# ปรับปรุงประสิทธิภาพการเข้ารหัสแบบอาร์เอสเอนเอาร์ม โปรเซสเซอร์โดยการปรับปรุงโปรแกรมต้นฉบับ

# PERFOMANCE OPTIMIZATION RSA ENCRYPTION ON ARM PROCESSOR BY MODIFYING SOURCE PROGRAM


พิชชา เตียวิริยะกุล

PITCHA TYOVIRIYAKUL


วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต

สาขาวิชาวิศวกรรมคอมพิวเตอร์


คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง



พ.ศ. 2551

KMITL-2008-EN-M-070-184

# ปรับปรุงประสิทธิภาพการเข้ารหัสแบบอาร์เอสเอบนอาร์ม โปรเซสเซอร์โดยการปรับปรุงโปรแกรมต้นฉบับ

# PERFORMANCE OPTIMIZATION RSA ENCRYPTION ON ARM PROCESSOR BY MODIFYING SOURCE PROGRAM


พิชชา เตียววิริยะกุล

PITCHA TYOVIRIYAKUL


วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต

สาขาวิชาวิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

พ.ศ. 2551


KMITL-2008-EN-M-070-184

# PERFORMANCE OPTIMIZATION RSA ENCRYPTION ON ARM PROCESSOR BY MODIFYING SOURCE PROGRAM

PITCHA TYOVIRIYAKUL

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENT FOR THE DEGREE OF

MASTER OF ENGINEERING IN COMPUTER ENGINEERING

FACULTY OF ENGINEERING

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

2008

KMITL-2008-EN-M-070-184

**คณะวิศวกรรมศาสตร์**

**สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง**

**ใบรับรองวิทยานิพนธ์**

---

| | |
|---|---|
| **หัวข้อวิทยานิพนธ์** | ปรับปรุงประสิทธิภาพการเข้ารหัสแบบอาร์เอสเอนอาร์มโปรเซสเซอร์โดยการปรับปรุงโปรแกรมต้นฉบับ |
| **Thesis Title** | Performance Optimization RSA Encryption on ARM Processor by Modifying Source Program |
| **นักศึกษา** | นายพิชชา เตียววิริยะกุล |
| **รหัสประจำตัว** | 48060720 |
| **ปริญญา** | วิศวกรรมศาสตรมหาบัณฑิต |
| **สาขาวิชา** | วิศวกรรมคอมพิวเตอร์ |
| **อาจารย์ที่ปรึกษาวิทยานิพนธ์** | ผศ.ดร.สุรินทร์ กิตติธรกุล |
| **หมายเลขวิทยานิพนธ์** | KMITL-2008-EN-M-070-184 |

| คณะกรรมการสอบวิทยานิพนธ์ | | ลายมือชื่อ |
|---|---|---|
| ผศ.อภิเนตร | อูนากูล | |
| ผศ.ธนา | หงษ์สุวรรณ | |
| ผศ.ดร.เผ่าภัค | ศิริสุข | |
| ผศ.เกียรติกูล | เจียรนัยธนะกิจ | |
| ผศ.ดร.สุรินทร์ | กิตติธรกุล | |

**วัน / เดือน/ ปี ที่สอบ** วันอังคารที่ 23 กันยายน พ.ศ. 2551 เวลา 10.00-12.00 น.

**สถานที่สอบ** ณ อาคาร A ชั้น 5 ห้องประชุม 5



**คณะวิศวกรรมศาสตร์ รับรองแล้ว**

**(รองศาสตราจารย์ ดร.กอบชัย เดชหาญ)**

**คณบดี คณะวิศวกรรมศาสตร์**

| | |
|---|---|
| **หัวข้อวิทยานิพนธ์** | ปรับปรุงประสิทธิภาพการเข้ารหัสแบบอาร์เอสเอบน อาร์มโปรเซสเซอร์โดยการปรับปรุงโปรแกรมต้นฉบับ |
| **นักศึกษา** | นายพิชชา เตียววิริยะกุล |
| **รหัสประจำตัว** | 48060720 |
| **ปริญญา** | วิศวกรรมศาสตรมหาบัณฑิต |
| **สาขาวิชา** | วิศวกรรมคอมพิวเตอร์ |
| **พ.ศ.** | 2551 |
| **อาจารย์ผู้ควบคุมวิทยานิพนธ์** | ผศ.ดร สุรินทร์ กิตติธรกุล |


**บทคัดย่อ**

คอมไพเลอร์ ส่วนใหญ่ จะเน้น การ ปรับปรุง ทางด้านความเร็ว ในวิทยานิพนธ์ จะแสดงการปรับปรุงโค้ดการเข้ารหัส แบบอาร์เอสเอ โดยเน้นการลดการเกิดการหาข้อมูลไม่พบในหน่วยความจำแคช ในการใช้หน่วยความจำ ในการเข้ารหัสแบบอาร์เอสเอ บน อาร์มโปรเซสเซอร์ ซึ่งเป็น โปรเซสเซอร์ที่ ใช้งานกันอย่างแพร่หลาย ทั้งใน โทรศัพท์มือถือ หรือ คอมพิวเตอร์แบบพกพา และ อุปกรณ์อื่นๆ โดยในการทดลอง ในวิทยานิพนธ์ จะทำการเข้ารหัสแบบอาร์เอสเอ บนอาร์ม โปรเซสเซอร์ด้วยความยาว รหัสขนาด 1024 บิต โดยแสดงให้เห็น ว่า กระบวนการปรับปรุงโค้ดการเข้ารหัส แบบอาร์เอสเอ ที่ได้นำเสนอสามารถเพิ่มความเร็วในการทำงาน ลดจำนวนการเข้าถึงหน่วยความจำหลัก และทำให้ช่วยลดการใช้พลังงาน ในทุกระดับการปรับปรุงประสิทธิภาพของ ARM920T C++ คอมไพเลอร์บนพื้นฐานที่มีแคชของชุดคำสั่ง 16 กิโลบิตและแคชของข้อมูล 16 กิโลบิต ในARM920T โดยสามารถเพิ่มด้านความเร็ว เพิ่มขึ้นจากเดิม 7.19-11.72 % และลดการใช้พลังงานลง 6.25%

| | |
|---|---|
| **Thesis Title** | Performance optimization of RSA encryption on ARM processor by modifying source program |
| **Student** | Mr.Pitcha Tyoviriyakul |
| **Student ID** | 48060720 |
| **Degree** | Master of Engineering |
| **Programme** | Computer Engineering |
| **Year** | 2008 |
| **Thesis Advisor** | Asst.Prof.Dr Surin Kittitornkun |


## ABSTRACT

Most compiler optimization techniques concern most about speed. In this thesis, we present two high-level memory optimization methods and four high-level speed optimization methods for ARM-based secure applications on mobile phones, pocket PCs, etc. The experiments using RSA encryption on ARM920T with 1024-bit random public keys show that the proposed techniques can complement the existing speed-oriented ones to achieve less number of memory accesses, shorter execution time, and lower memory allocations to all ARM C++ optimization levels despite the 16-KB instruction and 16-KB data caches of ARM 920T core.

# กิตติกรรมประกาศ

คุณความดีอันใดที่บังเกิดจากวิทยานิพนธ์ฉบับนี้ ขอมอบแด่บิดาและมารดาของผู้วิจัย ผู้ที่คอยห่วงใย เข้าใจและให้การสนับสนุนในการศึกษามาโดยตลอด

วิทยานิพนธ์ฉบับนี้สำเร็จลุล่วงลงได้ด้วยดี โดยได้รับความกรุณาจาก ผศ.ดร สุรินทร์ กิตติธรกุล อาจารย์ที่ปรึกษา ที่ได้ช่วยเหลือในการให้คำแนะนำ ความรู้ทางทฤษฎีต่างๆที่ใช้ และชี้แนะแนวทางในการแก้ปัญหาต่างๆอย่างทุ่มเทรวมทั้งฝึกฝนผู้วิจัยให้มีความสามารถในการทำวิจัยและพัฒนาได้อย่างมีประสิทธิภาพ ผู้วิจัยรู้สึกซาบซึ้งในความอนุเคราะห์จากท่านและขอกราบขอบพระคุณเป็นอย่างสูง

ขอขอบคุณกรรมการสอบวิทยานิพนธ์ทุกท่านที่ได้กรุณาให้คำแนะนำในทุกๆเรื่อง ทั้งวิธีแก้ปัญหาที่เกิดขึ้นในการทำวิทยานิพนธ์และมุมมองในเชิงวิศวกรรมอื่นๆซึ่งช่วยให้ผู้วิจัยมีวิสัยทัศน์ที่กว้างไกลขึ้น

ขอขอบคุณบัณฑิตวิทยาลัย สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง ที่ให้การสนับสนุนการทำวิทยานิพนธ์นี้

ขอบคุณพี่ๆเพื่อนๆและน้องๆนักศึกษาทุกคนในห้องวิจัย รวมทั้งเพื่อนๆหลายๆคนในอินเตอร์เน็ตที่ช่วยเหลือให้คำแนะนำต่างๆและให้กำลังใจแก่ผู้วิจัยตลอดมา

พิชชา เตียววิริยะกุล

# สารบัญ

หน้า

# สารบัญ (ต่อ)

หน้า

# สารบัญ (ต่อ)

หน้า

# สารบัญ (ต่อ)

หน้า

# สารบัญตาราง

# สารบัญรูป

# สารบัญรูป

รูปที่ หน้า

# บทที่ 1
# บทนำ

## 1.1 ความเป็นมาและความสำคัญของปัญหา

ในปัจจุบันจากที่สมัยก่อนเราใช้โทรศัพท์เพียงพูดคุยส่งข้อความ จนกระทั่งเทคโนโลยีต่างๆได้พัฒนาขึ้นจนปัจจุบันโทรศัพท์เคลื่อนที่ได้เข้ามามีบทบาทในชีวิตประจำวันมากขึ้น ดังจะเห็นได้จากวิวัฒนาการของโทรศัพท์เคลื่อน จากอดีตที่มีการทำงานด้วยระบบอนาล็อก (Analog) มาสู่ยุคปัจจุบันที่ทำงานด้วยระบบดิจิตอล (Digital) หรือยุคของ 2จี, 2.5จี และกำลังจะก้าวไปสู่ยุค 3จี ในเวลาอันใกล้นี้ จึงทำให้โทรศัพท์เคลื่อนที่ไม่ได้เป็นแค่เพียงอุปกรณ์ที่ใช้สำหรับการสื่อสารด้วยเสียงเพียงอย่างเดียวเท่านั้น แต่ยังมีการส่งข้อมูลต่างๆเพิ่มเข้าไปอีกด้วย ช่วยให้สามารถสื่อสารค้นหาข้อมูลที่ต้องการได้อย่างสะดวกรวดเร็ว ทั้งบนโทรศัพท์มือถือ, พีดีเอ ซึ่งทำให้ข้อมูลส่วนตัว ต่างๆมีความสำคัญมากยิ่งขึ้น ปัญหาเรื่องความปลอดภัยของข้อมูลในการสื่อสาร จนรวมไปถึงข้อมูลส่วนตัวได้เข้ามาเป็นปัญหาสำคัญ ดังนั้นการเข้ารหัสข้อมูลจึงเป็นสิ่งจำเป็นมากขึ้นเรื่อยๆ แต่อย่างไรก็ตามการเข้ารหัสข้อมูลนั้น โปรเซสเซอร์ยังต้องคำนวณมากและใช้พลังงานจากแบตเตอรี่มาก RISC โปรเซสเซอร์ในโมบายดีไวซ์ที่ได้รับความนิยมมากคืออาร์ม เนื่องด้วยกินพลังงานน้อยและขนาดเล็ก โดยเฉพาะอย่างยิ่งในมือถือที่มีพลังงานในการคำนวณที่จำกัดและมีผลกระทบโดยตรงกับเวลาในการใช้งานแบตเตอรี่

ดังนั้นจึงมีความจำเป็นที่จะต้องปรับปรุงประสิทธิภาพ(Optimize) ทางด้านพลังงานและความเร็วของการเข้ารหัส โดยในงานวิทยานิพนธ์ชิ้นนี้จะใช้การเข้ารหัสแบบอาร์เอสเอ ในงานวิทยานิพนธ์ โดยที่จะมุ่งเน้นการปรับปรุงประสิทธิภาพเพื่อลดการใช้พลังงานและเพิ่มความเร็วสำหรับการเข้ารหัสแบบอาร์เอสเอ ที่ทำงานบนอาร์มโปรเซสเซอร์

การเข้ารหัสแบบต่างๆมีมานานและมีการให้โค้ดโปรแกรมการเข้ารหัสอยู่มากมาย แต่ได้เลือกอาร์เอสเอ เนื่องจากได้รับความนิยมแพร่หลายจากนั้นได่ทำการแปลงโค้ดการเข้ารหัส แบบอาร์เอสเอนี้ไปทำงานบนมือถือ Nokia 6600 ที่ใช้ ระบบปฏิบัติการซิมเบียน(Symbian) ที่ใช้อาร์มโปรเซสเซอร์เป็นหน่วยประมวลผล และจะใช้โค้ดที่ปรับปรุงนี้ทดลองในงานวิจัย

## 1.2 วัตถุประสงค์ของการศึกษา

1. ศึกษาหลักการของ การเข้ารหัสแบบอาร์เอสเอ
2. ศึกษา การเข้ารหัสแบบอาร์เอสเอ ที่นำมาทำให้ใช้งานได้บนมือถือ
3. ศึกษาหลักการทำงานของ อาร์มโปรเซสเซอร์ซีรี่ที่ใช้ในมือถือ
4. ลดการใช้พลังงานและเพิ่มความเร็ว การเข้ารหัสแบบอาร์เอสเอ โดยนำมาทำให้ใช้งานบนมือถือ
5. เปรียบเทียบผลการลดการใช้พลังงานและเพิ่มความเร็วทั้งก่อนและหลังการปรับปรุง

## 1.3 สมมุติฐานของการศึกษา

เพื่อให้บรรลุวัตถุประสงค์ของการศึกษา ได้นำเสนอหลักการตามทฤษฎีที่เกี่ยวข้องเพื่อทำให้สามารถลดการใช้พลังงานและเพิ่มความเร็วในการทำงานของการเข้ารหัสแบบอาร์เอสเอ และเพิ่มความเร็วด้วยวิธีเช่นการลดการเรียกใช้ฟังก์ชัน การลดการจองหน่วยความจำ ซึ่งเป็นการลดจำนวนการใช้หน่วยความจำลง ซึ่งจะส่งผลทำให้การค้นหาข้อมูลหน่วยความจำหลักลดลงด้วย โดยในแอพพลิเคชันต่างๆจะใช้พลังงานส่วนใหญ่ไปกับการใช้งานหน่วยความจำ ดังนั้นด้วยวิธีการนี้ก็เท่ากับลดการพลังงานด้วยนั้นเอง

## 1.4 ขอบเขตของงานวิจัย

วิทยานิพนธ์ชิ้นนี้ได้ศึกษาการเข้ารหัสแบบอาร์เอสเอ แล้วไปทำงานบนมือถือที่ใช้อาร์มโปรเซสเซอร์ และศึกษาหลักการทำงานของอาร์มโปรเซสเซอร์ในรุ่นที่ใช้งานบนมือถือเช่น ARM920T, ARM922T เป็นต้น แล้วปรับปรุงประสิทธิภาพด้านการใช้พลังงานและเพิ่มความเร็วในการทำงานโดยมุ่งเน้นการลดการเข้าใช้หน่วยความจำหลัก โดยใช้วิธีการที่สอดคล้องกับการทำงานของอาร์มโปรเซสเซอร์ จากนั้นทำการวัดประสิทธิภาพเปรียบเทียบผลก่อนและหลังการปรับปรุง

## 1.5 ขั้นตอนการศึกษา

1. ศึกษาหลักการของการเข้ารหัสแบบอาร์เอสเอ
2. ศึกษาหลักการของสถาปัตยกรรมอาร์มโปรเซสเซอร์ เนื่องจากอาร์มโปรเซสเซอร์ได้รับความนิยมอย่างกว้างขวางใช้ในอุปกรณ์มือถือ
3. ศึกษางานวิจัยที่คล้ายคลึงหรือสอดคล้องกับงานวิจัยนี้
4. ทำการทดลองการปรับปรุงประสิทธิภาพของเราโดยเปรียบเทียบก่อนและหลังการทดลองโดยใช้ ARM Developer Suite ในการจำลองการทำงานของ ARM920T โปรเซสเซอร์ ที่ทำงานเข้ารหัสแบบอาร์เอสเอ เพื่อวัดประสิทธิภาพทางด้านความเร็ว และ นำไปวัดการใช้พลังงานบนโทรศัพท์จริงเพื่อวัดประสิทธิภาพทางด้านพลังงาน
5. สรุปและวิเคราะห์ผลการทดลอง

## 1.6 ข้อตกลงเบื้องต้น

งานวิจัยนี้คือ การปรับปรุงประสิทธิภาพของ การเข้ารหัสแบบอาร์เอสเอ บนอาร์มโปรเซสเซอร์ และต้องยอมรับก่อนว่าโปรเซสเซอร์ส่วนมากที่ใช้ในอุปกรณ์โทรศัพท์มือถือจะใช้อาร์มโปรเซสเซอร์ ซึ่งวิธีการของเราจะเป็นการปรับปรุงประสิทธิภาพบนพื้นฐานสถาปัตยกรรมอาร์ม

## 1.7 ข้อจำกัดของการศึกษา

ในการจะวัดประสิทธิภาพของการเข้ารหัสแบบอาร์เอสเอ บนอุปกรณ์มือถือ ทั้งในด้านความเร็วและด้านการใช้พลังงานทำได้ยาก ดังนั้นจะใช้โปรแกรม ที่ชื่อว่า ARM developer suite จำลองการทำงานของอาร์มโปรเซสเซอร์ แล้วจึงวัดจำนวนรอบสัญญาณนาฬิกาที่ใช้ในการทำงานเพื่อวัดประสิทธิภาพทางด้านความเร็วและจากนั้นจึงนำไปทดสอบ บนโทรศัพท์มือถือจริงที่ใช้อาร์มโปรเซสเซอร์ เพื่อวัดประสิทธิภาพทางด้านการใช้พลังงาน

## 1.8 รายละเอียดของวิทยานิพนธ์

วิทยานิพนธ์ฉบับนี้ เป็นการนำเสนอวิธีการปรับปรุงเพื่อเพิ่มประสิทธิภาพของ ของการเข้ารหัสแบบอาร์เอสเอ ในแง่ของการลดการพลังงานและเพิ่มความเร็ว เพื่อให้สามารถทำงานบนมือถือได้ดีมากขึ้น โดยรายละเอียดต่างๆภายในวิทยานิพนธ์นี้ได้จัดแบ่งในส่วนเนื้อหาออกเป็น 5 บท ซึ่งแต่ละบทมีหัวข้อและเนื้อหาดังต่อไปนี้

บทที่ 1 “บทนำ” อธิบายถึงวัตถุประสงค์ สมมุติฐานของการศึกษา ขอบเขตและขั้นตอนการศึกษา รวมไปถึงรายละเอียดเนื้อหาโดยสรุปของแต่ละบท

บทที่ 2 “หลักการและงานวิจัยที่เกี่ยวข้อง” อธิบายการทำงานของ การเข้ารหัสแบบอาร์เอสเอ นอกจากนี้ยังได้อธิบายถึงการทำงานของ ARM920T โปรเซสเซอร์

บทที่ 3 “วิธีการปรับปรุงประสิทธิภาพเพื่อลดการใช้พลังงานและเพิ่มความเร็ว” บทนี้จะเสนอหลักการปรับปรุงประสิทธิภาพเพื่อเพิ่มประสิทธิภาพของ การเข้ารหัสแบบอาร์เอสเอ บนอาร์มโปรเซสเซอร์ ทั้งในแง่ของการลดการใช้พลังงานและการเพิ่มความเร็วในการทำงานของการเข้ารหัส อธิบายถึงหลักการและเหตุผลที่เป็นเช่นนี้

บทที่ 4 “การทดลองและผลการทดลอง” กล่าวถึงการเปรียบเทียบประสิทธิภาพของ การเข้ารหัสแบบอาร์เอสเอ บน อาร์มโปรเซสเซอร์ ทั้งก่อนและหลังการเพิ่มประสิทธิภาพในแง่ของพลังงานที่ใช้และความเร็วในการทำงานของการเข้ารหัส

บทที่ 5 “บทสรุป” เป็นการสรุปและวิจารณ์ผลที่ได้จากการวิเคราะห์และการทดลอง พร้อมทั้งปัญหาที่เกิดขึ้น ตลอดจนข้อเสนอแนะสำหรับนำงานวิจัยในวิทยานิพนธ์นี้ไปพัฒนาใช้ประโยชน์ต่อไป

# บทที่ 2
# หลักการและงานวิจัยที่เกี่ยวข้อง

## 2.1 งานวิจัยที่เกี่ยวข้อง

ในส่วนนี้จะนำเสนองานวิจัยที่คล้ายคลึงหรือเกี่ยวข้องกับงานวิจัยที่เราพัฒนา ซึ่งในวิทยานิพนธ์ฉบับนี้ จะสนใจงานวิจัยทางด้านการปรับปรุงประสิทธิภาพทางด้านการเพิ่มความเร็วและ การลดการใช้พลังงาน ทั้งบนแพลตฟอร์มพีซีและระบบสมองกลฝังตัว

### 2.1.1 งานวิจัยเกี่ยวกับการเพิ่มความเร็วในการทำงาน

**Real time implementation of MPEG-4 video decoder on ARM7TDMI** [1]

อธิบายเกี่ยวกับรายละเอียดในการทำให้ MPEG-4 Video Decoder บน AMR7TDMI ให้ใช้งานแบบเรียล-ไทม์ (real time) ได้โดยได้ทำการปรับปรุงประสิทธิภาพที่ระดับอัลกอริธึม และระดับแอสเซมบลีโค้ด

การปรับปรุงประสิทธิภาพที่ระดับอัลกอริธึม จะทำในภาษา C เป็นการเปลี่ยนอัลกอริธึมเพื่อลดการคำนวณเทคนิคที่ใช้ในการปรับปรุงประสิทธิภาพได้แก่

- ลดการทำงานของส่วนที่ซ้ำซ้อน

ลดการทำงานของโค้ดในส่วนที่ซ้ำซ้อน โดยในงานวิจัยชิ้นนี้จะมีการนำข้อมูลที่บ่งบอกได้ว่า ข้อมูลที่จัดเก็บในสถานที่จัดเก็บข้อมูลนั้น จะนำมาใช้ในการคำนวณด้วยหรือไม่จึงสามารถช่วยลดการทำงานที่ซ้ำซ้อนลงไปได้

- ลดการเข้าใช้หน่วยความจำ

การลดการใช้งานหน่วยความจำโดย ARM7TDMI นั้นไม่มีแคชอยู่ในตัว การเข้าใช้งานหน่วยความจำ ย่อมหมายถึงการเข้าใช้งานหน่วยความจำภายนอก ซึ่งทำให้เกิดกระบวนการที่เพิ่มขึ้น ตั้งแต่การเข้าใช้งานบัส เพื่อติดต่อกับหน่วยความจำภายนอกซึ่งอาจต้องรอช่วงเวลาเพื่อรอให้บัสว่างและพร้อมใช้งาน การถอดรหัสข้อมูลที่อยู่เพื่อเข้าถึงหน่วยความจำที่ต้องการ ซึ่งล้วนแล้วแต่เป็นกระบวนการทำงานที่เพิ่มขึ้น ทำให้ต้องใช้จำนวน Clock Cycle ในการทำงานมากขึ้น และจากวิธีการลดการทำงานของส่วนที่ซ้ำซ้อน ยังช่วยให้การใช้งานหน่วยความจำลดลงเนื่องจากไม่จำเป็นต้องใช้งานหน่วยความจำทั้งหมด ทำให้ลด Clock Cycle ที่ใช้ในการทำงานลงได้อีกด้วย

- ใช้การคูณแทนการหาร และเก็บค่าตัวคูณไว้ในตารางข้อมูล

ในการคำนวณการหารจะต้องใช้ Clock Cycle จำนวนมาก ราว 60-120 Cycles ในงานวิจัยชิ้นนี้จึงได้ทำการสร้างตารางข้อมูลสำหรับเก็บค่าที่คำนวณการหาร ที่ใช้งานบ่อยๆไว้แล้ว เมื่อมีการคำนวณ X/Y จะทำการค้นหาค่าในตารางที่เก็บ ค่า 1/Y เอาไว้เป็นค่า Z และนำค่า Z ที่ได้ไปคูณกับค่า X โดยจะได้ผลลัพธ์เป็น X x Z ทำให้โปรเซสเซอร์สามารถทำงานได้เร็วขึ้นเนื่องจากโปรเซสเซอร์สามารถคำนวณการคูณ โดยใช้ Clock Cycle สูงสุดเพียง 4 Cycles เท่านั้น

การปรับปรุงประสิทธิภาพที่ระดับแอสเซมบลี จะทำการปรับปรุงที่แอสเซมบลีโค้ดโดยใช้วิธีการได้แก่

- ปรับปรุงจำนวนตัวแปรที่ส่งให้กับฟังก์ชัน

การลดจำนวนตัวแปรที่ส่งให้กับฟังก์ชัน มีผลเนื่องจากใน ARM7TDMI มี register 4 ตัว ในการส่งค่าให้กับฟังก์ชัน ถ้าหากมีค่าที่ต้องส่งมากกว่า 4 ตัวต้องมีการนำค่าใส่สแต็ค (stack) เก็บไว้ ซึ่งก็จะมีกระบวนการทำงานที่เพิ่มขึ้นดั้งนั้นการลดจำนวนค่าที่ส่งให้น้อยกว่าเท่ากับ 4 ตัวจึงช่วยลดกระบวนการทำงานลงได้

- ลดการเข้าใช้หน่วยความจำโดยใช้คำสั่ง LDM และ STM

การลดจำนวนการเข้าใช้หน่วยความจำโดยใช้ LDM และ STM แทนเนื่องจาก LDM และ STM เป็นการ Load และ Store ข้อมูลที่ล่ะหลายเวิร์ด เมื่อนำมาใช้แทนคำสั่ง Load และ Store สามารถเข้าใช้งานหน่วยความจำโดยใช้จำนวนมากโดยใช้ Clock Cycle น้อยลงกว่าการใช้คำสั่ง Load และ Store

- หลีกเลี่ยงการใช้งานครึ่งเวิร์ด (Half Word) และ ไบต์ (Byte) Load/Store

หลีกเลี่ยงการใช้งานหน่วยความจำแบบครึ่งเวิร์ดและ ไบต์ โดยเปลี่ยนมาใช้ LDM และ STM แทนเนื่องจากมีประสิทธิภาพมากกว่า

- ปรับปรุงฟังก์ชัน Memset และ Memcpy

คอมไพเลอร์จะทำการสร้างฟังก์ชัน Memset และ Memcpy โดยใช้งานคำสั่ง LDM และ STM โดยฟังก์ชันเหล่านี้ถูกนำไปใช้งานกับข้อมูลทุกประเภทและทุกขนาดของบล็อก หลากหลายแตกต่างไป การตรวจสอบบางอย่างนั้นไม่จำเป็นจะต้องทำจึงได้ทำการปรับปรุงเพื่อให้ทำงานได้อย่างมีประสิทธิภาพมากขึ้น

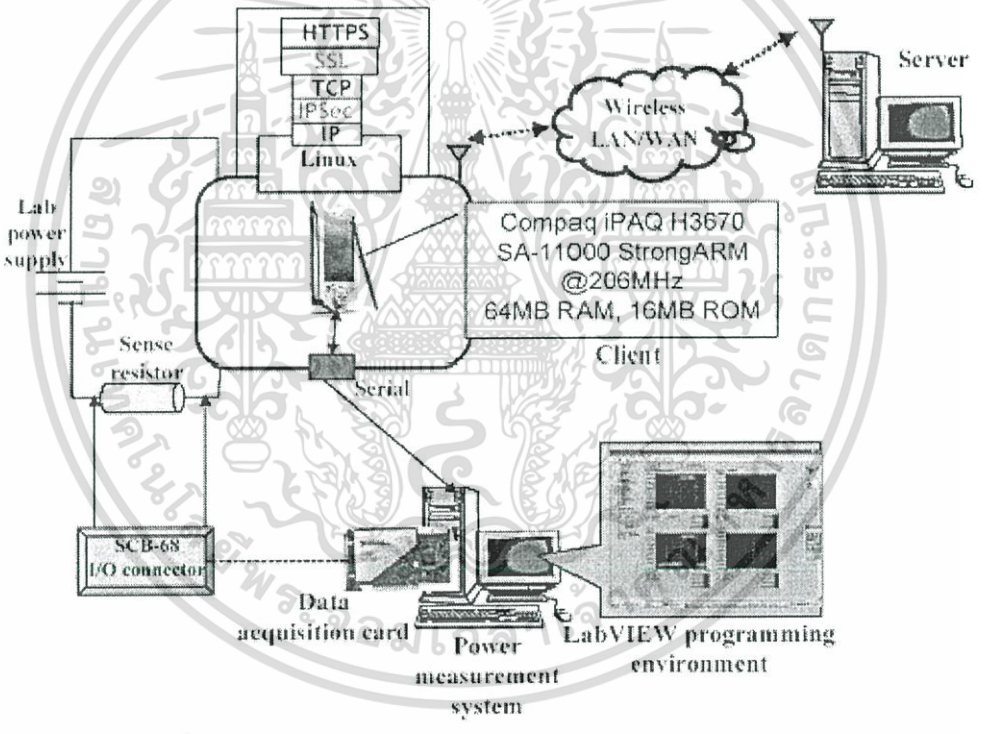
ผลลัพธ์ที่ได้จากการปรับปรุงประสิทธิภาพจากแต่เดิม ถอดรหัสภาพวีดีโอ MPEG-4 แบบ foreman ที่มีขนาด 15 fps เดิมใช้ 100 MHz ลดลงเหลือเหลือเพียง 16.0 MHz

### 2.1.2 งานวิจัยเกี่ยวกับวัดการใช้พลังงานในการเข้ารหัส

ระบบและแอพพลิเคชันที่เกี่ยวกับการเข้ารหัสจะใช้พลังงานส่วนใหญ่ไปกับการคำนวณ ดังนั้นในวิธีการลดพลังงานจึงเป็นสิ่งที่ท้าทายอย่างยิ่ง ในงานวิจัย

**A Study of the Energy Consumption Characteristics of Cryptographic Algorithms and Security Protocols** [2]

ได้ทำการศึกษาการใช้พลังงานในการเข้ารหัสแบบต่างๆ โดยใช้อุปกรณ์คือ Compaq iPAQ H3670 ซึ่งใช้อาร์มโปรเซสเซอร์ ที่มีความเร็ว 206 MHz โดยทำการทดลอง สร้างรหัส, ลายเซ็นต์ดิจิตอล, ตรวจสอบ และวัดพลังงานที่ใช้ไปโดยได้ทำการสร้างอุปกรณ์ มาวัดปริมาณกระแสไฟที่ใช้ ตั้งแต่เริ่มทดลองนั้นๆ จนได้ผลการทดลองซึ่งแสดงให้เห็นว่า การเข้ารหัสแบบอาร์เอสเอ เหมาะสมอย่างยิ่งสำหรับการเข้ารหัสบนอุปกรณ์คอมพิวเตอร์พกพา




**รูปที่ 2.1** การทดลองวัดการใช้พลังงานในการเข้ารหัสแบบต่างๆ

| Algorithm | Key size ($bits$) | Key generation ($mJ$) | Sign ($mJ$) | Verify ($mJ$) |
|---|---|---|---|---|
| RSA | 1,024 | 270.13 | 546.50 | 15.97 |
| DSA | 1,024 | 293.20 | 313.60 | 338.02 |
| ECDSA | 163 | 226.65 | 134.20 | 196.23 |
| ECDSA | 193 | 281.65 | 166.75 | 243.84 |
| ECDSA | 233 | 323.30 | 191.37 | 279.82 |
| ECDSA | 283 | 504.96 | 298.86 | 437.00 |
| ECDSA | 409 | 1034.92 | 611.40 | 895.98 |

**ตารางที่ 2.1** ผลการทดลองที่แสดงให้เห็นการใช้พลังงานของการเข้ารหัสแบบต่างๆ

**Memory Exploration for Low Power, Embedded Systems** [17]

ในงานวิจัยชิ้นนี้เป็นงานวิจัยเพื่อศึกษาเกี่ยวกับ การใช้พลังงานของหน่วยความจำ เพื่อที่จะหาขนาดของแคช ที่เหมาะสมเพื่อให้ใช้พลังงานให้น้อยที่สุด โดยในงานวิจัยนี้จะให้ความสนใจกับตัวแปร 3 ตัวได้แก่

- ขนาดของแคช โดยการเลือกขนาดของแคช จะรวมถึงจำนวนขนาดของแถว ในแคชซึ่งจะมีผลกับ อัตราการค้นหาข้อมูลในแคชไม่พบ(Miss Rate) โดยหากขนาดของแถวเพิ่มขึ้นจะมีผลทำให้อัตราการค้นหาข้อมูลในแคชไม่พบลดลง เพราะจะทำให้มีโอกาสที่ข้อมูลในหน่วยความจำหลักถูกดึงขึ้นมาแล้วจากการอ่านข้อมูลจากหน่วยความจำมาเก็บไว้ในแคชเพิ่มขึ้น
- จำนวนของ Clock Cycles ที่ใช้ในกรณีการทดลอง ซึ่งมีค่าเท่ากับ

จำนวน **Clock Cycles = Hit rate x trip_count x (number of cycles per hit)**

**+ Miss Rate x trip_count x (tiling size + number of cycles per miss)**

- พลังงานที่ใช้ในการใช้งานหน่วยความจำ ซึ่งมีค่าเท่ากับ

**Energy = Hit Rate x Energy_hit + Miss Rate x Energy_miss**

Energy_hit = E_dec+E_cell

Energy_miss = E_dec+E_cell+E_io+E_main

= Energy_hit + E_io+E_main

- E_dec = a x (Add_bs)

- E_cell = b x (Word_line_size) x (Bit_line_size)

- E_io = g x (Data_bs x Cache_line_size + Add_bs)

- E_main = g x (Data_bs x Cache_line_size ) + Em x Cache_line_size

Add_bs = ขนาดของบัสแอดเดรส ต่อคำสั่ง

Data_bs = ขนาดขอบัสข้อมูล ต่อคำสั่ง

Word_line_size = จำนวนของหน่วยความจำในรูปแบบเวิร์ด (word) ต่อแถว.

Bit_line_size = จำนวนของหน่วยความจำในรูปแบบบิตต่อแถว.

Em = พลังงานที่ใช้ในการใช้งานหน่วยความจำหลัก

a=0.001, b=2 and g =20 สำหรับ 0.8 mm CMOS technology

โดยในงานวิจัยนี้ได้ทำการทดลอง โดยจำลองการทำงานของการถอดรหัสภาพด้วยวิธี MPEG-4 และคำนวณหาค่าอัตราการใช้พลังงาน ในกรณีที่แคชมีขนาดและรูปแบบต่างๆกันไป โดยสามารถคำนวณหาค่าพลังงาน ที่ใช้ในกรณีที่แคชมีขนาด 64 ไบต์ ขนาดแถวของแคช ขนาด 4 ไบต์ 8-way ขนาด tiling 16 จะใช้พลังงาน 293,000 nJ ใช้ Clock Cycles 142,000 รอบและสำหรับในกรณีที่กำหนดให้แคชนั้นมีขนาด 512 ไบต์ และแต่ละแถวของแคชจะมีขนาด 16 ไบต์ 8-way ขนาด tiling 8 จะใช้พลังงาน 1,110,000 nJ และใช้ 121,000 Clock Cycles

## 2.2 การเข้ารหัสข้อมูลแบบอาร์เอสเอ

**2.2.1** บทนำ อาร์เอสเอ คืออัลกอริธึมสำหรับการเข้ารหัสแบบกุญแจสาธารณะ (public-key encryption) เป็นอัลกอริธึมแรกที่ทราบว่าเหมาะสำหรับลายเซ็นดิจิตอลรวมถึงการเข้ารหัส เป็นหนึ่งในความก้าวหน้าครั้งใหญ่ครั้งแรกในการเข้ารหัสแบบกุญแจสาธารณะ(public-key) โดย อาร์เอสเอ ยังคงใช้ในโปรโตคอลสำหรับการค้าอิเล็กทรอนิกส์ (electronic commerce) และเชื่อว่า มีความปลอดภัย เมื่อมีความยาวของรหัสที่ยาวพอ

**ความเป็นมาของการเข้ารหัสแบบอาร์เอสเอ**

อัลกอริธึมได้ถูกอธิบายเมื่อพ.ศ. 2520 โดย รอน ริเวสต์ (Ron Rivest), อาดี ชามีร์ (Adi Shamir) และ เล็น แอเดิลแมน (Len Adleman) ที่ MIT โดยที่มาชื่ออาร์เอสเอ มาจากนามสกุล ของทั้ง 3 คน เป็นที่เล่ากันว่า คิดค้นขึ้นระหว่างพิธีกรรมทางศาสนาของชาวยิว (Passover seder) ในเมือง สเกเน็กตาดี มลรัฐนิวยอร์ก (Schenectady, NY)

คลิฟฟอร์ด ค็อกส์ (Clifford Cocks) นักคณิตศาสตร์ชาวอังกฤษที่ทำงานใน GCHQ ได้อธิบายระบบที่มีความเหมือนกันในเอกสารภายใน เมื่อพ.ศ. 2516 แต่เนื่องจากในตอนนั้น จะต้องใช้คอมพิวเตอร์ราคาแพงเพื่อนำไปใช้จริง จึงถือเป็นความแปลกใหม่ในขณะนั้น และเท่าที่ปรากฏต่อสาธารณะ ไม่เคยใช้งานจริงนอกจากนี้ การค้นพบครั้งนี้ไม่ถูกเปิดเผยจนถึงพ.ศ. 2540 เนื่องจากเป็นความลับ

อัลกอริธึมนี้ได้ถูกจดสิทธิบัตรโดย MIT เมื่อปี พ.ศ. 2526 ในประเทศสหรัฐอเมริกา เป็นสิทธิบัตรหมายเลข 4405829 ซึ่งได้หมดอายุลงเมื่อวันที่ 21 กันยายน พ.ศ. 2543 แต่เนื่องจากอัลกอริธึมนี้ได้พิมพ์แล้วก่อนที่จะจดสิทธิบัตร กฎหมายในส่วนอื่นๆของโลก ทำให้ไม่สามารถจดสิทธิบัตรที่อื่นได้ และเนื่องจากในกรณีที่ผลงานของค็อกส์ ได้เป็นที่รู้จักกันในสาธารณะ การจดสิทธิบัตรในสหรัฐอเมริกาจึงไม่สามารถจะกระทำได้เช่นกัน

**2.2.2** กระบวนการเข้ารหัสแบบอาร์เอสเอ

การสร้างรหัส(Key Generator) ทำได้โดยการหาจำนวนเฉพาะ สองตัวคือ P และ Q เพื่อนำมาใช้ในการคำนวณ ค่าดังต่อไปนี้

N (MODULUS): ได้จากการนำค่า P x Q โดยขนาดของข้อมูลที่จะทำการเข้ารหัสจะต้องมีขนาดไม่เกินค่า N เพราะจะทำให้ข้อมูลที่เข้ารหัสไม่สามารถ ถอดได้ด้วย Private Key ที่เป็นคู่กัน

M : ได้จากการนำค่า (P-1) x (Q-1)

E (Public Key): ได้จากการเลือกค่าตัวเลข ที่ไม่เป็นตัวประกอบของ M และน้อยกว่า M หรือ ตัวหารร่วมของ M และ E จะเป็น 1 เท่านั้น

D (Private Key): ได้จากการนำค่า $E^{-1}$ mod (M) หรือก็คือ Inverse ของ E modulo M

*** หมายเหตุ การเข้ารหัสโดยวิธีอาร์เอสเอ จะต้องเก็บค่า P และ Q ไว้เป็นความลับ เพราะถ้าหากทราบค่า P และQ จะสามารถหารหัสได้

**2.2.3** ตัวอย่างการทำงานของการเข้ารหัสแบบอาร์เอสเอ

กำหนดให้ค่า P = 19 และ Q = 7 จะสามารถคำนวณค่าต่างๆ ได้ดังนี้

$N = P \times Q = 133$

$M = (P-1) \times (Q-1) = 108$

ทำการสุ่มเลือกค่า E ที่น้อยกว่า M และไม่เป็นตัวประกอบของ M โดยให้เท่ากับ 5

$D = E^{-1} \bmod (M) = 65$

ดังนั้นจะได้ Public key เป็นการจับคู่ของ E และ N

Public key {E, N} = {5, 133}

และจะได้ Private key เป็นการขับคู่ของ D และ N

Private key {D, N} = {65, 133}

การเข้ารหัสข้อมูลจะทำได้โดย

- แบ่งขนาดของข้อมูล ถ้าค่าของข้อมูลมากกว่า N-1 จะไม่ทำงานได้ เพราะอาร์เอสเอเป็นการเข้ารหัสแบบเป็นบล็อคโดยขนาดของข้อมูลที่จะทำการเข้ารหัสไม่สามารถมีขนาดมากกว่า N-1 ได้ เพราะจะทำให้เมื่อทำการถอดรหัสโดยใช้ Private Key ที่เป็นคู่กันแล้ว ไม่สามารถถอดได้อย่างถูกต้อง
- แทนค่าข้อมูลที่ต้องการเข้ารหัสให้อยู่ในรูปของตัวเลข โดยตัวเลขนั้นจะต้องมีค่าน้อยกว่าค่า N เช่น แปลงค่า A เป็น 1, B เป็น 2 ฯลฯ แต่ตามมาตรฐาน จะมีวิธีการเปลี่ยนคือ OS2IP เปลี่ยนค่าข้อมูลไปเป็นค่า Integer และวิธีเปลี่ยนค่ากลับคือ ISOSP เปลี่ยนค่า Integer ที่จากการเข้ารหัสแบบอาร์เอสเอแล้วกลับเป็นข้อมูล
- นำค่าตัวเลขที่เป็นข้อมูลที่ต้องการจะเข้ารหัสไปยกกำลังด้วย Public key (E) และ modulo ด้วยค่า N ด้วยสมการ Encrypt data $= (\text{Data}^{E}) \bmod N$

ข้อมูลที่ได้จะเป็นข้อมูลตัวเลขที่ทำการเข้ารหัสแล้วซึ่งสามารถนำไปใช้ได้ต่อไป

- ส่วนการถอดรหัส ก็จะทำได้ด้วยสมการเดียวกันเพียงแต่เปลี่ยนค่า ดังสมการนี้

  Data = (Encrypt data $^{D}$) mod N

ตัวอย่างเช่น

ต้องการเข้ารหัสข้อมูล ตัวอักษร C เพียงตัวเดียวโดยใช้ ค่า modulus (N) และ Public key (E) สามารถทำได้ดังนี้

- แทนค่า C ให้อยู่ในรูปของตัวเลข C = 3
- ข้อมูลที่ถูกเข้ารหัสจะเป็นดังนี้ $3^{5}$ mod 133 = 110 ซึ่ง 110 นี้ก็คือ ข้อมูลที่ถูกเข้ารหัส
- เมื่อต้องการจะถอดรหัสข้อมูล ก็จะเป็นดังนี้ $110^{65}$ mod 133 เท่ากับ 3
- จะเห็นว่า ถ้าหากข้อมูล C ซึ่งถูกแปลงค่าแล้วนั้น มากกว่า N จะทำให้ไม่สามารถถอดรหัสได้ด้วย Private Key ที่สร้างขึ้นมา ดังนั้นการเข้ารหัสแบบอาร์เอสเอ จึงต้องทำการเข้ารหัสแบบบล็อค คือจะต้องทำการแบ่ง ข้อมูลที่จะทำการเข้ารหัสออกเป็นบล็อค โดยที่มีค่าไม่เกินค่า N-1

**2.2.4** ตัวอย่างรหัสความยาวขนาด 1024 บิต

E=7838162666637764660488506072069633402758550968767305597001995267016039450753870941765191196559064086531146869581726024003668713773612868051457112298081133630661830955224946756386490602942845606348691551116490806172581156608186811340305409527578241815320247188813852579393316788607529224958883842957287498847 9

N=9771205012606097877813393948207228623551699794974803978331750966752367597035927437354883116708046953787655222087870033126353643093243441842345706393762250421221735188098316954426928993308791986982460536680709370785530723936279398218718173607960782048876769784413333592430796474863017249843061794168497342943 1

**รูปที่ 2.2** ตัวอย่างของอาร์เอสเอรหัสขนาด 1024 บิต

วิธีการ เตรียมข้อมูลก่อนทำการเข้ารหัส

ในมาตรฐาน การเข้ารหัสแบบกุญแจสาธารณะ เราจะมีกระบวนการที่ทำหน้าที่แปลงข้อมูลให้เป็นตัวเลขและตัวเลขให้เป็นข้อมูล เพื่อที่จะสามารถนำมาทำการเข้ารหัสแบบกุญแจสาธารณะ ได้อย่างเป็นมาตรฐาน คือ

- I2OSP: Integer-to-Octet-String primitive คือการแปลงค่า Integer ให้เป็น Octet String
- OS2IP: Octet-String-to-Integer primitive คือแปลง Octet String ให้กลับเป็น Integer

## 2.3 สถาปัตยกรรม ARM920T โปรเซสเซอร์

ARM โปรเซสเซอร์เป็นโปรเซสเซอร์ที่ได้รับความนิยมอย่างมากในระบบสมองกลฝังตัว โดยเฉพาะอย่างยิ่งในมือถือและพีดีเอซึ่ง ARM โปรเซสเซอร์จะแบ่งออกเป็นหลายซีรี่เพื่อรองรับความต้องการที่หลากหลาย งานวิจัยเรามุ่งเน้นที่จะใช้ได้จริงบนมือถือและโค้ดของเราทำงานบนระบบปฏิบัติการซิมเบียน ซึ่งมือถือที่ใช้ระบบปฏิบัติการนี่จะใช้ ARM ซีรี่ 9 ซึ่งใน ARM ซีรี่ 9 มีสองตัวที่ใช้ในมือถือและพีดีคือ ARM920T และ ARM922T ทั้งสองตัวนี้มีสถาปัตยกรรมการทำงานที่เหมือนกันต่างกันตรงที่ขนาดของแคช (cache) ที่ ARM920T มีขนาด 16 กิโลไบต์ ส่วน ARM922T ขนาด 8 กิโลไบต์ ในงานวิจัยเราได้เลือกศึกษา ARM920T เพราะมีแคชขนาดที่ใหญ่กว่า ARM922T เพื่อที่จะได้เห็นผลในการทดลองที่ชัดเจน

### 2.3.1 เกี่ยวกับ ARM920T

ARM920T เป็นสมาชิกหนึ่งในตระกูล ARM9TDMT ที่เป็นโปรเซสเซอร์ที่ใช้ในงานทั่วไป ซึ่งประกอบด้วย

ARM9TDMI (core)

ARM940T (core บวก แคชและ protection unit)

ARM920T และ ARM922T (core บวก แคชและ MMU)

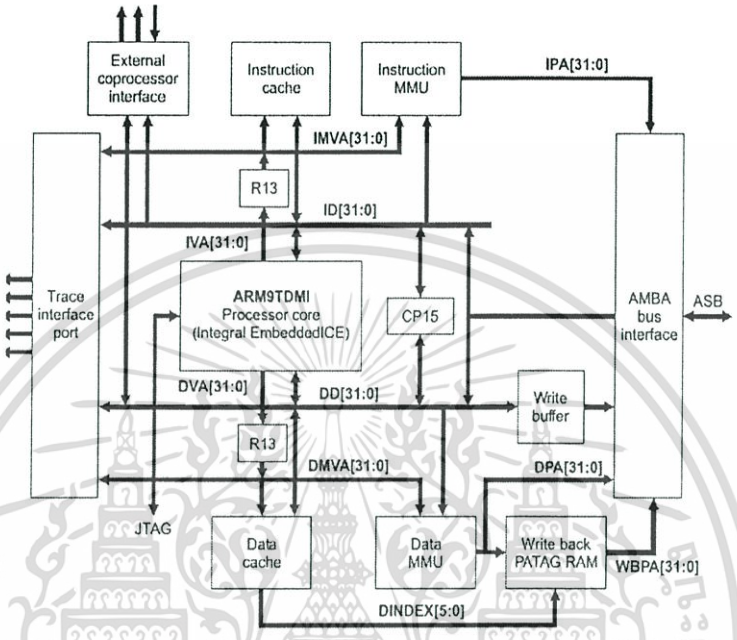
ARM9TDMI โปรเซสเซอร์ใช้สถาปัตยกรรมที่เรียกว่า Harvard โดยมีห้าสถานะของไปป์ไลน์ ซึ่งประกอบด้วย เฟ็ตช์ (fetch), ดีโค้ด (decode), ประมวลผล (execute), เข้าใช้หน่อยความจำ (memory access) และเขียนรีจีสเตอร์ (write register back) โดยARM9TDMI สามารถทำงานได้ด้วยตัวเอง นั้นคือสามารถที่จะนำ ARM9TDMI ไปฝังไว้ในอุปกรณ์ที่ซับซ้อนได้ โดยการจะให้ ARM9TDMI สามารทำงานได้ด้วยตัวเอง จะต้องมี simple bus interface ที่อนุญาตให้ออกแบบระบบแคช และหน่วยความจำหลักได้

โปรเซสเซอร์ตระกูล ARM9TDMI รองรับทั้งชุดคำสั่ง 32 บิต ARM และ 16 บิต Thumb ทำให้เราสามารถเลือกได้ว่าต้องการโค้ดที่มีประสิทธิภาพสูงหรือจะเอาโค้ดที่มีขนาดเล็ก

ARM920T โปรเซสเซอร์ใช้สถาปัตยกรรมชื่อว่า Harvard cache จุดประสงค์เพื่อใช้กับมัลติโปรแกรมเมอร์แอพพลิเคชัน ที่มีการจัดการหน่วยความจำที่มีประสิทธิภาพสูง และใช้พลังงานน้อย มีการแยกแคชคำสั่งและแคชข้อมูล ขนาด 16 กิโลไบต์ที่มีแถวล่ะ 8 เวิร์ด (word) ARM920T และมีการสร้าง enhanced ARM architecture v4 MMU ที่ทำหน้าที่การจัดการการตรวจสอบสิทธิในการเข้าใช้ และแปลงค่าสำหรับหาตำแหน่ง คำสั่ง และ ข้อมูล

### 2.3.2 Block diagram ของฟังก์ชันการทำงานของโปรเซสเซอร์

รูป 2.3 แสดง Block diagram ของฟังก์ชันการทำงานของ ARM920T โปรเซสเซอร์




**รูปที่ 2.3** ARM920T Block diagram

สถาปัตยกรรม ARM920T เป็น RISC โปรเซสเซอร์ขนาด 32 บิตที่ใน ARM920T เป็นโปรเซสเซอร์ภายในประกอบด้วย ARM9TDMI โปรเซสเซอร์กับ:

-แคชของชุดคำสั่งและแคชของข้อมูลขนาด 16 กิโลไบต์

-MMU หรือหน่วยจัดการหน่วยความจำคำสั่งและข้อมูล

(Instruction and data Memory Management Units)

-บัฟเฟอร์ก่อนเขียนลงหน่อยความจำ (Write buffer)

-AMBA™ (Advanced Microprocessor Bus Architecture) บัสอินเตอร์เฟส

-บัสในการส่งข้อมูลและคำสั่งขนาด 32 บิต

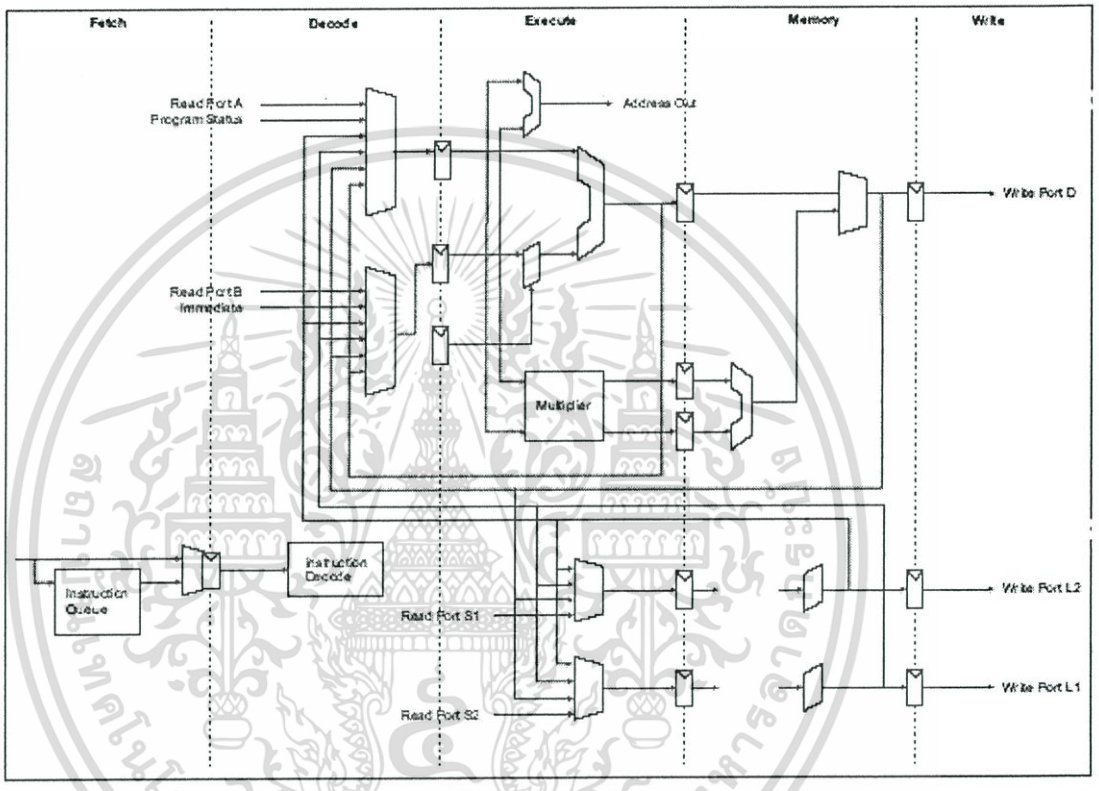
-Embedded Trace Macrocell (ETM) อินเตอร์เฟส

### 2.3.3 ARM920T Macrocell

ARM920T Macrocell อยู่บนพื้นฐานสถาปัตยกรรม ARM9TDMI Harvard โดยมีไปป์ไลน์ ขนาดห้าขั้น: เฟ็ตช์ (fetch), ดีโค้ด (decode), ประมวลผล (execute), เข้าใช้หน่วยความจำ (memory access) และเขียนรีจีสเตอร์ (write register back) แสดงดังรูปที่ 2.4

เพื่อลดแบนด์วิดท์และความหนาแน่นของข้อมูลที่หน่วยความจำหลัก ARM920T จึงมี

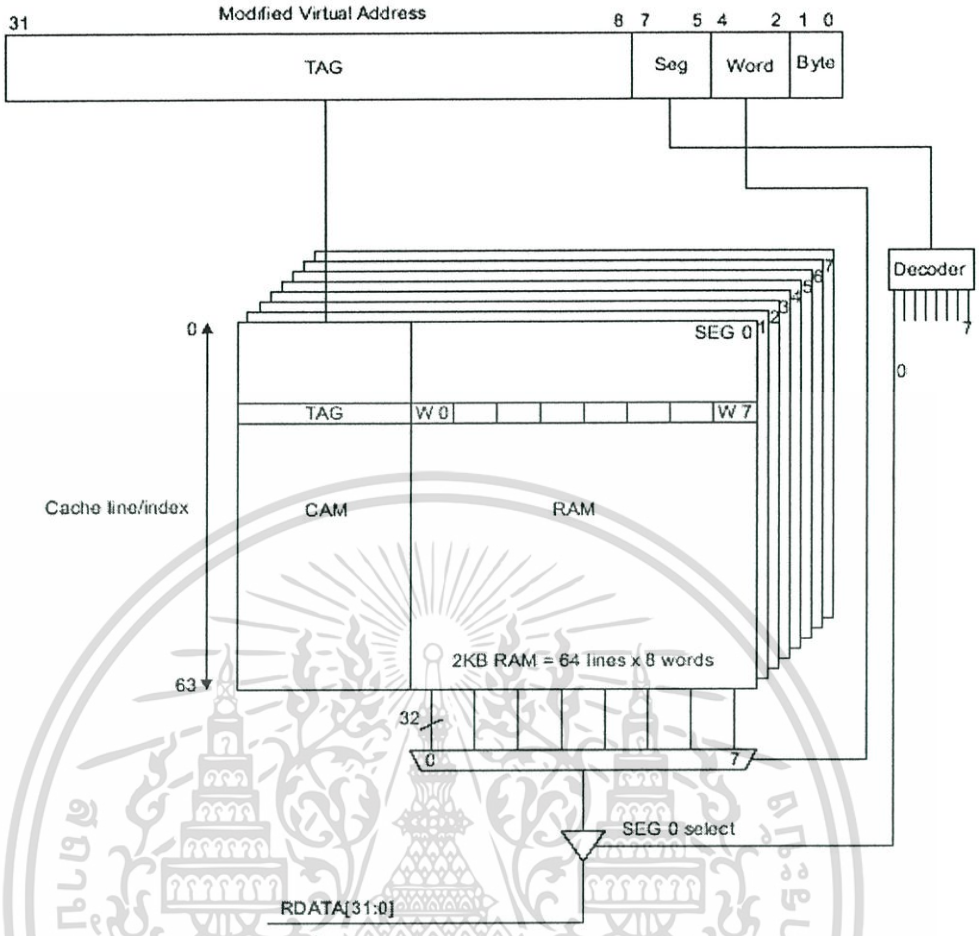
- แคชชุดคำสั่ง
- แคชข้อมูล
- MMU
- TLBs
- Write buffer




รูปที่ 2.4 ไปป์ไลนของ ARM9TDMI

### 2.3.4 แคชของ ARM920T

ทั้งแคชของคำสั่งและของข้อมูลทั้งสองมีขนาด 16 กิโลไบต์ มีขนาดแถวละ 8 เวิร์ดและเป็นแบบ 64-way set associative [4] จึงมีขนาด 8 Segments และมีบัสข้อมูลขนาด 32 บิตเชื่อมต่อไปที่ ARM9TDMI และเป็นการจองข้อมูลบนแคชเมื่อหาข้อมูลไม่เจอในแคช นอกจากนี้ยังใช้นโยบาย Write thought และไม่ทำการจองหน่วยความจำเมื่อการเขียนข้อมูลแล้วไม่พบในแคช (Write miss) ส่วนการเขียนข้อมูลที่มีในแคช (Write hit) จะใช้ Write back




รูปที่ 2.5 โครงสร้างการจัดการของแคช

**2.3.5 แอพพลิเคชันที่ใช้บน ARM920T**

ระบบปฏิบัติการ

-EPOC (ประเภทระบบปฏิบัติการซิมเบียน)

-Linux

-WindowsCE

แอพพลิเคชันไร้สายประสิทธิภาพสูง

-Smart Phone

-พีดีเอ

-แอพพลิเคชันบนเครือข่าย

-การเข้ารหัสและคล้ายรหัสของเสียงและภาพ

## 2.4 ระบบปฏิบัติการซิมเบียน (Symbian OS)

ระบบปฏิบัติการซิมเบียน เป็นระบบปฏิบัติการที่ออกแบบมาเพื่อรองรับเทคโนโลยีการสื่อสารแบบไร้สายบนโทรศัพท์เคลื่อนที่ในปัจจุบัน สามารถที่จะทำงานได้โดยไม่ขึ้นต่อฮาร์ดแวร์ (Hardware) ของโทรศัพท์เคลื่อนที่ แม้มีความแตกต่างกัน และสามารถจัดการกับการพัฒนาอย่างต่อเนื่องในอนาคตได้ และช่วยในด้านของการส่งข้อมูลของโทรศัพท์เคลื่อนที่ อีกทั้งยังเป็นระบบที่ใช้งานได้ง่าย มีความปลอดภัยสูง ช่วยประหยัดพลังงาน และใช้หน่วยความจำที่มีขนาดเล็ก เพื่อรองรับกับโทรศัพท์เคลื่อนที่ทั้งในปัจจุบันและอนาคต

ทั้งนี้ระบบปฏิบัติการซิมเบียนนั้นยังเป็นระบบปฏิบัติการเปิด อนุญาตให้โปรแกรมเมอร์รวมถึงผู้ใช้สามารถพัฒนาซอฟท์แวร์ต่างๆ บนระบบปฏิบัติการซิมเบียนได้เอง ทำให้อาจกล่าวได้ว่าในอนาคตจะมีแอพพลิเคชันมากมายที่ถูกสร้างขึ้นบนระบบปฏิบัติการซิมเบียน และยังส่งผลให้เป็นตลาดผลิตภัณฑ์ซอฟท์แวร์ที่ยิ่งใหญ่ในอนาคตอีกด้วย

### 2.4.1 ความเป็นมาและพัฒนาการของระบบปฏิบัติการซิมเบียน

ซิมเบียน คือ บริษัทผลิตซอฟท์แวร์ (Software) ที่เป็นผู้นำในด้านการผลิตซอฟท์แวร์ที่รองรับเทคโนโลยีการสื่อสารแบบไร้สาย (Wireless) ซึ่งผลิตภัณฑ์ที่ได้รับการยอมรับ จากบริษัทคู่ค้าก็คือ ระบบปฏิบัติการซิมเบียนนั่นเอง

ระบบปฏิบัติการซิมเบียนเกิดขึ้นในเดือนมิถุนายน ปี ค.ศ.1998 ซึ่งในตอนนั้นมีพันธมิตรร่วมกัน 4 ราย คือ Ericsson, Motorola, Nokia และ Psion มีสำนักงานใหญ่อยู่ที่ประเทศอังกฤษ

ถัดมาในปี ค.ศ. 1999 ซิมเบียนก็ได้พันธมิตรรายใหม่นั่นคือ Matsushita (Panasonic) ในปี ค.ศ. 2000 พันธมิตรของซิมเบียน ก็มากขึ้นอีกโดยมีการจับมือกับ Sony, Sanyo และ Kenwood รวมถึงการเปิดตัวโทรศัพท์ที่ใช้ระบบปฏิบัติการซิมเบียน เครื่องแรกในรูปแบบของสมาร์ทโฟนนั่นก็คือ Ericsson R380 ซึ่งมีคุณสมบัติการใช้งานที่มากมาย และยังมีทำงานด้วยหน้าจอระบบสัมผัสอีกด้วย

ในปี 2001 Nokia ที่เคยใช้ระบบปฏิบัติการ GEOS (Graphical Environment Operating System) ได้เปลี่ยนมาใช้ระบบปฏิบัติการซิมเบียนแทนระบบปฏิบัติการเดิมโดยทาง Nokia ได้พัฒนาคอมมิวนิเคเตอร์ (Communicator) รุ่นใหม่ออกมาอีกคือ Nokia 9210 และโทรศัพท์ที่ใช้แพลทฟอร์ม (Platform) Series 60 รุ่นแรกคือ Nokia 7650 ซึ่งระบบปฏิบัติการที่ใช้นั้นแตกต่างจาก Ericsson R380 เนื่องจากระบบปฏิบัติการซิมเบียนของ Nokia 9210 และ Nokia 7650 นั้นเป็นระบบปฏิบัติการเปิด (Open Symbian OS) คือสามารถที่จะนำโปรแกรมอื่นที่สามารถทำงานบนระบบปฏิบัติการซิมเบียนมาลงเพิ่มในเครื่องได้

ในปี 2002 ทางซิมเบียนได้มีการพัฒนาระบบปฏิบัติการรุ่นใหม่ที่เรียกว่าระบบปฏิบัติการซิมเบียนเวอร์ชัน 7.0 และทาง Sony Ericsson ก็เข้ามาเป็นพันธมิตรและเป็นหุ้นส่วนรายใหญ่ รวมทั้งทาง Sony Ericsson ก็เปิดตัวสมาร์ทโฟนรุ่นล่าสุด คือ Sony Ericsson P800 และในปี 2003

ก็ได้มีการพัฒนาระบบปฏิบัติการต่อจากเวอร์ชัน 7.0 เป็นเวอร์ชัน 7.0s โดยได้เพิ่มส่วนของการรองรับระบบโทรศัพท์แบบดับบลิวซีดีเอ็มเอ (WCDMA: Wideband Code Division Multiple Access) และได้มีการปรับปรุงการรองรับ Java เป็น MIDP 2.0

และในปี 2004 ทางซิมเบียน ก็ได้ทำการเปิดตัวระบบปฏิบัติการซิมเบียนเวอร์ชันใหม่ล่าสุดนั่นก็คือเวอร์ชัน8.0 โดยที่ระบบปฏิบัติการตัวนี้ได้ถูกสร้างขึ้นมาเพื่อพยายามลดค่าใช้จ่ายที่เกิดขึ้นจากการพัฒนาโทรศัพท์เคลื่อนที่ในอนาคต จนกระทั่งปัจจุบันเวอร์ชันของระบบปฏิบัติการซิมเบียน อยู่ที่เวอร์ชัน 9.0 ในปี 2006

**2.4.2 โครงสร้างของระบบปฏิบัติการซิมเบียน**

ในส่วนโครงสร้างของระบบปฏิบัติการซิมเบียนประกอบไปด้วยส่วนต่างๆ ที่สำคัญดังนี้

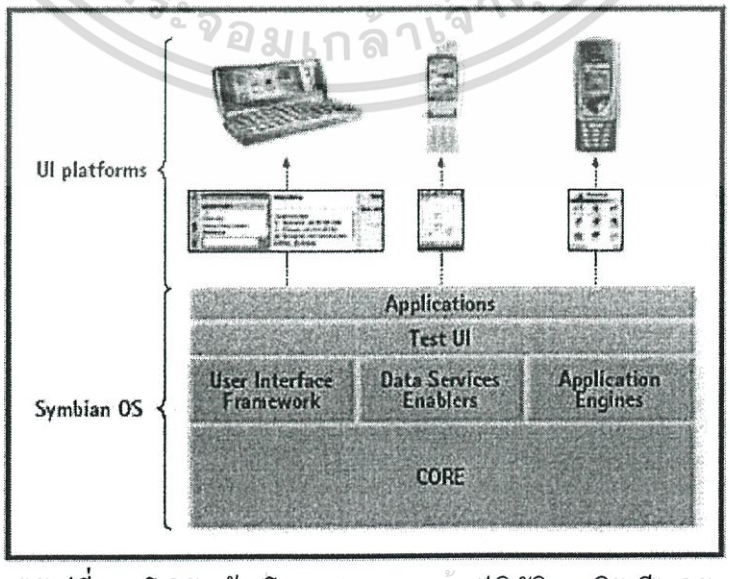
1. ส่วนแกนหลัก (Core) ของระบบปฏิบัติการซิมเบียนเป็นแกนหลัก ในการทำงานของอุปกรณ์ เช่น เคอร์เนล (Kernel) ไฟล์เซิร์ฟเวอร์ (File server) ทำหน้าที่จัดการด้านหน่วยความจำ (Memory management) และไดรเวอร์ของอุปกรณ์ (Device driver) โดยที่ส่วนประกอบต่างๆ นั้นสามารถที่จะเพิ่มหรือลดได้ขึ้นอยู่กับอุปกรณ์แต่ละชนิด

2. ชั้นของระบบ (System Layer) เป็นส่วนที่จัดการเกี่ยวกับการคำนวณ การติดต่อสื่อสาร เช่น TCP/IP, IMAP4, SMS และ การจัดการฐานข้อมูล

3. แอพพลิเคชันเอนจิน (Application Engine) ซึ่งอยู่บนชั้นของระบบ ทำให้นักพัฒนาโปรแกรมสามารถสร้างส่วนติดต่อกับผู้ใช้ (User Interface) ติดต่อกับข้อมูลได้

4. ส่วนติดต่อกับผู้ใช้ โดยผู้ผลิตแต่ละรายสามารถที่จะสร้างขึ้นได้ เช่นรูปแบบ Series 60 ของบริษัท Nokia

5. ชั้นของแอพพลิเคชัน (Application Layer) ซึ่งอยู่ส่วนบนของส่วนติดต่อกับผู้ใช้




**รูปที่ 2.6** โครงสร้างโดยรวมของระบบปฏิบัติการซิมเบียน

**2.4.3 การจัดการเกี่ยวกับพลังงาน (Power management)**

การจัดการเกี่ยวกับพลังงานได้ถูกสร้างขึ้นในเคอร์เนลของระบบปฏิบัติการซิมเบียน การออกแบบช่วยทำให้การใช้พลังงานของโปรเซสเซอร์(Processor) และอุปกรณ์ต่างๆ เป็นไปอย่างมีประสิทธิภาพมากยิ่งขึ้น เมื่ออุปกรณ์ชนิดใดไม่ถูกใช้งานก็จะถูกปิดโดยระบบ และการที่มีระบบการจัดการพลังงานที่ดีเช่นนี้เองทำให้สามารถยืดอายุการใช้งานของแบตเตอร์รี่ที่มีขนาดเล็กและมีอยู่อย่างจำกัดได้

**2.4.4 ความทนทานและความน่าเชื่อถือ (Robustness and Reliability)**

ระบบปฏิบัติการซิมเบียน ได้รับความนิยมและประสบความสำเร็จ ในด้านประสิทธิภาพในระบบโทรศัพท์เคลื่อนที่ โดยอุปกรณ์ไม่ควรที่จะสูญเสียข้อมูลเมื่อ ระบบล่มหรือมีปัญหา หรือเมื่อมีการรีบูต สิ่งที่ทำให้ระบบปฏิบัติการซิมเบียน ประสบความสำเร็จ คือ

1. แต่ละกระบวนการทำงานอยู่บนพื้นที่แอดเดรส (Address space) ที่มีการป้องกัน ทำให้แอพพลิเคชันใดๆ ไม่สามารถเขียนข้อมูลทับในพื้นที่แอดเดรสของแอพพลิเคชันอื่นได้
2. เคอร์เนลทำงานอยู่บนพื้นที่แอดเดรสที่มีการป้องกันจึงทำให้ข้อผิดพลาดของโปรแกรมต่างๆไม่สามารถเขียนข้อมูลทับลงในส่วนสแต็ค หรือหน่วยความจำฮีพ(Heap) ของเคอร์เนลได้

**2.4.5 การจัดการหน่วยความจำ (Memory management)**

ในอุปกรณ์เคลื่อนที่การจัดการหน่วยความจำมีความจำเป็นอย่างยิ่ง อันเนื่องมาจากการที่หน่วยความจำมีขนาดจำกัด โดยในระบบปฏิบัติการซิมเบียนนั้นระบบจะมองว่าหน่วยความจำมีขนาดจำกัดและจะใช้หน่วยความจำให้น้อยที่สุดในการทำงานแต่ละครั้ง การใช้หน่วยความจำน้อยนี้ทำให้การใช้พลังงานลดลงด้วย

**2.4.6 การทำงานแบบมัลติทาสกิ้ง (Multitasking)**

แอพพลิเคชันต่างๆในระบบปฏิบัติการซิมเบียนทำงานอยู่ต่างกระบวนการกัน จึงสามารถทำให้หลายแอพพลิเคชันทำงานได้พร้อมกันด้วย ตัวอย่างเช่น เมื่อผู้ใช้ต้องการตรวจสอบรายการประจำวันและรับสายโทรเข้า ระบบต้องสามารถที่จะสลับการทำงานระหว่างสองกระบวนการได้อย่างรวดเร็ว ผู้ใช้สามารถดูรายการประจำวันขณะที่มีการโทรได้ ซึ่งในปัจจุบันโทรศัพท์เป็นสิ่งที่ให้สามารถให้ข้อมูลอย่างมาก ดังนั้นความสามารถนี้จึงเป็นสิ่งที่สำคัญ

**2.4.7 แอพพลิเคชันเฟรมเวิร์ค (Application Framework)**

การทำงานของแอพลิเคชันที่ใช้ระบบปฏิบัติการซิมเบียน บนอีมูเลเตอร์ (Emulator) ที่อยู่บนระบบปฏิบัติการวินโดวส์ (Windows) (ในทางเทคนิคจะขอเรียกว่า "WINS" โดยที่ทำงานแบบกระบวนการเดี่ยว) และอุปกรณ์จริงอย่างเช่น Nokia 3650 จะมีความแตกต่างกันอยู่เล็กน้อย ซึ่งความแตกต่างนี้ ทำให้ผู้พัฒนาโปรแกรมนั้นมีปัญหากับความไม่สัมพันธ์กันระหว่างอีมูเลเตอร์กับอุปกรณ์ที่ใช้งานจริงโดยเฉพาะสำหรับผู้พัฒนาโปรแกรม ที่เป็นโปรแกรมที่มีความซับซ้อนมากจะเป็นห่วงเกี่ยวกับความแตกต่างระหว่าง WINS กับอุปกรณ์จริง

กุญแจหลักสำคัญของแอพพลิเคชันเฟรมเวิร์คนี้คือ UIKON ซึ่ง UIKON ก็คือ มาตรฐานของเฟรมเวิร์คที่ทุกๆ อุปกรณ์ซึ่งทำงานอยู่บนระบบปฏิบัติการซิมเบียนต่างมีเหมือนกัน UIKON ไม่เพียงแต่จะมีเฟรมเวิร์ค ที่จัดเตรียมไว้สำหรับการทำงานของแอพพลิเคชันเท่านั้น แต่ยังมีส่วนสนับสนุนสำหรับ การทำงานของส่วนประกอบสำหรับการควบคุมทั่วๆไปอีกด้วย ยกตัวอย่างเช่น กล่องข้อความ (dialog box) ตัวแก้ไขตัวเลข (number editor) และตัวแก้ไขวันที่ (date editor) ซึ่งกลุ่มของแอพพลิเคชันจะสามารถใช้ประโยชน์ได้ขณะทำงาน

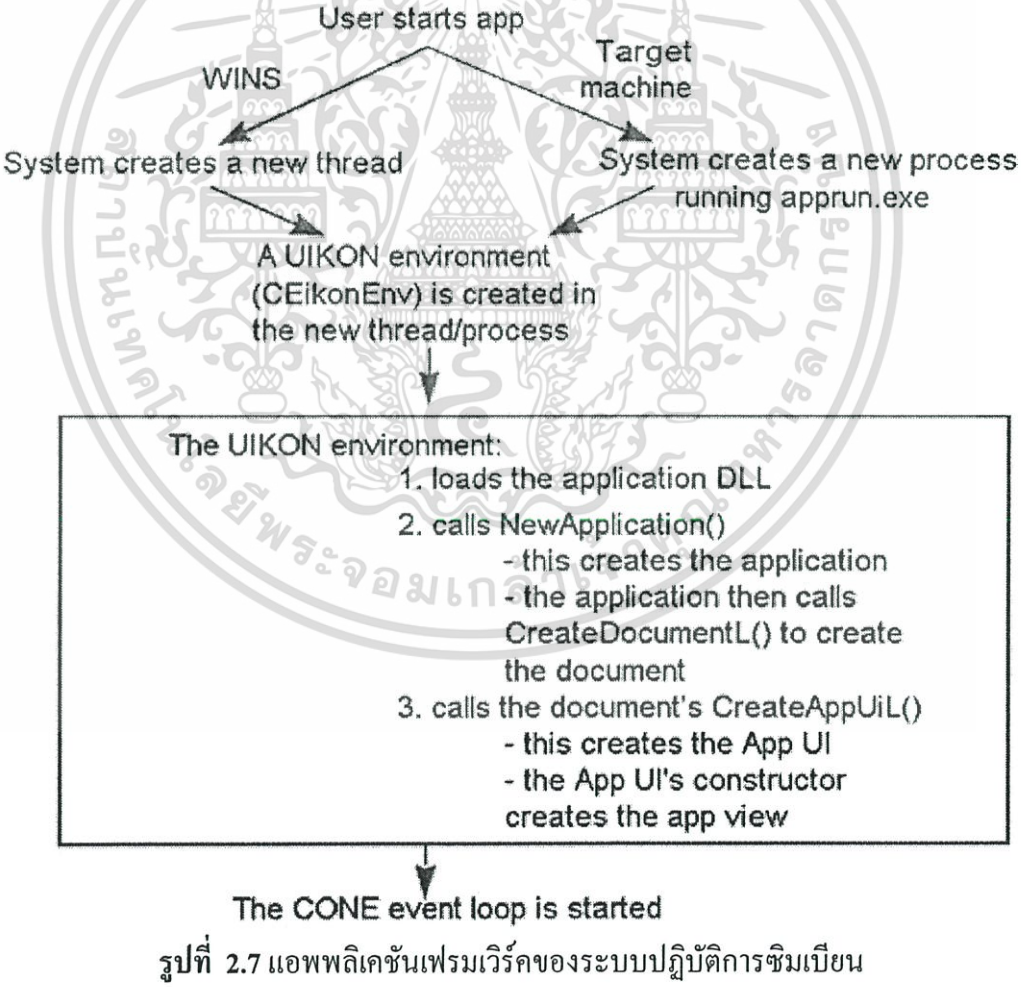
แอพพลิเคชันบนระบบระบบปฏิบัติการซิมเบียน ประกอบด้วย 4 ส่วนประกอบ ซึ่งแต่และส่วนมีความเกี่ยวข้องกับคลาส (Class) ใน UIKON เฟรมเวิร์ค ได้แก่

1. แอพพลิเคชันเชลล์ (Application shell) ในส่วนนี้ได้รับการถ่ายทอดคุณสมบัติมาจากคลาส CEikApplication และคลาสนี้จะถูกสร้างขึ้นตอนแรกสุดโดยเฟรมเวิร์ค ทันทีที่ถูกสร้างขึ้นมันจะมีหน้าที่รับผิดชอบต่อการกำหนดค่าเริ่มต้นต่างๆให้กับโค้ด (Code) ส่วนที่เหลือ

2. ดอคคิวเมนท์ (Document) ซึ่งได้รับการถ่ายทอดคุณสมบัติมาจากคลาส CEikDocument ส่วนนี้จะทำให้เข้าใจผิดได้ในกรณีที่ว่า ไม่ใช่ทุกแอพพลิเคชันจะต้องมีดอคคิวเมนท์ ที่จะต้องมาเกี่ยวข้องกับมัน เหมือนกับที่แอพพลิเคชันเหล่านี้จะต้องเกี่ยวข้องกับผู้ใช้ตัวอย่างเช่น โปรแกรม Notepad ที่เป็นโปรแกรมเกี่ยวข้องกับดอคคิวเมนท์ โดยตรงอย่างชัดเจน ในขณะที่โปรแกรม เช่น นาฬิกา นั้นจะไม่ถูกกำหนดความสามารถในด้านการจัดการดอคคิวเมนท์ทั้งการสร้าง การเปิดเอกสารและการแก้ไขดอคคิวเมนท์ใดๆ แต่ในความเป็นจริงแล้วทุกๆ แอพพลิเคชันจะต้องมีคลาส CEikDocument ที่เฟรมเวิร์คจะต้องการสร้างขึ้นมา

3. ส่วนติดต่อผู้ใช้งานของแอพพลิเคชัน (App UI) ซึ่งได้รับการถ่ายทอดคุณสมบัติมาจาก CEikAppUi ซึ่งคลาสนี้จัดเตรียมหน้าที่การทำงานหลักสำหรับทุกๆ แอพพลิเคชัน เช่น การรับมือกับเหตุการณ์ (event handing) การควบคุมเหตุการณ์ (event control) การสร้างการควบคุม การเข้าถึงการเรียกของระบบ (system call) ที่เป็นประโยชน์หลายตัว เป็นต้น

4. วิว (View) ในส่วนนี้คือส่วนที่ผู้ใช้สามารถที่จะมองเห็นได้จริง ในหน้าจอของอุปกรณ์ มันสามารถใช้งานสำหรับการแสดงข้อมูลอย่างง่าย หรือรวบรวมข้อมูลจากผู้ใช้ในแอพพลิเคชันที่ซับซ้อนมากขึ้น ตัวอย่างเช่น ในแอพพลิเคชันประเภทการแก้ไขข้อความ ตัวหนังสือที่ถูกพิมพ์นั้นก็คือการควบคุมมาตรฐานซึ่งถูกจัดไว้โดย UIKON โดยจัดเก็บอยู่ในส่วนของวิว ในขณะที่ทุกแอพพลิเคชันจะมีอยู่หนึ่งวิวโดยปกติ แต่แอพพลิเคชันที่มีความซับซ้อนมากสามารถมีได้หลายวิว




**รูปที่ 2.7** แอพพลิเคชันเฟรมเวิร์คของระบบปฏิบัติการซิมเบียน

## 2.5 หลักการการปรับปรุงประสิทธิภาพของARM Compiler

คอมไพเลอร์ ARM สามารถประสิทธิภาพในด้านความเร็วได้ระดับสูง โดยเฉพาะขนาดของโค้ดจะมีขนาดเล็ก และหลักการปรับปรุงประสิทธิภาพ หลักๆของคอมไพเลอร์นี้คือ

### 2.5.1. Common Subexpression Elimination (CSE)

เป็นการระบุ Expression ย่อยภายในโค้ดและใช้ผลลัพธ์ของ Expression ย่อยที่ได้กับทุกส่วนดีกว่าการคำนวณใหม่ทุกครั้ง ยกตัวอย่างเช่น ในโค้ดอาจมีการใช้ Expression a+1 ในหลายที่ต้องคำนวณค่าทุกครั้งที่ใช้ คอมไพเลอร์จะระบุ Expression ย่อยนี้และจะทำการคำนวณเพียงครั้งเดียว และใช้นำมันไปใช้หลายครั้งโดยตัวของ Expression อาจจะซับซ้อนมาก ดังนั้นนี้เป็นการปรับปรุงที่มีประสิทธิภาพมากของคอมไพเลอร์

### 2.5.2 Loop invariant Motion (Expression Lifting)

เป็นการยก Expression ออกจากลูป คอมไพเลอร์จะระบุ Expression ในลูปซึ่งไม่มีการเปลี่ยนแปลงขณะลูปกำลังทำงาน การที่เราคำนวณ Expression ใหม่ทุกครั้งทำให้เกิดกระบวนการที่สูญเปล่าดังนั้นคอมไพเลอร์จะยก Expression ออกจากลูปทำให้คำนวณมันเพียงครั้งเดียว

### 2.5.3 Live range splitting (for dynamic register allocation)

เป็นการระบุสถานะปัจจุบันของตัวแปรใดๆภายใต้โปรแกรม ตัวอย่างเช่นตัวแปร ที่ใช้ในสถานการณ์หนึ่ง เช่นเป็นตัวนับของลูป จากนั้นตัวแปรถูกใช้ในการคำนวณ ถ้าทั้งสองไม่มีความเกี่ยวข้องกัน เราสามารถจองพื้นที่ใน register ที่ต่างกันและเมื่อตัวแปรไม่ใช้งาน (เมื่อค่าของมันไม่ถูกใช้แล้วภายหลัง) register ที่เก็บค่านั้นก็จะถูกนำไปใช้ทำงานอย่างอื่น

### 2.5.4 Constant Folding

ทำการแทนที่ Expression ที่ไม่มีการเปลี่ยนแปลงด้วยค่าที่คอมไพเลอร์คำนวณจาก Expression เหล่านั้น

### 2.5.5 Tail Call Optimization and Tail Recursion

Tailcall เป็นการคืนค่าฟังก์ชันที่เกิดขึ้นทันทีหลังจากการคืนค่า โดยปกติแล้วฟังก์ชันจะถูกเรียก และเมื่อมันคืนค่ากลับไปหา ผู้เรียกแล้วจากนั้น ผู้เรียกจะทำการคืนค่าอีกที โดย Tailcall optimization จะหลีกเลี่ยงเหตุการณ์นี้โดยการเก็บค่าใน registers ก่อนที่จะกระโดดไปที่ tailcall ทำให้ฟังก์ชัน ที่ถูกเรียกจะคืนตรงไปที่ผู้เรียกเลย คอมไพเลอร์สนับสนุน tail call recursion ซึ่งเป็นไปได้เมื่อ tailcall ทำให้ ฟังก์ชันเดียวกัน ในกรณีนี้มันเป็นไปได้ที่จะ ข้ามทั้งลำดับการเข้าและการออก แปลงการเรียกใช้ในลูป

**2.5.6 Cross Jump Elimination**

เป็นการรวมกันตั้งแต่สองส่วนของโค้ดที่เหมือนกัน ตัวอย่างเช่น ในหลายกรณี การคืนค่าจากฟังก์ชันจะเกิดจากโค้ดที่บ่อยครั้งจะเหมือนๆกัน และจะปรับปรุงให้คืนค่าเพียงหนึ่งเดียวโดยการปรับปรุงนี้ทำให้ประหยัดขนาดจองโค้ด

**2.5.7 Table Driven Peepholing**

ระหว่างการทำการคอมไพเลอร์โค้ดไปเป็น ARM หรือ Thumb โค้ดจะมีจุดที่พบว่าโค้ดจะถูกแทนที่ด้วยโค้ดที่เป็นโค้ดปรับปรุง โดยทำจากการมองโค้ดผ่าน window ที่เรียกว่า peephole และแทนที่คำสั่งด้วยคำสั่งที่เขียนขึ้นด้วยมือจากตารางของ peephole มีการเพิ่มเติมปรับปรุงโค้ดเข้าไปเพิ่มด้วย ซึ่งทำการเพิ่มโดยวิศวกรของ ARM

**2.5.8 Structure Splitting**

Structure splitting เป็นวิธีการแบ่งโครงสร้างเป็นส่วนๆ โดยวิธีการนี้ แต่ละส่วนอาจถูกกำหนดให้ register ทำให้เพื่อให้เข้าถึงได้เร็วขึ้น มันจะมีประโยชน์เมื่อคืน ตัวโครงสร้างจากฟังก์ชัน ตัวโครงสร้างสามารถคืนค่าใน register ดีกว่าบนสแต็ค

**2.5.9 Conditional Execution (or Branch Elimination)**

อาร์มคอมไพเลอร์ ใช้ Conditional execution เพื่อหลีกเลี่ยง Branch Conditional Execution ซึ่งช่วยให้ประหยัดทั้งพื้นที่และเวลาทำงาน เพราะลดหลาย Conditional execution

## 2.6 โครงสร้างที่ใช้ในการเก็บข้อมูล

จาก[16] ซึ่งเป็นที่มาของโค้ดที่ใช้ในการทดลองนี้ซึ่งเป็นโค้ดที่ใช้ในงานวิจัยการเข้ารหัสแบบอาร์เอสเอ-129 มีลักษณะการเก็บข้อมูลดังนี้

ข้อมูลจะเก็บอยู่ใน ARRAY ของตัวแปร ชนิด long ซึ่งตามภาษา C/C++ จะมีขนาด 4 ไบต์ซึ่งจะเก็บข้อมูลได้ ระหว่าง -2,147,483,648 จนถึง +2,147,483,647 ต่อ ตัวแปรชนิด long 1 ตัว แต่ในโค้ดนี้จะเก็บเป็น ARRAY โดยหลักแรก จะเก็บข้อมูลขนาดของ ARRAY และตำแหน่งถัดๆ ไปเป็นค่าที่จะต้องไปคูณกับ ค่าสูงสุดที่เก็บได้ในแต่ละช่องซึ่งเท่ากัน ยกกำลังกับตำแหน่ง ARRAY นั้น – 1 ซึ่งจะทำให้สามารถเก็บข้อมูลได้สูงถึง

2,147,483,648

$$\sum_{N=1} \text{ARRAY}[N] \times 2{,}147{,}483{,}648^{N-1}$$

ซึ่งก่อนจะใช้งาน ARRAY ก็จำเป็นที่จะต้องจองหน่วยความจำให้กับตัวแปร ARRAY นี้

# บทที่ 3
# วิธีการปรับปรุงประสิทธิภาพเพื่อความเร็วและลดการใช้พลังงาน

## 3.1 วิธีการเพิ่มความเร็ว

ในส่วนนี้จะเสนอเทคนิคต่างๆที่ใช้ในการปรับปรุงประสิทธิภาพเพื่อเพิ่มความเร็ว เช่น การทำกำจัดความซ้ำซ้อนในโค้ด หรือการลดการเรียกฟังก์ชัน ซึ่งเป็นการปรับปรุงประสิทธิภาพทางด้านความเร็วและช่วยทำให้ไปป์ไลน์ทำงานได้ดีขึ้น ซึ่งทำให้ได้ความเร็วในการประมวลผลที่เร็วขึ้น ซึ่งเทคนิคต่างมีดังนี้

### 3.1.1 การทำกำจัดความซ้ำซ้อนในโค้ด

กระบวนการนี้คือการย้ายการทำงานของโค้ดส่วนที่ซ้ำซ้อนที่ไม่จำเป็นต้องทำหรือทำงานสูญเปล่าในตัวอย่าง รูปที่ 3.1(a) จะเห็นว่าในฟังก์ชันมีการตรวจสอบว่ามีการกำหนดค่าให้กับตัวแปร log10rad หรือไม่ถ้ายังไม่มี จะทำการคำนวณค่าให้กับตัวแปรนี้ซึ่งหากเรามีการเรียกฟังก์ชันนี้ จะมีการตรวจสอบและกำหนดค่าให้ทุกครั้ง ซึ่งไม่มีความจำเป็นที่จะทำเช่นนี้ทุกครั้งที่มีการเรียกฟังก์ชัน เราจึงทำการย้ายการกำหนดค่าเริ่มต้นให้กับตัวแปรนี้ไปไว้ที่ Constructor ของคลาสนั้นดังรูปที่ 3.1 (b) ซึ่งจะช่วยลดในเรื่องของ Branch prediction อีกด้วย

```
long LongInteger::zsread(char *str,long **outputVerylong)
{
    static char *inmem = 0;
    char *in;
    register long d = 0;
    register long anegative = 0;
    register long return_value = 1;
    long *a = *outputVerylong;
    long *digit = &glosho[1];

    if (log10rad < 0)
        log10rad = log((double)RADIX) / log((double)10);

    if (!inmem)
        inmem = (char *)calloc((size_t)IN_LINE, sizeof(char));
```

(a)

**รูปที่ 3.1** (a) ส่วนของโค้ดจากเข้ารหัสแบบอาร์เอสเอส่วนโค้ดที่มีการทำงานซ้ำซ้อน

```
LongInteger::LongInteger(void)
{
    log10rad = log((double)RADIX) / log((double)10);

long LongInteger::zsread(char *str,long **outputVerylong)
{
    static char *inmem = 0;
    char *in;
    register long d = 0;
    register long anegative = 0;
    register long return_value = 1;
    long *a = *outputVerylong;
    long *digit = &glosho[1];

    if (!inmem)
        inmem = (char *)calloc((size_t)IN_LINE, sizeof(char));
```

(b)

**รูปที่ 3.1** (b) ส่วนของโค้ดจากเข้ารหัสแบบอาร์เอสเอที่ทำการกำจัดส่วนที่ซ้ำซ้อน

### 3.1.2 การทำการลดจำนวนของ branch prediction

การลดจำนวนของ Branch prediction โดยการทำ Control flow transformation ซึ่งเป็นสิ่งที่สำคัญมากในการทำงานของระบบไปป์ไลน์นั้นคือถ้าเราทำการลดจำนวนของBranch prediction ก็จะทำให้ความเร็วในการทำงานเพิ่มขึ้นด้วย

จากรูปที่ 3.1(b) จะเห็นว่าตัวแปร inmem มีการประกาศค่าให้เป็นตัวแปรแบบสแตติค และมีการตรวจสอบว่ามีการกำหนดค่าในหน่วยความจำให้หรือไม่ถ้ายังไม่มีจะมีการกำหนดค่าให้ ซึ่งเราจะสามารถลด Branch prediction ตรงนี้ลงไปได้โดยการย้าย inmem ไปเป็น Attribute ของคลาส และทำการจองหน่วยความจำให้กับตัวแปร inmem ที่ Constructor ของคลาส ดังรูปที่ 3.2

```
class LongInteger
{
    char *inmem;

LongInteger::LongInteger(void)
{
    log10rad = log((double)RADIX) / log((double)10);
    inmem = (char *)calloc((size_t)IN_LINE, sizeof(char));

long LongInteger::zsread(char *str,long **outputVerylong)
{
    char *in;
    register long d = 0;
    register long anegative = 0;
    register long return_value = 1;
    long *a = *outputVerylong;
    long *digit = &glosho[1];
```

**รูปที่ 3.2** ส่วนของโค้ดจากเข้ารหัส แบบ อาร์เอสเอ ที่มีการลด Branch prediction

### 3.1.3 การทำการลดจำนวนการเรียกใช้ฟังก์ชัน

ในการเรียกใช้ฟังก์ชันในแต่ละครั้ง จะมีการสูญเสียอันเกิดจากการเรียกใช้ฟังก์ชัน โดยการเก็บและดึงค่าในสแต็ค เพื่อเก็บค่าต่างๆก่อนและหลังการเรียกใช้ฟังก์ชัน โดยเราสามารถลดจำนวนการเรียกใช้ฟังก์ชันได้จากการนำฟังก์ชันนั้นนำไปเปลี่ยนเป็น Macro ซึ่งเมื่อคอมไพล์เลอร์คอมไพล์โปรแกรมแล้ว จะทำให้ได้โปรแกรมที่มีขนาดใหญ่ขึ้น เนื่องจากคอมไพล์เลอร์จะนำโค้ดส่วนที่เป็น Macro มาแทนที่ทุกจุดที่มีการเรียกใช้งานฟังก์ชันนั้น โปรแกรมที่ได้จึงมีขนาดใหญ่ขึ้นดังตัวอย่างจะเป็นรูปฟังก์ชัน zaddmulone ซึ่งฟังก์ชันนี้ถูกเรียกใช้งานบ่อยและไม่มีการเรียกฟังก์ชันอื่นๆ ใช้งานอีกภายใต้ฟังก์ชันนี้ จึงเหมาะสมที่จะนำมาเปลี่ยนเป็น Macro ดังรูปที่ 3.3

```
void zaddmulone(long *ama,long *amb)
{
    register long lami;
    register long lams = 0;
    register long *lama = (ama);
    register long *lamb = (amb);

    lams = 0;
    for (lami = (*lamb++); lami > 0; lami--)
    {
        lams += (*lama + *lamb++);
        *lama++ = lams & RADIXM;
        lams >>= NBITS;
    }
    *lama += lams;
}
```

(a)

```
#define zaddmulone(ama, amb)
{
    register long lami;
    register long lams = 0;
    register long *lama = (ama);
    register long *lamb = (amb);

    lams = 0;
    for (lami = (*lamb++); lami > 0; lami--)
    {
        lams += (*lama + *lamb++);
        *lama++ = lams & RADIXM;
        lams >>= NBITS;
    }
    *lama += lams;
}
```

(b)

**รูปที่ 3.3** ส่วนของโค้ดจากเข้ารหัสแบบอาร์เอสเอ (a) ส่วนโค้ดที่มีฟังก์ชัน zaddmulone (b) ที่ทำการเปลี่ยนฟังก์ชัน zaddmulone ไปเป็น Macro

### 3.1.4 การลดจำนวนตัวแปร สำหรับการเรียกฟังก์ชัน

จาก [1], [15] ทำให้ทราบว่าคอมไพเลอร์ที่ใช้ในการคอมไพล์โค้ด C/C++ ที่เราใช้นั้นคือ อาร์มคอมไพล์เลอร์ เมื่อทำการคอมไพล์โค้ดแล้วจะได้โปรแกรมที่เมื่อมีการเรียกใช้งานฟังก์ชันนั้นจะใช้ Register 4 ตัวสำหรับการส่งค่าตัวแปรต่างๆไปให้กับฟังก์ชันนั้นๆ หากว่าเรามีตัวแปรที่ต้องการส่งค่าไปให้กับฟังก์ชันนั้นมากกว่า 4 ตัวโดยโปรแกรมจะต้องทำการเก็บค่าลงไว้ใน สแต็คแล้วจึงให้โปรแกรมเรียกใช้ผ่านสแต็คอีกทีหนึ่ง ดังนั้นเราจึงได้ทำการปรับจำนวนตัวแปร ที่จะส่งค่าให้กับฟังก์ชัน โดยให้มีจำนวนไม่เกิน 4 ตัวแปรเพื่อที่จะไม่ต้องเก็บค่าตัวแปรไว้ที่สแต็คซึ่งจะทำให้ต้องใช้งานหน่วยความจำเพิ่มขึ้นและใช้จำนวนคำสั่ง เพิ่มมากขึ้นอีกด้วย

## 3.2 วิธีการลดการใช้พลังงาน

ในการประมวลผลข้อมูลต่างๆ อุปกรณ์ที่ใช้พลังงานเยอะที่สุดก็คือหน่วยความจำ ทำให้วิธีปรับปรุงเพื่อลดพลังงานที่เราใช้จะมุ่งไปการลดจำนวนการเกิดการค้นหาข้อมูลไม่พบในแคช อันเนื่องมาจากการอ้างหน่วยความจำโดยทำปรับปรุงในภาษาระดับสูง

### 3.2.1 ทฤษฎีการลดการใช้พลังงาน

จาก[17] สมการของการใช้พลังงานในหน่วยความจำจะได้ว่า

**Energy = Hit Rate x Energy_hit + Miss Rate x Energy_miss**

Energy_hit = E_dec+E_cell

Energy_miss = E_dec+E_cell+E_io+E_main

= Energy_hit + E_io+E_main

Energy = Hit Rate x Energy_hit + Miss Rate x (Energy_hit + E_io+E_main)

เนื่องมาจาก ARM920T มีแคชขนาด 16 กิโลไบต์และ มีนโยบายแบบ Write Through และ Write Back ดังนั้นเมื่อไรก็ตามเมื่อมีการอ้างถึงหน่วยความจำ โปรเซสเซอร์จะทำการค้นหาในหน่วยความจำแคชก่อน ถ้าไม่พบจึงจะทำการอ้างถึงหน่วยความจำหลักอีกทีหนึ่ง ดังนั้นหากเราสามารถทำให้หน่วยความจำที่เราต้องการใช้งาน อยู่ภายในแคชให้มากที่สุดก็จะสามารถลดอัตราการเกิดการค้นหาข้อมูลไม่พบในแคช ทำให้การใช้พลังงานที่ใช้ในการเข้าใช้หน่วยความจำลดลง

### 3.2.2 วิธีการปรับปรุงเพื่อลดการใช้พลังงาน

กรณีการเขียนข้อมูลแล้วไม่เจอข้อมูลในแคช (Write miss) โปรเซสเซอร์ ARM920T จะใช้นโยบาย Write thought และ Non-allocate ส่วนการเขียนข้อมูลที่มีในแคช (Write hit) ในโปรเซสเซอร์นี้จะใช้ Write back และเขียนใน Write back บัฟเฟอร์ วิธีการปรับปรุงของเรา จะเน้นการลดการเกิดการค้นหาข้อมูลไม่พบในแคช ไม่ว่าจะเพื่อค้นหาข้อมูลในหน่วยความจำ หรือค้นหาพื้นที่ว่างเพื่อจองหน่วยความจำ โดยจะปรับปรุงเปลี่ยนแปลงโปรแกรมเพื่อเอื้อประโยชน์ต่อการใช้แคชมากขึ้น โดยผลที่ออกมาจะมีจำนวนการค้นหาข้อมูลไม่พบในแคชน้อยลงซึ่งก็คือลดเหตุการณ์การใช้หน่วยความจำหลักใน อาร์มโปรเซสเซอร์

**3.2.2.1 วิธี Minimizing Buffer Allocation** ในการปรับปรุงโค้ดที่เราใช้เป็นภาษา C++ ซึ่งในฟังก์ชัน จะมีบ่อยๆที่ใช้บัฟเฟอร์ในการเก็บผลก่อนในระหว่างการคำนวณ จากที่สังเกตดูบัฟเฟอร์พวกนี้จะจองหน่วยความจำแบบ local แล้ว assign(written) ต่อมาจะเอาค่าไปใช้ (read) ท้ายสุดทำลายทิ้ง (deallocate) และเมื่อฟังก์ชันถูกเรียกทำงาน ตำแหน่งของหน่วยความจำจะถูกจองในตำแหน่งที่แตกต่างกันในแต่ละครั้ง ซึ่งคือปัญหาที่ยากในการจัดการกับแคชข้อมูลที่มีขนาดจำกัด วิธีของเราในการแก้ปัญหานี้โดยการประกาศบัฟเฟอร์ข้อมูลเป็น Attribute ของคลาส ทำให้สามารถเข้าสโคปที่ใหญ่ขึ้น โดยผลที่ได้จากที่บัฟเฟอร์ข้อมูลถูกจองเพียงครั้งเดียวแล้วใช้มันอีกเรื่อยๆเป็นผลให้ การค้นหาข้อมูลไม่พบในแคชลดลง

```
void LongInteger::zmulin(long *b,long **a,char *log)
{
    long *mem = 0;
    if (ALLOCATE && (!*a || !b))
    {
        zzero(a,log);
        return;
    }
    zcopy(*a, &mem,"set mem from zmulin");
    zmul(mem, b, a,log);
    zfree(&mem,"fr  mem from zmulin");
}
```

(a)

**รูปที่ 3.4** (a) ส่วนของโค้ดจากเข้ารหัสแบบอาร์เอสเอโค้ดเดิม

```
class LongInteger
{
    long * mem;

LongInteger::LongInteger(void)
{
    mem = 0;


void LongInteger::zmulin(long *b,long **a,char *log)
{

    if (ALLOCATE && (!*a || !b))
    {
        zzero(a,log);
        return;
    }
    zcopy(*a, &mem,"set mem from zmulin");
    zmul(mem, b, a,log);

}

LongInteger::~LongInteger(void)
{
    zfree(&mem,"fr  mem from zmulin");
```

(b)

**รูปที่ 3.4** (b) ส่วนของโค้ดจากเข้ารหัสแบบอาร์เอสเอที่ได้ทำการMinimizing Buffer Allocation

ซึ่งจากเหตุผลข้างต้นนี้ เราจึงได้ทำการตรวจสอบการจองหน่วยความจำทั้งหมดที่ใช้ในการทำงานในการเข้ารหัสแบบอาร์เอสเอ โดยในแต่ละครั้งที่มีการจองหน่วยความจำ ปรับปรุงการจองหน่วยความจำ หรือการคืนหน่วยความจำ โดยจะบันทึกไว้ว่าทำงานมาจากฟังก์ชัน ใดใครเป็นคนจอง หรือปรับปรุงการจองและถูกคืนเมื่อใด และถูกใช้งานผ่านตัวแปรใด จากนั้นจะนำมาประมวลผลโดยใช้โปรแกรมที่เราเขียนขึ้น เพื่อทำการตรวจสอบลำดับการจองหน่วยความจำ และการคืนหน่วยความจำ เพื่อทำการลดการจองหน่วยความจำ โดยให้ไปใช้งานหน่วยความจำที่ได้ทำการจองหน่วยความจำไว้แล้วและไม่ใช้งานแล้ว เพื่อให้เกิดความต่อเนื่องในการใช้งานหน่วยความจำในแคชและลดปัญหาการเกิดการค้นหาข้อมูลไม่พบในแคช

**3.2.2.2 วิธี Reused Variable** เมื่อเรามีการสร้างตัวแปรและมีการจองหน่วยความจำให้กับตัวแปรจะมีการจองพื้นที่ในหน่วยความจำให้กับตัวแปรนั้นๆ ซึ่งจะทำให้เกิดการเปลี่ยนแปลงข้อมูลขึ้นในแคช ซึ่งหากว่าเราสามารถเปลี่ยนการจองหน่วยความจำให้กับตัวแปรหลายๆตัว มาใช้ตัวใดตัวหนึ่งร่วมกันได้ ก็จะช่วยทำให้โอกาสที่จะค้นหาข้อมูลไม่พบในแคชนั้นลดลงได้เนื่องจากหน่วยความจำที่อ้างอิงถึงมีโอกาสที่จะอยู่ในแคชอยู่แล้วเพิ่มมากขึ้นอีกด้วยเช่นกัน

เนื่องจากโค้ดของการเข้ารหัสแบบอาร์เอสเอ มีการทำงานที่ซับซ้อนและมีการเรียกใช้งานหน่วยความจำเพื่อใช้ในการทำงานในส่วนต่างๆหลายจุดจึงเป็นการยากที่เราจะสามารถเข้าใจการทำงานได้ทั้งหมด เราจึงจำเป็นที่จะต้องเขียนโปรแกรมเข้าช่วยเพื่อใช้ในการตรวจสอบจำนวนการจองหน่วยความจำ ลำดับการทำงาน, การคืนหน่วยความจำ, การปรับปรุงการจองหน่วยความจำ, จองจากฟังก์ชันไหน จองด้วยตัวแปรใดจองเป็นพื้นที่เท่าไร เพื่อที่จะได้ใช้ในการตรวจสอบเพื่อหาทางลดการใช้งานตัวแปรต่อไป

| Event No. | Type | Variable | Method | length |
|---|---|---|---|---|
| 2018 | Allocate | mem | zsubmul | 20 |
| 2019 | Allocate | a | zsubmul | 20 |
| 2020 | Free | a | zsubmul | |
| 2021 | Allocate | c | zsubmul | 20 |
| 2022 | Free | mem | zsubmul | |

**ตารางที่ 3.1** ตัวอย่างของข้อมูลการใช้งานหน่วยความจำที่ โปรแกรมที่เราเขียนขึ้นบันทึกไว้

### 3.3 ขั้นตอนการปรับปรุงประสิทธิภาพการเข้ารหัสแบบอาร์เอสเอ และข้อดีข้อเสียที่ได้จากการทดลองของวิธีการแต่ละแบบ

3.3.1 ทำโปรไฟล์(Profile) โค้ดโปรแกรมของการเข้ารหัสแบบอาร์เอสเอโดยใช้ Microsoft Visual C++ เพื่อให้ทราบถึงส่วนต่างๆว่ามีการทำงานเป็นเช่นไร ใช้เวลาในการทำงานในส่วนนั้นๆ เป็น เช่นไรเพื่อจะได้เริ่มตรวจสอบการทำงานในจุดที่มีการทำงานมาก ก่อนโดยจากการทำโปรไฟล์สามารถให้ข้อมูลรายละเอียดดังตารางที่ 3.2 ดังนี้

| Function Time | % | Hit Count | Function Name |
|---|---|---|---|
| 1234.987 | 58.7 | 1 | _main |
| 478.677 | 22.8 | 1730387 | zaddmulp(long *,long,long,long *) |
| 211.825 | 10.1 | 35837 | LongInteger::zsubmul(long,long *,long *) |
| 96.109 | 4.6 | 53716 | zaddmulsq(long,long *,long *) |
| 19.842 | 0.9 | 4067 | LongInteger::kar_sq(long *,long * *,long,char *) |
| 15.497 | 0.7 | 1557 | LongInteger::zmod(long *,long *,long * *,char *) |
| 12.583 | 0.6 | 24832 | LongInteger::zsetlength(long * *,long,char *,char *) |
| 9.943 | 0.5 | 18063 | zaddmulone(long *,long *) |
| 4.785 | 0.2 | 1150 | Zaddmul(long,long *,long *) |
| 4.224 | 0.2 | 6242 | LongInteger::zfree(long * *,char *) |
| 2.549 | 0.1 | 4139 | LongInteger::zcopy(long *,long * *,char *) |
| 2.229 | 0.1 | 2030 | LongInteger::zsubpos(long *,long *,long * *,char *) |
| 1.856 | 0.1 | 2535 | zaddls(long *,long *,long *) |
| 1.775 | 0.1 | 2653 | LongInteger::zadd(long *,long *,long * *,char *) |
| 1.105 | 0.1 | 1022 | LongInteger::zsqmod(long *,long *,long * *,char *) |
| 0.815 | 0.0 | 1022 | LongInteger::zlshift(long *,long,long * *,char *) |
| 0.641 | 0.0 | 1023 | LongInteger::zlowbits(long *,long,long * *,char *) |
| 0.619 | 0.0 | 1023 | LongInteger::z2div(long *,long * *,char *) |
| 0.541 | 0.0 | 534 | LongInteger::zmulmod(long *,long *,long *,long * *,char *) |
| 0.492 | 0.0 | 1022 | LongInteger::zsq(long *,long * *,char *) |
| 0.479 | 0.0 | 534 | LongInteger::kar_mul(long *,long *,long * *,long,char *) |
| 0.375 | 0.0 | 1 | LongInteger::verylong_to_bits(char *,long *,long *) |
| 0.364 | 0.0 | 1 | LongInteger::zexpmod(long *,long *,long *,long * *,char *) |
| 0.360 | 0.0 | 616 | LongInteger::zsmul(long *,long,long * *,char *) |
| 0.304 | 0.0 | 1023 | LongInteger::zrshift(long *,long,long * *,char *) |
| 0.277 | 0.0 | 534 | LongInteger::zmul(long *,long *,long * *,char *) |
| 0.263 | 0.0 | 1024 | set_bit(char *,int,long) |
| 0.251 | 0.0 | 1023 | LongInteger::ziszero(long *) |
| 0.145 | 0.0 | 1 | LongInteger::RSA_encryption(char *,char *,long,long *,long *,long *,long *) |
| 0.136 | 0.0 | 2 | LongInteger::zsread(char *,long * *) |
| 0.006 | 0.0 | 7 | LongInteger::z2mul(long *,long * *,char *) |
| 0.006 | 0.0 | 1 | _strrev |
| 0.003 | 0.0 | 8 | LongInteger::zintoz(long,long * *,char *) |
| 0.003 | 0.0 | 7 | LongInteger::zsadd(long *,long,long * *,char *) |
| 0.003 | 0.0 | 2 | LongInteger::LongInteger(void) |
| 0.002 | 0.0 | 8 | get_bit(char *,long) |
| 0.002 | 0.0 | 1 | LongInteger::bits_to_verylong(char *,long,long * *,char *) |
| 0.001 | 0.0 | 2 | LongInteger::z2log(long *) |
| 0.001 | 0.0 | 2 | LongInteger::z2logs(long) |
| 0.001 | 0.0 | 1 | LongInteger::zstart(void) |

**ตารางที่ 3.2** โปรไฟล์การทำงานของโค้ดโปรแกรมของการเข้ารหัสแบบ อาร์เอสเอ

3.3.2 ทำการตรวจสอบโค้ดโปรแกรม ตามลำดับฟังก์ชันที่ได้ทำโปรไฟล์

ทำการตรวจสอบโค้ดโปรแกรม ตามลำดับฟังก์ชันที่ได้ทำโปรไฟล์ไว้ตามลำดับเวลาที่ใช้ในการทำงานเพราะหากเราสามารถปรับฟังก์ชันที่ใช้เวลาทำงานมากหรือบ่อยให้ดีขึ้นได้ก็จะได้ประสิทธิภาพที่ดีมากขึ้น

3.3.2.1 การทำการลดจำนวนฟังก์ชันที่เรียกใช้

จากการตรวจสอบพบว่าฟังก์ชันหลายฟังก์ชันเป็นฟังก์ชันที่สามารถนำมาเปลี่ยน เป็น Macro ได้เนื่องจากเป็นฟังก์ชัน ที่ไม่มีการเรียกใช้งานฟังก์ชันอื่น และมีการทำงานที่เสร็จสิ้นภายในตัวฟังก์ชันนั้นๆ เอง

ฟังก์ชัน ที่พบ

zaddmulsq(long,long *,long *)

zaddmulone(long *,long *)

zaddmul(long,long *,long *)

จึงได้ทำการเปลี่ยนฟังก์ชันทั้ง 3 ให้เป็น Macro เมื่อทำการคอมไพล์โปรแกรม จะทำให้โปรแกรมมีขนาดใหญ่ขึ้น เพราะคอมไพล์เลอร์จะนำโค้ดที่ได้นิยามไว้เป็น Macro ไปแทนที่ทุกส่วนที่ทำการเรียกใช้ ทำให้จะได้โปรแกรมที่เมื่อคอมไพล์โปรแกรมแล้วมีขนาดใหญ่ขึ้น แต่จะมีความเร็วสูงขึ้น เนื่องจากไม่จำเป็นที่จะต้องทำการเรียกใช้งานฟังก์ชัน ซึ่งจะต้องมีกระบวนการทำงานทางด้านหน่วยความจำเพิ่มขึ้นอีกด้วย จากการปรับปรุงด้วยวิธีนี้พบว่ามีข้อดีข้อเสียคือ

ข้อดี โปรแกรมสามารถทำงานได้เร็วขึ้นโดยจากการทดลองเปรียบเทียบการทำงานระหว่างโปรแกรมที่ทำเป็น Macro และโปรแกรมที่เรียกฟังก์ชันธรรมดาโดยใช้ Armulator พบว่าโปรแกรมที่ใช้ Macro นั้น Clock Cycle ลดลงดังนี้

| **Optimize LV.** | **LV. 0** | **Macro + LV. 0** | **Speed Up %** | **LV. 2** | **Macro + LV. 2** | **Speed Up %** |
|---|---|---|---|---|---|---|
| Clock Cycle | 114,257,379 | 104,248,168 | 8.76 | 81,288,697 | 76,412,188 | 5.999 |

**ตารางที่ 3.3 จำนวน Clock Cycle ที่ใช้ในการทำงาน หลังปรับปรุงโดยลดการเรียกฟังก์ชัน**

ข้อเสีย โปรแกรมที่ได้มีขนาดใหญ่ขึ้นเมื่อเปรียบเทียบกับโปรแกรมที่เรียกใช้ฟังก์ชันตามปกติ โดยโปรแกรมจากเดิมที่มีการเรียกใช้ฟังก์ชันมีขนาด 160 กิโลไบต์ แต่หลังจากทำเป็น Macro จะมีขนาดเพิ่มขึ้นเป็น 210 กิโลไบต์ และหากจำนวนการเรียกใช้ฟังก์ชันที่ถูกเปลี่ยนเป็น Macro มีเพิ่มมากขึ้นขนาดของโปรแกรมก็จะเพิ่มมากขึ้นตามไปด้วย แต่จะทำให้โปรแกรมทำงานได้เร็วขึ้นด้วยเช่นกัน

3.3.2.2 การกำจัดความซ้ำซ้อนในโค้ดและลดจำนวนของ Branch prediction

มีหลายจุดในหลายฟังก์ชัน ที่มีการตรวจสอบตัวแปรชนิด Verylong ว่ามีการจองพื้นที่หน่วยความจำและกำหนดค่าให้หรือไม่ ถ้ายังไม่มีจะทำการจองพื้นที่ในหน่วยความจำให้ และทำการกำหนดค่า 0 ให้กับตัวแปรในจุดนี้ เมื่อเริ่มมีการสร้างตัวแปรชนิด Verylong เมื่อใดก็ตาม เราสามารถที่จะจองพื้นที่ในหน่วยความจำและกำหนดค่าให้ก่อน เมื่อมีการสร้างตัวแปลตัวนั้นซึ่งจะทำให้ไม่จำเป็นที่จะต้องทำการตรวจสอบว่ามีการจองพื้นที่ในหน่วยความจำให้หรือไม่ สามารถลดความซ้ำซ้อนของการทำงานและลดการเกิด Branch prediction อันเนื่องมาจากการต้องตรวจสอบการจองพื้นที่หน่วยความจำและกำหนดค่าให้ ในกรณีที่ยังไม่เคยมีการจอง โดยจากการตรวจสอบพบว่ามีหลายจุดดังนี้

- การตรวจสอบว่ามีการจองหน่วยความจำให้กับตัวแปรชนิด Verylong

ตัวอย่างโค้ด

ก่อนปรับปรุง if(ALLOCATE && !a) หลังปรับปรุง if(!a)

ฟังก์ชัน ที่พบ

void LongInteger::zcopy

void LongInteger::zadd

long LongInteger::zcompare

void LongInteger::zsubpos

void LongInteger::zsmul

void LongInteger::zmulin

void LongInteger::zmul

void LongInteger::zsq

void LongInteger::zlshift

void LongInteger::zrshift

void LongInteger::z2mul

long LongInteger::zsdiv

เราสามารถตัดข้อแม้การตรวจสอบว่ามีการจองหน่วยความจำทิ้งได้ เพื่อลดข้อแม้ในการทำงานอันเนื่องมาจากตัวแปรชนิด Verylong นั้นจากการปรับปรุงโค้ด ในทุกครั้งที่มีการประกาศตัวแปรชนิดนี้ขึ้นมา จะทำการจองหน่วยความจำให้กับตัวแปรชนิดนี้ทันที จึงไม่จำเป็นที่จะต้องทำการตรวจสอบว่ามีการจองหน่วยความจำให้กับตัวแปรนี้หรือไม่

- การคำนวณค่าซ้ำทุกครั้งและผลลัพธ์ที่ได้ ได้ค่าเดิมทุกครั้ง

ตัวอย่างโค้ด

```
if (log10rad < 0)
        log10rad = log((double)RADIX) / log((double)10);
if (!inmem)
        inmem = (char *)calloc((size_t)IN_LINE, sizeof(char));
```

ฟังก์ชัน ที่พบ

long LongInteger::zsread

จากการตรวจสอบพบว่า โค้ดจุดนี้จะเป็นการคำนวณค่า log10rad ที่จะทำการตรวจสอบว่ามีการคำนวณค่าไว้แล้วหรือไม่ ถ้าหากยังไม่มีจะทำการคำนวณและกำหนดค่าให้ ซึ่งเมื่อทำการเรียกใช้งานฟังก์ชันจะทำให้เกิด Branch prediction เนื่องมาจากต้องตรวจสอบว่ามีการคำนวณไว้แล้วหรือไม่ซึ่งเราสามารถนำโค้ดตรงส่วนนี้ไปทำงาน ที่ Constructor ของคลาส ซึ่งจะถูกเรียกใช้งานเพียงครั้งเดียว สามารถลดการทำงานที่ซ้ำซ้อนและยังลด Branch prediction ได้อีกด้วย

จากการปรับปรุงด้วยวิธีนี้พบว่ามีข้อดีข้อเสียคือ

ข้อดี โปรแกรมสามารถทำงานได้เร็วขึ้น โดยจากการทดลองเปรียบเทียบการทำงานระหว่างโปรแกรมก่อนและหลังทำการปรับปรุงจำนวน Clock Cycle ที่ใช้ลดลงดังนี้

| **Optimize LV.** | **LV. 0** | **Macro + LV. 0** | **Speed Up %** | **LV. 2** | **Macro + LV. 2** | **Speed Up %** |
|---|---|---|---|---|---|---|
| Clock Cycle | 114,257,379 | 113,430,708 | 0.723 | 81,288,697 | 81,065,672 | 0.274 |

**ตารางที่ 3.4** จำนวน Clock Cycle ที่ใช้ในการทำงานหลังปรับปรุงโดยกำจัดความซ้ำซ้อนในโค้ดและลดจำนวนของ Branch prediction

ข้อเสีย เราจำเป็นที่จะต้องเข้าใจโค้ดการทำงานในส่วนที่จะแก้ไข และ จะต้องตรวจสอบการประกาศตัวแปรชนิด Verylong ทุกจุดเมื่อมีการประกาศแล้วจะต้องจองหน่วยความจำให้ทันที พร้อมทั้งกำหนดค่า 0 ให้เพื่อป้องกันความผิดพลาด อันที่จะเกิดจากการเข้าถึงหน่วยความจำที่ยังไม่ได้มีการจอง ซึ่งเป็นการมีความยุ่งยาก

3.3.2.3 การลดจำนวนตัวแปรสำหรับการเรียกใช้ฟังก์ชัน

จากโปรไฟล์จะพบว่า เราจะสามารถตรวจสอบได้ว่าฟังก์ชันใดที่มีการส่งค่าให้มากกว่า 4 ตัว และพบว่ามีด้วยกัน 3 ฟังก์ชัน ได้แก่

LongInteger::kar_mul(long *,long *,long * *,long,char *)

LongInteger::zmulmod(long *,long *,long *,long * *,char *)

LongInteger::zexpmod(long *,long *,long *,long * *,char *)

และจากการสังเกตพบได้ว่าค่าที่ส่งให้ตัวสุดท้ายนั้น มีหน้าที่เก็บสถานะที่ส่งเข้ามาเพื่อเก็บสถานะหากเกิดข้อผิดพลาดขึ้น เราจึงได้ทำการปรับปรุงแก้ไขโดยการตัดตัวแปรตัวสุดท้ายที่เก็บสถานะนี้ออก เนื่องจากไม่ส่งผลกระทบกับการทำงาน จากนั้นได้ทำการปรับปรุงแก้ไขโดยถ้าหากเกิดข้อผิดพลาด ขึ้นให้ทำการแสดงผลออกที่หน้าจอทันที

จากการปรับปรุงด้วยวิธีนี้พบว่ามีข้อดีข้อเสียคือ

ข้อดี โปรแกรมสามารถทำงานได้เร็วขึ้น โดยจากการทดลองเปรียบเทียบการทำงานระหว่างโปรแกรมก่อนและหลังทำการปรับปรุงจำนวน Clock Cycle ที่ใช้ลดลงดังนี้

| **Optimize LV.** | **LV. 0** | **Macro + LV. 0** | **Speed Up %** | **LV. 2** | **Macro + LV. 2** | **Speed Up %** |
|---|---|---|---|---|---|---|
| Clock Cycle | 114,257,379 | 114,241,319 | 0.014 | 81,288,697 | 81,278,943 | 0.0119 |

**ตารางที่ 3.5** จำนวน Clock Cycle ที่ใช้ในการทำงาน หลังปรับปรุงโดยลดจำนวนตัวแปรสำหรับการเรียกใช้ฟังก์ชัน

ข้อเสีย จากการปรับปรุงวิธีนี้ส่งผลให้ Clock Cycle ลดลงขึ้นกับจำนวนครั้งการทำงานของฟังก์ชัน ซึ่งถ้าหากว่าฟังก์ชันที่จะทำการปรับปรุงไม่มีการใช้งานบ่อยครั้งจะทำให้ สามารถลดจำนวน Clock Cycle ได้น้อย

3.3.2.4 การทำ Minimizing Buffer Allocation เราสามารถทำการปรับปรุงโดยนำตัวแปรชนิด Verylong ที่เป็นตัวแปรชนิด Local ซึ่งอยู่ตามฟังก์ชันต่างๆ มาทำให้เป็น Attribute ของคลาส เพื่อที่เมื่อได้ทำการสร้างคลาสแล้ว ตัวแปรเหล่านี้จะถูกจองพื้นที่ให้อย่างแน่นอน และตัวแปรที่เป็น Attribute ของคลาสจะมีอายุการใช้งานที่ยาวนานกว่า แบบ Local ซึ่งจะทำให้หน่วยความจำที่ทำการจองให้กับ ตัวแปรเหล่านี้มีโอกาสอยู่ในแคชมากขึ้นไปด้วย

3.3.2.5 การทำ Reused Variable หลังจากการทำ Minimizing Buffer Allocation แล้วจะทำการตรวจสอบการจองพื้นที่หน่วยความจำโดยใช้โปรแกรมที่เราพัฒนาขึ้น ทำการเก็บบันทึกสถิติการจองพื้นที่หน่วยความจำ โดยทำการปรับปรุงการจองพื้นที่หน่วยความจำ และการคืนพื้นที่หน่วยความจำ ซึ่งโปรแกรมของเราจะสามารถเก็บบันทึกเหตุการณ์ที่เกิดขึ้นได้ โดยมีรายละเอียด คือ ลำดับเหตุการณ์ที่เกิดขึ้น กระบวนการทำงานและตัวแปรใดเป็นผู้ทำ ทำจากฟังก์ชันใด จากนั้นทำการรวมตัวแปรที่สามรถที่สามารถใช้ร่วมกันได้ เพื่อให้เหลือตัวแปรน้อยที่สุดเพื่อที่จะได้มีการจองหน่วยความจำให้น้อยที่สุดเพื่อที่จะได้มีข้อมูลอยู่ในแคชให้มากที่สุด หลังจากนั้นนำไปทำการทดลองวัดการใช้งานหน่วยความจำหลักผลที่ได้ปรากฏดังตารางที่ 3.5

| Optimize LV. | LV. 0 | Our + LV. 0 | Reduce % | LV. 2 | Our + LV. 2 | Reduce % |
|---|---|---|---|---|---|---|
| Instruction | 554 | 551 | 0.541516 | 426 | 424 | 0.469484 |
| Data | 129 | 123 | 4. 651163 | 131 | 129 | 1.526718 |
| Sum | 683 | 674 | 5.192679 | 557 | 553 | 1.996201 |

**ตารางที่ 3.6** จำนวน การดึงข้อมูลจากหน่วยความจำหลักที่ทำงานโดย MMU ก่อนและหลังการปรับปรุงทางด้านการลดพลังงาน

จากตาราง ที่ 3.6 แสดงให้เห็นว่าวิธีการลดการใช้พลังงานที่ได้นำเสนอ สามารถลดการเข้าใช้งานหน่วยความจำหลักลงได้ทั้งจากการเข้าใช้หน่วยความจำหลักเพื่อดึงข้อมูล และดึงคำสั่งเข้ามาเก็บไว้ในแคช

ข้อดี จากการที่โปรแกรมจะต้องทำการจองพื้นหน่วยความจำ เมื่อเข้าใช้งานฟังก์ชันและทำการคืนเมื่อจบฟังก์ชันทำให้เกิดการจองพื้นที่และการคืนพื้นที่หน่วยความจำถึง 6,243 ครั้ง แต่หลังจากการทำ Minimizing Buffer Allocation การจองพื้นที่และการคืนพื้นที่จะทำที่ Constructor และ Destructor ของคลาสเท่านั้น ทำให้ลดจำนวนกระบวนการจองพื้นที่หน่วยความจำ ซึ่งช่วยลดระยะเวลาการทำงานอย่างแน่นอน และยังช่วยเพิ่มโอกาสที่ข้อมูลจะอยู่ในแคชเพิ่มมากขึ้นอีกด้วย นอกจากนี้หลังจากทำการ Reused Variable แล้วยังช่วยลดการจองพื้นที่หน่วยความจำและให้ใช้หน่วยความจำที่มีอยู่เดิม ซึ่งก็มีโอกาสที่จะอยู่ในแคชอยู่แล้วเช่นกัน

ข้อเสีย จะต้องทำการวิเคราะห์ว่า ตัวแปรใดสามารถถอดออกมาเป็น Attribute ของคลาสและตัวแปรใดสามารถ นำกลับมาใช้ได้อีกครั้ง

สรุป จากการทดลองปรับปรุงประสิทธิภาพการเข้ารหัสแบบอาร์เอสเอทางด้านความเร็ว ทำให้พบว่า จากการทำการลดจำนวนฟังก์ชันที่เรียกใช้ สามารถทำให้โปรแกรมทำงานได้เร็วขึ้นถึง 8.76 ถึง 5.99 % รองลงมาจากนั้นจึงเป็นวิธีการปรับปรุงโดยใช้วิธี การกำจัดความซ้ำซ้อนในโค้ดและลดจำนวนของ Branch prediction โปรแกรมทำงานได้เร็วขึ้น 0.274 - 0.723 % จากนั้นจึงเป็นวิธีการลดจำนวนตัวแปรสำหรับการเรียกใช้ฟังก์ชัน สามารถทำให้โปรแกรมทำงานได้เร็วขึ้น 0.014 - 0.0119 % ส่วนการปรับปรุงประสิทธิภาพทางด้านพลังงานนั้น เราสามารถพิสูจน์ให้เห็นว่าวิธีการที่ได้นำเสนอ สามารถลดการเกิดการค้นหาข้อมูลไม่พบในแคชได้ราว 5.19 - 1.99 % ซึ่งจะนำไปวัดการใช้พลังงานจริง เพื่อแสดงให้เห็นว่าสามารถลดการใช้พลังงานลงได้จริง

# บทที่ 4

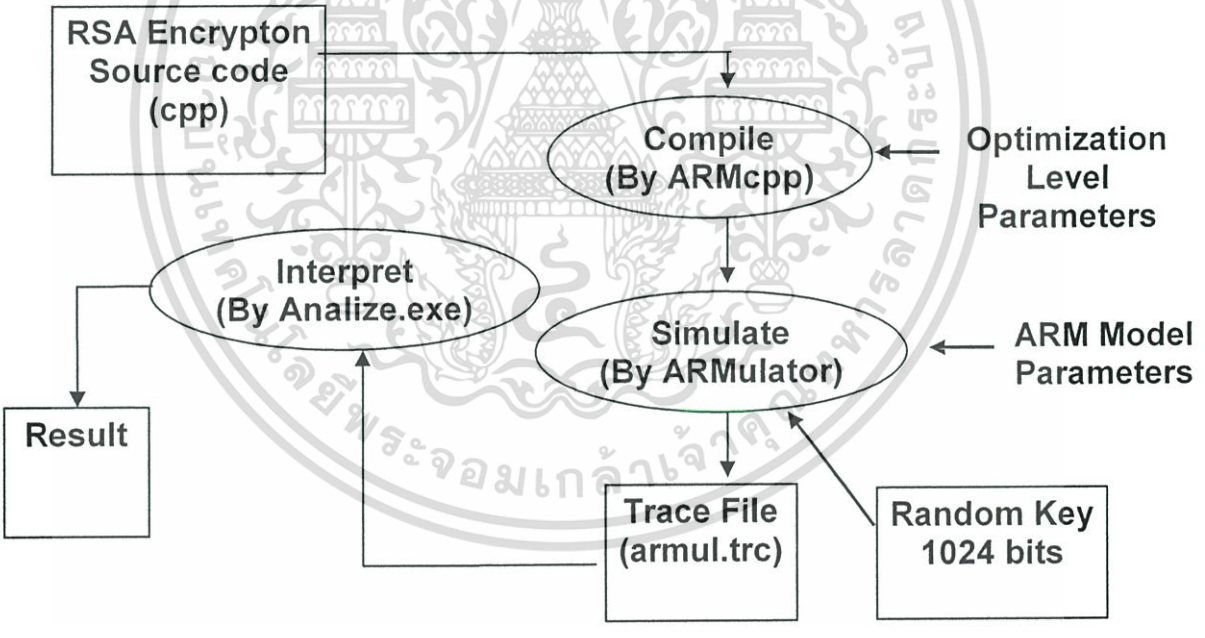
# การทดลองและผลการทดลอง

## 4.1 การทดลอง

การทดลองจะแบ่งออกเป็นสองส่วนคือการทดลองเพื่อหาประสิทธิภาพด้านความเร็ว และ การทดลองเพื่อหาประสิทธิภาพด้านพลังงาน

### 4.1.1 ขั้นตอนการทดลองการเพิ่มประสิทธิภาพด้านเพิ่มความเร็ว

เพื่อแสดงถึงประสิทธิภาพด้านความเร็วที่เราเสนอ รูปที่ 4.1 แสดงวิธีการทดลอง เราจะใช้โค้ดการเข้ารหัสแบบอาร์เอสเอ ที่ทำการแปลงโค้ดไปทำงานบนระบบปฏิบัติการซิมเบียน ใช้อาร์มคอมไพเลอร์ (ARMcpp) ซึ่งมีความสามารถปรับปรุงประสิทธิภาพได้สูงคือมีโค้ดขนาดเล็กและมีสามลำดับชั้นในการปรับปรุงประสิทธิภาพทางด้านความเร็ว ปรับปรุงประสิทธิภาพระดับ 0, ปรับปรุงประสิทธิภาพระดับ 1 และ ปรับปรุงประสิทธิภาพระดับ 2




**รูปที่ 4.1** แสดงขั้นตอนการวัดประสิทธิภาพด้านความเร็ว

หลังจากคอมไพล์เสร็จจะได้ไบนารี่ไฟล์ จะนำไปจำลองการทำงานโดย ARMulator [8] ซึ่งสามารถจำลองการทำงานของ ARM โปรเซสเซอร์ได้หลายเวอร์ชันของสถาปัตยกรรมชุดคำสั่ง ARM และสามารถนับจำนวนสัญญาณนาฬิกา, การเข้าใช้หน่วยความจำ ตลอดจนเหตุการณ์พิเศษ (เช่น แคชข้อมูลหรือชุดคำสั่ง, write back stall) โดยเราใช้ ARMulator จำลองการทำงานของ ARM920T โดยที่ไม่มีหน่วย floating-point

ในการจำลองการทำงานเข้ารหัสแบบอาร์เอสเอ เราสุ่มรหัสที่มีความยาวขนาน 1024 บิต ในการทดลองโดยมีข้อมูลที่จะทำการเข้ารหัสขนาด 1 กิโลไบต์

หลังจากการจำลองจะได้ไฟล์ข้อมูลการทำงานคือ armul.trc โดยภายในประกอบด้วยเหตุการณ์ของโพรเซสเซอร์ต่างเช่นการเข้าใช้หน่วยความจำ, การไม่เจอข้อมูลในแคชและข้อมูลที่เกี่ยวข้อง โดยเราจะแปลข้อมูลที่ได้จากไฟล์ข้อมูลบันทึกการทำงาน โดยใช้โปรแกรมวิเคราะห์ที่เราพัฒนาขึ้นคือ Analize.exe เพื่อเอาข้อมูลที่เกี่ยวกับการใช้หน่วยความจำและเหตุการณ์ที่เกี่ยวกับแคช ผลลัพธ์ที่ได้จากการวิเคราะห์โดย Analize.exe จะได้จำนวนการอ้างหน่วยความจำทั้งแบบการอ่านและการเขียน จำนวนการเกิดการค้นหาข้อมูลในแคชข้อมูลไม่พบและ จำนวนการเกิดการค้นหาข้อมูลในแคชคำสั่งไม่พบ และนำเอาข้อมูลเหล่านี้ไปแสดงเป็นกราฟ และตาราง โดยเราจะวัดผลการทดลองประสิทธิภาพ ของโค้ดการเข้ารหัสแบบอาร์เอสเอ ก่อนและหลังการปรับปรุงประสิทธิภาพแสดงในส่วนของผลการทดลอง ในรูปที่ 4.2 แสดงบางส่วนของ armul.trc

```
Date: Sat May 07 18:47:11 2005
Source: Armul
Options: Trace Events

E 00008000 00000000 10005
BSR4O___ A0000000 00000C1E
E 00008000 00000000 10002
BSR8O___ 00008018 E240B001 E242C001
IT 00008000 e28f8090 ADD r8,pc,#0x90 ; #0x8098
IT 00008004 e898000f LDMIA r8,{r0-r3}
BNR4O___ A0000000 00000C1E
BNR8O___ 00008098 00007804 00007828
BSR8O___ 00008080 10844009 E3C44003
BSR8O___ 00008088 E2555004 24847004
```

**รูปที่ 4.2** แสดงตัวอย่างข้อมูลใน “armul.trc”

ต่อไปเราจะแสดงวิธีการแปลข้อมูลการเข้าใช้หน่วยความจำและเหตุการณ์ต่างๆในไฟล์ข้อมูลการ

1) Memory Bus Trace (Bline): บรรทัดที่เริ่มต้นด้วย B แสดงถึงการเข้าใช้บัสของหน่วยความจำ โดยจะมีรูปแบบดังนี้

B<type><rw><size>[O][L][S] <address> <data>

โดย

<type> แสดงถึงชนิดของสัญญาณนาฬิกา

S sequential

N nonsequential

<rw> แสดงถึงการอ่านหรือการเขียน

R การอ่าน (Read)

W การเขียน (Write)

<size> แสดงถึงขนาดในการเข้าใช้บัสหน่วยความจำ

4 word (32 บิต)

2 halfword (16 บิต)

1 ไบต์ (8บิต)

O คือ Opcode fetch (instruction fetch)

L คือ locked access (SWR instruction)

S คือ speculative instruction fetch.

<address> จะอยู่ในรูปแบบเลขฐาน 16 ตัวอย่างเช่น 00008008

<data> จะแสดงตามนี้:

ค่าจากการอ่านหรือเขียน เช่น EB00000C

2) Events Trace (E lines): รูปแบบของบรรทัดเหตุการณ์ (E) เป็นดังนี้:

E <word1> <word2> <event_number>

ตัวอย่างเช่น E 00000048 00000000 10001

โดย

<word1> แสดงเป็น word บอกเช่นค่า PC

<word2> แสดงเป็น word บอกเช่นตำแหน่งที่เกิด event

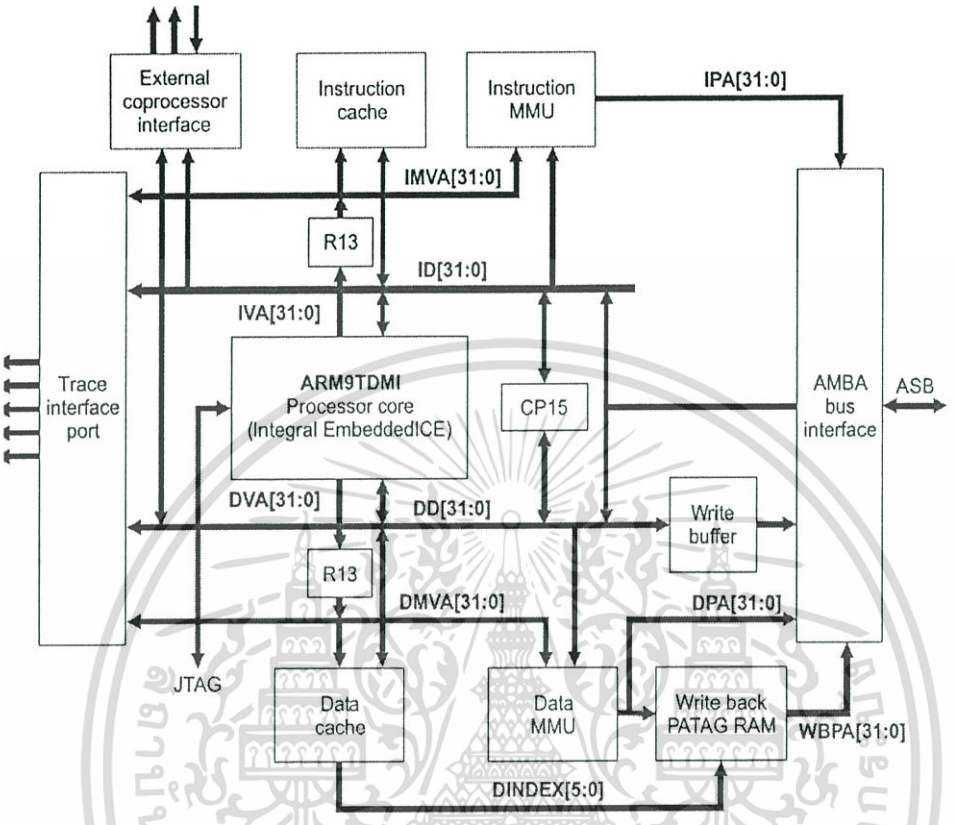
<event_number> คือตัวเลขของเหตุการณ์ ตัวอย่างเช่น 0x10001 คือ MMUEvent_DLineFetch

โดยเหตุการณ์ทั้งหมดจะอธิบายในตารางที่ 4.1

| Event name (MMUEvent_) | Word 1 | Word 2 | Event number |
|---|---|---|---|
| DLineFetch | Miss address | Victim address | 0x10001 |
| ILineFetch | Miss address | Victim address | 0x10002 |

**ตารางที่ 4.1** เหตุการณ์พิเศษต่างๆที่ตรวจได้จาก ARMulator

รูปที่ 4.3 แสดงฟังก์ชันการทำงานของ ARM920T [4] โดย ARMulator ตรวจจับข้อมูลจาก trace interface port มันถูกกำหนดให้ตรวจจับการเข้าใช้บัสของหน่วยความจำในการอ่านและเขียนจาก AMBA 32-bit bus interface port




**รูปที่ 4.3** ARM920T ไดอะแกรมจาก ARM Limited [4]

การทดลองเพิ่มประสิทธิภาพของการเข้ารหัสแบบอาร์เอสเอ ในด้านการเพิ่มความเร็วที่ทำงานบนอาร์มโปรเซสเซอร์ จะทำตามวิธีการที่ได้กล่าวไปแล้วข้างต้น โดยที่เราจะทำการทดลองเปรียบเทียบผลความเร็ว (จำนวน Clock Cycle) ก่อนและหลังการปรับปรุงเพื่อเพิ่มความเร็วในการทำงานของการเข้ารหัสแบบอาร์เอสเอ บนอาร์มโปรเซสเซอร์

ทำการปรับปรุงประสิทธิภาพสี่แบบที่แตกต่างเปรียบเทียบกันที่ระดับการปรับปรุงที่ระดับ 0, ระดับ 2, ระดับ 0 + การปรับปรุงความเร็วของเรา และ ระดับ 2 + การปรับปรุงความเร็วของเราโดยการปรับปรุงความเร็วของเรา ด้วยวิธีการที่เราเสนอไปในบทที่ 3.1 โดยใช้โดนใช้รหัสสุ่มที่มีความยาวขนาด 1024 บิต

เราจะหาประสิทธิภาพ ของวิธีการปรับปรุงทางด้านความเร็วของเรา จากจำนวน Clock Cycles และ จำนวน คำสั่ง ซึ่งแม่นยำกว่าการจับเวลาการทำงานของโปรแกรม

### 4.1.2 ขั้นตอนการทดลองการเพิ่มประสิทธิภาพด้านการลดการใช้พลังงาน

การทดลองเพิ่มประสิทธิภาพการเข้ารหัสแบบอาร์เอสเอ ทางด้านการลดการใช้พลังงานที่ทำงานบนอาร์มโปรเซสเซอร์ เราจะการทดลองเปรียบเทียบการใช้พลังงาน ก่อนและหลังการปรับปรุงเพื่อลดการใช้พลังงานของการเข้ารหัสแบบอาร์เอสเอบนอาร์มโปรเซสเซอร์

โดยในการวัดการใช้พลังงานในอาร์มโปรเซสเซอร์ เราจะตรวจสอบจากระบบปฏิบัติการบนโทรศัพท์มือถือที่ใช้อาร์มโปรเซสเซอร์ ในการทดลองนี้คือระบบปฏิบัติการซิมเบียนนั้นเอง ซึ่งซิมเบียนได้มีบริการ ให้เราเรียกใช้เพื่อทำการตรวจสอบ ระดับพลังงานของแบตเตอรี่ โดยในระบบปฏิบัติการซิมเบียนนั้น ได้แบ่งระดับของพลังงานออกเป็น 7 ระดับโดยแต่ละระดับจะแสดงช่วงพลังงานแต่ละช่วงต่างกันไปดังนี้

| ระดับพลังงานในแบตเตอรี่ | แบตเตอรี่ มีพลังงานตั้งแต่ (%) | แบตเตอรี่ มีพลังงานจนถึง (%) |
|---|---|---|
| ระดับที่ 1 | 0 | 14.2857 |
| ระดับที่ 2 | 14.2858 | 28.5715 |
| ระดับที่ 3 | 28.5716 | 42.8573 |
| ระดับที่ 4 | 42.8574 | 57.1431 |
| ระดับที่ 5 | 57.1432 | 71.4289 |
| ระดับที่ 6 | 71.4290 | 85.7147 |
| ระดับที่ 7 | 85.7148 | 100.0000 |

**ตารางที่ 4.2** ระดับพลังงานในแบตเตอรี่ ที่มีในแต่ละระดับของโทรศัพท์มือถือที่ใช้วัด

เราได้ทำการโดยทดลองโดยใช้โปรแกรม ที่เราเขียนขึ้นเพื่อตรวจสอบว่าในแต่ละระดับมีปริมาณพลังงานเท่ากันหรือไม่โดยให้โปรแกรมของเรา ทำการจำลองการทำงานเหมือนๆกันไปเรื่อยๆ ตั้งแต่ขณะที่แบตเตอรี่เต็มจนถึงแบตเตอรี่ลดไปทุกๆหนึ่งระดับจะทำการบันทึกว่าสามารถจำลองการทำงานได้กี่ครั้งในแต่ละระดับพลังงานโดย จะต้องตั้งค่าโทรศัพท์มือถือที่ใช้ทดลองให้ทำงานที่ Flight Mode หรือ Offline Mode เพื่อไม่ให้ ระหว่างการทดลองมีการนำพลังงานไปใช้ในการสื่อสารซึ่งจะส่งผลให้การผลการทดลองคลาดเคลื่อน ผลปรากฏว่าแต่ละระดับมีพลังงานเฉลี่ยเท่ากันเราจึงใช้ตารางข้างบนเป็นมาตรฐานในการทดลองของเรา

จากนั้นเราจะทำการจำลองการทำงานของการเข้ารหัสแบบ อาร์เอสเอ ก่อนและหลังการปรับปรุงประสิทธิภาพว่ามีการใช้พลังงานเป็นเช่นไร โดยจะทำการทดลองโดยสุ่มใช้ รหัส ขนาด 1024 บิต จำนวน100 ครั้ง ในการเข้ารหัสข้อมูลขนาด 1 กิโลไบต์ ชุดเดิมผลที่ได้จะเป็นจำนวนเฉลี่ยที่สามารถทำการเข้ารหัสได้ในการใช้พลังงาน 14.2857% และคำนวณหาค่าเฉลี่ยต่อการใช้พลังงานในการเข้ารหัส 1 ครั้งต่อไป จากนั้นจะนำผลที่ได้มาเปรียบเทียบโค้ด ก่อนและหลังการปรับปรุงว่ามีการใช้พลังงานเป็นเช่นไร

### 4.1.3 ขั้นตอนการทดลองการเพิ่มประสิทธิภาพด้านการลดพลังงานและเพิ่มความเร็ว

วิธีการทดลองในการวัดการเพิ่มประสิทธิภาพด้านการลดพลังงานและเพิ่มความเร็ว ทำการทดลองเหมือนกับข้อ4.1.1 และ 4.1.2 โดยนำโค้ด ที่ได้ทำการปรับปรุงประสิทธิภาพทั้งในด้านการเพิ่มความเร็ว และการลดพลังงาน มาทำการทดลองทั้งแบบข้อ 4.1.1 และ 4.1.2 เพื่อให้ได้ผลการเปรียบเทียบประสิทธิภาพการเพิ่มความเร็วและการลดการใช้พลังงาน

## 4.2 วิเคราะห์ประสิทธิภาพและผลการทดลอง

### 4.2.1 วิเคราะห์ประสิทธิภาพการทดลองและผลการทดลองด้านความเร็ว

สิ่งที่เราสนใจในการทดลองนี้คือ จำนวน Clock Cycle ที่ใช้ในการทำงาน โดยเฉลี่ย จากการทดลองโดยสุ่มใช้ รหัส ขนาด 1024 บิต ทำการเข้ารหัสข้อมูลชุดเดียวกันที่มีขนาด 1 กิโลไบต์ ทำการทดลองโดยใช้ Armulator ทำการจำลองการทำงานโดยทำการทดลอง 4 แบบ จำนวนแบบละ 100 ครั้งคือ

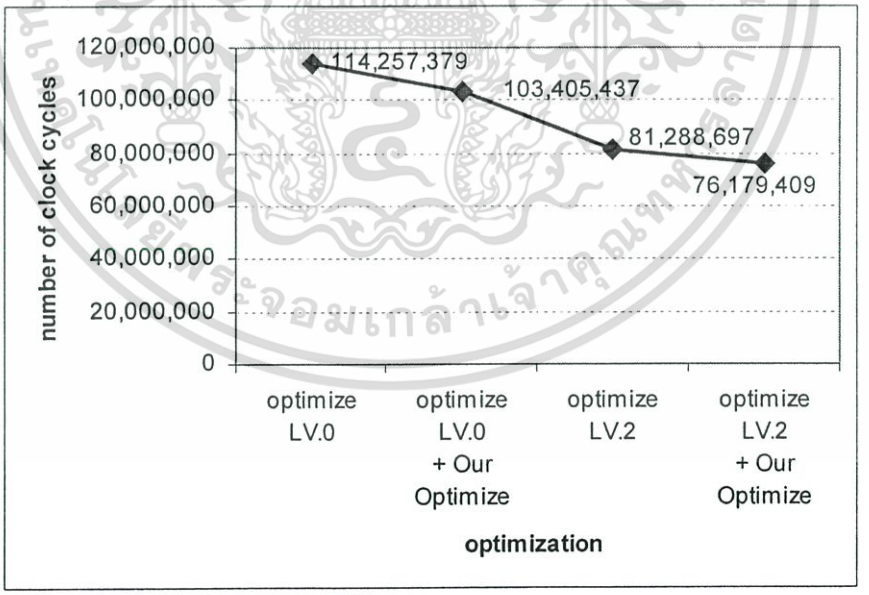
- ปรับปรุงประสิทธิภาพด้วยอาร์มของคอมไพเลอร์ที่ระดับ 0
- ปรับปรุงประสิทธิภาพด้วยอาร์มของคอมไพเลอร์ที่ระดับ 0 + ปรับปรุงประสิทธิภาพเพื่อเพิ่มความเร็วของเรา
- ปรับปรุงประสิทธิภาพด้วยอาร์มของคอมไพเลอร์ที่ระดับ 2
- ปรับปรุงประสิทธิภาพด้วยอาร์มของคอมไพเลอร์ที่ระดับ 2 + ปรับปรุงประสิทธิภาพเพื่อเพิ่มความเร็วของเรา

จากนั้นนำผลการทดลองที่ได้มาหาค่าเฉลี่ยของจำนวน Clock Cycle ที่ใช้ในการทำงาน ซึ่งได้ผลการทดลองดังตารางที่ 4.3

| Optimize LV. | LV. 0 | Ours Optimize + LV. 0 | Speed Up % | LV. 2 | Ours Optimize + LV. 2 | Speed Up % |
|---|---|---|---|---|---|---|
| Clock Cycle | 114,257,379 | 103,405,437 | 9.497 | 81,288,697 | 76,179,409 | 6.285 |

**ตารางที่ 4.3** จำนวน Clock Cycle เฉลี่ยจากการทดลองในการปรับปรุงประสิทธิภาพเพื่อเพิ่มความเร็ว

ด้วยวิธีการของเราเทียบกับการปรับปรุงประสิทธิภาพของอาร์มของคอมไพเลอร์วิธีการของเราสามารถลดจำนวน Clock Cycle ได้มาก ที่ระดับการปรับปรุงประสิทธิภาพของอาร์มคอมไพเลอร์ที่ระดับ 0 แต่เมื่อเปรียบเทียบกับการปรับปรุงประสิทธิภาพของอาร์มคอมไพเลอร์ที่ที่ระดับ 2 การปรับปรุงประสิทธิภาพของเราทำให้จำนวน Clock Cycle ลดลงน้อยกว่า ที่ระดับ 0 แต่อย่างไรก็ตามวิธีการของเรา เมื่อใช้รวมกับการปรับปรุงประสิทธิภาพที่ระดับ 2 ก็ทำให้เห็นว่าช่วยทำให้จำนวน Clock Cycle ลดลงอีกด้วย ดังรูปที่ 4.4



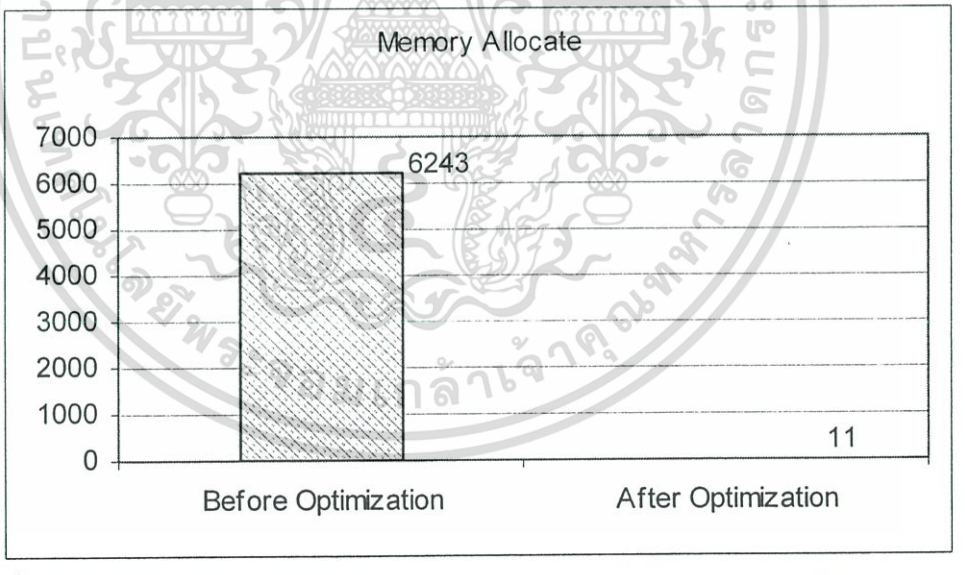

**รูปที่ 4.4** จำนวน Clock Cycles ที่ลดลงเมื่อเปรียบเทียบการปรับปรุงประสิทธิภาพของอาร์มคอมไพเลอร์กับการปรับปรุงประสิทธิภาพทางด้านความเร็วของเรา

จากการทดลองแสดงให้เห็นว่าวิธีการปรับปรุงประสิทธิภาพความเร็ว ที่เรานำมาใช้ในการปรับปรุงการทำงานของการเข้ารหัสแบบอาร์เอสเอ สามารถปรับปรุงทำให้การเข้ารหัสแบบอาร์เอสเอ ให้ทำงานได้เร็วขึ้นกว่าเดิมประมาณ 6.28-9.49 %

**4.2.2 วิเคราะห์ประสิทธิภาพการทดลองและผลการทดลองในการเพิ่มประสิทธิภาพด้านการลดพลังงาน**

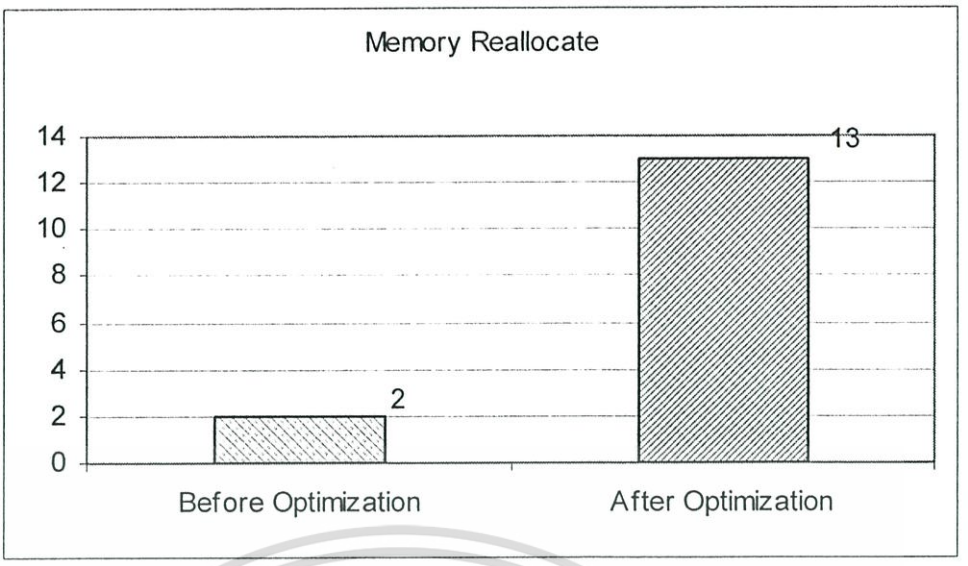
การปรับปรุงประสิทธิภาพในด้านการลดการใช้พลังงาน จะเห็นได้ว่ามุ่งเน้นไปที่การใช้หน่วยความจำให้น้อยที่สุดโดยใช้วิธีการหลักๆ สองวิธีก็คือ Minimizing Buffer Allocation และ Reused Variable โดยจะมุ่งเน้นการ Reused Variable ที่มีการจองหน่วยความจำให้มาใช้ตัวแปรตัวเดียวกัน เพื่อช่วยลดการเกิดปัญหาการค้นหาข้อมูลในแคชไม่พบด้วยเช่นกัน จากการปรับปรุงประสิทธิภาพด้านพลังงานของเราจะทำให้ลดการจองหน่วยความจำ และมีปริมาณหน่วยความจำที่ถูกนำกลับมาใช้เพิ่มขึ้น

โดยสามารถลดจำนวนการจองหน่วยความจำลงได้ถึง 99.82% โดยก่อนการปรับปรุงประสิทธิภาพ มีการจองหน่วยความจำอยู่ที่ 6,243 ครั้ง แต่หลังจากการปรับปรุงประสิทธิภาพสามารถลดการจองหน่วยความจำลงไปเหลือ 11 ครั้ง ดังรูปที่ 4.5



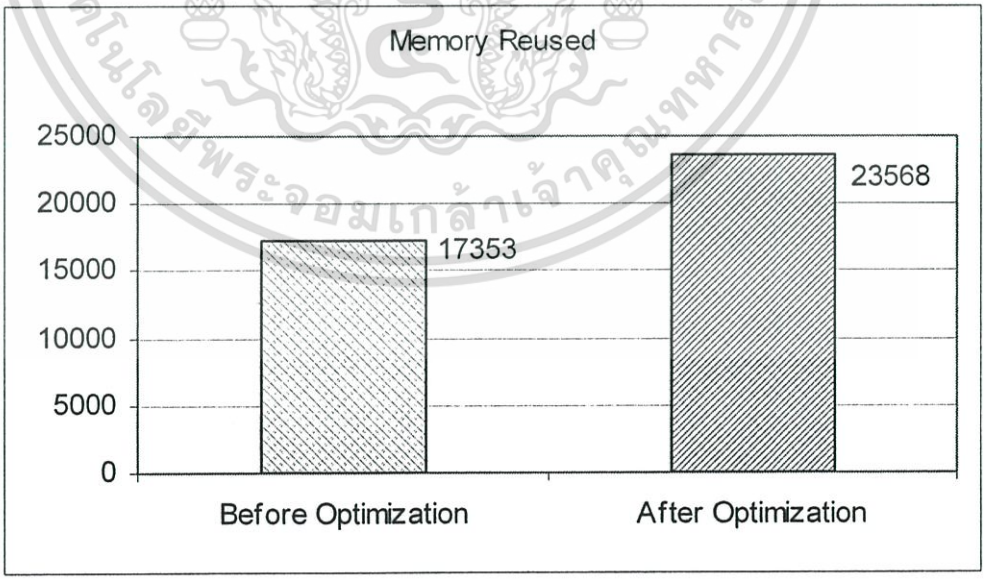

**รูปที่ 4.5** เปรียบเทียบจำนวนการจองหน่วยความจำก่อนและหลังการปรับปรุงประสิทธิภาพทางด้านพลังงาน

แต่ส่งผลกระทบให้กับการจองหน่วยความจำใหม่แทนที่ของเดิมเนื่องจากของเดิมขนาดไม่พอกับข้อมูลที่จะต้องใช้ โดยเพิ่มขึ้น 84.61% เมื่อเทียบกับของเดิมซึ่งใช้เพียง 2 ครั้งมาเป็น 13 ครั้ง




**รูปที่ 4.6** เปรียบเทียบจำนวนการจองหน่วยความจำใหม่แทนที่เดิม ก่อนและหลังการปรับปรุงประสิทธิภาพทางด้านพลังงาน

ส่วนการนำหน่วยความจำที่ได้จองไปแล้วกลับมาใช้ใหม่อีกครั้งด้วยการปรับปรุงประสิทธิภาพของเราสามารถเพิ่มปริมาณการนำหน่วยความจำกลับมาใช้ใหม่ได้ถึง 26.37% โดยก่อนทำการปรับปรุงประสิทธิภาพ มีการนำหน่วยความจำกลับมาใช้ใหม่จำนวน 17,353 ครั้ง แต่หลังจากทำการปรับปรุงประสิทธิภาพแล้ว สามารถนำหน่วยความจำกลับมาใช้ใหม่ได้ถึง 23,568 ครั้ง ดังรูปที่4.7




**รูปที่ 4.7** เปรียบเทียบจำนวนการนำหน่วยความจำเดิมกลับมาใช้ใหม่ ก่อนและหลังการปรับปรุงประสิทธิภาพทางด้านพลังงาน

โดยในเรื่องของปริมาณพลังงานที่ใช้ในการทำงาน โดยเฉลี่ยจากการทดลองโดยสุ่มใช้รหัสขนาด 1024 บิต ทำการเข้ารหัสข้อมูลชุดเดียวกันที่มีขนาด 1 กิโลไบต์ ทำการทดลองปรับปรุงโค้ด และนำไปทำงานจริงบนโทรศัพท์มือถือที่ใช้อาร์มโปรเซสเซอร์ โดยเมื่อนำไปทำงานบนโทรศัพท์มือถือนั้นอาร์มคอมไพเลอร์ จะทำการปรับปรุงประสิทธิภาพที่ระดับสูงสุดระดับเดียว ดังนั้นการทดลองในการวัดการใช้พลังงานก็จะได้ผลการทดลองการใช้พลังงาน 2 แบบคือ

- ปรับปรุงประสิทธิภาพ ด้วย อาร์มของคอมไพเลอร์ที่ระดับ 2
- ปรับปรุงประสิทธิภาพ ด้วย อาร์มของคอมไพเลอร์ที่ระดับ 2 + ปรับปรุงประสิทธิภาพเพื่อลดการใช้พลังงาน

จากนั้นนำผลการทดลองที่ได้มาหาค่าเฉลี่ยของการใช้พลังงานที่ใช้ในการเข้ารหัสอาร์เอสเอ ซึ่งจะได้ผลการทดลองตามตารางที่ 4.4 และ 4.5

| Test Case | Original | Optimize | Increst % |
|---|---|---|---|
| จำนวนเฉลี่ย การเข้ารหัสที่ทำได้ (ครั้ง) | 49,543 | 51,203 | 3.241 |

**ตารางที่ 4.4** จำนวนครั้งในการเข้ารหัสที่ใช้พลังงานในการเข้ารหัสไป 14.28 % ก่อนและหลังการปรับปรุงประสิทธิภาพด้านพลังงาน

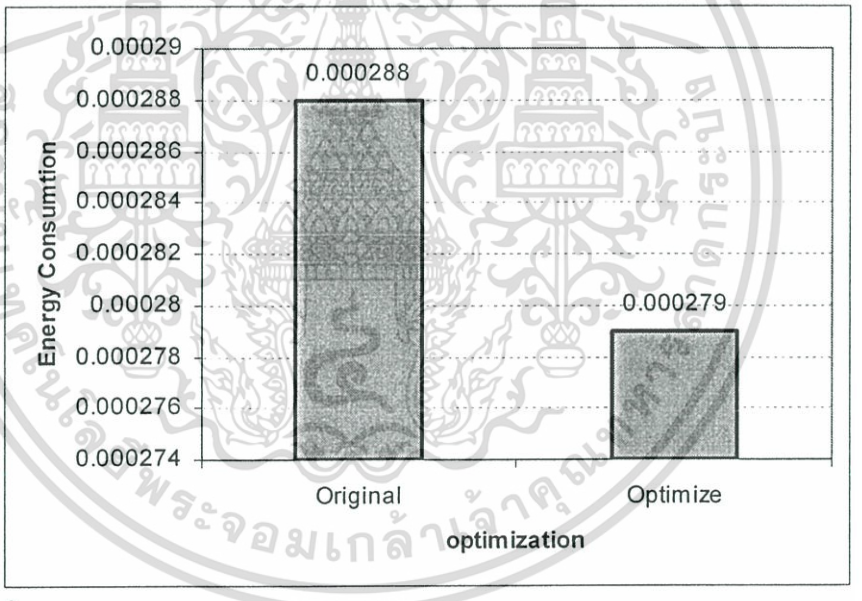
จากตารางที่ 4.4 แสดงให้เห็นว่า ในการใช้พลังงานใน แบตเตอรี่ 14.28% ด้วยโปรแกรมการเข้ารหัส อาร์เอสเอ เดิม ด้วย รหัส 1024 บิต สามารถทำการเข้ารหัสได้ 49,543 ครั้ง แต่หลังจากทำการปรับปรุงประสิทธิภาพทางด้านพลังงานแล้วสามารถทำงานได้มากขึ้นโดยใช้พลังงานเท่าเดิมคือ 14.28% โดยทำงานเพิ่มขึ้นเป็นสามารถทำงานได้ 51,203 ครั้งในการใช้พลังงาน 14.28% เท่าเดิม โดยส่วนต่างที่เพิ่มขึ้นเป็นจำนวน 1,660 ครั้ง คิดเป็น 3.241 %

จากนั้นเราจะทำการหาค่าพลังงานที่ใช้ในการเข้ารหัสแบบ อาร์เอสเอ ด้วย รหัส 1024 บิต 1 ครั้งออกมาได้โดย จำนวน พลังงานที่ใช้ไป / จำนวนการทำงานที่ทำได้ จะได้ว่าพลังงานที่ใช้ไปใน 1 ครั้งใช้ไปเท่าไร ดังตารางที่ 4.5

| Test Case | Original | Optimize | Reduce % |
|---|---|---|---|
| ปริมาณพลังงานที่ใช้ไปในการเข้ารหัส(%) | 0.000288 | 0.000279 | 3.125 |

**ตารางที่ 4.5** พลังงานเฉลี่ยที่ใช้ไปในการเข้ารหัสต่อ 1 ครั้งก่อนและหลังการปรับปรุงประสิทธิภาพ

จากตารางที่ 4.5 แสดงให้เห็นว่า ในการใช้พลังงานในเข้ารหัสแบบ อาร์เอสเอ 1 ครั้งในการทดลองของเรา ก่อนทำการปรับปรุงประสิทธิภาพ ใช้พลังงาน 0.000288 หน่วย แต่หลังจากการทำการปรับปรุงประสิทธิภาพแล้วใช้พลังงานลดลงเหลือ 0.000279 โดยส่วนต่างที่ลดลงเป็นจำนวน 0.000009 หน่วย คิดเป็น 3.241 % ปริมาณการใช้พลังงานในการเข้ารหัสแบบ อาร์เอสเอเดิม




**รูปที่ 4.8** เปอร์เซ็นต์ของการบริโภคพลังงานเปรียบเทียบก่อนและหลังการปรับปรุงประสิทธิภาพทางด้านพลังงาน

จากการทดลองแสดงให้เห็นว่าวิธีการปรับปรุงประสิทธิภาพ พื่อลดการใช้พลังงานที่เรานำมาใช้ในการปรับปรุงการทำงานของการเข้ารหัสแบบอาร์เอสเอนั้น สามารถปรับปรุงทำให้การเข้ารหัสแบบอาร์เอสเอ ให้ทำงานได้โดยใช้พลังงานน้อยลงราว 3.125% ของการทำงานก่อนการปรับปรุงประสิทธิภาพ

**4.2.3 วิเคราะห์ประสิทธิภาพการทดลองและผลการทดลองในการเพิ่มประสิทธิภาพด้านการเพิ่มความเร็วและการลดพลังงาน**

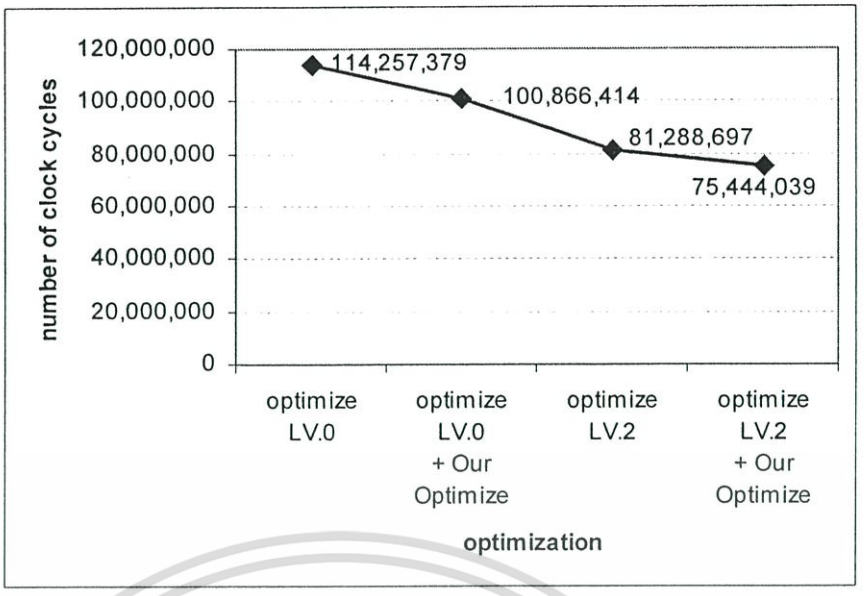
ผลที่เราสนใจในการวัดประสิทธิภาพทั้งความเร็วและพลังงานคือเราจะดูที่ จำนวน Clock Cycle และจำนวนพลังงานที่ใช้ในการทำงานของการเข้ารหัสแบบ อาร์เอสเอ โดยในการทดลองจะแบ่งการทดลองออกเป็นสองส่วนคือ

1. การทดลองและผลการทดลองและวิเคราะห์ผลในแง่ของการเพิ่มความเร็วเมื่อเรานำการปรับปรุงประสิทธิภาพในการเพิ่มความเร็วและลดการใช้พลังงานมาทำการทดลองร่วมกันโดยจะใช้วิธีการทดลองเหมือนกับวิธีการทดลองในเรื่องการเพิ่มความเร็ว ตามที่ได้กล่าวไว้ในข้อ 4.1.1 ซึ่งเมื่อได้นำโปรแกรมการเข้ารหัสแบบ อาร์เอสเอ ก่อนทำการปรับปรุงประสิทธิภาพ และหลังทำการปรับปรุงประสิทธิภาพทั้งในเรื่องการเพิ่มความเร็วและการลดการใช้พลังงาน มาทดลองตามข้อ 4.1.1 จะได้ผลการทดลองดังตารางที่ 4.6

| Optimize LV. | LV. 0 | Ours Optimize + LV. 0 | Speed Up % | LV. 2 | Ours Optimize + LV. 2 | Speed Up % |
|---|---|---|---|---|---|---|
| Clock Cycle | 114,257,379 | 100,866,414 | 11.720 | 81,288,697 | 75,444,039 | 7.190 |

**ตารางที่ 4.6** จำนวน Clock Cycle เฉลี่ยจากการทดลองโดยการปรับปรุงประสิทธิภาพเพื่อเพิ่มความเร็วและลดการใช้พลังงาน

จากตารางที่ 4.6 ด้วยวิธีการปรับปรุงประสิทธิภาพของเรา เมื่อเทียบกับการปรับปรุงประสิทธิภาพของอาร์มของคอมไพเลอร์ จะเห็นว่าวิธีการของเราสามารถลดจำนวน Clock Cycle ได้มาก ที่ระดับการปรับปรุงประสิทธิภาพของอาร์มคอมไพเลอร์ที่ระดับ 0 แต่เมื่อเปรียบเทียบกับการปรับปรุงประสิทธิภาพของอาร์มคอมไพเลอร์ที่ระดับ 2 การปรับปรุงประสิทธิภาพของเราทำให้จำนวน Clock Cycle ลดลงน้อยกว่า ที่ระดับ 0 แต่อย่างไรก็ตามตารางที่ 4.6 แสดงให้เห็นว่าจากการที่นำวิธีการปรับปรุงประสิทธิภาพทางด้านพลังงาน มารวมกับการปรับปรุงประสิทธิภาพทางด้านความเร็ว ก็ช่วยให้โปรแกรมทำงานเร็วขึ้นจากการลดการจองหน่วยความจำ และการเพิ่มการนำหน่วยความจำกลับมาใช้ใหม่




**รูปที่ 4.9** จำนวน Clock Cycles ที่ลดลงเมื่อเปรียบเทียบการปรับปรุงประสิทธิภาพของ อาร์มคอมไพเลอร์กับการปรับปรุงประสิทธิภาพทางด้านความเร็วและพลังงาน

จากการทดลองแสดงให้เห็นว่าวิธีการปรับปรุงประสิทธิภาพทางด้านความเร็วและลดการใช้พลังงาน ที่เรานำมาใช้ร่วมกันในการปรับปรุงการทำงานของการเข้ารหัสแบบ อาร์เอสเอ สามารถปรับปรุงทำให้การเข้ารหัสแบบอาร์เอสเอ ให้ทำงานได้เร็วขึ้นกว่าเดิมประมาณ 7.19-11.72 %

2. การทดลองผลการทดลองและวิเคราะห์ผลในแง่ของการใช้พลังงาน โดยเมื่อเรานำการปรับปรุงประสิทธิภาพในการเพิ่มความเร็วและ ลดการใช้พลังงานมาทำการทดลองร่วมกัน โดยจะใช้วิธีการทดลองเหมือนกับวิธีการทดลองในเรื่องการลดการใช้พลังงาน ตามที่ได้กล่าวไว้ในข้อ 4.1.2 ซึ่งเมื่อได้นำโปรแกรมการเข้ารหัสแบบอาร์เอสเอ ก่อนทำการปรับปรุงประสิทธิภาพ และหลังทำการปรับปรุงประสิทธิภาพ ทั้งเรื่องการเพิ่มความเร็วและการลดการใช้พลังงานมาทดลองตามข้อ 4.1.2 จะได้ผลการทดลองดังตารางที่ 4.7

| Test Case | Original | Optimize | Increst % |
|---|---|---|---|
| จำนวนเฉลี่ย การเข้ารหัสที่ทำได้ (ครั้ง) | 49,543 | 52,901 | 6.351 |

**ตารางที่ 4.7** จำนวนครั้งในการเข้ารหัสที่ใช้พลังงานในการเข้ารหัสไป 14.28 % ก่อนและหลังการปรับปรุงประสิทธิภาพด้านความเร็วและพลังงาน

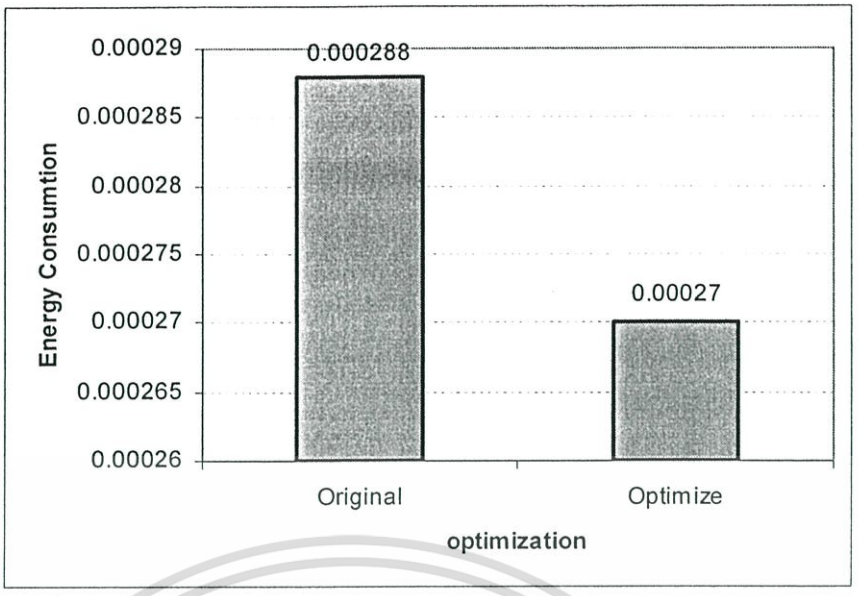
จากตารางที่ 4.7 แสดงให้เห็นว่า ในการใช้พลังงานใน แบตเตอรี่ 14.28% ด้วยโปรแกรมการเข้ารหัสอาร์เอสเอ เดิม ด้วย รหัส 1024 บิต สามารถทำการเข้ารหัสได้ 49,543 ครั้ง แต่หลังจากทำการปรับปรุงประสิทธิภาพทางด้านพลังงานและความเร็วแล้วสามารถทำงานได้มากขึ้นโดยใช้พลังงานเท่าเดิมคือราว 14.28% โดยทำงานเพิ่มขึ้นเป็นสามารถทำงานได้ 59,901 ครั้งในการใช้พลังงาน 14.28% เท่าเดิมโดยส่วนต่างที่เพิ่มขึ้นเป็นจำนวน 3,358 ครั้งคิดเป็น6.351%

จากนั้นเราจะทำการหาค่าพลังงานที่ใช้ในการเข้ารหัสแบบ อาร์เอสเอ ด้วยรหัส 1024 บิต 1 ครั้งออกมาได้โดยจำนวนพลังงานที่ใช้ไป / จำนวนการทำงานที่ทำได้ จะได้ว่าพลังงานที่ใช้ไปใน 1 ครั้งใช้ไปเท่าไร ดังตารางที่ 4.8

| Test Case | Original | Optimize | Reduce % |
|---|---|---|---|
| ปริมาณพลังงานที่ใช้ไปในการเข้ารหัส(%) | 0.000288 | 0.000270 | 6.25 |

**ตารางที่ 4.8** พลังงานเฉลี่ยที่ใช้ไปในการเข้ารหัสต่อครั้งก่อนและหลังการปรับปรุงประสิทธิภาพด้านความเร็วและพลังงาน

จะเห็นว่าเมื่อปรับปรุงทั้งความเร็วและพลังงานร่วมกันแล้วจะยิ่งช่วยทำให้ลดปริมาณการใช้พลังงานดังจะเห็นได้จากตารางที่ 4.8 จะเห็นว่า สามารถลดการใช้พลังงานได้มากกว่า 6.25 % ซึ่งมากกว่า การปรับปรุงประสิทธิภาพทางด้านพลังงานอย่างเดียวถึงเท่าตัวแสดงให้เห็นได้ว่า การปรับปรุงประสิทธิภาพทางด้านความเร็ว ก็มีผลทำให้การใช้พลังงานลดลงอย่างมากทั้งเนื่องมาจากการลดการเรียกใช้ฟังก์ชัน ก็เป็นส่วนช่วยการลดการใช้หน่วยความจำเพราะต้องมีการ เก็บ ค่าลงหน่วยความจำก่อนทำงานและ ดึงค่ากลับมาหลังทำงาน การลด Branch prediction ซึ่งช่วยให้ไปป์ไลน์ ให้โปรเซสเซอร์ทำงานได้ดียิ่งขึ้นดังรูปที่ 4.10




**รูปที่ 4.10** เปอร์เซ็นต์ของการบริโภคพลังงานเปรียบเทียบก่อนและหลังการปรับปรุงประสิทธิภาพ ทั้งความเร็วและพลังงานร่วมกัน

# บทที่ 5
# สรุป

## 5.1 สรุปงานวิจัยที่นำเสนอ

วิทยานิพนธ์นี้ได้ทำการเพิ่มประสิทธิภาพ ของการเข้ารหัสแบบอาร์เอสเอ ที่ทำงานบนอาร์มโปรเซสเซอร์ โดยได้ทำการทดลองโดยสุ่มใช้รหัส ความยาวรหัสขนาด 1024 บิต ที่ทำงานบนโทรศัพท์เคลื่อนที่ โดยที่โพรเซสเซอร์มีความสามารถในการประมวลผลต่ำและมีแบตเตอรี่ที่จำกัด จากนั้นเราจะทำการเพิ่มประสิทธิภาพความเร็วในการทำงาน และลดการใช้พลังงานในการทำงานลง โดยได้เสนอวิธีเพิ่มประสิทธิภาพซึ่งนำมาแก้ไขการเข้ารหัสแบบอาร์เอสเอ บนอาร์มโปรเซสเซอร์ ดังนี้

**การเพิ่มประสิทธิภาพด้านความเร็วในการทำงาน**

- การทำกำจัดความซ้ำซ้อน คือการแทนการทำงานที่ซ้ำซ้อนโดยรวมไปไว้ที่จุดเดียวกันเพื่อทำครั้งเดียวหรือเพื่อให้เกิดการทำงานที่มีประสิทธิภาพ

- การทำการลดจำนวนของ Branch prediction การลดจำนวนของ Branch prediction โดยการทำControl flow transformation ซึ่งเป็นสิ่งที่สำคัญมากในการทำงานของระบบไปป์ไลน์ นั้นก็คือถ้าหากเราทำการลดจำนวนของ Branch prediction ได้ก็จะทำให้ความเร็วในการทำงานเพิ่มขึ้น

- การทำการลดจำนวนการเรียกใช้ฟังก์ชัน ในการเรียกใช้ฟังก์ชัน โปรเซสเซอร์จะสูญเสียการทำงานส่วนหนึ่งไปกับ การเก็บค่าต่างๆไว้ในสแต็ค เพื่อเก็บค่าและนำค่าต่างๆมาใช้ก่อนและหลังการเรียกใช้ฟังก์ชัน ซึ่งการลด การเรียกใช้ฟังก์ชัน ซึ่งมีผลต่อประสิทธิภาพของไปป์ไลน์และการใช้หน่วยความจำ แต่การที่เราลดการเรียกใช้ฟังก์ชัน จะทำให้ขนาดของโปรแกรมใหญ่ขึ้น ดังนั้นการที่เราจะลดการเรียกใช้ฟังก์ชัน ควรที่จะใช้กับฟังก์ชัน ที่มีการเรียกใช้งานบ่อยครั้ง

- การลดจำนวนตัวแปรสำหรับเรียกใช้ฟังก์ชัน จากงานวิจัย [1],[15] ทำให้เราทราบว่าเมื่อมีการเรียกใช้ฟังก์ชัน และมีการส่งตัวแปรให้กับฟังก์ชัน ถ้าจำนวนตัวแปรมากกว่า 4 ตัวจะทำให้โปรแกรมจะต้องทำการเก็บค่าตัวแปร ลงไว้ในหน่วยความจำจากนั้นจึงจะส่งค่าไปใช้งานได้ ถ้าหากเราสามารถลดตัวแปรที่ส่งให้กับฟังก์ชัน ได้ไม่เกิน 4 ตัวก็จะช่วยให้โปรแกรมสามารถทำงานได้เร็วขึ้นและยังช่วยลดการใช้หน่วยความจำลงอีกด้วย

**การเพิ่มประสิทธิภาพด้านลดการใช้พลังงาน**

- การทำ Minimizing Buffer Allocation ในการพัฒนา C++ ในฟังก์ชัน จะมีบ่อยครั้งที่ใช้บัฟเฟอร์ในการเก็บผลก่อนในระหว่างการคำนวณ จากที่สังเกตดูบัฟเฟอร์พวกนี้จะ กำหนดแบบ local แล้ว assign(written) ต่อมาจะเอาค่าไปใช้ (read) ท้ายสุดทำลายทิ้ง (deallocate) เมื่อ ฟังก์ชันถูกเรียกทำงาน ตำแหน่งหน่วยความจำจะถูกกำหนดให้ในตำแหน่งที่แตกต่างกันในแต่ล่ะครั้ง ซึ่งเป็นปัญหาที่ยากในการจัดการกับแคชข้อมูลที่มีขนาดจำกัด โดยวิธีของเราในการแก้ปัญหานี้โดยการประกาศบัฟเฟอร์ข้อมูลเป็น Attribute ของคลาส ทำให้สามารถเข้าถึงจากสโคปที่ใหญ่ขึ้น โดยผลที่ได้จากที่บัฟเฟอร์ข้อมูลถูกจองเพียงครั้งเดียว แล้วจึงนำมาใช้มันอีกเรื่อยๆเป็นผลให้ปัญหาการค้นข้อมูลไม่พบในแคชลดลง

- การ Reused Variable เมื่อเรามีการสร้างตัวแปรและมีการจองหน่วยความจำให้กับตัวแปรจะมีการจองพื้นที่ในหน่วยความจำให้กับตัวแปรนั้นๆ ซึ่งจะทำให้เกิดการเปลี่ยนแปลงข้อมูลขึ้นในแคช ซึ่งหากว่าเราสามารถเปลี่ยนการจองในหน่วยความจำ ให้กับตัวแปรหลายๆ มาใช้ตัวใดตัวหนึ่งร่วมกันได้ก็จะช่วยทำให้การค้นข้อมูลไม่พบในแคช ลดลงเนื่องจากหน่วยความจำ ที่อ้างอิงถึงมีโอกาสที่จะอยู่ในแคชอยู่แล้ว เพิ่มมากขึ้นอีกด้วยเช่นกัน

ซึ่งจากผลการทดลองนี้แสดงให้เห็นว่า จากการเพิ่มประสิทธิภาพเพิ่มความเร็วและลดการบริโภคพลังงาน ได้ผลที่ดีทั้งในด้านความเร็ว เพิ่มขึ้นจากเดิม 7.19-11.72 % และในด้านการลดพลังงานสามารถลดได้ถึง 6.25% ถึงแม้เราจะใช้วิธีการเพิ่มประสิทธิภาพกับการปรับปรุงของคอมไพล์เลอร์ เช่น ARMCpp ที่ระดับ 2 ก็ยังมีประสิทธิภาพอยู่

## 5.2 ปัญหาและอุปสรรค

ในปัจจุบันยังไม่มีโปรแกรมใดๆ ที่มีความสามารถในการที่จะใช้วัดความเร็วและการใช้พลังงานของโปรแกรม อื่นๆบนโทรศัพท์มือถือ จึงเป็นความยากลำบากที่จะวัดประสิทธิภาพของโปรแกรมต่างๆ ที่ทำงานบนโทรศัพท์มือถือว่ามีการพัฒนาอย่างมีประสิทธิภาพหรือไม่ทำให้เราต้องนำโค้ดของโปรแกรมต่างๆ ที่เราต้องการจะวัดประสิทธิภาพมา วัดประสิทธิภาพด้านความเร็วโดยใช้ ARM developer suite โดยจำเป็นที่จะต้อง ปรับปรุงโปรแกรมที่ทำงานบนโทรศัพท์มือถือหรือ คอมพิวเตอร์พกพา มาปรับปรุงเพื่อให้ทำงานบนคอมพิวเตอร์ทั่วไป เพื่อจะวัดประสิทธิภาพด้านความเร็วและ ทางด้านการใช้พลังงาน เราก็จะต้องพัฒนาโค้ด โปรแกรมต่อยอดจากโปรแกรมที่เราต้องการจะวัดพลังงาน เพื่อที่จะสามารถวัดได้ว่าโปรแกรม นั้นๆ ใช้พลังงานปริมาณมากน้อยเพียงใดในการทำงาน

**5.3 แนวทางการพัฒนาต่อ**

ในอนาคตเราจะสามารถนำ โปรแกรมและส่วนประกอบต่างๆที่ได้พัฒนาขึ้นมา เพื่อวัดความเร็วและการใช้พลังงานของการเข้ารหัสแบบอาร์เอสเอ บนอาร์มโปรเซสเซอร์นี้ มารวมกันเป็นโปรแกรม ที่สามารถติดตั้งลงไปบนโทรศัพท์มือถือหรือ คอมพิวเตอร์เคลื่อนที่ เพื่อที่จะใช้วัดอัตราการใช้พลังงานของโปรแกรม อื่นบนโทรศัพท์มือถือได้อย่างสะดวกมากยิ่งขึ้น

# บรรณานุกรม


[1] K. Ramkishor and V. Gunashree, **Real time implementation of MPEG-4 video decoder on ARM7TDMI**, in IEEE Proceedings of International Symposium on Intelligent Multimedia, Video and Speech Processing, pp. 3653, 2001.

[2] Anand Raghunathan, Senior Member, IEEE, and Niraj K. Jha, Fellow, IEE Nachiketh R. Potlapally, Student Member, IEEE, Srivaths Ravi, Member, IEEE, **A Study of the Energy Consumption Characteristics of Cryptographic Algorithms and Security Protocols**, IEEE Transactions on mobile compuing, vol.5 no.2, Fabuary 2006

[3] Alfred J. Menezes, **Handbook of Applied Cryptography**, October 1996

[4] ARM Limited, 20 April 2001, **ARM920T (Rev1) Technical Reference Manual**

[5] R. Gonzales and M. Horowitz, **Energy dissipation in general-purpose microprocessors**. IEEE J. of Solid-state Circ., SC-31(9):1277-1283, 1996

[6] ARM Limited, 21 Nov 2001, **ARM Developer Suite – Version 1.2 – AXD and armsd Debuggers Guide**

[7] J. Bormans, K. Denolf, S. Wuytack, L. Nachtergaele, I. Bolsens, **Integrating system-level low power methodologies into a real-life design flow**, PATMOS'99 Ninth International Workshop Power and Timing Modeling, Optimization and Simulation, pp. 19-28, Oct 6-8, 1999

[8] ARM Limited, 21 Nov 2001, **ARM Developer Suite - Version 1.2 -Debug Target Guide**

[9] ARM Limited, **ARM Programming Technique**

[10] Michael J. Flynn, **Computer Architecture Pipelined and Parallel Processor Design**

[11] L. Edwards, R Barker, and Staff of EMCC Software Ltd, **Developing Series 60 Applications A Guide for Symbian OS C++ Developers**

[12] ARM Limited, January 1998, **Writing Efficient C for ARM**

[13] http://www.arm.com/

[14] http://www.symbian.com/

[15] http://ecos.sourceware.org/docs-1.3.1/ref/gnupro-ref/arm/ARM_COMBO_ch01.html

[16] http://www.win.tue.nl/~klenstra/

[17] Wen-Tsong Shiue, Chaitali Chakrabarti, **Memory Exploration for Low Power, Embedded Systems**, Journal of VLSI Signal Processing Systems, pp. 167-178, Nov, 2001

# ภาคผนวก ก วิธีการใช้งาน ARM Complier

วิธีการใช้งาน ARM Complier

เราสามารถ กำหนดให้ Arm Complier คอมไพล์ โปรแกรมของเราได้ด้วยคำสั่ง

- ARMCPP สำหรับ Source code ที่เป็น ภาษา C++
- ARMCC สำหรับ Source code ที่เป็น ภาษา C

โดยมี Option ดังนี้

ARM C++ Compiler, ADS1.2 [Build 805]

Usage: armcpp [options] file1 file2 ... filen

Main options:

-c Do not link the files being compiled

-C Prevent the preprocessor from removing comments (Use with -E)

-D <symbol> Define <symbol> on entry to the compiler

-E Preprocess the C source code only

-f <options> Enable a selection of compiler defined features

-g <options> Generate tables for high-level debugging

-I <directory> Include <directory> on the #include search path

-J <directory> Replace the default #include path with <directory>

-o <file> Name the file that holds the final output of the compilation

-O0 Minimum optimization

-O1 Restricted optimization for debugging

-O2 Maximum optimization

-S Output assembly code instead of object code

-U <symbol> Undefine <symbol> on entry to the compiler

-W <options> Disable all or selected warning messages

-cpu processor for process program

รูป ก.1 parameter สำหรับ ARM COMPLIER

ตัวอย่างการใช้งาน

```
C:\> armcpp -c -g -O0 HELLOWORLD.cpp -cpu ARM920T
```

หมายความว่า

- คอมไพล์ ให้สามารถ Debug Source code ได้ (-g)
- ด้วยระดับการออพติไมซ์ 0 (-O0)
- คอมไพล์ file HELLOWORLD.cpp
- คอมไพล์ เพื่อนำไปใช้งานกับโปรเซสเซอร์ ARM920T

# ภาคผนวก ข วิธีการใช้งาน ARMULATOR

เมื่อเราคอมไพล์โค้ดโปรแกรมของเราด้วย ARM Complier แล้ว จะได้ Object file ซึ่งจะต้องทำการ Link Object file นี้ให้ไปเป็นBinary file เพื่อให้ ARMULATOR สามารถนำไปจำลองการทำงานได้ด้วยคำสั่ง ARMLINK

โดยมี Option ดังนี้

ARM Linker, ADS1.2 [Build 805]

Usage: ARMLINK option-list input-file-list

where

option-list is a list of case-insensitive options.

input-file-list is a list of input object and library files.

Main options (abbreviations shown capitalised):

General options:

| | |
|---|---|
| -Help | Print this summary. |
| -Output file | Specify the name of the output file. |

Options for specifying memory map information:

| | | |
|---|---|---|
| -partial | | Generate a partially linked object. |
| -scatter | file | Create the memory map as described in file. |
| -ro-base | n | Set exec addr of region containing RO sections. |
| -rw-base | n | Set exec addr of region containing RW/ZI sections. |

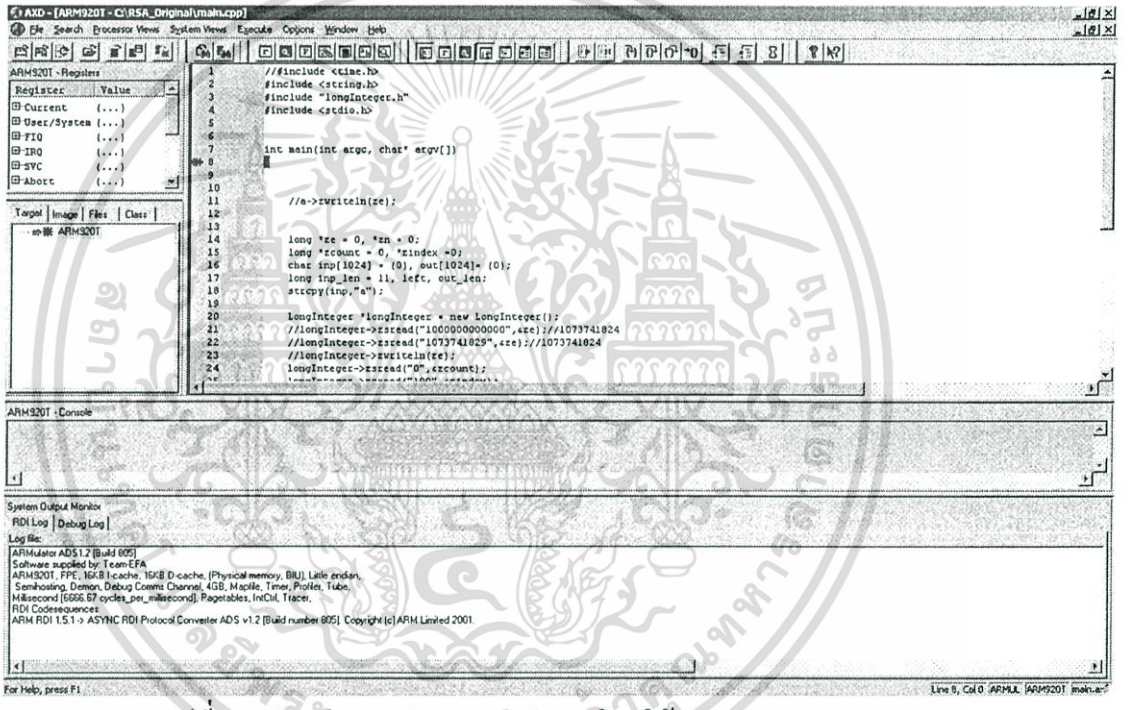
Software supplied by: Team-EFA

รูป ข.1 parameter สำหรับ ARM Linker

ตัวอย่างเช่นคำสั่ง C:\>ARMLINK HELLOWORLD.o -o HELLOWORLD.axf

หมายความว่า

- เมื่อคอมไพล์โค้ดโปรแกรมจะได้ file Object (นามสกุล o)
- ทำการ Link Object เข้าด้วยกัน เนื่องจากหากเรามี file สำหรับงานของเราหลาย file และแต่ละ file เมื่อนำไปคอมไพล์ แล้วได้ file Object หลายตัวเราจะต้องทำการนำ file Object เหล่านี้มารวมกันเป็นโปรแกรมที่สามารถทำงานได้ (HELLOWORLD.o)
- กำหนด Output file (-o) เมื่อทำการ Link file Object แล้วจะได้ file นามสกุล axf ซึ่งจะสามารถนำไปจำลองการทำงานบน ARMULATOR ได้ดังรูปที่ ข.1



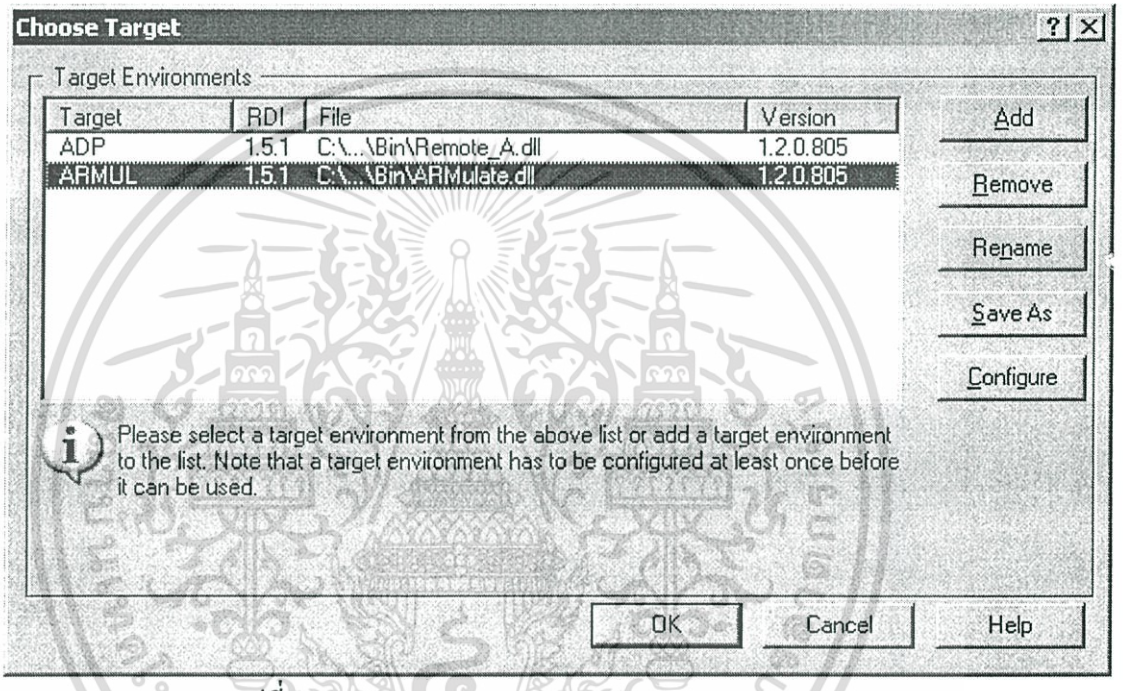

**รูปที่ ข.1** ตัวอย่างการจำลองการทำงานโดยใช้ ARMULATOR

เราสามารถกำหนดค่า Configure ต่างๆที่เราใช้ในการทดลองให้กับ ARMULATOR ได้ดังนี้

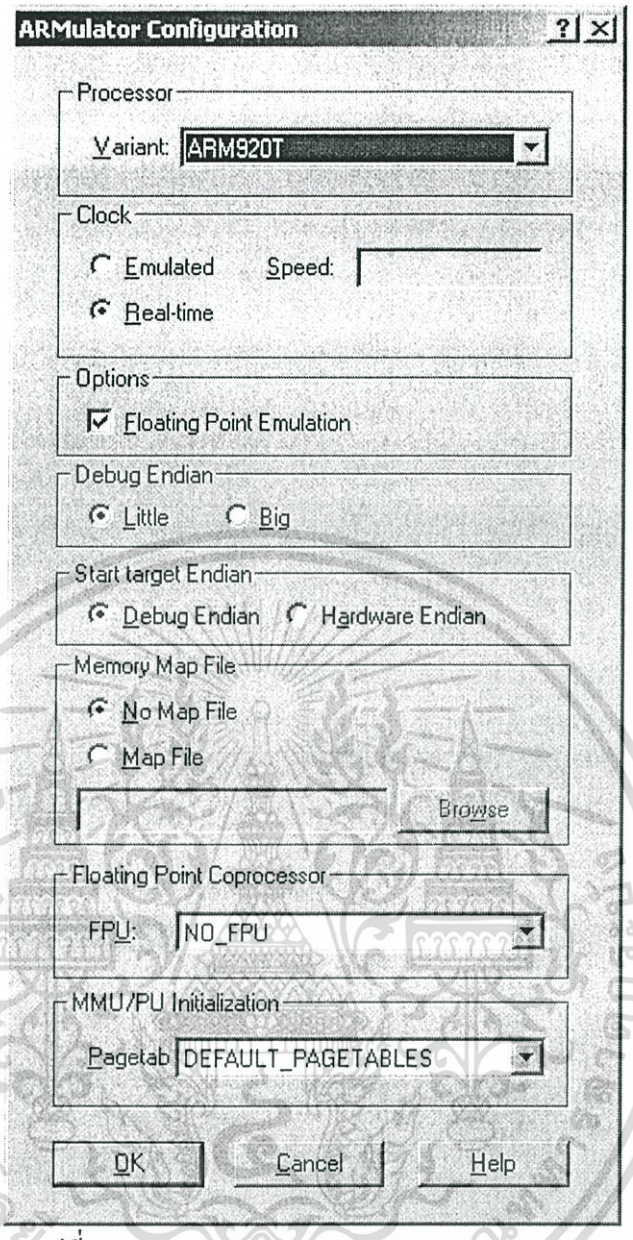
- กำหนดค่า CPU

ที่เมนู Option > Configure Target…

จะปรากฏหน้าต่างดังรูปที่ ข.2 และเราจะเลือกทำการ Configure ARMULATOR โดยเลือก ที่ ARMUL และกดปุ่ม Configure จะปรากฏหน้าต่างดังรูปที่ ข.3




**รูปที่ ข.2** การ Configure CPU ARMULATOR (1)




**รูปที่ ข.3** การ Configure CPU ARMULATOR (2)

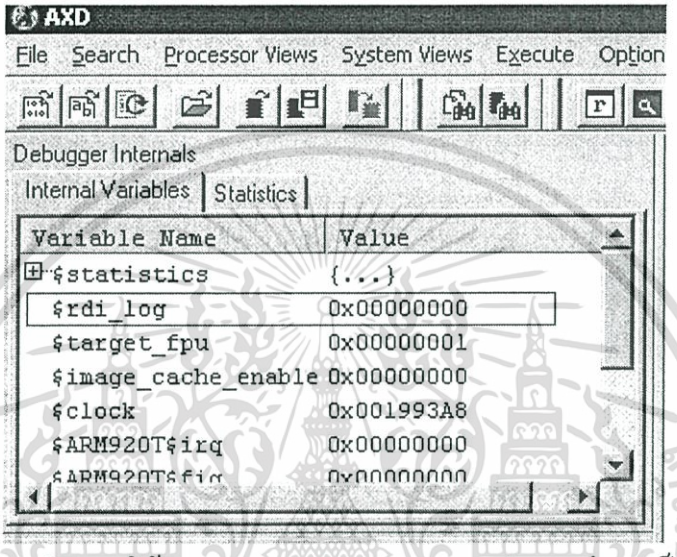
โดยเราจะสามารถ Configure

- ชนิดโปรเซสเซอร์
- Clock ที่ใช้ในการจำลองการทำงาน
- Option Floating Point Emulation
- Debug Endian
- Start target Endian
- Memory Map File
- Floating Point Coprocessor
- MMU/PU Initialization

- กำหนดให้ ARMULATOR TRACE การทำงาน ต่างๆ เป็น LOG FILE

เราจะสามารถสั่งให้ ARMULATOR เริ่ม TRACE การทำงาน ต่างๆ เป็น LOG FILE ณ จุดใดจุดหนึ่งหรือ หยุด TRACE การทำงาน ต่างๆ เป็น LOG FILE ได้ดังนี้

แสดงหน้าต่าง Debug Internal โดยเลือกเมนู System View > Debug Internal จะปรากฏหน้าต่างดังรูปที่ ข.4

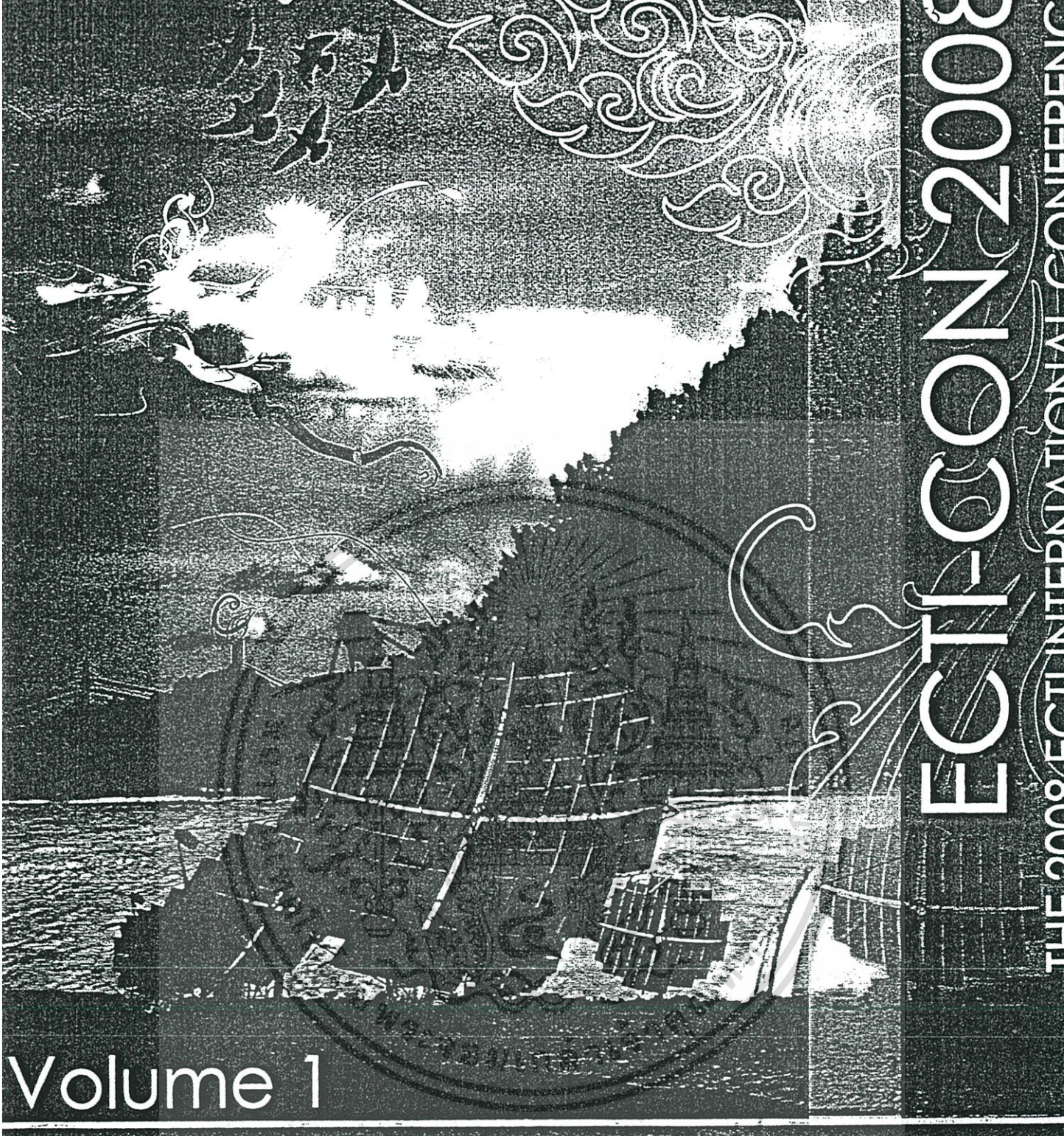



**รูปที่ ข.4** กำหนดให้ ARMULATOR TRACE การทำงาน ต่างๆ เป็น LOG FILE

จากนั้นเมื่อถึงจุดที่เราต้องการเริ่ม trace การทำงานของอาร์มโปรเซสเซอร์ เราจะต้องกำหนดค่าให้ $rdi_log เป็น 0x00000010 และเมื่อเราต้องการหยุดการ trace การทำงาน เราก็ต้องกำหนดค่าให้ $rdi_log เป็น 0x00000000 จะเป็นการหยุดการ trace โดย trace file ที่ได้ จะชื่อ armul.trc ซึ่งจะอยู่ที่ทำการจำลองการทำงานของโค้ดโปรแกรมนั้นๆ

# ผลงานวิจัยในระหว่างการศึกษาที่ได้รับการตีพิมพ์เผยแพร่

1. P Tyoviriyakul and S Kittitornkun, "Performance Optimization of RSA Encryption for ARM Processor", Electrical Engineering/Electronics, Computer, Telecommunications, and Information Technoly Conference 27, Thailand 2008



# ECTI-CON2008

THE 2008 ECTI INTERNATIONAL CONFERENCE

## Volume 1

Proceedings of the 2008 Electrical Engineering/ Electronics, Computer
Telecommunications and Information Technology (ECTI)
International Conference

May 14-17, 2008
Maritime Park and Spa Resort, Krabi, THAILAND

ECTI Association

TRIDI
Telecommunications Research and Industrial Development Institute

NECTEC
a member of NSTDA

Seagate

TNGC

IEEE THAILAND SECTION

# Optimizing RSA Encryption for ARM Microprocessor

Pitcha Tyoviriyakul and Surin Kittitornkun
Faculty of Engineering, Dept. of Computer Engineering,
King Mongkut's Institute of Technology Ladkrabang
Bangkok, Thailand 10520
Email: peach_mail99@yahoo.com, kksurin@kmitl.ac.th

## Abstract

*Most compiler optimization techniques concern most about speed. In this paper, we present two high-level memory optimization methods for ARM-based secure applications on mobile phones, pocket PCs, etc. The experiments using RSA encryption on ARM920T with 1024-bit random public keys show that the proposed techniques can complement the existing speed-oriented ones to achieve less number of memory accesses, shorter execution time, and lower memory allocations to all ARM C++ optimization levels despite the 16-KB instruction and 16-KB data caches of ARM 920T core.*

**Key Words:** Optimization memory, ARM Processor

## 1. Introduction

Security on mobile device become commonplace for mobile smart phones, PDAs and other other portable devices. However, an Encryption software on low-power RISC processors still requires significant amount of computational as well as battery power. The most popular RISC processor in these portable systems is ARM because of its low power consumption and smaller size. Especially in mobile smart phones, the amount of processing power available is limited and has direct impact on the battery life.

Therefore, memory optimization of RSA Encryption is an essential task for achieving effective low power. However, most commercial compiler optimization techniques concern most about speed. This paper describes two simple and effective memory optimization techniques for RSA Encryption on ARM Microprocessor.

## 2. Literature Review

### 2.1 Memory Optimization

Security systems and applications intrinsically have a high memory and computation costs, making the design of high performance, low-power solutions a real challenge.

### 2.2 ARM9 Architecture

ARM9 architecture is targeted at low-power 32 bit RISC processor for battery-powered smart phones and PDAs. To maximize instruction throughput, the integer pipeline consists of five stages: fetch, decode, execute, memory access and write register back. We picked the ARM920T because it is used in many mobile smart phones such as Nokia phones with Symbian Series 60 user interface. The ARM920T is processor macrocell combining an ARM9TDMI™ processor core with:

- 16KB instruction and 16KB data caches
- Instruction and data Memory Management Units (MMUs)
- Write memory
- An AMBA™ (Advanced Microprocessor Bus Architecture) bus interface
- 32bit data and instruction bus transferred.
- An Embedded Trace Macrocell (ETM) interface.

Both of instruction and data cache are 64-way set associative [3].

### 2.3 RSA Encryption

RSA is an algorithm for public-key cryptography. It was the first algorithm known to be suitable for signing as well as encryption, and one of the first great advances in public key cryptography. RSA is widely used in electronic commerce protocols, and is believed to be secure given sufficiently long.

#### 2.3.1 Operation:

RSA involves a public key and a private key. The public key can be known to everyone and is used for encrypting messages. Messages encrypted with the public key can only be decrypted using the private key. The keys for the RSA algorithm are generated the following way

1. Choose two distinct large random prime numbers $p$, $q$
2. Compute $n = p\,q$
3. Compute $\varphi(n) = (p-1)(q-1)$.
4. Choose an integer $e$ such that $1 < e < \varphi(n)$, and $e$ and $\varphi(n)$ share no factors other than 1
5. Compute $d$ to satisfy the congruence relation $de \equiv 1 \pmod{\varphi(n)}$

The **public key** ($n$, e) consists of the modulus $n$ and the public exponent $e$.
The **private key** ($d$, e) consists of the modulus $n$ and the private exponent $d$ which must be kept secret.



978-1-4244-2101-5/08/$25.00 ©2008 IEEE

We can interpret the trace files using our analysis program, *Analize.exe*, to extract information about memory bus accesses and cache events. Figure 3 shows some part of armul.trc.

```
Date: Sun Jan 07 19:47:11 2008
Source: Armul
Options: Trace Events

E 00008000 00000000 10005
BSR4O___ A0000000 00000C1E
IT 00008000 e28f8090 ADD r8,pc,#0x90 ; #0x8098
BSR8O___ 00008088 E2555004 24847004
```

**Figure 3** Excerpt of file "armul.trc"

We will show how to interpret memory bus accesses and special events.

1) Memory Bus Trace (B lines): The lines started with B indicate memory bus accesses. They have the following format for general memory accesses

B<type><rw><size>[O][L][S] <address> <data>
Where:
<type> indicates the cycle type:
- S sequential
- N nonsequential.

<rw> indicates either a read or a write operation:
- R read
- W write

<size> indicates the size of the memory bus access:
- 4 word (32 bits)
- 2 halfword (16 bits)
- 1 byte (8 bits)

O indicates an opcode fetch (instruction fetch).
L indicates a locked access (SWP instruction).
S indicates a speculative instruction fetch.
<address> gives the address in hexadecimal format, for example 00008008.
<data> can show one of the following:
*value* gives the read/written value, for example EB00000C

2) Events Trace (E lines): The format of the event (E) lines is as follows:
E <word1> <word2> <event_number>
where:
<word1> gives the first of a pair of words, such as the pc value.
<word2> gives the second of a pair of words, such as the aborting address.
<event_number> gives an event number, for example 0x10001. This is MMUEvent_DLineFetch.

## 4.2 Results and Discussions

We can evaluate the performance of our methods using the following parameters: instruction, memory accesses, execution time and the amount of energy reduced.

### 4.2.1 Instruction:

Table 1 show. Average of total instruction number for RSA Encryption 1024 bits combine instruction take and instruction skip (Lv.0 our optimize reduce instruction 1.07% and LV.2 our optimize reduce instruction 1.43%)

| LV. | Opt 0 | Ours+ Opt 0 | Opt 2 | Ours+ Opt 2 |
|---|---|---|---|---|
| Instruction | 114,257,379 | 113,034,816 | 81,288,697 | 80,122,267 |

**Table 1** Number of total instructions on RSA 1024 bits average from 10 random public key 1024 bits

### 4.2.2 Memory Accesses:

Figure 4 shows the average total number memory accesses for each optimization on RSA Encryption. It shows that our optimization methodologies can effectively reduce memory accesses more than those of ARMcpp optimize level 2. (Lv.0 our optimize reduce access memory 1.48% and LV.2 our optimize reduce access memory 2.18%)

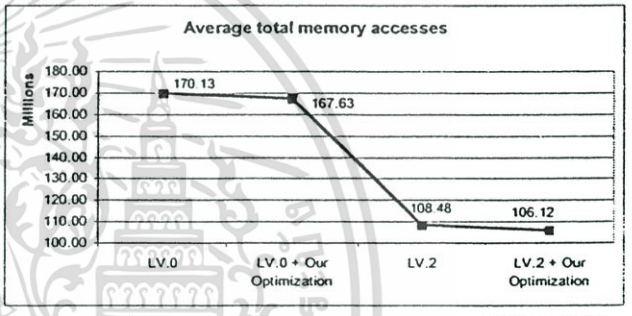



**Figure 4** Average total memory accesses of RSA 1024 bits for each optimization

### 4.2.3 Execution Time:

Figure 5 shows the average execution time of RSA Encryption at different optimization methods. It can be noticed that our methods work well with the commercial compiler. (Lv.0 our optimize reduce execution time 1.31% and LV.2 our optimize reduce execution time 1.92%)

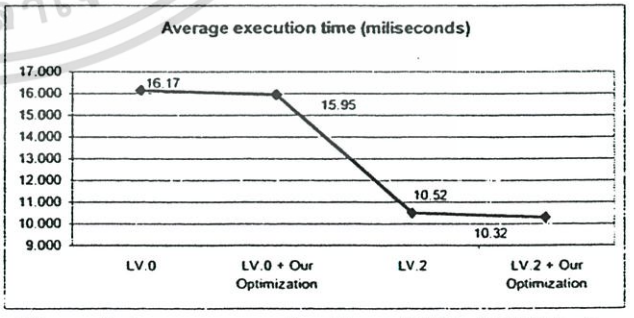



**Figure 5** Average execution time of RSA 1024 bits for each optimization

# ประวัติผู้เขียน

| | |
|---|---|
| ชื่อ-นามสกุล | นาย พิชชา เตียววิริยะกุล |
| ประวัติการศึกษา | สำเร็จการศึกษาระดับปริญญาตรี หลักสูตรวิศวกรรมศาสตรบัณฑิต สาขาวิศวกรรมคอมพิวเตอร์ จากคณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง ปีการศึกษา 2545 |
| งานวิจัยที่สนใจ | Mobile Computing, Computer Architecture และ Multimedia |