12/03/08

מבנה מחשבים הרצאה 8

Single Cycle Architecture

נרצה לממש בשלבים מעבד מסוג MIPS. לשם כך ניזכר בשלבי ביצוע פקודת MIPS:

1) **שלב הFETCH-** גישה לזיכרון, לכתובת שכרגע שמורה בPC (Program Counter), שם מאוכסנת הפקודה שנרצה לבצע (למי שלא זוכר, הPC הוא הרגיסטר שמכיל את הכתובת של הפקודה שמתבצעת.)

2) **שלב הDECODE-** פענוח הפקודה וקריאת התוכן של האוגרים הנחוצים (אוגר אחד ב-I Type, שניים ב R-Type).

3) **שלב הEXECUTE-** ביצוע הפקודה.

4) **שלב הMEMORY-** השלב שבו טוענים מהזיכרון (LOAD) או קוראים מהזיכרון. (זיכרון הפקודות, אשר נפרד מזיכרון הDATA הרגיל, מכיל את כל הפקודות בתוכנית **לפי הסדר**) אם בשלב הMEMORY אנו אמורים לבצע טעינה מהזיכרון, כמו למשל בפקודת lw, שלב זה יתבצע לפני שלב 3 (EXECUTE). אם בשלב זה אנו אמורים לבצע קריאה מהזיכרון, כמו למשל בפקודת sw, שלב זה יתבצע אחרי שלב 3.

5) **שלב הWRITE BACK-** שלב זה קיים רק בפקודות, כמו ADD, שבהם אנו אמורים לאחסן את התוצאה באוגר.

המעבד שנתכנן יהיה מבוסס על רכיבים שהם Edge Triggered, כלומר שנטענים **במשך** עליית השעון, ולא משנים את ערכם כל מחזור השעון.

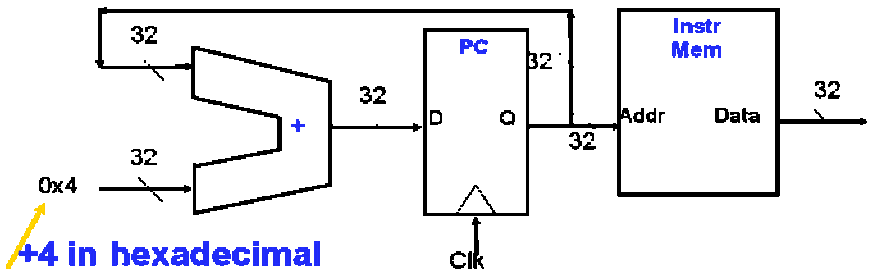
לשם הפשטות, המעבד הראשוני שנלמד יהיה מסוג Single Cycle, כלומר כל פקודה מבוצעת במחזור שעון יחיד (וארוך...). מעבד כזה הוא לא יעיל ולכן לא משתמשים בו במציאות.

<u>נסביר מה תפקידו של כל רכיב:</u>

רכיב ה Instruction Memory- זיכרון הפקודות. זיכרון הפקודות הוא מאגר הפקודות, והוא נפרד מהזיכרון הרגיל של הנתונים. הרכיב מקבל כקלט כתובת בת 32 ביטים ומוציא כפלט את הפקודה שנמצאת בכתובת, הפקודה מיוצגת באמצעות 32 ביטים.

רכיב הPC(Program Counter)- זהו רגיסטר הבנוי ממערך של 32 FF (על מנת לאחסן כתובת שגודלה 32 ביטים).בכל עליית שעון נרצה לעלות את הכתובת שבPC ב-4 על מנת לעבור לכתובת של הפקודה הבאה (מפני שכל הפקודות בתוכנית נמצאות בזיכרון הפקודות לפי הסדר ואורך כל פקודה הוא 32 ביטים=4 בתים), אלא אם כן יש לנו פקודה מסוג branch או jump, שאז נממש לוגיקה מיוחדת על מנת לחשב את הכתובת הבאה בPC.

רכיב הALU- מבצע פעולה בינארית (כגון חיבור) על שני מספרים בני 32 ביטים. כניסת בקרה בשם op היא הקובעת איזה פעולה בינארית תתבצע. למשל באמצעות הALU נוכל בין היתר להעלות את ערכו של הPC ב4 כל מחזור שעון, באופן הבא:



*****ראשית נדון בפקודות מסוג R-Type***:**

רכיב ה(RegFile (Register File או בקיצור Reg- הפקודה שמתקבלת ביציאה של ה InstrMem מתפצלת באופן המתאים לייצוג של פקודה מסוג R-Type,

opcode	rs	rt	rd	shamt	funct

כלומר 6 הביטים הראשונים הם של opcode,

5 ביטים הבאים הם של rs (מייצג את המספר של הרגיסטר הראשון שקוראים ממנו),

5 הביטים הבאים הם של rt (מייצג את המספר של הרגיסטר השני שקוראים ממנו),

5 הביטים הבאים הם של rd (הרגיסטר שבו מאחסנים את התוצאה),

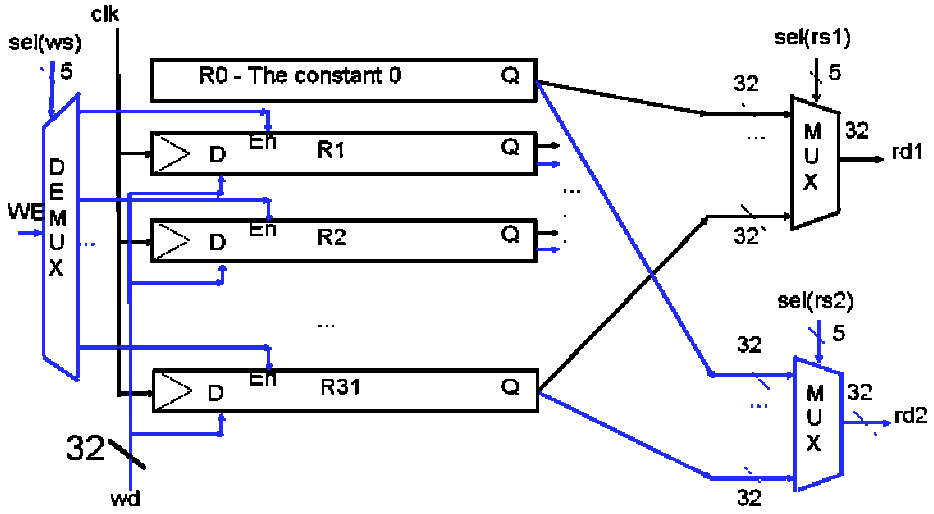
5 הביטים הבאים נקראים shamt (זה קיצור של shift amount וזה מיועד עבור פקודות שבהם עושים הזזה, לא נראה לי שזה הוסבר בקורס).

6 הביטים הבאים נקראים funct.

(ראו סיכום ההרצאה על MIPS).

רכיב הRegFile אחראי על קריאה מהרגיסטרים המתאימים (הרגיסטרים שמספרם נמצא בrs וrt) וכתיבה לרגיסטר המבוקש (הרגיסטר שמספרו מיוצג בrd).

נזכור כי קיימים 32 רגיסטרים. הRegFile ממומש באופן הבא:



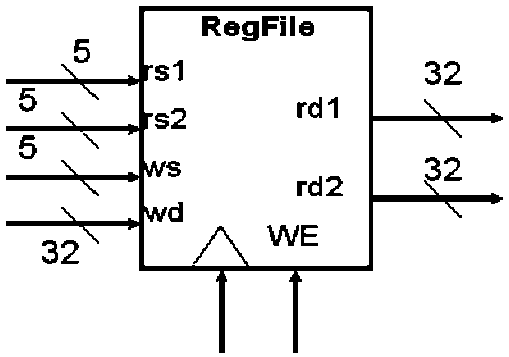
באמצעות ה DEMUX נבחר לאיזה מהרגיסטרים יכתב ה wd (write data). יש לנו 31 אפשריות כי לרגיסטר הראשון- רגיסטר האפס R0- לא נוכל לכתוב והערך שנמצא בו הוא תמיד 0. באמצעות 2 הMUX-ים שבחלק הימני של הציור נבחר את 2 הרגיסטרים שאותם נרצה לקרוא (למשל בפקודת ADD אנו קוראים משני רגיסטרים, אותם נבחר באמצעות 2 הMUX-ים, ובסוף כותבים לRegister שאותו בוחרים באמצעות ה DEMUX).

דוגמה מההרצאה: אם הפקודה שלנו היא ADD $8 $9 $10, כלומר הפקודה מיוצגת באמצעות

opcode	rs	rt	rd	shamt	funct

כאשר rs הם 5 ביטים המייצגים את מספרו (מתוך 32) של הרגיסטר $9, rt הם 5 ביטים המייצגים את מספרו של $10 וrd הם 5 הביטים שמייצגים את מספרו של $8.

את הייצוג של הפקודה נרצה לחבר לרכיב הRegFile באופן הבא: 5 הביטים שבrs יהיו מחוברים לכניסת הבקרה של הMUX העליון בציור, 5 הביטים שבrd נרצה לחבר לכניסת הבקרה של ה MUX התחתון, ו5 הביטים שבrd נרצה לחבר לכניסת הבקרה של הDEMUX. רכיב הRegFile יסומן באופן הבא:



כאשר כניסת rs1 תחובר לכניסת הבקרה של הMUX הראשון,

כניסת rs2 תחובר לכניסת הבקרה של הMUX השני,

כניסת ws תחובר לכניסת הבקרה של הDEMUX,

כניסת wd תכיל את 32 הביטים של הDATA אותו נרצה לכתוב לרגיסטר הנבחר בDEMUX (הרגיסטר שבו נרצה לאחסן את התוצאה של הפקודה).

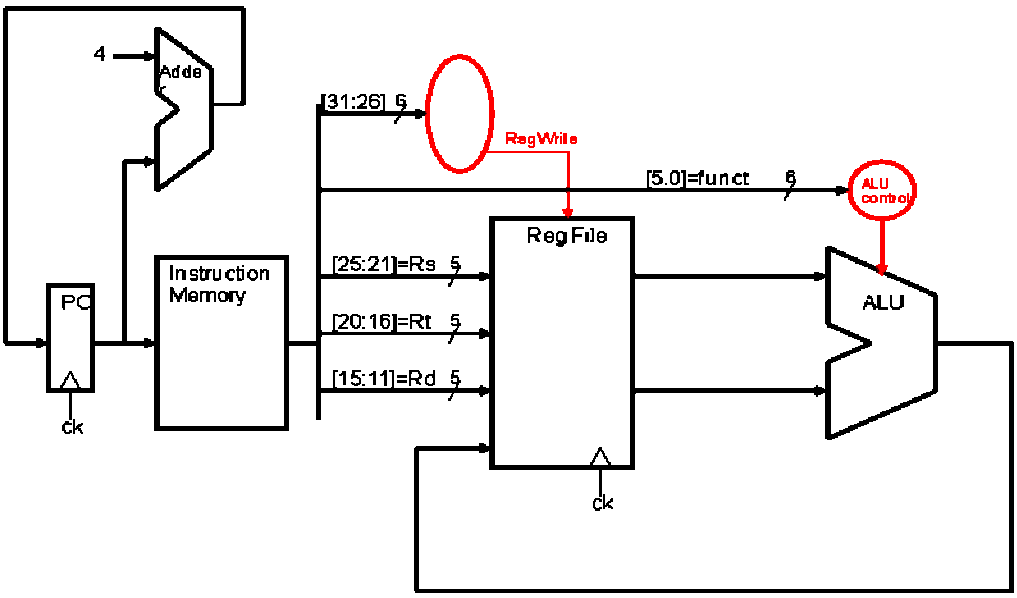
הכניסה המסומנת במשולש היא כידוע "לשמצה" כניסת הCP, או בעברית Clock Pulse.

כניסת הWE היא ביט שיסמן לנו אם כרגע אנו כותבים את המידע בכניסה wd לתוך הרגיסטר שמספרו נתון בכניסה ws (ראו את המימוש של RegFile למעלה).

יציאת הrd1 היא היציאה של MUX1,

יציאת הrd2 היא היציאה של MUX2.

חיבור של הרכיבים שהצגנו עד כה יעשה באופן הבא:



היציאה של הALU היא התוצאה של הפעולה שבוצעה על התוכן של שני הרגיסטרים שמספרם מיוצג בRs וRt, והיא מחוברת חזרה לכניסה התחתונה של ה RegFile כדי שנוכל לכתוב את התוצאה חזרה לרגיסטר שמספרו מיוצג בRd.

ביט הבקרה "RegWrite" זה מה שקראנו לו קודם WE- אומר לנו אם כותבים את המידע בכניסה שנכנס בכניסה התחתונה של ה RegFile לתוך הרגיסטר שמספרו נתון בכניסה ב5 הביטים של הrd.

חישוב או בחירת הפקודה שתעשה בALU (שזה כניסת הop בALU) נעשית ע"י שימוש בביטים של הopcode והfunct בייצוג של הפקודה (הייצוג הוסבר בעמוד הקודם).

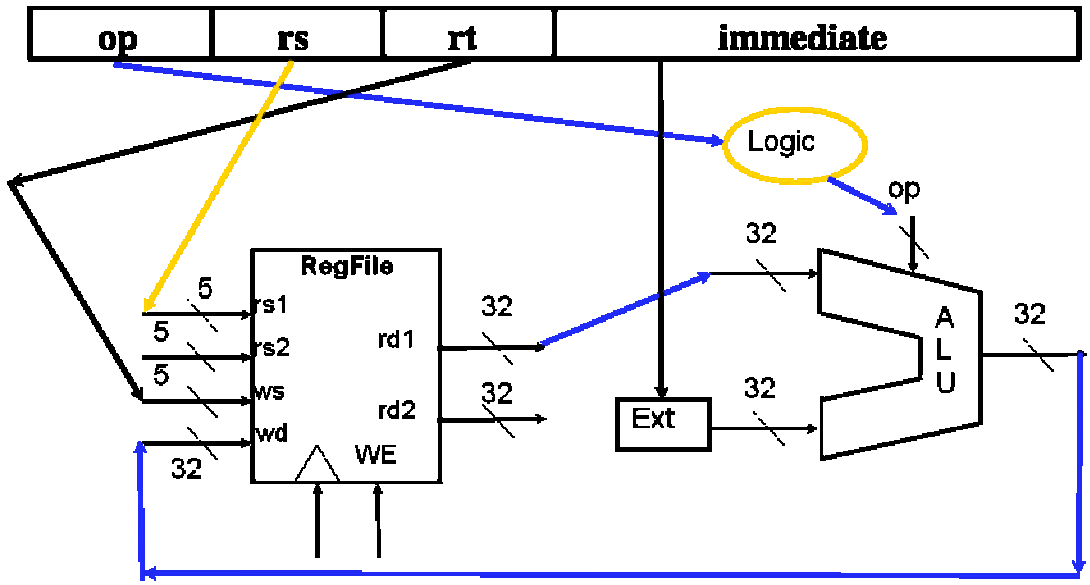
*****כעת נדון בפקודות מסוג I-Type***:**

פקודה מסוג I-Type מיוצגת כזכור באופן הבא:

op	rs	rt	immediate

במצב זה לא נוכל לעבוד עם המימוש הקודם מפני ש5 הביטים שב rt מייצגים את הרגיסטר שכותבים אליו ולא את הרגיסטר שקוראים ממנו.

לכן במקרה זה נצטרך לחבר את הביטים של הפקודה (שביציאה מזיכרון הפקודות) לRegFile באופן הבא:



כאשר Ext הוא רכיב חדש שתפקידו להרחיב את 16 הביטים בimmediate למספר של 32 ביטים בלי לשנות את הערך (שהרי הכניסה של הALU אמורה לקבל מספר בין 32 ביטים).

*****כעת נשלב בין המימוש של פקודות I-Type ופקודות R-Type*****

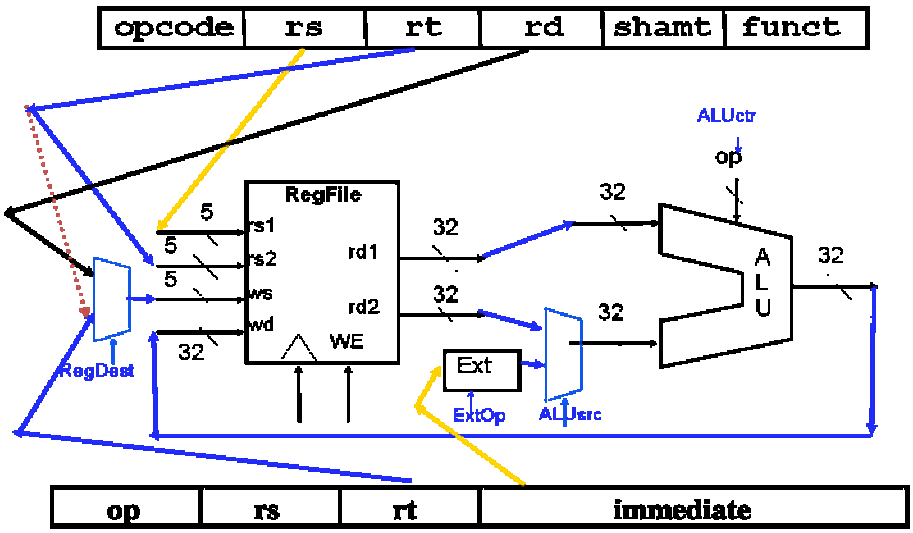
בשביל לשלב בין שני המימושים נצטרך להוסיף שני MUX-ים:

אחד שיכריע איזה חלק מהפקודה (rt במקרה של I-Type או rd במקרה של R-Type) יכנסו לכניסת ws ב RegFile,

ואחד שיכריע מה יכנס לכניסה התחתונה של הALU (הimmediate המורחב במקרה של I-Type או הערך שביציאת הrd2 במקרה של R-Type).

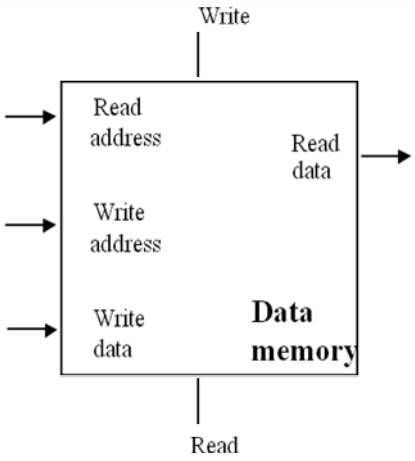
כמו כן נצטרך כמובן ביט בקרה עבור הMUX-ים הללו שיאמר לנו האם הפקודה היא I-Type או R-Type.

והמימוש במשולב הוא:



*****כעת נטפל בפקודות מסוג lw ו sw*****

בשביל פקודות אלה נצטרך להוסיף רכיב (תיאורטי) בעל גישה לזיכרון (זיכרון הData). רכיב זה נקרא Data Memory וסימונו:

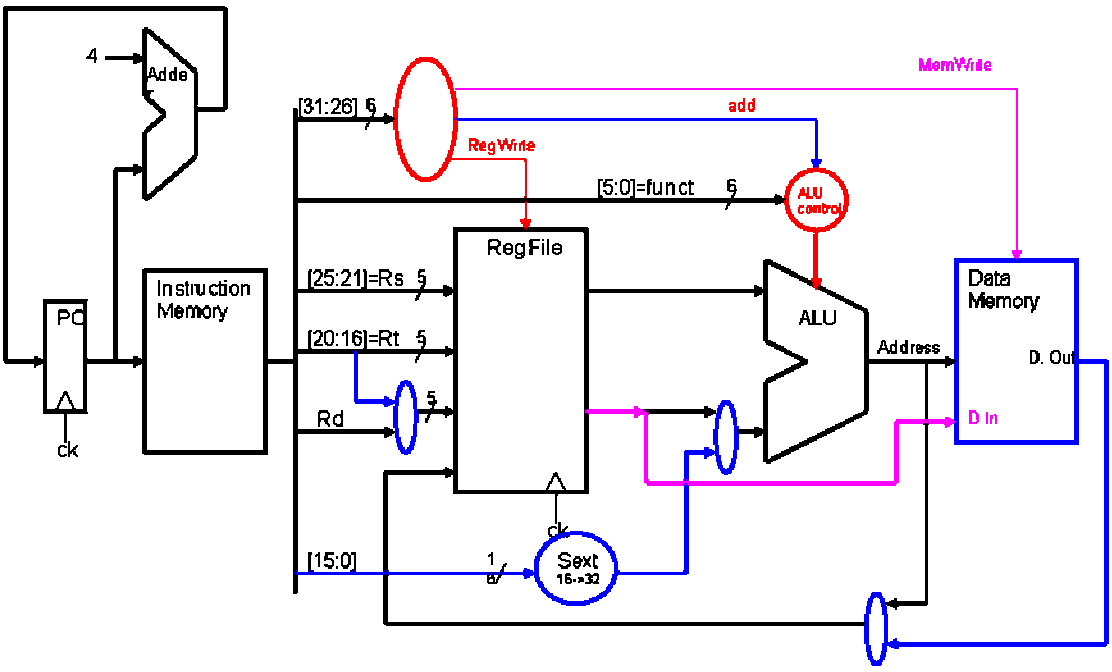


כאשר קריאה מהזיכרון אינה סינכרונית (אם מכניסים כתובת כלשהי לכניסה Read address, זמן קצר לאחר מכן תוכנו של הזיכרון יתקבל ביציאה Read data.

כתיבה לזיכרון היא סינכרונית, כלומר היא תתבצע רק כאשר ערכו של WE היא 1 והclock ב1 לוגי.

מכיוון שפקודות lw וsw הן מסוג I-Type כלומר הגישה לזיכרון תלויה בoffset (immediate) בעל 16 ביטים, אותו צריך לחבר לתוכן של רגיסטר שזה 32 ביטים, נשתמש ברכיב בשם Sign Extend שמרחיב את הimmediate ל32 ביטים בלי לשנות את ערכו, ע"י "מריחת" סיבית הסימן ל16 ביטים שמאליים.

כעת נציג את המימוש של הרכיב שתומך בפקודות R-Type ופקודות lw/sw:



Sext הוא קיצור של Sign Extend.

מה שבכחול קשור הן לפקודות lw והן לפקודות sw.

מה שבורוד קשור לפקודת הsw. בפקודה זו אנו מבצעים כתיבה מרגיסטר לזיכרון, לכן סיבית הבקרה MemWrite "מופעלת" (נמצאת ב1 לוגי). הכתובת **שאליה רוצים לכתוב** מחושבת ע"י חיבור (באמצעות רכיב הALU) של הערך של הרגיסטר שמספרו מיוצג בRs של הפקודה ל offset המורחב. כמו כן, הערך של הרגיסטר אותו רוצים **להטעין לזיכרון** מועבר ישירות מהיציאה התחתונה של RegFile לכניסה D in ב Data Memory (הקו הורוד בציור).

מה שבאדום קשור לפקודות הlw. בפקודה זו מתבצעת כתיבה מהזיכרון לרגיסטר, לכן סיבית הבקרה RegWrite "מופעלת" (נמצאת ב1 לוגי). מכיוון שזוהי פקודה מסוג I-Type, הMUX בכניסה לRegFile בוחר את Rt. כמו כן על מנת לחשב את הכתובת בזיכרון **ממנה רוצים לקרוא** אנו מבצעים Sign Extend לOffset ומחברים אותו (באמצעות רכיב הALU) לערך של הרגיסטר שמספרו מיוצג בRs (אם אתם לא זוכרים ראו הרצאה על MIPS). שימו לב שאין צורך בסיבית בקרה שתצהיר על קריאה מהזיכרון כי הקריאות, כפי שנאמר, הן א-סינכרוניות. כעת ביציאה D Out מה Data Memory יש את התוכן של הכתובת אותה רצינו לקרוא, תוכן זה יעבור לכניסה התחתונה בRegFile **ויאוחסן ברגיסטר הרצוי**.

פקודות הR-Type עובדות כמו שהוסבר קודם לכן.

*****כעת נטפל בפקודות מסוג branch *****

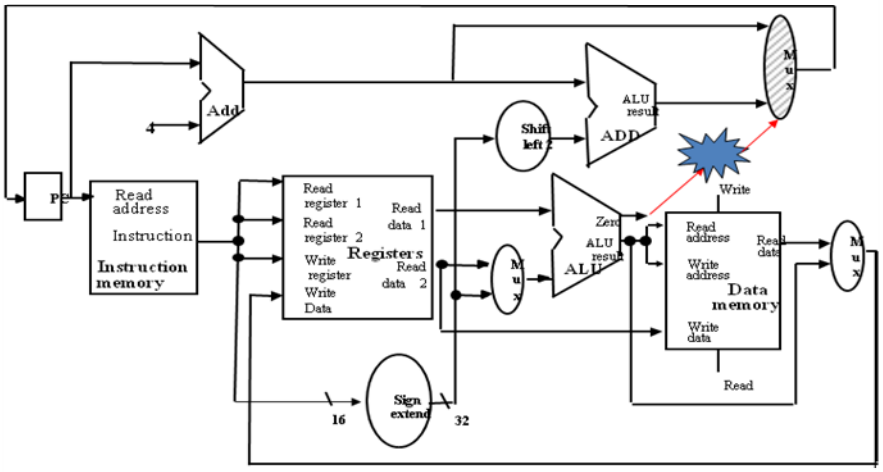
כפי שהוסבר בהרצאה הקודמת, פקודות מסוג branch הם פקודות I-Type, כלומר הייצוג שלהם הוא באופן הבא:

op	rs	rt	immediate

כאשר פקודת BEQ (Branch Equal) למשל היא פקודה אשר משווה בין התוכן של הרגיסטר שמספרו מיוצג בrs והתוכן של הרגיסטר שמספרו מיוצג בrt. אם הם שווים הפקודה הבאה שתתבצע היא immediate פקודות קדימה, כלומר הPC צריך לגדול ב immediate*4 (שכן כל פקודה גודלה הוא 4 בתים= 32 ביטים). אם הם לא שווים הפקודה שתתבצע היא הפקודה הבאה בסדר הרגיל של הפקודות.

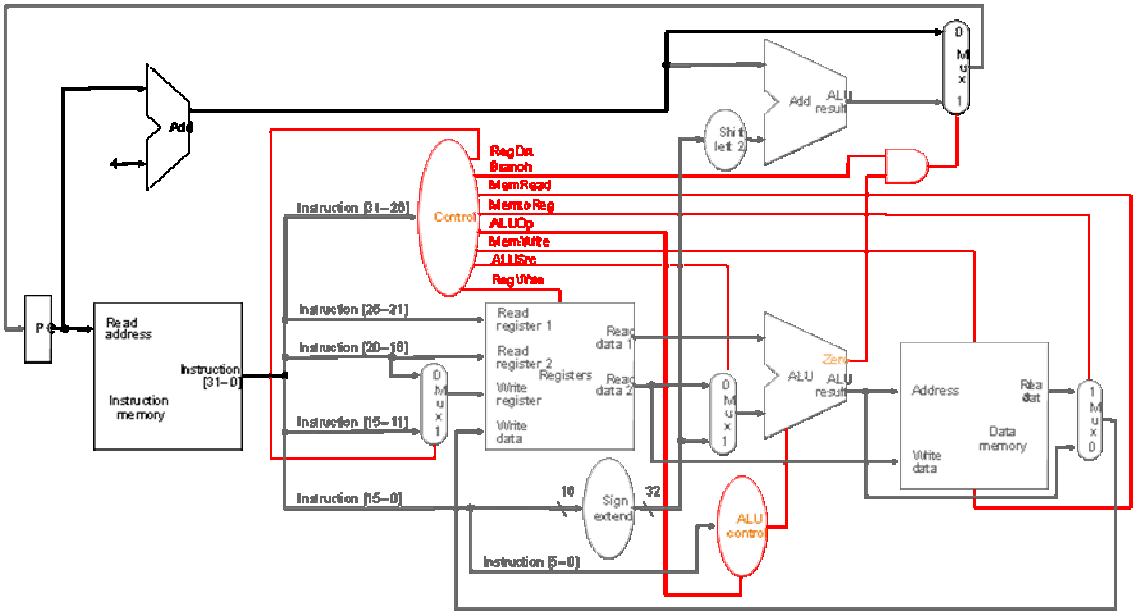
לשם כך נצטרך להוסיף למימוש של המעבד שלנו "משווה" כלומר רכיב ALU שיחסר בין התוכן של הרגיסטר שמספרו מיוצג בrs והתוכן של הרגיסטר שמספרו מיוצג בrt. נרצה שאם התוצאה תהיה שונה מאפס, כלומר התכנים שונים, לא יתבצע שום שינוי (כלומר הPC יעלה ב4 במחזור השעון הבא כרגיל). אך אם התכנים שווים נרצה שהPC יגדל בimmediate*4 במחזור השעון הבא, וזאת נעשה באמצעות הוספת מחבר (ALU) שיחבר בין immediate*4 לערך הנוכחי ב PC. מכיוון שהimmediate הוא 16 ביטים והALU מצפה לקבל בכניסה 32 ביטים נוסיף רכיב שיעשה zero extend לimmediate , כלומר יוסיף 16 אפסים משמאל לimmediate (בפקודות branch הimmediate הוא unsigned), וכן על מנת להכפיל ב4 נוסיף רכיב (Shift Left 2) שיזיז את הimmediate שני ביטים שמאלה ושני הביטים האחרונים החדשים שנוצרו יהיו אפסים.

כעת המימוש המשלב את כל הפקודות עד כה הוא:



הMUX החדש (הסגול) בוחר את הכניסה התחתונה אם אנו בפקודת branch וההפרש בין ערכי הרגיסטרים rs וrt היה אפס (היציאה zero נוספה לALU ה"ותיק"), כאשר הכניסה התחתונה היא הכניסה שבה מחושבת הכתובת אליה צריך לקפוץ במקרה זה (שהbranch התקבל), ואילו בחירה בכניסה העליונה היא המסלול "הרגיל" שבו פשוט מוסיפים 4 לPC.

את כל קווי הבקרה של כל הMUX-ים +ביט הבקרה הקובע קריאה מהזיכרון+ ביט הבקרה הקובע כתיבה לזיכרון בRegFile+ ביט הבקרה של הbranch+ הביטים שבאמצעותם מחליטים על הפעולה שיבצע הALU נרכז ברכיב יחיד, בשם Control, והמימוש נראה באופן הבא:



ALUOp הם שני ביטים, 00 ייצג פקודת lw או sw, 01 ייצג פקודת beq, 10 ייצג פקודה אריתמטית.

מכיוון שבפקודות R-Type כמו ששת הביטים של הfunct משחקים תפקיד בבחירת הפעולה של ה ALU (ניתן לראות בציור כי אחת הכניסות של רכיב הALU Control מתחברת מביטים 0-5 בפקודה), נציג את הטבלה הבאה (שהוצגה בהרצאה) המציגה את היציאה של רכיב ה ALUControl כתלות בשני הביטים של הALUOp והביטים של הfunct:

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
0	1	X	X	X	X	X	X	110
1	0	X	X	0	0	0	0	010
1	0	X	X	0	0	1	0	110
1	0	X	X	0	1	0	0	000
1	0	X	X	0	1	0	1	001
1	0	X	X	1	0	1	0	111

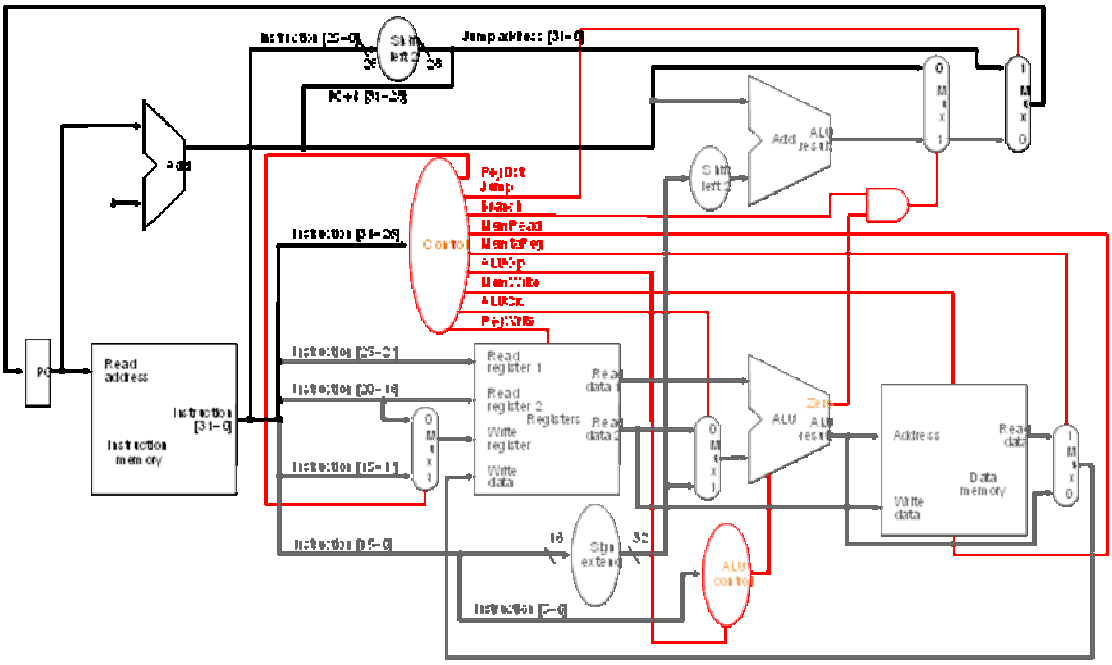
Operation ממיצג את הOUTPUT של ה ALU Control, אשר מפורש ע"י הALU באופן הבא:

```
ALU control output
000     AND
001     OR
010     add
110     subtract
111     set-on-less-than
        (sign of rs-rt -> rd)
```

*****הוספת פקודת jump למימוש*****

קוראים לפקודה מסוג jump עם כתובת מטרה ("target") בת 26 ביטים, המייצגת את הכתובת אליה רוצים לזנק. אבל הכתובת המדויקת אמורה להיות בת 32 ביטים, לכן נשלים את כתובת המטרה ע"י הוספת 4 הביטים הMSB של הכתובת של הפקודה **<u>הנוכחית</u>** בPC לתחילת כתובת המטרה (כדי שלא נוכל לקפוץ יותר מדי רחוק), ועוד שני אפסים נוסיף לסוף כתובת המטרה (כדי להכפיל ב4).

המימוש הסופי של הכל ביחד נראה באופן הבא:



כאשר הוספנו קו בקרה חדש עבור jump, ומה שבמודגש משחק תפקיד בפקודת jump, שזה העיבוד של ה26 ביטים של הtarget כפי שתואר לעיל, וכניסת ששת הביטים הראשונים ל Control.