การบีบอัดข้อมูลสำหรับระบบเครือข่ายเซนเซอร์ไร้สาย

Data Compression for Wireless Sensor Networks

ภควัฒน์ ติณสิริสุข

Pakawat Tinsirisuk

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญา
วิศวกรรมศาสตรมหาบัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์
มหาวิทยาลัยสงขลานครินทร์

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Computer Engineering
Prince of Songkla University
2559

การบีบอัดข้อมูลสำหรับระบบเครือข่ายเซนเซอร์ไร้สาย

Data Compression for Wireless Sensor Networks

ภควัฒน์ ติณสิริสุข

Pakawat Tinsirisuk

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญา
วิศวกรรมศาสตรมหาบัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์
มหาวิทยาลัยสงขลานครินทร์

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Computer Engineering
Prince of Songkla University
2559

ชื่อวิทยานิพนธ์ การบีบอัดข้อมูลสำหรับเครือข่ายเซนเซอร์ไร้สาย
ผู้เขียน นาย ภควัฒน์ ติณสิริสุข
สาขาวิชา วิศวกรรมคอมพิวเตอร์

อาจารย์ที่ปรึกษาวิทยานิพนธ์หลัก

.....
(ผู้ช่วยศาสตราจารย์ ดร.วรรณรัช สันติอมรทัต)

คณะกรรมการสอบ

.....ประธานกรรมการ
(ดร.ปัญญยศ ไชยกาฬ)

.....กรรมการ
(ดร.ชนันท์กรณ์ จันแดง)

.....กรรมการ
(ผู้ช่วยศาสตราจารย์ ดร.วรรณรัช สันติอมรทัต)

บัณฑิตวิทยาลัย มหาวิทยาลัยสงขลานครินทร์ อนุมัติให้นับวิทยานิพนธ์ฉบับนี้เป็นส่วนหนึ่งของการศึกษา ตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์

.....
(รองศาสตราจารย์ ดร.ธีระพล ศรีชนะ)
คณบดีบัณฑิตวิทยาลัย

ขอรับรองว่า ผลงานวิจัยนี้มาจากการศึกษาวิจัยของนักศึกษาเอง และได้แสดงความขอบคุณบุคคลที่มีส่วนช่วยเหลือแล้ว

ลงชื่อ.....
(ผู้ช่วยศาสตราจารย์ ดร.วรรณรัช สันติอมรทัต)
อาจารย์ที่ปรึกษาวิทยานิพนธ์

ลงชื่อ.....
(ภควัฒน์ ติณสิริสุข)
นักศึกษา

ข้าพเจ้าขอรับรองว่า ผลงานวิจัยนี้ไม่เคยเป็นส่วนหนึ่งในการอนุมัติปริญญาในระดับใดมาก่อน และไม่ได้ถูกใช้ในการยื่นขออนุมัติปริญญาในขณะนี้

ลงชื่อ.....
(ภควัฒน์ ติณสิริสุข)
นักศึกษา

ชื่อวิทยานิพนธ์ การบีบอัดข้อมูลสำหรับเครือข่ายเซนเซอร์ไร้สาย
ผู้เขียน นาย ภควัฒน์ ติณสิริสุข
สาขาวิชา วิศวกรรมคอมพิวเตอร์
ปีการศึกษา 2558

บทคัดย่อ

การบีบอัดข้อมูลบนระบบเครือข่ายเซนเซอร์ไร้สายเป็นหนึ่งแนวคิดที่ถูกพัฒนาขึ้นมาด้วยหลายเหตุผล ทั้งการประหยัดพลังงาน การเพิ่มขีดความสามารถในการรับส่งข้อมูล รวมถึงความปลอดภัยของข้อมูล การประยุกต์ใช้งานที่หลากหลายของระบบเครือข่ายเซนเซอร์ไร้สาย ทำให้อัลกอริทึมบีบอัดข้อมูลโดยทั่วไป ไม่มีความเหมาะสมที่จะนำมาใช้งานกับระบบเครือข่ายเซนเซอร์ไร้สาย

งานวิจัยนี้จึงมุ่งเน้นการพัฒนาอัลกอริทึมที่สามารถทำการบีบอัดข้อมูลได้หลากหลายประเภท เพื่อเพิ่มความยืดหยุ่น และมีส่วนของ Overhead น้อยที่สุด อัลกอริทึมที่ออกแบบขึ้นมานั้นจะประกอบด้วย DZ (Double Zero), SZ (Single Zero) และ SZAVG (Single Zero with Average) ทั้ง 3 อัลกอริทึมมีการทำงานที่คล้ายกัน แตกต่างกันเฉพาะในส่วนของการแบ่งวรรคข้อมูล และการคำนวณหารูปแบบการบีบอัดที่เหมาะสมเท่านั้น ทำให้สามารถพัฒนาโมดูลอื่นร่วมกันได้ เช่น ส่วนของการจัดการกับ Buffer เป็นต้น อัลกอริทึมที่พัฒนาขึ้นจะถูกนำมาสร้างเป็นวงจรบนเทคโนโลยี Field Programmable Gate Array (FPGA) เพื่อให้ทราบขนาดของวงจรบีบอัดข้อมูล และประเมินประสิทธิภาพในการทำงานเมื่อนำไปใช้งานในวงจรจริงได้

ข้อมูลสำหรับการทดสอบ ประกอบด้วย ข้อมูลอุณหภูมิ ข้อมูลความชื้นในอากาศ ข้อมูลคลื่นไฟฟ้าสมอง และข้อมูลรูปภาพ ผลจากการทดสอบพบว่าอัลกอริทึมที่นำเสนอในวิทยานิพนธ์นี้ สามารถทำการบีบอัดข้อมูลประเภทข้อมูลสัญญาณชีพ (คลื่นไฟฟ้าสมอง) และข้อมูลรูปภาพ ได้ดีกว่าอัลกอริทึมบีบอัดข้อมูลแบบ CoXoH ประมาณ 3.8 – 5.5 % อีกทั้งยังมีการใช้งานทรัพยากรฮาร์ดแวร์ที่น้อยกว่าประมาณ 5 เท่า สำหรับส่วนของ Slice LUTs และ 9 เท่า สำหรับ Register

คำสำคัญ: การบีบอัดข้อมูล, เครือข่ายเซนเซอร์ไร้สาย, FPGA

Thesis Title Data Compression for Wireless Sensor Networks
Author Mr. Pakawat Tinsirisuk
Major Program Computer Engineering
Academic Year 2015

ABSTRACT

Data compression algorithm on Wireless Sensor Network is developed in order to save energy, increase transmission rate and data security. Because of the varities of data type in WSNs, the original data compression algorithm is unsuitable.

This research represents a new compression algorithm which can compress the various data type for more flexibility and less overhead. The designed algorithms included DZ, SZ and SZAVG have the same concepts. However, they are different in term of data partitioning and suitable pattern. Thus, the buffer management is able to be implemented separately. The selected algorithm is designed and developed on FPGA to assess the size of circuitry and performance.

Tested data included humidity, temperature, electroencephalography (EEG) and picture are deployed. From the experimental result, we found that our proposed algorithm gives the best performance when it compresses the data likes EEG and image than CoXoH around 3.8 – 5.5% and consume less hardware resource than CoXoH 5 times for slice LUTs and 9 times for register.

KEYWORDS: Data Compression, Wireless Sensor Networks, FPGA

กิตติกรรมประกาศ

ขอขอบพระคุณ ผู้ช่วยศาสตราจารย์ ดร.วรรณรัช สันติอมรทัต อาจารย์ที่ปรึกษาวิทยานิพนธ์ ที่กรุณาเสียสละเวลา ให้คำปรึกษา ชี้แนะแนวทางในการดำเนินการวิจัย พร้อมทั้งสนับสนุนเอกสารข้อมูล และอุปกรณ์ที่เกี่ยวข้อง ตลอดจนตรวจสอบแก้ไขวารสาร และวิทยานิพนธ์เป็นอย่างดี จนกระทั่งวิทยานิพนธ์ชิ้นนี้สำเร็จลุล่วงสมบูรณ์

ขอขอบพระคุณ ดร.ปัญญยศ ไชยกาฬ ที่กรุณาเสียสละเวลาเป็นประธานกรรมการสอบวิทยานิพนธ์ พร้อมกับให้คำแนะนำที่เป็นประโยชน์อย่างยิ่งสำหรับวิทยานิพนธ์ ตลอดจนตรวจสอบแก้ไขวิทยานิพนธ์ให้มีความสมบูรณ์

ขอขอบพระคุณ ดร.ชนันท์กรณ์ จันแดง ที่กรุณาเสียสละเวลาเป็นกรรมการสอบวิทยานิพนธ์ และให้คำแนะนำที่เป็นประโยชน์อย่างยิ่งสำหรับวิทยานิพนธ์ ตลอดจนตรวจสอบแก้ไขวิทยานิพนธ์ให้มีความสมบูรณ์

ขอขอบพระคุณ บัณฑิตวิทยาลัย มหาวิทยาลัยสงขลานครินทร์ วิทยาเขตหาดใหญ่ที่ให้การสนับสนุนทุนในการทำวิจัย และให้ความช่วยเหลือด้านการประสานงานต่าง ๆ

ขอขอบพระคุณ คณะวิศวกรรมศาสตร์ มหาวิทยาลัยสงขลานครินทร์ ที่กรุณาให้ทุนสนับสนุนการเผยแพร่ผลงานวิจัยแก่ข้าพเจ้า

ขอขอบพระคุณ ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ มหาวิทยาลัยสงขลานครินทร์ ที่กรุณาให้คำปรึกษา ประสานงาน ตลอดจนอำนวยความสะดวกในการจัดทำวิยานิพนธ์ฉบับนี้จนลุล่วงสมบูรณ์

ขอขอบพระคุณ คณาจารย์ บุคลากร นักศึกษาปริญญาเอก นักศึกษาปริญญาโท ภาควิชาวิศวกรรมคอมพิวเตอร์ทุกคนที่ได้ช่วยเหลือ ให้คำปรึกษา และเป็นกำลังใจในการทำงานเป็นอย่างดีเสมอมา

และสุดท้าย ข้าพเจ้าขอน้อมรำลึกถึงพระคุณของบิดา มารดา และครอบครัวที่ส่งเสริม สนับสนุน ให้คำแนะนำ ให้คำปรึกษา ให้กำลังใจที่ดีเสมอมาแก่ข้าพเจ้าจนสำเร็จการศึกษา

ภควัฒน์ ติณสิริสุข

สารบัญ

สารบัญ (ต่อ)

หน้า

สารบัญ (ต่อ)

หน้า

รายการตาราง

หน้า

รายการภาพประกอบ

หน้า

รายการภาพประกอบ (ต่อ)

หน้า

สัญลักษณ์คำย่อและตัวย่อ

CR	Compression Ratio
DZ	Double Zero
DZLBS	Double Zero using Limited Buffer by Shift
FPGA	Field Programmable Gate Array
LB	Limited Buffer
LBS	Limited Buffer by Shift
LZ	Lempel-Ziv
RF	Radio Frequency
SZ	Single Zero
SZLBS	Single Zero using Limited Buffer by Shift
SZAVG	Single Zero with Average
SZAVGLB	Single Zero with Average using Limited Buffer
SZAVGLBS	Single Zero with Average using Limited Buffer by Shift
WSNs	Wireless Sensor Networks

บทที่ 1
บทนำ

1.1 ที่มาและความสำคัญของการวิจัย

ระบบเครือข่ายเซนเซอร์ไร้สาย (Wireless Sensor Network : WSNs) ประกอบด้วยโหนด (Node) จำนวนมากทำหน้าที่รับข้อมูลจากเซนเซอร์ และติดต่อสื่อสารกันในลักษณะเป็นเครือข่ายไร้สาย ปัจจุบันเทคโนโลยีเครือข่ายเซนเซอร์ไร้สายได้รับการพัฒนาเปลี่ยนไปจากเดิมในหลายส่วน ตัวอย่างเช่น โหนดมีขนาดเล็กลงแต่มีความสามารถในการประมวลผลมากขึ้น มีความสามารถส่งข้อมูลได้ไกลขึ้น และมีการพัฒนาให้ประหยัดพลังงานมากกว่าเดิม ซึ่งเป็นส่วนที่มีประโยชน์มาก เนื่องจากการที่โหนดใช้พลังงานน้อยลง ทำให้สามารถทำงานได้นาน ส่งผลโดยตรงให้ระบบเครือข่ายทำงานได้นานยิ่งขึ้น ดังนั้นพลังงาน จึงเป็นปัจจัยหลักในการทำงานของเครือข่ายเซนเซอร์ไร้สาย

พลังงานที่เสียไปในระบบเครือข่ายเซนเซอร์ไร้สายนั้นมีอยู่หลายส่วนด้วยกัน เช่น พลังงานที่ใช้ในการประมวลผล พลังงานที่สูญเสียไปในการใช้งานเซนเซอร์ และพลังงานที่สูญเสียไปในการรับส่งข้อมูลแบบไร้สายเป็นต้น ส่วนที่ใช้พลังงานมากที่สุด คือส่วนของการรับส่งข้อมูล ซึ่งอาจใช้พลังงานสูงถึง 80% ของพลังงานที่ใช้ทั้งหมด [1] ดังนั้นงานวิจัยส่วนใหญ่จึงได้เน้นให้มีการประหยัดพลังงานในการรับส่งข้อมูลมากขึ้น ตัวอย่างเช่น การปิดระบบสื่อสารแบบไร้สาย (RF Module) ขณะที่ไม่ได้ส่งข้อมูล หรือการส่งข้อมูลขณะที่ข้อมูลมีการเปลี่ยนแปลงเท่านั้น (ไม่ส่งข้อมูลตามระยะเวลา หรือ Sampling rate) ใช้ในกรณีที่ข้อมูลไม่ค่อยเปลี่ยนแปลงมากในสภาวะการณ์ปกติ เช่น อุณหภูมิ และความชื้นในอากาศเป็นต้น อีกวิธีที่น่าสนใจก็คือ การบีบอัดข้อมูล วิธีนี้จะทำให้ข้อมูลที่จะส่งมีขนาดเล็กลง หรือก็คือสามารถลดปริมาณข้อมูลที่ต้องส่งได้ อีกทั้งยังไม่ต้องเข้าไปแก้ไขพฤติกรรมการทำงานของระบบมากนัก ทำให้วิธีนี้เป็นวิธีที่น่าสนใจที่จะนำมาพัฒนาต่อยอด

การบีบอัดข้อมูลบนระบบเครือข่ายเซนเซอร์ไร้สายนั้น ไม่เหมือนกับการบีบอัดข้อมูลโดยทั่วไป เนื่องจากการบีบอัดข้อมูลโดยทั่วไป ระบบจะสามารถเห็นข้อมูลทั้งหมดที่ต้องทำการบีบอัด ทำให้สามารถใช้ประโยชน์จากค่าทางสถิติของข้อมูลได้อย่างเต็มที่ ตัวอย่างอัลกอริทึมบีบอัดที่ใช้ค่าทางสถิติของข้อมูลได้แก่ Huffman algorithm เป็นต้น ในขณะที่การบีบอัดข้อมูลบนระบบเครือข่ายเซนเซอร์ไร้สายข้อมูลจะเข้ามาเป็นลำดับ (Sequential Data) และเป็นค่าจริงในขณะนั้น ทำให้ข้อมูลที่นำมาประมวลผลได้ มีเพียงข้อมูลทางสถิติของข้อมูลในอดีตเท่านั้น ซึ่งจะทำให้ได้ประสิทธิภาพน้อยกว่า แต่อย่างไรก็ตามด้วยสภาพของข้อมูลที่ส่งอยู่บนระบบเครือข่ายเซนเซอร์ไร้สายที่ส่วนใหญ่แล้วเป็นข้อมูลที่ไม่ค่อยมีการเปลี่ยนแปลง เช่น ข้อมูลอุณหภูมิ ข้อมูลระดับน้ำ ข้อมูลความชื้น หรือข้อมูลภาพที่ไม่ค่อยมีการเปลี่ยนแปลง ข้อมูลดังกล่าวเป็นข้อมูลที่สามารถทำการบีบอัดได้ง่าย จึงทำให้เทคนิคการบีบอัดข้อมูลยังมีประโยชน์สำหรับใช้ในระบบเครือข่ายเซนเซอร์ไร้สาย

งานวิจัยบางชิ้นจะนำเสนอถึงการบีบอัดข้อมูลโดยการดัดแปลงอัลกอริทึม (Algorithm) ที่มีอยู่แล้วให้สามารถทำงานได้ดีขึ้นบนเครือข่ายเซนเซอร์ไร้สาย เช่นการดัดแปลงการทำงานของ Huffman Algorithm, Adaptive Huffman Algorithm [2][3][4][5][6] และ LZ (Lempel-Ziv) [7][8][9][10] ซึ่งก็สามารถบีบอัดข้อมูลได้มีประสิทธิภาพในสภาพแวดล้อมนั้น ๆ ทั้งนี้ผลลัพธ์ที่ได้ขึ้นอยู่กับลักษณะของข้อมูลตัวอย่างที่ใช้ในการทดสอบด้วย

การนำเครือข่ายเซนเซอร์ไร้สายมาใช้งานในบางกรณี เช่น ระบบเฝ้าติดตามผู้ป่วย (Health CareMonitoring System) นั้นมีความต้องการของระบบที่แตกต่างจากการใช้งานโดยทั่วไป เนื่องจากมีความต้องการของระบบที่เพิ่มขึ้นมา คือ ความน่าเชื่อถือ (Trustworthiness) ในส่วนของความเป็นเรียลไทม์ (Real time) ของระบบในการวัดค่าออกซิเจน (oxygen) ในเลือด ที่จะต้องตรวจสอบอย่างน้อยทุก 30 วินาที ความลับของข้อมูล (Privacy and Security) ซึ่งในบางกรณีที่ข้อมูลของผู้ป่วยจะต้องเป็นความลับ ความจำกัดของทรัพยากรระบบ (Resource Scarcity) เนื่องจากอุปกรณ์ในระบบเฝ้าติดตามผู้ป่วย (Health Care Monitoring System) มีขนาดค่อนข้างเล็ก และถูกออกแบบให้พกพาได้สะดวก จึงมีข้อจำกัดด้านพลังงาน ทำให้การเลือกใช้อุปกรณ์ และการออกแบบแอพพลิเคชัน (Application) จะต้องสอดคล้องกับแหล่งพลังงานที่มีจำกัดด้วย[12]

ทั้งนี้การใช้การบีบอัดข้อมูลบนระบบเครือข่ายเซนเซอร์ไร้สายจะช่วยลดพลังงานในการรับส่งข้อมูล แต่จะต้องสูญเสียพลังงานในส่วนของการประมวลผลที่เพิ่มมากขึ้น แปรผันตามความซับซ้อนในการทำงานของอัลกอริทึมบีบอัดข้อมูลที่ได้ออกแบบไว้ อัลกอริทึมที่จะทำการออกแบบขึ้นมา จะต้องมีประสิทธิภาพในการบีบอัดข้อมูลสูง ประหยัดพลังงาน และใช้ทรัพยากรฮาร์ดแวร์ให้น้อยที่สุด ดังนั้นการพัฒนาและวิจัยในส่วนนี้ จึงเป็นส่วนสำคัญที่จะทำให้ช่วยยืดอายุการทำงานของเครือข่ายเซนเซอร์ไร้สายให้ทำงานได้ยาวนานขึ้น

1.2 วัตถุประสงค์

1. เพื่อพัฒนาวิธีการบีบอัดข้อมูลที่เหมาะสมกับพฤติกรรมการทำงาน และลักษณะของข้อมูลบนระบบเครือข่ายเซนเซอร์ไร้สาย
2. เพื่อพัฒนาวงจรของอัลกอริทึมในการบีบอัดข้อมูลที่นำเสนอ

1.3 ประโยชน์ที่คาดว่าจะได้รับ

1. อัลกอริทึม และวงจรที่นำเสนอจะช่วยลดอัตราการใช้พลังงาน ส่งผลให้เซนเซอร์โหนดมีอายุการใช้งานที่ยาวนานยิ่งขึ้น
2. อัลกอริทึมที่นำเสนอ จะช่วยลดขนาดของข้อมูล ทำให้ระบบมีอัตราการรับส่งข้อมูลมากยิ่งขึ้น

บทที่ 2
ทฤษฎีและหลักการ

ในบทนี้ จะกล่าวถึงวิธีการจำแนกประเภทของอัลกอริทึมบีบอัดข้อมูล และวิธีการทำงานของอัลกอริทึมมาตรฐาน รวมไปถึงอัลกอริทึมอื่นที่เกี่ยวข้อง รวมทั้งชนิดของฮาร์ดแวร์ที่เหมาะสมกับการนำมาใช้งานร่วมกับงานวิจัยชิ้นนี้

2.1 การบีบอัดข้อมูล (Data Compression)

การบีบอัดข้อมูล คือ กระบวนการ หรือวิธีการในการเข้ารหัสข้อมูล โดยมีวัตถุประสงค์ให้ขนาดของข้อมูลลดลง เพื่อทำการจัดเก็บ หรือถ่ายโอน โดยที่ข้อมูลดังกล่าวจะต้องสามารถทำกระบวนการย้อนกลับ เพื่อให้ได้ข้อมูลต้นฉบับกลับมา โดยที่ข้อมูลต้นฉบับที่ได้จากการถอดรหัส อาจจะเหมือน หรือแตกต่างจากข้อมูลเดิมเล็กน้อย ขึ้นอยู่กับกระบวนการ หรือวิธีการที่ใช้ในการบีบอัดนั้น ๆ การบีบอัดข้อมูล สามารถจำแนกออกเป็น 2 กลุ่มใหญ่ ดังนี้

2.1.1 Lossless

เป็นการบีบอัดข้อมูลที่ไม่ยอมให้ข้อมูลเสียหาย คือ วิธีการที่เมื่อข้อมูลใด ๆ ถูกบีบอัด หรือเข้ารหัสแล้ว จะสามารถขยาย หรือถอดรหัส แล้วข้อมูลผลลัพธ์ จะเหมือนกับข้อมูลต้นฉบับทุกประการ วิธีการบีบอัดข้อมูลประเภทนี้ จะมีอัตราบีบอัดที่ไม่สูงมากนัก (ทั้งนี้ขึ้นอยู่กับประเภทของข้อมูลที่นำมาทำการบีบอัด) เหมาะกับการบีบอัดที่ต้องการรักษาความถูกต้องของข้อมูลเอาไว้ ตัวอย่างวิธีการบีบอัดแบบ Lossless เช่น Huffman Algorithm, Run length และ LZ(Lempel-Ziv) ใช้สำหรับบีบอัดข้อมูลประเภทเอกสาร หรือข้อมูลที่ต้องการความละเอียดในการนำมาใช้งาน

2.1.2 Lossy

เป็นการบีบอัดข้อมูลที่ยอมให้ข้อมูลเสียหายได้ คือ วิธีการที่เมื่อข้อมูลใด ๆ ถูกบีบอัด หรือเข้ารหัสแล้ว ข้อมูลผลลัพธ์จากการขยาย หรือถอดรหัส จะมีความแตกต่างจากข้อมูลต้นฉบับเล็กน้อย แต่ยังมีใจความสำคัญของข้อมูลเหมือนกันกับข้อมูลต้นฉบับ วิธีการบีบอัดข้อมูลประเภทนี้ จะมีอัตราการบีบอัดข้อมูลค่อนข้างสูง เหมาะสำหรับการบีบอัดข้อมูลที่ยินยอมให้สูญเสียรายละเอียดบางส่วนได้ ตัวอย่างวิธีการบีบอัดแบบ Lossy เช่น MPEG, JPEG และ MP3 เป็นต้น ส่วนใหญ่จะใช้การบีบอัดประเภทนี้กับข้อมูลจำพวก รูปภาพ วิดีโอ และเสียง เป็นต้น ซึ่งข้อมูลดังกล่าว สามารถสูญเสียรายละเอียดได้เล็กน้อย

2.2 Code Based Schemes

การบีบอัดข้อมูลนั้น ส่วนใหญ่จะใช้วิธีการแบ่งข้อมูลออกเป็นส่วนย่อย แล้วแทนที่ข้อมูลส่วนนั้น ด้วยสัญลักษณ์ (Symbol) หรือ รหัส (Code) เมื่อต้องการทำการขยายข้อมูล หรือถอดรหัส ก็จะทำการแทนที่สัญลักษณ์ หรือรหัส ด้วยส่วนของข้อมูล ซึ่งในการบีบอัดข้อมูล

ส่วนใหญ่นั้น จะต้องทำการสร้างส่วนของข้อมูลที่เอาไว้บ่งบอกว่า ข้อมูลตัวใด เชื่อมโยงกับสัญลักษณ์ หรือรหัสตัวใดเอาไว้ก่อน เพื่อใช้ในการขยายข้อมูลในภายหลัง การแทนที่นั้น สามารถแบ่งออกเป็นประเภทย่อย ตามลักษณะของข้อมูล และรหัส ดังนี้

- Fixed-to-Fixed เป็นการแทนที่ข้อมูลที่มีขนาดที่แน่นอน ด้วยรหัสที่มีขนาดแน่นอน
- Variable-to-Fixed เป็นการแทนที่ข้อมูลที่มีขนาดไม่แน่นอน ด้วยรหัสที่มีขนาดแน่นอน
- Fixed-to-Variable เป็นการแทนที่ข้อมูลที่มีขนาดแน่นอน ด้วยรหัสที่มีขนาดไม่แน่นอน
- Variable-to-Variable เป็นการแทนที่ข้อมูลที่มีขนาดไม่แน่นอน ด้วยรหัสที่มีขนาดไม่แน่นอน

ข้อมูลที่มีขนาดแน่นอน ในที่นี้หมายถึง ส่วนของข้อมูลที่จะถูกแทนที่ ในการบีบอัดนั้น จะมีขนาดความยาวคงที่ เท่ากันหมดทั้งการบีบอัด

รหัสที่มีขนาดแน่นอน ในที่นี้หมายถึง รหัสที่จะนำมาแทนที่ข้อมูลในการบีบอัดนั้น จะมีขนาดความยาวคงที่ เท่ากันหมดทั้งการบีบอัด

ข้อมูลที่มีขนาดไม่แน่นอน ในที่นี้หมายถึง ส่วนของข้อมูลที่จะถูกแทนที่ ในการบีบอัดนั้น จะมีขนาดความยาวที่แตกต่างกันไป ขึ้นอยู่กับวิธีการบีบอัดนั้น ว่ามีการแบ่งวรรคของข้อมูลอย่างไร

รหัสที่มีขนาดไม่แน่นอน ในที่นี้หมายถึง รหัสที่จะนำมาแทนที่ข้อมูลในการบีบอัด จะมีขนาดความยาวที่แตกต่างกันไป ขึ้นอยู่กับวิธีการบีบอัดนั้น ว่ามีการจัดสรรรหัสอย่างไร

การใช้ Code Based Schemes ในการจำแนกประเภทของการบีบอัดข้อมูล จะช่วยให้เข้าใจวิธีการบีบอัดข้อมูลได้ง่ายยิ่งขึ้น เนื่องจากทราบแน่ชัดว่า การตัดแบ่งข้อมูลต้นฉบับเป็นลักษณะใด และการแทนด้วยสัญลักษณ์ หรือรหัส จะเป็นไปในลักษณะใด

2.3 Run Length Encoding

วิธีการ Run Length Encoding นั้นเป็นวิธีการแทนที่ข้อมูลที่ซ้ำกัน และอยู่ติดกันด้วยรหัสที่มีความสามารถในการระบุความยาวของการซ้ำกันของข้อมูล โดยที่จะทำการสร้างตารางเพื่อจำลองการสร้างรหัสที่ใช้แทนข้อมูล

ตารางที่ 2-1 Run Length Code ชนิด 3 บิต

Data	Code
1	000
01	001
001	010
0001	011
00001	100
000001	101
0000001	110
0000000	111

จากตารางที่ 2-1 ถ้าหากจะใช้เพื่อเข้ารหัสข้อมูล 0001 0110 0000 0000 0001 0100 0000 0001 อาจเข้ารหัสได้เป็น 011 001 000 111 101 001 111 010

เช่นเดียวกันกับการเข้ารหัสข้อมูล การถอดรหัสก็จำเป็นที่จะต้องใช้ตาราง ในการเชื่อมโยงระหว่างรหัสกับข้อมูลไว้ด้วยกัน ตัวอย่างเช่น ถ้ากำหนดให้ข้อมูลที่ต้องการขยายหรือถอดรหัส คือ 011 001 000 111 101 001 111 010 เมื่อใช้ตารางที่ 2-1 ในการถอดรหัสแล้ว จะสามารถถอดรหัสข้อมูลออกมาได้เป็น 0001 010 1 0000000 000001 01 0000000 001 (แบ่งการเว้นวรรค ตามรหัส) ซึ่งจะมีค่าเหมือนกับข้อมูลต้นฉบับ

วิธีนี้ เมื่อจำแนกด้วย Code Based Schemes จะถูกจัดอยู่ใน Variable-to-Fixed เนื่องจากข้อมูลที่ถูกแทนที่ จะมีความยาวที่แตกต่างกันไป ในขณะที่ รหัสที่ใช้แทนที่ จะมีความยาวคงที่ ที่ 3 บิต ทั้งนี้การบีบอัดข้อมูลด้วยวิธีนี้ จะใช้ได้อย่างมีประสิทธิภาพก็ต่อเมื่อข้อมูลที่ต้องการนำมาบีบอัดนั้น มีข้อมูลซ้ำกันอยู่ติดกันในปริมาณมาก

2.4 Dictionary

Dictionary เป็นวิธีการบีบอัดแบบ Lossless และหากจำแนกด้วย Code Based Schemes จะจัดอยู่ในประเภท Fixed-to-Fixed คือ เปลี่ยนจากข้อมูลที่มีความยาวคงที่ ไปเป็นรหัสที่มีความยาวคงที่ โดยอาศัยปริมาณรูปแบบที่เป็นไปได้ของข้อมูลที่มีอยู่น้อย เมื่อเทียบกับขนาดของข้อมูลจริง ตัวอย่างเช่น ถ้ากำหนดให้ข้อมูลต้นฉบับ เป็นข้อมูลบทความภาษาอังกฤษ ซึ่งภายในบทความ จะประกอบด้วยตัวอักษรภาษาอังกฤษ ตัวพิมพ์เล็ก 26 ตัวอักษร ตัวอักษรภาษาอังกฤษตัวพิมพ์ใหญ่ 26 ตัวอักษร ‘.’ และ ‘?’ ทั้งหมดรวมเป็น 54 ตัวอักษรซึ่งข้อมูลประเภท Character นั้น โดยทั่วไปแล้ว จะมีขนาด 8 บิต แต่เมื่อวิเคราะห์จากข้อมูลต้นฉบับ ซึ่งมีตัวอักษรทั้งหมด 54 แบบ จะเห็นได้ว่า สามารถใช้รหัสเพียง 6 บิต เพื่ออ้างถึงตัวอักษรทุกตัวในบทความได้ แสดงให้เห็นว่าวิธีการนี้ สามารถสร้างอัตราการบีบอัดได้ถึง 25%

อัลกอริทึมที่นำหลักการของ Dictionary มาใช้ ได้แก่ LZ และ GZIP (GZIP ใช้ LZ กับ Huffman algorithm มาทำงานร่วมกัน) เป็นต้น

แต่ในบางกรณีนั้น Dictionary จะถูกจำแนกตาม Code Based Schemes ได้เป็น Variable-to-Fixed เนื่องจากการบีบอัดข้อมูลจะตัดแบ่งข้อมูลออกเป็นคำ เช่น I am a Doctor จะตัดแบ่งข้อมูลออกเป็น 4 คำ คือ I, am, a และ Doctor ซึ่งแต่ละคำก็จะได้รหัสที่แตกต่างกันออกไป

2.5 Golomb Code

Golomb Code เป็นวิธีที่มีการนำ Run Length Encoding มาพัฒนา โดยที่ยังคงเป็นการบีบอัดแบบ Lossless อยู่ แต่หากจำแนกด้วย Code Based Schemes แล้ว จะอยู่ในประเภท Variable-to-Variable คือ ใช้วิธีในการตัดแบ่งข้อมูลเหมือนกับ Run Length Encoding แต่ใช้วิธีในการกำหนดค่ารหัสที่แตกต่างกัน

การกำหนดค่ารหัสของ Golomb Code นั้น จะแบ่งรหัส ออกเป็น 2 ส่วน คือ Prefix กับ Tail โดยมีข้อมูลการเชื่อมโยงระหว่าง ข้อมูล กับ รหัส ดังตารางที่ 2-2

ตารางที่ 2-2 ความเชื่อมโยงระหว่าง ข้อมูล กับรหัสของ Golomb Code

Group	Run-Length	Group Prefix	Tail	Code
A1	0	0	00	000
	1		01	001
	2		10	010
	3		11	011
A2	4	10	00	1000
	5		01	1001
	6		10	1010
	7		11	1011
A3	8	110	00	11000
	9		01	11001
	10		10	11010
	11		11	11011
…	…	…	…	…

รหัสผลลัพธ์จะประกอบด้วย Prefix ตามด้วย Tail โดยที่ Prefix จะใช้เพื่อแบ่งข้อมูลออกเป็นกลุ่ม ๆ เพื่อให้เกิดความแตกต่างขึ้นระหว่างกลุ่มข้อมูล จากตารางที่ 2-2 จะเห็นได้ว่าข้อมูลที่มีความยาวเป็น 0, 1, 2 และ 3 จะมีค่า Prefix เป็น 0 ในขณะที่ข้อมูลที่มีความยาว 4, 5, 6 และ 7 จะมีค่า Prefix เป็น 10 กล่าวคือ ข้อมูลที่มีความยาวน้อย ก็จะได้ Prefix ที่มีความยาวน้อย ส่วนข้อมูลที่มีความยาวมาก ก็จะได้ Prefix ที่มีความยาวมากขึ้น เพื่อให้เหมาะสมกับสถานการณ์ในการบีบอัดข้อมูล ในส่วนของ Tail นั้น จะใช้เพื่อระบุข้อมูลอย่างเจาะจงอีกครั้งหนึ่ง หลังจากการแบ่งแยกโดย Prefix ซึ่ง Tail นั้น จะมีความยาวคงที่

2.6 Huffman algorithm

Huffman algorithm เป็นวิธีที่นิยมในการนำมาศึกษาเพื่อพัฒนาต่อยอด เนื่องจาก Huffman algorithm เป็นวิธีที่ไม่ซับซ้อนมากนัก ทำความเข้าใจได้ง่าย และมีประสิทธิภาพสูง Huffman algorithm จัดเป็นการบีบอัดข้อมูลแบบ Lossless และหากแบ่งตาม Code Based Schemes จะจัดอยู่ในประเภท Fixed-to-Variable

Huffman algorithm นั้นจะใช้ข้อมูลทางสถิติเข้ามาช่วยในการจัดสรรรหัส ให้กับข้อมูลแต่ละตัว โดยมีแนวคิดพื้นฐานว่า ข้อมูลที่มีความถี่สูงที่สุด (ปรากฏบ่อยครั้งที่สุด หรือมีอยู่มากที่สุด) จะถูกแทนที่ด้วยรหัสที่มีความยาวสั้นที่สุด และข้อมูลที่มีความถี่รองลงมา ก็จะถูกแทนที่ด้วยรหัสที่มีความยาวมากขึ้น โดยวิธีการในการจัดสรรรหัสนั้น จะใช้วิธีการสร้าง Binary Tree เพื่อเชื่อมโยงรหัสกับข้อมูล รหัสที่ถูกสร้างขึ้นด้วยวิธีนี้ จะมีความยาวที่เหมาะสมกับความถี่ของข้อมูล ทำให้การบีบอัดข้อมูล มีอัตราการบีบอัดที่สูงตามไปด้วย

ก่อนที่ Huffman algorithm จะทำการบีบอัดข้อมูล ขั้นตอนแรกที่จะต้องทำ คือ การสร้างตาราง เพื่อระบุว่า ข้อมูลตัวใด มีค่าความถี่เท่าใด ด้วยการนับความถี่ของข้อมูลแต่ละตัวจากข้อมูลต้นฉบับ หลังจากที่ทราบความถี่ของข้อมูลทั้งหมดแล้ว ก็จะทำการเรียงข้อมูล จากข้อมูลที่มีความถี่มากไปยังข้อมูลที่มีความถี่น้อย ดังตารางที่ 2-3

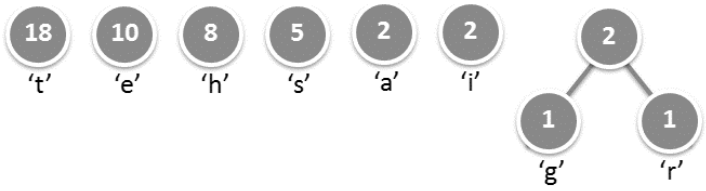
ตารางที่ 2-3 ค่าความถี่ของข้อมูลต้นฉบับ

Frequency	Data
18	‘t’
10	‘e’
8	‘h’
5	‘s’
2	‘a’
2	‘i’
1	‘g’
1	‘r’

จากตารางที่ 2 3 ข้อมูลที่มีความถี่สูงสุดจะอยู่บนสุด และข้อมูลที่มีความถี่น้อยสุดจะอยู่ล่างสุด เมื่อได้ตารางดังตารางที่ 2 3 แล้ว ก็จะเข้าสู่กระบวนการสร้าง Binary Tree ซึ่งการสร้าง Binary Tree นั้น จะอาศัยการเปรียบเทียบค่าความถี่ของข้อมูล โดยการนำข้อมูลมาเรียงจากข้อมูลที่มีค่าความถี่มากไปยังข้อมูลที่มีค่าความถี่น้อย ดังภาพประกอบที่ 2-1

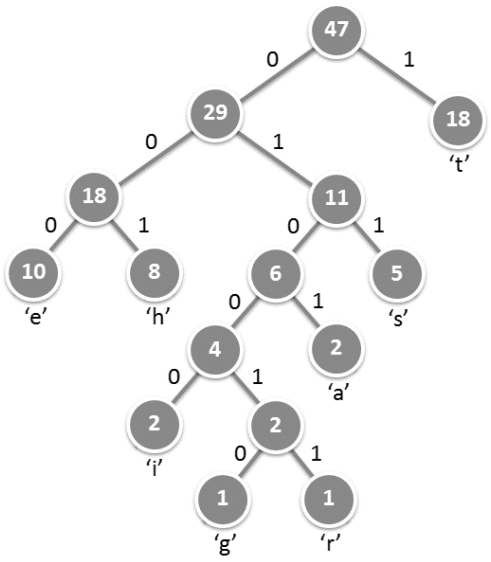
ภาพประกอบที่ 2-1 การเรียงข้อมูลตามค่าความถี่จากมากไปหาน้อย

หลังจากเรียงค่าความถี่แล้ว ให้ทำการสร้าง Binary Tree จากข้อมูลที่มีค่าความถี่น้อยที่สุด 2 อันดับ แล้วรวมค่าความถี่ไว้ด้วยกัน จากนั้น ให้ทำการเรียงข้อมูลตามค่าความถี่อีกครั้ง ซึ่งจะได้ผลลัพธ์ดังภาพประกอบที่ 2-2



ภาพประกอบที่ 2-2 การเรียงข้อมูลตามค่าความถี่จากมากไปหาน้อยในรอบที่ 2

หลังจากนั้น ทำการ สร้าง Binary Tree จากข้อมูลที่มีค่าความถี่น้อยที่สุด 2 อันดับ และรวมค่าความถี่ไว้ด้วยกันแล้ว ทำการเรียงข้อมูลตามค่าความถี่อีกครั้ง ทำเช่นนี้ไปเรื่อย ๆ จนกระทั่งข้อมูลทั้งหมดถูกรวมเอาไว้ใน Tree เดียวกัน หลังจากนั้น จึงทำการกำหนดรหัสให้กับแขนของ Binary Tree โดยที่แขนด้านซ้ายของ Binary Tree มีค่าเป็น 0 และแขนด้านขวาของ Binary Tree มีค่าเป็น 1 ซึ่งจะทำให้ Tree มีลักษณะดังภาพประกอบที่ 2-3



ภาพประกอบที่ 2-3 Binary Tree ที่เสร็จสมบูรณ์

หลังจากนั้น ให้ทำการสร้างรหัสให้กับข้อมูลแต่ละตัว โดยใช้รหัสที่อยู่ที่แขนของ Binary Tree แต่ละตัว ที่เชื่อมไปยังข้อมูลตัวนั้น ๆ มาต่อกัน ตัวอย่างเช่น ‘e’ จะได้รหัสเป็น 000, ‘h’ จะได้รหัสเป็น 001 และ ‘r’ จะได้รหัสเป็น 010011 เป็นต้น ซึ่งความสัมพันธ์ของ ค่าความถี่ ข้อมูล และรหัส จะเป็นไปตาม ตารางที่ 2-4

ตารางที่ 2-4 ความสัมพันธ์ระหว่าง ค่าความถี่ ข้อมูล และรหัส

Frequency	Data	Code
18	‘t’	1
10	‘e’	000
8	‘h’	001
5	‘s’	011
2	‘a’	0101
2	‘i’	01000
1	‘g’	010010
1	‘r’	010011

จากตารางที่ 2-4 สามารถคำนวณหาจำนวนของข้อมูลหลังจากทำการบีบอัดได้จาก

จำนวนข้อมูล(บิต) = ผลรวมของ (ค่าความถี่×ความยาวของรหัส(บิต)) (2.1)

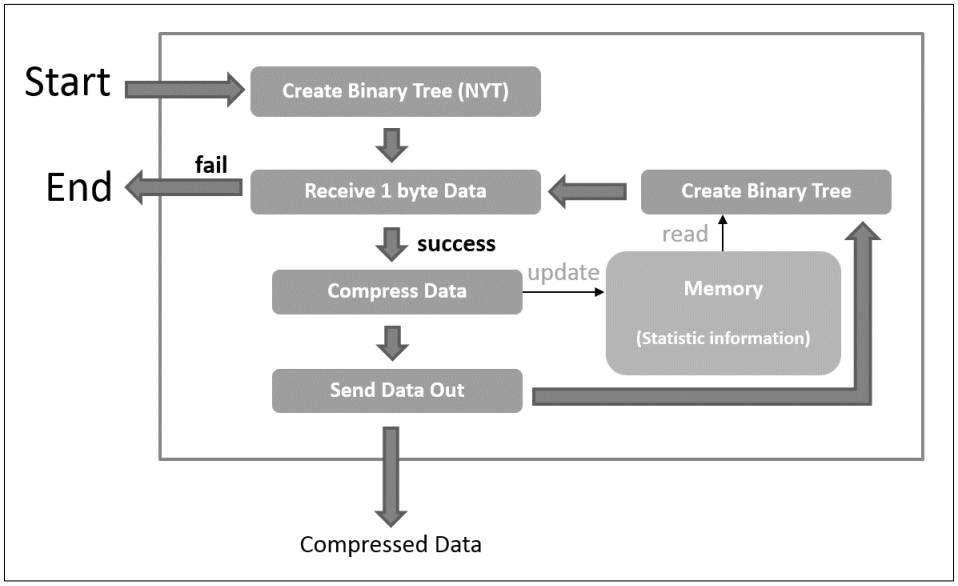
ซึ่งจะมีค่าเท่ากับ 18(1) + 10(3) + 8(3) + 5(3) + 2(4) + 2(5) + 1(6) + 1(6) = 117 บิต ในขณะที่ข้อมูลที่ไม่ได้มีการบีบอัด จะมีจำนวนเท่ากับ ผลรวมของค่าความถี่ คูณด้วย ขนาดของข้อมูลแบบ Character ซึ่งเท่ากับ (18 + 10 + 8 + 5 + 2 + 2 +1 + 1)(8) = 376 ซึ่งมีขนาดใหญ่กว่าผลลัพธ์จากการบีบอัดค่อนข้างมาก

ในการถอดรหัสข้อมูล ก็จำเป็นจะต้องใช้ตารางที่ 2-4 ในการถอดรหัสข้อมูลด้วย โดยการนำข้อมูลกลับมาแทนที่รหัส ตัวอย่างเช่น ถ้ารหัสเป็น 10100001001000010011 สามารถถอดรหัสออกได้เป็น tiger คือ 1 แทนที่ด้วย t, 01000 แทนที่ด้วย i, 010010 แทนที่ด้วย g, 000 แทนที่ด้วย e และ 010011 แทนที่ด้วย r

การบีบอัดข้อมูลด้วย Huffman algorithm นั้น จำเป็นจะต้องส่งตารางที่ใช้ในการเข้ารหัสข้อมูลไปกับข้อมูลที่ได้ทำการบีบอัดแล้วด้วย เพื่อให้สามารถถอดรหัสกลับมาเป็นข้อมูลต้นฉบับได้ ซึ่งข้อมูลส่วนที่เพิ่มขึ้นมา ถือว่าเป็น Overhead ที่เกิดจากการบีบอัดข้อมูล

2.7 Adaptive Huffman algorithm

Adaptive Huffman algorithm เป็นวิธีการที่พัฒนาต่อมาจาก Huffman algorithm ซึ่งจะลดข้อด้อยของ Huffman algorithm ในส่วนของ Overhead ที่เกิดจากการแนบตาราง หรือ Binary Tree ที่ใช้เชื่อมโยงรหัสกับข้อมูล ไปกับข้อมูลที่ถูกบีบอัดแล้วด้วย โดยที่ Adaptive Huffman algorithm นั้น จะทำงานซับซ้อนกว่า Huffman algorithm คือ Huffman algorithm จะอ่านข้อมูลทั้งหมดแล้วทำการสร้าง Binary Tree ขึ้นมาเพียงครั้งเดียว ในขณะที่ Adaptive Huffman algorithm จะทำการอ่านข้อมูลเพียง 1 ตัว แล้วทำการสร้าง Binary Tree ขึ้นมา จากนั้นก็จะใช้ Binary Tree ดังกล่าว เป็นตารางในการเข้ารหัสข้อมูลตัวที่ 2 หลังจากนั้นจะทำการสร้าง Binary Tree ขึ้นมาใหม่อีกครั้ง แล้วอ่านข้อมูลตัวถัดไป ทำเช่นนี้ไปเรื่อย ๆ จนกระทั่ง ไม่เหลือข้อมูลให้ทำการบีบอัด โดยขั้นตอนดังกล่าว สามารถสรุปเป็นแผนภาพได้ดังภาพประกอบที่ 2-4



ภาพประกอบที่ 2-4 ขั้นตอนการทำงานของ Adaptive Huffman algorithm

จากภาพประกอบที่ 2-4 การทำงานของ Adaptive Huffman algorithm จะทำงานเป็นรอบ โดยการทำงานของ Adaptive Huffman algorithm มีขั้นตอนดังนี้

- Create Binary Tree (NYT) เมื่อเริ่มทำการบีบอัดข้อมูล ขั้นตอนนี้จะเป็นขั้นตอนแรก โดยที่ขั้นตอนนี้ จะทำการสร้าง Binary Tree ที่มีสมาชิกเพียงตัวเดียว คือ NYT
  - สำหรับผู้ทำการเข้ารหัสข้อมูล NYT จะถูกใช้ในกรณีที่พบข้อมูลที่ไม่เคยเข้ามาในระบบมาก่อน ซึ่งในรอบนั้น จะทำการส่งรหัสที่เชื่อมโยงกับ NYT ออกไป ตามด้วยข้อมูลตัวใหม่ หรืออีกนัยหนึ่ง NYT เป็นรหัสที่ใช้แสดงว่าข้อมูลที่ตามหลังมาไม่ใช่รหัส แต่เป็นข้อมูลต้นฉบับที่ยังไม่เคยเข้ามาในระบบนั้นเอง
  - สำหรับผู้ที่ทำการถอดรหัสข้อมูล รหัสที่เชื่อมโยงกับ NYT จะเป็นรหัสที่เอาไว้บอกว่า รหัสหลังจากนี้ไป ไม่ใช่ข้อมูลที่ถูกเข้ารหัส แต่เป็นข้อมูลต้นฉบับ
- Receive 1 byte Data หลังจากที่ได้ทำการสร้าง Binary Tree แล้ว ขั้นตอนนี้จะเป็นเพียงขั้นตอนในการอ่านข้อมูลตัวใหม่เข้ามาในระบบ ถ้ายังมีข้อมูลเหลืออยู่ ก็จะเอาข้อมูลนั้นไปเข้ารหัสในขั้นตอนต่อไป ถ้าไม่เหลือข้อมูลแล้วก็จะจบการทำงาน
- Compress Data เป็นขั้นตอนสำหรับค้นหารหัสของข้อมูลที่ได้มาจากขั้นตอน “Receive 1 byte Data” ถ้าหากมีข้อมูลดังกล่าวอยู่ใน Binary Tree ก็จะทำการแทนที่ข้อมูลด้วยรหัสที่หาพบ แต่ถ้าหากหาไม่พบ ก็ให้ทำการแทนที่ข้อมูลด้วยรหัสของ NYT ตามด้วยข้อมูลตัวนั้น หลังจากนั้นให้ทำการเพิ่มค่าความถี่ให้กับข้อมูลตัวดังกล่าวขึ้น 1 (ค่าข้อมูลความถี่ถูกเก็บอยู่ในหน่วยความจำ)
- Send Data Out เป็นขั้นตอนของการส่งมอบข้อมูลที่ถูกเข้ารหัสแล้ว ในส่วนของวิธีการนั้น จะขึ้นอยู่กับการใช้งานระบบ ว่าถูกใช้งานในรูปแบบใด
- Create Binary Tree เป็นการสร้าง Binary Tree โดยขั้นตอนจะเหมือนกับการสร้าง Binary Tree ของ Huffman algorithm โดยใช้ข้อมูลที่อยู่ภายในหน่วยความจำ และจะมี NYT เพิ่มเข้ามาอีก 1 ตัว (NYT มีค่าความถี่เป็น 0 เสมอ)

หลังจากทำขั้นตอน Create Binary Tree เสร็จ ระบบก็จะวนกลับไปทำขั้นตอน Receive 1 byte Data อีก ทำเช่นนี้จนกว่าจะไม่สามารถอ่านข้อมูลออกมาได้ (ได้ fail ที่ขั้นตอน Receive 1 byte Data)

เมื่อ Huffman algorithm ถูกพัฒนาให้เป็น Adaptive Huffman algorithm ก็จะทำให้ระบบนี้ สามารถที่จะนำไปประยุกต์ใช้กับเครือข่ายเซนเซอร์ไร้สาย ได้ดียิ่งขึ้น เนื่องจากในเครือข่ายเซนเซอร์ไร้สาย ข้อมูลที่เข้ามาในระบบจะเป็นแบบอนุกรม (Serial) ซึ่งในขณะใดขณะหนึ่งนั้น จะมีข้อมูลอยู่ในระบบไม่มากนัก ถ้าหากต้องการใช้ Huffman algorithm ในการเข้ารหัส ก็จำเป็นต้องหน่วงข้อมูลให้อยู่ในระบบมากพอที่จะทำให้อัตราการบีบอัดข้อมูล มีอัตราส่วนมากกว่าค่า

Overhead ที่เกิดจาก Binary Tree ซึ่งการหน่วงข้อมูลไว้ในระบบเป็นจำนวนมาก ก็จะเป็นผลเสียกับระบบที่ต้องการความเป็นเรียลไทม์ (Real time) ในขณะที่ Adaptive Huffman สามารถตอบสนองต่อความต้องการในจุดนี้ได้ดีกว่า

2.8 Modified Adaptive Huffman algorithm (MAH)

MAH [3] เป็นหนึ่งในงานวิจัยที่ได้มีการพัฒนาต่อยอดมาจาก Adaptive Huffman algorithm โดยใช้หลักของการสร้าง Binary Tree ของเซ็ตที่เก็บค่าส่วนต่างของข้อมูล โดยที่รหัสจะถูกแบ่งออกเป็น 2 ส่วน คือ Prefix และ Suffix โดยที่ Prefix จะทำหน้าที่ในการระบุเซ็ตใน Binary Tree ส่วน Suffix จะใช้ในการอ้างอิงถึงตำแหน่งของข้อมูลที่อยู่ภายในเซ็ต

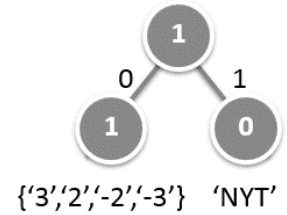
MAH จะมีวิธีการคล้ายกับ Adaptive Huffman algorithm ในส่วนของ Prefix โดยที่จุดเริ่มต้นนั้น Binary Tree จะมีเพียง NYT เป็นสมาชิกเพียงตัวเดียว ดังภาพประกอบที่ 2-5 ซึ่งทำให้ได้ตารางแสดงความเชื่อมโยง ดังตารางที่ 2-5

ภาพประกอบที่ 2-5 Binary Tree ของ MAH ณ เวลาเริ่มต้น

ตารางที่ 2-5 ความเชื่อมโยง ของ MAH algorithm ในรอบแรก

Frequency	Symbol	Code
0	NYT	0

จากนั้น เมื่อมีข้อมูล '2' เข้ามาในระบบ ก็จะต้องหาค่า '2' ใน Binary Tree แต่เนื่องจาก '2' เข้ามาในระบบเป็นครั้งแรก ดังนั้น ระบบจะทำการส่งรหัสของ 'NYT' ออกไปก่อนแล้วตามด้วย '0000 0010' หลังจากนั้น ก็จะต้องทำการเพิ่มค่าความถี่ให้กับ '2' แต่เนื่องจากการเพิ่มค่านั้นจะต้องทำเป็นเซ็ต ซึ่งจะทำให้ได้ Binary Tree ดังภาพประกอบที่ 2-6 และนำมาสร้างตารางแสดงความเชื่อมโยงได้ดังตารางที่ 2-6



ภาพประกอบที่ 2-6 Binary Tree ของ MAH ในรอบที่ 2

ตารางที่ 2-6 ความเชื่อมโยงของ MAH ในรอบที่ 2

Frequency	Symbol	Code
1	{-3,-2,2,3}	0
0	NYT	1

จากนั้น กำหนดให้มีข้อมูลตัวที่ 3 เข้ามาในระบบ มีค่าเท่ากับ 5 ซึ่งเท่ากับว่า ข้อมูลผลต่างมีค่าเท่ากับ ข้อมูลในอดีต – ข้อมูลปัจจุบัน ซึ่งเท่ากับ 2 – 5 = -3 เมื่อนำค่าไปหาดูใน Binary Tree ก็จะทำให้ได้รหัสเป็น 0 11 หมายความว่า Prefix มีค่าเท่ากับ 0 และ Suffix มีค่าเท่ากับ 11

2.9 Hardware Simulation

การบีบอัดข้อมูลในเครือข่ายเซนเซอร์ไร้สายสามารถทำได้ทั้งแบบการเขียนโปรแกรมบีบอัดลงบนไมโครคอนโทรลเลอร์ของโหนด หรือทำการสร้างวงจรสำหรับบีบอัดไว้โดยเฉพาะ เพื่อให้การบีบอัดทำงานได้อย่างรวดเร็วและประหยัดพลังงาน จึงจำเป็นต้องมีวงจรสำหรับบีบอัดเป็นวงจรพิเศษบนโหนด โดยในงานวิจัยชิ้นนี้ จะทำการจำลองการทำงานของการบีบอัดข้อมูลลงในฮาร์ดแวร์ด้วยเทคโนโลยีเอฟพีจีเอ (FPGA) เนื่องจากมีความยืดหยุ่น ราคาถูก และรวดเร็วในการพัฒนาวงจร เพื่อเป็นข้อมูลสำหรับการพัฒนาต่อในอนาคต ที่จะสามารถสร้างวงจรจากอัลกอริทึมบีบอัดที่ได้ มาเป็นส่วนหนึ่งของ System-on-Chip (SoC) สำหรับโมดูลสื่อสารไร้สาย

การจำลองการทำงานของอัลกอริทึมลงบนฮาร์ดแวร์ ก็เป็นส่วนที่มีความสำคัญ เพื่อช่วยในการรับรองว่า ระบบที่ได้ออกแบบขึ้นมานั้น มีความเป็นไปได้ที่จะนำไปดำเนินการใช้งานได้จริง และเพื่อให้ทราบถึง มูลค่า (Cost) และทรัพยากร (Resource) พื้นฐาน ที่จะต้องเสียไปกับระบบ ซึ่งเป็นส่วนสำคัญที่จะสามารถบอกได้ว่า วงจรสำหรับการบีบอัดที่นำเสนอนั้น มีการใช้ทรัพยากรและพลังงานอย่างไร รวมถึงคาดการณ์ประสิทธิภาพ (เวลา) ของการบีบอัดข้อมูลด้วย

2.9.1 Microcontroller

Microcontroller คือ อุปกรณ์ฮาร์ดแวร์ที่ทำหน้าที่ในการควบคุมการทำงานของระบบ มีความสามารถเช่นเดียวกับระบบคอมพิวเตอร์ โดยพื้นฐานแล้ว จะประกอบด้วย 3 ส่วน ได้แก่ ส่วนประมวลผล (CPU : Central Processing Unit) หน่วยความจำ (Memory) และช่องทางของสัญญาณ (Port) สามารถที่จะทำการเขียนโปรแกรมลงไป เพื่อควบคุมการทำงานของอุปกรณ์อิเล็กทรอนิกส์ได้ โดยมีลำดับการทำงานเป็นขั้นตอนนิยมนำมาใช้ในระบบฝังตัว (Embedded) เพื่อเป็นตัวควบคุมการทำงานของระบบ เช่น ใช้ในการสร้างระบบรดน้ำผัก โดยทำหน้าที่ในการนับเวลา เมื่อถึงเวลาที่กำหนดแล้ว จึงส่งสัญญาณออกไป เพื่อควบคุมวาล์วน้ำไฟฟ้าให้ทำการเปิดปิด

2.9.2 FPGA

FPGA หรือ Field Programmable Gate Array เป็นหน่วยของ Slice เล็ก ๆ ที่ถูกนำมาเรียง และเชื่อมต่อกันแบบเมทริกส์ (Matrix) โดยที่ภายใน Slice แต่ละตัวก็จะประกอบด้วย Logic gate และ Flip-Flops ซึ่งตัว FPGA สามารถที่จะทำการโปรแกรมลงไปเพื่อสร้างวงจรดิจิตอล

ขึ้นมา โดยที่วงจรดังกล่าว จะเกิดจากการเชื่อมต่อของ Slice ต่าง ๆ ที่อยู่ภายใน และ FPGA ยังมีความสามารถในการที่จะโปรแกรมลงไปใหม่ เช่นเดียวกับ Microcontroller รวมถึงการที่สามารถโปรแกรมวงจรของ Microcontroller ลงไปได้ด้วย หรือที่เรียกว่า Micro Blaze ดังนั้น FPGA จึงมีความสามารถที่หลากหลาย แต่ในขณะเดียวกัน การที่จะเขียนโปรแกรมลงไปบน FPGA ก็จำเป็นต้องใช้ภาษาเฉพาะทาง เช่น VHDL และ Verilog ซึ่งเป็นภาษาที่สื่อถึงลักษณะของวงจรที่ต้องการจะออกแบบ

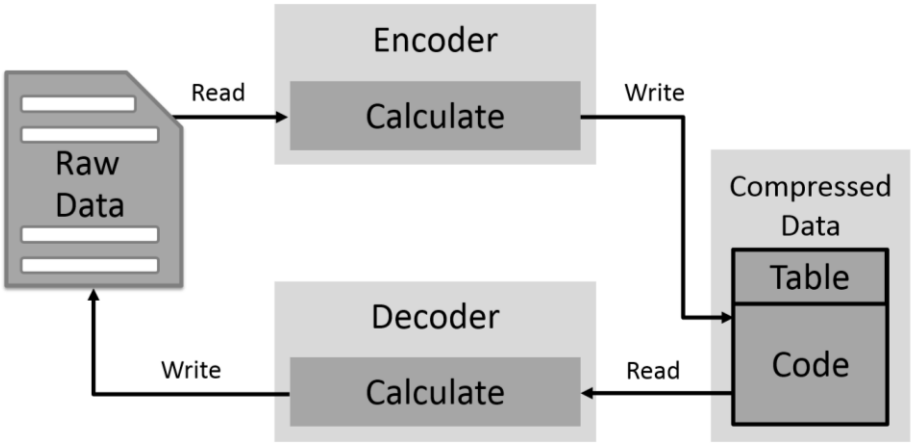
ในงานวิจัยชิ้นนี้ จะเลือกใช้ FPGA เนื่องจากความสามารถในการสังเคราะห์วงจรได้ค่อนข้างอิสระ และยังสื่อถึงลักษณะของวงจรจริง ทำให้ง่ายต่อการนำไปพัฒนาต่อยอด เพื่อผลิตเป็นวงจรรวม (IC : Integrated Circuits) ทำให้กระบวนการบีบอัดข้อมูลสามารถที่จะทำงานได้อย่างอิสระ โดยที่ผู้ใช้งานไม่จำเป็นต้องเข้าไปจัดการ

บทที่ 3
การออกแบบอัลกอริทึม

ในบทนี้ จะกล่าวถึงแนวคิดในการออกแบบวิธีการบีบอัดข้อมูล และหลักการทำงานของแต่ละอัลกอริทึมที่ได้ทำการออกแบบ ข้อจำกัดของอัลกอริทึมในการนำไปใช้งานจริง วิธีการแก้ไข และการสร้างอัลกอริทึมแบบ Hybrid

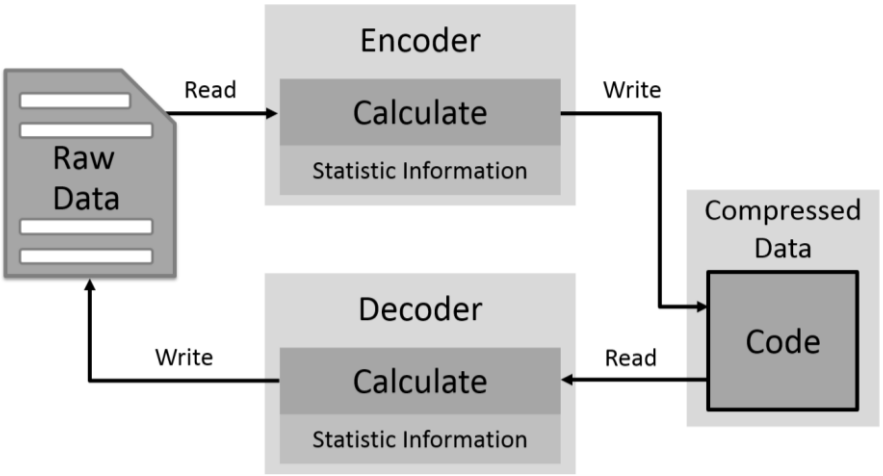
3.1 แนวคิดในการบีบอัดข้อมูล

การบีบอัดข้อมูล หรือการเข้ารหัสข้อมูลนั้น คือการแทนที่ข้อมูลต้นฉบับด้วยรหัส ซึ่งทั้งผู้ส่ง และผู้รับ จำเป็นจะต้องเข้าใจตรงกันถึงวิธีการในการแทนค่าด้วยรหัส ซึ่งในจุดนี้แต่ละอัลกอริทึมก็จะใช้วิธีการที่แตกต่างกันออกไป แต่ในการบีบอัดโดยทั่วไป จะนิยมทำการสร้างตารางข้อมูลขึ้นมา เพื่อใช้ในการเชื่อมโยงกันระหว่างรหัสกับข้อมูล ซึ่งตารางดังกล่าว อาจถูกแนบไปกับข้อมูลที่ถูกบีบอัด หรือถูกสร้างขึ้นภายหลังก็ได้ เช่น การบีบอัดด้วย Huffman Algorithm จะใช้การแนบตารางไปกับข้อมูล ในขณะที่ Adaptive Huffman Algorithm จะใช้วิธีการสร้างตารางจากข้อมูลในอดีตที่เคยอ่านมา คือมีลักษณะคล้ายกับการเก็บข้อมูลทางสถิติ เพื่อสร้างตารางที่ใช้ในการทำนายผลลัพธ์ ซึ่งทั้งสองวิธีนี้ก็จะให้อัตราการบีบอัดข้อมูลที่ไม่เท่ากัน ขึ้นอยู่กับลักษณะของข้อมูลด้วย



ภาพประกอบที่ 3-1 การบีบอัดข้อมูลโดยการแนบตารางไปกับข้อมูล

ภาพประกอบที่ 3-1 แสดงถึงวิธีการบีบอัดข้อมูลที่จะต้องทำการแนบตาราง ไปกับตัวข้อมูล โดยหลังจากที่ได้ทำการคำนวณข้อมูลต้นฉบับแล้ว ก็จะสร้างตารางขึ้นมาเพื่อบอกว่ารหัสตัวใด ถูกแทนค่าด้วยข้อมูลตัวใด หลังจากนั้น ก็จะส่งทั้งตาราง และข้อมูลไปด้วยกัน เพื่อให้ผู้รับสามารถใช้ตารางดังกล่าว ในการถอดรหัสข้อมูลในภายหลังได้ ทั้งนี้ อัลกอริทึมต่าง ๆ ก็จะมีวิธีการอ่านตารางที่แตกต่างกัน ขึ้นอยู่กับการออกแบบ



ภาพประกอบที่ 3-2 การบีบอัดข้อมูลโดยไม่แนบตารางไปกับข้อมูล

ภาพประกอบที่ 3-2 แสดงถึงการบีบอัดข้อมูล โดยไม่ต้องทำการแนบตารางข้อมูลไปกับข้อมูลที่ถูกบีบอัด โดยที่วิธีนี้จะทำการอ่านข้อมูลทางสถิติออกมา แล้วทำการคำนวณค่าสถิติดังกล่าวร่วมกับข้อมูลต้นฉบับตัวแรก และได้ผลลัพธ์ออกมาเป็นข้อมูลที่ถูกเข้ารหัส แล้วทำการจัดเก็บข้อมูลที่ถูกเข้ารหัสนี้เอาไว้ จากนั้นจึงนำตัวข้อมูลต้นฉบับดังกล่าวไปวิเคราะห์ และปรับปรุงข้อมูลทางสถิติ เพื่อรอข้อมูลตัวถัดไปที่จะถูกบีบอัด ในขณะที่ผู้ถอดรหัสข้อมูลนั้น ก็จะเริ่มจากการอ่านข้อมูลตัวแรกเข้ามา แล้วทำการถอดรหัสด้วยข้อมูลทางสถิติที่มีอยู่ (ซึ่ง ณ จุดเริ่มแรกของการถอดรหัส จะไม่มีข้อมูลทางสถิติ) จากนั้นจะได้ข้อมูลต้นฉบับตัวแรกออกมาแล้วจึงทำการนำข้อมูลมาวิเคราะห์ เพื่อปรับปรุงข้อมูลทางสถิติที่มีอยู่ เพื่อใช้ในการคำนวณในรอบต่อไป ทำเช่นนี้ไปเรื่อย ๆ จนไม่มีข้อมูลเหลืออยู่ จะเห็นได้ว่าการบีบอัดข้อมูลทั้งสองแบบนั้น จะมีความแตกต่างกันในเรื่องของการสร้างตาราง ซึ่งในจุดนี้ จะเป็นข้อได้เปรียบ เสียเปรียบกัน ในสภาวะการใช้งานที่แตกต่างกัน

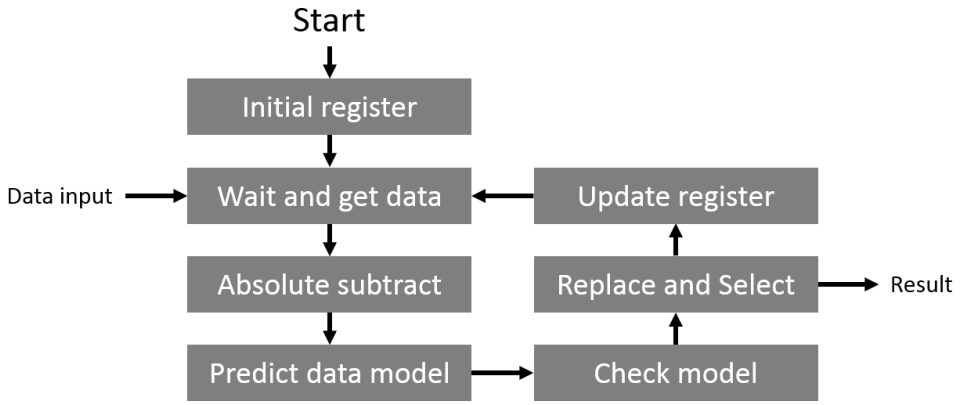
อัลกอริทึมที่จะทำการออกแบบนั้น จะต้องมีความเหมาะสมสำหรับการนำมาใช้งานกับระบบเครือข่ายเซนเซอร์ไร้สาย ทั้งนี้ การใช้งานระบบเครือข่ายเซนเซอร์ไร้สายนั้นสามารถใช้งานเพื่อส่งข้อมูลได้หลายรูปแบบ เช่น อาจจะต้องการส่งข้อมูลในปริมาณน้อย ๆ ทุก ๆ 1 ชั่วโมง ซึ่งการส่งข้อมูลในลักษณะนี้ ไม่เหมาะสมกับการใช้ Huffman Algorithm เนื่องจากจะทำให้เกิด Overhead ที่มีขนาดใหญ่ เป็นผลให้การบีบอัดข้อมูลขาดประสิทธิภาพ ในขณะที่การใช้ Adaptive Huffman Algorithm จะให้ผลลัพธ์ที่ดีกว่า เนื่องจากไม่ต้องทำการแนบตารางไปกับข้อมูล จึงมีปริมาณ Overhead ที่น้อยกว่า ดังนั้น อัลกอริทึมที่จะถูกออกแบบนั้น ควรจะมีพฤติกรรมเช่นเดียวกับ Adaptive Huffman Algorithm และมุ่งเน้นการลด Overhead ที่เกิดจากการบีบอัดให้น้อยที่สุด เพื่อให้ครอบคลุมทุกลักษณะการใช้งานของเครือข่ายเซนเซอร์ไร้สาย

3.2 Algorithm Design

อัลกอริทึมที่จะนำเสนอประกอบไปด้วย 3 อัลกอริทึม ได้แก่ Double Zero (DZ), Single Zero (SZ) และ Single Zero with Average (SZAVG) ซึ่งทั้งสามวิธีนี้มีลำดับการทำงานที่เหมือนกัน

3.2.1 การเข้ารหัสข้อมูล

ในส่วนของวงจรเข้ารหัสที่ได้ทำการออกแบบไว้นั้น ทั้ง 3 อัลกอริทึมจะมีเค้าโครงการทำงานที่เหมือนกัน ดังภาพประกอบที่ 3-3



ภาพประกอบที่ 3-3 ภาพรวมการทำงานของอัลกอริทึมในการเข้ารหัส

ภาพประกอบที่ 3-3 แสดงถึงลำดับการทำงานในภาคของการเข้ารหัสของทั้ง 3 อัลกอริทึม ซึ่งมีการทำงานที่เหมือนกัน โดยในแต่ละขั้นตอน จะมีการทำงานดังนี้

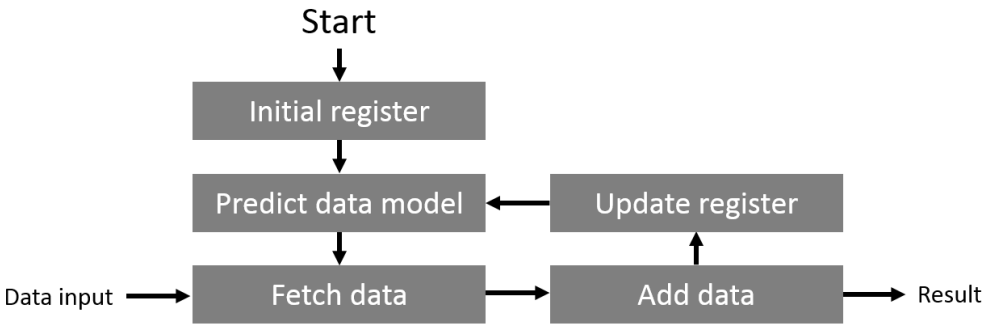
- Initial register เป็นขั้นตอนเริ่มต้นของการทำงาน จะทำการกำหนดค่าเริ่มต้นให้กับ Register ต่าง ๆ เช่น การ Reset ข้อมูลใน Register ให้เป็น 0
- Wait and get data เป็นขั้นตอนในการรอรับข้อมูล หรืออ่านข้อมูลเข้ามา 1 ตัว เพื่อให้ง่ายในการอธิบาย จะขอกำหนดให้ ข้อมูล 1 ตัว มีขนาด 8 บิต
- Absolute subtract เป็นขั้นตอนในการหาค่าสัมบูรณ์ของผลต่างระหว่าง ข้อมูลก่อนหน้า (Previous data : P-Data) กับข้อมูลปัจจุบัน (Current data : C-Data) โดยที่ผลลัพธ์ที่ได้จากขั้นตอนนี้ คือ ค่าสัมบูรณ์ของผลต่าง (|SUB|) และ Flag ซึ่งมีขนาด 1 บิต ใช้ในการบอกว่า ผลต่างที่ได้นั้น มีค่าเป็นลบหรือไม่
- Predict data model เป็นขั้นตอนในการทำนายว่าผลลัพธ์จาก Absolute subtract (ค่าผลต่าง) มีขนาดกี่บิต ตัวอย่างเช่น ถ้าทำนายว่า มีขนาด 5 บิต หมายถึงค่าสัมบูรณ์ของผลต่างที่ได้ จะมีค่าตั้งแต่ 0000 0000 ไปจนถึง 0001 1111 โดยการทำนายนั้น จะคำนวณมากจากค่าข้อมูลทางสถิติที่อยู่ใน Register

- Check model เป็นขั้นตอนการสร้างสัญญาณในการกำหนดผลลัพธ์ของการเข้ารหัสข้อมูลในรอบนั้น โดยที่รูปแบบของสัญญาณจะมีด้วยกัน 3 แบบ ได้แก่
  - รูปแบบที่ 1 ไม่ต้องทำการเข้ารหัส เนื่องจากการประเมินข้อมูลทางสถิติบอกว่า แม้ว่าจะทำนายโมเดลได้ถูกต้อง ผลลัพธ์ที่ได้จากการเข้ารหัสก็จะมีขนาดเท่ากับหรือใหญ่กว่าข้อมูลต้นฉบับ จึงไม่จำเป็นต้องเข้ารหัส
  - รูปแบบที่ 2 ทำนายโมเดลได้ถูกต้อง คือ ค่าผลต่างที่ได้จากขั้นตอน Absolute subtract มีค่าอยู่ในช่วงที่ทำนาย ก็จะได้สัญญาณในรูปแบบที่ 2
  - รูปแบบที่ 3 ทำนายโมเดลได้ไม่ถูกต้อง คือ ค่าผลต่างที่ได้จากขั้นตอน Absolutesubtract มีค่าอยู่นอกช่วงที่ได้ทำนายไว้ ก็จะได้สัญญาณเป็นรูปแบบที่ 3
- Replace and select เป็นขั้นตอนในการสร้างผลลัพธ์ของการเข้ารหัส โดยจะมีรูปแบบของผลลัพธ์อยู่ 3 รูปแบบ ตามสัญญาณที่ได้จากขั้นตอน Check Model
  - รูปแบบที่ 1 จะได้ผลลัพธ์เป็น {Data input} คือเอาข้อมูลฉบับที่ได้รับมาส่งออกไปเป็นเอาต์พุตได้เลย ทำให้ผลลัพธ์มีขนาด 8 บิต
  - รูปแบบที่ 2 จะได้ผลลัพธ์เป็น {0, Flag, Replacement} โดยที่ Flag นั้นจะได้มาจากขั้นตอน Absolute subtract ส่วน Replacement ได้มาจากการเทียบผลลัพธ์จาก Absolute subtract กับโมเดลที่ทำนายไว้ โดยเลือกเฉพาะบางบิตของ Absolute subtract ที่อยู่ในตำแหน่งเดียวกับโมเดลที่มีค่าเป็น 1 ตัวอย่างเช่น ถ้ากำหนดให้โมเดลมีค่าเป็น 0001 1111 และ Absolute Subtract มีค่าเป็น 0001 1001 จะได้ในส่วนของ Replacement เป็น 1 1001 ซึ่งมีจำนวน 5 บิต จะเห็นได้ว่า ในขั้นตอนนี้ จะได้ผลลัพธ์ที่มีขนาดไม่คงที่ ขึ้นอยู่กับโมเดล ที่คำนวณออกมา
  - รูปแบบที่ 3 จะได้ผลลัพธ์เป็น {1, Data input} คือผลลัพธ์จะเริ่มต้นด้วย 1 แล้วตามด้วยข้อมูลต้นฉบับ ทำให้ผลลัพธ์มีขนาด 9 บิต
- Update register เป็นขั้นตอนในการนำผลลัพธ์ที่ได้จากขั้นตอน Absolute subtract มาทำการวิเคราะห์ เพื่อเพิ่มค่าของ Register โดยที่จะต้องประเมินผลลัพธ์ดังกล่าว ว่าจัดอยู่ในลักษณะของโมเดลแบบใด ซึ่ง DZ และ SZ จะมีการจำแนกโมเดลที่แตกต่างกัน ส่วน SZAVG จะมีการจำแนกโมเดลเหมือนกันกับ SZ

หลังจากเสร็จขั้นตอน Update register ระบบก็จะวนกลับไปทำงานในขั้นตอน Wait and get data อีก และทำวนเช่นนี้ไปเรื่อย ๆ สำหรับ DZ, SZ และ SZAVG ก็จะมีรายละเอียดที่แตกต่างออกไปจากที่ได้อธิบายไว้ข้างต้นเล็กน้อย

3.2.2 การถอดรหัสข้อมูล

ในส่วนของการถอดรหัสข้อมูล จะมีลำดับการทำงานที่คล้ายคลึงกับการเข้ารหัสในหลาย ๆ ส่วน ดังภาพประกอบที่ 3-4



ภาพประกอบที่ 3-4 ภาพรวมการทำงานของอัลกอริทึมในการถอดรหัส

ภาพประกอบที่ 3-4 เป็นภาพที่แสดงลำดับการถอดรหัสข้อมูลของทั้ง 3 อัลกอริทึม ซึ่งมีลำดับการทำงานที่เหมือนกัน แต่จะแตกต่างกันในเรื่องของการสร้างโมเดล และจำนวน Register โดยในแต่ละขั้นตอนมีลำดับการทำงานดังนี้

- Initial register เป็นขั้นตอนเริ่มต้นของการทำงาน ซึ่งจะทำการกำหนดค่าเริ่มต้นให้กับ Register ต่าง ๆ เช่น การ Reset ข้อมูลใน Register ให้เป็น 0
- Predict data model เป็นขั้นตอนในการทำนายรูปแบบในการเข้ารหัส ว่าฝั่งที่ทำการเข้ารหัสข้อมูล จะเข้ารหัสมาในรูปแบบใด โดยจะสามารถแบ่งได้เป็น 2 รูปแบบ
  - รูปแบบที่ 1 ข้อมูลที่กำลังจะเข้ามา ไม่ได้ถูกทำการเข้ารหัส เนื่องจากการเข้ารหัสข้อมูลไม่ทำให้ขนาดของข้อมูลเล็กลง จึงไม่ต้องทำการบีบอัด
  - รูปแบบที่ 2 ข้อมูลที่กำลังจะเข้ามาถูกเข้ารหัสเอาไว้ แต่ในขณะเดียวกันการเข้ารหัสเอง ก็สามารถแบ่งได้เป็นสองกรณี คือ เข้ารหัสโดยทำนายโมเดลได้ถูกต้อง และเข้ารหัสโดยทำนายโมเดลไม่ถูกต้อง ในกรณีที่เข้ารหัสโดยทำนายโมเดลได้ถูกต้องนั้น ยังขึ้นอยู่กับโมเดลที่ถูกทำนายว่าทำนายเป็นโมเดลรูปแบบใด โดยการทำนายนั้น จะนำข้อมูลทางสถิติที่อยู่ใน Register มาทำการคำนวณ
- Fetch data เป็นขั้นตอนในการอ่านข้อมูลเข้ามาในระบบ โดยที่การอ่านข้อมูลจะถูกแบ่งออกเป็น 2 กรณี ตามการทำนายโมเดลในขั้นตอน Predict data model โดยที่
  - กรณีที่ 1 ถ้าผลลัพธ์จาก Predict data model ได้เป็นรูปแบบที่ 1 ให้ทำการ Fetch ข้อมูลเข้ามา 8 บิต

- กรณีที่ 2 ถ้าผลลัพธ์จาก Predict data model ได้เป็นรูปแบบที่ 2 ให้ทำการ Fetch ข้อมูลเข้ามา 1 บิต ก่อน แล้วตรวจสอบว่าข้อมูลดังกล่าวมีค่าเป็น 0 หรือ 1 ถ้ามีค่าเป็น 0 ให้ Fetch ข้อมูลเพิ่มเข้ามาตามโมเดลที่ได้ทำนายไว้บวกด้วย 1 (ค่าที่ Fetch เพิ่มเข้ามาคือ Flag) ถ้าหากมีค่าเป็น 1 ให้ทำการ Fetch ข้อมูลเข้ามาอีก 8 บิต

- Add data เป็นขั้นตอนที่จะทำการเพิ่มค่าข้อมูล หรือก็คือการคำนวณย้อนกลับขั้นตอน Absolute subtract แต่จะทำการบวกข้อมูล ในเฉพาะกรณีที่ข้อมูลถูกเข้ารหัส (กรณีที่ 2 ของขั้นตอน Fetch data) และข้อมูลที่ทำการ Fetch เข้ามาก่อน 1 บิต ในขั้นตอน Fetch data นั้นจะต้องมีค่าเป็น 0 เท่านั้นโดยการบวกค่า จะต้องทำการพิจารณา Flag ด้วย ถ้าหาก Flag มีค่าเป็น 0 ให้ทำการบวกค่าตามปกติ แต่ถ้าหากมีค่าเป็น 1 ให้เปลี่ยนจากการบวก เป็นการลบแทน โดยที่ตัวตั้งในการบวกหรือการลบ จะเป็นข้อมูลตัวก่อนหน้าที่ถูกถอดรหัสเสร็จสิ้น (Last result) แล้วข้อมูลที่ได้มาจากขั้นตอนนี้ ก็จะเป็นข้อมูลที่ถูกถอดรหัสเสร็จสิ้นโดยสมบูรณ์ (Result) แล้วทำการส่งข้อมูลนี้ออกไป (กรณีที่ข้อมูลไม่ได้ถูกทำการบวกข้อมูล ไม่จำเป็นต้องทำการถอดรหัส เนื่องจากข้อมูลดังกล่าว เป็นข้อมูลต้นฉบับ ซึ่งไม่ได้ถูกเข้ารหัสอยู่แล้ว)

- Update register ในขั้นตอนนี้ จะนำ Result ตัวก่อนหน้า มาหาค่าสัมบูรณ์ของผลต่าง กับ Result ตัวปัจจุบัน เพื่อนำผลลัพธ์ที่ได้มาทำการวิเคราะห์แล้วทำการเพิ่มค่าให้กับ Register

หลังจากจบขั้นตอน Update register แล้วจะกลับไปทำงานในขั้นตอน Predict data model อีกครั้ง ทำวนซ้ำเช่นนี้ไปเรื่อย ๆ โดยที่ภาพรวมในการทำงานของระบบนั้น จะเหมือนกันทั้ง DZ, SZ และ SZAVG แต่จะมีความแตกต่างกันในส่วนของการสร้างโมเดลและปริมาณ Register

3.2.3 Double Zero (DZ)

ก่อนที่จะอธิบายหลักการทำงานของ DZ จำเป็นจะต้องทำความเข้าใจเกี่ยวกับการเรียกชื่อโมเดลแบบ DZ เสียก่อน โดย DZ คือ Double Zero หรือคู่ศูนย์ โดยถ้ามีคู่ศูนย์อยู่ติดกันทางด้าน MSB 1 คู่ ก็จะเรียกว่า 1 DZ ถ้ามี 2 คู่ก็จะเรียกว่า 2 DZ ตัวอย่างเช่น

0100 1000	เรียกว่า	0 DZ
0000 0100	เรียกว่า	2 DZ
0010 1101	เรียกว่า	1 DZ
0000 0001	เรียกว่า	3 DZ

การทำงานในส่วน Predict data model ของ DZ นั้น จะมี Register คอยเก็บข้อมูลว่าในอดีตที่ผ่านมานั้น มี DZ แบบใดเข้ามาในระบบมากที่สุด ก็จะทำนายโมเดลตามรูปแบบ DZ นั้น เช่น ถ้าหาก 2 DZ มีค่ามากที่สุด ก็จะได้โมเดลเป็น 0000 1111 ซึ่งก็จะสามารถนำไปใช้ต่อในขั้นตอนอื่น ๆ

ขั้นตอนการทำงานของ DZ ในส่วนของการเข้ารหัส จะเป็นไปตามขั้นตอนดังภาพประกอบที่ 3-3 แต่เพื่อให้ง่ายต่อการทำความเข้าใจ จึงจะขออธิบายในลักษณะของ

Pseudocode ในแต่ละรอบของการเข้ารหัสจะสามารถแบ่งการทำงานออกเป็น 3 ส่วน ได้แก่ Prediction part, Output creation part และ Update part โดยแต่ละส่วนจะมีการทำงานดังนี้

```
// === === Encoder === ===

// Prediction part.
Mn = P(Reg);

// Output creation part.
Sn,Fn = AbsSub(Dn-1, Dn);
if (Mn ϵ {DZ0, DZ1})
       Rn = Dn;
       // Result size = 8 bit.
else if (Mn | Sn != Mn)
       Rn = {1, Dn};
       // Result size = 9 bit.
else if (Mn | Sn == Mn)
       Rn = {0, Fn, Sn(Mn bit)};
       // Result size = 3 – 7 bit.

// Update part.
Update(Reg, Sn);

```

1. Prediction part เป็นส่วนที่ใช้ทำนายลักษณะของโมเดลในรอบนั้น ๆ ว่าจะมีลักษณะตรงกับ 0DZ, 1DZ, 2DZ, 3DZ หรือ 4DZ แล้วเก็บผลการทำนายไว้ใน Mn (บรรทัดที่ 4) โดยวิธีการทำนายนั้น จะใช้การหาว่าใน Register (Reg) รูปแบบใดเคยปรากฏออกมามากที่สุด ก็จะเลือกเอารูปแบบนั้นมาเป็น Model
   หมายเหตุ 0DZ มีค่าเป็น 1111 1111
   1DZ มีค่าเป็น 0011 1111
   2DZ มีค่าเป็น 0000 1111
   3DZ มีค่าเป็น 0000 0011
   4DZ มีค่าเป็น 0000 0000
2. Output creation part เป็นส่วนที่ใช้สร้างผลลัพธ์โดยเริ่มจากการหาผลต่างระหว่างข้อมูลตัวปัจจุบัน (Dn) กับข้อมูลตัวก่อนหน้า (Dn-1) โดยผลลัพธ์ที่ได้จะประกอบด้วยค่าสัมบูรณ์ของผลต่าง (Sn) และทิศทางของผลต่าง (Fn)

จากนั้นจะทำการเปรียบเทียบเงื่อนไข โดยเริ่มจากเงื่อนไขที่ 1 (บรรทัดที่ 8) เปรียบเทียบว่า Mn มีค่าเป็น DZ0 หรือ DZ1 หรือไม่ ถ้าใช่ ให้ผลลัพธ์ (Rn) มีค่าเป็น Dn ได้ทันที เงื่อนไขที่ 2 เมื่อนำ Mn มา or กับ Sn แบบบิตต่อบิตแล้วมีค่าไม่เท่ากับ Mn ใช่หรือไม่ ถ้าใช่ ผลลัพธ์ที่ได้จะเป็น บิต 1 แล้วตามด้วย Dn เงื่อนไขที่ 3 ถ้าหากผล or ของ Mn กับ Sn มีค่าเท่ากับ Mn ผลลัพธ์ที่ได้ จะประกอบด้วยบิต 0 ตามด้วย Fn ตามด้วย Sn เลือกเฉพาะบิตที่ตรงกับบิตที่มีค่า 1 ของ Mn เรียงตามลำดับจากซ้ายไปขวา

3. Update part ขั้นตอนนี้จะทำการนำ Sn ไปเพิ่มค่าให้กับ Register (Reg) เพื่อใช้ในการทำนายในรอบต่อไป โดยการเพิ่มค่า จะพิจารณาว่า Sn ที่ได้ ตรงกับ DZ รูปแบบใด แล้วจึงเพิ่มค่าให้รูปแบบนั้นขึ้น 1 ค่า

หลังจากจบขั้นตอนการทำงานในส่วน Update part แล้ว ก็จะวนกลับไปทำงานในส่วนของ Prediction part อีกครั้ง แล้ววนซ้ำเช่นนี้ไปเรื่อย ๆ

ในส่วนของการถอดรหัสข้อมูลนั้น ก็จะมีลำดับการทำงานดังภาพประกอบที่ 3-4 การที่จะเข้าใจการทำงานในส่วนของการถอดรหัสข้อมูลนั้น จำเป็นที่จะต้องเข้าใจการเรียกชื่อโมเดลแบบต่าง ๆ ของ DZ เสียก่อน ซึ่งได้อธิบายไว้ในส่วนเข้ารหัสข้อมูลแล้ว เนื่องจากการเข้ารหัสแบบ DZ เป็นแบบ Fixed to Variable ทำให้การเข้ารหัสสามารถทำได้อย่างสะดวกเนื่องจากสามารถรับข้อมูลเข้ามาได้ครั้งละเท่า ๆ กัน แต่การถอดรหัสจะไม่ทราบว่าข้อมูลที่จะต้องนำมาใช้ถอดรหัสมีขนาดเท่าใด ซึ่งขนาดของข้อมูลจะขึ้นอยู่กับการทำนายโมเดลในแต่ละรอบ การถอดรหัสข้อมูลด้วยวิธี DZ จะแบ่งเป็น 3 ส่วน ได้แก่ Prediction part, Output creation part และ Update part โดยการทำงานจะเป็นดังแสดงใน Pseudocode ต่อไปนี้

```
1  // === === Decoder === ===
2
3  // Prediction part.
4  Mn = P(Reg);
5
6  // Output creation part.
7  if (Mn ϵ {DZ0, DZ1})
8        Rn = D[0:7];
9  else if (D[0] == 1'b1)
10       Rn = D[1:8];
11 else if (D[0] == 1'b0)
12       Fn = D[1];
13       Sn = D[2:2+(bit 1 of Mn)];
14       Rn = Rn-1 + pow(-1,Fn+1)*Sn;
15
```

```
16  Sn = AbsSub(Rn-1, Rn);
17
18  // Update part.
19  Update(Reg, Sn);
20
```

1. Prediction part เป็นส่วนที่ใช้ในการทำนายว่าโมเดล (Mn) ในรอบนั้น ๆ จะมีค่าตรงกับ 0DZ, 1DZ, 2DZ, 3DZ หรือ 4DZ โดยการทำงานจะเหมือนกันกับ Prediction part ของส่วนเข้ารหัสทุกประการ
2. Output creation part เป็นส่วนที่ใช้ในการสร้างผลลัพธ์ โดยจะเริ่มจากการพิจารณาเงื่อนไขแต่ละเงื่อนไข เริ่มจากเงื่อนไขที่ 1 ตรวจสอบว่า Mn มีค่าตรงกับ 0DZ หรือ 1DZ หรือไม่ ถ้าตรงกัน ให้ผลลัพธ์มีค่าเป็น 8 บิตล่าสุดได้ทันที เงื่อนไขที่ 2 ข้อมูลบิตแรกที่เข้ามาในระบบมีค่าเป็น 1 หรือไม่ ถ้าใช่ผลลัพธ์จะมีค่าเป็น 8 บิตถัดมาได้ทันที เงื่อนไขที่ 3 ถ้าข้อมูลบิตแรกมีค่าเป็น 1 ให้บิตถัดมาเป็นตัวระบุทิศทางของผลต่าง Fn และให้ บิตถัดมาอีกจำนวนเท่ากับจำนวนบิต 1 ใน Mn เป็น ค่าสัมบูรณ์ของผลต่าง (Sn) หลังจากได้ Fn และ Sn มาแล้ว ก็จะทำการหาผลลัพธ์ (Rn) โดยจะใช้ผลลัพธ์ครั้งก่อนหน้า (Rn-1) บวกหรือลบกับ Sn โดยจะบวกกันเมื่อ Fn เป็น 1 และจะลบกัน เมื่อ Fn เป็น 0 หลังจากพิจารณาเงื่อนไขทั้งหมดแล้ว จะทำการหาค่า Sn ด้วยการหาค่าสัมบูรณ์ของผลต่างระหว่าง Rn-1 กับ Rn
3. Update part เป็นส่วนที่จะทำการนำ Sn ที่ได้จากขั้นตอนก่อนหน้ามาเพิ่มค่าลงไปใน Reg โดยจะพิจารณาว่า Sn ที่ได้ ตรงกับโมเดลลักษณะใด แล้วจึงเพิ่มค่าให้กับรูปแบบนั้นขึ้น 1 ค่า

หลังจากจบขั้นตอนในส่วนของ Update part แล้วก็จะกลับไปทำงานในส่วนของ Prediction part อีก แล้วทำวนซ้ำเช่นนี้ไปเรื่อย ๆ ซึ่งในแต่ละรอบก็จะได้ค่าของ Register ที่เปลี่ยนแปลงไป

3.2.4 Single Zero (SZ)

การทำงานของ SZ จะมีลักษณะเหมือนกันกับ DZ แต่จะแตกต่างกันในเรื่องโมเดล โดยที่ DZ จะมีโมเดลทั้งหมด 5 แบบในขณะที่ SZ จะมีโมเดลทั้งหมด 9 แบบ โดยจะมีการจำแนกด้วยการนับจำนวนบิต 0 ทางด้าน MSB ที่อยู่ติดกัน ตัวอย่าง เช่น

0100 1000	เรียกว่า	1 SZ
0000 0100	เรียกว่า	5 SZ
0010 1101	เรียกว่า	2 SZ
0000 0001	เรียกว่า	7 SZ

ตัวอย่างข้างต้นเป็นการจำแนกรูปแบบโมเดลของ SZ ซึ่ง SZ จะมีรูปแบบโมเดลทั้งหมด 9 รูปแบบ ได้แก่ 0SZ, 1SZ, 2SZ, 3SZ, 4SZ, 5SZ, 6SZ, 7SZ และ 8SZ จะเห็นว่าแนวทาง

การทำงานของ SZ นั้น จะเหมือนกันกับ DZ แต่จะเพิ่มจำนวน Register ให้มากขึ้น โดยจะมี Pseudocode ในส่วนเข้ารหัส และส่วนถอดรหัส ดังนี้

```
// === === Encoder === ===

// Prediction part.
Mn = P(Reg);

// Output creation part.
Sn,Fn = AbsSub(Dn-1, Dn);
if (Mn ϵ {SZ0, SZ1, SZ2})
    Rn = Dn;
    // Result size = 8 bit.
else if (Mn | Sn != Mn)
    Rn = {1, Dn};
    // Result size = 9 bit.
else if (Mn | Sn == Mn)
    Rn = {0, Fn, Sn(Mn bit)};
    // Result size = 3–7 bit.

// Update part.
Update(Reg, Sn);
```

```
// === === Decoder === ===

// Prediction part.
Mn = P(Reg);

// Output creation part.
if (Mn ϵ {SZ0, SZ1, SZ2})
    Rn = D[0:7];
else if (D[0] == 1’b1)
    Rn = D[1:8];
else if (D[0] == 1’b0)
    Fn = D[1];
    Sn = D[2:2+(bit 1 of Mn)];
    Rn = Rn-1 + pow(-1,Fn+1)*Sn;

Sn = AbsSub(Rn-1, Rn);

// Update part.
Update(Reg, Sn);
```

เมื่อพิจารณา Pseudocode ข้างต้นแล้ว จะพบว่าขั้นตอนการทำงานเหมือนกันกับ DZ เกือบทั้งหมด ต่างกันเพียงบรรทัดที่ 8 ของส่วนเข้ารหัส กับบรรทัดที่ 7 ของส่วนถอดรหัส เท่านั้น ซึ่งความแตกต่างนี้ เกิดจากปริมาณ Register ที่ไม่เท่ากัน ซึ่งจะส่งผลให้ประสิทธิภาพในการบีบอัดข้อมูลแตกต่างกันด้วย

3.2.5 Single Zero with Average (SZAVG)

การทำงานของ SZAVG จัดว่าเหมือนกับการทำงานของ SZ แต่ถูกปรับเปลี่ยนวิธีคำนวณหาโมเดล เนื่องจากการคำนวณหาโมเดลของ SZ และ DZ นั้น จะใช้การสร้างโมเดลจากรูปแบบที่ถูกพบมากที่สุด แต่ SZAVG นั้น จะทำการหาค่าเฉลี่ยของ Register เพื่อให้ได้โมเดลที่อยู่กึ่งกลางที่สุด โดยมีสูตรในการคำนวณเพื่อหาโมเดลดังนี้

$$Model = \frac{0(0SZ) + 1(1SZ) + 2(2SZ) + \ \dots\ + 8(8SZ)}{0SZ + 1SZ + 2SZ + \ \dots\ + 8SZ} \tag{3.1}$$

โดยจากสูตรข้างต้น 0SZ นั้น จะหมายถึงค่าที่อยู่ใน Register 0SZ และในกรณีที่ตัวหารมีค่าเป็น 0 ให้ผลลัพธ์ของการหารเป็น 0 ทันที เพื่อป้องกันการเกิดปัญหาจากการหาผลลัพธ์จากการหารไม่ได้ จากสูตรข้างต้นสามารถแสดงตัวอย่างในการคำนวณหาโมเดลดังนี้

กำหนดให้ข้อมูลที่อยู่ภายใน Register ต่าง ๆ มีค่าดังนี้ 0SZ = 1, 1SZ = 0, 2SZ = 1, 3SZ = 1, 4SZ = 8, 5SZ = 2, 6SZ = 3, 7SZ =1 และ 8SZ = 1 ซึ่งจากข้อมูลข้างต้น สามารถแทนค่าในสูตรได้ ดังนี้

$$Model = \frac{0(1) + 1(0) + 2(1) + 3(1) + 4(8) + 5(2) + 6(3) + 7(1) + 8(1)}{1 + 0 + 1 + 1 + 8 + 2 + 3 + 1 + 1}$$

ซึ่งจากการแทนค่าข้างต้น จะได้ผลลัพธ์เป็น 80/18 = 4.44 ซึ่งเมื่อได้ผลลัพธ์ออกมาเป็นเศษส่วน ให้ทำการปัดเศษส่วนลง ทำให้ได้ผลลัพธ์เป็น 4 หรือก็คือได้โมเดลเป็น 4SZ

ในส่วนของการทำงานของ SZAVG นั้น นอกจากการคำนวณโมเดลแล้ว ส่วนอื่นจะไม่มีความแตกต่างจาก SZ เลย ทั้งส่วนเข้ารหัส และส่วนถอดรหัส ทำให้ Pseudocode ของ SZAVG เหมือนกันกับ SZ เกือบทุกประการ ดังแสดงต่อไปนี้

```
// === === Encoder === ===

// Prediction part.
Mn = PAVG(Reg);

// Output creation part.
Sn,Fn = AbsSub(Dn-1, Dn);
if (Mn ϵ {SZ0, SZ1, SZ2})
    Rn = Dn;
    // Result size = 8 bit.
else if (Mn | Sn != Mn)
    Rn = {1, Dn};
       // Result size = 9 bit.
else if (Mn | Sn == Mn)
    Rn = {0, Fn, Sn(Mn bit)};
    // Result size = 3–7 bit.

// Update part.
Update(Reg, Sn);
```

```
// === === Decoder === ===

// Prediction part.
Mn = PAVG(Reg);

// Output creation part.
if (Mn ϵ {SZ0, SZ1, SZ2})
    Rn = D[0:7];
else if (D[0] == 1’b1)
    Rn = D[1:8];
else if (D[0] == 1’b0)
    Fn = D[1];
    Sn = D[2:2+(bit 1 of Mn)];
    Rn = Rn-1 + pow(-1,Fn+1)*Sn;

Sn = AbsSub(Rn-1, Rn);

// Update part.
Update(Reg, Sn);
```

ความแตกต่างเพียงเล็กน้อยระหว่าง Pseudocode ของ SZAVG กับ Pseudocode ของ SZ อยู่ในบรรทัดที่ 4 ของทั้งส่วนเข้ารหัส และส่วนถอดรหัส โดยที่ SZ และ DZ จะใช้ฟังก์ชัน P(Reg) ในขณะที่ SZAVG จะใช้ PAVG(Reg) เนื่องมาจาก SZAVG ทำนายโมเดลโดยใช้การหาค่าเฉลี่ย ขณะที่ SZ และ DZ จะหาโมเดลจากรูปแบบที่เกิดขึ้นมากที่สุด ทำให้ SZAVG ใช้ฟังก์ชันที่ต่างออกไป แต่ในส่วนอื่น ๆ ของ Pseudocode จะมีการทำงานเหมือนกับ SZ ทุกประการ

ในส่วนของการทำงานของทั้ง 3 วิธีที่ได้กล่าวมา มีการทำงานที่ไม่ซับซ้อนมากนัก เนื่องจากการทำงานที่ซับซ้อนมาก จะต้องอาศัยทรัพยากรของระบบที่มากตามไปด้วย อีกทั้งยังอาจใช้เวลาในการเข้ารหัสที่นานกว่า จึงขาดความเหมาะสมที่จะใช้งานกับระบบเครือข่ายเซนเซอร์ไร้สาย ซึ่งทั้ง 3 วิธีนั้น เป็นวิธีที่มีการทำงานไม่ซับซ้อน อีกทั้งยังไม่ต้องมีการวนรอบเพื่อคำนวณเหมือนกับ Adaptive Huffman ทำให้มีความเหมาะสมกับระบบเครือข่ายเซนเซอร์ไร้สายมากกว่า ทั้ง 3 วิธีที่ได้ออกแบบไว้ มีความคล้ายคลึงกันมาก มีความแตกต่างกันเฉพาะส่วนของ Register และการทำนายโมเดลเท่านั้น ซึ่งข้อแตกต่างในจุดนี้ ก็จะทำให้ความสามารถในการบีบอัดข้อมูลมีความแตกต่างกันด้วย

3.3 Buffer Design

ในส่วนของการนำอัลกอริทึมไปใช้งานจริง จะมีข้อจำกัดที่เพิ่มขึ้นอีกประการหนึ่ง คือขนาดของ Register จะต้องมีขนาดที่ชัดเจน ทำให้ไม่สามารถที่จะเพิ่มค่าของ Register ขึ้นไปเรื่อย ๆ ได้ เนื่องจากเมื่อระบบทำงานไประยะหนึ่ง จะทำให้ Register ถูกเพิ่มค่าจนถึงค่าสูงสุดที่จะสามารถเก็บค่าได้ จากนั้นก็จะเกิดการ Overflow ขึ้น ทำให้เกิดปัญหาในการใช้งาน จึงจำเป็นที่จะต้องทำการจำกัดขนาดของ Register รวมทั้งจะต้องมีวิธีการจัดการกับค่าต่าง ๆ ที่เหมาะสม เพื่อให้สามารถจำกัดขนาดของ Register ในขณะที่กระบวนการในการบีบอัดข้อมูลสามารถดำเนินต่อไปได้อย่างต่อเนื่อง แต่อย่างไรก็ตาม การเพิ่มส่วนของการจัดการ Register เข้ามา จะทำให้การเข้ารหัสข้อมูลมีประสิทธิภาพที่แตกต่างออกไปจากเดิม

หลักในการจัดการ Register คือ การป้องกันการเกิด Overflow จากการดำเนินการกับ Register หรือก็คือ สามารถที่จะเพิ่มค่าของ Register ได้โดยไม่เกิดการ Overflow ซึ่งการจัดการ Register จะต้องเพิ่มขั้นตอนเข้าไปในการทำงานหลัก และอาจจะต้องเข้าไปแก้ไขรายละเอียดในการทำงานของขั้นตอน Update register ด้วย ในงานวิจัยชิ้นนี้ได้มีการออกแบบการจัดการกับ Register ไว้ 2 วิธีด้วยกัน คือ Limited Buffer (LB) และ Limited Buffer by Shift (LBS)

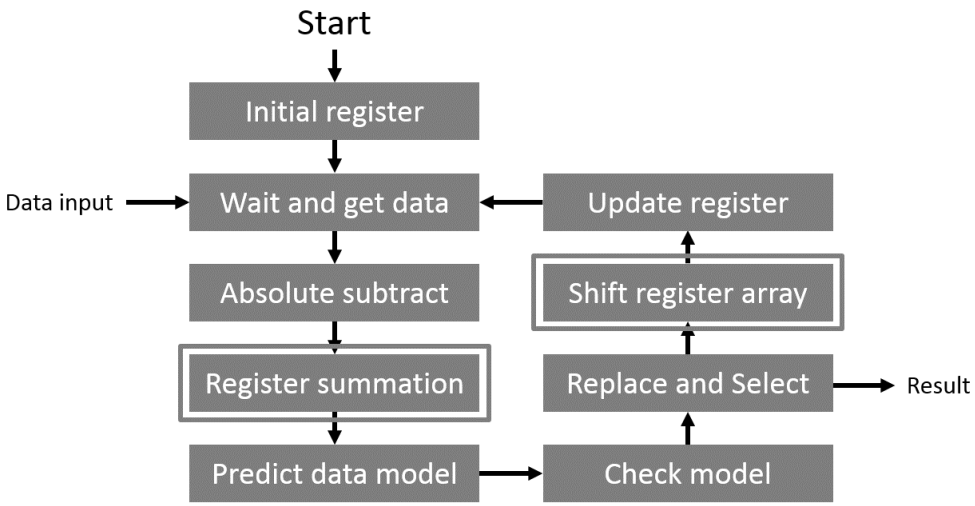
3.3.1 Limited Buffer (LB)

การจัดการ Register ด้วยวิธีนี้ จำเป็นที่จะต้องเข้าไปแก้ไขการทำงานของ Register เนื่องจากการทำงานแบบเก่า จะเก็บเพียงแค่ว่า มีการเข้ามาของแต่ละโมเดลเป็นจำนวนกี่ครั้ง ทำให้มีจำนวน Register เท่ากันกับจำนวนโมเดลที่มี ส่วนการทำงานโดยใช้ LB นั้น จะทำการเก็บข้อมูลว่าในแต่ละรอบ เกิดโมเดลรูปแบบใด การใช้วิธีนี้ จำเป็นที่จะต้องกำหนดเอาไว้ก่อนว่า จะเก็บข้อมูลย้อนหลังทั้งหมดกี่รอบ แล้วการจะนำข้อมูลมาใช้ ก็จะต้องทำการหาผลรวมของข้อมูลที่เก็บเอาไว้ทั้งหมด เสียก่อน ดังภาพประกอบที่ 3-5

	0DZ	1DZ	2DZ	3DZ	4DZ
	0	1	0	0	0
	0	0	0	1	0
	0	0	1	0	0
	0	0	1	0	0
Summation	0	1	2	1	0

ภาพประกอบที่ 3-5 การจัดการ Register ด้วยวิธี Limited Buffer

ภาพประกอบที่ 3-5 แสดงวิธีการจัดเก็บข้อมูลแบบ LB ซึ่งจากรูปตัวอย่าง เป็นการเก็บข้อมูลย้อนหลัง 4 ครั้งล่าสุด เมื่อต้องการนำข้อมูลมาใช้ ก็จะต้องหาผลรวมของ Buffer ทั้งหมด เพื่อให้ทราบว่ารูปแบบของโมเดลแบบใด ที่เข้ามาในระบบมากที่สุด โดยที่การจะนำ LB มาใช้งานร่วมกันกับ DZ,SZ หรือ SZAVG จะต้องเพิ่มขั้นตอนการทำงานเข้าไปดังภาพประกอบที่ 3-6



ภาพประกอบที่ 3-6 การนำ Limited Buffer มาทำงานร่วมกับการเข้ารหัสข้อมูล

ภาพประกอบที่ 3-6 การนำ Limited Buffer มาทำงานร่วมกับส่วนเข้ารหัสของ DZ, SZ และ SZAVG จะต้องทำการเพิ่มขั้นตอนเข้ามา 2 ขั้นตอน คือ Register summation และ Shift register array โดยแต่ละขั้นตอนมีการทำงานดังนี้

- Register summation เป็นขั้นตอนในการหาผลรวมของค่าทั้งหมดใน Buffer ก่อนที่จะนำผลรวมที่ได้ มาใช้งานในขั้นตอน Predict data model
- Shift register array เป็นขั้นตอนในการเลื่อนค่าต่าง ๆ ใน Array ตัวอย่างเช่น ถ้าหากในเวลาปัจจุบัน ภายใน Array มีค่าต่าง ๆ เหมือนกับภาพประกอบที่ 3-5

เมื่อทำการเลื่อนค่าใน Array แล้ว จะทำให้ผลลัพธ์ในขั้นตอนนี้ เป็นไปตามภาพประกอบที่ 3-7

0DZ	1DZ	2DZ	3DZ	4DZ
0	0	0	1	0
0	0	1	0	0
0	0	1	0	0
0	0	0	0	0

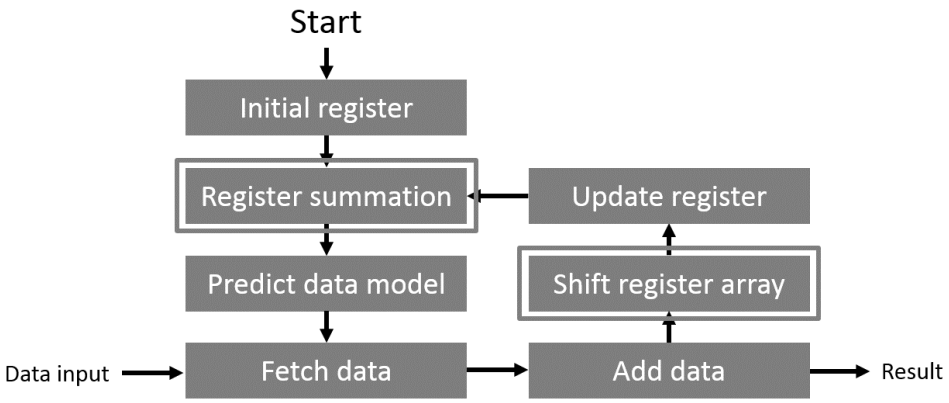
ภาพประกอบที่ 3-7 ผลลัพธ์จากการทำงานในขั้นตอน Shift register array

จากภาพประกอบที่ 3-7 จะสามารถสังเกตได้ว่า ในแถวล่างสุดของ Array มีค่าเป็น 0 ทั้งหมด เนื่องจากข้อมูลในแถวล่างสุดของ Array จะถูกเพิ่มค่าในขั้นตอน Update register ซึ่งขั้นตอน Update register นั้น ก็จะต้องมีการดัดแปลงการทำงานด้วยเล็กน้อย เพื่อให้สามารถใช้งานร่วมกับ LB ได้ คือสมมติให้จากเดิมนั้น ในขั้นตอนนี้สามารถที่จะเพิ่มค่าลงไปใน Register 0DZ, 1DZ, 2DZ, 3DZ และ 4DZ ได้ทันที แต่ถ้าหากมีการทำงานร่วมกันกับ LB จะทำการวิเคราะห์ข้อมูลที่ได้จาก Absolute subtract ว่าตรงกับโมเดลแบบใด แล้วจะทำการเพิ่มค่าของแถวสุดท้ายของ Array ขึ้น 1 ค่า ซึ่งถ้าหากว่าเป็นการทำงานต่อเนื่องจากข้อมูลในรูปที่ 13 และผลลัพธ์ของ Absolute subtract มีค่าตรงกับ 1DZ ก็จะทำให้ข้อมูลใน Array มีค่าเหมือนกับภาพประกอบที่ 3-8

0DZ	1DZ	2DZ	3DZ	4DZ
0	0	0	1	0
0	0	1	0	0
0	0	1	0	0
0	1	0	0	0

ภาพประกอบที่ 3-8 การเพิ่มค่าลงใน Array เมื่อค่าที่เพิ่มเข้าไปเป็นโมเดล 1DZ

ในขณะเดียวกัน ถ้าหากนำ LB มาใช้ร่วมกันกับการถอดรหัสข้อมูล ก็จะต้องมีการเพิ่มขั้นตอนการทำงาน และมีการแก้ไขการทำงานของ Update register โดยลำดับการทำงานจะเป็นดังภาพประกอบที่ 3-9



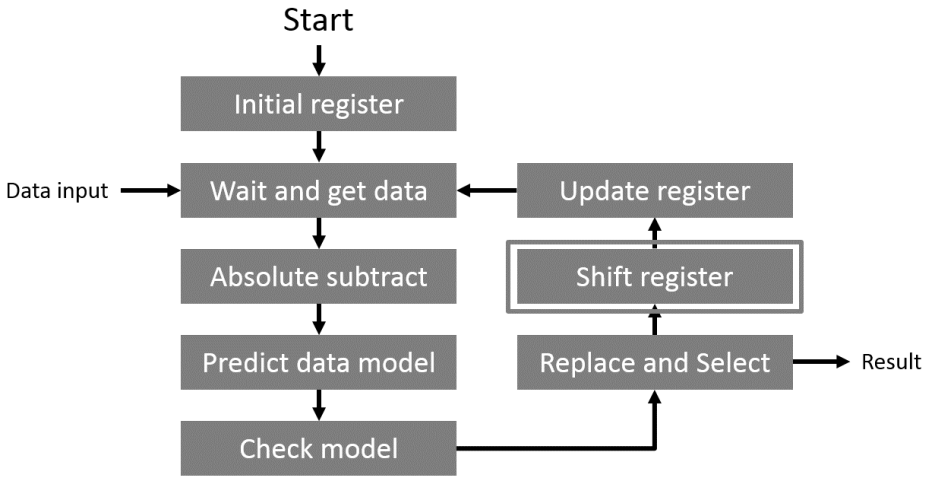
ภาพประกอบที่ 3-9 การเพิ่มขั้นตอนการทำงานเมื่อใช้งาน LB ร่วมกับการถอดรหัสข้อมูล

จากภาพประกอบที่ 3-9 ขั้นตอนที่ถูกเพิ่มขึ้นมา คือ Register summation และ Shift register array ซึ่งจะมีการทำงานเหมือนกันกับ Register summation และ Shift register array ในภาคของการเข้ารหัส ทุกประการ และการ Update register ก็จะถูกแก้ไขการทำงานในลักษณะเดียวกัน

การใช้วิธี LB ในการจัดการ Register นั้น มีข้อเสียประการหนึ่ง คือจำเป็นที่จะต้องสร้าง Register เพิ่มขึ้นมาจำนวนมาก ซึ่งทำให้ต้องใช้ทรัพยากรในระบบค่อนข้างมาก จึงจำเป็นที่จะต้องพัฒนาวิธีอื่นที่น่าจะใช้ทรัพยากรน้อยกว่า LB

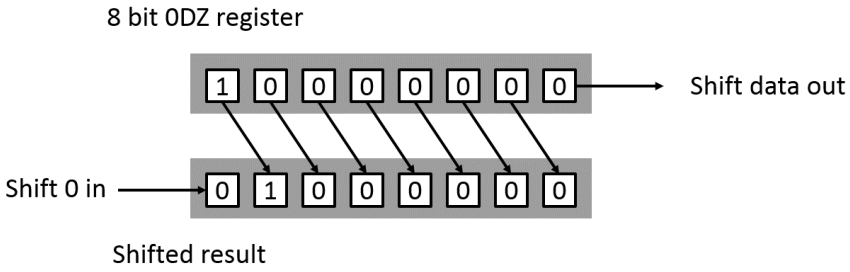
3.3.2 Limited Buffer by Shift (LBS)

วิธีนี้ถูกพัฒนาขึ้นมาหลังจาก LB โดยมีแนวคิดที่ว่า LB มีการสร้าง Register เพิ่มเติมขึ้นมาเป็นจำนวนมาก ดังนั้น LBS เป็นอีกวิธีหนึ่งที่ถูกสร้างขึ้นมาเพื่อจัดการกับ Register โดยในส่วนของ LBS นั้น จะใช้วิธีการ Shift ค่าใน Register ลงเมื่อค่าของข้อมูลมีค่ามากเกินไปโดยการที่จะนำ LBS มาใช้งานร่วมกันกับการเข้ารหัสของ DZ, SZ และ SZAVG จำเป็นที่จะต้องเพิ่มขั้นตอน เข้าไปในลำดับการทำงานหลัก ดังภาพประกอบที่ 3-10



ภาพประกอบที่ 3-10 ขั้นตอนการทำงานในการเข้ารหัส เมื่อใช้งานร่วมกับ LBS

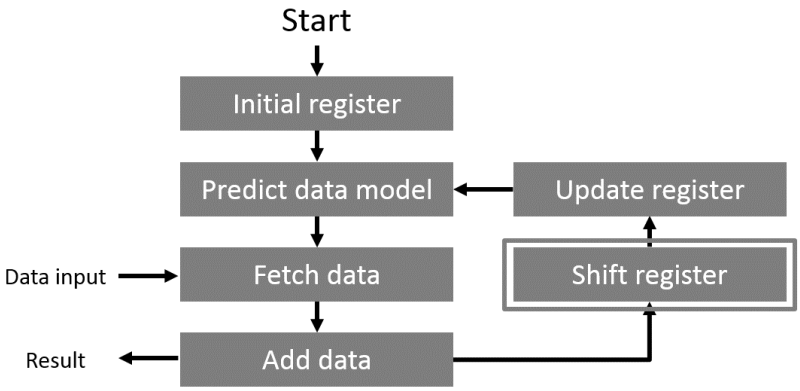
จากภาพประกอบที่ 3-10 เมื่อต้องการนำ LBS มาใช้งานร่วมกับ DZ, SZ และ SZAVG จะต้องมีการเพิ่มขั้นตอน Shift register เข้ามาในระบบ โดยที่ขั้นตอนดังกล่าวใช้ในการตรวจสอบ ว่ามี Register ตัวใดตัวหนึ่งมี MSB มีค่าเป็น 1 หรือไม่ ถ้าไม่มี Register ตัวใดเลย ที่มี MSB เป็น 1 ก็จะไม่มีการดำเนินการใด ๆ ในขั้นตอนนี้ แต่ถ้าหากมี Register ตัวใดตัวหนึ่งมี MSB เป็น 1 ให้ทำการ Shift right ทุก Register จำนวน 1 ครั้ง โดยที่การ Shift register นั้น จะมีการทำงานดังภาพประกอบที่ 3-11



ภาพประกอบที่ 3-11 การ Shift ข้อมูลในขั้นตอน Shift register

ภาพประกอบที่ 3-11 เป็นการยกตัวอย่าง Register 0DZ ซึ่งมีค่า MSB เป็น 1 และต้องทำการ Shift right โดยจะทำการ Shift LSB ทิ้งไป และ Shift ค่า 0 เข้ามาทาง MSB โดยจะต้องทำการ Shift ทุก Register กล่าวคือ ต้องทำการ Shift ค่าใน Register 1DZ, 2DZ, 3DZ และ 4DZ ด้วย แม้ว่าจะมีเพียง Register 0DZ ที่มี MSB เป็น 1 ด้วยการทำงานในลักษณะนี้ จะทำให้มีขั้นตอนในการจัดการ Register น้อยลง และไม่ต้องทำการแก้ไขการทำงานของ Update register

ในส่วนของการนำ LBS มาใช้งานร่วมกับการถอดรหัสของ DZ, SZ และ SZAVG จะต้องเพิ่มขั้นตอนขึ้นมาเช่นกัน คือขั้นตอน Shift register ดังภาพประกอบที่ 3-12



ภาพประกอบที่ 3-12 ขั้นตอนการทำงานในการถอดรหัส เมื่อใช้งานร่วมกับ LBS

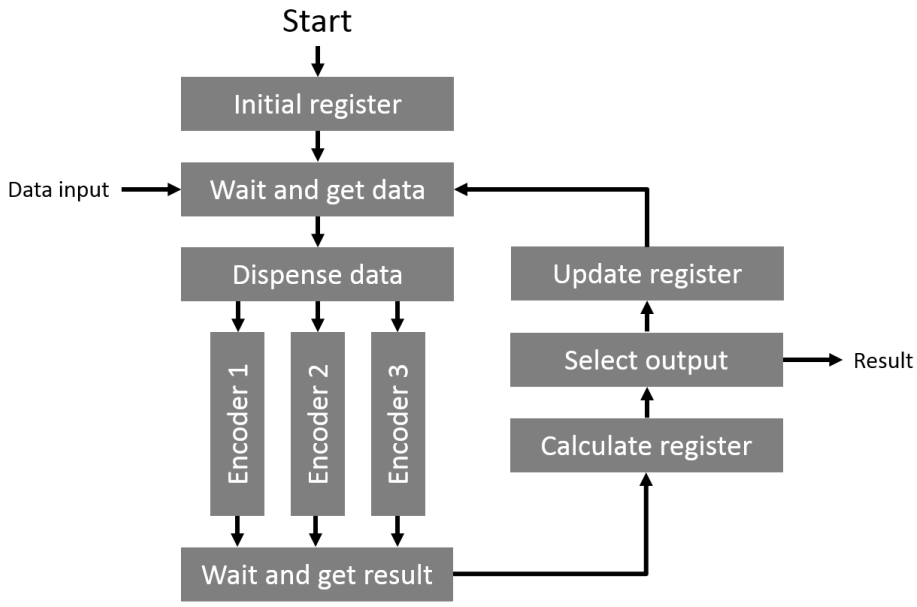
ภาพประกอบที่ 3-12 แสดงขั้นตอนที่ถูกเพิ่มขึ้นมา เมื่อนำ LBS มาใช้งานร่วมกับการถอดรหัสข้อมูล โดยที่การทำงานในขั้นตอน Shift register นั้น จะทำงานเหมือนกับการ Shift register ในภาคของการเข้ารหัสทุกประการ และไม่ต้องทำการแก้ไขการทำงานของ Update register ด้วยเช่นกัน

การนำ LB หรือ LBS มาใช้งานนั้น จำเป็นต้องใช้ทั้งส่วนเข้ารหัส และส่วนถอดรหัส เมื่อมีการนำ LB หรือ LBS มาใช้งาน จะทำให้ผลลัพธ์ในการเข้ารหัสข้อมูล แตกต่างจากการไม่นำ LB หรือ LBS มาใช้ ซึ่งผลลัพธ์ที่ต่างออกมานี้ อาจเป็นการเพิ่มหรือลดประสิทธิภาพของการบีบอัดข้อมูลก็ได้ แต่อย่างไรก็ตาม การนำ LB หรือ LBS มาใช้งาน สามารถที่จะจัดการกับข้อจำกัดของ Register ได้

3.4 Hybrid Algorithm

การบีบอัดข้อมูลในรูปแบบ Hybrid เกิดขึ้นมาจากแนวคิดที่ว่า แต่ละอัลกอริทึมจะเหมาะสมกับข้อมูลในลักษณะที่แตกต่างกัน ถ้าหากต้องการสร้างอัลกอริทึมที่มีประสิทธิภาพในการบีบอัดข้อมูลสูง ในทุกรูปแบบของข้อมูล ก็จำเป็นที่จะต้องมีการวิเคราะห์ตัวข้อมูล ว่ามีลักษณะอย่างไร แล้วเลือกใช้อัลกอริทึมที่เหมาะสม และเมื่อข้อมูลมีลักษณะที่แตกต่างออกไปก็จะต้องทำการเปลี่ยนไปใช้งานอัลกอริทึมที่มีความเหมาะสมมากกว่า

จากหลักการข้างต้นจะทำให้เกิดปัญหาที่ตามมา คือ จะทราบได้อย่างไร ว่าควรใช้อัลกอริทึมใด ในเวลาใด ซึ่ง Hybrid algorithm ที่ได้ถูกพัฒนาขึ้นนั้น เป็นเค้าโครงที่สามารถเพิ่มอัลกอริทึมเข้าไปในระบบ แล้วจะมีส่วนจัดการที่ทำการเรียกใช้อัลกอริทึมเหล่านั้น โดยที่แต่ละรอบของการเข้ารหัส ตัว Hybrid algorithm จะทำการเรียกใช้งานส่วนเข้ารหัส (Encoder) ของทุก ๆ อัลกอริทึมที่มี แล้วเก็บบันทึกข้อมูลไว้ว่าอัลกอริทึมใด ให้ผลลัพธ์ที่มีขนาดเท่าใด แล้วคำนวณว่าในอดีตที่ผ่านมานั้น อัลกอริทึมใด มีขนาดของผลลัพธ์รวมน้อยที่สุด ก็จะทำการเลือกเอาผลลัพธ์ของอัลกอริทึมนั้น เป็นผลลัพธ์สุดท้าย และจะถูกส่งไปยังฝั่งถอดรหัส ด้วยวิธีดังกล่าวก็จะทำให้สามารถเลือกอัลกอริทึมที่เหมาะสมในการใช้งานได้ Hybrid algorithm จะประกอบไปด้วยส่วนของการเข้ารหัส และส่วนของการถอดรหัส โดยส่วนเข้ารหัสจะมีการทำงาน ดังภาพประกอบที่ 3-13



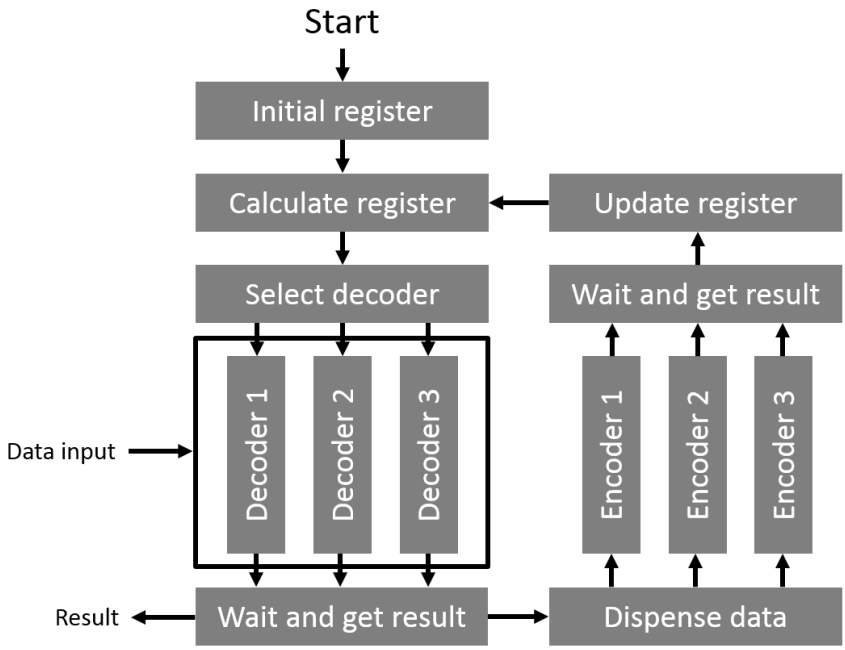
ภาพประกอบที่ 3-13 ลำดับการทำงานของ Hybrid algorithm ในส่วนของการเข้ารหัส

ภาพประกอบที่ 3-13 แสดงให้เห็นถึงลำดับการทำงานของ Hybrid algorithm ในส่วนของการเข้ารหัส ซึ่งมีด้วยกันหลายขั้นตอน แต่ละขั้นตอนจะมีการทำงานดังนี้

- Initial register ในขั้นตอนนี้จะทำการกำหนดค่าเริ่มต้นให้กับ Register ต่าง ๆ โดยปกติจะมี Array ของ Register ที่ใช้ในการเก็บผลลัพธ์ของการเข้ารหัส ซึ่งจะนำมาใช้คำนวณในขั้นตอน Calculate register โดยในที่นี้จะสมมติให้มี Register ดังนี้ A[0][x] A[1][x] และ A[2][x] โดย x คือจำนวนรอบที่ต้องการเก็บผลลัพธ์ย้อนหลัง ในขณะที่เลข 0, 1 และ 2 นั้นเป็นเลขประจำอัลกอริทึมแต่ละอัลกอริทึม โดยที่ขั้นตอนนี้ จะทำการกำหนดค่าของ Register ทุกตัวให้มีค่าเป็น 0
- Wait and get data เป็นขั้นตอนที่ใช้ในการรอรับข้อมูลเข้ามาในระบบ
- Dispense data เป็นขั้นตอนที่จะแจกจ่ายข้อมูลที่รับเข้ามาไปยังตัว Encoder ของแต่ละอัลกอริทึม เพื่อให้ได้ผลลัพธ์ของการเข้ารหัสแต่ละแบบ
- Wait and get result เป็นขั้นตอนในการรอรับข้อมูลผลลัพธ์ของการเข้ารหัสของแต่ละอัลกอริทึม เนื่องจากแต่ละอัลกอริทึมมีระยะเวลาในการเข้ารหัสที่ไม่เท่ากัน
- Calculate register เป็นขั้นตอนในการหาผลรวมของ Register เพื่อหาว่าผลลัพธ์จากการเข้ารหัสโดยรวมนั้น อัลกอริทึมใดสามารถเข้ารหัสได้ดีที่สุด
- Select output เป็นขั้นตอนในการเลือกผลลัพธ์จากแต่ละอัลกอริทึม โดยให้นำผลการคำนวณจากขั้นตอน Calculate register มาเป็นตัวตัดสินใจ แล้วทำการส่งข้อมูลออกไปยังฝั่งที่ทำการถอดรหัส

- Update register เป็นการนำผลลัพธ์ที่ได้จากการเข้ารหัส ที่ได้รวบรวมมาจากขั้นตอน Wait and get result มาทำการวิเคราะห์ เพื่อเพิ่มค่าเข้าไปใน Register array โดยในการเพิ่มค่านั้น จะต้องทำการ Shift ค่าข้อมูลเดิมที่เก่าที่สุดใน Register array ออกไป แล้วเพิ่มผลลัพธ์ในรอบนี้เข้าไปใน Register array แทน

หลังจากจบขั้นตอน Update register แล้ว ก็จะวนกลับไปทำงานในขั้นตอน Wait and get data แล้วทำวนซ้ำเช่นนี้ไปเรื่อย ๆ โดยที่การทำงานในลักษณะนี้ จะทำให้สามารถเลือกวิธีเข้ารหัสที่น่าจะเหมาะสมที่สุดในเวลานั้นได้ ในส่วนของการถอดรหัสจะมีการทำงานที่แตกต่างจากการเข้ารหัสอยู่มากพอสมควร โดยในส่วนของการถอดรหัสข้อมูล จะมีขั้นตอนการทำงานดังภาพประกอบที่ 3-14



ภาพประกอบที่ 3-14 ลำดับการทำงานของ Hybrid algorithm ในส่วนของการถอดรหัส

ภาพประกอบที่ 3-14 เป็นการแสดงลำดับการทำงานของการถอดรหัสแบบ Hybrid โดยในแต่ละขั้นตอนก็จะมีการทำงานที่แตกต่างกันดังนี้

- Initial register เป็นขั้นตอนที่ใช้ในการกำหนดค่าเริ่มต้นของ Register ต่าง ๆ ในระบบ โดยที่ขั้นตอนนี้จะมีการทำงานเหมือนกับขั้นตอน Initial register ในฝั่งของการเข้ารหัสของ Hybrid algorithm คือ ทำการกำหนดค่าใน Register ทุกตัวให้มีค่าเป็น 0 โดย Register ที่อยู่ในระบบนั้นจะเป็นแบบ Array
- Calculate register เป็นขั้นตอนในการนำข้อมูลทั้งหมดที่อยู่ใน Register array มาหาผลรวมของจำนวนบิต เพื่อหาว่าในอดีตที่ผ่านมา อัลกอริทึมใดมีผลลัพธ์ที่ดีที่สุด

- Select decoder เป็นขั้นตอนที่ใช้ในการเลือกอัลกอริทึม ที่จะทำการถอดรหัส โดยใช้ผลลัพธ์จากขั้นตอน Calculate register และในส่วนของการรับข้อมูลเข้ามานั้น จะเป็นหน้าที่ของตัวถอดรหัส
- Wait and get result เป็นขั้นตอนที่ใช้ในการรอรับผลลัพธ์ในการถอดรหัส และทำการส่งผลลัพธ์ที่ได้ออกไป เนื่องจากผลลัพธ์ดังกล่าว จะเป็นผลลัพธ์ที่ได้จากการถอดรหัสที่เสร็จสมบูรณ์แล้ว
- Dispense data เป็นขั้นตอนที่จะนำผลลัพธ์ที่ได้ไปแจกจ่ายให้กับตัวเข้ารหัสของแต่ละอัลกอริทึม เพื่อให้ทราบว่าถ้านำข้อมูลดังกล่าวมาทำการเข้ารหัสด้วยวิธีต่าง ๆ จะได้ผลลัพธ์อย่างไร
- Wait and get result เป็นขั้นตอนการรอรับผลลัพธ์จากการเข้ารหัสของแต่ละอัลกอริทึม เนื่องจากแต่ละอัลกอริทึมใช้เวลาในการทำงานไม่เท่ากัน
- Update register เป็นขั้นตอนการนำผลลัพธ์ของการเข้ารหัส มาเพิ่มเข้าไปใน Register array โดยที่การทำงานในขั้นตอนนี้จะเหมือนกันกับขั้นตอน Update register ในส่วนของการเข้ารหัสของ Hybrid algorithm ซึ่งจะต้องทำการ Shift ข้อมูลตัวที่เก่าที่สุดออกจาก Array แล้วจึงทำการเพิ่มข้อมูลที่ได้จากขั้นตอน Wait and get data เข้าไปในระบบ เพื่อรอการเรียกใช้งานในรอบการทำงานถัดไป

หลังจากทำงานในขั้นตอน Update register เสร็จแล้ว ก็จะวนกลับไปทำงานในขั้นตอน Calculate register แล้วทำงานวนซ้ำเช่นนี้ไปเรื่อย ๆ โดยการทำงานของ Hybrid algorithm จะมีประสิทธิภาพมากหรือไม่นั้น จะขึ้นอยู่กับขนาดของ Register array หรือก็คือจำนวนรอบที่ใช้ในการเก็บข้อมูลย้อนหลัง และยังขึ้นอยู่กับอัลกอริทึมที่อยู่ภายใน ว่าสามารถทำงานครอบคลุมข้อมูลทุก ๆ ลักษณะหรือไม่ ซึ่งงานวิจัยชิ้นนี้ จะทำการทดลองใช้ Hybrid algorithm ด้วยกัน 3 แบบ คือ Hybrid 001, Hybrid 002 และ Hybrid 003 ซึ่งทั้งสามวิธีดังกล่าวจะมีเค้าโครงการทำงานเหมือนกับขั้นตอนที่ได้อธิบายไว้ข้างต้น แต่จะแตกต่างกันในส่วนของการเลือกอัลกอริทึมที่จะนำมาใช้ โดยแต่ละแบบ จะมีข้อแตกต่างในการเลือกใช้อัลกอริทึมดังนี้

- Hybrid 001 นั้นจะเป็นการทำ Hybrid ของ 2 อัลกอริทึม คือ SZAVGLB (เป็นการนำ Single Zero with Average มาใช้งานร่วมกันกับเทคนิค Limited Buffer) และ CoXoH [2] ซึ่งเป็นหนึ่งในงานวิจัยที่ได้ทำการศึกษา และเอามาเป็นตัวอย่าง โดยที่ CoXoH นั้น จะไม่ทำการกำหนดข้อจำกัดของ Register
- Hybrid 002 เป็นการทำ Hybrid กันระหว่าง SZAVGLBS (เป็นการนำ Single Zero with Average มาใช้งานร่วมกับเทคนิค Limited Buffer by Shift) และ CoXoH โดยที่ทางด้านของ CoXoH จะไม่ถูกจำกัดปริมาณการใช้งาน Register
- Hybrid 003 เป็นการทำ Hybrid กัน 3 อัลกอริทึม ได้แก่ Blank, SZAVGLBS และ CoXoH โดยที่ Blank เป็นการส่งข้อมูลต้นฉบับออกไปตรง ๆ ทั้งนี้เพื่อตรวจสอบในกรณีที่อัลกอริทึมทั้งหมดที่มีอยู่ในระบบ ไม่สามารถบีบอัดข้อมูลได้อย่างมีประสิทธิภาพ

3.5 วิเคราะห์ และสรุปผล

การออกแบบวิธีการเข้ารหัส เพื่อนำมาใช้งานกับเครือข่ายเซนเซอร์ไร้สาย จะต้องคำนึงถึงหลักการใช้งานหลายประการ เนื่องจากข้อมูลที่ใช้งานอยู่บนเครือข่ายเซนเซอร์ไร้สายมีหลากหลายประเภท ไม่ว่าจะเป็น ข้อมูลสภาพแวดล้อม ข้อมูลผู้ป่วย ข้อมูลรูปภาพ ฯลฯ ซึ่งการใช้งานที่หลากหลาย ก็จะทำให้เกิดข้อจำกัดต่ออัลกอริทึมที่จะออกแบบขึ้นมา โดยจะต้องส่งผลกระทบต่อระบบให้น้อยที่สุด ทั้งเรื่องระยะเวลาที่ใช้ในการเข้ารหัส ค่า Overhead ที่เกิดจากการเข้ารหัส หรือแม้กระทั่งการใช้ทรัพยากรของระบบ ที่จะต้องให้น้อยที่สุดเท่าที่จะทำได้ ทำให้วิธีเข้ารหัสที่ถูกออกแบบนั้น มีพื้นฐานการทำงานที่ไม่ซับซ้อน มีขั้นตอนและระยะเวลาในการทำงานแต่ละรอบที่ชัดเจน ทั้งส่วนของการเข้ารหัส และส่วนของการถอดรหัส แต่อย่างไรก็ตาม วิธีการเข้ารหัสที่ได้ทำการออกแบบนั้น ยังต้องคงไว้ซึ่งประสิทธิภาพในการบีบอัดที่ดีด้วย

วิธีการเข้ารหัสที่ได้ถูกออกแบบขึ้นมา จะมีด้วยกัน 3 วิธีหลัก ๆ คือ DZ, SZ และ SZAVG ซึ่งทั้งสามวิธีนี้ จะมีเค้าโครงการทำงานที่เหมือนกัน แต่อย่างไรก็ตาม ทั้ง 3 วิธีที่ได้กล่าวมายังไม่สามารถที่จะนำไปใช้งานจริงได้ เนื่องจากปัญหาในเรื่องของการใช้งาน Register ที่ใช้ในการเก็บข้อมูลทางสถิติ ซึ่งจะต้องกำหนดขนาดของ Register ให้ชัดเจน เพื่อให้สามารถนำไปสังเคราะห์เป็นวงจรจริง ๆ ได้ โดยในส่วนนี้ ได้มีการสร้างส่วนที่จะใช้จัดการกับ Register ขึ้นมา ได้แก่ LB และ LBS โดยทั้งสองวิธีมีวิธีการจัดการกับ Register ที่แตกต่างกัน ซึ่งจะส่งผลต่อขนาดของวงจรที่จะทำการสังเคราะห์ แต่อย่างไรก็ตาม การจัดการ Register ด้วย LB หรือ LBS จะทำให้มีผลลัพธ์ที่แตกต่างออกไป โดยสามารถดูผลลัพธ์ของการบีบอัดข้อมูลได้จากบทที่ 5 และจากผลลัพธ์ดังกล่าวจึงได้มีการพัฒนาเพิ่มเติมในส่วนของการเข้ารหัสข้อมูลแบบ Hybrid โดยจะนำวิธีที่ได้ทำการพัฒนามาใช้งานร่วมกันกับ CoXoH ซึ่งคาดว่าจะได้ผลลัพธ์ที่มีประสิทธิภาพยิ่งขึ้น

อย่างไรก็ตาม วิธีการเข้ารหัสที่ได้ออกแบบไว้นั้น ยังคงไว้ซึ่งความสามารถในการใช้งานร่วมกันกับเครือข่ายเซนเซอร์ไร้สาย อีกทั้งยังสามารถนำไปออกแบบเป็นวงจรบน FPGA ได้ และสามารถทำงานได้อย่างถูกต้องตามที่ได้ออกแบบไว้ โดยการออกแบบการใช้งานบน FPGA จะกล่าวไว้ในบทที่ 4

บทที่ 4
การออกแบบฮาร์ดแวร์

ในบทนี้จะกล่าวถึงการนำวิธีการเข้ารหัสข้อมูลที่ได้ออกแบบไว้ มาทำการออกแบบใหม่ลงบน FPGA เพื่อทำการทดสอบความเป็นไปได้ของอัลกอริทึมในการใช้งานจริง และเพื่อวิเคราะห์ปริมาณทรัพยากรที่ถูกใช้งาน โดยจะเลือกเอาเฉพาะ SZAVGLB และ SZAVGLBS มาทำการวิเคราะห์ เนื่องจากวิธีดังกล่าว น่าจะใช้ทรัพยากรของระบบมากที่สุด และมีประสิทธิภาพโดยรวมค่อนข้างดี

4.1 ภาพรวมของระบบ

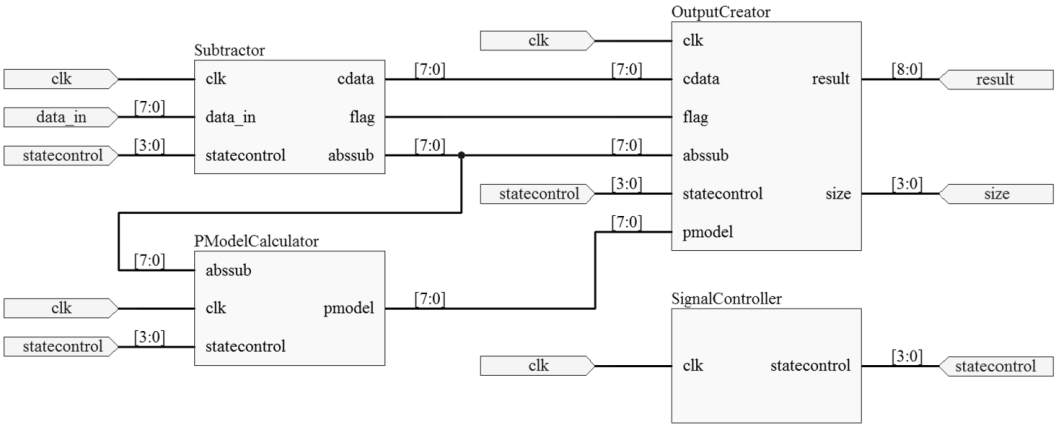
การนำวิธีการ DZ, SZ หรือ SZAVG มาทำการสร้างเป็นฮาร์ดแวร์นั้น สามารถทำได้หลายวิธี แต่ด้วยวัตถุประสงค์ที่ต้องการวิเคราะห์ปริมาณทรัพยากรของระบบ เพื่อให้สามารถนำไปต่อยอดพัฒนาเป็นวงจรรวมได้ในภายหลัง จึงเลือกใช้ FPGA (Field Programmable Gate Array) เนื่องจากการเขียนโปรแกรมเพื่อใช้งาน FPGA สามารถสะท้อนให้เห็นถึงปริมาณการใช้งานทรัพยากรฮาร์ดแวร์ได้ ในขณะที่การใช้งานไมโครคอนโทรลเลอร์ก็สามารถทำได้ แต่จะไม่สามารถต่อยอดไปสู่การพัฒนาระบบให้เป็นวงจรรวม

การนำ SZAVGLB และ SZAVGLBS มาทำการสังเคราะห์วงจรลงบน FPGA นั้น จะต้องทำการเรียงลำดับการทำงานของอัลกอริทึมใหม่ เนื่องจากการทำงานบนฮาร์ดแวร์ มีความแตกต่างจากการทำงานด้วยซอฟต์แวร์พอสมควร และยังต้องมีการแยกส่วนของภาคเข้ารหัสกับภาคถอดรหัส ซึ่งทำงานแยกออกจากกันอย่างชัดเจน แต่เค้าโครงในการเข้ารหัสของทั้ง SZAVGLB และ SZAVGLBS จะยังคงเหมือนกัน รวมถึงเค้าโครงในการถอดรหัสด้วย ดังนั้นจะทำให้โครงร่างของระบบถูกแบ่งออกเป็น 2 ส่วนด้วยกัน คือ ส่วนของตัวเข้ารหัส (Encoder part) และส่วนของตัวถอดรหัส (Decoder part) ทั้งนี้ ระบบที่ได้ออกแบบไว้ จะไม่มีส่วนของการจัดการกับพอร์ทอนุกรม (Serial port) หรือช่องทางการสื่อสารใด ๆ จะมีเฉพาะส่วนของวงจรเข้ารหัส และส่วนของวงจรถอดรหัส โดยระบบที่จำลองขึ้นมาจะอ้างอิงการทำงานกับข้อมูลขนาด 8 บิต เท่านั้น

4.1.1 Encoder part

การสร้างส่วนของฮาร์ดแวร์สำหรับเข้ารหัสขึ้นมา จะต้องทำการเรียงลำดับขั้นตอนการทำงานขึ้นมาใหม่ และต้องทำการควบรวมการทำงานบางส่วนไว้ในจุดเดียวกัน โดยที่ลำดับของการทำงานเดิมของอัลกอริทึมที่ได้ออกแบบไว้นั้น จะมีขั้นตอนการทำงานทั้งหมด 7 ขั้นตอน คือ Initial register, Wait and get data, Absolute subtract, Predict data model, Check model, Replace and select และ Update register จากขั้นตอนเหล่านี้ จะต้องทำการควบรวมส่วนของทรัพยากรที่ถูกใช้งานร่วมกัน ไว้ที่ส่วนเดียวกัน เช่น ส่วนของ Register ที่จะถูกใช้งานในขั้นตอน Predict data model และขั้นตอน Update register จะต้องถูกนำมาควบรวมไว้ด้วยกัน และยังมีส่วนที่สามารถตัดออกไปจากระบบได้ คือส่วนของ Initial register เนื่องจากส่วนดังกล่าว จะต้องถูกกระจายไปยัง

ส่วนต่าง ๆ ของฮาร์ดแวร์ ทำให้ผลลัพธ์สุดท้ายมีส่วนของฮาร์ดแวร์อยู่ 3 ส่วนหลัก ๆ และมีอีก 1 ส่วน ที่ใช้ควบคุมลำดับการทำงาน ดังภาพประกอบที่ 4-1



ภาพประกอบที่ 4-1 โครงสร้างของวงจรที่ใช้สำหรับการเข้ารหัส

จากภาพประกอบที่ 4-1 จะเห็นได้ว่ามีวงจรใหญ่อยู่ทั้งหมด 4 ส่วนด้วยกัน คือ Subtractor, PModelCalculator, OutputCreator และ SignalController ซึ่งทั้ง 3 ส่วนแรก จะถูกควบคุมลำดับการทำงานจาก SignalController โดยจะทำงานตามสัญญาณนาฬิกา แต่ละวงจรจะมีการทำงานดังนี้

1. Subtractor เกิดจากการทำงานในส่วนของ Absolute subtract โดยในวงจรนี้จะทำหน้าที่ในการหาค่าสัมบูรณ์ของผลต่างระหว่างข้อมูลตัวปัจจุบัน กับข้อมูลตัวก่อนหน้า (abssub : เป็นเอาท์พุต) และสัญญาณธง (flag : เป็นเอาต์พุต) ที่ใช้ในการบอกว่า ข้อมูลตัวปัจจุบันกับข้อมูลตัวก่อนหน้า ตัวใดมีค่ามากกว่า โดยวงจรนี้จะประกอบไปด้วย 3 ช่องสัญญาณอินพุต และ 3 ช่องสัญญาณเอาต์พุต ดังนี้
   - clk เป็นสัญญาณอินพุตที่เป็นสัญญาณนาฬิกา ใช้สำหรับการควบคุมจังหวะการทำงานของวงจรนี้
   - data_in เป็นสัญญาณอินพุต มีขนาด 8 บิต (สามารถปรับเปลี่ยนขนาดได้ขึ้นอยู่กับข้อมูลที่ต้องการนำไปทำการเข้ารหัส ว่ามีขนาดเท่าใด แต่ทั้งนี้การเพิ่ม หรือลดขนาดของข้อมูลต้นฉบับ จะสัมพันธ์กับหลาย ๆ ส่วนในระบบ) สัญญาณ data_in เป็นข้อมูลต้นฉบับ จะถูกอ่านเข้ามาเพื่อเข้ารหัสในลำดับถัดไป
   - statecontrol เป็นสัญญาณอินพุตของวงจร ใช้ในการควบคุมการทำงานของวงจรนี้ ร่วมกันกับวงจรอื่น ๆ โดยสัญญาณจะถูกสร้างขึ้นมาจากวงจร SignalController มีขนาด 4 บิต

- cdata เป็นเอาต์พุตของวงจร ซึ่งเป็นค่าของ data_in ในรอบการทำงานนั้น ซึ่งจะต้องมีขนาดเท่ากันกับ data_in ที่เป็น Input ของวงจร
- flag เป็นเอาต์พุต ใช้เพื่อแสดงให้ทราบว่าระหว่างข้อมูลก่อนหน้ากับข้อมูลปัจจุบัน ข้อมูลตัวใดมีค่ามากกว่า โดยหากข้อมูลก่อนหน้ามีค่ามากกว่า หรือเท่ากันกับข้อมูลปัจจุบัน จะทำให้ flag มีค่าเป็น 0 แต่ถ้าหากข้อมูลปัจจุบันมีค่ามากกว่า จะทำให้ flag มีค่าเป็น 1
- abssub เป็นเอาต์พุตของวงจรมีขนาดเท่ากันกับ data_in เป็นผลลัพธ์ของการหาค่าผลต่างระหว่างข้อมูลปัจจุบัน กับข้อมูลตัวก่อนหน้า

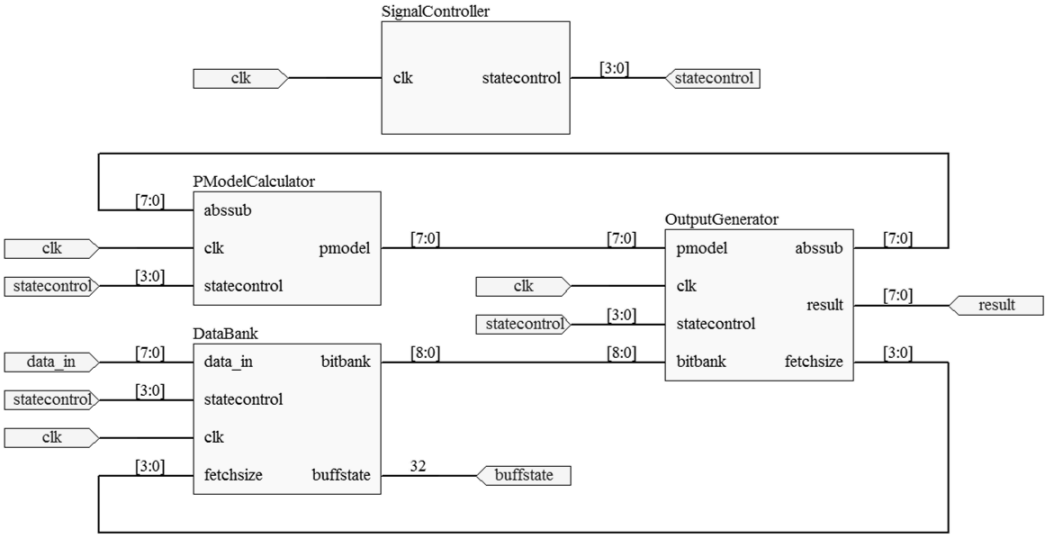
2. PModelCalculator เกิดจากการทำงานในขั้นตอน Predict data model และขั้นตอน Update register ผสมเข้าด้วยกัน โดยในขั้นตอนนี้ จะทำหน้าที่ในการคำนวณหาโมเดลของ abssub ที่ได้จากขั้นตอน Subtractor แล้วส่งออกไปเป็นผลลัพธ์ (pmodel) เพื่อใช้สำหรับตรวจสอบกับค่าสัมบูรณ์ของผลต่างในขั้นตอนถัดไป และอีกหน้าที่หนึ่ง คือการนำ abssub มาคำนวณเพื่อเพิ่มค่าลงใน Register วงจรนี้มี 3 อินพุต และมี 1 เอาต์พุต ดังนี้
   - abssub เป็นอินพุตของวงจร ขนาด 8 บิต ถูกนำเข้ามาในวงจรนี้ เพื่อใช้ในการเพิ่มค่าของ Register หลังจากคำนวณโมเดลของข้อมูลแล้ว
   - clk เป็นสัญญาณนาฬิกา ใช้สำหรับการควบคุมจังหวะการทำงานของวงจร ซึ่งจะต้องมาจากแหล่งเดียวกัน
   - statecontrol เป็นสัญญาณอินพุตของวงจร มีขนาด 4 บิต ใช้สำหรับควบคุมลำดับกับการทำงานของแต่ละขั้นตอนภายในวงจร ถูกสร้างขึ้นโดยวงจร SignalController
   - pmodel เป็นสัญญาณเอาต์พุตขนาด 8 บิต เป็นค่าของโมเดลที่ได้จากการคำนวณค่าต่าง ๆ ใน Register และจะถูกส่งไปยัง OutputCreator
3. OutputCreator เกิดจากการรวมการทำงานในขั้นตอน Check model และ Replace and select ไว้ด้วยกัน โดยวงจรนี้ จะทำหน้าที่ในการตรวจสอบว่าค่าจาก abssub ตรงกับโมเดลหรือไม่ แล้วทำการเลือกรูปแบบของผลลัพธ์ (result, size) ซึ่งวงจรนี้จะประกอบไปด้วยอินพุต 6 ช่องสัญญาณ และเอาต์พุต 2 ช่องสัญญาณ ดังนี้
   - clk เป็นสัญญาณนาฬิกา ใช้สำหรับกำหนดจังหวะการทำงานของวงจร
   - cdata เป็นสัญญาณอินพุต มีขนาด 8 บิต ใช้สำหรับรับค่าของข้อมูลปัจจุบัน เพื่อใช้ในการประกอบเป็นผลลัพธ์
   - flag เป็นสัญญาณอินพุตใช้บอกให้ทราบว่าข้อมูลตัวก่อนหน้ามีค่ามากกว่าข้อมูลตัวปัจจุบันหรือไม่ (เป็น 0 เมื่อข้อมูลก่อนหน้ามีค่ามากกว่าหรือเท่ากัน)

- abssub เป็นสัญญาณอินพุตขนาด 8 บิต ซึ่งเป็นค่าสัมบูรณ์ของผลต่างระหว่างข้อมูลตัวก่อนหน้า กับข้อมูลตัวปัจจุบัน ใช้ในการประกอบเป็นส่วนหนึ่งของผลลัพธ์
- statecontrol เป็นสัญญาณอินพุตขนาด 4 บิต ใช้ในการควบคุมขั้นตอนการทำงานของวงจร ถูกสร้างขึ้นจากวงจร SignalController
- pmodel เป็นสัญญาณอินพุต มีขนาด 8 บิต เป็นสัญญาณที่ถูกส่งมาจาก PModelCalculator เพื่อบอกว่าโมเดลที่ได้นั้น คือโมเดลใด
- result เป็นสัญญาณเอาต์พุตขนาด 9 บิต ซึ่งเป็นผลลัพธ์จากการบีบอัดข้อมูล
- size เป็นสัญญาณเอาต์พุตขนาด 4 บิต เป็นสัญญาณที่บอกว่าขนาดของผลลัพธ์ มีขนาดเท่าใด

4. SignalController เป็นวงจรที่ใช้ในการสร้างสัญญาณ statecontrol เพื่อควบคุมวงจรอื่น ๆ จะมีการทำงานเหมือนกันกับวงจร Counter ทั่ว ๆ ไป โดยวงจรนี้จะมีอินพุตเป็น clk และมีเอาต์พุตเป็น statecontrol

4.1.2 Decoder part

เช่นเดียวกับการสร้างวงจรเข้ารหัส ในการสร้างวงจรสำหรับการถอดรหัสนั้นก็จำเป็นต้องมีการออกแบบใหม่ในบางส่วน เพื่อให้สามารถนำมาสร้างเป็นวงจรได้ โดยจะต้องทำการควบรวมบางขั้นตอน และเพิ่มการทำงานในบางขั้นตอน จากเดิมนั้นการถอดรหัสข้อมูลจะมีขั้นตอนย่อย ๆ ทั้งหมด 5 ขั้นตอน คือ Initial register, Predict data model, Fetch data, Add data และ Update register ซึ่งทั้ง 5 ขั้นตอนดังกล่าว จะถูกแปลงให้เป็นวงจร 4 ดังภาพประกอบที่ 4-2



ภาพประกอบที่ 4-2 โครงสร้างของวงจรที่ใช้สำหรับการถอดรหัส

จากภาพประกอบที่ 4-2 วงจรถอดรหัสจะประกอบด้วยวงจรย่อยที่ทำงานร่วมกัน โดยจะประกอบไปด้วยวงจร 3 วงจร ได้แก่ PModelCalculator, DataBank และ OutputGenerator โดย PModelCalculator ทำหน้าที่ในการคำนวณหาโมเดลของข้อมูล แล้วส่งไปให้ OutputGenerator ส่วน DataBank ทำหน้าที่ในการช่วยดึงข้อมูลเข้ามาในระบบ และ OutputGenerator จะทำหน้าที่นำข้อมูลที่เข้ามาในระบบ ประมวลผลร่วมกันกับโมเดลแล้วสร้างผลลัพธ์ออกมา พร้อมทั้งคำนวณหาค่าสัมบูรณ์ของผลต่าง ส่งไปให้กับ PModelCalculator เพื่อใช้ในการเพิ่มค่าทางสถิติของ Register และยังส่งปริมาณข้อมูลที่ดึงออกมาใช้กลับไปให้ DataBank เพื่อใช้วิเคราะห์ในการรับข้อมูลเข้า ซึ่งทั้ง 3 วงจรข้างต้น จะถูกควบคุมลำดับการทำงานด้วยสัญญาณ statecontrol ที่ถูกสร้างขึ้นโดย SignalController

กระบวนการทำงานที่ได้กล่าวมานั้น เป็นเพียงภาพรวมของการทำงาน แต่ละวงจรก็จะมีการทำงานอีกหลายขั้นตอน ซึ่งจะขออธิบายแยกเป็นวงจรต่าง ๆ ดังนี้

1. PModelCalculator ทำหน้าที่ในการคำนวณว่าค่าสัมบูรณ์ของผลต่างของข้อมูลตัวปัจจุบัน กับตัวที่กำลังจะเข้ามาในระบบ น่าจะมีค่าเป็นเท่าใด ทำให้ได้ออกมาเป็น pmodel ส่งออกไปให้กับ OutputGenerator โดยวงจรนี้ จะมีการทำงาน 4 ขั้นตอน คือ หาโมเดลเบื้องต้น, แปลงค่าโมเดล, จัดการ Register และเพิ่มค่า Register แต่ละขั้นตอนจะทำงานเสร็จภายใน 1 รอบ สัญญาณนาฬิกาวงจรนี้มีการทำงานที่เหมือนกันกับวงจร PModelCalculator ในภาคของการเข้ารหัส ทั้งวิธีการทำงาน อินพุต และเอาต์พุต แต่จะมีความแตกต่างกันในลำดับการทำงาน เมื่ออ้างอิงกับสัญญาณ statecontrol
2. DataBank ทำหน้าที่ในการช่วยรับค่าเข้ามาในระบบ ซึ่งในการใช้งานจริงนั้นระบบจะได้รับข้อมูลครั้งละ 8 บิต แต่จะเรียกใช้ครั้งละ 3 - 9 บิต จึงจำเป็นต้องมีส่วนในการช่วยจัดการกับข้อมูลที่รับเข้ามาในระบบ วงจรนี้ จะทำการดึงข้อมูลเข้ามาสำรองในระบบ เพื่อให้ OutputGenerator เลือกดึงเอาไปใช้งาน หลังจากนั้น OutputGenerator จะส่งสัญญาณกลับมาบอกว่า ดึงข้อมูลไปใช้ทั้งหมดกี่บิต ในรอบนั้น ๆ ทำให้วงจร DataBank สามารถพิจารณาได้ว่าข้อมูลที่เหลืออยู่ เพียงพอต่อการทำงานในรอบต่อไปหรือไม่ ถ้าหากไม่พอจะทำการดึงข้อมูลเข้ามาเพิ่ม วงจรนี้ประกอบไปด้วยอินพุต 4 ตัว และเอาต์พุต 2 ตัว ดังนี้
   - clk เป็นสัญญาณนาฬิกาที่รับเข้ามา เพื่อเป็นจังหวะในการทำงานของวงจร
   - statecontrol เป็นสัญญาณอินพุตขนาด 4 บิต ใช้กำกับลำดับการทำงานของส่วนต่าง ๆ ภายในวงจร ถูกสร้างจากวงจร SignalController
   - data_in เป็นอินพุตขนาด 8 บิต เป็นข้อมูลเข้ารหัส ที่ถูกส่งเข้ามาในระบบจะถูกสำรองเอาไว้ ก่อนที่จะส่งต่อไปให้กับวงจร OutputCalculator
   - fetchsize เป็นอินพุตขนาด 4 บิต ถูกส่งมาจาก OutputCalculator เป็นค่าที่บอกถึงปริมาณข้อมูลที่ถูกดึงออกไปจาก bitbank ใช้เพื่อคำนวณปริมาณข้อมูลที่ทำการสำรองเอาไว้

- bitbank เป็นเอาต์พุตขนาด 9 บิต เป็นข้อมูลที่ถูกเข้ารหัส ใช้ในการป้อนให้กับวงจร OutputCalculator เพื่อใช้ในการถอดรหัส
- buffstate เป็นสัญญาณเอาต์พุตขนาด 1 บิต ใช้บ่งบอกสถานะการสำรองข้อมูล ว่ามีที่ว่างอยู่หรือไม่ เพื่อให้ระบบที่ทำการเชื่อมต่อกับตัวถอดรหัสทำการป้อนข้อมูลใหม่เข้ามา

3. OutputCreator ทำหน้าที่ในการนำโมเดลมาคำนวณเพื่อถอดรหัสข้อมูล หรือกล่าวได้ว่า ทำหน้าที่ในการหาผลลัพธ์ของการถอดรหัส ภายในวงจรนี้ จะมีการทำงานเป็นขั้นตอน ทั้งหมด 3 ขั้นตอนด้วยกัน ได้แก่ การคัดแยกกรณี, การสร้างผลลัพธ์ และการหาค่าสัมบูรณ์ของผลต่าง โดยที่แต่ละขั้นตอน จะใช้เวลาในการทำงาน 1 รอบสัญญาณนาฬิกา วงจรนี้จะประกอบด้วย 4 อินพุต และ 3 เอาต์พุต แต่ละช่องสัญญาณจะมีรายละเอียดดังนี้
   - clk เป็นสัญญาณนาฬิกา ใช้กำกับจังหวะในการทำงานในขั้นตอนต่าง ๆ
   - statecontrol เป็นสัญญาณอินพุตขนาด 4 บิต ใช้ควบคุมลำดับการทำงานของแต่ละขั้นตอนที่อยู่ภายในวงจร ถูกสร้างจากวงจร SignalController
   - bitbank เป็นสัญญาณอินพุตขนาด 9 บิต ถูกเข้ารหัส และถูกจัดการเบื้องต้นจากวงจร DataBank ทำให้มีขนาด 9 บิต ซึ่งเป็นขนาดใหญ่ที่สุดที่สามารถเกิดขึ้นได้จากการเข้ารหัส รับเข้ามาเพื่อใช้ในการถอดรหัส
   - pmodel เป็นสัญญาณอินพุตขนาด 8 บิต เป็นโมเดลของข้อมูลที่ได้ทำการคำนวณเอาไว้โดยวงจร PModelCalculator นำมาคำนวณร่วมกันกับข้อมูลที่ได้รับจาก bitbank เพื่อทำการถอดรหัส
   - result เป็นสัญญาณเอาต์พุตขนาด 8 บิต เป็นข้อมูลที่ถูกถอดรหัสเสร็จแล้ว ซึ่งเป็นผลลัพธ์ของระบบ
   - abssub เป็นสัญญาณเอาต์พุต มีขนาด 8 บิต ซึ่งเป็นค่าสัมบูรณ์ของผลต่างของผลลัพธ์ตัวก่อนหน้า กับผลลัพธ์ตัวปัจจุบัน เป็นข้อมูลที่จะถูกส่งต่อไปให้กับวงจร PModelCalculator เพื่อใช้ในการเพิ่มค่าให้กับ Register หรือก็คือ การปรับปรุงข้อมูลทางสถิติให้เป็นปัจจุบัน เพื่อให้สามารถใช้คำนวณหาโมเดลในรอบถัดไปได้
   - fetchsize เป็นสัญญาณเอาต์พุตขนาด 4 บิต ซึ่งเป็นจำนวนของบิตจากสัญญาณ bitbank ที่ถูกดึงมาใช้ในการคำนวณ สัญญาณนี้จะถูกส่งให้กับวงจร DataBank เพื่อทำการบรรจุข้อมูลใหม่เข้ามา ตามจำนวนของข้อมูลที่ถูกดึงออกไป

4. SignalController วงจรนี้ทำหน้าที่ในการสร้างสัญญาณ statecontrol เพื่อควบคุมลำดับการทำงานของวงจรอื่น ๆ โดยวงจร SignalController ฝั่งของตัวเข้ารหัส กับตัวถอดรหัสนั้น จะมีโครงสร้างเหมือนกัน คือทำหน้าที่เป็นวงจรนับขนาด 4 บิต ที่มี statecontrol เป็นผลลัพธ์ แต่ฝั่งของตัวเข้ารหัส จะนับในจำนวนที่มากกว่า เนื่องจากมีขั้นตอนการทำงานที่มากกว่า

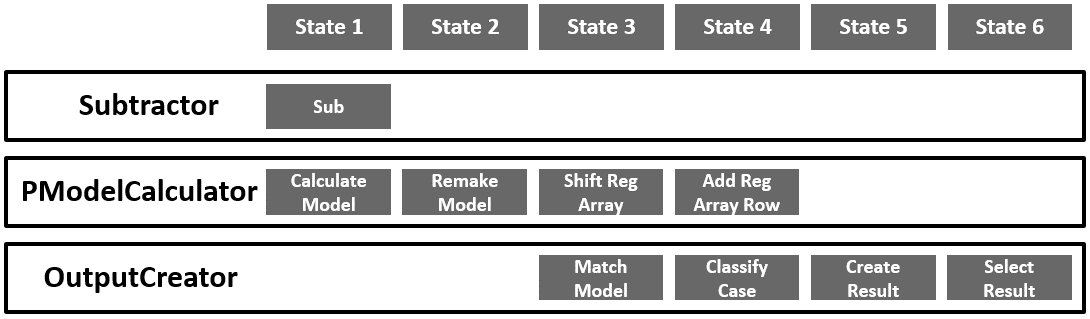
4.2 SZAVGLB

โครงสร้างวงจรของ SZAVGLB จะมีโครงสร้างเหมือนกันกับโครงสร้างในหัวข้อ 4.1.1 และ 4.1.2 ดังภาพประกอบที่ 4-1 และ ภาพประกอบที่ 4-2

โครงสร้างภายนอกจะมีการทำงานที่เหมือนกันทุกประการ แต่ในหัวข้อนี้ จะอธิบายถึงรายละเอียดในการทำงานของตัวเข้ารหัส และตัวถอดรหัส

4.2.1 Encoder circuit

โครงสร้างของตัวเข้ารหัสของ SZAVGLB จะมีโครงสร้างดังภาพประกอบที่ 4-1 โดยภายในแต่ละวงจรจะมีลำดับการทำงานแยกย่อยลงไปได้อีก โดยการทำงานแต่ละขั้นตอนจะอ้างอิงกับสัญญาณ statecontrol คือ แต่ละขั้นตอนจะทำงานที่ statecontrol ที่มีค่าแตกต่างกัน อาจกล่าวได้ว่า statecontrol เป็นสัญญาณที่ใช้ในการควบคุมลำดับการทำงานของระบบ สำหรับ SZAVGLB จะมีลำดับการทำงานดังภาพประกอบที่ 4-3



ภาพประกอบที่ 4-3 ลำดับการทำงานในการเข้ารหัสของ SZAVGLB

จากภาพประกอบที่ 4-3 เป็นการแสดงการทำงานของแต่ละขั้นตอนในวงจรต่าง ๆ จะเห็นได้ว่าแต่ละขั้นตอนจะมีการทำงานที่ statecontrol ที่แตกต่างกัน แต่ในบางขั้นตอนสามารถทำงานใน statecontrol เดียวกัน ทำให้การทำงานรวดเร็วยิ่งขึ้น ซึ่งแต่ละขั้นตอนที่แสดงไว้ในภาพประกอบที่ 4-3 จะมีการทำงานดังนี้

- Sub เป็นขั้นตอนของ Subtractor ที่ทำงานในจังหวะที่ statecontrol มีค่าเป็น 1 ซึ่งจะทำหน้าที่ในการหาค่าสัมบูรณ์ของผลต่างระหว่างข้อมูลอินพุตที่เข้ามาทาง data_in ในรอบปัจจุบัน กับข้อมูลอินพุตในรอบก่อนหน้านี้ ซึ่งจะให้ผลลัพธ์ออกมา 3 สัญญาณ ได้แก่ cdata คือค่าที่อ่านได้จาก data_in ในรอบปัจจุบัน, abssub คือค่าสัมบูรณ์ของผลต่างที่ได้กล่าวไว้ข้างต้น และ flag คือสัญญาณที่ใช้บอกให้ทราบว่าข้อมูลอินพุตที่เข้ามาทาง data_in ในรอบปัจจุบันมีค่ามากกว่าในรอบก่อนหน้านี้หรือไม่ ถ้ามีค่ามากกว่า จะทำให้ flag มีค่าเป็น 1 แต่ถ้ามีค่าน้อยกว่าหรือเท่ากัน จะมีค่าเป็น 0
- Calculate Model เป็นขั้นตอนที่จะนำข้อมูลทางสถิติต่าง ๆ ที่อยู่ใน Register มาทำการคำนวณเพื่อหาค่าเฉลี่ย (สูตรการคำนวณอยู่ในบทที่ 3 หัวข้อที่ 3.2.5)

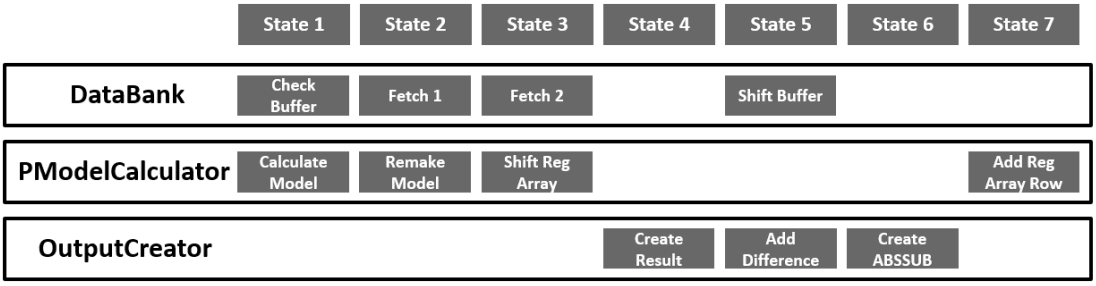
- Remake Model จะทำการแปลงค่าโมเดลที่ได้จากขั้นตอน Calculate Model ซึ่งเป็นค่าเฉลี่ย ให้มีลักษณะพร้อมใช้งาน เช่น ถ้าหากหาค่าเฉลี่ยได้เป็น 1.64 จะได้โมเดลเป็น 0x7F หรือถ้าหากค่าเฉลี่ยนเป็น 5.11 จะได้โมเดลเป็น 0x07 ซึ่งในขั้นตอนนี้จะได้ผลลัพธ์เป็นค่า pmodel ส่งไปให้กับ OutputCreator
- Shift Reg Array ในขั้นตอนนี้ จะทำการเลื่อน Array ของ Register ขึ้นไป 1 ช่อง เพื่อเตรียมพร้อม Register Array ให้สามารถทำการเพิ่มค่าเข้าไปได้
- Add Reg Array Row เป็นขั้นตอนต่อเนื่องจากขั้นตอน Shift Reg Array โดยจะนำ abssub ซึ่งเป็นอินพุตของวงจร มาวิเคราะห์ ว่าตรงกับโมเดลแบบใด แล้วทำการเพิ่มค่า Register ในช่องโมเดลนั้นขึ้น 1
- Match Model ขั้นตอนนี้จะทำการนำอินพุต 2 ตัว คือ abssub และ pmodel มาทำการเปรียบเทียบว่า abssub นั้น ถูกจำแนกอยู่ในประเภทเดียวกันกับ pmodel หรือไม่ ผลการเปรียบเทียบจะเก็บเอาไว้ใน Register ขนาด 2 บิต ถ้าหาก abssub ตรงตามแบบของ pmodel จะมีค่าเป็น 00 แต่ถ้าไม่ใช่ จะมีค่า 01 (ในวิธีการเปรียบเทียบนั้น สามารถเปรียบเทียบโดยการตรวจสอบว่า abssub มีค่ามากกว่า pmodel หรือไม่ โดยถ้าหากมีค่ามากกว่า จะหมายความว่า abssub ไม่ตรงตาม pmodel ที่ได้)
- Classify Case ในขั้นตอนนี้จะทำการตรวจสอบ pmodel ว่าเป็น 0Z, 1Z หรือ 2Z หรือไม่ ถ้าใช่ ก็จะทำการเปลี่ยนแปลงผลการเปรียบเทียบในขั้นตอน Match Model ให้เป็น 11 แต่ถ้าไม่ใช่ ก็จะไม่ทำการเปลี่ยนแปลง
- Create Result ขั้นตอนนี้จะทำการสร้างผลลัพธ์ขึ้นมา 3 แบบ คือ แบบที่ 1 ผลต่างเป็นไปตามโมเดล คือ {0,flag,only significant bit of abssub}, แบบที่ 2 ไม่เป็นไปตามโมเดล คือ {1,cdata} และ แบบที่ 3 ไม่จำเป็นต้องทำการเข้ารหัส เนื่องจากการบีบอัดไม่ทำให้จำนวนบิตลดลง คือ {cdata} (cdata หมายถึง ข้อมูลชุดปัจจุบัน เป็นค่าอินพุตของวงจร OutputCreator)
- SelectResult ใช้ผลการเปรียบเทียบในขั้นตอน Match Model ซึ่งอาจถูกเขียนทับในขั้นตอน Classify Case มาทำการเลือกรูปแบบของผลลัพธ์ที่ได้จากขั้นตอน Create Result โดยถ้าหากผลการเปรียบเทียบเป็น 00 จะเลือกผลลัพธ์แบบที่ 1 ถ้าผลการเปรียบเทียบเป็น 01 จะได้ผลลัพธ์แบบที่ 2 และถ้าผลการเปรียบเทียบเป็น 11 จะได้ผลลัพธ์แบบที่ 3

จะเห็นได้ว่าการทำงานของตัวเข้ารหัสของ SZAVGLB จะใช้เวลาในการทำงาน 6 รอบ ของสัญญาณนาฬิกา ซึ่งจะใช้เวลาในการเข้ารหัสข้อมูลเท่ากันในทุก ๆ รอบ ทำให้สามารถคำนวณระยะเวลาที่ใช้ในการเข้ารหัสได้อย่างชัดเจน โดยถ้าหากอ้างอิงกับ FPGA Spartan 6 LX9 Microboard ซึ่งใช้ความถี่สัญญาณนาฬิกาที่ 66.7 MHz จะสามารเข้ารหัสข้อมูลได้ประมาณ 11.1 ล้านชุด ต่อวินาที ทั้งนี้เนื่องจากมีการใช้งาน DSP ของ Spartan 6 LX9 จะทำให้การคำนวณแบบการคูณ และการหาร สามารถทำได้ในเวลาที่เร็วกว่าปกติ ทำให้การนำไปสังเคราะห์เป็นวงจรรวมจะใช้เวลาในการทำงานที่มากกว่า (DSP ใช้สัญญาณนาฬิกาจากคนละแหล่งกันกับส่วนอื่น ๆ ของ

วงจร ซึ่งมีความเร็วของสัญญาณนาฬิกามากกว่า จึงสามารถทำงานเสร็จได้ภายใน 1 สัญญาณนาฬิกาของระบบ)

4.2.2 Decoder circuit

โครงสร้างของตัวถอดรหัสของ SZAVGLB จะมีโครงสร้างดังภาพประกอบที่ 4-2 โดยภายในแต่ละวงจร จะมีลำดับการทำงานแยกย่อยลงไปได้อีก แต่ละขั้นตอนจะทำงานตามสัญญาณ statecontrol เช่นเดียวกันกับส่วนของการเข้ารหัส โดยจะมีขั้นตอนในการทำงานดังภาพประกอบที่ 4-4



ภาพประกอบที่ 4-4 ลำดับการทำงานในการถอดรหัสของ SZAVGLB

จากภาพประกอบที่ 4-4 จะเห็นได้ว่าแต่ละขั้นตอน จะมีการทำงานใน statecontrol ที่แตกต่างกัน บางขั้นตอนสามารถทำงานพร้อมกันได้ ในส่วนของ PModelCalculator จะมีการทำงานเหมือนกันกับ PModelCalculator ของส่วนเข้ารหัสทุกประการ แตกต่างกันเฉพาะช่วงที่ถูกเรียกใช้งาน แต่อย่างไรก็ตาม จะอธิบายขั้นตอนการทำงานทั้งหมดเพื่อให้มีความต่อเนื่องของขั้นตอน โดยแต่ละขั้นตอนจะมีรายละเอียดในการทำงานดังนี้

- Check Buffer เป็นขั้นตอนในการตรวจสอบว่า Buffer ของข้อมูลว่างพอที่จะทำการดึงข้อมูลขนาด 8 บิตเข้ามาหรือไม่ ถ้าพอ จะทำการเซ็ตค่าของ buffstate ซึ่งเป็นเอาต์พุตของวงจรให้มีค่าเป็น 1 เพื่อให้ระบบที่ทำงานร่วมกันทราบและส่งข้อมูลใหม่มาให้
- Fetch 1 เป็นขั้นตอนในการดึงข้อมูลเข้ามาสำรองไว้ใน Buffer ผ่านทางอินพุตที่ชื่อว่า data_in ซึ่งต้องพิจารณาจาก buffstate ด้วยว่า มีค่าเป็น 1 หรือไม่ (ทำงานเมื่อ buffstate มีค่าเป็น 1 เท่านั้น) นอกจากการสำรองข้อมูลแล้ว ขั้นตอนนี้ จะประเมินด้วยว่า ถ้านำข้อมูลเข้ามาไว้ใน Buffer แล้ว จะมีพื้นที่มากพอที่สำรองข้อมูลอีกชุดหรือไม่ ถ้ามีเหลือมากพอ จะทำการเซ็ตค่า buffstate ให้เป็น 1 แต่ถ้าหากเหลือพื้นที่ไม่มากพอ จะทำการเซ็ตค่า buffstate ให้เป็น 0
- Fetch 2 เป็นการทำงานต่อเนื่องจากขั้นตอน Fetch 1 คือจะทำการนำข้อมูลเข้ามาสำรองไว้ในระบบเหมือนกัน โดยจะทำงานเมื่อ buffstate มีค่าเป็น 1 เท่านั้น และในขั้นตอนนี้จะทำการเซ็ตค่าของ buffstate ให้เป็น 0 ด้วย

- Shift Buffer ขั้นตอนนี้จะทำงานหลังจากที่ OutputCreator ทำการดึงข้อมูลไปใช้แล้ว โดย OutputCreator จะส่ง fetchsize กลับมาบอกว่า ทำการดึงข้อมูลออกไปกี่ตัว ขั้นตอนนี้ก็จะทำการเลื่อนชุดข้อมูลใหม่เข้าไปแทนที่ (วงจร DataBank นั้น จะมีสัญญาณ bitbank ซึ่งเชื่อมต่อโดยตรงกับ 9 บิต สุดท้ายของ Buffer โดยที่ Buffer จะต้องมีขนาด 17 บิต เป็นอย่างน้อย)
- Calculate Model เป็นขั้นตอนในการหาค่าผลรวมของ Register ในแนวตั้งของ Register Array ซึ่งเป็นข้อมูลทางสถิติของข้อมูลในอดีต โดยจะได้มาทั้งหมด 9 ค่าด้วยกัน ซึ่งเป็นจำนวนของโมเดลที่เคยเกิดขึ้นในระบบ ในรูปแบบนั้น ๆ (โมเดลต่าง ๆ ได้แก่ 0Z, 1Z, 2Z, … , 8Z) แล้วนำมาทำการคำนวณหาค่าเฉลี่ยตามสูตรการคำนวณหาโมเดลของ SZAVG (สูตรดังกล่าวอยู่ในบทที่ 3 หัวข้อที่ 3.2.5)
- Remake Model จะนำค่าเฉลี่ยในขั้นตอน Calculate Model มาแปลงเป็นโมเดลในรูปแบบที่พร้อมใช้งาน เช่น ถ้าหากค่าเฉลี่ยของข้อมูลเป็น 2.26 จะได้โมเดลเป็น 0x3F หรือถ้าหากค่าเฉลี่ยเป็น 6.98 จะได้โมเดลเป็น 0x03 ซึ่งผลลัพธ์ดังกล่าว จะถูกส่งออกไปทางช่องสัญญาณ pmodel
- Shift Reg Array เป็นขั้นตอนในการเตรียม Array ของ Register ให้พร้อมสำหรับทำการเพิ่มค่าในขั้นตอนถัดไป ซึ่งจะทำการเลื่อน Array ของ Register ขึ้น 1 แถว โดยแถวบนสุดจะถูกลบออก และแถวล่างสุดจะใส่ค่า 0 ไว้ก่อน
- Add Reg Array Row ในขั้นตอนนี้ จะนำค่าที่อ่านได้จากสัญญาณ abssub มาทำการวิเคราะห์ ว่าตรงกับรูปแบบโมเดลแบบใด แล้วทำการเพิ่มค่าขึ้น 1 เข้าไปยัง Register ใน Array แถวล่างสุด ที่เก็บค่าความถี่รูปแบบโมเดลนั้นไว้
- Create Result ในขั้นตอนนี้ จะทำการสร้างผลลัพธ์จากการถอดรหัสขึ้นมา โดยอาศัยการคำนวณร่วมกันของ pmodel และ bitbank โดยจะแบ่งออกเป็น 3 กรณีใหญ่ ๆ คือ กรณีที่ 1 เมื่อวิเคราะห์ pmodel แล้ว พบว่าเป็นรูปแบบ 0Z, 1Z หรือ 2Z ก็จะสามารถนำค่าจาก bitbank 8 บิตบน มาเป็นผลลัพธ์ได้ทันที กรณีที่ 2 ถ้าหากบิตบนสุดของ bitbank มีค่าเป็น 1 ให้นำบิตที่เหลือมาเป็นผลลัพธ์ได้ทันที กรณีที่ 3 กรณีที่บิตบนสุดของ bitbank มีค่าเป็น 0 บิตถัดมาจะเป็น flag คือ บิตที่บอกให้ทราบว่าผลลัพธ์ในขั้นตอนนี้ จะต้องนำไปบวกหรือลบกับผลลัพธ์ก่อนหน้า เพื่อให้ได้เป็นผลลัพธ์สุดท้าย หลังจากนั้นให้วิเคราะห์ pmodel ว่าจะต้องดึงข้อมูลเข้ามากี่บิต และทั้ง 3 กรณี จะต้องทำการตั่งค่าของ fetchsize ตามปริมาณของบิตที่ดึงเข้ามาใช้งาน
- Add Difference ขั้นตอนนี้จะทำงานก็ต่อเมื่อ ในขั้นตอนของ Create Result เข้ากรณีที่ 3 เท่านั้น โดยจะทำการเพิ่มค่าของผลลัพธ์ชุดก่อนหน้า ด้วยค่าที่ดึงเข้ามา ถ้าหาก flag มีค่าเป็น 1 แต่หาก flag มีค่าเป็น 0 ให้ทำการลดค่าแทน
- CreateABSSUB ขั้นตอนนี้ จะทำการหาค่า abssub ซึ่งก็คือ ค่าสัมบูรณ์ของผลต่าง ระหว่างผลลัพธ์ในรอบก่อนหน้า กับผลลัพธ์ในรอบนี้

อย่างไรก็ตาม บางขั้นตอนได้มีการดัดแปลงการทำงาน เพื่อลดความซับซ้อนที่จะเกิดขึ้น ทำให้การทำงานจริงแตกต่างไปจากที่ได้อธิบายเอาไว้ข้างต้นเล็กน้อย แต่ก็สามารถให้ผลลัพธ์ที่เหมือนกันได้ เช่นส่วนของ Calculate Model จะมีการใช้ Register Array เพื่อเก็บข้อมูลว่าในรอบนั้น ๆ ค่าของ abssub ตรงกับโมเดลใด และใช้ Register อีกชุด ในการเก็บผลรวมเอาไว้ตลอดเวลา ทำให้ไม่ต้องหาผลรวมของ Register Array ในทุก ๆ รอบ

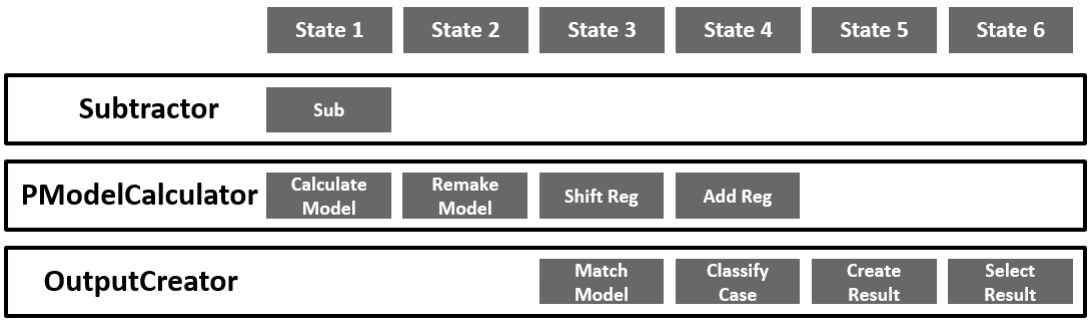
จะเห็นได้ว่าการทำงานในส่วนของการถอดรหัส จะใช้เวลาในการทำงานเท่ากันทุกรอบ ทำให้สามารถคำนวณระยะเวลาที่ใช้ในการถอดรหัสได้อย่างชัดเจน เช่น ถ้าหากอ้างอิงระบบกับ FPGA Spartan 6 LX9 Microboard ที่มีความถี่ของสัญญาณนาฬิกา 66.7 MHz จะทำให้สามารถถอดรหัสข้อมูลได้ประมาณ 9.5 ล้านชุดต่อวินาที และเช่นเดียวกันกับส่วนเข้ารหัสข้อมูลจะต้องมีส่วนที่ทำการใช้งาน DSP ดังนั้น ถ้าหากต้องการนำไปใช้งานจริง จะต้องออกแบบเพิ่มเติมในส่วนนี้ด้วย

4.3 SZAVGLBS

โครงสร้างวงจรของวิธีนี้ จะเหมือนกันกับวิธี SZAVGLB แต่จะมีความแตกต่างกันเล็กน้อยในส่วนของการจัดการ Register ทำให้เมื่อพิจารณาขั้นตอนการทำงานอย่างละเอียดจะเห็นได้ว่ามีขั้นตอนการทำงานที่ต่างกันในส่วนของการเพิ่มค่า Register เท่านั้น เนื่องจากวิธีนี้ถูกออกแบบให้มีการจัดการ Register ที่ดียิ่งขึ้นในเรื่องของปริมาณการใช้งานทรัพยากรฮาร์ดแวร์ แต่อย่างไรก็ตาม เมื่อมีการจัดการกับ Register ที่แตกต่างกัน จะส่งผลให้ประสิทธิภาพในการบีบอัดข้อมูล มีความแตกต่างกันด้วย ในหัวข้อนี้จะอธิบายแยกย่อยออกเป็น 2 ส่วน คือส่วนเข้ารหัส และส่วนถอดรหัส

4.3.1 Encoder circuit

วงจรหลักในการเข้ารหัส และการเชื่อมต่อนั้น จะมีลักษณะดังภาพประกอบที่ 4-1 ภายในแต่ละวงจรจะมีการทำงานแยกย่อยลงไปได้อีก ซึ่งลำดับในการทำงานเหล่านั้น จะถูกกำหนดด้วยสัญญาณ statecontrol ที่ถูกสร้างจากวงจร SignalController โดยจะมีขั้นตอนในการทำงานดังภาพประกอบที่ 4-5



ภาพประกอบที่ 4-5 ลำดับการทำงานในการเข้ารหัสของ SZAVGLBS

จากภาพประกอบที่ 4-5 จะเห็นได้ว่า แต่ละขั้นตอนจะทำงานใน statecontrol ที่แตกต่างกัน แต่จะมีบางขั้นตอนที่ทำงานพร้อมกันกับขั้นตอนอื่นได้ การทำงานส่วนใหญ่จะเหมือนกันกับวิธีการเข้ารหัสของ SZAVGLB แต่จะมีความแตกต่างกัน 2 ขั้นตอน คือ Shift Reg และ Add Reg ซึ่งอยู่ในวงจร PModelCalculator อย่างไรก็ตามจะอธิบายรายละเอียดของทุกขั้นตอน เพื่อให้ง่ายต่อการทำความเข้าใจ โดยแต่ละขั้นตอนของระบบ จะมีรายละเอียดในการทำงานดังนี้

- Sub เป็นการหาค่าสัมบูรณ์ของผลต่าง ระหว่างข้อมูลชุดก่อนหน้า กับข้อมูลชุดปัจจุบัน โดยจะใช้สัญญาณ data_in เป็นข้อมูลปัจจุบัน และ Register ภายใน ที่เก็บค่าข้อมูลชุดก่อนหน้า ในการคำนวณ ผลลัพธ์จะออกมาเป็น cdata (หรือก็คือ data_in), abssub คือค่าสัมบูรณ์ของผลต่าง และ flag คือ สัญญาณที่ใช้บอกว่าข้อมูลชุดปัจจุบันมีค่ามากกว่าหรือไม่ (เป็น 1 เมื่อข้อมูลปัจจุบันมีค่ามากกว่า)
- Calculate เป็นขั้นตอนในการคำนวณหาค่าเฉลี่ยของ Register เพื่อหาโมเดลที่ต้องการ (สูตรคำนวณในการหาค่าเฉลี่ย อยู่ในบทที่ 3 หัวข้อที่ 3.2.5)
- Remake Model จะทำการเปลี่ยนค่าเฉลี่ยที่ได้จาก Calculate ให้เป็นโมเดลที่พร้อมให้นำไปเปรียบเทียบได้ เช่น ถ้าหากค่าเฉลี่ยเป็น 3.21 จะได้โมเดลเป็น 0x1F หรือถ้าหากได้ค่าเฉลี่ยเป็น 7.55 จะได้โมเดลเป็น 0x01 โดยค่าโมเดลที่ถูกแปลงค่าแล้วนั้น จะถูกส่งออกไปทางช่องสัญญาณ pmodel ให้กับวงจร OutputCalculator
- Shift Reg ในขั้นตอนนี้ จะทำการตรวจสอง MSB ของ Register ทุกตัว ว่ามีค่าเป็น 1 หรือไม่ ถ้าหากมีตัวใดมีค่าเป็น 1 จะทำการ เลื่อนบิตของ Register ทุกตัว ลง 1 บิต (หรือทำการหารด้วย 2) ซึ่งเป็นวิธีจัดการกับ Register วิธีหนึ่งทำให้ไม่เกิด Overflow จากการเพิ่มค่า Register
- Add Reg ขั้นตอนนี้ จะทำการนำ abssub มาทำการวิเคราะห์ ว่าตรงกับโมเดลรูปแบบใด แล้วทำการเพิ่มค่า Register ที่เก็บค่าความถี่ของโมเดลรูปแบบนั้นขึ้น 1
- Match Model การที่จะทำงานในขั้นตอนนี้ได้นั้น จำเป็นต้องมี pmodel ซึ่งเป็นผลลัพธ์จากวงจร PModelCalculator เสียก่อน ซึ่งขั้นตอนนี้ จะทำการนำ abssub มาเปรียบเทียบกับ pmodel ว่า abssub ที่ได้นั้น ถูกจำแนกอยู่ในประเภทเดียวกับ pmodel ที่ได้หรือไม่ ซึ่งจะทำการเก็บผลลัพธ์เอาไว้ใน Register ขนาด 2 บิต โดยถ้าเปรียบเทียบแล้วตรงกัน จะได้ค่าเป็น 00 แต่หากไม่ตรงกัน จะได้ค่าเป็น 01
- Classify Case ในขั้นตอนนี้จะทำการนำ pmodel มาตรวจสอบว่าเป็นโมเดล 0Z, 1Z หรือ 2Z หรือไม่ ถ้าใช่ จะทำเขียนทับ Register ในขั้นตอน Match Model ให้มีค่าเป็น 11
- Create Result เป็นขั้นตอนในการหาผลลัพธ์ของการเข้ารหัสในรูปแบบต่าง ๆ โดยจะมีทั้งหมด 3 แบบด้วยกัน คือ แบบที่ 1 ผลลัพธ์ประกอบด้วย {0, flag,

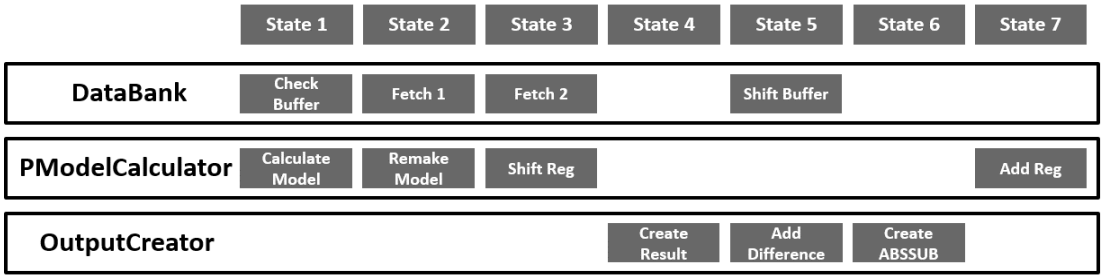
only significant bit of abssub}, แบบที่ 2 ผลลัพธ์ประกอบด้วย {1,cdata} และแบบที่ 3 ผลลัพธ์จะมีค่าเป็น cdata แบบตรงตัว

- Select Result ขั้นตอนนี้จะทำการเลือกคำตอบที่ได้จากขั้นตอน Create Result โดยใช้ Register ที่ถูกเขียนในขั้นตอน Match Model และผ่านการทำงานในขั้นตอน Classify Case เป็นตัวกำหนด โดยถ้าหากมีค่าเป็น 00 จะเลือกผลลัพธ์แบบที่ 1 แต่ถ้าหากมีค่าเป็น 01 จะเลือกผลลัพธ์แบบที่ 2 และจะเลือกผลลัพธ์แบบที่ 3 ในกรณีที่ Register นั้น มีค่าเป็น 11 ซึ่งผลลัพธ์ที่ถูกเลือกนั้นจะเป็นข้อมูลที่ถูกเข้ารหัสเสร็จสิ้นแล้ว

จากภาพรวมที่ได้กล่าวไว้ข้างต้น จะเห็นได้ว่า ระบบมีขั้นตอนการทำงานที่เป็นลำดับชัดเจน และใช้เวลาเท่ากันในทุก ๆ รอบ ทำให้สามารถคำนวณระยะเวลาที่ใช้เข้ารหัสข้อมูลได้ โดยถ้าหากอ้างอิงวงจรที่ได้ออกแบบกับ Spartan 6 LX9 Microboard ซึ่งมีความถี่สัญญาณนาฬิกา 66.7 MHz จะสามารถทำการเข้ารหัสได้ประมาณ 11.1 ล้านชุด ต่อวินาที ทั้งนี้ ถ้าหากจะนำวงจรไปใช้งานจริง จะต้องออกแบบส่วนของ PModelCalculator ใหม่ เนื่องจากมีการเรียกใช้งาน DSP ในการคูณ และการหาร ซึ่งหากสร้างวงจรขึ้นมา จะต้องใช้เวลาในการคำนวณในจุดนี้มากขึ้น

4.3.2 Decoder circuit

ในส่วนของการถอดรหัสของ SZAVGLBS จะมีความแตกต่างจากส่วนถอดรหัสของ SZAVGLB ไม่มากนัก จะมีส่วนที่แตกต่างกันเฉพาะส่วนของการเพิ่มค่าให้กับ Register ซึ่งอยู่ในวงจร PModelCalculator นอกเหนือจากนี้ จะเหมือนกันทุกส่วน โดยจะมีวงจรและการเชื่อมต่อดังภาพประกอบที่ 4-2 โดยแต่ละวงจรจะมีการทำงานแยกย่อยลงไปได้อีก ซึ่งขั้นตอนเหล่านี้ จะถูกควบคุมโดยสัญญาณ statecontrol ที่ถูกสร้างจากวงจร SignalController ให้มีลำดับการทำงานสอดคล้องกัน ดังภาพประกอบที่ 4-6



ภาพประกอบที่ 4-6 ลำดับการทำงานในการถอดรหัสของ SZAVGLBS

จากภาพประกอบที่ 4-6 จะเห็นว่าแต่ละขั้นตอนมีลำดับการทำงานที่สอดคล้องกัน โดยอาศัยสัญญาณ statecontrol ในการควบคุมลำดับการทำงาน ซึ่งลำดับการทำงานในการถอดรหัสของ SZAVGLBS นั้น จะเหมือนกันกับการถอดรหัสของ SZAVGLB เกือบทั้งหมด จะแตกต่างกันเฉพาะส่วนของวงจร PModelCalculator ในขั้นตอน Shift Reg และ Add Reg เท่านั้น อย่างไรก็ตาม

เพื่อให้ทำความเข้าใจได้ง่าย จะทำการอธิบายทุกขั้นตอน เพื่อให้เกิดความต่อเนื่อง ซึ่งแต่ละขั้นตอนจะมีรายละเอียดดังนี้

- Check Buffer วงจร DataBank นั้น ถูกสร้างขึ้นเพื่อทำการสำรองข้อมูลที่จะใช้ในการถอดรหัส เนื่องจากการถอดรหัสในแต่ละรอบ จะดึงข้อมูลมาใช้ไม่เท่ากัน โดยจะเป็นไปได้ตั้งแต่ 3 – 9 บิต จึงต้องมีการสำรองข้อมูลเอาไว้ใน Buffer อย่างเพียงพอ ซึ่งขั้นตอนนี้ จะทำการตรวจสอบ Buffer ภายในว่ามีที่ว่างเพียงพอจะใส่ข้อมูลขนาด 8 บิตลงไปหรือไม่ ถ้าหากมีช่องว่างพอก็จะทำการตั้งค่าของสัญญาณ buffstate ให้มีค่าเป็น 1 แต่ถ้าเหลือที่ว่างไม่พอจะทำการกำหนด buffstate ให้เป็น 0 โดยสัญญาณ buffstate ใช้ในการติดต่อกับระบบอื่น ๆ ให้ป้อนอินพุตตัวใหม่เข้ามาในระบบ
- Fetch 1 ขั้นตอนนี้จะทำงานก็ต่อเมื่อ buffstate มีค่าเป็น 1 เท่านั้น โดยขั้นตอนนี้ จะทำการนำข้อมูลที่อ่านได้จาก data_in มาเขียนลงไปใน Buffer ที่ว่างอยู่ และในขั้นตอนนี้ จะต้องตรวจสอบด้วยว่า ถ้าเขียนข้อมูลลงไปใน Buffer แล้ว จะมีที่ว่างเหลือพอที่จะเก็บข้อมูลอีก 8 บิต หรือไม่ ถ้ามากพอจะทำการตั้งค่า buffstate เป็น 1 แต่ถ้าไม่พอ จะทำการตั้งค่า buffstate เป็น 0
- Fetch 2 จะทำงานคล้ายกับ Fetch 1 โดยจะทำงานก็ต่อเมื่อ buffstate มีค่าเป็น 1 เท่านั้น โดยจะทำการนำข้อมูลที่อ่านได้จาก data_in มาเขียนลงบน Buffer หลังจากนั้นจะกำหนดให้ buffstate มีค่าเป็น 0
- Shift Buffer ขั้นตอนนี้จะทำงานในรอบที่ 5 หลังจากที่วงจร OutputCalculator ส่ง fetchsize ออกมาแล้ว จะนำ fetchsize มาเลื่อน Buffer เพื่อให้มีที่ว่างสำหรับขั้นตอน fetch 1 และ fetch 2 ในรอบถัดไป
- Calculate Model ขั้นตอนนี้จะทำการนำ Register ที่เก็บความถี่ของโมเดลมาหาค่าเฉลี่ย (สูตรในการหาค่าเฉลี่ย อยู่ในบทที่ 3 หัวข้อที่ 3.2.5)
- Remake Model นำค่าเฉลี่ยที่ได้จากขั้นตอน Calculate Model มาแปลงให้เป็นค่าของโมเดลที่สามารถนำไปเปรียบเทียบได้ เช่น ถ้าหากค่าเฉลี่ยเป็น 4.01 จะได้โมเดลเป็น 0x0F ถ้าหากค่าเฉลี่ยเป็น 8.00 จะได้โมเดลเป็น 0x00
- Shift Reg ทำการเตรียม Register ให้พร้อมสำหรับการเพิ่มค่า โดยจะตรวจสอบ MSB ของ Register ทุกตัว ถ้าหากมีตัวใดตัวหนึ่งมีค่าเป็น 1 จะทำการเลื่อนค่าของ Register ทุกตัว ลง 1 บิต (สามารถใช้การหารด้วย 2 แทนได้)
- Add Reg นำ abssub มาวิเคราะห์ว่าตรงกับโมเดลรูปแบบใด แล้วจึงเพิ่มค่าของ Register ที่เก็บความถี่ของโมเดลรูปแบบนั้นขึ้น 1
- Create Result ทำการสร้างผลลัพธ์ของการถอดรหัสออกมา โดยพิจารณาเงื่อนไขตามลำดับดังนี้ เงื่อนไขที่ 1 ถ้าหาก pmodel มีค่าเป็น 0Z, 1Z หรือ 2Z ให้ผลลัพธ์ของการถอดรหัส (result) มีค่าเป็น 8 บิตบนของ bitbank ได้ทันที และ fetchsize มีค่าเป็น 8 เงื่อนไขที่ 2 ถ้าหาก MSB ของ bitbank มีค่าเป็น 1 ให้นำบิตที่เหลือของ bitbank เป็นคำตอบได้ทันที และ fetchsize มีค่าเป็น 9

เงื่อนไขที่ 3 ถ้าหาก MSB ของ bitbank มีค่าเป็น 0 บิตถัดมาจะถูกเก็บไว้ใน Register ที่ชื่อ cflag จากนั้นต้องพิจารณาต่อว่า pmodel มีค่าเป็นเท่าใด โดยสามารถนับจำนวนบิตที่มีค่าเป็น 1 ของ pmodel มาเป็นจำนวนของข้อมูลที่จะดึงออกมา โดยบิตที่ถูกดึงออกมานั้นจะอยู่ถัดจาก 2 บิตแรก และ fetchsize จะมีค่ามากกว่าบิตที่ถูกดึงออกมา อยู่ 2 (เนื่องจากต้องรวม 2 บิตแรกด้วย)

- Add Difference ขั้นตอนนี้จะทำงานก็ต่อเมื่อ การทำงานของ Create Result เข้าเงื่อนไขที่ 3 เท่านั้น โดยจะพิจารณาจาก Register cflag ถ้าหาก cflag มีค่าเป็น 0 ผลลัพธ์ (result) จะเป็นผลต่างของผลลัพธ์ตัวก่อนหน้า กับค่าที่ถูกดึงออกมาในเงื่อนไขที่ 3 ในขั้นตอน Create Result แต่ถ้าหาก cflag มีค่าเป็น 1 ผลลัพธ์จะเป็นรวมของผลลัพธ์ตัวก่อนหน้า กับค่าที่ดึงออกมาในเงื่อนไขที่ 3 ของขั้นตอน Create Result
- Create ABSSUB เป็นการหาค่า abssub เพื่อส่งไปให้วงจร PModelCalculator ใช้ในการเพิ่มค่า Register โดยค่า abssub จะเป็นค่าสัมบูรณ์ของผลต่างระหว่างผลลัพธ์ตัวก่อนหน้า กับผลลัพธ์ในรอบปัจจุบัน

จะเห็นได้ว่า การทำงานในส่วนถอดรหัสของ SZAVGLBS มีความคล้ายคลึงกันกับส่วนถอดรหัสของ SZAVGLB ค่อนข้างมาก แต่จะมีขั้นตอนที่แตกต่างกันเล็กน้อย เนื่องจากวิธีการจัดการกับ Register ที่แตกต่างกัน ได้แก่ขั้นตอน Shift Reg และ Add Reg แต่อย่างไรก็ตามการถอดรหัสของทั้ง SZAVGLB และ SZAVGLBS จะใช้เวลาในการทำงาน 7 รอบสัญญาณนาฬิกา แต่ยังสามารถที่จะลดระยะเวลาในการทำงานได้อีก เนื่องจากการทำงานของ PModelCalculator ยังสามารถที่จะเลื่อนขั้นตอนการทำงานได้

จากวงจรถอดรหัสโดยรวมของ SZAVGLBS จะเห็นได้ว่ากระบวนการทำงานมีขั้นตอนและลำดับที่ชัดเจน ทำให้สามารถคำนวณระยะเวลาที่ใช้ในการถอดรหัสข้อมูลได้ โดยหากสังเคราะห์วงจรลงบน FPGA Spartan 6 LX9 Microboard ซึ่งมีความถี่ของสัญญาณนาฬิกาเท่ากับ 66.7 MHz แล้ว จะทำให้สามารถถอดรหัสข้อมูลได้ประมาณ 9.5 ล้านชุด ต่อวินาที ทั้งนี้การสังเคราะห์วงจรจริง ๆ นั้น จะต้องคำนึงถึงการนำ DSP มาใช้งานด้วย ซึ่งจำทำให้ประสิทธิภาพในการทำงานลดลงไปอีก ตามความสามารถของวงจรที่นำมาใช้ทดแทน

4.4 วิเคราะห์ และสรุปผล

ในส่วนของการนำวิธีการบีบอัดข้อมูลที่ได้ออกแบบไว้ มาทำการออกแบบใหม่ลงบนฮาร์ดแวร์นั้น สามารถทำงานได้เหมือนกันกับวิธีการต้นฉบับ แม้ว่าจะมีบางจุดที่มีการใช้เทคนิคที่แตกต่างออกไป แต่ก็สามารถทำงานได้ถูกต้องเช่นกัน ทั้งนี้โดยภาพรวมของระบบทั้ง SZAVGLB และ SZAVGLBS จะมีความแตกต่างกันเพียงเล็กน้อยเท่านั้น และใช้ระยะเวลาในการทำงานที่คงที่และชัดเจน ไม่มีความซับซ้อน ทำให้น่าจะมีความเหมาะสมกับเครือข่ายเซนเซอร์ไร้สาย แต่ทั้งนี้จะต้องทำการวิเคราะห์ร่วมกับผลลัพธ์ในการเข้ารหัสข้อมูล และปริมาณทรัพยากรเมื่อทำการสังเคราะห์วงจรลงบน FPGA ร่วมด้วย จึงจะสามารถตัดสินได้ว่าระบบที่ได้ทำการออกแบบนั้น มีประสิทธิภาพในการทำงานมากน้อยเพียงใด

ในบทต่อไป จะกล่าวถึงผลลัพธ์ในการทดสอบความสามารถในการเข้ารหัสของแต่ละวิธีที่ได้ทำการออกแบบไว้ รวมไปถึงปริมาณทรัพยากรที่เกิดจากการสังเคราะห์วงจรทั้งในส่วนของการเข้ารหัส และส่วนถอดรหัส ของ SZAVGLB และ SZAVGLBS

บทที่ 5
การทดสอบประสิทธิภาพของอัลกอริทึม

ในบทนี้จะกล่าวถึงวิธีการ และผลลัพธ์ในการทดสอบประสิทธิภาพของอัลกอริทึมที่ได้ออกแบบไว้ โดยจะแบ่งออกเป็นสองส่วน คือ ส่วนของการทดสอบความสามารถในการบีบอัดข้อมูล กับข้อมูลรูปแบบต่าง ๆ และส่วนของการทดสอบปริมาณทรัพยากรฮาร์ดแวร์ที่ถูกใช้ในการสร้างวงจร ทั้งส่วนของการเข้ารหัส และส่วนของการถอดรหัส

5.1 การทดสอบความสามารถในการบีบอัดข้อมูล

ในการทดสอบความสามารถในการบีบอัดข้อมูลนั้น จะทดสอบกับข้อมูลหลายประเภท เพื่อให้ครอบคลุมการใช้งานในหลาย ๆ รูปแบบ โดยข้อมูลที่นำมาทดสอบนั้นจะประกอบด้วย ข้อมูลสภาพอากาศ, ข้อมูลรูปภาพ และข้อมูลคลื่นไฟฟ้าสมอง (EEG) ซึ่งข้อมูลดังกล่าว จะมีคุณลักษณะที่แตกต่างกัน เป็นผลให้ประสิทธิภาพในการบีบอัดข้อมูลแตกต่างกันด้วย โดยอัลกอริทึมที่เลือกมาทดสอบนั้นจะมีทั้งหมด 3 กลุ่ม คือ CoXoH (เป็นอัลกอริทึมที่ทำการศึกษามา ซึ่งถูกพัฒนาต่อยอดมาจาก Adaptive Huffman ให้มีประสิทธิภาพมากยิ่งขึ้น), DZ SZ และ SZAVG ที่ทำงานร่วมกับ LBS และกลุ่มอัลกอริทึมแบบ Hybrid

การทดสอบความสามารถในการบีบอัดข้อมูลนั้น จะวัดด้วยอัตราการบีบอัดข้อมูล (CR : Compression Ratio) โดยสูตรในการคำนวณจะสามารถแจกแจงได้ดังนี้

$$CR = \frac{Rd - Cd}{Rd} \times 100 \qquad (5.1)$$

เมื่อกำหนดให้ CR คืออัตราการบีบอัดข้อมูล
Rd คือจำนวนบิตของข้อมูลต้นฉบับ
Cd คือจำนวนบิตของข้อมูลที่ถูกเข้ารหัสแล้ว

จากสูตรข้างต้น ค่า CR สามารถหาได้จากผลต่างระหว่างจำนวนบิตของข้อมูลต้นฉบับ กับจำนวนบิตของข้อมูลที่ถูกเข้ารหัสแล้ว หารด้วยจำนวนบิตของข้อมูลต้นฉบับ ทั้งหมดคูณด้วย 100 มีหน่วยเป็นเปอร์เซ็นต์

5.1.1 ข้อมูลสภาพอากาศ

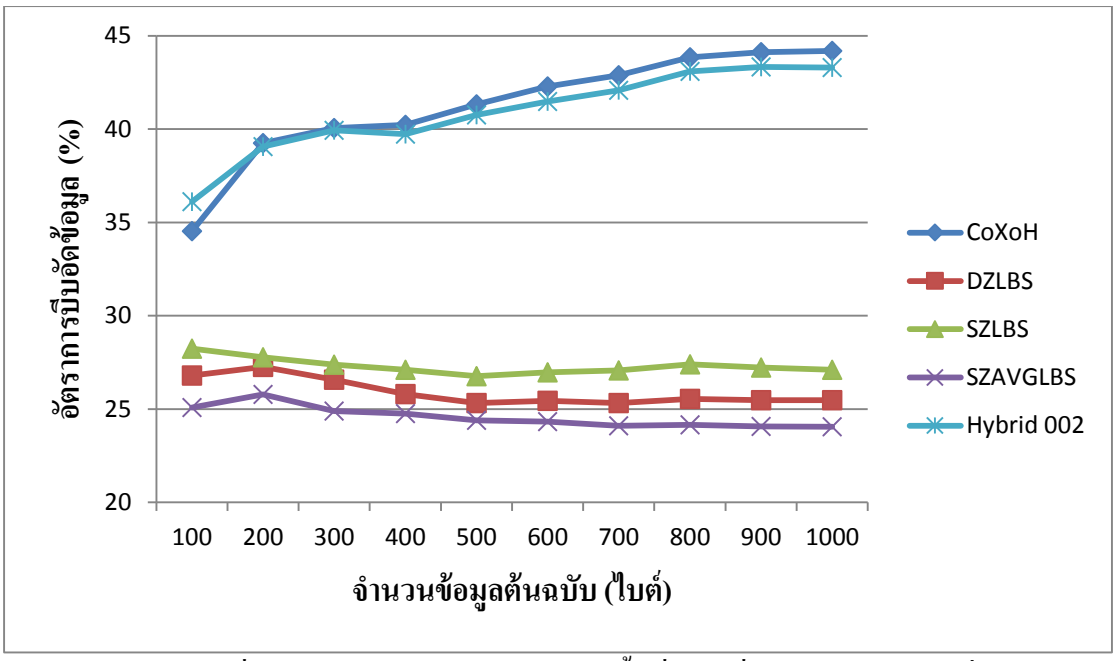
ในการทดสอบประสิทธิภาพในการบีบอัดข้อมูลกับการเข้ารหัสในรูปแบบต่าง ๆ จำเป็นที่จะต้องทราบถึงคุณลักษณะของข้อมูลนั้น ๆ ก่อน ซึ่งข้อมูลแต่ละแบบ จะมีคุณลักษณะที่แตกต่างกัน ทำให้การใช้อัลกอริทึมใด ๆ เพื่อทำการบีบอัดนั้น จะมีความสามารถแตกต่างกันกับข้อมูลต่างชนิดกัน โดยในหัวข้อนี้จะทดสอบความสามารถในการบีบอัดกับข้อมูลสภาพอากาศ โดยข้อมูลสภาพอากาศที่ถูกเลือกมาใช้ในการทำการทดสอบนั้น คือข้อมูลค่าความชื้น และข้อมูลค่าอุณหภูมิ

ข้อมูลความชื้นที่นำมาทดสอบ เป็นข้อมูลความชื้นที่ได้จากสนามบิน 10 แห่ง ตั้งแต่วันที่ 1 เดือนมกราคม ปีคริสตศักราช 2013 [16] ซึ่งข้อมูลแต่ละตัว จะมีขนาด 8 บิต โดยข้อมูลที่จะถูกนำมาทดสอบนั้น จะมีอัตราการสุ่มตัวอย่างอยู่ที่ 1 รอบต่อชั่วโมง, 1 รอบต่อ 2 ชั่วโมง, 1 รอบต่อ 4 ชั่วโมง และ 1 รอบต่อ 8 ชั่วโมง การแบ่งข้อมูลให้มีอัตราการสุ่มตัวอย่างที่แตกต่างกันนั้น จะทำให้ข้อมูลมีคุณสมบัติที่ต่างออกไป ทั้งค่าเฉลี่ยของข้อมูล และค่าความแปรปรวนของข้อมูล ซึ่งอาจจะทำให้ผลลัพธ์ในการบีบอัดข้อมูลมีความแตกต่างกันออกไปด้วย โดยค่าเฉลี่ยของข้อมูลและค่าความแปรปรวนจะถูกแจกแจงแยกกันไปตามข้อมูลแต่ละชุดที่ถูกทดสอบ โดยในส่วนของข้อมูลสภาพอากาศ จะมีข้อมูลที่ถูกทดสอบค่อนข้างมาก จึงจะทำการอธิบายผลในรูปแบบของกราฟ

อัลกอริทึมที่ถูกนำมาทดสอบ จะประกอบไปด้วย CoXoH, อัลกอริทึมที่ออกแบบ (SZ, DZ, SZAVG) ที่นำเทคนิค LBS มาประยุกต์ใช้งานด้วย โดยจะมีการกำหนดขนาดของ Buffer ของฐานข้อมูลไว้ที่ 4 บิต และในส่วนของ Hybrid จะเป็นการทำไฮบริดจ์กันระหว่าง CoXoH กับ SZAVGLBS โดยที่ตัวไฮบริดจะมีขนาดของ Buffer อยู่ที่ 8 รอบ ในขณะที่ SZAVGLBS ที่อยู่ภายในมีขนาดของ Buffer ที่ 4 บิต

การทดสอบข้อมูล จะมีข้อมูลทั้งหมด 4 กลุ่มใหญ่ แบ่งตามอัตราการสุ่มตัวอย่างที่แตกต่างกัน ข้อมูลแต่ละกลุ่ม จะมีข้อมูลทั้งหมด 10 ชุด ข้อมูลแต่ละชุด จะมีข้อมูลภายในทั้งหมด 1000 ตัว ข้อมูลแต่ละตัวมีขนาด 8 บิต การทดสอบจะทำการแบ่งข้อมูล 1000 ตัวออกเป็นช่วง โดยช่วงของข้อมูลจะเริ่มจาก ข้อมูล 100 ตัวแรก แล้วเพิ่มปริมาณเป็น 200 ตัว และจะเพิ่มปริมาณขึ้นครั้งละ 100 จนข้อมูลมีขนาด 1000 ตัว ทั้งนี้เพื่อทดสอบความสามารถในการเข้ารหัสในเวลาที่ต่างกัน

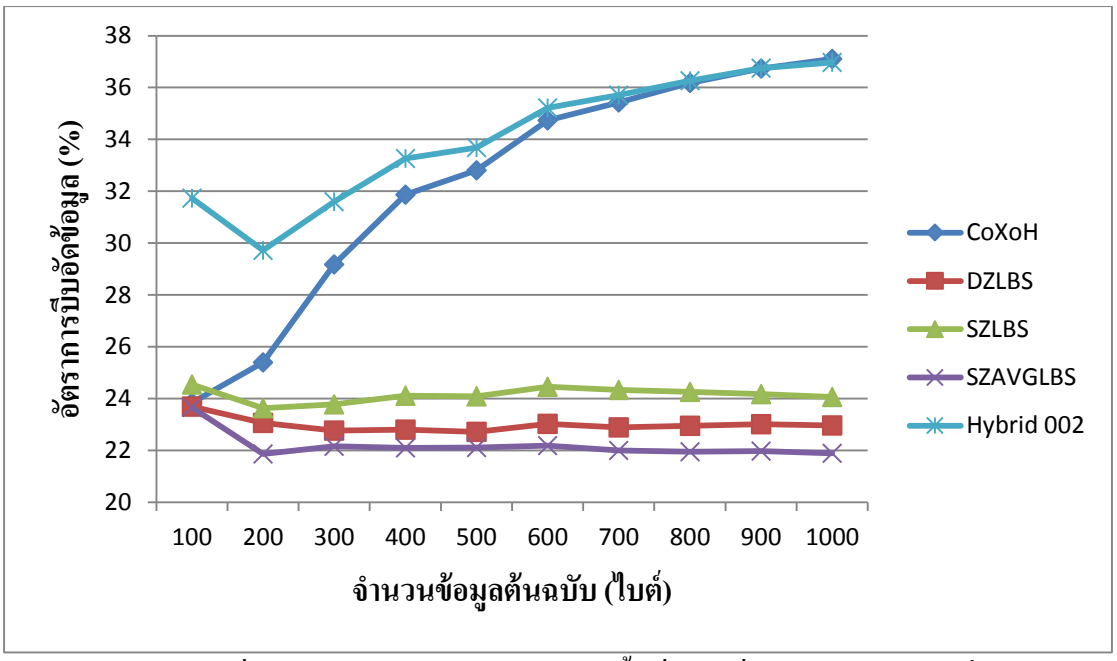
การทดสอบจะเริ่มจากข้อมูลความชื้นกลุ่มแรก ซึ่งเป็นข้อมูลความชื้นที่มีอัตราการสุ่มตัวอย่างที่ 1 ครั้งต่อชั่วโมง ซึ่งข้อมูลที่ถูกนำมาทดสอบนั้น จะมีค่าเฉลี่ยโดยเฉลี่ยที่ข้อมูล 1000 ตัว ของข้อมูล 10 ชุด อยู่ที่ 70 และมีค่าความแปรปรวนโดยเฉลี่ยที่ข้อมูล 1000 ตัว ของข้อมูล 10 ชุด อยู่ที่ 221 ค่า CR ที่ได้จะถูกนำมาหาค่าเฉลี่ย (จากข้อมูลจำนวน 10 ชุด) แล้วจึงนำไปเป็นผลลัพธ์ ดังแสดงไว้ในภาพประกอบที่ 5-1



ภาพประกอบที่ 5-1 การทดสอบด้วยข้อมูลความชื้นที่ความถี่ข้อมูล 1 รอบต่อ 1 ชั่วโมง

จากภาพประกอบที่ 5-1 เป็นการเปรียบเทียบความสามารถในการเข้ารหัสข้อมูลของแต่ละอัลกอริทึมกับข้อมูลในแต่ละช่วง จะเห็นได้ว่า CoXoH จะให้ค่า CR สูงที่สุดเมื่อเทียบกับอัลกอริทึมอื่น ๆ รองลงมาจะเป็นอัลกอริทึม Hybrid 002 ซึ่งมี CR น้อยกว่า CoXoH เล็กน้อย ในส่วนของ DZ, SZ และ SZAVG ที่ทำงานร่วมกันกับ LBS นั้น จะมีความสามารถในการบีบอัดข้อมูลที่แตกต่างกันเล็กน้อย โดยที่ SZLBS มีความสามารถในการบีบอัดข้อมูลสูงกว่าอัลกอริทึมอื่น และอัลกอริทึมที่มีความสามารถในการบีบอัดข้อมูลต่ำที่สุดจะเป็น SZAVGLBS อย่างไรก็ตาม เมื่อสภาพของข้อมูลแตกต่างกัน จะทำให้ความสามารถในการบีบอัดข้อมูลแตกต่างกันไปด้วย

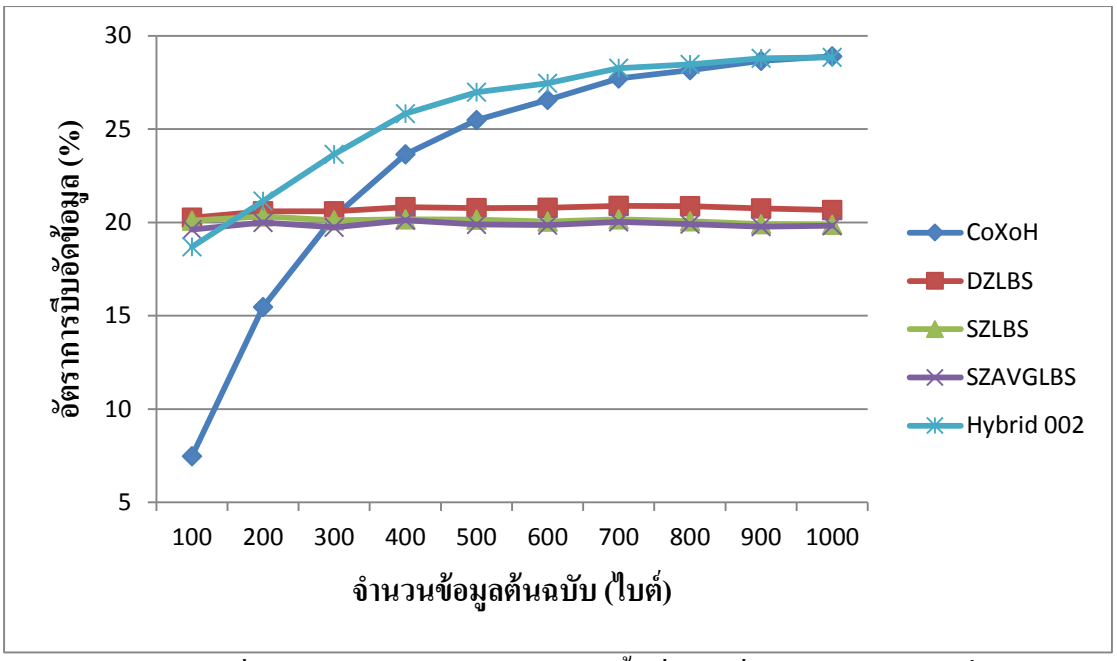
การทดสอบในส่วนถัดไปจะเป็นข้อมูลกลุ่มที่ 2 ซึ่งเป็นข้อมูลความชื้นที่มีอัตราการสุ่มตัวอย่างที่ 1 ครั้งต่อ 2 ชั่วโมง ข้อมูลดังกล่าวจะมีค่าเฉลี่ยโดยเฉลี่ยที่ข้อมูล 1000 ตัว ของข้อมูล 10 ชุด อยู่ที่ 72 และมีค่าความแปรปรวนโดยเฉลี่ยที่ข้อมูล 1000 ตัว ของข้อมูล 10 ชุด อยู่ที่ 249 ซึ่งจะเห็นได้ว่าค่าความแปรปรวนโดยเฉลี่ยนั้น จะมีค่าแตกต่างออกไป เนื่องจากการใช้อัตราการสุ่มตัวอย่างที่น้อยลง ซึ่งผลลัพธ์ในการบีบอัดข้อมูลนั้น จะแสดงไว้ในภาพประกอบที่ 5-2



ภาพประกอบที่ 5-2 การทดสอบด้วยข้อมูลความชื้นที่ความถี่ข้อมูล 1 รอบต่อ 2 ชั่วโมง

ภาพประกอบที่ 5-2 เป็นการเปรียบเทียบความสามารถในการบีบอัดข้อมูลของแต่ละอัลกอริทึมกับข้อมูลในช่วงต่าง ๆ โดยจะสังเกตได้ว่าอัลกอริทึม Hybrid 002 นั้น จะมีค่า CR สูงสุด ตามมาด้วย CoXoH ซึ่งจะมีค่า CR ในช่วงแรกไม่มากนัก แต่เมื่อทำการเข้ารหัสข้อมูลมากขึ้นจะสามารถสร้างค่า CR ที่มากขึ้นเช่นเดียวกันกับ Hybrid ในส่วนของอัลกอริทึมที่ได้ออกแบบไว้นั้นจะยังคงมีค่า CR ที่ใกล้เคียงกัน โดยที่ SZLBS ยังคงมีค่า CR ที่สูงที่สุด และ SZAVGLBS มีค่า CR ต่ำที่สุด ซึ่งจากทั้งภาพประกอบที่ 5-1 และภาพประกอบที่ 5-2 จะสังเกตได้ว่าอัลกอริทึมที่ได้ทำการออกแบบไว้นั้น จะมีค่า CR ที่คงที่ แม้ว่าข้อมูลจะมีปริมาณมากขึ้น

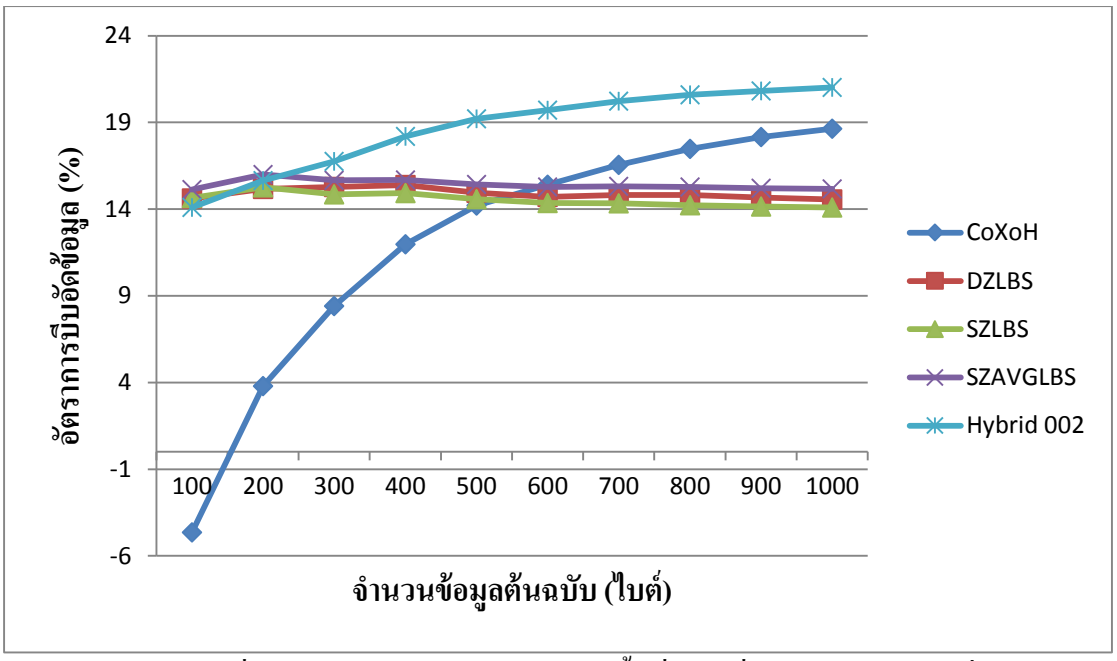
ส่วนของการทดสอบถัดไปนั้น จะทำการทดสอบข้อมูลกลุ่มที่ 3 ซึ่งเป็นข้อมูลความชื้นที่มีอัตราการสุ่มตัวอย่างอยู่ที่ 1 ครั้งต่อ 4 ชั่วโมง ข้อมูลดังกล่าวจะมีค่าเฉลี่ยโดยเฉลี่ยที่ข้อมูล 1000 ตัว ของข้อมูล 10 ชุด อยู่ที่ 70 และมีค่าความแปรปรวนโดยเฉลี่ยที่ข้อมูล 1000 ตัว ของข้อมูล 10 ชุด อยู่ที่ 267 ซึ่งจากข้อมูลสองชุดก่อนหน้านี้ จะเห็นว่า ค่าความแปรปรวนโดยเฉลี่ยจะมีแนวโน้มที่จะมีค่าเพิ่มขึ้น ผลลัพธ์ที่ได้จากการบีบอัดข้อมูลจะแสดงไว้ในภาพประกอบที่ 5-3



ภาพประกอบที่ 5-3 การทดสอบด้วยข้อมูลความชื้นที่ความถี่ข้อมูล 1 รอบต่อ 4 ชั่วโมง

จากภาพประกอบที่ 5-3 จะเห็นได้ว่าอัลกอริทึม Hybrid 002 นั้น สามารถสร้างค่า CR ได้สูงที่สุดในทุก ๆ ช่วงของข้อมูล ในขณะที่ CoXoH นั้น จะมีความสามารถในการเข้ารหัสข้อมูล ที่สูงเช่นกัน แต่การบีบอัดข้อมูลในช่วงแรก ๆ นั้น จะทำได้ไม่ดีนัก โดยจะมีค่า CR สูงขึ้นเมื่อทำการบีบอัดข้อมูลมากขึ้น ส่วนของอัลกอริทึมที่ได้ทำการออกแบบไว้นั้น จะมีความสามารถที่ใกล้เคียงกันโดยที่ DZLBS จะมีความสามารถสูงกว่าอัลกอริทึมอื่น ๆ เล็กน้อย ในขณะที่ SZLBS และ SZAVGLBS จะมีความสามารถในการบีบอัดที่ต่ำกว่าเล็กน้อย จากภาพประกอบที่ 5-1 ภาพประกอบที่ 5-2 และภาพประกอบที่ 5-3 จะเห็นได้ว่าอัลกอริทึมที่ออกแบบไว้ มีความสามารถในการบีบอัดข้อมูลที่ใกล้เคียงกันในทุก ๆ ช่วงของข้อมูล

การทดสอบต่อไป จะเป็นการทดสอบประสิทธิภาพในการบีบอัดข้อมูล เมื่อใช้กับข้อมูลกลุ่มที่ 4 ซึ่งมีอัตราการสุ่มตัวอย่างของข้อมูลที่ 1 ครั้งต่อ 8 ชั่วโมง ข้อมูลที่นำมาใช้ทดสอบจะมีค่าเฉลี่ยโดยเฉลี่ยที่ 1000 ตัว ของข้อมูล 10 ชุด อยู่ที่ 67 และมีค่าความแปรปรวนโดยเฉลี่ยที่ 1000 ตัว ของข้อมูล 10 ชุด อยู่ที่ 313 จะเห็นได้ว่าค่าความแปรปรวนโดยเฉลี่ย จะมีค่าเพิ่มขึ้นเมื่อข้อมูลมีอัตราการสุ่มตัวอย่างลดลง ผลลัพธ์ของการบีบอัดข้อมูล แสดงไว้ในภาพประกอบที่ 5-4



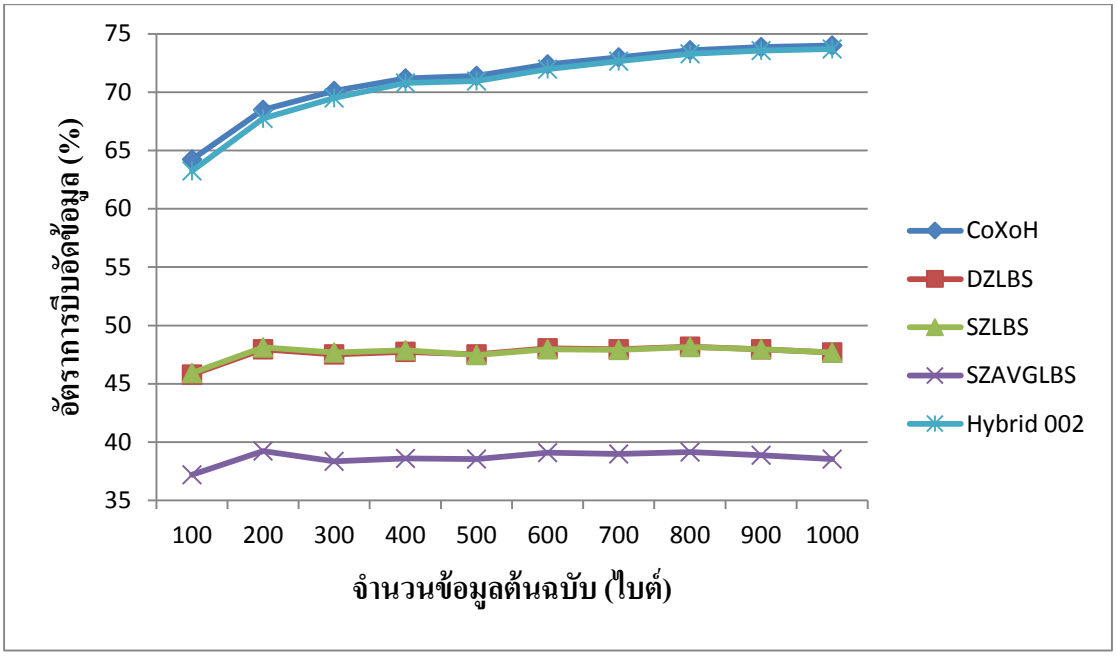
ภาพประกอบที่ 5-4 การทดสอบด้วยข้อมูลความชื้นที่ความถี่ข้อมูล 1 รอบต่อ 8 ชั่วโมง

จากภาพประกอบที่ 5-4 จะสังเกตได้ว่า อัลกอริทึม Hybrid 002 มีความสามารถในการบีบอัดข้อมูลสูงสุดทุกช่วงของข้อมูล ในขณะที่ CoXoH จะบีบอัดข้อมูลได้ไม่ดีในช่วงข้อมูลแรก ๆ แต่จะมีค่า CR สูงขึ้นเมื่อบีบอัดข้อมูลมากขึ้น โดยอัลกอริทึมที่ออกแบบไว้จะมีความสามารถในการบีบอัดข้อมูลเท่ากันในทุก ๆ ช่วงของข้อมูล และมีประสิทธิภาพสูงกว่า CoXoH ในช่วงแรก

จากการทดสอบความสามารถในการบีบอัดข้อมูลของแต่ละอัลกอริทึม เมื่อนำมาบีบอัดข้อมูลความชื้นที่มีอัตราการสุ่มตัวอย่างที่แตกต่างกัน ทำให้สามารถสรุปได้ว่า อัลกอริทึม Hybrid 002 สามารถสร้างค่า CR ได้สูงที่สุด และ CoXoH สามารถสร้าง CR ได้สูงรองลงมา ซึ่งทั้ง Hybrid 002 และ CoXoH นั้น จะสร้างค่า CR ได้ไม่สูงมากนักในการบีบอัดข้อมูลในช่วงแรก แต่เมื่อบีบอัดข้อมูลมากขึ้น ก็จะสามารถสร้างค่า CR ที่สูงขึ้นได้ ในส่วนของอัลกอริทึมที่ได้ออกแบบไว้จะมีความสามารถในการบีบอัดข้อมูลที่ใกล้เคียงกัน โดยจะมีความสามารถในการบีบอัดข้อมูลเท่ากันในทุก ๆ ช่วง แต่อย่างไรก็ตาม CoXoH จะมีความสามารถลดลง เมื่อใช้บีบอัดข้อมูลที่มีอัตราการสุ่มตัวอย่างที่น้อยลง ในส่วนของอัลกอริทึมที่ออกแบบไว้ก็เช่นเดียวกัน จะมีความสามารถในการเข้ารหัสที่ลดลงเมื่อใช้กับข้อมูลมีอัตราการสุ่มตัวอย่างที่น้อยลง

การทดสอบกับข้อมูลสภาพอากาศ จะมีการทดสอบกับข้อมูลอุณหภูมิ โดยมีแนวทางในการทดสอบที่เหมือนกันกับการทดสอบด้วยข้อมูลความชื้น ซึ่งข้อมูลที่นำมาทดสอบเป็นข้อมูลที่ได้มาจากแหล่งเดียวกับข้อมูลความชื้น โดยข้อมูล 1 ตัว จะมีขนาด 12 บิต และมีจำนวน 1000 ตัว ใน 1 ชุด มี 10 ชุดใน 1 กลุ่ม ข้อมูลแต่ละกลุ่มจะมีอัตราการสุ่มตัวอย่างที่แตกต่างกัน แบ่งเป็นข้อมูลที่มีอัตราการสุ่มตัวอย่าง 1 ครั้งต่อชั่วโมง, 1 ครั้งต่อ 2 ชั่วโมง, 1 ครั้งต่อ 4 ชั่วโมง และ 1 ครั้งต่อ 8 ชั่วโมง การทดสอบดังกล่าว จะนำเสนอข้อมูลในรูปของกราฟ เพื่อเปรียบเทียบกันระหว่างความสามารถในการบีบอัดข้อมูลด้วยอัลกอริทึมต่าง ๆ

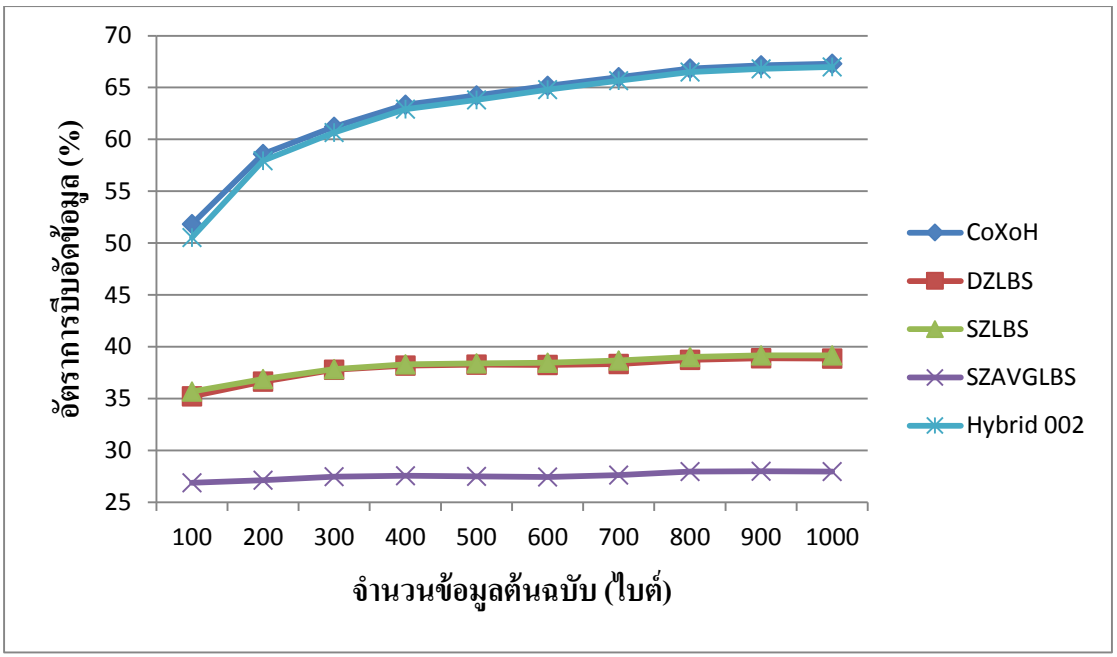
ข้อมูลกลุ่มแรก เป็นข้อมูลอุณหภูมิ ที่มีอัตราการสุ่มตัวอย่าง 1 ครั้งต่อชั่วโมง จะมีค่าเฉลี่ยโดยเฉลี่ยที่ 1000 ตัว ของข้อมูล 10 ชุด อยู่ที่ 2227 และมีค่าความแปรปรวนโดยเฉลี่ยที่ข้อมูล 1000 ตัว ของข้อมูล 10 ชุด อยู่ที่ 17604 ผลการทดสอบจะแสดงไว้ในภาพประกอบที่ 5-5



ภาพประกอบที่ 5-5 การทดสอบด้วยข้อมูลอุณหภูมิที่ความถี่ข้อมูล 1 รอบต่อ 1 ชั่วโมง

จากภาพประกอบที่ 5-5 จะสังเกตได้ว่า CoXoH เป็นอัลกอริทึมที่มีประสิทธิภาพสูงที่สุด โดยจะมีค่า CR สูงกว่า Hybrid 002 เล็กน้อย ส่วนอัลกอริทึมที่มีประสิทธิภาพต่ำที่สุดคือ SZAVGLBS

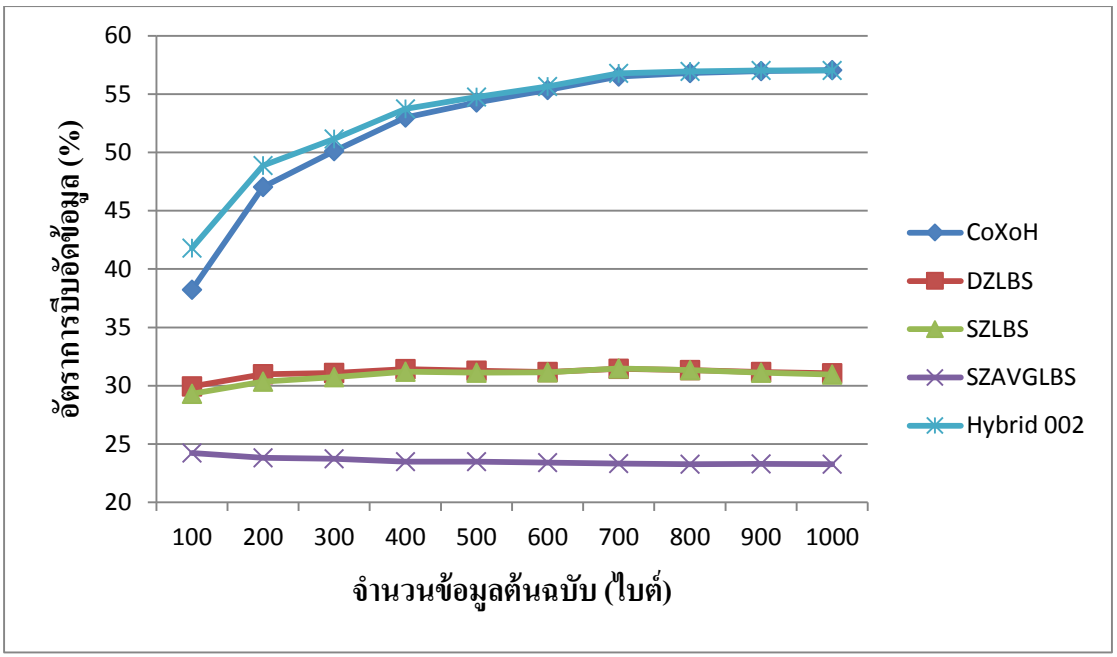
ข้อมูลกลุ่มที่ 2 เป็นข้อมูลอุณหภูมิ ที่มีอัตราการสุ่มตัวอย่าง 1 ครั้งต่อ 2 ชั่วโมง ซึ่งตัวข้อมูล จะมีค่าเฉลี่ยโดยเฉลี่ยที่ข้อมูล 1000 ตัว ของข้อมูล 10 ชุด อยู่ที่ 2227 และจะมีค่าความแปรปรวนโดยเฉลี่ยที่ข้อมูล 1000 ตัว ของข้อมูล 10 ชุด อยู่ที่ 15220 โดยผลลัพธ์ในการบีบอัดข้อมูล จะแสดงไว้ในภาพประกอบที่ 5-6



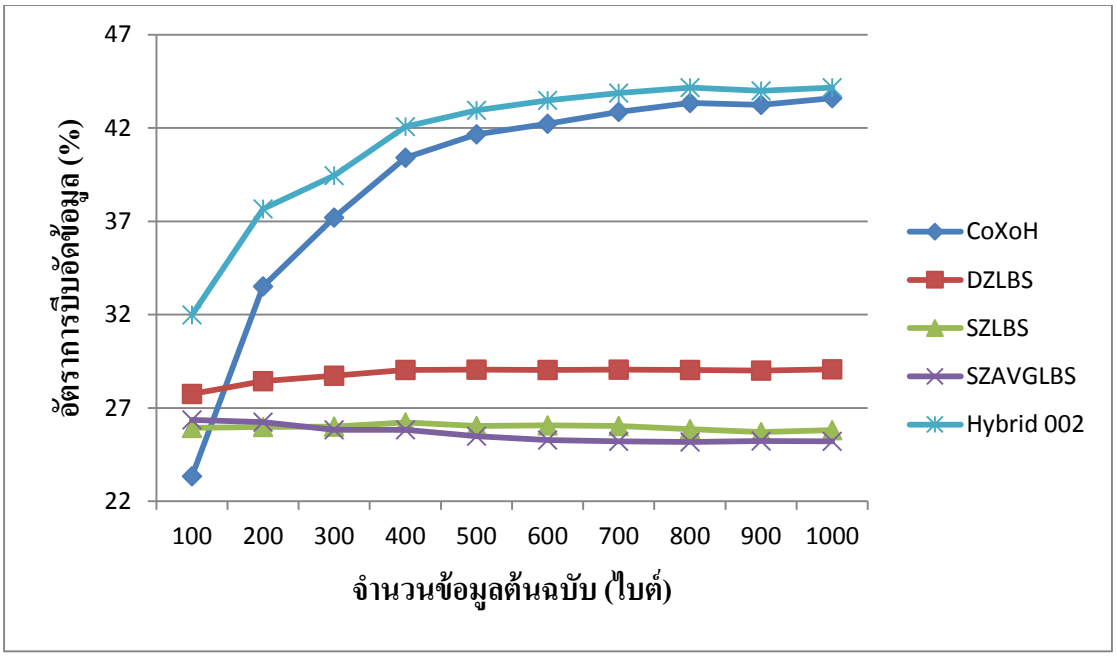
ภาพประกอบที่ 5-6 การทดสอบด้วยข้อมูลอุณหภูมิที่ความถี่ข้อมูล 1 รอบต่อ 2 ชั่วโมง

จากภาพประกอบที่ 5-6 อัลกอริทึมที่มีประสิทธิภาพสูงที่สุดยังคงเป็น CoXoH และมี Hybrid 002 อัลกอริทึมที่สามารถสร้างค่า CR ได้ใกล้เคียงกัน ในขณะที่ SZAVGLBS มีค่า CR ต่ำที่สุด แต่อย่างไรก็ตาม อัลกอริทึมที่ได้ออกแบบไว้นั้น มีแนวโน้มของการทำงานที่คงที่ คือมีค่า CR เกือบเท่ากัน ในทุก ๆ ช่วงของข้อมูล

ข้อมูลกลุ่มที่ 3 และข้อมูลกลุ่มที่ 4 เป็นข้อมูลอุณหภูมิที่มีอัตราการสุ่มตัวอย่าง 1 ครั้ง ต่อ 4 ชั่วโมง และการสุ่มตัวอย่าง 1 ครั้ง ต่อ 8 ชั่วโมงตามลำดับ ซึ่งข้อมูลกลุ่มที่ 3 มีค่าเฉลี่ยโดยเฉลี่ยที่ข้อมูล 1000 ตัว ของข้อมูล 10 ชุด อยู่ที่ 2231 และมีค่าความแปรปรวนโดยเฉลี่ยที่ข้อมูล 1000 ตัว ของข้อมูล 10 ชุด อยู่ที่ 13500 ผลการทดสอบ จะแสดงไว้ในภาพประกอบที่ 5-7 ในขณะที่ข้อมูลกลุ่มที่ 4 จะมีค่าเฉลี่ยโดยเฉลี่ยที่ข้อมูล 1000 ตัว ของข้อมูล 10 ชุด อยู่ที่ 2270 และมีค่าความแปรปรวนโดยเฉลี่ยที่ข้อมูล 1000 ตัว ของข้อมูล 10 ชุด อยู่ที่ 18514 ซึ่งผลการทดสอบจะแสดงไว้ในภาพประกอบที่ 5-8



ภาพประกอบที่ 5-7 การทดสอบด้วยข้อมูลอุณหภูมิที่ความถี่ข้อมูล 1 รอบต่อ 4 ชั่วโมง



ภาพประกอบที่ 5-8 การทดสอบด้วยข้อมูลอุณหภูมิที่ความถี่ข้อมูล 1 รอบต่อ 8 ชั่วโมง

จากภาพประกอบที่ 5-7 และภาพประกอบที่ 5-8 จะสังเกตเห็นได้ว่า อัลกอริทึม Hybrid 002 มีประสิทธิภาพที่สูงกว่า CoXoH เล็กน้อย และจะมีความสามารถสูงกว่าอย่างชัดเจนเมื่อใช้กับข้อมูลในช่วงแรก แต่อย่างไรก็ตาม เมื่อข้อมูลมีปริมาณที่มากขึ้นจะพบว่าทั้ง Hybrid 002 และ CoXoH มีประสิทธิภาพที่ใกล้เคียงกัน

5.1.2 ข้อมูลรูปภาพ

ในการทดสอบนี้ จะทำการเปรียบเทียบอัตราการบีบอัดข้อมูลของแต่ละอัลกอริทึม เมื่อใช้กับการบีบอัดข้อมูลรูปภาพ โดยรูปภาพที่นำมาใช้ในการทดสอบนี้นั้นจะเป็นรูปภาพในรูปแบบ Bitmap (BMP format) ที่มีโทนสีขาวดำ (Gray scale picture) ซึ่งมีขนาด 256 × 256 จุด ใช้ในการทดสอบทั้งหมด 30 รูป ในการทดสอบจะทำการทดสอบกับอัลกอริทึมหลายรูปแบบ ซึ่งจะแบ่งอัลกอริทึมออกเป็น 9 กลุ่ม ด้วยกัน ได้แก่

- CoXoH
- DZ,SZ และ SZAVG แบบที่ไม่มีการจัดการกับ Register
- DZ,SZ และ SZAVG ที่ทำงานร่วมกันกับ LB ซึ่งสำรองข้อมูลไว้ 40 รอบ
- DZ,SZ และ SZAVG ที่ทำงานร่วมกันกับ LB ซึ่งสำรองข้อมูลไว้ 80 รอบ
- DZ,SZ และ SZAVG ที่ทำงานร่วมกันกับ LBS ที่ Register ขนาด 4 บิต
- DZ,SZ และ SZAVG ที่ทำงานร่วมกันกับ LBS ที่ Register ขนาด 5 บิต
- Hybrid 001 มีขนาด Buffer ของ SZAVGLB 80 รอบ และมี Buffer ในการทำ Hybrid ขนาด 4 และ 8 รอบ
- Hybrid 002 มีขนาด Register ของ SZAVGLBS 4 บิต และมี Buffer ในการทำ Hybrid ขนาด 4 และ 8 รอบ
- Hybrid 003 มีขนาด Register ของ SZAVGLBS 4 บิต และมี Buffer ในการทำ Hybrid ขนาด 4 และ 8 รอบ

ข้อมูลที่ถูกนำมาทดสอบนั้นจะมี 30 กลุ่ม แต่ละกลุ่มคือรูปภาพ 1 รูป โดยรูปภาพจะถูกแบ่งให้มีข้อมูลภายในไม่เกิน 1000 ตัว ทำให้ข้อมูล 1 กลุ่ม มีข้อมูล 67 ชุด โดย 66 ชุดแรกมีข้อมูล 1000 ตัว และชุดสุดท้ายจะมีข้อมูล 682 ตัว ข้อมูลแต่ละตัวจะมีขนาด 8 บิต จากลักษณะข้อมูลที่ได้กล่าวมานั้น ข้อมูลจะมีค่าเฉลี่ยโดยเฉลี่ยของข้อมูลอยู่ที่ 117 และมีค่าความแปรปรวนโดยเฉลี่ยอยู่ที่ 2240 ซึ่งจะเห็นได้ว่าค่าความแปรปรวนโดยเฉลี่ย จะมีความแตกต่างจากข้อมูลความชื้นอยู่พอสมควร ซึ่งผลการทดสอบนั้นจะแสดงในรูปแบบของตารางของค่าเฉลี่ยของ CR ดังแสดงในตารางที่ 5-1

ตารางที่ 5-1 ประสิทธิภาพในการเข้ารหัสข้อมูลรูปภาพ

Algorithm		AVG CR
CoXoH		18.83
original	DZ	21.02
	SZ	19.91
	SZAVG	22.83
LB Buffer = 40	DZ	21.1
	SZ	20.48
	SZAVG	23.51
LB Buffer = 80	DZ	20.48
	SZ	19.75
	SZAVG	22.76
LBS Register = 4 bit	DZ	22.18
	SZ	21.66
	SZAVG	24.4
LBS Register = 5 bit	DZ	21.75
	SZ	20.8
	SZAVG	23.71
Hybrid 001	Buffer = 4,80	26.26
	Buffer = 8,80	26.74
Hybrid 002	Buffer = 4,4	26.57
	Buffer = 8,4	27.1
Hybrid 003	Buffer = 4,4	26.1
	Buffer = 8,4	26.82

จากผลลัพธ์ข้างต้น ช่องที่มีสีเขียวเข้มหมายถึงมีค่า CR สูงที่สุด ช่องที่มีสีเขียวอ่อนหมายถึงมีค่า CR สูงใน 5 อันดับรองลงมา ช่องที่มีสีส้มเข้ม หมายถึง มีค่า CR ต่ำที่สุด และช่องที่มีสีส้มอ่อน หมายถึง มีค่า CR น้อย 5 อันดับรองลงมา

จากตารางที่ 5-1 จะสังเกตได้ว่าอัลกอริทึม Hybrid 002 รูปแบบที่มีการสำรองข้อมูล 8 รอบ จะมีค่า CR สูงที่สุด และอัลกอริทึมที่ค่า CR สูงรองลงมาจะเป็นอัลกอริทึมประเภท Hybrid โดยในส่วนของอัลกอริทึมที่มีความสามารถในการเข้ารหัสต่ำที่สุดคือ CoXoH ในขณะที่อัลกอริทึมที่ได้ออกแบบไว้นั้นจะมีประสิทธิภาพสูงกว่า CoXoH เล็กน้อย โดยจะสังเกตได้ว่า SZAVG จะมีค่าสูงกว่า DZ และ SZ เล็กน้อย แต่อย่างไรก็ตาม ความสามารถในการบีบอัดข้อมูลของแต่ละอัลกอริทึม ไม่ได้มีค่าที่แตกต่างกันอย่างชัดเจน

5.1.3 ข้อมูลคลื่นไฟฟ้าสมอง (EEG)

การทดสอบในหัวข้อนี้ จะใช้ข้อมูลคลื่นไฟฟ้าสมอง (EEG) มาทำการทดสอบ [17] ข้อมูลดังกล่าวเป็นคลื่นไฟฟ้าสมองที่มีอัตราการสุ่มตัวอย่าง (Sampling rate) ที่ 173.61 Hz โดยจะมีข้อมูลทั้งหมด 5 กลุ่ม แต่ละกลุ่มจะมีข้อมูลทั้งหมด 99 ชุด ทำการเรียบเรียงใหม่ให้มีทั้งหมด 369 ชุด ทำให้แต่ละชุดมีข้อมูล 1000 ตัว ข้อมูลแต่ละตัวจะมีขนาด 12 บิต ข้อมูลแต่ละกลุ่มจะมีค่าเฉลี่ย

โดยเฉลี่ยอยู่ที่ F = 2042, N = 2300, O = 2337, S = 2142 และ Z = 2239 มีค่าความแปรปรวนโดยเฉลี่ยที่ F = 7667, N = 3453114, O = 3377532, S = 3377452 และ Z = 3412990 ในส่วนของอัลกอริทึมที่ใช้ในการทดสอบนั้น จะเหมือนกันกับอัลกอริทึมที่ทดสอบด้วยข้อมูลรูปภาพ

ตารางที่ 5-2 การทดสอบความสามารถในการเข้ารหัสโดยใช้ข้อมูล EEG

Algorithm		**AVG CR**	**Compression Ratio**				
			F	**N**	**O**	**S**	**Z**
CoXoH		27.48	37.32	42.3	26.64	-5.21	36.35
original	DZ	30.11	40.67	36.11	25.76	17.98	30.04
	SZ	28.62	37.62	33.27	25.58	17.57	29.08
	SZAVG	31.3	38.16	38.31	28.13	19.14	32.77
LB Buffer = 40	DZ	29.68	39.84	35.3	25.72	17.53	30.03
	SZ	27.73	37.39	32.43	24.33	17.12	27.39
	SZAVG	30.85	38.32	37.26	27.77	18.9	32.01
LB Buffer = 80	DZ	29.94	40.29	35.78	25.76	17.7	30.18
	SZ	28.21	37.7	32.96	24.9	17.23	28.24
	SZAVG	31.11	38.37	37.81	27.97	18.99	32.41
LBS Register = 4 bit	DZ	29.91	39.97	35.3	25.97	18.32	30
	SZ	28.31	38.23	33.1	24.64	17.83	27.76
	SZAVG	30.81	39.1	36.62	27.5	19.22	31.62
LBS Register = 5 bit	DZ	30.09	40.26	35.72	26.05	18.12	30.29
	SZ	28.6	38.19	33.38	25.23	17.62	28.59
	SZAVG	31.2	38.93	37.53	27.97	19.21	32.36
Hybrid 001	Buffer = 4,80	34.82	40.51	44.43	32.29	17.49	39.38
	Buffer = 8,80	35.23	41.31	44.75	32.21	18.46	39.44
Hybrid 002	Buffer = 4,4	34.79	40.95	44.15	32.08	17.64	39.13
	Buffer = 8,4	35.23	41.78	44.53	32.05	18.56	39.25
Hybrid 003	Buffer = 4,4	34.59	40.76	44.04	32.02	17.06	39.07
	Buffer = 8,4	35.15	41.7	44.49	32.03	18.29	39.22

จาก ตารางที่ 5-2 ช่องสีเขียวเข้ม คือช่องที่มีค่า CR มากที่สุด ช่องสีเขียวอ่อน คือช่องที่มีค่า CR มาก 5 อันดับรองลงมา ช่องสีส้มเข้ม คือช่องที่มีค่า CR น้อยสุด และช่องสีส้มอ่อน คือช่องที่มีค่า CR น้อย 5 อันดับรองลงมา จากตารางที่ 5-2 จะสังเกตได้ว่า อัลกอริทึม Hybrid 001 แบบที่ใช้ Buffer 8 รอบ และ Hybrid 002 แบบที่ใช้ Buffer 8 รอบ มีค่า CR สูงที่สุด ส่วนอัลกอริทึมแบบ Hybrid แบบอื่น ๆ จะมีค่า CR สูงรองลงมา ตามมาด้วย DZ, SZ และ SZAVG แบบต่าง ๆ และ CoXoH จะมีค่า CR ที่น้อยที่สุด แต่อย่างไรก็ตาม ค่า CR ที่ได้ จะมีความแตกต่างกันไม่มากนัก

5.1.4 สรุปผลการทดสอบ

การทดสอบความสามารถในการเข้ารหัสของอัลกอริทึมในรูปแบบต่าง ๆ นั้น จะสังเกตได้ว่า เมื่อข้อมูลมีลักษณะที่แตกต่างกัน จะทำให้ความสามารถในการบีบอัดข้อมูลแตกต่างออกไปด้วย และแต่ละอัลกอริทึมจะมีความสามารถในการเข้ารหัสข้อมูลในแต่ละรูปแบบที่ต่างกัน เช่น CoXoH จะมีความสามารถในการบีบอัดข้อมูลประเภทสภาพอากาศได้ดีกว่า ในขณะที่ SZAVG จะมีความสามารถในการบีบอัดข้อมูลรูปภาพ และคลื่นไฟฟ้าสมองได้ดีกว่า

แต่อย่างไรก็ตาม CoXoH มีข้อด้อยในเรื่องของการบีบอัดข้อมูลบางประเภท เช่น ข้อมูลที่มีการซ้ำกันน้อย ซึ่งอาจทำให้ผลลัพธ์ในการบีบอัดข้อมูลนั้นมีขนาดที่ใหญ่กว่าข้อมูลต้นฉบับ รวมทั้งประสิทธิภาพในการทำงานของ CoXoH ที่จะค่อย ๆ เพิ่มขึ้นเมื่อทำการเข้ารหัสข้อมูลที่มากขึ้น ซึ่งอาจทำให้เกิดปัญหาขึ้นในระบบ ในทางกลับกัน DZ, SZ และ SZAVG จะมีความสามารถในการเข้ารหัสข้อมูลที่คงที่ตลอดการทำงาน จึงเป็นที่มาของการออกแบบอัลกอริทึมแบบ Hybrid ที่สามารถสลับเอาจุดเด่นของแต่ละอัลกอริทึมออกมาใช้งานได้ตรงจังหวะ ทำให้มีประสิทธิภาพที่สูงกว่าอัลกอริทึมอื่น

ในส่วนของการทดสอบประสิทธิภาพในการเข้ารหัสข้อมูลนั้น แม้ว่าอัลกอริทึมแบบ DZ, SZ และ SZAVG จะมีประสิทธิภาพไม่สูงมากนัก แต่จะสังเกตได้ว่าอัลกอริทึมดังกล่าวมีขั้นตอนการทำงานที่ชัดเจน และไม่ซับซ้อน ซึ่งน่าจะเป็นผลให้วงจรที่จะสร้างขึ้นมา มีขนาดเล็ก โดยผลลัพธ์ในส่วนนี้ จะทำการทดสอบในหัวข้อถัดไป

5.2 การทดสอบปริมาณทรัพยากรฮาร์ดแวร์ และพลังงาน

ในการทดสอบปริมาณการใช้งานทรัพยากรฮาร์ดแวร์ จะเลือก SZAVGLBS มาทำการทดสอบ เนื่องจากเป็นอัลกอริทึมที่น่าจะใช้ปริมาณทรัพยากรมากที่สุด ในการทดสอบจะทำการจำลองวงจรของ SZAVGLBS ในส่วนของตัวเข้ารหัส ลงบน FPGA อ้างอิงกับ FPGA รุ่น Spartan 6 LX9 MicroBoard เพื่อให้ทราบถึงปริมาณทรัพยากรที่ใช้ และเปรียบเทียบปริมาณทรัพยากรกับ Huffman algorithm [15] โดยทรัพยากรที่ถูกนำมาพิจารณาจะมีหลากหลายประเภท ได้แก่

- Slice Register คือ Register หรือกลุ่มของ Flip-Flop ที่ใช้เพื่อเก็บข้อมูล
- Slice LUTs คือ กลุ่มของ Logic gate ที่ถูกผูกรวมกัน เพื่อให้ง่ายต่อการใช้งาน
- Fully used LUT-FF pairs เป็นสายที่เชื่อมต่อ LUT กับ Flip-Flop
- Bounded IOBs คือ Buffer ของอินพุต และเอาต์พุต
- BUFG/BUFGCTRLs หรือ Global clock buffer เป็นสัญญาณนาฬิกากลางของระบบ
- DSP (Data Signal Processing) เป็นส่วนที่ใช้ในการคำนวณทางคณิตศาสตร์ เช่น การคูณ หรือการหาร
- BRAM หรือ Block RAM เป็นส่วนของหน่วยความจำของระบบ ใช้ในการเก็บข้อมูล

การทดสอบจะทำการเปรียบเทียบในรูปแบบของตาราง เนื่องจาก SZAVGLBS ถูกอ้างอิงกับ Spartan 6 LX9 MicroBoard ในขณะที่ Huffman algorithm ถูกอ้างอิงอยู่บน Vertex 5

ML507 ซึ่งเป็น FPGA คนละรุ่นกัน และทรัพยากรแต่ละประเภทก็มีปริมาณการใช้งานที่แตกต่างกันค่อนข้างมาก

ตารางที่ 5-3 ปริมาณทรัพยากรฮาร์ดแวร์ของ SZAVGLBS เทียบกับ Huffman algorithm

Algorithm	Frequency	LUTs	Register	DSP	BRAMs	Bound IOBs
SZAVGLBS	66.7 MHz	488	98	4	0	30
Huffman	100 MHz	2396	910	0	3+1+4	N/A

จากตารางที่ 5-3 SZAVGLBS จะใช้ทรัพยากรที่น้อยกว่าอย่างชัดเจน โดยเฉพาะส่วนของ LUTs, Register และ BRAM แต่จะมีการใช้งาน DSP 4 ตัว ในขณะที่ Huffman algorithm จะไม่มีการใช้งานในส่วนของ DSP อย่างไรก็ตามแม้ว่า Huffman algorithm จะมีการใช้งานทรัพยากรที่มากกว่า นั้นก็อาจเป็นผลมาจากลักษณะการทำงานของตัวอัลกอริทึมด้วยส่วนหนึ่ง คือ Huffman algorithm จะทำการบีบอัดข้อมูลรอบละ 4KB ทำให้จำเป็นต้องมีการใช้งาน BRAM จำนวนมากในการเก็บข้อมูล ในขณะที่ SZAVGLBS จะทำการบีบอัดข้อมูลครั้งละ 1 Byte เท่านั้น จึงไม่จำเป็นต้องมีการใช้งาน BRAM

การคำนวณหาพลังงานที่ถูกใช้ในวงจร จะใช้การจำลองด้วย XPE (Xilinx Power Estimator) รุ่นที่ 14.3 โปรแกรมนี้ จะใช้ข้อมูลที่ได้จากการสังเคราะห์วงจร มาทำการคำนวณว่าระบบจะต้องเสียพลังงานไปกับส่วนใดบ้าง โดยพลังงานจะถูกแบ่งออกเป็น 3 ส่วนหลัก ได้แก่

- Device Static เป็นส่วนของพลังงานที่ตัว FPGA จะต้องใช้เพื่อหล่อเลี้ยงวงจรของตัวมันเองเมื่อเปิดการทำงาน พลังงานส่วนนี้จะถูกใช้อย่างคงที่ ไม่ว่ารูปแบบของวงจร จะเป็นลักษณะใด
- Core Dynamic เป็นส่วนของพลังงานที่เกิดจากการใช้ทรัพยากรภายใน FPGA เช่น Register, LUTs, DSP หรือ BRAM โดยปริมาณของพลังงานในส่วนนี้จะขึ้นอยู่กับปริมาณทรัพยากรที่ถูกใช้สังเคราะห์วงจร
- IO พลังงานในส่วนนี้ เกิดจากการใช้งานอินพุต และเอาต์พุตของตัว FPGA ดังนั้น ปริมาณพลังงานในส่วนนี้ จะขึ้นอยู่กับจำนวนขาสัญญาณอินพุต และเอาต์พุตของวงจร

จาก 3 ส่วนที่ได้กล่าวมา จะเห็นได้ว่าพลังงานในส่วนของ Device Static เป็นส่วนที่ไม่สามารถเข้าไปจัดการได้ และจะมีอัตราการบริโภคพลังงานที่คงที่ ในขณะที่อีก 2 ส่วนจะขึ้นอยู่กับขนาดของวงจรที่ได้ออกแบบ ผลการทดลองที่จะนำเสนอต่อไปนี้ จะแบ่งออกเป็น 2 ส่วน คือ วงจรเข้ารหัส และวงจรถอดรหัส ดังตารางที่ 5-4 และ ตารางที่ 5-5 ตามลำดับ

ตารางที่ 5-4 พลังงานในวงจรเข้ารหัสของ SZAVGLBS

Resource		Power	
		(W)	(%)
Core Dynamic	Clock	0.002	5.56
	Logic	0.002	5.56
	BRAM	0.000	0.00
	DSP	0.006	16.67
I/O	IO	0.012	33.33
Device Static		0.014	38.89
Total		0.036	100.00

ตารางที่ 5-5 พลังงานในวงจรถอดรหัสของ SZAVGLBS

Resource		Power	
		(W)	(%)
Core Dynamic	Clock	0.002	5.88
	Logic	0.002	5.88
	BRAM	0.000	0.00
	DSP	0.006	17.65
I/O	IO	0.010	29.41
Device Static		0.014	41.18
Total		0.034	100.00

จากตารางที่ 5-4 และตารางที่ 5-5 จะสังเกตได้ว่า พลังงานที่ถูกใช้ในวงจรเข้ารหัสและวงจรถอดรหัส มีปริมาณน้อยมาก โดยใช้พลังงานประมาณ 36mW และ 34mW ตามลำดับ จากพลังงานรวมของวงจรพบว่า สูญเสียพลังงานไปในส่วนของ Device Static ค่อนข้างมาก เนื่องจากเทคโนโลยีในการสร้าง FPGA ในปัจจุบันจะมีขนาดโพรเซสเทคโนโลยีที่เล็กเช่น 0.18 ไมโครอน หรือต่ำกว่า จะทำให้มีค่าพลังงานแบบ Static มาก ซึ่งเป็นคุณสมบัติของพลังงานที่สูญเสียในการทำงานของทรานซิสเตอร์แบบ CMOS ดังนั้น ถ้าหากนำวงจรดังกล่าวไปสังเคราะห์เป็นวงจรรวม หรือ IC จะทำให้ใช้พลังงานที่น้อยลงได้อีก

บทที่ 6
สรุปผลการวิจัยและแนวทางการพัฒนาต่อ

ในบทนี้ จะทำการสรุปผลการวิจัยที่ได้กล่าวไว้แล้วในบทก่อนหน้า โดยการสรุปจะกล่าวถึงความสามารถโดยรวมของอัลกอริทึมแต่ละรูปแบบ ข้อดีข้อด้อย รวมไปถึงแนวทางในการนำอัลกอริทึมที่ได้ออกแบบไว้ไปพัฒนาต่อ โดยจะกล่าวถึงประเด็นที่น่าสนใจ ซึ่งสามารถนำไปพัฒนาต่อได้

6.1 สรุปผลการวิจัย

ในงานวิจัยชิ้นนี้มีวัตถุประสงค์อยู่ 2 ประการด้วยกัน คือ การสร้างอัลกอริทึมสำหรับการเข้ารหัส และการถอดรหัสข้อมูล โดยมีวัตถุประสงค์เพื่อลดปริมาณการส่งข้อมูลในระบบเครือข่ายเซนเซอร์ไร้สาย และเพื่อทดสอบปริมาณทรัพยากรฮาร์ดแวร์เมื่อมีการนำอัลกอริทึมดังกล่าว มาทำการสังเคราะห์เป็นวงจร จากวัตถุประสงค์ที่ได้กล่าวมานั้น จำเป็นต้องทราบถึงลักษณะของข้อมูลบนระบบเครือข่ายเซนเซอร์ไร้สาย และวิธีบีบอัดข้อมูลในรูปแบบต่าง ๆ โดยจะมุ่งเน้นที่จะออกแบบระบบ ให้มีความสามารถในการเข้ารหัสข้อมูลหลากหลายประเภท เช่น ข้อมูลสภาพอากาศ ข้อมูลทางการแพทย์ ข้อมูลภาพถ่าย เป็นต้น โดยการเข้ารหัสที่ได้ออกแบบจะต้องเกิด Overhead น้อยที่สุด เพื่อไม่ให้มีการสร้างภาระเพิ่มเติมให้กับระบบ

จากที่มาข้างต้น ภายในงานวิจัยชิ้นนี้จึงได้มีการออกแบบอัลกอริทึมขึ้นมา ได้แก่ DZ, SZ และ SZAVG ซึ่งทั้ง 3 อัลกอริทึมนั้น จะมีลักษณะการทำงานที่ใกล้เคียงกัน โดยจะมีความสามารถในการลดค่า Overhead ที่เกิดจากการเข้ารหัส และยังสามารถส่งข้อมูลของระบบได้อย่างต่อเนื่อง โดยไม่มีการเก็บข้อมูลค้างเอาไว้ในระบบเป็นเวลานาน แต่อย่างไรก็ตาม เนื่องจากอัลกอริทึมดังกล่าว ถูกออกแบบในลักษณะของอัลกอริทึม เมื่อต้องการจะนำไปใช้งานจริง จำเป็นจะต้องมีการพัฒนาเพิ่มเติม ในส่วนของการจำกัดขนาดของ Register โดยในส่วนนี้ ได้สร้างอัลกอริทึม LB และ LBS ขึ้นมา เพื่อใช้ในการจัดการกับปัญหาของ Register ที่ใช้เก็บข้อมูลทางสถิติของอัลกอริทึมที่ออกแบบไว้ในตอนแรก แต่อย่างไรก็ตาม ความสามารถในการบีบอัดข้อมูลของอัลกอริทึมทั้ง 3 มีค่าไม่สูงมากนัก เมื่อเปรียบเทียบกับ CoXoH จึงได้มีการคิดอัลกอริทึมแบบ Hybrid เพื่อผนวกความสามารถของอัลกอริทึมทั้งสองกลุ่ม ทำให้มีประสิทธิภาพที่ดีที่สุด เมื่อทำการทดสอบอัลกอริทึมทั้ง 3 รูปแบบพร้อมกัน จะพบว่า DZ,SZ และ SZAVG นั้น ทั้งแบบที่ทำงานร่วมกันกับ LB และ LBS จะมีความสามารถในการเข้ารหัสที่ค่อนข้างคงที่ กล่าวคือ มีประสิทธิภาพเท่ากัน ไม่ว่าปริมาณข้อมูลจะมี 100 ตัว 200 ตัว หรือมากขึ้น ก็จะมีอัตราการบีบอัดที่ไม่แตกต่างไปจากเดิมมากนัก ในขณะที่ CoXoH จะมีอัตราการบีบอัดข้อมูลที่ไม่สูงมากนักในช่วงแรกที่เพิ่งเริ่มทำการเข้ารหัสข้อมูล แต่เมื่อมีข้อมูลทางสถิติในระบบมากขึ้น จะทำให้มีอัตราการบีบอัดข้อมูลที่สูงมากขึ้นด้วย แต่อัลกอริทึมแบบ CoXoH นั้น เหมาะจะใช้ในการบีบอัดข้อมูลที่มีการซ้ำกันมาก ๆ ดังนั้น เมื่อบีบอัดข้อมูลประเภทสภาพอากาศ จะทำให้มีประสิทธิภาพสูงกว่า DZ, SZ และ SZAVG แต่เมื่อนำมาบีบอัดข้อมูลรูปภาพ หรือข้อมูลคลื่นไฟฟ้าสมอง พบว่ามีอัตรา

การบีบอัดข้อมูลต่ำกว่า DZ, SZ และ SZAVG และถ้าหากเปรียบเทียบระยะเวลาในการทำงานโดยเทียบแบบ O(n) แล้ว จะพบว่า DZ, SZ และ SZAVG จะเป็นแบบ O(k) คือ มีขั้นตอนการทำงานที่คงที่ ในขณะที่ CoXoH นั้นจะเป็นแบบ O(n log(n)) โดยที่ n คือจำนวนลักษณะของข้อมูลที่มีอยู่ในระบบ หรือสามารถกล่าวได้ว่า ถ้าหากมีข้อมูลทางสถิติในระบบมากขึ้น CoXoH จะใช้เวลาในการทำงานที่มากขึ้นด้วย ในส่วนของอัลกอริทึมแบบ Hybrid นั้น มีความสามารถในการรวมเอาข้อดีของทั้ง SZAVG กับ CoXoH มาไว้ด้วยกัน ทำให้มีอัตราการบีบอัดข้อมูลที่สูงกว่า เมื่อเทียบกับอัลกอริทึมสองกลุ่มข้างต้น แต่ก็จะมีขนาดของอัลกอริทึมที่ใหญ่กว่าเช่นกัน ในการทดสอบส่วนฮาร์ดแวร์นั้น ได้มีการจำลองการทำงานของ SZAVGLBS เพื่อทดสอบปริมาณการใช้งานทรัพยากรฮาร์ดแวร์ ซึ่ง SZAVGLBS ใช้ทรัพยากรฮาร์ดแวร์น้อยกว่า Huffman algorithm ประมาณ 5 เท่า ในส่วนของ Slice LUTs และน้อยกว่า 9 เท่า ในส่วนของ Register นอกจากการทดสอบปริมาณฮาร์ดแวร์แล้ว ยังมีส่วนของการทดสอบปริมาณการใช้พลังงาน โดการใช้โปรแกรม XPE รุ่น 14.3 ผลปรากฏว่า ส่วนเข้ารหัสใช้พลังงาน 36 mW และส่วนถอดรหัสใช้พลังงาน 34 mW ซึ่งสามารถลดลงได้อีก เนื่องจากพลังงานดังกล่าว เป็นพลังงานรวมของ FPGA ทั้งตัว

แม้ว่า DZ, SZ และ SZAVG จะมีอัตราการบีบอัดข้อมูลที่ไม่สูงมากนัก แต่เมื่อพิจารณาถึงองค์ประกอบอื่น ๆ แล้ว จะพบว่า DZ, SZ และ SZAVG ก็จะมีจุดเด่นที่เหนือกว่า CoXoH ในหลายด้าน ทั้งในเรื่องของการสร้าง Overhead น้อยกว่า สามารถบีบอัดข้อมูลที่ซ้ำกันน้อยได้ และยังมีการทำงานที่ซับซ้อนน้อยกว่า ทำให้ DZ, SZ และ SZAVG เหมาะสมกับระบบเครือข่ายเซนเซอร์ไร้สาย มากกว่า CoXoH ในหลายงานประยุกต์ แต่อย่างไรก็ตาม DZ, SZ และ SZAVG ก็ยังคงต้องมีการพัฒนาต่อไป เพื่อให้มีประสิทธิภาพในการทำงานที่สูงยิ่งขึ้น

6.2 แนวทางในการพัฒนาต่อ

อัลกอริทึมที่ได้ออกแบบไว้นั้น ยังมีจุดอ่อนให้สามารถพัฒนาต่อได้อีกหลายจุด เช่นการพัฒนาความสามารถในการบีบอัด และการรองรับกับสถานการณ์ในการนำไปใช้งานจริง ทำให้ DZ, SZ และ SZAVG จะต้องมีการพัฒนาต่อ โดยแนวทางในการพัฒนาต่อในแง่มุมของผู้วิจัยสามารถแบ่งออกเป็น 2 กลุ่ม คือ การพัฒนาประสิทธิภาพในการบีบอัดข้อมูล และการจัดการกับสถานการณ์ในการใช้งานจริง

6.2.1 การพัฒนาประสิทธิภาพในการบีบอัดข้อมูล

ในส่วนของการพัฒนาประสิทธิภาพในการบีบอัดข้อมูลนั้น DZ, SZ และ SZAVG จะมีจุดเด่นในเรื่องของการสร้าง Overhead ที่มีประมาณน้อย แต่เมื่อนำไปบีบอัดข้อมูลที่ซ้ำกันมาก จะมีประสิทธิภาพต่ำกว่าอัลกอริทึมอื่น ซึ่งในจุดนี้ เป็นข้อด้อยที่แก้ได้ยาก เนื่องจาก DZ, SZ และ SZAVG ถูกออกแบบให้มีความซับซ้อนต่ำ ทำให้มีประสิทธิภาพน้อยตามไปด้วย แต่อย่างไรก็ตาม การพัฒนาจุดเด่นของ DZ, SZ และ SZAVG จะเป็นประโยชน์กับภาพรวมของระบบมากกว่า คือ การสร้าง Overhead ที่น้อย ทำให้ระบบไม่ต้องรับภาระจากปริมาณข้อมูลที่มากขึ้น เมื่อข้อมูลไม่เหมาะสมกับการบีบอัด ซึ่งในส่วนนี้ จะทำให้ผู้ใช้งานมั่นใจได้ว่าอัลกอริทึมที่นำมาใช้งานนั้นจะเป็นประโยชน์ในการทำงานอย่างแน่นอน

6.2.2 การจัดการกับสถานการณ์ในการใช้งานจริง

ในสถานการณ์ในการนำอัลกอริทึมไปใช้งานจริงนั้น ยังต้องคำนึงถึงรายละเอียดอีกจำนวนหนึ่ง ซึ่งประเด็นที่สำคัญก็คือ ความสามารถในการแทรกคำสั่งในขณะที่ทำการส่งข้อมูล กล่าวคือ ข้อมูลที่ถูกส่งออกไป หรือรับเข้ามานั้น อาจจะประกอบไปด้วย ข้อมูล หรือคำสั่ง ซึ่งตัวถอดรหัสต้องสามารถแยกสองกลุ่มนี้ออกจากกันได้ การส่งคำสั่งออกไปได้ จะทำให้การจัดการกับข้อมูลมีความหลากหลายมากขึ้น และรองรับกับการทำงานจริงในสถานะการฉุกเฉินได้ เช่น กรณีของการสูญเสียพลังงานขณะหนึ่ง หรือกรณีไฟดับ เนื่องจากอัลกอริทึมที่ออกแบบนั้น จะทำการสร้างฐานข้อมูลขึ้นทั้งฝั่งตัวเข้ารหัส และตัวถอดรหัส เมื่อเกิดเหตุไฟดับ อาจจะทำให้ระบบฝั่งใดฝั่งหนึ่งสูญเสียข้อมูลทั้งหมดไป และไม่สามารถสื่อสารกับฝั่งตรงข้ามได้ ทำให้ข้อมูลเกิดการผิดพลาดขึ้น ซึ่งถ้าหากมีการใช้งานคำสั่งของระบบก็จะสามารถส่งสัญญาณเพื่อขอข้อมูลจากฝั่งตรงข้าม หรือสั่งให้ฝั่งตรงข้ามทำการเริ่มการใช้งานระบบขึ้นมาใหม่ ทำให้ระบบมีฐานข้อมูลที่ตรงกัน และสามารถทำงานได้อย่างต่อเนื่อง แต่อย่างไรก็ตาม การที่จะแทรกคำสั่งปะปนไปกับข้อมูลนั้น จะต้องมีการออกแบบที่ดีในระดับหนึ่ง เพื่อให้ทั้งตัวถอดรหัส และตัวเข้ารหัส สามารถเข้าใจคำสั่งได้ตรงกัน ในทุกสถานการณ์ กล่าวคือ ข้อมูลเดียวกัน แต่ถูกส่งมาในเวลาที่แตกต่างกัน จะทำให้มีความหมายที่แตกต่างกัน ดังนั้น จำเป็นจะต้องมีรูปแบบของข้อมูลที่สามารถเข้าใจตรงกันในทุกสภาวะ ในส่วนของการส่งคำสั่งแทรกไปกับข้อมูล อัลกอริทึมที่ได้ออกแบบไว้นั้น ถูกออกแบบเพื่อรองรับความสามารถในส่วนนี้เอาไว้บ้างแล้ว ทำให้สามารถเอาไปพัฒนาต่อได้ไม่ยาก

บรรณานุกรม

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey", *Computer Networks*, vol. 38, no. 4, pp. 393–422, Mar. 2002.

[2] S. I. Hussian, H. Javed, W. ur Rehman, and F. N. Khalil, "CoXoH: Low cost energy efficient data compression for wireless sensor nodes using data encoding", *International Conference on Computer Networks and Information Technology (ICCNIT)*, pp. 149 –152, Jul. 2011.

[3] C. Tharini and P. V. Ranjan, "Design of Modified Adaptive Huffman Data Compression Algorithm for Wireless Sensor Network", *Journal of Computer Science*, vol. 5, no. 6, pp. 466–470, Jun. 2009.

[4] D. I. Sacaleanu, R. Stoian, and D. M. Ofrim, "An adaptive Huffman algorithm for data compression in wireless sensor networks", *Circuits and Systems (ISSCS), 2011 10th International Symposium*, pp. 1 –4, Jul. 2011.

[5] M. A. Soto Hernandez, O. Alvarado-Nava, E. Rodriguez-Martinez, and F. J. Zaragoza Martinez, "Tree-less Huffman coding algorithm for embedded systems", *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2013, pp. 1–6.

[6] J. Matai, J. Kim, and R. Kastner, "Energy efficient canonical huffman encoding", *IEEE 25th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2014, pp. 202–209.

[7] S. Rigler, W. Bishop, and A. Kennings, "FPGA-Based Lossless Data Compression using Huffman and LZ77 Algorithms", in *Electrical and Computer Engineering, 2007. CCECE 2007. Canadian Conference on*, pp. 1235 –1238, Apr. 2007.

[8] M.-B. Lin, J.-F. Lee, and G. E. Jan, "A Lossless Data Compression and Decompression Algorithm and Its Hardware Architecture", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 9, pp. 925 –936, Sep. 2006.

[9] F. Marcelloni and M. Vecchio, "A Simple Algorithm for Data Compression in Wireless Sensor Networks", *IEEE Communications Letters*, vol. 12, no. 6, pp. 411 –413, Jun. 2008.

[10] Y. Liang and Y. Li, "An Efficient and Robust Data Compression Algorithm in Wireless Sensor Networks", *IEEE Communications Letters*, vol. 18, no. 3, pp. 439–442, Mar. 2014.

[11] Y. Wang, X. Li, X. Wu, Z. Zhou, and H. Chen, "Horizontal hierarchy slicing based data compression for WSN", *IEEE Computing, Communications and IT Applications Conference (ComComAp)*, pp. 255–260, 2014.

[12] J. Ko, C. Lu, M. B. Srivastava, J. A. Stankovic, A. Terzis, and M. Welsh, "Wireless Sensor Networks for Healthcare", *Proceedings of the IEEE*, Nov. 2010.

[13] I. M. Pu, *Fundamental Data Compression*. Butterworth-Heinemann, 2005.

[14] L.-T. Wang, C.-W. Wu, and X. Wen, *VLSI Test Principles and Architectures: Design for Testability*. Academic Press, 2006.

[15] B. Ying, W. Liu, Y. Liu, H. Yang, and H. Wang, "Energy-efficient node-level compression arbitration for wireless sensor networks", *11th International Conference on Advanced Communication Technology, ICACT 2009*, pp.564-568, Feb. 2009.

[16] "Weather Forecast & Reports - Long Range & Local | Wunderground| Weather Underground." [Online]. Available: http://www.wunderground.com. [Accessed: 10-Feb-2014].

[17] "EEG time series." [Online]. Available: http://epileptologie-bonn.de/cms/front_content.php?idcat=193&lang=3&changelang=3. [Accessed: 16-Sep-2014].

ภาคผนวก

ภาคผนวก ก
ผลการทดสอบเพิ่มเติม

1. การทดสอบประสิทธิภาพในการบีบอัดข้อมูล

นอกเหนือจากผลการทดสอบในบทที่ 5 แล้ว ยังมีการทดสอบเพิ่มเติมอยู่อีก สามารถแบ่งออกเป็น 2 ส่วน คือ การทดสอบกับข้อมูลรูปภาพ และการทดสอบกับ Adaptive Huffman algorithm

1.1 การทดสอบกับข้อมูลรูปภาพ

แม้ว่าในบทที่ 5 จะมีส่วนของการทดสอบกับข้อมูลรูปภาพประกอบอยู่ด้วย แต่ก็เป็นส่วนย่อเท่านั้น เพื่อให้ง่ายต่อการทำความเข้าใจ โดยตารางที่จะนำเสนอต่อไปนี้เป็นผลลัพธ์ตัวเต็มโดยตารางที่ ก - 1 เป็นตารางแสดงค่าเฉลี่ยของการเข้ารหัสรวมทั้ง 30 รูป

ตารางที่ ก - 1 ค่า CR เฉลี่ยของแต่ละอัลกอริทึม ในการเข้ารหัสรูปภาพ 30 รูป

Algorithm		**AVG CR**
CoXoH		18.83
original	DZ	21.02
	SZ	19.91
	SZAVG	22.83
LMB 40	DZ	21.1
	SZ	20.48
	SZAVG	23.51
LMB 80	DZ	20.48
	SZ	19.75
	SZAVG	22.76
LBS 4	DZ	22.18
	SZ	21.66
	SZAVG	24.4
LBS 5	DZ	21.75
	SZ	20.8
	SZAVG	23.71
HB_001	B4_80	26.26
	B8_80	26.74
HB_002	B4_4	26.57
	B8_4	27.1
HB_003	B4_4	26.1
	B8_4	26.82

ตารางที่ ก - 2 ผลการเข้ารหัสรูปภาพ 30 รูป

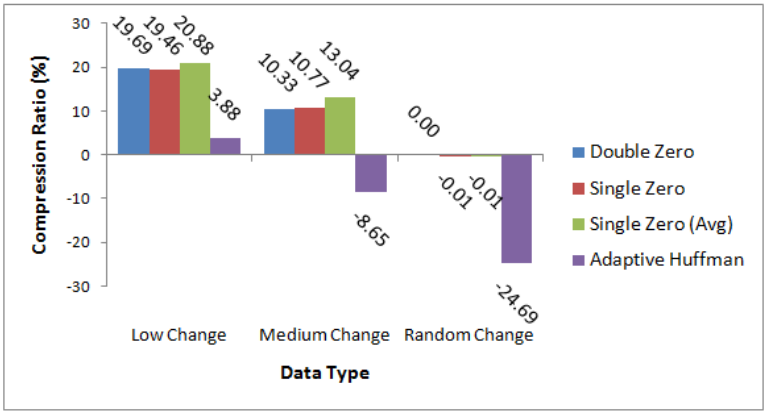
Algorithm		Compression Ratio																												
CoXoH		28.47	10.64	14.95	-5.23	26.41	17.37	-1.79	17.88	26.03	35.39	23.36	11.75	34.22	22.9	13.73	14.84	23.22	21.32	10.7	19.68	39.37	14.77	11.64	19.93	27.05	13.31	15.94	16.31	22.83	17.76
original	DZ	23.16	17.69	19.96	5.33	24.13	16.99	4.94	22.87	26.99	31.97	21.6	16.29	29.36	24.33	18.8	18.63	24.08	23.39	17.65	24.42	35.68	19.9	17.97	22.81	27.09	13.23	19.07	20.35	22.17	19.85
	SZ	21.88	15.58	17.48	7.15	21.42	17.79	7.07	18.88	25.63	29.1	18.89	16.88	29.15	21.02	15.2	17.82	24.8	24.63	17.67	22.15	34.1	19.55	17.1	21.08	25.71	13.15	18.08	18.58	21.49	18.12
	SZAVG	27.94	18.07	20.6	8.24	26.87	22.05	10.15	23.16	26.99	33.51	24.71	18.56	30.86	25.64	19.27	21.16	25.83	25.09	17.88	24.76	33.7	20.56	19.56	24.93	26.99	18.91	21.02	21.11	24.45	22.18
LMB 40	DZ	24.11	15.99	19.81	5.33	24.55	18.58	7.61	21.35	26.29	31.86	23.4	15.11	29.5	24.11	17.88	17.89	23.67	22.85	16.48	23.38	36.35	18.44	17.63	22.76	26.94	17.71	19.26	20.81	23.56	19.65
	SZ	23.66	14.82	17.39	6.28	24.08	18.81	8.46	19.43	26.22	28.77	23.34	16.1	29.34	22.39	17.22	17.43	23.92	24.5	15.76	21.09	36.31	18.58	16.03	21.46	26.58	17.42	19.82	18.97	21.96	18.16
	SZAVG	28.37	18.14	20.72	7.88	27.25	23.23	11.14	23.52	28.31	32.92	26.31	18.76	31.73	26.41	20.46	21.14	26.04	25.04	17.33	24.78	36.73	20.54	20.4	25.17	27.79	21.76	22.52	21.74	26.59	22.6
LMB 80	DZ	23.08	15.93	19.72	5.27	22.77	17.41	6.13	21.64	25.68	31.71	21.42	14.99	28.89	23.17	17.69	17.9	23.6	22.98	16.98	23.44	35.5	18.72	17.04	22.11	26.34	13.85	18.06	20.3	22.63	19.56
	SZ	22.17	14.05	17.12	6.42	21.93	17.64	7.38	18.79	25.4	28.93	21.34	16.1	28.89	21.09	16.58	17.2	24.09	24.44	16.25	21.08	35.5	18.81	15.34	20.73	25.47	14.13	18.46	18.05	21.05	18.05
	SZAVG	27.56	17.54	20.44	7.96	26.12	22.14	10.21	22.98	27.19	32.87	25.17	18.52	31.05	25.26	19.27	21.07	25.85	24.59	17.47	24.51	34.9	20.36	19.9	24.46	27	18.83	21.18	21.18	25.01	22.13
LBS 4	DZ	27.51	17.87	20.1	5.91	27.47	19.94	9.51	21.42	27.56	32.1	26.08	14.99	29.79	25.83	18.22	18.12	24.15	23.77	16.04	23.73	37.58	18.22	18.69	24.61	28.25	21.46	20.61	21.21	24.31	20.39
	SZ	26.79	16.57	18.44	6.85	26.62	19.78	9.91	20.19	27.39	29.08	25.49	16.3	29.74	24.07	18.17	18.1	24.5	25.11	16.05	22.06	37.47	18.7	17.27	23.39	27.86	20.17	21.18	20.25	22.95	19.44
	SZAVG	30.57	19.55	21.11	8.05	29.35	23.79	11.99	23.74	29.52	32.94	27.91	18.71	32.07	27.72	20.99	21.37	26.51	26.28	17.41	25.1	39.06	20.57	21.24	26.64	29.38	23.87	23.6	22.5	27.38	22.98
LBS 5	DZ	25.9	17.21	20.23	5.59	25.95	19.11	8.15	21.8	27.03	32.22	24.6	15.15	29.7	24.97	18.38	18.33	24.09	23.53	16.78	23.99	37.05	18.72	18.13	23.91	27.71	18.84	19.99	21.36	23.9	20.31
	SZ	24.27	15.57	18.01	6.71	24.15	18.78	8.31	19.7	26.54	29.31	23.33	16.38	29.51	22.5	17.49	17.77	24.47	24.86	16.6	21.74	36.43	19.01	16.42	22.16	26.78	16.91	20.07	19.29	22.1	18.81
	SZAVG	29.13	18.75	20.76	8.11	27.85	23.1	10.88	23.45	28.52	32.99	26.81	18.69	31.66	26.76	20.08	21.27	26.26	25.75	17.58	24.97	38.05	20.58	20.5	25.85	28.51	21.47	22.5	21.89	26.02	22.69
HB_001	B4_80	34.34	20.62	21.93	7.47	34.15	25.15	10.19	24.76	33.46	37.06	30	19.33	38.85	29.86	20.75	22.27	29.46	30.77	17.25	26.32	43.89	20.76	21.9	27.64	33.12	25.79	24.25	22.1	29.66	24.72
	B8_80	34.63	20.63	22.55	8.38	34.4	25.69	11.18	25.28	33.87	37.31	30.41	20.05	39.22	30.09	21.41	23.19	29.94	31.14	17.76	26.77	44.1	21.3	22.53	27.89	33.51	26.52	24.88	22.5	30.09	25.06
HB_002	B4_4	35.02	20.82	22.14	7.32	34.79	25.61	10.71	24.92	34.12	37.04	30.56	19.29	39.08	30.29	21.27	22.33	29.6	30.96	17.2	26.48	44.39	20.92	22.08	28.03	33.38	26.61	24.81	22.39	30.26	24.82
	B8_4	35.41	20.92	22.84	8.25	35.08	26.24	11.72	25.42	34.6	37.26	31.01	20.05	39.46	30.53	21.95	23.27	30.11	31.28	17.69	26.92	44.63	21.51	22.84	28.38	33.9	27.42	25.49	22.9	30.69	25.21
HB_003	B4_4	34.54	20.36	21.77	7	34.68	25.07	10.33	24.34	33.48	36.59	30.06	18.77	38.66	29.93	20.66	21.71	29.02	30.61	16.55	26.03	43.87	20.35	21.64	27.39	32.98	26.58	24.26	21.77	29.98	24.04
	B8_4	35.08	20.42	22.65	7.96	34.96	25.95	11.34	25.01	34.31	37.14	30.53	19.84	39.27	30.26	21.69	22.96	29.78	31.06	17.47	26.68	44.35	21.25	22.49	28.01	33.64	27.3	25.12	22.63	30.5	24.81

จากข้อมูลในตารางที่ ก - 1 เป็นตารางที่แสดงความสามารถในการบีบอัดข้อมูลของแต่ละอัลกอริทึม ผลลัพธ์ในแต่ละช่อง เป็นค่า CR เฉลี่ย ของ 30 ตัวอย่าง ตัวอย่างละ 67 ส่วน โดยช่องที่มีสีเขียวเข้มคือช่องที่มีอัตราการบีบอัดข้อมูลสูงที่สุด ช่องสีเขียวอ่อน คือช่องที่มีความสามารถในการบีบอัดค่อนข้างสูง ในขณะที่ช่องสีส้มเข้ม หมายถึงอัตราการบีบอัดข้อมูลต่ำที่สุด สีส้มอ่อนหมายถึงอัตราการบีบอัดข้อมูลค่อนข้างต่ำ ในส่วนของตารางที่ ก - 2 เป็นตารางที่แจกแจงผลลัพธ์ของการบีบอัดข้อมูลในแต่ละตัวอย่าง (ทั้งหมด 30 ตัวอย่าง แต่ละตัวอย่างถูกแบ่งเป็น 67 ส่วน) ว่าตัวอย่างแต่ละตัว เมื่อถูกบีบอัดด้วยอัลกอริทึมที่แตกต่างกันแล้ว จะได้ผลลัพธ์ที่แตกต่างกันมากน้อยเพียงใด

เมื่อเปรียบเทียบผลลัพธ์จากตารางที่ ก - 1 กับ ตารางที่ ก - 2 จะเห็นได้ว่าผลลัพธ์มีแนวโน้มไปในทางเดียวกัน คือ การเข้ารหัสแบบ Hybrid 002 โดยมีขนาด Buffer 8 ตัว จะมีค่า CR สูงที่สุด และ CoXoH ก็ยังคงมี CR ต่ำที่สุดเกือบทุกตัวอย่าง

1.2 การทดสอบกับ Adaptive Huffman algorithm

นอกจาก CoXoH ที่ถูกนำมาทดสอบ เพื่อเปรียบเทียบประสิทธิภาพในการบีบอัดข้อมูลแล้ว ยังมีส่วนของการทดสอบกับ Adaptive Huffman ซึ่งเป็นอัลกอริทึมต้นแบบของหลาย ๆ อัลกอริทึม รวมถึง CoXoH ด้วย โดยในการทดสอบ จะใช้ชุดข้อมูลทั้งหมด 3 ชุด คือ ข้อมูลที่มีการเปลี่ยนแปลงน้อย ข้อมูลที่มีการเปลี่ยนแปลงปานกลาง และข้อมูลแบบสุ่ม โดยในการทดสอบจะเปรียบเทียบประสิทธิภาพของ DZ, SZ, SZAVG กับ Adaptive Huffman ดังภาพประกอบที่ ก - 1



ภาพประกอบที่ ก - 1 การเปรียบเทียบประสิทธิภาพของ DZ, SZ, SZAVG และ Adaptive Huffman

จากภาพประกอบที่ ก - 1 ข้อมูลที่ถูกนำมาทดสอบนั้น เป็นส่วนของรูปภาพ 30 รูป โดยที่ Low Change เป็นข้อมูล 1000 byte แรกของรูปภาพ ซึ่งมีอัตราการเปลี่ยนแปลงของข้อมูลค่อนข้างน้อย Medium Change เป็นข้อมูล 1000 byte ที่อยู่ตรงกลางรูปภาพ ซึ่งจะมีอัตราการเปลี่ยนแปลงของข้อมูลที่มากกว่า Low Change และส่วนของ Random Change เป็นข้อมูลที่เกิดจากการสุ่มขึ้นมา 1000 byte ต่อ 1 ชุด จำนวน 30 ชุด จากภาพประกอบที่ ก - 1 จะสังเกตได้ว่า Adaptive Huffman มีประสิทธิภาพในการทำงานลดลง เมื่อข้อมูลมีอัตราการเปลี่ยนแปลงข้อมูลที่สูงขึ้น และ

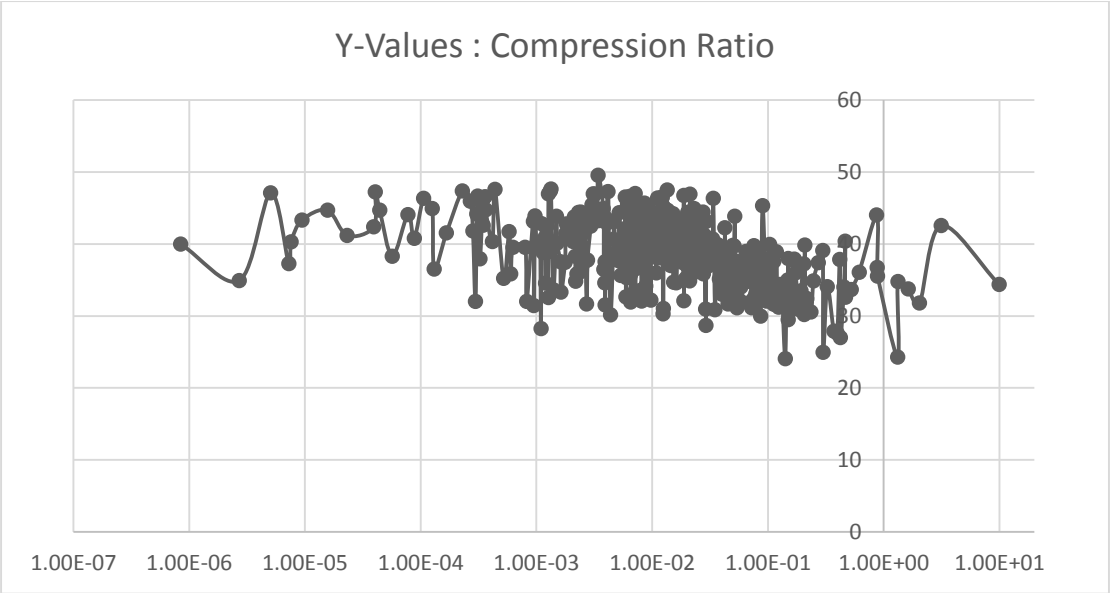
อาจมี CR ติดลบ หมายถึง ข้อมูลที่ถูกบีบอัด มีขนาดใหญ่กว่าข้อมูลต้นฉบับ ในขณะที่ DZ, SZ และ SZAVG ก็จะมีลักษณะคล้ายกัน แต่เมื่อเข้ารหัสข้อมูลที่มีอัตราการเปลี่ยนแปลงข้อมูลสูง (Random Change) ค่า CR ที่ได้จะเข้าใกล้ 0 คือ ผลลัพธ์ที่ได้จากการบีบอัด มีปริมาณเท่ากันกับข้อมูลต้นฉบับ ซึ่งถือว่าเป็นข้อดี อย่างไรก็ตาม ถ้าหากนำทั้ง 4 อัลกอริทึมมาบีบอัดข้อมูลที่มีอัตราการเปลี่ยนแปลงของข้อมูลที่น้อย จะทำให้ Adaptive Huffman มีประสิทธิภาพที่สูงกว่า DZ, SZ และ SZAVG

2. การวิเคราะห์ความสัมพันธ์ของข้อมูล

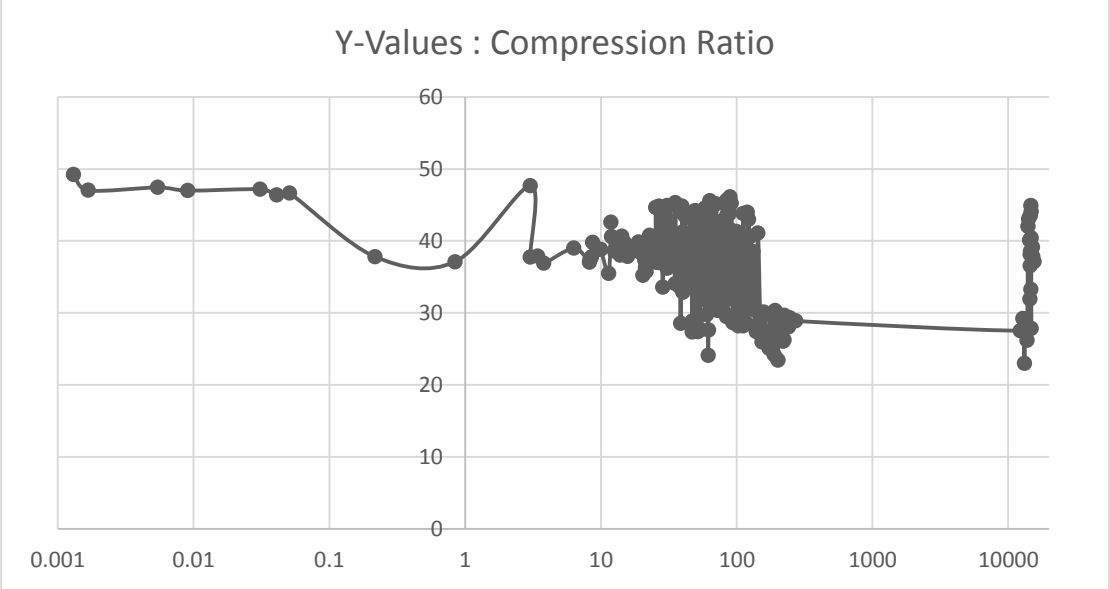
นอกจากการทดสอบประสิทธิภาพในการบีบอัดข้อมูลแล้ว ยังมีส่วนของการวิเคราะห์ข้อมูล เพื่อประเมิน และทำนายประสิทธิภาพของอัลกอริทึมต่อข้อมูลชุดนั้น ๆ อยู่ด้วย โดยในการทดสอบนี้จะใช้อัลกอริทึม SZAVGLB ที่มีขนาด Buffer 40 เป็นตัวทดสอบ โดยตัวชี้วัดที่นำมาทดสอบจะมีทั้งหมด 3 ตัว ด้วยกัน คือ ค่าความแปรปรวน (SD), ค่าเฉลี่ย (Mean) และค่าเฉลี่ยของลอการิทึม (Log mean)

ในการทดสอบ จะใช้ข้อมูล EEG มาทำการทดสอบ โดยจะแบ่งข้อมูลออกเป็นตัวอย่าง F, N, O, S และ Z แต่ละตัวอย่างจะมีข้อมูล 396 ชุด วิธีในการทดสอบจะนำชุดข้อมูลภายในทั้งหมดมาทำการเรียงใหม่ด้วยตัวชี้วัด (SD, Mean หรือ Log mean) โดยตัวชี้วัดจะไม่ได้คำนวณจากตัวข้อมูลโดยตรง แต่จะคำนวณจากผลต่างระหว่างค่าของข้อมูลตัวที่อยู่ติดกัน ดังนั้นจะทำให้ค่าความแปรปรวนในที่นี้ หมายถึงค่าความแปรปรวนของผลต่างของข้อมูล ค่าเฉลี่ยก็จะหมายถึงค่าเฉลี่ยของผลต่างของข้อมูล และค่าเฉลี่ยของลอการิทึมก็จะหมายถึงค่าเฉลี่ยของลอการิทึมของผลต่างของข้อมูล

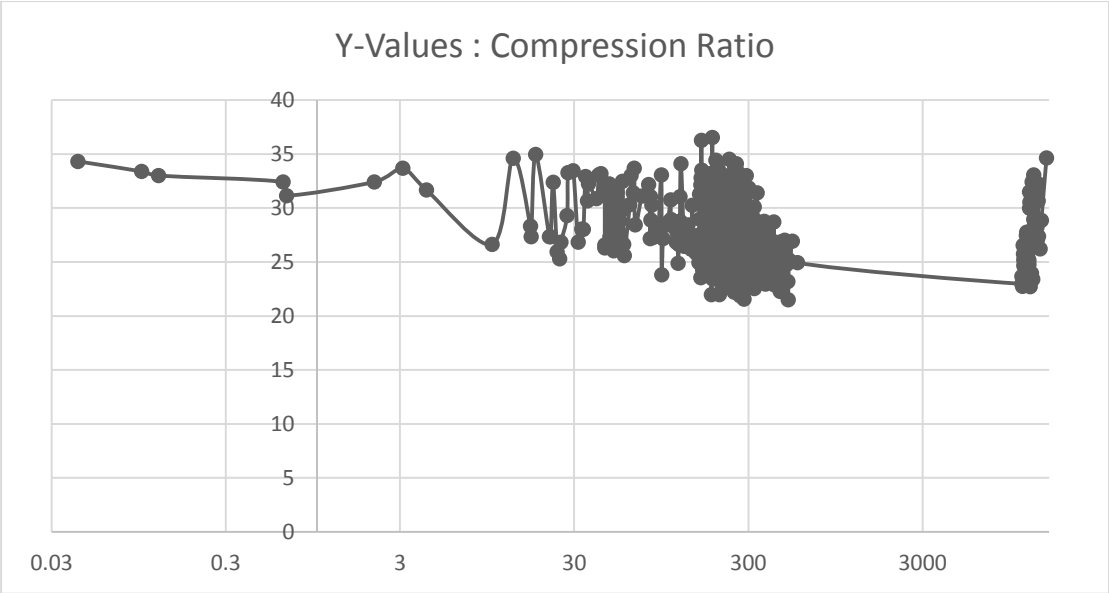
การทดสอบจะเริ่มต้นจากการทดสอบด้วยการใช้ค่าความแปรปรวนเป็นตัวชี้วัด โดยเมื่อนำข้อมูลมาเรียงใหม่ตามค่าความแปรปรวนจากน้อยไปมาก แล้วทำการบีบอัดด้วย SZAVGLB ที่ขนาด Buffer 40 จะทำให้ได้ผลลัพธ์ดังนี้



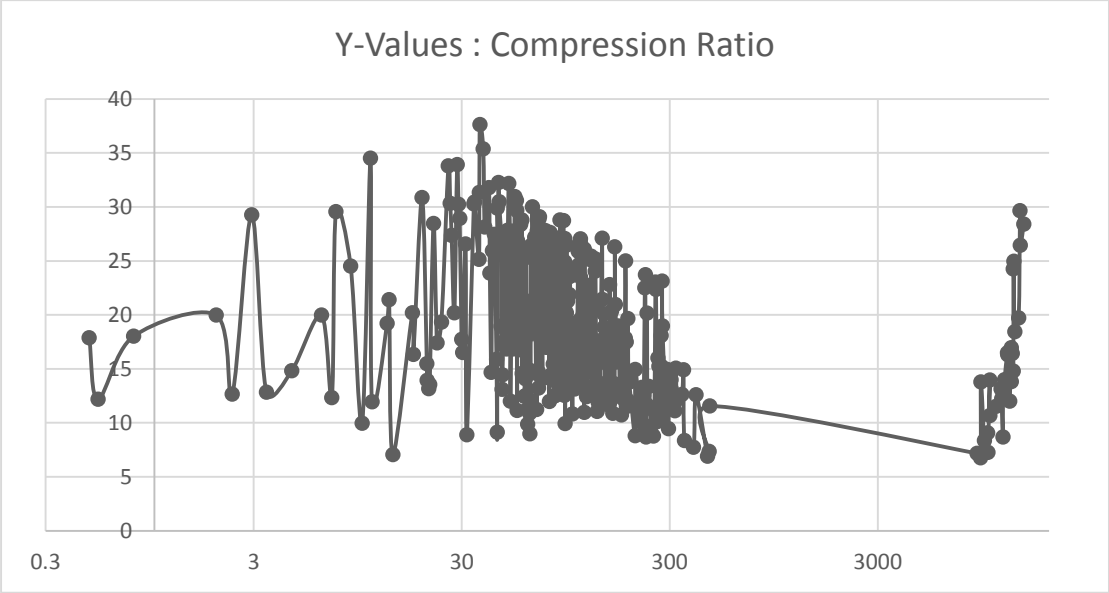
ภาพประกอบที่ ก - 2 ผลการบีบอัดเมื่อนำข้อมูล EEG ตัวอย่าง F มาเรียงตามค่า SD ของผลต่าง



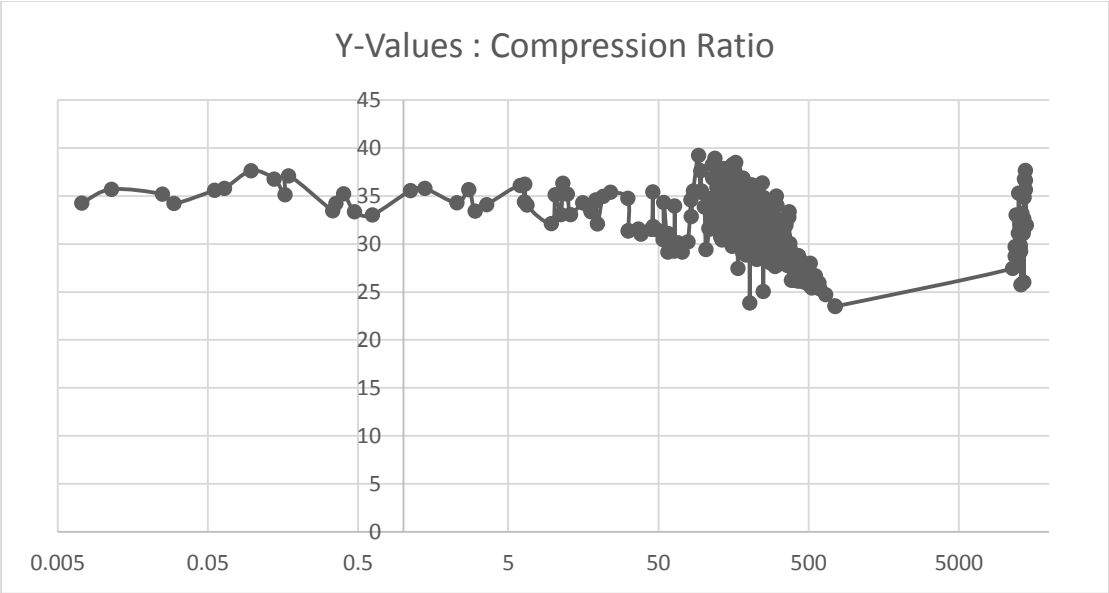
ภาพประกอบที่ ก - 3 ผลการบีบอัดเมื่อนำข้อมูล EEG ตัวอย่าง N มาเรียงตามค่า SD ของผลต่าง



ภาพประกอบที่ ก - 4 ผลการบีบอัดเมื่อนำข้อมูล EEG ตัวอย่าง O มาเรียงตามค่า SD ของผลต่าง



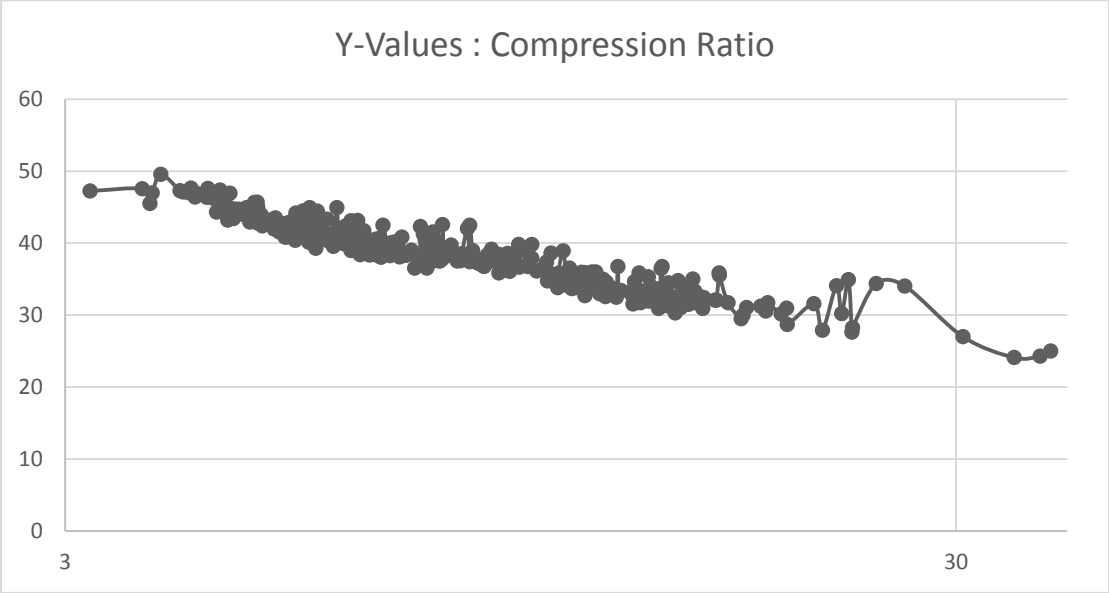
ภาพประกอบที่ ก - 5 ผลการบีบอัดเมื่อนำข้อมูล EEG ตัวอย่าง S มาเรียงตามค่า SD ของผลต่าง



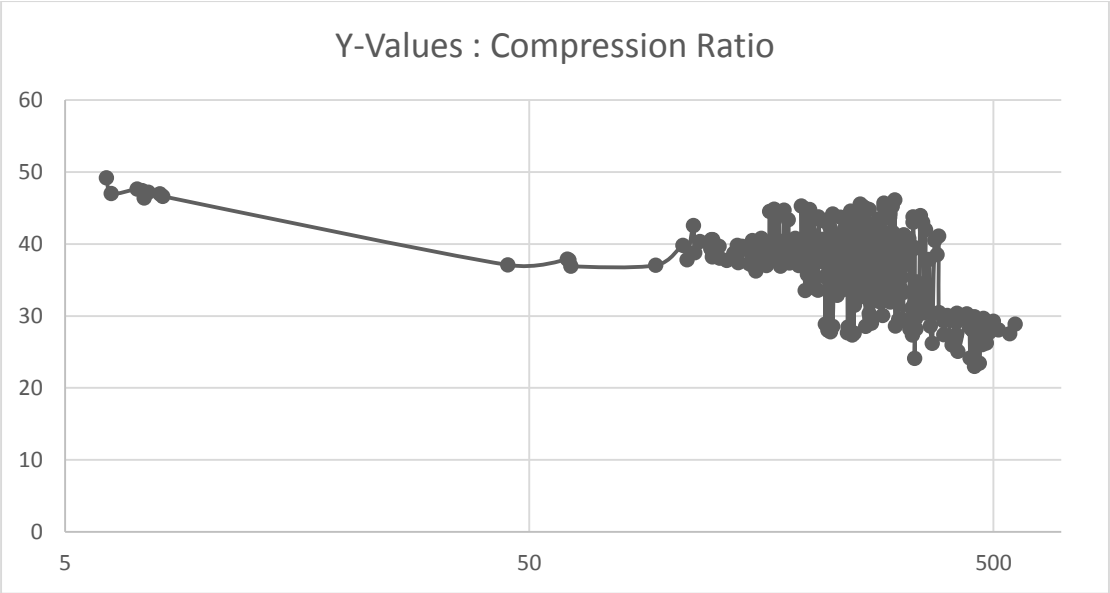
ภาพประกอบที่ ก - 6 ผลการบีบอัดเมื่อนำข้อมูล EEG ตัวอย่าง Z มาเรียงตามค่า SD ของผลต่าง

จากผลลัพธ์ที่แสดงข้างต้นจะสังเกตได้ว่า อัตราการบีบอัดข้อมูลไม่มีแนวโน้มไปในทิศทางใดเลย ไม่ว่าจะใช้กลุ่มข้อมูลใดมาทำการทดสอบ จึงทำให้สามารถสรุปได้ว่าค่าความแปรปรวนไม่มีคุณสมบัติเป็นตัวชี้วัดประสิทธิภาพในการทำงานของ SZAVGLB

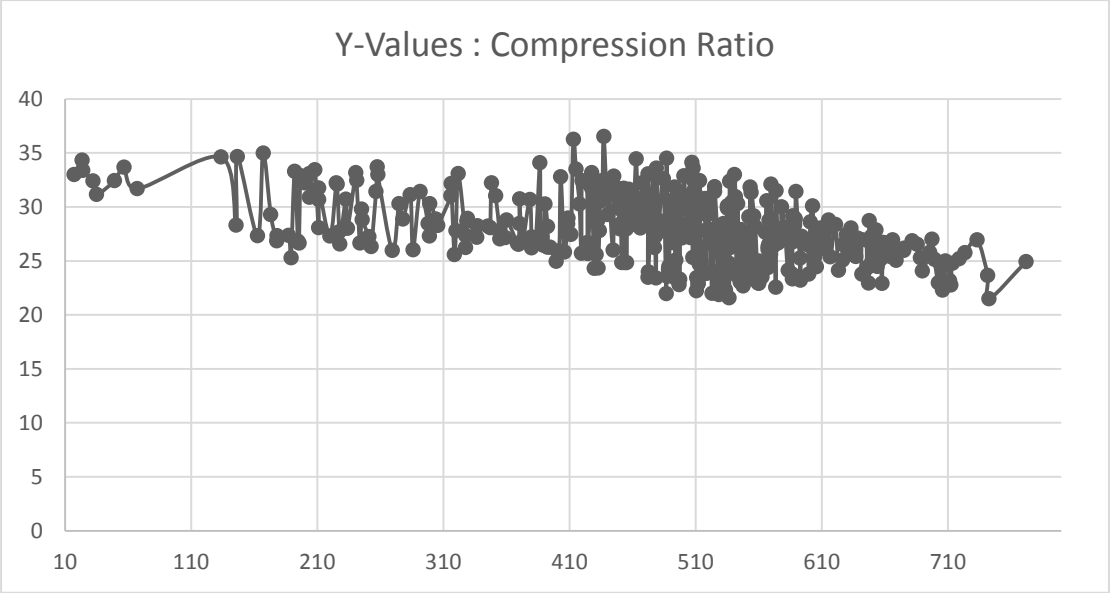
การทดสอบถัดมา จะเป็นการทดสอบคุณสมบัติของค่าเฉลี่ย ว่ามีคุณสมบัติเป็นตัวชี้วัดประสิทธิภาพในการบีบอัดข้อมูลของ SZAVGLB หรือไม่ โดยจะนำข้อมูลในแต่ละกลุ่มมาเรียงตามค่าเฉลี่ยของผลต่างของข้อมูลภายใน แล้วทำการบีบอัดโดยใช้ SZAVGLB ที่ขนาด Buffer 40 ซึ่งจะได้ผลลัพธ์ดังนี้



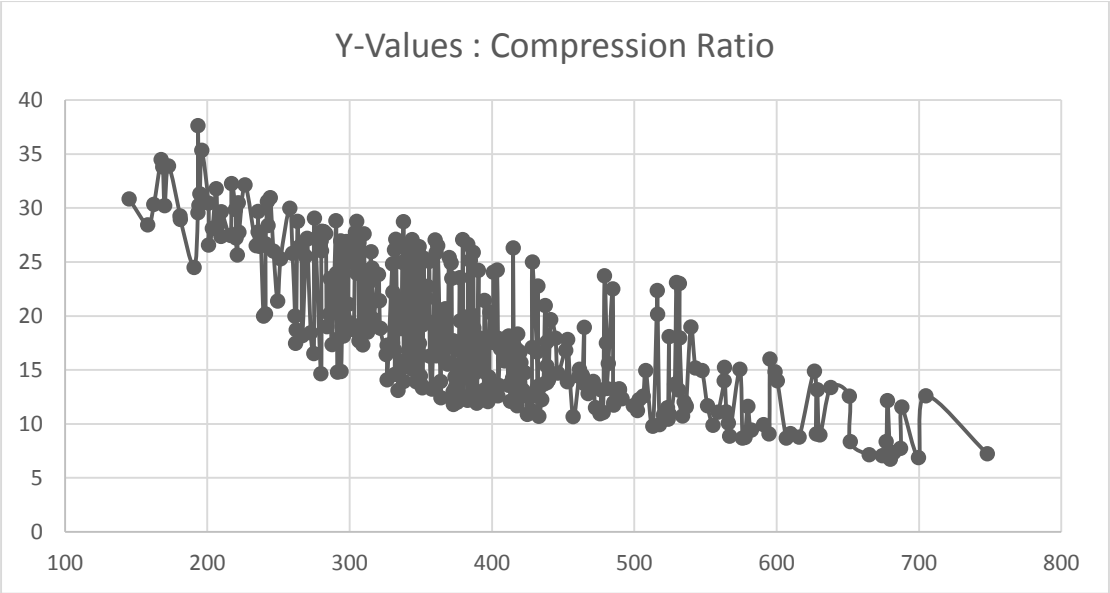
ภาพประกอบที่ ก - 7 ผลการบีบอัดเมื่อนำข้อมูล EEG ตัวอย่าง F มาเรียงตามค่า Mean ของผลต่าง



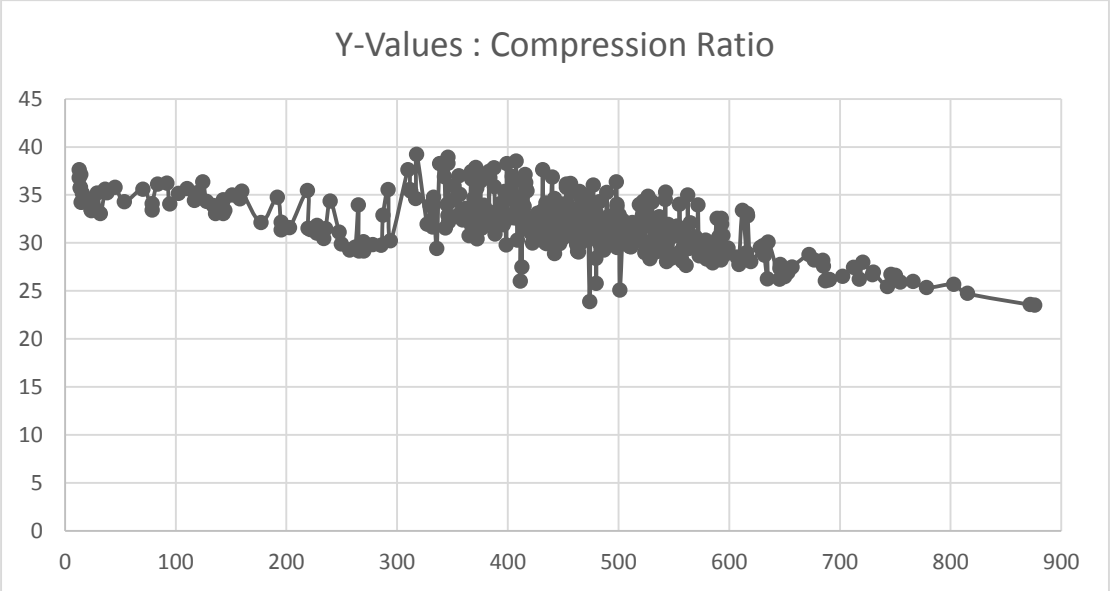
ภาพประกอบที่ ก - 8 ผลการบีบอัดเมื่อนำข้อมูล EEG ตัวอย่าง N มาเรียงตามค่า Mean ของผลต่าง



ภาพประกอบที่ ก - 9 ผลการบีบอัดเมื่อนำข้อมูล EEG ตัวอย่าง O มาเรียงตามค่า Mean ของผลต่าง



ภาพประกอบที่ ก - 10 ผลการบีบอัดเมื่อนำข้อมูล EEG ตัวอย่าง S มาเรียงตามค่า Mean ของผลต่าง

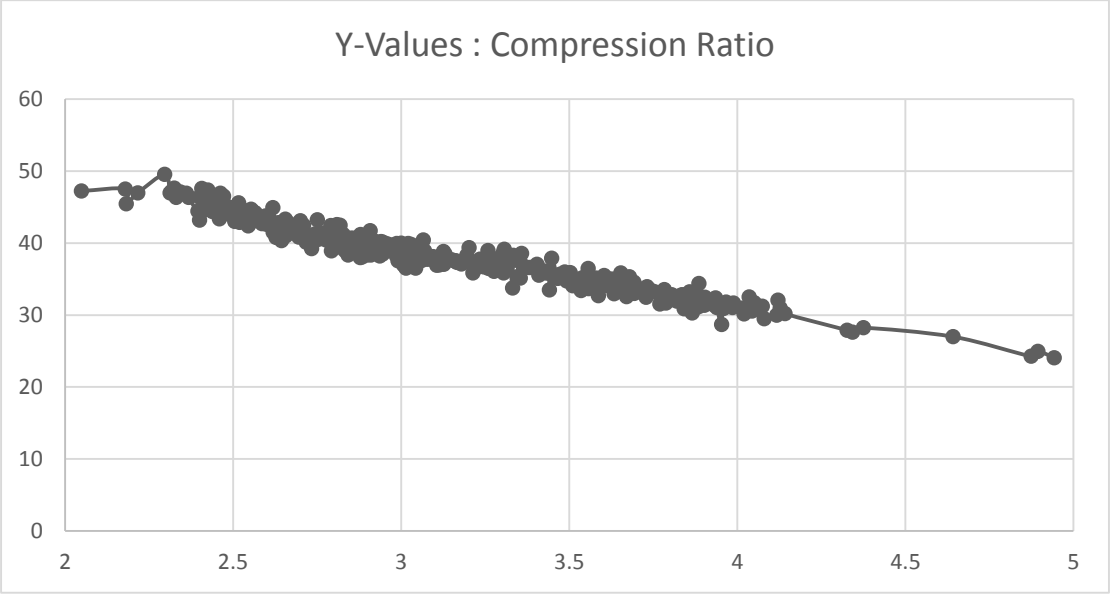


ภาพประกอบที่ ก - 11 ผลการบีบอัดเมื่อนำข้อมูล EEG ตัวอย่าง Z มาเรียงตามค่า Mean ของผลต่าง

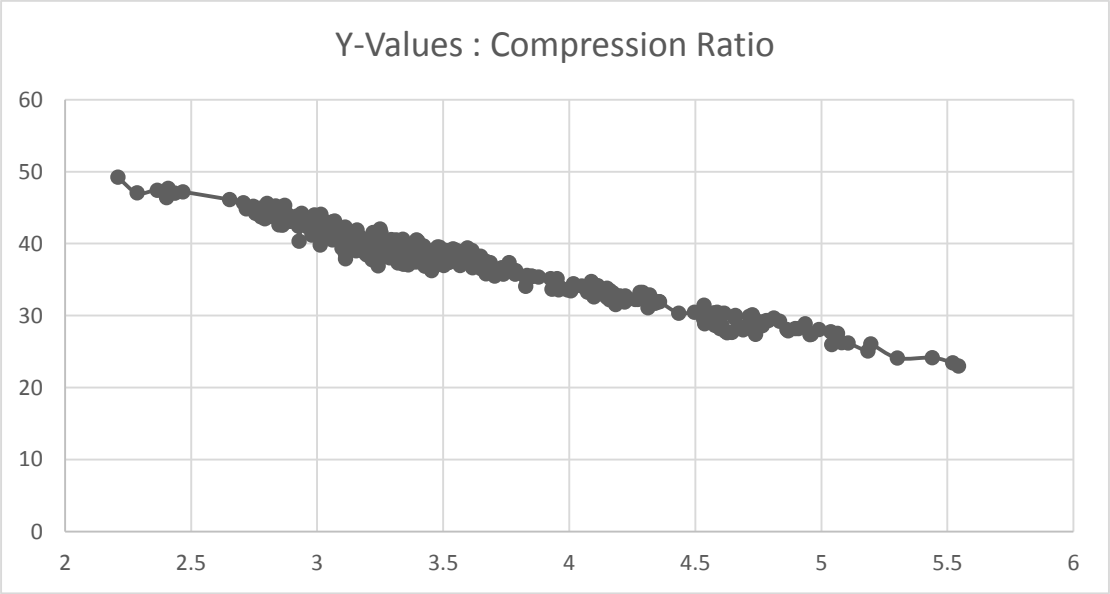
จากผลลัพธ์ข้างต้น จะเห็นได้ว่าผลการบีบอัดมีแนวโน้มที่ดีกว่าการเรียงข้อมูลตามค่า SD ของผลต่าง แต่ก็ยังขาดความแม่นยำอยู่มากพอสมควร เช่นผลลัพธ์จากภาพประกอบที่ ก - 8, ภาพประกอบที่ ก - 9 และภาพประกอบที่ ก - 10 ที่แม้จะมีแนวโน้มลดลงเมื่อค่าเฉลี่ยเพิ่มขึ้น แต่การแกว่งของผลลัพธ์ก็ยังคงมีค่ามาก ดังนั้นอาจสรุปได้ว่าการใช้ค่าเฉลี่ยของผลต่างมาเป็นตัวชี้วัดนั้น พอจะสามารถทำได้ แต่อาจมีความคลาดเคลื่อนอยู่บ้าง

ตัวชี้วัดตัวสุดท้ายที่จะนำมาทดสอบ คือค่าเฉลี่ยของลอการิทึมของผลต่าง ในที่นี้จะใช้ลอการิทึมฐาน 2 มาทำการทดสอบ โดยส่วนของข้อมูลที่ถูกทดสอบ จะถูกนำมาเรียงด้วย

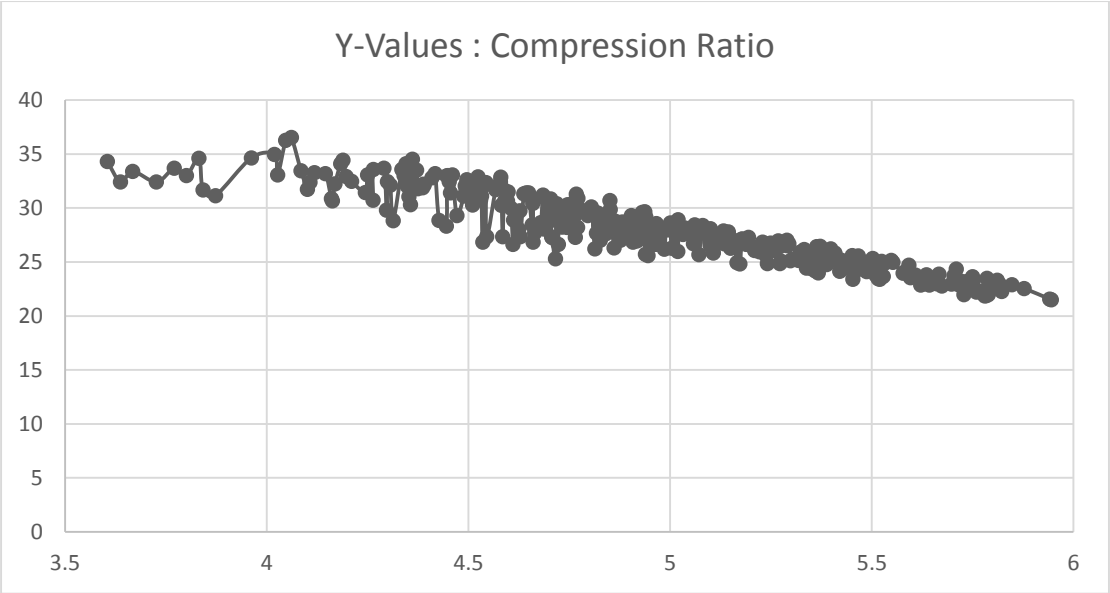
ค่าเฉลี่ยของลอการิทึมฐาน 2 ของผลต่างของข้อมูล แล้วทำการบีบอัดด้วย SZAVGLB ที่ขนาด Buffer 40 โดยข้อมูลที่นำมาใช้จะเป็น EEG จำนวน 5 กลุ่มตัวอย่าง แต่ละกลุ่มจะมีข้อมูล 396 ชุด ผลลัพธ์ในการทดสอบจะเป็นดังนี้



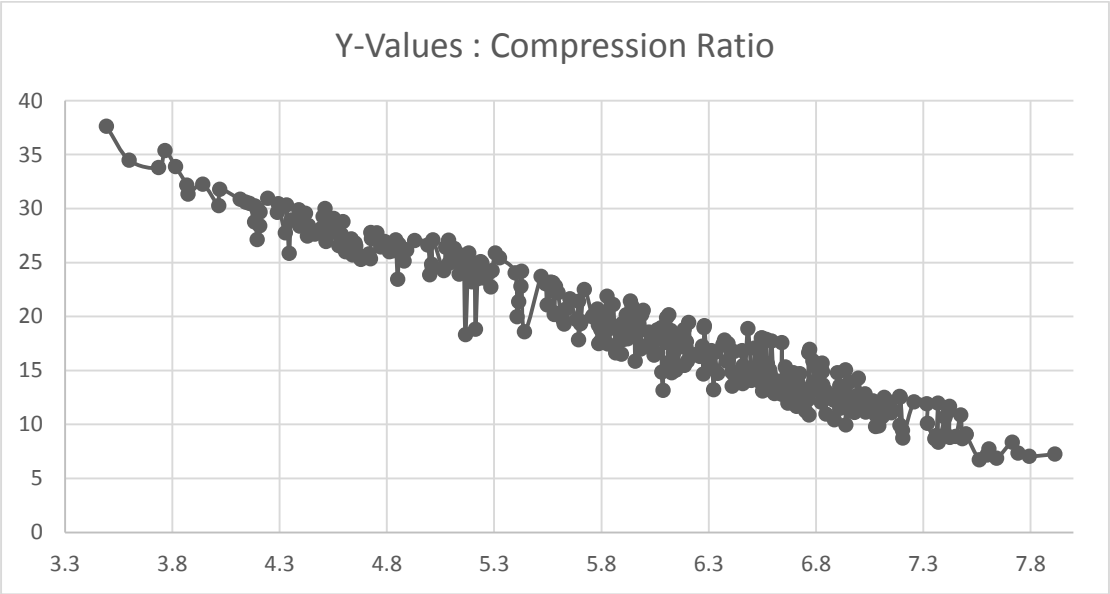
ภาพประกอบที่ ก - 12 ผลการบีบอัดเมื่อนำข้อมูล EEG ตัวอย่าง F มาเรียงตามค่าเฉลี่ยของลอการิทึมของผลต่าง



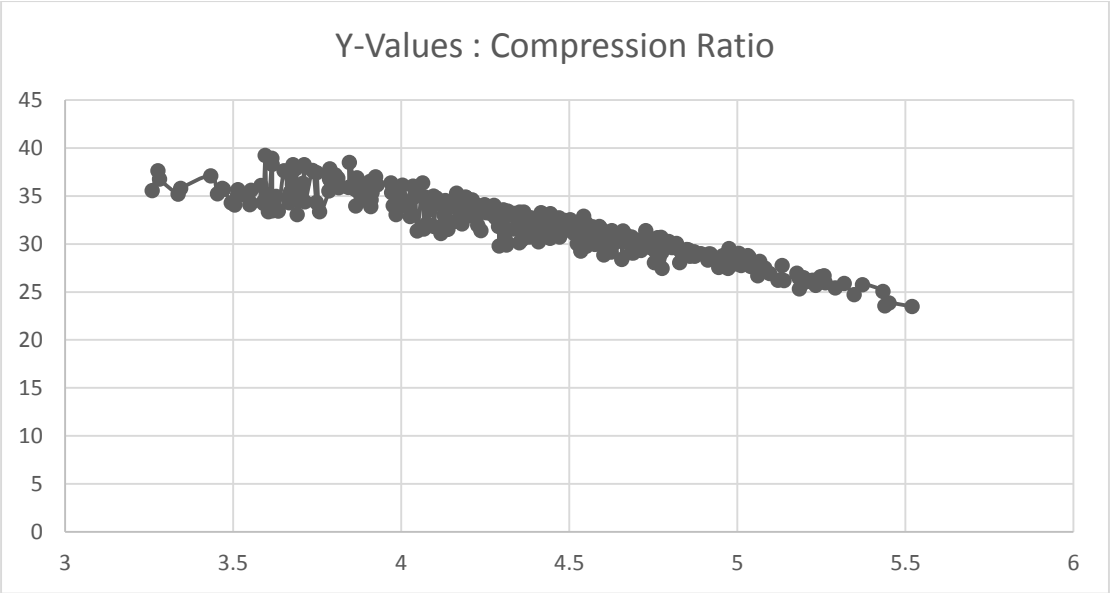
ภาพประกอบที่ ก - 13 ผลการบีบอัดเมื่อนำข้อมูล EEG ตัวอย่าง N มาเรียงตามค่าเฉลี่ยของลอการิทึมของผลต่าง



ภาพประกอบที่ ก - 14 ผลการบีบอัดเมื่อนำข้อมูล EEG ตัวอย่าง O มาเรียงตามค่าเฉลี่ยของลอการิทึมของผลต่าง



ภาพประกอบที่ ก - 15 ผลการบีบอัดเมื่อนำข้อมูล EEG ตัวอย่าง S มาเรียงตามค่าเฉลี่ยของลอการิทึมของผลต่าง



ภาพประกอบที่ ก - 16 ผลการบีบอัดเมื่อนำข้อมูล EEG ตัวอย่าง Z มาเรียงตามค่าเฉลี่ยของลอการิทึมของผลต่าง

จากผลลัพธ์ข้างต้น จะเห็นได้ว่าผลการบีบอัดข้อมูลที่ถูกเรียงด้วยค่าเฉลี่ยของลอการิทึมฐาน 2 ของผลต่าง มีแนวโน้มที่ชัดเจน คือเมื่อค่าเฉลี่ยของลอการิทึมฐาน 2 ของผลต่างมีค่าเพิ่มขึ้น จะทำให้อัตราการบีบอัดข้อมูลลดลง จากผลลัพธ์ที่ได้สามารถสรุปได้ว่า ค่าเฉลี่ยของลอการิทึม มีคุณสมบัติเป็นตัวชี้วัดที่ดี แต่อย่างไรก็ตามผลลัพธ์ที่ได้ยังคงมีความคลาดเคลื่อนเล็กน้อย และยังมีวิธีการคำนวณที่ยุ่งยากกว่าการหาค่าเฉลี่ยค่อนข้างมา จึงทำให้การใช้ค่าเฉลี่ยของลอการิทึมมาเป็นตัวชี้วัด อาจทำได้ยากกว่าวิธีอื่น ๆ เล็กน้อย

การมีตัวชี้วัดที่ดีจะทำให้สามารถนำอัลกอริทึมที่ได้ออกแบบไปประยุกต์ใช้งานได้หลากหลายยิ่งขึ้น เช่น การประเมินประสิทธิภาพเอาไว้ล่วงหน้า จะทำให้จัดการเลือกวิธีการบีบอัดข้อมูลได้เหมาะสมยิ่งขึ้น ทั้งยังสามารถปรับเลือกขนาดของ Buffer ให้เหมาะสม หรือกระทั่งการคาดการอายุการทำงานของเซนเซอร์โหนด แต่อย่างไรก็ตามตัวชี้วัดที่ได้ทำการทดสอบนั้นสามารถใช้ได้กับ SZAVG เท่านั้น ถ้าหากต้องการนำไปใช้กับอัลกอริทึมอื่น เช่น Adaptive Huffman หรือ CoXoH อาจไม่เหมาะสม

ภาคผนวก ข
ตรวจสอบความถูกต้องของอัลกอริทึม

1. ตรวจสอบความถูกต้องเชิงทฤษฎี

ปัจจัยที่สำคัญอีกอย่างหนึ่งของการออกแบบวิธีการเข้ารหัสแบบ Lossless ก็คือต้องสามารถมั่นใจได้ว่า ข้อมูลที่ผ่านการเข้ารหัสมาแล้วนั้น จะต้องทำการถอดรหัสกลับออกมาได้อย่างถูกต้อง ในหัวข้อนี้จึงนำเสนอสมการการเข้ารหัส และการถอดรหัส ในรูปแบบของ pseudocode พร้อมทั้งแก้สมการเพื่อตรวจสอบความถูกต้องของอัลกอริทึม โดยจะเลือกใช้ Single Zero algorithm เป็นตัวอย่างในการทดสอบ

รูปแบบของการทำงานของตัวเข้ารหัส และตัวถอดรหัส จะถูกแบ่งออกเป็น 3 ส่วน ได้แก่ ส่วนของการทำนาย ส่วนของการสร้างผลลัพธ์ และส่วนของการอัพเดทข้อมูล โดยส่วนของตัวเข้ารหัส และตัวถอดรหัส จะมี pseudocode ดังนี้

```
// === === Encoder === ===

// Prediction part.
Mn = P(Reg);

// Output creation part.
Sn,Fn = AbsSub(Dn-1, Dn);
if (Mn ϵ {M0, M1, M2})
       Rn = Dn;
       // Result size = 8 bit.
else if (Mn | Sn != Mn)
       Rn = {1, Dn};
       // Result size = 9 bit.
else if (Mn | Sn == Mn)
       Rn = {0, Fn, Sn(Mn bit)};
       // Result size = 3 – 7 bit.

// Update part.
Update(Reg, Sn);
```

```
// === === Decoder === ===

// Prediction part.
Mn = P(Reg);

// Output creation part.
if (Mn ϵ {M0, M1, M2})
       Rn = D[0:7];
else if (D[0] == 1’b1)
       Rn = D[1:8];
else if (D[0] == 1’b0)
       Fn = D[1];
       Sn = D[2:2+(bit 1 of Mn)];
       Rn = Rn-1 + pow(-1,Fn+1)*Sn;

Sn = AbsSub(Rn-1, Rn);

// Update part.
Update(Reg, Sn);
```

จาก pseudocode ข้างต้น จะสังเกตได้ว่าทั้งตัวเข้ารหัส และตัวถอดรหัส ต่างก็มีส่วนของการทำนาย และส่วนของการอัพเดทข้อมูลที่เหมือนกัน ซึ่งสองส่วนนี้ จะมีไว้เพื่อใช้ในการทำนายผล โดยผลการทำนายจะนำ Reg มาทำการคำนวณด้วยฟังก์ชัน P(x) แล้วรีเทิร์นผลลัพธ์ออกมาเก็บไว้ใน Mn และขั้นตอนการอัพเดท จะนำ Sn มาใช้อัพเดทค่าของ Reg ด้วยฟังก์ชัน Update(x,y) เพื่อใช้ในการทำนายในครั้งต่อไป จากส่วนที่เหมือนกันนี้ สามารถสรุปได้ว่า ถ้าหาก

ตัวเข้ารหัส และตัวถอดรหัส ได้รับ Sn ที่เหมือนกันในทุก ๆ รอบ ก็จะทำให้มีผลการทำนายที่เหมือนกัน หรืออาจกล่าวได้ว่า ถ้า Sn ของทั้ง 2 ฝั่ง เหมือนกันทุกรอบ ก็จะทำให้ Mn ที่ออกมามีค่าเหมือนกันในทุก ๆ รอบเช่นกัน

ส่วนที่จะต้องพิจารณาต่อไป คือส่วนของการสร้างผลลัพธ์ เนื่องจากตัวเข้ารหัส และตัวถอดรหัส มีการทำงานที่แตกต่างกัน ซึ่งทั้งฝั่ง จะแบ่งการทำงานในส่วนนี้ออกเป็น 3 กรณี โดยในการพิสูจน์ จะต้องแยกพิสูจน์แต่ละกรณีไป

- กรณีที่ 1 เข้าเงื่อนไขที่ 1 คือ Mn มีค่าเป็น M0, M1 หรือ M2 ในกรณีนี้ ทางด้านฝั่งส่งข้อมูลจะได้ผลลัพธ์ของการเข้ารหัสออกมา 8 บิต หรือก็คือ ไม่ต้องทำการเข้ารหัส แต่ให้ส่งข้อมูลดิบออกไปเป็นผลลัพธ์ได้ทันที ในกรณีนี้จะใช้หลักการของการเท่ากันของ Mn ของทั้งฝั่งเข้ารหัส และฝั่งถอดรหัส ทำให้ฝั่งถอดรหัสสามารถรับทราบได้ว่าข้อมูลในรอบดังกล่าว ไม่ได้ถูกเข้ารหัสมา จึงสามารถดึงเอาข้อมูลดิบ 8 บิต มาเป็นผลลัพธ์ของรอบนั้นได้ทันที ส่วนการสร้าง Sn เพื่อทำการอัพเดท Reg จะเห็นได้ว่าทั้งฝั่งเข้ารหัส และฝั่งถอดรหัสมีการทำงานที่เหมือนกัน จึงมั่นใจได้ว่า ค่า Sn ของทั้ง 2 ฝั่ง จะต้องเหมือนกันอย่างแน่นอน
- กรณีที่ 2 เข้าเงื่อนไขที่ 2 คือ เมื่อนำ Mn มาทำการ or แบบ บิตต่อบิต กับ Sn แล้วผลลัพธ์ที่ได้ มีค่าไม่เท่ากันกับ Mn ในกรณีนี้ ผลลัพธ์ของฝั่งเข้ารหัส (Rn) จะมีจำนวน 9 บิต ประกอบด้วย บิตที่มีค่า 1 ตามด้วยข้อมูลต้นฉบับที่ไม่ได้ผ่านการเข้ารหัสซึ่งมีขนาด 8 บิต ทางด้านตัวถอดรหัส จะสามารถทราบได้จากการดึงข้อมูลเข้ามาก่อน 1 บิต แล้วตรวจสอบว่าบิตดังกล่าวมีค่าเป็น 1 หรือไม่ ถ้าหากมีค่าเป็น 1 ก็จะเข้าเงื่อนไขที่สองของตัวถอดรหัส ในกรณีนี้จะทำการดึงข้อมูลเพิ่มเติมเข้ามาอีก 8 บิต ซึ่งข้อมูลส่วนนี้ จะเป็นผลลัพธ์ของการถอดรหัสทันที ในส่วนของ Sn ก็จะมีการคำนวณที่เหมือนกันทั่งฝั่งเข้ารหัส และฝั่งถอดรหัส ดังนั้น จึงมั่นใจได้ว่าค่า Sn ที่ได้ จะมีความถูกต้องแน่นอน
- กรณีที่ 3 เข้าเงื่อนไขที่ 3 คือ เมื่อนำ Mn มาทำการ or แบบ บิตต่อบิต กับ Sn แล้ว ผลลัพธ์ที่ได้ มีค่าเท่ากันกับ Mn ในกรณีนี้ ผลลัพธ์ของฝั่งเข้ารหัส (Rn) จะประกอบไปด้วย บิต 0 ตามด้วยค่า Fn (ขนาด 1 บิต) และตามด้วย Sn ที่มีขนาดเท่ากับจำนวนบิต 1 ของ Mn คือ เลือกเฉพาะบิตของ Sn ที่มีตำแหน่งตรงกันกับบิตที่มีค่า 1 ของ Mn ทำให้ผลลัพธ์ที่ได้ มีขนาดตั้งแต่ 3 – 7 บิต ในส่วนของตัวถอดรหัสจะทำการอ่านค่าเข้ามาก่อน 1 บิต ถ้าบิตดังกล่าวมีค่าเป็น 0 ก็จะเข้าเงื่อนไขที่ 3 ในเงื่อนไขที่ 3 จะทำการอ่านค่าเข้ามาอีก 1 บิต และเก็บค่านั้นเอาไว้ใน Fn และจะอ่านข้อมูลเข้ามาเก็บไว้ใน Sn เป็นจำนวนเท่ากับจำนวนบิต 1 ของ Mn หลังจากนั้น จึงทำการคำนวณหาผลลัพธ์ โดยผลลัพธ์จะขึ้นอยู่กับ Rn-1 คือ ผลลัพธ์ในรอบก่อนหน้า Sn ส่วนต่างระหว่างผลลัพธ์รอบก่อนหน้า เทียบกับรอบนี้ และ Fn คือ ทิศทางของส่วนต่าง จากตัวแปรทั้ง

3 ตัว จะได้ว่า $R_n = R_{n-1} + ((-1)^{Fn} \times S_n)$ ในส่วนของ Sn ที่ได้นั้น หลังจากที่คำนวณ Rn เสร็จแล้ว ก็จะนำ Rn กับ Rn-1 ไปคำนวณด้วยฟังก์ชันเดียวกันกับที่ใช้ในตัวเข้ารหัส ทำให้มั่นใจได้ว่า Sn ตัวใหม่ที่คำนวณออกมาจะต้องเหมือนกันกับ Sn ของตัวเข้ารหัส แน่นอน

2. ตรวจสอบความถูกต้องด้วยการจำลอง

นอกจากการตรวจสอบความถูกต้องในเชิงทฤษฎีแล้ว การตรวจสอบความถูกต้องในการทำงานจริง ก็เป็นส่วนสำคัญเช่นกัน ในงานวิจัยชิ้นนี้ ได้ทำการทดสอบความถูกต้องในการทำงานด้วยการสร้างโปรแกรมจำลองการทำงานขึ้นมา ในการทดสอบจะใช้การนำข้อมูลดิบที่มีปริมาณมากมาทำการเข้ารหัส แล้วถอดรหัส จากนั้นตรวจสอบความถูกต้องกับข้อมูลต้นฉบับแบบบิตต่อบิต ข้อมูลที่ถูกนำมาทดสอบ เป็นข้อมูลรูปภาพแบบ Grayscal ขนาด 1 MB แบ่งเป็น 100 KB จำนวน 10 ชุด ทดสอบด้วยอัลกอริทึม DZLBS, SZLBS และ SZAVGLBS ผลการทดสอบปรากฏว่าทุกอัลกอริทึมสามารถทำงานได้ถูกต้อง คือ ผลลัพธ์ที่ได้จากการถอดรหัส มีรายละเอียดเหมือนกับข้อมูลต้นฉบับทุกประการ

ภาคผนวก ค
ผลงานตีพิมพ์เผยแพร่จากวิทยานิพนธ์


ITC-CSCC 2014
Phuket Thailand
ITC-CSCC 2014
The 29th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC 2014)
July 1–4, 2014
Phuket Graceland Resort and Spa
Phuket, Thailand
IEIE
ECTI Association
eic
TCEB
Seagate
PIM
PAKGON
Aimagin

The 29th International Technical Conference on Circuit/Systems Computers and Communications (ITC-CSCC), Phuket, Thailand, July 1-4, 2014

A Novel Data Compression Circuits for Wireless Sensor Networks

Pakawat Tinsirisuk
Computer Engineering, Faculty of Engineering,
Prince of Songkla University,
Hatyai Songkhla 90112 Thailand.
pakawat_tinsirisuk@hotmail.com Authors Name/s per

W. Suntiamorntut
Computer Engineering, Faculty of Engineering,
Prince of Songkla University,
Hatyai Songkhla 90112 Thailand.
wannarat@coe.psu.ac.t

***Abstract*— Wireless sensor networks (WSNs) is well known and widely use in many applications. The main factor is power consumption, which most used in data transmission [1], because of reducing of power consumption will give more life time to sensor node. This paper present data compression in WSNs to reduce amount of data transmission, designed for compress different data kind. 3 algorithms called Double Zero, Single Zero and Single Zero with Average are designed and test with 3 difference data levels compare to Adaptive Huffman algorithm. This paper cover to FPGA circuit design for compare amount of resource used and calculate power consumption in the future.**

I. Introduction

WSNs have been developed in many ways such as smaller size higher performance and less power consumption which is very important because of it affect to life time of sensor node. Sensor node use power for computing, sensors reading and data communication which use most power as 80% [1]. In this point, if we can make system transmit less data, system will save more energy and it is reason to compress data before send it out.

Data compression in WSNs must have specific design because of normal compression algorithm cannot provide highest compression ratio for WSNs data such as climate monitoring system, health care system [2] and picture transmission [3][4]. This is reason why not we can't use normal compression algorithm to compress WSN data directly. Some papers have to modify existing algorithm to be more suitable for WSN. Popular algorithm, which is modified and compare, is Huffman algorithm and Adaptive Huffman algorithm [5][6][7] because of this algorithm is simple and give high compression ratio. Point of research is not only to compare compression ratio but cover to compare amount of hardware resource used and power consumption by simulate of FPGA circuit [8][9][10].

This paper present compression algorithm, which can use to compress serial continuous data by provide an acceptable delay for keep data up to date. This algorithm aims to be general purpose compression algorithm for compress difference kind of data and provide less compression overhead to prevent load increased based on lossless compression. By this reason, it describes why Huffman algorithm, which must have much data to able to compress, is not suitable for this system.

This paper present Double Zero (DZ) algorithm, Single Zero (SZ) algorithm and Single Zero with Average (SZAvg) algorithm compare compression ratio to Adaptive Huffman algorithm by 3 different data level such as low change, medium change and random change (30 sample for each data level). This paper covered to circuit simulation on FPGA for compare amount of hardware resource used to be database for calculate power consumption in next step of research (simulation based on FPGA Spartan 6 XC6SLX9).

II. Compression Algorithm

Principle of both three compression algorithm is use data statistic like data frequency to predict next data. After that, find different value between previous data and current data and compare to predicted data for select model of compression result. DZ algorithm, SZ algorithm and SZAvg algorithm have same structure but have difference in amount of statistic register and predictive model.

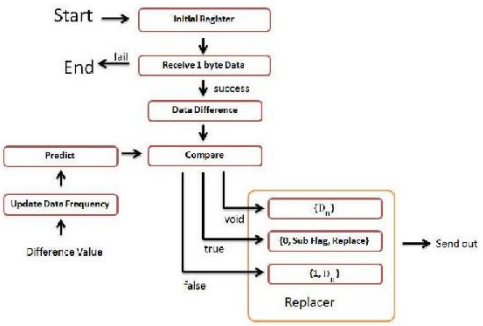


Fig. 1. Algorithm flow chart.

Figure 1 Show process of compression of DZ, SZ and SZAvg algorithm by step follows :

- Initial Register : this process use to initial register value such as reset statistic data and set value of previous data (D_{n-1}).
- Receive 1 byte Data : get 1 byte of data (D_n) to send to next state.
- Data Difference : Find absolute of different value between previous data and current data ($|D_{n-1} - D_n|$) and keep Sub Flag (0 when $D_{n-1} \geq D_n$ and 1 when $D_{n-1} < D_n$).
- Predict : use statistic data to predict model of next data.
- Compare : use to compare data from Data Difference and Predict and give 3 type of result (void : don't change because compression model don't give positive compression ratio, true : predicted value same to data difference, false : predicted value don't same to data difference).
- Replacer : replace value to compression model depend on result of Compare (void : $\{D_n\}$, true : {0,Sub Flag, Replace}, false : $\{1, D_n\}$).
- Update Data Frequency : use result of Data Difference to update statistic information for next prediction.

DZ, SZ and SZAvg algorithm are same but difference on statistic model, comparison and prediction.

A. *Double Zero algorithm (DZ algorithm)*

Statistic information of DZ algorithm is amount of couple of zero bits which is result of Data Difference state. Statistic information keep in register called 4DZ, 3DZ, 2DZ, 1DZ and 0 DZ. Categorization of data is in follows :

- If result of Data Difference is 0000 0010, categorize to 3DZ.
- If result of Data Difference is 0011 0100, categorize to 1DZ.
- If result of Data Difference is 0000 0100, categorize to 2DZ.
- If result of Data Difference is 0100 0010, categorize to 0DZ.

Predict state use highest frequency of DZ register to select model of prediction. After that, compare prediction model with result of Data difference. In case of prediction model is 0DZ and 1DZ, compression model will be $\{D_n\}$. If prediction model match to result of Data difference, compression model will be {0,Sub Flag, $|D_{n-1} - D_n|_{[x:0]}$}. Any things else, compression model will be $\{1, D_n\}$. Match or not match are from prediction model (Ex : 3DZ : 0000 0011, 2DZ : 0000 1111, 1DZ : 0011 1111) "and" operation with result of Data Difference, match if result of "and" operation is equal to result of Data Difference.

Example, start at initial 0DZ, 1DZ, 2DZ, 3DZ and 4DZ to 0. At the first time, model of prediction will be 0DZ (1111 1111) and assume previous data (D_{n-1}) is 1010 0001 and current data (D_n) is 1010 0111, so result of Difference Value is $|D_{n-1} - D_n|$ = 0000 0110 and Sub Flag is 1. Result of Prediction is 0DZ model and it is match to result of Difference Value (0DZ model "and" operation with difference value = difference value), so compression result will be 1010 0111 (D_n) and update statistic data (2DZ is 1, other is 0). When next data come in, prediction model will be 2DZ because of 2DZ is most frequency. If current data is 1010 1110, so result of Difference Value will be 0000 0111 and Sub Flag is 1 that mean compression result will be {0, Sub Flag, $|D_{n-1} - D_n|_{[x:0]}$} = {0 1 0111}.

B. *Single Zero algorithm (SZ algorithm)*

This algorithm is same to DZ algorithm but different on amount of statistic information. DZ algorithm have five register (0DZ, 1DZ, 2DZ, 3DZ and 4DZ) but SZ algorithm have nine (0Z, 1Z, 2Z, ... , 8Z) such as 0000 0110 it is 2DZ in DZ algorithm but 5Z in SZ algorithm. In Predict part, if result model is 0Z, 1Z or 2Z then compression result will be $\{D_n\}$. If result model is 3Z, 4Z, ... , 8Z and match to result from Data Difference then compression result will be {0, Subflag, $|D_{n-1} - D_n|_{[x:0]}$}, if not match, compression result will be {1, }.

C. *Single Zero with Average algorithm (SZAvg algorithm)*

Amount of statistical data of this algorithm is equals to SZ algorithm but difference on Predict part. DZ and SZ algorithm select statistic data which have most frequency to be model of prediction but SZAvg algorithm use average value of all statistic data solve by equation follows :

$$Average = \frac{0(0Z) + 1(1Z) + 2(2Z) + \cdots + 8(8Z)}{0Z + 1Z + 2Z + \cdots + 8Z}$$

Result will be float number, so cast it to integer by rounded down such as 4.31 rounded down to 4 and predict model is 4Z.

III. Circuit Simulation

Purpose of circuit simulation is to calculate amount of hardware resource used and use to calculate power consumption in next research. Both three algorithm have same hardware structure.

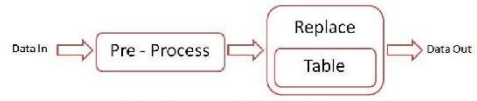


Fig. 2. Top view of circuit design.

Figure 2 Show circuit is formed by 2 modules called Pre-Process and Replace. Pre-Process use to receive data and calculate absolute of difference between D_{n-1} and D_n ($|D_{n-1} - D_n|$) including calculate Sub Flag then send to Replace which use to collect statistic information, predict and compare absolute value with predicted value.

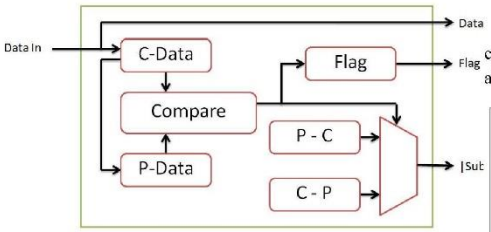


Fig. 3. Pre-Process circuit design.

From figure 3, Data In send to C-Data and go pass to next state then calculate Flag from compare C-Data and P-Data. After that, use Flag to select result of |Sub|. This module gives 3 values to next state including Data, Flag and |Sub|.

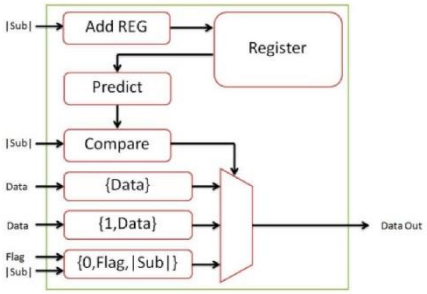


Fig. 4. Replacer circuit design.

From figure 4, after receive data from Pre-Process, Compare will compare value from Predict and |Sub| then the result of compare use to select model of result and send it out. Finally, add |Sub| to register for predict in next round.

IV. RESULT

3 levels of data called Low Change, Medium Change and Random Change are used for test. Each level has 30 groups of data and each group has 1000 byte. Each data level has different property as follows:

- Low Change : mean of variance is 36.50 and mean of difference is 11.06. This data is smooth tone of gray scale picture.
- Medium Change : mean of variance is 51.71 and mean of difference is 15.73. This data is little detail of gray scale picture.
- Random Change : mean of variance is 73.99 and mean of difference is 85.24. This data is random data.

Variance calculate from $S = \sqrt{\frac{\sum(X-\bar{X})^2}{N-1}}$ and difference calculate from $D = \frac{\sum|X_i - X_{i-1}|}{N}$. Compression ratio of each algorithm has show in figure 5.

Fig. 5. Compression Ratio of each algorithm.

Compression Ratio (CR) of each algorithm calculate from

$$CR = (1 - \frac{Numbe}{8($$

From figure 5, SZAvg algorithm is give highest CR 20.88% for Low Change, 13.04% for medium change and -0.01% for Random Change.

Circuit simulation tested by compare amount of hardware resource used of each algorithm reference to FPGA Spartan 6 XC6SLX9. Result show in figure 6 and 7.

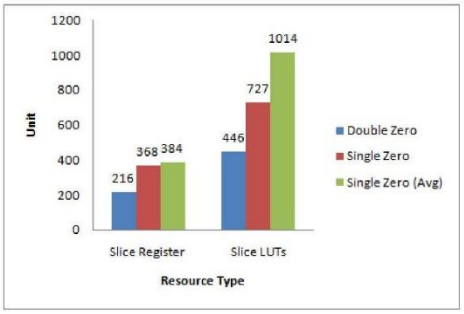


Fig. 6. Amount of Slice Resigter and Slice LUTs used.

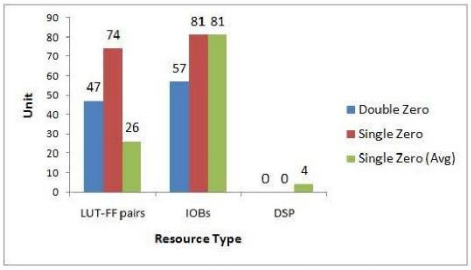


Fig. 7. Amount of LUT-FF pairs, IOBs and DSP used in each algorithm.

From figure 6 and 7, DZ algorithm use less resource and not use Data Signal Processing (DSP) but SZAvg use most hardware resource and use DSP because use divide operation for find average value.

V. CONCLUSIONS

Limit of power on WSNs is strong problem that need to be solved because of it directly affect to system life time. One way to do that is to reduce amount of data transmission because of data transmission consume most power. So data compression on WSNs is one choice that can be possible. This paper present compression algorithm which suitable for WSNs although this compression algorithm give little compression ratio as 10% – 20% but it is stability and less overhead including use little hardware resource. But this paper isn't guaranties to be reduced power consumption.

REFERENCES

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey", *Computer Networks*, vol. 38, no. 4, pp. 393–422, Mar. 2002.

[2] J. Ko, C. Lu, M. B. Srivastava, J. A. Stankovic, A. Terzis, and M. Welsh, "Wireless Sensor Networks for Healthcare", *Proceedings of the IEEE*, Nov. 2010.

[3] M. L. Kaddachi, A. Soudani, I. Nouira, V. Lecuire, and K. Torki, "Efficient hardware solution for low power and adaptive image-compression in WSN", presented at the 2010 17th IEEE International Conference on Electronics, Circuits, and Systems (ICECS), 2010, pp. 583–586.

[4] A. Chefi, A. Soudani, and G. Sicard, "Hardware compression solution based on HWT for low power image transmission in WSN", presented at the 2011 International Conference on Microelectronics (ICM), 2011, pp. 1–5.

[5] S. I. Hussian, H. Javed, W. ur Rehman, and F. N. Khalil, "CoXoH: Low cost energy efficient data compression for wireless sensor nodes using data encoding", *International Conference on Computer Networks and Information Technology (ICCNIT)*, pp. 149 – 152, Jul 2011.

[6] C. Tharini and P. V. Ranjan, "Design of Modified Adaptive Huffman Data Compression Algorithm for Wireless Sensor Network", *Journal of Computer Science*, vol. 5, no. 6, pp. 466–470, Jun. 2009.

[7] D. I. Sacaleanu, R. Stoian, and D. M. Ofrim, "An adaptive Huffman algorithm for data compression in wireless sensor networks", *Circuits and Systems (ISSCS), 2011 10th International Symposium*, pp. 1 –4, Jul. 2011.

[8] S. Rigler, W. Bishop, and A. Kennings, "FPGA-Based Lossless Data Compression using Huffman and LZ77 Algorithms", in *Electrical and Computer Engineering, 2007. CCECE 2007. Canadian Conference on*, pp. 1235 –1238, Apr. 2007.

[9] M.-B. Lin, J.-F. Lee, and G. E. Jan, "A Lossless Data Compression and Decompression Algorithm and Its Hardware Architecture", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 9, pp. 925 –936, Sep. 2006.

[10] F. Marcelloni and M. Vecchio, "A Simple Algorithm for Data Compression in Wireless Sensor Networks", *IEEE Communications Letters*, vol. 12, no. 6, pp. 411 –413, Jun. 2008.

ประวัติผู้เขียน

ชื่อ สกุล นายภควัฒน์ ติณสิริสุข

รหัสประจำตัวนักศึกษา 5510120082

วุฒิการศึกษา

วุฒิ	ชื่อสถาบัน	ปีที่สำเร็จการศึกษา
วิศวกรรมศาสตรบัณฑิต (วิศวกรรมคอมพิวเตอร์)	มหาวิทยาลัยสงขลานครินทร์	2554

การตีพิมพ์เผยแพร่ผลงาน

Pakawat Tinsirisuk and W. Suntiamorntut, "A Novel Data Compression Circuits for Wireless Sensor Networks," In Proceedings of 29th International Technical Conference on Circuit/Systems Computers and Communications (ITC-CSCC 2014), Phuket, Thailand, 1-4 July 2014, pp. 908-911.