סיכומים למבחן בקורס מבנה מחשבים

(סמסטר א' 2008-9, פרופ' נתן אינטרטור, פרופ' יהודה אפק)

חלק ראשון (פרופ' נתן אינטרטור)

הקדמה:

כל מחשב מורכב מחמישה חלקים עיקריים: *input, output, memory, processor: control and datapath*. כל אלו מקושרים ב-*busses*.

ישנו קו מפריד בין תוכנה לחומרה הקרוי *instruction set architecture*. אבסטרקציית שלבי החומרה תחתיו הם:

1. מעבד, זיכרון, מערכת הקלט/פלט
2. *datapath and control* (מעבד)
3. *digital design*
4. *circuit design*
5. *transistors*
6. *physics*

שלבי המעבר מתוכנה לחומרה: שפה גבוהה $\xleftarrow{compiler}$ שפת סף (*assembly*) $\xleftarrow{assembler}$ שפת מכונה $\xleftarrow{bytecode}$ שלבי החומרה. ה-*instruction set* היא אוסף פקודות המכונה, שמתורגמות בצורה ישירה משפת *assembly* לביטים. כלומר, *assembly* הוא תיאור מילולי מדוייק של אופן קריאת המכונה את הפקודות.

מושגים:

- *CPU (control processing unit)*: יחידת עיבוד מרכזית בסיסית. מבצע פעולות על:
- אוגרים (*registers*): רכיב חומרה המכיל 8/16/32/64 ביטים של מידע.
- *IS (instruction set)*: כאמור לעיל, אוסף הפקודות שמבצעת המכונה, שפת המכונה הממומשת על המחשב.
- *Addressing modes*: שיטת מיעון, כיצד מגיעים מהתוכנה למשתנים בזיכרון בחומרה.

תאימות (*compatibility*):

- תאימות לאחור (*backwards comp'*): כל מחשב חדש צריך להיות מסוגל להריץ תוכנות ישנות
- תאימות קדימה (*forward comp'*): תוכנה חדשה תהיה מסוגלת לרוץ על מחשב ישן (פחות קריטי מהראשון).
- *JIT – just in time compilation*: עקרון לפיו התוכנה מתקמפלת על כל מחשב ללא תלות בחומרה (כמו *JAVA* למשל).

יעילות ומהירות *CPU*:

- שימוש בחבילות תוכנה ואלגוריתמים שונים להשוואת יעילות פעולת *CPU* שונים.
- כיום מודדים *CPI (clock per instruction)*: כמה מחזורי שעון מבזבז ה-*CPU* עבור כל פקודה (נמדד ב-*nsec*).

CPU time:

המטרה שלנו היא לצמצם את זמן ה-*CPU*:

- הקטנת מחזור שעון (יותר מחזורי שעון לשניה).
- הקטנת ה-*CPI* (פחות מחזורי שעון לכל פעולה).
- צמצום מספר ה-*instructions* (ה-*IC*, *instruction count*), ע"י שיפור הקומפיילר, תוכנה.

Ahmdal's law: אם ניתן לשפר רק חלק מהביצועים, נשאף לשפר את החלק הגדול יותר, כלומר השכיח ביותר בזמן ריצת תוכנית

חשיבות תכנות ה-*ISA* (פקודות שפת המכונה):

- גודל כל פקודה: יכול להשפיע על זמן הקריאה לפקודה ועל כמות הזיכרון הנדרשת כדי לשמור אותה
- *IC* (מספר הפקודות): הפחת זמן הריצה ע"י הפחתת מספר הפקודות.

לכן נשאף לשפת מכונה פשוטה ומינימלית, תוך שניתן למשש איתה את כל התוכניות הרצויות.

***RISC vs. CISC* - שתי גישות לשפת המכונה**:

- *CISC (complex IS computer)*: כדוגמת ה-*x86* של אינטל, מגוון פקודות גדול שלא כולן באותו אורך, המביא לזמן קריאה לפקודה וביצועה ארוך (גם עבור פקודות פשוטות). בניגוד לחוק *Ahmdal*, דואג לטיפול במקרים הפחות שכיחים.
- *RISC (Reduced IS computer)*: כדוגמת ה-*MIPS*, מעט פקודות פשוטות בגודל קבוע, הגורר זמן *CPU* קצר יותר שכן זמן קריאה ופענוח פקודה קצר יותר. תואם את חוק *Ahmdal*, אך דורש מפקודות מסובכות להתבצע ע"י הקומפיילר – דורש תחכום רב יותר בתוכנה.

כיום שתי שיטות אלו רצות בערך באותה יעילות, ואף *x86* (*CISC*) רצה מהר יותר.

חוק מור:

טען כי מספר הטרנזיסטורים שניתן להכניס ביחידת עיבוד קבועה גדל פי 2 מידי 1.5 שנים. חוק זה אכן התקיים זמן רב , אך גידול כמות הטרנזיסטורים נבלם בשל בעיית ה-*power wall*, שהיא חימום יתר של הטרנזיסטורים הגורם לשריפתם

גודל הטרנזיסטור משפיע על מהירותו– קטן יותר הוא מהיר יותר, ולכן גידול כמות הטרנזיסטורים מביא להגדלת מהירות המעבדים

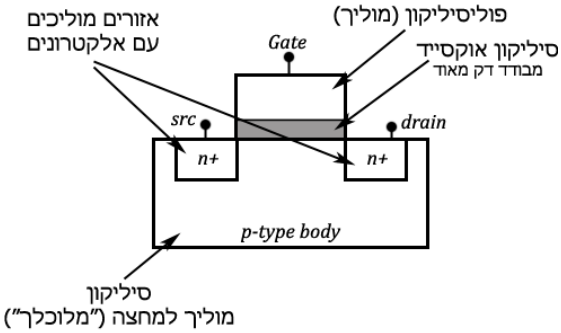
לעומת קצב גידול מהירות המעבד הגבוה , קצב גידול מהירות ה גישה לזיכרון נמוך (9% בשנה) וכדי להתגבר על בעיה זו התפתח תחום ה- *cache*, להגברת מהירות העברת מידע מהזיכרון למעבד.

טרנזיסטורים:

CPU: פיסת סיליקון $1.5 \times 1.5cm^2$ עליה מסודרים מיליוני טרנזיסטורים.

סיליקון: יסוד (*Si*) המתגבש בצורת סריג. על פיסת סיליקון עגולה גדולה המסנה *wafer* מייצרים עותקים רבים של שבבים.

כיצד עובד הטרנסיסטור (יוסבר ל-*nMOS*):

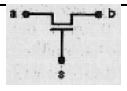
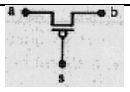


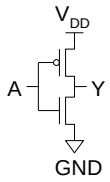
- ה-*src* וה-*drain* מלוכלכים במטען שלילי (ריבוי אלקטרונים), לעומת שאר גוף הטרנזיסטור המלוכלך מטען חיובי (חורים).
- הסיליקון אוקסייד מבודד בין ה-*gate* העשוי פוליסיליקון (מוליך) לגוף.
- כאשר ב-*gate* זורם מתח **חיובי**, אלקטרונים נמשכים (אך לא עוברים , בגלל הבידוד) ל-*gate*, וכך נוצרת תעלה בין ה- *src* ל-*drain* המאפשרת מעבר של אלקטרונים.
- ע"מ לעצור את הזרם (לסגור את המתג) עלינו לפרוק את הפוליסיליקון.

שיטה זו לייצור טרנז' נקראת *MOS*. סוג ה-*IC* (*integrated circuit*) העובד עם שני סוגי הטרנז' *nMOS* ו-*pMOS* נקרא ***cMOS***.

נסמן את שני סוגי המתח:

- $V_{DD} = 1$: מתח גבוה.
- $V_{SS} = 0$: מתח נמוך.

MOSFET	מתי מעביר (מתי השער סגור)	מה מעביר	סימון
nMOSFET	כאשר מקבל 1	0	
pMOSFET	כאשר מקבל 0	1	



דוגמא למימוש מעגל לוגי *cMOS inverter*:

מקבל מ-A והפלט ל-Y: אם מקבל 0, ה-*pMOSFET* סגור וה-*nMOSFET* פתוח, ולכן מועבר 1. אם מקבל 1, המצב בדיוק הפוך ולכן מועבר 0.

שיעור 1: ייצוג מספרים:

אלגברה בוליאנית: אלגברה המושתתת על שימוש ב-0 ו-1 בלבד. שימוש במחשבים: כל הפעולות האריתמטיות והלוגיות הן בייצוג זה.

מעבר בין בסיסים:

- מעבר לייצוג אוקטלי או הקסדצימלי פשוט: עבור אוקטלי נקבץ כל 3 ספרות בינאריות; עבור הקס': כל 4 ספרות.
- מעבר לדצימלי: הספרה במיקום ה-i מייצגת "כפול 2^i" למספר הדצימלי. למשל: $1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 1 * 2^{-1} = 10.11$
- מעבר מדצימלי לבסיס אחר r: כל פעם נחלק ב-r. השארית תהיה ה-*MSB*, ואת המנה נחלק ב-r: השארית תהיה ביט אחד ימינה , וחוזר חלילה עד שהמנה מתאפסת.
- מעבר משבר עשרוני לבסיס אחר: אותו דבר רק הכפלה ב-r במקום חלוקה ב-r.

LSB: הספרה הימנית ביותר (*least significant bit*) ; ***MSB***: הספרה השמאלית ביותר.

נשים לב כי הוספת 0 מימין משמעה הכפלה ב-2 ; הוספת 1 מימין משמעה הכפלה ב-2 והוספת 1 ; ניתן להסיק לבד על הורדת ביט.

חיבור מספרים בינאריים:

מחברים כל ביט במספר אחד לביט המתאים לו במספר השני, תוך התחשבות ב-*carry*.

מספרים שליליים:

ה-*MSB* ישמש לביט סימן.

<u>שיטת המשלים ל-1</u>: פשוט הופכים את כל הביטים, למשל $01101 \leftarrow 10010$.

<u>שיטת המשלים ל-2</u>: כמו שיטת המשלים ל-1, רק שמחברים בסוף 1 לתוצאה. למשל: $01101 \leftarrow 10010$. $10011 \leftarrow 10010$. שיטה זו טובה גם למעבר משלילי לחיובי (היפוך ביטים וחיבור 1). בשיטה זו נוכל לייצג חיסור מספרים ע"י חיבור הראשון עם שלילי של השני.

כמות המספרים שניתן לייצג ב-n ספרות:

- *unsigned* (בלי ביט סימן): $0, \dots, 2^n - 1$.
- *signed* (עם ביט סימן): $-2^{n-1}, \dots, 0, \dots, 2^{n-1} - 1$.

Sign extend:

כדי להפוך מספר מ-8 ביטים ל-16 ביטים "נמרח" את ה-*MSB* שלו שמאלה. למשל: $1011 \leftarrow 1111\ 1011$. כך יישמר ערכו וסימנו.

חיסור בעזרת משלים ל-2:

כאשר מבצעים חיסור, כמתואר קודם, אין חשש לגלישה (*overflow*), ולכן נתעלם מה-*carry* האחרון, לא משנה מה הוא יוצא.

overflow:

כאשר מחברים שני מספרים חיוביים או שני מספרים שליליים , יש חשש לגלישה. נסתכל על שני ה-*carry* האחרונים: אם הם זהים, הפלט תקין. אם הם שונים אזי יש גלישה והפלט לא תקין. עבור מספר בן n ביטים (כולל סימן), התשובה להאם יש *overflow* היא $C_{n-1} \oplus C_{n-2}$.

<u>לא תהיה התייחסות לכל הקטע על ה-*FA* וה-*HA* שעשה בשיעור</u>.

הורדת ביטים (שיכון מספר בן הרבה ביטים במספר בן פחות ביטים):

- הורדת ביטים מצד ימין (הורדת *LSBs* והשארת *MSBs*): אובדן רזולוציה.
- הורדת ביטים מצד שמאל: אובדן סקאלה.

<u>שיעור 2: אלגברה בוליאנית</u>:

משתנים לוגיים:

- מסומנים כמשתנים רגילים: x, y, z.
- תיאור פונקציות בלואניות ע"י טבלאות אמת.

AND: מתנהג כמו כפל; ***OR***: מתנהג כמו חיבור; ***NOT***: משלים.

שערים לוגיים:

אבני הבניין של פונקציות בוליאניות.

משפטי יסוד ומשפטים חשובים:

$X(\bar{X}+Y) = XY$	$X + \bar{X}Y = X + Y$	$X + X = X \cdot X = X$	$\bar{\bar{X}} = X$
$X(Y+Z) = XY + XZ$	$X + YZ = (X+Y)(X+Z)$	$X \cdot 1 = X$	$X + 0 = X$
$XY + \bar{X}Z + YZ = XY + \bar{X}Z$	$XY + \bar{X}Z = (\bar{X}+Y)(X+Z)$	$X \cdot 0 = 0$	$X + 1 = 1$
$(X+Y)(X+\bar{Y}) = X$	$XY + X\bar{Y} = X$	$X \cdot \bar{X} = 0$	$X + \bar{X} = 1$
	$(X+Y)(\bar{X}+Z)(Y+Z) = (X+Y)(\bar{X}+Z)$	$X(X+Y) = X$	$X + XY = X$
קדימויות: *NOT*, אח"כ *AND*, אח"כ *OR*		$\overline{X \cdot Y} = \bar{X} + \bar{Y}, \quad \overline{X+Y} = \bar{X} \cdot \bar{Y}$	*DeMorgan:*

צורה קנונית:

כל פונקציה בוליאנית יכולה להיות מוצגת ע"י אחת משתי הצורות הקנוניות:

- <u>סכום מכפלות (*SOP*)</u>: נסמן: m_i הוא האיבר ה-i בסכום. ניקח את m_i להיות מורכב ממכפלת המשתנים או משלימיהם, כך שהאיבר יהיה 1. למשל עבור $x, y, z = 1,0,1$ נקח את: $xy'z$. נשים בסכום את כל m_i שהפונקציה מחזירה 1. למשל אם $f(101) = 1$ נשים את הדוגמא לעיל בסכום, אחרת לא נשים אותה.
- <u>מכפלת סכומים (*POS*)</u>: M_i הוא האיבר ה-i במכפלה. ניקח אותו להיות מורכב מסכום המשתנים או משלימיהם כך שהאיבר יהיה 0. למשל עבור $x, y, z = 1,0,1$ נקח את $(x' + y + z')$. נשים במכפלה את כל M_i שהפונ' מחזירה עליהם 0. למשל אם $f(101) = 1$, **לא** נשים את הדוגמא לעיל במכפלה, אחרת נשים אותה.

המרה בין צורות: הצגה כסכום מכפלות תומר למכפלת סכומים ע"י פשוט לקיחת האינדקסים שלא הופיעו בראשון. למשל: ס(1,2) יומר למ(0,3).

מערכות שלמות (*Universal systems*):

כל פונקציה בוליאנית ניתנת למימוש ע"י שימוש באופרטורים: *NOT, AND, OR*. לפיכך, כל קבוצת אופרטורים שבאמצעותם ניתן לממש פעולות אלו היא מערכת שלמה, וניתן באמצעותה לממש כל פונקציה בוליאנית. הוכחת שלמות מערכת היא ע"י מימוש שלושת האופרטורים הללו. דוגמאות:

- *NOT, OR*.
- *NOT, AND*.

מספיק אם כן להראות על מערכת שמסוגלת לממש את אחת משתי הדוגמאות. למשל: *NAND* היא מערכת, כמו גם *NOR*. אם כן מספיק להראות על מערכת שהיא מסוגלת לממש אחת מהמערכות שהוצגו לעיל

NAND מתאים להצגת סכום מכפלות ו-*NOR* מתאים להצגת מכפלת סכומים.

פישוט פונקציות ע"י מפות קרנו:

- במפת קרנו כל תא יופרד מתא צמוד לו (לא אלכסון) בביט אחד בלבד.
- כדי לפשט את הפונקציה ל-*SOP* נחפש להקיף ריבועים "מוכללים" של "1", נקח להם את הביטים המשותפים הקבועים. למשל עבור שני תאים **10**0, **10**1 (*xyz*) ניקח את xy'.
- כדי לפשט את הפונקציה ל-*POS* נחפש להקיף ריבועים "מוכללים" של "0", ונקח בהתאם. למשל עבור הדוגמא לעיל נקח את $(x' + y)$ - לשים לב שלוקחים את המשלים של הביטים המשותפים, עבור $x = 1$ לקחנו x' ועבור $y = 0$ לקחנו את y, בניגוד לקודם.

don't cares: "ג'וקר", ניתן להמירו לשימוש הנוח ביותר עבורנו בעתפישוט באמצעות מפת קרנו, ולאו דווקא בעקביות באותו פישוט (יכול לשמש 1 בהצגה אחת ו-0 באחרת).

שיעור 3: לוגיקה צירופית:

שלבי מימוש מעגל צירופי לוגי:

1. תיאור הבעיה.
2. קביעת מספר משתני הכניסה הקיימים ומשתני היציאה הנדרשים.
3. התאמת סמלים למשתנים אלו.
4. בניית טבלת אמת המגדירה את היחסים הנדרשים בין הכניסות ליציאות
5. פישוט הפונ' הבוליאנית עבור כל יציאה.
6. "קיבוץ" ופישוט הפונ' הכוללת של כל הפונקציות לכל היציאות
7. תיאור וכתיבת הדיאגרמה הלוגית.

מחברים:

- *HA*: מחבר שני ביטים, מוציא *carry* ו-*sum*.
- *FA*: מחבר שני ביטים + *carry* מהחיבור הקודם, מוציא גם *s,c*.

מפענחים (*Decoders*):

רכיב הבורר עבור כל קלט את אחת היציאות. ל-*DEC* יכולה להיות כניסת *ENABLE* כדי לאפשר שרשור מספר *DECS*.

למפענח יש n כניסות ראשיות, 2^n יציאות ראשיות. למקודד (*ENC*) יש 2^n כניסות ו-n יציאות.

***MUX (Multiplexer)*:**

רכיב עם 2^n כניסות, n קוי בקרה לברירת הכניסות ויציאה אחת.

מימוש פונקציה בעזרת *MUX* בצורה ישירה: 2^n קוי הכניסה יהיו פלטי הפונקציה בסדר עולה. n קוי הבקרה יהיו קלט הפונקציה (2^n קלטים אפשריים), אשר יבררו בין הכניסות השונות (אילו מהן תצא כפלט).

מימוש חסכוני יותר: חלק ממשתני הפונ' ילכו לקוי הבקרה, וחלק ליצירת הכניסות.

* הערה: שיעור 3 עד סוף החלק הראשון של הסיכומים לוקה בחסר, לא להסתמך עליו.

חלק שני (פרופ' יהודה אפק)

הקדמה:

- שפה עילית: קלה לתכנות, לא חד חד ערכי לשפת מכונה, תלוי קומפיילר, *portable* – לא תלוי מכונה.
- שפת מכונה (*Assembly*): מעבר חד חד ערכי לשפת מכונה אך נוחה לקריאה, תלוית מכונה (למשל *x86* וכו').

***MIPS instruction set*:**

שפה זו היא מסוג *RISC* – פעולות מועטות ופשוטות. לכל שפה *ISA* – תיאור השפה.

פקודה מורכבת מ-32 ביטים המחולקים כך:

פקודות המתבצעות על אוגרים: ***R-Format*** *(add, mul...)*

6 bits	*5 bits*	*5 bits*	*5 bits*	*5 bits*	*6 bits*
opcode	*rs*	*rt*	*rd*	*shamt*	*funct*
זהות הפקודה	רגיסטרים			שדות נוספים של הפונקציה	

פקודות המתבצעות עם קבועים: ***I-Format*** *(lw, sw, branch)*

opcode	*rs*	*rt*	*16 bits address*

קפיצה ישירה לכתובת / מקום בתוכנית: ***JUMP***

opcode	*26 bits address*

פענוח פקודה:

- *fetch*: הבאת הפקודה מהזיכרון. כל פקודה יושבת בזיכרון וישנו מצביע של ה-*CPU*, ה-*PC* (*program counter*) המצביע על השורה בה נמצאים בתוכנית, וכל פעם מתקדם לשורה– כלומר לפקודה הבאה.
- פענוח הפקודה: 6 הביטים הראשונים הם הפקודה עצמה, למשל *add*.
- ביצוע: הוצאת האופרנדים, ביצוע הפעולה והשמת התוצאה ברגיסטר היעד.
- הבאת הפקודה הבאה.

הזיכרון:

אוסף בתים (*bytes = 8 bits*) המאורגן בשורות, כאשר כל שורה היא *byte*, כל 4 שורות (4 בתים) היא מילה, כלומר *word = 32 bits*. מכאן שכל פקודה תופסת מילה, וכל התקדמות בסוף פקודה לפקודה הבאה בעצם מקפיצה את ה-*PC* ב-4 *bytes*.

דוגמאות:

- *lw* $1 32($2): הולך לזיכרון במקום *$2* (הכתובת המאוחסנת ברגיסטר *$2*), קופץ 32 ***bytes*** קדימה = 8 ***words*** קדימה, מחזיר את **המילה** (32 ביטים) המתחילה ממקום זה ושם אותה בתוך רגיסטר *$1*.

  לא בטוח שהתוכן החדש שטענו לרגיסטר *$1* יהיה נראה לפקודה הבאה (שאליה קוראים עם סיום הפקודה).
- *beq* $1 $2 25: משווה את תכני הרגיסטרים *$2 ,$1*, ואם הם שווים: קופץ ב-*PC* לכתובת (ב-*bytes*) $PC + (1 + 25) \cdot 4$ - כלומר, הוא בכל מקרה מקדם את ה-*PC* בפקודה (מילה) אחת קדימה, ועליה עוד מוסיף 25 מילים, כלומר סה"כ יקפוץ 104 *bytes* (בתים) קדימה.

נשים לב: לכתובות *data* כמו *lw* נשתמש ב-*offset* של *bytes* לעומת כתובות *code* כמו *branch* בהן משתמשים ב-*offset* של *words*.

שימוש אוגרים: ישנה קונבנציה של שימוש באוגרים השונים במערכת: $\$zero, \$v0 - \$v1, \$a0 - \$a3, \ldots$.

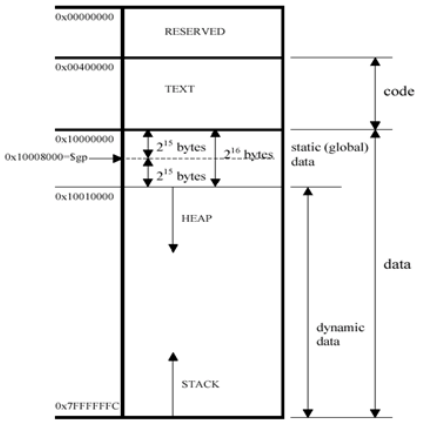
ישנן פקודות *pseudo-instructions* שאינן פקודות בסיסיות אלא מורכבות מכמה פקודות בסיסיות, אך ה-*assembler* יודע לתרגמן. למשל: *slt*.

קומפיילר:

בעת תהליך הקימפול מתבצע לינק בין תוכניות שונות: תחילה כל תוכנית מתקמפלת כאילו מתחילה מכתובת 0. לאחר מכן הלינקר דואג בשר שור התוכניות שכל הפקודות שהזזתן רלוונטית ישתנו בהתאם: *jump* למשל, הקוץ לכתובת אבסולוטית יצטרך להשתנות , אך *branch* הקופץ באופן יחסי לפקודה בה נמצא עכשיו, לא יצטרך שינוי. נשים לב: שינוי מתבצע בפקודות, לא בניהול *data* (האוגרים).

ניהול זיכרון:

הזיכרון מתנהל ב-*heap*:



- החלק העליון ביותר: שמור.
- החלק העליון מחזיק את קוד התוכנית.
- החלק התחתון מחזיק את ה-*data*:
  - תחילה יוחזקו משתנים גלובלים
  - מתחת יוחזק ה-*main* של התוכנית שלנו.
  - מתחת יוחזקו האוגרים.

MACRO:

הגדרת פקודות מקרו לשימוש אישי בתוכניות שלנו. למשל:

```
.macro  my_macro($arg)
<command>
<command>...
.end_macro
```

בשלב ה-*preprocessing* יוחלף המקרו בתוכנית עם הפקודות האמיתיות.

מה קורה בזמן ריצה:

נניח רוצים לקרוא לפרוצדורה *sub1* מתוך *main*. להלן הליך ריצה תקין של תוכנית:

1. הקצאת מקום לארגומנטים שנשלחים מ-*main* על ה-*stack*: **`addi $sp $sp -16`** ושמירתם על המקום שהוקצה: **`sw...`** בדוגמא זו מקצים מקום ל-4 מילים על ה-*stack*. נציין שחלק מהארגומנטים יכולים להיות מועברים דרך רגיסטרים *$a0-$a3*, אך אם צריך להעביר עוד, ה-*main* (הפרוצדורה הקוראת) דואגת לזה.
   - *$sp*: 4 מילים למעלה; מצביע על תחילת הארגומנט האחרון שהוכנס (זה עם ערך הכתובת **הנמוך** ביותר).
   - *$fp*: ה-*frame pointer* של *main*.
2. קריאה ב-*main*: **`jal sub1`** *(jump and link)* דואג לשים ברגיסטר *$ra (return address)* את הפקודה **הבאה** ב-*main*, כך שכשיסיים את ריצת *sub1*, יחזור כבר לפקודה הבאה שיש לבצע ב-*main*.
   - *$sp*: לא השתנה.
   - *$fp*: לא השתנה.

➔ נכנסים ל-*sub1*

3. ב-*sub1*: הקצאת מקום על המחסנית ל-*$fp*, *$ra*: **`addi $sp $sp -8`** ושמירת *$ra, $fp* (*$ra* בכתובת גבוהה יותר) עם כניסה לפרוצדורה *sub1* נקצה מקום על ה-*stack* ל-*$ra* (שעלול להשתנות ע"י קריאות *jal* בתוך *sub1*) ול-*$fp* של *main* כדי שנוכל לעדכן את *$fp* להיות ה-*frame pointer* של הפרוצדורה אליה נכנסנו, *sub1*.
   - *$sp*: מצביע על היכן ששמרנו את *$fp* הרגע.
   - *$fp*: עדיין מצביע על *$fp* של *main*.
4. ב-*sub1*: עדכון *$fp* להיות *$fp* של *sub1*: **`add $fp $sp $zero`**
   - *$sp*: מצביע לאותו מקום (היכן ששמרנו את *$fp* של *main*).
   - *$fp*: מצביע כעת ל "תחילת" המחסנית המקומית של *sub1*: מתחתיו (כתובות גבוהות): *$fp* של *main*, *$ra* ששמרנו, הארגומנטים שנשמרו ע"י *main* עבור *sub1*. מעליו (כתובות נמוכות) יבואו משתנים לוקאלים של *sub1*.
5. ב-*sub1*: הקצאת מקום על ה-*stack* עבור משתנים לוקאלים (נניח 2 מילים): **`addi $sp $sp -8`**

לאחר מכן תבוא שמירת משתנים לוקאלים. נציין שאם אנו רוצים להשתמש ברגיסטרים לא זמניים, כמו *$s0-$s3*, נצטרך להקצות להם מקום על ה-*stack* לפני שנשתמש בהם, ובסוף הפרוצדורה לטעון אותם חזרה.

- *$sp*: מצביע לכתובת של המשתנה האחרון הלוקאלי שהוכנס (זה עם הכתובת הנמוכה ביותר).
- *$fp*: מצביע לאותו מקום: נקודת המעבר בין משתנים לוקאלים לכל מה שלפני. גישה למשתנים לוקאליים: *offset* שלילי מה-*$fp*, וגישה למשתנים שקיבלנו מהקוראת: *offset* חיובי, ולשים לב לדלג על 2 מילים (ה-*$ra* וה-*$fp* של *main*).

6. סיום ריצה ב-*sub1*:
   - טוענים את כל הרגיסטרים שאנו רוצים להחזיר (אלה ששמרנו קודם על ה-*stack*).
   - שחרור משתנים מקומיים ע"י: *add $sp $fp $zero* ("*pop*" חזרה ל-*$sp*).
   - טוענים את *$ra* ואת *$fp*, ומשחררים את *$sp* בעוד 8 (2 המילים שהחזיקו את *$ra, $fp*).
   - ***jr $ra***: קופצים חזרה ל-*main*.

⬅ חוזרים ל-*main*

7. ב-*main*: שחרור הארגומנטים מה-*stack*: ***addi $sp $sp 16***
   - *$sp*: מצביע על ראש המחסנית לפי *main* (כנראה למשתנה הלוקאלי של *main* השמור בכתובת הנמוכה ביותר).
   - *$fp*: מצביע ל-*frame* של *main* (הכל חזר לקדמותו).

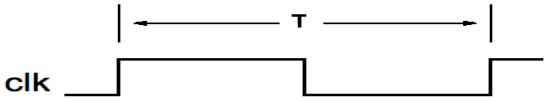
Single Cycle Architecture:

מעבד *MIPS* פשוט המבצע את כל הפעולות ב-*cycle* אחד של השעון (דורש אורך זמן *cycle* יחיד ארוך, מאט עבודה):

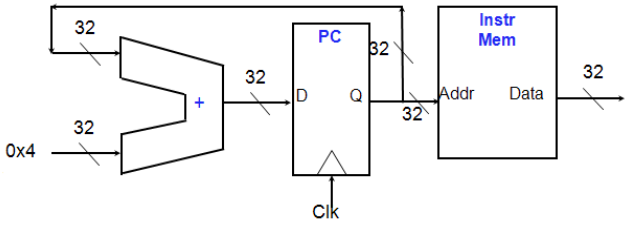
- *Fetch*: הבאת הפקודה.
- *Decode*: פענוח הפקודה, הוצאת משתנים מרגיסטרים.
- *Execute*: ביצוע.
- *Memory*: גישה לזיכרון.
- *write-back*.

מחזור שעון ועבודת *D-FF*:

בעליה של השעון: נקרא את הערך *D* ונשים אותו ב-*Q*. *Q* יחזיק בערך זה לכל אורך מחזור השעון (עד כולל ירידת השעון, ועד עליית השעון הבאה).
כתיבה תתבצע בירידת שעון.



פעולות *R-type*:



מוחזק זיכרון נפרד ל- *Data* ול-*Instructions*. אנו מניחים שבתחילת הריצה כל הפקודות (התוכנית) כבר טעונות ל-*inst. memory*.

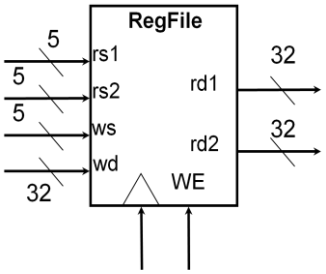
PC: ה-*program counter* מורכב מ-32 *D-FF* המחזיקים יחד 32 ביטים של כתובת זיכרון – הכתובת אליה צריך ללכת ב-*inst. memory*.

כל *CLK* תצא הכתובת הבאה שיש לקחת אל ה- *inst. memory*, ובנוסף הכתובת תיכנס למחבר 32ביט עם 4 ונותן ערך חדש ל-*PC* – מעלה אותו ב-4, עדכון לכתובת הבאה שנרצה לקחת מה-*inst. memory*.

inst. memory: בהינתן הכתובת מה-*PC*, מוציא את הפקודה היושבת אצלו בכתובת זו. פקודות הן בגודל מילה, לכן קופצים +4 כל קריאה.
סימון של חץ עם קו עליו ו-32 משמעו: מעבר 32 חוטים במקביל.

ALU: כמו המחבר הציור רק מקבל גם בקר *op*, ויש לו יציאת *zero?* בנוסף ליציאת התוצאה (עבור פעולות לוגיות).

רגיסטרים:



rs1, rs2: בחירת רגסטר ראשון ורגיסטר שני להוציא מהם מידע

rd1, rd2: יציאות אותם רגיסטרים שנבחרו.

ws (*write select*): כניסה לבחירת הרגיסטר אליו רוצים לכתוב

we (*write enable*): כניסה לבחירה האם רוצים לכתוב לרגיסטר ב-*ws*. יתבצע בעלית השעון הבאה.

wd (*write data*): כניסה לבחירת הערך שרוצים לשים ב-*ws*.

ישנם סה"כ 32 רגיסטרים.

כל הנ"ל ממומשים ב-*MUX*ים, כאשר הרגיסטר הראשון, הוא *$zero*, היחיד שלא מחובר לבקרות הכתיבה (שכן ערכו קבוע – 0).

ביצוע:

היציאות יחוברו ל-*ALU* עם *op* כלשהו, ויציאת ה-*ALU* תתחבר ל-*wd* כדי שיוכל להיכנס במחזור השעון הבא לאחד הרגיסטרים כנדרש.

לשים לב: כשנתרגם פקודה למימוש במערכת *rs, rt* הם הרגיסטרים שקוראים, ו-*rd* הוא רגיסטר היעד (זה שיתחבר ל-*ws* לצורך העניין).

היציאות מה-*inst. memory* יתחלקו: החלק של הרגיסטרים ילך לרגיסטרים, ושאר החלקים:

- *opcode* (ביטים *31-26*): ילך לבקר *LOGIC* כלשהו שיהיה אחראי על העברת ה-*we*: ה-*enable* של כתיבה לרגיסטר.
- *funct* (ביטים *5-0*): ילא ל-*ALU-OP*: איזו פקודה תתבצע.

פעולות *I-type*:

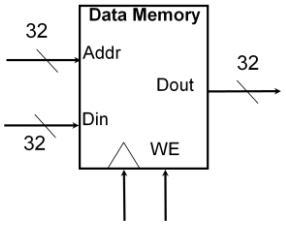
השינויים מ-*R-type*:

- קודם רגיסטר היעד היה ***rd*** (ביטים 11-15) וכעת הוא ***rt*** (ביטים 16-20).
- ה-*immediate* שמקבלים הוא ה-16 ביטים, ויש לעשות לו *sign-extend*: או ע"י מריחת אפסים או שיטת המשלים ל-2: מריחת 1ים.

משתמשים ב-*MUX* לבקרה על סוג הפעולה ושליחת הפרמטרים הנכונים:

- *MUX* לבקרת הכניסה ל-*ws* (לאיזה רגיסטר הולכים לכתוב): *rt* (*I-type*) או *rd* (*R-type*).
- *MUX* לבקרת הכניסה לאחד האופרנדים של ה-*ALU*: *sign extend* של *immediate* (*I-type*) או *rd2* (*R-type*).

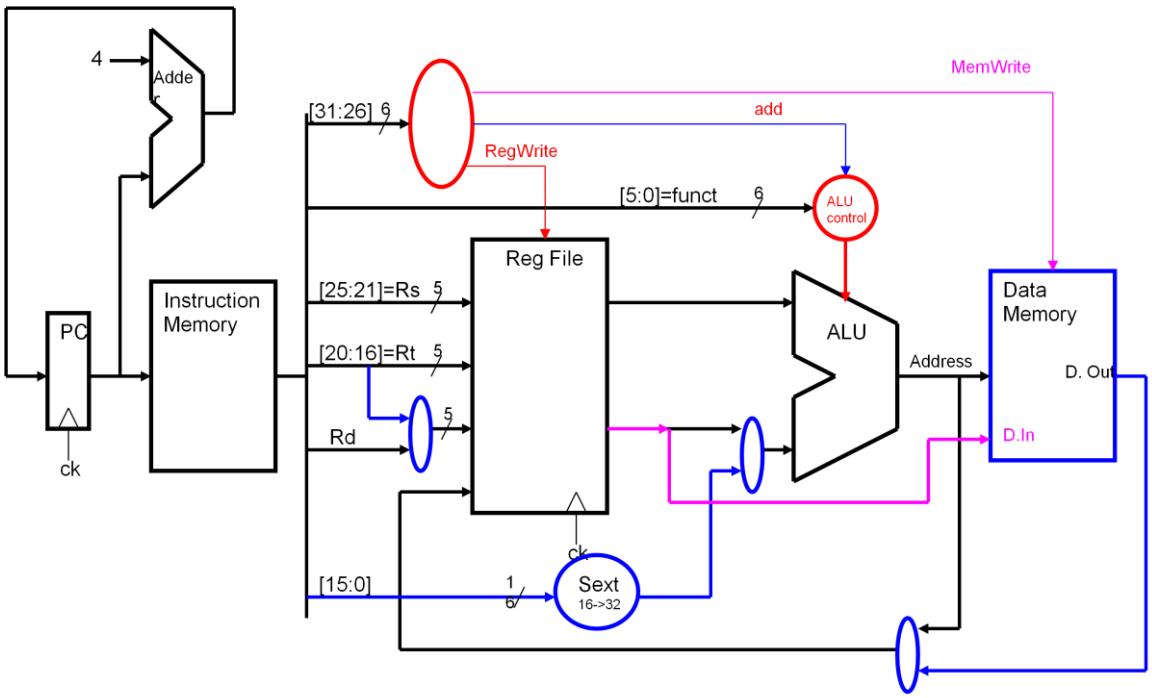
Load / Store word:



lw: *rs* מחזיק את כתובת המערך, *immediate* מחזיק את ה-*offset* (יזדקק ל-*sign-extend* שיהיה מריחת 0), *rt* מחזיק את הרגיסטר אליו רוצים לכתוב. *rs* ו-*imm* יחושבו ב-*ALU* ויכנסו ל-*Address*. מ-*D.out* יצא אל ה-*wd* (*data*) לרגיסטר אליו רוצים לכתוב, *rt*) דרך *MUX* הבורר בין *lw* ובין החזרת תוצאה מה-*ALU* (למשל *add*).

קריאה לכתובת מה-*data mem* תגרור הוצאת תוכן מייידי.

sw: *rs* יחזיק את כתובת המערך, *imm* את ה-*offset* ו-*rt* את התוכן שרוצים לכתוב לאותו מקום ב-*data mem* הנכנס ל-*Din*. ה-*WE* ב-*data mem* מקבל 1, וב-*reg* מן הסתם 0 (לא כותבים כלום לרגיסטרים). *sw* מתוזמנת עם השעון: כתיבה תעשה בעליית שעון (אם *WE* למעלה כמובן). הערה: *din* מתקבל בכל מקרה, אך כתיבה מותנית ב-*WE* של ה-*data mem*.



branch:

- משווה בין *rs* ל-*rt* ומקפיץ / לא מקפיץ את ה-*PC* בהתאם.
- ה-*imm* מחזיק את הקפיצה (היחסית ל-*PC+4* שאליה אמור לקפוץ ללא *branch*) **במילים**, ולא בבתים.
- נשתמש ב-*sign extend* כיוון שיתכן ונרצה קפיצה אחורה ולא קדימה

אלמנטים שנוסיף בשביל ה-*branch*:

- *Adder* שיחבר את ה-*shift-left 2* של ה-*sign extend* של ה-*imm* עם *PC+4* (ה-*shift left*: הכפלה ב-4).
- הוצאת ה-*Zero* מה-*ALU* הראשי ליחידת הבקרה: לבדוק את התוצאה הלוגית של ההשוואה.
- *MUX* הבורר מה ילך ל-*PC*: *PC+4* או *((PC+4+shift-left-2(sign-extend-16->32(imm))* – מבוקר ע"י *AND* של: האם אנחנו בפקודת *branch* + האם ההשוואה הוציאה 1 (מהסעיף הקודם), ולכן יש לבצע *branch*.

jump:

- נקח את *PC+4[31-28]* כ-*MSB*, ואת *26* הביטים שמקבלים ב-*jump* ונעשה להם *shift-left-2* כדי לקבל כתובת בבתים. נשרשר יחד (כך שה-28 הם ה-*LSB*) – זו הכתובת שצריך לשלוח.
- יהיה קו בקרה נוסף ל-*MUX* הבורר בין לקפוץ לכתובת ה-*jump* שזה עתה חושבה ובין האלטרנטיבה (*PC+4* או כתובת אחרת אם יש *branch*).

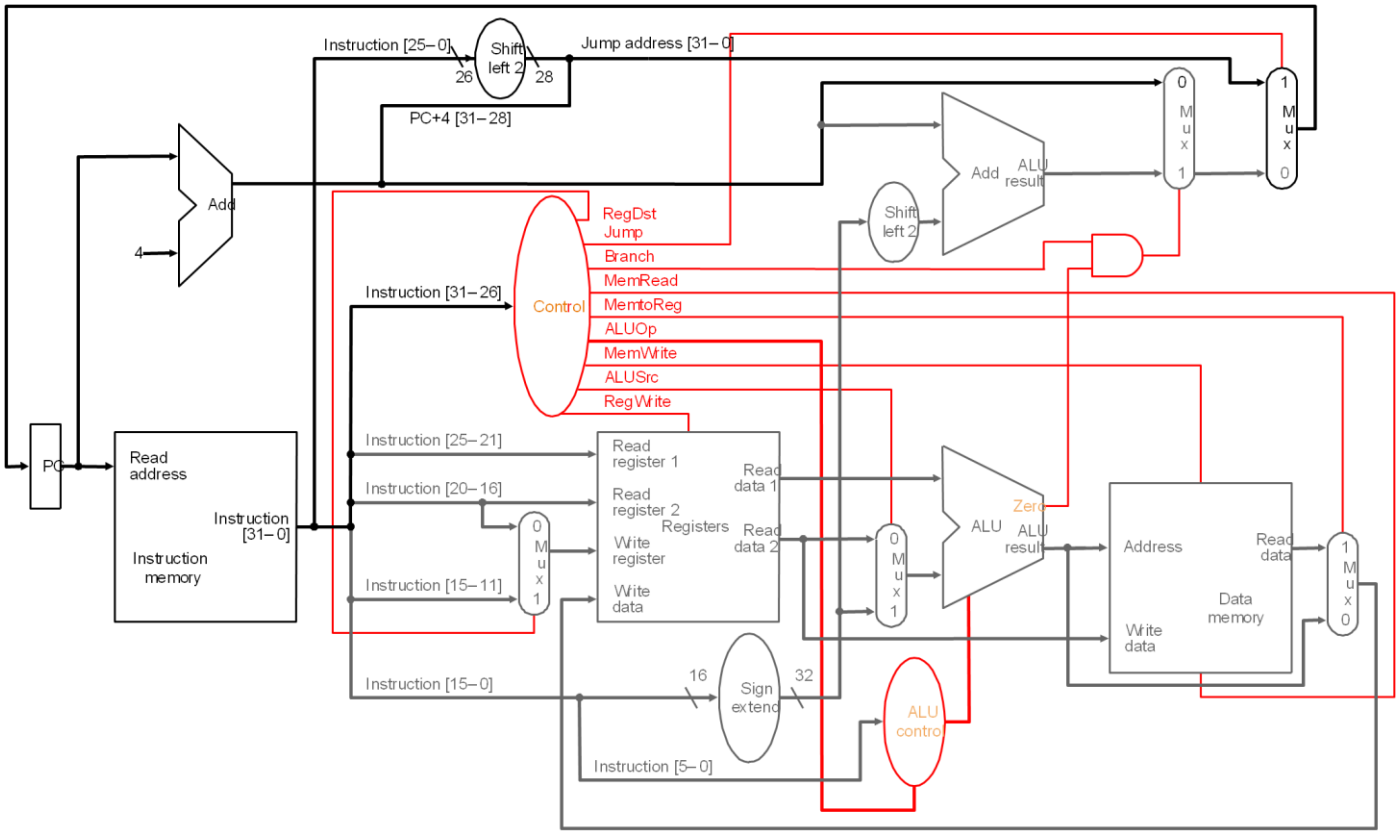
***Control*: קווי הבקרה**:

- *RegDest*: האם רגיסטר היעד הוא *rd* או *rt* (1 ל-*rd*, כלומר ל-*R-Type*).
- *jump*: האם זוהי פעולת *jump*.
- *branch*: האם זוהי פעולת *branch*.
- *memRead*: האם זו פעולת קריאה מהזיכרון (*lw, lb*).
- *memtoReg*: האם זו פעולת קריאה מהזיכרון לרגיסטר (החלופה: קריאה מפלט ה-*ALU*).
- *ALUop*: הולך ל-*ALU control*, שולט בפעולת ה-*ALU*.
- *memWrite*: האם זו פעולת כתיבה לזיכרון (*sw, sb*).
- *ALUSrc*: האם האופרנד התחתון של ה-*ALU* הוא מהרגיסטרים (*rt*) או מה-*imm*.
- *RegWrite*: האם כותבים לרגיסטר (בפעולות *add, lw,...*).

קו ה*zero* היוצא מה-*ALU* הוא בשביל ה-*branch*.

ה-*ALUcontrol*: מקבל *ALUop1, ALUop0, funct field* (*op1, op0*) מתקבלים מה-*control* הראשי).

- *op1,op0=00*: פעולת *lw, sw*.
- *op1,op0=01*: פעולת *branch*
- *op1,op0=10*: פעולה אריתמטית (חיבור חיסור וכו') – *funct field* יהיה אחראי על הגדרת איזו פעולה אריתמטית.



Pipeline Architecture :

בשיטה זו מבצעים פקודות במקביל, כל אחת בשלב אחר בתהליך (כאשר אחת ב-*decode/reg-retrieve*, הבאה כבר ב-*fetch* וכו'). זמן פקודה נשאר אותו דבר, אך ב-*overall* מתבצעות יותר פקודות בפרק זמן, כיוון שלא צריך לחכות עד סיום של אחת כדי להתחיל את הבאה.

מושגים :

- *latency* : הזמן שלוקח לבצע כל משימה (למשל : *fetch, execute*) – לא משתנה מ-*single cycle*.
- *throughput* : כמות העבודה שמתבצעת בפרק זמן מסוים – גדלה משמעותית לעומת *single cycle*. או : זמן ממוצע לפקודה.
- *fill* : זמן מילוי ה-*pipeline* עד שכל שלב מתבצע עבור פקודה כלשהי; *drain* : שלב ריקון ה-*pipeline* בעת סיום התוכנית.

עקרונות :

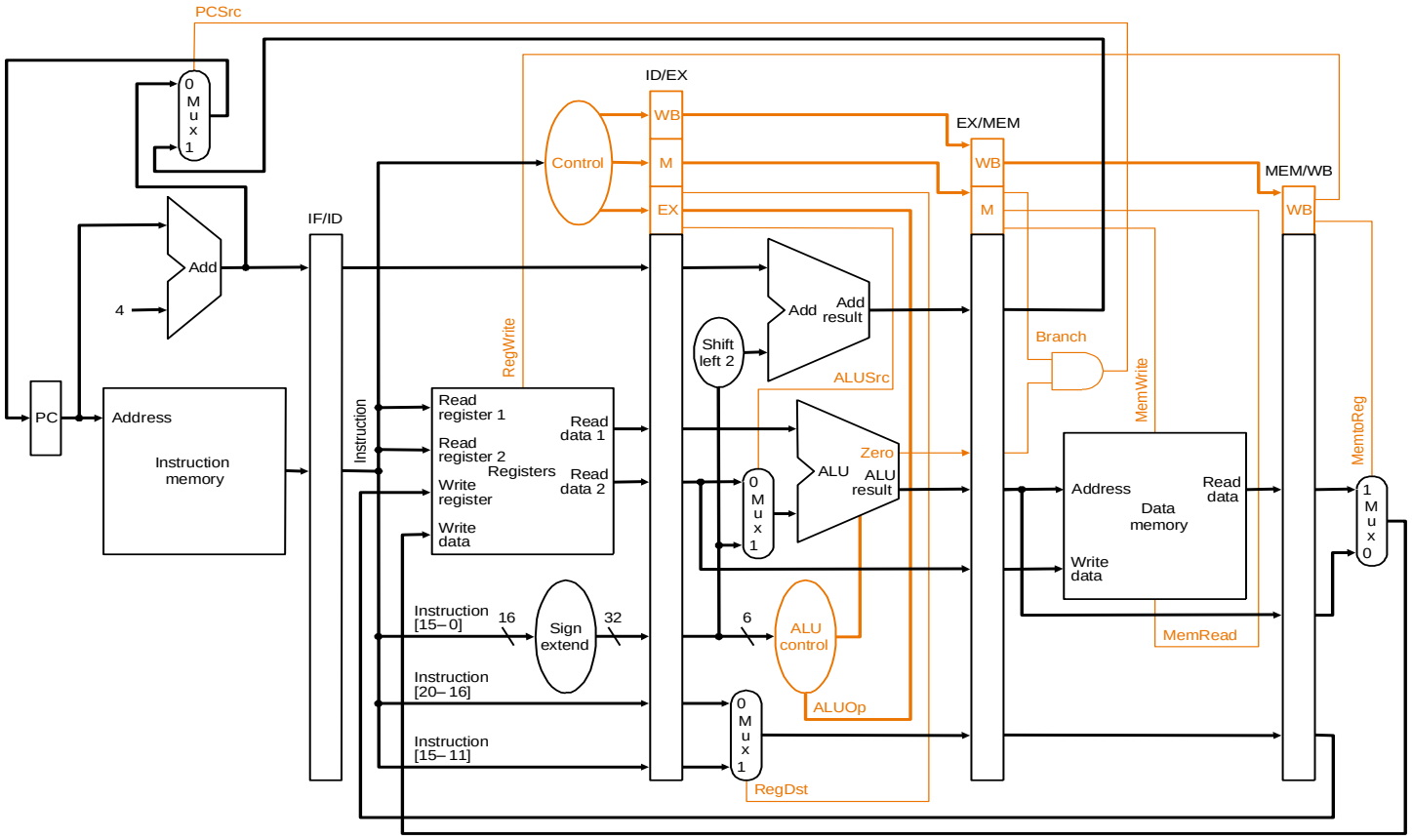
- המהירות הפוטנציאלית תלויה במספר השלבים ב-*pipeline*.
- מהירות כל שלב נקבעת ע"פ מהירות השלב האיטי ביותר בכל ה-*pipeline*. לכן, נשאף לפקודות קצרות ופשוטות, ומשך כל פקודה ופקודה פחות או יותר שווה.

5 השלבים ב-*pipeline* ב-*MIPS* :

1. *IFetch* : הבאת פקודה ובגדלת ה-*PC*.
2. *Decode* : פענוח הפקודה וקריאת הרגיסטרים.
3. *Execute* : פקודות לזיכרון : חישוב כתובת; פקודות אריתמטיות : חישוב הפעולה.
4. *Memory* : קריאה / כתיבה לזיכרון.
5. *Write Back* : כתיבת *Data* לרגיסטר.

כדי לשמר את ערכי הרגיסטרים לכל פקודה בכל שלב, נשים רגיסטרי ביניים בין השלבים: *IF/ID, ID/EX, EX/MEM, MEM/WB* (סה"כ 4).

control : נשמור לכל שלב את קווי הבקרה הרלוונטים ונעביר אותם קדימה כל שלב



סוגי בעיות:

- *structural hazard*: חוסר תמיכה בחומרה בצירוף הפקודות.
- *control hazard*: פקודות כמו *branch* המעכבות את ה-*pipeline* עד חלוף ה-*hazard*.
- *data hazard*: פעולות המסתמכות על תוצאות של פעולות שטרם סיימו לעבור ב-*pipeline* (למשל קריאה מרגיסטר שעובדים עליו).

פתרון הבעיות:

data:

שימוש בפקודות *NOP* שהן פקודות ריקות, כדי ליצור עיכוב עד חלוף הסכנת חסרון: בזבוז מחזורי שעון.

פתרון מוצע: נעביר את תוצאת החישוב כבר עם סיומה, במקום לחכות עד הסוף. למשל עבור *sub*: במקום לחכות שתוצאת החישוב תחזור ותכתב ברגיסטר היעד באופן רגיל, כבר עם סיום החישוב בשלב ה-*EX* נעביר את התוצאה אחורה. כך נוכל לחסוך במקרה זה *NOP* אחד.

פתרון סופי באמצעות *forwarding unit*:

יחידה זו מקבלת:

- רגיסטר היעד - *rd* משלב ה-*EX/MEM*.
- רגיסטר היעד - *rd* משלב ה-*MEM/WB*.
- קו הבקרה *RegWrite* משלב ה-*EX/MEM*: כלומר האם *rd* אמור להתעדכן, ולכן עלינו להיזהר משימוש בו הפקודות חדשות יותר.
- קו הבקרה *RegWrite* משלב ה-*MEM/WB*: כנ"ל.
- רגיסטר *rs* לפקודה הנמצאת כעת בשלב ה-*EX*.
- רגיסטר *rt* לפקודה הנמצאת כעת בשלב ה-*EX*.

לכניסות ה-*ALU* נחבר שני *MUX*:

- *MUX* לכניסה העליונה, שכעת תקבל בנוסף ל-*ID/EX.rs* את *EX/MEM.rd* ואת *MEM/WB.rd*.
- *MUX* לכניסה התחתונה שכעת תקבל בנוסף ל-*ID/EX.rt* את ...

ה-*forwarding unit* בודקת האם יש שוויון בין ה-*ID/EX.rs*, *ID/ES.rt* לאחד ה-*rd* מהשלבים המתקדמים יותר, ואם כן תברור ב-*MUX*ים לקחת את הערכים ה**מעודכנים**, דהיינו את ה-*rd* המתאים.

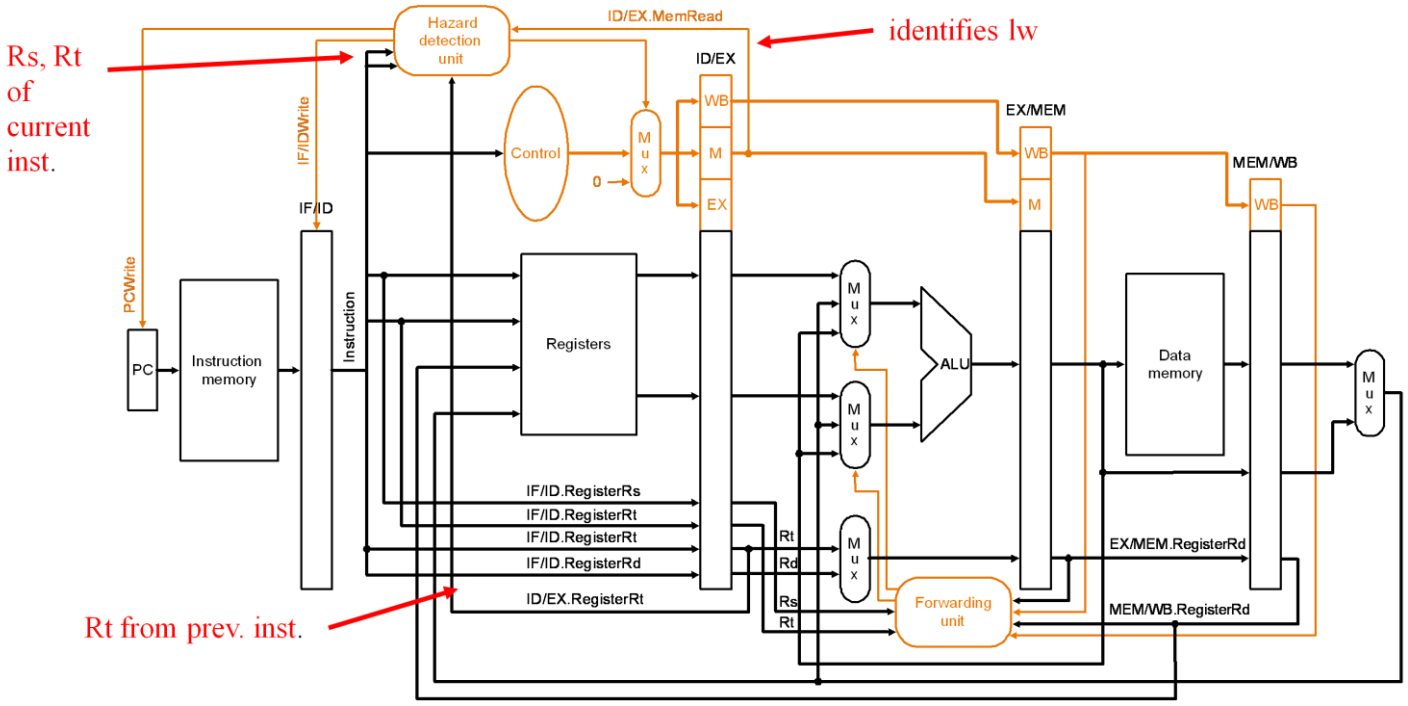
כך הורדנו את שני ה-*NOP* הנוספים וייעלנו את המערכת.

פתרון זה לא תמיד עובד:

למשל עבור *lw*, שם הערך ה"מוכן" איתו אמורים להשתמש (של *rt* הפעם, לא *rd*), מוכן רק בשלב ה-*WB* (נבדק שזו פקודת *lw* ע"י *memRead=1*).

במקרה זה נצטרך להשתמש ב-*NOP* או ב-*NOP* מובנה בחומרה הקרוי *BUBBLE*:

hazard detection unit המזהה את הבעיתיות ועוצר את כל קידום ה-*PC* וה-*IF*, ומכניס קיפאון בשלב ה-*ID*. זאת נעשה כמובן רק אם יש שוויון בין האוגרים – ה-*rt* שטרם עודכן והאוגר הרלוונטי בו רוצים להשתמש ב-*EX*.



If (ID/EX.MemRd)&& ((ID/EX.Rt= =IF/ID.Rs) || (ID/EX.Rt= =IF/ID.Rt)) we must "stall" the pipeline!

פתרון נוסף: *reordering* לפקודות, נמקם פקודה (אם ניתן) שלא תלויה ב-*lw* מיד אחריה, וכך נמנע מבזבוז מחזור שעון.

לסיכום:

- *hazard detection unit*: מזהה מקרים של *lw* בהם יש צורך ב-*bubble*.
- *forwarding unit*: מזהה מקרים בהם משתמשים במידע שחושב ב-*ALU* ואמור להיות מעודכן, ומעבירה ל-*ALU* את הערכים הרלוונטיים.

branch hazard:

תוצאת ה-*branch* מתבצעת רק בשלב הרביעי ועד אז כבר מתחילות להתבצע שלוש פקודות שלא אמורות להתבצע טיפול:

- נשווה בעזרה מעגל פשוט את *rs* ו-*rt* כבר בשלב ה-*decode*, וכך נחסוך צורך בשניים מתוך שלושת ה-*NOP*.
- *branch-delay slot*: פתרון ל-*NOP* השלישי: מוסכמה שפקודה אחת אחרי ה-*branch* **תמיד** תתקיים, ובמקום *NOP* (*bubble*), המתכנת או הקומפיילר ישימו פקודה זו להיות כזו שלא תלויה בערכי ה-*branch*. ב-50% תהיה פקודה מתאימה, ואם לא – יושם *NOP*. ניתן להשתמש בידע זה כדי לעשות אופטימיזציה לקוד: גם אם יש לופ, ניתן להעביר פקודה מתוך הלופ להיות מיד אחרי ה-*branch* האחראי על קיום הלולאה, שכן היא תתקיים בכל מקרה.

סכנות ב-*pipelining*: למרות ייעול הזמן, עולה הסיכון ל-*hazards* ולצווארי בקבוק.

שיטות נוספות להקפאה:

- *flushing*: איפוס רגיסטר ה-*IF/ID* והשארת ה-*PC* במקומו. שיטה חלופית ל-*branch delay slot*.
- *delayed load*: אותו עקרון של השארת מקום לפקודה לא תלויה עבור *lw*. כך זה ממומש במציאות.

Exceptions: לא יכלל בסיכום (נראה לא חשוב במיוחד).

<u>זיכרון מטמון (*Cache*)</u>:

- נועד לגשר על פער המהירות בין פעולות המעבד(מהיר) וגישה לזיכרון (איטי).
- רמות שונות של *cache*: רמה גבוהה, *static ram* = מהיר ויקר, ככל שיורדים ברמות המחיר יורד וגם המהירות, *dynamic ram* (הדיסק – הכי איטי).
- שני *cache*: *instructions cache* (שלב ה-*IF*) ו-*data-cache* (שלב ה-*MEM*).

RAM (Random Access Memory):

זיכרון שניתן לגשת לכל מקום בו, לא לפי סדר קבוע. המבנה הבסיסי הוא שני *MUX* (*high, low*) המאפשרים בחירה של הכתובת אליה רוצים לגשת לפי שורה ועמודה ברשת הזיכרון. שני הסוגים העיקריים הם:

- *SRAM* (*static ram*): לכל תא שתי יציאות – ערך הביט שמחזיק והמשלים שלו. תא הזיכרון עצמו מחזיק בטריה והרבה חומרה, לכן הוא יקר יותר ותופס יותר מקום, אך שיטת משיכת המידע היא דחיפת אלקי החוצה, לכן יותר מהיר.
- *DRAM* (*dynamic ram*): לקריאה – זרם עובר בתא הזיכרון וחש את השפעת התא עליו; לכתיבה – מעביר את הזרם דרך תא הזיכרון. כל תא מחזיק *capacitor* הדורש רענון מדי פעם כדי שהמידע לא יאבד.

לוקאליות:

- <u>בזמן</u>: אם ניגשים לכתובת מסויימת בזיכרון, סביר שניגש אליה שוב בזמן הקרוב. *I$*: שימוש מרובה בלולאות; *D$*: עדכון נתונים.
- <u>במרחב</u>: אם ניגשנו לכתובת מסוימת, סביר שניגש לכתובות בסביבתה. *I$*: פקודות התוכנית באות ברצף; *D$*: שימוש במערכים רציפים.

לכן: כאשר נביא מידע מהזיכרון נביא בלוק הכולל רצף מילים, ולא רק את הכתובת שאליה נדרשים ברגע זה. לכן גם גישה ראשונית לזיכרון תקח יותר זמן בד"כ מאשר גישות שאחריה – שכן אז חלק גדול מהמידע בו משתמשים (או סביר שנשתמש) כבר נמצא על ה-*cache*.

מושגים מרכזיים:

- <u>*HIT*</u>: כאשר ניגשים לנתון והוא כבר נמצא על ה-*cache*.
- <u>*MISS*</u>: כאשר ניגשים לנתון והוא **לא** נמצא על ה-*cache*. במקרה כזה עלינו להביא את המידע מרמה אחת מעל, איטית יותר, ומעכבים את ה-*pipeline* עד הבאת המידע.
- <u>*BLOCK*</u>: גודל היחידה הבסיסית שמביאים בכל *MISS* ל-*cache*. גודל בלוק יהיה לפחות מילה אחת.
- <u>*Compulsory*</u>: הגישה הראשונה (והאיטית) לזכרון.
- <u>*Capacity*</u>: גודל ה-*cache*, מוגבל.
- <u>*Conflict*</u>: מיפוי הרבה כתובות לכתובת אחת ב-*cache*.
- <u>*Coherence (invalidation)*</u>: כאשר כמה תהליכים משתמשים באותה מילה שעכשיו מתעדכנת. ברגע שכותבים לכתובת מסוימת, הוא מודיע שהמידע אינו עדכני, ונוצים *MISS* בתהליכים שמנסים לגשת.

סוגי זיכרון:

<u>*Direct mapped cache*</u>:

- כל תא בזיכרון ממופה באופן חד ערכי ל-*cache* לפי הכתובת שלו. למשל, כל הכתובות המסתיימות ב-100 יתמפו לתא ה-100 ב-*cache*.
- ה-*cache* יחזיק את המידע: *tag+data* כאשר *data* הוא תוכן אותו מקום מה זיכרון, וה-*tag* הוא ה-*MSB* של הכתובת בזיכרון ממנה קראנו, כדי שנדע למשל מאיזו כתובת שמסתיימת ב-100 קראנו (לצורכי כתיבה חזרה לזיכרון).

סידור הזיכרון:

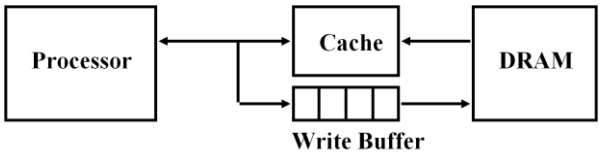
- נסתכל על כתובת בזיכרון: כיוון שמסתכלים על מילים, יש לנו 2 ביטים (*LSB*) של הכתובת שעבורם תמיד ניקח 00 (*byte offset*).
- 10 הביטים ה-*LSB* שאחריהם יהיו כתובת ה-*cache* אליה נמפה את המידע.
- 20 הביטים האחרונים (ה-*MSB*ים) של הכתובת מהזיכרון ישמרו ב-*tag*.
- ביט אחד של *valid* מחזיק 1 אמ"מ ה-*data* בתא *cache* זה מעודכן. *HIT* יבדוק: האם *valid* וגם האם מחזיק את הכתובת הרצויה.
- תא ב-*cache*: $[1\ valid?\ field] + [20\ tag\ field] + [32\ data\ field]$

אופציה נוספת: *tag* בגודל 16 ביטים, גודל כתובת ב-*cache* 14 ביטים (במקום 20 ו-10).

בצורה כללית: n – מספר הביטים של כתובת ב-*cache*; $30 - n$ - מספר הביטים של ה-*tag*.

write through :

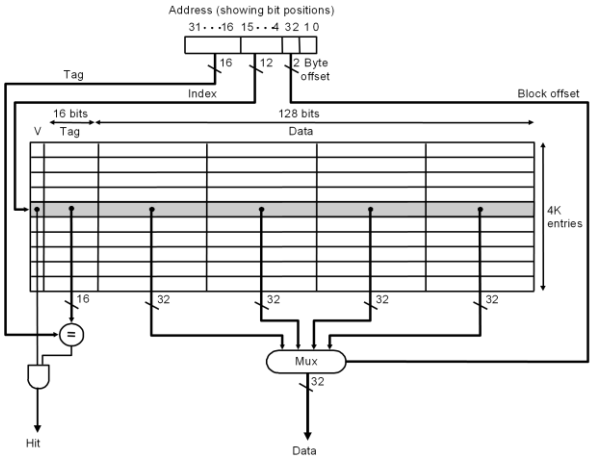
- כל מה שנכתב מה-*CPU* נכתב ל-*cache* ולזיכרון במקביל.
- הכתיבה לזיכרון נעשית באמצעות *write buffer*, שכן כתיבה היא תהליך איטי, וכך לא נדרשים להפסיק עד סיום הכתיבה.
- משפיע על קריאה מהזיכרון: בודקים ב-*cache*, אם לא שם **בודקים ב-*write buffer***, ואם לא שם רק אז ניגשים לזיכרון.
- אם ה-*write buffer* מלא, אז ה-*CPU* יעצור עד שיתפנה מקום.
- מתבסס על הנחה שישנו מרווח מסוים בין כתיבה לכתיבה, כאשר במרווח זה ישנן פעולות אחרות שלא זקוקות ל-*write buffer*.



write back :

שיטה נוספת לפיה נכתוב מה-*cache* לזיכרון רק כאשר הבלוק כולו מוחלף בבלוק אחר.

דוגמא ל-*cache* עם בלוק בגודל 4 מילים :



- לוקח *tag* של תחילת הבלוק מהזיכרון בגודל 16 ביטים.
- גודל כתובת ב-*cache*: 12 ביטים, ולא 14.
- 2 הביטים: ברירה בין אחת מ-4 המילים בבלוק.
- עוד 2 ביטים של *Byte offset* כמו קודם.

אותו מנגנון רק שבלוק ב-*cache* יכיל:

- *tag* בגודל 16.
- 4 מילים × 32 ביטים כל אחת.
- *valid field* (1 ביט).
- *MUX* הבורר בין אחת מ-4 המילים.

בצורה כללית לבלוק בגודל 2^m:

- גודל *tag*: *30-n-m*
- גודל כתובת ב-*cache*: *n*
- מספר מילים: 2^m, סה"כ $2^m \times 32$ ביטים.
- *MUX* בורר המקבל *m* קוי כניסה (נקרא קו בקרת ה-*block offset*).

גודל בלוק:

אם נגדיל את גודל הבלוק, ה-*miss rate* יקטן. עם זאת, אם לא נגדיל את גודל ה- *cache* במקביל, יהיו לנו פחות בלוקים ב-*cache*, ומגודל בלוק מסוים ה-*miss rate* יגדל.

<u>*Associative cache*</u> :

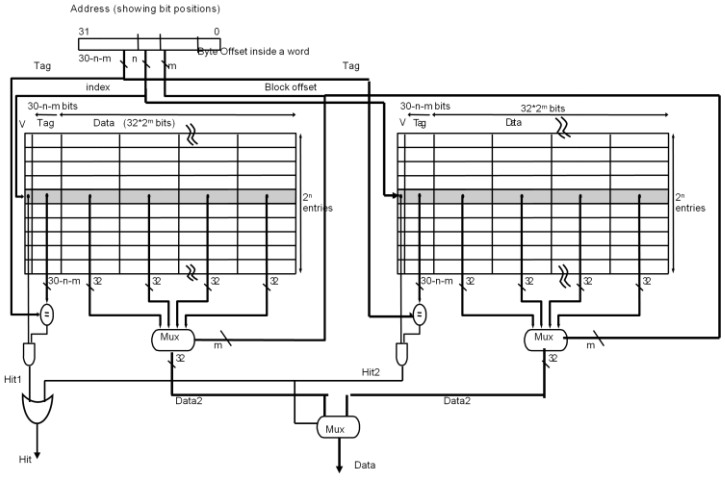
לכל בלוק יש **כמה** מקומות בזיכרון אליו יכול להתמפות . ב- *fully associative*: כל בלוק יכול להתמפות לכל בלוק ב-*cache*.

דוגמא ל-*2-way associative*:

נוסף לנו *MUX* הבורר בין הבלוקים האפשריים . הוא בורר בין ה-*MUX* של אותם בלוקים, שבוררים את המילה המתאימה.

HIT: *HIT* באחד הבלוקים.

בדיקת הבלוקים האפשריים מתבצעת **במקביל** ולכן פעולה זו יעילה (אך יקרה).



Associative: כיצד יוחלט איזה בלוק להחליף בעת MISS :

- *LRU (last recently used)* : מעיפים את הבלוק שהכי פחות שומש לאחרונה. קשה למימוש.
- *RANDOM* : מעיפים בלוק בצורה רנדומאלית. לא ברור איך יעבוד.

הפחתת MISS penalty :

שימוש בכמה *levels* של *caching* – כל שכבה נמוכה יותר היא גדולה יותר ואיטית יותר.

Virtual Memory :

שימוש ב-*VM* לצורכי :

- הגנה מפני גישת תוכניות לאזורים בעייתיים. ניהול הזיכרון הוירטואלי של התוכניות ע"י מערכת ההפעלה.
- "הגדלת" הזיכרון עבור תוכניות – *VM* גדול מה-*physical memory*.

<u>מושגים</u> :

- *<u>page</u>* : מקביל לבלוק ב-*cache*.
- *<u>page fault</u>* : מקביל ל-*miss* ב-*cache*. כתובת שלא נמצאת ב-*physical memory* אלא בדיסק (הזיכרון הגדול והאיטי ביותר).

<u>Address translation</u> :

- תוכניות ב-*CPU* עובדות עם *virtual address* : למשל אלו המחושבות ב-*ALU* עבור *sw,lw* או הבאת פקודה מה-*PC*.
- 12 הביטים ה-*LSB* של הכתובת נשארים אותו דבר : *page offset* – איפה אנחנו בתוך ה-*page*.
- 20 הביטים הנותרים מה-*VA* מתורגמים ל-**18** ביטים ב-*PA* – מה שאומר שהמרחב הכתובות הוירטואליות גדול יותר.
- לכל תוכנית *page table* משלה, וכל תוכנית מחזיקה מצביע אליה ברגיסטר מיוחד.

<u>valid field</u> :

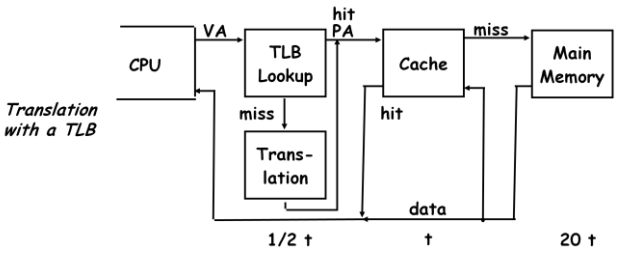
- ביט המחזיק 1 אם יש בטבלה את ה-*PA* של העמוד הנמצא ב-*Physical memory* או 0 אם הכתובת שמחזיק היא לדיסק.
- ישנה השמה *fully-associative* של עמודים מהדיסק ל- *physical memory*, כדי להוריד *page-faults*. כמו כן הכתיבה מהדיסק , הלוקחת זמן רב, נעשית ב-*write back* ולא *write-through* (בניגוד ל-*cache*). בקיצור : כמה שיותר מהיר.
- החלפת *pages* ב-*physical memory* : ה-*OS* אחראית על כך, לרוב לפי *LRU*, תשתמש בשדות : *use, ref* כדי לקבוע זאת.

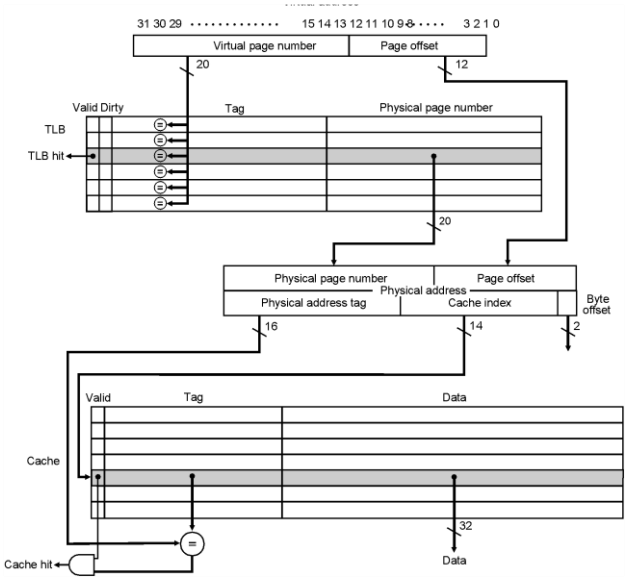
<u>dirty bit</u> :

שדה המאחסן 1 אמ"מ נעשתה כתיבה על ה-*page* ב-*physical memory*. אם כן, כאשר יוחלף העמוד באחר, נצטרך קודם לעדכן את העמוד שמוסר בדיסק, ורק לאחר מכן להחליפו בעמוד הרצוי. אם *dirty=0*, אין צורך להעתיק את העמוד לדיסק.

<u>TLB (Translation lookaside buffer)</u> :

- מעיין *cache* ל-*page table*, מאוד יעיל וקטן.
- כאשר מוחלפת רשומה ב-*TLB* עם אחת אחרת ב-*page table*, שדה ה-*dirty* של הרשומה יעודכן ב-*page table*.





משמאל מתוארים שני המעברים בדרך לזיכרון:

- מעבר ב-*TLB*, יכול להוביל למעבר ב- *page table* אם יש *miss* ב-*TLB*.
- מעבר ל-*cache* הרגיל עם הכתובת הפיסית.

דרכים לייעול:

- במקום לחכות לסיום קריאת ה-*PA* מה-*TLB*, נשתמש בשיטה לפיה ה-*page offset* מכיל את ה-*#set* (מספר הבלוק) ב-*cache*, כך שנוכל ישר לגשת אליו. ברגע שתהיה לנו *PA* מה-*TLB* (או ה-*page table*), נבדוק ב-*cache* את ה-*tag comparison*.

דרכים להגדל *cache-hit rate* :

Pseudo LRU (עבור *4-way*) :

- *full LRU* מחזיק רשומות של כניסות לכל הבלוקים לפי סדר גישה אחרונה כאן – יוחזקו 4 רשומות.
- *Pseudo LRU* מחזיק שלושה ביטים המשווים: *bit0* – האם 0/1 או 2/3 ; *bit1* – מבין 0/1 ; *bit2* – מבין 2/3.

שיפור בשלושת ה-*Cs* :

- *Compulsory* : לבצע *prefetching*. מפורט למטה.
- *Capacity* : הגדלת *block size*.
- *Conflict* : הגדלת *associativity*.

prefetching :

דרך אחת : ניתן להשתמש בשיטת ה-*request word first / wrapped fetch* : קודם מביאים את המילה שעליה יש*miss*, שולחים אותה ל-*CPU* כדי שימשיך לבצע ולא יתקע. במקביל ממשיכים להביא את שאר הבלוק.
דרך נוספת (כללי) : הבאת מידע לפני שנתבקשנו להביאו:

h/w prefetching :

- *instruction prefetching* : הבאת 2 בלוקים ב-*miss* ; *branch predictor* ;
- *data prefetching* : לנסות לחזות את הכניסות הבאות ל-*data* (תבניות וכו').

s/w prefetching :

- *data prefetching* : העלאת מידע לרגיסטר ; *cache prefetch* (כמו שתואר קודם) ; מושרש בשפה או נעשה ע"י הקומפיילר.

complier optimization :

- *instructions* : ארגון מחדש של הפקודות כדי למנוע*misses* ; שימוש בכלים שונים.
- *data* : איחוד מערכים ; שינוי קינון לולאות כדי שיעבוד לפי סדר אחסון בזיכרון; איחוד לולאות ; ייעול *locality* מקומית ע"י *blocking* : גישה למידע לפי בלוקים, במקום ללכת לפי שורות ועמודות שלמות(כמו שעשינו בפרוייקט תוכנה בחישוב מטריצות).

שיטות נוספות:

multi ported cache and banked cache :

- *n* גישות מקבילות ל-*cache*.
- בעיה : מקצר את זמן "תמותת" ה-*cache*. פתרון : *banking* : חלוקת כל שורה ל-*banks* והבאת *data* עבור *banks* מסויימים לכל שורה.

הפרדת *cache* ל-*code* ול-*data* : מאפשר גישה מקבילה לשניהם.

הגדלת *L2* : ה-*cache* עם *minimum latency lost*.

שימוש ב-*victim cache* : *cache* נוסף עם אותו *access time* אליו נשלחות שורות שנמחקות מה-*cache*, כך ששחזורן יהיה מהיר (במקום להביאן שוב מה-*memory*).

שימוש ב-*stream buffer* : כל מידע שנכנס ל-*cache* יעבור קודם דרך *stream buffer*. רק אם תהיה אינדיקציה שיגשו למידע זה שוב בעתיד, נכניס אותו ל-*cache*? כך למשל נמנע הכנסת מערך גדול ל-*cache* שימחק ממנו הכל, אם ידוע שלכל איבר במערך אנו צפויים לגשת רק פעם אחת

Virtual memory and process switch :

- שמירת מצב התוכנית.
- טעינת מצב התוכנית החדשה.
- טעינת הרגיסטרים המתאימים המצביעים ל-*Translation table*.
- ניקוי ה-*TLB* (אין צורך לנקות את ה-*cache*).

branch prediction :

- מחיר חיזוי מוטעה של *branch* עולה כאשר : אורך ה-*pipeline* עולה, אורך המכונה (משפיעים על מספר הפקודות שנעשה להן *flush*).

dynamic branch prediction :

- שיטה יעילה כאשר זמן החישוב להאם לוקחים את ה-*branch* גבוה מהזמן שלוקח לחשב את הכתובות האפשריות אליהן נלך.
- השיטה : מחזיקים *buffer (cache)* ה-*branch inst.*, יחד עם ביטים המעידים על האם ה-*branch* נלקח לאחרונה או לא.
- אם החיזוי מוטעה, משנים את הביט.
- מחזיקים *BTB – branch target buffer* : מחזיק את כתובת ה-*branch* ב-*PC*, הכתובת ה-*predicted* וביט הנלקח או לא נלקח. כל כתובת נבדקת : האם היא כתובת *branch* – יש למשוך את ה-*prediction*, או לא.
- כאשר מתבצע *branch* מעדכנים את ה-*BTB* : כתובת ה-*branch* ונלקח או לא.
- במקרה של *misprediction* : נעשה *flush* ונטען את הפקודה המתאימה מה-*PC*.
- אופציה נוספת : שינוי ביט ה-*taken* כל *2* פיספוסים.

הוספה ל-*mips* :