The Andrew and Erna Viterbi Faculty of ELECTRICAL & COMPUTER ENGINEERING

Technion Israel Institute of Technology



מעבדה ב- VLSI ספרתי - 0450111

ווריפיקציה - רקע

http://www.ee.technion.ac.il/vlsi/

הערות נא לשלוח ל-goel@ee

כל הערה תתקבל בברכה!

עדכון אחרון - 11:12 19/09/2024

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

תוכן עניינים

סביבת העבודה 3

הקדמה - Universal Verification Methodology (UVM) 3

איך להשיג תרחישים שמעולם לא חשבנו 4

מדידת הכיסוי של הבדיקות - Coverage 4

כיסוי קוד – לא מספיק 4

סיכום יתרונות הכיסוי הפונקציונלי: 4

הגדרת סביבת האימות 4

מחלקות ופאזות UVM 5

פאזות UVM 6

תיאור מבנה טיפוסי של סביבת אימות 7

סביבת העבודה

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 מאד (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 הווריפיקציה 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 ווריפיקציה 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 מאד לעלות על כל הבאגים ולשם כך, פתחו כלי ווריפיקציה ומטודולוגיות ווריפיקציה 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 הווריפיקציה.
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 מכוונת" (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 יתגלה. 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 מגדיר 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 מיושמת 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 שעון 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 צורכות משאבים יקרים.

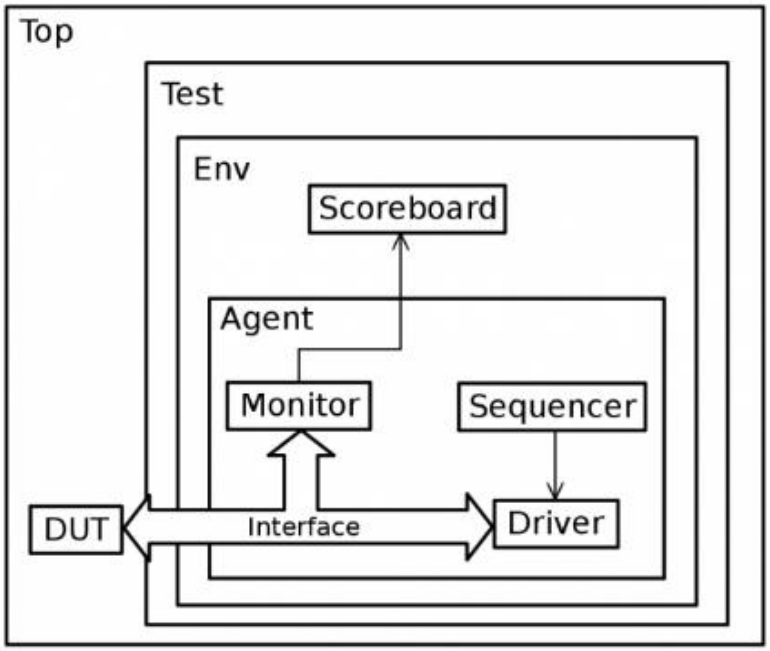
הגדרת סביבת האימות

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 ש"מקשיב" 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. מטודולוגית 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 בפרוטוקל- SPI לתקשורת. אם, נרצה לבדוק DUT דומה אבל עם רוטוקלI2C במקום SPI, כל מה שצריך לעשות הוא להוסיף driver ו- monitor עבור I2C במקום אלה של ה- SPI הקיים. יתר הבלוקים יישארו ללא שינוי.

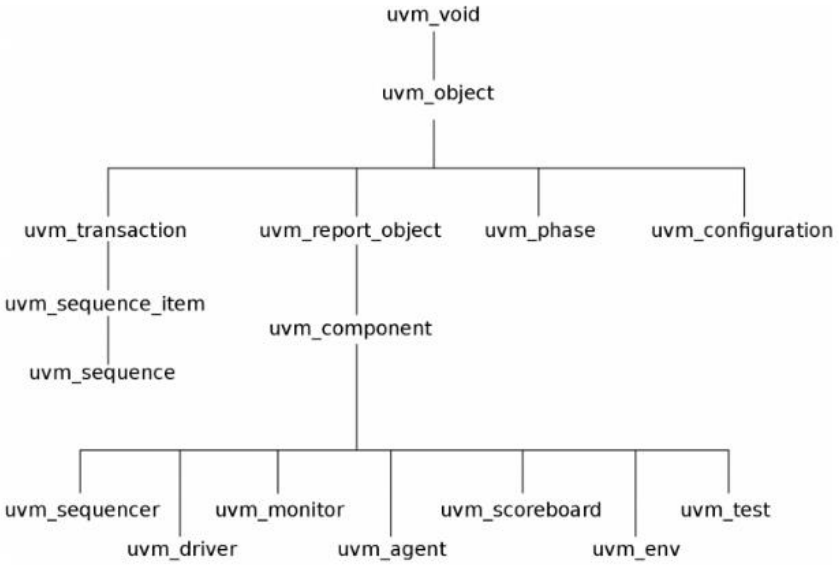
כעת נכיר מספר מושגים של עולם ה- UVM :

- מחלקות ופאזות UVM
- תיאור מבנה טיפוסי של סביבת אימות

חשוב : מפאת חוסר זמן, לא מוסברים כל עקרונות ה- UVM. נתרכז רק בחלקים שחשובים לניסוי זה.

מחלקות ופאזות UVM

ב- UVM (בדומה ל- Object Oriented Programming), כל הבלוקים המוזכרים מיוצגים מחלקות (classes) הנגזרים ממחלקות UVM קיימות והבך יורשים את התכונות שלהן. עץ המחלקות של מחלקות UVM החשובים ביותר מופיע באיור מס' 2.

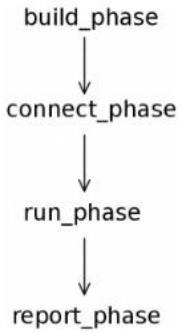


איור מס' 2 : עץ המחלקות של מחלקות UVM

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 יש פאזות של סימולציה. הפאזות הן שלבי ביצוע מסודרים שממומשים 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:

- פאזת הבנייה, build phase, משמש לבניית המרכיבים שלו בהיררכיה. לדוגמא, בשלב הבנייה של מחלקת ה- agent יבנו ה- sequencer, ה- driver וה- monitor.
- פאזת ה- connect phase תשמש לחיבור תתי הבלוקים השונים של המחלקה. בדוגמא זאת, פאזת ה- connect של ה- agent יחבר את ה-driver ל- sequencer ויחבר את ה- monitor ליציאה חיצונית.
- פאזת ה- run phase הוא השלב העיקרי של הביצוע, זה המקום שבו נמצא בפועל קוד הסימולציה שיבוצע.
- בסוף, פאזת ה- report היא הפאזה שבה מוצגות תוצאות הסימולציה.

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

תיאור מבנה טיפוסי של סביבת אימות

המערכת בנויה בצורה היררכית. במקרה הזה נבחר עבור ההיררכיה הגבוהה את השם Top . ניתן לראות באיור מס' 4 שהסביבה מורכבת מהיחידות הבאות :

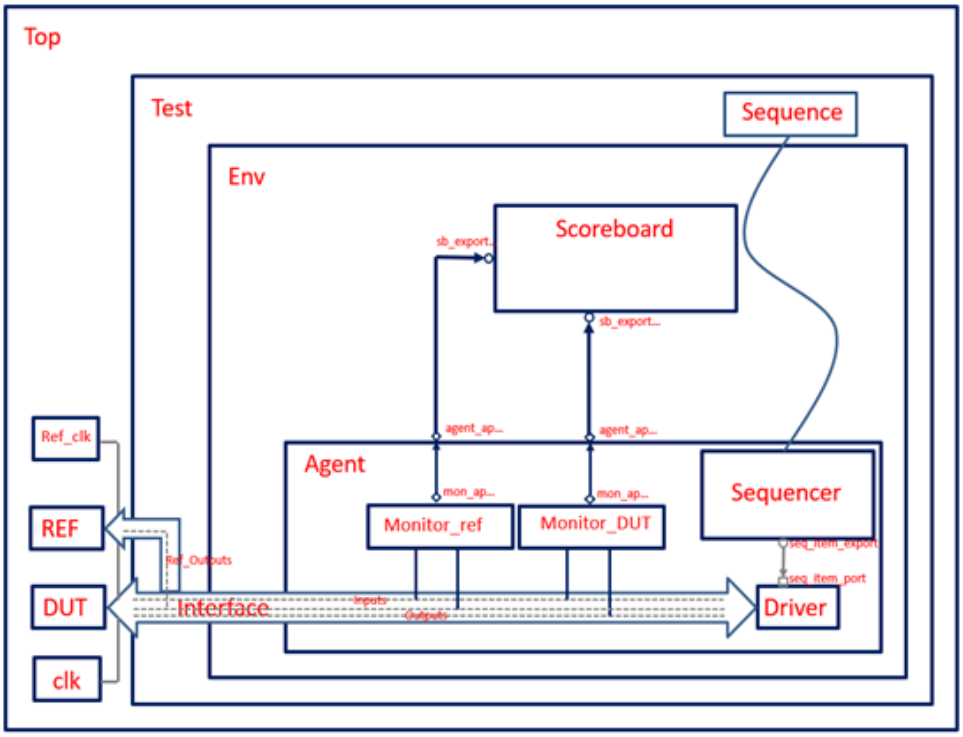
- הבלוק העליון
- הגדרת העירור ל- DUT – Transactions, Sequences and Sequencers
- מחלקת ה- Driver
- מחלקת ה- Monitor
- מחלקת הסוכן Agent
- מחלקת ה- Environment - env

מבנה הבלוק העליון :

בלוק העליון יהיה מודול SystemVerilog רגיל והוא יכיל:

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

באיור 4 ניתן לראות סכמה של סביבת האימות. בהמשך יובאו הסברים על חלק מהבלוקים.



איור מס' 4 : המרכיבים של Testbench_top

Interface

הממשק (interface) הוא מודול המכיל את כל הכניסות והיציאות של ה- DUT. ה- monitor, ה- driver וה- DUT מתחברים למודול זה. דוגמא של קוד עבור הממשק:

```
interface NeuralNet_if;
  logic clk, rst, Learn, Classify;
  logic [31:0] InData;
  logic [7:0] result;
endinterface : NeuralNet_if
```

קוד מס' 1: דוגמא של קוד להגדרת ממשק ל- NeuralNet

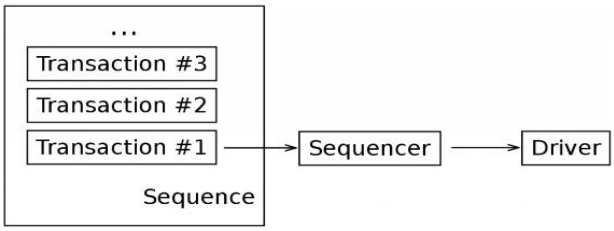
הגדרת העירור ל- DUT – Transactions, Sequences and Sequencers

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 שמכילה את המידע הדרוש כדי לאפשר תקשורת בין שני רכיבים או יותר. הם יכולים לכלול משתנים, אילוצים ואפילו מתודות הפועלות על הנתונים של עצמם. בשל רמת ההפשטה 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 נה- 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 שולחת אותם ל- driver. תיאור פעולה זו מוצג באיור 5.



איור מס' 5: 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- NeuralNet יש להגדיר transaction פשוט אשר יכלול את המשתנים הבאים:

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

מילת המפתח rand מציין שהמשתנה x יהיה משתנה אקראי שיוזן לכניסה של ה- DUT. משתנה result ידגום את יציאת ה- DUT.

רצף או Sequence

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

```
1.  class NeuralNet_sequence extends uvm_sequence#(NeuralNet_transaction);

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

5.  `uvm_object_utils(NeuralNet_sequence)

6.  task body();
7.  NeuralNet_transaction ml_pkt;
8.  `uvm_info("", "Randomize Try0", UVM_MEDIUM)
9.  ml_pkt = NeuralNet_transaction::type_id::create(.name("ml_pkt"), .contxt(get_full_name()));

10. repeat(6) begin
11. //`uvm_info("ml_pkt_sequence", ml_pkt.sprint(), UVM_LOW);
12. start_item(ml_pkt);
13. if (!ml_pkt.randomize() with { ml_pkt.InputImage inside
    {72'h01ff01ff01ff01ff01,72'hff01ff01ff01ff01ff,72'h01ffffff01ffffff01,72'hffff01ff01ff01ffff} ; })
    `uvm_error("USER_DEFINED_FLAG", "This is a randomize error")

14. finish_item(ml_pkt);
15. end
16. endtask: body
17. endclass: NeuralNet_sequence
```

קוד מס' 2 : קוד עבור ה- sequence

להלן הסבר על הקוד:

שורה 5 מאקרו UVM הרושם את ה- sequence במבנה נתונים בשם - factory.

שורה 6 הגדרת ה- task בשם ()body שמהווה את ה- task העיקרי של ה- sequence. ה- ()body מופעל כאשר ה- test קורא ל- start של ה- sequence (ראה בהסבר של test בהמשך).

שורה 7 הצהרה על ml_pkt כ- transaction מסוג NeuralNet_transaction

שורה 9 יצירת אובייקט מסוג transaction

שורה 10 תחילה של לולאה כדי ליצור ששה transactions

שורה 12 היא פקודה שמונעת התקדמות עד שה- driver מקבל את ה- transaction שנוצר

שורה 13 מפעילה את מילת המפתח rand של ה- transaction ומרנדמת את המשתנים של ה- transaction שיישלח ל- driver.

שורה 14 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#(NeuralNet_transaction) NeuralNet_sequencer;

הקוד הנ"ל יוצר טיפוס מסוג NeuralNet_sequencer שהוא למעשה uvm_sequencer שנבנה על סמך הפרמטר NeuralNet_transaction. ה- sequencer מוצב ב- agent.

נסביר כעת כיצד sequence מתחבר - sequencer , וכיצד sequencer מתחבר ל- driver ?

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 יכלול 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 NeuralNet_driver extends uvm_driver#(NeuralNet_transaction);
2.  `uvm_component_utils(NeuralNet_driver)

3.  virtual NeuralNet_if vif;
4.  virtual NeuralNet_Ref_if vrefif;

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

8.  function void build_phase(uvm_phase phase);
9.  super.build_phase(phase);
10.     void'(uvm_resource_db#(virtual NeuralNet_if)::read_by_name (.scope("ifs"),
        .name("NeuralNet_if"), .val(vif)));
11. endfunction: build_phase

12. task run_phase(uvm_phase phase);
13. drive();
14. endtask: run_phase
```

קוד מס' 3 : קוד עבור ה- driver

הסבר הקוד :

שורה 1 יוצרת מחלקה בשם NeuralNet_driver מ- uvm_driver. (NeuralNet_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 5 עד 7 הן ה- constructor של המחלקה.

בשורה 8 מתחילה פאזת הבנייה של המחלקה. פאזה זאת מבוצעת לפני פאזת ה- run.

שורה 10 מקבלת את הממשק ממסד הנתונים של ה- factory. זהו ממשק וירטואלי והוא למעשה מצביע לאותו ממשק שהוצב קודם בבלוק העליון.

בשורה 12 מתחילה פאזת ה- run, שבו יבוצע קוד ה- drive (שורה 13 : ראה בהמשך).

כעת נעבור להסבר על פאזת 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 drive() :
חשוב לציין שקוד זה הוא ההמשך של הקוד הקודם " קוד מס' 3 : קוד עבור ה- driver"

```
virtual task drive();
      NeuralNet_transaction ml_pkt;
      integer state = 0;
      integer count = 0;
      vif.Learn = 1'b0;
      vif.Classify = 1'b0;
      `uvm_info("", "Drive Function of Driver", UVM_MEDIUM)
      forever begin
         `uvm_info("", "Forever of Driver", UVM_LOW)

     @(negedge vif.clk)
   begin

           case(state)
              0: begin   //reset
                 vif.rst = 1'b1;
                 vrefif.rst = 1'b1;
                 state = 1;
              end
              1: begin   //Learn 10 cycles
                  vif.rst = 1'b0;
                  vrefif.rst = 1'b0;
                  vif.Learn = 1'b1;
                  vif.Classify = 1'b0;
                  count = count + 1;
                  if( count < 10)
                        begin
                          state = 1;
                        end
                        else begin
                          state = 2;
                        end
                  end
              2: begin   //Classify 1
                  vif.Learn = 1'b0;
                  vif.Classify = 1'b1;
                  seq_item_port.get_next_item(ml_pkt);
                  state = 3;
                  vif.InData =
   {ml_pkt.InputImage[71:64],ml_pkt.InputImage[63:56],ml_pkt.InputImage[47:40],ml_pkt.InputImage[39:32]};
                  vrefif.InputImage = ml_pkt.InputImage[71:0];
                end
```

```
                3: begin //Classify 2
                    state = 4;
                    vif.InData =
    {ml_pkt.InputImage[63:56],ml_pkt.InputImage[55:48],ml_pkt.InputImage[39:32],ml_pkt.InputImage[31:24]};
                  end
                4: begin //Classify 3
                    state = 5;
                    vif.InData =
    {ml_pkt.InputImage[47:40],ml_pkt.InputImage[39:32],ml_pkt.InputImage[23:16],ml_pkt.InputImage[15:8]};
                  end
                5: begin //Classify 4
                    vif.InData =
    {ml_pkt.InputImage[39:32],ml_pkt.InputImage[31:24],ml_pkt.InputImage[15:8],ml_pkt.InputImage[7:0]};
                    state = 2;
                    seq_item_port.item_done();
                  end

            endcase
          end
        end
endtask: drive
endclass:NeuralNet_driver
```

קוד מס' 4 : קוד עבור מתודת ה- drive

יש לשים לב שהגישה לכניסות והיציאות של ה- DUT היא באמצעות הממשקים הוירטואליים vif ו- vrefif.
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.

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 התחברות ל- DUT – vif.

לרוב התהליך המקובל בתוכנית אימות היא: דגימת הכניסות, חיזוי של התוצאה הצפויה והשוואת התוצאות עם התוצאות של ה- DUT.

מחלקת הסוכן Agent

השלב הבא הוא לחבר את כל היחידות שהגדרנו עד כה. זוהי העבודה של ה- agent. ל- agent אין צורך בפאזת ה- run, אין קוד סימולציה שיש לבצע בבלוק זה. לעומת זאת כן יש צורך בפאזה של connect ופאזה של build.

מחלקת ה- env

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

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.

מבנה ה- Testbench_top

ראשית, בבלוק העליון נבצע רישום הממשק ב- UVM factory. זה הכרחי כדי לאפשר חיבור הממשק לכל שאר המחלקות שמופיעות בסביבת האימות. הרישום יתבצע בפועל ב- UVM factory באמצעות מסד נתונים מסוג uvm_resource_db. דוגמא של מימוש של הבלוק העליון מיוצג בקוד 5.

```
1.  module NeuralNet_tb_top;
2.  import uvm_pkg::*;                          // import the UVM Library

3.  //Interface declaration
4.  NeuralNet_if vif();

5.  // Instantiate DUT and connect its ports
6.  NeuralNet dut(.clk(vif.clk), .rst(vif.rst), .Learn(vif.Learn), .Classify(vif.Classify), .InData(vif.InData),
    .result(vif.result));

7.  initial begin
8.  //Registers the Interface in the configuration block so that other blocks can use it
9.  uvm_resource_db#(virtual NeuralNet_if)::set (.scope("ifs"), .name("NeuralNet_if"), .val(vif));

10. //Executes the test
11. run_test("NeuralNet_test");
12. #320 $finish;
13. end

14. // Provide the values of H0,H1, WA and WB
15. initial
16. begin
17. repeat(2) @(negedge vif.clk);
18.         vif.InData = 32'hff0101ff ; //WB of X
19.         @(negedge vif.clk);
20.         vif.InData = 32'h01ffff01 ; //WB of 0
21.         @(negedge vif.clk);
22.         vif.InData = 32'hff0101ff ; //WB of /
23.         @(negedge vif.clk);
24.         vif.InData = 32'hffffffff ; //WB of \ @(negedge vif.clk);
25.         @(negedge vif.clk);
26.         vif.InData = 32'h01ffff01 ; //WA of X
27.         @(negedge vif.clk);
28.         vif.InData = 32'hff0101ff ; //WA of 0
29.         @(negedge vif.clk);
30.         vif.InData = 32'hffffffff ; //WA of /
31.         @(negedge vif.clk);
32.         vif.InData = 32'h01ffff01 ; //WA of \
33.         @(negedge vif.clk);
34.         vif.InData = 32'h01ffff01 ; //right filter looks for \
35.         @(negedge vif.clk);
```

```
36.         vif.InData = 32'hff0101ff ;  //left filter looks for /
37.  end


38.  //Variable initialization
39.  initial
40.  vif.clk <= 1'b0;

41.  //Clock generation
42.  always
43.  #5 vif.clk = ~vif.clk;
44.  endmodule
```

קוד מס' 5: דוגמא של קוד עבור NeuralNet_test_tb.sv

להלן הסבר קצר על הקוד:
משפט ה- import מייבא את ספריית ה- UVM.
שורות 4 ו- 6 הן הצבת ממשק ה- DUT ואת ה- DUT וחיבור שלהם לתוך הבלוק העליון.
בשורה 9 ה- uvm_resource_db רושמת את הממשק ב- UVM factory השם hamming_if.
בשורה 11 מופעלת ה- run_phase של NeuralNet_test בזמן ריצה.
בשורות 15-35 מספקים את הערכים של המשקלים שהם כידוע קבועים.
שורה 43 מייצרת את השעון עם מחזור של 10 timeunits. ה- timeunit מוגדר גם Makefile.

הערות חשובות

להלן מספר הערות חשובות שבאות להוסיף מידע ולהביר נקודות מסויימות :

- ה- run_phase חייב להחיות 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.
- run_test יוצר אובייקט בשם umv_test_top ומבצע את שיגרת ה- run_phase שלו.
- פונקצית finish_on_completion קוראת ל- $finish אחרי run_test על מנת לסיים את ה- test.

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