Benchmark:

- (1) Toy Benchmark – הרצה של תוכנה פשוטה כמו משחק פאזל. 10-100 שורות קוד.
- (2) Synthetic Benchmark – ניסיון להגיע לעומסי עבודה של העולם האמיתי.
- (3) Real Programs – הרצת תוכנות כבדות אמיתיות.
- (4) SPEC (System Performance Evaluation Cooperative) – זהו המודל בו משתמשים כיום הכולל אוסף של בדיקות. זהו המדד "האובייקטיבי" ביותר היום כאשר התקן שנעשה בו שימוש עיקרי הוא SPEC CPU2006.

השוואה וסיכום ביצועים:

ממוצע זמן ההרצה (הביצוע) אשר פרופורציונאלי באופן ישיר לזמן הביצוע הכללי נקרא ממוצע אריתמטי (Arithmetic Mean – AM).

$$AM = \frac{1}{n} \sum_{i=1}^n Time_i$$

n – מס' התוכניות בעומס עבודה.

$Time_i$ – זהו זמן הריצה של כל תוכנית.

העיקרון המנחה בדיווח המדידות הללו שהן ניתנות לשחזור. כלומר יש לציין את סוג מערכת ההפעלה, הגדרות המהדר, קונפיגורצית המחשב כמו סוג מעבד, זיכרון וכו'.

מדד תצרוכת חשמל:

זהו מדד חשוב במיוחד בשוק מעבדי ה-Embedded שבה לזמן סוללה יש חשיבות (כמו כן לטמפרטורה). כלומר מדד זה הוא קריטי לרכיבים הקצרים באנרגיה.

כיצד לבחון ISA (כיצד לתכנן נכון):

- (1) מדד תכנון זמן (Design Time Metric):
 - א. האם ניתן להטמיע? בכמה זמן ובכמה כסף?
 - ב. האם ניתן לתכנת את זה? קל לבצע לזה קומפילציה?
- (2) מדד סטטי (Static Metric):
 - א. כמה בייטים התוכנית תופסת בזיכרון?
- (3) מדד דינאמי (Dynamic Metric):
 - א. כמה פקודות מבוצעות?
 - ב. כמה בייטים המעבד שולף מהזיכרון כדי לבצע את הפקודה?
 - ג. כמה זמני מחזור של שעון לוקח כל פקודה?
 - ד. כמה "רזה" השעון עדיין פרקטי?

כלומר הכל תלוי ב-CPI, IC, ו-CCT.

בניית הרכיבים הלוגיים:

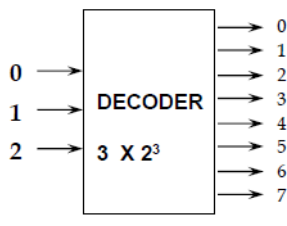
ALU (Arithmetic Logic Unit)

זוהי יחידה אריתמטית לוגית המורכבת מלוגיקה צירופית (Combinational Logic). לוגיקה צירופית אומר כי שינוי בקלט יגרום באופן ישיר לשינוי בפלט (לאחר זמן השהיה). הרכיבים העיקריים בלוגיה צירופית הם:

- (1) מפענח או מקודד (Decoder / Encoder)
- (2) מרבב או מפלג (Multiplexer / Demultiplexer)

מפענחים:

כל צירוף של קלטים מאפשר בדיוק '1' בפלט.

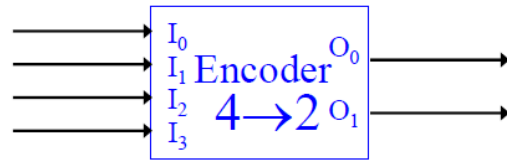


Inputs			Outputs							
I2	I1	I0	O7	O6	O5	O4	O3	O2	O1	O0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

3 X 8 Decoder

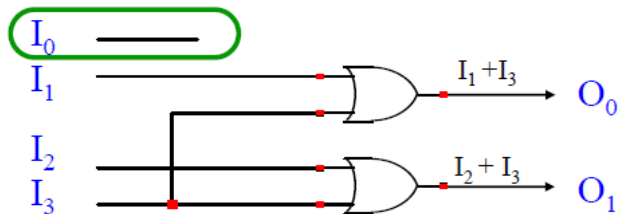
מקודדים:

ממש פונקציה "הפוכה" למפענח. 2^n קווי כניסה ו-n קווי יציאה. קלט שאינו "אונארי" (יחיד) יביא ליציאה שגויה או לא מוגדרת. כמו כן נהוג להוסיף קו אפשר (Enable).



קלט				פלט		
I3	I2	I1	I0	O1	O0	
0	0	0	1	0	0	→0
0	0	1	0	0	1	→1
0	1	0	0	1	0	→2
1	0	0	0	1	1	→3

מימוש מקודדים:



קו I_0 אינו מחובר. מכיוון שקל לממש מקודד ע"י שערי OR בלבד יש להיזהר לא להגדיר קלט שאינו חוקי.

מקודד סדר עדיפויות (Priority Encoder):

בעל 2^n כניסות, n יציאות + יציאת Valid. כלומר הוא בודק את תקינות הקלט. הפלט מציין את הביט הראשון (MSB) שהינו '1'.

I_3	I_2	I_1	I_0	O_1	O_0	V	
0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	1
0	0	1	0	0	1	1	2
0	1	0	0	1	0	1	3
1	0	0	0	1	1	1	

not valid
Valid

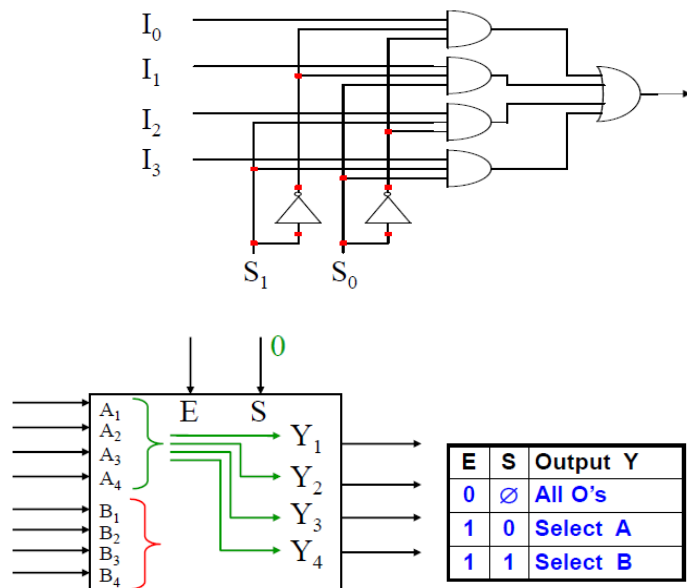
$$O_1 = I_2 + I_3 \quad O_0 = I_3 + I_1 I_2' \quad V = I_0 + I_1 + I_2 + I_3$$

מרבבים:

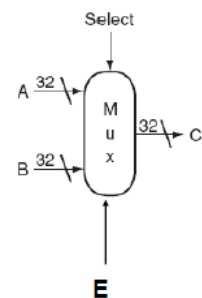
מרבב הוא כינוי להתקן אלקטרוני המממש פעולה בסיסית הנקראת ריבוב בו מתבצעת בחירה של אחד מכמה ערוצי קלט לערוץ פלט אחד בהתאם לערך בכניסות הבקרה. בעל 2^n קווי כניסה, n בוררים וקו יציאה אחד. קיימים גדלים שונים של מרבבים.

ריבוב: שידור מס' רב של יחידות מידע באמצעות מס' קטן יותר של קווים או ערוצים. מרבב ספרתי בורר קו יחיד בין קווי כניסה ומכוון את המידע הבינארי אל קו יציאה יחיד.

מימוש מרבב:



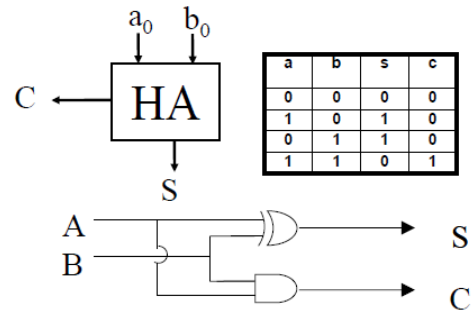
סימון מרבבים:



כאשר הקו נטוי מציין את רוחב הפס (bus) בביטים.

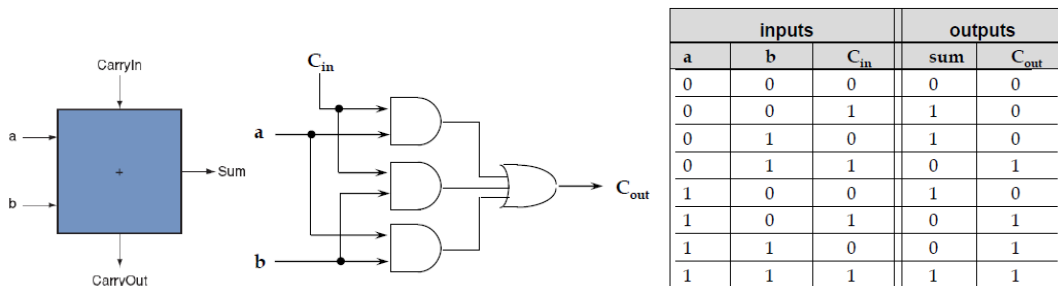
חצי מחבר (Half Adder):

מקבל 2 סיביות, מחזיר את סכומן ואת הנשא.



$$S = a \oplus b \quad C = a \cdot b$$

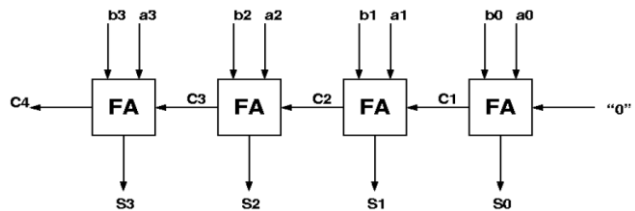
מחבר מלא (Full Adder):



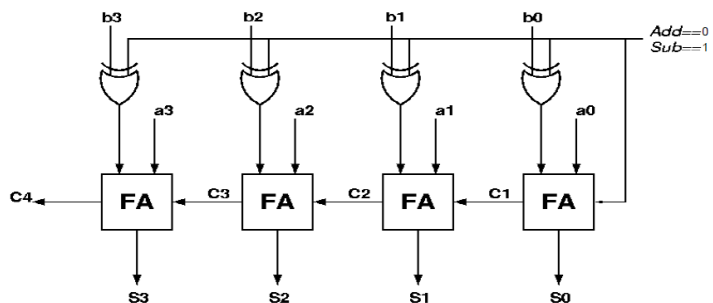
$$sum = a \oplus b \oplus C_{in} \quad C_{out} = ((a \oplus b) \cdot C_{in}) + (a \cdot b)$$

מחבר נשא גלי (Ripple Carry Adder):

סיביות הנשא מועברות בטור ולכן זמן החיבור יהיה ביחס ישר לגודל המחבר.



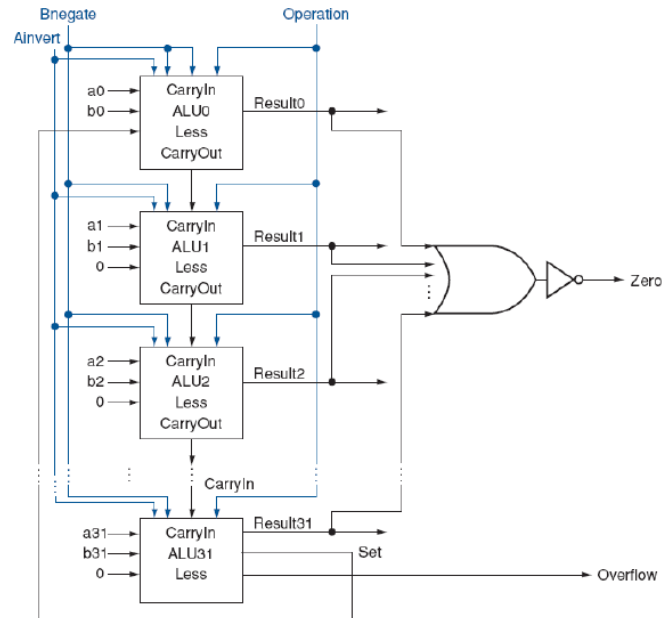
מחבר או מחסר:



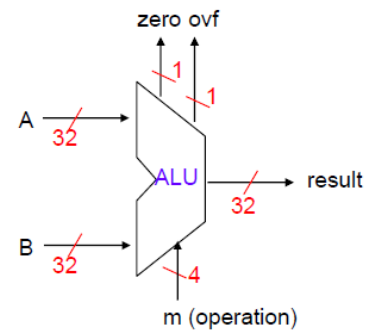
:ALU

מאפשר את הלוגיקה הבסיסית (NAND, AND, OR, NOR) ואת האריתמטיקה של חיבור וחסור במשלים ל-2. לרוב פעולות Shift, כפל וחילוק מחוץ לפעולות ALU.

פעולת ALU מלאה 32 ביט:



סיכום ALU של 4 ביט Control Line:



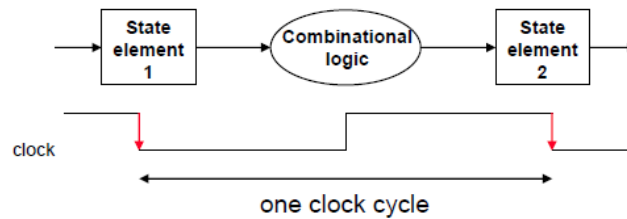
ביטים				פעולה
0	0	0	0	$a \cdot b$ AND
0	0	0	1	$a + b$ OR
0	0	1	0	חיבור
0	1	1	0	חסור $a - b$
0	1	1	1	Slt עם \bar{b}
1	1	0	0	$\bar{a} \cdot \bar{b} = \overline{a + b}$ NOR
1	1	0	1	$\bar{a} + \bar{b} = \overline{a \cdot b}$ NAND

:MIPS Registers File

מקבץ האוגרים. זהו רכיב מצבים (Sequential Logic) אשר כולל סט של אוגרים אשר יכולים להיקרא ולהיכתב ע"י הגדרת מס' אוגר אליו רוצים לגשת. לוגיקה סדרתית אומרת שהשינוי בפלט הינו תלוי שעון (זיכרון או אוגר).

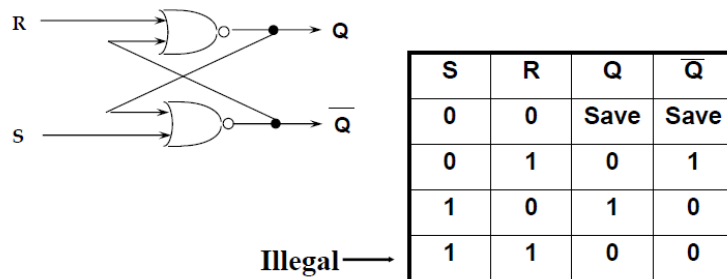
:סוגי עבודה עם שעון (Clocking Methodologies)

- (1) סוג השעון קבוע מתי אות יכול להיכתב או להיקרא.
- (2) פעולה טיפוסית:
 - א. קריאה של המידע ברכיבי מצב.
 - ב. שליחת נתונים דרך רכיבי קומבינטוריקה
 - ג. כתיבת התוצאה לרכיב מצב אחד או יותר.
- (3) ההנחה כי רכיבי מצב נכתבים אחת למחזור שעון, אחרת צריך קו בקרה מפורש.



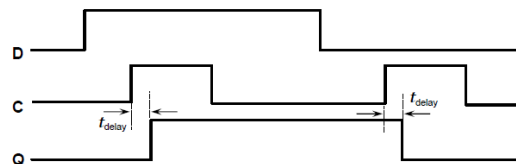
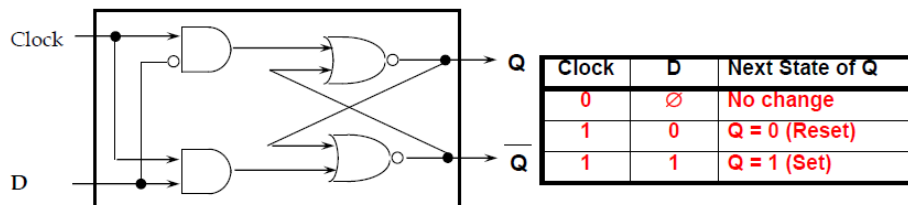
:רכיב מצב פשוט (Simple State Element)

זהו רכיב S-R Flip-Flop. כאשר הערך מוזן לרכיב הוא "נע" ברכיב ומחדש את עצמו גם לאחר שהקלט כובה.



:Clocked "D" Latch

לרכיב זה קו קלט בודד. הוא בנוי מ-S-R Flip Flop. כאשר השעון נמוך שערי AND מכריחים אפסים בכל קווי הקלט של S-R ולא יבוצע שינוי במצב. כאשר השעון גבוהה לעומת זאת הערך ב-D מוזן ל-S והשלילה של D נכנס ל-R. זהו אבן הבנייה של האוגר.

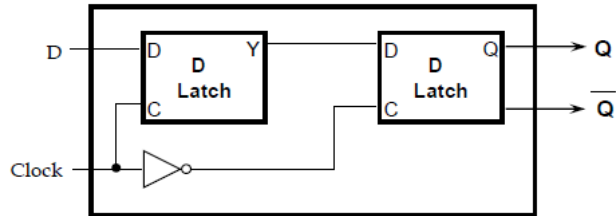


The output "Q" responds to the change in input, a characteristic delay t_{delay} after the clock goes high.

:Edge Triggered "D" Flip-Flop

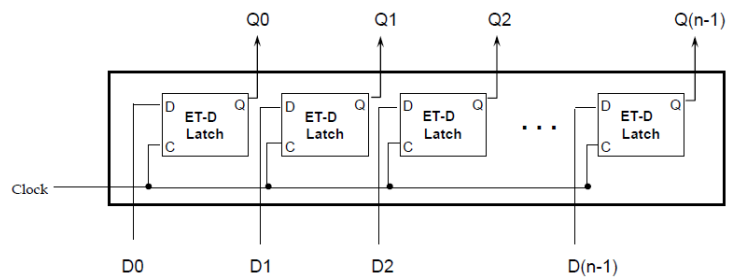
בנוי משני רכיבי Clocked "D" Latch כאשר אחד הוא שולט (Master). כאשר השעון עולה ה-D הראשון שהוא השולט מקבל את השינוי. בגלל השליטה השינוי אינו עובר ל-Clocked "D" Latch השני שהוא העבד (Slave). כאשר השעון יורד אז העבד מקבל את השינוי.

הערה: קיים Negative Clock Edge אשר פועל זהה פרט לעובדה שהשולט עובד בירידת שעון והעבד עובד בעליית שעון (כלומר הפוך מ-Edge).

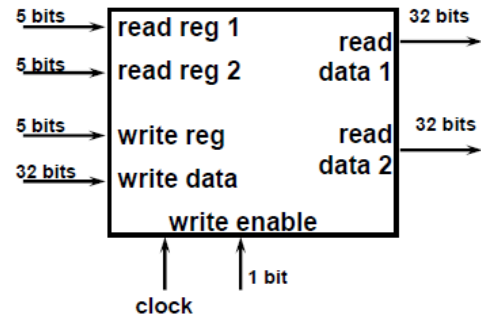


:אוגרים (Registers)

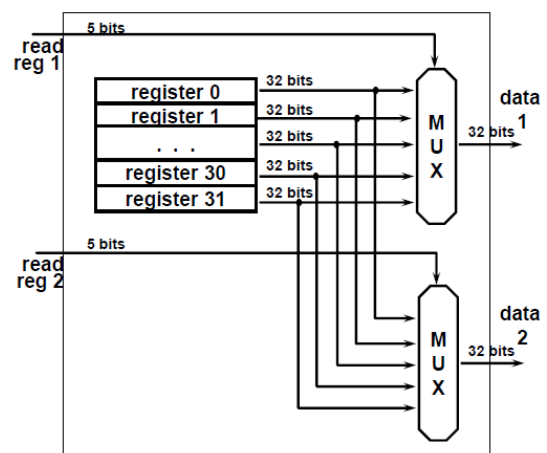
בנויים מסדרה של Edge Triggered "D" Flip-Flops המחוברים לאותו שעון.



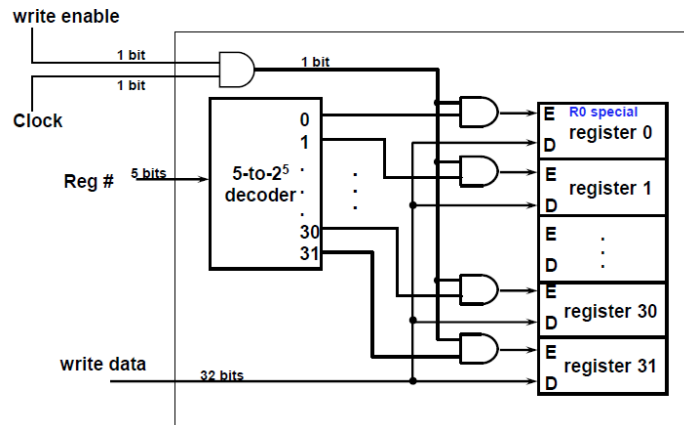
אנו נייצג את מקבץ האוגרים באופן הבא :



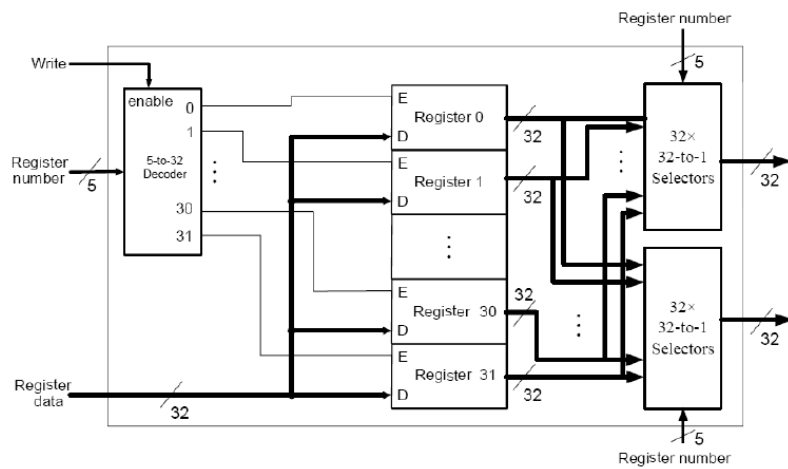
:הטמעה של קריאת שני אוגרים (Double Read Ports)



הטמעה של נתיב שמירת נתונים (Write Port):

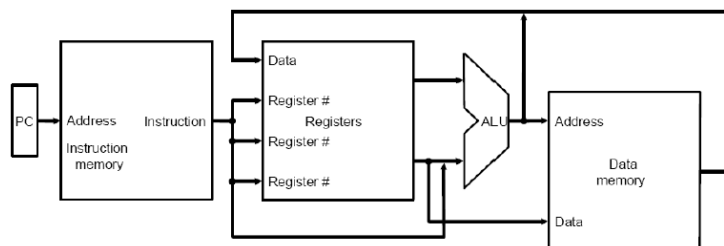


מבט כללי:



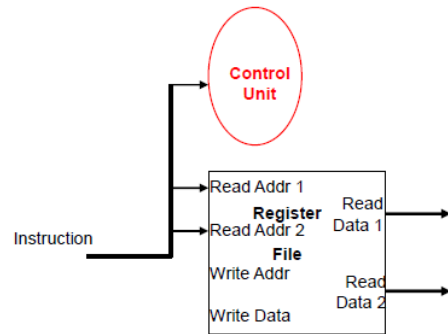
מרכיבי מסלול הנתונים:

- (1) זיכרון פקודות (Read Only)
- (2) זיכרון נתונים (Read / Write)
- (3) אוגרים
- (4) Program Counter (PC) – מס' הפקודה
- (5) ALU



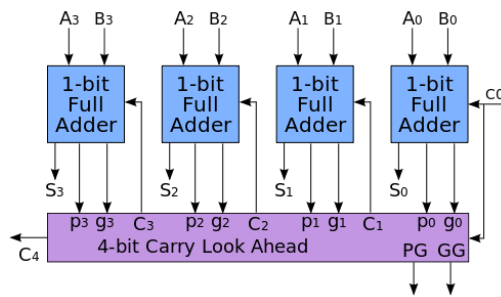
פענוח הפקודות:

פענוח הפקודות מצריך לשלוח את שדות הביטים של ה-opcode וה-function לבקר (Control Unit).



חיבור מהיר יותר:

פעולות אריתמטיות מרוכבות המצריכות שיטת גל (Ripple) הן איטיות מאוד. כדי לזרז את החישוב אנו משתמשים ב-Carry Lookahead (מבט קדימה לנשא). זהו סוג של מחבר שמשפר את המהירות ע"י הקטנת הזמן שלוקח להעריך את סיביות הנשא. הוא מחשב את הנשא לפני חישוב הסכום.



מעבד חד מחזורי (Single Cycle CPU):

נתיב נתונים ובקרה (Datapath and Control):

עקרונות מימוש:

- (1) מחזור שעון אחד לכל פקודה.
- (2) זמן התייצבות של כל רכיבי הלוגיקה הצירופית יכנס בפעימת שעון אחת.
- (3) הכתיבה לאוגרים תתבצע פעם אחת בסוף שעון.

כללי:

באמצעות שערים לוגיים, מחבר, מלא ו-Flip Flop נבנה מעבד (CPU – Central Processing Unit) שלם. אנו נבנה מעבד מסוג MIPS אך נשמיט חלקים שאינם מרכזיים ואינם קריטיים להבנה (MIPS הוא מעבד אמיתי שעובד בצורת Pipeline אותו נכיר בהמשך).

מבוא לפקודות MIPS:

המעבד החד מחזורי שנממש ידע לבצע רק חלק מפקודות ה-MIPS:

- (1) גישה לזיכרון
 - א. lw (Load Word) – הטען מילה
 - ב. sw (Store Word) – שמור מילה
- (2) פעולות אריתמטיות לוגיות
 - א. add – חיבור
 - ב. sub – חיסור
 - ג. or – או
 - ד. slt (Set Less Than) – תנאי הבודק האם ערך קטן מערך אחר
- (3) פקודות לנתיב בקרה
 - א. beq (Branch Equal) – תנאי לקפיצה
 - ב. j (Jump) – קפיצה

המשותף במימוש הפקודות:

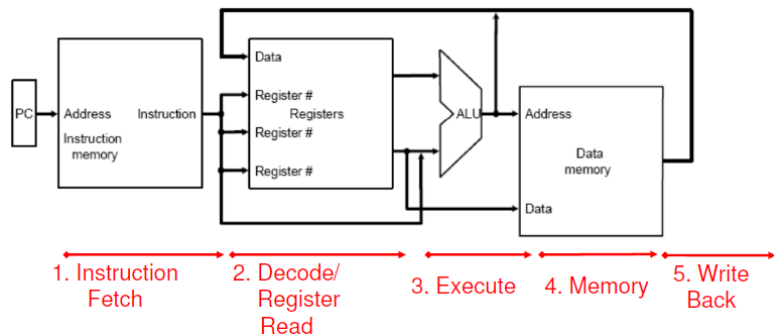
- (1) שימוש בכתובת הפקודה הנמצאת במונה הפקודות PC על מנת להביא (Fetch) את הפקודה מזיכרון הפקודות ועדכון מונה הפקודות.
- (2) פענוח הפקודה (Decode) וקריאת האוגרים המתאימים במקבץ האוגרים.
- (3) ביצוע הפקודה (Execute)
- (4) כל הפקודות למעט j משתמשות ב-ALU לאחר קריאת האוגרים.

שלבי ביצוע הוראת מכונה:

- (1) Fetch – הבאת פקודה מזיכרון הפקודות ע"פ הכתובת הנמצא ב-PC.
- (2) Decode – פענוח הפקודה וקריאת האוגרים הנחוצים (אפס, אחד או שניים) ממקבץ האוגרים.
- (3) Execute – חישוב התוצאה או הכתובת הרצויה באמצעות ALU.
- (4) Memory – השתמש בתוצאה לבצע במידת הצורך:
 - א. טעינה לזיכרון (Store)
 - ב. קריאה מהזיכרון (Load)
- (5) Write Back – בצע במידת הצורך טעינה לאוגר במקבץ האוגרים.

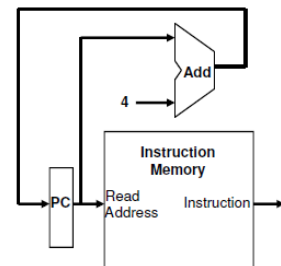
רכיבים בסיסיים למימוש נתיב נתונים:

- (1) זיכרון פקודות
- (2) זיכרון נתונים
- (3) PC – מונה פקודות
- (4) מקבץ אוגרים
- (5) יחידה אריתמטית לוגית (ALU)



שלב הבאת פקודות:

- (1) קריאת פקודה מזיכרון הפקודות
- (2) עדכון PC לכתובת הפקודה הבאה.



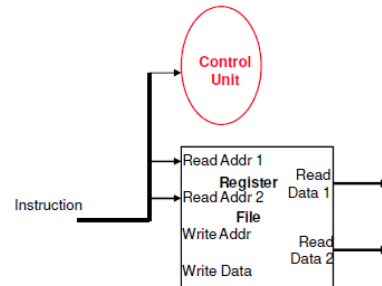
מתואר בתרשים זה מנגנון קריאת הפקודות. PC הוא אוגר המכיל את הכתובת שך הפקודה שמתבצעת במחזור השעון הנוכחי. ערכו של PC נכנס אל רכיב הזיכרון Instruction Memory שמוציא את הפקודה הנוכחית. גודל כל פקודה במעבד MIPS היא 32 סיביות, כלומר 4 בתים. המחבר ADD בתרשים מחבר את הערך הקבוע 4 (בבתים) לערך שנמצא באוגר PC ומחזיר אותו אל האוגר PC. כלומר מקדם את הפקודה לפקודה הבאה.

הערות:

- (1) מכיוון ש-PC מעודכן בכל פעימת שעון אין צורך באות כתיבה מיוחד (אפשר – Enabled).
- (2) היות וקוראים מזיכרון הפקודות בכל פעימת שעון אין צורת באות כתיבה מיוחד.

שלב הפענוח:

העברת שדה ה-opcode ושדה ה-function של הפקודה ליחידות הבקרה. כמו כן בשלב זה יש קריאה של שני אוגרים ממקבץ האוגרים כאשר הכתובות המתאימות נמצאות בשדות המתאימים בפקודה.



קידוד הפקודות:

כל הפקודות בנוות 32 סיביות, כלומר 4 בתים. כל סוג מגדיר את החלוקה הפנימית בתוך אותך 32 סיביות. ישנן שלושה סוגי פקודות:

- 1) R-Type – פקודות הקשורות לאוגרים (Register Instructions).
- 2) I-Type – פקודות הקשורות למס' מידיים (Immediate).
- 3) J-Type – פקודות הקשורות לקפיצות בזיכרון (Jumps).

שדה ה-op (opcode) קבוע בגודלו ונמצא תמיד. הוא חיוני כדי לדעת מהו סוג הפקודה וכיצד לבצע את החלוקה הפנימית השונה בין פורמט לפורמט.

פקודות מסוג R:

שדה ה-op תמיד שווה 0 ואילו שדה ה-func אומר מהי הפקודה. שדות ה-rs וה-rt עבור אוגרי המקור ושדה rd עבור אוגר המטרה. שדה ה-shamt (shift amount) מיועד לפקודות הזזת סיביות שמאלה או ימינה.

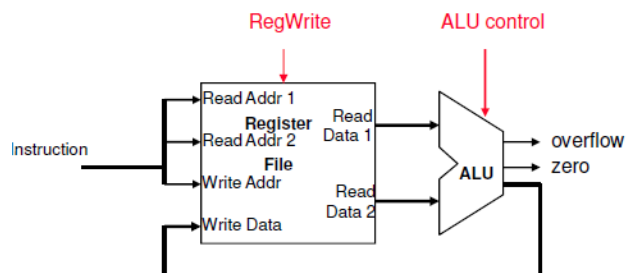
func \$rd, \$rs, \$rt

Opcode = 0 6 bit	Rs 5 bit	Rt 5 bit	Rd 5 bit	Shift n (shift amount) 5 bit	Func 6 bit
31-26	25-21	20-16	15-11	10-6	5-0

דוגמאות:

Operation	Syntax	The Action	# Function
add	add \$1,\$2,\$3	$\$1 = \$2 + \$3$	32
sub	sub \$1,\$2,\$3	$\$1 = \$2 - \$3$	34
and	and \$1,\$2,\$3	$\$1 = \$2 \& \$3$	36
or	or \$1,\$2,\$3	$\$1 = \$2 \$3$	37
nor	nor \$1,\$2,\$3	$\$1 = \sim \$2 \$3$	39
slt	slt \$1,\$2,\$3	If ($\$s2 < \$s3$) $\$s1 = 1$ else $\$s1 = 0$	42

ביצוע הפקודות:



היות ולא כל הפקודות מבצעות כתיבה למקבץ האוגרים יש צורך באות אפשר לכתוב הנקרא RegWrite.

מבנה הפקודה:

- (1) סיביות 5-0 מכילות את הפעולה של R-type.
- (2) סיביות 10-6 מכילות את ה-shift (אנו לא ניישם את האופציה הזאת)
- (3) סיביות 15-11 מכילות את מספרו של אוגר rd. סיביות אלו יחוברו אל הכניסה Write Register ברכיב אוסף האוגרים. זהו האוגר שאליו תכתב תוצאת פעולת ה-ALU. את תוצאת פעולת ה-ALU נחבר אל הכניסה Write Data ברכיב אוסף האוגרים. נשים לב כי גם בפקודת lw יש כתיבה לאוגרים מהזיכרון ועל כן יש להוסיף לפני הכניסה מרבב שבוחר את הערך הנכון. המרבב יודע איזה ערך יהיה זמין על פי ההחלטה של הבקרה הראשית.
- (4) סיביות 20-16 מכילות את מספרו של אוגר rt וסיביות 25-21 מכילות את מספרו של אוגר rs. נחבר את מסי האוגרים לכניסות 1,2 Read Register בהתאמה. נשים לב גם כאן לסתירה בין הצורך של פקודת lw ל-R-type. בפקודות lw ה-ALU מחבר את rs לסיביות 15-0 אשר מכילות את ההיסט בכתובת הזיכרון ואילו ב-R-type היא מחברת את rs עם rt ועל כן יש גם כאן צורך במרבב שערכו יבחר ע"י הבקרה הראשית.
- (5) סיביות 31-26 מכילות את ה-opcode שבמקרה זה הינו 0.

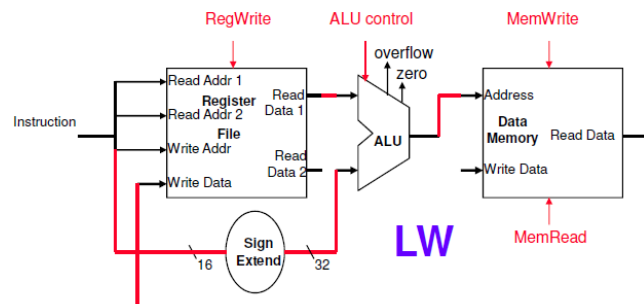
פקודות מסוג I-type:

- (1) בפקודות load ופקודות עם שדה Immediate שדה ה-rs הינו עבור האוגר הבסיס ושדה ה-rt עבור אוגר המטרה.
- (2) בפקודות store שדה ה-rs עבור אוגר הבסיס ושדה ה-rt מכיל את הערך לאחסון בזיכרון.
- (3) בפקודות branch שדות rs ו-rt הן עבור אוגרי המקור והקבוע מקודד את כתובת הקפיצה ביחס ל-pc.

פקודות lw:

lw \$rt,add(\$rs)

Opcode = 35 6 bit	Rs 5 bit	Rt 5 bit	address
31-26	25-21	20-16	15-0



הפקודה בעצם מבצעת את הפעולה הבאה: $\$rt = \text{MEM}(\text{add} + \$rs)$

אוגר rt מקבל את ערך המילה המתחילה בכתובת $\text{address} + \$rs$. כדי לבצע את חישוב כתובת המידע שדה ה-address עובר הרחבת סימן ל-32 סיביות.

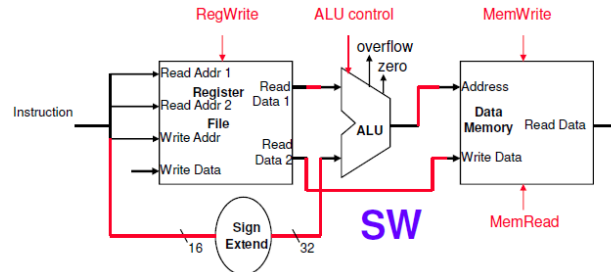
מבנה הפקודה:

- (1) סיביות 15-0 מכילות את ההיסט מן הכתובת שנמצאת באוגר rs. ה-ALU מחבר את ערך אוגר rs לסיביות שדה ה-Address לאחר הרחבת סימן והתוצאה מועברת לכניסה address של רכיב זיכרון המידע.
- (2) סיביות 16-20 מכילות את מסרו של אוגר rt. מספר האוגר נכנס אל הכניסה Write Register ברכיב מקבץ האוגרים והערך שנקרא מתוך הזיכרון יוצא ביציאה Read Data של רכיב זיכרון המידע ועובר אל הכניסה Write Data שברכיב מקבץ האוגרים.
- (3) סיביות 25-21 מכילות את מספרו של אוגר rs. סיביות אלו מחוברות לכניסה 1 Read Register של רכיב אוסף האוגרים.
- (4) סיביות 31-26 מכילות את ה-opcode שבמקרה זה שווה ל-35.

פקודות SW :

sw \$rt,add(\$rs)

Opcode = 43 6 bit	Rs 5 bit	Rt 5 bit	address
31-26	25-21	20-16	15-0



הפקודה בעצם מבצעת את הפעולה הבאה : $MEM(add + \$rs) = \rt

המילה בזיכרון בכתובת $address + \$rs$ תקבל את ערך האוגר $\$rt$ (ללא שינוי של מקבץ האוגרים). כדי לבצע את חישוב כתובת המידע שדה ה- $address$ עובר הרחבת סימן ל-32 סיביות.

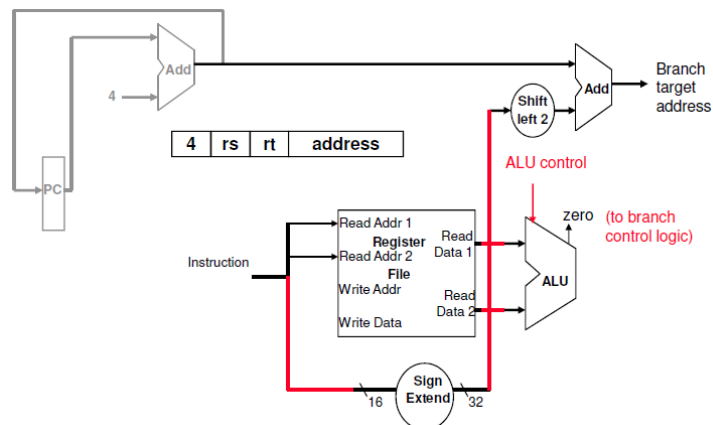
מבנה הפקודה :

הפקודה דומה מאוד לפקודת lw אך במקום לקרוא נתון מהזיכרון ולכתוב אותו באוסף האוגרים היא קוראת נתון מאוסף האוגרים וכותבת אותו לזיכרון. הנתון הנקרא מאוסף האוגרים מגיע מאוגר rt ולכן יציאה Read Data 2 באוסף האוגרים מחוברת לכניסה Write Data בזיכרון.

פקודות branch :

beq \$rs,\$rt,address

Opcode = 4 6 bit	Rs 5 bit	Rt 5 bit	Address / Immediate 16 bit
31-26	25-21	20-16	15-0



זוהי קפיצה יחסית, מותנת (פקודות הסתעפות) אשר חיונית למימוש משפטי תנאי, לולאות וניתוב בקרת התוכנית. קיימת פקודה bne הפועלת בלוגיקה הפוכה כאשר branch אינו שווה.

הפקודה בעצם מבצעת את הפעולה הבאה : $.if (\$rs == \$rt) \text{ go to } pc + 4 + 4 * \text{address}$

הערה: כאשר אנו מתכנתים הפקודה branch מכוונת לתווית (Label) אותה המהדר מתרגם לערך מספרי. מבחינת החומרה בשדה $address$ תמיד יושב מספר.

פעולת ה-Branch פועלת ע"י ביצוע השוואה ב-ALU של שני האוגרים שנקראו ממקבץ האוגרים והדלקת דגל Z (zero) במידת הצורך. לאחר מכן חישוב הקפיצה (Branch Target) ע"י הוספת PC מעודכן.

מבנה הפקודה:

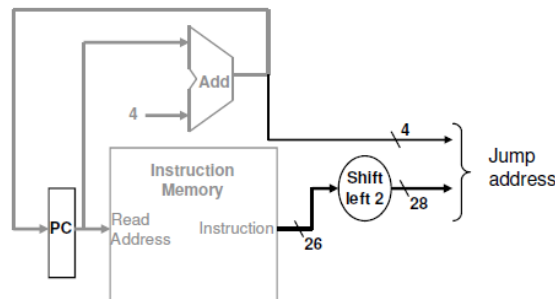
- 1) סיביות 0-15 מכילות את מס' הפקודות שאנו רוצים לקפוץ. נרחיב את מס' הסיביות ל-32 ע"י הרחבת סימן. מכיוון שזהו ערך מס' שיש לקפוץ ולא מס' בתים נכפיל ב-4 ערך זה ונקבל את מס' הבתים שיש לקפוץ. ההכפלה נעשית ע"י רכיב 2 shift left. את הערך המוכפל נחבר לתוצאה שהתקבלה מן המחבר של $PC+4$ וכך נקבל את כתובת שאליה נקפוץ אם התנאי יתקיים.
- 2) סיביות 16-20 מכילות את מספרו של אוגר rt וסיביות 21-25 מכילות את מספרו של אוגר rs. מחבר את האוגרים לכניסה 1,2 Read Register בהתאמה וביציאה נקבל 1,2 Read Data. בהתאמה. נחסר ערך אוגר rt מערך אוגר rs ע"י שימוש ב-ALU.
- 3) סיביות 26-31 מכילות את ה-opcode שבמקרה זה הינו 4.

פקודות מסוג j:

פקודת jump היא קפיצה אבסולוטית ללא תנאים. היא מבצעת קפיצה לכתובת התווית.

j label

Opcode = 2 6 bit	Address 26 bit
31-26	25-0



הפקודה מבצעת ע"י החלפת 28 הסיביות הנמוכות של PC ב-26 סיביות הנמוכות של הפקודה מוזזות שתי סיביות שמאלה ו"גניבת" 4 סיביות מ- $PC+4$.

מבנה הפקודה:

גודל כתובת בפקודה j הוא 26 סיביות. כדי להשלים את 6 הסיביות החסרות מתבצעות שתי פעולות משני צידי המס' המקורי בן 26 הסיביות.

- א. מצד ימין של המס' מוסיפים באופן קבוע שני אפסים (רכיב 2 Shift Left). כלומר מכפילים את המס' ב-100 בבסיס בינארי שהוא 4 בבסיס דצימאלי. ההכפלה נועדה כדי לוודא שהקפיצות של פקודה j יהיו כפולה של 4 בלבד.
- ב. מצד שמאל של המס' מעתיקים את ארבע הסיביות של $PC+4$ הנוכחי. העתקת הסיביות מגבילה את היכולת לקפוץ לכתובות רחוקות. ארבע הסיביות הגבוהות מחלקות את הזיכרון ל-16 חלקים ובגלל שאנו משתמשים באותן 4 סיביות בדיוק אנו נשאר באותו חלק ולא נוכל לעבור לחלק אחר מ-16 החלקים הקיימים.

הרכבת נתיב נתונים משותף:

חיבור נתיבי הנתונים השונים יעשה ע"י הוספת מרבבים וקווי בקרה. כל הפעולות באפשרויות נכנסות בפעימת שעות אחת ולכן:

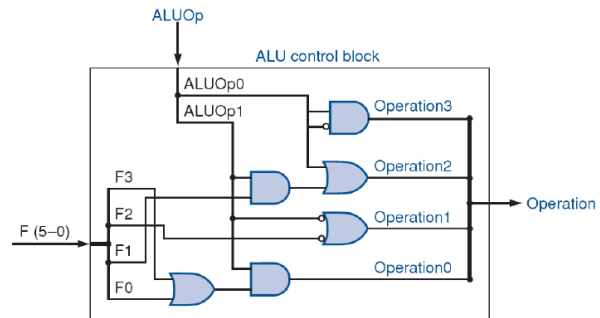
- 1) ניתן להשתמש בכל רכיב רק פעם אחת בכל פקודה, כך שחלק מהרכיבים משוכפלים.
- 2) נרבה (mux) את הכניסה לרכיבים משותפים כאשר קווי הבקרה יבחרו את הכניסה המתאימה.
- 3) קווי בקרה לנתיבה יהיו עבור מקבץ האוגרים וזיכרון הנתונים (כאשר הכתיבה עצמה מבוצעת בסוף השעות).
- 4) הפקודה עם נתיב הנתונים עם הזמן הארוך ביותר קובעת את תדר השעות.

כלומר בכל מקום בו יש סתירה בין צרכים של פקודות שונות ישנו מרבב ש"פ החלטת הבקרה הראשית מחליט איזה מידע להעביר ואיזה מידע להשמיט.

בקרה משנית – בקרת ALU:

בוחרים מה יעשה ה-ALU כתלות בערכי השדות Op, Func מן הפקודה.

מבנה הבקרה הראשית:



- (1) תפקיד הבקרה המשנית לקבוע איזה פעולה יבצע ה-ALU בכל מחזור שעון. ה-ALU מקבל ארבעה קווים שקובעים את פעולותיו.
- (2) על פי הפקודות שפותחו עד כה יש בעקרון שלוש אפשרויות לפעולה של ALU:
 - א. בצע חיבור (lw / sw)
 - ב. בצע חיסור (beq)
 - ג. בצע את מה שכתוב בשדה func של פקודת R-type.
- (3) נקרא לאפשרויות אלו ALUOp ונמנה את ערכן:
 - א. 00 עבור חיבור
 - ב. 01 עבור חיסור
 - ג. 10 עבור func
- (4) על פי ערכי ALUOp וערכי שדה func של פקודת R-type ניתן לקבוע מה צריכה להיות פעולת ה-ALU בכל פקודה. ה-ALUOp מופקים מערכי ה-opcode והפקה זאת נעשית ע"י הבקרה הראשית. הבקרה המשנית לוקחת את שתי הסיביות של ה-ALUOp ואת ששת הסיביות של שדה ה-func וע"י פונקציה בוליאנית הופכת אותם ל-4 סיביות הנכנסות ל-ALU.

Instruction	Op	Func	(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

בקרה ראשית:

- (1) בחירת הפעולות לביצוע ברכיבים השונים.
- (2) בקרה על זרימת הנתונים (שליטה במרבבים)

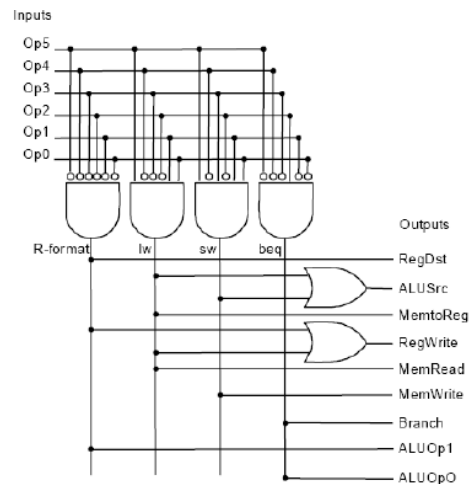
הבקר הראשי משתמש רק בערכי השדה opcode לקביעת כל הבוררים

Instruction	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0	Jump
R-Type	1	0	0	1	0	0	0	1	0	0
lw	0	1	1	1	1	0	0	0	0	0
sw	X	1	X	0	0	1	0	0	0	0
beq	X	0	X	0	0	0	1	0	1	0
J	X	X	X	0	0	0	X	X	X	1
addi	0	1	0	1	0	0	0	0	0	0

קווי הבקרה:

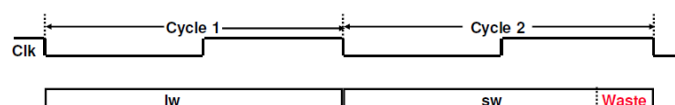
שם האות	תוצאה כאשר 0	תוצאה כאשר 1
MemRead	כלום	המידע ב-Data Memory הנמצא ב- Read address מושגים ביציאה Read Data. כלומר האם ייקרא נתון מהזיכרון במחזור שערך הנוכחי.
MemWrite	כלום	המידע ב-Data Memory הנמצא ב- write address מוחלף ע"י הערכים בכניסת Write Data. כלומר האם יכתב נתון לזיכרון במחזור השערך הנוכחי.
ALUSrc	המחובר השני של ה-ALU מגיע מהאוגר .rt	המחובר השני של ה-ALU מגיע מ-16 הסיביות הנמוכות של הפקודה (כלומר שדה ה-address)
RegDst	אוגר היעד של Register Write הינו אוגר .rt	אוגר היעד של Register Write הינו .rd
RegWrite	כלום במקרה זה אין משמעות לערכים ב-RegDst וב-MemtoReg.	האוגר שניתן ע"י מסי Write Register נכתב לתוך הערך שנמצא בכניסת Write Data. כלומר במחזור השערך הנוכחי יכתב אוגר כלשהו.
PCSrc תלוי גם ב-Zero היוצא מה-ALU.	ה-PC מתקדם רגיל. כלומר מקבל את הערך PC+4	ה-PC מקבל את הערך של תוצאת החישוב של ה-branch target.
MemtoReg בהנחה של RegWrite דלוק	הערך שמוזן לכניסת Write מגיע מה-ALU.	הערך המוזן לכניסת Register Write מגיע מה-Data Memory.
Branch	לא מתקיים חישוב	מתקיים חישוב
ALUOp0, ALUOp1	קובע את פעולת הבקרה המשנית	
Jump	לא מבוצעת קפיצה והחלטה היא בידי בקרת Branch.	מתקיים קפיצה ויילקח הכתובת מהפקודה כולל הכפלה ב-4 וגניבת סיביות מ-PC+4

מימוש הבקר הראשי:



יתרונות וחסרונות המעבד החד מחזורי:

- 1) ביצועים נמוכים בגלל שיש להתאים את זמן המחזור לפקודה האיטית ביותר (במקרה שלנו פקודת lw אך קיימות פקודות מורכבות יותר כמו כפל בנק' צפה).



- 2) היות בכל רכיב ניתן להשתמש פעם אחת במהלך פעימת שערך יש צורך בשכפול רכיבי חומרה.
- 3) מודל פשוט וקל להבנה וללימוד.

הערכת ביצוע מעבד חד מחזורי:

ה-CPI במעבד חד מחזורי הוא קבוע 1. IC קבוע לכל ייצוג של מעבד שכן ה-ISA זהה לכולם. כלומר הכול תלוי ב-CCT.

דוגמא:

I-type	stage1	stage2	stage3	stage4	stage5	total (ns)
R - type	I-fetch	regs	ALU	regs		38
Load	I-fetch	regs	ALU	mem	regs	48
Store	I-fetch	regs	ALU	mem		39
Branch	I-fetch	regs	ALU			29
Jump	I-fetch					9

table assumes ALU, Adders - 10ns; Memory - 10ns; register file - 9ns

I-type	%
Loads	22
Stores	11
R-type	49
Branch	16
Jump	2

מכאן ש-CPU Time ממוצע הוא:

$$0.49 * 28n + 0.22 * 48n + 0.11 * 39n + 0.16 * 29n + 0.2 * 9n = 38.29 [ns]$$

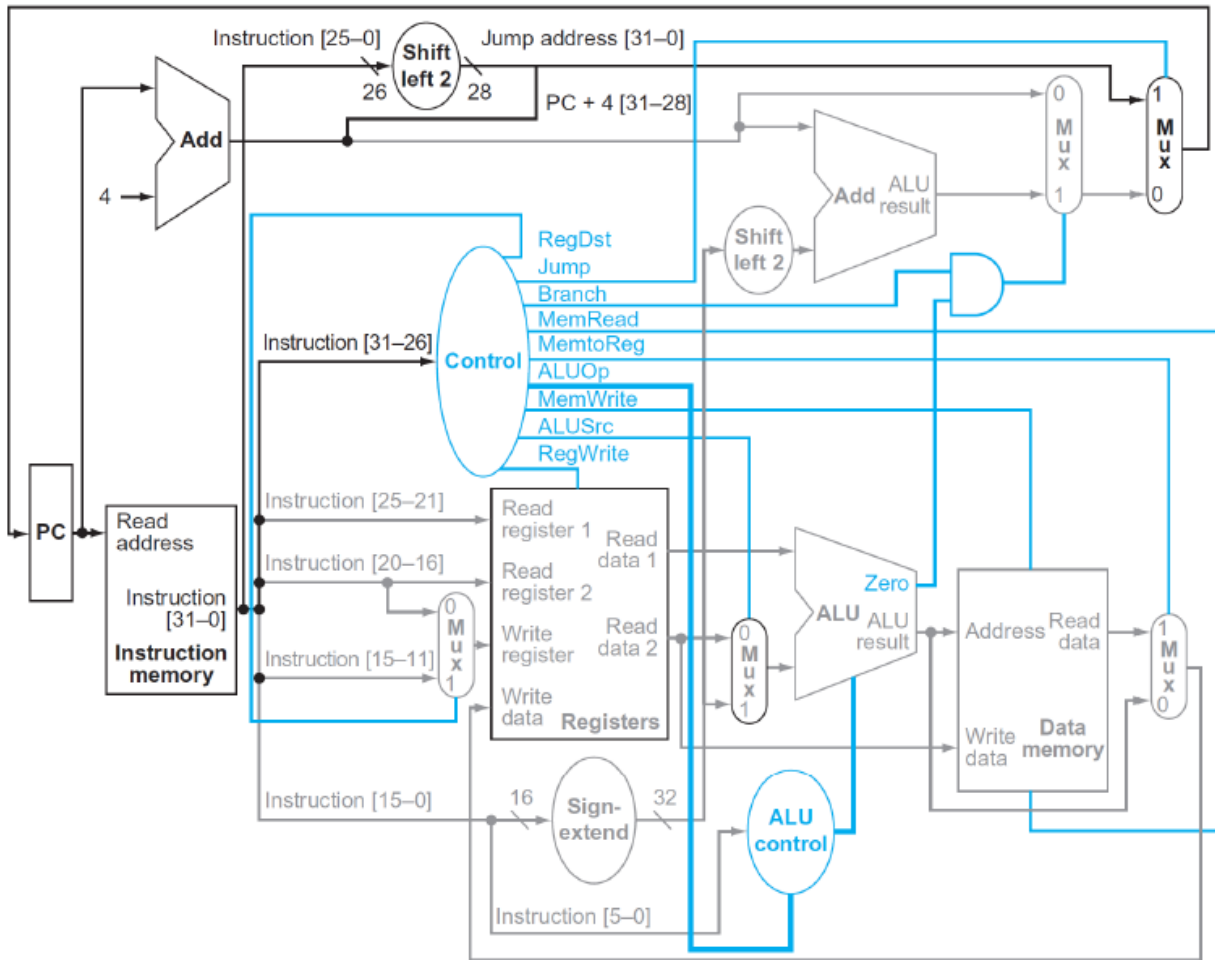
אולם בגלל שאנו מדברים על מעבד חד מחזורי אנו צריכים לקחת בחשבון את זמן הריצה הארוך ביותר ומכאן שהאטת המערכת היא בערך: $slowdown = \frac{48}{38.29} = 25.36\%$.

הערות:

- (1) ניתן לראות שאנו מייצרים שעון מאוד איטי בגלל פקודה אחת ארוכה. זהו ניצול בלתי יעיל של יכולות המעבד.
- (2) בעיית מחזור השעון הארוכה היא הבעיה המרכזית שבגינה המודל של מעבד חד מחזורי אינו מיושם במציאות.
- (3) חלוק הזיכרון לזיכרון פקודות וזיכרון נתונים הוא גם בעיה. במעבד הרב מחזורי כל פקודה תתבצע במס' מחזורים ולכן נוכל לגשת לאותו הזיכרון במחזור אחד לקריאת פקודה ומחזור אחר לקריאת הנתון מה שלא ניתן לבצע במעבד חד מחזורי.
- (4) בעיה נוספת היא שיבוץ מחברים (Adders) נוספים כאשר ה-ALU עצמו כבר יודע לבצע פעולת חיבור. כלומר חומרה מיותרת.

מבנה המעבד החד מחזורי הכולל:

FALUOP



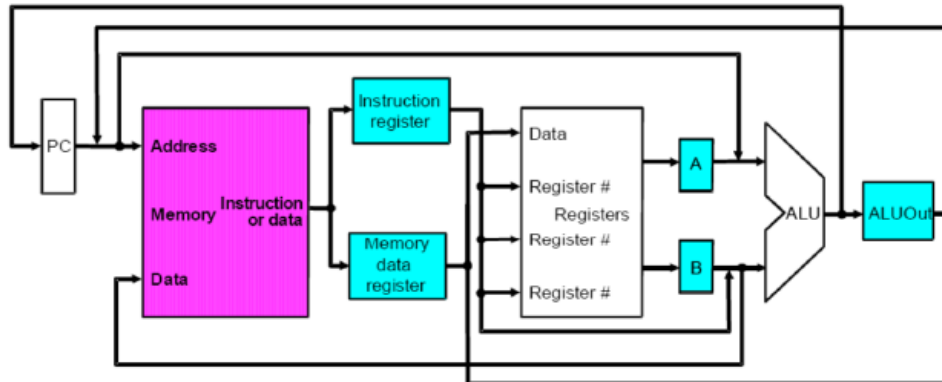
מעבד רב מחזורי (Multicycle Processor):

באופן כללי המבנה דומה מאוד למעבד חד מחזורי אולם הפקודות מבוצעות ב-5 שלבים (5 מחזורי שעות).

יתרונות וחסרונות של מעבד רב מחזורי:

- (1) ביצוע הפקודות מחולק למס' שלבים
- (2) בכל מחזור שעות נבצע רק שלב 1 בלבד
- (3) מס' משתנה של מחזורים לביצוע כל פקודה
- (4) יותר מהיר מאשר מחשב בעל מחזור יחיד
- (5) ניתן להשתמש מחדש ביחידות פונקציונאליות (בשלבים בהן אינן עסוקות בתפקידן ה"מקורי")
- (6) נזדקק לבקר FSM (מכונת מצבים) וזאת בניגוד לבקר צירופי שהספיק למחשב בעל מחזור יחיד.

מבנה עקרוני של מסלול הנתונים:



תמצית השינויים במסלול הנתונים:

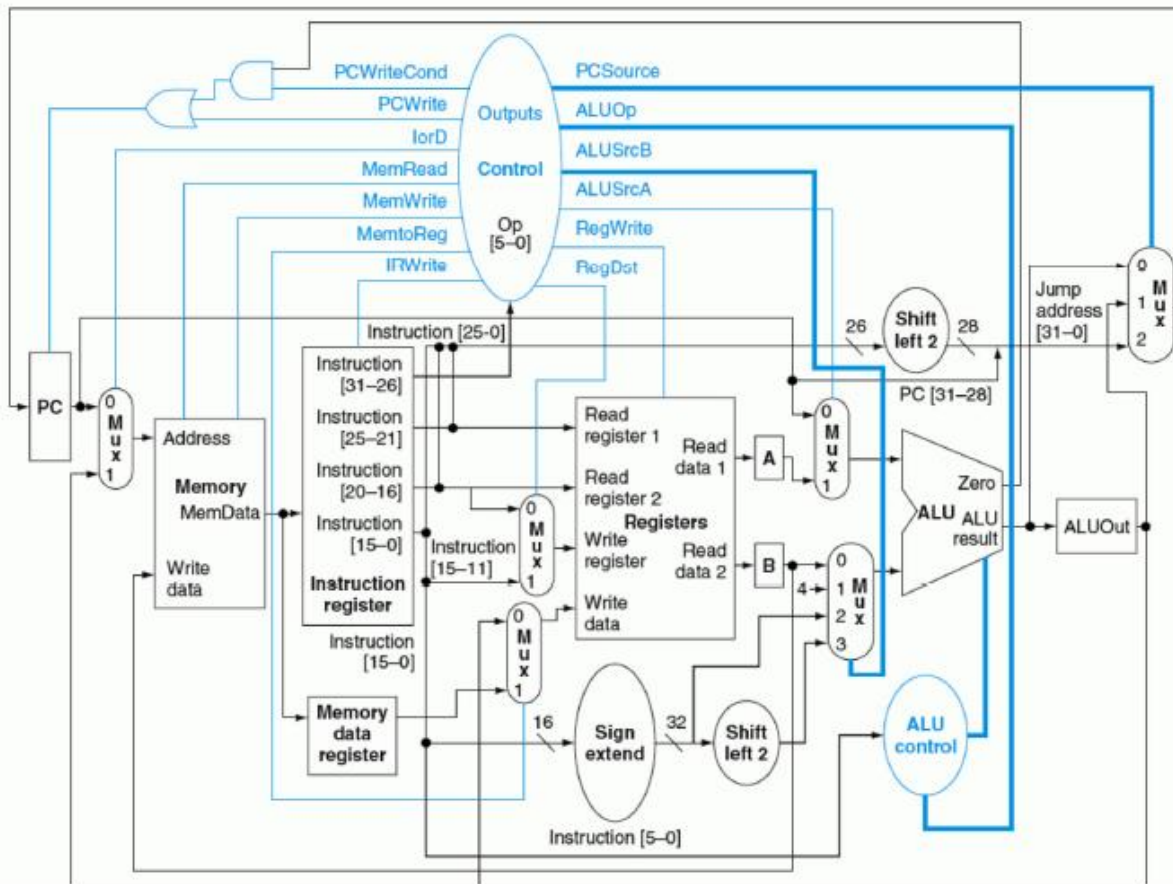
- (1) חסכון בחומרה:
 - א. זיכרון משותף לנתונים ופקודות
 - ב. ALU יחיד לכל החישובים (אריתמטיקה, כתובות נתונים, כתובות קפיצה והגדלת PC). החישובים נעשים בשלבים שונים.
- (2) מחיר בחומרה – אוגרים (רגיסטרים) חדשים לטובת ערכי ביניים:
 - א. אוגר הפקודה – Instruction Register (IR)
 - ב. אוגר הזיכרון – Memory Data Register (MDR)
 - ג. אוגרי A, B בכניסות ל-ALU
 - ד. אוגר ALUOut ביציאה מה-ALU
 - ה. תוספת בוררים (Mux)

סוגי אוגרים:

האוגרים מחולקים לשני סוגים עיקריים:

- (1) אוגרים המכילים קו אפשרות כתיבה / קריאה ומכילים נתונים ליותר מפעימת שעות אחת:
 - א. מקבץ האוגרים
 - ב. PC
 - ג. IR
- (2) אוגרים המתעדכנים בכל פעימת שעות (אוגרים אלו שקופים ברמת המתכנת):
 - א. MDR
 - ב. B
 - ג. A
 - ד. ALUOut

מסלול הנתונים המלא:



חמשת מחזורי השעון:

שני המחזורים הראשונים זהים לכלל הפקודות.

מחזור שעון ראשון – Fetch:

זהו שלב הבאת הפקודה שה-PC מצביע עליה מזיכרון הפקודות. בשלב זה מתרחשים הפעולות הבאות:

- IR=Memory[PC] (1)
- PC=PC+4 (2)

כלומר עבודת המעבד בשלב זה היא כדלהלן:

- (1) קריאת הפקודה החדשה מהזיכרון (הפקודה נקראת מהזיכרון שמשמש הן לפקודות והן לנתונים ונמצאת בכתובת PC).
- (2) הפקודה נשמרת באוגר IR על מנת שניתן יהיה להשתמש בנתונים שבתוך הפקודה במשך כל מחזורי השעון שבהם הפקודה מתבצעת.
- (3) PC מקודם ב-4 כך שיהיה מוכן להצביע על הפקודה הבאה בתור לביצוע.

מחזור שיעון שני – Decode:

זהו שלב פענוח הפקודה וקריאת שני אוגרים (כאשר השימוש בהם יהיה בהתאם לפקודה). כמו כן בשלב זה מבוצע חישוב ספקולטיבי של כתובת היעד של קפיצה מותנית.

$$\begin{aligned} (1) \quad A &= \text{Reg}[\text{IR}[25 \dots 21]] \\ (2) \quad B &= \text{Reg}[\text{IR}[20 \dots 16]] \\ (3) \quad \text{ALUOut} &= \text{PC} + (\text{signextended}(\text{IR}[15 \dots 0]) < 2) \end{aligned}$$

כלומר עבודת המעבד בשלב זה היא כדלהלן:

- (1) קריאת אוגרים rs, rt. במחזור השעון הקודם אנו יודעים מי הוא אוגר rs אשר נמצא בסיביות 21-25 של הפקודה ומי הוא אוגר rt שנמצא בסיביות 16-20 של הפקודה.
- (2) נפנה אל מקבץ האוגרים לקבלת ערכי האוגרים הללו.
- (3) חישוב ספקולטיבי של כתובת יעד של קפיצה מותנית (במקרה ולא מתבצעת פקודת beq נתעלם מחישוב הכתובת):
- א. מחברים ב-ALU 16 סיביות נמוכות של הפקודה בתוספת 2 אפסים מימין והרחבת סימן ל-32 סיביות עם ה-PC.
- ב. התוצאה נשמרת ב-ALUOut.
- ג. כלומר חישוב הכתובת: $\text{pc} + 4 + 4 * \text{address}$

הערות:

- (1) במחזור שיעון זה הבקרה מזהה את הפקודה ולכן רק במחזור השעון השלישי נדע להבחין בין הפקודות.
- (2) מהערה ראשונה נובע שמחזור שיעון זה ניתן לבצע פעולות שמאפשרות ליעל עבודת פקודות עתידיות.
- (3) גם קריאת אוגרים יכולה להיות מיותרת לפעמים (בפק' lw איננו מתעניינים באוגר rt ובפקודות j איננו מתעניינים כלל באוגרים rs, rt).

מחזור שיעון שלישי – Execute:

במחזור שיעון זה מבוצעות פעולות ה-ALU. במחזור שיעון זה הבקרה כבר יודעת מהי הפקודה המבוצעת ולכן יש הבדל בין הפעולות המבוצעות עבור הפקודות השונות:

- (1) פקודות ALU (R-Type): $\text{ALUOut} = A \text{ op } B$. ה-ALU מבצע את הפעולה המבוקשת על האוגרים A ו-B והתוצאה נשמרת ב-ALUOut.
- (2) חישוב כתובת זיכרון עבור פקודות Load / Store: $\text{ALUOut} = A + \text{signextend}(\text{IR}[15 \dots 0])$. ה-ALU מחשב את הכתובת בזיכרון. כלומר מחבר את 16 הסיביות הנמוכות של הפקודה עם הרחבת סימן ל-32 סיביות עם אוגר A.
- (3) השלמת קפיצה מותנית: $\text{if } (A=B) \text{ then } \text{PC} = \text{ALUOut}$. בפקודת beq ה-ALU מחסר את ערך אוגר B מערך אוגר A. אם התוצאה היא אפס הכתובת שחושבה במחזור שיעון 2 והייתה באוגר ALUOut תועתק ל-PC. זהו מחזור שיעון אחרון בפקודות beq.
- (4) ביצוע קפיצה בלתי מותנית: $\text{PC} = \text{PC}[31 \dots 28] \parallel (\text{IR}[25 \dots 0] < 2)$. בפקודת j מוסיפים מימין 2 אפסים, סיביות 0-25 מהפקודה וארבע סיביות הגבוהות מאוגר PC. ערך זה נכתב לאוגר PC. הערה: נשים לב כי זו פעולה מאוד קצרה ומחזור שיעון שלם הוא זמן רב מדי, אולם לא ניתן לחבר את הפעולה עם מחזור השעון השני כיוון שבמחזור השעון השני אנו לא יודעים את סוג הפקודה המבוצעת.

מחזור שיעון רביעי – Memory:

במחזור שיעון זה מבוצעת גישה לזיכרון.

- (1) בפקודות Load: $\text{MDR} = \text{Memory}[\text{ALUOut}]$. בפקודה זו נקרא הערך מהזיכרון, מכתובת הנמצאת ב-ALUOut, ונכתב באוגר MDR.
- (2) פקודת Store: $\text{Memory}[\text{ALUOut}] = B$. בפקודה זו נקרא הערך שנמצא באוגר B ונכתב לזיכרון בכתובת הנמצאת ב-ALUOut. זהו מחזור השעון האחרון בפקודות Store.
- (3) סיום פקודת ALU (R-Type): $\text{Reg}[\text{IR}[15 \dots 11]] = \text{ALUOut}$. בפקודה מסוג זה נלקח הערך שחושב במחזור השלישי ונשמר באוגר ALUOut. ערך זה נכתב לתוך אוגר rd במקבץ האוגרים. זהו מחזור השעון האחרון בפקודת R-Type.

מחזור שרון חמישי – Write-Back :

במחזור זה מבוצע כתיבה למקבץ האוגרים כחלק מפקודות Load : $Reg[IR[20...16]] = MDR$. הערך שנמצא באוגר MDR נכתב אל אוגר rt במקבץ האוגרים. מחזור שרון חמישי קיים רק בפקודות Load וזהו מחזור השרון האחרון בפקודות שלו. הערה : פקודת Load אורך מחזור שרון אחד יותר מאשר פקודת Store מכיוון שבמחזור השרון השני קראנו את אוגר rt לפני שידענו באיזו פקודה מדובר וכך נחסך מחזור שרון אחד מפקודת Store.

טבלת סיכום מחזורי השרון והפעולות המבוצעות בהן :

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	$IR = Memory[PC]$ $PC = PC + 4$			
Instruction decode/register fetch	$A = Reg[IR[25-21]]$ $B = Reg[IR[20-16]]$ $ALUOut = PC + (sign-extend(IR[15-0]) \ll 2)$			
Execution, address computation, branch/ jump completion	$ALUOut = A \text{ op } B$	$ALUOut = A + sign-extend(IR[15-0])$	if $(A == B)$ then $PC = ALUOut$	$PC = PC[31-28] \parallel (IR[25-0] \ll 2)$
Memory access or R-type completion	$Reg[IR[15-11]] = ALUOut$	Load: $MDR = Memory[ALUOut]$ or Store: $Memory[ALUOut] = B$		
Memory read completion		Load: $Reg[IR[20-16]] = MDR$		

קווי הבקרה במעבד הרב מחזורי :

במעבד רב מחזורי הבקרה הראשית מושפעת לא רק מה-Opcode אלא בנוסף גם באיזה מחזור שרון בתוך הפקודה הנוכחית הפקודה נמצאת. את קווי הבקרה נחלק לארבע קבוצות :

- 1) קווי מקבץ האוגרים
- 2) קווי ה-ALU
- 3) קווי ה-PC
- 4) קווי הזיכרון

קווי מקבץ האוגרים :

קו הבקרה	מה עושה	ערכים אותם מקבל
RegDst אוגר יעד	האם לכתוב לאוגר rt או לכתוב לאוגר rd.	1 – כתוב באוגר rd במחזור שרון רביעי של פקודת R-Type 0 – כתוב באוגר rt במחזור שרון חמישי של פקודת lw. בכל שאר מחזורי השרון של הפקודות השונות אין כתיבה לאף אוגר ולכן אין חשיבות איזה ערך יקבל הקו.
RegWrite כתיבה לאוגר	האם במחזור השרון הנוכחי ייכתב אוגר כלשהו.	1 – במחזור שרון רביעי של פקודת R-Type ובמחזור שרון חמישי של פקודת Load. 0 – בכל שאר מחזורי השרון. אם קו זה מכובה (0) אז אין משמעות לערכים בקו MementoReg ובקו RegDst.
MementoReg מזיכרון לאוגר	בהנחה ש-RegWrite דלוק מהיכן יילקח המידע שייכתב לאוגר במחזור השרון הנוכחי – מהזיכרון או מתוצאת חישוב ALU.	0 – המקור הוא תוצאת חישוב ALU במחזור שרון רביעי של R-Type. 1 – המקור הוא הזיכרון במחזור שרון חמישי של פקודת Load. בכל שאר מחזורי השרון של הפקודות השונות אין כתיבה לאף אוגר ולכן אין חשיבות איזה ערך יקבל הקו.

קווי ה-ALU:

קו הבקרה	מה עושה	ערכים אותם מקבל
ALUSrcA מקור A	האם הכניסה הראשונה של ה-ALU תהיה PC או rs.	0 – מדובר ב-PC במחזור שעון ראשון של כל פקודה כדי לחשב $PC=PC+4$ ובמחזור שעון שני של כל פקודה כדי לחשב כתובת צפויה לקפיצה אם יתברר שהפקודה היא beq.
		1 – מדובר באוגר rs במחזור השעון השלישי של פקודות Load / Store כדי לחשב כתובת בזיכרון. במחזור השעון השלישי של פקודות R-Type כדי לבצע פעולות אריתמטיות ובמחזור שעון שלישי של beq כדי לחסר ערך אוגר rt מערך אוגר rs.
		בכל שאר מחזורי השעון של הפקודות השונות אין שימוש ב-ALU ולכן אין חשיבות איזה ערך יקבל הקו.
ALUSrcB מקור B	האם הכניסה השנייה של ה-ALU תהיה rt, קבוע 4, 16 סיביות נמוכות של הפקודה מוכפלות ב-4 ומורחבות סימן או 16 סיביות נמוכות של הפקודה מורחבות סימן.	00 – הכניסה תקבל את rt במחזור השעון השלישי של פקודות R-Type כדי לבצע פעולות אריתמטיות ובמחזור השעון השלישי של beq כדי לחסר ערך אוגר rt מ-rs.
		01 – הכניסה תקבל את הקבוע 4 במחזור השעון הראשון של כל פקודה כדי לחשב $PC=PC+4$.
		10 – יקבל 16 סיביות נמוכות של הפקודה מורחבות סימן ל-32 סיביות במחזור השעון השלישי של פקודות Store / Load כדי לחשב כתובת בזיכרון.
		11 – יקבל 16 סיביות נמוכות של הפקודה מוכפלות ב-4 ומורחבות סימן ל-32 סיביות במחזור השעון השני של כל פקודה כדי לחשב כתובת צפויה לקפיצה.
ALUOp בקרה משנית	קובע לבקרה המשנית (בקרת ALU) איזו פעולה לבצע.	00 – פעולת חיבור במחזור השעון הראשון של כל פקודה כדי לחשב $PC=PC+4$, במחזור השעון השני של כל פקודה כדי לחשב כתובת צפויה לקפיצה ובמחזור השלישי של פקודות Load / Store כדי לחשב כתובת בזיכרון.
		01 – פעולת חיסור במחזור השעון של beq כדי לחסר את הערך של אוגר rt מערך אוגר rs.
		10 – בחירת פעולה המופיעה בשדה function במחזור השעון השלישי של פקודות R-Type.
		הבקרה המשנית של המעבד החד מחזורי זהה לחלוטין לבקרה המשנית של המעבד הרב מחזורי.

קווי ה-PC:

קו הבקרה	מה עושה	ערכים אותם מקבל
PCSource מקור ה-PC	האם PC יקודם ב-4, יקבל כתובת שחושבה עבור beq או כתובת שחושבה עבור j.	00 – יקבל $PC+4$ במחזור השעון הראשון של כל פקודה.
		01 – יקבל את הערך הנמצא ב-ALUOut במחזור השלישי של פקודת beq בה מעבירים את הערך שנמצא ב-ALUOut אל ה-PC אם ערכי אוגרים rs, rt שווים.
PCWrite כתיבה ל-PC	האם תהיה כתיבה לאוגר PC.	10 – יקבל את הכתובת שחושבה עבור פקודת j במחזור השעון השלישי של פקודה זו.
		1 – יש כתיבה במחזור השעון הראשון של כל פקודה כדי לקדם את PC ובמחזור השלישי של פקודת j.
		בכל שאר מחזורי השעון של הפקודות השונות אין כתיבה ל-PC ולכן יקבל ערך 0.
PCWriteCond האם יש beq	כתיבה לאוגר PC בתנאי שקו Zero מה-ALU דלוק.	גם אם קו זה מכובה תתכן כתיבה לאוגר PC אם גם קו PCWriteCond וגם קו Zero מה-ALU דלוקים.
		1 – במחזור השעון השלישי של beq כדי לכתוב את הערך שנמצא ב-ALUOut ב-PC אם גם קו Zero דלוק.
		0 – בכל מחזורי השעון בהם אין כתיבה ל-PC ו-PCWrite קיבל ערך 0.
		לקו זה יש משמעות רק אם קו PCWrite מכובה. אם PCWrite דלוק תהיה כתיבה ללא התחשבות בתנאי של קו Zero.

קווי הזיכרון:

קו הבקרה	מה עושה	ערכים אותם מקבל
IorD זיכרון או נתון	האם ייקרא מהזיכרון פקודה או נתון.	0 – יקבל פקודה במחזור השעון הראשון של כל פקודה כדי לקרוא את הפקודה החדשה. 1 – יקבל נתון במחזור השעון הרביעי של פקודות Load בכל שאר מחזורי השעון לא קוראים מהזיכרון דבר ולכן אין משמעות לערך הקו. הכתובות של הפקודות נמצאות ב-PC ואילו הכתובות של הנתונים מחשבות תוך כדי פעולת Load.
MemRead קריאה מזיכרון	האם ייקרא נתון מהזיכרון במחזור שעון נוכחי.	1 – במחזור הראשון של כל פקודה כדי לקרוא את הפקודה החדשה ובמחזור הרביעי של פקודות Load. בכל שאר מחזורי השעון של הפקודות השונות אין קריאה מהזיכרון ולכן יקבל את הערך 0.
MemWrite כתיבה לזיכרון	האם ייכתב נתון לזיכרון במחזור שעון הנוכחי.	1 – במחזור השעון הרביעי של פקודות Store בכל שאר מחזורי השעון של הפקודות אין כתיבה לזיכרון ולכן הקו יקבל ערך 0.
IRWrite כתיבה ל-IR	האם לכתוב את הנתון שיוצא מהזיכרון ל-IR.	1 – במחזור השעון הראשון של כל פקודה כדי לכתוב את הפקודה החדשה. ערך IR ולכן קו זה יקבל את הערך 0.

הערכת ביצועי מעבד רב מחזורי:

במחשב בעל ריבוי מחזורי CPI תלוי בסוג הפקודה.

- 1 Load : 5 מחזורי שעון
- 2 Store : 4 מחזורי שעון
- 3 R-Type (ALU) : 4 מחזורי שעון
- 4 Branch : 3 מחזורי שעון
- 5 Jump : 3 מחזורי שעון.

חישוב ה-CPI יהיה לפי פילוג הפקודות :

$$CPI = Load\% * 5 + Store\% * 4 + ALU\% * 4 + Branch\% * 3 + Jump\% * 3$$

מחשב בעל ריבוי מחזורי יהיה מהיר יותר ממחשב חד מחזורי אם הזמן הממוצע של ביצוע הפקודה הינו קצר יותר.

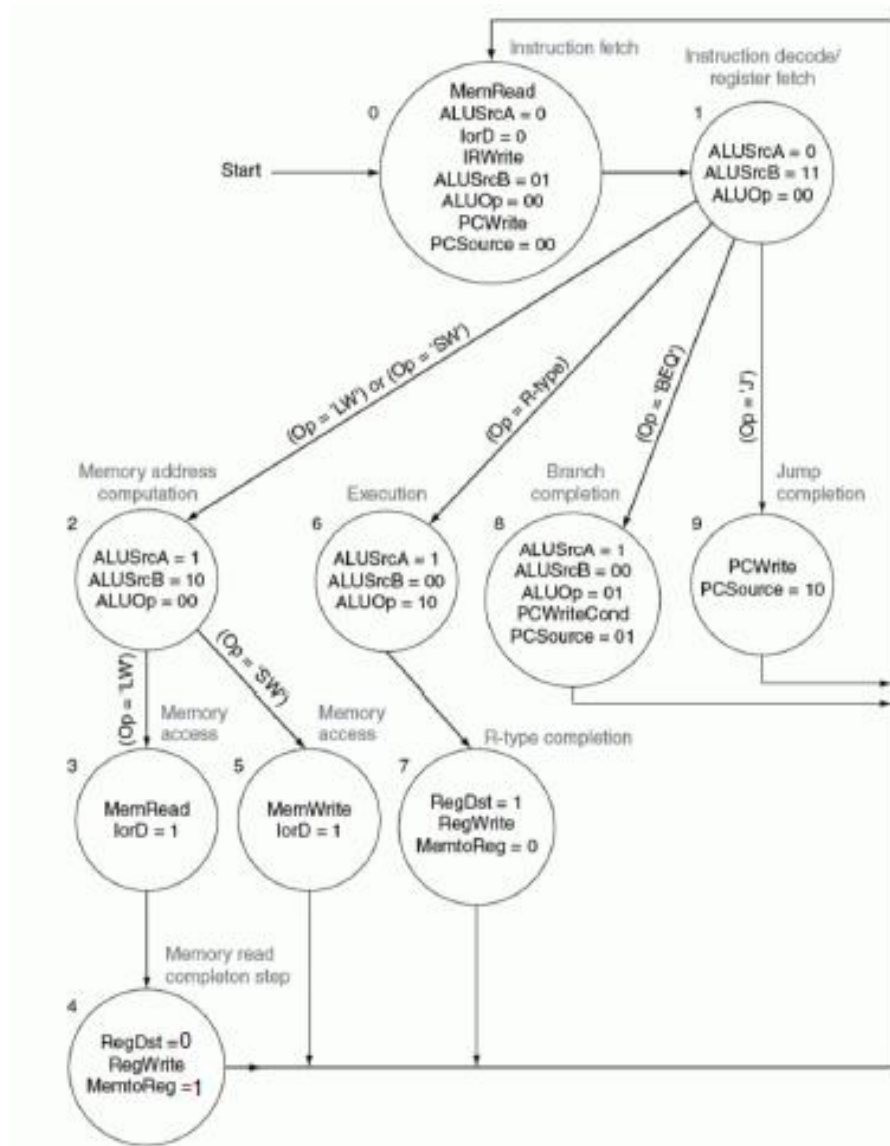
אידיאלית כיוון שחילקנו את הביצוע לחמישה חלקים השעון אמור להיות מהיר יותר פי 5 : $T_{MC} < \frac{T_{SC}}{5}$

אולם קשה לחלק באופן מאוזן שכן השלב האיטי ביותר מכתוב את זמן המחזור. כמו כן הוספת אוגרים בין השלבים מוסיפה גם כן תקורת זמן מסוימת.

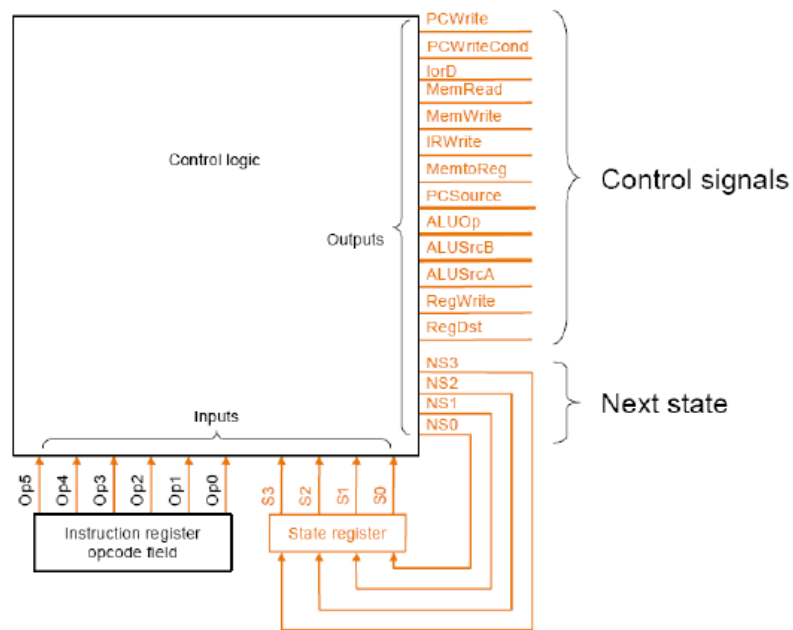
ע"פ חלוקה של הפקודות כפי שראינו בעבר : $CPI = 4.02$. במקרה הגרוע $CPI = 5$ (כאשר כל הפקודות בעלות 5 מחזורי) ועל כן במקרה האידיאלי השיפור המקסימלי במהירות הביצוע הינו : $\frac{5}{4.02} = 1.24$.

מבנה הבקרה הראשית:

כמו שאמרנו בתחילה הבקרה הראשית היא מכונת מצבים. ניתן לייצג אותה באופן הבא:



ייצוג נוסף:



מבנה:

- (1) כניסות:
 - א. 6 סיביות שדה Opcode
 - ב. 4 סיביות משתני המצב
 - ג. סה"כ 10 סיביות, $2^{10} = 1024$ מילים.
- (2) יציאות:
 - א. 16 אותות בקרה
 - ב. 4 סיביות המצב הבא
 - ג. סה"כ 20 סיביות למילה
- (3) מימוש: ע"י ROM בגודל $1024 * 20 = 20Kbits$

הקטנת ROM:

לצורך הקטנת ה-ROM נפצל אותו ל-2 טבלאות:

- (1) רק 4 סיביות מצב קובעות 16 יציאות בקרה: $2^4 * 16 = 256 bits$
- (2) 10 סיביות קובעות את המצב הבא: $2^{10} * 4 = 4 Kbits$
- (3) סה"כ 4.3 Kbits סיביות ROM

מימוש יעיל יותר של מכונת המצבים:

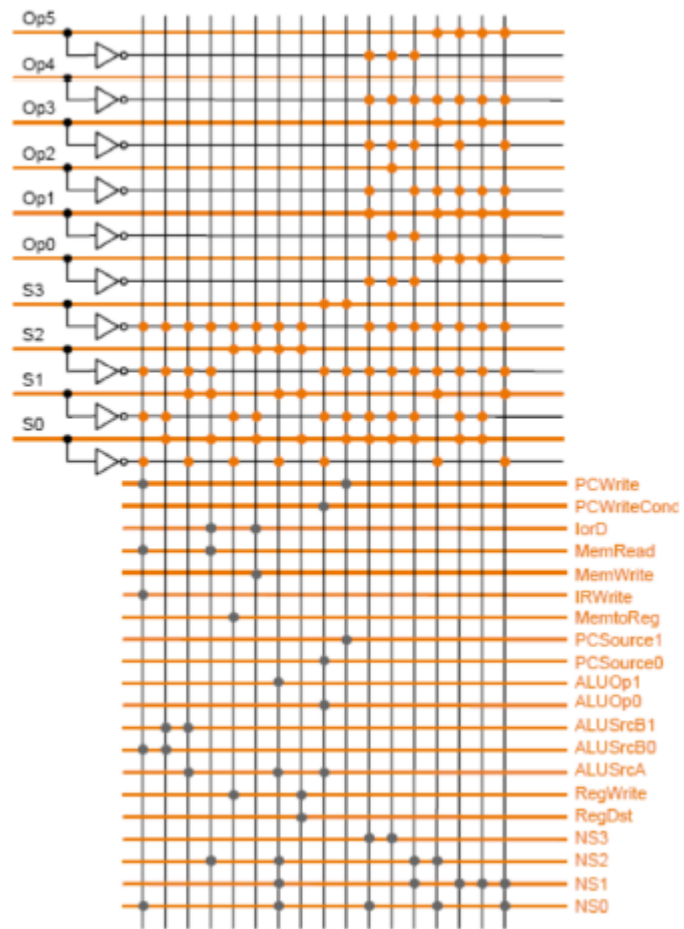
מימוש מכונת המצבים ע"י ROM אינו יעיל כיוון ש:

- (1) מכיל את כל הצירופים האפשריים של סיביות הבקרה וקוד המצב הבא
- (2) במקרים רבים ידוע בדיוק מהו המצב הבא
- (3) במקרים רבים המצב הבא הוא פשוט הכתובת הבאה ב-ROM

דרכים ליעילות:

- (1) ניתן להשתמש בשערים או PLA (Programmable Logic Array) כאשר ה-ROM לא מלא במיוחד.
- (2) נשתמש במונה (+1) למקרים של הכתובת הבאה ב-ROM ונוסיף קו בקרה AddrCtl לקבוע האן להשתמש בתוצאת המונה.

ההצגה של בקרה ראשית כ-PLA:



גודל ה-PLA נקבע ע"י (כניסות+יציאות)X(מכפלות). כלומר $510 = (20 + 10) * 17$ שערים.

PLA יעיל כיוון ש :

- (1) יש שיתוף של מכפלות
- (2) מכיל רק לוגיקה שמייצרת יציאות פעילות
- (3) מנצל ערכי Don't Care

גם PLA ניתן לצמצם ע"י פיצול לשני PLAs.

צנרת (Pipeline):

הגדרות:

:Throughput

תפוקה – קצב הייצור או הקצב שבו ניתן לעבד משו. זהו כמות המידע שמוצר ע"י מעבד (פלט) בזמן מסוים. בצנרת המשמעות כמות הפקודות המסתיימות בצנרת בזמן מסוים.

:Latency

זהו הזמן שלוקח לפקודה מסוימת להתבצע. כלומר הזמן שלוקח בין מתי שהפקודה מונפקת ולבין הזמן שהיא מסתיימת.

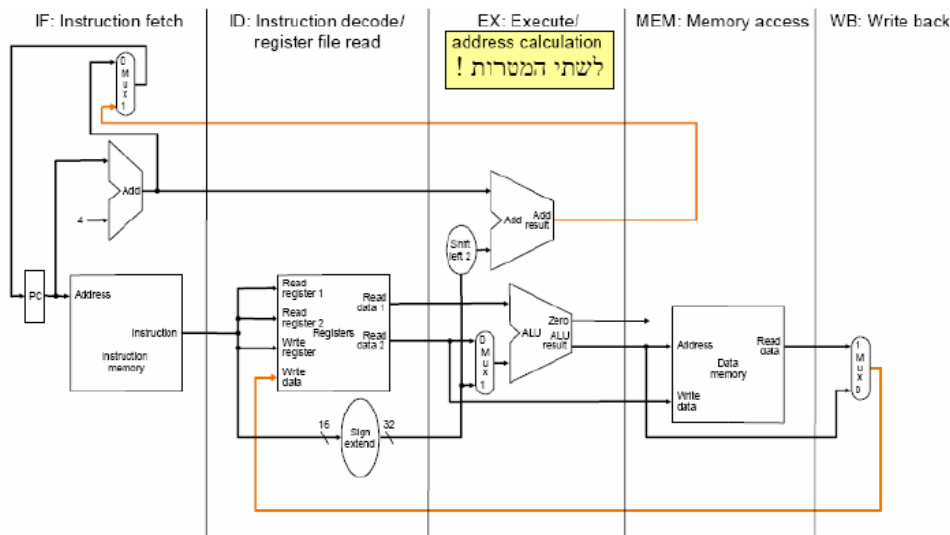
עקרון הצנרת:

הרעיון בהצנרת הוא שלא צריך לחכות עד שפקודת מכונה תסתיים אלא אפשר להתחיל את הפקודה הבאה כבר לאחר ביצוע שלב אחד של הפקודה הקודמת. באופן כללי אם כל פקודה מורכבת מחמישה שלבים אחרי סיום השלב הראשון של פקודה מסוימת אפשר להתחיל בביצוע פקודה חדשה. אחרי שהשלב השני של הפקודה הסתיים (שלב ראשון של הפקודה השנייה) ניתן להתחיל פקודה נוספת ובכל זמן נתון יכולות להתבצע חמש פקודות כאשר כל אחת נמצאת בשלב אחר של הביצוע. כלומר צנרת משפרת ביצועים כאשר היא נמצא במצב רוויה. שיפור הביצועים הוא שיפור בתפוקה של המעבד.

מבנה הצנרת:

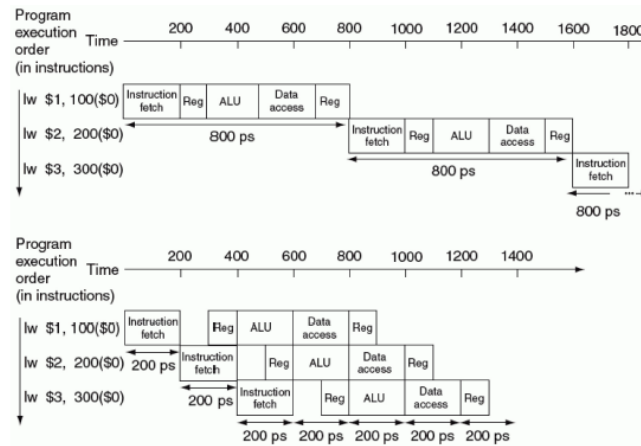
מסלול הנתונים כפי שהכרנו אותו במעבד חד מחזורי מחולק לחמישה שלבים. כל שלב הוא זמן מחזור שעון אחד. הקווים השחורים מעבירים מידע ממחזור שעון מוקדם אל מחזור שעון מאוחר יותר או לרכיב אחר באותו מחזור שעון. החיצים הכתומים מעבירים מידע אל מחזור שעון מוקדם יותר. כלומר מעבירים מידע לרכיב שכבר ביצע פעולה. חיצים אלו מצביעים לנו אם כן על הסיכונים. להלן השלבים:

- 1 Instruction Fetch – זהו השלב בו נקראת הפקודה מהזיכרון אל אוגר IR.
- 2 Instruction Decoding – זהו השלב בו מופענח ה-Opcode של פקודת המכונה הנקרא ממקבץ. כמו כן מחושבת כתובת צפויה לקפיצה.
- 3 Execution – זהו שלב הביצוע בו ה-ALU מבצע את הפעולה הספציפית הנדרשת ע"י פקודת המכונה.
- 4 Memory Access – זהו השלב בו נעשית פניה לזיכרון הראשי בפקודות lw, sw.
- 5 Write Back – זהו השלב בו נכתב ערך לאחד מן האוגרים במקבץ האוגרים בפקודות R-Type ופקודות lw.



חלוקת מסלול הנתונים לשלבים הופכת אותו למעבד מרובה מחזורים. $CPI=5$, רק פקודה אחת מתבצעת בכל רגע. זהו בזבוז ולכן נשתמש בעיקרון הצנרת. 5 פקודות שונות מתבצעות בו זמנית כאשר כל פקודה בשלב אחר.

ביצוע פקודות בטור (חד מחזורי) בהשוואה להצנרה:



נשים לב כי זמן מחזור בהצנרה הוא קבוע גם אם שלב מסוים הוא קצר יותר. למרות זאת בגלל שאנו מריצים מס' פקודות במקביל אנו מקבלים תפוקה טובה יותר וזמן ביצוע קצר יותר (כמובן רק כאשר הצנרת ברוויה – כלומר מלאה).

השהיה ו-CPI:

ביצוע פקודה בודדת:

זמן המחזור נקבע ע"י הפקודה האיטית ביותר (ע"פ התרשים מעלה [lw 800 [ps]. השהיה היא של מחזור אחד. CPI הוא 1 ותדר שעון מקסימאלי הוא 1.25 [GHz].

ביצוע Pipeline:

זמן מחזור נקבע ע"י התחנה האיטית ביותר (בדוגמא: [200 [ps]. השהיה זהו עומק הצנרת, כלומר 5 מחזורים (1000 [ps]. ה-CPI גם כאן הוא 1 ותדר השעון המקסימאלי 5 [GHz].

מכאן נקבל כי עבור מס' רב של פקודות נקבל $Speedup=4$.

יתרונות הצנרת ב-MIPS:

- (1) כל הפקודות באורך זהה של 32 סיביות (Fetch בשלב הראשון ו-Decode בשלב השני).
- (2) שלושה סוגי פקודות עם סימטריה בין כל הסוגים.
- (3) גישה לזיכרון נעשית רק בפקודות lw או sw מה שמאפשר שימוש בשלב ה-Execute (ביצוע) לחישוב כתובות.
- (4) כל פקודה ב-MIPS כותבת במקסימום תוצאה בודד ומבצעת אותה בסוף ה-Pipeline (בשלב ה-Mem ו-WB).

:Write Back

- (1) המידע נקרא מאוגר MEM/WB.
- (2) המידע נכתב למקבץ האוגרים.

חמשת השלבים של sw:

:Fetch

- (1) הפקודה נקראת מהזיכרון באמצעות הכתובת הרשומה ב-PC ונשמרת באוגר IF/ID.
- (2) $PC = PC + 4$ עבור שעון המחזור הבא. כתובת זאת נשמרת גם באוגר IF/ID למקרה ויצטרכו אותה לפקודת beq למשל. המחשב בשלב זה אינו יודע את סוג הפקודה ולכן הוא מעביר כל מידע שהוא חושב שיהיה נחוץ במחזורים הבאים.

:Decode

- (1) אוגר IF/ID מספק 16 סיביות של שדה ה-immediate בפקודה אשר עוברות הרחבה ל-32 סיביות וכן מעביר את מספרי שני האוגרים שיש לקרוא.
- (2) מידע זה נשמר לאוגר ID/EX כולל ה-PC+4 שחושב במחזור הקודם. גם כאן נעביר את כל המידע שאנו חושבים שיהיה נחוץ בעתיד.

:Execute

- (1) הכתובת הרלוונטית נשמרת באוגר EX/MEM.

:Memory

- (1) המידע נקרא מהזיכרון באמצעות הכתובת הנמצאת באוגר EX/MEM ושפוענחה בשלב שתיים.
- (2) המידע נשמר בזיכרון הראשי.

:Write Back

- (1) בשלב זה לא קורה דבר עבור פקודה מסוג sw.

העברת מידע באוגרי הצנרת:

ישנם מס' מקרים בהם נוצר נתון במחזור שעון מוקדם אך הצורך להשתמש בו יהיה לאו דווקא במחזור השעון העוקב. שלושת המקרים הבולטים הם:

- (1) קריאת פקודת מכונה במחזור הראשון ושימוש בחלקיה השונים במחזורי שעון שונים.
- (2) פק' lw המשתמשת בזהות האוגר rt רק במחזור השעון החמישי למרות שהנתון כבר קיים לאחר ביצוע מחזור השעון הראשון.
- (3) פק' sw הקוראת את אוגר rt במחזור שעון שני אך כתיבתו לזיכרון נעשית רק במחזור השעון הרביעי.
- (4) שמירת PC+4 המחושב במחזור השעון הראשון אך פק' beq מוסיפה לערך זה את ההיסט רק במחזור השעון השלישי.

תצוגה גרפית של הצנרת:

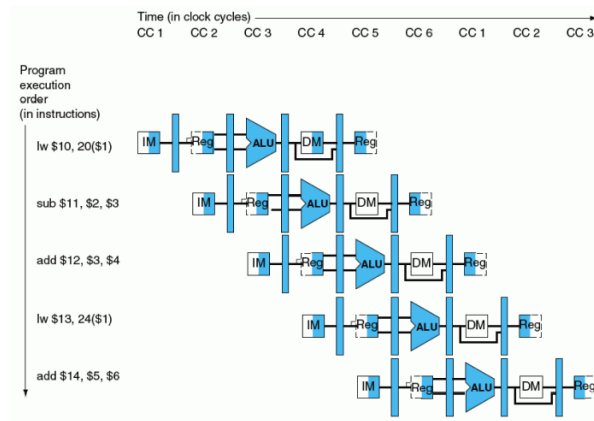
היות ובצנרת מבוצעות במקביל מס' פקודות יש צורך בהצגה גרפית על מנת לפשט את התמונה:

- (1) תרשימים המציגים מס' פעימות שעון (Multiple Clock Cycle Pipeline Diagrams)
 - א. הצגה ע"פ תצורת המשאב המייצג את השלב
 - ב. הצגה מסורתית ע"פ שם השלב
 - ג. הצגה ע"י החלפת צירים

- (2) תרשימים המציגים פעימת שעון בודדת (Single Clock Cycle Pipeline Diagrams)

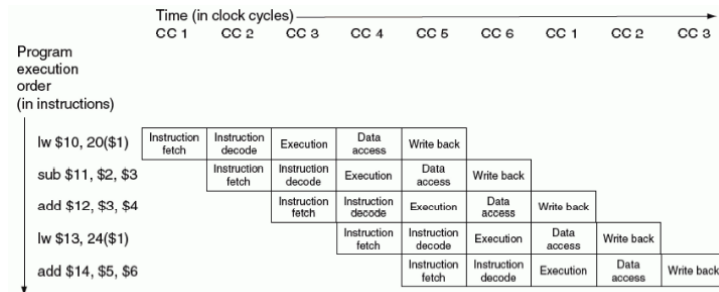
בשימוש בהצגה של מס' פעימות שעון ניתן לראות ראייה בפרספקטיבה רחבה של זמנים את מצב הצנרת מה שיאפשר להבחין בסיכויי הצנרת בצורה ברורה יותר. בהצגה של פעימת שעון בודדת נשתמש כאשר נרצה להציג את המתרחש בפעימת שעון מסוימת בצורה יותר מפורטת.

חתך בהצגה של הרכיבה בכל והתקדמות הפקודות בזמן:



חתך מסורתי ע"פ התקדמות הפקודה בזמן:

נשתמש בשיטה זו כדי לפתור שאלות ביתר נוחות.

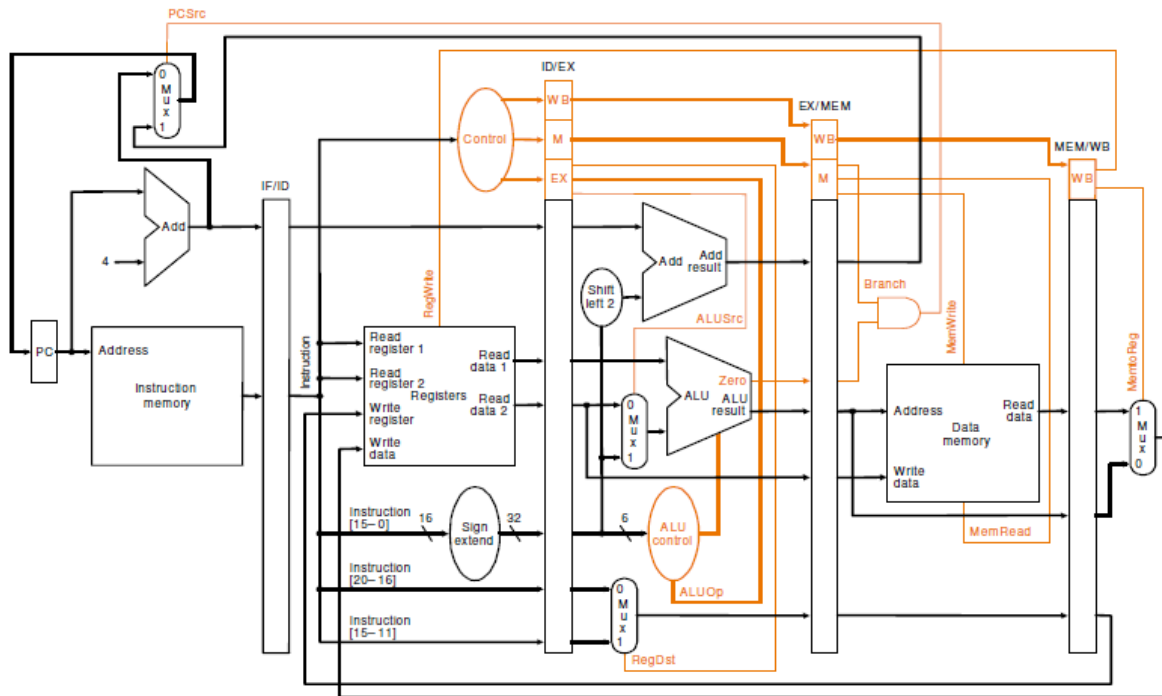


חתכים לפי השינוי במשאב / הרכיב בזמן:

Sample Program	Time: t1	t2	t3	t4	t5	t6	t7	t8
I1: ADD R4, R3, R2	IF:	I1	I2	I3	I4	I5	I6	I7
I2: AND R6, R5, R4	ID:		I1	I2	I3	I4	I5	I6
I3: SUB R1, R9, R8	EX:			I1	I2	I3	I4	I5
I4: XOR R3, R2, R1	MEM:				I1	I2	I3	I4
I5: OR R7, R6, R5	WB:					I1	I2	I3

Pipeline is "full"

בקרת המעבד:



קווי הבקרה:

Instruction	Execution / Address Calculation stage control lines				Memory Access stage control lines			Write Back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	MemtoReg
R-Type	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X
addi	0	0	0	1	0	0	0	1	0

אופן עבודת הבקרה:

נתעלם כרגע מהסיכונים בצנרת. ערך כל קווי הבקרה נקבע במחזור השעון השני והם נכתבים אל אוגר ID/EX.

- ערכים שיש צורך בהם במחזור השעון השלישי ייקראו ישירות מאוגר ID/EX.
- ערכים שיש צורך בהם במחזור השעון הרביעי או החמישי יועברו במחזור השעון השלישי לאוגר EX/MEM.
- ערכים שיש צורך בהם במחזור השעון החמישי ימשיכו במחזור השעון הרביעי אל אוגר MEM/WB.

זהו אותו מנגנון העברה שראינו עבור נתונים שנוצרים במחזור שעון מוקדם וצריכים לעבור אל מחזור שעון מאוחר יותר, לאו דווקא העוקב.

סיכונים בצנרת:

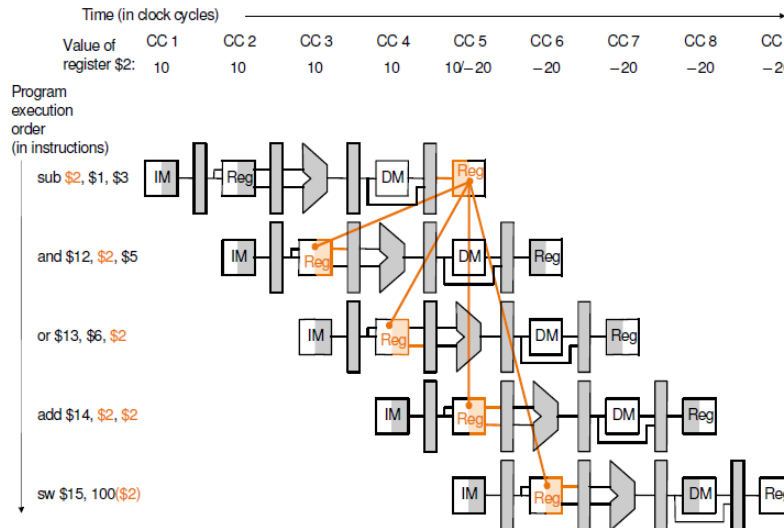
סוגי סיכונים:

- (1) סיכונים מבניים (Structural Hazards) – ניסיון לגשת לאותו משאב (אוגר לרוב) ע"י שתי פקודות שונות באותו הזמן. דוגמא: במחזור שעון הראשון בכל הפקודות מבוצע חישוב של $PC=PC+4$ ע"י ה-ALU. פקודת R-Type משתמשת במחזור השעון השלישי ב-ALU כדי לחשב את התוצאה של rs ו- rt . ה-ALU לא יכול לעשות שתי פעולות בעת ובעונה אחת ולכן נצטרך להכפיל את החומרה במקרים אלו או לשנות את מחזור השעון בו משתמשים ברכיב המבוקש בחלק מן הפקודות. אנו לא נתעסק בסיכון זה כיוון שה-MIPS נבנה מלכתחילה לעבוד עם צנרת ולכן הבעיה לא קיימת בו.
- (2) סיכונים נתונים (Data Hazards) – ניסיון להשתמש במידע לפני שהוא מוכן. כלומר האופרטורים של הפקודה עדיין בשימוש ע"י הפקודה הקודמת. במילים אחרות ניתן גם לומר שזהו נתון שערכו עדיין לא נכתב למקבץ האוגרים או לזיכרון הפקודות. במקרה זה עלינו ליצור מסלול העברה קדימה (Forwarding). ייתכן גם שערך הנתון עדיין לא קיים בצנרת ולכן במצב כזה נצטרך גם השהיה (Stall).
- (3) סיכון בקרה (Control Hazards) – הניסיון להחליט לגבי תוכנית את נתיב זרימת המידע שלה לפני שהתנאי להערכת הנתיב מוערך וה-PC החדש מחושב (כלומר בפקודות branch). הפקודות j ו- beq משנות את בקרת התוכנית לכתובת אחרת אך עד שהן לא מתבצעות המעבד לא יודע מהי הכתובת. יתר על כן beq לא תמיד מעבירה את בקרת התוכנית. במקרים אלו לא נדע אילו פקודות צריכות להיכנס לתוך הצנרת משום שאיננו יודעים תמיד האם תהיה קפיצה או לא וגם אם אכן תהיה קפיצה הכתובת לא מתקבל מיד.

בסיכונים בצנרת ניתן תמיד לטפל ע"י המתנה. כלומר השתלת בועות בצנרת.

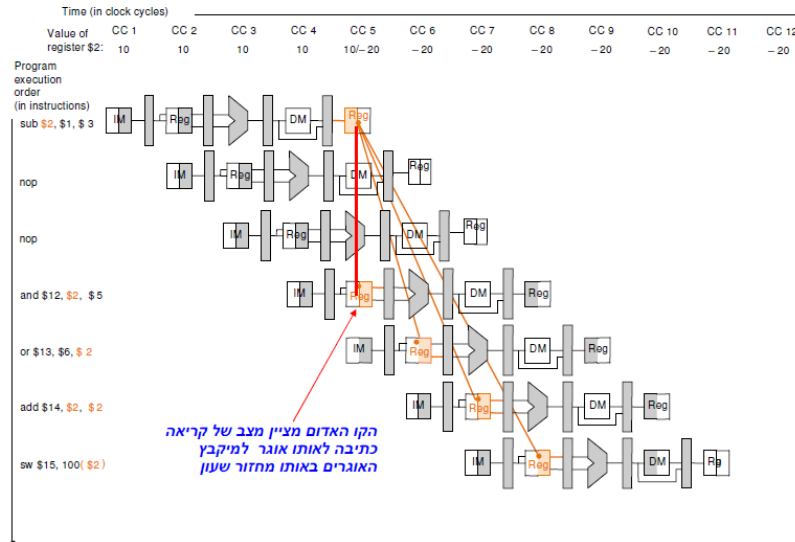
דוגמא לסיכון נתונים:

ישנן תוכניות בהן כותבים לתוך אוגר מסוים ובפקודה שלאחר מכן רוצים להוציא את הנתון מתוך האוגר. במעבד עם צנרת זו בעיה. הכתיבות לאוגר נעשות רק במחזור השעון החמישי בעוד שקריאות מהאוגר נעשות במחזור השעון השני. לכן כשמנסים להוציא את הנתון מהאוגר הוא עדיין לא נמצא שם. בתרשים מופיע רצף פקודות המשתמשות באוגר \$2. ע"פ השורה הראשונה sub מחליפה את ערך אוגר 10 ל-20. החלפה זאת נכנסת לתוקף רק במחזור השעון החמישי ולכן רק פקודת sw תתבצע באופן תקין.



טיפול בסיכונים ע"י nop:

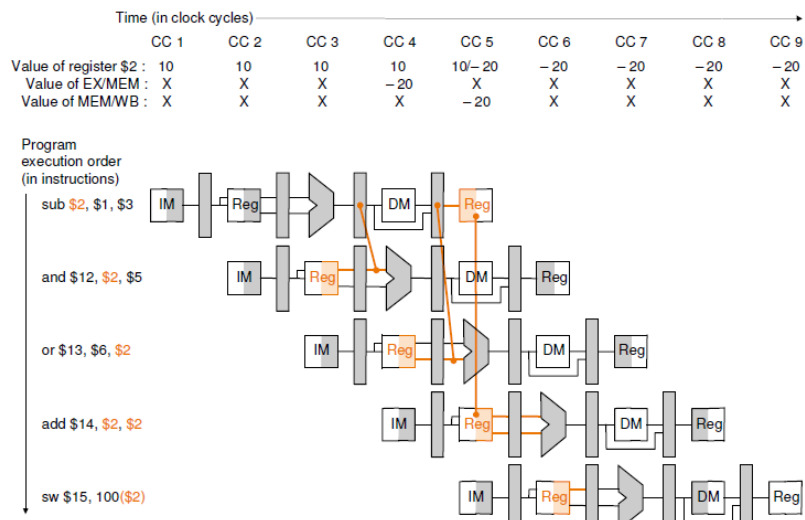
```
sub $2, $1, $3
nop
nop
nop
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```



זהו יצירת השהיה ברמת התוכנה כדי לטפל בסיכונים מידע. במקרה של קריאה וכתיבה באותו מחזור שעות לאותו אוגר ניתן לוותר על קסם אחד. זה נקרא חציית מקבץ האוגרים. במקרה זה בחצי השעות הראשון נכתב לאוגר ובחצי השני נקרא ממנו. יש לשים לב שחציית מקבץ האוגרים היא רק כאשר יש הפרש של שתי פקודות.

טיפול בסיכונים ע"י מנגנון העברה קדימה (Forwarding):

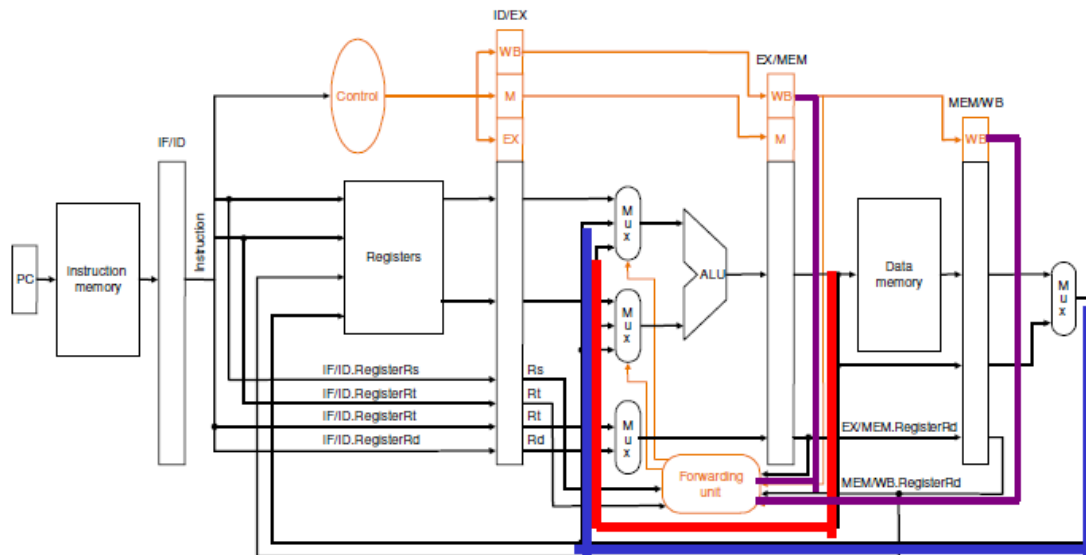
נשים לב כי הערך שצריך להיכתב לאוגר קיים כבר באוגרי הצנרת אחרי שעברנו את ה-ALU ומי שצריך אותו זה אותו ALU במחזור שעות הבא או במחזור שעות מאוחר יותר. לכן במקרים בהם לא נספיק לכתוב לאוגר ולקרוא ממנו נעשה העברה קדימה. כלומר נעביר את ערך האוגר ישירות ל-ALU. כאשר הנתון נמצא באוגרי הצנרת אך עדיין לא הגיע אל מקבץ האוגרים זהו המצב בו נעשה העברה קדימה וניתן לראות זאת בתרשים.



מקרים של Forwarding :

העברה קדימה מבוצעת במחזור השעון השלישי, כלומר בשלב ה-Execute. המקרים בהם מבוצעת העברה :

- (1) ID/EX.rs=EX/MEM.rd – כלומר במקרה ש-rd של הפקודה הקודמת שווה ל-rs של הפקודה הנוכחית (שנמצאת בשלב הפענוח) או נשתמש בערך של ה-ALUOut של הפקודה הקודמת ולא בערך של האוגר הכללי (GPR – General Purpose Register).
- (2) ID/EX.rs=MEM/WB.rd – כלומר במקרה ש-rd של הפקודה הקודמת שווה ל-rs של הפקודה הנוכחית (שנמצאת בשלב הפענוח) או נשתמש בערך של ה-ALUOut של הפקודה הקודמת ולא בערך האוגר הכללי.
- (3) באופן זה לאותם מצבים בהם ID/EX.rt=EX/MEM.rd ול- ID/EX.rt=MEM/WB.rd.



תנאים לביצוע העברה קדימה :

- (1) EX/MEM Hazard (מעביר קדימה מפקודה קודמת) :
 $\text{if (EX/MEM.RegWrite and (EX/MEM.Rd} \neq 0) \text{ and (EX/MEM.Rd=ID/EX.Rs))}$
 $\text{ForwardA}=10 \Rightarrow$
 $\text{if (EX/MEM.RegWrite and (EX/MEM.Rt} \neq 0) \text{ and (EX/MEM.Rt=ID/EX.Rs))}$
 $\text{ForwardB}=10 \Rightarrow$
- (2) MEM/WB Hazard (מעביר קדימה משתי פקודות אחורה) :
 $\text{if (MEM/WB.RegWrite and (Mem/WB.Rd} \neq 0) \text{ and (EX/MEM.Rd} \neq \text{ID/EX.Rs))}$
 $\text{and (MEM/WB.Rd=ID/EX.Rs))}$
 $\text{ForwardA}=01 \Rightarrow$
 $\text{if (MEM/WB.RegWrite and (Mem/WB.Rt} \neq 0) \text{ and (EX/MEM.Rt} \neq \text{ID/EX.Rs) and}$
 $\text{(MEM/WB.Rt=ID/EX.Rs))}$
 $\text{ForwardB}=01 \Rightarrow$

התנאי הבולט הוא לצורך טיפול בסיכון כפול (Double Hazard). כלומר אם יש סיכון מהפקודה הקודמת ומהפקודה הקודמת אליה או נבחר את המידע מהפקודה האחרונה בלבד שכן הוא הכי מעודכן.

הסבר קצר :

אם אחד התנאים של נתון יותר עדכני אכן מתקיים צריך שיתקיימו עוד שני תנאים כדי שתהיה הצדקה להפעלת מנגנון העברה קדימה :

- (1) שתהיה כתיבה לאוגר rd וקו בקרה RegWrite יהיה דולק.
- (2) שהכתיבה לא תהיה לאוגר \$0

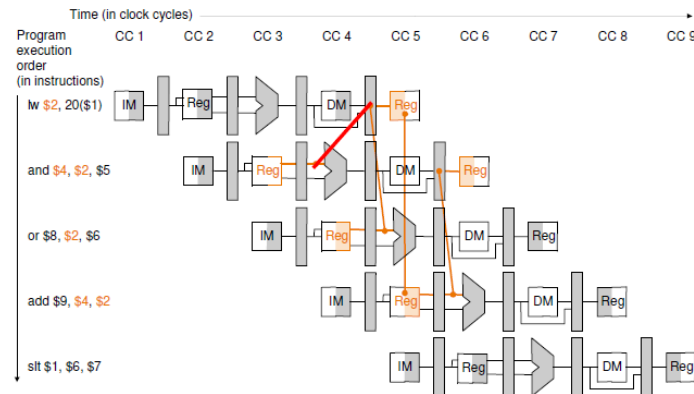
עבודת יחידת העברה קדימה:

- נשים לב כי יחידה זו מקבלת מספרי אוגרים ובודקת אם קיימים בצנרת.
- בודקים האם אנו קוראים מאוגר שעדיין לא עודכן. במידה ועדיין לא עודכן בשלבים הבאים.
 - הערך שאותו יש להעביר קדימה יכול להיות מאוגר הצנרת EX/MEM (בשלב MEM) או מאוגר הצנרת MEM/WB (בשלב WB).
 - קו בקרה RegWrite מציין לנו האם אנו כותבים לאוגר.
 - אנו צריכים להעביר קדימה את ערכי אוגרי rs ו- rt .
 - כתיבה לאוגר \$0 אסור שתהיה מועברת קדימה, היא אינה חוקית – ערך האוגר תמיד 0.
 - אם אנו יכולים להעביר משני אוגרי הצנרת אז נבחר בערך החדש ביותר שנמצא ב-EX/MEM.

מקרה מיוחד – פקודת lw:

פק' lw מביאה נתון מהזיכרון אל תוך אוגר. הנתון מגיע אל אוגרי הצנרת רק אחרי מחזור שעון רביעי. קריאת האוגרים בפקודת המכונה העוקבת לפקודת lw נעשית במחזור השעון השני ולכן אם הפקודה העוקבת משתמשת באוגר ש- lw צריכה לכתוב לתוכו אנו בבעיה. לא ניתן לעשות העברה קדימה כי הנתון עדיין לא קיים באוגרי הצנרת. מקרה כזה ניתן לפתור ע"י שתי דרכים:

- ברמת התוכנה השתלת nop ($sll \$0, \$0, 0$) או ע"י אופטימיזצור שמשבץ פקודה אחרת שלא גורמת לסיכון נתונים.
- ברמת החומרה ע"י המעבד שיבצע השהיה ($stall$).

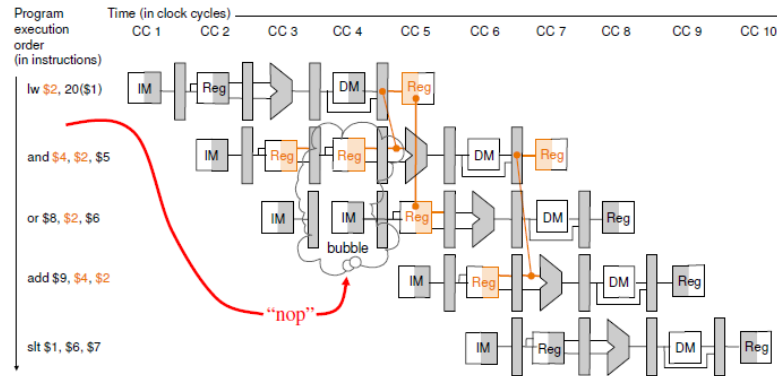


יחידת איתור סיכונים (Hazard Detection Unit - HDU):

ההשהיה בחומרה נעשית במעבד ע"י HDU לאחר שהוא מזהה מקרה סיכון נתונים. היחידה מבצעת את הפעולות הבאות:

- נתינת ערך אפס לכל קווי הבקרה שיוכנסו לאוגר ID/EX וכך היא למעשה הופכת את הפקודה ל- nop .
- שמירת הערך באוגר הצנרת IF/ID על מנת שהפקודה היושבת שם תבוצע עוד פעם (מאחר ולא נתנו לפקודה להגיע ל-ID/EX אלא החלפנו אותה ב- nop). זה מבוצע ע"י קו הבקרה IF/IDwrite אשר לא מאפשר כתיבה לאוגר.
- היות וע"י הקפאה של IF/ID ביצענו שוב את הפקודה בשלב הפענוח יחידת HDU צריכה שוב לבצע את הפקודה שהייתה בשלב Fetch. כלומר גם ערך אוגר PC ישמר וזאת ע"י קו $PCwrite$.

למעשה המעבד מפסיק מחזור שעון אחד. מקרה זה נקרא בועה בצנרת. כמו כן מכיוון ש-PC לא התקדם במחזור השעון של ההשהיה הפקודה העוקבת לפק' lw מבצעת שוב את מחזור השעון השני והפקודה שאחריה שוב מבצעת את מחזור השעון הראשון. במקרה זה גם לאחר ההשהיה יש צורך ביחידת העברה קדימה.

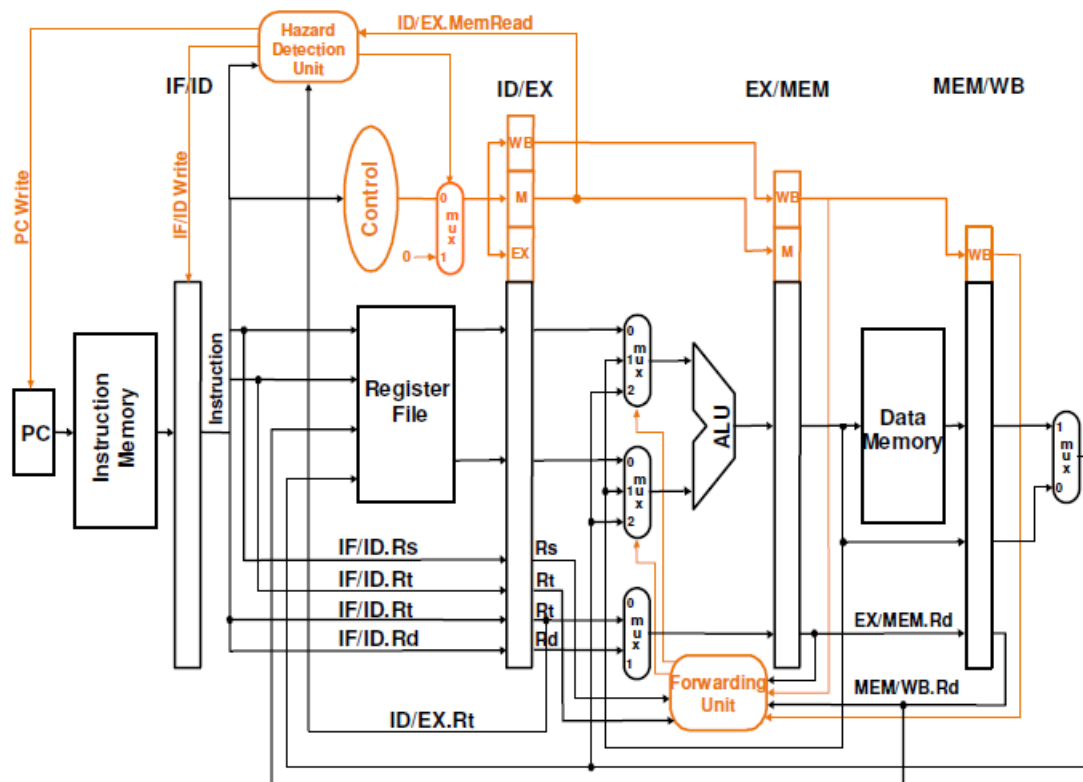


תנאים לעבודת HDU:

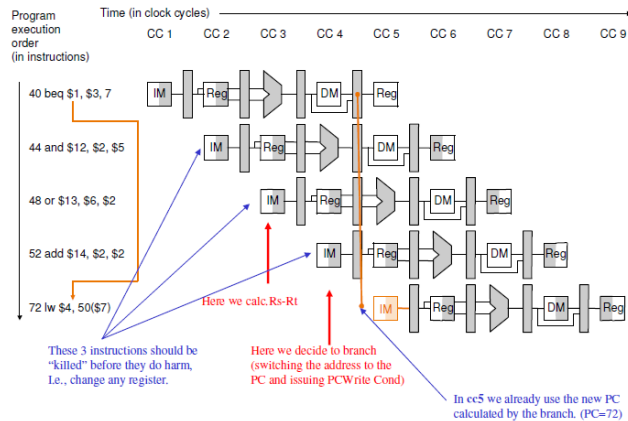
if (ID/EX.MemRd) and (ID/EX.Rt==IF/ID.Rs) or (ID/EX.Rt==IF/ID.Rt)

Stall⇒

תמונה מסכמת של טיפול בסיכונים ע"י HDU ו-Forwarding:



סיכויי בקרה (Control Hazards – Branch):



סיכויי בקרה מתרחשים כאשר במקום מסוים בתוכנית יש קפיצה לכתובת אחרת. פקודות המכונה שנמצאות אחר פקודת הקפיצה לעיתים אינן אמורות להתבצע בשלב זה. הקפיצה עצמה נעשית רק במחזור שעון רביעי של פקודת branch ולכן פקודות שבאות לאחר branch יספיקו להיכנס לצנרת. פקודות אלו מיותרות ואסור לבצען במקרה של קפיצה. בעיה זו ניתן לפתור ע"י השתלת 3 nop לאחר כל פקודת branch ללא קשר אם תילקח או לא ובכך להימנע מהכנסת פקודות מיותרות לצנרת. בפתרון חומרתי בשונה מסיכויי נתונים עיכוב בביצוע הפקודה אינו מספק ויש לבטל את ביצוע פקודות המכונה המיותרות שהמעבד התחיל לבצע. תהליך זה נקרא שטיפה (flush).

: Flush

בשטיפה אנו דואגים שה-PC ימשיך להתקדם אך פקודות המכונה שהחלו להתבצע לא תכתבנה לשום אוגר או זיכרון. הפקודות המיותרות יתקדמו בצנרת אך לא יגרמו לשינויים.

מסקנה מהתהליך:

במידה שהקפיצה תעשה רק במחזור השעון הרביעי יש צורך לבטל שלוש פקודות מכונה. כל פקודה מבוטלת מעכבת את הצנרת במחזור שעון אחד. כדי ליעל אנו צריכים להקדים את הקפיצה למחזור שעון מוקדם יותר כך נוכל לבטל פחות פקודות ולשפר את ביצועי המעבד.

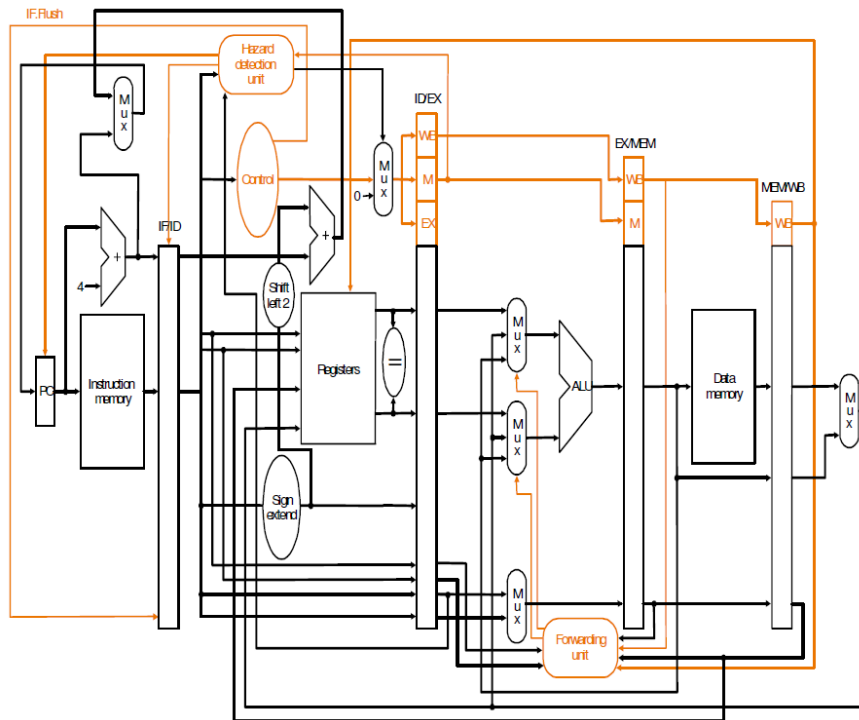
הקדמת הבדיקה של branch לשלב ID:

נזיז את הקפיצה לסוף המחזור השעון השני. לצורך זה נקדים את הפעולות של-beq מבצעת במחזור השלישי:

- חישוב כתובת קפיצה צפויה (הזזת המחבר לשלב השני וחיווטו ל-IF/ID). חישוב הכתובת אפשרית מכיוון שבתחילת המחזור השני כבר קיימים באוגר IF/ID הערך של PC+4 וגם ערך ההיסט הנדרש. יש להזיז את המחבר (Adder) שמצבע את החיבור כך שהכניסות שלו יהיו מחוברות אל אוגר IF/ID במקום אוגר ID/EX.
- בדיקה האם יש צורך בקפיצה (הוספת חומרה חדשה פשוטה וזריזה). הבדיקה מבוצעת במקור ב-ALU והוא נידרש למחזור השעון השלישי של הפקודות lw, sw, R-Type ולכן לא ניתן להזיזו. נצטרך אם כן להוסיף במחזור השעון השני חומרה חדשה שיודעת לבדוק האם שני מס' הם שווים.

כלומר צמצמנו את הבעיה ל-nop בודד מה שמשאיר לנקות פקודה אחת שנמצאת במחזור הראשון. רק יש לשים לב כי ההשוואה במחזור השני מייצרת data hazard חדש כבר במחזור השני. יכול להיות שההשוואה הנדרשת ע"י beq היא בין אוגרים שערך מעודכן שלהם נמצא בצנרת ויש להעבירם קדימה. נשים לב כי לא ניתן להשתמש במנגנון העברה קדימה שאנו מכירים שכן הנתונים הועברו משם ל-ALU וכן יש צורך להעביר את הנתונים לחומרה החדשה (גם נשים לב בין ה-ALU שנמצא במחזור השלישי לבין החומרה החדשה שנמצאת במחזור השני). במקרה זה ישנם שני מצבים שיש להתייחס אליהם:

- במקרה של פקודת lw קודמת שכותבת אל תוך האוגר שאיתו נעשית ההשוואה של beq יש צורך להשהות את הצנרת לשני מחזורי שעון ולאחר רק מחזור שעון אחד כפי שהיה במקרה של סיכויי נתונים ל-ALU ואז להעביר קדימה את הנתונים.
- במקרה של פקודה קודמת מסוג R-Type שכותבת אל תוך האוגר שאיתו נעשית ההשוואה של beq יהיה צורך להשהות את הצנרת מחזור שעון אחד ואז להעביר קדימה את תוצאת החישוב של ה-ALU וזאת בשונה במקרה של סיכויי נתונים ל-ALU שהספיק מנגנון העברה קדימה בלבד.



: Branch Delay Slot

ההגדרה המקורית של Branch היא שאם אנו מבצעים Branch כל פקודה שבאה אחריו אינה מתבצעת בטעות.

ההגדרה החדשה היא שגם אם מתבצע branch וגם אם לא הפקודה הראשונה שבאה אחריו מבוצעת תמיד (היא נקראת Branch Delay Slot) ובאחריות המהדר (לרוב) או המתכנת לקבוע פקודה מתאימה. זוהי בעצם מוסכמה של תוכנה עם החומרה.

: יתרונות וחסרונות:

- 1) במקרה הגרוע ביותר אנו שותלים nop ב-Branch Delay Slot.
- 2) במקרה הטוב אנו מוצאים פקודה שיכולה להיכנס ב-Branch Delay Slot ושאינה משנה את זרימת התוכנית.
- א. שינוי סדר פקודות זוהי דרך מוכרת לשיפור ביצועים.
- ב. הקומפיילר (המהדר) צריך להיות מספיק טוב כדי למצוא פקודות כאלו. לרוב מציאת הפקודה המתאימה קוראת 50% מהזמן ואם לא מושם nop.
- ג. פקודות j גם בעלות Delay Slot.
- ד. פקודות lw גם בעלות Delay Slot בגלל סיכויי נתונים.

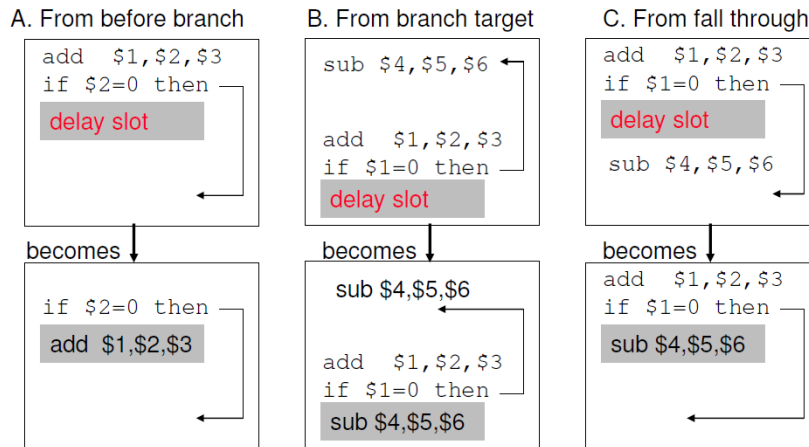
: תזמון Branch Delay Slots

בהנחה שאופן העבודה הינו Delay Slot אזי למהדר יש אופטימיזציה המחפש את הפקודה המתאימה לביצוע Delay Slot. תהליך זה של שיכתוב התכנית ביחס לקובץ המקור נקרא תזמון (Scheduling).

- 1) מקרה A – זהו המקרה הפשוט ביותר בו נמצאה פקודה שאינה תלויה בהחלטת beq. היא ממלאת את ה-Delay Slot ומקטינה (IC) Instruction Count.
- 2) מקרה B – האופטימיזציה החליט שברוב המקרים ה-branch taken ולכן מצא פקודה למילוי מהפקודות שבאות לאחריו. במקרה זה יכול להיווצר הגדלה של IC.
- 3) מקרה C – האופטימיזציה החליט שברוב המקרים ה-branch not taken ולכן מצא פקודה למילוי שהיא הפקודה שלאחר branch. במקרה זה יכול להיווצר הגדלה של IC.

: הערות לתזמון:

- 1) אנו מניחים כי במקרים B,C המהדר יודע לקבוע בצורה מובהקת taken / not taken.
- 2) אנו מניחים במקרים B,C כי שהפקודה למילוי (sub \$t4, \$t5, %t6) לא תגרום לטעות לוגית בתוכנית אם תבוצע בניתוב בקרה הפוך מהחזוי.



שיטת חיזוי סטאטית עבור צנרת עמוקה (Static Branch Prediction for Deeper Pipelines):

פותרת סיכויי בקרה ע"י הנחת תוצאת ה-branch: taken / not taken

- 1 Predict Not Taken – שיטה זו יעילה עבור לולאות עליונות (Top of the Loop) שבהן התנאי נמצא בראשית הלולאה. במקרה זה היא יוצרת השהיית קפיצה.
- 2 Predict Taken – במקרה זה החיזוי הינו תמיד taken. היא יוצרת השהיה של מחזור אחד (בהנחה ש-beq בשלב השני).

הערה: ככל שה-branch penalty גדל (עבור צנרות עמוקות יותר) חיזוי סטאטי יפגע בביצועים. הפתרון הוא ע"י הוספת חומרה שתנסה לחזות בצורה דינאמית.

שיטת חיזוי דינאמית (Dynamic Branch Prediction):

הרעיון בחיזוי דינאמי הינו לקבל את ההחלטה האם לבצע קפיצה בזמן ריצת התוכנית על סמך נתונים המתקבלים בזמן הריצה. עד עתה הנחנו כי הקפיצה אינו מתבצעת. אם טעינו ואכן התבצעה קפיצה השגיאה תוקנה ע"י ביטול הפקודה שנכנסה לצנרת. במקרה של חיזוי דינאמי אנו נניח שאכן מתבצעת קפיצה ורק אם היא לא מתבצעת נבטל את הפקודה שנכנסה לצנרת. ההנחה היא שבדרך כלל התנהגות של פקודת branch חוזרת על עצמה, או שפקודת branch מסוימת גורמת כמעט תמיד לקפיצה או שפקודת branch מסוימת כמעט ולא תגרום לקפיצה בתוכנית מסוימת.

: Branch History Table (BHT)

זהו סוג של באפר שנמצא בשלב IF שמתייחס לביטים הנמוכים של PC ומכיל ביט שמועבר לשלב ID ע"י אוגר צנרת IF/ID שמצביע אם ה-branch נלקח או לא בפעם האחרונה שבוצע. סיביות החיזוי תמצאנה בזיכרון קטן שנקרא טבלת חיזוי קפיצות BHT. (ניתן גם למקם בשלב השני והוא כולל עמודות כתובת יעד ועמודות היסטוריה). הערות:

- 1 סיבית החיזוי יכולה לחזות לא נכון אם בגלל שהפעם קרה הפוך מפעם קודמת או אם הסיביות הנמוכות של פקודת branch זה דומות לפקודת branch קודמת. זה לא משפיע על נכונות הביצוע אלא על יעילות הביצוע. החלטות branch נעשות במחזור השני לאחר שהפקודה שהובאה היא אכן branch ולאחר שנבדק סיבית החיזוי.
- 2 אם החיזוי שגוי אז יישטף הפקודה הלא מתאימה בצנרת, הצנרת תאוחל עם הפקודה הנכונה וסיבית החיזוי תתהפך. כלומר אם נטעה המחיר הוא בועות בצנרת.

דיוק חיזוי של סיבית בודדת:

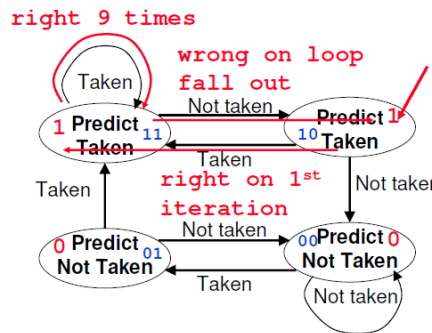
- 1 סיבית חיזוי בודדת תהיה שגויה פעמיים אם branch not taken.
 - א. מניחים כי סיבית חיזוי היא 0 ובקרת הלולאה היא בתחתית הלולאה.
 - ב. בפעם הראשונה בלולאה סיבית החיזוי טועה והופכת את ערכה ל-1.
 - ג. כל עוד ה-branch taken חיזוי נכון (כלומר כל עוד אנחנו בלולאה).
 - ד. ביציאה מהלולאה סיבית החיזוי שוב טועה שכן עכשיו branch not taken ומשנה את ערכה חזרה ל-0.
- 2 אם כן באופן סטטיסטי כאשר נבצע את הלולאה 10 פעמים יש לנו דיוק חיזוי של 80% עבור branch שמתרחש 90% מהזמן.

דיוק חיזוי של 2 סיביות:

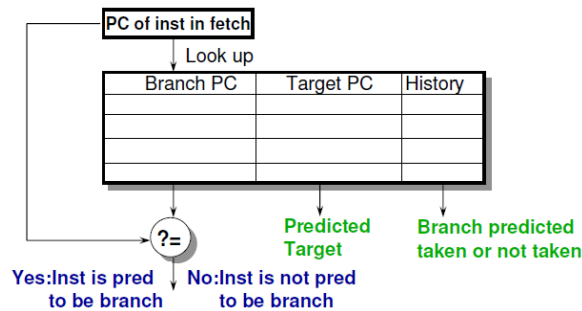
- (1) שיטה זו מביאה לנו דיוק של 90% מאחר שהחיזוי צריך להיות שגוי פעמיים לפני שמשנים את ערך הסיביות.
- (2) הטבלה שומרת גם את המצב הראשוני של מכונת המצבים.

הסבר:

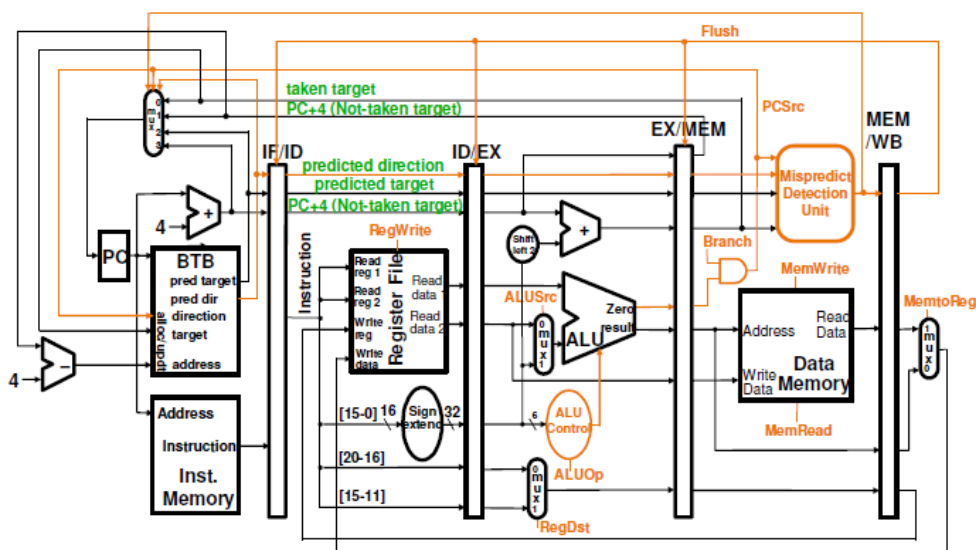
אם פקי branch בדרך כלל קופצת ובמקרה אחד לא הייתה קפיצה, הניחוש עדיין ימשיך להיות קפיצה. רק שתי פעמים רצופות בהן branch not taken יגרמו לחומרה לשנות את הניחוש ל-not taken. אותו רעיון תקף גם למצב ההפוך.



: Branch Target Buffer (BTB)



זהו סוג של זיכרון מטמון ששומר בנוסף להיסטוריית החיזוי גם את כתובת הקפיצה, בזמן שזיכרון הפקודות מושך את הפקודה הבאה, כך שבמידה ופקודת ה-branch כבר נמצאת בטבלה ניתן לעדכן כבר בשלב ההבאה את הכתובת החזויה של PC.



תרשים זה מבוסס על ביצוע branch במחזור הרביעי. ה-BTB נמצא במחזור הראשון ומבצע עדכון PC במידה ויש לו את הנתונים הדרושים (במידה ועדיין אין נלקח PC+4, ניתן לראות זאת כ-not taken). המידע הרלוונטי:

- PC+4 (1)
- Target Address (2)
- Prediction History (3)
- Prediction Decision (4)

מועבר במורד אוגרי הצנרת עד ליחידת עדכון החיזוי (MDU (Mispredict Detection Unit). יחידה זו בודקת האם הפקודה אשר מבוצעת במקביל להתקדמות המידע אכן התנהגה בהתאם לחיזוי כאשר במידה והייתה טעות דואגת לבצע flush לשלושת השלבים הקודמים ולהופכם לבועות, זאת בנוסף להכנסת ערך נכון ל-PC. היא גם אחראית על עדכון שוטף של BTB. גם פקודת j מעודכנת ב-BTB ובמקרה זה לאחר שהפקודה מבוצעת פעם אחת יש למעשה חיזוי וודאי של 100% (בהנחה ש-branch target address מכיל את כל הסיביות).

שכלול יכולות החיזוי:

ככל שהצנרת עמוקה יותר והבדיקה של נכונות החיזוי נמצאת בשלב מתקדם בצנרת כך הקנס על טעות בחיזוי גדל. לכן לא מסתפקים בהיסטוריה של 2 סיביות ומשקים מאמצים רבים בשיפור יכולות חיזוי:

- (1) חיזוי ע"י התאמה (Correlating Predictor) – בנוסף להתנהגות של branch ספציפי נשמר גם מידע על ההתנהגות הכללית של פקודות הקפיצה המותנית בתכנית ומידע זה משוכלל עם ההתנהגות המקומית.
- (2) Tournament Branch Predictor – זוהי שיטה חדשה יותר אשר האסטרטגיה הננקטת הינה בזמן ריצה להשוות בין מס' טכניקות חיזוי ולבחור עבור כל פקודת branch את זו הטובה יותר.

סיכום סיכונים:

- (1) סיכונים נתונים:
 - א. העברה קדימה מפקודה קודמת
 - ב. העברה קדימה משתי פקודות אחורה
 - ג. העברה קדימה של אוגר GPR "שקוף" משלוש פקודות אחורה
 - ד. אם לא ניתן להעביר קדימה (לאחר lw) אנו משהים את הצינור ע"י nop ומקפויים פעולת IF/ID ו-PC מחזור שעון אחד.
- (2) סיכונים בקרה:
 - א. אם branch taken אנו שוטפים את הפקודה שבאה לאחריו (נמצא ב-IF/ID).
 - ב. חיזוי branch
- (3) הערות:
 - א. במעבד MIPS אין flush. למהדר יש את האופציה לבחור פקודות מתאימות שיגיעו לאחר ה-branch. כלומר עובדים בשיטת Branch Delay Slot.
 - ב. במעבד MIPS אין השחיות lw. במקרה זה פעולת המהדר נקראת Delayed Load.

שיפורים נוספים בצנרת:

- (1) Superscalar – מבצעים יותר מ-5 שלבים של צנרת.
- (2) Dynamic Pipeline Scheduling – במקום בועות משנים את סדר ביצוע הפקודות כדי למלא מרווחים אם ניתן.
- (3) Superscalar – ביצוע שתי פקודות במקביל. כלומר הכפלת החומרה.

מנגנון הפסיקה / חריגה:

הגדרות:

Exception – חריגה: זהו אירוע בלתי צפוי הקורה במעבד.

Interrupts – פסיקה: זהו אירוע הגורם לשינוי בלתי מתוכנן בבקרת הזרימה אשר מגיע מחוץ למעבד. מוסכמה ב-MIPS שהמושג חריגה כולל את כל האירועים ומושג הפסיקה מתייחס רק לאירועים חיצוניים.

ההבדל בין פסיקה וחריגה:

סוג האירוע	מהיכן נובע	טרמינולוגיה ב-MIPS
בקשת של קלט ופלט	חיצוני	פסיקה
Arithmetic Overflow	פנימי	חריגה
שימוש בפקודה לא מוגדרת	פנימי	חריגה
בעיות בחומרה	פנימי או חיצוני	חריגה או פסיקה

רכיב הטיפול בחריגים:

- 1) רכיב הטיפול בחריגים נותן מענה הן לחריגות הנגרמות משגיאות פנימיות תוך כדי ביצוע פקודות והן לפסיקות חיצוניות הנגרמות מרכיבי קלט פלט.
- 2) במעבד MIPS, ישנו רכיב אשר הינו חלק מה-CPU ונקרא coprocessor 0 שתפקידו להקליט את המידע שהתוכנה צריכה כדי לנהל את החריגות והפסיקות.
- 3) תוכנת ה-MARS יודעת ליישם חלק מאוגרי ה-coprocessor 0.

אוגרי 0 coprocessor:

שם	מס' אוגר	שימוש
BadVAddr	8	כתובת של זיכרון שבה התקיימה שגיאה
Status	12	ביטים של enable ו-mask עבור פסיקות.
Cause	13	ביטים עובר סוג חריגות ופסיקות בתור.
EPC	14	כתובת הפקודה אשר גרמה לשגיאה.

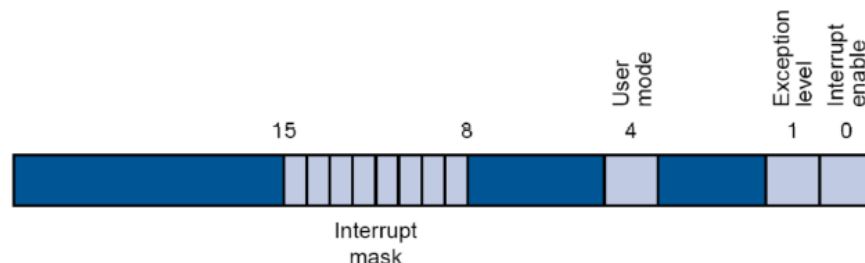
טיפול בחריגות:

Interrupt handler (ISR – Interrupt Service Routine): זהו חלק תוכנתי אשר רץ כתוצאה מפסיקה או חריגה.

ישנן שתי דרכים לטפל בחריגות:

- 1) Cause Register – אוגר המספק מידע לגבי אילו חריגות ופסיקות נמצאות בתהליך. MIPS עובד בשיטה זו.
- 2) Vectored Interrupts – זוהי טכניקה בה הגורם המפריע מכווין את המעבד לתהליך המפריע לריצת התוכנית.

מבנה ה-Status Register:



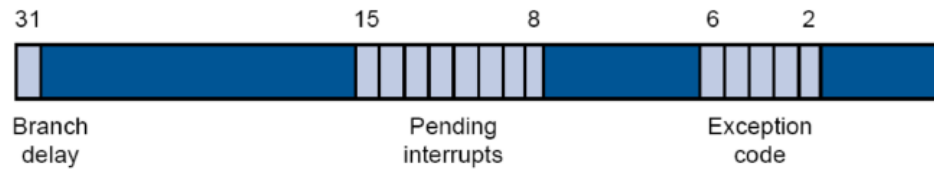
Interrupt Mask: סיביות 8-15 המשמשות לתיאור הרמה (Level) בה התרחשה פסיקה. 2 ביטים עבור רמת תוכנה ו-4 ביטים עבור רמות חומרה.

User Mode: מצב המעבד – האם עובד במצב Kernel '0' (ליבה) או במצב משתמש '1'.

Exception Level: מוגדרת כאשר ישנה חריגה. מטרתם למנוע הפרעה לרכיב הטיפול בחריגים.

Interrupt Enable: מאפשר או לא מאפשר פסיקות.

מבנה ה-Cause Register:



Exception Code: סיביות 2-6 (unsigned) אשר שומרות את סיבת הפסיקה (או מצביעה על חריגה).

Pending Interrupts: סיביות 8-15 המשמשות לתיאור הרמה (Level) בה התרחשה פסיקה.

Code	Name	Description
0	INT	Interrupt
4	ADDR1	Load from an illegal address
5	ADDRS	Store to an illegal address
6	IBUS	Bus error on instruction fetch
7	DBUS	Bus error on data reference
8	SYSCALL	syscall instruction executed
9	BKPT	break instruction executed
10	RI	Reserved instruction
12	OVF	Arithmetic overflow

a. Codes from 1 to 3 are reserved for virtual memory, (TLB exceptions), 11 is used to indicate that a particular coprocessor is missing, and codes above 12 are used for floating point exceptions or are reserved.

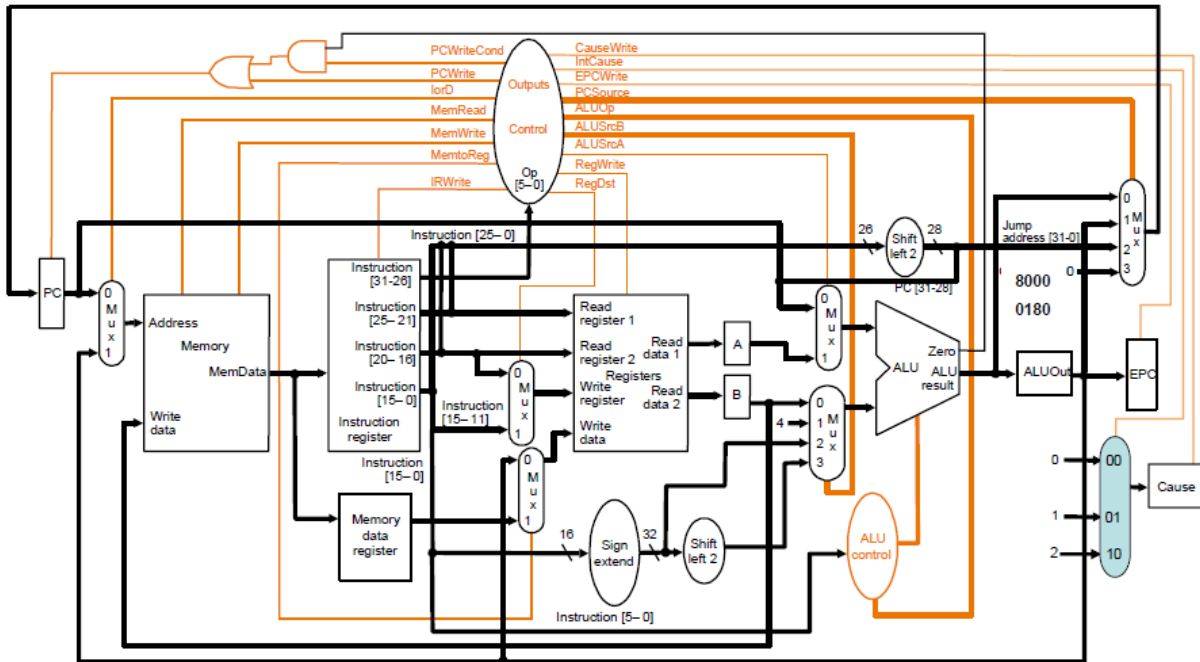
רכיב הטיפול בחריגות ופסיקות:

- 1) חריגות ופסיקות גורמות ל-MIPS לקפוץ לכתובת **0x80000180** (בליבה - Kernel) אשר נקראת Exception Handler.
- 2) קוד זה מאבחן את סיבת החריגה או הפסיקה וקופץ לנקודה המתאימה במערכת ההפעלה (לנקודה שבה מערכת ההפעלה מטפלת בכך). מערכת ההפעלה פועלת ע"י הפסקת התהליך הגורם לפסיקה או לחריגה או ע"י ביצוע פעולות מתאימות.

מנגנון הפסיקה או חריגה במעבד רב מחזורי:

אנו נבחן שני סוגי פסיקות או חריגות:

- (1) פקודה לא מוגדרת (Opcode לא קיים)
- (2) Arithmetic Overflow (גלישה בפעולת חיבור חיסור)
- (3) פקודה lw / sw שאינה מתחלקת ב-4.



כאשר מתבצעת חריגה יש להפסיק את ביצוע התוכנית ולקפוץ לקוד של מערכת ההפעלה המטפל בחריגות. קוד זה נמצא באופן קבוע בכתובת 0x80000180. בנוסף יש להודיע למערכת ההפעלה את סיבת הקפיצה אליה. כדי לידע את מערכת ההפעלה בסיבת הקפיצה נוסף אוגר בשם Cause שזהו מטרתו.

אוגר Cause: מקבל '0' כאשר החריגה נוצרה בעקבות Opcode ומקבל '1' כאשר החריגה נוצרה בעקבות גלישה של חיבור או חיסור.

בתרשים נוסף גם אוגר EPC. תפקידו של אוגר זה לשמור את הפקודה בה הופסקה התוכנית זאת מכיוון שינן מערכות הפעלה שמניחות כי במקרים מסוימים אפשר יהיה לתקן את התקלה ולחזור ולהריץ את התוכנית מהמקום שהופסקה.

אוגר EPC: מחובר ל-ALU מכיוון שבמקרה של חריגה ה-ALU מחסר 4 מאוגר PC לפני כתיבה ל-EPC כדי ששכשהמעבד יחזור אל התכנית שהופסקה הוא יבצע שוב את הפקודה שלא הצליח לבצע בפעם הקודמת.

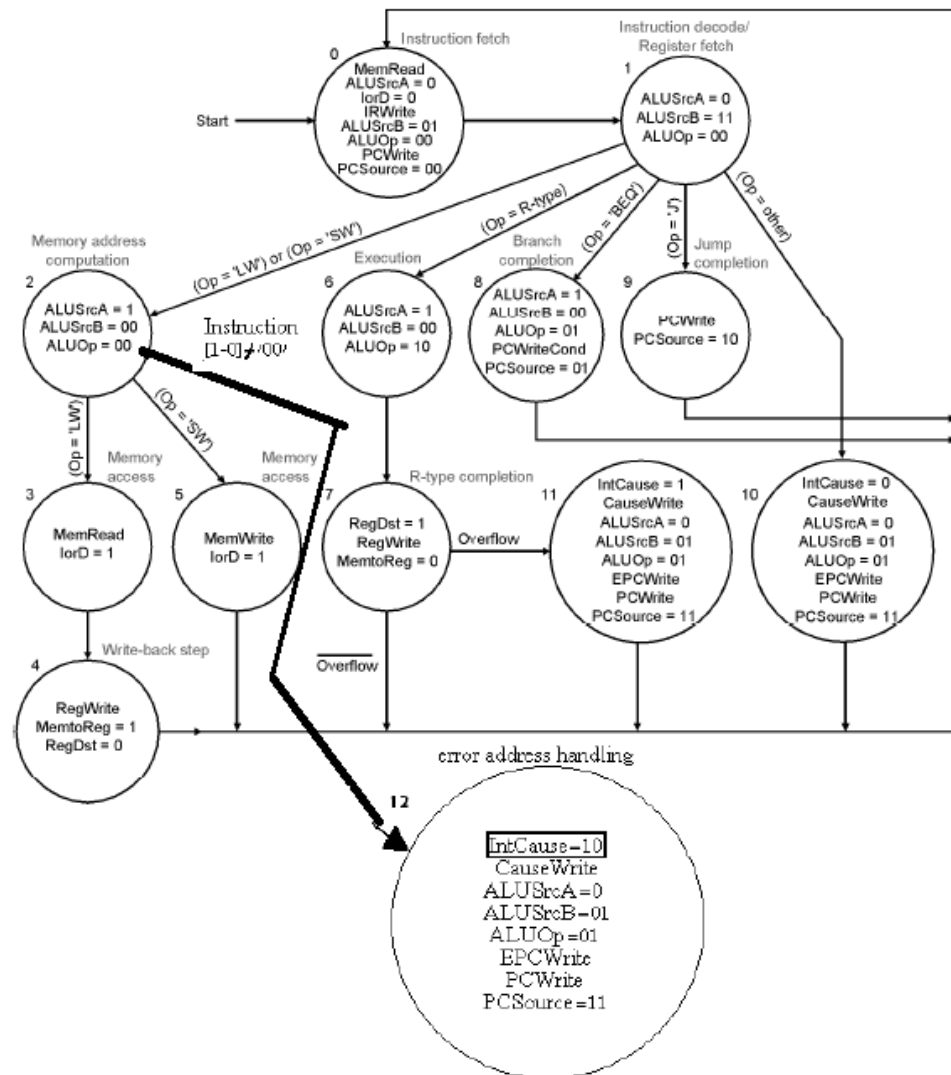
קווי בקרה שנוספו:

- (1) CauseWrite – קובע האם תהיה כתיבה לאוגר Cause.
- (2) EPCWrite – קובע האם תהיה כתיבה לאוגר EPC.
- (3) IntCause – קובע האם לכתוב את הקבוע 0 (Opcode לא מזוהה) או את הקבוע 1 (גלישה) באוגר Cause.

קו בקרה PCSource:

נוסף מקרה רביעי. הערך 11 ייצג את הקבוע 0x80000180 המייצג את כתובת הפונקציה של מערכת ההפעלה המטפל בחריגות.

שינוי מכונת המצבים:



מצב 10:

זהו המצב בו זוהתה פקודה בלתי מוגדרת. קווי הבקרה גורמים למעבד לכתוב סיבת חריגה ב-0 Cause, לשחזר את כתובת הפקודה שחרגה ולכתובה ב-EPC ולקפוץ למערכת ההפעלה. מצב 10 עוברים לפקודה הראשונה בפונקציה של מערכת ההפעלה ולכן עוברים למצב 0. זיהוי פקודה בלתי חוקי יכול להתבצע רק לאחר קריאת ה-Opcode כלומר רק במצב 1.

מצב 11:

זהו המצב בו זוהתה גלישה בפעולת חיבור או חיסור. קווי הבקרה גורמים למעבד לכתוב סיבת חריגה ב-1 Cause, לשחזר את כתובת הפקודה שחרגה ולכתובה ב-EPC ולקפוץ למערכת ההפעלה. בדומה למצב 10 גם מצב 11 עוברים לפקודה הראשונה בפונקציה של מערכת ההפעלה ולכן עוברים למצב 0. גלישה יכולה להתרחש רק תוכך כדי ביצוע R-Type, אחרי שה-ALU גמר לבצע את פעולתו כלומר רק במצב 7.

מצב 12:

זהו המצב שמטפל בחריגה עקב כתובת לא חוקית בזיכרון ב-lw / sw (כתובת שלא מתחלקת ב-4). היא מתבצעת במצב 2 של מכונת המצבים בעת חישוב הכתובת ב-ALU. נבחין בגלישה כאשר שתי הסיביות הנמוכות בתוצאת ה-ALU אינן 00. מצב זה נעבור למצב 12. במצב זה נדאג ל-PC=PC-4 ולשמור ערך זה באוגר EPC. ב-Cause נתעד את סיבת פסיקה (נניח 2) ולעדכן את PC בכתובת ה-Exception Handler (אופציה 3 ב-PCSource). נגדיל את מרבב Cause שיכלול את הערך 2 (קו הבוררים עכשיו הוא 2 סיביות).

מנגנון הפסיקה או חריגה במעבד צנרת:

במעבד צנרת חריגות ופסיקות זהו רק סוג אחר של מנגנון לשליטה בסיכונים (Control Hazard). מנגנון זה פועל במקרים הבאים:

- (1) גלישה עקב חיבור או חיסור
- (2) ביצוע של פקודה לא חוקית
- (3) בקשה של אמצעי קלט או פלט
- (4) שירות או תהליך של מערכת ההפעלה
- (5) תקלה חומרתית

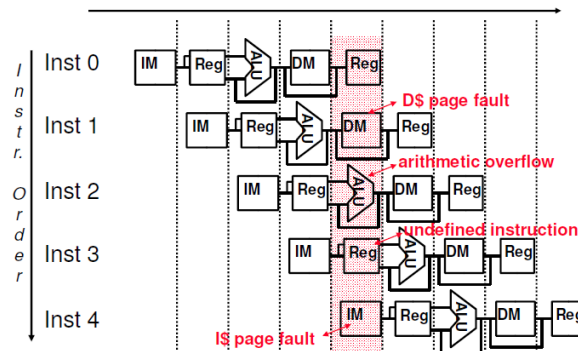
על הצנרת להפסיק את הפקודה הפוגעת תוך כדי ביצועה, לאפשר לכל הפקודות הקודמות לסיים, לנקות את כל הפקודות שבאוח אחרי הפקודה הפוגעת, הגדרת אוגר שישמור את סיבת ההפרעה, שמירה של כתובת הפקודה הבעייתית וקפיצה לכתובת ידועה מראש שמטפלת בשגיאות מסוג זה. מערכת ההפעלה היא המטפלת בשגיאות.

מיקום השגיאה בצנרת:

סיבה	שלב	סינכרוני?
גלישה עקב חיבור חיסור	EX	כן
פקודה לא חוקית	ID	כן
תקלה בזיכרון (או במיפוי)	IF, MEM	כן
רכיב קלט / פלט	any	לא
תקלה חומרתית	any	לא

יש לשים לב כי יכול להיווצר מצב של ריבוי חריגות ופסיקות בו זמנית במחזור שעון בודד.

טיפול במס' חריגות במקביל:

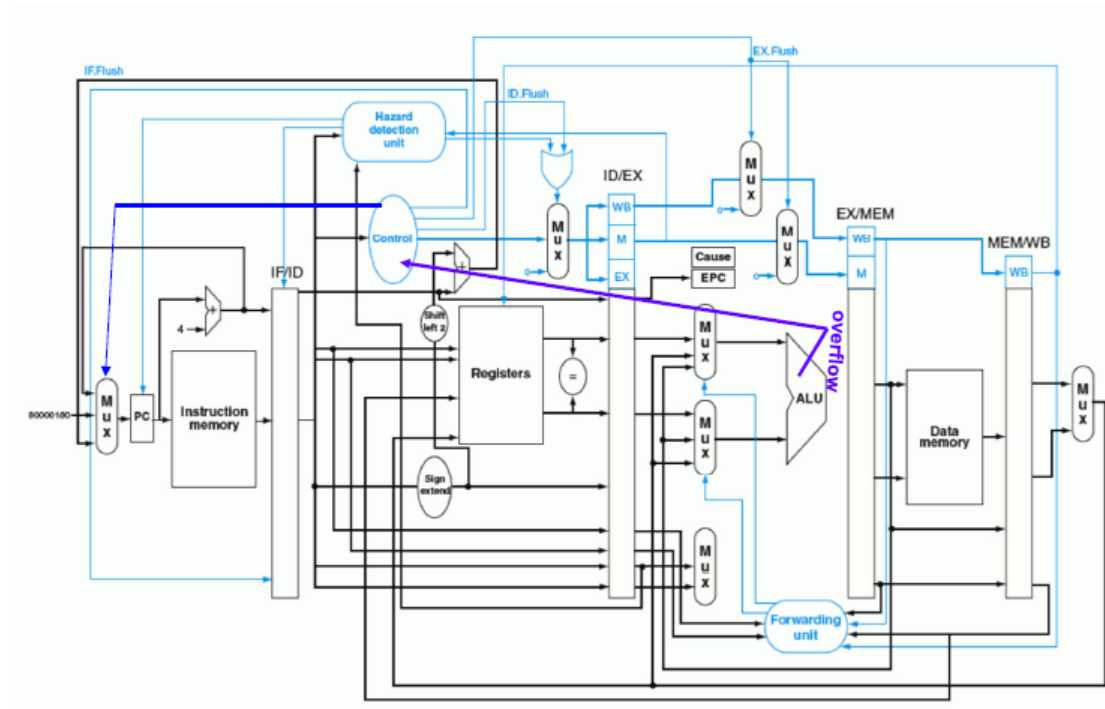


מכיוון שחריגות יכולות להיווצר במחזורי שעון שונים, לעיתים יכול להיווצר מצב בו שתי חריגות מגיעות בעת ובעונה אחת משתי פקודות מכונה שנמצאות בצנרת. במקרה כזה צריך ליצור מנגנון עדיפויות בין החריגות. מעבד MIPS במקרה זה מעדיף את החריגה שנגרמה עקב הפקודה המוקדמת ביותר שנכנסה אל הצנרת. כמובן קיימים מעבדים אחרים שמיישמים מנגנוני עדיפות אחרים.

תוספות ל-MIPS לצורך טיפול בחריגות:

- (1) אוגר Cause – רכיב חומרתי שתפקידו לשמור את סיבת השגיאה.
- (2) CauseWrite – קו בקרה שמטרתו לסמן לבקר שיש לרשום לאוגר Cause.
- (3) אוגר EPC – רכיב חומרתי שמטרתו לשמור את כתובת הפקודה הבעייתית.
- (4) EPCWrite – קו בקרה שמטרתו לסמן לבקר שיש לרשום לאוגר EPC.
- (5) הוספת אופציה לבורר של PC המאפשר לקפוץ לכתובת המטפלת בחריגות (כתובת 0x80000180).
- (6) קווי בקרה שמטרתן לרוקן את הפקודה הפוגעת וכל הפקודות שבאות אחריה.

גלישה אריתמטית בצנרת:



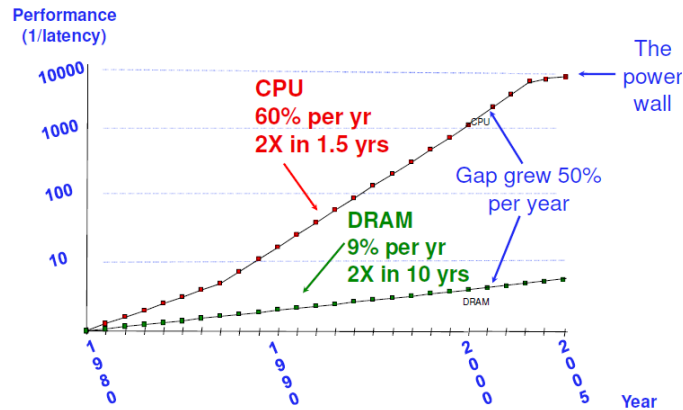
אם מתרחשת חריגה עקב גלישה (Overflow) בצנרת היא תתרחש במחזור השעון השלישי. לאחר זיהוי החריגה נשתמש באותה שיטה בה השתמשנו עבור פקודת beq: איפוס קווי הבקרה ע"י יחידת control. את הפקודה שבשלב IF הפכנו ל-nop וכדי לשטוף את הפקודה שבשלב ID נשתמש במררב שקיים ונוסיף לפניו שער OR אליו ייכנס אות בקרה ID.Flush בנוסף לאות המגיע מה-HDU. בשלב EX נוסף קו בקרה EX.Flush ונוספו שני מרבבים חדשים שבוחרים בין אותות הבקרה הקיימים ובין האות 0. כדי לעבור לקטע קוד של מערכת ההפעלה המטפל בחריגות ל-PC מוכנס דרך המררב כתובת הפקודה הראשונה של מערכת ההפעלה (0x80000180). לבסוף אוגר Cause מתעדכן בסיבה או מס' החריגה ואוגר EPC מקבל את כתובת הפקודה שגרמה לגלישה (למעשה הכתובת שמגיעה מ-PC היא כתובת הפקודה הבאה ולכן הפרוצדורה של החריגה תוריד 4 מהערך הנשמר).

חריגה עקב כתובת לא חוקית בזיכרון פק' lw / sw:

זיהוי הבעיה יהיה בשלב המחזור השלישי באותו אופן כמו במעבד רב מחזורי. תוצאת ה-ALU שלא מתחלקת ב-4. יש להוסיף על זיהוי זה את קווי הבקרה MemRead, MemWrite. במידה ויש זיהוי של חריגה יש להכניס ל-Cause את מס' הפסיקה, לשמור ב-EPC את PC+4 ולדאוג להעביר ל-PC את כתובת ה-Exception Handler. במקביל יש לבצע שטיפה לשלבים: ID, IF, EX. למעשה למעט שינוי המס' ב-Cause שאר הפעולות של הזיהוי הינן כמו בגלישה אריתמטית שגם מזוהה בשלב זה (תרשים 4.66 כמו באריתמטי).

היררכיות זיכרון:

היום קיים פעם בין מהירות המעבד למהירות זיכרון DRAM. נשאלת השאלה כיצד אם כן ניתן לטפל בפער הזה. התשובה היא יצירת זיכרון קטן ומהיר שנקרא זיכרון מטמון (Cache) בין המעבד לבין ה-DRAM.



מהו RAM?

RAM (Random Access Memory) הוא זיכרון שמאפשר להגיע לכל נקודה ב-Data Base באופן אוטומטי ומידי. ה-RAM מחזיק חלקי מידע קריטיים על תוכנות שרצות לטובת העברתם למעבד. הוא חייב לקבל זרם חשמל רציף. אם נפסק הזרם כל המידע נעלם (כלומר זהו זיכרון לטווח קצר).

טכנולוגיות RAM עיקריות:

ככל שהזיכרונות מהירים יותר הם יקרים יותר עבור כל סיבית. כמו כן זיכרונות איטיים בד"כ קטנים בגודל שטח עבור כל סיבית.

ישנן שני סוגי RAM עיקריים כאשר ההבדל בניהן טמון באופן אחסון הביטים:

- (1) Static RAM (SRAM) – שומר מטען ודורש טרנזיסטור בודד לכל סיבית כמו -DRAM. הוא אינו דורש ריענון ולכן קצב העבודה שלו מהיר יותר. החיסרון הוא נפח אחסון קטן. משמש בעיקר כזיכרון מטמון למעבד.
- (2) Dynamic RAM (DRAM) – מבוסס על יכולתו של קבל לשמור מטען. דורש ריענון. הריענון הוא מקו המילה "דינאמי". הוא דורש רק טרנזיסטור אחד לכל סיבית, הוא מכיל נפח אחסון גדול אך הוא מהיר פחות מ-SRAM.

Memory Technology	Typical Access Time	\$ Per Gbyte
SRAM	0.5-1.5 ns	2000\$-5000\$
DRAM	50-70 ns	20\$-70\$
Magnetic Disk	5-20 million ns	0.2\$-2\$

עקרון הלוקאליות:

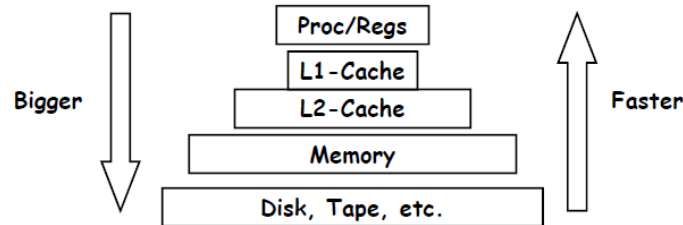
תוכנות ניגשות לחלק קטן ממרחב הכתובות שלהן בכל רגע נתון ועל כן עקרון הלוקאליות מדבר על שני מרחבים:

- (1) לוקאליות בזמן (Temporal Locality) – פריטים שניגשו אליהן לאחרונה הם בעלי סבירות גבוהה שייגשו אליהם שוב בקרוב (לדוגמא פקודות בלולה).
- (2) לוקאליות במרחב (Spatial Locality) – פריטים אשר נמצאים בסמוך לפריטים שניגשו אליהם לאחרונה הם בסבירות גבוהה שייגשו אליהם בקרוב (לדוגמא מערך מידע).

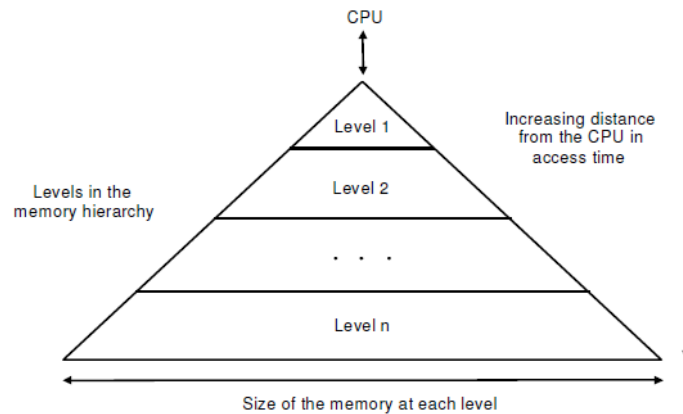
זיכרון מטמון (Cache Memory):

זיכרון מטמון הוא זיכרון מהיר שמטרתו לשפר את מהירות הגישה הממוצעת לזיכרון איטי. זיכרון המטמון מבוסס על עקרון הלוואיות. מבחינת ארכיטקטורת מחשב כמעט כל דבר הוא זיכרון מחשב:

- (1) אוגרים הם מטמון על משתנים
- (2) מטמון רמה ראשונה הוא מטמון על מטמון רמה שנייה
- (3) מטמון רמה שנייה הוא מטמון על זיכרון ראשי
- (4) זיכרון ראשי הוא מטמון על כונן (זיכרון וירטואלי)



היררכיה של הזיכרון (Memory Hierarchy):



היררכיה זיכרון גורסת כי הזיכרון המהיר ביותר, שהינו גם הקטן ביותר, הוא הקרוב ביותר למעבד ונעשה בו שימוש ברוב הזמן בעוד זיכרון איטי וגדול יותר נמצא רחוק מהמעבד ותפקידו לתת תחושת אשליה שיש לנו זיכרון רב וזול.

מונחים מרכזיים בזיכרון מטמון:

Hit – חיפוש מוצלח של מידע בזיכרון המטמון. אם המידע קיים אנו ממשיכים בביצוע התוכנית.

Miss – חיפוש לא מוצלח של מידע בזיכרון המטמון. אם המידע לא מאוחסן שם אנו צריכים לבקש את המידע מזיכרון איטי יותר שנמצא ברמה מעל בהיררכיה. עד אז עלינו לעקב את הצנרת.

Block – זהו יחידת המידע שנטענת לזיכרון המטמון כאשר מתקיים Miss. הגודל המינימאלי של Block הוא מילה בודדת.

Miss Rate – אחוז הפעמים שהתקבל miss ברמת היררכיה מסוימת של זיכרון.

Hit Time – הזמן שנדרש לגשת לרמת זיכרון מסוימת בהיררכיה כולל הזמן הלוך להחליט אם מדובר ב-hit או miss.

Miss Penalty – הזמן הלוך למשוך Block לתוך רמת זיכרון מסוימת בהיררכיה מרמה גבוהה יותר כולל הזמן שלוקח לגשת אל ה-Block, להעביר אותו מרמה לרמה ולהזין אותו לרמה שחוותה את ה-miss.

סוגי Cache Miss:

- (1) Compulsory – נובע מגישה ראשונית ל-Block. נקרא גם Cold Start. זהו עובדה ולא ניתן לטפל ב-miss מסוג זה (לדוגמא בהפעלת מחשב).
- (2) Capacity – זיכרון מטמון אינו יכול להכיל את כל הבלוקים שתוכנית ניגשת אליהם. הפתרון היחידי הוא הגדלת זיכרון המטמון.
- (3) Conflict (Collision) – מקומות שונים בזיכרון ממופים לאותו מקום בזיכרון המטמון. הפתרון הוא זיכרון מטמון אסוציאטיבי.
- (4) Coherence – אי תקפות של המידע או חוסר בהירות. לדוגמא שבמעבד מרובה ליבות כל ליבה רוצה לעדכן את אותו אזור בזיכרון באותו הזמן.

מיפוי ישיר (Direct Mapped Cache):

שדה Tag:

מציין את המידע. אנו משווים בין שדה ה-Tag בכתובת לבין שדה ה-Tag במיפוי.

שדה Index (n):

שדה זה מציין את השורה שבו המידע נשמר. שדה זה הוא בעל 2^n סיביות. 2^n מציין את כמות השורות במיפוי.

שדה Block Offset (m):

שדה זה מציין את כמות המילים בכל שורה. 2^m סיביות מציין את כמות המילים (רוחב המיפוי). במקרה ושדה זה קיים יש צורך ב-mux כדי לקבוע איזו מילה מתייחסים וממנה שולפים את המידע.

שדה Valid:

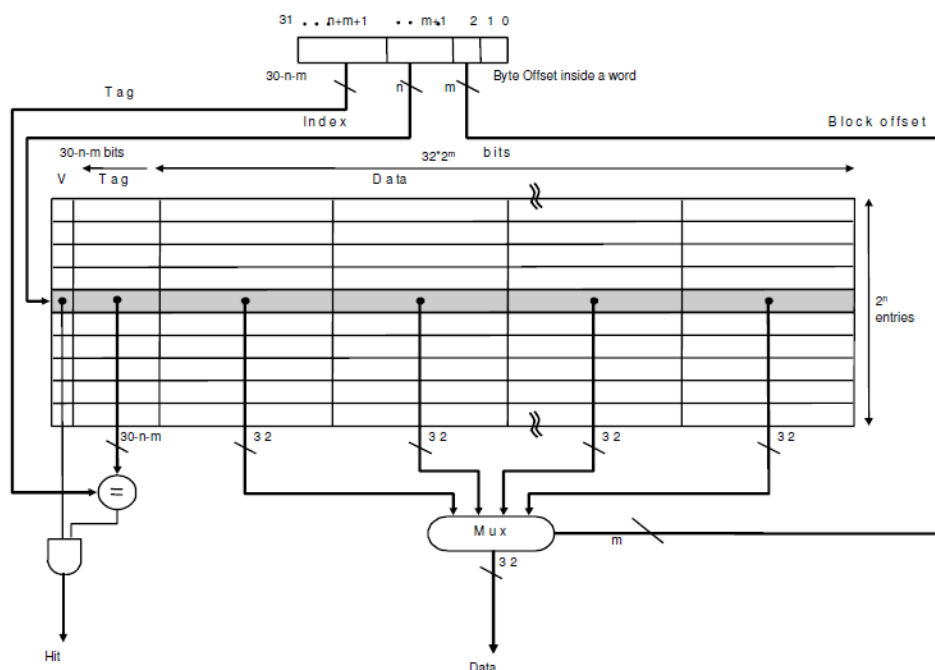
מציין אם המידע קיים (כלומר האם ניגשנו אליו כבר). בהתחלה השדה Valid תמיד יהיה 0 (Miss Compulsory) וכאשר ימופה מידע בפעם הראשונה שדה ה-Valid ישנה ערכו ל-1.

שדה Byte Offset:

תמיד 2 סיביות נמוכות ביותר. ברוב המקרים ערכן יהיה 00. בשדה זה לא נוגעים, הוא קבוע. כאשר נתון לנו כתובת במילים אז נתעלם ממנו ואת הכתובת בבינארי נתחיל מ-Index. במידה ונתון לנו כתובת בבתים מתחילים את הכתובת בבינארי מ-Offset כאשר משנים את שתי הסיביות הנמוכות ל-00.

שדה Data:

זהו השדה המכיל את המידע בפועל.



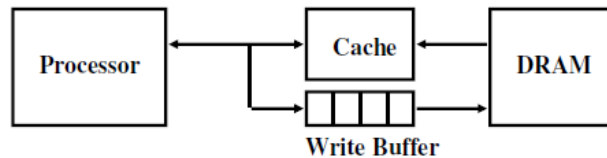
טיפול בכתיבה:

:Write Through

כל דבר שנכתב, נכתב גם לזיכרון המטמון וגם לזיכרון הראשי. כתיבה לזיכרון מטמון נעשית כיוון שאנו עובדים לפי עקרון הלוואיות בזמן. אם הרגע כתבנו, כנראה שבקרוב נצטרך את זה.

שימוש ב-Write Buffer:

מכיוון שכתובה לזיכרון האיטי יותר לוקחת הרבה זמן נשתמש לרוב במתווך בין זיכרון המטמון לזיכרון הראשי הנקרא Buffer. הוא מקבל את נתוני הכתיבה ולאט לאט כותב לזיכרון (הוא מטפטף את המידע לאט, לאט בין הרמות השונות של הזיכרון עד לזיכרון הראשי). אם ה-Buffer מתמלא יש לעצור את המעבד.



אופן העבודה ומבנה:

- (1) המעבד כותב את המידע לזיכרון המטמון ול-Buffer.
- (2) בקר הזיכרון (Memory Controller) אחראי על כתיבת התוכן של ה-Buffer לזיכרון הראשי.
- (3) ה-Buffer עובד בשיטת FIFO (First In First Out). המידע שנכנס ראשון זה המידע שישמר ראשון.
- (4) ה-Buffer לרוב מכיל 4 ערכים.
- (5) אם $store\ frequency > \frac{1}{DRAM\ write\ cycle}$ אז ה-Buffer ברויה (Saturation).

:Write Back

דרך נוספת היא להעתיק זיכרון המטמון לזיכרון הראשי רק כאשר מחליפים בלוק. שיטה זאת היא רלוונטית בעיקר עבור בלוקים גדולים. בשיטה זו כאשר יש כתיבת מידע נעדכן רק את הבלוק בזיכרון ונעקוב האם כל בלוק "מלוכלך" (Dirty). כלומר האם בוצע שינוי. כאשר בלוק "מלוכלך" מוחלף (בעל Dirty Bit) מבוצע כתיבה חזרה לזיכרון. ניתן לשלב שיטה זו עם Write Buffer כדי לאפשר החלפת בלוק שיש לקרוא קודם.

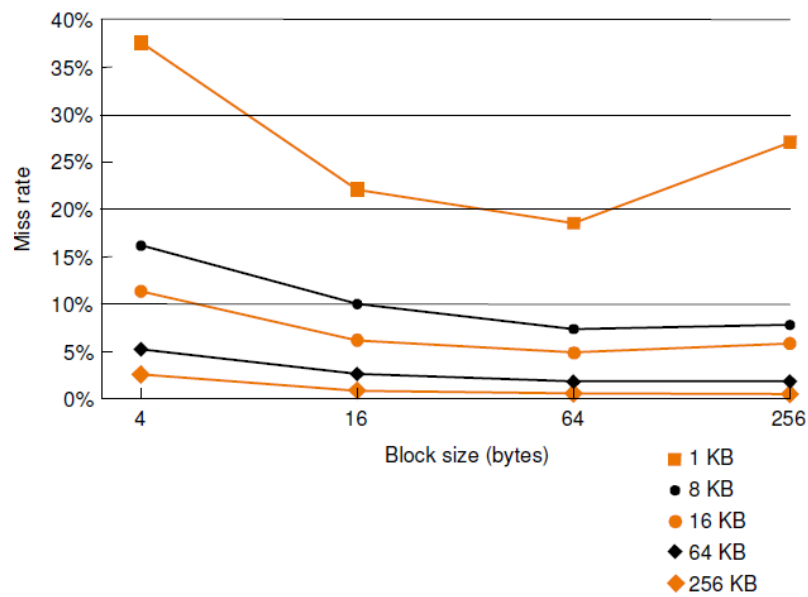
הפרדה של בין זיכרון מטמון לפקודות וזיכרון מטמון למידע:

לרוב קיים זיכרון מטמון אחד עבור פקודות ואחד עבור מידע. שימוש בזיכרון מטמון אחד בעבור הפקודות והמידע יכול לתת לנו גמישות מסוימת כי לעיתים יהיה לנו יותר מקום למידע אך היתרון של שני זיכרונות מטמון שונים הוא שרוחב הפס שלנו גדול פי שתיים. כלומר אנו יכולים לקרוא את הפקודה וגם את המידע באותו זמן (פי 2 יותר מהיר).

נקראים גם:

- (1) D-Cache – מטמון המידע
- (2) I-Cache – מטמון הפקודות

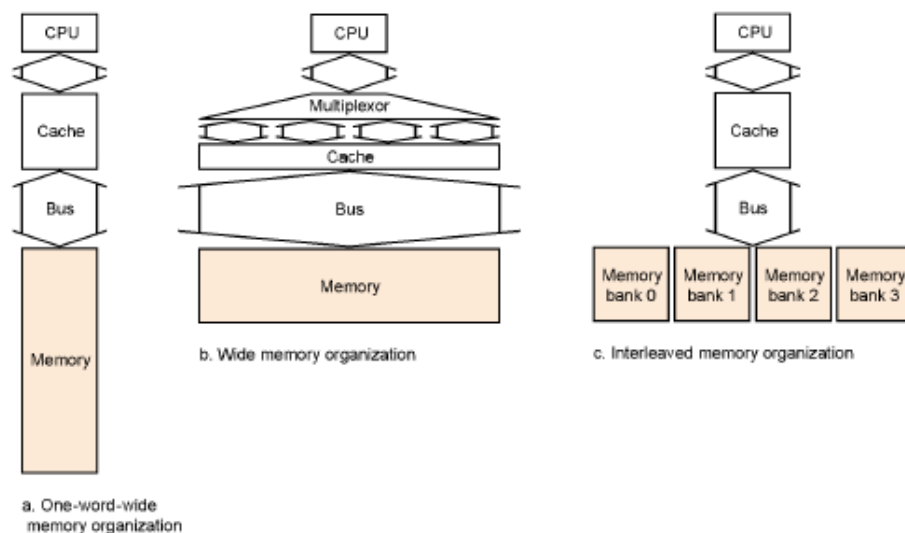
היחס בין גודל Block לביט Miss Rate:



כאשר אנו מגדילים את גודל ה-Block, ה-Miss Rate קטן. במיוחד עבור פקודות (Instructions). למרות זאת במקרה שנשאיר את גודל זיכרון המטמון כמו שהוא נגיע למצב בו יש מעט בלוקים כך שאנו צריכים להחליף אותם כל הזמן וע"י כך אנו מפסידים את היתרון של לוקאליות מה שיוביל להגדלת ה-Miss Rate.

גודל ה-Block וקריאה:

כאשר יש לנו יותר ממילה אחת ממופת בבלוק אנו צריכים לחכות זמן רב יותר לקריאה של כל הבלוק. ישנן טכניקות להתחלת כתיבה לזיכרון מטמון במהירות האפשרית. דרך נוספת היא לתכנן את הזיכרון כך שהקריאה תהיה מהירה יותר, במיוחד קריאה של נתונים עוקבים. קריאה זו נעשית ע"י קריאה של מס' מילים במקביל.



סיכום גדלי זיכרון:

- גודל זיכרון מטמון:
 - במילים: 2^{n+m}
 - בתים: 2^{n+m+2}
 - סיביות: $2^{n+m+2+3}$
- גודל זיכרון ראשי: 2^{32}
- פי כמה גדול זיכרון ראשי ממטמון: $2^{tag} = 2^{30-n-m}$

מציאת מיפוי בלוק:

- ניתן גם ע"י המרה לבינארי ולפי השדות.
- (1) נתונה כתובת (בעשרוני).
 - (2) נתון לנו גודל בלוק בבייטים (m).
 - (3) נחלק את הכתובת במס' הבייטים ונקבל את כתובת הבלוק (שורת הבלוק).
 - (4) נחלק את כתובת הבלוק שהתקבלה בכמות הבלוקים (מס' השורות) הקיימים. השארית היא מס' הבלוק שאליו מופה הכתובת.

$$Block\ index = (Block\ address) \bmod (\#Blocks\ in\ cahce)$$

שיקולים בגודל בלוק:

- (1) בלוקים גדולים יותר מקטינים Miss Rate – לוקאליות במרחב
- (2) זיכרון מטמון קבוע – בלוקים גדולים \Leftarrow פחות בלוקים \Leftarrow יותר תחרות בניהם \Leftarrow Miss Rate גבוהה יותר.
- (3) בלוקים גדולים מוביל גם ל-Miss Penalty גדול יותר. זמן החיפוש ארוך יותר. הדבר יכול לפגוע בתועלת של הקטנת ה-Miss Rate. אתחול מוקדם ומילים קריטיות קודם יכול לעזור.

מיפוי כתובת בזיכרון ראשי:

- ניתן גם ע"י המרה לבינארי ולפי השדות.
- (1) חלוקה בגודל בלוק של כתובת נתונה – השארית היא offset.
 - (2) $AB \bmod 2^n = index$
 - (3) $AB \div 2^n = tag$

יחס דחיסה ונצילות (הערכים הם בבייטים):

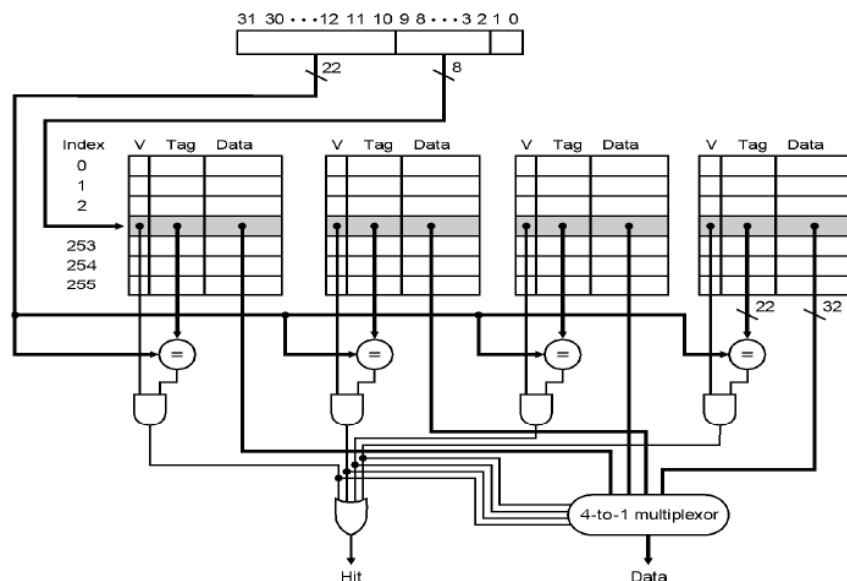
$$יחס\ דחיסה = \frac{valid + tag + data_size}{data_size}$$

$$נצילות\ זה = \frac{1}{יחס\ דחיסה}$$

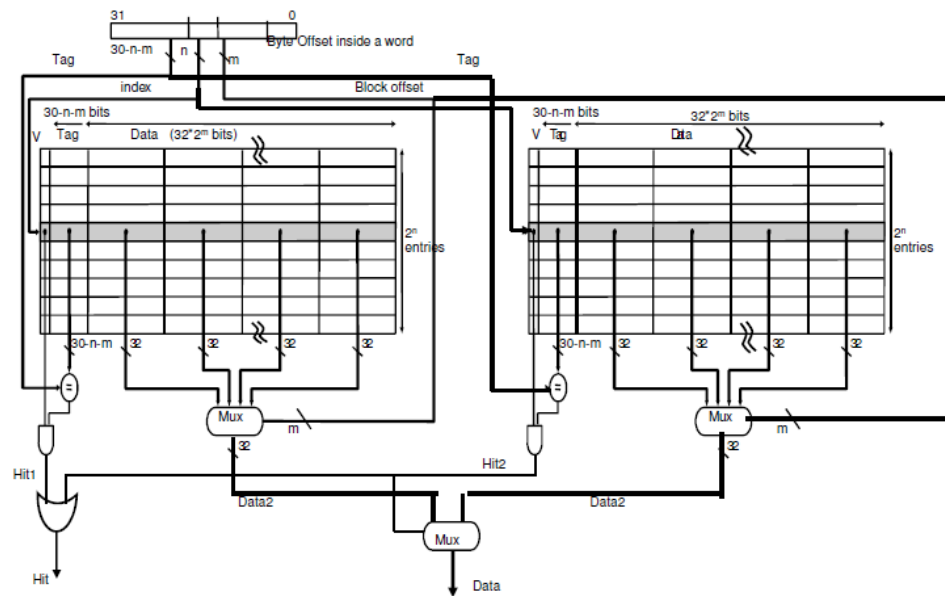
הקטנת ה-Miss Rate ע"י מיפוי אסוציאטיבי:

שיטה זאת נעשית ע"י מתן גמישות לזיכרון המטמון בשמירת המידע. עד עכשיו אפשרנו לבלוק זיכרון להיות ממופה לבלוק בודד בזיכרון המטמון. הגמישות היא האפשרות לתת לכל בלוק בזיכרון להיות ממופה לכל אחד מהבלוקים בזיכרון המטמון. בצורה זו אנו שומרים על בלוק של הזיכרון התכוף ביותר שבמיפוי ישיר היה "מתחרה" על המקום שלו. שיטה זו נקראת זיכרון מטמון אסוציאטיבי (Fully Associative Cache).

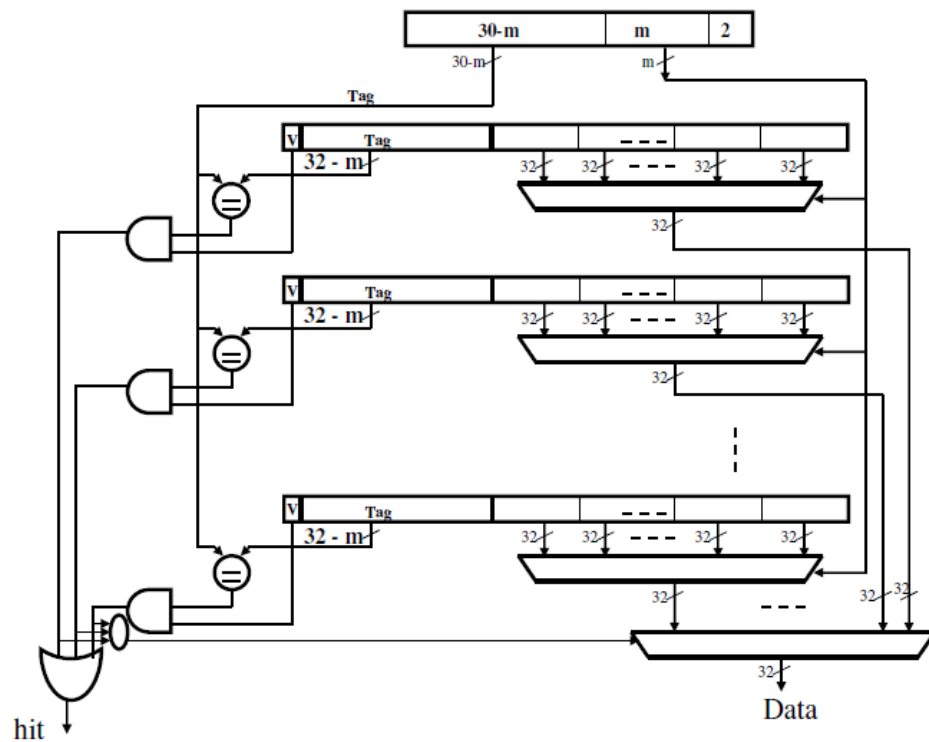
4-Way Associative Cache:



:2-Way Associative Cache



: Fully Associative Cache



הסבר:

במימוש אסוציאטיבי ההשוואות נעשות במקביל. העלאת דרגת האסוציאטיביות מקטינה בד"כ את שיעור ההחטאה אך עלולה להגדיל את זמן הפגיעה עקב הוספה וסיבוכן חומרה.

בזיכרון מטמון אסוציאטיבי יש לנו שתי אפשרויות להחלפת בלוקים:

- 1) רנדומאלי (Random) – החלפת בלוק תעשה באופן שרירותי.
- 2) LRU (Least Recently Used Block) – הבלוק שלא נעשה בו שימוש הכי הרבה זמן הוא זה שיוחלף.

הערות:

- (1) ככל שהזיכרון גדול יותר כך ההבדל בין LRU ל-Random קטן יותר
- (2) יש סכמות החלפה נוספות.

שיקולים ברמות זיכרון מטמון (Multilevel Cache):

- (1) זיכרון מטמון ראשי – מתמקד ב-Hit Time מינימאלי.
- (2) זיכרון רמה 2 – מתמקד ב-Miss Rate נמוך כדי לחסוך בגישה לזיכרון ראשי.

התוצאה של שיקולים אלו:

- (1) L-1 Cache לרוב קטן מרוב זיכרונות המטמון.
- (2) גודל בלוק של L-1 Cache קטן יותר מגודל בלוק של L-2 Cache.

הערכת ביצועי זיכרון:

בהנחה ש-Read, Write Penalties הם זהים ושאינן Buffer Stall.

AMAT=Average Memory Access Time.

$$\text{Miss Penalty} = \frac{\text{Main Memory Access Time}}{CCT} [\text{cycles}]$$

$$\text{Effective CPI} = \text{CPI} + (\text{Miss Rate per Instruction})\% * (\text{Miss Penalty})$$

$$\begin{aligned} \text{Memory Stall} &= \frac{\text{Memomry Accesses}}{\text{Program}} * \text{Miss Rate} * \text{Miss Penalty} \\ &= \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Misses}}{\text{Instructions}} * \text{Miss Penalty} \end{aligned}$$

$$\text{AMAT} = \text{Hit Time} + (\text{Miss Rate}) * (\text{Miss Penalty})$$

ביצועי זיכרון ב-Multilevel:

כאשר נתון מודל זיכרון מטמון בעל רמות שונות (כאן נתייחס ל-2 רמות ועוד זיכרון ראשי) אז:

$$\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} * [\text{Hit Time}_{L2} + (\text{Miss Rate}_{L2} * \text{Hit Rate}_{L3})]$$

סיכום:

- (1) כאשר משפרים את ביצועי המעבד ל-Miss Penalty יש יותר משמעות.
- (2) הקטנת CPI הבסיס מגדיל את הזמן בו נהיה במצב של תקיעה (Stall) בזיכרון.

הרחבה על מעבדים מתקדמים:

- (1) מעבדים מסוימים יכולים לבצע פקודות במהלך Miss:
 - א. הנתון שמחכה לכתיבה נשאר ביחידת ה-Store/Load.
 - ב. פקודות לא עצמאיות מחכות בתחנות המתנה בעוד פקודות עצמאיות ממשיכות.
- (2) השפעת ה-Miss תלויה בזרימת המידע של התוכנית.
 - א. קשה יותר לנתח את ההשפעה.
 - ב. משתמשים במערכות סימולציה.

תכנות:

מבנה תוכנית:

```
#####Data Segment#####
.data
.....
##### Code Segment####
.text
.global main
main:                #main program entry
....
li $v0, 10 # Exit Program
syscall #Execute
```

.Data	מגדיר את חלק המידע של התוכנית. בחלק זה נגדיר את המשנים של התוכנית.
.Text	בחלק זה אנו מגדירים את חלק הקוד של התוכנית.
.Global	מגדיר סמל כגלובאלי. מאפשר להגדיר נקודת ייחוס מקבצים אחרים. אנו תמיד נגיר את פרוצדורת main כ-global.

הגדרת משתנים ומחרוזות (יבוצע מתחת ל-.data):

באמצעותו ניתן להגדיר שם (label) למידע מסוים.

[שם]: [initializer], [מידע התחלתי-initializer] [סוג-directive]: [שם]; ...

.Byte	שומר ערכים בגודל 8 סיביות (בית)
.Half	שומר ערכים כ-16 סיביות (כחצי מילה – 2 בתים)
.Word	שומר ערכים כ-32 סיביות (מילה – 4 בתים)
.ASCII	הגדרת מחרוזת ASCII
.ASCIIZ	הגדרת מחרוזת ASCII כאשר בסוף המחרוזת מושם null (כמו ב-C)
.Space n	הגדרת מקום בזיכרון בגודל n בתים

אותיות חשובות במחרוזות:

- (1) \n – שורה חדשה
- (2) \t – tab
- (3) \" – מרכאות

מבנה פקודות:

פקודות מסוג R:

שדה ה-op תמיד שווה 0 ואילו שדה ה-func אומר מהי הפקודה. שדות ה-rs וה-rt עבור אוגרי המקור ושדה rd עבור אוגר המטרה. שדה ה-shamt (shift amount) מיועד לפקודת הזזת סיביות שמאלה או ימינה.

func \$rd, \$rs, \$rt

Opcode = 0 6 bit	Rs 5 bit	Rt 5 bit	Rd 5 bit	Shift n (shift amount) 5 bit	Func 6 bit
31-26	25-21	20-16	15-11	10-6	5-0

דוגמאות:

Operation	Syntax	The Action	# Function
add	add \$1,\$2,\$3	\$1=\$2+\$3	32
sub	sub \$1,\$2,\$3	\$1=\$2-\$3	34
and	and \$1,\$2,\$3	\$1=\$2&\$3	36
or	or \$1,\$2,\$3	\$1=\$2 \$3	37
nor	nor \$1,\$2,\$3	\$1=~\$2 \$3	39
slt	slt \$1,\$2,\$3	If (\$s2<\$s3) s\$1=1 else s\$1=0	42

פקודות מסוג I-type:

- (1) בפקודות load ופקודות עם שדה Immediate שדה ה-rs הינו עבור האוגר הבסיס ושדה ה-rt עבור אוגר המטרה.
- (2) בפקודות store שדה ה-rs עבור אוגר הבסיס ושדה ה-rt מכיל את הערך לאחסון בזיכרון.
- (3) בפקודות branch שדות rs ו-rt הן עבור אוגרי המקור והקבוע מקודד את כתובת הקפיצה ביחס ל-pc.

פקודות lw:

lw \$rt,add(\$rs)

Opcode = 35 6 bit	Rs 5 bit	Rt 5 bit	address
31-26	25-21	20-16	15-0

פקודות sw:

sw \$rt,add(\$rs)

Opcode = 43 6 bit	Rs 5 bit	Rt 5 bit	address
31-26	25-21	20-16	15-0

פקודות branch:

beq \$rs,\$rt,address

Opcode = 4 6 bit	Rs 5 bit	Rt 5 bit	Address / Immediate 16 bit
31-26	25-21	20-16	15-0

פקודות מסוג j:

פקודת jump היא קפיצה אבסולוטית ללא תנאים. היא מבצעת קפיצה לכתובת התווית.

j label

Opcode = 2 6 bit	Address 26 bit
31-26	25-0

:Syscall

זוהי אפשרות קריאה למערכת הפעלה (בעיקר לפעולות קלט / פלט). עבודה עם Syscall:

- (1) מעלים לאוגר \$v0 את ערך ה-syscall הרצוי: li \$v0, n
- (2) מעלים לאוגרים \$a0, \$a1 ערכים בהתאם ל-syscall
- (3) הפעלת פקודת syscall ע"י כתיבת פקודה זו.

Service	\$v0	Arguments / Result
Print Integer	1	\$a0 = integer value to print
Print Float	2	\$f12 = float value to print
Print Double	3	\$f12 = double value to print
Print String	4	\$a0 = address of null-terminated string
Read Integer	5	\$v0 = integer read
Read Float	6	\$f0 = float read
Read Double	7	\$f0 = double read
Read String	8	\$a0 = address of input buffer \$a1 = maximum number of characters to read
Exit Program	10	
Print Char	11	\$a0 = character to print
Read Char	12	\$a0 = character read

Supported by MARS

טריקים של ביטים:

XORI עם 1 זה not.

srl מחלק בחזקות של 2

sll מכפיל בחזקות של 2

כדי לדעת מס' זוגיים נספור את סיביות האפס של המס' הזוגיים andi עם 1.

ההפרש בין מס' שלם לייצוג התווי שלו הוא $-0x30$