

הטכניון, מכון טכנולוגי לישראל  
הפקולטה להנדסת חשמל  
מעבדה ל-VLSI



# ניסוי בווריקציה של מעגלי VLSI מבוסס UVM

מהדורה חדשה - הערות נא לשלוח ל-goel@ee-

כל הערה תתקבל בברכה!

עדכון אחרון - 11:55 19/11/2019

<https://vlsi.eelabs.technion.ac.il/>

מסמך זה כתוב בלשון זכר ע"מ להקל על הכתיבה אך מתייחס לנשים ולגברים כאחד. עמכם הסליחה.

## תוכן עניינים

5	כללי
5	פרק 1
6	פרק 2 : סביבת העבודה
6	מבוא
6	הקדמה - Universal Verification Methodology (UVM)
7	איך להשיג תרחישים שמעולם לא חשבנו
7	מדידת הכיסוי של הבדיקות - Coverage
7	כיסוי קוד – לא מספיק
7	סיכום יתרונות הכיסוי הפונקציונלי:
7	הגדרת סביבת האימות
8	מחלקות ופאזות UVM
9	פאזות UVM
10	מבנה טיפוסית של מחלקה ב-UVM
10	פקודות מאקרו ב-UVM
11	תיאור מנגנון ה-Factory
13	פורטים מסוג TLM
16	תיאור מבנה טיפוסית של סביבת אימות
29	הכנה ניסוי מספר 1
30	הכנה ניסוי מספר 2
30	מנגנון ה-TLM-FIFO
32	אילוצים על ערכים מוגרלים
33	ביצוע ניסוי מס' 1
33	1. מקודד Hamming : סימולציות רגילות
33	2. סימולציות עם יצירה אוטומטית ורנדומלית של הכניסות
35	3. בחירת מחלקה בעזרת ה-Factory
35	4. סדר ביצוע הפאזות
36	5. הוספת scoreboard ו-reference model לסביבה
39	ביצוע ניסוי מס' 2
40	העברת מידע בעזרת Transaction Level Modeling Ports
41	הגדרת ספציפית של המעברים
43	4. יצירת כניסות ארקאיות עם אילוצים

## הנחיות בטיחות לסטודנטים במעבדות לאלקטרוניקה

### כללי:

תמצית הנחיות בטיחות מובאת לידיעת הסטודנטים כאמצעי למניעת תאונות בעת ביצוע ניסויים ופעילות במעבדות לאלקטרוניקה של הפקולטה להנדסת חשמל.  
מטרתן להפנות תשומת לב לסיכונים הכרוכים בפעילויות המעבדה, כדי למנוע סבל לאדם ונזק לציוד.  
אנא קיראו הנחיות אלו בעיון ופעלו בהתאם להן.

### מסגרת הבטיחות במעבדה:

- ♦ אין לקיים ניסויים במעבדה ללא קבלת ציון עובר בקורס הבטיחות.
- ♦ לפני התחלת הניסויים יש להתייבץ בפני מדריך הניסוי לקבלת הנחיות בטיחות ותדריך ראשוני.
- ♦ אין לקיים ניסויים במעבדה ללא השגחת מדריך.
- ♦ מדריך הניסוי אחראי להסדרים בתחום פעילותכם במעבדה; הטו אוזן קשבת להוראותיו ונהגו על פיהן.

### עשו ואל תעשו:

- ♦ יש לידע את המדריך על מצב מסוכן וליקויים במעבדה או בסביבתה הקרובה.
- ♦ לא תיעשה במזיד ובלי סיבה סבירה פעולה העלולה לסכן את הנוכחים במעבדה.
- ♦ אסור להשתמש לרעה בכל אמצעי או התקן שסופק או הותקן במעבדה.
- ♦ היאבקות, קטטה והשתטות אסורים. מעשי קונדס מעוררים לפעמים צחוק אך הם עלולים לגרום לתאונה.
- ♦ אין להשתמש בתוך המעבדה בסמים או במשקאות אלכוהוליים, או להיות תחת השפעתם.
- ♦ אין לעשן במעבדה ואין להכניס דברי מאכל או משקה.
- ♦ יש לכבות מכשירי טלפון ניידים לפני הכניסה למעבדה.
- ♦ בסיום הפעולות יש להשאיר את השולחן נקי ומסודר.

### בטיחות חשמל:

- ♦ מדריך הניסוי עבר הכשרה בבטיחות חשמל והינו בעל תעודת חשמלאי בדרגה הנדרשת. היעזרו בו ובגורמים מקצועיים אחרים במעבדה, בעת חירום.
- ♦ בשולחנות המעבדה מותקנים בתי תקע ("שקעים") אשר ציוד המעבדה מוזן מהם. אין להפעיל ציוד המוזן מבית תקע פגום.
- ♦ אין להשתמש בציוד המוזן דרך פתילים ("כבלים גמישים") אשר הבידוד שלהם פגום או אשר התקע שלהם אינו מחוזק כראוי.
- ♦ אסור לתקן או לפרק ציוד חשמלי כולל החלפת נתיכים המותקנים בתוך הציוד; יש להשאיר זאת לטפול הגורם המוסמך.
- ♦ אין לגעת בלוח החשמל המרכזי, אלא בעת חירום וזאת לצורך ניתוק המפסק הראשי.

### בטיחות אש, החייאה ועזרה ראשונה:

- ♦ מדריך הניסוי עבר הכשרה בבטיחות אש, החייאה ועזרה ראשונה. העזרו בו ובגורמים מקצועיים אחרים במעבדה, בעת חירום.
- ♦ במעבדה ממוקם מטף כיבוי אש ותיק עזרה ראשונה, זהו את מקומו.
- ♦ אין להפעיל את המטפים ואין להשתמש בציוד העזרה הראשונה, אלא בעת חירום ובמידה והמדריך וגורמים מקצועיים אחרים במעבדה אינם יכולים לפעול.

### **יציאות חירום:**

- ♦ במעבדה ישנה פתח יציאה אחת והיא משמשת כפתח היציאה גם בשעת חירום.
- ♦ בארוע חירום הדורש פינוי, כגון שריפה, יש להתפנות מיד מהמעבדה.

### **דיווח בעת אירוע חירום:**

- ♦ יש לדווח **מידית** למדריך ולצוות המעבדה.
- ♦ המדריך או איש מצוות המעבדה ידווחו מיידיית לקצין הביטחון בטלפון; 2740, 2222, נייד; -050 544575. **במידה ואין הם יכולים** לעשות כך, ידווח אחד הסטודנטים לקצין הביטחון.
- ♦ לפי הוראת קצין הביטחון, או כאשר אין יכולת לדווח לקצין הביטחון, יש **לדווח, לפי הצורך; משטרה 7-100, מגן דוד אדום 7-101, מכבי אש 7-102** וגורמי בטיחות ו/או ביטחון אחרים. בנוסף לכך יש לדווח ליחידת סגן המנמ"פ לענייני בטיחות; 3033, 2146/7.
- ♦ בהמשך, יש לדווח לאחראי משק ותחזוקה; 4776, 052-419917.
- ♦ לסיום, יש לדווח לאחראי האקדמי; 4661, לעוזר למנהל; 4678, לאחראי ההנדסי; 4668, 4671 ולאחראי האדמיניסטרטיבי; 3276.

**עודכן: יוני 2001**

## כללי

חוברת זו מהווה תדריך והכנה לניסוי בווריקציה של מעגלי VLSI במעבדה ל- VLSI. הניסוי מתבצע על גבי תחנות LINUX לפי מטודולוגיית ה-UVM לבדיקת מעגלי VLSI. מטרת הניסוי:

1. הבנת הקשים הכרוכים בבדיקה מקיפה של תכנונים גדולים.
2. הכרת מטודולוגיית ה-UVM המיועדת לווריקציה של מעגלי VLSI.
3. לימוד בסיסי של סביבת ה-UVM.
4. הכרת המרכיבים העקריים של סביבת הווריקציה.
5. תכנון, מימוש והרצה של סביבת ווריקציה עבור מספר מעגלים.

## מבנה הניסוי:

הניסוי מורכב מ-2 פגישות. כל פגישה אורכה ארבע שעות. לפני כל ניסוי יש להכין דו"ח מכין ולהגישו למנחה עם תחילת הניסוי.

## חלק א':

1. סימולציה של מקודד Hamming בשיטה הרגילה.
2. סימולציות עם יצירה אוטומטית ורנדומלית של הכניסות.
3. הכרת מנגנון ה-resource\_db או factory
4. פאזות ב-UVM וסדר ביצוע הפאזות
5. הוספת scoreboard ו-reference model לסביבה

## חלק ב':

1. העברת מידע בעזרת Transaction Level Modeling Ports.
2. כיסוי הבדיקות Coverage
3. הגדרת ספציפית את מעברים עבור כיסוי
4. יצירת כניסות עם אילוצים

## דרישות הניסוי:

- עליך לקרוא חוברת זאת בעיון רב (אפילו יותר מפעם אחת).
- הגשת דו"ח הכנה לניסוי עפ"י שאלות פרק דו"ח ההכנה.
- בוחן הכנה לניסוי.
- ביצוע הניסוי על תחנת עבודה.
- הגשת דו"ח סיכום שבועיים לאחר ביצוע חלק ב' של הניסוי.

## דרישות דו"ח סיכום:

הגשת:

- דו"ח מכין לשני חלקי הניסוי.
  - הדפסות הסימולציות והתוצאות של הווריקציה שהתקבלו במהלך הניסוי.
  - הערות בכתב יד לגבי התוצאות שהתקבלו.
  - תשובות בכתב של כל השאלות שמופיעות במהלך הניסוי.
- "הסטודנט מתבקש למלא את טופס המשוב האלקטרוני הנמצא בקישור <http://www2.ee.technion.ac.il/Labs/EELabs/>, הטופס ממלא באופן אנונימי. אנו זקוקים לתגובותיכם על מנת לתקן ולשפר כמו גם לשבח".

## מבוא

תהליך תכנון טיפוסי של מעגל VLSI מורכב מהשלבים הבאים : הגדרת המערכת ותכנון הארכיטקטורה, מימוש המערכת בשפה עלית כגון VHDL או verilog, סימולציות, סינתזה ובנית ה-Layout. בשלב הסימולציות יש לוודא שהתכנון עובד נכון מבחינה לוגית. עבור תכנונים גדולים בעלי כניסות רבות שלב הסימולציה הוא ארוך מאוד (בערך פי 3 משלב התכנון). יש צורך לבדוק את פעולת התכנון עם כל כניסה אפשרית על מנת לוודא שאין שגיאות. הכנת כל צירופי הכניסות החוקיות דרוש זמן ומאמץ רב. במקרים רבים המתכנן רוצה לדעת באיזו מידה הבדיקות שביצע בודקות את חלקי המעגל השונים. בעזרת כלים רגילים קשה לקבל אומדן למדד זה.

מטרת הניסוי היא הכרה והפעלה של סביבת הוורייפיקציה של UVM שפותחה במיוחד על מנת להתגבר על כל קשיי הסימולציה שתוארו לעיל. במהלך ניסוי יעשה שימוש במימושים למיניהם כמעגלי הבדיקה. הדגש של הניסוי הוא וורייפיקציה ואין צורך בהבנה מלאה של המימוש אלא רק של הממשק שלו לסביבה. בכל פעם שיש צורך בהבנת פרט זה או אחר של המימוש יובאו הסברים מלאים.

מדוע אנחנו צריכים אימות פונקציונלי?

המטרה העיקרית של אימות פונקציונלי היא לזהות כשלים ובאגים שניתן לזהות ולתקן לפני שתכנון נשלח ליצור. בתכנונים גדולים, קשה מאוד לעלות על כל הבאגים ולשם כך, פתחו כלי וורייפיקציה ומטודולוגיות וורייפיקציה מסודרות.

## הקדמה - Universal Verification Methodology (UVM)

מתודולוגיית האימות האוניברסלית (UVM) היא מתודולוגיה סטנדרטית לאימות מעגלים משולבים. UVM נגזר בעיקר מ-OVM (מתודולוגיית אימות פתוחה) שבמידה רבה, מבוססת מבוססת על eRM (reuse methodology). ספריות UVM מביאות אוטומציה רבה לשפת SystemVerilog כגון רצפים ותכונות אוטומציה של נתונים (אריזה, העתקה, השוואה) וכו'.

UVM הוא כלי המאפשר ביצוע אוטומטי של חלקים ניכרים של תהליך אימות של מעגל VLSI, ואנליזה של מידת הכיסוי שהתקבל ע"י הבדיקות שבוצעו. מתודולוגיה זאת מאפשרת:

1. הגדרה פשוטה ומדויקת של דרישות התכנון וסביבת הוורייפיקציה.
2. יצירה מהירה ויעילה של כל הבדיקות הדרושות.
3. יצירה מהירה ופשוטה של שגרות לבדיקה אוטומטית של תוצאות הסימולציה.
4. אנליזה יעילה של כיסוי הבדיקות.

UVM מספק למעשה שלושה כלים עיקריים:

- א. Constraint Driven Test Generation - יצירה אוטומטית של בדיקות בהתאם לאילוצי המשתמש.
- ב. Data and Temporal Checking - בדיקה אוטומטית של תוצאות הסימולציה מבחינת ערך המידע והופעתו בזמן הנכון.
- ג. Functional Coverage Analysis - אנליזה של כיסוי הבדיקות ובעזרתה מניעת ביצוע סימולציות מיותרות.

עבור testbenches פשוטים, פלט מוצג בחלון waveform או הודעות נשלחות אל מסוף לבדיקה חזותית ע"י מהנדס אימות. לעתים, ה-TestBench עצמו בודק את התוצאות הצפויות מול התוצאות בפועל המתקבלות בסימולציה. למרות שמימוש testbench בעל יכולת של בדיקה עצמית דורש מאמץ רב יותר, טכניקה זו יכולה להפחית באופן דרמטי את המאמץ הדרוש כדי לבדוק מחדש את תכנון לאחר שינויים ל-DUT.

### איך להשיג תרחישים שמעולם לא חשבנו

העירור שנוצר במקרה "בדיקות מכוונות" (Directed Verification) עשוי להיות מוגבל במידע של אקראיות. בשיטת ה-Directed Verification, קשה לחשוב על כל תרחישים אפשריים וכל באג אפשרי, ולכן יש סיכוי לא זניח שבאגים רבים לא יתגלו. בשיטה זו לעתים קרובות באגים מתגלים רק לאחר הייצור. פתרון הבעיות אלה הוא בדיקות אקראיות.

בשיטה של בדיקות אקראיות תחת אילוצים - constrained random verification, מהנדס האימות מדגיר את קבוצת האילוצים והתרחישים הנדרשים לאימות התכנון.

### מדידת הכיסוי של הבדיקות - Coverage

ישנם מספר קריטריונים לכיסוי כגון:

- כיסוי שורות קוד
- כיסוי תנאים - conditions
- כיסוי פונקציונלי
- כיסוי Fsm

### כיסוי קוד – לא מספיק

קוד מכוסה היטב אינו בהכרח ללא באגים, אם כי זה בהחלט פחות סביר שיהיו. על פי ההגדרה, כיסוי הקוד מוגבל לקוד התכנון. הוא לא יודע שום דבר על מה התכנון אמור לעשות. גם אם תכונה מסוימת לא מיוסמת נכון במימוש, כיסוי קוד יכול לדווח על כיסוי 100%.

כיסוי פונקציונלי עונה על שאלות כגון:

- האם נבדקו כל אורכי המידע בין 64 ל 1518?
- האם DUT נבדק עם crc טוב ושגוי?
- האם התוצאה מגיעה 4 מחזורים של שעות לאחר הפעולת הקריאה?
- האם ה-fifos מלאים לגמרי?

### סיכום יתרונות הכיסוי הפונקציונלי:

- כיסוי פונקציונלי עוזר לקבוע כמה מתוך המפרט היה מכוסה.
- כיסוי פונקציונלי מחזק את testbenches.
- מספק משוב על התכונות שלא נבדקו.
- מספק את המידע על בדיקות מיותרות אשר צורכים משאבים יקרים.

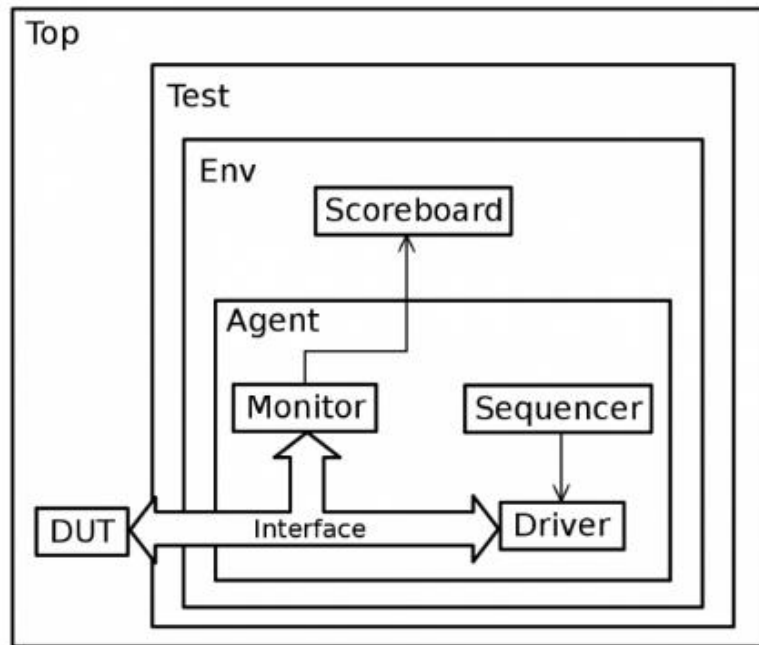
### הגדרת סביבת האימות

לפני הבנת UVM, עלינו להבין את סביבת האימות. המטרה היא לבדוק את ה-DUT (Device Under Test) ועל מנת לבצע זאת יש צורך בעירור ברכיב. עירור הרכיב נעשה בעזרת בלוק שיוצר רצפים (sequence ו-sequencer) שמועברים באמצעות בלוק אחר, ה-driver ל-DUT. דרוש עוד בלוק שמקשיב לתקשורת בין ה-driver ל-DUT ומנתח את הפלט של DUT. בלוק זה הוא ה-monitor.

מוניטורים דוגמים את הכניסות ואת היציאות של ה-DUT. הם מנסים לחזות את התוצאה הצפויה ולשלוח את החיזוי ופלט ה-DUT לבלוק נוסף, ה-scoreboard, לשם השוואה והערכה. כל הבלוקים האלה מהווים מערכת טיפוסית המשמשת לאימות. מטודולוגיה ה-UVM משתמשת בדיוק במבנה מסוג זה. ניתן לראות תיאור סכמטי של סביבה באיור מס' 1. ה-env היא מחלקה פשוטה מאוד המכילה את ה-agent ואת ה-scoreboard ומחבר ביניהם. בסוף, אנחנו יוצרים מחלקה נוספת – test ולו שתי מטרות:

- יצירת בלוק env
- חיבור בין ה-sequence ל-sequencer

הסיבה שה חיבור בין ה- sequence ל- sequencer נעשה במחלקת ה- test ולא במחלקה אחרת היא שנוכל בקלות לשנות את סוג הנתונים שמועבר DUT ללא צורך לבצע שינויים ב- agent או ברצף.



איור מס' 1 : תיאור סביבת בדיקה טיפוסית

בדרך כלל, sequencers, drivers, ו- monitors מהווים את הבלוקים של agent. ה- Agent וה- scoreboard מרכיבים את ה- environment. כל הבלוקים האלה נשלטים על ידי בלוק גדול יותר כלומר ב- test block. בלוק זה שולט על כל בלוקים ותתי בלוקים של testbench.

כדי להמחיש את היתרון של תכונה זו, נבחן מצב שבו בודקים DUT המשתמש ב- SPI לתקשורת. אם, נרצה לבדוק DUT דומה אבל עם I2C במקום SPI, כל מה שצריך לעשות הוא להוסיף driver ו- monitor עבור I2C במקום אלה של ה- SPI הקיים. יתר הבלוקים יישארו ללא שינוי.

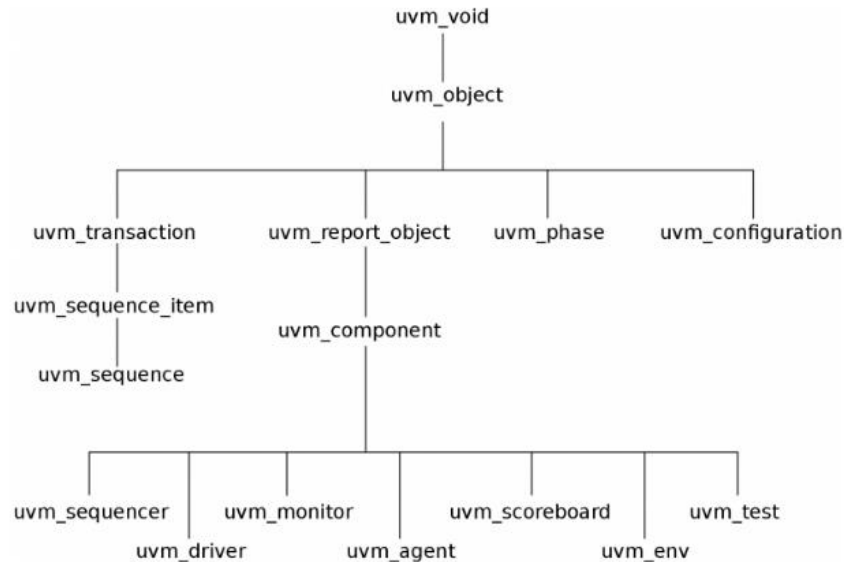
כעת נכיר מספר מושגים של עולם ה- UVM :

- מחלקות ופאזות UVM
- מבנה טיפוסית של מחלקה ב- UVM
- פקודות מאקרו ב- UVM
- תיאור מנגנון ה- Factory
- פורטים מסוג TLM
- תיאור מבנה טיפוסי של סביבת אימות

### מחלקות ופאזות UVM

ב- UVM, כל הבלוקים המוזכרים מיוצגים כאובייקטים הנגזרים ממחלקות UVM קיימות. עץ המחלקות של מחלקות UVM החשובים ביותר מופיע באיור מס' 2.



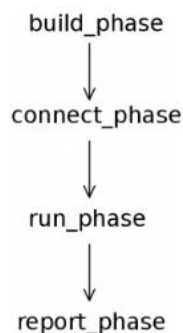


**איור מס' 2 : עץ המחלקות של מחלקות UVM**

הנתונים שמעוברים אל DUT מאוחסנים במחלקה שנגזרת מ- `uvm_sequence_item` או `uvm_sequence`. ה-sequencer ייגזר מ- `uvm_sequencer`, ה- driver מ- `uvm_driver`, וכן הלאה. לכל אחד ממחלקות האלה כבר יש כמה מתודות שימושיות מוכנות, כך שהמתכנן יכול רק להתמקד בחלק החשוב, כלומר בחלק הפונקציונלי של המחלקה שתאמת את הרכיב.

### פאזות UVM

לכל המחלקות פאזות של סימולציה. הפאזות הן שלבי ביצוע מסודרות שממומשות כמתודות. כאשר אנו יוצרים מחלקה חדשה, הסימולציה של testbench תבצע את הפאזות השונות כדי לבנות, להגדיר ולחבר את היררכיה מרכיבי ה- testbench. הפאזות החשובות ביותר מפיעות באיור מס' 3.



**איור 3 : הפאזות החשובות ביותר**

להלן הסבר קצר על כל פאזה:

- פאזה הבנייה, build phase, משמש לבניית המרכיבים בהיררכיה. לדוגמא, שלב הבנייה של המחלקת ה- agent יבנו ה- sequencer, ה- driver וה- monitor.
- פאזה ה- connect phase תשמש לחיבור תתי הבלוקים השונים של המחלקה. בדוגמא זאת, פאזה ה- connect של ה- agent יחבר את ה- driver ל- sequencer ויחבר את ה- monitor ליציאה חיצונית.
- פאזה ה- run phase הוא השלב העיקרי של הביצוע, זה המקום שבו נמצא בפועל הקוד של סימולציה יבוצע.
- בסוף, פאזה ה- report היא הפאזה שבה מצגות תוצאות הסימולציה.

ישנן פאזות רבות נוספות ואף אחת מהן אינה חובה. אם אין צורך לפאזה מסוימת, ניתן פשוט להשמיט אותה ו-UVM יתעלם ממנה.

### מבנה טיפוסית של מחלקה ב-UVM

מחלקה טיפוסית ב-UVM תראה דומה לזו המתואר בקוד מס' 1.

```
class generic_component extends uvm_component;
  `uvm_component_utils(generic_component)

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction: new

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    //Code for constructors goes here
  end_function: build_phase

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);

    //Code for connecting components goes here
  endfunction: connect_phase

  task run_phase(uvm_phase phase);
    //Code for simulation goes here
  endtask: run_phase

  function void report_phase(uvm_phase phase);
    //Code for showing simulation results goes here
  endfunction: report_phase
endclass: generic_component
```

### קוד מס' 1 : מבנה טיפוסית של מחלקה ב-UVM

הערה : ה- constructor (new) של מחלקה קורה את ה- constructor של מחלקת ההורה (`super.new(name, parent)`) על מנת שיוכר כל תוכן ההורה.

### פקודות מאקרו ב-UVM

היבט חשוב נוסף של UVM הם פקודות מאקרו. פקודות מאקרו מיישמות כמה מתודות שימושיות. הם אופציונליים, אך מאד מומלצים. הנפוצים ביותר הם:

``uvm_component_utils`

מאקרו זה רושם את סוג המחלקה החדשה ב- UVM factory (ראה עמוד הבא).

``uvm_field_int`

מאקרו זה רושם משתנה ב- UVM factory ומממש כמה פונקציות כגון `copy()`, `compare()`, `print()`.

``uvm_info`

זה מאקרו מאוד שימושי כדי להדפיס הודעות מסביבת ה-UVM.

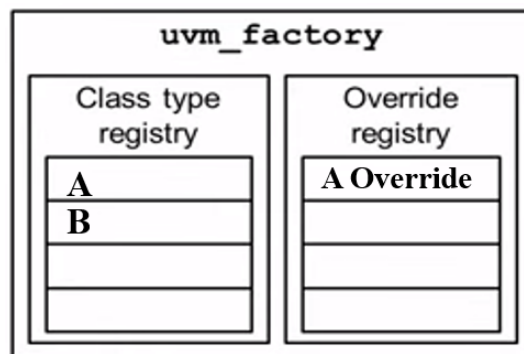
## תיאור מנגנון ה-UVM Factory

כדי ליצור מופע של אובייקט, ב-SystemVerilog ניתן להשתמש ב-`UVM.new()` מציעה גם שיטה נוספת : ה-`factory`. ה-`uvm_factory` הינו מנגנון ליצירת אובייקטים UVM ורכיבים. השימוש של `factory` ב-UVM מאפשר שליטה על התנהגות של כל מרכיבי הסביבה ב-`test` למשל בכל רצף או אובייקט ללא כל שינוי בקוד. על מנת לאפשר זאת יש לרשום את כל הרכיבים ב-`factory` באמצעות `uvm_component_utils` או `uvm_object_utils`, וכדי לאפשר החלפת הרכיב. יצירה של רכיב חדש נעשה בעזרת :

```
:: :: type_id :: create ()
```

במקום `new()` בכל פעם יוצרים אובייקט.

לפיכך, אם `testbench` ממומש כראוי באמצעות ירושה מתאימה ומנגנוני רישום המפעל נכונים, ניתן יהיה לשנות את התנהגות ה-`testbench` ללא כל שנוי בקוד. `Resource_db` הוא מסד הנתונים הבסיסי של ה-`factory.config_db`. היא שכבה על גבי `Resource_db` המספק פונקציונליות מורחבת. דוגמא :



איור מס' 4 : מבנה ה-`factory`

דוגמא זאת נכתבה פשוט על מנת להדגים את השימוש ב-`factory`. בדוגמא זאת הקוד מגדיר מחלקות מסוג `A`, `B`, `A_ovr` ו-`A_override`. לפי איור מס' 4, במקום מחלקה מסוג `A` משתמשים במחלקה מסוג `A_override`. ניתן לעשות זאת עבור הופעה בודדת של `A` או עבור הופעה שלו (פשוט ע"י שימוש בפקודה המתאימה). לעומת זאת עבור `B` אין שינוי והאובייקט יהיה מסוג `B`.

הדוגמא הבאה היא דוגמא מנוונת שבאה להדגים את התיאור הנ"ל של שימוש במנגנון ה-`factory`. בהמשך נרחיב את הדוגמאות כך שישקפו את הדרך המומלצת לממש סביבת אימות.

```
// set_inst_override_by_name
`include "uvm_macros.svh"
import uvm_pkg::*;
```

```
//-----uvm_object-----
class A extends uvm_object;
    `uvm_object_utils(A)
```

```
function new (string name="A");
    super.new(name);
    `uvm_info(get_full_name, $sformatf("A new"), UVM_LOW);
endfunction : new
```

```
virtual function hello();
    `uvm_info(get_full_name, $sformatf("HELLO from Original class 'A'"), UVM_LOW);
```

```
endfunction : hello
endclass : A
```

```
class A_ovr extends A;
  `uvm_object_utils(A_ovr)
```

```
function new (string name="A_ovr");
  super.new(name);
  `uvm_info(get_full_name, $sformatf("A_ovr new"), UVM_LOW);
endfunction : new
```

```
function hello();
  `uvm_info(get_full_name, $sformatf("HELLO from override class 'A_ovr'"), UVM_LOW);
endfunction : hello
endclass : A_ovr
```

```
class A_override extends A_ovr;
  `uvm_object_utils(A_override)
```

```
function new (string name="A_override");
  super.new(name);
  `uvm_info(get_full_name, $sformatf("A_override new"), UVM_LOW);
endfunction : new
```

```
function hello();
  `uvm_info(get_full_name, $sformatf("HELLO from override class 'A_override'"), UVM_LOW);
endfunction : hello
endclass : A_override
```

```
//-----env class-----
class environment extends uvm_env;
  `uvm_component_utils(environment)
  A a1, a2;
```

```
function new(string name="environment", uvm_component parent);
  super.new(name, parent);
endfunction : new
```

```
function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  a1 = A::type_id::create("a1", this);
  a2 = A::type_id::create("a2", this);
```

```
  a1.hello(); // This will print from overridden class A_ovr
  a2.hello(); // This will print from overridden class A_override
endfunction : build_phase
endclass : environment
```

```
//-----test class-----
class test extends uvm_test;
  // goel uvm_factory factory;
```

```

`uvm_component_utils(test)
environment env;
function new(string name = "test", uvm_component parent = null);
    super.new(name, parent);
endfunction : new

virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env = environment::type_id::create("env", this);
    `uvm_info(get_full_name, $sformatf("TEST set_inst_override_by_name"), UVM_LOW);

```

// Perform the override here! [Open the following document:](http://www.learnvmverification.com/index.php/2015/08/19/how-uvm-factory-works/)

<http://www.learnvmverification.com/index.php/2015/08/19/how-uvm-factory-works/>

Read the "Instance Overriding" section.

העזר במסמך הנ"ל על מנת לרשום משפט אשר מבצע override על A ל-A\_ovr עבור inst בשם a2.  
**שים לב שעבור המקרה שלנו, הרמת העליונה נקראת uvm test top.**

```

endfunction : build_phase
endclass : test

module top();
import uvm_pkg::run_test;
    initial begin
        run_test("test");
    end
end

```

קוד מס' 2 : דוגמא לשימוש ב-factory

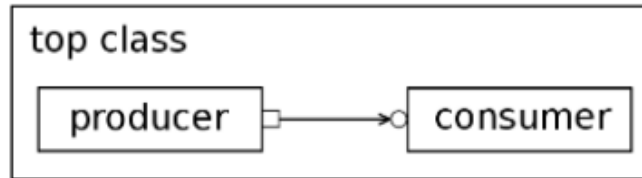
**תיאור הקוד :**

מוגדרת 3 מחלקות : A, A\_ovr ו-A\_override כל אחת עם משפט הדפסה יחודי. ב-environment ראשית מצהירים על שני אובייקטים a1 ו-a2 מסוג A. ב-build phase יוצרים את האובייקטים בעזרת מנגנון ה-factory עם create(). בסוף קיימת קריאה למתודת hello() פעם מ-a1 ופעם מ-a2.  
**שאלה :** עליך לרשום משפט אשר מבצע ל-A\_override override ל-A\_ovr עבור a1, ומשפט נוסף אשר מבצע ל-A\_override override ל-A\_override עבור a2.

**פורטים מסוג TLM**

ה-transactions הם היחידות הנתונים הבסיסית ביותר בסביבת אימות. חשוב להבין כיצד מועברים transactions בין רכיבים ב-UVM. הדרך שבה ה-driver מקבל transactions מ-sequencer, זה באמצעות TLM (Transaction Level Modeling).

TLM היא גישה למידול תקשורת בין מערכות דיגיטליות ברמת הפשטה גבוהה. גישה זו מיוצגת על ידי שתי יחידות עיקריות: port ו-export.  
 port מסוג TLM מגדיר קבוצה של מתודות ופונקציות שישמשו לחיבור מסוים. export מספק את המימוש של מתודות אלו. ה-port ו-export משתמשים באובייקטים של transactions כארגומנטים. ניתן לראות ייצוג של חיבור TLM באיור 5.



איור מס' 5 : TLM

הצרכן מממש פונקציה המקבלת transaction כארגומנט והמפיק קורא לאותה פונקציה ומעביר את ה- transaction כארגומנט. הבלוק העליון (top class) מחבר את המפיק לצרכן. קוד לדוגמה מופיע בקוד מס' 3. החיבר למעשה מתבצע בעזרת uvm\_analysis\_port ו- uvm\_analysis\_export כפי שמתואר בדוגמה הבאה:

```

class producer extends ABC;
    `uvm_component_utils(producer)

    //Step-1. Declaring analysis port
    uvm_analysis_port#(p_transaction) producer_port;

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction: new

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        //Step-2. Creating analysis port
        producer_port = new(.name("producer_port"), .parent(this));
    endfunction: build_phase

    task run_phase(uvm_phase phase);

        p_transaction p_tx;
        p_tx = p_transaction::type_id::create (.name("p_tx"), .contxt(get_full_name()));

        forever begin @(posedge some_clock)
            begin
                p_tx = value;
                //Send the transaction to the analysis port
                //Step-3. Calling write method
                producer_port.write(p_tx);
            end
        end

    endtask: run_phase
endclass: producer

```

```
class consumer extends XYZ;
  `uvm_component_utils(consumer)
  //Step-4. Declaring analysis export
  uvm_analysis_export #(p_transaction) consumer_export;
  p_transaction p_tx;
```

```
function new(string name, uvm_component parent);
  super.new(name, parent);
endfunction: new
```

```
function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  //Step-5. Creating analysis export
  consumer_export = new("consumer_export", this);
endfunction: build_phase
```

and so on ....

```
endclass: consumer
```

```
class top_env extends uvm_env;
  `uvm_component_utils(top_env)
```

```
producer producer_ref;
consumer consumer_ref;
```

```
function new(string name, uvm_component parent);
  super.new(name, parent);
endfunction: new
```

```
function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  producer_ref = producer::type_id::create(.name("producer"), .parent(this));
  consumer_ref = consumer::type_id::create(.name("consumer"), .parent(this));
endfunction: build_phase
```

```
function void connect_phase(uvm_phase phase);
  super.connect_phase(phase);
  //Step-6: Connect port to export
  producer_ref.producer_port.connect(consumer_ref.consumer_export);
endfunction: connect_phase
```

```
endclass: top_env
```

### קוד מס' 3: דוגמא לשימוש ב-TLM

ה-`uvm_analysis_port` מוגדר ב-`producer` וה-`uvm_analysis_export` מוגדר ב-`consumer`. ה-`producer` צריך לבצע את פעולת ה-`write()` כמתואר בקוד. ב-`top_env` מחברים בין ה-`uvm_analysis_port` ל-`uvm_analysis_export`. נראה בדוגמאות במהלך הניסוי שניתן לחבר `uvm_analysis_port` של יחידה מסוימת ל-`uvm_analysis_port` של ההורה ובהמשך לחבר את ה-`uvm_analysis_port` של ההורה ל-`uvm_analysis_export` של יחידה אחרת.

הערות חשובות

- להלן מספר הערות חשובות שבאות להוסיף מידע ולהביר נקודות מסויימות :
- ה- run\_phase חייב להיות task ולא פונקציה כי הוא צורך זמן
- בכל sequence (דוגמאות בהמשך), חייב להיות task בשם body שלמעשה קובע איך ה- sequence יראה.
- ברוב המקרים ה- top level testbench יבצע את הפעולות הבאות :
- הצבת ה- DUT ו- ה- interface,
- יבוא ה- package עם הגדרות ה- uvm
- הרצת ה- test באמצעות משפט initial שמכיל קריאה ל- run\_test עם שם ה- test.
- פונקציה ה- run\_test הינה פונקציה מוגדרת ב- uvm.
- run\_test יוצר אובייקט בשם umv\_test\_top ומבצע את שיגרת ה- run\_phase שלו.
- ניתן לקבוע שם שונה ל- test עם המנגנון של ה- factory בתנאי ששם ה- test נרשם ב- uvm\_component\_utils.
- פונקציה finish\_on\_completion קוראת ל- \$finish אחרי run\_test על מנת לסיים את ה- test.
- ל- driver לולאה אינסופית שמושכת את ה- transactions מה- sequence.
- בעזרת raise objection ו- drop objection ניתן לציין ש- task עדיין עסוק או שהוא סיים.
- ניתן לדווח או להדפיס בעזרת ה- macros הבאים :
- uvm\_info, uvm\_error, uvm\_warning, uvm\_fatal
- אם ה- interface נרשם ב- config/resource database כלומר במנגנון ה- factory אז מאד קל "לגשת" אליו מכל יחידה בעזרת פונקציה ה- get.
- מקובל להשתמש במנגנון ה- factory (create()) כאשר יוצרים : transaction ו- sequence ועוד.

### תיאור מבנה טיפוסי של סביבת אימות

המערכת בנויה בצורה היררכית. במקרה הזה נקרא להיררכיה הגבוהה ביותר בשם Top . ניתן לראות באיור מס' 6 שהסביבה מורכבת מהיחידות הבאות :

- הבלוק העליון
- הגדרת היעורר ל- DUT – Transactions, Sequences and Sequencers
- מחלקת ה- Driver
- מחלקת ה- Monitor
- מחלקת הסוכן Agent
- מחלקת ה- env - Environment

### מבנה הבלוק העליון :

בלוק העליון יהיה מודול SystemVerilog רגיל והוא יכול על:

- ממשק וירטואלי - Interface
- חיבור ה- DUT למחלקת ה- test, באמצעות הממשק שהוגדר.
- יצירת השעון עבור ה- DUT.
- רישום הממשק ב- UVM factory
- הפעלת הבדיקה

באיור 6 ניתן לראות סכמה של סביבת האימות. נציג בקווים כחולים עבים ושמותיהם מסומנים בטקסט אדום את כל היחידות שמימושם הושלם בקוד הנתון. בשלב זה מדובר רק על המרכיבים של Top.





```

19 end
20
21 //Variable initialization
22 initial begin
23     vif.sig_clock <= 1'b1;
24 end
25
26 //Clock generation
27 always
28     #5 vif.sig_clock = ~vif.sig_clock;
29
30 endmodule

```

קוד מס' 5: דוגמא של קוד עבור testbench

להלן הסבר קצר על הקוד:  
 משפט ה- import מייבא את ספריית ה-UVM.  
 ראשית מציבים את ה-DUT ואת הממשק לתוך הבלוק העליון.  
 שורות 9 ו-12 הן הצבת ממשק ה-DUT ואת ה-DUT וחיבור ביניהם.  
 בשורה 16 ה- uvm\_resource\_db רושמת את הממשק ב- UVM factory השם hamming\_if.  
 בשורה 18 מופעלת ה- run\_phase של hamming\_test בזמן ריצה.  
 שורה 28 מייצרת את השעון עם מחזור של 10 timeunits. ה- timeunit מוגדר גם ב- Makefile.

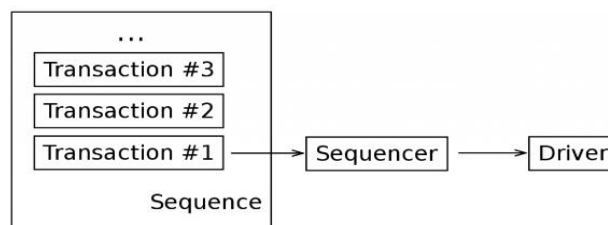
### הגדרת היעורר ל-DUT – Transactions, Sequences and Sequencers

הצעד הראשון באימות תכנון RTL הוא הגדרת איזה סוג של נתונים יש לשלוח ל-DUT. בעוד שה- driver מטפל בממשק ברמת הסיביות, זה לא נוח לשמור על רמת הפשטה זאת ככל שמתרחקים מה-DUT. לשם כך נולד הרעיון של transaction.

#### מחלקת ה- transaction

transaction הוא מחלקה, הנגזרת בדרך כלל ממחלקות uvm\_transaction או uvm\_sequence\_item, ומכיל את המידע הדרוש כדי למדל את התקשורת בין שני רכיבים או יותר. transactions הם העברות הנתונים הקטנות ביותר שניתן לבצע במודל אימות. הם יכולים לכלול משתנים, אילוצים ואפילו מתודות הפועלות על הנתונים של עצמם. בשל רמת הפשטה הגבוהה שלהם, הם אינם מודעים לפרוטוקול התקשורת בין המרכיבים, כך שניתן יהיה לעשות בהם שימוש חוזר.

כדי להזין כניסות (מידע) לתוך ה-DUT, ה- driver ממיר transactions לסיביות שמוזנות ל-DUT, בעוד ה- monitor מבצע את הפעולה ההפוכה, וממיר את סיביות היציאה ה- DUT ל- transactions. לאחר הגדרת transaction בסיסי, סביבת האימות תצטרך ליצור אוסף של transactions ולהכין אותם למשלוח אל ה- driver. זה נעשה ב- sequence. sequences הם אוסף מסודר של transactions. sequences נגזרים מ- uvm\_sequence ואת העבודה העיקרית שלהם היא יצירת transactions מרובות. לאחר יצירת transactions, מחלקת ה- sequencer שולח אותם ל- driver. תיאור פעולה זו מוצג באיור 7.



איור מס' 7: transactions, sequence, sequencer and driver

ה- sequence מכיל אוסף של transactions וה- sequencer מושך transaction מתוך ה- sequence ומעביר אותו ל- driver. עבור הדוגמא של מודול (DUT) ה- hamming יש להגדיר transaction פשוט, הנגזר מ- uvm\_sequence\_item. אשר יכול את המשתנים הבאים:

```
rand bit [1: 0] x;
bit[11:1] z;
```

מילת המפתח rand מציין שהמשתנה x יהיה משתנה אקראי שיוזן לכניסה של ה-DUT. משתנה z ידגום את יציאת ה-DUT. הקוד עבור ה-transaction מיוצג בקוד מס' 6.

```
1 class hamming_transaction extends uvm_sequence_item;
2   rand bit[7:1] x;
3   bit[11:1] z;
4
5   function new(string name = "");
6     super.new(name);
7   endfunction: new
8
9
10  `uvm_object_utils_begin(hamming_transaction)
11    `uvm_field_int(x, UVM_ALL_ON)
12    `uvm_field_int(z, UVM_ALL_ON)
13  `uvm_object_utils_end
14 endclass: hamming_transaction
```

קוד מס' 6: דוגמא של קוד עבור transaction

להלן הסבר על הקוד:

- שורות 2 ו-3 מכריזות על המשתנה של הכניסה. מילת המפתח rand מבקשת מהמהדר ליצור ולשמור ערכים אקראיים עבור משתנה זה.
- שורות 5 עד 8 כוללות את constructor הטיפוסי.
- שורות 10 עד 13 הן פקודות מאקרו של UVM שממשות סידרה של פונקציות (כגון copy) עבור המשתנה (z,x).

## רצף או Sequence

אחרי הגדרת transaction, יש ליצור sequence. להלן דוגמא של קוד עבור ל-sequence:

```
1 class hamming_sequence extends uvm_sequence #(hamming_transaction);
2   `uvm_object_utils(hamming_sequence)
3
4   function new(string name = "");
5     super.new(name);
6     `uvm_info("", "New of hamming_sequence", UVM_MEDIUM);
7   endfunction: new
8
9   task body();
10    hamming_transaction hm_tx;
11
12    repeat(15) begin
13      hm_tx = hamming_transaction::type_id::create(.name("hm_tx"), .context(get_full_name()));
14
15      start_item(hm_tx);
16      if (!hm_tx.randomize()) `uvm_error("USER_DEFINED_FLAG", "This is a randomize error");
17
18      finish_item(hm_tx);
19    end
20  endtask: body
21 endclass: hamming_sequence
```

קוד מס' 7 : קוד עבור ה-sequence

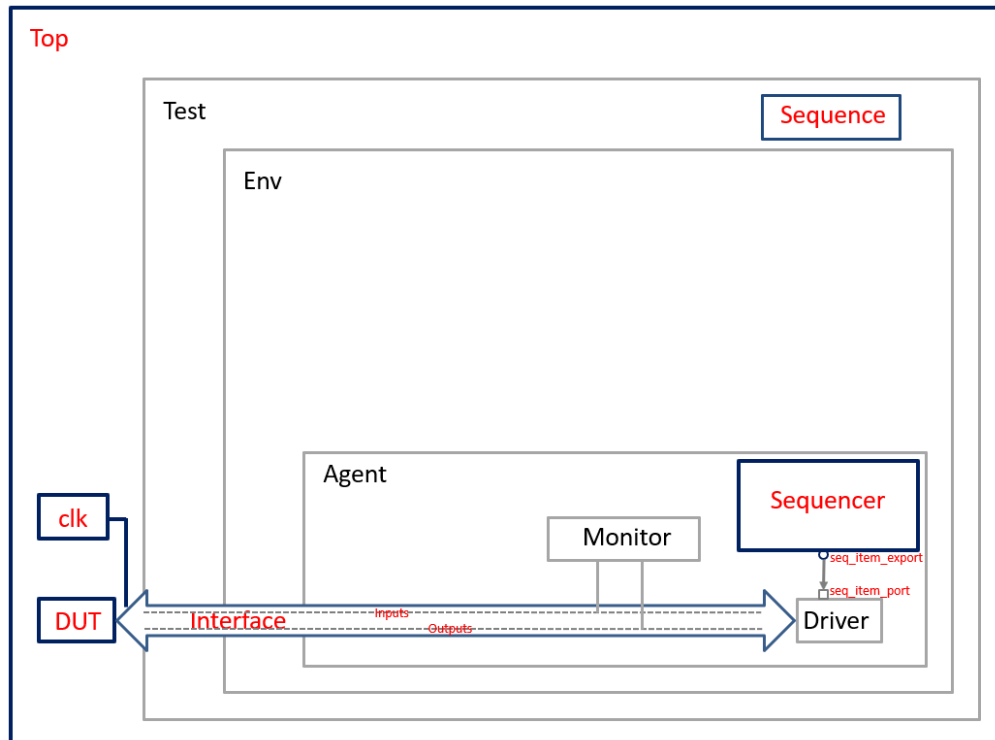
להלן הסבר על הקוד:

שורה 2 מאקרו UVM הרושם את ה- sequence ב- factory.  
שורה 9 הגדרת ה- task בשם body() שמהווה את ה- task העיקרי של ה- sequence. ה- body() מופעל כאשר ה- test קורה ל- start של ה- sequence (ראה בהסבר של test בהמשך).  
שורה 12 תחילה של לולאה כדי ליצור 15 transactions  
שורה 13 יצירת אובייקט מסוג transaction  
שורה 15 היא פקודה שמונעת התקדמות עד שה- driver מקבל את ה- transaction שנוצר  
שורה 16 מפעילה את מילת המפתח rand של ה- transaction ומרנדם את המשתנים של ה- transaction שיישלח ל- driver.  
שורה 18 היא פקודה נוספת שמונעת התקדמות עד שה- driver משלים את הפעולה עבור transaction הנוכחי.  
יחידת ה- sequencer

ה- sequencer ייגזר מן uvm\_sequencer מחלקה זו תהיה אחראית על שליחת ה- sequences ל- driver. מימוש היחידה מופיע בקוד הבא :

```
typedef uvm_sequencer#(hamming_transaction) hamming_sequencer;
```

הקוד הנ"ל יוצר טיפוס מסוג hamming\_sequencer שהוא למעשה uvm\_sequencer שנבנה על סמך הפרמטר hamming\_transaction. עד כה לסביבה המבנה הבא:



**איור מס' 8 : הסביבה לאחר הגדרת ה- sequence**

ה- sequencer מוצב ב- agent. אין צורך להגדיר את הפורטים שמופעלים באיור מס' 8 כי הם כבר המוגדרים מראש ב- sequencer וב- driver.  
ניתן להבחין בשני דברים שחסרים:  
כיצד sequence מתחבר - sequencer ? כיצד sequencer מתחבר ל- driver ? התשובות תוצגנה בהמשך.  
**מחלקת ה- driver**

ה- driver הוא בלוק שתפקידו לקיים את התקשורת עם ה- DUT. הוא מושך transactions מה- sequencer ומעביר אותם אחד אחרי השני לממשק ברמת הסיבית. אינטראקציה זו תיבדק ותוערך על ידי ה- monitor, ולכן, הפונקציונליות של ה- driver יכולה רק לשליחת הנתונים ל- DUT.

על מנת לקיים תקשורת עם ה- DUT, על ה- driver לשלוח את ה- transactions שנמשכו מה- sequencer אל כניסות ה- DUT ולהמתין עד שה- DUT מסיים להגיב לכניסה שסופקה. לכן, על ה- driver לבצע את הפעולות הבאות :

- גזירה של מחלקת ה- driver ממחלקת הבסיס uvm\_driver
  - חיבור ה- driver לממשק ה- DUT
  - משיכת הנתונים מה- sequencer ולהזנתם לממשק
  - הוספת פקודות מאקרו
- להלן קוד למימוש ה- driver:

```
1 class hamming_driver extends uvm_driver#(hamming_transaction);
2   `uvm_component_utils(hamming_driver)
3
4   virtual hamming_if vif;
5
6   function new(string name, uvm_component parent);
7     super.new(name, parent);
8   endfunction: new
9
10  function void build_phase(uvm_phase phase);
11    super.build_phase(phase);
12
13    void'(uvm_resource_db#(virtual hamming_if)::read_by_name (.scope("ifs"), .name("hamming_if"), .val(vif)));
14  endfunction: build_phase
15
16  task run_phase(uvm_phase phase);
17    drive();
18  endtask: run_phase
```

**קוד מס' 8 : קוד עבור ה- driver**

הסבר הקוד :

שורה 1 יוצרת מחלקה בשם hamming\_driver מ- uvm\_driver (hamming\_transaction) # הוא פרמטר SystemVerilog והוא מייצג את סוג הנתונים שהוא יימשכו מתוך ה- sequencer. שורה 2 מאקרו UVM הרושם את ה- driver ב- factory. שורות 6 עד 8 הן ה- constructor של המחלקה. בשורה 10 מתחילה פאזה הבנייה של המחלקה. פאזה זאת מבוצעת לפני פאזה ה- run. שורה 13 מקבלת את הממשק ממסד הנתונים של ה- factory. זהו ממשק וירטואלי והוא למעשה מצביע לאותו ממשק שהוצב קודם בבלוק העליון. בשורה 16 מתחילה פאזה ה- run, שבו יבוצע קוד ה- drive (ראה בהמשך). כעת נעבור להסבר על פאזה ה- run. בפאזה זו יש לבצע את הפעולות הבאות:

- לקבל transaction מה- sequence
- להזין את ה- transaction לכניסת ה- DUT
- להמתין מחזור שעון אחד לתגובת ה- DUT
- להודיע על סיום הפעולה ומכנות לקבל transaction חדש.

ה- driver יסיים את פעולתו ברגע שה- sequencer מפסיק לשלוח transactions. זה נעשה באופן אוטומטי על ידי ה- UVM API, ולכן אין צורך לטפל בנושא זה. כעת נשלים את ה- driver ע"י מימוש ה- task בשם drive שמעביר את כל ה- transactions ל- DUT. דוגמא של מימוש ה- drive():

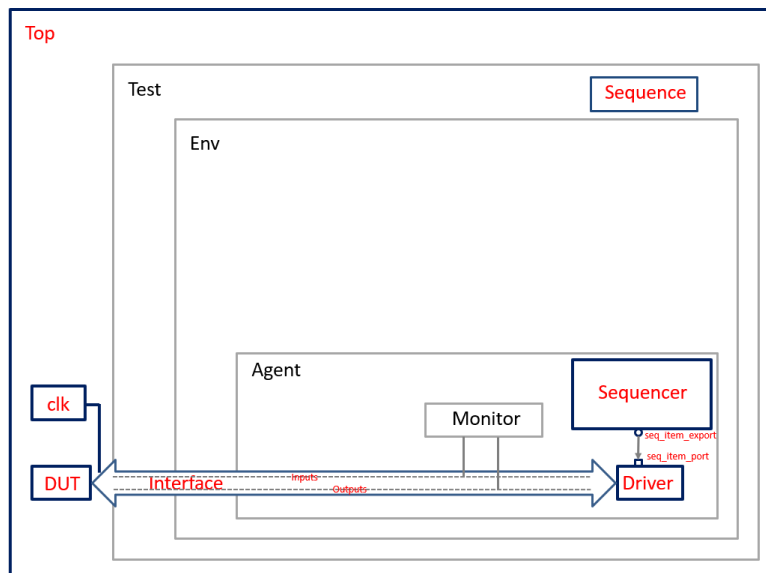
```

20 virtual task drive();
21   hamming_transaction hm_tx;
22   vif.sig_x = 7'b0000000;
23
24   forever begin
25     begin
26       seq_item_port.get_next_item(hm_tx);
27       //^ uvm_info("hm_sequence", hm_tx.sprint(), UVM_LOW);
28       vif.sig_x = hm_tx.x;
29     end
30
31     @(posedge vif.sig_clock)
32     begin
33       seq_item_port.item_done();
34     end
35   end
36 endtask: drive
37 endclass: hamming_driver

```

#### קוד מס' 8 : קוד עבור מתודת ה- drive

יש לשים לב שהגישה לכניסות והיציאות של ה- DUT היא באמצעות הממשק הוירטואלי : בשורות 26 ו- 33 ניתן לראות שימוש במשתנה מיוחד מ- UVM כלומר seq\_item\_port כדי לאפשר תקשורת עם ה- sequencer. ה- driver קורא את מתודה get\_next\_item() על מנת לקבל transaction חדש ועם סיום הפעולה עם ה- transaction הנוכחי, ה- drive קורא ל- item\_done(). אם ה- driver קורה ל- get\_next\_item() אבל ל- sequencer אין יותר transactions התהליך נעצר. seq\_item\_port הוא למעשה UVM port והוא מתחבר ל- export של ה- sequencer הנרקה seq\_item\_export. החיבור נעשה על ידי המחלקה הגבוהה יותר כלומר ה- agent. הסברים נוספים על ports בהמשך. עד כאן ההסבר על ה- driver.



#### איור מס' 9 : סביבת האימות לאחר הוספת ה- driver

## מחלקת ה-Monitor

ה-Monitor היא יחידה עצמאית שגם מקיימת תקשורת עם ה-DUT. ה-Monitor הינו רכיב פסיבי, ואינו מעביר אותות כלשהם לתוך ה-DUT. מטרתו היא לדגום את נתוני האות ולתרגם אותו למידע בעל משמעות. סביבת האימות אינה מוגבלת רק ל-Monitor אחד. על ה-Monitor לכלול:

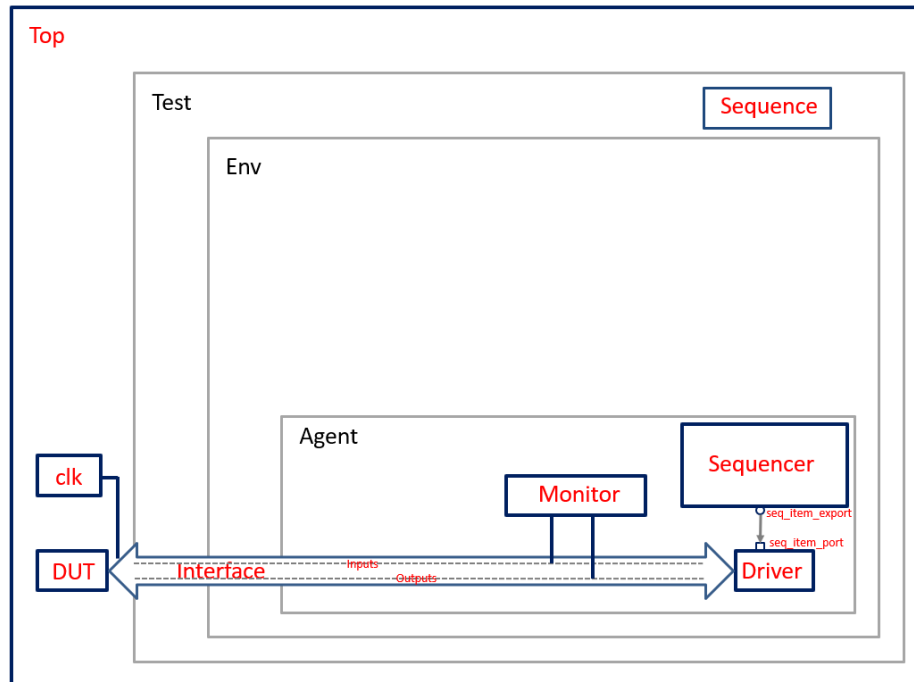
- יציאות ה-DUT לשם בדיקה
- כניסות ה-DUT לניתוח כיסוי פונקציונלי (יוסבר בהמשך)

לרוב התהליך המקובל בתוכנית אימות היא: דגימת הכניסות, חיזוי של התוצאה הצפויה והשוואת התוצאות עם התוצאות של ה-DUT. (ראה בהמשך). בשלב זה נממש monitor שרק דוגם את הכניסות ואת היציאות. בהמשך גם נדפיס ערכים אלה. להלן המימוש היחידה:

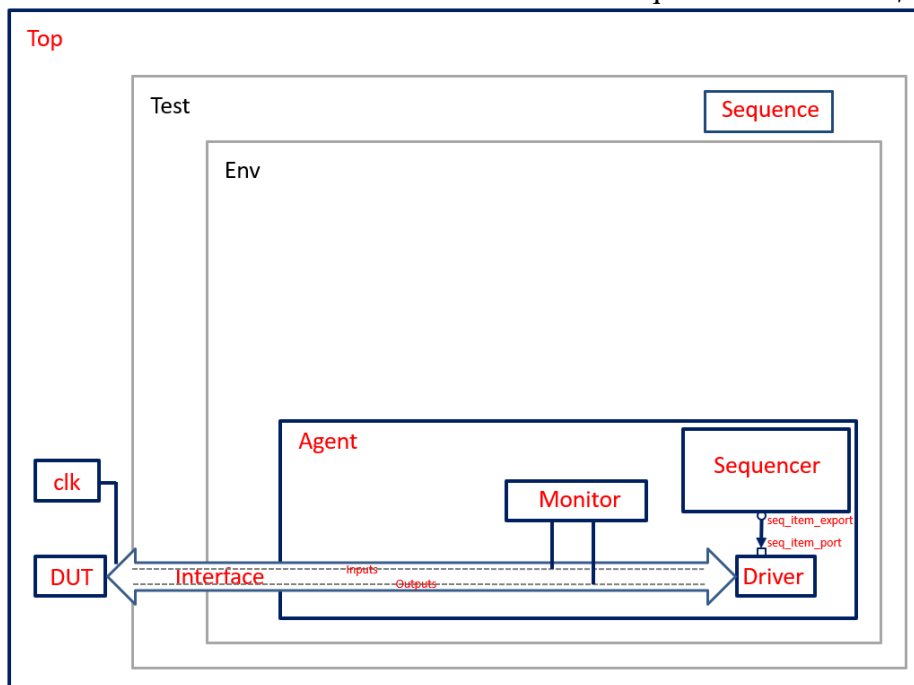
```
1 class hamming_monitor_dut extends uvm_monitor;
2   `uvm_component_utils(hamming_monitor_dut)
3
4   virtual hamming_if vif;
5
6   function new(string name, uvm_component parent);
7     super.new(name, parent);
8   endfunction: new
9
10  function void build_phase(uvm_phase phase);
11    super.build_phase(phase);
12
13    void'(uvm_resource_db#(virtual hamming_if)::read_by_name (.scope("ifs"), .name("hamming_if"), .val(vif)));
14
15  endfunction: build_phase
16
17  task run_phase(uvm_phase phase);
18
19    hamming_transaction hm_tx;
20    hm_tx = hamming_transaction::type_id::create (.name("hm_tx"), .contxt(get_full_name()));
21
22    forever begin
23      @(posedge vif.sig_clock)
24        begin
25          hm_tx.z = vif.sig_z;
26          hm_tx.x = vif.sig_x;
27          //Send the transaction to the analysis port
28        end
29    end
30  endtask: run_phase
31 endclass: hamming_monitor_dut
```

קוד מס' 9 : קוד מימוש ה-Monitor

ה-Monitor ממומש במבנה הקלסי של פאזות כפי שכבר הוסבר.



**מחלקת הסוכן Agent**  
השלב הבא הוא לחבר את כל היחידות שהגדרנו עד כה. זוהי העבודה של ה-agent. ל-agent אין צורך בפאזה ה-run, אין קוד סימולציה שיש לבצע בבlook זה. לעומת זאת כן יש צורך בפאזה של connect, מלבד פאזה של build. בקוד שמופיע בהמשך ניתן לראות שבפאזה של build יוצרים את ה-driver, sequencer, ו-monitor. בפאזה של connect מתבצע החיבור בין ה-driver ל-sequence.





בדור"כ ה-monitor יכיל גם ports על מנת להעביר את המידע הדגום להמשך ניתוח. בהמשך יובאו כמה דוגמאות.  
להלן קוד למימוש ה-agent :

```
class hamming_agent extends uvm_agent;
  `uvm_component_utils(hamming_agent)
  hamming_sequencer hm_seqr;
  hamming_driver hm_drvr;
  hamming_monitor_dut hm_mon_dut;
```

```
function new(string name, uvm_component parent);
  super.new(name, parent);
endfunction: new
```

```
function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  hm_seqr = hamming_sequencer::type_id::create(.name("hm_seqr"), .parent(this));
  hm_drvr = hamming_driver::type_id::create(.name("hm_drvr"), .parent(this));
  hm_mon_dut = hamming_monitor_dut::type_id::create(.name("hm_mon_dut"), .parent(this));
endfunction: build_phase
```

```
function void connect_phase(uvm_phase phase);
  super.connect_phase(phase);
  hm_drvr.seq_item_port.connect(hm_seqr.seq_item_export);
endfunction: connect_phase
endclass: hamming_agent
```

קוד מס' 10 : קוד מימוש ה-agent

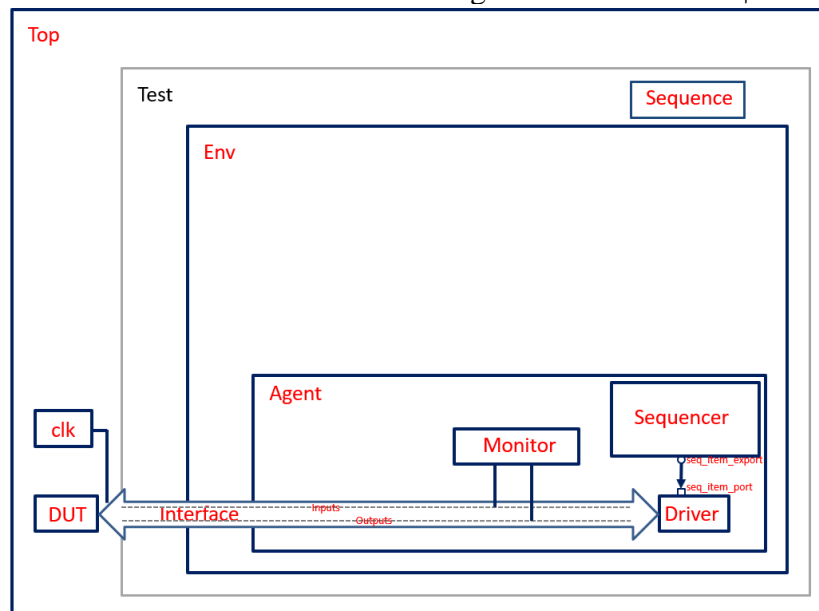
הערה : השורה –

```
hm_drvr.seq_item_port.connect(hm_seqr.seq_item_export);
```

מאפשרת העברת transactions מה-sequencer ל-driver. הסבר בהמשך.

**מחלקת ה-env**

יחידה זאת פשוטה מאד ובמקרה זה מכילה את ה-agent.



איור מס' 12 : סביבת האימות לאחר הוספת ה-agent

## קוד ה- env

להלן מימוש ה- env :

```
class hamming_env extends uvm_env;
  `uvm_component_utils(hamming_env)
  hamming_agent hm_agent;

function new(string name, uvm_component parent);
  super.new(name, parent);
  // uvm_info("", "New of hamming_env", UVM_MEDIUM);
endfunction: new

function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  hm_agent = hamming_agent::type_id::create(.name("hm_agent"), .parent(this));
endfunction: build_phase

function void connect_phase(uvm_phase phase);
  super.connect_phase(phase);
endfunction: connect_phase
endclass: hamming_env
```

קוד מס' 11 : קוד מימוש ה- env

## מחלקת ה- test

הבלוק האחרון שדרוש הוא ה- test. בלוק זה ייגזר מ- uvm\_test ולו שתי מטרות:

- הצבה של בלוק env
- חיבור ה- sequence ל- sequencer

הסיבה שמחברים את ה- sequence ל- sequencer כאן ולא ביחידה אחרת היא שזה מאפשר לשנות בקלות את סוג הנתונים שמועבר ל- DUT מבלי לשנות את הקוד של הסוכן או sequencer.

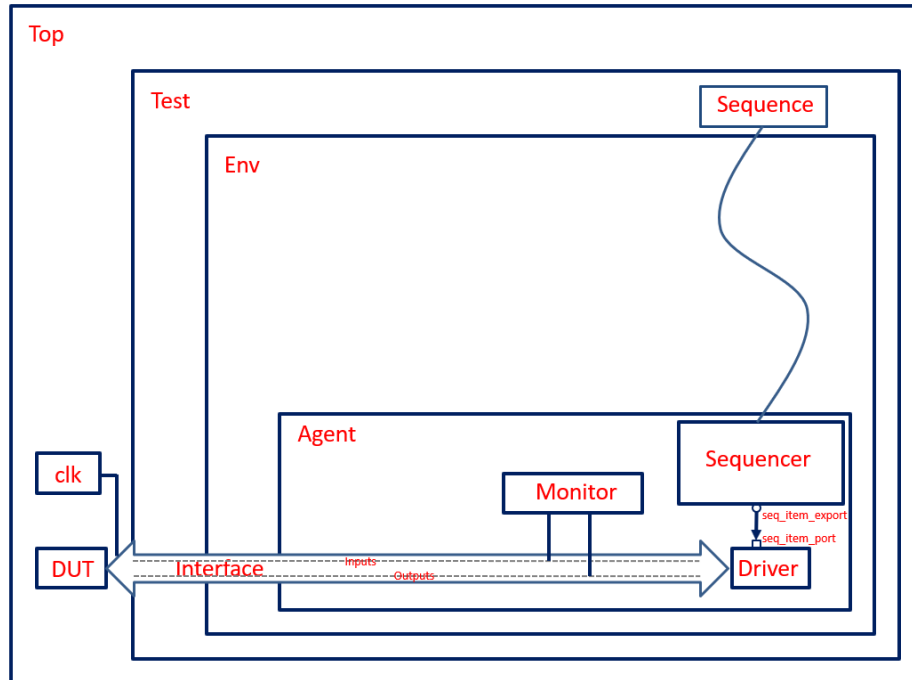
```
class hamming_test extends uvm_test;
  `uvm_component_utils(hamming_test)
  hamming_env hm_env;

function new(string name, uvm_component parent);
  super.new(name, parent);
endfunction: new

function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  hm_env = hamming_env::type_id::create(.name("hm_env"), .parent(this));
endfunction: build_phase

task run_phase(uvm_phase phase);
  hamming_sequence hm_seq;
  phase.raise_objection(.obj(this));
  hm_seq=hamming_sequence::type_id::create(.name("hm_seq"), .ctxt(get_full_name()));
  assert(hm_seq.randomize());
  hm_seq.start(hm_env.hm_agent.hm_seqr);
  phase.drop_objection(.obj(this));
endtask: run_phase
endclass: hamming_test
```

קוד מס' 12 : קוד מימוש ה- test



איור מס' 12 : סביבת האימות לאחר הוספת ה- testbench

מבנה ה- testbench

```

`include "hamming_pkg.sv"
`include "hamming.v"
`include "hamming_if.sv"

module hamming_tb_top;
    import uvm_pkg::*;

    //Interface declaration
    hamming_if vif();

    //Connects the Interface to the DUT
    hamming dut(vif.sig_clock, vif.sig_x, vif.sig_z);

    initial begin
        //Registers the Interface in the configuration block so that other blocks can use it
        uvm_resource_db#(virtual hamming_if)::set (.scope("ifs"), .name("hamming_if"), .val(vif));
        //Executes the test
        run_test();
    end

    //Variable initialization
    initial begin
        vif.sig_clock <= 1'b1;
    end

    //Clock generation
    always
        #5 vif.sig_clock = ~vif.sig_clock;
endmodule

```

קוד מס' 13 : קוד מימוש ה- top level testbench

## מחלקת ה-Scoreboard

כמעט בכל סביבת אימות מופיעה גם מחלקת ה-scoreboard. הסבר על המחלקה יובא במהלך ביצוע הניסוי.

עד כאן התיאור של סביבת אימות טיפוסית. כעת נתאר את ה-DUT שישמש אותנו בסעיפים רבים של הניסוי.

## מקודד Hamming – ה-DUT

ה-DUT שישמש אותנו ברוב חלקי הניסוי הראשון הוא מימוש של מקודד hamming. להלן תיאור קצר. קוד Hamming הוא קוד המאפשר גילוי ותיקון של שגיאה בודדת. נבנה מערכת לקידוד שבע סיביות  $x_6..x_0$ . על מנת לבצע גילוי ותיקון של שגיאה דרושות 4 סיביות נוספות  $h_3...h_0$ . למילה המקודדת המבנה הבאה:

	$x_6$	$x_5$	$x_4$	$h_3$	$x_3$	$x_2$	$x_1$	$h_2$	$x_0$	$h_1$	$h_0$
דוגמא	1	0	0	?	1	1	0	?	1	?	?
מיקום	11	10	9	8	7	6	5	4	3	2	1

חישוב ערך  $h_3...h_0$  מתבצע באופן הבא : יש לסכם את המיקום (המיוצג בבינרי) של כל הסיביות בעלי ערך '1'. הסכום הוא modulo 2 עבור כל סיבית בנפרד. עבור הדוגמא הנ"ל :

$$\begin{aligned}1011 &= 11 \\0111 &= 7 \\0110 &= 6 \\0011 &= 3\end{aligned}$$

-----  
**1001**

ולכן המילה המקודדת תהיה : 1 0 0 1 1 1 0 0 1 0 1 : המפענח מבצע סיכום דומה של כל המקומות בעלי ערך '1' (כולל ה- $h$  - ים). בדוגמא הנ"ל :

$$\begin{aligned}1011 &= 11 \\1000 &= 8 \\0111 &= 7 \\0110 &= 6 \\0011 &= 3 \\0001 &= 1\end{aligned}$$

-----  
0000

אם מתקבל ערך "0000" סימן שאין שגיאה. נניח שסיבית מס' 11 התהפכה בטעות. במקרה זה היה מתקבל :

$$\begin{aligned}1000 &= 8 \\0111 &= 7 \\0110 &= 6 \\0011 &= 3 \\0001 &= 1\end{aligned}$$

-----  
11 = 1011

סימן שיש טעות במיקום 11.

## הכנה ניסוי מספר 1

בחלק הראשון של הניסוי נבצע את הסעיפים הבאים :

1. מקודד Hamming : סימולציות קונבציונליות
2. סימולציות עם יצירה אוטומטית ורנדומלית של הכניסות
3. שינוי של מחלקה בעזרת מנגנון ה- UVM Factory
4. בדיקה של סדר ביצוע הפאזות
5. הכרה והוספת scoreboard ו- reference model לסביבה

**שאלות הכנה :**

### 1. מקודד Hamming : סימולציות קונבציונליות

**שאלה 1 :** תכנן וצייר סכמה שמממשת מקודד Hamming למילה בת 7 סיביות.  
במהלך הניסוי תקבל מימוש המקודד בשפת Verilog. להלן תיאור הממשק שלו :

```
module hamming ( x , z );  
    input  [7:1] x ;      // The seven-bit input  
    output [11:1] z ;     // The 11-bit output  
    reg [11:1] z ;
```

**שאלה 2 :** רשום קובץ סימולציה בשפת Verilog כדי שאפשר יהיה לבדוק את המקודד - hamming\_tb.v. **שים לב** שמחזור השעון המוגדר בקובץ verilog הוא 10ns. ה- default של יחידות הזמן הוא ns.

**שאלה 3 :** הסבר בקיצור מה זה מנגנון ה- factory .

**שאלה 4 :** מה זה reference model ?

**שאלה 5 :** מה הפאזות העקריות של המחלקות בסביבת ה- UVM ? הסבר בקיצור את התפקיד של כל הפאזה.

**שאלה 6 :** אלו מודולים מופיעים בבלוק העליון ?

**שאלה 7 :** באיזו מחלקה מגדירים את מבנה הכניסות ל- DUT ?

**שאלה 8 :** איזו מחלקה יוצרת סידרה של הכניסות שמוזנות ל- DUT ?

**שאלה 9 :** איזו מחלקה דוחפת כניסות ל- DUT ?

**שאלה 10 :** איזו מחלקה דוגמת את יציאות ה- DUT ?

**שאלה 11 :** בעזרת איזה מנגנון מועברות transactions מה- sequencer ל- driver. הסבר בקיצור את פעולת המנגנון.

**שאלה 12 :** איזו מתודה גורמת להפעלת ה- test ?

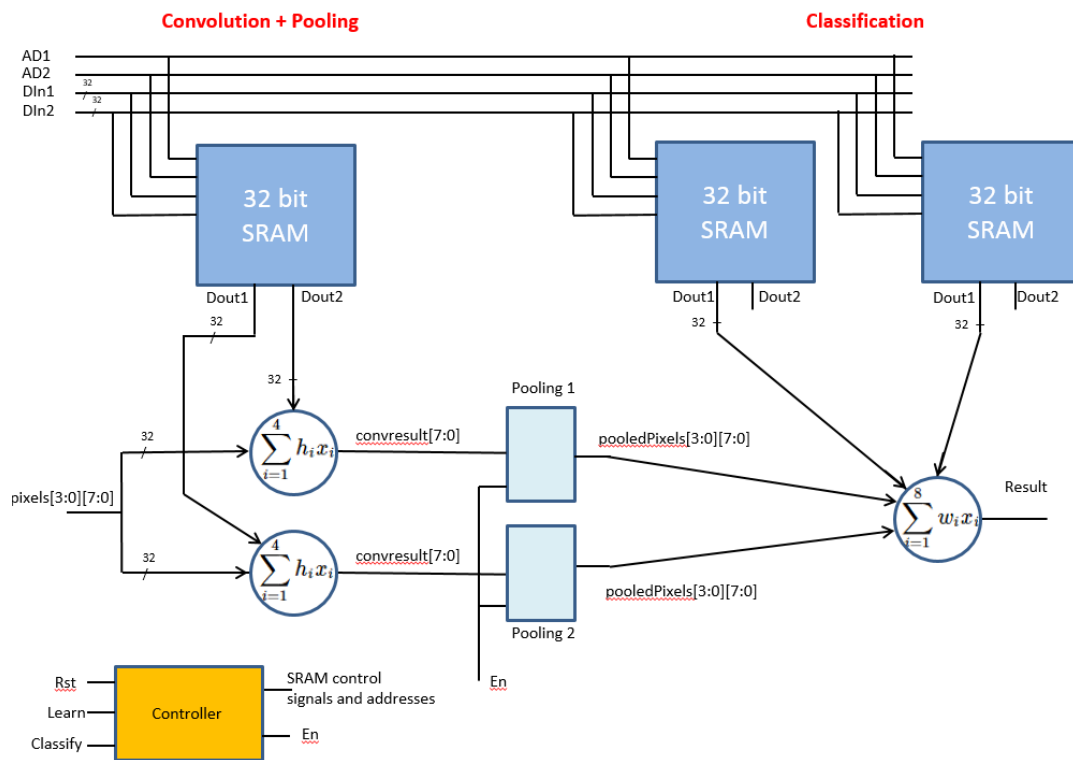
**שאלה 13 :** רשום את המשפט שהתבקשת לרשום בעמ' 13.

## הכנה ניסוי מספר 2

רכיב ה-DUT עבור החלק השני של הניסוי הוא מימוש מאיץ של מערכת לומדת שממושת בניסוי אחר של המעבדה, כלומר ניסוי 98 - תכנון ארכיטקטורה למאיץ עבור מערכת לומדת ממושת ב-Systemverilog.

### חשוב : ניסוי 98 אינו מהווה קדם לניסוי זה.

אין צורך להבין את כל פרטי הלוגיקה. רק חשוב להכיר את הממשק שלו ואת האופן שבו מוזנות כניסות למעגל. המעגל פועל בשני שלבים, לאחר ה- reset (שני מחזורים) מתבצע השלב הלימוד (שני מחזורים) ולאחר מכן (כל יתר הזמן) מתבצע שלב הסיווג. הכניסה למעגל היא למעשה תמונה בגודל  $3 \times 3$  פיקסלים ששמורה במערך בשם InputImage. מערך זה מופיע ב- testbench ואינו מופיע באיור מס' 13. בשלב הלימוד נכתב מידע לזיכרונות. בשלב הסיווג מוזנים ל- InputImage 32 – pixels (מתוך InputImage) בכל מחזור שעון. כל 4 מחזורי שעון מזינים ערך חדש ל- InputImage. להלן סכמת המלבנים של המערכת :



### איור מס' 13 – הארכיטקטורה של מאיץ של מערכת לומדת

כפי שניתן לראות, המערכת מורכבת משלש יחידות SRAM, שני Convolution Neurons, שתי יחידות Pooling, Fully Connected Neuron אחד והבקר. הסבר מלא על התכנון מופיע בחוברת של ניסוי 98 אבל לצרכים של הניסוי הזה, אין צורך להבין את הפונקציונליות של המעגל. לאחר תיאור ה-DUT נמשיך בהסבר על מנגנון נוסף של UVM, מנגנון ה-TLM-FIFO.

### מנגנון ה-TLM-FIFO

קיימים מקרים רבים שיחידות מידע מועברים בין שני רכיבים. על מנת לאפשר לשני רכיבים לפעול בצורה בלתי תלויה, ניתן להעביר את יחידות המידע דרך רכיב FIFO. רכיב A יכול לשלוח יחידות מידע ללא קשר למצב של רכיב B. מצד שני רכיב B יוכל למשוך את היחידות בזמן שנוח לו ללא קשר ל- A.



### איור מס' 14 : שימוש ב- FIFO

כפי שמתואר בקוד הבא רכיב A כותב ל- FIFO באמצעות מתודת ה- `put()` השייכת ל- `port` שלו, ורכיב B מושך את היחידות באמצעות מתודת ה- `get()` השייכת ל- `export()` שלו. ראה הסבר על TLM לעיל.

```

class my_env extends uvm_env;
  `uvm_component_utils (my_env)

  componentA compA;
  componentB compB;

  // Create the UVM TLM Fifo that can accept simple_packet
  uvm_tlm_fifo #(simple_packet) tlm_fifo;

  function new (string name = "my_env", uvm_component parent = null);
    super.new (name, parent);
  endfunction

  virtual function void build_phase (uvm_phase phase);
    super.build_phase (phase);
    // Create an object of both components
    compA = componentA::type_id::create ("compA", this);
    compB = componentB::type_id::create ("compB", this);

    // Create a FIFO with depth 2
    tlm_fifo = new ("uvm_tlm_fifo", this, 2);
  endfunction

  // Connect the ports to the export of FIFO.
  virtual function void connect_phase (uvm_phase phase);
    compA.put_port.connect (tlm_fifo.put_export);
    compB.get_port.connect (tlm_fifo.get_export);
  endfunction

  // Display a message when the FIFO is full
  virtual task run_phase (uvm_phase phase);
    forever begin
      #10 if (tlm_fifo.is_full ())
        `uvm_info ("UVM_TLM_FIFO", "Fifo is now FULL !", UVM_MEDIUM)
    end
  endtask
endclass
  
```

### קוד מס' 14 : קוד המציג חיבור TLM

## אילוצים על ערכים מוגרלים

קיימות שיטות רבות לגרום לערכים מוגרלים לקיים אילוצים המוגדרים. נכיר מספר דרכים.

דוגמא 1 :

```
`define MAX_D 100
`define MIN_D 50
class Base;
    rand integer Var;
    constraint range { Var < MAX_D ; Var > MIN_D ;}
endclass
```

דוגמא 2 :

```
class set_mem;
    rand bit [0:2] Var;
    constraint range { !( Var inside {0,1,5,6});}
endclass
```

**שאלה 1 :** הסבר מה זה analysis\_port ו-analysis\_export וכיצד ניתן להשתמש בהם על מנת להעביר transactions. רשום משפטים טיפוסיים שדרושים למימוש העברת transactions מיחידה ליחידה.

**שאלה 2 :** הסבר את מנגנון ה-TLM\_FIFO.

**שאלה 3 :** הסבר בקיצור את סוגי הכיסוי שמוזכרים במסמך זה.

**שאלה 4 :** מה חשוב לכסות במימוש של מוכנת מצבים ?

**שאלה 5 :** מה המשמעות של ציון 100% עבור בדיקת כיסוי ?

**שאלה 6 :** הסבר את הקוד של דוגמא 1.

**שאלה 7 :** הסבר את הקוד של דוגמא 2.

**שאלה 8 :** בעזרת ההסבר ב:

[http://www.testbench.in/CR\\_15\\_CONSTRAINT\\_EXPRESSION.html](http://www.testbench.in/CR_15_CONSTRAINT_EXPRESSION.html)

הגדר משתנה Var מסוג integer אשר מקיים את הפילוג הבא כאשר מגרילים אותו :

**var = 1 – 10% ,      var = 2 – 20% ,      var = 3 – 30% ,      var = 4 – 40%**

להלן מימוש ה- transaction :

```
class NeuralNet_transaction extends uvm_sequence_item;
    rand logic [71:0] InputImage;
    logic [7:0] result;

    function new(string name = "");
        super.new(name);
    endfunction: new

    `uvm_object_utils_begin(NeuralNet_transaction)
        `uvm_field_int(InputImage, UVM_ALL_ON)
    `uvm_object_utils_end
endclass: NeuralNet_transaction
```

**שאלה 9 :** רשום משפט להוספה של constraint בשם c\_InputImage למחלקה NeuralNet\_transaction על מנת להגביל את הערכים המוגרלים ל-

'72h01ff01ff01ff01ff01,72'hff01ff01ff01ff01ff,72'h01ffffff01ffffff01,72'hffff01ff01ff01ffff



## ביצוע ניסוי מס' 1

### 1. מקודד Hamming : סימולציות רגילות

עבור לספריה בשם HammingStart בעזרת :

cd HammingStart

בסעיף זה, תתבצע סימולציה של קוד verilog בעזרת ערכי כניסה שהוכנו באופן ידני בקובץ hamming\_tb.v.

- רשום את hamming\_tb.v כפי שהכנת בבית.
- הפעל את הסימולטור verilog (vcs) על מנת לבדוק את הנכונות הלוגית של המקודד hamming.v. מכיל את תיאור ה-verilog של המקודד. הרץ :

vcs -R -gui -full64 -sverilog -debug\_all hamming\_a.v hamming\_tb.v

simv -gui

- יופיעו שני חלונות.
- להצגת צורות הגל : סמן את היחידה hamming\_test בחלון DVE. לחץ על Ctrl-4.
- הרצת הסימולציה : לחץ על חץ ↓ בצד שמאל
- Q11 : הראה ששתי התוצאות הרשונות נכונות.
- Q12 : צרף את צורות הגל לדו"ח.
- סגור את DVE עם File->Exit.

המשך הניסוי יעסוק במימוש סביבת אימות הכוללת :

- יצירה אוטומטית של כניסות
- הכרת מנגנון ה-factory
- תרגול phases
- הוספת מימוש reference
- הוספת יחידת scoreboard
- השוואת תוצאות ה-DUT עם ה-reference

### 2. סימולציות עם יצירה אוטומטית ורנדומלית של הכניסות

נתחיל עם המבנה הבסיסי ביותר של סביבת הווריקציה שממומש בקבצים :

hamming\_sequencer.v  
hamming\_driver.v  
hamming\_monitor.v  
hamming\_agent.v  
hamming\_env.v  
hamming\_test.v

איחוד/קריאה של כל הקבצים תתבצע בעזרת קובץ:

hamming\_pkg.v

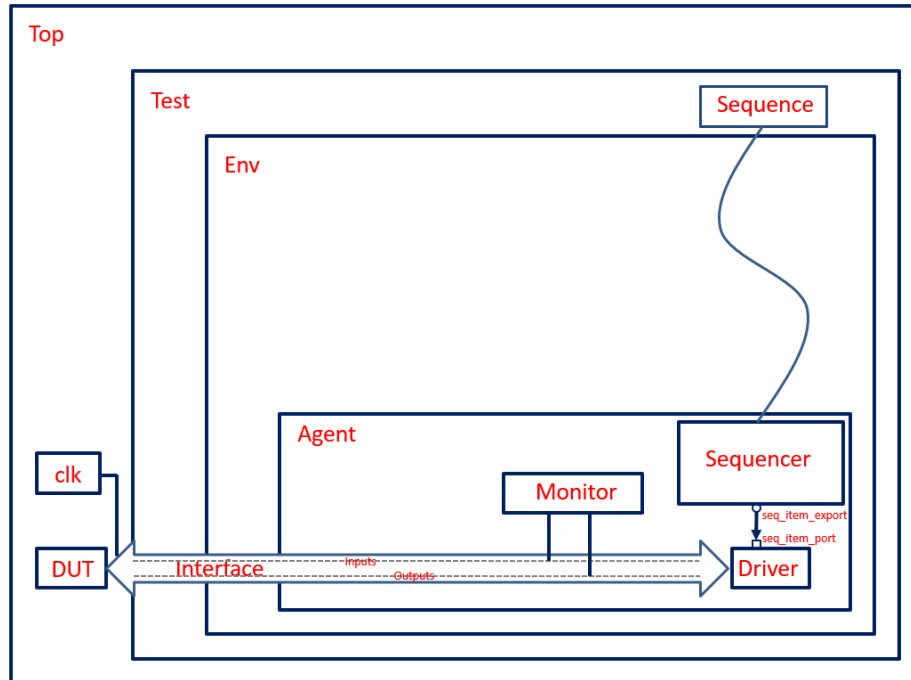
המימוש והמשק :

hamming.v  
hamming\_if.v

וה- testbench ב :

hamming\_tb\_top.v

שים לב שספריית העבודה HammingStart כבר מכילה את כל הקבצים שמתארים את סביבת הווריקציה שבאיור 15. **פתח את הקבצים לקריאה וודא שהינך מבין את התוכן שלהם.**



איור מס' 15 : סביבה ממומשת ב- HammingStart

לקומפילציה הרץ :

make

הפקודה :

make | grep "integral"

תציג רק את השורות עם x ו- z.

Q21 : הוסף פלט זה לדו"ח. כמה כניסות הוגרלו ?

הפעל את הסימולטור עם חלון גלים והוסף את האותות כפי שהוסבר בסעיף הקודם:

simv +UVM\_TESTNAME=hamming\_test -gui

לחץ על ה- "+" שליד hamming\_tb\_top. בחר ב- dut(hamming) ולחץ על 4 CNTR. יפתח חלון עבור הגלים.

הרצת הסימולציה : לחץ על חץ ↓ בצד שמאל.

Q22 : הוסף את חלון הגלים לדו"ח.

בחלון ה- DVE בצע File->Exit.

שנה את מספר הכניסות שמוגרלות ל- 20 והרץ את הסימולציה שוב בעזרת :

make | grep "integral"

Q23 : הסבר בדיוק כיצד עשית זאת ובאיזה קובץ. כמה כניסות הוגרלו ?

שנה את בדיקה כל שייווצרו כל 127 הכניסות האפשריות והרץ את הסימולציה שוב.

Q24 : הסבר בדיוק כיצד עשית זאת ובאיזה קובץ.

בעזרת הפונקציה :

\$dist\_uniform(seed,low,high);

למשל :

hm\_tx.x = \$dist\_uniform(seed,0,127);

הגבל את הערכים המוגרלים להיות בין 30 ל- 100. שנה את מספר הכניסות המוגרלות ל- 20. הרץ את הסימולציה שוב

הפעם בעזרת :

make | grep "integral"

על מנת לראות רק את הכניסות x.

Q25 : הסבר בדיוק כיצד עשית זאת ובאיזה קובץ והוסף את הכניסות המוגרלות לדו"ח.

### 3. בחירת מחלקה בעזרת ה- Factory

כאמור, מבנה הנתונים של ה- factory מאפשר קביעת שם ה- test בזמן קומפילציה ללא צורך בשינויים רבים בקוד. עבור לספריה Factory בעזרת :

cd ../Factory

הרץ את הסימולציה בעזרת :

- make | grep UVM\_INFO

Q31 : הוסף את השורות המודפסות לדו"ח. מאילו מחלקות מודפסות הודעות ה- HELLO ?

הוסף לקוד את השורה שהכנת שגורם ל- env.a1 להיות מסוג A\_ovr. הרץ את ה- test בעזרת :

- make | grep UVM\_INFO

Q32 : הוסף את השורות המודפסות לדו"ח. מאילו מחלקות מודפסות הודעות ה- HELLO ? הסבר האם אלו ההדפסות שציפת ?

הוסף לקוד את השורה שהכנת שגורם ל- env.a2 להיות מסוג A\_override. הרץ את ה- test בעזרת :

- make | grep UVM\_INFO

Q33 : הוסף את השורות המודפסות לדו"ח. מאילו מחלקות מודפסות הודעות ה- HELLO ? הסבר האם אלו ההדפסות שציפת ?

כעת נכיר שיטה נוספת לביצוע override, הפעם בעזרת פקודת הקומפילציה ב- Makefile.

בטל את כל משפטי ה- override שהוספת. פתח את קובץ ה- Makefile. בשני שלבים, בצע את שני ה- overrides הנ"ל ע"י שינוי שורת ה- SIMV בהתאם לדוגמא :

```
SIMV = ./simv +UVM_VERBOSITY=$(UVM_VERBOSITY) \
+UVM_TESTNAME=$(TEST) +UVM_TR_RECORD +UVM_LOG_RECORD \
+uvm_set_inst_override=A,A_ovr,uvm_test_top.env.a1 \
+verbose=1 +ntb_random_seed=244 -l vcs.log
```

הרץ שוב את הסימולציה כמו קודם.

Q34 : הוסף את ההדפסות הרלוונטיות לדו"ח. האם קיבלת מה שציפת? מה ההסבר לפלט שקיבלת ?

תקן את השורה שהוספת ל- makefile כך שיתבצע גם override של A ב- A\_ovr עבור a2. הראה את התיקון למנחה.

הרץ את ה- test בעזרת :

- make | grep UVM\_INFO

Q35 : הוסף את התיקון ל- makefile ואת ההדפסות הרלוונטיות לדו"ח. האם קיבלת מה שציפת? מה ההסבר לפלט שקיבלת הפעם ?

Q36 : סכם את השיטות לביצוע override שבוצעו בניסוי.

### 4. סדר ביצוע הפאזות

עבור לספריה HammingPhase בעזרת :

cd ../HammingPhase

הוסף משפט הדפסה לכל פאזה של כל מחלקה שמאפשרת לדעת איזו פאזה בדיוק הדפיסה את ההודעה. למשל :

```
`uvm_info("", "Connect_phase of hamming_env", UVM_MEDIUM);
```

יש להוסיף פקודת הדפסה גם ל- body() של ה- sequence. כמובן שיש לשנות את המילים Connect\_phase למילים שישקפו את שם השיגרה שמדפיסה.

הרץ את הטסט בעזרת :

make

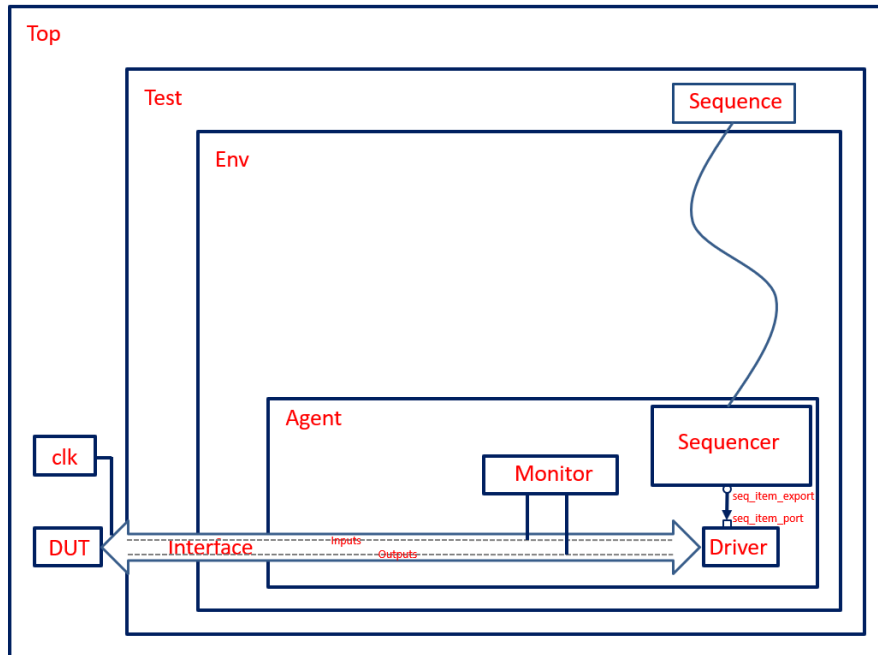
Q41 : הוסף לדו"ח את הפלט של כל משפטים שהדפסת שהוספת לקוד.

Q42 : לפי איזה סדר מבוצעות הפאזות של המחלקות השונות ?

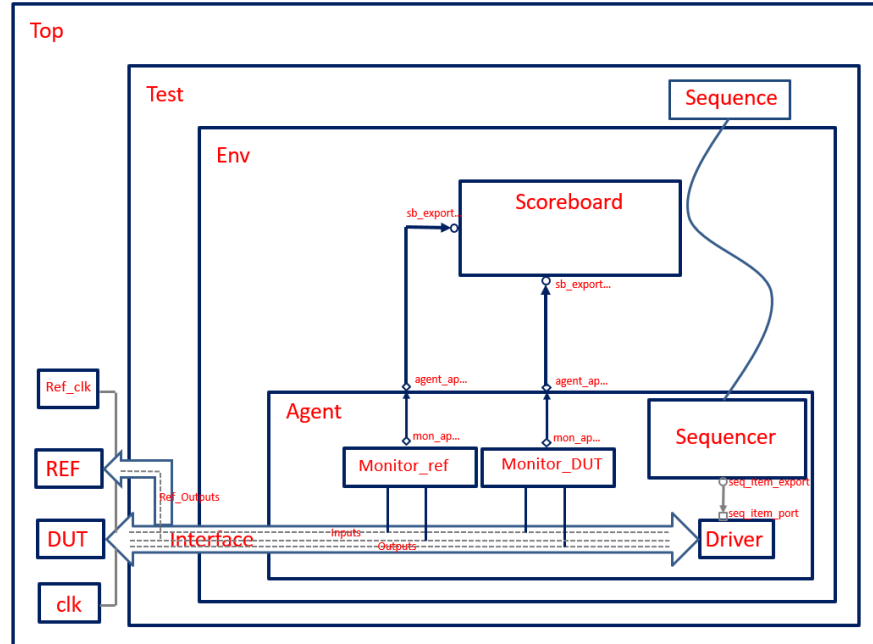
Q43 : כמה פעמים מבוצעת כל פאזה ?

## 5. הוספת scoreboard ו- reference model לסביבה

המטרה של סעיף זה היא לעבור מהמבנה הקיים שאינו מכיל reference model ו- scoreboard (איור מס' 16) למבנה הכולל את שתי היחידות החדשות (איור מס' 17) :



איור מס' 16 : סביבת באימות ללא reference model ו- scoreboard



איור מס' 17 : סביבת באימות כולל reference model ו- scoreboard

הערה : הכניסות והיציאות מועברים באותו transaction דרך פורט אחד בלבד.  
עבור לספריית HammingRef בעזרת :

cd ../HammingRef

בסעיף הזה נרצה להוסיף לסביבה שלנו reference model ונרצה להשוואות בין הפלט של ה-DUT לפלט את ה-reference model בעזרת מחלקה בשם scoreboard.  
מימוש ה-reference model והממשק שלו מופעים בקבצים :

- hamming\_ref.v
- hamming\_ref\_if.v

אם נשווה את האיור של סביבת הבדיקה ההתחלתית לאיור של הסביבה שיש לממש, אפשר לסכם שצריך לבצע את השינויים הבאים :

- hamming\_tb\_top: add reference module and ref\_if. Define ref\_clk
- hamming\_monitor: add second monitor for reference model
- hamming\_drive: drive must also drive reference model with new interface
- hamming\_scoreboard: define this new class
- hamming\_agent: add second monitor, scoreboard and communication channels between monitors and scoreboard.
- hamming\_env: add scoreboard
- hamming\_pkg: add -`include "hamming\_scoreboard.sv"

### תיקון hamming\_tb\_top.sv

- הוסף משפטי include עבור hamming\_ref.v ו-hamming\_ref\_if.sv
- הוסף הצבה (instantiation) של **ref\_vif** של hamming\_ref\_if (הממשק) ו-ref\_u1 של hamming\_ref (מודל רפרנס) בדומה להצבות hamming.v ו-hamming\_if.sv
- הוסף hamming\_ref\_if ל-factory בדומה ל-hamming\_if
- הוסף הגדרה ואיתחול של שעון חדש הממשק (ref\_vif.sig\_clock)

### תיקון hamming\_monitor.sv

בצע את השינויים הבאים לקובץ זה :

- הוסף (מיד אחרי uvm\_component\_utils) הצהרה של analysis port של scoreboard שבאמצעותו יישלחו transactions ל-scoreboard
- ל-build phase צור אובייקט מסוג זה בעזרת :  
uvm\_analysis\_port#(hamming\_transaction) mon\_ap\_dut;
- ב-run phase מיד אחרי המשפטים :  
mon\_ap\_dut = new(.name("mon\_ap\_dut"), .parent(this));
- הוסף משפט כתיבה של ה-transaction לפורט אחרי דגימת הממשק :  
mon\_ap\_dut.write(hm\_tx);
- הוסף monitor נוסף בשם hamming\_monitor\_ref.
- ראשית, שכפל את כל שורות הקוד ועל העותק בצע את השינויים הנדרשים.
- שנה את hamming\_monitor\_dut ל-hamming\_monitor\_ref.
- שם הממשק החדש הוא hamming\_ref\_if.
- וודא שההצרה על הממשק פונה לממשק החדש.
- וודא שה-analysis\_port הוא בעל שם חדש mon\_ap\_ref.
- וודא שהפניה ל-factory ב-build phase מתבצעת עם הממשק החדש
- וודא ש-b run phase הזנת הכניסות היא לממשק ה-reference ושעובדים עם השעון החדש.

- וודא ש-hamming\_monitor\_ref הוא בעל שם חדש mon\_ap\_ref.
- וודא שהפניה ל-factory ב-build phase מתבצעת עם הממשק החדש
- וודא ש-b run phase הזנת הכניסות היא לממשק ה-reference ושעובדים עם השעון החדש.

### תיקון hamming\_driver.sv

- הוסף הצרה של ref\_vif מסוג hamming\_ref\_if
- צור אובייקט חדש (ב- build phase) מסוג hamming\_ref\_if בעזרת ה- factory
- הוסף משפט שמאפס את : ref\_vif.sig\_x
- הוסף משפט שמזין ערכים ל- ref\_vif.sig\_x

### תיקון hamming\_agent.sv

- באמצעות שני המשפטים הבאים, יש להצהיר על שני analysis ports על מנת לאפשר העברת transactions ל- scoreboard (אחרי uvm\_component\_utils):

```
uvm_analysis_port#(hamming_transaction) agent_ap_dut;  
uvm_analysis_port#(hamming_transaction) agent_ap_ref;
```

- אחרי הצרת ה- hm\_mon\_dut, הוסף הצהרה של המשתנה hamming\_monitor\_ref מסוג hm\_mon\_ref
- אחרי משפט super.build\_phase(phase), באמצעות שני המשפטים הבאים, יש להוסיף ל- build phase שני analysis ports על מנת לאפשר העברת transactions ל- scoreboard.

```
agent_ap_dut = new(.name("agent_ap_dut"), .parent(this));  
agent_ap_ref = new(.name("agent_ap_ref"), .parent(this));
```

- בדומה ל- hm\_mon\_dut, צור אובייקט חדש (ב- build phase) מסוג hamming\_monitor\_ref בעזרת ה- factory
- באמצעות שני המשפטים הבאים, יש להוסיף ל- connect phase מחברים בין ה- **export של ה- monitor ל- port של ה- agent.**

```
hm_mon_dut.mon_ap_dut.connect(agent_ap_dut);  
hm_mon_ref.mon_ap_ref.connect(agent_ap_ref);
```

שים לב שזה חיבור בין port ל- port ולא port ל- export. זה יבוא בהמשך.

### תיקון hamming\_env.sv

- אחרי משפט ה- "hamming\_agent hm\_agent", הוסף הצהרה של המשתנה hm\_sb מסוג hamming\_scoreboard
- בדומה ל- hm\_agent, צור אובייקט חדש (ב- build phase) מסוג hamming\_scoreboard בעזרת ה- factory

```
hm_sb = hamming_scoreboard::type_id::create(.name("hm_sb"), .parent(this));
```

- באמצעות שני המשפטים הבאים, יש להוסיף ל- connect phase חיבור בין ה- **export של ה- agent ל- port של ה- scoreboard.**

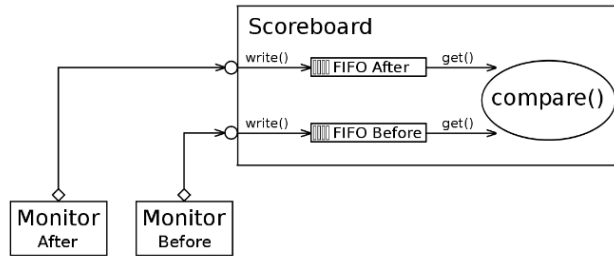
```
hm_agent.agent_ap_dut.connect(hm_sb.sb_export_dut);  
hm_agent.agent_ap_ref.connect(hm_sb.sb_export_ref);
```

### תיקון hamming\_pkg.sv

- הוסף משפט include עבור hamming\_scoreboard.sv (אחרי ה- agent).

### יצירה של hamming\_scoreboard.sv

- Q51 : קובץ זה ניתן לכם מוכן. פתח את הקובץ ובעזרת איורים 14, 17 ו- 18, הסבר בקיצור את התוכן שלו. הסבר בקיצור את התפקיד של כל המתודות של המחלקה.



איור מס' 18 : חיבור ה-scoreboard ל-monitors

הערה : before מתייחס ל-DUT ו-after ל-reference\_model.

- הרץ את הסימולציה עם :

- make | grep Test

Q52 : האם ה-DUT מתנהג באופן זהה ל-reference model ?

Q53 : הוסף לדו"ח את ההדפסות שמראות שהתשובה לסעיף הקודם נכונה.

## ביצוע ניסוי מס' 2

כאמור, רכיב ה-DUT עבור חלק זה של הניסוי הוא מימוש מאיץ של מערכת לומדת (בשם NeuralNet) הממומש בשפת Systemverilog. הגישה בחלק זה של הניסוי היא לספק את כל הקוד של ה-DUT וסביבת האימות. לא תמיד יסופק הקוד המלא. בכל סעיף יהיה צורך לזהות מה הבעיה ולהשלים את החסר בעזרת ההסברים שמופעים במסמך זה וכמובן במידת הצורך גם בעזרת המדריך. בצורה זאת ניתן יהיה להתמקד בסוגיות שונות של סביבת האימות. שים לב שמסופק גם reference model.

עבור לספריית ML\_tlm. ספרייה זאת מכילה את קבצי ה-DUT :

- cneuron.sv
- fcneuron.sv
- pooling.sv
- dpram32x32\_cb.v
- NeuralNet\_cont.sv
- NeuralNet.sv - top level module

מימוש ה-reference model :

- NeuralNet\_Ref.sv

מימוש המימשקים ל-DUT ול-reference model :

- NeuralNet\_if.sv
- NeuralNet\_Ref\_if.sv

מימוש כל קיבצי סביבת האימות :

- NeuralNet\_sequencer.sv
- NeuralNet\_driver.sv
- NeuralNet\_monitor.sv
- NeuralNet\_agent.sv
- NeuralNet\_scoreboard.sv
- NeuralNet\_env.sv
- NeuralNet\_pkg.sv

- NeuralNet\_test.sv
- NeuralNet\_test\_tb.sv

## 1. העברת מידע בעזרת Transaction Level Modeling Ports

המטרה של סעיף זה היא להתעמק בכל הנושא של העברת מידע מיחידה ליחידה. ה- monitor דוגם את המידע, אורז אותו ב- transaction ומעביר את ה- transaction ליתר היחידות.  
 - הרץ את הסימולציה באמצעות הפקודה :

make | grep -i Test

Q11 : האם ה- test עובר בהצלחה ?

- הרץ את הסימולציה שוב באמצעות הפקודה :

make

Q12 : דפדף בפלט של הריצה ובדוק את הערכים של Result של ה- DUT ושל ה- ref ? האם הם זהים ?

- שים לב שההדפסות של Result מתבצעות ב- monitor. פתח את הקובץ NeuralNet\_monitor.sv.

Q13 : בדוק את לולאת ה- forever של NeuralNet\_monitor\_ref. השוואה עם NeuralNet\_monitor\_dut. מה

חסר ? מה מבצעות משפטים אלה ? תקן את הקובץ והרץ את make | grep Test. רשום את התיקון לדו"ח.

Q14 : האם ה- test עובר בהצלחה ? איזו יחידה פולטת את הודעת הכישלון ?

- הודעת הכישלון נפלטת מה- scoreboard :

- uvm\_test\_top.ml\_env.ml\_sb [compare] Test: Fail!

- פתח את הקובץ NeuralNet\_scoreboard.sv. שים לב שקיימות יחידות בעלות שם xxx\_dut שתפקידן לטפל ב-

transactions שמקורם ב- dut. חייבות להיות יחידות דומות לטיפול ב- transactions שמקורם ב- ref model.

הוסף לקובץ את כל היחידות החסרות. שים לב שה- scoreboard מבצע את שגרת ה- compare ופולט :

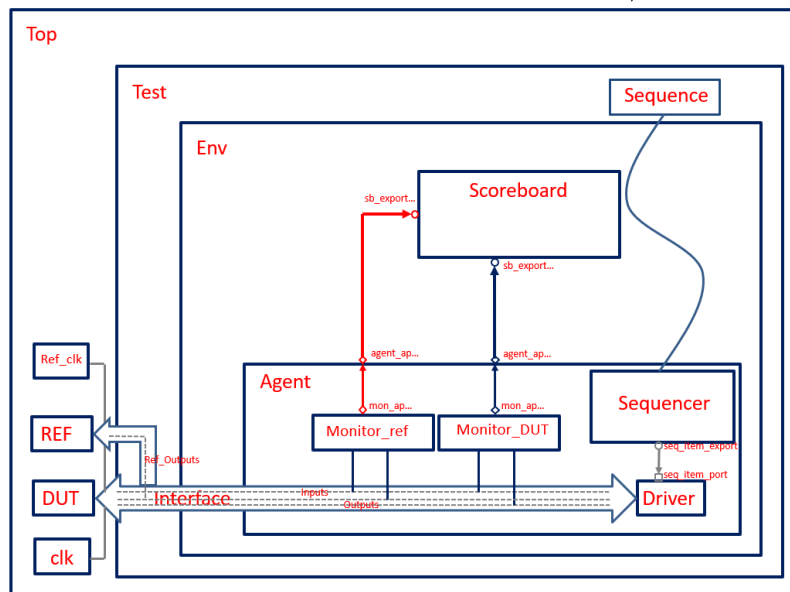
Test: Fail או OK

Q15 : הסבר בצורה מפורטת את משמעות כל השורות שהוספת לקובץ. מה מבצעות משפטים אלה ?  
 - בצע

make | grep test

- ניתן לראות שהבדיקה כבר לא פולטת Test: OK או Test: Fail כמו קודם. המשמעות היא שעכשיו- compare() אינה מתבצעת. הסיבה לכך היא שהקריאה מה- ref\_fifo (שהוספנו) אינה מצליחה!.

Q16 : נסה להסביר מדוע. מיד נתעמק בנושא.





- בדוק את איור 19. מסתבר שכל היחידות שממשות את המסלול האדום גם חסרות. הוסף את היחידות החסרות ל-  
NeuralNet\_env.sv ול- NeuralNet\_agent.sv.  
הערה: המסלול המקביל מה- DUT קיים. העזר בו על מנת לבנות את המסלול החסר.

Q17: הסבר בצורה מפורטת את משמעות כל השורות שהוספת לשני הקובצים. מה מבצעות משפטים אלה?  
- הרץ את הסימולציה באמצעות הפקודה:

```
make | grep Test
```

Q18: האם ה- test עובר בהצלחה? הסבר את ה- Fails שמופיעים בהתחלה.

Q19: סכם את המנגנון שבאמצעותו מעבירים את יציאות ה- DUT וה- reference model אל הפונקציה compare שב- scoreboard.

## 2. כיסוי הבדיקות Coverage

כאמור, coverage מהווה מדד של יכולת הבדיקות לעבור על כל הפונקציונליות של התכנון. בסעיף זה נכיר סוגים שונים של coverage. עבור לספריית ML\_Cov בעזרת:

```
cd ../ML_Cov
```

כמו שבסעיף הקודם, מופיעים כאן כל קבצי המימוש וכל קבצי סביבת האימות.

### כיסוי בעזרת הוראות קומפילציה

- הרץ את הסימולציה באמצעות הפקודה:

```
make
```

Q21: תוצאות הכיסוי נשמרים בספרייה בעלת שם עם סיומת vdb. האם קיימת ספרייה בשם זה?

- פתח את ה- Makefile והוסף את האופציות הבאות לפקודות הקומפילציה. יש לעשות זאת לקטע של ה- VCS ולמקם את השורות לפני השורה שמתחילה ב- \$(UVM\_HOME):

```
-cm_cond allops+anywidth+event -cm_noconst\  
-cm line+cond+fsm+branch+tgl -cm_dir ./coverage.vdb\  
Q22: הרץ את הפקודה vcs -help. הסבר כל אחד של האופציות בשתי השורות הנ"ל.
```

- הרץ את הסימולציה שוב באמצעות הפקודה:

```
make
```

Q23: תוצאות הכיסוי נשמרים בספרייה בעלת שם עם סיומת vdb. האם קיימת ספרייה בשם זה?

- על מנת לראות את תוצאות הכיסוי בצע:

```
urg -dir coverage.vdb
```

```
cd urgReport
```

```
chrome dashboard.html
```

- לחץ על modlist.

Q24: הוסף לדו"ח את הטבלה שמסכמת את ה- coverage?

Q25: עבור מכונת המצבים:

א. רשום את אחוז הכיסוי של השורות. רשום דוגמא של שורה שלא כוסתה.

ב. רשום את אחוז הכיסוי של התנאים. רשום דוגמא של תנאי שלא כוסה.

ג. רשום את אחוז הכיסוי של המצבים. רשום דוגמא של מצב שלא כוסה.

ד. רשום את אחוז הכיסוי של המעברים. רשום דוגמא של מעבר שלא כוסה.

- סגור chrome ובצע "cd .." על מנת לחזור לספריית העבודה

## הגדרת ספציפיות של המעברים

לעתים המשתמש רוצה לבדוק רק מעברים מסוימים. במקרה כזה הוא יכול להגדיר לבד את המעברים שיש לכסות.

- פתח את הקובץ NeuralNet\_cont.sv והוסף לו את השורות הבאות בשורה 26:

```

covergroup st_cg @ (posedge clk);
STATE_TRANSITIONS : coverpoint CUR_ST // WORKING
{
  bins S0_S0 = (Idle_st => Idle_st);
  bins S0_S1 = (Idle_st => Learn_st);
  bins S0_S2 = (Idle_st => Classify_st);
  bins S1_S1 = (Learn_st => Learn_st);
  bins S1_S0 = (Learn_st => Idle_st);
  bins S1_S2 = (Learn_st => Classify_st);
  bins S2_S2 = (Classify_st => Learn_st);
  bins S2_S0 = (Classify_st => Idle_st);
  bins S2_S1 = (Classify_st => Learn_st);
  bins S2_S1_S0 = (Classify_st => Learn_st=>Idle_st);
}
endgroup

covergroup s_cg @ (posedge clk);
STATES : coverpoint CUR_ST // WORKING
{
  bins ST0 = {Idle_st};
  bins ST1 = {Learn_st};
  bins ST2 = {Classify_st};
}
endgroup

st_cg stcg;
s_cg scg;

initial begin
  scg = new;
  stcg = new;
end

```

Q26 : הסבר הקיצור את השורות שנוספו. קרא למנחה ותציג לו את ההסברים שהוספת.  
 - הרץ את הסימולציה שוב באמצעות הפקודה :

make

- על מנת לראות את תוצאות הכיסוי בצע :

```

urg -dir coverage.vdb
cd urgReport
chrome dashboard.html


```

- לחץ על groups. כאן ניתן לראות את סיכום תוצאות הכיסוי עבור ה- groups שהגדרנו.  
 Q27 : עבור מכונת המצבים :

א. רשום את אחוז הכיסוי של המצבים. רשום דוגמא של מצב שלא כוסה.  
 ב. רשום את אחוז הכיסוי של המעברים. רשום דוגמא של מעבר שלא כוסה.  
 ג. כמה פעמים כוסה המעבר S2\_S1\_S0. מדוע ?  
 על מנת לענות על שאלה זאת נעזר בצורות הגל. ראשית סגור chrome ובצע "cd .." על מנת לחזור לספריית העבודה.  
 פתח את ה- Makefile ושנה את השורה:  
 \$(SIMV) ל :

\$(SIMV) -gui

בחלון שנפתח, לחץ על ה- + שליד NeuralNet\_tb\_uvm. לחץ על ה- + שליד dut. לחץ על U1 (NeuralNet\_cont).  
בחלון שליד בחר ב- clk וב- CUR\_ST[1:0] (העזר בכפתור Ctrl).

כעת בחר ב- Signals->Add To Waves->New Wave View. ייפתח חלון של צורת גל. לחץ על  להרצת הסימולציה. לחץ על View->Zoom->Zoom Full. בדוק כיצד משתנים המצבים באות CUR\_ST[1:0]. לאחר שענית על שאלה ג' החזר את ה- Makefile למצבו הקודם (ללא -gui) ובצע את הפעולות הבאות :

- שנה את ה- group כך שהתבצע כיסוי גם של : S0\_S1\_S2 ושל S0\_S2\_S1 ו- S0\_S1\_S1\_S2. בקובץ NeuralNet\_sequencer.sv הגדל את מספר ה- transactions שנוצרים ל- 20.

- **הראה את השינויים למנחה!**

- הרץ את הסימולציה שוב באמצעות הפקודה :

make

- פתח את התוצאות כפי שהוסבר לעיל.

Q28 : עבור מכונת המצבים :

א. כמה פעמים כוסו המעברים S0\_S1\_S2, S0\_S2\_S1, S0\_S1\_S1\_S2 ו- S0\_S1\_S1\_S2 ? מדוע ?

### 3. יצירת כניסות ארקאיות עם אילוצים

בסעיף זה אנו נראה כיצד ניתן לקבוע אילוצים על מנת להגביל את הערכים של הכניסות שמוגרלות בזמן יצירת ה- sequence.

עבור לתיקיה : ML\_Constr

cd ML\_Constr

- הרץ את הסימולציה שוב באמצעות הפקודה :

make | grep InputImage

Q31 : הוסף לדו"ח את כל הערכים שהוגרלו עבור InputImage.

- הוסף constraint בשם c\_InputImage למחלקה NeuralNet\_transaction על מנת להגביל את הערכים המוגרלים ל-

'72h01ff01ff01ff01ff01,72'hff01ff01ff01ff01ff,72'h01ffffff01ffffff01,72'hffff01ff01ff01ffff'

- הרץ את הסימולציה שוב באמצעות הפקודה :

make | grep InputImage

Q32 : הוסף לדו"ח את הקוד שרשמת. הוסף לדו"ח את כל הערכים שהוגרלו עבור InputImage.

- הוסף constraint נוסף למחלקה NeuralNet\_transaction על מנת להגביל את הערכים המוגרלים רק ל-

'72h01ff01ff01ff01ff01,72'hff01ff01ff01ff01ff'

- הרץ את הסימולציה שוב באמצעות הפקודה :

make | grep InputImage

Q33 : הוסף לדו"ח את כל הערכים שהוגרלו עבור InputImage.

- הפעם, ב- task body של NeuralNet\_sequence הוסף אילוץ למשפט :

- if (!ml\_pkt.randomize()) `uvm\_error("USER\_DEFINED\_FLAG", "This is a randomize error")

- שמגביל את הערכים המוגרלים למספרים זוגיים.

- שים לב שהוספת האילוץ מתבצע באופו הבא :

- if (!ml\_pkt.randomize() with { ml\_pkt.InputImage == משהו; }) .....

- הרץ את הסימולציה שוב באמצעות הפקודה :

make | grep InputImage

Q34 : הוסף לדו"ח את השינויים בקוד שרשמת. הוסף לדו"ח את כל הערכים שהוגרלו עבור InputImage. האם

המערכת הצליחה ערכים ל- InputImage שעומדים באילוצים ? מדוע לא ? אם אינך בטוח בתשובה ניתן להעזר

בקימפול מחדש אבל הפעם רק עם :

make

נטרל את האילוצים שבמחלקה NeuralNet\_transaction והרץ שוב את :

make | grep InputImage

Q35 : הוסף לדו"ח את השינויים בקוד שרשמת. האם הפעם הוגרלו ערכים שעומדים באילוצים ? הוסף לדו"ח את כל הערכים שהוגרלו עבור InputImage.

- שנה constraint זה כך הערכים המוגרלים יסתיימו ב- ff. הרץ את הסימולציה שוב באמצעות הפקודה :  
make | grep InputImage

Q36 : הוסף לדו"ח את כל הערכים שהוגרלו עבור InputImage.

- הוסף למחלקה NeuralNet\_transaction הגדרת של משתנה אקראי נוסף :  
rand logic [71:0] InputImage2;  
- הוסף ל -

- `uvm\_object\_utils\_begin

- את השורה :

- `uvm\_field\_int(InputImage2, UVM\_ALL\_ON)

- חזור למצב שיש רק אילוץ בודד שמגביל את הערכים המוגרלים ל-  
'72h01ff01ff01ff01ff01,72'hff01ff01ff01ff01ff,72'h01ffffff01ffffff01,72'hffff01ff01ff01ffff

- הוסף ל- InputImage2 אילוץ נוסף **סטטי**, זהה לקודם אבל בעל שם שונה :  
static constraint c\_InputImage2 { InputImage2 inside {72'h01ff01ff01ff01ff01 .....

- הוסף הצרה של אובייקט חדש ml\_pkt2 :

- NeuralNet\_transaction ml\_pkt, **ml\_pkt2**;

- צור אובייקט נוסף בשם ml\_pkt2 :

ml\_pkt2 = NeuralNet\_transaction::type\_id::create(.name("ml\_pkt2"),.context(get\_full\_name()));  
- הוסף משפט הגרלה ל - ml\_pkt2

- if (!ml\_pkt2.randomize() ) `uvm\_error("USER\_DEFINED\_FLAG", "This is a randomize error")

הוסף פקודות הדפסה של ImageInput ו- ImageInput2 עבור שני האובייקטים :

\$display ("ml\_pkt InputImage=%h", ml\_pkt.InputImage);

\$display ("ml\_pkt InputImage2=%h", ml\_pkt.InputImage2);

\$display ("ml\_pkt2 InputImage=%h", ml\_pkt2.InputImage);

\$display ("ml\_pkt2 InputImage2=%h", ml\_pkt2.InputImage2);

- הרץ את הסימולציה שוב באמצעות הפקודה :

make | grep InputImage

Q37 : הוסף לדו"ח את כל הערכים שהוגרלו עבור InputImage ו- ImageInput2 עבור ml\_pkt1 ו- ml\_pkt2.  
הסבר את התוצאות. האם ml\_pkt1 ו- ml\_pkt2 מקיימים את האילוצים ?

- עבור האובייקט ml\_pkt2 בלבד (לפני start\_item) נבטל את השפעת שני האילוצים בעזרת המשפטים:  
ml\_pkt2.c\_InputImage.constraint\_mode(0);  
ml\_pkt2.c\_InputImage2.constraint\_mode(0);

Q38 : הוסף לדו"ח את כל הערכים שהוגרלו עבור InputImage ו- ImageInput2 עבור ml\_pkt1 ו- ml\_pkt2.  
הסבר את התוצאות. האם ml\_pkt1 ו- ml\_pkt2 מקיימים את האילוצים ?

**הסבר תוצאות אלו למנחה!**