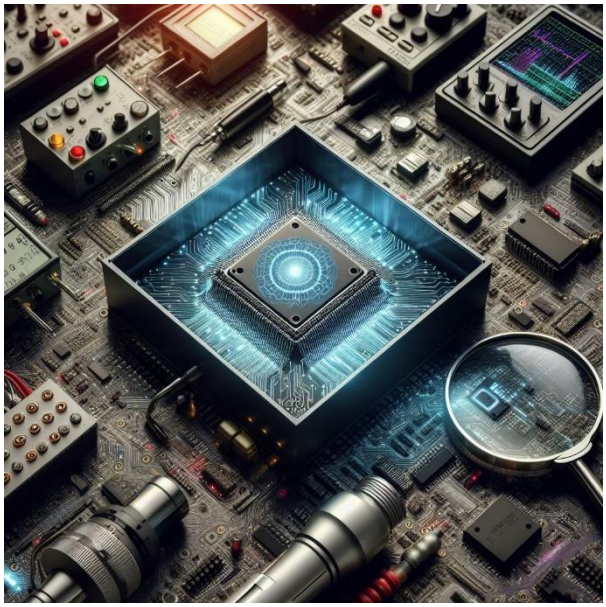
מכון טכנולוגי חולון
Holon Institute of Technology

הפקולטה להנדסת חשמל ואלקטרוניקה
שפת תכנון חומרה וורילוג

UVM Verification



מגישים:

אורי כהן 209044080
בר אליס 208545236
שון פזרקר 208426890
חיים עוזר 316063569

מרצה: ד"ר אביחי אהרון

תוכן עניינים

מבוא ל UVM Verification

UVM זה ראשי תיבות של Universal Verification Methodology, היא שיטה סטנדרטית בתעשיית המוליכים למחצה לורפיקציה של digital designs, במיוחד מעגלים משולבים (IC) ומערכות-על-שבב (SoCs). הוא בנוי על גבי שפת SystemVerilog שהיא שפת תיאור וורפיקציה של החומרה וניתן לשלב רכיבים רבים בקלות בתהליך design verification.

UVM כולל קבוצה של הנחיות ושיטות עבודה מומלצות לפיתוח test bench, הפעלת סימולציות וניתוח תוצאות. השימוש בUVM הפך לסטנדרט לורפיקציה העיצוב , ועוזר ל Chip Design ומהנדסי ולדיציה וורפיקציה (V&V) להבטיח את הפונקציונליות של הרכיבים שעיצבו.

UVM משתמש ב-TLM (Transaction-Level Modeling) לתקשורת בין רכיבים שונים ב test bench, מה שמקל על העברת נתונים ובקרת המידע.

UVM חיונית בתעשיית המוליכים למחצה מכמה סיבות:

1. Complexity : התקני מוליכים למחצה מודרניים הם מורכבים ביותר, מה שהופך את הורפיקציה לחלק קריטי בתהליך התכנון כדי להבטיח פונקציונליות וביצועים.
2. Efficiency : המערכת מייעלת את תהליך הורפיקציה, מה שהופך אותו ליעיל ושיטתי יותר בניגוד לתהליכי בדיקות אחרות, דבר חיוני לעמידה בלוחות הזמנים של שחרור המוצר.
3. Reusability : הUVM מקדם יצירה של רכיבים הניתנים לשימוש חוזר, שניתן למנף אותם על פני מספר פרויקטים, תוך חיסכון בזמן ומשאבים.
4. סטנדרטיזציה : כסטנדרט בתעשייה, UVM מספקת מערכת של שיטות עבודה משותפות, ומאפשרת שיתופי פעולה בין תאגידים וחברות שונות במשק העבודה.

OVM היא Open Verification Methodology מתודולוגיה ששימשה ליצירת סביבות ורפיקציה מובנות לשימוש חוזר בdesign verification כאשר OVM סיפקה בסיס ליצירת איסוף נתונים ובקרת תהליכי ורפיקציה, תוך תמיכה ב-SystemVerilog ו SystemC.

UVM נבנה על בסיס שילוב שיטות עבודה ותכונות מומלצות מ-OVM ומתודולוגיות אחרות, מה שמוביל למסגרת חזקה שמשפרת שימוש חוזר, יעילות וסטנדרטיזציה בורפיקציה של עיצובים דיגיטליים מורכבים.

עקרונות תכנות מונחה עצמים ב UVM Verfication

תכנות מונחה עצמים (OOP) ממלאים תפקיד חשוב ב UVM Verification.

תכנות מונחה עצמים היא מערכת תכנות המשתמשת ב"אובייקטים" לעיצוב יישומים ותוכנות מחשב. היא מאפשרת עטיפה של נתונים ופונקציות הפועלות על הנתונים, מה שמאפשר שימוש חוזר בקוד ומודולריות.

עקרונות המפתח:

1. מחלקות: מאפשר יצירת אוסף נתונים יחד עם ההתדרייברויות הקשורות אליהם. המשמעות היא שהקוד שפועל על הנתונים נשמר יחד, מה שמקל על הניהול וההבנה של המערכת.

   - דוגמה: מחלקה של Animal מכילה תכונות כמו גזע וגיל.

2. ירושה: ירושה היא תכונה מרכזית של OOP המאפשרת שימוש חוזר בקוד. בהקשר של UVM, הורשה מאפשרת יצירת רכיבי ורפיקציה (verification components) מיוחדים היורשים מאפיינים ושיטות משאר הרכיבים. זה מקל על יצירת סביבות בדיקה מורכבות.

   - דוגמה: המחלקה Dog יורשת מהמחלקה Animal פונקציות ייחודיות לתת מחלקה כמו bark ו fetch .

3. פולימורפיזם: פולימורפיזם מאפשר לאותו פיסת קוד להתנהג בצורה שונה בהתבסס על סוג האובייקט איתו הוא מתמודד. זה שימושי במיוחד ב-UVM מכיוון שהוא מאפשר קוד verification גמיש וניתן יותר להתאמה.

   - דוגמה: למחלקה של Cat ומחלקה של Dog עשויות להיות שניהם שיטת makeSound() אבל הפלט שונה ("meow" לעומת "woof").

נניח אנו מתכננים מערכת רכב. ב-UVM, היינו יוצרים מחלקה של `Car` שכוללת את כל המאפיינים של מכונית, כמו speed, fuelLevel ו-engineStatus, פונקציות כמו accelerate, brake, ו refuel.

עבשיו, נניח שיש לנו סוגים שונים של מכוניות, כמו 'SUV', 'Sedan', ו'Sports Car'. כל אחד מאלה יכול להיות מחלקה נפרדת שיורשת מהמחלקה - `Car`, אך יש להם מאפיינים נוספים או שיטות ספציפיות להם. לדוגמה, למחלקת `SportsCar` עשויה להיות פונקציה - turboBoost נוספת.

ב-UVM, היינו יוצרים Test Bench עבור המופעים של מחלקת מכוניות אלה. בTest Bench קיימת אינטראקציה עם המקרים הללו, קורא לשיטות שלהם ובודק את המאפיינים שלהם כדי לוודא שמערכת המכונית מתדרייברת כצפוי בתנאים שונים.

סטנדרטיזציה של UVM

UVM התקבל כמסגרת סטנדרטית לבדיקות בפברואר 2011 ע"י Accellera Systems בשיתוף עם מובילים בתחום ה-SoC וה-EDA (Electronic Design Automation), מאוחר יותר, הוא הוגש ל-IEEE לתקינה ופורסם ב-IEEE 1800.2 ב-14 בספטמבר 2020.

החשיבות של UVM טמונה בסטנדרטיזציה שלו בכל התעשייה.
ספקי EDA - EDA היא קטגוריה של כלי תוכנה המשמשים לתכנון מערכות אלקטרוניות כגון מעגלים מודפסים ומעגלים משולבים. כלים אלה משמשים לתכנון, הדמיה וייצור מערכות ומעגלים אלקטרוניים גדולים וספקי IP - בתכנון אלקטרוני, IP מתייחס ליחידה לשימוש חוזר של עיצוב לוגיקה, תא או שבב.IP אלו הן אבני בניין בתוך עיצובי שבבים שניתן לעשות בהם שימוש חוזר ביעילות לעיצוב שבבים חדשים.
כלי תוכנה אלו תומכים ב-UVM, מה שמביא לתמיכה רחבה בכלים שונים וכתובות IP לורפיקציה (VIPs).
סטנדרטיזציה זו מאפשרת יכולת פעולה הדדית ושימוש חוזר. מהנדסים יכולים לשלב בקלות VIP במערכת ורפיקציה הודות לממשק הסטנדרטי. למרות שהפונקציונלית עשויה להיות שונה, אופי ה-plug-and-play נשאר עקבי כל עוד מקפידים על התקן.

UVM שינתה לחלוטין את תהליך הורפיקציה ב- digital design. לפני ה-UVM, לוריפיקציות בחברות שונות היו קווי דמיון אבל היה צורך לקבלה אוניברסלית יותר ולשימוש חוזר. פיצול הבדיקות היה חיסרון משמעותי לבדיקות בעיצוב רכיבים אלקטרוניים.

UVM מקדם שימוש חוזר על ידי מתן מתודולוגיה סטנדרטית ליצירת רכיבי ורפיקציה מודולריים הניתנים להגדרה. גישה מודולרית זו מאפשרת למהנדסים לפתח testbench באמצעות אבני בניין הניתנות לשימוש חוזר, הפחתת יתירות וחיסכון בזמן.

להלן כמה מאפיינים נוספים לסטנדרט UVM:

1. אוטומציה: UVM כולל תכונות עוצמתיות לאוטומציה של משימות ורפיקציה נפוצות. לדוגמה, מנגנון רצף UVM מאפשר להגדיר תרחישי גירוי מורכבים ולעשות בהם שימוש חוזר. מנגנון לוח התוצאות של UVM מספק בדיקה אוטומטית של גירוי העיצוב.
2. ורפיקציה coverage-driven: UVM תומך במתודולוגיית ורפיקציה coverage-driven. המשמעות היא שהתקדמות מאמץ הורפיקציה נמדדת על סמך מדדי coverage, כגון coverage קוד או coverage פונקציונלי. זה עוזר להבטיח שכל ההיבטים החשובים של העיצוב נבדקו ביסודיות.
3. איתור באגים: UVM מספק מנגנון רישום סטנדרטי המקל על איתור באגים בtestbench.

לסיכום UVM הוא חיוני מכיוון שהוא מספק גישה סטנדרטית ויעילה לורפיקציה נכונותם של תכנוני חומרה מורכבים בתעשיית המוליכים למחצה. זה מעודד שימוש חוזר ברכיבי ורפיקציה ו testbench, מה שהופך את הורפיקציה למהיר ואמין יותר.

דוגמאות לשימוש UVM בתעשייה

בפועל, UVM נמצא בשימוש נרחב בתכנון וורפיקציה מוליכים למחצה על יעילותה בשיפור פרודוקטיביות ורפיקציה, שימוש חוזר וסקיילאביליות. דוגמאות מהעולם האמיתי ומחקרים אקדמים מדגישים כיצד UVM יושם בהצלחה בפרויקטים שונים כגון:

1. ורפיקציה מעבד: חברת intel השתמשה ב-UVM בורפיקציה מעבדי Intel Core - מעבדים מרובי ליבות על ידי יצירת רכיבי ורפיקציה UVM הניתנים לשימוש חוזר כמו מנהלי התקנים, צגים ולוחות תוצאות, הם השיגו ורפיקציה יעיל של פונקציונליות מעבדים שונים, כולל פרוטוקולי קוהרנטיות מטמון וורפיקציה לארכיטקטורת ערכות הוראות.

2. ורפיקציית ממשק במהירות גבוהה: חברת NVIDIA השתמשה ב-UVM בורפיקציה עיצובי הממשק המהיר שלה, כגון אלו המשמשים ביחידות עיבוד גרפיות (GPU) ופתרונות רשת של מרכזי נתונים, מה שמבטיח ורפיקציה לפרוטוקול ונכונות תפקודית.

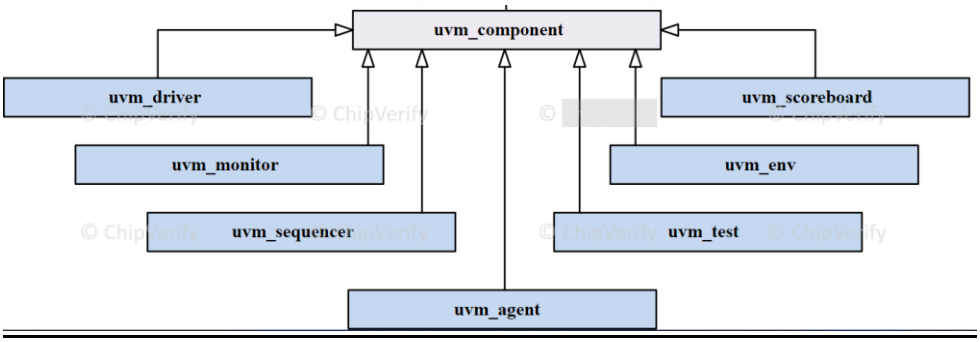
3. ורפיקציה סיגנלים מעורבים: חברת Analog Devices מיישמת UVM בורפיקציה של מעגלים משולבים של סיגנלים מעורבים, כולל ממירים אנלוגיים לדיגיטליים (ADC) ולוגיקת בקרה דיגיטלית, תוך מינוף רכיבי ורפיקציה הניתנים לשימוש חוזר לצורך ורפיקציה מקיף.

4. ורפיקציה ניהול צריכת חשמל: חברת Qualcomm השתמשה ב-UVM בורפיקציה של תכונות ניהול צריכת חשמל עבור עיצובי SoC שלה, תוך שילוב UVM עם טכניקות סימולציה מודעת לצריכת חשמל לורפיקציה מצבי הספק דינמיים ומעברי תחום הספק.

5. ורפיקציה אב טיפוס FPGA: חברת Xilinx השתמשה ב-UVM בורפיקציה של תכנונים מבוססי FPGA ויצרה סביבת ורפיקציה מקיפה עם יצירת גירוי אקראי מוגבל, coverage-driven פונקציונלי ובדיקות רגרסיה, בכך היא האיצה את תהליך הורפיקציה של התקני הלוגיקה הניתנים לתכנות שלהם לפני הייצור.

דוגמאות אלו מדגימות כיצד UVM מיושם בתחומי ורפיקציה מגוונים, תוך מינוף תכונותיו כגון מתודולוגיה מונחית עצמים, בדיקות אקראיות מוגבלות, transaction-level modeling וcoverage-driven verification כדי להשיג V&V מקיפים של עיצובים מורכבים של מוליכים למחצה.

רכיבים ב UVM



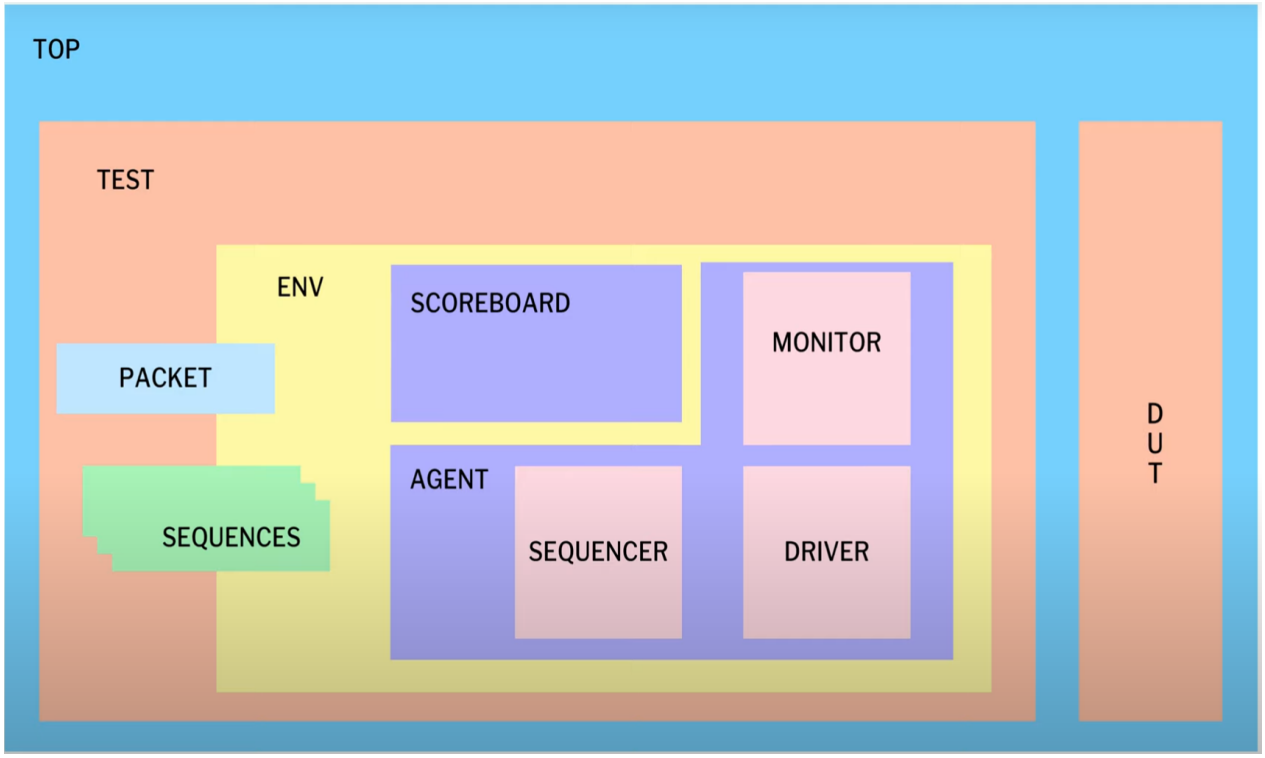
1. **Test** רכיב ה- Test הוא הרכיב Top-Level השולט בביצוע הבדיקה. הוא מציג את הסביבה, הרצפים ורכיבים אחרים, ומגדיר את ה Test Flow. רכיב הבדיקה גם מגדיר את התצורה של הבדיקות ומתחיל את הסימולציה.
2. **Environment** מכיל רכיבי ורפיקציה מרובים לשימוש חוזר. זה מגדיר את תצורת ברירת המחדל שלהם ואת האופן שבו הם מקיימים אינטראקציה זה עם זה.
3. **Agent** מכיל את ה Sequencer, Monitor והDriver. הAgent יכול להיות אקטיבי (מקיים אינטראקציה עם הDesign) או פסיבי (דוגם סיגנלים מה DUT ע"י תרגום סיגנלים לטרנזקציות).
4. **Sequencer** רכיב אקטיבי שולט בזרימת הטרנזקציות בין רצפים לדרייבר. הוא מייצר ושולח טרנזקציות לדרייבר.
5. **Driver** רכיב אקטיבי שעובד עם ה interface, בסוף מתרגם טרנזקציות לסיגנלים.
6. **Monitor** המוניטור לוקח סיגנלים מה DUT באמצעות ה- interface ומתרגם אותם לפורמט של פאקטות או Sequence Items. הפאקטות מועברות לרכיבי UVM אחרים כמו ל Sequencer או ל Scoreboard.
7. **Scoreboard** רכיב ורפיקציה שבודק את הפונקציונליות של הDesign. הוא מקבל טרנזקציות מהמוניטור ובודק אם הפלט תואם לתוצאה הצפויה.

Interface מתייחס לחיבור סטנדרטי או פרוטוקול תקשורת בין רכיבים או מודולים שונים בתוך מערכת.

אובייקטים בUVM

Squences אובייקטי רצף (Sequence) ב-UVM מייצגים רצף של עסקאות (Transactions) או אירועים המוחלים על ה-DUT או נבדקים ב Environment. הם שולטים ביצירת גירוי ובטיפול בתגובה, ומאפשרים תרחישי ורפיקציה מובנים וניתנים לשימוש חוזר.

Packet אובייקטים מסוג Packet, הידועים גם בשם Sequence Items, מקיפים עסקאות בודדות או פאקטות של נתונים בתוך רצף. הם מכילים מידע כגון שדות נתונים, אותות בקרה ותכונות תזמון הרלוונטיות לעסקה שנוצרת או נבדקת. הפאקטות מנוהלות על ידי ה- Sequencer בTestBench של UVM.



הקשר בין Verilog ל UVM Verification

הקוד הבא רשום בשפת Verilog והקוד מתאר Test Bench של רכיב מסוים.
ניתן להציג את הקוד ב Verilog איך הוא יראה לפי UVM Verification עם כל הרכיבים והאובייקטים שלו.

```
module verilog_testbench();

    // Inputs
    reg a;
    reg b;
    reg c;
    reg d;
    reg clock;

    // Outputs
    wire out;

    // Instantiate the Unit Under Test (UUT)
    verilog_module uut( .a(a), .b(b), .c(c), .d(d), .out(out), .clock(clock));

    always #1 clock = ~clock;

    initial begin

        clock = 1'b0;
        #10;
        a = 1'b1;
        b = 1'b0;
        c = 1'b1;
        d = 1'b0;
        #10;

        repeat(100)
        begin
            a = $random;
            b = $random;
            c = $random;
            d = $random;
            #5;
        end

        #30;
        $finish;

    end
endmodule
```

TOP

YOUR DESIGN

TEST

DRIVER

SEQUENCE

PACKET

<u>דוגמה לשימוש בכל קומפוננטה:</u>

בדוגמה נשתמש במונה פשוט שעושה increment בכל מחזור שעון ויש אפשרות לאפס אותו עם אות איפוס- CLR.

1. **Driver** הדרייבר ישלוט באות האיפוס. לדוגמה, זה יכול לייצר טריגר לאות האיפוס בתחילת הבדיקה כדי לאפס את המונה.
2. **Monitor** המוניטור יהיה צופה בפלט של המונה בכל מחזור שעון. כאשר המונה גדל או מתאפס, המוניטור יוצר טרנזקציה ושולח אותה לרכיבים אחרים.
3. **Sequencer** הסיקונסר ייצור טרנזקציות כדי לשלוט באות האיפוס. לדוגמה, זה יכול ליצור טרנזקציה כדי לקבוע את אות האיפוס בתחילת בדיקה.
4. **Scoreboard** הסקור בוארד יקבל טרנזקציות מהמוניטור ויבדוק שהמונה גדל בצורה נכונה. לדוגמה, זה יכול לבדוק שהערך של המונה שווה למספר מחזורי השעון שחלפו מאז האיפוס האחרון.
5. **Agent** האג'נט יכיל את הדרייבר, המוניטור והסקונסר. הוא יחבר את הסיקונסר לדריבר כך שהדריבר יוכל לקבל טרנזקציות מהסיקוונסר, והוא גם יחבר את המוניטור לScore board או Coverage Collector כך שרכיבים אלו יוכלו לקבל טרנזקציות מהמוניטור.
6. **Test** הבדיקה יכולה לציין את גירוי אות האיפוס. לדוגמה, זה יכול לציין כי יש להצהיר על אות האיפוס בתחילת הבדיקה ולאחר מכן לבטל לאחר מספר מסוים של מחזורי שעון.
7. **Environment** הסביבה תכיל את ה Agent וכל רכיב נוסף כמו ה Score board. זה יציין את החיבורים בין רכיבים אלה והוא יכול גם לציין ערכי ברירת מחדל.

מחזור החיים של רכיב UVM

רכיב UVM מגדיר קבוצה של שלבים שעוזרים לארגן את תהליך הורפיקציה. להלן השלבים העיקריים ב-UVM:

א. שלבי זמן בנייה:

build phase: במהלך שלב זה, רכיבי ה testbench נבנים והשלבים שלהם נוצרים. זה השלב הראשוני שבו אובייקטים של ה testbench מופקים.

connect phase: בשלב זה, רכיבי testbench שונים מחוברים באמצעות יציאות TLM (מודלים ברמת טרנזקציה). זה מבטיח שכל הרכיבים מוכנים לאינטראקציות נוספות.

end of elaboration phase: לאחר חיבור הרכיבים, שלב זה מטפל במשימות נוספות הנדרשות עבור ה testbench, כגון הצגת טופולוגיית ה-UVM.

start of simulation phase: שלב זה מגדיר תצורות זמן ריצה ראשוניות או מציג את הטופולוגיה.

ב. שלבי זמן ריצה:

run phase: הסימולציה בפועל מתרחשת במהלך שלב זה. גירוי מבחן מונע לעיצוב (design) והביצועים מתפתחים.

במקביל לrun phase פועלים: pre_reset, reset, post_reset, pre_configure, configure post_configure, pre_main, main, post_main, pre_shutdown, shutdown ו-post_shutdown.

שלב ה-pre_reset משמש לשילוב כל הפעילויות או הפונקציונליות הדרושים לפני שאות האיפוס הופך לפעיל, המדמה גירוי הפעלה. לאחר מכן, שלב האיפוס מייצר אות איפוס כדי לאתחל את הממשק ולהגדיר אותו למצב ברירת המחדל שלו. שלב post_reset מאפשר לבצע כל פעילות מיידית לאחר האיפוס.

לאחר השלמת רצף האיפוס, שלב ה-pre_configure מכין את ה-DUT לתכנות תצורה, ומספק הזדמנות אחרונה לעדכן מידע לפני שהוא יוחל על ה-DUT. שלב ההגדרה מתכנת את ה-DUT ואת כל הזכרונות המשויכים, ומבטיח מוכנות לביצוע מקרה בדיקה. שלב post_configure ממתין לתגובה לאחר קביעת התצורה של ה-DUT או למצב DUT ספציפי כדי להתחיל את גירוי הבדיקה הראשי.

שלב ה-pre_main מבטיח שכל הרכיבים הדרושים מוכנים ליצור גירוי, בעוד שהשלב הראשי מחיל את הגירוי שנוצר על ה-DUT, המנוהל בדרך כלל באמצעות רצפים. שלב post_main מטפל בכל ההשלמות הנדרשות לאחר שלב הגירוי הראשי.

שלב ה-pre_shutdown משמש כחיץ לגירויים הזקוקים לתשומת לב לפני שלב הכיבוי. שלב הכיבוי מבטיח שההשפעות של גירויים במהלך השלב הראשי מיושמות על ה-DUT ושכל הנתונים המתקבלים יטופלו כראוי, מה שעלול להפעיל רצפים גוזלים זמן. לבסוף, שלב post_shutdown מבצע פעולות אחרונות לפני היציאה מהסימולציה.

שלבים אלה ב-UVM עוזרים לייעל את תהליך הורפיקציה, להבטיח שה-DUT מאותחל כהלכה, גירויים מיושמים ביעילות, והתוצאות מנוהלות כראוי לאורך מחזור הורפיקציה.

ג. שלבי ניקוי:

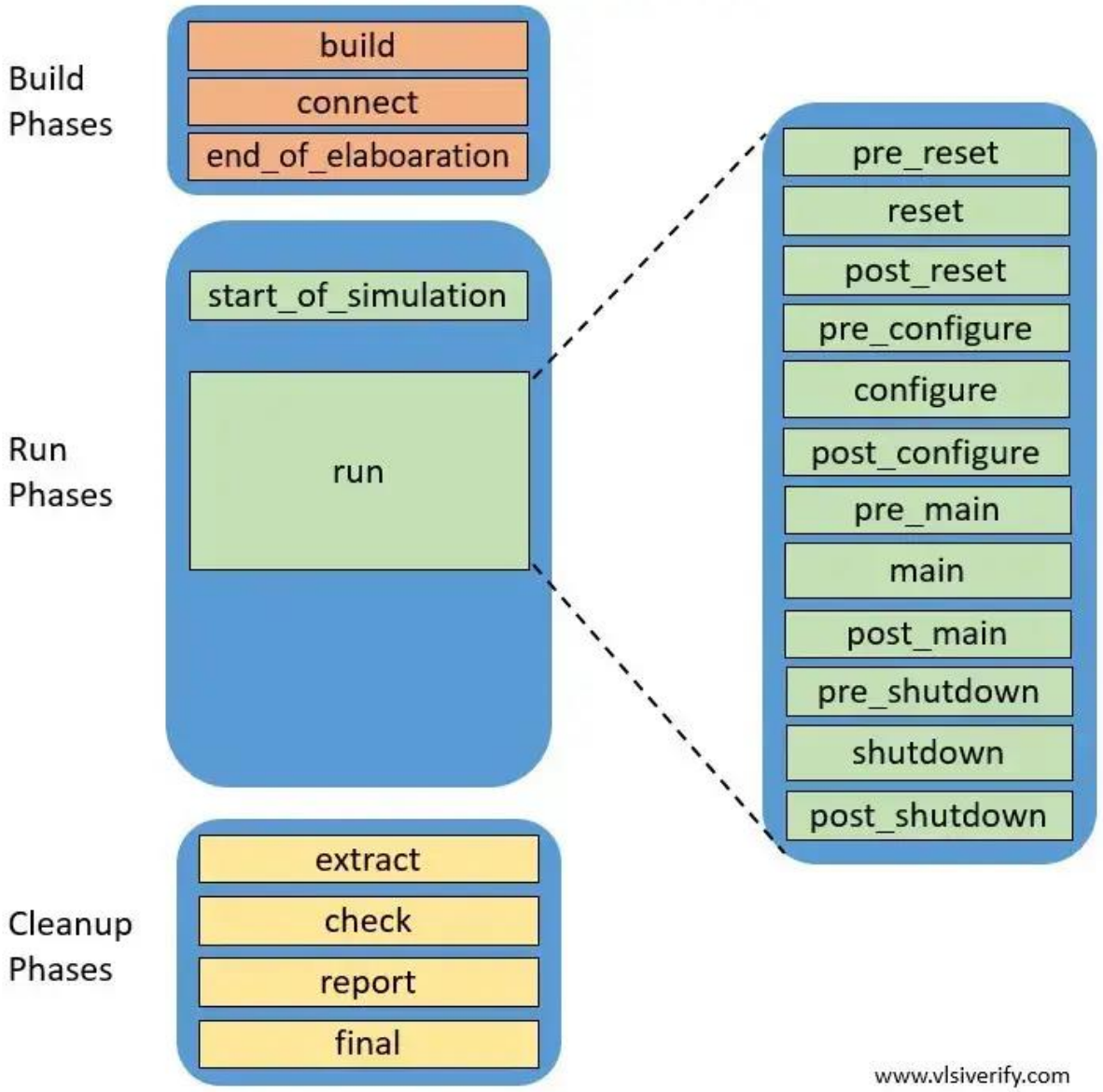
extract phase: משמש לחילוץ וחישוב נתונים צפויים מלוח התוצאות.

check phase: מבצע משימות לוח תוצאות, בודק שגיאות בין הערכים הצפויים והממשיים מהתכנון.

report phase: מציג תוצאות מבודקים או מסכם יעדי בדיקה אחרים.

final phase: משמש בדרך כלל לפעולות של הרגע האחרון לפני היציאה מהסימולציה.

תמונה להמחשת השלבים:



מטרת השלבים:

סנכרון: שלבי UVM מבטיחים שהרכיבים מתקדמים בצורה מסונכרנת. אף רכיב לא ממשיך לשלב הבא עד שכל תהליך סיים את השלב הנוכחי.

Callbacks: שלבים מוגדרים כהתקשרויות, המאפשרות למחלקות לבצע עבודה שימושית בשיטת Callback Phase.

גמישות: התקשרויות לפני ואחרי מספקות גמישות לכל שלבי זמן הריצה.

ההבדל אל מול Verilog Testbenches:

בtestbench של Verilog, הרכיבים הם מודולים סטטיים, כך שהם אינם זקוקים לשלבים מפורשים.

המבנה המקנן של UVM דורש בנייה מלמעלה למטה (סביבה, סוכן, מחלקת בדיקה).

שלבי UVM מנהלים את ביצועיהם של רכיבי הורפיקציה, ומבטיחים סימולציה שיטתית ומדויקת.

ALU - דוגמה מעשית של UVM Verification

בדוגמה הבאה אנחנו נציג שימוש של UVM Verification על רכיב ALU8 bit.
ALU הוא חלק חשוב של ה- CPU שמבצע חישובים מתמטיים ופעולות לוגיות על נתונים. הוא יכול לבצע פועלות אריתמטיות כמו חיבור חיסור וחילוק ולוגיות כמו And, OR וכו'. הפלטים של ה-ALU חיוניים לביצוע חישובים, ביצוע השוואות וביצוע פעולות שונות בתוך ה-CPU.

ל-ALU הבא הגדרנו את התנאים הבאים:

- הALU פועל בעליית שעון והמוצא משתנה בעת פעימת השעון הבאה.
- הALU בעל כניסת איפוס אסינכרונית הפעילה בגבוה (reset=1).
- לALU ארבעה מצבי הפעולה הבאים: חיבור, חיסור, כפל וחילוק. בעל אפשרות הוספת פעולות אריתמטיות נוספות - Reserved. (4 OP Code bit כלומר עד 16 פעולות בסה"כ).

Port Name	Input/Output	Property	Size
Clock	Input	Wire	1 bit
Reset	Input	Wire	1 bit
A	Input	Wire	8 bits
B	Input	Wire	8 bits
ALU_Sel	Input	Wire	4 bits
ALU_Out	Output	Reg	8 bits
CarryOut	Output	Bit	1 bit

ALU_Sel	Operation
4’b0000	A + B
4’b0001	A - B
4’b0010	A * B
4’b0011	A / B
4’b0100 – 4’b1111	Reserved

Design

```
/*
ALU Arithmetic and Logic Operations
-----------------------------------------------------------------------
|ALU_Sel|   ALU Operation
-----------------------------------------------------------------------
| 0000  |   ALU_Out = A + B;
-----------------------------------------------------------------------
| 0001  |   ALU_Out = A - B;
-----------------------------------------------------------------------
| 0010  |   ALU_Out = A * B;
-----------------------------------------------------------------------
| 0011  |   ALU_Out = A / B;
-----------------------------------------------------------------------
*/
module alu(
  input clock,
  input reset,
  input [7:0] A,B,  // ALU 8-bit Inputs
  input [3:0] ALU_Sel,// ALU Selection
  output reg [7:0] ALU_Out, // ALU 8-bit Output
  output bit CarryOut // Carry Out Flag
);
  reg [7:0] ALU_Result;
  wire [8:0] tmp;
  assign tmp = {1'b0,A} + {1'b0,B};

  always @(posedge clock or posedge reset) begin
    if(reset) begin
      ALU_Out  <= 8'd0;
      CarryOut <= 1'd0;
    end
    else begin
      ALU_Out <= ALU_Result;
      CarryOut <= tmp[8];
    end
  end

  always @(*)
    begin
      case(ALU_Sel)
        4'b0000: // Addition
          ALU_Result = A + B ;
        4'b0001: // Subtraction
          ALU_Result = A - B ;
        4'b0010: // Multiplication
          ALU_Result = A * B;
        4'b0011: // Division
          ALU_Result = A/B;
        default: ALU_Result = 8'hAC ; // Give out random BAD value
      endcase
    end
endmodule
```

- קובץ ה-Design מייצג יחידת לוגיקה אריתמטית (ALU) עם ארבעה מצבי פעולה (חיבור, חיסור, כפל וחילוק) המבוססים על סיגנל בחירה של 4 סיביות (ALU_Sel), עם קלט של 8 סיביות (A ו-B), פלט של 8 סיביות (ALU_Out), ו Carry Out. ה-ALU_Result מחושב על סמך הסיגנלים ALU_Sel, ו-ALU_Out ומשרשר אל ה CarryOut ל MSB של ALU OUT.

ALU Interface

```
interface alu_interface(input logic clock);
  logic reset;
  logic [7:0] a, b;
  logic [3:0] op_code;
  logic [7:0] result;
  bit carry_out;

endinterface: alu_interface
```

- ה- Interface חיוני מכיוון שהוא מגדיר את האותות בין הALU לרכיבי הTestbench, בכדי לאפשר העברת נתונים ובקרה בצורה חלקה עבור משימות הוריפקציה. בכך הוא משפר את המודולריות של סביבות הוריפיקציה ומאפשר שימוש חוזר.

Testbench

```
`timescale 1ns/1ns
import uvm_pkg::*;
`include "uvm_macros.svh"

`include "interface.sv"
`include "sequence_item.sv"
`include "sequence.sv"
`include "sequencer.sv"
`include "driver.sv"
`include "monitor.sv"
`include "agent.sv"
`include "scoreboard.sv"
`include "env.sv"
`include "test.sv"

module top;
  logic clock;
  alu_interface intf(.clock(clock));

  alu dut(
    .clock(intf.clock),
    .reset(intf.reset),
    .A(intf.a),
    .B(intf.b),
    .ALU_Sel(intf.op_code),
    .ALU_Out(intf.result),
    .CarryOut(intf.carry_out)
  );

  initial begin
    uvm_config_db #(virtual alu_interface)::set(null, "*", "vif", intf );
    //-- Refer:
https://www.synopsys.com/content/dam/synopsys/services/whitepapers/hierarchical-testbench-
configuration-using-uvm.pdf
  end

  initial begin
    run_test("alu_test");
  end

  //Clock Generation
  initial begin
    clock = 0;
    #5;
    forever begin
      clock = ~clock;
      #2;
    end
  end

  //Maximum Simulation Time
  initial begin
    #5000;
```

```
    $display("Sorry! Ran out of clock cycles!");
    $finish();
  end

  //Generate Waveforms
  initial begin
    $dumpfile("d.vcd");
    $dumpvars();
  end

endmodule: top
```

- הTestbench הוא ה top level והוא מגדיר בדיקה מבוססת UVM עבור הALU. הוא כולל קריאה למודולים עבור הרכיבים השונים, בנוסף מחבר את המודול של ה Design עם ה- ALU, ומגדיר אותו עם הInterface. ה-Testbench מייצר אות שעון,לאחר מכן מפעיל את סט הבדיקות דרך רכיב ה test, מגדיר זמן סימולציה מקסימלי ומייצר פלט של waveform (בקובץ).

Test

```
class alu_test extends uvm_test;
  `uvm_component_utils(alu_test)

  alu_env env;
  alu_base_sequence reset_seq;
  alu_test_sequence test_seq;

  //Constructor
  function new(string name = "alu_test", uvm_component parent);
    super.new(name, parent);
    `uvm_info("TEST_CLASS", "Inside Constructor!", UVM_HIGH)
  endfunction: new

   //Build Phase
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TEST_CLASS", "Build Phase!", UVM_HIGH)

    env = alu_env::type_id::create("env", this);

  endfunction: build_phase

  //Connect Phase
  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    `uvm_info("TEST_CLASS", "Connect Phase!", UVM_HIGH)

  endfunction: connect_phase

  //Run Phase
  task run_phase (uvm_phase phase);
    super.run_phase(phase);
    `uvm_info("TEST_CLASS", "Run Phase!", UVM_HIGH)

    phase.raise_objection(this);

    //reset_seq
    reset_seq = alu_base_sequence::type_id::create("reset_seq");
    reset_seq.start(env.agnt.seqr);
    #10;

    repeat(100) begin
      //test_seq
      test_seq = alu_test_sequence::type_id::create("test_seq");
      test_seq.start(env.agnt.seqr);
      #10;
    end

    phase.drop_objection(this);

  endtask: run_phase

endclass: alu_test
```

- ה- Test מציג את ה- Test Flow והוא מתאחל 3 טיפוסים מסוג - `alu_env`, `alu_base_sequence`, `alu_test_sequence`.
ה- Test פועל בשלושה שלבים: בנייה, חיבור והרצה.
בשלב הבנייה, הוא מאתחל את הרכיב Environment שהוא משתנה בתחת alu_test. בשלב החיבור מטפל בהגדרת הקישוריות.
בשלב הריצה ה test יוצר את את הרצפים test sequence ו rest sequence.
בתחילת הטסט בודקים את הריסט פעם אחת לאחר 10 יחידות זמן שמוגדרות ב TestBench ואז מריצים 100 חזרות של טסטים עבור test_sequence.
את הרצפים של הבדיקות מעבירים ל sequencer דרך הפקודה `.start(env.agnt.seqr;`
בצורה הזאת הוא שולח את הרצפים ל Sequencer כיוון שה Sequencer שנמצא בתוך הagent והagent בתוך ה- environment.

Sequence

```
class alu_base_sequence extends uvm_sequence;
  `uvm_object_utils(alu_base_sequence)
  alu_sequence_item reset_pkt;
  //Constructor
  function new(string name= "alu_base_sequence");
    super.new(name);
    `uvm_info("BASE_SEQ", "Inside Constructor!", UVM_HIGH)
  endfunction
  //Body Task
  task body();
    `uvm_info("BASE_SEQ", "Inside body task!", UVM_HIGH)

    reset_pkt = alu_sequence_item::type_id::create("reset_pkt");
    start_item(reset_pkt);
    reset_pkt.randomize() with {reset==1;};
    finish_item(reset_pkt);

  endtask: body
endclass: alu_base_sequence
class alu_test_sequence extends alu_base_sequence;
  `uvm_object_utils(alu_test_sequence)
  alu_sequence_item item;
  //Constructor
  function new(string name= "alu_test_sequence");
    super.new(name);
    `uvm_info("TEST_SEQ", "Inside Constructor!", UVM_HIGH)
  endfunction
  //Body Task
  task body();
    `uvm_info("TEST_SEQ", "Inside body task!", UVM_HIGH)
    item = alu_sequence_item::type_id::create("item");
    start_item(item);
    item.randomize() with {reset==0;};
    finish_item(item);
  endtask: body
endclass: alu_test_sequence
```

- המחלקה alu_base_sequence מטפלת ברצף האיפוס, ומבטיחה שה-ALU מאותחל לפני תחילת הבדיקה. מחלקת ה- alu_test_sequence אחראית על יצירת רצפי בדיקה על ידי הפצת ערכי קלט אקראיים (למעט האיפוס - Reset שנשאר שווה ל-0). בנוסף, ביצוע טרנזקציות כדי להעריך את הפונקציונליות של ה-ALU תחת OP Codes שונים. הטרנזקציה מכילה את כל התוכן בין ה start_item() ל finish_item() כך שכל הפעולות שמבוצעות בין ה Start ל-Finish נחשבות כטרנזקציה אחת.

Packet/Sequence Item

```
class alu_sequence_item extends uvm_sequence_item;
  `uvm_object_utils(alu_sequence_item)

  rand logic reset;
  rand logic [7:0] a, b;
  rand logic [3:0] op_code;

  logic [7:0] result; //output
  bit carry_out; // output

  //Default Constraints
  constraint input1_c {a inside {[10:20]};}
  constraint input2_c {b inside {[1:10]};}
  constraint op_code_c {op_code inside {0,1,2,3};}


  //Constructor
  function new(string name = "alu_sequence_item");
    super.new(name);

  endfunction: new

endclass: alu_sequence_item
```

- המחלקה Packet או Sequence Item בקוד כולל הגבלות על ערכי הכניסה האקראיים a ו b וגם עבור הopcode. בנוסף, הוא מגדיר את התוצאה כoutput. ההגבלות שהגדרנו מבטיחות תרחישי בדיקה מגוונים שנוצרים במהלך הסימולציה, המכסים מגוון של שילובי קלט וסוגי פעולה כדי לאמת ביסודיות את הפונקציונליות של ה-ALU.
בתרגיל זה הגדרנו הגבלה על ערך כניסה A שיקבל ערכים בין 10 ל-20 ולערך כניסה B ערכים בין 1 ל-10 (דצימלי). הסיבה שהגדרנו ערכים אלו, היא בשביל הפשטות שיהיה לנו קל לראות ב Waveform ובlog את התוצאות ולהבין למה קיבלנו תוצאה כזאת או אחרת.
sequence item נוצר ומנוצל בתוך רצפים כדי להניע טרנזקציות ולהעריך את ביצועי ה-ALU בצורה מדויקת.

Environment

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

  alu_agent agnt;
  alu_scoreboard scb;


  //Constructor
  function new(string name = "alu_env", uvm_component parent);
    super.new(name, parent);
    `uvm_info("ENV_CLASS", "Inside Constructor!", UVM_HIGH)
  endfunction: new

  //Build Phase
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("ENV_CLASS", "Build Phase!", UVM_HIGH)

    agnt = alu_agent::type_id::create("agnt", this);
    scb = alu_scoreboard::type_id::create("scb", this);

  endfunction: build_phase


  //Connect Phase
  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    `uvm_info("ENV_CLASS", "Connect Phase!", UVM_HIGH)

    agnt.mon.monitor_port.connect(scb.scoreboard_port);

  endfunction: connect_phase


  //Run Phase
  task run_phase (uvm_phase phase);
    super.run_phase(phase);

  endtask: run_phase


endclass: alu_env
```

- בתחילת הקוד הConstructor יוצר את המחלקה של הEnvironment. בשלב הבנייה הenvironment יוצרת ומאתחלת את הAgent ואת ה Scoreboard שהן מחלקות בתוך ה- environment. בשלב החיבור הוא יוצר קישור בין המוניטור שהוא תת מחלקה בתוך ה Agent לScoreboard מה שמאפשר לטרנזקציות בין הרכיבים. באופן כללי הEnviorment יוצרת את הרכיבים הנדרשים לבדיקה הפונקציונליות של הALU בשיטת UVM.

Agent

```
class alu_agent extends uvm_agent;
  `uvm_component_utils(alu_agent)
  alu_driver drv;
  alu_monitor mon;
  alu_sequencer seqr;

  //Constructor
  function new(string name = "alu_agent", uvm_component parent);
    super.new(name, parent);
    `uvm_info("AGENT_CLASS", "Inside Constructor!", UVM_HIGH)
  endfunction: new

  //Build Phase
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("AGENT_CLASS", "Build Phase!", UVM_HIGH)

    drv = alu_driver::type_id::create("drv", this);
    mon = alu_monitor::type_id::create("mon", this);
    seqr = alu_sequencer::type_id::create("seqr", this);

  endfunction: build_phase

   //Connect Phase
  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    `uvm_info("AGENT_CLASS", "Connect Phase!", UVM_HIGH)

    drv.seq_item_port.connect(seqr.seq_item_export);

  endfunction: connect_phase

  //Run Phase
  task run_phase (uvm_phase phase);
    super.run_phase(phase);

  endtask: run_phase


endclass: alu_agent
```

- מחלקת ה- agent ממלאת תפקיד מרכזי בתיאום וניהול הרכיבים השונים המעורבים בverification design של ה- ALU. הוא כולל את רכיבי הדרייבר, המוניטור והרצף, שלכל אחד מהם תחומי אחריות ספציפיים. הדרייבר אחראי להנעת טרנזקציות ל-ALU בהתבסס על פריטי הרצף המתקבלים מהרצף. המוניטור מקבל סיגנלים מה-ALU, דוגם כניסות ויציאות ושולח טרנזקציות ללוח התוצאות להשוואה. הSequencer מעביר רצף של טרנזקציות שיופעלו על ידי הדרייבר, ומבטיח זרימה של פעולות בדיקה. יחד, רכיבים אלו בתוך הAgent מאפשרים בדיקה ואימות יסודיים של הפונקציונליות של ה-ALU.

Sequencer

```
class alu_sequencer extends uvm_sequencer#(alu_sequence_item);
  `uvm_component_utils(alu_sequencer)

  //Constructor
  function new(string name = "alu_sequencer", uvm_component parent);
    super.new(name, parent);
    `uvm_info("SEQR_CLASS", "Inside Constructor!", UVM_HIGH)
  endfunction: new

   //Build Phase
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("SEQR_CLASS", "Build Phase!", UVM_HIGH)

  endfunction: build_phase

   //Connect Phase
  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    `uvm_info("SEQR_CLASS", "Connect Phase!", UVM_HIGH)

  endfunction: connect_phase


endclass: alu_sequencer
```

- מחלקת ה- Sequencer מתזמנת את זרימת הטרנזקציות הנשלחות לALU למטרות בדיקה. הוא פועל כבקר מרכזי להפקה וניהול של רצפי פעולות שיבוצעו על ידי ה-ALU במהלך סימולציה. הרצף מוודא שעסקאות מונעות בצורה מובנית בהתאם לדרישות הבדיקה המוגדרות ברצפים. על ידי תיאום סדר ותזמון הטרנזקציות, ה- Sequencer תורם לורפיקציה יסודית של הפונקציונליות של ה-ALU תחת תרחישי בדיקה שונים.

Driver

```
class alu_driver extends uvm_driver#(alu_sequence_item);
  `uvm_component_utils(alu_driver)
  virtual alu_interface vif;
  alu_sequence_item item;
  //Constructor
  function new(string name = "alu_driver", uvm_component parent);
    super.new(name, parent);
    `uvm_info("DRIVER_CLASS", "Inside Constructor!", UVM_HIGH)
  endfunction: new
  //Build Phase
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("DRIVER_CLASS", "Build Phase!", UVM_HIGH)
    if(!(uvm_config_db #(virtual alu_interface)::get(this, "*", "vif", vif))) begin
      `uvm_error("DRIVER_CLASS", "Failed to get VIF from config DB!")
    end
  endfunction: build_phase
  //Connect Phase
  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    `uvm_info("DRIVER_CLASS", "Connect Phase!", UVM_HIGH)
  endfunction: connect_phase
  //Run Phase
  task run_phase (uvm_phase phase);
    super.run_phase(phase);
    `uvm_info("DRIVER_CLASS", "Inside Run Phase!", UVM_HIGH)
    forever begin
      item = alu_sequence_item::type_id::create("item");
      seq_item_port.get_next_item(item);
      drive(item);
      seq_item_port.item_done();
    end
   endtask: run_phase
  //[Method] Drive
  task drive(alu_sequence_item item);
    @(posedge vif.clock);
    vif.reset <= item.reset;
    vif.a <= item.a;
    vif.b <= item.b;
    vif.op_code <= item.op_code;
  endtask: drive

endclass: alu_driver
```

- מחלקת דרייבר אחראית על העברת טרנזקציות מהSequencer לALU. הוא מקיים אינטראקציה עם הסיגנלים שבinterface של ה-ALU כגון שעון, reset, כניסות A ו-B וOPcode, ומבטיח שטרנזקציות מיושמות כהלכה על ה-ALU בהתאם לtest sequences. בשלב הריצה, הדרייבר עוקב באופן רציף אחר Sequence Items נכנסים, מעבירה אותם אל ה-DUT באמצעות סיגנלים של הinterface, ומסמנת אותם כ-הושלמו לאחר ביצוע הפעולה.

Monitor

```
class alu_monitor extends uvm_monitor;
  `uvm_component_utils(alu_monitor)
  virtual alu_interface vif;
  alu_sequence_item item;
  uvm_analysis_port #(alu_sequence_item) monitor_port;
  //Constructor
  function new(string name = "alu_monitor", uvm_component parent);
    super.new(name, parent);
    `uvm_info("MONITOR_CLASS", "Inside Constructor!", UVM_HIGH)
  endfunction: new
  //Build Phase
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("MONITOR_CLASS", "Build Phase!", UVM_HIGH)
    monitor_port = new("monitor_port", this);
    if(!(uvm_config_db #(virtual alu_interface)::get(this, "*", "vif", vif))) begin
      `uvm_error("MONITOR_CLASS", "Failed to get VIF from config DB!")
    end
  endfunction: build_phase

  //Connect Phase
  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    `uvm_info("MONITOR_CLASS", "Connect Phase!", UVM_HIGH)

  endfunction: connect_phase
  //Run Phase
  task run_phase (uvm_phase phase);
    super.run_phase(phase);
    `uvm_info("MONITOR_CLASS", "Inside Run Phase!", UVM_HIGH)
    forever begin
      item = alu_sequence_item::type_id::create("item");
      wait(!vif.reset);
      //sample inputs
      @(posedge vif.clock);
      item.a = vif.a;
      item.b = vif.b;
      item.op_code = vif.op_code;
      //sample output
      @(posedge vif.clock);
      item.result = vif.result;
      // send item to scoreboard
      monitor_port.write(item);
    end
  endtask: run_phase
endclass: alu_monitor
```

- מחלקת מוניטור משמשת כרכיב המנטר סיגנלים וטרנזקציות הקשורות לALU במהלך סימולציה. הוא לוקח כניסות ויציאות מה-ALU -כמו A ו-B, קוד פעולה (op_code), Resultו. לאחר מכן המוניטור דוגם אותם בנקודות זמן ספציפיות על סמך אות השעון. אחר כך המוניטור שולח את פריטי הטרנזקציות הנדגמים ללוח התוצאות להשוואה לתוצאות הצפויות. על ידי ניטור התנהגות ה-ALU בזמן אמת והפקת נתוני טרנזקציוצת לניתוח, המוניטור תורם לאימות הנכונות והפונקציונליות של ה-ALU.

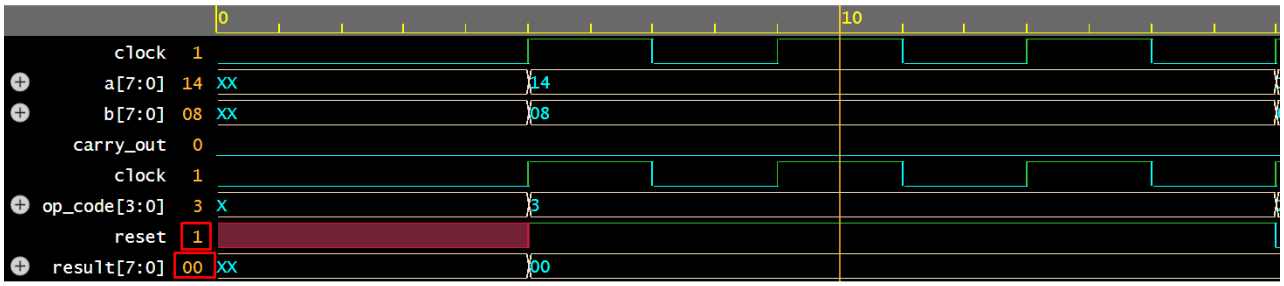
Scoreboard

```
class alu_scoreboard extends uvm_test;
  `uvm_component_utils(alu_scoreboard)

  uvm_analysis_imp #(alu_sequence_item, alu_scoreboard) scoreboard_port;

  alu_sequence_item transactions[$];

  //Constructor
  function new(string name = "alu_scoreboard", uvm_component parent);
    super.new(name, parent);
    `uvm_info("SCB_CLASS", "Inside Constructor!", UVM_HIGH)
  endfunction: new


  //Build Phase
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("SCB_CLASS", "Build Phase!", UVM_HIGH)

    scoreboard_port = new("scoreboard_port", this);

  endfunction: build_phase

  //Connect Phase
  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    `uvm_info("SCB_CLASS", "Connect Phase!", UVM_HIGH)


  endfunction: connect_phase

  //Write Method
  function void write(alu_sequence_item item);
    transactions.push_back(item);
  endfunction: write

  //Run Phase
  task run_phase (uvm_phase phase);
    super.run_phase(phase);
    `uvm_info("SCB_CLASS", "Run Phase!", UVM_HIGH)

    forever begin
      /*
      // get the packet
      // generate expected value
      // compare it with actual value
      // score the transactions accordingly
      */
      alu_sequence_item curr_trans;
      wait((transactions.size() != 0));
      curr_trans = transactions.pop_front();
      compare(curr_trans);
    end
  endtask: run_phase
```

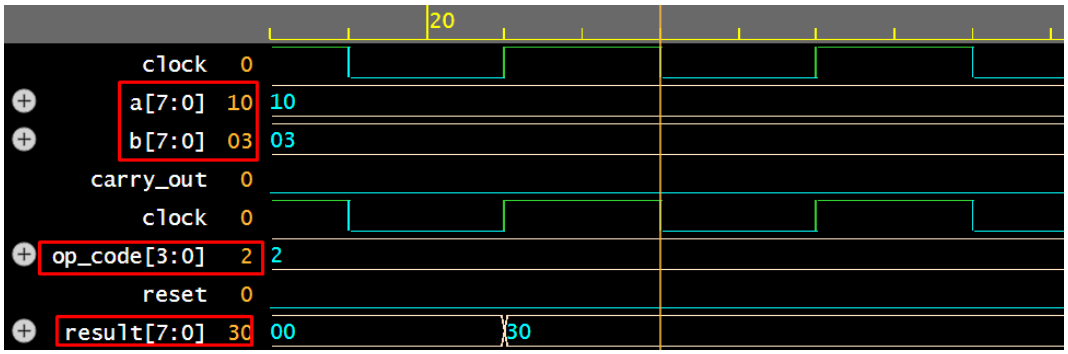
```
  //Compare : Generate Expected Result and Compare with Actual
  task compare(alu_sequence_item curr_trans);
    logic [7:0] expected;
    logic [7:0] actual;

    case(curr_trans.op_code)
      0: begin //A + B
        expected = curr_trans.a + curr_trans.b;
      end
      1: begin //A - B
        expected = curr_trans.a - curr_trans.b;
      end
      2: begin //A * B
        expected = curr_trans.a * curr_trans.b;
      end
      3: begin //A / B
        expected = curr_trans.a / curr_trans.b;
      end
    endcase

    actual = curr_trans.result;

    if(actual != expected) begin
      `uvm_error("COMPARE", $sformatf("Transaction failed! ACT=%d, EXP=%d", actual, expected))
    end
    else begin
      // Note: Can display the input and op_code as well if you want to see what's happening
      `uvm_info("COMPARE", $sformatf("Transaction Passed! ACT=%d, EXP=%d", actual, expected),
UVM_LOW)
    end

  endtask: compare


endclass: alu_scoreboard
```

- מחלקת ה- Scoreboard משמשת כרכיב קריטי לאימות הפונקציונליות של ה- Design של ה-ALU. הוא משווה את התוצאות בפועל שהפיק ה-ALU עם התוצאות הצפויות בהתבסס על הפעולה שבוצעה, כמפורט בSequence Item. ה- Scoreboard שומר רשימה של טרנזקציות ובודק באופן רציף האם הפלט המחושב תואם את הפלט הצפוי. אם מתגלה אי התאמה, הוא מדווח על שגיאות, ומספק תובנות לגבי נכונות פעולות ALU. מנגנון אימות זה מבטיח שה-ALU מתנהג כמצופה בתרחישי בדיקה שונים, ותורם לורפיקציה טובה של ה- Design.

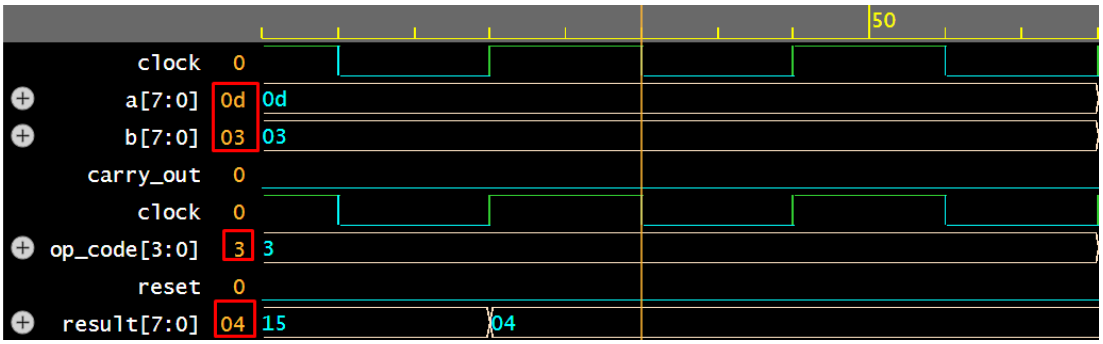
דיאגרמת זמנים Wave Form



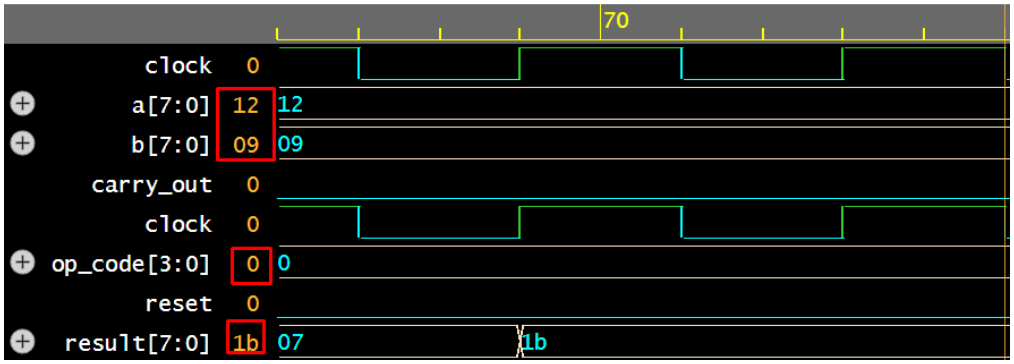
בדיקה עבור הReset, התוצאה תהיה hex00.



כאשר אנו ב- OP Code 0010 המגדיר פעולת כפל, לכן עבור A=10hex, B=03hex נקבל Result = 30hex. כפי שהגדרנו, את התוצאה נקבל רק בפעימת השעון הבאה.



כאשר אנו ב- OP Code 0011 המגדיר פעולת חילוק, לכן עבור A=0dhex, B=03hex נקבל $Result = \frac{13}{03} = 4$.



כאשר אנו ב- OP Code 0000 המגדיר פעולת חיבור, לכן עבור A=12hex, B=09hex נקבל $Result = 18 + 9 = 27dec = 1Bhex$.

סיכום ומסקנות

UVM היא שיטת עבודה סטנדרטית לdesign verification ומערכות-על-שבב (SoCs) בתעשיית המוליכים למחצה. הוא מספק מסגרת ליצירת רכיבי Testbench מודולריים הניתנים לשימוש חוזר שניתן לשלב בקלות בתהליך verification design. חלק ממרכיבי המפתח של UVM כוללים רכיבי Testbench שהם דרייבר, מוניטור, לוחות תוצאות וAgent.

מסקנות מUVM:

- שימוש חוזר: הדגש של UVM על מודולריות ושימוש חוזר מאפשר למהנדסי ורפיקציה לפתח ספרייה של רכיבים גנריים כמו רצפים, Agents, דרייבים ורכיבים אחרים
- Scaleability: שיטת UVM משפרת את הסקיילאביליות, ומאפשרת התאמה קלה לדרישות הפרויקט המשתנות.
- יעילות: UVM מייעלת את תהליך הוריפיקציה, מקדמת פרודוקטיביות ומבטיחה Testbench ניתנים להתאמה.
- מודולריות: המתודולוגיה מתוכננת כרכיבים מודולריים (Driver, Sequencer, Agents, Env וכו') וזה מאפשר שימוש חוזר ברכיבים פשוטים (ALU, Counterm, Full Adder) ומורכבים (SoC/Chip).
- הפרדה של ה-Test מ-Testbenches:טסטים של רצפים נשמרים בנפרד מההיררכיה של ה Testbench בפועל, ומכאן שניתן לעשות שימוש חוזר בStimulus על פני פרויקטים. לדוגמה, יהיו כמה קלאסים של Test אך כולם ירוצו באותו ה Testbench.
- Factory (Design Pattern): זה מפשט את השינוי של רכיבים בקלות. יצירת כל רכיב באמצעות קונסטרקטור מאפשרת לעקוף אותם בבדיקות או בסביבות שונות מבלי לגעת בקוד. (כיוון שיש הורשה של הבדיקות ממחלקת האב)