הטכניון - מ.ט.ל. הפקולטה להנדסת חשמל
המעבדה ל - VLSI

חוברת הדרכה על שפות Verilog ו- SystemVerilog

נכתב ע"י גואל סמואל ואמנון סטניסלבסקי
להערות ושאלות : goel@ee.technion.ac.il
עדכון אחרון : 15/09/2009

תוכן העניינים

הקדמה 3
מבוא לשפת Verilog 3
סוגי משתנים 4
אופרטורים נפוצים 4
הגדרת ממשק 5
פרמטרים למודול 5
תיאור פונקציונלי 5
אינסטנסיאציה – instantiation (הצבה) 5
משפטי always 6
משפטי השמה 7
השמה מסוג continuous 7
השמה סדרתית (procedural או sequential) 7
Functions ו- Tasks 8
משפט generate 10
כתיבת testbench 11
התוספות של SystemVerilog לשפת Verilog 13
סוגי משתנים חדשים 13
typedef 14
enum 14
struct 14
class 14
אינסטנסיאציה בשפת SystemVerilog 15
מערכים 15
משפטי always 16
always_comb 16
always_latch 17
always_ff 17
unique ו- priority 17
לולאות - loops 18
משפט final 19
שליטה באירועים (event)program block 19
randomization 20
משתנים אקראיים 20
אילוצים על משתנים אקראיים 21
משקולות 21
אקראיות מותנת 21
randcase 22
Assertions 22
Immediate Assertions 22
Concurrent Assertions 23
משפט expect 26
functional coverage 26
דוגמא מסכמת 28
סימולצייה של מעגלים ב- Verilog/SystemVerilog 31
סינטזה של מעגלים ב- Verilog/SystemVerilog 31

הקדמה

שפות תיאור החומרה (HDL-Hardware Description Language) למיניהן דוגמת VHDL ו-Verilog נוצרו על מנת לעזור למתכנתים לוגיים ליצור ולממש מעגלים במהירות, אולם ככל שהמעגלים גדלו בעיית הוריפיקציה הפכה לקשה יותר ויותר. לשם כך נולד תחום שפות הוריפקציה (HVL – Hardware Verification Language) דוגמת e ו- Vera אשר נוצרו על מנת להקל על מלאכת הבדיקה. אולם כעת נוצר מצב שמתכנת לוגי צריך ללמוד שתי שפות שונות לחלוטין על מנת ליצור ולבדוק רכיבים לוגיים.

על מנת לפתור בעיה זו וכן על מנת להעשיר ולשפר את השפה עצמה, הורחבה שפת ה- Verilog לשפת SystemVerilog אשר היא שפת ה- HVDL – Hardware Verification & Description Language הראשונה. שפה זו מוסיפה לשפת ה- Verilog הבסיסית יכולות וריפיקציה אוטומוטיות וגם מספר יכולות מעולם התוכנה, וכעת התכנון וגם ה-testbench שלו כתובים באותה השפה. מדריך זה בא להסביר על קצה המזלג כיצד לכתוב רכיבי חומרה ורכיבי בדיקה ב-SystemVerilog , אולם אינו מתיימר לכסות את מלוא החומר.

למדריכים מקיפים יותר ניתן לגשת ל:

http://www.asic-world.com/systemverilog/tutorial.html
http://electrosofts.com/SystemVerilog/

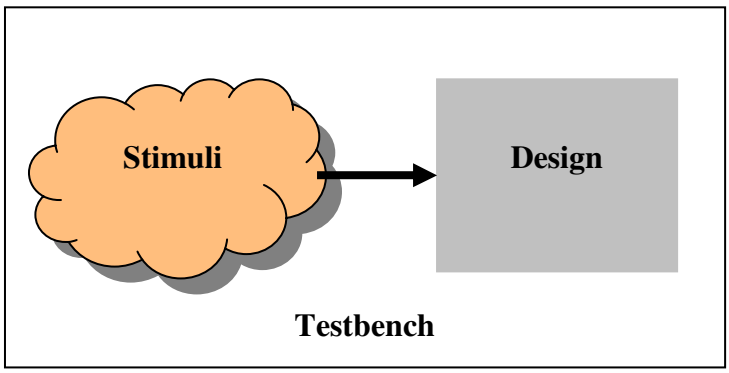
כאמור שפת SystemVerilog היא שפה לתיאור חומרה שבעזרתה ניתן לתכנן, לממש ולבצע סימולציה של מערכות מורכבות. במגבלות מסוימות ניתן להפוך תיאור SystemVerilog של מערכת למעגל חשמלי באמצעות כלי סינתזה. שפת SystemVerilog מאפשרת מימוש ברמת הפשטה גבוהה יותר לעומת Verilog. מבנים ופקודות כגון טיפוסי data חדשים (logic,int), enumerated types, מערכים, משפטי always ייעודיים (always_ff, always_comb) מאפשרים מימוש בקוד יותר קומפקטי. בשפה גם ממשקים ישירים לשפות נוספות כגון C, C++, systemC ועוד.

כפי שצוין, בנוסף להיותה שפה לתיאור החומרה Hardware Description Language (HDL) שפת SystemVerilog היא גם שפת וריפיקציה. היא מאפשרת מימוש של סביבת וריפיקציה שלמה הכוללת יצירה רנדומלית ומכוונת של אותות כניסה, ניתוח מידת הכיסוי של הבדיקות ווריפיקציה מבוססת assertions ובכך מאפשרת בניה של התכנון וסביבת הבדיקה באותה שפה. חשוב לציין שרק קוד שמתאר את ההתנהגות הלוגית מועבר לשלב הסינתה ורק קוד שעומד בחוקי הסינתזה יעבור שלב זה בהצלחה.

כאמור, שפת SystemVerilog היא הרחבה של שפת Verilog ולכן טבעי להתחיל בהסברים על שפת Verilog. בניסוי עצמו נעשה מאמץ להתמקד בעיקר על ה- features החדשים ש-SystemVerilog מציעה.

מבוא לשפת Verilog

המשימה העיקרית של העבודה עם שפת HDL היא מימוש התכנון וסביבת הסימולציה:



איור מס' 1

בשפת Verilog התכנון ממומש באמצאות משפט module שניתן לתאר באופן הבא :

```
module name ( )
  interface // input and output definition
  functional behaviour :
     // assign statements
     // always statements
     // instantiation of other module
endmodule
```

המודול מכיל את הגדרת הממשק (כניסות ויציאות) ואת תיאור ההתנהגות של הרכיב. ניתן לתאר את ההתנהגות ע"י משפטי השמה, משפטי always או ע"י הצבה של מודולים אחרים. ה-testbench מכיל הצבה של התכנון וקוד שיוצר את אותות הכניסה עבור התכנון. הסברים מפורטים יובאו בהמשך. לעתים (גם בניסוי זה) ליחידה stimuli בלבד קוראים testbench.

סוגי משתנים

בשפת Verilog קיימים מספר "סוגי משתנים". הנפוצים שבהם :

1. משתנה אשר מחזיק ערך (לדוגמא – יציאה של flip-flop) – מסומן ע"י reg
2. משתנה אשר אינו מחזיק ערך אלא מחבר בין שתי יחידות – מסומן ע"י wire
3. משתנה שהוא port של מודול יכול להיות מסוג כניסה(input), יציאה (output) או שניהם (inout)

אופרטורים נפוצים

סימנו ב Verilog	פעולת האופרטור
*	Multiply
/	Division
+	Add
-	Subtract
%	Modulus
+	Unary plus
-	Unary minus
!	Logical negation
&&	Logical and
||	Logical or
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
==	Equality
!=	inequality
~	Bitwise negation
~&	nand
|	or
~|	nor

^	xor
^~	xnor
~^	xnor
>>	Right shift
<<	Left shift
{ }	Concatenation
?	conditional

- המתן מספר יחידות זמן, לדוגמא : `#5 clk = ~clk;`
ימתין 5 יחידות זמן לפני ההשמה.

לאחר הגדרת הממשק החיצוני ומשתני המודול צריך לתאר את התנהגות המודול. ניתן לעשות זאת במספר דרכים שונות :

1. הצבה של מודולים אחרים (אינסטנציאציה)
2. משפטי השמה מסוג continuous
3. **שימוש במשפטים מסוג always לתיאור ההתנהגות הלוגית**

הגדרת ממשק

ראשית, יש להגדיר את הממשק לרכיב, ממשק זה יגדיר את סוגי ומימדי הכניסות והיציאות של הרכיב. לדוגמא – לשער and הממשק הוא : שתי כניסות מסוג ביט בודד ויציאה אחת מסוג ביט בודד.
ממשק בשפת Verilog מוגדר ע"י המילה **module** ולאחריה שם היחידה ובתוך סוגריים רשימת שמות הכניסות והיציאות. לאחר הסוגריים יופיעו שוב שמות הכניסות והיציאות והגדרת סוגן.
דוגמא לשימוש כתיבת הממשק של שער and :

```
module AndGate (a,b,z); //simple And Gate
input a,b;
output z;
```

נשים לב מספר נקודות :

- המשפטים הנ"ל מסתיים ב "; " (נקודה פסיק)
- הערות ב- Verilog מתבצעות על ידי "// " (שני קווים נטויים) או /* comment */.
- מימדי הכניסות/יציאות הוא ביט אחד כיוון שלא צוין אחרת

פרמטרים למודול

ניתן להגדיר למודול פרמטרים שיקבעו בזמן אינסטנציאציה ע"י משפט parameter. לדוגמא המשפט : `parameter MyParam = 0` , יקבע פרמטר למודול בשם MyParam אשר ערך ברירת המחדל שלו הוא 0.

תיאור פונקציונלי

אינסטנסיאציה – instantiation (הצבה)

אינסטנסיאציה היא הצבת עותק של מודול קיים וחיבורו עם הלוגיקה הנוספת במודול. לדוגמא Full Adder בנוי משני עותקים של Half Adder המחוברים ביניהם.
אינסטנסיאציה ב- Verilog נעשית ע"י כתיבת שם העותק ושם הרכיב ולאחר מכן קישור היציאות והכניסות של העותק עם הלוגיקה הנוספת של היחידה שמכילה אותו. לדוגמא :

```
nand U1 (.out(w), .in1(p), .in2(q));
```

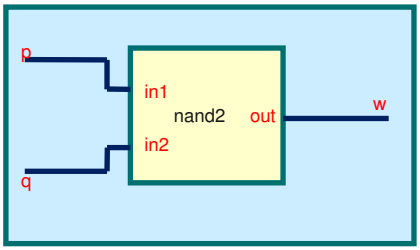
המשפט הנ"ל יוצר עותק של יחידה מסוג nand בשם U1 ומחבר את הכניסות והיציאות שלה (out,in1,in2) עם חיבורים של היחידה המכילה אותו (w,p,q)
אם הגדרנו פרמטר למודול, ניתן להציב לתוכו ערך ע"י הסימן #, המשפט הבא, לדוגמא, יצור עותק של nand כמקודם אך יקבע את ערך הפרמטר latency לערך 5.

```
nand #(.latency(5)) U1 (.out(w), .in1(p), .in2(q));
```

ניתן שלא לציין את שמות החיבורים של ה- nand באופן הבא :

```
nand U1 (w,p,q);
```

במקרה זה החיבור יתבצע לפי סדר רישום האותות, כלומר w יחובר לאות הראשון בהגדרה של ה- nand וכן הלאה.



משפטי always

משפט always נועדו לתיאור התנהגותי של לוגיקה סינכרונית וקומבינטורית. למשפט זה, רשימה של אותות (משתנים) הנקראת רשימת רגישויות (sensitivity list). כאשר משתנה המופיע ברשימת הרגישויות משתנה, משפט ה-always מתבצע. עם סיום הביצוע, הוא ממתין לשינוי נוסף במשתנה המופיע ברשימת הרגישויות ואז משפט ה- always מתבצע מחדש. כאשר אין רשימת רגישויות, משפט ה- always מתבצע שוב מייד בסיום הביצוע הקודם. במקרה זה, חייב להופיע משפט עם השהייה (כגון : 3 = a 10 #), אחרת משפט ה- always יתבצע ללא הפסקה בלולאה אין סופית.

משפטים מסוג if/else, משפטי case למיניהם, משפטי for חייבים להופיע בתוך משפט always (או initial – ראה המשך). ניתן לראות את המבנה של משפט if בדוגמא הבאה ותיאור של משפט case ו- for יובא בהמשך.

מומלץ להפריד משפטי always למשפטים קומבינטוריים וסינכרוניים. דוגמא למשפט always קומבינטורי. האותות a, b ו- sel מהווים את רשימת הרגישויות :

```
always  @(a or b or sel)
begin
   if (sel == 0) begin
      y = a;
  end
  else begin
     y = b;
  end
end
```

ממש MUX קומבינטורי.

דוגמא למשפט always סינכרוני. משפט זה מופעל עם **<u>עליית</u>** השעון :

```
always  @(posedge clk)
begin
  if (reset == 0) begin
     y <= 0;
  end
  else if (en == 1) begin
     y <= a;
  end
end
```

משפט זה יוצר flipflop עם אות reset ואות enable (en).

משפטי השמה

בשפת Verilog קיימים סוגים שונים של משפטי השמה וחשוב להבין בדיוק כיצד כל סוג מתנהג.
באופן כללי התחביר להשמת ערכים לוגיים קבועים הוא באופן הבא: x = <n>'<t><v>
כאשר <n> מסמן את כמות הביטים, <t> את אופן הקידוד של <v> בינארי, דצימלי, אוקטלי, או הקסדצימלי, ו- <v> הוא הערך עצמו לדוגמה

A=1'b0; A – a single bit variable is assigned a one bit zero logic
B=4'b1101; B – a 4 bit variable is assigned the binary value 1101
C=32'd17; C – a 32 bit variable is assigned the decimal value 17 = 00010001
D=8'h3F; D – an 8 bit variable is assigned the hexa' value 0x3F = 00111111

השמה מסוג continuous

השמה קומבינטורית למשתנה נעשית ע"י משפט assign כאשר צד ימין מוצב לתוך צד שמאל באופן מתמיד, לדוגמא:

```
assign y = x ;
```

יציב את הערך של x לתוך y באופן מיידי. דוגמא נוספת:

```
assign z  = (a >= b) ? 1 : 0 ;
```

אם התנאי מתקיים (במקרה זה a גדול או שווה ל- b) z יהיה 1 אחרת 0. משפט ה- assign יכול להופיע גם בתוך משפט always. חשוב לציין שגם בתוך always ההצבה תהיה מיידית וללא קשר ל- sensitivity list ולכן אין סיבה לרשום משפט assign במשפט always.

השמה סדרתית (procedural או sequential)

השמה סדרתית מתבצעת במשפט always (או initial).

חשוב: במשפט מסוג always – השמות יכולות להתבצע למשתנה מסוג reg או integer בלבד.

במשפט זה ישנן 2 סוגי השמות:

- השמה המתבצעת ע"י = תתבצע בצורה טורית ונקראת Blocking assignment
עבור האברים שבצד ימין של ההשמה, ההשמה משתמשת בערכם הנכון לאותה שורה. האבר שמצד שמאל של ההשמה מעדכן את ערכו באותה השורה.

- השמה המתבצעת ע"י => תתבצע בצורה מקבילית ונקראת Nonblocking assignment.
עבור האברים שבצד ימין של ההשמה, ההשמה משתמשת בערכם כפי שהיה בעת הכניסה למשפט always . האבר שמצד שמאל של ההשמה מעדכן את ערכו רק בסוף משפט ה-always או בעת עצירתו ע"י #.

דוגמא:

```
always  @(…..)
 begin
       …..
       y = a + b;
       x = y;
       …..
 end
```

x יקבל אותו ערך כמו y.

```
always  @(….)
 begin
      …..
       y < = a + b;
       x < = y;
      ….
```

```
end
```

x יקבל את הערך **<u>הקודם</u>** של y (הערך שהיה ל-y בעת הכניסה למשפט always).

Functions ו- Tasks

ניתן להפריד קוד שחוזר על עצמו לפונקציות ולחסוך שכפול קוד מיותר. פונקציות באות לתאר חומרה קומבינטורית ולכן אסור שיכילו השהיות. יוצרים פונקציה ע"י המילה השמורה function. פונקציות מוגדרות רק במודול בהן הן נכתבות ועל מנת להשתמש בהן במודול אחר יש להשתמש במילה 'include. לדוגמא, נניח שהקובץ myfunction. v מכיל:

```
function <type_returned> myfunction;
  input a, b, c, d;
  begin
    myfunction = ((a+b) + (c-d));
  end
endfunction
```

אופציה חלופית לשתי השורות הראשונות היא:

```
function <type_returned> myfunction(input a,b,c,d);
```

דוגמא לקריאה לפונקציה:

```
module  calling _function (a, b, c, d, e, f);
  input a, b, c, d, e ;
  output f;
  wire f;
  `include "myfunction.v"
   assign f =  (myfunction (a,b,c,d)) ? e :0;
endmodule
```

ניתן גם ליצור סברוטינות (תת-שיגרה) שכל ההבדל בינה לבין פונקציה הוא שסברוטינה אינה מחזירה ערך אלא מבצעת סט של פקודות . משפט זה נועד לפשט את הקוד ולמנוע שכפול מיותר של משפטים. יוצרים סברוטינה ע"י המילה השמורה task לדוגמא נניח שהקובץ mytask.v מכיל:

```
task convert;
   input [7:0] temp_in;
   output [7:0] temp_out;
   begin
     temp_out = (9/5) *( temp_in + 32) ;
   end
endtask
```

ודוגמא לקריאה ל- task :

```
module  calling_task (temp_a, temp_b, temp_c, temp_d);
  input [7:0] temp_a, temp_c;
  output [7:0] temp_b, temp_d;
  reg [7:0] temp_b, temp_d;
  `include "mytask.v"

  always @ (temp_a)
  begin
    convert (temp_a, temp_b);
  end

  always @ (temp_c)
  begin
    convert (temp_c, temp_d);
  end
endmodule
```

הערות :

- מותר ל- task "לצרוך" זמן, פונקציה מתבצעת באותה יחידה זמן.
- Task חייב להופיע במשפט always או initial (ראה הסבר בהמשך).
- פונקציה מחזירה ערך אחד. אם צריך לשנות מספר משתנים, יש להשתמש ב- task.
- ניתן לקרוא פונקציה מ- task אבל לא task מפונקציה.
- בהגדרה של פונקציה חייבת להיות השמה למשתנה בשם הפונקציה שזה בעצם הערך שהפונקציה מחזירה.
- לפונקציה רק ארגומנטים מסוג input וחייב להיות לפחות אחד. ל- task מתור ארגומנטים מכל הסוגים.

לולאות - loops

רוב הלולאות אינן סינתזביליות ולכן נועדו ל– testbench בלבד. כל הדוגמאות מובאות כאן יגרמו להדפסת כל המספרים מ- 0 ועד 9.

הערה : שים לב שניתן להשתמש במפשט for על מנת לרשום קוד סינתזבילי.

while – לולאה אשר בודקת את התנאי לקיום בתחילת הלולאה, לדוגמא :

```
 integer a  = 0;
 while  (a < 10) begin
    $display ("Current value of a = %0g", a);
    a = a + 1;
 end
```

do while – לולאה המתבצעת פעם אחת לפחות והתנאי לביצוע בשנית נבדק בסוף, לדוגמא :

```
 integer a  = 0;
 do begin
    $display ("Current value of a = %0g", a);
    a = a + 1;
 end while  (a < 10);
```

for – לולאה המתבצעת בעזרת משתנה עזר הסופר את מספר האיטרציות הרצוי, לדוגמא :

```
integer a  = 0;
for (a = 0 ; a < 10; a = a + 1) begin
    $display ("Current value of a = %0g", a);
end
```

repeat – לולאה המתבצעת כמספר האיטרציות הרצוי, לדוגמא :

```
integer a  = 0;
repeat(10) begin
   $display ("Current value of a = %0g", a);
a = a + 1;
end
```

הערה : יש להגדיר את משתנה הלולאה לפני השימוש בלולאה.

fork-join

משפט ה- fork גורם לפיצול התהליך, וכתוצאה מכך כל משפט/בלוק המתבצע לאחר משפט fork ולפני משפט join יתבצע במקביל . ישנם 3 סוגי משפטי join :

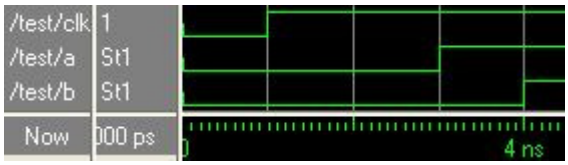
- join [all] – מחכה שכל התהליכים יסתיימו ואז ממשיך לבצע את התוכנית
- join_none – לא מחכה אלא ממשיך בהרצה
- join_any – מחכה לתהליך כלשהו שיסתיים ואז ממשיך בהרצה

משפטי fork-join מתאימים מאד לבדיקת bus או לרכיבים המחוברים לבורר (arbiter) כיוון שכל בלוק בתוך משפט ה- fork-join מתאר יחידה המשתמשת ב- bus וכל היחידות פועלות במקביל.

דוגמא להבדל בין fork-join לבין ביצוע רגיל :

ביצוע רגיל :

```
initial begin
    a = 0;b = 0;
  end
always @(posedge clk)
  begin
    #2 a = 1;
    #1 b = 1;
  end
```



ביצוע בעזרת fork-join :

```
initial begin
    a = 0;b = 0;
  end
always @(posedge clk)
  fork
    #2 a = 1;
    #1 b = 1;
  join  //join_any, join_none
```

חשוב : join ממתין שכל המשפטים אחרי fork יתבצעו (עד t=2) לפני שהוא ממשיך בביצוע. join_any ממתין שאחד המשפטים יתבצע ולכן ממשיך ב- t=1 ו-join_none לא ממתין ומיד ממשיך בביצוע ב- t=0.

הערה : בכל נושא של מקביליות – יש צורך בסנכרון ולכן אם שני תתי-תהליכים משתמשים באותם משתנים או מעוניינים להעביר ביניהם מידע – יש להשתמש ב- semaphores וב-mailbox בהתאמה אולם מסמך זה אינו מכסה נושאים אלו. מידע נוסף ניתן למצוא בלינק הבא :
http://www.asic-world.com/systemverilog/sema_mail_events.html

משפט generate

כאשר לחומרה מבנה רגולרי, ניתן להשתמש במשפט generate על מנת לקצר את הקוד :

```
module gray2bin1 (bin, gray);
```

```
  parameter SIZE = 8;
  output [SIZE-1:0] bin;
  input [SIZE-1:0] gray;
  genvar i; // Compile-time only
  generate for (i=0; i<SIZE; i=i+1)
  begin
    assign bin[i] = ^gray[SIZE-1:i]; //bitwise XOR on all bits in the range [SIZE-1 : i]
  end
 endgenerate
endmodule
```

משפט generate הוא למעשה macro ויכול להופיע רק מחוץ למשפט. משפט ה- generate משכפל את כל הקוד שמופיע בין generate ל- endgenerate.

הערה: יש להגדיר את המשתנה i מסוג genvar. משתנה זה קיים רק בזמן קומפילציה.

עד כה, ניתן תיאור מצומצם של מימוש חומרה, הסברים נוספים יובאו בפרק שמסביר על SystemVerilog.

כתיבת testbench

לאחר מימוש מודול צריך גם לבדוק אותו באמצעות סימולציות. לשם כך יש להכין סביבה פשוטה המכונה testbench. הסביבה היא למעשה מודול שמכיל הצבה של המעגל הנבדק בתוספת קוד שיוצר ערכים שונים לכניסות. מקובל לממש את הקוד באמצאות משפטי always ו/או initial לפי נוחיות המשתמש. בסעיף זה, נתאר גם פקודות הדפסה שלרוב גם מופיעות ב- testbench.

משפט initial

בניגוד למשפט always שמתבצע שוב ושוב, משפט initial מתבצע רק פעם אחת בתחילת הסימולציה. בעזרת משפט זה ניתן לקבוע ערכים התחלתיים למשתנים (לשימוש ב- testbench בלבד כיוון שמשפט זה אינו סינתזבילי) . לדוגמא, בעזרת משפט always מחליפים את הערך של אות השעון כל מחזור ובמשפט initial קובעים את הערך ההתחלתי:

```
initial
begin
   clk = 0;
end
```

```
always
begin
   #5  clock =  ~ clock;
end
```

ניתן לבצע בלוקי initial במקביל ורצוי שלפחות אחד הבלוקים יכיל את המילה $finish אשר גורמת לסיום הסימולציה.

הדפסות למסך

ניתן להדפיס למסך טקסט וערכי משתנים במהלך בלוק initial, על מנת לעזור בדיבוג. הודעה המודפסת באופן חד פעמי מתבצעת ע"י משפט:

```
$display ("sentence",variables)
```

הודעה המודפסת בכל שינוי של אחד המשתנים מתבצעת ע"י משפט:

```
$monitor ("sentence",variables)
```

הערכים שניתן להדפיס: %d – זמן. מקבלים את הזמן הנוכחי ע"י $time. %b – ביט , %h – ערך בהקס דצימלי, %d – integer

דוגמא:

```
initial  begin
  $display("\ttime,\tclk,\treset,\tenable,\tcount");
  $monitor("%d,\t%b,\t%b,\t%b,\t%d",$time, clk,reset,enable,count);
end
```

המשפט הראשון יתבצע פעם אחת בלבד וייצור כותרת. המשפט השני יתבצע בכל פעם שאחד המשתנים ישנה את ערכו (\t הוא טאב, \n הוא תו שורה חדשה). הרצה של הסימולציה תדפיס את השורות הבאות למסך:

```
time    clk,    reset,    enable, count
   0,    0,       0,       0,       x
   5,    1,       0,       0,       x
  10,    0,       0,       0,       x
  15,    1,       0,       0,       x
  20,    0,       0,       0,       x
  25,    1,       0,       0,       x
```

וכו'

דוגמא של מודול וה- testbench

```
module arbiter (clock, reset, req_0,req_1, gnt_0,gnt_1);

input clock, reset, req_0, req_1;
output gnt_0, gnt_1;

reg gnt_0, gnt_1;
```

נדחף ע"י משפט always ולכן חייב להיות reg

```
always @ (posedge clock or posedge reset)
  if (reset) begin
  gnt_0 <= 0;
  gnt_1 <= 0;
 end else if (req_0) begin
   gnt_0 <= 1;
   gnt_1 <= 0;
 end else if (req_1) begin
   gnt_0 <= 0;
   gnt_1 <= 1;
 end
endmodule
```

משפט זה יוצר flip flop ביציאת המעגל ולכן ברשימת הרגישויות יש להכניס רק את הכניסות שמשפיעות עליו.

להלן מימוש רכיב הבדיקה של הבורר :

```
module arbiter_test;

 reg clock, reset, req0,req1;
 wire gnt0,gnt1;
```

מדפיס את ערכי המשתנים ואת הזמן בכל שינוי :

```
 initial begin
  $monitor("req0=%b,req1=%b,gnt0=%b,gnt1=%b,time=%t",req0,req1,gnt0,gnt1,$time);
   clock = 0;
   reset = 0;
```

קביעת ערכים התחלתיים

```
  req0 = 0;
  req1 = 0;
   #5  reset = 1;
   #15  reset = 0;
   #10  req0 = 1;
   #10  req0 = 0
   #10  {req0,req1} = 2'b11;
   #10  {req0,req1} = 2'b00;
   #10  $finish;
 end

always begin
   #5  clock =  ~ clock;
end
```

יצירת שעון :

אינסטסיאציה של הבורר :

```
arbiter U1 (.clock (clock),.reset (reset),.req_0 (req0),.req_1 (req1),.gnt_0 (gnt0),.gnt_1 (gnt1));

endmodule
```

הרצת הסימולציה תדפיס את הערכים הבאים :

```
req0=0,req1=0,gnt0=x,gnt1=x,time=            0
req0=0,req1=0,gnt0=0,gnt1=0,time=            5
req0=1,req1=0,gnt0=0,gnt1=0,time=            30
req0=1,req1=0,gnt0=1,gnt1=0,time=            35
req0=0,req1=0,gnt0=1,gnt1=0,time=            40
req0=0,req1=1,gnt0=1,gnt1=0,time=            50
req0=0,req1=1,gnt0=0,gnt1=1,time=            55
req0=0,req1=0,gnt0=0,gnt1=1,time=            60
req0=1,req1=1,gnt0=0,gnt1=1,time=            70
req0=1,req1=1,gnt0=1,gnt1=0,time=            75
req0=0,req1=0,gnt0=1,gnt1=0,time=            80
$finish at simulation time          90
```

במהלך הניסוי בד"כ נסתכל על צורת גל ולא על הדפסות ערכי היציאות.

התוספות של SystemVerilog לשפת Verilog

SystemVerilog היא הרחבה משמעותית ל- Verilog ואין כל אפשרות להסביר את כל השינויים. במהלך הניסוי נתמקד על החלק מה- features החדשים שנמצאים בשימוש רחב.

סוגי משתנים חדשים

SystemVerilog מגדירה 2 סוגים של משתנים לוגים :

- משתנים המקבלים 2 ערכים ('0','1') : shortint,int,longint,byte,bit
- משתנים המקבלים 4 ערכים ('0','1','Z','X'): logic,time

כברירת מחדל משתנים מהסוג הראשון הם signed ומשתנים מהסוג השני הם unsigned. שפת systemverilog מאפשרת לקבוע שהסוג יהיה שונה מברירת המחדל וכל סוג משתנה יכול להיות signed או unsigned למשל :

```
int unsigned j;
```

אפשר גם להשתמש ב- casting. לדוגמא : 'signed(j).

ניתן להשתמש ב- logic גם עבור reg וגם עבור wire, ולכן במהלך הניסוי נשתמש רק ב- logic במקום שני הטיפוסים האחרים.

typedef

כמו בשפות תכנות אחרות ניתן לתת שם אחר לסוג משתנה ע"י משפט typedef, לדוגמא :

typedef integer myinteger;

ייצור סוג משתנה חדש בשם myinteger שהוא מסוג integer. עיקר השימוש של מילה זו הוא ביצירת סוג משתנה חדש עבור משפטי enum.

enum

משפט מסוג enum ייצור משתנה אנומרציה חדש (משתנה שניתן לספור את איבריו), לדוגמא :

enum {RED,GREEN,ORANGE} color1;

ייצור משתנה בשם color אשר מקבל את הערכים RED ו GREEN ORANGE

על מנת ליצור סוג משתנה חדש נשתמש במשפט מסוג typedef enum, לדוגמא :

typedef enum { red, green, blue, yellow, white, black } Colors;

ייצור סוג משתנה מסוג Colors. על מנת ליצור משתנה מסוג זה נכתוב Colors color2.

struct

בעזרת משפט struct ניתן לאחד מספר משתנים יחדיו, לדוגמא :

struct {byte a; logic b; shortint unsigned c; } myStructVar

ייצור משתנה בשם myStructVar המכיל את האברים a,b,c

typedef struct { a; b; shortint unsigned c; } myStructType;

ייצור סוג משתנה בשם myStructType המכיל את האברים a,b,c

גישה לאברים של struct מתבצעת ע"י אופרטור הנקודה, לדוגמא myStructVar.a

class

Class הוא מבנה הלקוח מתוך עולם ה- object oriented programming המאפשר איחוד של משתנים ופונקציות הפועלות על משתנים אלו (הנקראות מתודות).

מבנה של הגדרת class לדוגמא :

```
class Packet;
  int address;
  bit [63:0] data;
  shortint crc;
endclass:Packet
```

להסבר מקיף יותר על מבנה זה יש לגשת לספר המסביר תכנות מונחה עצמים.

Package

ניתן להגדיר פונקציות ו- tasks בתוך package ולקרוא ל- package מתוך המודול :

```
package PkgName;
  function …
   …..
  endfunction

endpackage : PkgName
```

במודול רושמים :

```
import PkgName::*;
```

אינסטנסיאציה בשפת SystemVerilog

ב- SystemVerilog ניתן לבצע אינסטנציאציה בדיוק כפי שהוסבר עבור verilog. מעבר לכך, אם במודול שבו מבצעים את האינסטנציאציה מופיעים סיגנלים בשמות זהים לכניסות וליציאות של הרכיב (במקרה הזה nand) אז ניתן לבצע אינסטנציאציה מבלי לציין אף שם. לדוגמא :

```
nand U1 (.*) ;
nand U2(.* , .Z(C) );
```

ה "*" מציין ש- pins של ה- nand יחובר לרשתות (חוטים) בעלי שם זהה.

מערכים

ישנם שני סוגים של מערכים ב- SystemVerilog על מנת לתמוך באפשרות של מערכים רב מימדיים :

1. packed – מערכים של ביט בודד בלבד והם מתארים וקטור (ב- VHDL זהו std_logic_vector). יוצרים סוג זה ע"י סוגריים מרובעות המכילות את הטווח שלו לפני שם המשתנה.
2. unpacked – מערך של משתנה אחר. מתאר מערך כפי שאנו מכירים משפת C. יוצרים סוג זה ע"י סוגריים מרובעות המכילות את הטווח שלו אחרי שם המשתנה.

הערה : למען לדיוק, מערכים אלה קיימים גם בשפת verilog אבל ההסבר מובא עקב שימוש במונחים וטיפוסים של שפת systemverilog.

דוגמא :

```
// packed array
bit [7:0] packed_array = 8'hAA;
```

יוצר וקטור של ביטים

```
// unpacked array
bit unpacked_array [7:0] = '{0,0,0,0,0,0,0,1};
```

יוצר מערך של ביטים

```
 $display ("packed array = %b", packed_array);
```

נשים לב שהפקודה הבאה יכולה להדפיס את השורה הבאה :

```
packed array  = 10101010
```

אולם הפקודה הבאה לא תתקמפל כיוון שזהו אינו וקטור :

```
$display("unpacked array = %b",unpacked_array);
```

ולכן על מנת להדפיס מערכים שהם unpacked יש ליצור לולאה שתדפיס כל אחד מהאברים בנפרד. דוגמא להדפסת אחד מהאברים :

```
$display ("unpacked array[0] = %b", unpacked_array[0]);
```

על מנת ליצור מערכים רב מימדיים ניתן לשרשר סוגריים מרובעות משני הסוגים, לדוגמא :

```
logic [1:0][3:0] mem0 = ‘8b00010010;
logic [7:0] mem [0:3] = '{8'h0,8'h1,8'h2,8'h3};
logic [7:0] mem1 [0:1] [0:3] = '{'{8'h0,8'h1,8'h2,8'h3},'{8'h4,8'h5,8'h6,8'h7}};
logic [7:0] [0:4] mem2 [0:1] = '{{8'h0,8'h1,8'h2,8'h3},{8'h4,8'h5,8'h6,8'h7}};
```

וניתן אף להגדיר מספר מערכים באותה שורה :

```
bit [7:0] [31:0] mem4 [1:5] [1:10], mem5 [0:255];
```

בשורה האחרונה יווצרו 2 מערכים אשר החלק ה- packed שלהם זהה אך החלק ה- unpacked שלהם שונה.

גישה לאיברי המערך היא למשל mem[0]=8'h0 ,mem2[1][2]=8'h6 ,mem2[1][2][1]=1'b1.

עבור הסיגנל שהוגדר כך

```
logic [2:0][7:0] mem3 [3:0][4:0]
```

וניגשים אליו כך

```
mem3[idx1][ idx2][ idx3][ idx4]
```

idx1 מתיחס לטווח [3:0], idx2 לטווח [4:0], idx3 לטווח [2:0] ו-idx4 לטווח [7:0].

במובן מסויים mem4 המוגדר להלן שקול להגדרה של mem3

```
logic [3:0][4:0][2:0][7:0] mem4
```

ההבדל ניכר בהסבר הקודם לגבי ההדפסה וגם בעובדה שכשם שגם mem0 וגם mem הם מערכים דו מימדיים, רק את mem0 ניתן לאתחל בצורה המוצגת בדוגמה.

בניסוי תדרשו להשתמש במערכים מסוג packed בלבד. כמובן שהטווח האחרון יהיה [7:0] או [15:0] בהאתאם לסיגנל הרלוונטי כפי שהדבר נעשה כאן עבור idx4.

כאמור, המערכים הנ"ל קיימים גם ב- verilog. שפת SV מוסיפה שני סוגים חדשים : **מערכים דינמיים ואסוצייטיבים.** לא מוקצה מקום בזיכרון עבור מערך דינמי עד שמופיע הקוד שיוצר אותו (ולא רק הקוד שמגדיר אותו). בנוסף לכך, ניתן לשנות את גודלו תוך כדי עבודה. עבור מערך אסוציאטיבי מקום בזיכרון מוקצה עבור כל איבר בנפרד בזמן שהאיבר מקבל ערך. מוגדרות מספר מתודות (לדוגמא size(), new(), delete() ועוד) למערכים דינמיים ואסוצייטיבים אשר פועלות על איברי המערכים.

קיימים סוגי משתנים אחרים כגון : queue, event, string שלא הורחב עליהם אך ניתן למצוא הסבר עליהם באתרים שהוצגו בהקדמה.

משפטי always

בשפת verilog – כאשר לא מוגדרת התנהגות היחידה עבור כל התנאים במשפטי if ו- case, יווצר latch לא רצוי בשלה הסינתזה. הסיבה לכך היא שאם הכלי אינו יודע מה לעשות עבור תנאי מסוים, במקום להציב ערך חדש כלשהו למשתנה הוא יציב בו את הערך הקודם שלו, או לחלופין, ישמור על מצבו הקודם והדבר יעשה בעזרת latch. בד"כ, זאת לא כוונת המתכנן ולכן יש למנוע זאת ע"י הגדרה שלמה של כל התנאים. שפת SystemVerilog מרחיבה את משפט ה- always למספר סוגים שונים על מנת למנוע ספק לגבי כוונת המתכנן בשלב הסינתזה.

always_comb

משפט זה מציין שכוונת המתכנן היא שהמשפטים בבלוק יתורגמו ללוגיקה צירופית בלבד. לכן אין צורך ברשימת רגישויות מפורשת. דוגמא – מחבר קומבינטורי תקין :

```
always_comb
begin : ADDER
  sum = b + a;
end
```

נשים לב לעוד תכונה מיוחדת ל- SystemVerilog : לבלוק הנ״ל יש שם (ADDER) שמוסיף לקריאות כאשר שמים בלוק בתוך בלוק. אם לא מוגדרים כל המקרים לתנאי ה-if ו/או case תתקבל הודעת שגיאה שתכיל את שם הבלוק.

always_latch

משפט זה מציין שכוונת המתכנן היא שהמשפטים בבלוק יתורגמו ללוגיקה אשר ביציאתה latch. גם כאן אין צורך ברשימת רגישויות.

דוגמא – מחבר קומבינטורי עם כניסת enable :

```
always_latch
begin : ADDER
  if (enable) begin
    sum    <= b + a;
    parity <= ^(b + a);
  end
end
```

always_ff

משפט מציין שכוונת המתכנן היא שהמשפטים בבלוק יתורגמו ללוגיקה סינכרונית (כלומר, כל היציאות מתקבלות דרך flip-flop). התהליך מתבצע בעליית/ירידת השעון. דוגמא : מחבר סינכרוני עם כניסת reset :

```
always_ff @(posedge clk iff rst == 0 or posedge rst)
begin : ADDER
  if (rst) begin
    sum    <= 0;
    parity <= 0;
    $display ("Reset is asserted BLOCK 1");
  end else begin
    sum    <= b + a;
    parity <= ^(b + a);
  end
end
```

נשים לב שבתוך הסוגריים מופיע הביטוי 0 == iff rst אשר קובע שהכניסה לבלוק אך ורק בעליית שעון וכאשר rst הוא '0'. ללא ביטוי זה – הבלוק יתבצע מחדש בכל עליית שעון ללא קשר לערך הסיגנל rst. מכיוון שמופיע גם posedge rst התהליך יתבצע גם כאשר rst עולה ללא קשר למצב השעון, משמעות הדבר היא שפעולת ה- rst היא א-סינכרונית. ללא posedge rst, פעולת ה- rst תתבצע רק עם עלית השעון.

unique ו- priority

מילים שמורות אלו גורמות למשפטי if ו- case להתנהג בצורה מסוימת :

unique – מציין שבשרשור if ובמשפטי case אין חפיפה בין התנאים ולכן ניתן לבדוק אותם במקביל. במידה ובזמן כלשהו שני תנאים מתקבלים – תתקבל הודעת שגיאה.

priority – מציין שבשרשור משפטי if – התנאים יבדקו לפי הסדר ובמשפט case – התנאי הראשון שמתקיים יתקבל ולכן יש לבדוק את התנאים בטור.

בשני המקרים תתקבל הודעת שגיאה אם אף אחד מהתנאים לא יתקיים. unique ו- priority גורמים להתנהגות לוגית שונה ולכן המעגל המסונתז יהיה שונה עבור שני המקרים.

דוגמא לשימוש :

```
always @ (*)
begin
 unique case(a)
```

```
      0, 1: y = in1;
      2: y = in2;
      4: y = in3;
 endcase //values 3,5,6,7 will cause a warning

 priority casez(a)
   2'b00?: y = in1; // a is 0 or 1
   2'b0??: y = in2; //a is 2 or 3;
   default: y = in3; //a is any other value
 endcase
end
```

במידה ו-a יהיה 2'b000 אז יתבצע המשפט הראשון ולא השני, למרות ששני התנאים ראשונים מתקיימים. שים לב שבמקרה השני שבו מופיע casez (במקום case), ניתן להשתמש ב- "?" כדי לסמן "don't care".

דוגמאות נוספות:

```
// error if none of irq0–irq2 is true
priority case (1'b1)
   irq0: irq = 3'b1 << 0;
   irq1: irq = 3'b1 << 1;
   irq2: irq = 3'b1 << 2;
endcase
// error if none of irq0–irq2 is true
priority if (irq0) irq = 3'b1;
   else if (irq1) irq = 3'b2;
   else if (irq2) irq = 3'b4;
// Default or else ignores priority
// This never raises an error:
priority if (irq0) irq = 3'b1;
   else irq = 3'b0;
// Nor does this:
priority case (1'b1)
   irq0: irq = 3'b1 << 0;
   default: irq = 0;
endcase
```

```
// Error if not exactly one of irq0–irq2 is true
unique case (1'b1)
   irq0: irq = 3'b1 << 0;
   irq1: irq = 3'b1 << 1;
   irq2: irq = 3'b1 << 2;
endcase
// Error if not exactly one of irq0–irq2 is true
unique if (irq0) irq = 3'b1;
   else if (irq1) irq = 3'b2;
   else if (irq2) irq = 3'b4;
// Error if both irq0 and irq1 are true
unique if (irq0) irq = 3'b1;
   else if (irq1) irq = 3'b2;
   else irq = 3'b0;
// Error if both irq0 and irq1 are true:
unique case (1'b1)
   irq0: irq = 3'b1 << 0;
   irq1: irq = 3'b1 << 1;
   default: irq = 0;
endcase
```

לולאות - loops

למרות שלולאות קיימות גם בשפת Verilog, שפת SystemVerilog הוסיפה מספר סוגים חדשים.

```
always_ff @(posedge clk) begin : label_count   // SV allows named begin-end blocks
  for (int i = 0; i <16; i++)  // automatic variable int, i visible only within for loop
  count = count + incr[i]
end : label_count
```

ניתן להוסיף שם ללולאה ואין צורך בהגדרת משתנה הלולאה לפני כן.

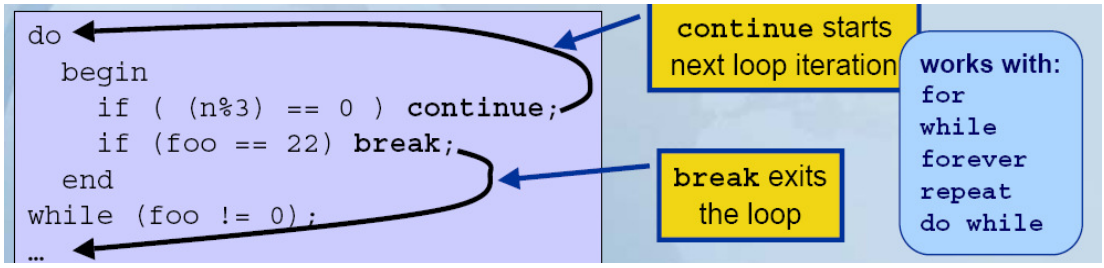
foreach – לולאה העוברת על כל אחד מהאברים של משתנה כלשהו, לדוגמא:

```
//foreach loop construct
logic [15:0] parity;
logic [15:0] [7:0] data;
foreach (parity[i]) parity[i] = ^data[i]; // generates parity for each data byte
```

דוגמא נוספת :

```
byte a [10] = '{0,1,2,3,4,5,6,7,8,9};
foreach (a[i]) begin
  $display ("Value of a is %0g",i);
end
```

כמו בשפת C – גם בלולאות ב- SystemVerilog ניתן לבצע את הפקודות continue אשר גורמת למעבר לאיטרציה הבאה ו- break אשר מפסיקה את ביצוע הלולאה.



משפט final

שפת systemverilog תומך במשפט final אשר מתבצע פעם אחת בלבד ובסוף הסימולציה. כיוון שהוא מתבצע לאחר תום הסימולצייה – הוא לא יכול להכיל השהיות וכל מטרתו היא להדפיס את התוצאות הסופיות של הסימולצייה, דוגמא לשימוש – משפט שיודיע מתי נגמרת הסימולצייה :

```
final begin
  $display ("Final block called at time %0g", $time);
end
```

שליטה באירועים (event)program block

כפי שהוסבר , module הוא בלוק אשר מתאר חומרה ולכן הוא מכיל בין היתר משפטי always, ו- assign. שפת SystemVerilog מוסיפה בלוק מסוג חדש, program block אשר נועד ליצירת testbench. יחד עם זאת, עדיין ניתן, ובהחלט מקובל, לממש testbench בעזרת מודולים בלבד. בלוק progam אינו יכול להכיל משפטי always או הצבה של מודול, ונועד בין היתר לשם הפרדה ברורה בין קוד המיועד לסינתזה וקוד המיועד לבדיקה. לכן משפטים כגון join-fork, לולאות, coverage אשר אינם סינתזבילים יופיעו בבלוק program. אסור למודול להשתמש ב- function או task שמוגדרים ב- program block, אבל ל- program block מותר להשתמש ב- function או task שמוגדרים ב- module. הסיבה עיקרית של העדפת program block על פני מודול למימוש testbench היא היכולת שלו למנוע מירוצים בין המעגל הנבדק ל- testbench (אם קיימים כאלה במערכת). הסבר מפורט ניתן למצוא ב :

www.project-veripage.com/program_blocks_1.php

דוגמא של שימוש ב- program block :

```
// Simple Program with ports
program simple(input wire clk,output logic
  reset,enable, input logic [3:0] count);
  initial
  begin
  $monitor("@%0dns count =
%0d",$time,count);
  reset = 1;
  enable = 0;
  #20 reset = 0;
  @ (posedge clk);
 enable = 1;
 repeat (5) @ (posedge clk);
 enable = 0;
 // Call task in module
 simple_program.do_it();
 end
 task do_it();
   $display("%m I am inside program");
 endtask
endprogram
```

```
module simple_program();
  logic clk  = 0;
  always #1 clk ++;
  logic [3:0] count;
  logic reset,enable;
  always @ (posedge clk)
  if (reset) count <= 0;
    else if (enable) count++;
  //Program is connected like a
module
  simple
prg_simple(clk,reset,enable,count);
  //Task inside a module//
  task do_it();
   $display("%m I am inside
module");
  endtask
  //Below code is illegal
  //initial begin
  //prg_simple.do_it();
  //end
endmodule
```

במהלך הניסוי אנו נשתמש בעיקר במודול ליצירת testbench.

randomization

לעתים קרובות רצוי להריץ סימולציות עם כניסות אקראיות. לשם כך יש מספר מנגנונים ב-SystemVerilog.

משתנים אקראיים

משתנים אקראיים מוגדרים ע"י מילות המפתח rand ו- randc. rand מגדיר משתנה המפולג באחידות על כל הטווח אותו הוא יכול לקבל. randc מגדיר משתנה אקראי ציקלי אשר יקבל ערכים אקראיים על כל הטווח אולם לא יתקבלו אותם שני ערכים עד שהמשתנה לא יקבל את כל הערכים האפשריים במהלך ההרצה. משתנים אקראיים יכולים להיות שייכים ל-class בלבד והם מקבלים ערך חדש אקראי בכל פעם שנקרא למתודה randomize() של המחלקה שבו נמצא המשתנה. randomize() היא מתודה מוגדרת מראש לכל class וכשהיא נקראת, היא מגרילה ערך לכל משתנה מסוג rand או randc של ה- class.

דוגמא :

```
logic [7:0] x;
class ForRand;
  rand logic [7:0] len;
endclass
ForRand RandVar = new();
if (RandVar.randomize() != 0)  x = RandVar.len ;
```

אילוצים על משתנים אקראיים

ניתן לאלץ משתנים אקראיים לקבל ערכים מסוימים בלבד ע״י בלוק של constraint. לכל משתנה יש בלוק משלו ולכל בלוק יש שם משלו.
סוגי האילוצים האפשריים:

- ניתן להגדיר שהמשתנה יקבל ערך קטן מ... / גדול מ... ערך מסוים
- ניתן להגדיר טווח ערכים אפשריים
- ניתן להגדיר רשימת ערכים ממנה יבחר ערך אחד למשתנה
- ניתן להגדיר רשימת ערכים שממנה אסור יהיה לבחור ערכים

דוגמא לשימוש:

```
class ForRand;
 rand integer len;
 constraint legal {
     len >= 2;
     len inside { [8'h0:8'hA] };
     len inside { 8'h0,8'h14,8'h18 };
     !(len inside { 8'h20,8'h54 });
   }
}
```

סוג ראשון
סוג שני
סוג שלישי
סוג רביעי

הערה: legal הוא שם הבלוק של האילוץ.

משקולות

ניתן לתת למשתנה אקראי מספר אופציות לקבלת ערך אפשרי ולהגדיר משקולות לכל ערך, כלומר – ככל שהמשקולת בעלת ערך גבוה יותר – כך יש יותר סיכוי שערך זה יתקבל. מגדירים משקולות ע״י מילת המפתח dist והסימן ״:= ״. דוגמא לשימוש:

```
class ForRand;
rand bit [7:0] src_port;
constraint legal {
     src_port dist {
       0  := 1,
       1  := 1,
       2  := 5,
       4  := 1
      };
   }
}
```

דוגמא זו תאפשר למשתנה src_port לקבל את הערכים 0,1,2,4 אולם לקבלת המספר 2 יהיה סיכוי פי 5 מלשאר האפשרויות.

אקראיות מותנת

ניתן להתנות את קביעת הערך של משתנה כלשהו ע״י משפט if או בעזרת האופרטור <=. בלוק ה-if יתבצע אם התנאי שבתוך הסוגריים מתקיים, אחרת יתבצע הקוד שבבלוק ה- else. אופרטור ה- <= יגרום לקוד מימינו להתבצע אם הקוד שמשמאלו מחזיר ערך נכון.
דוגמא לשני סוגי השימוש:

```
typedef enum {RUNT,NORMAL,OVERSIZE} size_t;
class ForRand;
rand bit [15:0] length;
rand size_t size
  constraint frame_sizes {
    size == NORMAL => {
      length dist {
```

```
      [64  :  127 ] := 10,
      [128 :  511 ] := 10,
      [512 :  1500] := 10
     };
   }
   if (size == RUNT)  {
    length >= 0;
    length <= 63;
   } else if (size == OVERSIZE) {
    length >= 1501;
    length <= 5000;
   }
  }
}
```

בדוגמא זו יבחר באקראיות סוג גודל (size) ולפי ערכו האקראי – יבחר גודלו של המשתנה length.

randcase

בתוך בלוק של randcase ניתן להגדיר תת-בלוקים בעלי משקולות אשר כל הרצה של randcase תבצע תת-בלוק אחר לפי ההסתברויות של המשקולות. לדוגמא :

```
randcase
  20 : begin
       $write ("What should I do ? \n");
      end
  20 : begin
       $write ("Should I work ? \n");
      end
  20 : begin
       $write ("Should I watch Movie ? \n");
      end
  40 : begin
       $write ("Should I complete tutorial ? \n");
      end
endcase
```

Assertions

assertions הוא מנגנון שנועד לתיאור ההתנהגות הרצויה (והלא רצויה) של רכיב חומרה. ישנם כלים היודעים להשתמש במשפטים אלו על מנת לבדוק האם הרכיב עובד בצורה הרצויה. ניתן לחלק מנגנון זה לשניים :

- Immediate Assertions (assert) – בדיקה רגעית של קיום של התנאי בזמן סימולציה. משפטים אלה יכולים להופיע רק בתוך למשפטי always, ו- initial.
- Concurrent Assertions (assert property) – בודק את ההתנהגות של תכנון לאורך זמן (כלומר במשך מספר מחזורי שעון) ולא באופן רגעית כמו במקרה הקודם. הבדיקה מתבצעת בסמוך לעליה או ירדת השעון לאורך כל זמן הסימולציה. משפטים אלה יכולים להופיע רק **מחוץ** למשפטי always, ו- initial.

Immediate Assertions

סוג זה מוגדר ע״י שימוש במילה השמורה assert.

```
assert (expression)
    pass_block;
```

```
else
    fail_block
```

אם תוצאת הביטוי בסוגריים היא '1' – יתבצע הבלוק לאחר הפקודה (pass_block) ואם לא, יתבצע הקוד שבבלוק ה- else (fail_block). (ניתן גם לתת שם לכל בלוק ה- assert). הדפסות בשני בלוקים אלו יש לבצע בעזרת אחד מהמשפטים הבאים:

- $fatal – טעות מאד חמורה שלא ניתן לחזור ממנה
- $error – טעות חמורה
- $warning - אזהרה
- $info – מידע למשתמש

ניתן גם להשתמש במשפט $display על מנת להדפיס שלא חלה טעות.

דוגמא לשימוש:

```
always @ (posedge clk)
 begin
  if (grant == 1) begin
    CHECK_REQ_WHEN_GNT : assert (grant && request) begin
      $display ("Seems to be working as expected");
    end else begin
      current_time = $time;
       #1  $error("assert failed at time %0t", current_time);
    end
  end
 end
```

אם מתקיים התנאי grant&&request=='1' אז יתבצע הקוד הזה

אם לא – תודפס הודעת שגיאה ויתבצע קטע הקוד הזה

Concurrent Assertions

כאמור הבדיקה מתבצעת סמוך לשינוי השעון ונקראת concurrent (מקבילי) כיוון שנבדק במהלך כל זמן הסימולציה. למעשה היא בנויה ממספר שכבות וכל שכבה יכולה להשתמש בשכבה מתחתיה (כלומר – ניתן לשרשר שכבות):

- שכבה בוליאנית
- שכבת ה- sequence
- שכבת ה- property
- שכבת ה- property directive

א. שכבה בוליאנית - ביטויים בוליאניים שמחזירים ערך true או false

דוגמא:

```
(en && ce && addr < 100);
```

ב. שכבת ה- sequence – משתמשת בשכבה הבוליאנית כדי לייצר סדרה (sequence) של אירועים כאשר בין כל שכבה מגדירים את הקשר לשכבה שלפניה.

הערה: אם נדרש לציין שהסיגנל (en למשל) עלה, ירד או לא השתנה, ניתן להשתמש ב- $rose(en), $fell(en) או ב- $stable(en).

בעזרת sequence ניתן להגדיר רצף של אירועים. קיימים מספר אופרטורים שמשמשים להגדרת sequence. להלן מספר דוגמאות של אופרטורים שימושיים שבעזרתם הוגדרו מספר sequences:

(num)## מגדיר שהשכבות מופרדות ב- num מחזורי שעון ביניהם. (num יכול להיות גם טווח)

דוגמאות:

```
sequence req_gnt_1clock_seq;
   req[0]  ##1  gnt[0];
endsequence
```

הסיגנל gnt[0] מגיע מחזור שעון אחד לאחר הסיגנל req[0].

```
sequence master_seq;
   seq1 ##1 seq2 ##1 seq3;
endsequence
```

seq3 מגיע יחידת זמן לאחר seq2 שמגיע יחידת זמן לאחר sqe1 וכל השלושה הם sequences.

[*num] – משכפל את השכבה לפניו num פעמים כאשר בין כל שכפול יש רק מחזור שעון אחד. כל אחד מהשכפולים מתקבל בנפרד. לדוגמא שתי השורות הבאות שקולות:

```
req ##1 busy ##1 busy ##1 gnt;
req ##1 busy [*2] ##1 gnt;
```

[->num] – משכפל את השכבה לפניו num פעמים אולם בין שתי שכפולים יכול להופיע כל דבר. לדוגמא שתי השורות הבאות שקולות:

```
req ##1 (( ! busy[*0:$] ##1 busy) [*3]) ##1 gnt;
req ##1 (busy [->3]) ##1 gnt;
```

[=num] – משכפל את השכבה לפניו num פעמים כאשר בין כל שכפול יש רק מחזור שעון אחד. מתקבלת רק תוצאת השכפול. לדוגמא שתי השורות הבאות שקולות:

```
req ##1 (( ! busy[*0:$] ##1 busy) [*3]) ##1  ! busy[*0:$] ##1 gnt;
req ##1 (busy [=3]) ##1 gnt;
```

דוגמאות נוספות:

```
a ##1 b       // a must be true on the current clock tick
              // and b on the next clock tick
a ##N b       // Check b on the Nth clock tick after a
a ##[1:4] b // a must be true on the current clock tick and b
            // on some clock tick between the first and fourth
            // after the current clock tick
a ##1 b [*3] ##1 c // Equiv. to a ##1 b ##1 b ##1 b ##1 c
(a ##2 b) [*2]       // Equiv. to (a ##2 b ##1 a ##2 b)
(a ##2 b)[*1:3]      // Equiv. to (a ##2 b)
                     // or (a ##2 b ##1 a ##2 b)
                     // or (a ##2 b ##1 a ##2 b ##1 a ##2 b)
a ##1 b [*1:$] ##1 c  // E.g. a b b b b c
a ##1 b[->1:3] ##1 c // E.g. a !b b b !b !b b c
a ##1 b [=1:3] ##1 c // E.g. a !b b b !b !b b !b !b c
```

ג. שכבת ה- property

בעזרת properties ניתן לבטא בצורה מדויקת את ההתנהגות מצופה של תכנון. שכבה זאת משתמשת בשכבה הבוליאנית ובשכבת ה- sequence על מנת להגדיר התנהגות של תכנון. קיימים דרכים רבות להגדיר properties, להלן מספר דוגמאות:

```
property sequence_example;
    s1; // s1 is a sequence defined elsewhere
endproperty
```

שימוש ב- property אחר:

```
property property_example;
    sequence_example;
endproperty
```

קיום של s0 גורר קיום של sequence_example:

```
property implication_example;
    s0 |-> sequence_example;
endproperty
```

ניתן לציין באיזו נקודת זמן ה- property מתקיים, לדוגמא בעלית השעון :

```
property clocking_example;
    @(posedge clk) sequence_example;
endproperty
```

ה- property תקף רק אם תנאי מסוים לא מתקיים :

```
 property disable_iff_example;
    disable iff (reset_expr) sequence_example;
endproperty
```

דוגמא :

על מנת שלא יתבצע הבלוק כאשר reset גבוה , לדוגמא :

```
property req_gnt_prop;
  @ (posedge clk) // At every posedge clk
   disable iff (reset) // disable if reset is asserted
    req |=> req_gnt_seq;
 endproperty
```

ד. שכבת ה- property directive – מגדירה פעולות אותן יש לבצע על שכבת ה- property.
הפעולות האפשריות הן :

- assert : בודק האם ה- property מתקיים
- assume : מניח שה- property מתקיים
- cover : מגדיר את התכונה לבדיקת coverage

Assert :

```
property top_prop;
    seq0 |-> prop0;
endproperty

assert_top_prop:
  assert property (top_prop)
  begin
     pass_block()
  end
  else
  begin
     fail_block()
  end
```

אם top_prop מתקיים, כלומר ה- assert הצליח, אז יתבצע pass_block() אחרת יתבצע fail_block(). אין חובה להגדיר בלוקים אלו. אם אף אחד מהם לא מוגדר, הכלי פשוט יציין האם ה- assert הצליח או נכשל.
דוגמא :

```
//+++++++++ DUT With assertions ++++++++++++
module concurrent_assertion( input wire clk,req,reset, output reg gnt);

//========= Sequence Layer ===============
sequence req_gnt_seq;
// (~req & gnt) and (~req & ~gnt) is Boolean Layer
    (~req & gnt) ##1 (~req & ~gnt);
endsequence
```

```
//======= Property Specification Layer =========
property req_gnt_prop;
   @ (posedge clk)
      disable iff (reset)
        req |-> ##2 req_gnt_seq;
endproperty

//=========Assertion Directive Layer ============
req_gnt_assert :
                    assert property (req_gnt_prop)
                    else
                    $display("@%0dns Assertion Failed", $time);

//========= Actual DUT RTL ==================
always @ (posedge clk)
    gnt <= req;
endmodule
```

בניסוי זה לא נעסוק ב- assume ו- cover. ניתן למצוא הסבר ב:
http://www.project-veripage.com/sva_15.php.

משפט expect

משפט זה הוא כמו משפט assert חוץ מכך שהוא מתבצע בתוך משפטי initial או always ומחכה עד שהסדרה שמוגדרת בו תתבצע. דוגמא לשימוש:

```
initial begin
   ##1 ;
  // Wait for the sequence if pass, terminate sim
  expect ( @ (posedge clk) a  ##1  b  ##1  c  ##1   ! c)
    $finish
   else
    $error ("Something is wrong");
end
```

functional coverage

functional coverage הוא כלי להערכת איכות הבדיקות שבוצעו. ניתן למדוד כמה מהקוד נבדק ע"י הסימולציה, למשל במקרה של מכונת מצבים – בכמה מצבים ובאילו קשתות המכונה עברה. בעזרת מדד זה, ניתן להעריך האם יש צורך בבדיקות נוספות. על מנת לאפשר מדידת כיסוי, יש לבצע שתי פעולות:

א. הגדרת בלוק מסוג covergroup.
ב. ביצוע אינסטנציאציה של ה- covergroup.

דוגמא:

```
enum { red, green, blue } color;
covergroup g1 @(posedge clk); // Sample at posedge clk
    c: coverpoint color;
endgroup
g1 g1_inst = new; // Create the coverage object
```

coverpoint הוא משתנה אשר יכיל את כל הערכים שקיבל הסיגנל אותו הוא דוגם.
דוגמא של ניטור של צירופים של משתנים שונים:

```
enum { red, green, blue } color;
bit [3:0] pixel_adr, pixel_offset, pixel_hue;
covergroup g2 @(posedge clk);
   Hue: coverpoint pixel_hue;
   Offset: coverpoint pixel_offset;
// Consider (color, pixel adr) pairs, e.g,.
// (red, 3'b000), (red, 3'b001), ..., (blue, 3'b111)
   AxC: cross color, pixel_adr;
// Consider (color, pixel hue, pixel offset) triplets
// Creates 3 * 16 * 16 = 768 bins
   all: cross color, Hue, Offset;
endgroup
g2 g2_inst = new; // Create a watcher
```

דוגמא של נטרול בתנאים מסוימים :

```
covergroup g4 @(posedge clk);
// check s0 only if reset is true
   coverpoint s0 iff(!reset);
endgroup
```

דוגמא של מעקב אחרי רק ערכים מסוימים של משתנה :

```
bit [9:0] a; // Takes values 0–1023
covergroup cg @(posedge clk);
coverpoint a {
// place values 0–63 and 65 in bin a
bins a = { [0:63], 65 };
// create 65 bins, one for 127, 128, ..., 191
bins b[] = { [127:150], [148:191] };
// create three bins: 200, 201, and 202
bins c[] = { 200, 201, 202 };
// place values 1000–1023 in bin d
bins d = {[1000:$] };
// place all other values (e.g., 64, 66, .., 126, 192, ...) in their own bin
bins others[] = default;
}
endgroup
```

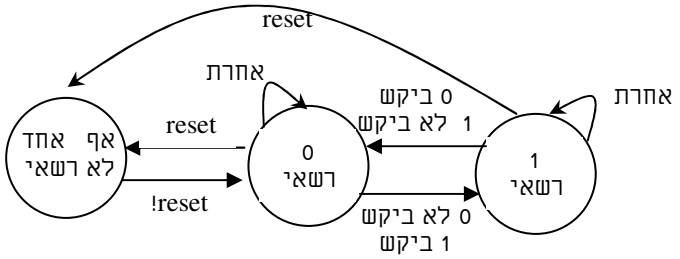
דוגמא של מעקב אחרי מעברים :

```
bit [3:0] a;
covergroup cg @(posedge clk);
coverpoint a {
// Place any of the sequences 4→5→6, 7→11, 8→11, 9→11, 10→11,
// 7→12, 8→12, 9→12, and 10→12 into bin sa.
bins sa = (4 => 5 => 6), ([7:9],10 => 11,12);
// Create separate bins for 4→5→6, 7→10, 8→10, and 9→10
bins sb[] = (4 => 5 => 6), ([7:9] => 10);
// Look for the sequence 3→3→3→3
bins sc = 3 [* 4];
// Look for any of the sequences 5→5, 5→5→5, or 5→5→5→5
```

```
bins sd = 5 [* 2:4];
// Look for any sequence of the form 6→_ _ _→6→_ _ _→6
// where "_ _ _" represents any sequence that excludes 6
bins se = 6 [-> 3];
}
endgroup
```

דוגמא מסכמת

בדוגמא זו נרצה ליצור בורר (arbiter) הוגן אשר נותן גישה לרכיב אחד בלבד בכל מחזור שעון. נממש אותו בעזרת מכונת מצבים ונעזר בכל כלי השפה השונים כדי לבדוק אותו דיאגרמת המצבים של הבורר :



הקובץ שמממש את הבורר :

```
typedef enum {Reset,Grant0,Grant1} states;
```
יצירת סוג משתנה חדש

```
module arbiter (
  input wire [0:1] req  , // Request
  input wire clk   , // Clock
  input wire reset , // Reset
  output reg  [0:1] grant // Grant
  );

states present_state,next_state;
  always_ff @ ( posedge clk iff reset == 0 or posedge reset)
  begin
    if (reset) begin
     present_state <= Reset;
    end else begin
     present_state <= next_state;
    end
  end
```

החלק הסינכרוני של מכונת המצבים

```
always_comb
begin
 case(present_state)
   Reset : begin
             grant[0] <= 1'b0;
             grant[1] <= 1'b0;
          next_state <= Grant0;
         end
   Grant0 : begin
              grant[0] <= 1'b1;
              grant[1] <= 1'b0;
              if ((req[1] == 1'b1) && (req[0] == 1'b0)) begin
```

החלק הקומבינטורי של מכונת המצבים

```
            next_state <= Grant0;
             end
            else begin
              next_state <= Grant1;
             end
           end
  Grant1 : begin
             grant[0] <= 1'b0;
             grant[1] <= 1'b1;
             if ((req[0] == 1'b1) && (req[1] == 1'b0)) begin
           next_state <= Grant1;
             end
             else begin
               next_state <= Grant0;
             end
            End
  default : $display("Error in SEL");
 endcase
end
// assert if both unit don't ask access
always @ (posedge clk iff reset == 0)
begin
 REQ : assert (req[0] || req[1]) begin

     end else begin
          $display ("No unit asked for access");
    end
end
endmodule
```

משפט assert שתפקידו לוודא שבכל עליית שעון – לפחות אחת מהיחידות מבקשת גישה

קובץ ה- testbench :

```
program arb_test (
  input wire [0:1] grant  , // Request
  input wire clk   , // Clock
  output reg reset , // Reset
  output reg  [0:1] req
);

covergroup sta @ (posedge clk);
st :  coverpoint top.arb.present_state {
   bins t0 = (Reset=>Reset);
   bins t1 = (Reset=>Grant0);
   bins t2 = (Reset=>Grant1);
   bins t3 = (Grant0=>Reset);
   bins t4 = (Grant0=>Grant0);
   bins t5 = (Grant0=>Grant1);
   bins t6 = (Grant1=>Reset);
   bins t7 = (Grant1=>Grant0);
   bins t8 = (Grant1=>Grant1);
}
endgroup
```

יצירת covergroup על המעברים בין המצבים השונים. כך נדע איזה אחוז של מכונת המצבים בדקנו

```
sta cov = new();

class forrand;
 rand bit request;
endclass

initial
begin
  forrand request0= new();
  forrand request1= new();
  $write("starting simulation \n");
  reset <= 1'b1;
  repeat (2) @(posedge clk);
  reset <= 1'b0;

  fork
    repeat(10)
    begin
      @(posedge clk);
      request0.randomize();
      req[0] <= request0.request;
      expect (@(posedge clk) ##[1:3] (grant[0] && req[0]));
      req[0] <= 1'b0;
    end

   repeat(10)
    begin
      @(posedge clk);
      request1.randomize();
       req[1] <= request1.request;
       expect (@(posedge clk) ##[1:3] (grant[1]&& req[1]));
       req[1] <= 1'b0;
    end
  join
end
endprogram
```

כיוון שמשתנים אקראיים יכולים להופיע רק בתוך מחלקה – זוהי מחלקה שכל תפקידה הוא להכיל משתנה אקראי

לאחר משפט ה- fork – כל בלוק repeat יתבצע במקביל לבלוק האחר ויהיה שייך ליחידה אחת המבקשת גישה.
פיצול זה מקל עלינו את עבודת הבדיקה עבור כל יחידה בנפרד

הקובץ שמחבר את שניהם :

```
module top ();

reg [0:1] grant  ; // Request
reg clk ; // Clock

reg reset ; // Reset
reg  [0:1] req;

initial begin
  clk = 0;
  #500 $finish;
end

always  #1  clk  = ~clk;
```

```
arb_test testbench (.grant(grant),.clk(clk),.reset(reset),.req(req));
arbiter arb (.grant(grant),.clk(clk),.reset(reset),.req(req));

endmodule
```

כעת, אם נבצע סימולצייה כפי שמוסבר בהמשך – נקבל בהסתברות כלשהיא טעויות בריצה. טעויות אלו באו להמחיש שלמרות שתכננו בקר הוגן – מימשנו בקר שאיננו הוגן ולכן אחד הצרכנים שלו יכול לגרום להרעבה של הצרכן השני. לולא שמנו את משפט ה- expect – לא היינו מודעים לבעייה זו.

בעיה נוספת המתעוררת היא דווקא בתוכנית הבדיקה אשר בהסתברות כלשהי גורמת לכך שאף יחידה לא תבקש גישה. לולא משפט ה-assert שבקובץ התכנון לא היינו מודעים לכך. לכן, גם כאשר מתכננים תכנון ללא testbench רצוי להוסיף לו assertions על מנת שנוכל לוודא את נכונותו בקלות.

סימולצייה של מעגלים ב- Verilog/SystemVerilog

ניתן לבצע סימולציה בעזרת שני סימולטורים :

א. הכלי NCSIM של חברת Cadence

```
ncvlog –sv –nowarn DCLPTH –nocopyright –work work basename.sv
ncvlog –sv –nowarn DCLPTH –nocopyright –work work basename_test.sv
ncelab –acess +rwc basename_test
ncsim –gui basename
```

הערה : ההנחה היא שהתכנון בשם basename רשום בקובץ basename.sv, ה- testbench בשם basename_test רשום בקובץ בשם basename_test.sv. הוכן script בשם run_sim אשר מבצע את כל הפקודות הנ"ל. הרצת ה- script מתבצע באופן הבא :

```
run_sim basename
```

ב. הכלי VCS של חברת Synopsys

על מנת להפעיל את הכלי כך שייצור functional coverage יש לכתוב את השורה הבאה :

```
vcs -RI -line -sverilog FILES -override_timescale=1ns/1ps -sverilog -debug_all
+define+ASSERT_ON+COVER_ON
```

כאשר FILES הם שמות כל הקבצים

לאחר שמסתיימת הסימולציה, על מנת לקבל את הcoverage שהגדרנו יש לכתוב :

```
urg -dir simv.vdb -format text
```

פקודה זו תיצור ספרייה חדשה בשם urgReport המכילה את התוצאות השונות.

סינטזה של מעגלים ב- Verilog/SystemVerilog

סינתזה מתבצעת בעזרת כלי ה- design_vision של חברת Synopsys. ראה את המדריך המתאים על מנת להפעיל את הכלי.

הערה : על מנת שהכלי יזהה שמדובר בקבצי Systemverilog יש לתת להם סיומת sv. הדברים הבאים אינם נתמכים בסינתזה:

1. בלוקים של program – הם ל- testbench בלבד
2. assertions – הכלי מתעלם מהם

3. משפטי iff – כלי הסינתזה בגרסה הנוכחית שלו לא תומך בפקודה זו

אם הכלי מודיע על שגיאות יש לבדוק שהתבצעה הפקודה {} analyze –format sverilog
אם התבצעה פקודה אחרת – יש להעתיק אותה ולשנות בהתאם