Preparation Report LAB5

ADVANCED CPU ARCHITECTURE AND HARDWARE ACCELERATORS LAB

361.1.4693

Roy Kislev 206917064

Michael Grenader 208839845

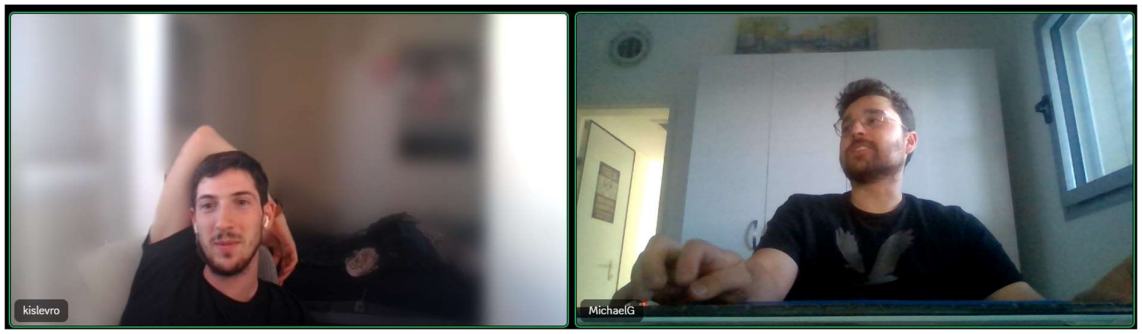

Figure 1 עבודה משותפת מרחוק

תוכן עניינים

מטרת המעבדה

במעבדה זו למדנו להשתמש ביכולות של תוכנת Quartus ובפרט לבצע סינתזה עבור מודלים שפיתחנו בעבר במעבדה 1. את הסינתזה ביצענו על גבי Cyclone V FGPA של כרטיס DE10 standard. כמו כן, המטרה הייתה לממש מעבר pipeline של MIPS נלמד בהרצאות התיאורטיות, שיודע לבצע ISA בסיסית של פקודות אסמבלי.

בדיקת ביצועים

קובץ הרצה

בבדיקת הביצועים הרצנו קובץ אסמבלי עם פקודות שנמצאות בISA של המעבד שפיתחנו במעבדה זו. קובץ האסמבלי מממש חיבור מטריצות $Mat1, Mat2$ שיושבות בזיכרון ומכינה את התוצאה למטריצת $resMat$ שנמצאת בזיכרון גם כן. להלן קובץ הC שפותר את הבעיה –

```c
void addMats(int Mat1[M][M], int Mat2[M][M], int resMat[M][M]){
    define it yourself …
}

void main(){ //int=32bit
    int Mat1[M][M]={{1,2,3,4},{5,6,7,8},{9,10,11,12},{13,14,15,16}};
    int Mat2[M][M]={{13,14,15,16},{9,10,11,12},{5,6,7,8},{1,2,3,4}};
    int resMat[M][M];

    addMats(Mat1,Mat2,resMat); // resMat = Mat1 + Mat2
}
```

Figure **2** קוד *QA* ב*C*

כאשר החיבור מתבצע איבר-איבר. להלן קובץ ה*ASM* שפותר את הבעיה תחת המעבד שלנו –

```
.data

Mat1:   .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
Mat2:   .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
resMat: .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 , 0 , 0 , 0 , 0 , 0 , 0
N:      .word 4

.text
main:
la $a0, Mat1     # $a0 = address of Mat1
la $a1, Mat2     # $a1 = address of Mat2
la $a2, resMat   # $a2 = address of resMat
lw $a3, N        # $a3 = matrix size
jal addMats      # Call addMats function

j finish


addMats:
## Sums two matrixes $a0 and $a1 and put it in $a2
addi    $s0, $0, 0       # element_bytes_pointer = 0
addi    $t0, $0, 4       # const of 4
mul     $s1, $a3, $a3
mul     $s1, $s1, $t0    # num_elements_bytes = 4*N*N

add_loop:
beq     $s0, $s1, done   # while element_bytes_pointer != num_elements_bytes
add     $t0, $a0, $s0    # find Mat1 pointer with offset
add     $t1, $a1, $s0    # find Mat2 pointer with offset
add     $t2, $a2, $s0    # find resMat pointer with offset

lw      $t0, 0($t0)      # get Mat1 pointer value
lw      $t1, 0($t1)      # get Mat2 pointer value
add     $t3, $t1, $t0
sw      $t3, 0($t2)      # resMat[i] <=  Mat1[i]+Mat2[i]

addi    $s0, $s0, 4      # next word
j       add_loop

done:
jr      $ra
```

Figure **3** קוד *QA* ב*ASM*

הערה – פקודת la מקודדת לפקודות שאכן נמצאות ב*ISA* שלנו.

לאחר הרצת התוכנית נצפה לקבל את הזיכרון הבא –

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	1	2	3	4	5	6	7	8
0x00000020	9	10	11	12	13	14	15	16
0x00000040	1	2	3	4	5	6	7	8
0x00000060	9	10	11	12	13	14	15	16
0x00000080	2	4	6	8	10	12	14	16
0x000000a0	18	20	22	24	26	28	30	32
0x000000c0	4	0	0	0	0	0	0	0

Figure **4 זיכרון בסוף** *MARS*

ניתן לראות ש2 השורות הראשונות הן $Mat1$, 2 השורות הבאות הן $Mat2$ ו2 השורות לאחר מכן הן $resMat$.

הערה – כתובת $0xC0$ מחזיקה את גודל המטריצה הריבועית ואכן נראה שזה 4 כי המטריצות הן 4×4.

סימולצית ModelSim

בסימולציה זו נריץ את קטע הקוד המצורף לאחר קידוד ל$intel\ hex$ יחד עם הdata mem.

להלן סימולציית המודלסים –

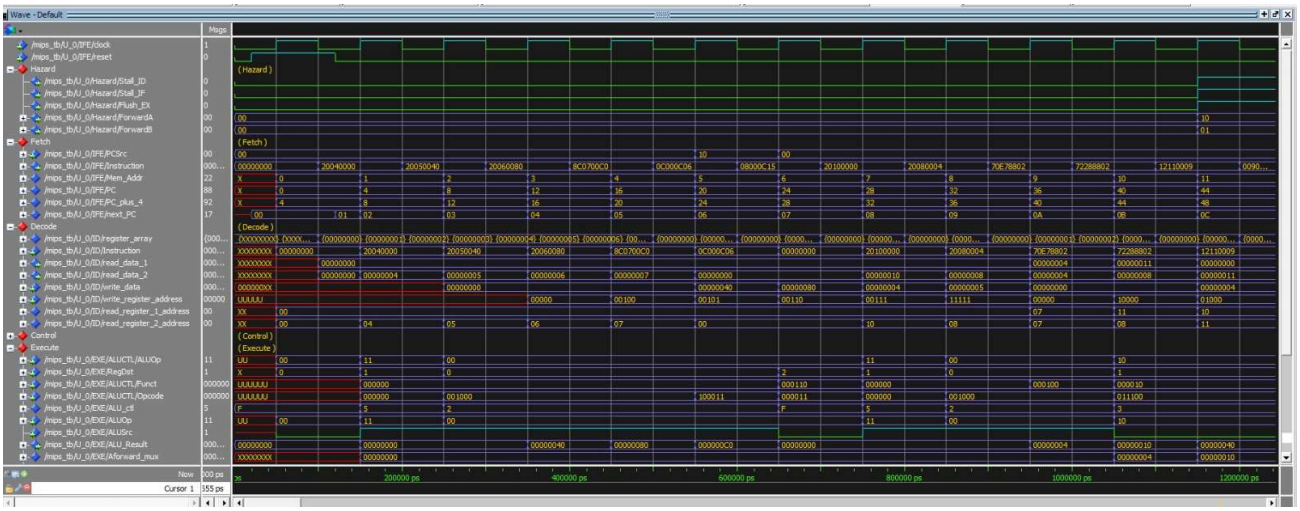

Figure **5 סימולציה מלאה**

ניתן לראות שההוראות אכן מתקבלות בIF לפי הסדר המתאים וכנל הגלים המתאימים.

כאשר נפתח את הregFile ניתן לראות שהתבצעו כתיבות מתאימות –

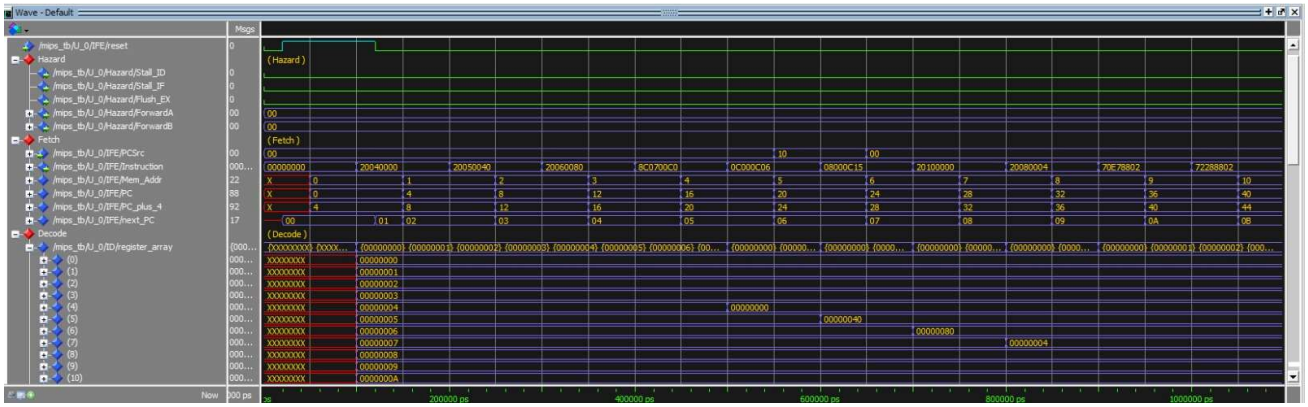

Figure **6 רגיסטרים ב**ModelSim

ניתן לראות שרגיסטר 4 מקבל את הערך 0, רגיסטר 5 את הערך $0x40$ וכדומה. זה כצפוי מפני שהקוד אסמבלי מכניס את הפוינטרים על הכתובות שמתאימות למטריצות הdata memory.

כעת נבחין בקובץ האסמבלי שיש פקודת הadd בגלל שיש lw לפניה מאותו הרגיסטר t1 ולכן נצפה שיהיה stall –

0x0000303c	0x8d290000	lw $9,0($9)	44: lw	$t1, 0($t1)	# get Mat2 pointer value
0x00003040	0x01285820	add $11,$9,$8	45: add	$t3, $t1, $t0	
0x00003044	0xad4b0000	sw $11,0($10)	46: sw	$t3, 0($t2)	# resMat[i] <= Mat1[i]+Mat2

Figure **7 סט פקודות שמביאות ל**stall

כעת נבחין שאכן סיגנל הstall עלה במקרה זה –

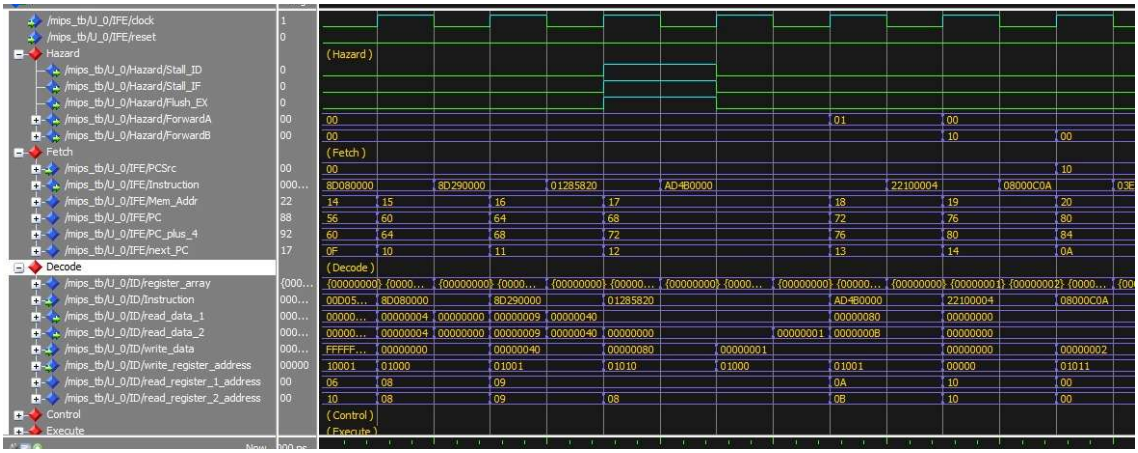

Figure **8 ביצוע** stall **ב**modelSIM

כמו כן ניתן לראות שאכן התבצע stall – ההוראה מושהת למחזור שעון נוסף.

כעת הרצנו לסוף התוכנית לראות את הזיכרונות. תחילה ראינו את הזיכרון בMARS כדי לראות שהתוכנית אכן מתבצעת כראוי וגם שנקבל רפרנס למה אמור להתקבל במעבד שלנו –

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	1	2	3	4	5	6	7	8
0x00000020	9	10	11	12	13	14	15	16
0x00000040	1	2	3	4	5	6	7	8
0x00000060	9	10	11	12	13	14	15	16
0x00000080	2	4	6	8	10	12	14	16
0x000000a0	18	20	22	24	26	28	30	32
0x000000c0	4	0	0	0	0	0	0	0

Figure **9 זיכרון בסוף הQA ב***MARS*

נבחין שתוכן שורות 5 – 4 הן אכן תוצאת חיבור השורות 1 – 0 וגם 3 – 2 בצורת איבר-איבר כפי שציפינו.
נביט בזיכרון במודלסים כדי לוודא כעת נכונות של הקוד שלנו –

Memory Data - /mips_tb/U_0/MEM/data_memory/MEMORY/m_mem_data_a - Default

255	0	0	0	0	0	0	0	0	0
246	0	0	0	0	0	0	0	0	0
237	0	0	0	0	0	0	0	0	0
228	0	0	0	0	0	0	0	0	0
219	0	0	0	0	0	0	0	0	0
210	0	0	0	0	0	0	0	0	0
201	0	0	0	0	0	0	0	0	0
192	0	0	0	0	0	0	0	0	0
183	0	0	0	0	0	0	0	0	0
174	0	0	0	0	0	0	0	0	0
165	0	0	0	0	0	0	0	0	0
156	0	0	0	0	0	0	0	0	0
147	0	0	0	0	0	0	0	0	0
138	0	0	0	0	0	0	0	0	0
129	0	0	0	0	0	0	0	0	0
120	0	0	0	0	0	0	0	0	0
111	0	0	0	0	0	0	0	0	0
102	0	0	0	0	0	0	0	0	0
93	0	0	0	0	0	0	0	0	0
84	0	0	0	0	0	0	0	0	0
75	0	0	0	0	0	0	0	0	0
66	0	0	0	0	0	0	0	0	0
57	0	0	0	0	0	0	0	0	0
48	4	32	30	28	26	24	22	20	18
39	16	14	12	10	8	6	4	2	16
30	15	14	13	12	11	10	9	8	7
21	6	5	4	3	2	1	16	15	14
12	13	12	11	10	9	8	7	6	5
3	4	3	2	1					
-6									

Figure **10 זיכרון בסוף הQA ב***modelSIM*

ניתן לראות שאכן התקבל מידע זהה למה שהתקבל ב*MARS* וגם למה שהתוצאה של חיבור מטריצות אלו אמורה לתת.

מציאת תדר מקסימלי

בכדי למצוא תדר מקסימלי של המערכת שלנו, נוסיף את קובץ הSDC הבא:

```
#*************************************************************
# Time Information
#*************************************************************

set_time_format -unit ns -decimal_places 3


#*************************************************************
# Create Clock
#*************************************************************

create_clock -name {clock} -period 56.000 -waveform { 0.000 30.000 } [get_ports { clock }]
```

Figure **11 קובץ הSDC בתוכנית**

נזכור שהמערכת שלנו היא סינכרונית כי הMIPS פועל לפי שעון בו הוא מקדם את הpipeline ולכן ניתן למצוא את התדר המקסימלי במערכת הנוכחית.

נבצע את הקימפול והסינתזה ללא השמה לפינים כפי שמבוקש (מלבד השעון עצמו). לאחר ביצוע קימפול וסינתזה, תוכנת Quartus יודעת להביא לנו את התדר המקסימלי של המערכת כך שהתקבל –

	Fmax	Restricted Fmax	Clock Name	Note
1	43.84 MHz	43.84 MHz	clock	

Figure **12 מציאת** *Fmax*

– ניתן להגדיל אפילו יותר את התדר המקסימלי על ידי הוספת PLL, אך לא ביצענו את זה בפועל במעבדה.

מציאת נתיב קריטי

בחלק זה נשים את הנתיב הקריטי של המערכת שלנו –

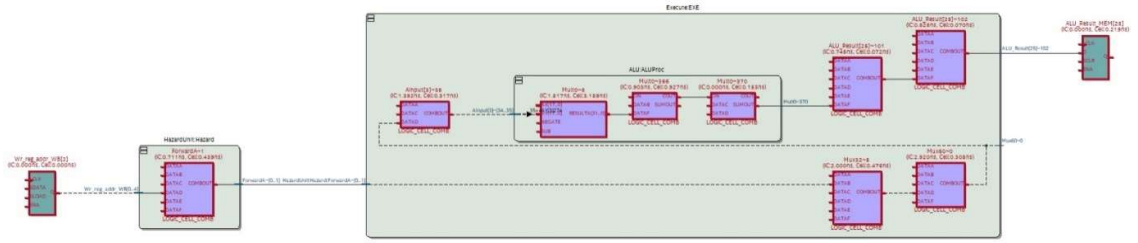

Figure **13נתיב קריטי**

שימוש בלוגיקה עבור כל מודול

כידוע אנחנו מפתחים קוד לוגי אשר משתמש באלמנטים לוגים שניתן לקנפג אותם בהתאם לקוד, דבר שנעשה כחלק מתהליך הסינתזה. בניתוח זה נציג את הלוגיה עבור כל מודול כנדרש –

Analysis & Synthesis Resource Utilization by Entity

<<Filter>>

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits	DSP Blocks	Pins	Virtual Pins
1	|MIPS	1528 (53)	1336 (336)	65536	2	54	0
1	|Execute:EXE|	508 (297)	0 (0)	0	2	0	0
2	|HazardUnit:Hazard|	26 (26)	0 (0)	0	0	0	0
3	|Idecode:ID|	842 (842)	992 (992)	0	0	0	0
4	|Ifetch:IFE|	19 (19)	8 (8)	32768	0	0	0
5	|WRITE_BACK:WB|	64 (64)	0 (0)	0	0	0	0
6	|control:CTL|	16 (16)	0 (0)	0	0	0	0
7	|dmemory:MEM|	0 (0)	0 (0)	32768	0	0	0

Figure **14 שימוש בלוגיקה**

פירוט המערכת

בסעיף זה נסביר על המערכת שלנו ככלל ועל תתי המודולים שלה בפרט. עבור כל אחד ניתן סקירה קצרה על אופן פעילותו, נציג את הRTL שלו לאחר ביצוע הסינתזה, נפרט את הלוגיקה בה הוא משתמש, נמצא נתיב קריטי של פעילותו ונציג אותו בעזרת תוכנת הQuartus.

להלן שרטוט המערכת הסופי –

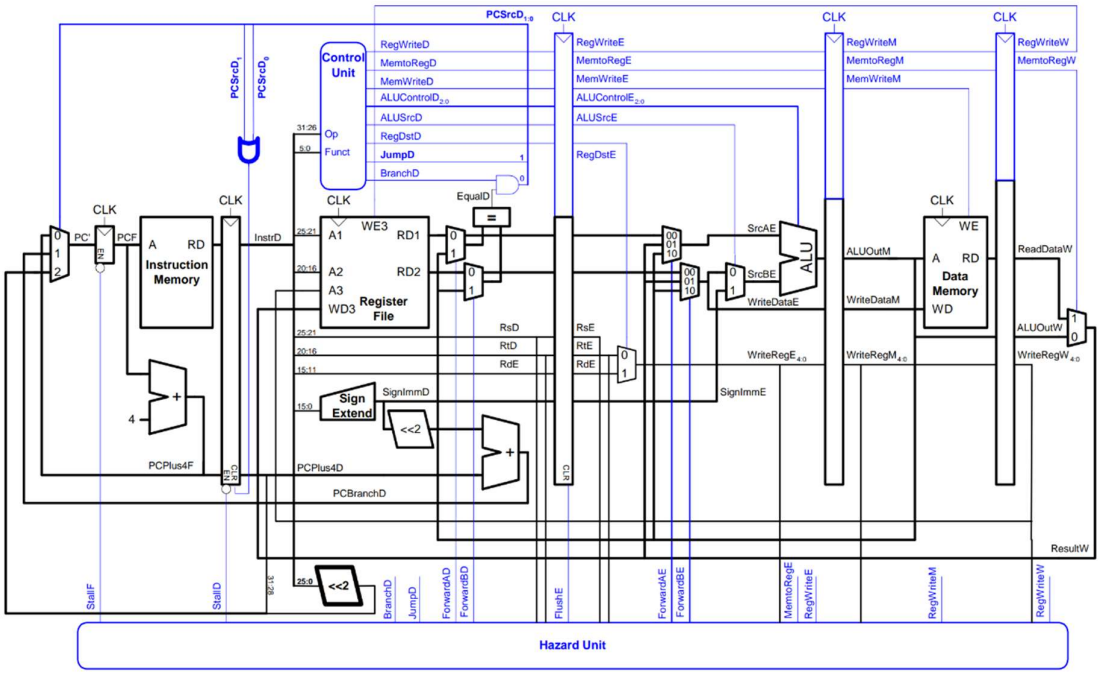

Figure 15 סכימת מערכת

להלן הRTL של המערכת לאחר סינתזה בתוכנת Quartus –

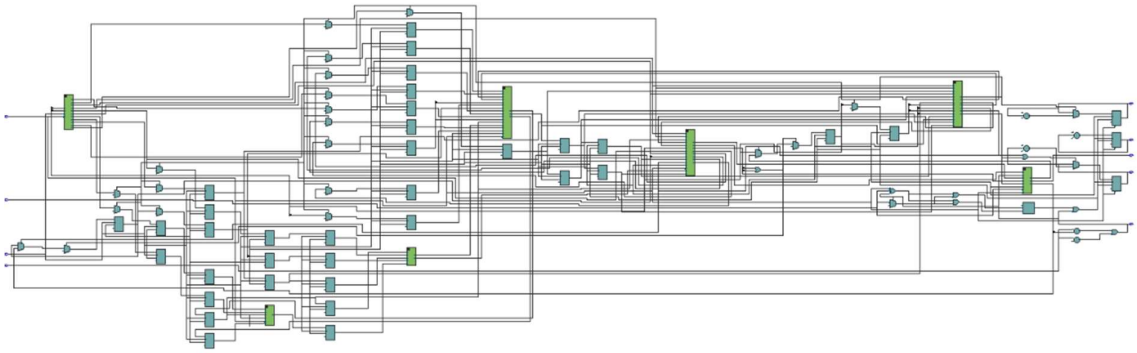

Figure 16 המערכת לאחר סינתזה בQuartus

כמו כן, הוספנו סיגנלי מוצא מתאימים לצורך דיבוג –

• CLKCNT – מונה כמות עליות השעון	• FHCNT – מונה כמות הflushים
• STCNT – מונה כמות הstallים	• BPADD – כתובת הBreakPoint

שלב הInstruction Fetch

סקירת פעולת המודול

בשלב זה נרצה להביא את הפקודה בכתובת הPC מהזיכרון של ההוראות. כמו כן, בחלק זה ישב הטיפול בPC הנבחר, מפני שבארכיטקטורה המצורפת קיימים תמיד מספר אופציות לשינוי הPC –

- נרצה להכניס לPC ערך של כתובת אליה נרצה לבצע jump
- נרצה להכניס לPC ערך של כתובת אליה נרצה לבצע branch
- $PC \leftarrow PC + 4$ כאשר נרצה לקדם את הפקודה הבאה

שרטוט הRTL

להלן הentity של המודול –

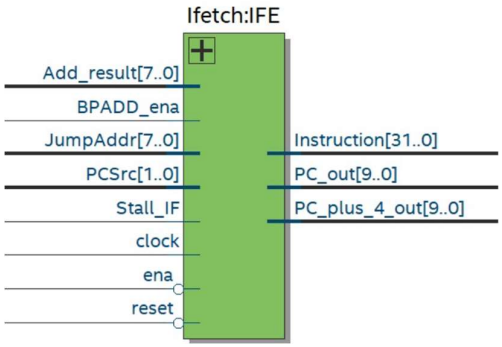

Figure 17 דיאגרמת בלוק של הIF

שרטוט הRTL של מודול זה –

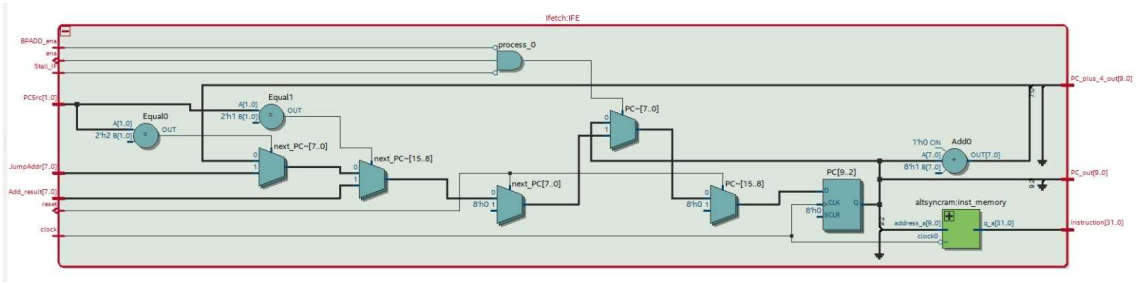

Figure 18 הRTL של הIF

מודול Instruction Decode

סקירת פעולת המודול

בשלב זה נרצה לקחת את הפקודה שהוראה משלב הfetch ובעצם לפרק אותה לפי הסוג של הפקודה באופן הבא –

Type	-31-			format (bits)		-0-
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

כך נכניס את כתובת הרגיסטרים המתאימים לRF שנמצא בשלב הdecode ואך נוכל להוציא את תוכן הרגיסטרים המתאימים.

כמו כן, בשלב זה נבצע את חישוב הbranch והjump אם אחד מהם מתקיים, מפני שבשלב זה כבר נוכל לדעת הן אם התבצע jump (לפי מבנה הפקודה) או branch לאחר השוואת תוכן הרגיסטרים בהנחה ומעודכן (נדאג לזה בHazard unit).

כמו כן, בשלב זה יושבת גם יחידת הבקרה שמקבלת את הOp והFunct (פשוט לא מתייחסת לקו זה אם הפקודה אינה מסוג R-type) והיא מוציאה קווי בקרה מתאימים לאופן הפקודה שהתקבלה. לדוגמא, אם התקבלה פקודת חיבור נצטרך להוציא קווים מתאימים לALUSrc, ALUctl, RegWrite וכו'.

שרטוט הRTL של הID

להלן הentity של המודול –

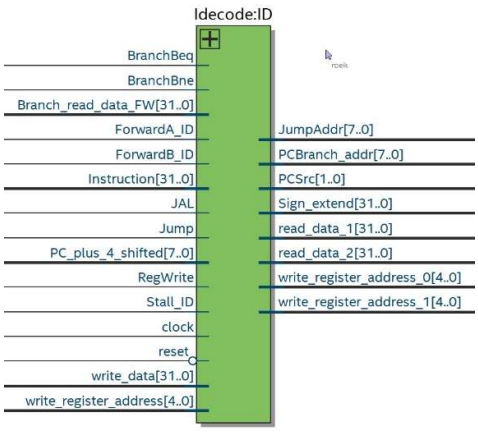

Figure 19 **דיגאגרמת בלוק של הID**

שרטוט הRTL של מודול ID –

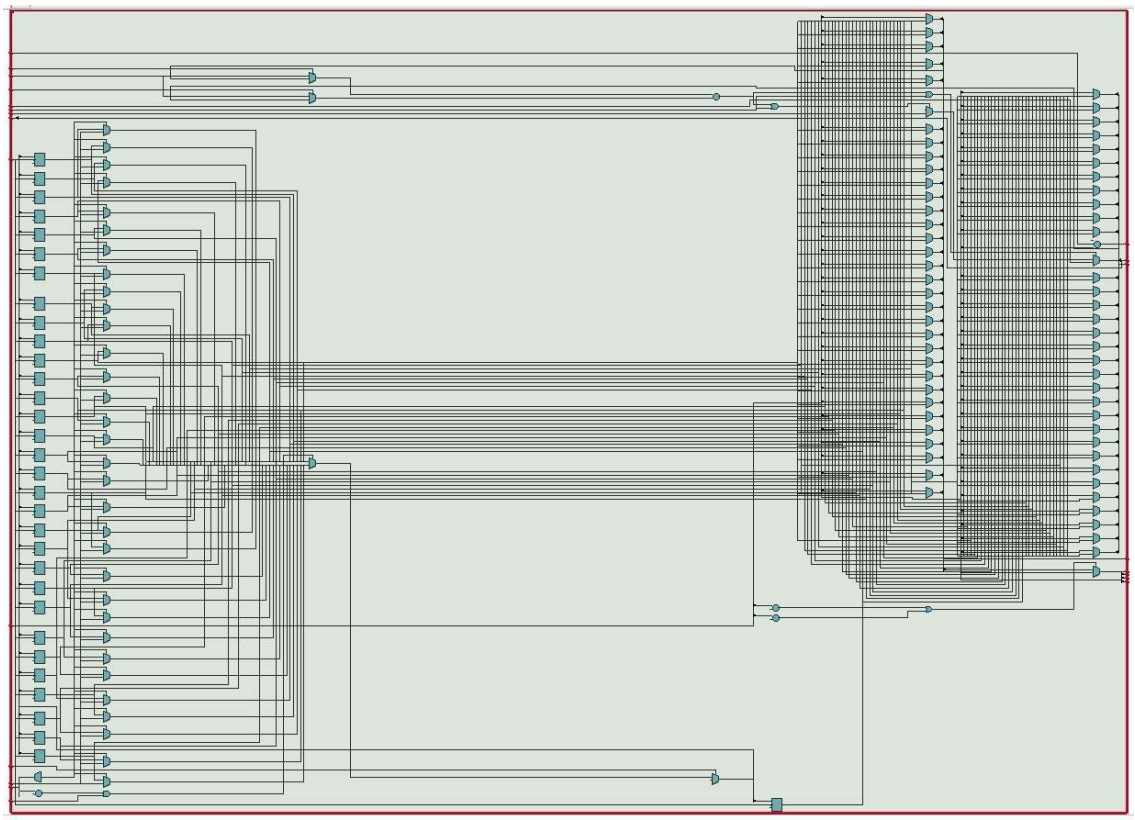

Figure 20 ה-RTL של ה-ID

שרטוט RTL של Control Unit

להלן הentity של המודול –

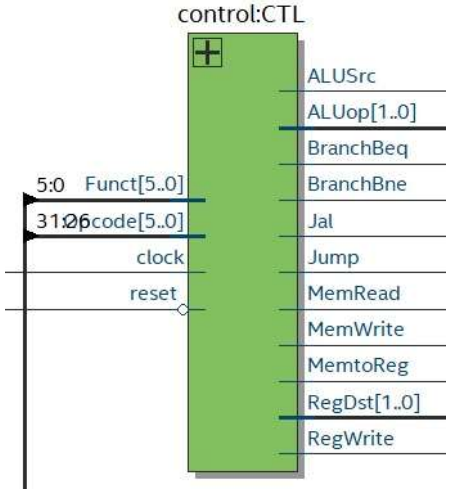

Figure 21 דיגאגרמת בלוק של הCTL

שרטוט הRTL של מודול CTL –

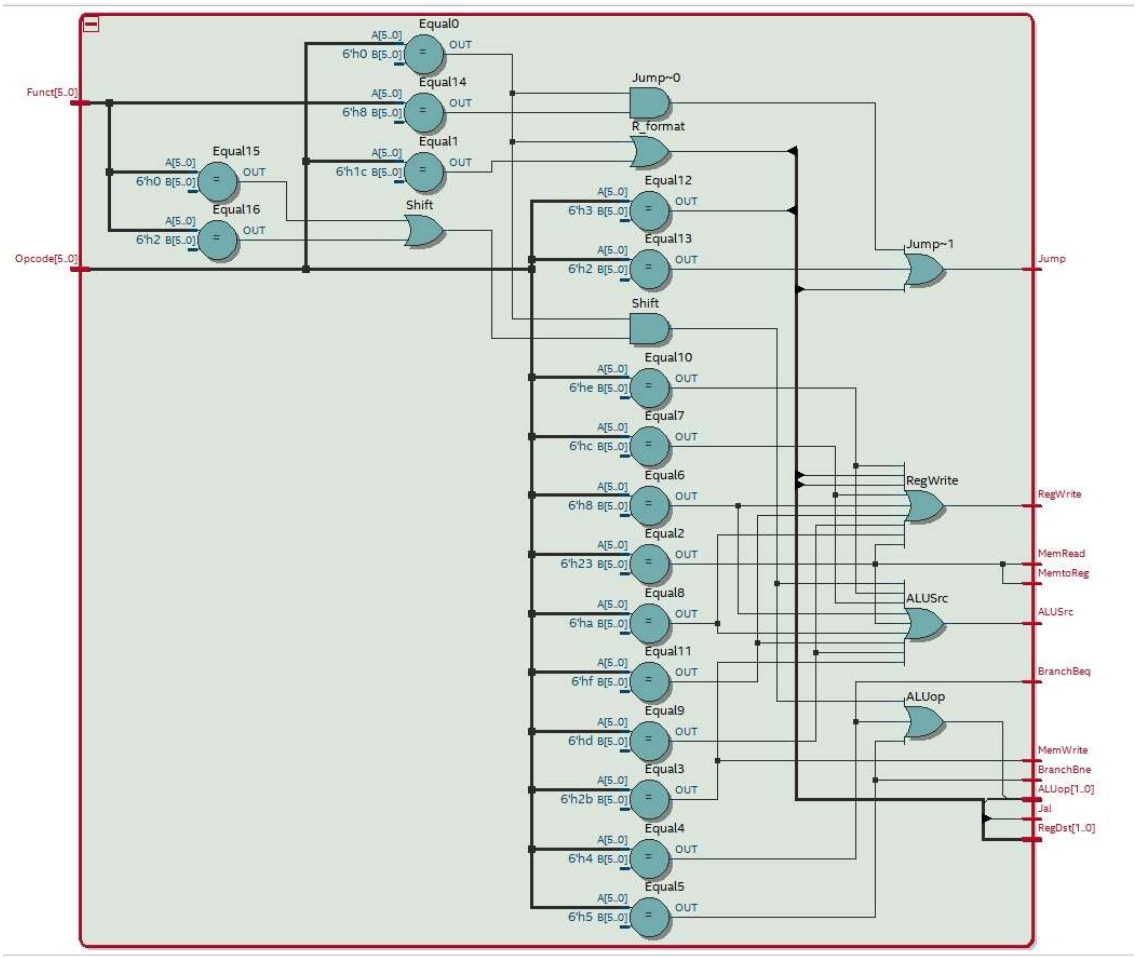

Figure 22 ה-RTL של ה-CTL

מודול Execute

סקירת פעולת המודול

שלב הביצוע, כאן אנחנו מבצעים פקודות לזיכרון כמו חישוב כתובת או ביצוע פעולה מסוימת. זה נעשה באמצעות פקודות אריתמטיות שונות שנעשות ברכיב הALU.

כאמור עבור המעבד single-cycle, בשלב הביצוע מבוצעת גם חישוב הכתובת PC הבאה ופעולת הBranch. בנוסף כאשר עוברים למעבד Pipelined, נראה כי פקודת הbranch מביאה איתי control hazard, המעבד לא יודע מה הפקודה הבאה לבצע עבורה fetch מפני שהחלטת הbranch עוד לא בוצעה בזמן שהפקודה הבאה בוצעה לה fetch. דרך אחת לפתור זאת היא ע״י stall אבל זה יגרום להורדה ביעילות המערכת. דרך אחרת היא לחזות אם פקודת הbranch צריכה לקרות או לא ולבצע את הפקודות בהתאם לחיזוי. ברגע שהחלטת הbranch ידועה, ניתן לבצע flush ולזרוק את הפקודות אם החיזוי שגוי. זה יגרום לקצת שיפור אבל עדיין ביצוע flush להרבה פקודות כאשר מבצעם branch, מוריד את יעילות המערכת.

דרך נוספת והיא הדרך שבחרנו שבה אפשר להוריד את הבזבוז סייקלים של פקודות מבוזבזות במקרה של חיזוי שגוי היא בכך שנבצע את החלטת הbranch בשלב הdecode ביחד עם חישוב הכתובת PC הבאה. ביצוע ההחלטה היא בעצם לבצע השוואה בין שני ערכי רגיסטר. אז בכך שהזזנו את ההשוואה וחישוב הכתובת PC הבאה לשלב אחד קודם, ייעלנו את ביצועי המערכת.

בשלב הביצוע בנוסף אנחנו בודקים האם ישנו צורך לבצע forwarding הנועד לפתור בעיות של data dependencies עבור פקודות R-type וLW/SW.

שרטוט הRTL

להלן הentity של המודול –

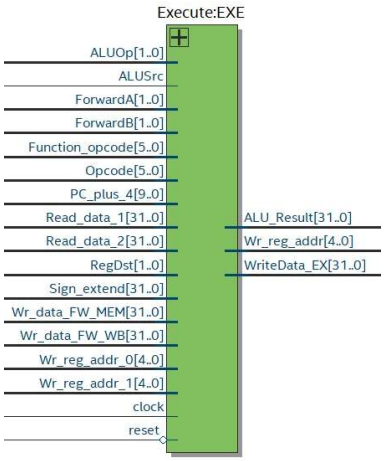

Figure 23 **דיגאגרמת בלוק של ה-***EXE*

שרטוט הRTL של מודול זה –

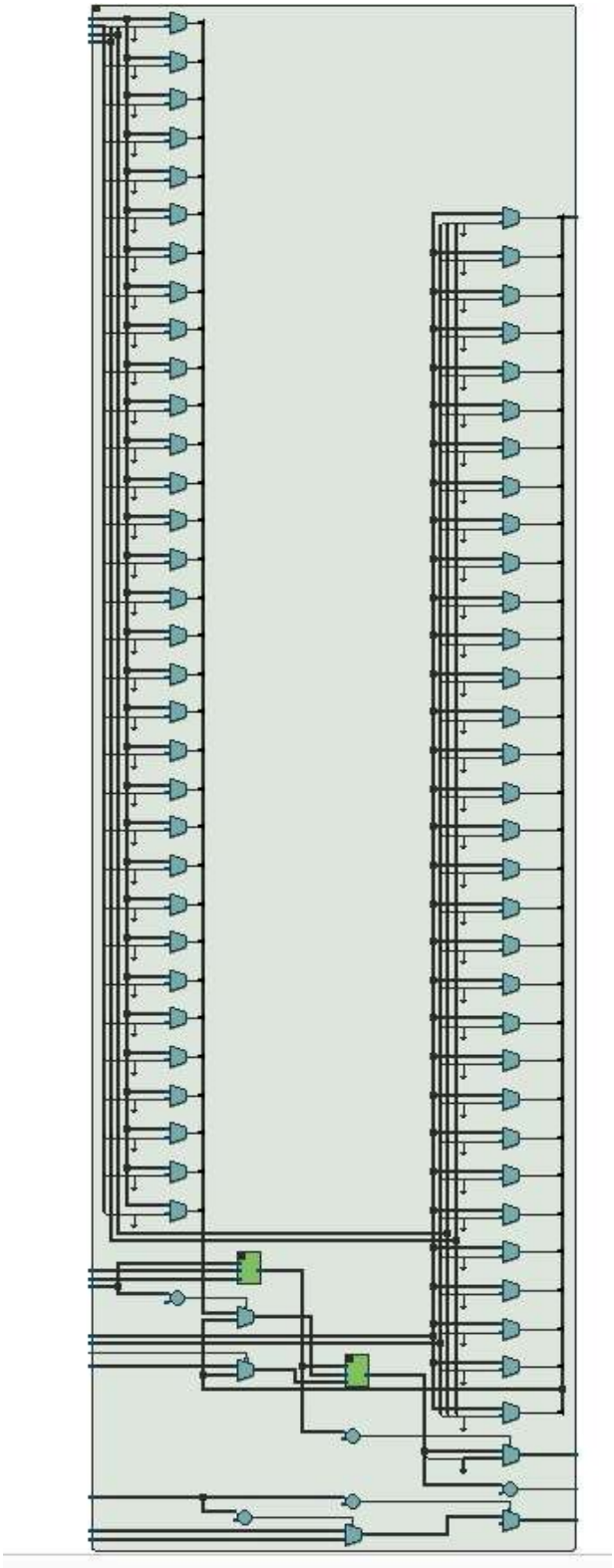

Figure 24 ה-RTL של ה-EXE

מודול Data Memory

סקירת פעולת המודול

מודול זה אחראי על הכתיבה והקריאה מתוכן הData Memory שזה זיכרון הRAM של המערכת שלנו. אנחנו במשימה השתמשנו בגודל RAM של 2^{12}.

שרטוט הRTL

להלן הentity של המודול –

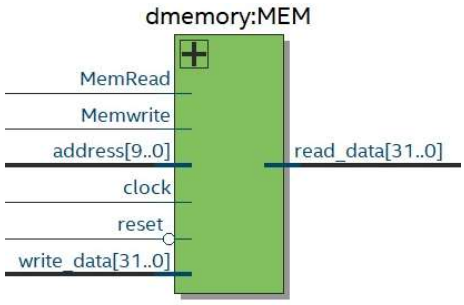

Figure 25 **דיאגרמת בלוק של ה-***DM*

שרטוט הRTL של מודול זה –

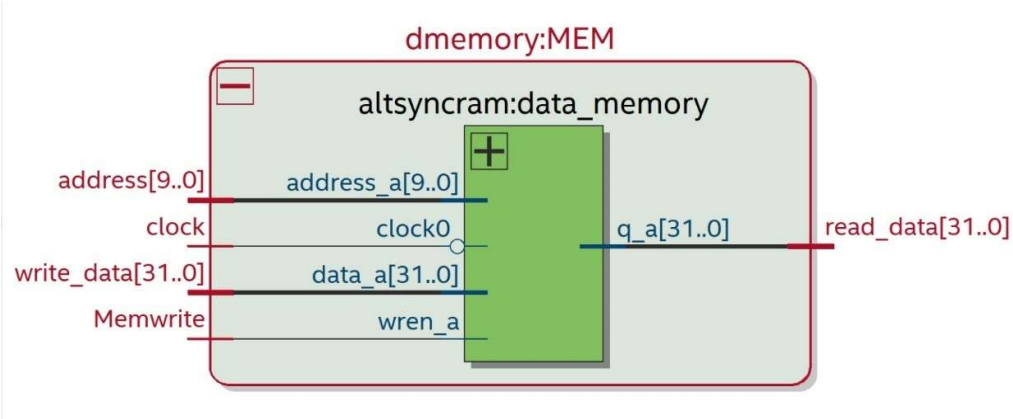

Figure 26 **ה-***RTL* **של ה-***DM*

מודול Write Back

סקירת פעולת המודול

מודול זה אחראי על ניהול החזרת המידע בסוף הpipeline והכתיבה שלו לזיכרון. קיימים הוראות שמבצעות שינוי ברגיסטרים כתוצאה מפעולות על רגיסטרים בALU וכתוצאה מפעולות מהזיכרון, לכן על מודול זה לנהל ולדעת איזה קו להחזיר ולכתוב אותו לרגיסטר.

שרטוט הRTL

להלן הentity של המודול –

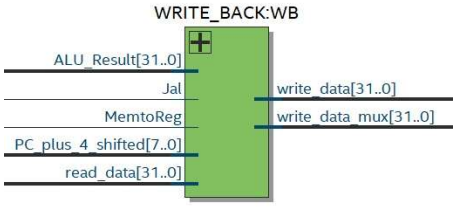

Figure 27 **דיגאגרמת בלוק של ה-*WB***

שרטוט הRTL של מודול זה –

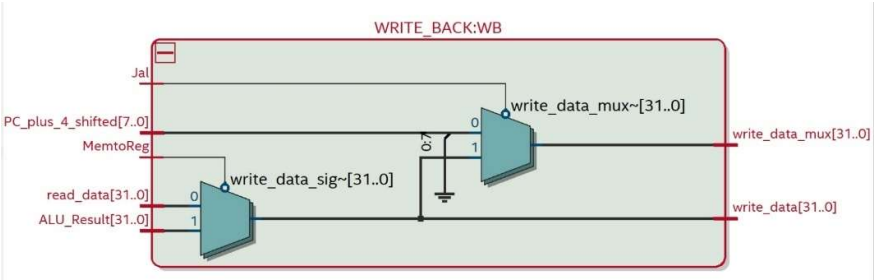

Figure 28 **ה-*RTL* של ה-*WB***

Signal Tap

נרצה לבצע ווריפיקציה של החומרה על ידי פונקציית ה-*signal tap* של ה-*quartus*. נתפוס בזמן אמת את מצב הסיגנלים של הרכיב, ובהתאם לסיגנל שאותו נרצה לתפוס, ברגע שהסיגנל ישתנה למה שאנחנו רוצים נקבל את תוצאות הסיגנלים שנדפיס למסך.

נשים את הסיגנלים כפי ששמנו ב*ModelSim* ונחלק אותם לקבוצות עבור כל שלב מה*Stages*. אנחנו נתפוס זמן עלייה עבור קו ה*RUN* שהוא הקו שנועד להפעיל את הרצת התכנית (כמו כפתור הריצה ב*MARS*). וגם נשים טריגר בעלייה עבור הטריגר של ה*Breakpoint*, כלומר כאשר נרצה לעצור בכתובת מסוימת קו ה*ST* טריגר יעלה ל-1. ב*KEY0* חיברנו את קו ה*RESET* וב-*KEY3* את קו ה*enable*. נזכור שהם במצב *pull-down*. כאשר נלחץ על קו ה-*ena*, אנחנו נעלה את *RUN* ל-1 והתכנית תרוץ עד לרגע שבוא נלחץ על כפתור ה*reset*.

בנוסף נשים את תנאי הלכידה כ*Basic Or*, כלומר שרק מספיק שאחד משתנה ולא כל הסיגנלים.

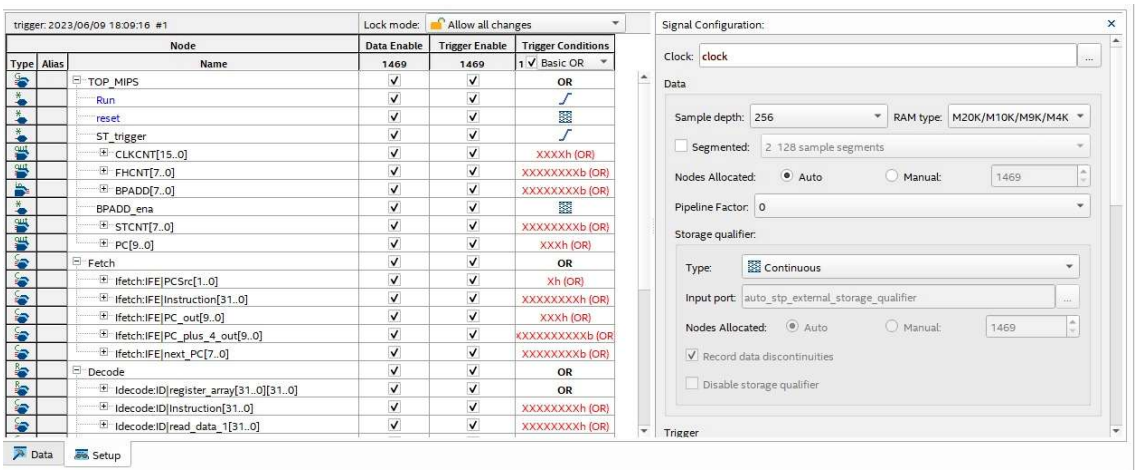

Figure 29 Signal Tap Configuration

נראה הרצה לדוגמא :

Figure 30 הרצת התכנית בסיגנל טאפ

כלומר אנחנו רואים, שברגע שלחצנו על *KEY3*, התכנית התחילה לרוץ, ואנחנו נוכל לראות את הפקודות הראשונות בתכנית בהתאם לגודל החלון שנוכל להבחין בו. אנחנו רואים גם את ספירת עליות השעון כאשר המונה עולה ב-1 בכל עליית שעון.

נראה ביותר בירור את הדוגמא למונה ה*STALL*

Figure 31 **הדגמת** *STALL*

כלומר אנחנו רואים שכאשר קיבלנו *stall*, המנייה עלה ל-1 ובנוסף הפקודה ב*fetch* קיבלה השהייה של מחזור אחד.

נראה כעת כיצד נראה ה*REG FILE* לאחר סיום התכנית (נזכור כי הייתה בעיה בלראות את הזיכרון ב*quartus* ולכן הדרישה הזו נפלה מהמעבדה, ואת בדיקת תוכן הזיכרון נבצע רק ב*modelsim*)

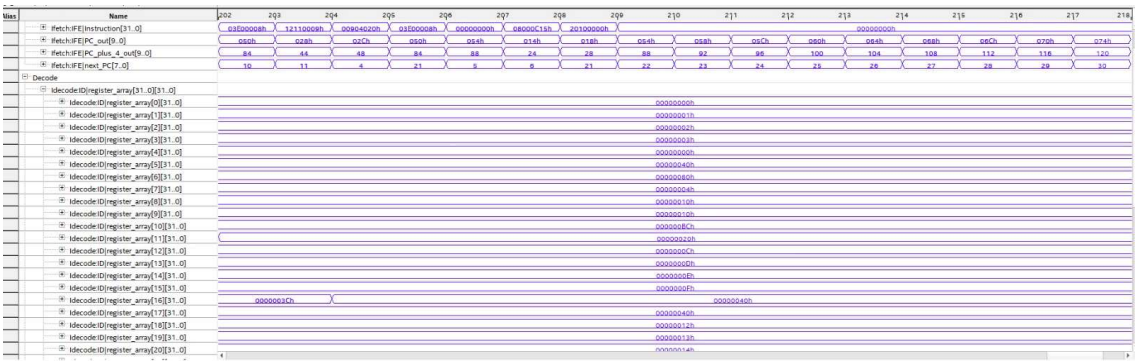

Figure 32 **תוכן** *REG FILE* **בסיום התכנית**

נציין כי לאחר סיום התכנית, התכנית לא ממשיכה בלולאה אינסופית אלא מגיעה לפקודת NOP וה*PC* ימשיך לגדול עד שנרסט את התכנית. אפשר לפתור זאת באמצעות לולאה אינסופית עם פקודות j/branch אבל זה לא קריטי.