DIGITAL DESIGN WITH FPGA CAMP

DAY 2 SIMULATION

OVERVIEW

HOW TO CHECK DESIGN

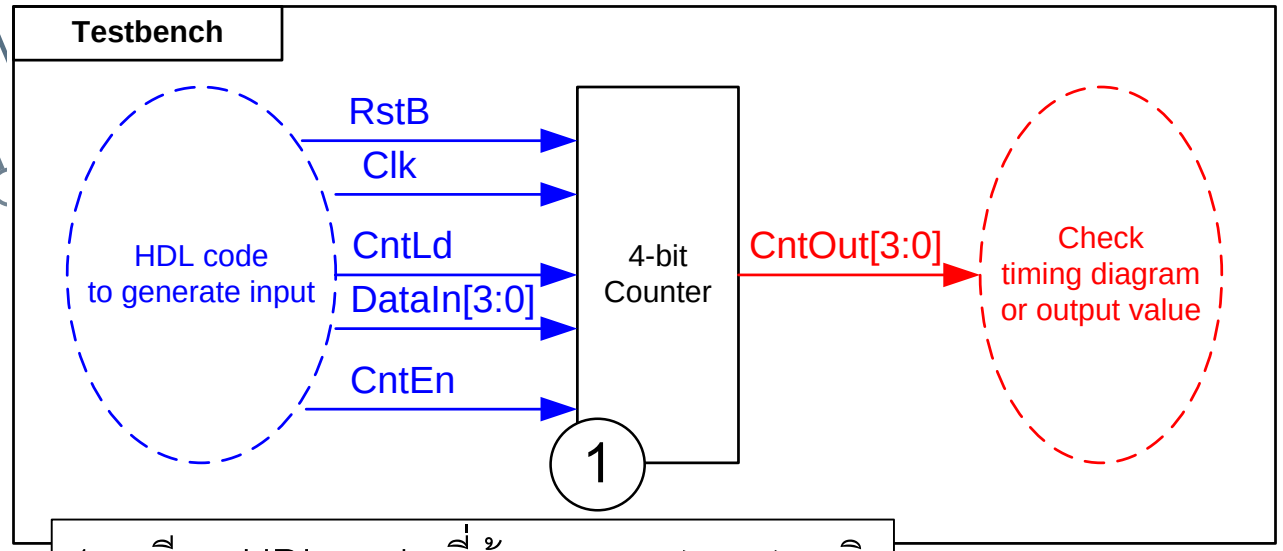
ทดสอบบนบอร์ดจริง

- ต้องมี hardware พร้อมใช้งานจึงจะทดสอบได้
- หากทำงานผิด hardware อาจเสียหายได้
- เมื่อมีการแก้ไขวงจร จะต้องรอเป็นเวลานาน เพื่อให้ tool แปลง HDL code มาเป็น hardware บน FPGA
- ไม่สามารถเช็คได้ว่า ทำงานเร็ว/ช้า เพราะไม่เห็น timing diagram
- เมื่อทำงานผิด จะวิเคราะห์ยากว่า สัญญาณไหนที่ทำงานผิดไปบ้าง

ทดสอบโดยการจำลองการทำงาน

- สามารถทดสอบวงจรได้ โดยไม่ต้องมี hardware
- เมื่อแก้ไขวงจร ใช้เวลารอไม่นานในการที่จะได้ผลลัพธ์ออกมา เพื่อตรวจสอบอีกครั้ง
- สามารถเห็นรายละเอียดการทำงานได้ทุกสัญญาณในวงจรที่เราออกแบบ
- ตรวจสอบ timing diagram ได้ ทำให้ทราบจังหวะการทำงาน ว่าเร็ว/ช้า อย่างไรบ้าง
- สามารถทดสอบกรณีแปลก ๆ ได้มากมาย โดยการจำลองใส่ input แปลก ๆ ลงไป

HOW TO SIMULATION



1. เตรียม HDL code ที่ต้องการทดสอบ สมมุติว่าเป็นวงจร 4-bit counter แบบนับขึ้น

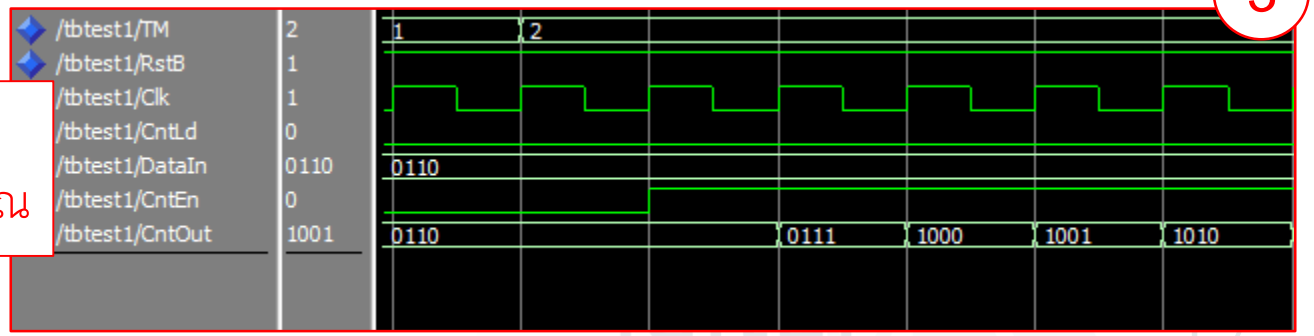
2

```
-- Concurrent signal

u_Clk : Process
Begin
    Clk     <= '1';
    wait for tClk/2;
    Clk     <= '0';
    wait for tClk/2;
End Process u_Clk;

u_RstB : Process
Begin
    RstB    <= '0';
    wait for 10*tClk;
    RstB    <= '1';
    wait;
End Process u_RstB;

u_Test1 : Test1
Port map
(
    RstB        => RstB     ,
    Clk         => Clk      ,

    CntLd       => CntLd    ,
    DataIn      => DataIn   ,
    CntEn       => CntEn    ,
```

```
u_Test : Process
Begin
    -- TM=0 : Reset module
    TM <= 0; wait for 1 ns;
    Report "TM=" & integer'image(TM);

    CntLd       <= '0';
    DataIn      <= "0000";
    CntEn       <= '0';
    wait for 20*tClk;

    -- TM=1 : Load input
    TM <= 1; wait for 1 ns;
    Report "TM=" & integer'image(TM);

    wait until rising_edge(Clk);
    CntLd       <= '1';
    DataIn      <= "0110";
    wait until rising_edge(Clk);
    CntLd       <= '0';

    wait for 20*tClk;

    -- TM=2 : Counter enable
(TM);
```

2. เขียน HDL code เพื่อสร้างสัญญาณ input ทั้งหมดป้อนให้กับวงจร ได้แก่ สัญญาณ RstB, Clk, CntLd, DataIn และ CntEn

3

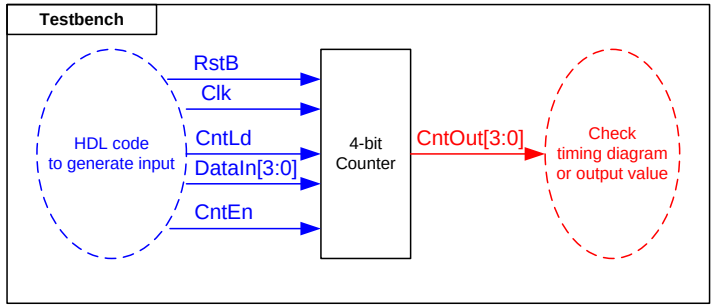


3. ตรวจสอบสัญญาณ output ของวงจร (CntOut) ว่า มีค่าถูกต้องหรือไม่ โดยดูจาก waveform ที่แสดง timing diagram ของสัญญาณ

ELECTRICAL ENGINEERING CHULALONGKORN UNIVERSITY · TESA · NDR Solution · DG DESIGN GATEWAY · Silicon craft

TESTBENCH

TESTBENCH DESIGN



ไฟล์สำหรับการทดสอบเพื่อจำลองการทำงานของ HDL ที่เราออกแบบ จะถูกเรียกว่า testbench เป็นไฟล์ HDL ที่จะประกอบด้วย 3 ส่วนหลัก ๆ คือ

1. การเรียกใช้ HDL ที่เราออกแบบไว้ และต้องการจะทดสอบ โดยเรียกผ่าน component
2. การเขียน HDL code เพิ่มเติม เพื่อสร้างสัญญาณ input ป้อนให้กับวงจรของเรา โดย VHDL code จะมีคำสั่งพิเศษจำนวนมาก ที่ออกแบบไว้ เพื่อใช้งานสำหรับ testbench เท่านั้น เช่น time, wait for, wait until, after, while … loop, exit, assert
3. การ**ตรวจสอบสัญญาณ** output ที่ได้จากวงจร ว่าถูกต้องหรือไม่ ในส่วนนี้สามารถทำได้ 2 แบบ คือ
   a) การตรวจสอบด้วยตาเปล่า : นิยมใช้สำหรับคนที่ออกแบบใหม่ ๆ และระบบยังไม่ใหญ่มาก (ไม่แนะนำวิธีนี้)
   b) การตรวจสอบอัตโนมัติ : มีการเขียน HDL ไว้ เพื่อตรวจสอบค่า output ในแต่ละจังหวะการทำงาน ว่ามีค่าเท่ากับค่าที่เราคาดหวังไว้ไหม ถ้าไม่ตรง จะหยุดการทดสอบ และแจ้ง error ขึ้นมา

1. COMPONENT

COMPONENT DECLARATION

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

Entity TbTest1 Is
End Entity TbTest1;

Architecture rtl of TbTest1 is

------------------------------------------------------------
-- Constant Declaration
------------------------------------------------------------

    constant    tClk        : time := 10 ns;

------------------------------------------------------------
-- Component Declaration
------------------------------------------------------------

    Component Test1 Is
    Port
    (
        RstB        : in    std_logic;
        Clk         : in    std_logic;

        CntLd       : in    std_logic;
        DataIn      : in    std_logic_vector( 3 downto 0 );
        CntEn       : in    std_logic;

        CntOut      : out   std_logic_vector( 3 downto 0 )
    );
    End Component Test1;

------------------------------------------------------------
-- Signal Declaration
```

Component: เป็นการประกาศเรียกใช้ entity ในไฟล์ HDL ที่เราได้ออกแบบไว้ เพื่อนำมาทดสอบใน testbench

ตัวอย่างนี้ วงจรที่ทดสอบคือ 4-bit counter นั้นออกแบบภายใต้ entity ที่มีชื่อว่า Test1

ข้อสังเกต : เราสามารถ copy ส่วนที่ประกาศ entity มาใช้งานได้เลย แล้วแก้ไขคำว่า Entity เป็น Compoent และ End Entity เป็น End Component

การเขียน testbench จะเหมือน VHDL ทั่วไป คือ ประกอบด้วย

- การประกาศ library ที่จะใช้งาน (ใน testbench อาจจะมีการเรียกใช้ package พิเศษที่เราออกแบบไว้เพิ่มเติมได้)
- การประกาศ Entity ใหม่ แต่ที่พิเศษคือ ไม่มีส่วนของ input/output เพราะเป็นไฟล์สำหรับทดสอบวงจรเท่านั้น
- การประกาศ constant ชนิด time เพื่อกำหนดคาบเวลาของสัญญาณ clock ที่จะใช้ทดสอบวงจร
- การประกาศ component ที่จะทดสอบ

ยังมีต่ออีกนะ

COMPONENT MAPPING

```
-------------------------------------------------------------------
-- Signal Declaration
-------------------------------------------------------------------

    signal  TM          : integer   range 0 to 65535;

    signal  RstB        : std_logic;
    signal  Clk         : std_logic;
    signal  CntLd       : std_logic;
    signal  DataIn      : std_logic_vector( 3 downto 0 );
    signal  CntEn       : std_logic;
    signal  CntOut      : std_logic_vector( 3 downto 0 );

Begin

-------------------------------------------------------------------
-- Concurrent signal
-------------------------------------------------------------------

    u_Test1 : Test1
    Port map
    (
        RstB        => RstB     ,
        Clk         => Clk      ,

        CntLd       => CntLd    ,
        DataIn      => DataIn   ,
        CntEn       => CntEn    ,

        CntOut      => CntOut
    );
```

ชื่อ Port ของ Test1 ที่ออกแบบไว้

ชื่อ signal บน testbench

เป็นการเรียกใช้ component ที่เราประกาศไว้ โดยการเชื่อมโยงสัญญาณที่เราประกาศไว้ใน HDL code นี้ (testbench) เข้ากับ input และ output ทั้งหมดของ component ในตัวอย่างนี้ เพื่อความง่าย เราประกาศใช้ชื่อสัญญาณที่มีชื่อเดียวกับ port ทั้งหมดของ Test1

ส่วนประกอบของ testbench (ต่อจากหน้าที่แล้ว)

- การประกาศ signal ทั้งหมดที่จะใช้งานใน testbench ซึ่งจะประกอบด้วย ชื่อสัญญาณที่ต่อกับ port ทั้งหมดของ component ที่เราจะทดสอบ และอาจจะมีสัญญาณเพิ่มเติมที่เราจะใช้งานใน testbench (ในตัวอย่าง เช่น สัญญาณชื่อ TM ที่จะใช้ระบุว่า ลำดับหัวข้อที่กำลัง test อยู่ เพื่อเวลาที่วงจรทำงานไม่ผ่าน จะได้ทราบว่า ไม่ผ่านที่หัวข้อใด)
- การ map วงจรที่เราจะทดสอบ เข้ากับสัญญาณ (Component mapping)

จบส่วนที่ 1 แต่ testbench ยังมีส่วนที่ 2 ต่อนะ

MORE INFO (COMPONENT)

- Component เป็นคำสั่งทั่วไป ที่ไม่ได้ใช้ใน testbench เท่านั้น แต่สามารถนำไปใช้ในการออกแบบวงจรได้ เมื่อเราต้องการเรียกใช้วงจรที่เราได้เคยออกแบบไปแล้ว ก็จะเรียกผ่านการประกาศ component เช่นกัน และ map สัญญาณต่อที่ port input/output ไป เหมือนกับใน testbench
- หากวงจรเราต้องการใช้ component นี้มากกว่า 1 ตัว ไม่ต้องประกาศซ้ำ 2 รอบ แต่สามารถเขียน code สำหรับ การ mapping สัญญาณเข้าไป 2 ชุดได้เลย แต่ต้องตั้งชื่อหน้าส่วนของ mapping ให้ต่างกัน เช่น u_Test1a, u_Test1b เป็นต้น
- วงจรหนึ่ง ๆ อาจจะเรียกใช้ component หลาย ๆ ตัวได้

EXAMPLE COMPONENT IN HARDWARE DESIGN (NOT TESTBENCH)

```
Entity MainModule Is
    Port
    (
        RstB        : in    std_logic;
        Clk50       : in    std_logic;
        Button      : in    std_logic;
        LED         : out   std_logic_vector( 7 downto 0 )
    );
End Entity MainModule;

Architecture rtl Of MainModule Is
------------------------------------------------------------------
-- Component declaration
------------------------------------------------------------------
    Component BlinkLED Is
        Port
        (
            RstB        : in    std_logic;
            Clk50       : in    std_logic;
            Enable      : in    std_logic;
            LED         : out   std_logic_vector( 7 downto 0 )
        );
    End Component BlinkLED;

    Component Debouncer Is
        Port
        (
            Clk50       : in    std_logic;
            Button      : in    std_logic;
            Enable      : out   std_logic
        );
    End Component Debouncer;
------------------------------------------------------------------
```

```
------------------------------------------------------------------
-- Signal declaration
------------------------------------------------------------------
    signal  LEDEn       : std_logic;
Begin
------------------------------------------------------------------
-- Component mapping
------------------------------------------------------------------
    u_BlinkLED : BlinkLED
    Port Map
    (
        RstB        => RstB     ,
        Clk50       => Clk50    ,
        Enable      => LEDEn    ,
        LED         => LED
    );
    u_Debouncer : Debouncer
    Port Map
    (
        Clk50       => Clk50        ,
        Button      => PushButton   ,
        Enable      => LEDEn
    );
End Architecture rtl;
```

- วงจรหนึ่ง ๆ อาจจะเรียกใช้ component หลาย ๆ ตัวได้
- Port ของ component ที่เรียกใช้งาน อาจจะต่อกับ input/output ของ entity เลยก็ได้ ไม่จำเป็นต้องประกาศสัญญาณเพิ่มเติม

2. GENERATE INPUT IN TESTBENCH

PROCESS TO GENERATE INPUT

การสร้างสัญญาณ input เพื่อป้อนให้กับ component ที่เราจะทดสอบ โดยทั่วไปประกอบด้วยอย่างน้อย 3 Process ได้แก่

1. Process สำหรับการสร้างสัญญาณ reset (RstB) ให้กับระบบ
2. Process สำหรับการสร้างสัญญาณ clock (Clk) ให้กับระบบ
3. Process สำหรับการสร้างสัญญาณ input อื่น ๆ ให้กับระบบ (u_Test)

```
u_RstB : Process
Begin
    RstB     <= '0';
    wait for 10*tClk;
    RstB     <= '1';
    wait;
End Process u_RstB;
```

1

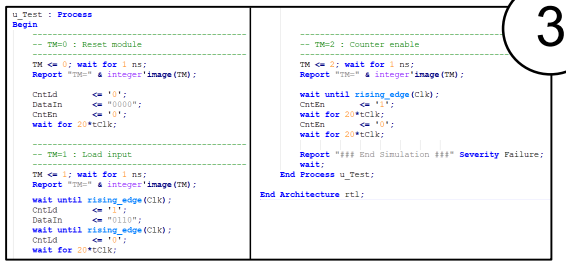
```
u_Clk : Process
Begin
    Clk      <= '1';
    wait for tClk/2;
    Clk      <= '0';
    wait for tClk/2;
End Process u_Clk;
```

2

```
u_Test : Process
Begin
    ------------------------------------------
    -- TM=0 : Reset module
    ------------------------------------------
    TM <= 0; wait for 1 ns;
    Report "TM=" & integer'image(TM);

    CntLd       <= '0';
    DataIn      <= "0000";
    CntEn       <= '0';
    wait for 20*tClk;

    ------------------------------------------
    -- TM=1 : Load input
    ------------------------------------------
    TM <= 1; wait for 1 ns;
    Report "TM=" & integer'image(TM);

    wait until rising_edge(Clk);
    CntLd       <= '1';
    DataIn      <= "0110";
    wait until rising_edge(Clk);
    CntLd       <= '0';
    wait for 20*tClk;

    ------------------------------------------
    -- TM=2 : Counter enable
    ------------------------------------------
    TM <= 2; wait for 1 ns;
    Report "TM=" & integer'image(TM);

    wait until rising_edge(Clk);
    CntEn       <= '1';
    wait for 20*tClk;
    CntEn       <= '0';
    wait for 20*tClk;

    Report "### End Simulation ###" Severity Failure;
    wait;
End Process u_Test;

End Architecture rtl;
```

3

TEST PROCESS



- Process สำหรับการสร้างสัญญาณ input จะทำงานแบบ sequential (สร้างสัญญาณ input ตามลำดับจากบรรทัดบนสุดถึงบรรทัดล่างสุด)
- Process จะไม่ได้ใส่ sensitivity list ไว้เลย เพื่อเริ่มต้นทำงานทันที
- จะแบ่งการทดสอบออกเป็นหลาย ๆ TM เพื่อแยกทดสอบทีละ function เช่น

  TM=0 ใช้ reset ค่า input ทั้งหมด

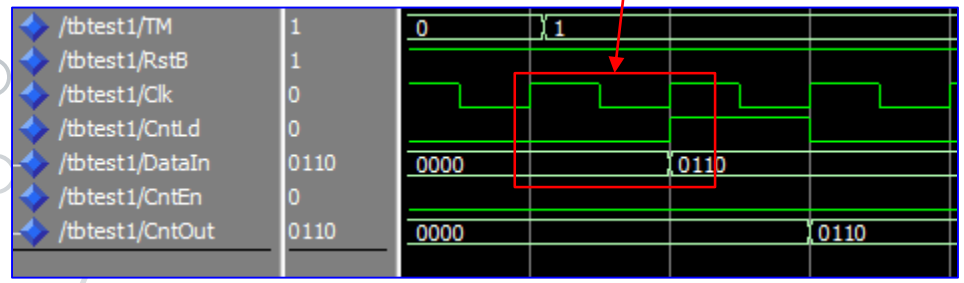
  TM=1 ใช้ทดสอบฟังก์ชันการโหลดค่าของ 4-input counter ว่าทำงานได้ถูกต้องไหม (สร้างสัญญาณ CntLd/DataIn)

  TM=2 ใช้ทดสอบฟังก์ชันสำหรับการนับของ 4-input counter ว่าทำงานได้ถูกต้องไหม (สร้างสัญญาณ CntEn)

  ในแต่ละ TM อาจจะไม่ต้องกำหนดค่า input ทุกสัญญาณ เราจะเขียน code เฉพาะสัญญาณที่ต้องการแก้ไขเท่านั้น สัญญาณที่ไม่ได้แก้ไข ก็จะคงค่าเดิมไว้ก่อน
- เมื่อจบบรรทัดสุดท้าย จะใส่ report แบบ failure ไว้ เพื่อจบการทดสอบระบบ (ถ้าไม่ใส่ จะกลับไปเริ่มต้นทำงานที่บรรทัดแรกของ Process ใหม่ หากไม่ต้องการให้กลับไปเริ่มต้นบรรทัดแรก เราจะใส่คำสั่ง wait; เพื่อเป็นการรอแบบ forever ไว้ที่บรรทัดสุดท้าย)
- Process การสร้าง input อาจจะมีมากกว่า 1 Process เมื่อต้องการสร้าง input ที่ซับซ้อนและทำงานอิสระต่อกัน

WAIT UNTIL

```
-------------------------------------
-- TM=1 : Load input
-------------------------------------
TM <= 1; wait for 1 ns;
Report "TM=" & integer'image(TM);
wait until rising_edge(Clk);
CntLd       <= '1';
DataIn      <= "0110";
wait until rising_edge(Clk);
CntLd       <= '0';
wait for 20*tClk;
```



จะเห็นว่า CntLd และ DataIn เปลี่ยนหลังจาก TM=1 ไปอีก 1 Clock จากคำสั่ง wait until

Wait until condition;

เป็นคำสั่งให้รอจนกว่า condition ที่เรากำหนดนั้น จะมีการเปลี่ยนแปลงจาก False เป็น True เช่น

wait until rising_edge(Clk); คือ จะรอจนกว่าเจอขอบขาขึ้นของ Clk แล้วจึงจะทำงานบรรทัดต่อไป (มักจะใช้เพื่อสร้างสัญญาณ input ให้เปลี่ยนแปลงตามจังหวะการทำงานของสัญญาณนาฬิกา)

เพิ่มเติม

wait until A='1'; เมื่อเจอคำสั่งนี้ ตัว tool จะรอจนกว่า A จะเปลี่ยนจากค่าอื่น ๆ มาเป็นค่า '1' แล้วจึงจะทำงานบรรทัดถัดไป แต่หาก A มีค่าเท่ากับ '1' อยู่แล้ว วงจรจะรอจนกว่า A จะเปลี่ยนค่าไปเป็นค่าอื่นก่อน แล้วเปลี่ยนกลับมาเป็น '1' อีกครั้งจึงจะทำงานบรรทัดถัดไป

WAIT FOR

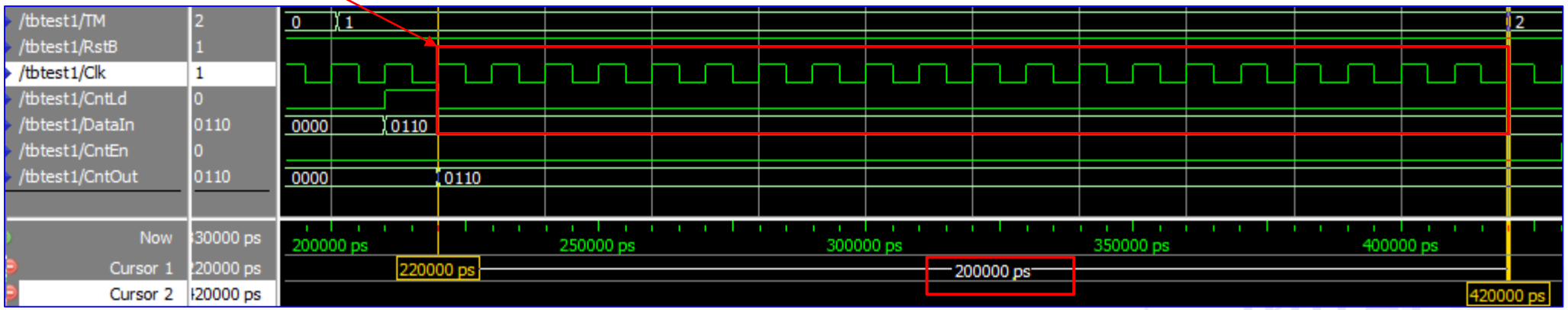
```
-------------------------------------
-- TM=1 : Load input
-------------------------------------
TM <= 1; wait for 1 ns;
Report "TM=" & integer'image(TM);

wait until rising_edge(Clk);
CntLd       <= '1';
DataIn      <= "0110";
wait until rising_edge(Clk);
CntLd       <= '0';
wait for 20*tClk;
```

wait for time;

เป็นการระบุเวลาที่จะรอ เช่น

wait for 10 ns;

ในตัวอย่าง ด้านซ้ายนั้น ค่า tClk เป็น constant ที่ได้ประกาศไว้แล้ว ซึ่งระบุเป็นคาบของสัญญาณ Clk เท่ากับ 10 ns (ย้อนกลับไปดูเพิ่มเติมได้ใน Page 8) ดังนั้นการรอ 20*tClk คือ การเขียน code ว่าให้รอไปอีก 20*10 = 200 ns นั่นเอง



REPORT

```
    TM <= 2; wait for 1 ns;
    Report "TM=" & integer'image(TM);

    wait until rising_edge(Clk);
    CntEn       <= '1';
    wait for 20*tClk;
    CntEn       <= '0';
    wait for 20*tClk;

    Report "### End Simulation ###" Severity Failure;
    wait;
End Process u_Test;
```

```
** Note: TM=0
   Time: 1 ns  Iteration: 0  Instance: /tbtest1
** Note: TM=1
   Time: 202 ns  Iteration: 0  Instance: /tbtest1
** Note: TM=2
   Time: 421 ns  Iteration: 0  Instance: /tbtest1
** Failure: ### End Simulation ###
```

ชุดคำสั่งสำหรับ Report แบบเต็ม ๆ จะมี Assert อยู่ด้านหน้าด้วย เพื่อใช้ในการตรวจสอบสัญญาณ ซึ่งจะกล่าวถึงในส่วนที่ 3 ในตัวอย่างนี้ แยกเขียนมาเฉพาะส่วนที่เป็น Report และ Severity

Report ใช้ส่ง message ขึ้นมาบน console ของ tool ที่เรากำลังใช้จำลองการทำงานอยู่

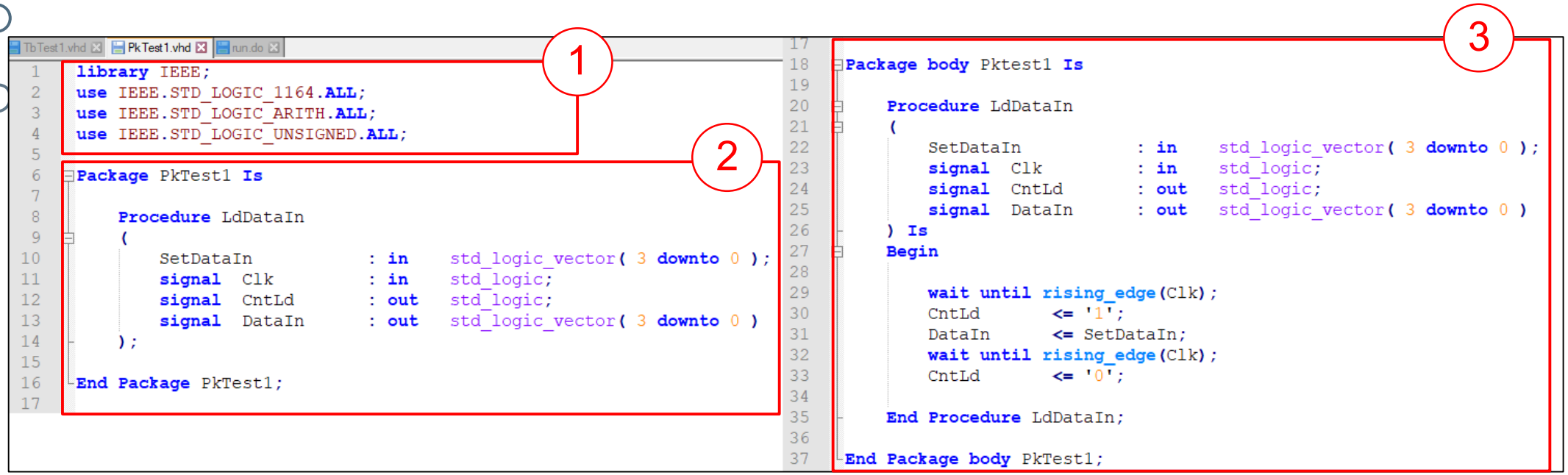
Severity ใช้ระบุความรุนแรงของ Report โดยทั่วไปจะใส่ค่า Failure เพื่อระบุว่าเป็น error ที่ร้ายแรง และให้เบรกการทำงาน หรือใช้เบรกการทำงานเพื่อจบ simulation เลย หากไม่ใส่ Severity วงจรจะทำงานต่อปกติ

INPUT PROBLEM

ข้อเสีย : หากสัญญาณ input มีจำนวนมาก มีเงื่อนไขที่ต้องสร้างมาก และหลาย ๆ ครั้งต้องสร้างสัญญาณในรูปแบบเดิม ๆ จะต้องเขียน code หลาย ๆ บรรทัด ในรูปแบบเดิม ๆ จำนวนมาก

แนวทางแก้ไข : สร้าง Procedure ขึ้นมา เพื่อตัด code ส่วนที่ต้องทำงานเดิม ๆ ซ้ำ ๆ หลาย ๆ รอบไปใส่ใน Package ต่างหาก แล้วเรียกมาใช้งานโดยการเขียน code เพียงบรรทัดเดียว แทนการเขียน code หลาย ๆ บรรทัด

PACKAGE



Package เป็น HDL code ที่เหมือนเป็นศูนย์รวมของ constant function และ procedure ที่เรียกใช้งานบ่อย ๆ ประกอบด้วย

ส่วนแรก คือ library standard เหมือน HDL code อื่น ๆ

ส่วนที่สอง คือ Package declaration เพื่อประกาศ constant function และ procedure ที่มีอยู่ใน code นี้

ส่วนที่สาม คือ Package body ซึ่งจะมี HDL ที่ออกแบบไว้ในแต่ละ function และแต่ละ procedure

PROCEDURE

```
Procedure LdDataIn
(
    SetDataIn           : in    std_logic_vector( 3 downto 0 );
    signal  Clk         : in    std_logic;
    signal  CntLd       : out   std_logic;
    signal  DataIn      : out   std_logic_vector( 3 downto 0 )
) Is
Begin

    wait until rising_edge(Clk);
    CntLd       <= '1';
    DataIn      <= SetDataIn;
    wait until rising_edge(Clk);
    CntLd       <= '0';

End Procedure LdDataIn;
```

ประกาศชื่อ port ต่าง ๆ เพื่อติดต่อกับ testbench โดย ประเภทของสัญญาณที่นิยมใช้งานได้แก่

1) ไม่มีประเภท (SetDataIn) ใช้เมื่อรับค่าเข้ามาจาก testbench เป็นค่าคงที่ ไม่ผ่าน signal/variable ใด ๆ

2) signal ใช้เมื่อสัญญาณที่รับ/ส่งกับ testbench นั้น เป็นสัญญาณประเภท signal เช่น Clk, CntLd, DataIn

3) variable ใช้เมื่อสัญญาณที่รับ/ส่งกับ testbench นั้น เป็นสัญญาณประเภท variable (ไม่ได้แสดงในตัวอย่าง)

ส่วน direction ของสัญญาณจะมี 2 แบบคือ in และ out

รายละเอียดของ Procedure เขียนเหมือน HDL code ใน testbench

ในตัวอย่างจะลอก code ของ TM=1 ใน testbench ที่ใช้สำหรับทดสอบการโหลดค่าของ 4-bit counter มา

EXAMPLE PROCEDURE USAGE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.PkTest1.all;

Entity TbTest1 Is
End Entity TbTest1;

Architecture rtl of TbTest1 is
```

ตัวอย่างแสดงการเรียกใช้งาน Procedure ใน testbench มีขั้นตอนดังนี้

(1) ในส่วนที่ประกาศ library ให้เพิ่ม use work.<package name>.all;
เช่น use.work.PkTest1.all;

```
        ------------------------------------------
        -- TM=1 : Load input
        ------------------------------------------
        TM <= 1; wait for 1 ns;
        Report "TM=" & integer'image(TM);

--      wait until rising_edge(Clk);
--      CntLd       <= '1';
--      DataIn      <= "0110";
--      wait until rising_edge(Clk);
--      CntLd       <= '0';
        LdDataIn("0110",Clk,CntLd,DataIn);
```

(2) ใน Process ที่ใช้สร้างสัญญาณ input (u_Test) จะเรียกใช้ Procedure แทนการเขียน code โดยการเรียกใช้ Procedure จะต้องเรียงลำดับสัญญาณให้ตรงกับที่เราประกาศ ดังตัวอย่าง “0110” คือ SetDataIn และหลังจากนั้นจะ assign สัญญาณเพื่อเชื่อมกับ Clk CntLd และ DataIn ตามลำดับ
(ในตัวอย่างใช้ชื่อสัญญาณเดียวกัน เพื่อไม่ให้สับสน แต่ความเป็นจริง ชื่อสัญญาณใน testbench ไม่จำเป็นต้องเหมือนกับชื่อที่ประกาศไว้ใน Procedure)

RECORD

```
-- Package declaration
package example_package is
    type InFifo is record
        Full        : std_logic;                -- FIFO Full Flag
        Empty       : std_logic;                -- FIFO Empty Flag
        RdData      : std_logic_vector(7 downto 0);
    end record InFifo;

    type OutFifo is record
        RdEn        : std_logic;                -- FIFO Read Enable
        WrEn        : std_logic;                -- FIFO Write Enable
        WrData      : std_logic_vector(7 downto 0);
    end record OutFifo;

    Procedure WriteFifo
    (
        signal  ip  : in    InFifo;
        signal  op  : out   OutFifo
    );
end package example_package;
```

ปัญหา : เมื่อสัญญาณที่ต้องใช้ต่อกับ Procedure มีจำนวนมาก การเรียกใช้งาน Procedure จะไม่สะดวก เพราะต้องประกาศชื่อสัญญาณที่ใช้เชื่อมต่อกับ testbench ทุกครั้งที่ใช้งาน

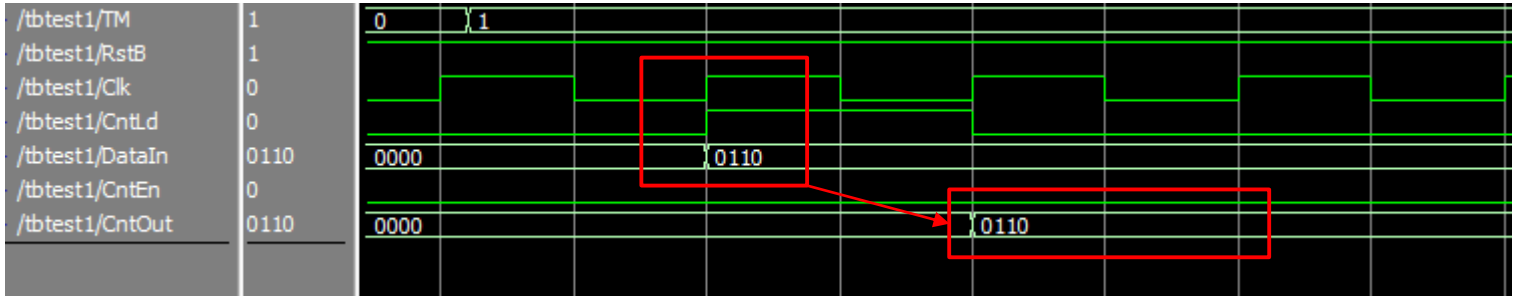
แก้ไข : ประกาศสัญญาณชนิดใหม่ โดยใช้ Record เพื่อจัดกลุ่มสัญญาณที่ต้องใช้งานติดต่อ
ตัวอย่าง สร้างสัญญาณชนิดใหม่ ชื่อ InFifo และ OutFifo ขึ้น

```
-- Package Body Section
package body example_package is
    Procedure WriteFifo
    (
        signal  ip  : in    InFifo;
        signal  op  : out   OutFifo
    ) Is
    Begin
        op.WrData   <= x"A2";
        op.WrEn     <= '1';
        wait until ip.Full = '1';
        op.WrEn     <= '0';
    End Procedure WriteFifo;
end package body example_package;
```

ใน Procedure ที่ต้องการใช้งาน record ก็จะประกาศชื่อ สัญญาณ และกำหนดชนิดของสัญญาณเป็น InFifo และ OutFifo
และเมื่อใช้งานเพื่อกำหนดค่า หรือ อ่านค่า จะต้องบอกชื่อสัญญาณ และตามด้วย “.” เพื่อระบุว่าจะเชื่อมกับสัญญาณใดในกลุ่ม record นั้น ๆ

3. CHECK OUTPUT

WAVEFORM MONITORING (1)

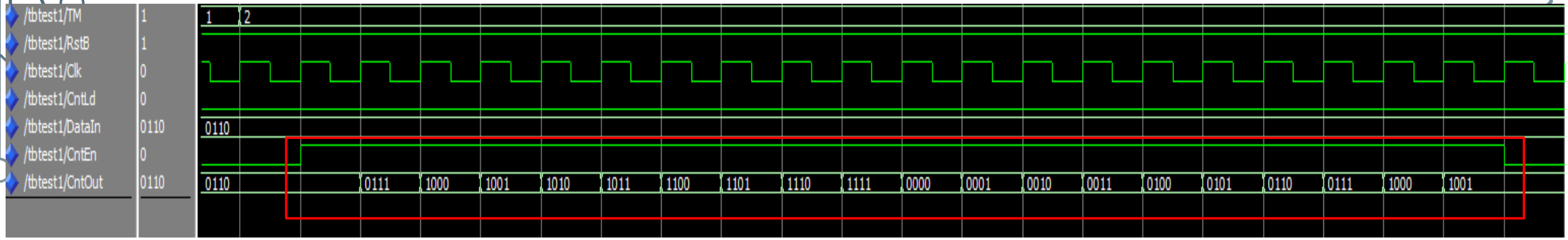


TM=1 ใช้ตรวจสอบฟังก์ชันว่า 4-bit counter สามารถโหลดค่าจากสัญญาณ DataIn เมื่อ CntLd='1' ได้จริงหรือไม่
จาก waveform พบว่า CntOut สามารถเปลี่ยนค่าจาก 0000 เป็น 0110 ได้จริง ถือว่า มีการทำงานพื้นฐานที่ถูกต้อง

เพิ่มเติม: หากต้องการทดสอบระบบให้เข้มข้นขึ้น ควรสร้างเงื่อนไขอื่น ๆ เพิ่มเติม เช่น

- ให้ค่า DataIn มีการเปลี่ยนค่าเป็นค่าอื่น โดย CntLd='0' อยู่ แล้วดูว่า CntOut มีการเปลี่ยนแปลงหรือไม่ ถ้าเปลี่ยน แสดงว่าวงจรเราทำงานผิด เพราะไปโหลดค่า DataIn ในช่วงที่ CntLd ยังมีค่าเป็น '0' อยู่
- ให้สัญญาณ CntLd มีความยาวมากกว่า 1 clock period แล้วเปลี่ยนค่า DataIn ไปเรื่อย ๆ ดูว่า CntOut เปลี่ยนแปลงตามขอบขาขึ้นของ Clk ทุกครั้งในช่วงที่ CntLd มีค่าเป็น '1' อยู่ไหม ถ้ามีช่วงไหนไม่เปลี่ยน แสดงว่าวงจรเราทำงานผิดแล้ว

WAVEFORM MONITORING (2)



TM=2 ใช้ตรวจสอบฟังก์ชันว่า 4-bit counter จะนับขึ้นเฉพาะช่วงที่ CntEn=‘1’ เท่านั้นหรือไม่

จาก waveform พบว่า CntOut สามารถจะนับขึ้น เริ่มจากค่า 0110 ไปเรื่อย ๆ และจบค่าที่ 1001 ตามจังหวะ CntEn ที่มีค่าเป็น ‘1’ ทั้งหมด 20 clock ได้ถูกต้อง

เพิ่มเติม: หากต้องการทดสอบระบบให้เข้มข้นขึ้น ควรสร้างเงื่อนไขอื่น ๆ เพิ่มเติม เช่น

- ให้สัญญาณ CntEn นั้นเปลี่ยนค่าเป็น ‘1’ และ ‘0’ สลับกันถี่ ๆ กว่านี้ แล้วดูว่า CntOut สามารถนับและหยุดได้ตรงตามจังหวะหรือไม่
- ให้สัญญาณ CntLd และ CntEn มีค่าเป็น ‘1’ พร้อมกัน แล้วดูว่า CntOut นั้นจะทำงานโหลดค่าจาก DataIn หรือนับขึ้น ในส่วนนี้ต้องตกลงกันตั้งแต่เริ่มออกแบบว่า จะให้การทำงานใดมี priority ที่สูงกว่า

WAVEFORM MONITORING PROBLEM

ข้อเสีย : หากสัญญาณ output มีมากกว่า 1 สัญญาณ การตรวจสอบการทำงานด้วยการดู waveform จะทำได้ยากขึ้น และผิดพลาดได้ง่าย เพราะต้องดูสัญญาณทุก clock ตั้งแต่เริ่มต้นการทำงานจนจบการทำงาน

แนวทางแก้ไข : ออกแบบวงจรเพิ่มเติม สำหรับการตรวจสอบค่า output โดยอาจจะเขียนเป็น Process เพิ่มแยกต่างหากไปเลย หรืออาจจะแทรกคำสั่ง Assert ที่ใช้ในการตรวจสอบค่าไปสลับกับการสร้างสัญญาณ input

ASSERT

```
u_Test : Process
variable    iExpCntOut  : std_logic_vector( 3 downto 0 );
Begin
    ------------------------------------------
    -- TM=0 : Reset module
    ------------------------------------------
    TM <= 0; wait for 1 ns;
    Report "TM=" & integer'image(TM);

    CntLd       <= '0';
    DataIn      <= "0000";
    CntEn       <= '0';
    wait for 20*tClk;

    ------------------------------------------
    -- TM=1 : Load input
    ------------------------------------------
    TM <= 1; wait for 1 ns;
    Report "TM=" & integer'image(TM);

    wait until rising_edge(Clk);
    CntLd       <= '1';
    DataIn      <= "0110";
    wait until rising_edge(Clk);
    CntLd       <= '0';

    wait for 1 ns;
    iExpCntOut  := DataIn;
    Assert (CntOut=iExpCntOut) Report "ERROR: CntOut"
    Severity Failure;
    wait for 20*tClk;
```

- ใช้เพื่อตรวจสอบค่าของสัญญาณว่ามีค่าถูกต้องหรือไม่
- ประกอบด้วย 3 ส่วนหลัก ๆ คือ Assert Report และ Severity ดังนี้

Assert condition Report string Severity severity_level;

หาก Cntout มีค่าไม่เท่ากับ iExpCntOut จะมีข้อความ ERROR เกิดขึ้น และ tool จะเบรกการทำงานทันทีจาก Severity Failure

เพิ่มเติม

- เพิ่ม wait for 1 ns; เพื่อรอให้ CntOut นั้นเปลี่ยนค่าหลังจากขอบของ clock ก่อนเสมือนใน hardware จริง เพียงแต่ใน tool จะไม่ได้แสดง delay นั้นให้เราเห็น
- ตัวแปร variable เป็นตัวแปรแบบ local ที่จะประกาศภายใน Process แต่ละตัว เพื่อใช้งานได้ภายใน Process เท่านั้น มักจะใช้งาน variable ในการออกแบบ testbench เพื่อสร้างสัญญาณเพื่อเปรียบเทียบกับ output

VARIABLE

VARIABLE

- เป็นตัวแปรแบบ local ที่จะประกาศภายใน Process แต่ละตัว เพื่อใช้งานได้ภายใน Process เท่านั้น ไม่ใช้ข้าม Process
- จะเปลี่ยนค่า ทันที เมื่อทำงานผ่านบรรทัดนั้นไป
- การกำหนดค่าจะใช้เครื่องหมาย :=
- นิยมใช้ในการเขียน testbench แต่ ไม่แนะนำ ให้ใช้งานในการออกแบบ hardware

SIGNAL

- เป็นตัวแปร global ที่ทุก Process ใน code จะใช้งานได้หมด
- จะไม่เปลี่ยนค่าทันทีที่ทำงานผ่านบรรทัดนั้น แต่จะเปลี่ยนเมื่อมี delay เกิดขึ้นในวงจรแล้ว ส่วนมากจะหลังผ่านคำสั่ง wait for หรือ wait until แล้ว
- การกำหนดค่าจะใช้เครื่องหมาย <=
- ใช้งานในการเขียน testbench และออกแบบ hardware

เพิ่มเติม

Signal ใน code ที่ออกแบบ hardware จะเป็นการระบุว่าเราต้องการให้ tool สร้างสัญญาณนี้ออกมาบน hardware จริง ๆ

Variable ใน code ที่ออกแบบ hardware จะเป็นการให้ tool ตัดสินใจเอาเองว่า สัญญาณนี้จะมีการสร้างออกมาจริง ๆ หรือเป็นแค่ตัวทดที่ไม่สร้างออกมาจริง ไม่แนะนำให้ใช้ variable ในการออกแบบ hardware เพราะจะ debug ยาก ไม่รู้ว่ามีสัญญาณนั้นอยู่หรือไม่

บทความเพิ่มเติม สำหรับวิธีการดู waveform ของสัญญาณประเภท variable

https://www.nandland.com/vhdl/tips/tip-viewing-variables-in-modelsim.html

FOR … LOOP

```
-----------------------------------------
-- TM=2 : Counter enable
-----------------------------------------
TM <= 2; wait for 1 ns;
Report "TM=" & integer'image(TM);

wait until rising_edge(Clk);

CntEn       <= '1';
For i in 0 to 19 loop
    iExpCntOut  := iExpCntOut + 1;
    wait until rising_edge(Clk);
    wait for 1 ns;
    Assert (CntOut=iExpCntOut) Report "ERROR: CntOut"
    Severity Failure;
End loop;
CntEn       <= '0';
wait for 20*tClk;

Report "### End Simulation ###" Severity Failure;
wait;
```

- TM=2 : แก้ไข code เพื่อเพิ่มจังหวะในการตรวจสอบ CntOut ในทุก ๆ clock โดยการเปลี่ยนจาก wait for เฉย ๆ มาเป็น For .. Loop เพื่อทำงาน 20 clock period แทน
- ในแต่ละ clock จะมีการเปลี่ยนค่า iExpCntOut โดยการเพิ่มค่าทีละ 1 เพื่อตรวจสอบว่า CntOut สามารถนับขึ้นได้อย่างถูกต้อง

เพิ่มเติม

- คำสั่ง For .. Loop ใช้ระบุจำนวนรอบที่จะทำงาน โดยตัวแปรหลังจาก For จะต้องเป็นตัวแปรใหม่ ที่ไม่เคยประกาศเป็น signal หรือ variable มาก่อน เพื่อใช้งานใน For … Loop เท่านั้น เป็นตัวแปรชนิด integer
- การออกจาก Loop จะเกิดขึ้นได้ 2 แบบ คือ ทำงานครบรอบตามที่กำหนดไว้ หรือเจอคำสั่ง Exit;

ADDITIONAL STATEMENT

LOOP + EXIT

```
u_Test : Process
variable    iExpCntOut  : std_logic_vector( 3 downto 0 );
variable    iTotalCnt   : std_logic_vector( 3 downto 0 );
Begin
```

```
iTotalCnt   := "0000";
Loop
    -- Run 10 rounds
    if ( iTotalCnt=10 ) then
        CntEn   <= '0';
        wait until rising_edge(Clk);
        Report "End of CntEn loop";
        exit;
    else
        CntEn   <= '1';
        wait until rising_edge(Clk);
    end if;
    iTotalCnt   := iTotalCnt + 1;
End Loop;
```

Loop + exit จะทำงานต่างจาก For … loop ตรงที่ การใช้ For … loop มักจะใช้เมื่อเรารู้จำนวนรอบที่เราจะทำงานที่ชัดเจนเลย แต่การใช้ Loop + exit นี้ จะใช้เมื่อเราไม่รู้จำนวน แต่เราจะออกจาก loop ก็ต่อเมื่อเจอเงื่อนไขอะไรบางอย่างแล้ว เช่น เจอสัญญาณ Finish ที่ระบุว่าการทำงานได้เสร็จสิ้นลงแล้ว ก็ออกจาก loop ได้ เป็นต้น

ตัวอย่างนี้ เป็นเพียงตัวอย่างเพื่อทำความเข้าใจการทำงานของคำสั่ง Loop + exit เท่านั้น จึงแสดงตัวอย่างของการใช้ Loop + exit เพื่อทำงานตามจำนวนรอบที่แน่นอน ผ่านตัวนับรอบคือสัญญาณ iTotalCnt โดยการทำงานแต่ละรอบ จะไปเพิ่มค่าของ iTotalCnt ทีละหนึ่ง จนเมื่อทำงานจบ ครบ 10 รอบ ก็จะเข้าเงื่อนไข if (iTotalCnt=10) แล้วเจอคำสั่ง Exit เป็นคำสั่งให้ออกจาก Loop นี้ไป

FOR… GENERATE

```
u_CntX0 : Process (Clk) Is
Begin
    if ( rising_edge(Clk) ) then
        if ( RstB = '0' ) then
            AryOfX(0) <= x"00";
        else
            AryOfX(0) <= AryOfX(0)+1;
        end if;
    end if;
End Process u_CntX0;

u_CntX1 : Process (Clk) Is
Begin
    if ( rising_edge(Clk) ) then
        if ( RstB = '0' ) then
            AryOfX(1) <= x"01";
        else
            AryOfX(1) <= AryOfX(1)+1;
        end if;
    end if;
End Process u_CntX1;

u_CntX2 : Process (Clk) Is
Begin
    if ( rising_edge(Clk) ) then
        if ( RstB = '0' ) then
            AryOfX(2) <= x"02";
        else
            AryOfX(2) <= AryOfX(2)+1;
        end if;
    end if;
End Process u_CntX2;
```

```
u_CounterArray : For i in 0 to 2 Generate

    u_CntX : Process (Clk) Is
    Begin
        if ( rising_edge(Clk) ) then
            if ( RstB = '0' ) then
                AryOfX(i) <= conv_std_logic_vector(i, 8);
            else
                AryOfX(i) <= AryOfX(i)+1;
            end if;
        end if;
    End Process u_CntX;

End Generate u_CounterArray;
```

For … Generate เป็นคำสั่งที่ช่วยลดความยาวของ code ซ้ำ ๆ ลงได้ ดังตัวอย่างด้านซ้ายคือแทนที่เราจะเขียน process เดิม ซ้ำกันทั้งหมด 3 รอบ เราใช้ For … Generate เพื่อแจ้งให้ tool รู้ว่า เราต้องการทำงานแบบนี้ทั้งหมด 3 สัญญาณ พร้อม ๆ กัน

เรามักจะเจอ For … Generate ในการเรียกใช้ component ตัวหนึ่ง กับสัญญาณหลาย ๆ bit

FOR… GENERATE VS FOR … LOOP

For … Generate

- เป็นการสร้างวงจรหลาย ๆ ตัว โดยทุกตัวทำงานพร้อม ๆ กัน และใช้งานเพื่อจุดประสงค์ในการลดความยาว code ลงเท่านั้น
- สามารถนำไปสังเคราะห์เป็น hardware หลาย ๆ ตัวได้จริง
- มักจะเจอใน HDL code ที่ใช้ออกแบบ hardware จริง

For … Loop

- เป็นการวนการทำงานของสัญญาณ ๆ หนึ่ง ให้ทำงานทีละรอบ โดยการทำงานรอบที่ 2 จะขึ้นกับผลลัพธ์ของรอบแรก และวนทำงานจนครบ
- การวนลูป ไม่สามารถสังเคราะห์เป็น hardware ได้จริง
- มักจะใช้งานใน HDL code ของ testbench เพื่อทำงาน simulation เท่านั้น

FUNCTION CONV_INTEGER

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
signal a_std : std_logic_vector(7 downto 0);
signal a_int : integer  range 0 to 255;
.
.
.
a_std   <= x"A2";
a_int   <= conv_integer(a_std); -- a_int = 162
```

- เป็นฟังก์ชันที่อยู่ใน library ชื่อ std_logic_arith ที่เราประกาศเรียกใช้ไว้ช่วง header ของ HDL code
- ใช้สำหรับแปลงสัญญาณชนิด std_logic_vector ให้เป็นชนิด integer
- จะรับ input 1 ค่า คือสัญญาณชนิด std_logic_vector เช่น ในตัวอย่างคือ สัญญาณ a_std ขนาด 8-bit และจะได้ output ออกมาเป็นสัญญาณชนิด integer เช่น ตัวอย่างคือ สัญญาณ a_int
- สัญญาณ output และ input ควรมีขนาดที่ match กัน กล่าวคือ integer range ควรเป็น 0 ถึง (2^n – 1) ดังในตัวอย่าง เมื่อ input มีขนาด 8-bit สัญญาณ output ที่มารองรับ function ประกาศเป็น range 0 to 255

FUNCTION CONV_STD_LOGIC_VECTOR

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
signal a_std : std_logic_vector(7 downto 0);
signal a_int : integer  range 0 to 255;
.
.
.
a_int   <= 162;
a_std   <= conv_std_logic_vector(a_int, 8);  -- a_std = x"A2"
```

- เป็นฟังก์ชันที่อยู่ใน library ชื่อ std_logic_arith ที่เราประกาศเรียกใช้ไว้ช่วง header ของ HDL code
- ใช้สำหรับแปลงสัญญาณชนิด integer ให้กลายเป็นชนิด std_logic_vector แทน
- จะรับ input ทั้งหมด 2 ค่า ค่าแรกคือ สัญญาณชนิด integer ที่เราต้องการแปลง เช่น a_int และค่าที่สองคือ จำนวน bit ของสัญญาณ std_logic_vector ที่เป็น output เช่น ในตัวอย่างคือทั้งหมด 8 – bit
- สัญญาณ output และ input ควรมีขนาดที่ match กัน กล่าวคือ output ต้องรองรับค่าของ integer ทั้งหมดได้ อย่างในตัวอย่าง เมื่อ integer มีค่า 0-255 ก็ควรใช้ output ที่มีขนาดอย่างน้อย 8-bit

EXAMPLE ARRAY + TYPE DECLARATION

```
type    array8ofu8  is array (0 to 7) of std logic vector( 7 downto 0 );
signal  rTestData   : array8ofu8 :=
(
    "00000001", -- Initial value of rTestData(0)
    "00000010", -- Initial value of rTestData(1)
    "00000100", -- Initial value of rTestData(2)
    "00001000", -- Initial value of rTestData(3)
    "00010000", -- Initial value of rTestData(4)
    "00100000", -- Initial value of rTestData(5)
    "01000000", -- Initial value of rTestData(6)
    "10000000"  -- Initial value of rTestData(7)
);
```

- array จะใช้เพื่อรวมกลุ่มของสัญญาณชนิดเดียวกัน แล้วอ้างอิงผ่าน index แบบ integer เพื่อความสะดวกในการเขียน code และเรียกใช้งาน
- มักจะใช้งานเพื่อการสร้าง RAM หรือใช้คู่กับ For … Generate

เมื่อต้องการสร้างสัญญาณที่เป็น array สามารถประกาศเป็น signal พร้อมระบุชนิดของสัญญาณเป็นชนิด array8ofu8 ที่เราเพิ่งตั้งใหม่ขึ้นมาได้เลย ในตัวอย่างหลังจากประกาศแล้ว มีการใช้ := เพื่อระบุค่าเริ่มต้นของสัญญาณไว้ โดยลำดับของสัญญาณจะเป็นการกำหนดค่าจาก index = 0 ไปจนตัวสุดท้ายคือ index = 7

การใช้งาน array จะมาคู่กับการประกาศสัญญาณชนิดใหม่ขึ้น ผ่านคำสั่ง type ดังตัวอย่างด้านขวา เป็นการประกาศกลุ่มสัญญาณขนาด 8-bit ทั้งหมด 8 ตัว ผ่านการใช้ array (0 to 7) โดยสร้างชนิดสัญญาณใหม่ชื่อว่า array8ofu8

EXAMPLE USAGE OF ARRAY

```
Component Test1 Is
Port
(
    RstB        : in    std_logic;
    Clk         : in    std_logic;

    CntLd       : in    std_logic;          (1)
    DataIn      : in    std_logic_vector( 3 downto 0 );   (2)
    CntEn       : in    std_logic;          (3)

    CntOut      : out   std_logic_vector( 3 downto 0 )    (4)
);
End Component Test1;
```

```
u_TestAll : For i in 0 to 7 Generate
    u_Test1 : Test1
    Port map
    (
        RstB        => RstB         ,
        Clk         => Clk          ,

        CntLd       => CntLdAll(i)  ,   (1)
        DataIn      => DataInAll(i) ,   (2)
        CntEn       => CntEnAll(i)  ,   (3)

        CntOut      => CntOutAll(i)     (4)
    );
End Generate u_TestAll;
```

ตัวอย่าง เป็นการเรียกใช้วงจร 4-bit counter พร้อมกันทั้งหมด 8 ตัว โดยต้องการเรียกผ่าน For...Generate

```
type    array8ofu4  is array (0 to 7) of std_logic_vector( 3 downto 0 );
signal  CntLdAll    : std_logic_vector( 7 downto 0 );   (1)
signal  DataInAll   : array8ofu4;                       (2)
signal  CntEnAll    : std_logic_vector( 7 downto 0 );   (3)
signal  CntOutAll   : array8ofu4;                       (4)
```

สัญญาณขนาด 1-bit สามารถขยายเป็น 8 ชุด ได้โดยขยายเป็น std_logic_vector(7 downto 0) และใช้ i เป็น index ในการอ้างอิงสัญญาณของแต่ละชุด เช่น CntEnAll(5) จะเป็น CntEn ของ Counter ชุดที่ 5

1/3

ส่วนสัญญาณขนาด 4-bit จะขยายเป็น 8 ชุด ได้โดยการใช้ array ช่วย และใช้ i เป็น index ในการอ้างอิงสัญญาณของแต่ละชุด เช่น DataInAll(0) จะหมายถึงสัญญาณ DataIn ของ counter ชุดที่ 0

2/4

EXAMPLE TWO-PORT RAM (1 WRITE PORT, 1 READ PORT)

```
Entity Ram16x8 Is
    Port
    (
        Clk             : in    std_logic;
        WrEn            : in    std_logic;
        WrAddr          : in    std_logic_vector( 3 downto 0 );
        WrData          : in    std_logic_vector( 7 downto 0 );
        RdAddr          : in    std_logic_vector( 3 downto 0 );
        RdData          : out   std_logic_vector( 7 downto 0 )
    );
End Entity Ram16x8;
```

Write Port

Read Port

ตัวอย่างการสร้าง RAM ขนาด 16 x 8 –bit (depth=16, databus=8 bit)

- Address สำหรับ Write (WrAddr) และ Address สำหรับ Read (RdAddr) จะมีขนาด 4 bit เพื่ออ้างอิงข้อมูลทั้ง 16 ตัว
- Data สำหรับ Write (WrData) และ สำหรับ Read (RdData) มีขนาด 8 bit ซึ่งเป็นขนาดของข้อมูลในแต่ละตำแหน่ง
- การเขียนและอ่าน สามารถทำได้พร้อมกัน เพราะมี address และ data แยกกันโดยเด็ดขาด แต่ต้องระวังช่วงที่ Write และ Read ไปที่ Address เดียวกัน เพราะข้อมูลที่อ่านในช่วงเวลานั้น จะเพี้ยนได้ (อาจจะได้ค่าใหม่ หรือค่าเก่า)

SIGNAL DECLARATION OF RAM16X8 HDL

```
Architecture rtl Of Ram16x8 Is

-------------------------------------------------------------------------------
-- Signal declaration
-------------------------------------------------------------------------------

    type    RamType is array (0 to 15) of std_logic_vector( 7 downto 0 );
    signal  Ram16x8     : RamType :=
    (
        "11111110",
        "11111101",
        "11111011",
        "11110111",
        "11101111",
        "11011111",
        "10111111",
        "01111111",
        "00000001",
        "00000010",
        "00000100",
        "00001000",
        "00010000",
        "00100000",
        "01000000",
        "10000000"
    );

    signal  rRdData     : std_logic_vector( 7 downto 0 );
```

- การประกาศสัญญาณภายใน RAM16x8 จะใช้ array เพื่ออ้างอิงถึงตำแหน่งทั้ง 16 ที่
- ในตัวอย่าง มีการกำหนดค่าเริ่มต้นของข้อมูลทั้ง 16 ตำแหน่ง โดยบรรทัดแรก “11111110” เป็นการกำหนดค่าเริ่มต้นให้ Ram16x8(0) หรือค่าที่ Address = 0 ส่วนบรรทัดต่อ ๆ ไปเป็น Address = 1, 2, …, 15 ตามลำดับ (บรรทัดสุดท้ายคือ “10000000” คือค่าเริ่มต้นของ Ram16x8(15)
- rRdData สร้างขึ้นเพื่อใช้รับค่าที่อ่านได้จาก RAM

RAM16X8 HDL DESIGN

```
Begin

-------------------------------------------------------------
-- Output assignment
-------------------------------------------------------------

    RdData(7 downto 0)  <= rRdData(7 downto 0);

-------------------------------------------------------------
-- DFF
-------------------------------------------------------------

    u_RamWr : Process (Clk) Is
    Begin
        if ( rising_edge(Clk) ) then
            if ( WrEn='1' ) then
                Ram16x8(conv_integer(WrAddr))    <= WrData;
            end if;
        end if;
    End Process u_RamWr;

    u_RamRd : Process (Clk) Is
    Begin
        if ( rising_edge(Clk) ) then
            rRdData <= Ram16x8(conv_integer(RdAddr));
        end if;
    End Process u_RamRd;

End Architecture rtl;
```

RamWr เป็น Process สำหรับการเขียนข้อมูลจาก WrData ไปที่ Ram16x8 เมื่อสัญญาณ WrEn=‘1’ ตามจังหวะการทำงานของ clock (เป็น RAM แบบ synchronous) ดังนั้นข้อมูลจะถูกเขียนเมื่อเจอขอบขาขึ้นของ Clk และ WrEn มีค่าเป็น ‘1’ เท่านั้น

conv_integer ใช้เพื่อแปลงสัญญาณ WrAddr ที่เป็นชนิด std_logic_vector ให้กลายเป็น integer เพื่อใช้เป็น index ชี้ตำแหน่งที่ต้องการเขียนของ Ram16x8

RamRd เป็น Process สำหรับการอ่านข้อมูลไปเก็บไว้ที่ rRdData โดยจะอ่านแบบ synchronous เช่นกัน คือ รอจังหวะขอบขาขึ้นของ Clk ก่อน แล้วจึงค่อยทำงาน เช่นเดียวกับ Write คือมีการเรียกใช้ conv_integer เพื่อแปลงสัญญาณ RdAddr ให้กลายเป็น integer เพื่อใช้เป็น index สำหรับการอ่านค่าจาก Ram16x8

ADDITIONAL INFO FOR SIMULATION

- การออกแบบ testbench ส่วนใหญ่ตัว code จะอยู่ภายใต้ Process ใหญ่ ๆ ไม่กี่ Process และการทดสอบวงจร จะค่อย ๆ ทดสอบทีละฟังก์ชัน ไล่จากบรรทัดบนสุด ไปบรรทัดล่างสุดตามลำดับ และเรียงการทำงานตามช่วงเวลาไป
- มุมมองของ testbench จึงไปคล้ายกับการเขียนโปรแกรมบนคอมพิวเตอร์ที่เราคุ้นเคยกัน คือ เรียงลำดับไปทีละคำสั่ง
- การเขียน testbench ที่ดี จึงมีหลักการคล้ายกับการเขียนโปรแกรมทั่วไป มากกว่าการออกแบบ hardware คือ ควรรู้จักคำสั่งต่าง ๆ ให้มากไว้ เพื่อจะออกแบบ code ให้กระชับ และมีประสิทธิภาพที่ดี สามารถอ่าน เข้าใจได้ง่าย และยืดหยุ่นสำหรับการเรียกใช้งานหลาย ๆ แบบ รวมถึงการเอากลับมาเรียกใช้งานได้ในงานที่ต่าง ๆ กัน

Q & A

https://www.facebook.com/DigitalDesignThailand/

https://forfpgadesign.wordpress.com/