บทที่ 4

หลักการทวนสอบวงจรดิจิตอล

การออกแบบระบบที่ซับซ้อนทุกๆระบบไม่ว่าจะด้วยฮาร์ดแวร์, ซอฟแวร์ หรือแบบผสมสามารถทำได้ยาก แต่การออกแบบที่จะให้ระบบทำงานได้อย่างถูกต้องมีความยากมากกว่า งานทางด้านการออกแบบวงจรรวมขนาดใหญ่ VLSI Design (Very-Large-Scale Integration) แบ่งออกเป็น 2 ส่วนคือ กระบวนการออกแบบ และการตรวจสอบว่าระบบมีการทำงานอย่างถูกต้อง ในกระบวนการออกแบบและวิเคราะห์ระบบจะถูกปฏิบัติเป็น 2 เฟสที่แยกออกจากกัน หลังจากวงจรถูกออกแบบแล้วจะมีการตรวจสอบก่อนนำไปผ่านกระบวนการเจือสาร (Fabrication) โดยการจำลองการทำงานเพื่อตรวจดูพฤติกรรมของวงจร และแก้ไขข้อผิดพลาดที่เกิดขึ้น หลังจากผ่านกระบวนการเจือสารแล้ววงจรจะถูกทดสอบด้วยการใช้ตัวอย่างของข้อมูลเข้าตรวจสอบเพื่อดูว่าผ่านกระบวนการเจือสารได้อย่างสมบูรณ์ ข้อผิดพลาดต่างๆในวงจรควรที่จะถูกตรวจพบก่อนนำไปสู่กระบวนการเจือสาร เนื่องจากค่าใช้จ่ายของกระบวนนี้สูงมาก จึงเป็นหัวข้อที่สำคัญของการหาข้อผิดพลาดในวงจรก่อนมีการส่งไปเจือสาร

การทวนสอบทางฮาร์ดแวร์ไม่มีความแตกต่างจากการทวนสอบซอฟแวร์หรือโปรแกรมต่างๆ เนื่องจากในปัจจุบันการออกแบบวงจรดิจิตอลหรือแอนะล็อกสามารถทำได้ด้วยการเขียนอธิบายในภาษาระดับสูง แล้วผ่านโปรแกรมช่วยในการสังเคราะห์เพื่อให้ได้วงจรที่สามารถนำไปใช้งานจริงได้

4.1 การทวนสอบและการตรวจสอบความสมเหตุสมผล (Verification and Validation)

การทวนสอบและการตรวจสอบความสมเหตุสมผลเป็นกระบวนการตรวจสอบเพื่อยืนยันคุณลักษณะให้ตรงกับความต้องการ ระหว่างการทวนสอบและการตรวจสอบความสมเหตุสมผลมักเกิดความสับสนในการใช้และความหมายเป็นอย่างมาก จากนิยามของ Boehm ในปี1979[อ้างตาม Ian Sommervile, 1992] กล่าวไว้ว่า *Validation : Build right product? Verification : Build product right?* การทวนสอบเป็นการยืนยันเพื่อให้ได้ตรงกับข้อกำหนดคุณลักษณะที่ต้องการ ส่วนการตรวจสอบความสมเหตุสมผลเกี่ยวกับการตรวจโมดูลที่สร้างให้มีการทำงานถูกต้องโดยเปรียบเทียบการความต้องการของผลิต การทวนสอบทำได้หลายวิธีเช่น การทดสอบ (Testing), การทวนสอบอย่างมีแบบแผน (Formal Verification) หรือการจำลองการทำงาน (Simulation-based Verification) แต่ในงานวิจัยนี้ใช้การทวนสอบอย่างมีแบบแผนและการจำลองการทำงานเพื่อทวสอบการทำงานของไมโครโพรเซสเซอร์ที่ออกแบบซึ่งได้ประยุกต์หลักของการทดสอบระดับชิ้นส่วนย่อยในระหว่างอยู่ในขั้นตอนการออกแบบวงจร รวมถึงการทดสอบทั้งระบบที่มีค่าคงที่ของเวลาเข้ามาเกี่ยวข้องด้วย กระบวนการทดสอบมีทั้งแบบ Static และ Dynamics ดังรูปที่ 4.1

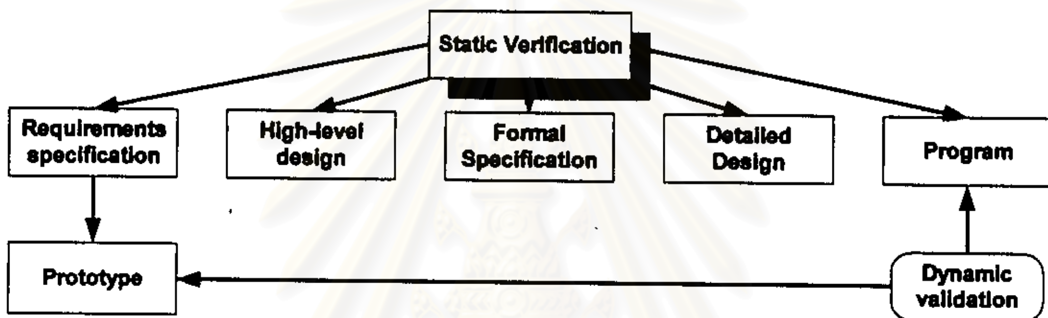
1. Static เป็นการวิเคราะห์และทดสอบระบบที่ไม่มีการเปลี่ยนแปลงและคงที่ เช่นการวิเคราะห์เอกสาร, ไดอะแกรม ที่ออกแบบ และโค้ดของโปรแกรม ซึ่งสามารถนำไปประยุกต์เข้ากับทุกขั้นตอนของการออกแบบได้

2. Dynamics เป็นเทคนิคการทดสอบการทำงานของโมดูลที่สร้างขึ้น สามารถมีการเปลี่ยนแปลงได้ตามสถานการณ์หรืออินพุทที่ได้รับมา

ชนิดของการทดสอบข้อมูลสามารถแบ่งออกได้เป็น

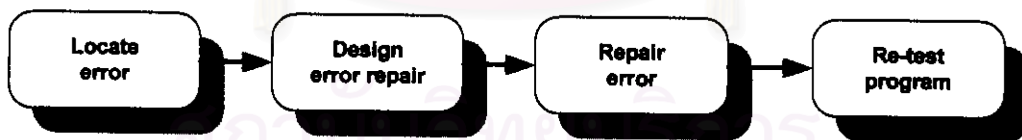
1. Statistical Testing ใช้ในการทดสอบประสิทธิภาพของโมดูลและความน่าเชื่อถือของระบบที่ทำการออกแบบ

2. Defect Testing เป็นการหาจุดที่โมดูลทำงานไม่ตรงกับคุณลักษณะที่กำหนด



รูปที่ 4. 1 การตรวจสอบความถูกต้องและการทวนสอบแบบ Static และ dynamic

การหาจุดบกพร่อง (Debugging) คือกระบวนการที่สามารถตรวจหาข้อผิดพลาดในโปรแกรมแล้วทำการปรับเปลี่ยนให้ถูกต้อง ดังอธิบายได้ในรูปที่ 4.2



รูปที่ 4. 2 ขั้นตอนการหาจุดบกพร่อง

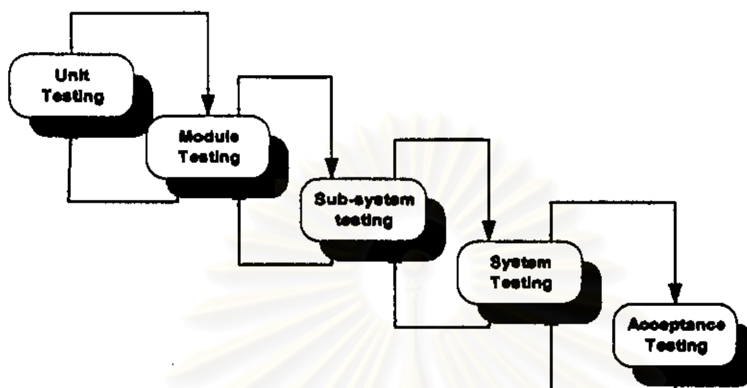
4.2 กระบวนการทดสอบ (Testing Process)

ในระบบที่มีขนาดใหญ่และซับซ้อน การทดสอบระบบควรทำการแบ่งระบบใหญ่ให้เป็นระบบย่อย ดังนั้นกระบวนการทดสอบจึงสามารถแบ่งออกได้เป็นหลายระดับ ดังรูปที่ 4.3 ดังนี้

4.2.1 การทดสอบแต่ละชิ้นส่วน (Component Testing) คือการแยกทดสอบการทำงานในแต่ละชิ้นส่วนอย่างอิสระ โดยถือเอา 1 เอนทิตี้ เป็น 1 ชิ้นส่วน

4.2.2 การทดสอบเมื่อรวมส่วนต่าง (Integration Testing) คือเป็นการทดสอบการเชื่อมต่อและทำงานร่วมกันของแต่ละชิ้นส่วน เพื่อให้สามารถทำงานรวมกันทั้งระบบได้อย่างถูกต้อง

4.2.3 การทดสอบส่วนผู้ใช้ (User Testing) คือการทดสอบส่วนที่มีการติดต่อกับผู้ใช้โดยตรงได้แก่ส่วนของอินพุทและเอาท์พุทของระบบ

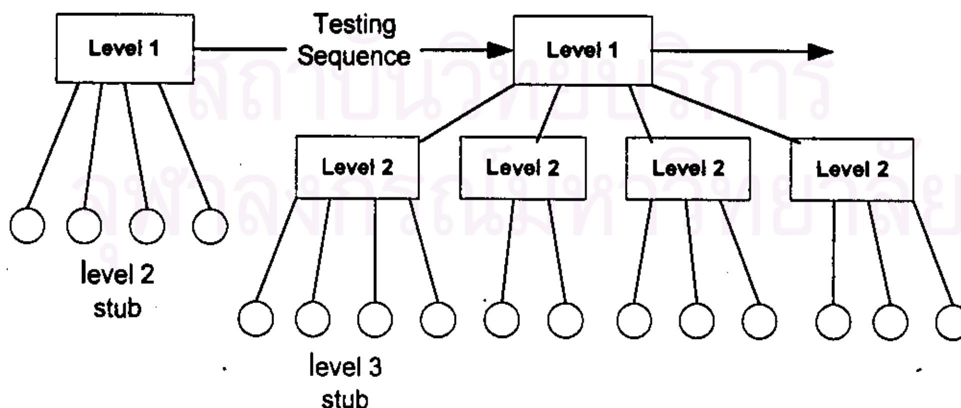


รูปที่ 4. 3 กระบวนการทดสอบ

4.3 กลยุทธ์การทดสอบ (Testing Strategies)

4.3.1. การทดสอบจากบนลงล่าง (Top-down testing)

การทดสอบจากบนลงล่างเป็นการทดสอบระบบในระดับสูงก่อนที่จะมีการทดสอบในรายละเอียดของการทำงาน นั่นคือจะมีความเป็นหลักนามธรรมมากที่สุดซึ่งจะแทนหนึ่งบล็อกของระบบด้วย ชิ้นส่วนเล็กๆที่เรียกว่า สตับ (Stub) ดังแสดงในรูปที่ 4.4



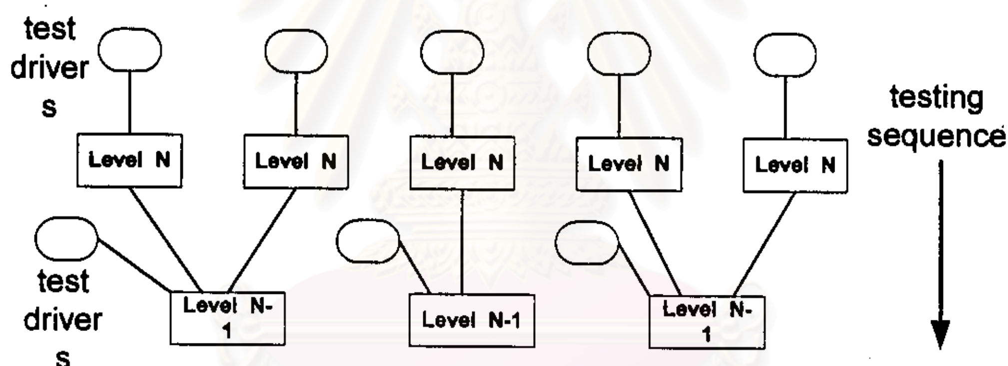
รูปที่ 4. 4 การทดสอบแบบบนลงล่าง

การทดสอบจากบนลงล่างควรใช้ควบคู่กับการออกแบบแบบบนลงล่าง ข้อดีของการทดสอบแบบนี้คือสามารถพบข้อผิดพลาดได้เร็วในช่วงแรกของการพัฒนาระบบก่อนที่จะลงไปในรายละเอียดของ

ระบบแต่มีข้อจำกัดที่ การทดสอบแบบบนลงล่างสามารถทำได้ยาก เนื่องจากจำเป็นต้องสร้างสตับ ในการจำลองการทำงานในระดับที่ต่ำกว่าลงมาซึ่งถ้าแต่ละชิ้นส่วนมีความซับซ้อนทำให้ไม่สามารถสร้างสตับเพื่อจำลองการทำงานได้ถูกต้อง ข้อเสียอีกประการหนึ่งเกิดจากผลลัพธ์ของการทดสอบอาจยากที่จะสังเกตในระดับบนของระบบส่วนมากจะไม่สร้างผลลัพธ์ออกมา ทำให้ผู้ทดสอบจำเป็นต้องสร้างสิ่งแวดล้อมขึ้นเองให้กับระบบเพื่อให้ได้ผลลัพธ์ที่ต้องการทดสอบ

4.3.2. การทดสอบจากล่างขึ้นบน (Bottom-up testing)

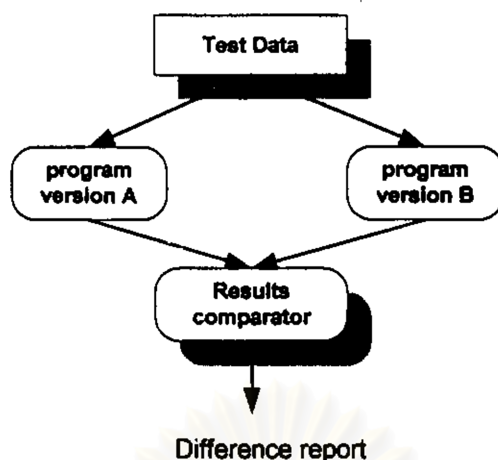
การทดสอบจากลางขึ้นบนจะตรงกับข้ามกับการทดสอบจากบนลงล่าง ซึ่งทำการทดสอบในรายละเอียดของแต่ละชิ้นส่วนขึ้นไปหาระบบใหญ่ทั้งหมดดังรูปที่ 4.5 ถ้านำเอาการพัฒนาแบบบนลงล่างมารวมกับการทดสอบจากล่างขึ้นบนทำให้ทุกส่วนของระบบมีการสร้างขึ้นก่อนที่มีการทดสอบ ซึ่งการแก้ไขข้อผิดพลาดอาจทำให้ต้องทำการออกแบบใหม่ เนื่องจากระบบทั้งหมดถูกสร้างเสร็จทั้งหมดแล้ว แต่อย่างไรก็ตามถ้ามีการตรวจสอบการทำงานในแต่ละขั้นตอนของการออกแบบแบบบนลงล่างจะทำให้ระบบและแต่ละชิ้นส่วนมีความสมบูรณ์มากขึ้น



รูปที่ 4. 5 การทดสอบจากล่างขึ้นบน

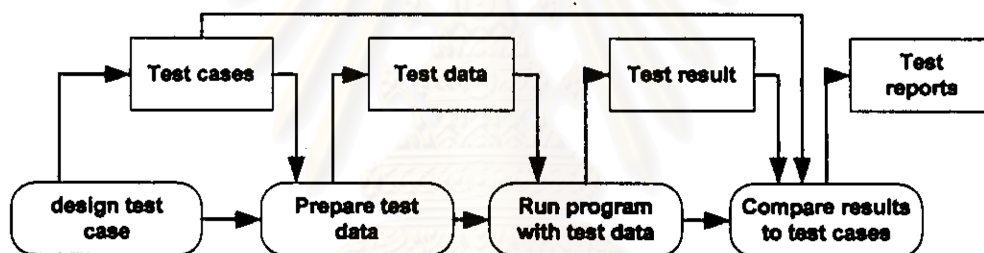
4.3.3 Back-to-back testing

เป็นการทดสอบที่นำเอาระบบ 2 ตัวมาทำการเปรียบเทียบกันดังรูปที่ 4.6 ดังนั้นสิ่งที่ต้องเตรียมในการทดสอบแบบ Back-to-back คือการสร้างชุดทดสอบ โปรแกรมที่ช่วยในการจำลองการทำงานและส่วนเก็บชุดทดสอบพร้อมผลลัพธ์ที่ถูกต้องแยกอยู่คนละที่ สุดท้ายคือส่วนของการเปรียบเทียบผลลัพธ์ที่ควรเป็นกับผลลัพธ์ที่ได้จริงจากการจำลองการทำงาน



รูปที่ 4. 6 Back-to-Back Testing

4.4 การตรวจสอบหาข้อผิดพลาด



รูปที่ 4. 7 ผลกระทบในแต่ละขั้นตอนการทดสอบ

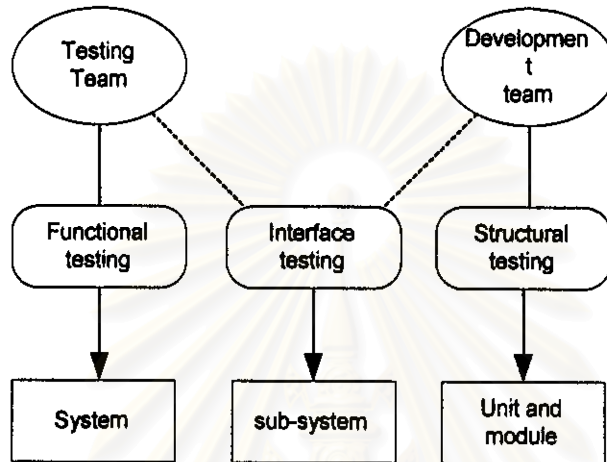
จากรูปที่ 4.7 สามารถแยกความแตกต่างของชุดทดสอบและชุดข้อมูลได้โดยชุดข้อมูลเป็นอินพุท ที่ถูกป้อนให้กับระบบที่ต้องการทดสอบ ส่วนชุดทดสอบเป็นทั้งอินพุทและเอาท์พุทที่เป็นคุณลักษณะที่ต้องการภายใต้การทดสอบการทำงานทั้งหมดของระบบ โดยชุดข้อมูลอาจถูกสร้างได้โดยอัตโนมัติ (ซึ่งแตกต่างจากในวิทยานิพนธ์ที่จะใช้ "Test case" ตามนิยามของชุดข้อมูล) ซึ่งสามารถแบ่งลักษณะของการตรวจสอบข้อผิดพลาดออกได้เป็น

1. Functional (Black-box testing) เป็นการทดสอบที่ถูกวิเคราะห์จากคุณลักษณะที่กำหนด (Specification) ถือเป็นการทดสอบที่ละเอียดน้อยที่สุดเนื่องจากสามารถสังเกตและตรวจสอบได้เฉพาะส่วนที่เป็นอินพุทและเอาท์พุทของระบบเท่านั้น

2. Structural (white-box testing) เป็นการทดสอบที่วิเคราะห์จากโครงสร้างและโมเดลที่สร้างเป็นสำคัญ การทดสอบในรูปแบบนี้สามารถตรวจการทำงานของดาต้าพาทในระบบได้

3. Interface test เป็นการทดสอบที่วิเคราะห์จากคุณลักษณะที่กำหนดไว้ผสมผสานกับความรู้ทางด้านการทำงานเชื่อมต่อกันระหว่างแต่โมดูลในระบบ การทดสอบชนิดนี้เหมาะอย่างยิ่งในงานทางด้านระบบเชิงวัตถุ (Object-oriented)

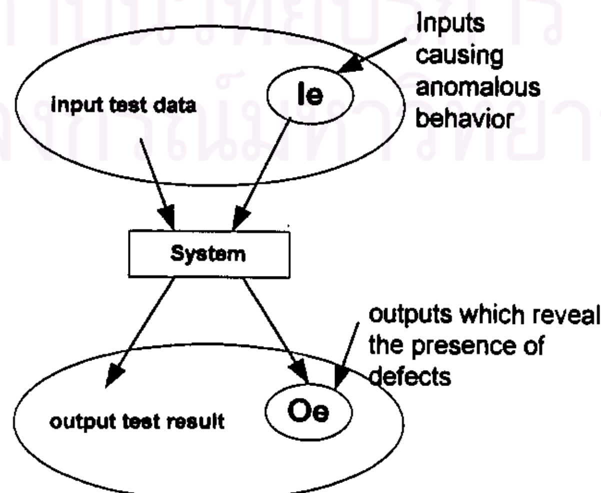
ดังแสดงในรูปที่ 4.8 เป็นการนำเอาการทดสอบแบบต่างๆประยุกต์เข้ากับการตรวจสอบระบบที่ออกแบบ



รูปที่ 4. 8 การประยุกต์วิธีการทดสอบเข้ากับระบบ

4.4.1 Black-box Testing

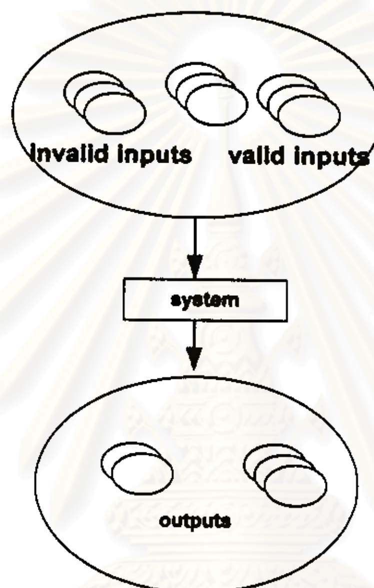
เป็นการทดสอบเฉพาะในส่วนของอินพุทและเอาท์พุทของระบบที่ออกแบบ ซึ่งเรียกอีกชื่อหนึ่งว่า Functional Testing เนื่องจากเป็นการตรวจสอบเฉพาะการทำงานตามคำสั่งที่ได้รับ มักใช้กับงานทางด้านการคำนวณทางคณิตศาสตร์หรือลอจิก แต่การทดสอบแบบนี้เหมาะสำหรับการตรวจสอบการทำงานในลักษณะของการออกแบบโมดูลที่ไม่สนใจโครงสร้างภายในแต่ให้ความสำคัญกับผลลัพธ์ที่ต้อง[illegible]มากกว่า ดังรูปที่ 4.9 ทั้งนี้ขึ้นอยู่กับประสบการณ์มากกว่ากฎเกณฑ์ที่ตายตัวแบ่งออกได้เป็นรูปแบบต่างๆดังนี้



รูปที่ 4. 9 black-box testing

Equivalence partitioning

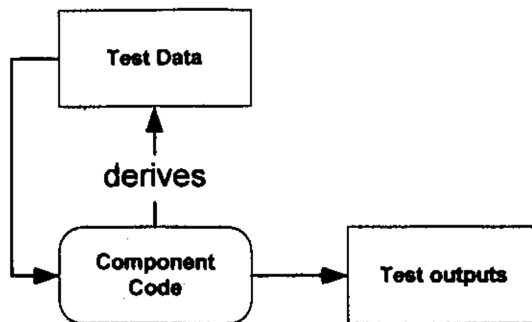
ข้อมูลที่เป็นอินพุทที่เข้าสู่ระบบทดสอบสามารถแบ่งออกเป็นประเภทต่างๆซึ่งในแต่ละประเภทจะมีคุณสมบัติเหมือนกัน เช่น ประเภทของจำนวนเต็มบวก, เต็มลบ จากรูปที่ 4.10 ในแต่ละส่วนของประเภทสามารถแสดงได้ดังรูปวงรี ดังนั้นชุดทดสอบในแต่ละประเภทจะถูกเลือกเอาขึ้นมาทดสอบเป็นตัวแทนของแต่ละประเภทนั้นๆ จากแนวความคิดนี้นำไปประยุกต์ใช้ในการเลือกชุดทดสอบเพื่อจำลองการทำงานโดยจะเลือกคำสั่งทดสอบที่เป็นตัวแทนกลุ่มซึ่งมีคุณสมบัติหรือการทำงานที่เหมือนกันทั้งหมด เพื่อลดจำนวนชุดคำสั่งในการทดสอบไมโครโพรเซสเซอร์ที่มีขนาดใหญ่และซับซ้อน



รูปที่ 4. 10 Equivalence partitioning

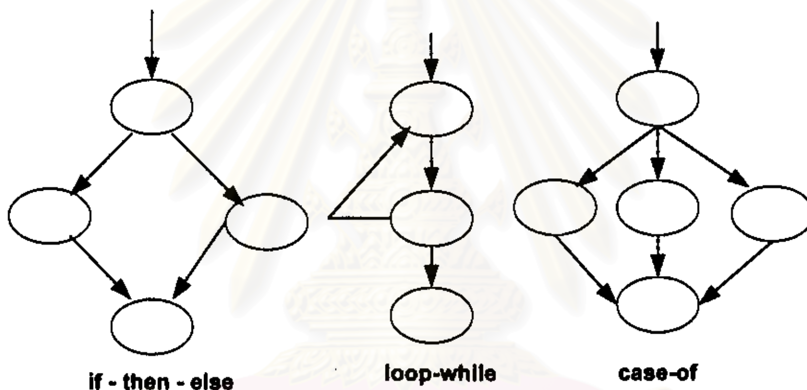
4.4.2 Structural Testing

การตรวจสอบโครงสร้างหรือเรียกได้อีกชื่อว่า "White-box or glass-box" ตามรูปที่ 4.11 โดยผู้ตรวจสอบสามารถวิเคราะห์โค้ดและโครงสร้างของโมดูลแต่ละชิ้นส่วนได้อย่างละเอียด ข้อดีของการตรวจสอบแบบนี้คือสามารถวิเคราะห์ได้ว่าชุดคำสั่งที่ให้ทดสอบสามารถครอบคลุมการทำงานทั้งหมดของระบบที่ต้องการตรวจสอบหรือไม่ โดยจะมีการวิเคราะห์เส้นทางการทำงานของในแต่ละคำสั่งเพื่อพิจารณา



รูปที่ 4. 11 White-box Testing

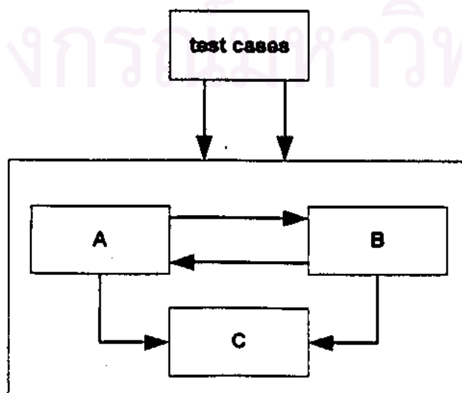
ในการพิจารณาเส้นทางการทำงานของแต่ละคำสั่งสามารถใช้แผนผังสายงานแสดงการทำงานและความครอบคลุมการตรวจสอบในแต่ละบรรทัดของโค้ดที่เขียนอธิบายการทำงานตัวอย่างดังในรูปที่ 4.12 ซึ่งเป็นแผนผังแสดงเส้นทางของโค้ดที่อธิบายการทำงานได้แก่ If-then-else, loop-while และ case-of เป็นต้น



รูปที่ 4. 12 รูปแบบของแผนภาพสายงาน

4.1.3 Interface testing

เป็นการทดสอบเมื่อเอาโมดูลแต่ละชิ้นส่วนมารวมกันเพื่อตรวจสอบหาข้อผิดพลาดของการเชื่อมต่อการทำงานโดยการป้อนอินพุทให้กับระบบซึ่งสามารถอธิบายได้ดังรูปที่ 4.13



รูปที่ 4. 13 Interface testing

ข้อผิดพลาดที่เกิดจากการเชื่อมต่อโดยส่วนมากพบบ่อยในระบบที่มีความซับซ้อนและสามารถแบ่งข้อผิดพลาดออกได้เป็น 3 ชนิด

1. ข้อผิดพลาดจากการใช้งานผิด ทั้งนี้เกิดเนื่องจากรูปแบบของการส่งข้อมูลระหว่างแต่ละชิ้นส่วนผิดพลาด

2. ข้อผิดพลาดที่เกิดจากความเข้าใจผิด เนื่องจากความไม่เข้าใจในการส่งข้อมูลหรือการเชื่อมต่อที่ใช้รูปแบบผิดพลาด โดยนักออกแบบทำการแบ่งส่วนกันออกแบบและไม่เข้าใจมาตรฐานในการนำเอาแต่ละส่วนมาเชื่อมต่อกัน จึงทำให้เกิดความผิดพลาด

3. ข้อผิดพลาดที่เกิดจากเวลาในการทำงาน มักเกิดขึ้นบ่อยกับระบบที่เป็น การทำงานแบบทันที (Real time) ดังนั้นเมื่อส่วนใดส่วนหนึ่งมีช่วงเวลาการทำงานที่ผิดพลาดไปทำให้ทั้งระบบทำงานผิดพลาดด้วย ข้อผิดพลาดชนิดนี้มักเกิดขึ้นและปัญหาที่สำคัญในระบบการทำงานแบบอสมวารด้วย

4.5 รูปแบบของการทวนสอบอย่างมีแบบแผน

ในงานวิจัยนี้ใช้การทวนสอบ 2 แบบคือการทวนสอบอย่างมีแบบแผนและการจำลองการทำงาน ในส่วนนี้จะกล่าวถึงการทวนสอบอย่างมีแบบแผนที่ใช้ในงานวิจัยนี้ งานวิจัยได้นำเสนอการนำแนวคิดการใช้การทวนสอบอย่างมีแบบแผนชนิดที่เรียกว่า Model Checking[E.M Clarke and Kurshan, 1996] และใช้วิธีการของASM ในระดับของหลักนามธรรมโดยผลและการวิเคราะห์อยู่ในภาคผนวก ข.

การตรวจสอบโมดูล (Model Checking)

เป็นเทคนิคในการทวนสอบโครงสร้างโมดูลของระบบ ทำการตรวจสอบคุณสมบัติของวงจรที่ต้องการในโมดูลนั้นๆ และระบบจะมีรูปร่างลักษณะที่แน่นอนเนื่องจากเป็นไฟน์ใน วิธีการนี้ถูกใช้ในการทวนสอบฮาร์แวร์และโปรโตคอลเป็นหลัก การใช้การตรวจสอบโมดูลสามารถทำการแยกระบบออกเป็นส่วนย่อยเพื่อตรวจสอบได้ ซึ่งทำให้ง่ายและสะดวกขึ้น โดยปัจจุบันถูกใช้ในระดับที่มีขนาด 100 – 200 สถานะ และสามารถประยุกต์ใช้กับสถานะในรูปแบบที่เป็นหลักนามธรรมได้สูงถึง 10^{120} กรณี ดังนั้นจึงเป็นวิธีการที่มีประสิทธิภาพสูงที่ควรนำมาพัฒนาเป็นเครื่องมือในการทวนสอบการออกแบบวงจรเชิงพาณิชย์

เทคนิคการทวนสอบแบบนี้จะเน้นที่การทวนสอบคุณสมบัติและลักษณะเฉพาะที่ต้องการให้ทำงานเท่านั้น เนื่องจากไม่สามารถทำการทวนสอบระบบทั้งหมดที่มีขนาดใหญ่ได้ ตัวอย่างเช่น การทวนสอบรีจิสเตอร์ จะทำการทวนสอบเฉพาะส่วนของการอ่านและเขียนข้อมูลโดยยึดถือลักษณะสำคัญของรีจิสเตอร์ว่าจะไม่สามารถให้สัญญาณการอ่านและเขียนเกิดขึ้นพร้อมกัน หากมีการทำงานในส่วนนี้ถูกต้องก็ถือว่าผ่านการทวนสอบแบบตรวจสอบโมดูล ดังนั้นภาษาที่ใช้ในการออกแบบเพื่อให้สามารถทวนสอบกับวิธีการแบบนี้จึงมีรายละเอียดของการทำงานของวงจรค่อนข้างน้อยกว่า เช่น ภาษา Temporal Logic

การทวนสอบอย่างมีแบบแผนโดยการใช้วิธีการของ Abstract State Machine (ASM)

วิธีการของ ASM เป็นที่รู้จักซึ่งนำหลักการของ Algebra และถูกนำเสนอครั้งแรกโดย Gurevich ในปี 1988 เป็นวิธีการที่ง่ายและนับว่ามีประสิทธิภาพเหมาะสำหรับการกำหนดคุณลักษณะและทวนสอบทั้งระบบที่เป็นซอฟแวร์และฮาร์แวร์ ASM ถูกนำไปประยุกต์ใช้กับภาษาของการเขียนโปรแกรม, โปรโตคอลการกระจาย, สถาปัตยกรรมและอื่นๆ จากการศึกษาค้นคว้า วิธีการของ ASM พบว่าระบบทั้งซอฟแวร์และฮาร์ดแวร์สามารถถูกสร้างขึ้นได้ในระดับของ natural abstraction โดยการใช้ ASM ผลที่ได้จากการเปลี่ยนโครงสร้างภาษาที่ออกแบบ VHDL ให้อยู่ในรูปของทรานสิชั่น เพื่อนำไปเป็นเงื่อนไขในการทำงานในไฟไนสเตตแมชชีนที่เป็นคุณลักษณะที่ต้องการสร้างขึ้นจากภาษาการทำงานที่ได้จากงานของ Gurevich ในปี 1988 ซึ่งผลการทวนสอบถูกต้องสถานะสุดท้ายของการทำงานจะต้องอยู่ในสถานะเอกเซบชั่นเสมอ

สถานะ

สถานะของ ASM เป็นโครงสร้างที่มีความหมายในแนวของ First-order logic (FOL) ยกเว้นส่วนของความสัมพันธ์จะอยู่ในรูปของฟังก์ชันค่าตรรกะแบบบูล คำศัพท์ (Vocabulary) ที่ใช้งานเป็นกลุ่มของชื่อฟังก์ชันการทำงาน ทุกๆคำศัพท์ใน ASM จะประกอบไปด้วยสัญลักษณ์ทางลอจิกเช่น True, False, Undef, เครื่องหมายเหมือนที่ใช้กับตรรกะแบบบูลโดยการใช้แทนว่า Bool สถานะ State(S) ของคำศัพท์จะไม่อยู่เซตของ X สัญลักษณ์ f ของ Arity r เป็นการทำงานนอกเหนือจากในเซต X

ให้ f เป็นสัญญลักษณะความสัมพันธ์ของ Arity r ซึ่งจะมีค่าเป็นจริงหรือเท็จสำหรับทุกๆ r-tuple ของ element ของ S ถ้า f เป็น unary จะสามารถมองเป็นเหมือน Universe ดังนั้นกลุ่มของ element a $f(a)$ เป็นจริง เช่น Bool เป็น Universe ที่ประกอบด้วย 2 elements คือจริงและเท็จ ดังตารางที่ 4.1 และ 4.2

ให้ f เป็น r-ary ฟังก์ชันพื้นฐานและ U0, ..., Ur เป็น Universe เรากล่าวได้ว่า f มีชนิดเป็น U1 x ... x Ur → U0 ซึ่งถ้า $f(x)$ เป็น Universe U0 สำหรับทุกๆ X เป็นสมาชิกของ U1 x ... x Ur และ $f(x)$ มีค่าเป็น undef

ตารางที่ 4. 1 Universes in Specification Model

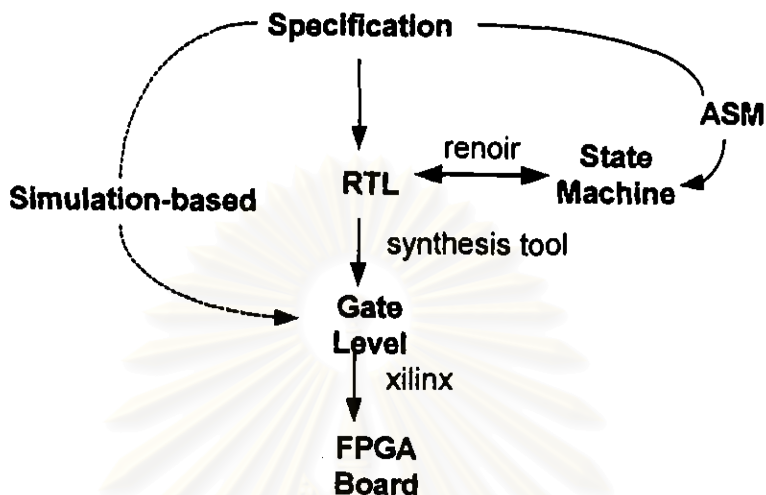
Stages	{fetch, decode, execute}
Bool	{true, false}
Bit	{0, 1}
Words	{0, ... , $2^{32}-1$}
Instructions	Valid ARM instructions
Register	{R0, ..., R15}
Flaglist	List of control flags
ALUOps	Opcodes for ALU instructions
Shifts	Type of shifts
Mode	{reg_write, reg_read}

ตารางที่ 4. 2 Function in Universe

Vocabulary	Functions
ExecuteOK	Bool
ALU	ALUOps x words x words x bits → words
Instr	Instructions
Satisfies	Flaglists x flaglists → Bool
ConCode	Instruction → flaglists
WritesResult	Instruction → Bool
Contents	Register → words
DesReg	Register
ALUOps	Instructions → ALUOps
Aop, Bop	Words
Reg	Words
Carry	Flaglists → Bits
Status	Flaglists
UpdateStatus	Flaglists x ALUOps x words x words x bits →flaglists
ShiftCarry	Words x shifts x words x bits → bits
AopReg, BopReg, ShiftReg, DestOp	Instructions → registers

งานในวิทยานิพนธ์นี้ได้นำเอาหลักการของ Finite State Machine (FSM) มาแทนคุณลักษณะที่กำหนดไว้ของระบบและใช้การยอมรับระบบ (Accepted) สเตตแมชชีนด้วยวิธีการของ Finite Automata (FA) ซึ่งแทนการทำงานของวงจรที่จะทวนสอบด้วย String ที่เป็นเงื่อนไขของการเปลี่ยนสถานะการทำงาน โดยลักษณะของภาษา VHDL ซึ่งมีพื้นฐานบนภาษาปาสคาล สามารถแทนโค้ดในแต่ละบรรทัดให้ออกมาเป็นรูปของสายอักขระ (String) ได้ แนวความคิดนี้มาจากการสร้าง FA ด้วยภาษา LISP ซึ่งก็มีโครงสร้างพื้นฐานเป็นภาษาปาสคาลเหมือน VHDL จึงทำให้สามารถเทียบโครงสร้างภาษาของ VHDL กับ LISP ได้ หมายถึงสามารถออกแบบวงจรดิจิตอลได้ด้วยภาษา LISP ซึ่งมีลักษณะและการใช้งานคล้ายกับภาษา VHDL ดังนั้นถ้าสร้างตัวแปลภาษาให้อยู่ในรูปของ LISP พร้อมทั้งสร้างกฎของการเปลี่ยนสถานะ

การทำงาน เพื่อตรวจสอบการทำงานที่เข้าสู่ในแต่ละสถานะที่เป็นไปได้ หลักการและแนวความคิดนี้ได้ถูกประยุกต์เพื่อใช้ในการทวนสอบในขั้นตอนการออกแบบดังรูปที่ 4.14 ระหว่างโมดูลที่ออกแบบรูปแบบของอาร์ทีแอล กับคุณลักษณะที่ต้องการ



รูปที่ 4.14 การประยุกต์เอา ASM ใช้ในขั้นตอนการออกแบบในระดับสูง

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย