หนังสือเล่มนี้ แต่งโดย นอ. ชาติชาย ดิษฐกุล รน.

ภาควิชาวิศวกรรมคอมพิวเตอร์ได้รับอนุญาตให้ทดลองใช้หนังสือเล่มนี้ในการเรียนการสอนในวิชาที่มีการใช้ภาษา VHDL

บทที่ 1
บทนำ (Introduction)

1.1 กล่าวนำ

จุดประสงค์ของหนังสือเล่มนี้เพื่อแนะนำพื้นฐานของภาษา VHDL อันได้แก่การเขียนรูปแบบ (modeling) และการตรวจสอบการทำงานของรูปแบบดังกล่าว โดยการจำลองการทำงาน (model simulation) อย่างไรก็ตามไม่สามารถที่จะหลีกเลี่ยงการศึกษาถึงกฎเกณฑ์ (syntax) ของภาษา ตลอดจนความหมายต่างๆ ของภาษา (semantics) ได้ แต่จะกล่าวถึง เท่าที่จำเป็น สำหรับความเข้าใจเบื้องต้นเท่านั้น และพอเพียงที่จะสามารถนำไปใช้เขียนรูปแบบระบบดิจิตอลพื้นฐานได้ ส่วนในเรื่องของรายละเอียดนั้น สามารถศึกษาได้จากหนังสือคู่มือ Language Reference Manual (LRM) ของ IEEE (Institute of Electrical and Electronics Engineers) ฉนั้นในบทนี้จะบรรยายถึงประวัติความเป็นมา และวิวัฒนาการของภาษา VHDL อธิบายคำจำกัดความ (definition) ตลอดจนคำต่างๆ และวิธีการเขียนที่ใช้ในหนังสือเล่มนี้

1.2 ประวัติความเป็นมาของภาษา VHDL

วิวัฒนาการของภาษา VHDL นั้นเริ่มต้นประมาณปี ค.ศ. 1981 โดยที่กระทรวงกลาโหมสหรัฐอเมริกาหรือ Department of Defense (DOD) มองเห็นว่าอุปกรณ์อิเล็กทรอนิกส์และคอมพิวเตอร์ที่ใช้ในกิจการทางทหาร เป็นอุปกรณ์ที่ได้รับการพัฒนามาเมื่อประมาณ 20 ปีก่อน เพราะเทคโนโลยีในขณะนั้นทำให้การพัฒนาอุปกรณ์อิเล็กทรอนิกส์เป็นไปอย่างล่าช้า ซึ่งเป็นสภาพที่ไม่อาจยอมรับได้ในปัจจุบัน เพราะเทคโนโลยีทางด้านไมโครอิเล็กทรอนิกส์ ได้รับการพัฒนาไปอย่างรวดเร็ว ดังที่จะเห็นได้ว่ามีวงจรดิจิตอลอิเล็กทรอนิกส์หลายวงจร ที่แต่เดิมถูกสร้างขึ้นมาจากชิ้นส่วนอุปกรณ์อิเล็กทรอนิกส์จำนวนหลายชิ้น ถูกนำประกอบกันอยู่บนแผงวงจรไฟฟ้า (Printed Circuit Board หรือ PCB) ที่มีขนาดใหญ่ แต่ในปัจจุบันสามารถที่จะใช้เทคโนโลยีการออกแบบและผลิตวงจรรวมขนาดใหญ่มาก (Very Large Scale Integration หรือ VLSI) รวมอุปกรณ์

ต่างๆ เหล่านั้นให้อยู่บนชิ้นอุปกรณ์สารกึ่งตัวนำ ที่มีขนาดประมาณ 1-2 ตร.ซม. ได้ ซึ่งเป็นผลให้ประสิทธิภาพในการทำงานของวงจรสูงขึ้น (ความเร็วในการทำงานของวงจร) ตลอดจนความน่าเชื่อถือ (reliability) และความคงทนต่อสภาพแวดล้อมสูง ขณะเดียวกันนั้นในวงการทหารได้มีการนำระบบคอมพิวเตอร์และอิเล็กทรอนิกส์ มาใช้ในระบบอาวุธอย่างแพร่หลาย โดยเฉพาะอย่างยิ่งในระบบอาวุธ ดังนั้นอุปกรณ์ที่มีใช้อยู่จึงไม่เหมาะสมกับเทคโนโลยีด้านอาวุธของประเทศคู่แข่งขัน การที่จะเปลี่ยนอุปกรณ์ใหม่เป็นสิ่งที่ต้องใช้งบประมาณมาก และก็จะประสบกับปัญหาเช่นเดิมคืออุปกรณ์ใหม่ก็ได้รับการพัฒนามานานแล้วเช่นกัน เพราะในขณะนั้นขั้นตอนของการออกแบบ ผลิต และตรวจสอบวงจรต้นแบบ เป็นขบวนการที่ต้องใช้วิศวกร และเวลาสำหรับดำเนินการมาก ฉนั้นทาง DOD จึงตั้งโครงการขึ้นมาเพื่อศึกษา วิธีการที่จะช่วยพัฒนาวงจรอิเล็กทรอนิกส์ โดยเฉพาะอย่างยิ่งวงจรระบบดิจิตอล ให้สามารถนำไปผลิตได้เร็วขึ้น และโครงการดังกล่าวมีชื่อว่า Very High Speed Integrated Circuits หรือ VHSIC ในระยะแรกนั้นโครงการเป็นความลับทางด้านความมั่งคงของประเทศ และอยู่ในความดูแลควบคุมของ United States International Traffic and Arms Regulations (ITAR) ในปี ค.ศ. 1983 ตามคำแนะนำของคณะทำงาน ("Woods Hole" workshop) ทาง DOD ได้ออกความต้องการมาตรฐานของภาษาที่ใช้สำหรับบรรยายพฤติกรรมของวงจรหรือ Hardware ของระบบสำหรับโครงการ VHSIC ซึ่งมีสาระสำคัญพอสรุปได้ดังนี้

- ต้องเป็นภาษาที่นำไปเขียนรูปแบบระบบดิจิตอล และมีคุณสมบัติที่สามารถจะเข้าใจได้ทั้งคนและเครื่องโดยไม่ต้องมีการแปลหรือเปลี่ยนแปลงอีก
- สามารถนำไปใช้เป็นเอกสารประกอบโครงการได้ (Project Documentation)
- ต้องเป็นภาษาที่เขียนขึ้นสำหรับใช้จำลองการทำงานของวงจร (Simulation Language)

ฉนั้นภาษาดังกล่าวนี้จึงจัดเป็นภาษาโปรแกรมระดับสูง (High Level Language) เช่นเดียวกับภาษา PASCAL, FORTRAN และ ADA ซึ่งในทางวิศวกรรมการออกแบบ hardware เรียกว่า Hardware Description Language หรือ HDL ดังนั้นภาษามาตรฐานนี้จึงมีชื่อว่า **VHSIC-HDL** หรือ **VHDL** นั่นเอง

เริ่มต้นโครงการ DOD ได้มอบหมายให้บริษัท IBM, Texas Instruments และ Intermetrics เป็นผู้ศึกษาและพัฒนา การดำเนินการได้กระทำไปอย่างต่อเนื่อง และได้ผลเป็นที่น่าพอใจ จนกระทั่งปี ค.ศ. 1985 ทาง ITAR ได้ยกเลิกข้อจำกัดในการถ่ายทอดเทคโนโลยีทางทหาร ออกจากโครงการนี้ ดังนั้น VHDL จึงเริ่มเป็นที่รู้จักกันโดยทั่วไป จนกระทั่งทาง IEEE จึงได้รับภาษานี้เข้ามาศึกษาและประมาณปี ค.ศ. 1987 ได้ยอมรับกำหนดมาตรฐานของภาษา โดยให้ชื่อว่า IEEE 1076-1987 และมีชื่อเรียกว่า VHDL มาตรฐานนี้ก็ได้รับการปรับปรุงจนปัจจุบัน (ช่วงเวลาที่เขียนหนังสือ) ได้ชื่อว่า IEEE 1076-1993 หรือ VHDL 1993

การที่ทาง DOD ในขณะนั้น เป็นลูกค้ารายใหญ่ของอุตสาหกรรมอิเล็กทรอนิกส์และคอมพิวเตอร์ จึงมีผู้รับโครงการต่างๆ จาก DOD ไปดำเนินการด้านวิจัยและพัฒนามาก เพื่อที่จะให้เป็นมาตราฐานเดียวกันหมด ทาง DOD จึงกำหนดว่า ในการส่งโครงการนั้นจะต้องเขียนอยู่ในรูปของภาษา VHDL เท่านั้น ซึ่งทำให้เกิดข้อดีต่อ DOD เองที่เป็นมาตรฐานเดียวกัน สามารถนำไปจำลองกับเครื่องคอมพิวเตอร์ได้หลายๆ ระบบ

1.3 Top-Down Design

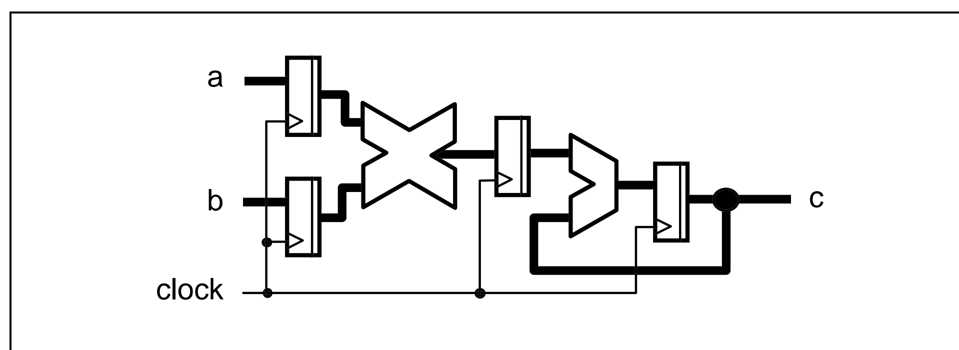
สิ่งที่ท้าทายอันเนื่องมาจากการนำภาษา VHDL มาใช้กำหนดและบรรยายพฤติกรรมฟังก์ชันการทำงานของ hardware ในระบบดิจิตอลนั้น คือความอ่อนตัวของภาษาที่สามารถจำลองการทำงานจากหลักการของรูปแบบ (simulate conceptual designs) แต่ในขณะเดียวกันก็สามารถจำลองการทำงานของ hardware ที่ให้รายละเอียดเกี่ยวกับเวลาอย่างถูกต้อง (timing based simulation) และจากโครงสร้างของภาษายังสามารถจำลองการทำงานในรูปของลำดับชั้น (hierarchy of simulation levels) ความสามารถดังกล่าวนี้จึงช่วยให้วิศวกรออกแบบ สามารถที่จะเขียนรูปแบบบรรยาย จากระดับบนสุดของวงจรที่อยู่ในรูปสังเขป (high level of abstraction) ลงสู่รายละเอียดในระดับล่างของวงจรได้เช่น gate level เป็นต้น ในช่วงเวลานั้นเองวงการอุตสาหกรรมไมโครอิเล็กทรอนิกส์ ตลอดจนสถาบันวิจัยและศึกษา กำลังพัฒนาภาษาที่จะใช้สำหรับการสังเคราะห์วงจรแบบอัตโนมัติ เพื่อลดเวลาในการพัฒนาวงจรลง ภาษา VHDL จึงถูกนำเข้าพิจารณาในโครงการนี้ด้วย โดยเพิ่มขีดความสามารถของภาษาขึ้นอีกประการหนึ่ง นอกเหนือจากสิ่งที่ทาง DOD กำหนดในครั้งแรกคือเป็นภาษาที่ใช้สำหรับสังเคราะห์วงจร (synthesis language)

ภาษา VHDL เป็นภาษาที่สนับสนุนการเขียนรูปแบบในทุกๆ ลักษณะและวิธีการ ดังเช่น ตัวอย่างที่แสดงในรูปที่ 1.1 คือรูปแบบ (model) ของลำดับขั้นตอนของการคูณและหาผลรวม (multiply accumulate algorithm) ของตัวแปรสองตัว ***a*** และ ***b*** ส่วนผลลัพธ์คือ ***c*** ในลักษณะของเลขฐานสอง (binary number) รูปแบบนี้แสดงในระดับที่สังเขปที่สุดของแนวความคิดที่จะแก้ปัญหา เพื่อหาผลลัพธ์ โดยไม่ได้คำนึงถึงโครงสร้างของวงจรอย่างที่เคยชิน เช่นอุปกรณ์วงจรรวม Arithmetic and Logic Unit (ALU)

```
FOR i IN 1 TO 1024 LOOP
          result := result + a(i) * b(i);
END LOOP;
c <= result;
```

รูปที่ 1.1: VHDL Statement **สำหรับ** Multiply Accumulate Algorithm

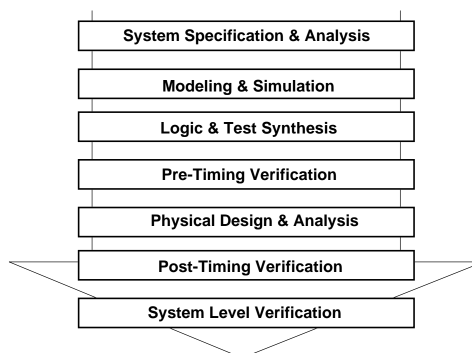
ในขณะที่อีกด้านหนึ่งของมุมมองในปัญหาเดียวกันนี้ ภาษา VHDL ก็สามารถใช้บรรยายลำดับขั้นตอนของการคูณและหาผลรวม (multiply accumulate algorithm) โดยแสดงได้ในรายละเอียดของการสร้างวงจรดังกล่าวจริงๆ ตามที่เห็นได้จากรูปที่ 1.2



รูปที่ 1.2: **โครงสร้างของ** Multiply-Accumulate Unit

ฉนั้นจากความสามารถที่จะเขียนรูปแบบ (modeling) ได้ในลักษณะต่างๆ นี้เอง จึงเปิดโอกาสให้วิศวกรผู้ออกแบบได้พัฒนาและจำลองการทำงานของรูปแบบได้เร็ว ตั้งแต่ในระยะเริ่มต้นของแนวความคิดเกี่ยวกับฟังก์ชันการทำงานของวงจรอย่างสังเขป โดยที่ยังไม่ต้องไปคำนึงถึงรายละเอียดเกี่ยวกับโครงสร้างวงจรจริง นอกจากนั้น VHDL ยังเป็นภาษาที่สนับสนุนลักษณะต่างๆ ของระบบดิจิตอล (digital system) ที่มีความซับซ้อนได้ทั้งหมด

การเริ่มต้นด้วยวิธีการเขียนรูปแบบจากแนวความคิดอย่างสังเขป พร้อมทั้งการจำลองการทำงานของรูปแบบที่เขียนขึ้น เพื่อตรวจสอบความถูกต้อง ประกอบกับการกลั่นกรองเพิ่มเติมรายละเอียดลงสู่ระบบดิจิตอลที่สมบูรณ์ในรูปของวงจรไฟฟ้าที่ละขั้น นั้นเป็นขบวนการของ ***Top-Down Design*** การที่เริ่มต้นด้วยการเขียนรูปแบบ (modeling) ในระดับบน (top-level) ของแนวความคิดอย่างสังเขปนั้น วิศวกรออกแบบสามารถที่จะพัฒนาสภาพแวดล้อมต่างๆ เพื่อการตรวจสอบการทำงานของวงจร (test environment) ได้ตั้งแต่ในระยะแรกๆ ของการออกแบบ เพื่อใช้สำหรับตรวจสอบความถูกต้องของแนวความคิดกับสิ่งที่ต้องการจริงหรือ specification ของงาน ดังนั้นจึงเป็นไปได้ยากที่ในระดับล่างลงมา โดยที่ได้เพิ่มรายละเอียดของรูปแบบให้มากขึ้นตามลำดับจะเกิดข้อผิดพลาด หรือเบี่ยงเบนไปจากจุดประสงค์เดิม เพราะในแต่ละขั้นตอนย่อยจะมีการสร้างสภาพแวดล้อมเพื่อการตรวจสอบขึ้นใหม่ โดยอ้างอิงจากระดับที่อยู่สูงกว่าขึ้นไปเสมอ จากรูปที่ 1.3 แสดงให้เห็นขั้นตอนของการออกแบบในลักษณะของ Top-Down Design ทั้งนี้ในทางปฏิบัติอาจจะมีข้อแตกต่างไปจากนี้บ้างเล็กน้อย ก็เนื่องมาจากขั้นตอนของการผลิต (implementation) สามารถกระทำได้ในหลายๆ เทคโนโลยี เช่น Programmable Logic Devices อันได้แก่ PLA, FPGA หรือ CPLD เป็นต้น นอกจากนั้นยังมี Semi-Custom IC (Gate Array, Standard Cell) และ Full Custom IC



รูปที่ 1.3: **ขั้นตอนของ** Top-Down Design

ขั้นตอนของขบวนการออกแบบโดยใช้วิธี Top-Down Design มีรายละเอียดดังนี้

- ***System Specification and Analysis*** ขั้นตอนของการสร้างข้อกำหนดของความต้องการ (specification) และวิเคราะห์ระบบ เพื่อหาแนวความคิดและหลักการ (Idea and Concept) ในการแก้ปัญหา
- ***Modeling and Simulation*** การเขียนรูปแบบของระบบที่ต้องการออกแบบโดยภาษา VHDL หรือ ภาษา HDL อื่นๆ จากแนวความคิดอย่างสังเขปที่ได้ สำหรับบรรยายพฤติกรรมการทำงาน พร้อมทั้งจำลองการทำงาน เพื่อเปรียบเทียบและตรวจสอบความถูกต้องกับข้อกำหนด (specification)
- ***Logic and Test Synthesis*** หลังจากที่ได้หลักการขั้นต้นพร้อมกับแนวความคิดที่ผ่านการตรวจสอบแล้ว หลักการนี้จะถูกเพิ่มเติมในรายละเอียดลงมาเป็นลำดับขั้นตอนที่เหมือนกัน คือ modeling and simulation จนกระทั่งอยู่ในระดับที่จะนำไปผลิตวงจรหรือสังเคราะห์ (synthesis) ในขั้นตอนนี้เองเทคโนโลยีที่จะมารองรับวงจรออกแบบจะถูกกำหนดขึ้น และระบบช่วยการออกแบบจะสังเคราะห์วงจรที่ได้จากรูปแบบที่เขียนขึ้น ให้อยู่ในรูปของวงจรที่ประกอบด้วยอุปกรณ์อิเล็กทรอนิกส์ (gate-level) และการเชื่อมต่อระหว่างกันของอุปกรณ์เหล่านั้น หรือไม่ก็อยู่ในรูปของ netlist ที่สามารถนำไปผลิตลงบนอุปกรณ์อื่นได้ นอกจากนั้นการผลิตบางเทคโนโลยี อาทิเช่น Gate Array หรือ Standard Cell และ Full Custom IC อาจจะมีความจำเป็นที่ต้องสร้างโครงสร้างของวงจรใหม่หลังจากที่สังเคราะห์ครั้งแรกแล้ว เพื่อความสะดวกต่อการตรวจสอบการทำงานหลังจากที่ผลิตเป็นวงจรต้นแบบแล้ว หรือที่เรียกว่า "**design for test**" (DFT) พร้อมทั้งข้อมูลในการตรวจสอบ (test pattern) จะถูกกำหนดในขั้นตอนนี้
- ***Pre-Timing Verification*** หลังจากการสังเคราะห์วงจรให้อยู่ในรูป gate-level หรือ netlist แล้ว ข้อมูลที่ได้จากผู้ผลิตอุปกรณ์วงจรนั้น นอกจากจะเป็นข้อมูลสำหรับจำลองการทำงาน ในเรื่องของความถูกต้องของฟังก์ชัน (functional simulation) แล้ว ยังมีข้อมูลที่เกี่ยวกับเวลาด้วย ซึ่งเป็นความจริงที่ว่า อุปกรณ์ทางอิเล็กทรอนิกส์ทุกชิ้นจะมี propagation delay เสมอ ถึงแม้ว่าจะเป็นเวลาที่น้อยมากในระดับ nanosecond (10^{-9} second) แต่ถ้าภายในวงจรหนึ่งประกอบด้วย gate ของฟังก์ชันต่างๆ จำนวน 10,000 gate ขึ้นไป เวลาดังกล่าวนี้จะสะสมกันมากขึ้น จนอาจจะทำให้การทำงานของวงจรรวมทั้งหมดผิดไป หรือไม่สามารถทำงานในย่านความถี่สัญญาณนาฬิกาที่สูงได้

- ***Physical Design and Analysis*** คือขั้นตอนของการผลิตเป็นวงจรจริง (technology and device mapping) โดยนำข้อมูลที่ได้จากการสังเคราะห์มาผลิต ซึ่งอาจจะอยู่ในรูปของแผงวงจรไฟฟ้า (Printed Circuit Board: PCB) ที่ประกอบด้วยอุปกรณ์หลายๆ ชิ้น หรืออยู่ในรูปของวงจรรวมเฉพาะงาน (ASIC)

- ***Post-Timing Verification*** หลังจากที่ได้วงจรจริงมาแล้ว ยังต้องมีความจำเป็นที่ต้องตรวจสอบการทำงานที่คำนึงถึงเวลาด้วย เพื่อความถูกต้องของวงจรครั้งสุดท้ายก่อนที่จะนำไปรวมเข้ากับอุปกรณ์อื่นๆ ให้เป็นระบบดิจิตอล เพราะในขั้นตอนนี้วงจรที่ออกแบบ จะประกอบด้วย input และ output pad ซึ่งเป็นจุดต่อสำหรับรับและส่งสัญญาณกับภายนอก

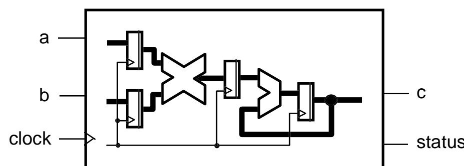
- ***System Level Verification*** หลังจากที่นำวงจรที่ออกแบบรวมเข้ากับอุปกรณ์อื่นๆ ให้เป็นระบบดิจิตอลแล้วนั้น จะต้องทดสอบการทำงานรวมทั้งระบบร่วมกับอุปกรณ์อื่นๆ อีกครั้ง เป็นการควบคุมคุณภาพของผลิตภัณฑ์

จากความอ่อนตัวของภาษา และความสามารถที่จะเขียนรูปแบบได้หลายลักษณะนี้เอง VHDL จึงเป็นเครื่องมือที่ใช้สำหรับออกแบบตั้งแต่ขั้นตอนบนสุด คือแนวความคิดที่จะแก้ปัญหาลงไปที่ละขั้นจนถึงขั้นตอนของการผลิตวงจรจริง (form idea to implementation) ข้อดีที่เห็นได้ชัดของการนำ VHDL มาใช้ในการออกแบบลักษณะ top-down นี้คือ วิศวกรออกแบบสามารถที่จะสร้างรูปแบบ (model) และจำลองการทำงาน เพื่อตรวจสอบความถูกต้องกับข้อกำหนด (specification) ตั้งแต่เริ่มแรกที่มีแนวความคิดอย่างสังเขป จากการจำลองการทำงานในระยะต้นๆ ของการออกแบบนั้น หลักการต่างๆ ที่ถูกกำหนดขึ้นใช้ในการแก้ปัญหา (สร้างวงจรให้เป็นตามความต้องการของข้อกำหนด) จะถูกตรวจสอบด้วยทุกครั้ง ก่อนที่จะมีการลงทุนในขั้นตอนสุดท้ายของการออกแบบ หรือการสร้างวงจรนั้นเอง นั้นหมายความว่าข้อผิดพลาดที่อาจจะเกิดขึ้นได้จากหลักการที่กำหนดขึ้น จะถูกตรวจพบและจัดการแก้ไขให้ถูกต้องได้ ก่อนที่จะทำงานในขั้นตอนต่อๆ ไปของขบวนการการออกแบบ

ในตัวอย่างของ multiply accumulate algorithm สามารถแสดงให้เห็นขบวนการออกแบบในลักษณะของ Top-Down Design ได้ โดยการใช้ multiply accumulate function จุดประสงค์แรกก็คือการเขียนรูปแบบของฟังก์ชันในระดับบนสุดด้วยภาษา VHDL และจำลองการทำงานของรูปแบบที่ได้ เพื่อตรวจสอบความถูกต้อง ซึ่งฟังก์ชันนี้อาจจะเป็น Discrete Fourier Transform (DFT) ก็ได้ หลังจากที่ได้ฟังก์ชันที่ต้องการแล้ว (เช่นในรูปที่ 1.1 และมีชื่อว่า "multiply_accum") สามารถนำไปใช้ได้ในรูปของ function call คือ

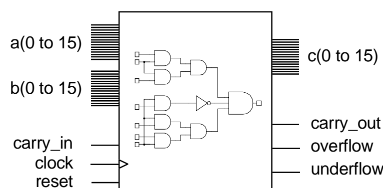
C <= multiply_accum(a, b);

ถ้าฟังก์ชันที่ได้ทำงานเป็นที่พอใจแล้ว ในขั้นตอนต่อไปจะเป็นการเริ่มต้นของการแปลงแนวความคิด ไปสู่การสร้างวงจรจริง ขบวนการดังกล่าวเกิดขึ้นโดยการ เพิ่มรายละเอียดให้กับแนวความคิดเดิม รูปที่ 1.4 แสดงให้เห็นวิธีการบรรยายแบบ dataflow (dataflow description) ของ multiply accumulate function



รูปที่ 1.4: Dataflow Description **ของ** Multiply-Accumulate Function

จาก dataflow format ที่ได้นี้ ขบวนการต่อไปคือการสร้างวงจรให้อยู่ในรูปของอุปกรณ์พื้นฐาน หรือ gate-level implementation ซึ่งอาจใช้วิธีการสังเคราะห์อัตโนมัติ ผลลัพธ์ที่ได้จากการสังเคราะห์จะเป็นการแสดงภาพวงจรด้วย gate ของฟังก์ชันต่างๆ หรือในรูปของ netlist และจะเป็นการกำหนดเทคโนโลยีจำเพาะที่จะนำไปผลิตในภายหลัง ในรูปที่ 1.5 แสดงให้เห็นรูปร่างของวงจรที่ได้จากการสังเคราะห์ในระดับ gate-level



รูปที่ 1.5: Gate-level Representation **ของ** Multiply-Accumulate Function

คุณสมบัติหลักของภาษา VHDL คือความสามารถที่จะใช้บรรยาย hardware ได้ในทุกๆ ระดับของภาพรวมทั้งระบบ ฉนั้นวิศวกรออกแบบจึงสามารถใช้เครื่องมือ (ภาษา) เพียงอันเดียวในการบรรยายทั้งระบบ ซึ่งก็เช่นเดียวกันกับเครื่องมือจำลองการทำงาน (simulator)

1.4 Terminology and Conventions

การเขียนรูปแบบของระบบดิจิตอลด้วยภาษา VHDL นั้น จะมีศัพทเทคนิคเฉพาะ ฉนั้นในบทนี้จะเป็นการบรรยาย และอธิบายศัพท์บางคำที่จะต้องพบในหนังสือเล่มนี้

- ***ลักษณะของรูปแบบ (model styles)***: ลักษณะของการเขียนรูปแบบ (model) ด้วยภาษา VHDL สามารถแบ่งได้เป็น
    - ***Behavioral Model:*** หรือเรียกอีกอย่างได้ว่า **algorithmic description** เป็นรูปแบบที่บรรยายพฤติกรรมของระบบดิจิตอล ในส่วนที่บรรยายมีโครงสร้างคล้ายกับภาษาชั้นสูง (high level language) ทั่วๆ ไป เช่น PASCAL หรือ C เป็นต้น ในการจำลองการทำงาน (simulation) คำสั่งแต่ละคำสั่ง (statement) จะถูกประเมินผลเป็นไปตามลำดับ (sequential) จากบนลงล่าง ยกเว้นในกรณีของคำสั่ง LOOP หรือการเรียกใช้โปรแกรมย่อย รูปแบบลักษณะนี้จะไม่ให้รายละเอียดที่เกี่ยวกับผลิต หรือโครงสร้างของ hardware แต่ในทางตรงข้ามที่จะให้รายละเอียดเกี่ยวกับความสัมพันธ์ระหว่าง input กับ output ที่ดี
    - ***Dataflow Model:*** เรียกอีกอย่างหนึ่งได้ว่า "Register Transfer Level" (RTL) เป็น รูปแบบที่ถูกเขียนขึ้น เพื่อจุดประสงค์ที่จะใช้เครื่องมือสำหรับสังเคราะห์วงจรอัตโนมัติ รูปแบบลักษณะนี้ส่วนใหญ่จะเป็น procedural constructs และ functional operators (ดูรูปที่ 1.1)
    - ***Structural Model:*** เป็นรูปแบบที่แสดงการเชื่อมต่อกันระหว่างอุปกรณ์ต่างๆ ที่ประกอบกันขึ้นเป็นวงจรหรือระบบดิจิตอล และสามารถเรียกอีกอย่างได้ว่า "netlist representation" เป็นการเขียนที่แสดงให้เห็นโครงสร้างของ hardware

- *Mixed-Level Model:* จากคุณสมบัติที่อ่อนตัวของภาษา VHDL จึงสามารถที่จะเขียนรูปแบบ โดยใช้ลักษณะต่างๆ ที่กล่าวมาแล้วข้างต้น บรรยายวงจรหรือระบบดิจิตอลเดียวกันได้ ฉนั้นรูปแบบเช่นนี้จึงมีการเขียมแบบผสม

♦ ***Concurrency***: ในภาษา VHDL นั้น ชุดคำสั่ง (statements) แต่ละชุดจะทำงานในเวลาเดียวกันและอิสระต่อกัน ลักษณะเช่นนี้เป็นคุณสมบัติที่เป็นความจริงทางฟิสิกค์ของวงจรอิเล็กทรอนิกส์ ชุดคำสั่งนี้เรียกว่า "**concurrent statement**" และจะทำงานก็ต่อเมื่อมีการเปลี่ยนแปลงค่าของสัญญาณ

♦ ***Sequential***: นอกจากความสามารถที่ชุดคำสั่งจะทำงานแบบ concurrent แล้ว บางครั้งการเขียนรูปแบบในลักษณะที่บรรยายพฤติกรรมของวงจร มีความจำเป็นที่จะต้องให้ชุดคำสั่งทำงานเป็นลำดับขั้นเรียงกันจากบนลงล่าง อย่างเช่นการเขียนแบบ behavioral model เป็นต้น ชุดคำสั่งที่เป็น sequential นี้จะใช้ในโปรแกรมย่อย (subprogram) และ process statement ซึ่งจะกล่าวถึงต่อไปในภายหลัง

♦ ***Driver***: สัญญาณต่างๆ (signal) ใน VHDL นั้นจะถูกควบคุมด้วยตัวขับหรือ "driver" สัญญาณเหล่านี้จะรับค่าใหม่ (ระดับของสัญญาณ) ได้ด้วยตัวขับนี้เอง

♦ ***Transaction***: การเกิด transaction กับ signal นั้นจะเกิดขึ้นเมื่อมีการกำหนดค่าๆ หนึ่งให้กับ signal นั้น ค่าใหม่ที่ signal ได้รับอาจะมีผลหรือไม่มีผลทำให้เกิดการเปลี่ยนแปลงของระดับสัญญาณ (event) เช่นการเปลี่ยนจากค่า Logic '0' เป็นค่า Logic '0' เป็นต้น

♦ ***Event***: คือการเปลี่ยนระดับค่าของ SIGNAL จากระดับหนึ่งไปสู่ระดับอื่น อย่างเช่นในระบบดิจิตอลการเปลี่ยนจาก Logic '0' เป็น Logic '1' หรือในทางตรงกันข้ามถือว่า SIGNAL นั้นเกิด "event" ฉนั้นจะเห็นได้ว่า การที่จะเกิด event ได้นั้นจะต้องเกิด transaction ด้วย แต่ในทางตรงข้ามการเกิด transaction ไม่จำเป็นต้องเกิด event ทุกครั้ง

♦ ***Sensitivity List***: คือรายชื่อของ signal ต่างๆ ที่มีผลให้เกิดการทำงานของ concurrent statement เมื่อเกิด event ขึ้นกับ signal ตัวใดตัวหนึ่งหรือหลายตัวพร้อมกันในรายชื่อนั้น

♦ ***Objects***: ในภาษา VHDL นั้นคำว่า object ใช้เขียนเพื่อบ่งบอกถึงองค์ประกอบส่วนหนึ่งของรูปแบบ ซึ่งเปรียบได้เหมือนกับภาชนะที่มีไว้สำหรับบรรจุค่าต่างๆ สามารถแบ่งออกได้เป็นสามชั้น (**class**) ด้วยกันคือ

- ***CONSTANT***: ได้แก่ object ประเภทหนึ่งที่เมื่อกำหนดค่าเริ่มต้นให้แล้ว จะคงค่านั้นไว้ตลอด ไม่สามารถดัดแปลง หรือแก้ไขได้ สามารถประกาศใช้ได้ในส่วนที่เป็นส่วนประกาศต่างๆ ของรูปแบบ (model)
- ***SIGNAL:*** หมายถึง object ประเภทหนึ่งที่สามารถกำหนดค่าที่สัมพันธ์กับเวลาให้ได้ นั้นหมายความว่า SIGNAL สามารถรับค่าได้เพียงค่าเดียวเท่านั้นในขณะเวลาหนึ่ง SIGNAL จะรับค่าๆ หนึ่งได้จากตัวขับสัญญาณหรือ driver ซึ่งตัวขับนี้อาจจะเก็บค่าในอนาคตสำหรับ SIGNAL ไว้ด้วย SIGNAL สามารถประกาศใช้ได้ในส่วนที่เป็นเนื้อที่ของ concurrent body เท่านั้น ดังนั้น SIGNAL จึงสามารถถูกนำไปใช้ได้ตลอดโครงสร้างของรูปแบบ (model) หรือที่เรียกว่า **global object**
- ***VARIABLE:*** หรือตัวแปรได้แก่ object ที่สามารถกำหนดค่าใดๆ ให้ได้ และสามารถที่จะเปลี่ยนแปลงค่าได้ตลอดการจำลองการทำงาน แต่จะเก็บค่าเพียงค่าเดียวเท่านั้นในขณะเวลาหนึ่ง เนื่องจาก VARIABLE สามารถประกาศใช้ได้ในส่วนที่เป็น sequential body เท่านั้นอันได้แก่ส่วนประกาศของ PROCESS, FUNCTION หรือ PROCEDURE ดังนั้น VARIABLE จึงสามารถนำไปใช้ได้เฉพาะในขอบเขตที่ถูกประกาศใช้เท่านั้น (local object)[1]

♦ ***การประกาศใช้ object (object declaration)***: การที่จะใช้ object ชั้นต่างๆ ตามที่กล่าวมาแล้วในการเขียนรูปแบบด้วยภาษา VHDL นั้นจะต้องมีการประกาศใช้ก่อน การประกาศใช้ object สามารถใช้ชุดคำสั่งตามโครงสร้างดังนี้

```
object_class identifier : TYPE [:= initial_value];
```

ซึ่ง object_class ได้แก่ CONSTANT, SIGNAL หรือ VARIABLE การตั้งชื่อ (identifier) เป็นไปตามกฎของภาษา VHDL ซึ่งจะกล่าวถึงในส่วนต่อไป TYPE คือการกำหนดประเภทของ object ที่ประกาศนั้นๆ นอกจากนั้นยังสามารถกำหนดค่าเริ่มต้นของ object ได้ (initial_value) ซึ่งส่วนนี้เป็นเพียง option และการประกาศจะต้องอยู่ในพื้นที่ที่กำหนดให้ของแต่ละส่วนของรูปแบบ (declarative area)

---

1 มาตรฐาน IEEE 1076-1987

- ***การตั้งชื่อ object***: การตั้งชื่อจะต้องเป็นไปตามกฎต่อไปนี้
  1) ชื่อ (identifier) ประกอบด้วยตัวหนังสือ (พยัญชนะและตัวเลข) ในภาษาอังกฤษได้แก่
     - พยัญชนะ A-Z, a-z
     - ตัวเลข 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
     - เครื่องหมายขีดเส้นใต้ (underscore) "_"
  2) ชื่อจะต้องขึ้นต้นด้วยพยัญชนะเสมอ
  3) ชื่อสามารถประกอบด้วย พยัญชนะ ตัวเลข และเครื่องหมายขีดเส้นใต้จำนวนไม่จำกัด
  4) การใช้เครื่องหมายขีดเส้นใต้ (_) ทุกครั้ง จะต้องนำหน้าด้วยตัวพยัญชนะหรือตัวเลข และตามด้วยตัวพยัญชนะหรือตัวเลขเช่นกัน
  5) พยัญชนะตัวใหญ่หรือตัวเล็กไม่มีความแตกต่างกัน (case insensitive)
  6) ห้ามใช้คำสงวน (reserved word) ของภาษา VHDL (ผนวก ง.)
- ***ค่าเริ่มต้น (initial value)***: การประกาศใช้ object ทุกครั้งควรจะต้องกำหนดค่าเริ่มต้นให้ด้วย เพราะค่าที่กำหนดนี้จะถูกนำไปใช้ เมื่อเริ่มต้นจำลองการทำงาน (simulation) ของรูปแบบ

```
SIGNAL example_signal : BIT := '1';
```

  ในกรณีที่ไม่มีการกำหนดค่าเริ่มต้น ระบบจำลองการทำงานจะนำค่าที่น้อยที่สุด (อยู่ทางซ้ายมือสุดของกลุ่มค่า) ของแต่ละประเภท (TYPE) มาเป็นค่าเริ่มต้นแทน
- ***ประเภทข้อมูล (data type)***: ได้แก่ TYPE ของ object ที่จะเป็นตัวกำหนดว่าค่าใดบ้างในกลุ่มของค่า (set of value) ของแต่ละ TYPE สามารถที่จะกำหนดให้กับ object ได้ นอกจากนั้น TYPE ยังเป็นตัวกำหนดการทำงานในลักษณะต่างๆ (operation) ของ object นั้นๆ TYPE แบ่งออกเป็น 4 ประเภทคือ
  1) **Scalar** ได้แก่ตัวเลข (numeric) ซึ่งในภาษา VHDL มีตัวเลขจำนวนเต็มบวก (INTEGER) เช่น 1, 30, 100 เป็นต้น และเลขจำนวนจริง (REAL)
     เช่น 1.0, 30., 1E2 เป็นต้น
  2) **Enumeration** กลุ่มของค่าประเภทนี้ได้แก่ ตัวหนังสือ หรือชื่อต่างๆ
  3) **Physical** ได้แก่หน่วยทางฟิสิกส์ในระบบ SI
  4) **File** เป็น TYPE ภายนอกสามารถมีค่าได้หลายๆ อย่าง[2]

---

2 เนื่องจาก TYPE ชนิดนี้ไม่มีความเกี่ยวข้องกับ hardware จึงจะไม่กล่าวในขอบเขตของหนังสือเล่มนี้

- ***ประเภทย่อยของข้อมูล (subtypes)***: สำหรับ TYPE ที่กำหนดไว้แล้ว (predefined type และ user-defined type) สามารถที่จะแบ่งออกเป็นกลุ่มย่อยลงไปได้อีก โดยที่องค์ประกอบของ TYPE ใหม่นี้จะเป็นส่วนหนึ่งของ TYPE เดิม หรือที่เรียกว่า **subtype**

```
TYPE qit IS ('0', '1', 'Z', 'X');
SUBTYPE tit IS qit RANGE '0' TO 'Z';
```

- ***ประเภทของ object ที่กำหนดไว้แล้ว (predefined type)***: ได้แก่ TYPE ที่กำหนดไว้ใน package ชื่อ STANDARD และกำหนดโดย IEEE ว่าจะต้องมีในระบบที่ใช้พัฒนา VHDL ฉนั้นจึงไม่จำเป็นต้องประกาศใช้ในทุกๆ รูปแบบที่เขียนขึ้น TYPE ประเภทนี้ได้แก่
  1) **BOOLEAN** คือกลุ่มของค่า FALSE และ TRUE
  2) **BIT** คือกลุ่มของค่า '0' และ '1'
  3) **INTEGER** คือกลุ่มของค่า -214748347 ถึง 214748347
  4) **REAL** คือกลุ่มของค่า -1.0E38 ถึง 1.05E38[3]
  5) **CHARACTER** คือกลุ่มของค่า พยัญชนะ 'A'-'Z', 'a'-'z', อักษรหรือเครื่องหมายพิเศษ และตัวอักษรควบคุม[4]
  6) **TIME** ได้แก่หน่วยเวลาที่มีค่าพื้นฐานเป็นวินาที (second ย่อด้วย s หรือ S)
  7) **SEVERITY LEVEL** คือกลุ่มของค่า NOTE, WARNING, ERROR, FAILURE (จะกล่าวในบทที่ 5 เรื่อง Assertion Statement)

- **Operator**: เนื่องจาก TYPE ของ object จะเป็นตัวบ่งบอกถึง operator ที่ object นั้นๆ สามารถกระทำได้ ตลอดจน TYPE ของผลลัพธ์ที่ได้จากการทำงาน ในภาษา VHDL แบ่ง operator ออกได้เป็นประเภทต่างตามที่แสดงในรูปที่ 1.6

- ***Delay***: หมายถึงช่วงระยะเวลาระหว่างที่เริ่มต้นของสาเหตุ จนกระทั่งเป็นผลออกมาให้เห็นของปรากฏการณ์หนึ่งๆ ในความเป็นจริงทางธรรมชาติอุปกรณ์ hardware ทุกอย่าง อาทิเช่น gate ต่างๆ ในระบบดิจิตอล จะมี delay แฝงอยู่เสมอ (ที่เรียกว่า propagation delay time) การที่ภาษา VHDL เป็นภาษาที่ใช้บรรยาย hardware จึงสามารถบรรยายพฤติกรรมของ delay ได้ ซึ่งจำแนก delay ออกเป็น

---

[3] จำนวนน้อยสุดและจำนวนมากสุดที่สามารถใช้ได้ ขึ้นอยู่กับขนาดความกว้างในการใช้ข้อมูลของ hardware (data width) ที่ใช้เป็นระบบพัฒนา VHDL ในที่นี้ใช้ขนาด 32 bits เป็นหลัก

[4] การที่ TYPE ชนิดนี้เขียนอยู่ระหว่างเครื่องหมาย '_' เพราะต้องการแยกข้อแตกต่างระหว่างพยัญชนะ A กับ a ออกจากกัน

Predefined	Operators
Logical Operators:	NOT, AND, OR, NAND, NOR, XOR
Operand TYPE:	BIT, BOOLEAN
Result TYPE:	BIT, BOOLEAN
Relational Operators:	=, /=, <, <=, >, >=
Operand TYPE:	TYPE ใดๆ ก็ได้
Result TYPE:	BOOLEAN
Arithmetic Operator:	+, -, *, /, **, MOD, REM, ABS
Operand TYPE:	INTEGER, REAL, Physical
Result TYPE:	INTEGER, REAL, Physical
Concatenation Operators:	&
Operand TYPE:	array ของ TYPE ทุกประเภท
Result TYPE:	array ของ TYPE ทุกประเภท

รูปที่ 1.6: Operator **ที่กำหนดไว้ในภาษา** VHDL [5]

1) **Delay Selection** คือการกำหนด delay ที่จะมีผลต่อ SIGNAL ในรูปแบบลักษณะใด
   Inertial Delay ได้แก่ปฏิกิริยาต่อต้านการเปลี่ยนแปลง ระบบที่ประกอบด้วย inertial delay จะแสดงผลของการหน่วงเวลาต่อเมื่อ สัญญาณที่มาบังคับให้เกิดการเปลี่ยนแปลง (จาก driver) มีช่วงระยะเวลานานกว่า inertial delay ของระบบนั้นๆ
   Transport Delay จากความหมายของคำว่า transport คือการขนส่งจากจุดหนึ่งไปยังอีกจุดหนึ่งนั้น ฉนั้น transport delay จึงแสดงผลของการหน่วงเวลาทุกครั้งเสมอไม่ว่าการเปลี่ยนแปลงของตัวขับ (driver) นั้นจะมีช่วงระยะเวลานานเท่าไร
2) **Internal Delay** ได้แก่ delay ภายในระบบของ VHDL ทั้งนี้เนื่องจากภาษา VHDL เป็นภาษาที่ชุดคำสั่งทั้งหลายทำงานแบบ concurrent ต่อกัน ซึ่งแตกต่างไปจากภาษาโปรแกรมชั้นสูงทั่วๆ ไป แต่การทำงานของระบบจำลองการทำงานด้วยเครื่องคอมพิวเตอร์ (simulator) เครื่องไม่สามารถที่จะทำงานสองคำสั่ง (หรือมากกว่า) ได้ กลไกที่จะทำให้เป็นไปตามหลักการของ concurrent statement ได้นั้นเรียกว่า **delta**

---

[5] ตามมาตรฐาน IEEE 1076-1987 ในส่วนของมาตรฐาน IEEE 1076-1993 ได้เพิ่ม operator อื่นๆ อีกเช่น logical o p e r a t o r เพิ่มฟังก์ชัน XNOR (ผนวก จ.)

delay (δ) ที่หมายถึง internal delay ของระบบพัฒนา VHDL (รายละเอียดจะกล่าวในบทที่ 5)

- ***LRM***: คือคู่มืออ้างอิง (Language Reference Manual) ของมาตรฐาน IEEE 1076 (ปัจจุบัน IEEE 1076-1993)

ตลอดหนังสือเล่มนี้การแสดงกฎเกณฑ์การเขียนคำสั่งในภาษา VHDL จะอยู่ในรูปของ Bachus-Naur Format (BNF) ซึ่งแน่นอนที่ว่าไม่สามารถที่จะนำกฎเกณฑ์ทุกอย่างมาบรรยายในหนังสือได้หมด สิ่งที่ขาดหายไปสามารถค้นคว้าเพิ่มเติมได้จากหนังสือคู่มือ IEEE Standard 1076 Language Reference Manual (LRM) การแสดงวิธีการเขียนแบบ BNF นั้น ในหนังสือได้มีการดัดแปลงไปบ้างเล็กน้อย เพื่อความสะดวกในการศึกษา และมีสิ่งที่ต้องอธิบาย ณ ที่นี้ดังนี้

- เครื่องหมายวงเล็บเหลี่ยม **[** square brackets **]**
  - สิ่งที่อยู่ในเครื่องหมายนี้เป็น option สามารถที่จะเขียนหรือไม่เขียนได้
- เครื่องหมายวงเล็บปีกกา { squiggly brackets }
  - สิ่งที่อยู่ในเครื่องหมายแสดงว่า ในภาษา VHDL อนุญาตให้มีได้ 0, 1 หรือหลายๆ ครั้ง
- ตัวพิมพ์ใหญ่ 'CAPITALIZE'
  - ในหนังสือเล่มนี้ใช้ในความหมายที่แสดงว่า คำที่เขียนด้วยตัวพิมพ์ใหญ่ เป็นคำของ VHDL และคำที่อยู่ใน standard package ที่สามารถนำมาใช้ได้ทันทีโดยไม่ต้องมีการประกาศใหม่ ปกติภาษา VHDL เป็นภาษาที่มี case insensitive เช่นการเขียนในลักษณะนี้จะมีความหมายเดียวกันหมด

```
entity = Entity = EnTiTy = ENTITY
```

นอกจากนั้นยังมีจุดประสงค์ เพื่อที่จะแยกความหมายทั่วไปของภาษาประจำวัน กับภาษา VHDL เช่น ฟังก์ชัน (function) ในภาษาทั่วไปหมายถึงหน้าที่ในการทำงาน เพื่อให้ได้ผลอย่างใดอย่างหนึ่งออกมา แต่ในขณะที่ FUNCTION เป็นโปรแกรมย่อยประเภทหนึ่งของ VHDL

- ตัวพิมพ์หนาทึบ '**BOLD**'
  - มีจุดประสงค์เพื่อต้องการเน้น ไม่ได้เป็นส่วนของกฎเกณฑ์ของภาษาแต่อย่างใด

มาดูตัวอย่างของการเขียนในลักษณะ BNF จาก IF-THEN-ELSE statement

```
    IF condition THEN
    {sequential-statement(s)}
[{  ELSIF condition THEN      }
        {sequential_statement(s)}]
[   ELSE
        {sequential_statements}]
    END IF;
```

ตามกฎเกณฑ์การเขียนสามารถที่จะเขียน **ELSIF** ได้หลายครั้ง หรือไม่มีก็ได้ คำว่า **ELSE** เป็น option จะเขียนหรือไม่เขียนก็ได้ แต่ถ้าเขียนจะมีได้เพียงครั้งเดียว ในขอบเขตของ sequential statement(s) สามารถเขียน sequential statement ต่างๆ ลงไปได้หลายๆ คำสั่งหรือไม่มีก็ได้

นอกจากนั้นในหนังเล่มนี้ผู้เรียบเรียงพยายามที่จะใช้คำศัพท์ที่เป็นภาษาไทยเท่าที่สามารถกระทำได้ และส่วนใหญ่จะเป็นคำที่ผู้อ่านทั่วไปมีความเข้าใจในความหมายดีอยู่แล้ว แต่ก็มีบางคำในภาษา VHDL ที่ให้ความหมายเฉพาะ คำเหล่านี้ผู้เรียบเรียงขอใช้ทับศัพท์ด้วยเหตุผลที่ว่า ถ้าผู้อ่านได้มีโอกาสศึกษาเพิ่มเติมจากหนังสือต่างประเทศ จะสามารถสร้างความเข้าใจได้ทันทีโดยไม่ต้องแปลกลับเป็นภาษาอังกฤษอีก สุดท้ายจะขอยกตัวอย่างบางคำเช่น

- function หรือ ฟังก์ชัน หมายถึงหน้าที่ที่กระทำการใดการหนึ่ง แต่ FUNCTION หมายถึงโปรแกรมย่อย (subprogram) ชนิดหนึ่งในภาษา VHDL[6]
- process หมายถึงขบวนการใดๆ ในการทำงาน แต่ PROCESS หมายถึง concurrent statement ในภาษา VHDL
- signal หมายถึงสัญญาณต่างๆ แต่ SIGNAL หมายถึงการประกาศใช้ object ที่มีชั้นเป็นประเภท SIGNAL

6 โปรแกรมย่อยอีกประเภทได้แก่ PROCEDURE

1.5 สรุป

ในบทแรกนี้ได้ศึกษาถึงประวัติความเป็นมาของภาษา VHDL ซึ่งในอนาคตนับว่าเป็นสิ่งที่ต้องติดตามตลอดเวลา เพราะแนวโน้มของ EDA Vendor ที่เสนอเครื่องช่วยในการออกแบบวงจรดิจิตอล ได้หันมาใช้ภาษา VHDL เป็นสื่อสำหรับสังเคราะห์วงจร ตลอดจนสามารถที่จะใช้เป็นเอกสารประกอบโครงการได้ (project document)

หลังจากนั้นได้แสดงให้เห็นถึงวิธีการออกแบบลักษณะใหม่ที่เรียกว่า Top-Down Design โดยการนำภาษา VHDL มาใช้ และสุดท้ายได้อธิบายถึงวิธีการอ่านรูปแบบที่ใช้เขียนบรรยาย กฎเกณฑ์การเขียน และลักษณะที่ใช้ในหนังสือเล่มนี้ ทั้งนี้ผู้เรียบเรียงขอใช้คำศัพท์บางตัวเป็นคำภาษาอังกฤษตามที่แสดงไว้ใน IEEE 1076 เพื่อให้เกิดมาตรฐาน และสำหรับผู้ที่ต้องการค้นคว้าเพิ่มเติมจากหนังสือหรือเอกสารทางวิชาการจากต่างประเทศ สามารถที่จะเข้าใจได้ทันทีโดยไม่ต้องแปลกลับอีกครั้ง

บทที่ 2

ส่วนต่างๆ ของแบบ (Design Units)

2.1 กล่าวนำ

ก่อนที่จะศึกษาต่อไปถึงชุดคำสั่งอื่นๆ ที่ใช้ในการเขียนรูปแบบหรือ modeling[1] ด้วยภาษา VHDL มีความจำเป็นที่จะต้องแนะนำให้รู้จักกับส่วนต่างๆ ของแบบ (design units) ที่ใช้ในภาษาเสียก่อน และนี่ก็เป็นขั้นตอนแรกที่สำคัญที่สุดของการศึกษาเรียนรู้การใช้ภาษา VHDL เขียนรูปแบบบรรยายระบบดิจิตอลในมุมมองของการออกแบบลักษณะ Top-Down Design นอกจากนั้น การที่จะเข้าใจในกฎเกณฑ์ของภาษาได้นั้น จะต้องทำความเข้าใจในเรื่องของโครงสร้าง และส่วนต่างๆ ของรูปแบบ VHDL ให้ถูกต้องเสียก่อน

ภาษา VHDL นั้นประกอบด้วยส่วนต่างๆ ที่สำคัญและเป็นพื้นฐานของการเขียนรูปแบบระบบดิจิตอลที่สำคัญ 4 หน่วยคือ

1. Entity Design Unit
2. Architecture Design Unit
3. Package Design Unit
4. Configuration Design Unit

ในบทนี้จะเป็นการศึกษาถึงความหมาย และความสัมพันธ์ระหว่างกันของหน่วยเหล่านี้ เพื่อแสดงให้เห็นถึงหลักการอย่างสังเขป ส่วนรายละเอียดนั้นจะกล่าวในบทต่อๆ ไป

---

[1] ซึ่งในภาษาคอมพิวเตอร์ชั้นสูงทั่วไปเช่น C หรือ PASCAL และในสาขา software engineer จะใช้คำว่า "programming"

เนื่องจากเอกสารและคู่มือที่เกี่ยวข้องส่วนใหญ่อ้างอิงมาตรฐาน IEEE Standard VHDL Language Reference Manual (IEEE Std 1076-1987) ฉนั้นกฎเกณฑ์ต่างๆ ในหนังสือนี้จะยึดหนังสือดัง-กล่าวเป็นหลักเช่นกัน ซึ่งปัจจุบันแล้ว (เวลาขณะเขียนหนังสือเล่มนี้) ทางสมาคมฯ[2] ได้กำหนดเป็นมาตราฐาน IEEE Std 1076-1993 สิ่งที่แตกต่างจากฉบับปี 1987 เป็นเพียงมีการเพิ่มเติมบางอย่างในภาษา[3] แต่สิ่งที่อ้างถึงในหนังสือเล่มนี้เป็นบทพื้นฐานที่ไม่มีการเปลี่ยนแปลง และจาก LRM นี้เองสามารถที่จะค้นคว้ารายละเอียดเกี่ยวกับกฎเกณฑ์ของภาษาได้

2.2 Entity Design Unit

หน่วยของแบบ (design unit) ส่วนที่ใช้สำหรับติดต่อระหว่างโลกภายนอกกับรูปแบบ (model) ที่จะเขียนขึ้น ส่วนนี้เรียกว่า "**entity design unit**" ในส่วนนี้ใช้กำหนดจุดต่อ (connection point) ของรูปแบบ กำหนดทิศทางการไหลของสัญญาณ (mode) และประเภทของค่า[4] (type of value) ที่สามารถกำหนดให้กับสัญญาณตามจุดต่างๆ (PORT) ของข้อมูลที่ไหล่ผ่านจุดต่อเหล่านั้น รูปที่ 2.1 แสดงให้เห็นโครงสร้างอย่างง่ายๆ ของ entity design unit

```
ENTITY component_name IS
    input and output ports
    physical and other parameters
END [component_name] ;
```

รูปที่ 2.1: **โครงสร้างอย่างง่าย ๆ ของ** entity design unit

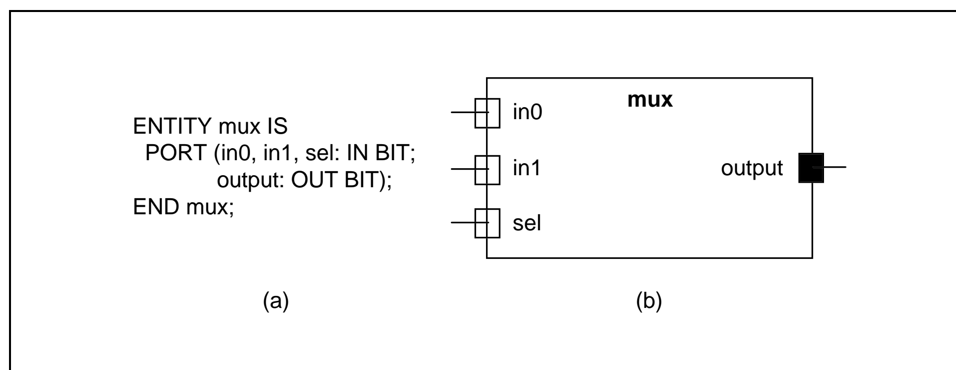
---

[2] IEEE: Institute of Electrical and Electronics Engineer

[3] ดูผนวก จ.

[4] เพื่อความเป็นสากลของการใช้ภาษา VHDL ประเภทของค่าที่กำหนดให้กับสัญญาณจะใช้คำว่า TYPE

ส่วนนี้จะขึ้นต้นด้วยคำ **ENTITY** และ **IS** ระหว่างคำทั้งสองเป็นส่วนสำหรับชื่อของรูปแบบที่ต้องการจะเขียน (component_name) สำหรับการตั้งชื่อนั้นจะต้องเป็นไปตามกฎเกณฑ์ของภาษา หลังจากนั้นจะตามด้วยส่วนที่ใช้กำหนดช่องทาง เข้า-ออก ของข้อมูล (input-output) รวมทั้งพารามิเตอร์อื่นๆ ส่วนนี้เรียกว่าส่วนหัว (entity header) และที่สำคัญคือ entity design unit จะต้องปิดท้ายด้วยคำว่า **END** และเครื่องหมายอัฒภาค หรือ semicolon (;)[5]



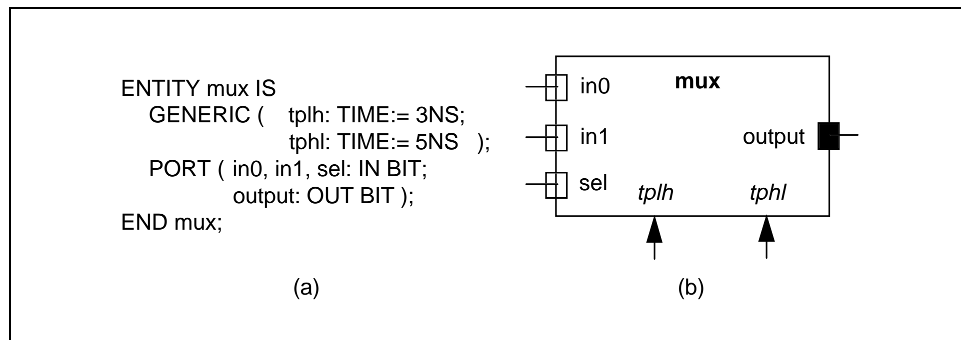
รูปที่ 2.2: **รูปแบบของ** 2:1 multiplexer, (a) VHDL entity design, (b) **มุมมองของ** interface

ในรูปที่ 2.2 เป็น entity design unit ที่บรรยายอุปกรณ์ที่มีชื่อว่า **mux** ในส่วนหัวของ entity (header) มีการกำหนดจุดต่อ 4 จุดภายใต้ชุดคำสั่ง PORT โดยที่ 3 จุดแรกเป็นจุดให้ข้อมูลไหลผ่านเข้า (input) ได้แก่ ***in0***, ***in1*** และ ***sel*** ซึ่งกำหนดด้วยทิศทางการติดต่อกับภายนอก (mode) เป็นการไหลเข้า (IN) ที่แสดงด้วยรูปสี่เหลี่ยมโปร่งในรูปที่ 2.1 ส่วนจุด ***output*** เป็นจุดให้ข้อมูลไหลออก (output) ซึ่งกำหนดด้วยทิศทางการติดต่อกับภายนอกเป็นการไหลออก (OUT) ที่แสดงด้วยรูปสี่เหลี่ยมทึบในรูปที่ 2.2 ส่วนประเภท (type) ของข้อมูลที่จะไหล เข้า-ออก นั้น เป็นประเภท BIT ที่สามารถมีค่าได้เพียงสองค่าคือ '0' และ '1' เท่านั้น

นอกจากนั้นผู้ออกแบบยังสามารถกำหนดค่าพารามิเตอร์ทางฟิสิคส์ที่เป็นข้อมูลเพิ่มเติมอื่นๆ ลงในส่วนหัวของ entity ได้อีก อาทิเช่น ข้อมูลเกี่ยวกับความเร็วในการทำงานของอุปกรณ์ อันได้แก่ propagation delay time พารามิเตอร์เหล่านี้ เรียกว่า generic ที่กำหนดด้วยคำสั่ง GENERIC จากตัวอย่างในรูปที่ 2.2 สามารถที่จะเพิ่มข้อมูลเกี่ยวกับความเร็วในการทำงานได้ตามที่แสดงในรูปที่ 2.3

[5] เช่นเดียวกับทุกๆ statement ใน VHDL จะต้องปิดท้ายด้วยเครื่องหมายอัฒภาค (;) เสมอ

เช่นเดียวกับตัวอย่างแรกหน่วยของแบบนี้มีช่องทางติดต่อกับภายนอก 4 จุด แต่ได้เพิ่มข้อมูลของ propagation delay time สำหรับนำไปใช้ในการบรรยายพฤติกรรมของอุปกรณ์ ตัวพารามิเตอร์หรือที่เรียกว่า generic นี้มีชื่อว่า **tplh** และ **tphl** มีประเภทของข้อมูลเป็นเวลา (TIME) และจะเป็นค่าตายตัว (default value) สำหรับรูปแบบนี้เสมอ ซึ่งในที่นี้จะมีค่า 3 nanosecond[6] และ 5 nanosecond ตามลำดับ ค่าตายตัวนี้สามารถที่จะเปลี่ยนแปลงให้มีค่าอื่นได้แล้วแต่กรณี ทั้งนี้ขึ้นอยู่กับอุปกรณ์ที่จะทำการเขียนรูปแบบ ซึ่งจะเห็นได้ภายหลังในบทที่ 9 ได้ว่า generic นั้นเป็นส่วนที่มีประโยชน์มาก เพราะสามารถสร้างรูปแบบที่มีความอ่อนตัวสูง สามารถนำไปใช้บรรยายอุปกรณ์ประเภทเดียวกัน แต่มีความแตกต่างกันทางเทคโนโลยีได้



รูปที่ 2.3: **รูปแบบ** 2:1 multiplexer **ที่ประกอบด้วยข้อมูลเกี่ยวกับเวลา**
(a) VHDL entity design, (b) **มุมมองของ** interface

ในบางกรณีสามารถที่จะใช้ภาษา VHDL สร้างรูปแบบที่ปราศจากช่องทางไหล เข้า-ออกของข้อมูล (input-output) ได้ ซึ่งส่วนใหญ่จะพบในการสร้างรูปแบบ สำหรับตรวจสอบการทำงานของอีกรูปแบบหนึ่ง (VHDL test bench)

```
ENTITY test_bench IS
END test_bench;
```

รูปที่ 2.4: Entity design unit **ที่ไม่มีการกำหนดช่องติดต่อกับภายนอก**

[6] 1 nanosecond = 10^{-9} second, ใช้คำย่อว่า ns. หรือ NS.

2.3 Architecture Design Unit

คือส่วนที่ใช้เขียนบรรยายกำหนดพฤติกรรมของรูปแบบ ในมุมมองของการจำลองการทำงาน (simulation) พฤติกรรมต่างๆ ที่บรรยายในส่วนนี้ขึ้นอยู่กับข้อมูลที่ผ่าน เข้า-ออก ตรงช่องทางตลอดจนพารามิเตอร์ต่างๆ (ports and generics) ที่กำหนดใน entity design unit รูปที่ 2.5 แสดง ให้เห็นโครงสร้างอย่างง่ายๆ ของ architecture design unit

```
ARCHITECTURE identifier OF component_name IS
    [ declaration ]
BEGIN
    specification of the functionality of the
    component in terms of its input lines and as
    influenced by physical and other parameters
END [identifier] ;
```

รูปที่ 2.5: **โครงสร้างอย่างง่าย ๆ ของ** architecture design unit

ส่วนของ architecture design unit นั้น เริ่มต้นด้วยคำ **ARCHITECTURE** และตามด้วยชื่อ (identifier) สิ่งที่ต้องกำหนดลงไปได้แก่ สิ่งที่แสดงให้เห็นว่า architecture นั้นใช้บรรยาย entity design unit ใด (**OF** <*entity design unit*> **IS**) ส่วนที่อยู่ระหว่าง **ARCHITECTURE** และ **BEGIN** เป็นส่วนประกาศกำหนด (architecture declarative area) ที่เป็นเพียงส่วนเผื่อเลือก (option) ในบริเวณนี้สามารถใช้เขียนประกาศกำหนดค่าต่างๆ ที่จะนำไปใช้ภายใน architecture นั้นได้ อาทิเช่น ประเภท (type) ต่างๆ (ตัวอย่างเช่น BIT, BIT_VECTOR), สัญญาณ (SIGNAL), ตัวคงที่ (CONSTANT), โปรแกรมย่อย (ได้แก่ FUNCTION และ PROCEDURE) และอุปกรณ์ (COMPONENT) ส่วนที่ใช้บรรยายความสัมพันธ์ระหว่างข้อมูลที่ไหลเข้า และไหลออกของรูปแบบ (สัญญาณที่กำหนดในชุดคำสั่ง PORT) นั้นจะถูกบรรยายในบริเวณเนื้อที่ระหว่างคำว่า **BEGIN** กับ **END** ของ architecture design unit และนอกจากนั้นชุดคำสั่งทุกคำสั่งที่อยู่ภายในบริเวณนี้จะเป็นชุดคำสั่งแบบแข่งขนาน (concurrent statement) เท่านั้น architecture design unit จะต้องปิดท้ายด้วยคำสั่ง **END** และชื่อของ architecture (identifier) นั้นๆ ที่เป็นส่วนเผื่อเลือก

โดยทั่วไปการเขียนรูปแบบระบบดิจิตอลด้วยภาษา VHDL สามารถเขียนได้ในลักษณะต่างๆ ตามที่กล่าวมาแล้วในบทที่ 1 ดังนี้

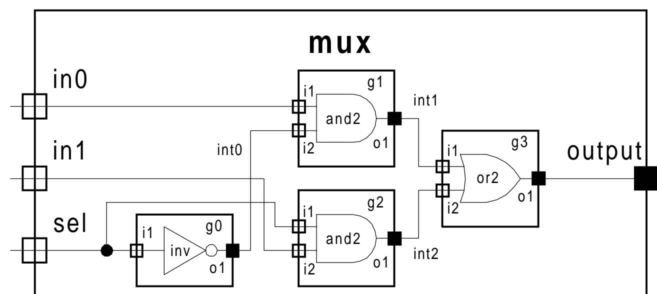
- Dataflow description
- Behavioral description
- Structure description
- Mixed model description

```
ARCHITECTURE data_flow OF mux IS
BEGIN
  output   <=   ((NOT sel) AND in0) OR (sel AND in1);
END data_flow;
```

รูปที่ 2.6: Architecture design unit **ของ** 2:1 mux **ตาม** boolean expression

$$output = (\overline{sel} \cdot in0) + (sel \cdot in1)$$

รูปที่ 2.6 ส่วนที่บรรยายความสัมพันธ์ระหว่างข้อมูลที่ไหลเข้า (*in0, in1*) กับข้อมูลที่ไหลออก (*output*) ประกอบด้วยชุดคำสั่งแบบแข่งขนานเพียงชุดเดียว ซึ่งเป็นการบรรยายพฤติกรรมของ 2:1 mux การบรรยายลักษณะนี้เรียกว่า dataflow description หรือ register transfer level (RTL)



รูปที่ 2.7(a): Architecture description **ของ** 2:1 multiplexer (structural description)

รูปที่ 2.7 เป็น architecture ของการบรรยาย 2:1 mux ในลักษณะของ structural description โดยใช้ inverter (inv ที่อุปกรณ์ g0), AND-gate 2 inputs จำนวน 2 gates (and2 ที่อุปกรณ์ g1 และ g2) และ OR-gate 2 inputs (or2 ที่อุปกรณ์ g3) มาสร้างตาม boolean expression ของรูปที่ 2.6

```
ARCHITECTURE struct OF mux IS
    COMPONENT inv
    PORT(    i1: IN BIT; o1: OUT BIT    );
    END COMPONENT;
    COMPONENT and2
    PORT(    i1, i2: IN BIT; o1: OUT BIT    );
    END COMPONENT;
    COMPONENT or2
    PORT(    i1, i2: IN BIT; o1: OUT BIT    );
END COMPONENT;
    SIGNAL int0, int1, int2: BIT;
BEGIN
    g0: inv
    PORT MAP (i1 => sel, o1 => int0);
    g1: and2
    PORT MAP (i1 => in0, i2 => int0, o1 => int1);
    g2: and2
    PORT MAP (i1 => sel, i2 => in1,  o1 => int2);
    g3: or2
    PORT MAP (i1 => int1, i2 => int2, o1 => output);
END struct;
```

รูปที่ 2.7(b): Architecture description **ของ** 2:1 mux (structural description)

รูปที่ 2.7 (a) เป็นโครงสร้าง (structure) ของการต่อวงจรภายในโดยที่มี สัญญาณ int0, int1 และ int2 เป็นสัญญาณภายใน รูปที่ 2.7 (b) เป็น architecture ของ VHDL model ที่เขียนกำหนดการเชื่อมต่อภายในด้วย VHDL netlist

```
ARCHITECTURE behav OF mux IS
BEGIN
    PROCESS (in0, in1, sel)
    BEGIN
        IF (sel = '0') THEN
            output <= in0;
        ELSE
            output <= in1;
        END IF;
    END PROCESS;
END behav;
```

รูปที่ 2.8: Architecture description **ของ** 2:1 mux (behavioral description)

การบรรยาย 2:1 mux ในลักษณะของ behavioral description ได้แสดงให้เห็นอีกครั้งในรูปที่ 2.8 ซึ่งจะเห็นได้ว่าส่วนที่เป็น architecture design unit ทั้งหมด (รูปที่ 2.6, 2.7 และ 2.8) ต่างบรรยายพฤติกรรมเดียวกัน และจะให้ผลลัพธ์จากการจำลองการทำงานที่เหมือนกัน

2.4 Package Design Unit

ข้อมูลต่างๆ ตลอดจนโปรแกรมย่อย (subprogram) ที่เป็นประโยชน์ต่อการเขียนรูปแบบบรรยายระบบดิจิตอล สามารถเก็บไว้ในส่วนที่เรียกว่า package ได้ และข้อมูลเหล่านี้สามารถนำไปใช้ได้โดย entity design unit, architecture design unit หรือจาก package design unit อื่นๆ ด้วยชุดคำสั่ง USE statement[7] นอกจากนั้นสิ่งที่นิยมทำกันมากคือรูปแบบ (model) มาตรฐานต่างๆ อาทิเช่น standard components (model ของ IC ตระกูล 74xx)[8] จะถูกเก็บไว้ใน package ที่ทุกคนสามารถเข้าถึง และนำไปใช้ได้ สิ่งที่สามารถประกาศหรือบรรจุได้ใน package ได้แก่

- Subprogram
- Types
- Constants
- Signals
- Aliases *
- Attributes **
- Component
- Disconnection Specification *

* *จะยังไม่กล่าวในขอบเขตของหนังสือเล่มนี้*

** *จะกล่าวในบทที่ 9*

โดยปกติแล้ว package จะแบ่งเป็นสองส่วนคือ[9]

1) Package declaration
2) Package body

เนื่องจาก package ถูกสร้างเป็นส่วนแยกต่างหากออกจากรูปแบบ (model) ที่กำลังเขียนอยู่ ฉนั้นการที่จะนำ package ไปใช้นั้น จะต้องมีการเชื่อมโยงหรืออ้างอิงเสียก่อน ซึ่งในภาษา VHDL สามารถกระทำได้ด้วยชุดคำสั่ง USE statement

---

[7] กล่าวในบทที่ 3

[8] ดูผนวก ข.

[9] "An introduction to Modeling VHDL", Mentor Graphics, Revision 1.3, November 1991

2.4.1 Package declaration

ส่วนที่มีความสำคัญที่สุดของ package (มองในแง่ของการนำไปใช้จากภายนอก) ได้แก่ package declaration เพราะจะเป็นส่วนที่กำหนดชื่อ (identifier) ของสิ่งที่ประกาศอยู่ภายใน package สำหรับนำไปใช้ภายนอกตัวของ package เอง ถ้าสิ่งใดๆ ถูกประกาศในส่วนของ package body แต่ไม่ถูกประกาศใน package declaration จะไม่สามารถถูกนำค่า และพฤติกรรมไปใช้จากส่วนนอกได้ ซึ่งสามารถเปรียบเทียบได้กับสิ่งที่ประกาศไว้ในส่วนของ entity declaration คือ interface ที่มีหน้าที่ติดต่อกับโลกภายนอก ฉนั้นโดยทั่วไปแล้ว package สามารถสร้างขึ้นได้โดยไม่จำเป็นต้องมีส่วน body และยังสามารถถูกนำไปใช้จากรูปแบบ (model) ภายนอกได้เช่นใช้สำหรับประกาศ TYPE หรือ signal (global) เช่นเดียวกันกับ package body ที่ไม่จำเป็นต้องมี package declaration แต่ package นั้นจะไม่สามารถถูกนำไปใช้จากรูปแบบ (model) อื่นได้ การเขียน package declaration มีกำเกณฑ์ตามที่แสดงในรูปที่ 2.9

```
PACKAGE package_name IS
   package_declarative_part
END package_name ;
```

รูปที่ 2.9: **โครงสร้างของ** package declaration

คำว่า PACKAGE และ END PACKAGE กำหนดขอบเขตของ package declaration ระหว่างนั้นจะเป็นส่วนที่ใช้ประกาศต่างๆ สิ่งที่สามารถประกาศในส่วนนี้ได้แก่

- ส่วนประกาศกำหนดโปรแกรมย่อย (Subprogram declarations)
- Type declaration
- Subtype declarations *
- Object declarations (signals, constants)
- Alias declarations *
- Attribute specifications **
- Component specifications
- Disconnection specification *

* *จะยังไม่กล่าวในขอบเขตของหนังสือเล่มนี้*

** *จะกล่าวในบทที่ 9*

รูปที่ 2.10 เป็นตัวอย่างของการเขียน package declaration ที่มีการประกาศ TYPE, CONSTANT, COMPONENT และ SIGNAL

```
PACKAGE example IS
        TYPE cd IS ('C', 'D');
        CONSTANT pi : REAL := 3.14159;
        COMPONENT ttl_74163 IS
            PORT (   a, b: IN BIT;
                        c: OUT BIT   );
        END COMPONENT;
        SIGNAL global_clock: BIT;
END example;
```

รูปที่ 2.10: **ตัวอย่างของ** package declaration

2.4.2 Package body

โครงสร้างที่ประกอบด้วยคำสั่งต่างๆ ในรูปของคำสั่งลำดับ (sequential statement) ที่ใช้บรรยายฟังก์ชันการทำงานของโปรแกรมย่อย (subprogram) ทั้งหลายที่ชื่อของโปรแกรมย่อยนั้นๆ ที่ถูกประกาศไปในส่วนของ package declaration แล้ว[10] จะถูกเก็บไว้ใน package body ทั้งนี้รวมทั้ง deferred constants (อันได้แก่ตัวคงที่ที่ถูกประกาศชื่อก่อนในส่วนของ declaration แต่ถูกกำหนดค่าในส่วนของ body ของ package) ฉนั้นส่วน package body จึงไม่จำเป็นต้องมี ถ้าในส่วน package declaration ไม่มีการประกาศชื่อ (identifier) ที่เป็นโปรแกรมย่อย (subprogram) หรือ deferred constant การเขียน package body นั้นเป็นไปตามกฎเกณฑ์ที่แสดงในรูปที่ 2.11

```
PACKAGE BODY package_name IS
        declarative part
END package_name ;
```

รูปที่ 2.11: **โครงสร้างของ** package body

[10] โดยมีจุดประสงค์ที่จะทำให้ package นั้นเป็นที่เก็บโปรแกรมย่อย (subprogram) ทั้งหลายไว้สำหรับนำไปใช้กับรูปแบบอื่นได้

ชื่อของ package_name ที่ใช้ใน package body จะต้องเป็นชื่อเดียวกับชื่อที่กำหนดไว้ใน package declaration ในรูปที่ 2.12 แสดงตัวอย่างของการเขียน package (declaration และ body ที่สัมพันธ์กัน) โดยนำการกำหนดโปรแกรมย่อย (subprogram) ประเภท FUNCTION

```
--  package declaration
PACKAGE pack_funct IS
        FUNCTION mean (a, b, c : REAL) RETURN REAL;
END pack_funct;
--  package body
PACKAGE BODY pack_funct IS
        FUNCTION mean (a, b, c : REAL) RETURN REAL IS
        BEGIN
            RETURN (a + b + c)/3.0;
        END mean;
END pack_funct;
```

รูปที่ 2.12: **ตัวอย่างการเขียน** package

ในส่วนของ package declaration จะบรรจุส่วนที่เรียกว่า function declaration ในที่นี้เป็นการประกาศชื่อของโปรแกรมย่อย (FUNCTION) **mean** และส่วนที่บรรยายการทำงานของโปรแกรมย่อย **mean** (เรียกว่า function body) จะถูกเก็บไว้ในส่วน package body แต่สิ่งที่สำคัญอย่างหนึ่งของการเขียน package คือ ***ก่อนที่จะนำ package ไปใช้ (โดยการอ้างจากภายนอก) package นั้นจะต้องถูกวิเคราะห์เสียก่อนว่าถูกต้อง หรือพูดง่ายๆ ได้ว่า จะต้องผ่านการ compile ก่อนนั้นเอง***

2.5 Configuration Design Unit

ดังที่ทราบกันแล้วว่ารูปแบบ หนึ่งของระบบดิจิตอลไม่ว่าจะเป็นอะไร จะมี entity design unit ได้เพียงหน่วยเดียวเท่านั้น แต่ในขณะที่ entity design unit หนึ่งหน่วยนี้อาจจะมี architecture ที่เป็นหน่วยรองได้หลายหน่วย

ดังนั้นจึงเกิดคำถามขึ้นว่า ***ในการจำลองการทำงานของรูปแบบ (model) นั้น simulator จะนำ architecture อันไหนไปจำลอง?*** คำตอบของคำถามนี้คือ ต้องบอกให้ simulator ทราบ และในรูปแบบ VHDL นั้นการบอกหรือกำหนดคือการใช้ CONFIGURATION ประกอบ entity กับ architectur design unit ที่ต้องการเข้าด้วยกัน รูปที่ 2.13 แสดงกฎเกณฑ์การเขียน configuration

```
CONFIGURATION identifier OF entity_name IS
      configuration_declarative_part
END;
```

รูปที่ 2.13: **โครงสร้างของ** configuration

ในรูปที่ 2.14 เป็นรูปแบบ (model) ง่ายๆ ของ AND-gate 2 input ที่มีส่วนรองอันได้แก่ architecture สองแบบคือ dataflow description และ behavioral description

```
ENTITY and2 IS
      GENERIC (ttl_delay : TIME := 3NS);
      PORT (   in1, in2 : IN BIT;
               output : OUT BIT    );
END and2;

ARCHITECTURE dataflow OF and2 IS
BEGIN
      output <= in1 AND in2 AFTER ttl_delay;
END dataflow;

ARCHITECTURE behave OF and2 IS
BEGIN
      PROCESS (in1, in2)
      BEGIN
         IF (in1 = '1' AND in2 = '1') THEN
            output <= '1' AFTER ttl_delay;
         ELSE
         output <= '0';
      END PROCESS;
END behave;
```

รูปที่ 2.14: VHDL model **ของ** AND-gate 2 inputs

ก่อนที่จะนำรูปแบบ (model) ไปจำลองการทำงานจะต้องมีการประกอบ architecture ที่ต้องการเข้ากับ entity design unit เสียก่อน (configuration) ซึ่งระบบ VHDL ส่วนใหญ่ถ้าไม่กำหนดการประกอบ architecture เครื่อง simulator จะนำ architecture หน่วยสุดท้ายที่ผ่านการวิเคราะห์ไปใช้จำลองการทำงาน ในรูปที่ 2.15 แสดงการประกอบ architecture ชื่อ dataflow เข้ากับ entity design unit

```
CONFIGURATION dataflow_and OF and2 IS
        FOR dataflow
        END FOR;
END dataflow_and;
```

รูปที่ 2.15: Configuration **ของรูปแบบ** (model) and2

ในตัวอย่างแสดงการประกอบโครงร่าง (configuration) ชื่อ dataflow_and ให้เป็นตัวกำหนดการเชื่อม entity ชื่อ and2 เข้ากับ architecture ชื่อ dataflow ซึ่งจะเห็นได้ว่าเป็นการประกอบโครงร่างอย่างง่ายๆ ในรูปที่ 2.16 จะเป็นโครงสร้างที่ซับซ้อนมากขึ้น[11]

ตัวอย่างในรูปที่ 2.16 (a) configuration ชื่อ decode_llcon เชื่อมต่อ entity ชื่อ decode กับ architecture ชื่อ structure และภายใน architecture ชื่อ structure นี้ประกอบด้วยอุปกรณ์ย่อยๆ ซึ่งแต่ละตัวมีลักษณะการประกอบกับส่วนอื่นด้วยโครงสร้าง FOR....END และ FOR....END ที่กำหนด configuration จำเพาะที่อยู่ภายใต้อีกครั้ง (เรียกว่า configuration specification) และ configuration ภายในนี้จะต้องผ่านการวิเคราะห์สำหรับ configuration ปัจจุบัน (วงนอก) ผลลัพธ์ที่ได้จากการ วิเคราะห์จะต้องถูกเก็บไว้ภายใต้ working directory (symbolic name WORK) การกำหนดอุปกรณ์จำเพาะนั้น สามารถที่จะ กระทำได้โดยการบอกชื่อ label เช่นในกรณีของ i1 สำหรับ inv หรือใช้ ALL หรือ OTHERS ดังเช่นในกรณีของอุปกรณ์ and3 เป็นต้น ในที่นี้หมายความว่าอุปกรณ์ inv จะใช้ configuration ชื่อ invcon จาก library WORK (หรือ working directory ที่มีชื่อตายตัวว่า WORK) ที่กำลังทำงานอยู่ จะเห็นได้ว่าเป็นการสร้าง configuration ซ้อน configuration การที่จะใช้งาน configuration บนสุด (decode_llcon) ได้ configuration ล่างสุด (invcon และ and3con) จะต้องผ่านการ วิเคราะห์ว่าถูกต้องมาแล้ว การสร้างโครงสร้างเช่นนี้เรียกว่า lower-level configuration

[11] Douglas L. Perry. "VHDL" : หน้า 196-197, McGRAW-HILL International Edition, 1991

รูปที่ 2.16 (b) ใช้ entity และ architecture design unit เดียวกับรูป (a) สำหรับอุปกรณ์ inv (กำหนดในส่วน architecture declaration) ถูกกำหนดจำเพาะด้วย label i1 กับอุปกรณ์ที่มี entity ชื่อ inv (สามารถมีชื่อเหมือนหรือต่างกันได้) และใช้ architecture ชื่อ behave ที่ยังคงเหลือในโครงสร้าง (ถ้ามีการใช้อุปกรณ์ inv มากว่าหนึ่งตัว) จะถูกประกอบเข้ากับอุปกรณ์ที่มี entity ชื่อ inv เช่นกันแต่จะใช้ architecture ชื่อ dataflow แทน ทั้งนี้ entity design unit ชื่อ inv และ architecture design unit ชื่อ behave และ dataflow จะต้องถูกวิเคราะห์ก่อน ผลลัพธ์ที่ได้จากการวิเคราะห์จะต้องถูกเก็บไว้ภายใต้ working directory การประกอบลักษณะนี้เรียกว่า entity-architecture pair configuration

```
ARCHITECTURE examp_config OF decode IS
    COMPONENT inv PORT (in1: IN BIT; o1: OUT BIT);
    END COMPONENT;
    COMPONENT and3 PORT (in1, in2, in3 : BIT; o1: OUT BIT);
    END COMPONENT;
    :
END examp_config;
```

```
CONFIGURATION decode_llcon OF decode IS
    FOR structural
        FOR i1: inv USE CONFIGURATION WORK.invcon;
        END FOR;
        FOR i2: inv USE CONFIGURATION WORK.invcon;
        END FOR;
        FOR ALL: and3 USE CONFIGURATION WORK.and3con;
        END FOR;
    END FOR;
END decode_llcon;
```

รูปที่ 2.16(a): **ตัวอย่างของ** configuration

```
CONFIGURATION decode_eacon OF decode IS
    FOR structural
        FOR i1: inv USE ENTITY WORK.inv (behave);
        END FOR;
        FOR OTHERS: inv USE ENTITY WORK.inv (dataflow);
        END FOR;
        FOR a1: and3 USE ENTITY WORK.and3 (behave);
        END FOR;
        FOR OTHERS: and3 USE ENTITY WORK.and3 (dataflow);
        END FOR;
    END FOR;
END decode_eacon;
```

รูปที่ 2.16(b): **ตัวอย่างของ** configuration

2.6 สรุป

ในบทนี้ได้ศึกษาถึงส่วนที่สำคัญที่สุดของโครงสร้างใน VHDL model ที่มีอยู่ด้วยกัน 4 หน่วยคือ entity-, architecture-, package- และ configuration design unit

ส่วนของ entity, architecture และ configuration ประกอบกันเข้าเป็นรูปร่างของวงจรที่ออกแบบ รูปแบบเหล่านี้รวมกันในรูปของฟังก์ชันการทำงานของส่วนต่างๆ โดยที่ entity design unit มีหน้าที่กำหนดการติดต่อข้อมูล (interface) กับภายนอกของอุปกรณ์ ส่วน architecture กำหนดพฤติกรรมของอุปกรณ์ เนื่องจากอาจจะมีการบรรยายพฤติกรรมได้หลายลักษณะ สำหรับ entity ตัวเดียว จึงมีความจำเป็นที่ต้องประกอบ architecture ที่ต้องการเข้ากับ entity ดังนั้น configuration design unit จึงทำหน้าที่นี้ด้วยการเชื่อ entity- กับ architecture design unit เข้าด้วยกัน นอกจากนั้น entity ยังเป็นส่วนที่จะผ่านพารามิเตอร์ต่างๆ เข้าสู่โครงสร้างภายในเช่น architecture

ส่วนที่เป็น package design unit นั้นคือที่รวมของฟังก์ชันต่างๆ ที่สามารถนำไปใช้ได้จากรูปแบบ (model) อื่นๆ สิ่งที่บรรจุอยู่ภายใน package อาจจะเป็น TYPE declaration หรือโปรแกรมย่อย (subprogram) ตลอดจนอุปกรณ์ต่างๆ และสามารถที่จะเข้าสู่ได้ด้วยคำสั่ง USE statement

บทที่ 3
ความสัมพันธ์ระหว่างส่วนต่างๆ ของแบบ
(Design Unit Relationships)

3.1 กล่าวนำ

จากบทที่แล้วผ่านมาได้รู้จักหน่วยต่างๆ ของรูปแบบ (design unit) ที่มีทั้งหมด 4 หน่วยด้วยกัน design unit เหล่านี้จะประกอบกันขึ้นเป็นรูปแบบ VHDL ที่สมบูรณ์ ในแต่ละหน่วยมีกฎเกณฑ์การเขียนที่แตกต่างกันออกไป ตามที่แสดงให้เห็นในรูปของ BNF พร้อมทั้งตัวอย่างประกอบ แต่ความสัมพันธ์ภายใน model ระหว่าง design unit ต่างๆ นั้นยังไม่ได้มีการกล่าวถึง

ฉนั้นในบทนี้จะเป็นการบรรยายที่ครอบคลุมหัวข้อ ที่เกี่ยวกับความสัมพันธ์ระหว่างหน่วยตลอดจนกลไกของการทำงานที่มีผลต่อกัน เมื่อแต่ละส่วนถูกนำมาประกอบเข้าด้วยกัน ซึ่งจะเป็นการศึกษาในหลักการของ "library" ในภาษา VHDL ในบางครั้งอาจจะมอง library ในภาษา VHDL เสมือนกับตู้ที่ใช้เก็บอุปกรณ์อิเล็กทรอนิกส์ต่างๆ (architecture) ที่ถูกจัดให้อยู่ตามลิ้นชักต่างๆ ของตู้ (entity design unit)[1] ดังนั้นจึงสามารถแบ่ง design unit ใน VHDL ออกได้เป็นสองส่วนคือ หน่วยหลัก (primary) และหน่วยรอง (secondary) ที่มีความสัมพันธ์กัน

3.2 Libraries

หลักการของ library คือพื้นฐานสำคัญที่จะสร้างความเข้าใจในความสัมพันธ์ระหว่าง design unit ต่างๆ ของ VHDL หลังจากที่ design unit ถูกวิเคราะห์ตรวจสอบความถูกต้องตามกฎเกณฑ์การเขียน และความถูกต้องของฟังก์ชันหรือพฤติกรรมการทำงานแล้ว (compiled and simulated) ผลลัพธ์ที่ได้จากการวิเคราะห์จะถูกเก็บไว้ในส่วนที่เรียกว่า library

---

[1] Louis Baker, VHDL Programming with Advanced Topics, John Wiley and Sons, Inc., 1993

ในมาตรฐาน IEEE 1076[2] ไม่ได้กำหนดกลไกหรือกฎเกณฑ์ไว้ว่า library จะถูกสร้างและมีการจัดการอย่างไร ฉนั้นในปัจจุบันการจัดการต่างๆ ที่เกี่ยวกับ library จึงขึ้นอยู่กับผู้พัฒนาระบบ VHDL ที่แต่ละระบบจะมีความแตกต่างกัน ในบทนี้จึงจะกล่าวเฉพาะหลักการทั่วไปของ library ใน VHDL ที่ทาง IEEE กำหนด

ส่วนที่เป็น design library ของ VHDL นั้น สามารถนำไปใช้ได้ใน design unit อื่นๆ โดยการอ้างถึงชื่อของ library นั้นๆ ชื่อของ library นี้เรียกว่า ชื่อสัญลักษณ์ หรือ symbolic name และการอ้างถึงดังกล่าวนี้ก็เพื่อที่จะเข้าสู่สิ่งที่อยู่ภายใน library นั้นเอง ข้อูลที่อยู่ภายในจะแบ่งเป็น

- **หน่วยหลัก (primary units)**
  - entity declarations
  - package declarations
  - configuration specifications
- **หน่วยรอง (secondary units)**
  - architecture bodies
  - package bodies

เนื่องจากหน่วยรองมีความสัมพันธ์กับหน่วยหลัก ฉนั้นหน่วยรองจึงต้องถูกวิเคราะห์ภายหลังจากที่หน่วยหลักถูกวิเคราะห์แล้ว package, entity, architecture หรือ configuration design unit ที่อยู่ภายในเหล่านี้จะต้องถูกกฎเกณฑ์การเขียน ขบวนการที่บรรจุ design unit ต่างๆ ลงใน library เป็นไปอย่างอัตโนมัติเมื่อ VHDL source file ที่ผู้ออกแบบเขียนขึ้นผ่านการวิเคราะห์ และตรวจสอบว่าถูกต้องตามกฎเกณฑ์การเขียนแล้ว (หรือเรียกได้อีกอย่างว่าผ่านการ compile) หน่วยรองจะต้องอยู่ใน library เดียวกันกับหน่วยหลักที่เกี่ยวข้องกัน โดยปกติแล้วจะมี library อยู่สองประเภท[3] ได้แก่

1) **Working library**
   หมายถึง working directory ในระบบคอมพิวเตอร์ นั้นคือ directory ที่กำลังทำงานอยู่ ชื่อของ working library จะถูกกำหนดให้เป็น **WORK** เสมอ

---

[2] IEEE Std. 1076-1987

[3] Mentor Graphics, "An Introduction to Modeling in VHDL", Instructors Note, Revision 1.3, November 1991

2) **Resource libraries**

ทำหน้าที่เป็นที่เก็บข้อมูลเพิ่มเติมสำหรับ design unit สามารถตั้งชื่ออะไรก็ได้เพื่อกำหนดสถานที่อยู่ หรือเรียกว่า ชื่อสัญลักษณ์ (symbolic name) ในการกำหนดชื่อ VHDL ได้สงวนชื่อของ library ไว้สองชื่อ คือ **STD** และ **WORK** (หมายถึง working library เสมอ) ใน STD ประกอบด้วยสอง packages คือ

- ***Package STANDARD*** กำหนดโดย IEEE 1076 โดยที่ภายในจะประกอบด้วยการประกาศ (declaration) ต่างๆ อาทิเช่น VHDL type (INTEGER, REAL, TIME, BIT และ BOOLEAN เป็นต้น)
- ***Package TEXTIO*** ภายในประกอบด้วยโปรแกรมย่อยต่างๆ ที่ใช้สำหรับแก้ไขดัดแปลงข้อมูลที่เขียนในรูปของ ASCII code

เมื่อ VHDL source file ที่ผู้ออกแบบเขียนถูกวิเคราะห์ (compile) ผลลัพธ์ที่ได้ (เมื่อไม่มีข้อผิดกฎเกณฑ์การเขียน) จะถูกเก็บไว้ใน library ที่มีชื่อว่า WORK ทุกๆ ตำแหน่งและเวลาสามารถที่จะอ้างอิง library ที่จะบรรจุผลลัพธ์จากการวิเคราะห์ได้สองวิธีคือ

1) ใช้ absolute address ของ library นั้นคือชื่อสัญลักษณ์ (symbolic name) ของ library
2) ใช้ relative address ของ library คือ WORK

3.3 Design Unit Names

หน่วยหลักที่อยู่ภายใน library จะต้องมีชื่อที่บ่งบอกถึงเอกลักษณ์ หรือที่เรียกว่า "identifier" และ identifier ที่ตั้งขึ้นนี้จะต้องเป็นไปตามกฎการเขียนของภาษา VHDL (บทที่ 1 การตั้งชื่อ object)

ตัวอย่างของการตั้งชื่อที่ถูกต้อง:	ตัวอย่างของการตั้งชื่อที่ไม่ถูกต้อง:
entity1	1_entity
ENTITY1	_and_gate
a_long_legal_identifier_with_underscores	and
CasE_InsEnsITIve	ampersand&

รูปที่ 3.1: **ตัวอย่างการตั้งชื่อในภาษา** VHDL

ชื่อของหน่วยรองจะต้องบ่งบอกให้เห็นเอกลักษณ์ของหน่วยหลักที่ตัวเองเกี่ยวข้องอยู่ ดังเช่นถ้ามี ENTITY ชื่อ and_gate เป็นหน่วยหลัก ดังนั้นหน่วยรองได้แก่ ARCHITECTURE จะมีชื่อใดๆ ก็ได้ตราบเท่าที่มีการบ่งบอกความสัมพันธ์ระหว่างตัวเองกับหน่วยหลัก ว่าเป็น architecture ของใครดังตัวอย่างในรูปที่ 3.2

```
ARCHITECTURE behavioral OF and_gate IS
หรือ
ARCHITECTURE dataflow OF and_gate IS
```

รูปที่ 3.2: **การตั้งชื่อหน่วยรองที่สัมพันธ์กับหน่วยหลัก**

แต่อย่างไรก็ตามบางครั้งอาจจะมี entity สองหน่วยที่มีชื่อต่างกันแต่มีชื่อ architecture เดียวกันได้ดั่งในรูปที่ 3.3

```
LIBRARY ttl_lib

ENTITY "and_gate"
ARCHITECTURE "behave"

ENTITY "or_gate"
ARCHITECTURE "behave"
```

รูปที่ 3.3: **หน่วยหลักสองหน่วยมีหน่วยรองที่ชื่อเดียวกัน**

3.4 File Organization

ในมาตรฐาน IEEE 1076 VHDL ไม่ได้กำหนดไว้ว่า design unit ที่เขียนขึ้นในระหว่างการออกแบบ จะถูกจัดระเบียบอย่างไร รูปแบบที่เขียนขึ้นนั้นเป็นข้อมูลที่ประกอบด้วยตัวหนังสือ (text

file) ข้อมูลนี้สามารถที่จะเขียนขึ้นมา แก้ไข หรือดัดแปลงจากผู้ออกแบบได้ หลังจากนั้นจะถูกวิเคราะห์ด้วย VHDL analyzer (เปรียบเทียบได้กับ compiler ในภาษาโปรแกรมเช่น C หรือ

PASCAL เป็นต้น) VHDL analyzer ทำการวิเคราะห์ design file เพื่อตรวจสอบความถูกต้องของกฎการเขียน (syntax) มีอยู่เพียงอย่างเดียวเท่านั้นที่ VHDL ต้องการคือ design unit ต่างๆ ที่อยู่ใน design file จะต้องครบสมบูรณ์ในตัวภายใน design file เดียวกัน ซึ่งเป็นเรื่องธรรมดามากที่ใน design file หนึ่งจะประกอบด้วย design unit จำนวน 1, 2, 3, ...100, .. หน่วย

3.5 Visibility

ความหมายของคำว่า **visibility** ในภาษา VHDL เป็นพื้นฐานที่จะทำให้เข้าใจถึงความสัมพันธ์ระหว่างส่วนต่างๆ ของรูปแบบที่ดี เพราะการที่โครงสร้างต่างๆ ที่ประกาศขึ้นเพื่อนำไปใช้กับรูปแบบที่กำลังเขียน (ออกแบบ) จะต้องมีการสร้างให้มีความสัมพันธ์กันก่อน จึงจะสามารถนำมาอ้างอิงได้ ดังในตัวอย่างของ architecture design unit สามารถที่จะแก้ไขดัดแปลงข้อมูลที่ผ่านเข้ามาทาง PORT ของ entity design unit ได้ (รวมทั้งส่งข้อมูลออก) โดยที่ไม่ต้องมีการประกาศกำหนดขึ้นใหม่ในส่วนของ architecture สาเหตุที่ทำเช่นนี้ได้เพราะว่าส่วนที่เป็น architecture เป็นหน่วยรอง (secondary unit) ที่อยู่ภายใต้หน่วยหลัก (primary unit) อันได้แก่ entity และการที่ใช้ชุดคำสั่ง PORT กำหนดช่องทาง เข้า-ออก ของสัญญาณ (กำหนดด้วยชื่อ) คือการทำให้ชื่อเหล่านั้น **visible** สำหรับ entity design unit และสามารถนำไปใช้กับหน่วยรองได้ อันได้แก่ส่วนที่เป็น architecture ของ entity นั้นๆ เช่นเดียวกันกับค่า generic ที่ประกาศกำหนดใช้ในส่วน entity ฉนั้น โดยหลักการแล้วข้อมูลใดๆ ที่ visible สำหรับหน่วยหลักแล้วจะ visible สำหรับหน่วยรอง ด้วยเช่นเดียวกับคู่ของ package (package declaration/package body)

การที่จะทำให้ข้อมูลที่บรรจุอยู่ใน package สามารถถูกนำออกไปใช้ในส่วนของ entity และ architecture หรือใน package อื่นๆ ภายนอกได้จะต้องทำให้ package นั้น visible สำหรับส่วนต่างๆ เหล่านั้นเสียก่อน ซึ่งสามารถกระทำได้โดยใช้ชุดคำสั่ง LIBRARY และ USE ก่อนที่จะถึงส่วนที่เป็นรูปแบบที่กำลังออกแบบ

3.6 ชุดคำสั่ง LIBRARY

ในตอนต้นของบทนี้ได้กล่าวถึงการสร้าง design library ไปแล้ว การที่จะเข้าสู่หรืออ้างอิงส่วนดังกล่าว กระทำได้โดยการอ้างอิงชื่อ (identifier) ของ library นั้น ก่อนที่จะทำเช่นนั้นได้ต้องทำให้ชื่อของ library นั้น visible สำหรับหน่วยที่ต้องการใช้ ชุดคำสั่ง LIBRARY ในรูปที่ 3.4 ชุดคำสั่งทำให้ library ที่มีชื่อสัญลักษณ์ว่า **IEEE** สามารถนำมาใช้ในส่วนที่เป็น entity ชื่อ **test** ได้ นั้นคือ library ชื่อ **IEEE** จะ visible สำหรับ entity ชื่อ **test** รวมทั้งส่วนที่เป็นหน่วยรองด้วย (architecture)

```
LIBRARY IEEE;
--
ENTITY test IS
END test;
```

รูปที่ 3.4: **การทำให้** library **ชื่อสัญลักษณ์** IEEE visible

โดยทั่วไประบบพัฒนา VHDL[4] ทุกระบบจะแฝงคำสั่งในรูปที่ 3.5 ไว้บนบรรทัดแรกของรูปแบบทุกครั้งที่ผู้ออกแบบเริ่มทำงาน คำสั่งดังกล่าวทำให้ library ชื่อสัญลักษณ์ (symbolic name) STD และ WORK สามารถนำมาอ้างอิงได้โดยไม่ต้องใช้คำสั่ง LIBRARY ใหม่

```
LIBRARY STD;
LIBRARY WORK;
```

รูปที่ 3.5: **คำสั่ง** LIBRARY **ที่แฝงอยู่ในทุก ๆ รูปแบบ**

[4] ข้อบังคับของมาตรฐาน IEEE 1076

3.7 ชุดคำสั่ง USE

เนื่องจากภายใน library (กำหนดชื่อเป็นชื่อสัญลักษณ์) อาจประกอบด้วย package หลายๆ ชุด หลังจากทำให้ library นั้น visible แล้ว (คำสั่ง LIBRARY) จะต้องกำหนดเฉพาะด้วยคำสั่ง USE คำสั่งนี้มีหน้าที่คล้ายกับคำสั่ง INCLUDE ในภาษาโปรแกรมทั่วไป รูปที่ 3.6 แสดงกฎการเขียนคำสั่ง USE

```
USE library_name.package_name[.element_of_package] ;
```

รูปที่ 3.6: **กฎการเขียนชุดคำสั่ง** USE

คำสั่ง USE จะตามด้วยชื่อสัญลักษณ์ของ library และเครื่องหมายจุด (.) หลังเครื่องหมายจุดอันแรกเป็นชื่อของ package ที่ต้องการภายใน library นั้นๆ และตามด้วยเครื่องหมายจุด (.) อันที่สอง สิ่งที่ตามมาได้แก่ชื่อขององค์ประกอบย่อยที่ประกาศอยู่ภายใน package นั้นๆ ซึ่งอาจจะเป็น TYPE, CONSTANT, SIGNAL, FUNCTION หรือ PROCEDURE คำสั่ง USE จะปิดท้ายด้วยเครื่องหมายอัฒภาค (;)

```
LIBRARY IEEE;                    -- make library symbolic name "IEEE" visible
--
USE ieee.std_logic_1164.and;       -- make the "and" FUNCTION in package
                                   -- std_logic_1164 in library ieee visible
--
ENTITY test IS
end test;
```

รูปที่ 3.7: **การใช้คำสั่ง** LIBRARY **และ** USE

รูปที่ 3.7 คำสั่ง LIBRARY ทำให้ library ชื่อสัญลักษณ์ **IEEE** สามารถ visible สำหรับการออกแบบ คำสั่ง USE กำหนด package ชื่อ **std_logic_1164** ที่อยู่ภายใน **IEEE** องค์ประกอบภายใน package คือการเรียกโปรแกรมย่อยชื่อ **and** ซึ่งสามารถสรุปได้ว่าชุดคำสั่ง LIBRARY และ USE ทำให้ FUNCTION ภายใน (ชื่อ and) visible สำหรับ entity ชื่อ **test**

ในกรณีที่ผู้ออกแบบมีจุดประสงค์ทุกๆ องค์ประกอบที่ประกาศไว้ภายใน package หนึ่งๆ visible สำหรับการออกแบบ ในภาษา VHDL สามารถใช้คำว่า **ALL** แทนการบอกชื่อเฉพาะขององค์ประกอบ ดังที่แสดงในรูปที่ 3.8

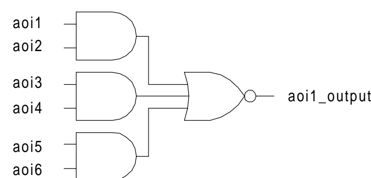
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY test IS
end test;
```

รูปที่ 3.8: **การกำหนดใช้ทุกอย่างภายใต้** package **ชื่อ** STANDARD

คำสั่งดังกล่าวมีความหมายว่าทำให้ทุกๆ สิ่งที่อยู่ใน package ชื่อ **std_logic_1164** ที่อยู่ใน library ที่สัมพันธ์กันด้วยชื่อสัญลักษณ์ (ในที่นี้ ieee) visible สำหรับ entity ชื่อ **test** และส่วนที่เป็นหน่วยรองทั้งหมด

3.8 ตัวอย่างการออกแบบ

หลังจากได้ศึกษาถึงส่วนประกอบของภาษา VHDL ตลอดจนความสัมพันธ์ระหว่างส่วนต่างๆ ของรูปแบบ และการอ้างอิงซึ่งกันและกันแล้ว ในบทนี้จะแสดงให้เห็นรูปแบบของวงจรดิจิตอลอย่างง่ายๆ โดยอยู่ในรูปของรูปแบบที่เขียนด้วยภาษา VHDL เพื่อเปรียบเทียบให้เห็นกับโครงสร้างที่อยู่ในรูปของ schematic ของอุปกรณ์ (gate) ตามรูปที่ 3.9



รูปที่ 3.9: **วงจรดิจิตอลลักษณะของ** schematic diagram

จากรูปที่ 3.9 จะสังเกตได้ทันที่ว่า ทั้งวงจรประกอบด้วยอุปกรณ์พื้นฐานเพียงสองประเภท ได้แก่ **2_input AND-gate** (จำนวน 3 ชิ้น) และ **3_input NOR-gate** (จำนวน 1 ชิ้น) โดยสมมุติว่า อุปกรณ์เหล่านี้ถูกเก็บไว้ในตู้เก็บอุปกรณ์ชื่อ(สัญลักษณ์) **ieee** ลิ้นชักใส่ของชื่อ **std_logic_1164** [5]

การออกแบบโดยวิธีการเขียนรูปแบบ (modeling) ด้วยภาษา VHDL เริ่มต้นด้วยการทำให้ตู้เก็บอุปกรณ์ (สำหรับระบบ VHDL หมายถึง library) ชื่อสัญลักษณ์ **ieee** ให้ visible และบ่งบอกถึงลิ้นชัก (สำหรับระบบ VHDL หมายถึง package) ชื่อ **std_logic_1164** ที่เก็บอุปกรณ์ (component) ทั้งสองโดยชุดคำสั่ง **LIBRARY** และ **USE** ตามที่แสดงในรูปที่ 3.10

```
LIBRARY IEEE;
USE ieee.std_logic_1164.ALL;
ENTITY and2 IS
    PORT (   in1, in2: IN std_logic;
             output: OUT std_logic   );
END and2;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY nor3 IS
    PORT (   in1, in2, in3: IN std_logic;
             output: OUT std_logic   );
END nor3;
```

รูปที่ 3.10: **ส่วนที่บรรยายการติดต่อกับภายนอกของ** gate

หลังจากที่ทำให้ทุกๆ อย่าง (ALL) ที่อยู่ใน package ชื่อ **std_logic_1164** ภายใต้ library ชื่อสัญลักษณ์ ieee สำหรับการออกแบบต่อไป visible แล้ว การเขียนรูปแบบจะเริ่มในระดับล่างสุดของวงจร (gete-level) ซึ่งได้แก่ส่วนที่ติดต่อกับโลกภายนอกของอุปกรณ์ (entity design unit) สิ่งที่ต้องบรรยายต่อมาได้แก่ พฤติกรรมของวงจร หรือความสัมพันธ์ระหว่างสัญญาณ output และ input ซึ่งหมายถึง architecture design unit ตามที่แสดงในรูปที่ 3.11

---

[5] ดูผนวก ค.

```
ARCHITECTURE and2_behave OF and2 IS
BEGIN
    output <= in1 AND in2 AFTER 3 NS;
END and2_behave;

ARCHITECTURE nor3_behave OF nor3 IS
BEGIN
    output <= NOT(in1 OR(in2 OR in3)) AFTER 4 NS;
END nor3_behave;
```

รูปที่ 3.11: **ส่วนที่บรรยายพฤติกรรมของ** gate

สำหรับการบรรยายวงจรสมบูรณ์นั้น จำเป็นที่ต้องกำหนดส่วน entity ใหม่ โดยที่ชื่อของสัญญาณต่างๆ เป็นไปตามวงจร schematic ที่เขียนได้ตามรูปที่ 3.12

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY aoi IS
PORT (   aoi1, aoi2, aoi3, aoi4, aoi5, aoi6: IN std_logic;
         aoi_output: OUT std_logic   );
END aoi;
```

รูปที่ 3.12: **ส่วน** entity **ของวงจร**

เนื่องจากส่วนที่เป็น architecture ของรูปแบบเป็นหน่วยรองของส่วนที่เป็น entity ดังนั้นข้อมูลทั้งหลายที่ visible สำหรับหน่วยหลัก จะสามารถนำไปใช้ได้กับหน่วยรอง

หลังจากที่สร้างรูปแบบในระดับล่างสุดแล้ว (ในตัวอย่างหมายถึงการบรรยายส่วนที่เป็น entity และ architecture ของอุปกรณ์ 2_input AND-gate และ 3_input NOR-gate) ขั้นตอนต่อไปคือการออกแบบในระดับสูงขึ้นอีกชั้น ได้แก่การนำอุปกรณ์ (gate) มาประกอบกันตามโครงสร้างของวงจร ตามที่แสดงในรูปที่ 3.13

```
ARCHITECTURE struct OF aoi IS
    COMPONENT and2
        PORT (   in1, in2: IN std_logic;
                 output: OUT std_logic  );
    END COMPONENT;
    COMPONENT nor3
        PORT (   in1, in2, in3: IN std_logic;
                 output: OUT std_logic  );
    END COMPONENT;
    SIGNAL internal1, internal2, internal3: std_logic;
BEGIN
    u1:and2  PORT MAP (aoi1, aoi2, internal1);
    u2:and2  PORT MAP (aoi3, aoi4, internal2);
    u3:and2  PORT MAP (aoi5, aoi6, internal3);
    u4:nor3   PORT MAP (internal1, internal2, internal3, aoi_output);
END struct;
```

รูปที่ 3.13: **การเชื่อมต่อสัญญาณภายในของวงจร** (architecture)

เนื่องจากการใช้ภาษา VHDL เขียนบรรยายระบบดิจิตอล อนุญาตให้ผู้ออกแบบสามารถเขียนส่วนที่เป็น architecture สำหรับหน่วยหลักได้หลายๆ ลักษณะ ดังนั้นในขั้นตอนของการจำลองการทำงาน (simulation) หรือสังเคราะห์วงจร (circuit synthesis) ผู้ออกแบบจะต้องกำหนดโครงสร้าง (configuration specification) เพื่อเชื่อมหน่วยหลักเข้ากับหน่วยรองที่ต้องการ ซึ่งได้แก่การเขียนส่วนที่เป็น configuration (configuration design unit) ที่เป็นส่วนบนสุด (top level) ของรูปแบบตามที่แสดงในรูปที่ 3.14

```
LIBRARY ieee;
CONFIGURATION top_level OF aoi IS
    FOR struct
        FOR u1: and2 USE ENTITY WORK.and2(and2_behave);
        END FOR;
        FOR u2: and2 USE ENTITY WORK.and2(and2_behave);
        END FOR;
        FOR u3:  and2 USE ENTITY WORK.and2(and2_behave);
        END FOR;
        FOR u4: nor3 USE ENTITY WORK.nor3(nor3_behave);
        END FOR;
    END FOR;
END top_level;
```

รูปที่ 3.14: **โครงสร้างระดับบนสุดของรูปแบบ**

โครงสร้างของ configuration ในรูปที่ 3.14 เป็นโครงสร้างแบบง่าย เป็นเพียงลักษณะหนึ่งที่สามารถสร้างได้ ในส่วนของรายละเอียดจะกล่าวในบทที่ 9

3.9 สรุป

หลังจากที่ได้ศึกษาหน่วยต่างๆ ของ VHDL model ในบทที่แล้ว ในบทนี้ได้บรรยายเพิ่มเติมในเรื่องของความสัมพันธ์ระหว่างหน่วยต่างๆ เหล่านั้น ที่สามารถเชื่อมโยง และอ้างอิงระหว่างกันได้ เพื่อให้ได้ model ที่สมบูรณ์

สิ่งที่แตกต่างกันอยู่ในปัจจุบันคือ การจัดระบบข้อมูลใน design file เพราะทาง IEEE 1076 ไม่ได้มีการกำหนดวิธีการไว้ แต่โดยทั่วๆ ไปแล้วระบบที่พัฒนา VHDL จะยึดหลักเดียวกับการจัดการในระบบ UNIX เพราะต้นกำเนิดของ VHDL นั้นได้รับการพัฒนามาบนเครื่องระบบดังกล่าว

ในตอนท้ายของบทได้แสดงให้เห็นตัวอย่างการเขียนรูปแบบ โดยเปรียบเทียบกับโครงสร้างของ schematic ของวงจรดิจิตอล ดังนั้นถึงจุดนี้ผู้อ่านควรจะมีความเข้าใจ ในโครงสร้างของรูปแบบใช้ภาษา VHDL บรรยายระบบดิจิตอล ตลอดจนความสัมพันธ์และการจัดการข้อมูลต่างๆ เป็นอย่างดี

บทที่ 4
การออกแบบในลักษณะโครงสร้าง และการบรรยายพฤติกรรม
(Structural Design & Behavioral Design)

4.1 กล่าวนำ

ภาษา VHDL เป็นเครื่องมือสำหรับช่วยสร้างรูปแบบ (model) ของระบบดิจิตอลที่ซับซ้อน โดยอาศัยขบวนการของ Top-Down Design ขบวนการดังกล่าวคือการบรรยายระบบดิจิตอลในระดับบนสุด (top-level) ในรูปของแนวความคิดฟังก์ชันการทำงานอย่างสังเขป (abstract) เขียนรูปแบบและจำลองการทำงาน เพื่อตรวจสอบความถูกต้อง หลังจากที่ผ่านการตรวจสอบแล้ว แนวความคิดอย่างสังเขปนี้ จะถูกแบ่ง (partition) ให้เป็นส่วนย่อยๆ และลำดับชั้น (hierarchical) ตามกลุ่มของฟังก์ชันการทำงาน และเช่นเดียวกันส่วนย่อยๆ ที่สร้างขึ้นเหล่านั้น จะถูกจำลองการทำงาน (simulation) ตรวจสอบความถูกต้อง (test and verification) เป็นเช่นนี้ไปเรื่อยๆ ของวงรอบการทำงาน ระดับสุดท้ายคือระดับล่างสุด (gate-level) สามารถที่จะนำไปเปรียบเทียบกับอุปกรณ์ digital hardware ต่างๆ ได้ อาทิเช่น microprocessors, RAM, ROM, PLD และ FPGA โดยผ่านขั้นตอนของการสังเคราะห์วงจร (circuit synthesis)

ฉนั้นการเขียนรูปแบบในลักษณะของ structural description จึงเป็นการบรรยายที่แสดงให้เห็นโครงสร้างของระบบในรูปของอุปกรณ์ต่างๆ และการเชื่อมต่อสัญญาณระหว่างกัน อุปกรณ์แต่ละตัวอาจจะถูกบรรยายพฤติกรรมในลักษณะ behavioral description สำหรับการทำงานของตัวเอง หรืออาจจะบรรยายด้วยอุปกรณ์ระดับล่างลงไปอีกเช่น gates หรือ transistor เป็นต้น หลักการง่ายๆ ที่จะทำให้เข้าใจลักษณะการบรรยายในลักษณะของ behavioral และ structural description คือ behavioral description จะเป็นตัวอย่างที่ดีของรูปแบบอย่างสังเขป (abstract model) รูปแบบลักษณะนี้จะไม่ชี้ให้เห็นชัดว่า วงจรจะมีรูปร่างและโครงสร้างเป็นอย่างไร ส่วนรูปแบบลักษณะ structural description นั้นจะเป็นรูปแบบที่สามารถมองเห็นรูปร่างของวงจรได้ชัดเจน เช่นวงจรประกอบด้วยอุปกรณ์อะไรบ้าง และแต่ละประเภทมีจำนวนเท่าไร มีการเชื่อมต่อกัน และลำดับชั้นอย่างไร

ในบทนี้จะเป็นการขยายความของหลักการ behavioral description และ structural description พร้อมกับแสดงให้เห็นว่า โครงสร้างทั้งสองมีความสำคัญอย่างไรกับวิธีการออกแบบในลักษณะของ Top-Down Design

4.2 Behavioral Design

การเขียนรูปแบบลักษณะของ behavioral description ของระบบดิจิตอลด้วยภาษา VHDL นั้น[1] ถูกจัดให้อยู่ในประเภทของการบรรยายที่ไม่ต้องมีการอ้างถึงรูปแบบย่อย (submodel) ภายใน architecture นั้นอีก ทั้งนี้จะไม่รวมถึงการเรียกโปรแกรมย่อย (subprogram) ที่สามารถเกิดขึ้นได้เสมอในรูปแบบลักษณะนี้ แต่จะหมายถึงการอ้างถึงอุปกรณ์อื่นที่ถูกกำหนดด้วย VHDL ก่อนแล้ว

ลักษณะของ behavioral description โดยทั่วไปแล้วจะเป็นขั้นตอนของการบรรยายที่จะกำหนดฟังก์ชันการทำงานของแบบ ดังในรูปที่ 4.1 ที่แสดงให้เห็น behavioral description ของอุปกรณ์ที่ใช้สำหรับการคูณและหาผลรวม (multiply accumulate device) ซึ่งในที่นี้มีชื่อว่า ***mac***

ในที่นี้จะไม่กล่าวถึงกฎเกณฑ์ (syntax) ของภาษา VHDL ในรูปที่ 4.1 แต่สิ่งที่สำคัญขณะนี้คือความเข้าใจการทำงานของฟังก์ชันการคูณและหาผลรวมของตัวแปรสองตัว สำหรับกฎเกณฑ์ที่ถูกต้องนั้นจะกล่าวในบทต่อไปอย่างละเอียด

ส่วนที่กำหนดการติดต่อกับโลกภายนอกคือ entity declaration (คำสั่ง ENTITY) มีชื่อว่า ***mac*** มีช่องทางเข้าสู่ภายใน (input port) 4 ช่อง ได้แก่ ***in1***, ***in2***, ***clk*** และ ***reset*** และมีช่องทางออก 1 ช่องคือ ***out1*** ช่องทางเข้า ***clk*** และ ***reset*** สามารถรับประเภทของสัญญาณ (และเพื่อความเป็นมาตรฐานตาม IEEE 1076 ต่อไปนี้ประเภทของสัญญาณจะใช้คำว่า TYPE) ประเภท BIT (BIT เป็น standard TYPE ของ IEEE 1076) ได้เท่านั้น ซึ่ง TYPE ชนิดนี้จะมีค่าของสัญญาณอันใดอันหนึ่งใน 2 ค่าได้เท่านั้นอันได้แก่ **'0'** หรือ **'1'** ส่วนช่อง ***in1***, ***in2*** และ ***out1*** มี TYPE เป็น BIT_VECTOR ซึ่งก็คือ array ที่มีขนาด 1 มิติ (one dimension array บางที่เรียกว่า vector) และองค์ประกอบ (element) ทุกตัวภายในมี TYPE เป็น BIT ความกว้างของ array ถูกกำหนดอยู่ในส่วนของการกำหนดช่อง

---

[1] บางครั้งเรียกว่า Algorithm Description

เข้า-ออก (port statement) ในที่นี้สัญญาณ ***in1*** และ ***in2*** มีขนาด 16 bits ในขณะที่ ***out1*** มีขนาด 32 bits สัญญาณ input และ output เช่นนี้สามารถที่จะเรียกว่า bus ได้ เช่น bus ขนาด 16 และ 32 bits ตามลำดับ

```
1.  USE WORK.util.ALL;
2.  ENTITY mac IS
3.    GENERIC (tco: time:= 10 NS);
4.    PORT (  in1,in2: IN BIT_VECTOR(15 DOWNTO 0);
5.            clk,reset: IN BIT;
6.            out1: OUT BIT_VECTOR(31 DOWNTO 0) );
7.  END mac;
8.  ARCHITECTURE behave OF mac IS
9.  BEGIN
10.   PROCESS (clk,reset)
11.     VARIABLE reg_in1, reg_in2, reg_mul, accum: INTEGER;
12.   BEGIN
13.     IF reset = '0' THEN
14.     reg_in1 := 0;
15.     reg_in2     := 0;
16.     reg_mul     := 0;
17.     accum       := 0;
18.     ELSIF rising_edge (clk) THEN
19.     accum       := accum + reg_mul;
20.     reg_mul         := reg_in1 + reg_in2;
21.     reg_in1     := vect_to_int (in1);
22.     reg_in2     := vect_to_int (in2);
23.     END IF;
24.     out1 <= int_to_vec (accum, 32) AFTER tco;
25.   END PROCESS;
26. END behave;
```

รูปที่ 4.1: **รูปแบบ** VHDL **ในลักษณะ** behavioral description **ของอุปกรณ์ที่ใช้สำหรับคูณและหาผลรวม** (Multiply-Accumulate Device) [2]

โดยการใช้คำสั่ง GENERIC ในส่วนที่ใช้ประกาศกำหนดการติดต่อกับภายนอก (entity declaration) นั้น ทำให้สามารถเพิ่มเติม แก้ไข และดัดแปลงค่าพารามิเตอร์ โดยไม่ต้องแก้ไขภายในส่วนของ architecture ในที่นี้ได้แก่พารามิเตอร์เวลา (tco) ซึ่ง ***tco*** จะเป็น propagation delay time ระหว่างช่วงเวลาเริ่มกระตุ้นสัญญาณทาง input ถึงเวลาที่ระบบตอบสนองของสัญญาณ output (รูปที่ 4.1 บรรทัดที่ 24) ค่าหน่วงเวลาของ ***tco*** นี้เป็นค่าตายตัว (default value) และสามารถที่จะเขียนทับ (override) ด้วยค่าใหม่ได้ [3]

---

[2] ตัวเลขบรรทัดแสดงลำดับที่ ไม่มีส่วนเกี่ยวข้องกับกฎเกณฑ์การเขียนรูปแบบ

[3] "VHDL" Navabi Zainalabedin, หน้า 99 - 109, McGraw-Hill, Inc, 1993

ก่อนอื่นจะทำความเข้าใจถึงชุดคำสั่งที่อยู่เหนือ entity declaration ได้แก่คำสั่ง USE คำสั่งนี้มีหน้าที่ทำให้ TYPE หรือ subprogram ตลอดจน utility ต่างๆ ที่เป็นประโยชน์ต่อการเขียนรูปแบบที่เก็บไว้ใน package ชื่อ util สามารถถูกเรียกใช้ได้ในส่วนของ entity declaration และส่วนที่อยู่ภายใต้ อันได้แก่ architecture design unit (use statement อยู่ก่อน entity declaration และ architecture declaration) หรือสามารถพูดได้อีกอย่างหนึ่งว่า ***ทำให้ทุกๆ สิ่ง (ALL) ที่กำหนดขึ้นที่อยู่ใน PACKAGE ที่มีชื่อว่า util และอยู่ภายใต้ LIBRARY ชื่อสัญลักษณ์ WORK สามารถถูกมองเห็น (visibility) ได้โดย entity- และ architecture design unit นี้***

ส่วนที่เป็น architecture ของรูปแบบ มีชื่อว่า *behave* เป็นส่วนที่ถูกจัดให้อยู่ภายใต้ entity ชื่อ *mac* คำสั่งแรกภายใต้ architecture declaration ได้แก่คำสั่ง PROCESS ซึ่งเป็น concurrent statement แต่การบรรยายทั้งหมด (statement ต่างๆ) ที่อยู่ภายใต้ process statement (process statement เริ่มต้นด้วยคำ PROCESS และปิดท้ายด้วยคำ END PROCESS;) นั้นเป็นการบรรยายแบบลำดับ (sequential[4]) นั้นคือจะประกอบด้วย sequential statement เท่านั้น สัญญาณ ***clk*** และ ***reset*** เป็นสัญญาณใน sensitivity list ของ process statement สัญญาณทั้งสองทำหน้าที่เป็นตัวควบคุมการทำงานของ process statement โดยที่เมื่อเกิดการเปลี่ยนแปลงระดับค่าของสัญญาณ (event) ของสัญญาณตัวใดตัวหนึ่งใน sensitivity list จะมีผลทำให้ process statement ถูกกระตุ้นให้ทำงานและคำสั่งทุกๆ คำสั่งภายในจะทำงานตามลำดับ (sequentially) เช่นในกรณีที่สัญญาณมี TYPE เป็น BIT การเกิด event ในตัวสัญญาณคือการเปลี่ยนจากค่าเดิมที่เป็น '0' เป็นค่าใหม่ '1' หรือในทางตรงข้ามเป็นต้น

หลังจากที่ process ถูกกระตุ้น ลำดับของชุดคำสั่ง IF-THEN-ELSE จะทำงาน คำสั่ง IF จะตรวจสอบค่าของสัญญาณ *reset* ว่าเป็น '0' หรือเปล่า ถ้าเป็น '0' จะทำให้ boolean expression (reset = '0') ในส่วนของข้อแม้ IF clause มีผลลัพธ์เป็น TRUE (boolean expression สามารถให้ผลลัพธ์ที่มี TYPE เป็นประเภท boolean ได้เพียงสองอย่างคือ **TRUE** กับ **FALSE**) ตัวแปร (variable) ภายในทั้งหมด (*reg_in1, reg_in2, reg_mul และ accum*) จะถูกกำหนดค่าเป็น '0' แต่ถ้าสัญญาณ *reset* ไม่เท่ากับ '0' (reset = '1') ทำให้ boolean expression (reset = '0') ในส่วนของข้อแม้ IF clause มีผลลัพธ์เป็น FALSE คำสั่งในลักษณะของการตรวจสอบ ELSIF (ในภาษา VHDL ใช้คำสั่ง ELSIF **ไม่ใช่** ELSEIF หรือ ELSE IF) จะทำงาน เพื่อสอบถามว่าสัญญาณ ***clk*** เกิดการเปลี่ยนค่าของสัญญาณจาก

---

[4] ชุดคำสั่งลำดับ (sequential statement)

'0' เป็น '1' (ในขณะเวลาปัจจุบันของการจำลองวงจร) หรือเปล่า ในที่นี้จะมีการเรียกโปรแกรมย่อย[5] (subprogram) ประเภท FUNCTION (subprogram ในภาษา VHDL มีสองประเภทคือ FUNCTION และ PROCEDURE รายละเอียดจะกล่าวในบทต่อไป) ชื่อ rising_edge ที่ถูกเก็บไว้ใน package ชื่อ util (นี่คือเหตุผลที่ว่าทำไมถึงต้องกำหนด use statement บรรทัดที่ 1 ในรูปที่ 4.1) โปรแกรมย่อยนี้จะตรวจสอบการเปลี่ยนระดับค่าจาก '0' เป็น '1' ของสัญญาณ ***clk*** ถ้ามีเกิดขึ้น โปรแกรมย่อยจะให้ผลลัพธ์ที่เป็นค่า boolean คือ TRUE ทำให้ข้อแม้ ELSIF เป็นจริง คำสั่งทั้งสี่ที่อยู่ทัดลงมาจะทำงานตามลำดับ ซึ่งก็คือการทำขั้นตอนของการคูณและหาผลรวม (multiply accumulate) นั้นเอง ในที่นี่โปรแกรมย่อยประเภท FUNCTION ชื่อ **vect_to_int** ทำหน้าที่แปลงค่าของสัญญาณ ที่มี TYPE ประเภท BIT_VECTOR ของสัญญาณ ***in1*** และ ***in2*** เป็น INTEGER (เลขจำนวนเต็มบวก) ในที่นี้คือการเปลี่ยนเลข binary ขนาด 16 bits ให้เป็นเลข integer เพื่อที่จะสามารถใช้ arithmetic operator ได้

บรรทัดสุดท้ายของ sequential statement (บรรทัดที่ 24) คือการกำหนดค่าของ output ให้กับช่องทางออก ***out1*** โดยที่สัญญาณมีการหน่วงเวลาเท่ากับ ***tco*** (AFTER clause) โปรแกรมย่อยประเภท FUNCTION ชื่อ **int_to_vect** ทำหน้าที่แปลงค่าของสัญญาณ ที่มี TYPE ประเภท INTEGER ของตัวแปร ***accum*** ให้เป็นเลข binary ขนาด 32 bits

จากตัวอย่างในรูปที่ 4.1 จะเห็นได้ว่ารูปแบบ VHDL ที่ใช้บรรยาย multiply accumulate function มีลักษณะคล้ายกับการเขียนโปรแกรมด้วยภาษา C หรือ PASCAL มากกว่าที่จะเป็นโครงสร้างของ hardware ฉนั้นด้วยเหตุผลนี้เอง จึงไม่สามารถที่จะหาความสัมพันธ์ของการบรรยายในรูปของฟังก์ชันการทำงาน กับโครงสร้างทางฟิสิกส์ในรูปของอุปกรณ์ดิจิตอลได้ รูปแบบ VHDL เช่นนี้จึงจัดให้อยู่ในประเภท **behavioral**

จากความจริงอย่างหนึ่งที่เห็นในตัวอย่างคือ ข้อดีของการบรรยายแบบ behavioral ที่ว่าวิศวกรออกแบบสามารถที่จะสร้างรูปแบบของระบบดิจิตอล โดยไม่ต้องคำนึงถึงรายละเอียดของการสร้างวงจรจริง จึงสามารถที่จะเขียนรูปแบบ ให้เข้าสู่จุดประสงค์ของงาน (specification) โดยไม่ต้องเบี่ยงแบนความคิด ว่าจะใช้อุปกรณ์อะไรมาสร้างให้ได้ฟังก์ชันตามต้องการ

---

[5] โปรแกรมย่อย (subprogram) กล่าวในบทที่ 8

4.3 Structural Design

การบรรยายในลักษณะโครงสร้าง (structural description) ด้วยภาษา VHDL จัดอยู่ในประเภทการแสดงด้วยการแทนที่โดยอุปกรณ์ (ในที่นี้หมายถึงอุปกรณ์ที่อยู่ในรูปแบบของ VHDL คือ entity และ architecture design unit) และการเชื่อมต่อภายในระหว่างอุปกรณ์เหล่านั้น (instantiation and interconnection) ด้วยโครงสร้าง VHDL (VHDL component) การบรรยายเช่นนี้จะอยู่ในรูปที่เรียกว่า VHDL netlist ดังตัวอย่างของ architecture description ของ multiply and accumulate function ที่แสดงในรูปที่ 4.2

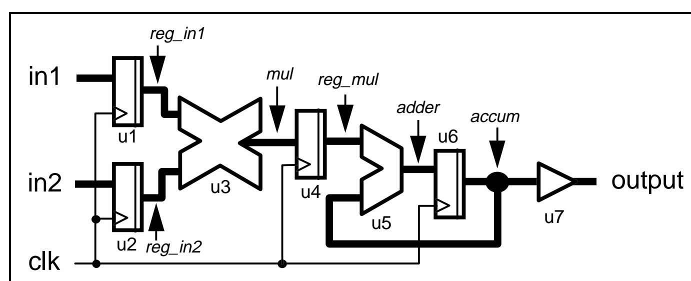
จากตัวอย่างในรูปที่ 4.2 ในส่วนของ architecture declaration จะสังเกตได้ว่ามีการประกาศกำหนดอุปกรณ์ด้วย component statement (กลุ่มคำสั่ง COMPONENT … END COMPONENT;) ภายในโครงสร้าง และสัญญาณภายใน (local signal) อีก 6 สัญญาณ (คำสั่ง SIGNAL) ที่ใช้สำหรับเชื่อมต่อระหว่างอุปกรณ์ต่างๆ ภายในตัวของโครงสร้างเอง

ส่วนที่ใช้บรรยาย architecture เองนั้นจะใช้อุปกรณ์ (component) เท่าที่ได้ประกาศกำหนดไว้แล้ว (เรียกว่า component declaration) และการเชื่อมต่อกันนั้น จะใช้ชุดคำสั่ง PORT MAP ในส่วนนี้เองจะเป็นการเขียนในรูปของ VHDL netlist ซึ่งสามารถที่จะมองเปรียบเทียบกับโครงสร้างที่เป็นรูปภาพได้ตามรูปที่ 4.3

ตัวอย่างในรูปที่ 4.1 และ 4.2 แสดงให้เห็นถึงข้อแตกต่างในระดับของการออกแบบคือระหว่าง behavioral level (รูปที่ 4.1) กับ structural level (รูปที่ 4.2) วิศวกรออกแบบที่ประสงค์จะเขียนรูปแบบของระบบดิจิตอลด้วยภาษา VHDL ไม่ควรตั้งข้อสมมุติ ว่าจะเขียนรูปแบบในลักษณะหนึ่งลักษณะใด โดยเหตุที่ว่าสมมุติฐานเหล่านั้นจะหมดไปตามข้อกำหนด และข้อบังคับของภาษาเอง เพราะภาษา VHDL สนับสนุนแนวทางขบวนการของการแบ่งและปฏิบัติให้ได้ผล (divide and conquer) โดยการแยกแบบออกเป็นส่วนๆ (design partition) แล้วพัฒนารายละเอียดในแต่ละกลุ่มและสร้างรูปแบบลักษณะโครงสร้าง (structural) เป็นชิ้นสมบูรณ์ที่ละชั้น (hierarchical)

```
1.  ARCHITECTURE structure OF mac IS
2.    COMPONENT reg
3.    GENERIC (width: INTEGER:= 16);
4.    PORT ( d: IN BIT_VECTOR ((width - 1) DOWNTO 0);
5.             clk: IN BIT;
6.             q: OUT BIT_VECTOR ((width - 1) DOWNTO 0)  );
7.    END COMPONENT;
8.    COMPONENT adder
9.    PORT (  port1, port2: IN BIT_VECTOR (31 DOWNTO 0);
10.            output: OUT BIT_VECTOR (31 DOWNTO 0)  );
11.   END COMPONENT;
12.   COMPONENT multiply
13.   PORT (  port1, port2: IN BIT_VECTOR (15 DOWNTO 0);
14.            output: OUT BIT_VECTOR (31 DOWNTO 0)  );
15.   END COMPONENT;
16.   COMPONENT buf
17.   PORT (  input: IN BIT_VECTOR (31 DOWNTO 0);
18.            output: OUT BIT_VECTOR (31 DOWNTO 0) );
19.   END COMPONENT;
20.   SIGNAL   reg_in1, reg_in2: BIT_VECTOR (15 DOWNTO 0);
21.   SIGNAL   mul, mul_reg, adder, accum: BIT_VECTOR (31 DOWNTO 0);
22. BEGIN
23.   u1: reg GENERIC MAP (16) PORT MAP (in1, clk, reg_in1);
24.   u2: reg GENERIC MAP (16) PORT MAP (in2, clk, reg_in2);
25.   u3: multiply PORT MAP (reg_in1, reg_in2, mul);
26.   u4: reg GENERIC MAP (32) PORT MAP (mul, clk, reg_mul);
27.   u5: adder PORT MAP (reg_mul, accum, adder);
28.   u6: reg GENERIC MAP (32) PORT MAP (adder, clk, accum);
29.   u7: buf PORT MAP (accum, out1);
30. END structure;
```

รูปที่ 4.2: โครงสร้าง (structure) ของวงจร Multiply-Accumulate Unit



รูปที่ 4.3: โครงสร้างอุปกรณ์ของวงจร Multiply-Accumulate Unit

4.4 Mixed Level Modeling

ด้วยเหตุผลที่ชุดคำสั่งแบบแข่งขนาน[6] (concurrent statement) สามารถที่จะเขียนลงในตำแหน่งใดๆ ภายในส่วนของ architecture ได้นี่เอง ภาษา VHDL จึงเปิดโอกาสให้วิศวกรออกแบบค่อยๆ แปลงรูปแบบจาก behavioral model ไปสู่ structural model ได้

จนกระทั่งถึงจุดนี้ยังไม่ได้มีการศึกษาถึงรายละเอียดของ concurrent statement แต่จากตัวอย่างที่ผ่านมา ได้มีการนำ concurrent statement มาใช้บ้างแล้ว concurrent statement สามารถมองให้เป็นส่วนย่อยของแบบ (subdesign) ที่เป็นอิสระในการทำงาน และจากการที่ concurrent statement ทั้งหลายเป็นอิสระต่อกัน ฉนั้นภายใน architecture เดียวกันสามารถที่จะมีชุดคำสั่งประเภท concurrent statement ได้หลายๆ ชุด

ต่อไปจะมาศึกษา concurrent statement ทั้งสามที่ได้เคยนำมาใช้ในตัวอย่าง โดยที่ชนิดแรกได้แก่

- *Signal assignment statement:*

```
C <= a AND b AFTER 10 NS;
```

- *Process statement:*

```
PROCESS(signal1, signal2,...)
.
.
END PROCESS;
```

- *Component instantiation statement:*

```
u2: and2
      GENERIC MAP (2 NS, 1 NS);
      PORT MAP (in1, in2, in3);
```

ทั้งสาม concurrent statement นี้แสดงให้เห็นหลักการของการเขียนรูปแบบ (model) ในลักษณะของ mixed level modeling

[6] ชุดคำสั่งแบบแข่งขนาน (concurrent statement) กล่าวในบทที่ 7

เพื่อให้เห็นตัวอย่างของ mixed level modeling จึงของยกตัวอย่างของ **multiply accumulate function** มาแสดงให้เห็นอีกครั้งในรูปที่ 4.4

```
1.  USE WORK.util.ALL;
2.  ARCHITECTURE mixed OF mac IS
3.    COMPONENT reg
4.    GENERIC (width: INTEGER:= 16);
5.    PORT (  D: IN BIT_VECTOR (width - 1 DOWNTO 0);
6.            clk: IN BIT;
7.            Q: OUT BIT_VECTOR (width - 1, DOWNTO 0)  );
8.    END COMPONENT;
9.    SIGNAL reg_in1, reg_in2, accum: BIT_VECTOR (15 DOWNTO 0);
10. BEGIN
11. -- structural description
12.   u1: reg    GENERIC MAP (16) PORT MAP (in1, clk, reg_in1);
13.   u2: reg    GENERIC MAP (16) PORT MAP (in2, clk, reg_in2);
14. -- behavioral description
15.   PROCESS
16.     VARIABLE temp1,temp2,mul,adder: INTEGER:= 0;
17.   BEGIN
18.     WAIT UNTIL clk = '1';                    -- rising edge of clk
19.     temp1   := vect_to_int (reg_in1);
20.     temp2   := vect_to_int (reg_in2);
21.     mul     := temp1 * temp2;
22.     adder   := accum + mul;
23.     accum  <= int_to_vect (adder, 32);
24.   END PROCESS;
25. -- dataflow description
26.   out1 <= accum;
27. END mixed;
```

รูปที่ 4.4: **โครงสร้างแบบ** mixed level modeling **ของ** Multiply-Accumulate Unit

4.5 สรุป

ในบทนี้ได้แสดงให้เห็นถึงหลักการของวิธีการเขียนรูปแบบในลักษณะต่างๆ ดังที่จะเห็นได้ว่าภาษา VHDL สนับสนุนการเขียนรูปแบบทั้งลักษณะ behavioral และ structural ในลักษณะของ behavioral modeling นั้นจะเป็นการเขียนรูปแบบของฟังก์ชัน แต่ไม่ได้แสดงรายละเอียดของโครงสร้าง (structure) ของวงจรจริง ข้อดีของการเขียนลักษณะนี้คือ วิศวกรออกแบบสามารถที่จะเขียนบรรยายฟังก์ชันได้ตั้งแต่ระดับบนของแนวความคิดอย่างสังเขป อาทิเช่นสามารถใช้ TYPE สำหรับข้อมูลประเภท INTEGER ได้ ซึ่งในระบบดิจิตอลจริงๆ แล้ว (hardware) ลักษณะของข้อมูลจะเป็น

ระบบ **binary** นอกจากนั้นยังสามารถใช้ภาษา VHDL บรรยายให้รายละเอียดเพิ่มเติม ในระดับล่างลงมา โดยที่การเขียนแบบ structural modeling จะแสดงให้เห็นว่าอุปกรณ์ใด ใช้ทำอะไร และมีการเชื่อมต่อกันอย่างไร เพื่อทำให้เกิดฟังก์ชันที่ต้องการ

สุดท้ายของบทนี้ได้แสดงให้เห็นถึงความอ่อนตัวของภาษา VHDL ที่สามารถใช้เขียนบรรยายรูปแบบของระบบดิจิตอลในลักษณะผสม (mixed modeling) ระหว่าง behavioral และ structural modeling ภายในโครงสร้าง architecture เดียวกัน เพื่อแสดงรายละเอียดของรูปแบบที่เขียนในระดับต่างๆ ซึ่งขั้นตอนที่กล่าวมานี้จะครอบคลุมวิธีการของ Top-Down Design Process ทั้งหมด

บทที่ 5
หลักการเบื้องต้นของการจำลองการทำงานโดย VHDL
(VHDL Simulation Conceptual)

5.1 กล่าวนำ

ในคู่มืออ้างอิงภาษา (LRM) ตีพิมพ์โดยสมาคม IEEE เพื่อกำหนดมาตรฐานและกฎเกณฑ์ของภาษา VHDL ซึ่งรวมการวิเคราะห์และความสัมพันธ์ระหว่างส่วนต่างๆ ของรูปแบบภายใต้สภาพแวดล้อมของการออกแบบเดียวกัน ตลอดจนการเพิ่มความสมบูรณ์ของรูปแบบ การเตรียมค่าเริ่มต้น และการทำงานของวงรอบจำลองการทำงาน จากมาตรฐานที่กำหนดนี้เองที่ช่วยทำให้ภาษา VHDL สามารถนำไปใช้ได้กับระบบพัฒนา VHDL จากผู้ผลิต (vendor) อื่นๆ ได้ ซึ่งเรียกได้ว่ามีความเป็นอิสระต่อระบบ (vendor independence) ซึ่งทำให้ผู้ออกแบบสามารถเลือกระบบออกแบบได้หลายแบบ ในบทนี้จะเป็นการศึกษาลักษณะของการจำลองการทำงานของภาษา VHDL ที่เป็นหลักการของระบบพัฒนา VHDL ซึ่งจะช่วยทำให้ผู้อ่านเกิดความเข้าใจหลักการของภาษามากขึ้น

5.2 ค่าจำลองเบื้องต้น (Simulation Primitives)

ในระบบจำลองการทำงานในลักษณะของดิจิตอลลอจิกนั้น ผู้ใช้จะคุ้นเคยกับคำว่าค่าจำลองเบื้องต้นหรือ simulation primitive[1] ในระบบดังกล่าวนี้ค่า simulation primitive จะเป็นค่าลอจิกที่คงที่ และมีฟังก์ชันการทำงานที่ถูกกำหนดตายตัวลงไปในแกนหลักของการจำลอง ค่าจำลองเบื้องต้นที่พบกันมากได้แก่ฟังก์ชันของ AND, OR, XOR, DFF, LATCH, BUFFER และ PULL-UP เป็นต้น ในการออกแบบทุกครั้งจะต้องกำหนดวงจรลงสู่ฟังก์ชันเหล่านี้

---

[1] "An Introduction to Digital Simulation", Mentor Graphics Corporation, 1989

ข้อเสียที่สำคัญของระบบจำลองการทำงานในลักษณะของดิจิตอลลอจิกแบบเดิมนี้ คือความจำเป็นที่ต้องกำหนดแบบให้ลงสู่ค่าจำลองเบื้องต้นนี้ เนื่องจากฟังก์ชันเบื้องต้นเหล่านี้ประกอบกันเป็นอุปกรณ์ดิจิตอล และนำไปใช้ในการออกแบบอุปกรณ์ที่มีความซับซ้อนสูง ซึ่งเป็นการบังคับให้ต้องออกแบบในลักษณะที่เรียกว่า Bottom-Up Design โดยที่ผู้ออกแบบจะออกแบบจากพื้นฐานของฟังก์ชันของค่าจำลองเบื้องต้น เพื่อพัฒนาระบบและจำลองการทำงานของระบบดิจิตอล นั่นหมายความว่าการออกแบบลักษณะของ Bottom-Up Design ที่ผู้ออกแบบจะต้องคำนึงถึงการผลิต (implementation) มากกว่าฟังก์ชันการทำงาน (functionality) ของงาน

ในระบบออกแบบด้วย VHDL (Top-Down Design) จะไม่มีสิ่งที่เรียกว่า built-in simulator ค่าจำลองเบื้องต้นที่มีอยู่อย่างแท้จริงระดับล่างสุดของชั้นลำดับ (hierarchical model) ในแบบ จะเป็นตัวสร้างค่าจำลองเบื้องต้น ซึ่งสิ่งที่สำคัญที่สุดที่ต้องจดจำไว้คือ ค่าเบื้องต้นนั้นเป็นค่าที่สามารถกำหนดได้โดยผู้ใช้ระบบ และกำหนดด้วยภาษา VHDL

ผู้ออกแบบสามารถที่จะควบคุมค่าจำลองเบื้องต้นในระบบ VHDL ได้เต็มที่ และเป็นไปได้ที่จะพัฒนารูปแบบในระดับบนสุดของแนวความคิดอย่างสังเขป การออกแบบตามฟังก์ชันการทำงานคือกุญแจสำคัญที่นำไปสู่จุดหมาย และความจริงอีกอย่างหนึ่งที่ว่า จากรูปแบบที่ออกมานั้นไม่จำเป็นต้องเป็นระบบดิจิตอลเสมอไป ซึ่งหลักการนี้คล้ายกับการเขียนภาษาโปรแกรมที่ว่าโปรแกรมถูกเขียนขึ้น แล้วถูกนำไปแปล (compile) และนำไปใช้งาน (execute) ในที่สุด

จากความสามารถในการจำลองการทำงานในลักษณะพิเศษของระบบ VHDL นี่เอง จึงเปิดโอกาสให้วิศวกรออกแบบสามารถที่จะเริ่มต้นทำงานโดยการเริ่มต้นสร้างแนวทางการออกแบบจากแนวความคิดอย่างสังเขปและเริ่มสร้างรูปแบบ พร้อมทั้งจำลองการทำงานของรูปแบบ (แนวความคิด) นั้นได้อย่างรวดเร็ว และค่อยๆ เพิ่มเติมในรายละเอียดที่ละชั้น ถ้าทดลองเปรียบเทียบกับการออกแบบลักษณะ Bottom-Up Design จะเห็นได้ว่าวิศวกรออกแบบจะใช้เวลามากกว่า 90% สำหรับการป้อนแบบวงจรด้วยอุปกรณ์ (schematic capture) จำลองการทำงาน ตรวจสอบความถูกต้อง และงานขีดเขียนแบบวงจร ซึ่งในการนำวงจรเก่ามาออกแบบใหม่ โดยที่มีความต้องการที่จะลดขนาดของวงจรให้เล็กลง และใช้อุปกรณ์ที่มีความเร็วในการทำงานสูงขึ้น หรือใช้อุปกรณ์ที่มีความหนาแน่นของวงจรสูงมากขึ้นเพื่อรวมฟังก์ชันการทำงาน การออกแบบในลักษณะดังกล่าวจะไม่มีเวลาเหลือให้มากสำหรับงานต่างๆ เหล่านั้น ดังนั้นการใช้ภาษา VHDL ในการออกแบบลักษณะของ Top-Down Design วิศวกรออกแบบจะสามารถค้นพบความจริงใหม่ๆ ของศิลป์และศาสตร์แห่งวิศวกรรมของการออกแบบ hardware ระบบดิจิตอล

5.3 ระบบสภาวะ (State System)

การที่ระบบจำลองการทำงานแบบเดิมที่มีค่าเบื้องต้นต่างๆ ที่แน่นอน (fixed values) จะทำให้มีระบบสภาวะที่คงที่และหรือขึ้นอยู่กับระบบที่ใช้จำลองเช่นกัน ระบบสภาวะเหล่านี้ (ปกติจะมี 8 ถึง 12 สภาวะ) ใช้ได้กับการจำลองการทำงานของระบบดิจิตอลที่มีค่าสภาวะแน่นอน (discrete digital system) เช่นเดียวกันกับข้อจำกัดของค่าเบื้องต้น ผู้ออกแบบจะต้องคำนึงถึงขั้นตอนการผลิตทันทีที่เกิดความคิดใหม่ๆ และจะต้องแก้ไขแบบวงจร

ระบบจำลองการทำงานแบบเดิมจะบังคับด้วยข้อจำกัดเหล่านี้ ทำให้ต้องใช้วิธีการออกแบบโดยใช้การป้อนแบบด้วยอุปกรณ์ (schematic capture) วิศวกรออกแบบจะต้องคิดในรูปของ gate (NOT, AND, OR, ...), FlipFlop, Multiplexer และ Register แทนที่จะใช้ความคิดสำหรับหาฟังก์ชันการทำงานของงานที่ได้รับมอบได้อย่างเต็มที่

ด้วยระบบ VHDL ที่ผู้ออกแบบเป็นผู้กำหนดระบบสภาวะ และยังคงสามารถใช้ระบบสภาวะของระบบเดิม หรือกำหนดให้สังเขปมากยิ่งขึ้น ตัวอย่างเช่นถ้าต้องการที่จะเขียนรูปแบบของระบบสี สามารถที่จะกำหนดระบบสภาวะในระดับบนสุดได้ตามที่แสดงในรูปที่ 5.1 เป็นต้น

```
TYPE color IS (red, green, blue, yellow, violet, orange);
```

รูปที่ 5.1: การกำหนดระบบสภาวะสำหรับระบบสี

หรือสามารถที่จะกำหนดระบบสภาวะเพื่อการจำลองการทำงานของระบบดิจิตอลตามรูปที่ 5.2

```
TYPE std_ulogic IS ('U', '0', '1', 'Z', 'X', 'W', 'L', 'H', '-');
```

รูปที่ 5.2: การกำหนดระบบสภาวะสำหรับระบบดิจิตอล

ตามที่ได้ศึกษาในหัวข้อระบบสภาวะของภาษา VHDL และการกำหนดค่าโดยใช้คำสั่ง **TYPE** ไปแล้ว โดยที่ความหมายในภาษา VHDL คำว่า TYPE คือกลุ่มของค่า (set of value) กลุ่มของค่าเหล่านี้จะเป็นตัวบอกค่าที่สามารถเป็นไปได้ของ SIGNAL, VARIABLE และ CONSTANT หรือที่เรียกรวมว่า object เมื่อ object ถูกประกาศขึ้น กลุ่มของค่า (TYPE) ก็จะถูกกำหนดขึ้นด้วย ระบบจำลองการทำงานจะไม่อนุญาตให้ค่าที่ไม่ได้ถูกกำหนดไว้ในกลุ่มของค่า (set of value) ไว้สำหรับ object นั้น กำหนด (assignment) ให้กับ object ได้

ตัวอย่าง:

```
SIGNAL clock : std_ulogic;
VARIABLE crayon : color := yellow;
CONSTANT clear : std_ulogic := '0';
```

ในแต่ละกรณีเมื่อ object ถูกประกาศขึ้นจะมีความสัมพันธ์กับ TYPE ด้วย เช่นในกรณีของ crayon ที่ประกาศเป็นตัวแปร (คำสั่ง VARIABLE) มีการกำหนดค่าเริ่มต้นได้แก่ yellow (ซึ่งเป็นองค์ประกอบในกลุ่มของค่าประเภท color ตามรูปที่ 5.1) และตัวคงที่ clear มีค่าเท่ากับ '0' ใน TYPE ชนิด std_ulogic

หลังจากได้รู้จักกับกฎการเขียนกำหนดประเภท (TYPE) ในภาษา VHDL แล้ว ขั้นตอนต่อไปจะเป็นการศึกษาความหมายในการใช้งาน TYPE โดยทั่วไปแล้วจะประกาศอยู่ใน package ซึ่งจะเปิดโอกาสให้นำไปใช้ได้ในการออกแบบ (เขียนรูปแบบ) ที่มีการอ้างอิงถึง package นั้นๆ การประกาศ TYPE ใน package มีโครงสร้างโดยทั่วไปดังที่แสดงในรูปที่ 5.3

```
PACKAGE std_logic_1164 IS
    TYPE std_ulogic IS ('U', '0', '1', 'Z', 'X', 'W', 'L', 'H', '-');
END std_logic_1164;
--
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
--
ENTITY mux IS
    GENERIC (  tplh: TIME := 3 NS; tphl: TIME := 5 NS;
               sel_out: TIME := 3 NS  );
    PORT (   in1, in2, sel: IN std_ulogic;
             output: OUT std_ulogic   );
END mux;
```

รูปที่ 5.3: **การประกาศ** TYPE **ใน** package **และการอ้างอิง**

การประกาศกำหนด TYPE ชื่อ std_ulogic ทำใน package ชื่อ std_logic_1164 ทุกๆ ส่วนของรูปแบบที่อ้างอิง package นี้ สามารถใช้ std_ulogic ได้ เช่นตัวอย่างของส่วนที่เป็น entity ในรูปที่ 5.3 ด้วยคำสั่ง LIBRARY และ USE เป็นการเปิด (making visible) package ชื่อ std_logic_1164 ฉนั้น object ต่างๆ ที่ประกาศในส่วนของ entity สามารถใช้ค่าทั้งหมดของ TYPE ที่ชื่อ std_ulogic ได้

โดยปกติแล้วผู้เขียนรูปแบบสามารถกำหนด TYPE ได้ในส่วนประกาศ (declarative area) ของส่วนต่างๆ ได้ ตัวอย่างเช่นการกำหนด TYPE ชนิด std_ulogic ในส่วนที่เป็นเนื้อที่ประกาศของ entity ตามที่แสดงในรูปที่ 5.4

```
ENTITY mux IS
    TYPE std_ulogic IS ('U', '0', '1', 'Z', 'X', 'W', 'L', 'H', '-');
    GENERIC (  tplh: TIME := 3 NS; tphl: TIME := 5 NS;
                    sel_out: TIME := 3 NS  );
    PORT (   in1, in2, sel: IN std_ulogic;
                output: OUT std_ulogic   );
END mux;
```

รูปที่ 5.4: **การประกาศ** TYPE **ใน** entity design unit

จากตัวอย่างในรูปที่ 5.3 และ 5.4 การเขียนส่วนที่เป็น entity เหมือนกัน สิ่งที่แตกต่างกันได้แก่ TYPE ชนิด std_ulogic โดยที่ในรูปที่ 5.3 นั้นสามารถที่จะอ้างอิงได้โดยรวม (global) ในขณะที่ในรูปที่ 5.4 นั้นสามารถอ้างอิงได้เฉพาะใน entity ชื่อ mux และส่วนที่เป็น architecture และส่วนรอง (secondary unit) ของ entity เท่านั้น

เป็นที่เข้าใจอย่างชัดเจนว่าสิ่งที่ประกาศในส่วนที่เป็น entity นั้น (PORT และ GENERIC) ได้แก่สิ่งที่เรียกว่า object ที่สามารถมีชั้น (class) ได้สามประเภทคือ SIGNAL, VARIABLE และ CONSTANT สิ่งที่ประกาศด้วยชุดคำสั่ง PORT จะเป็น SIGNAL และสิ่งที่ประกาศด้วยชุดคำสั่ง GENERIC จะเป็น CONSTANT

5.4 ลำดับเวลาของสัญญาณและเวลา δ (Signal Queues & Delta Times)

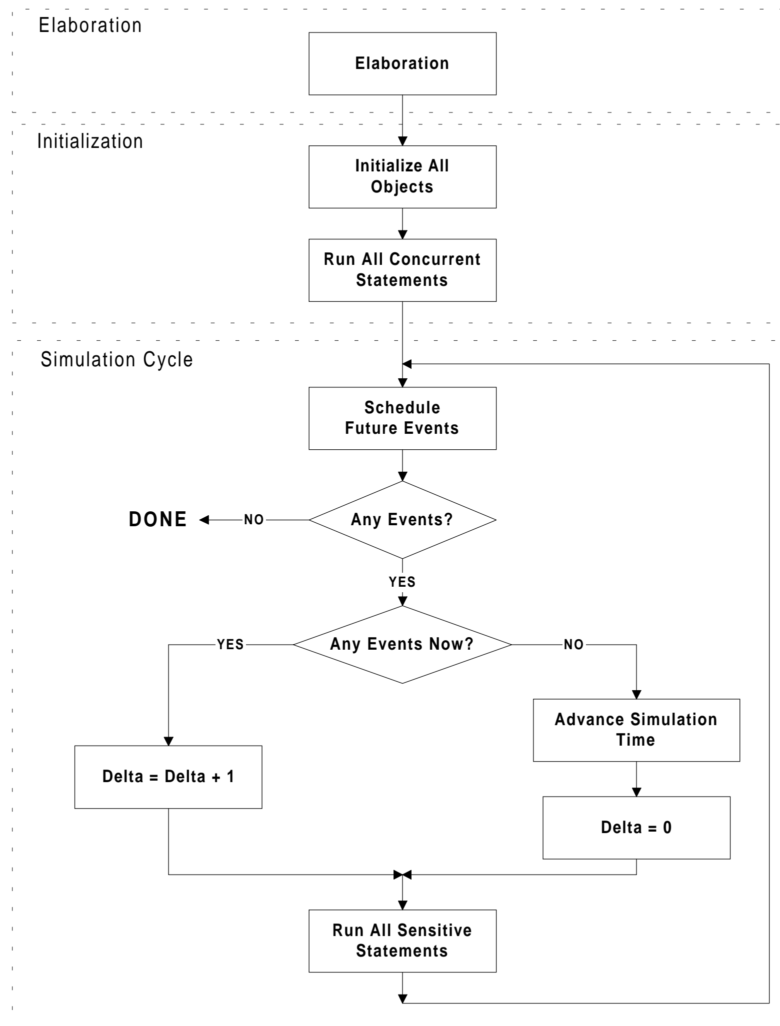
การจำลองการทำงานในลักษณะของ VHDL นั้นเป็นการทำงานตามการเปลี่ยนแปลงระดับค่าของสัญญาณ (event driven simulation) คือระบบจะตรวจสอบและมีปฏิกิริยาต่อการเปลี่ยนแปลงภายใต้กรอบบังคับที่กำหนด กรอบบังคับดังกล่าวในระบบของ VHDL คือสิ่งที่เกิดการเปลี่ยนแปลงอันได้แก่สัญญาณ (SIGNAL) และปฏิกิริยาต่อเนื่องที่เกิดจากการเปลี่ยนแปลงครั้งแรกของสัญญาณที่บรรยายในชุดคำสั่งแบบแข่งขนาน (concurrent statement) ที่มีความไว (sensitive) ต่อสัญญาณนั้นๆ

ในวงรอบการจำลองการทำงานของระบบ VHDL สัญญาณจะถูกปรับปรุงค่าใหม่ทุกครั้งที่มีการตรวจพบการเปลี่ยนแปลงระดับค่า (event) ชุดคำสั่งใด (concurrent) ที่ไวต่อการเปลี่ยนแปลงนั้น (สามารถมีได้มากกว่า 1 ชุดคำสั่ง) จะจัดลำดับการทำงาน ชุดคำสั่งหลายๆ ชุดสามารถเข้าลำดับและทำงานได้ ชุดคำสั่งเหล่านี้อาจเป็นสาเหตุที่ทำให้เกิดการเปลี่ยนแปลงเพิ่มเติมขึ้นอีก โดยที่ไม่มีการทำให้เวลาของการจำลองการทำงาน (simulation time) เพิ่มขึ้น (เกิดขึ้นในเวลาของการจำลองเดียวกัน) ดังนั้นภายในหนึ่งวงรอบการทำงานของการจำลอง ทุกๆ ชุดคำสั่งที่อยู่ในลำดับ จะถูกปฏิบัติ ปรับปรุงค่าของสัญญาณใหม่ ตรวจสอบการเปลี่ยนแปลงถ้ามี จะสร้างลำดับการทำงานขึ้นมาใหม่ (โดยที่ยังคงอยู่ในเวลาของการจำลองการทำงานเดิม)

การที่การจำลองในระบบ VHDL สามารถที่จะทำงานในลักษณะดังกล่าวได้ โดยที่ไม่มีการเพิ่มเวลาจำลองทำงานนั้น กลไกสำคัญที่ช่วยทำให้การทำงานโดยปราศจากข้อแม้ทางเวลานี้เรียกว่า **delta time (δ-time)** หรือ internal delay

เวลาของการจำลองการทำงานในระบบ VHDL สามารถที่จะมองให้อยู่ในภาพระนาบที่มี 2 มิติได้ โดยที่มิติแรกได้แก่ระนาบเวลาจำลองการทำงาน และมิติที่สองได้แก่ระนาบของ δ-time ซึ่งเป็นระนาบที่อยู่เหนือระนาบเวลาการจำลองทำงาน นั้นคือเวลาจำลองทำงานจะไม่มีการเพิ่มขึ้นจนกว่า เวลา δ-time ทั้งหมดจะผ่านไป

รูปที่ 5.5 แสดงรายละเอียดของขบวนการทำงานการจำลองการทำงานของ VHDL ที่สามารถแบ่งออกได้เป็นสามส่วนคือ **elaboration**, **initialization** และ **simulation cycle** เมื่อระบบจำลองถูกเรียกใช้งาน ระบบจะเริ่มด้วยขั้นตอนของ elaboration และตามด้วยการ initialization ค่าต่างๆ ในรูปแบบ และจากจุดนี้เองวงรอบของ simulation cycle จะทำงานจนกระทั่งสิ้นสุดเวลาการจำลอง



รูปที่ 5.5: **แผนภาพลำดับการทำงานของระบบจำลองการทำงาน** VHDL

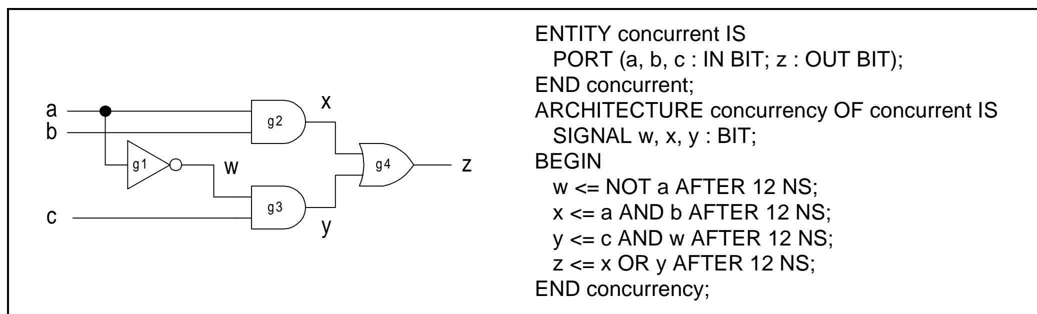
5.4.1 จังหวะเวลาและการทำงานแบบแข่งขนาน (Timing and Concurrency) [2]

ในวงจรอิเล็กทรอนิกส์นั้นอุปกรณ์ทุกชิ้นจะตื่นตัว (active) อยู่ตลอดเวลาต่อสิ่งที่เปลี่ยนแปลงในวงจร การที่ภาษา VHDL เป็นภาษาที่ใช้บรรยายพฤติกรรมของวงจรในลักษณะดังกล่าว ฉนั้นระบบพัฒนา VHDL จึงต้องสามารถที่จะนำมาใช้เขียนรูปแบบที่ให้ความถูกต้องในจังหวะเวลาการทำงาน และการทำงานแบบแข่งขนานในวงจรดิจิตอล โครงสร้างแบบแข่งขนานของภาษา

---

[2] ผู้เรียบเรียงขอใช้คำว่า **"แข่งขนาน"** สำหรับการทำงานแบบ concurrency

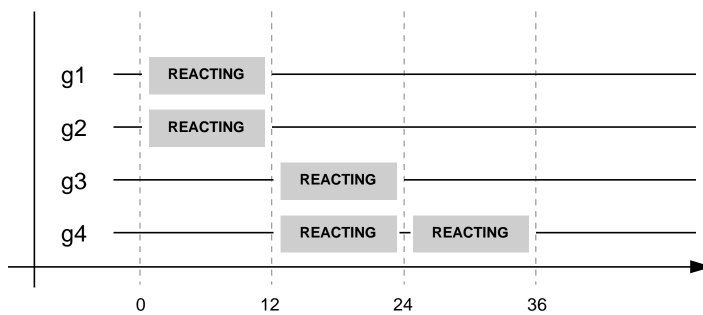
จะใช้ในส่วนที่เป็น concurrent body และชุดคำสั่งแบบลำดับจะใช้ในส่วนที่เป็น sequential body ในส่วนที่เป็น architecture ของรูปแบบจะเป็น concurrent body ในขณะที่ภายใน process statement จะเป็น sequential body ชุดคำสั่งแบบลำดับที่เป็น sequential body ภายใน concurrent body (PROCESS) จะทำงานไปพร้อมๆ กับโครงสร้างแบบแข่งขนานที่อยู่ในรูปแบบเดียวกัน ความสัมพันธ์ด้านจังหวะเวลาระหว่างสัญญาณ และส่วนต่างๆ ในรูปแบบ เป็นสิ่งที่มีความซับซ้อนและสำคัญมาก เพื่อที่จะให้เห็นถึงหลักการดังกล่าวจะแสดงให้เห็นด้วยตัวอย่างต่อไป



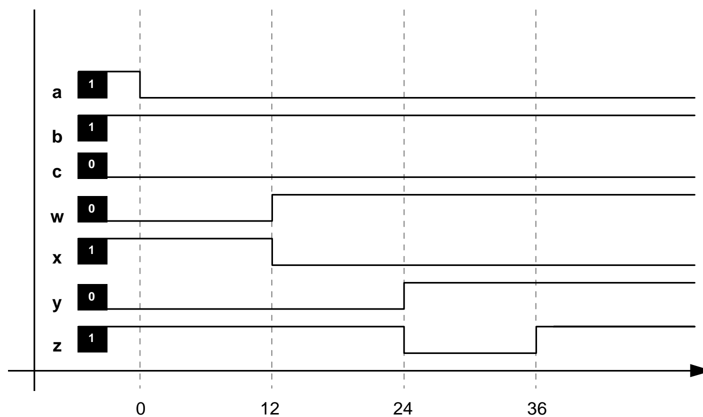
รูปที่ 5.6: **วงจรในระดับ** gate-level **และรูปแบบ** VHDL

รูปที่ 5.6 เป็นวงจรและรูปแบบ VHDL ที่บรรยายวงจรดังกล่าว โดยที่ตั้งข้อสมมุติไว้ว่าทุกๆ gate (inverter, 2-input AND และ 2-input OR) มีการหน่วงเวลาปฏิกิริยาระหว่าง output ต่อการเปลี่ยนแปลงทาง input เท่ากันหมดคือ 12 ns. และระดับค่าของสัญญาณ input ทั้งหมดคงตัวอยู่ที่ระดับค่าลอจิก '1' ในขณะเวลาเริ่มต้นการจำลองการทำงาน (หรือเป็นค่าเริ่มต้นของระบบ) นั่นหมายความว่าที่ตำแหน่ง w, x และ y จะมีระดับค่าของสัญญาณเป็น '0', '1' และ '0' ตามลำดับ ตามที่แสดงในรูปที่ 5.7(b) ถ้าสัญญาณ a เปลี่ยนระดับค่าจาก '1' เป็น '0' ซึ่งการเปลี่ยนแปลงนี้เป็นสาเหตุให้อุปกรณ์ g1 และ g2 มีปฏิกิริยาที่จะเปลี่ยนแปลงพร้อมขนานกัน ตามที่แสดงให้เห็นในลักษณะของ timing diagram ในรูปที่ 5.7(a) อุปกรณ์ g1 จะทำให้ระดับสัญญาณที่ตำแหน่ง w เปลี่ยนเป็น '1' ในเวลา 12 ns. หลังจากนั้น และอุปกรณ์ g2 เป็นสาเหตุทำให้ระดับสัญญาณที่ตำแหน่ง x เปลี่ยนเป็น '0' ในระยะเวลาเดียวกัน ในเวลานี้เอง (ที่ 12 ns.) อุปกรณ์ g3 และ g4 จะมองเห็นการเปลี่ยนแปลงดังกล่าวที่เกิดขึ้นกับ input ของตัวเอง และเริ่มต้นมีปฏิกิริยาต่อค่าใหม่นั้น การเปลี่ยนแปลงที่ตำแหน่ง x ทำให้อุปกรณ์ g4 เปลี่ยนค่า output จาก '1' เป็น '0' ในอีก 12 ns. ข้างหน้า ในขณะเวลาเดียวกันนั้นเอง การเปลี่ยนแปลงที่ตำแหน่ง w เป็นสาเหตุทำให้ระดับ

สัญญาณที่ตำแหน่ง y เป็น '1' ในอีก 12 ns. ข้างหน้าเช่นกัน (การเปลี่ยนแปลงสมบูรณ์ที่ 24 ns. จากเวลาเริ่มต้น) อุปกรณ์ g4 ที่เป็น OR-gate ขณะเวลาที่ 24 ns. จากจุดเริ่มต้นมีระดับค่าของสัญญาณที่ขา input ข้างหนึ่งเป็น '1' จะมีผลทำให้ระดับค่าสัญญาณ output ที่ตำแหน่ง z เปลี่ยนกลับไปเป็น '1' อีกครั้งหลังจากที่ได้มีค่า '0' เพียง 12 ns. เท่านั้น จากรูปที่ 5.7(b) สังเกตได้ว่าระดับของสัญญาณที่ตำแหน่ง z มีค่าเท่ากับ '0' เป็นเวลาเพียง 12 ns. (24 ns. ถึง 36 ns.) และมีชื่อเรียกว่า zero glitch ระบบพัฒนา VHDL จะต้องแสดงการเกิดอาการดังที่กล่าวมาแล้วนี้ได้



(a)



(b)

รูปที่ 5.7: Timing diagram **ของการจำลองการทำงานวงจรในรูปที่** 5.6

การวิเคราะห์ดังกล่าวจะซับซ้อนและยุ่งยากมากยิ่งขึ้นถ้าในกรณีที่อุปกรณ์แต่ละชิ้นมีค่าหน่วงเวลา (delay) ที่ไม่เท่ากัน[3] หรือเมื่อเกิดการเปลี่ยนแปลงขึ้นที่ input อีกในขณะที่วงจรยังไม่อยู่ในสภาวะคงตัว (stable) การวิเคราะห์ที่ผ่านมานี้ถึงแม้จะเป็นเพียงตัวอย่างเล็กๆ แต่ให้ความเข้าใจถึงหลักการของการทำงานแบบแข่งขนาน ซึ่งเป็นหัวใจสำคัญของภาษาที่ใช้บรรยาย hardware โดยตรง จากเหตุผลนี้เองภาษา VHDL จึงเป็นภาษาที่มีส่วนของจังหวะเวลาที่มีความสัมพันธ์กับสัญญาณทุกตัว และสามารถมีโครงสร้างของการกำหนดค่าสัญญาณ ในลักษณะของการแข่งขนานได้ การเขียนรูปแบบระบบดิจิตอลด้วยภาษา VHDL มีความจำเป็นอย่างยิ่งที่จะต้องมีความเข้าใจในหลักการที่กล่าวมาแล้ว

5.4.2 ตัวหน่วงเวลา δ (δ-Delay Time)

ตามที่ได้กล่าวถึงไปแล้วในบทที่ 1 ว่าตัวหน่วงเวลา δ (δ-delay time) จัดเป็นประเภท internal delay ซึ่งเป็นค่าหน่วงเวลาที่เกิดขึ้นภายในระบบพัฒนา VHDL เองโดยไม่มีผลต่อเวลาในการจำลองการทำงาน (simulation time) ของรูปแบบ ตามในตัวอย่างที่แสดงในรูปที่ 5.6 และ 5.7 เป็นวงจร (รูปแบบ) ที่อุปกรณ์ทุกตัวมีค่าหน่วงเวลา เพื่อที่จะบรรยายในหลักการของตัวหน่วงเวลา δ จึงต้องตั้งสมมติฐานว่าอุปกรณ์ g1, g2 และ g3 ในรูปที่ 5.6 ไม่มีค่าหน่วงเวลา (inertial delay) ยกเว้นอุปกรณ์ g4 ด้วยความตั้งใจที่จะแสดงให้เห็นผลลัพธ์ของวงจรที่ตำแหน่ง z และมีรูปแบบตามที่แสดงในรูปที่ 5.8

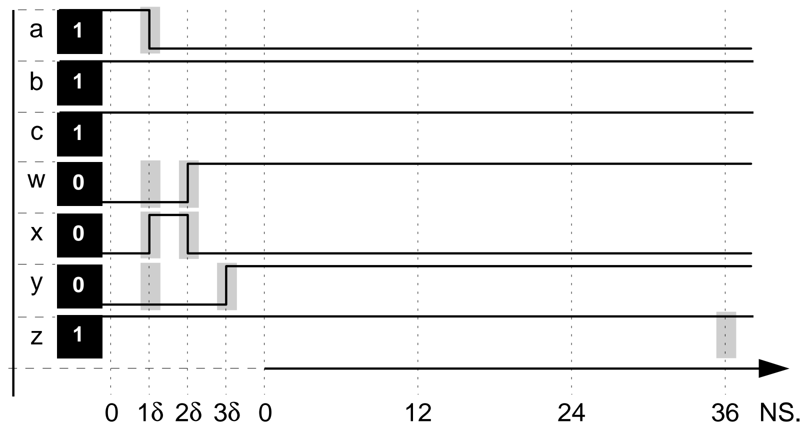
```
ARCHITECTURE delta_time OF concurrent IS
  SIGNAL w, x, y : BIT := '0';
BEGIN
  y <= c AND w;
  w <= NOT a;
  x <= a AND b;
  z <= x OR y AFTER 36 NS;
END delta_time;
```

รูปที่ 5.8: **ตัวอย่างรูปแบบแสดง** delta-time

---

[3] ซึ่งในความเป็นจริงอุปกรณ์อิเล็กทรอนิกส์จะมีคุณสมบัติในการหน่วงเวลาที่ไม่เท่ากัน

การวิเคราะห์ครั้งนี้จะใช้ค่าของสัญญาณที่ป้อนให้ทาง input เดียวกับในรูปที่ 5.7 รูปของ timing diagram ใหม่ที่เกิดขึ้นแสดงในรูปที่ 5.9 ซึ่งมีการแสดงตำแหน่งเวลาที่เกิด transaction ของสัญญาณด้วยบริเวณแถบสีเทา (▌) เช่นเดิมในการเริ่มต้นสมมติให้สัญญาณ a, b และ c เป็นสัญญาณภายนอกและมีค่าเริ่มต้นเท่ากับ '1' (หมายความว่าระดับค่าของสัญญาณในเวลาก่อนศูนย์ มีค่าเท่ากับ '1' หรือ $\lim_{t \to -0}(a,b,c) = \lim_{t=0}(a,b,c) = '1'$) ขณะเวลา 0 ns. สัญญาณ a ถูกกำหนดจากภายนอกให้มีค่า '0' ซึ่งสัญญาณ a จะรับค่าใหม่นี้ในเวลา 1δ ต่อมา หลังจากที่ a เปลี่ยนค่าไปได้อีก δ ต่อมา ตำแหน่ง w และ x จะรับค่าใหม่ที่ input นั้นคือ w จะเปลี่ยนเป็น '1' (ผลลัพธ์ของ NOT a) และ x จะเปลี่ยนค่าเป็น '0' (ผลลัพธ์ของ a AND b) โดยที่ทั้งหมดเกิดขึ้นที่เวลา 2δ การเปลี่ยนแปลงระดับค่า (event) ที่ตำแหน่ง x เป็นสาเหตุทำให้ค่า '0' จะถูกกำหนดให้กับ z (output) ในอีก 36 ns. ข้างหน้า (ชุดคำสั่ง z <= x OR y AFTER 36 NS;) ส่วนการเปลี่ยนแปลงระดับค่า (event) ที่ตำแหน่ง w จะสาเหตุทำให้ชุดคำสั่ง y <= c AND w ทำงาน และผลลัพธ์ที่ได้คือค่าตรงตำแหน่ง y จะเปลี่ยนในเวลา 1δ ต่อมา หลังจากที่ w มีการเปลี่ยนแปลง ซึ่งเป็นการเปลี่ยนจาก '0' เป็น '1' เป็นผลทำให้ที่เวลา 3δ เกิดมีการเปลี่ยนแปลง (event) ที่ y ทำให้ชุดคำสั่ง z <= x OR y AFTER 36 NS; ทำงานอีกครั้ง โดยที่จะเป็นการกำหนดค่า '1' ในอีก 36 ns. ข้างหน้าเช่นกัน



รูปที่ 5.9: Timing Diagram **ที่แสดง** delta-time

การกำหนดค่า '1' ในอีก 36 ns. ครั้งหลังนี้มีผลเหนือการกำหนดค่า '0' ในอีก 36 ns. ครั้งแรก แต่เนื่องจากค่า z เป็น '1' อยู่เดิมแล้วจึงไม่ทำให้เกิดการเปลี่ยนแปลงระดับค่า (event) ในเวลา 36 ns. คงเกิดแต่เพียง transaction เท่านั้น นั่นหมายความว่าระดับคงที่ของค่าสัญญาณที่ z ถูกต้องแล้ว อย่างไรก็ตามการกำหนดค่าต่างๆ ของ x และ y ในช่วงเวลาที่ 0 ns. จะไม่มีผลต่อระดับค่า

สัญญาณของ z ในอีก 36 ns. ข้างหน้า ด้วยเหตุผลที่ว่า ค่าหน่วงเวลาในทางฟิสิกส์ จะดูดกลืน (absorb) ค่าเวลา δ ทั้งหมด

เนื่องจากเวลา δ เป็นส่วนสำคัญที่จะทำให้เกิดความเข้าใจการทำงานระบบจำลองการทำงานแบบ VHDL จึงขอแสดงตัวอย่างให้เห็นอีกครั้งตามรูปที่ 5.10 และ 5.11

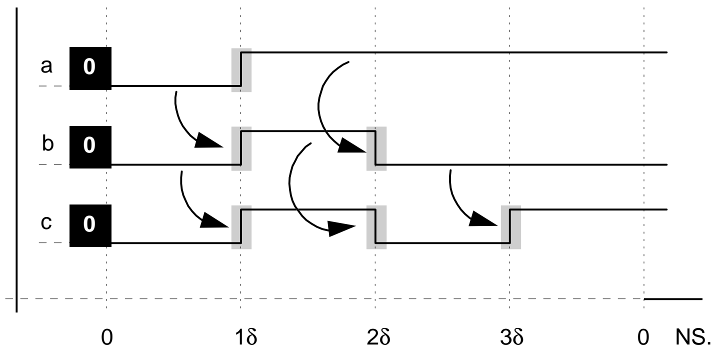
```
ARCHITECTURE concurrent OF delta_time IS
    SIGNAL a, b, c : BIT := '0';
BEGIN
    a <= '1';
    b <= NOT a;
    c <= NOT b;
END concurrent;
```

รูปที่ 5.10: **ตัวอย่างรูปแบบแสดง** concurrency **และ** delta-time

ในรูปที่ 5.10 เป็นรูปแบบของวงจรที่ประกอบด้วยอุปกรณ์ inverter ที่ไม่มีการหน่วงเวลา และต่อเรียงกันสองชิ้น โดยที่จุด a เป็น input จุด c เป็น output และจุด b เป็นจุดระหว่างกลาง ระดับสัญญาณที่ตำแหน่ง a, b และ c ถูกกำหนดค่าเริ่มต้นเป็น '0' ขณะเวลาที่ 0 ns. ระดับค่าสัญญาณ '1' ถูกกำหนดให้กับ a ซึ่งเป็นสาเหตุทำให้เกิด transaction ที่ตำแหน่ง b และ c

รูปที่ 5.11 เป็น timing diagram ที่แสดงให้เห็นการทำงานของชุดคำสั่งแบบแข่งขนาน และการเปลี่ยนแปลงของสัญญาณ a, b และ c ในช่วงเวลาที่ 0 ns. ทุกๆ transaction ในที่นี้จะทำให้เกิดการเปลี่ยนแปลงของระดับสัญญาณในทุกๆ ตำแหน่ง ที่เวลา 0 ns. ระดับค่าของสัญญาณที่ตำแหน่ง a, b และ c เป็นไปตามที่ประกาศในรูปแบบ (SIGNAL) ณ เวลานี้เองค่าลอจิก '1' ถูกกำหนดให้กับ a และค่าตรงข้าม (complement) ของ a ซึ่งยังคงเป็น '0' ที่เวลา 0 ns. (จากค่าเริ่มต้น) จะเป็นค่าที่กำหนดให้กับ b นั้นคือทั้ง a และ b จะมีเกิด transaction ด้วยค่า '1' ที่ 0 ns. เป็นตัวขับ (driver) ในเวลา 0 ns. นี้เองค่าตรงข้าม (complement) ของ b ซึ่งเป็น '0' (จากค่าเริ่มต้น) จะเป็นค่าที่กำหนดให้กับ c เป็นสาเหตุทำให้เกิด transaction ด้วยค่า '1' ที่ 0 ns. เป็นตัวขับให้กับ c เวลา δ ต่อมาที่ 1δ สัญญาณ a, b และ c รับค่าใหม่คือ '1' ค่าใหม่ของ a ทำให้เกิด transaction บน b ในอีก δ ต่อมาที่ 2δ ซึ่งได้แก่การเปลี่ยนระดับค่าเป็น '0' ในลักษณะเดียวกันค่าของ b ที่ 1δ เป็นเหตุทำให้เกิดการเปลี่ยน

แปลงระดับค่าสัญญาณบน c ที่ 2δ การเปลี่ยนแปลงที่เวลา 2δ ทำให้ชุดคำสั่ง c <= NOT b ทำงาน และเป็นเหตุทำให้เกิดการเปลี่ยนแปลงของ c ใน δ ต่อมาที่ 3δ



รูปที่ 5.11: Timing Diagram สำหรับรูปแบบในรูปที่ 5.10

อย่างไรก็ตามระบบจำลองการทำงานของ VHDL ที่แสดง timing diagram ในรูปของ waveform จะไม่แสดงเวลา δ ให้เห็น เพราะเป็น internal delay ที่ไม่ใช่ physical time ที่มีหน่วยวัดทางฟิสิกส์ (เช่น ns.) การที่จะดูการเปลี่ยนแปลงระดับค่าสัญญาณในช่วงเวลา δ นั้นจำเป็นต้องแสดงด้วยหน้าต่างแสดงผล list ของระบบ (list windows) ตามที่แสดงในรูปที่ 5.12

ns	delta	a	b	c
0	+0	0	0	0
0	+1	1	1	1
0	+2	1	0	0
0	+3	1	0	1

รูปที่ 5.12: ผลลัพธ์การจำลองการทำงานแบบแข่งขนาน และมีเวลา δ [4]

ความแตกต่างของการแสดงผลระหว่างหน้าต่าง waveform (timing diagram) กับหน้าต่าง list นั้นจะสังเกตได้ว่าในหน้าต่าง list จะแสดงค่าและเวลาที่สัญญาณมีการเปลี่ยนแปลงระดับค่าของสัญญาณ รวมทั้งเวลา δ ด้วย ในการทำงานจริงกับระบบพัฒนา VHDL นับว่าหน้าต่าง list มีประโยชน์มากในการตรวจสอบขั้นตอนการทำงานที่ละขั้น (debugging) และช่วยให้เกิดความเข้าใจในเรื่องของเวลา δ อย่างมาก

[4] จำลองการทำงานด้วยระบบ V-System ของ Model Technology Inc.

5.5 สรุป

ระบบจำลองการทำงานแบบ VHDL เป็นระบบที่ไม่มีระบบสภาวะที่คงตัว หรือค่าจำลองพื้นฐานที่กำหนดไว้แล้ว ผู้ใช้หรือวิศวกรออกแบบสามารถที่จะกำหนดค่าจำลองได้เอง โดยให้เหมาะสมกับงาน และระดับความละเอียดของงาน สำหรับการทำงานในลักษณะแข่งขนานกันระหว่างชุดคำสั่งต่างๆ ลักษณะพิเศษของระบบจำลองแบบ VHDL นี้ทำให้วิศวกรสามารถที่จะเริ่มต้นทำงานได้ตั้งแต่ในระดับบนสุด และค่อยๆ เพิ่มเติมในรายละเอียดที่ละขั้นตอน ซึ่งนอกจากข้อดีตามที่กล่าวมานี้ ผู้ใช้ยังสามารถที่จะทดสอบแนวความคิดในการแก้ปัญหา โดยการจำลองการทำงานของแนวความคิดนั้นในรูปของรูปแบบก่อนได้ นอกจากนั้นระบบนี้ยังสนับสนุนการทำงานแบบผสมระหว่างความละเอียดของขั้นตอนแต่ละระดับ

ระบบจำลองการทำงานแบบ VHDL เป็นระบบเป็นระบบจำลองการทำงานในลักษณะแข่งขนาน และสามารถที่จะถ่ายทอดรูปแบบสู่ระบบพัฒนา VHDL ได้จากหลายระบบ ด้วยเหตุผลที่ความมีมาตรฐานกำกับอยู่นั้นเอง

ในส่วนหลังของบทนี้ ได้อุทิศให้กับการบรรยายลักษณะการทำงานในระหว่างการจำลองการทำงานแบบแข่งขนาน ซึ่งต้องอาศัยเวลา δ ช่วยในการทำงาน พร้อมทั้งยกตัวอย่างแสดงให้เห็นอย่างละเอียด หลังจากบทนี้ผู้อ่านควรจะมีความสามารถที่จะเข้าใจลักษณะของการทำงานแบบแข่งขนาน ตลอดจนกลไกสำคัญที่สามารถทำให้การจำลองการทำงาน เป็นไปตามคุณสมบัติของวงจรอิเล็กทรอนิกส์จริง

บทที่ 6
ชุดคำสั่งลำดับ (Sequential Statements)

6.1 กล่าวนำ

จากตัวอย่างในบทก่อนๆ เห็นได้ว่าภาษา VHDL สามารถใช้เขียนรูปแบบ (modeling) บรรยายระบบดิจิตอลในลักษณะของ behavioral description ที่โครงสร้างภายในประกอบด้วย sequential statement ฉนั้นในบทนี้จะเป็นการศึกษาในรายละเอียดของโครงสร้างดังกล่าว สำหรับ software engineer ที่มีความคุ้นเคยกับการเขียนโปรแกรมด้วยภาษาชั้นสูง อาทิเช่น C หรือ PASCAL อยู่ก่อนแล้ว จะสามารถทำความเข้าใจโครงสร้างแบบ sequential ได้ง่าย เพียงแต่ต้องทำความเข้าใจเกี่ยวกับลักษณะการทำงานของ hardware เพิ่มเติม ในภาษา VHDL มีคำสั่งที่เป็น sequential statement ดังต่อไปนี้

- WAIT statement
- VARIABLE assignment
- Signal assignment *
- IF-THEN-ELSE statement
- CASE statement
- Loops
- NEXT statement
- EXIT statement
- RETURN statement
- NULL statement
- Procedure call *
- ASSERTION statement *

* *ทำงานได้ทั้ง sequential และ concurrent*

ในบทนี้จะกล่าวเฉพาะ statement ที่สำคัญๆ เพื่อความเข้าใจในการทำงานแบบ sequential เท่านั้น ส่วนที่เหลือสามารถศึกษาเพิ่มเติมในหนังสือคู่มือ LRM หรือจะพบในตัวอย่างที่ใช้ร่วมกับคำสั่ง (statement) อื่นในหนังสือเล่มนี้

ตามที่เคยกล่าวมาแล้วว่าภาษา VHDL เป็นภาษาที่มีคุณสมบัติเป็นแบบแข่งขนาน (concurrent language) นั้นคือชุดคำสั่ง (statement) ภายในตัวโครงสร้างจะเป็นชุดคำสั่งแบบแข่งขนาน (concurrent statement)[1] เช่นเดียวกับภาษา ADA จึงเกิดคำถามขึ้นว่า *"แล้วโครงสร้างที่ประกอบด้วย sequential statement จะไปอยู่ส่วนไหน และอย่างไรภายใน architecture ของ model ?"* คำตอบสำหรับคำถามนี้คือ ชุดคำสั่งลำดับหรือ sequential statement ที่สร้างขึ้น จะถูกบรรจุอยู่ภายในเปลือกของส่วนที่เป็น concurrent หรือที่เรียกกันว่า "concurrent shell" อันก็ได้แก่ process statement

6.2 Process Statement

หัวใจสำคัญของ concurrent shell ที่ทำให้สามารถเขียน VHDL model เพื่อบรรยายพฤติกรรมของระบบดิจิตอลอิเล็กทรอนิกส์ในลักษณะ behavioral description ได้แก่ชุดคำสั่ง process ที่โครงสร้างภายในตัวจะประกอบด้วยชุดคำสั่งแบบลำดับเท่านั้น ชุดคำสั่งเหล่านี้จะทำงานเป็นลำดับจากบนลงล่าง เมื่อ PROCESS ถูกกระตุ้นให้ทำงาน

ชุดคำสั่ง process เป็นพื้นฐานที่สำคัญที่สุด สำหรับการการเขียนรูปแบบ ในลักษณะของ behavioral ซึ่งมีโครงสร้าง และกฎเกณฑ์ในการเขียนตามที่แสดงในรูปที่ 6.1

```
[label]:    PROCESS [(sensitivity list)]
              process declarative part
            BEGIN
              process statement part
              <sequential statement(s)>
            END PROCESS [label] ;
```

รูปที่ 6.1: **โครงสร้างของ** process statement

[1] ชุดคำสั่งแบบแข่งขนาน (concurrent) กล่าวในบทที่ 8

คำ **PROCESS** ในบรรทัดแรก ของโครงสร้าง แสดงถึงจุดเริ่มต้นของชุดคำสั่ง process ส่วนที่เป็น option ได้แก่ label นั้น สามารถใช้เป็นที่สำหรับเขียนชื่อของ PROCESS เพื่อป้องกันการสับสน เพราะในบางกรณีใน architecture หนึ่งอาจจะมีโครงสร้างของชุดคำสั่ง process หลายชุดได้ คำ **END PROCESS** บอกถึงจุดสิ้นสุดของชุดคำสั่ง process และเช่นเดียวกัน label เป็นเพียงสิ่งเผื่อเลือกหรือ option แต่ถ้าจะเขียนกำกับลงไป จะต้องเป็นชื่อเดียวกับที่เขียนไว้ตอนต้นก่อนคำ **PROCESS**

ดังที่เห็นได้จากโครงสร้างในรูปที่ 6.1 ชุดคำสั่ง process อาจจะถูกขยายได้ด้วย sensitivity list (ซึ่งเป็น option) ในส่วนนี้จะเป็นรายชื่อของสัญญาณ ที่ทำหน้าที่กระตุ้น (trigger) การทำงานของ ชุดคำสั่ง process เมื่อมีสัญญาณตัวใดตัวหนึ่งในรายชื่อนี้เกิดการเปลี่ยนแปลงของระดับค่าสัญญาณ (event) ขึ้น หลังจากที่ชุดคำสั่ง process ถูกกระตุ้นด้วยสัญญาณใน sensitivity list แล้ว จะเริ่มทำงานตามคำสั่ง โดยเริ่มจากคำสั่งแรกของชุดคำสั่ง process (ต่อจากคำสั่ง **BEGIN**) เรียงลำดับลงสู่คำสั่งสุดท้ายที่บรรทัดล่างสุด อันได้แก่คำสั่ง **END PROCESS** ซึ่งเรียกการทำงานในลักษณะเช่นนี้ว่า "การทำงานแบบลำดับหรือ sequential" เมื่อคำสั่ง **END PROCESS** ถูกปฏิบัติแล้วชุดคำสั่ง process จะหยุดการทำงานลงชั่วขณะ (แต่ยังคง active อยู่ตลอดเวลา) จนกว่าจะมีสัญญาณอย่างน้อยตัวใดตัวหนึ่งใน sensitivity list เกิด event ขึ้นอีก

เป็นที่เข้าใจดีแล้วว่าชุดคำสั่ง process (PROCESS ... END PROCESS;) เป็นชุดคำสั่งแบบแข่งขนาน นั้นหมายความว่า โดยปกติแล้วชุดคำสั่ง process จะทำงานตลอดเวลา การที่ชุดคำสั่งทั้งหลายที่อยู่ภายในบล็อกของชุดคำสั่ง process เป็นชุดคำสั่งแบบลำดับ นั้นถ้าชุดคำสั่ง process ใดที่ไม่มี sensitivity list เป็นตัวควบคุมการทำงาน จะทำให้เกิดการทำงานที่เปรียบเสมือนว่าเป็นวงรอบ (loop) ที่ไม่รู้จบขึ้นภายในส่วนของชุดคำสั่งแบบลำดับของชุดคำสั่ง process วิธีการที่จะป้องกันการเกิดเหตุการณ์เช่นนี้ คือการเติม wait statement[2] ลงในส่วนของชุดคำสั่ง process

```
do_nothing:  PROCESS
             BEGIN
             END PROCESS do_nothing;
```

รูปที่ 6.2: Process statement

[2] กล่าวในบทที่ 6.3

รูปที่ 6.2 เป็นตัวอย่างของชุดคำสั่ง process ที่มีชื่อว่า ***do_nothing*** และไม่มีการทำงานใดๆ ที่เป็นลำดับภายใน นอกจากตัวเองที่ทำงานตลอดเวลา

ในตัวอย่างต่อไปจะเป็นชุดคำสั่ง process ที่ใช้สร้างสัญญาณนาฬิกา (สัญญาณที่เปลี่ยนระดับค่าของสัญญาณสม่ำเสมอ) ตามรูปที่ 6.3

```
clock: PROCESS (clk)
          VARIABLE periodic : BIT := 0;
       BEGIN
          periodic := NOT (periodic) AFTER 50 NS;
          clk <= periodic;
       END PROCESS clock;
```

รูปที่ 6.3: Process statement **ที่สร้างสัญญาณนาฬิกา** (clk)

ชุดคำสั่ง process ชื่อ ***clock*** ถูกควบคุมการทำงานด้วยสัญญาณ ***clk*** ฉนั้นเมื่อสัญญาณนี้เกิด event จะกระตุ้นให้ชุดคำสั่ง process ทำงาน ที่ภายในโครงสร้างประกอบด้วยชุดคำสั่งลำดับสองคำสั่ง คำสั่งแรกเป็นคำสั่งที่กำหนดค่าตัวแปร (variable assignment[3]) ซึ่งจะกำหนดค่าตรงข้ามของตัวแปร periodic ปัจจุบันใหม่ (update) ในอีกเวลา 50 ns. ข้างหน้า คำสั่งที่สองเป็นคำสั่งกำหนดค่าสัญญาณ (signal assignment) โดยกำหนดค่าของตัวแปร periodic ให้กับสัญญาณ clk นั้นแสดงว่าหลังจากเวลาใดเวลาหนึ่งที่สัญญาณ ***clk*** มีระดับสัญญาณคงที่ (stable) ไปอีก 50 ns. จะเกิด event ขึ้น และชุดคำสั่ง process ชื่อ clock จะทำวงรอบขึ้นมาอีกครั้ง ฉนั้นผลลัพธ์ที่ได้คือสัญญาณ ***clk*** จะเปลี่ยนระดับค่าสัญญาณทุกๆ 50 ns. นั้นเอง

[3] signal assignment statement เป็น concurrent statement และ sequential statement

6.3 Wait Statement

จากการที่ชุดคำสั่ง process สามารถมี sensitivity list ได้เพียงอันเดียว (แต่ใน list สามารถมีสัญญาณได้หลายสัญญาณ) นั้น หมายความว่าชุดคำสั่ง process จะถูกกระตุ้นได้จากการที่สัญญาณใดสัญญาณหนึ่งในรายชื่อที่เกิด event ขึ้นเท่านั้น หลังจากที่ถูกกระตุ้นแล้วคำสั่งทั้งหลายที่อยู่ภายในจะทำงานแบบลำดับลงมาจนกระทั้งหมด และชุดคำสั่ง process จะหยุดการทำงานชั่วคราวจนกว่าจะมี event เกิดขึ้นอีกกับสัญญาณตัวใดตัวหนึ่งในรายชื่อนั้นอีก ถ้าในกรณีที่ภายในรายชื่อประกอบด้วยสัญญาณหลายตัว และเกิด event ขึ้นในเวลาเดียวกัน จะมีสัญญาณเพียงตัวเดียวจากทั้งหมดเท่านั้น ที่กระตุ้นการทำงานของชุดคำสั่ง process ซึ่งไม่สามารถที่จะบอกได้ว่าเป็นตัวใด ดังนั้นการใช้ชุดคำสั่ง process ร่วมกับ sensitivity list จึงมีขีดจำกัดอยู่มาก ในภาษา VHDL มีวิธีการหลีกเลี่ยงปัญหาเช่นนี้โดยใช้คำสั่งที่มีความอ่อนตัวมากกว่า อันได้แก่ชุดคำสั่ง wait statement

คำสั่ง wait statement นั้นให้ความคล่องตัวในการใช้มากกว่าการใช้ sensitivity list เพราะการใช้ wait statement มีข้อดีคือ ประการแรก สามารถที่จะกำหนดลงตรงตำแหน่งใดๆ ภายในโครงสร้างของชุดคำสั่งลำดับได้ เพื่อระงับการทำงานภายในชุดคำสั่ง process ตรงตำแหน่งที่ wait statement อยู่ ส่วนการใช้ sensitivity list เป็นตัวควบคุมการทำงาน ชุดคำสั่ง process จะหยุดตรงตำแหน่งสุดท้ายของชุดคำสั่ง process เท่านั้น ประการที่สอง สามารถที่จะกำหนด wait statement ได้หลายๆ อันในแต่ละตำแหน่งตามความต้องการโดยไม่จำกัด ส่วนประการสุดท้าย wait statement มีหลายรูปแบบ นั่นหมายความว่า สามารถที่จะใช้ควบคุม PROCESS ได้หลายลักษณะ แต่มีสิ่งที่ต้องจำไว้ว่า ***wait statement ไม่สามารถที่จะใช้ร่วมกับ sensitivity list ภายใน PROCESS เดียวกันได้*** ฉนั้นผู้เขียนรูปแบบจึงต้องตัดสินใจก่อนว่าจะใช้อะไร ทั้งนี้ขึ้นอยู่กับระบบดิจิตอลที่จะเขียนบรรยาย และประสบการณ์ในการเขียน

เนื่องจากในหน้าที่แล้วทั้ง wait statement และ sensitivity list ต่างก็ให้ผลเช่นกัน คือหยุดยั้งการทำงานของ PROCESS ฉนั้นทั้งคู่จึงมีความสมมูลย์กันถ้าคำสั่ง

WAIT ON signal1, signal2,... ;

อยู่บรรทัดสุดท้ายของชุดคำสั่งลำดับใน PROCESS คำสั่งนี้หมายความว่าถ้าสัญญาณใดในรายชื่อ (signal1, signal2,...) มี event จะส่งผลทำให้ sequential statement ที่หยุดอยู่ ณ ตำแหน่งนั้นจะเริ่มต้นทำงานตามลำดับต่อไป

ในภาษา VHDL สามารถใช้ wait statement ได้ 4 แบบคือ

- WAIT ON signal_list; -- signal sensitivity
- WAIT UNTIL condition; -- condition
- WAIT FOR time; -- timeout
- WAIT; -- forever

ซึ่งแต่ละอย่างมีความหมายและวิธีใช้ที่แตกต่างกันตามตัวอย่างต่อไปนี้

- **WAIT ON clock, clear, preset, d;**

  คำสั่งนี้จะหยุดการทำงานของชุดคำสั่งลำดับ ไว้จนกว่าจะเกิด event ขึ้นที่สัญญาณ **clock** หรือ **clear** หรือ **preset** หรือ **d**

- **WAIT UNTIL (clock = '1');**

  การทำงานของชุดคำสั่งลำดับจะหยุดที่ตำแหน่งนี้ และจะทำงานต่อไปเมื่อสัญญาณ clock เกิด event และ boolean expression ภายในวงเล็บ (clock = '1') เป็น TRUE ทุกครั้งเมื่อลำดับการทำงานถูกหยุดตรงจุดนี้ และก่อนที่จะทำต่อไปได้นั้น ก่อนอื่นต้องตรวจสอบว่ามี event เกิดขึ้นบนสัญญาณ clock หรือเปล่า และข้อแม้ที่ว่า (หลังจากเกิดแล้ว) สัญญาณ clock มีค่าเป็น '1' ฉนั้น boolean expression จะต้องเป็นจริง (**TRUE**)

- **WAIT FOR 10 NS;**

  คำสั่งนี้เป็นการรอหรือหยุดในขณะที่จำลองการทำงาน (simulation) จนกว่าเวลาการจำลอง (simulation time) จะล่วงเลยไปแล้ว 10 ns. จะมีผลทำให้ PROCESS เริ่มทำงานต่อไปได้ คำสั่งนี้สามารถนำไปใช้เขียนรูปแบบได้อย่างมีประสิทธิภาพ ในสถานการณ์ที่เรียกว่า "timeout"

- **WAIT;**

  เป็นการคำสั่งให้ PROCESS หยุดการทำงานตลอดไป โดยไม่มีการเริ่มต้นการทำงานใหม่ บางครั้งการเขียนรูปแบบบรรยายการทำงานของระบบดิจิตอล มีความจำเป็นที่ต้องใช้คำสั่งนี้ เพื่อหยุดการทำงานของ PROCESS อย่างถาวร

นอกจากนั้นยังสามารถใช้ตัวควบคุม (ON, UNTIL และ FOR) ร่วมกันใน wait statement ได้ เช่น ใช้

- *ON ร่วมกับ UNTIL การทำงานจะถูกหยุดจนกระทั่งสัญญาณในส่วนของ ON มี event* ***และ*** *ข้อแม้ในส่วน UNTIL เป็นจริง*
- *ON ร่วมกับ FOR การทำงานจะถูกหยุดจนกระทั่งสัญญาณในส่วนของ ON มี event* ***หรือ*** *เมื่อเวลาในส่วน FOR ผ่านไป*
- *UNTIL ร่วมกับ FOR การทำงานจะถูกหยุดจนกระทั่งข้อแม้ในส่วน UNTIL เป็นจริง* ***หรือ*** *เมื่อเวลาในส่วน FOR ผ่านไป*
- *ทั้งสามใช้ร่วมกับใน wait statement เดียวกัน การทำงานจะถูกหยุดจนกระทั่งสัญญาณในส่วนของ ON มี event* ***และ*** *ข้อแม้ในส่วน UNTIL เป็นจริง* ***หรือ*** *เมื่อเวลาในส่วน FOR ผ่านไป*

ฉนั้นถ้าชุดคำสั่ง process ใดไม่มี sensitivity list และ wait statement ภายในโครงสร้าง PROCESS นั้นจะเกิดการทำงานเป็นวัฏจักรที่ไม่มีจุดจบ ความจริงข้อนี้เป็นสิ่งที่จะต้องจำไว้อยู่เสมอ เพราะในช่วงเวลาของการเตรียมค่าเริ่มต้น (initialization) ในการจำลองการทำงาน (simulation) ชุดคำสั่งแบบแข่งขนานจะทำงานก่อนหนึ่งครั้ง เช่นเดียวกันชุดคำสั่ง process ซึ่งเป็นชุดคำสั่งแบบแข่งขนานจะทำงานจนกว่าจะถูกหยุดด้วย sensitivity list หรือ wait statement ถ้าปราศจากตัวควบคุมทั้งสองอย่างนี้ PROCESS จะไม่หยุดการทำงาน ซึ่งส่วนใหญ่ของ VHDL analyzer จะตรวจสอบเพื่อให้เป็นที่แน่ใจว่าใน PROCESS มี sensitivity list หรือ wait statement อย่างใดอย่างหนึ่ง เพื่อป้องกันปัญหาอันอาจจะเกิดขึ้นได้ คือในการพยายามจำลองการทำงาน (simulation) เครื่องไม่สามารถทำได้ เพราะเครื่องจะหยุดนิ่งหรือที่เรียกว่า "hang" ในระหว่างการเตรียมค่าเริ่มต้น เนื่องจากเกิดการทำงานไม่รู้จบ

6.4 IF-THEN-ELSE statement

สำหรับผู้ที่เคยเขียนโปรแกรมด้วยภาษาคอมพิวเตอร์ชั้นสูงอาทิเช่น C หรือ PASCAL จะคุ้นเคยกับชุดคำสั่งนี้ดี (ถึงแม้ว่าการใช้ภาษา VHDL จะมีความคล้ายกับการเขียนโปรแกรม แต่สำหรับ hardware design แล้วจะใช้คำว่า **การเขียนรูปแบบ หรือ modeling** แทนคำว่า programming) เพราะเป็นชุดคำสั่งพื้นฐานที่ใช้สอบถาม เพื่อการตัดสินใจกระทำอะไรบางสิ่งบางอย่าง สิ่งที่จะแตกต่างไปจากภาษาโปรแกรมคือ กฎเกณฑ์ (syntax) ในการเขียน ภาษา VHDL มีกฎเกณฑ์ในการเขียนดั่งในรูปที่ 6.5

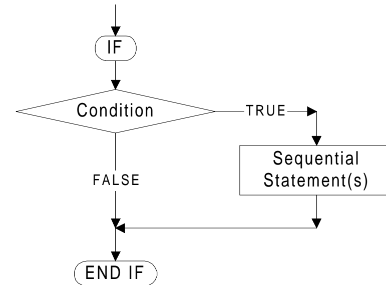
```
    IF condition THEN
    {sequential-statement(s)}
[{  ELSIF condition THEN      }
        {sequential_statement(s)}]
[   ELSE
        {sequential_statements}]
    END IF;
```

รูปที่ 6.5: **โครงสร้างของ** IF-THEN-ELSE statement

จะเห็นได้ในรูปที่ 6.5 ว่าสามารถที่จะเขียน IF-THEN-ELSE statement ได้หลายรูปแบบ เพื่อความง่ายต่อการศึกษา จะขอยกตัวอย่างดังนี้

IF statement

```
Format:  IF condition THEN
              sequential_statement(s)
         END IF;
```



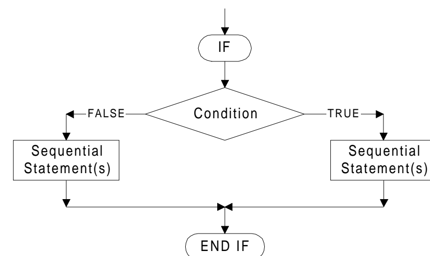
ตัวอย่าง:

```
IF a = '1' THEN
    count := count + 1;
END IF;
```

ข้อแม้ (condition) ในบรรทัดแรกของ IF statement จะให้ผลลัพธ์ที่มี TYPE เป็น BOOLEAN (TRUE หรือ FALSE) ชุดคำสั่งที่เป็น (และต้องเป็นเท่านั้น) ลำดับจะทำงานเมื่อ ผลลัพธ์ที่ได้จากข้อแม้เป็น TRUE แต่ถ้าเป็น FALSE การทำงานของ PROCESS จะข้ามชุดคำสั่งลำดับทั้งหมดไป และเริ่มต้นทำคำสั่งที่อยู่ต่อจากคำสั่ง END IF ต่อไป

IF-ELSE construct

```
Format:  IF condition THEN
              sequential_statement(s)
         ELSE
              sequential_statement(s)
         END IF;
```



ตัวอย่าง:

```
IF a = '1' THEN
  one_count := count + 1;
ELSE
  two_count := count + 2;
END IF;
```

เช่นเดียวกับกรณีแรก คือโครงสร้าง IF-ELSE จะมีข้อแม้ (condition) เพื่อการตัดสินใจ แต่เพิ่มหนทางปฏิบัติให้กับทางเลือกอีกหนทางหนึ่ง และจากในตัวอย่างข้างบน จะเห็นได้ว่าเป็นการทำงานของชุดคำสั่งที่เรียกว่า variable assignment

one_count := count + 1;

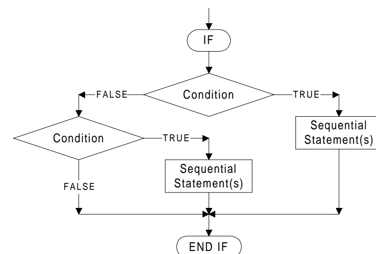
จะทำงาน และทำเป็นลำดับต่อไป ถ้าสมมุติว่ามี sequential statement อื่นอีก ต่อเมื่อข้อแม้ (condition) เป็น TRUE เท่านั้น นั้นคือ a จะต้องมีค่าเท่ากับ '1' และก็จะข้ามกลุ่มของชุดคำสั่งลำดับที่เป็นทางเลือกอีกหนทางหนึ่งไปทำคำสั่งแรกที่ต่อจากคำสั่ง END IF ต่อไป สำหรับคำสั่ง

two_count := count + 2;

นั้นจะทำงานเมื่อ condition ให้ค่าเป็น FALSE นั้นคือ a จะต้องมีค่าอื่นๆ ที่ไม่ใช้ค่า '1' (ไม่จำเป็นต้องเป็น '0' เสมอไป อาจจะเป็น 'X' ค่า unknow ได้ โดยที่จะข้ามกลุ่มของ sequential statement ที่เป็นทางเลือกอีกหนทางหนึ่ง (หนทางที่ต่อจาก THEN) ไป และหลังจากนั้น จะไปทำคำสั่งแรกที่ต่อจากคำสั่ง END IF ต่อไป

IF - ELSIF construct

```
Format:  IF condition THEN
             sequential_statement(s)
         ELSIF condition THEN
             sequential_statement(s)
         END IF;
```




```
ตัวอย่าง:   IF a = 'X'      THEN
                b := b + 1;
            ELSIF a = '0'    THEN
                c := c + 1;
            ELSIF a = '1'    THEN
                d := d+1;
            ELSIF a = 'Z'    THEN
                e := e + 1;
            E    N    D      I    F    ;
```

โครงสร้าง ELSIF นับว่ามีประโยชน์ในการเขียนรูปแบบด้วยชุดคำสั่งลำดับที่มีข้อแม้ (condition) หลายอัน และในขณะเดียวกันก็มี หลายหนทางปฏิบัติ ดังที่แสดงในตัวอย่าง ในกรณีที่ a (เป็น signal หรือ variable) สามารถมีค่าได้ 4 ค่า ได้แก่ 'X', '0', '1' และ 'Z' ซึ่งแต่ละค่านั้นให้ผลของการตัดสินใจ และหนทางปฏิบัติที่แตกต่างกัน คำถาม ELSIF สุดท้ายสามารถแทนได้ด้วยคำสั่ง

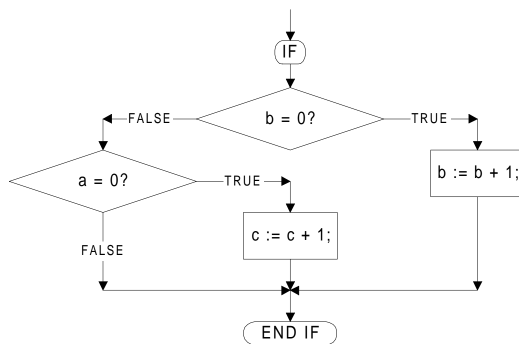
ELSE e := e + 1;

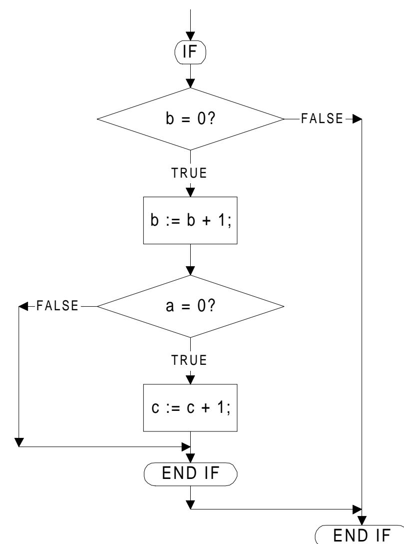
เพราะความเป็นไปได้สุดท้ายที่ a สามารถมีค่าได้คือ 'Z' เท่านั้น ฉนั้นถึงแม้ว่าโครงสร้าง ELSIF จะมีประโยชน์มาก แต่ในการใช้ต้องคำนึงถึงกฎเกณฑ์ต่อไปนี้เสมอ

- ☑ ELSE สามารถใช้เป็นคำสั่งสุดท้ายเท่านั้น และใช้แทน ELSIF เมื่อต้องการรวมความเป็นไปได้ทั้งหมดที่ยังเหลืออยู่เข้าไว้ในการสอบถามเดียวกัน
- ☑ ELSE ที่อยู่สุดท้ายนี้ จะสัมพันธ์กับ ELSIF ที่อยู่ติดกันก่อนที่จะถึงตำแหน่งที่ ELSE อยู่
- ☑ ข้อแม้ใน ELSIF จะต้องไม่สอบถามในคำถามที่ได้ตรวจสอบมาก่อนแล้ว
- ☑ เพียงคำสั่งแบบ sequential (คำสั่งเดียว หรือลำดับของคำสั่ง) ที่อยู่ต่อจากการสอบถาม ELSIF ที่ให้ผลลัพธ์ TRUE เท่านั้น ที่จะทำงานส่วนที่เหลือจากการสอบถามอื่นๆ จะถูกข้ามไปหมดโดยไม่สอบถามข้อแม้ต่อมาอีก
- ☑ การใช้ ELSIF จะให้ผลไม่เหมือนกับการสร้าง IF- ซ้อน IF statement

```
IF b = '0' THEN
    b := b +1;
ELSIF a = '0' THEN
    c := c + 1;
END IF;
```

```
IF b = '0' THEN
b := b + 1;
    IF a = '1' THEN
        c := c +1;
    END IF;
END IF;
```

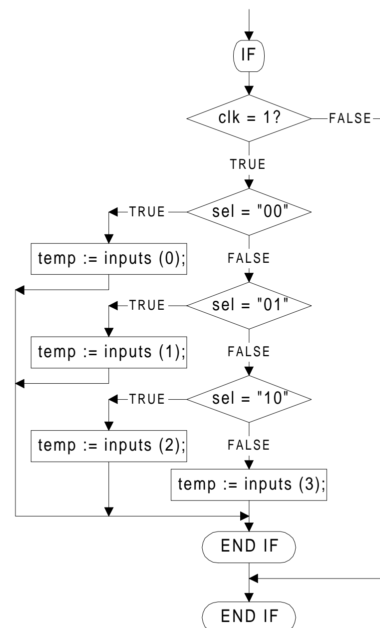




หลังจากที่ได้เห็นลักษณะการใช้ ตลอดจนความหมายต่างๆ แล้วต่อไปจะมาศึกษาจากตัวอย่างเสริมความเข้าใจอีกครั้งในรูปที่ 6.6

ส่วนที่เป็น architecture ชื่อ **behave_if** ใช้บรรยายพฤติกรรมของ entity design unit ชื่อ **clocked_mux** ประกอบด้วยชุดคำสั่ง process เดียว และมีสัญญาณ ***clk*** เป็น sensitivity list ในส่วนของชุดคำสั่งลำดับนั้นเขียนขึ้นโดยใช้ชุดคำสั่ง IF-THEN-ELSE ที่ซ้อนกันอยู่สองชุด โดยที่ชุดแรกที่อยู่วงนอกสุด ใช้สอบถามสัญญาณ ***clk*** ชุดที่สองอยู่วงในมีหน้าที่สอบถามข้อมูล ***sel*** เพื่อการตัดสินใจกำหนดค่า ***output*** เนื่องจาก PROCESS จะทำงานได้ก็ต่อเมื่อเกิด event ที่สัญญาณ ***clk*** นั้นหมายความว่าต้องมีการเปลี่ยนแปลงจาก '1' เป็น '0' หรือจาก '0' เป็น '1'

```
ENTITY clocked_mux IS

  PORT ( inputs: IN BIT_VECTOR (0 TO 3);
         sel: IN BIT_VECTOR (0 TO 1);
         clk: IN BIT; output: OUT BIT);
END clocked_mux;

ARCHITECTURE behave_if OF clocked_mux
IS
BEGIN
  PROCESS (clk)
     VARIABLE temp: BIT;
  BEGIN
     IF clk = '1' THEN
        IF sel = "00"        THEN
           temp := inputs (0);
        ELSIF sel = "01"     THEN
           temp := inputs (1);
        ELSIF sel = "10"     THEN
           temp := inputs (2);
        ELSE
           temp := inputs (3);
        END IF;
        output <= temp AFTER 5 NS;
     END IF;
  END PROCESS;
END behave_if;
```



รูปที่ 6.6: Behavioral model **ของ** 4:1 clocked multiplex (IF cluase)

ถ้าสัญญาณ ***clk*** เปลี่ยนแปลงจากค่า '1' เป็น '0' ถึงแม้ว่า PROCESS จะถูกกระตุ้นแต่คำถามข้อแม้ IF ที่ถาม clk = '1' ? จะให้คำตอบ FALSE ซึ่งหมายความว่า การทำงานจะข้ามชุดคำสั่งลำดับที่อยู่ภายในทั้งหมด รวมทั้งบล๊อกของคำถามข้อแม้ IF วงในด้วย และจะหยุดการทำงานจนกว่าจะเกิด event ที่สัญญาณ ***clk*** อีก นั่นคือจะไม่มีการเปลี่ยนแปลงที่สัญญาณ ***output*** ในทางตรงข้ามถ้าเป็นการเปลี่ยนแปลงจาก '0' เป็น '1' ที่สัญญาณ ***clk*** คำถามข้อแม้ IF ที่ถามว่าสัญญาณ clk = '1' ? จะให้คำตอบ TRUE การทำงานขั้นต่อไปคือคำถามข้อแม้ IF วงใน ซึ่งจะเป็นการสอบถามค่าสัญญาณ ***sel*** เพื่อเลือก ***inputs*** ตัวใดตัวหนึ่งส่งไปยัง ***output*** โดยที่ค่าที่ได้ครั้งแรกจะถูกกำหนดไปที่ตัวแปร (variable) ชื่อ ***temp*** ก่อน และกำหนดให้กับสัญญาณ (signal) ***output*** ภายหลังด้วย signal assignment statement และมีค่า inertial delay 5 ns. ฉนั้นจากโครงสร้างนี้ถึงแม้ว่าสัญญาณ ***inputs*** จะมีการเปลี่ยนแปลงได้ตลอดเวลา แต่ค่าที่จะถูกเลือกออกไปยัง ***output*** นั้นจะเป็นค่าปัจจุบันในขณะที่สัญญาณ ***clk*** เปลี่ยนจาก '0' เป็น '1' เท่านั้น

จากรูปที่ 6.6 นี้จะเห็นได้ว่ามีการใช้คำจำกัดความใหม่ๆ เช่น VARIABLE ซึ่งจะมีความแตกต่างไปจาก SIGNAL คือ

1) จากกฎเกณฑ์การใช้ภาษา VARIABLE operator จะใช้เครื่องหมาย **:=** ในขณะที่ SIGNAL operator ใช้เครื่องหมาย <=
2) ค่าต่างๆ ที่กำหนดให้ VARIABLE จะกระทำ **ทันที** ส่วน SIGNAL จะรับค่าใหม่อย่างน้อยในเวลา $+\delta$ ต่อมา
3) VARIABLE สามารถใช้ได้เฉพาะในส่วนที่ตัวมันถูกกำหนด (local) ในขณะที่ SIGNAL สามารถใช้ได้ตลอด architecture (global)
4) VARIABLE เป็นเพียงตัวแปร ไม่ได้มีความเกี่ยวข้องกับ hardware แต่อย่างใด ส่วน SIGNAL นั้นเป็นตัวที่ทำหน้าที่เป็นพาหนะ รับ-ส่ง ค่าของสัญญาณ จึงเป็นส่วนของ hardware

6.5 CASE Statement

โครงสร้างอีกอันหนึ่งที่เป็นลำดับ และมีความคล้ายคลึงกับ IF-THEN-ELSE คือ case statement ที่มีความใกล้เคียงกันนั้น เพราะใช้เป็นคำสั่งเลือกหนทางปฏิบัติ ตามข้อแม้ (condition) ที่กำหนดให้ case statement มีกฎเกณฑ์การเขียนดังนี้

```
CASE expression IS
   WHEN choice => sequential statement(s)
[   WHEN choice => sequential statement(s) ]
END CASE;
```

รูปที่ 6.7: **โครงสร้างของ** CASE statement

คำสั่ง **CASE** และ **END CASE** กำหนดจุดเริ่มต้น และจุดสิ้นสุดของ case statement คำสั่ง **WHEN** ใช้สำหรับกำหนดตัวเลือก ที่จะนำมาเปรียบเทียบกับ expression ตัวเลือกใดเป็นไปตาม expression ข้างบน PROCESS จะเริ่มต้นทำงานที่ชุดคำสั่งลำดับที่ตามมาจนกระทั่งคำสั่งสุดท้ายของตัวเลือกนั้นๆ และจะออกจาก CASE statement โดยไม่ทำหนทางเลือกอื่นๆ ที่ยังคงเหลืออยู่ ตัวอย่างในรูปที่ 6.8 เป็นการเขียน 4:1 clocked multiplexer ขึ้นใหม่โดยใช้ CASE statement

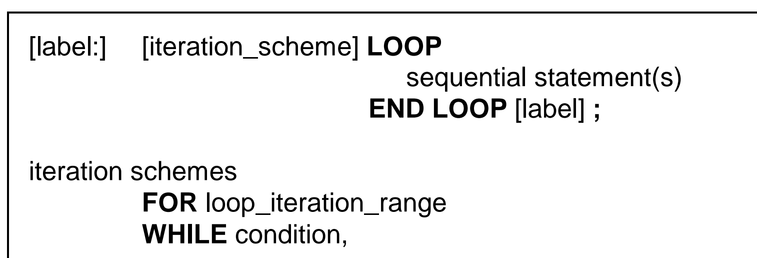
```
ARCHITECTURE behave_case OF clocked_mux IS
BEGIN
   PROCESS (clk)
   VARIABLE temp: BIT;
   BEGIN
      CASE clk IS
         WHEN '1' => CASE sel IS
                        WHEN "00"  => temp := inputs (0);
                        WHEN "01"  => temp := inputs (1);
                        WHEN "10"  => temp := inputs (2);
                        WHEN "11"  => temp := inputs (3);
                     END CASE;
                     output <= temp AFTER 5 NS;
         WHEN OTHERS => NULL;       -- do nothing
      END CASE;
   END PROCESS;
END behave_case;
```

รูปที่ 6.8: Behavioral model **ของ** 4:1 clocked multiplex (CASE cluase)

เช่นเดียวกันในที่นี้เป็นโครงสร้างของ CASE ซ้อน CASE statement โดยที่วงนอกสุดทำหน้าที่ตรวจค่าสัญญาณ ***clk*** ว่าเป็น '1' หรือ '0' ถ้าเป็น '1' CASE วงในจะทำงาน ซึ่งชุดคำสั่งลำดับทำหน้าที่กำหนดค่าตัวแปรจาก ***input*** ที่ถูกต้องส่งไปยัง ***output*** ทั้งนี้การเลือกกำหนดจะขึ้นอยู่กับค่าของสัญญาณ ***sel*** สำหรับความเป็นไปได้กรณีที่สองของ CASE วงนอกคือ clk = '0' นั้นในที่นี้ใช้ **OTHERS** แทน ก็เพื่อแสดงให้เห็นว่า สามารถใช้คำเพียงคำเดียวแทนความเป็นไปได้อื่นๆ ที่ยังเหลืออยู่ แต่ให้ผลของการตัดสินใจอย่างเดียวกันได้ทั้งหมด ซึ่งบางกรณีสามารถใช้ประหยัดการเขียนวิธีเลือกหนทางปฏิบัติด้วย WHEN ที่ไม่จำเป็นได้ เพราะการใช้ CASE statement จะต้องกำหนดความเป็นไปได้ทุกกรณีให้กับหนทางเลือก

6.6 LOOP Statement

คือการทำงานในลักษณะลำดับที่เป็นวงรอบ (loop) เช่นเดียวกับภาษาคอมพิวเตอร์ชั้นสูงทั่วๆ ไป มีกฎเกณฑ์ในการเขียนตามรูปที่ 6.9



```
[label:]    [iteration_scheme] LOOP
                    sequential statement(s)
                END LOOP [label] ;

iteration schemes
            FOR loop_iteration_range
            WHILE condition,
```

รูปที่ 6.9: **โครงสร้างของ** Loop statement

คำสั่ง **LOOP** และ **END LOOP** แสดงตำแหน่งของจุดเริ่มต้น และจุดสิ้นสุดของ loop statement จากโครงสร้างในรูปที่ 6.9 จะเห็นว่าสามารถสร้าง loop ได้สามลักษณะคือ LOOP ธรรมดา, WHILE-LOOP และ FOR-LOOP ในที่นี้เนื่องจากไม่ต้องการให้ผู้อ่านเกิดความสับสนมาก จะขอกล่าวเฉพาะ FOR-LOOP และ WHILE-LOOP เท่านั้น

FOR-LOOP:

กลุ่มคำสั่งลำดับใน FOR-LOOP จะทำงานเป็นวงรอบตราบเท่าที่ ค่าของพารามิเตอร์ยังอยู่ในขอบข่าย (range) ที่กำหนด

```
[label:]    FOR loop_parameter IN discrete_range
            LOOP
                sequential statement(s)
            END LOOP [label] ;
```

รูปที่ 6.10: **โครงสร้างของ** FOR-LOOP

ตัว loop_parameter เป็นพารามิเตอร์ที่แฝงอยู่ในโครงสร้าง ที่ไม่ต้องประกาศหรือกำหนดขึ้น ชื่อที่ตั้งให้ต้องเป็นไปตามกฎของการตั้งชื่อในภาษา VHDL ค่าของพารามิเตอร์นี้ห้ามถูกแก้ไขดัดแปลงด้วยคำสั่งใดๆ ทั้งจากภายในและภายนอกโครงสร้าง FOR-LOOP อย่างเด็ดขาด แต่สามารถนำค่าไปใช้อ้างอิงได้จากภายในโครงสร้าง (ไม่สามารถใช้อ้างอิงจากภายนอกได้) ดังตัวอย่างการใช้ในรูปที่ 6.11

```
lb1:   FOR index IN 0 TO 7 LOOP
           ray_out (index) <=  ray_in (index);
       END LOOP lb1;
```

รูปที่ 6.11: **ตัวอย่างการใช้** FOR-LOOP

ตัวอย่างในรูปที่ 6.11 แสดงวิธีการย้ายข้อมูลที่อยู่ในรูปของ array ชื่อ ***ray_in*** ไปยัง array ที่ชื่อ ***ray_out*** ตำแหน่งต่อตำแหน่ง โดยอาศัย loop parameter ชื่อ ***index*** เป็นตัวอ้างอิงจากโครงสร้างของ FOR-LOOP นี้ ค่าของ ***index*** จะเริ่มต้นด้วยเลขจำนวนเต็มบวก[4] (integer) 0 ทุกรอบของการทำงานและก่อนที่จะเริ่มทำงานในวงรอบใหม่ ***index*** จะเพิ่มค่าตัวเองขึ้นอีก 1 (เปรียบเสมือน

---

[4] ค่าพารามิเตอร์ไม่จำเป็นต้องเริ่มต้นด้วยค่าต่ำสุดเช่น 0 เสมอ

ว่ามีคำสั่ง index := index + 1 แฝงอยู่เพราะมีการกำหนดทิศทางเพิ่มขึ้นจาก 0 TO 7) วงรอบสุดท้ายของการทำงาน คือวงรอบที่ค่าของ loop parameter มีค่าน้อยกว่าหรือเท่ากับค่าสูงสุดของ discrete range (ใน ต.ย. ค่าสูงสุด = 7) ฉนั้นในตัวอย่างจึงทำงานทั้งหมด 8 วงรอบ

บางครั้งมีความจำเป็นที่จะต้องลดค่าของ loop parameter ลงเป็นจำนวนเท่ากับ 1 ทุกครั้งที่ทำงานครบวงจรหนึ่งรอบ ซึ่งสามารถกระทำได้โดยการเขียน discrete range ใหม่เพื่อบ่งบอกถึงทิศทางในการนับคือ

```
lb1:    FOR index IN 7 DOWNTO 0 LOOP
```

สิ่งที่พึงจำไว้ในการสร้างคือ discrete range จะต้องเป็นค่าที่ลงตัว ตัวเลขจำนวนเต็มสามารถใช้ได้หรือตัวอักษรเช่น a TO f เป็นต้น

WHILE-LOOP:

```
[label:]    WHILE boolean_expression
            LOOP
                sequential statement(s)
            END LOOP [label] ;
```

รูปที่ 6.12: **โครงสร้างของ** WHILE-LOOP

กลุ่มคำสั่งลำดับใน WHILE-LOOP จะทำงานเป็นวงรอบไปเรื่อยๆ จนตราบเท่าที่ข้อแม้ของ boolean expression ใน WHILE-LOOP ยังคงมีค่าเป็น **TRUE** จำนวนครั้ง (รอบ) ที่ทำจะถูกควบคุมจากภายในตัวของ loop เอง นั้นคือการจะออกจากวงรอบประเภทนี้ได้ ก็ต่อเมื่อผลลัพธ์ของ boolean expression ได้ค่าเป็น **FALSE** ข้อแม้ที่ใช้ควบคุมการทำงานนี้ จะถูกตรวจสอบทุกครั้งก่อนการทำงานต่อไปภายในวงรอบ นั้นคือถ้าตรวจพบว่า boolean expression มีค่าเป็น FALSE เมื่อใดกลุ่มคำสั่งลำดับทุกๆ อันจะไม่ถูกทำงาน แต่ถ้าพบว่าค่า boolean expression มีค่าเป็น TRUE ทำให้กลุ่มคำสั่งลำดับภายในวงรอบทำงาน ทั้งนี้จะต้องไม่มีคำสั่งใดๆ ที่เป็นสาเหตุทำให้เกิดวงรอบของการทำงานที่ไม่สิ้นสุด (infinite loop) ตัวอย่างของ WHILE-LOOP ที่ถูกต้องแสดงให้เห็นในรูปที่ 6.13

```
p1:PROCESS (signal_a)
        VARIABLE index: INTEGER := 0;
    BEGIN
        from_in_to_out:     WHILE index < 8 LOOP
                                ray_out (index)     <=      ray_in (index);
                                        index       :=      index + 1;
                            END LOOP from_in_to_out;
    END PROCESS p1;
```

รูปที่ 6.13: **ตัวอย่างสำหรับการใช้** WHILE-LOOP

ตัวอย่างนี้เป็นการสร้างรูปแบบ (model) ที่ให้ผลลัพธ์ของการทำงานได้เช่นเดียวกับตัวอย่างในรูปที่ 6.11 ซึ่งนอกจากโครงสร้างที่ต่างกันแล้วยังมีข้อแตกต่างที่น่าสนใจดังนี้

1) ใน WHILE-LOOP จะไม่มีพารามิเตอร์ที่ถูกกำหนดแฝงไว้ เหมือนอย่างกรณีของ FOR-LOOP ถ้ามีความจำเป็นต้องใช้ จะต้องประกาศหรือกำหนดขึ้นมาเสียก่อน (ใน ต.ย. ได้แก่ตัวแปร *index*) ฉนั้นตัวแปรนี้จะต้องถูกแก้ไข ดัดแปลง ภายในวงรอบการทำงานด้วยคำสั่ง เช่น arithmetic operator เป็นต้น เพราะเหตุนี้เองที่ต้องมีการกำหนดตัวแปรเพิ่มขึ้น เนื่องจากจุดประสงค์ที่ต้องการเปลี่ยนแปลงค่า เพื่อกำหนดจำนวนรอบของวงรอบการทำงาน ฉนั้นในการใช้ WHILE-LOOP จึงไม่จำเป็นต้องเพิ่มหรือลดค่าตัวแปรที่ละ 1 เหมือนอย่างเช่นที่เกิดขึ้นอัตโนมัติใน FOR-LOOP) boolean expression ในตัวอย่างนี้ได้แก่ index < 8 หมายความว่าตราบที่ค่าของ index ยังน้อยกว่า 8 จะเกิดวงรอบการทำงานไปเรื่อยๆ เนื่องจากค่าเริ่มต้นของ index = 0 และภายในวงรอบจะเพิ่มขึ้นที่ละ 1 ดังนั้นจะเกิดวงรอบการทำงานทั้งหมด 8 ครั้ง สมมุติว่าไม่มีคำสั่ง index := index + 1 อยู่ภายใน loop จะทำให้เกิดวงรอบไม่รู้จบขึ้น

2) ในตัวอย่างที่แสดงในรูปที่ 6.13 การจำลองการทำงาน (simulation) จะเป็นไปตามความต้องการคือ เกิดวงรอบ 8 ครั้งสำหรับการจำลองครั้งแรกเท่านั้น เพราะหลังจากนั้นค่าของ index จะไม่มีโอกาสน้อยกว่า 8 เลย เนื่องจากว่าไม่มีการกำหนดค่าเริ่มต้นให้เป็น 0 ใหม่ นอกเสียจากว่าจะเป็นการเริ่มต้นการทำงานใหม่โดยการกระตุ้น PROCESS เพราะทุกครั้งที่ PROCESS ถูกกระตุ้น index จะถูกกำหนดค่าเริ่มต้นใหม่ (ในที่นี้ได้แก่ 0)

6.7 Assertion Statement

ในการจำลองการทำงานของรูปแบบ ที่เขียนขึ้นด้วยภาษา VHDL นั้น บางครั้งมีความจำเป็นที่ต้องการตรวจสอบการทำงานบางอย่าง อาทิเช่นระดับค่าของสัญญาณ ลักษณะการเปลี่ยนแปลง หรือพฤติกรรมต่างๆ ที่มีเวลาเข้ามาเกี่ยวข้องของสัญญาณ ตลอดจนการรายงานผลการคำนวณทางคณิตศาสตร์ว่ามี overflow และ underflow เป็นต้น เครื่องมือช่วยในภาษา VHDL ได้แก่ คำสั่ง ASSERT คำสั่งนี้จะตรวจสอบ boolean expression ถ้าผลลัพธ์จากการตรวจสอบเป็น **FALSE** จะแสดงรายงานของการตรวจพบให้ทราบ กฎเกณฑ์ของการเขียน assertion statement มีดังนี้

```
    ASSERT condition
[   REPORT string  ]
[   SEVERITY level    ] ;
```

รูปที่ 6.14: **โครงสร้างของ** Assertion statement

ข้อแม้ในการเตือน (assert condition) ในที่นี้ได้แก่ boolean condition ถ้าข้อแม้นี้ให้ผลลัพธ์เป็น FALSE จะมีผลทำให้ assert statement รายงานข้อผิดพลาดโดยแสดงให้เห็นที่เครื่องจำลอง ซึ่งจะต้องประกอบด้วยอย่างน้อยข้อมูลต่อไปนี้

1) ASSERT message identification
2) SEVERITY level ซึ่งค่าตายตัวได้แก่ ERROR
3) REPORT ลำดับตัวอักษร (string)
4) ชื่อของ design unit ที่ ASSERT statement อยู่

บรรทัดที่เป็น option ของ REPORT สามารถที่จะเขียนข้อความลงไปได้ เช่นเดียวกัน SEVERITY level (ถ้าไม่มีการกำหนดแน่นอนจะมีค่าตายตัวคือ ERROR) เป็น TYPE ชนิดหนึ่งที่ถูกกำหนดไว้ใน package ชื่อ STD (มาตรฐาน IEEE 1076 กำหนด)[5] ซึ่งมีด้วยกันสี่ค่าคือ NOTE, WARNING,

[5] ดูผนวก ค.

ERROR และ FAILURE รูปที่ 6.15 แสดงให้เห็นตัวอย่างการใช้ assert statement ในลักษณะของ sequential statement

ในรูปที่ 6.15 (a) assert statement จะทำงานทุกครั้ง ถ้าในขณะที่จำลองการทำงานของ model เมื่อ assert condition มีค่า FALSE นั่นคือ IF statement จะต้องทำงานก่อน ซึ่งหมายความว่าตรวจพบสัญญาณ ***clk*** มีค่าเป็น 'X' จริง ในรูปที่ 6.15 (b) นั้น ตราบเท่าที่ address ยังมีค่าน้อยกว่า 1024 จะให้ค่า assert condition เป็น TRUE จะไม่มีการทำงานของ assert statement แต่ถ้าเมื่อใด address มากกว่าหรือเท่ากับ 1024 เมื่อนั้น assert จะทำงาน การทำงานของ assert statement นั้นจะมีผลต่อการจำลองการทำงานหรือไม่นั้น ขึ้นอยู่กับ TYPE ของ SEVERITY ที่ตั้งไว้ในรูปแบบ ในกรณีที่ตั้งค่าไว้เป็น ERROR หรือ FAILURE เมื่อ assert statement ทำงานจะแสดง REPORT และหยุดการทำงานต่อไปของการจำลอง ในทางตรงข้ามถ้าตั้งค่าเป็น NOTE หรือ WARNING จะเป็นการแสดงเพียง REPORT แต่ขบวนการจำลองการทำงานยังคงทำงานต่อไป [6]

```
PROCESS (clk)
BEGIN
    IF clk = 'X' THEN
        ASSERT FALSE
        REPORT "Clock is unknow"
        SEVERITY ERROR ;
    END IF;
END PROCESS;
```

(a)

```
PROCESS (clk)
BEGIN
        ASSERT (address < 1024)
        REPORT "Address out of range!"
        SEVERITY WARNING ;
END PROCESS;
```

(b)

รูปที่ 6.15: **ตัวอย่างการใช้** assert statement **ในโครงสร้างแบบ** sequential

[6] Mentor Graphics, "An Introduction to Modeling in VHDL", Instructors Note, Revision 1.3, November 1991

รูปที่ 6.16 แสดงให้เห็นถึงตัวอย่างการใช้งานอีกลักษณะหนึ่งของ assert statement คือใช้สำหรับตรวจสอบเวลา

```
ASSERT (clock'LAST_EVENT >= 50 NS))
REPORT "Pulse width violation on clock!"
SEVERITY ERROR ;
```

รูปที่ 6.16: **ตัวอย่างการใช้** assert statement **ตรวจสอบช่วงเวลา**

ชุดคำสั่ง assert statement นี้จะตรวจสอบความกว้าง pulse ของสัญญาณ clock ว่ามากกว่าหรือเท่ากับ 50 ns. หรือเปล่า ถ้าเมื่อใดไม่เป็นเช่นนี้ (น้อยกว่า 50 ns.) จะแสดงรายงาน "Pulse width violation on clock!" พร้อมกับหยุดการจำลอง ในตัวอย่างนี้ใช้ตัวบอกคุณสมบัติสำหรับสัญญาณ (signal attribute) ซึ่งจะเป็นตัวที่ให้ข้อมูลเกี่ยวกับสัญญาณนั้นๆ ในที่นี้ได้แก่ **'LAST_EVENT** ตรวจสอบการเปลี่ยนแปลงระดับสัญญาณทุกครั้งที่เกิดขึ้น signal attribute นี้มีประโยชน์ในการเขียนรูปแบบมาก และจะศึกษารายละเอียดในบทที่ 9

6.8 สรุป

ในบทนี้ได้ศึกษาเกี่ยวกับชุดคำสั่งลำดับของภาษา VHDL และได้แสดงให้เห็นว่า ภาษา VHDL เป็นภาษาที่มีคุณสมบัติการทำงานลักษณะแข่งขนาน (concurrent) คือทุกชุดคำสั่งจะทำงานตลอดเวลาและอิสระต่อกันหรือมีผลกระทบถึงกันในเวลาเดียวกัน ฉนั้นการที่จะใช้ส่วนที่เป็นชุดคำสั่งลำดับใน VHDL จึงจำเป็นที่จะต้องบรรจุส่วนดังกล่าวไว้ในเปลือกของคำสั่งแบบแข่งขนาน (concurrent shell) ที่เรียกว่าชุดคำสั่ง process (PROCESS ... END PROCESS)

ชุดคำสั่งลำดับนั้นประกอบด้วยคำสั่ง IF-THEN-ELSE, LOOP และ ASSERT เป็นต้น กฎเกณฑ์การเขียน และตัวอย่างประกอบได้แสดงให้เห็นว่า โครงสร้างแบบลำดับไม่ได้มีความแตกต่างไปจากภาษาโปรแกรมชั้นสูงสำหรับคอมพิวเตอร์เท่าไร และการเขียนลักษณะนี้จะเป็นการบรรยายพฤติกรรมของ hardware ในลักษณะ behavioral

บทที่ 7
ชุดคำสั่งแบบแข่งขนาน (Concurrent Statement)

7.1 กล่าวนำ

ในบทนี้จะเป็นกล่าวถึงชุดคำสั่งแบบแข่งขนาน concurrent statement ของภาษา VHDL ซึ่งในบทที่แล้วได้อ้างถึงในบางส่วนเพื่อเป็นการเปรียบเทียบ ดังที่ทราบกันดีว่าภาษา VHDL เป็นภาษาที่มีการทำงานในลักษณะแข่งขนาน concurrency หรือสามารถที่จะมองชุดคำสั่งแบบแข่งขนาน (concurrent statement) แต่ละอันเป็น PROCESS ที่เชื่อมต่อกันด้วย signal แต่ละ PROCESS ทำงานอิสระไม่ขึ้นต่อกันแบบ หรือที่เรียกว่า asynchronous ดังนั้น concurrent statement ส่วนใหญ่จึงสามารถเขียนแทนได้ด้วย PROCESS statement

ชุดคำสั่งต่างๆ ที่เสนอไปในบทที่แล้วเป็นคำสั่งแบบ sequential ไม่สามารถที่จะใช้ในรูปของ ชุดคำสั่งแบบแข่งขนานได้ ทั้งนี้มีข้อยกเว้นบางคำสั่งที่สามารถใช้ใน VHDL ได้ทั้งสองรูปแบบ เช่น signal assignment เป็นต้น ในบทนี้เช่นเดียวกัน จะเป็นการแนะนำชุดคำสั่งที่ใช้ในโครงสร้างแบบ concurrent ซึ่งมีทั้งหมดได้แก่

1) Signal assignment statement
2) Component instantiation statement
3) Assert statement
4) Generate statement
5) Process statement
6) Procedure statement
7) Block statement

ซึ่งในบทนี้จะศึกษาเฉพาะสี่ชุดคำสั่งแรกเท่านั้น

7.2 Signal Assignment Statement

จนกระทั้งถึงบทนี้ในตัวอย่างต่างๆ ได้นำ signal assignment statement ไปใช้ค่อนข้างจะบ่อยครั้ง โดยที่ยังไม่ได้ศึกษาในรายละเอียด แต่ในลักษณะที่ใช้นั้นเป็นการใช้ในรูปแบบของ sequential statement หรือไม่ก็ concurrent แบบง่ายๆ ฉนั้นในบทนี้จึงจะขอลงไปในรายละเอียด ตลอดจนลักษณะของการใช้ที่แตกต่างออกไป signal assignment มีกฎเกณฑ์ในการเขียนดังนี้

```
target    <=    [delay_options] straight_transforms or
                                conditional_transforms or
                                selected_transforms ;
```

รูปที่ 7.1: **โครงสร้างของ** signal assignment statement

ในส่วนของ delay_option นั้นมีอยู่สองประเภทได้แก่ INERTIAL และ TRANSPORT ซึ่งได้กล่าวไปแล้วในบทที่ 1 และถ้าไม่มีการกำหนด option ของ delay นี้ ชุดคำสั่งจะกำหนดให้เป็น inertial delay เป็นค่าตายตัว

จากคำจำกัดความของ concurrent statement นั้น signal assignment จะเป็น PROCESS ในตัวของมันเอง ทั้งนี้ขึ้นอยู่กับลักษณะของการใช้เป็นคำสั่งในการส่งผ่านค่าของสัญญาณ ว่าจะเป็น straight_transforms, conditional_transforms หรือ selected_transform ฉนั้น signal assignment จึงสามารถเขียนแทนได้ด้วย PROCESS เดี่ยวๆ หรือ PROCESS ร่วมกับ IF-THEN-ELSE หรือ CASE statement ตามลำดับ ดังที่แสดงในรูปที่ 7.2

```
ARCITECTURE arch1 OF and2 IS
BEGIN
  output <= in1 AND in2;
END arch1;
```

```
ARCITECTURE arch2 OF and2 IS
BEGIN
    PROCESS (in1, in2)
    BEGIN
        output <= in1 AND in2;
    END PROCESS;
END arch2;
```

รูปที่ 7.2: Concurrent statement **และ** PROCESS statement

ดังจะเห็นได้ชัดจากหน่วยที่เป็น architecture ทั้งสอง ว่าสัญญาณ ***output*** ได้รับค่าจากผลลัพธ์ของการทำ boolean operator "AND" ระหว่าง ***in1*** กับ ***in2*** เช่นเดียวกันการที่ใน PROCESS มี sensitivity list (ในตัวอย่าง *in1* และ *in2*) เพื่อใช้กระตุ้นและควบคุมการทำงาน ในโครงสร้างของ concurrent สัญญาณใดๆ ที่อยู่ทางขวามือของเครื่องหมาย signal assignment (<=) จะทำหน้าที่กระตุ้นการทำงานของ statement เมื่อเกิด event ขึ้นกับสัญญาณตัวใดตัวหนึ่ง (ในตัวอย่าง ***in1*** และ ***in2***) หรือเรียกได้ว่าเป็น sensitivity list ของ concurrent statement ต่อไปนี้จะศึกษาลักษณะการใช้ signal assignment

- ***Straight transforms***

```
ARCHITECTURE and2 OF and2 IS
BEGIN
output <= in1 AND in2 AFTER 10 NS;
END and2;
```

ตัวอย่างของการใช้ signal assignment แบบ straight transforms ใช้เป็นรูปแบบ (model) ของ AND-gate 2 inputs และมี (inertial) delay เท่ากับ 10 ns.

- ***Conditional transforms (conditional signal assignment)***

```
ARCHITECTURE rtl OF mux IS
    FUNCTION rising (sig: BIT) RETURN BOOLEAN IS
    BEGIN
        IF sig := '0' THEN
            RETURN FALSE;
        ELSE
            RETURN TRUE;
        END IF;
    END rising;
    FUNCTION falling (sig: BIT) RETURN BOOLEAN IS
    BEGIN
        IF sig = '0' THEN
            RETURN TRUE;
        ELSE
            RETURN FALSE;
        END IF;
    END falling;
BEGIN
    output <=in1 AFTER tplh     WHEN sel = '0' AND rising   (in1)  ELSE
                in1 AFTER tphl     WHEN sel = '0' AND falling  (in1)  ELSE
                in2 AFTER tplh     WHEN sel = '1' AND rising   (in2)  ELSE
                in2 AFTER tphl     WHEN sel = '1' AND falling  (in2)  ELSE
                '0' AFTER tphl;
END rtl;
```

ตัวอย่างการใช้ signal assignment แบบ conditional transform โดยใช้รูปแบบ model ของ 2:1 multiplexer ชุดคำสั่ง **WHEN-ELSE** แสดงให้เห็นถึงการใช้แบบ conditional transform ในตัวอย่างนี้เปรียบได้กับการใช้ PROCESS ร่วมกับ IF-THEN-ELSE

- ***Selected transforms (selected signal assignment)***

```
ENTITY mux IS
    PORT (   input: IN BIT_VECTOR (0 TO 3);
             sel: IN BIT_VECTOR (0 TO 1);
             output: OUT BIT     );
END mux;
ARCHITECTURE simple_con OF mux IS
BEGIN
    WITH sel SELECT
        output <=input (0)  AFTER 5 NS    WHEN "00",
                 input (1)  AFTER 5 NS    WHEN "01",
                 input (2)  AFTER 5 NS    WHEN "10",
                 input (3)  AFTER 5 NS    WHEN "11";
END simple_con;
```

ในตัวอย่างข้างบนแสดงให้เห็นรูปแบบ (model) ของ 4:1 multiplexer ในลักษณะของ selected transforms ของ concurrent signal assignment โดยใช้โครงสร้าง **WITH-SELECT-WHEN** เนื่องจากสัญญาณ **input** และ **sel** อยู่ทางขวามือของเครื่องหมาย signal assignment จึงทำหน้าที่เหมือนกับ sensitivity list ใน process statement ที่ใช้ case construct ดังที่แสดงในรูปต่อไป

```
ARCHITECTURE simple_seq OF mux IS
BEGIN
    PROCESS (input, sel)
    BEGIN
        CASE sel IS
            WHEN "00"    =>    output    <=    input (0)  AFTER 5 NS;
            WHEN "01"    =>    output    <=    input (1)  AFTER 5 NS;
            WHEN "10"    =>    output    <=    input (2)  AFTER 5 NS;
            WHEN "11"    =>    output    <=    input (3)  AFTER 5 NS;
        END CASE;
    END PROCESS;
END simple_seq;
```

7.3 Component Instantitation Statement

ชุดคำสั่ง component instantitation statement เป็นเครื่องมือที่ช่วยการเขียนรูปแบบบรรยายระบบดิจิตอลด้วยภาษา VHDL ในลักษณะของ hierarchical เป็นชุดคำสั่งที่อนุญาตให้นำ entity design unit ที่ผ่านการวิเคราะห์และตรวจสอบความถูกต้องแล้ว มาใช้เป็นอุปกรณ์ในรูปแบบแทนการเขียนด้วยการบรรยายแบบ behavioral รวมทั้งการเชื่อมต่อระหว่างกัน เปรียบเสมือนกับการออกแบบโดยใช้ schematic capture ในลักษณะการออกแบบ Bottom-Up และถ้าจำเป็นสามารถที่จะเปลี่ยนแปลงค่า generic ที่กำหนดไว้ใน entity design unit ของอุปกรณ์ที่นำมาใช้ได้ กฎเกณฑ์การเขียนมีดังนี้

```
label: component_name
        [GENERIC MAP (association_list)]
        [PORT MAP (association_list)] ;
```

รูปที่ 7.3: **โครงสร้างของ** component instantitation statement

สิ่งที่ VHDL ต้องการจากชุดคำสั่งนี้คือ **label** และ **component_name** ส่วน **GENERIC MAP** นั้นเป็นเพียง option ไม่จำเป็นต้องกำหนดลงไปถ้าในกรณีที่ entity design unit ของอุปกรณ์ไม่มีการกำหนด generic ไว้ หรือถ้ามีการกำหนดแต่ไม่มีความจำเป็นที่จะกำหนดค่าใหม่แทน (override) ค่าตายตัวของอุปกรณ์ เช่นเดียวกันกับ PORT MAP ที่เป็น option ไม่จำเป็นต้องกำหนดถ้าไม่ต้องการที่จะเชื่อมอุปกรณ์ (entity design unit) กับอุปกรณ์อื่นๆ หรือเป็นอุปกรณ์ที่ไม่มีช่องทาง เข้า-ออก (port)

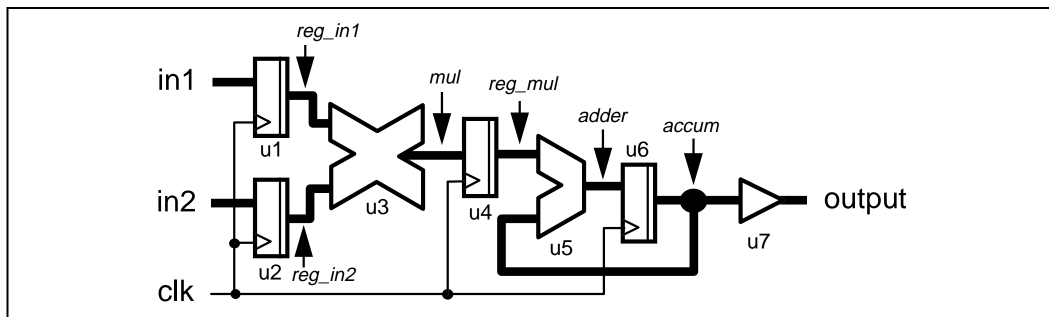
ดังตัวอย่างที่แสดงเพื่อให้เห็นข้อเปรียบเทียบ ระหว่าง behavioral description กับ structural description ในบทที่ 4 ได้นำมาแสดงให้เห็นอีกครั้งในรูปที่ 7.4

ก่อนที่จะกล่าวต่อไปในรายละเอียดของคำสั่ง component instantiation statement มีความจำเป็นที่จะต้องกลับมาศึกษาถึงส่วนที่เรียกว่า component declaration statement อีกครั้ง การประกาศหรือกำหนดอุปกรณ์นั้น สามารถกระทำได้ในส่วน declarative area ของ architecture body หรืออาจจะเก็บไว้ใน package แล้วทำให้เข้าสู่ส่วนนี้ได้ โดยการใช้ use statement ผ่าน library

การประกาศหรือกำหนดอุปกรณ์ (component declaration) นั้นเป็นการกำหนดเพียงส่วนที่เป็น entity ของอุปกรณ์ ไม่ได้มีส่วน architecture ร่วมอยู่ด้วย ถ้ามีความต้องการที่จะประกาศหรือกำหนด architecture ที่บรรยายพฤติกรรมของ entity นั้นๆ สามารถกระทำได้โดยใช้การกำหนดโครงร่าง (configuration specification) ซึ่งจะเป็นตัวเชื่อม architecture เข้ากับ entity เหตุที่เป็นเช่นนี้เพราะ entity design unit หนึ่งสามารถมี architecture ได้หลายรูปแบบ จึงจำเป็นต้องเลือกใช้เพียงอันใดอันหนึ่งสำหรับจำลองการทำงาน configuration specification จะกล่าวโดยละเอียดอีกครั้งในบทที่ 9

```
ARCHITECTURE structure OF mac IS
    COMPONENT reg
    GENERIC (width: INTEGER:= 16);
    PORT (    d: IN BIT_VECTOR ((width - 1) DOWNTO 0);
              clk: IN BIT;
              q: OUT BIT_VECTOR ((width - 1) DOWNTO 0) );
    END COMPONENT;
    COMPONENT adder
    PORT (    port1, port2: IN BIT_VECTOR (31 DOWNTO 0);
              output: OUT BIT_VECTOR (31 DOWNTO 0) );
    END COMPONENT;
    COMPONENT multiply
    PORT (    port1, port2: IN BIT_VECTOR (15 DOWNTO 0);
              output: OUT BIT_VECTOR (31 DOWNTO 0) );
    END COMPONENT;
    COMPONENT buf
    PORT (    input: IN BIT_VECTOR (31 DOWNTO 0);
              output: OUT BIT_VECTOR (31 DOWNTO 0) );
    END COMPONENT;

    SIGNAL  reg_in1, reg_in2: BIT_VECTOR (15 DOWNTO 0);
    SIGNAL  mul, mul_reg, adder, accum: BIT_VECTOR (31 DOWNTO 0);

BEGIN
    u1: reg GENERIC MAP (16) PORT MAP (in1, clk, reg_in1);
    u2: reg GENERIC MAP (16) PORT MAP (in2, clk, reg_in2);
    u3: multiply PORT MAP (reg_in1, reg_in2, mul);
    u4: reg GENERIC MAP (32) PORT MAP (clk => clk, D => mul, Q => reg_mul);
    u5: adder PORT MAP (reg_mul, accum, adder);
    u6: reg GENERIC MAP (32) PORT MAP (adder, clk, accum);
    u7: buf PORT MAP (accum, out1);
END structure;
```

รูปที่ 7.4 (a): Component instantiation **บรรยาย** Multiply-Accumulate Unit



รูปที่ 7.4 (b): Component instantiation **บรรยาย** Multiply-Accumulate Unit

จากรูปที่ 7.4 จะเห็นได้ว่าในส่วนของโครงสร้างของ architecture มีอุปกรณ์ทั้งหมด 7 ชิ้น ที่มีชื่อ (label) **u1**, **u2**, **u3**, **u4**, **u5**, **u6** และ **u7** หลังจาก label ตามด้วยอุปกรณ์ที่ใช้ (**reg**, **multiply**, **adder** และ **buf**) คำสั่ง PORT MAP นั้นเป็นการต่ออุปกรณ์เข้าด้วยกัน (แทนการ wired ในระบบของ schematic capture) ส่วนคำสั่ง GENERIC MAP อนุญาตให้เขียนค่า generic ใหม่ทับค่าตายตัว (default value) ที่ติดมากับอุปกรณ์ย่อยที่นำมาใช้ ดังนั้นโครงสร้างลักษณะนี้สามารถเรียกได้ว่าเป็น VHDL netlist

คำสั่ง PORT MAP สำหรับอุปกรณ์ **u4** มีการเขียนที่แตกต่างไปจากอุปกรณ์อื่นๆ ที่เหลือ การอ้างถึงความสัมพันธ์ (เชื่อมต่อกัน) ลักษณะนี้เรียกว่า named association ส่วนอุปกรณ์ที่เหลืออีก 6 ชิ้น นั้นเป็นการอ้างถึงความสัมพันธ์ลักษณะ position association ลักษณะตายตัวของการอ้างถึงความสัมพันธ์จะเป็นแบบ position association นั้นคือสัญญาณตัวแรกใน PORT MAP จะเชื่อมต่อกันกับสัญญาณตัวแรกที่กำหนดไว้ใน PORT ของอุปกรณ์ แต่ก็สามารถที่จะบังคับการเชื่อมต่อใหม่ได้โดยการใช้ named association เชื่อมต่อ การทำ named association นั้นใช้เครื่องหมาย **'=>'** เป็นตัวกำหนด

7.4 Assert Statement

ชุดคำสั่งนี้ได้ศึกษามาแล้วในบทที่ 6 ซึ่งเป็นการใช้งานในลักษณะของ sequential statement แต่ assert statement สามารถที่จะใช้ใน concurrent statement ได้เช่นเดียวกับ signal assignment

statement ดังนั้นในบทนี้จะแสดงให้เห็นถึงอีกลักษณะหนึ่งของการใช้ คือการใช้แบบ concurrent statement

ถ้ากลับไปดูในบทที่ 6 แทนที่จะใช้ assert statement ภายใน PROCESS ในลักษณะของ sequential สามารถที่จะใช้ในลักษณะแบบแข่งขนาน (concurrent) ได้ด้วยเพียงคำสั่งเดียวเท่านั้น ซึ่งจะให้ผลลัพธ์ที่เหมือนกัน สัญญาณทุกตัวที่อยู่ใน assert statement แบบ concurrent จะเป็นตัวกระตุ้นการทำงาน และจะเกิดขึ้นได้ก็ต่อเมื่อ boolean expression ให้ค่าเป็น **FALSE**

จากตัวอย่างในรูปที่ 7.5 ถ้าเกิด event ขึ้นกับสัญญาณ clk หรือ address และ boolean expression ให้ค่าเป็น FALSE เมื่อไร assert statement จะทำงานโดยแสดงข้อความของ REPORT และ SEVERITY level

```
ASSERT (clk /= 'X')         REPORT "Clock is unknow"
                            SEVERITY ERROR;


ASSERT (address < 1024)     REPORT "Address out of range!"
                            SEVERITY WARNING;
```

รูปที่ 7.5: Concurrent ASSERT statement

7.5 Generate Statement

ในระบบออกแบบวงจรอิเล็กทรอนิกส์อัตโนมัติ หรือ (Electronics Design Automation หรือ EDA) วิศวกรออกแบบสามารถป้อนรูปสัญลักษณ์ของอุปกรณ์อิเล็กทรอนิกส์ต่างๆ (schematic capture) เพียงชิ้นเดียว แล้วเติมคุณสมบัติบางอย่าง เพื่อบอกให้เครื่องสร้างอุปกรณ์ชิ้นนั้นซ้ำเท่ากับจำนวนที่ต้องการ การขยายเพิ่มเติมนี้เป็นหน้าที่ของ schematic compiler การทำเช่นนี้เป็นการทำงานในระดับบนของการออกแบบในชั้นของ gate-level ตัวอย่างที่เห็นได้ชัดอีกอันได้แก่ ถ้าต้องการต่อ bus ขนาด 16 bits เข้ากับ register ที่มีขนาด 16 bits เช่นกัน วิศวกรออกแบบสามารถใช้วิธีการต่อสายตัวนำ 1 เส้นเข้ากับ flip-flop 1 ตัว แล้วบอกเครื่องให้ขยายสายตัวนำ และจำนวน register ให้เป็น 16 ก็จะได้การต่อตามต้องการ โดยที่ไม่ต้องทำการต่อสายตัวนำที่ละเส้นเข้ากับ register ที่ละตัว การขยายเช่นนี้เรียกว่า "iterative elaboration"

อีกลักษณะหนึ่งของการทำงานที่เรียกว่า conditional elaboration ซึ่งเป็นลักษณะหนึ่งของ generate statement ใน VHDL จะมีลักษณะคล้ายกับ switches หรือ compiler directives ที่มีอยู่ในภาษา C หรือ PASCAL ตัวกำกับนี้จะบอก compiler ให้ทำงานภายใต้ข้อแม้ สำหรับ (conditional) generate statement ในระหว่างการวิเคราะห์จะถูก compile โดยไม่คำนึงถึง condition แต่ในระหว่างการจำลองการทำงานถ้า condition เป็น FALSE เมื่อไร พฤติกรรม หรือ statement ต่างๆ ที่บรรยายไว้ใน generate statement จะถูกยกเลิก รูปที่ 7.6 เป็นโครงสร้างของการเขียน generate statement

```
generate_label: generate_scheme  GENERATE
                      {concurrent_statements}
                    END GENERATE [generate_label] ;

                    generate_scheme ::=

                    FOR generate_specification
                    |   IF condition
```

รูปที่ 7.6: **โครงสร้างของ** generate statement

คำ GENERATE แสดงจุดเริ่มต้น และ END GENERATE แสดงจุดสิ้นสุดของ statement ในคำสั่งนี้ต้องมี label กำกับ ส่วน label ตำแหน่งสุดท้าย (END GENERATE) เป็น option แต่ถ้าจะเขียนต้องเขียนชื่อเดียวกับ generate_label

เนื่องจากมีสองรูปแบบของ generation scheme ของ generate statement อันได้แก่ FOR generation (iterative) และ IF generation (conditional) ดังนั้นการทำงานของคำสั่งนี้จะออกมาเป็นผลลัพธ์ของพฤติกรรมที่ได้จะเป็นการทำวนซ้ำ หรือทำงานภายใต้ข้อแม้ ขึ้นอยู่กับว่าใช้รูปแบบใด ถ้าเป็นรูปแบบ FOR จะเป็นการทำวนซ้ำ แต่ถ้าเป็นรูปแบบ IF จะทำงานภายใต้ข้อแม้ ในหนังสือเล่มนี้จะบรรยายเฉพาะรูปแบบ FOR ก่อน ซึ่งแสดงให้เห็นตัวอย่างใน รูปที่ 7.7

```
ENTITY reg_16 IS
    PORT (   input: IN BIT_VECTOR (0 TO 15);
             clock: IN BIT;
             output: OUT BIT_VECTOR (0 TO 15)  );
END reg_16;

ARCHITECTURE struct OF reg_16 IS
    COMPONENT dff
        PORT d, clk: IN BIT; q: OUT BIT);
    END COMPONENT;
BEGIN
    g1: FOR i IN 0 TO 15 GENERATE
        g1: dff PORT MAP (input(i), clock, output (i));
    END GENERATE g1;
END struct;
```

รูปที่ 7.7: VHDL model **ของ** 16 bits register

ในตัวอย่างได้แสดงให้เห็น VHDL model ที่ใช้แทนการป้อนรูปอุปกรณ์ ซึ่งในที่นี้ได้แก่ อุปกรณ์ 16 bits register โดยไม่ต้องใช้วิธีกำหนดจำนวนอุปกรณ์ flip-flop ถึง 16 ครั้ง แต่ใช้ generate statement ในรูปแบบของ FOR generation สร้างรูปแบบซ้ำของ D-FlipFlop

สิ่งที่พึงสังเกตอย่างหนึ่งคือ ตัวแปร (index) ที่เป็นตัวนับจำนวนครั้งของการทำงาน "i" ไม่จำเป็นต้องประกาศใช้ (declaration) และตัวแปรนี้จะเพิ่มค่าครั้งละ 1 ต่อการทำงานของ generate statement เมื่อครบ 16 ครั้ง การทำงานจะหยุด

7.7 สรุป

ในบทนี้ได้ศึกษาถึง concurrent statement ที่มีอยู่ด้วยกัน 7 อย่างในภาษา VHDL จนจบบทนี้ได้รู้จักแล้ว 5 คำสั่ง โดยที่ในบทนี้ได้เรียน 4 คำสั่ง และอีกคำสั่งได้ศึกษาในบทก่อน ซึ่งได้แก่ PROCESS นั้นเอง

เช่นเดียวกันกับ process statement ที่ต้องมีสัญญาณที่ใช้ควบคุมการทำงานในชุดคำสั่งแข่งขนาน concurrent statement ก็เช่นกัน สัญญาณที่อยู่ทางขวามือของเครื่องหมาย signal assignment หรืออยู่ใน PORT MAP หรือใน GENERATE จะทำหน้าที่เช่นเดียวกับ sensitivity list ใน PROCESS ดังนั้น concurrent statement ส่วนใหญ่จึงเขียนแทนได้ด้วย process statement

บทที่ 8
โปรแกรมย่อย (Subprogram)

8.1 กล่าวนำ

ความหมายของโปรแกรมย่อย หรือ subprogram คือกลุ่มของคำสั่งลำดับ (sequential statement) ที่สามารถเรียกหรืออ้างอิงได้จากทุกตำแหน่งของรูปแบบ หน้าที่ของโปรแกรมย่อย คือรวบรวมคำสั่งแบบลำดับเหล่านั้นเข้าไว้ด้วยกัน เพื่อเป็นการลดการเขียนชุดคำสั่งซ้ำกันหลายๆ ครั้งขึ้นในรูปแบบเดียวกัน

ผู้ออกแบบที่เขียนรูปแบบ VHDL บรรยายการทำงานของระบบดิจิตอล จะพบได้เสมอว่าภายในรูปแบบหนึ่ง บางครั้งมีกลุ่มของลำดับคำสั่งเดียวกันปรากฏในตำแหน่งต่างๆ ของรูปแบบ ซึ่งในกรณีเช่นนี้สามารถที่จะแยกกลุ่มของคำสั่งเหล่านั้นออกเป็นโปรแกรมย่อยเพียงชุดเดียวแทน และสามารถที่จะอ้างอิง หรือเรียกมาใช้ (call) ได้จากทุกๆ ตำแหน่งของรูปแบบได้ การทำเช่นนี้มีผลให้ความยาวของรูปแบบที่เขียนขึ้นลดลงจากเดิม (ก่อนการใช้โปรแกรมย่อยช่วย) ซึ่งจะทำให้ง่ายต่อการอ่านทบทวนและตรวจทาน นอกจากนั้นยังมีผลทำให้ลดความซับซ้อนของรูปแบบลง

ฉนั้นโปรแกรมย่อยก็คือการรวบรวมชุดคำสั่งที่เป็นลำดับเข้าไว้ด้วยกัน และสามารถที่จะถูกเรียกใช้ได้จากรูปแบบหลัก โดยการกำหนดชื่อให้กับโปรแกรมย่อย สิ่งที่ต้องจำไว้อย่างหนึ่งคือคำสั่งแบบแข่งขนานไม่สามารถที่จะปรากฏอยู่ในโครงสร้างของโปรแกรมย่อยได้ แต่ในขณะที่โปรแกรมย่อยนั้นสามารถที่จะถูกเรียกได้จากทั้งส่วนที่เป็นคำสั่งแบบลำดับ และคำสั่งแบบแข่งขนานภายในรูปแบบ

ในภาษา VHDL นั้นแบ่งโปรแกรมย่อยออกได้เป็นสองประเภทคือฟังก์ชัน (function) และโปรซีเยอร์ (procedure) ดังนั้นในบทนี้จะเป็นการกล่าวถึงรายละเอียดข้อแตกต่างระหว่างโปรแกรมย่อยทั้งสองประเภท วิธีการประกาศโปรแกรมย่อย (subprogram declaration) ตลอดจนผลของการประกาศที่มีต่อขอบเขตของการใช้โปรแกรมย่อย และการเรียกโปรแกรมย่อย (subprogram call)

8.2 ฟังก์ชัน (Function)

ฟังก์ชันคือลักษณะหนึ่งของโปรแกรมย่อย เมื่อถูกเรียกใช้ค่าของ object บางตัวในรูปแบบ จะถูกส่งผ่านไปยังฟังก์ชันซึ่งตัว object นี้ถูกเรียกว่า "passing parameter"[1] ลักษณะการทำงานของฟังก์ชันจะทำงานโดยไม่มีการนำค่าที่ได้รับจาก passing parameter มาเปลี่ยนแปลงภายในตัวของฟังก์ชันแต่โครงสร้างภายในซึ่งประกอบด้วยชุดคำสั่งลำดับต่างๆ นั้นจะนำค่าของ object ที่ได้รับผ่านเข้ามาไปใช้คำนวณ เพื่อให้ได้ผลลัพธ์อย่างใดอย่างหนึ่งออกมา ค่าใหม่ที่ได้รับ (ผลลัพธ์จากการคำนวณ) จะถูกส่งกลับไปยังรูปแบบ ตรงตำแหน่งที่ฟังก์ชันนั้นถูกเรียกใช้หรืออ้างอิง ฟังก์ชันมีคุณสมบัติที่เป็นเอกลักษณ์ดังนี้

1) ทำงานในลักษณะของ expression ไม่ใช่ statement

ตัวอย่าง: signal_a <= **function_a** (signal_b, signal_c);

Function ถูกเรียกด้วยชื่อของฟังก์ชันเองคือ function_a การใช้ชื่อของฟังก์ชันร่วมกับชื่อของ object ได้แก่ signal_b, signal_c ใน statement นั้นเป็น expression (ที่ประกอบกับส่วนอื่นๆ เป็นชุดคำสั่ง)

---

[1] Mentor Graphics, An Introduction to Modeling in VHDL, Instructor Notes, Revision 3.1, November 1991

2) **FUNCTION จะคำนวณให้ผลลัพธ์เพียงค่าเดียวและส่งกลับคืน model**
เมื่อฟังก์ชันทำงานเรียบร้อยแล้ว ค่าเพียงค่าเดียวเท่านั้นที่ถูกส่งกลับไปยังตำแหน่งของ statement ที่มีการเรียกใช้ FUNCTION

3) **ภายในฟังก์ชันจะต้องมีคำสั่ง RETURN**
คำสั่งนี้มีหน้าที่กำหนดตำแหน่งที่ฟังก์ชันทำงานสมบูรณ์แล้วส่งผลลัพธ์ที่เป็นค่าๆ เดียวกลับ

4) **การทำงานของฟังก์ชันเองจะต้องไม่มีผลกระทบต่อค่าของ object ที่ถูกส่งผ่านเข้าไป (passing parameters)**
ความหมายของข้อกำหนดนี้คือ ค่าของ object ที่อยู่ในรูปของการเรียกใช้ฟังก์ชันดังในตัวอย่างคือ signal_b, signal_c ห้ามถูกดัดแปลง แก้ไข

5) **FUNCTION จะต้องประกอบด้วย function body และอาจจะมีส่วนประกาศ (function declaration)ได้**

8.2.1 Function Declaration

โดยทั่วไปฟังก์ชันจะประกอบด้วยสองส่วนคือ function declaration และ function body ในส่วนของ function body นั้นเป็นสิ่งที่ต้องมี (หมายถึงกรณีที่มีการกำหนด FUNCTION) ส่วน function declaration นั้นจะมีหรือไม่มีก็ได้ รูปที่ 8.1 แสดงให้เห็นรูปแบบการเขียน function declaration ในภาษา VHDL

```
FUNCTION name [(formal_parameter_list)] RETURN TYPE;
```

รูปที่ 8.1: **รูปแบบการเขียน** function declaration

การเขียน function declaration เป็นการให้ข้อมูลรายละเอียดของฟังก์ชันได้แก่

1) **ชื่อของ function**: ชื่อของฟังก์ชันจะถูกใช้ในการเรียกให้ทำงาน ในที่นี้คือ name

2) **formal parameter list**: จะเป็นรายชื่อของพารามิเตอร์ที่ใช้ในฟังก์ชันจะเป็นตัวรับค่าที่ผ่านให้กับฟังก์ชันจากจุดตำแหน่งที่ถูกเรียก สิ่งที่สามารถกำหนดเพิ่มเติมได้แก่คุณสมบัติของพารามิเตอร์เช่น CLASS, ชื่อ (name), MODE และ TYPE เช่นการเขียน

   (SIGNAL signal_a : IN BIT)

   มีความหมายว่าพารามิเตอร์มีชั้น (CLASS) ประเภท SIGNAL ชื่อ signal_a มี MODE เป็น IN (input) และ TYPE ประเภท BIT เป็น parameter list ชื่อของพารามิเตอร์ไม่จำเป็นต้องเป็นชื่อเดียวกับ object ของรูปแบบที่จะส่งค่าให้

3) **คำ RETURN**: ในที่นี้ไม่ใช้คำสั่ง (statement) แต่มีหน้าที่กำหนด TYPE ของค่าที่จะส่งกลับ TYPE จะเป็นตัวบอกกลุ่มของค่าที่สามารถส่งกลับจากฟังก์ชันได้เช่น

   RETURN BIT

   หมายความว่าค่าที่จะได้รับจากฟังก์ชันได้นั้นเป็นเพียงกลุ่มของค่า '0' หรือ '1' เท่านั้น

ดังที่กล่าวไว้ข้างต้นว่า function declaration ไม่จำเป็นต้องมีเพราะฟังก์ชันที่ไม่มีส่วนนี้สามารถที่จะทำงานได้เช่นกัน แต่ในบางกรณีที่มีความจำเป็นจะต้องมีส่วนนี้เมื่อ

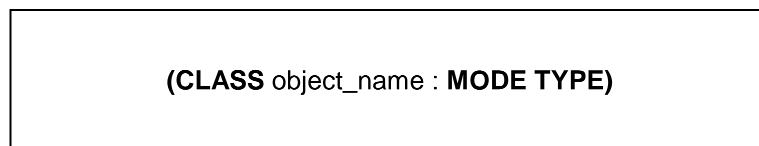
- ☑ เมื่อฟังก์ชันเป็น recursive function หมายความว่าในกรณีที่มีการอ้างถึงฟังก์ชันนั้นจากภายในตัวโครงสร้างของฟังก์ชันเอง (เรียกตัวเอง)

- ☑ เมื่อมีตำแหน่งที่เรียกฟังก์ชันอยู่ก่อนที่จะถึงตำแหน่งที่ function body อยู่ ดังนั้นจึงต้องถูกประกาศในพื้นที่สำหรับเขียนประกาศของ architecture design unit (ระหว่างคำว่า ARCHITECTURE กับ BEGIN)

สิ่งที่สำคัญในการสร้างฟังก์ชันคือถ้ามีส่วนประกาศฟังก์ชัน(function declaration) จะต้องประกาศในพื้นที่เดียวกันกับส่วนที่เป็น function body อยู่ (ข้อกำหนดนี้มีข้อยกเว้นในกรณีที่สร้าง PACKAGE สำหรับเก็บโครงสร้างของฟังก์ชันคือ function body จะอยู่ใน package declaration และ function body จะอยู่ใน package body) และ function declaration จะต้องมาก่อน function body

8.2.2 Formal Parameter List

ในหัวข้อที่ผ่านมาได้กล่าวถึง formal parameter list อย่างย่อไปแล้ว หน้าที่ของ parameter list จะเป็นตัวบอกถึง CLASS, NAME (identifier), MODE และ TYPE ของ object ที่มีค่าสำหรับส่งให้กับฟังก์ชันมีรูปแบบการเขียนดังในรูปที่ 8.2



รูปที่ 8.2: **รูปแบบการเขียน** formal parameter list

จากรูปที่ 8.2 สามารถได้ข้อมูลเกี่ยวกับตัว parameter ดังนี้

- **CLASS**: หมายถึงชั้นของ object ซึ่งอาจจะเป็น SIGNAL หรือ CONSTANT ส่วนตัวแปร (variable) ไม่สามารถเป็น parameter ในฟังก์ชันได้ ถ้าไม่มีการกำหนด CLASS ของ object ให้ VHDL จะกำหนด (ค่าตายตัว) ให้เป็น CONSTANT (ค่าตายตัว)

- **object_name**: กำหนดโดยผู้ออกแบบ สามารถเป็นชื่อใดๆ ได้ที่เป็นไปตามกฎของภาษา VHDL และชื่อที่ตั้งขึ้นนี้ไม่จำเป็นต้องเป็นคล้องจอง หรือชื่อเดียวกันกับ object อื่นๆ ใน model

- **MODE**: กำหนดทิศทางการไหลของข้อมูล โดยคำนึงถึงหลักการของฟังก์ชันมีเพียง MODE เดียวเท่านั้นที่สามารถใช้ได้คือ IN และจะเป็นค่าตายตัว ถ้าไม่มีการเขียนกำหนดไว้

- **TYPE**: จะเป็นตัวกำหนดกลุ่มของค่า (value set) ต่างๆ ที่ object สามารถจะมีได้ ค่าใดๆ ที่ไม่อยู่ในกลุ่มของค่า ไม่สามารถที่จะส่งผ่านจากรูปแบบ (model) ไปยังฟังก์ชันได้ แต่อย่างไรก็ตาม TYPE ของพารามิเตอร์ใน formal parameter list จะต้องเป็นประเภทเดียวกันกับ TYPE ของ object ที่นำมาจากรูปแบบ (model) เพื่อส่งผ่านไปยัง FUNCTION

นอกจากนั้นหน้าที่สำคัญอีกอย่างของ formal parameter list คือ เป็นสื่อรับค่า (value) ต่างๆ ที่ส่งให้กับฟังก์ชันเพื่อนำไปคำนวณภายใน [2]

8.2.3 RETURN TYPE

จะเป็นตัวกำหนดประเภทกลุ่มของค่า (value set) ของผลลัพธ์ที่ฟังก์ชันจะส่งกลับ มีรูปแบบการเขียนตามรูปที่ 8.3

รูปที่ 8.3: **รูปแบบการเขียน** RETURN TYPE

คำนี้ไม่ใช่ตัวกำหนดเฉพาะว่าจะเป็นค่าอะไรที่จะส่งกลับ แต่จะกำหนดกลุ่มของค่าที่ผลลัพธ์จะมีค่าได้สำหรับการส่งกลับ นั้นหมายความว่าค่าที่จะส่งกลับต้องมี TYPE ตามที่กำหนดไว้

---

[2] Formal parameter list ใน FUNCTION นั้นสามารถเปรียบเทียบได้กับ object ต่างๆ ที่ประกาศโดยคำสั่ง PORT ใน entity design unit

คำสั่งนี้จะต้องกำหนดทุกครั้ง เพราะไม่มีค่าตายตัว (default value) การตัดสินใจเลือกกลุ่มของค่านี้สำคัญมาก เนื่องจาก TYPE ที่เลือกจะต้องคล้องจองกับการใช้ เช่น TYPE ประเภท BIT ไม่สามารถที่จะส่งกลับไปยังตำแหน่งที่เรียกฟังก์ชันที่ถูกกำหนดให้มี TYPE เป็น BOOLEAN

รูปที่ 8.4 เป็นตัวอย่างการเขียน function declaration ที่บรรจุอยู่ใน package declaration

```
PACKAGE util IS
  FUNCTION vect_to_int (vect : BIT_VECTOR (0 TO 3)) RETURN INTEGER;
  FUNCTION rising (SIGNAL sig : BIT) RETURN BOOLEAN;
  FUNCTION falling (SIGNAL sig : BIT) RETURN BOOLEAN;
END util;
```

รูปที่ 8.4: **ตัวอย่างการเขียน** function declaration

8.2.4 Function Body

ส่วนนี้เป็นส่วนที่บรรยายพฤติกรรมของฟังก์ชันและจะต้องมีส่วนนี้ถ้ามีภารกิจที่จะต้องให้ฟังก์ชันทำในส่วนของ body นี้จะประกอบด้วยคำสั่งลำดับ (sequential statement) ที่บรรยายความสัมพันธ์ระหว่าง input parameter กับ ค่าที่จะส่งกลับ (ผลลัพธ์จากการทำงานของ FUNCTION) ในส่วนนี้ห้ามมี concurrent statement (ยกเว้น statement ที่เป็นได้ทั้งสองประเภท) และจะต้องมีคำสั่ง RETURN (statement) อย่างน้อยหนึ่งคำสั่ง คำสั่งนี้แตกต่างจากคำ RETURN type ในหัวข้อที่แล้ว เพราะคำสั่งนี้จะเป็นตัวบ่งบอกค่าที่จะส่งกลับไปยังรูปแบบ (model) ใน function body สามารถที่จะมีคำสั่งนี้ได้หลายๆ ครั้ง แต่เพียงค่าๆ เดียวเท่านั้นที่จะถูกส่งกลับ หน้าที่ของคำสั่งนี้มิใช่เพียงแต่กำหนดค่าเท่านั้น แต่ยังมีหน้าที่หยุดการทำงานของฟังก์ชันด้วย ซึ่งรายละเอียดของคำสั่งนี้จะกล่าวในหัวข้อต่อไป

Function body จะเขียนอยู่ในส่วนประกาศของ architecture design unit (พื้นที่ระหว่างคำ ARCHITECTURE กับ BEGIN) การเรียก body ไปใช้สามารถกระทำได้โดยปราศจากส่วนที่เป็น function declaration ของฟังก์ชันเอง แต่การเรียกใช้จะต้องเป็นการเรียกจากตำแหน่งที่อยู่ระดับเดียวกัน หรือระดับต่ำกว่าตำแหน่งที่ body อยู่เท่านั้น การเขียน function body นั้นมีรูปแบบของกฎเกณฑ์การเขียนตามที่แสดงในรูปที่ 8.5

```
FUNCTION name [(formal_parameter_list)] RETURN TYPE IS
        [declarative_statements]
BEGIN
        [sequential_statements]
        [including RETURN expression]
END name;
```

รูปที่ 8.5: **โครงสร้างของ** function body

ด้วยเหตุผลที่ว่าค่าของ object ที่เป็น formal parameter list ไม่สามารถถูกเปลี่ยนแปลงได้ บางครั้งมีความจำเป็นที่จะต้องมีตัวแปรที่ใช้ในการคำนวณของฟังก์ชันตัวแปรนี้สามารถประกาศในส่วนที่เป็น declarative_part ของ function body ให้เป็น VARIABLE ได้ และตัวแปรนี้จะใช้ได้เฉพาะใน body ที่ตัวมันถูกประกาศ

ส่วนที่บรรยายพฤติกรรมของความสัมพันธ์ระหว่าง input parameter กับค่าที่จะส่งกลับนั้นเขียนอยู่ในพื้นที่ระหว่างคำว่า BEGIN และ END name คำสั่งที่เขียนในส่วนนี้จะต้องเป็น sequential statement เท่านั้น ภายในส่วนนี้จะต้องมีคำสั่ง RETURN อย่างน้อยหนึ่งคำสั่ง สำหรับชื่อ (name) ที่อยู่ต่อจากคำ END ในบรรทัดสุดท้ายนั้น เป็นสิ่งที่ต้องมี และต้องเป็นชื่อเดียวกันกับชื่อของฟังก์ชัน

8.2.5 RETURN Statement

หน้าที่ของคำสั่ง RETURN มีอยู่สองอย่างคือ

1) ส่งกลับค่าของผลลัพธ์ที่สร้างขึ้นมาจากฟังก์ชันไปยัง object ที่ถูกกำหนดให้รับค่า output จาก FUNCTION

2) หยุดการทำงานของ subprogram

คำสั่งนี้จะใช้ในเฉพาะโปรแกรมย่อย (subprogram) เท่านั้น และเนื่องจากมีโปรแกรมย่อย (subprogram) สองประเภทในภาษา VHDL คำสั่ง RETURN จึงเขียนได้สองรูปแบบ

- สำหรับการใช้ในฟังก์ชันคำสั่งนี้จะเขียนในลักษณะดังนี้

RETURN expression ;

รูปแบบนี้จะประกอบด้วย expression ที่เป็นตัวบ่งบอกค่าที่จะส่งกลับโดยฟังก์ชันส่วนที่เป็น expression นี้สามารถที่จะเป็นค่าใดๆ (value), ชื่อของ object หรือ expression ที่ใช้สำหรับสร้างค่าใดค่าหนึ่ง หน้าที่ของคำสั่งประการแรกคือ ส่งค่ากลับไปยังส่วนที่เรียกใช้ฟังก์ชันประการที่สอง หยุดการทำงานของ FUNCTION

- สำหรับการใช้ใน PROCEDURE (จะกล่าวในหัวข้อต่อไป) มีรูปแบบการเขียน

RETURN ;

รูปแบบนี้จะไม่มี expression และใช้ได้เฉพาะใน PROCEDURE เท่านั้น

รูปที่ 8.6 แสดงให้เห็นการเขียน function body เพื่ออธิบายการทำงานของ RETURN statement ใน FUNCTION

```
FUNCTION example (a, b : IN BIT) RETURN BOOLEAN IS
BEGIN
RETURN a = b;
END example;
```

รูปที่ 8.6: **ตัวอย่างของ** function body

โครงสร้างของฟังก์ชันประกอบด้วย RETURN statement โดยที่ค่าของ a และ b จะถูกเปรียบเทียบด้วยเครื่องหมายเสมอภาค (เป็น relational operator) ผลลัพธ์ที่ได้อาจจะเป็น TRUE (ในกรณีที่ทั้งคู่มีค่าเท่ากัน) หรือ FALSE (ในกรณีที่ทั้งคู่มีค่าไม่เท่ากัน) ฉนั้นค่า TRUE หรือ FALSE จะถูกส่งกลับไปยังตำแหน่งที่ฟังก์ชันถูกเรียก ฉนั้นจะเห็นได้ว่าค่าที่ส่งกลับจัดอยู่ในกลุ่มของค่าประเภท BOOLEAN ซึ่งตรงกับการเขียน RETURN type

ในรูปที่ 8.7 เป็นตัวอย่างของ function body ที่เก็บอยู่ใน package body

```
PACKAGE BODY util IS
  FUNCTION vect_to_int (vect : BIT_VECTOR (0 TO 3)) RETURN INTEGER IS
      VARIABLE result : INTEGER := 0;
      VARIABLE weight : INTEGER := 1;
  BEGIN
      FOR i IN 0 TO 3 LOOP
          IF vect (i) = '1' THEN
              result := result + weight;
          END IF;
          weight := weight * 2;
      END LOOP;
      RETURN result;
  END vect_to_int;

  FUNCTION rising (SIGNAL sig : BIT) RETURN BOOLEAN IS
  BEGIN
      IF sig = '0' THEN
          RETURN FALSE;
      ELSE
          RETURN TRUE;
      END IF;
  END rising;

  FUNCTION falling (SIGNAL sig : BIT) RETURN BOOLEAN IS
  BEGIN
      IF sig = '0' THEN
          RETURN TRUE;
      ELSE
          RETURN FALSE;
      END IF;
  END falling;
END util;
```

รูปที่ 8.7: **ตัวอย่าง** function body

8.2.6 FUNCTION Calls

การทำงานของฟังก์ชันเกิดขึ้นได้โดยการเรียก (call) ในลักษณะของ expression ซึ่งอยู่ในรูปของ function call และจะเป็นการเรียก function body มาใช้

การเรียกนั้นสามารถกระทำได้จากทุกๆ ตำแหน่งที่อยู่ระดับเดียวกันกับ function declaration หรือในทุกๆ ระดับที่อยู่ต่ำกว่าระดับ declaration อย่างเช่น ถ้า function declaration อยู่ในบริเวณของส่วนที่ใช้ประกาศของ architecture (พื้นที่ระหว่าง ARCHITECTURE และ BEGIN) ฉนั้นฟังก์ชันจะถูกเรียกใช้ได้จากทุกๆ ส่วนของ statement (concurrent และ sequential) ที่อยู่ใน architecture นั้นและต่ำกว่าตำแหน่งของ function declaration แต่ถ้ามีการประกาศ function declaration ใน PROCESS ใดๆฟังก์ชันจะสามารถถูกเรียกใช้ได้เฉพาะภายใน PROCESS นั้นๆ เท่านั้น ถึงแม้ว่า PROCESS อื่นๆ จะอยู่ใน architecture เดียวกันก็ตาม

เช่นเดียวกับกรณีที่มีแต่ function body การเรียกใช้สามารถที่กระทำได้จากระดับเดียวกัน หรือระดับที่ต่ำกว่า ข้อดีของการเขียนทั้ง declaration และ body นั้นคือ ส่วนที่เป็น declaration สามารถอยู่ในระดับที่สูงกว่า body ได้ เพื่อเปิดโอกาสให้เรียกใช้ส่วนที่เป็น body ข้ามระหว่างโครงสร้างที่ขนานกัน โดยผ่านทาง function declaration

การเรียกฟังก์ชันมาใช้ทำได้โดยเขียนชื่อของฟังก์ชันตามด้วยรายชื่อของ object ที่ต้องการส่งผ่านค่าไปให้ฟังก์ชันซึ่งการเรียกนี้มีรูปแบบการเขียนดังนี้

function_name (passing_parameter_list)

การที่ฟังก์ชันส่งกลับเพียงค่าๆ เดียว ดังนั้นการเรียกใช้จะต้องเรียกจากตำแหน่งที่สามารถรับค่าๆ เดียวนี้ที่ส่งมาจากฟังก์ชันได้

8.2.7 Passing Parameter List

พารามิเตอร์ในส่วนของ function call expression เรียกว่า passing parameter list รายชื่อนี้เป็นตัวกำหนด "ค่า" (value) ของ object ตัวใดๆ ในรูปแบบที่จะส่งผ่านไปให้กับฟังก์ชัน(โดยผ่านทาง formal parameter list) วิธีการเขียนมีอยู่ด้วยกันสองแบบคือ

1) **สัมพันธ์โดยตำแหน่ง (Position Association)**

การเขียนความสัมพันธ์ลักษณะนี้ ชื่อของ object ที่อยู่ใน passing parameter list (จาก function call expression) จะสัมพันธ์กับ object ที่อยู่ใน parameter list ของ function declaration หรือ body ตามตำแหน่งที่เขียนตามลำดับเช่น

Function declaration/body:

```
FUNCTION tran_ck (SIGNAL a_1 : IN BIT; SIGNAL b_2 : IN BIT     )
            RETURN BOOLEAN IS ...
```

Function call:

```
IF tran_ck (ray_1, ray_2) ...
```

ค่าของ ray_1 จะสัมพันธ์กับ a_1 และ ค่าของ ray_2 จะสัมพันธ์กับ b_2 โดยตำแหน่ง

2) **สัมพันธ์โดยชื่อ (Named Association)**

เป็นการบอกความสัมพันธ์ด้วยการกำหนดชื่อว่า object ชื่ออะไรสัมพันธ์กับใครเช่น

Function call:

```
IF tran_ck (b_2 => ray_2, a_1 => ray_1) ...
```

จะเห็นได้ว่า b_2 สัมพันธ์กับ ray_2 และ a_1 สัมพันธ์กับ ray_1 โดยใช้เครื่องหมาย "=>" เป็นตัวกำหนดความสัมพันธ์ด้วยชื่อ

8.3 PROCEDURE

ทั้งฟังก์ชันและ PROCEDURE ต่างก็เป็นโปรแกรมย่อย (subprogram) ด้วยกันทั้งคู่ ฉนั้นจึงตอบสนองจุดประสงค์ของการเขียนรูปแบบเดียวกัน และมีโครงสร้างที่คล้ายกันคือ ประกอบด้วยส่วนที่เป็น declaration และ body แต่จะมีข้อแตกต่างระหว่างกันอยู่ ได้แก่

1) พื้นฐานที่แตกต่างกันได้แก่ฟังก์ชันไม่สามารถแก้ไข เปลี่ยนแปลงค่าที่ส่งเข้าไปได้ แต่ขณะที่ PROCEDURE สามารถที่จะแก้ไข และดัดแปลงค่าต่างๆ ที่ส่งผ่านเข้าไปได้

2) ในฟังก์ชันจะต้องมี RETURN statement แต่ใน PROCEDURE จะมีหรือไม่ก็ได้ ทั้งนี้ด้วยเหตุผลที่ PROCEDURE ไม่ได้ส่งค่าเฉพาะกลับไปยังตำแหน่งที่เรียก ฉนั้นคำสั่ง RETURN ที่อาจจะอยู่ใน PROCEDURE ได้นั้น จึงมีจุดประสงค์เฉพาะเพื่อหยุดการทำงานของ PROCEDURE เอง

3) FUNCTION ส่งค่าเพียงค่าเดียวกลับ แต่ PROCEDURE สามารถส่งค่าหลายๆ ค่ากลับได้ ดังนั้น object ที่อยู่ใน formal parameter list ของ PROCEDURE จึงสามารถมี MODE ประเภท OUT (output) ได้ และ object ตัวนั้นจะไม่รับค่าจากรูปแบบ (model) เมื่อ PROCEDURE ถูกเรียก

4) ภายในโครงสร้างของฟังก์ชันห้ามมีคำสั่งที่ส่งผลกระทบต่อค่าที่ส่งเข้าไป แต่ PROCEDURE กระทำได้ ฉนั้น MODE ของ object ใน formal parameter list ของฟังก์ชันจึงเป็นประเภท IN เท่านั้น ในขณะที่ใน PROCEDURE สามารถเป็น IN, INOUT[3] และ OUT ได้

5) การเรียกฟังก์ชันเป็นการเรียกในรูปของ expression แต่การเรียก PROCEDURE เป็นการเรียกในรูปของ statement

8.3.1 Procedure Declaration

เช่นเดียวกันกับฟังก์ชันคือ PROCEDURE ประกอบด้วยส่วนที่เป็น declaration และ body ส่วนที่เป็น body นั้นต้องมีถ้ามีภารกิจใดๆ ให้ ส่วน declaration ไม่จำเป็นต้องมี PROCEDURE ก็สามารถทำงานได้

[3] INOUT เป็น bidirectional port

```
PROCEDURE name (formal_paramete_list) ;
```

รูปที่ 8.8: **รูปแบบการเขียน** procedure declaration

จากรูปแบบนี้สามารถอธิบายความหมายต่างๆ ได้ดังนี้

1) **ชื่อของ procedure**: ชื่อของ PROCEDURE จะถูกใช้ในการเรียกให้ทำงาน ในที่นี้คือ name

2) **formal parameter list**: จะเป็นรายชื่อของพารามิเตอร์ที่ใช้ใน PROCEDURE จะเป็นตัวรับค่าที่ผ่านให้กับ PROCEDURE จากจุดตำแหน่งที่ถูกเรียก สิ่งที่สามารถกำหนดเพิ่มเติมได้แก่คุณสมบัติของพารามิเตอร์เช่น CLASS, ชื่อ (name), MODE และ TYPE เช่นการเขียน

ส่วนที่เป็น declaration ไม่จำเป็นต้องมี PROCEDURE ที่ทำงานได้ตามจุดประสงค์ที่เขียน สามารถที่จะมีได้โดยปราศจากส่วนนี้ แต่บางกรณีจะต้องมีส่วนนี้เมื่อ

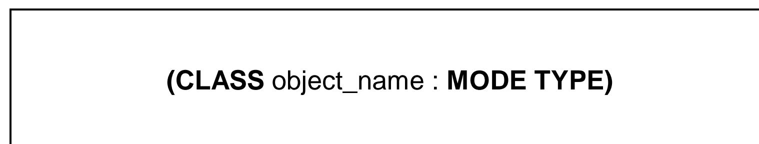
☑ เมื่อ PROCEDURE เป็น recursive procedure หมายความว่าในกรณีที่มีการอ้างถึง PROCEDURE นั้นจากภายในตัวโครงสร้างของ PROCEDURE เอง (เรียกตัวเอง)

☑ เมื่อมีตำแหน่งที่เรียก PROCEDURE อยู่ก่อนที่จะถึงตำแหน่งที่ procedure body อยู่ ดังนั้นจึงต้องถูกประกาศในพื้นที่สำหรับเขียนประกาศของ architecture design unit (ระหว่างคำว่า ARCHITECTURE กับ BEGIN)

สิ่งที่สำคัญในการสร้าง PROCEDURE คือถ้ามีการประกาศ PROCEDURE (procedure declaration) จะต้องประกาศในพื้นที่เดียวกันกับส่วนที่เป็น procedure body อยู่ (ข้อกำหนดนี้มีข้อยกเว้นในกรณีที่สร้าง PACKAGE เก็บโครงสร้างของ PROCEDURE คือ procedure body จะอยู่ใน package declaration และ function body จะอยู่ใน package body) และ procedure declaration จะต้องมาก่อน procedure body

8.3.2 Formal Parameter List

หน้าที่ของ formal parameter list จะเป็นตัวบอก CLASS, NAME (identifier), MODE และ TYPE ของ object ที่มีค่าสำหรับส่งให้กับ PROCEDURE มีรูปแบบการเขียนดังในรูปที่ 8.9



รูปที่ 8.9: **รูปแบบการเขียน** formal parameter list

จากรูปที่ 8.9 สามารถได้ข้อมูลเกี่ยวกับตัว parameter ดังนี้

- **CLASS**: หมายถึงชั้นของ object ซึ่งอาจจะเป็น SIGNAL, VARIABLE หรือ CONSTANT ส่วนตัวแปร (variable) สามารถเป็น parameter ใน PROCEDURE ได้ แต่ถ้าไม่มีการกำหนด CLASS และ MODE ของ object VHDL จะกำหนด (ค่าตายตัว) ให้เป็น CONSTANT MODE IN ถ้าไม่กำหนด CLASS แต่กำหนด MODE เป็น IN หรือ INOUT จะกำหนด (ค่าตายตัว) ให้เป็น VARIABLE

- **object_name**: กำหนดโดยผู้ออกแบบ สามารถกำหนดเป็นชื่อใดๆ ได้ที่เป็นไปตาม กฎของภาษา VHDL และชื่อที่ตั้งขึ้นนี้ไม่จำเป็นต้องเป็นคล้องจอง หรือชื่อเดียวกันกับ object อื่นๆ ใน model

- **MODE**: กำหนดทิศทางการไหลของข้อมูลโดยคำนึงถึงหลักการของ PROCEDURE และสามารถมี MODE ได้สามชนิดคือ IN, INOUT และ OUT ค่าตายตัวคือ IN ถ้าไม่มีการเขียนกำหนดไว้

- **TYPE**: จะเป็นตัวกำหนดกลุ่มของค่า (value set) ต่างๆ ที่ object สามารถจะมีได้ ค่า (value) ที่ไม่อยู่ในกลุ่มของค่า ไม่สามารถที่จะส่งผ่านจากรูปแบบ (model) ไปยัง

PROCEDURE ได้ แต่อย่างไรก็ตาม TYPE ของพารามิเตอร์ใน formal parameter list จะต้องเป็นประเภทเดียวกันกับ TYPE ของ object ที่นำมาจากรูปแบบ (model) เพื่อส่งผ่านไปยัง PROCEDURE

8.3.3 Parameter Modes

ชนิดหรือ MODE ที่ปรากฏอยู่ใน parameter list นั้นจะเป็นตัวบอกว่า parameter นั้นๆ เกี่ยวข้องกับ PROCEDURE ในลักษณะใดของการส่งผ่านข้อมูล (เข้า - ออก) และจะกระทำการใดๆ กับ parameter เหล่านั้นได้บ้าง ในที่นี้มีอยู่ด้วยกันสามประเภทคือ

1) **IN**: ค่าของ parameter จะถูกส่งผ่านเข้าไปใน PROCEDURE แต่ค่าเหล่านั้นไม่สามารถแก้ไข หรือเปลี่ยนแปลงได้

2) **OUT**: ค่าของ parameter จะไม่ถูกส่งเข้าไปใน PROCEDURE เมื่อถูกเรียก แต่จะส่งเป็นการส่งค่าที่ได้จากการทำงานภายในออกสู่ตำแหน่งที่เรียก

3) **INOUT**: ค่าประเภทนี้จะถูกอ่านเข้าไปใน PROCEDURE จากจุดตำแหน่งที่เรียก และถูกแก้ไข หรือดัดแปลงภายใน ค่าผลลัพธ์ที่ได้จะถูกส่งออกมายังตำแหน่งที่เรียกอีกครั้ง

ในรูปที่ 8.10 แสดงให้เห็นตัวอย่างของ procedure declaration ที่อยู่ใน package declaration

```
PACKAGE util IS
   PROCEDURE  add_element (element : IN REAL;
                  VARIABLE filter_data : INOUT filter_data_type);
   PROCEDURE  zero_out (input : INOUT filter_data_type);
   PROCEDURE  still_busy;
END util;
```

รูปที่ 8.10: **ตัวอย่างของ** procedure declation

8.3.4 Procedure Body

ส่วนนี้เป็นส่วนที่บรรยายพฤติกรรมของ PROCEDURE และจะต้องมีส่วนนี้ถ้ามีภารกิจที่จะต้องให้ PROCEDURE ทำ

ในส่วนของ body นี้จะประกอบด้วยลำดับของ sequential statement ที่บรรยายความสัมพันธ์ระหว่างค่าของ input parameter กับ ค่าที่จะส่งกลับ (ผลลัพธ์จากการทำงานของ PROCEDURE) ในส่วนนี้ห้ามมี concurrent statement (นอกจาก statement ที่เป็นได้ทั้งสองประเภท) และใน PROCEDURE ไม่จำเป็นต้องมีคำสั่ง RETURN (statement) แต่ถ้ามีคำสั่งนี้ จะเป็นคำสั่ง RETURN ที่ไม่ได้กำหนดค่าที่จะส่งกลับ เพราะคำสั่งนี้จะไม่เป็นตัวบ่งบอกค่าที่จะส่งกลับไปยังรูปแบบ (model) ใน procedure body แต่จะมีหน้าที่หยุดการทำงานของ PROCEDURE ในโครงสร้างที่ไม่มี RETURN การทำงานของ PROCEDURE จะหยุดด้วยคำสั่ง END บรรทัดสุดท้ายของ PROCEDURE

Procedure body จะเขียนอยู่ในส่วนประกาศของ architecture design unit (พื้นที่ระหว่างคำ ARCHITECTURE กับ BEGIN) การเรียก body ไปใช้สามารถกระทำได้โดยปราศจากส่วนที่เป็น procedure declaration ของ PROCEDURE เอง แต่การเรียกใช้จะต้องเป็นการเรียกจากตำแหน่งที่อยู่ระดับเดียวกัน หรือระดับต่ำกว่าตำแหน่งที่ body อยู่เท่านั้น การเขียน procedure body นั้นมีรูปแบบของกฎเกณฑ์การเขียนตามที่แสดงในรูปที่ 8.11

```
PROCEDURE name (formal_parameter_list)  IS
        -- declarative_statements
BEGIN
        -- sequential_statements
END name;
```

รูปที่ 8.11: **โครงสร้างของ** procedure body

ในบรรทัดแรกของ procedure body จะเหมือนกับ declaration ยกเว้นคำว่า **IS** ที่ในส่วน declaration ไม่มี สิ่งที่สำคัญคือ name, parameter และ ค่าที่จะถูกส่งกลับต้องเป็นชนิดเดียวกัน

ขึ้นอยู่กับประเภทของ MODE ในแต่ละ object ที่ปรากฏอยู่ใน parameter list ค่าที่ผ่านเข้าไป อาจจะถูกดัดแปลง แก้ไข หรือไม่ก็ได้ หรือบางที่อาจจะมีแหล่งกำเนิดค่าใหม่ภายใน และส่งค่าไปให้รูปแบบ (model) ที่เรียกใช้ PROCEDURE

ส่วนที่เป็น declarative area ใช้ประกาศ object ที่ใช้เฉพาะภายใน PROCEDURE เท่านั้น และไม่สามารถที่จะส่งออกสู่ภายนอกได้ ในเนื้อที่บริเวณระหว่าง BEGIN และ END เป็นส่วนที่ใช้เขียนบรรยายพฤติกรรมของ PROCEDURE ในการทำงาน ซึ่งจะประกอบด้วย sequential statement เท่านั้น เช่นเดียวกันกับฟังก์ชันที่ name ชื่อของ PROCEDURE จะต้องมีในบรรทัดของ END statement

รูปที่ 8.12 เป็นตัวอย่างของการเขียน procedure body ที่อยู่ใน package body

```
PACKAGE BODY util IS
    PROCEDURE   add_element (element : IN REAL;
                    VARIABLE filter_data : INOUT filter_data_type) IS
    BEGIN
        FOR IN filter_data'HIGH DOWNTO filter_data'LOW + 1 LOOP
            filter_data (i) := filter_data (i - 1);
        END LOOP;
        filter_data (filter_data'LOW) := elememt;
    END add_element;

    PROCEDURE   zero_out (input : INOUT filter_data_type) IS
    BEGIN
        FOR i IN input'RANGE LOOP
            input (i) := 0.0;
        END LOOP;
    END zero_out;

    PROCEDURE   still_busy IS
    BEGIN
        ASSERT FALSE REPORT "Still Busy!" SEVERITY NOTE;
    END still_busy;
END util;
```

รูปที่ 8.12: **ตัวอย่างการเขียน** procedure body

8.3.5 Procedure Calls

การทำงานของ PROCEDURE เกิดขึ้นได้ด้วยการเรียกใช้ในรูปของคำสั่ง (statement) หรือที่ใช้คำว่า "procedure call" ซึ่งจะเป็นตัวทำให้ procedure body ทำงาน procedure call statement นั้นเป็นทั้ง sequential และ concurrent statement

ตำแหน่งของการเรียกใช้ในรูปแบบ (model) สามารถอยู่ในระดับเดียวกันกับ procedure declaration หรือระดับใดๆ ที่อยู่ต่ำกว่า เช่นถ้าส่วนประกาศ declaration อยู่บริเวณสำหรับประกาศของ architecture (พื้นที่ระหว่าง ARCHITECTURE และ BEGIN) ฉนั้น PROCEDURE จะถูกเรียกใช้ได้จากทุกๆ ส่วนของ statement (concurrent และ sequential) ที่อยู่ใน architecture นั้นและต่ำกว่าตำแหน่งของ procedure declaration แต่ถ้ามีการประกาศ procedure declaration ใน PROCESS ใดๆ PROCEDURE นั้นจะสามารถถูกเรียกใช้ได้ฌฉพาะภายใน PROCESS นั้นๆ เท่านั้น ถึงแม้ว่า PROCESS อื่นๆ จะอยู่ใน architecture เดียวกันก็ตาม

เช่นเดียวกับกรณีที่มีแต่ procedure body การเรียกใช้สามารถที่กระทำได้จากระดับเดียวกัน หรือระดับที่ต่ำกว่า ข้อดีของการเขียนทั้ง declaration และ body นั้นคือ ส่วนที่เป็น declaration สามารถอยู่ในระดับที่สูงกว่า body ได้ เพื่อเปิดโอกาสให้เรียกใช้ส่วนที่เป็น body ข้ามระหว่างโครงสร้างที่ขนานกัน โดยผ่านทาง procedure declaration

การเรียก PROCEDURE มาใช้ทำได้โดยการเขียนชื่อ (name) ของ PROCEDURE นั้น ตามด้วยรายชื่อของ object ที่ค่าของมันจะถูกส่งผ่านให้กับ PROCEDURE ตามรูปแบบการเขียนดังนี้

procedure_name (passing_parameter_list) ;

เนื่องจากเป็นชุดคำสั่ง statement จึงต้องปิดท้ายด้วยเครื่องหมายอัฒภาค (;) หรือ semicolon และต้องไม่ปรากฏเป็น expression ใน statement อื่น

เมื่อใน concurrent statement มีคำสั่ง procedure call อยู่ PROCEDURE จะถูกกระตุ้นให้ทำงาน (ถูกเรียก) เมื่อมีสัญญาณตัวใดตัวหนึ่งที่มีความสัมพันธ์กับ parameter ที่มี MODE ประเภท IN หรือ INOUT ใน passing parameter list มีการเปลี่ยนแปลงระดับของสัญญาณ

8.3.6 Passing Parameter List

Parameter list ในส่วนของ procedure call statement เรียกว่า passing parameter list หรือ formal parameter list รายชื่อนี้เป็นตัวกำหนด "ค่า" ของ object ตัวใดๆ ในรูปแบบที่จะส่งผ่านไปให้กับ PROCEDURE (โดยผ่าน parameter list) วิธีการเขียนมีอยู่ด้วยกันสองแบบคือ

1) **สัมพันธ์โดยตำแหน่ง (Position Association)**

การเขียนความสัมพันธ์ลักษณะนี้ ชื่อของ object ที่อยู่ใน passing parameter list (จาก procedure call expression) จะสัมพันธ์กับ object ที่อยู่ใน parameter list ของ procedure declaration หรือ body ตามตำแหน่งที่เขียนตามลำดับเช่น***Procedure declaration/body:***

```
PROCEDURE find_min ( VARIABLE a_ray : IN int_array;
                     SIGNAL min_val  : INOUT INTEGER;
                     SIGNAL old_min  : OUT INTEGER  ) IS ...
```

Procedure call:

```
find_min (array_1, out_1, old_1) ;
```

ค่าของ array_1 จะสัมพันธ์กับ a_ray และ ค่าของ out_1 จะสัมพันธ์กับ min_val และ old_1 สัมพันธ์กับ old_min

2) **สัมพันธ์โดยชื่อ (Named Association)**

เป็นการบอกความสัมพันธ์ด้วยการกำหนดชื่อว่า object ชื่ออะไรสัมพันธ์กับใครเช่น

Procedure call:

```
find_min (min_val => out_1, a_ray => array_1, old_min => old1) ;
```

จะเห็นได้ว่า min_val สัมพันธ์กับ out_1 และ a_ray สัมพันธ์กับ array_1 โดยมีเครื่องหมาย "=>" เป็นตัวกำหนดความสัมพันธ์ด้วยชื่อ

8.4 สรุป

เช่นเดียวกับภาษาโปรแกรมชั้นสูงทั่วๆ ไป หลักการภาษา VHDL สนับสนุนแนวความคิดของการใช้โปรแกรมย่อย สำหรับบรรยายพฤติกรรมทั่วไป ที่บางครั้งอาจจะมีการทำงานในลักษณะเดียวกัน ซ้ำกันหลายครั้งภายในรูปแบบ

โปรแกรมย่อย (subprogram) ที่มีอยู่ในภาษา VHDL มีอยู่ด้วยกันสองประเภท ได้แก่ประเภทฟังก์ชันและ PROCEDURE โดยปกติแล้วจะเก็บไว้ใน package design unit โดยที่ส่วนที่เป็น declaration ของทั้งฟังก์ชันและ PROCEDURE จะเก็บอยู่ใน package declaration ส่วนที่เป็น body จะเก็บไว้ใน package body โปรแกรมย่อยที่เขียนเช่นนี้จะถูกเรียกนำไปใช้ (call) จากรูปแบบ อื่นๆ ได้ด้วยการทำให้ package design unit สามารถถูกมองเห็นได้ (visibility) จากรูปแบบ (model) นั้นๆ ก่อนโดยใช้คำสั่ง LIBRARY- และ USE statement

สำหรับฟังก์ชันโดยทั่วไปจะเป็น expression ที่ผู้ออกแบบเขียนขึ้น ในฟังก์ชันสามารถมี parameter list ที่มีความสัมพันธ์กันทั้ง CLASS, MODE และ TYPE ส่วนที่บรรยายพฤติกรรมของฟังก์ชันเรียกว่า function body ที่ภายในประกอบด้วยลำดับของ sequential statement และเนื่องจากฟังก์ชันไม่สามารถดัดแปลง แก้ไขค่าที่ผ่านเข้าไปได้ ฉนั้น object จึงมี MODE เป็น IN การเรียกใช้ฟังก์ชันจะอยู่ในรูปของ expression

ในส่วนของ PROCEDURE นั้น มีความอ่อนตัวมากกว่า สามารถที่จะดัดแปลง แก้ไขค่าที่ผ่านเข้าไปได้ ฉนั้น CLASS ของ object ที่อยู่ใน parameter list จึงสามารถเป็นได้ทั้ง CONSTANT, VARIABLE และ SIGNAL และเช่นเดียวกับ MODE ที่เป็น IN, INOUT และ OUT ได้ ทั้งนี้ในการเรียกใช้ PROCEDURE จะต้องคำนึงว่าเรียกจากส่วนที่เป็นโครงสร้างแบบแข่งขนาน (concurrent body) หรือโครงสร้างแบบลำดับ (sequential body) ของรูปแบบ เพราะถ้าใน PROCEDURE มีการประกาศใช้ VARIABLE จะทำให้ไม่สามารถเรียก PROCEDURE นั้นไปใช้ในเส่วนของรูปแบบ ที่เป็นโครงสร้างแบบแข่งขนานได้ เพราะ VARIABLE ไม่สามารถอยู่ใน หรือประกาศใช้ในส่วนนี้ได้ ส่วนการเรียกใช้จะอยู่ในรูปของชุดคำสั่งไม่ใช่ expression เช่นกรณีของ FUNCTION

ตัวอย่างต่อไปแสดงให้เห็นการสร้างโปรแกรมย่อย การเก็บโปรแกรมย่อยใน package และการเรียกใช้ในลักษณะต่างๆ รูปที่ 8.13 เป็น package ที่ชื่อ utility ภายในประกอบด้วยส่วนประกาศของโปรแกรมย่อย (procedure-, function declaration) และส่วนบรรยายการทำงาน (procedure-, function body)

```
PACKAGE utility IS
  PROCEDURE bin_to_int (bin_in : IN BIT_VECTOR; int_out : OUT INTEGER);
  PROCEDURE int_to_bin (int_in : IN INTEGER; bin_out : OUT BIT_VECTOR);
  FUNCTION inc_bin (x : BIT_VECTOR) RETURN BIT_VECTOR;
END utility;
--
PACKAGE BODY utility IS
    PROCEDURE bin_to_int (bin_in : IN BIT_VECTOR; int_out : OUT INTEGER) IS
      VARIABLE result : INTEGER;
   BEGIN
      result := 0;
      FOR i IN 0 TO (bin_in'LENGTH - 1) LOOP
         IF bin_in(i) = '1' THEN
            result := result + 2**i;
         END IF;
      END LOOP;
      int_out := result;
   END bin_to_int;
--
   PROCEDURE int_to_bin (int_in : IN INTEGER; bin_out : OUT BIT_VECTOR) IS
      VARIABLE tmp : INTEGER;
   BEGIN
      tmp := int_in;
      FOR i IN 0 TO (bin_out'LENGTH - 1) LOOP
         IF (tmp MOD 2 = 1) THEN
           bin_out(i) := '1';
         ELSE  bin_out(i) := '0';
         END IF;
         tmp := tmp/2;
      END LOOP;
   END  int_to_bin;
   FUNCTION inc_bin (x : BIT_VECTOR) RETURN BIT_VECTOR IS
     VARIABLE i : INTEGER;
     VARIABLE t : BIT_VECTOR (x'RANGE);
   BEGIN
     bin_to_int (x, i);
     i := i + 1;
     IF i >= 2**x'LENGTH THEN i := 0;
     END IF;
     int_to_bin (i, t);
     RETURN t;
   END inc_bin;
END utility;
```

รูปที่ 8.13: Package utility

ใน package ชื่อ utility ประกอบด้วย procedure ชื่อ bin_to_int ที่ใช้สำหรับแปลงเลขฐานสองให้เป็นเลขจำนวนเต็ม (binary to integer) และ procedure ชื่อ int_to_bin ที่ใช้สำหรับแปลงเลขจำนวน

เต็มเป็นเลขฐานสอง (integer to binary) และ function ชื่อ (inc_bin) ที่ใช้สำหรับเพิ่มค่าเลขฐานสองที่ละ 1 โดยที่ใน function ดังกล่าวจะเรียก procedure (bin_to_int) เพื่อแปลงเลขที่เข้ามาให้เป็นเลขจำนวนเต็ม แล้วเพิ่มค่าเลขจำนวนเต็มด้วยค่า 1 และก่อนที่จะส่งผลลัพธ์กลับได้เรียก procedure (int_to_bin) เพื่อแปลงผลลัพธ์ที่เป็นเลขจำนวนเต็มให้เป็นเลขฐานสอง ฉนั้นจึงมีความจำเป็นที่ต้องมีส่วนประกาศของ procedure ใน package declaration

ตัวอย่างการใช้ package utility แสดงในรูปที่ 8.14 โดยใช้รูปแบบของ Synchronous 4-Bit Binary Counter ซึ่งตรงกับอุปกรณ์ 74LS163

```
USE WORK.utility.ALL;
--
ENTITY  ttl_74ls163_counter IS
    GENERIC (prop_delay : TIME := 18 NS);
    PORT  (  clk, clr_bar, ld_bar, enp, ent : IN BIT;
                abcd : IN BIT_VECTOR (3  DOWNTO 0);
                q_abcd : OUT BIT_VECTOR (3 DOWNTO 0); rco : OUT BIT  );
END ttl_74ls163_counter;
ARCHITECTURE behavioral OF ttl_74ls163_counter IS
BEGIN
    counting : PROCESS (clk)
        VARIABLE internal_count : BIT_VECTOR (3 DOWNTO 0) := "0000";
    BEGIN
        IF (clk = '1') THEN
            IF (clr_bar = '0') THEN
                internal_count := "0000";
            ELSIF (ld_bar = '0' ) THEN
                internal_count := abcd;
            ELSIF (enp = '1' AND ent = '1' ) THEN
                internal_count := inc_bin (internal_count);
                IF (internal_count = "1111") THEN
                    rco <= '1' AFTER  prop_delay;
                ELSE
                    rco <= '0';
                END IF;
            END IF;
            q_abcd <= internal_count AFTER prop_delay;
        END IF;
    END PROCESS counting;
END behavioral;
```

รูปที่ 8.14: **รูปแบบ** Synchronous 4-Bit Binary Counter

บรรทัดแรกของรูปที่ 8.14 เป็นคำสั่ง USE ที่ใช้สำหรับเปิด package และสิ่งที่ประกาศภายใน ดังนั้นคำสั่งนี้จึงต้องอยู่ก่อนส่วนที่เป็น entity design unit

```
--
ENTITY  ttl_74ls163_counter IS
    GENERIC (prop_delay : TIME := 18 NS);
    PORT  (  clk, clr_bar, ld_bar, enp, ent : IN BIT;
                abcd : IN BIT_VECTOR (3  DOWNTO 0);
                q_abcd : OUT BIT_VECTOR (3 DOWNTO 0); rco : OUT BIT  );
END ttl_74ls163_counter;
ARCHITECTURE behavioral OF ttl_74ls163_counter IS
--
   PROCEDURE bin_to_int (bin_in : IN BIT_VECTOR; int_out : OUT INTEGER) IS
     VARIABLE result : INTEGER;
  BEGIN
     result := 0;
     FOR i IN 0 TO (bin_in'LENGTH - 1) LOOP
       IF bin_in(i) = '1' THEN
          result := result + 2**i;
       END IF;
     END LOOP;
     int_out := result;
  END bin_to_int;
--
  PROCEDURE int_to_bin (int_in : IN INTEGER; bin_out : OUT BIT_VECTOR) IS
     VARIABLE tmp : INTEGER;
  BEGIN
     tmp := int_in;
     FOR i IN 0 TO (bin_out'LENGTH - 1) LOOP
       IF (tmp MOD 2 = 1) THEN
         bin_out(i) := '1';
       ELSE  bin_out(i) := '0';
       END IF;
       tmp := tmp/2;
     END LOOP;
  END  int_to_bin;
  FUNCTION inc_bin (x : BIT_VECTOR) RETURN BIT_VECTOR IS
    VARIABLE i : INTEGER;
    VARIABLE t : BIT_VECTOR (x'RANGE);
  BEGIN
    bin_to_int (x, i);
    i := i + 1;
    IF i >= 2**x'LENGTH THEN i := 0;
    END IF;
    int_to_bin (i, t);
    RETURN t;
  END inc_bin;
--
BEGIN
    counting : PROCESS (clk)
        VARIABLE internal_count : BIT_VECTOR (3 DOWNTO 0) := "0000";
    BEGIN
        IF (clk = '1') THEN
            IF (clr_bar = '0') THEN
                internal_count := "0000";
            ELSIF (ld_bar = '0' ) THEN
                internal_count := abcd;
            ELSIF (enp = '1' AND ent = '1' ) THEN
```

```
                internal_count := inc_bin (internal_count);
                IF (internal_count = "1111") THEN
                    rco <= '1' AFTER  prop_delay;
                ELSE
                    rco <= '0';
                END IF;
            END IF;
            q_abcd <= internal_count AFTER prop_delay;
        END IF;
    END PROCESS counting;
END behavioral;
```

บทที่ 9

บทเสริม (Advanced Topics)

9.1 กล่าวนำ

ความมุ่งหมายที่เรียบเรียงหนังสือเล่มนี้ขึ้นมา เพื่อที่จะเชื่อมโยงพื้นฐานของภาษา VHDL ไปสู่หลักการของการเขียนรูปแบบบรรยายพฤติกรรมระบบดิจิตอล จนกระทั้งถึงตำแหน่งนี้ ผู้อ่านควรจะมีความสามารถที่จะเข้าใจในหลักและกฎเกณฑ์ของภาษาพอสมควร และสามารถที่จะพัฒนารูปแบบของระบบดิจิตอลด้วยภาษา VHDL

เนื่องจากความจำกัดทางด้านเวลา หนังสือจึงไม่สามารถที่จะครอบคลุมเนื้อหาของภาษา VHDL ไว้ได้ทั้งหมด ฉนั้นเพื่อเป็นแนวทางให้กับผู้ที่สนใจจะศึกษารายละเอียดของภาษา ในบทนี้จะเป็นการศึกษาในหัวข้อที่ยังไม่ได้กล่าวถึง แต่มีความสำคัญต่อความเข้าใจและเป็นประโยชน์ในการเขียนรูปแบบต่อไป ซึ่งจะเป็นการกล่าวในหัวข้อ

- Configuration Specification
- Overloading
- Passive Process
- Attributes
- Resolution Function

9.2 การกำหนดโครงร่าง (Configuration Specification)

บทที่ 2 ของหนังสือเล่มนี้ได้กล่าวถึงส่วนต่างๆ ของรูปแบบที่เขียนด้วยภาษา VHDL ได้แก่ entity, architecture และ configuration ส่วนที่เป็น entity ใช้บรรยายการติดต่อกับภายนอกของระบบในรูปของสัญญาณ เข้า-ออก และพารามิเตอร์ทางฟิสิกส์ ส่วนที่เป็น architecture ใช้บรรยายพฤติกรรมความสัมพันธ์ระหว่างสัญญาณที่เข้าและออก ที่กำหนดไว้ในส่วน entity ซึ่งรูปแบบหนึ่งๆ ผู้ออกแบบสามารถที่จะเขียน architecture บรรยายได้หลายลักษณะ ดังนั้นส่วนที่เป็น configuration จะมีหน้าที่เชื่อมส่วนที่เป็น entity เข้ากับ architecture ที่ต้องการ (สำหรับการจำลองการทำงานหรือสังเคราะห์วงจร) ส่วนของ configuration จะแสดงถึงรายละเอียดของรูปแบบในลักษณะของลำดับชั้น (hierarchical) โดยเชื่อมโยงส่วนที่เป็น entity เข้ากับ architecture ในขั้นตอนของการนำอุปกรณ์ (component) มาใช้ในการเขียนรูปแบบ

รูปที่ 9.1 เป็นตัวอย่างของการเขียนรูปแบบ VHDL บรรยายพฤติกรรมของวงจรดิจิตอล โดยแยกส่วนที่เป็น entity และส่วนที่เป็น architecture

```
ENTITY and_or IS
    PORT (   in1, in2, in3: IN BIT;
             out1: OUT BIT  );
END and_or;
```

รูปที่ 9.1(a): **ส่วนที่เป็น** entity **ของรูปแบบ**

```
ARCHITECTURE structure OF and_or IS
  COMPONENT and2 IS
      PORT (   and1, and2: IN BIT;
               and_out: OUT BIT );
  END COMPONENT;

  COMPONENT or2 IS
      PORT (   or1, or2: IN BIT;
               or_out: OUT BIT  );
  END COMPONENT;
  SIGNAL internal_1: BIT;
BEGIN
  u1:and2       PORT MAP (in1, in2, internal_1);
  u2:or2        PORT MAP (in3, internal_1, out_1);
END structure;
```

รูปที่ 9.1(b): **ส่วนที่เป็น** architecture **ของรูปแบบ**

ส่วนที่เป็น configuration ที่บรรยายการเชื่อมต่อของโครงสร้างระหว่างส่วนที่เป็น architecture เข้ากับส่วนที่เป็น entity ที่มีชื่อว่า **and_or** รูปที่ 9.2 แสดงให้เห็นโครงสร้างของคู่ entity-architecture โดยใช้วิธีการทำ CONFIGURATION

```
LIBRARY parts;
CONFIGURATION ao_con OF and_or IS
    FOR structure
        FOR u1:  and2 USE ENTITY WORK.and2 (behave);
        END FOR;
        FOR u2:  or2 USE CONFIGURATION parts.or2_con
        END FOR;
    END FOR;
END mux_con;
```

รูปที่ 9.2: **ส่วนที่เป็น** comfiguration **ของรูปแบบ**

คำสั่ง FOR ในรูปที่ 9.2 คือส่วนที่เรียกว่า configuration specification ซึ่งมีอยู่ด้วยกันสองแบบด้วยกันคือ การเขียนในลักษณะที่เรียกว่า **entity-architecture pair configuration**[1] ตามที่แสดงในรูปที่ 9.3

```
FOR u1:    and2 USE ENTITY WORK.and2(behave);
END FOR;
```

รูปที่ 9.3: Entity-architecture pair configuration

ในตัวอย่าง (รูปที่ 9.2 และรูปที่ 9.3) ให้ความหมายของการกำหนดโครงร่าง (configuration specification) ในรูปที่ 9.1(b) **"สำหรับชิ้นส่วน u1 ที่นำมาประกอบในรูปแบบ ได้แก่อุปกรณ์ (component) ที่ประกาศด้วยชื่อ and2 ซึ่งเป็นรูปแบบที่อยู่ใน library ชื่อสัญลักษณ์ WORK (working directory) ส่วนที่เป็น entity ชื่อ and2**[2] **(ในที่นี้ใช้ชื่อเดียวกันกับรูปแบบที่จะนำมาใช้) และส่วนที่เป็น architecture ชื่อ behave ของ entity นั้น"**

---

[1] Douglas L. Peery. "VHDL" : หน้า 196-197, McGRAW-HILL International Edition, 1991

[2] การกำหนดชื่ออุปกรณ์ (component) ไม่จำเป็นต้องใช้ชื่อเดียวกันกับชื่อของ entity ที่เป็นรูปแบบที่จะนำมาใช้

การเขียน configuration specification อีกลักษณะหนึ่งที่เรียกว่า **lower-level configuration**[3] ได้แสดงให้เห็นอีกครั้งในรูปที่ 9.4

```
FOR u2:  or2 USE CONFIGURATION parts.o2_con;
END FOR;
```

รูปที่ 9.4: Lower-level configuration

ในตัวอย่าง (รูปที่ 9.2 และรูปที่ 9.4) ให้ความหมายของการกำหนดโครงร่าง (configuration specification) ในรูปที่ 9.1 (b) "**สำหรับชิ้นส่วน u2 ที่นำมาประกอบในรูปแบบ ได้แก่อุปกรณ์ (component) ที่ประกาศด้วยชื่อ or2 ซึ่งเป็นโครงร่าง (configuration) ที่ถูกเก็บไว้ใน library ที่มีชื่อสัญลักษณ์ parts และตัวโครงร่างชื่อ or2_con**" หรืออาจมองในรูปของ configuration ซ้อน configurationได้

การกำหนดโครงร่างจำเพาะ (configuration specification) นั้นไม่มีข้อจำกัดในการเขียน ดังเช่นในรูปที่ 9.2 นั้นเป็นการเขียนแยกออกมาต่างหาก และถูกนำมาเก็บไว้ใน library (อยู่ใน directory อื่นที่ไม่ใช้ working directory) ผู้ออกแบบสามารถเขียนกำหนดโครงร่างส่วนประกอบของ architecture ได้ตามที่แสดงให้เห็นจากตัวอย่างเดียวกันอีกครั้งในรูปที่ 9.5

รูปที่ 9.5 แสดงให้เห็นการเขียนกำหนดโครงร่างในส่วนประกาศของ architecture ซึ่งจะให้ผลลัพธ์เช่นเดียวกับการเขียนในรูปที่ 9.1-9.2 นอกจากนั้นยังมีความง่ายในกฎเกณฑ์การเขียน แต่ในกรณีที่ผู้ออกแบบประสงค์ที่จะแก้ไขโครงร่างใหม่ นั้นหมายความว่าต้องแก้ไขในส่วนที่เป็น architecture ซึ่งก่อนที่จะจำลองการทำงานรูปแบบ จะต้องทำการวิเคราะห์ (analysis และ compile) ส่วนที่เป็น architecture ใหม่ทั้งหมด ในทางตรงข้ามการเขียนกำหนดโตรงร่างตามรูปที่ 9.1-9.2 นั้น ผู้ออกแบบสามารถที่จะยังคงรูปแบบของ architecture เดิมไว้ได้ เพียงแต่แก้ไขเฉพาะส่วนที่เป็น configuration ใหม่ ซึ่งโดยทั่วไปแล้วจะมีขนาดเล็กกว่าส่วนที่เป็น architecture ทั้งหมด และทำการวิเคราะห์เฉพาะส่วนนี้ใหม่ก่อนที่จะจำลองการทำงานต่อไป

---

[3] เช่นเดียวกับ 1

```
LIBRARY parts;

ARCHITECTURE structure OF and_or IS
    COMPONENT and2
        PORT (   and1, and2: IN BIT;
                 and_out: OUT BIT );
    END COMPONENT;
-- configuration specification in architecture
    FOR u1:  and2 USE ENTITY WORK.and2(behave);

    COMPONENT or2
        PORT (   or1, or2: IN BIT;
                 or_out: OUT BIT  );
    END COMPONENT;
-- configuration specification in architecture
    FOR u2:  or2 USE CONFIGURATION parts.or2_con;

    SIGNAL internal_1: BIT;
BEGIN
    u1:and2  PORT MAP (in1, in2, internal_1);
    u2:or2    PORT MAP (in3, internal_1, out_1);
END structure;
```

รูปที่ 9.5: Configuration specification in architecture

9.3 Overloading

ในภาษา VHDL การทำ overloading หมายถึงการใช้ชื่อ (อาจจะเป็นชื่อของ FUNCTION หรือ operator ต่างๆ) เดียวกัน ให้มีความหมายแตกต่างกัน ทั้งนี้ความแตกต่างกันขึ้นอยู่กับความหมายในการใช้ชื่อนั้นๆ เอง ในภาษา VHDL สามารถทำ overloading ได้สามลักษณะคือ

- Subprogram Overloading
- Enumerated Value Overloading
- Operator Overloading

รูปที่ 9.6 แสดง package declaration ที่ภายในประกอบด้วยการประกาศ TYPE ประเภทต่างๆ 4 ประเภท ตลอดจนโปรแกรมย่อยสำหรับการหมุนซ้าย (rotate left) ของตัวประกอบ (element) ที่อยู่ใน vector

```
PACKAGE util IS
    TYPE mvl3 IS ('X', '0', '1');
    TYPE mvl4 IS ('X', '0', '1', 'Z');
    TYPE mvl3_vector IS ARRAY (natural RANGE <>) OF mvl3;
    TYPE mvl4_vector IS ARRAY (natural RANGE <>) OF mvl4;
-- overloaded functions:
    FUNCTION rotate_left (input: mvl3_vector) RETURN mvl3_vector
    FUNCTION rotate_left (input: mvl4_vector) RETURN mvl4_vector
END util;
```

รูปที่ 9.6: **ตัวอย่างการทำ** subprogram (FUNCTION) overloading

ส่วนของ package declaration มีการประกาศ TYPE 4 ประเภทได้แก่ mvl3, mvl4, mvl3_vector และ mvl4_vector นอกจากนั้นยังมี overloaded function ชื่อ **rotate_left** ในบทที่ 8 ได้ศึกษาถึงโครงสร้างและการเขียนโปรแกรมย่อยไปแล้ว โดยที่โปรแกรมย่อยจะต้องมีชื่อ (identifier) รายชื่อพารามิเตอร์ของโปรแกรมย่อย (formal parameter list) และ TYPE ของ ตลอดจน TYPE ของค่าที่จะส่งกลับ (RETURN TYPE) ในตัวอย่างจะเห็นได้ว่ามีการกำหนด FUNCTION ที่ใช้ชื่อเดียวกันคือ rotate_left นั้นหมายความว่า FUNCTION ทั้งสอง overload ซึ่งกันและกัน ปัญหามีอยู่ว่าระบบวิเคราะห์ทำงานอย่างไรเมื่อมีกรณีเช่นนี้

จาก formal parameter list ของ FUNCTION ทั้งสองที่มี TYPE ต่างกันคือ mvl3_vector กับ mvl4_vector ฉนั้นการเรียก FUNCTION ในรูปแบบโดยการกำหนด passing parameter list ด้วยนั้น TYPE ของพารามิเตอร์นี้เองที่จะเป็นตัวกำหนดว่า FUNCTION ที่ชื่อ rotate_left ใดจะถูกเรียก เพราะระบบวิเคราะห์จะตรวจสอบ TYPE ของพารามิเตอร์ก่อนทุกครั้ง เช่นถ้า passing parameter list มี TYPE เป็น mvl3_vector เมื่อนั้น FUNCTION ที่ formal parameter list เป็น mvl3_vector เช่นกันจะถูกเรียก

ฉนั้นจะเห็นได้ว่าการทำ function overloading นั้นมีประโยชน์มากในการเขียนฟังก์ชันการทำงานแบบเดียวกันที่มี TYPE ต่างกัน นอกจากนั้นการประกาศใช้ TYPE ประเภท mvl3 ที่มีองค์ประกอบ 'X', '0' และ '1' กับประเภท mvl4 ที่มีองค์ประกอบ 'X', '0', '1' และ 'Z' จะเห็นได้ว่า mvl3 นั้นเป็นกลุ่มย่อย (subset) ของ mvl4 ลักษณะเช่นนี้เรียกว่า enumerated overloading [4]

---

[4] เช่นเดียวกับ TYPE ประเภท BIT

สุดท้ายของบทนี้ที่ยังไม่ได้กล่าวถึงได้แก่ operator overloading ในภาษา VHDL เปิดโอกาสให้ผู้ออกแบบกำหนดการทำงานของ operator ใหม่ได้ การทำงานของ operator นี้อาจจะเป็น operator มาตรฐานที่ระบบกำหนด (predefined operator) ที่กล่าวไว้ในบทที่ 1 หรือจะเป็น operator ที่ผู้ออกแบบกำหนดขึ้นใหม่ (user's defined operator)

9.4 Passive Process

ทางเทคนิค passive process ได้แก่ขบวนการทำงานใดๆ ที่ไม่มีผลกระทบต่อพฤติกรรมของสัญญาณ ในระหว่างการจำลองการทำงาน นั้นหมายความว่าห้ามใช้ชุดคำสั่ง signal assignment ใน passive process และการที่ไม่มีผลต่อพฤติกรรมของสัญญาณต่างๆ นี้เอง หน้าที่หลักของ passive process ได้แก่การเฝ้าตรวจพฤติกรรมของรูปแบบระหว่างการจำลองการทำงาน โดยที่จะแสดงรายงาน (report) ในรูปของข้อความ ในขณะจำลองการทำงานเกิดพฤติกรรมที่ผิดปกติขึ้น

โครงสร้างของ passive process นั้นสามารถที่จะอยู่ได้ในส่วนที่เป็น entity และ architecture ของรูปแบบได้ สำหรับ PROCESS ที่อยู่ในส่วนของ entity นั้นจะเป็น passive process เพียงอย่างเดียว เพราะในส่วนนี้จะไม่มีผลต่อพฤติกรรมของรูปแบบ โดยทั่วไปแล้วขบวนการตรวจสอบทางเวลาในระหว่างจำลองการทำงาน จะเขียนอยู่ในรูปของ passive process ในส่วนของ entity ตามตัวอย่างที่แสดงในรูปที่ 9.7

จากตัวอย่างในรูปที่ 9.7 ที่เป็นส่วนของ entity ที่นอกจากจะมีการประกาศ PORT และ GENERIC แล้ว ยังประกอบด้วยชุดคำสั่ง BEGIN ซึ่งเป็นจุดเริ่มต้นของ passive process จากการกำหนดสัญญาณ **clk** ให้เป็น sensitivity list ของ PROCESS และสัญญาณนี้จะเป็นตัวกระตุ้นการทำงานเมื่อเกิดมี event ขึ้น และถ้าเป็นการเปลี่ยนแปลงจาก '0' เป็น '1' (rising edge) PROCESS จะตรวจสอบการเปลี่ยนแปลงครั้งสุดท้ายของสัญญาณ **d** อีกครั้ง ถ้าปรากฏว่าสัญญาณ **d** มีการเปลี่ยนแปลงครั้งสุดท้างไม่นานกว่าเวลา **tsu** ระบบจำลองการทำงานจะหยุด และมีรายงานปรากฏขึ้นว่า **"Setup Violation on th d Input!"** นั้นคือการตรวจสอบ setup time ของสัญญาณ **d** นั้นเอง

```
ENTITY reg IS
    GENERIC (tsu: TIME := 3 NS);
    PORT (    d: IN BIT_VECTOR (3 DOWNTO 0);
              clk: IN BIT;
              q: OUT BIT_VECTOR (3 DOWNTO 0)  );
BEGIN
    PROCESS (clk)
    BEGIN
        IF clk = '1' THEN
            ASSERT (d'LAST_EVENT > tsu)
                REPORT "Setup Violation on th d Input!"
                SEVERITY ERROR;
        END IF;
    END PROCESS;
END reg;
```

รูปที่ 9.7: ตัวอย่างการใช้ passive process

9.5 ตัวบอกคุณสมบัติ (Attributes)

ในตัวอย่างของรูปแบบที่กล่าวมาแล้วในบทก่อนๆ ได้มีการนำ attribute มาใช้บ้างแล้วเช่นการตรวจสอบขอบการเปลี่ยนแปลงของสัญญาณ ในบทนี้จะเป็นการศึกษาถึงหลักการและการใช้ ในภาษา VHDL นั้นแบ่ง attribute ออกเป็นสองประเภทคือ predefined attribute และ user defined attribute ในขอบเขตของหนังสือเล่มนี้จะขอกล่าวเฉพาะ predefined attribute เท่านั้น

ในภาษา VHDL นั้น predefined attribute เป็นกลไกที่มีประโยชน์และประสิทธิภาพ สำหรับการเขียนรูปแบบบรรยายพฤติกรรมและคุณสมบัติของ hardware ซึ่ง predefined attribute นี้สามารถใช้ได้กับ ARRAY, TYPE และ SIGNAL ที่จะกล่าวในรายละเอียดต่อไป การใช้ attribute มีกฎการเขียนตามรูปที่ 9.8

```
array_or_type_or_signal_name' ATTRIBUTE_NAME
```

รูปที่ 9.8: การใช้ attribute

สัญลักษณ์ขีด (') อ่านว่า **tick**

9.5.1 Array Attributes

ใช้สำหรับหาคุณสมบัติเหล่านี้ ช่วงระยะ (range), ความยาวหรือขนาด (length) และขอบเขตของ ARRAY และสามารถใช้ได้กับ object ที่เป็น ARRAY ได้เท่านั้น รูปที่ 9.9 เป็นตัวอย่างการใช้ array attribute

```
TYPE mvl4 IS ('X', '0', '1', 'Z');
TYPE mvl4_4by8 IS ARRAY (3 DOWNTO 0, 0 TO 7) OF mvl4;

SIGNAL sq_4_8: mvl4_4by8;
```

รูปที่ 9.9(a): การประกาศ TYPE และ SIGNAL

	3	**2**	**1**	**0**
0				
1				
2				
3				
4				
5				
6				
7				

รูปที่ 9.9(b): โครงสร้างของ ARRAY: mvl4_4by8

Attribute	คำอธิบาย	ตัวอย่าง	ผลลัพธ์
'LEFT	left bound	sq_4_8'LEFT(1)	3
'RIGHT	right bound	sq_4_8'RIGHT sq_4_8'RIGHT(2)	0 7
'HIGH	upper bound	sq_4_8'HIGH(2)	7
'LOW	lower bound	sq_4_8'LOW(2)	0
'RANGE	range	sq_4_8'RANGE(2) sq_4_8'RANGE(1)	0 TO 7 3 DOWNTO 0
'REVERSE_RANGE	reverse range	sq_4_8'REVERSE_RANGE(2) sq_4_8'REVERSE_RANGE(1)	7 DOWNTO 0 0 TO 3
'LENGTH	length	sq_4_8'LENGTH	4

รูปที่ 9.9(c): Array Attribute

9.5.2 Type Attributes

ใช้สำหรับเข้าสู่องค์ประกอบของ TYPE ที่ถูกกำหนดขึ้น (user-defined type และ predefined type) และไม่สามารถใช้ได้กับ TYPE ประเภท ARRAY เนื่องจากมี type attribute อยู่บางอย่างที่ใช้คำเดียวกับ array attribute ฉนั้นสิ่งที่สำคัญในการใช้อาจจะให้ความหมายที่แตกต่างไปเช่น ถ้าใช้ตัวบอกคุณสมบัติ **'RIGHT** กับ enumeration type ผลลัพธ์ที่ได้คือ องค์ประกอบตัวขวาสุดของ TYPE นั้นๆ ส่วนใน array attribute หมายถึงการหาดัชนีขององค์ประกอบที่อยู่ขวาสุดของ array นอกจากนั้นตัวบอกคุณสมบัติ **'BASE**, **'LEFT**, **'RIGHT**, **'HIGH** และ **'LOW** สามารถใช้ได้กับ TYPE ประเภท scalar ในขณะที่ตัวบอกคุณสมบัติ **'POS**, **'VAL**, **'SUCC**, **'PRED**, **'LEFTOF** และ **'RIGHTOF** สามารถใช้ได้เฉพาะกับ TYPE ประเภท INTEGER, enumeration และ physical เท่านั้น ตัวอย่างเช่นการใช้ 'VAL(2) กับ enumeration type ผลลัพธ์คือองค์-ประกอบตรงตำแหน่งตัวที่ 2 ของ TYPE นั้น ตำแหน่งขององค์ประกอบประเภท enumeration จะเริ่มนับจากซ้ายไปขวาโดยเริ่มต้นที่เลข 0

รูปที่ 9.10 แสดงให้เห็นตัวอย่างการใช้ type attribute ผลลัพธ์ที่ได้จากตัวบอกคุณสมบัติ **'LEFT**, **'RIGHT**, **'HIGH** และ **'LOW** จะเกี่ยวข้องกับ TYPE หรือ SUBTYPE ในขณะที่ตัวบอกคุณสมบัติ **'POS**, **'VAL**, **'SUCC**, **'PRED**, **'LEFTOF** และ **'RIGHTOF** จะให้ฟังก์ชันเฉพาะที่เป็นรากฐาน (base) ของ SUBTYPE เช่นในกรณีของ tit'POS('X') จะได้ผลลัพธ์เท่ากับ 3 ทั้งๆ ที่ 'X' ไม่ได้เป็นองค์ประกอบของ tit แต่ tit เป็น SUBTYPE ของ qit

```
TYPE qit IS ('0', '1', 'Z', 'X');
SUBTYPE tit IS qit RANGE '0' TO 'Z';
```

รูปที่ 9.10(a): **การประกาศ** TYPE **และ** SUBTYPE

Attribute	**คำอธิบาย**	**ตัวอย่าง**	**ผลลัพธ์**
'BASE	base of TYPE	tit'BASE	qit
'LEFT	left bound of TYPE or SUBTYPE	tit'LEFT qit'LEFT	'0' '0'
'RIGHT	right bound of TYPE or SUBTYPE	tit'RIGHT qit'RIGHT	'Z' 'X'
'HIGH	upper bound of TYPE or SUBTYPE	INTEGER'HIGH tit'HIGH	large 'Z'
'LOW	lower bound of TYPE or SUBTYPE	POSITIVE'LOW qit'LOW	1 '0'
'POS(V)	position of value V in base of TYPE	qit'POS('Z') tit'POS('X')	2 3
'VAL(P)	value at position P in base of TYPE	qit'VAL(3) tit'VAL(3)	'X' 'X'
'SUCC(V)	value, after value V in base of TYPE	tit'SUCC('Z')	'X'
'PRED(V)	value, before value V in base of TYPE	tit'PRED('1')	'0'
'LEFTOF(V)	value, left of value V in base of TYPE	tit'LEFTOF('1') tit'LEFTOF('0')	'0' ERROR
'RIGHTOF(V)	value, right of value V in base of TYPE	tit'RIGHTOF('1') tit'RIGHTOF('Z')	'Z' 'X'

รูปที่ 9.10(b): Type Attribute

9.5.3 Signal Attribute

เป็น attribute ที่ใช้กับ object ชนิด SIGNAL ที่มี TYPE ประเภทต่างๆ และใช้สำหรับหาการเปลี่ยนแปลงของระดับสัญญาณ (event) และการเกิด transaction หรือช่วงเวลาของการเกิด event และ transaction ตัวบอกคุณสมบัติ (attribute) เหล่านี้มีประโยชน์มากสำหรับการเขียนรูปแบบบรรยายพฤติกรรมของ hardware

ตัวบอกคุณสมบัติ **'STABLE**, **'EVENT**, **'LAST_EVENT** และ **'LAST_VALUE** ใช้ควบคู่กับการเกิด event ของสัญญาณ ตัวอย่างเช่น signal_a'EVENT ใช้กับ object ชนิด SIGNAL (ในที่นี้คือ signal_a) จะให้ผลลัพธ์ TRUE (boolean type) เมื่อสัญญาณเกิดการเปลี่ยนแปลงของระดับค่า ตัวบอกคุณสมบัติ **'QUIET**, **'ACTIVE**, **'LAST_ACTIVE** และ **'TRANSACTION** ใช้ควบคู่กับการเกิด transaction ของสัญญาณ ตัวอย่างเช่น signal_b'ACTIVE ใช้กับ object ชนิด SIGNAL (ในที่นี้คือ signal_b) จะให้ผลลัพธ์ TRUE (boolean type) เมื่อเกิด transaction กับสัญญาณ signal_b[5]

เนื่องจากผลลัพธ์ของตัวบอกคุณสมบัติ 'DELAYED, 'STABLE, 'QUIET และ 'TRANSACTION จะเป็น object ชนิด SIGNAL ฉนั้นผู้ออกแบบสามารถนำผลลัพธ์ดังกล่าวไปใช้ได้เหมือนกับ object ชนิด SIGNAL อีกตัวหนึ่ง ตัวอย่างเช่น signal_c'DELAYED'STABLE เป็นต้น

รูปที่ 9.11 แสดงคำอธิบายและวิธีการใช้ signal attribute โดยสมมติให้สัญญาณ s1 มี TYPE ประเภท BIT

---

[5] การเกิด transaction กับสัญญาณจะไม่เป็นสาเหตุที่ทำให้เกิดการเปลี่ยนแปลงระดับค่าของสัญญาณ แต่ถ้ามีการเปลี่ยน-แปลงดังกล่าวขึ้น นั้นแสดงว่า transaction นั้นทำให้เกิด event ขึ้นด้วย

<table>
<tr><td>Attribute</td><td>TR/EV</td><td>ตัวอย่าง</td><td>ชนิด object</td><td>TYPE</td></tr>
<tr><td colspan="5">คำอธิบาย</td></tr>
<tr><td>'DELAYED</td><td>-</td><td>s1'DELAYED(5 NS)</td><td>SIGNAL</td><td>เหมือน s1</td></tr>
<tr><td colspan="5">การเลียนแบบ (copy) สัญญาณ s1 โดยมีการหน่วงเวลา 5 NS. ถ้าไม่กำหนดพารามิเตอร์เวลา หรือเท่ากับ 0 จะหน่วงเวลาเท่ากับ δ, มีผลเช่นเดียวกับ transport delay ของ s1</td></tr>
<tr><td>'STABLE</td><td>EV</td><td>s1'STABLE(5 NS)</td><td>SIGNAL</td><td>BOOLEAN</td></tr>
<tr><td colspan="5">สัญญาณที่ได้จะเป็น TRUE ถ้า s1 ไม่เกิดการเปลี่ยนแปลงระดับสัญญาณอย่างน้อย 5 NS. ถ้าไม่กำหนดพา-รามิเตอร์เวลา หรือเท่ากับ 0 สัญญาณที่ได้จะเป็น TRUE ถ้า s1 ไม่เกิดการเปลี่ยนแปลงระดับสัญญาณขณะเวลาปัจจุบันในการจำลองการทำงาน (current simulation time)</td></tr>
<tr><td>'EVENT</td><td>EV</td><td>s1'EVENT</td><td>value</td><td>BOOLEAN</td></tr>
<tr><td colspan="5">ถ้า s1 เกิดการเปลี่ยนแปลงระดับสัญญาณขณะเวลาปัจจุบันในการจำลองการทำงาน จะทำให้ s1'EVENT มีผลลัพธ์ TRUE ในขณะเวลานั้น (เวลา δ)</td></tr>
<tr><td>'LAST_EVENT</td><td>EV</td><td>s1'LAST_EVENT</td><td>value</td><td>TIME</td></tr>
<tr><td colspan="5">ผลลัพธ์ที่ได้คือค่าของเวลาตั้งแต่สัญญาณ s1 เกิดการเปลี่ยนแปลงครั้งสุดท้าย ฉนั้นถ้า s1'EVENT เป็น TRUE ค่าของ s1'LAST_EVENT คือ 0 (หน่วยเวลา)</td></tr>
<tr><td>'LAST_VALUE</td><td>EV</td><td>s1'LAST_VALUE</td><td>value</td><td>เหมือน s1</td></tr>
<tr><td colspan="5">ได้แก่การหาค่าของสัญญาณ s1 ก่อนที่จะเกิดการเปลี่ยนแปลงระดับค่าครั้งหลังสุด</td></tr>
<tr><td>'QUIET</td><td>TR</td><td>s1'QUIET(5 NS)</td><td>SIGNAL</td><td>BOOLEAN</td></tr>
<tr><td colspan="5">สัญญาณที่ได้จะเป็น TRUE ถ้า s1 ไม่เกิด transaction อย่างน้อย 5 NS. ถ้าไม่กำหนดพารามิเตอร์เวลา หรือเท่ากับ 0 จะประมาณเวลาขณะจำลองการทำงาน</td></tr>
<tr><td>'ACTIVE</td><td>TR</td><td>s1'ACTIVE</td><td>value</td><td>BOOLEAN</td></tr>
<tr><td colspan="5">ถ้าสัญญาณ s1 ได้เกิด transaction ขึ้นในเวลาขณะจำลองการทำงาน s1'ACTIVE จะให้ผลลัพธ์ TRUE สำหรับเวลานั้น (เวลา δ)</td></tr>
<tr><td>'LAST_ACTION</td><td>TR</td><td>s1'LAST_ACTION</td><td>value</td><td>TIME</td></tr>
<tr><td colspan="5">ผลลัพธ์ที่ได้คือค่าของเวลาตั้งแต่สัญญาณ s1 เกิด transaction ครั้งสุดท้าย ฉนั้นถ้า s1'ACTIVE เป็น TRUE ค่าของ s1'LAST_ACTIVE คือ 0 (หน่วยเวลา)</td></tr>
<tr><td>'TRANSACTION</td><td>TR</td><td>s1'TRANSACTION</td><td>SIGNAL</td><td>BIT</td></tr>
<tr><td colspan="5">สัญญาณจะสลับระดับค่า (toggle) ทุกครั้งที่เกิด transaction ขึ้น</td></tr>
</table>

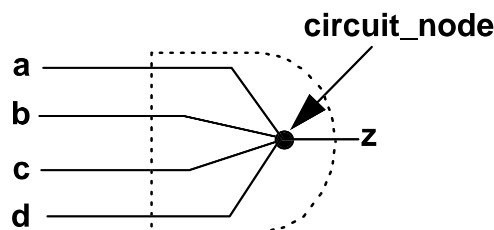
รูปที่ 9.11: Signal Attribute (TR := transaction, EV := event)

9.6 Resolution Functions

เนื่องจากโครงสร้างส่วนที่เป็น architecture design unit ของรูปแบบจะต้องประกอบด้วยชุดคำสั่งแบบแข่งขนานเท่านั้น ทำให้ไม่สามารถกำหนดค่าให้กับสัญญาณ (signal assignment) ตัวเดียวจากตัวขับ (driver) มากกว่าหนึ่งตัวในเวลาเดียวกัน (multiple signal assignment) ได้ ซึ่งนั่นคือข้อจำกัดของระบบจำลองการทำงาน เพราะว่าระบบจำลองการทำงานไม่สามารถกำหนดค่าที่แน่นอนให้กับสัญญาณได้ เนื่องจากเกิดข้อขัดแย้ง (conflict) ของสัญญาณตัวขับ (driver) ซึ่งในความเป็นจริงทางหลักการของฟิสิกส์ เหตุการณ์เช่นนี้สามารถเกิดขึ้นได้ ทั้งนี้ขึ้นอยู่กับเทคโนโลยีของวงจรอิเล็กทรอนิกส์ (เช่น TLL, ECL หรือ CMOS เป็นต้น) ตามที่รู้จักในลักษณะของวงจรที่เรียกว่า **wired-AND** หรือ **wired-OR** [6] เป็นต้น

ฉนั้นการที่ภาษา VHDL เป็นภาษาที่ใช้บรรยายพฤติกรรมของ hardware ในทุกลักษณะการทำงานเมื่อเกิดเหตุการณ์เช่นนี้ขึ้น จะเป็นปัญหาที่ต้องแก้ไขหรือหาทางออกให้กับระบบพัฒนา VHDL ซึ่งผู้ออกแบบสามารถกระทำได้โดยเขียนฟังก์ชันสำหรับแก้ปัญหาขึ้น หรือที่เรียกทั่วไปว่า resolution function ฟังก์ชันแก้ปัญหาจะทำหน้าที่กำหนดค่าให้กับสัญญาณ (ค่าดังกล่าวนี้เป็นไปตามการออกแบบของผู้เขียนรูปแบบ) เมื่อเกิดกรณีของ multiple signal assignment ขึ้น รูปที่ 9.12 เป็นตัวอย่างของ resolution function ที่ใช้แก้ปัญหาเพื่อทำให้เกิดผลของ wired-AND ขึ้น

ในรูปที่ 9.12 แสดงให้เห็นการเขียน resolution function สำหรับแก้ปัญหาการเกิด multiple driver (ในที่นี้เกิดกับสัญญาณ circuit_node) โดยให้มีการแก้ปัญหาเป็นฟังก์ชัน **wired-AND** การเรียก FUNCTION มาใช้นั้นใช้วิธีการเรียกในรูปของ expression ในส่วนประกาศกำหนด SIGNAL สำหรับสัญญาณที่จะเกิดปัญหา



รูปที่ 9.12(a): Resolution function **สำหรับวงจร** wired-AND

---

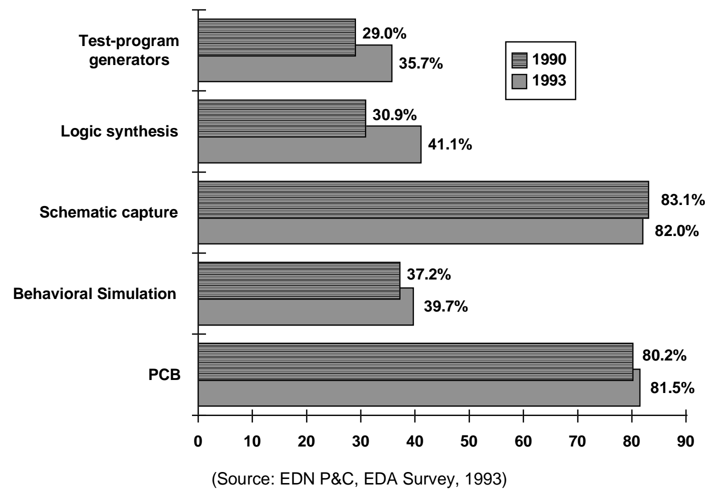
[6] "Basic Digital Circuit & Technologies", Adel S. Sedra, Kenneth C. Smith, Sannders College Publishing, 1991

```
ARCHITECTURE wired_and OF multiple_circuit IS
    FUNCTION anding (drivers : BIT_VECTOR) RETURN BIT IS
        VARIABLE accumulate : BIT := '1';
    BEGIN
        FOR i IN drivers'RANGE LOOP
            accumulate := accumulate AND drivers(i);
        END LOOP;
        RETURN accumulate;
    END anding;
    SIGNAL circuit_node : anding BIT;
BEGIN
    circuit_node <= a;
    circuit_node <= b;
    circuit_node <= c;
    circuit_node <= d;
            z  <= circuit_node;
END wired_and;
```

รูปที่ 9.12(b): Resolution function **สำหรับวงจร** wired-AND

9.7 อนาคตของภาษา VHDL ในการออกแบบระบบดิจิตอล

ตามที่กล่าวมาแล้วในบทที่ 1 ปัจจุบันการออกแบบระบบดิจิตอลมีการนำระบบ Electronic Design Automation (EDA) มาใช้ เพราะนอกจากจะทำให้ลดระยะเวลาตั้งแต่เริ่มต้นพัฒนาจนกระทั่งเป็นผลิตภัณฑ์สู่ตลาดให้สั้นลงแล้ว ยังทำให้ผลงานที่ได้รับการออกแบบมีความถูกต้องสูง ซึ่งทำให้ลดต้นทุนการพัฒนาได้มาก การนำระบบ EDA เข้ามาใช้ในการออกแบบทำให้เกิดเทคนิคใหม่ในการออกแบบได้แก่ Top-Down Design (เปรียบเทียบการเปลี่ยนแปลงของการใช้เครื่องมือช่วยออกแบบในลักษณะและงานต่างๆ ในรูปที่ 9.13)

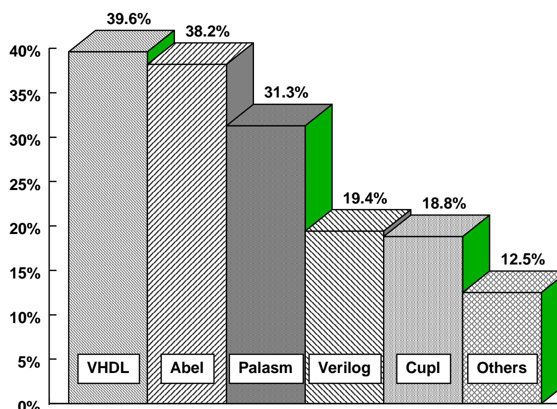


รูปที่ 9.13: **กราฟแสดงการเปลี่ยนแปลงของการใช้งานระบบ** EDA

จากรูปที่ 9.13 แสดงให้เห็นว่าอัตราการเพิ่มมากขึ้นของงาน logic synthesis, testprogram generators และ behavioral simulation นั้นสูงขึ้นมากในช่วงระยะ 3 ปีหลัง (1990-1993) ซึ่งทั้งหมดนี้มีส่วนเกี่ยวข้องโดยตรงกับการใช้ภาษา HDL ในทางตรงข้ามการใช้งานด้าน schematic capture ที่เป็นการใช้งานในการออกแบบลักษณะของ Bottom-Up Design กลับมีการใช้ลดลง

ฉนั้นภาษาที่เป็นลักษณะของ Hardware Description Language (HDL) จึงเป็นเครื่องมือสำคัญที่สุด เพราะเป็นจุดเริ่มต้นของการสังเคราะห์วงจร (circuit synthesis) ในระบบ EDA ที่สนับสนุนการออกแบบในลักษณะของ Top-Down Design นอกจากภาษา VHDL แล้วยังมีภาษา HDL อีกหลายระบบเช่น Verilog-HDL ที่มีคุณสมบัติคล้ายกัน แต่แตกต่างกันในเรื่องของกฎเกณฑ์ และนอกจากนั้นยังมีภาษาในลักษณะของ HDL ที่ใช้สังเคราะห์วงจรสำหรับอุปกรณ์บางประเภทเช่น Programmable Logic Device (PLD) ภาษากลุ่มนี้จะใช้ไดจำกัดเฉพาะระบบ EDA ซึ่งแตกต่างไปจากภาษา VHDL และ Verilog-HDL ที่ไม่จำกัดระบบพัฒนา (platform independent) การที่สมาคม IEEE รับภาษา VHDL มาเป็นมาตรฐาน จึงทำให้ในระยะหลังภาษา VHDL ได้รับความนิยมมากขึ้นตามลำดับ และมีแนวโน้มว่าจะเพิ่มขึ้นทุกปี ถึงอย่างไรก็ตามที่ระบบสังเคราะห์วงจรในปัจจุบันยังคงสามารถรับข้อมูล (input file) ได้ทั้งสองชนิด

นอกจากนั้นภาษา VHDL ยังได้รับการพัฒนาเพิ่มเติมให้สามารถเขียนบรรยายพฤติกรรมของวงจรที่อยู่ในรูปของระบบอนาล๊อก (anolog system) ซึ่งในปัจจุบันได้มีอยู่หลายระบบด้วยกัน และเป็นจุดเริ่มต้นของการพัฒนาภาษา ที่ใช้บรรยายระบบที่มีสัญญาณแบบผสม (mixed signal system) ต่อไป จะทำให้การออกแบบระบบอิเล็กทรอนิกส์ในอนาคตสมบูรณ์แบบยิ่งขึ้น



รูปที่ 9.14: สัดส่วนการใช้ภาษา HDL ในการออกแบบวงจรดิจิตอล

รูปที่ 9.14 แสดงสัดส่วนการใช้ภาษา HDL โดยวิศวกรออกแบบในการบรรยายพฤติกรรมระบบดิจิตอล ซึ่งทั้ง VHDL และ Verilog-HDL มีแนวโน้มสูงขึ้น ในขณะที่ภาษาอื่นๆ ลดบทบาทในวงการ EDA ลง

9.8 สรุป

บทนี้เป็นบทส่งท้ายของหนังสือเล่มนี้ โดยที่เนื้อหาต่อเนื่องจากบทก่อนๆ และได้เพิ่มเติมหัวข้อที่สำคัญ เพื่อจะให้เกิดความเข้าใจในการเขียนรูปแบบบรรยายพฤติกรรมของระบบดิจิตอลด้วยภาษา VHDL ซึ่งในบทนี้ได้กล่าวถึงหัวข้อดังนี้

- Configuration Specification
- Overloading

- Passive Process
- Attributes
- Resolution Function

ฉนั้นหลังจากที่ศึกษาในหนังสือเล่มนี้จบแล้ว ผู้อ่านจะมีความสามารถที่จะเขียนรูปแบบของระบบดิจิตอลได้ในระดับหนึ่ง สำหรับผู้ที่สนใจและต้องการศึกษาเพิ่มเติม โดยเฉพาะอย่างยิ่งแนวทางการเขียนเพื่อการสังเคราะห์ ผู้เรียบเรียงขอแนะนำให้ค้นคว้าเพิ่มเติมได้จากหนังสืออ้างอิงตามบรรณานุกรมข้อ 3

ผนวก ก.

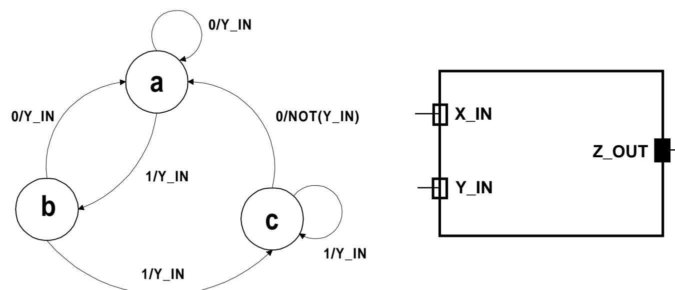
ตัวอย่างการจำลองรูปแบบระบบดิจิตอลด้วยภาษา VHDL

ผนวกนี้มีจุดประสงค์ เพื่อที่จะแสดงให้เห็นถึงขั้นตอนของการออกแบบ ในลักษณะของ Top-Down Design โดยใช้ภาษา VHDL ช่วยในการเขียนรูปแบบ และจำลองการทำงานเพื่อตรวจสอบความถูกต้องของรูปแบบที่เขียนขึ้น นอกจากนั้นยังแสดงให้เห็นข้อเปรียบเทียบระหว่าง การออกแบบในลักษณะของ structural และ behavioral ด้วย

ตัวอย่างที่ดีที่สุดได้แก่การออกแบบระบบ Finite State Machine (FSM) เพราะสามารถแสดงให้เห็นทุกรูปแบบของการเขียน เช่น behavioral-, structural- และ mixed level description ตลอดจนเทคนิคต่างๆ ในการออกแบบระบบดิจิตอลเช่น การกระจายเป็นกลุ่มย่อยตามฟังก์ชันของการทำงาน (design partition) เพื่อความสะดวกต่อการออกแบบ และตรวจสอบ เป็นต้น นอกจากนั้นในตัวอย่างยังแสดงให้เห็นถึงสภาพแวดล้อมของการเขียนรูปแบบ และการจำลองการทำงาน

รูปที่ ก-1 แสดงให้เห็น state diagram ที่เป็นจุดเริ่มต้นของการออกแบบ FSM และมุมมอง interface ของระบบ นอกจากนั้นเพื่อความสะดวกในการตรวจสอบจะรวมแหล่งกำเนิดสัญญาณนาฬิกา (clock) เข้าไว้ในระบบด้วย และควบคุมการทำงานด้วยสัญญาณ **enable**

จากมุมมองของ interface จะเห็นได้ว่าระบบนี้ประกอบด้วย สัญญาณ input สามสัญญาณคือ **X_IN**, **Y_IN** และ **enable** ส่วนสัญญาณ **Z_OUT** นั้นเป็น output และในที่นี้สมมติให้สัญญาณที่ผ่านตามช่องทาง เข้า-ออก (PORT) มี TYPE เป็น BIT



รูปที่ ก-1

ก่อนอื่นต้องมาศึกษาถึงการทำงานของระบบที่บรรยายในรูปของ state diagram ในภาพสัญญาณที่มีผลต่อพฤติกรรมของ FSM คือ X_IN, Y_IN และ Z_OUT ระบบจะทำหน้าที่ตรวจจับข้อมูลที่เข้ามาทางช่องผ่าน X_IN เพื่อตรวจจับรูปแบบของข้อมูล โดยที่ FSM ถูกเขียนขึ้นในลักษณะของ MEALY MACHINE[1] และมีการทำงานดังนี้

State: a ถ้า X_IN = '0' จะไม่ทำให้เกิดการเปลี่ยน state ในขณะเดียวกันค่า output (Z_OUT) จะเท่ากับค่าของสัญญาณ Y_IN

ถ้า X_IN = '1' จะทำให้ระบบเปลี่ยน state ไปยัง state 'b' และ Z_OUT = Y_IN

State: b ถ้า X_IN = '0' จะทำให้ระบบเปลี่ยน state กลับไปยัง state 'a' และ Z_OUT = Y_IN

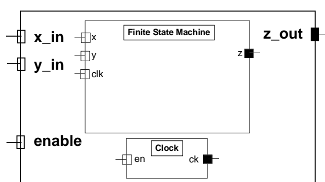
ถ้า X_IN = '1' จะทำให้ระบบเปลี่ยน state ไปยัง state และ Z_OUT = Y_IN

State: c ถ้า X_IN = '0' จะทำให้ระบบเปลี่ยน state กลับไปยัง state 'a' และ Z_OUT = NOT (Y_IN)

ถ้า X_IN = '1' จะทำให้ระบบไม่เปลี่ยน state และ Z_OUT = Y_IN

ดังนั้นพอสรุปได้ว่า FSM จะทำหน้าที่ตรวจจับค่าของสัญญาณที่เข้ามาทางช่อง X_IN เมื่อใดสัญญาณเข้ามาโดยมีลำดับ 1-1-0 เมื่อนั้น FSM จะทำการเปลี่ยนค่า Y_IN ให้มีค่าตรงข้ามกับที่เข้ามา และส่งไปยัง Z_OUT หรือที่เรียกว่า "sequence detector"

ขั้นตอนต่อไปคือการแบ่งระบบออกเป็นส่วนย่อย (design partition) โดยที่ในครั้งแรกจะได้ผลลัพธ์ของการแบ่งตามรูป ก-2 ได้แก่ Finite State Machine (FSM) และ clock

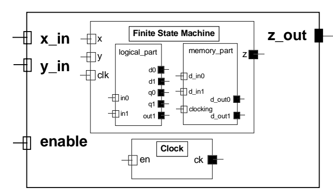


รูปที่ ก-2

---

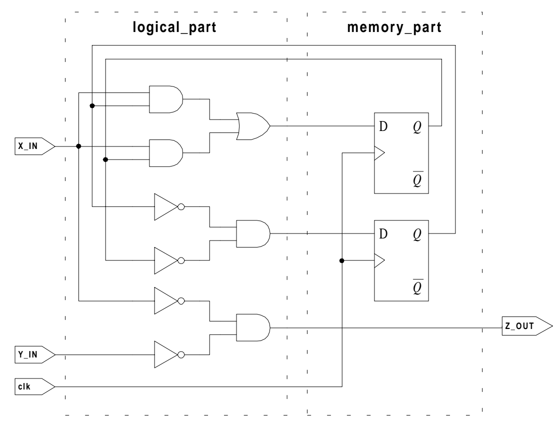
1 M. Morris Mano, "Digital Design", Printice-Hall International Inc., 1991

อย่างที่ทราบดีอยู่แล้วว่า FSM นั้นเกิดจากการประกอบกันของสองส่วนคือ logical part และ memory part ซึ่ง logical part มีหน้าที่กำหนด code ของ state และกำหนดค่า output ส่วน memory part มีหน้าที่เก็บ code ของ state ดังนั้นจึงสามารถแบ่ง FSM ออกไปได้อีกสองส่วนตามที่แสดงในรูปที่ ก-3



รูปที่ ก-3

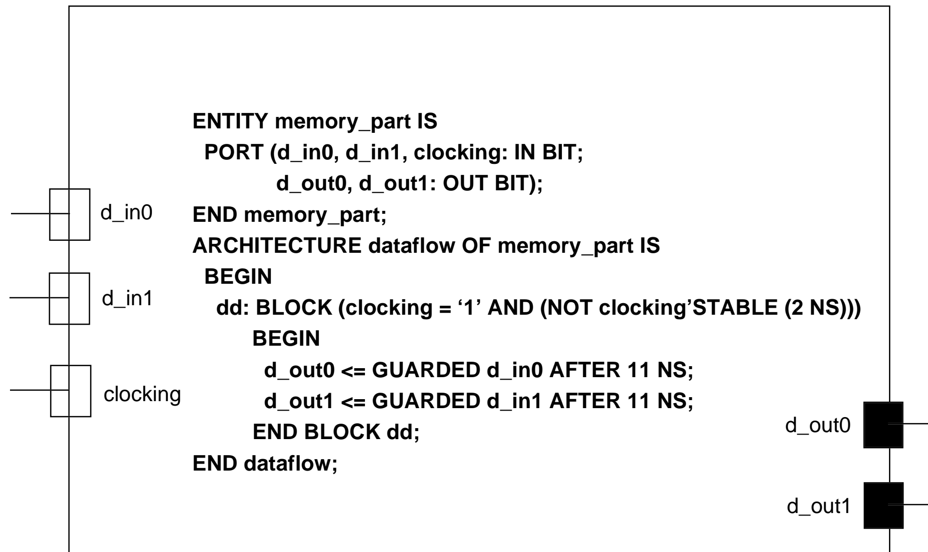
เพื่อไม่ให้เป็นการเสียเวลา จะไม่ขอกล่าววิธีทำต่อไปในที่นี้ สำหรับผู้ที่สนใจสามารถศึกษาได้จากเอกสารอ้างอิงที่แจ้งไว้ในบรรณานุกรม ขั้นตอนต่อไปคือการสร้างวงจรในระดับ gate-level ที่ประกอบด้วย gate ต่างๆ และ D-FlipFlop ที่ทำหน้าที่เป็นหน่วยความจำสำหรับ memory part ในรูปที่ ก-4 แสดงให้เห็นการสร้างวงจรระดับ gate level



รูปที่ ก-4

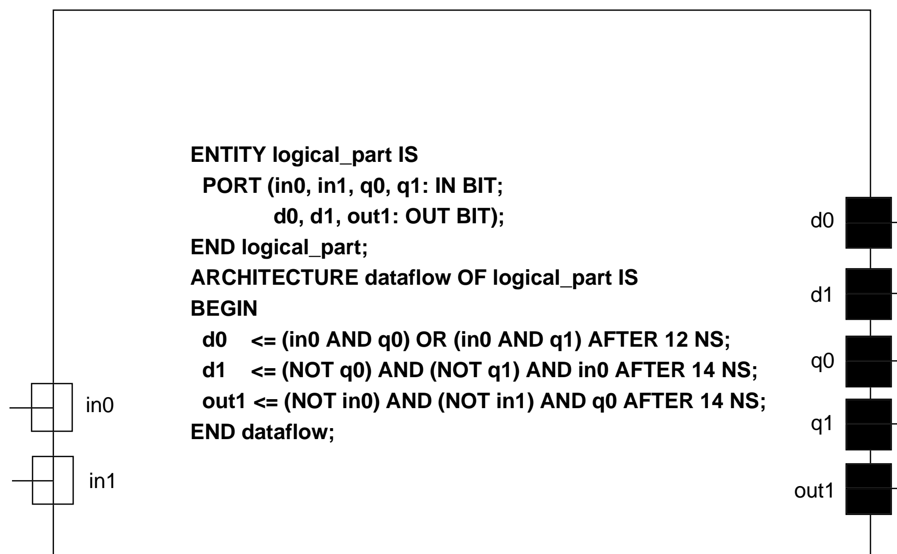
ในตัวอย่างนี้จะเห็นได้ว่า VHDL ยังไม่ได้ช่วยในการออกแบบอะไร เพราะยังคงต้องใช้ขบวนการแบบเดิมอยู่ ซึ่งในที่นี้คือจุดประสงค์ของหนังสือเล่มนี้ เพื่อที่จะแสดงในครั้งแรกว่าจะสามารถใช้ VHDL เขียนรูปแบบของอุปกรณ์ย่อยเหล่านี้อย่างไร และนำมาประกอบกันอย่างไรจนเป็นระบบขึ้นมาได้ ในรูปที่ ก-5.1 - ก-5.5 จะแสดงให้เห็น VHDL model ที่ใช้แทนอุปกรณ์ต่างๆ ในรูปของ behavioral description ตลอดจนการเชื่อมต่อระหว่างกันในรูปของ structural description

Memory Part



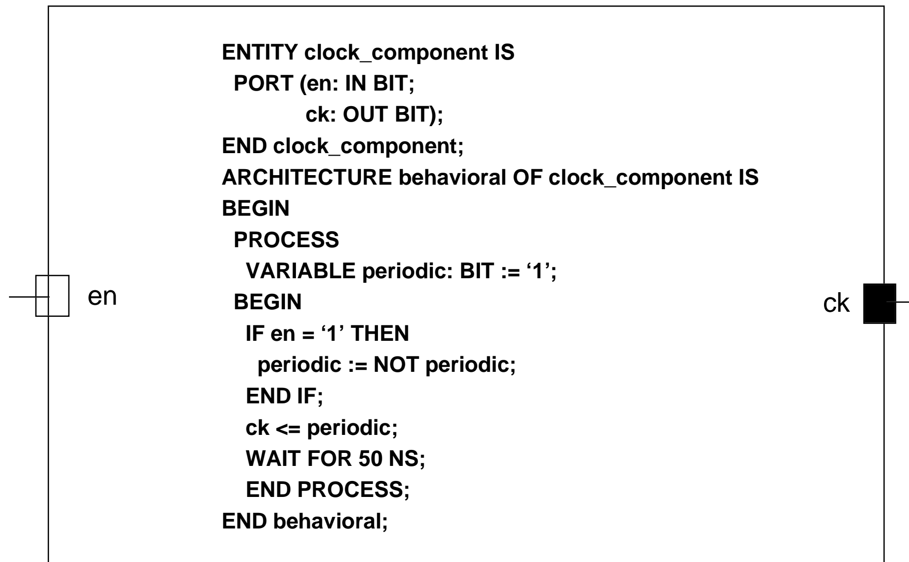
รูปที่ ก-5.1

Logical Part



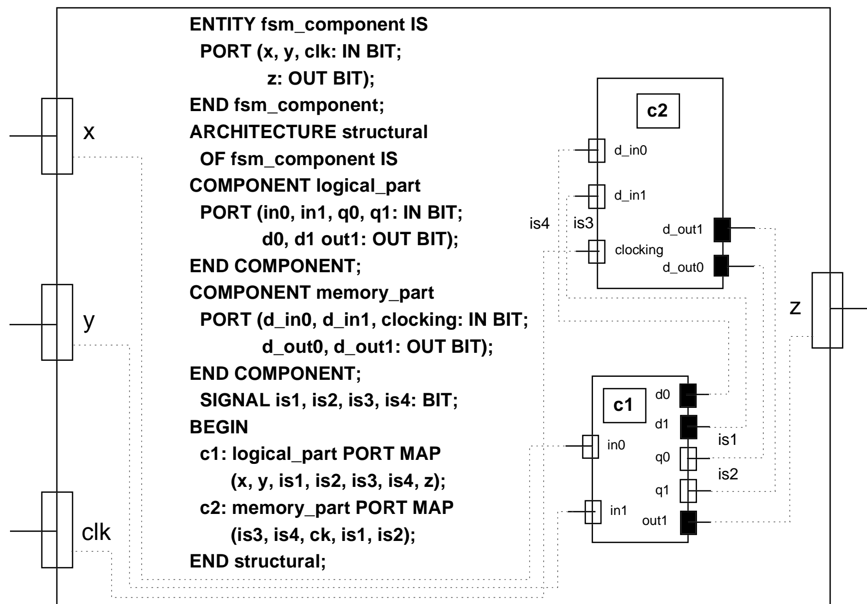
รูปที่ ก-5.2

Clock Component



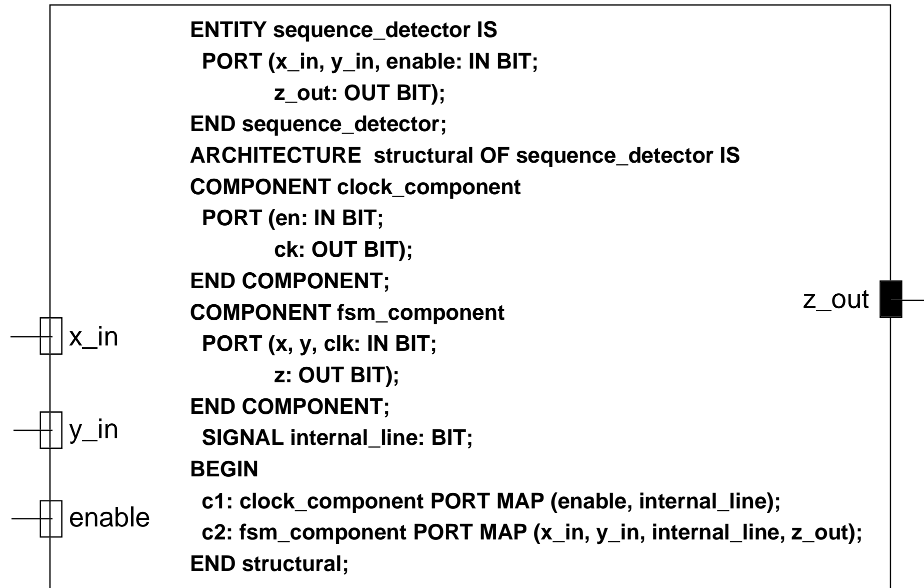
รูปที่ ก-5.3

Finite State Machine Component



รูปที่ ก-5.4

Sequence Detector



รูปที่ ก-5.5

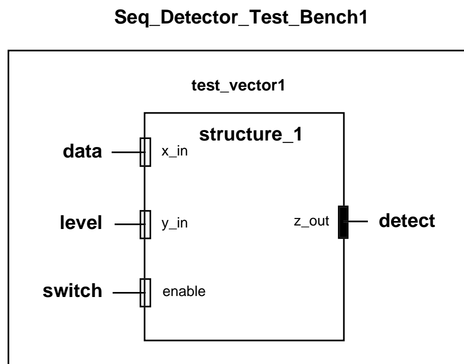
เพื่อแสดงให้เห็นถึงโครงสร้างของข้อมูล VHDL source file สำหรับนำไปวิเคราะห์ด้วยระบบพัฒนา VHDL-Analyzer และ Simulator จึงสรุปข้อมูลดังกล่าวตามที่แสดงต่อไปนี้

```
----------------------------------------------------------------------------------------------
--  Interface description for sequence detector.
----------------------------------------------------------------------------------------------
-- Top-level
----------------------------------------------------------------------------------------------
ENTITYsequence_detector IS
    PORT ( x_in, y_in, enable: IN BIT;
              z_out: OUT BIT);
END sequence_detector;
---------------------------------------------------------------------------------------------
--  Interface descriptions for sequence detector
--  after 1st partition.
----------------------------------------------------------------------------------------------
-- Interface description for the finite state machine component.
----------------------------------------------------------------------------------------------
ENTITYfsm_component IS
    PORT ( x, y, clk: IN BIT;
              z: OUT BIT);
END fsm_component;
---------------------------------------------------------------------------------------------
-- Interface description for the clock component.
---------------------------------------------------------------------------------------------
ENTITYclock_component IS
    PORT ( en: IN BIT;
              clk: OUT BIT);
END clock_component;
```

```
------------------------------------------------------------------------------------------
--  Interface description for sequence detector
--  after 2nd partition.
------------------------------------------------------------------------------------------
-- Interface description for the logical part.
------------------------------------------------------------------------------------------
ENTITYlogical_part IS
    PORT ( in0, in1, q0, q1: IN BIT;
              d0, d1, out1: OUT BIT);
END logical_part;
------------------------------------------------------------------------------------------
-- Interface description for the memory part.
------------------------------------------------------------------------------------------
ENTITYmemory_part IS
    PORT ( d_in1, d_in1, clocking: IN BIT;
              d_out0, d_out1: OUT BIT);
END memory_part;
------------------------------------------------------------------------------------------
-- Architecture description for memory part.
------------------------------------------------------------------------------------------
ARCHITECTURE dataflow OF memory_part IS
BEGIN
  dd: BLOCK (clocking = '1' AND (NOT clocking'STABLE(2 NS)))
     BEGIN
       d_out0 <= GUARDED d_in0 AFTER 11 NS;
       d_out1 <= GUARDED d_in1 AFTER 11 NS;
     END BLOCK dd;
END dataflow;
----------------------------------------------------------------------------------------
-- Architecture description for logical part.
----------------------------------------------------------------------------------------
ARCHITECTURE dataflow OF logical_part IS
BEGIN
  d0      <= (in0 AND q0) OR (in0 AND q1)   AFTER 12 NS;
  d1      <= (NOT q0) AND (NOT q1) AND in0  AFTER 14 NS;
  out1    <= (NOT in0) AND (NOT in1) AND q0 AFTER 14 NS;
END dataflow;
---------------------------------------------------------------------------------------
-- Archtecture description for clock component.
---------------------------------------------------------------------------------------
ARCHITECTURE behavioral OF clock_component IS
BEGIN
  PROCESS
    VARIABLE periodic: BIT := '1';
  BEGIN
    IF en = '1' THEN
       periodic := NOT periodic;
    END IF;
    ck <= periodic;
    WAIT FOR 50 NS;
  END PROCESS;
END behavioral;
```

```
-------------------------------------------------------------------------------------------
-- Architecture description for finite state machine component.
-------------------------------------------------------------------------------------------
ARCHITECTURE structural OF fsm_component IS
-- component declaration
  COMPONENT logical_part
    PORT (in0, in1, q0, q1: IN BIT;
            d0, d1, out1: OUT BIT);
  END COMPONENT;
  COMPONENT memory_part
    PORT (d_in0, d_in1, clocking: IN BIT;
            d_out0, d_out1: OUT BIT);
  END COMPONENT;
--
  SIGNAL is1, is2, is3, is4: BIT; -- intermediate signals
BEGIN
 c1: logical_part PORT MAP (x, y, is1, is2, is3, is4, z);
 c2: memory_part PORT MAP (is3, is4, clk, is1, is2);
END structural;
-------------------------------------------------------------------------------------------
-- Archtecture description for sequence_detector.
-------------------------------------------------------------------------------------------
ARCHITECTURE structural_1 OF sequence_detector IS
-- component declaration
      COMPONENT clock_component
      PORT ( en: IN BIT;
              ck: OUT BIT);
      END COMPONENT;
      COMPONENT fsm_component
      PORT ( x, y, clk: IN BIT;
              z: OUT BIT);
      END COMPONENT;
      FOR ALL: fsm_component
          USE ENTITY WORK.fsm_component(structural);
      SIGNAL internal_line: BIT; -- internal line (signal)
BEGIN
  c1: clock_component PORT MAP (enable, internal_line);
  c2: fsm_component   PORT MAP (x_in, y_in, internal_line, z_out);
END structural_1;
```

เพื่อความสะดวกในการตรวจสอบความถูกต้องของการทำงาน จำเป็นต้องสร้าง model ใหม่ขึ้นมา โดยที่ model นี้จะมีหน้าที่ผลิตชุดของสัญญาณตรวจสอบ (test vector) แล้วป้อนให้กับ model ที่ต้องการตรวจสอบ แล้วตรวจวัดค่า output เพื่อหาข้อผิด และดำเนินการแก้ไขต่อไป model ใหม่นี้มีชื่อว่า test_vector1 ดังแสดงในรูปที่ ก-6

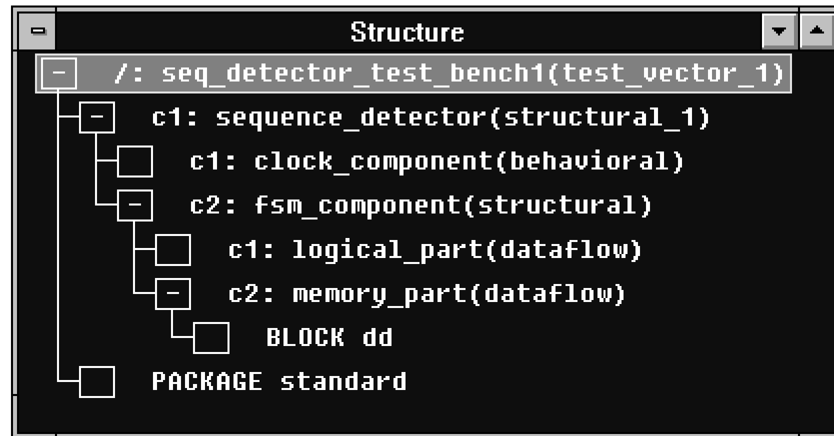


รูปที่ ก-6

การสร้างโครงสร้าง (structural) และรูปแบบของtest bench สำหรับจำลองการทำงานของ sequence detector ตามรูปแบบของ structure_1 สามารถเขียนได้ตามที่แสดงต่อไปนี้ รูปแบบดังกล่าวจะทำหน้าที่สร้างรูปแบบของสัญญาณที่ป้อนให้กับ test bench และจากระบบจำลองการทำงานสามารถที่จะตรวจผลลัพธ์ของการตอบสนองได้

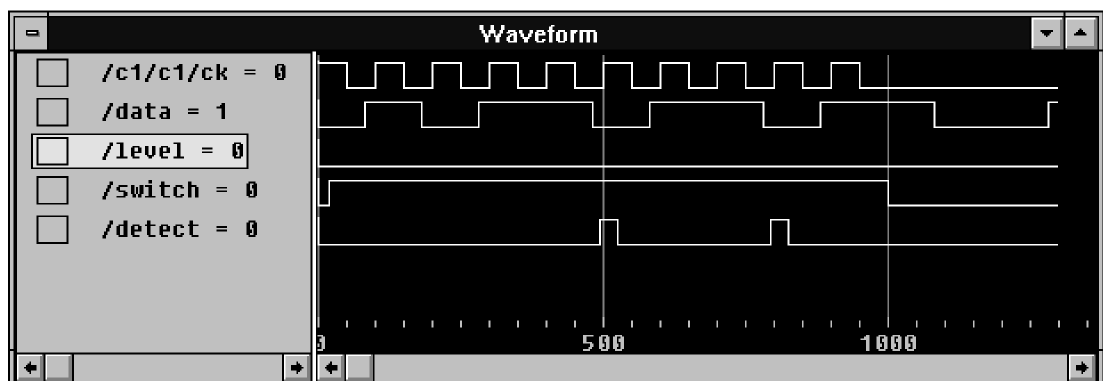
```
-- VHDL Description of Test Bench
--
ENTITY seq_detector_test_bench1 IS
END seq_detector_test_bench1;
--
ARCHITECTURE test_vector_1 OF seq_detector_test_bench1 IS
 COMPONENT sequence_detector
     PORT ( x_in, y_in, enable: IN BIT;
               z_out: OUT BIT  );
   END COMPONENT;
   FOR ALL: sequence_detector
      USE ENTITYWORK.sequence_detector(structural_1);
   SIGNAL data:      BIT:= '0';
   SIGNAL level:     BIT:= '0';
   SIGNAL switch:    BIT:= '0';
   SIGNAL detect:    BIT;
BEGIN
 c1: sequence_detector PORT MAP (data, level, switch, detect);
 c2: data   <= '0',
           '1' AFTER    80    NS,
           '0' AFTER   180    NS,
           '1' AFTER   280    NS,
           '0' AFTER   480    NS,
           '1' AFTER   580    NS,
           '0' AFTER   780    NS,
           '1' AFTER   880    NS,
           '0' AFTER 1080    NS,
           '1' AFTER 1280    NS;
 c3: switch <=  '1' AFTER     20 NS,
                '0' AFTER 1000 NS;
END test_vector_1;
```

รูปที่ ก-7 แสดงให้เห็นลักษณะของโครงสร้าง test bench ที่ภายในแต่ละขั้นจะประกอบด้วยอุปกรณ์ที่ได้เขียนเป็นรูปแบบ VHDL ในลักษณะของ hierarchical design



รูปที่ ก-7

หลังจากที่ปล่อยให้ระบบจำลองการทำงาน เริ่มทำงานค่าต่างๆ ตามรูปแบบที่กำหนดจะถูกป้อนให้กับ sequence detector ซึ่งจะได้ผลลัพธ์ของการตอบสนองทาง output ตามรูปที่ ก-8



รูปที่ ก-8

จากรูปที่ ก-8 ในหน้าต่าง Structure แสดงให้เห็นโครงสร้างที่ซ้อนกันอยู่ของ model ซึ่งเป็นไปตามการแบ่งกลุ่ม การออกแบบลักษณะนี้เรียกว่า hierarchical design ในหน้าต่าง Waveform แสดงให้เห็นการทำงานของ model เมื่อถูกกระตุ้นด้วย test vector ที่เป็น stimulus จากรูปร่างของ waveform จะเห็นว่า model ทำงานถูกต้อง

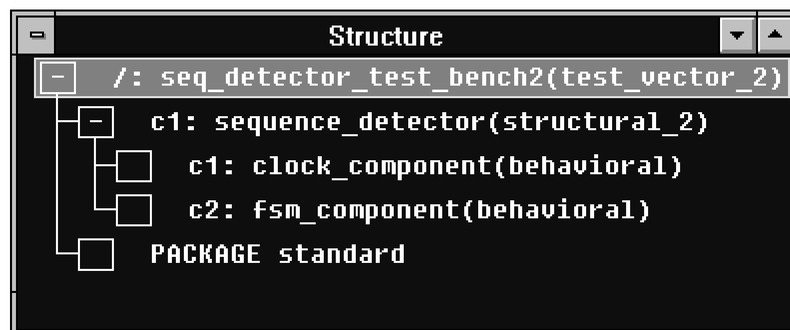
คราวนี้ทดลองเขียน model ของ FSM ใหม่ โดยไม่ใช้การเขียนแบบผสม แต่จะบรรยายให้อยู่ในรูป behavioral description เพียงอย่างเดียว ดังที่แสดงในรูปที่ ก-11

```
ARCHITECTURE behavioral OF fsm_component IS
  TYPE state IS (reset, got1, got11);
  SIGNAL present_state: state := reset;
BEGIN
  st: PROCESS (clk)
      BEGIN
       IF clk = '1' THEN
        CASE present_state IS
         WHEN reset =>
           IF x = '1' THEN
              present_state <= got1;
           ELSE
             present_state <= reset;
           END IF;
         WHEN got1 =>
           IF x = '1' THEN
             present_state <= got11;
           ELSE
             present_state <= reset;
           END IF;
WHEN got11 =>
           IF x = '1' THEN
             present_state <= got11;
           ELSE
             present_state <= reset;
           END IF;
         END CASE;
       END IF;
      END PROCESS st;
  ot: PROCESS (x)
      BEGIN
        IF (present_state = got11 AND x = '0') THEN
        z <= TRANSPORT '1' AFTER 14 NS;
        z <= TRANSPORT '0' AFTER 45 NS;
        ELSE
        z <= '0';
        END IF;
      END PROCESS ot;
END behavioral;
```

หลังจากนั้นนำมาประกอบกับ clock component เพื่อให้ได้ sequence detector ดังต่อไปนี้

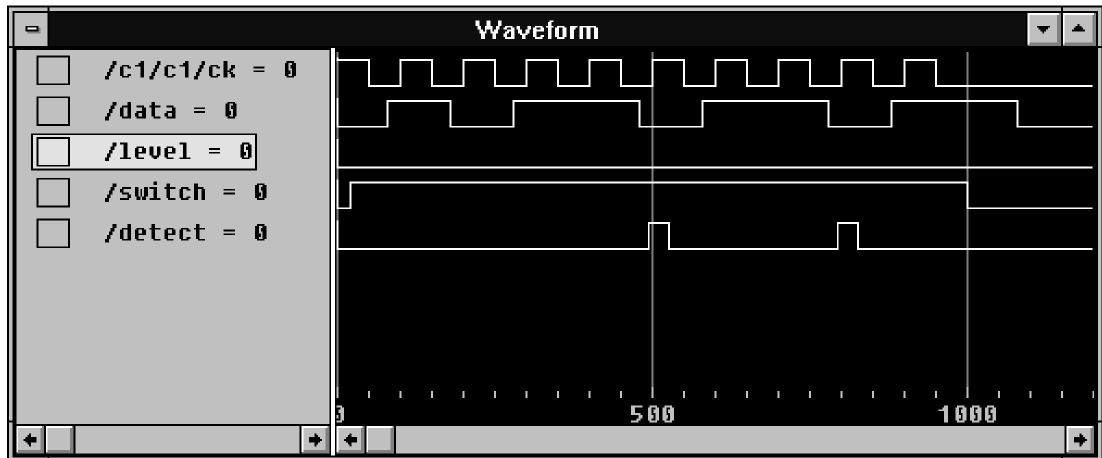
```
ARCHITECTURE structural_2 OF sequence_detector IS

-- component declaration
  COMPONENT clock_component
    PORT (en: IN BIT;
            ck: OUT BIT );
  END COMPONENT;
  COMPONENT fsm_component
    PORT (x, y, clk: IN BIT;
            z: OUT BIT);
  END COMPONENT;
  FOR ALL: fsm_component USE ENTITY
  WORK.fsm_component(behavioral);
--
  SIGNAL internal_line: BIT; -- internal line (signal)
BEGIN
  c1: clock_component PORT MAP (enable, internal_line);
  c2: fsm_component   PORT MAP (x_in, y_in, internal_line, z_out);
END structural_2;
```

รูปที่ ก-9 แสดงให้เห็นผลของการจำลองการทำงานของ model ใหม่ โดยที่ใช้ test vector เดียวกัน ซึ่งจะเห็นว่าได้ผลลัพธ์เดียวกัน และนี่คือข้อพิสูจน์ให้เห็นว่า สามารถที่จะใช้ VHDL จำลองรูปแบบของ FMS ในลักษณะของ behavioral description ได้โดยไม่ต้องคำนึงถึงวงจรจริง



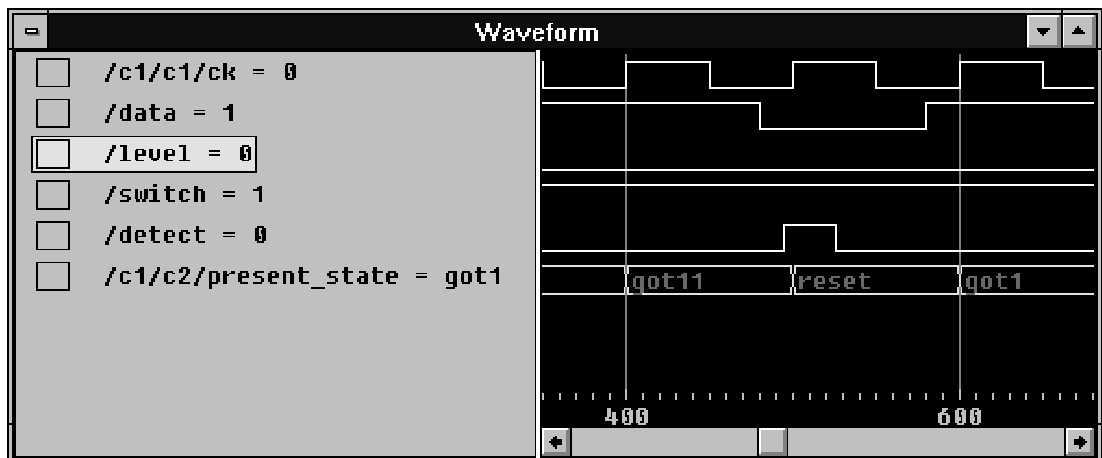
รูปที่ ก-9

จะสังเกตได้ว่าผลลัพธ์ที่ได้จากการจำลองการทำงานของรูปแบบหลัง จะเหมือนกับรูปแบบแรก (รูปที่ ก-8



รูปที่ ก-10

เพื่อที่จะแสดงให้เห็นถึงการเปลี่ยนสภาวะของระบบ FSM จึงขยายให้เห็นในบางส่วนตามที่แสดงในรูปที่ ก-11



รูปที่ ก-11

ผนวก ข.

STANDARD MSI 74LS PACKAGES

ในบทนี้จะเป็นการรวม model ของอุปกรณ์มาตรฐานตระกูล TTL 74xx ที่ใช้ในการออกแบบมาก ไว้ใน PACKAGE

ข.1 Package utility

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
--
PACKAGE utility IS
  PROCEDURE stdlogicvector_to_int (stdlogicvector_in : IN std_logic_vector;
     int_out : OUT INTEGER);
  PROCEDURE int_to_stdlogicvector (int_in : IN INTEGER;
     stdlogicvector_out : OUT std_logic_vector);
  FUNCTION inc_int (x : std_logic_vector) RETURN std_logic_vector;
END utility;
PACKAGE BODY utility IS
   PROCEDURE stdlogicvector_to_int (stdlogicvector_in : IN std_logic_vector;
     int_out : OUT INTEGER) IS
     VARIABLE result : INTEGER;
   BEGIN
      result := 0;
      FOR i IN 0 TO (stdlogicvector_in'LENGTH - 1) LOOP
         ASSERT NOT((stdlogicvector_in(i) /= '0') OR (stdlogicvector_in(i) /= '1'))
           REPORT "Input data contents not '0' and '1' values!"
           SEVERITY NOTE;
         IF stdlogicvector_in(i) = '1' THEN
            result := result + 2**i;
         END IF;
      END LOOP;
      int_out := result;
   END stdlogicvector_to_int;
  PROCEDURE int_to_stdlogicvector (int_in : IN INTEGER;
     stdlogicvector_out : OUT std_logic_vector) IS
     VARIABLE tmp : INTEGER;
   BEGIN
      tmp := int_in;
      FOR i IN 0 TO (stdlogicvector_out'LENGTH - 1) LOOP
         IF (tmp MOD 2 = 1) THEN
            stdlogicvector_out(i) := '1';
         ELSE  stdlogicvector_out(i) := '0';
         END IF;
         tmp := tmp/2;
      END LOOP;
   END  int_to_stdlogicvector;
```

```
  FUNCTION inc_int (x : std_logic_vector) RETURN std_logic_vector IS
    VARIABLE i : INTEGER;
    VARIABLE t : std_logic_vector (x'RANGE);
  BEGIN
    stdlogicvector_to_int (x, i);
    i := i + 1;
    IF i >= 2**x'LENGTH THEN i := 0;
    END IF;
    int_to_stdlogicvector (i, t);
    RETURN t;
  END inc_int;
END utility;
```

ข.2 74LS85 4-BIT MAGNITUDE COMMPARATOR

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE WORK.utility.ALL;

ENTITY ttl_74ls85_comparator IS
  GENERIC (prop_delay : TIME := 11 NS);
PORT (   a,b : IN std_logic_vector (3 DOWNTO 0);   gt, eq, lt :IN std_logic;
         a_gt_b, a_eq_b, a_lt_b : OUT std_logic     );
END ttl_74ls85_comparator;
ARCHITECTURE behavioral OF ttl_74ls85_comparator IS
BEGIN
  PROCESS (a, b, gt, eq, lt)
  VARIABLE ai, bi : INTEGER;
  BEGIN
      stdlogicvector_to_int (a, ai);
      stdlogicvector_to_int (b, bi);
    IF ai > bi THEN                          -- a greater than b
      a_gt_b    <= '1' AFTER prop_delay;
      a_eq_b    <= '0' AFTER prop_delay;
      a_lt_b    <= '0' AFTER prop_delay;
    ELSIF ai < bi THEN                         -- a least than b
      a_gt_b    <= '0' AFTER prop_delay;
      a_eq_b    <= '0' AFTER prop_delay;
      a_lt_b    <= '1' AFTER prop_delay;
    ELSIF ai = bi THEN                         -- a equal b
      a_gt_b    <= gt  AFTER prop_delay;
      a_eq_b    <= eq AFTER prop_delay;
      a_lt_b    <= lt   AFTER prop_delay;
    END IF;
  END PROCESS;
END behavioral;
```

ข.3 74LS157 QUADRUPLE 2- LINE TO 1-LINE MULTIPLEXER

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE WORK.utility.ALL;

ENTITY ttl_74ls157 IS
    GENERIC  (prop_delay : TIME := 18 NS);
    PORT (   g_bar, s : IN std_logic;
             a4, b4 : IN std_logic_vector (3 DOWNTO 0 );
             y4 : OUT std_logic_vector (3 DOWNTO 0)  );
END ttl_74ls157;
ARCHITECTURE dataflow OF ttl_74ls157 IS
BEGIN
    PROCESS (a4, b4, g_bar, s)
    BEGIN
        IF g_bar = '0' THEN
            IF s =  '0' THEN
                y4 <= a4 AFTER prop_delay;
            ELSE
                y4 <= b4 AFTER prop_delay;
            END IF;
        ELSE
        y4 <= "0000";
        END IF;
    END PROCESS;
END dataflow;
```

ข.4 74LS373 OCTAL D-TYPE TRANSPARENT LATCHES

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE WORK.utility.ALL;

ENTITY ttl_74ls373_register IS
    GENERIC (prop_delay : TIME := 15 NS);
    PORT (   enable, oc_bar : IN std_logic; d8 : IN std_logic_vector (7 DOWNTO 0);
             q8 : OUT std_logic_vector ( 7 DOWNTO 0)   );
END ttl_74ls373_register;
ARCHITECTURE dataflow OF ttl_74ls373_register IS
    SIGNAL state : std_logic_vector (7 DOWNTO 0);
BEGIN
    reg: BLOCK ( enable = '1')
    BEGIN
        state <= GUARDED d8 AFTER prop_delay;
    END BLOCK reg;
    q8 <= state WHEN oc_bar = '0' ELSE "ZZZZZZZZ";
END dataflow;
```

ข.5 74LS163 SYNCHRONOUS 4-BIT COUNTER

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE WORK.utility.ALL;

ENTITY  ttl_74ls163_counter IS
    GENERIC (prop_delay : TIME := 18 NS);
    PORT  (   clk, clr_bar, ld_bar, enp, ent : IN std_logic;
                abcd : IN std_logic_vector (3  DOWNTO 0);
                q_abcd : OUT std_logic_vector (3 DOWNTO 0); rco : OUT
std_logic  );
END ttl_74ls163_counter;
ARCHITECTURE behavioral OF ttl_74ls163_counter IS
BEGIN
    counting : PROCESS (clk)
        VARIABLE internal_count : std_logic_vector (3 DOWNTO 0) := "0000";
    BEGIN
        IF (clk = '1') THEN
            IF (clr_bar = '0') THEN
                internal_count := "0000";
            ELSIF (ld_bar = '0' ) THEN
                internal_count := abcd;
            ELSIF (enp = '1' AND ent = '1' ) THEN
                internal_count := inc_int (internal_count);
                IF (internal_count = "1111") THEN
                    rco <= '1' AFTER  prop_delay;
                ELSE
                    rco <= '0';
                END IF;
            END IF;
            q_abcd <= internal_count AFTER prop_delay;
        END IF;
    END PROCESS counting;
END behavioral;
```

ข.6 74LS541 TRANCEIVER

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE WORK.utility.ALL;

ENTITY ttl_74ls541 IS
    GENERIC (prop_delay : TIME := 10 NS);
    PORT (    g_bar : IN std_logic_vector ( 1 DOWNTO 0);
                a8 : IN std_logic_vector ( 7 DOWNTO 0);
                y8 : OUT std_logic_vector ( 7 DOWNTO 0)    );
END ttl_74ls541;
ARCHITECTURE dataflow OF ttl_74ls541 IS
BEGIN
    y8 <= a8 AFTER prop_delay WHEN g_bar = "00" ELSE "ZZZZZZZZ";
END dataflow;
```

ข.7 74LS283 4- BIT BINARY FULL ADDER

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE WORK.utility.ALL;

ENTITY ttl_74ls283 IS
  GENERIC  (prop_delay : TIME := 14 NS; prop_delay1 : TIME := 16 NS);
    PORT (   c_in : IN std_logic; c_out : OUT std_logic;
             a4,b4 : IN std_logic_vector (3 DOWNTO 0 );
             sum : OUT std_logic_vector (3 DOWNTO 0 )    );
END ttl_74ls283;
ARCHITECTURE behavioral OF ttl_74ls283 IS
BEGIN
    adder : PROCESS (a4,b4,c_in)
        VARIABLE atemp,btemp,ytemp : INTEGER := 0;
        VARIABLE stemp : std_logic_vector (3 DOWNTO 0) := "0000";
    BEGIN
        stdlogicvector_to_int (a4,atemp);
        stdlogicvector_to_int (b4,btemp);
        IF (c_in = '1') THEN
            ytemp := atemp + btemp + 1;
        ELSE
            ytemp := atemp + btemp;
        END IF;
        IF ytemp > 15 THEN
            c_out <= '1' AFTER prop_delay;
        ELSE
            c_out <= '0' AFTER prop_delay;
        END IF;
        int_to_stdlogicvector (ytemp,stemp);
        sum <= stemp AFTER prop_delay1;
    END PROCESS adder;
END behavioral;
```

ข.8 74LS377 OCTAL D-TYPE FILP-FLOPS

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE WORK.utility.ALL;

ENTITY ttl_74ls377_register IS
    GENERIC (prop_delay : TIME := 17 NS);
    PORT (   clk, g_bar : IN std_logic; d8 : IN std_logic_vector (7 DOWNTO 0);
             q8 : OUT std_logic_vector ( 7 DOWNTO 0)    );
END ttl_74ls377_register;
ARCHITECTURE dataflow OF ttl_74ls377_register IS
    SIGNAL GUARD : BOOLEAN;
BEGIN
    GUARD <= NOT clk'STABLE AND clk = '1' AND (g_bar = '0');
    q8 <= GUARDED  d8 AFTER prop_delay;
END dataflow;
```

ข.9 74LS299 UNIVERSAL SHIFT-REGISTER

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE WORK.utility.ALL;

ENTITY ttl_74ls299 IS
    GENERIC (prop_delay : TIME := 27 NS);
    PORT (   clk, clr_bar, lin, rin : IN std_logic;
             s, g_bar  : IN std_logic_vector (1 DOWNTO 0 );
             qa, qh : OUT std_logic;
             qq : INOUT std_logic_vector (7 DOWNTO 0 )    );
END ttl_74ls299;
ARCHITECTURE behavioral OF ttl_74ls299 IS
    SIGNAL iq : std_logic_vector ( 7 DOWNTO 0);
BEGIN
    clocking : PROCESS (clk, clr_bar)
    BEGIN
        IF clr_bar = '0' THEN
            iq <= "00000000";                              -- clear
            ELSIF ((clk'EVENT) AND (clk = '1')) THEN
                CASE s IS
                    WHEN "01" =>
                        iq <= rin & iq (7 DOWNTO 1 );          -- shift right
                    WHEN "10" =>
                        iq <= iq (6) & iq (5 DOWNTO 0 ) & lin;  -- shift left
                    WHEN "11" =>
                        iq <= qq;                              -- load
                    WHEN  OTHERS => NULL;                      -- hold
                END CASE;
        END IF;
    END PROCESS clocking;
    tri_state: PROCESS (iq, g_bar)
    BEGIN
        IF g_bar = "00" THEN
            IF s /= "11" THEN
                qq <= iq;
            ELSE
                qq <= "ZZZZZZZZ";
            END IF;
        ELSE
            qq <= "ZZZZZZZZ";
        END IF;
    END PROCESS tri_state;
    qa <= iq(7);
    qh <= iq(0);
END  behavioral;
```

ผนวก ค.

PACKAGE STANDARD and STD_LOGIC_1164

ในระบบพัฒนา VHDL ทางสมาคม IEEE ได้กำหนดให้มี PACKAGE ต่อไปนี้ในระบบ และถูกเก็บไว้ที่ ...\std\standard; ...\std\textio และ ...\ieee\std_logic_1164

```
--------------------------------------------------------------------
--                                                                --
--                          STANDARD                              --
--                                                                --
--    VHDL standard types, as defined by IEEE Std 1076.           --
--                                                                --
--------------------------------------------------------------------

package Standard is

--  Predefined enumeration types:

type BOOLEAN is (FALSE, TRUE);

type BIT is ('0', '1');

type CHARACTER is (
      NUL,  SOH,  STX,  ETX,  EOT,  ENQ,  ACK,  BEL,
      BS,   HT,   LF,   VT,   FF,   CR,   SO,   SI,
      DLE,  DC1,  DC2,  DC3,  DC4,  NAK,  SYN,  ETB,
      CAN,  EM,   SUB,  ESC,  FSP,  GSP,  RSP,  USP,

      ' ',  '!',  '"',  '#',  '$',  '%',  '&',  ''',
      '(',  ')',  '*',  '+',  ',',  '-',  '.',  '/',
      '0',  '1',  '2',  '3',  '4',  '5',  '6',  '7',
      '8',  '9',  ':',  ';',  '<',  '=',  '>',  '?',

      '@',  'A',  'B',  'C',  'D',  'E',  'F',  'G',
      'H',  'I',  'J',  'K',  'L',  'M',  'N',  'O',
      'P',  'Q',  'R',  'S',  'T',  'U',  'V',  'W',
      'X',  'Y',  'Z',  '[',  '\',  ']',  '^',  '_',

      '`',  'a',  'b',  'c',  'd',  'e',  'f',  'g',
      'h',  'i',  'j',  'k',  'l',  'm',  'n',  'o',
      'p',  'q',  'r',  's',  't',  'u',  'v',  'w',
      'x',  'y',  'z',  '{',  '|',  '}',  '~',  DEL);
```

```
type SEVERITY_LEVEL is (NOTE, WARNING, ERROR, FAILURE);

-- Predefined numeric types:

type INTEGER is range -2147483647 to 2147483647;

type REAL is range -1.0E38 to 1.0E38;

-- Predefined type TIME:

type TIME is range INTEGER'LOW to INTEGER'HIGH
            units
            fs;                         -- femtosecond
            ps    =  1000 fs;           -- picosecond
            ns    =  1000 ps;           -- nanosecond
            us    =  1000 ns;           -- microsecond
            ms    =  1000 us;           -- milisecond
            sec   =  1000 ms;           -- second
            min   =  60   sec;          -- minute
            hr    =  60   min;          -- hour
      end units;

--------------------------------------------------------------------
--
--             STANDARD PACKAGE TEXTIO (DECLARATION)             --
--
--------------------------------------------------------------------
package Textio is
--------------------------------------------------------------------
-- Type Definitions for Text I/O

type LINE is access STRING ;        -- a LINE is a pointer to a
STRING value

type TEXT is file of STRING ;       -- a file of variable-length
ASCII records

type SIDE is (RIGHT,LEFT) ;         -- for justifying output data
within fields

subtype WIDTH is NATURAL ;          -- for specifying widths of
output fields

-- Standard Text Files

file INPUT: TEXT is in "STD_INPUT" ;

file OUTPUT: TEXT is out "STD_OUTPUT" ;

-- Input Routines for Standard Types

procedure READLINE(variable F :in TEXT;   L:out LINE) ;

procedure READ(L:inout LINE;  VALUE:out BIT;
      GOOD:out BOOLEAN) ;
procedure READ(L:inout LINE;  VALUE:out BIT) ;

procedure READ(L:inout LINE;  VALUE:out BIT_VECTOR;
      GOOD:out BOOLEAN) ;
procedure READ(L:inout LINE;  VALUE:out BIT_VECTOR) ;
```

```
procedure READ(L:inout LINE;  VALUE:out BOOLEAN;
      GOOD:out BOOLEAN) ;
procedure READ(L:inout LINE;  VALUE:out BOOLEAN) ;

procedure READ(L:inout LINE;  VALUE:out CHARACTER;
      GOOD:out BOOLEAN) ;
procedure READ(L:inout LINE;  VALUE:out CHARACTER) ;

procedure READ(L:inout LINE;  VALUE:out INTEGER;
      GOOD:out BOOLEAN) ;
procedure READ(L:inout LINE;  VALUE:out INTEGER) ;

-- procedure READ(L:inout LINE;     VALUE:out REAL;
      GOOD:out BOOLEAN) ;
-- procedure READ(L:inout LINE;     VALUE:out REAL) ;

procedure READ(L:inout LINE;  VALUE:out STRING;
      GOOD:out BOOLEAN) ;
procedure READ(L:inout LINE;  VALUE:out STRING) ;

procedure READ(L:inout LINE;  VALUE:out TIME;
      GOOD:out BOOLEAN) ;
procedure READ(L:inout LINE;  VALUE:out TIME) ;

-- Output Routines for Standard Types

procedure WRITELINE(variable F:in TEXT;   L:inout LINE) ;

procedure STD_WRITELINE(L:inout LINE) ;

procedure WRITE(L:inout LINE; VALUE:in BIT;
            JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0) ;

procedure WRITE(L:inout LINE; VALUE:in BIT_VECTOR;
            JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0) ;

procedure WRITE(L:inout LINE; VALUE:in BOOLEAN;
            JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0) ;

procedure WRITE(L:inout LINE; VALUE:in CHARACTER;
            JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0) ;

procedure WRITE(L:inout LINE; VALUE:in INTEGER;
            JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0) ;

-- procedure WRITE(L:inout LINE;    VALUE:in REAL;
--          JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0;
--          DIGITS:in NATURAL := 0);

procedure WRITE(L:inout LINE; VALUE:in STRING;
            JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0) ;

procedure WRITE(L:inout LINE; VALUE:in TIME;
            JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0;
            UNIT:in TIME := 1 ns);
-- File Position Predicates
function ENDLINE(L:in LINE) return BOOLEAN ;
function ENDFILE(F:in TEXT) return BOOLEAN ;
end Textio ;
```

```
-------------------------------------------------------------------
--
--                STANDARD PACKAGE TEXTIO (BODY)                  --
--
-------------------------------------------------------------------
package body Textio is

-----------------------------------------------------------------

procedure READA(L:inout LINE; VALUE:out BIT;
      GOOD:out BOOLEAN) ;
procedure READB(L:inout LINE; VALUE:out BIT) ;

procedure READC(L:inout LINE; VALUE:out BIT_VECTOR;
      GOOD:out BOOLEAN) ;
procedure READD(L:inout LINE; VALUE:out BIT_VECTOR) ;

procedure READE(L:inout LINE; VALUE:out BOOLEAN;
      GOOD:out BOOLEAN) ;
procedure READF(L:inout LINE; VALUE:out BOOLEAN) ;

procedure READG(L:inout LINE; VALUE:out CHARACTER;
      GOOD:out BOOLEAN) ;
procedure READH(L:inout LINE; VALUE:out CHARACTER) ;

procedure READI(L:inout LINE; VALUE:out INTEGER;
      GOOD:out BOOLEAN) ;
procedure READJ(L:inout LINE; VALUE:out INTEGER) ;

procedure READM(L:inout LINE; VALUE:out STRING;
      GOOD:out BOOLEAN) ;
procedure READN(L:inout LINE; VALUE:out STRING) ;

procedure READO(L:inout LINE; VALUE:out TIME;
      GOOD:out BOOLEAN) ;
procedure READP(L:inout LINE; VALUE:out TIME) ;

procedure WRITEA(L:inout LINE;      VALUE:in BIT;
            JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0) ;

procedure WRITEB(L:inout LINE;      VALUE:in BIT_VECTOR;
            JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0) ;

procedure WRITEC(L:inout LINE;      VALUE:in BOOLEAN;
            JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0) ;

procedure WRITED(L:inout LINE;      VALUE:in CHARACTER;
            JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0) ;

procedure WRITEE(L:inout LINE;      VALUE:in INTEGER;
            JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0) ;

procedure WRITEG(L:inout LINE;      VALUE:in STRING;
            JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0) ;

procedure WRITEH(L:inout LINE;      VALUE:in TIME;
            JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0;
                UNIT:in TIME := 1 ns) ;
```

```
---------------------------------------------------------------------
-- Input Routines for Standard Types
---------------------------------------------------------------------
procedure READ(L:inout LINE;  VALUE:out BIT;
      GOOD:out BOOLEAN) is
Begin
      READA(L,    VALUE,      GOOD);
end READ;
procedure READ(L:inout LINE;  VALUE:out BIT)
      is
Begin
      READB(L,    VALUE) ;
end READ;

procedure READ(L:inout LINE;  VALUE:out BIT_VECTOR;
      GOOD:out BOOLEAN) is
Begin
      READC(L,    VALUE,      GOOD);
end READ;
procedure READ(L:inout LINE;  VALUE:out BIT_VECTOR)
      is
Begin
      READD(L,    VALUE);
end READ;

procedure READ(L:inout LINE;  VALUE:out BOOLEAN;
      GOOD:out BOOLEAN) is
Begin
      READE(L,    VALUE,      GOOD);
end READ;
procedure READ(L:inout LINE;  VALUE:out BOOLEAN)
      is
Begin
      READF(L,    VALUE);
end READ;

procedure READ(L:inout LINE;  VALUE:out CHARACTER;
      GOOD:out BOOLEAN) is
Begin
      READG(L,    VALUE,      GOOD);
end READ;
procedure READ(L:inout LINE;  VALUE:out CHARACTER)
      is
Begin
      READH(L,    VALUE);
end READ;

procedure READ(L:inout LINE;  VALUE:out INTEGER;
      GOOD:out BOOLEAN) is
Begin
      READI(L,    VALUE,      GOOD);
end READ;
procedure READ(L:inout LINE;  VALUE:out INTEGER)
      is
Begin
      READJ(L,    VALUE);
end READ;
-- procedure READ(L:inout LINE;      VALUE:out REAL;
      GOOD:out BOOLEAN) ;
-- procedure READ(L:inout LINE;      VALUE:out REAL) ;
```

```
procedure READ(L:inout LINE;  VALUE:out STRING;
      GOOD:out BOOLEAN) is
Begin
      READM(L,    VALUE,      GOOD);
end READ;
procedure READ(L:inout LINE;  VALUE:out STRING)
      is
Begin
      READN(L,    VALUE);
end READ;
procedure READ(L:inout LINE;  VALUE:out TIME;
      GOOD:out BOOLEAN) is
Begin
      READO(L,    VALUE,      GOOD);
end READ;
procedure READ(L:inout LINE;  VALUE:out TIME)
      is
Begin
      READP(L,    VALUE);
end READ;

procedure WRITE(L:inout LINE; VALUE:in BIT;
      JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0)
      is
Begin
      WRITEA(L,   VALUE,
            JUSTIFIED, FIELD);
end WRITE;
procedure WRITE(L:inout LINE; VALUE:in BIT_VECTOR;
      JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0)
      is
Begin
      WRITEB(L,   VALUE,
            JUSTIFIED, FIELD);
end WRITE;

procedure WRITE(L:inout LINE; VALUE:in BOOLEAN;
      JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0)
      is
Begin
      WRITEC(L,   VALUE,
            JUSTIFIED, FIELD);
end WRITE;
procedure WRITE(L:inout LINE; VALUE:in CHARACTER;
      JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0)
      is
Begin
      WRITED(L,   VALUE,
            JUSTIFIED, FIELD);
end WRITE;
procedure WRITE(L:inout LINE; VALUE:in INTEGER;
      JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0)
      is
Begin
      WRITEE(L,   VALUE,
            JUSTIFIED, FIELD);
end WRITE;
-- procedure WRITE(L:inout LINE;    VALUE:in REAL;
--          JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0;
--          DIGITS:in NATURAL := 0);
```

```
procedure WRITE(L:inout LINE; VALUE:in STRING;
      JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0)
      is
Begin
      WRITEG(L,   VALUE,
            JUSTIFIED, FIELD);
end WRITE;
procedure WRITE(L:inout LINE; VALUE:in TIME;
      JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0;
      UNIT:in TIME := 1 ns)
      is
Begin
      WRITEH(L,   VALUE,
            JUSTIFIED, FIELD,UNIT);
end WRITE;

procedure STD_WRITELINE(L:inout LINE)
      is
Begin
       WRITELINE(OUTPUT,L);
end STD_WRITELINE;

end Textio ;
--------------------------------------------------------------------
--                                                                --
--          PORTABLE - STD_LOGIC_1164  (DECLARATION)              --
--                                                                --
-- This package defines the portable constructs that were defined --
-- by IEEE VHDL Model Standards Group.                            --
--                                                                --
--------------------------------------------------------------------
--------------------------------------------------------------------
--
--
--   Title     :  std_logic_1164 multi-value logic system
--   Library   :  This package shall be compiled into a library
--             :  symbolically named IEEE.
--             :
--   Developers:  IEEE model standards group (par 1164)
--   Purpose   :  This packages defines a standard for designers
--             :  to use in describing the interconnection data
--             :  types used in vhdl modeling.
--             :
--   Limitation:  The logic system defined in this package may
--             :  be insufficient for modeling switched transistors,
--             :  since such a requirement is out of the scope of
--             :  this effort. Furthermore, mathematics, primitives,
--             :  timing standards, etc. are considered orthogonal
--             :  issues as it relates to this package and are
--             :  therefore beyond the scope of this effort.
--             :
--   Note      :  No declarations or definitions shall be included
--             :  in, excluded from this package. The "package
--             :  declaration" defines the types, subtypes and
--             :  declarations of std_logic_1164. The std_logic_1164
--             :  package body shall be considered the formal
--             :  definition of the semantics of this package. Tool
--             :  developers may choose to implement the package
--             :   body in the most efficient manner to them.
--             :
```

```
--------------------------------------------------------------------
--   modification history :
--------------------------------------------------------------------
--  version | mod. date:|
--   v4.200 | 01/02/92  |
--------------------------------------------------------------------
PACKAGE Std_logic_1164 is
--------------------------------------------------------------------
-- Logic State System (unresolved)
--------------------------------------------------------------------
    TYPE std_ulogic is ( 'U',  -- Uninitialized
                         'X',  -- Forcing Unknown
                         '0',  -- Forcing 0
                         '1',  -- Forcing 1
                         'Z',  -- High Impedance
                         'W',  -- Weak    Unknown
                         'L',  -- Weak    0
                         'H',  -- Weak    1
                         '-'   -- don't care
                       );


-----------------------------------------------------------------
-- Unconstrained array of std_ulogic for use with the resolution
-- function
-----------------------------------------------------------------
    TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) of
      std_ulogic;
-----------------------------------------------------------------
-- Resolution function
-----------------------------------------------------------------
    FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;

-----------------------------------------------------------------
-- *** Industry Standard Logic Type ***
-----------------------------------------------------------------
    SUBTYPE std_logic IS resolved std_ulogic;
-----------------------------------------------------------------
-- Unconstrained array of std_logic for use in declaring signal
-- arrays
-----------------------------------------------------------------
    TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <> ) of
      std_logic;
-----------------------------------------------------------------
-- Basic states + Test
-----------------------------------------------------------------
    SUBTYPE X01    is resolved std_ulogic range 'X' to '1';
-- ('X','0','1')
    SUBTYPE X01Z   is resolved std_ulogic range 'X' to 'Z';
-- ('X','0','1','Z')
    SUBTYPE UX01   is resolved std_ulogic range 'U' to '1';
-- ('U','X','0','1')
    SUBTYPE UX01Z  is resolved std_ulogic range 'U' to 'Z';
-- ('U','X','0','1','Z')
```

```
-------------------------------------------------------------------
-- Overloaded Logical Operators
-------------------------------------------------------------------
    FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
--  function "xnor" ( l : std_ulogic; r : std_ulogic ) return ux01;
    FUNCTION "not"  ( l : std_ulogic                 ) RETURN UX01;

-------------------------------------------------------------------
-- Vectorized Overloaded Logical Operators
-------------------------------------------------------------------
    FUNCTION "and"  ( l, r : std_logic_vector )  RETURN
      std_logic_vector;
    FUNCTION "nand" ( l, r : std_logic_vector )  RETURN
      std_logic_vector;
    FUNCTION "or"   ( l, r : std_logic_vector )  RETURN
      std_logic_vector;
    FUNCTION "nor"  ( l, r : std_logic_vector )  RETURN
      std_logic_vector;
    FUNCTION "xor"  ( l, r : std_logic_vector )  RETURN
      std_logic_vector;
    FUNCTION "not"  ( l    : std_logic_vector )  RETURN
      std_logic_vector;
    FUNCTION "and"  ( l, r : std_ulogic_vector ) RETURN
      std_ulogic_vector;
    FUNCTION "nand" ( l, r : std_ulogic_vector ) RETURN
      std_ulogic_vector;
    FUNCTION "or"   ( l, r : std_ulogic_vector ) RETURN
      std_ulogic_vector;
    FUNCTION "nor"  ( l, r : std_ulogic_vector ) RETURN
      std_ulogic_vector;
    FUNCTION "xor"  ( l, r : std_ulogic_vector ) RETURN
      std_ulogic_vector;
    FUNCTION "not"  ( l    : std_ulogic_vector ) RETURN
      std_ulogic_vector;

-------------------------------------------------------------------
--  Note : The declaration and implementation of the "xnor" function
--  is specifically commented until at which time the VHDL language
--  has been  officially adopted as containing such a function. At
--  such a point, the following comments may be removed along with
--  this notice without further "official" ballotting of this
--  std_logic_1164 package. It is the intent of this effort to
--  provide such a function once it becomes available in the VHDL
--  standard.
-------------------------------------------------------------------
--  function "xnor" ( l, r : std_logic_vector  ) return
--    std_logic_vector;
--  function "xnor" ( l, r : std_ulogic_vector ) return
--    std_ulogic_vector;
```

```
-------------------------------------------------------------------
-- Conversion Functions
-------------------------------------------------------------------
    FUNCTION To_bit       ( s : std_ulogic;        xmap : BIT :=
                  '0') RETURN BIT;
    FUNCTION To_bitvector ( s : std_logic_vector ; xmap : BIT :=
                  '0') RETURN BIT_VECTOR;
    FUNCTION To_bitvector ( s : std_ulogic_vector; xmap : BIT :=
                  '0') RETURN BIT_VECTOR;
    FUNCTION To_StdULogic       ( b : BIT               )
             RETURN std_ulogic;
    FUNCTION To_StdLogicVector  ( b : BIT_VECTOR        )
             RETURN std_logic_vector;
    FUNCTION To_StdLogicVector  ( s : std_ulogic_vector )
             RETURN std_logic_vector;
    FUNCTION To_StdULogicVector ( b : BIT_VECTOR        )
             RETURN std_ulogic_vector;
    FUNCTION To_StdULogicVector ( s : std_logic_vector  )
             RETURN std_ulogic_vector;

 -------------------------------------------------------------------
 -- strength strippers and type convertors
 -------------------------------------------------------------------

    FUNCTION To_X01  ( s : std_logic_vector ) RETURN
      std_logic_vector;
    FUNCTION To_X01  ( s : std_ulogic_vector) RETURN
      std_ulogic_vector;
    FUNCTION To_X01  ( s : std_ulogic       ) RETURN
      X01;
    FUNCTION To_X01  ( b : bit_vector       ) RETURN
      std_logic_vector;
    FUNCTION To_X01  ( b : bit_vector       ) RETURN
      std_ulogic_vector;
    FUNCTION To_X01  ( b : bit              ) RETURN
      X01;
    FUNCTION To_X01Z ( s : std_logic_vector ) RETURN
      std_logic_vector;
    FUNCTION To_X01Z ( s : std_ulogic_vector) RETURN
      std_ulogic_vector;
    FUNCTION To_X01Z ( s : std_ulogic       ) RETURN
      X01Z;
    FUNCTION To_X01Z ( b : bit_vector       ) RETURN
      std_logic_vector;
    FUNCTION To_X01Z ( b : bit_vector       ) RETURN
      std_ulogic_vector;
    FUNCTION To_X01Z ( b : bit              ) RETURN
      X01Z;
    FUNCTION To_UX01 ( s : std_logic_vector ) RETURN
      std_logic_vector;
    FUNCTION To_UX01 ( s : std_ulogic_vector) RETURN
      std_ulogic_vector;
    FUNCTION To_UX01 ( s : std_ulogic       ) RETURN
      UX01;
    FUNCTION To_UX01 ( b : bit_vector       ) RETURN
      std_logic_vector;
    FUNCTION To_UX01 ( b : bit_vector       ) RETURN
      std_ulogic_vector;
    FUNCTION To_UX01 ( b : bit              ) RETURN
      UX01;
```

```
-------------------------------------------------------------------
-- Edge Detection
-------------------------------------------------------------------
    FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN boolean;
    FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN boolean;
-------------------------------------------------------------------
-- object contains an unknown
-------------------------------------------------------------------
    FUNCTION Is_X ( s : std_ulogic_vector ) RETURN  BOOLEAN;
    FUNCTION Is_X ( s : std_logic_vector  ) RETURN  BOOLEAN;
    FUNCTION Is_X ( s : std_ulogic        ) RETURN  BOOLEAN;

END Std_logic_1164;
-- Function that returns the current time of simulation:

function NOW return TIME;

-- Predefined numberic subtypes:

subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;

subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;

-- Predefined array types:

type STRING is array (POSITIVE RANGE <>) of CHARACTER;

type BIT_VECTOR is array (NATURAL range <>) of BIT;

end Standard;


--------------------------------------------------------------------
--                                                                --
--               PORTABLE - STD_LOGIC_1164 (BODY)                 --
--                                                                --
-- This package defines the portable constructs that were defined --
-- by IEEE VHDL Model Standards Group.                            --
--                                                                --
--------------------------------------------------------------------

--------------------------------------------------------------------
--
--   Title     :  std_logic_1164 multi-value logic system
--   Library   :  This package shall be compiled into a library
--             :  symbolically named IEEE.
--             :
--   Developers:  IEEE model standards group (par 1164)
--   Purpose   :  This packages defines a standard for designers
--             :  to use in describing the interconnection data
--             :  types used in vhdl modeling.
--             :
--   Limitation:  The logic system defined in this package may
--             :  be insufficient for modeling switched transistors,
--             :  since such a requirement is out of the scope of
--             :  this effort. Furthermore, mathematics, primitives,
--             :  timing standards, etc. are considered orthogonal
--             :  issues as it relates to this package and are
--             :  therefore beyond the scope of this effort.
--             :
```

```
--   Note      :  No declarations or definitions shall be included
--             :  in, or excluded from this package. The "package
--             :  declaration" defines the types, subtypes and
--             :  declarations of std_logic_1164. The std_logic_1164
--             :  package body shall be considered the formal
--             :  definition of the semantics of this package. Tool
--             :  developers may choose to implement the package
--             :  body in the most efficient manner to them.
--             :
--------------------------------------------------------------------
--   modification history :
--------------------------------------------------------------------
--  version | mod. date:|
--   v4.200 | 01/02/92  |
--------------------------------------------------------------------
--
PACKAGE Std_logic_1164 is
--
--------------------------------------------------------------------
-- Logic State System (unresolved)
--------------------------------------------------------------------
    TYPE std_ulogic is ( 'U',  -- Uninitialized
                         'X',  -- Forcing Unknown
                         '0',  -- Forcing 0
                         '1',  -- Forcing 1
                         'Z',  -- High Impedance
                         'W',  -- Weak    Unknown
                         'L',  -- Weak    0
                         'H',  -- Weak    1
                         '-'   -- don't care
                       );


-------------------------------------------------------------------
-- Unconstrained array of std_ulogic for use with the resolution
-- function
-------------------------------------------------------------------
    TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) of
    std_ulogic;
-------------------------------------------------------------------
-- Resolution function
-------------------------------------------------------------------
    FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;
-------------------------------------------------------------------
-- *** Industry Standard Logic Type ***
-------------------------------------------------------------------
    SUBTYPE std_logic IS resolved std_ulogic;
-------------------------------------------------------------------
-- Unconstrained array of std_logic for use in declaring signal
-- arrays
-------------------------------------------------------------------
    TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <> ) of
         std_logic;
```

```
-------------------------------------------------------------------
-- Basic states + Test
-------------------------------------------------------------------
    SUBTYPE X01    is resolved std_ulogic range 'X' to '1';
-- ('X','0','1')
    SUBTYPE X01Z   is resolved std_ulogic range 'X' to 'Z';
-- ('X','0','1','Z')
    SUBTYPE UX01   is resolved std_ulogic range 'U' to '1';
-- ('U','X','0','1')
    SUBTYPE UX01Z  is resolved std_ulogic range 'U' to 'Z';
-- ('U','X','0','1','Z')

-------------------------------------------------------------------
-- Overloaded Logical Operators
-------------------------------------------------------------------
    FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
--  function "xnor" ( l : std_ulogic; r : std_ulogic ) return ux01;
    FUNCTION "not"  ( l : std_ulogic                 ) RETURN UX01;

-------------------------------------------------------------------
-- Vectorized Overloaded Logical Operators
-------------------------------------------------------------------
    FUNCTION "and"  ( l, r : std_logic_vector )       RETURN
      std_logic_vector;
    FUNCTION "nand" ( l, r : std_logic_vector )       RETURN
      std_logic_vector;
    FUNCTION "or"   ( l, r : std_logic_vector )       RETURN
      std_logic_vector;
    FUNCTION "nor"  ( l, r : std_logic_vector )       RETURN
      std_logic_vector;
    FUNCTION "xor"  ( l, r : std_logic_vector )       RETURN
      std_logic_vector;
    FUNCTION "not"  ( l    : std_logic_vector )       RETURN
      std_logic_vector;
    FUNCTION "and"  ( l, r : std_ulogic_vector )      RETURN
      std_ulogic_vector;
    FUNCTION "nand" ( l, r : std_ulogic_vector )      RETURN
      std_ulogic_vector;
    FUNCTION "or"   ( l, r : std_ulogic_vector )      RETURN
      std_ulogic_vector;
    FUNCTION "nor"  ( l, r : std_ulogic_vector )      RETURN
      std_ulogic_vector;
    FUNCTION "xor"  ( l, r : std_ulogic_vector )      RETURN
      std_ulogic_vector;
    FUNCTION "not"  ( l    : std_ulogic_vector )      RETURN
      std_ulogic_vector;
```

```
--  Note : The declaration and implementation of the "xnor" function
--  is specifically commented until at which time the VHDL language
--  has officially adopted as containing such a function. At such a
--  point, the following comments may be removed along with this
--  notice further "official" ballotting of this std_logic_1164
--  package. It the intent of this effort to provide such a function
--  once it becomes available in the VHDL standard.
-------------------------------------------------------------------
--  function "xnor" ( l, r : std_logic_vector  ) return
--  std_logic_vector;
--  function "xnor" ( l, r : std_ulogic_vector ) return
--  std_ulogic_vector;
-------------------------------------------------------------------
-- Conversion Functions
-------------------------------------------------------------------
    FUNCTION To_bit ( s : std_ulogic; xmap : BIT := '0')
            RETURN BIT;
    FUNCTION To_bitvector ( s : std_logic_vector ;
            xmap : BIT := '0') RETURN BIT_VECTOR;
    FUNCTION To_bitvector ( s : std_ulogic_vector;
            xmap : BIT := '0') RETURN BIT_VECTOR;
    FUNCTION To_StdULogic       ( b : BIT               )
            RETURN std_ulogic;
    FUNCTION To_StdLogicVector  ( b : BIT_VECTOR        )
            RETURN std_logic_vector;
    FUNCTION To_StdLogicVector  ( s : std_ulogic_vector )
            RETURN std_logic_vector;
    FUNCTION To_StdULogicVector ( b : BIT_VECTOR        )
            RETURN std_ulogic_vector;
    FUNCTION To_StdULogicVector ( s : std_logic_vector  )
            RETURN std_ulogic_vector;
------------------------------------------------------------------
-- strength strippers and type convertors
------------------------------------------------------------------
    FUNCTION To_X01  ( s : std_logic_vector ) RETURN
      std_logic_vector;
    FUNCTION To_X01  ( s : std_ulogic_vector) RETURN
      std_ulogic_vector;
    FUNCTION To_X01  ( s : std_ulogic       ) RETURN
      X01;
    FUNCTION To_X01  ( b : bit_vector       ) RETURN
      std_logic_vector;
    FUNCTION To_X01  ( b : bit_vector       ) RETURN
      std_ulogic_vector;
    FUNCTION To_X01  ( b : bit              ) RETURN
      X01;
    FUNCTION To_X01Z ( s : std_logic_vector ) RETURN
      std_logic_vector;
    FUNCTION To_X01Z ( s : std_ulogic_vector) RETURN
      std_ulogic_vector;
    FUNCTION To_X01Z ( s : std_ulogic       ) RETURN
      X01Z;
    FUNCTION To_X01Z ( b : bit_vector       ) RETURN
      std_logic_vector;
    FUNCTION To_X01Z ( b : bit_vector       ) RETURN
      std_ulogic_vector;
    FUNCTION To_X01Z ( b : bit              ) RETURN
      X01Z;
    FUNCTION To_UX01 ( s : std_logic_vector ) RETURN
      std_logic_vector;
```

```
    FUNCTION To_UX01 ( s : std_ulogic_vector) RETURN
      std_ulogic_vector;
    FUNCTION To_UX01 ( s : std_ulogic       ) RETURN
      UX01;
    FUNCTION To_UX01 ( b : bit_vector       ) RETURN
      std_logic_vector;
    FUNCTION To_UX01 ( b : bit_vector       ) RETURN
      std_ulogic_vector;
    FUNCTION To_UX01 ( b : bit              ) RETURN UX01;

-------------------------------------------------------------------
-- Edge Detection
-------------------------------------------------------------------
    FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN boolean;
    FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN boolean;
-------------------------------------------------------------------
-- object contains an unknown
-------------------------------------------------------------------
    FUNCTION Is_X ( s : std_ulogic_vector ) RETURN  BOOLEAN;
    FUNCTION Is_X ( s : std_logic_vector  ) RETURN  BOOLEAN;
    FUNCTION Is_X ( s : std_ulogic        ) RETURN  BOOLEAN;

END Std_logic_1164;
--

PACKAGE BODY Std_logic_1164 is

--------------------------------------------------------------------
-- Local Types
--------------------------------------------------------------------
    TYPE stdlogic_1d    is array (std_ulogic) of std_ulogic;
    TYPE stdlogic_table is array (std_ulogic, std_ulogic) of
      std_ulogic;

--------------------------------------------------------------------
-- Resolution Function
--------------------------------------------------------------------
    CONSTANT resolution_table : stdlogic_table := (
--------------------------------------------------------------------
--        |  U    X    0    1    Z    W    L    H    -        |   |
-----------------------------------------------------------
          ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
          ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
          ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
          ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
          ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
          ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
          ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
          ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
          ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
                                                   );
```

```
    FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic IS
      VARIABLE result : std_ulogic := 'Z';  -- weakest state default
    BEGIN
-- the test for a single driver is essential otherwise the
-- loop would return 'X' for a single driver of '-' and that
-- would conflict with the value of a single driver unresolved
-- signal.
-- aw       IF   (s'LENGTH = 1) THEN  RETURN s(s'LOW);
-- aw         ELS
-- Iterate through all inputs
          FOR i IN s'RANGE LOOP
            result := resolution_table (result, s(i));
          END LOOP;
          -- Return the resultant value
          RETURN result;
-- aw END If;
    END resolved;

-------------------------------------------------------------------
-- Tables for Logical Operations
-------------------------------------------------------------------
-- truth table for "and" function
    CONSTANT and_table : stdlogic_table := (
-------------------------------------------------------------------
--     |  U    X    0    1    Z    W    L    H    -       |   |
-------------------------------------------------------------------
       ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ), -- | U |
       ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | X |
       ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | 0 |
       ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 1 |
       ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | Z |
       ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | W |
       ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | L |
       ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | H |
       ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )  -- | - |
                                            );

-- truth table for "or" function
    CONSTANT or_table : stdlogic_table := (
----------------------------------------------------------------
--        |  U    X    0    1    Z    W    L    H    -       |   |
----------------------------------------------------------------
          ( 'U', 'U', 'U', '1', 'U', 'U', 'U', '1', 'U' ), -- | U |
          ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ), -- | X |
          ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 0 |
          ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ), -- | 1 |
          ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ), -- | Z |
          ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ), -- | W |
          ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | L |
          ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ), -- | H |
                                          );
```

```
-- truth table for "xor" function
      CONSTANT xor_table : stdlogic_table := (
--------------------------------------------------------------------
--          |  U    X    0    1    Z    W    L    H    -        |   |
--------------------------------------------------------------------
           ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
           ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
           ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 0 |
           ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' ), -- | 1 |
           ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | Z |
           ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | W |
           ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | L |
           ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' ), -- | H |
           ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
                                                );

-- truth table for not function
    CONSTANT not_table : stdlogic_1D :=
---------------------------------------------------------------------
--    |  U    X    0    1    Z    W    L    H    -  |
---------------------------------------------------------------------
      ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' );
---------------------------------------------------------------------
-- Overloaded Logical Operators ( with optimizing hints )
---------------------------------------------------------------------
    FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01
      IS
    BEGIN
      RETURN (and_table(L, R));
    END "and";

    FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01
      IS
    BEGIN
      RETURN (not_table (and_table(L, R)));
    END "nand";

    FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01
      IS
    BEGIN
      RETURN (or_table(L, R));
    END "or";

    FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01
      IS
    BEGIN
      RETURN (not_table (or_table(L, R)));
    END "nor";

    FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01
      IS
    BEGIN
      RETURN (xor_table(L, R));
    END "xor";

--  function "xnor"  ( l : std_ulogic; r : std_ulogic ) return ux01
--    is
--  begin
--      return not_table(xor_table(l, r));
--  end "xnor";
```

```
    FUNCTION "not"  ( l : std_ulogic ) RETURN UX01 IS
    BEGIN
      RETURN (not_table(L));
    END "not";

-------------------------------------------------------------------
-- Vectorized Overloaded Logical Operators (resolved vectors)
-------------------------------------------------------------------

    FUNCTION "and"  ( L,R : std_logic_vector ) RETURN
      std_logic_vector IS
      ALIAS LV : std_logic_vector ( 1 to L'length ) IS L;
      ALIAS RV : std_logic_vector ( 1 to R'length ) IS R;
      VARIABLE result : std_logic_vector ( 1 to L'length );
    begin
      if ( L'length /= R'length ) then
          assert false
          report
"Arguments of overloaded 'and' operator are not of the same length"
          severity FAILURE;
      else
          for i in  result'range loop
            result(i) := and_table (LV(i), RV(i));
          end loop;
      end if;
      return result;
    end "and";

----------------------------------------------------------------
    FUNCTION "nand"  ( L,R : std_logic_vector ) RETURN
      std_logic_vector IS
      ALIAS LV : std_logic_vector ( 1 to L'length ) IS L;
      ALIAS RV : std_logic_vector ( 1 to R'length ) IS R;
      VARIABLE result : std_logic_vector ( 1 to L'length );
    begin
      if ( L'length /= R'length ) then
          assert false
          report
"Arguments of overloaded 'nand' operator are not of the same length"
          severity FAILURE;
      else
          for i in  result'range loop
            result(i) := not_table(and_table (LV(i), RV(i)));
          end loop;
      end if;
      return result;
    end "nand";

----------------------------------------------------------------
```

```
    FUNCTION "or"  ( L,R : std_logic_vector ) RETURN
      std_logic_vector IS
      ALIAS LV : std_logic_vector ( 1 to L'length ) IS L;
      ALIAS RV : std_logic_vector ( 1 to R'length ) IS R;
      VARIABLE result : std_logic_vector ( 1 to L'length );
    begin
      if ( L'length /= R'length ) then
          assert false
          report "Arguments of overloaded 'or' operator are not of
the same length"
          severity FAILURE;
      else
          for i in  result'range loop
            result(i) := or_table (LV(i), RV(i));
          end loop;
      end if;
      return result;
    end "or";

-------------------------------------------------------------------
    FUNCTION "nor"  ( L,R : std_logic_vector ) RETURN
      std_logic_vector IS
      ALIAS LV : std_logic_vector ( 1 to L'length ) IS L;
      ALIAS RV : std_logic_vector ( 1 to R'length ) IS R;
      VARIABLE result : std_logic_vector ( 1 to L'length );
    begin
      if ( L'length /= R'length ) then
          assert false
          report
"Arguments of overloaded 'nor' operator are not of the same length"
          severity FAILURE;
      else
          for i in  result'range loop
            result(i) := not_table(or_table (LV(i), RV(i)));
          end loop;
      end if;
      return result;
    end "nor";

-------------------------------------------------------------------
    FUNCTION "xor"  ( L,R : std_logic_vector ) RETURN
      std_logic_vector IS
      ALIAS LV : std_logic_vector ( 1 to L'length ) IS L;
      ALIAS RV : std_logic_vector ( 1 to R'length ) IS R;
      VARIABLE result : std_logic_vector ( 1 to L'length );
    begin
      if ( L'length /= R'length ) then
          assert false
          report
"Arguments of overloaded 'xor' operator are not of the same length"
          severity FAILURE;
      else
          for i in  result'range loop
            result(i) := xor_table (LV(i), RV(i));
          end loop;
      end if;
      return result;
    end "xor";

-------------------------------------------------------------------
```

```
    FUNCTION "not"  ( L : std_logic_vector ) RETURN std_logic_vector
      IS
      ALIAS LV : std_logic_vector ( 1 to L'length ) IS L;
      VARIABLE result : std_logic_vector ( 1 to L'length ) :=
      (Others => 'X');
    begin
      for i in  result'range loop
          result(i) := not_table(LV(i));
      end loop;
      return result;
    end "not";

-------------------------------------------------------------------
-- Vectorized Overloaded Logical Operators (unresolved  vectors)
-------------------------------------------------------------------

    FUNCTION "and"  ( L,R : std_ulogic_vector ) RETURN
      std_ulogic_vector IS
      ALIAS LV : std_ulogic_vector ( 1 to L'length ) IS L;
      ALIAS RV : std_ulogic_vector ( 1 to R'length ) IS R;
      VARIABLE result : std_ulogic_vector ( 1 to L'length );
    begin
      if ( L'length /= R'length ) then
          assert false
          report
"Arguments of overloaded 'and' operator are not of the same length"
          severity FAILURE;
      else
          for i in  result'range loop
            result(i) := and_table (LV(i), RV(i));
          end loop;
      end if;
      return result;
    end "and";

-------------------------------------------------------------------
    FUNCTION "nand"  ( L,R : std_ulogic_vector ) RETURN
      std_ulogic_vector IS
      ALIAS LV : std_ulogic_vector ( 1 to L'length ) IS L;
      ALIAS RV : std_ulogic_vector ( 1 to R'length ) IS R;
      VARIABLE result : std_ulogic_vector ( 1 to L'length );
    begin
      if ( L'length /= R'length ) then
          assert false
          report
"Arguments of overloaded 'nand' operator are not of the same length"
          severity FAILURE;
      else
          for i in  result'range loop
            result(i) := not_table(and_table (LV(i), RV(i)));
          end loop;
      end if;
      return result;
    end "nand";

-------------------------------------------------------------------
```

```
    FUNCTION "or"  ( L,R : std_ulogic_vector ) RETURN
      std_ulogic_vector IS
      ALIAS LV : std_ulogic_vector ( 1 to L'length ) IS L;
      ALIAS RV : std_ulogic_vector ( 1 to R'length ) IS R;
      VARIABLE result : std_ulogic_vector ( 1 to L'length );
    begin
      if ( L'length /= R'length ) then
          assert false
          report
"Arguments of overloaded 'or' operator are not of the same length"
          severity FAILURE;
      else
          for i in  result'range loop
            result(i) := or_table (LV(i), RV(i));
          end loop;
      end if;
      return result;
    end "or";

-----------------------------------------------------------------
    FUNCTION "nor"  ( L,R : std_ulogic_vector ) RETURN
      std_ulogic_vector IS
      ALIAS LV : std_ulogic_vector ( 1 to L'length ) IS L;
      ALIAS RV : std_ulogic_vector ( 1 to R'length ) IS R;
      VARIABLE result : std_ulogic_vector ( 1 to L'length );
    begin
      if ( L'length /= R'length ) then
          assert false
          report
"Arguments of overloaded 'nor' operator are not of the same length"
          severity FAILURE;
      else
          for i in  result'range loop
            result(i) := not_table(or_table (LV(i), RV(i)));
          end loop;
      end if;
      return result;
    end "nor";
-----------------------------------------------------------------
    FUNCTION "xor"  ( L,R : std_ulogic_vector ) RETURN
      std_ulogic_vector IS
      ALIAS LV : std_ulogic_vector ( 1 to L'length ) IS L;
      ALIAS RV : std_ulogic_vector ( 1 to R'length ) IS R;
      VARIABLE result : std_ulogic_vector ( 1 to L'length );
    begin
      if ( L'length /= R'length ) then
          assert false
          report
"Arguments of overloaded 'xor' operator are not of the same length"
          severity FAILURE;
      else
          for i in  result'range loop
            result(i) := xor_table (LV(i), RV(i));
          end loop;
      end if;
      return result;
    end "xor";
-----------------------------------------------------------------
```

```
    FUNCTION "not"  ( L : std_ulogic_vector ) RETURN
      std_ulogic_vector IS
      ALIAS LV : std_ulogic_vector ( 1 to L'length ) IS L;
      VARIABLE result : std_ulogic_vector ( 1 to L'length ) :=
      (Others => 'X');
    begin
      for i in  result'range loop
          result(i) := not_table(LV(i));
      end loop;
      return result;
    end "not";
-------------------------------------------------------------------
-- Conversion Tables
-------------------------------------------------------------------
    TYPE logic_x01_table  is array (std_ulogic'low to
      std_ulogic'high) of X01;
    TYPE logic_x01z_table is array (std_ulogic'low to
      std_ulogic'high) of X01Z;
    TYPE logic_ux01_table is array (std_ulogic'low to
      std_ulogic'high) of UX01;

-------------------------------------------------------------------
-- table name : cvt_to_x01
--
-- parameters :
--        in  :  std_ulogic  -- some logic value
-- returns    :  x01         -- state value of logic value
-- purpose    :  to convert state-strength to state only
--
-- example    : if (cvt_to_x01 (input_signal) = '1' ) then ...
--
-------------------------------------------------------------------
    CONSTANT cvt_to_X01 : logic_x01_table := (
                      'X',  -- 'U'
                      'X',  -- 'X'
                      '0',  -- '0'
                      '1',  -- '1'
                      'X',  -- 'Z'
                      'X',  -- 'W'
                      '0',  -- 'L'
                      '1',  -- 'H'
                      'X'   -- '-'
                                        );

-------------------------------------------------------------------
-- table name : cvt_to_x01z
--
-- parameters :
--        in  :  std_ulogic  -- some logic value
-- returns    :  x01z        -- state value of logic value
-- purpose    :  to convert state-strength to state only
--
-- example    : if (cvt_to_x01z (input_signal) = '1' ) then ...
--
-------------------------------------------------------------------
```

```
    CONSTANT cvt_to_x01z : logic_x01z_table := (
                         'X',  -- 'U'
                         'X',  -- 'X'
                         '0',  -- '0'
                         '1',  -- '1'
                         'Z',  -- 'Z'
                         'X',  -- 'W'
                         '0',  -- 'L'
                         '1',  -- 'H'
                         'X'   -- '-'
                                                );
-----------------------------------------------------------------
-- table name : cvt_to_ux01
--
-- parameters :
--        in   :  std_ulogic  -- some logic value
-- returns     :  ux01        -- state value of logic value
-- purpose     :  to convert state-strength to state only
--
-- example     : if (cvt_to_ux01 (input_signal) = '1' ) then ...
--
-----------------------------------------------------------------
    CONSTANT cvt_to_ux01 : logic_ux01_table := (
                         'U',  -- 'U'
                         'X',  -- 'X'
                         '0',  -- '0'
                         '1',  -- '1'
                         'X',  -- 'Z'
                         'X',  -- 'W'
                         '0',  -- 'L'
                         '1',  -- 'H'
                         'X'   -- '-'
                                                );
--------------------------------------------------------------------
-- Conversion Functions
--------------------------------------------------------------------
    FUNCTION To_bit ( s : std_ulogic;        xmap : BIT := '0')
             RETURN BIT IS
    BEGIN
            CASE s IS
                WHEN '0' | 'L' => RETURN ('0');
                WHEN '1' | 'H' => RETURN ('1');
                WHEN OTHERS => RETURN xmap;
            END CASE;
    END;
--------------------------------------------------------------------
    FUNCTION To_bitvector ( s : std_logic_vector ;
      xmap : BIT := '0') RETURN BIT_VECTOR IS
        ALIAS sv : std_logic_vector ( s'LENGTH-1 DOWNTO 0 ) IS s;
        VARIABLE result : BIT_VECTOR ( s'LENGTH-1 DOWNTO 0 );
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE sv(i) IS
                WHEN '0' | 'L' => result(i) := '0';
                WHEN '1' | 'H' => result(i) := '1';
                WHEN OTHERS => result(i) := xmap;
            END CASE;
        END LOOP;
        RETURN result;
    END;
```

```
--------------------------------------------------------------------
    FUNCTION To_bitvector ( s : std_ulogic_vector;
      xmap : BIT := '0') RETURN BIT_VECTOR IS
        ALIAS sv : std_ulogic_vector ( s'LENGTH-1 DOWNTO 0 ) IS s;
        VARIABLE result : BIT_VECTOR ( s'LENGTH-1 DOWNTO 0 );
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE sv(i) IS
                WHEN '0' | 'L' => result(i) := '0';
                WHEN '1' | 'H' => result(i) := '1';
                WHEN OTHERS => result(i) := xmap;
            END CASE;
        END LOOP;
        RETURN result;
    END;
--------------------------------------------------------------------
    FUNCTION To_StdULogic       ( b : BIT               )
             RETURN std_ulogic IS
    BEGIN
        CASE b IS
            WHEN '0' => RETURN '0';
            WHEN '1' => RETURN '1';
        END CASE;
    END;
--------------------------------------------------------------------
    FUNCTION To_StdLogicVector  ( b : BIT_VECTOR        )
             RETURN std_logic_vector IS
        ALIAS bv : BIT_VECTOR ( b'LENGTH-1 DOWNTO 0 ) IS b;
        VARIABLE result : std_logic_vector ( b'LENGTH-1 DOWNTO 0 );
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE bv(i) IS
                WHEN '0' => result(i) := '0';
                WHEN '1' => result(i) := '1';
            END CASE;
        END LOOP;
        RETURN result;
    END;
--------------------------------------------------------------------
    FUNCTION To_StdLogicVector  ( s : std_ulogic_vector )
             RETURN std_logic_vector IS
        ALIAS sv : std_ulogic_vector ( s'LENGTH-1 DOWNTO 0 ) IS s;
        VARIABLE result : std_logic_vector ( s'LENGTH-1 DOWNTO 0 );
    BEGIN
        FOR i IN result'RANGE LOOP
            result(i) := sv(i);
        END LOOP;
        RETURN result;
    END;
--------------------------------------------------------------------
```

```
    FUNCTION To_StdULogicVector ( b : BIT_VECTOR        )
         RETURN std_ulogic_vector IS
        ALIAS bv : BIT_VECTOR ( b'LENGTH-1 DOWNTO 0 ) IS b;
        VARIABLE result : std_ulogic_vector ( b'LENGTH-1 DOWNTO 0 );
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE bv(i) IS
                WHEN '0' => result(i) := '0';
                WHEN '1' => result(i) := '1';
            END CASE;
        END LOOP;
        RETURN result;
    END;
--------------------------------------------------------------------
    FUNCTION To_StdULogicVector ( s : std_logic_vector )
             RETURN std_ulogic_vector IS
        ALIAS sv : std_logic_vector ( s'LENGTH-1 DOWNTO 0 ) IS s;
        VARIABLE result : std_ulogic_vector ( s'LENGTH-1 DOWNTO 0 );
    BEGIN
        FOR i IN result'RANGE LOOP
            result(i) := sv(i);
        END LOOP;
        RETURN result;
    END;

-------------------------------------------------------------------
-- strength strippers and type convertors
-------------------------------------------------------------------
-- to_x01
-------------------------------------------------------------------
    FUNCTION To_X01  ( s : std_logic_vector ) RETURN
      std_logic_vector IS
      ALIAS SV : std_logic_vector ( 1 to s'length ) IS s;
      VARIABLE result : std_logic_vector ( 1 to s'length );
    BEGIN
      for i in result'range loop
          result(i) := cvt_to_x01 (SV(i));
      end loop;
      return result;
    END;

    FUNCTION To_X01  ( s : std_ulogic_vector ) RETURN
      std_ulogic_vector IS
      ALIAS SV : std_ulogic_vector ( 1 to s'length ) IS s;
      VARIABLE result : std_ulogic_vector ( 1 to s'length );
    BEGIN
      for i in result'range loop
          result(i) := cvt_to_x01 (SV(i));
      end loop;
      return result;
    END;

    FUNCTION To_X01  ( s : std_ulogic ) RETURN X01 IS
    BEGIN
      return (cvt_to_x01(s));
    END;
```

```
   FUNCTION To_X01  ( b : bit_vector ) RETURN std_logic_vector IS
     ALIAS BV : bit_vector ( 1 to b'length ) IS b;
     VARIABLE result : std_logic_vector ( 1 to b'length );
    BEGIN
     for i in result'range loop
         case BV(i) is
           when '0' => result(i) := '0';
           when '1' => result(i) := '1';
         end case;
     end loop;
     return result;
    END;

    FUNCTION To_X01  ( b : bit_vector ) RETURN std_ulogic_vector IS
     ALIAS BV : bit_vector ( 1 to b'length ) IS b;
     VARIABLE result : std_ulogic_vector ( 1 to b'length );
    BEGIN
     for i in result'range loop
         case BV(i) is
           when '0' => result(i) := '0';
           when '1' => result(i) := '1';
         end case;
     end loop;
     return result;
    END;

    FUNCTION To_X01  ( b : bit ) RETURN X01 IS
    BEGIN
     case b is
         when '0' => return ('0');
         when '1' => return ('1');
     end case;
    END;

-------------------------------------------------------------------
-- to_x01z
-------------------------------------------------------------------

    FUNCTION To_X01Z  ( s : std_logic_vector ) RETURN
     std_logic_vector IS
     ALIAS SV : std_logic_vector ( 1 to s'length ) IS s;
     VARIABLE result : std_logic_vector ( 1 to s'length );
    BEGIN
     for i in result'range loop
         result(i) := cvt_to_x01z (SV(i));
     end loop;
     return result;
    END;

    FUNCTION To_X01Z  ( s : std_ulogic_vector ) RETURN
     std_ulogic_vector IS
     ALIAS SV : std_ulogic_vector ( 1 to s'length ) IS s;
     VARIABLE result : std_ulogic_vector ( 1 to s'length );
    BEGIN
     for i in result'range loop
         result(i) := cvt_to_x01z (SV(i));
     end loop;
     return result;
    END;
```

```
    FUNCTION To_X01Z  ( s : std_ulogic ) RETURN X01Z IS
    BEGIN
      return (cvt_to_x01z(s));
    END;

    FUNCTION To_X01Z  ( b : bit_vector ) RETURN std_logic_vector IS
      ALIAS BV : bit_vector ( 1 to b'length ) IS b;
      VARIABLE result : std_logic_vector ( 1 to b'length );
    BEGIN
      for i in result'range loop
          case BV(i) is
            when '0' => result(i) := '0';
            when '1' => result(i) := '1';
          end case;
      end loop;
      return result;
    END;

    FUNCTION To_X01Z  ( b : bit_vector ) RETURN std_ulogic_vector IS
      ALIAS BV : bit_vector ( 1 to b'length ) IS b;
      VARIABLE result : std_ulogic_vector ( 1 to b'length );
    BEGIN
      for i in result'range loop
          case BV(i) is
            when '0' => result(i) := '0';
            when '1' => result(i) := '1';
          end case;
      end loop;
      return result;
    END;

    FUNCTION To_X01Z  ( b : bit ) RETURN X01Z IS
    BEGIN
      case b is
          when '0' => return ('0');
          when '1' => return ('1');
      end case;
    END;

--------------------------------------------------------------------
-- to_ux01
--------------------------------------------------------------------

    FUNCTION To_UX01  ( s : std_logic_vector ) RETURN
      std_logic_vector IS
      ALIAS SV : std_logic_vector ( 1 to s'length ) IS s;
      VARIABLE result : std_logic_vector ( 1 to s'length );
    BEGIN
      for i in result'range loop
          result(i) := cvt_to_ux01 (SV(i));
      end loop;
      return result;
    END;
```

```
FUNCTION To_UX01  ( s : std_ulogic_vector ) RETURN
  std_ulogic_vector IS
  ALIAS SV : std_ulogic_vector ( 1 to s'length ) IS s;
  VARIABLE result : std_ulogic_vector ( 1 to s'length );
BEGIN
  for i in result'range loop
      result(i) := cvt_to_ux01 (SV(i));
  end loop;
  return result;
END;

FUNCTION To_UX01  ( s : std_ulogic ) RETURN UX01 IS
BEGIN
  return (cvt_to_ux01(s));
END;

FUNCTION To_UX01  ( b : BIT_VECTOR ) RETURN  std_logic_vector IS
    ALIAS bv : BIT_VECTOR ( 1 TO b'LENGTH ) IS b;
    VARIABLE result : std_logic_vector ( 1 TO b'LENGTH );
BEGIN
    FOR i IN result'RANGE LOOP
        CASE bv(i) IS
            WHEN '0' => result(i) := '0';
            WHEN '1' => result(i) := '1';
        END CASE;
    END LOOP;
    RETURN result;
END;

FUNCTION To_UX01  ( b : BIT_VECTOR ) RETURN  std_ulogic_vector
IS
    ALIAS bv : BIT_VECTOR ( 1 TO b'LENGTH ) IS b;
    VARIABLE result : std_ulogic_vector ( 1 TO b'LENGTH );
BEGIN
    FOR i IN result'RANGE LOOP
        CASE bv(i) IS
            WHEN '0' => result(i) := '0';
            WHEN '1' => result(i) := '1';
        END CASE;
    END LOOP;
    RETURN result;
END;

FUNCTION To_UX01  ( b : BIT ) RETURN  UX01 IS
BEGIN
        CASE b IS
            WHEN '0' => RETURN('0');
            WHEN '1' => RETURN('1');
        END CASE;
END;
```

```
--------------------------------------------------------------------
-- Edge Detection
--------------------------------------------------------------------
    Function rising_edge  (SIGNAL s : std_ulogic) RETURN boolean is
    begin
      return  (s'event and (To_X01(s) = '1') and
                       (To_X01(s'last_value) = '0'));
    end;

    Function falling_edge (SIGNAL s : std_ulogic) RETURN boolean is
    begin
      return  (s'event and (To_X01(s) = '0') and
                       (To_X01(s'last_value) = '1'));
    end;

--------------------------------------------------------------------
-- object contains an unknown
--------------------------------------------------------------------
    FUNCTION Is_X ( s : std_ulogic_vector ) RETURN  BOOLEAN IS
    BEGIN
        FOR i IN s'RANGE LOOP
            CASE s(i) IS
                WHEN 'U' | 'X' | 'Z' | 'W' | '-' => RETURN TRUE;
                WHEN OTHERS => NULL;
            END CASE;
        END LOOP;
        RETURN FALSE;
    END;

    FUNCTION Is_X ( s : std_logic_vector  ) RETURN  BOOLEAN IS
    BEGIN
        FOR i IN s'RANGE LOOP
            CASE s(i) IS
                WHEN 'U' | 'X' | 'Z' | 'W' | '-' => RETURN TRUE;
                WHEN OTHERS => NULL;
            END CASE;
        END LOOP;
        RETURN FALSE;
    END;

    FUNCTION Is_X ( s : std_ulogic        ) RETURN  BOOLEAN IS
    BEGIN
        CASE s IS
            WHEN 'U' | 'X' | 'Z' | 'W' | '-' => RETURN TRUE;
            WHEN OTHERS => NULL;
        END CASE;
        RETURN FALSE;
    END;

END Std_logic_1164;
```

ผนวก ง.

คำสงวนในภาษา VHDL

การตั้งชื่อ (identifier) ให้กับ object ในภาษา VHDL นั้น ตามมาตรฐาน IEEE 1076-1987 มีข้อจำกัดในการใช้คำสงวน (reserved word) ที่ไม่สามารถนำไปเป็นชื่อของ object ได้ในทุกลักษณะของตัวหนังสือ (case insensitive) คำเหล่านี้ได้แก่

ABS	DISCONNECT	LABEL	PACKAGE	UNITS
ACCESS	DOWNTO	LIBRARY	PORT	UNTIL
AFTER		LINKAGE	PROCEDURE	USE
ALIAS	ELSE	LOOP	PROCESS	
ALL	ELSIF			VARIABLE
AND	END	MAP	RANGE	
ARCHITECTURE	ENTITY	MOD	RECORD	WAIT
ARRAY	EXIT		REGISTER	WHEN
ASSERT		NAND	REM	WHILE
ATTRIBUTE	FILE	NEW	REPORT	WITH
	FOR	NEXT	RETURN	
BEGIN	FUNCTION	NOR		XOR
BLOCK		NOT	SELECT	
BODY	GENERATE	NULL	SEVERITY	
BUFFER	GENERIC		SIGNAL	
BUS	GUARDED	OF	SUBTYPE	
		ON		
CASE	IF	OPEN	THEN	
COMPONENT	IN	OR	TO	
CONFUGURATION	INOUT	OTHERS	TRANSPORT	
CONSTANT	IS	OUT	TYPE	

ผนวก จ.
ข้อแตกต่างบางอย่างระหว่าง
IEEE 1076-1987 กับ IEEE 1076-1993

เนื่องจากระบบสังเคราะห์วงจรจากการบรรยายในรูปของพฤติกรรม (behavioral synthesis) ส่วนใหญ่ทำงานอยู่บนพื้นฐานของ IEEE 1076-1987 ฉนั้นในหนังสือเล่มนี้จึงใช้มาตรฐานนี้เป็นหลัก แต่อย่างไรก็ตามที่มาตรฐานใหม่ (1993) ได้เพิ่มคุณสมบัติบางประการซึ่งในอนาคตระบบสังเคราะห์วงจรคงจะใช้เป็นมาตรฐาน จึงขอเสนอข้อแตกต่างในบางหัวข้อไว้ในผนวกนี้ และเพื่อความสะดวกต่อการเขียนจะใช้คำว่า VHDL'87 สำหรับ IEEE 1076-1987 และ VHDL'93 แทน IEEE 1076-1993

1) Access to Driving Values

VHDL' 87:
- Concurrent statements do not have access to their driving values
- Process can use variables to remember driving values
- Concurrent signal assignments have no convenient way

VHDL' 93:
- 'Driving_Value returns current driving value of prefix signal
- 'Driving returns Boolean indicating whether 'Driving_Value will succeed
- No access to other concurrent statements' driving values

2) Deferred Interface Object Mapping

VHDL' 87:
- Generics could be bound with component instance
- Or with configuration binding indication
- Not with both
- Instance's unbound ports cannot be bound during configuration

VHDL' 93:
- Generic can bound with component instance
- Or with configuration binding indication
- Configuration binding indication may override instance's value
- Instance's unbound port may be bound during configuration

3) Direct Instantiation

VHDL' 87:

- Two-step binding
  - Must instantiate component declaration
  - Then bind design entity to instance
- Useful, especially for top-down design, design partition, and reconfiguration
- Can be cumbersome

VHDL' 93:

- Two-step approach may still be used
- Or may directly instantiate
  - Design entity
  - Configuration declaration
- No reconfiguration then possible

4) Extended Character Set

VHDL' 87:

- ISO 646-1983 (7-bit) character set
- Inadequate outside of U.S.

VHDL' 93:

- ISO 8859-1 (8-bit) character set
- Adequate for users employing roman alphabets

5) Extended Identifiers

VHDL' 87:

- Identifiers must contain only letters, digits and underscores
- Begin with a letter
- Not end and underscore or contain consecutive underscores
- Identifiers differing only in the case of their letter are equivalent
- Cannot be used if reserved by VHDL

VHDL' 93:

- Basic identifiers unchanged
- Extended identifiers can contain any printing characters, in any order
- Surrounded by backslashes (\), embedded backslashes doubled
- Case sensitive
- Not equivalent to any basic identifier or reserved word

6) Foreign Language Interface

VHDL' 87:

- Subprogram bodies did not have to be implemented in VHDL
- Only a note in the LRM
- What about architecture?

VHDL' 93:

- Std.Standard.FOREIGN
- Can decorate any architecture or subprogram name
- Architecture or subprogram is not elaborated
- Attribute value tells implementation what to do
- Interface object types, modes, etc. of foreign bodies may be restricted by implementation
- Not portable!

7) Generalized Aliasing

VHDL' 87:

- Only objects may have aliases

VHDL' 93:

- Anything with a name (except labels and loop and generate indices) may be aliased
- Subprogram aliases may take a signature
- Aliasing an enumaration type creates implicit aliases for all enumaration values and implicitly defined operators for the type
- Can be used to build packages out of other packages:

```
LIBRARY IEEE, project;
PACKAGE my_pack IS
 ALIAS my_bit IS IEEE.std_logic_1164.std_logic;
 ALIAS "+" [MVL, MVL RETURN MVL] IS
     project.my_arith."+"[MVL, MVL RETURN MVL];
END PACKAGE my_pack;
```

8) Groups

VHDL' 87:

- No way to express, annotate relationships
- E.g., regions of code, between ports

VHDL' 93:

- Groups: named relationships between names
- Groups may be attributed
- Every statement may be labelled; regions of code expressible as groups
- A group may relate two ports; pin-to-pin timing may be expressed

```
GROUP pin2pin IS (SIGNAL, SIGNAL);              -- a group template declaration
GROUP clk2q: pin2pin (clk, q);                  -- a group declaration

ATTRIBUTE timing; DELAY_LENGTH;
ATTRIBUTE timing of clk2q: GROUP IS 12 NS;      -- an attribute specification

q <= GUARDED d after clk2q'timing;              -- use of attribute
```

9) Hierarchical Pathnames

VHDL' 87:

- No standard way of expressing hierarchical paths
- E.g., assertions, error messages, tool navigation

VHDL' 93:

- 'SIMPLE_NAME: a string representation of the name of the prefix
- 'PATH_NAME: a string describing the hierarchical path from the root of the design hierarchy to the prefix, including the design entity names
- Values may not be unique
- If the prefix is an alias, for these attributes only, the attribute applies to the alias

10) Impure Functions

VHDL' 87:

- Functions are pure
- Same arguments, same return value
- Allow optimization across function calls
- Consequently, no access to gobal signals or variables
- Unfortunately, Std.Standard.NOW not pure

VHDL' 93:

- Introduces impure functions
- Can access gobal signals and variables
- Std.Standard.NOW now impure
- VHDL' 87 functions are pure
- Keyword pure may also be used for emphasis

11) "No Change" Assignment

VHDL' 87:

- Conditional signal assignments require terminal, unconditional waveform
- Selected signal assignment require a waveform in every selected waveform

→ Every time a concurrent signal assignment executes, it must assign to its target

- Can assign the target's effective value
- Effective value ≠ Driving value!

VHDL' 93:

- Conditional signal assignments no longer require unconditional waveform
- All concurrent signal assignments may assign unaffected

→ Concurrent assignment needn't assign to target

12) Port Driven With Expressions

VHDL' 87:

- Only actual signals may be associated with ports
- Constant driving values must be assigned to declared signal, then declared signal associated with ports

VHDL' 93:

- Expressions may be associated with ports of mode IN
- Supplies constant driving value port
- No events ever occur on port

13) Postponed Processes

VHDL' 87:

- Processes and concurrent statements access signals' present values
- Cannot determine if the value is stable at the current simulated time

VHDL' 93:

- Processes and concurrent statements may be POSTPONED
- Postponed processes execute just before time changes
- Stable values of signal are accessed
- May not cause delta cycles
- Non-postponed processes act as before

14) Pulse Rejection

VHDL' 87:

- Inertial delay ≡ propagation delay
  s <= '1' AFTER 12 NS;
- Two signals must be used to model inertial delay < propagation delay
- temp <= TRANSPORT '1' AFTER 7 NS;
  s <= temp AFTER 5 NS;

VHDL' 93:

- 0 ≤ inertial delay ≤ propagation delay
  s <= REJECT 5 NS INERTIAL '1' AFTER 12 NS;
  - Propagation delay = 12 NS
  - Inertial delay = 5 NS

15) Regularized Syntax

VHDL' 87:

- Three rules for bracketing keywords:
  - Design units, subprograms: END [simple_name];
  - All other statements: END keyword [simple_name];
  - Records: END RECORD;
  - ENTITY identifier IS ...
  - COMPONENT identifier -- no IS permitted!

VHDL' 93:

- One rule may be used for all: END keyword [simple_name];
- COMPONENT identifier [IS]
- Entirely upward compatible!

16) Report Statement

VHDL' 87:

- When assertion violation triggers actions, IF statement and "ASSERT FALSE ...;" must be used
- "ASSERT FALSE ...;" may also be used to generate messages
  - Dose not provide good documentation

VHDL' 93:

- Adds report statement:
  - Has report clause
  - Has severity clause (defaults to "NOTE")

17) Revamped File I/O

VHDL' 87:

- File objects are special variables
- Can be read from, or written to, but not both
- Opened when file declaration is elaborated
- Closed when file declaration is "de-elaborated"
- External formats incompletely specified
- Files are pipes

VHDL' 93:

- Files are a fourth class of object
- Can be opened and closed as needed
- Can be read from, written to, or appended to
- External formats (more) completely specified
- Not upward-compatible!

18) Shared Variables

VHDL' 87:

- Signal only communication path between process
- Models (largely) deterministic
- Very high-level, stochastic models difficult to write

VHDL' 93:

- Multiple processes may access SHARED variables
- Inherently non-deterministic!
- No language-defined synchronization mechanism
  - Models may exhibit non-algorithmic behavior on certain implementations
- Working group established to add synchronization
  - Currently favored approach is the monitor

19) Shift and Rotate Operators

VHDL' 87:

- Shifting and rotating via slicing and concatenation only
  s := s (30 DOWNTO 0) & '0'; -- shift left logical
  s := s (0) & s (31 DOWNTO 1); -- rotate right
- Difficult for tools to understand

VHDL' 93:

- **SLL, SRL, SLA, SRA, ROL** and **ROR**
- Predefined for 1-D arrays of BIT or BOOLEAN
- May be overloaded for all other types
- Predefined arithmetic shifts assume MSB is sign bit!

20) XNOR Operator

VHDL' 87:

- Logical operators: **AND, OR, NAND, NOR, XOR** and **NOT**
- No XNOR
- can use "="
- May have to overload
- May want different 'X' handling

VHDL' 93:

- Introduces **XNOR** operator
- Same precedence as dyadic logical operators
- Associative
- Overloadable