LAB 5

מגישים :

- יגל בן צבי 208584615
- רון בניטה 314882317

מחלקה : המחלקה להנדסת חשמל

קורס : מעבדת ארכיטקטורה מתקדמת ומאיצי חומרה - 361.1.4693

אוניברסיטת בן-גוריון בנגב
Ben-Gurion University of the Negev

הקדמה

מטרת המעבדה היא לתכנן, לממש ולאמת מעבד מסוג MIPS בארכיטקטורת צינור (Pipelined), התואם ל־ISA של MIPS32. המעבד נבנה תחילה כמעבד חד־מחזורי, ולאחר מכן שודרג למעבד בעל חמישה שלבים בצינור (IF, ID, EX, MEM, WB), תוך שילוב יחידות לזיהוי סכנות מידע (Hazards) ויחידות להעברת קדימה (Forwarding).

במהלך הפרויקט נדרשנו לממש את המעבד בשפת VHDL, לבצע סימולציה פונקציונלית באמצעות ModelSim, ולסנתז את התכנון ליישום על גבי לוח FPGA מתוצרת Altera. כמו כן, נעשה שימוש בכלי SignalTap לצורך איתור תקלות ובדיקת ביצועים בזמן אמת.

סקירת מבנה עליון של המערכת

המערכת מבוססת על ארכיטקטורת MIPS בצינור בעל חמישה שלבים (Pipeline) :

1. **IF – Instruction Fetch**
2. **ID – Instruction Decode and Register Read**
3. **EX – Execute / ALU Operations**
4. **MEM – Memory Access**
5. **WB – Write Back**

בין כל שלב קיימים רגיסטרים שלבים (Pipeline Registers) שמפרידים את השלבים ושומרים את הערכים הדרושים להמשך עיבוד נכון.
בנוסף, המבנה כולל את היחידות הבאות:

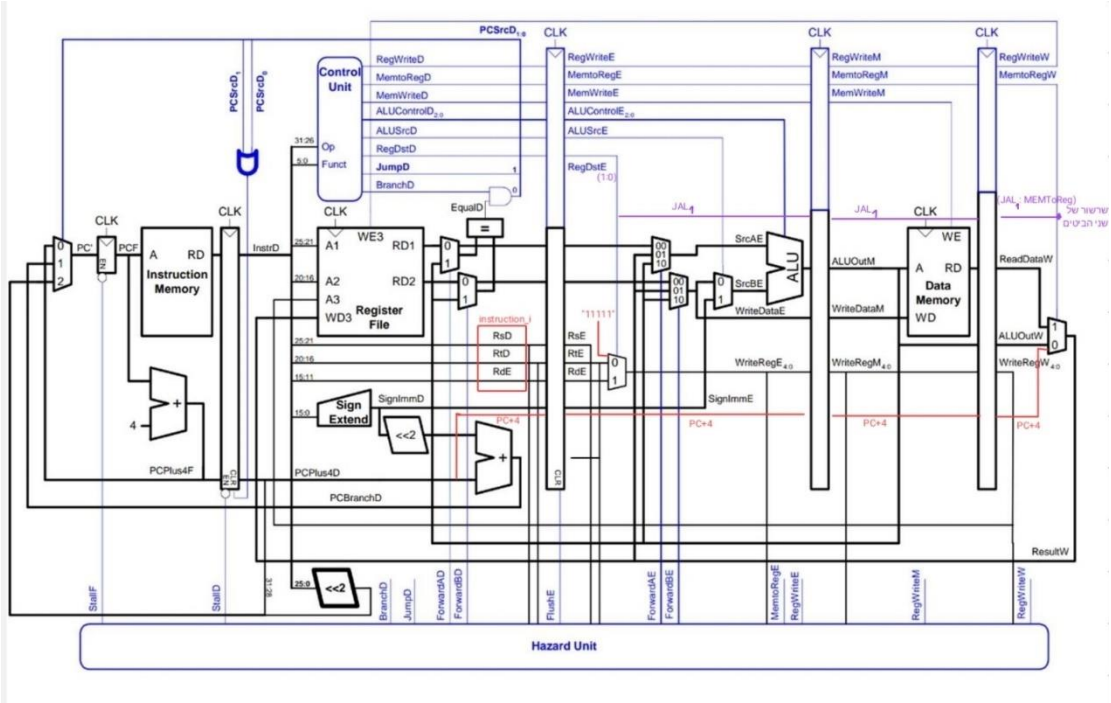
- **Register File**- כולל שלושה פורטים : שניים לקריאה ואחד לכתיבה.
- **ALU**- מבצעת חישובים אריתמטיים ולוגיים.
- **Data Memory**- זיכרון לקריאה וכתיבה.
- **Control Unit**- קובעת את אותות השליטה על פי פקודת ה־Opcode ו־Funct.
- **Hazard Detection Unit**- מונעת קונפליקטים על ידי איתותי Stall או Flush.
- **Forwarding Unit**- מנתבת ערכים ישירות בין שלבים EX/MEM/WB כדי למנוע עיכובים מיותרים.
- **Jump & Branch Logic**- כולל זיהוי תנאים והתאמות ל־PC בהתאם.

לצורכי בדיקות וביצועים, המערכת כוללת גם ממשק לשליטה מה־FPGA :

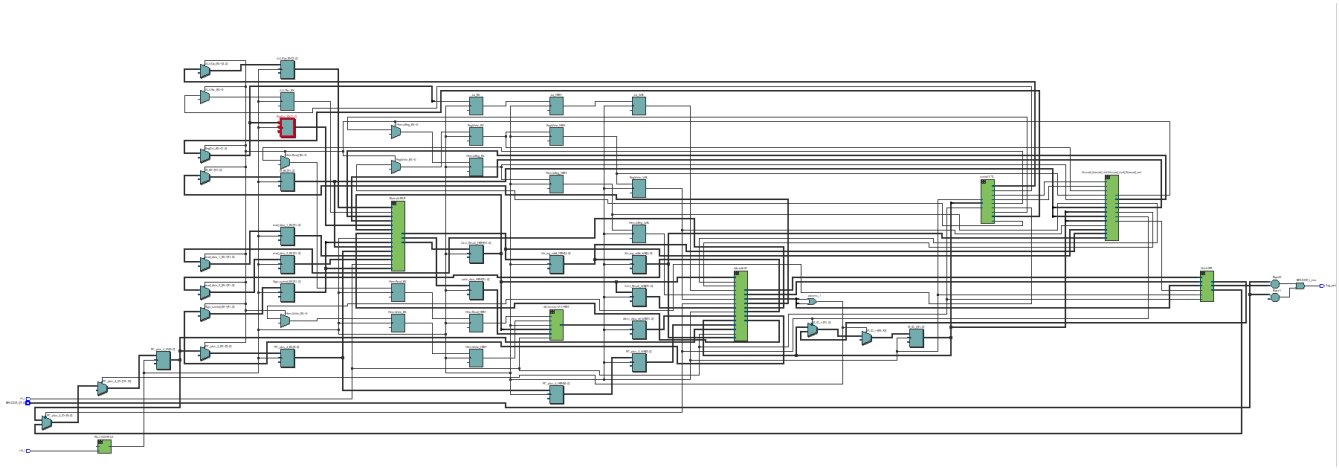
- **Reset (KEY0)**- אתחול סינכרוני שמאפס את ה־PC.
- **BPADDR_i**- לצורך Triggering ב־SignalTap.
- **STCNT_o, FHCNT_o**- מונים את מספר ה־Stalls וה־Flushes (בהתאם להגדרות IPC).

המערכת כולה פועלת על בסיס שעון יחיד (CLK)ומיישמת הפרדת זיכרונות (Harvard Architecture), עם הפרדה מלאה בין Instruction Memory ל־ Data Memory.

מצורף שרטוט המערכת:



RTL:



בדיקת המערכת

לפני שצרבנו את הקוד לבקר בדקנו שהתכנית שכתבנו באמת עובדת.
לקחנו קוד בשפת C והפכנו אותו לקוד אסמבלי.

```
.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
M:      .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, M       # $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*M*M

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


finish:
```

```
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
}
```

בבדיקת הביצועים הרצנו קובץ אסמבלי עם פקודות שנמצאות בISA של המעבד שפיתחנו במעבדה זו. קובץ האסמבלי מממש חיבור מטריצות MAT1,MAT2 שיושבות בזיכרון ומכניסה את התוצאות למטריצת יעד שנמצאת גם כן בזיכרון.

לאחר הרצה ב-MARS קיבלנו את הערכים הבאים בזיכרון.

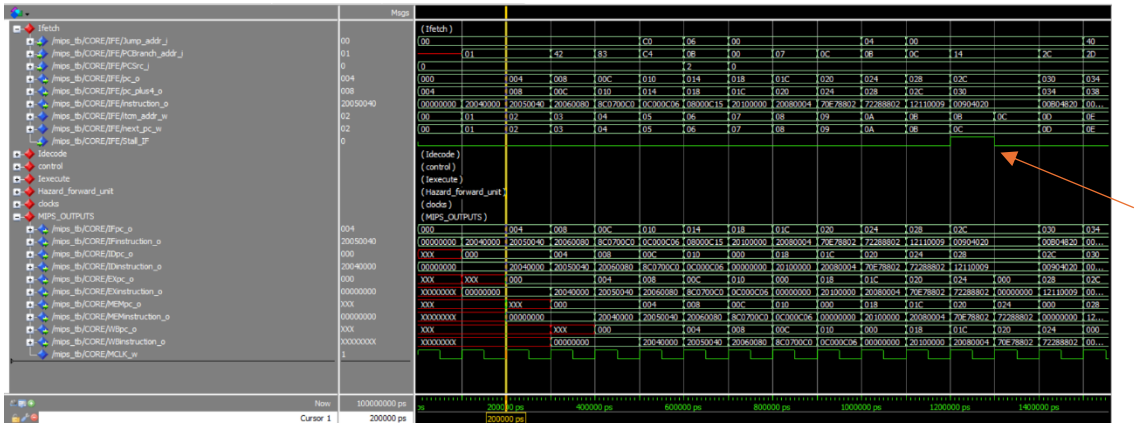
Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000001	0x00000002	0x00000003	0x00000004	0x00000005	0x00000006	0x00000007	0x00000008
0x10010020	0x00000009	0x0000000a	0x0000000b	0x0000000c	0x0000000d	0x0000000e	0x0000000f	0x00000010
0x10010040	0x00000001	0x00000002	0x00000003	0x00000004	0x00000005	0x00000006	0x00000007	0x00000008
0x10010060	0x00000009	0x0000000a	0x0000000b	0x0000000c	0x0000000d	0x0000000e	0x0000000f	0x00000010
0x10010080	0x00000002	0x00000004	0x00000006	0x00000008	0x0000000a	0x0000000c	0x0000000e	0x00000010
0x100100a0	0x00000012	0x00000014	0x00000016	0x00000018	0x0000001a	0x0000001c	0x0000001e	0x00000020
0x100100c0	0x00000004	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

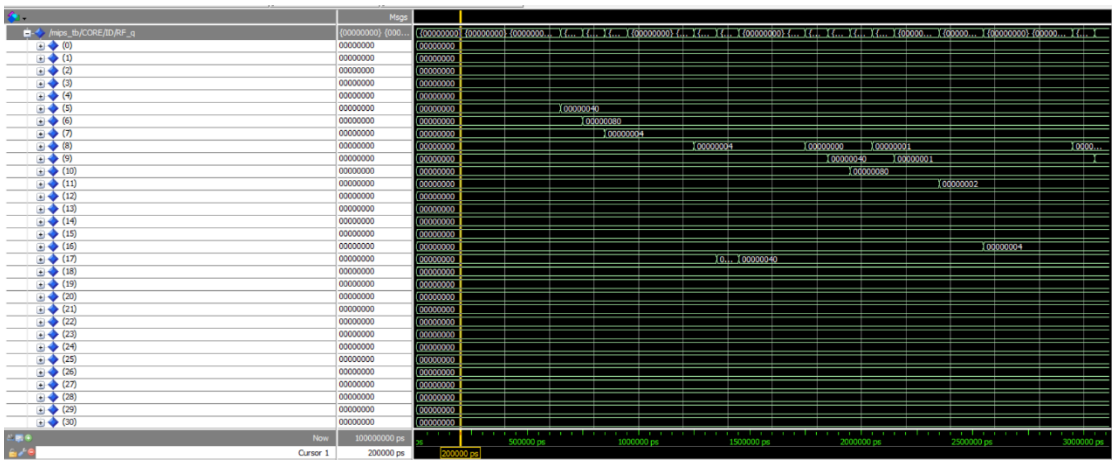
הערה – כתובת $0x100100c0$ מחזיקה את גודל המטריצה הריבועית ואכן נראה שזה 4 כי המטריצות הן 4×4.

את קובץ האסמבלי הרצנו על הקוד שלנו באמצעות MODELSIM כדי לוודא את תקינות המערכת שתכננו, נוודא שהתכנית עובדת כראוי ונשווה בין הערכים בזיכרון של בסימולציה לבין של ה-MARS.

ניתן לראות שהמערכת רצה כמו שצריך, ה-PC מתקדם ב-4 כל מחזור שעון בהתאם ל-ISA. בנוסף מסומן חץ שמציג את מצב stall, מצב בו ה-PC נשאר באותה כתובת ולא מתקדם זמנית ולאחר מכן מממשיך כרגיל.



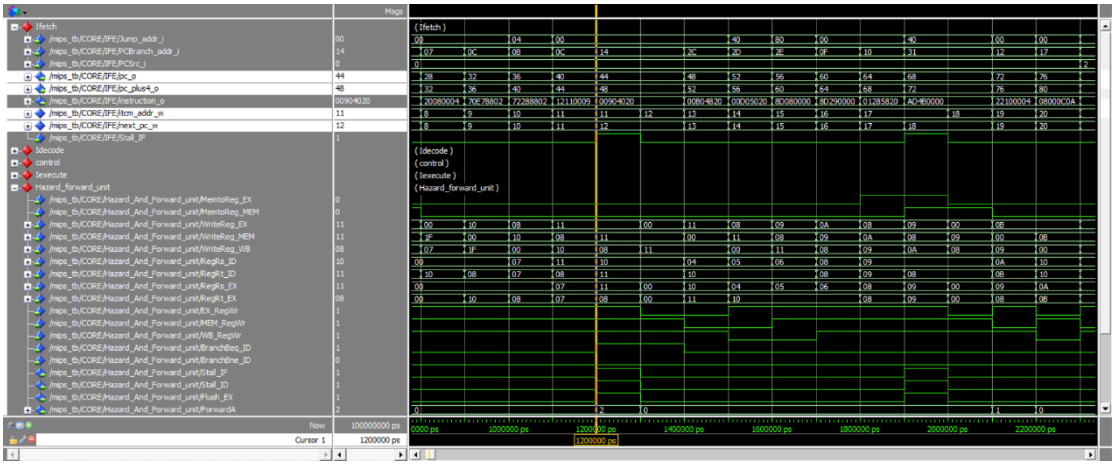
בתרשים גלים הבא מוצגים הרגיסטרים של המערכת.
ניתן לראות שרגיסטר 4 נטען בערך 0, רגיסטר 5 בערך $0x40$.
התרדים מציג את ההתנהגות המקבילה של הקוד אסמבלי שטוען לרגיסטרים כתובות בזיכרון. כך אנחנו מתקבלים התנהגות תקינה של שמירת ערכים בזיכרון.



דוגמה לפקודה באסמבלי שמבצעת stall

0x0040003c	0x00904020	add $8,$4,$16	27: add	$t0, $a0, $s0	# find Mat1 pointer with offset
0x00400040	0x00b04820	add $9,$5,$16	28: add	$t1, $a1, $s0	# find Mat2 pointer with offset
0x00400044	0x00d05020	add $10,$6,$16	29: add	$t2, $a2, $s0	# find resMat pointer with offset

ניתן לראות את ה-STALL עולה בכתובת הנכונה.



בנוסף ניתן לראות בזיכרון של הסימולציה כי אנחנו מקבלים תוצאה זהה לתוצאה שקיבלנו ב-MARS רק שויזואלית זה בסדר הפוך.

0000002f	00000020	0000001E	0000001C	0000001A	00000018	00000016	00000014	00000012
00000027	00000010	0000000E	0000000C	0000000A	00000008	00000006	00000004	00000002
0000001f	00000010	0000000F	0000000E	0000000D	0000000C	0000000B	0000000A	00000009
00000017	00000008	00000007	00000006	00000005	00000004	00000003	00000002	00000001
0000000f	00000010	0000000F	0000000E	0000000D	0000000C	0000000B	0000000A	00000009
00000007	00000008	00000007	00000006	00000005	00000004	00000003	00000002	00000001

קומפילציה ב-Quartus

לאחר שווידאנו שהמערכת שבנינו עובדת כראוי בסימולציה נעבור לשלב הבא והיא צריבת המערכת על גבי ה-FPGA.

ראשית אנחנו נמצא את התדר המקסימלי של המערכת, על מנת למצוא תדר מקסימלי של המערכת נוסיף את קובץ ה SDC הבא:

```
# Define the main clock on clk_i pin
create_clock -name clk_i -period 20.0 [get_ports clk_i]

# Ignore altera_reserved_tck to avoid confusion
set_false_path -from [get_ports altera_reserved_tck]

# Example input and output delays if needed:
# set_input_delay -clock clk_i 2.0 [all_inputs]
# set_output_delay -clock clk_i 2.0 [all_outputs]
```

הרגיסטרים של ה-pipeline מקודמים על שעון ולכן נוכל לומר כי המערכת שלנו היא סינכרונית, דבר זה מאפשר לנו למצוא את התדר המקסימלי של המערכת.

Slow 1100mV 85C Model Fmax Summary

<<Filter>>

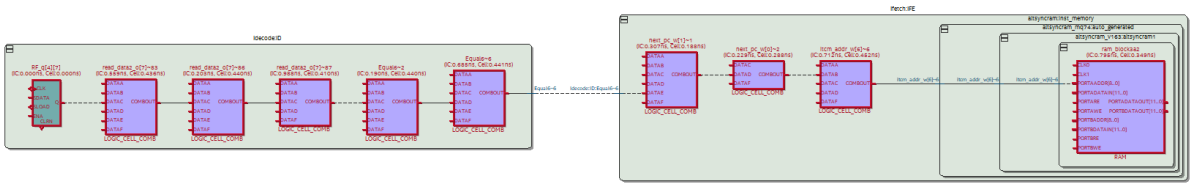
	Fmax	Restricted Fmax	Clock Name
1	60.05 MHz	60.05 MHz	clk_i

Slow 1100mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	62.24 MHz	62.24 MHz	clk_i	

בנוסף מצורף הנתיב הקריטי של המערכת שלנו.



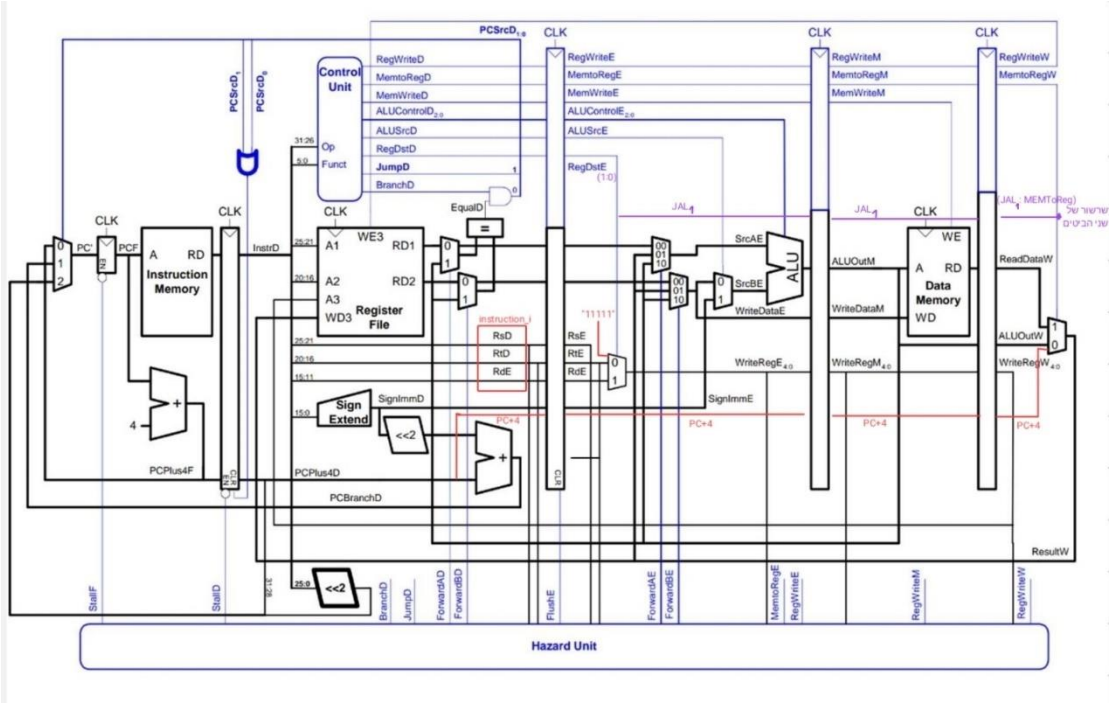
הנתיב שלנו הוא בין ה-ID לבין ה-IF. הוא מציג חישובים קומבינטורים, פענוח פקודה, חישוב ה-pc וגישה לזיכרון RAM.

לוגיקה עבור המודולים השונים

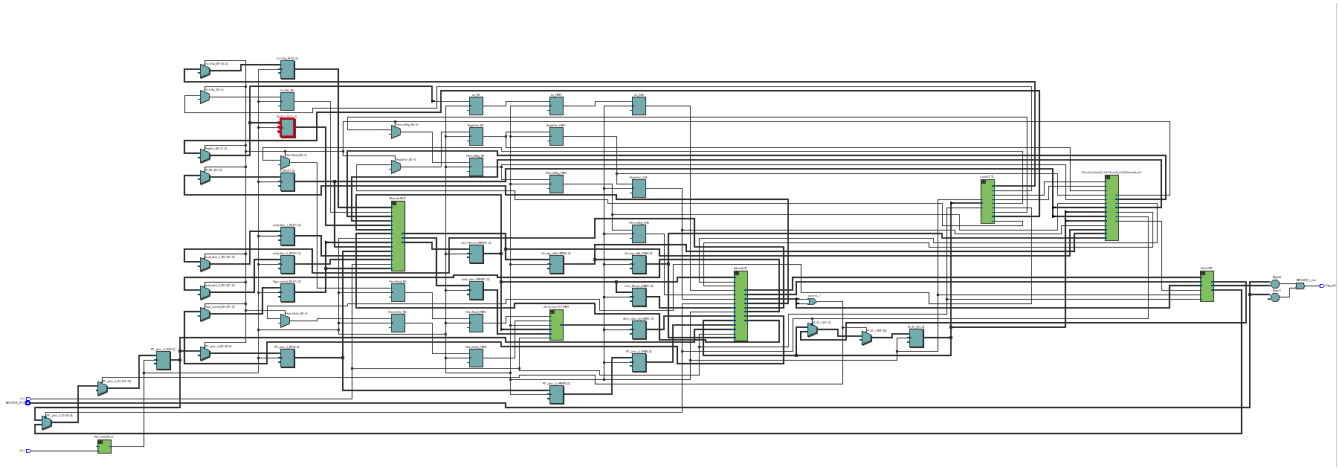
	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits	DSP Blocks	Pins	Virtual Pins
1	|MIPS	1689 (9)	1707 (324)	16384	1	11	0
1	|Execute:EXE|	609 (415)	0 (0)	0	1	0	0
2	|Hazard_forward_Unit:Hazard_And_Forward_unit|	26 (26)	0 (0)	0	0	0	0
3	|Idecode:ID|	795 (795)	1120 (1120)	0	0	0	0
4	|Ifetch:IFE|	82 (35)	79 (13)	8192	0	0	0
5	|PLL:\G0:MCLK|	0 (0)	0 (0)	0	0	0	0
6	|control:CTL|	10 (10)	0 (0)	0	0	0	0
7	|dmemory:\G1:MEM|	47 (0)	65 (0)	8192	0	0	0
8	|sld_hub:auto_hub|	111 (1)	119 (0)	0	0	0	0

נסביר עבור כל מודל איך הוא עובד ונצרף שרטוט של כל מודל.

מצורף שרטוט המערכת:



RTL לאחר סינטזה:



מודל FETCH

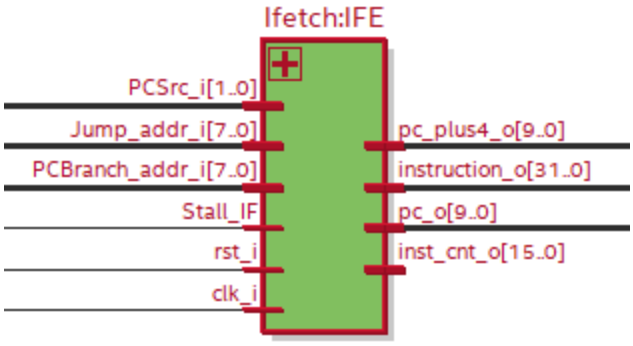
בשלב זה, המעבד מבצע את הבאת הפקודה (Instruction) מה־Instruction Memory על פי ערך מונה הפקודות (PC). ערך זה מכתיב איזו פקודה תתבצע במחזור הבא, ולכן חיוני לעקוב אחרי כל תנאי שמשפיע על עדכון ה־PC.

סקירת פעולת המודול:

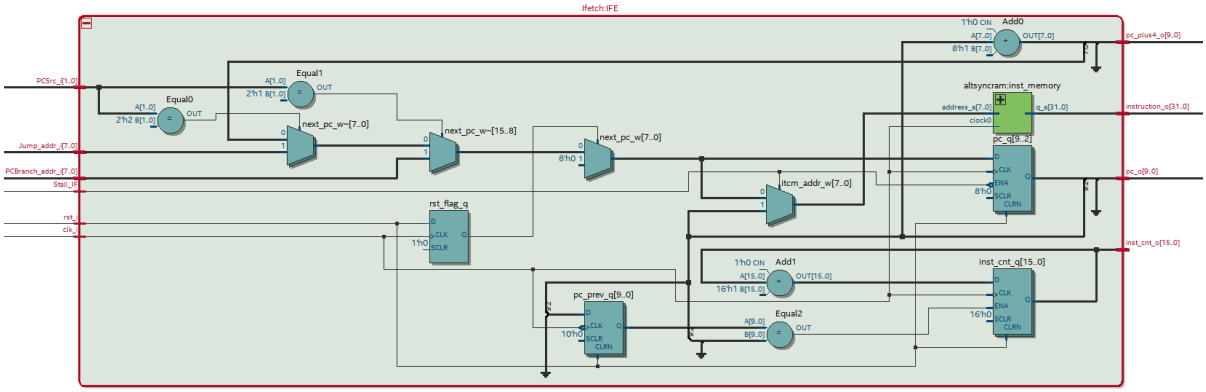
מנגנון העדכון של ה־PC מורכב מהתנאים הבאים:

- **פקודת Jump**: כאשר מזוהה פקודת jump, ה־PC מקבל את הכתובת שאליה יש לקפוץ (לדוגמה: $PC \leftarrow \{PC + 4[31:28], instr\_index, 2'b00\}$).
- **פקודת Branch מותנית**: במידה ומתקיים תנאי סניף (כמו BEQ, BNE) מתקבלת החלטה אם לעדכן את ה־PC בכתובת היעד של הסניף. זאת על בסיס השוואה בין רגיסטרים, ועל פי אות ה־Branch וה־Zero.
- **ברירת מחדל – ריצת תוכנה רגילה**: כאשר לא מתקיימים התנאים לעיל, ה־PC פשוט גדל ב־4, כלומר: $PC \leftarrow PC + 4$.

שרטוט RTL של ה-entity.



שרטוט של המודל



אם קיימת סכנה (Hazard) שדורשת עצירה של שלב ה־Fetch, אזי PC לא מתעדכן, וערכו נשמר. זה קורה כאשר יחידת הזיהוי (Hazard Detection Unit) שולחת $StallF = 1$.

מודל INSTRUCTION DECODE

בשלב זה, מתבצע פיענוח של הפקודה שהובאה בזיכרון בשלב ה־Fetch. מטרת שלב זה היא להבין מה סוג הפקודה ומה יש לבצע במחזורי השעון הבאים, וכן להביא את הערכים המתאימים מתוך רשימת הרגיסטרים.

פורמט הפקודה:

בארכיטקטורת MIPS קיימים שלושה סוגי פקודות עיקריים:

תיאור כללי	שדות	Type
פקודות אריתמטיות/לוגיות בין רגיסטרים	opcode, rs, rt, rd, shamt, funct	R
גישה לזיכרון, סניפים ועוד	opcode, rs, rt, immediate	I
פקודות קפיצה(jump)	opcode, address	J

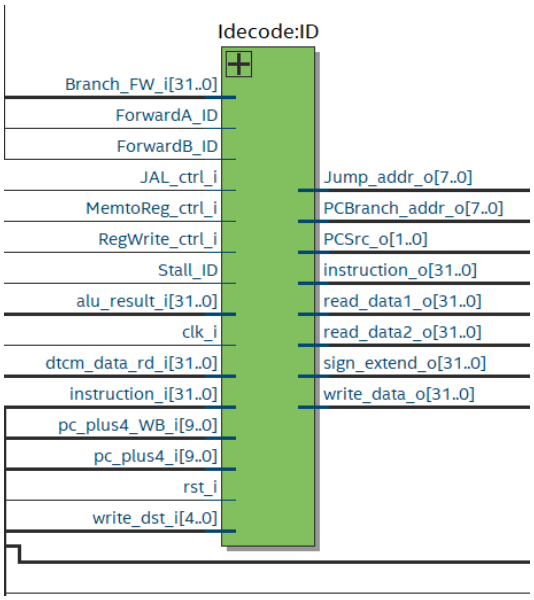
מה מתבצע בשלב זה:

- שליפת הערכים מתוך רשימת הרגיסטרים (RF) לפי השדות rs ו־rt.
- קבלת שדות הפקודה : opcode, funct, immediate, rd, rt, rs.
- שליחה של opcode ו־ funct ליחידת הבקרה (Control Unit), כדי לקבוע אותות שליטה בהמשך הצינור.
- הרחבת Immediate ל־32 סיביות בעזרת Sign Extend.
- קביעת אם יש צורך בעדכון PC כתוצאה מ־branch או jump, כולל ניתוח תנאים.

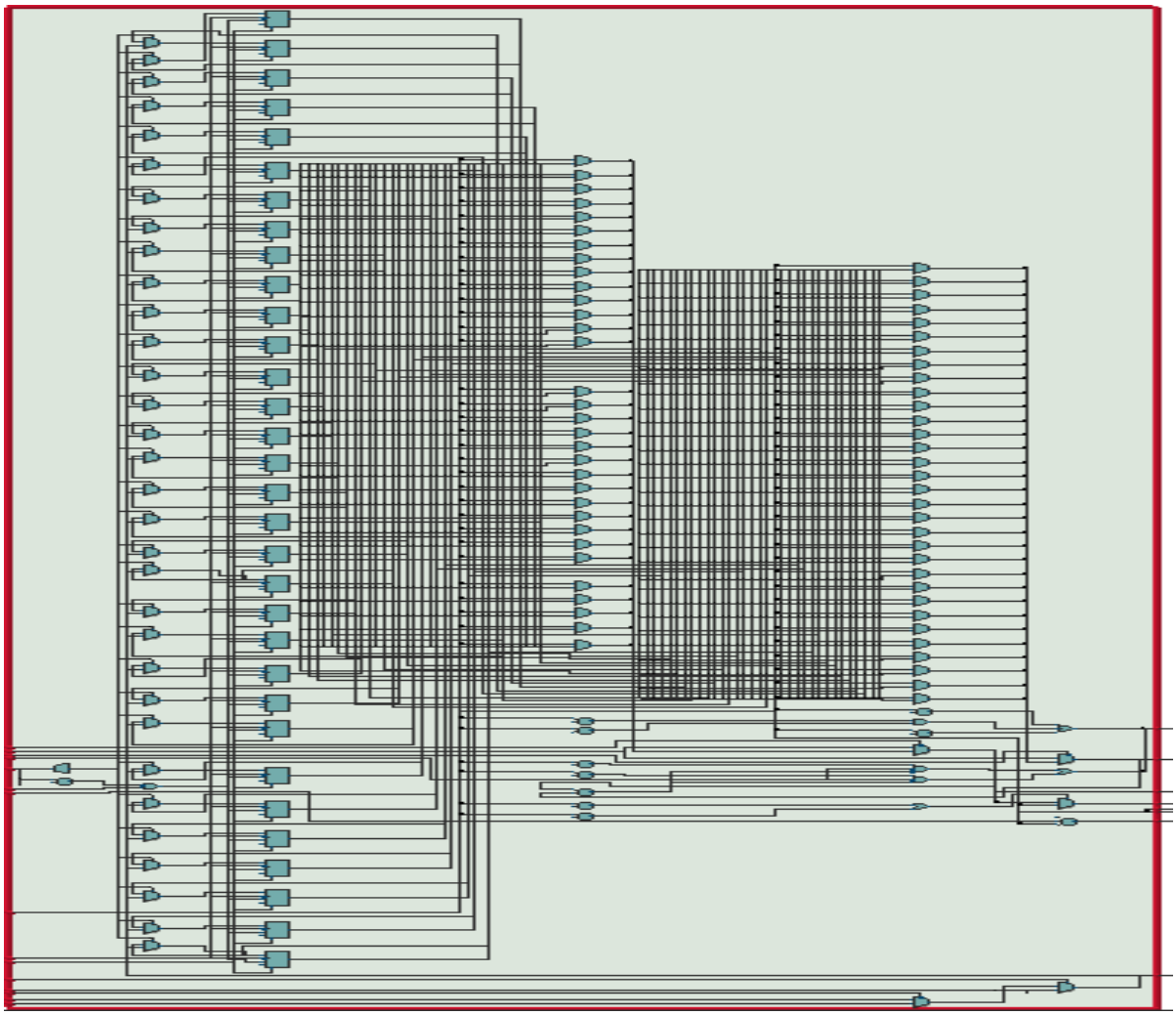
התמודדות עם פקודות מותנות:

- אם מדובר בפקודת branch, מתבצעת בדיקה בין הערכים שהובאו מ־RF.
- במידה ומתקיים תנאי הסעיף (zero = 1), ייתכן שנידרש לבצע flush בשל branch misprediction.

שרטוט RTL של ה-entity.



שרטוט של המודל



מודל EXECUTE

בשלב זה אנו מבצעים את הפעולה האריתמטית או הלוגית של הפקודה, כמו חיבור, חיסור, AND, OR, או חישוב כתובת לזיכרון. פעולות אלו מתבצעות על ידי יחידת ה־ALU של המעבד.

מטרות שלב EX:

- ביצוע פעולה אריתמטית/לוגית על פי פקודת הפעלה
- חישוב כתובת גישה לזיכרון (בפקודות lw, sw)
- חישוב כתובת קפיצה עבור פקודות branch
- איתור סכנות מידע (data hazards) ופתירתן בעזרת Forwarding

הבדל מול מעבד חד־מחזורי (single-cycle):

במעבד חד־מחזורי, חישוב תנאי הסניף מתבצע מיידית, וה־PC מתעדכן מיד.
במעבד בצינור, לעומת זאת, שלב חישוב תנאי ה־branch מתרחש ב־EX, ולכן אם התנאי מתקיים, יש צורך לבצע flush לפקודות שכבר התחילו להיטען.

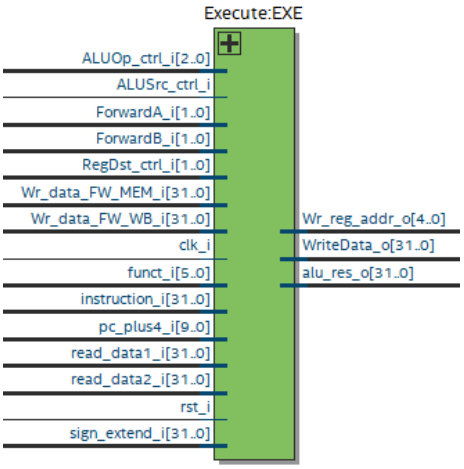
טיפול ב־branch hazard:

- הפלט של ALU (Zero) קובע האם התנאי מתקיים.
- במידה וכן, יחידת ה־control שולחת Flush כדי למחוק את הפקודה שנכנסה ל־IF או ID לפני שנודע שהתנאי אמיתי.
- זה גורם ל־1 delay slot.

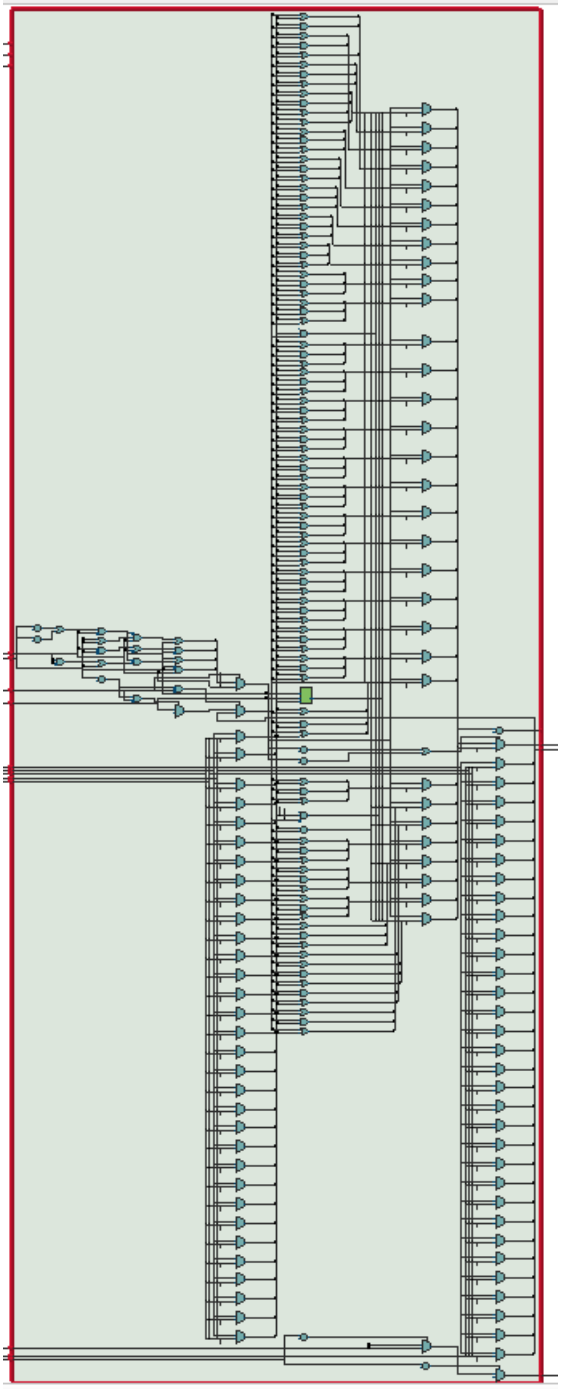
טיפול ב־data hazard:

- כאשר יש תלות בין פקודה ב־EX לפקודה קודמת שעדיין לא סיימה כתיבה ל־register, נעשה שימוש ב־Forwarding Unit.
- לדוגמה: אם פקודת add תלויה ב־lw שנמצאת ב־MEM, נזהה זאת בעזרת יחידת ה־Hazard ונעכב את הפקודה או נעביר לה ערך נכון ממחזור קודם.

שרטוט RTL של ה-entity.



שרטוט של המודל



מודל MEMORY

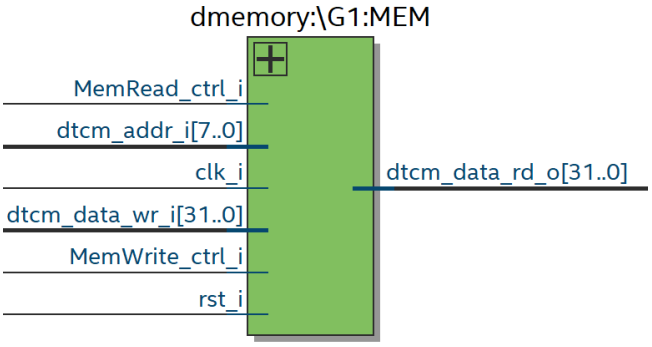
בשלב זה, המעבד מבצע גישה לזיכרון נתונים (Data Memory), בהתאם לסוג הפקודה:

- אם הפקודה היא **lw** (load word) - מתבצעת קריאה מהזיכרון לפי כתובת מחושבת.
- אם הפקודה היא **sw** (store word) - מתבצעת כתיבה לזיכרון של ערך מרגיסטר.
- אם מדובר בפקודה אחרת (למשל add, sub, and וכו') שלב זה פשוט מעביר את התוצאה קדימה ל־WB, ללא פעולה נוספת.

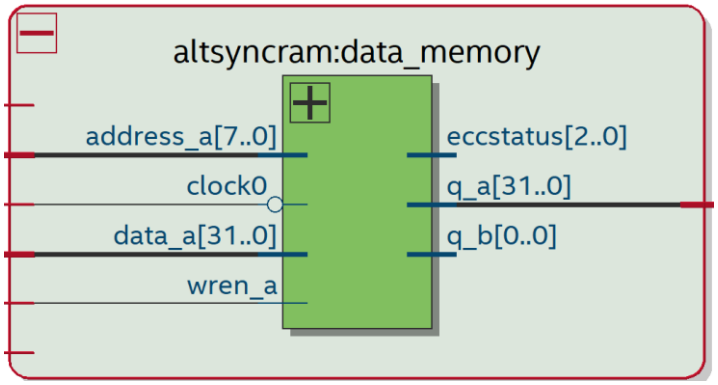
תפקידי השלב:

- ביצוע גישה לזיכרון בהתאם לכתובת שהתקבלה מה־ALU בשלב EX.
- שימוש באותות שליטה לקביעת אם לבצע קריאה (MemRead) או כתיבה (MemWrite).
- העברת תוצאה לשלב הבא – או תוצאת ALU, או הערך שהובא מהזיכרון.

שרטוט RTL של ה-entity.



שרטוט של המודל



מודל HAZARD UNIT

יחידת זיהוי הסכנות (Hazard Detection Unit) אחראית על איתור תלותיות בין פקודות הנמצאות בשלבים שונים בצינור, אשר עלולות לגרום לביצוע שגוי של פקודה. יחידה זו מונעת מצב שבו פקודה "קוראת" ערך רגיסטר לפני שהפקודה הקודמת סיימה "לכתוב" לו את התוצאה.

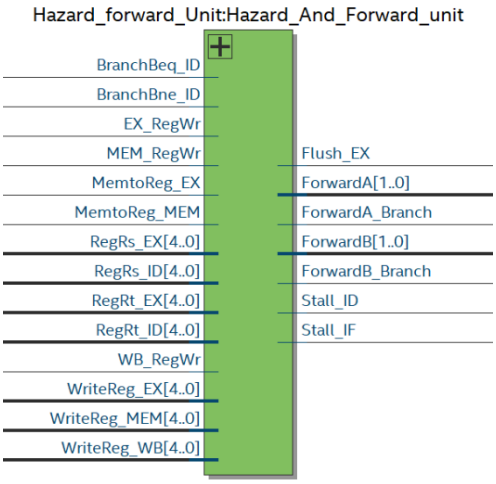
סוגי סכנות:

1. **Data Hazard**- קיים כאשר פקודה תלויה בערך שהפקודה הקודמת טרם עדכנה.
2. **Control Hazard**- נגרם מפקודות Branch או Jump שמבצעות שינוי ל־PC:

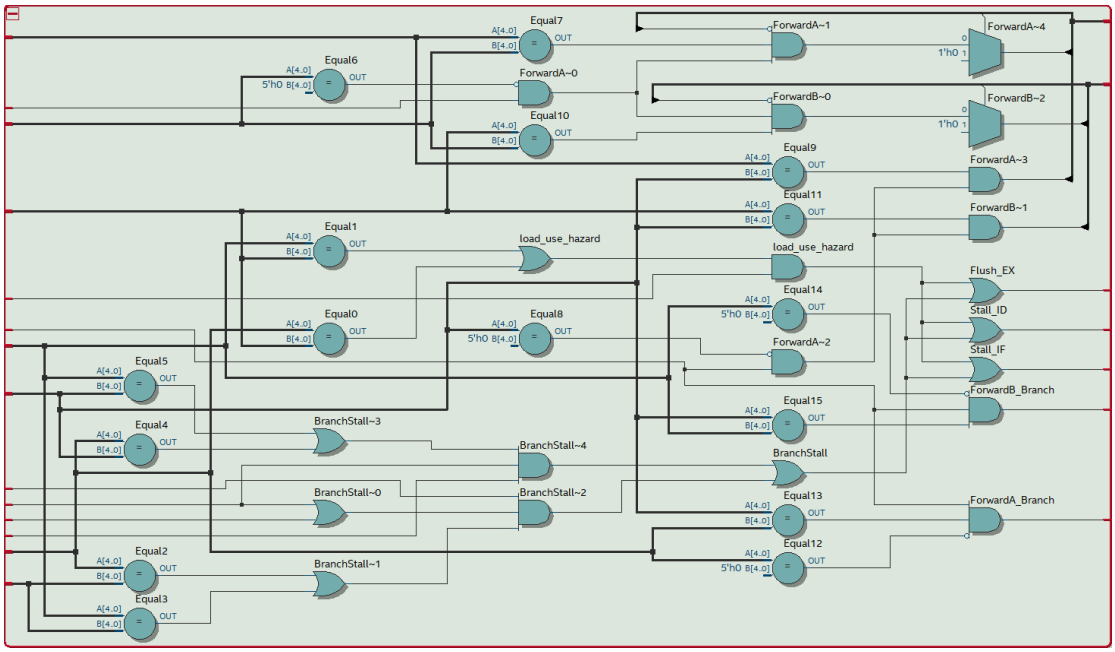
פתרונות שמיושמים:

- **Stall**- במידה ויש סכנת תלות שאין דרך לעקוף, הפקודה נעצרת (pipeline לא מתקדם במחזור זה).
- **Flush**- במידה שהתקבלה החלטה שפקודת Branch לא תתבצע, הפקודות שכבר נכנסו בטעות לצינור "נשטפות".
- **Forwarding**- פתרון אלטרנטיבי שמיושם ביחידה אחרת, אך מתואם עם Hazard Unit.

שרטוט RTL של ה-entity.



שרטוט של המודל



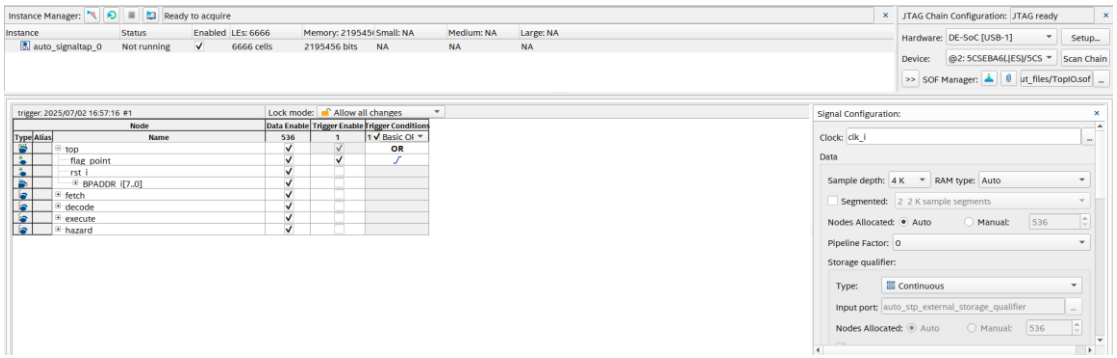
Signal Tap

במהלך העבודה בוצע אימות חומרתי של תפקוד המערכת על גבי לוח ה־FPGA באמצעות SignalTap, המובנה בסביבת הפיתוח של Quartus.הכלי מאפשר לצפות בזמן אמת באותות פנימיים של המערכת ולנתח את ההתנהגות הדינמית של רכיבי המעבד בצינור.

תהליך ההגדרה

SignalTap פועל לפי עקרון של הקלטת האותות כאשר מתקיים תנאי Trigger , כאשר ערך מסוים של ה־PC מתקבל. לצורך כך:

- הוגדר רגיסטר ייעודי בשם BPADDR_i אשר מקבל כתובת הפעלה מהמתגים על גבי הלוח (SW[7:0]).
- נעשה שימוש בהשוואה בין ערך ה־PC לבין BPADDR כדי ליצור נקודת עצירה (breakpoint).
- תנאי ההפעלה הוגדר כ־Basic OR עם Trigger יחיד, אשר מספיק כדי להתחיל תיעוד.



נציג דוגמה עבור $pc = 11h$ כאשר ב-MARS זה $29h$.

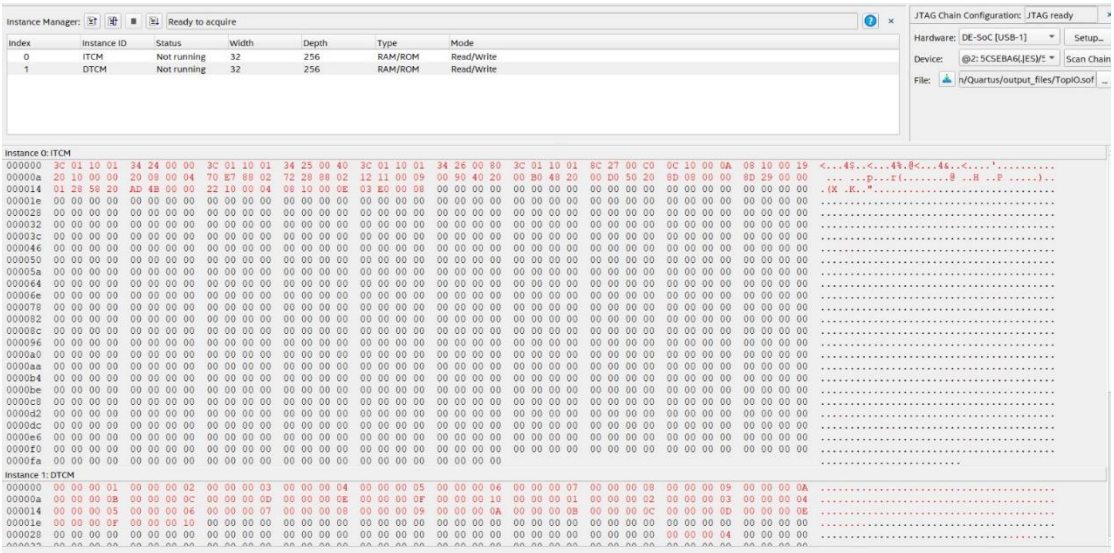
0x00400044	0x00d05020	add $10,$6,$16	29: add $t2, $a2, $s0 # find resMat pointer with offset

פקודה זו בעצם מוצאת את הכתובת של מטריצה יעד שאליה יכנסו תוצאות חיבור המטריצות. אנחנו מחברים את הערך שיש ברגיסטר 6 ($80h$) עם הערך של $offset$ (0)

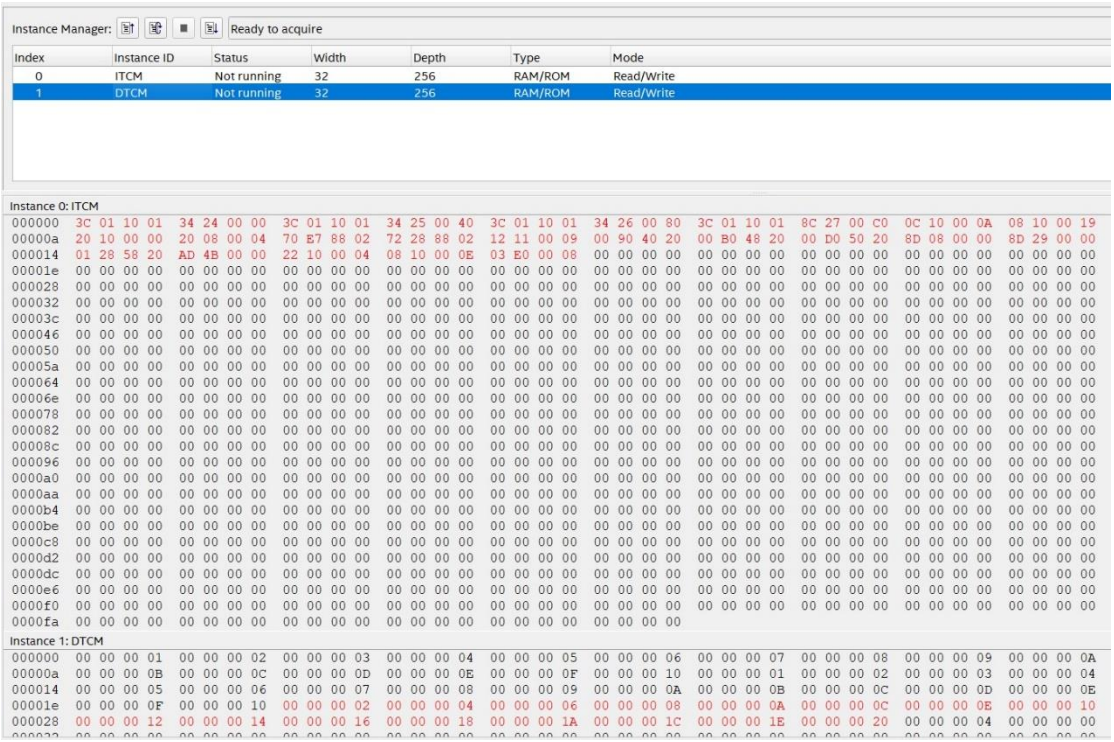
Type	Alias	Name	-2	-1	0	1	2	3	4	5	6
out		top									
*		flag_point									
*		rst_i									
bus		BPADDR_i[7..0]					11h				
C		fetch									
C		Ifetch:IFE|instruction_o[31..0]	00B04820h		00D05020h			8D080000h		8D290000h	
C		Ifetch:IFE|itcm_addr_w[7..0]	11h		12h			13h		14h	
C		Ifetch:IFE|pc_q[9..0]	040h		044h			048h		04Ch	
C		Ifetch:IFE|PCSrc_i[1..0]					0h				
C		Ifetch:IFE|PCBranch_addr_i[7..0]	30h		31h			32h		13h	
C		Ifetch:IFE|Jump_addr_i[7..0]	00h		40h			80h		00h	
C		Ifetch:IFE|next_pc_w[7..0]	11h		12h			13h		14h	
C		decode				12h					
*		Idecode:ID|ForwardA_ID									
*		Idecode:ID|ForwardB_ID									
*		Idecode:ID|MemtoReg_ctrl_i									
*		Idecode:ID|RegWrite_ctrl_i									
*		Idecode:ID|JAL_ctrl_i									
C		Idecode:ID|instruction_i[31..0]	00904020h		00B04820h			00D05020h		8D080000h	
*		Idecode:ID|Stall_ID									
C		Idecode:ID|read_data1_o[31..0]	10010000h		10010040h			10010080h		00000004h	10010000h
C		Idecode:ID|read_data2_o[31..0]			00000000h					00000004h	10010000h
C		Idecode:ID|PCBranch_addr_o[7..0]	30h		31h			32h		13h	
C		Idecode:ID|Jump_addr_o[7..0]	00h		40h			80h		00h	
C		execute									
*		Execute:EXE|ALUSrc_ctrl_i									
C		Execute:EXE|alu_ctl_w[3..0]	6h		2h						
C		Execute:EXE|alu_ctl_w1[3..0]	6h		2h						
C		Execute:EXE|b_input_w[31..0]	00000040h		00000000h						
C		Execute:EXE|read_data1_i[31..0]	00000000h		10010000h			10010040h		10010080h	
C		Execute:EXE|read_data2_i[31..0]	00000040h		00000000h						
C		Execute:EXE|instruction_i[31..0]	12110009h		00904020h			00B04820h		00D05020h	
C		Execute:EXE|funct_i[5..0]	09h		20h						
C		Execute:EXE|ForwardA_i[1..0]					0h				
C		Execute:EXE|ForwardB_i[1..0]	1h		0h						
C		Execute:EXE|Wr_data_FW_MEM_i[31..0]	00000000h		FFFFFFC0h			10010000h		10010040h	
C		Execute:EXE|Wr_data_FW_WB_i[31..0]	00000040h		00000000h			FFFFFFC0h		10010000h	
C		Execute:EXE|Wr_reg_addr_o[4..0]	11h		08h			09h		0Ah	
C		Execute:EXE|WriteData_o[31..0]	00000040h		00000000h						
C		Execute:EXE|ALUOp_ctrl_i[2..0]	1h		2h						
C		Execute:EXE|alu_res_o[31..0]	FFFFFFC0h		10010000h			10010040h		10010080h	
C		hazard									

ניתן לראות שהתפיסה מתבצעת כאשר $next\_pc = 11h$, בשלב ה-EX ניתן לראות שעבור הפקודה $0x00d05020$ תוצאת החיבור היא $80h$ וזאת הכתובת של מטריצת היעד והיא תישמר כנדרש ברגיסטר 10($0A\ h$).
ניתן לראות בתמונה את המעבר של השלבים כצינור בצורה ברורה.

נראה את תוצאת המערכת לאחר שסיימה להריץ את כל השלבים כדי להשוות את תוצאת המערכת שלנו אל מול ה-MARS ולראות שמערכת שלנו רצה כראוי גם חומרתית.



ניתן לראות באדום את האתחול של המערכת עם כל הערכים.



וכאן את תוצאת חיבור המטריצות.