Fetch -> Decode -> Execute -> Memory -> Writeback

כאשר מדברים על מעבד חדמחז מי שקובע את אורך מחזור השעון זו הפקודה שזמן הביצוע שלה הוא **הארוך ביותר.**

- הפקודה שלוקחת הכי פחות זמן - פקודת j
- הפקודה שלוקחת הכי הרבה זמן - פקודות load

פקודות ה load עוברות על כל 5 השלבים.

על מנת להבין את רעיון ההצנרה נקביל את הדבר למושג מהחיים האמיתיים - כביסה:

הסטודנט צבר ערימות כביסה, ויום אחד החליט ללכת למכבסה ולכבס את הבגדים.

הוא מגיע אל המכבסה ומבצע את הפעולות הבאות, כאשר כל אחת מהפעולות משתמשת במשאב שונה על מנת להתבצע:

פעולה	משאב
כיבוס	מכונת כביסה
ייבוש	מייבש
קיפול	שולחן קיפול כביסה
אחסון	ארון

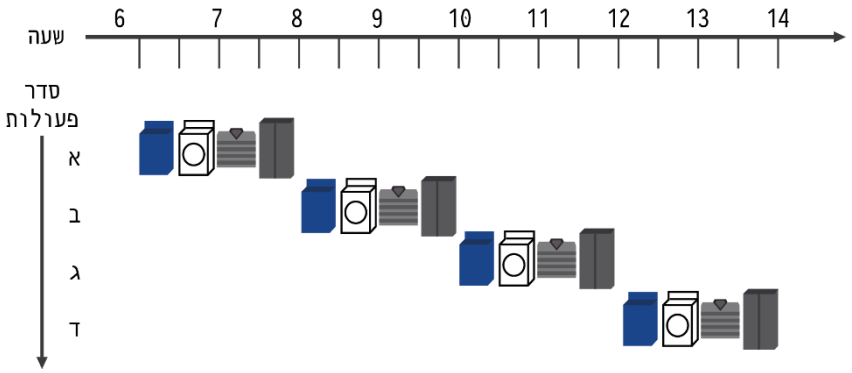
עכשיו אם נסתכל על העניין בפרספקטיבה של יעילות - נבין כי אם הסטודנט החמוד שלנו לא ייבצע את הפעולות בצורה מקבילה הוא **יבזבז זמן יקר!**

אם הסטודנט יחכה שהערימת כביסה הראשונה תהיה מאורגנת ומסודרת בארון ורק אז יתחיל עם הערימה השנייה, דבר זה לא יהיה יעיל!

נניח כי ישנן **4 ערימות בגדים** והזמן הממוצע לביצוע ארבעת הפעולות:

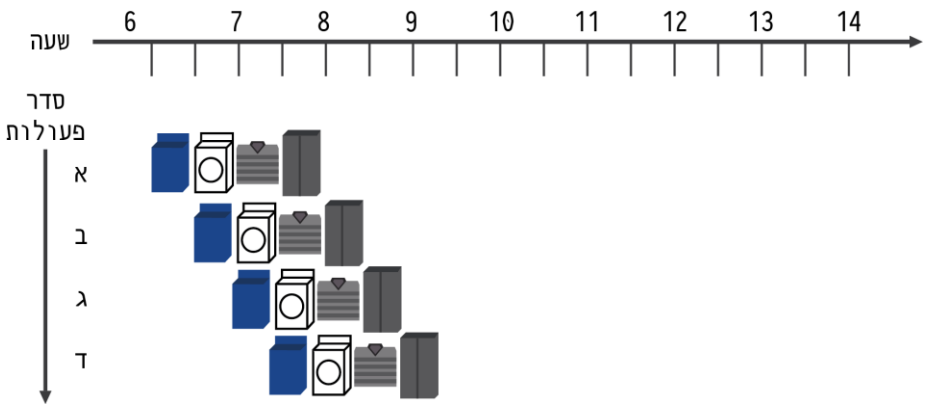
כיבוס ← ייבוש ← קיפול ← אחסון הוא שעתיים שלמות, במידה והוא יעשה הכל בצורה חד מחזורית (כל ערימה לבד) ייקח לו 8 שעות לכבס את כל ערימות הבגדים!

אילוסטרציה:



לעומת זאת, אם יעבוד בצורה מקבילה (הצנרה) הוא יוכל לצמצם את זמן הכביסה ל 3.5 שעות!

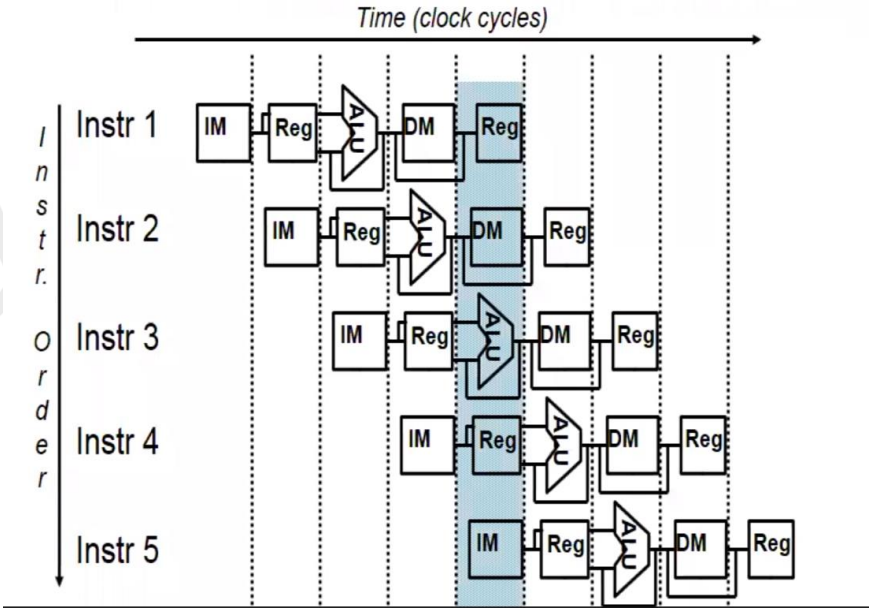
אילוסטרציה:



הסבר - בזמן שהסטודנט ייבש את הערימה הראשונה הוא יכניס את הערימה השנייה לכביסה וכן הלאה.

כל זאת בהנחה שכל ערימה לוקחת זמן שווה של טיפול!

עכשיו נביט במימוש ה PIPELINE על אותם צירים לפי רכיבים



נשים ❤ שרק החל מהמחזור החמישי הצנרת מלאה!

ובארבעת המחזורים האחרונים הצנרת מתרוקנת!

תזמונים

- **Example**:
  - Assume following times for MIPS pipeline stages
    - 100ps for ID and WB stages (decode/read regs, write-back reg)
    - 200ps for other stages

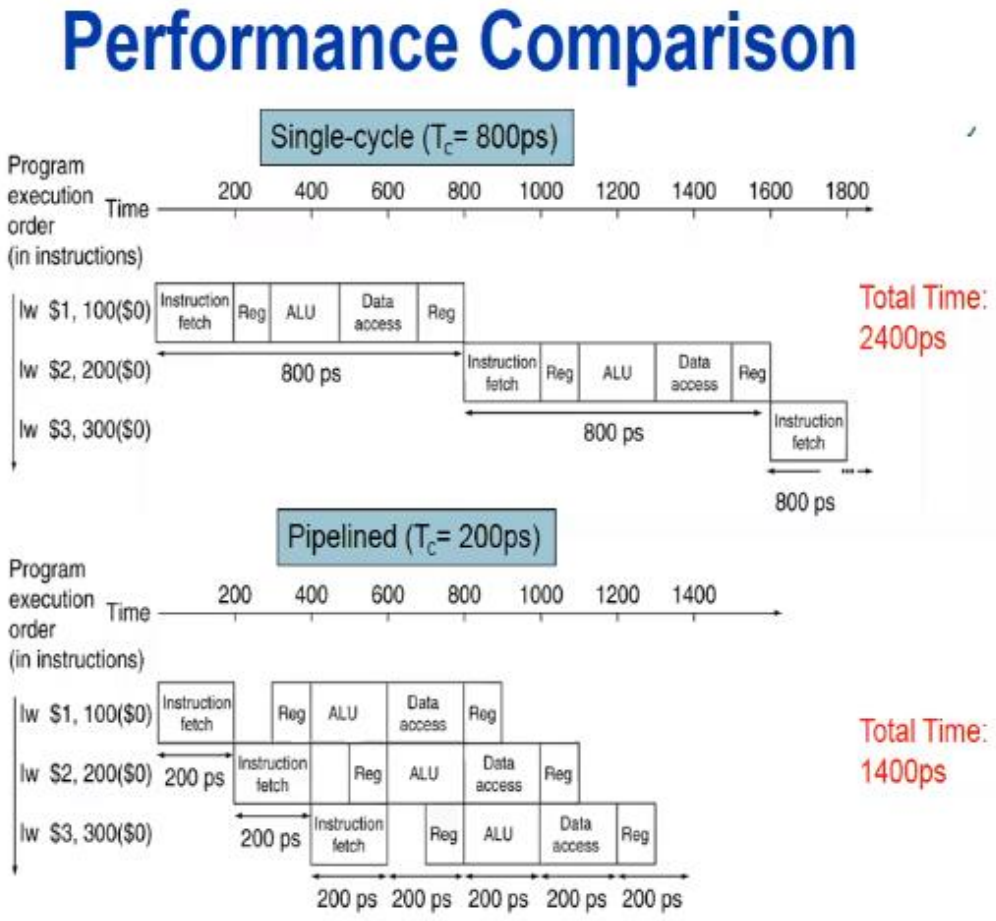
Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

- Single-cycle clock-period: 800ps (longest instruction)
- Pipeline clock-period: 200ps (longest stage)

במעבד החד מחזורי הדבר שקבע את זמן מחזור השעון שלנו הוא בעצם חיבור כל הזמנים של כל שלב בביצוע הפקודה. (באיור שלנו Single-cycle clock-period).

במעבד ההצנרה (Pipeline) הדבר שקובע את זמן מחזור השעון הוא בעצם הפקודה שלוקחת הכי הרבה זמן [ps] (באיור שלנו Pipeline clock-period)

איור נוסף להמחשה:

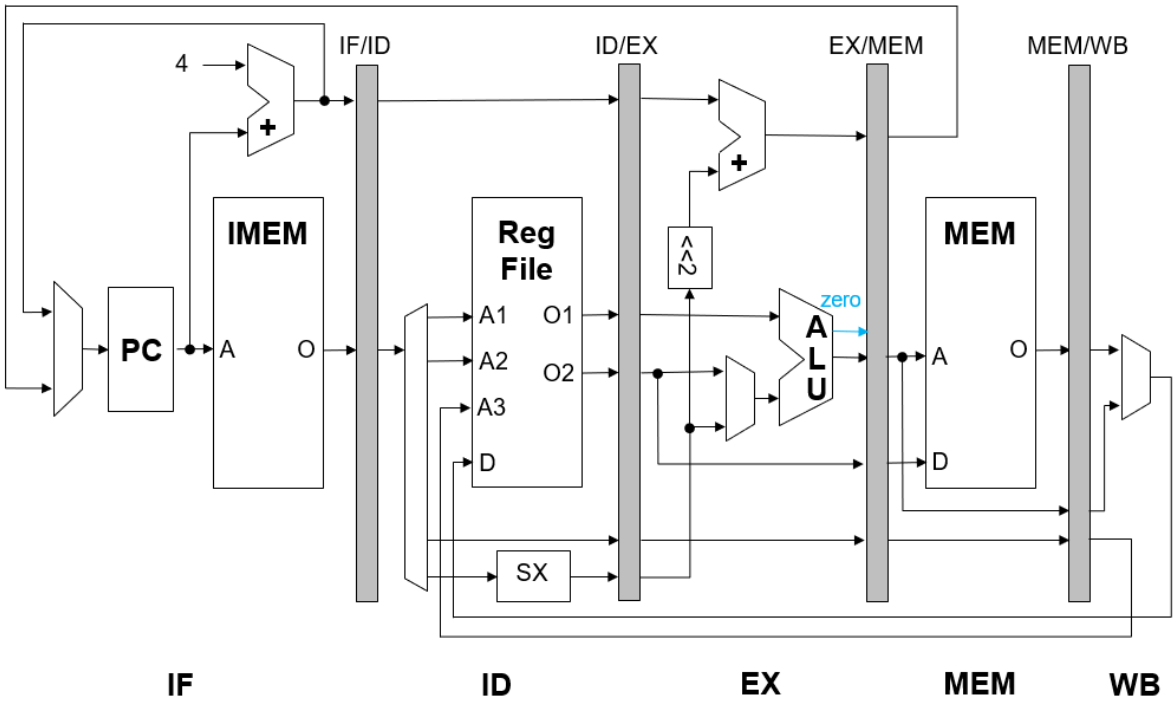


בואו נדבר דוגרי!

מה שמתבצע באמת במעבד בשיטת הצנרה הוא לא פשוט.

על מנת שנוכל לבצע כמה פעולות במקביל בלי לאבד נתונים אנחנו צריכים בעצם לאוסיף עוד אוגרים שיאגרו נתונים בין שלב לשלב.

האוגרים הללו הם החלקים האפורים בציור:



נשים לב שלכל אוגר ניתן שם: IF/ID ID/EX EX/MEM MEM/WB

שמות אלו מסמלים את המצב שממנו שומרים נתונים (מצד שמאל של הלוכסן) והמצב שאליו צריכים להעביר נתונים (מצד ימין של הלוכסן).

אז ראינו שנוצרו מקומות חדשים להטמיע בהם נתונים, YAY!

הלאה!

איך באמת פקודה מתבצעת עם האוגרים החדשים?

הצללה בצד ימין = קריאה

הצללה בצד שמאל = כתיבה

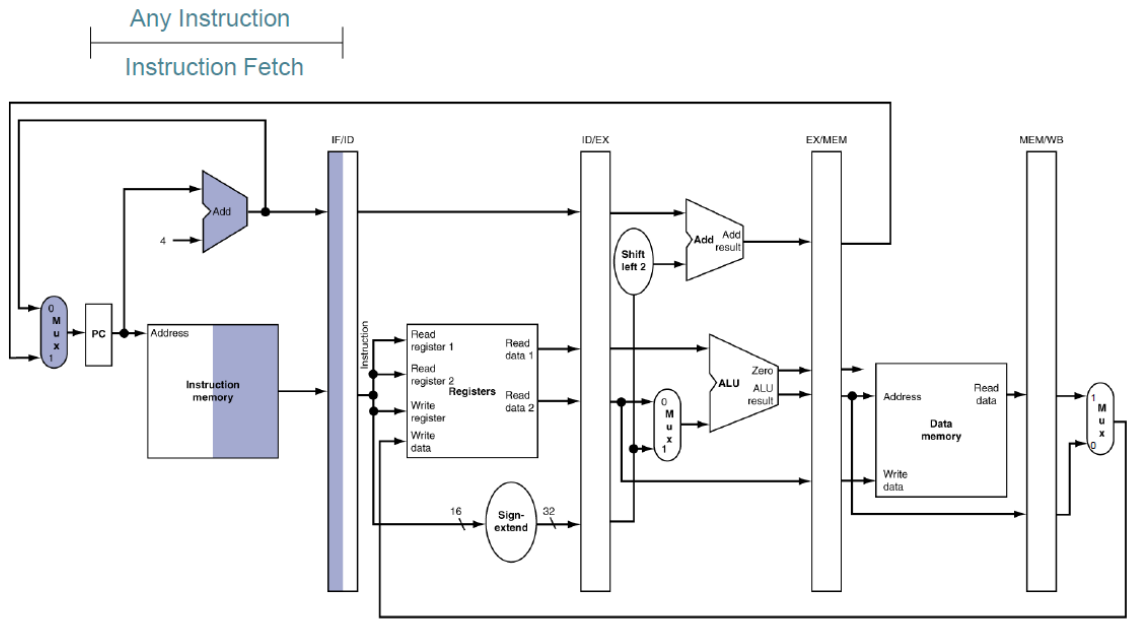
שלב ה Instruction fetch

תחילה מתבצעת פעולת ה instruction fetch.

נכנסת הכתובת מ pc אל ה instruction memory שמבצעת קריאה ממקבץ הפקודות.

במקביל ה pc גדל ב 4 בעזרת ה add .

כמו כן מתבצעת כתיבה לאוגר IF/ID.

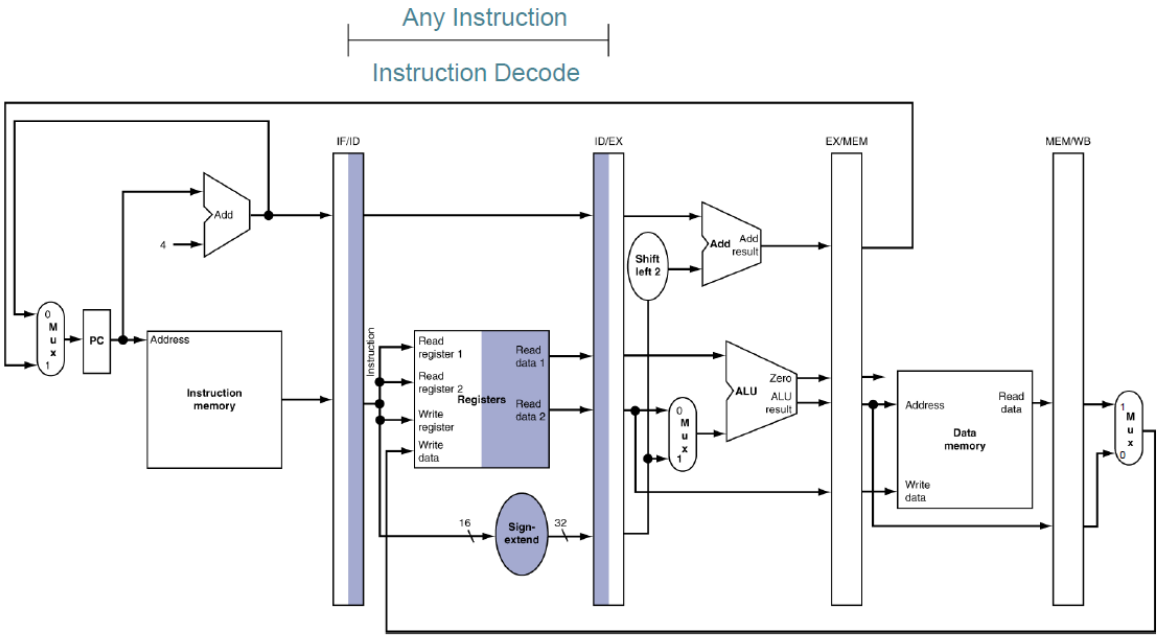


שלב ה Instruction decode

בשלב זה **קוראים** מהאוגר IF/ID את ערכי הפקודה שנכתבה ומוכנסים למקומות המיועדים.

16 הביטים האחרונים של הפקודה נכנסים על ה sign extend ואל שדות האוגרים ושאר המידע מועבר אל מקבץ האוגרים / יחידת ה control .

כמו כן מתבצעת **קריאה** ממקבץ האוגרים (בשביל לדעת אילו אוגרים המשתתפים בפקודה) **וכתיבה** אל האוגר הבא ID/EX את ערכי rs , rt וכמובן את ערך ה immediate אחרי sign extension ל 32 ביטים.



שלב ה Execute

בשלב ה execute יש הפרדה בין סוגי הפקודות.

פקודות sw,lw,R-type מפעילות את ה ALU מכיוון שהוא צריך לעבוד עבורן!

פקודות lw,sw צריכות לחשב כתובת בזכרון בעזרת כתובת היסט (offset) ולכן הן מפעילות אותו.

פקודות R-type צריכות לעשות חישובים (כמו פקודת add) ולכן הן גם מפעילות את ה ALU.

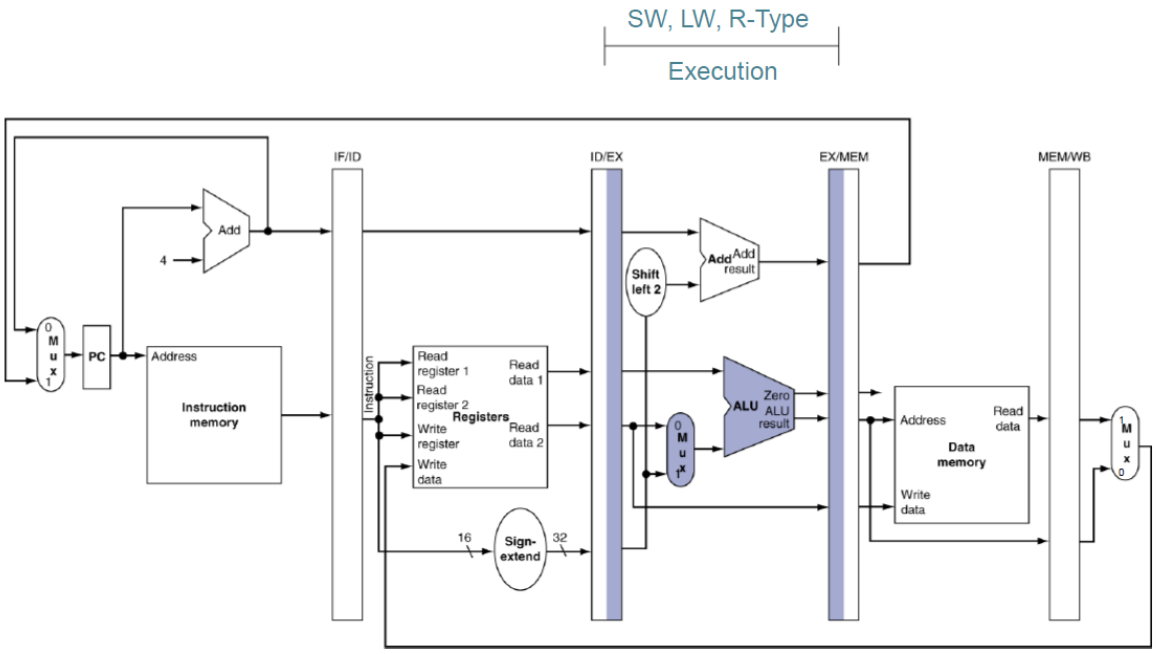
לכן בשלב זה מתבצעת **קריאה** מהאוגר ID/EX .

המרבב אחראי להבחין בין הכניסות ל ALU - הוא בורר האם צריך להכנס הערך מ ID/EX של אוגר rt או שמא אמור להכנס ה immediate.

בין כה וכה הוא מבצע את העבודה שלו (סמכו עלי).

כמו כן מתבצעת **כתיבה** אל אוגר ה EX/MEM של הערך שיוצא מה- ALU (כולל סיגנל ה zero וה- overflow) וכן הערך של rt ובנוסף התוצאת adder של ה branch .

מתבצע העברה של הערך של rt מכיוון שבמידה והפקודה המתבצעת היא פקודת store אז נרצה לשמור את ערך האוגר כדי לרשום לתוך המקום הנכון בזכרון!



שלב ה Memory

בשלב ה Memory מתבצעות הפקודות של שמירה וקריאה מהזכרון.

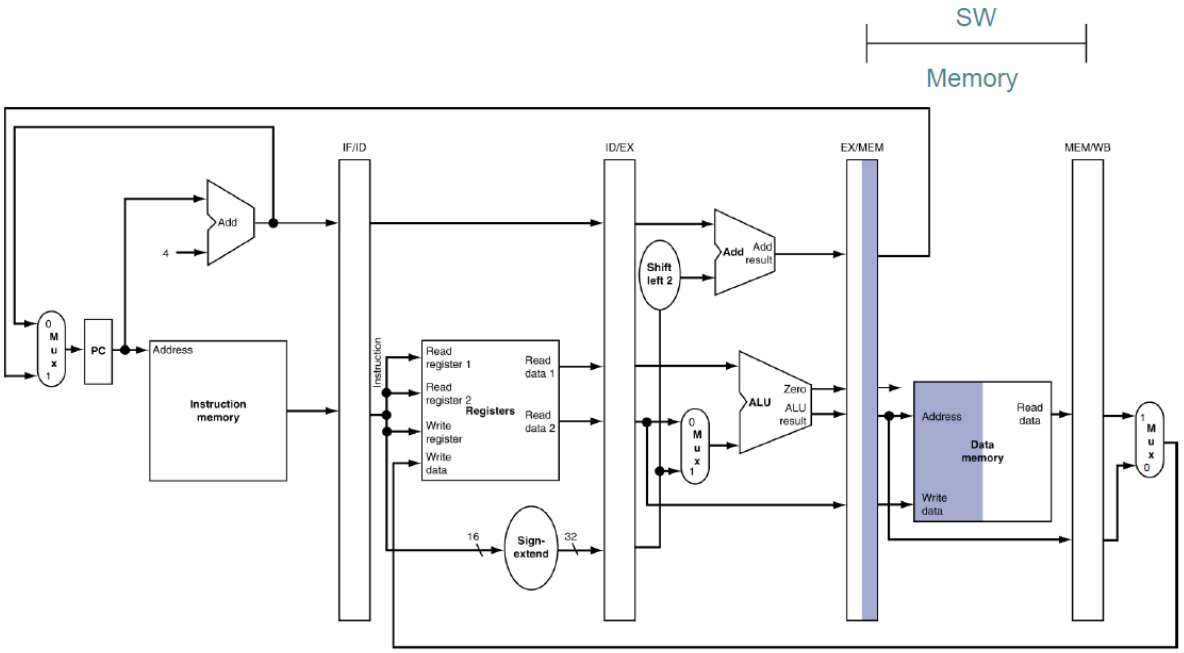
לכן מתבצעת **קריאה** של הנתונים מהאוגר EX/MEM על מנת לזהות את האוגרים שצריך לקרוא/לכתוב אליהם.

כמו כן בשלב זה מתבצעת פקודת ה store .

בזמן פקודת store

בפקודה זו מתבצעת **קריאה** של הכתובת שחושבה על ידי ה ALU אל ה Address וכמו כן מתבצעת **קריאה** של הערך של אוגר rt שנשמר בשלב הקודם אל ה write data .

לא מתבצעת כתיבה לרגיסטר הבא

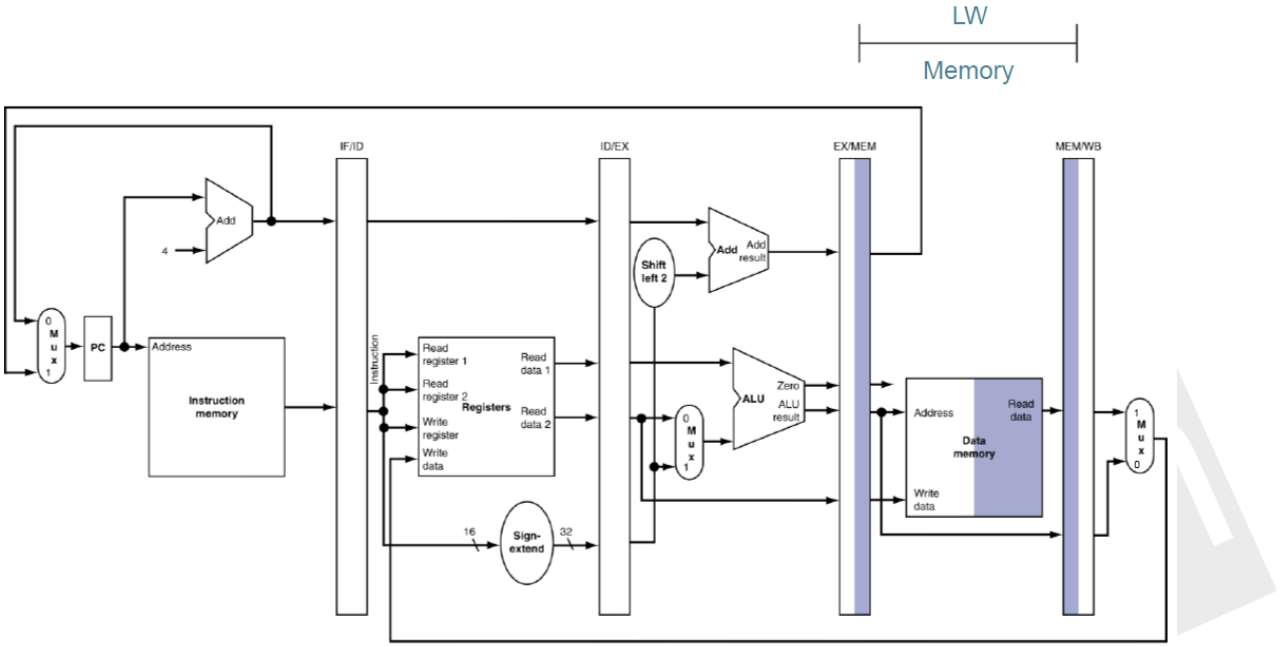


בזמן פקודת load

גם כאן **נקרא** ערך מתוך הרגיסטר EX/MEM אך כאן **נקרא רק הערך של Adress !**

כמו כן בגלל שהפקודה לא שומרת ערך לזכרון אלא לאוגר, היא **קוראת** את הערך מהכתובת שניתנה לה **וכותבת** את הערך לאוגר MEM/WB .

מתבצעת כתיבה לרגיסטר הבא



ישנה בעיה בביצוע ה Write back

כאן מפרידים בין פקודות load לבין פקודות בפורמט R .

שני הפקודות הללו כותבות לרגיסטרים.

בשלב הקודם נכתב אל הרגיסטר גם הערך שנכנס אל ה Address מכיוון שבפקודות מפורמט R מגיע משם חישוב שנחוץ כדי להשלים את הפקודה.

לכן מהרגיסטר MEM/WB מועבר המידע אל המרבב שבורר בין כניסת המידע בעזרת קו MemToReg .

מהמרבב הקו נכנס ישירות אל מקבץ הרגיסטרים.

ישנן 2 אפשרויות כתיבה:

1. אם הפקודה מפורמט I אז הכתיבה מתבצעת אל רגיסטר rt.
2. אם הפקודה מפורמט R אז הכתיבה מתבצעת אל רגיסטר rd.

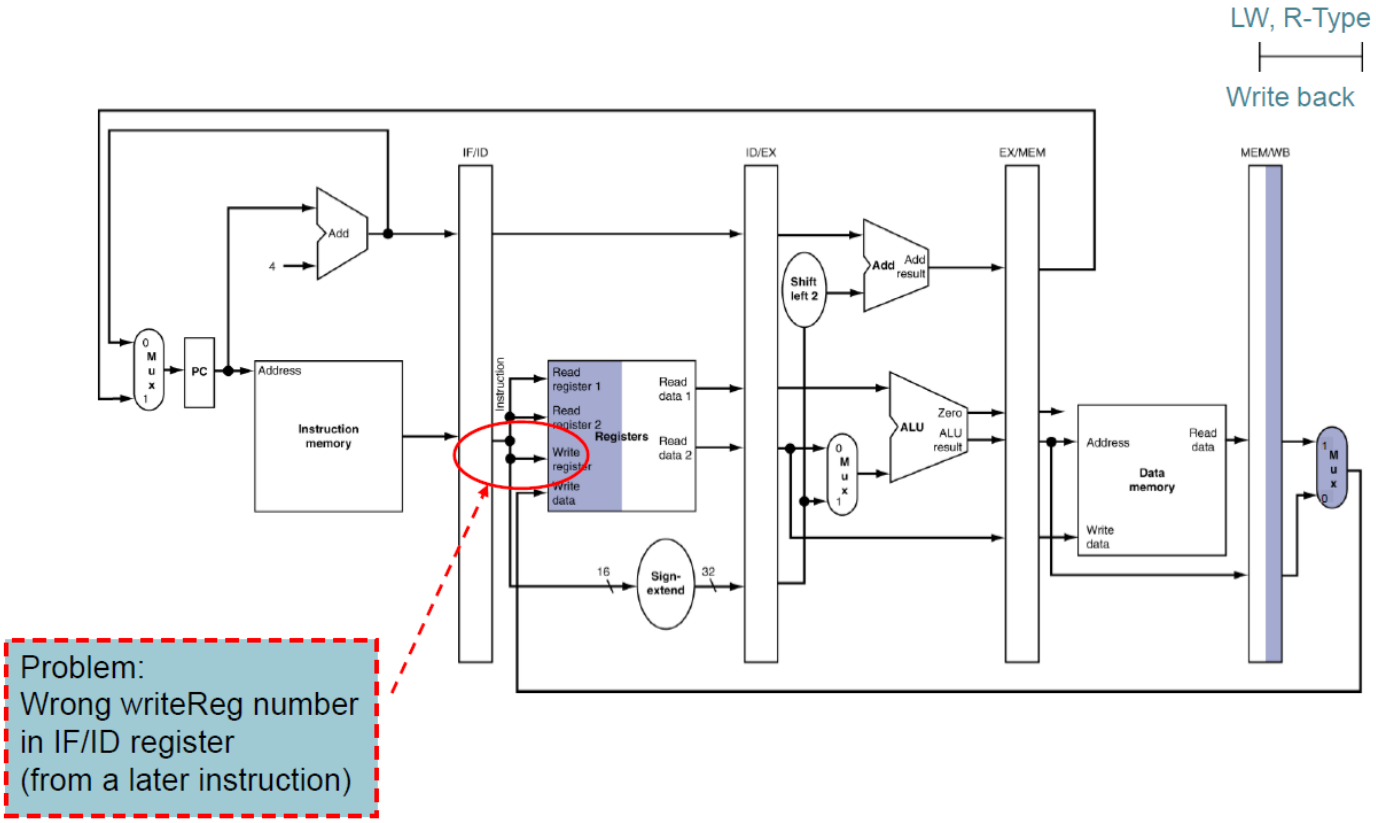
אבל רגע... מה הוא rt ומה הוא rd ?!

בזמן הכתיבה (שלב חמישי) של פקודה נוכחית , ישנו סיכוי שמתבצעת פקודה אחרת בשלב השני שלה!

ויש מצב שיש rt ו rd משלה!

אלו לא הרגיסטרים שאני רוצה לכתוב לתוכם !!

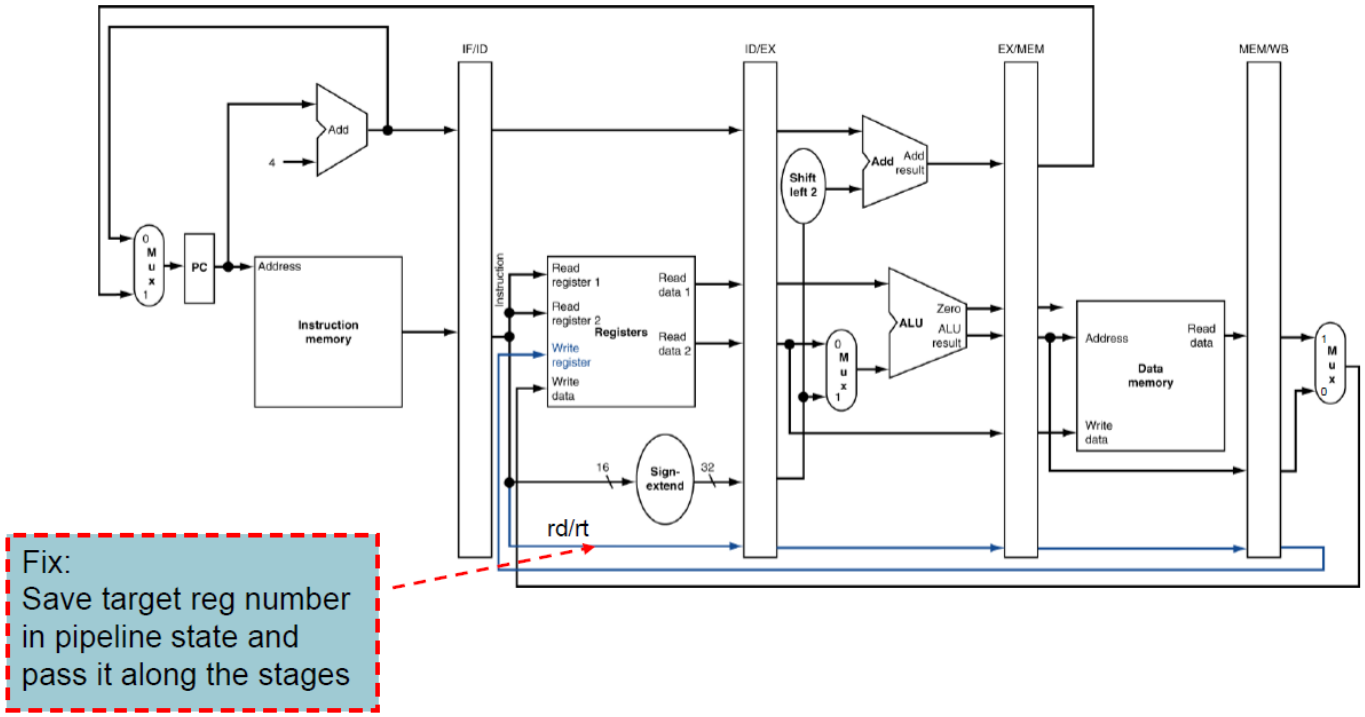
אז צריך להכניס את הערך לתוך הרגיסטר הרצוי ולא לרגיסטר של פקודה אחרת!



אז נוצרה לנו בעיה.. איך נפתור אותה?

מאוד פשוט!

נסתכל באיור הבא:



כאשר פקודה נמצאת בשלב השני שלה אנחנו לוקחים את rd/rt ורושמים אותם באוגר ההצנרה ID/EX ואז מחלחלים את המידע בין האוגרים הנוספים ובשלב ה WB הם יחזרו אל ה Write register וכך הערך ייכתב לאוגר הנכון!

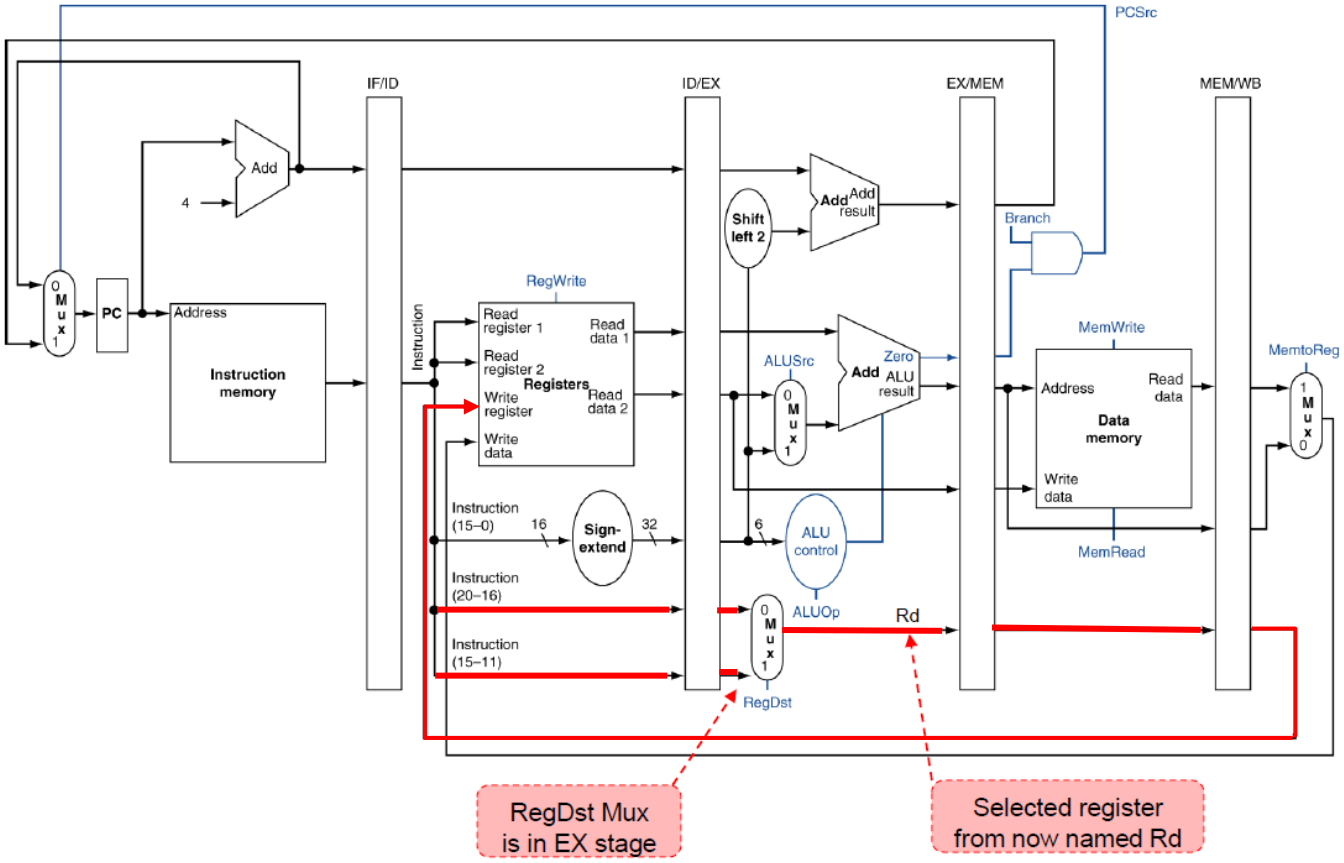
בעמוד הבא נראה תרשים יותר מפורט שמסביר את הפתרון לבעיה!

אז כאן רואים כי מתוך הפקודה נקראים גם ביטים 11-15 (rd) וגם ביטים 16-20 (rt)

אחרי הקריאה של הרגיסטרים הם נכתבים לתוך הרגיסטר ID/EX (pipeline register) שחוצץ בין השלבים.

בשלב השלישי הם נכנסים למרבב שמופעל על ידי אות RegDst (אותו מרבב שברר Single cycle לאיזה רגיסטר כותבים)

אותו מרבב מעביר הלאה ערך אחד בודד בתור Rd (עם R גדולה) שמייצג Register destination לתוך הרגיסטר EX/MEM ומשם לרגיסטר MEM/WB ובחזרה אל הכניסה Write register.



סיגנלים

מכיוון שבגרסת ה Single cycle לא היה לנו אכפת מתי נכתבים הסיגנלים (היה קורה בשלב השני) לא ייחסנו חשיבות לתזמון הסיגנלים.

מכיוון שכעת אנו עובדים בגרסאת Pipeline ישנה חשיבות לתזמון הסיגנלים שנשלחים מה control מכיוון שהסיגנלים צריכים להיות מתואמים לשלבי הביצוע של המעבד!

לכן נחלק את התזמונים לפי האיור הבא:

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	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

החלוקה מתבצעת לשלבים 3,4 ו- 5 (שלבי Execute, memory, write-back).

מדוע?

<u>בשלב 3 - Execute</u>

מתבצעות פעולות ב ALU ועם אילו ערכים!

כמו כן מוחלט לאיזה אוגר ייכתב ערך (בעזרת המרבב שדובר עליו בעמוד הקודם)

<u>בשלב 4 - Memory</u>

בשלב הנתון שבו אנחנו נמצאים בבניית מעבד ההצנרה נחליט כי חישוב ההסתעפות מתבצע בשלב ה Memory (דבר זה ישתנה בהמשך) כמו כן בשלב זה נצטרך את המידע האם לכתוב/לקרוא מהזיכרון ולכן הסיגנלים הרצויים נכנסים פה והם: Branch, Mem-Read, Mem-Write

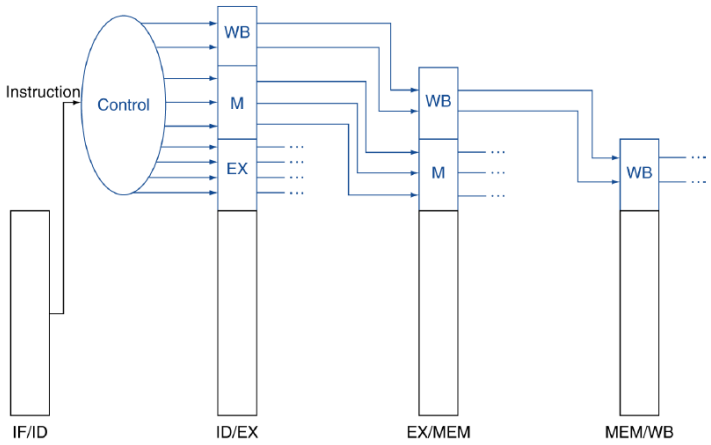
<u>בשלב 5 - Write-Back</u>

בשלב זה צריכה ליפול החלטה האם לכתוב בכלל לרגיסטר כלשהו? ואם כן, מהיכן עלי לקחת את המידע שלי? על החלטות אלו אחראיים הסיגנלים: Reg-Write, MemToReg

אז איך בתכלס מועברים הסיגנלים בין השלבים?

Pipelined Control

נתבונן באיור:



עתה נבחין במתרחש.

הערכים הרצויים מגיעים הישר מה Pipeline register הנחמד IF/ID אחרי שהפקודה נכתבה אליו ממקבץ הפקודות.

מהרגיסטר מגיעים הערכים הרצויים אל יחידת ה Control .

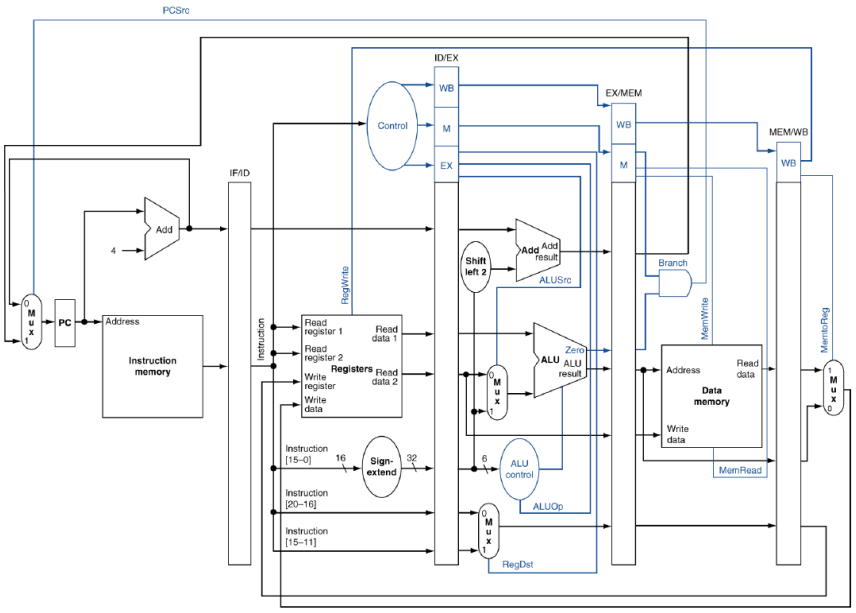
<u>אותו Control מייצר 9 סיגנלים.</u>

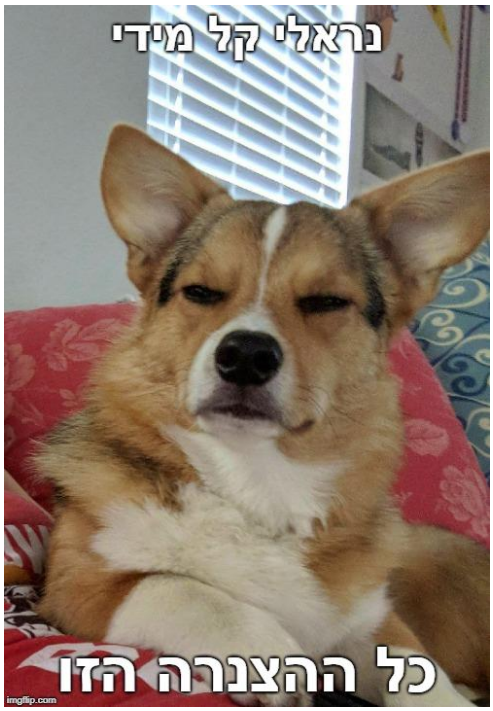
<u>4 סיגנלים</u> מועברים אל תוך רגיסטר ID/EX לתוך איזור בשם EX שנמצא ברגיסטר.
סיגנלים אלו יעברו בשלב Execute אל הרכיבים הרצויים.

<u>3 סיגנלים נוספים</u> מועברים גם הם אל תוך רגיסטר ID/EX לתוך איזור בשם Mem ובהמשך מועברים לרגיסטר EX/MEM לאיזור בשם Mem ובשלב Memory הם יעבור אל הרכיבים הרצויים.

<u>2 סיגנלים אחרונים</u> מועברים גם הם אל תוך רגיסטר ID/EX לתוך איזור בשם WB ובהמשך מועברים לרגיסטר EX/MEM לאיזור בשם WB ומשם ממשיכים לרגיסטר MEM/WB לתוך איזור באותו שם – WB כך שלבסוף הסיגנלים מגיעים לשלב ה Write-back ומועברים לרכיבים הרצויים.

אז מה יש לנו עד עכשיו?





WELL SAID SMART CORGI!

מה בעצם קורה אם נרצה לגשת לערך שערוד לא חושב? מה קורה אם ישנן התנגשויות?

מוכנים לעוד?

יופי!

מה איתך קורגי?

מצוין!

Pipeline Hazards

אז מהם בעצם סיכונים? (Hazards)

הסיכונים שלנו הם בעצם מצבים שמונעים מהמעבד להתחיל את הפקודה הבאה במחזור השעון הבא.

סוגי סיכונים

סיכוני מבנה - מצב בו 2 פקודות בשלבים שונים מנסות לגשת לאותו משאב (זיכרון או רגיסטרים) לא נעמיק בסוג הסיכון הזה מכיוון שהוא נפתר כבר על ידי הוספת חומרה.

סיכוני מידע (Data Hazards) - מצב בו מנסים לגשת לערך של רגיסטר למרות שהערך שלו עדיין לא התייצב (עדיין לא הערך שמצפים לקבל).
סיכון זה מתבטא בפקודות מפורמט R ופקודת lw.

נסתכל על הדוגמא הבאה:

```
add  $s0,$t0,$t1 //$s0 written in cycle 5
sub  $t2,$s0,$t3 //$s0 is read in cycle 3
```

הפקודה add כותבת לתוך הרגיסטר $s0 .

הפקודה sub שתתבצע מיד אחריה קוראת מתוך הרגיסטר $s0.

נניח כי התוכנית מתחילה בשתי שורות אלו

הפקודה הראשונה **מחשבת** את הערך **בשלב השלישי** אך **כותבת** את הערך **בשלב החמישי**(מחזור שעון חמישי).

הפקודה אחריה תקרא את הערך של הרגיסטר **בשלב השני** (כאשר באותו מחזור שעון רק מתבצע חישוב הערך של הפקודה הקודמת).

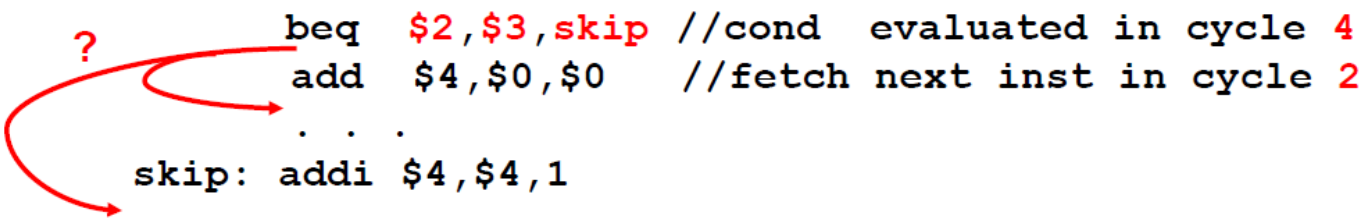
מה יקרה?

הפקודה השנייה תקרא את הערך הישן של רגיסטר $s0 !!!

הערך הנכון ייכנס לרגיסטר $s0 רק במחזור שעון החמישי!

סיכוני בקרה (Control Hazards) - מצב בו ממשיכים בביצוע הפקודות הבאות בזמן שההחלטה האם להסתעף בפקודת Branch עדיין לא התקבלה ולפני שהכתובת שאליה מסתעפים בכלל חושבה.
לכן קיימות עד 3 פקודות בתוך הצנרת שאנחנו אולי לא צריכים!

בשביל להסביר טוב יותר נתבונן בדוגמא הבאה:



בדוגמא רואים כי אחרי ההסתעפות ישנה פקודת add שבה מכניסים ערך לתוך הרגיסטר $4 ואחרי ההסתעפות בתווית skip מתבצעת פעולה עם הרגיסטר $4 , אז ישנו סיכוי שהערך שאנחנו נשתמש בו לא נכון! ישנה פקודת add שלא רצינו שתתבצע אך בוצעה בכ"מ לפני ההסתעפות!

Structure Hazards

סיכוני מבנה מסתכמים בשני מקרים:

1. גישה לזיכרון
2. גישה לרגיסטרים

גישה לזיכרון

כאשר יש פקודה שקוראת/כותבת לזיכרון/מהזיכרון (פקודות store/load) אז ישנה גישה לזיכרון.

מה קורה אם באותו מחזור שעון שבו פקודה אחת ניגשת ל Memory כדי לעשות store וקיימת פקודה שבאותו מחזור שעון נמצאת בשלב ה Instruction fetch שניגשת ל Instruction memory כדי לעשות fetch

קיימות 2 פקודות שונות שניגשות לזיכרון **באותו מחזור שעון!**

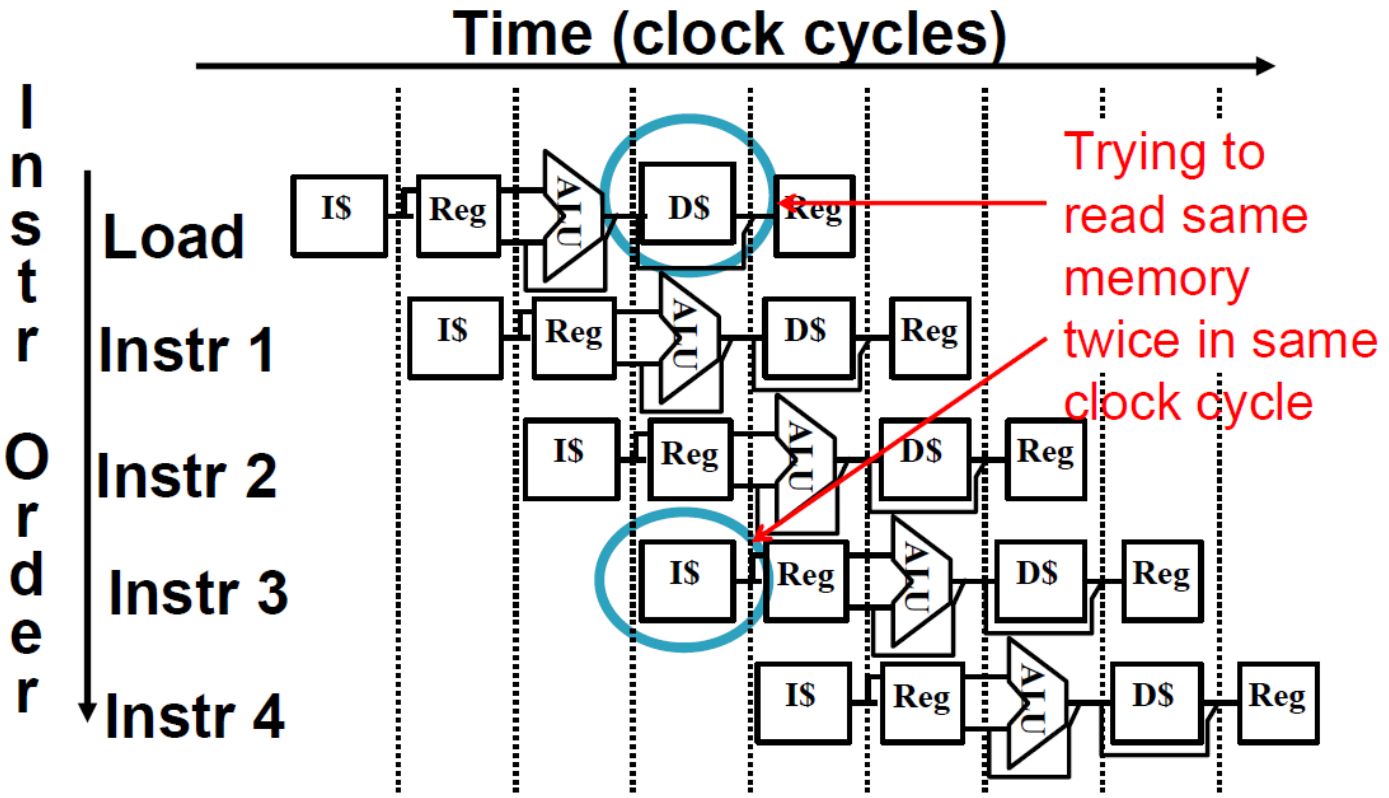
איך נפתור זאת?

בונים את הזיכרון כך שהבעיה לא תיווצר.

יוצרים הפרדה בין Instruction memory לבין ה data memory.

מה זאת אומרת? כלום.. מלכתחילה בונים את החומרה כך שלא תהיה בעיה כזו.

אילוסטרציה:



גישה לרגיסטרים

פקודות בשלב השני קוראות ערכים מתוך הרגיסטרים.

פקודות אחרות יכולות לכתוב לתוך הרגיסטרים בשלב החמישי.

מה קורה כשיש 2 פקודות כאלה במקביל אחת בשלב החמישי ואחת בשלב השני?

מה קורה כשיש פקודה אחת בשלב החמישי שכותבת לרגיסטר x ופקודה אחרת בשלב השני שקוראת מתוך רגיסטר x?

כלום.

מקבץ הרגיסטרים עומד בכתיבה וקריאה באותו מחזור שעון.

כאשר **כותבים** ערך לתוך רגיסטר הכתיבה נעשית ב**חצי הראשון** של מחזור השעון.

כאשר **קוראים** ערך מרגיסטר הקריאה נעשית ב**חצי השני** של מחזור השעון.

לכן אין התנגשות!

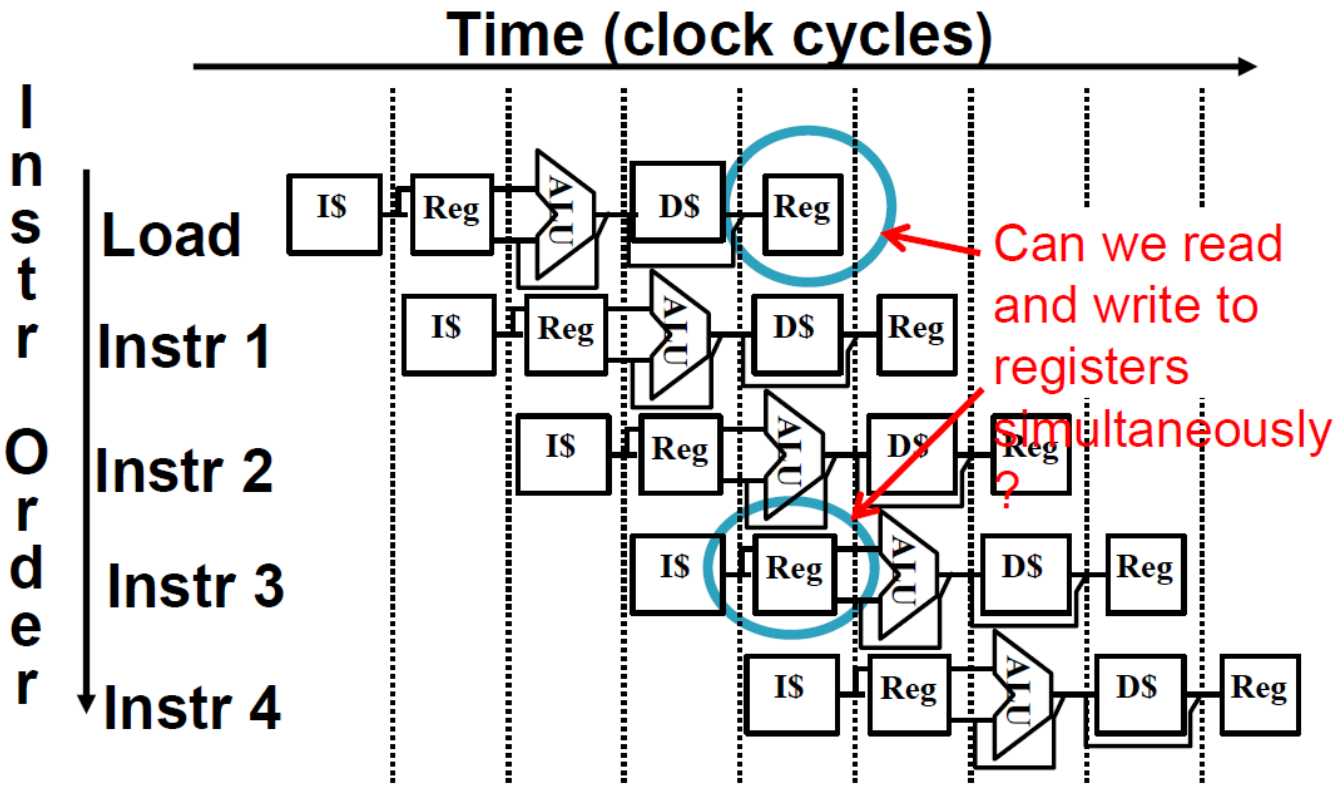
לפיצ'ר הזה קוראים "חצייית מקבץ האוגרים"

לכן, במידה ושתי פקודות נמצאות במצב הנתון, הפקודה שבשלב החמישי תכתוב **קודם** את הערך החדש לרגיסטר ובחצי השני של המחזור – הפקודה שבשלב השני תקרא את הערך **החדש.**

למה? בגלל חצייית מקבץ האוגרים!

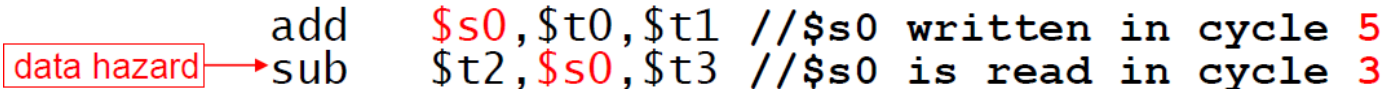
תזכרו את השם הזה! זה חשוב! תזכרו! עירן אמר אז תזכרו!

אילוסטרציה:



Data Hazard

נסתכל על הדוגמא הבאה שכבר ניתנה, ניתן לסוג הזה של data hazard את השם **write-read**



הפקודה add כותבת לתוך הרגיסטר $s0 .

הפקודה sub שתתבצע מיד אחריה קוראת מתוך הרגיסטר $s0.

נניח כי התוכנית מתחילה בשתי שורות אלו

הפקודה הראשונה **מחשבת** את הערך **בשלב השלישי** אך **כותבת** את הערך **בשלב החמישי**(מחזור שעון חמישי).

הפקודה אחריה תקרא את הערך של הרגיסטר **בשלב השני** (כאשר באותו מחזור שעון רק מתבצע חישוב הערך של הפקודה הקודמת).

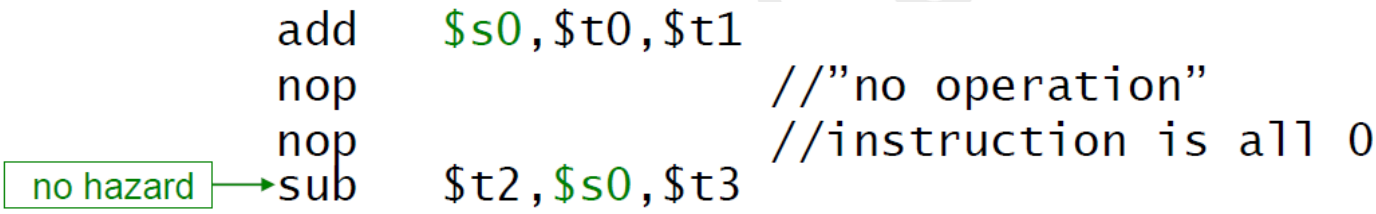
מה יקרה?

הפקודה השנייה תקרא את הערך הישן של רגיסטר $s0 !!!

הערך הנכון ייכנס לרגיסטר $s0 רק במחזור שעון החמישי!

אז מה נעשה?

הדרך הבנאלית לפתור write-read היא שבכל פעם שנתקל במצב דומה נכניס 2 פקודות nop (מה שיפגע בביצועים בצורה קריטית!!)



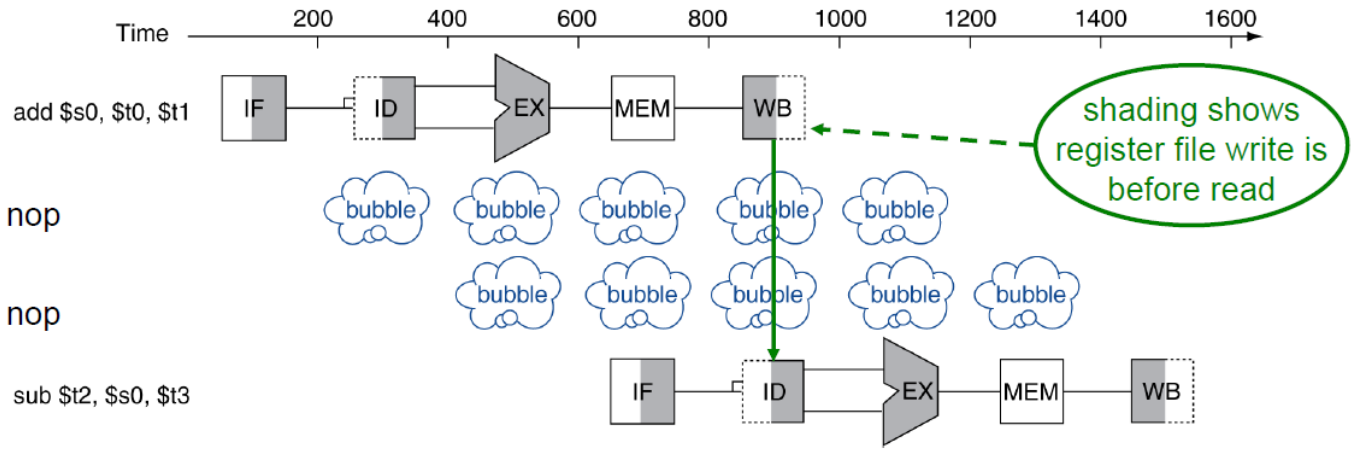
nop - היא פקודה שלא עושה דבר.
היא פקודה שמקודדת בשפת מכונה ל 32 אפסים ובעצם עושה shift left logical של 0 ביטים מרגיסטר 0 לרגיסטר 0 - או במילים אחרות "אל תעשה כלום".

נסתכל על nop בתור "בועה" שנתקעה בצינור ומשהה את התוכנית במקצת.

למה 2 פקודות nop ?

מכיוון שישנם 2 מחזורים בין המחזור החמישי לבין המחזור השלישי ולכן צריך לגשר על הפערים.

אילוסטרציה:



אבל רגע אחד.. בוא נתחכם קצת..

הרי אולי הפקודה add מכניסה את הערך הנכון לרגיסטר רק בשלב החמישי.. אבל היא יודעת את התוצאה כבר בסוף השלב השלישי!!!

במקביל, פקודת ה sub קוראת את הערך של הרגיסטר רק בשלב השני, אבל מתי היא **באמת** צריכה אותו?

בשלב השלישי!

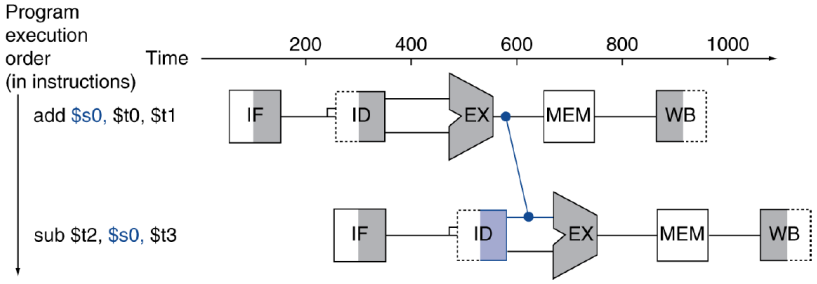
אז למה שלא נעשה קומבינה?

בוא נלך דרך השלב השלישי של פקודה add ישירות לשלב השלישי של הפקודה sub!

בעצם, אין סיבה שנעבור דרך הרגיסטרים אם נוצרת הבעיה!

ככה נפתור data hazard מסוג write-read

ניקח את התוצאה שניתנה מה- ALU במחזור השעון השלישי, נשמור את התוצאה והפקודה שלאחר מכן ב**מחזור השעון הבא** תיקח את אותה תוצאה שנשמרה



בואו נסתכל קצת יותר לעומק...

בתמונה הבאה נראה 5 שורות קוד ב MIPS כאשר באדום מסומנים בקוד פקודות עם סיכוני מידע (Data hazards) ובירוק מסומנים בקוד פקודות ללא סיכוני מידע.

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

בואו ונעקוב אחר התוכנית.

פקודת sub **כותבת לתוך** הרגיסטר $2.

שתי הפקודות שלאחריה **קוראות מתוך** הרגיסטר $2.

נסכים כי לא רלוונטי ש and קוראת אותו בתור rs ו or קוראת בתור rd.

נשים ❤ כי גם בפקודת ה and וגם בפקודת ה or קיימים data hazards!

למה?

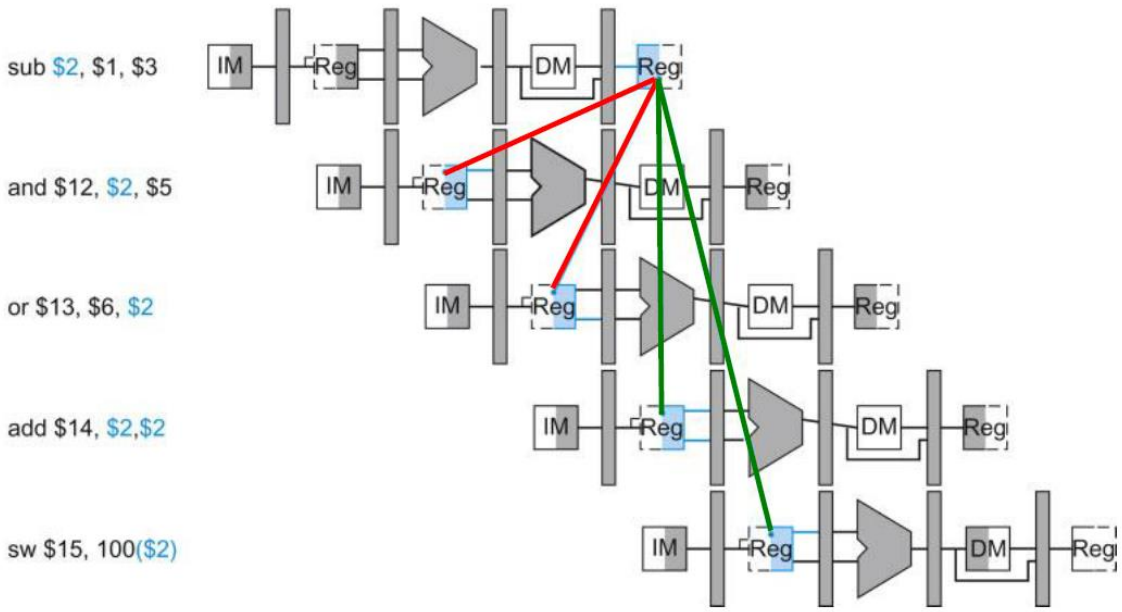
פקודת sub כותבת את הערך **במחזור החמישי** , פקודת ה and קוראת את הערך **במחזור שלישי** ופקודת ה or קוראת את הערך **במחזור רביעי**.

לכן הערכים שנקראים מ $2 הם עדיין הערכים **הישנים!**

הערך הנכון של $2 נכתב במחזור החמישי וכן פקודת add קוראת את הנתונים של $2 במחזור השעון החמישי ולכן היא קוראת את הנתונים הנכונים אפילו שקיים סיכון מידע (מכיוון שקיימת חצייית מקבץ האוגרים).

כמו כן גם פקודת ה sw מקבל את הערכים הנכונים!

להלן אילוסטרציה:



אחלה אחלה מתן.. אבל מה עם הקומבינה?!

אה כן.. נקרא לקומבינה Forwarding!

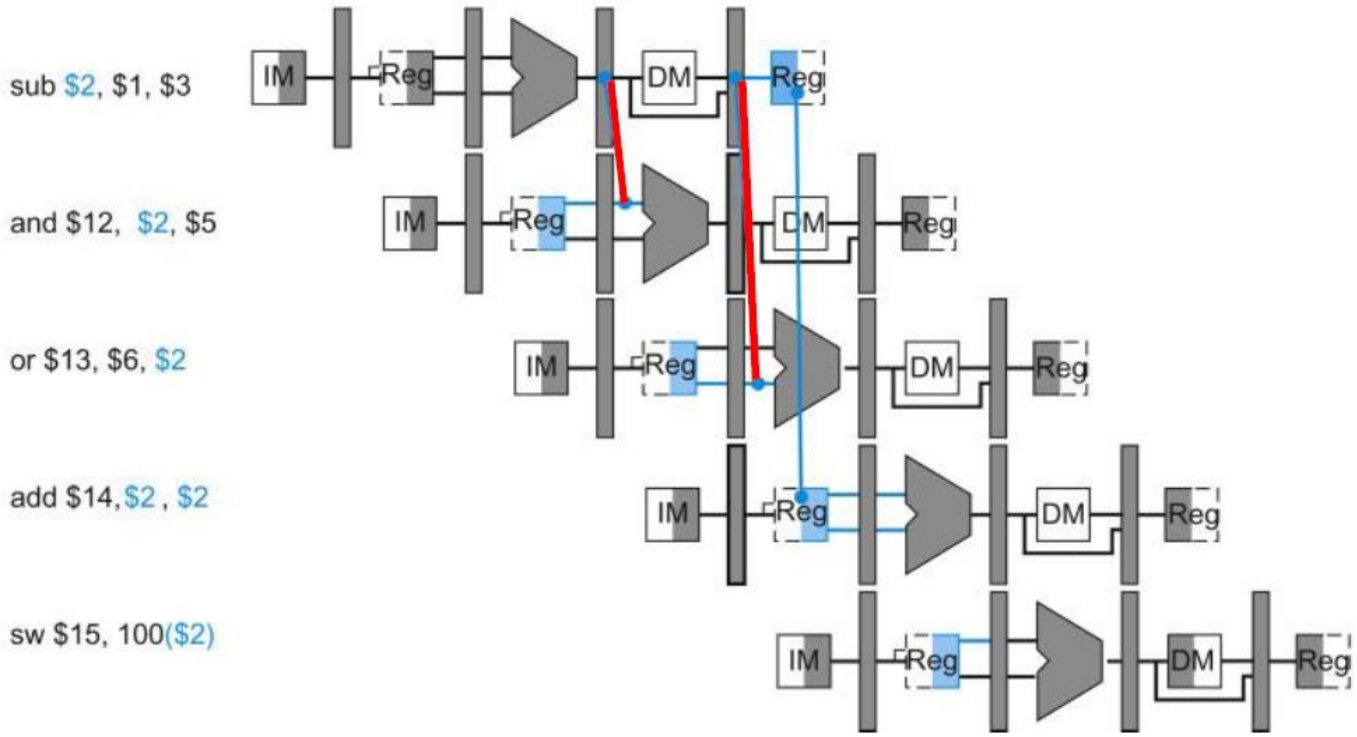
Forwarding

אז אנחנו רוצים לעקוף את המצב ה"לא נעים".. מה נעשה?

מעקף! (או forwarding במונח המקצועי).

במקום שנעביר את הערך הרצוי רק במחזור השעון החמישי "בצורה הפורמלית", נעביר את הערך הנכון ישירות מרגיסטר EX/MEM או מרגיסטר MEM/WB (בהתאם למחזור השעון בו מתבצע ה data hazard)

לדוגמא, באיור הבא:



נראה כי לפקודת and מועבר ערך ישירות מרגיסטר EX/MEM על מנת לתקן את המצב הבעייתי.

כמו כן לפקודת or מועבר ערך ישירות מרגיסטר MEM/WB על מנת לתקן את המצב הבעייתי.

נשים ❤ גם כי forwarding מעביר את הערך הנכון לכניסה הנכונה של ה ALU.

כנאמר מקודם פקודת ה add קוראת כבר ישירות ממקבץ האוגרים מטעמי הנוחות של חציית מקבץ האוגרים

אז איך באמת מבצעים זאת? בעמ' הבא!

יישום Data forwarding

על מנת ליישם את ה Data forwarding בשלב ה Execute (שלב 3) נצטרך לטפל בכמה וכמה דברים..

- ראשית נבין כי הכניסה ל ALU יכולה להגיע מכל pipeline register שאנחנו רוצים! לא רק מ ID/EX .

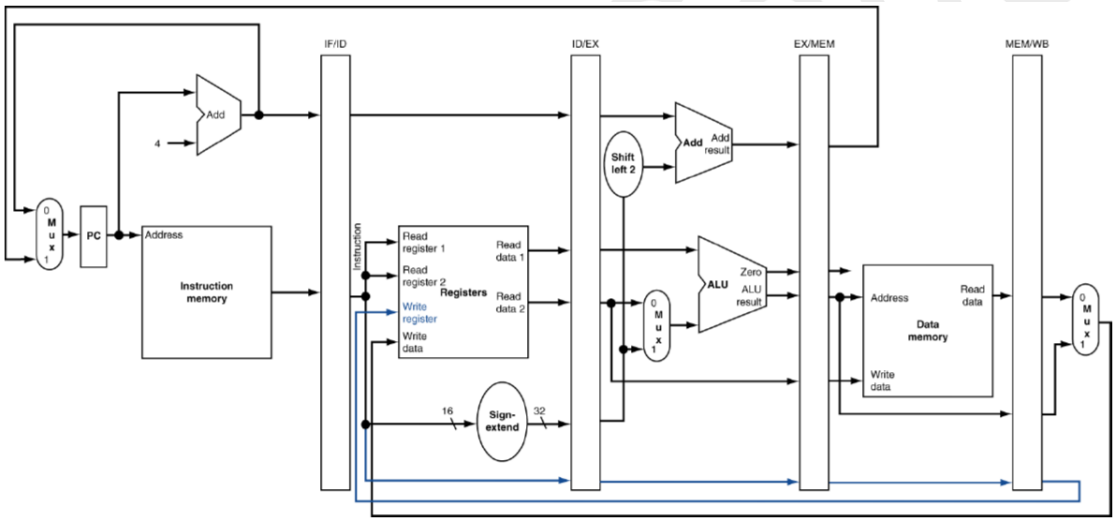
- נצטרך להוסיף שני מרבבים בעלי 3 כניסות כל אחד שיבררו בין הכניסות של ה ALU.
  - נקרא להם ForwardA ו ForwardB למען הנוחות.

- למרבבים נכניס את הכניסות:
  - הכניסה הרגילה שמגיעה ממקבץ האוגרים (מרגיסטר ID/EX).
  - הכניסה שיוצרת מעקף מרגיסטר EX/MEM.
  - הכניסה שיוצרת מעקף מרגיסטר MEM/WB.

- נוסיף יחידת בקרה למעקף שתוכל לנהל את המרבבים החדשים (נקרא לה forwarding-unit).
  - היחידה תבדוק מצבים של Data hazard.
  - היחידה תשלח את הסיגנלים הנכונים למרבבים ותברור בין הכניסות.

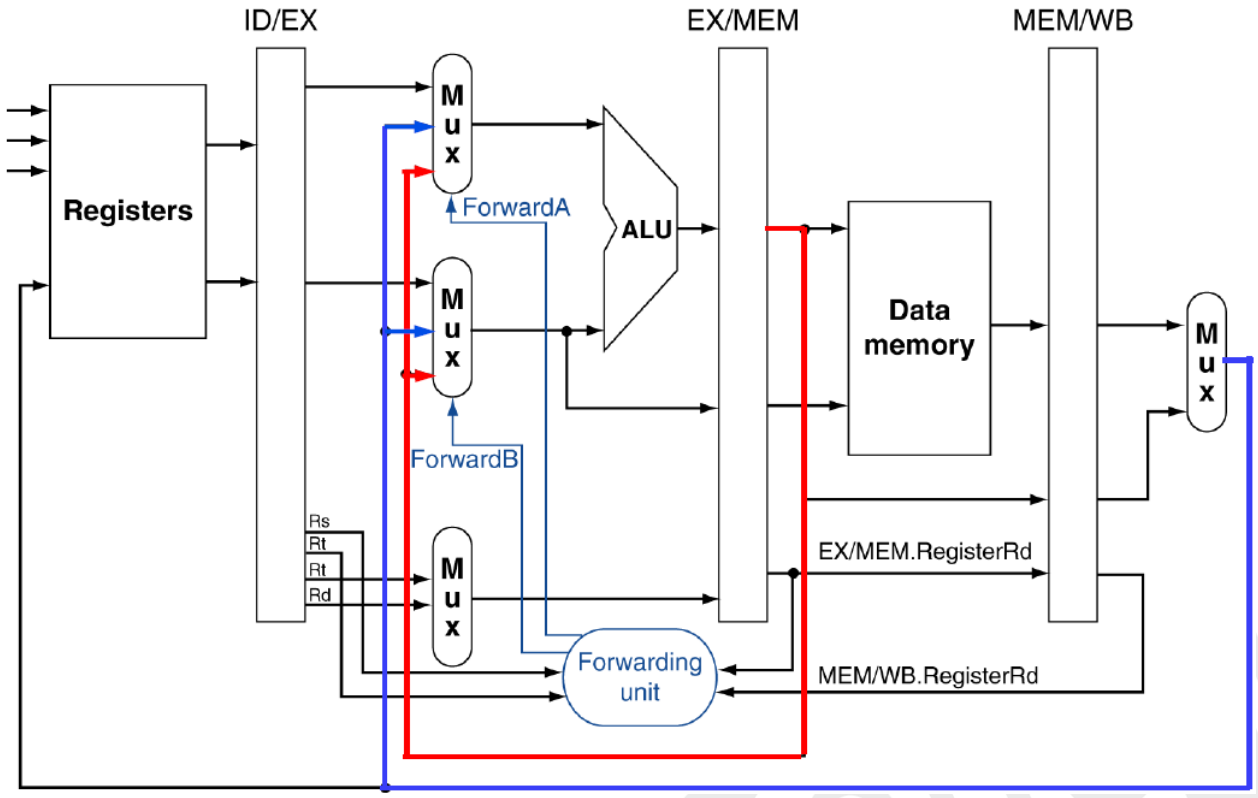
אבל מה אני מבלבל במוח.. בואו נסתכל על תרשימים שיעזרו להבין.

עד עכשיו הכרנו את המצב הבא:



בדף הבא נוכל לראות לראשונה את ה – Forwarding unit

אז אחרי שהכרנו בכלים ובכללים לביצוע השינויים , להלן ה Forwarding unit:



מה אנחנו רואים כאן?

על מנת שנוכל לבצע מעקף אנחנו צריכים לדעת אם קיים סיכון מידע.

ה forwarding unit אחראי על כך.

כמתואר בציור הרגיסטרים rt ו rs נכנסים אל ה forwarding unit (כדי לדעת איזה רגיסטרים הפקודה **הנוכחית** קוראת).

כמו כן, רגיסטר Rd (אותו רגיסטר שעבר במרבב שחיברנו קודם וחלחל קדימה בתור Rd) הוא אותו רגיסטר שאליו הפקודה **הקודמת** רשמה לתוכו.
לכן אותו רגיסטר Rd נכנס גם הוא לתוך ה forwarding unit.

אז מה בעצם עושה ה forwarding unit?

ה forwarding unit בעצם בודקת האם הכתובת של Rd שנכנס שווה לכתובת של rs **או** rt שנכנסו.

ההשוואה קוראת בשתי סיטואציות:

1. בזמן שיש הפרש של פקודה אחת בין הפקודות ואז הרגיסטר Rd שנלקח הוא של EX/MEM.RegisterRd
2. בזמן שיש הפרש של שתי פקודות בין הפקודות ואז הרגיסטר Rd שנלקח הוא של MEM/WB.RegisterRd

כלומר: האם הרגיסטר שהפקודה **שלפניי או זו שלפנייה כותבת אליו** הוא **אותו רגיסטר** שאני **קוראת ממנו?**
אם כן, יש Data hazard!

לבסוף במידה ויש Data hazard ואכן יש צורך ב Data forwarding אז ה forwarding unit שולח את הסיגנל המתאים (בהתאם ל rt ו rs) אל המרבבים ForwardA ו ForwardB על מנת להפעילם בצורה הנכונה ולהכניס את הערך הנכון אל ה ALU.

במידה והסיכון נוצר בהפרש של פקודה יחידה אז הערך יגיע מרגיסטר EX/MEM דרך החיבור באדום באיור.
במידה והסיכון נוצר בהפרש של שתי פקודות אז הערך יגיע מהמרבב שאחרי MEM/WB דרך החיבור בכחול באיור.

מתי מתרחשים Data hazards?

שאלה יפה.

קודם כל נגדיר את האוגרים בצורה הבאה: PipelineReg.RegisterRx

כאשר PieplineReg הוא אחד מהרגיסטרים של ההצנרה (ID/EX, EX/MEM, MEM/WB)

ו RegisterRx מסמן את אוגר היעד / מקור (rs, rt, rd).

נבחין כי סיכוני המידע מתקיימים במצבים הבאים:

מעקף מ EX/MEM
- 1א. EX/MEM.RegisterRd == ID/EX.RegisterRs
- 1ב. EX/MEM.RegisterRd == ID/EX.RegisterRt

מעקף מ MEM/WB
- 2א. MEM/WB.RegisterRd == ID/EX.RegisterRs
- 2ב. MEM/WB.RegisterRd == ID/EX.RegisterRt

הסימון של א' וב' נועד כדי להבחין בין 2 מצבים: הבחנה בין rs ל rt.

הסימון של 1 ו2 נועד כדי להבחין בין 2 מצבים נוספים: הבחנה בין Rd של EX/MEM או MEM/WB.

כמו כן נשים ❤ כי אם הפקודה "בטעות" מכניסה כתובת של רגיסטר שיוצר את אחת ההשוואות הללו גם אם היא בכלל לא קוראת ממנו אז ההשוואה עדיין תתקיים.

לכן צריך גם לוודא אם בכלל קיימת כתיבה לאותו אוגר שאנחנו חושדים בו בסיכון מידע ונבדוק גם:

- EX/MEM.RegWrite == 1 or MEM/WB.RegWrite == 1

כמו כן ידוע כי אם כותבים לאוגר $0 ($zero) הכתיבה לא מתבצעת בכלל אז נוודא שגם במצב הזה לא נתייחס לסיכון מידע:

- EX/MEM.RegisterRd ≠ 0
- MEM/WB.RegisterRd ≠ 0

בואו נקבץ זאת לתנאי:

מצב של מעקף בין שלב EX לשלב EX (בין פקודה אחת לזו שלפניה)

- if(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
  and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
  ForwardA = 10
- if(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
  and (EX/MEM.RegisterRd == ID/EX.RegisterRt))
  ForwardB = 10

מצב של מעקף בין שלב MEM לשלב EX (בין פקודה אחת לזו ששתיים לפניה):

- if(MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
  and (MEM/WB.RegisterRd == ID/EX.RegisterRs))
  ForwardA = 01
- if(MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
  and (MEM/WB.RegisterRd == ID/EX.RegisterRt))
  ForwardB = 01

Multiple Data Hazards

בואו נתבונן במצב הבא:

```
1    add $1,$1,$2
2    add $1,$1,$3
3    add $1,$1,$4
```

הפקודה הראשונה רושמת לתוך $1.

הפקודה השנייה קוראת מ $1 וגם כותבת ל $1.

הפקודה השלישית קוראת מ $1 וגם כותבת ל $1.

לכן נסכם כי קיים סיכון מידע מסוג **1א** בין שורה ראשונה לשנייה.

כמו כן קיים סיכון מידע מסוג **1א** בין הפקודה השנייה לשלישית.

וכמו כן קיים סיכון מידע מסוג **2א** בין הפקודה הראשונה לשלישית.

לכן קיימים 3 סיכוני מידע!! מה נעשה? איזה forwarding נבצע?

נזכור כי תמיד Data hazard מסוג 1 עוקף Data hazard מסוג 2.

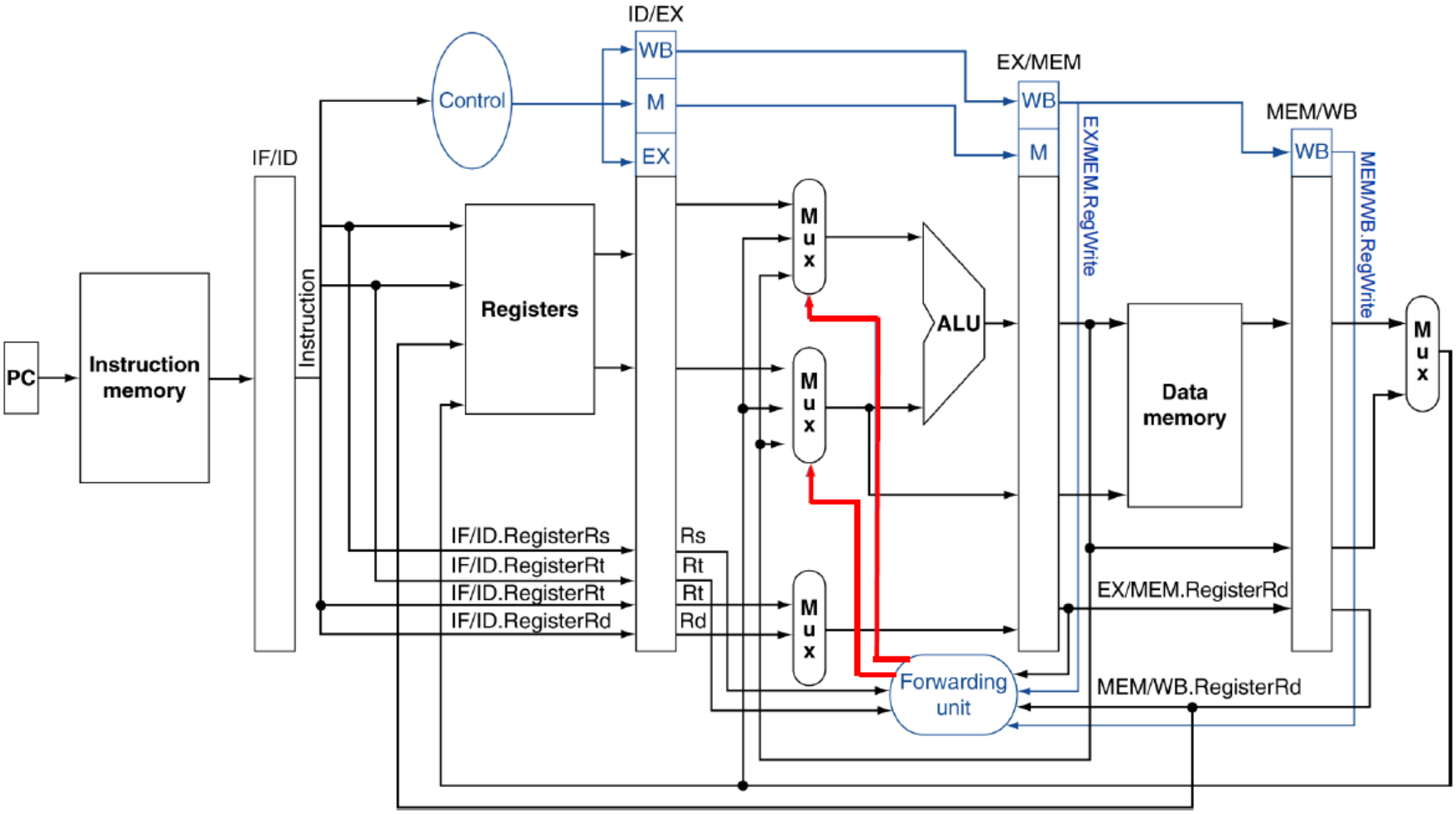
לכן צריך להכניס לנוסחה של סוג 2 פרמטר נוסף - בדיקה אם אין בו-זמנית Data hazard מסוג 1.

השינוי ימומש כך במצב של מעקף בין שלב MEM לשלב EX (בין פקודה אחת לזו ששתיים לפניה):

- ```
  if( MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
      and not { EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
                and (EX/MEM.RegisterRd == ID/EX.RegisterRs)}
      and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 01
  ```
- ```
  if( MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
      and not { EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
                and (EX/MEM.RegisterRd == ID/EX.RegisterRt)}
      and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 01
  ```

מה שנוסף (בכחול) הוא בעצם בדיקה שלא קיים Data Hazard מסוג 1.

בעמוד הבא תרשים של המעבד הכולל את ה Forwarding unit


ID/EX
EX/MEM
MEM/WB
IF/ID
Control
WB
M
EX
WB
M
WB
PC
Instruction memory
Instruction
Registers
M u x
ALU
Data memory
M u x
EX/MEM.RegWrite
MEM/WB.RegWrite
IF/ID.RegisterRs
IF/ID.RegisterRt
IF/ID.RegisterRt
IF/ID.RegisterRd
Rs
Rt
Rt
Rd
M u x
EX/MEM.RegisterRd
Forwarding unit
MEM/WB.RegisterRd

Data Hazard in MEM stage

בואו נתבונן בקטע הקוד הבא:

```
lw $1,4($2)  //memory read  in cycle 4
sw $1,24($3) //value needed in cycle 5
```

הפקודה הראשונה עושה **load** לאוגר $1.

הפקודה שלאחר מכן עושה **store** לאוגר $1.

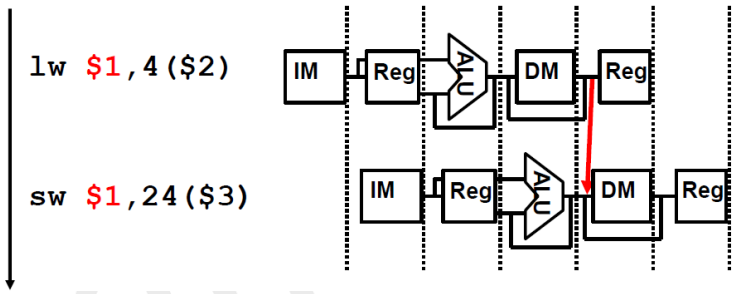
בעצם הטעינה של הערך לתוך האוגר $1 (הפקודה הראשונה) מתבצעת רק במחזור השעון החמישי בעוד הקריאה של הערך מ $1 לשם שמירתו קוראת במחזור שעון שלישי! (שלב שני של sw).

אז שוב יש Data hazard רק שהפעם בין פקודת load לפקודת store.

כאן נצטרך מעקף בין שלב MEM לשלב MEM!

מעקף זה יתבצע **רק** במקרים בהם יש load ומיד אחרי כן יש store!

המעקף יתבצע מהמידע שנמצא ב MEM/WB אל הכניסה של הזיכרון.



אז כמו בסוג הקודם של Data hazard, נכניס מרבב וקו בקרה שמגיע אליו ונבצע מעקף.

אבל לא תמיד אפשרי להשתמש ב Data forwarding!!!

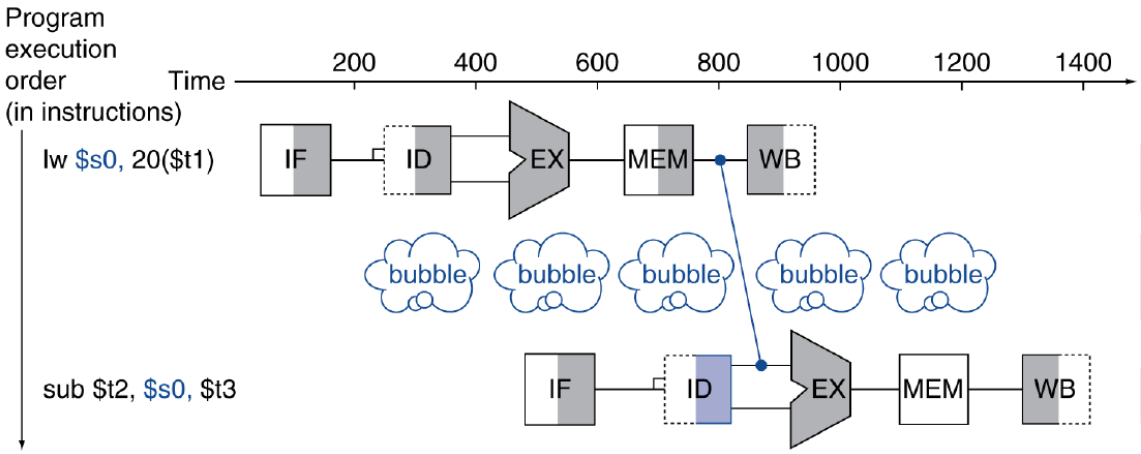
מה נעשה במצב הבא:

```
lw  $s0,20($s1) //memory is read  in cycle 4
sub $t2,$s0,$t3 //ALU needs value in cycle 4
```

פקודה שרושמת ערך מהזיכרון אל אוגר $s0 ופקודה אחריה (לא store) שצריכה את הערך של האוגר $s0.

פקודת sub צריכה את הערך כבר בשלב **השלישי**! הפתרון שהצענו למעקף לא טוב!

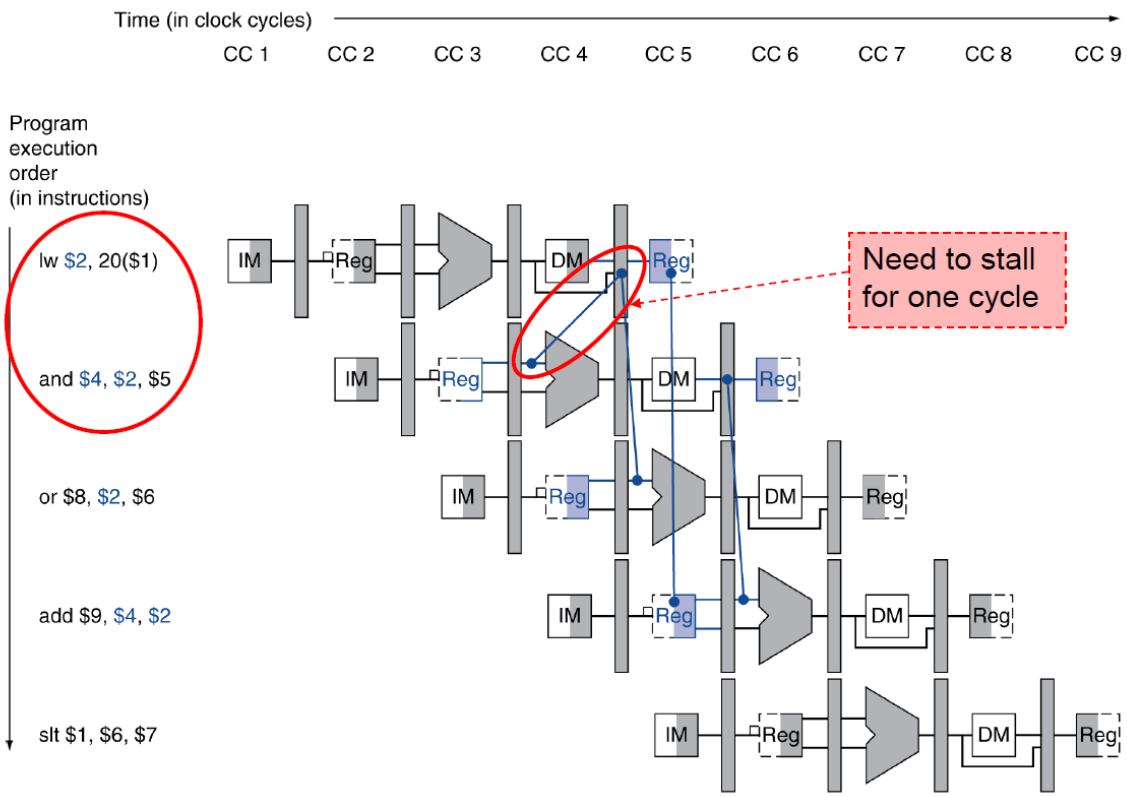
אין אפשרות להספיק לבצע forwarding בשל המרווח בין השלבים! חייבים להכניס Bubble (nop)



לסוג זה של Data hazard קוראים: Load-Use

קריאה של נתון לאוגר ופעולה צמודה!

אילוסטרציה של הבעיה (מסומן באדום):



זיהוי Load-Use

כיצד נזהה מצב של Load-Use Data hazard?

קודם כל, על מנת שנוכל לזהות את המצב צריך שפקודת ה Use תהיה בשלב השני שלה, מכיוון שרק בשלב השני אנחנו נדע מהי הפקודה ומהם האוגרים שבהם היא משתמשת.

במילים אחרות נוכל לזהות את המצב רק כשפקודת ה Load במצב Execute ופקודת ה Use בשלב ה Instruction decode.

מה תהיה הנוסחה שלנו למציאת הסיכון?

ID/EX.MemRead and

{(ID/EX.RegisterRt == IF/ID.RegisterRs) or

(ID/EX.RegisterRt == IF/ID.RegisterRt)}

בעצם נבדוק כי אכן הפקודה הקודמת היא פקודת load שכן ID/EX.MemRead מופעל.

לאחר מכן נבדוק אם האוגר אליו **טוענים** את הערך בפקודת ה load (אוגר rt) הוא אותו אוגר שבו פקודת ה use משתמשת (אוגר rt או rs).

במידה והתנאים מתקיימים ← נכניס Bubble!

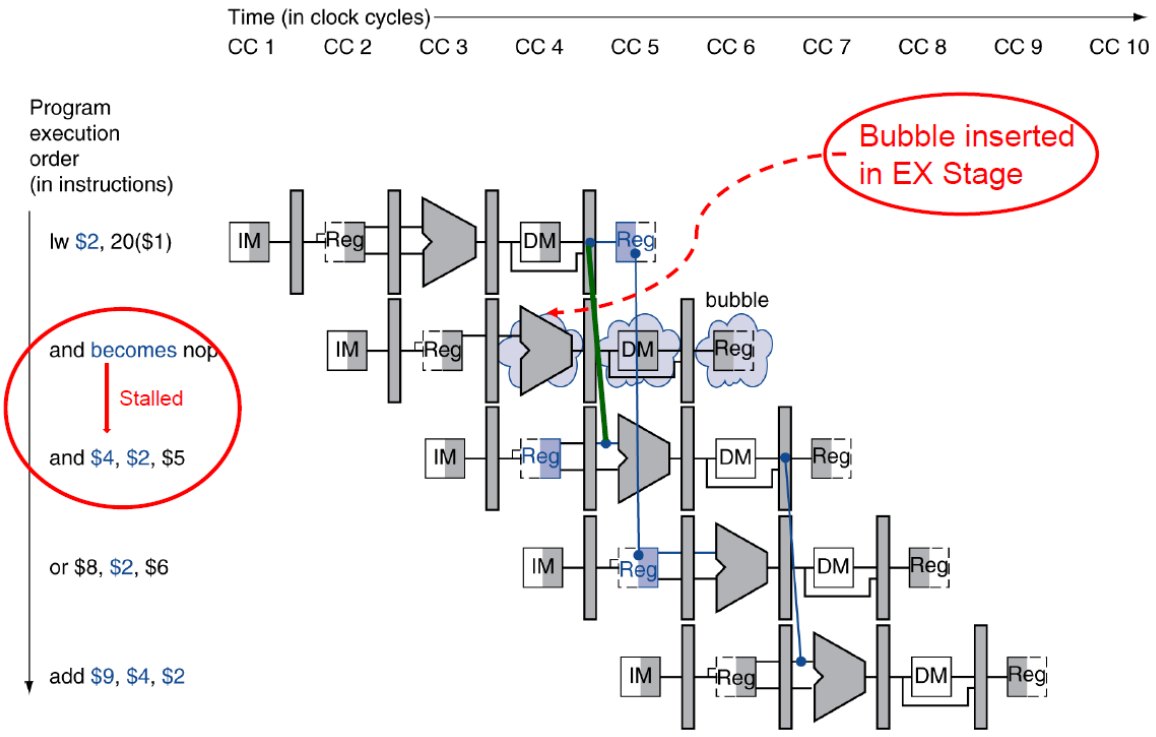
מיד אחרי ההכנסה של ה Bubble נכניס את ה forwarding שלנו על מנת לגשר על הפער!

נבצע forwarding בין הערך הנחוץ מ MEM/WB אל הכניסה המתאימה ב ALU.

אז אחרי שמזהים את הבעיה, **הופכים** את פקודת ה Use ל Bubble.

ז"א שפקודת ה and הופכת לפקודת nop בזמן שמקפיאים את השלב שבו היא והפקודות שאחריה נמצאות בו.

אילוסטרציה ראשונית:



איך מייצרים Bubble? בעמוד הבא!

אז איך מייצרים Bubble?

על מנת להפוך פקודה ל Bubble, ניקח את כל ה control bits **ונאפס** אותם!

משמעות הדבר היא שהפקודה תמשיך להתבצע, אך לדוגמא הסיגנלים של RegWrtie, MemRead, MemWrite, Branch (ניתן להפוך רק אותם) והסיגנלים הנוספים יהפכו ל 0 ! לכן לא תתבצע שום פעולה, הפקודה לא תשאיר את חותמה בשום מקום.

איך נקפיא את שלבי הפקודות?

על מנת "להקפיא" את שלבי הפקודות של פקודת ה Use והפקודות שאחריה נבצע את הדברים הבאים:

- נקפיא את ה pc (Program counter) למחזור שעון יחיד - אם ה pc לא יתקדם ב 4, לא נמשיך לפקודה הבאה בתוכנית.
- נקפיא את הרגיסטר IF/ID למחזור שעון יחיד - מכיוון שאותו רגיסטר שומר באותו זמן את הפקודה שרוצים להכניס לפני את ה Bubble, אז נקפיא אותו כדי לשמור את הנתונים שבו. לכן אותה פקודה תבצע Instruction decode **פעם נוספת!**

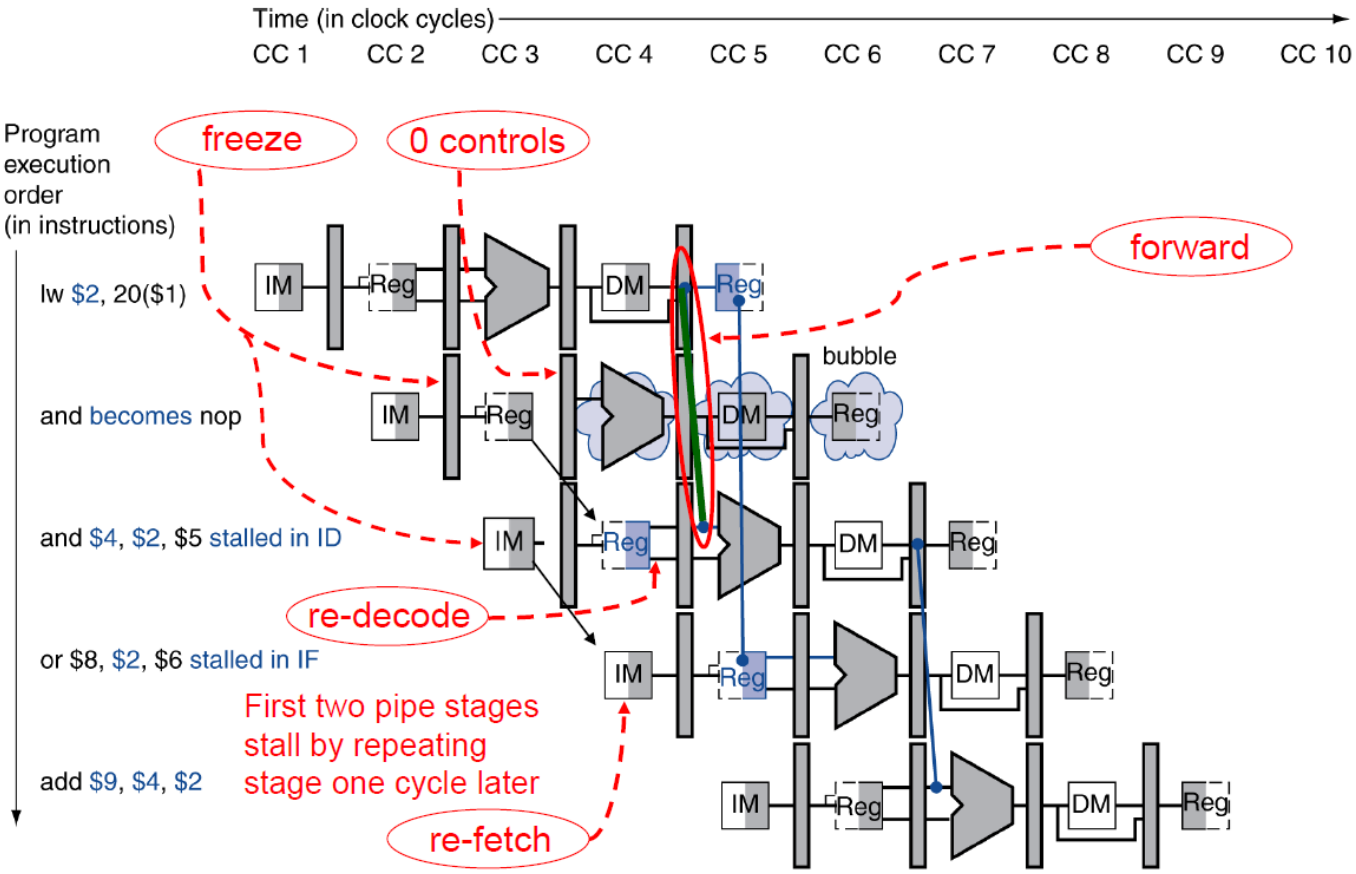
על מנת שנוכל לבצע את "ההקפאות" הללו נצטרך עוד **שני סיגנלים** חדשים שיופעלו על ידי יחידת בקרה חדשה שיהיה שמה בישראל - "Hazard detection unit" או בקיצור HDU.

לשני הסיגנלים החדשים נקרא:

- PCWrite שיהיה אחראי על "הקפאת" ה pc.
- IF/IDWrite שיהיה אחראי על "הקפאת" הערך ב IF/ID.

על מנת שנוכל להבין בצורה הטובה ביותר את כל מה שנכתב, בעמוד הבא נוספה אילוסטרציה והסברים מורחבים.
תחילה השרטוט נראה קצת מפחיד אבל מהר מאוד נבין אותו!

הלאה, לעמוד הבא!
ואל תשכחו ילדים, בכל הקשור לבועות - אל תהיו סקווידוויד!



באיור המתואר נראה כי אין דבר מעניין הקורה במחזור השעון הראשון והשני.

לעומת זאת, במחזור השעון **השלישי** מזהים כי ישנה בעיה וצריך להפוך את הפקודה השנייה ל Bubble.

על מנת לעשות זאת, נאפס את כל ה control bits (באיור 0 controls) ← מכאן והלאה הפקודה תהפוך ל Bubble!

כמו כן מקפיאים את הפקודה (את אוגר IF/ID ואת ה- pc) ולכן מצוירים שני חצים מהבועה freeze אחד אל תוך ה pipeline register IF/ID ואחד אל ה Instruction memory (IM).

בשלב שלאחר מכן פקודת ה Use (and) שוב עוברת decode (בגלל ההקפאה)לכן מצוירת בועת ה re-decode.

ולבסוף מתבצע re-fetch לפקודה שאחרי פקודת ה Use וכן מתבצע ה forwarding הרצוי בין אוגר MEM/WB לבין כניסת ה ALU של פקודת ה Use.

ולסיכום..

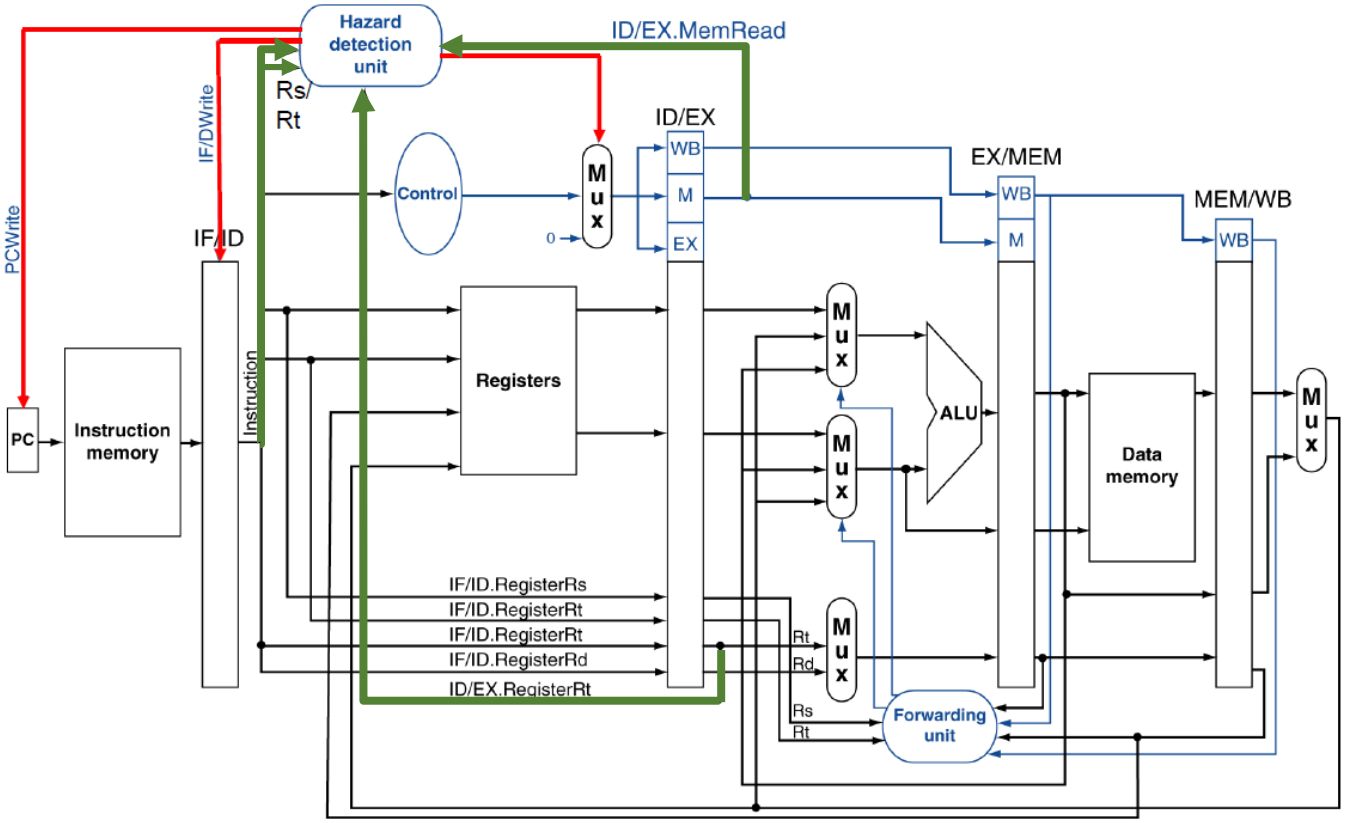
מי שאחראי לכל העבודה הנחמדה הזו הוא ה Hazard detection unit או בקיצור HDU

אל היחידה הזו נכנסים הרגיסטרים rs ו rt , את הרגיסטר Rd של שלב execute , את הסיגנל ID/EX.MemRead כדי לדעת האם הפקודה היא פקודת load.

אז במידה ו Rd=rs or Rd=rt וגם ID/EX.MemRead דולק, אז הוא ינתב את הדברים הבאים:

- סיגנל שיכנס למרבב שאחראי להעברת הסיגנלים ל control bits של הפקודה שנמצאת בשלב ה instruction decode ומאפס אותם בעזרת הכניסה 0 שנכנסת בנוסף לכניסה מה control וכך נייצר Bubble
- סיגנל IF/IDWrite שמאפשר/מונע כתיבה לאוגר IF/ID על מנת לאפשר לפקודה הנוכחית לעבור את שלב ה decode מחדש (re-decode).
- סיגנל PCWrite שמקפיא את ה pc ולא מעביר אותו ל pc+4

אילוסטרציה:

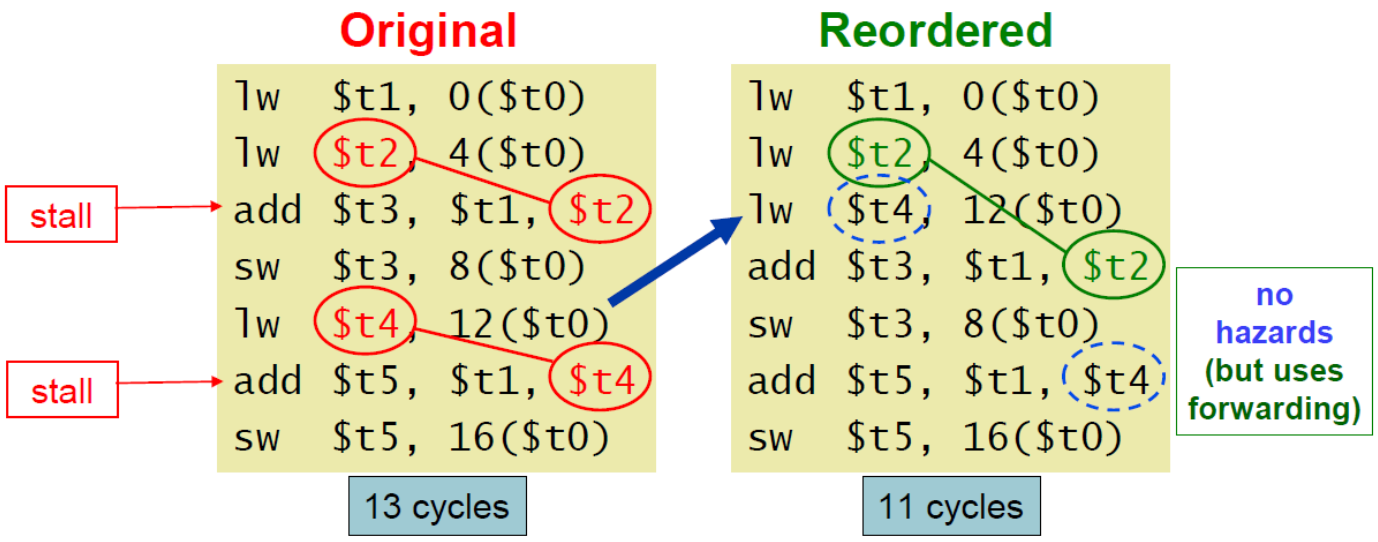


בואו ונסתכל על עוד פתרון לבעיית ה Load-Use, פתרון פשטני טיפה.

פתירת Load-Use בעזרת שינוי בקוד

במידה וישנה בעיית Load-Use ניתן לפתור את הבעיה בעזרת שינוי סדר הקוד.

נתבונן במעבר הבא:



הסידור מחדש בעצם שם את שתי פקודות ה load אחת אחרי השנייה ובכך מנה הכנסה של שני bubble לתוך הקוד (באיור השמאלי מוצג כ stall)