บทที่ 1
บทนำ

หนังสือเล่มนี้ครอบคลุมการออกแบบวงจรดิจิตอลด้วยภาษา Verilog HDL ซึ่ง HDL (Hardware Description Language) เป็นวิธีการออกแบบฮาร์ดแวร์ดิจิตอลด้วยวิธีการทางซอฟท์แวร์ ข้อดีของการออกแบบระบบด้วย HDL คือ ระยะเวลาการออกแบบสั้นลง ทำให้ลดเวลาในการผลิตสินค้าออกสู่ตลาด ข้อดีอีกประการคือ สามารถจำลองการทำงาน ทดสอบการทำงานให้ถูกต้องก่อนการสร้างบนฮาร์ดแวร์จริง ความผิดพลาด (errors) ที่ถูกพบในระหว่างการจำลองการทำงานนั้น สามารถถูกแก้ไขให้ถูกต้องได้ก่อนที่จะสร้างวงจรบนฮาร์ดแวร์ราคาแพง

1.1 ประวัติภาษา HDL

HDL ได้รับความนิยมในทศวรรษที่ 1980 และถูกใช้ในการออกแบบระบบดิจิตอลขนาดใหญ่โดยการเขียนบรรยายแทนการวาดผังวงจร HDL ทำให้การออกแบบระบบขนาดใหญ่ง่ายขึ้นด้วยการบรรยายแนวคิดเชิงสถาปัตยกรรมได้ชัดเจนโดยไม่ต้องวาดผังวงจร เนื่องด้วยความก้าวหน้าของเทคโนโลยีทางด้าน ASIC FPGA และ CPLD ทำให้เทคนิคทางคอมพิวเตอร์ช่วยออกแบบมีความจำเป็นมากขึ้น นักออกแบบสามารถใช้โปรแกรมภาษาเพื่อออกแบบและจำลองระบบ ในการนี้ Test Benches สามารถจำลองการทำงานทั้งระบบเพื่อให้ได้เอาท์พุทไบนารีและรูปคลื่น

ภาษา Verilog HDL เป็นภาษาบรรยายฮาร์ดแวร์ที่ได้รับความนิยมอย่างกว้างขวาง ภาษา VHDL ก็เป็นภาษาบรรยายฮาร์ดแวร์อีกภาษาหนึ่งที่ได้รับความนิยมเช่นกัน ทั้งสองภาษามีมาตรฐาน IEEE รองรับ ในปัจจุบันภาษา Verilog HDL ได้รับความนิยมอย่างกว้างขวางกว่าในวงการอุตสาหกรรมเนื่องจากมีความคล้ายคลึงกับภาษา C

1.2 Verilog HDL

Verilog HDL เป็นเทคนิคทันสมัยสำหรับการออกแบบวงจรดิจิตอลและระบบคอมพิวเตอร์ มีลักษณะคล้ายภาษา C ร่วมกับไวยากรณ์บางอย่างของภาษา Pascal สามารถโมเดลระบบดิจิตอลได้หลายระดับ จากระดับเกท ระบบดิจิตอลที่ซับซ้อน จนถึงระดับคอมพิวเตอร์เมนเฟรม การผสมผสานระหว่างภาษา C กับไวยากรณ์บางของภาษา Pascal ไว้ด้วยกันทำให้ภาษา Verilog นี้ง่ายต่อการเรียนรู้ ทำให้ Verilog HDL ได้รับความนิยมมากที่สุดในวงการอุตสาหกรรม

Verilog HDL สามารถบรรยายได้ทั้งวงจรเชิงจัดหมู่ (combinational circuit) และวงจรเชิงลำดับ (sequential circuit) รวมถึงอุปกรณ์เก็บความจำที่ทำงานทั้งที่ระดับสัญญาณ (level-sensitive) และที่ขอบสัญญาณ (edge-triggered) Verilog สามารถบรรยายความสัมพันธ์ระหว่างไวยากรณ์ของภาษากับฮาร์ดแวร์เชิงกายภาพได้อย่างชัดเจน

1.3 การแสดงระดับสัญญาณ

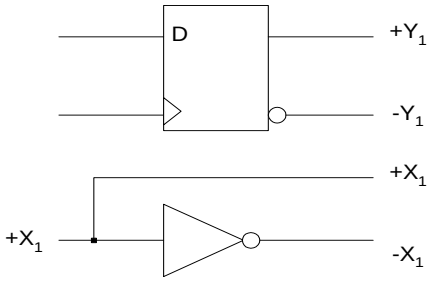
การแสดงระดับการทำงานของสัญญาณมีด้วยกันหลายวิธี ตารางที่ 1.1 แสดงรายการแสดงระดับการทำงานของสัญญาณด้วยวิธีต่าง ๆ เช่น เครื่องหมายบวก (+) และ ลบ (-) ที่อยู่หน้าตัวแปลหมายถึงระดับของ

แรงดันสูง (high) และ ต่ำ (low) ตามลำดับ ซึ่งแสดงถึงการทำงานที่ระดับสูง (active high) ของสัญญาณ หรือการทำงานที่ระดับต่ำของสัญญาณ (active low) ในเชิงตรรกะก็คือ ลอจิก 1 ซึ่งหมายถึง จริง (true) และ ลอจิก 0 ซึ่งหมายถึง เท็จ (false) นั่นเอง

ตารางที่ 1.1 การแสดงระดับสัญญาณ

การแสดงการทำงานที่ระดับสูง	+A	A	A(H)	A	A	A
การแสดงการทำงานที่ระดับต่ำ	-A	¬A	A(L)	*A	$\overline{A}$	A'

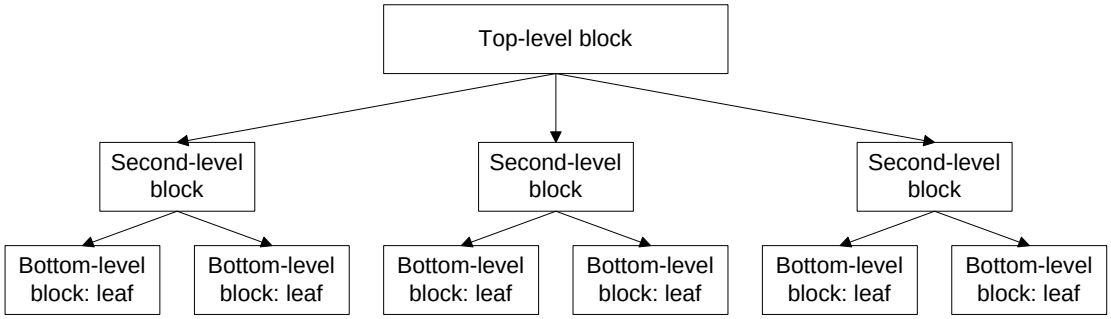
สัญญาณจะถูกป้อนเป็นบวกหรือลบนั้น ขันกับสภาวะการทำงาน ณ จุดที่พิจารณา สัญญาณยังสามารถทำงานที่ระดับสูงและระดับต่ำได้ในเวลาเดียวกันดังแสดงในรูปที่ 1.1



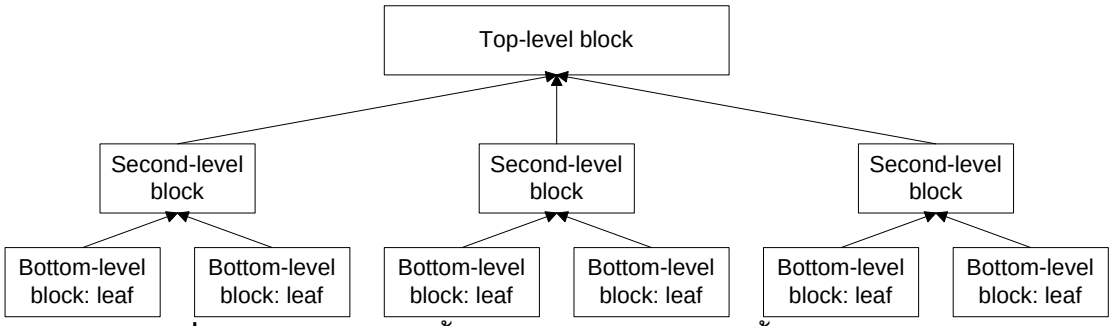
รูปที่ 1.1 สัญญาณสามารถทำงานที่ระดับสูงและต่ำได้ในเวลาเดียวกัน

1.4 ระเบียบวิธีการออกแบบ

ระเบียบวิธีการออกแบบมี 2 วิธีหลักคือ การออกแบบบนลงล่าง (top-down design) และการออกแบบล่างขึ้นบน (bottom-up design) รูปที่ 1.2 แสดงแผนผังเชิงลำดับชั้นของการออกแบบจากบนลงล่าง บล็อกด้านบนสุด (top-level block) จะถูกออกแบบก่อน จากนั้นบล็อกด้านล่างถัดลงมาจึงจะถูกออกแบบ ขั้นตอนนี้ถูกกระทำซ้ำจนกว่าทุกระดับในโครงสร้างถูกออกแบบหมด บล็อกล่างสุด (bottom-level block) เป็นส่วนที่ไม่สามารถแบ่งย่อยได้อีกแล้ว สามารถถูกพิจารณาให้เป็นเซลล์ใบไม้ (leaf cell) ของโครงสร้างต้นไม้ (tree structure) รูปที่ 1.3 แสดงแผนผังเชิงลำดับชั้นของการออกแบบจากล่างขึ้นบน ซึ่งเซลล์ใบไม้จะถูกออกแบบก่อน จากนั้นบล็อกด้านบนถัดขึ้นไปจึงจะถูกออกแบบ และถูกกระทำซ้ำจนกว่าจะถึงบล็อกด้านบนสุด

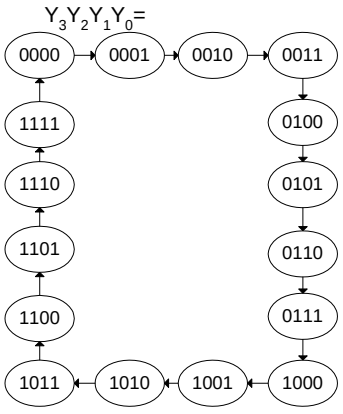


รูปที่ 1.2 แผนผังเชิงลำดับชั้นของการออกแบบจากบนลงล่าง



รูปที่ 1.3 แผนผังเชิงลำดับชั้นของการออกแบบจากล่างขึ้นบน

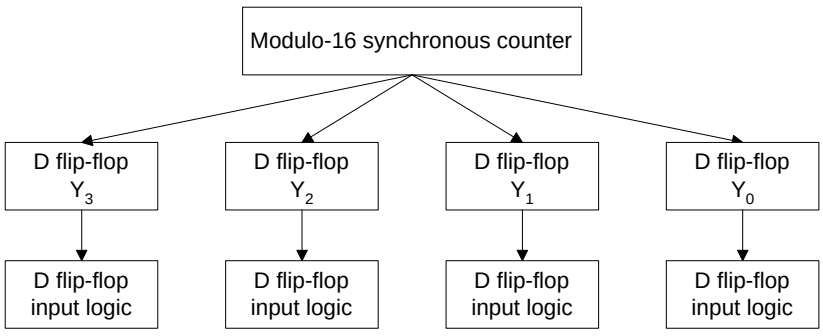
ตัวอย่างการออกแบบบนลงล่างได้แก่ การออกแบบวงจรนับมอดูโล-16 ดังรูปที่ 1.4 กำหนดให้ตัวแปรเสตทคือ $Y_3Y_2Y_1Y_0$ โดยใช้ D flip-flop เป็นตัวเก็บสถานะ แผนผังเชิงลำดับชั้นของวงจรนับมอดูโล-16 จึงเป็นดังรูปที่ 1.5 จากนั้นวงจรอินพุทลอจิกให้กับ D flip-flop แต่ละตัวสามารถถูกสังเคราะห์ได้จากแผนภาพคาร์นอดังรูปที่ 1.6 และได้วงจรดังรูปที่ 1.7 คราวนี้ก็สามารถวาดแผนผังเชิงลำดับชั้นของวงจรนับมอดูโล-16 ที่แสดงรายละเอียดวงจรได้ดังรูปที่ 1.8



State table

$Y_3Y_2Y_1Y_0$	$Y'_3Y'_2Y'_1Y'_0$
0000	0001
0001	0010
0010	0011
0011	0100
0100	0101
0101	0110
0110	0111
0111	1000
1000	1001
1001	1010
1010	1011
1011	1100
1100	1101
1101	1110
1110	1111
1111	0000

รูปที่ 1.4 ลำดับการนับมอดูโล-16



รูปที่ 1.5 แผนผังเชิงลำดับชั้นของวงจรนับมอดูโล-16

Y_3Y_2 \ Y_1Y_0	00	01	11	10
00	0	0	0	0
01	0	0	1	0
11	1	1	0	1
10	1	1	1	1

$DY_3=Y_3Y_2' +Y_3Y_1' +Y_3Y_0' +Y_3' Y_2Y_1Y_0$

Y_3Y_2 \ Y_1Y_0	00	01	11	10
00	0	0	1	0
01	1	1	0	1
11	1	1	0	1
10	0	0	1	0

$DY_1=Y_1Y_0' +Y_1' Y_0$

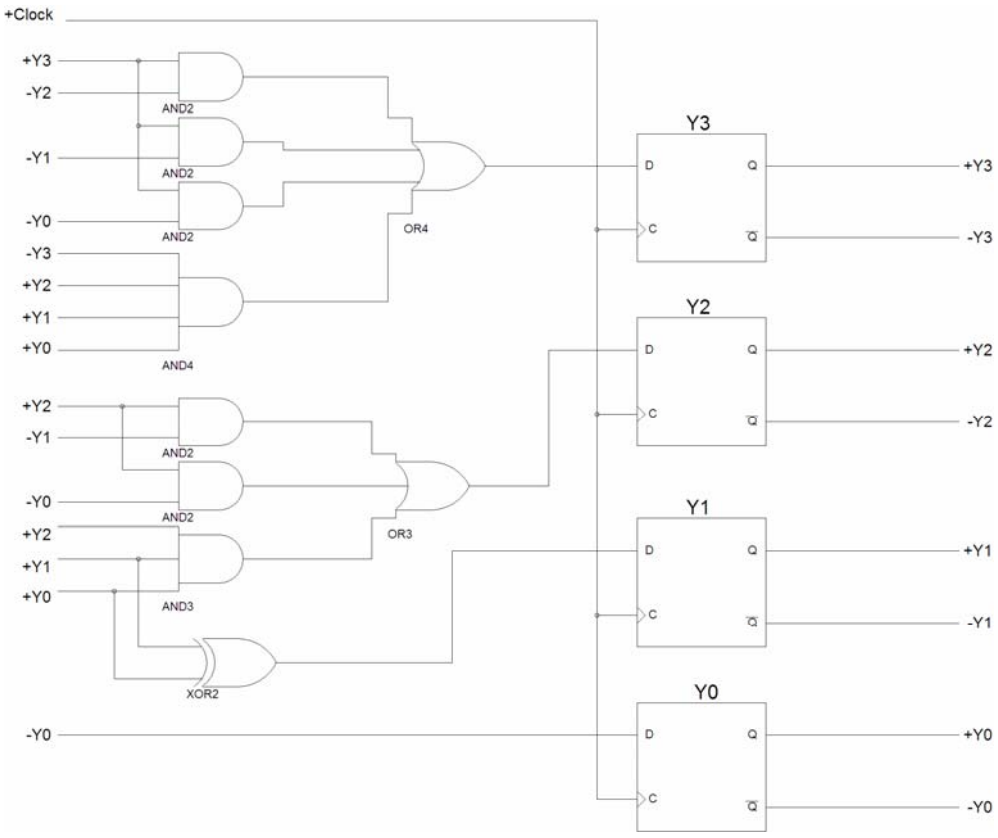
Y_3Y_2 \ Y_1Y_0	00	01	11	10
00	0	0	1	0
01	1	1	0	1
11	1	1	0	1
10	0	0	1	0

$DY_2=Y_2Y_1' +Y_2Y_0' +Y_2' Y_1Y_0$

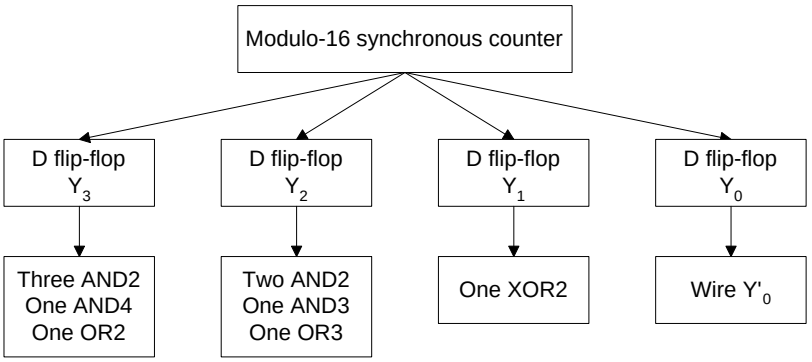
Y_3Y_2 \ Y_1Y_0	00	01	11	10
00	0	0	1	0
01	1	1	0	1
11	1	1	0	1
10	0	0	1	0

$DY_0=Y_0'$

รูปที่ 1.6 แผนคาร์นอแสดงการสังเคราะห์วงจรนับมอดูโล-16



รูปที่ 1.7 แผนผังลอจิกของวงจรนับมอดูโล-16



รูปที่ 1.8 แผนผังเชิงลำดับชั้นของรายละเอียดการสร้างวงจรนับมอดูโล-16

1.5 โมดูลและพอร์ท

โมดูล (module) เป็นหน่วยพื้นฐานของการออกแบบด้วยภาษา Verilog โมดูลอธิบายถึงฟังก์ชันการทำงานของวงจรต่างๆ อาจจะเป็น ลอจิกเกท วงจรบวก วงจรคูณ วงจรนับ หรือวงจรลอจิกอื่นๆ สามารถเป็นโมดูลเดี่ยวได้ หรือเป็นโมดูลที่รวมหลายโมดูลเข้าด้วยกัน และสามารถประกอบด้วยโมดูลชนิดเดียวกันหลายตัว โมดูลเหล่าถูกนี้เรียกว่า instantiation ซึ่งเป็นโมดูลระดับล่างที่ประกอบกันเป็นโมดูลระดับที่สูงขึ้น

โมดูลประกอบด้วยการประกาศ อินพุท เอาท์พุท ตัวแปร และฟังก์ชันการทำงาน มีคำหลักคือ **module** เพื่อเริ่ม และ **endmodule** เพื่อจบการสร้างโมดูล ดังโครงสร้างทั่วไปในรูปที่ 1.9 ภายในโมดูลมีช่องทาง (ports) ไว้สำหรับสื่อสารกับโมดูลอื่นภายนอก รูปที่ 1.10 แสดงการสร้างโมดูลสำหรับเกท AND ซึ่งมีช่องทางเข้า (input port) คือ x1 และ x2 และช่องทางออก (output port) คือ z1 คำหลัก **input** และ **output** ใช้สำหรับประกาศช่องทางเข้าและช่องทางออกตามลำดับ คำหลัก **wire** ใช้สำหรับการประกาศสายสัญญาณภายโมดูล คำหลัก **assign** ใช้สำหรับอธิบายความสัมพันธ์ของสัญญาณต่างในวงจร ในที่นี้โอเปอเรเตอร์ **&** หมายถึงการแอนด์กันของสัญญาณ

```
module <module name> (port list);
    declarations
        input, output,
        reg, wire, parameter, …
            …
    <module internals>
        statements
        initial, always, module instantiation, …
            …
endmodule
```

รูปที่ 1.9 โครงสร้างทั่วไปของ Verilog module

```
//dataflow and gate with two inputs
module and2_df (x1, x2, z1);
    input     x1, x2;
    output    z1;

    wire      x1, x2;
    wire      z1;

    assign    z1 = x1 & x2;

endmodule
```

รูปที่ 1.10 Verilog module สำหรับเกท AND แบบ 2 อินพุท

เมื่อสร้างโมดูลของฟังก์ชันที่ต้องการแล้ว ในภาษา Verilog ยังสามารถรองรับการออกแบบโมดูลที่ใช้สำหรับการจำลองแบบเพื่อทดสอบการทำงานของโมดูลดังกล่าวได้ด้วย เรียกโมดูลสำหรับการทดสอบนี้ว่า test bench รูปที่ 1.11 แสดง test bench อย่างง่ายสำหรับเกท AND ของรูปที่ 1.10

บรรทัดที่ 2 แสดงการประกาศชื่อโมดูลสำหรับการจำลองแบบ **ซึ่งต้องไม่ซ้ำกับชื่อโมดูลที่สร้างไว้** แต่ควรเป็นชื่อที่บ่งถึง test bench ของโมดูลที่จะทดสอบด้วย

อินพุทถูกกำหนดให้เป็นตัวแปรชนิดแบบรีจิสเตอร์โดยใช้คำหลัก **reg** ดังบรรทัดที่ 3 ซึ่งหมายความว่าค่าของตัวแปรต้องคงค่าจนกว่าค่าใหม่จะเข้ามา เอาท์พุทถูกกำหนดเป็นชนิดสายสัญญาณโดยคำหลัก **wire** ซึ่งค่าของเอาท์พุทขึ้นกับอินพุท จึงไม่จำเป็นต้องใช้รีจิสเตอร์

Verilog สามารถมอนิเตอร์ดูการเปลี่ยนแปลงของสัญญาณได้โดยใช้คำหลัก **$monitor** ดังบรรทัดที่ 7 ตัวแปรที่ถูกระบุในวงเล็บก็จะถูกแสดงออกมาเป็นไบนารีด้วย %b (ใช้ %o สำหรับเลขฐาน 8 %h สำหรับเลขฐาน 16 และ %d สำหรับเลขฐาน 10) มีคำหลัก **initial** หมายถึงการกระทำคำสั่งเพียงครั้งเดียว ในที่นี้ก็คือแสดงค่าตัวแปรเพียงครั้งเดียวเมื่อมีการเปลี่ยนแปลง

ในการจำลองแบบจำเป็นต้องมีการป้อนค่าอินพุทให้กับโมดูลที่จะทดสอบ ค่าอินพุทที่ป้อนในแต่ละช่วงเวลาถูกเรียกว่า เวกเตอร์อินพุท การป้อนค่าเวกเตอร์อินพุททำได้ดังนี้ บรรทัดที่ 9 มี **initial** อีกครั้งเพื่อให้คำสั่งที่อยู่ระหว่าง **begin…end** ถูกกระทำเพียงครั้งเดียว บรรทัดที่ 11 และ 12 เป็นการสั่งให้ ที่เวลา 0 (ซึ่งแทนด้วย #0) ป้อนค่าอินพุท x1 และ x2 เป็น 0 (ซึ่งใช้ 1'b เป็นการระบุค่าไบนารี 1 บิท) บรรทัดที่ 13 และ 14 #10 หมายถึงป้อนค่าอินพุทใหม่เมื่อเวลาผ่านไป 10 หน่วยเวลา บรรทัดที่ 19 คำหลัก **$stop** หมายถึงหยุดการป้อนค่าเวกเตอร์อินพุท

สุดท้ายใน test bench จำเป็นต้องมีการระบุโมดูลที่จะทดสอบ โดยการใช้ instantiation เรียกใช้โมดูลดังกล่าว บรรทัดที่ 22-26 แสดงการเรียกใช้โมดูล and2 พร้อมกับการระบุการเชื่อมต่อสายสัญญาณ โดยค่าในวงเล็บเป็นตัวแปรที่ถูกประกาศในโมดูล test bench นี้

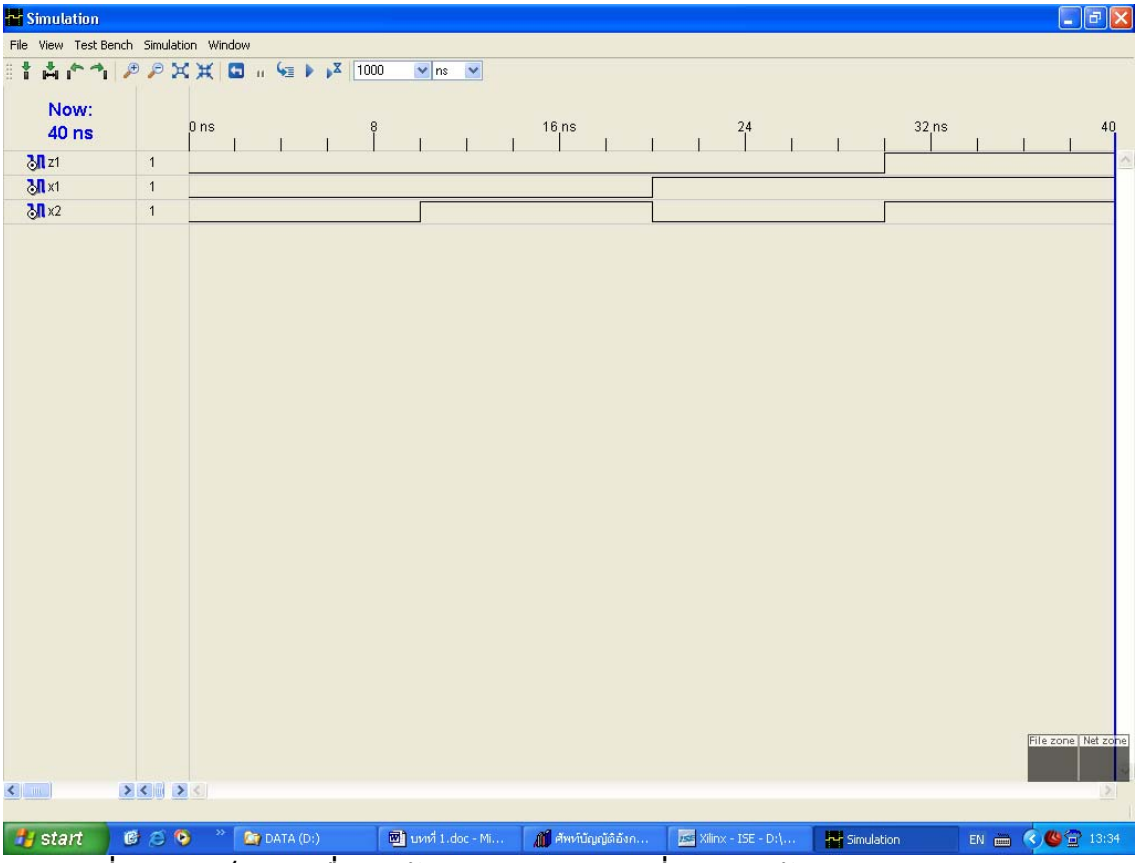
ผลการจำลองแบบการทำงานสามารถแสดงเป็นเอาท์พุทไบนารีดังรูปที่ 1.12 หรือรูปคลื่นสัญญาณดังรูปที่ 1.13 ก็ได้

```
1    //and2 test bench
     module and2_df_tb;
     reg      x1, x2;
     wire     z1;
5    //display  variables
     initial
     $monitor ("x1 = %b, x2 = %b, z1 = %b", x1, x2, z1);
     //apply input vectors
     initial
10   begin
               #0        x1 = 1'b0;
                         x2 = 1'b0;
               #10       x1 = 1'b0;
                         x2 = 1'b1;
15             #10       x1 = 1'b1;
                         x2 = 1'b0;
               #10       x1 = 1'b1;
                         x2 = 1'b1;
               #10       $stop;
20   end
     //instantiate the module into test bench
     and2_df inst1 (
               .x1(x1),
               .x2(x2),
25             .z1(z1)
               );
     endmodule
```

รูปที่ 1.11 Test bench สำหรับเกท AND ของรูปที่ 1.10

```
x1 = 0, x2 = 0, z1 = 0
x1 = 0, x2 = 1, z1 = 0
x1 = 1, x2 = 0, z1 = 0
x1 = 1, x2 = 1, z1 = 1
```

รูปที่ 1.12 เอาท์พุทไบนารีสำหรับ test bench ของรูปที่ 1.11 สำหรับเกท AND แบบ 2 อินพุท

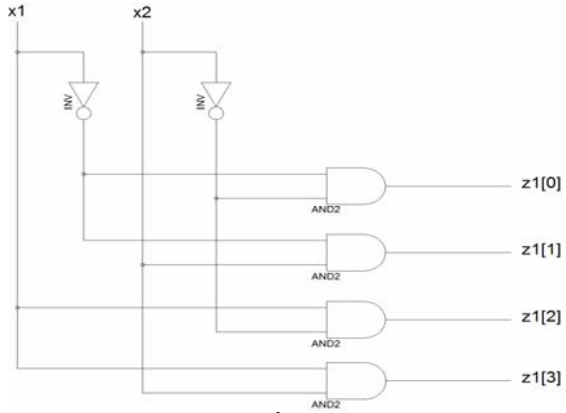


รูปที่ 1.13 เอาท์พุทรูปคลื่นสำหรับ test bench ของรูปที่ 1.11 สำหรับเกท AND แบบ 2 อินพุท

1.6 โมเดลกระแสข้อมูล

การโมเดลกระแสข้อมูล (Dataflow modeling) ถูกใช้เพื่อการออกแบบวงจรเชิงจัดหมู่เท่านั้น นักออกแบบสามารถสร้างฟังก์ชันลอจิกได้ที่ระดับการออกแบบที่สูงกว่าการโมเดลระดับเกท โดยการใช้เซลล์ปฐมฐานของภาษา (built-in primitives) วิธีพื้นฐานของการโมเดลกระแสข้อมูลคือ การใช้คำสั่งกำหนดค่าต่อเนื่อง (continuous assignment statement) โดยใช้คำหลัก **assign** ดังตัวอย่างโมเดลเกท AND แบบ 2 อินพุทของรูปที่ 1.10 โดย **assign** z1 = x1 & x2; หมายถึงกำหนดค่าให้ z1 เท่ากับ x1 แอนด์กับ x2 ในกรณีนี้ไม่มีการกำหนดเวลาถือว่าเป็นศูนย์ แต่ถ้าเป็น **assign** #5 z1 = x1 & x2; จะหมายถึงกำหนดค่าให้ z1 เท่ากับ x1 แอนด์กับ x2 หลังจากเวลาผ่านไป 5 หน่วยเวลา หน่วยเวลาจริงกำหนดด้วยคำหลัก **`timescale** เช่น **`timescale** 10ns/ 100ps หมายถึงหนึ่งหน่วยเวลาเท่ากับ 10 นาโนวินาที และความละเอียดเป็น 100 พิโควินาที

ตัวอย่างในรูปที่ 1.15 แสดงการโมเดลกระแสข้อมูลที่มีการกำหนดเวลาในการกำหนดค่าสำหรับวงจรรูปที่ 1.14 โดย x1 และ x2 เป็นอินพุทแบบสเกลาร์ และ z1=z1[0], z1[1], z1[2], z1[3] เป็นเอาท์พุทแบบเวกเตอร์ ในที่นี้กำหนดหน่วยเวลาเท่ากับ 10 นาโนวินาที และความละเอียดเป็น 1 นาโนวินาที เมื่อค่าอินพุทเปลี่ยนแปลงค่าเอาท์พุทจะเปลี่ยนตามหลังจากเวลาผ่านไป 20 นาโนวินาที ดัง test bench และรูปคลื่นแสดงผลการจำลองในรูปที่ 1.16 และ 1.17 ตามลำดับ



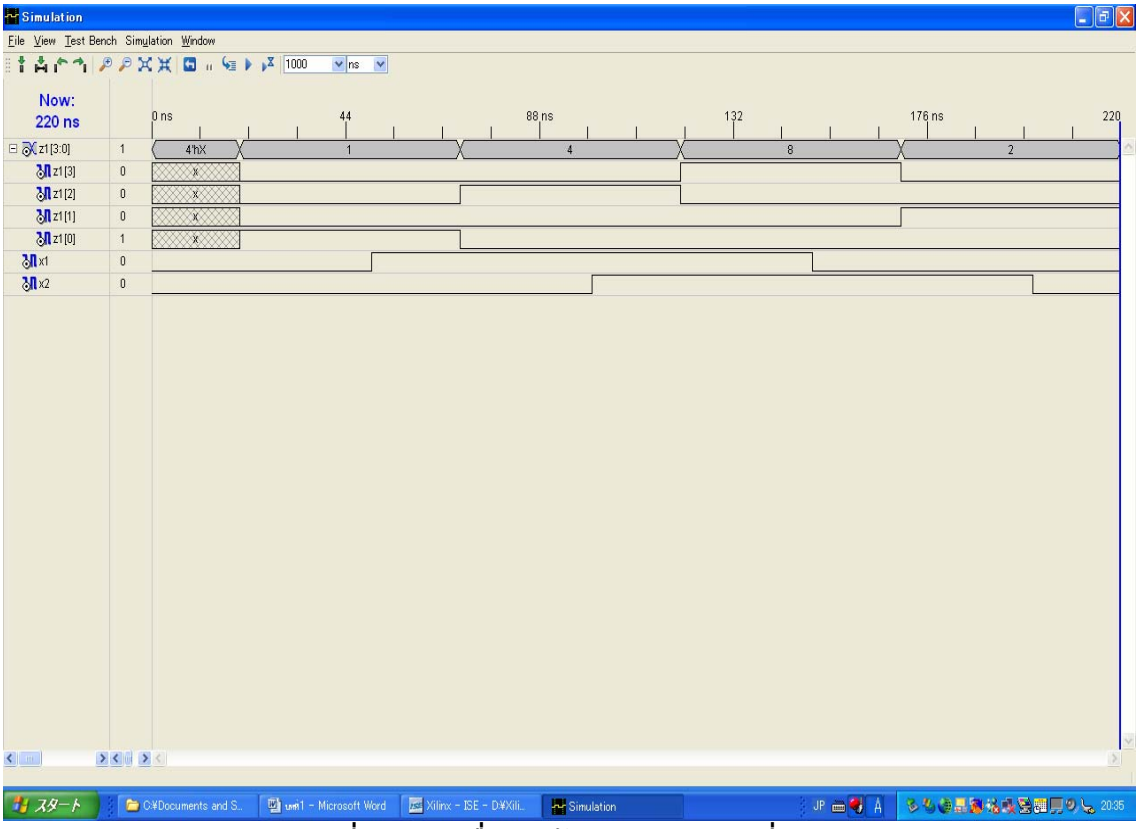
รูปที่ 1.14 วงจรเกท AND แบบ 2 อินพุท 4 ตัว ซึ่งมีอินพุทเป็นสเกลาร์และเอาท์พุทเป็นเวกเตอร์

```
//dataflow with delay
`timescale 10ns / 1ns
module   four_and_delay (x1, x2, z1);
input    x1, x2;
output   [3:0]      z1;
assign   #2 z1[0] = ~x1 & ~x2;
assign   #2 z1[1] = ~x1 & x2;
assign   #2 z1[2] = x1 & ~x2;
assign   #2 z1[3] = x1 & x2;
endmodule
```

รูปที่ 1.15 โมดูล Verilog สำหรับวงจรลอจิกรูปที่ 1.14 แบบมีการกำหนดเวลา

```
//four_and_delay test bench
module four_and_delay_tb;
reg    x1, x2;
wire   [3:0] z1;
initial
$monitor ("x1 x2 =%b, z1 =%b", {x1, x2}, z1);
//apply input vectors
initial
begin
   #0          x1 = 1'b0;
               x2 = 1'b0;
   #5          x1 = 1'b1;
               x2 = 1'b0;
   #5          x1 = 1'b1;
               x2 = 1'b1;
   #5          x1 = 1'b0;
               x2 = 1'b1;
   #5          x1 = 1'b0;
               x2 = 1'b0;
   #5          $stop;
end
four_and_delay inst1(.x1(x1), .x2(x2), .z1(z1));
endmodule
```

รูปที่ 1.16 Test bench สำหรับโมดูล Verilog สำหรับวงจรลอจิกรูปที่ 1.14



รูปที่ 1.17 รูปคลื่นสำหรับ Test bench รูปที่ 1.16

1.7 โมเดลเชิงพฤติกรรม

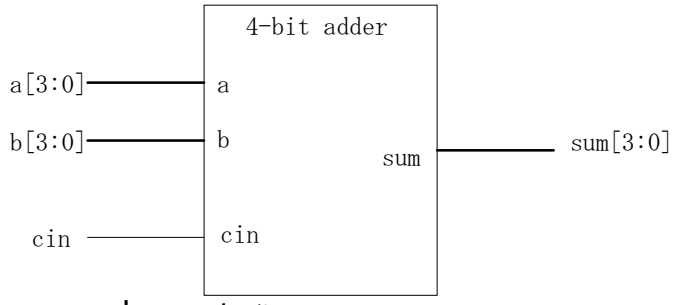
การโมเดลเชิงพฤติกรรม (Behavioral modeling) เป็นวิธีการอธิบายการทำงานของวงจรอีกแบบหนึ่ง โดยไม่ได้ออกแบบระดับเกท พฤติกรรมของวงจรถูกอธิบายด้วยรูปแบบเชิงระเบียบวิธีโดยใช้ประโยค **initial** และ **always**

ประโยค **initial** ถูกกระทำครั้งเดียวระหว่างการจำลองแบบการทำงานโดยเริ่มที่เวลาเท่ากับศูนย์ ประโยค **always** เริ่มที่เวลาเท่ากับศูนย์เช่นเดียวกัน แต่จะถูกทำซ้ำไปตลอด ทั้งสองประโยคใช้ข้อมูลชนิดรีจิสเตอร์เท่านั้น

รูปที่ 1.18 แสดงบล็อกไดอะแกรมของวงจรบวก 4 บิท รายละเอียดลอจิกภายในไม่ถูกแสดง โมดูลเชิงพฤติกรรมของวงจรนี้สามารถถูกอธิบายได้ดังรูปที่ 1.19 พฤติกรรมของวงจรกำหนดด้วย sum= a + b + cin สังเกตเห็นว่ามีเฉพาะพฤติกรรมของวงจรเท่านั้นที่ถูกกำหนด ไม่มีลอจิกเกทภายในเหมือนดังเช่นในโมดูลแบบโครงสร้าง โมดูลเชิงพฤติกรรมไม่ได้กำหนดวิธีการออกแบบวงจรว่าเป็นอย่างไร

ในการโมเดลเชิงพฤติกรรม อินพุทจะถูกประกาศเป็น **wire** และเอาท์พุทถูกประกาศเป็น **reg** คำสั่ง **always** จะทำการตรวจสอบการเปลี่ยนแปลงค่าของตัวแปรที่ระบุในวงเล็บอย่างต่อเนื่อง หากมีการเปลี่ยนแปลงของตัวแปรใดตัวแปรหนึ่ง คำสั่งที่อยู่ภายใน **begin ... in** จะถูกดำเนินการ ข้อความทางขวามือจะถูกคำนวณและจากนั้นก็กำหนดค่าให้กับตัวแปรทางซ้ายมือ

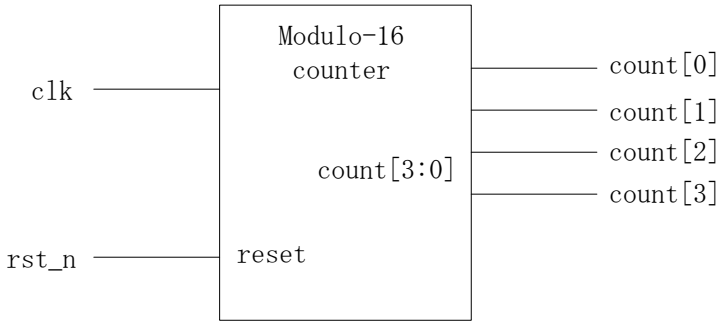
โมดูลเชิงพฤติกรรมของวงจรมอดูโล-16 ของรูปที่ 1.4 ถูกอธิบายได้ดังบล็อกไดอะแกรมรูปที่ 1.20 และโมดูลดังรูปที่ 1.21 คำสั่ง **always** มีสองเหตการณ์คือ ขอบขาขึ้นของ clk และขอบขาลงของ rst_n



รูปที่ 1.18 บล็อกไดอะแกรมของวงจรบวก 4 บิท

```
//behavioral 4-bit adder
module adder_4_behav(a, b, cin, sum);
input [3:0] a, b;
input cin;
output [4:0] sum;

wire [3:0] a, b;
wire cin;
reg [4:0] sum;

always@(a or b or cin)
begin
        sum = a + b + cin;
end
endmodule
```

รูปที่ 1.19 โมดูลเชิงพฤติกรรมของวงจรบวก 4 บิทของรูปที่ 1.18



รูปที่ 1.20 บล็อกไดอะแกรมของวงจรมอดูโม-16 ของรูปที่ 1.4

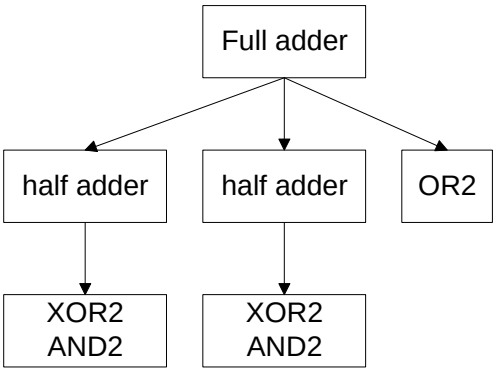
```
module cnt_mod_16(clk, rst_n, count);
input clk, rst_n;
output [3:0] count;

wire clk, rst_n;
reg [3:0] count;

always@(posedge clk or negedge rst_n)
begin
        if(rst_n == 0)
                count <= 4'b0000;
        else
                count <= (count + 1)%16;
end
endmodule
```

รูปที่ 1.21 โมดูลเชิงพฤติกรรมของวงจรมอดูโม-16 ของรูปที่ 1.4

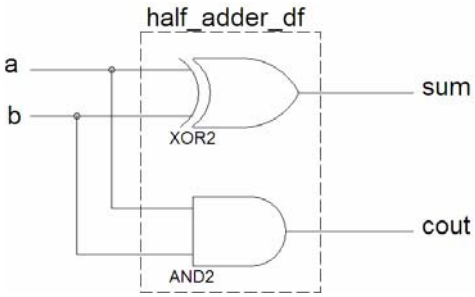
1.8 โมเดลเชิงโครงสร้าง

การโมเดลเชิงโครงสร้าง (Structural modeling) เป็นการออกแบบวงจรระดับบนของโครงสร้างต้นไม้ ซึ่งถูกประกอบขึ้นมาจาก instantiation ของโมดูลระดับต่ำกว่า หรือโมดูลย่อย (submodules) สิ่งจำเป็นก็คือแต่ละโมดูลย่อยจะต้องถูกคอมไพล์และทดสอบความถูกต้องของการทำงานเสียก่อน โมดูลแบบโครงสร้างนี้สามารถอธิบายได้ด้วย เกทปฐมฐานภายในของภาษา วงจรปฐมฐานที่ผู้ใช้สร้างขึ้นมา หรือโมดูลย่อย การเชื่อมต่อระหว่างโมดูลย่อยทำได้โดยการใช้สายสัญญาณ (nets)

ตัวอย่างโมเดลแบบโครงสร้าง เช่น วงจร full adder ที่มีแผนผังการออกแบบบนลงล่างดังรูปที่ 1.22 สมการและรูปการออกแบบระดับเกทสำหรับ half adder และ full adder แสดงดังรูปที่ 1.23 และ 1.24 ตามลำดับ จากนั้นโมดูลของแต่ละวงจรสามารถถูกอธิบายได้ด้วย Verilog ดังรูปที่ 1.25 และ 1.26



รูปที่ 1.22 การออกแบบบนลงล่างของวงจร full adder โดยใช้ half adder 2 ตัว



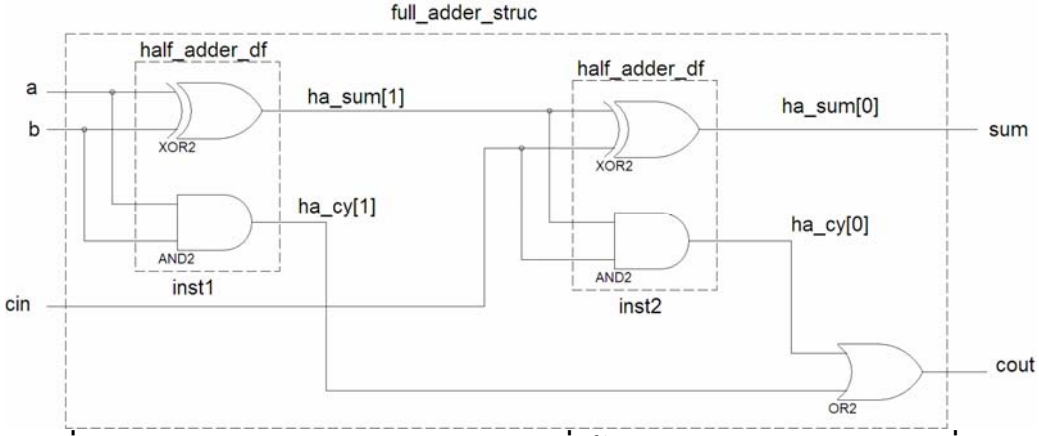
$$sum = a \oplus b$$

$$cout = ab$$

รูปที่ 1.23 สมการและวงจรลอจิกของ half adder ที่จะถูกนำไปใช้ในการออกแบบของ full adder

$$sum = a \oplus b \oplus cin$$

$$cout = cin(a \oplus b) + ab$$



รูปที่ 1.24 สมการและวงจรลอจิกของ full adder ที่สร้างจากวงจร half adder ของรูปที่ 1.23

```
//dataflow half_adder                    assign sum = a ^ b;
module half_adder_df(a, b, sum, cout);    assign cout = a & b;
input a, b;
output sum, cout;                         endmodule


wire a, b, sum, cout;
```

รูปที่ 1.25 โมดูลกระแสข้อมูลสำหรับ half adder ของรูปที่ 1.23

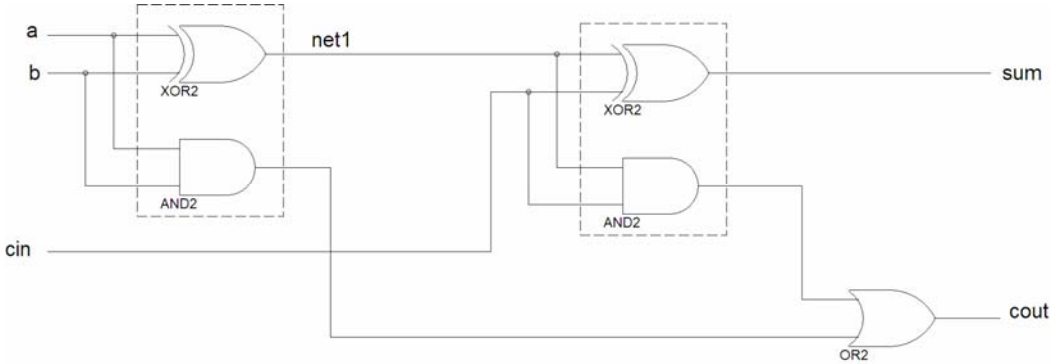
```
//structural full adder
module full_adder_struc(a, b, cin, sum, cout);
input a, b, cin;
output sum, cout;


wire [1:0] ha_sum, ha_cy;


//instantiate the half adder
half_adder_df inst1 (.a(a), .b(b), .sum(ha_sum[1]), .cout(ha_cy[1]));
half_adder_df inst2 (.a(ha_sum[1]), .b(cin), .sum(ha_sum[0]), .cout(ha_cy[0]));


assign sum = ha_sum[0];
assign cout = ha_cy[0] | ha_cy[1];
endmodule
```

รูปที่ 1.26 โมดูลเชิงโครงสร้างสำหรับ full adder ของรูปที่ 1.24

1.9 โมเดลผสม

การโมเดลผสม (Mixed-design modeling) เป็นการรวมเอาการโมเดลหลากหลายแบบมาไว้ในโมดูลเดียวกัน เช่นมีทั้งเกท โมดูลย่อย รวมถึงการกำหนดค่าแบบต่อเนื่อง และรูปแบบเชิงพฤติกรรม ตัวอย่างเช่น วงจร full adder ในรูปที่ 1.27 สามารถถูกออกแบบด้วย วงจรปฐมฐานของภาษา การโมเดลกระแสข้อมูล และการโมเดลเชิงพฤติกรรม ดังโมดูลในรูปที่ 1.28

$$sum = (a \oplus b) \oplus cin$$
$$cout = cin(a \oplus b) + ab$$



รูปที่ 1.27 สมการและวงจรลอจิกของ full adder แบบโมเดลผสม

```
//mixed-design full adder
module full_adder_mixed(a, b, cin, sum, cout);
input     a, b, cin;
output    sum, cout;

reg     cout;
wire    a, b, cin;
wire    sum;
wire    net1;

//built-in primitive
xor (net1, a, b);
```

```
//behavioral
always@(a or b or cin)
begin
        cout = cin & (a ^ b) | (a & b);
end

//dataflow
assign    sum = net1 ^ cin;

endmodule
```

รูปที่ 1.28 โมดูลแบบผสมสำหรับ full adder ของรูปที่ 1.27

เอกสารอ้างอิง

[1] Joseph Cavanagh, Verilog HDL: Digital Design and Modeling, CRC Press, Taylor & Francis Group, 2007

บทที่ 2
ส่วนประกอบของภาษา

เนื้อหาในบทนี้อธิบายรายละเอียดของส่วนประกอบของภาษา Verilog ประกอบด้วย คอมเมนต์ ตัวระบุ คำหลัก ชนิดข้อมูล พารามิเตอร์ และตัวชี้แนะคอมไพเลอร์

1.10 คอมเมนต์

คอมเมนต์ในภาษา Verilog มีสองแบบคือ แบบบรรทัดเดียว และแบบหลายบรรทัด ดังรูปที่ (a) และ (b) ตามลำดับ คอมเมนต์แบบบรรทัดเดียวอธิบายฟังก์ชันโดยอาจจะแยกใช้บรรทัดเดียวหรือเขียนตามหลังโค๊ดในบรรทัดเดียวกัน ข้อความหลังเครื่องหมาย // จะไม่ถูกนำไปคอมไพล์ ส่วนคอมเมนต์แบบหลายบรรทัดเริ่มด้วย /* และจบด้วย */ ข้อความภายในนี้ก็จะไม่ถูกนำไปคอมไพล์เช่นกัน

```
//This is a single-line comment on a dedicated line.
assign   z1 = x1 | x2 ; //This is a comment on a line of code.
```

(a) Single-line comments

```
/*This is a multiple-line comment.
  More comments go here.
  More comments. */
```

(b) A multiple-line comment

รูปที่ 2.1 วิธีการคอมเมนต์ในภาษา Verilog

1.11 ตัวระบุ

ตัวระบุ (Identifier) เป็นชื่อที่ถูกตั้งให้กับอ็อบเจกต์หรือตัวแปรเพื่อให้มันถูกอ้างถึงได้อีกในส่วนอื่นของการออกแบบ ชื่อตัวระบุถูกใช้สำหรับ โมดูล รีจิสเตอร์ พอร์ท สายสัญญาณ หรือโมดูลย่อย ดังตัวอย่างด้านล่าง a, b, cin เป็นตัวระบุของคำหลัก **input** sum, cout เป็นตัวระบุของคำหลัก **output** และ z1 เป็นตัวระบุของคำหลัก **reg** ชื่อตัวระบุสามารถเป็น ตัวอักษร ตัวเลข อักขระ $ หรือ เส้นใต้อักขระ _ แต่อักขระตัวแรกต้องเป็นตัวอักษรหรือเส้นใต้อักขระเท่านั้น อักขระ $ ถูกจองไว้สำหรับตั้งชื่อแทสค์ของระบบ ตัวอักษรพิมพ์เล็กและพิมพ์ใหญ่ถือว่าต่างกัน

```
input    a, b, cin;      //a, b, cin are identifiers.
output   sum, cout       //sum and cout are identifiers.
reg      z1;             // z1 is an identifier.
```

1.12 คำหลัก

ตารางที่ 2.1 แสดงรายการของคำหลักในภาษา Verilog จะเห็นว่ามีเฉพาะตัวอักษรพิมพ์เล็กเท่านั้น

ตารางที่ 2.1 รายการของคำหลักในภาษา Verilog

ประเภท	คำหลัก		
เกทเชิงจัดหมู่	and	buf	nand
	nor	not	or
	xnor	xor	
เกทแบบสองทิศทาง	rtran	rtranif0	rtranif1
	tran	tranif0	tranif1
การกำหนดค่าแบบต่อเนื่อง	assign		
ชนิดข้อมูล	integer	real	realtime
	reg	scalared	time
	tri	tri0	tri1
	triand	trior	trireg
	vectored	wand	wire
	wor		
การประกาศโมดูล	endmodule	module	
การแตกกิ่งหลายทาง	case	casex	casez
	default	endcase	
เหตุการณ์	event		
พารามิเตอร์	defparam	parameter	specparam
การประกาศพอร์ท	inout	input	output
กระบวนคำสั่ง	always	initial	
การกำหนดค่าแบบต่อเนื่องเชิงกระบวนคำสั่ง	assign	deassign	force
	release		
การควบคุมสายงานเชิงกระบวนคำสั่ง	begin	disable	else
	end	for	forever
	fork	if	join
	repeat	wait	while
บล็อกการระบุ	endspecify	specify	
แทสค์และฟังก์ชัน	endfunction	endtask	function
	task		
เกท 3 สถานะ	bufif0	bufif1	notif0
	notif1		
การควบคุมเวลา	edge	negedge	posedge
ปฐมฐานที่กำหนดโดยผู้ใช้	endprimitive	endtable	primitive
	table		

a. เกทเชิงจัดหมู่

ภาษา Verilog มีเกทปฐมฐานภายในจำนวนหนึ่ง มีทั้งแบบอินพุทเดียวและหลายอินพุท แต่มีเพียงเอาท์พุทเดียว การเรียกใช้แสดงดังรูปที่ 2.2(a) **gate_type** คือชนิดของเกทตามคำหลักในตารางที่ 2.1 เอาท์พุทต้องถูกแสดงรายการเป็นลำดับแรก จากนั้นจึงจะตามด้วยอินพุท inst1 คือชื่อโมดูลย่อย อาจจะแสดงหรือไม่ก็ได้ ในกรณีที่ใช้เกทชนิดเดียวกันหลายครั้งก็สามารถเรียกใช้ได้โดยรูปแบบเดียวกันดังรูปที่ 2.2(b)

```
gate_type inst1 (output, input_1, input_2, …, input_n); หรือ
gate_type (output, input_1, input_2, …, input_n);
ตัวอย่างเช่น เกท AND แบบ 2 อินพุท โดยมีอินพุทคือ x1 และ x2 และเอาท์พุทคือ z1
and (z1, x1, x2);
```

(a) เรียกใช้ครั้งเดียว

```
gate_type (output_1, input_11, input_12, …, input_1n),
          (output_2, input_21, input_22, …, input_2n),
                .
                .
                .
          (output_m, input_m1, input_m2, …, input_mn);
```

(b) เรียกใช้หลายครั้ง

รูปที่ 2.2 วิธีการเรียกใช้เกทปฐมฐานในภาษา Verilog

b. เกทแบบสองทิศทาง

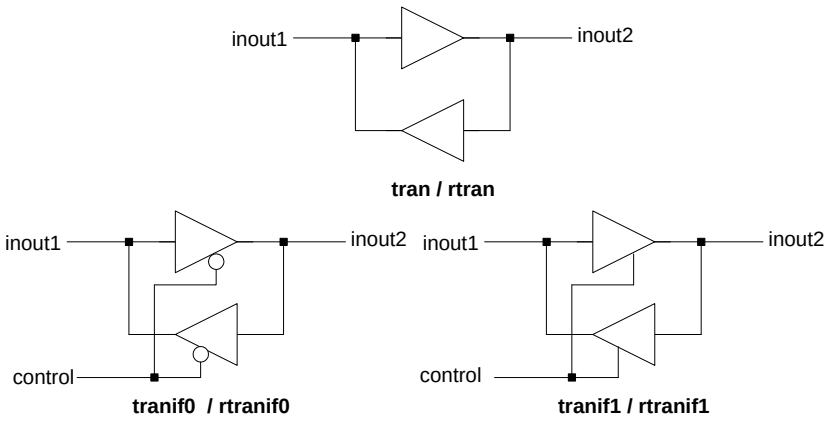
ภาษา Verilog มีเกทปฐมฐานที่มีสองทิศทาง ดังรูปที่ 2.3 สัญญาณสามารถถูกระบุเป็นอินพุทหรือเอาท์พุทอย่างใดอย่างหนึ่ง กล่าวคือสัญญาณใดสัญญาณหนึ่งสามารถเป็นตัวขับ (driver) เกท **tran** ทำหน้าที่เป็นบัฟเฟอร์ ด้านหนึ่งสามารถถูกประกาศเป็น **input** หรือ **inout** และอีกด้านหนึ่งถูกประกาศเป็น **output** หรือ **inout** วิธีการเรียกใช้เป็นดังบรรทัดที่ตามมา โดยชื่อโมดูลย่อยอาจจะระบุหรือไม่ก็ได้

tran inst1 (inout1, inout2);

เกท **tranif0** และ **tranif1** ทำหน้าที่เป็นบัฟเฟอร์สองทิศทางที่มีสัญญาณควบคุม โดยจะทำงานเมื่อสัญญาณควบคุมเป็น **0** และ **1** ตามลำดับ หรือไม่เช่นนั้นเอาท์พุทของเกทจะอยู่ในสภาวะอิมพิแดนซ์สูง วิธีการเรียกใช้เป็นดังบรรทัดที่ตามมา โดยต้องระบุชื่อโมดูลย่อยและสัญญาณควบคุมถูกแสดงรายการไว้หลังสุด

tranif0 inst1 (inout1, inout2, control);
tranif1 inst1 (inout1, inout2, control);

นอกจากนี้ยังมีเกทสองทิศทางที่เป็นเกทต้านทาน (resistive gate) ประกาศโดยเติม **r** ไปข้างหน้าเกททั้งสามข้างต้น การทำงานเป็นเช่นเดียวกัน แต่มีอิมพิแดนซ์จากซอสไปเดรนสูงกว่า จึงมีความแรงของสัญญาณลดลง



รูปที่ 2.3 เกทแบบสองทิศทาง

c. การกำหนดค่าแบบต่อเนื่อง

การกำหนดค่าแบบต่อเนื่องถูกใช้ในการโมเดลแบบข้อมูลกระแสเพื่ออธิบายวงจรเชิงจัดหมู่ และใช้ได้กับสายสัญญาณ (net) เท่านั้น นั่นคือตัวแปรด้านซ้ายของเครื่องหมาย = ต้องถูกประกาศเป็น **wire** ไม่ใช่ **reg** ไวยากรณ์ในการใช้เป็นดังนี้

assign <optional delay> Left-hand side net = Right-hand side expression;

การกำหนดค่าแบบต่อเนื่องถูกใช้เพื่อการกำหนดค่าให้กับสายสัญญาณ โดยเมื่อค่าตัวแปรในนิพจน์ (expression) ทางขวามือเปลี่ยนแปลง นิพจน์จะถูกคำนวณแล้วกำหนดค่าให้กับตัวแปรทางซ้ายมือหลังเวลาผ่านไปเท่ากับที่กำหนด หรือก็คือดีเลย์ในการทำงานของฮาร์ดแวร์จริงนั่นเอง

d. ชนิดข้อมูล

ภาษา Verilog มีชนิดข้อมูล 2 ชนิด คือ สายสัญญาณ (net หรือ wire) สำหรับการเชื่อมต่อกันระหว่างส่วนชิ้นส่วนฮาร์ดแวร์ และรีจิสเตอร์ (register) สำหรับการเก็บข้อมูล ตารางที่ 2.2 แสดงรายละเอียดพอสังเขปของแต่ละชนิดของข้อมูล

ตารางที่ 2.2 รายละเอียดของชนิดของข้อมูล

ชนิดข้อมูล	รายละเอียด
integer	เป็นรีจิสเตอร์อเนกประสงค์เก็บค่าจำนวนเต็ม รองรับการคำนวณเชิงกระบวนคำสั่ง
real	เป็นจำนวนจริงสำหรับระบุค่าข้อมูล มีรูปแบบเป็นเลขทศนิยมลอยตัวแบบ double precision สามารถแสดงด้วยเลขฐานสิบหรือจำนวนทางวิทยาศาสตร์ (เลขชี้กำลัง)
realtime	เหมือนกับ real เพียงแต่ค่าที่เก็บเป็นค่าของเวลาในรูปแบบจำนวนจริง
reg	เป็นข้อมูลชนิดรีจิสเตอร์ ซึ่งเก็บค่าไว้จนกระทั่งมีการกำหนดค่าใหม่ ตัวแปรชนิด **reg** นี้มีความใกล้เคียงกับฮาร์ดแวร์ซึ่งถูกสังเคราะห์เป็น D flip-flops, JK flip-flops หรือ SR latches
scalared	ใช้ในการประกาศ net ซึ่งมีบิทที่สามารถเลือกได้ว่าเป็นแบบปัจเจกหรือเลือกบางส่วน
time	ใช้ในการเก็บเวลาในการจำลองแบบการทำงาน เป็นจำนวนไม่ระบุเครื่องหมาย 64 บิท

ตารางที่ 2.2 รายละเอียดของชนิดของข้อมูล (ต่อ)

tri	ใช้ในการกำหนด net ที่มีตัวขับหลายตัว มีฟังก์ชันเหมือน **wire** แต่ใช้อธิบาย 3-state net
triiand	ใช้กำหนด 3-state net ที่มีตัวขับหลายตัว เป็นโมเดลของ **wand** ในทางฮาร์ดแวร์
trior	ใช้กำหนด 3-state net ที่มีตัวขับหลายตัว เป็นโมเดลของ **wor** ในทางฮาร์ดแวร์
vectored	ใช้ในการประกาศ net ซึ่งมีบิทที่ไม่สามารถเลือกได้ว่าเป็นแบบปัจเจกหรือเลือกบางส่วน
wand	เป็น wired-AND net ซึ่งค่าของ net จะเป็น 0 ถ้าเอาท์พุทใดเอาท์พุทหนึ่งของตัวขับเป็น 0
wire	ใช้แสดงการเชื่อมต่อชิ้นส่วนฮาร์ดแวร์ทางการกายภาพ ซึ่งก็คือ net นั่นเอง
wor	เป็น wired-OR net ซึ่งค่าของ net จะเป็น 1 ถ้าเอาท์พุทใดเอาท์พุทหนึ่งของตัวขับเป็น 1

e. การประกาศโมดูล

ภาษา Verilog อธิบายฟังก์ชันการทำงานของวงจรไว้ในโมดูล โดยใช้ **module** และ **endmodule** ในการเริ่มและจบตามลำดับ

f. การแตกกิ่งหลายทาง

เมื่อมีหลายเส้นทางต้องเลือก การใช้คำสั่ง **if-else if** ซ้อนกันหลายชั้นอาจทำให้เกิดความรุงรัง วิธีที่เหมาะสมกว่าก็คือการใช้คำสั่ง **case** โดยมีวิธีการใช้เป็นดังรูปที่ 2.4 มีการเปรียบเทียบนิพจน์เคส case_expression กับแต่ละทางเลือก (alternative) ในลักษณะบิทต่อบิทตามลำดับของรายการที่แสดง คำสั่งในแต่ละทางเลือกอาจจะมีข้อความเดียว ในกรณีหลายข้อความจะต้องอยู่ภายใต้คำหลัก **begin … end** ถ้าไม่ทางเลือกใหนตรงกับนิพจน์เคสเลยคำสั่งที่อยู่ภายใต้ **default** จะถูกกระทำ คำสั่ง **case** มักถูกใช้สำหรับโมเดลมัลติเพลกเซอร์ ดังโมดูลเชิงพฤติกรรมในรูปที่ 2.5

นอกจากนี้ยังมีคำสั่ง **casex** และ **casez** โดยในคำสั่ง **case** มองค่าสัญญาณ **x** และ **z** ว่าเป็น ค่าที่ไม่รู้ (unknown) และอิมพิแดนซ์สูง (high impedance) ตามลำดับ แต่ในคำสั่ง **casex** และ **casez** มองว่าสัญญาณ **x** และ **z** ดังกล่าวเป็น don't care และใน **casez** ยังมองว่าสัญญลักษณ์ **?** เป็น don't care เช่นเดียวกัน

```
case (case_expression)
        alternative_1: statement_1;
        alternative_2: statement_2;
                .
                .
                .
        alternative_n: statement_n;
        default : default_statement;
endcase
```

รูปที่ 2.4 รูปแบบการใช้คำสั่ง case

```
//4:1 multiplexer using a case statement
module mux4_1_case(sel, data, out);

input   [1:0] sel;
input   [3:0] data;
output  out;

reg     out;

always@ (sel or data)
begin
        case (sel)
                (0) : out = data[0];
                (1) : out = data[1];
                (2) : out = data[2];
                (3) : out = data[3];
        endcase
end
endmodule
```

รูปที่ 2.5 การออกแบบมัลติเพลกเซอร์โดยใช้คำสั่ง case

g. เหตุการณ์

เหตุการณ์คือการเปลี่ยนแปลงค่าของ net หรือ รีจิสเตอร์ การควบคุมเหตุการณ์ปรกติถูกกำหนดโดยใช้สัญญลักณ์ @ และถูกกระทำเมื่อมีการเปลี่ยนค่าจาก 1→0 หรือ 0→1 ของสัญญาณ ซึ่งหมายถึงการควบคุมที่ขอบของสัญญาณ (edge-sensitive control)

การควบคุมเหตุการณ์ OR มีไว้รองรับการเปลี่ยนแปลงของหลายสัญญาณ สัญญาณจะถูก OR ด้วยกัน ถ้าสัญญาณหนึ่งหรือมากกว่าหนึ่งมีการเปลี่ยนแปลง ข้อความคำสั่งที่ตามมาจะถูกกระทำ

คำหลัก **wait** มีไว้รองรับการควบคุมที่ระดับสัญญาณ (level-sensitive control) วิธีนี้มีการรอจนกว่าเงื่อนไขเป็นจริง จึงจะกระทำข้อความคำสั่งที่ตามมา

h. พารามิเตอร์

พารามิเตอร์มักถูกใช้สำหรับระบุค่าค่าคงที่ (constant) เช่น ดีเลย์ และเรนจ์ของตัวแปร พารามิเตอร์ไม่ใช่ตัวแปร สามารถถูกประกาศในโมดูลได้โดยการใช้คำหลัก **parameter** ไวยากรณ์สำหรับการประกาศพารามิเตอร์เป็นดังตัวอย่างในรูปที่ 2.6

นอกจากนี้ยังมี **specparam** ไว้ใช้สำหรับการประกาศพารามิเตอร์ภายในบล็อก **specify … endspecify** พารามิเตอร์นี้สามารถถูกใช้ได้เฉพาะภายในบล็อกนี้เท่านั้น มักถูกใช้ในการกำหนดเวลาระหว่างอินพุทกับเอาท์พุทของโมดูล ค่าคงที่ใดที่ประกาศโดยใช้ **parameter** จะไม่สามารถมองเห็นภายใต้บล็อกนี้

```
parameter   bus_width = 32;                     //integer
parameter   cache_size = 1024;                  //integer
parameter   initialize_counter = 1000_0011;     //register
parameter   real_value = 6.72;                  //real
parameter   width = 8, depth = 32;              //integers
parameter   byte = 8; word = byte * 4           //integers
```

รูปที่ 2.6 ตัวอย่างการประกาศพารามิเตอร์

ค่าคงที่ไม่สามารถถูกเปลี่ยนแปลงระหว่างการจำลองแบบการทำงาน แต่อย่างไรก็ตามค่าของตัวคงที่สามารถถูกเปลี่ยนในระหว่างการคอมไพล์ได้โดยการใช้คำหลัก **defparam** ดังตัวอย่างในรูปที่ 2.7

```
//example of defparam
module def_param1;

parameter x1 = 0;

initial
$display ("value = %d", x1);

endmodule
```

```
//define top level module for defparam1
module top_level;

defparam value1.x1 = 4;
defparam value2.x1 = 8;

def_param1 value1 ( );
def_param1 value2 ( );

endmodule
```

ผลการจำลองแบบเป็นดังนี้

```
value = 4
value = 8
```

รูปที่ 2.7 ตัวอย่างการเปลี่ยนค่าของตัวคงที่โดยการใช้คำหลัก defparam

i. การประกาศพอร์ท

พอร์ทหรือช่องทางในโมดูลมีไว้สำหรับการเชื่อมต่อชิ้นส่วนภายกับสิ่งแวดล้อมภายนอกโมดูล พอร์ทสามารถถูกประกาศเป็น ช่องทางเข้า (**input**) ช่องทางออก (**output**) หรือ ช่องทางเข้า-ออก (**inout**) สามารถถูกประกาศเป็นสเกลาร์หรือเวกเตอร์ก็ได้ ซึ่งสเกลาร์คือค่าเดี่ยว และเวกเตอร์เป็นอาร์เรย์มิติเดียว

j. กระบวนคำสั่ง

กระบวนคำสั่ง (procedure) มีไว้สำหรับอธิบายโมเดลเชิงพฤติกรรม สามารถถูกสร้างขึ้นมาได้ภายใต้บล็อก **always** และ **initial** โดยทั้งสองเริ่มที่เวลา 0 แต่ข้อความคำสั่งภายใต้ **initial** ถูกกระทำครั้งเดียว ในขณะที่ข้อความคำสั่งภายใต้ **always** ถูกกระทำซ้ำอย่างต่อเนื่องในลักษณะเดียวกับการวนลูป

บล็อก **always** และ **initial** นี้มีไว้รองรับการทำงานแบบแข่งขนาน (concurrency) ในโมดูลหนึ่งสามารถมีบล็อกดังกล่าวหลายบล็อก ข้อความคำสั่งภายใต้บล็อกต่างกันจะทำงานพร้อมๆกัน

k. การกำหนดค่าแบบต่อเนื่องเชิงกระบวนคำสั่ง

การกำหนดค่าแบบต่อเนื่องเชิงกระบวนคำสั่ง เป็นการกำหนดค่าแบบต่อเนื่องภายใต้บล็อก **always** และ **initial** ทำให้เน็ทหรือรีจิสเตอร์ถูกกำหนดค่าอย่างต่อเนื่อง มีความแตกต่างจากการกำหนดค่าแบบต่อเนื่องในหัวข้อ 2.3.3 คือการกำหนดค่าแบบต่อเนื่องเป็นการกำหนดค่าภายนอกบล็อก **always** และ **initial**

ในที่นี้คำหลัก **assign** ภายใต้บล็อก **always** และ **initial** นี้ถูกใช้เพื่อกำหนดค่าเชิงกระบวนคำสั่งทับบนรีจิสเตอร์ (ห้ามใช้กับเน็ท) ผลของการกำหนดค่าแบบต่อเนื่องเชิงกระบวนคำสั่งจะยังคงอยู่ต่อไปจนกระทั่งถูกการกำหนดค่าเชิงกระบวนคำสั่งอื่นทับ หรือจนกระทั่งพบคำสั่ง **deassign**

ถ้าต้องการให้สามารถกำหนดค่าเชิงกระบวนคำสั่งกับทั้งรีจิสเตอร์และเน็ทควรใช้ **force ... release** แทน ซึ่งมักถูกใช้ใน test bench เมื่อ **force** ถูกใช้กับเน็ท สถานะของเน็ทจะถูกเปลี่ยนตามค่าที่บังคับ และเน็ทจะมีค่าดังสถานะเดิมเมื่อพบคำสั่ง **release** เมื่อ **force** ถูกใช้กับรีจิสเตอร์ สถานะของริจิสเตอร์จะถูกเปลี่ยนตามค่าที่บังคับ และยังคงค่าเดิมหลังจากคำสั่ง แต่ก็สามารถเปลี่ยนค่าได้โดยการกำหนดค่าเชิงกระบวนคำสั่งที่ตามมา

I. การควบคุมสายงานเชิงกระบวนคำสั่ง

ประโยคการควบคุมสายงานเชิงกระบวนคำสั่งดัดแปลงสายงานโดย การเลือกกิ่ง การกระทำกิจกรรมซ้ำ การเลือกกิจกรรมขนาน หรือการหยุดกระทำกิจกรรมใดกิจกรรมหนึ่ง กิจกรรมสามารถเกิดในบล็อกแบบลำดับ (sequential block) หรือบล็อกแบบขนาน (parallel block)

begin ... end ถูกใช้สำหรับการรวมกลุ่มของประโยคคำสั่งเข้ามาเป็นบล็อกแบบลำดับ คำสั่งจะถูกทำงานตามลำดับรายการ ยกเว้นประโยคแบบ nonblocking ซึ่งจะถูกกระทำไปพร้อมกัน การใช้งานแสดงดังข้างล่าง โดยชื่อบล็อกอาจจะมีหรือไม่มีก็ได้ ถ้าในบล็อกมีประโยคเดียว **begin ... end** ไม่ต้องแสดงก็ได้

```
begin :block_name
        block of sequential procedural statements
end
```

disable ถูกใช้สำหรับหยุดการทำงานของบล็อกประโยคเชิงกระบวนการ หรือหยุดแทสค์และโอนย้ายการควบคุมไปยังประโยคที่ตามมา การหยุดนี้สามารถถูกใช้สำหรับการออกจากลูปได้อีกด้วย

for ถูกใช้สำหรับกำหนดลูป ให้กระทำซ้ำตามจำนวนครั้งที่ถูกระบุ รูปแบบการใช้งานคล้ายกับลูปในภาษา C ดังนี้

```
for (i=0; i<10; i++)            //หมายถึงการวนลูป 10 ครั้ง
        block of sequential procedural statements
```

forever ถูกใช้สำหรับการกำหนดลูปที่ทำซ้ำอย่างต่อเนื่อง มักถูกใช้สำหรับการควบคุมเวลา เช่น การสร้างสัญญาณนาฬิกา ต้องบรรจุอยู่ในบล็อก **initial** หรือ **always** เสมอ การออกจากลูปทำได้โดยคำสั่ง **disable**

fork ... join ถูกใช้สำหรับการกำหนดบล็อกขนานซึ่งกระทำคำสั่งภายในบล็อกพร้อมๆกัน ซึ่งตรงข้ามกับ **begin ... end**

if ... else ถูกใช้เป็นประโยคเงื่อนไข ให้เลือกสายงานตามเงื่อนไข (condition) ซึ่งเป็นค่าบูลีน มีไวยากรณ์รูปที่ 2.8(a) ถ้าเงื่อนไขเป็นจริง procedural statement 1 ก็จะถูกกระทำ นอกนั้น procedural statement 2 ก็จะถูกกระทำ กรณีมีหลายทางเลือกก็สามารถใช้ **if ... else** ซ้อนกันดังรูปที่ 2.8(b)

```
if (condition)
    {procedural statement 1}
else
    {procedural statement 2}
```

(a) แบบทางเลือกเดียว

```
if (condition_1)
    {procedural statement 1}
else if (condition_2)
    {procedural statement 2}
else if (condition_3)
    {procedural statement 3}
else
    {procedural statement 4}
```

(b) แบบหลายทางเลือก

รูปที่ 2.8 ไวยากรณ์การใช้งาน if … else

repeat ถูกใช้สำหรับกำหนดลูป ให้กระทำซ้ำตามจำนวนครั้งที่ถูกระบุภายในวงเล็บ ไวยากรณ์การใช้งานเป็นดังนี้

repeat (expression)

statement of block of statements

โดย expression อาจจะเป็นค่าคงที่ ตัวแปร หรือค่าสัญญาณ ถ้าผลลัพธ์ของการคำนวณ expression เป็น **x** หรือ **z** ค่านี้จะถูกมองเป็น 0 ลูปก็จะไม่ถูกกระทำ ตัวอย่างการใช้งานแแสดงดังรูปที่ 2.9

repeat สามารถถูกใช้ในการควบคุมเหตุการณ์ซ้ำ เพื่อหน่วงเวลาการกำหนดค่าทางขวามือไปยังค่าทางซ้ายมือของนิพจน์ ตัวอย่างเช่น

reg3 = **repeat** (3) @ (**posedge** clk) reg1 + reg2;

หมายความว่าผลลัพธ์จากการบวก reg1 กับ reg2 จะถูกกำหนดค่าให้กับ reg3 หลังจากขอบขาขึ้นของสัญญาณ clk ผ่านไปสามครั้งติดต่อกัน

wait เป็นการควบคุมที่ระดับสัญญาณ ที่รอให้นิพจน์ในวงเล็บเป็นจริงก่อนการกระทำคำสั่งหรือบล็อกของคำสั่ง ตัวอย่างเช่น

wait (data_ready) data_reg = data_in;

หมายถึงการกำหนดค่า data_in ให้กับ data_reg จะรอจนกว่า data_ready เป็นจริง (ลอจิก 1)

while ถูกใช้ในการกำหนดลูป โดยคำสั่งหรือบล็อกคำสั่งที่อยู่ภายใต้ **while** จะถูกกระทำซ้ำไปเรื่อยๆ ถ้านิพจน์ในวงเล็บเป็นจริง (ลอจิก 1) ไม่เช่นนั้นก็ออกจากลูปถ้านิพจน์ในวงเล็บเป็นเท็จ (ลอจิก 0) ไวยากรณ์การใช้งานเป็นดังนี้

while (expression) statement or block of statements

```
//Example of the repeat keyword
module repeat_example;
integer count;
initial
begin
        count = 0;
        repeat (8)
        begin
            $display ("count = %d", count);
                count = count + 1;
        end
end
endmodule
```

ผลลัพธ์ที่ได้คือ

```
count = 0
count = 1
count = 2
count = 3
count = 4
count = 5
count = 6
count = 7
```

รูปที่ 2.9 ตัวอย่างการใช้งาน repeat

m. บล็อกการระบุ

บล็อกการระบุ (specify block) ถูกใช้สำหรับการกำหนดค่าดีเลย์ของเส้นทางในโมดูล (module path delay) สามารถกำหนดค่าดีเลย์จากอินพุทไปยังเอาท์พุท ไวยากรณ์การใช้งานเป็นดังนี้

```
specify
        Timing specification and timing checks
        Define specparam constants
endspecify
```

คำหลัก **specparam** ถูกใช้สำหรับการประกาศพารามิเตอร์ภายในบล็อก **specify…endspecify** ซึ่งต้องเป็นบล็อกที่แยกออกจากบล็อกอื่น (เช่น **initial** หรือ **always**) ในโมดูลเดียวกัน ตัวอย่างการใช้งานบล็อกการระบุแสดงดังรูปที่ 2.10 ดีเลย์ tplh และ tphl เป็นดีย์ของการเปลี่ยนแรงดันจากต่ำไปสูง และจากสูงไปต่ำตามลำดับ โดยถูกระบุเป็น 3 ค่าคือค่าต่ำสุด ค่าปรกติ และค่าสูงสุด ข้อความ (x1 => z1) หมายถึงดีเลย์ของการแพร่ผ่านจากอินพุทไปยังเอาท์พุท ซึ่งถูกกำหนดให้เท่ากับค่าดีเลย์ tplh และ tphl ที่ระบุไว้ข้างต้น

```
//Example of specify block with delays
module specify_block(x1, x2, z1);
input x1, x2;
output z1;

nor (z1, x1, x2);
specify
  specparam
    tplh = 0.55 : 0.90 : 0.12,    //min : typ : max
    tphl = 0.50 : 0.70 : 1.55;
    (x1 => z1) = tplh, tphl;
    (x2 => z1) = tplh, tphl;
endspecify
endmodule
```

รูปที่ 2.10 ตัวอย่างการใช้งาน specify block

n. แทสค์และฟังก์ชัน

แทสค์และฟังก์ชันมีลักษณะคล้ายกระบวนคำสั่ง (หรือรูทีนย่อย) ที่สามารถถูกเรียกใช้ได้มากกว่าหนึ่งครั้ง แทสค์สามารถคืนค่าตัวแปรได้มากกว่าหนึ่งค่า ในขณะที่ฟังก์ชันคืนค่าตัวแปรได้เพียงค่าเดียวและจำเป็นต้องมีอย่างน้อยหนึ่งอินพุท ทั้งแทสค์และฟังก์ชันถูกจำกัดอยู่ภายใต้โมดูลที่เรียกใช้ แทสค์สามารถเรียกฟังก์ชันและแทสค์อื่นได้ ในขณะที่ฟังก์ชันสามารถเรียกฟังก์ชันอื่นได้แต่ไม่สามารถเรียกแทสค์ได้ ไวยากรณ์การใช้งานเป็นดังนี้

task task_name;
Declarations
Procedural statements
endtask

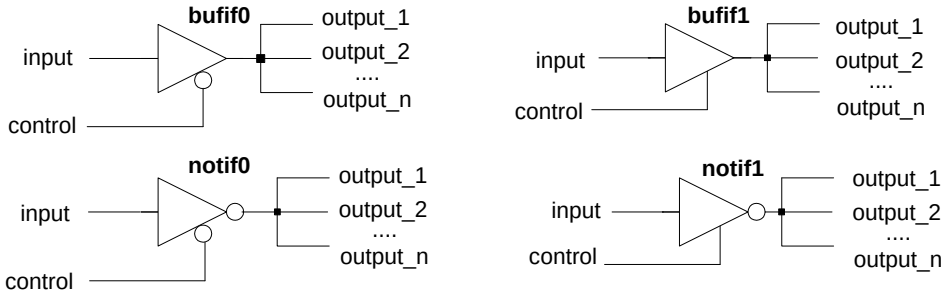
function [range] function_name;
input declararion(s)
other declarations
procedural statements
endfunction

o. เกท 3 สถานะ

เกท 3 สถานะ ถูกใช้สำหรับการโมเดลตัวขับ 3 สถานะ โดยมีอินพุทแบบสเกลาร์ 1 อินพุท มีเอาท์พุทแบบสเกลาร์ 1 เอาท์พุทหรือมากกว่า และอินพุทควบคุม 1 อินพุท เอาท์พุทจะมีสถานะอิมพิแดนซ์สูงถ้าอินพุทควบคุมไม่แอ็กทิฟ ไวยากรณ์การใช้งานเป็นดังนี้

gate_type inst1 (output_1, output_2, …, output_n, input, control);

โดยเอาท์พุทจะถูกแสดงรายการก่อน ตามด้วยอินพุท และอินพุทควบคุมถูกแสดงไว้หลังสุด



รูปที่ 2.11 เกท 3 สถานะ

p. การควบคุมเวลา

กิจกรรมในโมดูลสามารถถูกควบคุมด้วยขอบของสัญญาณ ตัวอย่างเช่น ขอบขาขึ้น (positive edge) หรือขอบขาลง (negative edge) ตารางที่ 2.3 แสดงตัวอย่างของขอบสัญญาณ ขอบสัญญาณเหล่านี้ถูกใช้ควบคุมการทำงานงานของฟลิบฟลอบและอุปกรณ์อื่นที่ทำงานที่ขอบ ถ้าสัญญาณนาฬิกาถูกใช้ควบคุมเหตุการณ์ในระบบ คำหลัก **posedge** ถูกใช้เพื่อควบคุมการทำงานที่ขอบขาขึ้น คำหลัก **negedge** ถูกใช้เพื่อควบคุมการทำงานที่ขอบขาลง ดังตัวอย่างดังนี้

การทำงานที่ขอบขาขึ้นของสัญญาณ clk

always @ (**posedge** clk)
z1 = x1 & x2;

การทำงานที่ขอบขาลงของสัญญาณ clk

always @ (**negedge** clk)
z1 = x1 & x2;

ตารางที่ 2.3 ตัวอย่างขอบสัญญาณ

ขอบขาขึ้น		ขอบขาลง	
0→x	เปลี่ยนจาก 0 เป็น unknown value	1→x	เปลี่ยนจาก 1 เป็น unknown value
0→z	เปลี่ยนจาก 0 เป็น high impedance	1→z	เปลี่ยนจาก 1 เป็น high impedance
0→1	เปลี่ยนจาก 0 เป็น 1	1→0	เปลี่ยนจาก 1 เป็น 0
x→1	เปลี่ยนจาก unknown value เป็น 1	x→0	เปลี่ยนจาก unknown value เป็น 0
z→1	เปลี่ยนจาก high impedance เป็น 1	z→0	เปลี่ยนจาก high impedance เป็น 0

นอกจากนี้ยังมี **edge** ซึ่งถูกใช้สำหรับการกำหนดเวลาของเหตุการณ์ให้ละเอียดยิ่งขึ้น ดังตัวอย่างด้านล่างเป็นการระบุเวลา setup = 6 และเวลา hold = 3 ให้กับฟลิบฟลอบโดยใช้ซิสเต็มแทสค์ **$setuphold** กล่าวคือ data_1 ต้องเสถียรที่อินพุทของฟลิบฟลอบก่อนขอบขาขึ้นของ clk เป็นเวลา 6 หน่วยเวลา และต้องคงค่าต่อไปนานเท่ากับ 3 หน่วยเวลาหลังขอบขาขึ้นของ clk

$setuphold (data_1, **edge** 01 clk, 6, 3);

q. ปฐมฐานที่ผู้ใช้กำหนด

โมดูลปฐมฐานสามารถถูกสร้างขึ้นมาโดยผู้ใช้ โมดูลปฐมฐานนี้ถูกเรียกว่า UDP (user-defined primitive) ซึ่งถูกสร้างโดยการใช้ตารางซึ่งนิยามฟังก์ชันของโมดูลปฐมฐาน โมดูลอาจจะเป็นเชิงจัดหมู่หรือเชิงลำดับก็ได้ สามารถมีอินพุทสเกลาร์ 1 อินพุทหรือมากกว่า แต่มีเอาท์พุทสเกลาร์ได้เพียง 1 เอาท์พุท ไวยากรณ์การใช้งานเป็นดังนี้

primitive udp_name (output, input_1, input_2, …, input_n);
  Output declaration
  Input declarations
  table
    Define the functionality of the primitive
  endtable
endprimitive

```
//User-defined primoitive for a 2-input OR gate
primitive udp_or2(out, a, b); //list output first

output out;

input a, b;
```

```
//state table definition
  table
    // a   b  :  out; comment is for readability
      0   0  :   0;
      0   1  :   1;
      1   0  :   1;
      1   1  :   1;
  endtable
endprimitive
```

รูปที่ 2.12 โมดูล UDP สำหรับเกท OR แบบ 2 อินพุท

1.13 เซทของค่า

เอาท์พุทของลอจิกเกททำหน้าที่ขับอินพุทของลอจิกเกทอื่น ภาษา Verilog แทนสถานะของเอาท์พุทด้วยค่า 4 ค่าดังตารางที่ 2.4

ตารางที่ 2.4 เซทของค่าสถานะของเอาท์พุท

ระดับ	รายละเอียด
0	ลอจิก 0 หรือภาวะเท็จ
1	ลอจิก 1 หรือภาวะะจริง
x	ลอจิกไม่รู้ค่า
z	ภาวะอิมพิแดนซ์สูง หรือสถานะลอย

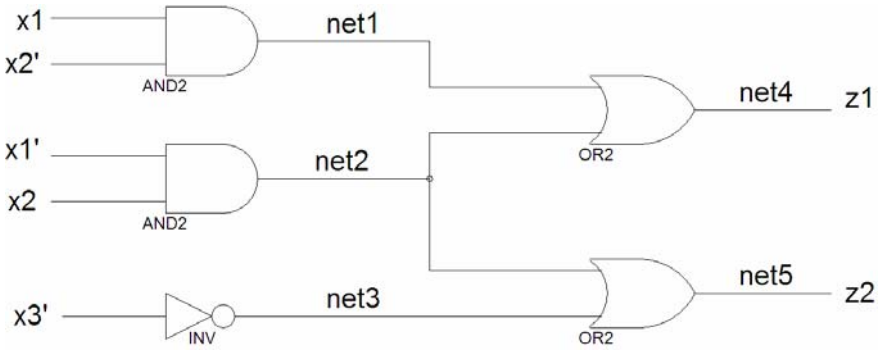
ค่า 0 และ 1 แทนลอจิกต่ำและสูงของลอจิกเกททั้งหลาย ค่า x แสดงถึงความไม่ชัดเจนว่าเป็นลอจิก 0 หรือ 1 ค่า z แสดงภาวะอิมพิแดนซ์สูง ในทางกายภาพก็คือตัวขับไม่ทำงาน หรือไม่ต่อ ตัวขับ 3 สถานะนี้มักถูกใช้เพื่อต่อวงจรเข้ากับบัส เมื่อค่า z ถูกป้อนให้กับอินพุทของลอจิกเกท มันจะถูกมองว่าเป็นค่า x

1.14 ชนิดข้อมูล

ภาษา Verilog มีข้อมูล 2 ชนิด คือ เน็ทและรีจิสเตอร์ ซึ่งถูกใช้เพื่อเชื่อมต่อวงจรลอจิกและเพื่อเป็นหน่วยความจำ เน็ทก็คือสายสัญญาณ (wire) หรือกลุ่มของสายสัญญาณที่เชื่อมต่อชิ้นส่วนฮาร์ดแวร์ภายในหรือภายนอกโมดูล ค่าของเน็ทหนึ่งๆถูกคำนวณจากลอจิกที่ขับเน็ทนั้นๆ รีจิสเตอร์ก็คือหน่วยความจำซึ่งคงค่าไว้ตลอดจนกว่าจะมีค่าใหม่ถูกป้อนเข้ามา

2.5.1. เน็ท

ตัวอย่างของข้อมูลชนิดเน็ทแสดงดังรูปที่ 2.13 มีเน็ทภายในคือ net1…net5 ทำหน้าที่เชื่อมต่อลอจิกเกทต่างๆเข้าด้วยกัน โมดูล Verilog สำหรับวงจรลอจิกดังกล่าวแสดงดังรูปที่ 2.14 โมดูลถูกออกแบบด้วยโมเดลเชิงกระแสข้อมูล ในที่นี้เน็ทถูกประกาศเป็น **wire** และเกทปฐมฐานภายในถูกเรียกใช้เพื่อสร้างลอจิกเกท



รูปที่ 2.13 วงจรลอจิกแสดงการใช้เน็ท

```
//module showing use of wire
//connecting logic primitives
module log_diag_eqn4(x1, x2, x3, z1, z2);

input x1, x2, x3;
output z1, z2;

wire x1, x2, x3;
wire z1, z2;

//define internal nets as wire
wire net1, net2, net3, net4, net5;
```

```
//instantiate the built-in primitives
and (net1, x1, ~x2);
and (net2, ~x1, x2);
not (net3, ~x3);
or (net4, net1, net2);
or (net5, net2, net3);

assign z1 = net4;
assign z2 = net5;

endmodule
```

รูปที่ 2.14 โมดูล Verilog สำหรับวงจรลอจิกรูปที่ 2.13

2.5.2. รีจิสเตอร์

ข้อมูลชนิดรีจิสเตอร์แทนตัวแปรที่คงค่าได้ รีจิสเตอร์ในภาษา Verilog เปรียบได้กับรีจิสเตอร์ทางฮาร์ดแวร์ แต่ต่างกันเชิงแนวคิด รีจิสเตอร์ทางฮาร์ดแวร์ถูกสร้างด้วยวงจรหน่วยความจำเช่น D flip-flops JK flip-flops และ SR latches ในขณะที่รีจิสเตอร์ในภาษา Verilog เป็นตัวแทนนามธรรมของรีจิสเตอร์ทางฮาร์ดแวร์

ขนาดโดยปริยายเมื่อไม่ได้ถูกกำหนดของรีจิสเตอร์ก็คือ 1 บิท ถ้าต้องการขนาดที่กว้างขึ้น ต้องการให้รีจิสเตอร์ A มีขนาด 16 บิท สามารถประกาศได้ดังนี้

reg [15:0] A;

รีจิสเตอร์สามารถถูกกำหนดค่าได้โดยการใช้คำสั่งประโยคเดียวดังนี้

A = 16'h7ab5; สำหรับการกำหนดค่าด้วยเลขฐานสิบหก

หรือ A = 16'b0111101010110101; สำหรับการกำหนดค่าด้วยเลขฐานสอง

หน่วยความจำสามารถถูกสร้างได้โดยการใช้อาร์เรย์ของรีจิสเตอร์ได้ดังนี้

Number of bits per register number of registers

reg [msb:lsb] memory_name [first address:last address];

ตัวอย่างเช่นหน่วยความจำขนาด 32 word x 1 byte สามารถถูกประกาศได้ดังนี้

reg [7:0] memory_name [0:31];

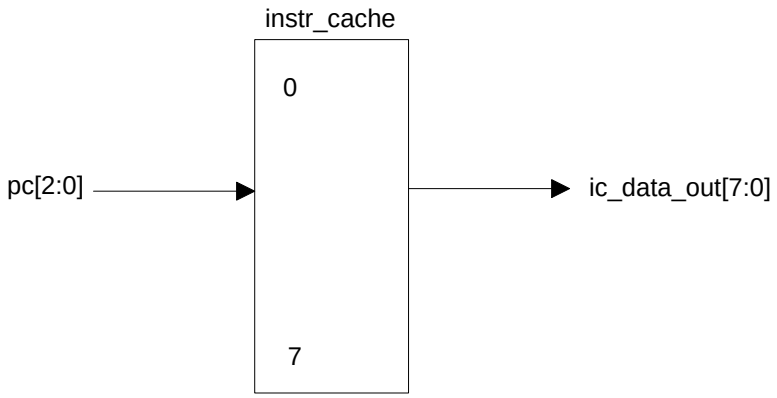
ข้อมูลสามารถถูกจัดเก็บไว้ในหน่วยความจำได้โดยการกำหนดค่าให้กับแต่ละรีจิสเตอร์แบบตัวต่อตัว ดังแสดงในรูปที่ 2.15 ซึ่งแสดงหน่วยความจำขนาด 8 word x 2 byte

```
reg [15:0] instr_cache [0:7];
-------------------------------------------------
instr_cache[0] = 16'h0808;
instr_cache[1] = 16'h0909;
instr_cache[2] = 16'h0a0a;
instr_cache[3] = 16'h0b0b;
instr_cache[4] = 16'h0c0c;
instr_cache[5] = 16'h0d0d;
instr_cache[6] = 16'h0e0e;
instr_cache[7] = 16'h0f0f;
-------------------------------------------------
```

รูปที่ 2.15 แคชคำสั่ง (Instruction Cache) ของ 8 รีจิสเตอร์ซึ่งมีขนาด 16 บิทต่อรีจิสเตอร์

การจัดเก็บข้อมูลไว้ในหน่วยความจำอีกวิธีหนึ่งคือการใช้ซิสเต็มแทสค์ **$readmemb** สำหรับข้อมูลที่เป็นเลขฐานสอง หรือข้อมูลไบนารี (binary data) หรือ **$readmemh** สำหรับข้อมูลที่เป็นเลขฐานสิบหก (hexadecimal data) ข้อมูลถูกเตรียมไว้เป็นไฟล์ข้อความ (text file) จากนั้นซิสเต็มแทสค์จะทำการอ่านไฟล์และบันทึกเนื้อหาลงไปยังหน่วยความจำ ตัวอย่างการบรรจุแคชคำสั่งได้วยข้อมูลไบนารี แสดงดังตัวอย่างที่ 3.1

ตัวอย่างที่ 3.1 บล็อกไดอะแกรมของแคชคำสั่งในรูปที่ 2.16 มีโปรแกรมเคาเตอร์ขนาด 3 บิทคือ pc [2:0] เป็นอินพุทเวกเตอร์เพื่อเป็นแอสเดรสของแคชคำสั่ง และมีบัสขนาด 8 บิทเป็นเอาท์พุทเวกเตอร์คือ ic_data_out[7:0] แต่ละแอดเดรสถือเป็นแคชไลน์ที่เก็บค่าหนึ่งบล็อกข้อมูล ซึ่งในที่นี้ก็คือหนึ่งไบท์ เนื้อหาของแคชเป็นคำสั่งเพื่อกระทำงานต่างๆ



รูปที่ 2.16 บล็อกไดอะแกรมของแคชคำสั่งสำหรับตัวอย่างที่ 3.1

ไฟล์ข้อความในตารางที่ 2.5 ถูกสร้างขึ้นมาแล้วบันทึกเป็นชื่อ icache.instr หมายเหตุแอดเดรสที่แสดงมีไว้เพื่ออ้างอิงเท่านั้น โดยไฟล์ดังกล่าวถูกบันทึกไว้ในโฟลเดอร์เดียวกับโปรเจคที่กำลังทำอยู่

ตารางที่ 2.5 ไฟล์ icache.instr

Address	Data	Address	Data
Word 0	00001000	Word 4	00001100
Word 1	00001001	Word 5	00001101
Word 2	00001010	Word 6	00001110
Word 3	00001011	Word 7	00001111

เนื้อหาของ icache.instr ถูกบรรจุเข้าไปในหน่วยความจำ instr_cache โดยเริ่มที่ตำแหน่ง 0 ได้ดังสองคำสั่งนี้

reg [7:0] instr_cache [0:7];

$readmemb (“icache.instr”, instr_cache);

โมดูล Verilog สำหรับกระบวนวิธีข้างต้นแสดงดังรูปที่ 2.17 คำสั่ง **initial** ถูกใช้เพื่อดึงค่ามาจากไฟล์ icache.instr ซึ่งถูกกระทำเพียงครั้งเดียว จากนั้นคำสั่ง **always** ถูกใช้เพื่ออ่านเนื้อหาของแคชคำสั่งตามตำแหน่งที่ถูกระบุโดย program counter โดยบล็อก **begin…end** ถูกกระทำทุกครั้งที่ค่า program counter เปลี่ยน

Test bench สำหรับจำลองแบบการทำงาน ไฟล์ icache.instr เอาท์พุทไบนารี และผลรูปคลื่น แสดงดังรูปที่ 2.18 2.19 2.20 และ 2.21 ตามลำดับ

```
//procedure for loading memory with
//binary data from file icache.instr
module mem_load(pc, ic_data_out);
input     pc;
output    ic_data_out;

wire [2:0] pc;          // a program counter
reg [7:0] ic_data_out;

//define memory size
//instr_cache is an array of eight 8-bit regs
reg [7:0] instr_cache [0:7];
```

```
//define memory contents
//load instr_cache from file icache.instr
initial
begin
  $readmemb ("icache.instr", instr_cache);
end

//use a program counter to access the instr_cache
always @(pc)
begin
        ic_data_out = instr_cache [pc];
end
endmodule
```

รูปที่ 2.17 โมดูล Verilog สำหรับการใช้ $readmemb เพื่อบรรจุแคชคำสั่ง

```
//mem_load test bench
module mem_load_tb_v;
reg [2:0] pc;                   // Inputs
wire [7:0] ic_data_out;     // Outputs
integer     i;     //used for display contents

//assign values to the program counter
initial
begin
            #0          pc = 3'b000;
            #10         pc = 3'b001;
            #10         pc = 3'b010;
            #10         pc = 3'b011;
            #10         pc = 3'b100;
            #10         pc = 3'b101;
            #10         pc = 3'b110;
            #10         pc = 3'b111;
            #10         $stop;
end
```

```
//display the contents of the instruction cache
initial
begin
            for (i=0; i<8; i=i+1)
            begin
            #10 $display ("address %h = %b", i,
ic_data_out);
            end
            #150 $stop;
end
// Instantiate the Unit Under Test (UUT)
  mem_load uut (
                        .pc(pc),
                        .ic_data_out(ic_data_out)
                      );
endmodule
```

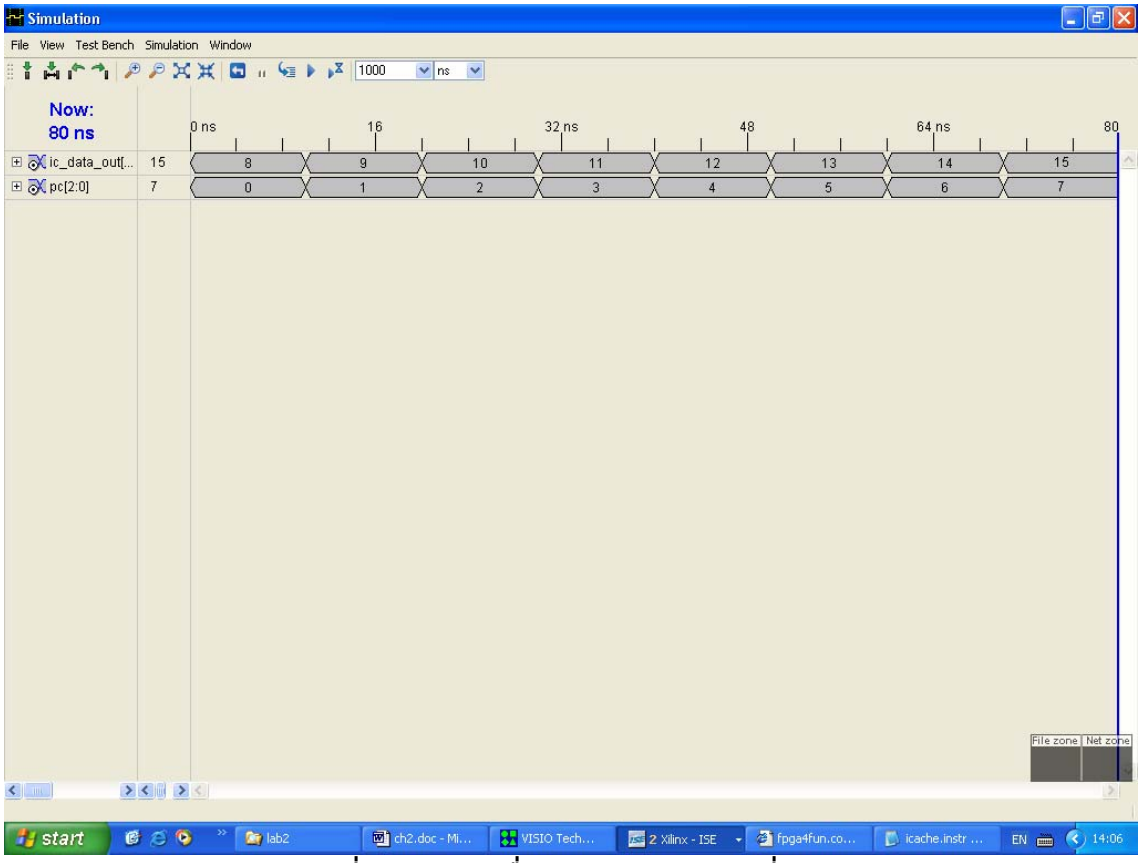
รูปที่ 2.18 Test bench สำหรับจำลองแบบการทำงานของรูปที่ 2.17

```
00001000
00001001
00001010
00001011
00001100
00001101
00001110
00001111
```

รูปที่ 2.19 ไฟล์ icache.instr ที่ถูกบันทึกในโฟลเดอร์เดียวกับโปรเจคที่กำลังทำอยู่

```
address 00000000 = 00001000
address 00000001 = 00001001
address 00000002 = 00001010
address 00000003 = 00001011
address 00000004 = 00001100
address 00000005 = 00001101
address 00000006 = 00001110
address 00000007 = 00001111
```

รูปที่ 2.20 เอาท์พุทไบนารีของ Test bench รูปที่ 2.18



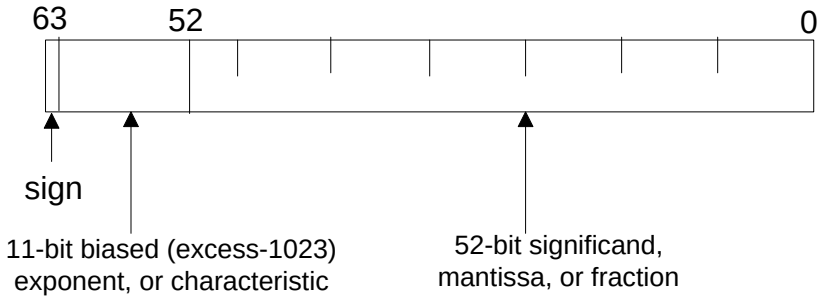
รูปที่ 2.21 ผลรูปคลื่นของ Test bench รูปที่ 2.18

รีจิสเตอร์แบบ **integer** เป็นรีจิสเตอร์อเนกประสงค์ ถูกใช้สำหรับการคำนวณและการจัดการข้อมูล มีขนาดอย่างน้อย 32 บิท และถูกกำหนดในรูปแบบ 2's complement ตัวอย่างเช่นจำนวนเต็ม +24 และ -24 สามารถถูกเขียนในรูป 2's complement ได้ดังนี้

$0000\_0000\_0000\_0000\_0000\_0000\_0001\_1000_2$ (+24)

$1111\_1111\_1111\_1111\_1111\_1111\_1110\_1000_2$ (-24)

ค่าคงที่จำนวนจริงและรีจิสเตอร์จำนวนจริงสามารถถูกประกาศได้โดยคำหลัก **real** ข้อมูลจำนวนจริงถูกจัดเก็บในรูปแบบเลขทศนิยมลอยตัวความเที่ยงสองเท่าขนาด 64 บิทแสดงดังรูปที่ 2.22



รูปที่ 2.22 รูปแบบเลขทศนิยมลอยตัวความเที่ยงสองเท่าขนาด 64 บิท

รีจิสเตอร์เวลาถูกใช้สำหรับการเก็บข้อมูลเวลาของการจำลองแบบ ซึ่งถูกประกาศได้ดังนี้

time time_1, time_2, …, time_n [msb:lsb];

ตัวแปรเวลาถูกเก็บในรูปแบบไม่คิดเครื่องหมาย (unsigned) ขนาดอย่างน้อย 64 บิท ซิสเต็มฟังก์ชัน **$time** ถูกใช้สำหรับการรับค่าเวลาจำลองแบบปัจจุบัน คำหลัก **realtime** จะเก็บค่าเวลาในรูปแบบจำนวนจริง

1.15 ตัวชี้แนะคอมไพเลอร์

ตัวชี้แนะคอมไพเลอร์ (compiler directives) ถูกใช้เป็นตัวสื่อสารระหว่างผู้ใช้กับคอมไพเลอร์ เพื่อช่วยให้คอมไพเลอร์ทำงานได้ง่ายขึ้น ตัวชี้แนะคอมไพเลอร์ในภาษา Verilog ที่ใช้บ่อยมีดังนี้

`define** ถูกใช้เพื่อสร้างข้อความนิยามค่าตัวเลขคล้ายในภาษา C ในขณะที่ **`undefine ถูกใช้เพื่อยกเลิกข้อความที่ถูกนิยามไว้แล้ว ตัวอย่างการใช้งานเป็นดังนี้

```
`define   bus_size  16          //define a bus of 16 bits
          …
//addr_in bus is 16 bits wide, 15:0
reg [`bus_size-1:0] addr_in;
          …
`undefined bus_size             //definition of bus_size is removed
```

ไม่ต้องมีสัญลักษณ์ ; หลังการประกาศ แต่ต้องมีสัญลักษณ์ ` ที่ข้อความที่ถูกกำหนดไว้เมื่อมีการอ้างอิง

`ifdef, `else, `endif** ถูกใช้สำหรับการคอมไพล์แบบมีเงื่อนไข ในการใช้งาน **`ifdef จะถูกตามด้วยชื่อข้อความมาโครที่ถูกนิยามไว้โดย **`define** ดังตัวอย่างข้างล่าง

```
`define Text_Macro_Name
          …
`ifdef Text_Macro_Name
          Statement_1;
          Statement_2;
                    …
          Statement_n;
`else
          Alternative_statement_1;
          Alternative_statement_2;
                    …
          Alternative_statement_n;
`endif
```

โดย **`else** อาจจะไม่มีก็ได้ถ้ามีทางเลือกเดียว

`include ถูกใช้เพื่อดึงไฟล์ Verilog อื่นเข้ามาในไฟล์ที่อยู่ภายใต้การคอมไพล์

`resetall ถูกใช้เพื่อรีเซทตัวชี้แนะคอมไพเลอร์ทุกตัวให้มีค่าเท่ากับค่าโดยปริยายของมัน (default values) ตัวอย่างเช่น ชนิดข้อมูลโดยปริยายของเน็ทก็คือ **wire**

`timescale ถูกใช้เพื่อกำหนดหน่วยของดีเลย์ ตัวอย่างการใช้งานเช่น

`timescale 10ns / 10ps

หมายถึงการกำหนดให้หนึ่งหน่วยเวลามีค่าเท่ากับ 10 นาโนวินาที และมีความละเอียดเท่ากับ 10 พิโควินาที หน่วยเวลาและความละเอียดในภาษา Verilog ดังตารางที่ 2.6

ตารางที่ 2.6 หน่วยเวลาและความละเอียดในภาษา Verilog

Time units	Definition
s	Seconds
ms	Milliseconds
us	Microseconds
ns	Nanoseconds
ps	Picoseconds
fs	Femtoseconds

โจทย์

2.1 Obtain the module, test bench, binary outputs, and waveforms for the equations shown below. Use built-in primitives and declare any internal wires.

$$z_1 = (x_1 \oplus x_2)x_3'$$
$$z_2 = (x_1 \oplus x_2)' \oplus x_3$$

2.2 Load a 16-byte data cache called *dcache* with the hexadecimal characters 80, C0, E0, F0, F8, FC, FE, FF, 7F, 3F, 1F, 0F, 07, 03, 01, 00. Then design a test bench and obtain the binary outputs and waveforms. The data cache has a vector input called *dc_addr[3:0]* and a vector output called *dc_data_out [7:0]*. Generate a hexadecimal file called *dcache.data* that will be loaded into the data cache.

บทที่ 3
นิพจน์

นิพจน์ (Expressions) ประกอบด้วยตัวถูกดำเนินการ (Operands) และตัวดำเนินการ (Operators) โดยผลลัพธ์จากการคำนวณของนิพจน์ทางด้านขวาสามารถถูกกำหนดค่าให้กับตัวแปรที่เป็นเน็ทหรือรีจิสเตอร์ทางซ้ายมือ ซึ่งนิพจน์หนึ่งๆสามารถประกอบด้วยตัวถูกดำเนินการเพียงตัวเดียวหรือมากกว่า ซึ่งมีตัวดำเนินการเพียงตัวเดียวหรือมากกว่าก็ได้ ผลลัพธ์จากการดำเนินการอาจจะมีเพียงบิตเดียวหรือมากกว่า

1.16 ตัวถูกดำเนินการ

ตัวถูกดำเนินการสามารถเป็นข้อมูลชนิดใดก็ได้ตามตารางที่ 3.1

ตารางที่ 3.1 ตัวถูกดำเนินการในภาษา Verilog

ตัวถูกดำเนินการ	รายละเอียด
ค่าคงที่ (constant)	แบบคิดเครื่องหมาย (signed) หรือแบบไม่คิดเครื่องหมาย (unsigned)
พารามิเตอร์ (parameter)	แบบคิดเครื่องหมาย (signed) หรือแบบไม่คิดเครื่องหมาย (unsigned)
เน็ท (net)	แบบสเกลาร์ หรือ เวกเตอร์
รีจิสเตอร์ (register)	แบบสเกลาร์ หรือ เวกเตอร์
เลือกบิต (bit-select)	เลือก 1 บิตจากเวกเตอร์
เลือกบางส่วน (part-select)	เลือกบางส่วนของบิตที่ติดกันจากเวกเตอร์
ส่วนย่อยของหน่วยความจำ (memory element)	หนึ่งเวิร์ดของหน่วยความจำ
เรียกฟังก์ชัน (function call)	การเรียกใช้ฟังก์ชันที่ผู้ใช้สร้างขึ้น หรือเรียกซิสเต็มฟังก์ชัน

a. ค่าคงที่ (Constant)

ค่าคงที่สามารถเป็นได้ทั้งแบบคิดเครื่องหมายหรือแบบไม่คิดเครื่องหมาย จำนวนเต็มฐานสิบถูกพิจารณาเป็นจำนวนแบบคิดเครื่องหมาย จำนวนเต็มที่ถูกกำหนดด้วยเลขฐานจะถูกแปลงเป็นจำนวนแบบไม่คิดเครื่องหมาย ดังตัวอย่างในตารางที่ 3.2

สองตัวอย่างสุดท้ายในตารางที่ 3.2 มีค่าเท่ากันเมื่อแทนด้วยเลขฐานสอง แต่แทนค่าเลขฐานสิบที่แตกต่างกัน -22_{10} เป็นจำนวนฐานสิบแบบคิดเครื่องหมาย ในขณะที่จำนวน -9'o352 ถูกมองเป็นจำนวนแบบไม่คิดเครื่องหมายซึ่งมีค่าฐานสิบเป็น 234_{10}

ตารางที่ 3.2 ค่าคงที่แบบคิดเครื่องหมายและแบบไม่คิดเครื่องหมาย

ค่าคงที่	รายละเอียด
127	Signed decimal: value = 8-bit binary vector: 01111111
-1	Signed decimal: value = 8-bit binary vector: 11111111

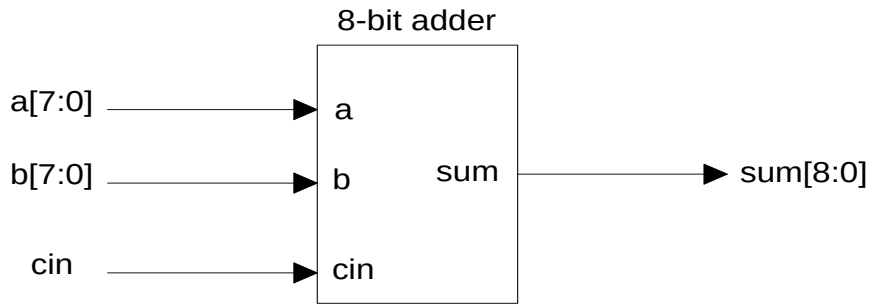
ตารางที่ 3.2 ค่าคงที่แบบคิดเครื่องหมายและแบบไม่คิดเครื่องหมาย (ต่อ)

-128	Signed decimal: value = 8-bit binary vector: 10000000
4'b1110	Binary base: value = unsigned decimal 14
8'b00111010	Binary base: value = unsigned decimal 58
16'h1A3C	Hexadecimal base: value = unsigned decimal 6,716
16'hBCDE	Hexadecimal base: value = unsigned decimal 48,350
9'o536	Octal base: value = unsigned decimal 350
-22	Signed decimal: value = 8-bit binary vector: 11101010
-9'o352	Octal base: value = 8-bit binary vector: 11101010 = unsigned decimal 234

b. พารามิเตอร์ (Parameter)

จำนวนในพารามิเตอร์เหมือนกับจำนวนในค่าคงที่ คำสั่งพารามิเตอร์ใช้คำหลัก **parameter** เป็นการกำหนดค่าคงที่ให้กับตัวแปร ซึ่งค่านี้จะไม่สามารถถูกเปลี่ยนแปลงได้ในระหว่างการจำลองแบบการทำงาน

พารามิเตอร์มีประโยชน์ในการนิยามความกว้างของบัส ดังตัวอย่างในรูปที่ 3.1 วงจรบวกมีอินพุทแบบเวกเตอร์ขนาด 8 บิตจำนวน 2 อินพุท และมีอินพุทแบบสเกลาร์ 1อินพุท ผลบวกที่ได้เป็นจำนวนแบบเวกเตอร์ 9 บิต โค๊ด Verilog สำหรับวงจรบวกนี้แสดงดังรูปที่ 3.2 คำสั่งพารามิเตอร์ถูกใช้นิยามความกว้างของบัส width = 8 บิต เมื่อมีการใช้ width ที่ใดก็ตามในโค๊ดจะหมายถึงจำนวน 8 รูปที่ 3.3 และ 3.4 แสดง test bench และผลการจำลองการทำงานของวงจรบวก ตามลำดับ



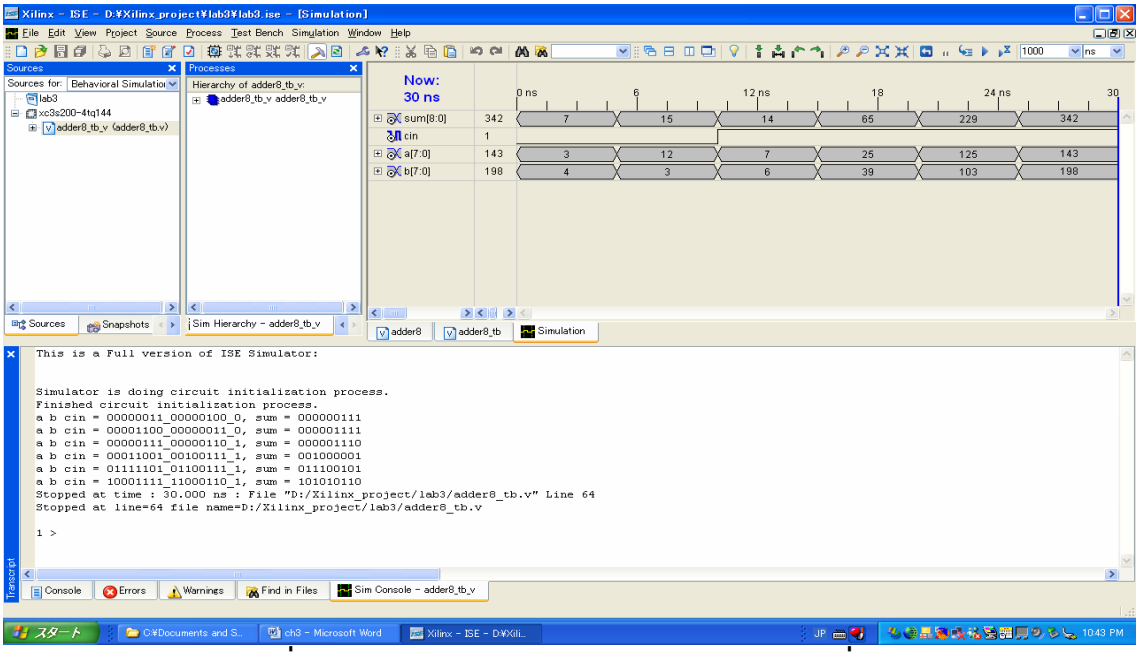
รูปที่ 3.1 วงจรบวกขนาด 8 บิต แสดงการใช้งาน parameter

```
//example of using paramerter
module adder8 (a, b, cin, sum);
parameter   width = 8;
input   [width-1:0] a, b;
input   cin;
output reg   [width:0] sum;
always @(a or b or cin)
    begin
        sum = a + b + cin;
    end
endmodule
```

รูปที่ 3.2 โค๊ด Verilog สำหรับวงจรบวกรูปที่ 3.1

```
//adder8 test bench
module adder8_tb_v;
    parameter width = 8;
        reg [width-1:0] a, b;
        reg cin;

        wire [width:0] sum;

// Instantiate the Unit Under Test (UUT)
   adder8 uut (.a(a), .b(b), .cin(cin), .sum(sum));
//display variables
   initial
   $monitor ("a b cin = %b_%b_%b,
     sum = %b", a, b, cin, sum);
 //apply input vectors
   initial
     begin
       #0      a = 8'b00000011;
               b = 8'b00000100;
               cin = 1'b0;
       #5      a = 8'b00001100;
               b = 8'b00000011;
               cin = 1'b0;
       #5      a = 8'b00000111;
               b = 8'b00000110;
               cin = 1'b1;
       #5      a = 8'b00011001;   //25
               b = 8'b00100111;   //39
               cin = 1'b1;   //sum = 65
       #5      a = 8'b01111101;   //125
               b = 8'b01100111;   //103
               cin = 1'b1;   //sum = 229
       #5      a = 8'b10001111;   //143
               b = 8'b11000110;   //198
               cin = 1'b1;  //sum = 342
       #5      $stop;
     end
endmodule
```

รูปที่ 3.3 Test bench สำหรับวงจรบวกรูปที่ 3.1



รูปที่ 3.4 ผลการจำลองการทำงานสำหรับวงจรบวกรูปที่ 3.1

c. เน็ท (Net)

เน็ทสามารถเป็นทั้งสเกลาร์ (บิตเดียว) และเวกเตอร์ (หลายบิต) ถูกใช้เพื่อเชื่อมต่ออุปกรณ์ฮาร์ดแวร์ต่างๆ ค่าที่กำหนดให้กับเน็ทถูกแปลเป็นค่าที่ไม่คิดเครื่องหมาย ดังตัวอย่างด้านล่าง

```
wire    [7:0]    bus_in, bus_out;
…
assign      bus_in = -59;              //(11000101)2 = 19710 unsigned
assign      bus_out = 16'he0;          //(11000000)2 = -3210 signed; 224 unsigned
```

d. รีจิสเตอร์ (Register)

รีจิสเตอร์เป็นตัวแทนของอุปกรณ์หน่วยความจำ และถูกประกาศด้วยคำหลัก **reg integer time real** หรือ **realtime** รีจิสเตอร์สามารถเก็บทั้งปริมาณสเกลาร์และเวกเตอร์ ค่าที่ถูกนิยามด้วย **reg** จะถูกแปลความหมายเป็นจำนวนแบบไม่คิดเครื่องหมายซึ่งเก็บค่าลอจิกในฟลิบฟลอบหรือแลทช์ ในขณะที่ค่าในรีจิสเตอร์ที่ถูกประกาศด้วย **integer** จะเก็บจำนวนแบบคิดเครื่องหมายในรูปแบบ 2's complement ดังตัวอย่างด้านล่าง (รายละเอียดของ **time real** หรือ **realtime** ทบทวนได้จากหัวข้อ 2.5.2)

```
reg       [15:0]    bus_out;
integer       [15:0] accumulator;
…
initial
begin
    bus_out = -20;          // -20 คือ 1111_1111_1110_1100 แบบคิดเครื่องหมาย
                            // ซึ่งก็คือ 65,516 แบบไม่คิดเครื่องหมาย
    accumulator = -57;      // -57 คือ 1111_1111_1100_0111 แบบคิดเครื่องหมาย
end
```

e. เลือกบิต (Bit-select)

การเลือกบิตเป็นการบิตเดี่ยวออกจากเวกเตอร์ของเน็ทหรือรีจิสเตอร์ เพื่อใช้ในการดำเนินการ (operation) ตัวอย่างเช่นเพื่อใช้ในการหาเครื่องหมายของตัวถูกดำเนินการ 16-บิต บิต 15 จะถูกทดสอบว่าเป็น 0 หรือ 1 ซึ่งจะหมายถึงจำนวนบวกและลบตามรูปแบบของ 2's complement โดยบิตที่ถูกเลือกสามารถถูกกำหนดด้วยค่าคงที่หรือนิพจน์ก็ได้ ดังตัวอย่างด้านล่าง

```
wire    [15:0] bus_out;      // บัส 16-บิต โดยมี bus_out[15] เป็นบิตสูงสุด
assign bus_out[14] = a & b;    // เลือกบิต 14
assign bus_out[x1+3] = 1'b0;    // เลือกบิตของ bus_out โดยขึ้นกับค่า x1+3
```

f. เลือกบางส่วน (Part-select)

การเลือกบางส่วนเป็นการเลือกบิตหลายบิตที่ติดกันจากเวกเตอร์ของเน็ทหรือรีจิสเตอร์เพื่อใช้ในการดำเนินการ โดยมีรูปแบบการเลือกดังตัวอย่างการกำหนดให้ออปโค้ดคือ 8 บิตบนหรือไบท์ซ้ายมือสุดของรีจิสเตอร์คำสั่ง

reg [7:0] opcode;

reg [31:0] instr; // คำสั่ง 32-บิต

opcode = instr[31:24]; // เลือก 8 บิตบนหรือไบท์ซ้ายมือสุดของรีจิสเตอร์คำสั่ง

g. ส่วนย่อยของหน่วยความจำ (Memory element)

ส่วนย่อยของหน่วยความจำหมายถึงเวิร์ด (word) ในหน่วยความจำ ซึ่งอาจจะเป็นบิตเดียวหรือหลายบิต หน่วยความจำถูกอ้างอิงโดยการใช้เวิร์ดเท่านั้น ไม่มีการเลือกบิตหรือเลือกบางส่วน แต่ละบิตในหน่วยความจำไม่สามารถถูกเข้าถึงโดยตรงได้ จำเป็นต้องนำค่าในเวิร์ดที่ถูกระบุโดยแอดเดรสมาเก็บที่ word register เสียก่อนจึงจะสามารถเข้าถึงแต่ละบิตได้ ตัวอย่างเช่น หน่วยความจำ icache ขนาด 2048-เวิร์ด แต่ละเวิร์ดมี 32 บิต ดังแสดงด้านล่าง รีจิสเตอร์ขนาด 32 บิต instr_reg ถูกใช้เพื่อการเข้าถึงแต่ละบิตในเวิร์ดหนึ่งๆ ของหน่วยความจำ

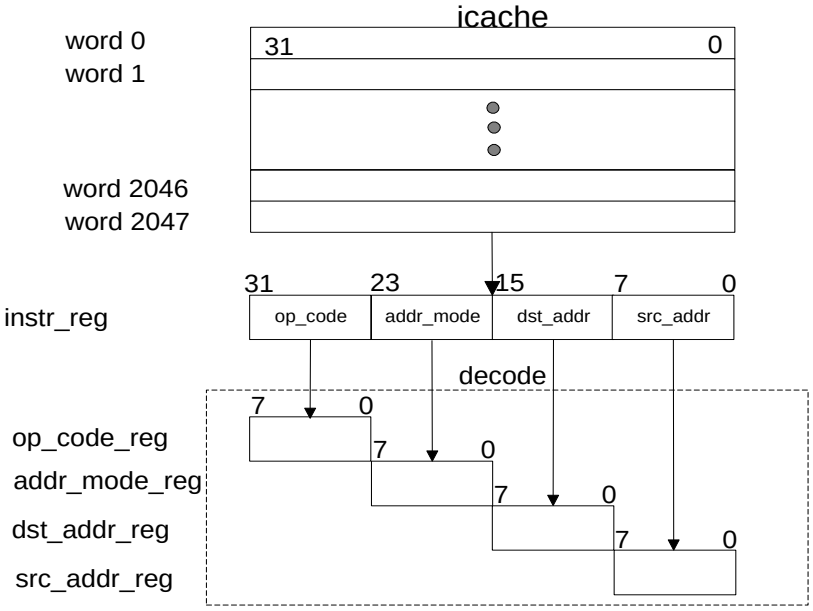
reg [31:0] icache [0:2047];

reg [31:0] instr_reg;

รูปที่ 3.5 แสดงตัวอย่างการเข้าถึงบางบิตจากหน่วยความจำในโพรเซสเซอร์แบบ RISC (Reduced Instruction Set Computer) หน่วยความจำ icache ทำหน้าที่เก็บคำสั่ง สามารถถูกเข้าถึงได้โดยการนำค่าในเวิร์ดที่ถูกระบุโดยแอดเดรสมาเก็บที่ instr_reg จากนั้นก็ถูกส่งไปยังหน่วย decode ซึ่งมีการเก็บค่าคำสั่งแยกเป็นส่วนๆ คือ op_code addr_mode dst_addr และ src_addr เพื่อที่จะเลือกเวิร์ดที่แอดเดรส 127 (เวิร์ดที่ 128) จาก icache และนำ op_code มาเก็บที่ op_code_reg ทำได้ดังนี้

instr_reg = icache [127];

op_code_reg = instr_reg[31:24];



รูปที่ 3.5 บล็อกไดอะแกรมของบางส่วนในโพรเซสเซอร์ RISC **เพื่อแสดงเข้าถึงบางบิตจากหน่วยความจำ**

1.17 ตัวดำเนินการ

ภาษา Verilog มีตัวดำเนินการมากมายหลายประเภท บางประเภทมีความคล้ายกับภาษา C ตามรายการในตารางที่ 3.3

ตารางที่ 3.3 ตัวดำเนินการในภาษา Verilog

ประเภทตัวดำเนินการ	สัญลักษณ์	การดำเนินการ	จำนวนตัวถูกดำเนินการ
ตัวดำเนินการเลขคณิต	+	บวก	1 หรือ 2
(Arithmetic)	-	ลบ	1 หรือ 2
	*	คูณ	2
	/	หารแบบไม่คิดเศษ	2
	%	เศษ	2
ตัวดำเนินการตรรกะ	&&	และ	2
(Logical)	||	หรือ	2
	!	นิเสธ	1
ตัวดำเนินการสัมพันธ์	>	มากกว่า	2
(Relational)	<	น้อยกว่า	2
	>=	มากกว่า หรือ เท่ากับ	2
	<=	น้อยกว่า หรือ เท่ากับ	2
ตัวดำเนินการเท่ากัน	==	เท่ากัน	2
(Equality)	!=	ไม่เท่ากัน	2
	===	กรณีเท่ากัน (case equality)	2
	!==	กรณีไม่เท่ากัน (case inequality)	2
ตัวดำเนินการบิต	&	AND	2
(Bitwise)	|	OR	2
	~	NOT	2
	^	Exclusive-OR	2
	^~ หรือ ~^	Exclusive-NOR	2
ตัวดำเนินการลด	&	AND	1
(Reduction)	~&	NAND	1
	|	OR	1
	~|	NOR	1
	^	Exclusive-OR	1
	^~ หรือ ~^	Exclusive-NOR	1
ตัวดำเนินการเลื่อน	<<	เลื่อนซ้าย	1
(Shift)	>>	เลื่อนขวา	1
ตัวดำเนินการเงื่อนไข (Conditional)	?:	เงื่อนไข	3
ตัวดำเนินการต่อกัน (Concatenation)	{,}	ต่อกัน	2 หรือ 3
ตัวดำเนินการทำซ้ำ (Replication)	{{ }}	ทำซ้ำ	2 หรือ 3

3.2.1 ตัวดำเนินการเลขคณิต

การดำเนินการเลขคณิตสามารถเป็นการดำเนินการที่มีตัวถูกดำเนินการเพียงตัวเดียวหรือสองตัว เลขฐานในการดำเนินการอาจจะเป็น ฐานสอง ฐานแปด ฐานสิบ หรือฐานสิบหก ผลลัพธ์จากการดำเนินการถูกแปลงเป็นจำนวนแบบไม่คิดเครื่องหมาย หรือแบบคิดเครื่องหมายในรูปแแบบ 2's complement รูปที่ 3.6 แสดงตัวอย่างการใช้งานตัวดำเนินการเลขคณิต พร้อมด้วย test bench และผลรูปคลื่นในรูปที่ 3.7 และ 3.8 ตามลำดับ

```
//demonstrate arithmetic operations
module arith_ops1(a, b, opcode, result);
input    [3:0]  a,b;
input    [2:0]  opcode;
output reg [7:0]  result;

parameter addop = 3'b000,
          subop = 3'b001,
          mulop = 3'b010,
          divop = 3'b011,
          modop = 3'b100;
```

```
always @(a or b or opcode)
begin
   case(opcode)
                addop: result = a + b;
                subop: result = a - b;
                mulop: result = a * b;
        //(*)   divop: result = a / b;
        //(*)   modop: result = a % b;
                default: result = 8'bxxxxxxxx;
        endcase
end
endmodule
```

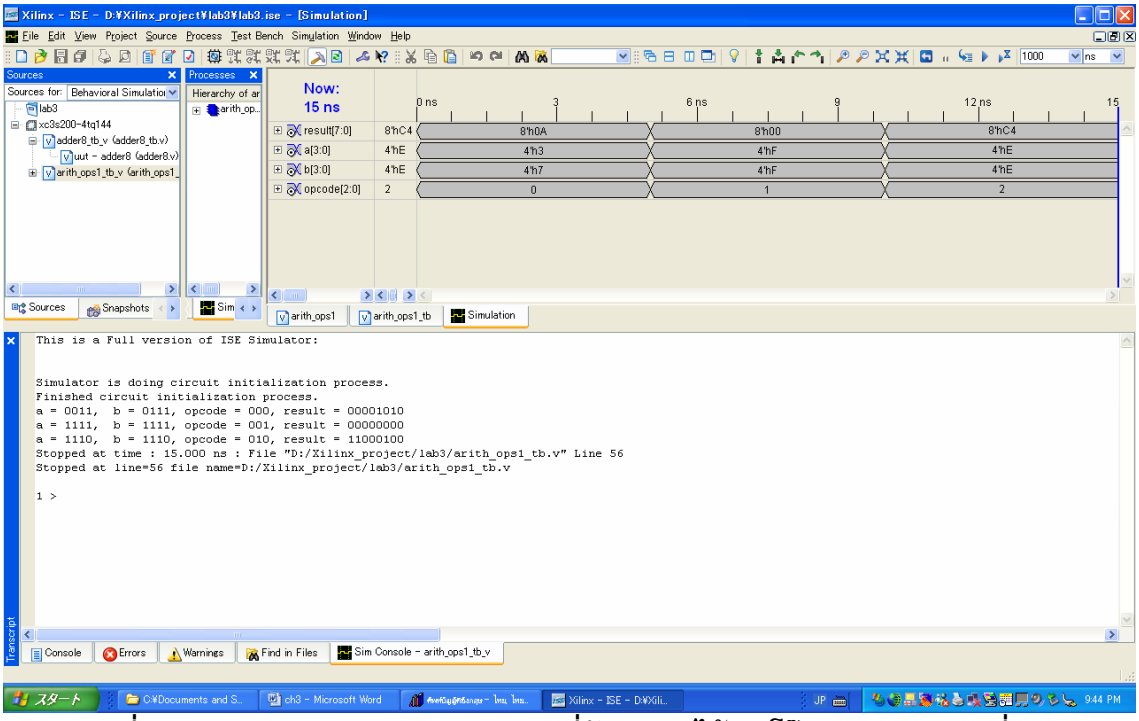
รูปที่ 3.6 โค๊ด Verilog สำหรับแสดงตัวอย่างการใช้งานตัวดำเนินการเลขคณิต

```
//Arithmetic operations test bench
module arith_ops1_tb_v;
        // Inputs
        reg [3:0] a, b;
        reg [2:0] opcode;

        // Outputs
        wire [7:0] result;

        // Instantiate the Unit Under Test (UUT)
        arith_ops1 uut (
                .a(a),
                .b(b),
                .opcode(opcode),
                .result(result)
        );
```

```
initial
$monitor ("a = %b,   b = %b, opcode = %b,
result = %b", a, b, opcode, result);
initial
begin
     #0   a = 4'b0011;
          b = 4'b0111;
          opcode = 3'b000;
     #5   a = 4'b1111;
          b = 4'b1111;
          opcode = 3'b001;
     #5   a = 4'b1110;
          b = 4'b1110;
          opcode = 3'b010;
   #5 $stop;
end
endmodule
```

รูปที่ 3.7 Test bench สำหรับโค๊ด Verilog ของรูปที่ 3.6

หมายเหตุ โปรแกรม Xilinx XST ไม่รองรับการหารและการคำนวณเศษ แต่โปรแกรมสามารถสังเคราะห์วงจรหารที่หารด้วย 2 และเลขยกกำลังของ 2



รูปที่ 3.8 ผลการจำลองการทำงานของวงจรที่สังเคราะห์ได้จากโค๊ด Verilog ของรูปที่ 3.6

3.2.2 ตัวดำเนินการตรรกะ

ตัวดำเนินการตรรกะคำนวณเพื่อให้ทราบว่าเป็น จริง (true) ซึ่งแทนด้วยลอจิก 1 เท็จ (false) ซึ่งแทนด้วยลอจิก 0 หรือ ไม่ทราบค่า (ambiguous) ถ้าผลลัพธ์จากการคำนวณได้ค่าที่ไม่เป็นศูนย์ กรณีนี้จะถือว่าเป็นลอจิก 1 (จริง) แต่ถ้าบิตใดบิตหนึ่งของตัวถูกดำเนินการเป็น **x** หรือ **z** กรณีนี้จะถือว่าเป็นไม่ทราบค่าและจะถูกกำหนดให้เป็นเท็จ กรณีตัวถูกดำเนินการเป็นเวกเตอร์ (มีมากกว่า 1 บิต) ถ้ามีค่าไม่เป็นศูนย์จะถูกพิจารณาให้เป็นลอจิก 1 (จริง) รูปที่ 3.9 แสดงตัวอย่างการใช้งานตัวดำเนินการตรรกะ พร้อมด้วย test bench และผลรูปคลื่นในรูปที่ 3.10 และ 3.11 ตามลำดับ

3.2.3 ตัวดำเนินการสัมพันธ์

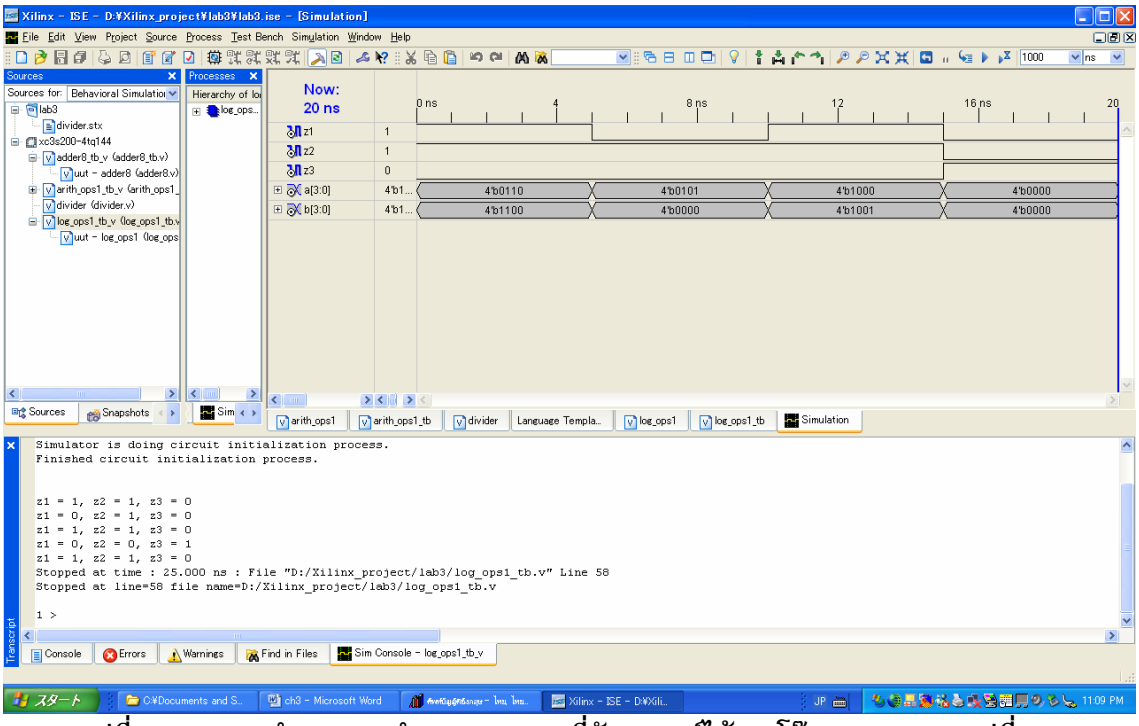
ตัวดำเนินการสัมพันธ์ทำการเปรียบเทียบค่าของตัวถูกดำเนินการ ให้ผลลัพธ์เป็นลอจิก 1 (จริง) หรือ ลอจิก 0 (เท็จ) เพื่อบอกความสัมพันธ์ของตัวถูกดำเนินการ 2 ตัว ถ้าความสัมพันธ์เป็นจริงผลลัพธ์จะเป็นลอจิก 1 ไม่เช่นนั้นเป็นลอจิก 0 ตัวถูกดำเนินการที่เป็นเน็ท (ประกาศด้วย **wire**) หรือรีจิสเตอร์ (ประกาศด้วย **reg**) ถูกพิจารณาว่าเป็นจำนวนแบบไม่คิดเครื่องหมาย ในขณะที่ **real** หรือ **integer** ถูกพิจารณาว่าเป็นจำนวนแบบคิดเครื่องหมาย ในกรณีที่ตัวถูกดำเนินการมีจำนวนบิตไม่เท่ากัน ตัวดำเนินการที่มีบิตน้อยกว่าจะถูกเติมศูนย์เข้าไปด้านซ้ายเพื่อให้จำนวนบิตของตัวถูกดำเนินการทั้งสองเท่ากัน รูปที่ 3.12 แสดงตัวอย่างการใช้งานตัวดำเนินการสัมพันธ์ พร้อมด้วย test bench และผลรูปคลื่นในรูปที่ 3.13 และ 3.14 ตามลำดับ

```
//examples of logical operators
module log_ops1(a, b, z1, z2, z3);
input     [3:0] a, b;
output z1, z2, z3;

assign     z1 = a && b;
assign     z2 = a || b;
assign     z3 = !a;
endmodule
```

รูปที่ 3.9 โค๊ด Verilog สำหรับแสดงตัวอย่างการใช้งานตัวดำเนินการตรรกะ

```
//test bench for logical operators
module log_ops1_tb_v;

reg [3:0] a, b;    // Inputs

wire z1, z2, z3;        // Outputs

// Instantiate the Unit Under Test (UUT)
            log_ops1 uut (
                        .a(a),
                        .b(b),
                        .z1(z1),
                        .z2(z2),
                        .z3(z3)
            );
initial
$monitor ("z1 = %d, z2 = %d, z3 = %d",
            z1, z2, z3);

            //apply input vectors
            initial
            begin
              #0        a = 4'b0110;
                        b = 4'b1100;
              #5        a = 4'b0101;
                        b = 4'b0000;
                #5      a = 4'b1000;
                        b = 4'b1001;
                #5      a = 4'b0000;
                         b = 4'b0000;
                #5      a = 4'b1111;
                         b = 4'b1111;
                 #5     $stop;
            end
endmodule
```

รูปที่ 3.10 Test bench สำหรับโค๊ด Verilog ของรูปที่ 3.9



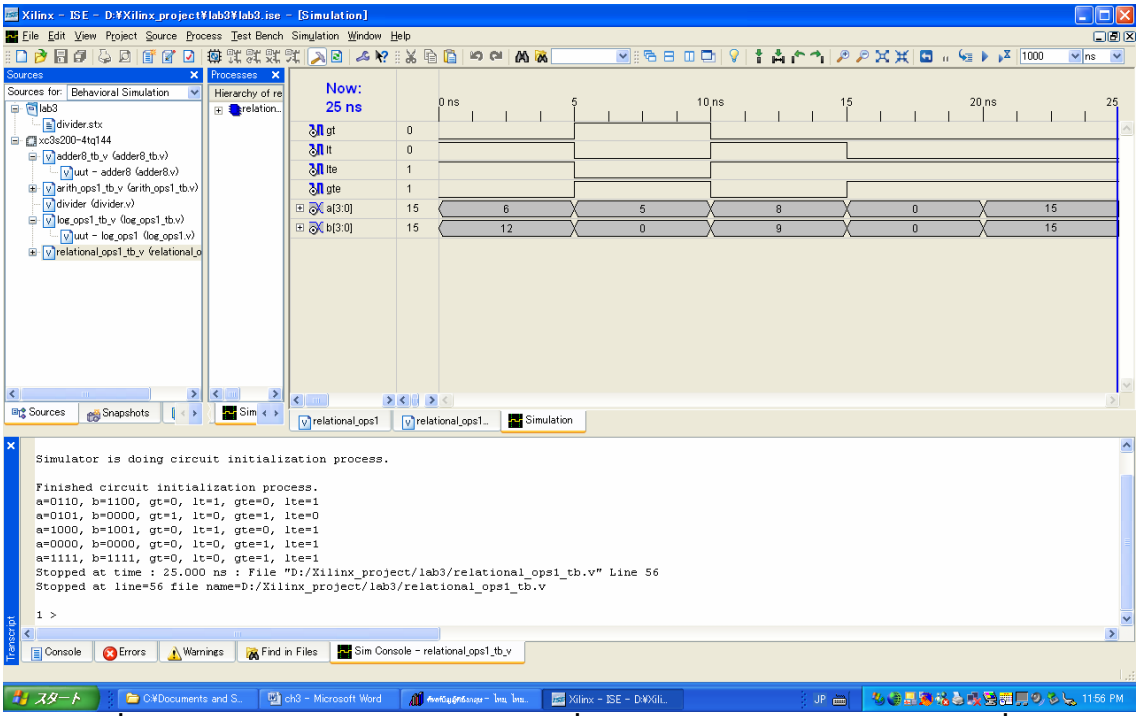
รูปที่ 3.11 ผลการจำลองการทำงานของวงจรที่สังเคราะห์ได้จากโค๊ด Verilog ของรูปที่ 3.9

```
//examples of relational operations
module relational_ops1(a, b, gt, lt, gte, lte);
input [3:0] a, b;
output gt, lt, gte, lte;

assign    gt = a > b;
assign    lt = a < b;
assign    gte = a >= b;
assign    lte = a <= b;
endmodule
```

รูปที่ 3.12 โค๊ด Verilog สำหรับแสดงตัวอย่างการใช้งานตัวดำเนินการสัมพันธ์

```
//test bench relational operations
module relational_ops1_tb_v;
reg [3:0] a, b;          // Inputs
wire gt, lt, gte, lte;   // Outputs

// Instantiate the Unit Under Test (UUT)
        relational_ops1 uut (
                .a(a),
                .b(b),
                .gt(gt),
                .lt(lt),
                .gte(gte),
                .lte(lte)
        );
initial
$monitor ("a=%b, b=%b, gt=%d, lt=%d, gte=%d, lte=%d", a, b, gt, lt, gte, lte);

//apply input vectors
initial
begin
        #0      a = 4'b0110;
                b = 4'b1100;
        #5      a = 4'b0101;
                b = 4'b0000;
        #5      a = 4'b1000;
                b = 4'b1001;
        #5      a = 4'b0000;
                b = 4'b0000;
        #5      a = 4'b1111;
                b = 4'b1111;
        #5      $stop;
end
endmodule
```

รูปที่ 3.13 Test bench สำหรับโค๊ด Verilog ของรูปที่ 3.12



รูปที่ 3.14 ผลการจำลองการทำงานของวงจรที่สังเคราะห์ได้จากโค๊ด Verilog ของรูปที่ 3.12

3.2.4 ตัวดำเนินการเท่ากัน

ตัวดำเนินการเท่ากัน (==) ถูกใช้ในนิพจน์เพื่อตรวจสอบว่าจำนวนสองจำนวนเท่ากันหรือไม่ ผลลัพธ์ของการเปรียบเทียบเป็นลอจิก 1 ถ้าทั้งสองจำนวนเท่ากัน และเป็นลอจิก 0 เมื่อสองจำนวนไม่เท่ากัน ในทางตรงข้ามตัวดำเนินการไม่เท่ากัน (!=) จะให้ผลตรงกันข้าม ผลลัพธ์ของการเปรียบเทียบเป็นลอจิก 0 ถ้าทั้งสองจำนวนเท่ากัน และเป็นลอจิก 1 เมื่อสองจำนวนไม่เท่ากัน ถ้าผลการเปรียบเทียบคลุมเครือผลที่ได้จะเป็น **x** ถ้าตัวถูกดำเนินการเป็นเน็ทหรือรีจิสเตอร์จะถูกจัดว่าเป็นจำนวนแบบไม่คิดเครื่องหมาย แต่ถ้าเป็น real หรือ integer จะถูกจัดว่าเป็นจำนวนแบบคิดเครื่องหมาย

ตัวดำเนินการกรณีเท่ากัน (case equality; ===) ทำการเปรียบเทียบบิตต่อบิตเช่นเดียวกัน และยังรวมถึงกรณีของ **x** และ **z** ด้วย ผลลัพธ์ของการเปรียบเทียบเป็นลอจิก 1 เมื่อทุกบิตของทั้งสองตัวถูกดำเนินการเท่ากันซึ่งพิจารณารวมถึงบิตที่เป็น **x** และ **z** ด้วย ส่วนตัวดำเนินการกรณีไม่เท่ากัน (case inequality; !==) จะให้ผลลัพธ์ลอจิก 1 เมื่อมีบางบิตของทั้งสองตัวถูกดำเนินการไม่เท่ากันซึ่งพิจารณารวมถึงบิตที่เป็น **x** และ **z** ด้วย

ตัวอย่างการใช้งานตัวดำเนินการเท่ากันถูกแสดงดังรูปที่ 3.15 รูป พร้อมด้วย test bench และผลรูปคลื่นในรูปที่ 3.16 และ 3.17 ตามลำดับ ผลลัพธ์ของ case equality (z3) เป็นลอจิก 1 ในอินพุทเวกเตอร์ชุดที่ 1 และ 2 เนื่องจากทุกบิตของ x3 และ x4 เท่ากันทุกบิต ส่วนผลลัพธ์ของ case inequality (z4) เป็นลอจิก 1 ในอินพุทเวกเตอร์ทั้ง 3 ชุด แม้แต่ในชุดที่ 3 ที่มีบิตต่ำสุดของ x4 และ x5 เท่านั้นที่ต่างกัน โดยบิตต่ำสุดของ x4 เป็น **x** และบิตต่ำสุดของ x5 เป็น **z**

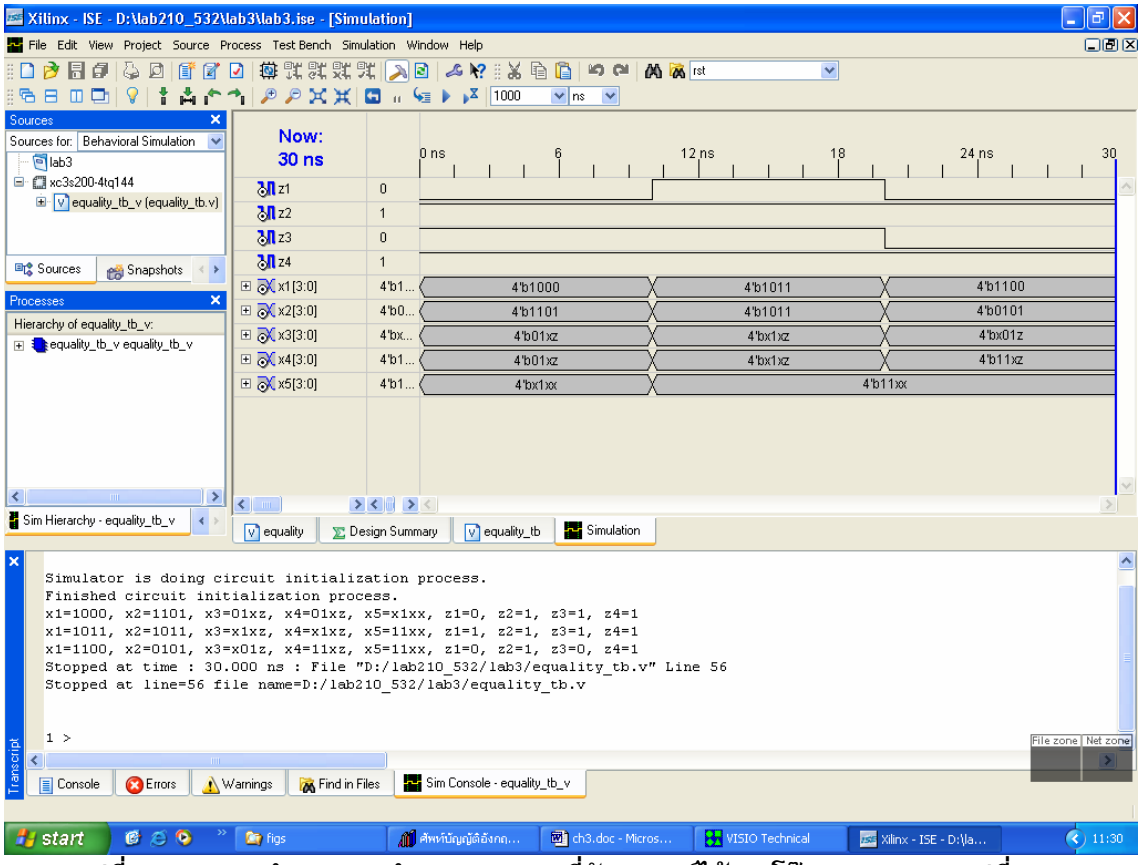
```
//illustrate the use of equality operators
module equality(x1, x2, x3, x4, x5, z1, z2, z3, z4);
input     [3:0]    x1, x2, x3, x4, x5;
output reg         z1, z2, z3, z4;

always @(x1 or x2 or x3 or x4 or x5)
begin
        if (x1 == x2)          //logical equality
            z1 = 1;
        else z1 = 0;
end

always @(x1 or x2 or x3 or x4 or x5)
begin
        if (x2 != x3)          //logical inequality
            z2 = 1;
        else z2 = 0;
end

always @(x1 or x2 or x3 or x4 or x5)
begin
        if (x3 === x4)         //case equality
            z3 = 1;
        else z3 = 0;
end

always @(x1 or x2 or x3 or x4 or x5)
begin
        if (x4 !== x5)         //case inequality
            z4 = 1;
        else z4 = 0;
end
endmodule
```

รูปที่ 3.15 โค๊ด Verilog สำหรับแสดงตัวอย่างการใช้งานตัวดำเนินการเท่ากัน

```
//equality operators test bench
module equality_tb_v;
reg [3:0] x1, x2, x3, x4, x5;     // Inputs
wire z1, z2, z3, z4;              // Outputs

// Instantiate the Unit Under Test (UUT)
        equality uut (.x1(x1), .x2(x2), .x3(x3),
        .x4(x4), .x5(x5), .z1(z1), .z2(z2),
        .z3(z3), .z4(z4));

initial
$monitor ("x1=%b, x2=%b, x3=%b, x4=%b,
x5=%b, z1=%b, z2=%b, z3=%b, z4=%b",
x1, x2, x3, x4, x5, z1, z2, z3, z4);
```

```
//apply input vectors
initial
begin
    #0    x1 = 4'b1000;
          x2 = 4'b1101;
          x3 = 4'b01xz;
          x4 = 4'b01xz;
          x5 = 4'bx1xx;
    #10   x1 = 4'b1011;
          x2 = 4'b1011;
          x3 = 4'bx1xz;
          x4 = 4'bx1xz;
          x5 = 4'b11xx;
    #10   x1 = 4'b1100;
          x2 = 4'b0101;
          x3 = 4'bx01z;
          x4 = 4'b11xz;
          x5 = 4'b11xx;
    #10   $stop;
end
endmodule
```

รูปที่ 3.16 Test bench สำหรับโค๊ด Verilog ของรูปที่ 3.16



รูปที่ 3.17 ผลการจำลองการทำงานของวงจรที่สังเคราะห์ได้จากโค๊ด Verilog ของรูปที่ 3.16

3.2.5 ตัวดำเนินการบิต

ตัวดำเนินการบิตทำการดำเนินการทางลอจิกกับตัวดำเนินการแบบบิตต่อบิต โดยบิตที่มีหลักตรงกันของตัวดำเนินการสองตัวจะถูกดำเนินการ ผลลัพธ์ที่ได้ยังคงมีจำนวนความกว้างของบิตเท่ากับตัวดำเนินการที่มีจำนวนบิตมากกว่า ในการดำเนินการหากตัวถูกดำเนินการใดมีจำนวนบิตน้อยกว่า ศูนย์จะถูกเติมเข้าไปด้านหน้าของตัวถูกดำเนินการนั้นเพื่อให้ตัวถูกดำเนินการทั้งสองมีจำนวนบิตเท่ากัน

ตัวอย่างตัวดำเนินการบิต AND OR และ NOT เป็นดังนี้

```
      z1 = x1 & x2                  z2 = x1 | x2                  z3 = ~x1
x1=   1 0 1 1 0 1 1 0       x1=   1 0 1 1 0 1 1 0       x1=   1 0 1 1 0 1 1 0 (~
x2=   1 1 0 1 0 1 1 1 (&    x2=   1 1 0 1 0 1 1 1 (|    z3=   0 1 0 0 1 0 0 1
z1=   1 0 0 1 0 1 1 0       z2=   1 1 1 1 0 1 1 1
```

ตัวอย่างการใช้งานตัวดำเนินการบิตถูกแสดงดังรูปที่ 3.18 รูป พร้อมด้วย test bench และผลรูปคลื่นในรูปที่ 3.19 และ 3.20 ตามลำดับ

```
//example of the bitwise operators
module bitwise1 (a, b, and_result, or_result,
          neg_result, xor_result, xnor_result);

input    [7:0] a, b;
output reg [7:0] and_result, or_result, neg_result,
                 xor_result, xnor_result;
```

```
always @( a or b)
begin
  and_result = a & b; //bitwise AND
  or_result = a | b;     //bitwise OR
  neg_result = ~a;       //bitwise negation
  xor_result = a ^ b;    //bitwise XOR
  xnor_result = a ^~ b; //bitwise XNOR
end
endmodule
```

รูปที่ 3.18 โค๊ด Verilog สำหรับแสดงตัวอย่างการใช้งานตัวดำเนินการบิต

```
//test bench for bitwise1 module
module bitwise1_tb_v;
reg [7:0] a, b;     // Inputs
wire  [7:0] and_result, or_result, neg_result,
xor_result, xnor_result; // Outputs

// Instantiate the Unit Under Test (UUT)
bitwise1 uut (.a(a), .b(b), .and_result(and_result),
          .or_result(or_result),
          .neg_result(neg_result),
          .xor_result(xor_result),
          .xnor_result(xnor_result)
          );

initial
$monitor ("a=%b, b=%b, and_result=%b,
or_result=%b, neg_result=%b, xor_result=%b,
xnor_result=%b", a, b, and_result, or_result,
neg_result, xor_result, xnor_result);
```

```
//apply input vectors
initial
begin
   #0     a = 8'b11000011;
          b = 8'b10011001;
  #10     a = 8'b10010011;
          b = 8'b11011001;
  #10     a = 8'b00001111;
          b = 8'b11011001;
  #10     a = 8'b01001111;
          b = 8'b11011001;
  #10     a = 8'b11001111;
          b = 8'b11011001;
  #10   $stop;
end
endmodule
```

รูปที่ 3.19 Test bench สำหรับโค๊ด Verilog ของรูปที่ 3.18

```
a=11000011,b=10011001, and_result=10000001, or_result=11011011, neg_result=00111100,
                       xor_result=01011010, xnor_result=10100101
a=10010011,b=11011001, and_result=10010001, or_result=11011011, neg_result=01101100,
                       xor_result=01001010, xnor_result=10110101
a=00001111, b=11011001, and_result=00001001, or_result=11011111, neg_result=11110000,
                       xor_result=11010110, xnor_result=00101001
a=01001111, b=11011001, and_result=01001001, or_result=11011111, neg_result=10110000,
                       xor_result=10010110, xnor_result=01101001
a=11001111, b=11011001, and_result=11001001, or_result=11011111, neg_result=00110000,
                       xor_result=00010110, xnor_result=11101001
```

รูปที่ 3.20 ผลการจำลองการทำงานของวงจรที่สังเคราะห์ได้จากโค๊ด Verilog ของรูปที่ 3.18

3.2.6 ตัวดำเนินการลด

ตัวดำเนินการลดกระทำบนตัวถูกดำเนินการเพียงตัวเดียว หรืออินพุทเวกเตอร์เดียว ผลลัพธ์ที่ได้มีเพียงบิตเดียว ถ้ามีบิตใดบิตหนึ่งเป็น **x** หรือ **z** ผลลัพธ์ที่ได้จะเป็น **x** การดำเนินการจะทำบิตต่อบิตจากด้านซ้ายมายังบิตด้านขวา ดังตัวอย่างตัวดำเนินการลด AND NAND OR NOR XOR XNOR ด้านล่างนี้

Reduction AND z1 = &x1	Reduction NAND z2 = ~&x1
x1 = 1 1 1 0 1 0 1 1	x1 = 1 1 1 0 1 0 1 1
z1 = 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 = 0	z2 = 1 ~& 1 ~& 1 ~& 0 ~& 1 ~& 0 ~& 1 ~& 1 = 1
Reduction OR z3 = |x1	**Reduction NOR z4 = ~|x1**
x1 = 1 1 1 0 1 0 1 1	x1 = 1 1 1 0 1 0 1 1
z3 = 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 = 1	z4 = 1 ~| 1 ~| 1 ~| 0 ~| 1 ~| 0 ~| 1 ~| 1 = 0
Reduction XOR z5 = ^x1	**Reduction XNOR z6 = ^~x1**
x1 = 1 1 1 0 1 0 1 1	x1 = 1 1 1 0 1 0 1 1
z5 = 1 ^ 1 ^ 1 ^ 0 ^ 1 ^ 0 ^ 1 ^ 1 = 0	z6 = 1 ^~ 1 ^~ 1 ^~ 0 ^~ 1 ^~ 0 ^~ 1 ^~ 1 = 1

ตัวอย่างการใช้งานตัวดำเนินการลดถูกแสดงดังรูปที่ 3.21 รูป พร้อมด้วย test bench และผลรูปคลื่นในรูปที่ 3.22 และ 3.23 ตามลำดับ

```
//module to illustrate the use of reduction operators
module reduction (a, and_result, nand_result, or_result, nor_result, xor_result, xnor_result);

input [7:0] a;
output reg and_result, nand_result, or_result, nor_result, xor_result, xnor_result;

always @( a )
begin
        and_result = &a;    //reduction AND
        nand_result = ~&a;  //reduction NAND
        or_result = |a;     //reduction OR
        nor_result = ~|a;   //reduction NOR
        xor_result = ^a;    //reduction XOR
        xnor_result = ^~a;  //reduction XNOR
end
endmodule
```

รูปที่ 3.21 โค๊ด Verilog สำหรับแสดงตัวอย่างการใช้งานตัวดำเนินการลด

```
//test bench for reduction module
module reduction_tb_v;
reg [7:0] a;            // Inputs
// Outputs
wire and_result, nand_result, or_result,
nor_result, xor_result, xnor_result;

// Instantiate the Unit Under Test (UUT)
reduction uut (.a(a),
        .and_result(and_result),
        .nand_result(nand_result),
        .or_result(or_result),
        .nor_result(nor_result),
        .xor_result(xor_result),
        .xnor_result(xnor_result)
        );
initial
$monitor ("a=%b, and_result=%b,
nand_result=%b, or_result=%b, nor_result=%b,
xor_result=%b, xnor_result=%b", a, and_result,
nand_result, or_result, nor_result, xor_result,
xnor_result);
```

```
//apply input vectors
initial
begin
    #0      a = 8'b11000011;

  #10       a = 8'b10010011;

  #10       a = 8'b00001111;

  #10       a = 8'b01001111;

  #10       a = 8'b11001111;

  #10    $stop;
end

endmodule
```

รูปที่ 3.22 Test bench สำหรับโค๊ด Verilog ของรูปที่ 3.21

```
a=11000011, and_result=0, nand_result=1, or_result=1, nor_result=0, xor_result=0, xnor_result=1
a=10010011, and_result=0, nand_result=1, or_result=1, nor_result=0, xor_result=0, xnor_result=1
a=00001111, and_result=0, nand_result=1, or_result=1, nor_result=0, xor_result=0, xnor_result=1
a=01001111, and_result=0, nand_result=1, or_result=1, nor_result=0, xor_result=1, xnor_result=0
a=11001111, and_result=0, nand_result=1, or_result=1, nor_result=0, xor_result=0, xnor_result=1
```

รูปที่ 3.23 ผลการจำลองการทำงานของวงจรที่สังเคราะห์ได้จากโค๊ด Verilog ของรูปที่ 3.21

3.2.7 ตัวดำเนินการเลื่อน

ตัวดำเนินการเลื่อนทำการเลื่อนเวกเตอร์เดี่ยวไปทางซ้ายหรือทางขวาตามจำนวนบิตที่กำหนด เมื่อมีการเลื่อนจะมีบิตว่างเกิดขึ้น ศูนย์จะถูกเติมลงไปในบิตว่างเหล่านี้ ข้อมูลในบิตที่ถูกเลื่อนไปจะหายไป ไม่มีการวนกลับมาใหม่ ถ้าจำนวนการเลื่อนเป็น **x** หรือ **z** ผลลัพธ์ของการเลื่อนจะเป็น **x**

การเลื่อนซ้ายเวกเตอร์ไปหนึ่งบิตเทียบเท่ากับการคูณด้วย 2 หนึ่งครั้ง การเลื่อนขวาเวกเตอร์ไปหนึ่งบิตเทียบเท่ากับการหารด้วย 2 หนึ่งครั้ง ตัวดำเนินการเลื่อนมีประโยชน์ในการโมเดลระเบียบวิธี sequential add-shift multiplication และ sequential shift-subtract division

รูปที่ 3.24 แสดงตัวอย่างการใช้งานตัวดำเนินการเลื่อนโดยการเขียนโมดูลเชิงพฤติกรรมซึ่งใช้การกำหนดค่าแบบ blocking พร้อมด้วย test bench และผลการจำลองแบบการทำงานในรูปที่ 3.25 และ 3.26 ตามลำดับ

```
//examples of shift operators
module shift(a_reg, b_reg, result_a, result_b);

input [7:0] a_reg, b_reg;
output reg [7:0] result_a, result_b;

always @(a_reg or b_reg)
begin
   result_a = a_reg << 3;        //multiply by 8
   result_b = b_reg >> 2;       //divide by 4
end
endmodule
```

รูปที่ 3.24 โค๊ด Verilog สำหรับแสดงตัวอย่างการใช้งานตัวดำเนินการเลื่อน

```
//shift test bench
module shift_tb_v;
reg [7:0] a_reg, b_reg;          // Inputs
wire [7:0] result_a, result_b;   // Outputs

// Instantiate the Unit Under Test (UUT)
        shift uut (
                .a_reg(a_reg),
                .b_reg(b_reg),
                .result_a(result_a),
                .result_b(result_b)
        );
initial
$monitor ("a_reg=%b, b_reg=%b, result_a=%b,
result_b=%b", a_reg, b_reg, result_a, result_b);

//apply input vectors
initial
begin
  #0    a_reg = 8'b00000010;        //2
        b_reg = 8'b00001000;        //8
  #10   a_reg = 8'b00000110;        //6
        b_reg = 8'b00011000;        //24
   #10  a_reg = 8'b00001111;        //15
        b_reg = 8'b00111000;        //56
   #10  a_reg = 8'b11100000;        //224
        b_reg = 8'b00000011;        //3
   #10 $stop;
end
endmodule
```

รูปที่ 3.25 Test bench สำหรับโค๊ด Verilog ของรูปที่ 3.24

```
a_reg=00000010, b_reg=00001000, result_a=00010000, result_b=00000010
a_reg=00000110, b_reg=00011000, result_a=00110000, result_b=00000110
a_reg=00001111, b_reg=00111000, result_a=01111000, result_b=00001110
a_reg=11100000, b_reg=00000011, result_a=00000000, result_b=00000000
```

รูปที่ 3.26 ผลการจำลองการทำงานของวงจรที่สังเคราะห์ได้จากโค๊ด Verilog ของรูปที่ 3.24

3.2.8 ตัวดำเนินการเงื่อนไข

ตัวดำเนินการเงื่อนไขประกอบด้วยตัวถูกดำเนินการ 3 ตัวคือ นิพจน์เงื่อนไข (conditional expression) นิพจน์จริง (true expression) และนิพจน์เท็จ (false expression) เมื่อเงื่อนไขเป็นจริง (ลอจิก 1) นิพจน์จริงจะถูกดำเนินการ และเมื่ออเงื่อนไขเป็นเท็จ (ลอจิก 0) นิพจน์เท็จจะถูกดำเนินการ ไวยากรณ์การใช้งานเป็นดังนี้

conditional_expression **?** true_expression **:** false_expression;

ตัวอย่างเช่น z1 = (x1 >= x2)? x3 : x4; ถ้า x1 มากกว่าหรือเท่ากับ x2 แล้ว z1 มีค่าเท่ากับ x3 แต่ถ้า x1 น้อยกว่า x2 แล้ว z1 มีค่าเท่ากับ x4

ถ้านิพจน์เงื่อนไขถูกคำนวณได้ **x** แล้วผลลัพธ์ที่ได้จะถูกคำนวณจากนิพจน์จริงและนิพจน์เท็จได้ดัง truth table ดังตารางที่ 3.4

ตารางที่ 3.4 Truth table สำหรับนิพจน์เงื่อนไขที่เป็น x

True_expression	False_expression	Result
0	0	0
0	1	**x**
0	**x**	**x**
1	0	**x**
1	1	1
1	**x**	**x**
x	0	**x**
x	1	**x**
x	**x**	**x**

เนื่องจากตัวดำเนินการเงื่อนไขเป็นการเลือกค่าหนึ่งจากสองค่าขึ้นกับเงื่อนไข ดังนั้นตัวดำเนินการเงื่อนไขจึงสามารถถูกใช้แทนคำสั่ง **if…else** ได้ และเหมาะสมสำหรับโมเดลมัลติเพล็กเซอร์ 2:1 ดังรูปที่ 3.27

ตัวดำเนินการเงื่อนไขสามารถซ้อนกันได้ กล่าวคือนิพจน์จริงและนิพจน์เท็จสามารถเป็นตัวดำเนินการเงื่อนไขซ้อนอยู่ภายในได้ ซึ่งมีประโยชน์สำหรับการโมเดลมัลติเพล็กเซอร์ 4:1 ดังรูปที่ 3.28 พร้อมด้วย test bench และผลการจำลองแบบการทำงานดังรูปที่ 3.29 และ 3.30 ตามลำดับ

```
//dataflow 2:1 mux using conditional operator
module mux2_1_cond(s0, x0, x1, z1);
input   s0, x0, x1;
output  z1;
assign    z1 = s0 ? x0 : x1;
endmodule
```

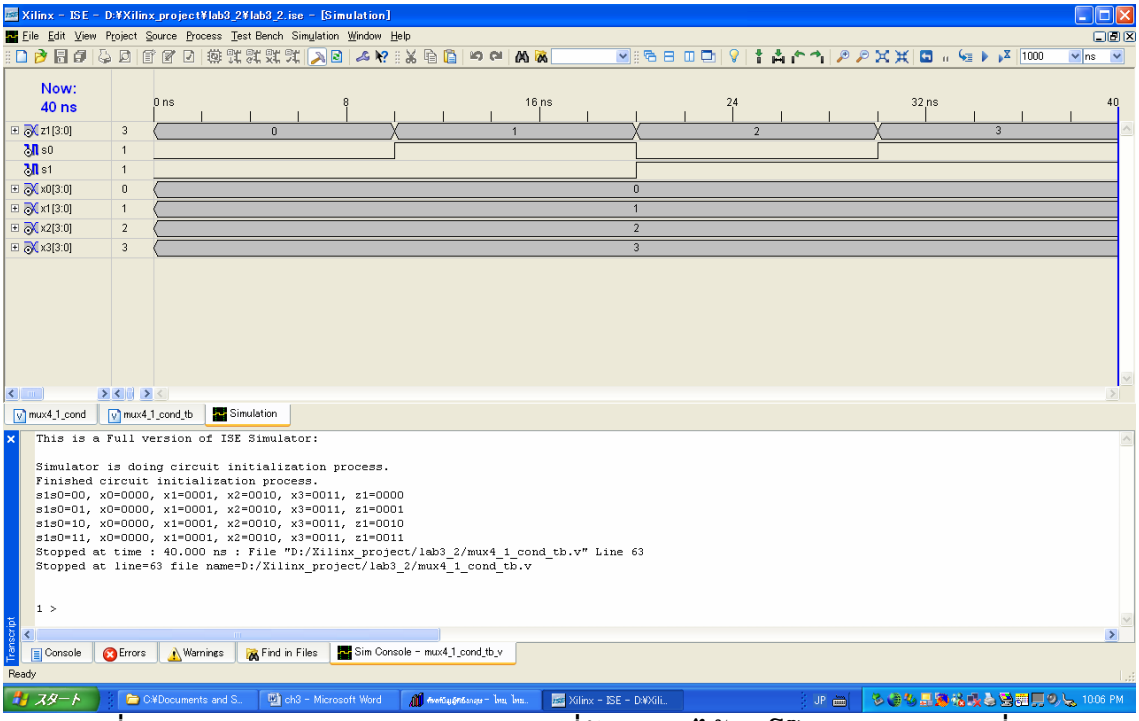
รูปที่ 3.27 โค๊ด Verilog สำหรับมัลติเพล็กเซอร์ 2:1 โดยใช้ตัวดำเนินการเงื่อนไข

```
//dataflow 4:1 mux using conditional operator
module mux4_1_cond (s0, s1, x0, x1, x2, x3, z1);
input    s0, s1;
input    [3:0] x0, x1, x2, x3;
output    [3:0] z1;
assign    z1 = s1 ? (s0 ? x3 : x2) : (s0 ? x1 : x0);

endmodule
```

รูปที่ 3.28 โค๊ด Verilog สำหรับมัลติเพล็กเซอร์ 4:1 โดยใช้ตัวดำเนินการเงื่อนไข

```
//4:1 multiplexer test bench
module mux4_1_cond_tb_v;
reg     s0, s1; // Inputs
reg     [3:0] x0, x1, x2, x3; // Inputs
wire    [3:0] z1;     // Outputs

// Instantiate the Unit Under Test (UUT)
mux4_1_cond uut (.s0(s0), .s1(s1), .x0(x0),
            .x1(x1), .x2(x2), .x3(x3), .z1(z1) );
initial
$monitor ("s1s0=%b%b, x0=%b, x1=%b, x2=%b,
x3=%b, z1=%b", s1, s0, x0, x1, x2, x3, z1);
//apply input vectors
initial
begin
        #0      s1 = 1'b0; s0 = 1'b0;
                x0 = 4'b0000;
                x1 = 4'b0001;
                x2 = 4'b0010;
                x3 = 4'b0011;
      #10       s1 = 1'b0; s0 = 1'b1;
      #10       s1 = 1'b1; s0 = 1'b0;
      #10       s1 = 1'b1; s0 = 1'b1;
      #10       $stop;
end
endmodule
```

รูปที่ 3.29 Test bench สำหรับโค๊ด Verilog ของรูปที่ 3.28



รูปที่ 3.30 ผลการจำลองการทำงานของวงจรที่สังเคราะห์ได้จากโค๊ด Verilog ของรูปที่ 3.28

3.2.9 ตัวดำเนินการต่อกัน

ตัวดำเนินการต่อกันสร้างเวกเตอร์ที่มีขนาดกว้างขึ้นจากตัวถูกดำเนินการ 2 ตัวหรือมากกว่า โดยการนำมาต่อกัน ขนาดของตัวถูกดำเนินการต้องถูกทราบก่อนการดำเนินการ ตัวอย่างการใช้งานตัวดำเนินการต่อกันถูกแสดงดังรูปที่ 3.31 พร้อมด้วย test bench และผลการจำลองการทำงานดังรูปที่ 3.32 และ 3.33 ตามลำดับ z1…z4 ถูกประกาศให้มีขนาดเท่ากับ 10 บิต เพื่อรองรับการต่อกันของ a b c และ d บิตบนของ z5 ถูกแทนด้วยบิตล่างของ a_bus[3:0] และบิตล่างของ z5 ถูกแทนด้วยบิตบนของ a_bus[7:4] ตัวดำเนินการต่อกันรองรับการต่อกันของเวกเตอร์ขนาดต่างกัน รวมทั้งการรวมเวกเตอร์โดยตรงเช่น 4'b0111 ใน z6 เป็นต้น

```
//examples of concatination
module concat(a, b, c, d, a_bus, z1, z2, z3, z4,
z5, z6);
input      [1:0] a;
input [2:0] b;
input      [3:0] c;
input      d;
input      [7:0] a_bus;

output   [9:0] z1, z2, z3, z4;
output   [7:0] z5;
output   [11:0] z6;
```

```
assign z1 = {a, c};
assign z2 = {b, a};
assign z3 = {c, b, a};
assign z4 = {a, b, c, d};

assign z5 = {a_bus[3:0], a_bus[7:4]};
assign z6 = {b, c, d, 4'b0111};

endmodule
```

รูปที่ 3.31 โค๊ด Verilog สำหรับแสดงตัวอย่างการใช้งานตัวดำเนินการต่อกัน

```
//concatenation test bench
module concat_tb_v;
// Inputs
reg   [1:0] a;
reg   [2:0] b;
reg   [3:0] c;
reg   d;
reg   [7:0] a_bus;
// Outputs
wire   [9:0] z1, z2, z3, z4;
wire   [7:0] z5;
wire   [11:0] z6;

// Instantiate the Unit Under Test (UUT)
concat uut ( .a(a), .b(b), .c(c), .d(d),
        .a_bus(a_bus), .z1(z1), .z2(z2),
        .z3(z3), .z4(z4), .z5(z5), .z6(z6)
            );
```

```
initial
$monitor ("a=%b, b=%b, c=%b, d=%b,
a_bus=%b, z1=%b, z2=%b, z3=%b, z4=%b,
z5=%b, z6=%b", a, b, c, d, a_bus, z1, z2, z3, z4,
z5, z6);

initial
begin
            #0          a = 2'b11;
                        b = 3'b001;
                        c = 4'b1100;
                        d = 1'b1;
                        a_bus = 8'b1111_0000;
            #10         $stop;
end
endmodule
```

รูปที่ 3.32 Test bench สำหรับโค๊ด Verilog ของรูปที่ 3.31

```
a=11, b=001, c=1100, d=1, a_bus=11110000,
z1=0000111100, z2=0000000111, z3=0110000111, z4=1100111001, z5=00001111, z6=001110010111
```

รูปที่ 3.33 ผลการจำลองการทำงานของวงจรที่สังเคราะห์ได้จากโค๊ด Verilog ของรูปที่ 3.31

3.2.10 ตัวดำเนินการทำซ้ำ

ตัวดำเนินการทำซ้ำเป็นการทำซ้ำการต่อเวกเตอร์เท่ากับจำนวนครั้งที่ถูกกำหนด ไวยากรณ์การใช้งานเป็นดังนี้

{number_of_repetations {expression_1, expression_2, …, expression_n}};

ตัวอย่างการใช้งานตัวดำเนินการทำซ้ำถูกแสดงดังรูปที่ 3.34 พร้อมด้วย test bench และผลการจำลองการทำงานดังรูปที่ 3.35 และ 3.36 ตามลำดับ

```
//example of replication
module replication (a, b, c, z1, z2);
input [1:0] a;
input [2:0] b;
input [3:0] c;

output [11:0] z1;
output [21:0] z2;
```

```
assign z1 = {2{a, c}};
assign z2 = {2{b, c, 4'b0111}};

endmodule
```

รูปที่ 3.34 โค๊ด Verilog สำหรับแสดงตัวอย่างการใช้งานตัวดำเนินการทำซ้ำ

```
//replication test bench
module replication_tb_v;

// Inputs
reg   [1:0] a;
reg   [2:0] b;
reg   [3:0] c;

// Outputs
wire   [11:0] z1;
wire   [21:0] z2;

// Instantiate the Unit Under Test (UUT)
replication uut (.a(a), .b(b), .c(c), .z1(z1), .z2(z2) );
```

```
initial
$monitor ("a=%b, b=%b, c=%b, z1=%b, z2=%b",
a, b, c, z1, z2);

initial
begin
        #0      a = 2'b11;
                b = 3'b010;
                c = 4'b0011;
        #10     $stop;
end

endmodule
```

รูปที่ 3.35 Test bench สำหรับโค๊ด Verilog ของรูปที่ 3.34

```
a=11, b=010, c=0011,
z1=110011110011, z2=010001101110100011011
```

รูปที่ 3.36 ผลการจำลองการทำงานของวงจรที่สังเคราะห์ได้จากโค๊ด Verilog ของรูปที่ 3.34

โจทย์

3.1 Design a five-function arithmetic unit for addition, subtraction, multiplication, division, and modulus. There will be three 4-bit operands: a, b, and c and one result **reg** variable, which must be sufficient width to contain the largest result. Design test bench and obtain the outputs for several values of the inputs. The operations are defined below.

```
Add operation       = a + b + c;
Subtract operation  = (a + b) – c;
Multiply operation  = a * b * c;
Divide operation    = (a + b) / 2;
Modulus operation   = (b + c) % 4;
```

3.2 Design a four-function logic unit for the three logical operations: AND, OR, and NOT. There will be three 4-bit operands: a, b, and c and one result **reg** variable. Generate the test bench and obtains the outputs for the following operations:

```
AND operation       = (a && b) && c;
OR operation        = (a || c) || b;
AND OR operation    = (b && c) || a;
NOT operation       = !((b && c) || a;
```

3.3 Design a five-function logic unit to show the operation of the five bit-wise operators: AND(&), OR(|), NOT(~), XOR(^), and XNOR(^~ or ~^). There will be three 4-bit operands: a, b, and c and one result **reg** variable. Generate the test bench and obtain the outputs for the following operations:

AND operation = (a & b) & c;

OR operation = (a | c) | b;

NOT operation = ~((b & c) | a);

XOR operation = (b ^ c) ^ a;

XNOR operation = (a ^~ c) ^~ b;

3.4 Design an odd parity generator using the XOR and XNOR operators. There are four data inputs and one output that is logic 1 when the number of 1s in the input vector is even. Use dataflow modeling and test all combinations of the inputs. Generate a test bench and obtain the outputs.

บทที่ 4
การโมเดลวงจรระดับเกท

เนื้อหาในบทนี้อธิบายรายละเอียดของการโมเดลวงจรระดับเกท (gate-level modeling) ซึ่งเป็นการออกแบบที่ระดับต่ำ (low-level synthesis) วงจรถูกสร้างจากการต่อกันของเกทปฐมฐาน (primitive) ในที่นี้หมายถึงเกทปฐมฐานที่มีอยู่แล้วในภาษา Verilog เท่านั้น เกทปฐมฐานดังกล่าวถูกเรียกว่า built-in primitive นอกจากนี้ภาษา Verilog ยังรองรับการสร้างเกทปฐมฐานที่ผู้ใช้สร้างขึ้น (user defined primitive; UDP) ทั้งนี้ซอฟท์แวร์ช่วยออกแบบอาจจะไม่รองรับการสังเคราะห์ UDP ก็ได้ แต่อย่างไรก็ตามซอฟท์แวร์ช่วยออกแบบอาจจะมี built-in primitive หลากหลายชนิดแตกต่างกันไปเพื่อรองรับการโมเดลวงจรระดับเกท

1.18 เกทปฐมฐานภายในภาษา Verilog (Built-in Primitives)

4.1.1 เกทหลายอินพุท (Multiple-input gate)

เกทหลายอินพุทเช่น **and nand or nor xor** และ **xnor** เป็น built-in primitive ที่ถูกนิยามไว้แล้วในภาษา Verilog ในการอธิบายวงจรสามารถเรียกใช้โดยคำหลักเหล่านี้ได้เลย ไม่จำเป็นต้องนิยามขึ้นมาใหม่ แต่ละเกทอาจมีหลายอินพุท แต่มีเพียงเอาท์พุทเดียว ไวยากรณ์การใช้งานเป็นดังนี้

gate_type inst1 (output, input_1, input_2, …, input_n); หรือ

gate_type (output, input_1, input_2, …, input_n);

โดยรายการในวงเล็บต้องแสดงเอาท์พุทเป็นลำดับแรก จากนั้นจึงจะตามด้วยอินพุท inst1 คือชื่อโมดูลย่อย อาจจะแสดงหรือไม่ก็ได้ ในกรณีที่ใช้เกทชนิดเดียวกันหลายครั้งก็สามารถเรียกใช้ได้ดังนี้

```
gate_type (output_1, input_11, input_12, …, input_1n),
          (output_2, input_21, input_22, …, input_2n),
                …
          (output_m, input_m1, input_m2, …, input_mn);
```

โดยแต่ละครั้งถูกคั่นด้วยสัญลักษณ์ , และจบด้วยสัญลักษณ์ ;

ตัวอย่างที่ 4.1 ออกแบบวงจรเกท AND 3-อินพุท และ OR 3-อินพุท โดยใช้ built-in primitives รูปที่ 4.1 แสดงโมดูล Verilog แบบใช้ built-in primitives พร้อมด้วย test bench และผลการจำลองการทำงานในรูปที่ 4.2 และ 4.3 ตามลำดับ

```
//gate-level modeling for and/or gate
module and3_or3(x1, x2, x3, and3_out, or3_out);
input    x1, x2, x3;
output   and3_out, or3_out;

and (and3_out, x1, x2, x3);
or (or3_out, x1, x2, x3);

endmodule
```

รูปที่ 4.1 โมดูล Verilog แบบใช้ built-in primitives สำหรับวงจรเกท AND 3-อินพุท และ OR 3-อินพุท

```
//test bench for and3_or3 module
module    and3_or3_tb_v;
reg       x1, x2, x3;
wire      and3_out, or3_out;

// Instantiate the Unit Under Test (UUT)
and3_or3 uut ( .x1(x1), .x2(x2), .x3(x3),
            .and3_out(and3_out), .or3_out(or3_out)
            );
//monitor variables
initial
$monitor ("x1x2x3 = %b, and3_out = %b,
or3_out = %b", {x1, x2, x3}, and3_out, or3_out);
```

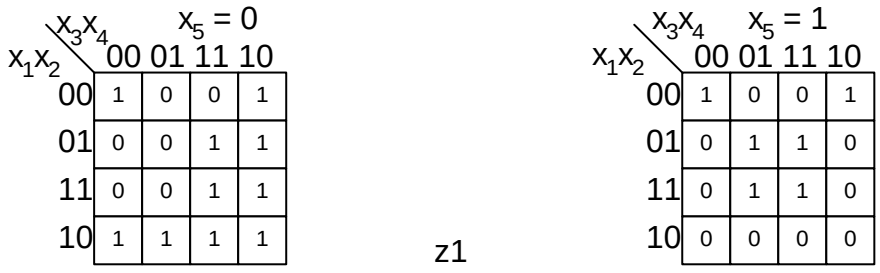
```
//apply input vectors
initial
begin
        #0      {x1, x2, x3} = 3'b000;
        #10     {x1, x2, x3} = 3'b001;
        #10     {x1, x2, x3} = 3'b010;
        #10     {x1, x2, x3} = 3'b011;
        #10     {x1, x2, x3} = 3'b100;
        #10     {x1, x2, x3} = 3'b101;
        #10     {x1, x2, x3} = 3'b110;
        #10     {x1, x2, x3} = 3'b111;
        #10     $stop;
end
endmodule
```

รูปที่ 4.2 Test bench สำหรับวงจรเกท AND 3-อินพุท และ OR 3-อินพุทของรูปที่ 4.1

```
x1x2x3 = 000, and3_out = 0, or3_out = 0
x1x2x3 = 001, and3_out = 0, or3_out = 1
x1x2x3 = 010, and3_out = 0, or3_out = 1
x1x2x3 = 011, and3_out = 0, or3_out = 1
x1x2x3 = 100, and3_out = 0, or3_out = 1
x1x2x3 = 101, and3_out = 0, or3_out = 1
x1x2x3 = 110, and3_out = 0, or3_out = 1
x1x2x3 = 111, and3_out = 1, or3_out = 1
```

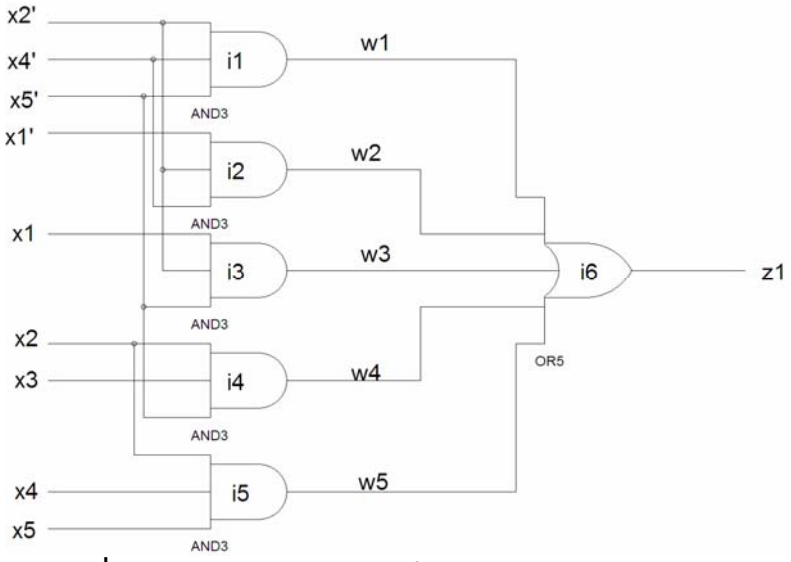
รูปที่ 4.3 ผลการจำลองการทำงานสำหรับ Test bench ของรูปที่ 4.2

ตัวอย่างที่ 4.2 แผนภาพคาร์นอฟแสดงดังรูปที่ 4.4 ถูกใช้เพื่อให้ได้มาซึ่งนิพจน์ที่เล็กที่สุดสำหรับเอาท์พุท z1 ในรูปแบบของ sum-of-product ดังสมการที่ 4.1 รูปที่ 4.5 แสดงวงจรลอจิกที่ระบุชื่อโมดูลย่อย และเน็ท วงจรถูกอธิบายด้วย built-in primitive ดังรูปที่ 4.6 พร้อมด้วย test bench และผลการจำลองการทำงานในรูปที่ 4.7 และ 4.8 ตามลำดับ



รูปที่ 4.4 แผนภาพคาร์นอฟสำหรับการหานิพจน์ที่เล็กที่สุดสำหรับเอาท์พุท z1 แบบ sum-of-product

$$z1 = x_2'x_4'x_5' + x_1'x_2'x_4' + x_1x_2'x_5' + x_2x_3x_5' + x_2x_4x_5 \qquad (4.1)$$



รูปที่ 4.5 วงจรลอจิกสำหรับเอาท์พุท z1 แบบ sum-of-product

```
//logic diagram using built-in primitives
module log_eqn_sop7(x1, x2, x3, x4, x5, z1);
input   x1, x2, x3, x4, x5;
output   z1;

and        i1 (w1, ~x2, ~x4, ~x5),
           i2 (w2, ~x1, ~x2, ~x4),
           i3 (w3, x1, ~x2, ~x5),
           i4 (w4, x2, x3, ~x5),
           i5 (w5, x2, x4, x5);
or         i6 (z1, w1, w2, w3, w4, w5);

endmodule
```

รูปที่ 4.6 โมดูล Verilog แบบใช้ built-in primitives สำหรับวงจรรูปที่ 4.5

```
//test bench for log_eqn_sop7
module log_eqn_sop7_tb_v;
reg   x1, x2, x3, x4, x5;
wire   z1;

// Instantiate the Unit Under Test (UUT)
          log_eqn_sop7 uut (
                    .x1(x1),
                    .x2(x2),
                    .x3(x3),
                    .x4(x4),
                    .x5(x5),
                    .z1(z1)
          );
```

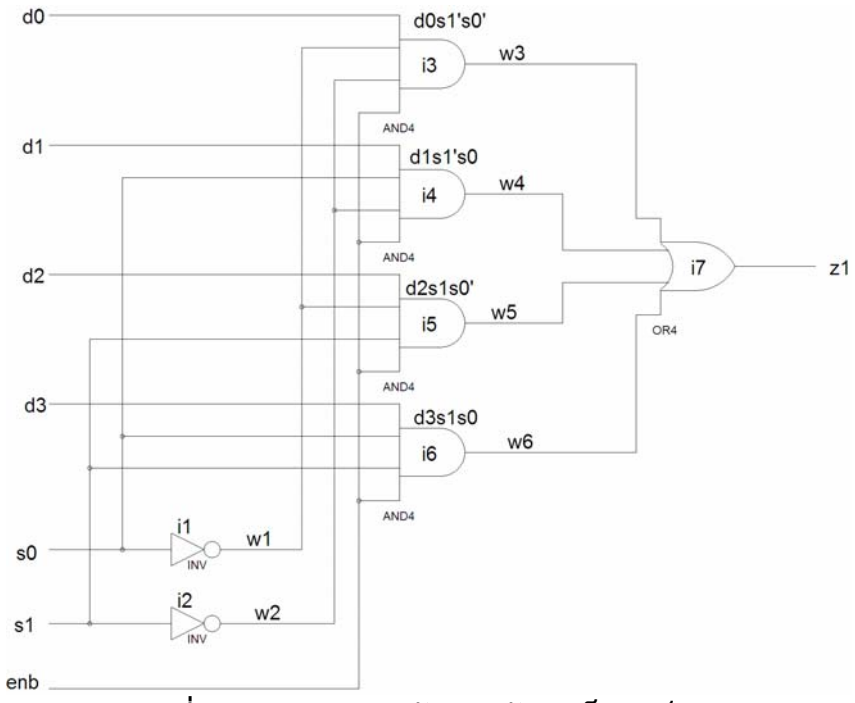
```
//apply input vectors
initial
begin : apply_stimulus
    reg [6:0] invect;
    for (invect = 0; invect < 32; invect = invect+1)
        begin
            {x1, x2, x3, x4, x5} = invect[6:0];
            #10  $display ("x1x2x3x4x5  =  %b,  z1
= %b", {x1, x2, x3, x4, x5}, z1);
        end
end

endmodule
```

รูปที่ 4.7 Test bench สำหรับสำหรับวงจรรูปที่ 4.5

x1x2x3x4x5 = 00000, z1 = 1	x1x2x3x4x5 = 10000, z1 = 1
x1x2x3x4x5 = 00001, z1 = 1	x1x2x3x4x5 = 10001, z1 = 0
x1x2x3x4x5 = 00010, z1 = 0	x1x2x3x4x5 = 10010, z1 = 1
x1x2x3x4x5 = 00011, z1 = 0	x1x2x3x4x5 = 10011, z1 = 0
x1x2x3x4x5 = 00100, z1 = 1	x1x2x3x4x5 = 10100, z1 = 1
x1x2x3x4x5 = 00101, z1 = 1	x1x2x3x4x5 = 10101, z1 = 0
x1x2x3x4x5 = 00110, z1 = 0	x1x2x3x4x5 = 10110, z1 = 1
x1x2x3x4x5 = 00111, z1 = 0	x1x2x3x4x5 = 10111, z1 = 0
x1x2x3x4x5 = 01000, z1 = 0	x1x2x3x4x5 = 11000, z1 = 0
x1x2x3x4x5 = 01001, z1 = 0	x1x2x3x4x5 = 11001, z1 = 0
x1x2x3x4x5 = 01010, z1 = 0	x1x2x3x4x5 = 11010, z1 = 0
x1x2x3x4x5 = 01011, z1 = 1	x1x2x3x4x5 = 11011, z1 = 1
x1x2x3x4x5 = 01100, z1 = 1	x1x2x3x4x5 = 11100, z1 = 1
x1x2x3x4x5 = 01101, z1 = 0	x1x2x3x4x5 = 11101, z1 = 0
x1x2x3x4x5 = 01110, z1 = 1	x1x2x3x4x5 = 11110, z1 = 1
x1x2x3x4x5 = 01111, z1 = 1	x1x2x3x4x5 = 11111, z1 = 1

รูปที่ 4.8 ผลการจำลองการทำงานสำหรับ Test bench ของรูปที่ 4.7

ตัวอย่างที่ 4.3 ออกแบบวงจรมัลติเพล็กเซอร์ 4:1 ดังรูปที่ 4.9 โดยใช้ built-in primitives รูปที่ 4.10 แสดงโมดูล Verilog แบบใช้ built-in primitives พร้อมด้วย test bench และผลการจำลองการทำงานในรูปที่ 4.11 และ 4.12 ตามลำดับ ใน test bench มีการใช้ซิสเต็มฟังก์ชัน **$time** เพื่อนำค่าเวลาในการจำลองการทำงานที่เวลาปัจจุบันมาแสดง



รูปที่ 4.9 วงจรลอจิกสำหรับวงจรมัลติเพล็กเซอร์ 4:1

```
//a 4:1 multiplexer using built-in primitives
module mux4_primitive(d, s, enb, z1);
input   [3:0] d;
input   [1:0] s;
input   enb;
output   z1;

not        i1 (w1, s[0]),
           i2 (w2, s[1]);
and        i3 (w3, d[0], w1, w2, enb),
           i4 (w4, d[1], s[0], w2, enb),
           i5 (w5, d[2], w1, s[1], enb),
           i6 (w6, d[3], s[0], s[1], enb);
or         i7 (z1, w3, w4, w5, w6);
endmodule
```

รูปที่ 4.10 โมดูล Verilog แบบใช้ built-in primitives สำหรับวงจรมัลติเพล็กเซอร์ 4:1 ในรูปที่ 4.9

```
//test bench for 4:1 multiplexer
module mux4_primitive_tb_v;
reg   [3:0] d;
reg   [1:0] s;
reg   enb;
wire   z1;

// Instantiate the Unit Under Test (UUT)
mux4_primitive uut (.d(d), .s(s), .enb(enb), .z1(z1)
          );

initial
$monitor ($time, "ns, select:s = %b, inputs:d
= %b, output:z1 = %b", s, d, z1);
```

```
//apply input vectors
initial
begin
#0        s = 2'b00; d = 4'b1010; enb = 1'b1;
#10       s = 2'b00; d = 4'b1011; enb = 1'b1;
#10       s = 2'b01; d = 4'b1011; enb = 1'b1;
#10       s = 2'b10; d = 4'b1011; enb = 1'b1;
#10       s = 2'b01; d = 4'b1011; enb = 1'b1;
#10       s = 2'b11; d = 4'b1011; enb = 1'b1;
#10       s = 2'b11; d = 4'b0011; enb = 1'b1;
#10       $stop;

end
endmodule
```

รูปที่ 4.11 Test bench สำหรับโมดูลวงจรมัลติเพล็กเซอร์ 4:1 ในรูปที่ 4.10

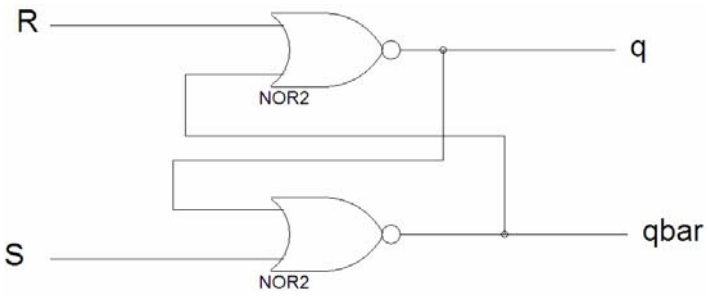
```
0ns, select:s = 00, inputs:d = 1010, output:z1 = 0
10ns, select:s = 00, inputs:d = 1011, output:z1 = 1
20ns, select:s = 01, inputs:d = 1011, output:z1 = 1
30ns, select:s = 10, inputs:d = 1011, output:z1 = 0
40ns, select:s = 01, inputs:d = 1011, output:z1 = 1
50ns, select:s = 11, inputs:d = 1011, output:z1 = 1
60ns, select:s = 11, inputs:d = 0011, output:z1 = 0
```

รูปที่ 4.12 ผลการจำลองการทำงานสำหรับ Test bench ของรูปที่ 4.11

ตัวอย่างที่ 4.4 ออกแบบวงจร SR latch โดยใช้ built-in primitives ตารางที่ 4.1 แสดง truth table ของ SR latch และรูปที่ 4.13 แสดงวงจรที่สร้างจากเกท NOR โมดูล Verilog แบบใช้ built-in primitives แสดงดังรูปที่ 4.14 พร้อมด้วย test bench และผลการจำลองการทำงานในรูปที่ 4.15 และ 4.16 ตามลำดับ

ตารางที่ 4.1 Truth table ของ SR latch

Inputs		Outputs		Comments
S	R	q	qbar	
0	0	q	qbar	
0	1	0	1	Reset
1	0	1	0	Set
1	1	0	0	Invalid state



รูปที่ 4.13 วงจรลอจิกสำหรับวงจร SR latch

```
//SR latch using built-in primitives
module SR_latch(S, R, q, qbar);
input    S, R;
output    q, qbar;

nor        (q, qbar, R),
           (qbar, q, S);
endmodule
```

รูปที่ 4.14 โมดูล Verilog แบบใช้ built-in primitives สำหรับวงจรวงจร SR latch

```
//test bench for SR_latch
module SR_latch_tb_v;
reg    S, R;
wire    q, qbar;
// Instantiate the Unit Under Test (UUT)
SR_latch uut (.S(S), .R(R), .q(q), .qbar(qbar)
           );
initial
$monitor ("S = %b, R = %b, q = %b, qbar = %b",
S, R, q, qbar);
```

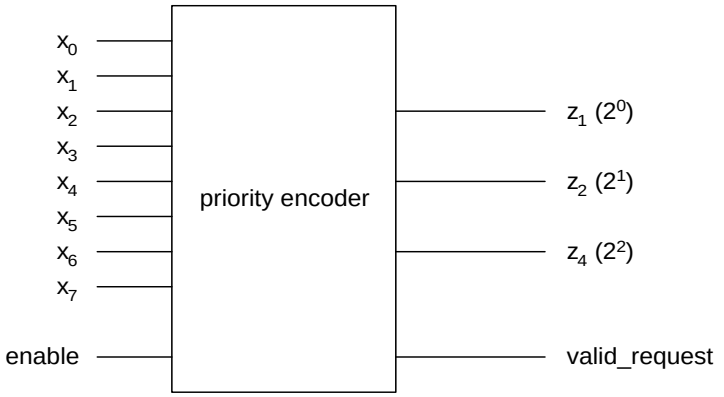
```
//apply stimulus
initial
begin
#0        S = 1'b1; R = 1'b0;
#10       S = 1'b0; R = 1'b0;
#10       S = 1'b0; R = 1'b1;
#10       S = 1'b1; R = 1'b1;
#10       $stop;
end
endmodule
```

รูปที่ 4.15 Test bench สำหรับวงจร SR latchของรูปที่ 4.13

```
S = 1, R = 0, q = 1, qbar = 0
S = 0, R = 0, q = 1, qbar = 0
S = 0, R = 1, q = 0, qbar = 1
S = 1, R = 1, q = 0, qbar = 0
```

รูปที่ 4.16 ผลการจำลองการทำงานสำหรับ Test bench ของรูปที่ 4.15

ตัวอย่างที่ 4.5 ออกแบบวงจร Priority encoder โดยใช้ built-in primitives วงจร Priority encoder มีความแตกต่างจากวงจรเข้ารหัสโดยทั่วไป คือสัญญาณอินพุทสามารถถูกป้อนเข้าได้มากกว่าหนึ่งอินพุทพร้อมๆกัน มีประโยชน์ในงานประยุกต์ที่มีการใช้ทรัพยากรร่วมกันของอุปกรณ์ในระบบ อุปกรณ์ที่มีลำดับความสำคัญสูงกว่าก็จะได้รับการบริการก่อน กล่าวคือถ้าอินพุท x_i และ x_j โดยที่จำนวนเต็ม $i > j$ ร้องขอบริการมาพร้อมกัน อินพุท x_i จะได้รับบริการก่อน x_j โดยทั่วไปวงจรที่มี n อินพุท จะมี $\log_2 n$ เอาท์พุท ตัวอย่างเช่นวงจรมี 8 อินพุท จะมี 3 เอาท์พุท ดังบล็อกไดอะแกรมรูปที่ 4.17 ซึ่งมีอินพุท enable เพื่อควบคุมให้วงจรทำงานหรือหยุดทำงาน และมีเอาท์พุท valid_request เพื่อบอกว่ามีสัญญาณร้องขอเข้ามา ตารางที่ 4.2 แสดง truth table ของวงจร Priority encoder ขนาด 8-อินพุท 3-เอาท์พุท สมการที่ 4.2 แสดงลอจิกฟังก์ชันที่ได้มาจาก truth table และรูปที่ 4.18 วงจรลอจิก โมดูล Verilog แบบใช้ built-in primitives สำหรับวงจรดังกล่าวแสดงดังรูปที่ 4.19 พร้อมด้วย test bench และผลการจำลองการทำงานในรูปที่ 4.20 และ 4.21 ตามลำดับ



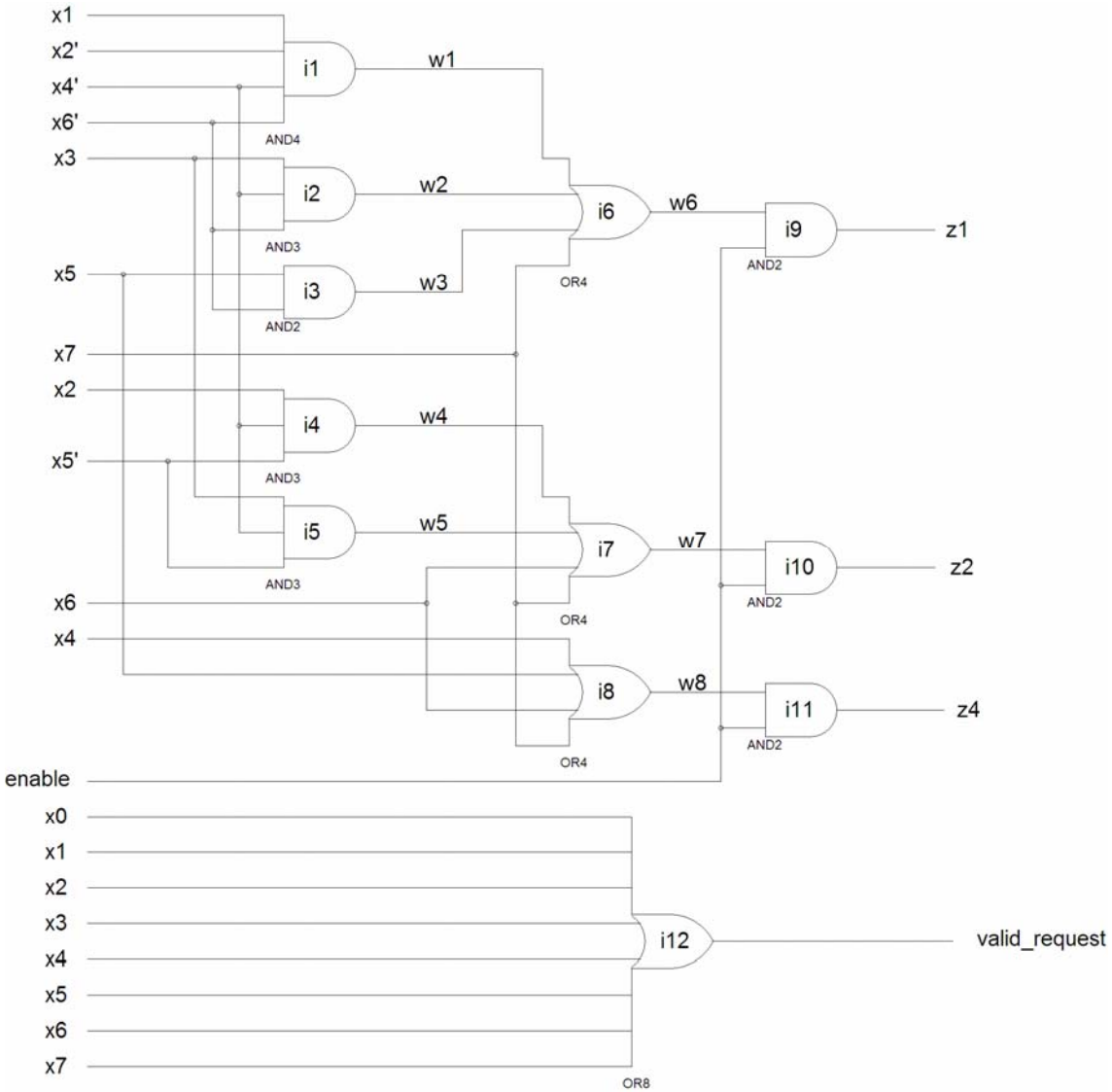
รูปที่ 4.17 บล็อกไดอะแกรมของวงจร Priority encoder ขนาด 8-อินพุท 3-เอาท์พุท

ตารางที่ 4.2 Truth table ของวงจร Priority encoder ขนาด 8-อินพุท 3-เอาท์พุท

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	z_1 (2^0)	z_2(2^1)	z_4(2^2)
1	0	0	0	0	0	0	0	0	0	0
d	1	0	0	0	0	0	0	0	0	1
d	d	1	0	0	0	0	0	0	1	0
d	d	d	1	0	0	0	0	0	1	1
d	d	d	d	1	0	0	0	1	0	0
d	d	d	d	d	1	0	0	1	0	1
d	d	d	d	d	d	1	0	1	1	0
d	d	d	d	d	d	d	1	1	1	1

หมายเหตุ d: don't care

$$
\begin{aligned}
z_1(2^0) &= x_1x_2'x_3'x_4'x_5'x_6'x_7' + x_3x_4'x_5'x_6'x_7' + x_5x_6'x_7' + x_7 \\
&= x_1x_2'x_4'x_6' + x_3x_4'x_6' + x_5x_6' + x_7 \\
Z_2(2^1) &= x_2x_3'x_4'x_5'x_6'x_7' + x_3x_4'x_5'x_6'x_7' + x_6x_7' + x_7 \\
&= x_2x_4'x_5' + x_3x_4'x_5' + x_6 + x_7 \\
Z_4(2^2) &= x_4x_5'x_6'x_7' + x_5x_6'x_7' + x_6x_7' + x_7 \\
&= x_4 + x_5 + x_6 + x_7 \qquad (4.2)
\end{aligned}
$$



รูปที่ 4.18 วงจรลอจิกสำหรับวงจร Priority encoder ขนาด 8-อินพุท 3-เอาท์พุทของรูปที่ 4.17

```
//8-bit priority encoder
module priority_8(x0, x1, x2, x3, x4, x5, x6, x7, enbable, z1, z2, z4, valid_request);
input      x0, x1, x2, x3, x4, x5, x6, x7, enbable;
output     z1, z2, z4, valid_request;
   and       i1 (w1, x1, ~x2, ~x4, ~x6),
             i2 (w2, x3, ~x4, ~x6),
             i3 (w3, x5, ~x6),
             i4 (w4, x2, ~x4, ~x5),
             i5 (w5, x3, ~x4, ~x5);
   or        i6 (w6, w1, w2, w3, x7),
             i7 (w7, w4, w5, x6, x7),
             i8 (w8, x4, x5, x6, x7);
   and       i9 (z1, w6, enable),
            i10 (z2, w7, enable),
            i11 (z4, w8, enable);
   or        i12 (valid_request, x0, x1, x2, x3, x4, x5, x6, x7);
endmodule
```

รูปที่ 4.19 โมดูล Verilog แบบใช้ built-in primitives สำหรับวงจร Priority encoder ขนาด 8-อินพุท 3-เอาท์พุทของรูปที่ 4.17

```
//test bench for 8-bit priority encoder
module priority_8_tb_v;
reg        x0, x1, x2, x3, x4, x5, x6, x7, enable;
wire       z1, z2, z4, valid_request;

// Instantiate the Unit Under Test (UUT)
priority_8 uut (.x0(x0), .x1(x1), .x2(x2), .x3(x3),
            .x4(x4), .x5(x5), .x6(x6), .x7(x7),
            .enable(enable),
            .z1(z1), .z2(z2), .z4(z4),
            .valid_request(valid_request)
            );

initial
$monitor ("x01x2x3x4x5x6x7 = %b, z4z2z1
= %b, valid_request = %b", {x0, x1, x2, x3, x4,
x5, x6, x7}, {z4, z2, z1}, valid_request);
```

```
//apply input vectors
initial
begin
#0 {x0, x1, x2, x3, x4, x5, x6, x7}= 8'b00000000;
enable = 1'b1;
#10 {x0, x1, x2, x3, x4, x5, x6, x7}= 8'b00100000;
enable = 1'b1;
#10 {x0, x1, x2, x3, x4, x5, x6, x7}= 8'b00000100;
enable = 1'b1;
#10 {x0, x1, x2, x3, x4, x5, x6, x7}= 8'b00100010;
enable = 1'b1;
#10 {x0, x1, x2, x3, x4, x5, x6, x7}= 8'b10110000;
enable = 1'b1;
#10 {x0, x1, x2, x3, x4, x5, x6, x7}= 8'b11111111;
enable = 1'b1;
#10        $stop;
end
endmodule
```

รูปที่ 4.20 Test bench สำหรับวงจร Priority encoder ขนาด 8-อินพุท 3-เอาท์พุทของรูปที่ 4.17

```
x01x2x3x4x5x6x7 = 00000000, z4z2z1 = 000, valid_request = 0
x01x2x3x4x5x6x7 = 00100000, z4z2z1 = 010, valid_request = 1
x01x2x3x4x5x6x7 = 00000100, z4z2z1 = 101, valid_request = 1
x01x2x3x4x5x6x7 = 00100010, z4z2z1 = 110, valid_request = 1
x01x2x3x4x5x6x7 = 10110000, z4z2z1 = 011, valid_request = 1
x01x2x3x4x5x6x7 = 11111111, z4z2z1 = 111, valid_request = 1
```

รูปที่ 4.21 ผลการจำลองการทำงานสำหรับ Test bench ของรูปที่ 4.20

บทที่ 5
การโมเดลกระแสข้อมูล

เนื้อหาในบทนี้อธิบายรายละเอียดของการโมเดลกระแสข้อมูล (Dataflow Modeling) ซึ่งเป็นวิธีการบรรยายการออกแบบที่ระดับสูงกว่าการโมเดลระดับเกท ซอฟท์แวร์ช่วยออกแบบอัตโนมัติ (Design automation tools) ถูกนำใช้เพื่อสร้างวงจรลอจิกเกทด้วยวิธีการ การสังเคราะห์วงจรลอจิก (logic synthesis)

1.19 การกำหนดค่าแบบต่อเนื่อง (Continuous assignment)

คำสั่งแบบการกำหนดค่าแบบต่อเนื่องสามารถโมเดลพฤติกรรมของกระแสข้อมูลได้ และถูกใช้ในการออกแบบวงจรเชิงจัดหมู่ (combinational circuit) โดยไม่ต้องใช้เกทและเน็ทในการเชื่อมต่อ คำสั่งแบบการกำหนดค่าแบบต่อเนื่องเป็นความสัมนัยระหว่างนิพจน์ทางด้านขวาและเป้าหมายทางด้านซ้าย ไวยากรณ์การใช้งานเป็นดังนี้ โดยการกำหนดเวลาเป็นทางเลือก ไม่จำเป็นต้องระบุก็ได้ ไม่ถูกนำมาพิจารณาในการสังเคราะห์วงจรลอจิก แต่ใช้ในการจำลองแบบการทำงานของวงจร

assign [#time] left-hand side target = right-hand side expression

คำสั่งแบบการกำหนดค่าแบบต่อเนื่องกำหนดค่าไปยังเน็ท (**wire**) ที่ถูกประกาศไว้ ไม่สามารถกำหนดค่าไปยังรีจิสเตอร์ ดังนั้นเป้าหมายทางด้านซ้ายต้องเป็นเน็ทเท่านั้น อาจจะเป็นเน็ทแบบสเกลาร์ เวกเตอร์ หรือเวกเตอร์ที่ต่อกัน ตัวถูกดำเนินการในนิพจน์ทางด้านขวาสามารถเป็นรีจิสเตอร์ เน็ท หรือการเรียกฟังก์ชัน

ตัวอย่างของคำสั่งแบบการกำหนดค่าแบบต่อเนื่องสำหรับเน็ทที่เป็นสเกลาร์เป็นดังนี้

```
assign    z1 = x1 & x2 & x3;
assign    z2 = x1 ^ x2;
assign    z3 = (x1 & x2) | x3;
```

ตัวอย่างของคำสั่งแบบการกำหนดค่าแบบต่อเนื่องสำหรับเน็ทที่มีทั้งแบบเวกเตอร์และสเกลาร์เป็นดังนี้ sum เป็นเว็กเตอร์ขนาด 9 บิตที่รองรับผลบวกและตัวทดของการบวก a และ b ซึ่งเป็นเว็กเตอร์ขนาด 8 บิต และ cin เป็นสเกลาร์

```
assign    sum = a + b + cin;
```

ตัวอย่างของคำสั่งแบบการกำหนดค่าแบบต่อเนื่องสำหรับเน็ทที่มีทั้งแบบเวกเตอร์และเวกเตอร์ที่ต่อกันเป็นดังนี้ sum a และ b เป็นเว็กเตอร์ขนาด 8 บิต cout และ cin เป็นสเกลาร์ ผลบวกของ a และ b มีขนาด 9 บิตซึ่งถูกกำหนดค่าให้เวกเตอร์ที่ต่อกันของ cout และ sum ซึ่งมี 9 บิตโดย cout เป็นบิทบนสุด

```
assign    {cout, sum} = a + b + cin;
```

คำสั่งแบบการกำหนดค่าแบบต่อเนื่องนี้จะเฝ้าสังเกตอินพุทอย่างต่อเนื่อง เมื่อมีการเปลี่ยนแปลงของตัวแปรใดตัวแปรหนึ่งในนิพจน์ทางด้านขวามือ นิพจน์ก็จะถูกคำนวณและนำค่าผลลัพธ์กำหนดให้กับเป้าหมายหลังจากระยะเวลาผ่านไปตามที่กำหนด ถ้าไม่กำหนดถือว่าเป็นศูนย์

5.1.1 เกท AND 3-อินพุท

หัวข้อนี้แสดงตัวอย่างการโมเดลเกท AND ขนาด 3-อินพุทด้วยโมเดลกระแสข้อมูล รูปที่ 5.1 แสดงโมดูล Verilog สำหรับเกท AND ดังกล่าว พร้อมด้วย test bench และผลการจำลองการทำงานในรูปที่ 5.2 และรูปที่ 5.3 ตามลำดับ

```
//and3 dataflow
module and3_df(x1, x2, x3, z1);
input    x1, x2, x3;
output   z1;
wire x1, x2, x3, z1; //define signals as wire for dataflow
//continuous assignment for dataflow
assign z1 = x1 & x2 & x3;
endmodule
```

รูปที่ 5.1 โมดูล Verilog แบบกระแสข้อมูลสำหรับเกท AND ขนาด 3-อินพุท

```
//test bench for the 3-input AND gate
module and3_df_tb_v;
reg x1, x2, x3; // Inputs are reg for test bench
wire z1; // Outputs are wire for test bench

// Instantiate the Unit Under Test (UUT)
        and3_df uut (
                .x1(x1),
                .x2(x2),
                .x3(x3),
                .z1(z1)
        );
```

```
//apply input vector and display variables
initial
begin: apply_stimulus
    reg [3:0] invect;
    for (invect = 0; invect < 8; invect = invect + 1)
        begin
            {x1, x2, x3} = invect [3:0];
            #10 $display ("x1 x2 x3 = %b, z1
= %b", {x1, x2, x3}, z1);
        end
end
endmodule
```

รูปที่ 5.2 Test bench สำหรับเกท AND ขนาด 3-อินพุทแบบกระแสข้อมูลของรูปที่ 5.1

```
x1 x2 x3 = 000, z1 = 0
x1 x2 x3 = 001, z1 = 0
x1 x2 x3 = 010, z1 = 0
x1 x2 x3 = 011, z1 = 0
x1 x2 x3 = 100, z1 = 0
x1 x2 x3 = 101, z1 = 0
x1 x2 x3 = 110, z1 = 0
x1 x2 x3 = 111, z1 = 1
```

รูปที่ 5.3 ผลการจำลองการทำงานสำหรับ Test bench ของรูปที่ 5.2

5.1.2 ตัวดำเนินการลด

หัวข้อนี้แสดงตัวอย่างการโมเดลวงจรตัวดำเนินการลดด้วยโมเดลกระแสข้อมูล รูปที่ 5.4 แสดงโมดูล Verilog สำหรับวงจรตัวดำเนินการลด AND NAND OR NOR XOR และ XNOR สังเกตุเห็นว่าการกำหนดค่าสามารถใช้คำหลัก **assign** เพียงครั้งเดียวได้หากว่าไม่มีการกำหนดค่าดีเลย์ในการกำหนดค่าแต่ละคำสั่ง โดยตามด้วยสัญลักษณ์ , ในแต่ละคำสั่ง และจบด้วยสัญลักษณ์ ; ในคำสุดท้าย test bench และผลการจำลองการทำงานในรูปที่ 5.5 และรูปที่ 5.6 ตามลำดับ

```
//module to illustrate the use of reduction operators
module reduction2(a, r_and, r_nand, r_or, r_nor, r_xor, r_xnor);
input      [7:0] a;
output     r_and, r_nand, r_or, r_nor, r_xor, r_xnor;

assign     r_and     = &a,     //reduction AND
           r_nand    = ~&a,    //reduction NAND
           r_or      = |a,     //reduction OR
           r_nor     = ~|a,    //reduction NOR
           r_xor     = ^a,     //reduction XOR
           r_xnor    = ^~a;    //reduction XNOR
endmodule
```

รูปที่ 5.4 โมดูล Verilog แบบกระแสข้อมูลสำหรับวงจรตัวดำเนินการลด

```
//test bench for reduction2 module
module reduction2_tb_v;
reg [7:0] a;
wire r_and, r_nand, r_or, r_nor, r_xor, r_xnor;
// Instantiate the Unit Under Test (UUT)
reduction2 uut ( .a(a), .r_and(r_and),
           .r_nand(r_nand), .r_or(r_or),
           .r_nor(r_nor), .r_xor(r_xor),
           .r_xnor(r_xnor));
initial
$monitor ("a=%b, r_and=%b, r_nand=%b,
r_or=%b, r_nor=%b, r_xor=%b, r_xnor=%b", a,
r_and, r_nand, r_or, r_nor, r_xor, r_xnor);

//apply input vectors
initial
begin
           #0        a = 8'b11000011;
           #10       a = 8'b10010111;
           #10       a = 8'b00000000;
           #10       a = 8'b01001111;
           #10       a = 8'b11111111;
           #10       $stop;
end
endmodule
```

รูปที่ 5.5 Test bench สำหรับวงจรตัวดำเนินการลดแบบกระแสข้อมูลของรูปที่ 5.4

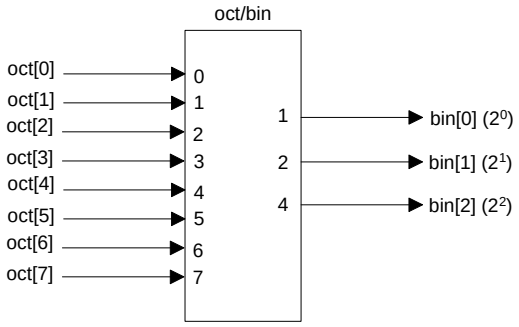
```
a=11000011, r_and=0, r_nand=1, r_or=1, r_nor=0, r_xor=0, r_xnor=1
a=10010111, r_and=0, r_nand=1, r_or=1, r_nor=0, r_xor=1, r_xnor=0
a=00000000, r_and=0, r_nand=1, r_or=0, r_nor=1, r_xor=0, r_xnor=1
a=01001111, r_and=0, r_nand=1, r_or=1, r_nor=0, r_xor=1, r_xnor=0
a=11111111, r_and=1, r_nand=0, r_or=1, r_nor=0, r_xor=0, r_xnor=1
```

รูปที่ 5.6 ผลการจำลองการทำงานสำหรับ Test bench ของรูปที่ 5.5

5.1.3 วงจรแปลงเลขฐานแปดเป็นฐานสอง

หัวข้อนี้แสดงตัวอย่างการโมเดลวงจรแปลงหรือวงจรเข้ารหัส (encoder) ด้วยโมเดลกระแสข้อมูล รูปที่ 5.7 แสดงบล็อกไดอะแกรมของวงจรแปลงเลขฐานแปดเป็นเลขฐานสอง (Octal-to-binary encoder) พร้อมด้วย truth table ในตารางที่ 5.1 สมการสำหรับวงจรแปลงเลขฐานแปดเป็นเลขฐานสองสามารถได้มาโดยตรงจาก truth table ดังสมการที่ 5.1

โมดูล Verilog ที่ใช้โมเดลกระแสข้อมูลอธิบายวงจรแปลงเลขฐานแปดเป็นเลขฐานสองแสดงดังรูปที่ 5.8 พร้อมด้วย test bench และผลการจำลองการทำงานดังรูปที่ 5.9 และ 5.10 ตามลำดับ



รูปที่ 5.7 บล็อกไดอะแกรมวงจรแปลงเลขฐานแปดเป็นเลขฐานสอง (Octal-to-binary encoder)

ตารางที่ 5.1 Truth table สำหรับวงจรแปลงเลขฐานแปดเป็นเลขฐานสองของรูปที่ 5.7

Inputs								Outputs		
oct[0]	oct[1]	oct[2]	oct[3]	oct[4]	oct[5]	oct[6]	oct[7]	bin[2]	bin[1]	bin[0]
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

$$
\begin{aligned}
bin[0] &= oct[1] + oct[3] + oct[5] + oct[7] \\
bin[1] &= oct[2] + oct[3] + oct[6] + oct[7] \\
bin[2] &= oct[4] + oct[5] + oct[6] + oct[7]
\end{aligned}
\qquad (5.1)
$$

```
//an octal-to-binary encoder
module encoder_8_to_3_df(oct, bin);
input    [0:7]    oct;
output   [2:0] bin;

assign   bin[0] = oct[1] + oct[3] + oct[5] + oct[7],
         bin[1] = oct[2] + oct[3] + oct[6] + oct[7],
         bin[2] = oct[4] + oct[5] + oct[6] + oct[7];
endmodule
```

รูปที่ 5.8 โมดูล Verilog แบบกระแสข้อมูลสำหรับวงจรแปลงเลขฐานแปดเป็นเลขฐานสองรูปที่ 5.7

```
//test bench for the 8-to-3 encoder
module encoder_8_to_3_df_tb_v;
reg [0:7] oct;
wire [2:0] bin;

// Instantiate the Unit Under Test (UUT)
        encoder_8_to_3_df uut (
                .oct(oct),
                .bin(bin)
        );
//display variables
initial
$monitor ("octal = %b, binary = %b", oct[0:7],
bin[2:0]);
```

```
//apply input vectors
initial
begin
        #0      oct[0:7]    = 8'b10000000;
        #10     oct[0:7]    = 8'b01000000;
        #10     oct[0:7]    = 8'b00100000;
        #10     oct[0:7]    = 8'b00010000;
        #10     oct[0:7]    = 8'b00001000;
        #10     oct[0:7]    = 8'b00000100;
        #10     oct[0:7]    = 8'b00000010;
        #10     oct[0:7]    = 8'b00000001;
        #10     $stop;
end
endmodule
```

รูปที่ 5.9 Test bench สำหรับวงจรแปลงเลขฐานแปดเป็นเลขฐานสองแบบกระแสข้อมูลของรูปที่ 5.8

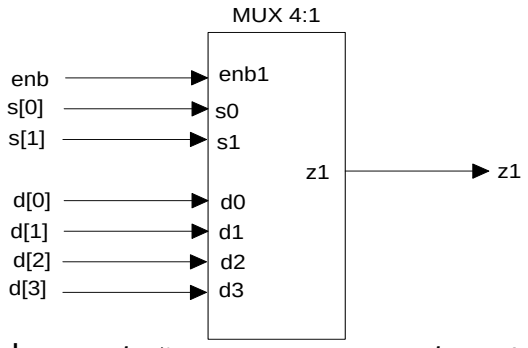
```
octal = 10000000, binary = 000
octal = 01000000, binary = 001
octal = 00100000, binary = 010
octal = 00010000, binary = 011
octal = 00001000, binary = 100
octal = 00000100, binary = 101
octal = 00000010, binary = 110
octal = 00000001, binary = 111
```

รูปที่ 5.10 ผลการจำลองการทำงานสำหรับ Test bench ของรูปที่ 5.9

5.1.4 มัลติเพลกเซอร์ 4:1

หัวข้อนี้แสดงตัวอย่างการโมเดลวงจรมัลติเพล็กเซอร์ด้วยโมเดลกระแสข้อมูล ข้อมูล รูปที่ 5.11 แสดงบล็อกไดอะแกรมของวงจรมัลติเพล็กเซอร์ 4:1 พร้อมด้วย truth table ในตารางที่ 5.2 สมการสำหรับวงจรมัลติเพล็กเซอร์ 4:1 สามารถได้มาโดยตรงจาก truth table ดังสมการที่ 5.2

โมดูล Verilog ที่ใช้โมเดลกระแสข้อมูลอธิบายวงจรมัลติเพล็กเซอร์ 4:1 แสดงดังรูปที่ 5.12 พร้อมด้วย test bench และผลการจำลองการทำงานดังรูปที่ 5.13 และ 5.14 ตามลำดับ



รูปที่ 5.11 บล็อกไดอะแกรมวงจรมัลติเพล็กเซอร์ 4:1

ตารางที่ 5.2 Truth table สำหรับวงจรมัลติเพล็กเซอร์ 4:1 ของรูปที่ 5.11

Select Inputs		Output
s1	s0	z1
0	0	d0
0	1	d1
1	0	d2
1	1	d3

z1 = s1’s0’d0 + s1’s0d1+ s1s0’d2 + s1s0d3 (5.2)

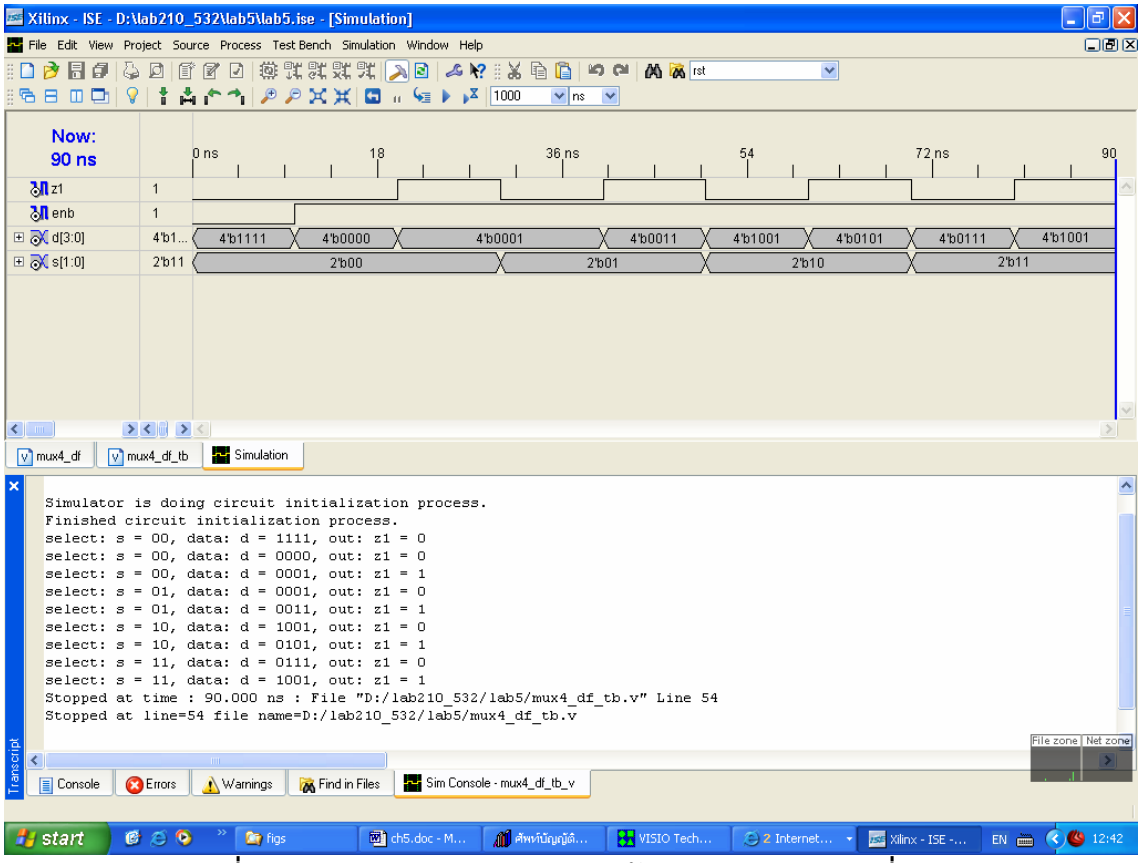
```
//dataflow 4:1 multiplexer
module mux4_df(s, d, enb, z1);
input    [1:0] s;
input    [3:0] d;
input      enb;
output     z1;

assign     z1 = (~s[1] & ~s[0] & d[0] & enb) |
                (~s[1] & s[0] & d[1] & enb) |
                ( s[1] & ~s[0] & d[2] & enb) |
                ( s[1] & s[0] & d[3] & enb);
endmodule
```

รูปที่ 5.12 โมดูล Verilog แบบกระแสข้อมูลสำหรับวงจรมัลติเพล็กเซอร์ 4:1 รูปที่ 5.11

```
//test bench for multiplexer 4:1
module mux4_df_tb_v;
reg [1:0] s;
reg [3:0] d;
reg enb;
wire z1;

// Instantiate the Unit Under Test (UUT)
        mux4_df uut ( .s(s), .d(d),
                      .enb(enb), .z1(z1)
        );
//display variables
initial
$monitor ("select: s = %b, data: d = %b, out: z1
= %b", s, d, z1);
```

```
//apply input vectors
initial
begin
#0      s = 2'b00; d = 4'b1111; enb = 1'b0;
#10     s = 2'b00; d = 4'b0000; enb = 1'b1;
#10     s = 2'b00; d = 4'b0001; enb = 1'b1;
#10     s = 2'b01; d = 4'b0001; enb = 1'b1;
#10     s = 2'b01; d = 4'b0011; enb = 1'b1;
#10     s = 2'b10; d = 4'b1001; enb = 1'b1;
#10     s = 2'b10; d = 4'b0101; enb = 1'b1;
#10     s = 2'b11; d = 4'b0111; enb = 1'b1;
#10     s = 2'b11; d = 4'b1001; enb = 1'b1;
#10     $stop;
end
endmodule
```

รูปที่ 5.13 Test bench สำหรับวงจรมัลติเพล็กเซอร์ 4:1 แบบกระแสข้อมูลของรูปที่ 5.12



รูปที่ 5.14 ผลการจำลองการทำงานสำหรับ Test bench ของรูปที่ 5.13

บทที่ 6
การโมเดลเชิงพฤติกรรม

เนื้อหาในบทนี้อธิบายรายละเอียดของการโมเดลวงจรเชิงพฤติกรรม (Behavioral modeling) ซึ่งเป็นการบรรยายพฤติกรรมของระบบดิจิตอลในระดับสูงหรือระดับสถาปัตยกรรม (high-level or architecture level) เป็นวิธีการสร้างฮาร์แวร์ด้วยวิธีการทางอัลกอริธึม ไม่ได้พิจารณาถึงการสร้างโดยตรงด้วยลอจิกเกท รูปแบบการโมเดลเชิงพฤติกรรมมีความคล้ายคลึงกับการเขียนภาษา C มาก

1.20 รูปแบบเชิงกระบวนคำสั่ง (Procedural construct)

กระบวนคำสั่ง (procedure) เป็นอนุกรมของการดำเนินการ (operations) ที่ถูกจัดขึ้นเพื่อการออกแบบโมดูลหนึ่งๆ ซึ่งถือว่าเป็นการออกแบบเชิงพฤติกรรม ภายในโมดูลจะไม่มีรายละเอียดของโครงสร้าง จึงง่ายต่อการบรรยายเชิงพฤติกรรมของฮาร์ดแวร์ ภาษา Verilog มีโครงสร้างบล็อกพฤติกรรม 2 แบบคือ **initial** และ **always** ในบล็อกพฤติกรรมทั้งสองแบบอาจมีคำสั่งเดียวหรือหลายคำสั่งก็ได้ และภายในโมดูลเดียวกันอาจมีบล็อก **initial** และ **always** หลายบล็อกก็ได้ บล็อกเหล่านี้เป็นพื้นฐานของการโมเดลเชิงพฤติกรรม และแต่ละบล็อกทำงานพร้อมๆกันโดยเริ่มต้นที่เวลาเท่ากับศูนย์ ภายในบล็อกเหล่านี้จะบรรจุคำสั่งเชิงกระบวนคำสั่งอื่นๆ ดังเช่น

การกำหนดค่าเชิงกระบวนคำสั่ง (procedural assignment)
- blocking
- non-blocking

คำสั่งแบบเงื่อนไข (conditional statement)

คำสั่ง **case**

คำสั่ง loop
- **for**
- **while**
- **repeat**
- **forever**

คำสั่งบล็อก (block statement)
- บล็อกเชิงลำดับ (sequential)
- บล็อกแบบขนาน (parallel)

การกำหนดค่าต่อเนื่องเชิงกระบวนคำสั่ง
- **assign**
- **deassign**
- **force**
- **release**

a. คำสั่ง initial

ทุกคำสั่งที่อยู่ภายในคำสั่ง **initial** รวมกันเป็นบล็อก **initial** จะถูกดำเนินการเพียงครั้งเดียวที่เวลาเท่ากับศูนย์ คำสั่ง **initial** เป็นวิธีสำหรับการกำหนดค่าเริ่มต้นและมอนิเตอร์ค่าตัวแปรก่อนที่ตัวแปรจะถูกใช้ในโมดูล และถูกใช้เพื่อสร้างรูปคลื่น (waveform) ด้วย ทุกคำสั่งภายในบล็อก **initial** จะถูกดำเนินการตามลำดับ การดำเนินการหรือการกำหนดค่าถูกควบคุมโดยสัญลักษณ์ # ไวยากรณ์การใช้งานคำสั่ง **initial** เป็นดังนี้

initial [optional timing control]

Procedural statement or block of procedural statements

แต่ละบล็อกจะมีความแข่งขนานกัน (concurrency) โดยเริ่มทำงานพร้อมๆกันที่เวลาเท่ากับศูนย์ และจบการทำงานที่เวลาต่างๆแยกกันโดยอิสระ ถ้าในบล็อกมีเพียงคำสั่งเดียวไม่จำเป็นต้องมีคำหลัก **begin...end** แต่จำเป็นต้องมีถ้าในบล็อกมีสองคำสั่งขึ้นไป

ตัวอย่างที่ 6.1 โมดูลแสดงตัวอย่างการใช้งานคำสั่ง **initial** แสดงดังรูปที่ 6.1 ตัวแปร x1 x2 x3 x4 และ x5 ถูกกำหนดให้มีค่าเริ่มต้นค่าหนึ่ง จากนั้นแต่ละตัวแปรถูกกำหนดค่าด้วยคำสั่ง **initial** หลายๆแบบ มีทั้งแบบที่มีคำสั่งเชิงกระบวนคำสั่งเดียว และเป็นบล็อกของคำสั่งเชิงกระบวนคำสั่ง ผลลัพธ์ของโมดูลและรูปคลื่นแสดงดังรูปที่ 6.2 และ 6.3 ตามลำดับ

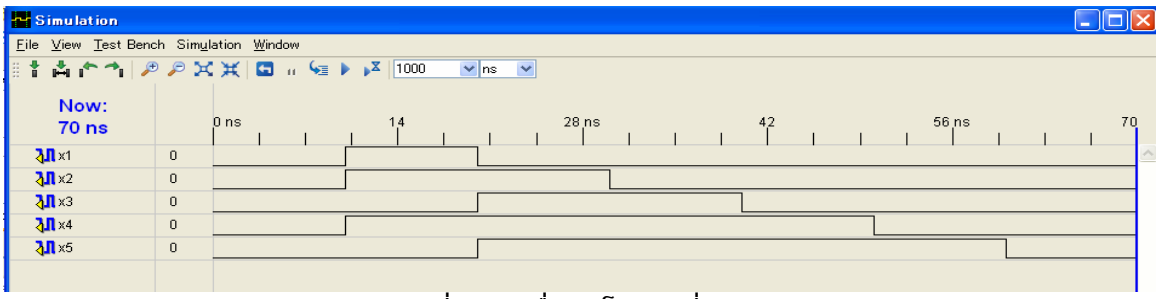
```
//module showing use of the initial keyword
module ex6_1 (x1, x2, x3, x4, x5);
output    x1, x2, x3, x4, x5;
reg       x1, x2, x3, x4, x5;

//display variables
initial
$monitor ($time, "  x1x2x3x4x5 = %b",
                  {x1,x2,x3,x4,x5});

//initialize variables to 0
//multiple statements require begin...end
initial
begin
          #0        x1 = 1'b0;
                    x2 = 1'b0;
                    x3 = 1'b0;
                    x4 = 1'b0;
                    x5 = 1'b0;
end

//set x1
//single statement requires no begin...end
initial
          #10       x1 = 1'b1;
```

```
//set x2 and x3
initial
begin
          #10       x2 = 1'b1;
          #10       x3 = 1'b1;
end

//set x4 and x5
initial
begin
          #10       x4 = 1'b1;
          #10       x5 = 1'b1;
end

//reset variables
initial
begin
          #20       x1 = 1'b0;
          #10       x2 = 1'b0;
          #10       x3 = 1'b0;
          #10       x4 = 1'b0;
          #10       x5 = 1'b0;
end

//determine length of simulation
initial
          #70       $finish;

endmodule
```

รูปที่ 6.1 โมดูลแสดงตัวอย่างการใช้งานคำสั่ง initial

```
 0    x1x2x3x4x5 = 00000
10    x1x2x3x4x5 = 11010
20    x1x2x3x4x5 = 01111
30    x1x2x3x4x5 = 00111
40    x1x2x3x4x5 = 00011
50    x1x2x3x4x5 = 00001
60    x1x2x3x4x5 = 00000
```

รูปที่ 6.2 ผลลัพธ์ของโมดูลรูปที่ 6.1



รูปที่ 6.3 รูปคลื่นของโมดูลรูปที่ 6.1

รูปที่ 6.1 เป็นตัวอย่างการสร้างรูปคลื่น มีคำสั่ง **initial** ถึง 7 คำสั่ง โดยคำสั่งแรกเป็นการเรียกซิสเต็มแทสค์ **$monitor** ซึ่งทำให้ข้อความภายในเครื่องหมายคำพูดถูกแสดงออกมาทุกครั้งที่มีการเปลี่ยนแปลงค่าตัวแปรในอาร์กิวเมนต์ที่ถูกแสดงรายการไว้ ซิสเต็มแทสค์ **$time** ทำการคืนค่าเวลาของการจำลองการทำงานในรูปแบบจำนวนเต็ม 64 บิต

คำสั่ง **initial** คำสั่งที่ 2 เป็นการกำหนดค่าเริ่มต้นเท่ากับศูนย์ให้กับทุกตัวแปร คำสั่งที่ 3 เป็นการเซทค่า x1 ที่เวลา 10 หน่วยเวลา เนื่องจากคำสั่ง **initial** ทุกคำสั่งเริ่มพร้อมกันที่เวลาศูนย์ คำสั่งที่ 4 ก็เป็นการเซทค่า x2 ที่เวลา 10 หน่วยเวลาเช่นกัน และเซท x3 ที่เวลา 20 หน่วยเวลา ทำนองเดียวกันสำหรับคำสั่งที่ 5 คำสั่งที่ 6 เป็นการรีเซทค่าตัวแปรทุกตัวตามลำดับเวลาที่ระบุ คำสั่งที่ 7 เป็นการสิ้นสุดการจำลองการทำงาน

b. คำสั่ง always

คำสั่ง **always** ดำเนินการตามคำสั่งเชิงพฤติกรรมที่อยู่ภายในบล็อก **always** ซ้ำๆในลักษณะการวนลูปและเริ่มการดำเนินการที่เวลาศูนย์ การดำเนินการจะทำซ้ำต่อเนื่องจนกระทั่งการจำลองการทำงานสิ้นสุด คำสั่งภายในบล็อก **always** เป็นคำสั่งเชิงพฤติกรรม หรือก็คือคำสั่งเชิงกระบวนคำสั่งนั่นเอง ไวยากรณ์การใช้งานคำสั่ง **always** เป็นดังนี้

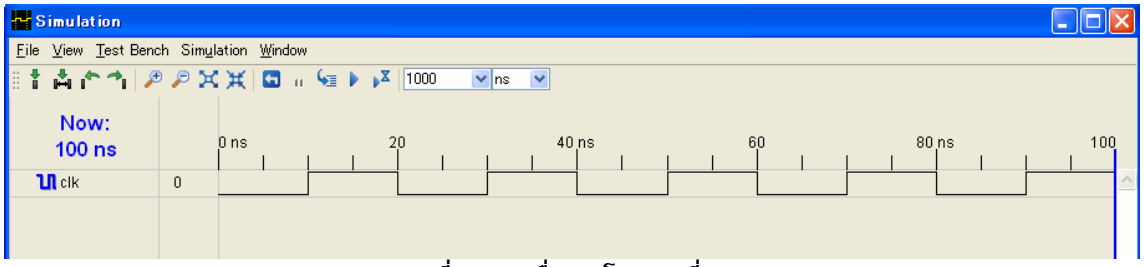
always [optional timing control]

Procedural statement or block of procedural statements

ตัวอย่างที่ 6.2 การใช้งานโดยทั่วไปของคำสั่ง **always** ที่แสดงดังรูปที่ 6.4 เป็นการสร้างสัญญาณนาฬิกาเพื่อใช้ใน test bench สัญญาณ clk ถูกกำหนดค่าเริ่มต้นเป็นศูนย์ด้วยคำสั่ง **initial** คำสั่งแรก จากนั้นค่าของสัญญาณ clk ถูกสลับค่าทุกๆ 10 หน่วยเวลาด้วยคำสั่ง **always** ทำให้ได้สัญญาณนาฬิกาที่มีคาบเท่ากับ 20 หน่วยเวลา และสัญญาณ clk ถูกหยุดเมื่อเวลาผ่านไป 100 หน่วยเวลาด้วยคำสั่ง **initial** คำสั่งสุดท้ายที่มีการเรียกใช้ซิสเต็มแทสค์ **$finish** ซึ่งทำให้การจำลองแบบการทำงานสิ้นสุด รูปคลื่นที่ได้ถูกแสดงดังรูปที่ 6.5

```
//clock generation using initial and always statements
module ex6_2(clk);
output  clk;
reg     clk;

//initialize clock to 0
initial
        clk = 1'b0;

//toggle clock every 10 time units
always
        #10     clk = ~clk;

//determine length of simulation
initial
        #100    $finish;

endmodule
```

รูปที่ 6.4 โมดูลแสดงตัวอย่างการใช้สร้างสัญญาณนาฬิกา



รูปที่ 6.5 รูปคลื่นของโมดูลรูปที่ 6.4

ตัวอย่างที่ 6.3 คำสั่ง **always** มักถูกใช้ร่วมกับ รายการควบคุมเหตุการณ์ (event control list) หรือ รายการความไวต่อเหตุการณ์ (sensitivity list) เพื่อดำเนินการบล็อกเชิงลำดับ (sequential block) เมื่อมีการเปลี่ยนแปลงของสัญญาณภายใน sensitivity list คำสั่งหรือบล็อกของคำสั่งภายในบล็อก **always** จะถูกดำเนินการ คำหลัก **or** มีไว้สำหรับการระบุกรณีที่มีหลายเหตุการณ์แสดงดังรูปที่ 6.6 ซึ่งเป็นโมดูลของเกท AND 3 อินพุท เมื่อมีการเปลี่ยนแปลงสภาวะของอินพุท x1 x2 หรือ x3 คำสั่งภายในบล็อก **always** จะถูกดำเนินการ ในกรณีนี้ไม่จำเป็นต้องมีคำหลัก begin...end เนื่องจากมีคำสั่งเดียว ตัวแปรเป้าหมาย z1 ภายในบล็อก **always** ต้องถูกประกาศเป็นชนิด **reg** ในตัวอย่างนี้ดีเลย์ 5 หน่วยเวลาถูกกำหนดให้เป็นเวลาแพร่กระจายของเกท (propagation delay)

รูปที่ 6.7 แสดง test bench ของโมดูลเกท AND 3 อินพุทข้างต้น อินพุทเวกเตอร์ invect มีความกว้าง 4 บิตเพื่อรองรับการป้อนอินพุททั้งสามให้ครบทั้ง 8 รูปแบบ ผลลัพธ์ของโมดูลและรูปคลื่นแสดงดังรูปที่ 6.8 และ 6.9 ตามลำดับ เอาท์พุท z1 อยู่ในสภาวะไม่ทราบค่าเป็นเวลา 5 หน่วยเวลาแรกจนกระทั่งอินพุทจะแพร่กระจายผ่านเกท AND สังเกตเห็นว่าอินพุทถูกป้อนเสร็จที่เวลา 42 หน่วยเวลา แต่เอาท์พุทยังไม่เปลี่ยนแปลงไปจนกระทั่งเวลาผ่านไป 47 หน่วยเวลา ซึ่งก็คือ 5 หน่วยเวลาถัดมา

```
//behavioral 3-input AND gate
module and3_bh(x1, x2, x3, z1);
input   x1, x2, x3;
output  reg     z1;

always @(x1 or x2 or x3)
        z1 = #5 (x1 & x2 & x3);

endmodule
```

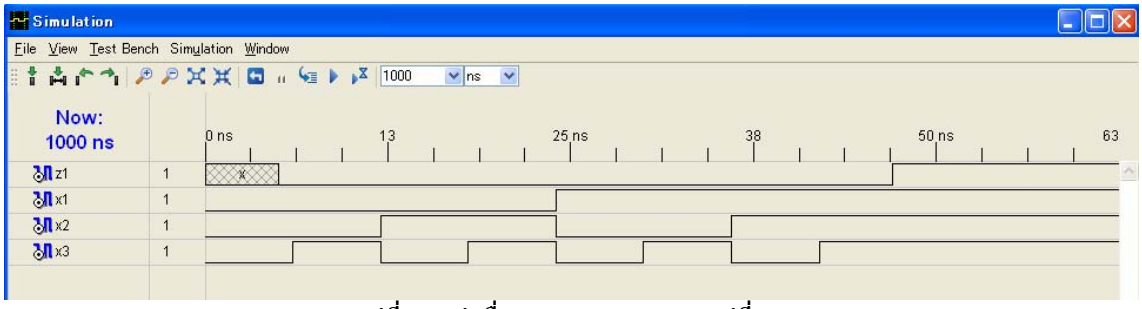
รูปที่ 6.6 โมดูลแสดงตัวอย่างการใช้งานคำสั่ง always ที่มี event control list

```
//test bench for behovioral 3-input AND gate
module and3_bh_tb_v;
reg   x1, x2, x3;
wire   z1;
// Instantiate the Unit Under Test (UUT)
        and3_bh uut (.x1(x1), .x2(x2), .x3(x3), .z1(z1));

//generate stimulus and display variables
initial
begin: apply_stimulus
        reg [3:0] invect;
        for (invect=0; invect<8; invect=invect+1)
        begin
                {x1, x2, x3} = invect[2:0];
                #6 $display ($time, "          x1x2x3 = %b, z1 = %b", {x1,x2,x3}, z1);
        end
end
endmodule
```

รูปที่ 6.7 Test bench ของโมดูลเกท AND 3 อินพุทของรูปที่ 6.6

```
 6      x1x2x3 = 000, z1 = 0
12      x1x2x3 = 001, z1 = 0
18      x1x2x3 = 010, z1 = 0
24      x1x2x3 = 011, z1 = 0
30      x1x2x3 = 100, z1 = 0
36      x1x2x3 = 101, z1 = 0
42      x1x2x3 = 110, z1 = 0
48      x1x2x3 = 111, z1 = 1
```

รูปที่ 6.8 ผลลัพธ์ของ test bench ของรูปที่ 6.7



รูปที่ 6.9 รูปคลื่นของ test bench ของรูปที่ 6.7

ตัวอย่างที่ 6.4 ตัวอย่างนี้แสดงการออกแบบวงจร add-shift อย่างง่ายเพื่อแสดงการโมเดลเชิงพฤติกรรมของวงจรบวก 8 บิต พร้อมด้วยการเลื่อนซ้ายและเลื่อนขวา ดังโมดูลในรูปที่ 6.10 รายการใน sensitivity list ของบล็อก **always** มีตัวแปร a และ b เมื่อมีการเปลี่ยนแปลงของ a หรือ b จะได้ผลบวก sum จากนั้น จะได้ผลการเลื่อนซ้าย 4 บิตซึ่งให้ผลเท่าการคูณด้วย 16 เก็บไว้ที่ left_shift_result และผลการเลื่อนขวา 4 บิตซึ่งให้ผลเท่าการหารด้วย 16 เก็บไว้ที่ right_shift_result ในที่นี้ต้องประกาศให้ sum มีขนาด 9 บิตเพื่อรองรับผลการบวก 8 บิต และประกาศให้ left_shift_result และ right_shift_result มีขนาด 16 บิต เพื่อรองรับการเลื่อนหลายตำแหน่ง สังเกตเห็นได้ว่าภายในโมดูลมีการบรรยายพฤติกรรมอย่างง่าย ไม่มีรายละเอียดของการออกแบบระดับเกทเลย

รูปที่ 6.11 แสดง test bench ของโมดูลในรูปที่ 6.10 พร้อมด้วยผลการจำลองการทำงานและรูปคลื่นในรูปที่ 6.12 และ 6.13 ตามลำดับ

```
//add shift operations
module add_shift_bh(a, b, sum, right_shift_result, left_shift_result);
input      [7:0]      a, b;
output reg  [8:0]     sum;
output reg [15:0]     right_shift_result, left_shift_result;

always @(a or b)
begin
          sum = a + b;
          left_shift_result = sum << 4;
          right_shift_result = sum >> 4;
end
endmodule
```

รูปที่ 6.10 โมดูลเชิงพฤติกรรมสำหรับงวงจร add shift

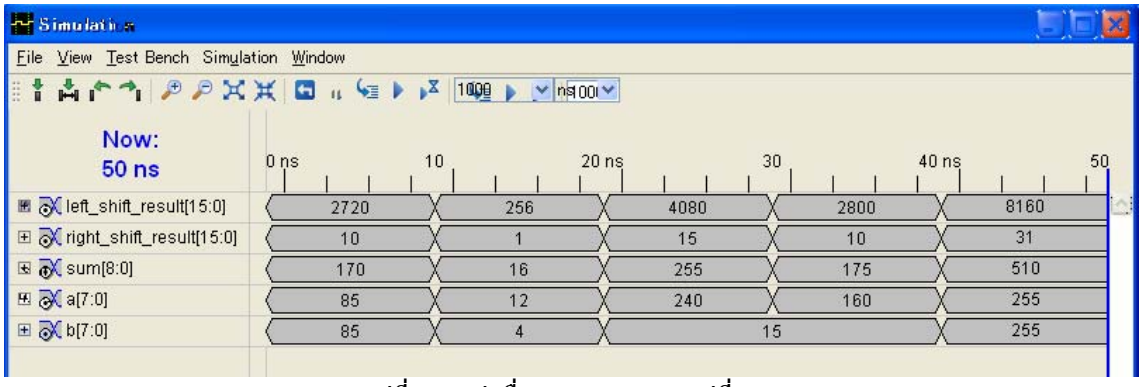
```
//add shift test bench
module add_shift_bh_tb_v;
reg [7:0] a, b;
wire [8:0] sum;
wire [15:0] right_shift_result, left_shift_result;

// Instantiate the Unit Under Test (UUT)
add_shift_bh uut (.a(a), .b(b), .sum(sum),
          .right_shift_result(right_shift_result),
          .left_shift_result(left_shift_result));

//display variables
initial
$monitor ("a=%b, b=%b, sum=%b,
left_shift_result=%b, right_shift_result=%b,", a, b,
sum, left_shift_result, right_shift_result);
```

```
//apply input vectors
initial
begin
          #0        a = 8'b01010101;
                    b = 8'b01010101;
          #10       a = 8'b00001100;
                    b = 8'b00000100;
          #10       a = 8'b11110000;
                    b = 8'b00001111;
          #10       a = 8'b10100000;
                    b = 8'b00001111;
          #10       a = 8'b11111111;
                    b = 8'b11111111;
          #10       $stop;
end
endmodule
```

รูปที่ 6.11 Test bench สำหรับโมดูล add shift ของรูปที่ 6.10

```
a=01010101, b=01010101, sum=010101010,
left_shift_result=0000101010100000, right_shift_result=0000000000001010,
a=00001100, b=00000100, sum=000010000,
left_shift_result=0000000100000000, right_shift_result=0000000000000001,
a=11110000, b=00001111, sum=011111111,
left_shift_result=0000111111110000, right_shift_result=0000000000001111,
a=10100000, b=00001111, sum=010101111,
left_shift_result=0000101011110000, right_shift_result=0000000000001010,
a=11111111, b=11111111, sum=111111110,
left_shift_result=0001111111100000, right_shift_result=0000000000011111,
```

รูปที่ 6.12 ผลลัพธ์ของ test bench รูปที่ 6.11



รูปที่ 6.13 รูปคลื่นของ test bench รูปที่ 6.11

ตัวอย่างที่ 6.5 แสดงตัวอย่างการออกแบบวงจร 8-bit odd parity generator โดยการใช้โมเดลเชิงพฤติกรรม อินพุทของวงจรเป็นเวกเตอร์ขนาด 8 บิต x[7:0] เอาท์พุทของวงจรเป็นสเกลาร์ z1 ซึ่งต้องถูกประกาศเป็นข้อมูลชนิด **reg** เนื่องจากอยู่ภายในบล็อก **always** เมื่อมีการเปลี่ยนแปลงของอินพุท คำสั่งภายในบล็อก **always** จะถูกดำเนินการ

รูปที่ 6.14 แสดงโมดูลเชิงพฤติกรรมของวงจร 8-bit odd parity generator ข้างต้น พร้อมด้วยtest bench ผลการจำลองการทำงาน และผลรูปคลื่นในรูปที่ 6.15 6.16 และ 6.17 ตามลำดับ

```
//behavioral 8-bit odd parity generator
module par_gen8_bh(x, z1);
input    [7:0]   x;
output    reg    z1;

always @(x)
          z1 = ~^x;

endmodule
```

รูปที่ 6.14 โมดูลเชิงพฤติกรรมสำหรับงวงจร 8-bit odd parity generator

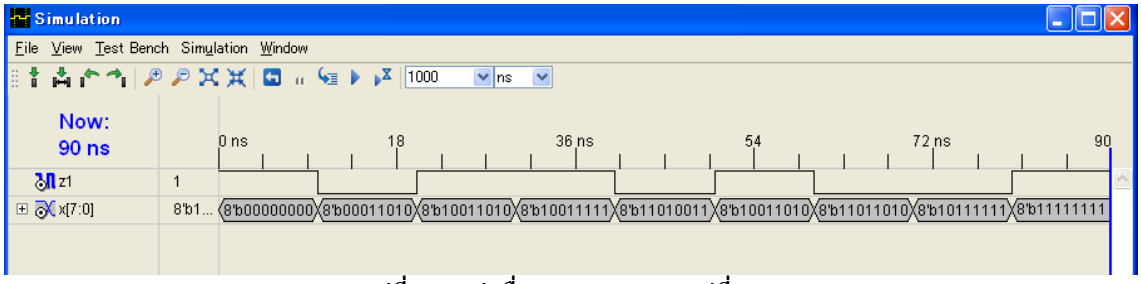
```
//test bench for the behavioral 8-bit odd parity generator
module par_gen8_bh_tb_v;
reg     [7:0] x;
wire     z1;

// Instantiate the Unit Under Test (UUT)
          par_gen8_bh uut (.x(x), .z1(z1));

//display variables
initial
$monitor ("x=%b, z1=%b", x, z1);
```

```
//apply input vectors
initial
begin
          #0        x = 8'b00000000;
          #10       x = 8'b00011010;
          #10       x = 8'b10011010;
          #10       x = 8'b10011111;
          #10       x = 8'b11010011;
          #10       x = 8'b10011010;
          #10       x = 8'b11011010;
          #10       x = 8'b10111111;
          #10       x = 8'b11111111;
          #10       $stop;
end
endmodule
```

รูปที่ 6.15 Test bench สำหรับโมดูล 8-bit odd parity generator ของรูปที่ 6.14

```
x=00000000, z1=1
x=00011010, z1=0
x=10011010, z1=1
x=10011111, z1=1
x=11010011, z1=0
x=10011010, z1=1
x=11011010, z1=0
x=10111111, z1=0
x=11111111, z1=1
```

รูปที่ 6.16 ผลลัพธ์ของ test bench รูปที่ 6.15



รูปที่ 6.17 รูปคลื่นของ test bench รูปที่ 6.15

1.21 การกำหนดค่าเชิงกระบวนคำสั่ง

การกำหนดค่าเชิงกระบวนคำสั่ง (procedural assignment) เป็นคำสั่งสำหรับการกำหนดค่าให้กับตัวแปรรีจิสเตอร์ชนิด **reg integer real realtime** และ **time** การกำหนดค่าเชิงกระบวนคำสั่งเป็นการกำหนดค่าที่ปรากฏอยู่ภายในบล็อก **initial** และ **always** อาจจะเลือกกำหนดค่าให้กับบางบิต บางส่วนของเวกเตอร์ หรือเวกเตอร์ที่ต่อกัน การกำหนดค่าเชิงกระบวนคำสั่งถูกดำเนินการตามลำดับ ภายในบล็อกเชิงลำดับ (sequential block) อาจมีการกำหนดค่าเชิงกระบวนคำสั่งซึ่งมีการกำหนดเวลาด้วยดังเช่น

```
reg [15:0] bus_a;
...
initial
begin
    #0      bus_a = 16'h00ff;
    #5      bus_a = 16'hff00;
    #5      bus_a = 16'habdc;
    #5      bus_a = 16'h65ab;
end
```

การกำหนดค่าเชิงกระบวนคำสั่ง (ซึ่งใช้ในการโมเดลเชิงพฤติกรรม) และการกำหนดค่าแบบต่อเนื่อง (ซึ่งใช้ในการโมเดลแบบกระแสข้อมูล) มีความแตกต่างกันดังตารางที่ 6.1

ตารางที่ 6.1 ความแตกต่างระหว่างการกำหนดค่าเชิงกระบวนคำสั่งและการกำหนดค่าแบบต่อเนื่อง

การกำหนดค่าเชิงกระบวนคำสั่ง	การกำหนดค่าแบบต่อเนื่อง
i. ปรากฏในบล็อก **initial** หรือ **always**	v. ไม่ถูกใช้ในบล็อก **initial** หรือ **always**
ii. ดำเนินการสัมพันธ์กับคำสั่งอื่นในบล็อก **initial** หรือ **always**	vi. ดำเนินการแข่งขนาน (concurrency) กับคำสั่งอื่น
ii. ขับรีจิสเตอร์	ii. ขับเน็ท
v. ใช้สัญลักษณ์ = (blocking) หรือ <= (nonblocking)	ii. ใช้สัญลักษณ์ = เพื่อการกำหนดค่า

6.2.1 ดีเลย์ในคำสั่ง

การกำหนดค่าเชิงกระบวนคำสั่งอาจมีการกำหนดดีเลย์ได้ ดีเลย์ที่ปรากฏทางด้านหน้าของนิพจน์ทางขวาของการกำหนดค่าคือ ดีเลย์ในคำสั่ง (intra-statement delay) เป็นการหน่วงเวลาในการกำหนดค่าผลลัพธ์ทางด้านขวาให้กับเป้าหมายทางด้านซ้าย ตัวอย่างเช่น

```
z1 = #5 (x1 & x2);
```

เมื่อนิพจน์ (x1 & x2) ถูกคำนวณ จะมีการดีเลย์ 5 หน่วยเวลา ก่อนการกำหนดค่าผลลัพธ์ให้กับ z1

จุดประสงค์ของการมีดีเลย์ในคำสั่งก็เพื่อจำลองเวลาในการทำงานของลอจิกเกท หรือดีเลย์แพร่กระจายของเกท (gate propagation delay) ซึ่งก็คือเวลาของการแพร่กระจายของสัญญาณอินพุทผ่านไปยังเอาท์พุทของเกทหนึ่งๆนั่นเอง

ตัวอย่างที่ 6.6 แสดงตัวอย่างการใช้ดีเลย์ในคำสั่งสำหรับการโมเดลเชิงพฤติกรรมของวงจรที่มีเอาท์พุท z1 z2 และ z3 ดังรูปที่ 6.18 พร้อมด้วยtest bench ผลการจำลองการทำงาน และผลรูปคลื่นในรูปที่ 6.19 6.20 และ 6.21 ตามลำดับ ผลลัพธ์ของวงจรจะไม่ปรากฏจนกระทั่งถึงเวลาที่ถูกกำหนด เนื่องจากในบล็อก always เป็นการกำหนดค่าเชิงกระบวนคำสั่งแบบ blocking ดังนั้นจึงเกิดการสะสมของดีเลย์ กล่าวคือ หลังจากอินพุทมีการเปลี่ยนค่า z1 จะรับค่าใหม่เมื่อเวลาผ่านไป 2 หน่วยเวลา z2 จะรับค่าใหม่เมื่อเวลาผ่านไป 5 หน่วยเวลา และ z3 จะรับค่าใหม่เมื่อเวลาผ่านไป 9 หน่วยเวลา

```
//behavioral model to demonstrate intra-statement delay
module intra_delay_bh(x1, x2, x3, x4, z1, z2, z3);
input   x1, x2, x3, x4;
output reg z1, z2, z3;

always @(x1 or x2 or x3 or x4)
begin
        z1 = #2 (x1 & ~x2) ^ (~x3 & x4);
        z2 = #3 (x1 >= x2) ? x3 : x4;
        z3 = #4 ~^{x1,x2,x3,x4};
end
endmodule
```

รูปที่ 6.18 ตัวอย่างการใช้ดีเลย์ในคำสั่งสำหรับการโมเดลเชิงพฤติกรรม

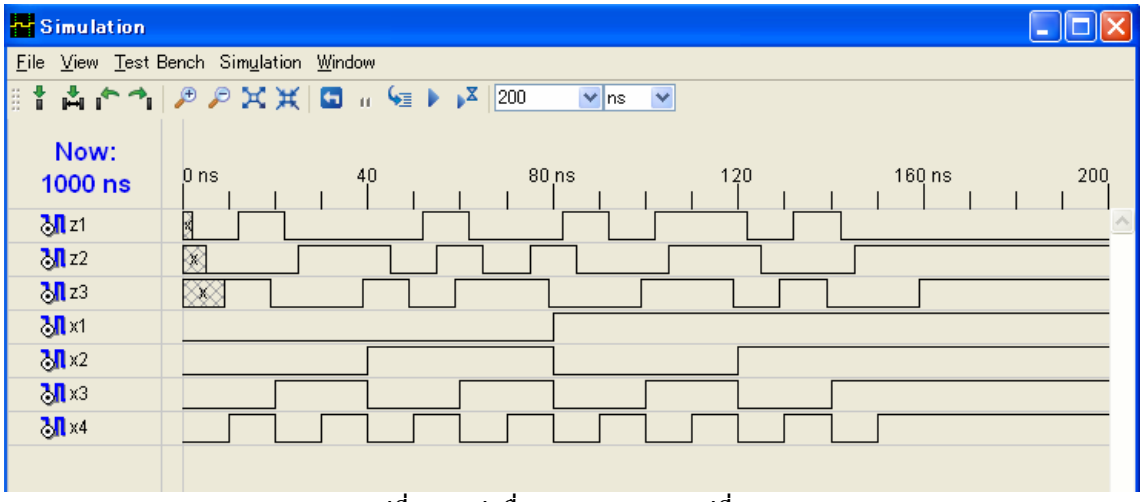
```
//test bench for intra-statement delay
module intra_delay_bh_tb_v;
reg   x1, x2, x3, x4;
wire   z1, z2, z3;

// Instantiate the Unit Under Test (UUT)
intra_delay_bh uut (.x1(x1), .x2(x2),
        .x3(x3), .x4(x4),
        .z1(z1), .z2(z2), .z3(z3));
```

```
//apply input vectors and display variables
initial
begin: apply_stimulus
  reg [4:0] invect;
  for (invect=0; invect<16; invect=invect+1)
    begin
      {x1,x2,x3,x4} = invect[4:0];
       #10  $display("x1x2x3x4 = %b, z1=%b, z2=%b, z3=%b", {x1,x2,x3,x4}, z1, z2, z3);
    end
end
endmodule
```

รูปที่ 6.19 Test bench สำหรับโมดูลของรูปที่ 6.18

x1x2x3x4 = 0000, z1=0, z2=0, z3=1	x1x2x3x4 = 1000, z1=1, z2=0, z3=0
x1x2x3x4 = 0001, z1=1, z2=0, z3=0	x1x2x3x4 = 1001, z1=0, z2=0, z3=1
x1x2x3x4 = 0010, z1=0, z2=1, z3=0	x1x2x3x4 = 1010, z1=1, z2=1, z3=1
x1x2x3x4 = 0011, z1=0, z2=1, z3=1	x1x2x3x4 = 1011, z1=1, z2=1, z3=0
x1x2x3x4 = 0100, z1=0, z2=0, z3=0	x1x2x3x4 = 1100, z1=0, z2=0, z3=1
x1x2x3x4 = 0101, z1=1, z2=1, z3=1	x1x2x3x4 = 1101, z1=1, z2=0, z3=0
x1x2x3x4 = 0110, z1=0, z2=0, z3=1	x1x2x3x4 = 1110, z1=0, z2=1, z3=0
x1x2x3x4 = 0111, z1=0, z2=1, z3=0	x1x2x3x4 = 1111, z1=0, z2=1, z3=1

รูปที่ 6.20 ผลลัพธ์ของ test bench รูปที่ 6.19



รูปที่ 6.21 รูปคลื่นของ test bench รูปที่ 6.19

6.2.2 ดีเลย์ระหว่างคำสั่ง

ดีเลย์ระหว่างคำสั่ง (inter-statement delay) เป็นดีเลย์ที่เกิดขึ้นก่อนที่คำสั่งจะถูกดำเนินการ ตัวอย่าง เช่น

```
z1 = (x1 & x2) | x3;
#5 z2 = x4 ^ x5;
```

ดีเลย์ที่ถูกกำหนดให้กับคำสั่งที่สองมีความหมายว่า เมื่อคำสั่งแรกถูกดำเนินการเสร็จแล้ว ให้รอ 5 หน่วยเวลาก่อนเริ่มดำเนินการคำสั่งที่สอง ถ้าไม่มีการกำหนดดีเลย์ในการกำหนดค่าเชิงกระบวนคำสั่ง ถือว่าดีเลย์เป็นศูนย์

ตัวอย่างที่ 6.7 แสดงตัวอย่างการใช้ดีเลย์ระหว่างคำสั่งสำหรับการโมเดลเชิงพฤติกรรมของวงจรที่มีเอาท์พุท z1 และ z2 ดังรูปที่ 6.22 พร้อมด้วยtest bench ผลการจำลองการทำงาน และผลรูปคลื่นในรูปที่ 6.23 6.24 และ 6.25 ตามลำดับ ผลลัพธ์ของวงจรจะไม่ปรากฏจนกระทั่งถึงเวลาที่ถูกกำหนด เมื่อคำสั่งสำหรับ z1 ถูกดำเนินการเสร็จแล้ว จะมีดีเลย์ 5 หน่วยเวลาเกิดขึ้นก่อนที่ คำสั่งสำหรับ z2 ถูกดำเนินการ

```
//behavioral module to illustrate inter-statement delay
module inter_delay_bh(x1, x2, x3, z1, z2);
input   x1, x2, x3;
output reg   z1, z2;

always @(x1 or x2 or x3)
begin
        z1 = (x1 & x2 & x3);
   #5   z2 = ~^{x1,x2,x3};
end
endmodule
```

รูปที่ 6.22 ตัวอย่างการใช้ดีเลย์ระหว่างคำสั่งสำหรับการโมเดลเชิงพฤติกรรม

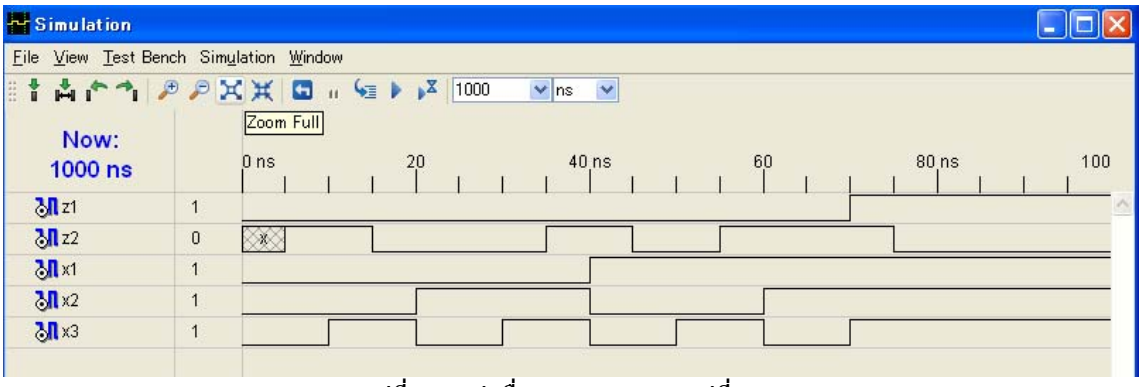
```
//test bench for inter-statement delay
module inter_delay_bh_tb_v;
reg    x1, x2, x3;
wire   z1, z2;

// Instantiate the Unit Under Test (UUT)
inter_delay_bh uut (.x1(x1), .x2(x2),
            .x3(x3), .z1(z1), .z2(z2));
```

```
//apply input vectors and display variables
initial
begin: apply_stimulus
    reg [3:0] invect;
    for (invect=0; invect<8; invect=invect+1)
        begin
            {x1, x2, x3} = invect [3:0];
            #10 $display ("x1 x2 x3 = %b, z1 = %b, z2
= %b", {x1, x2, x3}, z1, z2);
        end
end
endmodule
```

รูปที่ 6.23 Test bench สำหรับโมดูลของรูปที่ 6.22

```
x1 x2 x3 = 000, z1 = 0, z2 = 1
x1 x2 x3 = 001, z1 = 0, z2 = 0
x1 x2 x3 = 010, z1 = 0, z2 = 0
x1 x2 x3 = 011, z1 = 0, z2 = 1
x1 x2 x3 = 100, z1 = 0, z2 = 0
x1 x2 x3 = 101, z1 = 0, z2 = 1
x1 x2 x3 = 110, z1 = 0, z2 = 1
x1 x2 x3 = 111, z1 = 1, z2 = 0
```

รูปที่ 6.24 ผลลัพธ์ของ test bench รูปที่ 6.23



รูปที่ 6.25 รูปคลื่นของ test bench รูปที่ 6.23

6.2.3 การกำหนดค่าแบบ blocking

การกำหนดค่าเชิงกระบวนคำสั่งที่เป็นแบบ blocking จะทำงานลงมาทีละบรรทัดหรือ เป็นลำดับก่อนหลังนั่นเอง โดยใช้สัญลักษณ์ "=" เป็นตัวกำหนดค่าให้กับตัวแปรเป้าหมายทางด้านขวา คำสั่งที่มาหลังจะถูกบล็อกไว้จนกระทั่งคำสั่งที่มาก่อนถูกดำเนินการเสร็จ

ตัวอย่างที่ 6.8 แสดงตัวอย่างการกำหนดค่าแบบ blocking โดยใช้ดีเลย์ในคำสั่งเพื่อช่วยให้เข้าใจง่ายขึ้น โมเดลเชิงพฤติกรรมของวงจรที่มีเอาท์พุท z1 z2 และ z3 แสดงดังรูปที่ 6.26 พร้อมด้วยtest bench และผลรูปคลื่นในรูปที่ 6.27 และ 6.28 ตามลำดับ ผลลัพธ์ของวงจรจะไม่ปรากฏจนกระทั่งถึงเวลาที่ถูกกำหนด คำสั่งสำหรับ z1 z2 และ z3 จะถูกดำเนินการตามลำดับโดยห่างกัน 1 หน่วยเวลาดังผลรูปคลื่น

```
//blocking intra-statement delay
module blocking1_bh(x1, x2, x3, z1, z2, z3);
input     x1, x2, x3;
output reg    z1, z2, z3;
```

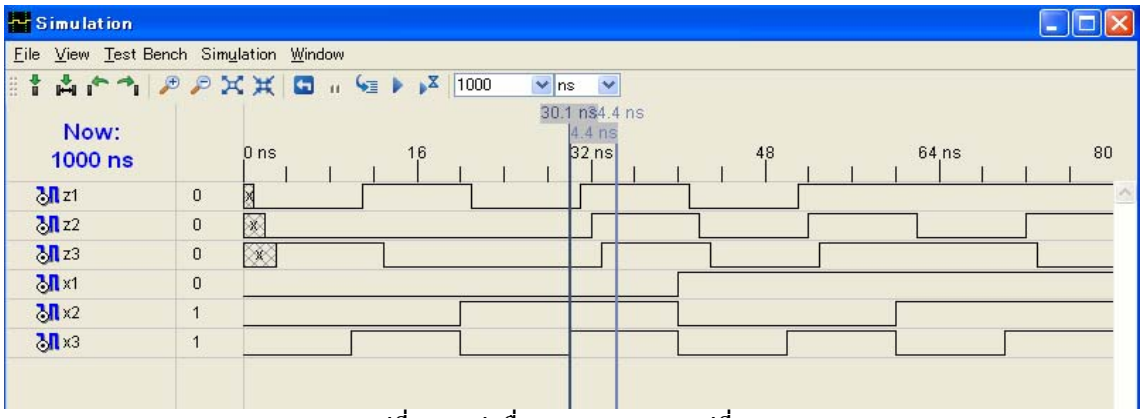
```
always @(x1 or x2 or x3)
begin
        z1 = #1 (x1 & x2) | x3;
        z2 = #1 (x1 | x2) & x3;
        z3 = #1 ~^{x1, x2, x3};
end
endmodule
```

รูปที่ 6.26 ตัวอย่างการกำหนดค่าแบบ blocking ซึ่งมีดีเลย์ในคำสั่ง

```
//test bench for blocking intra-statement delay
module blocking1_bh_tb_v;
reg   x1, x2, x3;
wire  z1, z2, z3;

// Instantiate the Unit Under Test (UUT)
blocking1_bh uut (.x1(x1), .x2(x2), .x3(x3),
                  .z1(z1), .z2(z2), .z3(z3)
                );
```

```
//apply input vectors and display variables
initial
begin: apply_stimulus
    reg [3:0] invect;
    for (invect=0; invect<8; invect=invect+1)
        begin
            {x1, x2, x3} = invect [3:0];
            #10 $display ("x1 x2 x3 = %b, z1 = %b,
z2 = %b, z3 = %b", {x1, x2, x3}, z1, z2, z3);
        end
end
endmodule
```

รูปที่ 6.27 Test bench สำหรับโมดูลของรูปที่ 6.26



รูปที่ 6.28 รูปคลื่นของ test bench รูปที่ 6.27

6.2.4 การกำหนดค่าแบบ non-blocking

การกำหนดค่าเชิงกระบวนคำสั่งที่เป็น non-blocking จะทำงานพร้อมกันทุกบรรทัดหรือ เป็นแบบแข่งขานนั่นเอง โดยใช้สัญลักษณ์ "<=" เป็นตัวกำหนดค่าให้กับตัวแปรเป้าหมายทางด้านขวา คำสั่งที่มาหลังจะไม่ถูกบล็อกไว้จนกระทั่งคำสั่งที่มาก่อนถูกดำเนินการเสร็จ แต่ทุกคำสั่งจะถูกดำเนินการไปพร้อมๆกัน

ตัวอย่างที่ 6.9 ออกแบบวงจร 4-bit parallel-in serial-out shift register โดยใช้การกำหนดค่าเชิงกระบวนคำสั่งที่เป็น non-blocking ดังโมดูลเชิงพฤติกรรมในรูปที่ 6.29 การดำเนินการโหลดและเลื่อนข้อมูลถูกควบคุมด้วยสัญญาณ load เมื่อสัญญาณ load เป็นลอจิก 1 ข้อมูลจะถูกโหลดจากบัสอินพุท x[1:4] มาเก็บที่รีจิสเตอร์ภายใน y[1:4] และเมื่อสัญญาณ load เป็นลอจิก 0 ข้อมูลจะถูกเลื่อนไปทางขวา 1 บิต และบิตซ้ายสุดจะถูกแทนด้วย

ศูนย์ เอาท์พุทอนุกรมของวงจร z1 ก็คือเอาท์พุทของ y[4] โมดูล test bench ผลการจำลองการทำงาน และผลรูปคลื่นในรูปที่ 6.30 6.31 และ 6.32 ตามลำดับ

```
//parallel-in serial-out shift register
module piso_bh(clk, load, x, z1);
input   clk, load;
input   [1:4] x;
output    z1;
reg [1:4] y;

assign z1 = y[4];
```

```
always @(posedge clk)
begin
        y[1] <= ((load & x[1]) | (~load & 1'b0));
        y[2] <= ((load & x[2]) | (~load & y[1]));
        y[3] <= ((load & x[3]) | (~load & y[2]));
        y[4] <= ((load & x[4]) | (~load & y[3]));
end
endmodule
```

รูปที่ 6.29 โมดูลเชิงพฤติกรรมของวงจร 4-bit parallel-in serial-out shift register โดยใช้คำสั่งที่เป็น non-blocking

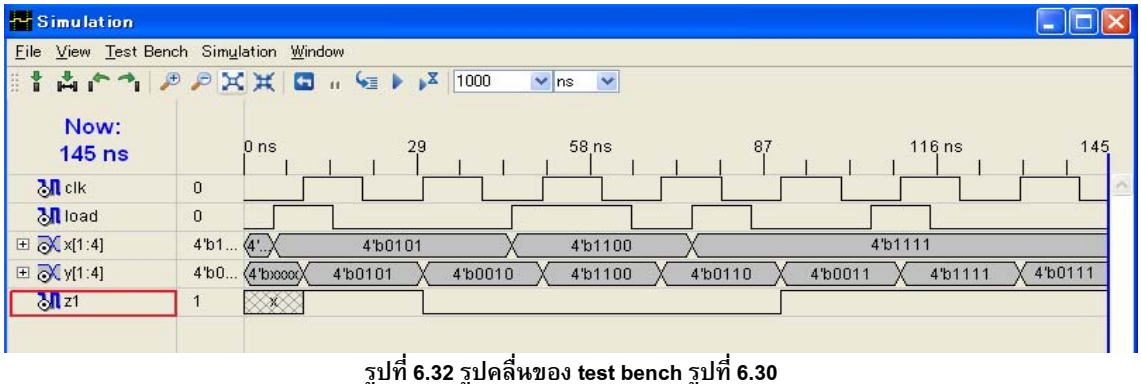
```
//test bench for piso shift register
module piso_bh_tb_v;
reg   clk, load;
reg   [1:4] x;
wire z1;
// Instantiate the Unit Under Test (UUT)
piso_bh uut (.clk(clk), .load(load), .x(x), .z1(z1));
initial     //define clock
begin
        clk = 1'b0;
        forever
        #10 clk = ~clk;
end
initial
$monitor ("load = %b, x = %b, z1 = %b", load, x, z1);
```

```
//apply input vectors
initial
begin
#0      load = 1'b0;        x = 4'b0000;
#5      load = 1'b1;        x = 4'b0101;
#10     load = 1'b0;
#30     load = 1'b1;        x = 4'b1100;
#20     load = 1'b0;
#10     load = 1'b1;        x = 4'b1111;
#10     load = 1'b0;
#20     load = 1'b1;        x = 4'b1111;
#10     load = 1'b0;
#30     $stop;
end
endmodule
```

รูปที่ 6.30 Test bench สำหรับโมดูลของรูปที่ 6.29

```
load = 0, x = 0000, z1 = x
load = 1, x = 0101, z1 = x
load = 1, x = 0101, z1 = 1
load = 0, x = 0101, z1 = 1
load = 0, x = 0101, z1 = 0
load = 1, x = 1100, z1 = 0
load = 0, x = 1100, z1 = 0
load = 1, x = 1111, z1 = 0
load = 0, x = 1111, z1 = 0
load = 0, x = 1111, z1 = 1
load = 1, x = 1111, z1 = 1
load = 0, x = 1111, z1 = 1
```

รูปที่ 6.31 ผลลัพธ์ของ test bench รูปที่ 6.30



รูปที่ 6.32 รูปคลื่นของ test bench รูปที่ 6.30

6.2.5 Blocking/Non-blocking กับการสังเคราะห์วงจร

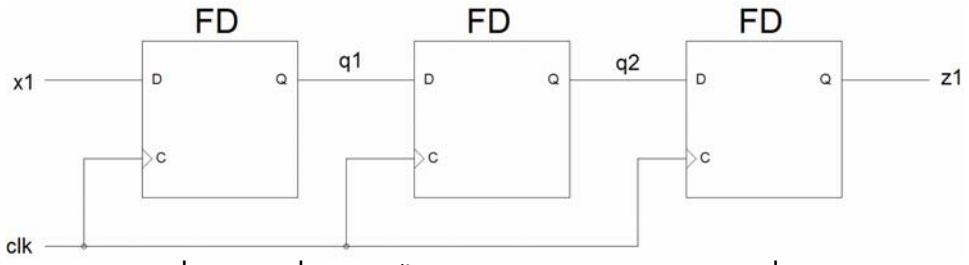
ตัวอย่างต่อไปนี้แสดงให้เห็นถึงความแตกต่างของวงจรที่สังเคราะห์ได้จากการกำหนดค่าค่าเชิงกระบวนคำสั่งที่เป็นแบบ blocking และ non-blocking

ตัวอย่างที่ 6.10 ออกแบบวงจร Flip-flop based digital delay line โมเดลเชิงพฤติกรรมของการกำหนดค่าค่าเชิงกระบวนคำสั่งที่เป็นแบบ blocking และ non-blocking แสดงดังรูปที่ 6.33 วงจรที่สังเคราะห์ได้และผลการจำลองการทำงานแสดงดังรูปที่ 6.34-6.37 สามารถแสดงให้เห็นความแตกต่างระหว่าง blocking และ non-blocking วงจรที่สังเคราะห์ได้จากคำสั่ง non-blocking เป็นวงจร shift register ซึ่งสามารถทำงานเป็น Flip-flop based digital delay line ได้ 3 สเตจ ที่ทุกๆขอบขาขึ้นของสัญญาณนาฬิกา clk สัญญาณ q1 q2 และ z1 จะรับค่าเดิมของ in q1 และ q2 ตามลำดับมาเก็บพร้อมๆกัน ในขณะที่วงจรที่สังเคราะห์ได้จากคำสั่ง blocking เป็นวงจรรีจิสเตอร์ D flip-flop เพียงสเตจเดียว สัญญาณ q1 และ q2 จะถูกลบออกไปในขั้นตอนการสังเคราะห์วงจร

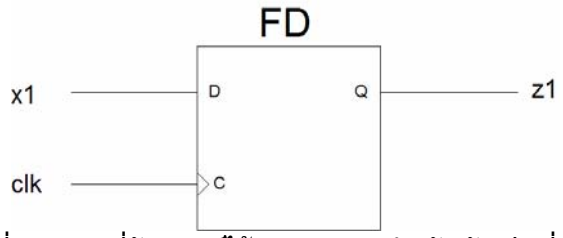
non-blocking	blocking

module nonblocking(x1, clk, z1);
input x1, clk;
output z1;
reg q1, q2, z1;
 	
always @(**posedge** clk)
begin
q1 <= x1;
q2 <= q1;
z1 <= q2;
end
endmodule
	module blocking(x1, clk, z1);
	input x1, clk;
	output z1;
	reg q1, q2, z1;
	
	always @(**posedge** clk)
	begin
	q1 = x1;
	q2 = q1;
	z1 = q2;
	end
	endmodule

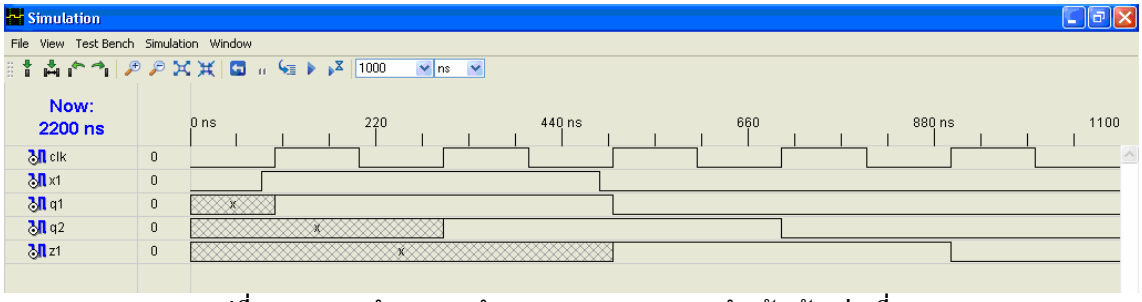
รูปที่ 6.33 โมดูล Verilog ของ non-blocking และ blocking สำหรับ ตัวอย่างที่ 6.10



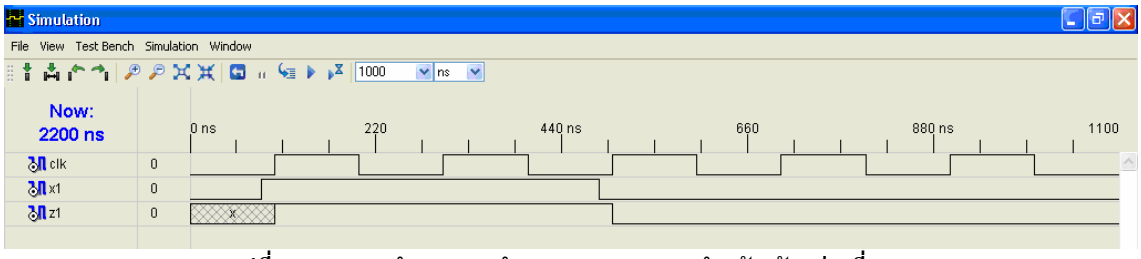
รูปที่ 6.34 วงจรที่สังเคราะห์ได้จาก non-blocking สำหรับ ตัวอย่างที่ 6.10



รูปที่ 6.35 วงจรที่สังเคราะห์ได้จาก blocking สำหรับ ตัวอย่างที่ 6.10



รูปที่ 6.36 ผลการจำลองการทำงานของ non-blocking สำหรับ ตัวอย่างที่ 6.10



รูปที่ 6.37 ผลการจำลองการทำงานของ blocking สำหรับ ตัวอย่างที่ 6.10

1.22 คำสั่งแบบเงื่อนไข

คำสั่งเงื่อนไข (conditional statement) ทำการสลับการไหลของสายงานภายใต้เงื่อนไขโดยมีค่าบูลลีนของนิพจน์เป็นตัวกำหนดทางเลือก เงื่อนไขที่เป็นจริงมีค่าเท่ากับลอจิก 1 เงื่อนไขที่เป็นเท็จมีค่าเท่ากับลอจิก 0 **x** หรือ **z** มีการใช้คำหลัก if และ else สำหรับเขียนคำสั่งเงื่อนไขได้ 3 แบบดังนี้

```
if (expression) statement1;          // ถ้านิพจน์ expression เป็นจริง statement1 จะถูกดำเนินการ

if (expression) statement1;          // ถ้านิพจน์ expression เป็นจริง statement1 จะถูกดำเนินการ
else statement2;                     // ไม่เช่นนั้น (เป็นเท็จ) statement2 จะถูกดำเนินการ

if (expression1) statement1;         //ถ้านิพจน์ expression1 เป็นจริง statement1 จะถูกดำเนินการ
else if (expression2) statement2;    //ไม่เช่นนั้นถ้า expression2 เป็นจริง statement2
...                                  // จะถูกดำเนินการ ...เงื่อนไขซ้อนๆกัน
else default statement;              //ไม่เช่นนั้น  statement จะถูกดำเนินการ
```

ตัวอย่างที่ 6.11 ออกแบบวงจร SOP (sum-of-products) สำหรับเกท AND และ OR ดังโมดูลเชิงพฤติกรรมในรูปที่ 6.38 เอาท์พุท z1 เป็น 1 เมื่อ x1x2=11 หรือ x3x4=11 รูปที่ 6.39 แสดง test bench ของวงจรดังกล่าว พร้อมด้วยผลการจำลองการทำงานดังรูปที่ 6.40 และ 6.41

```
//sum-of-products equation using if-else
module sop_if_else(x1, x2, x3, x4, z1);
input    x1, x2, x3, x4;
output reg    z1;
```

```
always @ (x1 or x2 or x3 or x4)
begin
        if ((x1 && x2) || (x3 && x4))
                z1 = #2 1;
        else        z1 = #2 0;
end
endmodule
```

รูปที่ 6.38 โมดูลเชิงพฤติกรรมของวงจรตัวอย่างที่ 6.11

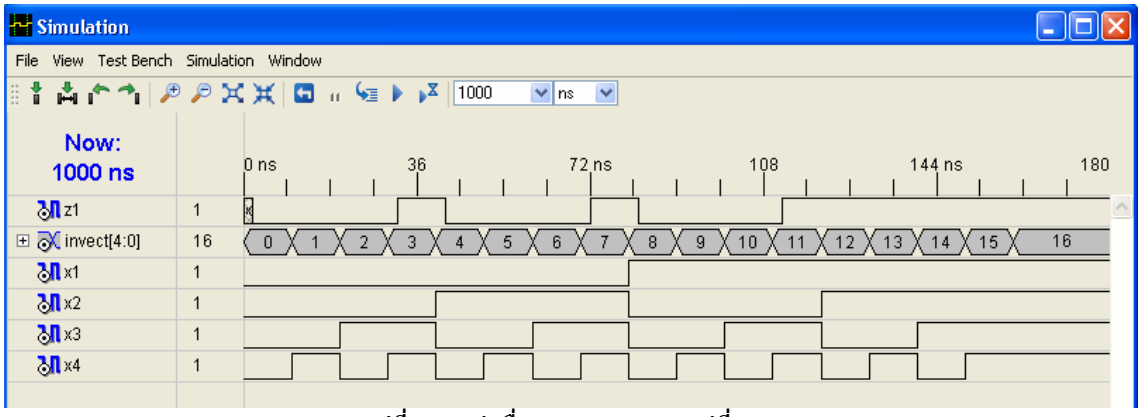
```
//test bench for sop_if_else
module sop_if_else_tb_v;
reg     x1, x2, x3, x4;
wire z1;

// Instantiate the Unit Under Test (UUT)
            sop_if_else uut (.x1(x1), .x2(x2),
                        .x3(x3), .x4(x4), .z1(z1));
```

```
//apply input vectors and display variables
initial
begin: apply_stimulus
   reg [4:0] invect;
   for (invect=0; invect<16; invect=invect+1)
      begin
            {x1, x2, x3, x4} = invect [4:0];
            #10 $display ("x1 x2 x3 x4 = %b, z1
= %b", {x1, x2, x3, x4}, z1);
      end
end
endmodule
```

รูปที่ 6.39 Test bench สำหรับโมดูลของรูปที่ 6.38

x1 x2 x3 x4 = 0000, z1 = 0	x1 x2 x3 x4 = 0110, z1 = 0	x1 x2 x3 x4 = 1011, z1 = 1
x1 x2 x3 x4 = 0001, z1 = 0	x1 x2 x3 x4 = 0111, z1 = 1	x1 x2 x3 x4 = 1100, z1 = 1
x1 x2 x3 x4 = 0010, z1 = 0	x1 x2 x3 x4 = 1000, z1 = 0	x1 x2 x3 x4 = 1101, z1 = 1
x1 x2 x3 x4 = 0011, z1 = 1	x1 x2 x3 x4 = 1001, z1 = 0	x1 x2 x3 x4 = 1110, z1 = 1
x1 x2 x3 x4 = 0100, z1 = 0	x1 x2 x3 x4 = 1010, z1 = 0	x1 x2 x3 x4 = 1111, z1 = 1
x1 x2 x3 x4 = 0101, z1 = 0		

รูปที่ 6.40 ผลลัพธ์ของ test bench รูปที่ 6.39



รูปที่ 6.41 รูปคลื่นของ test bench รูปที่ 6.39

ตัวอย่างที่ 6.12 ออกแบบวงจร modulo-16 ด้วยโมเดลเชิงพฤติกรรมดังรูปที่ 6.42 ลำดับของการนับถูกกำหนดโดยตัวดำเนินการ modulus ได้เป็นลำดับ 0000, 0001, 0010, 0011, …, 1111, 0000 รูปที่ 6.43 แสดง test bench ของวงจร พร้อมด้วยผลการจำลองการทำงานดังรูปที่ 6.44 และ 6.45

```
//behavior modulo-16 counter
module mod_16(clk, rst_n, count);
input       clk, rst_n;
output reg     [3:0] count;
```

```
always @(posedge clk or negedge rst_n)
begin
            if (rst_n==0)
                        count <= 4'b0000;
            else
                        count <= (count + 1) % 16;
end
endmodule
```

รูปที่ 6.42 โมดูลเชิงพฤติกรรมของวงจร 4-bit parallel-in serial-out shift register โดยใช้คำสั่งที่เป็น non-blocking

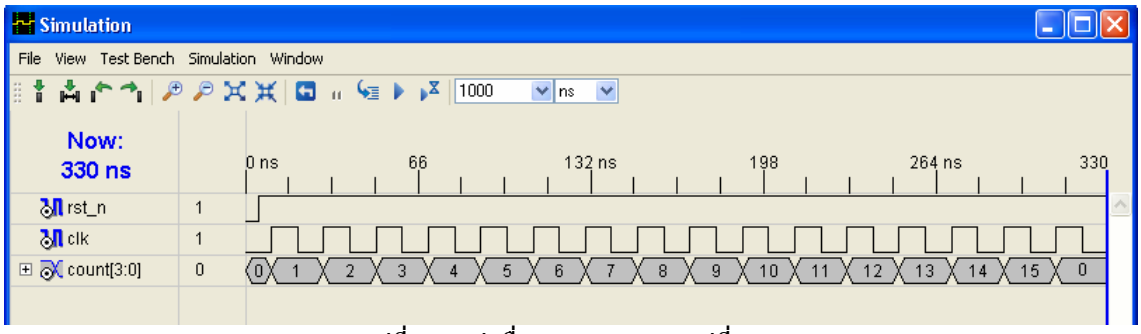
```
//test bench for modulo-16 counter
module mod_16_tb_v;
reg     clk, rst_n;
wire    [3:0] count;

// Instantiate the Unit Under Test (UUT)
mod_16 uut (.clk(clk), .rst_n(rst_n), .count(count));

//display count
initial
$monitor ("count = %b", count);

//define reset
initial
begin
    #0    rst_n = 1'b0;        //assert reset
    #5    rst_n = 1'b1;        //deassert reset
end
```

```
//define clock
initial
begin
    #0    clk = 1'b0;
    forever
    #10   clk = ~clk;
end

//define length of simulation
initial
begin
        #330 $stop;
end
endmodule
```

รูปที่ 6.43 Test bench สำหรับโมดูลของรูปที่ 6.42

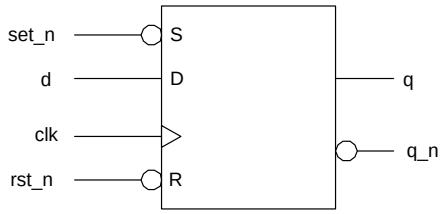
```
count = 0000
count = 0001
count = 0010
count = 0011
count = 0100
count = 0101
count = 0110
count = 0111
count = 1000
count = 1001
count = 1010
count = 1011
count = 1100
count = 1101
count = 1110
count = 1111
count = 0000
```

รูปที่ 6.44 ผลลัพธ์ของ test bench รูปที่ 6.43



รูปที่ 6.45 รูปคลื่นของ test bench รูปที่ 6.43

ตัวอย่างที่ 6.13 ออกแบบวงจร D flip-flop ซึ่งมีพอร์ทตามรูปที่ 6.46 โดยใช้โมเดลเชิงพฤติกรรมและคำสั่งเงื่อนไขได้ดังรูปที่ 6.47 พร้อมด้วย test bench ของวงจร และผลการจำลองการทำงานดังรูปที่ 6.48 6.49 และ 6.50 ตามลำดับ



รูปที่ 6.46 บล็อกไดอะแกรมของ D flip-flop สำหรับตัวอย่างที่ 6.13

```
// behavioral D flip-flop
module d_ff_bh(rst_n, clk, d, q, q_n);
input     rst_n, clk, d;
output    q, q_n;
reg    q;

assign q_n = ~q;
```

```
always @(negedge rst_n or posedge clk)
begin
        if (rst_n == 0)
                q <= 1'b0;
        else
                q <= d;
end
endmodule
```

รูปที่ 6.47 โมดูลเชิงพฤติกรรมของวงจร D flip-flop สำหรับตัวอย่างที่ 6.13

```
//test bench for d_ff_bh
module d_ff_bh_tb_v;
reg rst_n, clk, d;
wire q, q_n;

// Instantiate the Unit Under Test (UUT)
d_ff_bh uut (.rst_n(rst_n), .clk(clk),
                .d(d), .q(q), .q_n(q_n));

initial
$monitor ("rst_n=%b, clk=%b, d=%b, q=%b,
q_n=%b", rst_n, clk, d, q, q_n);

//define clock
initial
begin
        #0      clk = 1'b0;
        forever
        #10     clk = ~clk;
end
```

```

initial
begin
        #0      rst_n = 1'b0; d = 1'b0;
        #5      rst_n = 1'b1;

        #10     d = 1'b1;
        #10     d = 1'b1;
        #10     d = 1'b0;
        #10     d = 1'b1;
        #10     d = 1'b0;
        #10     d = 1'b0;
        #10     d = 1'b0;
        #10     $stop;
end
endmodule
```

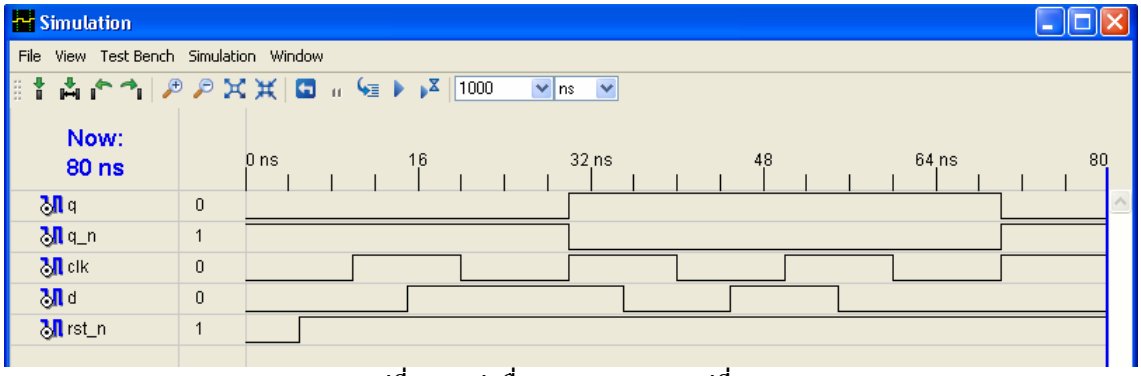
รูปที่ 6.48 Test bench สำหรับโมดูลของรูปที่ 6.47

```
rst_n=0, clk=0, d=0, q=0, q_n=1
rst_n=1, clk=0, d=0, q=0, q_n=1
rst_n=1, clk=1, d=0, q=0, q_n=1
rst_n=1, clk=1, d=1, q=0, q_n=1
rst_n=1, clk=0, d=1, q=0, q_n=1
```

```
rst_n=1, clk=1, d=1, q=1, q_n=0
rst_n=1, clk=1, d=0, q=1, q_n=0
rst_n=1, clk=0, d=0, q=1, q_n=0
rst_n=1, clk=0, d=1, q=1, q_n=0
rst_n=1, clk=1, d=1, q=1, q_n=0
```

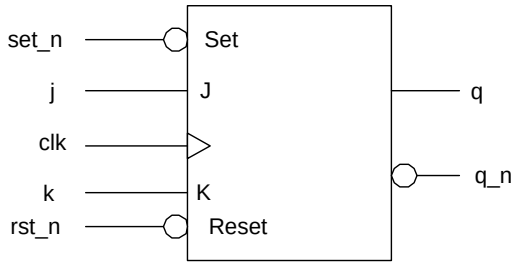
```
rst_n=1, clk=1, d=0, q=1, q_n=0
rst_n=1, clk=0, d=0, q=1, q_n=0
rst_n=1, clk=1, d=0, q=0, q_n=1
rst_n=1, clk=0, d=0, q=0, q_n=1
```

รูปที่ 6.49 ผลลัพธ์ของ test bench รูปที่ 6.48



รูปที่ 6.50 รูปคลื่นของ test bench รูปที่ 6.48

ตัวอย่างที่ 6.14 ออกแบบวงจร JK flip-flop ซึ่งมีพอร์ทตามรูปที่ 6.51 โดยใช้โมเดลเชิงพฤติกรรมและคำสั่งเงื่อนไขได้ดังรูปที่ 6.52 พร้อมด้วย test bench ของวงจร และผลการจำลองการทำงานดังรูปที่ 6.53 6.54 และ 6.55 ตามลำดับ



รูปที่ 6.51 บล็อกไดอะแกรมของ JK flip-flop สำหรับตัวอย่างที่ 6.14

```
//behavioral JK flip-flop
module jkff_bh(clk, j, k, set_n, rst_n, q, q_n);
input       clk, j, k, set_n, rst_n;
output reg     q, q_n;

initial     q = 1'b0;

always @(posedge clk or negedge rst_n or
negedge set_n)
begin
            if (~rst_n)
                begin
                        q <= 1'b0;
                        q_n <= 1'b1;
                end
            else if (~set_n)
                begin
                        q <= 1'b1;
                        q_n <= 1'b0;
                end
            else if (j==1'b0 && k==1'b0)
                begin     q <= q;
                          q_n <= q_n;
                end
            else if (j==1'b0 && k==1'b1)
                begin
                        q <= 1'b0;
                        q_n <= 1'b1;
                end
            else if (j==1'b1 && k==1'b0)
                begin
                        q <= 1'b1;
                        q_n <= 1'b0;
                end
            else if (j==1'b1 && k==1'b1)
                begin
                        q <= q_n;
                        q_n <= q;
                end
end
endmodule
```

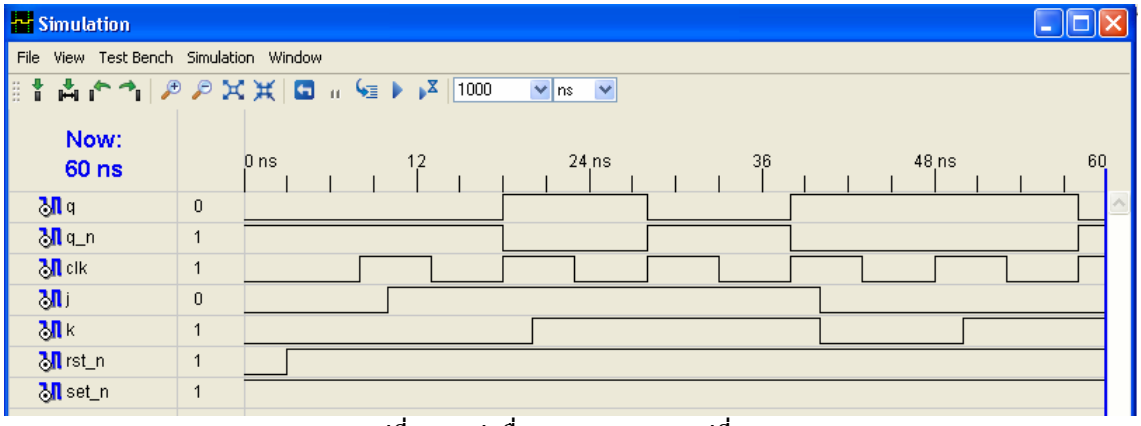
รูปที่ 6.52 โมดูลเชิงพฤติกรรมของวงจร JK flip-flop สำหรับตัวอย่างที่ 6.14

```
// test bench for JK flip-flop
module jkff_bh_tb_v;
reg clk, j, k, set_n, rst_n;
wire q, q_n;
jkff_bh uut (.clk(clk), .j(j), .k(k), .set_n(set_n),
      .rst_n(rst_n), .q(q), .q_n(q_n));
//display outputs at simulation time
initial
$monitor ($time, "  q = %b, q_n = %b", q, q_n);

initial
begin
            set_n = 1'b1; rst_n = 1'b0;
            j = 1'b0; k = 1'b0; clk = 1'b0;
            #3 rst_n = 1'b1;
            forever  #5 clk = ~clk;
end
initial begin
#10         j = 1'b1; k = 1'b0;    //set
#10         j = 1'b1; k = 1'b1;    //toggle reset
#10         j = 1'b1; k = 1'b1;    //togle set
#10         j = 1'b0; k = 1'b0;    //no change (set)
#10         j = 1'b0; k = 1'b1;    //reset
#10         $stop;
end
endmodule
```

รูปที่ 6.53 Test bench สำหรับโมดูลของรูปที่ 6.52

```
 0        q = 0, q_n = 1
18        q = 1, q_n = 0
28        q = 0, q_n = 1
38        q = 1, q_n = 0
58        q = 0, q_n = 1
```

รูปที่ 6.54 ผลลัพธ์ของ test bench รูปที่ 6.53



รูปที่ 6.55 รูปคลื่นของ test bench รูปที่ 6.53

1.23 คำสั่ง case

คำสั่ง **case** เป็นคำสั่งเงื่อนไขแบบหลายทาง มีความเหมาะสมสำหรับการใช้แทนคำสั่ง **if...else if** ที่ซ้อนกันหลายๆชั้น ไวยากรณ์การใช้งานเป็นดังนี้

```
case (expression)
    case_item1 : procedural_statement1;
    case_item2 : procedural_statement2;
    case_item3 : procedural_statement3;
            …
    case_itemn : procedural_statementn;
    default : default_statement;
endcase
```

นิพจน์ (expression) ในวงเล็บถูกเปรียบเทียบกับทางเลือก (case item) แบบบิตต่อบิต ในแต่ละรอบของการทำงานมีเพียงทางเลือกเดียวที่เท่ากับทุกบิตในนิพจน์เท่านั้นที่จะถูกดำเนินการ ในกรณีไม่มีทางเลือกใดตรงกับนิพจน์เลย ทางเลือกโดยปริยาย (**default**) จะถูกดำเนินการ

ตัวอย่างที่ 6.15 ออกแบบวงจร 4-bit Gray code counter โดยใช้โมเดลเชิงพฤติกรรมและคำสั่ง **case** ได้ดังรูปที่ 6.56 พร้อมด้วย test bench ของวงจร และผลการจำลองการทำงานดังรูปที่ 6.57 6.58 และ 6.59 ตามลำดับ

```
//behavioral 4-bit Gray code counter
module gray4_case(clk, rst_n, count);
input clk, rst_n;
output reg [3:0] count;

reg [3:0] next_count; //internal register

//set next count
always @ (posedge clk or negedge rst_n)
begin
        if (~rst_n)
                count       <= 4'b0000;
        else
                count <= next_count;
end
```

```
//determine next count
always @(count)
begin
   case (count)
          4'b0000 : next_count = 4'b0001;
          4'b0001 : next_count = 4'b0011;
          4'b0011 : next_count = 4'b0010;
          4'b0010 : next_count = 4'b0110;
          4'b0110 : next_count = 4'b0111;
          4'b0111 : next_count = 4'b0101;
          4'b0101 : next_count = 4'b0100;
          4'b0100 : next_count = 4'b1100;
          4'b1100 : next_count = 4'b1101;
          4'b1101 : next_count = 4'b1111;
          4'b1111 : next_count = 4'b1110;
          4'b1110 : next_count = 4'b1010;
          4'b1010 : next_count = 4'b1011;
          4'b1011 : next_count = 4'b1001;
          4'b1001 : next_count = 4'b1000;
          4'b1000 : next_count = 4'b0000;
          default : next_count = 4'b0000;
   endcase
end
endmodule
```

รูปที่ 6.56 โมดูลเชิงพฤติกรรมของวงจร 4-bit Gray code counter

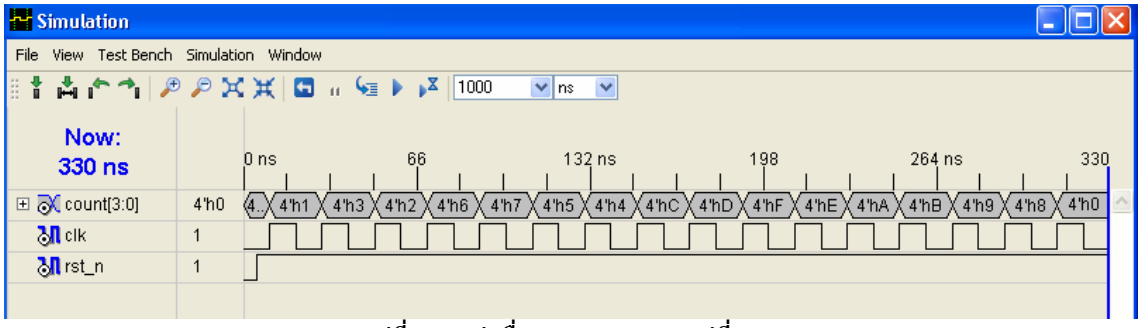
```
//test bench for 4-bit Gray code counter using case
module gray4_case_tb_v;
reg clk, rst_n;
wire [3:0] count;
// Instantiate the Unit Under Test (UUT)
gray4_case uut (.clk(clk), .rst_n(rst_n), .count(count));

initial
$monitor ("count = %b", count);
//define reset
initial
begin
        #0          rst_n = 1'b0;
        #5           rst_n = 1'b1;
end
```

```
//define clock
initial
begin
        #0          clk = 1'b0;
        forever
                #10 clk = ~clk;
end

//define length of simulation
initial
begin
        #330 $stop;
end
endmodule
```

รูปที่ 6.57 Test bench สำหรับโมดูลของรูปที่ 6.56

count = 0000	count = 0101	count = 1010
count = 0001	count = 0100	count = 1011
count = 0011	count = 1100	count = 1001
count = 0010	count = 1101	count = 1000
count = 0110	count = 1111	count = 0000
count = 0111	count = 1110	

รูปที่ 6.58 ผลลัพธ์ของ test bench รูปที่ 6.57



รูปที่ 6.59 รูปคลื่นของ test bench รูปที่ 6.57

ตัวอย่างที่ 6.16 ออกแบบวงจร 4-bit even-odd counter โดยใช้โมเดลเชิงพฤติกรรมและคำสั่ง **case** ได้ดังรูปที่ 6.60 พร้อมด้วย test bench ของวงจร และผลการจำลองการทำงานดังรูปที่ 6.61 6.62 และ 6.63 ตามลำดับ

```
//behavioral even-odd counter
module even_odd_case(clk, rst_n, count);
input clk, rst_n;
output reg [3:0] count;
reg [3:0] next_count; //internal register

//set next count
always @ (posedge clk or negedge rst_n)
begin
        if (~rst_n)
                count       <= 4'b0000;
        else
                count <= next_count;
end
```

```
//determine next count
always @ (count)
begin
    case (count)
        4'b0000 : next_count = 4'b0010;
        4'b0010 : next_count = 4'b0100;
        4'b0100 : next_count = 4'b0110;
        4'b0110 : next_count = 4'b1000;
        4'b1000 : next_count = 4'b1010;
        4'b1010 : next_count = 4'b1100;
        4'b1100 : next_count = 4'b1110;
        4'b1110 : next_count = 4'b0001;
        4'b0001 : next_count = 4'b0011;
        4'b0011 : next_count = 4'b0101;
        4'b0101 : next_count = 4'b0111;
        4'b0111 : next_count = 4'b1001;
        4'b1001 : next_count = 4'b1011;
        4'b1011 : next_count = 4'b1101;
        4'b1101 : next_count = 4'b1111;
        4'b1111 : next_count = 4'b0000;
        default : next_count = 4'b0000;
        endcase
end
endmodule
```

รูปที่ 6.60 โมดูลเชิงพฤติกรรมของวงจร 4-bit even-odd counter

```
// test bench for even-odd counter
module even_odd_case_tb_v;
reg clk, rst_n;
wire [3:0] count;
// Instantiate the Unit Under Test (UUT)
even_odd_case uut (.clk(clk), .rst_n(rst_n), .count(count));

initial
$monitor ("count = %b", count);

Initial      //define reset
begin
        #0          rst_n = 1'b0;
        #5          rst_n = 1'b1;
end
```

```
//define clock
initial
begin
        #0          clk = 1'b0;
        forever
                    #10 clk = ~clk;
end

//define length of simulation
initial
begin
        #330 $stop;
end

endmodule
```

รูปที่ 6.61 Test bench สำหรับโมดูลของรูปที่ 6.60

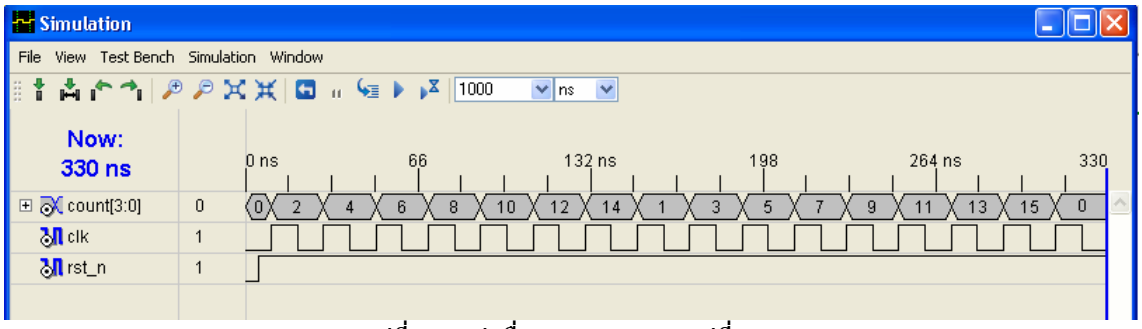
```
count = 0000
count = 0010
count = 0100
count = 0110
count = 1000
```

```
count = 1010
count = 1100
count = 1110
count = 0001
```

```
count = 0011
count = 0101
count = 0111
count = 1001
```

```
count = 1011
count = 1101
count = 1111
count = 0000
```

รูปที่ 6.62 ผลลัพธ์ของ test bench รูปที่ 6.61



รูปที่ 6.63 รูปคลื่นของ test bench รูปที่ 6.61

ตัวอย่างที่ 6.17 ออกแบบวงจร 4-function ALU โดยใช้โมเดลเชิงพฤติกรรมและคำสั่ง **case** ได้ดังรูปที่ 6.64 พร้อมด้วย test bench ของวงจร และผลการจำลองการทำงานดังรูปที่ 6.65 6.66 และ 6.67 ตามลำดับ

```
//behavioral 4-bit ALU
module alu4_bh(a, b, opcode, z);
input    [3:0] a, b;
input    [1:0] opcode;
output reg [8:0] z;

//define operation codes
parameter    addop = 2'b00,
             subop = 2'b01,
             mulop = 2'b10,
             div2op = 2'b11;
```

```
//perform operations
always @ (a or b or opcode)
begin
        case (opcode)
                addop: z = a + b;
                subop: z = a - b;
                mulop: z = a * b;
                div2op: z = a / 2;
        endcase
end
endmodule
```

รูปที่ 6.64 โมดูลเชิงพฤติกรรมของวงจร 4-function ALU

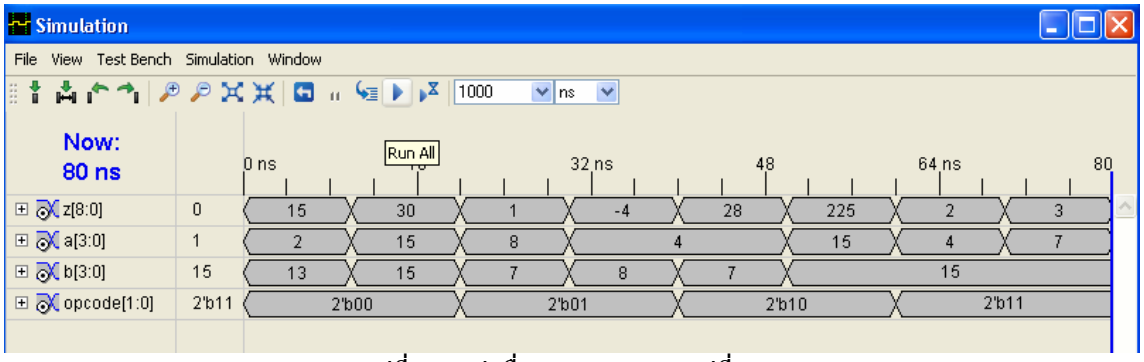
```
//test bench for 4-bit ALU
module alu4_bh_tb_v;
reg     [3:0] a, b;
reg     [1:0] opcode;
wire     [8:0] z;

// Instantiate the Unit Under Test (UUT)
alu4_bh uut (.a(a), .b(b),
                        .opcode(opcode), .z(z));

//display variables
initial
$monitor     ("a=%b,     b=%b,     opcode=%b,
result=%b", a, b, opcode, z);
```

```
//apply input vectors
initial
begin
// add operation
#0          a=4'b0010; b=4'b1101; opcode=2'b00;
#10         a=4'b1111; b=4'b1111;opcode=2'b00;

// subtract operation
#10         a=4'b1000; b=4'b0111; opcode=2'b01;
#10         a=4'b0100; b=4'b1000; opcode=2'b01;

//multiply operation
#10         a=4'b0100; b=4'b0111; opcode=2'b10;
#10         a=4'b1111; b=4'b1111; opcode=2'b10;

// divided by 2 operation
#10         a=4'b0100; opcode=2'b11;
#10         a=4'b0111; opcode=2'b11;
#10         a=4'b0001; opcode=2'b11;
end
endmodule
```

รูปที่ 6.65 Test bench สำหรับโมดูลของรูปที่ 6.64

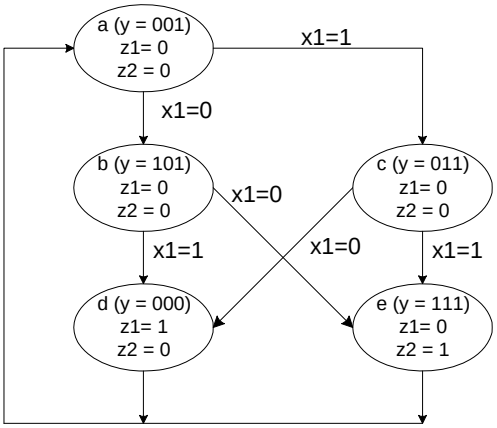
```
a=0010, b=1101, opcode=00, result=000001111
a=1111, b=1111, opcode=00, result=000011110
a=1000, b=0111, opcode=01, result=000000001
a=0100, b=1000, opcode=01, result=111111100
a=0100, b=0111, opcode=10, result=000011100
a=1111, b=1111, opcode=10, result=011100001
a=0100, b=1111, opcode=11, result=000000010
a=0111, b=1111, opcode=11, result=000000011
a=0001, b=1111, opcode=11, result=000000000
```

รูปที่ 6.66 ผลลัพธ์ของ test bench รูปที่ 6.65



รูปที่ 6.67 รูปคลื่นของ test bench รูปที่ 6.65

ตัวอย่างที่ 6.18 ออกแบบวงจร Moore finite state machine (FSM) ตามรูปแผนภาพสถานะ (state diagram) ดังรูปที่ 6.68 โดยหลักการของ Moore เอาท์พุทจะขึ้นกับสเตตแต่เพียงอย่างเดียว และสเตตถัดไป (next state) ขึ้นกับสเตตปัจจุบัน (present state) และอินพุท โมดูลเชิงพฤติกรรมสำหรับแผนภาพดังกล่าวถูกแสดงดังรูปที่ 6.69 สังเกตเห็นว่ามีการใช้บล็อก always สำหรับการคำนวณเอาท์พุทซึ่งมีเพียงสเตตเท่านั้นที่เป็นตัวกำหนดเอาท์พุท และมีการใช้บล็อก always แยกต่างหากสำหรับการคำนวณสเตตถัดไปซึ่งมีทั้งสเตตปัจจุบันและอินพุทเป็นตัวกำหนดสเตตถัดไป test bench ของวงจร และผลการจำลองการทำงานดังรูปที่ 6.70 6.71 และ 6.72 ตามลำดับ



รูปที่ 6.68 แผนภาพสถานะของ Moore finite state machine (FSM) สำหรับตัวอย่างที่ 6.18

```
//behavioral Moore FSM1
module moore_fsm1(clk, rst_n, x1, y, z1, z2);
input     clk, rst_n, x1;
output reg [2:0] y;
output reg z1, z2;

reg [2:0] next_state;

//assign state codes
parameter          state_a = 3'b001,
                   state_b = 3'b101,
                   state_c = 3'b011,
                   state_d = 3'b000,
                   state_e = 3'b111;
//set next state
always @ (posedge clk)
begin
          if (~rst_n)
                   y <= state_a;
          else
                   y <= next_state;
end
```

```
//determine outputs
always @ (y)
begin
          if (y == state_d) z1 = 1'b1;
          else  z1 = 1'b0;

          if (y == state_e) z2 = 1'b1;
          else  z2 = 1'b0;
end

//determine next state
always @ (y or x1)
begin
          case (y)
          state_a : if (x1) next_state = state_c;
                    else   next_state = state_b;
          state_b : if (x1) next_state = state_d;
                    else  next_state = state_e;
          state_c : if (x1) next_state = state_e;
                    else  next_state = state_d;
          state_d : next_state = state_a;
          state_e : next_state = state_a;
          default : next_state = state_a;
          endcase
end
endmodule
```

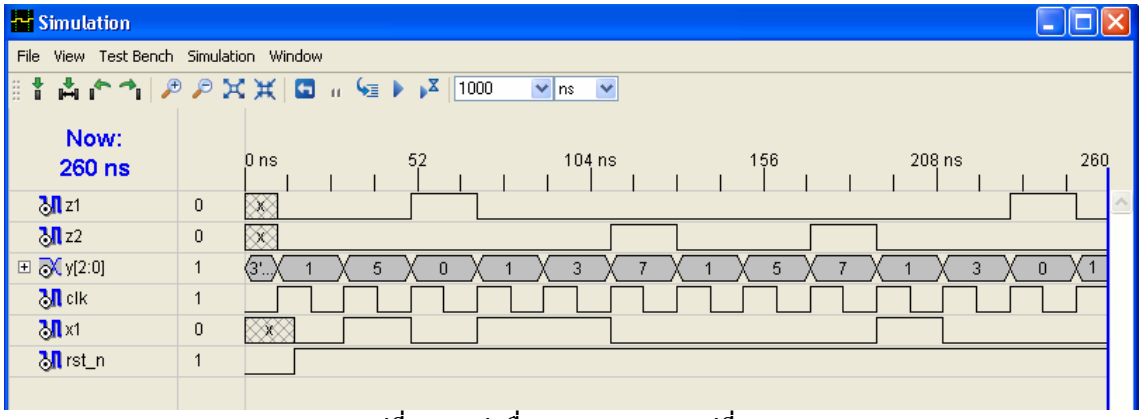
รูปที่ 6.69 โมดูลเชิงพฤติกรรมของวงจร Moore finite state machine (FSM) สำหรับตัวอย่างที่ 6.18

```
//test bench for Moore FSM1
module moore_fsm1_tb_v;
reg clk, rst_n, x1;
wire [2:0] y;
wire z1, z2;

// Instantiate the Unit Under Test (UUT)
moore_fsm1 uut (.clk(clk), .rst_n(rst_n),
            .x1(x1), .y(y), .z1(z1), .z2(z2));

initial
$monitor ("x1 = %b, state = %b, z1 = %b, z2
= %b", x1, y, z1, z2);

//define clock
initial
begin
            clk = 1'b0;
            forever
                        #10 clk = ~clk;
end

//define input sequence
initial
begin
            #0 rst_n = 1'b0;       //reset to state_a
            #15 rst_n = 1'b1;

            x1 = 1'b0;
            @ (posedge clk)     //go to state_b

            x1 = 1'b1;
            @ (posedge clk) //go to state_d; assert z1
            x1 = 1'b0;
            @ (posedge clk)     //go to state_a

            x1 = 1'b1;
            @ (posedge clk)     //go to state_c

            x1 = 1'b1;
            @ (posedge clk) //go to state_e; assert z2

            x1 = 1'b0;
            @ (posedge clk)     //go to state_a

            x1 = 1'b0;
            @ (posedge clk)     //go to state_b

            x1 = 1'b0;
            @ (posedge clk) //go to state_e; assert z2

            x1 = 1'b0;
            @ (posedge clk)     //go to state_a

            x1 = 1'b1;
            @ (posedge clk)     //go to state_c

            x1 = 1'b0;
            @ (posedge clk) //go to state_d; assert z1

            x1 = 1'b0;
            @ (posedge clk)     //go to state_a

            #10         $stop;
end
endmodule
```

รูปที่ 6.70 Test bench สำหรับโมดูลของรูปที่ 6.69

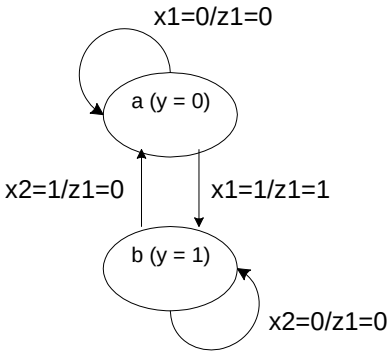
```
x1 = x, state = xxx, z1 = x, z2 = x
x1 = x, state = 001, z1 = 0, z2 = 0
x1 = 0, state = 001, z1 = 0, z2 = 0
x1 = 1, state = 101, z1 = 0, z2 = 0
x1 = 0, state = 000, z1 = 1, z2 = 0
x1 = 1, state = 001, z1 = 0, z2 = 0
x1 = 1, state = 011, z1 = 0, z2 = 0
x1 = 0, state = 111, z1 = 0, z2 = 1
x1 = 0, state = 001, z1 = 0, z2 = 0
x1 = 0, state = 101, z1 = 0, z2 = 0
x1 = 0, state = 111, z1 = 0, z2 = 1
x1 = 1, state = 001, z1 = 0, z2 = 0
x1 = 0, state = 011, z1 = 0, z2 = 0
x1 = 0, state = 000, z1 = 1, z2 = 0
x1 = 0, state = 001, z1 = 0, z2 = 0
```

รูปที่ 6.71 ผลลัพธ์ของ test bench รูปที่ 6.70



รูปที่ 6.72 รูปคลื่นของ test bench รูปที่ 6.70

ตัวอย่างที่ 6.19 ออกแบบวงจร Mealy finite state machine (FSM) ตามรูปแผนภาพสถานะ (state diagram) ดังรูปที่ 6.73 โดยหลักการของ Mealy ทั้งเอาท์พุทและสเตตถัดไป (next state)ขึ้นกับสเตตปัจจุบัน (present state) และอินพุท โมดูลเชิงพฤติกรรมสำหรับแผนภาพดังกล่าวถูกแสดงดังรูปที่ 6.74 สังเกตเห็นว่ามีการใช้บล็อก always สำหรับการคำนวณเอาท์พุทซึ่งมีทั้งสเตตและอินพุทเป็นตัวกำหนดเอาท์พุท และมีการใช้บล็อก always แยกต่างหากสำหรับการคำนวณสเตตถัดไปซึ่งมีทั้งสเตตปัจจุบันและอินพุทเป็นตัวกำหนดสเตตถัดไป เช่นกัน test bench ของวงจร และผลการจำลองการทำงานดังรูปที่ 6.75 6.76 และ 6.77 ตามลำดับ



รูปที่ 6.73 แผนภาพสถานะของ Mealy finite state machine (FSM) สำหรับตัวอย่างที่ 6.19

```
//behavioral Mealy FSM
module mealy_fsm1(clk, rst_n, x1, x2, y, z1);
input clk, rst_n, x1, x2;
output reg z1, y;
reg next_state;

//define state codes
parameter state_a = 1'b0, state_b = 1'b1;

//set next state
always @ (posedge clk)
begin
        if (~rst_n)
                y <= state_a;
        else
                y <= next_state;
end
```

```
//determine outputs
always @ (x1)
begin
        if ((y == state_a) && (x1 == 1'b1))
                z1 = 1'b1;
        else
                z1 = 1'b0;
end

//determine outputs
always @ (y or x1 or x2)
begin
        case (y)
        state_a:  if (x1) next_state = state_b;
                  else next_state = state_a;
        state_b:   if (x2) next_state = state_a;
                  else next_state = state_b;
        default:  next_state = state_a;
        endcase
end
endmodule
```

รูปที่ 6.74 โมดูลเชิงพฤติกรรมของวงจร Mealy finite state machine (FSM) สำหรับตัวอย่างที่ 6.19

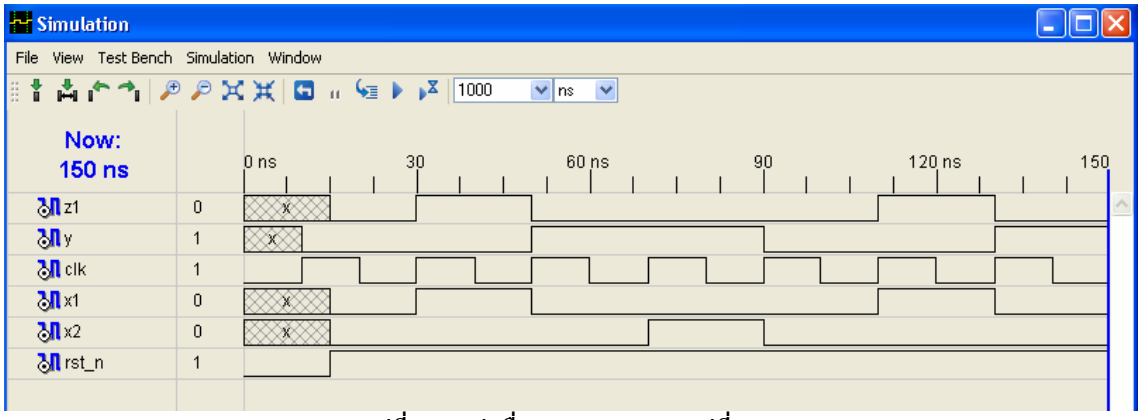
```
//test bench for Mealy FSM
module mealy_fsm1_tb_v;
reg clk, rst_n, x1, x2;
wire y, z1;

// Instantiate the Unit Under Test (UUT)
mealy_fsm1 uut (.clk(clk), .rst_n(rst_n),
        .x1(x1), .x2(x2), .y(y), .z1(z1));

//display variables
initial
$monitor ("x1=%b, x2=%b, state=%b, z1=%b",
x1, x2, y, z1);

//define clock
initial
begin
        clk = 1'b0;
        forever
                #10 clk = ~clk;
end

//define input sequence
initial
begin
#0      rst_n = 1'b0;       //reset to state_a
#15     rst_n = 1'b1;
```

```
x1 = 1'b0; x2 = 1'b0;
@ (posedge clk)             //go to state_a

x1 = 1'b1; x2 = 1'b0;
@ (posedge clk)    //assert z1; go to state_b

x1 = 1'b0; x2 = 1'b0;
@ (posedge clk)             //go to state_b

x1 = 1'b0; x2 = 1'b1;
@ (posedge clk)             //go to state_a

x1 = 1'b0; x2 = 1'b0;
@ (posedge clk)             //go to state_a

x1 = 1'b1; x2 = 1'b0;
@ (posedge clk)    //assert z1; go to state_b

x1 = 1'b0; x2 = 1'b0;
@ (posedge clk)             //go to state_a

#10     $stop;
end
endmodule
```

รูปที่ 6.75 Test bench สำหรับโมดูลของรูปที่ 6.74

x1=x, x2=x, state=x, z1=x x1=x, x2=x, state=0, z1=x x1=0, x2=0, state=0, z1=0 x1=1, x2=0, state=0, z1=1 x1=0, x2=0, state=1, z1=0	x1=0, x2=1, state=1, z1=0 x1=0, x2=0, state=0, z1=0 x1=1, x2=0, state=0, z1=1 x1=0, x2=0, state=1, z1=0

รูปที่ 6.76 ผลลัพธ์ของ test bench รูปที่ 6.75



รูปที่ 6.77 รูปคลื่นของ test bench รูปที่ 6.75

1.24 คำสั่ง loop

6.5.1 For loop

6.5.2 While loop

6.5.3 Repeat loop

6.5.4 Forever loop

1.25 คำสั่งบล็อก

6.6.1 บล็อกเชิงลำดับ

6.6.2 บล็อกขนาน

1.26 การกำหนดค่าต่อเนื่องเชิงกระบวนคำสั่ง

6.7.1 Assign … Deassign

6.7.2 Force … Release

ภาคผนวก

VHDL / Verilog

1. entity / module

VHDL	Verilog
library ieee; **use ieee.std_logic_1164.all;** **entity** entity_name **is** **port** (port_name_1 : mode type; …. port_name_n : mode type); **end** entity_name; **architecture** arch_name **of** entity_name **is** architecture declarations **begin** architecture statements **end** arch_name;	**`timescale** 1ns / 1ns **module** module_name (port lists); **input** <[bit_width]> in_1,…, in_n ; **output** <[bit_width]> out_1, …, out_n; **inout** <[bit_width]> inout_1, …, inout_n; … **endmodule**

Ex. AND-OR-INVERTER

VHDL	Verilog

library ieee;
use ieee.std_logic_1164.all;
entity aoi **is**
port (
a, b, c : **in std_logic**;
x : **out std_logic**);
end aoi;
 	
architecture rtl_aoi **of** aoi **is**
begin
x **<= not** ((a **and** b) **or** c);
end rtl_aoi;
	`timescale 1ns / 1ns
	module aoi (x, a, b, c);
	
	input a, b, c;
	output x;
	
	assign x = ~((a & b) | c);
	
	endmodule

2. Object, type

VHDL	Verilog
signal signal_name : type;	**reg** <[bit_width]> register_name;
variable variable_name : type;	**wire** <[bit_width]> wire_name;
constant constant_name : type;	**parameter** <[bit_width]> parameter_name;

Ex.

VHDL	Verilog
signal s : **std_logic**;	**reg** s;
variable v: **std_logic_vector** (7 **downto** 0);	**wire** [7:0] v;
constant c: **std_logic** := '1';	**parameter** p = 1;

3. Operators

VHDL		Verilog	
Arithmetic		**Arithmetic**	
+	Add	**+**	Add
-	Subtract	**-**	Subtract
*	Multiply	*	Multiply
/	Divide	/	Divide
**	Power	**%**	Modulus
abs	Absolute		
Logical		**Bitwise**	
and	AND	**~**	Bitwise inverter
or	OR	**&**	Bitwise AND
not	NOT	**|**	Bitwise OR
nand	NAND	**^**	Bitwise EX-OR
nor	NOR	**~^**	Bitwise EX-NOR
xor	EX-OR		
		Reduction	
		&	Reduction AND
		~&	Reduction NAND
		|	Reduction OR
		~|	Reduction NOR
		^	Reduction EX-OR
		~^	Reduction EX-NOR

		Logical	
		!	Logical negation
		&&	Logical AND
		||	Logical OR
Equality		**Equality**	
=	Equality	**==**	Logical equality
/=	Inequality	**!=**	Logical inequality
		===	Case equality
		!==	Case inequality
Relational		**Relational**	
>	Greater than	**>**	Greater than
<	Less than	**<**	Less than
>=	Greater than or equal	**>=**	Greater than or equal
<=	Less than or equal	**<=**	Less than or equal
		Shift	
		<<	Left shift
		>>	Right shift
		Conditional	
		?:	Conditional
Concatenation		**Concatenation**	
&	Concatenation	**{,}**	Concatenation
		Replication	
		{{ }}	Replication

4. Concurrent signal assignment / assign statement

VHDL	Verilog
signal_name <= expression <**after** time>;	**assign** <#time> wire_name = expression;

Ex.

VHDL	Verilog
x <= a **and** b **after** 10 ns;	**`timescale** 1 ns / 1 ns …. **assign** #10 x = a **&** b;

5. Conditional signal assignment / conditional assign statement

VHDL	Verilog
signal_name <= expression_1 <**after** time_1> **when** condition **else** expression_2 <**after** time_2>;	**assign** <#time> wire_name = condition**?** expression_1 **:** expression_2;

6. process / always , initial

VHDL	Verilog
<label:> **process** <(sensitivity list)> {Declarations} **begin** {Sequential statements} **end process**;	**always** <**@**(sensitivity list)> **begin** <:label> {Declaration} {Sequential statements} **end** **inital** <**@**(sensitivity list)> **begin** <:label> {Declaration} {Sequential statements} **end**
Inside process	**Inside always, initial blocks**
Sequential signal assignment Variable assignment **if** **case** **for loop, while loop, loop** **wait on, wait for, wait until** etc.	Procedural assignment **if** **case** **for, while, forever** **wait** etc.

Ex. Combinational circuit: multiplexer

VHDL	Verilog

library ieee;
use ieee.std_logic_1164.all;
 	
entity mux **is**
  port (
  a, b, s : **in std_logic;**
  x : **out std_logic**);
end mux;
 	
architecture arch_mux **of** mux **is**
  signal w1 : **std_logic;**
begin
  process (a, b, s)
  variable tmp: **std_logic;**
  begin
  if (s = '1') **then** tmp := a;
  else tmp := b;
  end if;
  w1 <= tmp;
  end process;
 	
  process (w1) **begin**
  x <= **not** w1;
  end process;
end arch_mux;
	`timescale 1 ns / 1 ns
	
	module mux (x, a, b, s);
	  input a, b, s;
	  output x;
	
	  reg x, w1, tmp;
	
	  always @(a **or** b **or** s) **begin**
	  if (s == 1'b1) tmp = a;
	  else tmp = b;
	  w1 <= tmp;
	  end
	
	  always @(w1) **begin**
	  x <= ~w1;
	  end
	endmodule

Ex. Sequential circuit: flip-flop

VHDL	Verilog

library ieee;
use ieee.std_logic_1164.all;
 	
entity dff **is**
  port (
  d, clk, rst : in std_logic;
  q, xq : out std_logic);
end dff;
 	
architecture arch_dff **of** dff **is**
begin
  process (clk, rst) **begin**
  if (rst = '0') **then**
  q <= '0';
  xq <= '1';
  elsif (clk**'event and** clk = '1')
  then q<= d;
  xq <= **not** d;
  end if;
  end process;
end arch_dff;
	module dff (d, clk, rst, q, xq);
	input d, clk, rst;
	output q, xq;
	reg q, xq;
	
	always @(posedge clk **or negedge** rst) **begin**
	  if (rst == 1'b0) **begin**
	  q <= 1'b0;
	  xq <= 1'b1;
	  end else begin
	  q <= d;
	  xq <= ~d;
	  end
	end
	endmodule

7. Sequential signal assignment / procedural non-blocking assignment

VHDL	Verilog
signal_name **<=** expression **<after** time**>**;	register_name **<= <#**time**>** expression;

Ex. Swap a and b

VHDL	Verilog

process (a, b) **begin**
  a <= b;
  b <= a;
end process;
	always @(a **or** b) **begin**
	  a <= b;
	  b <= a;
	end

8. Variable assignment / procedural blocking assignment

VHDL	Verilog
variable_name **:=** expression **<after** time**>**;	register_name **= <#**time**>** expression;

Ex. Does not swap a and b

VHDL	Verilog

process (a, b) **begin**
a := b;
b := a;
end process;
	always @(a **or** b) **begin**
	a = b;
	b = a;
	end

9. if statement

VHDL	Verilog
if (condition_1) **then** statement_1 **elsif** (condition_2) **then** statement_2 **else** Statement_3 **end if;**	**if** (condition_1) statement_1 **else if** (condition_2) statement_2 **else** Statement_3

Ex.

VHDL	Verilog

if (cond_1 = '1') **then** x <= a;
elsif (cond_2 = '1') **then** x <= b;
else
x <= c;
end if;
	if (cond_1 == 1'b1) x <= a;
	else if (cond_2 == 1'b1) **then** x <= b;
	else
	x <= c;

10. case statement

VHDL	Verilog
case expression **is** **when** value_1 => statement_1 **when** value_2 => statement_2 … **when** others => statement_x **end case;**	**case** (expression) value_1 : statement_1 value_2 : statement_2 … **default** : statement_x **endcase**

Ex.

VHDL	Verilog

case sel **is**
  when “00” => x <= a;
  when “01” => x <= b;
  when “10” => x <= c;
  when “11” => x <= d;
  when others => x <= ‘X’;
end case;
	case (sel)
	  2’b00: x <= a;
	  2’b01: x <= b;
	  2’b10: x <= c;
	  2’b11: x <= d;
	  default: x <= 1’bx;
	endcase

11. for statement

VHDL	Verilog
for index **in** range **loop** Statement **end loop;**	**for** (expression_1; condition; expression_2) Statement

Ex.

VHDL	Verilog
for i **in** 0 **to** 7 **loop** x(i) <= y(7 - i); **end loop;**	**for** (i = 0; i <= 7; i = i+1) x[i] <= y[7 - i];

12. while statement

VHDL	Verilog

while condition **loop**
  Statement
end loop;
	while (condition)
	  Statement

Ex.

VHDL	Verilog

i := 0;
while (i <= 7) **loop**
  x(i) <= y(7-i);
  i := i + 1;
end loop;
	i = 0;
	while (i <= 7) **begin**
	  x[i] <= y[7-i];
	  i = i + 1;
	end

13. loop / forever

VHDL	Verilog
loop Statement **end loop;**	**forever** statement

Ex.

VHDL	Verilog

i := 0;
loop
x(i) <= y(7-i);
i := i + 1;
exit when (i = 8);
end loop;
	i := 0;
	forever begin: forever_block
	x[i] <= y[7-i];
	i = i + 1;
	if (i == 8) **disable** forever_block;
	end

14. wait for / #

VHDL	Verilog
wait for time;	**#** time;

Ex.

VHDL	Verilog
wait for 100 ns;	**#** 100;

15. wait until / wait

VHDL	Verilog
wait until condition;	**wait** (condition);

Ex.

VHDL	Verilog
wait until c = '1';	**wait** (c == 1'b1);

16. wait on / @

VHDL	Verilog
wait on sensitivity list;	**@**(sensitivity list);

Ex.

VHDL	Verilog
wait on a, b, c;	**@**(a **or** b **or** c);

17. wait / $stop, $finish

VHDL	Verilog
wait; หยุด process แต่ไม่หยุด simulation	**$stop;** หยุด simulation **$finish;** จบ simulation

18. function

VHDL	Verilog
function function_name ({<object type> parameter_name {, parameter_name}: **<in>** type;} <object type> parameter_name {, parameter_name}: **<in>** type) **return** type **is** {declaration} **begin** {sequential statement} **return** expression; **end** <function_name>;	**function** <[bit_width]> function_name; {**input** <[bit_width]> parameter_name {, parameter_name};} {declaration} {sequential statement} **endfunction**

Ex.

VHDL	Verilog

function sign_extend
(a: std_logic_vector (3 downto 0))
return std_logic_vector **is**
begin
if (a(3) = '1') **then**
return ("1111" & a);
else
return ("0000" & a);
end if;
end sign_extend;
x <= sign_extend(a); // call function
	function [7:0] sign_extend;
	input [3:0] a;
	if (a[3]) sign_extend = {4'b1111, a};
	else sign_extend = {4'b0000, a};
	endfunction
	
	x <= sign_extend(a); // call function

19. procedure / task

VHDL	Verilog
procedure procedure_name ({<object_type> parameter_name {, parameter_name}: <mode> type;} <object_type> parameter_name {, parameter_name}: <mode> type) **is** {declaration} **begin** {sequential statement} **return;** **end** procedure_name;	**task** task_name; {**input** <[bit_width]> parameter_name {, parameter_name};} {**output** <[bit_width]> parameter_name {, parameter_name};} {**inout** <[bit_width]> parameter_name {, parameter_name};} {declaration} {sequential statement} **endtask**

Ex.

VHDL	Verilog

procedure sign_extend (
constant a: in std_logic_vector (3 downto 0);
signal x: out std_logic_vector (7 downto 0)) **is**
begin
if (a(3) = '1') **then**
x <= "1111" & a;
else
x <= "0000" & a;
end if;
end sign_extend;
 	
sign_extend(a, x); // cal procedure
	task sign_extend;
	input [3:0] a;
	output [7:0] x;
	if (a[3]) x = {4'b1111, a};
	else x = {4'b0000, a};
	endtask
	
	sign_extend(a, x); // call task

20. component, instantiation, configuration

VHDL	Verilog

component component_name
  port (
  {port_name{, port_name}: mode type;}
  port_name{, port_name}: mode type);
end component;
 	
label: component_name **port map** (
  <port_name **=>>** object_name
  {, <port_name **=>>** object_name});
 	
configuration configuration_name
  of entity_name **is**
  for arch_name
  for label: component_name
  use entity
  library_name.entity_name(arch_name);
  end for;
  for label: component_name
  use configuration
  library_name.configuration_name;
  end for;
  end for;
end configuration_name;
	module_name instance_name
	  (object_name{, object_name});
	module_name instance_name
	  (.port_name(object_name)
	  {, .port_name(object_name)});

Ex. AND-OR-INV

VHDL	Verilog

library ieee;
use ieee.std_logic_1164.all;
 	
entity aoi **is**
  port (a, b, c: **in std_logic;**
  x: **out std_logic**);
end aoi;
	module aoi (x, a, b, c);
	input a, b, c;
	output x;

architecture struct **of** aoi **is**
  component and2
  port (a, b: **in std_logic;**
  x: **out std_logic**);
  end component;
  component or2
  port (a, b: **in std_logic;**
  x: **out std_logic**);
  end component;
  component inv
  port (a: **in std_logic;**
  x: **out std_logic**);
  end component;
  signal w1, w2**: std_logic;**
begin
  i0: and2 **port map** (a => a, b => b, x => w1);
  i1: or2 **port map** (a => w1, b => c, x => w2);
  i2: inv **port map** (a => w2, x => x);
end struct;
	wire w1, w2;
	
	and2 i0 (.a(a), .b(b), .x(w1));
	or2 i1 (.a(w1), .b(c), .x(w2));
	inv i2 (.a(w2), .x(x));
	endmodule

21. package, library, use / ‘include

VHDL	Verilog

package package_name **is**
  {declaration}
end package_name;
package body package_name **is**
  {sub-program definitions}
end package_name;
library library_name;
use library_name.package_name.**all;**
	‘include “file_name”

Ex.

VHDL	Verilog

package component_pack **is**
  component and2
  port (a, b: **in std_logic;**
  x: **out std_logic**);
  end component;
  component or2
  port (a, b: **in std_logic;**
  x: **out std_logic**);
  end component;
  component inv
  port (a: **in std_logic;**
  x: **out std_logic**);
  end component;
end component_pack;
 	
library ieee;
use ieee.std_logic_1164.all;
use work.component_pack.all;
entity aoi **is**
  port(a, b, c: **in std_logic;**
  x: **out std_logic**);
end aoi;
architecture struct **of** aoi **is**
  signal w1, w2: std_logic;
begin
  i0: and2 **port map**(a=>a, b=>b, x=>w1);
…..
end struct;
	‘include “global_parameters.v”

ภาคผนวก

Blocking / Non-Blocking

คำสั่งที่เป็น blocking จะทำงานลงมาทีละบรรทัดหรือ เป็นลำดับขั้นนั่นเอง และเป็นลักษณะของการกำหนดค่าตัวแปรโดยใช้สัญลักษณ์ ”=” เป็นตัวกำหนดค่า

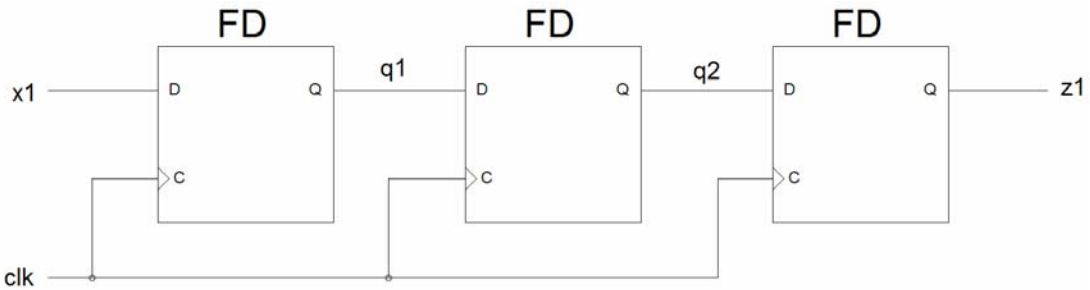
คำสั่งที่เป็น non-blocking จะทำงานพร้อมกันทุกบรรทัดหรือ เป็นแบบแข่งขนานนั่นเอง และเป็นลักษณะของการกำหนดค่าทางด้านซ้ายของ สัญลักษณ์ ”<=” มาไว้ที่ตัวแปลทางด้านขวามือ

ตัวอย่างที่ 1 Flip-flop based digital delay line

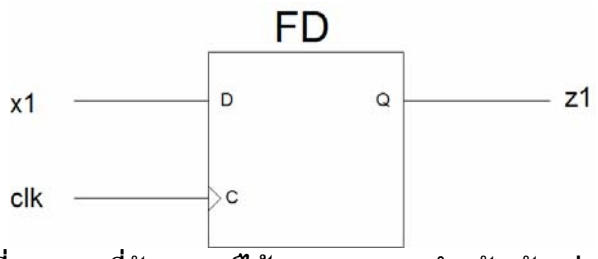
non-blocking	blocking

module nonblocking(x1, clk, z1);
input x1, clk;
output z1;
reg q1, q2, z1;
 	
always @(**posedge** clk)
begin
q1 <= x1;
q2 <= q1;
z1 <= q2;
end
endmodule
	module blocking(x1, clk, z1);
	input x1, clk;
	output z1;
	reg q1, q2, z1;
	
	always @(**posedge** clk)
	begin
	q1 = x1;
	q2 = q1;
	z1 = q2;
	end
	endmodule

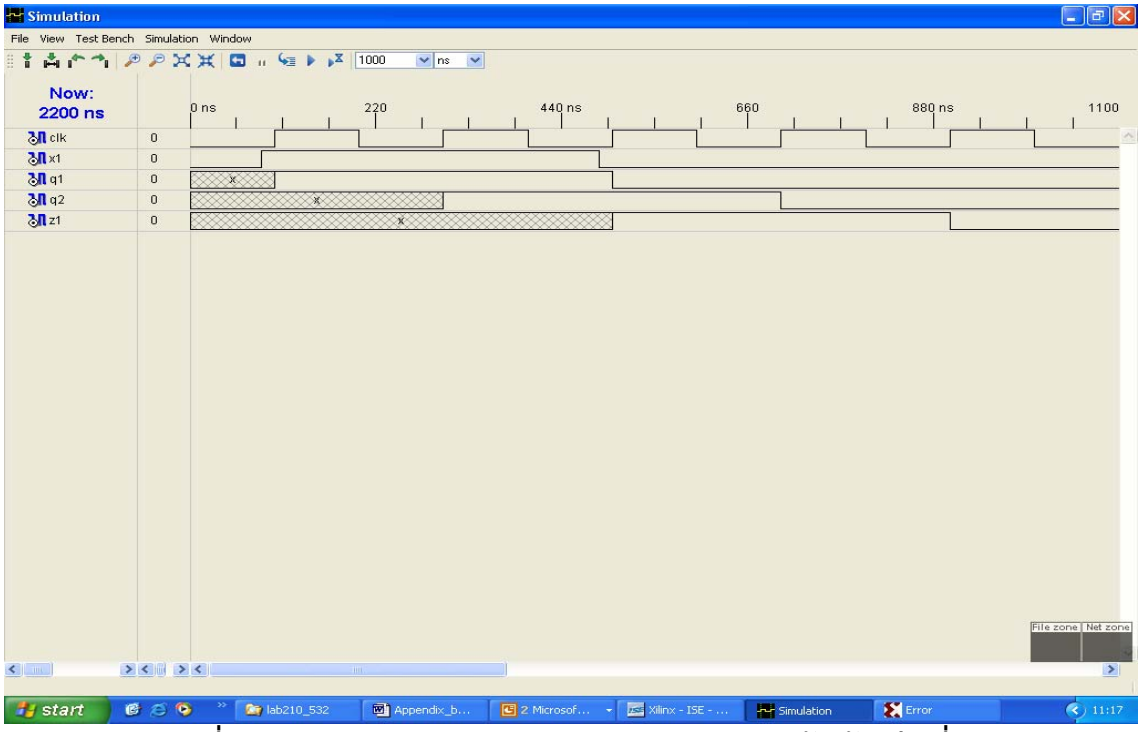
รูปที่ b1 โมดูล Verilog ของ non-blocking และ blocking สำหรับ ตัวอย่างที่ 1



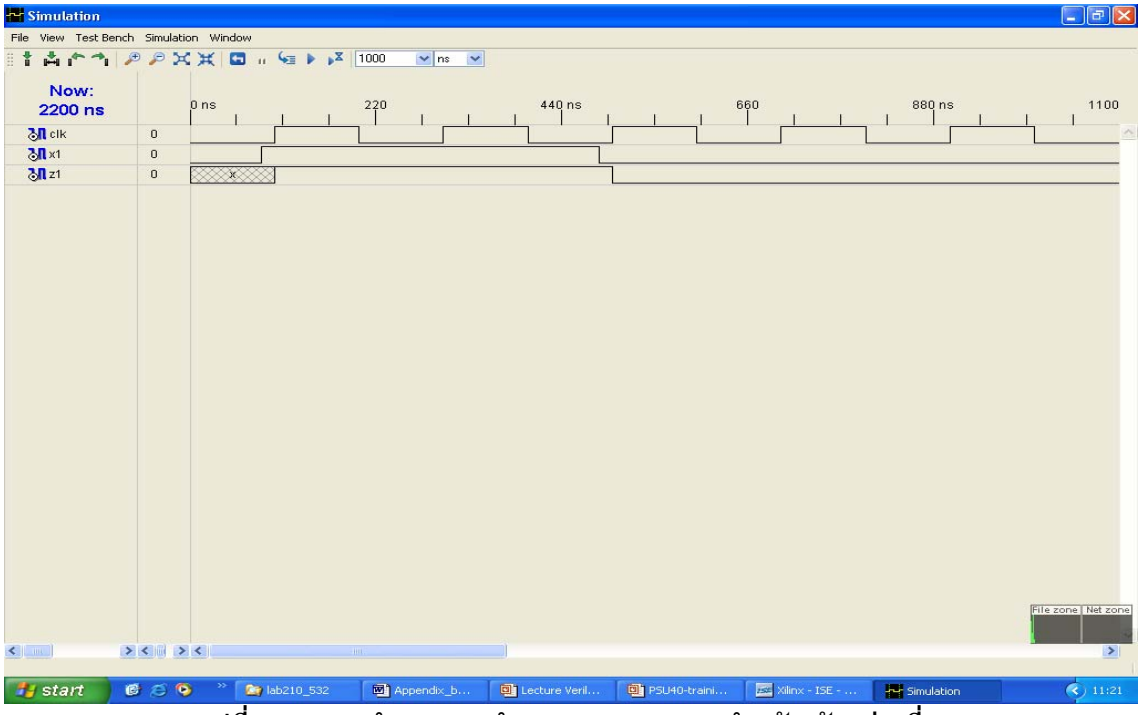
รูปที่ b2 วงจรที่สังเคราะห์ได้จาก non-blocking สำหรับ ตัวอย่างที่ 1



รูปที่ b3 วงจรที่สังเคราะห์ได้จาก blocking สำหรับ ตัวอย่างที่ 1



รูปที่ b4 ผลการจำลองการทำงานของ non-blocking สำหรับ ตัวอย่างที่ 1



รูปที่ b5 ผลการจำลองการทำงานของ blocking สำหรับ ตัวอย่างที่ 1

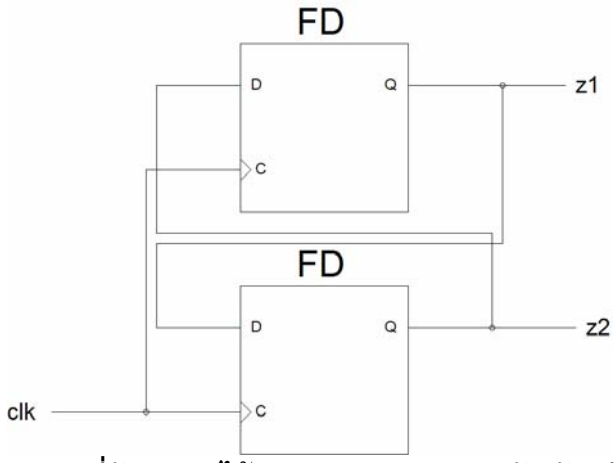
วงจรที่สังเคราะห์ได้และผลการจำลองการทำงานของคำสั่งแบบ block และ non-blocking ในรูปที่ b1-b5 สามารถแสดงให้เห็นความแตกต่างระหว่าง block และ non-blocking วงจรที่สังเคราะห์ได้จากคำสั่ง non-blocking เป็นวงจร shift register ซึ่งสามารถทำงานเป็น Flip-flop based digital delay line ได้ 3 สเตจ ที่

ทุกๆขอบขาขึ้นของสัญญาณนาฬิกา clk สัญญาณ q1 q2 และ z1 จะรับค่าเดิมของ in q1 และ q2 ตามลำดับมาเก็บพร้อมๆกัน ในขณะที่วงจรที่สังเคราะห์ได้จากคำสั่ง blocking เป็นวงจรรีจิสเตอร์ D flip-flop เพียงสเตจเดียว สัญญาณ q1 และ q2 จะถูกลบออกไปในขั้นตอนการสังเคราะห์วงจร

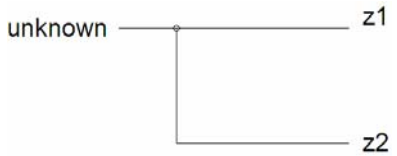
ตัวอย่างที่ 2 ออกแบบวงจรสลับค่าสัญญาณ

non-blocking	blocking

module nb_swap(clk, z1, z2);
input clk;
output reg z1, z2;
 	
always @(posedge clk**)**
begin
z1 <= z2;
z2 <= z1;
end
endmodule
	module b_swap(clk, z1, z2);
	input clk;
	output reg z1, z2;
	
	always @(posedge clk**)**
	begin
	z1 = z2;
	z2 = z1;
	end
	endmodule

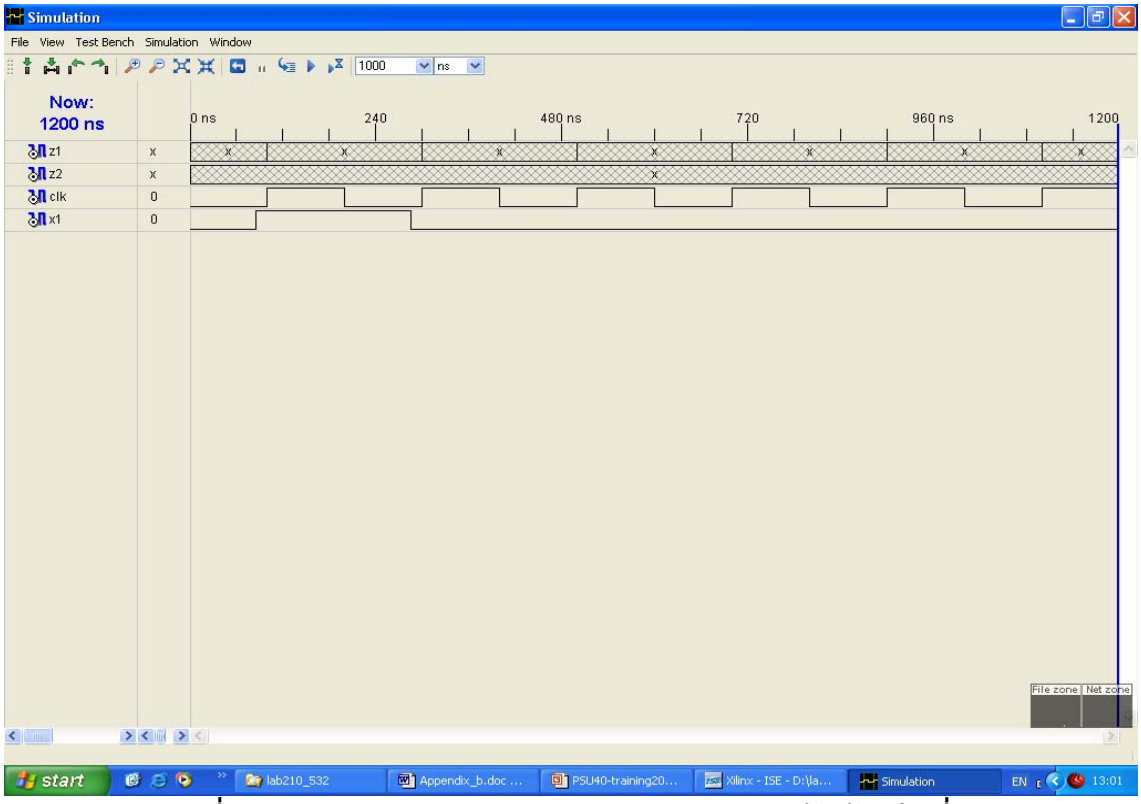


รูปที่ b6 วงจรที่สังเคราะห์ได้จาก non-blocking สำหรับ ตัวอย่างที่ 2

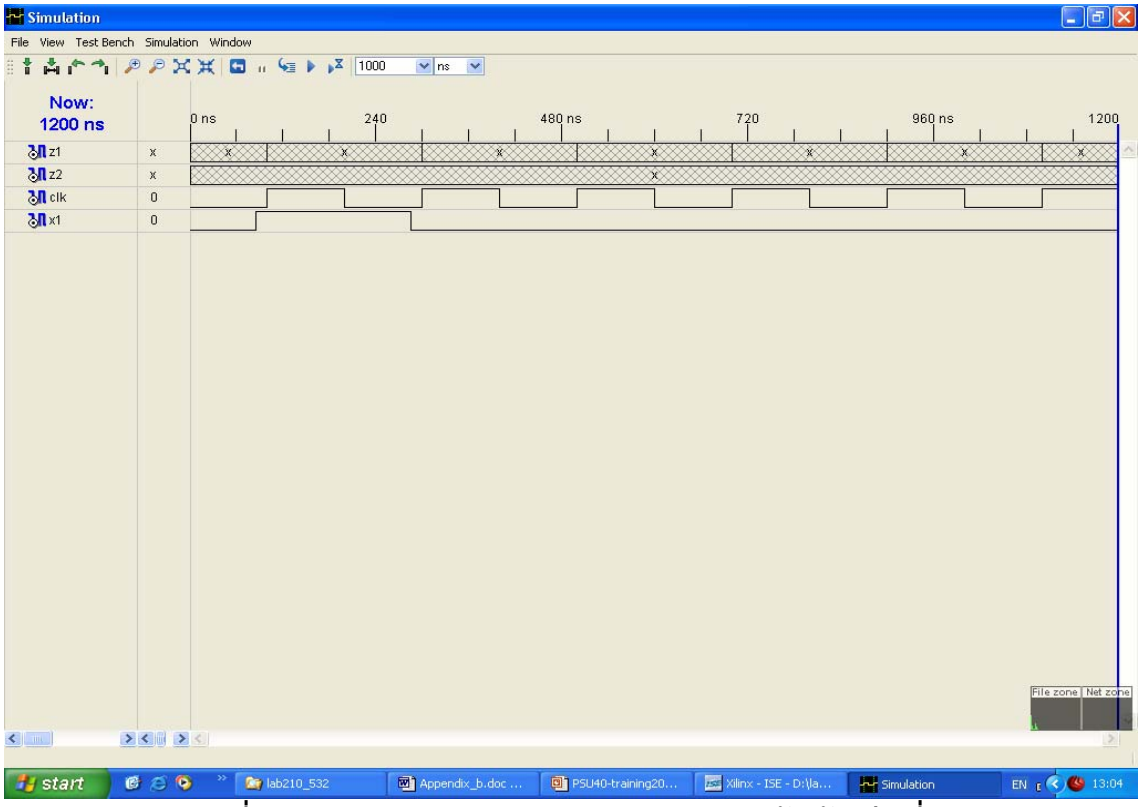


รูปที่ b7 วงจรที่สังเคราะห์ได้จาก blocking สำหรับ ตัวอย่างที่ 1

วงจรที่สังเคราะห์ได้และผลการจำลองการทำงานของคำสั่งแบบ block และ non-blocking สำหรับตัวอย่างที่ 2 แสดงดังรูปที่ b6-b9 สังเกตเห็นว่า วงจรที่สังเคราะห์ได้จากคำสั่งแบบ **non-blocking** เป็นวงจรสลับค่าสัญญาณได้ แต่วงจรที่สังเคราะห์ได้จากคำสั่งแบบ **blocking** เป็นเน็ทที่ค่าสัญญาณเท่ากัน ผลการจำลองการทำงานเอาท์พุทเป็นค่า unknown ทั้งหมดทั้งสองแบบทั้งนี้เพราะอินพุทไม่สามารถถูกป้อนในวงจรดังกล่าว ดังนั้นวงจรทั้งสองจึงไม่สามารถถูกใช้งานได้จริง



รูปที่ b8 ผลการจำลองการทำงานของ non-blocking สำหรับ ตัวอย่างที่ 2



รูปที่ b9 ผลการจำลองการทำงานของ blocking สำหรับ ตัวอย่างที่ 2