**LAB 5**

**מגישים**:

* יגל בן צבי 208584615
* רון בניטה 314882317

**מחלקה :** המחלקה להנדסת חשמל

**קורס :** מעבדת ארכיטקטורה מתקדמת ומאיצי חומרה - 361.1.4693



**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:

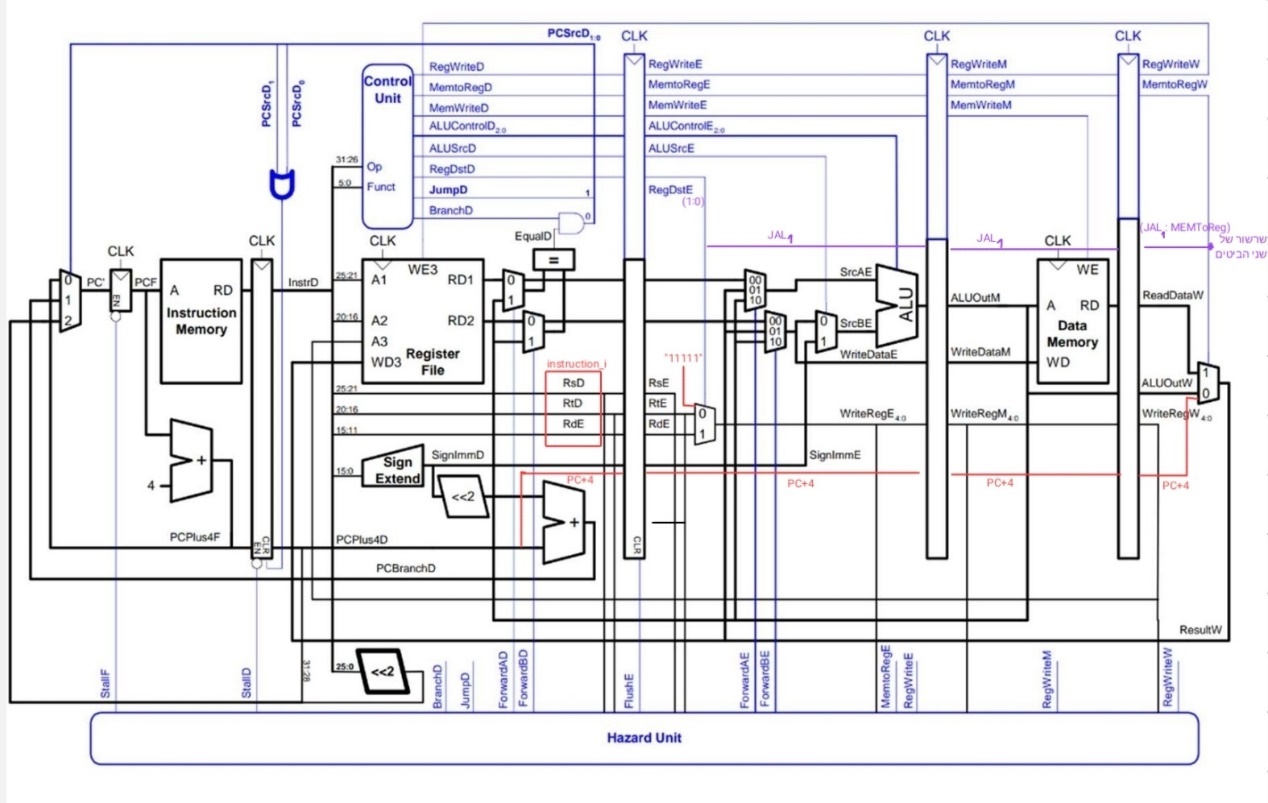
* **Register File** - כולל שלושה פורטים: שניים לקריאה ואחד לכתיבה.
* **ALU** - מבצעת חישובים אריתמטיים ולוגיים.
* **Data Memory** - זיכרון לקריאה וכתיבה.
* **Control Unit** - קובעת את אותות השליטה על פי פקודת ה־Opcode ו־Funct.
* **Hazard Detection Unit** - מונעת קונפליקטים על ידי איתותי Stall או Flush.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

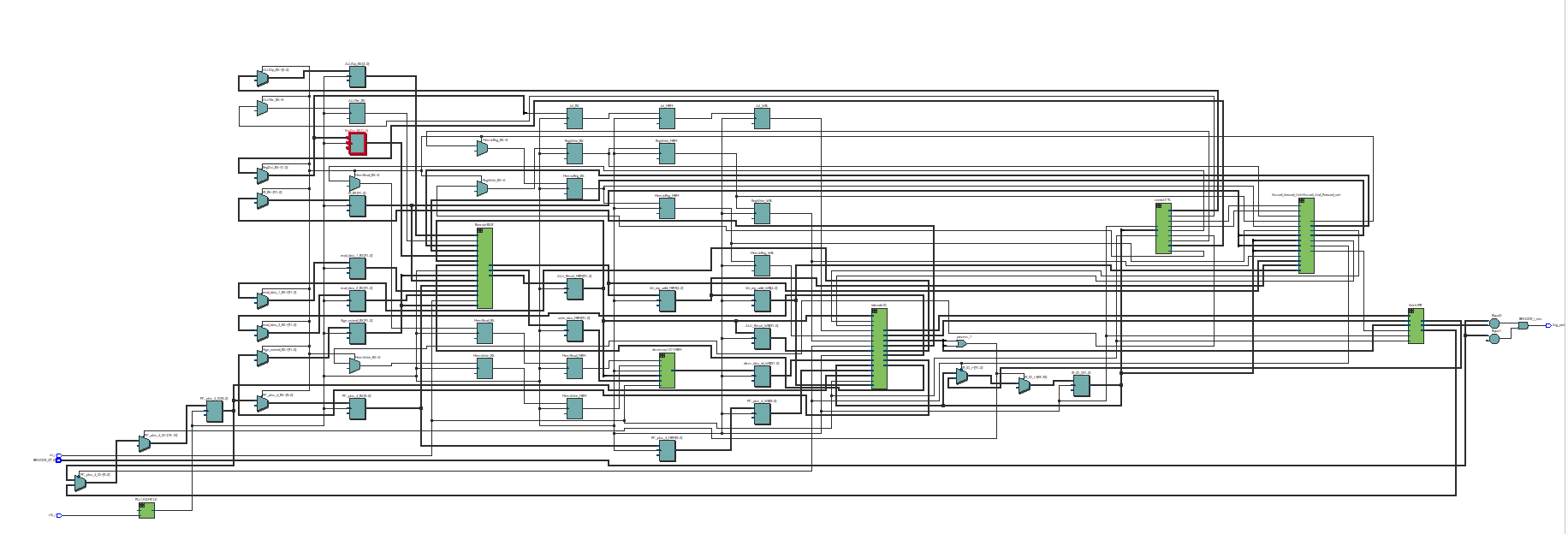
לצורכי בדיקות וביצועים, המערכת כוללת גם ממשק לשליטה מה־FPGA:

* **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.

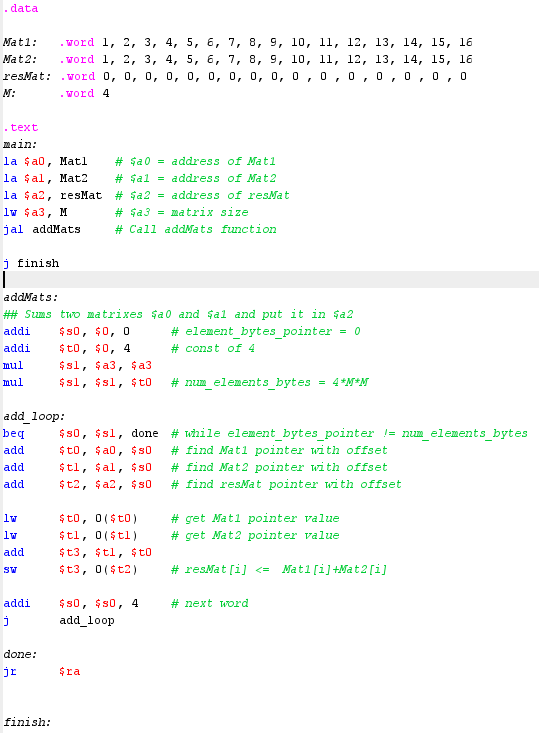
מצורף שרטוט המערכת:

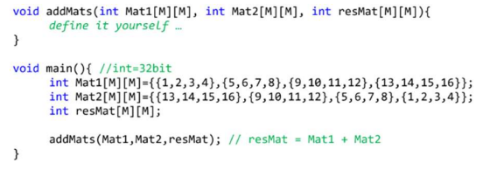


RTL:

**בדיקת המערכת**

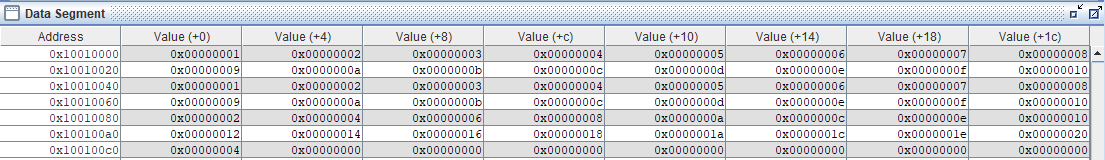
לפני שצרבנו את הקוד לבקר בדקנו שהתכנית שכתבנו באמת עובדת.  
לקחנו קוד בשפת C והפכנו אותו לקוד אסמבלי.





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.

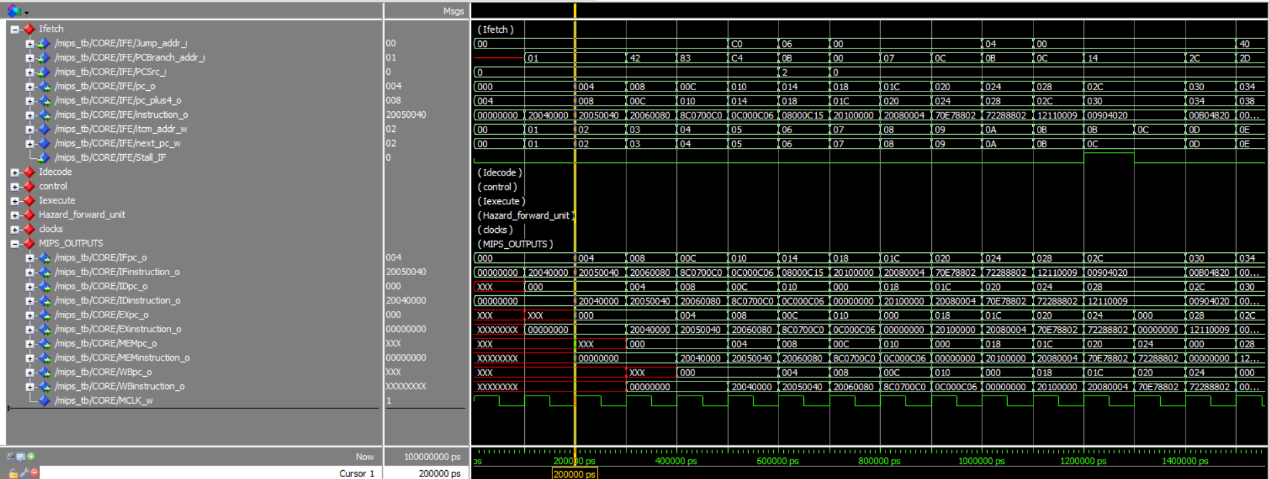
לאחר הרצה ב-MARS קיבלנו את הערכים הבאים בזיכרון.



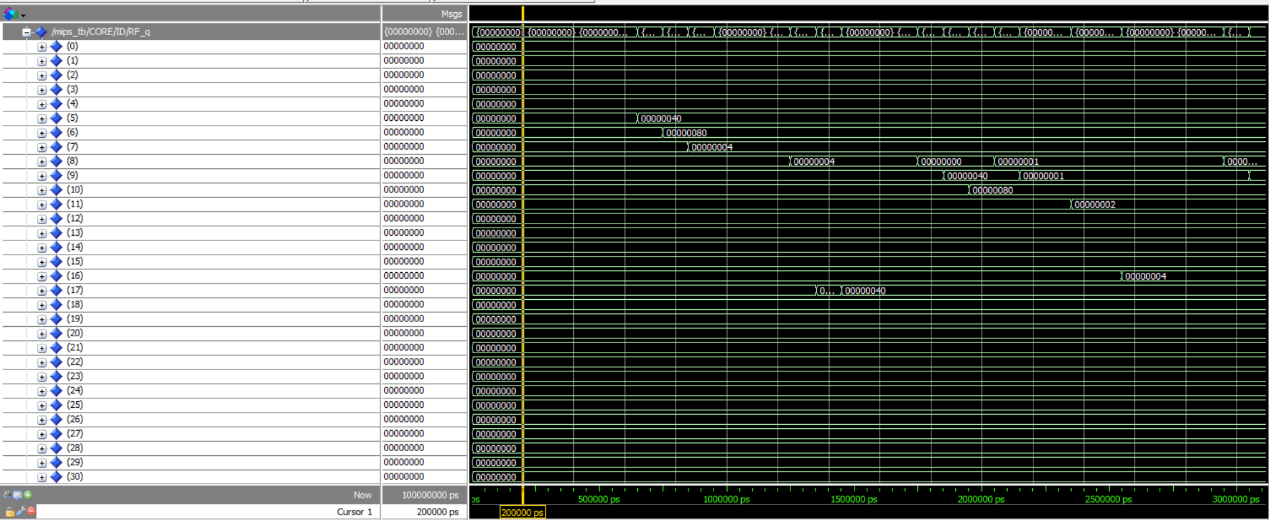
הערה – כתובת מחזיקה את גודל המטריצה הריבועית ואכן נראה שזה 4 כי המטריצות הן 4×4.

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.

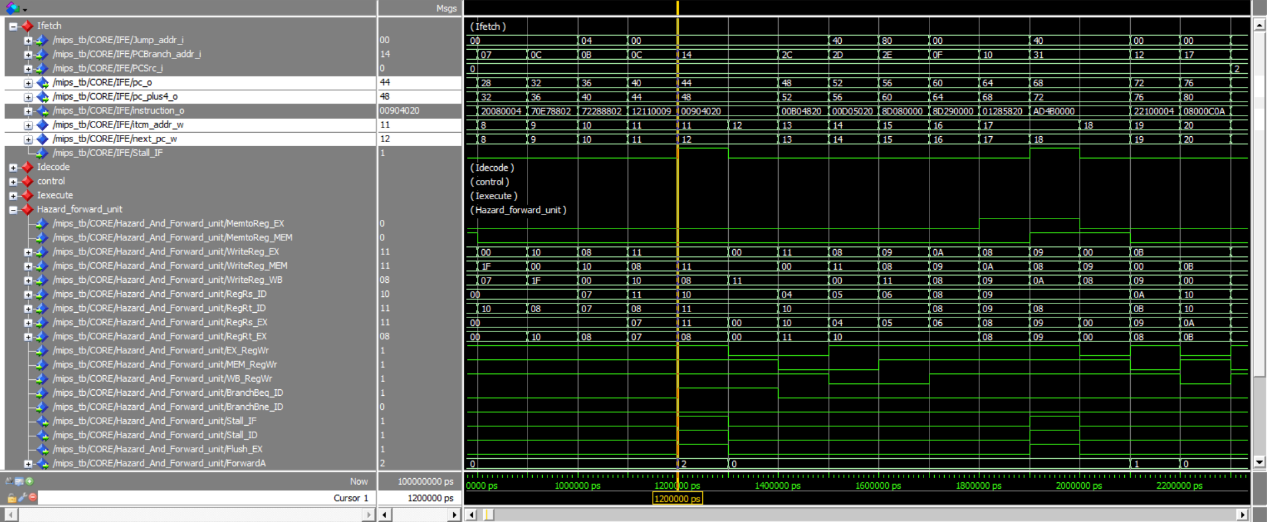


בתרשים גלים הבא מוצגים הרגיסטרים של המערכת.  
ניתן לראות שרגיסטר 4 נטען בערך 0, רגיסטר 5 בערך .  
התרדים מציג את ההתנהגות המקבילה של הקוד אסמבלי שטוען לרגיסטרים כתובות בזיכרון. כך אנחנו מתקבלים התנהגות תקינה של שמירת ערכים בזיכרון.



דוגמה לפקודה באסמבלי שמבצעת stall



 ניתן לראות את ה-STALL עולה בכתובת הנכונה.

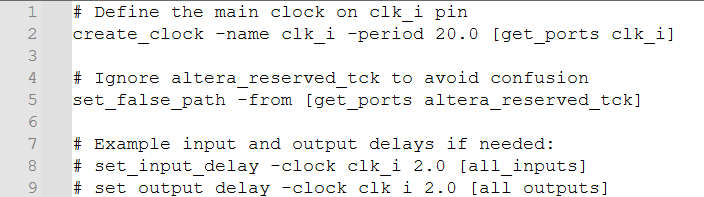
בנוסף ניתן לראות בזיכרון של הסימולציה כי אנחנו מקבלים תוצאה זהה לתוצאה שקיבלנו ב-MARS רק שויזואלית זה בסדר הפוך.



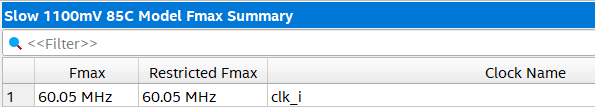
**קומפילציה ב-Quartus**

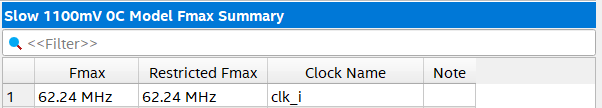
לאחר שווידאנו שהמערכת שבנינו עובדת כראוי בסימולציה נעבור לשלב הבא והיא צריבת המערכת על גבי ה-FPGA.

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

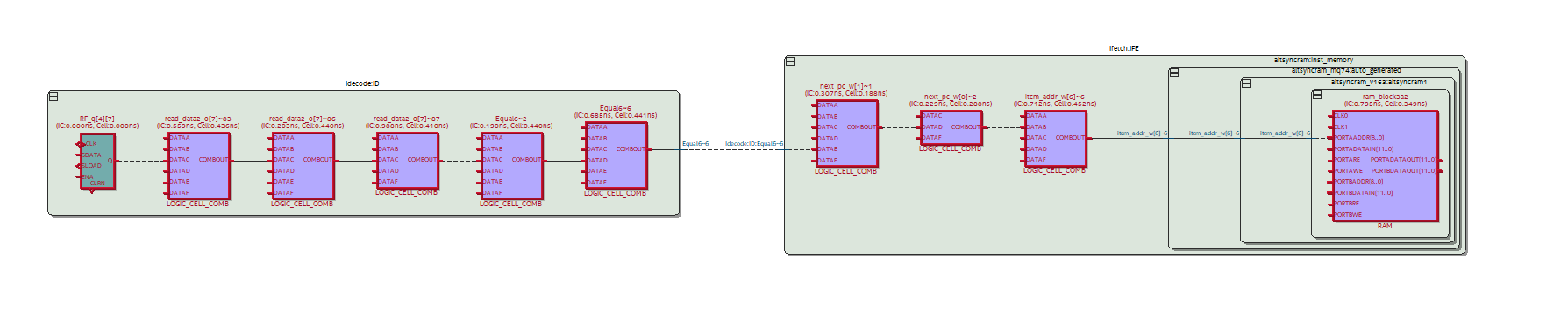


הרגיסטרים של ה-pipeline מקודמים על שעון ולכן נוכל לומר כי המערכת שלנו היא סינכרונית, דבר זה מאפשר לנו למצוא את התדר המקסימלי של המערכת.



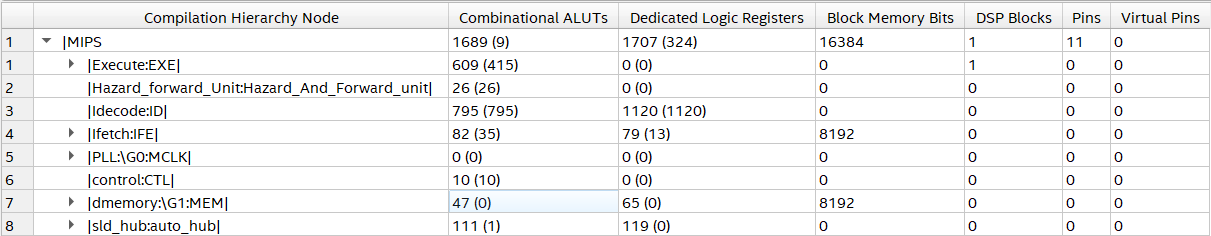


בנוסף מצורף הנתיב הקריטי של המערכת שלנו.



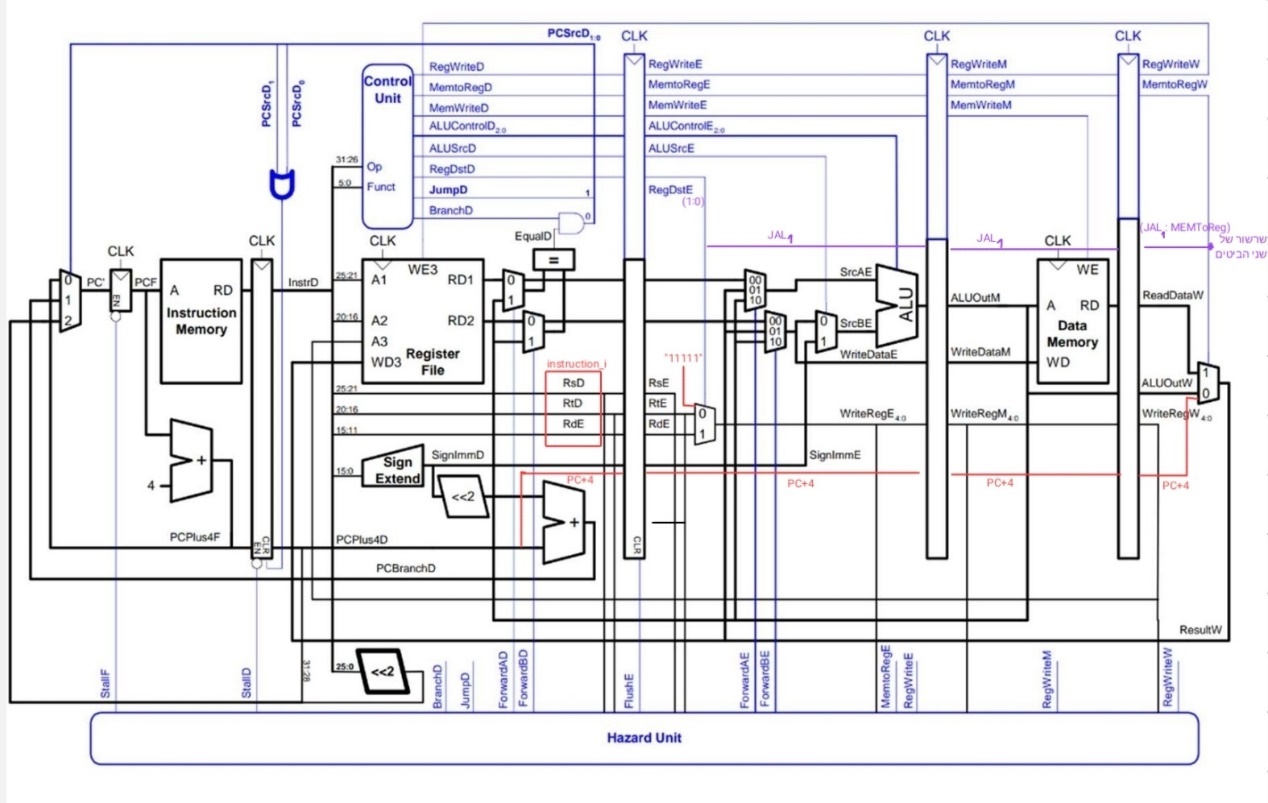
הנתיב שלנו הוא בין ה-ID לבין ה-IF. הוא מציג חישובים קומבינטורים, פענוח פקודה, חישוב ה-pc וגישה לזיכרון RAM.

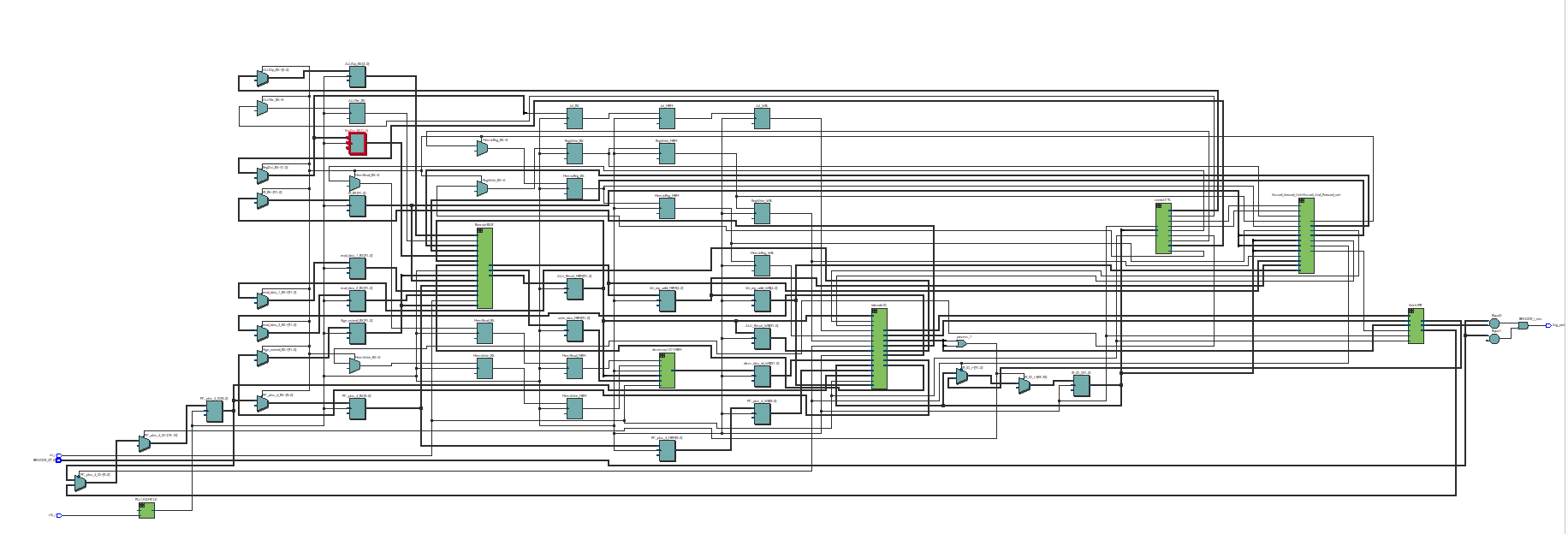
**לוגיקה עבור המודולים השונים**



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

מצורף שרטוט המערכת:



RTL לאחר סינטזה:

מודל FETCH

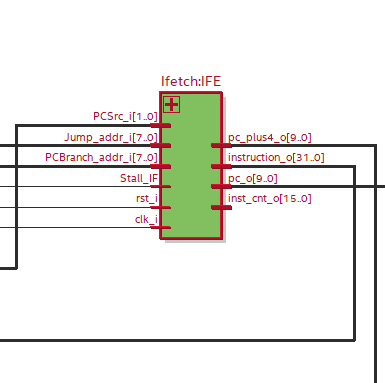
בשלב זה, המעבד מבצע את הבאת הפקודה (Instruction) מה־Instruction Memory על פי ערך מונה הפקודות (PC). ערך זה מכתיב איזו פקודה תתבצע במחזור הבא, ולכן חיוני לעקוב אחרי כל תנאי שמשפיע על עדכון ה־PC.

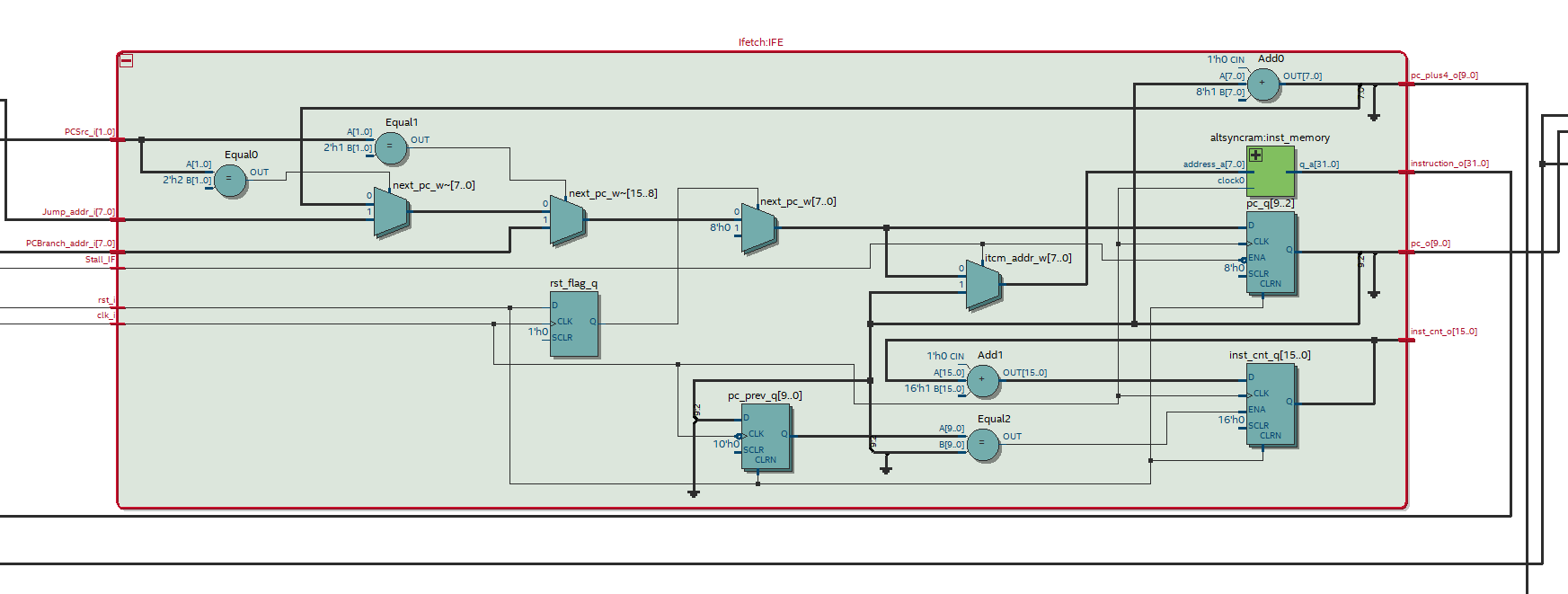
**סקירת פעולת המודול:**

מנגנון העדכון של ה־PC מורכב מהתנאים הבאים:

* **פקודת Jump**:כאשר מזוהה פקודת jump, ה־PC מקבל את הכתובת שאליה יש לקפוץ (לדוגמה: ).
* **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: .

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 .

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:

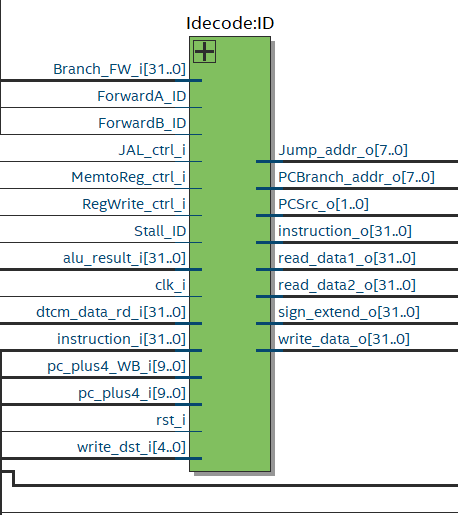
| **Type** | **שדות** | **תיאור כללי** |
| --- | --- | --- |
| R | opcode, rs, rt, rd, shamt, funct | פקודות אריתמטיות/לוגיות בין רגיסטרים |
| I | opcode, rs, rt, immediate | גישה לזיכרון, סניפים ועוד |
| J | opcode, address | פקודות קפיצה (jump) |

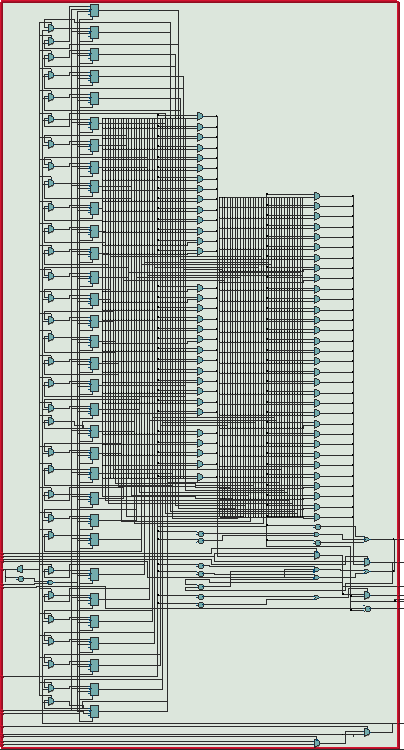
**מה מתבצע בשלב זה:**

* 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.

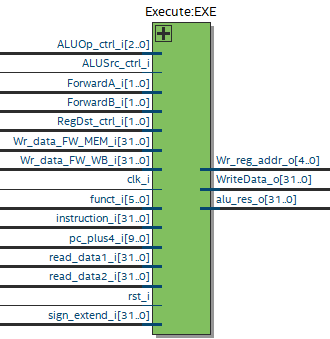
**טיפול ב־branch hazard:**

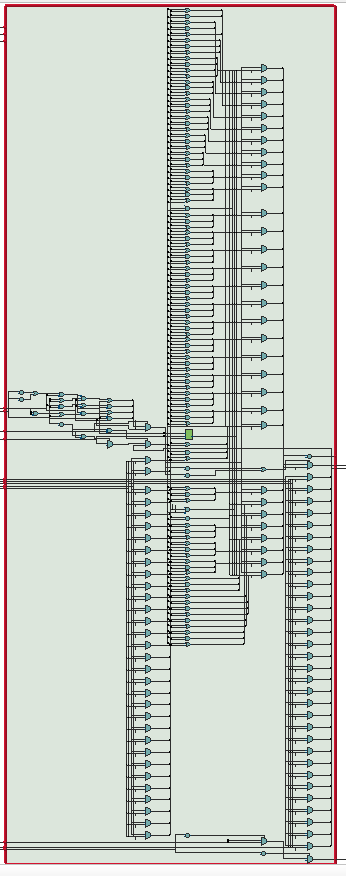
* הפלט של ALU (Zero) קובע האם התנאי מתקיים.
* במידה וכן, יחידת ה־control שולחת Flush כדי למחוק את הפקודה שנכנסה ל־IF או ID לפני שנודע שהתנאי אמיתי.
* זה גורם ל־1 delay slot.

**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

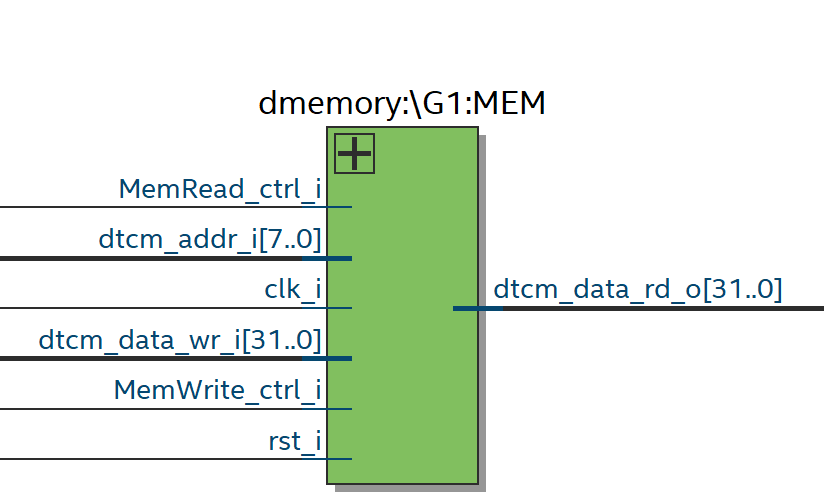
בשלב זה, המעבד מבצע גישה לזיכרון נתונים (Data Memory), בהתאם לסוג הפקודה:

* 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.

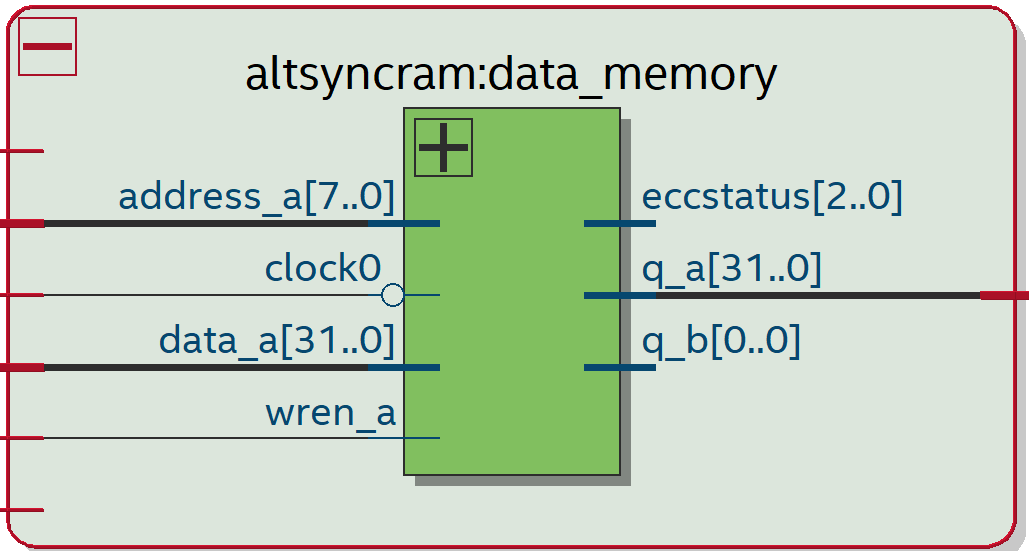
**תפקידי השלב:**

* 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.

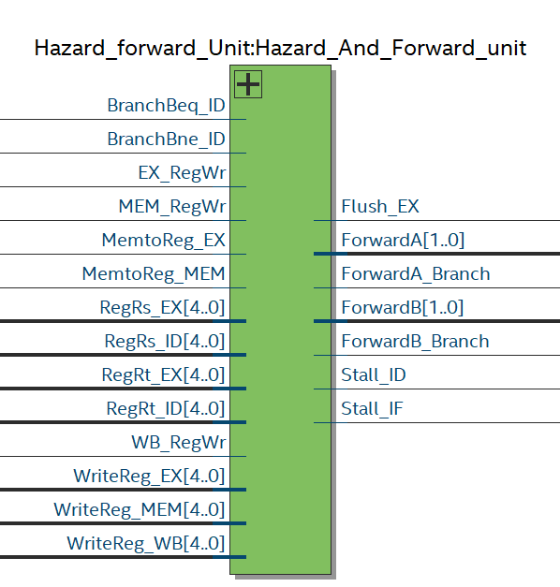
**סוגי סכנות:**

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

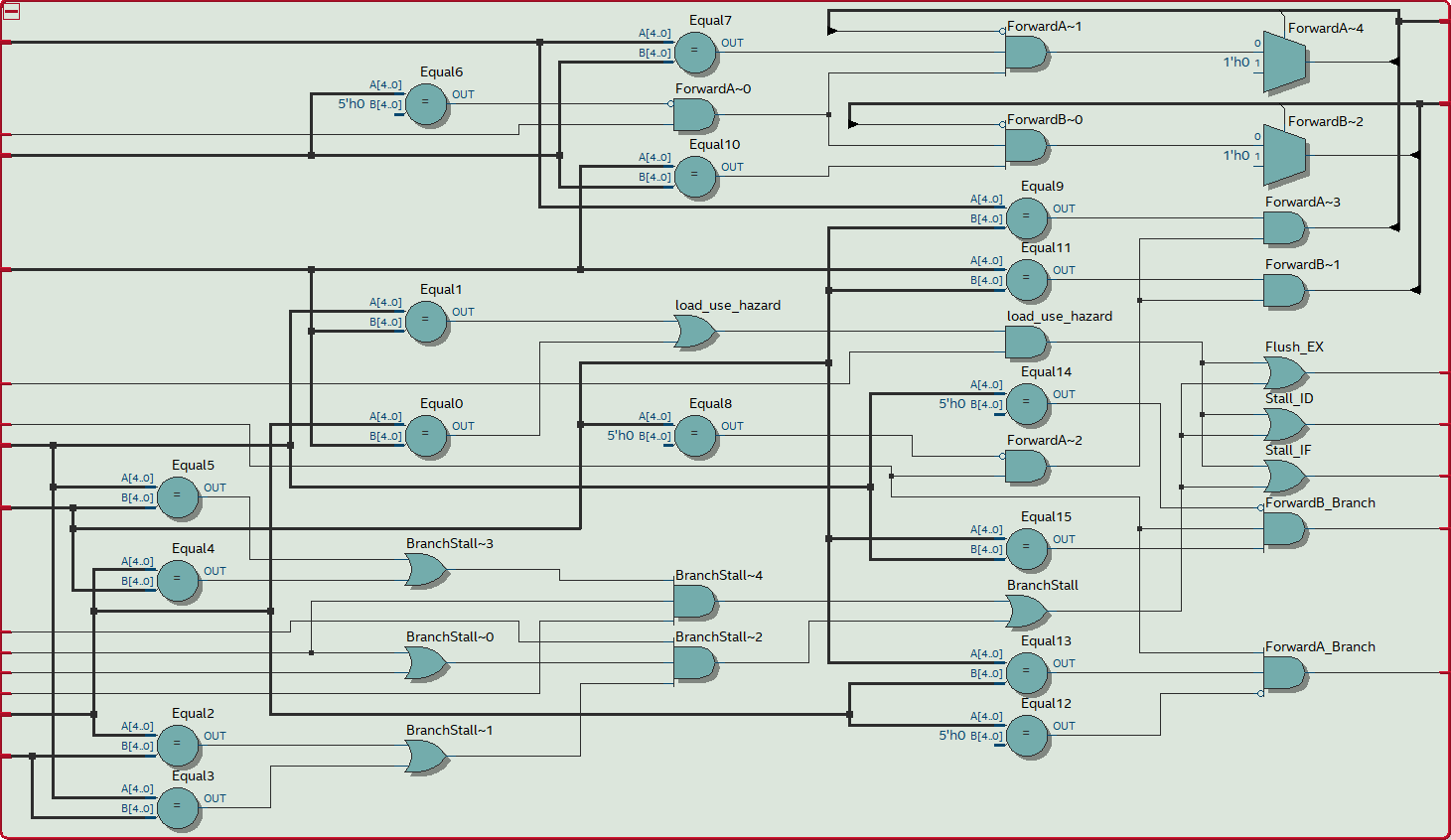
**פתרונות שמיושמים:**

* **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



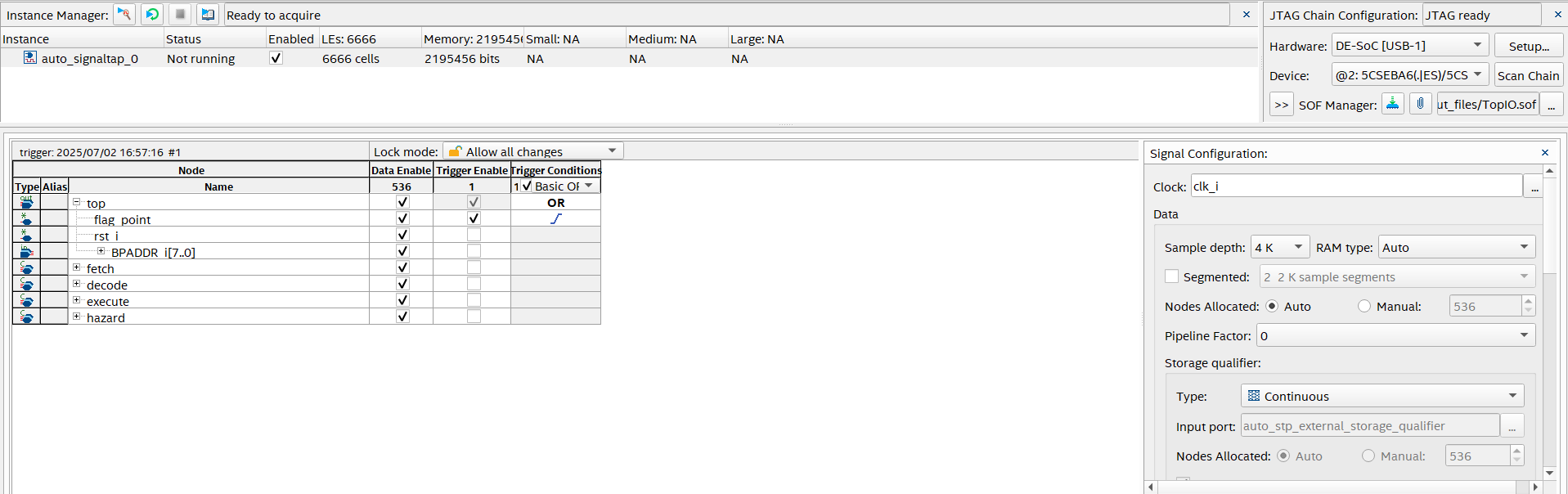
**Signal Tap**

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:

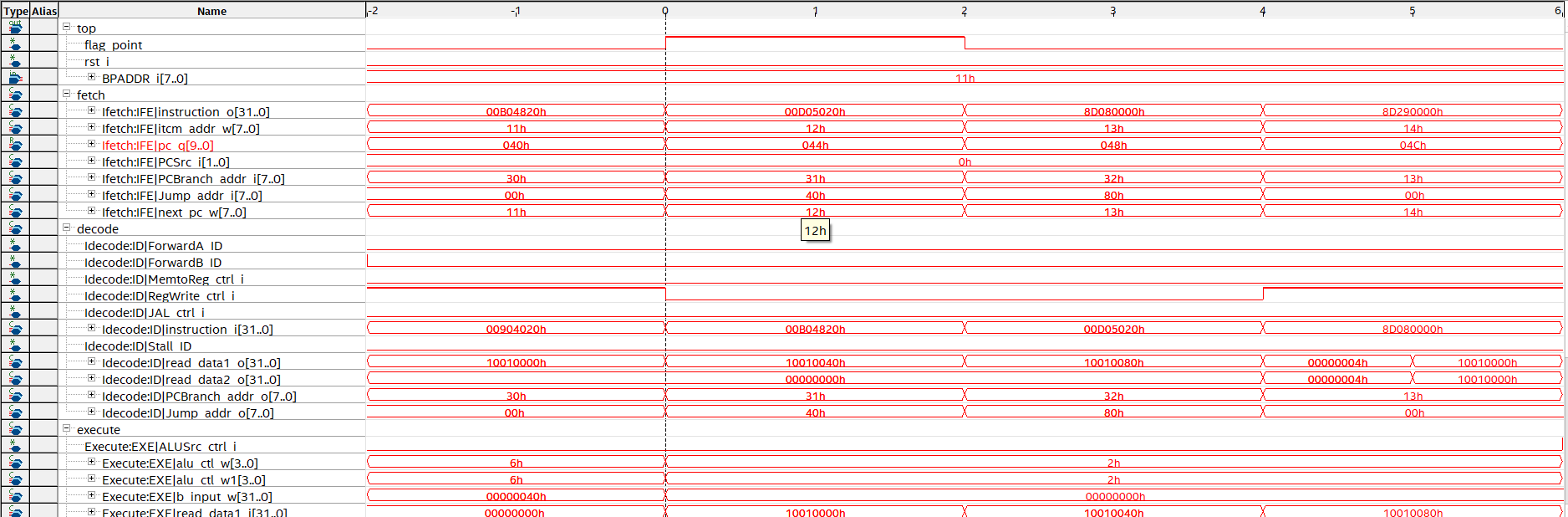
* הוגדר רגיסטר ייעודי בשם BPADDR\_i אשר מקבל כתובת הפעלה מהמתגים על גבי הלוח (SW[7:0]).
* נעשה שימוש בהשוואה בין ערך ה־PC לבין BPADDR כדי ליצור נקודת עצירה (breakpoint).
* תנאי ההפעלה הוגדר כ־Basic OR עם Trigger יחיד, אשר מספיק כדי להתחיל תיעוד.

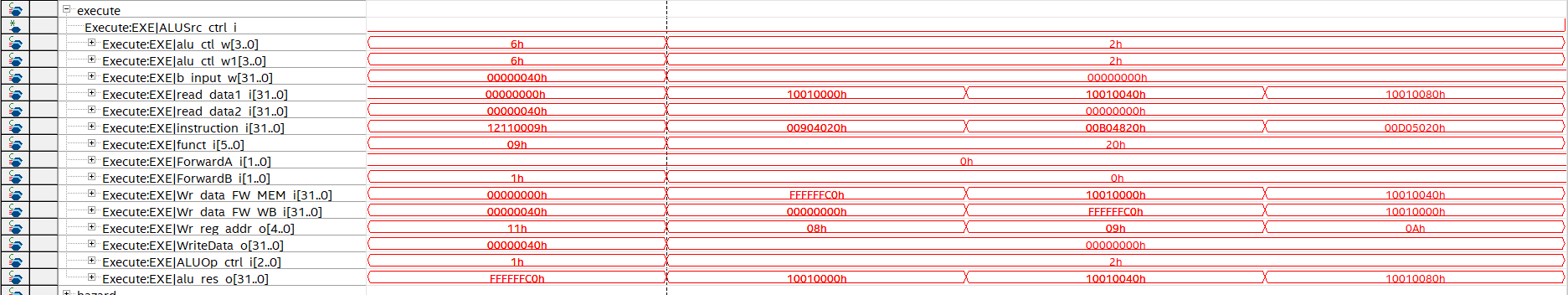


נציג דוגמה עבור כאשר ב-MARS זה .



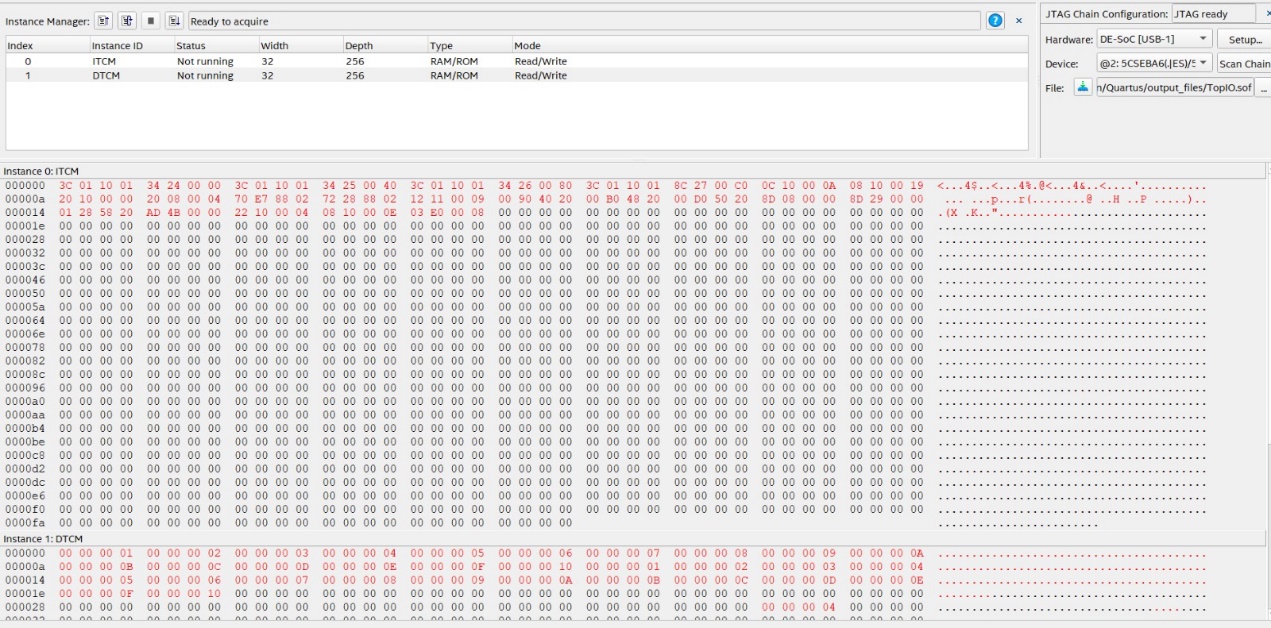
פקודה זו בעצם מוצאת את הכתובת של מטריצה יעד שאליה יכנסו תוצאות חיבור המטריצות. אנחנו מחברים את הערך שיש ברגיסטר 6 () עם הערך של (0)



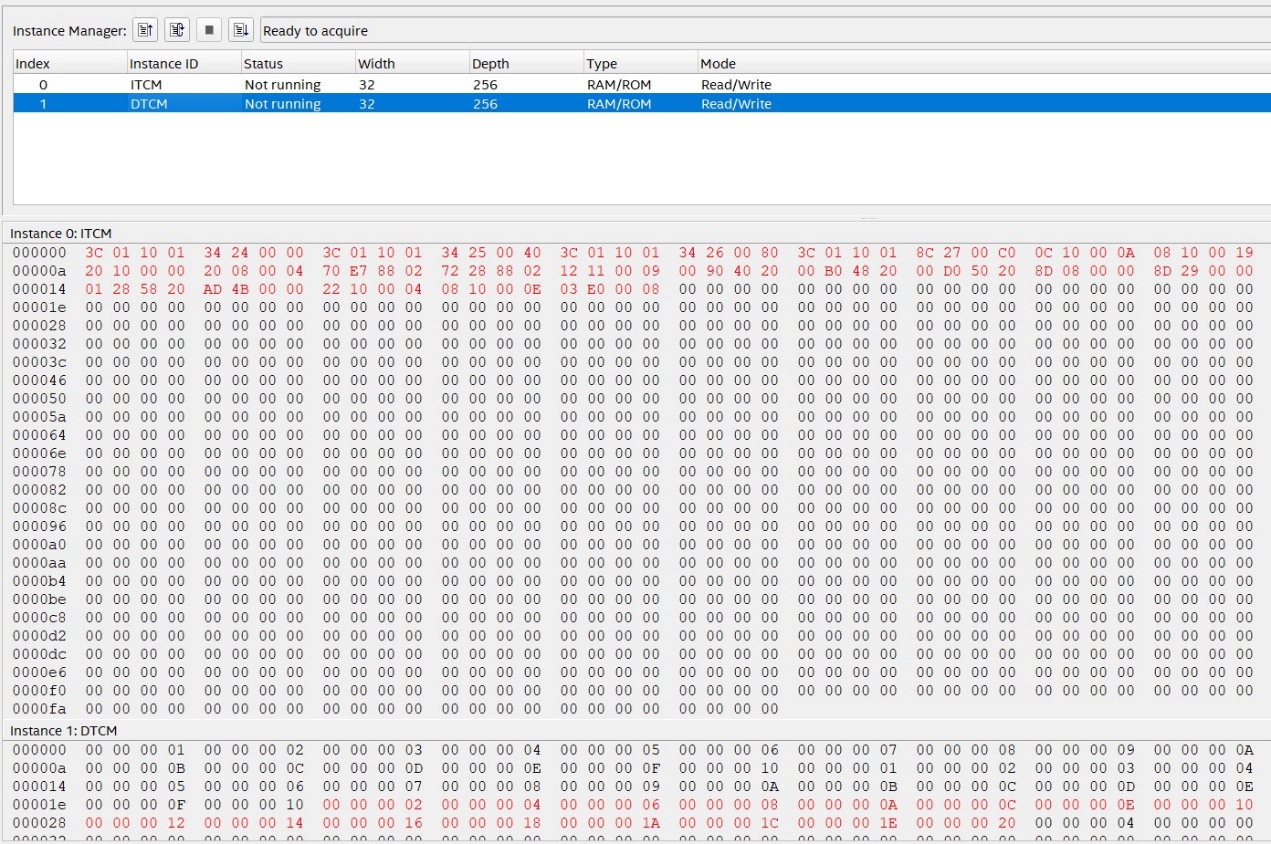


ניתן לראות שהתפיסה מתבצעת כאשר *, בשלב ה-EX ניתן לראות שעבור הפקודה תוצאת החיבור היא וזאת הכתובת של מטריצת היעד והיא תישמר כנדרש ברגיסטר 10().  
ניתן לראות בתמונה את המעבר של השלבים כצינור בצורה ברורה.*

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.