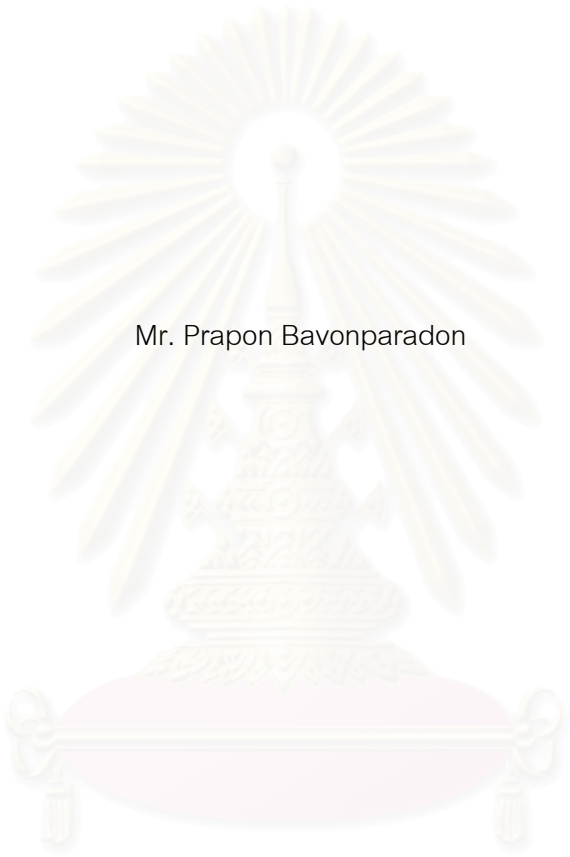
การทวนสอบรูปนัยในระดับถ่ายโอนเรจิสเตอร์ของหน่วยประมวลผล
โดยการตรวจสอบแบบจำลองเชิงสัญลักษณ์

นาย ประพนธ์ บวรภราดร

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2545
ISBN 974-17-1537-4

FORMAL VERIFICATION AT A REGISTER TRANSFER LEVEL OF A PROCESSOR BY SYMBOLIC MODEL CHECKING

Mr. Prapon Bavonparadon



A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering in Computer Engineering
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University
Academic Year 2002

ISBN 974-17-1537-4

หัวข้อวิทยานิพนธ์	การทวนสอบรูปนัยในระดับถ่ายโอนเรจิสเตอร์ของหน่วยประมวลผลโดยการตรวจสอบแบบจำลองเชิงสัญลักษณ์
โดย	นาย ประพนธ์ บวรภราดร
สาขาวิชา	วิศวกรรมคอมพิวเตอร์
อาจารย์ที่ปรึกษา	รองศาสตราจารย์ ดร. ประภาส จงสถิตย์วัฒนา

คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย อนุมัติให้นับวิทยานิพนธ์ฉบับนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญามหาบัณฑิต

…….. คณบดีคณะวิศวกรรมศาสตร์
(ศาสตราจารย์ ดร. สมศักดิ์ ปัญญาแก้ว)

คณะกรรมการสอบวิทยานิพนธ์

…… ประธานกรรมการ
(รองศาสตราจารย์ ดร. สาธิต วงศ์ประทีป)

…….. อาจารย์ที่ปรึกษา
(รองศาสตราจารย์ ดร. ประภาส จงสถิตย์วัฒนา)

……กรรมการ
(อาจารย์ ดร. อาทิตย์ ทองทักษ์)

……กรรมการ
(อาจารย์ ดร. ทรงศักดิ์ รองวิริยะพานิช)

Thesis Title	Formal Verification at a Register Transfer Level of a Processor by Symbolic Model Checking
By	Prapon Bavonparadon
Field of Study	Computer Engineering
Thesis Advisor	Associate Professor Prabhas Chongstitvatana , Ph.D.

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master 's Degree

…… Dean of Faculty of Engineering
(Professor Somsak Panyakeow , D. Eng.)

THESIS COMMITTEE

…….. Chairman
(Associate Professor Sartid Vongpradhip , Ph.D.)

…….. Thesis Advisor
(Associate Professor Prabhas Chongstitvatana , Ph.D.)

…….. Member
(Arthit Thongtak , D.Eng.)

…….. Member
(Songsakdi Rongviriyapanish , Ph.D.)

ประพนธ์ บวรภราดร : การทวนสอบรูปนัยในระดับถ่ายโอนเรจิสเตอร์ของหน่วยประมวลผลโดยการตรวจสอบแบบจำลองเชิงสัญลักษณ์. (FORMAL VERIFICATION AT A REGISTER TRANSFER LEVEL OF A PROCESSOR BY SYMBOLIC MODEL CHECKING) อ. ที่ปรึกษา : รศ. ดร. ประภาส จงสถิตย์วัฒนา, 126 หน้า. ISBN 974-17-1537-4.

ในวิทยานิพนธ์นี้ได้นำเสนอการทวนสอบรูปนัยของหน่วยประมวลผลที่ถูกออกแบบเพื่อใช้ในระบบเว็บเซิร์ฟเวอร์แบบฝังตัว โดยที่การทวนสอบกระทำกับรายละเอียดการออกแบบในระดับถ่ายโอนเรจิสเตอร์ ซึ่งเป็นระดับที่สามารถนำไปสังเคราะห์วงจรได้ ในการทวนสอบจะทำโดยใช้โปรแกรมคาเดนค์เอสเอมวีซึ่งทำงานโดยใช้เทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์ซึ่งมีปัญหาการเพิ่มอย่างรวดเร็วของสถานะ ในวิทยานิพนธ์จึงนำเสนอวิธีต่างๆที่ใช้แก้ปัญหาดังกล่าว และในวิทยานิพนธ์นี้ยังได้นำเสนอวิธีทวนสอบแบบลำดับขั้นซึ่งแบ่งการทวนสอบเป็นหลายขั้นตอนโดยแต่ละขั้นตอนจะมีรายละเอียดเพิ่มขึ้นจากขั้นตอนแรกไปจนถึงขั้นตอนสุดท้าย ซึ่งวิธีการนี้ช่วยทำให้การหาสาเหตุของปัญหาทำได้ง่ายขึ้นในกรณีที่เกิดข้อผิดพลาดในวงจรและทำให้สามารถทวนสอบได้เสร็จในเวลาที่สมเหตุผล การทวนสอบได้กระทำกับหน่วยประมวลผลที่ไม่ซับซ้อนและกระบวนการทวนสอบสามารถกระทำได้สำเร็จ

ภาควิชา วิศวกรรมคอมพิวเตอร์	ลายมือชื่อนิสิต.....
สาขาวิชา วิศวกรรมคอมพิวเตอร์	ลายมือชื่ออาจารย์ที่ปรึกษา.....
ปีการศึกษา 2545	ลายมือชื่ออาจารย์ที่ปรึกษาร่วม.....

4470395921 : MAJOR COMPUTER ENGINEERING

KEY WORD: FORMAL VERIFICATION / CADENCE SMV / SYMBOLIC MODEL CHECKING / PROCESSOR VERIFICATION / RTL (REGISTER TRANSFER LEVEL)

PRAPON BAVONPARADON : FORMAL VERIFICATION AT A REGISTER TRANSFER LEVEL OF A PROCESSOR BY SYMBOLIC MODEL CHECKING. THESIS ADVISOR : ASSOC. PROF. PRABHAS CHONGSTITVATANA, 126 pp. ISBN 974-17-1537-4.

This thesis proposes a technique for formal verification of a processor used in an embedded web server. The verification process is performed at the RTL level of implementation, which can be synthesized by a synthesis tool. We use Cadence SMV tool, which employs the symbolic model checking technique, as the verification tool. This technique has the state explosion problem. In this thesis, we propose many methods to solve this problem. Furthermore, we propose a stepwise verification method that the details of design are increased in each step. This method makes the error finding process easier and enables the verification process to finish in a reasonable amount of time. The methods are illustrated on a simple processor. The whole design can be verified successfully.

Department Computer Engineering Student's signature.....
Field of study Computer Engineering Advisor's signature.....
Academic year 2002 Co-advisor's signature.....

กิตติกรรมประกาศ

ขอขอบคุณคุณพ่อคุณแม่ที่ได้ให้ชีวิตและการเลี้ยงดูเอาใจใส่เป็นอย่างดีตลอดมา รวมทั้งเป็นแรงบันดาลใจจนกระทั่งข้าพเจ้าสามารถก้าวมาถึงจุดนี้ได้ ขอขอบคุณ รศ. ดร. ประภาส จงสถิตย์วัฒนา อาจารย์ที่ปรึกษาของข้าพเจ้า ที่เอาใจใส่ความก้าวหน้าของวิทยานิพนธ์ของข้าพเจ้าตลอดมา และขอบคุณที่ช่วยขัดเกลาวิทยานิพนธ์จนออกมาสมบูรณ์เช่นนี้ ขอขอบคุณทุกคนใน Intelligent System Lab ซึ่งช่วยให้คำแนะนำที่ดีเสมอมา และบรรยากาศวิชาการของที่นี่ช่วยหล่อหลอมความเป็นนักวิชาการให้กับข้าพเจ้า และขอขอบคุณเพื่อนทุกคนที่เรียน วศ.ม. ด้วยกันที่ให้มิตรภาพที่ดีต่อกันเสมอมา

และท้ายสุดขอขอบคุณทุนอุดหนุนการศึกษาในระดับบัณฑิตศึกษาของจุฬาลงกรณ์มหาวิทยาลัยเพื่อเฉลิมฉลองในวโรกาสที่พระบาทสมเด็จพระเจ้าอยู่หัวทรงเจริญพระชนมายุครบ 72 พรรษา หากปราศจากทุนนี้ข้าพเจ้าคงไม่ได้สามารถสำเร็จการศึกษาได้

สารบัญ

หน้า

สารบัญ(ต่อ)

หน้า

สารบัญ(ต่อ)

หน้า

สารบัญตาราง

หน้า

สารบัญภาพ

หน้า

สารบัญภาพ(ต่อ)

หน้า

บทที่ 1

บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

1.1.1 การทวนสอบ

เนื่องจากการแข่งขันในทางการตลาดที่สูงขึ้นทำให้บริษัทที่พัฒนาผลิตภัณฑ์ด้านฮาร์ดแวร์จำเป็นต้องเพิ่มขีดความสามารถของผลิตภัณฑ์เป็นผลให้ความซับซ้อนของวงจรฮาร์ดแวร์เพิ่มขึ้น ในขณะเดียวกันก็จำเป็นต้องลดระยะเวลาการพัฒนาผลิตภัณฑ์เพื่อให้เท่าทันกับการแข่งขัน จากสถานการณ์เช่นนี้ทำให้ความสำคัญของกระบวนการทวนสอบวงจร (verification process) ย่อมมีมากขึ้น เนื่องจากเมื่อวงจรมีความซับซ้อนมากขึ้นโอกาสที่จะเกิดข้อผิดพลาดย่อมมีมากขึ้นด้วย และเมื่อระยะเวลาการพัฒนาผลิตภัณฑ์สั้นลงจึงจำเป็นต้องใช้กระบวนการทวนสอบที่มีประสิทธิภาพสูงสุดเพื่อหาข้อผิดพลาดให้ได้มากที่สุดในเวลาอันจำกัด และสามารถรับประกันความถูกต้องของวงจรที่ถูกทวนสอบได้ดีที่สุด ข้อสังเกตหนึ่งที่น่าสนใจ (Rahinkar, Peterson และ Singh, 2001) คือ ในบริษัทชั้นนำซึ่งออกแบบผลิตภัณฑ์ฮาร์ดแวร์ประมาณ 70% ของต้นทุนการออกแบบผลิตภัณฑ์ต้องหมดไปกับกระบวนการทวนสอบวงจร

การทวนสอบคือกระบวนการตรวจสอบความถูกต้องตรงกันระหว่างข้อกำหนดวงจร (specification) และรายละเอียดการออกแบบ (implementation) โดยกระบวนการทวนสอบเป็นการตรวจสอบความถูกต้องของการออกแบบว่าทำงานได้อย่างที่ต้องการหรือไม่ ประโยชน์ของกระบวนการทวนสอบคือ ช่วยรับประกันความถูกต้องของวงจรที่ออกแบบ, ทำให้ผลิตภัณฑ์มีความน่าเชื่อถือมากขึ้น และช่วยค้นหาข้อผิดพลาดภายในวงจรระหว่างการออกแบบ ซึ่งถ้าหากข้อผิดพลาดนี้ยังคงอยู่จนถึงขั้นตอนการผลิตจะทำให้เกิดการสูญเสียเป็นอย่างมาก ดังนั้นจึงมีความจำเป็นอย่างมากที่จะต้องกำจัดข้อผิดพลาดทั้งหมดให้ได้ก่อนกระบวนการผลิต ทำให้การทวนสอบมีความสำคัญเป็นอย่างยิ่ง ในปัจจุบันการทวนสอบเป็นเรื่องที่ได้รับความสนใจเป็นอย่างมากทั้งในบริษัทเอกชนและมหาวิทยาลัย

แต่สำหรับการตรวจสอบความถูกต้องของวงจรจากกระบวนการผลิตเราจะเรียกว่ากระบวนการทดสอบ (testing process) ซึ่งตรวจสอบว่าวงจรที่ได้จากกระบวนการผลิตมีความผิดพร่อง (fault) เกิดขึ้นหรือไม่ โดยที่ความผิดพร่องเหล่านี้ไม่ได้เกิดจากความผิดพลาดในการออก

แบบแต่เป็นความผิดพลาดจากกระบวนการผลิต จะเห็นได้ว่าการทวนสอบและการทดสอบมีบทบาทที่ต่างกัน ในกระบวนการออกแบบผลิตภัณฑ์ฮาร์ดแวร์จำเป็นต้องมีทั้งการทวนสอบและการทดสอบ เพื่อให้ผลิตภัณฑ์มีความถูกต้องของการทำงานมากที่สุด สำหรับในวิทยานิพนธ์นี้จะสนใจเฉพาะการทวนสอบเท่านั้น

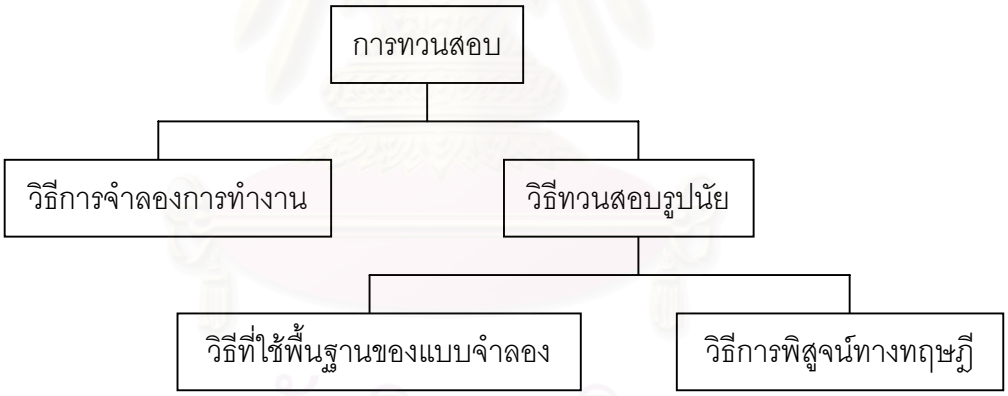
ปัจจุบันวิธีการทวนสอบวงจรที่นิยมใช้ในอุตสาหกรรม (Kropf, 1999) คือ วิธีการจำลองการทำงานของวงจร (simulation method) ซึ่งทำงานโดยการใส่อินพุตให้กับวงจรที่จะทวนสอบแล้วทำการสังเกตเอาต์พุต จากนั้นนำไปเปรียบเทียบกับเอาต์พุตที่ถูกต้องก็จะสามารถตรวจสอบได้ว่าวงจรเกิดข้อผิดพลาดขึ้นหรือไม่ ข้อเสียของวิธีการนี้คือ ไม่สามารถรับประกันความถูกต้องได้ 100% เนื่องจากไม่สามารถสร้างอินพุตให้ครอบคลุมอินพุตทุกแบบที่เป็นไปได้สำหรับวงจร วิธีการทวนสอบวงจรอีกกลุ่มหนึ่งที่กำลังได้รับความสนใจมากขึ้นคือ วิธีการทวนสอบรูปนัย (formal verification) ซึ่งเป็นเทคนิคที่ใช้กระบวนการทางคณิตศาสตร์มาเพื่อพิสูจน์ว่ารายละเอียดการออกแบบสามารถทำงานได้สอดคล้องกับข้อกำหนดของวงจร (Kropf, 1999) โดยที่ทั้งรายละเอียดของการออกแบบและข้อกำหนดของวงจรจะต้องอยู่ในรูปแบบที่มีความหมายชัดเจนเช่นในรูปแบบของสัญลักษณ์ทางคณิตศาสตร์ ดังนั้นความมั่นใจในผลการทวนสอบของวิธีการนี้จึงมีมากกว่าวิธีการจำลองการทำงาน แต่ปัญหาหนึ่งคือ วิธีการนี้ยังไม่สามารถใช้กับวงจรที่มีขนาดใหญ่มากได้ ในปัจจุบันมีนักวิจัยจำนวนมากที่พยายามประยุกต์วิธีการนี้เพื่อใช้ในกระบวนการทวนสอบวงจรฮาร์ดแวร์ที่ถูกออกแบบในเชิงพาณิชย์ให้มากขึ้น

1.1.2 การทวนสอบรูปนัย

วิธีการทวนสอบรูปนัยสามารถแบ่งได้เป็น (Kropf, 1999) วิธีที่ใช้พื้นฐานของแบบจำลอง (model-based approach) และวิธีการพิสูจน์ทางทฤษฎี (proof-theoretic approach) วิธีการแรกนั้นจะใช้ทฤษฎีที่เกี่ยวกับตรรกเชิงเวลา (temporal logic) และเครื่องสถานะจำกัด (finite state machine) โดยที่วิธีการนี้จะใช้วิธีการสำรวจในทุกสถานะที่เป็นไปได้ของวงจร และวิธีการนี้จะเหมาะสมเป็นพิเศษเมื่อใช้ทวนสอบวงจรควบคุม (control circuit) ที่มีความซับซ้อนมาก ข้อดีของวิธีการนี้คือกระบวนการทวนสอบวงจรสามารถดำเนินไปอย่างอัตโนมัติ วิธีนี้มีข้อเสียคือไม่สามารถใช้ทวนสอบวงจรที่มีขนาดใหญ่มากได้ เนื่องจากในวงจรที่มีขนาดใหญ่มากจะมีสถานะที่เป็นไปได้ทั้งหมดของวงจรมากเกินกว่าที่จะทดสอบได้ ปัญหานี้ถูกเรียกว่าปัญหาการเพิ่มอย่างรวดเร็วของสถานะ (state explosion problem) แต่เนื่องจากที่วิธีนี้มีความอัตโนมัติสูงทำให้ในปัจจุบัน

มีเครื่องมือสำหรับทวนสอบวงจรที่ใช้ในเชิงพาณิชย์ซึ่งใช้วิธีการนี้แล้ว (Bell Labs Design Automation, 1998)

ส่วนวิธีการพิสูจน์ทางทฤษฎีใช้พื้นฐานของตรรกแบบประพจน์ (propositional logic) โดยจะแทนวงจรที่ถูกทวนสอบในรูปของประพจน์ (proposition) ซึ่งถูกเชื่อมด้วยตัวปฏิบัติการเชิงตรรก (logic operator) และใช้สัจพจน์ (axiom) และทฤษฎี (theorem) ทางตรรกเพื่อพิสูจน์ให้ได้ว่ารายละเอียดการออกแบบมีความสอดคล้องกับข้อกำหนดของวงจร วิธีการนี้จะเหมาะสมกับการทวนสอบวงจรทางเดินข้อมูล (datapath circuit) ข้อดีของวิธีการนี้คือไม่ถูกจำกัดด้วยขนาดของวงจรที่ถูกทวนสอบ หรืออีกนัยหนึ่งคือสามารถทวนสอบวงจรที่มีขนาดใหญ่มากได้ดี แต่วิธีนี้ก็มีข้อเสียตรงที่ไม่สามารถดำเนินการทวนสอบอย่างอัตโนมัติได้ เนื่องจากในหลายขั้นตอนของการทวนสอบ ผู้ทวนสอบจะต้องโต้ตอบกับโปรแกรมช่วยทวนสอบ และผู้ที่จะทวนสอบวงจรจะต้องมีความเชี่ยวชาญในเรื่องตรรกแบบประพจน์อย่างเพียงพอ ซึ่งผู้ออกแบบวงจรโดยทั่วไปจะไม่มีพื้นฐานในส่วนนี้ เนื่องด้วยข้อไม่ดีในส่วนนี้ทำให้วิธีการนี้ยังคงเป็นเพียงการวิจัยระดับมหาวิทยาลัย และยังไม่ถูกใช้ในเชิงพาณิชย์มากนัก



รูปที่ 1.1 วิธีการในกลุ่มการทวนสอบ

ตารางที่ 1.1 เปรียบเทียบวิธีการจำลองการทำงานและวิธีการทวนสอบรูปนัย

วิธีการ	ข้อดี	ข้อเสีย
วิธีการจำลองการทำงาน	กระบวนการทวนสอบไม่ยุ่งยาก	รับประกันความถูกต้องไม่ได้
วิธีทวนสอบรูปนัย	รับประกันความถูกต้องได้, ไม่ต้องสร้างอินพุตเพื่อทวนสอบ	กระบวนการค่อนข้างซับซ้อน

ตารางที่ 1.2 เปรียบเทียบวิธีที่ใช้พื้นฐานของแบบจำลองและวิธีการพิสูจน์ทางทฤษฎี

วิธีการ	ข้อดี	ข้อเสีย
วิธีที่ใช้พื้นฐานของแบบจำลอง	มีความอัตโนมัติสูง	ปัญหาการเพิ่มอย่างรวดเร็วของสถานะ
วิธีการพิสูจน์ทางทฤษฎี	ไม่มีปัญหาเรื่องขนาดของวงจร	กระบวนการไม่อัตโนมัติ, ผู้ทวนสอบต้องมีความเชี่ยวชาญในทางตรรกแบบประพจน์

1.1.3 การทวนสอบรูปนัยของหน่วยประมวลผล

ในวิทยานิพนธ์นี้จะนำเสนอการทวนสอบรูปนัยของหน่วยประมวลผล (processor) ที่ถูกออกแบบเพื่อใช้ในระบบเว็บเซิร์ฟเวอร์แบบฝังตัว (embedded web server) (Piromsopa, 2000) โดยที่การทวนสอบจะกระทำกับหน่วยประมวลผลที่ถูกออกแบบในระดับถ่ายโอนเรจิสเตอร์ (Register Transfer Level) ซึ่งเป็นระดับที่สามารถนำไปสังเคราะห์วงจรได้ ในการทวนสอบจะทำโดยใช้โปรแกรมคาเดนค์เอสเอมวี (Cadence SMV) (McMillan, 1998) ซึ่งทำงานโดยใช้เทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์ (symbolic model checking) (McMillan, 1992 a.; Burch และคณะ, 1992) ซึ่งเป็นเทคนิคหนึ่งในกลุ่มวิธีที่ใช้พื้นฐานของแบบจำลอง

หลายปีมานี้มีการใช้วิธีการทวนสอบรูปนัยเพื่อตรวจสอบความถูกต้องของหน่วยประมวลผล (Biere และคณะ, 1999; Appenzeller และ Kuehlmann, 1995; Miller และ Srivas, 1995; Hosabettu, Srivas และ Gopalakrishnan, 1999; Windley, 1995; Burch, 1996) ซึ่งหลายงานวิจัยเป็นการประยุกต์กับหน่วยประมวลผลที่สร้างขึ้นในเชิงการค้า (Biere และคณะ, 1999; Appenzeller และ Kuehlmann, 1995; Miller และ Srivas, 1995) และส่วนใหญ่แล้วมักใช้วิธีการพิสูจน์ทฤษฎีเพื่อพิสูจน์ความถูกต้องของหน่วยประมวลผล (Miller และ Srivas, 1995; Hosabettu, Srivas และ Gopalakrishnan, 1999; Windley, 1995; Burch, 1996) ที่เป็นเช่นนี้เนื่องจากวิธีดังกล่าวสามารถใช้กับวงจรมีขนาดใหญ่มาก แต่เนื่องจากวิธีดังกล่าวไม่สามารถทำงานได้อย่างอัตโนมัติ และผู้ทวนสอบต้องมีความเชี่ยวชาญในทางตรรกแบบประพจน์ทำให้งานวิจัยเหล่านี้มักทำในมหาวิทยาลัย หรือบริษัทเอกชนขนาดใหญ่ที่มีผู้เชี่ยวชาญเฉพาะด้านเพื่อทำการทวนสอบโดยเฉพาะ อีกประการหนึ่งงานวิจัยที่ทำการทวนสอบความถูกต้องของหน่วยประมวลผลมักไม่ทวนสอบรายละเอียดทั้งหมดของหน่วยประมวลผล แต่จะทวนสอบเฉพาะบางส่วน (Biere และคณะ, 1999; Hosabettu, Srivas และ Gopalakrishnan, 1999; Burch, 1996) ที่

เป็นเช่นนี้เนื่องจากหน่วยประมวลผลที่ถูกทวนสอบมักมีความซับซ้อนสูงมากและมีขนาดของวงจรใหญ่มาก ดังนั้นการทวนสอบรายละเอียดทั้งหมดจึงเป็นเรื่องเหลือวิสัย

1.1.4 การทวนสอบรูปนัยโดยเทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์

สำหรับการทวนสอบโดยเทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์มักใช้ทวนสอบวงจรควบคุม (control circuit) (Mir, Balakrishnan และ Tahar, 2000; Choi และคณะ, 2001; Goel และ Lee, 2000), ทวนสอบโพรโทคอล (protocol) (Barakatain และคณะ, 2001) หรือวงจรควบคุมแบบอสมวาร (asynchronous control circuit) (Vakilotojar และ Beerel, 1997) ถึงแม้ว่าจะมีการประยุกต์ใช้วิธีการในกลุ่มนี้เพื่อทวนสอบหน่วยประมวลผล (Jhala และ McMillan, 2001) แต่ก็เป็นการทวนสอบในระดับพฤติกรรม (behavior level) ซึ่งไม่สามารถนำไปสังเคราะห์วงจรได้ จากที่กล่าวมาจะเห็นได้ว่ายังไม่มีงานวิจัยมากนักที่ประยุกต์ใช้เทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์เพื่อทวนสอบหน่วยประมวลผล และไม่มีงานวิจัยมากนักที่จะทวนสอบรายละเอียดทั้งหมดของหน่วยประมวลผล นอกจากนั้นยังไม่เคยมีผู้ใดใช้วิธีการนี้เพื่อทวนสอบหน่วยประมวลผลในระดับถ่ายโอนเรจิสเตอร์มาก่อน ซึ่งอาจเป็นเพราะว่าระดับถ่ายโอนเรจิสเตอร์เป็นระดับที่มีรายละเอียดมากจนทำให้ในหลายวงจรไม่สามารถทำการทวนสอบได้ด้วยเทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์

ดังนั้นในวิทยานิพนธ์นี้จึงตั้งใจที่จะใช้เทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์เพื่อทวนสอบรายละเอียดของหน่วยประมวลผลทั้งหมดในระดับถ่ายโอนเรจิสเตอร์ ซึ่งเป็นระดับที่มีรายละเอียดมากพอที่จะนำไปสังเคราะห์วงจรได้ โดยในวิทยานิพนธ์นี้เลือกที่จะทวนสอบหน่วยประมวลผลที่มีความซับซ้อนไม่มากนักซึ่งสามารถทวนสอบรายละเอียดทั้งหมดในระดับถ่ายโอนเรจิสเตอร์ได้ ข้อดีของแนวทางนี้คือ ความอัตโนมัติของวิธีดังกล่าวและการที่ไม่ต้องใช้ผู้เชี่ยวชาญทางคณิตศาสตร์เพื่อทวนสอบทำให้กลุ่มวิจัยหรือบริษัทเอกชนขนาดเล็กสามารถนำวิธีการนี้ไปใช้ทวนสอบวงจรของตนเองได้ ข้อดีอีกประการหนึ่งคือ การที่สามารถทำการทวนสอบในระดับถ่ายโอนเรจิสเตอร์ได้จะเป็นประโยชน์ต่อการนำไปใช้จริงในอุตสาหกรรม และในวิทยานิพนธ์นี้ยังได้นำเสนอวิธีทวนสอบแบบลำดับขั้นซึ่งแบ่งการทวนสอบเป็นหลายขั้นตอนโดยแต่ละขั้นตอนจะมีรายละเอียดเพิ่มขึ้นจากขั้นตอนแรกไปจนถึงขั้นตอนสุดท้าย ซึ่งวิธีการนี้ช่วยทำให้การหาสาเหตุของข้อผิดพลาดได้ง่ายในวงจร นอกจากนั้นยังได้กล่าวถึงเทคนิคที่ช่วยทำให้สามารถทวนสอบได้เสร็จในเวลาที่สมเหตุผล

1.2 วัตถุประสงค์ของการวิจัย

1. ทำการทวนสอบหน่วยประมวลผลที่ถูกออกแบบในระดับถ่ายโอนเรจิสเตอร์โดยใช้วิธีการทวนสอบรูปนัย
2. หาวิธีการที่ใช้แก้ปัญหาการเพิ่มอย่างรวดเร็วของสถานะซึ่งเป็นอุปสรรคเมื่อใช้เทคนิคการตรวจสอบแบบจำลองสัญลักษณ์เพื่อทวนสอบหน่วยประมวลผลที่ถูกออกแบบในระดับถ่ายโอนเรจิสเตอร์

1.3 ขอบเขตของการวิจัย

1. ในการวิจัยนี้เป็นการประยุกต์โปรแกรมคาเดนค์เอสเอมวีซึ่งเป็นเครื่องมือช่วยทวนสอบที่มีผู้พัฒนาไว้แล้ว เพื่อทวนสอบหน่วยประมวลผล
2. การวิจัยนี้ทำการทวนสอบหน่วยประมวลผลที่ถูกออกแบบในระดับถ่ายโอนเรจิสเตอร์เพื่อใช้ในระบบเว็บเซิร์ฟเวอร์แบบฝังตัว โดยที่หน่วยประมวลผลนี้ถูกสร้างขึ้นเองภายในภาควิชาวิศวกรรมคอมพิวเตอร์
3. การทวนสอบในการวิจัยนี้เป็นการทวนสอบว่า รายละเอียดการออกแบบของหน่วยประมวลผลสามารถทำงานได้สอดคล้องกับข้อกำหนดวงจร ซึ่งอธิบายสถาปัตยกรรมชุดคำสั่งของหน่วยประมวลผลนั้น

1.4 ประโยชน์ที่ได้รับ

1. ช่วยหาข้อผิดพลาดที่อาจมีอยู่ภายในหน่วยประมวลผลหลังจากการทวนสอบด้วยวิธีการจำลองการทำงาน
2. ช่วยเพิ่มความมั่นใจของหน่วยประมวลผลที่ถูกทวนสอบแล้ว
3. แสดงตัวอย่างการประยุกต์โปรแกรมคาเดนค์เอสเอมวี เพื่อใช้ทวนสอบหน่วยประมวลผลที่ถูกออกแบบในระดับถ่ายโอนเรจิสเตอร์ ซึ่งยังไม่เคยมีผู้อื่นทำมาก่อน

4. พัฒนาเทคนิคต่างๆที่ช่วยลดความซับซ้อนของวงจร เพื่อให้โปรแกรมคาเดนค์เอสเอมวีสามารถทวนสอบหน่วยประมวลผลที่ถูกออกแบบในระดับถ่ายโอนเรจิสเตอร์ได้

1.5 ขั้นตอนการดำเนินการวิจัย

1. ศึกษางานวิจัยในกลุ่มการทวนสอบรูปนัย และหาเครื่องมือช่วยทวนสอบที่เหมาะสมกับการทวนสอบหน่วยประมวลผลที่ถูกออกแบบในระดับถ่ายโอนเรจิสเตอร์
2. ศึกษาการใช้งานโปรแกรมคาเดนค์เอสเอมวี
3. ทำการทวนสอบหน่วยประมวลผลด้วยโปรแกรมคาเดนค์เอสเอมวี
4. เขียนบทความวิชาการเพื่อพิจารณานำเสนอในการประชุมวิชาการ
5. ทำการพิสูจน์ว่าความถูกต้องของการทวนสอบรูปนัยของหน่วยประมวลผลมีความน่าเชื่อถือมากเพียงพอ
6. เขียนวิทยานิพนธ์ และดำเนินการสอบวิทยานิพนธ์

1.6 ผลงานตีพิมพ์จากงานวิจัย

งานวิจัยจากวิทยานิพนธ์นี้ได้ถูกนำไปเขียนบทความวิชาการชื่อ "RTL Formal Verification of Embedded Processors" ซึ่งได้ร่วมนำเสนอในลักษณะ Oral Presentation ในการประชุมวิชาการนานาชาติ ICIT'2002 (2002 IEEE International Conference on Industrial Technology) ซึ่งจัดที่กรุงเทพมหานคร ระหว่างวันที่ 11-14 ธันวาคม 2545

1.7 เนื้อหาและรูปแบบการนำเสนอวิทยานิพนธ์

เนื้อหาต่อจากบทนี้เป็นดังนี้ บทที่ 2 กล่าวถึงงานวิจัยที่เกี่ยวข้อง บทที่ 3 อธิบายรายละเอียดทางทฤษฎีของการตรวจสอบแบบจำลองเชิงสัญลักษณ์ บทที่4 นำเสนอรายละเอียดของหน่วยประมวลผลแบบฝังตัวที่ถูกทวนสอบ บทที่ 5 นำเสนอรายละเอียดการทวนสอบรูปนัยของหน่วยประมวลผล บทที่ 6 นำเสนอผลการทวนสอบ และบทที่ 7 สรุปผลการวิจัย

บทที่ 2

งานวิจัยที่เกี่ยวข้อง

2.1 การทวนสอบรูปนัยของหน่วยประมวลผล

ในวิทยานิพนธ์นี้จะนำเสนอการทวนสอบรูปนัยของหน่วยประมวลผล (processor) ที่ถูกออกแบบเพื่อใช้ในระบบเว็บเซิร์ฟเวอร์แบบฝังตัว (embedded web server) (Piromsopa, 2000) โดยที่การทวนสอบจะกระทำกับหน่วยประมวลผลที่ถูกออกแบบในระดับถ่ายโอนเรจิสเตอร์ (register transfer level) ซึ่งเป็นระดับที่สามารถนำไปสังเคราะห์วงจรได้ ในการทวนสอบจะทำโดยใช้โปรแกรมคาเดนค์เอสเอมวี (Cadence SMV) (McMillan, 1998) ซึ่งทำงานโดยใช้เทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์ (symbolic model checking) (McMillan, 1992 a.; Burch และคณะ, 1992) ซึ่งเป็นเทคนิคหนึ่งในกลุ่มวิธีที่ใช้พื้นฐานของแบบจำลอง

หลายปีมานี้มีการใช้วิธีการทวนสอบรูปนัยเพื่อตรวจสอบความถูกต้องของหน่วยประมวลผลในหลายงานวิจัยได้แก่

- Brock และ Hunt (1991) ได้นำเสนอบทความที่สรุปเกี่ยวกับการทวนสอบรูปนัยของหน่วยประมวลผลไวเปอร์ (VIPER - Verifiable Integrated Processor for Enhanced Reliability) ซึ่งถูกทวนสอบโดยนักวิจัยหลายกลุ่ม หน่วยประมวลผลไวเปอร์นี้เป็นหน่วยประมวลผลที่ถูกออกแบบเพื่อใช้ในงานที่ความถูกต้องของการทำงานเป็นเรื่องสำคัญมาก และถือได้ว่าเป็นหน่วยประมวลผลตัวแรกซึ่งสร้างขึ้นในเชิงการค้าและได้รับการทวนสอบด้วยวิธีการทวนสอบรูปนัย หน่วยประมวลผลตัวนี้เป็นหน่วยประมวลผลขนาด 32 บิต ทำงานโดยไม่ใช้เทคนิคการทำงานแบบสายท่อ (pipeline technique) และไม่รองรับสัญญาณขัดจังหวะ (interruption signal) แต่เนื่องจากเป็นหน่วยประมวลผลที่ออกแบบเพื่อใช้ในงานที่ความถูกต้องของการทำงานเป็นเรื่องสำคัญมาก จึงมีคุณสมบัติที่สามารถหยุดการทำงานและส่งสัญญาณเตือนเมื่อเกิดข้อผิดพลาดของการทำงานภายในหน่วยประมวลผล หน่วยประมวลผลไวเปอร์ถูกทวนสอบโดยนักวิจัยหลายกลุ่มโดยใช้หลายวิธีการ วิธีการทวนสอบที่สำคัญของหน่วยประมวลผลตัวนี้ได้แก่วิธีที่สร้างขึ้นบนพื้นฐานของวิธีการพิสูจน์ทางทฤษฎีที่ชื่อว่า เฮชโอแอล (HOL - Higher Order Logic) (Gordon, 1988) ในบทความนี้แสดงให้เห็นว่างานวิจัยที่ทำการทวนสอบทั้งหมดในเวลานั้นของหน่วยประมวลผลไวเปอร์ ยังมีจุดซึ่งยังไม่สมบูรณ์ในการรับประกัน

ว่ารายละเอียดการออกแบบในระดับเกต (gate level) ทำงานได้สอดคล้องกับข้อกำหนดของวงจรที่อธิบายสถาปัตยกรรมชุดคำสั่ง (Instruction Set Architecture - ISA)

- Graham (1992) ได้นำเสนอการทวนสอบรูปนัยของหน่วยประมวลผลเอสอีซีดี (SECD) ซึ่งเป็นหน่วยประมวลผลขนาด 32 บิตแบบสแต็ก (stack processor) ที่ถูกออกแบบโดยใช้ไมโครโปรแกรม (microprogram) เพื่อทำงานกับภาษาโปรแกรมเชิงหน้าที่ (functional programming language) กระบวนการทวนสอบของบทความนี้จะทำการทวนสอบ 2 ขั้นตอน คือ ขั้นตอนแรกทวนสอบว่ารายละเอียดการออกแบบในระดับถ่ายโอนเรจิสเตอร์มีความสอดคล้องกับข้อกำหนดของวงจรซึ่งอธิบายสถาปัตยกรรมชุดคำสั่ง และในขั้นตอนที่สองทำการทวนสอบว่ารายละเอียดการออกแบบในระดับเกตมีความสอดคล้องกับข้อกำหนดของวงจรในระดับถ่ายโอนเรจิสเตอร์ การทวนสอบในบทความนี้ใช้วิธีการที่ชื่อเฮชโอแอลเช่นเดียวกับ (Brock และ Hunt, 1991) ในช่วงเวลานั้นหน่วยประมวลผลเอสอีซีดีถือได้ว่าเป็นหน่วยประมวลผลที่ใหญ่ที่สุดซึ่งถูกทวนสอบด้วยวิธีการทวนสอบรูปนัย

- วิทยานิพนธ์ของ Beatty (1993) ได้นำเสนอระเบียบวิธี (methodology) ที่เหมาะสมกับการทวนสอบรูปนัยของหน่วยประมวลผล และในวิทยานิพนธ์ดังกล่าวได้นำระเบียบวิธีที่นำเสนอไปใช้ทวนสอบหน่วยประมวลผลเฮ็คเตอร์ (Hector) หน่วยประมวลผลนี้เป็นหน่วยประมวลผลที่ถูกออกแบบไว้แล้วก่อนที่จะนำมาทวนสอบรูปนัย ซึ่งเป็นการแสดงว่าระเบียบวิธีที่นำเสนอในวิทยานิพนธ์ดังกล่าวสามารถใช้ทวนสอบหน่วยประมวลผลที่ไม่ได้ออกแบบให้เหมาะสมกับการทวนสอบรูปนัยได้ ซึ่งต่างจากหน่วยประมวลผลไวเปอร์ (Brock และ Hunt, 1991) และหน่วยประมวลผลเอสอีซีดี (Graham, 1992) ที่ถูกออกแบบเพื่อให้ง่ายต่อการทวนสอบ หน่วยประมวลผลเฮ็คเตอร์เป็นหน่วยประมวลผลขนาด 16 บิต มีสถาปัตยกรรมชุดคำสั่งคล้ายกับหน่วยประมวลผลพีดีพีสิบเอ็ด (PDP-11) ทำงานโดยไม่ใช้เทคนิคการทำงานแบบสายท่อ แต่สามารถรองรับสัญญาณขัดจังหวะได้ การทวนสอบรูปนัยของหน่วยประมวลผลเฮ็คเตอร์เป็นการพิสูจน์ว่ารายละเอียดการออกแบบในระดับทรานซิสเตอร์ (transistor level) สอดคล้องกับข้อกำหนดวงจรซึ่งอธิบายสถาปัตยกรรมชุดคำสั่ง ระเบียบวิธีที่ใช้ทวนสอบจะทำการกำหนดข้อความยืนยัน (assertion) เพื่ออธิบายสถาปัตยกรรมชุดคำสั่งในลักษณะเครื่องสถานะจำกัด และกำหนดความสัมพันธ์ระหว่างสถานะภายในเครื่องสถานะจำกัดกับวงจรในระดับทรานซิสเตอร์ และใช้การจำลองการทำงานเชิงสัญลักษณ์ (symbolic simulation) (Cory, 1983) เพื่อทดสอบว่าวงจรในระดับทรานซิสเตอร์สามารถทำงานได้สอดคล้องกับสถาปัตยกรรมชุดคำสั่ง วิธีการจำลองการทำงานเชิงสัญลักษณ์ถือว่าเป็นวิธีหนึ่งในกลุ่มวิธีที่ใช้พื้นฐานของแบบจำลอง

- Appenzeller และ Kuehlmann (1995) ได้นำเสนอการทวนสอบรูปนัยของหน่วยประมวลผลเพาเวอร์พีซี (PowerPC) ซึ่งเป็นหน่วยประมวลผลที่สร้างขึ้นในเชิงการค้า หน่วยประมวลผลเพาเวอร์พีซีเป็นหน่วยประมวลผลขนาด 32 บิตซึ่งทำงานโดยใช้เทคนิคการทำงานแบบสายท่อ ในบทความนี้ทำการทวนสอบว่ารายะเอียดการออกแบบในระดับทรานซิสเตอร์สามารถทำงานได้สอดคล้องกับข้อกำหนดของวงจรในระดับถ่ายโอนเรจิสเตอร์ การทวนสอบใช้วิธีการที่ใช้พื้นฐานของแผนภาพตัดสินใจทวิภาค (Binary Decision Diagram - BDD) (Bryant, 1986) โดยทำการทวนสอบว่าวงจรเชิงผสม (combinational circuit) ในระดับถ่ายโอนเรจิสเตอร์และระดับทรานซิสเตอร์มีความสมมูลกัน ในกระบวนการทวนสอบได้แบ่งหน่วยประมวลผลออกเป็นหลายส่วนโดยแต่ละส่วนจะมีจำนวนทรานซิสเตอร์ไม่เกิน 25,000 ตัว ซึ่งเป็นขนาดของวงจรที่โปรแกรมช่วยทวนสอบของบทความนี้สามารถทำงานได้

- Miller และ Srivas (1995) ได้นำเสนอการทวนสอบรูปนัยของหน่วยประมวลผลเอเอเอมพีห้า (AAMP5) ซึ่งเป็นหน่วยประมวลผลที่ถูกออกแบบเพื่อใช้ในงานที่ความถูกต้องของการทำงานเป็นเรื่องสำคัญมาก เช่นในระบบควบคุมการจราจรทางอากาศ เป็นต้น หน่วยประมวลผลเอเอเอมพีห้า เป็นหน่วยประมวลผลขนาด 32 บิตแบบสแต็ก ซึ่งทำงานโดยใช้เทคนิคการทำงานแบบสายท่อ และสามารถรองรับสัญญาณขัดจังหวะ หน่วยประมวลผลนี้เป็นหน่วยประมวลผลแบบซีไอเอสซี (CISC - Complex Instruction Set Computer) มีคำสั่ง 208 คำสั่ง ถูกออกแบบโดยใช้ไมโครโปรแกรม และเนื่องจากเป็นหน่วยประมวลผลที่ออกแบบเพื่อใช้ในงานที่ความถูกต้องของการทำงานเป็นเรื่องสำคัญมาก หน่วยประมวลผลนี้จึงมีคุณสมบัติที่สามารถตรวจสอบและจัดการข้อผิดพลาดของการทำงานภายในหน่วยประมวลผลได้ การทวนสอบรูปนัยของบทความนี้เป็นการพิสูจน์ว่ารายละเอียดการออกแบบในระดับถ่ายโอนเรจิสเตอร์สามารถทำงานสอดคล้องกับข้อกำหนดของวงจรซึ่งอธิบายสถาปัตยกรรมชุดคำสั่ง ในกระบวนการทวนสอบใช้โปรแกรมชื่อพีวีเอส (PVS - Prototype Verification System) (Owre, Rushby และ Shankar, 1992) ซึ่งทำงานโดยวิธีการพิสูจน์ทางทฤษฎี

- Windley (1995) ได้นำเสนอระเบียบวิธีของการทวนสอบรูปนัยสำหรับหน่วยประมวลผล โดยระเบียบวิธีสร้างขึ้นบนพื้นฐานของวิธีการพิสูจน์ทางทฤษฎีที่ชื่อว่าเฮชโอแอลเช่นเดียวกับ (Brock และ Hunt, 1991) และ (Graham, 1992) ในบทความนี้นำระเบียบวิธีที่นำเสนอไปใช้ทวนสอบหน่วยประมวลผลเอวีเอมหนึ่ง (AVM-1) ซึ่งเป็นหน่วยประมวลผลที่ถูกออกแบบเพื่อใช้แสดงตัวอย่างของการทวนสอบ หน่วยประมวลผลนี้เป็นหน่วยประมวลผลขนาด 32 บิตแบบอาร์ไอเอสซี (RISC - Reduced Instruction Set Computer) ทำงานโดยใช้เทคนิคการทำงานแบบสายท่อ สามารถรองรับสัญญาณขัดจังหวะ มีหน่วยจัดการหน่วยความจำ

(memory management unit) อยู่ภายในหน่วยประมวลผล และสามารถทำการคำนวณแบบจุดลอยตัว (floating point) ได้ ในการทวนสอบทำการพิสูจน์ว่ารายละเอียดการออกแบบระดับถ่ายโอนเรจิสเตอร์ทำงานได้สอดคล้องกับข้อกำหนดของวงจรซึ่งอธิบายสถาปัตยกรรมชุดคำสั่ง โดยระเบียบวิธีที่นำเสนอคือการสร้างแบบจำลองของหน่วยประมวลผลขึ้นหลายแบบจำลองที่มีความซับซ้อนมากขึ้นในแต่ละลำดับชั้นระหว่างระดับถ่ายโอนเรจิสเตอร์และสถาปัตยกรรมชุดคำสั่ง และทำการทวนสอบว่าแบบจำลองแต่ละลำดับชั้นที่อยู่ติดกันมีความสอดคล้องกันจึงจะสามารถสรุปได้ว่ารายละเอียดการออกแบบสอดคล้องกับข้อกำหนดของวงจร

- Bundgen, Kunchlin และ Shankar (1996) ได้นำเสนอการทวนสอบรูปนัยของหน่วยประมวลผลสปาร์โรว (Sparrow) หน่วยประมวลผลนี้เป็นหน่วยประมวลผลขนาด 8 บิต ถูกออกแบบเพื่อใช้กับเทคโนโลยีเอฟพีจีเอ (FPGA - Field Programmable Gate Array) และถูกสร้างขึ้นเพื่อจุดประสงค์ทางการศึกษา การทวนสอบในบทความนี้เป็นการพิสูจน์ว่ารายละเอียดการออกแบบในระดับเกตสอดคล้องกับข้อกำหนดของวงจรซึ่งอธิบายสถาปัตยกรรมชุดคำสั่ง การทวนสอบในบทความนี้ใช้เครื่องมือชื่อ รีดูเอ็ก (ReDuX) (Bundgen, 1993) ซึ่งทำงานโดยใช้วิธีการในกลุ่มวิธีการพิสูจน์ทางทฤษฎี

- Brock และ Hunt (1997) ได้นำเสนอการทวนสอบรูปนัยของหน่วยประมวลผลซีเอพี (CAP - Complex Arithmetic Processor) ซึ่งถูกสร้างโดยบริษัทโมโตโรล่า (Motorola) หน่วยประมวลผลซีเอพีเป็นหน่วยประมวลผลประเภทดีเอสพี (DSP - Digital Signal Processor) ทำงานโดยใช้เทคนิคการทำงานแบบซูเปอร์สเกลาร์ (super-scalar technique) การทวนสอบในบทความนี้เป็นการพิสูจน์ว่ารายละเอียดการออกแบบในระดับถ่ายโอนเรจิสเตอร์สอดคล้องกับข้อกำหนดของวงจรซึ่งอธิบายสถาปัตยกรรมชุดคำสั่ง การทวนสอบใช้เครื่องมือชื่อ เอซีแอลทู (ACL2 - A Computational Logic for Application Common Lisp) (Kaufmann และ Moore, 1996) ซึ่งทำงานโดยใช้วิธีการในกลุ่มวิธีการพิสูจน์ทางทฤษฎี

- Huggins และ Campenhout (1998) ได้นำเสนอการทวนสอบรูปนัยของหน่วยประมวลผลเออาร์เอมสอง (ARM2) ซึ่งเป็นหน่วยประมวลผลที่สร้างขึ้นในเชิงการค้า โดยการทวนสอบเป็นการพิสูจน์ว่ารายละเอียดการออกแบบในระดับถ่ายโอนเรจิสเตอร์ซึ่งทำงานโดยใช้เทคนิคการทำงานแบบสายท่อสามารถทำงานได้สอดคล้องกับข้อกำหนดของวงจรซึ่งอธิบายสถาปัตยกรรมชุดคำสั่งของหน่วยประมวลผล โดยการทวนสอบใช้เทคนิคของเครื่องสถานะสาระสำคัญ (abstract state machine) (Gurevich, 1993) ซึ่งเป็นหนึ่งในเทคนิคของกลุ่มวิธีที่

ใช้พื้นฐานของแบบจำลอง การทวนสอบโดยเทคนิคนี้ทำได้โดยการสร้างแบบจำลองของหน่วยประมวลผลขึ้นหลายแบบจำลองซึ่งมีรายละเอียดมากขึ้นในแต่ละลำดับชั้น และทำการพิสูจน์ว่าแบบจำลองในลำดับชั้นที่อยู่ติดกันมีความสอดคล้องกัน เมื่อทำการพิสูจน์ได้ว่าทุกลำดับชั้นที่ติดกันมีความสอดคล้องกัน ก็จะสามารถสรุปได้ว่า แบบจำลองที่รายละเอียดน้อยสุด (ข้อกำหนดของวงจรซึ่งอธิบายสถาปัตยกรรมชุดคำสั่ง) กับแบบจำลองที่มีรายละเอียดมากที่สุด (รายละเอียดการออกแบบในระดับถ่ายโอนเรจิสเตอร์ซึ่งทำงานโดยใช้เทคนิคการทำงานแบบสายท่อ) สามารถทำงานได้สอดคล้องกันด้วย

- Patankar, Jain และ Bryant (1999) ได้นำเสนอการทวนสอบรูปนัยของหน่วยประมวลผลซึ่งเป็นลูกผสมระหว่างหน่วยประมวลผลเออาร์เอมเจ็ด (ARM7) และหน่วยประมวลผลสตรองเออาร์เอม (StrongARM) หน่วยประมวลผลนี้เป็นหน่วยประมวลผลขนาด 32 บิต ทำงานโดยใช้เทคนิคการทำงานแบบสายท่อ การทวนสอบในบทความนี้เป็นการพิสูจน์ว่ารายละเอียดการออกแบบในระดับเกตสอดคล้องกับข้อกำหนดของวงจรซึ่งอธิบายสถาปัตยกรรมชุดคำสั่ง วิธีการทวนสอบที่ใช้ในบทความนี้ชื่อ เอสทีอี (STE - Symbolic Trajectory Evaluation) (Bryant, Beatty และ Seger, 1991) ซึ่งเป็นวิธีการหนึ่งในกลุ่มวิธีที่ใช้พื้นฐานของแบบจำลอง วิธีการนี้ถูกพัฒนาต่อเนื่องจากวิธีการจำลองการทำงานเชิงสัญลักษณ์ (Cory, 1983) โดย Beatty (1993) ได้ใช้วิธีการจำลองการทำงานเชิงสัญลักษณ์เพื่อทวนสอบหน่วยประมวลผลเฮ็คเตอร์ ซึ่งได้กล่าวถึงแล้วก่อนหน้านี้

จากที่กล่าวมาทั้งหมดจะเห็นได้ว่า เมื่อเวลาผ่านไปวิธีการและเครื่องมือที่ใช้ทวนสอบรูปนัยมีความสามารถที่เพิ่มขึ้น ทำให้สามารถทวนสอบหน่วยประมวลผลที่มีความซับซ้อนได้มากขึ้น นอกจากนั้นยังมีข้อสังเกตอีกประการหนึ่งคือ ส่วนใหญ่แล้วมักใช้เทคนิคในกลุ่มวิธีการพิสูจน์ทางทฤษฎีเพื่อพิสูจน์ความถูกต้องของหน่วยประมวลผล ที่เป็นเช่นนี้เนื่องจากวิธีดังกล่าวสามารถใช้กับวงจรที่มีขนาดใหญ่มาก แต่เนื่องจากวิธีดังกล่าวไม่สามารถทำงานได้อย่างอัตโนมัติ และผู้ทวนสอบต้องมีความเชี่ยวชาญในทางตรรกแบบประพจน์ ทำให้งานวิจัยเหล่านี้มักทำในมหาวิทยาลัยหรือบริษัทเอกชนขนาดใหญ่ที่มีผู้เชี่ยวชาญเฉพาะด้านเพื่อทำการทวนสอบโดยเฉพาะ

ตารางที่ 2.1 สรุปข้อมูลการทวนสอบรูปนัยของหน่วยประมวลผล

ปี	หน่วยประมวลผล	วิธีการทวนสอบ *	Imp.<->Spec.
1991	VIPER	HOL(T)	Gate<->ISA
1992	SCED	HOL(T)	Gate<->ISA
1993	Hector	Symbolic Simulation(M)	Transistor<->ISA
1995	PowerPC	Combinational Equivalence Checking (M)	Transistor<->RTL
1995	AAMP5	PVS(T)	RTL<->ISA
1995	AVM-1	HOL(T)	RTL<->ISA
1996	Sparrow	ReDuX(T)	Gate<->ISA
1997	CAP	ACL2(T)	RTL<->ISA
1998	ARM2	Abstract State Machine(M)	RTL<->ISA
1999	ARM7	STE(M)	Gate<->ISA

* หมายเหตุ (M) หมายถึงวิธีการในกลุ่มวิธีที่ใช้พื้นฐานของแบบจำลอง
(T) หมายถึงวิธีการในกลุ่มวิธีการพิสูจน์ทางทฤษฎี

2.2 การทวนสอบรูปนัยโดยเทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์

เทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์ถูกนำไปใช้ทำการทวนสอบรูปนัยในหลายงานวิจัยซึ่งได้แก่

- Vakilotojar และ Beerel (1997) ได้นำเสนอระเบียบวิธีสำหรับใช้ทวนสอบรูปนัยวงจรแบบอสมวาร (asynchronous circuit) โดยใช้เทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์ ในบทความนี้สนใจเฉพาะวงจรควบคุมภายในวงจรอสมวารเท่านั้น และการทวนสอบในบทความนี้จะเป็นการตรวจสอบคุณสมบัติที่สำคัญของวงจร เช่น คุณสมบัติความปลอดภัย (safety property) เป็นต้น ระเบียบวิธีที่นำเสนอนั้นแบ่งการทำงานเป็น 2 ขั้นตอน ขั้นตอนแรกจะแปลงจากวงจรอสมวารให้กลายเป็นเครื่องสถานะจำกัดแบบสมวาร (synchronous finite state machine) ซึ่งอธิบายพฤติกรรมของวงจร ขั้นตอนต่อมาจึงนำเครื่องสถานะจำกัดแบบสมวารที่ได้ไปทวนสอบด้วยโปรแกรมเอสเอมวี (SMV) (McMillan, 1992 b.) ซึ่งโปรแกรมนี้ในภายหลัง

ถูกพัฒนาต่อเป็นโปรแกรมคาเดนด์เอสเอมวี สาเหตุที่ต้องแปลงวงจรอสมวารให้เป็นเครื่องสถานะจำกัดแบบสมวารเนื่องจากโปรแกรมเอสเอมวีเหมาะสมเฉพาะกับวงจรสมวารเท่านั้น

- Schonherr และคณะ (1999) ได้นำเทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์ไปใช้ทวนสอบรูปนัยหน่วยประมวลผลซึ่งทำงานโดยใช้เทคนิคการทำงานแบบสายท่อ การทวนสอบในบทความนี้จะเป็นการตรวจสอบภัยแบบควบคุม (control hazard checking) ที่อาจเกิดขึ้นภายในหน่วยประมวลผล การทวนสอบจะแบ่งเป็น 2 ขั้นตอน ขั้นตอนแรกจะแปลงหน่วยประมวลผลซึ่งถูกออกแบบในระดับถ่ายโอนเรจิสเตอร์ ให้เป็นเครื่องสถานะจำกัดที่อธิบายพฤติกรรมของหน่วยประมวลผล หลังจากนั้นจึงใช้โปรแกรมเวอร์ดีส (VERDIS) (Straube และคณะ, 1996) ซึ่งทำงานโดยใช้เทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์ ทำการตรวจสอบเครื่องสถานะจำกัดที่ถูกแปลงแล้ว

- Goel และ Lee (2000) ได้นำเสนอการทวนสอบรูปนัยของวงจรควบคุมบัสซึ่งถูกออกแบบเพื่อใช้ในระบบบนชิปเดียว (system-on-a-chip) วงจรนี้มีหน้าที่ควบคุมการติดต่อสื่อสารผ่านบัสระหว่างแต่ละวงจรย่อยภายในระบบ วงจรควบคุมบัสดังกล่าวถูกพัฒนาโดยบริษัทไอบีเอ็ม การทวนสอบในบทความนี้เป็นการตรวจสอบว่าวงจรควบคุมบัสซึ่งถูกออกแบบในระดับถ่ายโอนเรจิสเตอร์มีคุณสมบัติตามที่ต้องการหรือไม่ และการทวนสอบในบทความนี้จะทำการทวนสอบเฉพาะวงจรควบคุมเท่านั้น เครื่องมือช่วยทวนสอบที่ใช้ชื่อ รูล์เบส (RuleBase) (Beer และคณะ, 1996) ซึ่งทำงานโดยใช้เทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์ โปรแกรมนี้ถูกพัฒนาต่อเนื่องจากโปรแกรมเอสเอมวี (McMillan, 1992 b.)

- Mir, Balakrishman และ Takar (2000) ได้นำเสนอการทวนสอบรูปนัยของซอฟต์แวร์แบบฝังตัว (embedded software) โดยการทวนสอบใช้โปรแกรมคาเดนค์เอสเอมวี (McMillan, 1998) ซึ่งเป็นโปรแกรมที่ใช้ช่วยทวนสอบในวิทยานิพนธ์นี้ ในบทความนี้ทำการทวนสอบโปรแกรมควบคุมการทำงานของเมาส์ผ่านพอร์ตอนุกรม (serial port) ซึ่งพัฒนาบนหน่วยประมวลผลพีไอซี (PIC) การทวนสอบจะรวมรายละเอียดของโปรแกรมควบคุมกับสถาปัตยกรรมชุดคำสั่งของหน่วยประมวลผลพิคเข้าด้วยกันเพื่อใช้ในการทวนสอบ การทวนสอบในบทความนี้เป็นการตรวจสอบคุณสมบัติว่าโปรแกรมควบคุมมีคุณสมบัติที่ต้องการหรือไม่

- Choi และคณะ (2001) ได้นำเสนอการทวนสอบรูปนัยของระบบบนชิปเดียว ซึ่งในบทความนี้ทำการทวนสอบวงจรดีเอ็มเอ (DMA - Direct Memory Access) และวงจรควบคุมยูเอสบี

(USB controller) ภายในระบบดังกล่าว การทวนสอบทำโดยใช้โปรแกรมคาเดนค์เอสเอมวี (McMillan, 1998) ซึ่งเป็นโปรแกรมที่ใช้ช่วยทวนสอบในวิทยานิพนธ์นี้ การทวนสอบในบทความนี้เป็นการทวนสอบว่าวงจรซึ่งถูกออกแบบในระดับถ่ายโอนเรจิสเตอร์มีคุณสมบัติตามที่ต้องการหรือไม่ และในงานวิจัยนี้ทำการทวนสอบเฉพาะส่วนที่เป็นวงจรควบคุมเท่านั้น

จากที่กล่าวมาทั้งหมดจะเห็นได้ว่าเทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์มักใช้ทวนสอบวงจรควบคุม ถึงแม้ว่าจะมีการประยุกต์ใช้เทคนิคนี้เพื่อทวนสอบหน่วยประมวลผลบ้าง แต่ก็เป็นการทวนสอบในระดับพฤติกรรมซึ่งไม่สามารถนำไปสังเคราะห์วงจรได้ จากที่กล่าวมาจะเห็นได้ว่ายังไม่มีงานวิจัยมากนักที่ประยุกต์เทคนิคการจำลองการทำงานเชิงสัญลักษณ์เพื่อทวนสอบหน่วยประมวลผล นอกจากนั้นยังไม่เคยมีผู้ใดใช้วิธีการนี้เพื่อทวนสอบหน่วยประมวลผลในระดับถ่ายโอนเรจิสเตอร์มาก่อน ซึ่งอาจเป็นเพราะว่าระดับถ่ายโอนเรจิสเตอร์เป็นระดับที่มีรายละเอียดมากจนทำให้ในหลายวงจรไม่สามารถทำการทวนสอบได้ด้วยเทคนิคการจำลองการทำงานเชิงสัญลักษณ์

ดังนั้นในวิทยานิพนธ์นี้จึงตั้งใจที่จะใช้เทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์เพื่อทวนสอบรายละเอียดของหน่วยประมวลผลทั้งหมดในระดับถ่ายโอนเรจิสเตอร์ ซึ่งเป็นระดับที่มีรายละเอียดมากพอที่จะนำไปสังเคราะห์วงจรได้ โดยในวิทยานิพนธ์นี้เลือกที่จะทวนสอบหน่วยประมวลผลที่มีความซับซ้อนไม่มากนักซึ่งสามารถทวนสอบรายละเอียดทั้งหมดในระดับถ่ายโอนเรจิสเตอร์ได้ ข้อดีของแนวทางนี้คือ ความอัตโนมัติของวิธีดังกล่าวและการที่ไม่ต้องใช้ผู้เชี่ยวชาญทางคณิตศาสตร์เพื่อทวนสอบทำให้กลุ่มวิจัยหรือบริษัทเอกชนขนาดเล็กสามารถนำวิธีการนี้ไปใช้ทวนสอบวงจรของตนเองได้ ข้อดีอีกประการหนึ่งคือ การที่สามารถทำการทวนสอบในระดับถ่ายโอนเรจิสเตอร์ได้จะเป็นประโยชน์ต่อการนำไปใช้จริงในอุตสาหกรรม และในวิทยานิพนธ์นี้ยังได้นำเสนอวิธีทวนสอบแบบลำดับขั้นซึ่งแบ่งการทวนสอบเป็นหลายขั้นตอนโดยแต่ละขั้นตอนจะมีรายละเอียดเพิ่มขึ้นจากขั้นตอนแรกไปจนถึงขั้นตอนสุดท้าย ซึ่งวิธีการนี้ช่วยทำให้การหาสาเหตุของข้อผิดพลาดได้ง่ายในวงจร นอกจากนั้นยังได้กล่าวถึงเทคนิคที่ช่วยทำให้สามารถทวนสอบได้เสร็จในเวลาที่สมเหตุผล

บทที่ 3

การตรวจสอบแบบจำลองเชิงสัญลักษณ์

วิธีการตรวจสอบแบบจำลองเชิงสัญลักษณ์ (symbolic model checking) (McMillan, 1992 a.; Clarke, Grumberg และ Peled, 2001) เป็นเทคนิคหนึ่งในกลุ่มวิธีที่ใช้พื้นฐานของแบบจำลอง โดยเทคนิคนี้เหมาะเป็นอย่างยิ่งที่จะใช้ทวนสอบระบบสถานะจำกัด (finite state system) เมื่อจะนำเทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์ไปใช้ทวนสอบวงจร งานที่ต้องทำสามารถแบ่งเป็น 3 ขั้นตอนคือ การสร้างแบบจำลอง (modeling), การสร้างข้อกำหนด (specification) และ การทวนสอบ (verification)

เนื้อหาที่นำเสนอในบทนี้เรียบเรียงจากหนังสือ “Model Checking” (Clarke, Grumberg และ Peled, 2001) และในบทนี้มีลำดับของเนื้อหาดังนี้ หัวข้อ 3.1 นำเสนอวิธีการจำลองแบบระบบโดยโครงสร้างคริปเก (kripke structure) และตรรกอันดับที่หนึ่ง โดยที่โครงสร้างคริปเกเป็นระบบการเปลี่ยนสถานะแบบหนึ่ง หัวข้อ 3.2 นำเสนอเกี่ยวกับตรรกเชิงเวลาซึ่งถูกใช้เพื่อสร้างข้อกำหนดสำหรับกระบวนการทวนสอบ หัวข้อ 3.3 นำเสนออัลกอริทึมของวิธีการตรวจสอบแบบจำลอง หัวข้อ 3.4 นำเสนอการใช้แผนภาพตัดสินใจทวิภาคเพื่อแทนโครงสร้างคริปเก หัวข้อ 3.5 นำเสนอรายละเอียดอัลกอริทึมของการตรวจสอบแบบจำลองเชิงสัญลักษณ์ และสุดท้ายหัวข้อ 3.6 นำเสนอเกี่ยวกับโปรแกรมคาเดนซ์เอสเอมวีซึ่งเป็นโปรแกรมช่วยทวนสอบที่วิทยานิพนธ์นี้ใช้

3.1 การจำลองแบบระบบ

ขั้นตอนแรกคือการจำลองแบบระบบ (modeling system) ซึ่งเป็นการแทนวงจรที่จะทวนสอบให้อยู่ในรูปแบบที่เครื่องมือช่วยทวนสอบสามารถเข้าใจได้ โดยในที่นี้จะแทนวงจรในลักษณะของโครงสร้างคริปเก (kripke structure) ซึ่งเป็นชนิดหนึ่งของระบบการเปลี่ยนสถานะ (state transition system) นอกจากนั้นในหัวข้อนี้ยังกล่าวถึงการแทนวงจรโดยใช้ตรรกอันดับที่หนึ่ง (first order logic) อีกด้วย

3.1.1 โครงสร้างคริปเก

ให้ P เป็นเซตของประพจน์เดี่ยว (atomic proposition)

โครงสร้างคริปเก M บนเซต P นิยามด้วย $M = (S, S_0, R, L)$ โดยที่

1. S คือเซตของสถานะ (state)

2. $S_0 \subseteq S$ คือเซตของสถานะเริ่มต้น

3. $R \subseteq S \times S$ คือความสัมพันธ์การเปลี่ยนสถานะ (state transition relation) ซึ่งต้องเป็นความสัมพันธ์แบบทั่วถึงในเซต S หรืออีกนัยหนึ่งหมายถึงว่าทุกสถานะ $s \in S$ จะต้องมี $s' \in S$ ซึ่งทำให้ความสัมพันธ์ $R(s, s')$ เป็นจริง

4. $L : S \rightarrow 2^P$ คือฟังก์ชันซึ่งกำหนดความสัมพันธ์ระหว่างสถานะกับเซตของประพจน์เดี่ยวที่เป็นจริงในสถานะนั้น

วิถี (path) ของโครงสร้าง M จากสถานะ s คือลำดับอนันต์ของสถานะ $\pi = s_0 s_1 s_2 \ldots$ โดยที่ $s_0 = s$ และความสัมพันธ์ $R(s_i, s_{i+1})$ เป็นจริงสำหรับทุกค่า $i \geq 0$

3.1.2 การแทนโครงสร้างคริปเกด้วยตรรกอันดับที่หนึ่ง

ให้ $V = \{v_1, \ldots, v_n\}$ เป็นเซตของตัวแปร (variable) ของระบบ โดยตัวแปรเหล่านี้มีค่าในช่วงของเซตจำกัด D

การแทนค่า (valuation) ของเซต V คือฟังก์ชันซึ่งกำหนดความสัมพันธ์ระหว่างตัวแปรในเซต V กับค่าภายในเซต D และการแทนค่าสามารถเขียนในรูปของสูตรทางตรรกอันดับที่หนึ่ง (first order logic formula) ได้

ตัวอย่างเช่น ให้ $V = \{v_1, v_2, v_3\}$ เราสามารถเขียนแทนการแทนค่า $\langle v_1 \leftarrow 2, v_2 \leftarrow 3, v_3 \leftarrow 5 \rangle$ ด้วยสูตรทางตรรก $(v_1 = 2) \wedge (v_2 = 3) \wedge (v_3 = 5)$ นอกจากนั้นสูตรทางตรรกหนึ่งสูตรยังสามารถใช้แทนการแทนค่าได้หลายชุด หรือกล่าวอีกนัยหนึ่งคือ สูตรทางตรรกสามารถใช้แทนเซตของการแทนค่าได้

สถานะของระบบสามารถถูกพิจารณาเป็นการแทนค่าของเซต V ได้ นั้นคือ $s : V \rightarrow D$ และเนื่องจากสูตรทางตรรกสามารถใช้แทนเซตของการแทนค่าได้ ดังนั้นจึงสามารถแทนเซตของสถานะได้ด้วยสูตรทางตรรก และในที่นี้ใช้ S_0 เพื่อแทนเซตของสถานะเริ่มต้น

นอกจากนั้นสูตรทางตรรกยังสามารถแทนความสัมพันธ์การเปลี่ยนสถานะได้ กำหนดให้มีตัวแปรสองชุดคือ V และ V' โดยที่ V เป็นชุดของตัวแปรสถานะปัจจุบัน และ V' เป็นชุดของตัวแปรสถานะถัดไป เราสามารถพิจารณาว่าการเปลี่ยนสถานะเป็นการแทนค่าคู่อันดับของตัวแปรซึ่งมาจากเซต V และ V' ได้ และเนื่องจากว่าสามารถแทนเซตของการแทนค่าด้วยสูตรทางตรรก ดังนั้นจึงสามารถแทนความสัมพันธ์การเปลี่ยนสถานะได้ด้วยสูตรทางตรรกเช่นกัน และในที่นี้ใช้ $R(V, V')$ แทนสูตรทางตรรกที่ใช้แทนความสัมพันธ์การเปลี่ยนสถานะ

สำหรับประพจน์เดี่ยวสามารถนิยามได้ในรูปของ $v = d$ ซึ่ง $v \in V$ และ $d \in D$, โดยที่ประพจน์ $v = d$ จะเป็นจริงในสถานะ s เมื่อ $s(v) = d$ และในที่นี้จะพิจารณาเฉพาะตัวแปรในโดเมนแบบบูล (boolean domain) ซึ่งประกอบด้วยค่าความจริงสองค่าคือ *True* และ *False* นอกจากนั้นเราจะใช้ v เพื่อแทน $s(v) = True$ และ $\neg v$ เพื่อแทน $s(v) = False$

ต่อไปจะแสดงถึงการสร้างโครงสร้างคริปเก $M = (S, S_0, R, L)$ จากสูตรทางตรรกอันดับที่หนึ่งโดยใช้สูตรทางตรรก S_0 และ R ซึ่งได้กล่าวถึงไว้แล้ว

- เซตของสถานะ S คือเซตของการแทนค่าของเซต V
- เซตของสถานะเริ่มต้น S_0 สามารถแทนได้ด้วยเซตของการแทนค่าของเซต V ซึ่งสอดคล้องกับสูตรทางตรรก S_0
- ให้ s และ s' คือสถานะของระบบ จะได้ว่าความสัมพันธ์ $R(s, s')$ เป็นจริงเมื่อสูตรทางตรรก $R(V, V')$ มีค่าความจริงเป็น *True* เมื่อแต่ละตัวแปร $v \in V$ ถูกแทนค่าด้วย $s(v)$ และแต่ละตัวแปร $v' \in V'$ ถูกแทนค่าด้วย $s'(v')$
- ฟังก์ชัน $L : S \rightarrow 2^P$ โดยที่ $L(s)$ เป็นเซตของประพจน์ซึ่งเป็นจริงในสถานะ s ซึ่ง $v \in L(s)$ หมายถึงว่า $s(v) = True$ และ $v \notin L(s)$ หมายถึงว่า $s(v) = False$

และเนื่องจากความสัมพันธ์การเปลี่ยนสถานะของโครงสร้างคริปเกต้องเป็นความสัมพันธ์แบบทั่วถึง ซึ่งอาจไม่จริงในกรณีที่บางสถานะ s ไม่มีสถานะถัดไป ในกรณีนี้เราต้องปรับความสัมพันธ์ R ให้กรณี $R(s, s)$ อยู่ในความสัมพันธ์ด้วย

<u>ตัวอย่าง</u> กำหนดให้ระบบมีตัวแปรสองตัวคือ x และ y ซึ่งมีค่าในเซต $D = \{0, 1\}$ การแทนค่า $(d_1, d_2) \in D \times D$ หมายถึงตัวแปร x มีค่าเป็น d_1 และตัวแปร y มีค่าเป็น d_2

ความสัมพันธ์การเปลี่ยนสถานะนิยามดังนี้

$$x := (x + y) \bmod 2$$

สถานะเริ่มต้นคือ $x = 1$ และ $y = 1$ เซตของสถานะเริ่มต้นสามารถแทนได้ด้วยสูตรทางตรรกนี้

$$S_0 \equiv x = 1 \wedge y = 1$$

และความสัมพันธ์การเปลี่ยนสถานะแทนได้ด้วยสูตรทางตรรก

$$R(x, y, x', y') \equiv x' = (x + y) \bmod 2 \wedge y' = y$$

โครงสร้างคริปเก $M = (S, S_0, R, L)$ ซึ่งได้จากสูตรทางตรรก S_0 และ R คือ

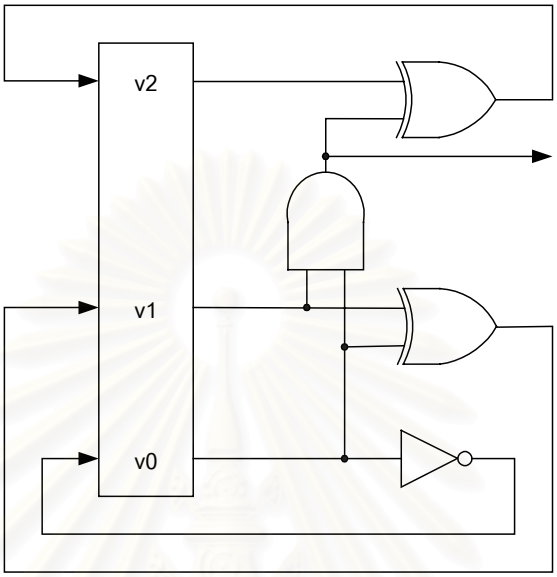
- $S = D \times D$
- $S_0 = \{(1, 1)\}$
- $R = \{((1, 1), (0, 1)), ((0, 1), (1, 1)), ((1, 0), (1, 0)), ((0, 0), (0, 0))\}$
- $L((1, 1)) = \{x = 1, y = 1\}$, $L((0, 1)) = \{x = 0, y = 1\}$, $L((1, 0)) = \{x = 1, y = 0\}$, และ $L((0, 0)) = \{x = 0, y = 0\}$

วิถีที่เริ่มจากสถานะเริ่มต้นที่เกิดขึ้นในโครงสร้างคริปเกของตัวอย่าง คือ (1,1) (0,1) (1,1) (0,1) … ซึ่งเป็นวิถีเดียวที่เกิดขึ้นในระบบนี้

3.1.3 การแทนวงจรดิจิทัลด้วยตรรกอันดับที่หนึ่ง

ในส่วนนี้นำเสนอวิธีการในการแทนวงจรด้วยสูตรทางตรรก เพื่อความง่ายในที่นี้จะพิจารณาว่า ส่วนประกอบซึ่งสามารถเก็บสถานะมีค่าได้สองอย่างคือ 0 หรือ 1 เท่านั้น และในที่นี้ V เป็นเซตของส่วนประกอบในวงจรซึ่งสามารถเก็บสถานะได้ สำหรับกรณีของวงจรสมวาร

(synchronous circuit) V คือเซตซึ่งประกอบด้วยเอาต์พุตของเรจิสเตอร์ในวงจรและอินพุตของวงจร และในกรณีของวงจรอสมวาร (asynchronous circuit) V คือเซตของสายสัญญาณทุกเส้นในวงจร



รูปที่ 3.1 วงจรตัวนับมอดุโลแปด

3.1.3.1 การแทนวงจรสมวาร

การทำงานของวงจรสมวารประกอบด้วยลำดับของขั้นตอนการทำงาน ในแต่ละขั้นตอนเมื่ออินพุตของวงจรมีการเปลี่ยนแปลงจะทำให้สถานะในวงจรมีการเปลี่ยนแปลงแล้วจึงเข้าสู่ภาวะสมดุล โดยเรจิสเตอร์ภายในวงจรจะเปลี่ยนค่าเมื่อสัญญาณนาฬิกาปรากฏ ในที่นี้จะนำเสนอตัวอย่างการแทนวงจรสมวารด้วยสูตรทางตรรก โดยตัวอย่างที่ใช้คือวงจรในรูปที่ 3.1 วงจรตัวนับมอดุโลแปด (modulo 8 counter)

ให้ $V = \{v_0, v_1, v_2\}$ เป็นเซตของตัวแปรสถานะ และ $V' = \{v'_0, v'_1, v'_2\}$ เป็นเซตของตัวแปรสถานะอีกชุดหนึ่ง การเปลี่ยนตัวแปรสถานะนิยามจาก

$$v'_0 = \neg v_0$$

$$v'_1 = v_0 \oplus v_1$$

$$v'_2 = (v_0 \wedge v_1) \oplus v_2$$

โดยที่ $\oplus$ คือ ตัวปฏิบัติการออร์เฉพาะ (exclusive-or operator) สมการด้านบนสามารถนิยามสูตรทางตรรกของการเปลี่ยนตัวแปรสถานะได้ดังนี้

$$R_0(V, V') \equiv (v'_0 \leftrightarrow \neg v_0)$$

$$R_1(V, V') \equiv (v'_1 \leftrightarrow v_0 \oplus v_1)$$

$$R_2(V, V') \equiv (v'_2 \leftrightarrow (v_0 \wedge v_1) \oplus v_2)$$

และเนื่องจากการเปลี่ยนแปลงของทุกเรจิสเตอร์ต้องเกิดขึ้นพร้อมกัน ดังนั้นสูตรทางตรรกทุกสูตรด้านบนจึงต้องเป็นจริงพร้อมกัน ทำให้ความสัมพันธ์การเปลี่ยนสถานะถูกสร้างจากการเชื่อม (conjunction) ของสูตรทางตรรกด้านบน

$$R(V, V') \equiv R_0(V, V') \wedge R_1(V, V') \wedge R_2(V, V')$$

ในกรณีทั่วไปของวงจรสมวารที่มีเรจิสเตอร์ n ตัว กำหนดให้เซตของตัวแปรสถานะคือเซต $V = \{v_0, \ldots, v_{n-1}\}$ และเซต $V' = \{v'_0, \ldots, v'_{n-1}\}$ สำหรับตัวแปรสถานะแต่ละตัวจะได้ว่า

$$v'_i = f_i(V)$$

สามารถนิยามสูตรทางตรรกของการเปลี่ยนตัวแปรสถานะได้เป็น

$$R_i(V, V') \equiv (v'_i \leftrightarrow f_i(V))$$

และได้สูตรทางตรรกของความสัมพันธ์การเปลี่ยนสถานะคือ

$$R(V, V') \equiv R_0(V, V') \wedge \ldots \wedge R_{n-1}(V, V')$$

3.1.3.2 การแทนวงจรอสมวาร

ในที่นี้เราจะพิจารณาเฉพาะกรณีที่ไม่มีส่วนประกอบภายในวงจรมากกว่าหนึ่งส่วนที่มีการเปลี่ยนค่าพร้อมกัน สูตรทางตรรกของความสัมพันธ์การเปลี่ยนสถานะของวงจรอสมวารคือ

$$R(V, V') \equiv R_0(V, V') \vee \ldots \vee R_{n-1}(V, V')$$

โดยที่

$$R_i(V, V') \equiv (v'_i \leftrightarrow f_i(V)) \wedge \bigwedge_{j \neq i} (v'_j \leftrightarrow v_j)$$

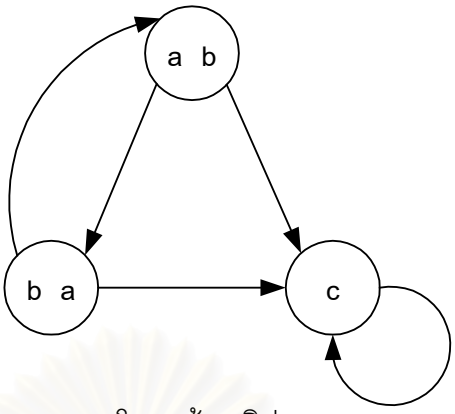
จากที่กล่าวมาจะเห็นได้ว่าทั้งวงจรสมวารและวงจรอสมวารสามารถถูกแทนโดยใช้ตรรกอันดับที่หนึ่งได้ ซึ่งเป็นประโยชน์เพราะว่าทำให้สามารถแทนวงจรในลักษณะของแผนภาพตัดสินใจทวิภาค (binary decision diagram) ซึ่งทำให้วิธีการตรวจสอบแบบจำลองเชิงสัญลักษณ์สามารถทวนสอบวงจรที่มีขนาดใหญ่ได้ สำหรับประเด็นนี้จะได้กล่าวถึงรายละเอียดกันต่อไปในบทนี้

3.2 ตรรกเชิงเวลา

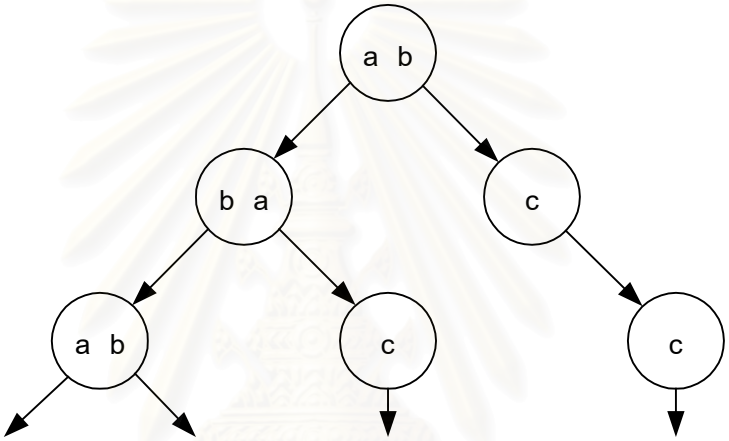
ถัดจากขั้นตอนการสร้างแบบจำลองคือ ขั้นตอนการสร้างข้อกำหนดซึ่งเป็นการกำหนดคุณสมบัติของวงจรโดยใช้ตรรกเชิงเวลา (temporal logic) ในที่นี้จะนำเสนอตรรกเชิงเวลาสองชนิดคือ ตรรกซีทีแอลสตาร์ (CTL*) และตรรกซีทีแอล (CTL) โดยตรรกซีทีแอลเป็นเซตย่อยของตรรกซีทีแอลสตาร์และเป็นตรรกเชิงเวลาชนิดที่โปรแกรมคาเดนค์เอสเอมวีสามารถทวนสอบได้

3.2.1 ตรรกซีทีแอลสตาร์

ตรรกเชิงเวลาชนิดแรกที่จะกล่าวถึงคือ ตรรกซีทีแอลสตาร์ (CTL*) ตรรกเชิงเวลาชนิดนี้สามารถใช้กำหนดคุณสมบัติของโครงสร้างคริปเกได้ โดยตรรกชนิดนี้จะเกี่ยวข้องกับคุณสมบัติของต้นไม้การคำนวณ (computation tree) ซึ่งสร้างได้จากโครงสร้างคริปเก โดยที่รากของต้นไม้คือสถานะเริ่มต้นของโครงสร้างคริปเก และบัพลูก (child node) คือสถานะที่มีเส้นเชื่อมจากสถานะปัจจุบัน ตัวอย่างเช่นในรูปที่ 3.2 (ข) เป็นต้นไม้การคำนวณที่สร้างจากโครงสร้างคริปเกในรูปที่ 3.2 (ก) จากรูปจะเห็นได้ว่าต้นไม้การคำนวณแสดงวิถีทุกทางที่เป็นไปได้จากสถานะเริ่มต้นของโครงสร้างคริปเก



(ก) โครงสร้างคริปเก



(ข) ต้นไม้การคำนวณที่ได้จากโครงสร้างคริปเก

รูปที่ 3.2 ตัวอย่างของต้นไม้การคำนวณ

ตรรกซีทีแอลสตาร์ประกอบด้วยตัวกำหนดวิถี (path quantifier) และตัวปฏิบัติการเชิงเวลา (temporal operator) ตัวกำหนดวิถีประกอบด้วยตัวกำหนดวิถีแบบทุกกรณี (universal path quantifier - A) ซึ่งจะเป็นจริงเมื่อทุกวิถีเป็นจริงตามเงื่อนไขที่ระบุ และตัวกำหนดวิถีแบบบางกรณี (existential path quantifier - E) จะเป็นจริงเมื่อมีบางวิถีที่เป็นจริงตามเงื่อนไขที่ระบุ ตัวกำหนดวิถีจะกำหนดคุณสมบัติที่เกี่ยวข้องกับวิถีที่สนใจ โดยวิถีที่สนใจจะเริ่มต้นจากสถานะนั้น สำหรับตัวปฏิบัติการเชิงเวลาประกอบด้วยตัวปฏิบัติการพื้นฐานห้าตัวคือ

- X *f* (“เป็นจริงในสถานะถัดไป”) จะเป็นจริงเมื่อคุณสมบัติ *f* เป็นจริงในสถานะที่สองบนวิถีที่พิจารณาอยู่

- F *f* (“เป็นจริงบางเวลา”) จะเป็นจริงเมื่อคุณสมบัติ *f* เป็นจริงในบางสถานะบนวิถีที่พิจารณาอยู่
- G *f* (“เป็นจริงตลอดเวลา”) จะเป็นจริงเมื่อคุณสมบัติ *f* เป็นจริงในทุกสถานะบนวิถีที่พิจารณาอยู่
- *f* U *g* (“เป็นจริงจนกระทั่ง”) จะเป็นจริงถ้ามีบางสถานะบนวิถีที่พิจารณาอยู่ซึ่งคุณสมบัติ *g* เป็นจริงและทุกสถานะก่อนหน้านั้นคุณสมบัติ *f* เป็นจริงตลอด

ตรรกซีทีแอลสตาร์มีสูตรทางตรรกอยู่สองประเภทคือ สูตรสถานะ (state formula) และสูตรวิถี (path formula) โดยสูตรสถานะจะมีค่าความจริงขึ้นกับสถานะ ขณะที่สูตรวิถีจะมีค่าความจริงขึ้นกับวิถี

ให้ P เป็นเซตของประพจน์เดี่ยว ไวยากรณ์ (syntax) ของสูตรสถานะเป็นดังนี้

- ถ้า $p \in P$, แล้ว p เป็นสูตรสถานะ
- ถ้า f และ g เป็นสูตรสถานะ, แล้ว $\neg f$, $f \vee g$ และ $f \wedge g$ เป็นสูตรสถานะ
- ถ้า f เป็นสูตรวิถี, แล้ว E f และ A f เป็นสูตรสถานะ

ไวยากรณ์ของสูตรวิถีเป็นดังนี้

- ถ้า f เป็นสูตรสถานะ, แล้ว f เป็นสูตรวิถีด้วย
- ถ้า f และ g เป็นสูตรวิถี, แล้ว $\neg f$, $f \vee g$, $f \wedge g$, X f, F f, G f, และ f U g เป็นสูตรวิถี

ตรรกซีทีแอลสตาร์คือเซตของสูตรสถานะซึ่งสร้างจากไวยากรณ์ที่กล่าวมาแล้ว ต่อไปจะกล่าวถึงความหมาย (semantic) ของตรรกซีทีแอลสตาร์ ถ้า f เป็นสูตรสถานะ สัญลักษณ์ $M, s \models f$ มีความหมายว่าสูตรสถานะ f เป็นจริงในสถานะ s ของโครงสร้างคริปเก $M = (S, R, L)$ และถ้า f เป็นสูตรวิถี สัญลักษณ์ $M, \pi \models f$ มีความหมายว่าสูตรวิถี f เป็นจริงในวิถี π ของโครงสร้างคริปเก M ในกรณีที่โครงสร้าง M เป็นที่รู้กันเราสามารถละไม่เขียน M ได้

กำหนดให้ f_1 และ f_2 เป็นสูตรสถานะ g_1 และ g_2 เป็นสูตรวิถี และนิยามว่า π^i หมายถึงวิถีที่เริ่มจากสถานะ s_i ของวิถี π เราสามารถนิยามความสัมพันธ์ $\models$ จากกฎดังนี้

1. $M, s \models p \quad \leftrightarrow \quad p \in L(s)$

2. $M, s \models \neg f_1 \quad \leftrightarrow \quad M, s \not\models f_1$

3. $M, s \models f_1 \vee f_2 \quad \leftrightarrow \quad M, s \models f_1$ หรือ $M, s \models f_2$

4. $M, s \models f_1 \wedge f_2 \quad \leftrightarrow \quad M, s \models f_1$ และ $M, s \models f_2$

5. $M, s \models \mathsf{E}\, g_1 \quad \leftrightarrow \quad$ มีบางวิถี π ที่เริ่มต้นจากสถานะ s ซึ่ง $M, \pi \models g_1$

6. $M, s \models \mathsf{A}\, g_1 \quad \leftrightarrow \quad$ สำหรับทุกวิถี π ที่เริ่มต้นจากสถานะ s, $M, \pi \models g_1$

7. $M, \pi \models f_1 \quad \leftrightarrow \quad$ ให้ s เป็นสถานะแรกของวิถี π และ $M, s \models f_1$

8. $M, \pi \models \neg g_1 \quad \leftrightarrow \quad M, \pi \not\models g_1$

9. $M, \pi \models g_1 \vee g_2 \quad \leftrightarrow \quad M, \pi \models g_1$ หรือ $M, \pi \models g_2$

10. $M, \pi \models g_1 \wedge g_2 \quad \leftrightarrow \quad M, \pi \models g_1$ และ $M, \pi \models g_2$

11. $M, \pi \models \mathsf{X}\, g_1 \quad \leftrightarrow \quad M, \pi^1 \models g_1$

12. $M, \pi \models \mathsf{F}\, g_1 \quad \leftrightarrow \quad$ มีบางค่า $k \geq 0$ ซึ่ง $M, \pi^k \models g_1$

13. $M, \pi \models \mathsf{G}\, g_1 \quad \leftrightarrow \quad$ สำหรับทุกค่า $i \geq 0$, $M, \pi^i \models g_1$

14. $M, \pi \models g_1 \,\mathsf{U}\, g_2 \quad \leftrightarrow \quad$ มีบางค่า $k \geq 0$ ซึ่ง $M, \pi^k \models g_2$ และ
สำหรับทุกค่า $0 \leq j < k$, $M, \pi^j \models g_1$

นอกจากนั้นจะสังเกตได้ว่าตัวปฏิบัติการ $\vee$, $\neg$, X, U, และ E เพียงพอที่จะใช้แทนสูตรทางตรรกทุกสูตรซึ่งเป็นตรรกซีทีแอลสตาร์ ตัวอย่างเช่น

- $f \wedge g \equiv \neg (\neg f \vee \neg g)$
- $\mathsf{F}\, f \equiv \mathit{True}\, \mathsf{U}\, f$
- $\mathsf{G}\, f \equiv \neg \mathsf{F} \neg f$

- $A(f) \equiv \neg E(\neg f)$

3.2.2 ตรรกซีทีแอล

ตรรกซีทีแอล (CTL – Computation Tree Logic) (Clarke, Emerson และ Sislta, 1986) เป็นเซตย่อยของตรรกซีทีแอลสตาร์ ซึ่งมีเงื่อนไขว่าตัวกำหนดวิถีต้องใช้คู่กับตัวปฏิบัติการเชิงเวลาเสมอ นอกจากนั้นตรรกซีทีแอลยังมีการเพิ่มไวยากรณ์ด้วยดังนี้

- ถ้า f และ g เป็นสูตรสถานะ, แล้ว $\neg f$, $f \vee g$, $f \wedge g$, $\mathbf{X} f$, $\mathbf{F} f$, $\mathbf{G} f$, และ $f \mathbf{U} g$ เป็นสูตรวิถี

ตัวปฏิบัติการพื้นฐานของตรรกซีทีแอลประกอบด้วย

- AX และ EX
- AF และ EF
- AG และ EG
- AU และ EU

ตัวปฏิบัติการทั้งแปดตัวสามารถเขียนในรูปของตัวปฏิบัติการเพียงสามตัวคือ EX, EG, และ EU ดังนี้

- $AX f = \neg EX(\neg f)$
- $EF f = E[True \mathbf{U} f]$
- $AG f = \neg EF(\neg f)$
- $AF f = \neg EG(\neg f)$
- $A[f \mathbf{U} g] = \neg E[\neg g \mathbf{U} (\neg f \wedge \neg g)] \wedge \neg EG \neg g$

รูปที่ 3.3 แสดงตัวปฏิบัติการพื้นฐานที่ใช้บ่อยของตรรกซีทีแอล ตัวปฏิบัติการเหล่านี้สามารถเข้าใจได้ง่ายเมื่อถูกแสดงในลักษณะของต้นไม้การคำนวณ ต่อไปนี้เป็นตัวอย่างของคุณสมบัติที่เขียนด้วยตรรกซีทีแอล

- EF(*Start* $\wedge \neg$ *Ready*) : ในบางสถานะเป็นไปได้ที่สัญญาณ *Start* เป็นจริงแต่สัญญาณ *Ready* ไม่เป็นจริง
- AG(*Reg* $\rightarrow$ AF *Ack*) : ถ้าสัญญาณ *Reg* เป็นจริง, หลังจากนั้นสัญญาณ *Ack* จะต้องเป็นจริง
- AG(AF *DeviceEnabled*) : สัญญาณ *DeviceEnabled* จะต้องเป็นจริงบ่อยครั้งจนถึงอนันต์
- AG(EF *Restart*) : ในสถานะใดก็ตามเป็นไปได้เสมอที่จะสัญญาณ *Restart* จะเป็นจริง
- AG(*Send* $\rightarrow$ A(*Send* U *Recv*)) : ถ้าสัญญาณ *Send* เป็นจริง หลังจากนั้นสัญญาณ *Recv* จะต้องเป็นจริง และจนกว่าจะถึงเวลานั้นสัญญาณ *Send* ต้องเป็นจริงตลอด

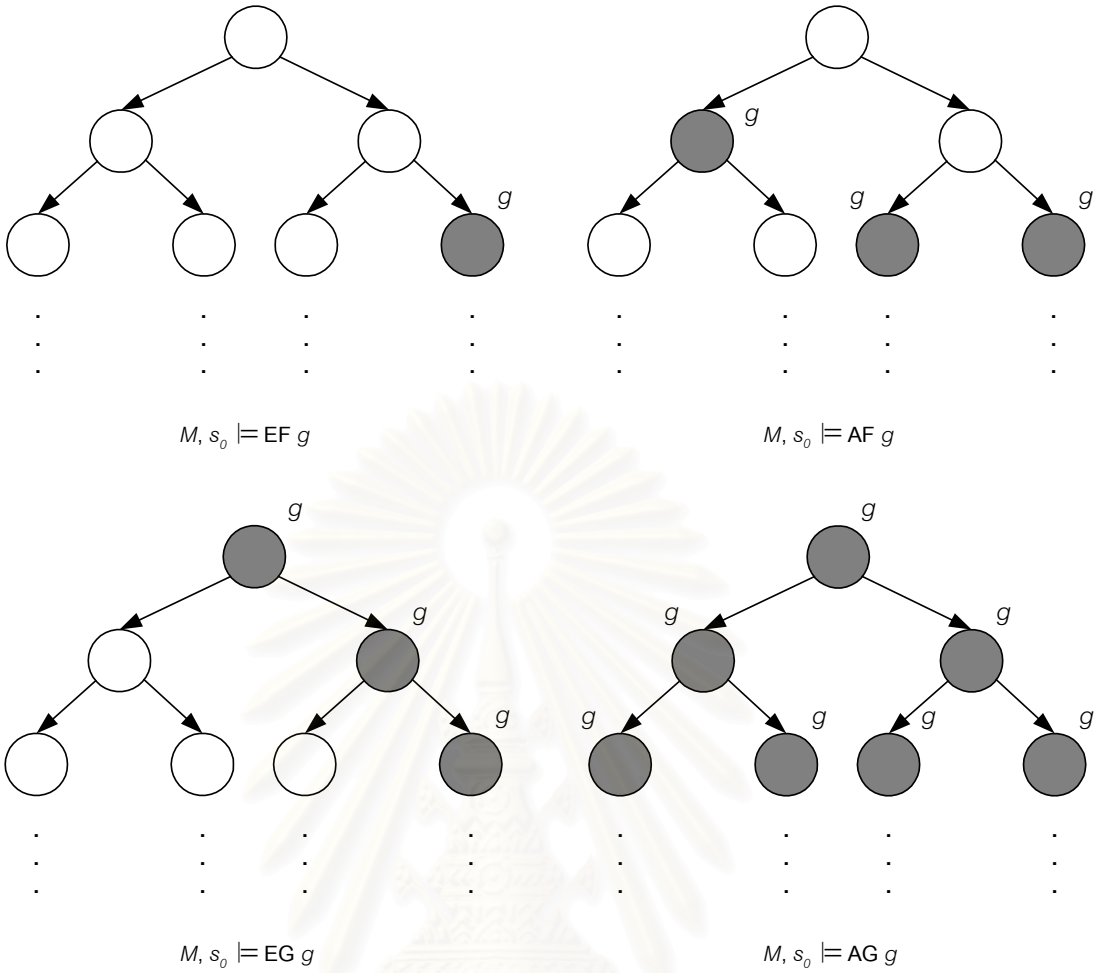
3.3 การตรวจสอบแบบจำลอง

หลังจากขั้นตอนการสร้างแบบจำลองและขั้นตอนการสร้างข้อกำหนด ก็มาถึงขั้นตอนที่สามคือการทวนสอบ โดยในที่นี้ใช้วิธีการที่ชื่อว่าการตรวจสอบแบบจำลอง (model checking) โดยการตรวจสอบแบบจำลองเป็นกระบวนการอัตโนมัติ คุณสมบัติที่กำหนดไว้จะถูกตรวจสอบ และในกรณีที่แบบจำลองที่สร้างขึ้นไม่สอดคล้องกับคุณสมบัติ โปรแกรมช่วยทวนสอบจะสร้างตัวอย่างกรณีที่เกิดปัญหาให้ด้วยซึ่งจะมีประโยชน์ในการแก้ไขแบบจำลอง

ในที่นี้จะนำเสนอเฉพาะการตรวจสอบแบบจำลองสำหรับตรรกซีทีแอลเท่านั้น การตรวจสอบแบบจำลองสามารถนิยามดังนี้ กำหนดให้ $M = (S, R, L)$ เป็นโครงสร้างคริปเก และ f เป็นคุณสมบัติในรูปตรรกซีทีแอล การตรวจสอบแบบจำลองคือการหาเซตของสถานะซึ่งทำให้คุณสมบัติ f เป็นจริง

$$\{ s \in S \mid M, s \models f \}$$

โครงสร้างคริปเกจะสอดคล้องกับคุณสมบัติที่ต้องการเมื่อสถานะเริ่มต้นทุกสถานะอยู่ในเซตที่ได้จากการตรวจสอบแบบจำลอง



รูปที่ 3.3 ตัวปฏิบัติการพื้นฐานของตรรกซีทีแอล

อัลกอริทึมที่ใช้ตรวจสอบแบบจำลองซึ่งนำเสนอในหัวข้อนี้ จะทำการสร้างกราฟของโครงสร้างคริปเกแล้วจึงทำการตรวจสอบคุณสมบัติในกราฟดังกล่าว บัพ (node) ของกราฟใช้แทนสถานะภายในเซต S และเส้นเชื่อมแบบมีทิศทางใช้แทนการเปลี่ยนสถานะในเซต R นอกจากนั้นในแต่ละบัพจะมีป้ายซึ่งระบุประพจน์ที่เป็นจริงในสถานะนั้นตามฟังก์ชัน $L : S \rightarrow 2^P$ ในรูปที่ 3.2 (ก) แสดงตัวอย่างของโครงสร้างคริปเก

ให้ f เป็นคุณสมบัติในรูปตรรกซีทีแอล อัลกอริทึมของการตรวจสอบแบบจำลองทำงานโดยการสร้างฟังก์ชัน *label(s)* ซึ่งทำหน้าที่เหมือนป้าย (label) ที่บอกว่าในแต่ละสถานะมีประพจน์ใดเป็นจริงบ้าง โดยในที่นี้ใช้ระบุว่าสูตรทางตรรก f เป็นจริงหรือไม่ การสร้างฟังก์ชัน *label(s)* จะเริ่มต้นจากการสร้างฟังก์ชันของประพจน์เดี่ยว แล้วจึงสร้างฟังก์ชันของตัวปฏิบัติการต่างๆ กล่าวอีกนัยหนึ่งคือทำงานในลักษณะจากล่างขึ้นบน (bottom-up)

เนื่องจากตรรกซีทีแอลมีรูปแบบพื้นฐานที่สุดอยู่หกแบบ คือ ประพจน์เดี่ยว f_1, $\neg f_1$, $f_1 \vee f_2$, EX f_1, E[f_1 U f_2], และ EG f_1 อัลกอริทึมของการตรวจสอบแบบจำลองจึงแบ่งออกได้เป็นหกกลุ่มได้แก่

1. ประพจน์เดี่ยว f_1 : ฟังก์ชัน *label(s)* ประกอบด้วยสถานะที่มีป้ายบอกว่า f_1 เป็นจริง กรณีนี้สามารถพิจารณาจาก *L(s)* ได้โดยตรง

2. ตรรกซีทีแอลในรูป $\neg f_1$: ฟังก์ชัน *label(s)* ประกอบด้วยสถานะที่ไม่มีป้ายบอกว่า f_1 เป็นจริง กรณีนี้สามารถพิจารณาจาก *L(s)* โดยตรง

3. ตรรกซีทีแอลในรูป $f_1 \vee f_2$: ฟังก์ชัน *label(s)* ประกอบด้วยสถานะที่มีป้ายบอกว่า f_1 เป็นจริง หรือ f_2 เป็นจริง กรณีนี้สามารถพิจารณาจาก *L(s)* ได้โดยตรง

4. ตรรกซีทีแอลในรูป EX f_1 : ฟังก์ชัน *label(s)* ประกอบด้วยสถานะซึ่งมีสถานะถัดไปถูกระบุว่า f_1 เป็นจริง กรณีนี้สามารถพิจารณาจาก *L(s)* ได้โดยตรงเช่นกัน

5. ตรรกซีทีแอลในรูป E[f_1 U f_2] : อัลกอริทึมเริ่มต้นด้วยการหาสถานะที่ f_2 เป็นจริง จากนั้นจึงทำการค้นหาแบบย้อนหลังจากสถานะนั้น เพื่อหาวิถีที่สถานะก่อนหน้าสถานะดังกล่าวมี f_1 เป็นจริงตลอด โดยที่สมมุติว่าสูตรทางตรรก f_1 และ f_2 ผ่านอัลกอริทึมการตรวจสอบแบบจำลองแล้ว ในรูปที่ 3.4 แสดงอัลกอริทึมชื่อ *CheckEU* ซึ่งใช้สร้างฟังก์ชัน *label(s)* ของกรณีนี้โดยที่อัลกอริทึมนี้ใช้เวลาทำงาน $O(|S| + |R|)$

6. ตรรกซีทีแอลในรูป EG f_1 : อัลกอริทึมในกรณีนี้ใช้พื้นฐานของการแบ่งกราฟให้เป็นส่วนประกอบซึ่งถูกเชื่อมต่อกันอย่างทั่วถึง (strongly connected component) *C* ซึ่งคือกราฟย่อยที่ใหญ่ที่สุดซึ่งทุกบัพใน *C* มีเส้นทางเชื่อมต่อถึงกันได้หมด และนอกจากนั้นใน *C* จะต้องมีมากกว่าหนึ่งบัพ หรือในกรณีที่มีเพียงหนึ่งบัพจะต้องมีเส้นเชื่อมเป็นวงวน (loop) เข้าหาบัพนั้นเอง

ให้ *M'* เป็นโครงสร้างคริปเกซึ่งสร้างจาก *M* โดยลบสถานะในเซต *S* ซึ่ง f_1 ไม่เป็นจริง สำหรับ *R* และ *L* ก็เช่นเดียวกัน ดังนั้นได้ว่า $M'=(S', R', L')$ โดยที่ $S' = \{ s \in S \mid M, s \models f_1 \}$, $R' = R|_{S' \times S'}$ และ $L' = L|_{S'}$ อัลกอริทึมที่จะนำเสนอต่อไปขึ้นกับข้อสังเกตต่อไปนี้ (สามารถดูการพิสูจน์ได้ใน (Clarke, Grumberg และ Peled, 2001: 36))

Lemma1 $M, s \models \mathsf{EG}\, f_1$ ก็ต่อเมื่อเงื่อนไขสองข้อต่อไปนี้เป็นจริง

1. $s \in S'$
2. มีบางวิถีใน M' จากบัพ s ไปสู่บัพ t ซึ่งอยู่ภายในส่วนประกอบซึ่งถูกเชื่อมต่อกันอย่างทั่วถึง C ของกราฟ (S', R')

อัลกอริทึมสำหรับกรณี $\mathsf{EG}\, f_1$ สร้างโดยตรงจากข้อสังเกตด้านบน โดยทำการสร้างโครงสร้างคริปเก $M'=(S', R', L')$ และแบ่งกราฟ (S', R') เป็นส่วนประกอบซึ่งถูกเชื่อมต่อกันอย่างทั่วถึงโดยใช้อัลกอริทึมของทาร์จาน (algorithm of Tarjan) (Aho, Hopcroft และ Ullman, 1974) โดยอัลกอริทึมนี้ใช้เวลาการทำงานเป็น $O(|S'| + |R'|)$ จากนั้นเริ่มต้นหาสถานะที่ f_1 เป็นจริงในส่วนประกอบซึ่งถูกเชื่อมต่อกันอย่างทั่วถึง แล้วจึงค้นหาแบบย้อนหลังจากสถานะนั้นเพื่อหาวิถีที่ f_1 เป็นจริงตลอด โดยที่สมมุติว่าสูตรทางตรรก f_1 ผ่านอัลกอริทึมการตรวจสอบแบบจำลองแล้ว ในรูปที่ 3.5 แสดงอัลกอริทึมชื่อ *CheckEG* ซึ่งใช้สร้างฟังก์ชัน *label(s)* ของกรณีนี้โดยที่อัลกอริทึมนี้ใช้เวลาการทำงานเป็น $O(|S| + |R|)$

อัลกอริทึมที่ใช้ในการตรวจสอบแบบจำลองของตรรกซีทีแอลทำงานโดยเริ่มต้นจากสร้างฟังก์ชัน *label(s)* ของประพจน์เดี่ยว แล้วจึงสร้างฟังก์ชันของสูตรย่อยแต่ละสูตรภายในสูตรทางตรรก f และจากข้อสังเกตที่ว่าการทำงานของอัลกอริทึมแต่ละรอบใช้เวลา $O(|S| + |R|)$ และในสูตรทางตรรก f มีสูตรย่อยมากที่สุดอยู่ $|f|$ ดังนั้นเวลาการทำงานทั้งหมดจึงเป็น

$$O(|f| \cdot (|S| + |R|))$$

<u>ตัวอย่าง</u> ต่อไปนี้แสดงตัวอย่างการตรวจสอบแบบจำลอง โดยใช้ตัวอย่างซึ่งทำการตรวจสอบคุณสมบัติของเตาไมโครเวฟ ในรูปที่ 3.6 แสดงโครงสร้างคริปเกของเตาไมโครเวฟ ข้อความที่เส้นเชื่อมคือการกระทำที่ทำให้มีการเปลี่ยนสถานะซึ่งไม่รวมอยู่ในโครงสร้างคริปเก และไม่ได้ถูกใช้ในอัลกอริทึมของการตรวจสอบแบบจำลอง

ในที่นี้ต้องการทวนสอบสูตรทางตรรก $\mathsf{AG}(\mathit{Start} \rightarrow \mathsf{AF}\, \mathit{Heat})$ โดยที่สูตรนี้สมมูลกับ $\neg\, \mathsf{EF}(\mathit{Start} \wedge \mathsf{EG} \neg\, \mathit{Heat})$ (ในที่นี้ใช้ $\mathsf{EF}\, f$ เพื่อแทน $\mathsf{E}[\mathit{True}\ \mathsf{U}\ f]$) อัลกอริทึมเริ่มต้นด้วยการสร้างเซตของสถานะซึ่งประพจน์เดี่ยวเป็นจริง แล้วจึงหาเซตของสถานะที่สูตรย่อยเป็นจริง ให้ $S(g)$

เป็นเซตของสถานะซึ่งสูตรทางตรรก g เป็นจริง อัลกอริทึมที่ใช้หาเซต $S(p)$ สำหรับทุกประพจน์เดี่ยว $p \in P$ ใช้เวลาทำงานเป็น $O(|S| + |R|)$

```
procedure CheckEU (f1, f2)
        T := { s | f2 ∈ label(s) };
        for all s ∈ T do label(s) := label(s) ∪ { E[f1 U f2] };
        while T ≠ ∅ do
                choose s ∈ T;
                T := T \ { s };
                for all t such that R(t, s) do
                        if  E[f1 U f2] ∉ label(t) and f1 ∈ label(t) then
                                label(t) := label(t) ∪ { E[f1 U f2] };
                                T := T ∪ { t };
                        end if;
                end for all;
        end while;
end procedure
```

รูปที่ 3.4 อัลกอริทึมเพื่อใช้สร้างฟังก์ชัน *label(s)* ของ $E[f_1 \, U \, f_2]$

$S(Start) = \{2, 5, 6, 7\}$

$S(\neg Heat) = \{1, 2, 3, 5, 6\}$

เพื่อที่จะสร้าง $S(EG \neg Heat)$ เราเริ่มต้นด้วยการสร้างเซตของส่วนประกอบซึ่งถูกเชื่อมต่อกันอย่างทั่วถึงในเซต $S' = S(\neg Heat)$ ได้ว่า $SCC = \{ \{1, 2, 3, 5\} \}$ จากนั้นจึงสร้างเซต T ซึ่งเป็นเซตของสถานะซึ่งได้จากการยูเนียน (union) ทุกสถานะในสมาชิกของ SCC จึงได้ว่าในตอนเริ่มต้น $T = \{1, 2, 3, 5\}$ และเนื่องจากไม่มีสถานะอื่นในเซต S' ซึ่งไม่อยู่ใน T แต่มีเส้นทางไปสู่สถานะในเซต T ได้ ดังนั้นจึงได้ว่า

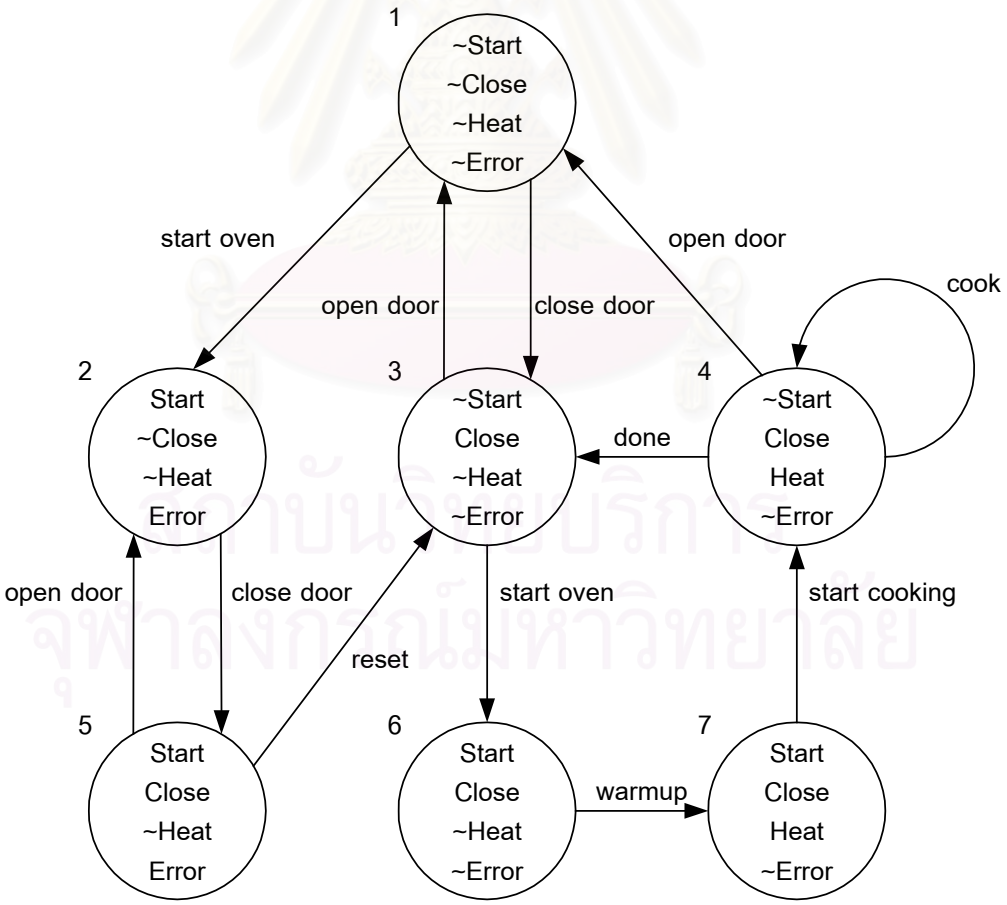
$S(EG \neg Heat) = \{1, 2, 3, 5\}$

และ

$S(Start \wedge EG \neg Heat) = \{2, 5\}$

```
procedure CheckEG (f1)
    S' := { s | f1 ∈ label(s) };
    SCC := { C | C is a nontrivial Strongly Connected Component of S' };
    T := ⋃c∈SCC { s | s ∈ C };
    for all s ∈ T do label(s) := label(s) ∪ { EG f1 };
    while T ≠ ∅ do
        choose s ∈ T;
        T := T \ { s };
        for all t such that t ∈ S' and R(t, s) do
            if EG f1 ∉ label(t) then
                label(t) := label(t) ∪ { EG f1 };
                T := T ∪ { t };
            end if;
        end for all;
    end while;
end procedure
```

รูปที่ 3.5 อัลกอริทึมเพื่อใช้สร้างฟังก์ชัน *label(s)* ของ EG f_1



รูปที่ 3.6 ตัวอย่างโครงสร้างคริปเกของเตาไมโครเวฟ

จากนั้นจึงสร้างเซต $S(\mathbf{EF}(Start \wedge \mathbf{EG} \neg Heat))$ โดยเริ่มจากให้ $T = S(Start \wedge \mathbf{EG} \neg Heat)$ จากนั้นจึงทำการค้นหาแบบย้อนหลังได้ว่า

$$S(\mathbf{EF}(Start \wedge \mathbf{EG} \neg Heat)) = \{1, 2, 3, 4, 5, 6, 7\}$$

สุดท้ายได้ว่า

$$S(\neg \mathbf{EF}(Start \wedge \mathbf{EG} \neg Heat)) = \varnothing$$

และเนื่องจากสถานะ '1' ซึ่งเป็นสถานะเริ่มต้นไม่อยู่ในเซตนี้ จึงสรุปได้ว่าโครงสร้างคริปเกของเตาไมโครเวฟไม่สอดคล้องกับคุณสมบัติที่ต้องการ

3.4 แผนภาพตัดสินใจทวิภาค

3.4.1 การแทนฟังก์ชันแบบบูล

ในหัวข้อนี้นำเสนอการแทนโครงสร้างคริปเกด้วยแผนภาพตัดสินใจทวิภาค (binary decision diagram) โดยก่อนที่จะอธิบายถึงแผนภาพตัดสินใจทวิภาค ขอกล่าวถึงต้นไม้ตัดสินใจทวิภาค (binary decision tree) ก่อน ต้นไม้ชนิดนี้จะมีบัพ (node) อยู่สองชนิดคือ บัพปลายทาง (terminal node) และบัพต่อ (nonterminal node) บัพต่อ v ถูกกำหนดป้ายด้วยตัวแปร $var(v)$ และมีบัพลูก (child node) อยู่สองบัพคือ $low(v)$ ซึ่งจะสอดคล้องกับกรณีที่ตัวแปร v เป็น *'0'* และ $high(v)$ ซึ่งจะสอดคล้องกับกรณีที่ตัวแปร v เป็น *'1'* สำหรับบัพปลายทาง v ถูกกำหนดป้ายด้วย $value(v)$ ซึ่งมีค่าเป็น *'0'* หรือ *'1'*

ในที่นี้แสดงตัวอย่างของอุปกรณ์เปรียบเทียบขนาดสองบิต (two-bit comparator) ซึ่งมีสูตรทางตรรกเป็น $f(a_1, a_2, b_1, b_2) = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$ ในรูปที่ 3.7 แสดงต้นไม้ตัดสินใจทวิภาคของวงจรนี้ การหาค่าความจริงของ f ทำได้โดยการสำรวจในต้นไม้จากบัพราก (root node) จนถึงบัพปลายทาง ถ้าตัวแปร v เป็น *'0'* จะสำรวจต่อที่บัพ $low(v)$ แต่ถ้าเป็น *'1'* จะสำรวจต่อที่บัพ $high(v)$ เมื่อถึงบัพปลายทางค่าความจริงของป้ายในบัพนั้นคือค่าความจริงของ f ตัวอย่างเช่นกรณี $\langle a_1 := 1, a_2 := 0, b_1 := 1, b_2 := 1 \rangle$ จะพบว่าค่าฟังก์ชัน f เป็น *'0'* หรือหมายถึงเป็น *False*

จากรูปที่ 3.7 จะเห็นได้ว่าต้นไม้ตัดสินใจทวิภาคมีขนาดค่อนข้างใหญ่ โดยมีขนาดใกล้เคียงกับขนาดของตารางค่าความจริง และในต้นไม้มีส่วนที่ซ้ำซ้อนกันค่อนข้างมาก เช่นในรูปที่ 3.7 จะเห็นว่ามีต้นไม้ย่อยซึ่งมีบัพรากคือ b_2 อยู่แปดต้น แต่เป็นต้นไม้ที่ต่างกันเพียงสามต้นเท่านั้น ดังนั้นจึงสามารถรวมต้นไม้ย่อยที่เหมือนกันเข้าด้วยกันได้เพื่อให้ได้รูปแบบการแทนที่กระชับมากขึ้น ผลลัพธ์ที่ได้จากการรวมส่วนที่ซ้ำซ้อนเข้าด้วยกันมีชื่อเรียกว่า แผนภาพตัดสินใจทวิภาค

ต่อไปจะกล่าวถึงนิยามของแผนภาพตัดสินใจทวิภาค สำหรับแต่ละบัพต่อ v ในแผนภาพจะถูกกำหนดป้ายด้วยตัวแปร *var(v)* ซึ่งมีบัพลูกสองบัพคือ *low(v)* และ *high(v)* สำหรับแต่ละบัพปลายทางจะถูกกำหนดป้ายเป็น *'0'* หรือ *'1'* และสำหรับทุกแผนภาพตัดสินใจทวิภาค B ซึ่งมีบัพรากคือ v จะสมมูลกับฟังก์ชันแบบบูล $f_v(x_1, \ldots, x_n)$ ซึ่งนิยามดังนี้

1. เมื่อ v เป็นบัพปลายทาง :

(a) ถ้า *value(v)* = 1, แล้ว $f_v(x_1, \ldots, x_n) = 1$.

(b) ถ้า *value(v)* = 0, แล้ว $f_v(x_1, \ldots, x_n) = 0$.

2. เมื่อ v เป็นบัพต่อซึ่ง *var(v)* = x_i แล้ว f_v คือฟังก์ชัน

$$f_v(x_1, \ldots, x_n) = (\neg x_i \wedge f_{low(v)}(x_1, \ldots, x_n)) \vee (x_i \wedge f_{high(v)}(x_1, \ldots, x_n))$$

เพื่อประสิทธิภาพของกระบวนการทวนสอบ เราต้องการรูปแบบการแทนซึ่งทำให้ฟังก์ชันแบบบูลสองฟังก์ชันจะสมมูลกันก็ต่อเมื่อรูปแบบการแทนของทั้งสองฟังก์ชันมีสัณฐานเหมือนกัน (isomorphic) ซึ่งคุณสมบัตินี้จะช่วยให้การตรวจสอบความสมมูลง่ายขึ้น โดยแผนภาพตัดสินใจทวิภาคสองแผนภาพจะมีสัณฐานเหมือนกันเมื่อมีฟังก์ชันแบบหนึ่งต่อหนึ่ง h ซึ่งกำหนดความสัมพันธ์ระหว่างบัพในแผนภาพทั้งสอง โดยบัพต่อจะถูกกำหนดความสัมพันธ์เฉพาะกับบัพต่อ และบัพปลายทางจะถูกกำหนดความสัมพันธ์เฉพาะกับบัพปลายทางเท่านั้น สำหรับทุกบัพปลายทาง v ได้ว่า *value(v) = value(h(v))* และสำหรับทุกบัพต่อ v ได้ว่า *var(v) = var(h(v))*, *h(low(v)) = low(h(v))*, และ *h(high(v)) = high(h(v))*

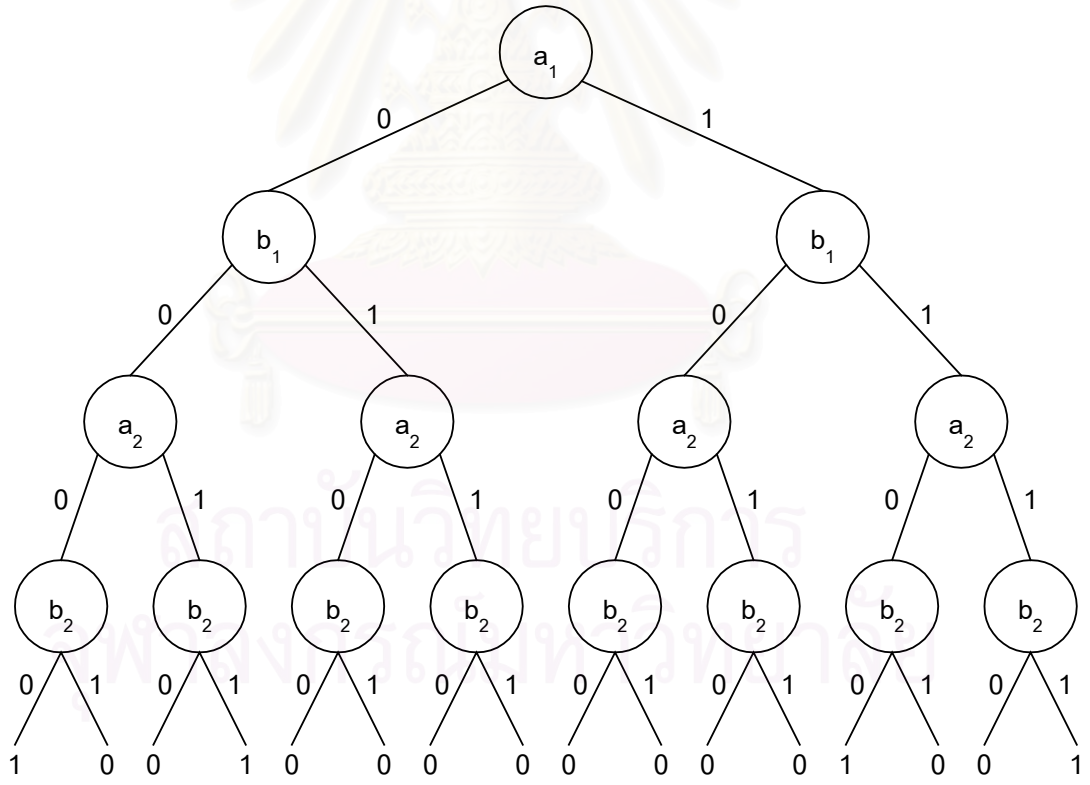
Bryant (1986) เสนอวิธีการสร้างรูปแบบการแทนซึ่งมีคุณสมบัติดังอธิบายในย่อหน้าที่แล้วโดยทำการเพิ่มเงื่อนไขสองอย่าง เงื่อนไขแรกคือตัวแปรในแผนภาพต้องปรากฏด้วยอันดับเดียวกันในทุกวิถีจากบัพรากถึงบัพปลายทาง เงื่อนไขที่สองคือจะต้องไม่มีต้นไม้ย่อยซึ่งมี

สัณฐานเหมือนกันและไม่มีบัพที่ซ้ำซ้อนกันในแผนภาพ สำหรับเงื่อนไขแรกสามารถสร้างได้โดยใช้อันดับ < ของตัวแปรในแผนภาพ โดยกำหนดว่าถ้า *u* มีบัพลูกซึ่งเป็นบัพต่อ แล้ว *var(u) < var(v)* สำหรับเงื่อนไขที่สองสามารถสร้างได้โดยใช้กฎการแปลงสามข้อดังนี้

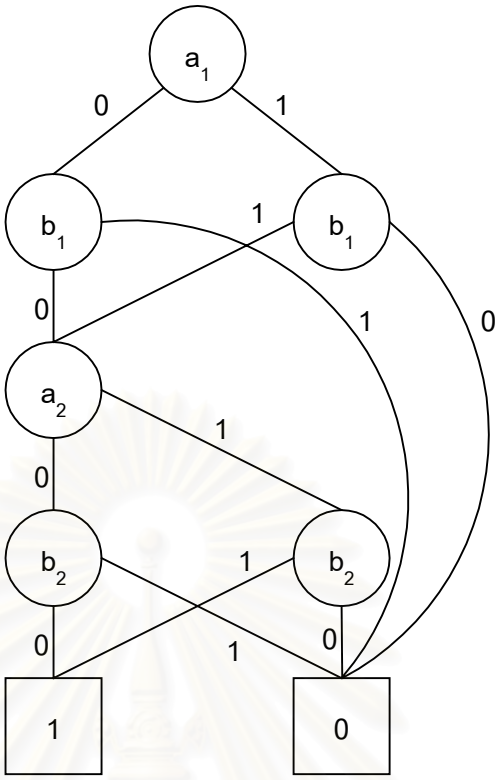
- <u>กำจัดบัพปลายทางที่ซ้ำซ้อน</u>: กำจัดบัพปลายทางที่มีป้ายเหมือนกันให้เหลือเพียงบัพเดียว และเปลี่ยนเส้นเชื่อมมายังบัพที่เหลืออยู่

- <u>กำจัดบัพต่อที่ซ้ำซ้อน</u>: ถ้าบัพต่อสองบัพ *u* และ *v* มีเงื่อนไขต่อไปนี้เป็นจริง *var(u) = var(v)*, *low(u) = low(v)*, และ *high(u) = high(v)* แล้วให้ตัดบัพ *u* หรือ *v* และทำการเปลี่ยนเส้นเชื่อมมายังบัพที่เหลือ

- <u>กำจัดบัพที่ไม่จำเป็น</u>: ถ้าบัพต่อ *v* มีเงื่อนไขต่อไปนี้เป็นจริง *low(v) = high(v)* แล้วให้ตัดบัพ *v* และเปลี่ยนเส้นเชื่อมที่เข้าบัพ *v* ให้ไปที่บัพ *low(v)* แทน



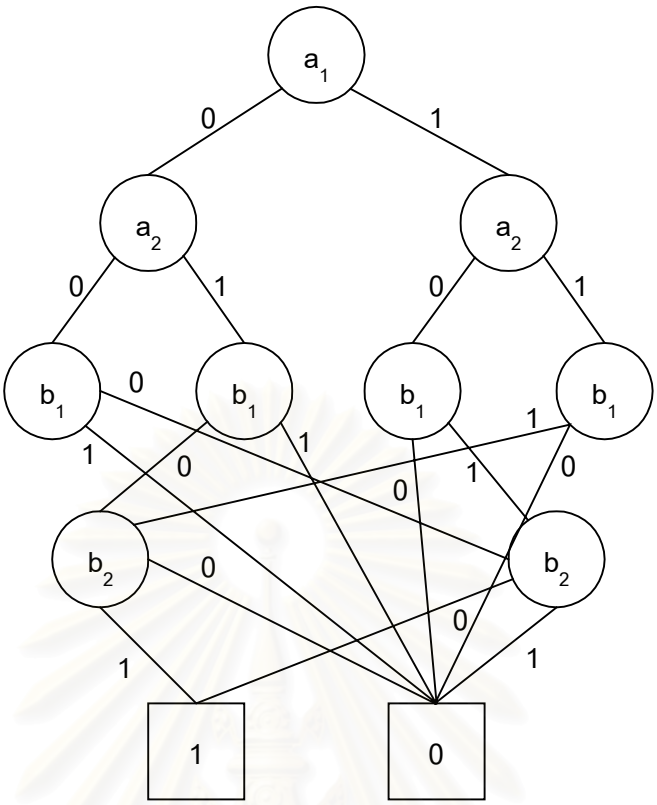
รูปที่ 3.7 ต้นไม้ตัดสินใจทวิภาคของอุปกรณ์เปรียบเทียบขนาดสองบิต



รูปที่ 3.8 แผนภาพตัดสินใจทวิภาคแบบอันดับกรณีที่เล็กที่สุด

กระบวนการสร้างแผนภาพเริ่มต้นด้วยการสร้างแผนภาพตัดสินใจทวิภาคซึ่งสอดคล้องกับคุณสมบัติเรื่องอันดับของตัวแปร จากนั้นจึงใช้กฎการแปลงจนกระทั่งขนาดของแผนภาพไม่ลดลงอีก โดย Bryant (1986) ได้เสนออัลกอริทึม *Reduce* ซึ่งทำการแปลงด้วยกฎการแปลงที่ได้นำเสนอ อัลกอริทึมนี้เป็นกระบวนการทำงานแบบจากล่างขึ้นบน (bottom-up) ซึ่งใช้เวลาทำงานเป็นเชิงเส้นของขนาดแผนภาพตัดสินใจทวิภาค แผนภาพซึ่งสอดคล้องกับเงื่อนไขเหล่านี้มีชื่อว่าแผนภาพตัดสินใจทวิภาคแบบอันดับ (ordered binary decision diagram)

ตัวอย่างของแผนภาพตัดสินใจทวิภาคแบบอันดับเช่น ถ้าให้อันดับของตัวแปรเป็น $a_1 < b_1 < a_2 < b_2$ จะได้แผนภาพตัดสินใจทวิภาคแบบอันดับของอุปกรณ์เปรียบเทียบขนาดสองบิตดังแสดงในรูปที่ 3.8 ซึ่งเป็นแผนภาพในกรณีที่เล็กที่สุด ข้อสังเกตหนึ่งที่น่าสนใจคือขนาดของแผนภาพตัดสินใจทวิภาคแบบอันดับขึ้นกับอันดับของตัวแปร ตัวอย่างเช่นถ้าให้อันดับของตัวแปรเป็น $a_1 < a_2 < b_1 < b_2$ จะได้แผนภาพตัดสินใจทวิภาคแบบอันดับของอุปกรณ์เปรียบเทียบขนาดสองบิตดังแสดงในรูปที่ 3.9 ซึ่งมีจำนวนบัพถึงสิบเอ็ดบัพ ขณะที่แผนภาพในรูปที่ 3.8 มีเพียงแปดบัพเท่านั้น



รูปที่ 3.9 แผนภาพตัดสินใจทวิภาคแบบอันดับกรณีอื่น

การหาอันดับของตัวแปรซึ่งทำให้แผนภาพมีขนาดเล็กที่สุดเป็นเรื่องเป็นไปไม่ได้ในทางทฤษฎี และการตรวจสอบว่าอันดับของตัวแปรที่กำหนดทำให้แผนภาพมีขนาดเล็กที่สุดหรือไม่นั้นเป็นปัญหาในกลุ่มเอ็นพีสมบูรณ์ (NP-complete) (Bryant, 1992) นอกจากนั้นยังมีวงจรบางกลุ่มซึ่งไม่สามารถหาอันดับของตัวแปรที่ทำให้แผนภาพมีขนาดเล็กได้เช่น วงจรคูณ เป็นต้น อย่างไรก็ตามสำหรับกรณีวงจรที่ใช้กันทั่วไปจะมีอัลกอริทึมซึ่งสามารถหาอันดับของตัวแปรที่เหมาะสมได้ในเวลาสมเหตุสมผล ทำให้แผนภาพตัดสินใจทวิภาคแบบอันดับถูกใช้อย่างกว้างขวางในปัจจุบัน

ต่อไปจะเป็นการนำเสนออัลกอริทึมในการจัดการกับตัวปฏิบัติการทางตรรกโดยใช้แผนภาพตัดสินใจทวิภาคแบบอันดับ โดยจะเริ่มต้นด้วยฟังก์ชันการแทนค่าตัวแปร x_i ของฟังก์ชันแบบบูล f ด้วยค่าคงที่ b ฟังก์ชันนี้ถูกแทนด้วยสัญลักษณ์ $f|_{x_i \leftarrow b}$ โดยที่

$$f|_{x_i \leftarrow b}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n)$$

ถ้า f ถูกแทนด้วยแผนภาพตัดสินใจทวิภาคแบบอันดับ แล้วการหาแผนภาพของฟังก์ชัน $f|_{x_i \leftarrow b}$ ทำได้โดยการแวะผ่านในแนวลึก (depth-first traversal) ภายในแผนภาพ โดยที่สำหรับทุกบัพ v ที่บัพถัดไปเป็น w ซึ่งมี $var(w) = x_i$ เราจะเปลี่ยนบัพถัดไปเป็น $low(w)$ ในกรณีที่ b เป็น *'0'* หรือเปลี่ยนบัพถัดไปเป็น $high(w)$ ในกรณีที่ b เป็น *'1'* หลังจากผ่านกระบวนการนี้แล้วจะต้องใช้อัลกอริทึม *Reduce* เพื่อปรับแผนภาพให้ยังคงมีคุณสมบัติของแผนภาพตัดสินใจทวิภาคแบบอันดับเช่นเดิม

ตัวปฏิบัติการทางตรรกแบบสองอาร์กิวเมนต์ (two-argument logical operator) สามารถสร้างด้วยอัลกอริทึมที่ใช้เวลาเป็นเชิงเส้นกับขนาดของแผนภาพตัดสินใจทวิภาคแบบอันดับ โดยอัลกอริทึมนี้ใช้พื้นฐานจากการขยายของแชนนอน (Shannon expansion):

$$f = (\neg x \wedge f|_{x \leftarrow 0}) \vee (x \wedge f|_{x \leftarrow 1})$$

Bryant (1986) นำเสนออัลกอริทึมชื่อ *Apply* ซึ่งสามารถใช้คำนวณตัวปฏิบัติการทางตรรกดังจะได้อธิบายกันต่อไป ให้ ■ แทนตัวปฏิบัติการทางตรรกแบบสองอาร์กิวเมนต์ และให้ f และ f' เป็นฟังก์ชันแบบบูล และกำหนดให้

- v และ v' เป็นบัพรากในแผนภาพตัดสินใจทวิภาคแบบอันดับของ f และ f'
- $x = var(v)$ และ $x' = var(v')$

อัลกอริทึม *Apply* นิยามดังนี้

- ถ้า v และ v' เป็นบัพปลายทาง, แล้ว $f \blacksquare f' = value(v) \blacksquare value(v')$
- ถ้า $x = x'$, แล้วจะใช้การขยายของแชนนอน

$$f \blacksquare f' = (\neg x \wedge (f|_{x \leftarrow 0} \blacksquare f'|_{x \leftarrow 0})) \vee (x \wedge (f|_{x \leftarrow 1} \blacksquare f'|_{x \leftarrow 1}))$$

จะเห็นได้ว่ามีการแบ่งปัญหาออกเป็นปัญหาย่อยสองปัญหา ซึ่งจะถูกแก้ได้ด้วยวิธีแบบเรียกซ้ำ (recursive) ให้บัพรากของแผนภาพตัดสินใจทวิภาคแบบอันดับของคำตอบคือ w โดยที่ $var(w) = x$, $low(w)$ เป็นแผนภาพของสูตรทางตรรก $(f|_{x \leftarrow 0} \blacksquare f'|_{x \leftarrow 0})$, และ $high(w)$ เป็นแผนภาพของสูตรทางตรรก $(f|_{x \leftarrow 1} \blacksquare f'|_{x \leftarrow 1})$

- ถ้า $x < x'$ (อันดับน้อยกว่า), แล้ว $f' |_{x \leftarrow 0} = f' |_{x \leftarrow 1} = f'$ เพราะว่า f' ไม่ขึ้นกับตัวแปร x ดังนั้นในกรณีนี้ได้การขยายของแชนนอนเป็น

$$f \blacksquare f' = (\neg x \wedge (f |_{x \leftarrow 0} \blacksquare f')) \vee (x \wedge (f |_{x \leftarrow 1} \blacksquare f'))$$

ซึ่งแผนภาพตัดสินใจทวิภาคแบบอันดับของ $f \blacksquare f'$ สามารถสร้างได้ด้วยวิธีการแบบเรียกซ้ำเช่นเดียวกับกรณีที่สอง

- ถ้า $x > x'$ (อันดับมากกว่า), ใช้วิธีการคล้ายกับกรณีก่อนหน้านี้

ฟังก์ชันนิเสธ (negation) สามารถสร้างได้โดยอัลกอริทึม *Apply* โดยการเปลี่ยนค่าในบัพปลายทางของแผนภาพตัดสินใจทวิภาคแบบอันดับของฟังก์ชัน f ให้เป็นค่าตรงข้าม

เนื่องจากอัลกอริทึมเป็นแบบเรียกซ้ำ ซึ่งปัญหาหนึ่งจะถูกแบ่งเป็นปัญหาย่อยสองปัญหา จึงทำให้เวลาการทำงานมีโอกาสเป็นฟังก์ชันเอ็กซ์โพเนนเชียล (exponential function) ดังนั้นจึงจำเป็นต้องใช้เทคนิคบางอย่างช่วยจัดการ โดยเทคนิคหนึ่งได้แก่การใช้ตารางแฮช (hash table) ที่เรียกว่าแคชผลลัพธ์ (result cache) ซึ่งทำหน้าที่เก็บผลลัพธ์ที่ถูกคำนวณไว้แล้ว ดังนั้นก่อนที่จะเรียกอัลกอริทึมซ้ำ จะต้องตรวจสอบในแคชผลลัพธ์ดูก่อนว่าเคยคำนวณปัญหาย่อยนี้แล้วหรือไม่ ถ้าเคยคำนวณแล้วก็นำผลลัพธ์ที่เก็บไว้ไปใช้ได้เลย ถ้าไม่เคยจึงค่อยเรียกซ้ำอัลกอริทึม จากการใช้เทคนิคนี้ทำให้เวลาการทำงานลดลงเป็นเพียงฟังก์ชันพหุนาม (polynomial function) เท่านั้น

3.4.2 การแทนโครงสร้างคริปเก

ถ้า Q เป็นความสัมพันธ์บนเซต {0, 1} แล้ว Q สามารถถูกแทนด้วยแผนภาพตัดสินใจทวิภาคแบบอันดับของฟังก์ชันนี้ซึ่ง

$$f_Q(x_1, \ldots, x_n) = 1 \text{ ก็ต่อเมื่อ } Q(x_1, \ldots, x_n)$$

ให้ Q เป็นความสัมพันธ์บนเซตจำกัด D ซึ่งสมมุติว่า D มีสมาชิก 2^m ตัวโดยที่ $m > 1$ ในการแทนความสัมพันธ์ Q ด้วยแผนภาพตัดสินใจทวิภาคแบบอันดับ เราใช้ฟังก์ชัน

$\phi : \{0, 1\}^m \rightarrow D$ ซึ่งกำหนดความสัมพันธ์ระหว่างเวกเตอร์แบบบูลความยาว m ไปยังสมาชิกของเซต D จากการใช้ฟังก์ชันเข้ารหัส ϕ เราสามารถสร้างความสัมพันธ์แบบบูล Q' :

$$Q'(\mathbf{x}_1, \ldots, \mathbf{x}_n) = Q(\phi(\mathbf{x}_1), \ldots, \phi(\mathbf{x}_n))$$

โดยที่ $\mathbf{x}_i$ เป็นเวกเตอร์ของตัวแปรแบบบูล m ตัวที่ใช้เข้ารหัสตัวแปร x_i ซึ่งมีค่าอยู่ในเซต D นอกจากนั้น Q สามารถถูกแทนด้วยแผนภาพตัดสินใจทวิภาคแบบอันดับของฟังก์ชัน $f_{Q'}$ ของความสัมพันธ์ Q' จากวิธีการนี้ทำให้สามารถขยายความสัมพันธ์ไปบนโดเมนหลายเซต $D_1, \ldots, D_n$ นอกจากนั้นเนื่องจากเซตสามารถมองเป็นความสัมพันธ์ได้ ดังนั้นด้วยเทคนิคเดียวกันจึงสามารถแทนเซตด้วยแผนภาพตัดสินใจทวิภาคแบบอันดับได้

พิจารณาโครงสร้างคริปเก $M = (S, R, L)$ เพื่อที่จะแทนโครงสร้างนี้ด้วยแผนภาพตัดสินใจทวิภาคแบบอันดับ เราต้องแทนเซต S, ความสัมพันธ์ R, และฟังก์ชัน L สำหรับเซตของสถานะ S เราต้องทำการเข้ารหัสสถานะ โดยในที่นี้สมมุติว่ามีจำนวนสถานะอยู่ 2^m สถานะ กำหนดให้ $\phi : \{0, 1\}^m \rightarrow S$ เป็นฟังก์ชันเข้ารหัสซึ่งกำหนดความสัมพันธ์ระหว่างเวกเตอร์แบบบูลกับสถานะ จากวิธีการที่กล่าวมาแล้วโดยใช้ฟังก์ชันเข้ารหัส ϕ เราสามารถแทนเซต S ด้วยแผนภาพตัดสินใจทวิภาคแบบอันดับได้ สำหรับความสัมพันธ์การเปลี่ยนสถานะ R จะใช้ฟังก์ชันการเข้ารหัสเดียวกับที่ใช้กับเซต S ดังได้กล่าวแล้วในหัวข้อ 3.1 ว่าจำเป็นต้องมีตัวแปรสองเซต โดยเซตแรกใช้แทนสถานะปัจจุบัน และเซตที่สองใช้แทนสถานะถัดไป กำหนดให้ความสัมพันธ์ R ถูกแทนด้วยความสัมพันธ์แบบบูล $R'(\mathbf{x}, \mathbf{x}')$ แล้ว R จะถูกแทนด้วยแผนภาพตัดสินใจทวิภาคแบบอันดับของฟังก์ชัน $f_{R'}$ และสำหรับฟังก์ชัน L เราจะเปลี่ยนจากเดิมที่เป็นฟังก์ชันจากสถานะไปยังเซตย่อยของประพจน์เดี่ยว ไปเป็นฟังก์ชันจากประพจน์เดี่ยวไปเป็นเซตย่อยของสถานะแทนเพื่อความสะดวก สำหรับประพจน์เดี่ยว p ถูกกำหนดความสัมพันธ์กับเซตของสถานะซึ่งประพจน์นั้นเป็นจริง $\{s \mid p \in L(s)\}$ เซตของสถานะ L_p สามารถถูกแทนด้วยฟังก์ชันการเข้ารหัสโดยใช้เวกเตอร์แบบบูลเพื่อแทนเซตของประพจน์เดี่ยว ด้วยวิธีดังที่ได้นำเสนอเราสามารถแทนฟังก์ชัน L ด้วยแผนภาพตัดสินใจทวิภาคแบบอันดับได้

<u>ตัวอย่าง</u> พิจารณาโครงสร้างคริปเกในรูปที่ 3.10 เป็นโครงสร้างคริปเกที่มีสองสถานะ โดยมีตัวแปรสถานะสองตัวคือ a และ b (จากรูปในสถานะ s_1 ทั้งตัวแปร a และ b เป็นจริง และในสถานะ s_2 ตัวแปร a เป็นจริงแต่ตัวแปร b เป็นเท็จ) ในกรณีนี้เราต้องสร้างตัวแปรสถานะอีก

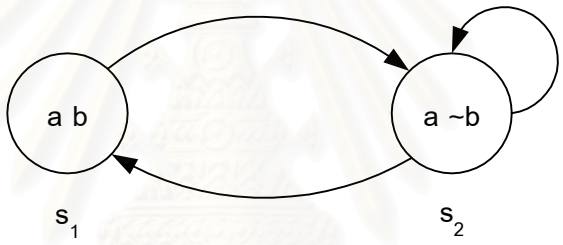
สองตัวคือ a' และ b' เพื่อใช้แทนสถานะถัดไป ดังนั้นการเปลี่ยนสถานะจากสถานะ s_1 ไปยังสถานะ s_2 แทนได้ด้วยสูตรแบบบูลนี้

$$(a \wedge b \wedge a' \wedge \neg b')$$

สูตรแบบบูลที่ใช้แทนความสัมพันธ์การเปลี่ยนสถานะทั้งหมดคือ

$$(a \wedge b \wedge a' \wedge \neg b') \vee (a \wedge \neg b \wedge a' \wedge \neg b') \vee (a \wedge \neg b \wedge a' \wedge b')$$

เห็นได้ว่ามีการเลือก (disjunction) ในสูตรอยู่สามตัวเนื่องจากในโครงสร้างคริปเกมีการเปลี่ยนสถานะอยู่สามกรณี และสูตรนี้สามารถถูกแปลงเป็นแผนภาพตัดสินใจทวิภาคแบบอันดับได้ด้วยวิธีที่แสดงในหัวข้อที่ 3.4.1



รูปที่ 3.10 โครงสร้างคริปเกสองสถานะ

3.5 การตรวจสอบแบบจำลองเชิงสัญลักษณ์

3.5.1 รูปแบบการแทนฟิกซ์พอยน์

ให้ $M = (S, R, L)$ เป็นโครงสร้างคริปเก เซต $P(S)$ ของเซต S คือแลตทิซ (lattice) ซึ่งมีคุณสมบัติเกี่ยวกับอันดับ สมาชิก S' แต่ละตัวของแลตทิซสามารถมองเป็นเพรดิเคต (predicate) บนเซต S ได้ โดยที่เพรดิเคตจะเป็น *True* บนบางสถานะใน S' แน่นอน สมาชิกที่น้อยที่สุด (least element) ในแลตทิซคือเซตว่างซึ่งเราอ้างถึงได้ด้วยค่า *False* และสมาชิกที่มากที่สุด (greatest element) ในแลตทิซคือเซต S ซึ่งบางทีก็อ้างถึงได้ด้วยค่า *True*

กำหนดให้ฟังก์ชันจาก $P(S)$ ไปยัง $P(S)$ ถูกเรียกว่าตัวแปลงเพรดิเคต (predicate transformer) โดยให้ $\tau : P(S) \rightarrow P(S)$ เป็นฟังก์ชัน แล้ว

1. τ มีคุณสมบัติโมโนโทนิก (monotonic) เมื่อ $P \subseteq Q$ แล้ว $\tau(P) \subseteq \tau(Q)$ ด้วย

2. τ มีคุณสมบัติต่อเนื่องของการยูเนียน ($\cup$-continuous) เมื่อ $P_1 \subseteq P_2 \subseteq \ldots$ แล้ว $\tau(\cup_i P_i) = \cup_i \tau(P_i)$ ด้วย

3. τ มีคุณสมบัติต่อเนื่องของการอินเตอร์เซคชัน ($\cap$-continuous) เมื่อ $P_1 \supseteq P_2 \supseteq \ldots$ แล้ว $\tau(\cap_i P_i) = \cap_i \tau(P_i)$ ด้วย

เราใช้ $\tau^i(Z)$ เพื่อแทนการใช้ฟังก์ชัน τ เป็นจำนวน i ครั้งกับเซต Z โดยมีนิยามว่า $\tau^0(Z) = Z$ และ $\tau^{i+1}(Z) = \tau(\tau^i(Z))$ ตัวแปลงเพรดิเคตโมโนโทนิก (monotonic predicate transformer) τ บนเซต $P(S)$ จะมีฟิกซ์พอยน์แบบน้อยที่สุด (least fixpoint) $\mu Z \,.\, \tau(Z)$ เสมอ และมีฟิกซ์พอยน์แบบมากที่สุด $\nu Z \,.\, \tau(Z)$ เสมอเช่นกัน โดยมีนิยามดังนี้

$\mu Z \,.\, \tau(Z) = \cap\{ Z \mid \tau(Z) \subseteq Z\}$ เมื่อ τ มีคุณสมบัติโมโนโทนิก และ $\mu Z \,.\, \tau(Z) = \cup_i \tau^i(\mathit{False})$ เมื่อ τ มีคุณสมบัติต่อเนื่องของการยูเนียน และ

$\nu Z \,.\, \tau(Z) = \cup\{ Z \mid \tau(Z) \supseteq Z\}$ เมื่อ τ มีคุณสมบัติโมโนโทนิก และ $\nu Z \,.\, \tau(Z) = \cap_i \tau^i(\mathit{True})$ เมื่อ τ มีคุณสมบัติต่อเนื่องของการอินเตอร์เซคชัน

ต่อไปจะได้แสดงคุณสมบัติของฟังก์ชันตัวแปลงเพรดิเคต ซึ่งจะได้ใช้ประโยชน์ต่อไปในหัวข้อนี้ ในที่นี้ไม่ได้นำเสนอการพิสูจน์ ถ้าหากสนใจสามารถหาอ่านได้จาก (Clarke, Grumberg และ Peled, 2001: 62)

Lemma 2 ถ้า S เป็นเซตจำกัด และ τ มีคุณสมบัติโมโนโทนิก, แล้ว τ จะมีคุณสมบัติต่อเนื่องของการยูเนียน และคุณสมบัติต่อเนื่องของการอินเตอร์เซคชันด้วย

Lemma 3 ถ้า τ มีคุณสมบัติโมโนโทนิก, แล้วสำหรับทุกค่า i, $\tau^i(\mathit{False}) \subseteq \tau^{i+1}(\mathit{False})$ และ $\tau^i(\mathit{True}) \supseteq \tau^{i+1}(\mathit{True})$

Lemma 4 ถ้า τ มีคุณสมบัติโมโนโทนิกและ S เป็นเซตจำกัด, แล้วจะมีจำนวนเต็ม i_0 ซึ่งสำหรับทุกค่า $j \geq i_0$, $\tau^j(\mathit{False}) = \tau^{i_0}(\mathit{False})$ และจะมีจำนวนเต็ม j_0 ซึ่งสำหรับทุกค่า $j \geq j_0$, $\tau^j(\mathit{True}) = \tau^{j_0}(\mathit{True})$

Lemma 5 ถ้า τ มีคุณสมบัติโมโนโทนิกและ S เป็นเซตจำกัด, แล้วจะมีจำนวนเต็ม i_0 ซึ่ง $\mu Z \,.\, \tau(Z) = \tau^{i_0}(False)$ และจะมีจำนวนเต็ม j_0 ซึ่ง $\nu Z \,.\, \tau(Z) = \tau^{j_0}(True)$

จากคุณสมบัติที่กล่าวมาทำให้ได้อัลกอริทึมที่ใช้คำนวณหาฟิกซ์พอยน์แบบน้อยที่สุดดังแสดงในรูปที่ 3.11 จากกระบวนการในรูปสามารถเขียนข้อความยืนยัน (assertion) ได้เป็น

$$(Q' = \tau(Q)) \wedge (Q' \subseteq \mu Z \,.\, \tau(Z))$$

จะเห็นได้ว่าในการวนซ้ำรอบที่ i จะได้ว่า $Q = \tau^{i-1}(False)$ และ $Q' = \tau^{i}(False)$ จากเลมมา (lemma) ที่ 3 ได้ว่า

$$False \subseteq \tau(False) \subseteq \tau^{2}(False) \subseteq \ldots$$

ดังนั้นจำนวนรอบในการวนซ้ำจะถูกจำกัดด้วยจำนวนสมาชิกในเซต S และเมื่อการวนซ้ำสิ้นสุดลงจะได้ว่า $Q = \tau(Q)$ ซึ่งทำให้ $Q \subseteq \mu Z \,.\, \tau(Z)$ และเนื่องจาก Q เป็นฟิกซ์พอยน์ซึ่งทำให้ $\mu Z \,.\, \tau(Z) \subseteq Q$ ดังนั้นสรุปได้ว่า $Q = \mu Z \,.\, \tau(Z)$ คำตอบที่ได้จากอัลกอริทึมจึงเป็นฟิกซ์พอยน์แบบน้อยที่สุด สำหรับการหาฟิกซ์พอยน์แบบมากที่สุดสามารถหาได้ด้วยวิธีที่คล้ายกันดังแสดงในรูปที่ 3.12 และด้วยวิธีการที่คล้ายกันจะสามารถแสดงให้เห็นได้ว่าผลลัพธ์ของอัลกอริทึมคือค่าของ $\nu Z \,.\, \tau(Z)$

```
function Lfp(Tau : PredicateTransfomer) : Predicate
        Q := False;
        Q' := Tau(Q);
        while (Q ≠ Q') do
                Q := Q';
                Q' := Tau(Q');
        end while;
        return(Q);
end function
```

รูปที่ 3.11 อัลกอริทึมที่ใช้หาฟิกซ์พอยน์แบบน้อยที่สุด

```
function Gfp(Tau : PredicateTransfomer) : Predicate
        Q := True;
        Q' := Tau(Q);
        while (Q ≠ Q') do
                Q := Q';
                Q' := Tau(Q');
        end while;
        return(Q);
end function
```

รูปที่ 3.12 อัลกอริทึมที่ใช้หาฟิกซ์พอยน์แบบมากที่สุด

ถ้าเราระบุสูตรของตรรกซีทีแอล f ด้วยเพรดิเคต $\{ s \mid M, s \models f \}$ ในเซต $P(S)$ แล้วตัวปฏิบัติการพื้นฐานของตรรกซีทีแอลสามารถนิยามโดยใช้ฟิกซ์พอยน์แบบน้อยที่สุดหรือแบบมากที่สุดของตัวแปลงเพรดิเคตดังนี้

- AF $f_1 = \mu Z . f_1 \vee \mathrm{AX}\, Z$
- EF $f_1 = \mu Z . f_1 \vee \mathrm{EX}\, Z$
- AG $f_1 = \nu Z . f_1 \vee \mathrm{AX}\, Z$
- EG $f_1 = \nu Z . f_1 \vee \mathrm{EX}\, Z$
- A$[f_1\ \mathrm{U}\ f_2] = \mu Z . f_2 \vee (f_1 \wedge \mathrm{AX}\, Z)$
- E$[f_1\ \mathrm{U}\ f_2] = \mu Z . f_2 \vee (f_1 \wedge \mathrm{EX}\, Z)$

จะเห็นได้ว่าฟิกซ์พอยน์แบบน้อยที่สุดจะสอดคล้องกับกรณีเป็นจริงบางสถานะ และฟิกซ์พอยน์แบบมากที่สุดจะสอดคล้องกับกรณีเป็นจริงในทุกสถานะ ดังนั้น AF f_1 หาได้โดยใช้ฟิกซ์พอยน์แบบน้อยที่สุด และ EG f_1 หาได้โดยใช้ฟิกซ์พอยน์แบบมากที่สุด

ต่อไปจะเป็นการพิสูจน์การหาฟิกซ์พอยน์ของ EG และ EU สำหรับตัวปฏิบัติการอื่นของตรรกซีทีแอลจะสามารถพิสูจน์ได้ในแนวทางเดียวกัน ในเลมมาที่ 6 ถึง 9 แสดงว่า EG $f_1 = \nu Z . f_1 \vee \mathrm{EX}\, Z$ เป็นจริง

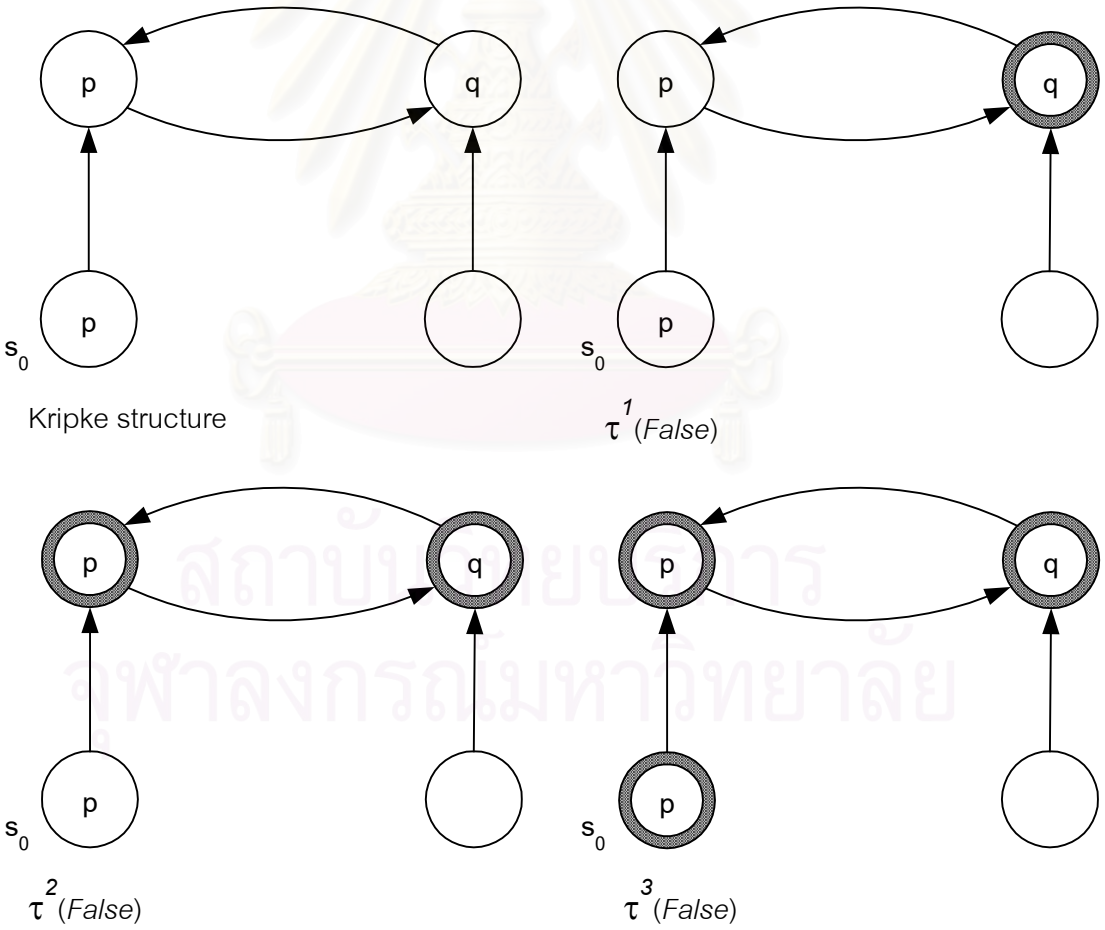
Lemma 6 $\tau(Z) = f_1 \vee \text{EX } Z$ มีคุณสมบัติโมโนโทนิก

Lemma 7 ให้ $\tau(Z) = f_1 \vee \text{EX } Z$ และให้ $\tau^{io}(True)$ เป็นลิมิตของลำดับ $True \supseteq \tau(True) \supseteq \ldots$ สำหรับทุกสถานะ $s \in S$, ถ้า $s \in \tau^{io}(True)$ แล้ว $s \models f_1$ และมีสถานะ s' ซึ่ง $(s, s') \in R$ และ $s' \in \tau^{io}(True)$

Lemma 8 EG f_1 คือฟิกซ์พอยน์ของฟังก์ชัน $\tau(Z) = f_1 \vee \text{EX } Z$

Lemma 9 EG f_1 คือฟิกซ์พอยน์แบบมากที่สุดของฟังก์ชัน $\tau(Z) = f_1 \vee \text{EX } Z$

Lemma 10 $\text{E}[f_1 \text{ U } f_2]$ คือฟิกซ์พอยน์แบบน้อยที่สุดของฟังก์ชัน $\tau(Z) = f_2 \vee (f_1 \wedge \text{EX } Z)$



รูปที่ 3.13 ลำดับของการคำนวณหา E[*p* U *g*]

ในรูปที่ 3.13 แสดงตัวอย่างการคำนวณหาเซตของสถานะซึ่งสอดคล้องกับ $E[p \mathbf{U} g]$ โดยใช้อัลกอริทึม *Lfp* (จากรูปในสถานะที่มีอักษร p อยู่หมายถึงว่าประพจน์ p เป็นจริงในสถานะนั้น ในสถานะที่มีอักษร q อยู่หมายถึงว่าประพจน์ q เป็นจริงในสถานะนั้น และในกรณีที่ไม่มีทั้งคู่หมายถึงว่าในสถานะนั้นทั้งประพจน์ p และ q ไม่เป็นจริง) ในกรณีนี้ฟังก์ชัน τ คือ

$$\tau(Z) = q \vee (p \wedge \mathbf{EX}\, Z)$$

ในรูปแสดงลำดับของ $\tau^i(False)$ ซึ่งลู่เข้าสู่ $E[p \mathbf{U} g]$ โดยสถานะที่ถูกระบายสีคือสถานะที่อยู่ใน $\tau^i(False)$ จากรูปจะเห็นได้ว่า $\tau^3(False) = \tau^4(False)$ ดังนั้น $E[p \mathbf{U} g] = \tau^3(False)$ และเพราะว่า s_0 อยู่ในเซต $\tau^3(False)$ ดังนั้นสรุปได้ว่า $M, s \models E[p \mathbf{U} g]$

3.5.2 การตรวจสอบแบบจำลองเชิงสัญลักษณ์ของตรรกซีทีแอล

3.5.2.1 สูตรคิวบีเอฟ

กำหนดให้เซต $V = \{v_0, \ldots, v_{n-1}\}$ เป็นเซตของตัวแปรประพจน์ (propositional variable) *QBF*(V) คือเซตที่เล็กที่สุดของสูตรคิวบีเอฟ (QBF – Quantified Boolean Formula) ซึ่งนิยามดังนี้

- ตัวแปรทุกตัวในเซต V เป็นสูตรคิวบีเอฟ
- ถ้า f และ g เป็นสูตรคิวบีเอฟ, แล้ว $\neg f$, $f \vee g$, $f \wedge g$ เป็นสูตรคิวบีเอฟด้วย
- ถ้า f เป็นสูตรคิวบีเอฟ และ $v \in V$,แล้ว $\exists vf$ และ $\forall vf$ เป็นสูตรคิวบีเอฟด้วย

การกำหนดค่าความจริงสำหรับ *QBF*(V) นิยามด้วยฟังก์ชัน $\sigma : V \rightarrow \{0, 1\}$ และถ้า $a \in \{0, 1\}$ แล้วเราจะใช้สัญลักษณ์ $\sigma\langle v \leftarrow a\rangle$ แทนการกำหนดค่าความจริงซึ่งนิยามดังนี้

$$\sigma\langle v \leftarrow a\rangle(w) = \begin{cases} a & \text{if } v = w \\ \sigma(w) & \text{otherwise} \end{cases}$$

ถ้า f เป็นสูตรคิวบีเอฟในเซต $QBF(V)$ และ σ เป็นการกำหนดค่าความจริง เราจะเขียน $\sigma \models f$ เพื่อบอกว่า f เป็นจริงเมื่อกำหนดค่าความจริงด้วย σ และความสัมพันธ์ $\models$ นิยามได้ดังนี้

- $\sigma \models v$ ก็ต่อเมื่อ $\sigma(v) = 1$
- $\sigma \models \neg f$ ก็ต่อเมื่อ $\sigma \not\models f$
- $\sigma \models f \vee g$ ก็ต่อเมื่อ $\sigma \models f$ หรือ $\sigma \models g$
- $\sigma \models f \wedge g$ ก็ต่อเมื่อ $\sigma \models f$ และ $\sigma \models g$
- $\sigma \models \exists v f$ ก็ต่อเมื่อ $\sigma\langle v \leftarrow 0\rangle \models f$ หรือ $\sigma\langle v \leftarrow 1\rangle \models f$
- $\sigma \models \forall v f$ ก็ต่อเมื่อ $\sigma\langle v \leftarrow 0\rangle \models f$ และ $\sigma\langle v \leftarrow 1\rangle \models f$

ตัวบ่งปริมาณ (quantifier) ในสูตรคิวบีเอฟสามารถสร้างได้จากการรวมอัลกอริทึม *Restrict* และ *Apply* ซึ่งได้เคยแสดงไว้แล้ว

- $\exists v f = f|_{x \leftarrow 0} \vee f|_{x \leftarrow 1}$
- $\forall v f = f|_{x \leftarrow 0} \wedge f|_{x \leftarrow 1}$

ตัวปฏิบัติการผลคูณเชิงสัมพันธ์ (relational product) เป็นตัวปฏิบัติการที่ถูกใช้บ่อยมากที่สุดตัวหนึ่ง ตัวปฏิบัติการนี้อยู่ในรูปดังนี้

$$\exists \mathbf{v} [f(\mathbf{v}, \mathbf{w}) \wedge g(\mathbf{v}, \mathbf{x})]$$

3.5.2.2 อัลกอริทึมของการตรวจสอบแบบจำลองเชิงสัญลักษณ์

การตรวจสอบแบบจำลองเชิงสัญลักษณ์สร้างด้วยการใช้อัลกอริทึมหลักที่ชื่อ *Check* ซึ่งรับอินพุตเป็นสูตรของตรรกชีทีแอล และให้เอาต์พุตเป็นแผนภาพตัดสินใจทวิภาคแบบอันดับซึ่งแทนสถานะของระบบซึ่งสอดคล้องกับสูตรดังกล่าว โดยอัลกอริทึม *Check* นิยามบนโครงสร้างของตรรกชีทีแอลดังนี้ ถ้า f เป็นประพจน์เดี่ยว a แล้ว *Check(f)* คือแผนภาพตัดสินใจทวิภาคแบบอันดับซึ่งแทนเซตของสถานะซึ่งสอดคล้องกับประพจน์ a และถ้า $f = f_1 \wedge f_2$ หรือ $f = \neg f_1$

แล้ว *Check(f)* สร้างได้โดยใช้อัลกอริทึม *Apply* โดยใช้*Check*(f_1) และ *Check*(f_2) เป็นอินพุตของอัลกอริทึม สูตรในรูป **EX** *f*, **E**[*f* **U** *g*],และ **EG** *f* สามารถทวนสอบได้โดย

$$Check(\mathbf{EX}\ f) = CheckEX(Check(f))$$

$$Check(\mathbf{E}[f\ \mathbf{U}\ g]) = CheckEU(Check(f), Check(g))$$

$$Check(\mathbf{EG}\ f) = CheckEG(Check(f))$$

สังเกตว่ากระบวนการด้านบนใช้แผนภาพตัดสินใจทวิภาคแบบอันดับเป็นอาร์กิวเมนต์ ขณะที่อัลกอริทึม *Check* ใช้สูตรของตรรกซีทีแอลเป็นอาร์กิวเมนต์ ในกรณีของตรรกซีทีแอลในรูป $f \vee g$ หรือ $\neg f$ เราสามารถจัดการได้ด้วยอัลกอริทึมที่ใช้จัดการกับตัวเชื่อมแบบบูล (boolean connective) ของแผนภาพตัดสินใจทวิภาคแบบอันดับ และเนื่องจากตัวปฏิบัติการเชิงเวลาอื่นของตรรกซีทีแอลสามารถเขียนโดยใช้ตัวปฏิบัติการที่นิยามด้านบน ดังนั้นนิยามของอัลกอริทึม *Check* ที่กล่าวมาจึงครอบคลุมสูตรทุกแบบที่เป็นไปได้ของตรรกซีทีแอล

อัลกอริทึม *CheckEX* สามารถสร้างได้อย่างตรงไปตรงมาจากนิยามของ **EX** *f* ซึ่งจะเป็นจริงในสถานะนั้นเมื่อสถานะถัดไปมี *f* เป็นจริง

$$CheckEX(f(\mathbf{v})) = \exists \mathbf{v}'\ [f(\mathbf{v}') \wedge R(\mathbf{v}, \mathbf{v}')]$$

โดยที่ $R(\mathbf{v}, \mathbf{v}')$ เป็นแผนภาพตัดสินใจทวิภาคซึ่งแทนความสัมพันธ์การเปลี่ยนสถานะ ถ้าเรามีแผนภาพตัดสินใจทวิภาคแบบอันดับของ *f* และ *R* เราสามารถคำนวณหาแผนภาพตัดสินใจทวิภาคแบบอันดับของ $\exists \mathbf{v}'\ [f(\mathbf{v}') \wedge R(\mathbf{v}, \mathbf{v}')]$ ได้โดยใช้ตัวปฏิบัติการของสูตรคิวบีเอฟ

อัลกอริทึม CheckEU สามารถสร้างได้โดยใช้ฟิกซ์พอยน์แบบน้อยที่สุดของตัวปฏิบัติการของตรรกซีทีแอล ดังที่นิยามไว้แล้วในหัวข้อที่ 3.5.1 ดังนี้

$$\mathbf{E}[f_1\ \mathbf{U}\ f_2] = \mu Z\ .\ f_2 \vee (f_1 \wedge \mathbf{EX}\ Z)$$

เราใช้อัลกอริทึม *Lfp* เพื่อหาฟิกซ์พอยน์ของลำดับ $Q_0, Q_1, \ldots, Q_i, \ldots$ ซึ่งจะลู่เข้าสู่ $\mathbf{E}[f\ \mathbf{U}\ g]$ โดยใช้เวลาการทำงานจำกัด ถ้าเรามีแผนภาพตัดสินใจทวิภาคแบบอันดับของ *f*, *g*, และ Q_i เราสามารถตรวจสอบว่าลำดับลู่เข้าแล้วหรือไม่ด้วยการเปรียบเทียบแผนภาพดังกล่าว

และเมื่อ $Q_i = Q_{i+1}$ อัลกอริทึม *Lfp* จะหยุดทำงาน เซตของสถานะซึ่งสอดคล้องกับสูตร $\mathsf{E}[f\ \mathsf{U}\ g]$ จะถูกแทนด้วยแผนภาพตัดสินใจทวิภาคแบบอันดับของ Q_i

อัลกอริทึม *CheckEG* สามารถสร้างโดยใช้วิธีการที่คล้ายกัน โดยในที่นี้ใช้ฟิกซ์พอยน์แบบมากที่สุดของตัวปฏิบัติการ EG ซึ่งนิยามไว้แล้วดังนี้

$$\mathsf{EG}\ f_1 = \nu Z\ .\ f_1 \wedge \mathsf{EX}\ Z$$

ถ้าเรามีแผนภาพตัดสินใจทวิภาคแบบอันดับของ *f* แล้วอัลกอริทึม *Gfp* สามารถจะคำนวณหาแผนภาพตัดสินใจทวิภาคซึ่งแทนเซตของสถานะซึ่งสอดคล้องกับ EG *f*

ในบทนี้ได้นำเสนอพื้นฐานทางทฤษฎีที่จำเป็นเพื่อใช้ทำความเข้าใจการทำงานของการตรวจสอบแบบจำลองเชิงสัญลักษณ์ โดยจุดเด่นของวิธีการนี้คือการแทนระบบโดยใช้แผนภาพตัดสินใจทวิภาคแบบอันดับ ซึ่งทำให้ไม่จำเป็นต้องสร้างกราฟของโครงสร้างคริปเก เป็นผลให้ประหยัดหน่วยความจำลงได้มาก จากเหตุผลดังกล่าวส่งผลให้วิธีการตรวจสอบแบบจำลองเชิงสัญลักษณ์สามารถทวนสอบวงจรที่มีขนาดใหญ่ ซึ่งไม่สามารถทวนสอบด้วยวิธีการตรวจสอบแบบจำลองธรรมดา

3.6 โปรแกรมคาเดนค์เอสเอมวี

ในวิทยานิพนธ์นี้ใช้โปรแกรมคาเดนค์เอสเอมวี (Cadence SMV - Cadence Symbolic Model Verifier) (McMillan, 1998) ซึ่งทำงานโดยใช้เทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์ และโปรแกรมนี้จะรับอินพุตด้วยภาษาเอสเอมวี (SMV language) ซึ่งเป็นภาษาที่ถูกกำหนดขึ้นเพื่อใช้กับการทวนสอบโดยเฉพาะ ภาษานี้มีลักษณะคล้ายกับภาษาการพรรณนาฮาร์ดแวร์ (hardware description language) ทำให้การเขียนรหัสต้นฉบับด้วยภาษาเอสเอมวีไม่เป็นเรื่องยากนักเนื่องจากภาษามีความเหมาะสมกับการออกแบบวงจรฮาร์ดแวร์อยู่แล้ว อีกประการหนึ่งภาษานี้สามารถเขียนข้อกำหนดเป็นตรรกซีทีแอล โดยสามารถใช้ตรวจสอบคุณสมบัติเชิงเวลาของวงจรได้ โปรแกรมคาเดนค์เอสเอมวีจะสร้างโครงสร้างคริปเกซึ่งเป็นชนิดหนึ่งของแผนภาพการเปลี่ยนสถานะจากรหัสต้นฉบับภาษาเอสเอมวี และมีการใช้แผนภาพตัดสินใจทวิภาคเพื่อแทนความสัมพันธ์การเปลี่ยนสถานะและทำการประมวลผลบนแผนภาพดังกล่าว ซึ่งการใช้แผนภาพนี้เพื่อแทนวงจรถือว่าเป็นจุดเด่นที่สำคัญของความสำเร็จของเทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์

บทที่ 4

หน่วยประมวลผลแบบฝังตัว

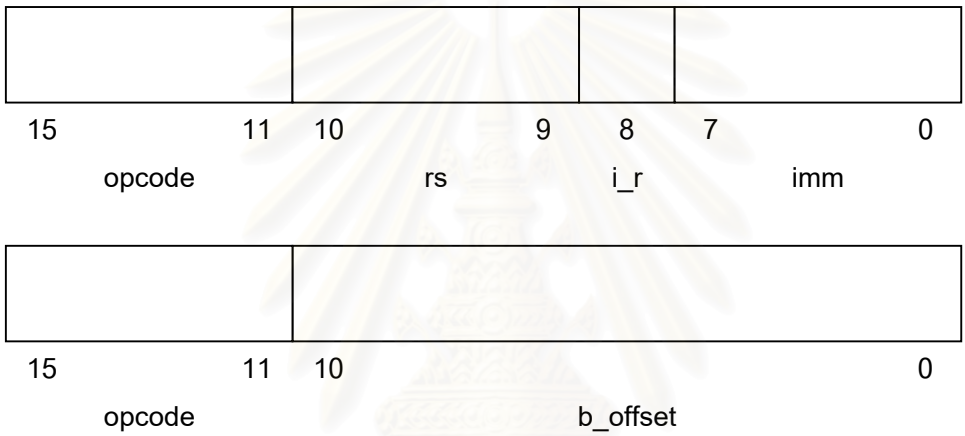
ในบทนี้จะกล่าวถึงรายละเอียดของหน่วยประมวลผลซึ่งถูกทวนสอบในวิทยานิพนธ์นี้ โดยในหัวข้อที่ 4.1 จะกล่าวถึงสถาปัตยกรรมชุดคำสั่งซึ่งเป็นสิ่งที่ผู้ใช้หน่วยประมวลผลจะสามารถมองเห็นได้ และในหัวข้อที่ 4.2 จะกล่าวถึงรายละเอียดการออกแบบภายในหน่วยประมวลผลซึ่งมีรายละเอียดเพียงพอที่จะทำความเข้าใจการออกแบบของหน่วยประมวลผลได้

4.1 สถาปัตยกรรมชุดคำสั่ง

หน่วยประมวลผลที่วิทยานิพนธ์นี้ทำการทวนสอบเป็นหน่วยประมวลผลที่ถูกสร้างขึ้นเพื่อใช้ในระบบเว็บเซิร์ฟเวอร์แบบฝังตัว (Piromsopa, 2000) ซึ่งเดิมนั้นออกแบบโดยใช้ไมโครคอนโทรลเลอร์เอมซีเอส-51 (MCS-51) ระบบเว็บเซิร์ฟเวอร์แบบฝังตัวนี้สามารถใช้เป็นอุปกรณ์ควบคุมระยะไกลผ่านอินเตอร์เน็ตได้ และเนื่องจากต้องการให้ทั้งระบบมีขนาดเล็กและราคาประหยัด ทำให้หน่วยประมวลผลที่ออกแบบขึ้นเพื่องานนี้มีขนาดเล็กและไม่ซับซ้อน หน่วยประมวลผลตัวนี้ทำงานแบบลำดับโดยไม่ใช้เทคนิคการทำงานแบบสายท่อ (pipeline technique) และไม่สามารถรองรับสัญญาณขัดจังหวะ (interruption signal) เพื่อให้มีขนาดเล็กและไม่ซับซ้อน การออกแบบทำในระดับการถ่ายโอนเรจิสเตอร์ซึ่งเป็นระดับที่สามารถนำไปใช้สังเคราะห์วงจรได้ โดยจะบอกรายละเอียดของวงจรอย่างชัดเจน เช่นการเปลี่ยนแปลงของสัญญาณควบคุมภายในวงจร และ รายละเอียดขององค์ประกอบภายในวงจร เป็นต้น แต่ในระดับนี้ก็ไม่ถึงกับระบุรายละเอียดในระดับเกต (gate level) ซึ่งจะทำให้การออกแบบมีรายละเอียดมากเกินไป และในการออกแบบหน่วยประมวลผลตัวนี้ใช้ภาษาเวอริล็อกเฮชดีแอล (Verilog HDL - Verilog Hardware Description Language)

หน่วยประมวลผลนี้ใช้สถาปัตยกรรมแบบโหลด/สโตร (load/store architecture) ซึ่งการคำนวณจะกระทำกับค่าในเรจิสเตอร์ (register) เท่านั้นและจะมีเฉพาะคำสั่งโหลดและสโตรเท่านั้นที่ติดต่อกับหน่วยความจำ และหน่วยประมวลผลนี้เป็นหน่วยประมวลผลขนาด 16 บิต ซึ่งหมายถึงเรจิสเตอร์มีขนาด 16 บิต นอกจากนั้นยังมีแอดเดรสบัส (address bus) และดาต้าบัส (data bus) ขนาด 16 บิตด้วย หน่วยประมวลผลตัวนี้ออกแบบให้หน่วยความจำโปรแกรม

(program memory) กับหน่วยความจำข้อมูล (data memory) แยกจากกัน โดยหน่วยความจำข้อมูลจะใช้แรม (RAM-Random Access Memory) และหน่วยความจำโปรแกรมจะใช้รอม (ROM-Read Only Memory) คำสั่งของหน่วยประมวลผลนี้แบ่งได้เป็น 3 กลุ่ม คือ กลุ่มคำสั่งคำนวณ (computation instruction), กลุ่มคำสั่งกระโดด (jump instruction) และ กลุ่มคำสั่งโหลด/สโตร (load/store instruction) โดยในตารางที่ 4.1 ได้แสดงชุดคำสั่งของหน่วยประมวลผลไว้ และในหน่วยประมวลผลมีเรจิสเตอร์ขนาด 16 บิต อยู่ 4 ตัวคือ *A*, *B*, *T* (temporary), และ *SP* (stack pointer) ทั้งยังมีตัวบ่งชี้ (flag) อยู่ 2 ตัวคือ ตัวบ่งชี้การทด (carry flag) และตัวบ่งชี้ค่าศูนย์ (zero flag)



รูปที่ 4.1 รูปแบบของคำสั่ง

ในรูปที่ 4.1 แสดงรูปแบบของคำสั่งซึ่งมีขนาด 16 บิต โดยมีรูปแบบของคำสั่งอยู่สองแบบคือแบบ (a) และ (b) รูปแบบ (a) ใช้กับคำสั่งในกลุ่มคำสั่งคำนวณและกลุ่มคำสั่งโหลด/สโตร สำหรับรูปแบบ (b) ใช้กับคำสั่งในกลุ่มคำสั่งกระโดด สัญญาณแต่ละตัวที่แสดงในรูปมีความหมายดังนี้

- *opcode*: ใช้ระบุว่าคำสั่งนี้คือคำสั่งอะไร จากตารางที่ 4.1 ถึงแม้ว่าจะมีคำสั่งอยู่ทั้งหมด 27 คำสั่ง แต่เป็นคำสั่งที่แตกต่างกันเพียง 20 คำสั่งดังแสดงในตารางที่ 4.2

- *rs* (source and destination register): ใช้ระบุถึงเรจิสเตอร์ตัวที่ใช้ในการคำนวณ โดยที่ *'0'=A*, *'1'=B*, *'2'=SP*, และ *'3'=T*

- *i_r* (instruction mode): สำหรับคำสั่งคำนวณซึ่งได้แก่คำสั่ง ADD, SUB, AND, ORR, และ XOR ถ้าเป็น *'0'* หมายถึงให้ใช้เรจิสเตอร์ *T* เพื่อคำนวณร่วมกับค่าในเรจิสเตอร์ที่ระบุในค่าของ *rs* แต่ถ้าเป็น *'1'* หมายถึงให้ใช้ค่าจาก *imm* เพื่อคำนวณแทน สำหรับคำสั่ง LDB และ LDS จะมีค่า *opcode* เหมือนกันคือเป็นคำสั่ง LD โดยที่ถ้า *i_r* เป็น *'0'* คือคำสั่ง LDS (Load from mem[SP] to [r]) และถ้า *i_r* เป็น *'1'* คือคำสั่ง LDB (Load from mem[B] to [r]) และสำหรับคำสั่ง STB และ STS จะมีค่า *opcode* เหมือนกันคือเป็นคำสั่ง ST โดยที่ถ้า *i_r* เป็น *'0'* คือคำสั่ง STS (Store [T] to mem[SP]) และถ้า *i_r* เป็น *'1'* คือคำสั่ง STB (Store [T] to mem[B])

- *imm* (immediate value): ค่าคงที่ซึ่งถูกระบุในคำสั่ง ซึ่งจะถูกใช้ในการคำนวณของคำสั่งคำนวณกรณีที่สัญญาณ *i_r* เป็น *'1'*

- *b_offset* (base offset): ค่าคงที่ซึ่งใช้เป็นเลขที่อยู่ (address) ในคำสั่งกลุ่มกระโดด

คำอธิบายเกี่ยวกับคำสั่งของหน่วยประมวลผลได้อธิบายไว้ชัดเจนแล้วในตารางที่ 4.1 คำสั่งที่คำอธิบายยังไม่ชัดเจนคือคำสั่ง LUI (Load upper immediate to [r]) ซึ่งหมายถึงว่าให้นำค่า *imm* ซึ่งมีขนาด 8 บิต ไปใส่ในเรจิสเตอร์ที่ถูกระบุไว้ในบิตที่ 8 ถึงบิตที่ 15

ในสถาปัตยกรรมชุดคำสั่งกำหนดให้คำสั่งที่มีผลต่อตัวบ่งชี้การทดได้แก่ CLC, STC, ADD, ADDI, SUB, SUBI, ROL, และ ROR ขณะที่คำสั่งซึ่งมีผลต่อตัวบ่งชี้ค่าศูนย์ได้แก่ ADD, ADDI, SUB, SUBI, AND, ANDI, ORR, ORRI, XOR, XORI, ROL, ROR, COM, และ LUI

ตารางที่ 4.1 ชุดคำสั่งของหน่วยประมวลผล

(r = register[A,B,T,SP], i = immediate, ad = address)

Instruction		Description
ADD	r	[r] = [r] + [T]
ADDI	r, i	[r] = [r] + immediate
SUB	r	[r] = [r] - [T]
SUBI	r, i	[r] = [r] - immediate
AND	r	[r] = [r] & [T]
ANDI	r, i	[r] = [r] & immediate
ORR	r	[r] = [r] | [T]
ORRI	r, i	[r] = [r] | immediate
XOR	r	[r] = [r] ^ [T]
XORI	r, i	[r] = [r] ^ immediate
COM	r	[r] = ~[r]
ROL	r	Rotate left [r] by c flag
ROR	r	Rotate right [r] by c flag
NOP		No Operation
LUI	r	Load upper immediate to [r]
CLC		Clear carry flag
STC		Set carry flag
JNZ	ad	Jump if not zero(z=0) to [PC]+1+ad
JNC	ad	Jump if not carry(c=0) to [PC]+1+ad
JMP	ad	Unconditional jump to [PC]+1+ad
LDB	r	Load from mem[B] to [r]
LDS	r	Load from mem[SP] to [r]
STB		Store [T] to mem[B]
STS		Store [T] to mem[SP]
LPC		Load from mem[SP] to [PC]
SPC		Store [PC]+1 to mem[SP]
R2T	r	Move [r] to [T]

ตารางที่ 4.2 ค่าของ *opcode* ของแต่ละคำสั่ง

คำสั่ง	ค่าของ *opcode* (ฐานสิบ)
NOP	0
JNZ	1
JNC	2
JMP	3
CLC	4
STC	5
R2T	7
ADD	8
SUB	9
AND	10
ORR	11
XOR	12
COM	13
ROL	14
ROR	15
LUI	16
LD	17
LPC	19
ST	25
SPC	27

4.2 รายละเอียดภายในหน่วยประมวลผล

หน่วยประมวลผลตัวนี้ถูกออกแบบในระดับการถ่ายโอนเรจิสเตอร์ด้วยภาษาเวอริล็อก รูปที่ 4.2 แสดงสายสัญญาณที่ต่อระหว่างหน่วยประมวลผลและหน่วยความจำซึ่งได้แก่ รอมและแรม โดยที่รอมทำหน้าที่เป็นหน่วยความจำโปรแกรมซึ่งเก็บโปรแกรมที่ใช้ควบคุมการทำงานของหน่วยประมวลผล ขณะที่แรมทำหน้าที่เป็นหน่วยความจำข้อมูลซึ่งเก็บข้อมูลที่ถูกใช้ในการคำนวณ หน่วยประมวลผลถูกแบ่งออกเป็นสองส่วนหลักคือ ส่วนทางเดินข้อมูล (datapath) และ หน่วยควบคุม (control unit) ส่วนทางเดินข้อมูลเป็นส่วนที่ทำงานกับข้อมูลจริงประกอบด้วยวงจรเชิงผสม (combinational circuit) และเรจิสเตอร์ต่างๆ สำหรับหน่วยควบคุมมีลักษณะเป็นเครื่องสถานะจำกัด (finite state machine) ซึ่งสร้างสัญญาณเพื่อควบคุมส่วนทางเดินข้อมูล

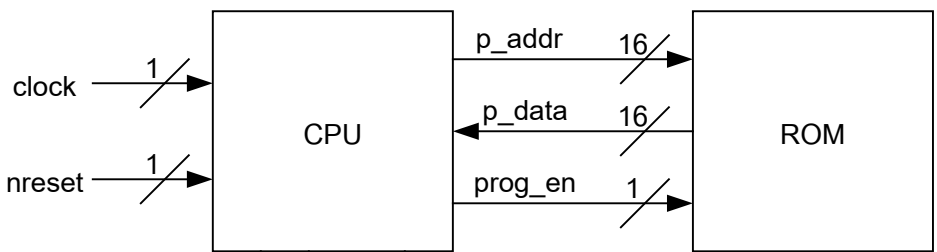
4.2.1 ส่วนทางเดินข้อมูล

ในรูปที่ 4.3 แสดงรายละเอียดภายในส่วนทางเดินข้อมูล สำหรับ *IR*, *PC*, *REGFILE* จะมีสัญญาณ *clock* และ *reset* ต่ออยู่ด้วยแต่ไม่ได้แสดงไว้ในรูป โดยที่กำหนดให้ *reset = ~ nreset*

ส่วนประกอบที่สำคัญภายในส่วนทางเดินข้อมูลได้แก่

- *IR* (Instruction Register): เป็นเรจิสเตอร์ที่ใช้เก็บคำสั่งปัจจุบัน โดยในรูปที่ 4.4 แสดงแผนภาพบล็อก (block diagram) ของส่วนนี้ สัญญาณควบคุมที่ใช้คือ *ir_ld* ซึ่งเมื่อเป็น *'1'* จะเก็บค่าจากอินพุตไปเก็บในเรจิสเตอร์ สำหรับเอาต์พุตของส่วนนี้จะต่อโดยตรงกับเรจิสเตอร์ภายในทำให้สามารถอ่านค่าได้ตลอดเวลาโดยไม่ต้องใช้สัญญาณควบคุม

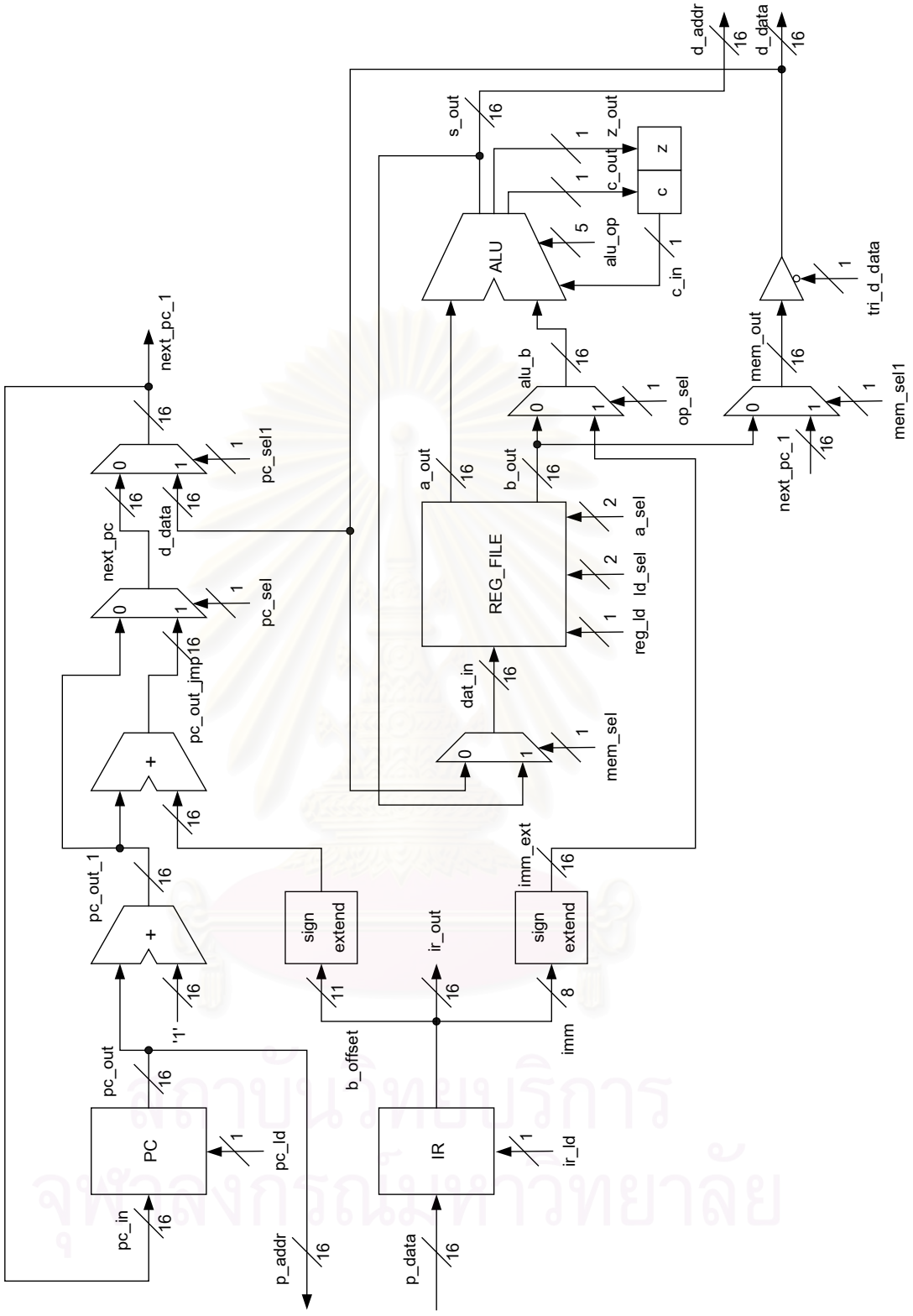
- *PC* (Program Counter): เป็นเรจิสเตอร์ที่ใช้เก็บตำแหน่งที่อยู่ในหน่วยความจำโปรแกรมของคำสั่งปัจจุบัน ในรูปที่ 4.5 แสดงแผนภาพบล็อกของส่วนนี้ สัญญาณควบคุมที่ใช้คือ *pc_ld* ซึ่งเมื่อเป็น *'1'* จะเก็บค่าจากอินพุตไปเก็บในเรจิสเตอร์ สำหรับเอาต์พุตของส่วนนี้จะต่อโดยตรงกับเรจิสเตอร์ภายในตลอดเวลา



รูปที่ 4.2 สายสัญญาณระหว่างหน่วยประมวลผลกับหน่วยความจำ

- *REGFILE* (Register File): เป็นกลุ่มของเรจิสเตอร์ที่สามารถเห็นได้ในระดับสถาปัตยกรรมชุดคำสั่งซึ่งได้แก่ *A*, *B*, *SP*, และ *T* ในรูปที่ 4.6 แสดงแผนภาพบล็อกของกลุ่มเรจิสเตอร์ สัญญาณควบคุมได้แก่ *reg_ld*, *ld_sel*, และ *a_sel* สัญญาณ *reg_ld* เมื่อเป็น *'1'* หมายถึงจะเก็บค่าจากอินพุตไปใส่ในเรจิสเตอร์ที่ถูกระบุโดยสัญญาณ *ld_sel* ซึ่งกำหนดว่า *'0'=A*, *'1'=B*, *'2'=SP*, และ *'3'=T* เอาต์พุตของกลุ่มเรจิสเตอร์มีสองชุดคือ *a_out* และ *b_out* เอาต์พุตชุดแรก *a_out* จะถูกควบคุมด้วยสัญญาณ *a_sel* เพื่อเลือกค่าที่จะออกมาที่เอาต์พุตชุดนี้ นิยามของค่า *a_sel* กำหนดเช่นเดียวกับ *ld_sel* สำหรับเอาต์พุตชุดที่สอง *b_out* จะต่อกับเรจิสเตอร์ *T* ไว้ตลอดเวลา

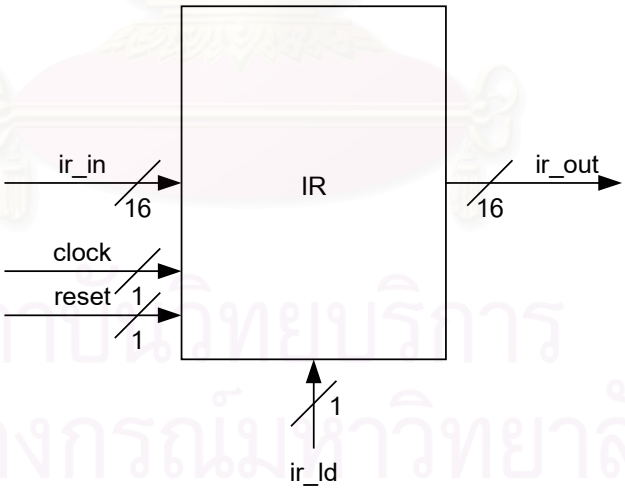
- *ALU* (Arithmetic Logic Unit): เป็นวงจรเชิงผสมที่ใช้คำนวณค่าสำหรับคำสั่งต่างๆ โดยในรูปที่ 4.7 แสดงแผนภาพบล็อกของส่วนนี้ สัญญาณควบคุมคือ *alu_op* ซึ่งมีนิยามดังแสดงในตารางที่ 4.3



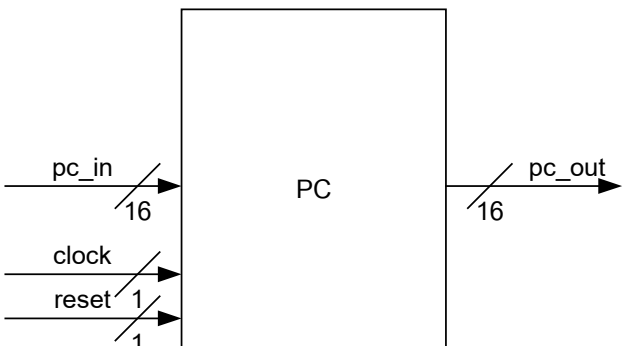
รูปที่ 4.3 รายละเอียดภายในส่วนทางเดินข้อมูลของหน่วยประมวลผล

ตารางที่ 4.3 นิยามของสัญญาณ *alu_op*

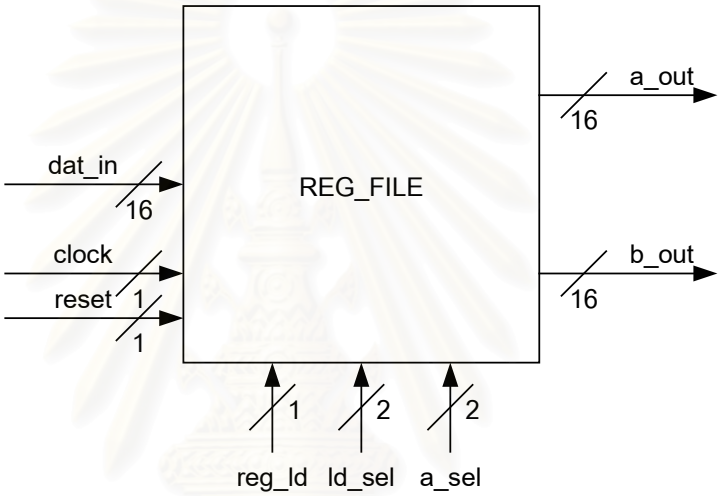
คำสั่ง	ค่าของ *alu_op* (เลขฐานสอง)
SUB	00101
AND	00000
ORR	00001
XOR	00010
COM	00011
ROL	10110
ROR	11110
LUI	01111
R2T	00110



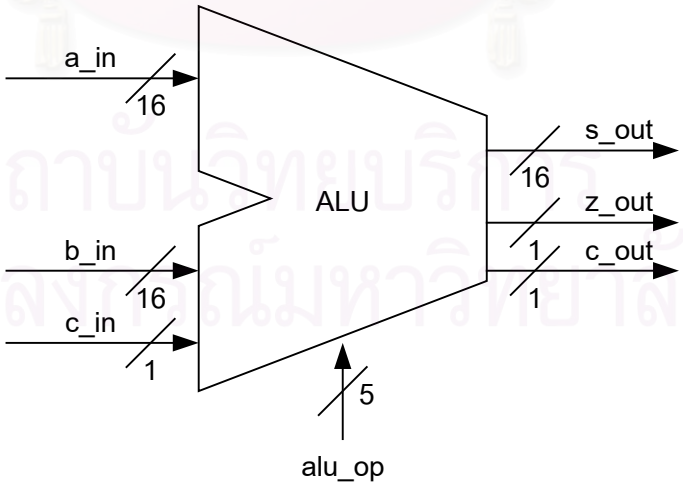
รูปที่ 4.4 แผนภาพบล็อกของเรจิสเตอร์ *IR*



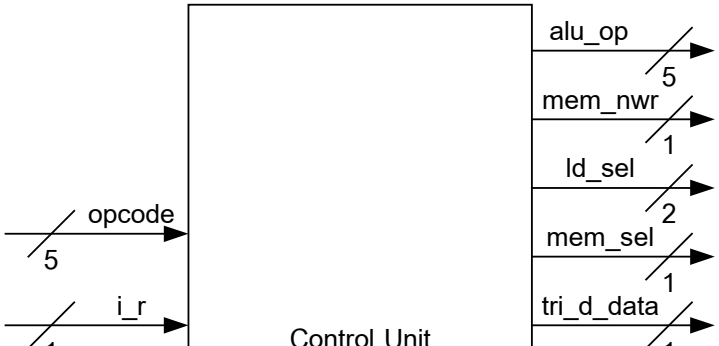
รูปที่ 4.5 แผนภาพบล็อกของเรจิสเตอร์ *PC*



รูปที่ 4.6 แผนภาพบล็อกของกลุ่มเรจิสเตอร์



รูปที่ 4.7 แผนภาพบล็อกของ *ALU*



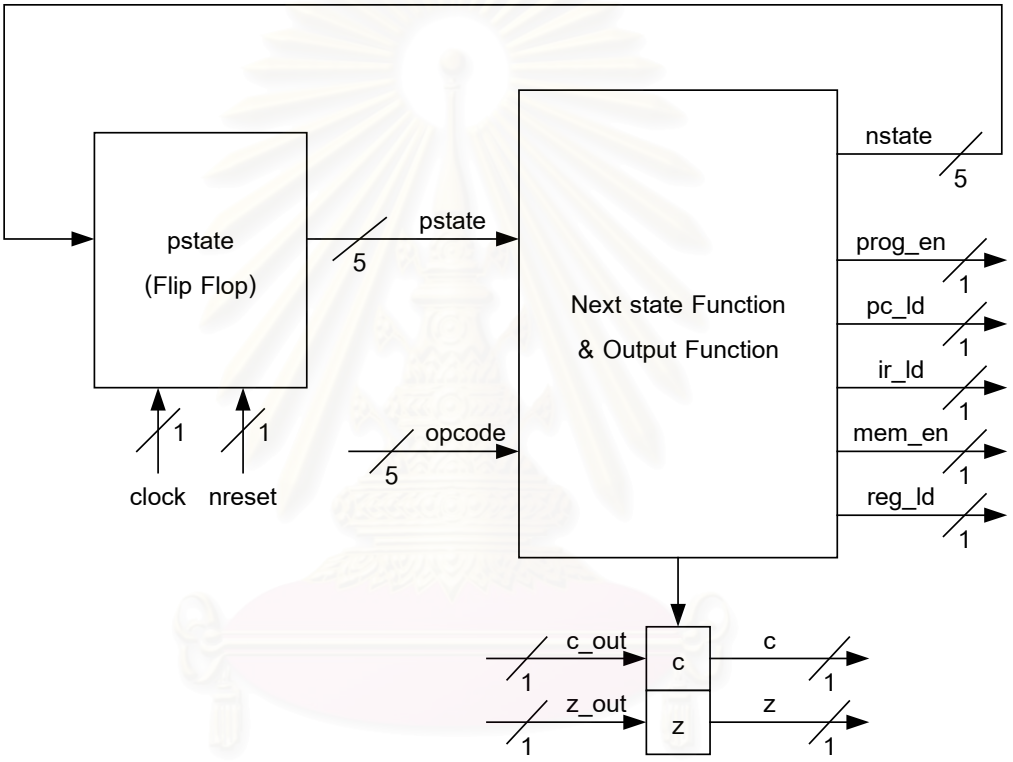
รูปที่ 4.8 แผนภาพบล็อกของส่วนที่เป็นวงจรเชิงผสมของหน่วยควบคุม

4.2.2 หน่วยควบคุม

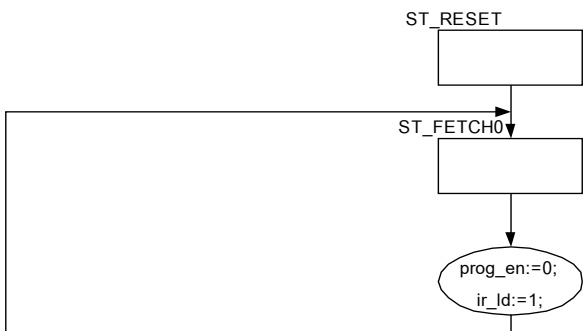
ในหน่วยควบคุมของหน่วยประมวลผลประกอบด้วยสองส่วนคือ ส่วนที่เป็นเครื่องสถานะจำกัดและส่วนที่เป็นวงจรเชิงผสม ก่อนอื่นขอนำเสนอส่วนที่เป็นวงจรเชิงผสมก่อน รูปที่ 4.8 แสดงแผนภาพบล็อกของส่วนนี้ ในส่วนนี้จะทำการกำหนดค่าสัญญาณควบคุมซึ่งได้แก่ *alu_op, mem_nwr, ld_sel, mem_sel, tri_d_data, mem_sel1, pc_sel1, a_sel, op_sel,* และ *pc_sel* สัญญาณเหล่านี้มีตัวหนึ่งที่ไม่ได้ปรากฏในรูปที่ 4.3 คือสัญญาณ *mem_nwr* ซึ่งเป็นสัญญาณที่ใช้ควบคุมการอ่านเขียนค่าในแรม โดยเมื่อเป็น *'0'* จะหมายถึงเขียนค่า และเมื่อเป็น *'1'* จะหมายถึงอ่านค่า ในหัวข้อ ก.1 ของภาคผนวก ก นำเสนอรหัสเทียม (pseudo code) ซึ่งอธิบายการทำงานของส่วนนี้

สำหรับอีกส่วนหนึ่งของหน่วยควบคุมมีลักษณะเป็นเครื่องสถานจำกัด รูปที่ 4.9 แสดงแผนภาพบล็อกของส่วนนี้ โดยส่วนนี้ทำการกำหนดสัญญาณควบคุมซึ่งได้แก่ *prog_en, pc_ld, ir_ld, mem_en,* และ *reg_ld* สัญญาณ *prog_en* และ *mem_en* ปรากฏอยู่ในรูปที่ 4.2 โดยที่สัญญาณ *prog_en* เป็นสัญญาณที่ใช้ควบคุมการทำงานของรอม โดยเมื่อเป็น *'0'* จะหมายถึงให้รอมทำงาน และเมื่อเป็น *'1'* จะหมายถึงไม่ทำงาน และสัญญาณ *mem_en* เป็นสัญญาณที่ใช้ควบคุมการทำงานของแรม โดยเมื่อเป็น *'0'* จะหมายถึงให้แรมทำงาน และเมื่อเป็น *'1'* จะหมาย

ถึงไม่ทำงาน โดยทำงานร่วมกับสัญญาณ *mem_nwr* เพื่อระบุว่าจะให้อ่านหรือเขียนค่า และในรูปที่ 4.9 จะพบสัญญาณที่ไม่เคยกล่าวถึงมาก่อนอีกสองตัวคือ *pstate* (present state) และ *nstate* (next state) ซึ่งใช้เก็บค่าสถานะปัจจุบันและค่าสถานะถัดไปของเครื่องสถานะจำกัด ในหัวข้อ ก.2 ของภาคผนวก ก นำเสนอรหัสเทียมซึ่งอธิบายการทำงานของส่วนนี้ ในรูปที่ 4.10 แสดงแผนภาพสถานะของเครื่องสถานะจำกัดของหน่วยควบคุมซึ่งสอดคล้องกับที่แสดงในหัวข้อ ก.2 ของภาคผนวก ก และในตารางที่ 4.4 แสดงจำนวนสัญญาณนาฬิกาที่ใช้ในการทำงานของแต่ละคำสั่ง



รูปที่ 4.9 แผนภาพบล็อกของส่วนที่เป็นเครื่องสถานะจำกัดของหน่วยควบคุม



รูปที่ 4.10 แผนภาพสถานะของส่วนที่เป็นเครื่องสถานะจำกัดของหน่วยควบคุม

ตารางที่ 4.4 สรุปจำนวนสัญญาณนาฬิกาการทำงานของแต่ละคำสั่ง

คำสั่ง	จำนวนสัญญาณนาฬิกา
ADD	4
ADDI	4
SUB	4
SUBI	4
AND	4
ANDI	4
ORR	4
ORRI	4
XOR	4
XORI	4
COM	4
ROL	4
ROR	4
NOP	3
LUI	4
CLC	3
STC	3
JNZ	3
JNC	3
JMP	3
LDB	5
LDS	5
STB	4
STS	4
LPC	4
SPC	4
R2T	4

บทที่ 5

กระบวนการทวนสอบ

ในบทนี้จะกล่าวถึงรายละเอียดของกระบวนการทวนสอบ โดยในหัวข้อ 5.1 จะกล่าวถึงลักษณะของกระบวนการทวนสอบ ในหัวข้อ 5.2 กล่าวถึงวิธีการทวนสอบแบบลำดับขั้น โดยแบ่งงานของการทวนสอบเป็นงานย่อยซึ่งทำให้กระบวนการทวนสอบง่ายขึ้น หัวข้อ 5.3 เสนอปัญหาการเพิ่มอย่างรวดเร็วของสถานะซึ่งเป็นอุปสรรคที่สำคัญของวิธีการตรวจสอบแบบจำลองเชิงสัญลักษณ์ หัวข้อ 5.4 แสดงที่มาของสมการตัวแปรสถานะ หัวข้อ 5.5 กล่าวถึงวิธีแก้ปัญหาการเพิ่มอย่างรวดเร็วของสถานะที่ใช้ในวิทยานิพนธ์นี้อันได้แก่ การลดขนาดของสัญญาณข้อมูล, ฟังก์ชันที่ไม่ต้องตีความ, การทวนสอบแบบองค์ประกอบ, การตัดผลของตัวบ่งชี้การทดที่มีต่อเอแอลยู และการกำหนดค่าฟังก์ชันของฟังก์ชันที่ไม่ต้องตีความ และในหัวข้อที่ 5.6 ทำการสรุปสิ่งที่ได้นำเสนอในบทนี้

5.1 ลักษณะของการทวนสอบ

การทวนสอบในวิทยานิพนธ์นี้มีจุดประสงค์เพื่อตรวจสอบว่ารายละเอียดการออกแบบ (implementation) ของหน่วยประมวลผลทำงานสอดคล้องกับข้อกำหนดวงจร (specification) หรือไม่ รายละเอียดการออกแบบถูกเขียนด้วยภาษาเวอริล็อกในระดับการถ่ายโอนเรจิสเตอร์ ขณะที่ข้อกำหนดวงจรอธิบายสถาปัตยกรรมชุดคำสั่งของหน่วยประมวลผล การทวนสอบในลักษณะนี้มีชื่อเรียกว่า การทวนสอบการแบ่งละเอียด (refinement verification) (Kropf, 1999)

เนื่องจากโปรแกรมคาเดนซ์เอสเอมวีรับอินพุตด้วยภาษาเวอริล็อกไม่ได้ ทำให้ต้องนำรายละเอียดการออกแบบมาเขียนใหม่ด้วยภาษาเอสเอมวี ซึ่งเป็นภาษาเฉพาะของโปรแกรมช่วยทวนสอบ โดยพยายามเขียนรหัสต้นฉบับภาษาเอสเอมวีให้ใกล้เคียงมากที่สุดกับรหัสต้นฉบับภาษาเวอริล็อก และให้มีการแบ่งมอดูล (module) และลำดับชั้น (hierarchy) เช่นเดียวกัน

นอกจากจะเขียนรายละเอียดการออกแบบด้วยภาษาเอสเอมวีแล้ว ยังเขียนข้อกำหนดวงจรด้วยภาษาเอสเอมวีเช่นกัน โดยมีลักษณะเป็นวงจรเชิงลำดับ (sequential circuit) ซึ่งอธิบายรายละเอียดระดับพฤติกรรมของหน่วยประมวลผล นอกจากนั้นการเขียนข้อกำหนดวงจร

จำเป็นต้องมีการอ้างอิงจากรายละเอียดการออกแบบด้วย เนื่องจากต้องสร้างให้ทั้งคู่มีการเปลี่ยนสถานะของเครื่องสถานะจำกัดพร้อมกัน และต้องอยู่ในสถานะเดียวกันเสมอด้วย จึงต้องกำหนดฟังก์ชันการเปลี่ยนสถานะของข้อกำหนดวงจรให้เหมือนกับหน่วยควบคุม (control unit) ของรายละเอียดการออกแบบ ดังนั้นทั้งข้อกำหนดวงจรและรายละเอียดการออกแบบจะเริ่มต้นและสิ้นสุดการทำงานแต่ละคำสั่งพร้อมกันเสมอ สาเหตุที่ต้องกำหนดเช่นนี้เพราะต้องการให้สัญญาณสำคัญในรายละเอียดการออกแบบ และข้อกำหนดวงจรมีค่าเท่ากันตลอดเวลา

ในเรื่องนี้อาจจะขัดกับสามัญสำนึกที่ว่า ควรเขียนข้อกำหนดวงจรก่อนแล้วจึงเขียนรายละเอียดการออกแบบ และข้อกำหนดวงจรควรจะต้องเป็นอิสระจากรายละเอียดการออกแบบอีกด้วย แต่เนื่องจากในที่นี้รายละเอียดการออกแบบเป็นสิ่งที่มีอยู่แล้ว และไม่สามารถปรับแก้ได้ เพราะจะทำให้เกิดความแตกต่างจากต้นฉบับภาษาเวอริล็อก ถ้าจะแก้ไขก็ต้องแก้ไขข้อกำหนดวงจรให้มีการทำงานเหมือนกับรายละเอียดการออกแบบเท่านั้น จึงจำเป็นต้องเขียนข้อกำหนดวงจรโดยอ้างอิงจากรายละเอียดการออกแบบ

หลังจากที่เขียนรายละเอียดการออกแบบและข้อกำหนดวงจรด้วยภาษาเอสเอมวีแล้ว ขั้นต่อไปจะต้องกำหนดความสัมพันธ์การแบ่งละเอียด (refinement relation) ซึ่งระบุสิ่งที่ต้องการให้โปรแกรมคาเดนค์เอสเอมวีทำการทวนสอบ โดยจะกำหนดในลักษณะว่าต้องการให้สัญญาณใดในรายละเอียดการออกแบบและข้อกำหนดวงจรต้องมีค่าเท่ากันบ้าง ซึ่งในวิทยานิพนธ์นี้กำหนดให้สัญญาณสำคัญที่เห็นได้ในระดับสถาปัตยกรรมชุดคำสั่งมีค่าเท่ากัน ซึ่งได้แก่ *PC*, เรจิสเตอร์ภายในกลุ่มเรจิสเตอร์, ตัวบ่งชี้ และค่าภายในแรม นอกจากนั้นยังกำหนดให้สัญญาณซึ่งอาจไม่เห็นในระดับสถาปัตยกรรมชุดคำสั่งแต่มีความสำคัญต้องเท่ากันด้วย ได้แก่ *IR*, *opcode* และค่าสถานะปัจจุบันของหน่วยควบคุมซึ่งเป็นเครื่องสถานะจำกัด และเนื่องจากต้องตรวจสอบว่าสัญญาณสำคัญดังกล่าวมีค่าเท่ากันตลอดจึงจำเป็นต้องกำหนดค่าเริ่มต้นให้เท่ากันก่อนด้วย

เมื่อเขียนรายละเอียดการออกแบบและข้อกำหนดวงจรเป็นภาษาเอสเอมวีและกำหนดความสัมพันธ์การแบ่งละเอียดแล้ว จึงทำการทวนสอบด้วยโปรแกรมคาเดนค์เอสเอมวีซึ่งกระบวนการทวนสอบเป็นไปอย่างอัตโนมัติ ผลการทำงานของโปรแกรมจะบอกว่า รายละเอียดการออกแบบและข้อกำหนดวงจรมีความสัมพันธ์กันตามที่ระบุในความสัมพันธ์การแบ่งละเอียดหรือไม่ ถ้าไม่โปรแกรมจะสร้างตัวอย่างในกรณีที่ทำให้ความสัมพันธ์ดังกล่าวไม่เป็นจริง ตัวอย่างที่โปรแกรมสร้างให้จะระบุถึงค่าสัญญาณต่างๆในแต่ละสถานะตั้งแต่เริ่มต้นจนถึงเวลาที่ความ

สัมพันธ์ดังกล่าวไม่เป็นจริง ผู้ทวนสอบจึงสามารถนำตัวอย่างนี้ไปใช้แก้ไขข้อกำหนดวงจรเพื่อให้ทำงานสอดคล้องกับรายละเอียดการออกแบบได้

และเมื่อทำการแก้ไขจนกระทั่งข้อกำหนดวงจรและรายละเอียดการออกแบบทำงานสอดคล้องกันตามที่กำหนดในความสัมพันธ์การแบ่งละเอียด แสดงว่าข้อกำหนดวงจรสามารถใช้เป็นตัวแทนที่แสดงพฤติกรรมของรายละเอียดการออกแบบได้ ขั้นตอนต่อไปก็จะเปรียบเทียบข้อกำหนดวงจรกับสถาปัตยกรรมชุดคำสั่งที่ได้รับจากผู้ออกแบบหน่วยประมวลผล เพื่อตรวจสอบหาความแตกต่างและนำข้อแตกต่างซึ่งก็คือข้อผิดพลาดที่เกิดในรายละเอียดการออกแบบไปบอกแก่ผู้ออกแบบหน่วยประมวลผลเพื่อทำการแก้ไขต่อไป

จากที่กล่าวมาทั้งหมดจะสามารถสรุปเป็นหัวข้อได้ดังนี้

<u>จุดประสงค์ของการทวนสอบ</u>: ทวนสอบว่ารายละเอียดการออกแบบในระดับถ่ายโอนเรจิสเตอร์ของหน่วยประมวลผลทำงานสอดคล้องกับข้อกำหนดวงจรซึ่งอธิบายสถาปัตยกรรมชุดคำสั่ง

<u>ผลการทวนสอบรับประกันได้ว่า</u>: รายละเอียดการออกแบบและข้อกำหนดวงจรในภาษาเอสเอมวีทำงานสอดคล้องกันด้วยสมมติฐานบางข้อ

<u>ทำงานสอดคล้องหมายถึง</u>: สัญญาณสำคัญอันได้แก่ *PC*, *IR*, *opcode*, เรจิสเตอร์ภายในกลุ่มเรจิสเตอร์, ตัวบ่งชี้, ค่าภายในแรม และค่าสถานะปัจจุบันของหน่วยควบคุมซึ่งเป็นเครื่องสถานะจำกัด ของทั้งรายละเอียดการออกแบบและข้อกำหนดวงจรมีค่าเท่ากันตลอดเวลา

<u>สมมติฐานของการทวนสอบ</u>

- ต้องกำหนดให้ฟังก์ชันการเปลี่ยนสถานะของข้อกำหนดวงจรเหมือนกับหน่วยควบคุมของรายละเอียดการออกแบบ เพื่อให้สถานะของเครื่องสถานะจำกัดของทั้งคู่มีค่าเท่ากันตลอด
- ต้องกำหนดค่าเริ่มต้นของสัญญาณที่ทำการทวนสอบให้เท่ากัน แต่ไม่จำเป็นต้องระบุว่าเป็นค่าใดซึ่งภาษาเอสเอมวีสามารถทำได้

สาเหตุที่ต้องแก้ไขข้อกำหนดวงจรตามรายละเอียดการออกแบบ

- ไม่สามารถแก้ไขรายละเอียดการออกแบบในภาษาเอสเอมวีเนื่องจากจะทำให้เกิดความแตกต่างจากรหัสต้นฉบับภาษาเวอริล็อก
- จำเป็นต้องให้รายละเอียดการออกแบบและข้อกำหนดวงจรมีสถานะปัจจุบันของเครื่องสถานะจำกัดเท่ากันตลอด จึงต้องปรับแก้ข้อกำหนดวงจรให้เหมือนกับรายละเอียดการออกแบบ

ขั้นตอนของกระบวนการทวนสอบ

ในรูปที่ 5.1 แสดงแผนภาพสรุปขั้นตอนของกระบวนการทวนสอบ ซึ่งประกอบด้วยขั้นตอนดังนี้

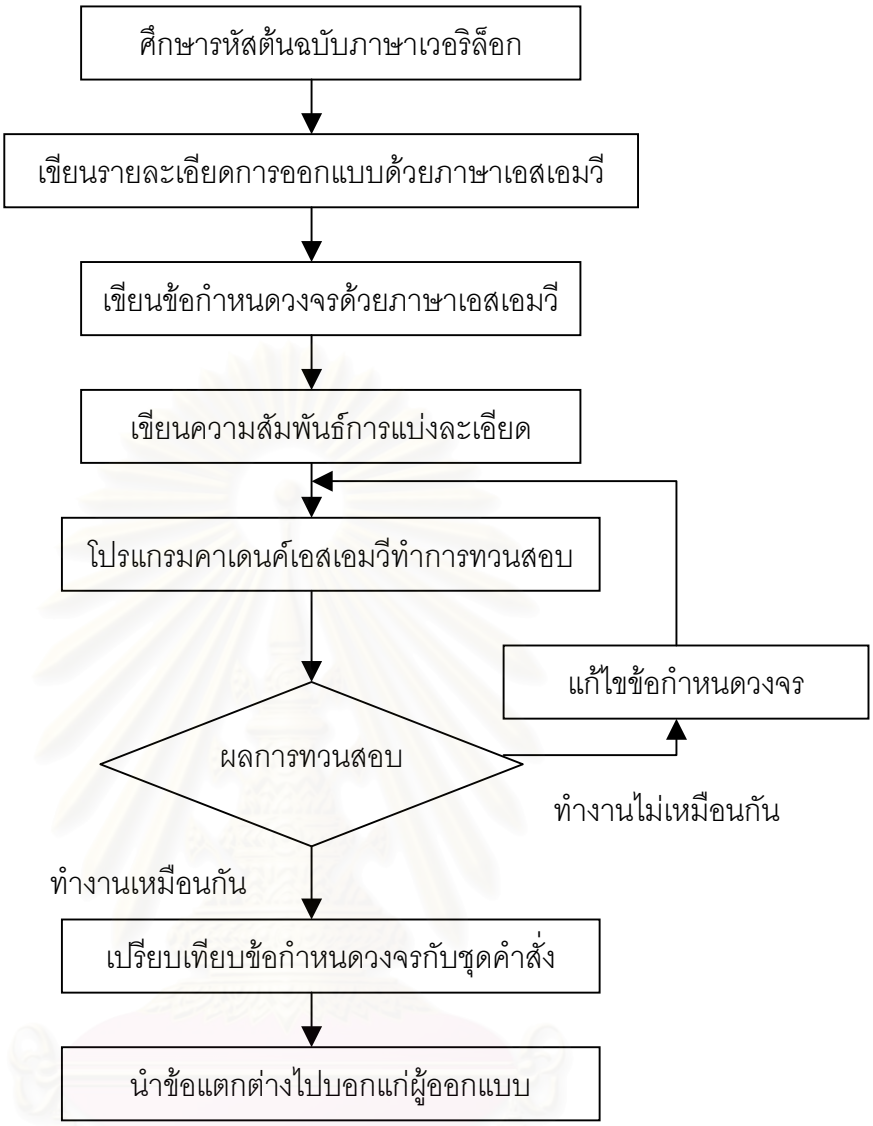
1. ศึกษารหัสต้นฉบับภาษาเวอริล็อกของหน่วยประมวลผล
2. เขียนรายละเอียดการออกแบบด้วยภาษาเอสเอมวี โดยพยายามให้ใกล้เคียงมากที่สุดกับรหัสต้นฉบับภาษาเวอริล็อก
3. เขียนข้อกำหนดวงจรด้วยภาษาเอสเอมวี ในระดับพฤติกรรมโดยอ้างอิงจากรายละเอียดการออกแบบ
4. เขียนความสัมพันธ์การแบ่งละเอียด ซึ่งระบุสัญญาณสำคัญที่ต้องการทวนสอบ
5. โปรแกรมคาเดนค์เอสเอมวีทำการทวนสอบ ซึ่งกระบวนการทวนสอบเป็นไปอย่างอัตโนมัติ
6. ถ้าผลการทวนสอบระบุว่ารายละเอียดการออกแบบและข้อกำหนดวงจรทำงานไม่เหมือนกัน จะต้องทำการแก้ไขข้อกำหนดจนกว่าทั้งคู่จะทำงานได้เหมือนกัน
7. เมื่อรายละเอียดการออกแบบและข้อกำหนดวงจรทำงานเหมือนกันแล้ว จึงนำข้อกำหนดวงจรไปเปรียบเทียบกับสถาปัตยกรรมชุดคำสั่งที่ได้รับจากผู้ออกแบบหน่วยประมวลผล
8. นำข้อแตกต่างที่ได้ซึ่งก็คือข้อผิดพลาดภายในรายละเอียดการออกแบบไปบอกผู้ออกแบบหน่วยประมวลผลเพื่อทำการแก้ไขต่อไป

5.2 วิธีทวนสอบแบบลำดับขั้น

ในหัวข้อที่แล้วได้นำเสนอลักษณะของการทวนสอบที่เริ่มจากเขียนรายละเอียดการออกแบบด้วยภาษาเอสเอมวีโดยอ้างอิงจากรหัสภาษาต้นฉบับภาษาเวอริล็อก เขียนข้อกำหนดวงจรด้วยภาษาเอสเอมวีโดยอ้างอิงจากรายละเอียดการออกแบบ เขียนความสัมพันธ์การแบ่งละเอียด และใช้โปรแกรมคาเดนค์เอสเอมวีทำการทวนสอบ แต่เนื่องจากการนำรายละเอียดการออกแบบทั้งหมดมาเขียนด้วยภาษาเอสเอมวีในคราวเดียวแล้วทำการทวนสอบ เมื่อเกิดปัญหารายละเอียดการออกแบบและข้อกำหนดวงจรทำงานไม่เหมือนกัน การหาสาเหตุของความแตกต่างจะทำได้ลำบากอย่างมาก เพราะรายละเอียดทุกส่วนอาจเป็นสาเหตุของปัญหาได้ ดังนั้นจึงควรเริ่มการทวนสอบด้วยรายละเอียดการออกแบบที่น้อยที่สุดที่สามารถทวนสอบการทำงานบางอย่างได้ จากนั้นเมื่อทวนสอบขั้นตอนแรกสำเร็จจึงเพิ่มรายละเอียดให้มากขึ้นแล้วทำการทวนสอบอีก จนกระทั่งในขั้นตอนสุดท้ายรายละเอียดการออกแบบในภาษาเอสเอมวีก็จะมีรายละเอียดครบถ้วนตามรหัสต้นฉบับภาษาเวอริล็อก

ลักษณะการทวนสอบแบบที่กล่าวมาในย่อหน้าที่แล้ว ในวิทยานิพนธ์นี้ตั้งชื่อให้ว่าวิธีทวนสอบแบบลำดับขั้น (stepwise verification method) ซึ่งจุดสำคัญคือจะไม่ทวนสอบรายละเอียดทั้งหมดของหน่วยประมวลผลในขั้นตอนเดียว แต่จะแบ่งเป็นหลายขั้นตอนและทำการเพิ่มรายละเอียดในแต่ละขั้นตอน โดยที่ขั้นตอนแรกจะมีรายละเอียดน้อยที่สุดและในขั้นตอนสุดท้ายจะมีรายละเอียดครบถ้วน นอกจากนั้นในแต่ละขั้นตอนจะมีรายละเอียดเท่าที่จำเป็นสำหรับเป้าหมายของขั้นตอนนั้น

ข้อดีของวิธีทวนสอบแบบลำดับขั้นคือ การหาสาเหตุของความแตกต่างระหว่างรายละเอียดการออกแบบและข้อกำหนดวงจรทำได้ง่ายขึ้น โดยสนใจเพียงรายละเอียดในส่วนที่เพิ่มเติมขึ้นในขั้นตอนนั้น ในทางกลับกันการทวนสอบรายละเอียดทั้งหมดในคราวเดียวแล้วพบปัญหา การหาสาเหตุของปัญหาจะทำได้ลำบากอย่างมากเนื่องจากมีรายละเอียดมาก และรายละเอียดทุกส่วนมีโอกาสเป็นสาเหตุของปัญหาได้ทั้งสิ้น



รูปที่ 5.1 ขั้นตอนของกระบวนการทวนสอบ

ในรูปที่ 5.2 สรุปลักษณะการทำงานของวิธีทวนสอบแบบลำดับขั้นซึ่งได้กล่าวไปแล้ว ส่วนที่อยู่ในกรอบเส้นประของรูปที่ 5.2 คือส่วนที่เพิ่มเติมจากรูปที่ 5.1 โดยส่วนนี้เป็นส่วนที่ใช้เปลี่ยนจากขั้นตอนหนึ่งไปสู่อีกขั้นตอนหนึ่งของวิธีการทวนสอบแบบลำดับขั้น ขั้นตอนทั้ง 6 ขั้นตอนของวิธีทวนสอบแบบลำดับขั้นประกอบด้วย

1. ทวนสอบว่าหน่วยประมวลผลสามารถอ่านคำสั่งจากหน่วยความจำโปรแกรมได้ถูกต้อง: โดยทวนสอบว่าค่าของ *IR* (instruction register) ในข้อกำหนดวงจรและรายละเอียดการออกแบบมีค่าเท่ากันตลอดเวลา

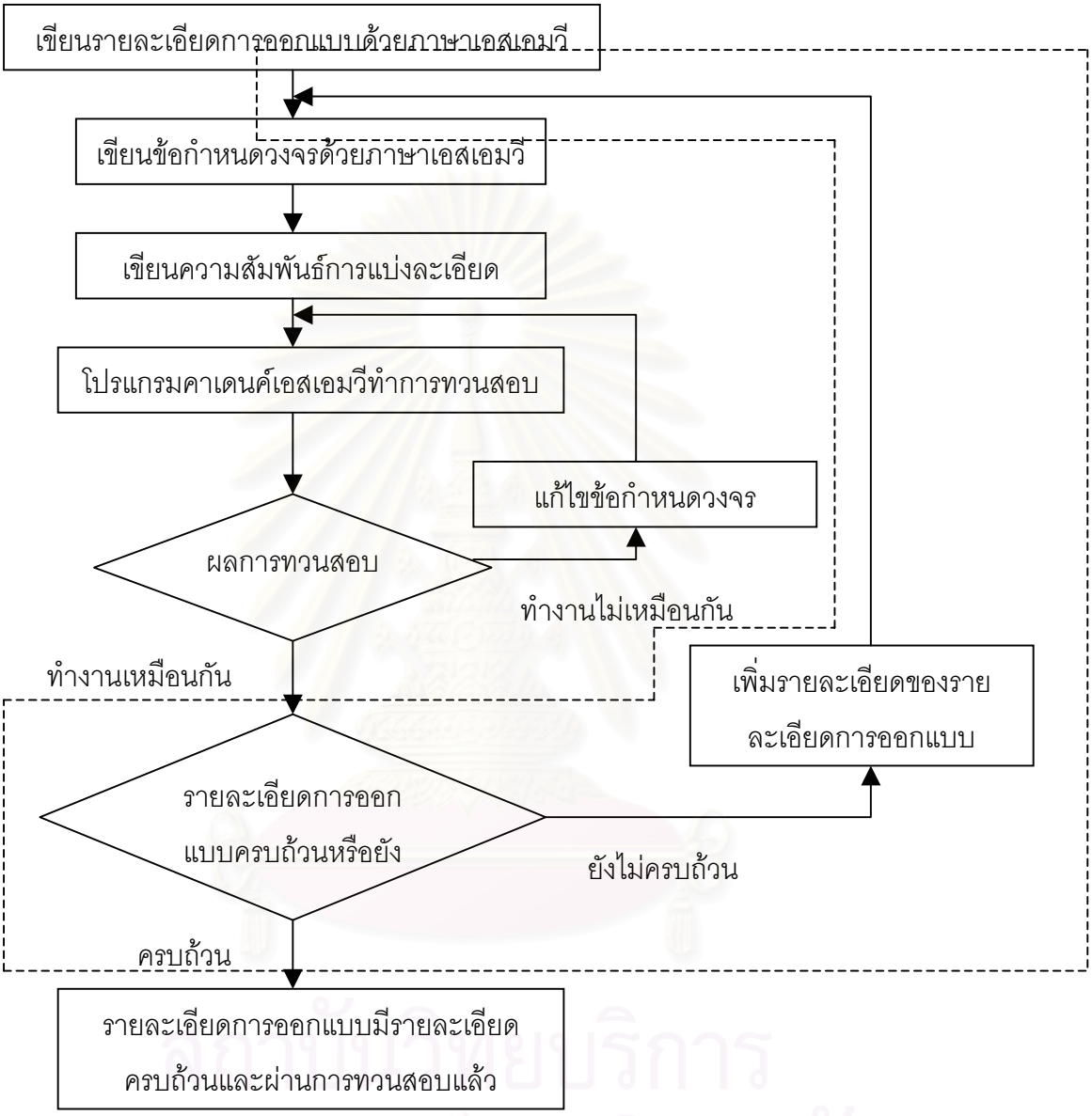
2. ทวนสอบว่าค่าของ *opcode* ในข้อกำหนดวงจรและรายละเอียดการออกแบบมีค่าเท่ากัน: ที่ต้องทวนสอบขั้นตอนนี้เนื่องจากค่าของ *opcode* ได้จากการใช้ฟังก์ชันที่ไม่ต้องตีความ (uninterpreted function) โดยฟังก์ชันนี้จะแสดงความสัมพันธ์ระหว่างค่าของ *IR* กับค่าของ *opcode* ในขั้นตอนนี้จึงต้องการทวนสอบการทำงานของฟังก์ชันที่ไม่ต้องตีความเป็นหลัก ซึ่งเป็นสิ่งที่ถูกเพิ่มเข้ามาในขั้นตอนนี้ สำหรับรายละเอียดของฟังก์ชันที่ไม่ต้องตีความจะกล่าวถึงในหัวข้อถัดไป

3. ทวนสอบว่าคำสั่งในกลุ่มคำสั่งคำนวณของหน่วยประมวลผลทำงานได้ถูกต้อง: เนื่องจากคำสั่งในกลุ่มคำสั่งคำนวณมีผลกระทบต่อค่าของเรจิสเตอร์ในกลุ่มเรจิสเตอร์และค่าของตัวบ่งชี้ ดังนั้นขั้นตอนนี้จึงทวนสอบว่าค่าของเรจิสเตอร์ภายในกลุ่มเรจิสเตอร์และค่าของตัวบ่งชี้ในข้อกำหนดวงจรและรายละเอียดการออกแบบมีค่าเท่ากันตลอดเวลา โดยสนใจเฉพาะกลุ่มคำสั่งคำนวณ

4. ทวนสอบว่าคำสั่งในกลุ่มคำสั่งกระโดดของหน่วยประมวลผลทำงานได้ถูกต้อง: เนื่องจากคำสั่งในกลุ่มคำสั่งกระโดดมีผลกระทบต่อค่าของ *PC* ดังนั้นขั้นตอนนี้จึงทวนสอบว่าค่าของ *PC* ในข้อกำหนดวงจรและรายละเอียดการออกแบบมีค่าเท่ากันตลอดเวลา โดยสนใจเฉพาะกลุ่มคำสั่งกระโดด

5. ทวนสอบว่าคำสั่งในกลุ่มคำสั่งโหลด/สโตรของหน่วยประมวลผลทำงานได้ถูกต้อง: เนื่องจากคำสั่งในกลุ่มคำสั่งโหลด/สโตรมีผลกระทบต่อค่าภายในแรมและค่าของเรจิสเตอร์ในกลุ่มเรจิสเตอร์ ดังนั้นขั้นตอนนี้จึงทวนสอบว่าค่าภายในแรมและค่าของเรจิสเตอร์ในกลุ่มเรจิสเตอร์ของข้อกำหนดวงจรและรายละเอียดการออกแบบมีค่าเท่ากันตลอดเวลา โดยสนใจเฉพาะกลุ่มคำสั่งโหลด/สโตร

6. ทวนสอบรายละเอียดทั้งหมดของหน่วยประมวลผล: ขั้นตอนนี้เป็นขั้นตอนสุดท้ายซึ่งรายละเอียดการออกแบบมีรายละเอียดครบถ้วน สัญญาณที่ทำการทวนสอบในขั้นตอนนี้ได้แก่ *IR*, *opcode*, เรจิสเตอร์ภายในกลุ่มเรจิสเตอร์, ตัวบ่งชี้, *PC* และค่าภายในแรม นอกจากนั้นยังทวนสอบค่าสถานะปัจจุบันของเครื่องสถานะจำกัดด้วย การทวนสอบเป็นการตรวจสอบว่าสัญญาณสำคัญในรายละเอียดการออกแบบและข้อกำหนดวงจรมีค่าเท่ากันตลอดเวลา โดยขั้นตอนสุดท้ายนี้ทำการทวนสอบทุกกลุ่มคำสั่งของหน่วยประมวลผล



รูปที่ 5.2 ลักษณะการทำงานของวิธีทวนสอบแบบลำดับขั้น

ตารางที่ 5.1 ตัวแปรสถานะของฟังก์ชันที่ไม่ต้องตีความของวงจรเชิงผสม

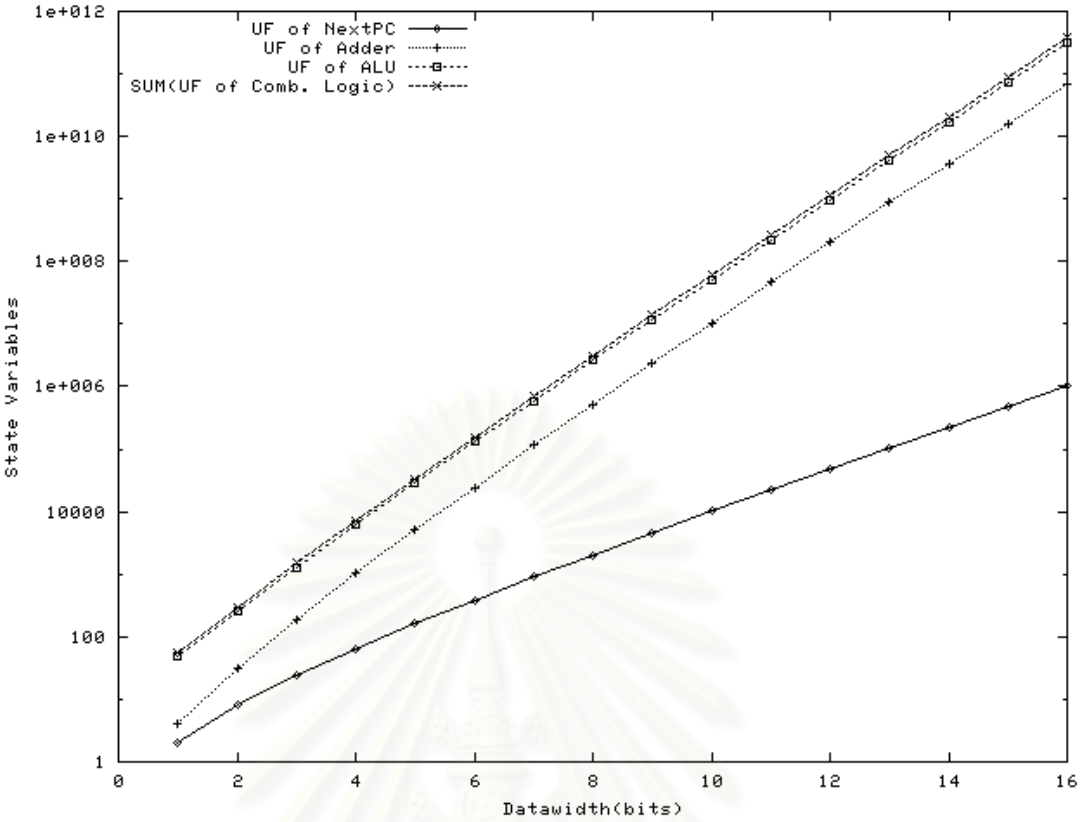
Bits	State Variables			
	UF of NextPC	UF of Adder	UF of ALU	SUM (UF of Comb. Logic)
1	2	4	48	54
2	8	32	256	296
3	24	192	1,280	1,496
4	64	1,024	6,144	7,232
5	160	5,120	28,672	33,952
6	384	24,576	131,072	156,032
7	896	114,688	589,824	705,408
8	2,048	524,288	2,621,440	3,147,776
9	4,608	2,359,296	11,534,336	13,898,240
10	10,240	10,485,760	50,331,648	60,827,648
11	22,528	46,137,344	218,103,808	264,263,680
12	49,152	201,326,592	939,524,096	1,140,899,840
13	106,496	872,415,232	4,026,531,840	4,899,053,568
14	229,376	3,758,096,384	17,179,869,184	20,938,194,944
15	491,520	16,106,127,360	73,014,444,032	89,121,062,912
16	1,048,576	68,719,476,736	309,237,645,312	377,958,170,624

ตารางที่ 5.2 ตัวแปรสถานะของวงจรส่วนอื่น

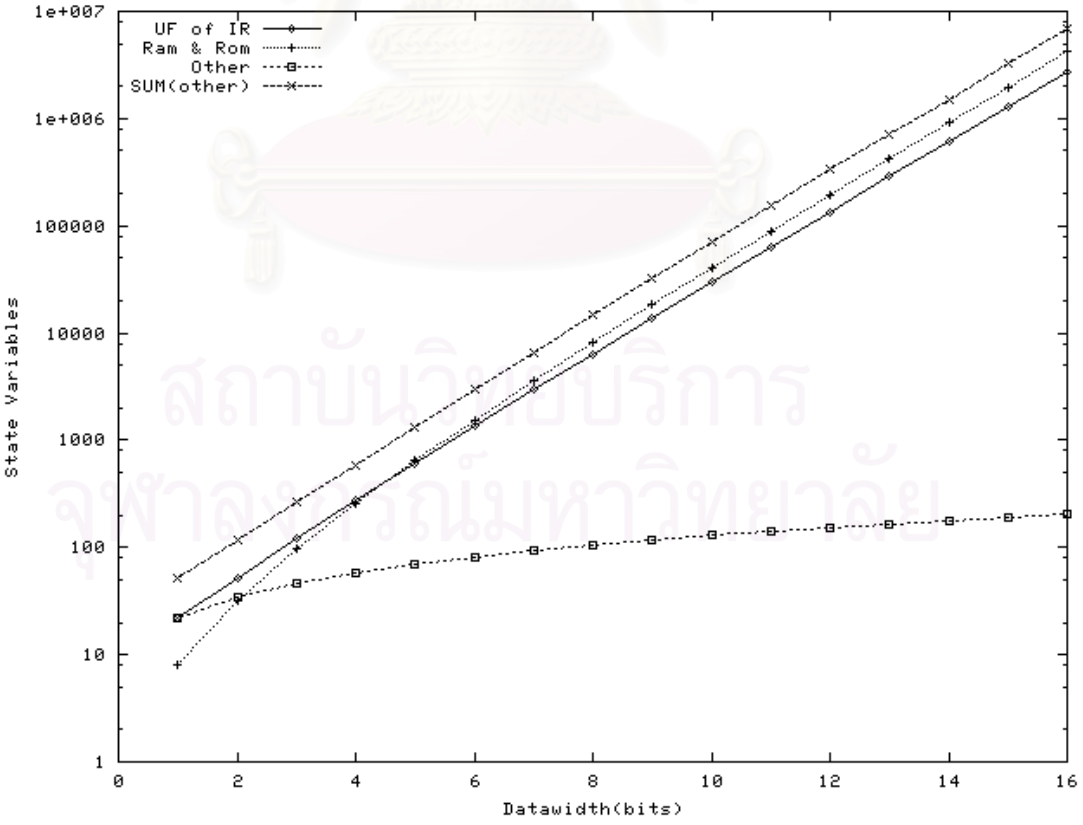
Bits	State Variables			
	UF of IR	Ram & Rom	Other	SUM (other)
1	22	8	22	52
2	52	32	34	118
3	120	96	46	262
4	272	256	58	586
5	608	640	70	1,318
6	1,344	1,536	82	2,962
7	2,944	3,584	94	6,622
8	6,400	8,192	106	14,698
9	13,824	18,432	118	32,374
10	29,696	40,960	130	70,786
11	63,488	90,112	142	153,742
12	135,168	196,608	154	331,930
13	286,720	425,984	166	712,870
14	606,208	917,504	178	1,523,890
15	1,277,952	1,966,080	190	3,244,222
16	2,686,976	4,194,304	202	6,881,482

ตารางที่ 5.3 ตัวแปรสถานะทั้งหมดของหน่วยประมวลผล

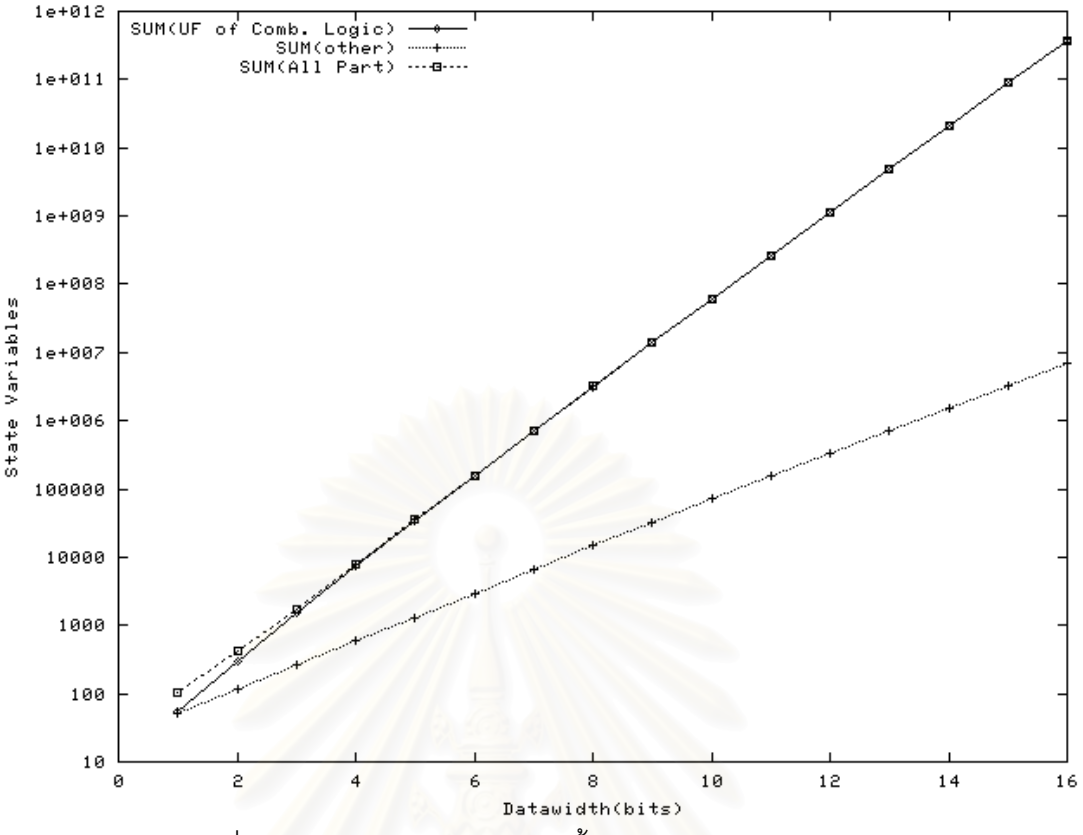
Bits	State Variables		
	SUM (UF of Comb. Logic)	SUM (other)	SUM(All Part)
1	54	52	106
2	296	118	414
3	1,496	262	1,758
4	7,232	586	7,818
5	33,952	1,318	35,270
6	156,032	2,962	158,994
7	705,408	6,622	712,030
8	3,147,776	14,698	3,162,474
9	13,898,240	32,374	13,930,614
10	60,827,648	70,786	60,898,434
11	264,263,680	153,742	264,417,422
12	1,140,899,840	331,930	1,141,231,770
13	4,899,053,568	712,870	4,899,766,438
14	20,938,194,944	1,523,890	20,939,718,834
15	89,121,062,912	3,244,222	89,124,307,134
16	377,958,170,624	6,881,482	377,965,052,106



รูปที่ 5.3 กราฟของตัวแปรสถานะของฟังก์ชันที่ไม่ต้องตีความของวงจรเชิงผสม



รูปที่ 5.4 กราฟของตัวแปรสถานะของวงจรส่วนอื่น



รูปที่ 5.5 กราฟของตัวแปรสถานะทั้งหมดของหน่วยประมวลผล

5.3 ปัญหาการเพิ่มอย่างรวดเร็วของสถานะ

จากที่กล่าวมาแล้วว่า กระบวนการทวนสอบใช้โปรแกรมคาเดนค์เอสเอมวีทวนสอบหน่วยประมวลผลให้อย่างอัตโนมัติ โปรแกรมดังกล่าวทำงานด้วยวิธีการตรวจสอบแบบจำลองเชิงสัญลักษณ์ (symbolic model checking) ซึ่งมีปัญหาการเพิ่มอย่างรวดเร็วของสถานะ (state explosion problem) ปัญหาที่ว่านี้คือ เวลาและหน่วยความจำของคอมพิวเตอร์ที่ใช้ทวนสอบจะมากขึ้นตามรายละเอียดของวงจรที่ถูกทวนสอบ และวงจรที่ถูกทวนสอบในที่นี้คือหน่วยประมวลผลซึ่งถูกทวนสอบในระดับถ่ายโอนเรจิสเตอร์ ทำให้หน่วยประมวลผลมีรายละเอียดและสถานะที่เป็นไปได้มากเกินกว่าที่จะทวนสอบด้วยโปรแกรมดังกล่าว

ตารางที่ 5.1 ถึง 5.3 แสดงจำนวนของตัวแปรสถานะ (state variable) ภายในหน่วยประมวลผล โดยตารางที่ 5.1 แสดงเฉพาะตัวแปรสถานะของฟังก์ชันที่ไม่ต้องตีความของวงจรเชิงผสมซึ่งประกอบด้วยวงจรสามส่วนคือ วงจรคำนวณค่าถัดไปสำหรับ *PC*, วงจรบวกสำหรับ *PC* และเอแอลยู (ALU) ในตารางที่ 5.2 แสดงตัวแปรสถานะของวงจรส่วนอื่นซึ่งประกอบด้วยฟังก์ชันที่ไม่ต้องตีความที่กำหนดความสัมพันธ์ระหว่าง *IR* กับสัญญาณอื่น, แรมและรอม และ

วงจรอื่นนอกจากที่กล่าวมา และในตารางที่ 5.3 แสดงตัวแปรสถานะทั้งหมดของหน่วยประมวลผลสำหรับรายละเอียดของฟังก์ชันที่ไม่ต้องตีความจะได้กล่าวถึงต่อไปในบทนี้

ในตารางที่ 5.1 ถึง 5.3 แสดงจำนวนตัวแปรสถานะเมื่อเปลี่ยนจำนวนบิตภายในส่วนทางเดินข้อมูลจากตั้งแต่ 1 บิตถึง 16 บิต โดยข้อมูลในตารางคำนวณจากสมการของจำนวนตัวแปรสถานะซึ่งแสดงในตารางที่ 5.4 (ที่มาของสมการเหล่านี้แสดงในหัวข้อที่ 5.4) โดยที่ n คือขนาดของสัญญาณข้อมูลในส่วนทางเดินข้อมูลซึ่งมีหน่วยเป็นบิต ที่มาของสมการเหล่านี้จะได้แสดงไว้ในหัวข้อถัดไป นอกจากนั้นในรูปที่ 5.3 ถึง 5.5 แสดงกราฟซึ่งสร้างโดยใช้ข้อมูลจากตารางที่ 5.1 ถึง 5.3 โดยในแกนตั้งใช้มาตราส่วนแบบลอการิทึม (logarithm scale) จากสมการจำนวนตัวแปรสถานะและกราฟในรูปที่ 5.3 ถึง 5.5 จะเห็นได้ว่าจำนวนตัวแปรสถานะเพิ่มเป็นฟังก์ชันเอกซ์โพเนนเชียลของขนาดของสัญญาณข้อมูลในส่วนทางเดินข้อมูล โปรแกรมคาเดนซ์เอสเอมวีจึงไม่สามารถทวนสอบหน่วยประมวลผลซึ่งมีขนาดของสัญญาณข้อมูลในส่วนทางเดินข้อมูลถึง 16 บิตได้ เป็นผลให้มีความจำเป็นที่จะต้องใช้เทคนิควิธีการต่างๆดังแสดงในหัวข้อถัดไปมาทำการลดจำนวนตัวแปรสถานะ เพื่อให้โปรแกรมช่วยทวนสอบสามารถทวนสอบหน่วยประมวลผลได้เสร็จได้ในเวลาที่สมเหตุผล

ตารางที่ 5.4 สมการของจำนวนตัวแปรสถานะ

วงจร	สมการของจำนวนตัวแปรสถานะ
ฟังก์ชันที่ไม่ต้องตีความของวงจรคำนวณค่าถัดไปสำหรับ *PC*	$n \bullet 2^n$
ฟังก์ชันที่ไม่ต้องตีความของวงจรบวกสำหรับ *PC*	$n \bullet 4^n$
ฟังก์ชันที่ไม่ต้องตีความของเอแอลยู	$4(n+2) \bullet 4^n$
ฟังก์ชันที่ไม่ต้องตีความที่กำหนดความสัมพันธ์ระหว่าง *IR* กับสัญญาณอื่น	$(2n+9) \bullet 2^n$
แรมและรอม	$4n \bullet 2^n$
วงจรอื่น	$10+12n$

5.4 ที่มาของสมการจำนวนตัวแปรสถานะ

ในหัวข้อที่แล้วได้นำเสนอตารางที่ 5.4 ซึ่งแสดงสมการของจำนวนตัวแปรสถานะ ในหัวข้อนี้จึงต้องการนำเสนอที่มาของสมการดังกล่าว ข้อมูลที่ใช้เพื่อสรุปสมการจำนวนตัวแปรสถานะได้จากรายงานจากโปรแกรมดาเคนต์เอสเอมวีซึ่งจะบอกข้อมูลของตัวแปรสถานะทั้งหมดในหน่วยประมวลผล ในที่นี้แบ่งกลุ่มของตัวแปรสถานะเป็นฟังก์ชันที่ไม่ต้องตีความของวงจรเชิงผสม, ฟังก์ชันที่ไม่ต้องตีความของสัญญาณที่เกี่ยวกับ *IR* , แรมและรอม, และวงจรส่วนอื่นๆ

5.4.1 ฟังก์ชันที่ไม่ต้องตีความของวงจรเชิงผสม

ในส่วนนี้ประกอบด้วยฟังก์ชันที่ไม่ต้องตีความของมอดูล *nextpc*, มอดูล *adder*, และมอดูล *alu* เริ่มต้นด้วยฟังก์ชันที่ไม่ต้องตีความของมอดูล *nextpc* มอดูลนี้ไม่ปรากฏในรูปที่ 4.3 ส่วนทางเดินข้อมูลของหน่วยประมวลผล โดยมอดูลนี้ถูกสร้างในรายละเอียดการออกแบบของรหัสต้นฉบับภาษาเอสเอมวี เพื่อใช้แทนวงจรบวกเพิ่มค่าหนึ่งให้กับสัญญาณ *pc_out* (ดูรูปที่ 4.3) สาเหตุที่ต้องสร้างมอดูลนี้ขึ้นเนื่องจากเป็นการบวกค่าคงที่ ซึ่งไม่สามารถสร้างโดยใช้ฟังก์ชันที่ไม่ต้องตีความ ดังนั้นจึงทำการสร้างมอดูลนี้ให้เป็นมอดูลที่มีหนึ่งพอร์ตอินพุตขึ้นแทน และสร้างฟังก์ชันที่ไม่ต้องตีความของมอดูลนี้ให้มีหนึ่งพอร์ตอินพุตเช่นกัน ฟังก์ชันที่ไม่ต้องตีความของมอดูล *nextpc* มีลักษณะเป็นแถวลำดับหนึ่งมิติที่ดัชนี (หรือก็คืออินพุตของฟังก์ชัน) มีขนาดเท่ากับสัญญาณข้อมูล เมื่อสัญญาณข้อมูลมีขนาด *n* บิตแถวลำดับนี้จึงมีจำนวนช่องเท่ากับ 2^n ช่อง และเนื่องจากแต่ละช่องของแถวลำดับมีขนาดเท่ากับสัญญาณข้อมูล ทำให้ฟังก์ชันที่ไม่ต้องตีความของมอดูล *nextpc* มีจำนวนของตัวแปรสถานะเป็น $n \cdot 2^n$

สำหรับฟังก์ชันที่ไม่ต้องตีความของมอดูล *adder* ก็สามารถพิจารณาได้ในลักษณะที่คล้ายกัน ฟังก์ชันที่ไม่ต้องตีความของมอดูลนี้มีลักษณะเป็นแถวลำดับสองมิติที่มีดัชนีขนาดเท่ากับสัญญาณข้อมูล เมื่อสัญญาณข้อมูลมีขนาด *n* บิตแถวลำดับนี้จะมีจำนวนช่องเท่ากับ $2^n \cdot 2^n$ หรือเป็น 4^n นั้นเอง และเนื่องจากแต่ละช่องของแถวลำดับมีขนาดเท่ากับสัญญาณข้อมูลทำให้ฟังก์ชันที่ไม่ต้องตีความของมอดูล *adder* มีจำนวนตัวแปรสถานะเป็น $n \cdot 4^n$

และสุดท้ายฟังก์ชันที่ไม่ต้องตีความในมอดูล *alu* ในมอดูลนี้มีฟังก์ชันที่ไม่ต้องตีความอยู่สามฟังก์ชันคือ ฟังก์ชันของเอาต์พุตที่เป็นสัญญาณข้อมูล, ฟังก์ชันของเอาต์พุตที่เป็นตัวบ่งชี้การทด, และฟังก์ชันของเอาต์พุตที่เป็นตัวบ่งชี้ค่าศูนย์ เริ่มต้นจากฟังก์ชันของเอาต์พุตที่เป็นสัญญาณข้อมูล ซึ่งมีลักษณะเป็นแถวลำดับสี่มิติโดยมีดัชนีสี่ตัว (หรือมีอินพุตสี่ตัว) สองตัวมีขนาด

คงที่ตัวละหนึ่งบิต อีกสองตัวมีขนาดเท่ากับสัญญาณข้อมูล เมื่อสัญญาณข้อมูลมีขนาด *n* บิต แถวลำดับนี้จึงมีจำนวนช่องเท่ากับ $2 \cdot 2^n \cdot 2^n \cdot 2$ หรือเป็น $4 \cdot 4^n$ นั้นเอง และเนื่องจากแต่ละช่องของแถวลำดับมีขนาดเท่ากับสัญญาณข้อมูล ทำให้ฟังก์ชันที่ไม่ต้องตีความของเอาต์พุตที่เป็นสัญญาณข้อมูลมีจำนวนของตัวแปรสถานะเป็น $n \cdot 4 \cdot 4^n$ สำหรับฟังก์ชันที่ไม่ต้องตีความของเอาต์พุตที่เป็นของตัวบ่งชี้การทดและของตัวบ่งชี้ค่าศูนย์ก็สามารถพิจารณาได้ในลักษณะที่คล้ายกัน แต่เนื่องจากแต่ละช่องของแถวลำดับมีขนาดคงที่เป็นหนึ่งบิต (เอาต์พุตเป็นตัวบ่งชี้จึงมีขนาดคงที่เป็นหนึ่งบิต) สมการของตัวแปรสถานะของแต่ละฟังก์ชันจึงเป็น $1 \cdot 4 \cdot 4^n$ เมื่อรวมทั้งสามฟังก์ชันมอดูล *alu* จะมีตัวแปรสถานะเท่ากับ $n \cdot 4 \cdot 4^n + 1 \cdot 4 \cdot 4^n + 1 \cdot 4 \cdot 4^n$ หรือเป็น $4(n+2) \cdot 4^n$

5.4.2 ฟังก์ชันที่ไม่ต้องตีความของสัญญาณที่เกี่ยวข้องกับ *IR*

ฟังก์ชันที่ไม่ต้องตีความของสัญญาณที่เกี่ยวข้องกับ *IR* ประกอบด้วยฟังก์ชันหกฟังก์ชัน ห้าฟังก์ชันถูกใช้เพื่อกำหนดค่าให้กับสัญญาณ *opcode*, *rs*, *i_r*, *imm*, และ *b_offset* ซึ่งเป็นส่วนหนึ่งของ *IR* (ดูรูปที่ 4.1) และอีกหนึ่งฟังก์ชันใช้กำหนดค่าให้กับสัญญาณ *alu_op* ซึ่งเป็นสัญญาณควบคุมของเอแอลยู แต่ละฟังก์ชันเป็นแถวลำดับหนึ่งมิติที่ดัชนีมีขนาดเท่ากับสัญญาณข้อมูล เมื่อสัญญาณข้อมูลมีขนาด *n* บิตแถวลำดับนี้จึงมีจำนวนช่องเท่ากับ 2^n สำหรับฟังก์ชันที่ใช้กำหนดค่าให้สัญญาณ *imm* และ *b_offset* เอาต์พุตของฟังก์ชันมีขนาดเท่ากับสัญญาณข้อมูล (หรือก็คือแต่ละช่องของแถวลำดับมีขนาดเท่ากับสัญญาณข้อมูล) ทำให้ฟังก์ชันทั้งสองมีจำนวนตัวแปรสถานะฟังก์ชันละ $n \cdot 2^n$ ตัว สำหรับฟังก์ชันที่ใช้กำหนดค่าให้สัญญาณ *aluop* เอาต์พุตของฟังก์ชันมีขนาดคงที่เป็นหนึ่งบิต ฟังก์ชันนี้จึงมีจำนวนตัวแปรสถานะเป็น $1 \cdot 2^n$ สำหรับฟังก์ชันที่ใช้กำหนดค่าให้สัญญาณ *opcode* เอาต์พุตของฟังก์ชันมีขนาดคงที่เป็นห้าบิต ฟังก์ชันนี้จึงมีจำนวนตัวแปรสถานะเป็น $5 \cdot 2^n$ สำหรับฟังก์ชันที่ใช้กำหนดค่าให้สัญญาณ *rs* เอาต์พุตของฟังก์ชันมีขนาดคงที่เป็นสองบิต ฟังก์ชันนี้จึงมีจำนวนตัวแปรสถานะเป็น $2 \cdot 2^n$ และสำหรับฟังก์ชันที่ใช้กำหนดค่าให้สัญญาณ *i_r* เอาต์พุตของฟังก์ชันมีขนาดคงที่เป็นหนึ่งบิต ฟังก์ชันนี้จึงมีจำนวนตัวแปรสถานะเป็น $1 \cdot 2^n$ เมื่อรวมทั้งหมดจำนวนตัวแปรสถานะของฟังก์ชันที่ไม่ต้องตีความของสัญญาณที่เกี่ยวข้องกับ *IR* จึงเป็น $n \cdot 2^n + n \cdot 2^n + 1 \cdot 2^n + 5 \cdot 2^n + 2 \cdot 2^n + 1 \cdot 2^n$ หรือเป็น $(2n+9) \cdot 2^n$

5.4.3 แรมและรอม

รอมและแรมมีลักษณะเป็นแถวลำดับหนึ่งมิติที่ดัชนีมีขนาดเท่ากับสัญญาณข้อมูล เมื่อสัญญาณข้อมูลมีขนาด n บิตแถวลำดับนี้จึงมีจำนวนช่องเท่ากับ 2^n และเนื่องจากเอาต์พุตของรอมและแรมมีขนาดเท่ากับสัญญาณข้อมูลทำให้แต่ละช่องของแถวลำดับมีขนาด n บิต ดังนั้นรอมและแรมแต่ละตัวจะมีจำนวนตัวแปรสถานะเป็น $n \cdot 2^n$ และเนื่องจากในข้อกำหนดวงจรและรายละเอียดการออกแบบใช้หน่วยความจำแยกกัน จึงต้องมีรอมสองชุดและมีแรมสองชุด จำนวนตัวแปรสถานะของทั้งหมดจึงเป็น $n \cdot 2^n + n \cdot 2^n + n \cdot 2^n + n \cdot 2^n$ หรือเป็น $4n \cdot 2^n$

5.4.4 วงจรส่วนอื่น

วงจรส่วนอื่นที่เหลือของหน่วยประมวลผลประกอบด้วย ตัวบ่งชี้, กลุ่มของเรจิสเตอร์, *IR*, *PC*, และตัวแปรที่เก็บสถานะของหน่วยควบคุม สำหรับตัวบ่งชี้ประกอบด้วยตัวบ่งชี้การทดและตัวบ่งชี้ค่าศูนย์ และเนื่องจากข้อกำหนดวงจรและรายละเอียดการออกแบบใช้ตัวบ่งชี้แยกจากกัน จึงรวมเป็นว่ามีตัวบ่งชี้ทั้งหมดสี่ตัว และเนื่องจากตัวบ่งชี้มีขนาดเป็นหนึ่งบิตเสมอโดยไม่ขึ้นกับขนาดของสัญญาณข้อมูล ส่วนของตัวบ่งชี้จึงมีจำนวนตัวแปรสถานะเท่ากับ 4 ตัว สำหรับกลุ่มของเรจิสเตอร์ประกอบด้วยเรจิสเตอร์สี่ตัว เมื่อรวมทั้งของข้อกำหนดวงจรและรายละเอียดการออกแบบจึงเป็นแปดตัว และเนื่องจากขนาดของเรจิสเตอร์เท่ากับขนาดของสัญญาณข้อมูล จำนวนตัวแปรสถานะของส่วนนี้จึงเป็น $8n$ เมื่อ n เป็นขนาดของสัญญาณข้อมูล สำหรับ *IR* และ *PC* เมื่อรวมทั้งของข้อกำหนดวงจรและรายละเอียดการออกแบบจะรวมมีเรจิสเตอร์สี่ตัว และเนื่องจากขนาดของเรจิสเตอร์เท่ากับขนาดของสัญญาณข้อมูล จำนวนตัวแปรสถานะของส่วนนี้จึงเป็น $4n$ เมื่อ n เป็นขนาดของสัญญาณข้อมูล และสำหรับตัวแปรที่เก็บสถานะของหน่วยควบคุมจะมีสองตัว ตัวหนึ่งจากข้อกำหนดวงจรและอีกตัวหนึ่งจากรายละเอียดการออกแบบ ตัวแปรแต่ละตัวมีขนาดสามบิต รวมทั้งสองตัวแปรจึงมีจำนวนตัวแปรสถานะเป็น 6 ตัว รวมทั้งหมดจึงเป็น $4 + 8n + 4n + 6$ หรือเป็น $10 + 12n$

5.5 วิธีแก้ปัญหา

เนื่องจากปัญหาการเพิ่มอย่างรวดเร็วของสถานะจึงจำเป็นต้องทำการกำหนดสาระสำคัญ (abstraction) เพื่อลดรายละเอียดของหน่วยประมวลผล วิธีการที่ใช้ได้แก่ การลด

ขนาดของสัญญาณข้อมูล, ฟังก์ชันที่ไม่ต้องตีความ, การทวนสอบแบบองค์ประกอบ, การตัดผลของตัวบ่งชี้การทดที่มีต่อเอแอลยู และการกำหนดค่าฟังก์ชันของฟังก์ชันที่ไม่ต้องตีความ

5.5.1 การลดขนาดของสัญญาณข้อมูล

หน่วยประมวลผลที่ถูกทวนสอบเป็นหน่วยประมวลผลขนาด 16 บิต ซึ่งมีรายละเอียดมากเกินกว่าที่โปรแกรมคาเดนซ์เอสเอมวีจะสามารถทวนสอบได้ ดังนั้นจึงมีความจำเป็นที่จะต้องลดขนาดของสัญญาณข้อมูลจาก 16 บิตเหลือเพียง 1 บิต เพื่อลดรายละเอียดของหน่วยประมวลผล อีกประการหนึ่งการลดขนาดของสัญญาณข้อมูลถูกใช้เฉพาะกับส่วนทางเดินข้อมูล (datapath) ภายในหน่วยประมวลผลแต่ไม่ได้ใช้กับหน่วยควบคุม (control unit) ดังนั้นรายละเอียดของหน่วยควบคุมในหน่วยประมวลผลที่ถูกทวนสอบจะเหมือนกับในรหัสต้นฉบับภาษาเวอริล็อกทุกประการ

การลดขนาดของสัญญาณข้อมูลมีผลทำให้ไม่สามารถทวนสอบค่าที่แท้จริงของข้อมูลภายในส่วนทางเดินข้อมูลได้ สิ่งที่สามารถทวนสอบได้ก็คือ สัญญาณข้อมูลในข้อกำหนดวงจรและในรายละเอียดการออกแบบมีค่าเท่ากันหรือไม่ และเนื่องจากสัญญาณข้อมูลถูกลดขนาดเหลือ 1 บิต ทำให้สถานะที่เป็นไปได้มี 2 สถานะ ซึ่งพอเพียงที่จะใช้ตรวจสอบว่าสัญญาณเท่ากันหรือไม่เท่ากัน

ในแง่ของความน่าเชื่อถือของกระบวนการทวนสอบ การลดขนาดของสัญญาณข้อมูลมีผลต่อความน่าเชื่อถืออย่างแน่นอน เนื่องจากไม่สามารถทวนสอบค่าที่แท้จริงของสัญญาณข้อมูลได้ แต่ถ้าพิจารณาอีกแง่หนึ่งจะเห็นว่ากระบวนการทวนสอบสามารถรับประกันความถูกต้องของหน่วยควบคุมได้ เนื่องจากไม่ได้ใช้การลดขนาดของสัญญาณข้อมูลกับหน่วยควบคุมของหน่วยประมวลผล ดังนั้นถึงแม้ว่ากระบวนการทวนสอบจะรับประกันความถูกต้องได้เพียงว่า หน่วยควบคุมและวงจรส่วนทางเดินข้อมูลที่ถูกลดขนาดสัญญาณข้อมูลสามารถทำงานร่วมกันได้อย่างถูกต้อง การรับประกันในส่วนนี้ได้จึงถือว่าสำคัญเพราะสามารถรับประกันความถูกต้องของหน่วยควบคุมซึ่งเป็นส่วนที่ซับซ้อนมากที่สุดของหน่วยประมวลผล

นอกจากนั้นถึงแม้ว่าการลดขนาดของสัญญาณข้อมูลจะทำให้ไม่สามารถทวนสอบค่าที่แท้จริงของข้อมูลได้ แต่กระบวนการทวนสอบก็สามารถทวนสอบความถูกต้องของส่วนทางเดินข้อมูลที่ไม่เกี่ยวข้องกับค่าที่แท้จริงอันได้แก่ การทำงานของอุปกรณ์รวมสัญญาณ (multiplexer) ซึ่งมีอยู่เป็นจำนวนมากในส่วนทางเดินข้อมูล หรือการทำงานของกลุ่มเรจิสเตอร์ว่า

สามารถดึงข้อมูลจากเรจิสเตอร์ที่ต้องการหรือเก็บค่าในเรจิสเตอร์ที่กำหนดได้หรือไม่ เหตุผลที่ทำให้กล่าวเช่นนี้ได้เนื่องจากถึงแม้จะมีการลดขนาดของสัญญาณข้อมูล แต่การเชื่อมสายสัญญาณภายในส่วนทางเดินข้อมูลยังเหมือนเดิมทุกประการ ดังนั้นกระบวนการทวนสอบจึงสามารถรับประกันความถูกต้องของส่วนทางเดินข้อมูลได้ในระดับหนึ่ง

จากที่กล่าวมาจะเห็นได้ว่า ถึงแม้การลดขนาดของสัญญาณข้อมูลจะทำให้ความน่าเชื่อถือของกระบวนการทวนสอบลดลงไปบ้าง แต่ผลลัพธ์ที่กระบวนการทวนสอบสามารถรับประกันได้ก็ยังมีความสำคัญ และสามารถช่วยเพิ่มความมั่นใจในความถูกต้องของหน่วยประมวลผลได้มากพอสมควร

5.5.2 ฟังก์ชันที่ไม่ต้องตีความ

จากการลดขนาดของสัญญาณข้อมูลเหลือเพียง 1 บิต ทำให้การคำนวณต่างๆที่เกิดขึ้นกับสัญญาณข้อมูลต้องมีการเปลี่ยนแปลงไป เช่น คำสั่งบวกซึ่งเดิมเป็นการบวกแบบ 16 บิต ก็ต้องปรับแก้ให้เป็นการบวกแบบ 1 บิตแทน เป็นต้น นอกจากนี้ยังมีคำสั่งบางคำสั่งซึ่งเมื่อสัญญาณข้อมูลถูกลดขนาดเหลือ 1 บิตแล้วจะไม่สามารถทำงานได้ เช่นคำสั่ง *“LUI r”* (Load upper immediate to [r]) ซึ่งจะนำค่าของ *imm* (immediate value) ไปใส่ใน 8 บิตบนของเรจิสเตอร์ที่กำหนดในคำสั่ง ในกรณีของคำสั่งนี้ไม่สามารถปรับแก้วงจรเชิงผสมเพื่อให้มีผลการทำงานคล้ายเดิมได้ จากความยุ่งยากในการต้องปรับแก้วงจรเชิงผสม (combinational circuit) ซึ่งทำหน้าที่คำนวณข้อมูล และการที่สัญญาณข้อมูลมีความสำคัญลดลงเนื่องจากการลดขนาดของสัญญาณข้อมูลเหลือเพียง 1 บิต ทำให้ตัดสินใจใช้ฟังก์ชันที่ไม่ต้องตีความ (uninterpreted function) (McMillan, 1998) เพื่อแทนวงจรเชิงผสมภายในส่วนทางเดินข้อมูลของหน่วยประมวลผล

ฟังก์ชันที่ไม่ต้องตีความมีลักษณะเป็นแถวลำดับ (array) หลายมิติ โดยที่ค่าดัชนี (index) ของแถวลำดับเปรียบได้กับอินพุตของวงจรเชิงผสม และค่าของฟังก์ชันเปรียบได้กับค่าเอาต์พุตของวงจร นอกจากนั้นฟังก์ชันที่ไม่ต้องตีความจะไม่ได้ถูกกำหนดค่าฟังก์ชันเอาไว้ ทำให้ค่าฟังก์ชันเป็นค่าใดก็ได้ตามแต่โปรแกรมคาเดนซ์เอสเอมวีจะกำหนดให้ เป็นผลให้โปรแกรมช่วยทวนสอบสามารถเปลี่ยนค่าฟังก์ชันให้ครอบคลุมทุกกรณี ซึ่งทำให้กระบวนการทวนสอบมีความน่าเชื่อถือมากขึ้น อีกประการหนึ่งในข้อกำหนดวงจรและรายละเอียดการออกแบบจะใช้ฟังก์ชันที่ไม่ต้องตีความร่วมกัน เพื่อให้การทำงานของทั้งคู่มีความสอดคล้องกัน

วงจรที่ถูกแทนที่ด้วยฟังก์ชันที่ไม่ต้องตีความได้แก่ เอแอลยู (ALU–Arithmetic Logic Unit), วงจรคำนวณค่าถัดไปสำหรับ *PC* และวงจรบวกสำหรับ *PC* ซึ่งสามส่วนนี้เป็นทั้งหมดของวงจรเชิงผสมภายในส่วนทางเดินข้อมูลของหน่วยประมวลผล นอกจากนั้นฟังก์ชันที่ไม่ต้องตีความยังถูกนำไปใช้ในการกำหนดความสัมพันธ์ระหว่าง *IR* กับสัญญาณหลายตัว เช่น *opcode*, *rs* (source and destination register), *i_r* (instruction mode), *imm* (immediate value), และ *b_offset* (base offset) เนื่องจากสัญญาณเหล่านี้คือบางบิตของ *IR* แต่เนื่องจากการลดขนาดของสัญญาณข้อมูลจาก 16 บิตเหลือ 1 บิต ทำให้ไม่สามารถแยกบางบิตจาก 1 บิตของ *IR* ได้ จึงจำเป็นต้องใช้ฟังก์ชันที่ไม่ต้องตีความมาแทนที่

5.5.3 การทวนสอบแบบองค์ประกอบ

จากที่กล่าวแล้วในตอนต้นของบทนี้ว่า การทวนสอบในที่นี้เป็นการตรวจสอบว่าสัญญาณสำคัญบางตัวในรายละเอียดการออกแบบและข้อกำหนดวงจรมีค่าเท่ากันหรือไม่ และเนื่องจากสัญญาณที่ต้องการทวนสอบมีหลายตัว การทวนสอบความถูกต้องของสัญญาณทุกตัวพร้อมกันจึงเป็นสิ่งที่เป็นไปไม่ได้ เพราะจะทำให้จำนวนสถานะที่เป็นไปได้ของวงจรมีมากเกินกว่าที่โปรแกรมช่วยทวนสอบจะสามารถทำงานได้ เนื่องด้วยเหตุนี้โปรแกรมคาเดนค์เอสเอมวีจึงทวนสอบความถูกต้องของสัญญาณครั้งละตัว โดยขณะที่กำลังทวนสอบความถูกต้องของสัญญาณหนึ่ง โปรแกรมจะกำหนดให้สัญญาณอื่นๆที่ต้องถูกทวนสอบมีความถูกต้องก่อน และใช้ความถูกต้องของสัญญาณอื่นๆเหล่านั้นเพื่อทวนสอบความถูกต้องของสัญญาณที่กำลังสนใจ การกำหนดให้สัญญาณอื่นๆถูกต้องก่อนนี้ ช่วยลดจำนวนสถานะที่เป็นไปได้ทั้งหมด ซึ่งโปรแกรมช่วยทวนสอบจะต้องทำการสำรวจขณะที่ทวนสอบความถูกต้องของสัญญาณที่กำลังสนใจ การทวนสอบในลักษณะนี้มีชื่อเรียกว่า การทวนสอบแบบองค์ประกอบ (compositional verification) (Clarke, Long และ McMillan, 1989)

ตารางที่ 5.5 แสดงจำนวนตัวแปรสถานะของสัญญาณแต่ละตัวที่ถูกทวนสอบ จะเห็นได้ว่าจำนวนตัวแปรสถานะลดลงเช่นกรณีของ *ir_out* มีตัวแปรสถานะ 94 ตัวซึ่งถ้าหากไม่ใช้การทวนสอบแบบองค์ประกอบจะมีตัวแปรสถานะ 106 ตัว (จากข้อมูลในตารางที่ 5.3) หรืออีกนัยหนึ่งเทคนิคนี้ช่วยทำให้ตัวแปรสถานะลดลงได้ 12 ตัว หรือสถานะที่เป็นไปได้ทั้งหมดลดลง 2^{12} เท่า

ตารางที่ 5.5 จำนวนตัวแปรสถานะของสัญญาณที่ถูกทวนสอบ

สัญญาณที่ถูกทวนสอบ	จำนวนตัวแปรสถานะ
c	92
ir_out	94
opcode	91
pc_out	92
regfile.ra	92
regfile.rb	92
regfile.rsp	92
regfile.rt	92
z	92
pstate	94
ram.mem[0]	92
ram.mem[1]	92

5.5.4 ตัดผลของตัวบ่งชี้การทดที่มีต่อเอแอลยู

ในฟังก์ชันที่ไม่ต้องตีความของเอแอลยู (ALU) ปกติจะต้องสร้างให้เป็นฟังก์ชันที่ขึ้นกับอินพุตทั้ง 2 ชุดของเอแอลยู และตัวบ่งชี้การทด แต่เนื่องจากถ้าเป็นเช่นนั้นความซับซ้อนของวงจรจะมากเกินไปจนโปรแกรมทวนสอบไม่สามารถทำงานได้ ทำให้จำเป็นต้องเปลี่ยนฟังก์ชันที่ไม่ต้องตีความให้ขึ้นกับเฉพาะอินพุตทั้ง 2 ชุดของเอแอลยูแต่ไม่ขึ้นกับตัวบ่งชี้การทด ซึ่งจากการลดในส่วนนี้ทำให้ตัวแปรสถานะ (state variable) ลดลง 24 ตัว (ในเอแอลยูมีฟังก์ชันที่ไม่ต้องตีความสามชุดสำหรับ ผลลัพธ์ของเอแอลยู, ตัวบ่งชี้การทด และตัวบ่งชี้ค่าศูนย์ ซึ่งเดิมแต่ละชุดมีตัวแปรสถานะ 16 ตัวต่อชุดเนื่องจากเป็นแถวลำดับขนาดสี่มิติ เมื่อตัดผลของตัวบ่งชี้การทดทำให้ลดขนาดเป็นแถวลำดับขนาดสามมิติจึงมีตัวแปรสถานะเป็น 8 ตัว หรือสำหรับฟังก์ชันที่ไม่ต้องตีความแต่ละชุดลดได้ 8 ตัว เมื่อรวมของสามชุดจึงลดได้ทั้งหมด 24 ตัว) หรือกล่าวอีกนัยหนึ่งคือทำให้สถานะที่เป็นไปได้ทั้งหมดลดลงถึง 2^{24} เท่า

ดังนั้นจะเห็นได้ว่าการทำเช่นนี้สามารถช่วยลดจำนวนสถานะลงได้เป็นจำนวนมาก ถึงแม้ว่าจะสามารถลดจำนวนสถานะ แต่ก็อาจเกิดคำถามเกี่ยวกับความถูกต้องของการทวน

สอบ โดยการตัดผลของตัวบ่งชี้การทดที่มีต่อฟังก์ชันที่ไม่ต้องตีความของเอแอลยูทำให้เอาต์พุตของเอแอลยูไม่ขึ้นกับตัวบ่งชี้การทด ซึ่งเป็นผลให้โปรแกรมช่วยทวนสอบไม่มีโอกาสตรวจสอบว่าสายสัญญาณที่ต่อค่าของตัวบ่งชี้การทดกับอินพุตของเอแอลยูทำงานได้ถูกต้องหรือไม่ ในจุดนี้ถือว่าเป็นปัญหาจากการตัดผลของตัวบ่งชี้การทด แต่เมื่อเทียบกับประโยชน์ที่สามารถลดจำนวนสถานะที่เป็นไปได้ทั้งหมดลงถึง 2^{24} เท่าจึงถือได้ว่าคุ้มค่ามาก อีกประการหนึ่งผู้ทวนสอบสามารถทวนสอบสายสัญญาณเส้นนี้ในภายหลังด้วยวิธีการจำลองการทำงานได้ ซึ่งการทวนสอบเฉพาะจุดนี้ไม่เป็นเรื่องยากเพราะสถานะที่เป็นไปได้ของสายสัญญาณนี้มีไม่มากนัก นอกจากนั้นเทคนิคที่จะได้กล่าวถึงในหัวข้อถัดไปทำให้สามารถทวนสอบผลของตัวบ่งชี้การทดที่มีต่อเอแอลยูได้

5.5.5 กำหนดค่าฟังก์ชันของฟังก์ชันที่ไม่ต้องตีความ

ในหัวข้อที่แล้วได้กล่าวถึงการตัดผลของตัวบ่งชี้การทดที่มีต่อวงจรเอแอลยูของหน่วยประมวลผล สาเหตุที่ต้องทำเช่นนี้เนื่องจากถ้าไม่ตัดผลของตัวบ่งชี้การทดที่มีต่อเอแอลยูจะทำให้สถานะที่เป็นไปได้ของวงจรมากเกินกว่าที่โปรแกรมช่วยทวนสอบจะทำงานได้ ดังนั้นในขั้นต้นจึงใช้วิธีการนี้เพื่อให้สามารถทวนสอบหน่วยประมวลผลได้ หลังจากนั้นจึงทำการวิเคราะห์เพื่อปรับปรุงกระบวนการทวนสอบให้สามารถทวนสอบผลของตัวบ่งชี้การทดที่มีต่อเอแอลยู จึงพบว่าในกรณีที่ตัวบ่งชี้การทดมีผลต่อเอแอลยู ฟังก์ชันที่ไม่ต้องตีความของเอแอลยูจะมีตัวแปรสถานะมากถึง 48 ตัว (ในเอแอลยูมีฟังก์ชันที่ไม่ต้องตีความสามชุดสำหรับ ผลลัพธ์ของเอแอลยู, ตัวบ่งชี้การทด และตัวบ่งชี้ค่าศูนย์ แต่ละชุดมีตัวแปรสถานะ 16 ตัวต่อชุดเนื่องจากเป็นแถวลำดับขนาดสี่มิติ เมื่อรวมสามชุดจึงมีตัวแปรสถานะทั้งหมด 48 ตัว) และเนื่องจากค่าฟังก์ชันของฟังก์ชันที่ไม่ต้องตีความไม่ได้ถูกกำหนดค่าไว้ โปรแกรมช่วยทวนสอบจึงทำการทวนสอบโดยเปลี่ยนค่าฟังก์ชันให้ครบทุกกรณี ซึ่งในที่นี้เป็นไปได้ถึง 2^{48} กรณี จึงเป็นสาเหตุทำให้ในกรณีที่ไม่ตัดผลของตัวบ่งชี้การทดที่มีต่อเอแอลยู โปรแกรมช่วยทวนสอบไม่สามารถทวนสอบหน่วยประมวลผลได้สำเร็จ

เมื่อพบว่าฟังก์ชันที่ไม่ต้องตีความของเอแอลยูมีตัวแปรสถานะ 48 ตัว ขณะที่ตัวแปรสถานะทั้งหมดของหน่วยประมวลผลมี 106 ตัว จะเห็นได้ว่าตัวแปรสถานะของฟังก์ชันที่ไม่ต้องตีความของเอแอลยูเป็น 40% ของตัวแปรสถานะทั้งหมด ด้วยเหตุนี้จึงเกิดแนวคิดที่จะกำหนดค่าฟังก์ชันที่แน่นอนให้กับฟังก์ชันที่ไม่ต้องตีความของเอแอลยูแทนที่จะปล่อยให้เป็นค่าใดก็ได้ เพราะจะทำให้สถานะที่เป็นไปได้ทั้งหมดของวงจรลดลงอย่างมาก

จากผลการทดลองพบว่าในกรณีที่ไม่กำหนดค่าฟังก์ชันให้กับฟังก์ชันที่ไม่ต้องตีความของเอแอยู จะไม่สามารถทวนสอบผลของตัวบ่งชี้การทดที่มีต่อเอแอลยูได้ และเมื่อทดลองกรณีที่ตัดผลของตัวบ่งชี้การทดที่มีต่อเอแอลยูโปรแกรมช่วยทวนสอบสามารถทวนสอบสำเร็จโดยใช้เวลาทำงาน 15 นาที และใช้หน่วยความจำประมาณ 500 เมกะไบต์ และเมื่อทดลองกำหนดค่าฟังก์ชันให้กับฟังก์ชันที่ไม่ต้องตีความของเอแอลยู พบว่าเวลาและหน่วยความจำที่ใช้ลดลงอย่างมาก คือสามารถทวนสอบเสร็จในเวลา 1 นาที และใช้หน่วยความจำเพียงแค่ 30 เมกะไบต์ นอกจากนั้นเมื่อกำหนดค่าฟังก์ชันให้กับฟังก์ชันที่ไม่ต้องตีความของเอแอลยูแล้วทำให้สามารถทวนสอบผลของตัวบ่งชี้การทดที่มีต่อเอแอลยู โดยเวลาและหน่วยความจำที่ได้แสดงไปนั้นคือกรณีที่รวมผลของตัวบ่งชี้การทดที่มีต่อเอแอลยูไปด้วย การทดลองทั้งหมดนี้ทดลองโดยใช้เครื่องคอมพิวเตอร์ที่มีหน่วยประมวลผลรุ่นเพนเทียมทรีความเร็วสัญญาณนาฬิกา 1 กิกะเฮิรตซ์และมีหน่วยความจำขนาด 2 กิกะไบต์ทำงานโดยใช้ระบบปฏิบัติการลินุกซ์

จากผลการทดลองสามารถสรุปได้ว่า การกำหนดค่าฟังก์ชันที่แน่นอนให้กับฟังก์ชันที่ไม่ต้องตีความของเอแอลยูช่วยลดเวลาการทำงานและหน่วยความจำที่ต้องใช้ได้อย่างมาก แต่ก็อาจเกิดคำถามว่าการกำหนดค่าฟังก์ชันที่แน่นอนให้กับฟังก์ชันที่ไม่ต้องตีความของเอแอลยูจะทำให้ความน่าเชื่อถือของผลการทวนสอบลดลงหรือไม่ ซึ่งในจุดนี้สามารถตอบได้ว่าไม่ทำให้ความน่าเชื่อถือของผลการทวนสอบลดลง เพราะเมื่อพิจารณาความจริงที่ว่าวงจรเอแอลยูเป็นวงจรเชิงผสม ซึ่งถ้าพิจารณาเป็นฟังก์ชันก็เป็นฟังก์ชันที่ถูกกำหนดค่าฟังก์ชันที่แน่นอนไว้แล้ว ดังนั้นการกำหนดค่าฟังก์ชันให้กับฟังก์ชันที่ไม่ต้องตีความของเอแอลยูจึงเป็นเรื่องสมเหตุสมผล และเมื่อพิจารณาถึงการปล่อยค่าฟังก์ชันของฟังก์ชันที่ไม่ต้องตีความให้เป็นค่าใดก็ได้ เป็นการสร้างเงื่อนไขสำหรับการทวนสอบให้ยากเกินความจำเป็น เนื่องจากในความเป็นจริงวงจรเอแอลยูก็เป็นวงจรที่ถูกกำหนดฟังก์ชันการทำงานที่แน่นอนไว้แล้ว

ในตารางที่ 5.6 แสดงค่าฟังก์ชันซึ่งกำหนดให้กับฟังก์ชันที่ไม่ต้องตีความของเอแอลยู ด้านซ้ายของตารางคืออินพุตของเอแอลยู ขณะที่ด้านขวาของตารางแสดงค่าเอาต์พุตของเอแอลยู สัญญาณ *alu_op* เป็นตัวกำหนดคำสั่งสำหรับเอแอลยู และเนื่องจากการลดขนาดของสัญญาณข้อมูลเหลือเพียง 1 บิต สัญญาณ *alu_op* จึงมีค่าได้เพียง 2 ค่าคือ *'0'* และ *'1'* ในที่นี้จึงกำหนดให้ค่า *'0'* แทนคำสั่ง ADD และค่า *'1'* แทนคำสั่ง AND สาเหตุที่เลือก 2 คำสั่งนี้เนื่องจากวงจรเอแอลยูประกอบด้วยคำสั่ง 2 กลุ่ม คือกลุ่มคำสั่งคำนวณ (arithmetic instruction) และกลุ่มคำสั่งเชิงตรรก (logic instruction) ซึ่งคำสั่งคำนวณจะมีผลต่อค่าของตัวบ่งชี้การทด ขณะที่คำสั่งเชิงตรรกจะไม่มีผลต่อตัวบ่งชี้การทด และเนื่องจาก *alu_op* สามารถแทนได้เพียง 2 คำสั่งเท่านั้น

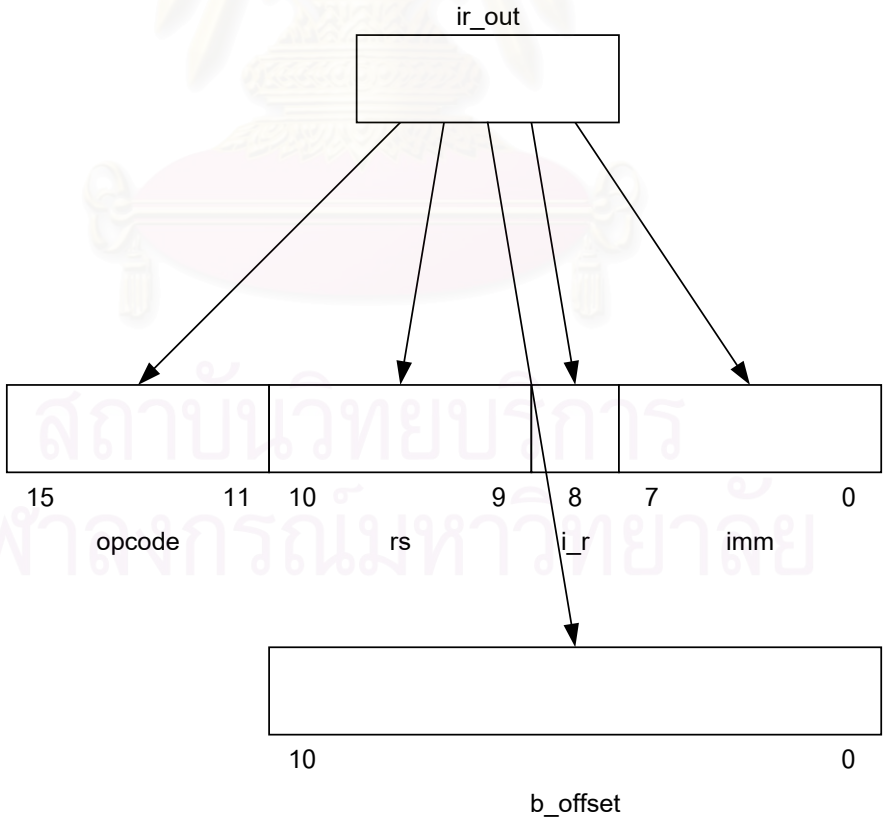
จึงเลือกคำสั่ง ADD เพื่อใช้แทนกลุ่มคำสั่งคำนวณ และเลือกคำสั่ง AND เพื่อใช้แทนกลุ่มคำสั่งเชิงตรรก

นอกจากนั้นยังพบว่า ฟังก์ชันที่ไม่ต้องตีความที่สามารถกำหนดค่าฟังก์ชันได้ คือ ฟังก์ชันที่ไม่ต้องตีความที่ใช้แทนวงจรเชิงผสมอันได้แก่ เอแอลยู, วงจรคำนวณค่าถัดไปสำหรับ *PC* และวงจรบวกสำหรับ *PC* ขณะที่ฟังก์ชันที่ไม่ต้องตีความที่ใช้กำหนดความสัมพันธ์ระหว่าง *IR* กับสัญญาณอื่นๆ ไม่สามารถกำหนดค่าฟังก์ชันได้ สาเหตุที่เป็นเช่นนี้เนื่องจากการลดขนาดของสัญญาณข้อมูลทำให้ความสำคัญของข้อมูลลดลง เป็นผลให้ฟังก์ชันการทำงานที่แท้จริงของวงจรเชิงผสมถูกลดความสำคัญลงไปด้วย การกำหนดค่าฟังก์ชันให้กับฟังก์ชันที่ไม่ต้องตีความของวงจรเชิงผสมจึงสมเหตุสมผล ขณะที่สัญญาณที่ได้จากฟังก์ชันของ *IR* ต้องนำไปใช้ในส่วนควบคุมของหน่วยประมวลผล และเนื่องจากการลดขนาดของสัญญาณข้อมูลทำให้ฟังก์ชันที่กำหนดความสัมพันธ์ระหว่าง *IR* กับสัญญาณอื่นๆมีกรณีที่เป็นไปได้น้อยกว่าความเป็นจริง จึงจำเป็นต้องปล่อยให้ค่าฟังก์ชันเหล่านั้นเป็นค่าใดก็ได้เพื่อช่วยให้สามารถทวนสอบได้ในกรณีที่หลากหลายขึ้น เพราะสัญญาณที่ได้จาก *IR* มีความสำคัญ ดังนั้นการกำหนดค่าฟังก์ชันให้กับฟังก์ชันที่ไม่ต้องตีความที่กำหนดความสัมพันธ์ระหว่าง *IR* กับสัญญาณอื่นๆจึงทำไม่ได้ เพราะจะทำให้ความน่าเชื่อถือของกระบวนการทวนสอบลดลง

รูปที่ 5.6 คือแผนภาพที่แสดงความสัมพันธ์ระหว่าง *IR* กับสัญญาณอื่นซึ่งจะเห็นได้ว่าสัญญาณอื่นเหล่านั้นเป็นบางบิตของ *IR* สัญญาณเหล่านี้ได้แก่ *opcode* ขนาดห้าบิต, *rs* (source register) ขนาดสองบิต, *i_r* (instruction mode) ขนาดหนึ่งบิต, *imm* (immediate value) ขนาดแปดบิต, และ *b_offset* (base offset) ขนาดสิบเอ็ดบิต เมื่อ *IR* ถูกลดขนาดเหลือเพียงหนึ่งบิต การแบ่งบางบิตจึงเป็นไปไม่ได้ทำให้จำเป็นต้องใช้ฟังก์ชันที่ไม่ต้องตีความ และฟังก์ชันที่ไม่ต้องตีความในส่วนนี้ไม่ควรจะถูกกำหนดค่า เพราะถ้าถูกกำหนดค่าจะทำให้ไม่สามารถทวนสอบได้ในกรณีที่หลากหลาย เช่นกรณีของสัญญาณ *opcode* ซึ่งมีค่าที่เป็นไปได้ถึง 20 ค่า (จำนวนคำสั่งที่มีของหน่วยประมวลผล) ถ้ากำหนดค่าฟังก์ชันที่ไม่ต้องตีความจะทำให้สามารถทวนสอบค่าใน *opcode* ได้แค่สองค่า ตัวอย่างเช่น *IR = '0'* $\rightarrow$ *opcode = "ADD"*, *IR = '1'* $\rightarrow$ *opcode = "SUB"* แต่ถ้าหากปล่อยไว้ไม่กำหนดค่าฟังก์ชันที่ไม่ต้องตีความ โปรแกรมคาเดนซ์เอสเอมวีจะทำทวนสอบทุกกรณีที่เป็นไปได้ ทำให้ทุกค่าที่เป็นไปได้ของ *opcode* ถูกทวนสอบ

ตารางที่ 5.6 ค่าฟังก์ชันของฟังก์ชันที่ไม่ต้องตีความของเอแอลยู

	Input				Output		
	alu_op	a_in	b_in	c_in	s_out	c_out	z_out
ADD	0	0	0	0	0	0	1
	0	0	0	1	1	0	0
	0	0	1	0	1	0	0
	0	0	1	1	0	1	1
	0	1	0	0	1	0	0
	0	1	0	1	0	1	1
	0	1	1	0	0	1	1
	0	1	1	1	1	1	0
AND	1	0	0	0	0	0	1
	1	0	0	1	0	0	1
	1	0	1	0	0	0	1
	1	0	1	1	0	0	1
	1	1	0	0	0	0	1
	1	1	0	1	0	0	1
	1	1	1	0	1	0	0
	1	1	1	1	1	0	0



รูปที่ 5.6 แผนภาพแสดงความสัมพันธ์ระหว่าง *IR* กับสัญญาณอื่น

5.6 สรุป

บทนี้ได้นำเสนอรายละเอียดของกระบวนการทวนสอบ โดยเริ่มต้นจากการนำเสนอลักษณะของการทวนสอบว่าเป็นอย่างไร สิ่งที่กระบวนการทวนสอบสามารถรับประกันได้และสมมติฐานที่ใช้ในการทวนสอบ ในส่วนนี้ทำให้เห็นภาพว่าการทวนสอบเป็นอย่างไร และเข้าใจว่าการทวนสอบมีขีดความสามารถแค่ไหน ต่อมาได้นำเสนอวิธีทวนสอบแบบลำดับขั้น ซึ่งเป็นการแบ่งงานของการทวนสอบทั้งหมดออกเป็นงานย่อย โดยทำการทวนสอบคุณสมบัติของหน่วยประมวลผลทีละอย่าง การทำเช่นนี้ทำให้กระบวนการทวนสอบง่ายขึ้น และการหาสาเหตุของปัญหาเมื่อเกิดข้อผิดพลาดของหน่วยประมวลผลทำได้ง่ายขึ้นด้วย หลังจากนั้นได้นำเสนอปัญหาการเพิ่มอย่างรวดเร็วของสถานะ โดยแสดงข้อมูลจำนวนตัวแปรสถานะเมื่อสัญญาณข้อมูลมีขนาดต่างๆกัน ซึ่งทำให้เห็นได้ว่าจำนวนตัวแปรสถานะเป็นฟังก์ชันเอ็กซ์โพเนนเชียลของขนาดสัญญาณข้อมูล และหัวข้อสุดท้ายที่สำคัญของบทนี้ก็ได้นำเสนอเทคนิควิธีการที่ใช้แก้ปัญหาการเพิ่มอย่างรวดเร็วของสถานะ ซึ่งได้แก่การลดขนาดของสัญญาณข้อมูลจาก 16 บิตเหลือเพียง 1 บิต, การใช้ฟังก์ชันที่ไม่ต้องตีความเพื่อแทนวงจรเชิงผสมและส่วนที่สร้างสัญญาณต่างๆที่เกี่ยวข้องกับ *IR*, การทวนสอบแบบองค์ประกอบซึ่งจะทวนสอบความถูกต้องของคุณสมบัติที่ต้องการครั้งละคุณสมบัติโดยกำหนดให้คุณสมบัติอื่นๆเป็นจริง, การตัดผลของตัวบ่งชี้การทดที่มีต่อเอแอลยู ซึ่งถ้าไม่ตัดผลของตัวบ่งชี้การทด ในเอแอลยูจะมีจำนวนตัวแปรสถานะถึง 48 ตัว หรือมีกรณีที่เป็นไปได้มากถึง 2^{48} กรณีทำให้การทวนสอบไม่สำเร็จ, และเทคนิคท้ายสุดการกำหนดค่าฟังก์ชันของฟังก์ชันที่ไม่ต้องตีความในเอแอลยู ซึ่งทำให้การทวนสอบผลของตัวบ่งชี้การทดที่มีต่อเอแอลยูเป็นไปได้ ขณะเดียวกันจำนวนตัวแปรสถานะก็ลดลงอย่างมาก จนกระทั่งกระบวนการทวนสอบสามารถสำเร็จในเวลาเพียงหนึ่งนาทีเท่านั้น

บทที่ 6

ผลการทวนสอบ

ในบทนี้นำเสนอผลการทวนสอบจากกระบวนการทวนสอบในวิทยานิพนธ์นี้ โดยหัวข้อ 6.1 นำเสนอข้อผิดพลาดที่พบในรายละเอียดการออกแบบของหน่วยประมวลผล หัวข้อ 6.2 นำเสนอการทดลองใส่ข้อผิดพลาดในหน่วยประมวลผลเพื่อทดสอบประสิทธิภาพของการทวนสอบ และในหัวข้อ 6.3 นำเสนอเวลาและหน่วยความจำที่ใช้ในกระบวนการทวนสอบ

6.1 ข้อผิดพลาดในหน่วยประมวลผล

จากกระบวนการทวนสอบพบว่ารายละเอียดการออกแบบของหน่วยประมวลผลมีข้อผิดพลาดอยู่ นั่นคือ พบข้อผิดพลาดในคำสั่ง COM (complement) โดยที่ตามสถาปัตยกรรมชุดคำสั่งกำหนดให้ผลการทำงานของคำสั่งนี้กระทบต่อค่าของตัวบ่งชี้ค่าศูนย์ แต่โปรแกรมช่วยทวนสอบสามารถตรวจสอบได้ว่า คำสั่งนี้ในรายละเอียดการออกแบบไม่มีผลกระทบต่อค่าของตัวบ่งชี้ศูนย์ นอกจากนั้นยังพบว่าในคำสั่ง JNZ (jump if not zero) และ JNC (jump if not carry) ถูกกำหนดในรายละเอียดการออกแบบให้กระโดด เมื่อค่าตัวบ่งชี้ค่าศูนย์และค่าตัวบ่งชี้การทดตามลำดับมีค่าเป็น *'1'* ทั้งที่ในสถาปัตยกรรมชุดคำสั่งกำหนดให้ทำการกระโดดเพื่อค่าตัวบ่งชี้เป็น *'0'* ซึ่งหลังจากแก้ไขรายละเอียดการออกแบบแล้วผลการทวนสอบก็ออกมาถูกต้อง นั่นคือรายละเอียดการออกแบบสามารถทำงานได้ตามสถาปัตยกรรมชุดคำสั่งที่กำหนดไว้ สาเหตุที่พบข้อผิดพลาดเพียงสามข้อเนื่องจากรายละเอียดการออกแบบของหน่วยประมวลผลได้รับการทวนสอบด้วยวิธีการจำลองการทำงานมาแล้ว ข้อผิดพลาดจำนวนไม่น้อยถูกพบและแก้ไขก่อนการทวนสอบด้วยวิธีการทวนสอบรูปนัย

6.2 การทดลองใส่ข้อผิดพลาดในหน่วยประมวลผล

นอกเหนือจากข้อผิดพลาดที่พบในรายละเอียดการออกแบบของหน่วยประมวลผล ในที่นี้ยังได้ทำการทดลองใส่ข้อผิดพลาดต่างๆลงในรายละเอียดการออกแบบเพื่อทดลองการทำงานของการทวนสอบ ข้อผิดพลาดที่ทำการใส่แบ่งได้เป็นสี่กลุ่ม คือ ข้อผิดพลาดเกี่ยวกับการต่อ

สายสัญญาณภายในส่วนทางเดินข้อมูล, ข้อผิดพลาดเกี่ยวกับการกำหนดสัญญาณควบคุมของหน่วยควบคุม, ข้อผิดพลาดเกี่ยวกับการทำงานของส่วนที่เป็นเครื่องสถานะ (state machine) ภายในหน่วยควบคุม, และข้อผิดพลาดภายในมอดูลต่างๆของส่วนทางเดินข้อมูล ข้อผิดพลาดทั้งหมดที่ถูกใส่ในรายละเอียดมีทั้งหมด 20 จุด โดยที่แบ่งเป็น 4 กลุ่ม กลุ่มละ 5 จุด และในการทดลองหนึ่งครั้งจะใส่ข้อผิดพลาดเพียงหนึ่งจุดเท่านั้น

รูปที่ 6.1 แสดงข้อผิดพลาดที่ถูกใส่ในส่วนทางเดินข้อมูล ซึ่งก็คือข้อผิดพลาดกลุ่มแรกที่เกี่ยวกับการต่อสายสัญญาณในส่วนทางเดินข้อมูล จากรูปจะเห็นได้ว่ามีข้อผิดพลาดอยู่ห้าจุด จุดแรกเป็นการเปลี่ยนสายสัญญาณที่จะเข้า *PC* จากเดิมที่เป็น *next_pc_1* เปลี่ยนเป็น *next_pc* แทน จุดที่สองเปลี่ยนอินพุตของวงจรบวกตัวที่สองของ *PC* จากเดิมที่เป็น *pc_out_1* เปลี่ยนเป็น *pc_out* จุดที่สามเปลี่ยนสัญญาณควบคุมของกลุ่มเรจิสเตอร์ โดยสลับสายสัญญาณ *a_sel* และ *ld_sel* จุดที่สี่ทำการเปลี่ยนสัญญาณควบคุมของอุปกรณ์รวมสัญญาณที่เลือกสัญญาณเข้าสู่ *next_pc_1* จากเดิมที่เป็น *pc_sel1* เปลี่ยนเป็น *pc_sel* และจุดที่ห้าทำการสลับอินพุตของอุปกรณ์รวมสัญญาณที่เลือกสัญญาณเข้าสู่ *mem_out* โดยสลับสายสัญญาณของ *next_pc_1* กับ *b_out*

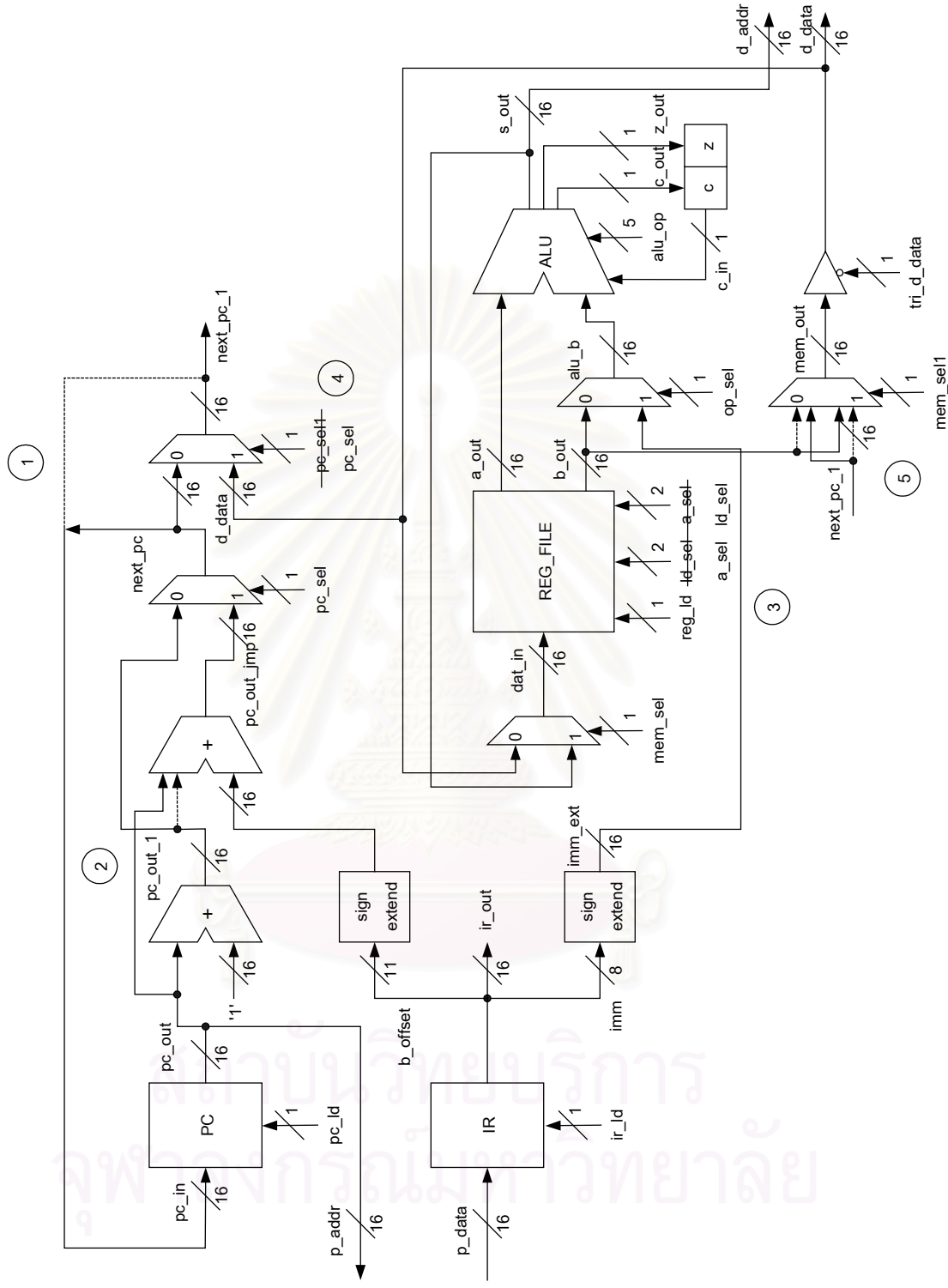
รูปที่ 6.2 แสดงข้อผิดพลาดเกี่ยวกับการกำหนดสัญญาณควบคุมของหน่วยควบคุม โดยในรูปแสดงส่วนของรหัสต้นฉบับภาษาเอสเอ็มวีของส่วนที่ใช้กำหนดสัญญาณควบคุมในรายละเอียดการออกแบบ และในรูปได้แสดงข้อผิดพลาดที่ถูกใส่เข้าไป จุดแรกเป็นการแก้การกำหนดสัญญาณ *op_sel* โดยแก้เงื่อนไขในกรณีที่สามของคำสั่ง *switch* จากเดิมที่เป็น *op_sel:= inst_mode;* ให้เป็น *op_sel:=0;* จุดที่สองเป็นส่วนที่ใช้กำหนดสัญญาณ *a_sel* โดยแก้การตรวจสอบเงื่อนไขของคำสั่ง *if* จากเดิมที่เป็น *if (inst_mode=1)* ให้เป็น *if (inst_mode=0)* จุดที่สามเป็นการแก้การกำหนดสัญญาณ *ld_sel* โดยแก้เงื่อนไขของกรณี *default* ของคำสั่ง *switch* จากเดิมที่เป็น *ld_sel:=rs;* ให้เป็น *ld_sel:=RA;* จุดที่สี่เป็นส่วนที่กำหนดสัญญาณ *pc_sel* โดยแก้ส่วนตรวจสอบเงื่อนไขของคำสั่ง *switch* โดยกรณีที่สองแก้จากเดิมที่ต้องการเงื่อนไขเป็น JNZ ให้เป็น JNC และกรณีที่สามแก้จาก JNC เป็น JNZ และจุดที่ห้าเป็นการแก้การกำหนดสัญญาณควบคุมของสัญญาณ *mem_sel1* โดยแก้กรณีที่หนึ่งและสองของคำสั่ง *switch* จากเดิมที่เป็น *mem_sel1:=1;* ให้เป็น *mem_sel:=0;*

รูปที่ 6.3 แสดงข้อผิดพลาดเกี่ยวกับการทำงานของส่วนที่เป็นเครื่องสถานะภายในหน่วยควบคุม ในรูปแสดงแผนภาพสถานะของเครื่องสถานะภายในหน่วยควบคุมพร้อมทั้งระบุสิ่งที่ทำการแก้ไข จุดแรกทำการแก้ไขการเปลี่ยนสถานะจากสถานะ ST_EXEC กรณีคำสั่ง CLC

และ STC จากเดิมที่ไปยังสถานะ ST_FETCH0 ให้ไปยังสถานะ ST_MEMACC แทน จุดที่สองทำการแก้ไขให้สถานะ ST_MEMACC ในทุกกรณีเพิ่มคำสั่ง *ir_ld:=1;* เข้าไป จุดที่สามแก้ไขการเปลี่ยนสถานะจากสถานะ ST_MEMACC กรณี default จากเดิมที่ไปยังสถานะ ST_WRBACK ให้ไปยัง ST_FETCH0 แทน จุดที่สี่ทำการตัดคำสั่ง *pc_ld:=1;* ออกจากทุกกรณีของสถานะ ST_WRBACK และจุดที่ห้าทำการตัดคำสั่ง *z:=z_out;* ออกจากทุกกรณีของสถานะ ST_WRBACK

รูปที่ 6.4 แสดงข้อผิดพลาดภายในมอดูลต่างๆของส่วนทางเดินข้อมูล ในรูปแสดงส่วนของรหัสต้นฉบับภาษาเอสเอมวี โดยตัดบางส่วนมาจากแต่ละมอดูลที่ถูกแก้ไข จุดแรกทำการแก้ไขการตรวจสอบเงื่อนไขภายในมอดูล *IR* โดยเปลี่ยนเงื่อนไขการเก็บค่าจากเดิมที่เป็น *(ld=1)* ให้เป็น *(ld=0)* จุดที่สองทำการแก้ไขภายในมอดูลแรม โดยในกรณีการเขียนค่าปกติจะเก็บค่าจาก *data_in* เปลี่ยนเป็นเก็บค่าจาก *data_out* แทน จุดที่สามทำการแก้ไขภายในมอดูล REGFILE โดยกรณีที่ *a_sel=RSP* จะนำค่าจาก *rsp* ออกมายังเอาต์พุต เปลี่ยนเป็นนำค่า *rt* ออกเอาต์พุตแทน จุดที่สี่เปลี่ยนฟังก์ชันที่ไม่ต้องตีความในมอดูล ALU โดยทำการเปลี่ยนดัชนีตัวที่สามของฟังก์ชัน *f_alu_sout* จากเดิมที่เป็น *b_in* ให้เป็น *a_in* แทน และจุดที่ห้าแก้ไขภายในมอดูล MUX21 จากเดิมที่เมื่อสัญญาณ *sel=0* นำค่า *a_in* ออกเอาต์พุตเปลี่ยนให้เป็น *b_in* แทน และกรณี *sel=1* ปกตินำค่า *b_in* ออกเอาต์พุตเปลี่ยนให้เป็น *a_in* แทน

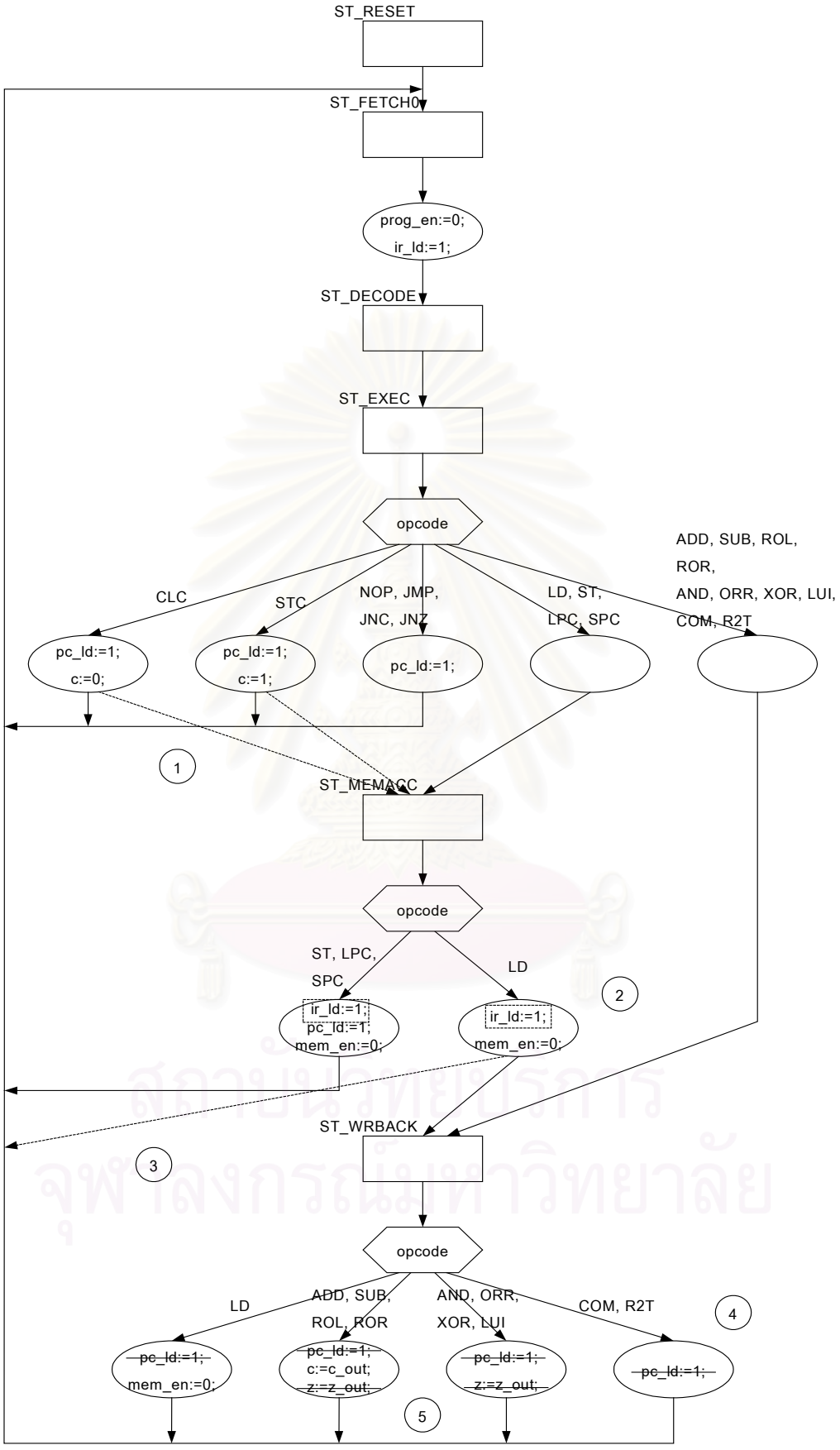
จากผลการทดลองโปรแกรมช่วยทวนสอบสามารถตรวจพบข้อผิดพลาดในการทดลองทุกครั้ง



รูปที่ 6.1 ข้อผิดพลาดที่ถูกใส่ในส่วนทางเดินข้อมูล

```
switch(opcode) {
      {LD, ST}                      : op_sel := 1;
      LUI                           : op_sel := 1;
      {ADD, SUB, AND, ORR, XOR}     : op_sel := 0;
                                    //op_sel := inst_mode;
      default                       : op_sel := 0;
};

switch(opcode) {
      {ST, LD}    : if (inst_mode = 0) // if (inst_mode = 1)
                                         a_sel := RB;
                    else                 a_sel := RSP;
      {LPC, SPC}  : a_sel := RSP;
      default     : a_sel := rs;
};

switch(opcode) {
      R2T         : ld_sel := RT;
      Default     : ld_sel := RA; // ld_sel := rs;
};

switch(opcode) {
      JMP         : pc_sel := 1;
      JNC         : // JNZ
                     if (z = 1) pc_sel := 1; else pc_sel := 0;
      JNZ         : // JNC
                    if (c = 1) pc_sel := 1; else pc_sel := 0;
      Default     : pc_sel := 0;
};

switch(opcode) {
      {ST, SPC}   : mem_nwr := 0;
      default     : mem_nwr := 1;
};

switch(opcode) {
      {ST, SPC}   : { mem_sel := 0; tri_d_data := 0; }
      LD          : { mem_sel := 0; tri_d_data := 1; }
      default     : { mem_sel := 1; tri_d_data := 1; }
};

switch(opcode) {
      LPC         : { mem_sel1 := 0; pc_sel1 := 1; } //mem_sel1 := 1;
      SPC         : { mem_sel1 := 0; pc_sel1 := 0; } //mem_sel1 := 1;
      default     : { mem_sel1 := 0; pc_sel1 := 0; }
};
```

รูปที่ 6.2 ข้อผิดพลาดเกี่ยวกับการกำหนดสัญญาณควบคุมของหน่วยควบคุม



รูปที่ 6.3 ข้อผิดพลาดเกี่ยวกับการทำงานของส่วนที่เครื่องสถานะภายในหน่วยควบคุม

```
/*---------- IR ----------*/
next(ir_out) :=
      case {
      (ld = 0)    : //(ld = 1)
                    ir_in;
      default     : ir_out;
      };

/*---------- RAM ----------*/
default
      forall(i in WORD)
            next(mem[i]) := mem[i];
in
      if ((ncs=0)&(nwr=0)) {
            next(mem[address]) := data_out;
            //next(mem[address]) := data_in;

      }

/*---------- REGFILE ----------*/
a_out :=    switch(a_sel) {
            RA    : ra;
            RB    : rb;
            RSP   : rt; //rsp;
            RT    : rt;
            };

/*---------- ALU ----------*/
s_out := f_alu_sout[alu_op][a_in][a_in][c_in];
//s_out := f_alu_sout[alu_op][a_in][b_in][c_in];
z_out := f_alu_zout[alu_op][a_in][b_in][c_in];
c_out := f_alu_cout[alu_op][a_in][b_in][c_in];

/*---------- MUX21 ----------*/
s_out :=    switch(sel) {
            0     : b_in; //a_in;
            1     : a_in; //b_in;
            };
```

1 2 3 4 5

รูปที่ 6.4 ข้อผิดพลาดภายในมอดูลของส่วนทางเดินข้อมูล

6.3 เวลาและหน่วยความจำที่ใช้

อย่างที่ได้กล่าวไปแล้วว่า ในวิทยานิพนธ์หัวข้อนี้ใช้โปรแกรมคาเดนค์เอสเอมวีช่วยทำการทวนสอบประมวลผล โดยโปรแกรมนี้ทำงานบนคอมพิวเตอร์ที่ใช้ระบบปฏิบัติการลีนุกซ์ ซึ่งทำงานบนคอมพิวเตอร์ที่ใช้หน่วยประมวลผลรุ่นเพนเทียมทรีความเร็วสัญญาณนาฬิกา 1 กิกะเฮิรตซ์ และมีหน่วยความจำขนาด 2 กิกะไบต์ รายงานเวลาและหน่วยความจำที่ใช้แบ่งเป็น 3 กรณี คือ

- ในกรณีที่ยังไม่ได้ทำการกำหนดค่าของฟังก์ชันที่ไม่ต้องตีความ โปรแกรมไม่สามารถทวนสอบผลของตัวบ่งชี้การทดที่มีต่อเอแอลยูได้
- เมื่อตัดผลดังกล่าว กรณีนี้โปรแกรมช่วยทวนสอบทำงานเสร็จในเวลาประมาณ 15 นาที และใช้หน่วยความจำประมาณ 500 เมกะไบต์
- และสำหรับกรณีที่ทำการกำหนดค่าให้กับฟังก์ชันที่ไม่ต้องตีความของมอดูล *alu* โปรแกรมสามารถทวนสอบผลของตัวบ่งชี้การทดที่มีต่อเอแอลยูได้ กรณีนี้โปรแกรมช่วยทวนสอบทำงานเสร็จในเวลาเพียง 1 นาที และใช้หน่วยความจำแค่ 30 เมกะไบต์เท่านั้น

เห็นได้ว่าการกำหนดค่าฟังก์ชันที่ไม่ต้องตีความเอแอลยูสามารถลดปัญหาการเพิ่มอย่างรวดเร็วของสถานะได้เป็นอย่างดี

บทที่ 7

สรุปผลการวิจัยและข้อเสนอแนะ

7.1 สรุปผลการวิจัย

ในวิทยานิพนธ์นี้แสดงถึงตัวอย่างของการประยุกต์ใช้วิธีการทวนสอบรูปนัยเพื่อทวนสอบหน่วยประมวลผลที่ถูกออกแบบให้ใช้ในระบบเว็บเซิร์ฟเวอร์แบบฝังตัว ซึ่งการทวนสอบกระทำกับรายละเอียดการออกแบบในระดับถ่ายโอนเรจิสเตอร์ที่มีรายละเอียดมากพอที่จะนำไปสังเคราะห์วงจรได้ โดยการทวนสอบใช้โปรแกรมคาเดนค์เอสเอมวีซึ่งทำงานโดยใช้เทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์

การทวนสอบหน่วยประมวลผลในระดับถ่ายโอนเรจิสเตอร์โดยใช้เทคนิคการตรวจสอบแบบจำลองเชิงสัญลักษณ์เป็นสิ่งที่ยังไม่เคยมีผู้ใดทำมาก่อน เนื่องจากเมื่อความซับซ้อนของวงจรมีมากขึ้นเทคนิคดังกล่าวจะใช้เวลาในการทำงานและหน่วยความจำของคอมพิวเตอร์มากจนไม่สามารถทำการทวนสอบได้สำเร็จ ในวิทยานิพนธ์นี้จึงได้นำเสนอการประยุกต์ใช้เทคนิคหลายอย่างเพื่อลดความซับซ้อนของหน่วยประมวลผล จนทำให้โปรแกรมช่วยทวนสอบสามารถทำงานได้เสร็จในเวลาที่สมเหตุผล เทคนิคที่ช่วยลดความซับซ้อนได้แก่ การลดขนาดของสัญญาณข้อมูลจาก 16 บิตเหลือเพียง 1 บิต, การใช้ฟังก์ชันที่ไม่ต้องตีความเพื่อแทนวงจรเชิงผสมและใช้ฟังก์ชันที่ไม่ต้องตีความกับสัญญาณที่เกี่ยวข้องกับ *IR*, การทวนสอบแบบองค์ประกอบที่ทำการทวนสอบคุณสมบัติทีละตัวโดยกำหนดให้คุณสมบัติอื่นเป็นจริงก่อน, การตัดผลของตัวบ่งชี้การทดที่มีต่อเอแอลยู, และการกำหนดค่าของฟังก์ชันที่ไม่ต้องตีความของเอแอลยู จากการใช้เทคนิคเหล่านี้เพื่อแก้ปัญหาการเพิ่มอย่างรวดเร็วของสถานะ ทำให้กระบวนการทวนสอบสามารถทำได้สำเร็จในเวลาที่สมเหตุผล แต่อย่างไรก็ตามการใช้เทคนิคต่างๆเหล่านี้ก็ทำให้เกิดปัญหาและข้อจำกัดอยู่บ้างดังได้แสดงไว้แล้วในหัวข้อที่ 5.5 นอกจากนั้นสิ่งที่น่าสนใจซึ่งวิทยานิพนธ์นี้ได้นำเสนออีกประการหนึ่งคือ การทวนสอบแบบลำดับขั้น ซึ่งทำการแบ่งกระบวนการทวนสอบออกเป็นหลายขั้นตอนโดยแต่ละขั้นตอนจะมีรายละเอียดเพิ่มขึ้นจากขั้นตอนแรกไปจนถึงขั้นตอนสุดท้าย ซึ่งวิธีการนี้ช่วยทำให้การหาสาเหตุของข้อผิดพลาดในวงจรทำได้ง่ายขึ้น

ประโยชน์ของวิทยานิพนธ์นี้คือ ช่วยเพิ่มความมั่นใจของหน่วยประมวลผลที่ถูกทวนสอบ โดยที่ในกระบวนการทวนสอบสามารถพบข้อผิดพลาดในรายละเอียดการออกแบบซึ่งเป็นข้อผิดพลาดที่ไม่ถูกพบในการทวนสอบโดยวิธีการจำลองการทำงาน ประโยชน์อีกประการหนึ่ง

คือ แสดงแนวทางในการประยุกต์วิธีการทวนสอบรูปนัยเพื่อใช้ทวนสอบวงจรในระดับถ่ายโอนเรจิสเตอร์ ซึ่งด้วยความอัตโนมัติของวิธีดังกล่าวและการที่ไม่ต้องใช้ผู้เชี่ยวชาญทางคณิตศาสตร์เพื่อทวนสอบ ทำให้แนวทางนี้สามารถนำไปใช้จริงในทางอุตสาหกรรมได้

7.2 **ข้อเสนอแนะ**

งานวิจัยชิ้นนี้เป็นการเริ่มต้นนำวิธีการตรวจสอบแบบจำลองเชิงสัญลักษณ์มาใช้เพื่อทวนสอบหน่วยประมวลผลในระดับการถ่ายโอนเรจิสเตอร์ซึ่งยังไม่มีผู้ใดทำมาก่อน ดังนั้นจึงยังมีอีกหลายสิ่งที่สามารถปรับปรุงต่อไปได้ และยังมีอีกหลายอย่างที่ควรทำต่อไปอีก สิ่งที่ควรทำต่อเนื่องไปจากวิทยานิพนธ์นี้ได้แก่

- ทำการสร้างโปรแกรมที่แปลงจากรายละเอียดการออกแบบในภาษาเอสเอมวีให้เป็นภาษาเวอริล็อกซึ่งเป็นภาษาที่ใช้ออกแบบวงจรและสามารถนำไปใช้สังเคราะห์วงจรได้ และทำการสร้างโปรแกรมที่แปลงข้อกำหนดวงจรในภาษาเอสเอมวีซึ่งอธิบายสถาปัตยกรรมชุดคำสั่งให้เป็นโปรแกรมจำลองการทำงาน (Simulator) สาเหตุที่สร้างโปรแกรมช่วยแปลงนี้เพื่อให้กระบวนการทวนสอบสามารถเชื่อมโยงต่อไปยังการใช้งานจริงได้

- ปัจจุบันส่วนข้อกำหนดวงจรในภาษาเอสเอมวีถูกเขียนในลักษณะที่อ้างอิงจากส่วนรายละเอียดการออกแบบ ดังนั้นจึงควรศึกษาหาวิธีการทวนสอบที่ทำให้ข้อกำหนดวงจรสามารถเป็นอิสระจากรายละเอียดการออกแบบ เพื่อให้สามารถเขียนข้อกำหนดวงจรโดยอ้างอิงจากเฉพาะสถาปัตยกรรมชุดคำสั่งเท่านั้น ซึ่งจะทำให้กระบวนการทวนสอบเหมาะสมในการใช้งานจริงมากขึ้น

- โปรแกรมคาเดนซ์สามารถรับรหัสต้นฉบับด้วยภาษาซึ่งเป็นเซตย่อยของภาษาเวอริล็อก ดังนั้นจึงควรศึกษาการเขียนรายละเอียดการออกแบบด้วยเซตย่อยดังกล่าวของภาษาเวอริล็อก ซึ่งจะทำให้รายละเอียดการออกแบบที่ผ่านการทวนสอบแล้วสามารถนำไปใช้ในการสังเคราะห์วงจรได้สะดวกยิ่งขึ้น

- ทดลองนำโปรแกรมคาเดนซ์เอสเอมวี และเทคนิควิธีการต่างๆที่ได้นำเสนอในวิทยานิพนธ์นี้ไปใช้ทวนสอบหน่วยประมวลผลตัวอื่น หรือทวนสอบวงจรประเภทอื่น ซึ่งอาจจะทำให้ได้พบปัญหาอื่นที่แตกต่างจากที่วิทยานิพนธ์นี้ได้พบ

รายการอ้างอิง

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. 1974. The Design and Analysis of Computer Algorithms. Addison Wesley.

Appenzeller, D. P., and Kuehlmann A. 1995. Formal verification of a PowerPC microprocessor. Proceedings of the International Conference on Computer Design (ICCD'95), pp. 79-84. Texas.

Barakatain, L., Tahar, S., Lamarche, J., and Gendreau, J. 2001. Practical approaches to the verification of a Telecom Megacell using FormalCheck. Proceedings of the 2001 Conference on Great Lakes Symposium on VLSI, United States of America.

Beatty, D. L. 1993. A methodology for formal hardware verification with application to microprocessor. Doctoral dissertation, Computer Science Department, Carnegie Mellon University.

Beer, I., Ben-David, S., Eisner, C., and Landver, A. 1996. RuleBase: an industry-oriented formal verification tool. Proceedings of the 33rd Design Automation Conference, pp. 655-660.

Bell Labs Design Automation. 1998. FormalCheck user guide. Version 2.1. Lucent Technologies.

Biere, A., Clarke, E., Raimi, R., and Zhu, Y. 1999. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99), pp. 60-71.

Brock, B., and Hunt, W. A. 1991. Report on the formal specification and partial verification of the VIPER microprocessor. Proceedings of the 6th Annual Conference on Computer Assurance, Systems Integrity, Software Safety and Process Security (COMPASS'91), pp. 91-98. Maryland.

Brock, B. C., and Hunt, W. A. 1997. Formally specifying and mechanically verifying programs for the motorola complex arithmetic processor DSP. Proceedings of the IEEE International Conference on Computer Design (ICCD'97), pp. 31-36.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers 35 (August 1986) : 677-691.

Bryant, R. E. 1992. Symbolic boolean manipulation with ordered binary decision diagrams. ACM Computing Surveys 24 (September 1992) : 293-318.

Bryant, R. E., Beatty, D. L., and Seger, C. J. H. 1991. Formal hardware verification by symbolic ternary trajectory evaluation. Proceedings of the 28th Design Automation Conference, pp. 397-402.

Bundgen, R. 1993. Reduce the redex -> ReDuX. In C. Kirchner (ed.), Rewriting Techniques and Applications, Springer-Verlag.

Bundgen, R., Kunchlin, W., and Lauterbach, W. 1996. Verification of the Sparrow processor. Proceedings of the IEEE Symposium and Workshop on Engineering of Computer-Based Systems, pp. 86-93.

Burch, J. R. 1996. Techniques for verifying superscalar microprocessors. Proceedings of the 33rd Design Automation Conference (DAC'96).

Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. J. 1992. Symbolic model checking: 10^20 states and beyond. Information and Computation 98 (June 1992) : 142-70.

Choi, H., Yun, B., Lee, Y., and Roh, H. 2001. Model checking of S3C2400X industrial embedded SOC product. Proceedings of the 38th Conference on Design Automation Conference (DAC'2001), pp. 611-616.

Clarke, E. M., Emerson, E. A., and Sistla, A. P. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems 8 (1986) : 244-263.

Clarke, E. M., Grumberg, O., and Peled, D. A. 2001. Model Checking. United States of America : MIT Press.

Clarke, E. M., Long, D. E., and McMillan, K. L. 1989. Compositional model checking. Proceedings of the 4th IEEE Symposium on Logic in Computer Science, California.

Cory, W. E. 1983. Verification of hardware design correctness: symbolic execution techniques and criteria for consistency. Doctoral dissertation, Computer Science Department, Stanford University.

Goel, A., and Lee, W. R. 2000. Formal verification of an IBM CoreConnect processor local bus arbiter core. Proceedings of the 37th Conference on Design

Automation Conference (DAC'2000), pp. 196-200. California.

Gordon, M. J. 1988. HOL: A proof generating system for higher order logic. In G. Birtwhistle, and P. Subrahmanyam (eds.), VLSI Specification Verification and Synthesis, Kluwer Academic.

Graham, B. T. 1992. The SECD microprocessor: a verification case study. United States of America : Kluwer Academic.

Gurevich., Y. 1993. Evolving algebras: an attempt to discover semantics. In Rozenberg, and A. Salomaa (eds.), Current Trends in Theoretical Computer Science, World Scientific.

Hosabettu, R., Srivas, M., and Gopalakrishnan, G. 1999. Proof of correctness of a processor with reorder buffer using the completion functions approach. Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99).

Huggins, J. K., and Campenhout, D. V. 1998. Specification and verification of pipelining in the ARM2 RISC microprocessor. ACM Transactions on Design Automation of Electronic Systems 3 (October 1998) : 563-580.

Jhala, R., and McMillan, K. L. 2001. Microarchitecture verification by compositional model checking. Proceedings of the *13th International Conference on Computer Aided Verification (CAV'2001)*, pp. 18-22. France.

Kaufmann, M., and Moore, J. S. 1996. ACL2: An industrial strength version of Nqthm. Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS'96), pp. 23-34. Maryland.

Kropf, T. 1999. Introduction to formal hardware verification. Berlin : Springer.

McMillan, K. L. 1992 a. Symbolic model checking. Doctoral dissertation, Computer Science Department, Carnegie Mellon University.

McMillan, K. L. 1992 b. The SMV system. Pittsburgh : Carnegie Mellon University.

McMillan, K. L. 1998. Getting started with SMV: user's manual. Cadence Berkeley Laboratories.

Miller, S. P., and Srivas, M. 1995. Formal verification of the AAMP5 microprocessor: a case study in the industrial use of formal methods. Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques (WIFT'95), pp. 2-16.

Maryland.

Mir, A. A., Balakrishnan, S., and Tahar, S. 2000. Modeling and verification of embedded systems using Cadence SMV. Proceedings of the 2000 Canadian Conference on Electrical and Computer Engineering, pp. 179-183.

Owre, S., Rushby, J., and Shankar, N. 1992. PVS: a prototype verification system. Proceedings of the 11th International Conference on Automated Deduction (CADE'92), pp. 748-752. New York.

Patankar, V. A., Jain, A., and Bryant, R. E. 1999. Formal verification of an ARM processor. Proceedings of the12th International Conference On VLSI Design, pp. 282-287.

Piromsopa, K. 2000. Development of a reconfigurable embedded web server. Master's Thesis, Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University.

Rahinkar, P., Peterson, P., and Singh, L. 2001. System-on-a-chip verification: methodology and techniques. United States of America : Kluwer Academic.

Schonherr, J., Schreiber, I., Fordran, E., and Straube, B. 1999. Hazard checking in pipelined processor designs using symbolic model checking. Proceedings of the 25th EUROMICRO Conference, pp. 75-78.

Straube, B., Schonherr, J., Schreiber, I., and Fordran, E. 1996. VERDIS - A tool for verification of finite automata. Technical Report SFB 358-C1-3/96. TU Dresden. Germany.

Vakilotojar, V., and Beerel, P. A. 1997. RTL verification of timed asynchronous and heterogeneous systems using symbolic model checking. Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'97), pp. 181-188.

Windley, P. J. 1995. Formal modeling and verification of microprocessors. IEEE Transactions on Computers 44 (January 1995) : 54-72.

ภาคผนวก

ภาคผนวก ก

รหัสเทียมของหน่วยควบคุมในหน่วยประมวลผล

บทนี้นำเสนอรหัสเทียม (pseudo code) ซึ่งอธิบายการทำงานของหน่วยควบคุม โดยรหัสเทียมนี้ใกล้เคียงกับภาษาเวอริล็อกแต่ตัดรายละเอียดบางอย่างออกไป ในหน่วยควบคุมของหน่วยประมวลผลประกอบด้วยสองส่วนคือ ส่วนที่เป็นวงจรเชิงผสมและส่วนที่เป็นเครื่องสถานะจำกัด ในหัวข้อ ก.1 จะนำเสนอรหัสเทียมของส่วนที่เป็นวงจรเชิงผสม และในหัวข้อที่ ก.2 นำเสนอรหัสเทียมของส่วนที่เป็นเครื่องสถานะจำกัด

ก.1 ส่วนที่เป็นวงจรเชิงผสม

ในส่วนนี้จะทำการกำหนดค่าสัญญาณควบคุมซึ่งได้แก่ *alu_op*, *mem_nwr*, *ld_sel*, *mem_sel*, *tri_d_data*, *mem_sel1*, *pc_sel1*, *a_sel*, *op_sel*, และ *pc_sel*

สัญญาณ *alu_op*

```
case (opcode)
     SUB :      alu_op := 5'b 00101;
     AND :      alu_op := 5'b 00000;
     ORR :      alu_op := 5'b 00001;
     XOR :      alu_op := 5'b 00010;
     COM :      alu_op := 5'b 00011;
     ROL :      alu_op := 5'b 10110;
     ROR :      alu_op := 5'b 11110;
     LUI :      alu_op := 5'b 01111;
     R2T :      alu_op := 5'b 00110;
     default :  alu_op := 5'b 00100;
endcase
```

สัญญาณ *mem_nwr*

```
case (opcode)
     ST, SPC :  mem_nwr := 0;
     default :  mem_nwr := 1;
endcase
```

สัญญาณ *ld_sel*

```
case (opcode)
     R2T :      ld_sel := 3;    // 'T'
     default :  ld_sel := rs;
endcase
```

สัญญาณ *mem_sel* และ *tri_d_data*

```
case (opcode)
     ST, SPC :  mem_sel := 0;   tri_d_data := 0;
     LD :       mem_sel := 0;   tri_d_data := 1;
     default :  mem_sel := 1;   tri_d_data := 1;
endcase
```

สัญญาณ *mem_sel1* และ *pc_sel1*

```
case (opcode)
     LPC :      mem_sel1 := 1;  pc_sel1 := 1;
     SPC :      mem_sel1 := 1;  pc_sel1 := 0;
     default :  mem_sel1 := 0;  pc_sel1 := 0;
endcase
```

สัญญาณ *a_sel*

```
case (opcode)
     ST, LD :   if (i_r = 1)    a_sel := 1; // 'B'
                else            a_sel := 2; // 'SP'
     LPC, SPC :           a_sel := 2; // 'SP'
     default :            a_sel := rs;
endcase
```

สัญญาณ *op_sel*

```
case (opcode)
     LD, ST :                   op_sel := 1;
     LUI :                      op_sel := 1;
     ADD, SUB, AND, ORR, XOR :  op_sel := i_r;
     default :                  op_sel := 0;
endcase
```

สัญญาณ *pc_sel*

```
case (opcode)
     JMP :          pc_sel := 1;
     JNZ :     if (z = 1) pc_sel := 1;
               else       pc_sel := 0;
     JNC :     if (c = 1) pc_sel := 1;
               else       pc_sel := 0;
     default :       pc_sel := 0;
endcase
```

ก.2 ส่วนที่เป็นเครื่องสถานะจำกัด

ในส่วนนี้ทำการกำหนดสัญญาณควบคุมซึ่งได้แก่ *prog_en, pc_ld, ir_ld, mem_en,* และ *reg_ld*

ส่วนที่เก็บค่าสถานะปัจจุบัน

```
always @(posedge clock)
begin
     if (nreset = 0)  pstate := ST_RESET;
     else             pstate := nstate;
end
```

ส่วนที่เป็นสร้างค่าสถานะถัดไปและเอาต์พุต

```
case (pstate)
ST_RESET : begin
               prog_en := 1; pc_ld := 0; ir_ld := 0;
               mem_en := 1; reg_ld := 0;

               nstate := ST_FETCH0;
           end
ST_FETCH0 : begin
               prog_en := 0; pc_ld := 0; ir_ld := 1;
               mem_en := 1; reg_ld := 0;

               nstate := ST_DECODE;
            end
ST_DECODE : begin
               prog_en := 1; pc_ld := 0; ir_ld := 0;
               mem_en := 1; reg_ld := 0;
               // wait for all the decode circuit
```

```
                nstate := ST_EXEC;
             end
ST_EXEC : begin
                prog_en := 1; ir_ld := 0;
                mem_en := 1; reg_ld := 0;
                // wait for completion of ALU
                // and complete Branch/etc

                case (opcode)
                CLC, STC: begin
                                pc_ld := 1;
                                nstate := ST_FETCH0;
                          end
                NOP, JMP, JNC, JNZ: begin
                                pc_ld := 1;
                                nstate := ST_FETCH0;
                          end
                LD, ST, LPC, SPC: begin
                                pc_ld := 0;
                                nstate := ST_MEMACC;
                          end
                default: begin
                                pc_ld := 0;
                                nstate := ST_WRBACK;
                          end
                endcase

                // carry flag
                case (opcode)
                     CLC : c := 0;
                     STC : c := 1;
                     Default:   c := c; //old value
                endcase
           end
ST_MEMACC : begin
                prog_en := 1; ir_ld := 0;
                mem_en := 0; reg_ld := 0;

                case (opcode)
                ST, SPC:  begin
                                pc_ld := 1;
                                nstate := ST_FETCH0;
                          end
                LPC:  begin
                                pc_ld := 1;
                                nstate := ST_FETCH0;
                          end
                default:  begin
                                pc_ld := 0;
                                nstate := ST_WRBACK;
                          end
                endcase
           end
ST_WRBACK : begin
                prog_en := 1; pc_ld := 1; ir_ld := 0;
```

```
                reg_ld := 1;

                case (opcode)
                      LD :  mem_en := 0;
                      default:   mem_en := 1;
                endcase

                nstate := ST_FETCH0;

                // carry flag
                case (opcode)
                      ADD, SUB, ROL, ROR : c := c_out;
                            // output from ALU
                      default :              c := c;
                            // old value
                endcase

                // zero flag
                case (opcode)
                      ADD, SUB, AND, ORR,
                      XOR, ROL, ROR, LUI : z := z_out;
                            // output from ALU
                      default :              z := z;
                            // old value
                endcase
            end
default : begin
                prog_en := 1; pc_ld := 0; ir_ld := 0;
                mem_en := 1; reg_ld := 0;

                nstate := ST_RESET;
            end
endcase
```

ภาคผนวก ข

รหัสต้นฉบับภาษาเอสเอมวี

บทนี้นำเสนอรหัสต้นฉบับ (source code) ภาษาเอสเอมวีที่ใช้ในการทวนสอบของวิทยานิพนธ์นี้ รหัสต้นฉบับที่แสดงนี้เป็นชุดที่มีรายละเอียดการออกแบบเหมือนกับรายละเอียดในรหัสต้นฉบับภาษาเวอริล็อกจึงเป็นชุดที่ยังมีข้อผิดพลาดอยู่ ซึ่งในบทที่ 7 ผลการทวนสอบจะได้แสดงถึงข้อผิดพลาดที่พบ

```
/*-------------------- Type Definition --------------------*/

typedef     WORD  0..1;
typedef     ALUOP 0..1;
typedef     FLAG  0..1;

typedef     REG         {RA, RB, RSP, RT};

typedef     OPCODE      {NOP, JNZ, JNC, JMP, CLC,
                         STC, R2T, ADD, SUB, AND,
                         ORR, XOR, COM, ROL, ROR,
                         LUI,  LD,  ST, LPC, SPC};

typedef     STATE       {ST_RESET, ST_FETCH0,
                          ST_DECODE, ST_EXEC,
                          ST_MEMACC, ST_WRBACK};

/*-------------------- Main Module --------------------*/

module main() {

/*-------------- Uninterpreted Function ---------------*/

f_opcode        : array WORD of OPCODE;
f_imm           : array WORD of WORD;
f_instmode      : array WORD of boolean;
f_rs            : array WORD of REG;
f_boffset       : array WORD of WORD;
f_aluop         : array WORD of ALUOP;

next(f_opcode)    := f_opcode;
next(f_imm)       := f_imm;
next(f_instmode)  := f_instmode;
next(f_rs)        := f_rs;
next(f_boffset)   := f_boffset;
next(f_aluop)     := f_aluop;

/*-------------- Specification ---------------*/

pc_ab           : WORD;
ir_ab           : WORD;
```

```
c_ab              : FLAG;
z_ab              : FLAG;
ra_ab             : WORD;
rb_ab             : WORD;
rsp_ab            : WORD;
rt_ab             : WORD;

ain_ab            : WORD;
bin_ab            : WORD;
sout_ab           : WORD;
aluop_ab          : ALUOP;
ain2_ab           : WORD;

opcode_ab         : OPCODE;
imm_ab            : WORD;
inst_mode_ab      : boolean;
rs_ab             : REG;
boffset_ab        : WORD;

pstate_ab         : STATE;
nstate_ab         : STATE;

rom_mem_ab        : array WORD of WORD;
ram_mem_ab        : array WORD of WORD;

/*------------------------------*/

next(pstate_ab)   := nstate_ab;
next(rom_mem_ab)  := rom_mem_ab;

opcode_ab         := f_opcode[ir_ab];
imm_ab            := f_imm[ir_ab];
inst_mode_ab      := f_instmode[ir_ab];
rs_ab             := f_rs[ir_ab];
boffset_ab        := f_boffset[ir_ab];
aluop_ab          := f_aluop[ir_ab];

/*------------------------------*/

switch(rs_ab) {
RA    :     ain_ab := ra_ab;
RB    :     ain_ab := rb_ab;
RSP   :     ain_ab := rsp_ab;
RT    :     ain_ab := rt_ab;
}

if (inst_mode_ab = 1)   ain2_ab := rb_ab;
else                    ain2_ab := rsp_ab;

if (inst_mode_ab = 0)   bin_ab := rt_ab;
else                    bin_ab := imm_ab;

/*------------------------------*/

default{
      next(pc_ab)       := pc_ab;
      next(ir_ab)       := ir_ab;

      next(c_ab)        := c_ab;
      next(z_ab)        := z_ab;
```

```
      next(ra_ab)       := ra_ab;
      next(rb_ab)       := rb_ab;
      next(rsp_ab)      := rsp_ab;
      next(rt_ab)       := rt_ab;

      next(ram_mem_ab)  := ram_mem_ab;
}

in    switch(pstate_ab) {

ST_RESET :  {
            nstate_ab := ST_FETCH0;
            }

ST_FETCH0 : {
            next(ir_ab) := rom_mem_ab[pc_ab];

            nstate_ab := ST_DECODE;
            }

ST_DECODE : {
            /* Wait for all the decode circuit */

            nstate_ab := ST_EXEC;
            }

/*------------------------------*/

ST_EXEC : {
      /* Wait for completion of ALU / complete Branch / etc.*/

      /* carry flag */
      switch(opcode_ab) {
      CLC   :     next(c_ab) := 0;
      STC   :     next(c_ab) := 1;
      }

      switch(opcode_ab) {

      {CLC, STC}  :{
                        next(pc_ab) := nextpc.f_nextpc[pc_ab];

                        nstate_ab := ST_FETCH0;
                  }

      NOP         :{
                        next(pc_ab) := nextpc.f_nextpc[pc_ab];

                        nstate_ab := ST_FETCH0;
                  }

      JMP         :{

next(pc_ab) := adder.f_adder[nextpc.f_nextpc[pc_ab]][boffset_ab];

                        nstate_ab := ST_FETCH0;
                  }

      JNC         :{
```

```
if (c_ab = 1)
next(pc_ab) := adder.f_adder[nextpc.f_nextpc[pc_ab]][boffset_ab];
else
next(pc_ab) := nextpc.f_nextpc[pc_ab];
                        nstate_ab := ST_FETCH0;
                  }
      JNZ         :{
if (z_ab = 1)
next(pc_ab) := adder.f_adder[nextpc.f_nextpc[pc_ab]][boffset_ab];
else
next(pc_ab) := nextpc.f_nextpc[pc_ab];
                        nstate_ab := ST_FETCH0;
                  }
      {LD, ST, LPC, SPC}      : nstate_ab := ST_MEMACC;
      default                 : nstate_ab := ST_WRBACK;
            } /* end switch(opcode_ab) */
      } /* end ST_EXEC */
/*------------------------------*/
ST_MEMACC : {
      switch(opcode_ab) {
      ST          :{
next(pc_ab) := nextpc.f_nextpc[pc_ab];
next(ram_mem_ab[alu.f_alu_sout[aluop_ab][ain2_ab][imm_ab][c_ab]])
                                    := rt_ab;
                        nstate_ab := ST_FETCH0;
                  }
      LPC         :{
next(pc_ab) :=
      ram_mem_ab[alu.f_alu_sout[aluop_ab][rsp_ab][rt_ab][c_ab]];
                        nstate_ab := ST_FETCH0;
                  }
      SPC         :{
next(pc_ab) := nextpc.f_nextpc[pc_ab];
next(ram_mem_ab[alu.f_alu_sout[aluop_ab][rsp_ab][rt_ab][c_ab]])
```

```
                                        := nextpc.f_nextpc[pc_ab];

                        nstate_ab := ST_FETCH0;
                }

        default     :     nstate_ab := ST_WRBACK;

              } /* end switch(opcode_ab) */

        } /* end ST_MEMACC */

/*------------------------------*/

ST_WRBACK : {

        /* carry flag */
        switch(opcode_ab) {

        {ADD, SUB}  :
        next(c_ab) := alu.f_alu_cout[aluop_ab][ain_ab][bin_ab][c_ab];

        {ROL, ROR}  :
        next(c_ab) := alu.f_alu_cout[aluop_ab][ain_ab][rt_ab][c_ab];

        }

        /* zero flag */
        switch(opcode_ab) {

        {ADD, SUB, AND, ORR, XOR}     :
        next(z_ab) := alu.f_alu_zout[aluop_ab][ain_ab][bin_ab][c_ab];

        {ROL, ROR}          :
        next(z_ab) := alu.f_alu_zout[aluop_ab][ain_ab][rt_ab][c_ab];

        LUI                 :
        next(z_ab) := alu.f_alu_zout[aluop_ab][ain_ab][imm_ab][c_ab];

        }

        /* register file */
        switch(opcode_ab) {

        {ADD, SUB, AND, ORR, XOR}    :{

        sout_ab := alu.f_alu_sout[aluop_ab][ain_ab][bin_ab][c_ab];

                        switch(rs_ab) {
                        RA    : next(ra_ab)     := sout_ab;
                        RB    : next(rb_ab)     := sout_ab;
                        RSP   : next(rsp_ab)    := sout_ab;
                        RT    : next(rt_ab)     := sout_ab;
                        }
                }

        {COM, ROL, ROR}   :{

        sout_ab := alu.f_alu_sout[aluop_ab][ain_ab][rt_ab][c_ab];

                        switch(rs_ab) {
                        RA    : next(ra_ab)     := sout_ab;
```

```
                        RB    : next(rb_ab)     := sout_ab;
                        RSP   : next(rsp_ab)    := sout_ab;
                        RT    : next(rt_ab)     := sout_ab;
                        }
                }

        LUI         :{

        sout_ab := alu.f_alu_sout[aluop_ab][ain_ab][imm_ab][c_ab];

                        switch(rs_ab) {
                        RA    : next(ra_ab)     := sout_ab;
                        RB    : next(rb_ab)     := sout_ab;
                        RSP   : next(rsp_ab)    := sout_ab;
                        RT    : next(rt_ab)     := sout_ab;
                        }
                }

        R2T         :{

        sout_ab := alu.f_alu_sout[aluop_ab][ain_ab][rt_ab][c_ab];

                        next(rt_ab) := sout_ab;
                }
        LD          :{

sout_ab := ram_mem_ab[alu.f_alu_sout[aluop_ab][ain2_ab][imm_ab]
                                                              [c_ab]];

                        switch(rs_ab) {
                        RA    : next(ra_ab)     := sout_ab;
                        RB    : next(rb_ab)     := sout_ab;
                        RSP   : next(rsp_ab)    := sout_ab;
                        RT    : next(rt_ab)     := sout_ab;
                        }
                }
              } /* end switch(opcode_ab) */

        next(pc_ab) := nextpc.f_nextpc[pc_ab];

        nstate_ab := ST_FETCH0;

        } /* end ST_WRBACK*/

/*------------------------------*/

default : {
              nstate_ab := ST_RESET;
              }

} /* end switch(pstate_ab) */

/*--------------- Implementation ---------------*/

/*---------- datapath ----------*/

/* signal in data path */
pc_out              : WORD;
ir_out              : WORD;
pc_out_jmp          : WORD;
pc_out_1            : WORD;
```

```
next_pc          : WORD;
rom_data         : WORD;
a_out            : WORD;
b_out            : WORD;
s_out            : WORD;
alu_b            : WORD;
next_pc_1        : WORD;
d_data_in        : WORD;
d_data_out       : WORD;
dat_in           : WORD;
d_data_in2       : WORD;
d_data_out2      : WORD;

/* output of state machine */
prog_en          : boolean;
pc_ld            : boolean;
ir_ld            : boolean;
reg_ld           : boolean;
mem_en           : boolean;

/* output of decode circuit */
op_sel           : boolean;
a_sel            : REG;
ld_sel           : REG;
pc_sel           : boolean;
pc_sel1          : boolean;
mem_sel1         : boolean;
mem_sel          : boolean;
mem_nwr          : boolean;
tri_d_data       : boolean;

/* flag signal */
c_out            : FLAG;
z_out            : FLAG;
c                : FLAG;
z                : FLAG;
c_in             : FLAG;

/* signal from ir_out*/
opcode           : OPCODE;
imm              : WORD;
inst_mode        : boolean;
rs               : REG;
b_offset         : WORD;
alu_op           : ALUOP;

/* state variable */
pstate           : STATE;
nstate           : STATE;


/*------------------------------*/

c_in  := c;

opcode           := f_opcode[ir_out];
imm              := f_imm[ir_out];
inst_mode        := f_instmode[ir_out];
rs               := f_rs[ir_out];
b_offset         := f_boffset[ir_out];
alu_op           := f_aluop[ir_out];
```

```
/*------------------------------*/

/* component of datapath */
pc          : pc(pc_out, next_pc_1, pc_ld);
ir          : ir(ir_out, rom_data, ir_ld);
nextpc      : nextpc(pc_out_1, pc_out);
adder       : adder(pc_out_jmp, pc_out_1, b_offset);
pc_mux      : mux21(next_pc, pc_out_1, pc_out_jmp, pc_sel);
pc_mux1     : mux21(next_pc_1, next_pc, d_data_in, pc_sel1);

regfile     : regfile(a_out, b_out, dat_in, a_sel, ld_sel, reg_ld);
alu         : alu(s_out, z_out, c_out, a_out, alu_b, c_in, alu_op);
mux_alu     : mux21(alu_b, b_out, imm, op_sel);
mem_out_mux : mux21(d_data_out, b_out, next_pc_1, mem_sel1);
mem_mux     : mux21(dat_in, d_data_in, s_out, mem_sel);

rom         : rom(rom_data, pc_out, prog_en);
ram         : ram(d_data_in2, d_data_out2, s_out, mem_en, mem_nwr);

/*------------------------------*/

if (tri_d_data = 1)          d_data_in := d_data_in2;

if (tri_d_data = 0)          d_data_out2 := d_data_out;

/*---------- control unit ----------*/

switch(opcode) {
      {LD, ST}                    : op_sel := 1;
      LUI                         : op_sel := 1;
      {ADD, SUB, AND, ORR, XOR}   : op_sel := inst_mode;
      default                     : op_sel := 0;
};

switch(opcode) {
      {ST, LD}    : if (inst_mode = 1)    a_sel := RB;
                    else                  a_sel := RSP;
      {LPC, SPC}  : a_sel := RSP;
      default     : a_sel := rs;
};

switch(opcode) {
      R2T         : ld_sel := RT;
      Default     : ld_sel := rs;
};

switch(opcode) {
      JMP         : pc_sel := 1;
      JNZ         : if (z = 1) pc_sel := 1; else pc_sel := 0;
      JNC         : if (c = 1) pc_sel := 1; else pc_sel := 0;
      Default     : pc_sel := 0;
};

switch(opcode) {
      {ST, SPC}   : mem_nwr := 0;
      default     : mem_nwr := 1;
};

switch(opcode) {
      {ST, SPC}   : { mem_sel := 0; tri_d_data := 0; }
```

```
      LD          : { mem_sel := 0; tri_d_data := 1; }
      default     : { mem_sel := 1; tri_d_data := 1; }
};

switch(opcode) {
      LPC         : { mem_sel1 := 1; pc_sel1 := 1; }
      SPC         : { mem_sel1 := 1; pc_sel1 := 0; }
      default     : { mem_sel1 := 0; pc_sel1 := 0; }
};

/*------------------------------*/

next(pstate) := nstate;

default{
            next(c) := c;
            next(z) := z;
}

in    switch(pstate) {

ST_RESET : {
                  prog_en := 1; pc_ld := 0; ir_ld := 0;
                  reg_ld := 0; mem_en := 1;

                  nstate := ST_FETCH0;
            }

ST_FETCH0 : {
                  prog_en := 0; pc_ld := 0; ir_ld := 1;
                  reg_ld := 0; mem_en := 1;

                  nstate := ST_DECODE;
            }

ST_DECODE : {
                  /* Wait for all the decode circuit */

                  prog_en := 1; pc_ld := 0; ir_ld := 0;
                  reg_ld := 0; mem_en := 1;

                  nstate := ST_EXEC;
            }

ST_EXEC : {

            /* Wait for completion of ALU / complete Branch / etc. */

            prog_en := 1; ir_ld := 0;
            reg_ld := 0; mem_en := 1;

            /* carry flag */
            switch(opcode) {
            CLC         : next(c) := 0;
            STC         : next(c) := 1;
            default     : next(c) := c;
            }

            switch(opcode) {

            {CLC, STC}              :{
```

```
                                    pc_ld := 1;
                                    nstate := ST_FETCH0;
                              }

            {NOP, JMP, JNC, JNZ}    :{
                                    pc_ld := 1;
                                    nstate := ST_FETCH0;
                              }

            {LD, ST, LPC, SPC}      :{
                                    pc_ld := 0;
                                    nstate := ST_MEMACC;
                              }

            default                 :{
                                    pc_ld := 0;
                                    nstate := ST_WRBACK;
                              }
                  } /* end switch(opcode) */

            } /* end ST_EXEC */

ST_MEMACC : {

            prog_en := 1; ir_ld := 0;
            reg_ld := 0; mem_en := 0;

            switch(opcode) {

            {ST, SPC, LPC}    :{
                                    pc_ld := 1;
                                    nstate := ST_FETCH0;
                              }

            default           :{
                                    pc_ld := 0;
                                    nstate := ST_WRBACK;
                              }

                  } /* end switch(opcode) */

            } /* end ST_MEMACC */

ST_WRBACK : {

            prog_en := 1; pc_ld := 1; ir_ld := 0;
            reg_ld := 1;

            switch(opcode) {
            LD          : mem_en := 0;
            default     : mem_en := 1;
            }

            /* carry flag */
            switch(opcode) {
            {ADD, SUB, ROL, ROR}    : next(c) := c_out;
            default                 : next(c) := c;
            }

            /* zero flag */
            switch(opcode) {
```

```
            {ADD, SUB, AND, ORR, XOR, ROL, ROR, LUI}
                                                : next(z) := z_out;
            default                             : next(z) := z;
            }

            nstate := ST_FETCH0;

            } /* end ST_WRBACK */

default : {
                  prog_en := 1; pc_ld := 0; ir_ld := 0;
                  reg_ld := 0; mem_en := 1;

                  nstate := ST_RESET;
            }

} /* end switch(pstate) */

/*--------------- Signal Initialization ---------------*/

init(ir.ir_out)   := ir_ab;
init(pc.pc_out)   := pc_ab;
init(rom.mem)     := rom_mem_ab;
init(ram.mem)     := ram_mem_ab;

init(c)           := c_ab;
init(z)           := z_ab;
init(regfile.ra)  := ra_ab;
init(regfile.rb)  := rb_ab;
init(regfile.rsp) := rsp_ab;
init(regfile.rt)  := rt_ab;

init(pstate)      := ST_RESET;
init(pstate_ab)   := ST_RESET;

forall(i in WORD)
      init(f_opcode[i]) := {NOP, ADD, SUB, AND, ORR,
                  XOR, COM, ROL, ROR, LUI, R2T, CLC, STC,
                  JMP, JNC, JNZ,
                  LD, ST, LPC, SPC};

/*--------------- Refinement Relation ---------------*/

layer lemma: {
      ir_out := ir_ab;

      opcode := opcode_ab;

      regfile.ra  := ra_ab;
      regfile.rb  := rb_ab;
      regfile.rsp := rsp_ab;
      regfile.rt  := rt_ab;

      c := c_ab;
      z := z_ab;

      pc_out := pc_ab;

      pstate := pstate_ab;

      ram.mem := ram_mem_ab;
```

```
      }

} /*end of module main*/

/*-------------------- Other Module Definition --------------------*/

/*-------------------------------------------*/
/* rewrite in SMV lang. from "pc.v" */

module pc(pc_out, pc_in, ld) {

output      pc_out      : WORD;
input       pc_in       : WORD;
input       ld          : boolean;

next(pc_out) :=
      case {
      (ld = 1)    : pc_in;
      default     : pc_out;
      };

}

/*-------------------------------------------*/
/* rewrite in SMV lang. from "ir.v" */

module ir(ir_out, ir_in, ld) {

output      ir_out      : WORD;
input       ir_in       : WORD;
input       ld          : boolean;

next(ir_out) :=
      case {
      (ld = 1)    : ir_in;
      default     : ir_out;
      };

}

/*-------------------------------------------*/
/* rewrite in SMV lang. from "regfile.v" */

module regfile(a_out, b_out, dat_in, a_sel, ld_sel, ld) {

output      a_out       : WORD;
output      b_out       : WORD;
input       dat_in      : WORD;
input       a_sel       : REG;
input       ld_sel      : REG;
input       ld          : boolean;

ra    : WORD;
rb    : WORD;
rsp   : WORD;
rt    : WORD;

b_out := rt;

a_out :=    switch(a_sel) {
            RA    : ra;
```

```
            RB    : rb;
            RSP   : rsp;
            RT    : rt;
            };

next(ra) :=
      case {
      (ld=1)&(ld_sel=RA)      : dat_in;
      default                 : ra;
      };

next(rb) :=
      case {
      (ld=1)&(ld_sel=RB)      : dat_in;
      default                 : rb;
      };

next(rsp) :=
      case {
      (ld=1)&(ld_sel=RSP)     : dat_in;
      default                 : rsp;
      };

next(rt) :=
      case {
      (ld=1)&(ld_sel=RT)      : dat_in;
      default                 : rt;
      };

}

/*-------------------------------------------*/
/* rewrite in SMV lang. from "alu.v" */

module alu(s_out, z_out, c_out, a_in, b_in, c_in, alu_op) {

output      s_out       : WORD;
output      z_out       : FLAG;
output      c_out       : FLAG;
input       a_in        : WORD;
input       b_in        : WORD;
input       c_in        : FLAG;
input       alu_op      : ALUOP;

/*----- an uninterpreted function -----*/
f_alu_sout : array ALUOP of array WORD of array WORD of array FLAG of
                                                              WORD;
f_alu_zout : array ALUOP of array WORD of array WORD of array FLAG of
                                                              FLAG;
f_alu_cout : array ALUOP of array WORD of array WORD of array FLAG of
                                                              FLAG;

init(f_alu_sout[0][0][0][0]) := 0;
init(f_alu_sout[0][0][0][1]) := 1;
init(f_alu_sout[0][0][1][0]) := 1;
init(f_alu_sout[0][0][1][1]) := 0;
init(f_alu_sout[0][1][0][0]) := 1;
init(f_alu_sout[0][1][0][1]) := 0;
init(f_alu_sout[0][1][1][0]) := 0;
init(f_alu_sout[0][1][1][1]) := 1;
init(f_alu_sout[1][0][0][0]) := 0;
```

```
init(f_alu_sout[1][0][0][1]) := 0;
init(f_alu_sout[1][0][1][0]) := 0;
init(f_alu_sout[1][0][1][1]) := 0;
init(f_alu_sout[1][1][0][0]) := 0;
init(f_alu_sout[1][1][0][1]) := 0;
init(f_alu_sout[1][1][1][0]) := 1;
init(f_alu_sout[1][1][1][1]) := 1;

init(f_alu_zout[0][0][0][0]) := 1;
init(f_alu_zout[0][0][0][1]) := 0;
init(f_alu_zout[0][0][1][0]) := 0;
init(f_alu_zout[0][0][1][1]) := 1;
init(f_alu_zout[0][1][0][0]) := 0;
init(f_alu_zout[0][1][0][1]) := 1;
init(f_alu_zout[0][1][1][0]) := 1;
init(f_alu_zout[0][1][1][1]) := 0;
init(f_alu_zout[1][0][0][0]) := 1;
init(f_alu_zout[1][0][0][1]) := 1;
init(f_alu_zout[1][0][1][0]) := 1;
init(f_alu_zout[1][0][1][1]) := 1;
init(f_alu_zout[1][1][0][0]) := 1;
init(f_alu_zout[1][1][0][1]) := 1;
init(f_alu_zout[1][1][1][0]) := 0;
init(f_alu_zout[1][1][1][1]) := 0;

init(f_alu_cout[0][0][0][0]) := 0;
init(f_alu_cout[0][0][0][1]) := 0;
init(f_alu_cout[0][0][1][0]) := 0;
init(f_alu_cout[0][0][1][1]) := 1;
init(f_alu_cout[0][1][0][0]) := 0;
init(f_alu_cout[0][1][0][1]) := 1;
init(f_alu_cout[0][1][1][0]) := 1;
init(f_alu_cout[0][1][1][1]) := 1;
init(f_alu_cout[1][0][0][0]) := 0;
init(f_alu_cout[1][0][0][1]) := 0;
init(f_alu_cout[1][0][1][0]) := 0;
init(f_alu_cout[1][0][1][1]) := 0;
init(f_alu_cout[1][1][0][0]) := 0;
init(f_alu_cout[1][1][0][1]) := 0;
init(f_alu_cout[1][1][1][0]) := 0;
init(f_alu_cout[1][1][1][1]) := 0;

next(f_alu_sout) := f_alu_sout;
next(f_alu_zout) := f_alu_zout;
next(f_alu_cout) := f_alu_cout;

/*----- body of module -----*/
s_out := f_alu_sout[alu_op][a_in][b_in][c_in];
z_out := f_alu_zout[alu_op][a_in][b_in][c_in];
c_out := f_alu_cout[alu_op][a_in][b_in][c_in];

}

/*-----------------------------------------------*/
/* rewrite in SMV lang. from "adder.v" */

module adder(s_out, a_in, b_in) {

output      s_out : WORD;
input       a_in  : WORD;
input       b_in  : WORD;
```

```
/*----- an uninterpreted function -----*/

      f_adder : array WORD of array WORD of WORD;

      next(f_adder) := f_adder;

/*----- body of module -----*/

      s_out := f_adder[a_in][b_in];

}

/*-------------------------------------------*/
/* rewrite in SMV lang. from "adder.v" */

module nextpc(s_out, a_in) {

output      s_out : WORD;
input       a_in  : WORD;

/*----- an uninterpreted function -----*/

      f_nextpc    : array WORD of WORD;

      next(f_nextpc) := f_nextpc;

/*----- body of module -----*/

      s_out := f_nextpc[a_in];

}

/*-------------------------------------------*/
/* rewrite in SMV lang. from "mux2_1.v" */

module mux21(s_out, a_in, b_in, sel) {

output      s_out : WORD;
input       a_in  : WORD;
input       b_in  : WORD;
input       sel   : boolean;

s_out :=    switch(sel) {
            0     : a_in;
            1     : b_in;
            };

}

/*-------------------------------------------*/
/* rewrite in SMV lang. from "rom.v" */

module rom(data, address, ncs) {

output      data        : WORD;
input       address     : WORD;
input       ncs         : boolean;

mem   : array WORD of WORD;
```

```
next(mem) := mem;

if (ncs = 0) {
      data := mem[address];
      }

}

/*-------------------------------------------*/
/* rewrite in SMV lang. from "ram.v" */

module ram(data_out, data_in, address, ncs, nwr) {

output      data_out    : WORD;
input       data_in     : WORD;
input       address     : WORD;
input       ncs         : boolean;
input       nwr         : boolean;

mem   : array WORD of WORD;

if ((ncs=0)&(nwr=1)) {
      data_out := mem[address];
      }

default
      forall(i in WORD)
            next(mem[i]) := mem[i];
in
      if ((ncs=0)&(nwr=0)) {
            next(mem[address]) := data_in;
      }

}
```

ประวัติผู้เขียนวิทยานิพนธ์

นาย ประพนธ์ บวรภราดร เกิดเมื่อวันที่ 6 กุมภาพันธ์ 2523 ที่โรงพยาบาลเยาวรักษ์ เขตธนบุรี กรุงเทพมหานคร จบการศึกษาวิศวกรรมศาสตร์บัณฑิต (เกียรตินิยมอันดับหนึ่ง) สาขาวิศวกรรมคอมพิวเตอร์ จากจุฬาลงกรณ์มหาวิทยาลัยในปีการศึกษา 2543 และเข้าศึกษาต่อโดยได้รับทุนอุดหนุนการศึกษาในระดับบัณฑิตศึกษาของจุฬาลงกรณ์มหาวิทยาลัยเพื่อเฉลิมฉลองในวโรกาสที่พระบาทสมเด็จพระเจ้าอยู่หัวทรงเจริญพระชนมายุครบ 72 พรรษา