19/03/08

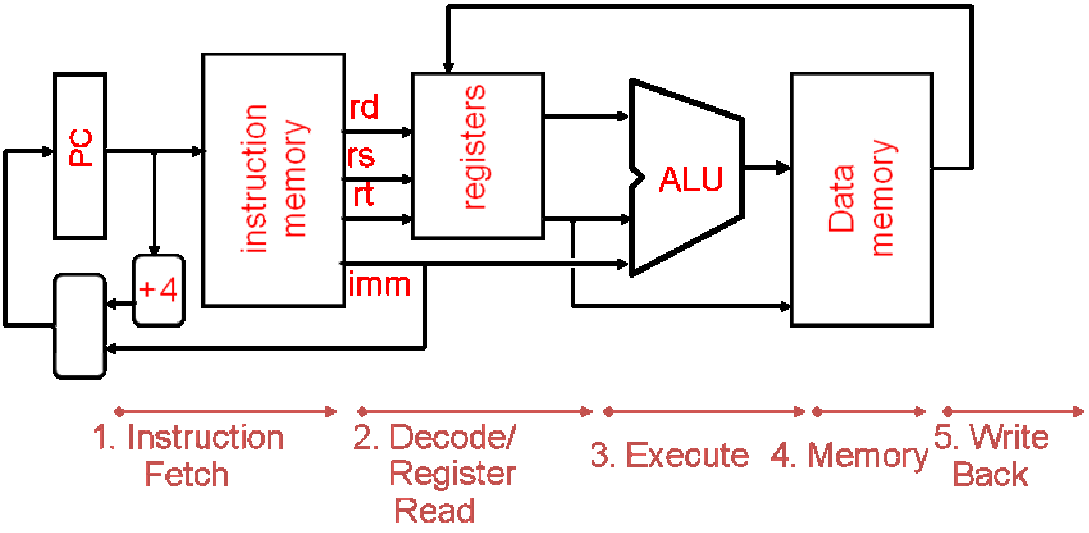
מבנה מחשבים הרצאה 9

ארכיטקטורת ה Pipeline

הרעיון של CPU העובד בארכיטקטורת Single Cycle שראינו בהרצאה הקודמת הוא פשוט להבנה אך לא מעשי בשל קצב הפעולה האיטי שלו.

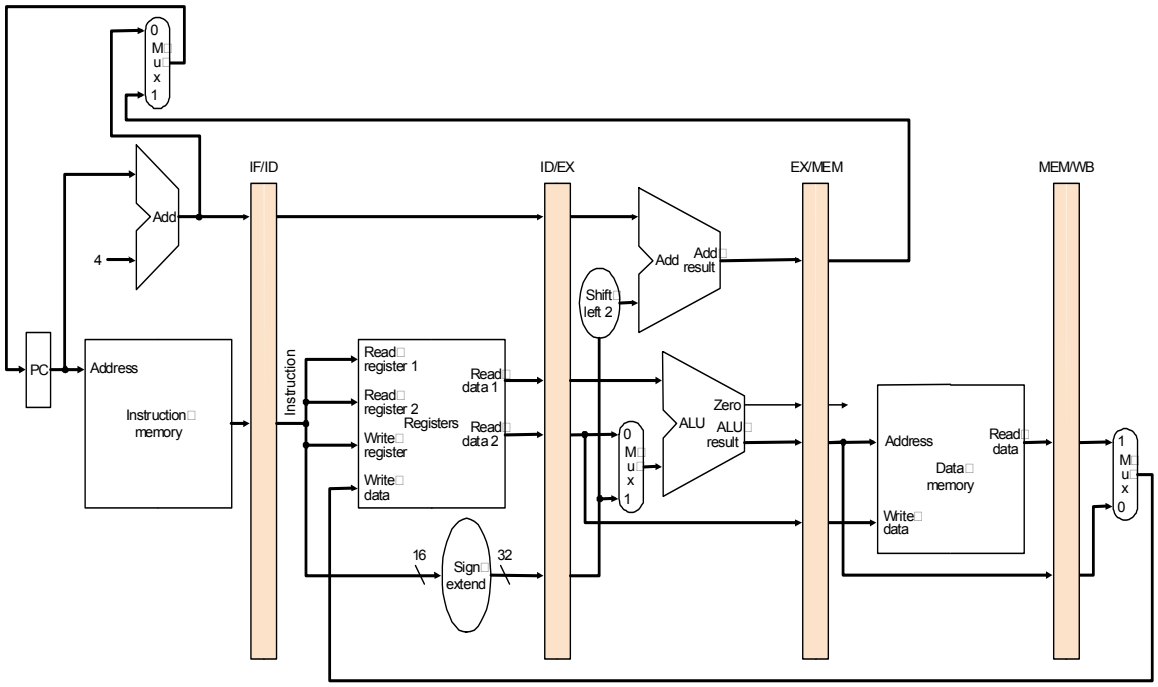
הרעיון מאחורי אכיטקטורת הpipeline הוא לבצע מספר פקודות בו זמנית, ולא רק אחת בכל פעם. מכיוון שכפי שראינו בהרצאה הקודמת, כל פקודה עוברת מספר רב של רכיבים עד שהיא מסתיימת להתבצע, אין שום סיבה שברגע שהיא עברה ברכיב מסוים הוא יהיה "מושבת" עד למחזור השעון הבא. רעיון הpipeline הוא לחלק את המעגל למספר "תחנות", כך ש(כמעט) בכל פעם שתחנה מתפנית פקודה חדשה נכנסת אליה, ומשתדלים לעשות זאת בצורה היעילה ביותר. בדיוק כמו במפעל המכוניות.....

נזכיר את השלבים בביצוע הפעולה:



כעת נרצה לחלק (פיזית) את המעגל שלנו לשלבים 1-5, כך שבמקום לחכות עד שפקודה תסיים את כל השלבים בשביל שהפקודה הבאה תתחיל אותם (כמו בSingle Cycle) נרצה שברגע שפקודה תסיים תחנה הפקודה הבאה מיד תתחיל את התחנה וכו' (זה לא תמיד אפשרי, נראה בהמשך). חיסכון בזמן היא מילת המפתח.

מכיוון שאנו מבצעים מספר פקודות בכל Cycle, עלינו איכשהו לזכור מה כל פקודה "עברה", כלומר לזכור את השלבים שכבר התבצעו עבור כל פקודה (אנלוגי לדוגמה של המכבסה- עלינו בכל שלב להחזיק סל כשהו עם הבגדים בשביל לעבור לשלב הבא). לשם כך אנו מוסיפים רגיסטרים מאסיביים בין כל שני שלבים, בכל מחזור שעון (עכשיו מחזורי השעון קצרים יותר) כל הפקודות מתקדמות לרגיסטר הבא. השמו של הרגיסטרים הם לפי שני השמות של שתי התחנות שהם מפרידים ביניהם- למשל הרגיסטר הראשון נקרא IF/ID כי הוא מפריד בין התחנה של ה IF (Instruction Fetch) לבין תחנת ה(Instruction Decode) ID.

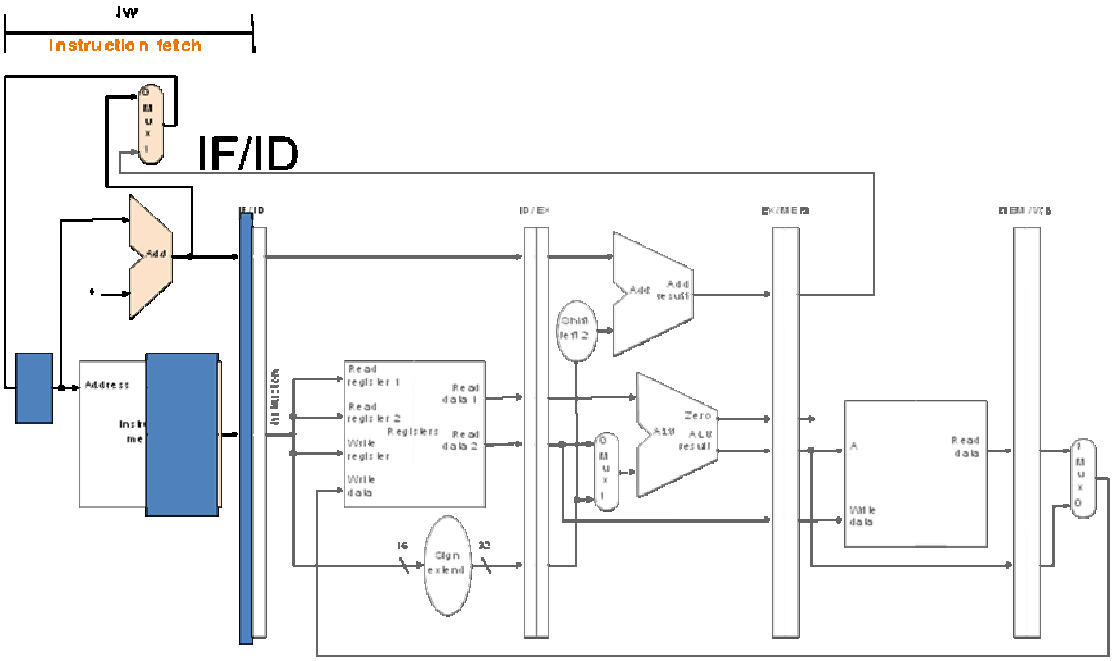


שימו לב שאין רגיסטר pipeline בסיום שלב הwrite back. הסיבה לכך היא שעד סוף השלב הזה אנו כבר בטוחים שהפעולה סיימה להתבצע- כלומר או שהיא כבר עדכנה רגיסטר (כמו בפקודות R-type או load), או שהיא כבר כתבה לזיכרון (כמו בפקודות store) או (בפקודות מסוג branch) היא כבר עדכנה את ערך הקפיצה של הPC.

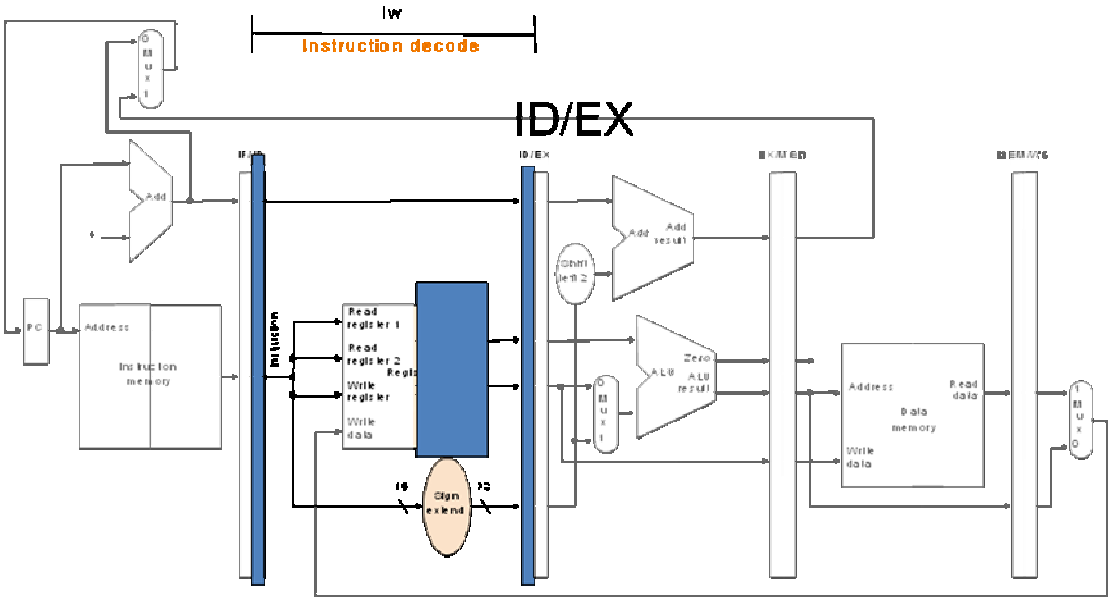
נסביר לדוגמה איך פועלת פקודת lw ברגיסטרים:

בסוף כל שלב יהיה ציור המתאר את השלב, כאשר נצבע את הצד הימני של זיכרון או רגיסטר כאשר מתבצעת קריאה **ממנו**, ונצבע את צדו השמאלי כאשר מתבצעת כתיבה **אליו**.

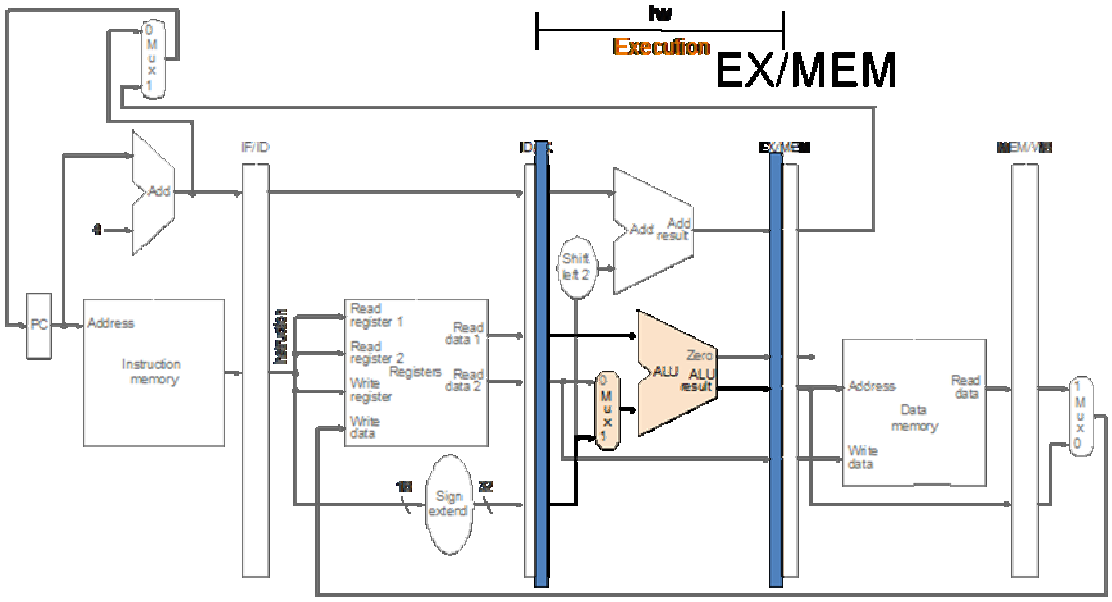
1. שלב הInstruction Fetch: בציור אנו רואים כי הפקודה נקראת מזיכרון הפקודות באמצעות הכתובת בPC, ואז מאוכסנת ברגיסטר ה IF/ID. רגיסטר IF/ID שומר בתוכו את הפקודה שהתקבלה מזיכרון הפקודות כאשר היא כבר מפוצלת. הכתובת שנמצאת בPC מועלת ב-4 ואז מוכנסת חזרה ל PC, מוכנה למחזור השעון הבא. הכתובת החדשה גם היא נשמרת ברגיסטר ה IF/ID למקרה שהיא תדרש מאוחר יותר בשביל איזשהי פקודה, כמו beq. המחשב לא מסוגל לדעת מהו סוג הפקודה שהוא עומד לבצע, לכן כל המידע שיכול להיות שיידרש נשמר.



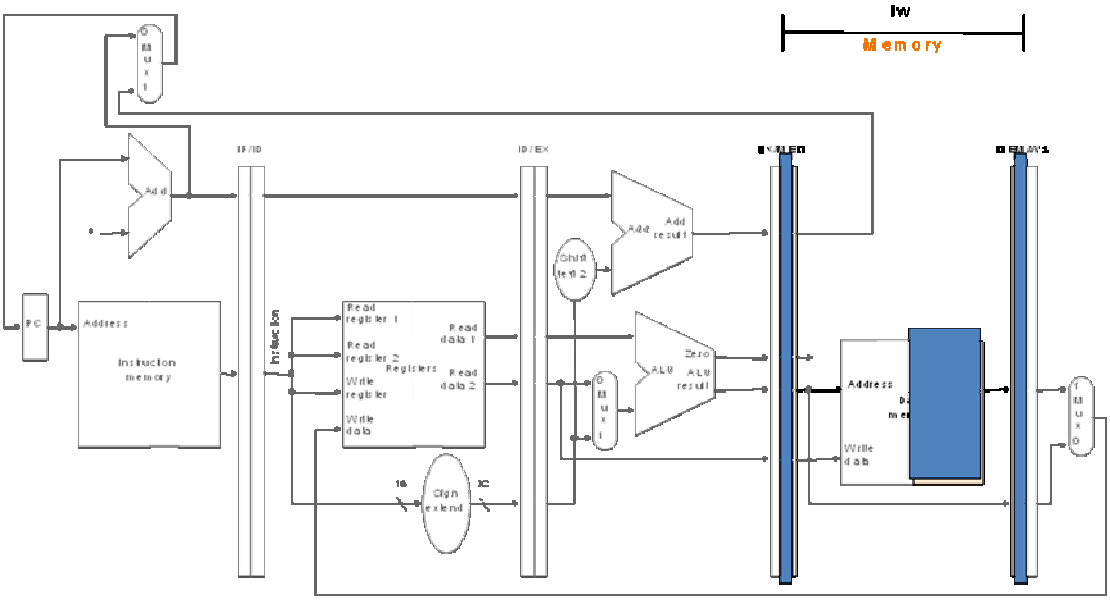
2. שלב הInstruction decode and register file read: בציור ניתן לראות כי ה IF/ID מספק את ה immediate של ה-16 ביט, שעובר הרחבת סימן ל32 ביט, וכן מספק ה IF/ID את מספרי הרגיסטר שאותו נרצה לקרוא (זה המייצג, יחד עם ה offset, את המקום בזיכרון ממנו נרצה לקרוא, וכרגע לא נשמור את מספרו של הרגיסטר השני, שהוא מספר הרגיסטר אליו נרצה לכתוב). שלושת ערכים אלו (immediate מורחב+ תוכן של הרגיסטר הראשון) וכן הPC שקודם העלנו ב4, כולם נשמרים ברגיסטר ה ID/EX. כמו מקודם, כל המידע האפשרי נשמר כי המחשב לא יודע מה הוא יצטרך בהמשך.



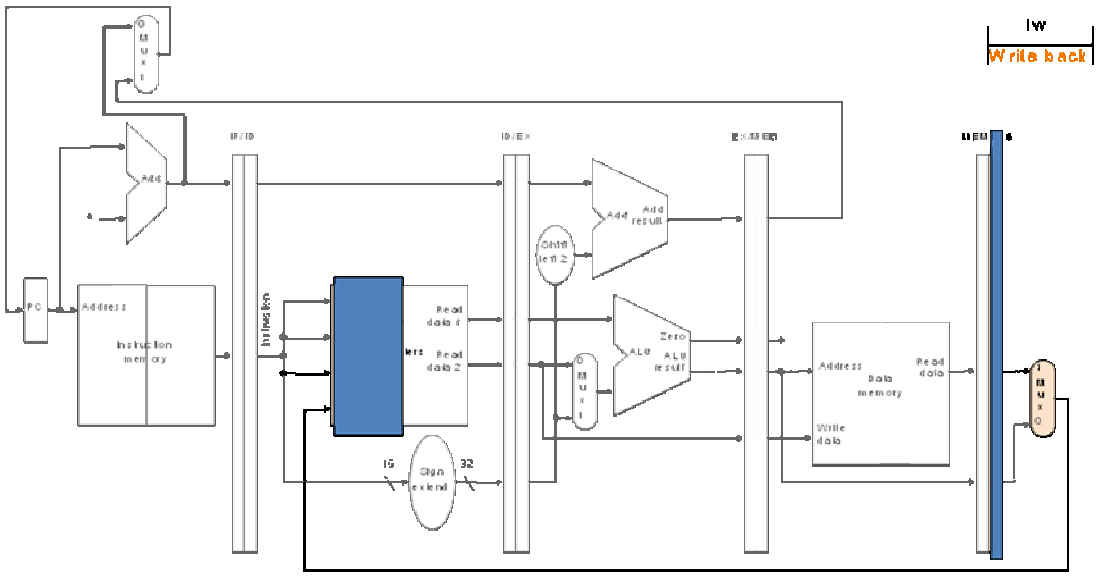
3. שלב ה Execute or address calculation: פקודת ה load קוראת את התוכן של הרגיסטר הראשון וה immediate המורחב (שני הנתונים הללו כזכור אוחסנו במחזור השעון הקודם ברגיסטר ה ID/EX ולכן נלקחים משם), ומחברת ביניהם. התוצאה מאוחסנת ברגיסטר ה EX/MEM (מכיוון שזוהי פקודת lw זה בעצם כתובת של מקום בזיכרון, אבל בכך נשתמש בשלב הבא).



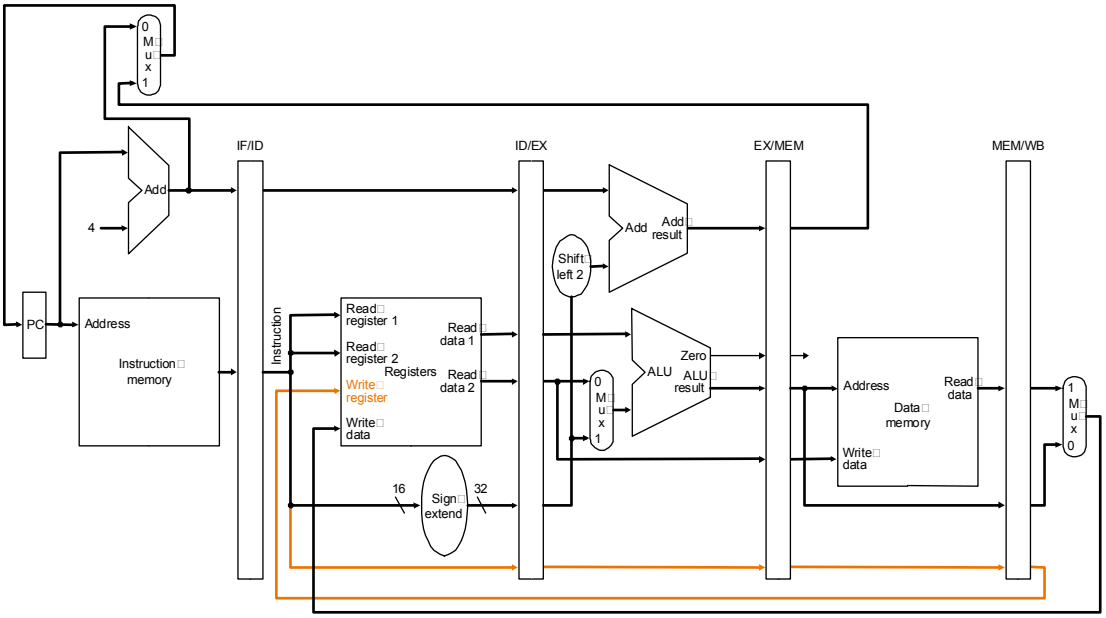
4. שלב ה Memory access: פקודת ה load ניגשת מקום בזיכרון שכתובתו אוחסנה במחזור הקודם ברגיסטר EX/MEM. התוכן של מקום זה בזיכרון מאוחסן ברגיסטר ה MEM/WB.



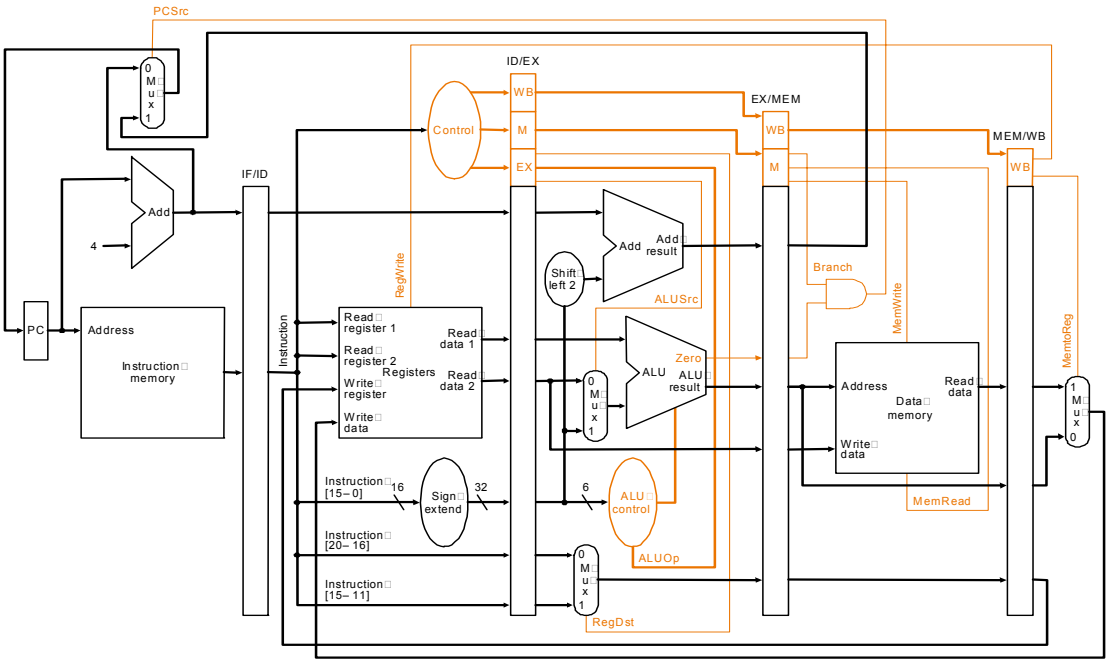
5. שלב ה write back: זה השלב הסופי שבו נלקח המידע שאוחסן במחזור הקודם ברגיסטר ה MEM/WB, ומידע זה מאוחסן ב Register File.



מכיוון שאין דרך אחרת לזכור לאיזה רגיסטר אנו מעניינים לאחסן בפקודת הload (רגיסטר ה Rd), אנו בכל שלב **<u>נוסיף</u>** לאוגרי הpipeline את מספר הרגיסטר שאליו רוצים לאחסן, זה נעשה באופן הבא:



כפי שעשינו במימוש של ה Single Cycle, אנו מוסיפים קווי בקרה עבור כל השלבים.
סיביות הבקרה גם הם נשמרים באוגרי ה pipeline יחד אם שאר נתוני הפקודה שעוברים בין האוגרים:



בהרצאה על ה pipeline שקפים 41-46 ישנה דוגמה של ביצוע סדרה של פקודות, מספיק פשוט בשביל להבין.

Pipeline Hazards:

אם עד עכשיו הכל היה ורוד, עכשיו נכנסים לתמונה מספר סיכונים (hazards) שלא לקחנו בחשבון.

ישנם 3 סוגים של hazrds:

- Structural Hazards: מקרה שבו פקודה אינה יכולה להתבצע במועד המתוכנן, מפני שהחומרה אינה יכולה להתמודד עם השילוב של הפקודות שהיא הייתה אמורה לבצע באותו מחזור שעון נתון.
- Control Hazards: נקרא גם branch hazard, קורה כאשר הפקודה הרצויה לא יכולה להתבצע מחזור השעון הרצוי, כי יכול להיות שהפקודה שמתבצעת היא לא זו שאמורה להתבצע, נגיד אם היה לפניה beq אז נצטרך לכאורה "לחכות" לתוצאת ההשוואה לפני שנחליט אם היא תתבצע או לא.
- Data Hazards: נקרא גם pipeline data hazards. מקרה שבו אנו לא יכולים לבצע פקודה במחזור השעון הרצוי מפני שההנתונים הדרושים לביצוע הפקודה עדיין לא מעודכנים, לדוגמה אם פקודה כלשהי אמורה לאחסן משהו בתוך רגיסטר והפקודה שאחריה אמורה לקרוא מרגיסטר זה, עליה לחכות עד שהפקודה הקודמת תסיים להתבצע.

פתרון של Data Hazards:

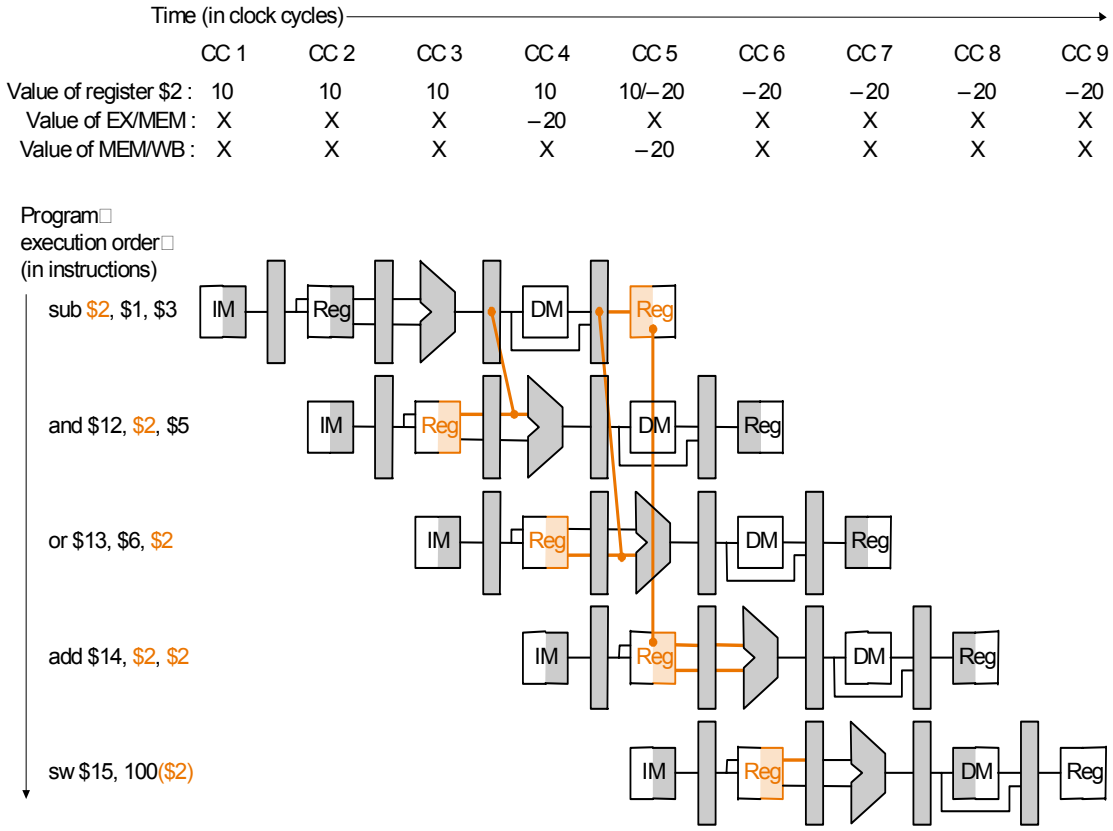
1) אחת הדרכים (המאוד לא מומלצות!) לפתרון סיכונים מסוג data hazards היא באמת לבדוק האם אחת הפקודות שכרגע במהלך ביצוע בpipeline אמורות לכתוב לתוך הרגיסטר שהפקודה הבאה צריכה לבצע, ואם כן "לחכות" עד שהפקודות יסיימו להתבצע. זה ניתן לעשייה ע"י הוספת פקודות מסוג nop- שזה קיצור של no operation – פקודות ש"מורחות" מחזורי שעון בלי לעשות כלום. עבור כל פקודה נבדוק מהם הרגיסטרים מהם היא אמורה לקרוא. אם הפקודה הקודמת כותבת לאחד מהם, נוסיף לפני הפקודה נוכחית 3 nop-ים. אם הפקודה שלפני הקודמת כותבת לאחד מהם, נוסיף לפני הפקודה הנוכחית 2 nop-ים. אם הפקודה שלפני הלפני קודמת (...מה לעשות...) כותבת לאחד מהם נוסיף לפני הפקודה הנוכחית nop אחד. כנאמר, הפתרון לא מומלץ בשל בזבזנות מחזורי השעון האופיינית לו.

2) דרך שניה, שהיא שיפור קטן של הפתרון הקודם היא, לשנות קצת את מימוש ה RegFile, ונוסיף משווה, שישווה בין הרגיסטר שקוראים ממנו לבין הרגיסטר שכותבים אליו. יש להבין שכתיבה לרגיסטר מתבצעת בחצי הראשון של השעון וקריאה היא בחצי השני, לכן אם נגלה כי הרגיסטר שאנו רוצים לכתוב אליו הוא גם הרגיסטר שאנו רוצים לקרוא ממנו, לא נחכה עד שהתוכן יכתב לתוך הרגיסטר בשביל לכתוב ממנו, אלא ישר נעביר את התוכן של הdata שעומד להיכתב ליציאת ה "read data" המתאימה. זה יחסוך לנו מחזור שעון (nop) אחד. כלומר אם בפקודה כלשהי רוצים לקרוא מרגיסטר שכותבים אליו בפקודה הקודמת, נשים לפניה רק 2 nop-ים. כבר טוב.

3) Forwarding- (גניבת ערכים)- ההבחנה שנעשית בפיתרון הזה היא שלפעמים מתקיים כי אנו רוצים לקרוא מרגיסטר, שהערך שלו אינו מעודכן, אבל (ופה האבל) הערך המעודכן שעומד להיכנס לרגיסטר ממנו אנו רוצים לקרוא כבר קיים איפשהו במעגל. לדוגמה נסתכל על רשימת הפקודות הבאה:

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

ניתן לייצג את הפקודות בתרשים זמנים:



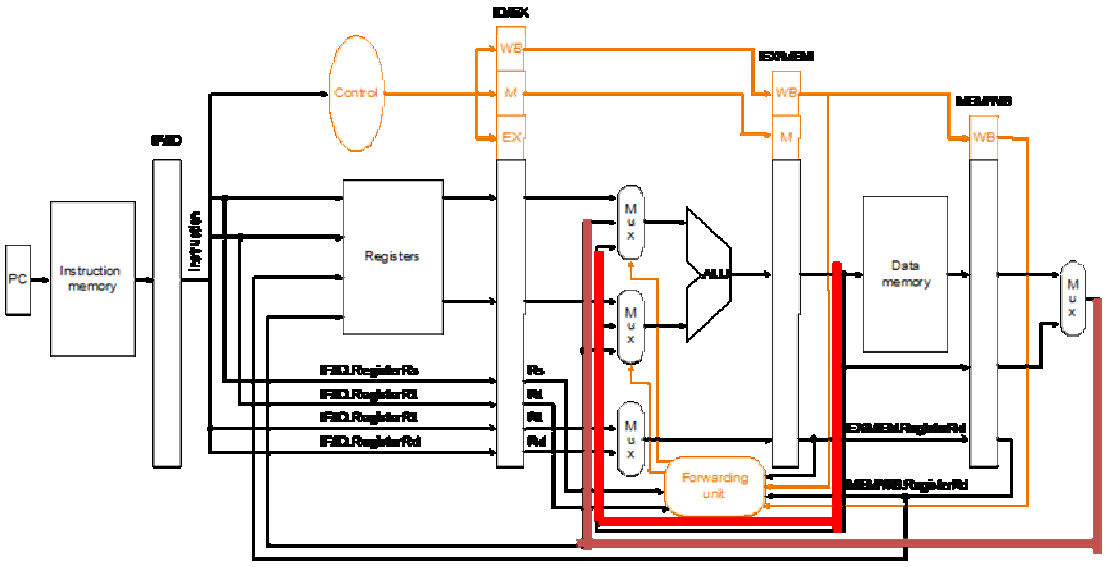
אנו רואים שהפקודה הראשונה- sub- כותבת לתוך $2.

הפקודה הבאה, and, צריכה לקרוא מרגיסטר זה, אבל כמובן שהיינו מצפים שהערך שבו פקודה זו תשתמש יהיה הערך המעודכן אשר חושב בפקודה הקודמת. לפי תרשים הזמנים אנו מבחינים שברגע שפקודת הand זקוקה לערך של $2, שזה בכניסה ל ALU, הערך המעודכן של $2 בדיוק נמצא **<u>ביציאה</u>** מהALU, כלומר בדיוק אוחסנה ברגיסטר ה EX/MEM . נגנוב את הערך! נחזיר אותו אחורה לכניסה של ה ALU (מבלי, כמובן, לפגוע באיזשהי מידה בביצוע פקודת ה sub). זה נעשה באמצעות חיבור המחבר בין רגיסטר ה EX/MEM לכניסות של ה ALU.

דומה עבור פקודת הor העוקבת, רק שהפעם הערך המעודכן של $2 כבר הגיע לרגיסטר ה MEM/WB, ולכן נצטרך "לגנוב" אותו משם אל הכניסה אל ה ALU.

בפקודה הבאה, פקודת הAND, יש לנו מצב שבאותו מחזור שעון שבו אנו רוצים לכתוב אל תוך רגיסטר $2 אנו גם רוצים לקרוא ממנו. אבל על מצב כזה כבר חשבנו בפיתרון 2.

המימוש של פתרון זה נעשה באמצעות רכיב חדש בשם Forwarding Unit שהתפקיד שלו הוא כרכיב בקרה שבאמצעותו אנו מחליטים מה יכנס לALU- האם זה יהיה אופן הפעולה הרגיל שבו פשוט ניקח את הערכים שנמצאים ברגיסטר ה ID/EX או שמא נצטרך לגנוב מרגיסטר ה EX/MEM או מרגיסטר ה MEM/WB....? שימו לב למימוש:



<u>הפיתרון לא תמיד עובד- בעית הlw</u>

מקרה אחד שבו פיתרון ה forwarding לא עובד הוא כאשר פקודה שמנסה לקרוא מרגיסטר עוקבת פקודת lw שרוצה לכתוב לאותו רגיסטר.

הבעיה במקרה זה מתוארת בסדרת הפקודות הבאה:

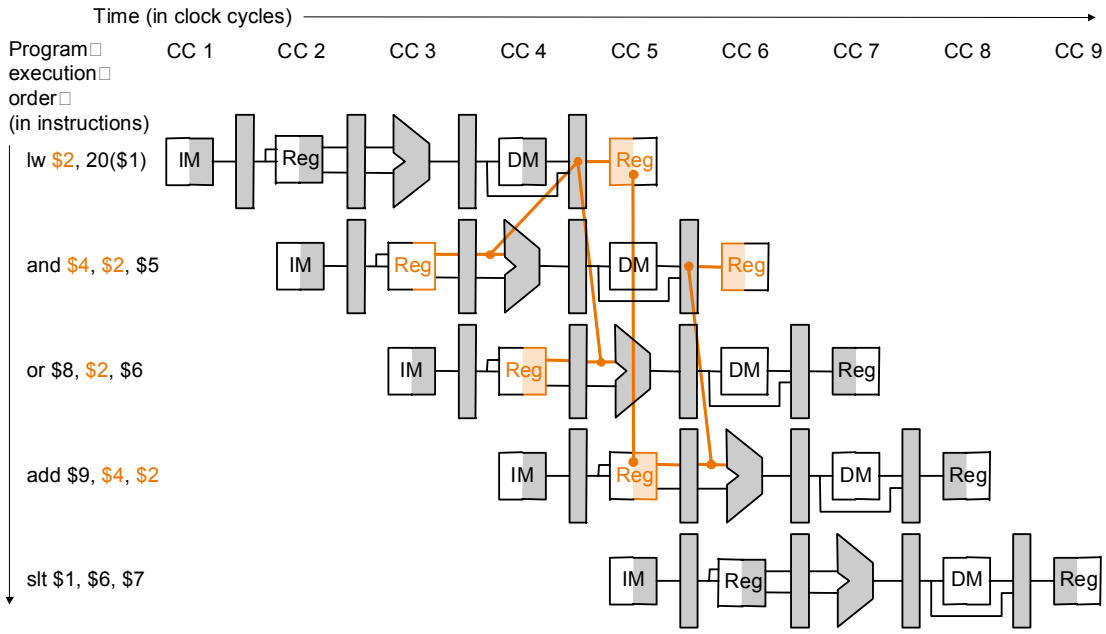
lw $2, 20($1)

and $4, $2, $5

or $8, $2, $6

add $9, $4, $2

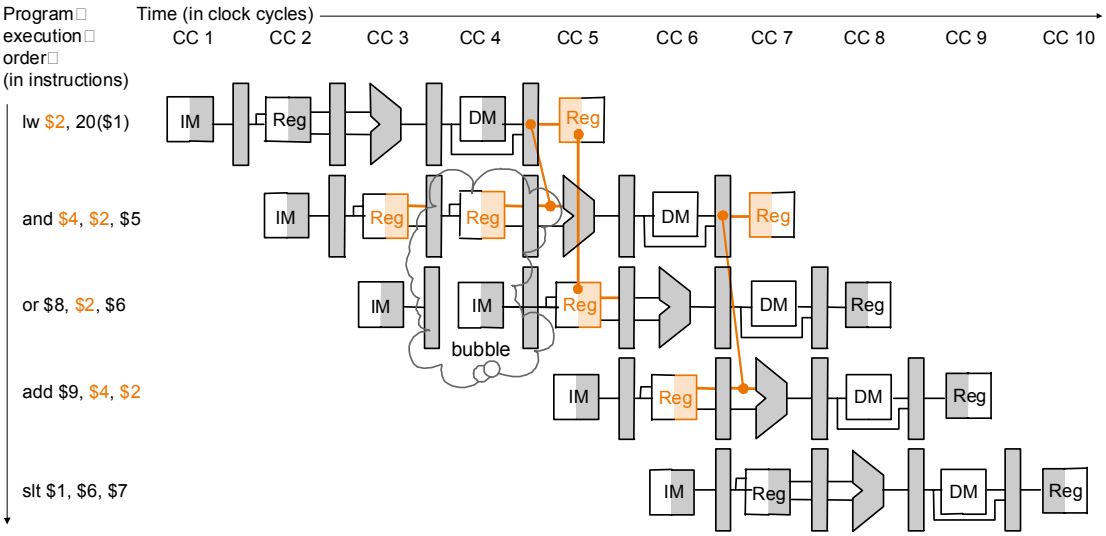
slt $1, $6, $7



הבעיה במקרה זה היא שפקודת הand רוצה לקרוא מרגיסטר $2, כאשר פקודת ה lw הקודמת אליה עדיין בתהליך כתיבה לתוך רגיסטר זה. למה אי אפשר לבצע כאן forwarding מהיציאה של ה ALU אל הכניסה שלו, כפי שעשינו קודם? כי זיכרו שבפקודות lw מה שממחושב בALU זה הכתובת של המקום בזיכרון אליו רוצים לגשת, ולא הערך שרוצים לאכסן ברגיסטר, אז הטכניקה הקודמת לא תעזור לנו. אנו נצטרך לחכות עד שבאמת יהיה לנו את הערך שאמור להיכנס לרגיסטר $2, נצטרך לחכות עד שערך זה יקרא מהכתובת בזיכרון (ב data memory) וימצא ביציאה של ה data memory, כלומר נאלץ לחכות מחזור שעון.

פיתרון זה נקרא "bubble"- במידה ונזהה מקרה כמו שהוזכר לעיל, נעכב את המעגל מחזור שעון אחד, מאין nop חומרתי, כדי לחכות שהערך שרוצים להכניס לרגיסטר ימצא ביציאה מה data memory, ואז "נגנוב" אותו משם.

מימוש פתרון הbubble מתבצע באמצעות הוספת רכיב בשם "Hazard Detection Unit". אנו מזהים את בעיית הlw כבר בשלב ה ID, ע"י כך שאנו רואים מה הערך של ביט ה MemRead, אם מתבצעת קריאה מהזיכרון, אז זוהי פקודת lw, וכן אנו משווים בין הרגיסטר שפקודת הlw רוצה לכתוב אליו לבין הרגיסטרים שהפקודה הבאה רוצה לקרוא. אם יש התאמה אנו מאפסים את כל שדות הבקרה EX, MEM, WB ברגיסטר ה ID/EX כך שלא תתבצע כתיבה לרגיסטרים או לזיכרון. כמו כן אנו לא נותנים לPC לעלות, כלומר הוא נשאר באותו מקום. אנו מכריחים את 2 הפקודות העוקבות לפקודת הlw לבצע את הלב האחרון שהם ביצעו פעם נוספת. כפי שמתואר בציור הבא:



כאשר פה פקודת הand והor מבצעות במחזור שעון 4 בדיוק מה שהם ביצעו במחזור שעון 3, כי מכיוון שלא העלינו את הPC, פקודת הor נאספת מחדש מזיכרון הפקודות, ופקודת הand קוראת את אותם רגיסטרים בRegFile ומפענחת אותם.

Branch Hazards

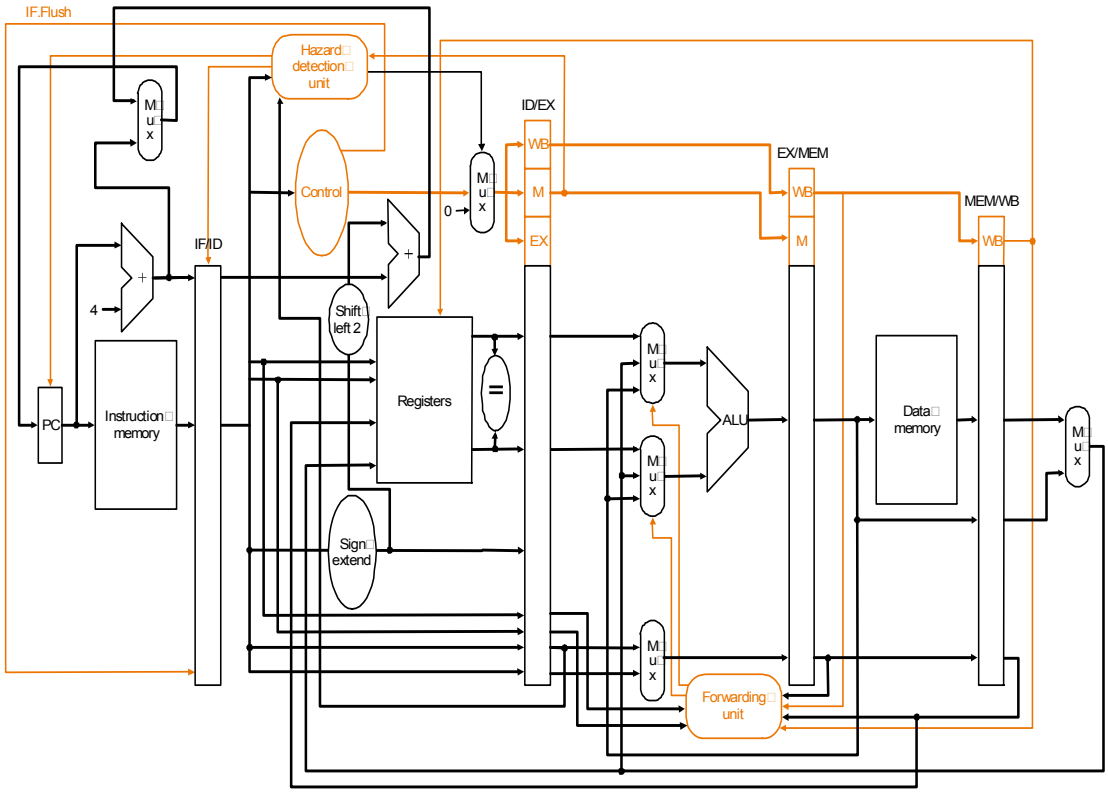
בעיה נוספת שצצה היא בעית ה branch: איך נידע אם הbranch יצליח או לא? שהרי עד שנחליט כבר הפקודה הבאה תיכנס לpipeline! יכול להיות שהbranch יצליח, אבל נידע זאת רק אחרי שהפקודה הבאה התחילה להתבצע, (יכול להיות שהפקודה העוקבת לbranch משנה ערך של רגיסטר). עלינו להרוג את הפקודה לפני שהיא תתבצע!

פתרונות:

1) הפתרון הראשוני הוא הוספת 3 nop-ים אחרי פקודות branch, אחרי 3 מחזורי שעון פקודת הbranch תסתיים להתבצע והPC יעודכן בהתאם לתוצאה. אבל זה פיתרון לא יעיל.

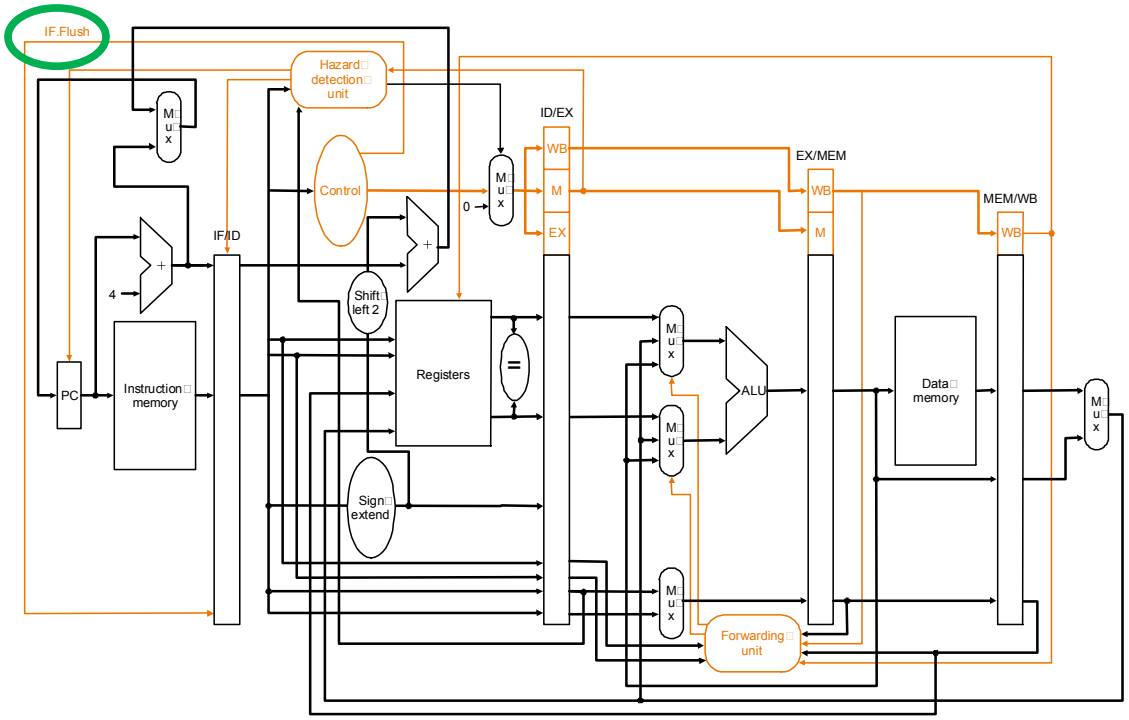
2) שיפור ראשון הוא העברת המשווה שבין ערכי הרגיסטרים שלב אחד אחורה, כלומר לשלב של הinstruction decode. אז נצטרך רק nop אחד אחרי כל פקודת branch, שזה עדיין טעון שיפור. יש להעיר כי שינוי כזה יגרום לכך שפקודת branch לא תעשה כלום בשלבי ה EX,MEM,WB.

השינוי ימומש באופן הבא:



3) שיפור נוסף הוא להגדיר מחדש את פקודת הbranch- לא משנה מה תהיה התוצאה, בכל זאת נחל בביצוע הפקודה הבאה שמיד אחרי פקודת הbranch. לשם כך, בהינתן רשימה של פקודות, נחפש מבין הפקודות שנמצאות לפני פקודת הbranch שלנו פקודה כזאת, כך שאם נשנה את סדר הפקודות ונשים אותה מיד אחרי פקודת הbranch, זה לא ישנה את התוכנית שלנו. אם מצאנו פקודה כזאת- הידד. אם לא נמצא כזאת פקודה, אז מיד אחרי פקודת הbranch נשים פקודת nop. סטטיסטית, שיפור זה יחסוך לך מחזור שעון במחצית מהמקרים! יש דוגמה פשוטה לאופטימזציה כזאת בהרצאה על הpipeline, שקפים 87-91.

4) Flushing- ניקח לדוגמה את פקודת beq. במקום לבצע טריקים של הקומפיילר נעשה משהו חומרתי. שוב נדרוש שבכל אופן הפקודה שאחרי פקודת הbranch תתחיל להתבצע, אבל אם הbranch הצליח נצטרך ל"נקות" אותה. מכיוון שהעברנו את ההשוואה של ה branch צעד אחד אחורה, לשלב הID, תהיה לנו, במצב זה, רק פקודה אחת "לנקות". הניקוי הזה נקרא flushing, והוא מתבצע ע"י הוספת קו בקרה שנקרא IF.Flush, שמאפס את רגיסטר ה IF/ID. דבר זה הופך את הפקודה שנלקחה למעין פקודת nop (כי היא סתם "בזבזה" מחזור שעון ולא עשתה כלום). המימוש נראה כך:


IF.Flush
Hazard detection unit
M u x
Control
0
ID/EX
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
IF/ID
4
Shift left 2
PC
Instruction memory
Registers
=
ALU
Data memory
Sign extend
Forwarding unit