

# เรียนรู้การออกแบบระบบดิจิทัล ด้วยภาษา Verilog เบื้องต้น



โดย... ธีรยศ เวียงทอง  
ภาควิชาอิเล็กทรอนิกส์  
มหาวิทยาลัยเทคโนโลยีมหานคร



Version 3.0 : Copyright © 2005

## คำนำ

ปัจจุบันฮาร์ดแวร์ที่สามารถโปรแกรมได้ หรือที่เรียกกันว่า FPGA (Field Programmable Gate Array) มีความนิยมในการใช้งานในการออกแบบระบบดิจิทัลสูงมากขึ้นเรื่อย ๆ เนื่องจากความง่ายในการออกแบบและความรวดเร็วในการผลิตและสร้างโดยที่สามารถกระทำได้โดยใช้เครื่องคอมพิวเตอร์มาตรฐานทั่ว ๆ ไป นอกจากนี้สิ่งที่ทำให้แตกต่างจากระบบฮาร์ดแวร์เดิม ๆ เช่น ASIC (Application Specific IC) คือความสามารถในการโปรแกรม หรือใช้งานได้หลาย ๆ ครั้ง โดยไม่มีความเสียหาย หรือค่าใช้จ่ายเพิ่มเติมเกิดขึ้น ซึ่งขีฟประเภท FPGA นี้ทำให้มีการเปลี่ยนแปลงที่สำคัญของการออกแบบระบบดิจิทัลในปัจจุบัน

ภาษาที่ใช้ในการออกแบบระบบดิจิทัลใน FPGA ซึ่งมีสองภาษาที่สำคัญคือ VHDL และ Verilog ซึ่งส่วนใหญ่เท่าที่ทราบมาสำหรับในเมืองไทยเองนั้น VHDL จะเป็นที่ยอมรับใช้ในการออกแบบมากกว่า Verilog ผู้เขียนเองได้เรียนรู้ภาษา VHDL ก่อน แต่ต่อมาได้มีโอกาสเรียนรู้ และออกแบบระบบโดยใช้ภาษา Verilog ซึ่งง่ายและสะดวกกว่าที่คิดไว้อย่างมาก เนื่องจากเป็นภาษาที่คล้ายคลึงกับภาษา C และมีไวยากรณ์ หรือข้อบังคับที่ถูกต้องน้อยกว่าภาษา VHDL

ผู้เขียนตั้งใจนำเสนอภาษา Verilog ในฉบับภาษาไทย โดยมีความตั้งใจมานานพอสมควร จนหาเวลาที่เหมาะสมได้ จึงมาเป็น Verilog ฉบับย่อเล่มนี้ โดยอ้างอิงมาจาก Evita HDL ของ Aldec Inc. (สามารถดาวน์โหลดได้ที่ <http://www.aldec.com/products/tutorials/>) ซึ่งเหมาะสำหรับผู้เริ่มต้น ส่วนของรายละเอียดลึก ๆ ของภาษา Verilog คาดว่าคงจะนำเสนอเป็นหนังสือฉบับสมบูรณ์ต่อไป

ธีรยศ เวียงทอง

7 กันยายน 2548

# สารบัญ

บทที่	หน้า
1. Why do we need Verilog? .....	4
2. A System in Verilog Representation.....	9
3. Signals.....	16
4. A Structural View of a System.....	23
5. Specification with Signal Transformations.....	32
6. The Behavioral Approach.....	41
<b>Appendix</b>	
Verilog Quick Reference.....	52

# CHAPTER 1

## Why do we need Verilog?

ในบทนี้จะเริ่มต้นด้วยคำถามที่ว่า ทำไมถึงต้องเรียนรู้ Verilog ซึ่งเป็นคำถามแรกที่สำคัญ สำหรับผู้ที่ยังไม่เคยใช้ภาษาชั้นสูงในการออกแบบฮาร์ดแวร์ หรือผู้ที่เคยใช้ภาษา VHDL มาก่อน ในบทนี้จะช่วยให้ผู้อ่านในการตอบคำถามนี้ หลังจากนั้น จะพูดถึงวิธีการออกแบบโดยทั่วไปที่มีอยู่สองวิธีได้แก่ การออกแบบลอจิกโดยใช้สมการบูลีน (Boolean equation) และวิธีการที่นิยมใช้คือการออกแบบด้วย Schematic ในโปรแกรม CAD (Computer aided design) ซึ่งทั้งสองวิธีนี้ยังใช้กันอยู่ในปัจจุบัน แต่มีข้อเสียมากมายโดยเฉพาะความสามารถในการรองรับการออกแบบที่ซับซ้อนในปัจจุบัน ซึ่งข้อเสียบางอย่างของการออกแบบโดยทั่วไปทั้งสองวิธีนี้สามารถจัดไปได้ด้วยภาษาที่ใช้อธิบายฮาร์ดแวร์ระดับต่ำ (Low-level hardware description language)

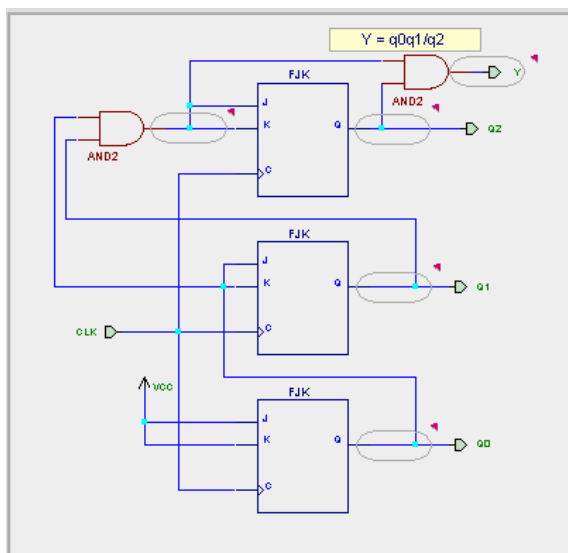
เนื่องจากในปัจจุบัน การออกแบบทางดิจิทัลฮาร์ดแวร์ ประกอบด้วยจำนวนของลอจิกเกตนับพัน ๆ ถึงแสน ๆ ตัว ผู้ออกแบบจำเป็นต้องมีเครื่องมือในการออกแบบที่สามารถรองรับความต้องการนี้ได้ อย่างมีประสิทธิภาพ มากกว่าการออกแบบด้วยสมการบูลีน หรือ Schematic เครื่องมือที่เราจะพูดถึงนี้ได้แก่ภาษาชั้นสูงที่ใช้ในการอธิบายการทำงานของฮาร์ดแวร์ (High-level hardware description language) ซึ่งภาษาประเภทนี้ที่เป็นที่นิยมใช้กันอย่างกว้างขวางในการออกแบบดิจิทัลฮาร์ดแวร์คือ VHDL และ Verilog โดยจะมีการเปรียบเทียบให้เห็นจุดเด่น และด้อยระหว่างภาษาทั้งสองในช่วงท้ายของบทนี้

### โลกของการออกแบบก่อนหน้า Verilog (The World Before Verilog)

#### การออกแบบด้วยสมการบูลีน (Designing with Boolean Equations)

คงเป็นการยากสำหรับการออกแบบระบบดิจิทัลระบบหนึ่งโดยปราศจากความเข้าใจถึงอุปกรณ์พื้นฐาน เช่น เกต (Gate) หรือฟลิปฟล็อป (Flip-flop) เพราะวงจรลอจิกโดยส่วนใหญ่แล้วจะสร้างมาจากเกต และฟลิปฟล็อป ซึ่งกำหนดได้จากสมการบูลีน มีหลาย ๆ เทคนิคที่ถูกออกแบบมาสำหรับการ Optimize การออกแบบ ประกอบด้วยการลดรูปของสมการบูลีนโดยใช้ Karnaugh map (K-Map) สำหรับการใช้ทรัพยากรคือ เกต หรือฟลิปฟล็อปให้น้อยที่สุด

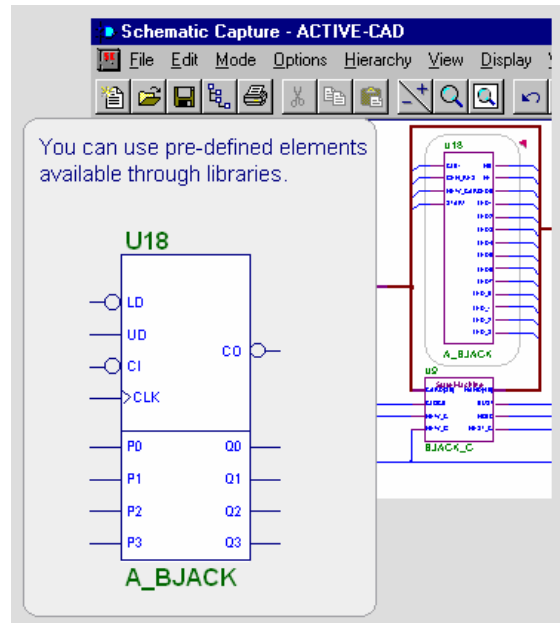
การออกแบบด้วยสมการบูลีนต้องการสมการสำหรับการกำหนดการใช้งานของแต่ละเกต ดังนั้นจึงทำให้ไม่สามารถใช้งานได้จริงสำหรับระบบที่มีขนาดใหญ่ ที่ประกอบด้วยจำนวนเกตเป็นแสน ๆ ตัว ที่จะต้องกำหนดมาจากสมการบูลีนหลาย ๆ พันสมการ แม้ในทางทฤษฎีแล้ว ทุก ๆ ระบบสามารถแทนได้ด้วยสมการบูลีน แต่อย่างไรก็ตามมันไม่สามารถนำมาใช้ในการออกแบบจริงของระบบดิจิทัลที่ซับซ้อนในปัจจุบัน



### การออกแบบด้วย Schematic (Schematic-based Design)

การออกแบบด้วย Schematic ช่วยในการขยายความสามารถในการออกแบบด้วยสมการบูลีน เนื่องจากไม่มีแค่เกต หรือฟลิปฟล็อปที่เป็นอุปกรณ์พื้นฐานเท่านั้น แต่ยังมียังจร หรือตัวไอซีที่อยู่ในไลบรารี (Library) ด้วย นอกจากนี้ วิธีนี้ยังสามารถออกแบบระบบเป็นลักษณะของลำดับชั้น (Hierarchy) ได้อีกด้วย ซึ่งทำให้สามารถออกแบบระบบที่ซับซ้อนได้ และสามารถประหยัดเวลาในการออกแบบเนื่องจากสามารถมองเห็นเป็นรูปร่างของระบบที่ชัดเจน สามารถเรียกใช้อุปกรณ์ได้หลาย ๆ ครั้ง (Design reuse) และแชร์การใช้งานระหว่างผู้ออกแบบโดยสร้างเป็นไลบรารีร่วม (Shared library) ทำให้ง่ายในการออกแบบเป็นกลุ่ม

หลายคนจะชอบลักษณะการออกแบบที่มองเห็นเป็นภาพ (Graphical representation) โดยสามารถมองเห็นได้ว่าอุปกรณ์ตัวไหน ต่อกันอย่างไร ซึ่งจากการออกแบบที่ง่ายขึ้นในลักษณะ Schematic นั้นทำให้เป็นที่นิยมกัน โดยในหลายๆ ปีก่อนหน้านั้น การออกแบบลักษณะนี้ถูกมองว่าจะเป็นวิธีการออกแบบที่ดีสำหรับระบบต่าง ๆ จนกระทั่งเราต้องการระบบที่ซับซ้อนที่มีจำนวนอุปกรณ์มากขึ้น อาจจะเป็นหลาย ๆ พันตัว ซึ่งก็ทำให้เวลาในการออกแบบนั้นมากขึ้นเรื่อย ๆ เนื่องจากจะต้องเสียเวลาในการเชื่อมต่อแต่ละขาของอุปกรณ์ รวมทั้งยากในการตรวจสอบความถูกต้อง และการแก้ไข ที่อาจเป็นบริเวณใดบริเวณหนึ่งของวงจร Schematic ที่มีการเชื่อมต่อกันอย่างมากมาย



### ข้อเสียของการออกแบบแบบเดิม (Drawback of Traditional Methods)

แม้จะเป็นลักษณะของการออกแบบที่ง่าย ของการออกแบบระบบแบบเดิมที่ใช้สมการบูลีน หรือใช้ Schematic แต่มีข้อเสียที่สำคัญคือ ระบบถูกกำหนดเป็นการเชื่อมต่อเป็นโครงข่ายของอุปกรณ์พื้นฐานต่าง ๆ เช่น เกต ฟลิปฟล็อป หรืออุปกรณ์ในไลบรารี แต่ไม่ได้เป็นการสร้างจากวิธีการในการกำหนดความสามารถในการทำงานของระบบ (System specification) โดยตรง ดังนั้นผู้ออกแบบจำเป็นต้องมีพื้นฐานความรู้พอสมควร ในการออกแบบเช่น การออกแบบวงจรบวก ผู้ออกแบบก็ต้องรู้ตั้งแต่ Half-adder หรือ Full-adder จะต้องใช้การต่อกันของ And/Or/Xor gate อย่างไรบ้าง อีกทั้งความง่ายในการทำความเข้าใจว่าเป็นวงจรที่ทำงานอะไรโดยผู้ออกแบบคนอื่น โดยดูจาก Schematic หรือสมการบูลีนนั้นค่อนข้างยาก

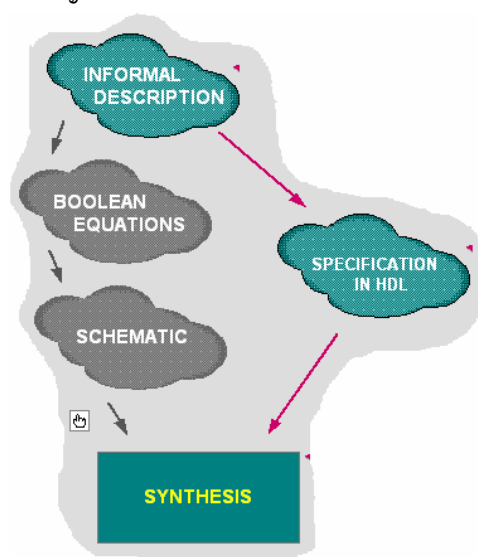
ข้อเสียอีกอย่างคือ ความสามารถในการรองรับการออกแบบระบบที่ซับซ้อน การจัดการกับสมการบูลีนแค่ร้อยละสมการ ก็ค่อนข้างยาก (แต่สามารถทำได้ ถ้ามีเวลาพอ) อย่างไรก็ตามถ้าเพิ่มเป็นพัน หรือหมื่นสมการ ถือว่าเป็นเรื่องที่ยากมาก กล่าวกันว่า การออกแบบโดยใช้ Schematic นั้นสามารถรองรับระบบที่มีเพียงหกพันเกตเท่านั้น ถ้าเกินกว่านั้นจะทำให้ระบบยากในการตรวจสอบ หรือทำความเข้าใจ

จะเห็นได้ว่าในปัจจุบัน ความก้าวหน้าทางด้านวงจรรวม (Integrated circuit) ที่สามารถบรรจุเกต ได้เป็นล้าน ๆ ตัวในชิพ เราต้องการการออกแบบระบบที่ใหญ่ และซับซ้อน เพื่อตอบสนองความต้องการการใช้งานชิพในด้านต่าง ๆ ดังนั้นจึงจำเป็นต้องมีวิธีที่สามารถรองรับการออกแบบระบบดิจิทัลในลักษณะนี้ได้

## ภาษาฮาร์ดแวร์ (Hardware Description Language)

ภาษาฮาร์ดแวร์ คือภาษาชั้นสูงที่ใช้ในการอธิบายการทำงานของฮาร์ดแวร์ โดยที่เราไม่จำเป็นต้องแปลงเป็นสมการบูลีน หรือ Schematic ตัวอย่างเช่น ภาษา HDL ส่วนใหญ่สามารถสร้าง หรืออธิบาย Finite state machine สำหรับวงจร Sequential logic ได้ หรือสร้างเป็น ตารางค่าความจริง (Truth table) สำหรับวงจร Combinational logic ได้

ภาษา HDL ใช้เป็นภาษาหลักในการออกแบบระบบใน อุปกรณ์ประเภท Programmable logic device (PLD) หรือ Complex PLD (CPLD) และ Field programmable gate array (FPGA) ซึ่งในปัจจุบันมีภาษาลักษณะนี้ใช้งานกัน หลาย ๆ ตัว แต่ตัวที่เป็นที่นิยม และรู้จักกันได้แก่ Abel, Palasm, Cupl ที่ใช้สำหรับการออกแบบวงจรที่ไม่ซับซ้อน นักใน PLD หรือ Verilog, VHDL สำหรับการออกแบบ ระบบที่ซับซ้อนในอุปกรณ์พวก CPLD หรือ FPGA

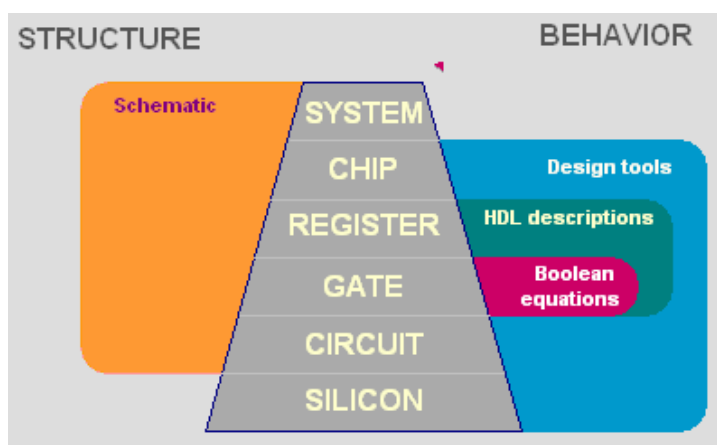


## การออกแบบในระดับต่าง ๆ (Dealing with Different Design Levels)

เนื่องจากวงจรในปัจจุบันเพิ่มความซับซ้อนมากขึ้นเรื่อย ๆ ดังนั้นแนวโน้มของความต้องการในการออกแบบคือ ความสามารถในการเปลี่ยน หรือรองรับลักษณะของ Design entry ได้หลาย ๆ แบบ เช่น สามารถออกแบบได้ทั้งการใช้ภาษาชั้นสูงในการอธิบายระบบ หรือการใช้สมการบูลีน หรือการใช้ Schematic

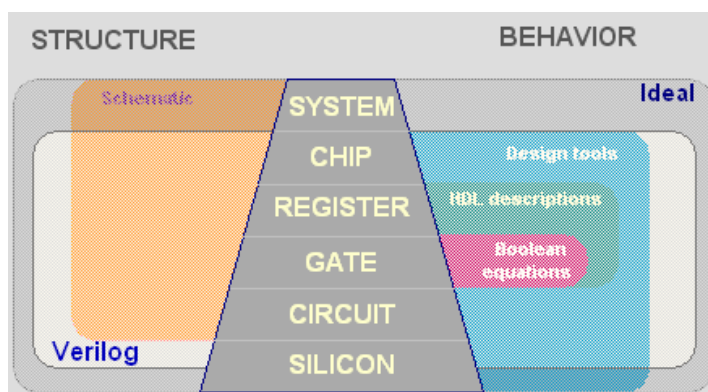
เราสามารถแบ่งระดับของการอธิบายระบบได้เป็นชั้น จากชั้นล่างสุดได้แก่ Silicon level ไปยังชั้นสูงสุดได้แก่ System level โดยที่แต่ละระดับสามารถแยกได้เป็นการอธิบายในลักษณะของโครงสร้าง (Structure) หรือการทำงาน (Behavior)

การออกแบบในระดับล่าง ๆ นั้นสามารถกำหนดรายละเอียด หรือคุณสมบัติของวงจรได้ทุกขั้นตอน เริ่มตั้งแต่ขนาดของทรานซิสเตอร์ ที่จะนำมาสร้างเป็นเกต ทำให้เราได้วงจรตามที่กำหนดร็อยเปอร์เซ็นต์ แต่ทว่าผู้ออกแบบจะต้องใช้เวลาในการออกแบบมาก รวมทั้งจำเป็นต้องมีประสบการณ์พอสมควร ส่วนการออกแบบในระดับสูง ๆ นั้นเราสามารถทำการออกแบบได้เร็ว จากการกำหนดการทำงานของระบบในระดับที่สูงขึ้นทำให้ทำได้ง่ายขึ้น หน้าที่ของการสร้างวงจรในระดับล่างลงมาในแต่ละระดับนั้น ตัวเครื่องมือจะเป็นตัวรับผิดชอบ



## Verilog HDL: คำตอบสำหรับการออกแบบฮาร์ดแวร์

การออกแบบด้วยภาษาชั้นสูงอย่าง Verilog สามารถครอบคลุมลักษณะของการออกแบบได้เกือบทุกระดับ ทั้งทางด้านโครงสร้าง (Structure) หรือลักษณะการทำงาน (Behavior) ซึ่งถือว่าเป็นประโยชน์สำหรับการเปลี่ยนหรือย้าย (Transfer) การออกแบบไปยังระดับต่าง ๆ ในแต่ละเฟสของการออกแบบ หรือการทำเอกสารประกอบในการออกแบบ เนื่องจากความเป็นเอกภาพของการออกแบบนั่นเอง



### อะไรคือ Verilog HDL (What is Verilog HDL?)

Verilog หรือชื่อเต็ม ๆ คือ Verilog HDL เป็นภาษาที่ใช้ในการอธิบายลักษณะการทำงานของฮาร์ดแวร์ (Hardware description language) ภาษาหนึ่งที่ถูกสร้างขึ้นมาในช่วง คศ 1984-1985 โดย Philip Moorby ที่ต้องการภาษาที่ง่าย และมีประสิทธิภาพในการอธิบายลักษณะของวงจรดิจิทัล สำหรับการโมเดล การจำลอง การทำงาน และการวิเคราะห์การทำงาน ภาษานี้ได้กลายเป็นลิขสิทธิ์ของบริษัท Design Gateway Automation ซึ่งต่อมาได้รวมกับบริษัท Cadence Design Systems และตั้งแต่นั้นปี คศ 1990 เป็นต้นมา ทาง Cadence ได้เปิดให้เป็นภาษาที่ทุกคนสามารถใช้งานได้ ทั้งนี้เพื่อให้ง่ายในการกำหนดมาตรฐาน และพัฒนา ร่วมกัน และในที่สุด ภาษานี้ได้เข้าเป็นภาษามาตรฐานของ IEEE ในปี คศ 1995 (IEEE Standard 1364)

คุณลักษณะที่สำคัญในภาษานี้ได้แก่

- **Universal:** ภาษา Verilog อนุญาตให้การออกแบบทั้งระบบทำได้ภายใต้สภาวะแวดล้อมของการออกแบบ (Design environment) เดียวกัน ที่ประกอบด้วย การวิเคราะห์ และการตรวจสอบ (Analysis and verification) อย่างไรก็ตาม Verilog อาจไม่เหมาะกับระดับของการออกแบบที่เป็น Complex system design ที่ต้องการภาษาชั้นสูงกว่านี้ที่อาจต้องการความสามารถในการอธิบายการทำงานได้ทั้งซอฟต์แวร์ และฮาร์ดแวร์
- **Extensibility:** มาตรฐาน IEEE std 1364 ประกอบด้วย Verilog PLI (Programming Language Interface) ซึ่งช่วยขยายความสามารถของภาษา Verilog โดยจะประกอบด้วยวิธีการที่จะเพิ่มเติมฟังก์ชันต่าง ๆ ขึ้นมา หรือการติดต่อกับภาษาอื่น เป็นต้น
- **Industrial support:** ภาษา Verilog ได้รับการต้อนรับเป็นอย่างดีจากผู้ออกแบบวงจรประเภท ASIC (Application specific circuit) เนื่องจากเป็นภาษาที่ง่ายในการเรียนรู้ และช่วยให้การจำลองการทำงาน หรือตรวจสอบระบบทำได้อย่างรวดเร็ว จากข้อมูลของนิตยสาร EE Times ได้พูดถึงเกี่ยวกับการใช้งานภาษา Verilog ในการออกแบบ ASIC ในปี 1993 นั้นมีถึง 85% ของการออกแบบทั้งหมด

## Verilog เปรียบเทียบกับ VHDL

ผู้ออกแบบมือใหม่หลาย ๆ คนมักตั้งคำถามกับตัวเองว่า แล้วภาษา HDL ภาษาไหนควรนำมาใช้ในการออกแบบดี VHDL หรือ Verilog คำตอบก็คือทั้งสองภาษาต่างมีข้อได้เปรียบ เสียเปรียบ หรือยากง่ายต่างกัน แล้วแต่ความถนัด หรือความชอบของผู้ใช้งาน ดังนั้นเมื่อเปรียบเทียบกันทั้งสองภาษา ถือว่าไม่มีผู้ชนะ

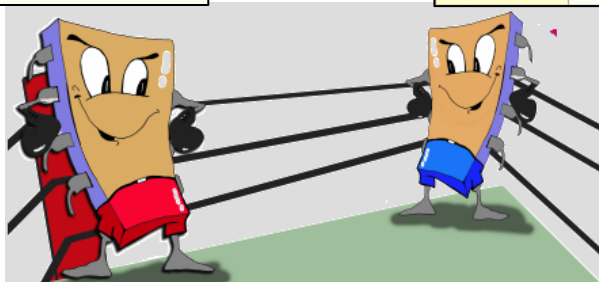
ความได้เปรียบของภาษา Verilog คือ ความง่ายในการทำความเข้าใจ และใช้งาน โครงสร้างไวยากรณ์เหมือนภาษา C และไม่จุจิก ถือว่ามีความยืดหยุ่นในการเขียนมากกว่า VHDL ดังนั้นจึงค่อนข้างเป็นที่นิยมในการใช้งานในการออกแบบทั่วไปในอุตสาหกรรมวงจรรวม โดยเฉพาะทางอเมริกา และญี่ปุ่น อย่างไรก็ตาม ภาษา Verilog ค่อนข้างจะด้อยในด้านความสามารถในการกำหนดการทำงานของระบบในระดับที่สูงขึ้น (System level specification)

สำหรับภาษา VHDL จะซับซ้อนกว่า มีคุณสมบัติด้านต่าง ๆ (Feature) ที่เยอะกว่า (ซึ่งบางทีไม่จำเป็นในการทำงานการออกแบบโดยทั่วไป) จึงมีไวยากรณ์ของภาษา หรือกฎต่าง ๆ มากกว่า ทำให้ค่อนข้างยากในการเรียนรู้ และใช้งาน แต่ข้อดีคือมีความยืดหยุ่นในการใช้งานสูง เนื่องจากสามารถใช้การเขียนออกแบบรูปแบบต่าง ๆ มากมาย (Permissible coding styles) ดังนั้น VHDL จึงเหมาะสำหรับการออกแบบระบบที่ซับซ้อน ทำให้ได้รับความนิยมมากจากนักออกแบบวงจรดิจิทัลโดยทั่วไป โดยเฉพาะแถบทางด้านยุโรป

<b>Name:</b>	Verilog HDL
<b>Coach:</b>	Open Verilog International (www.oiv.org)
<b>Characteristics:</b>	Has very good acceptance in ASIC, particularly lower level designs (register level transfer and below); results in fast simulations, relatively simple, easy in first contacts, especially for C Language users. In long term might have problems with handling system level designs.
<b>Fan clubs:</b>	Mostly in North America and Asia Japan, especially among industrial supporters. Not popular in Europe.

## Verilog Vs VHDL

<b>Name:</b>	VHDL
<b>Coach:</b>	VHDL International (www.vhdl.org)
<b>Characteristics:</b>	Relatively weaker in lower level designs, but superior in higher- and system level designs; results in slower simulations, but constantly improving; very flexible, but also difficult, complex character; very popular in academia; lots of his skills were taught by Special Forces Officer Ada; many believe that in long term presents better condition and adaptability than its competitor.
<b>Fan clubs:</b>	Especially in Europe, but significant number also in US and Canada. Disliked in Japan, but gaining popularity worldwide.





## CHAPTER 2

# A System and Verilog Representation

ภาษา Verilog ได้ถูกพัฒนาขึ้นมาเพื่อใช้ในการอธิบายการทำงานของวงจรดิจิทัล ที่มีการทำงานพร้อม ๆ กันหลาย ๆ วงจรย่อย เป็นลักษณะแบบขนาน (Concurrency) และความต้องการที่จะต้องเป็นภาษาที่สามารถใช้ในการจำลองการทำงานได้ โดยที่ภาษาขั้นสูงทั่วไปอย่างเช่น ภาษา C, Basic ไม่สามารถนำมาใช้งานในลักษณะนี้ได้ ในบทนี้จะแนะนำให้รู้จักภาษา Verilog โดยจะพิจารณาถึงความสามารถในการอธิบายระบบดิจิทัลโดยทั่วไป

ในส่วนแรก อธิบายถึงแนวคิดของการสร้างระบบโดยใช้ภาษา Verilog ที่มองระบบเหมือนกับกล่องดำ (Black box) อันหนึ่งที่มีส่วนที่ต่อเชื่อมกับภายนอกอยู่รอบ ๆ ซึ่งตัวกล่องดำอันนี้ใน Verilog เราเรียกว่า โมดูล (Module) หลังจากนั้นจะพูดถึงส่วนประกอบของ Verilog ที่อยู่ในโมดูล รวมทั้งวิธีการตั้งชื่อโมดูลที่ถูกต้อง ท้ายสุดของบทนี้จะบอกวิธีการกำหนดการทำงานของกล่องดำนี้ ซึ่งมีอยู่หลัก ๆ ด้วยกันสามแบบ ได้แก่ Structural, Dataflow และ Behavioral โดยจะมีตัวอย่างวิธีการกำหนดโดยทั่วไปของการอธิบายระบบโดยใช้วิธีการทั้งสามแบบนี้

### ส่วนประกอบของระบบ (The Anatomy of a System)

#### ระบบ วงจร และโมดูล (System, Circuit and Module)

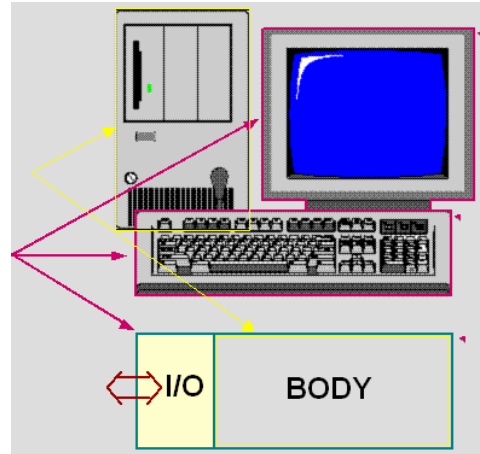
ภาษา HDL ถูกใช้ในการอธิบายลักษณะของวงจร หรือระบบอิเล็กทรอนิกส์ ซึ่งโดยทั่วไปแล้วระบบ (System) จะถูกอ้างอิงในภาษา HDL เหล่านี้ในลักษณะเป็น Entity โดยที่ส่วนประกอบของระบบนั้น ได้แก่ วงจร (Circuit) ต่าง ๆ ที่ทำงานสอดคล้องกัน ให้ได้เป็นผลลัพธ์หรืออย่างที่ผู้ออกแบบต้องการ แต่อย่างไรก็ตามทั้งความหมายของวงจร หรือระบบนั้นสามารถนำมาใช้แทนกันได้ แล้วแต่การมองของผู้ออกแบบแต่ละคน

ข้อกำหนดของระบบ (System specification) นั้นจะต้องสามารถอธิบายองค์ประกอบ (Elements) ต่าง ๆ ในระบบ และความเกี่ยวข้องระหว่างกันให้ได้ สำหรับ Verilog เองนั้น ระบบจะถูกกำหนดได้ด้วยคำสั่ง **module** และจบด้วย **endmodule** ดังแสดงในรูป



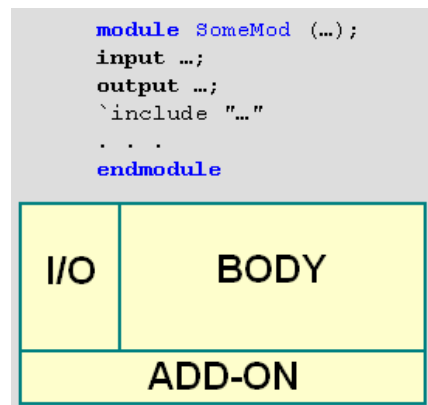
เมื่อไรก็ตามที่ระบบทำงาน ระบบจะรับอินพุตข้อมูลเข้ามา และสร้างเอาต์พุตตามฟังก์ชันที่ถูกกำหนด ขึ้นมาในระบบนั้น ๆ หรือมองอีกนัยหนึ่งก็คือ ระบบทำการติดต่อ (Interface) กับสิ่งต่าง ๆ รอบตัวมันผ่านทาง พอร์ตอินพุต และพอร์ตเอาต์พุต ถ้าปราศจากการอินเทอร์เฟซ ระบบก็จะไม่สามารถทำงานได้ ลองคิดดูง่าย ๆ ว่าถ้าเรามีคอมพิวเตอร์โดยปราศจากเมาส์ หรือคีย์บอร์ด (ซึ่งถือว่าเป็นอินพุตของระบบ) และมอนิเตอร์ (เอาต์พุตของระบบ) นั้นจะเป็นอย่างไร

ในระบบต่าง ๆ นั้นเราสามารถแยกได้เป็นสองส่วน หลัก ๆ คือ Body และ Interface หน้าที่ของ Body คือ ทำ การประมวลผลข้อมูลที่รับเข้าไป และส่งออกมา ผ่านทาง ส่วนของการ Interface นอกจากนี้ระบบบางระบบอาจมี ส่วนเพิ่มเติม หรือที่เรียกว่า Add-on ที่จะมาใช้งานควบคู่ กับระบบที่มีอยู่ ซึ่งถ้าเทียบกับเครื่องคอมพิวเตอร์เครื่อง หนึ่งแล้ว ส่วนของ Body คือ Main board นั้นเอง ส่วนของ Interface คือ เมาส์ และคีย์บอร์ด และในส่วนของ Add-on อาจจะเป็น Sound card หรือ Video card เป็นต้น



### Interface และ Body ในโมดูล Verilog (Interface and Body in Verilog Module)

เนื่องจาก **module** ใน Verilog ใช้แทนระบบดิจิทัล ดังนั้น **module** จะต้องสามารถแทนส่วนประกอบ ของระบบได้ นั่นคือ Body และ Interface



พอร์ตของโมดูลจะถูกกำหนดไว้ในวงเล็บ หลังจากชื่อของ โมดูล โดยส่วนนี้จะทำหน้าที่เป็น Interface ของระบบ โดยที่ รายละเอียดต่าง ๆ ของแต่ละพอร์ต (Direction and type) เช่น เป็นอินพุต หรือเอาต์พุตพอร์ท ขนาดกี่บิต จะถูกกำหนดข้างล่าง ต่อมา หลังจากวงเล็บปิด (ดูรูป) คำสั่ง **'include** เหมือนกับ Add-on ของระบบที่จะเรียกไฟล์อื่นมาใช้งาน หลังจากนั้นจะเป็นส่วน ของ Body ที่จะกำหนดการทำงาน หรือเรียกใช้โมดูลย่อยที่มีอยู่ แล้วมาใช้งานในระบบ ซึ่งมีวิธีการกำหนดการทำงานแตกต่างกัน สามแบบคือ Structural, Dataflow หรือ Behavioral ดังจะแสดง ให้อีกต่อไป

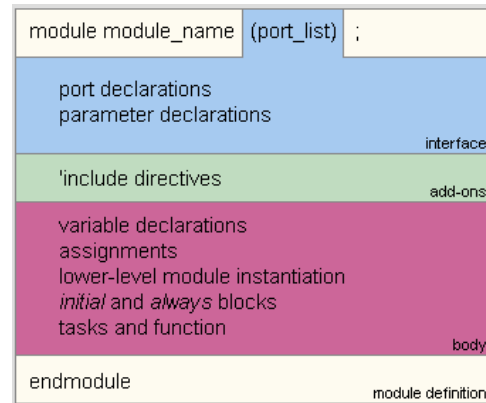
### โครงสร้างของโมดูล Module (The Structure of a Module)

#### คำจำกัดความของโมดูล (The Definition of a Module)

สำหรับโมดูล ใน Verilog หมายถึง บล็อกพื้นฐานที่สร้างขึ้นมาสำหรับการกำหนดคุณสมบัติของระบบ ทั้ง ระบบเล็ก ๆ ที่เป็นเกตเพียงตัวเดียว หรือระบบใหญ่ ที่ประกอบด้วยโมดูลย่อย ๆ หลาย ๆ โมดูล อย่างไรก็ตาม ไม่ว่าระบบเล็ก หรือใหญ่ ที่อธิบายด้วยโมดูลนั้น จะต้องมีการสร้างเหมือนกัน

ทุก ๆ โมดูล จะต้องเริ่มต้นด้วยคำสงวน (Reserve words) คือ **module** ตามด้วยชื่อของโมดูล และจบ ด้วยคำสงวน **endmodule** (สังเกตว่า ไม่มีช่องว่างระหว่างคำว่า end กับคำว่า module) ภายในตัวโมดูลจะ ประกอบด้วยสามส่วนดังที่กล่าวมาแล้วข้างต้น คือ

- **Interface:** ประกอบด้วยพอร์ต และการกำหนดตัวพารามิเตอร์ของโมดูล
- **Body:** จะเป็นการกำหนดการทำงานของตัวโมดูล
- **Optional add-on:** สามารถกำหนดได้ทุกอย่างในโมดูล เพื่อเรียกใช้ส่วนประกอบอื่น ๆ ที่เขียนไว้แล้วเป็นไฟล์ มาใช้เพิ่มเติม โดยใช้คำสั่ง 'include



### ชื่อของโมดูล (The Name of a Module)

หลักของการตั้งชื่อโมดูลคือ กระทัดรัด และมีความหมายที่สามารถเข้าใจได้ง่าย ที่เป็นการอธิบายว่าโมดูลตัวนั้นทำหน้าที่อะไร เนื่องจากข้อกำหนดของภาษา Verilog จะเป็นภาษาที่คำนึงถึงขนาดตัวอักษรที่ใช้ (Case sensitive) ซึ่งไม่เหมือนกับ VHDL ที่ตัวอักษรเล็กใหญ่สามารถใช้แทนกันได้ ดังนั้นเราสามารถใช้ลักษณะตัวอักษรเล็กใหญ่สลับกันเพื่อให้อ่านได้ง่ายขึ้น ตัวอย่างเช่น FastBinaryCnt เป็นต้น (สังเกตว่าคำสงวนใน Verilog จะใช้ตัวเล็กเท่านั้น)

โดยสรุป มีข้อกำหนดในภาษา Verilog ในการตั้งชื่อโมดูลดังนี้

- ประกอบด้วย ตัวอักษรภาษาอังกฤษ (Letter) ตัวเลข (Digit) เครื่องหมาย \$ (Dollar sign) และ \_ (Underscore) เท่านั้น
- ต้องเริ่มต้นด้วยตัวอักษร หรือ Underscore เท่านั้น
- ไม่สามารถเว้นว่างระหว่างตัวอักษร หรือเครื่องหมายได้
- ตัวอักษรเล็ก ใหญ่มีความหมายต่างกัน (Case sensitive)
- ต้องไม่เป็นคำสงวน

#### Examples of names of modules:

Counter\_4Bit

Mux\_4\_To\_1

ALU

UART\_Transmit

Receiver

### การเขียนข้อสังเกต หรือหมายเหตุใน Verilog (Comments in Verilog)

วิธีการเขียนโค้ด (Coding style) ที่ดีนั้นจำเป็นจะต้องคำนึงถึงผู้อ่านในการทำความเข้าใจส่วนต่าง ๆ ของการเขียนด้วย หรือแม้กระทั่งตัวผู้ออกแบบเองในการที่จะกลับมาตรวจสอบ แก้ไขส่วนต่าง ๆ ในภายหลัง ดังนั้นลักษณะการเขียนที่ดีนอกจากจะต้องเขียนให้อ่านง่าย เป็นสัดส่วนแล้วควรจะต้องมีคำอธิบายสั้น ๆ ในแต่ละส่วนของโปรแกรมด้วย

```
// one-line comment
module CommEx (A, B, C);
  input B, C;
  output A;
  . . .
endmodule
```

สำหรับภาษา Verilog นั้นจะอนุญาตให้เขียนหมายเหตุในตัวโปรแกรม โดยที่ตัวคอมไพเลอร์จะไม่นำไปประมวลผลแต่อย่างใด นั่นคือ

- **One-line comment:** เริ่มต้นด้วย // และจบด้วยการจบบรรทัด (end-of-line)
- **Block comment:** จะเริ่มต้นด้วย /\* จนกระทั่งถึงจุดที่มีเครื่องหมาย \*/ ทำให้สามารถกำหนดหมายเหตุได้ครอบคลุมหลาย ๆ บรรทัดเท่าที่ต้องการ

การกำหนดข้อสังเกต หรือหมายเหตุนั้นสามารถกำหนดที่ได้ก็ได้ในตัวโปรแกรม ยกเว้นกำหนดภายในคำสงวน หรือชื่อของสิ่งต่าง ๆ ที่เรารวบรวมมา เช่นชื่อของพอร์ต โมดูล เป็นต้น

### รายละเอียดของระบบ (Descriptions of a System)

การเริ่มต้นการเขียนออกแบบที่ดีนั้นควรมีส่วนหัวของไฟล์ในการอธิบายลักษณะของระบบ (System description) เพื่อให้ง่ายในการตรวจสอบ ทั้งจากผู้ออกแบบเอง หรือผู้ร่วมงานอื่น ๆ รวมทั้งผู้ใช้งานด้วย ทั้งนี้ ข้อมูลต่าง ๆ อาจประกอบด้วยส่วนต่าง ๆ เช่น ชื่อไฟล์ ลักษณะของการใช้งาน ข้อจำกัด ข้อผิดพลาดที่เกิดขึ้นแล้วได้แก้ไขในเวอร์ชันก่อน ชื่อของผู้ออกแบบ เวอร์ชันในการออกแบบ และวันที่ของการออกแบบ เป็นต้น ดังแสดงในรูป

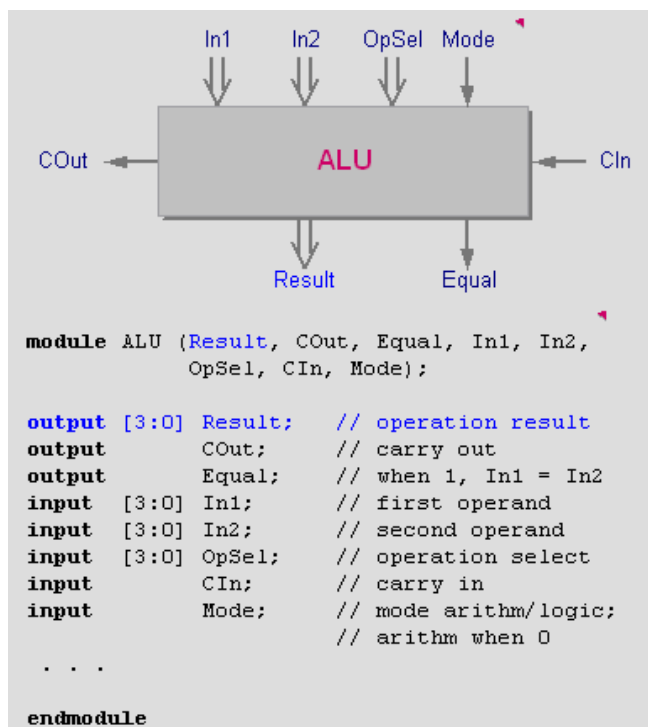
```
// Design      : 8086 (RTL)
// File name   : 8086.v
// Purpose     : model of a 8086 microprocessor for the
//              design of "system-on-chip" embedded
//              modules. Fully compliant with the
//              specification by Intel.
//
// Note        : This model can be synthesized with
//              Active-CAD tools.
//
// Limitations  : A Clk frequency of 33 MHz is assumed.
//
// Errors       : None known.
//
// Include files : none.
//
// Author      : Evita Team
//              ALDEC Inc.
//              2230 Corporate Circle,
//              Henderson, Nevada 89014
//
// Simulator    : Active-CAD
// -----
// Revision list
// Version  Author   Date      Changes
// 1.0      ET       01 Jan 98   new version
```

### ส่วน Interface ในโมดูล (The Module Interface)

ในการที่จะกำหนดส่วนของการ Interface ในตัวโมดูล จะต้องประกอบด้วยสองส่วนดังนี้

- Port list: เป็นรายชื่อของพอร์ตที่จะใช้ในตัวโมดูล ซึ่งกำหนดอยู่ในวงเล็บหลังจากชื่อของโมดูล โดยมีเครื่องหมาย Comma (,) ระหว่างพอร์ตต่าง ๆ
- Port declaration: เป็นส่วนที่กำหนดขึ้นมาเพื่ออธิบายคุณลักษณะของพอร์ตต่าง ๆ ที่อยู่ใน Port list โดยจะเป็นข้อมูลของทิศทางว่าเป็นอินพุต หรือเอาต์พุต หรือทั้งสอง และความกว้างของพอร์ต เป็นต้น

สำหรับชื่อของพอร์ตควรตั้งให้เหมาะสม เพื่อให้ง่ายในการอ่านทำความเข้าใจ หรืออาจจะเพิ่ม Comment ตามหลังแต่ละพอร์ตโดยใช้ One-line comment (//) ก็ได้ ดังแสดงในรูป



## การเขียนในส่วนของ Body (The Body of a Module)

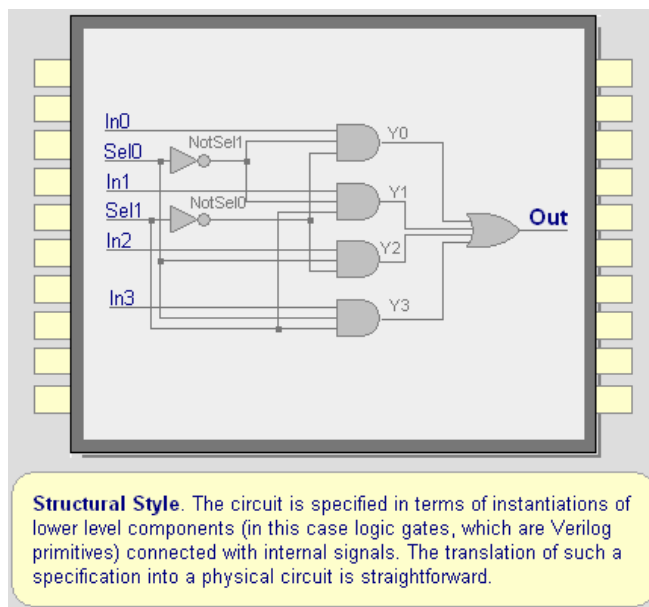
เนื่องจากความหลากหลายของวงจรที่เราต้องการออกแบบ ทั้งในด้านการทำงาน และในด้านโครงสร้าง การออกแบบ ซึ่งบางทีในวงจรชนิดเดียวกันสามารถออกแบบได้หลายวิธีเป็นต้น ดังนั้นภาษาที่สมควรจะสนับสนุนลักษณะการออกแบบ หรือลักษณะการเขียนอธิบายวงจร (Coding style) ได้หลายแบบ ทั้งแบบที่เหมาะสมสำหรับวงจรเล็ก ๆ ไม่กี่เกต แต่บางทีวิธีการนี้อาจไม่เหมาะสมกับวงจรใหญ่ ๆ ที่มีหลายพันเกต เนื่องจากทำความเข้าใจได้ยาก

สำหรับภาษา Verilog HDL สามารถมีวิธีการเขียนในการอธิบายวงจร แบ่งได้เป็น 3 ลักษณะตามความต้องการในวงจรแต่ละระดับ (Hierarchy) หรือความถนัดที่แตกต่างกันของผู้ออกแบบ ได้แก่

- Structural design: เป็นการใช้อุปกรณ์ที่มีอยู่แล้วในไลบรารีมาตรฐาน หรือที่เรียกว่า Primitives และโมดูลย่อย ๆ มาต่อกันโครงสร้าง (Structure) แบบพอร์ตต่อพอร์ต เพื่อให้เป็นระบบขึ้นมา
- Dataflow style: เป็นการส่งผ่านค่าเอาต์พุตที่ขึ้นอยู่กับค่าของอินพุต หลังจากผ่านตัวกระทำ หรือฟังก์ชันต่าง ๆ โดยสามารถเห็นเป็นการไหลของข้อมูล (Data flow) ว่าค่าในแต่ละจุดในวงจรนั้นมาจากไหน
- Behavioral style: เป็นการเขียนลักษณะการทำงาน (Behavior) ของวงจรได้โดยตรง

ตัวอย่างเช่น เราต้องการออกแบบ Multiplexer ที่มี 4 inputs (In0, In1, In2, In3) และ 1 output (Out) โดยมีสัญญาณควบคุมการเลือกคือ Sel0 และ Sel1 นั้นสามารถแสดงเป็นการออกแบบในแต่ละสไลด์ได้ดังต่อไปนี้

### MUX in structural design style



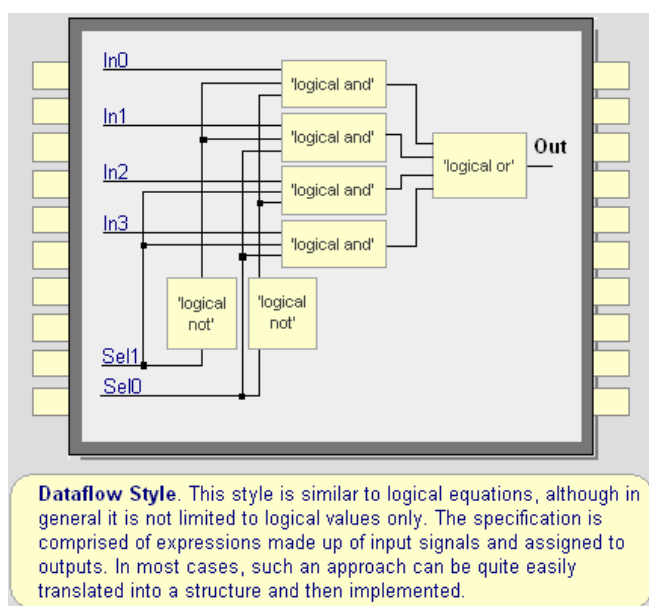
```
module mux_4_to_1 (Out, In0, In1, In2, In3, Sel1, Sel0);
output Out;
input In0, In1, In2, In3, Sel0, Sel1;

wire NotSel0, NotSel1;
wire Y0, Y1, Y2, Y3;

not (NotSel0, Sel0);
not (NotSel1, Sel1);
and (Y0, In0, NotSel1, NotSel0);
and (Y1, In1, NotSel1, Sel0);
and (Y2, In2, Sel1, NotSel0);
and (Y3, In3, Sel1, Sel0);
or (Out, Y0, Y1, Y2, Y3);

endmodule
```

### MUX in dataflow design style



จะเห็นได้ว่าวงจรสามารถกำหนดได้โดยอุปกรณ์พื้นฐานต่าง ๆ นั่นคือ And Or Not เกต ซึ่งเป็นอุปกรณ์ประเภท Primitive ในภาษา Verilog ที่สามารถเรียกใช้งานได้ทันที หรือเป็นโมดูลย่อยที่เราสร้างขึ้นมาแล้วเก็บไว้ในไลบรารี เราสามารถกำหนดการต่อเชื่อมกันระหว่างพอร์ตของอุปกรณ์พื้นฐาน หรือโมดูลย่อยนั้น ๆ ซึ่งผู้ออกแบบก็จำเป็นต้องรู้ว่า MUX นั้นจะต้องมีการต่อเชื่อมกันของอุปกรณ์ต่าง ๆ อย่างไร

การออกแบบลักษณะนี้จะเหมือนกับการออกแบบโดยสมการลอจิก (แต่อาจไม่จำกัดเพียงแค่ลอจิก '0' หรือ '1' เท่านั้น) ข้อกำหนดต่าง ๆ ในวงจรจะสร้างมาจากการสมการการกำหนด (Assign) อินพุตไปยังเอาต์พุตผ่านฟังก์ชันต่าง ๆ ที่อาจเป็นตัวกระทำทางลอจิก (Logical operator) หรือตัวกระทำทางคณิตศาสตร์ (Mathematical operator) ต่าง ๆ เป็นต้นแทนที่จะเป็นการต่อกันของอุปกรณ์ต่าง ๆ ทำให้วิธีการนี้ง่ายในการดูทำความเข้าใจ

```

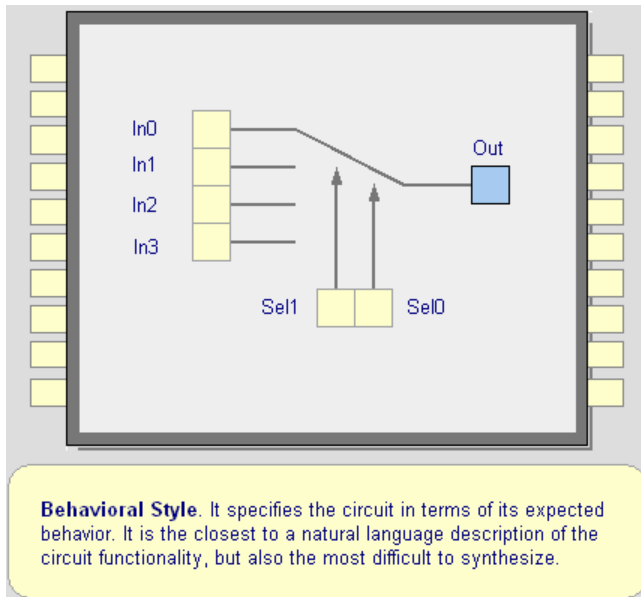
module mux_4_to_1 (Out, In0, In1, In2, In3, Sel1, Sel0);
output Out;
input In0, In1, In2, In3, Sel0, Sel1;

assign Out = (~Sel1 & ~Sel0 & In0) | (~Sel1 & Sel0 & In1)
             | (Sel1 & ~Sel0 & In2) | (Sel1 & Sel0 & In3);

endmodule

```

### MUX in behavioral design style



```

module mux_4_to_1 (Out, In0, In1, In2, In3, Sel1, Sel0);
output Out;
input In0, In1, In2, In3, Sel0, Sel1;
reg Out;

always @(Sel1 or Sel0 or In0 or In1 or In2 or In3)
begin
    case ({Sel1, Sel0})
        2'b00 : Out = In0;
        2'b01 : Out = In1;
        2'b10 : Out = In2;
        2'b11 : Out = In3;
        default : Out = 1'bx;
    endcase
end

```

การออกแบบจะเป็นการอธิบายการทำงานของ MUX มากกว่าว่าจะทำงานได้อย่างไร ในการที่จะส่งผ่านอินพุตไปยัง เอาท์พุท จากเงื่อนไขตัวเลือก Sel0 Sel1 โดยใช้คำสั่ง Case เป็นต้น ซึ่งการออกแบบลักษณะนี้ไม่จำเป็นต้องรู้ถึงโครงสร้างวงจร ทำให้ง่าย และยืดหยุ่นมาก แต่ต้องรู้วิธีการอธิบายการทำงานวงจร การอธิบายจำเป็นต้องขึ้นอยู่กับไวยากรณ์ และคำสั่งของภาษาชั้นสูงที่ใช้ ดังนั้นจึงจำเป็นต้องมีเครื่องมือในการสังเคราะห์วงจร (Synthesis tools) ในการแปล และสร้างวงจรขึ้นมา

# CHAPTER 3

## Signals in Verilog

ทุกอย่างในวงจรอิเล็กทรอนิกส์ สามารถมองได้เป็นลักษณะของการรับ การแปลง และการส่งสัญญาณทางไฟฟ้า (Receiving, transforming, and sending signals) จึงไม่แปลกใจว่าสัญญาณเป็นสิ่งที่สำคัญในระบบอิเล็กทรอนิกส์ ดังนั้นในบทนี้จะพูดถึงเรื่องสัญญาณเป็นสำคัญ และวิธีการที่เราจะเรียกใช้ในภาษา Verilog โดยจะเริ่มแนะนำให้รู้จักกับนิยาม และลักษณะต่าง ๆ ของสัญญาณ รวมทั้งหน้าที่ของสัญญาณที่มีอยู่ในวงจรดิจิทัล

ลำดับต่อมาจะเป็นเรื่องของข้อกำหนดต่าง ๆ ในการกำหนดสัญญาณในภาษา Verilog ซึ่งโดยทั่วไปแล้วสัญญาณจะมีอยู่สองลักษณะคือ nets และ registers แต่ในบทนี้จะพูดถึง nets เท่านั้น เนื่องจาก registers จะต้องเกี่ยวข้องกับการอธิบายวงจรแบบ Behavioral ซึ่งอยู่ในบทต่อไป

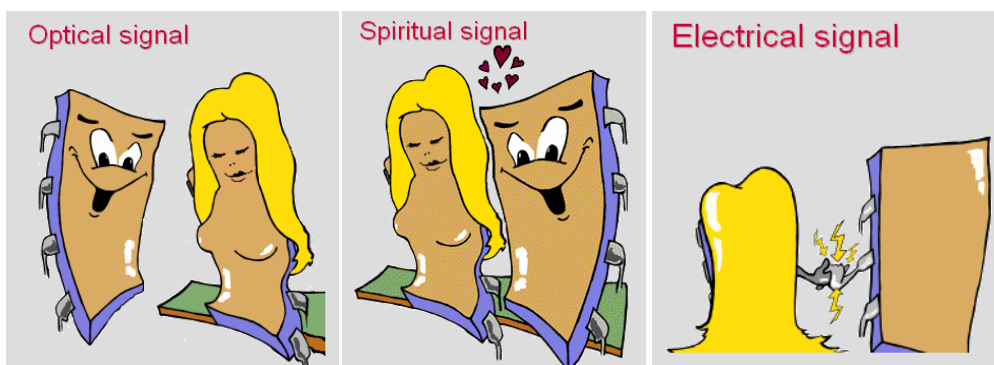
จากที่กล่าวมาในเบื้องต้นถึงลักษณะของโมดูลว่า สามารถแบ่งออกเป็นสองส่วนหลัก ๆ คือ Body และ Interface ดังนั้นลักษณะของสัญญาณก็เช่นกันสามารถแบ่งได้เป็น Internal ที่ใช้ในส่วนของ Body และ External ที่ใช้ในส่วนของ Interface ที่เป็นการกำหนดการทำงานของพอร์ต

### เกริ่นนำ (Introduction to Signals)

#### แนวคิดเกี่ยวกับสัญญาณ (Concept of Signals)

สิ่งหนึ่งที่เป็นปรากฏการณ์สำคัญในโลกเราก็คือ การติดต่อสื่อสาร ซึ่งทำให้เราสามารถแชร์ข้อมูลระหว่างกัน ซึ่งเป็นกระบวนการในการแลกเปลี่ยน มีการเกี่ยวข้องกับการส่งข้อมูลจากผู้ส่งไปยังปลายทางที่เป็นผู้รับ และถึงแม้ว่าจะมีวิธีการหลาย ๆ วิธีการในการรับส่งข้อมูลนี้ แต่มีสิ่งหนึ่งที่ร่วมกันคือ สัญญาณ หรือ Signals

สัญญาณสามารถมีได้หลาย ๆ รูปแบบ เช่น อยู่ในรูปของแสง (Optical) เสียง (vocal) ทางกล (mechanical) หรือทางไฟฟ้า (Electrical) เป็นต้น แต่ไม่ว่าจะอยู่ในรูปใดก็ตาม สัญญาณจะเป็นตัวที่มีข้อมูลบรรจุอยู่เสมอ



#### สัญญาณในอุปกรณ์อิเล็กทรอนิกส์ (Signals in Electronic Device)

การติดต่อสื่อสารไม่ได้จำกัดเฉพาะสิ่งมีชีวิตเท่านั้น เครื่องจักร หรือเครื่องมือต่าง ๆ ก็สามารถทำได้เช่นกัน ตัวอย่างเช่น ตัววัดอุณหภูมิ (Temperature sensor) อาจบอกวงจรควบคุมเมื่อมีอุณหภูมิเกินระดับที่กำหนดไว้ ให้วงจรควบคุมทำการสั่งแอร์ให้ทำงาน เป็นต้น



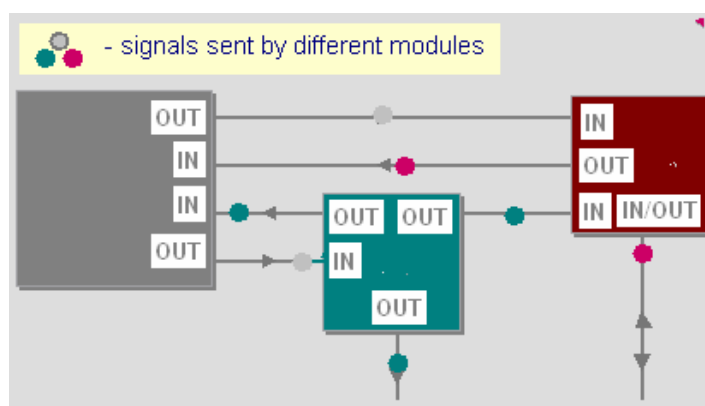
สัญญาณที่ส่งผ่านระหว่างอุปกรณ์ หรือเครื่องมือต่าง ๆ สามารถมีอยู่ได้หลาย ๆ แบบแต่ที่พบได้บ่อยที่สุดก็คือสัญญาณที่เกี่ยวข้องกับวงจรไฟฟ้า ดังนั้นทุกสัญญาณจากสภาวะแวดล้อมภายนอกจำเป็นต้องถูกแปลงเป็นสัญญาณทางไฟฟ้า (Electrical signals) เพื่อที่จะนำไปใช้งาน หรือประมวลผลด้วยตัวอุปกรณ์ทางไฟฟ้าได้

### ความสำคัญของสัญญาณทางไฟฟ้า (The Importance of Electrical Signals)

สัญญาณทางไฟฟ้าเป็นสิ่งสำคัญในการทำงานในวงจร หรืออุปกรณ์อิเล็กทรอนิกส์ ซึ่งความจริงแล้วถ้าปราศจากสัญญาณไฟฟ้า อุปกรณ์ต่าง ๆ เหล่านั้นไม่สามารถถูกเรียกได้ว่าเป็นอุปกรณ์อิเล็กทรอนิกส์ได้นั่นเอง

ถ้าพิจารณาทางกายภาพ สัญญาณทางไฟฟ้าคือการไหลของอิเล็กตรอน และโปรตรอน ซึ่งเป็นการมองในทางฟิสิกส์นั่นเอง แต่ทางภาษาชั้นสูงอย่าง Verilog เราไม่จำเป็นต้องไปกังวลถึง เพียงแต่ทำการกำหนดหน้าที่การทำงานของวงจร โดยคำนึงถึงค่า หรือความหมายของสัญญาณทางไฟฟ้าที่จะเกิดขึ้นเท่านั้นเอง โดยจะมีตัวสังเคราะห์วงจร (Synthesis tools) ที่ทำการสร้างวงจรขึ้นมาให้รับ และสร้างสัญญาณไฟฟ้าตามที่เราต้องการ

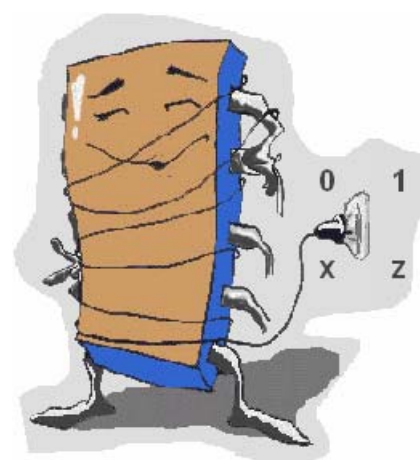
จากนี้ไปสัญญาณไฟฟ้าจะถูกเรียกสั้น ๆ ว่าสัญญาณ ซึ่งไหลไปตามเส้นทางที่ต่อเชื่อมกันระหว่างอุปกรณ์อิเล็กทรอนิกส์ หรือโมดูลต่าง ๆ ตามลักษณะของทิศทางที่ถูกกำหนดขึ้นมาโดยชนิดของพอร์ตที่ตัวอุปกรณ์ ซึ่งอาจเป็นอินพุต เอาท์พุตพอร์ต หรือพอร์ตที่เป็นได้ทั้งอินพุต/เอาท์พุต



### ค่าที่เป็นไปได้ของสัญญาณ (Available Values of Signals)

ค่าของสัญญาณที่เป็นไปได้ในภาษา Verilog มีเพียงแค่ 4 ค่าตามลักษณะของสัญญาณ และความเป็นไปได้จริง ๆ ในวงจรอิเล็กทรอนิกส์ ซึ่งได้แก่ '0', '1', 'X', 'Z' โดยที่

- '0' คือค่าลอจิกศูนย์ (Logic zero) หรือลอจิกต่ำ (Logic low) หรือค่าที่ไม่จริง (False condition) จากการเปรียบเทียบ
- '1' คือค่าลอจิกหนึ่ง (Logic one) หรือลอจิกสูง (Logic high) หรือค่าที่เป็นจริง (True condition) จากการเปรียบเทียบ
- 'x' หรือ 'X' เป็นลอจิกไม่ทราบค่า (Logic unknown) ที่อาจเกิดจากการไม่ลงรอยกัน (Conflict) ของสัญญาณที่จุด ๆ นั้นที่อาจเป็นไปได้ทั้ง '0' '1' 'Z'
- 'z' หรือ 'Z' เป็นลอจิกที่เกิดจากการเปิดวงจร (Open circuit) ทำให้เกิดสภาวะ High impedance ณ จุดนั้น ๆ





สังเกตได้ว่า ถึงแม้ภาษา Verilog จะเป็น Case sensitive แต่สำหรับสภาวะ unknown หรือ high impedance สามารถใช้ได้ทั้งตัวเล็ก และตัวใหญ่

### การกระทำทางลอจิกสำหรับค่าของสัญญาณต่าง ๆ (Logic Operations on Four-value Signals)

วงจรทางดิจิทัลที่ซับซ้อนเพียงใด เราสามารถแบ่งแยกย่อยออกมาเป็นลอจิกเกตต่าง ๆ (สามารถกำหนดได้เป็นสมการบูลีน) และเราใช้ตารางค่าความจริง (Truth table) สำหรับแสดงความสัมพันธ์กันระหว่างอินพุต และเอาต์พุต โดยที่สัญญาณทั้งอินพุต และเอาต์พุตสามารถเป็นไปได้อีกทั้ง 4 ค่าทางสัญญาณไฟฟ้า นั่นคือ '0' '1' 'X' 'Z'

ตัวอย่างของตารางค่าความจริงของ AND และ OR เกต แสดงได้ดังรูป

 and	and	0	1	x	z
	0	0	0	0	0
	1	0	1	x	x
	x	0	x	x	x
	z	0	x	x	x

 or	or	0	1	x	z
	0	0	1	x	x
	1	1	1	1	1
	x	x	1	x	x
	z	x	1	x	x

### สัญญาณในภาษา Verilog (Signals in Verilog)

#### ประเภทของสัญญาณ (Class of Signals)

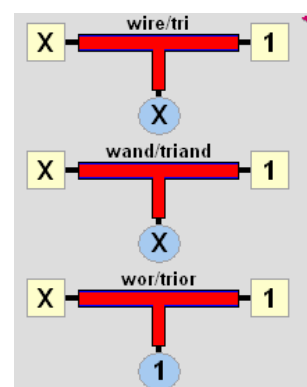
แต่ละสัญญาณในภาษา Verilog สามารถจัดให้อยู่ในสองประเภทของสัญญาณ คือ nets และ registers

- Nets แทนการเชื่อมต่อระหว่างส่วนต่าง ๆ ของวงจร มันไม่สามารถเก็บ (Storage) ค่าของสัญญาณไว้ได้ และค่าที่ปรากฏบน nets กำหนดได้โดยตัวขับ (Driver) หรือแหล่งจ่าย (Source) ดังนั้นเมื่อตัดแหล่งจ่ายออกไปจาก nets ก็เกิดสภาวะ High impedance 'Z' (ไม่ได้ต่อกับแหล่งจ่ายใด ๆ ) เกิดขึ้นที่ nets
- Registers ต่างกับ nets คือสามารถเก็บค่าของสัญญาณได้ โดยยังคงค่าเดิมเมื่อไม่ได้ต่อกับแหล่งขับ ค่าที่ถูกกำหนดก่อนหน้านี้ สามารถถูกแทนได้ด้วยสัญญาณใหม่ที่เพิ่งเข้ามา ดังนั้นสัญญาณประเภท registers ก็คล้ายกับรีจิสเตอร์ หรือฟลิปฟล็อป (Flip-flop) ที่ใช้งานในวงจรดิจิทัลทั่วไป แต่มันต่างกันตรงที่ไม่จำเป็นต้องใช้สัญญาณนาฬิกา (Clock) ในการเก็บค่า

#### Nets

ในภาษา Verilog สัญญาณที่จัดอยู่ในประเภท nets ประกอบด้วย

- **wire** และ **tri** : เป็น nets ที่ใช้กันมากที่สุด ชื่อที่ต่างกันนี้ใช้ตามลักษณะการใช้งานนั่นคือ wire ควรจะในจุดที่มีตัวขับตัวเดียว (Single drive nets) ส่วน tri ควรใช้สำหรับจุดที่มีหลาย ๆ ตัวขับ
- **wand/triand** และ **wor/trior** : เป็น nets ที่มีการกระทำทางลอจิก And หรือ Or ตามลำดับ
- **supply0, supply1, tri0, tri1, trireg** : เป็น nets ที่มีลักษณะการใช้งานพิเศษสำหรับการออกแบบในระดับล่าง (Low level) เช่น ทรานซิสเตอร์



ตารางค่าความจริงของ Nets แต่ละชนิดสามารถแสดงเป็นตารางค่าความจริงได้ดังนี้

wire/ tri	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

wand/ triand	0	1	X	Z
0	0	0	0	0
1	0	1	X	1
X	0	X	X	X
Z	0	1	X	Z

wor/ trior	0	1	X	Z
0	0	1	X	0
1	1	1	1	1
X	X	1	X	X
Z	0	1	X	Z

### การกำหนดสัญญาณ (Signal Specification)

ก่อนที่จะทำการใช้ประเภทของสัญญาณในภาษา Verilog เราจำเป็นต้องมีการกำหนด (Declaration) ชนิด และชื่อก่อน ในที่นี้จะพูดถึงการกำหนดสัญญาณประเภท internal ที่อยู่ในส่วนของ body ของโมดูลเท่านั้น ส่วนสัญญาณประเภท external ที่อยู่ในส่วนของการเชื่อมต่อ (Interface) จะพูดถึงในส่วนต่อไป

กฎ หรือไวยากรณ์ของการกำหนดการใช้สัญญาณสามารถทำได้โดยง่ายคือ บอกชนิดของสัญญาณก่อนว่าเป็น **wire**, **tri**, **wand**,... แล้วตามด้วยชื่อ โดยอาจใส่พร้อม ๆ กันหลาย ๆ ชื่อได้โดยใส่เครื่องหมาย , (comma) กั้นระหว่างชื่อ เช่น **wire a, b, c, d;** เป็นต้น

สิ่งที่สะดวกอย่างหนึ่งของภาษา Verilog ก็คือเราสามารถกำหนดการใช้สัญญาณได้ทุกที่ บรรทัดใดก็ได้ในส่วน of body แต่ต้องกำหนดไว้ก่อนการใช้งาน

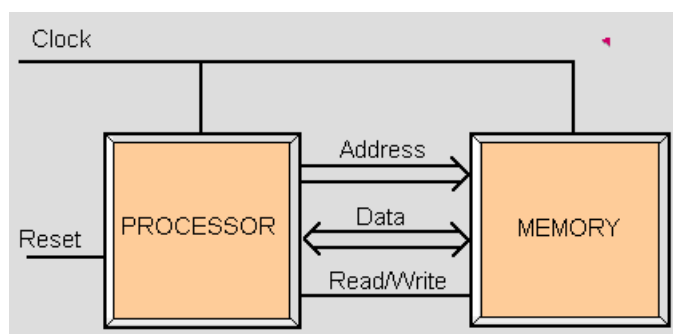
### สัญญาณสเกลลาร์ และเวกเตอร์ (Scalar Signals and Vectors)

ถึงตอนนี้เราได้พูดถึงเฉพาะสัญญาณที่มีเส้นเดียว หรือบิตเดียว ที่เราอาจเรียกว่า Scalar signal ซึ่งมีค่าสัญญาณทางลอจิกค่าเดียว ณ เวลาใดเวลาหนึ่ง ตัวอย่างเช่น สัญญาณนาฬิกาที่ใช้เป็นจังหวะ (Synchronize) ของการทำงานในระบบดิจิทัล

หลาย ๆ ระบบจำเป็นต้องมีกลุ่มของสัญญาณหลาย ๆ เส้นที่เรียกว่า บัส หรือเวกเตอร์ (Buses or vectors) ซึ่งจะทำให้การรับส่งข้อมูลที่ประกอบด้วยค่าทางลอจิกที่อยู่ในแต่ละเส้นหลาย ๆ ค่า ตัวอย่างที่มักจะเห็นกันเป็นประจำก็คือ ระบบ

ไมโครโปรเซสเซอร์ เมื่อได้ยี่ห้อถึง 32-bit microprocessor นั้นหมายถึงว่า

บัสข้อมูลมีขนาด 32 bit หรือ 32 เส้น แต่ละบิตในบัสสามารถเข้าถึงได้ทั้งการอ่าน และการเขียน โดยใช้ตัวชี้ (Index) เช่น data[31:0], address[31:0]

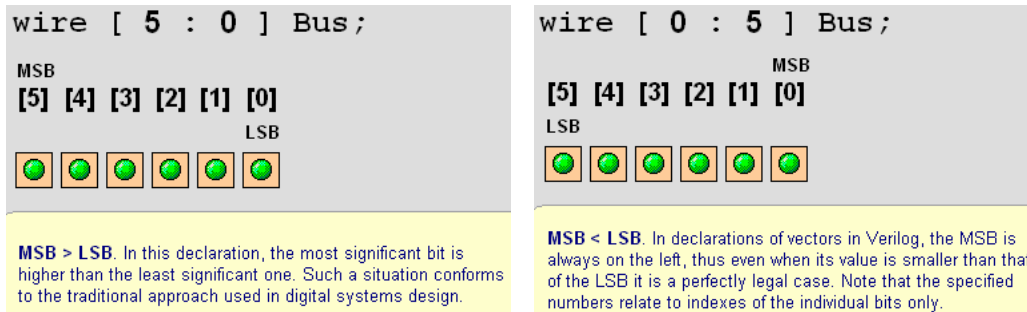


### การกำหนดการใช้งานแบบเวกเตอร์ (Vector Specification)

ในการกำหนดการใช้งานแบบเวกเตอร์ เราไม่จำเป็นต้องมีชนิดของสัญญาณขึ้นมาใหม่สำหรับเวกเตอร์ แท้จริงแล้ว สัญญาณสเกลลาร์นั้นอาจถือได้ว่าเป็นกรณีเฉพาะของสัญญาณแบบเวกเตอร์ที่มีเส้นเดียวโดยที่บิตสำคัญมากที่สุด MSB (Most significant bit) กับบิตสำคัญน้อยสุด LSB (Least significant bit) นั้นอยู่ในตำแหน่งเดียวกัน

เมื่อสัญญาณแบบเวกเตอร์ประกอบด้วยหลาย ๆ บิต ซึ่งในแต่ละบิตจะต้องสามารถเข้าถึงได้โดยการอาศัยตัวชี้ (Index) สำหรับภาษา Verilog นั้นเราสามารถกำหนดตัวเลขชี้บิต (Indexing number) ที่ใช้ได้ทั้ง

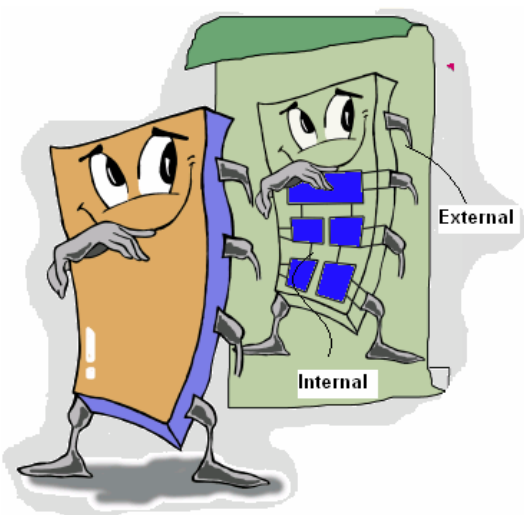
เลขลบ และเลขบวก กำหนดลงใน [ ] โดยที่ตัวเลขชี้บิตจะมีอยู่สองตัวสำหรับการกำหนดช่วงโดยจะถูกคั่นด้วยเครื่องหมาย : (Colon) สำหรับตัวเลขด้านซ้ายจะเป็น MSB และตัวเลขด้านขวาคือ LSB ตัวอย่างเช่น wire [0:5] Bus1; wire [5:0] Bus2; wire [5:-5] Address; เป็นต้น



## สัญญาณภายนอก (External Signals)

### สัญญาณ Internal เทียบกับ External (Internal vs External Signals)

สัญญาณที่พูดถึงก่อนหน้านี้ที่อยู่ในประเภท Internal ที่ใช้ภายในโมดูลในส่วนของ Body และไม่สามารถเข้าถึง (Accessible) ได้จากภายนอกโมดูลถ้าปราศจากสัญญาณ External ที่อยู่ในส่วนของการ Interface ดังนั้นความแตกต่างระหว่างสัญญาณ Internal และ External คือ สัญญาณ Internal จะใช้ภายในตัวโมดูลสำหรับการรับส่งข้อมูลต่าง ๆ ส่วนสัญญาณ External จะเป็นตัวที่ติดต่อกับอุปกรณ์ต่าง ๆ ที่อยู่ภายนอกโมดูล โดยจะกำหนดเป็นพอร์ตของโมดูล (Module port) นั่นเอง สำหรับการรับข้อมูลอินพุตเข้ามา หรือส่งข้อมูลเอาต์พุตออกไป

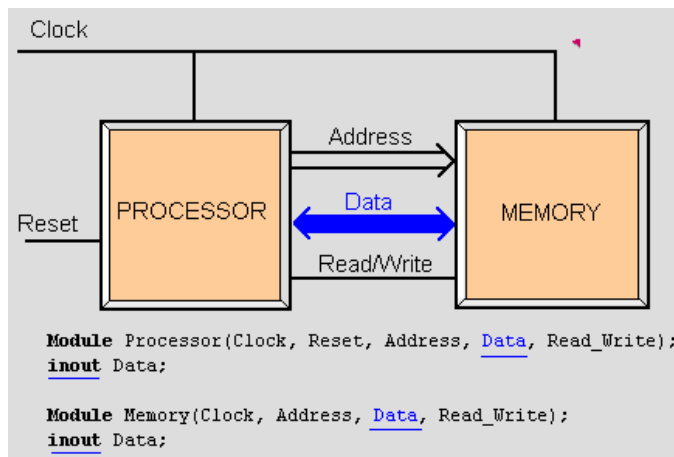


### พอร์ตของโมดูล (Module Ports)

โมดูลสามารถสื่อสารระหว่างภายใน และภายนอกผ่านทางพอร์ต ในการกำหนดการใช้งานพอร์ตนั้น จะต้องกำหนดความกว้างของพอร์ต และทิศทางของข้อมูลที่จะผ่านพอร์ต (เข้าหรือออก เทียบกับตัวโมดูลนั้น ๆ )

ในภาษา Verilog เราสามารถกำหนดทิศทางของพอร์ตได้เป็น

- input: ข้อมูลจะถูกอ่านโดยโมดูลจากภายนอกผ่านทาง input ports เราไม่สามารถเขียนข้อมูลจากภายในโมดูลไปยังพอร์ตชนิดนี้ได้
- output: ข้อมูลจะถูกส่งไปยังภายนอกโมดูลผ่านทาง output ports เราไม่สามารถอ่านข้อมูลเข้ามายังภายในโมดูลผ่านทางพอร์ตชนิดนี้ได้
- inout: ข้อมูลสามารถถูกอ่าน และเขียนผ่านทางพอร์ตชนิดนี้ได้ ดังนั้นจึงถูกเรียกว่า bi-directional ports



สำหรับตัวอย่างในรูปจะแสดงการกำหนดการใช้งาน Bi-directional port สำหรับเป็นพอร์ตข้อมูลระหว่าง Processor กับ Memory โดยที่ข้อมูลสามารถไปกลับได้สองทิศทาง คือจากซ้ายไปขวา หรือจากขวาไปซ้าย

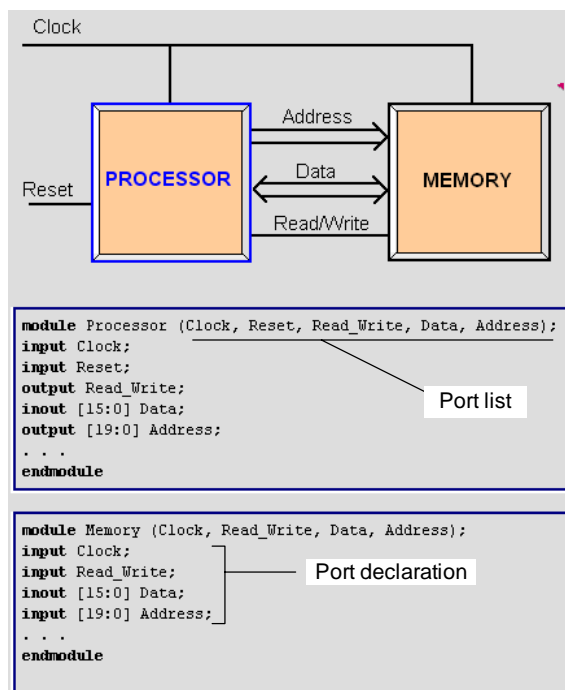
### การกำหนดการใช้งานพอร์ต (The Specification of Ports)

การกำหนดการใช้งานพอร์ตในภาษา Verilog ประกอบด้วยสองส่วนดังนี้

- ชื่อของพอร์ตที่ถูกเรียงอยู่ในส่วนที่เรียกว่า Port list ที่ตามหลังชื่อของโมดูล รายชื่อของพอร์ตต่าง ๆ จะอยู่ในวงเล็บ โดยถูกคั่นด้วยเครื่องหมาย , (comma)
- หลังจากส่วนของ Port list จะต้องตามด้วย Port declaration ที่เป็นตัวกำหนดทิศทางของพอร์ต รวมทั้งขนาดของพอร์ตแต่ละพอร์ต การกำหนดการใช้งานในส่วนนี้จะคล้ายกับการกำหนดการใช้งานของ Internal signals โดยมีไวยากรณ์ดังนี้

*Keyword*      *vector\_range*      *identifier*;

โดยที่ *keyword* คือ *input*, *output* หรือ *inout*

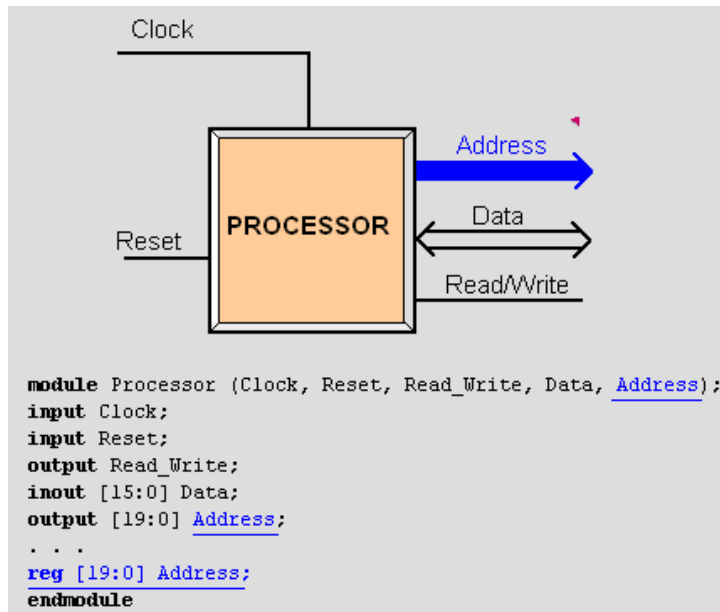


ตัวอย่างในรูปแสดงการกำหนดการใช้งานพอร์ตในโมดูล Processor และโมดูล Memory ซึ่งเราจะเห็นส่วนของ Port list ตามด้วย Port declaration เช่นในกรณีของ Processor ที่มีพอร์ตต่าง ๆ คือ Clock, Reset, Read\_Write, Data และ Address กำหนดไว้ใน Port list ภายใต้วงเล็บที่อยู่หลังชื่อโมดูล บรรทัดหลาย ๆ บรรทัดต่อมาจะเป็นส่วนของ Port declaration ที่เป็นกำหนดทิศทาง และขนาดของสัญญาณ ก่อนที่จะนำไปใช้งานต่อไป

### การสร้างรีจิสเตอร์ที่เอาต์พุต (Registered Outputs)

ในการออกแบบระบบใหญ่ ๆ ที่ดีนั้น เรามักจะกำหนดให้ค่าเอาต์พุตของแต่ละโมดูลมีการเก็บค่าไว้ด้วยรีจิสเตอร์ ทั้งนี้เพื่อให้ง่ายในการเชื่อมต่อ และได้ค่าที่เสถียรก่อนที่จะนำไปใช้งาน หรือเป็นอินพุตของโมดูลอื่น ๆ เนื่องจากเราทราบว่าถ้าเอาต์พุตต่อโดยตรงกับสัญญาณประเภท Nets เช่น wire นั้นจะมีสถานะเป็น High impedance เมื่อไม่มีตัวขับ หรือแหล่งจ่ายต่ออยู่ ซึ่งสถานะนี้ไม่เป็นที่ต้องการในการใช้งาน

วิธีการที่จะทำให้เป็นเอาต์พุตแบบรีจิสเตอร์ สามารถทำได้ง่าย ๆ โดยกำหนด Internal signals แบบ **reg** (Register) และให้สัญญาณเป็นชื่อเดียวกันกับเอาต์พุตพอร์ต ดังตัวอย่าง



# CHAPTER 4

## A Structural View of a System

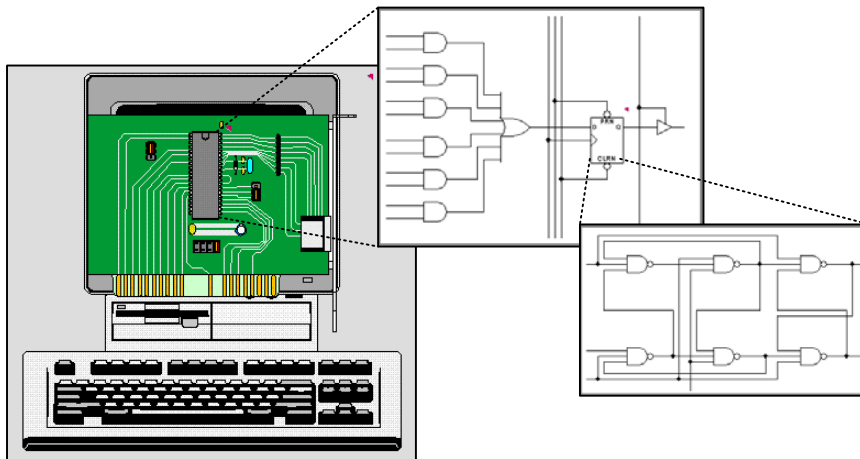
หลังจากที่ทราบถึงการกำหนดการใช้งานของสัญญาณที่เป็นทั้ง External และ Internal แล้ว ในบทนี้จะเป็นการแนะนำให้รู้จักการเขียนในส่วนที่เป็น Body และวิธีการสร้างระบบที่ซับซ้อนจากส่วนประกอบย่อย ๆ ที่อาจเป็นโมดูล หรืออุปกรณ์พื้นฐาน (Primitive) ต่าง ๆ ทั้งที่เป็นมาตรฐานที่สามารถเรียกใช้ได้เลย เช่น ลอจิกเกตต่าง ๆ หรือที่เราสร้างขึ้นมาเองที่เรียกว่า UDP (User defined primitive) โดยในการเชื่อมต่อนั้นจะใช้ nets เป็นตัวเชื่อมต่อ

### อุปกรณ์พื้นฐานใน Verilog (Verilog Primitives)

#### วิธีการกำหนดออกแบบโครงสร้างเบื้องต้น (Introduction to Structural Specifications)

เราสามารถมองได้ว่า ทุก ๆ ระบบประกอบด้วยโมดูลหลาย ๆ โมดูล ที่ถูกเชื่อมต่อกันสำหรับการติดต่อสื่อสารระหว่างกัน ในแต่ละโมดูลนี้ สามารถมีโมดูลย่อย ๆ ภายในตัวมันเองที่ต่อเชื่อมกัน ลักษณะการออกแบบระบบแบบนี้เราเรียกว่า Hierarchy หรือลำดับชั้นนั่นเอง

การออกแบบระบบโดยดูจากโครงสร้างนั้น มีลักษณะคล้ายกับการมองระบบในทางกายภาพที่ประกอบด้วยส่วนต่าง ๆ และส่วนต่าง ๆ เหล่านั้นก็สามารถแยกย่อยลงไปอีก ดังตัวอย่างในรูปของระบบคอมพิวเตอร์ที่ประกอบด้วยส่วนต่าง ๆ เช่น การ์ด และในตัวการ์ดเองก็ประกอบด้วย IC โดยที่ใน IC ก็มีเกตอยู่ภายใน รวมทั้งฟลิปฟล็อป และในตัวฟลิปฟล็อปเองก็มีเกตต่อกันอยู่ ซึ่งถ้ามองลึกลงไปอีกถึงขั้นล่างสุด เกตก็จะประกอบด้วยตัวทรานซิสเตอร์ต่อกัน



สำหรับในตัวยานภาษา Verilog นั้น เราสามารถออกแบบเป็นลักษณะโครงสร้างได้ จากระดับล่างที่เป็นตัวทรานซิสเตอร์ จนกระทั่งเป็นโมดูลย่อย ๆ รวมกันเป็นโมดูลหลัก และประกอบกันขึ้นมาเป็นระบบที่ซับซ้อนได้ในที่สุด

เนื่องจากลอจิกเกต เป็นอุปกรณ์พื้นฐานที่สุด และจำเป็นที่สุดที่ประกอบกันเป็นวงจรดิจิทัลขึ้นมา และใน Verilog เองก็มีอยู่ในไลบรารี Primitive ซึ่งสามารถเรียกใช้ได้ทันที ดังนั้นเราจะมาทำความรู้จักกับอุปกรณ์พื้นฐานเหล่านี้ และวิธีการเรียกใช้งานด้วยภาษา Verilog ก่อน

## เกตพื้นฐานที่สร้างไว้แล้ว (Predefined Gate Primitives)

ใน Verilog เราสามารถเรียกใช้เกตพื้นฐานที่มีอยู่แล้วได้ 14 ชนิดด้วยกัน ทั้งหมดนี้เราสามารถแบ่งออกได้เป็น 4 กลุ่มด้วยกันคือ

- Multiple-input gates ประกอบด้วย 6 ลอจิกเกตด้วยกันคือ **and**, **nand**, **or**, **nor**, **xor**, และ **xnor** แต่ละตัวนั้นเป็นตัวกระทำทางลอจิก โดยสามารถกำหนดอินพุตได้มากเท่าไรก็ได้ โดยแต่ละตัวมีการทำงานสำหรับสองอินพุตตามตารางค่าความจริงดังต่อไปนี้

and

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

nand

nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

or

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

nor

nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

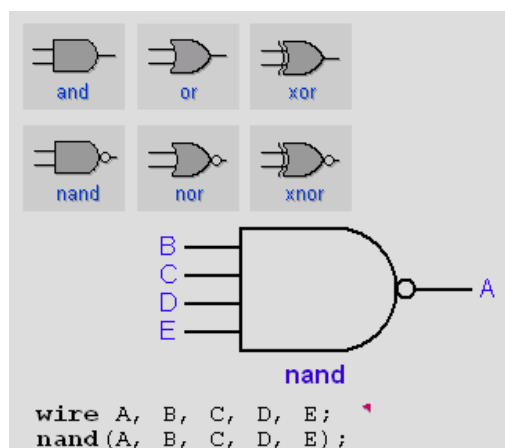
xor

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

xnor

xnor	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

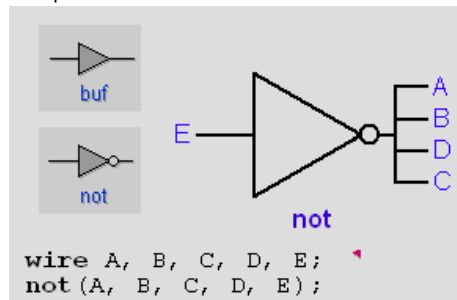
โดยมีตัวอย่างการใช้งานดังนี้



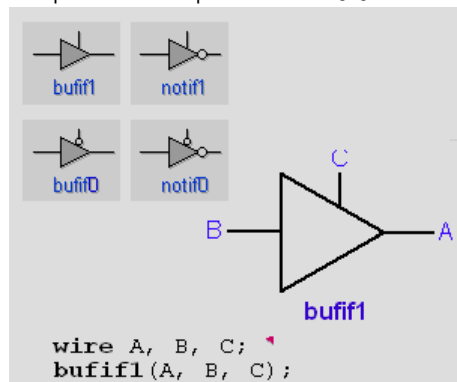
สังเกตว่า พอร์ตแรกจะเป็นเอาต์พุตพอร์ต ตามด้วยแต่ละอินพุตพอร์ตตามจำนวนที่ต้องการ



- Multiple-output gates ได้แก่ **buf** และ **not** โดยที่สามารถมีได้หลาย ๆ เอาต์พุต จากหนึ่ง อินพุต ตัวอย่างเช่น



- Tri-state gates ได้แก่ **bufif0**, **bufif1**, **notif0**, **notif1** ใช้แทนบัฟเฟอร์แบบสามสถานะ (Tri-state buffer) ได้แก่ '0', '1', และ 'Z' โดยมีสัญญาณควบคุมการปิดเปิดเกต เกตชนิดนี้มีหนึ่ง อินพุต หนึ่งเอาต์พุต และหนึ่งสัญญาณควบคุม ตัวอย่างเช่น



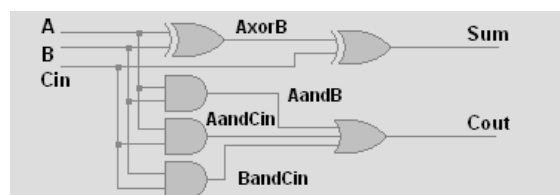
- Pull gates สามารถหาข้อมูลได้จากข้อมูลอ้างอิงในภาษา Verilog

### การออกแบบโดยใช้อุปกรณ์พื้นฐาน (Design with Primitives)

สำหรับการออกแบบประเภทนี้ อันดับแรกเรา จำเป็นต้องกำหนดชื่อของเกต (Gate instance declaration) และการเชื่อมต่อของสัญญาณที่ขาของ เกต (Interconnection signal declaration)

การกำหนดการใช้งานเกตนั้นจะต้องกำหนดชื่อ (Primitive name) และรายการของสัญญาณที่ขาต่าง ๆ (Signal list) อีกอย่างคืออุปกรณ์พื้นฐานแต่ละตัว สามารถมีข้อมูลเพิ่มเติมต่าง ๆ ขึ้นมา เช่น เกตดีเลย์ (Propagation delay) ความแรงของการขับ (Drive strength) จำนวนของอุปกรณ์ที่ต่อเรียงกันแบบ อาร์เรย์ เป็นต้น

ตัวอย่างของการสร้างวงจร 1-bit full adder จาก เกตพื้นฐานต่าง ๆ แสดงได้ดังนี้ โดยที่ค่าดีเลย์ ของ



```

module FullAdd_1Bit (Sum,Cout,A,B, Cin);
output Sum, Cout;
input A, B, Cin;
wire AxorB, AandB, AandCin, BandCin;

xor #5 HalfSum (AxorB, A, B);
and #4 (AandB, A, B);
xor FullSum (Sum, AxorB, Cin);
and #4 And_2 (AandCin, A, Cin);
and #4 And_3 (BandCin, B, Cin);
or (Cout, AandB, AandCin, BandCin);

endmodule

```

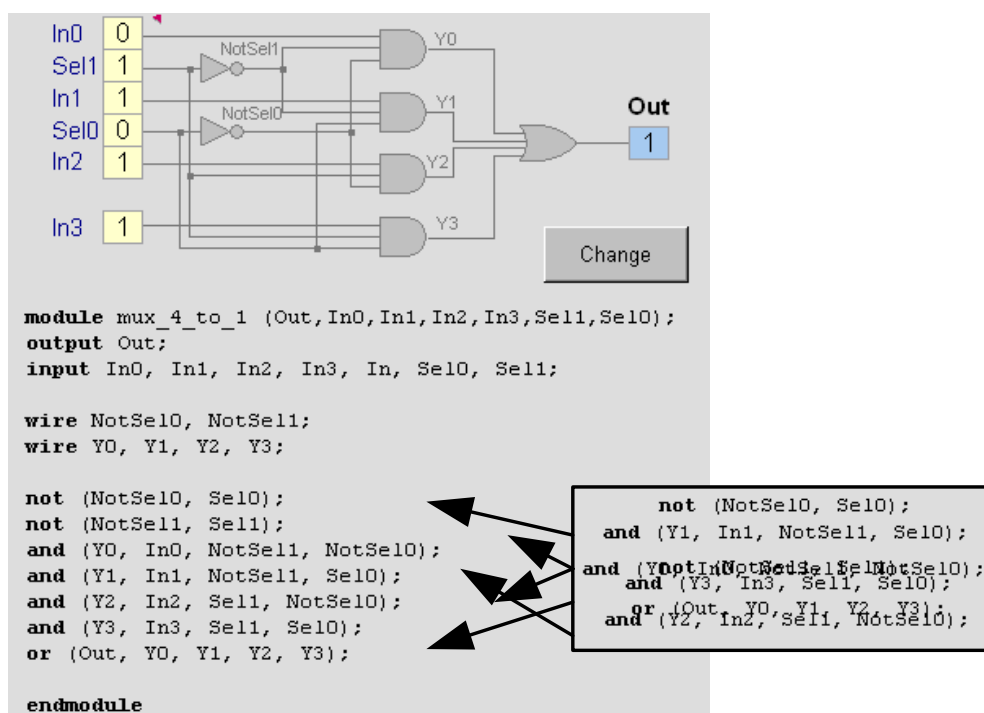
The example shows a 1-bit full adder. Note that although the code is syntactically and semantically correct, the style used here is for illustration purposes only and should not be followed.

แต่ละตัวสามารถกำหนดได้เป็นตัวเลขหลังเครื่องหมาย # และในวงจรประกอบด้วย XOR 2 ตัว — ที่มีชื่อ (Instance name) เป็น HalfSum และ FullSum — AND 3 ตัว และ OR 1 ตัว เชื่อมต่อกันด้วย wire (internal nets) คือ AxorB, AandB, AandCin และ BandCin ตามจุดต่าง ๆ ในวงจร

### การวางลำดับของตัวอุปกรณ์ (The Order of Instances)

ในวงจรที่ใช้งานจริงนั้น เราจะเห็นได้ว่าแต่ละตัวสามารถทำงานได้พร้อม ๆ กัน นั่นคือถ้ามีการเปลี่ยนแปลงที่สัญญาณอินพุตเมื่อไหร่ ค่าของเกตตัวนั้น ๆ ก็จะมีการปรับปรุงตามอินพุตที่เข้ามาในขณะนั้น โดยไม่จำเป็นต้องรอ

ในการเขียนอธิบายวงจรเป็นภาษา แน่นอนว่าจำเป็นต้องเขียนเป็นบรรทัดเรียงลำดับจากบนลงล่าง โดยที่เราไม่สามารถเขียนได้พร้อม ๆ กัน ดังนั้นสิ่งที่ทำได้คือการกำหนดให้คอมไพเลอร์ หรือตัวจำลองการทำงาน สามารถมองให้เป็นลักษณะของการทำงานแบบขนาน ซึ่งเหมือนกับการทำงานจริง โดยไม่จำเป็นต้องว่าอุปกรณ์ตัวไหนจะอยู่ในบรรทัดก่อน หรือหลัง ดังนั้นถึงแม้ว่าเราจะสลับตำแหน่งกัน แต่การทำงานก็ยังคงเหมือนเดิม



### อุปกรณ์พื้นฐานที่กำหนดโดยผู้ออกแบบ (User-defined Primitives: UDP)

ถึงแม้ว่าเราจะมีเกตพื้นฐานที่ถูกกำหนดไว้แล้ว และสามารถใช้งานได้ทันที (Predefined primitives) แต่อาจไม่เพียงพอกับการออกแบบที่ต้องการอุปกรณ์หลาย ๆ ประเภท นอกเหนือจาก 14 ลอจิกเกตพื้นฐาน ดังนั้นภาษา Verilog จึงอนุญาตให้ผู้ออกแบบสามารถกำหนดอุปกรณ์พื้นฐานของตัวเองได้ (User-defined Primitives: UDP) โดยสามารถกำหนดฟังก์ชันการทำงานได้ตามที่ต้องการ สามารถทำได้ทั้ง Combinational logic หรือ Sequential logic

UDP กับ Module สามารถเรียกใช้ได้เหมือนกัน ทั้งลักษณะของโครงสร้างวิธีการเขียน (ดูรูป) และการเรียกใช้งาน แต่ UDP สามารถมีได้แค่หนึ่งเอาต์พุตเท่านั้น ไม่สามารถมีอินพุต หรือเอาต์พุตเป็นแบบ

เวกเตอร์ได้ และต้องกำหนดการใช้งานเป็นตารางค่าความจริง โดยมีรูปแบบที่แตกต่างกันระหว่างตารางค่าความจริงของอุปกรณ์ประเภท Combinational และ Sequential ดังตัวอย่างของการสร้าง MUX2to1 และ T-Flip-flop ข้างล่าง

## UDP Structure

<b>primitive</b> <b>UDP_name</b> ( <b>port_list</b> )                      ;
<b>port declarations</b>
<b>UDP initialization</b>
<b>truth- or state table</b>
<b>endprimitive</b>

สำหรับโครงสร้างจะประกอบด้วย **primitive UDP\_name** ซึ่งเป็นส่วนที่กำหนดชื่อของ Primitive ตามด้วย (**port list**) ซึ่งจะเหมือนกับการกำหนดการใช้งานในโมดูล แต่พอร์ตแรกจำเป็นต้องเป็นเอาต์พุตพอร์ต และพอร์ตต่อ ๆ ไปเป็นอินพุต โดยที่แต่ละพอร์ตไม่สามารถมีสัญญาณได้มากกว่าหนึ่ง หรือไม่สามารถเป็นเวกเตอร์ได้ ต่อมาคือ **port declaration** ซึ่งกำหนด

ทิศทางของพอร์ตว่าเป็นเข้า หรือออกเท่านั้น ไม่มี bidirectional port เหมือนกับโมดูล และเราสามารถกำหนดค่าเริ่มต้นของสัญญาณได้ที่ส่วนของ **UDP initialization** มีเฉพาะวงจร Sequential logic โดยเริ่มต้นด้วยค่าสวอนคือ initial แล้วใช้เครื่องหมายเท่ากับทำการกำหนดสัญญาณเริ่มต้นที่เอาต์พุต ส่วนของ **truth- or state table** เป็นการกำหนดลักษณะการทำงานของตัว Primitive นั้น ๆ เริ่มต้นด้วยค่าสวอน table และจบด้วย endtable ซึ่งแสดงค่าที่อาจเป็นไปได้ทั้งหมดของอินพุต และค่าที่เอาต์พุตที่จะเกิดขึ้น โดยลำดับของอินพุตจะเรียงตามที่กำหนดใน (port list) ส่วนท้ายสุดเป็นการจบด้วยคำว่า **endprimitive**

## UDP: Mux2to1

<b>primitive</b> Mux2to1 (Out, Sel, In0, In1) ;
<b>output</b> Out;
<b>input</b> Sel, In0, In1;
// no initialization for combinational // primitives
<b>table</b>
//    Sel    In0    In1    :    Out
0     0     ?     :    0 ;
0     1     ?     :    1 ;
1     ?     0     :    0 ;
1     ?     1     :    1 ;
x     ?     ?     :    x ;
<b>endtable</b>
<b>endprimitive</b>

เป็นการสร้าง Multiplexer ที่มี 2 inputs ได้แก่ In0 และ In1 ส่วน output คือ Out โดยมีตัวเลือกคือ Sel ส่วนการกำหนดการทำงานจะอยู่ในตารางค่าความจริงโดยที่สัญลักษณ์ '?' หมายถึงเป็นค่าอะไรก็ได้ อาจเป็น '0', '1', 'x', หรือ 'z' สังเกตว่าจะไม่มีการกำหนดค่าเริ่มต้น (Initial values) เนื่องจากเป็น Combinational logic

**A combinatorial example – 2 to 1 multiplexer.** The value '?' in the truth table denotes 'any value'. Note that the table does not list cases when Sel is '0' or '1' and the active input (In0 or In1, respectively) is 'x'. By definition, non-listed combinations lead to 'x' on the output, which is a correct value in such cases. The case where Sel = 'x' could be omitted for the same reason.

## UDP: TFF

```
primitive TFF (Q, Clk, Clr) ;
output Q; reg Q;
input Clr, Clk;

initial
  Q = 0;

table
  //Clk Clr: Q : Q+
  ? 1 : ? : 0 ;// asynchronous clear
  r 0 : 0 : 1 ;// toggle on rising edge of
  r 0 : 1 : 0 ;// Clk
  f 0 : ? : - ;// ignore falling edge of Clk
  ? f : ? : 0 ;// ignore falling edge of Clr
endtable
endprimitive
```

**A sequential example – T-type flip-flop.** This primitive is initialized to '0' with the **initial** construct. The table contains three groups of columns: inputs, present state and next state of the output. Rising and falling edges are specified with clear shorthand notation. The '-' symbol for the next state of the output denotes "no change".

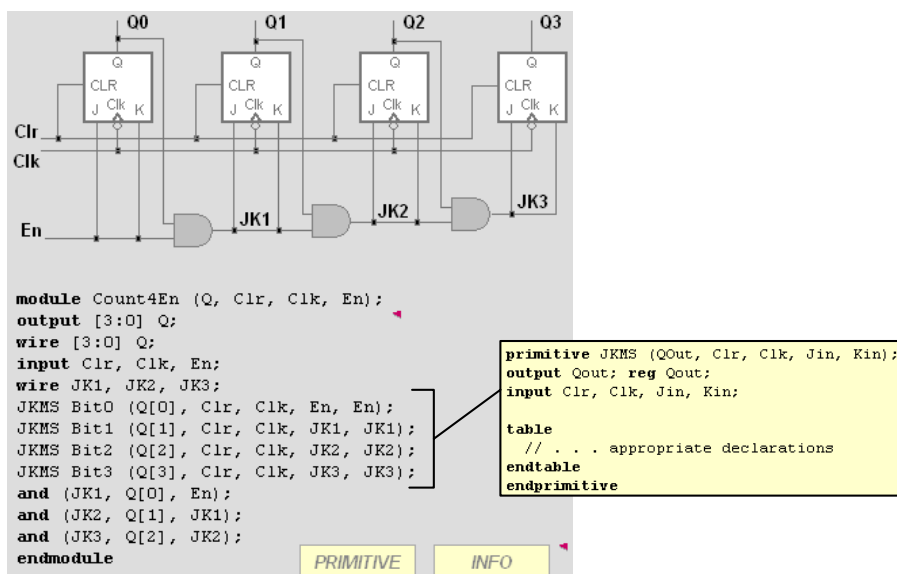
ตัวอย่างการสร้าง Primitive ที่เป็น Sequential logic นี้จะเห็นได้ว่าการกำหนดค่าเริ่มต้นของเอาต์พุต Q = 0; ในตารางที่เป็น state table จะประกอบด้วยสามคอลัมน์ด้วยกันคือ Input, Present state และ Next state ที่เอาต์พุต ค่าของเหตุการณ์ที่เกิดขึ้นขอบขาขึ้น (Rising edge) หรือขอบขาลง (Falling edge) สามารถเขียนได้สั้น ๆ คือ 'r' และ 'f' ตามลำดับ ส่วน '-' หมายถึงการไม่มีการเปลี่ยนแปลงของค่าแต่อย่างใด

## การใช้งาน UDPs (Using UDPs)

การใช้งาน UDPs สามารถใช้ได้เหมือนกับ Predefined primitive หรืออุปกรณ์พื้นฐานที่กำหนดไว้แล้วในไลบรารี สิ่งที่ต้องกระทำเพิ่มขึ้นมาคือ ต้องสร้างตัวอุปกรณ์นั้นก่อน แล้วทำการประกาศ (Declare) ก่อนใช้งาน

เนื่องจาก UDP อยู่ในระดับเดียวกันกับโมดูล เราจึงไม่สามารถสร้าง UDP ได้ในตัวโมดูล ดังนั้นเราจะต้องเขียนแยกไว้ก่อน หรือหลังตัวโมดูลที่ต้องการเรียก UDP มาใช้งานโดยสามารถอยู่ในไฟล์เดียวกันได้ หรือแยกเขียนเป็นไฟล์สำหรับ UDP ต่างหาก แล้วใช้คำสั่ง **'include'** สำหรับการนำไฟล์นั้นมาใช้งาน

UDP เป็นการกำหนดการใช้งานของอุปกรณ์พื้นฐานเท่านั้น ซึ่งยังมีข้อจำกัดต่าง ๆ มากมาย เช่นเราไม่สามารถสร้างตัวอุปกรณ์ที่มีหลายเอาต์พุต หรือที่มีอินพุต เอาต์พุตเป็นแบบเวกเตอร์ได้ ซึ่งเป็นข้อจำกัดที่สำคัญ แต่ข้อจำกัดต่าง ๆ เหล่านี้สามารถหลีกเลี่ยงได้โดยการกำหนดเป็นโมดูลย่อยแทน



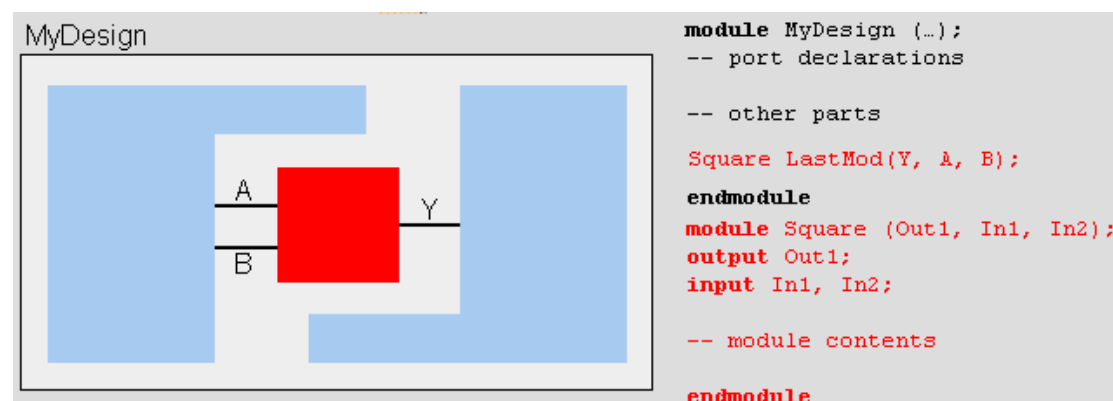
## โมดูลใน Verilog (Module in Verilog)

### การเรียกใช้งานโมดูล (Module Instantiation)

การออกแบบแบบลำดับชั้น (Hierarchical design) ไม่จำเป็นต้องใช้แค่อุปกรณ์พื้นฐานทั่วไป (Primitives) เท่านั้น เรายังสามารถเรียกใช้โมดูลที่เราออกแบบไว้แล้วมาประกอบกันเป็นวงจร หรือระบบขนาดใหญ่ที่ซับซ้อน วิธีการนี้ทำให้เราสามารถตรวจสอบ ค้นหา และทำความเข้าใจได้ง่าย มากกว่าการออกแบบโดยให้ทุกอย่างอยู่ในระดับเดียวกัน หรืออยู่ในโมดูล ๑ เดียว (Flattened design)

การเรียกใช้งานโมดูลใน Verilog มีวิธีการเหมือนกับการเรียกใช้อุปกรณ์พื้นฐานต่าง ๆ โดยที่ไม่จำเป็นต้องใช้การเรียนรู้เพิ่มเติม นอกจากนี้การใช้โมดูลย่อย ในการออกแบบ จะมีความยืดหยุ่นในการออกแบบมากกว่าอุปกรณ์พื้นฐาน เนื่องจากสามารถมีหลายอินพุต หลายเอาต์พุต เป็นต้น อีกอย่างคือ โมดูลสามารถเรียกใช้อุปกรณ์พื้นฐานได้ แต่ในทางกลับกันอุปกรณ์พื้นฐานไม่สามารถเรียกใช้โมดูลได้ ดังนั้นการออกแบบโดยใช้โมดูล ที่ประกอบด้วยโมดูลย่อย จึงสามารถทำการออกแบบเป็นลำดับชั้นได้โดยง่าย และสะดวกกว่า

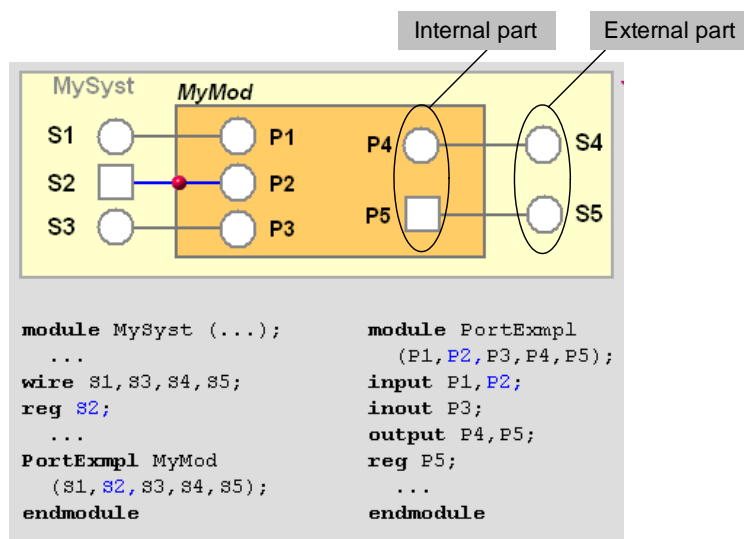
ข้อจำกัดที่สำคัญอย่างหนึ่งในการใช้งานโมดูลคือ เราไม่สามารถออกแบบโมดูลภายในโมดูลใด ๆ ได้ ดังนั้นเราจะต้องเขียนแยกไว้ก่อน หรือหลังตัวโมดูลที่จะเรียกมาใช้งาน โดยสามารถอยู่ในไฟล์เดียวกันได้ หรือแยกเขียนเป็นไฟล์สำหรับโมดูลต่างหาก แล้วใช้คำสั่ง **'include'** สำหรับการนำโมดูลนั้นมาใช้งาน ดังเช่นตัวอย่างในรูป เป็นการเรียกใช้โมดูล Square ในโมดูล MyDesign โดยที่โมดูล Square นั้นถูกเขียนแยกไว้ข้างล่าง



### หลักการต่อเชื่อมพอร์ต (Port Connection Rules)

ส่วนนี้จะอธิบายการเชื่อมต่อพอร์ตระหว่างโมดูล กับสิ่งต่าง ๆ รอบตัว แต่ละพอร์ตประกอบด้วยส่วนที่ต่อเชื่อมภายในโมดูล (Internal part) และส่วนที่อยู่ภายนอก (External part) โดยที่ชนิดของสัญญาณส่วนภายใน และภายนอกโมดูลที่สามารถเป็นได้ทั้ง **net** และ **reg** มีข้อกำหนดดังนี้

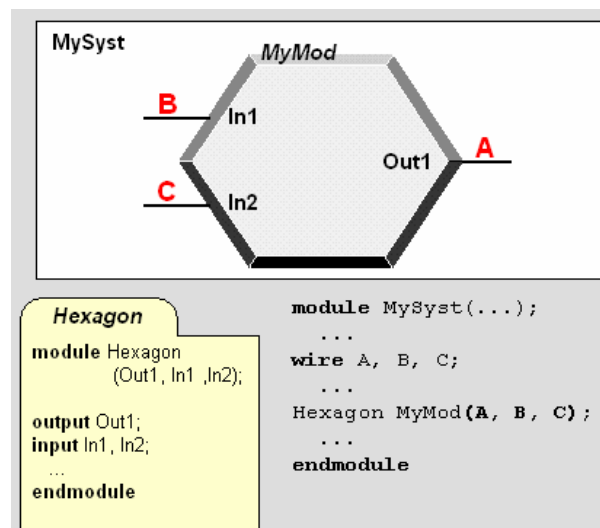
	Internal part	External part
Input	net	net, reg
Output	net, reg	net
Inout	net	net



จากตัวอย่างในรูป Internal part ที่อยู่ในโมดูล MyMod ก็คือ {P1, P2, P3, P4, P5} ส่วน External part คือ {S1, S2, S3, S4, S5} และสังเกตได้ว่า สัญญาณประเภท **reg** แทนด้วยสี่เหลี่ยม ส่วนสัญญาณประเภท **net** จะแทนด้วยวงกลม

### การเชื่อมต่อพอร์ตโดยลำดับ (Connecting Ports by Ordered Port List)

การเชื่อมต่อสัญญาณไปยังพอร์ตในโมดูลสามารถทำได้โดยใช้ภาษาทำการแมป (Map) ตามตำแหน่งของขาที่กำหนดไว้ในโมดูล ดังตัวอย่างในรูปจะเห็นว่าเราเรียกใช้โมดูล Hexagon มาใช้งานในโมดูล MySyst และกำหนดให้มีการเชื่อมต่อสัญญาณ A, B, C เข้าไปที่พอร์ตต่าง ๆ ของโมดูล Hexagon ที่ชื่อว่า MyMod เนื่องจากโมดูล Hexagon กำหนดพอร์ตเรียงลำดับดังนี้คือ (Out1, In1, In2) ดังนั้นตอนที่นำมาใช้งานใน MySyst ในบรรทัด Hexagon MyMod (A, B, C); นั้นหมายถึงว่าตามตำแหน่งแล้ว A จะต่อเชื่อมกับ Out1, B จะต่อเชื่อมกับ In1, และ C จะต่อเชื่อมกับ In2 ตามลำดับของพอร์ตในโมดูลนั้น ๆ



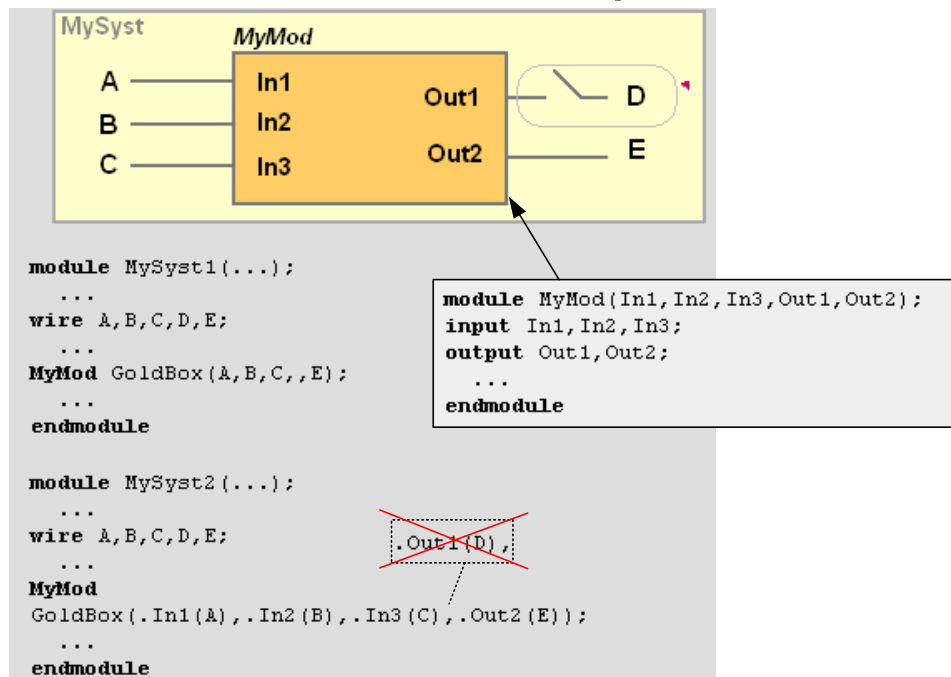
### การเชื่อมต่อพอร์ตโดยชื่อ (Connecting Ports by Name)

สำหรับการเชื่อมต่อด้วยวิธีการนี้ เราสามารถสลับที่ของพอร์ตในโมดูลที่ถูกเรียกมาใช้งานได้ แต่เราจะต้องกำกับชื่อของพอร์ตไปด้วย ตัวอย่างเช่น Hexagon MyMod (.In1(B), .In2(C), .Out1(A)); โดยมีการเชื่อมต่อของสัญญาณเหมือนกันกับตัวอย่างในรูปข้างบน เพียงแต่ว่าเราสามารถสลับตำแหน่งของพอร์ตโดยมีชื่อพอร์ตกำกับไว้ได้

### การกำหนดพอร์ตที่ไม่มีการเชื่อมต่อ (Unconnected Ports)

นอกจากการกำหนดพอร์ตโดยอ้างอิงถึง Port list และอ้างอิงถึงชื่อแล้ว ภาษา Verilog มีการกำหนดพอร์ตที่ไม่ได้ใช้งาน โดยสามารถทำได้ตามลักษณะการกำหนดการเชื่อมต่อพอร์ตดังนี้

- เมื่อมีการเชื่อมต่อโดยใช้ชื่อพอร์ต เราสามารถตัดพอร์ตที่ไม่ได้ใช้งานออกได้ทันที ดังนั้นก็จะเหลือเพียงแค่พอร์ตที่ใช้งานแสดงไว้เท่านั้น ดังตัวอย่างในรูป



- เมื่อมีการเชื่อมต่อโดยอ้างถึงลำดับ Port list เนื่องจากเราต้องรักษาลำดับตำแหน่งลำดับของพอร์ตในโมดูล ดังนั้นเมื่อไม่มีการใช้งานของพอร์ตใด พอร์ตหนึ่ง เราจะต้องเว้นที่ว่างไว้โดยใช้ Blank space แทนชื่อของสัญญาณ เช่น GoldBox (A, B, C, , E);

# CHAPTER 5

## Specification with Signal Transformations

การออกแบบระบบด้วย Structural style โดยการเขียนอธิบายโครงสร้างของวงจรใดวงจรหนึ่งด้วยภาษาชั้นสูงสามารถทำได้ดังแสดงในบทที่ผ่านมา ในบทนี้จะเน้นลักษณะการออกแบบโดยการอธิบายความสัมพันธ์ระหว่างเอาต์พุต และอินพุต ซึ่งการออกแบบลักษณะนี้เราเรียกว่า Dataflow style นั่นเอง โดยจะแนะนำให้รู้จัก Expression ที่ประกอบด้วย Operands (หรือตัวแปรอินพุตต่าง ๆ) ที่ถูกกระทำด้วย Operators (หรือตัวกระทำ) อยู่ภายใน

สำหรับตัว Operators ในภาษา Verilog นั้นมีให้ใช้งานหลากหลาย ตามชนิดของ Operands และการใช้งานที่แตกต่างกัน ในการกำหนดเป็น Dataflow expression นั้น ผลจากการกระทำใน Expression จะต้องถูกกำหนดหรือส่งผ่าน (Assign) มาให้กับสัญญาณที่ต้องการ (Target signal) ลักษณะการส่งผ่านค่าสัญญาณในลักษณะนี้ เราเรียกว่า Continuous assignments ซึ่งจะกล่าวในบทนี้เช่นกัน

### Expressions

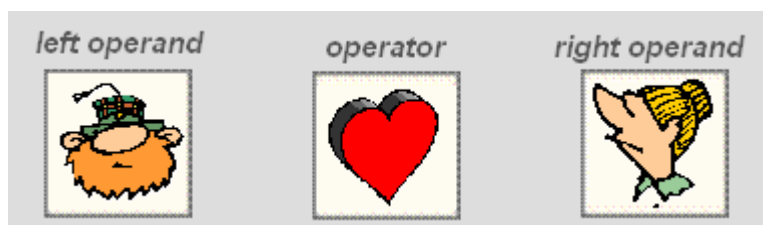
#### รู้จักกับ Expressions (Introduction to Expressions)

โดยปกติแล้วในระบบทั่วไป สัญญาณอินพุตจะถูกแปลง (Transform) ด้วยวิธีการใด ๆ วิธีการหนึ่ง เพื่อสร้างสัญญาณเอาต์พุตที่ต้องการออกมา กระบวนการแปลงเช่นนี้ สามารถเขียนได้เป็น

*Outputs <- transformations (Inputs)*

นิยามตามพจนานุกรมของ Expressions นั่นคือ “การสร้างผลลัพธ์จากการที่ **Operands** กระทำกันด้วย **Operators** ในการที่จะกำหนดฟังก์ชันการทำงานใด ๆ ขึ้นมา” ในภาษา Verilog นั้น Operator หนึ่งตัวสามารถมี Operands ได้ตั้งแต่ 1, 2, หรือ 3 ตัว ตามแต่ความต้องการใช้งาน โดยอาจเป็น Operands ต่างชนิดกันก็ได้ ซึ่งเป็นข้อยืดหยุ่นที่สำคัญในภาษา Verilog ที่ต่างกับภาษา VHDL

ตัวอย่างของ Expression แสดงได้ดังรูป



ในหัวข้อต่อไปจะแนะนำให้รู้จักกับแต่ละชนิดของ Operands



## Operands

แต่ละ Expression จำเป็นต้องมี Operands ที่ถูกกระทำด้วยตัว Operators ในการที่จะกำหนดฟังก์ชันการทำงานเพื่อที่จะสร้างผลลัพธ์ที่ต้องการออกมา และเนื่องจากตัว Operands มีหลายชนิด ดังนั้นการเลือกใช้ Operators จะต้องสอดคล้องกันกับชนิดของ Operands ด้วย ซึ่งสามารถแบ่งได้เป็นดังนี้

- ตัวอย่างที่เห็นได้ง่าย และชัดเจนที่สุดสำหรับชนิดของ Operands คือการอ้างถึงถึงสัญญาณ Nets และ Registers ด้วยชื่อต่าง ๆ หรือที่เรียกว่า *Instance names* นั้นเอง
- ค่าคงที่ หรือ Constant values ก็เป็นอีกชนิดหนึ่งของ Operands ที่สำคัญ ซึ่งในภาษา Verilog นั้นมีค่าคงที่หลาย ๆ ประเภทเช่น Integer, Real เป็นต้น
- ค่าที่ประกอบด้วยหลาย ๆ บิต หรือที่เรียกว่า Vector ซึ่งสามารถเป็นได้ทั้ง Nets หรือ Registers และเราสามารถกำหนดการเรียกใช้งานในแต่ละบิต (Bit-select) หรือบางส่วน (Part-select) ของค่าเวกเตอร์นั้น ๆ ได้
- ชนิดสุดท้ายของ Operands อาจจะเป็นการเรียกใช้ฟังก์ชัน (A call to a function) ทั้งฟังก์ชันของระบบ หรือที่ผู้ใช้งานออกแบบขึ้นเอง โดยที่หลังจากเรียกใช้จะมีค่าที่ส่งกลับมา ที่มีความสอดคล้องกับชนิดของตัวกระทำ (Operators)

```
wire in1;
real Radius;
reg [7:0] DataBus;

in1
    Radius
    DataBus[7:4]
    DataBus[0]
    CircleArea(Radius)
    1.3e7      127
              4'b1001
```

ตัวอย่าง Operands สามารถแสดงได้ในรูป จะเห็นว่า in1, Radius และ DataBus เป็น Simple reference ที่เป็นการอ้างถึงถึงสัญญาณ Nets และ Registers ที่เป็นแบบเวกเตอร์ ส่วนการใช้งานบางส่วน (Part-select) สามารถใช้เป็น DataBus[7:4] หรือการใช้งานเฉพาะบิตคือ DataBus[0] เป็นต้น สำหรับตัวอย่างของ Operands ที่เป็นค่าคงที่ (Constant values) เช่น 1.3e7 (Scientific) , 127 (Decimal), 4'b1001 (Binary) สำหรับ

ตัวอย่างที่เป็นชนิดของการเรียกใช้ฟังก์ชันคือ CircleArea(Radius) เป็นต้น

## ค่าคงที่จำนวนเต็ม (Integer Constants)

ค่าคงที่จำนวนเต็ม คือค่าจำนวนเต็มที่เขียนในเลขฐานสิบที่ใช้ในชีวิตประจำวัน สามารถเป็นไปได้ทั้งค่าที่เป็นบวก หรือลบ เช่น 1, 3, 457, -34, -872 เป็นต้น

ในภาษา Verilog ค่าเลขฐานของจำนวนเต็มโดยปริยาย (Default) คือฐานสิบ (Decimal) ถ้าหากไม่กำหนดเลขฐานกำกับ แต่เราสามารถกำหนดใหม่ให้เป็นฐานสอง ฐานแปด หรือฐานสิบหกได้ตามต้องการ โดยต้องมีสัญลักษณ์ ' ตามด้วยตัวอักษรที่บอกค่าของเลขฐาน เช่น 'h' หรือ 'H' สำหรับฐานสิบหก, 'o' หรือ 'O' สำหรับเลขฐานแปด, 'b' หรือ 'B' สำหรับเลขฐานสอง, 'd' หรือ 'D' สำหรับเลขฐานสิบ

การกำหนดค่าในลักษณะนี้จะถูกกำหนดให้เป็นเลขจำนวนเต็มบวก เช่น 8'b10100001, 12'B0001\_0010\_1010, 16'H12AB เป็นต้น เราจะสังเกตเห็นว่าตัวเลขด้านหน้าของเครื่องหมาย ' จะเป็นค่าที่แสดงจำนวนบิตของค่าจำนวนเต็ม นั้น ๆ ซึ่งบางครั้งเราอาจจะไว้ถ้าเราใส่ค่าครบทุกบิต เช่น 'b0001\_0010\_1010 ซึ่งเป็นการแสดงค่าในเลขฐานสอง และมีอยู่ 12 บิต ส่วนเครื่องหมาย \_ (Underscore) ใช้สำหรับช่วยในการกำหนดกลุ่มให้ง่ายสำหรับการอ่านค่า และไม่มีผลต่อค่าของจำนวนเต็มที่กำหนดแต่อย่างใด

- **Unsigned integer:** เป็นการกำหนดค่าเลขจำนวนเต็มโดยไม่บอกจำนวนบิต ค่าจำนวนบิตจะถือเป็นค่า Default นั่นคือ 32 บิต แต่การกำหนดเลขฐานสำหรับค่าที่ไม่ใช่ฐานสิบต้องมีกำกับตามปกติ ตัวอย่างเช่น

```
12      // no base specified -> decimal number
'h12    // hexadecimal number, equal to decimal 18
'ha0    // another hexadecimal number
'b1001  // binary number
a0      // illegal - 'a' is not a decimal digit
```

- **Sized integer:** จำเป็นต้องมีจำนวนบิตกำกับ โดยกำหนดให้เป็นตัวเลขจำนวนเต็มบวกฐานสิบ ซึ่งค่าที่แท้จริงของค่าคงที่นั้นจะต้องมีจำนวนบิตที่น้อยกว่า หรือเท่ากับจำนวนบิตที่กำหนด ในกรณีที่น้อยกว่า ตัวคอมไพเลอร์จะเติมค่า '0' ให้ในบิตสูงที่ไม่ถูกกำหนด เช่น

```
4'd4      // decimal 4 to be written on 4 bits instead of default 32
8'b10011001 // 8-bit binary value
8'b1      // will be represented as 00000001 (padded with zeros)
8'h1      // will be represented as 00000001
```

- **Negative value:** ค่าที่เป็นลบของจำนวนเต็ม สามารถกำหนดได้โดยการใส่เครื่องหมายลบไว้ข้างหน้าสุด โดยตัวคอมไพเลอร์จะมองเป็น 2's complement form ดังตัวอย่าง

```
-10      // internally represented as two's complement to
          // decimal 10 and written on 32 bits
-8'd10   // internally represented as two's complement to
          // decimal 10 and written on 8 bits; equivalent to - (8'd10)
```

- **Use of '?':** เครื่องหมายคำถาม ? ใช้แทนการกำหนดค่า 'z' หรือ High impedance ที่เป็นการกำหนดค่า "don't care condition" ของค่าคงที่นั้น ๆ โดยจำนวนบิตของ 'z' จะเท่ากับ 4, 3, 1 เมื่อกำหนดในเลขฐานสิบหก ฐานแปด ฐานสอง ตามลำดับ

```
8'h1?    // equivalent of 0001zzzz
2'b1?    // equivalent of 1z
```

- **Use of 'x' and 'z':** การแทนบิตค่าคงที่ด้วย 'x' หรือ 'z' (สามารถใช้ตัวเล็ก หรือใหญ่ก็ได้) เป็นการกำหนดค่า "Unknown" หรือ ค่า "High impedance" ตามลำดับ สามารถใช้ในเลขฐานสอง ฐานแปด หรือฐานสิบหก (แต่ไม่สามารถใช้กับฐานสิบได้) โดยจำนวนบิตของ 'z' จะเท่ากับ 4, 3, 1 เมื่อกำหนดในเลขฐานสิบหก ฐานแปด ฐานสอง ตามลำดับ

```
4'b01xx // last two bits of this four bit number unknown
8'h1x    // last four bits unknown (equivalent of binary 0001xxxx)
'hx      // 32-bit unknown number
8'hxx    // 8-bit unknown number (could as well be written as 8'hx
          // - see left padding)
```

- **Use of '\_':** เครื่องหมาย Underscore เป็นการกำหนดกลุ่ม หรือวรรคของค่าคงที่ที่กำหนด ทำให้สามารถอ่านได้โดยง่าย สามารถแทรกได้ตามจุดต่าง ๆ ภายในค่า แต่ไม่สามารถเขียนไว้หน้าสุดของค่านั้น ๆ ได้ เช่น

```
16'b1001_1100_1110_0001
197_832_001
```

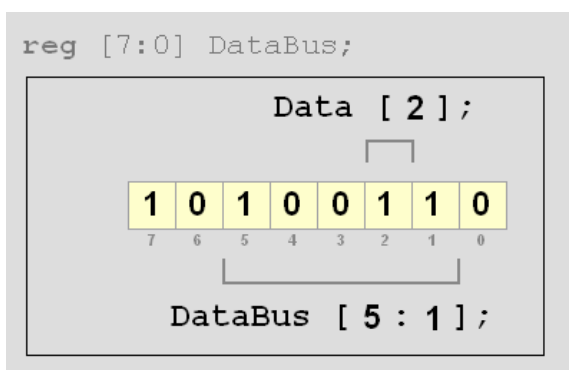
- **Left padding:** ในกรณีที่ขนาดของค่าจำนวนเต็มน้อยกว่าค่าของจำนวนบิต การทำ Left padding หรือการเติมค่าเข้าทางด้านซ้ายในตำแหน่งที่ไม่มีการกำหนด จะถูกนำมาใช้งาน กรณี

ปกติจะเป็นค่า '0' สำหรับการทำ Left padding แต่ถ้าค่าด้านซ้ายสุดของค่าที่กำหนดเป็น 'x' หรือ 'z' จะต้องใส่ค่า 'x' หรือ 'z' เข้าไปให้เต็มแทนที่จะเป็น '0' ตามตัวอย่าง

```
8'b0      // equivalent to 00000000
8'b1      // equivalent to 00000001
8'bx      // equivalent to xxxxxxxx
16'hx10   // equivalent to xxxxxxxx00010000
```

### Bit-select และ Part-select สำหรับตัวแปรเวกเตอร์ (The Bit-select and Part-select Operands)

เมื่อเราต้องการส่วนหนึ่งของตัวแปรเวกเตอร์ อาจจะเป็นแค่บิตเดียว หรือหลาย ๆ บิตต่อกัน สามารถทำได้โดยใช้ Select operand โดยมีไวยากรณ์ที่ไม่ซับซ้อน แค่กำหนดบิต หรือช่วงบิตที่ต้องการในเครื่องหมาย [ ] โดยถ้าเป็นการกำหนดช่วงในลักษณะของ Part select การกำหนดช่วงบิตจะต้องมีเครื่องหมาย : (Colon) เป็นตัวคั่นระหว่าง MSB กับ LSB นั่นคือ [MSB:LSB] ดังรูป



ถ้าเรากำหนดช่วงของบิต (Indexes) เกินกว่าที่กำหนดไว้ในตัวแปรเวกเตอร์ ค่าของบิตในตำแหน่งที่เลยออกมาจะถูกกำหนดเป็น 'x' โดยอัตโนมัติ

## ตัวกระทำ (Operators)

### ตัวกระทำทางคณิตศาสตร์ (Arithmetic Operators)

ในภาษา Verilog มีตัวกระทำมากถึง 32 ตัว สามารถแบ่งได้เป็นสามกลุ่มตามจำนวนของ Operands ที่ใช้งานด้วย คือ Unary (with one operand), Binary (with two operands), Ternary (with three operands) หรือถ้าแยกตามประเภทของการทำงานสามารถกำหนด ได้เป็นชนิดหลัก ๆ ได้ดังนี้คือ

- ตัวกระทำเชิงคณิตศาสตร์ (Arithmetic operator)
- ตัวกระทำเชิงตรรก (Logical operator)
- ตัวกระทำเชิงความสัมพันธ์ (Relational operator)

ซึ่งในหัวข้อนี้จะพูดถึงตัวกระทำเชิงคณิตศาสตร์ก่อน ซึ่งคงคุ้นเคยกันดีสำหรับ Operators ประเภทนี้ ซึ่งได้แก่ + แทนการบวก, - แทนการลบ, \* แทนการคูณ, / แทนการหาร โดยที่สองตัวแรกคือ + และ - สามารถเป็น Unary operator ที่มี Operand เพียงแค่ตัวเดียวสำหรับการแทนเป็นจำนวนบวก หรือจำนวนลบ

นอกจากตัวกระทำพื้นฐานทั้งสี่ตัวนี้แล้ว % เป็นอีกตัวหนึ่งสำหรับการ Modulo ซึ่งจะได้ผลลัพธ์เป็นค่าที่เหลือ (Remainder) จากการหารกันระหว่างตัวตั้ง และตัวหาร

สิ่งสำคัญอย่างหนึ่งสำหรับการกระทำโดยใช้ Arithmetic operators คือค่าที่ได้จาก Operands ประเภท Register และ Integer จะแตกต่างกันเนื่องจาก ค่าของสัญญาณ หรือตัวแปรประเภท Register จะคิดเป็นค่าจำนวนเต็มบวก (Unsigned) แต่สำหรับ Integer จะคิดเป็นค่าจำนวนเต็มที่มีค่าได้ทั้งบวก และลบ (Signed) ทำให้บางครั้งค่าผลลัพธ์ที่ได้จะแตกต่างกัน ถึงแม้ว่าจะมีการกำหนดค่าที่เหมือนกัน ดังตัวอย่าง

```
integer intA;

intA = -4'd12
```

left operand	operator	right operand	result
intA	/	3	= -4

*intA is an integer and is treated as a signed number. This results in an expected -4 as a result of the operation. However, should the result be assigned to a register variable, it would be stored as 65532 – the two's complement representation of -4.*

IntA ถูกกำหนดเป็นตัวแปรประเภท integer ซึ่งค่า -12 จะถูกแปลงเป็นเลข 2's ก่อนทำการหาร โดยได้ผลลัพธ์เท่ากับ 65532 ซึ่งเป็นค่า 2's ของ -4

```
reg[15:0] intA;

intA = -4'd12
```

left operand	operator	right operand	result
intA	/	3	= 21841

*Since RegA is a register, it stores negative values in two's complements. In this case -12 will be represented internally as 65524. Such a number when divided by three results in 21841, which is the result of the operation.*

IntA ถูกกำหนดเป็น Register ซึ่งถ้ากำหนดเป็น -12 ซึ่งแทนได้ด้วยค่าภายในตัวคอมพิวเตอร์ เป็นจำนวนบวกที่เท่ากับ 65524 ดังนั้นจะได้ผลลัพธ์เท่ากับ 21841 ซึ่งไม่ถูกต้อง

### ตัวกระทำเชิงความสัมพันธ์ (Relational Operators)

เป็นตัวกระทำที่ใช้ในการเปรียบเทียบว่า มากกว่า น้อยกว่า หรือเท่ากับ ระหว่างตัว Operands สองตัวใด ๆ โดยผลลัพธ์ที่ได้ออกมาเป็น 1 ถ้าเป็นไปตามเงื่อนไขของความสัมพันธ์ (ว่ามีตัวใดมีค่ามากกว่า หรือน้อยกว่า หรือเท่ากัน) จะมีค่าเป็น '1' เมื่อความสัมพันธ์นั้นเป็นจริง และจะมีค่าเป็น '0' เมื่อเป็นเท็จ

ในกรณีที่บิตใด บิตหนึ่งของ Operands มีค่าเป็น 'x' หรือ 'z' ค่าผลลัพธ์ที่ได้จะเท่ากับ 'x' (Unknown) และถ้าจำนวนบิตของ Operands ไม่เท่ากัน ตัวที่น้อยกว่าจะถูกเติมด้วย '0' ในด้านซ้ายมือ (Left padding)

ลำดับความสำคัญของการทำงาน หรือที่เรียกว่า Precedence ของ Relational operators จะน้อยกว่า Arithmetic operators

left operand	right operand	<	>	<=	>=
101	0110	1	0	1	0

### ตัวกระทำที่เป็น Equality Operators

นอกจากลักษณะของการกระทำเปรียบเทียบว่าเท่ากับ '==' หรือไม่เท่ากับ '!=' แล้ว ในภาษา Verilog มีลักษณะพิเศษเพิ่มขึ้นสำหรับการเปรียบเทียบโดยมีเงื่อนไขสำหรับสัญญาณ 'x' หรือ 'z' เรียกว่า Case equality (inequality) โดยมีสัญลักษณ์เป็น '=== ' หรือ '!== ' โดยจะเป็นการเปรียบเทียบค่าของทั้งสอง Operands โดยไม่คำนึงถึงว่าเป็น 'x' หรือ 'z' ดังนั้นผลลัพธ์จะออกมาเป็น '0' หรือ '1' เท่านั้นแล้วแต่เงื่อนไขที่ใช้ในการเปรียบเทียบ เช่น ถ้าเท่ากันทุกบิตไม่ว่าแต่ละบิตจะเป็นค่า '0' '1' 'x' หรือ 'z' ผลลัพธ์ของการเปรียบเทียบโดยใช้เงื่อนไข '===' จะเป็นจริง ดังตัวอย่าง

left operand	right operand	===	!==	==	!=
0xx0	0xx0	1	0	x	x

### ตัวกระทำทางตรรก (Logical Operators)

ในภาษา Verilog ได้แบ่งตัวกระทำทางตรรก หรือทางลอจิกเป็น 3 ประเภทใหญ่ ๆ ด้วยกันคือ

- Logical operators: จะทำการกระทำทางลอจิกกับ Operands สองตัว ที่ความยาวบิตใด ๆ ผลลัพธ์ที่ได้จะเป็น 1 บิต ที่มีค่าเป็นจริง ('1') หรือ เท็จ ('0') เท่านั้น

#### Logical and (&&)

for A = 0 and B = 2    A && B evaluates to 0  
for A = 2'b00 and B = 2'b0x    A && B evaluates to x

- Bit-wise operators: จะนำเอา Operands สองตัวที่ความยาวบิตใด ๆ มากระทำทางลอจิกบิตต่อบิต ตามตำแหน่ง ตัวอย่างเช่น บิตที่ k ของตัว Operand ซ้าย จะกระทำทางลอจิกกับบิตที่ k ของตัวขวา ดังนั้นผลลัพธ์จะได้ความยาวบิตเท่ากับ ความยาวบิตของตัว Operand ที่ยาวที่สุด สำหรับตัว Operand ที่ยาวน้อยกว่าจะมีการทำ Left padding ในตำแหน่งบิตที่ขาดไป ก่อนจะนำมากระทำทางลอจิกในแต่ละบิต

#### Bit-wise and (&)

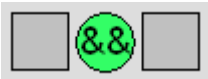




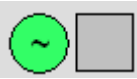





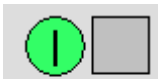

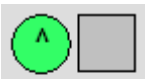


for A = 4'b1010 and B = 4'b0011    A & B evaluates to 0010  
for A = 4'b1010 and B = 4'b001x    A & B evaluates to 001x

- Reduction operators: จะใช้กับ Operand เพียงตัวเดียวที่เป็นเวกเตอร์ โดยผลลัพธ์จะเท่ากับการนำเอาแต่ละบิตใน Operand ตัวนั้นมากระทำทางลอจิกกันในตัวมันเองจนครบ ดังนั้นผลลัพธ์ที่ได้จะมีเพียงแคบิตเดียว

### Reduction and (&)

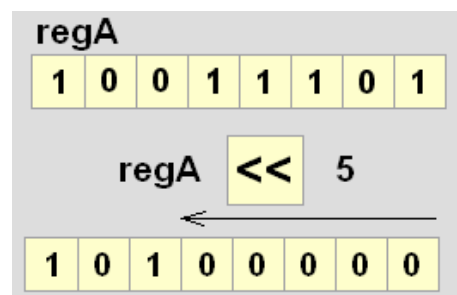
for A = 4'b1010    & A evaluates to 0  
 for B = 4'b0001    & B evaluates to 0  
 for C = 4'b111x    & C evaluates to x

ตารางสรุปการใช้งานตัวกระทำทางตรรกในประเภทต่าง ๆ

	AND	OR	NOT	XOR/XNOR
<b>Logical operators</b>				—
<b>Bit-wise operators</b>				  
<b>Reduction operators</b>	 	 	—	  

### ตัวกระทำที่เป็น Shift Operators

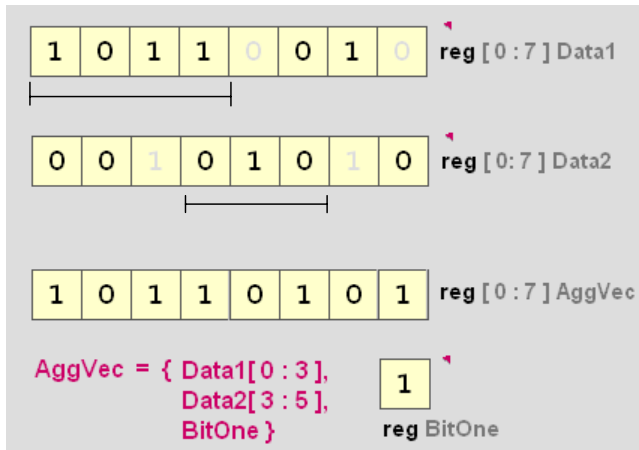
การใช้งานจะประกอบด้วย Operands สองตัว ตัวแรกจะเป็นสัญญาณเวกเตอร์ที่ต้องการให้เกิดการชิฟ และตัวที่สองจะเป็นจำนวนของการชิฟ ว่าต้องการชิฟไปกี่บิต เราสามารถทำการชิฟได้ทั้งซ้าย และขวาโดยใช้เครื่องหมาย '>>' สำหรับการชิฟไปด้านขวา และ '<<' สำหรับการชิฟไปด้านซ้าย โดยบิตที่ถูกชิฟจะถูกแทนด้วยลอจิก '0'



### การต่อเชื่อมสัญญาณเวกเตอร์ (Vector Concatenations)

การต่อเชื่อมค่าต่าง ๆ ของสัญญาณ หรือที่เรียกว่า Concatenation สามารถทำได้โดยใช้เครื่องหมาย {, } สำหรับการต่อสัญญาณต่าง ๆ รวมกันเป็นเวกเตอร์ตัวใหม่ โดยจะใช้ , (Comma) คั่นระหว่างค่าของแต่ละตัว ซึ่งอาจเป็น Scalars, nets, registers, vector nets, vector registers, bit-select, part-select หรือแม้กระทั่งค่าคงที่ (ที่มีจำนวนบิตที่แน่นอน)

นอกจากนี้เราสามารถต่อค่าสัญญาณชุดที่ซ้ำ ๆ กันได้โดยเพียงแต่กำหนดจำนวนการซ้ำไว้ก่อนหน้าสัญญาณชุดนั้น เช่น {DATA [1:0], DATA [1:0], DATA [1:0], DATA [1:0]} สามารถลดรูปได้เป็น {4{DATA[1:0]}}



ตัวอย่างนี้แสดงการสร้าง AggVec ขนาด 8 บิต จากเวกเตอร์ Data1, Data2, และสเกลาร์ BitOne โดยเลือกเอาบางส่วนของ Data1 และ Data2 มาเรียงกัน (Part-select)

### การส่งผ่านค่าแบบต่อเนื่อง (The Continuous Assignment)

เมื่อไหร่ก็ตามที่มีการกระทำของ Expressions ที่ประกอบด้วย Operands และ Operators แล้วได้ผลลัพธ์ออกมา เราจำเป็นต้องทำการส่งผ่านค่า (Assign) ผลลัพธ์นั้น ๆ ให้กับตัวแปรสัญญาณใด ๆ เพื่อให้สามารถนำสัญญาณไปใช้งานต่อได้ในระบบ

การส่งผ่านค่าสามารถทำได้สองวิธีคือ Continuous assignment หรือ Procedural assignment ซึ่งในบทนี้เราจะพูดถึงวิธีแรกเท่านั้น ส่วนอีกวิธีจะมีอยู่ในบทต่อไป

การใช้งานในลักษณะของ Continuous assignment เป็นการส่งผ่านค่าจากผลลัพธ์ที่ได้จาก Expression ทางด้านขวามือ ของสัญลักษณ์ Assignment symbol มายังตัวแปรสัญญาณที่อยู่ทางด้านซ้ายมือ ซึ่งเป็นลักษณะของ Dataflow model โดย Continuous assignment จะมีส่วนประกอบต่าง ๆ ดังนี้คือ

- คำสั่ง **assign**
- ค่าดีเลย์ (มีหรือไม่มีก็ได้)
- ตัวแปรสัญญาณทางซ้ายมือ ซึ่งจะเป็นปลายทางในการรับผลลัพธ์จาก Expression ทางด้านขวามือที่จะเป็นต้นทาง โดยตัวรับสัญญาณนี้จะต้องกำหนดเป็น Nets (scalar or vector) เท่านั้น จะกำหนดให้เป็นประเภท Registers ไม่ได้
- สัญลักษณ์ของการส่งผ่านค่า เราใช้เครื่องหมายเท่ากับ '=' ธรรมดา

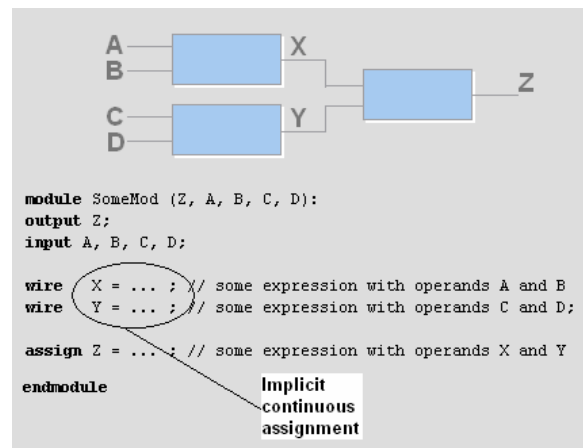
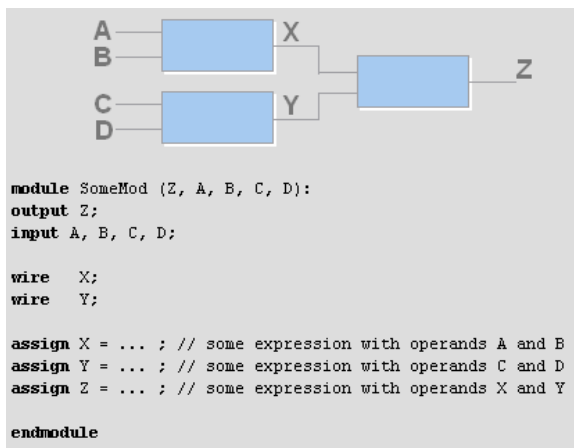
ตัวอย่างเช่น

```
assign Out1 = A & B & C;  
assign Out2 = A + B;  
assign Out3 = {A, B, C};
```

สำหรับการส่งผ่านค่าแบบ Continuous assignment จะมีการกระทำพร้อม ๆ กันทันที ถ้ามีการเปลี่ยนแปลงค่าของ Operands ใด ๆ ที่อยู่ใน Expression ทางด้านขวามือเกิดขึ้น โดยจะมีการหาผลลัพธ์ที่เกิดขึ้นใหม่ จากการเปลี่ยนแปลงค่าของตัว Operands ก่อนที่จะทำการส่งผ่านค่ามายังตัวแปรทางด้านซ้ายมือ

### การส่งผ่านค่าแบบแฝง (The Implicit Continuous Assignment)

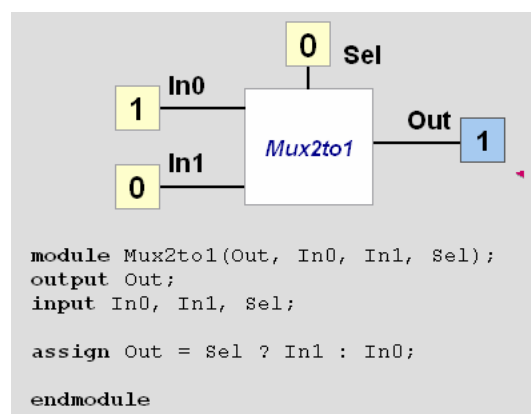
เราสามารถส่งผ่านค่าแบบ Continuous assignment ให้กับ Net ตอนที่เรากำหนดใช้งาน (Net declaration) ได้ทันที โดยไม่ต้องมีคำสั่ง **assign** วิธีการนี้เรียกว่า Implicit assignment หรือ Net declaration assignment เช่น `wire x = .....;`



รูปทางด้านซ้ายมือ เป็นการกำหนดโดยวิธีการปกติที่เห็นการส่งผ่านค่าให้กับ X, Y ได้อย่างชัดเจน โดยใช้คำสั่ง **assign** เรียกว่า Explicit assignment ส่วนรูปทางด้านขวามือ จะเป็นการส่งผ่านค่าให้กับ X, Y ทันที หลังจากการกำหนดเป็นตัวแปรสัญญาณแบบ **wire** โดยไม่ต้องใช้คำสั่ง **assign** เรียกว่า Implicit assignment

### การส่งผ่านค่าแบบมีเงื่อนไข (The Conditional Assignment)

การกำหนดการทำงานแบบมีเงื่อนไข มีประโยชน์มากในการอธิบายการทำงานของทั้งซอฟต์แวร์ และ ฮาร์ดแวร์ แต่อย่างไรก็ตามเราไม่สามารถใช้คำสั่ง *if-then-else* ในลักษณะการกำหนดค่าที่เป็น Continuous assignment ได้ ดังนั้นจึงมีตัวกระทำทางเงื่อนไข (Conditional operator) มาทดแทนการใช้งานที่ไม่ซับซ้อน



ตัวอย่างเช่น **assign** Out = Sel ? In1 : In0 ;  
 In0 : In1; เป็นการส่งผ่านค่าให้กับ Out โดยมีเงื่อนไขคือ ถ้า Sel เป็นจริง ค่า In0 จะถูกกำหนดมาให้กับ Out แต่ถ้า Sel เป็นเท็จ ค่า In1 จะถูกกำหนดมาให้แทน ซึ่งก็คือการทำงานของ Multiplexer ที่มีสองอินพุต และหนึ่งเอาต์พุตนั่นเอง



# CHAPTER 6

## The Behavioral Approach

การอธิบายการทำงานของระบบด้วยสมการบูลีน หรือ Dataflow style มีข้อจำกัดต่าง ๆ มากมาย เนื่องจากการอธิบายการทำงานในระดับต่ำ สำหรับภาษา Verilog สามารถใช้ในการอธิบายการทำงานได้ในระดับที่สูงขึ้นในการกำหนดการทำงานของระบบ โดยมีการอธิบายเป็นพฤติกรรมของวงจร (Behavioral description) ทำให้ง่ายขึ้น และสะดวกรวดเร็วในการออกแบบระบบ โดยที่หน้าที่ของการสร้างวงจรในระดับต่ำ ๆ เช่น RTL หรือ Gate level ก็จะเป็นหน้าที่ของตัวสังเคราะห์วงจรแทน

จากที่ผ่านมา ในการเขียนในลักษณะของ Dataflow สามารถทำได้โดยใช้คำสั่ง **assign** ซึ่งเป็นการส่งผ่านค่าที่ได้จาก Expression ทางด้านขวามือของเครื่องหมาย = มายังสัญญาณประเภท Nets เท่านั้น ในทางกลับกัน สำหรับการเขียนในลักษณะของ Behavioral style จะมีการใช้งานกับสัญญาณประเภท Registers ซึ่งมีหลาย ๆ ประเภทที่จะกล่าวถึงต่อไปในบทนี้

การเขียนอธิบายการทำงานในลักษณะของ Behavioral style จะมีวิธีการเขียนที่คล้ายกับภาษาซี เป็นอย่างมาก เช่นมีการใช้คำสั่ง *if-then-else*, *case* statement เป็นต้น ซึ่งในบทนี้เราจะได้เรียนรู้ ลักษณะของไวยากรณ์ในการใช้งานสำหรับการอธิบายการทำงานของฮาร์ดแวร์ด้วยภาษา Verilog

### ตัวแปร และพารามิเตอร์ (Variables and Parameters)

#### ทำไม Nets จึงไม่เพียงพอ (Aren't Nets Enough?)

จากที่ทราบมาแล้วว่า กรณีที่มีสัญญาณที่มีค่า ๆ หนึ่งที่ตัวส่ง (หรือที่เราเรียกว่า Source หรือ Driver) แล้วต้องการส่งผ่านค่านี้ไปยังอีกที่ ๆ หนึ่ง เราสามารถกำหนดให้เป็นการทำงานในลักษณะของ Dataflow ได้ โดยใช้ Nets ซึ่งถ้าสัญญาณที่ Driver ถูกตัดขาดแล้ว สถานะที่ Nets ที่ใช้ในการรับค่าจะเป็น High impedance แต่สำหรับ Registers แล้วถึงแม้ว่าสัญญาณที่ Driver จะถูกตัดออกไป แต่มันก็ยังสามารถคงสถานะของค่าสุดท้ายที่ได้รับอยู่ได้

คุณสมบัติในการคงค่าได้ของ Registers จำเป็นต้องใช้ในการอธิบายวงจรในแบบของ Behavioral style ที่ส่วนใหญ่เป็นการเขียนอัลกอริทึมในการอธิบายพฤติกรรมการทำงาน และจำเป็นต้องมีการใช้งานของตัวแปร (Variables) ต่าง ๆ อยู่ภายใน คุณสมบัติของตัวแปรโดยทั่วไปคือ สามารถคงค่าได้ตราบเท่าที่ไม่มีการเปลี่ยนแปลงค่าใหม่เกิดขึ้น ดังนั้น Registers ในภาษา Verilog จึงเปรียบเสมือนตัวแปรที่ใช้ในการอธิบายวงจรลักษณะนี้ แทนที่จะเป็น Nets

#### รีจิสเตอร์ (Registers)

ความหมายโดยทั่วไปของ Registers ในภาษา Verilog ก็คือเป็นตัวที่ใช้ในการเก็บค่าต่าง ๆ ซึ่งเปรียบได้กับ Variables ในภาษาชั้นสูงทั่วไป เช่น C หรือ Pascal

วิธีการกำหนดการใช้งานของ Registers จะใช้คำสั่ง **reg** นำหน้าชื่อของสัญญาณที่จะกำหนด สำหรับการใช้งานปกติที่ใช้เก็บค่าต่าง ๆ ของสัญญาณลอจิก นอกจากนี้เรายังสามารถกำหนดการใช้งานของตัวแปรรีจิสเตอร์ประเภทอื่น ๆ สำหรับการใช้งานในภาษา Verilog เพื่อให้มีความสะดวกมากยิ่งขึ้น คือ **integer**, **time**, **real**, **realtime** ดังรายละเอียดต่อไปนี้

- **reg**: ใช้ในการเก็บค่าลอจิก คล้ายกับ Flip-flop ในฮาร์ดแวร์ แต่ไม่จำเป็นต้องมีสัญญาณนาฬิกา มาใช้ในการเก็บค่า ค่าเริ่มต้นของ register จะเป็น 'x' (ไม่เหมือนกับ Nets ที่เป็น 'z') เช่น
  - **reg** A;
  - **reg** [3:0] B, C;
- **time**: เป็นรีจิสเตอร์ประเภทพิเศษ ที่ใช้สำหรับการจำลองการทำงาน (Simulation) การแก้ไข (Debugging) รวมทั้งการรายงาน (Reporting) เป็นตัวแปรที่มีความกว้าง 64 บิต เช่น
  - **time** sim\_time;
  - **time** setup\_time;
- **integer**: เป็นรีจิสเตอร์ที่ใช้ในทางคณิตศาสตร์ สำหรับการเก็บค่าจำนวนเต็ม สามารถเป็นไปได้ทั้งจำนวนบวก และลบ โดยตัวแปรรีจิสเตอร์นี้มีขนาดเท่ากับ 32 บิต ต่างกับ **reg** คือถ้ามีค่าลบและใช้งานใน Expression ค่าของมันก็ยังเป็นลบอยู่ (ต่างกับ **reg** จะมีค่าเป็นบวกเสมอ) เช่น
  - **integer** loop\_count;
  - **integer** counter;
- **real, realtime**: เป็นรีจิสเตอร์ที่ใช้ในทางคณิตศาสตร์ สำหรับการเก็บค่าจำนวนจริงที่มีจุดทศนิยม เขียนได้ทั้งเป็นเลขยกกำลังในแบบของ Scientific form หรือจุดทศนิยมธรรมดา โดยที่ **realtime** จะใช้ในการแทนค่าทางเวลา แต่ทั้งสองประเภทนี้สามารถใช้แทนกันได้ เช่น
  - **real** exact, average;
  - **realtime** exact\_simtime;

### เวกเตอร์ และอาร์เรย์ (Vectors and Arrays)

เราสามารถกำหนด **reg** เป็นตัวแปรประเภทเวกเตอร์ (Vector) ได้ โดยกำหนดช่วงบิตที่จะใช้งาน (Indexing) อยู่ในเครื่องหมาย [ MSB: LSB] ที่อยู่ระหว่างคำว่า **reg** และ *Instance name* เช่น

```
reg [7:0] Data; //เป็นเวกเตอร์ชื่อ Data ที่มีขนาดความกว้าง 8 บิต
```

อีกประเภทหนึ่งของรีจิสเตอร์ที่สามารถทำได้ คือตัวแปรแบบอาร์เรย์ (Array) — ที่ไม่มีอยู่ในสัญญาณประเภท Nets — ความแตกต่างที่ชัดเจนระหว่างเวกเตอร์ และอาร์เรย์คือ มิติ โดยที่เวกเตอร์มีเพียงมิติเดียวนั้นคือความกว้าง (Width) แต่อาร์เรย์มีสองมิติคือทั้งความกว้าง (Width) และความลึก (Depth) หรืออาจมองอีกนัยหนึ่งคือ อาร์เรย์เป็นการนำเอาเวกเตอร์มาเรียงซ้อน ๆ กัน

สำหรับวิธีการกำหนดอาร์เรย์ จะมีการกำหนดช่วงความกว้างอยู่หลังจากคำว่า **reg** และความลึกหลังจาก *Instance name* เช่น

```
reg Data[7:0];
//เป็นการกำหนดอาร์เรย์ชื่อ Data ที่มีความกว้าง 1 บิต และความลึก 8 ตำแหน่ง
reg [7:0] MyMem [3:0];
//เป็นอาร์เรย์ชื่อ MyMem มีความกว้าง 8 บิตและความลึก 4 ตำแหน่ง
```

ความแตกต่างระหว่างเวกเตอร์ และอาร์เรย์อีกอย่างคือ เวกเตอร์สามารถเป็นรีจิสเตอร์ประเภท **reg** เท่านั้น แต่อาร์เรย์สามารถประกอบด้วยรีจิสเตอร์ประเภทต่าง ๆ เช่น **reg**, **integer** หรือ **time** แต่ไม่ใช่ **real** หรือ **realtime**

สำหรับการอ้างอิงถึงในแต่ละตำแหน่งของอาร์เรย์นั้นสามารถกำหนดได้โดยตรง เช่น `MyMem[2]` ; เป็นการอ้างอิงถึงตำแหน่งที่สองของอาร์เรย์ `MyMem` (ดูรูป) แต่กรณีที่เราต้องการอ้างอิงถึงระดับบิต เราไม่สามารถทำได้โดยตรง เราจำเป็นต้องมีตัวแปรเวกเตอร์ชั่วคราว (`TempReg`) มารับค่าของอาร์เรย์ในตำแหน่งที่ต้องการก่อน แล้วจึงอ้างอิงถึงบิตที่ต้องการในเวกเตอร์ชั่วคราว

```
reg [7:0] MyMem [3:0];
reg [7:0] TempReg;
```

0	1	1	1	0	1	1	1
1	1	1	0	1	1	1	0
0	1	...	0	0	1	0	0
0	0	1	0	0	1	1	1

```
TempReg = MyMem[2];
TempReg[2];
```

### ค่าคงที่ในภาษา Verilog (Constants in Verilog)

เหตุผลของการกำหนดตัวเลข ๑ หนึ่ง ให้เป็นค่าคงที่ที่มีชื่อ ๑ หนึ่งเพื่อให้่ายในการทำมาเข้าใจ และสามารถเรียกใช้งานได้หลาย ๑ ครั้ง นอกจากนี้นี้ยังง่ายในการแก้ไขค่า โดยที่เราสามารถเปลี่ยนแปลงค่าได้จากที่เดียว แล้วส่วนอื่นของโปรแกรมที่ใช้ค่าที่นั้น ๑ ก็จะไม่เปลี่ยนแปลงตาม โดยที่เราไม่ต้องเสียเวลาไปไล่เปลี่ยนทีละบรรทัด ซึ่งง่ายในการผิดพลาด และช้า

ค่าคงที่จะไม่สามารถเปลี่ยนแปลงค่า หรือกำหนดค่าให้ใหม่ได้ในโมดูล ดังนั้นเราจึงไม่สามารถใช้ค่าคงที่ในลักษณะของตัวแปรได้ ทำให้ค่าคงที่ในภาษา Verilog จึงไม่ถูกจัดอยู่ในประเภทของทั้ง Nets และ Registers แต่มันถูกเรียกชื่อใหม่ว่า **parameter**

```
parameter BusSize = 8;
reg [BusSize-1 : 0 ] DataBus;
. . .

for (cntr=0; cntr<BusSize-1; cntr=cntr+1)
. . .

for (cntr=BusSize; cntr>=0; cntr=cntr-1)
. . .
```

ตัวอย่างในรูป เราให้ `BusSize` เป็นพารามิเตอร์ ที่ใช้กำหนดขนาดของบิต `DataBus` นอกจากนี้ มันยังใช้เป็นเงื่อนไขในการทำคำสั่ง `for loop` ในตัวโปรแกรมอีกด้วย สมมุติว่าเราต้องการเปลี่ยนขนาดของบิต และเงื่อนไขจากตัวพารามิเตอร์

`BusSize` จาก 8 เป็น 16 เราสามารถทำได้ที่บรรทัด `parameter BusSize = 16;` ที่เดียว

### การกำหนด และการใช้งานพารามิเตอร์ (The Declaration and Use of Parameters)

การกำหนดพารามิเตอร์สำหรับใช้งานในโปรแกรมสามารถทำได้ดังนี้

- เริ่มต้นด้วยคำสั่ง **parameter**
- ชื่อของพารามิเตอร์
- ค่าของพารามิเตอร์ที่กำหนดไว้หลังจากเครื่องหมาย '='

- เครื่องหมาย ; (Semi colon) สำหรับการจบคำสั่ง

เราสามารถกำหนดค่าคงที่ได้มากกว่าหนึ่ง ในการกำหนดครั้งเดียว โดยใช้เครื่องหมาย , (Comma) แยกชุดของค่าคงที่แต่ละตัว (ชื่อพารามิเตอร์ พร้อมค่าที่กำหนด)

สำหรับวิธีการใช้งานพารามิเตอร์ สามารถใช้ได้ในลักษณะต่าง ๆ เช่น

- การกำหนดขนาดของเวกเตอร์ อาร์เรย์ เช่น บัส หน่วยความจำ

description
Parameter as size of an object
parameter declaration
<b>parameter</b> BusWidth = 8;
use in Verilog code
<b>reg</b> [BusWidth-1:0] DataBus;

- การกำหนดค่าของดีเลย์ หรือข้อมูลเวลา เช่น

description
Parameter as a timing parameter
parameter declaration
<b>parameter</b> PropDel = 3;
use in Verilog code
<b>assign</b> #PropDel Y = X;

- การกำหนดจำนวนรอบในการวนคำสั่ง *for-loop* ดังตัวอย่าง

description
Parameter as a loop counter
parameter declaration
<b>parameter</b> LoopIterations= 3;
use in Verilog code
<b>for</b> (k=0; k<LoopIterations; k=k+1);

## การอธิบายพฤติกรรมเบื้องต้น (Behavioral Basics)

### จาก Dataflow ถึง Behavioral (From Dataflow to Behavioral)

ในการอธิบายพฤติกรรมการทำงานของวงจร จำเป็นจะต้องกระทำเป็นส่วน ๆ ที่เราเรียกว่า Block โดยมีคำสั่งต่าง ๆ ที่เป็น Procedure statements สำหรับการกำหนดการทำงานในแต่ละส่วน ในโมดูลหนึ่ง อาจมีได้หลาย ๆ Blocks ซึ่งแต่ละ Block ทำงานเป็นอิสระต่อกันแบบขนาน (Concurrency) เหมือนกับ Continuous assignments ในบทที่ผ่านมา เมื่อสัญญาณใดสัญญาณหนึ่งใน Block มีการเปลี่ยนแปลง Block นั้นก็จะมีการทำงาน หรือถูกประมวลผล โดยคำสั่งภายในจะถูกแปลตามความหมาย แบบเรียงลำดับ (Sequential) ตามลักษณะของคำสั่ง

ชนิดของ Blocks สามารถแบ่งออกได้เป็นสองลักษณะคือ

- Initial blocks
- Always blocks

เมื่อ Block ใด ๆ มีคำสั่งหลาย ๆ คำสั่ง เราสามารถรวมเป็นกลุ่มของชุดคำสั่งได้ ด้วยคำว่า **begin** แล้วจบด้วย **end** สำหรับการทำงานแบบเรียงลำดับ (Sequential statement execution) หรือคำว่า **fork** และ **join** สำหรับการทำงานแบบขนาน (Concurrent statement execution)

```
module DataFlow;
...

assign...;
assign...;
assign...;
assign...;
...
endmodule
```

```
module Behavior
...
initial
...
always
...
always
...
assign...;
...
endmodule
```

ตัวอย่างด้านซ้ายมือจะเป็นการเขียนด้วย Continuous assignment สำหรับการอธิบายด้วย Dataflow style และด้านขวามือจะเป็นการอธิบายด้วย Behavioral style โดยใช้คำสั่ง **initial** และ **always**

ในแต่ละ Block จะทำงานพร้อม ๆ กัน รวมทั้งพร้อมกับ Continuous assignment (assign ...;) ในบรรทัดสุดท้ายด้วย

### Behavioral Blocks

ทั้งสองชนิดของ Behavioral blocks มีลักษณะการทำงานเหมือนกัน ยกเว้น **initial** block จะถูกคอมไพล์ที่ตอนเริ่มต้นของการจำลองการทำงาน (time 0) และกระทำเพียงครั้งเดียวเท่านั้น ส่วน **always** block มีการเริ่มการทำงานที่เวลาเท่ากับ 0 เหมือนกัน แต่จะวนรอบไปเรื่อย ๆ ไม่มีสิ้นสุด (Infinite loop) トラバเท่ากับกระบวนการจำลองการทำงานยังมีอยู่ โดยที่ทุก Blocks ในโมดูลจะทำงานพร้อม ๆ กันจากเวลาเริ่มต้น

สำหรับค่าปริยายของรีจิสเตอร์ จะมีค่าเป็น High impedance ('z') ซึ่งในการจำลองการทำงานให้ถูกต้องจำเป็นต้องเซตค่าเริ่มต้นให้ก่อน อาจจะเป็น '0' หรือค่าที่เหมาะสม โดยใช้คำสั่ง **initial** นั่นเอง ส่วนหน้าที่ของคำสั่ง **always** เป็นการเริ่มต้นการอธิบายพฤติกรรมการทำงานของวงจรในโมดูล

ถึงแม้ว่าในแต่ละ Blocks จะทำงานพร้อม ๆ กัน แต่ในการเขียนเป็นโปรแกรมเราจำเป็นต้องเขียนเรียงกันลงมาที่ละบรรทัด เนื่องจากเราไม่สามารถเขียนทุก Blocks ได้พร้อม ๆ กัน ดังรูป

```
module Behavior;
initial
always begin
always
always begin
initial begin
endmodule
```



```
module Behavior;
initial
always begin
end
always
always begin
end
initial begin
end
endmodule
```

## การกำหนดค่าตัวแปร (Variable Assignment)

สิ่งสำคัญที่สุดในการกระทำให้การอธิบายพฤติกรรมการทำงาน คือการส่งผ่านค่าจาก Expression ไปยังตัวแปร การส่งผ่านค่านี้เราเรียกว่า Procedural assignment ซึ่งแตกต่างจาก Continuous assignment ดังต่อไปนี้

- ตัวแปรทางด้านซ้ายมือของการส่งผ่านค่า (The target of assignment) จะต้องเป็น Registers (**reg**, **integer**, **real**, **time**), บิต หรือส่วนหนึ่งของรีจิสเตอร์เวกเตอร์ โดยที่ไม่สามารถเป็น Nets ได้
- ไม่มีคำว่า **assign** ในการส่งผ่านค่า
- ต้องกำหนดภายใน Behavioral block (**initial** or **always**)
- ต้องเป็นสัญญาณชนิดเดียวกัน
- การส่งผ่านค่าแบบ Continuous assignment จะมีการเปลี่ยนแปลงค่าใหม่เมื่อมีการเปลี่ยนแปลงค่าของสัญญาณใดสัญญาณหนึ่งใน Expression ทางด้านขวามือ แต่สำหรับ Procedural assignment ใน **always** จะมีการประมวลผลค่าใหม่เมื่อมีการเปลี่ยนแปลงค่าของสัญญาณใดสัญญาณหนึ่งใน Sensitivity list ที่อยู่ข้างหลัง **always**

```
reg A;
integer LoopCount;
reg [7:0] VecM;

initial
begin
  A = 1'b0;
  LoopCount = 0;
  VecM = 8'b0;
end
```

## การใช้งานที่ซับซ้อนมากขึ้น (Complex Statements)

การใช้งานในลักษณะของการส่งผ่านค่า (Assignments) มักไม่เพียงพอสำหรับการใช้งานในการอธิบายพฤติกรรมการทำงานของระบบ เราอาจต้องใช้เงื่อนไขของการกระทำ (Conditional statements), เงื่อนไขที่มีหลาย ๆ ทางเลือก (Multi-branch choices), หรือ การวนรอบ (Loops) ที่คล้าย ๆ กับที่มีอยู่ในภาษาซี

### Conditional Operations

การกระทำบางอย่างจะเกิดขึ้น เมื่อมีเงื่อนไขที่ถูกต้องเกิดขึ้นก่อน การกระทำลักษณะนี้ เราเรียกว่า Conditional operations ซึ่งในภาษา Verilog มีไวยากรณ์ดังนี้คือ

```
if (expression_true) true_statement;
    else false_statement;
```

การทำงานสามารถอธิบายได้ดังนี้คือ เมื่อ *expression\_true* มีค่าเป็นจริง *true\_statement* จะมีการทำงาน แต่ถ้าเป็นเท็จ *false\_statement* จะมีการทำงานแทน

แต่ถ้าค่าจาก *expression\_true* เป็น 'x' หรือ 'z' จะมีการทำงานโดย *false\_statement* โดยปริยาย

ถ้า *true(false)\_statement* ประกอบด้วย Statements มากกว่าหนึ่ง จะต้องกำหนดจุดเริ่มต้นของกลุ่ม Statements นั้น ด้วยคำสั่ง **begin** และจบท้ายด้วย **end** แต่ถ้ามีแค่หนึ่ง Statement เราอาจไม่จำเป็นต้องใช้ก็ได้

```

module ShiftReg (Outs, Ins, Clk, Clr, Set, Shr, Shl);
parameter Size = 8;
parameter MSB = Size - 1;
output [MSB:0] Outs; reg [MSB:0] Outs;
input [MSB:0] Ins;
input Clk, Clr, Set, Shl, Shr;

initial
    Outs = 0;

always @ (posedge Clk)
    if (Clr == 1) Outs = 0;
    else if (Set == 1) Outs = {Size{1'b1}};
    else if (Shl == 1) Outs = Outs << 1;
    else if (Shr == 1) Outs = Outs >> 1;
    else Outs = Ins;

endmodule

```

ตัวอย่างเป็นการสร้างชิฟรืจิสเตอร์ โดยใช้คำสั่งเงื่อนไข แบบซ้อนกันหลาย ๆ ชั้น เนื่องจากมีหลาย ๆ เงื่อนไข คือ Clr, Set, Shr, Shl สำหรับการเคลียร์ค่าเอาต์พุต การเซตค่าเอาต์พุต การชิฟไปด้านขวา และการชิฟไปด้านซ้าย ตามลำดับ

### การมีตัวเลือกหลายกรณี (Multiple Choice)

การสร้างเงื่อนไขแบบซ้อน ๆ กันหลาย ๆ กรณีโดยใช้คำสั่ง *if-else-if* มีข้อเสียคือซับซ้อน และยากในการทำความเข้าใจ อีกทางเลือกหนึ่งสำหรับการเขียนอธิบายพฤติกรรมของวงจร ที่มีการกระทำแบบมีหลาย ๆ เงื่อนไข คือการใช้คำสั่ง **case**

สัญญาณที่เป็นเงื่อนไขในแต่ละเงื่อนไข จะอยู่ภายในวงเล็บ ตามหลังคำสั่ง **case** โดยบรรทัดต่อมาจะเป็นค่าในแต่ละกรณีที่เป็นไปได้ ตามด้วยการกระทำ (Statements) ถ้าเงื่อนไขในกรณีนั้นถูกต้อง ดังตัวอย่าง

```

module ShiftReg (Outs,Ins,Clk,Clr,Set,Shl,Shr);
parameter Size = 8;
parameter MSB = Size - 1;
output [MSB:0] Outs; reg [MSB:0] Outs;
input [MSB:0] Ins;
input Clk, Clr, Set, Shl, Shr;

initial
    Outs = 0;

always @ (posedge Clk)
    case ({Clr, Set, Shl, Shr})
        4'b1xxx : Outs = 0;
        4'bx1xx : Outs = {Size{1'b1}};
        4'bxx1x : Outs = Outs << 1;
        4'bxxx1 : Outs = Outs >> 1;
        default : Outs = Ins;
    endcase
endmodule

```

จะเห็นได้ว่ามีคำสั่ง **default** ที่เป็นค่าสำหรับการกระทำ (Outs = Ins) โดยปริยาย กรณีที่สัญญาณไม่ตรงกับเงื่อนไขใด ๆ ที่กำหนด

นอกจากคำสั่ง **case** แล้ว เรายังมี **casex** และ **casez** โดยมีความแตกต่างกันในแง่ของการแปลความหมายของบิตที่เป็น 'x' และ 'z' ดังนี้

- ในคำสั่ง **case** ความหมายของบิต 'x' หรือ 'z' ก็คือค่าที่เป็น Unknown และ High impedance ตามปกติ
- สำหรับ **casex** จะมอง 'x' และ 'z' ในความหมายของ Don't care ทั้งหมด
- ในกรณีของ **casez** จะแปล 'x' เป็น Unknown และ 'z' เป็น Don't care

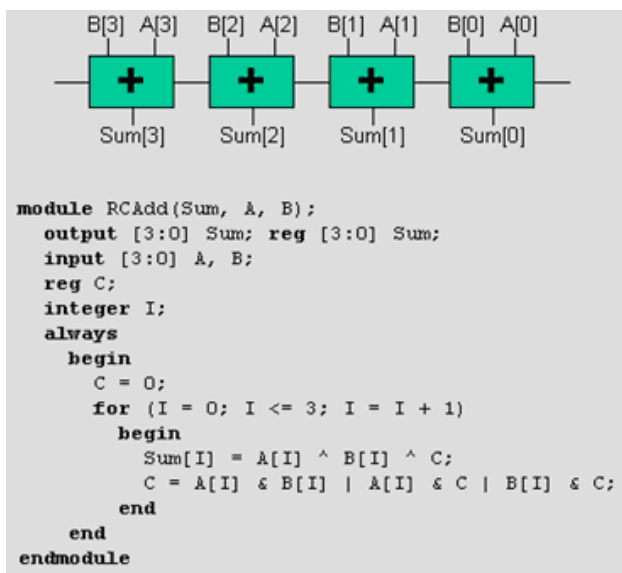
### การวนรอบ (Loops)

จุดประสงค์ของการใช้คำสั่งในการวนรอบ เพื่อให้มีการทำซ้ำกันหลาย ๆ รอบตามจำนวนครั้งที่เราต้องการ ซึ่งช่วยให้เราเขียนโปรแกรมได้สั้นมากขึ้น และสามารถกำหนดจำนวนรอบของการทำซ้ำได้ง่ายมากขึ้นด้วย

จำนวนรอบของการกระทำ กำหนดได้ด้วยเงื่อนไข (ในการหยุด) ในตัวคำสั่ง โดยที่การวนรอบจะเกิดขึ้นเรื่อย ๆ จนกระทั่งเงื่อนไขนั้นเป็นจริง

ภาษา Verilog มีคำสั่งที่ใช้ในการวนรอบดังนี้คือ **forever**, **repeat**, **while** และ **for** ซึ่งโครงสร้างไวยากรณ์คล้ายกับภาษาซี โดยมีรายละเอียดการใช้งานดังนี้

- **forever**: เป็นคำสั่งในการวนรอบแบบไม่มีที่สิ้นสุด
- **repeat**: เป็นคำสั่งในการวนรอบว่าต้องการกี่ครั้ง โดยมีจำนวนครั้งกำหนดเป็นค่าคงที่เอาไว้
- **while**: เป็นการวนรอบแบบมีเงื่อนไขสำหรับการเริ่มต้นในการวน โดยไม่มีจำนวนครั้งที่แน่นอน และในแต่ละรอบจะมีการตรวจเงื่อนไขในการทำงานก่อนทุกครั้ง
- **for**: เป็นคำสั่งที่นิยมใช้กันมากที่สุด เนื่องจากมีความยืดหยุ่นสูง ในการใช้งานจะมีการกำหนดเงื่อนไขเริ่มต้น เงื่อนไขในการหยุด และการปรับปรุงค่าตัวแปรที่ใช้ในการควบคุมการวนรอบ



ในรูปเป็นตัวอย่างการใช้ **for-loop** ในการวนรอบเพื่อสร้างตัวบวกที่มีจำนวนบิตเท่ากับการวนรอบ ซึ่งในที่นี้คือ 4-bit full adder สังเกตได้ว่าตัวแปร **I** จะเป็นตัวควบคุมการวนรอบที่มีค่าเริ่มต้น เท่ากับ 0 และจะเพิ่มขึ้น 1 ( $I = I + 1$ ) เมื่อมีการวนรอบหนึ่งครั้ง จำนวนในการวนรอบจำกัดอยู่ที่  $I \leq 3$  ดังนั้นจะมีการทำงานอยู่ 4 ครั้งด้วยกัน ( $I = 0, 1, 2, 3$ )

สมมุติว่าเราต้องการเปลี่ยนเป็น 8-bit full adder สามารถทำได้โดยแค่เปลี่ยน  $I \leq 3$  เป็น  $I \leq 7$  ที่เดียวเท่านั้น



## การควบคุมพฤติกรรมของวงจรขั้นสูง (Advance Control over Behavior)

### เหตุการณ์ (Events)

เราใช้เครื่องหมาย @ สำหรับการกำหนดการเกิดขึ้นของเหตุการณ์ (Event control statement) สามารถทำได้โดยใช้ @ ตามด้วยชื่อของ Registers หรือ Nets ดังตัวอย่างต่อไปนี้

```
@ (posedge CLK) Q = D;  
@ (negedge CLK) Q = D;
```

**posedge** และ **negedge** เป็นคำเริ่มต้น (Prefix) ที่กำหนดเหตุการณ์ในช่วงขอบขาขึ้น หรือขอบขาลงของสัญญาณนั้น ๆ เช่น จะทำการส่งผ่านค่า (Assignment)  $Q = D$  ที่ขอบขาขึ้นของ CLK เป็นต้น

### คำสั่งรอ (Wait Statement)

การควบคุมเหตุการณ์ (Event control) เป็นการควบคุมการเปลี่ยนแปลงค่าสัญญาณ หรือตัวแปร โดยอาจจะมีเงื่อนไข หรือช่วงเวลามาเกี่ยวข้อง

คำสั่ง **wait** สามารถนำมาใช้ควบคุมการกระทำอย่างมีเงื่อนไข เช่นอาจใช้สร้างตัว 3-state buffer หรือตัว Level-sensitive event control สำหรับการใช้งานจะต้องมีเงื่อนไขการรอ (Enable) อยู่ข้างในวงเล็บหลังจากคำสั่ง **wait** แล้วตามด้วย Statements

```
wait (enable) statement;
```

เช่น wait (EN) #5 C=A+B; สามารถแปลได้ดังนี้คือ การกระทำ  $C = A+B$ ; จะเกิดขึ้นภายหลัง 5 หน่วยเวลา หลังจากที่สัญญาณ EN เป็น '1'

แต่ถ้าใช้ wait ; โดยไม่มีเงื่อนไขใด ๆ ทั้งสิ้น จะเป็นการให้รอโดยไม่มีที่สิ้นสุด หรือเป็นการหยุดการจำลองการทำงานนั่นเอง

### Sensitivity List

เมื่อไรก็ตามที่คำสั่งถูกกระทำโดยเหตุการณ์ที่เกิดขึ้นในสัญญาณหนึ่ง เรากรณีนี้ว่าคำสั่งนั้นไว (Sensitive) ต่อสัญญาณนั้น ๆ ซึ่งในภาษา Verilog เองไม่จำกัดว่าการไวต่อสัญญาณจะเป็นสัญญาณเดียวหรือหลาย ๆ สัญญาณ ตามแต่ที่จะปรากฏในคำสั่ง หรือชุดคำสั่งนั้น ๆ และเราสามารถกำหนดรายชื่อของสัญญาณต่าง ๆ เหล่านั้นได้ (โดยใช้ **or** เป็นตัวกัน) เรียกว่า *Sensitivity list*

การกำหนด Sensitivity signals ส่วนมากจะใช้ในคำสั่ง **always** สำหรับการควบคุมการทำงานของชุดคำสั่งต่าง ๆ ที่อยู่ระหว่าง **begin** และ **end** ใน **always** block นั่นคือ ถ้าหากตัวใดตัวหนึ่งใน Sensitivity list เกิดการเปลี่ยนแปลง ตัว **always** block ก็จะมีการประมวลผล และปรับปรุงค่าเกิดขึ้น

ตัวอย่างเช่น การสร้าง D-flipflop ที่มีขอบขาขึ้น (**posedge**) ของสัญญาณ CLK และ RST เป็นตัวกำหนดการทำงาน ในคำสั่ง *if-then-else* สามารถเขียนได้ดังนี้

```

always @(posedge RST or posedge CLK)
begin
    if (RST)
        Q = 1'b0;
    else if (posedge CLK)
        Q = D;
end

```

### การส่งผ่านค่าแบบ Non-blocking (Non-blocking Assignment)

การส่งผ่านค่าแบบ Non-blocking assignment เป็นวิธีการกำหนดการส่งผ่านค่าในทุก ๆ บรรทัดที่เขียนเรียงกันนั้น เกิดขึ้นพร้อม ๆ กัน การใช้งานในลักษณะนี้จะใช้เครื่องหมาย '<=' แทนที่จะเป็น '=' (Blocking assignment)

เหตุผลที่เราต้องการ Non-blocking assignment เพื่อให้ทุก ๆ การส่งผ่านค่าเกิดขึ้นพร้อม ๆ กันเป็นลักษณะของ Concurrent data transfer ซึ่งสามารถหลีกเลี่ยงการเกิด Race condition จากการใช้ Blocking assignment

ตัวอย่างการทำงานของ **always** block ที่มีการใช้ Blocking และ Non-blocking assignment จะเห็นได้ว่า Blocking assignment จะมีการ Update ค่าทันทีหลังจากจบบรรทัด แต่ Non-blocking การ Update ค่าของตัวแปรทั้ง x, y, z จะเกิดขึ้นพร้อม ๆ กันหลังจากเจอคำสั่ง **end**

- Blocking assignment:

```

always @ (a or b or c)
begin
    x = a | b;           1. Evaluate a | b, assign result to x
    y = a ^ b ^ c;       2. Evaluate a^b^c, assign result to y
    z = b & ~c;          3. Evaluate b&(~c), assign result to z
end

```

- Non-blocking assignment:

```

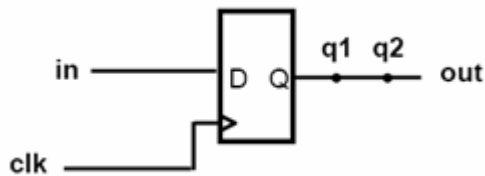
always @ (a or b or c)
begin
    x <= a | b;           1. Evaluate a | b but defer assignment of x
    y <= a ^ b ^ c;       2. Evaluate a^b^c but defer assignment of y
    z <= b & ~c;          3. Evaluate b&(~c) but defer assignment of z
end                       4. Assign x, y, and z with their new values

```

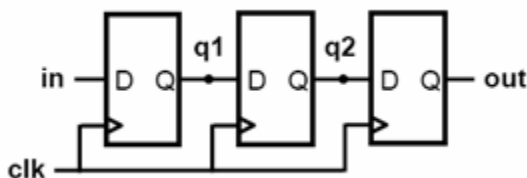
ผลลัพธ์จากการใช้ Assignment ต่างชนิดกัน บางครั้งจะเหมือนกันดังตัวอย่างข้างบน แต่บางครั้งจะไม่เหมือนกัน เช่นตัวอย่างต่อไปนี้

<pre> module nonblocking(in, clk, out);     input in, clk;     output out;     reg q1, q2, out;     always @ (posedge clk)     begin         q1 &lt;= in;         q2 &lt;= q1;         out &lt;= q2;     end endmodule </pre>	<pre> module blocking(in, clk, out);     input in, clk;     output out;     reg q1, q2, out;     always @ (posedge clk)     begin         q1 = in;         q2 = q1;         out = q2;     end endmodule </pre>
---	--

จากโปรแกรมจะเป็นการสร้างซีพรีจิสเตอร์ใน **always** block โดยการใช้การส่งผ่านค่าในลักษณะของ Blocking และ Non-blocking assignment เปรียบเทียบกัน กรณีกำหนดโดยใช้ Blocking assignment '=' (ในโมดูลทางด้านขวามือ) ค่าในแต่ละบรรทัดจะถูกปรับปรุงค่าทันที นั่นคือ  $q1 = in$ ; และหลังจากนั้น  $q2 = q1$ ; แต่เนื่องจาก  $q1$  ถูกปรับปรุงค่าเป็น  $in$  เรียบร้อยแล้วจากบรรทัดก่อนหน้านี้ ดังนั้น  $q2 = q1 = in$ ; ทำนองเดียวกันในบรรทัดต่อมา  $out = q2 = q1 = in$ ; จะเห็นได้ว่าสัญญาณทุกตัวมีค่าเท่ากันหมด ตัวสังเคราะห์วงจรจึงแปลงเป็นวงจรได้ดังนี้



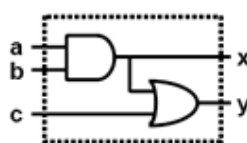
แต่ถ้าใช้ Non-blocking assignment '<=' ที่มีการปรับปรุงค่าทีละตัวพร้อม ๆ กัน จะได้วงจรซีพรีจิสเตอร์ตามต้องการ ดังรูปต่อไปนี้



เพื่อป้องกันการเกิดกรณีที่ไมต้องการอย่างนี้เกิดขึ้น ง่ายๆ ก็คือ ถ้ามีการส่งผ่านค่าใน **always** block ให้ใช้ Non-blocking assignment '<=' เสมอ ยกเว้นการสร้าง Combinational circuit ใน **always** block ให้ใช้ '=' ดังตัวอย่างในรูปข้างล่าง (ซึ่งจริง ๆ เรานิยมทำเป็น Continuous assignment ที่อยู่ข้างนอก **always** block มากกว่า)

### Blocking Behavior

	a	b	c	x	y
(Given) Initial Condition	1	1	0	1	1
a changes; always block triggered	0	1	0	1	1
x = a & b;	0	1	0	0	1
y = x   c;	0	1	0	0	0



```
module blocking(a,b,c,x,y);
  input a,b,c;
  output x,y;
  reg x,y;
  always @ (a or b or c)
  begin
    x = a & b;
    y = x | c;
  end
endmodule
```

### Nonblocking Behavior

	a	b	c	x	y	Deferred
(Given) Initial Condition	1	1	0	1	1	
a changes; always block triggered	0	1	0	1	1	
x <= a & b;	0	1	0	1	1	x<=0
y <= x   c;	0	1	0	1	1	x<=0, y<=1
Assignment completion	0	1	0	0	1	

```
module nonblocking(a,b,c,x,y);
  input a,b,c;
  output x,y;
  reg x,y;
  always @ (a or b or c)
  begin
    x <= a & b;
    y <= x | c;
  end
endmodule
```

## Quick Reference

for

## Verilog<sup>®</sup> HDL

---

1.0	Lexical Elements .....	1
1.1	Integer Literals .....	1
1.2	Data Types .....	1
2.0	Registers and Nets .....	2
3.0	Compiler Directives .....	3
4.0	System Tasks and Functions .....	4
5.0	Reserved Keywords .....	5
6.0	Structures and Hierarchy .....	6
6.1	Module Declarations .....	6
6.2	UDP Declarations .....	7
7.0	Expressions and Operators .....	10
7.1	Parallel Expressions .....	13
7.2	Conditional Statements .....	13
7.3	Looping Statements .....	15
8.0	Named Blocks, Disabling Blocks .....	16
9.0	Tasks and Functions .....	16
10.0	Continuous Assignments .....	18
11.0	Procedural Assignments .....	18
11.1	Blocking Assignment .....	19
11.2	Non-Blocking Assignment .....	19
12.0	Gate Types, MOS and Bidirectional Switches .....	19
12.1	Gate Delays .....	21
13.0	Specify Blocks .....	22
14.0	Verilog Synthesis Constructs .....	23
14.1	Fully Supported Constructs .....	23
14.2	Partially Supported Constructs .....	24
14.3	Ignored Constructs .....	25
14.4	Unsupported Constructs .....	25
15.0	Index .....	27

---

Rajeev Madhavan  
AMBIT Design Systems, Inc.

## 1.0 Lexical Elements

The language is case sensitive and all the keywords are lower case. White space, namely, spaces, tabs and new-lines are ignored. Verilog has two types of comments:

1. One line comments start with `//` and end at the end of the line
2. Multi-line comments start with `/*` and end with `*/`

Variable names have to start with an alphabetic character or underscore followed by alphanumeric or underscore characters. The only exception to this are the system tasks and functions which start with a dollar sign. Escaped identifiers (identifier whose first character is a backslash ( `\` )) permit non alphanumeric characters in Verilog name. The escaped name includes all the characters following the backslash until the first white space character.

### 1.1 Integer Literals

```
Binary literal  2'b1z
Octal literal  2'o17
Decimal literal 9 or 'd9
Hexadecimal literal 3'h189
```

Integer literals can have underscores embedded in them for improved readability. For example,

```
Decimal literal 24_000
```

### 1.2 Data Types

The values `z` and `Z` stand for high impedance, and `x` and `X` stand for uninitialized variables or nets with conflicting drivers. String symbols are enclosed within double quotes ( `"string"` ).and cannot span multiple lines. Real number literals can be either in fixed notation or in scientific notation.

#### Real and Integer Variables example

```
real a, b, c ; // a,b,c to be real

integer j, k ; // integer variable
integer i[1:32] ; // array of integer variables
```

### Time, registers and variable usage

```
time newtime ;
/* time and integer are similar in functionality,
time is an unsigned 64-bit used for time variables
*/

reg [8*14:1] string ;
/* This defines a vector with range
[msb_expr: lsb_expr] */

initial begin
  a = 0.5 ; // same as 5.0e-1. real variable
  b = 1.2E12 ;
  c = 26.19_60_e-11 ; // _'s are
                    // used for readability
  string = " string example " ;
  newtime =$time;
end
```

## 2.0 Registers and Nets

A register stores its value from one assignment to the next and is used to model data storage elements.

```
reg [5:0] din ;
/* a 6-bit vector register: individual bits
din[5],... din[0] */
```

Nets correspond to physical wires that connect instances. The default range of a wire or `reg` is one bit. Nets do not store values and have to be continuously driven. If a net has multiple drivers (for example two gate outputs are tied together), then the net value is resolved according to its type.

#### Net types

wire	tri
wand	triand
wor	trior
tri0	tril
supply0	supply1
triereg	

For a wire, if all the drivers have the same value then the wire resolves to this value. If all the drivers except one have a value of `z` then the wire resolves to the non `z` value. If two or more non `z` drivers have different drive strength, then the wire resolves to the stronger driver. If two drivers of equal strength have different values, then the

wire resolves to `x`. A `triereg` net behaves like a wire except that when all the drivers of the net are in high impedance (`z`) state, then the net retains its last driven value. `triereg` 's are used to model capacitive networks.

```
wire net1 ;
/* wire and tri have same functionality. tri is
used for multiple drive internal wire */

triereg (medium) capacitor ;
/* small, medium, weak are used for charge
strength modeling */
```

A `wand` net or `triand` net operates as a wired and(`wand`), and a `wor` net or `trior` net operates as a wired or (`wor`), `tri0` and `tril` nets model nets with resistive pulldown or pullup devices on them. When a `tri0` net is not driven, then its value is 0. When a `tril` net is not driven, then its value is 1. `supply0` and `supply1` model nets that are connected to the ground or power supply.

```
wand net2 ; // wired-and
wor net3 ; // wired-or
triand [4:0] net4 ; // multiple drive wand
trior net5 ; // multiple drive wor
tri0 net6 ;
tril net7 ;
supply0 gnd ; // logic 0 supply wire
supply1 vcc ; // logic 1 supply wire
```

Memories are declared using register statements with the address range specified as in the following example,

```
reg [15:0] mem16X512 [0:511];
// 16-bit by 512 word memory
// mem16X512[4] addresses word 4
// the order lsb:msb or msb:lsb is not important
```

The keyword `scalared` allows access to bits and parts of a bus and `vectored` allows the vector to be modified only collectively.

```
wire vectored [5:0] neta;
/* a 6-bit vectored net */
tril vectored [5:0] netb;
/* a 6-bit vectored tril */
```

## 3.0 Compiler Directives

Verilog has compiler directives which affect the processing of the input

files. The directives start with a grave accent ( ` ) followed by some keyword. A directive takes effect from the point that it appears in the file until either the end of all the files, or until another directive that cancels the effect of the first one is encountered. For example,

```
'define OPCODEADD 00010
```

This defines a macro named OPCODEADD. When the text 'OPCODEADD appears in the text, then it is replaced by 00010. Verilog macros are simple text substitutions and do not permit arguments.

```
`ifdef SYNTH <Verilog code> `endif
```

If "SYNTH" is a defined macro, then the Verilog code until 'endif is inserted for the next processing phase. If "SYNTH" is not defined macro then the code is discarded.

```
`include <Verilog file>
```

The code in <Verilog file> is inserted for the next processing phase. Other standard compiler directives are listed below:

```
'resetall - resets all compiler directives to default values
'define - text-macro substitution
'timescale lns / lops - specifies time unit/precision
'ifdef, 'else, 'endif - conditional compilation
'include - file inclusion
'signed, 'unsigned - operator selection (OVI 2.0 only)
'celldefine, 'endcelldefine - library modules
'default_nettype wire - default net types
'unconnected_drive pull0|pull1,
'nounconnected_drive -pullup or down unconnected ports
'protect and 'endprotect - encryption capability
'protected and 'endprotected - encryption capability
'expand_vectornets, 'noexpand_vectornets,
'autoexpand_vectornets - vector expansion options
'remove_gatename, 'noremove_gatenames
- remove gate names for more than one instance
'remove_netname, 'noremove_netnames
- remove net names for more than one instance
```

## 4.0 System Tasks and Functions

System tasks are tool specific tasks and functions..

```
$display( "Example of using function");
/* display to screen */
$monitor($time, "a=%b, clk = %b,
add=%h",a,clk,add); // monitor signals
$setuphold( posedge clk, datain, setup, hold);
// setup and hold checks
```

A list of standard system tasks and functions are listed below:

```
$display, $write - utility to display information
$fdisplay, $fwrite - write to file
$strobe, $fstrobe - display/write simulation data
$monitor, $fmonitor - monitor, display/write information to file
$time, $realttime - current simulation time
$finish - exit the simulator
$stop - stop the simulator
$setup - setup timing check
$hold, $width- hold/width timing check
$setuphold - combines hold and setup
$readmemb/$readmemh - read stimulus patterns into memory
$sreadmemb/$sreadmemh - load data into memory
$getpattern - fast processing of stimulus patterns
$history - print command history
$save, $restart, $incsave
- saving, restarting, incremental saving
$scale - scaling timeunits from another module
$scope - descend to a particular hierarchy level
$showscopes - complete list of named blocks, tasks, modules...
$showvars - show variables at scope
```

## 5.0 Reserved Keywords

The following lists the reserved words of Verilog hardware description language, as of OVI LRM 2.0.

and	always	assign	attribute
begin	buf	bufif0	bufif1
case	cmos	deassign	default
defparam	disable	else	endattribute
end	endcase	endfunction	endprimitive
endmodule	endtable	endtask	event
for	force	forever	fork
function	highz0	highz1	if
initial	inout	input	integer
join	large	medium	module
nand	negedge	nor	not
notif0	notif1	nmos	or
output	parameter	pmos	posedge
primitive	pulldown	pullup	pull0
pull1	rcmos	reg	release
repeat	rnmos	rpmos	rtran
rtranif0	rtranif1	scalared	small
specify	specparam	strong0	strong1
supply0	supply1	table	task
tran	tranif0	tranif1	time
tri	triand	trior	triereg
tri0	tril	vectored	wait
wand	weak0	weak1	while
wire	wor		

## 6.0 Structures and Hierarchy

Hierarchical HDL structures are achieved by defining modules and instantiating modules. Nested module definitions (i.e. one module definition within another) are not permitted.

### 6.1 Module Declarations

The module name must be unique and no other module or primitive can have the same name. The port list is optional. A module without a port list or with an empty port list is typically a top level module. A macro-module is a module with a flattened hierarchy and is used by some simulators for efficiency.

module *definition example*

```
module dff (q,qb,clk,d,rst);
    input clk,d,rst ; // input signals
    output q,qb ; // output definition

    //inout for bidirectionals

    // Net type declarations
    wire dl,dbl ;

    // parameter value assignment
    paramter delay1 = 3,
        delay2 = delay1 + 1; // delay2
    // shows parameter dependance

    /* Hierarchy primitive instantiation, port
    connection in this section is by
    ordered list */

    nand #delay1 n1(cf,dl,cbf),
        n2(cbf,clk,cf,rst);
    nand #delay2 n3(dl,d,dbl,rst),
        n4(dbl,dl,clk,cbf),
        n5(q,cbf,qb),
        n6(qb,dbl,q,rst);

    /**** for debugging model initial begin
        #500 force dff_lab.rst = 1 ;
        #550 release dff_lab.rst;
        // upward path referencing
        end *****/

endmodule
```

**Overriding parameters example**

```

module dff_lab;
  reg data,rst;
  // Connecting ports by name.(map)
  dff d1 (.qb(outb), .q(out),
    .clk(clk),.d(data),.rst(rst));
  // overriding module parameters
  defparam
    dff_lab.dff.n1.delay1 = 5 ,
    dff_lab.dff.n2.delay2 = 6 ;
  // full-path referencing is used
  // over-riding by using #(8,9) delay1=8..

  dff d2 #(8,9) (outc, outd, clk, outb, rst);
  // clock generator
  always clk = #10 ~clk ;
  // stimulus ... contd

```

**Stimulus and Hierarchy example**

```

initial begin: stimuli // named block stimulus
  clk = 1; data = 1; rst = 0;
  #20 rst = 1;
  #20 data = 0;
  #600 $finish;
end

initial // hierarchy: downward path referencing
begin
  #100 force dff.n2.rst = 0 ;
  #200 release dff.n2.rst;
end
endmodule

```

**6.2 User Defined Primitive (UDP) Declarations**

The UDP's are used to augment the gate primitives and are defined by truth tables. Instances of UDP's can be used in the same way as gate primitives. There are 2 types of primitives:

1. Sequential UDP's permit initialization of output terminals, which are declared to be of `reg` type and they store values. Level-sensitive entries take precedence over edge-sensitive declarations. An input logic state `z` is interpreted as an `x`. Similarly, only `0`, `1`, `x` or `-` (unchanged) logic values are permitted on the output.

2. Combinational UDP's do not store values and cannot be initialized.

The following additional abbreviations are permitted in UDP declarations.

Logic/state Representation/transition	Abbreviation
don't care (0, 1 or X)	?
Transitions from logic x to logic y (xy). (01), (10), (0x), (1x), (x1), (x0) (?1) ..	(xy)
Transition from (01)	R or r
Transition from (10)	F or f
(01), (0X), (X1): positive transition	P or p
(10), (1x), (x0): negative transition	N or n
Any transition	* or (??)
binary don't care (0, 1)	B or b

**Combinational UDP's example**

```

// 3 to 1 mulitplexor with 2 select

primitive mux32 (Y, in1, in2, in3, s1, s2);
  input in1, in2, in3, s1, s2;
  output Y;

  table

//in1 in2 in3 s1 s2 Y
    0 ? ? 0 0 : 0 ;
    1 ? ? 0 0 : 1 ;
    ? 0 ? 1 0 : 0 ;
    ? 1 ? 1 0 : 1 ;
    ? ? 0 ? 1 : 0 ;
    ? ? 1 ? 1 : 1 ;
    0 0 ? ? 0 : 0 ;
    1 1 ? ? 0 : 1 ;
    0 ? 0 0 ? : 0 ;
    1 ? 1 0 ? : 1 ;
    ? 0 0 1 ? : 0 ;
    ? 1 1 1 ? : 1 ;

  endtable

endprimitive

```

**Sequential Level Sensitive UDP's example**

```

// latch with async reset
primitive latch (q, clock, reset, data);
  input clock, reset, data ;
  output q;
  reg q;

  initial q = 1'b1; // initialization

  table

// clock reset data q, q+
  ? 1 ? : ? : 1 ;
  0 0 0 : ? : 0 ;
  1 0 ? : ? : - ;
  0 0 1 : ? : 1 ;

  endtable
endprimitive

```

**Sequential Edge Sensitive UDP's example**

```

// edge triggered D Flip Flop with active high,
// async set and reset
primitive dff (QN, D, CP, R, S);
  output QN;
  input D, CP, R, S;
  reg QN;

  table
// D CP R S : Qtn : Qtn+1
    1 (01) 0 0 : ? : 0;
    1 (01) 0 x : ? : 0;
    ? ? 0 x : 0 : 0;
    0 (01) 0 0 : ? : 1; // clocked data
    0 (01) x 0 : ? : 1; // pessimism
    ? ? x 0 : 1 : 1; // pessimism
    1 (x1) 0 0 : 0 : 0;
    0 (x1) 0 0 : 1 : 1;
    1 (0x) 0 0 : 0 : 0;
    0 (0x) 0 0 : 1 : 1;
    ? ? 1 ? : ? : 1; // asynch clear
    ? ? 0 1 : ? : 0; // asynchronous set
    ? n 0 0 : ? : -;
    * ? ? ? : ? : -;
    ? ? (?0) ? : ? : -;
    ? ? ? (?0): ? : -;
    ? ? ? ? : ? : x;

  endtable
endprimitive

```

## 7.0 Expressions and Operators

Arithmetic and logical operators are used to build expressions. Expressions perform operation on one or more operands, the operands being vectored or scalar nets, registers, bit-selects, part selects, function calls or concatenations thereof.

- Unary Expression  
<operator> <operand>

```
a = !b;
```

- Binary and Other Expressions  
<operand> <operator> <operand>

```
if (a < b) // if (<expression>)
{c,d} = a + b ;
// concatenate and add operator
```

- Parentheses can be used to change the precedence of operators. For example, ((a+b) \* c)

### Operator precedence

Operator	Precedence
+, -, !, ~ (unary)	Highest
*, / %	
+, - (binary)	
<<, >>	
<, <=, >, >=	
=, ==, !=	
==, !=	
&, ~&	
^, ^~	
, ~	
&&	
?:	Lowest

- All operators associate left to right, except for the ternary operator “?:” which associates from right to left.

### Relational Operators

Operator	Application
<	a < b // is a less than b? // return 1-bit true/false
>	a > b // is a greater than b?
>=	a >= b // is a greater than or // equal to b
<=	a <= b // is a less than or // equal to b

### Arithmetic Operators

Operator	Application
*	c = a * b ; // multiply a with b
/	c = a / b ; // int divide a by b
+	sum = a + b ; // add a and b
-	diff = a - b ; // subtract b // from a
%	amodb = a % b ; // a mod(b)

### Logical Operators

Operator	Application
&&	a && b ; // is a and b true? // returns 1-bit true/false
	a    b ; // is a or b true? // returns 1-bit true/false
!	if (!a) ; // if a is not true c = b ; // assign b to c

### Equality and Identity Operators

Operator	Application
=	c = a ; // assign a to c
==	c == a ; /* is c equal to a returns 1-bit true/false applies for 1 or 0, logic equality, using X or Z oper- ands returns always false 'hx == 'h5 returns 0 */
!=	c != a ; // is c not equal to // a, retruns 1-bit true/ // false logic equality
===	a === b ; // is a identical to // b (includes 0, 1, x, z) / // 'hx === 'h5 returns 0
!==	a !== b ; // is a not // identical to b returns 1- // bit true/false

### Unary, Bitwise and Reduction Operators

Operator	Application
+	Unary plus & arithmetic(binary) addition
-	Unary negation & arithmetic (binary) subtraction
&	b = &a ; // AND all bits of a
	b =  a ; // OR all bits
^	b = ^a ; // Exclusive or all bits of a
~&, ~ , ~^	NAND, NOR, EX-NOR all bits to-gether c = ~& b ; d = ~  a ; e = ^c ;
~, &,  , ^	bit-wise NOT, AND, OR, EX-OR b = ~a ; // invert a c = b & a ; // bitwise AND a,b e = b   a ; // bitwise OR f = b ^ a ; // bitwise EX-OR
~&, ~ , ~^	bit-wise NAND, NOR, EX-NOR c = a ~& b ; d = a ~  b ; e = a ~^ b ;



**Shift Operators and other Operators**

Operator	Application
<<	a << 1 ; // shift left a by // 1-bit
>>	a >> 1 ; // shift right a by 1
?:	c = sel ? a : b ; /* if sel is true c = a, else c = b , ?: ternary operator */
{}	{co, sum } = a + b + ci ; /* add a, b, ci assign the overflow to co and the re- sult to sum: operator is called concatenation */
{{}}	b = {3{a}} /* replicate a 3 times, equivalent to {a, a, a} */

**7.1 Parallel Expressions**

fork ... join are used for concurrent expression assignments.

fork ... join *example*

```

initial
begin: block
fork
    // This waits for the first event a
    // or b to occur
    @a disable block ;
    @b disable block ;

    // reset at absolute time 20
    #20 reset = 1 ;
    // data at absolute time 100
    #100 data = 0 ;
    // data at absolute time 120
    #120 data = 1 ;

join
end

```

**7.2 Conditional Statements**

The most commonly used conditional statement is the if, if ... else ... conditions. The statement occurs if the expressions controlling the if statement evaluates to true.

**if .. else ...conditions example**

```

always @(rst)// simple if -else
if (rst)
    // procedural assignment
    q = 0;
else // remove the above continous assign
deassign q;

always @(WRITE or READ or STATUS)
begin
    // if - else - if
    if (!WRITE) begin
        out = oldvalue ;
    end
    else if (!STATUS) begin
        q = newstatus ;
        STATUS = hold ;
    end
    else if (!READ) begin
        out = newvalue ;
    end
end
end

```

case, casex, casez: case statements are used for switching between multiple selections (if (case1) ... else if (case2) ... else ...). If there are multiple matches only the first is evaluated. casez treats high impedance values as don't care's and casex treats both unknown and high-impedance as don't care's.

**case statement example**

```

module d2X8 (select, out); // priority encode
input [0:2] select;
output [0:7] out;
reg [0:7] out;
always @(select) begin
    out = 0;
    case (select)
        0: out[0] = 1;
        1: out[1] = 1;
        2: out[2] = 1;
        3: out[3] = 1;
        4: out[4] = 1;
        5: out[5] = 1;
        6: out[6] = 1;
        7: out[7] = 1;
    endcase
end
endmodule

```

**casex statement example**

```

casex (state)
    // treats both x and z as don't care
    // during comparison : 3'b01z, 3'b01x, 3'b'011
    // ... match case 3'b01x
    3'b01x: fsm = 0 ;
    3'b0xx: fsm = 1 ;
    default: begin
        // default matches all other occurances
        fsm = 1 ;
        next_state = 3'b011 ;
    end
endcase

```

**casez statement example**

```

casez (state)
    // treats z as don't care during comparison :
    // 3'b11z, 3'b1zz, ... match 3'b1?: fsm = 0 ;
    3'b1?: fsm = 0 ; // if MSB is 1, matches 3?b1?
    3'b01?: fsm = 1 ;
    default: $display("wrong state") ;
endcase

```

**7.3 Looping Statements**

forever, for, while and repeat *loops example*

```

forever
    // should be used with disable or timing control
    @(posedge clock) {co, sum} = a + b + ci ;

for (i = 0 ; i < 7 ; i=i+1)
    memory[i] = 0 ; // initialize to 0

for (i = 0 ; i <= bit-width ; i=i+1)
    // multiplier using shift left and add
    if (a[i]) out = out + ( b << (i-1) ) ;

repeat(bit-width) begin
    if (a[0]) out = b + out ;
    b = b << 1 ; // muliplier using
    a = a << 1 ; // shift left and add
end

while(delay) begin @(posedge clk) ;
    ldlang = oldldlang ;
    delay = delay - 1 ;
end
end

```

## 8.0 Named Blocks, Disabling Blocks

Named blocks are used to create hierarchy within modules and can be used to group a collection of assignments or expressions. `disable` statement is used to disable or de-activate any named block, tasks or modules. Named blocks, tasks can be accessed by full or reference hierarchy paths (example `dff_lab.stimuli`). Named blocks can have local variables.

### Named blocks and disable statement example

```
initial forever @(posedge reset)
  disable MAIN ; // disable named block
  // tasks, modules can also be disabled

always begin: MAIN // defining named blocks
  if (!qfull) begin
    #30 recv(new, newdata) ; // call task
    if (new) begin
      q[head] = newdata ;
      head = head + 1 ; // queue
    end
  end
  end
else
  disable recv ;
end // MAIN
```

## 9.0 Tasks and Functions

Tasks and functions permit the grouping of common procedures and then executing these procedures from different places. Arguments are passed in the form of input/inout values and all calls to functions and tasks share variables. The differences between tasks and functions are

Tasks	Functions
Permits time control	Executes in one simulation time
Can have zero or more arguments	Require at least one input
Does not return value, assigns value to outputs	Returns a single value, no special output declarations required
Can have output arguments, permits #, @, ->, wait, task calls.	Does not permit outputs, #, @, ->, wait, task calls

### task Example

```
// task are declared within modules
task recv ;
  output valid ;
  output [9:0] data ;
  begin
    valid = inreg ;
    if (valid) begin
      ackin = 1 ;
      data = qin ;
      wait(inreg) ;
      ackin = 0 ;
    end
  end
end

// task instantiation
always begin: MAIN //named definition
  if (!qfull) begin
    recv(new, newdata) ; // call task
    if (new) begin
      q[head] = newdata ;
      head = head + 1 ;
    end
  end
end else
  disable recv ;
end // MAIN
```

### function Example

```
module foo2 (cs, in1, in2, ns);
  input [1:0] cs;
  input in1, in2;
  output [1:0] ns;
  function [1:0] generate_next_state;
    input [1:0] current_state ;
    input input1, input2 ;
    reg [1:0] next_state ;
    // input1 causes 0->1 transition
    // input2 causes 1->2 transition
    // 2->0 illegal and unknown states go to 0
    begin
      case (current_state)
        2'h0 : next_state = input1 ? 2'h1 : 2'h0 ;
        2'h1 : next_state = input2 ? 2'h2 : 2'h1 ;
        2'h2 : next_state = 2'h0 ;
        default: next_state = 2'h0 ;
      endcase
      generate_next_state = next_state;
    end
  endfunction // generate_next_state

  assign ns = generate_next_state(cs, in1,in2) ;
endmodule
```

## 10.0 Continous Assignments

Continous assignments imply that whenever any change on the RHS of the assignment occurs, it is evaluated and assigned to the LHS. These assignments thus drive both vector and scalar values onto nets. Continous assignments always implement combinational logic (possibly with delays). The driving strengths of a continous assignment can be specified by the user on the net types.

- Continous assignment on declaration

```
/* since only one net15 declaration exists in a
   given module only one such declarative continous
   assignment per signal is allowed */
```

```
wire #10 (atrong1, pull0) net15 = enable ;
/* delay of 10 for continous assignment with
   strengths of logic 1 as strong1 and logic 0 as
   pull0 */
```

- Continous assignment on already declared nets

```
assign #10 net15 = enable ;
assign (weak1, strong0) {s,c} = a + b ;
```

## 11.0 Procedural Assignments

Assignments to register data types may occur within `always`, `initial`, `task` and `functions`. These expressions are controlled by triggers which cause the assignments to evaluate. The variables to which the expressions are assigned must be made of bit-select or part-select or whole element of a reg, integer, real or time. These triggers can be controlled by loops, `if`, `else`... constructs. `assign` and `deassign` are used for procedural assignments and to remove the continous assignments.

```
module dff (q,qb,clk,d,rst);
  output q, qb;
  input d, rst, clk;
  reg q, qb, temp;
  always
    #1 qb = ~q ; // procedural assignment

  always @(rst)
    // procedural assignment with triggers
    if (rst) assign q = temp;
    else deassign q;

  always @(posedge clk)
    temp = d;
endmodule
```

force and release are also procedural assignments. However, they can force or release values on net data types and registers.

## 11.1 Blocking Assignment

```
module adder (a, b, ci, co, sum, clk) ;
  input a, b, ci, clk ;
  output co, sum ;
  reg co, sum;
  always @(posedge clk) // edge control
    // assign co, sum with previous value of a,b,ci
    {co,sum} = #10 a + b + ci ;
endmodule
```

## 11.2 Non-Blocking Assignment

Allows scheduling of assignments without blocking the procedural flow. Blocking assignments allow timing control which are delays, whereas, non-blocking assignments permit timing control which can be delays or event control. The non-blocking assignment is used to avoid race conditions and can model RTL assignments.

```
/* assume a = 10, b= 20 c = 30 d = 40 at start of
block */

always @(posedge clk)
  begin:block
    a <= #10 b ;
    b <= #10 c ;
    c <= #10 d ;
  end

/* at end of block + 10 time units, a = 20, b = 30,
c = 40 */
```

## 12.0 Gate Types, MOS and Bidirectional Switches

Gate declarations permit the user to instantiate different gate-types and assign drive-strengths to the logic values and also any delays

```
<gate-declaration> ::= <component>
  <drive_strength>? <delay>? <gate_instance>
  <,?<gate_instance...> ;
```

Gate Types		Component
Gates	Allows strengths	and, nand, or, nor,xor, xnor buf, not
Three State Drivers	Allows strengths	buif0,buif1 notif0,notif1
MOS Switches	No strengths	nmos,pmos,cmos, rmos,rpmos,rcmos
Bi-directional switches	No strengths, non resistive	tran, tranif0, tranif1
	No strengths, resistive	rtran,rtranif0, rtranif1
	Allows strengths	pullup pulldown

### Gates, switch types, and their instantiations

```
cmos i1 (out, datain, ncontrol, pcontrol);
nmos i2 (out, datain, ncontrol);
pmos i3 (out, datain, pcontrol);
pullup (neta) (netb);
pulldown (netc);
nor i4 (out, in1, in2, ...);
and i5 (out, in1, in2, ...);
nand i6 (out, in1, in2, ...);
buf i7 (out1, out2, in);
bufif1 i8 (out, in, control);
tranif1 i9 (inout1, inout2, control);
```

### Gate level instantiation example

```
// Gate level instantiations
nor (highz1, strong0) #(2:3:5) (out, in1,
  in2);
// instantiates a nor gate with out
// strength of highz1 (for 1) and
// strong0 for 0 #(2:3:5) is the
// min:typ:max delay

pullup1 (strong1) net1;
// instantiates a logic high pullup
cmos (out, data, ncontrol, pcontrol);
// MOS devices
```

The following strength definitions exists

- 4 drive strengths (supply, strong, pull, weak)
- 3 capacitor strengths (large, medium, small)
- 1 high impedance state highz

The drive strengths for each of the output signals are

- Strength of an output signal with logic value 1  
supply1, strong1, pull1, large1, weak1, highz1
- Strength of an output signal with logic value 0  
supply0, strong0, pull0, large0, weak0, highz0

Logic 0		Logic 1		Strength
supply0	Su0	supply1	Su1	7
strong0	St0	strong1	St1	6
pull0	Pu0	pull1	Pu1	5
large	La0	large	La1	4
weak0	We0	weak1	We1	3
medium	Me0	medium	Me1	2
small	Sm0	small	Sm1	1
highz0	HiZ0	highz1	HiZ0	0

## 12.1 Gate Delays

The delays allow the modeling of rise time, fall time and turn-off delays for the gates. Each of these delay types may be in the min:typ:max format. The order of the delays are #(trise, tfall, tturn-off). For example,

```
nand #(6:7:8, 5:6:7, 122:16:19)
  (out, a, b);
```

Delay	Model
#(delay)	min:typ:max delay
#(delay, delay)	rise-time delay, fall-time delay, each delay can be with min:typ:max
#(delay, delay, delay)	rise-time delay, fall-time delay and turn-off delay, each min:t- yp:max

For `trireg`, the decay of the capacitive network is modeled using the rise-time delay, fall-time delay and charge-decay. For example,

```
trireg (large) #(0,1,9) capacitor
// charge strength is large
// decay with tr=0, tf=1, tdecay=9
```

### 13.0 Specify Blocks

A specify block is used to specify timing information for the module in which the specify block is used. Specparams are used to declare delay constants, much like regular parameters inside a module, but unlike module parameters they cannot be overridden. Paths are used to declare time delays between inputs and outputs.

#### Timing Information using specify blocks

```
specify // similar to defparam, used for timing
specparam delay1 = 25.0, delay2 = 24.0;

// edge sensitive delays -- some simulators
// do not support this
(posedge clock) => (out1 +: in1) =
    (delay1, delay2) ;
// conditional delays
if (OPCODE == 3'h4) (in1, in2 *> out1)
    = (delay1, delay2) ;
// +: implies edge-sensitive +ve polarity
// -: implies edge sensitive -ve polarity
// *> implies multiple paths

// level sensitive delays
if (clock) (in1, in2 *> out1, out2) = 30 ;
// setuphold
$setuphold(posedge clock &&& reset,
    in1 &&& reset, 3:5:6, 2:3:6);
(reset *> out1, out2) = (2:3:5,3:4:5);

endspecify
```

## Verilog

### Synthesis Constructs

The following is a set of Verilog constructs that are supported by most synthesis tools at the time of this writing. To prevent variations in supported synthesis constructs from tool to tool, this is the least common denominator of supported constructs. Tool reference guides cover specific constructs.

#### 14.0 Verilog Synthesis Constructs

Since it is very difficult for the synthesis tool to find hardware with exact delays, all absolute and relative time declarations are ignored by the tools. Also, all signals are assumed to be of maximum strength (strength 7). Boolean operations on `X` and `Z` are not permitted. The constructs are classified as

- Fully supported constructs — Constructs that are supported as defined in the Verilog Language Reference Manual
- Partially supported — Constructs supported with restrictions on them
- Ignored constructs — Constructs that are ignored by the synthesis tool
- Unsupported constructs — Constructs which if used, may cause the synthesis tool to not accept the Verilog input or may cause different results between synthesis and simulation.

#### 14.1 Fully Supported Constructs

```
<module instantiation,
    with named and positional notations>
<integer data types, with all bases>
<identifiers>
<subranges and slices on right-hand
    side of assignment>
<continuous assignments>
>>, <<, ? : {}
assign (procedural and declarative), begin, end
case, casex, casez, endcase
default
```

```
disable
function, endfunction
if, else, else if
input, output, inout
wire, wand, wor, tri
integer, reg
macromodule, module
parameter
supply0, supply1
task, endtask
```

#### 14.2 Partially Supported Constructs

Construct	Constraints
<code>*</code> , <code>/</code> , <code>%</code>	when both operands constants, or 2nd operand power of 2.
<code>always</code>	only edge-triggered events.
<code>for</code>	bounded by static variables: only use “+” or “-” to index.
<code>posedge</code> , <code>negedge</code>	only with <code>always @ .</code>
<code>primitive</code> , <code>endprimitive</code> <code>table</code> , <code>endtable</code>	Combinational and edge-sensitive user defined primitives are often supported.
<code>&lt;=</code>	limitations on usage with blocking assignment.
<code>and</code> , <code>nand</code> , <code>or</code> , <code>nor</code> , <code>xor</code> , <code>xnor</code> , <code>buf</code> , <code>not</code> , <code>bufif0</code> , <code>bufif1</code> , <code>notif0</code> , <code>notif1</code>	gate types supported without <code>X</code> or <code>Z</code> constructs
<code>!</code> , <code>&amp;&amp;</code> , <code>  </code> , <code>~</code> , <code>&amp;</code> , <code> </code> , <code>^</code> , <code>^~</code> , <code>~^</code> , <code>~&amp;</code> , <code>~ </code> , <code>+</code> , <code>-</code> , <code>&lt;</code> , <code>&gt;</code> , <code>&lt;=</code> , <code>&gt;=</code> , <code>==</code> , <code>!=</code>	operators supported without <code>X</code> or <code>Z</code> constructs

### 14.3 Ignored Constructs

```
<intra-assignment timing controls>
<delay specifications>
scaled, vectored
small, large, medium
specify
time (some tools treat these as integers)
weak1, weak0, highz0, highz1, pull0, pull1
$keyword (some tools use these to set
          synthesis constraints)
wait (some tools support wait with a
      bounded condition)
```

- NOTES -

- NOTES -

### 14.4 Unsupported Constructs

```
<assignment with variable used as bit select
      on LHS of assignment>
<global variables>
==, !=
cmos, nmos, rcmos, rnmos, pmos, rpmos
deassign
defparam
event
force
fork, join
forever, while
initial
pullup, pulldown
release
repeat
rtran, tran, tranif0, tranif1, rtranif0,
      rtranif1
table, endtable, primitive, endprimitive
```

All rights reserved. Please send any feedback to the author.  
Verilog® is a registered trademark of Cadence Design Systems, Inc.