# ไมโครโปรเซสเซอร์สำหรับประมวลผลสัญญาณเสียง

Voice Signal Processing Microprocessor


ณัฐวริศร์ เดชานุกูลกิจ

Nathvaris Dechanukunkij

วรชัย บุญชุม

Worrachai Bunchum


**ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต**

**ภาควิชาวิศวกรรมอิเล็กทรอนิกส์**

**คณะวิศวกรรมศาสตร์**

**สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง**

**พ.ศ.2563**

# ไมโครโปรเซสเซอร์สำหรับประมวลผลสัญญาณเสียง

Voice Signal Processing Microprocessor

**โดย**


**ณัฐวริศร์ เดชานุกูลกิจ**

**Nathvaris Dechanukunkij**

**วรชัย บุญชุม**

**Worrachai Bunchum**


**อาจารย์ที่ปรึกษา**

**ผศ.ดร.สุเมฆ วิศยทักษิณ**

**ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต**

**ภาควิชาวิศวกรรมอิเล็กทรอนิกส์**

**คณะวิศวกรรมศาสตร์**

**สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง**

**พ.ศ.2563**

## ใบรับรองวิชาโครงงาน 2

**ภาควิชา** วิศวกรรมอิเล็กทรอนิกส์

**คณะ** วิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

**เรื่อง** ไมโครโปรเซสเซอร์สำหรับประมวลผลสัญญาณเสียง
Voice Signal Processing Microprocessor

**ผู้จัดทำ** นายณัฐวริศร์ เดชานุกูลกิจ รหัสประจำตัว 60010330
นายวรชัย บุญชุม รหัสประจำตัว 60010889

ปริญญานิพนธ์นี้ผ่านการตรวจสอบโดยอาจารย์ที่ปรึกษาแล้ว

Sumet Wiatm

(ผศ.ดร.สุเมฆ วิศยทักษิณ)
อาจารย์ที่ปรึกษา

| | | | |
|---|---|---|---|
| **หัวข้อปริญญานิพนธ์** | ไมโครโปรเซสเซอร์สำหรับประมวลผลสัญญาณเสียง | | |
| **นักศึกษา** | นายณัฐวริศร์ | เดชานุกูลกิจ | รหัสประจำตัว 60010330 |
| | นายวรชัย | บุญชุม | รหัสประจำตัว 60010889 |
| **ปริญญา** | วิศวกรรมศาสตรบัณฑิต | | |
| **ภาควิชา** | วิศวกรรมอิเล็กทรอนิกส์ | | |
| **ปีการศึกษา** | 2563 | | |
| **อาจารย์ที่ปรึกษาโครงงาน** | ผศ.ดร.สุเมฆ วิศยทักษิณ | | |

# บทคัดย่อ


วิทยานิพนธ์นี้มีวัตถุประสงค์ 1.) เพื่อศึกษาวิธีการออกแบบไมโครโปรเซสเซอร์ประมวลผลสัญญาณเสียงบนบอร์ด FPGA ด้วยภาษา Verilog 2.) เพื่อเปรียบเทียบเวลาในการประมวลผลระหว่าง ARM-CPU และไมโครโปรเซสเซอร์ของเรา วิทยานิพนธ์นี้ประกอบด้วย 2 ส่วน

ส่วนแรกคือ Assembler เราออกแบบ Assembler โดยใช้ภาษา Python บน Visual Studio Assembler นี้สามารถทำการแปลงภาษา Assembly เป็นเลขฐาน 16 ได้ (ภาษาเครื่อง) เพื่อความสะดวกในการทดสอบไมโครโปรเซสเซอร์

ส่วนที่สองคือการออกแบบฮาร์ดแวร์ เราออกแบบไมโครโปรเซสเซอร์ตามสถาปัตยกรรมคอมพิวเตอร์พื้นฐานและและมีคำสั่งพื้นฐาน, คำสั่งเฉพาะเพื่อรองรับภาษาเครื่อง เราใช้สถาปัตยกรรม Harvard และ RISC ในการออกแบบนี้

ผลการศึกษาพบว่า 1. ) เราสามารถใช้ความเร็วสัญญาณนาฬิกา 31 MHz 2. ) ความเร็วในการประมวลผลของไมโครโปรเซสเซอร์เร็วกว่า ARM-CPU ประมาณ 3 เท่า 3. ) เข้าใจและเรียนรู้หลักการพื้นฐานของการออกแบบไมโครโปรเซสเซอร์

| | |
|---|---|
| **Thesis Title** | Voice Signal Processing Microprocessor |
| **Student** | Mr.Nathvaris Dechanukunkij Student ID 60010330 |
| | Mr.Worrachai Bunchum Student ID 60010889 |
| **Degree** | Bachelor of Engineering |
| **Program** | Electronics Engineering |
| **Year** | 2020 |
| **Project Advisor** | Asst.Prof.Dr.Sumek Wisayataksin |

# ABSTRACT

The purpose of this thesis was 1.)to study how to design Voice signal processing microprocessor on FPGA board with Verilog language. 2.)to comparison processing time between ARM-CPU and our microprocessor. This thesis consists of 2 parts.

The first part is Assembler. We design Assembler by using Python language on Visual Studio. This Assembler can provide convert Assembly language into hexadecimal code(Machine language) for our convenient to test Microprocessor.

Another part is Hardware design. We design microprocessor base-on basic Computer Architecture and having basics and specific operations to support signal processing. We use Harvard and RISC architecture in this Design.

The result of this thesis was as follows 1.)We can use clock speed 31 MHz 2.)This Microprocessor processing speed faster than ARM-CPU around 3 times 3.)We achieved our purpose and learned the principle fundamental of microprocessor design as well .

# กิตติกรรมประกาศ

ในการจัดทำปริญญานิพนธ์ไมโครโปรเซสเซอร์สำหรับประมวลผลสัญญาณเสียง สามารถดำเนินการสำเร็จลุล่วงไปได้ด้วยความกรุณาและความช่วยเหลือจากอาจารย์ที่ปรึกษา ผศ.ดร.สุเมฆ วิศยทักษิณ และอาจารย์ในภาควิชาวิศวกรรมอิเล็กทรอนิกส์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบังที่คอยให้คำปรึกษา ให้ความรู้เกี่ยวกับเรื่องการออกแบบ ให้คำแนะนำและคำชี้แนะในการแก้ไขปัญหาต่างๆตลอดการทำโครงงานนี้ ผู้จัดทำขอกราบขอบพระคุณเป็นอย่างสูงในความอนุเคราะห์จากทุกท่าน

ขอขอบพระคุณผู้ปกครองที่คอยสนับสนุน ให้ความช่วยเหลือและให้กำลังใจตลอดมา รวมทั้งรุ่นพี่และเพื่อนๆ ในภาควิชาวิศวกรรมอิเล็กโทรนิกส์และภาควิชาอื่นๆที่ให้ความช่วยเหลือในด้านต่างๆ รวมถึงให้คำปรึกษาในด้านการเขียนโปรแกรม

สุดท้ายนี้คณะผู้จัดทำหวังเป็นอย่างยิ่งว่าปริญญานิพนธ์จะเป็นประโยชน์สำหรับผู้ที่สนใจและผู้ที่หวังที่จะนำแนวคิดของปริญญานิพนธ์นี้ไปพัฒนาต่อไปในอนาคต

ณัฐวริศร์ เดชานุกูลกิจ

วรชัย บุญชุม

# สารบัญ

# สารบัญรูปภาพ

หน้า

# สารบัญตาราง

หน้า

# บทที่1

# บทนำ

## 1.1 ที่มาและความสำคัญของโครงงาน

เนื่องจากปัจจุบันเทคโนโลยีมีการพัฒนามากขึ้นอย่างก้าวกระโดด แต่ยังมีความสามารถในการรับรู้หรือการรับข้อมูลทางเสียงได้ไม่เทียบเท่ามนุษย์ ดังนั้นคณะผู้จัดทำจึงคิดที่จะทำระบบประมวลผลสัญญาณเสียงขึ้นมา เพื่อให้สามารถนำมาต่อยอดและพัฒนาโดยใช้กระบวณการที่ชื่อว่า Speech Recognition โดย Speech Recognition คือระบบโปรแกรมคอมพิวเตอร์ที่สามารถแปลงเสียงพูด (Audio File) เป็นข้อความตัวอักษร (Text) โดยสามารถแจกแจงคำพูดต่างๆ ที่มนุษย์สามารถพูดใส่ไมโครโฟน โทรศัพท์หรืออุปกรณ์อื่นๆ และเข้าใจคำศัพท์ทุกคำอย่างถูกต้องเกือบ 100% โดยเป็นอิสระจากขนาดของกลุ่มคำศัพท์ ความดังของเสียงและลักษณะการออกเสียงของผู้พูด โดยระบบจะรับฟังเสียงพูดและตัดสินใจว่าเสียงที่ได้ยินนั้นเป็นคำ ๆ ใด

เทคโนโลยีที่เป็นส่วนสำคัญในการทำเกี่ยวกับเอเอสอาร์(ASR) นั้นเรียกว่า Hidden Markov Model (HMM)เทคโนโลยีชนิดนี้สามารถที่จะเข้าใจคำพูด จากการจำแนกความแตกต่างและการประมาณการถึงความเป็นไปได้ของส่วนประกอบของหน่วยที่เป็นพื้นฐานของเสียงที่อยู่ติดๆกัน โดยอาศัยหลักการที่ว่าเสียงแต่ละเสียงจะมีขอบเขตของสัญญาณและลักษณะเฉพาะที่มีความแตกต่างกัน

## 1.2 ความมุ่งหมายและวัตถุประสงค์ของการศึกษา

1.2.1 เพื่อสร้างไมโครโปรเซสเซอร์ที่สามารถนำมาใช้ได้จริง

1.2.2 เพื่อศึกษาระบบการทำงานที่ใช้เสียงเป็นตัวสั่งการ

1.2.3 เพื่อศึกษาการจำแนกเสียงแต่ละประเภทเพื่อที่จะพัฒนาต่อไปในอนาคต

## 1.3 สมมติฐานของการศึกษา

1.3.1 มีความรู้ความเข้าใจในการทำงานของกระบวนการรับส่งข้อมูลด้วยเสียง

1.3.2 พัฒนาทักษะการเขียนระบบ Hardware

1.3.3 สามารถนำมาใช้ได้จริง

## 1.4 ขอบเขตการวิจัย

**ระยะเวลาการทำงาน**

วันที่ 11 มกราคม 2564 - วันที่ 20 พฤษภาคม 2564

**สถานที่ทำการศึกษา**

ตึกภาควิชาวิศวกรรมอิเล็กทรอนิกส์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

## 1.5 ประโยชน์ที่คาดว่าจะได้รับ

1.5.1 สามารถนำความรู้ที่ได้มาประยุกต์ใช้ในเชิงปฏิบัติได้

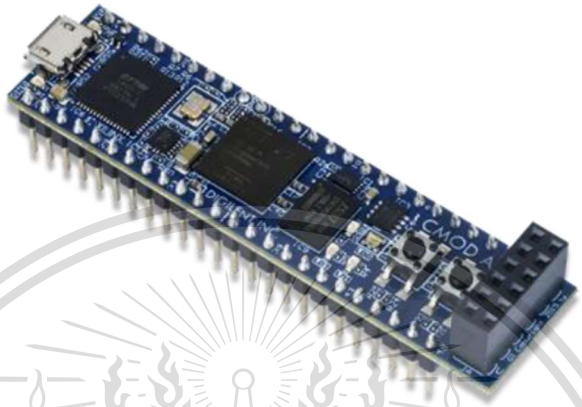
1.5.2 พัฒนาระบบการคิดวิเคราะห์และแก้ปัญหา

1.5.3 สามารถพัฒนาทักษะการเขียนโปรแกรม

# บทที่ 2

# ทฤษฎีและหลักการที่เกี่ยวข้อง

## 2.1 FPGA



รูปที่ 2.1 FPGA Board (The Cmod A7)

FPGA เป็นชิพที่ประกอบไปด้วยหน่วยเล็กๆ จำนวนมาก โดยหน่วยเล็กๆเหล่านี้ มีความสามารถพิเศษ คือ สามารถโปรแกรมให้แปลงเป็นวงจรดิจิทัลได้ ไม่ว่าจะเป็น Counter, Multiplexer, Decoder, RAM, Adder, Flip-Flop และ Gate และด้วยความที่มันมีจำนวนมากจึงเป็นจุดแข็งของชิปนี้ คือทำให้ FPGA เพียงตัวเดียวสามารถแปลงเป็นวงจรดิจิทอลได้หลากหลายและสามารถทำให้ทุกวงจรสามารถทำงานได้พร้อมกันหมด

### 2.1.1 รายละเอียด FPGA

**System Feature**

1. 512KB SRAM with an 8-bit and 8ns access times
2. 4MB Quad-SPI Flash
3. USB-JTAG Programming Circuitry
4. Powered from USB or external 3.3-5.5V supply connected to DIP pins

**System Connectivity**

USB-UART bridge

**Interaction and Sensory Devices**

1. 2 LEDs
2. 1 RGB LED
3. 2 Push buttons

**Expansion Connectors**

1.48-pin DIP connector with 44 Digital I/O and 2 Analog inputs (0-3.3V)

2.One Pmod connector with 8 Digital I/O

**2.1.2 ขั้นตอนการทำงานของ FPGA**

1.การสังเคราะห์วงจร (Logic Synthesis) ขั้นตอนนี้จะใช้ซอฟต์แวร์ในการสังเคราะห์วงจร (Synthesis Tools) ทำการสังเคราะห์พฤติกรรมวงจรที่ได้จากการออกแบบด้วย Schematic หรือ VHDL

2.การแบ่งวงจร (Partitioning) ขั้นตอนนี้เป็นการแบ่งวงจรที่ได้จากการสังเคราะห์เป็นส่วนย่อย ๆ สำหรับลงใน CLBs, IOBs หรือองค์ประกอบอื่น ๆ ภายในอุปกรณ์ FPGA

3.การวางอุปกรณ์ (Placement) ขั้นตอนนี้เป็นการเลือกทำเลที่ตั้งของแต่ละส่วนของวงจรที่ผ่านการแบ่งวงจรมาแล้วว่าควรจะอยู่ตำแหน่งไหนในอุปกรณ์ FPGA เพื่อให้ได้ผลลัพธ์ที่ดีที่สุด

4.การเชื่อมต่อสัญญาณ (Routing) ขั้นตอนนี้เป็นการเชื่อต่อสัญญาณระหว่างองค์ประกอบต่างๆ ภายในอุปกรณ์ FPGA ขั้นตอนนี้จะทำต่อเนื่องจากการวางอุปกรณ์

5.ความหน่วงด้านเวลา (Delay) ในการทำ FPGA นั้น ความหน่วงที่เกิดขึ้นเป็นความหน่วงที่เกิดจากการวางตำแหน่ง (Layout) ของอุปกรณ์ ซึ่งผู้ที่ออกแบบไม่สามารถเข้าไปแก้ไขได้ แต่สามารถทำให้น้อยที่สุดได้ ความหน่วงนั้นสามารถแยกได้เป็น 2 ประเภท คือ

5.1 ความหน่วงโลจิก (Logic Delay) เป็นความหน่วงภายในขององค์ประกอบของอุปกรณ์ FPGA

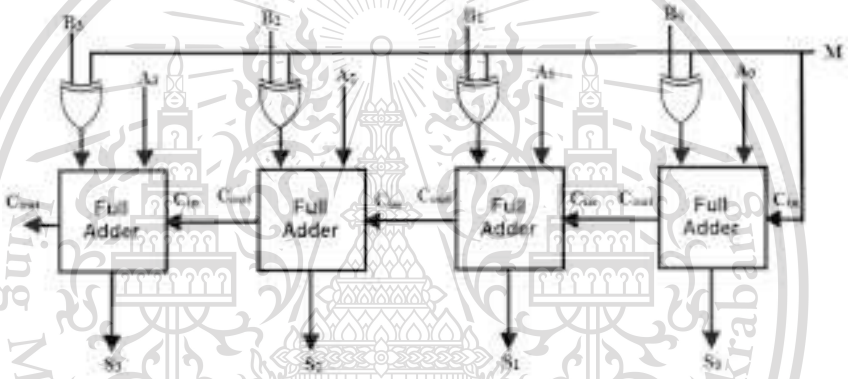
5.2 ความหน่วงที่เกิดจากการเชื่อมต่อสัญญาณ (Routing Delay) เป็นความหน่วงที่เกิดจากเชื่อมต่อสัญญาณระหว่างองค์ประกอบภายในอุปกรณ์ FPGA

6.การจำลองการทำงานของวงจร (Simulation) ขั้นตอนนี้เป็นขั้นตอนที่สำคัญเพราะเป็นขั้นตอนที่ผู้ออกแบบตรวจสอบฟังก์ชั่นการทำงานของโมเดลว่าถูกต้องหรือไม่ มีข้อผิดพลาดตรงไหนเพื่อที่จะได้ทำการแก้ไขให้ถูกต้อง

7.การโปรแกรมอุปกรณ์ FPGA (Configuration) หลังจากที่โมเดลได้ผ่านขั้นตอนต่างๆมาแล้ว ตอนนี้ก็จะสามารถที่จะดาวน์โหลดลงในอุปกรณ์ FPGA ได้แล้ว ในการดาวน์โหลดนี้ก่อนอื่นต้องแปลงแบบวงจรรวมที่ได้เป็นข้อมูลวงจร ซึ่งอยู่ในรูปของบิตสตรีม ก่อนแล้วจึงดาวน์โหลดลงไปเพื่อให้อุปกรณ์ FPGA

## 2.2 Verilog

การออกแบบฮาร์ดแวร์อิเล็กทรอนิกส์มีด้วยกันหลายรูปแบบ ทั้งการออกแบบด้วยสมการบูลีน (Boolean Equation), การออกแบบด้วยSchematic ซึ่งการออกแบบในสองรูปแบบนี้เป็นการออกแบบที่ใช้อุปกรณ์พื้นฐานมาเชื่อมต่อกันสามารถทำได้ง่ายสำหรับผู้ที่มีความรู้ความเข้าใจในวงจรพื้นฐานพอสมควร เช่น ถ้าจะออกแบบวงจรบวก ผู้ออกแบบจะต้องรู้เรื่องของวงจร Half-Adder, Full-Adder และต้องรู้ว่าต้องใช้ AND gate, OR gate, XOR gate ต่อกันอย่างไรบ้าง แต่การออกแบบๆนี้ ไม่ได้เป็นการออกแบบโดยการกำหนดความสามารถของระบบโดยตรง(System specification) อีกทั้งความสามารถในการออกแบบระบบที่ซับซ้อน Boolean equation และ Schematic ทำได้ไม่ค่อยดี เนื่องจากต้องใช้เวลามาก ถ้าจะเพิ่มสมการเข้าไปอีกเป็นพัน ๆ หมื่น ๆ สมการและยากต่อการตรวจสอบความผิดพลาด




**รูปที่ 2.2** ตัวอย่าง Schematic ของวงจร Full-Adder

ภาษา HDL (Hardware Description Language) ถูกออกแบบมาเพื่ออธิบายการทำงานของฮาร์ดแวร์อิเล็กทรอนิกส์ที่ซับซ้อน ภาษา HDL เป็นภาษาที่ใช้ในการออกแบบระบบอุปกรณ์ประเภท Programmable Logic Device (PLD), Complex Programmable Logic Device (CPLD), Field Programmable Gate Array(FPGA) ภาษา Verilog ก็เป็นหนึ่งในการออกแบบประเภทนี้เช่นกัน

การออกแบบด้วยภาษา Verilog ครอบคลุมลักษณะของการออกแบบได้เกือบทุกระดับ ทั้งด้านโครงสร้างและการทำงาน ภาษา Verilog มีข้อดีตรงที่มีความง่ายมีรูปแบบโครงสร้างภาษาคล้ายคลึงกับภาษา C ทำให้ง่ายต่อการใช้งานมีความยืนหยุ่นในการเขียนจึงได้รับความนิยมนำไปใช้ส่วนมากในงานอุตสาหกรรมฝั่งอเมริกาและญี่ปุ่น อย่างไรก็ตามภาษา Verilog ก็ยังมีข้อด้อยอยู่ตรงที่ไม่สามารถกำหนดความสามารถในการทำงานของระบบขั้นสูงได้

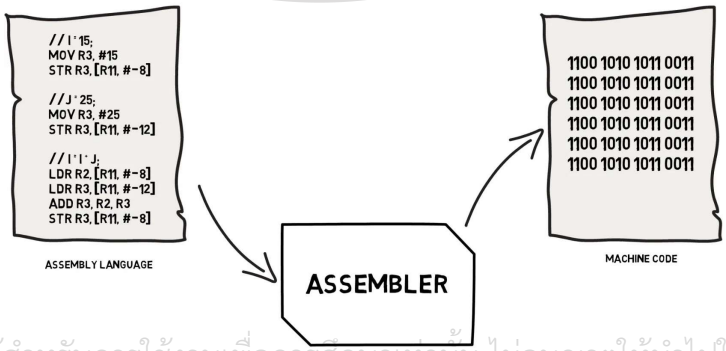
## 2.3 Assembly

ภาษาแอสเซมบลี (Assembly Language) เป็นภาษาที่มีการใช้ตัวอักษรในภาษาอังกฤษมาแทนคำสั่งที่เป็นเลขฐานสอง (0,1) และเรียกอักษรสัญลักษณ์ที่เป็นคำสั่งนี้ว่าสัญลักษณ์ข้อความ (mnemonic codes) เพื่อให้ง่ายต่อการเขียนและการจดจำมากกว่าภาษาเครื่อง ภาษาแอสเซมบลียังจัดเป็นภาษาระดับต่ำ (Low-level Language) มีการใช้สัญลักษณ์มาใช้ในการเขียนโปรแกรม เช่น

| สัญลักษณ์ | คือ | หมายถึง |
|---|---|---|
| A | Add | การบวก |
| S | Subtract | การลบ |
| C | Compare | การเปรียบเทียบ |
| MP | Multiply | การคูณ |
| ST | Store | การเก็บข้อมูล |

**รูปที่ 2.3** ตัวอย่างการใช้ภาษา Assembly

สัญลักษณ์เหล่านี้จะไม่ใช่คำที่มีความหมายในภาษาอังกฤษแต่สามารถทำให้นักเขียนโปรแกรมสามารถเขียนโปรแกรมได้สะดวกสะบายมากขึ้นเนื่องจากไม่ต้องจดจำเลข 0 และ 1 ของเลขฐานสอง อีกนอกจากนี้ภาษาแอสเซมบลียังให้ผู้เขียนใช้ตัวแปรที่ตั้งขึ้นมาเพื่อการเก็บค่าข้อมูลใดๆ เช่น X, Y, RATE หรือ TOTAL แทนการอ้างถึงตำแหน่งที่เก็บข้อมูลจริง ๆ ภายในหน่วยความจำด้วย การเขียนโปรแกรมด้วยภาษาแอสเซมบลีนั้นเมื่อนำมาใช้ในเครื่องคอมพิวเตอร์ เครื่องคอมพิวเตอร์จะไม่สามารถที่จะเข้าใจภาษาแอสเซมบลีได้จึงต้องมีการแปลภาษาแอสเซมบลีนั้นให้กลายเป็นภาษาเครื่องก่อน โดยใช้ตัวแปลภาษาแอสเซมบลีที่เรียกว่า แอสเซมเบลอร์ (Assembler) เป็นตัวแปล นอกจากนี้ผู้ที่จะเขียนโปรแกรมภาษาแอสเซมบลีได้จะต้องมีความรู้ความเข้าใจในเรื่องของฮาร์ดแวร์เป็นอย่างดีเนื่องจากจะต้องควบคุมการทำงานของหน่วยความจำหรืออุปกรณ์ภายในเครื่องคอมพิวเตอร์อื่นๆ ดังนั้นภาษาแอสเซมบลีจึงเหมาะที่จะใช้เขียนงานที่ต้องการความเร็วในการทำงานสูง เช่น งานทางด้านกราฟิก หรือ งานพัฒนาซอฟต์แวร์ ระบบต่าง ๆ

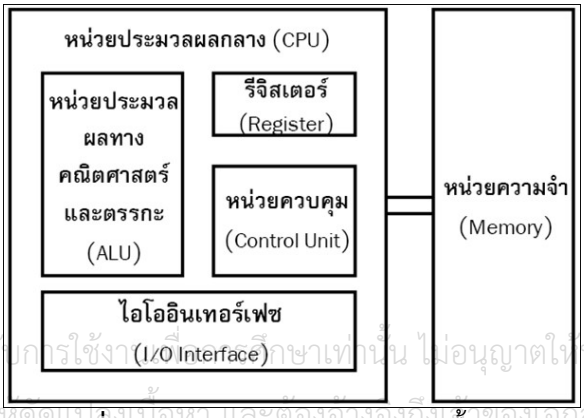



**รูปที่ 2.4** ตัวอย่างการแปลง Assembly ไปเป็น ภาษาเครื่อง

## 2.4 Machine cycle

การทำงานของคอมพิวเตอร์จะต้องทำตามโปรแกรมที่กำหนดไว้ในหน่วยความจำ โดยโปรแกรมเกิดจากการนำคำสั่งมาต่อเรียงกันเมื่อคอมพิวเตอร์ทำงานหน่วยควบคุมทำการอ่านคำสั่งต่างๆ เข้ามาประมวลผลในซีพียูโดยการทำคำสั่งของซีพียูประกอบด้วยขั้นตอนการทำงาน 3 ขั้นตอน ดังนี้

1. ขั้นตอนการรับเข้าข้อมูล (fetch) เริ่มแรกหน่วยควบคุมรับรหัสคำสั่งและข้อมูลที่จะประมวลผลจากหน่วยความจำ
2. ขั้นตอนการถอดรหัส (decode) เมื่อรหัสคำสั่งเข้ามาอยู่ในซีพียูแล้วหน่วยควบคุมจะถอดรหัสคำสั่งแล้วส่งคำสั่งและข้อมูลไปยังหน่วยคำนวณและตรรกะ
3. ขั้นตอนการทำงาน (execute) หน่วยคำนวณและตรรกะทำการคำนวณโดยใช้ข้อมูลที่ได้รับมาถอดรหัสคำสั่งและทราบแล้วว่าต้องทำอะไร ซีพียูก็จะทำตามคำสั่งนั้นและนำผลลัพธ์ที่ได้ไปเก็บไว้ในหน่วยความจำ ซีพียุคเก่าการทำคำสั่งแต่ละคำสั่งจะต้องทำวงรอบคำสั่งให้จบก่อนจากนั้นจึงทำวงรอบคำสั่งของคำสั่งต่อไป สำหรับซีพียูในยุคปัจจุบันได้มีการพัฒนาให้ทำงานได้เร็วขึ้น โดยมีการแบ่งวงรอบคำสั่งนี้เป็นวงรอบย่อย ๆ อีก มีการนำเทคนิคการทำงานแบบสายท่อ (pipeline) มาใช้ โดยขณะที่ทำวงรอบคำสั่งแรกอยู่ ก็มีการอ่านรหัสคำสั่งของคำสั่งถัดไปเข้ามาด้วย ซึ่งจะทำให้การทำงานโดยรวมของซีพียูเร็วขึ้นมาก หน่วยควบคุม (control unit) เป็นหน่วยที่ทำหน้าที่ประสานงาน และควบคุมการทำงานของคอมพิวเตอร์ ควบคุมให้อุปกรณ์รับและส่งข้อมูลไปที่หน่วยความจำ ให้ติดต่อกับอุปกรณ์แสดงผลเพื่อสั่งให้นำข้อมูลจากหน่วยความจำไปยังอุปกรณ์แสดงผล โดยหน่วยควบคุมของคอมพิวเตอร์จะแปลความหมายของคำสั่งในโปรแกรมของผู้ใช้ และควบคุมให้อุปกรณ์ต่างๆ ทำงานตามคำสั่งนั้นๆ หน่วยคำนวณและตรรกะ (Arithmetic-Logic Unit : ALU) เป็นหน่วยที่ทำหน้าที่ในการคำนวณต่างๆ ทางคณิตศาสตร์ ได้แก่ การบวก ลบ คูณ หาร และเปรียบเทียบทางตรรกะ เพื่อทำการตัดสินใจ เช่น การเปรียบเทียบข้อมูล การเปรียบเทียบว่าจริงหรือเท็จ ซึ่งการทำงานของ ALU จะรับข้อมูลจากหน่วยความจำมาไว้ในที่เก็บชั่วคราวของเอแอลยูที่เรียกว่ารีจิสเตอร์ (register) เพื่อทำการคำนวณแล้วส่งผลลัพธ์กลับไปยังหน่วยความจำ




**รูปที่ 2.5** CPU Architecture แบบพื้นฐาน

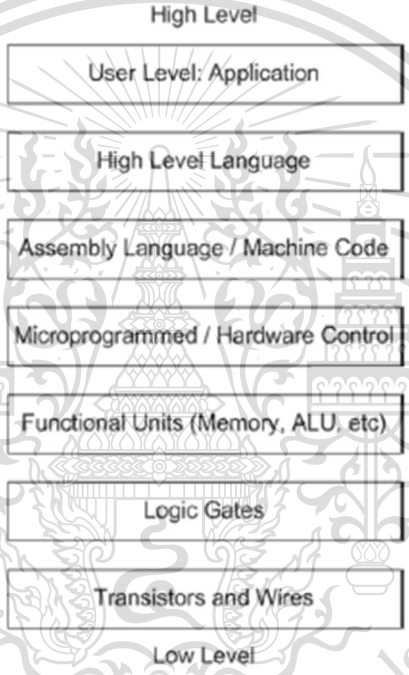
## 2.5 Computer Architecture

สถาปัตยกรรมคอมพิวเตอร์คอมพิวเตอร์หมายถึงโครงสร้างของระบบคอมพิวเตอร์ที่โปรแกรมของระบบจะต้องเข้าใจในภาษาเครื่องเพื่อเขียนโปรแกรมในเครื่องทำงานได้อย่างถูกต้อง ประกอบไปด้วย

- รีจิสเตอร์
- หน่วยความจำ
- ชุดคำสั่ง

### 2.5.1 โครงสร้างของคอมพิวเตอร์

ภาษาคอมพิวเตอร์สามารถแบ่งได้เป็น 7 ระดับ




**รูปที่ 2.6** ระดับภาษาคอมพิวเตอร์

**2.5.1.1 ระดับผู้ใช้งานหรือแอปพลิเคชัน(User or Application Program Level)**

เป็นภาษาระดับที่ผู้ใช้งานสามารถโต้ตอบกับคอมพิวเตอร์ผ่านโปรแกรมต่างๆ

**2.5.1.2 ภาษาระดับสูง (High-Level Language Level)**

เป็นระดับของผู้พัฒนาโปรแกรม (Programmer) โดยใช้ตัวแปลภาษา(**Compiler**) ที่ทำหน้าที่แปลจากภาษาระดับสูงเป็นภาษาเครื่อง(Machine Code) เพื่อนำไปใช้กับคอมพิวเตอร์ เช่น ภาษาซี ปาสคาล หรือจาวา

**2.5.1.3 ภาษาแอสเซ็มบลีหรือภาษาเครื่อง(Assembly language or Machine code level)**

โดยภาษาเครื่องจะอยู่ในรูปแบบของ Binary code เช่น 1111001 แทนการบวก เป็นรูปแบบที่จดจำยาก ต่อมามีผู้พัฒนา Assembly โดยมีตัวแปรภาษาคือ Assembler เพื่อให้ใช้งานได้สะดวกยิ่งขึ้น ตัวอย่างภาษาเช่น

| Assemble language | Machine language | ความหมาย |
|---|---|---|
| ADD | 00111001 | การบวก |
| MOVE | 00010110 | การย้ายค่า |

**ตารางที่ 2.1** ตัวอย่างภาษา Assembly

**2.5.1.4 ระดับควบคุม(Control Level)**

เป็นระดับของสัญญาณควบคุมที่มีผลต่อการถ่ายโอนข้อมูลระหว่างรีจิสเตอร์เป็นลักษณะของคำสั่งในแบบของไมโครโปรแกรม โดยไมโครโปรแกรมเป็นภาษาระดับต่ำเพื่อควบคุมฮาร์ดแวร์

**2.5.1.5 ระดับหน่วยฟังก์ชั่น (Function Unit Level)**

เป็นภาษาการถ่ายโอนข้อมูลระหว่างรีจิสเตอร์ที่หน่วยควบคุมย้ายเข้าออกจากหน่วยฟังก์ชั่นเช่น การบวก ลบ คูณ หาร รวมถึงรีจิสเตอร์ภายในซีพียู ALU และหน่วยความจำของเครื่อง

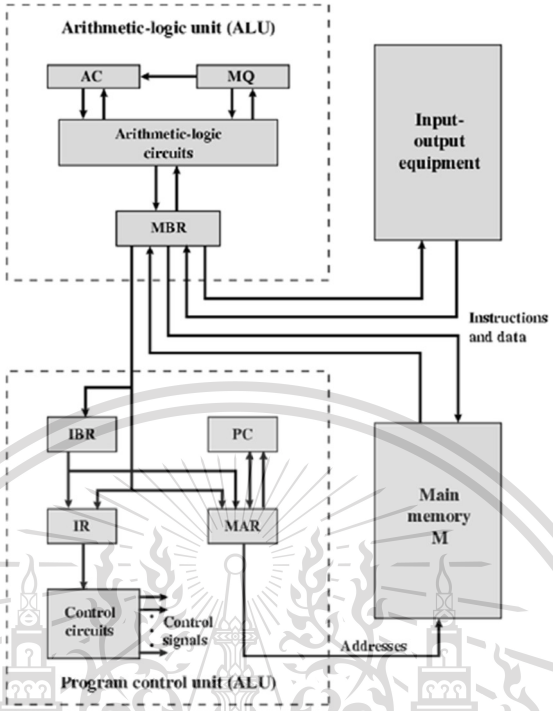
**2.5.1.6 ระดับลอจิกเกตและทรานซิสเตอร์ (Logic gate and Transistors Level)**

เป็นระดับในส่วนของฮาร์ดแวร์ของเครื่องที่อาศัยกระแสไฟฟ้า ความต้านทาน

**2.5.2 โครงสร้างของคอมพิวเตอร์**

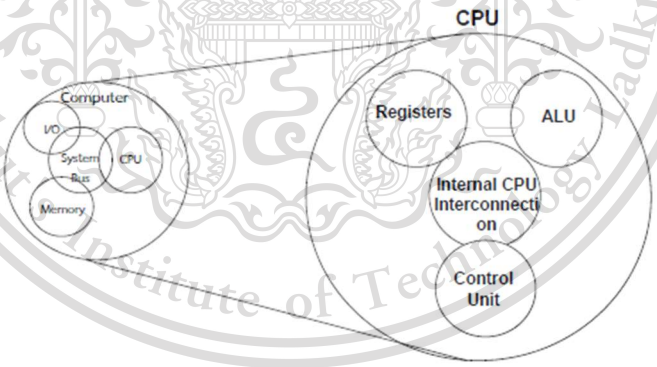
- **Accumulator (AC), Multiplier quotient (MQ)** ใช้เก็บข้อมูลหรือผลลัพธ์ที่ได้จากการทำงานของ ALU
- **Memory Buffer Register (MBR)** ทำหน้าที่เก็บข้อมูลขนาดประมาณ 1 เวิร์ดที่บันทึกลงในหน่วยความจำหรืออ่านข้อมูลจากหน่วยความจำ
- **Instruction Buffer Register (IBR)** ทำหน้าที่เก็บคำสั่งทางฝั่งขวาของแต่ละเวิร์ดเพื่อรอการประมวลผล(เก็บข้อมูลชั่วคราว)
- **Program Counter (PC)** จัดเก็บที่อยู่ข้อมูล (Address) ของคำสั่งต่อไปที่จะถูกนำมาประมวลผล
- **Instruction Register (IR)** จัดเก็บ Opcode ของคำสั่งที่กำลังจะประมวลผล

- **Memory Address Register (MAR)** เก็บตำแหน่งที่อยู่ของข้อมูลที่กำลังจะถูกบันทึกหรืออ่านข้อมูลมาเก็บใน MBR




**รูปที่ 2.7** โครงสร้างของคอมพิวเตอร์

### 2.5.3 CPU (Central Processing Unit)




**รูปที่ 2.8** โครงสร้างภายใน CPU

CPU เปรียบเสมือนเป็นสมองของเครื่องคอมพิวเตอร์โดยทำหน้าที่ในการคำนวณค่าต่างๆ ตามคำสั่งที่ได้รับและควบคุมการทำงานของส่วนประกอบอื่นๆทั้งหมดในระบบไมโครคอมพิวเตอร์

- หน่วยคำนวณทางคณิตศาสตร์และตรรกะ (Arithmetic and Logic Unit: ALU) เป็นหน่วยที่ทำหน้าที่ประมวลผลโดยใช้วิธีคณิตศาสตร์ เช่นบวก ลบ คูณ หาร หรือ

ทำหน้าที่ประมวลผลทางตรรกะ เช่น AND OR NOT COMPLEMENT รวมถึงเปรียบเทียบค่าต่างๆ

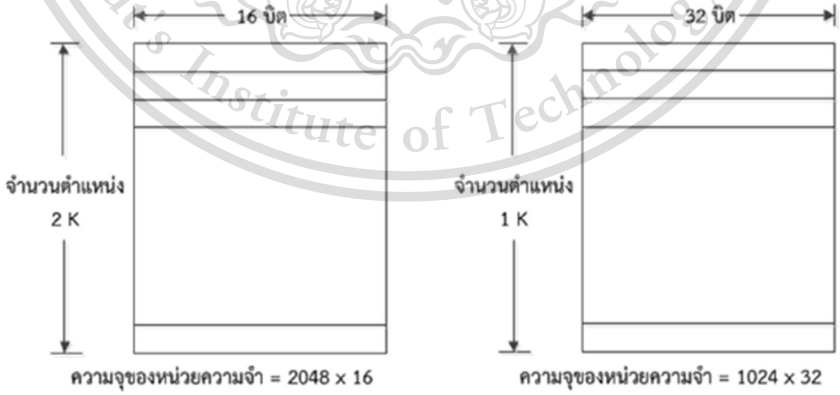
- หน่วยเก็บข้อมูลชั่วคราว (Register) เป็นหน่วยความจำขนาดเล็ก ทำหน้าที่เป็นที่จัดเก็บข้อมูลชั่วคราวก่อนที่จะประมวลผล การอ้างถึงข้อมูลของ Register จะมีความเร็วเท่ากับความเร็วของหน่วยประมวลผลกลางเพราะเป็นหน่วยความจำส่วนที่อยู่ภายในหน่วยประมวลผลกลางจึงไม่ต้องไปอ้างถึงภายนอกประมวลผล
- หน่วยควบคุม (Control Unit) ทำหน้าที่ควบคุมระบบการทำงานของคอมพิวเตอร์ทั้งหมด กำหนดจังหวะการทำงานต่างๆของคอมพิวเตอร์และส่วนประกอบอื่น ๆ ของ CPU และทำหน้าที่ควบคุมการส่งข้อมูลระหว่างหน่วยต่างๆในคอมพิวเตอร์

### 2.5.4 Memory

#### 2.5.4.1 หน่วยความจำหลัก (Main Memory)

ทำหน้าที่จัดเก็บ Instruction ข้อมูลหรือโปรแกรมเพื่อที่จะถูกเรียกใช้งานจาก CPU โดยจะมีการจัดพื้นที่และตำแหน่งสำหรับจัดเก็บ Instruction และข้อมูลโดยแต่ละตำแหน่งจะมีค่า Address ที่ไม่ซ้ำกัน

ตำแหน่ง (Address) ของหน่วยความจำถูกนำมาใช้เพื่ออ้างอิงถึงตำแหน่งของข้อมูลที่เก็บในหน่วยความจำหลักขนาดของความจุของหน่วยความจำหาได้จากตำแหน่งคูณกับความยาวของ Word เช่น หน่วยความจำหลักมี 16 ตำแหน่งและความยาว Word มีขนาด 4 bit เพราะฉะนั้นความจุรวมของหน่วยความจำหลักหาได้จาก 16 * 4 เท่ากับ 64 bit หรือ 8 byte การอ้างอิงถึงตำแหน่งของหน่วยความจำจะถูกระบุด้วย Address ที่แตกต่างกัน จำนวณ Address บนหน่วยความจำเป็น $2^n$ ตำแหน่ง

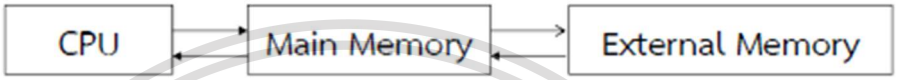



**รูปที่ 2.9** ตัวอย่างขนาดของหน่วยความจำ

- **Ram (Random Access Memory)** เป็นหน่วยความจำที่ใช้จัดเก็บข้อมูลชั่วคราว สามารถลบและเขียนข้อมูลใหม่ได้ สามารถจัดเก็บข้อมูลได้ในขณะที่มีไฟฟ้าหล่อเลี้ยงเท่านั้นและถูกนำมาใช้เป็นหน่วยความจำหลักของคอมพิวเตอร์

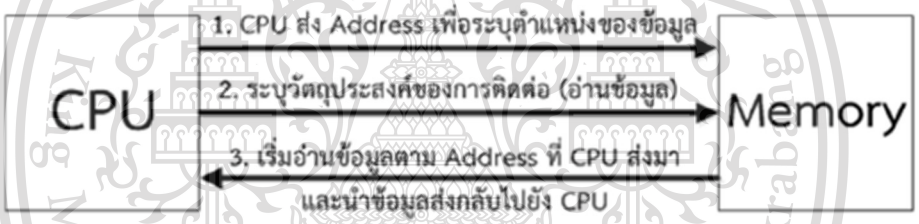
- **ROM (Read Only Memory)** เป็นหน่วยความจำที่ไม่สามารถบันทึกข้อมูลลงใหม่ลงไปได้ แต่สามารถจัดเก็บข้อมูลโดยไม่จำเป็นต้องมีไฟฟ้าหล่อเลี้ยง ดังนั้น ROM จึงถูกนำมาใช้จัดเก็บชุดคำสั่ง ROM Bootstrap โดย ROM Bootstrap ใช้เพื่อยบอกให้ CPU ทราบว่าตอนเปิดเครื่องต้องเริ่มต้นการทำงานด้วยคำสั่งใดบ้าง โดยปกติชุดคำสั่งจะถูกบันทึกมาจากโรงงาน

**2.5.4.2 หน่วยความจำภายนอก (External Memory)** เป็นหน่วยความจำสำรอง (Secondary Memory) ใช้จัดเก็บข้อมูลเหมือนหน่วยความจำหลักแต่ CPU ไม่สามารถเข้าถึงหน่วยความจำภายนอกได้โดยตรง

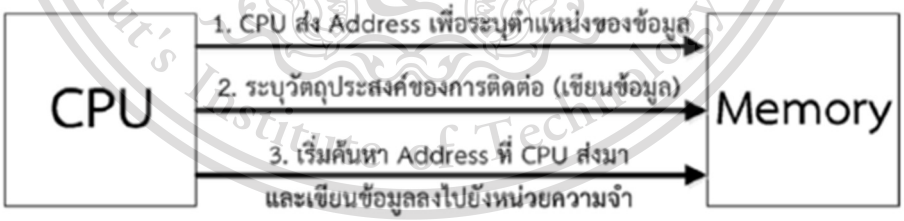



**รูปที่ 2.10** การติดต่อระหว่าง CPU และหน่วยความจำภายนอก

**2.5.5 การติดต่อระหว่าง CPU และหน่วยความจำ**



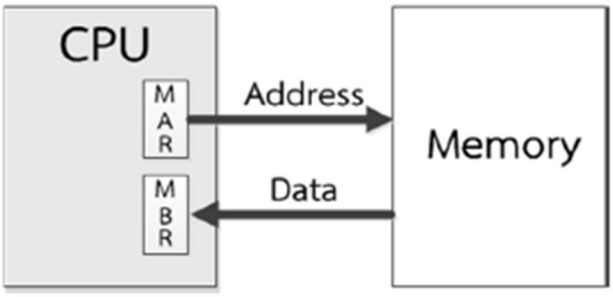

**รูปที่ 2.11** รูปแสดงการอ่านข้อมูลบนหน่วยความจำ




**รูปที่ 2.12** รูปแสดงการเขียนข้อมูลลงบนหน่วยความจำ

**2.5.6 การติดต่อระหว่าง CPU และ Register**

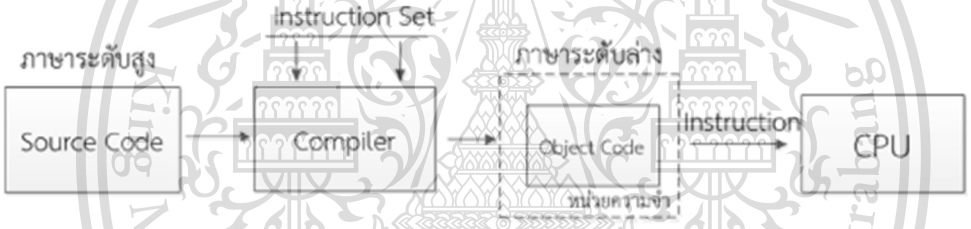
CPU จะใช้ Register ซึ่งเป็นหน่วยความจำชั่วคราวที่อยู่ใน CPU 2 ตัวคือ MAR และ MBR เพื่ออ่านเขียนข้อมูลลงบนหน่วยความจำ โดย CPU จะนำค่า Address ที่ต้องการเก็บลงใน MAR ก่อนนำไปชี้ตำแหน่งบนหน่วยความจำ แล MBR จะใช้เก็บข้อมูลจาก CPU ในกรณีการเขียนข้อมูลหรือเก็บข้อมูลจากหน่วยความจำในกรณีของการอ่านข้อมูล




**รูปที่ 2.13** รูปแสดงการติดต่อระหว่าง CPU และ Register

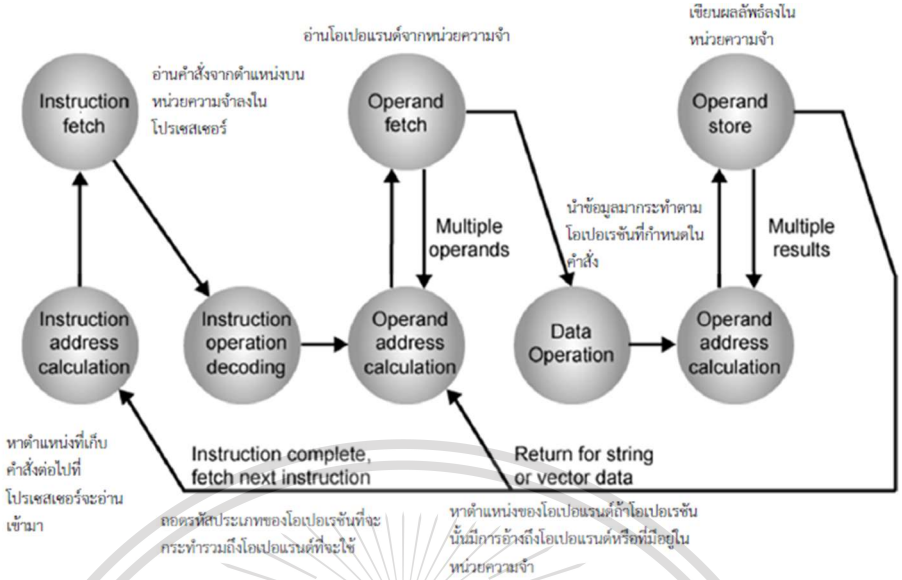
## 2.6 Instruction Set

คือชุดคำสั่งที่ใช้เป็นตัวบอกขั้นตอนการทำงานแก่ CPU และ CPU จะทำตาม Instruction นั้น ภายใน Instruction set นั้นจะประกอบด้วย Instruction มากกว่าหนึ่งตัวตัวอย่างได้แก่ ADD, SUBSTRACT, MULTIPLY, MOVE เป็นต้น หน้าที่ของ Instruction set จะช่วยตัวคอมไพร์เลอร์ทำการแปลง Source Code ของโปรแกรมที่เป็นภาษาระดับสูงให้เป็น Object Code ไปเก็บลงในหน่วยความจำหลักเพื่อรอส่งชุดคำสั่งที่ต้องการประมวลผลไปยัง CPU




**รูปที่ 2.14** การแปลง Instruction set ก่อนจะนำมาเข้า CPU

### 2.6.1 Instruction Cycle




**รูปที่ 2.15** Diagram ของวัฏจักรคำสั่ง

ในการอ่านคำสั่ง โปรเซสเซอร์อาจจะอ่านจากหน่วยความจำโดยตรงแต่โดยทั่วไปแล้ว โปรเซสเซอร์จะอ่านตำแหน่งที่เก็บคำสั่งต่อไปจากโปรแกรมเคาเตอร์ (Program Counter: PC) ตำแหน่งเหล่านี้จะอยู่บนหน่วยความจำแล้วเอ็กซิคิวต์ตามคำสั่งที่อ่านเข้ามาซึ่ง PC จะทำการเพิ่มค่าขึ้นเรื่อย ๆ ทำให้โปรเซสเซอร์ทำงานเรื่อยไป

- **Instruction Address Calculate (IAC)** หาตำแหน่งที่เก็บคำสั่งต่อไปที่โปรเซสเซอร์ที่จะอ่านเข้ามา
- **Instruction Fetch (IF)** อ่านคำสั่งจากตำแหน่งบนหน่วยความจำลงในโปรเซสเซอร์
- **Instruction Operation Decoding (IOD)** วิเคราะห์คำสั่งเพื่อพิจารณาประเภทของโอเปอเรชั่นที่จะกระทำ รวมทั้งโอเปอแรนด์ที่จะใช้
- **Operand Address Calculate (OAC)** หาตำแหน่งของโอเปอแรนด์ ถ้า Operation นั้นมีการอ้างถึงโอเปอแรนด์หรือที่มีอยู่ในหน่วยความจำผ่านทางอุปกรณ์ Input / Output
- **Operand Fetch (OF)** อ่านโอเปอแรนด์จากหน่วยความจำหรืออ่านจากอุปกรณ์ Input / Output
- **Data Operation (DO)** นำข้อมูลมากระทำตาม Operation ที่กำหนดในคำสั่ง
- **Operand Store (OS)** เขียนผลลัพธ์ลงในหน่วยความจำหรือส่งออกไปทางอุปกรณ์ Input / Output

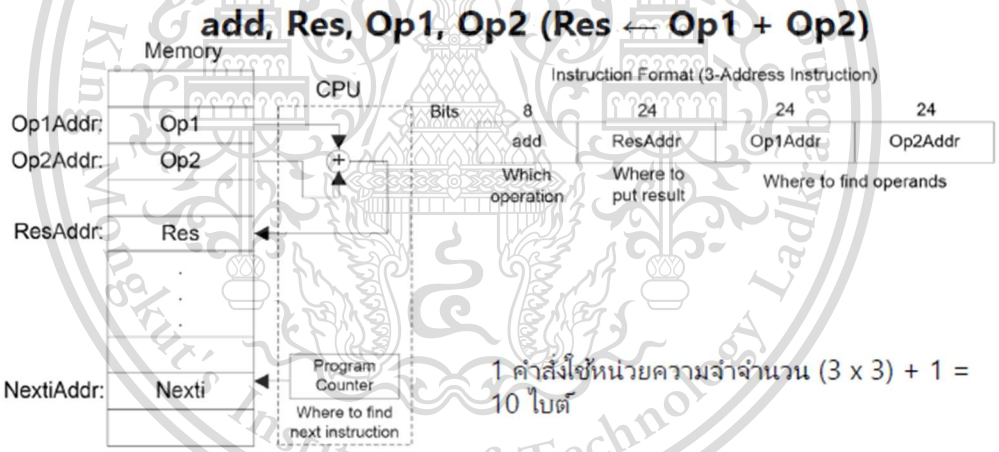
**2.6.2 การออกแบบชุดคำสั่ง**

ชุดคำสั่งคือสิ่งที่โปรแกรมเมอร์ใช้ควบคุมซีพียู ดังนั้นการออกแบบชุดคำสั่งเป็นงานที่มีความซับซ้อนเนื่องจากมีผลกระทบกับระบบคอมพิวเตอร์ จะพิจารณาจาก

- **Operation repertory:** จำนวน Operation ที่มีให้เลือกใช้รวมทั้งความซับซ้อนของ Operation ที่ควรจะเป็น
- **Data type:** ความหลากหลายของประเภทข้อมูลที่ทำ Operation
- **Instruction format:** ความยาวของคำสั่ง (เป็นบิต) จำนวน Address ขนาดของฟิลด์และอื่นๆ
- **Register:** จำนวนรีจิสเตอร์ที่คำสั่งสามารถอ้างอิงและใช้ประโยชน์ได้
- **Addressing:** การกำหนดโหมดของ Address สำหรับ Operand

**2.6.2.1 รหัสคำสั่งแบบ 3 – Address Instruction**

เป็นการอ้าง Address แบบ 3 ตำแหน่งโดยเพิ่ม Program Counter (PC) เพื่อใช้ชี้ Address ที่เก็บคำสั่งถัดไป โดยใช้ข้อมูลจำนวน 24 bit

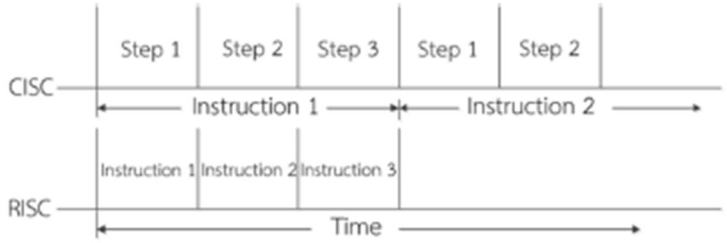



**รูปที่ 2.16** รูปแบบการใช้คำสั่งแบบ 3 Address Instruction

**2.6.2.2 สถาปัตยกรรมแบบ CISC และ RISC**

สถาปัตยกรรมของไมโครโปรเซสเซอร์หากแบ่งตามชุดคำสั่งสามารถแบ่งได้เป็น 2 กลุ่ม คือแบบ CISC (Complex Instruction Set Computer) และแบบ RISC (Reduced Instruction Set Complex) ในไมโครโปรเซสเซอร์รุ่นเก่าๆ จะเป็นแบบ CISC ทั้งสิ้นแต่ในปัจจุบันจะแยกความแตกต่างระหว่างสถาปัตยกรรมทั้ง 2 แบบได้ยากเนื่องจากชุดคำสั่งทั้งสองถูกออกแบบให้มีลักษณะใกล้เคียงกัน ในการทำคำสั่งแบบ CISC จะมีกระบวนการย่อย ๆ ประกอบอยู่ภายในแต่ละคำสั่งจะมีขั้นตอนการทำงานที่ไม่เท่ากันได้และมีขนาด

ของรหัสคำสั่งที่เท่ากันได้ ส่วนคำสั่งแบบ RISC ทุกคำสั่งของไมโครโปรเซสเซอร์จะมีความยาวของรหัสคำสั่งเท่ากันหมด



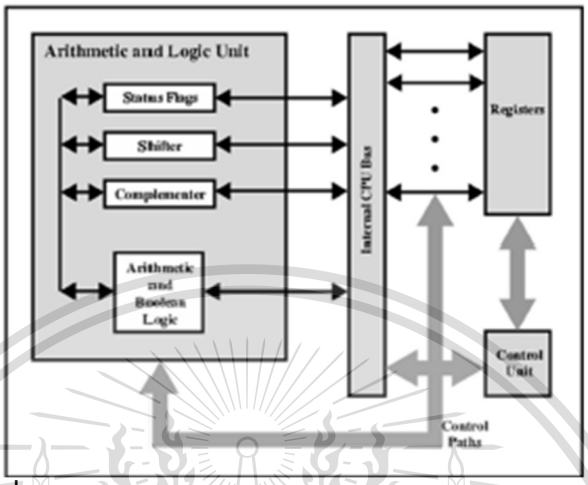

รูปที่ 2.17 รูปแบบของสั่งของ CISC และ RISC

▪ สถาปัตยกรรมแบบ CISC ออกแบบให้ไมโครโปรเซสเซอร์มีชุดคำสั่งเป็นจำนวนมาก บางคำสั่งสามารถทำงานที่ซับซ้อนได้และบางคำสั่งสามารถอ้างแอดเดรสได้หลายแบบเพื่อให้นักโปรแกรมสามารถเขียนโปรแกรมได้ง่าย การอ้างแอดเดรสได้หลายแบบจะทำให้บางคำสั่งทำงานได้หลากหลาย แต่ก็ทำให้คำสั่งของไมโครโปรเซสเซอร์มีความยาวแตกต่างกันได้ ทำให้โครงสร้างภายในของไมโครโปรเซสเซอร์มีจำเป็นต้องมีรีจิสเตอร์เป็นจำนวนมาก การเพิ่มจำนวนรีจิสเตอร์จะทำให้โครงสร้างภายในมีความหนาแน่นของวงจรซับซ้อนยิ่งขึ้น คำสั่งของไมโครโปรเซสเซอร์ 80*86 โดยทั่วไปจะมีความยาวอยู่ระหว่าง 2-6 byte ส่วนคำสั่งที่ทำงานยากๆจะใช้หน่วยความจำถึง 13 byte เวลาที่ใช้แต่ละคำสั่งจะใช้เวลามากจึงมีการแก้ปัญหาโดยนำการทำงานแบบไปป์ไลน์มาใช้ แต่ก็เกิดปัญหาใหม่เมื่อมีการทำคำสั่งบางคำสั่งที่มีความยาวไม่เท่ากัน

▪ สถาปัตยกรรมแบบ RISC ขนาดของแต่ละคำสั่งจะมีขนาดเท่ากันทำให้ไปป์ไลน์สามารถช่วยเพิ่มประสิทธิภาพในการทำงานของไมโครโปรเซสเซอร์ได้ดีขึ้น โดยปกติจะออกแบบชุดคำสั่งให้มีความยาวเท่ากับขนาดของบัสข้อมูลเช่นระบบบัสแบบ 32 บิต ก็จะออกแบบคำสั่งให้มีความยาว 32 บิต แต่ละคำสั่งจะทำงานเพียงขั้นตอนเดียวและต้องทำงานเสร็จภายในสัญญาณนาฬิกา 1 ลูกและสำหรับการติดต่อกับหน่วยความจำภายนอกเพื่อดึงข้อมูลหรือเก็บข้อมูลจะใช้คำสั่ง Load และ Store เท่านั้นส่วนคำสั่งอื่นๆจะเป็นการกระทำระหว่างรีจิสเตอร์ด้วยกัน

## 2.7 โครงสร้างภายในไมโครโปรเซสเซอร์

โครงสร้างภายในของไมโครโปรเซสเซอร์จะมีความคล้ายคลึงกับโครงสร้างภายในของคอมพิวเตอร์ ภายในตัวซีพียูจะมีบัสภายใน (Internal CPU Bus) สำหรับการถ่ายโอนข้อมูลระหว่าง Register ต่าง ๆ โดย ALU จะทำงานกับข้อมูลที่อยู่ภายในตัว CPU เท่านั้น




**รูปที่ 2.18** แบบจำลองโครงสร้างภายในไมโครโปรเซสเซอร์

### 2.7.1 รีจิสเตอร์สำหรับการควบคุมและรายงานสถานการณ์ทำงาน

**1. Program Counter (PC)** เป็นรีจิสเตอร์ที่เก็บที่อยู่ของคำสั่งที่จะถูกประมวลผลลำดับถัดไป

**2. Instruction Register (IR)** เป็นรีจิสเตอร์ที่ใช้เก็บคำสั่งเครื่องที่พึ่งถูกอ่านเข้ามาในซีพียู

**3. Memory Address Register (MAR)** เป็นรีจิสเตอร์ใช้เก็บตำแหน่งที่อยู่ที่อ้างถึงในหน่วยความจำหลัก

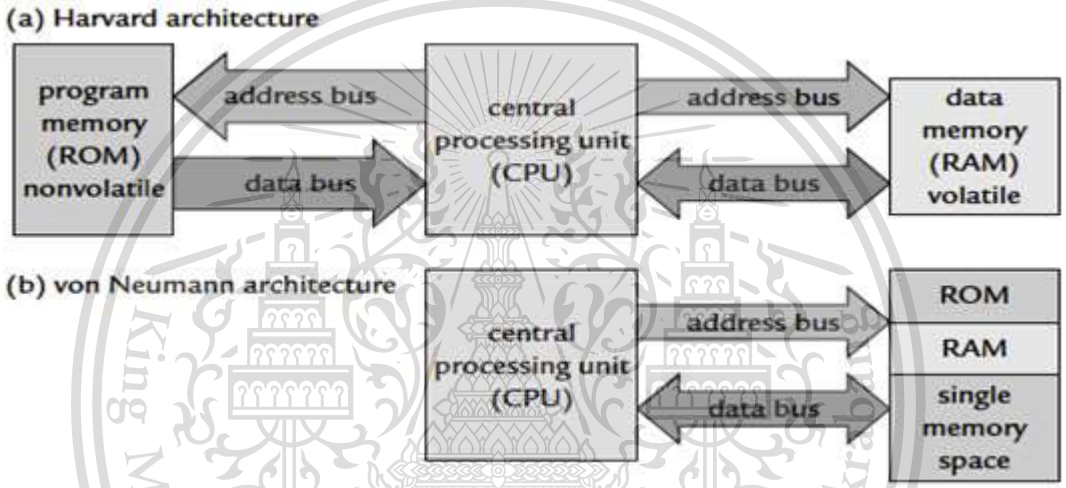
**4. Memory Buffer Register (MBR)** เป็นรีจิสเตอร์ที่ใช้เก็บข้อมูลเพื่อเตรียมบันทึกลงในหน่วยความจำหลักหรือใช้เก็บข้อมูลขนาด 1 เวิร์ดล่าสุดที่ถูกอ่านเข้ามาใน CPU

โดยทั่วไปซีพียูจะปรับปรุงค่าในรีจิสเตอร์ Program Counter ทันทีภายหลังที่ได้อ่านคำสั่งเข้ามา (Instruction fetch) เพื่อให้รีจิสเตอร์ PC นี้ชี้ตำแหน่งคำสั่งจะถูกอ่านเข้ามาในลำดับถัดไปต่อเสมอโดยคำสั่งประเภท branch หรือ skip จะมีผลให้เกิดการเปลี่ยนแปลงค่าในรีจิสเตอร์นี้ คำสั่งที่ถูกอ่านเข้ามาจะถูกนำไปจัดเก็บไว้ที่ IR ซึ่งจะถูกนำไปวิเคราะห์รหัสดำเนินงาน (Opcode) และกำหนดตัวค่าถูกกระทำ (Operand) ซึ่งอาจมีการแลกเปลี่ยนข้อมูลกับหน่วยความจำหลักผ่าน MAR และ MBR ในระบบคอมพิวเตอร์ที่ใช้บัส MAR จะเชื่อมต่อโดยตรงกับ Address Bus และ MBR จะเชื่อมต่อโดยตรงกับ Data Bus

## 2.8. Processor Architecture

สถาปัตยกรรมแบบ Harvard จะแยกหน่วยความจำออกเป็นสองส่วนอย่างชัดเจน ได้แก่ หน่วยความจำโปรแกรมและหน่วยความจำข้อมูล โดยข้อกำหนดของสถาปัตยกรรมนี้จะกำหนดให้เก็บแอปพลิเคชันโปรแกรมและระบบปฏิบัติการไว้ในหน่วยความจำเท่านั้น ดังในรูป 2.19(a)

สถาปัตยกรรม Von-Neumann เป็นสถาปัตยกรรมที่ไม่ได้แบ่งหน่วยความจำของข้อมูลและโปรแกรมออกจากกัน ซึ่งสถาปัตยกรรมนี้อนุญาติให้สามารถเปลี่ยนแปลง code โปรแกรมขณะรันแอปพลิเคชันได้โดยในสถาปัตยกรรมนี้เราจะเก็บแอปพลิเคชันโปรแกรมระบบปฏิบัติการและข้อมูลที่ประมวลผลไว้ในหน่วยความจำหลักไว้ที่ใดก็ได้ดังในรูป 2.20(b)




**รูปที่ 2.19** ความแตกต่างระหว่าง Harvard และ Von Neumann Architecture

## 2.9. Python

Python เป็นภาษาเขียนโปรแกรมระดับสูงที่ใช้กันอย่างกว้างขวางในการเขียนโปรแกรมสำหรับวัตถุประสงค์ทั่วไป ภาษา Python นั้นสร้างโดย Guido van Rossum และถูกเผยแพร่ครั้งแรกในปี 1991 Python นั้นเป็นภาษาแบบ interpreter ที่ถูกออกแบบโดยมีแนวคิดที่จะทำให้โค้ดอ่านได้ง่ายขึ้นและโครงสร้างของภาษานั้นจะทำให้โปรแกรมเมอร์สามารถเข้าใจแนวคิดการเขียนโค้ดโดยใช้บรรทัดที่น้อยลงกว่าภาษาอย่าง C++ และ Java ซึ่งภาษานั้นถูกกำหนดให้มีโครงสร้างที่ตั้งใจให้การเขียนโค้ดเข้าใจง่ายทั้งในโปรแกรมขนาดเล็กไปจนถึงโปรแกรมขนาดใหญ่

Python นั้นมีคุณสมบัติเป็นภาษาเขียนโปรแกรมแบบไดนามิกส์และมีระบบการจัดการหน่วยความจำอัตโนมัติและสนับสนุนการเขียนโปรแกรมหลายรูปแบบที่ประกอบไปด้วยการเขียนโปรแกรมเชิงวัตถุ imperative การเขียนโปรแกรมแบบฟังก์ชันและการเขียนโปรแกรมแบบขั้นตอน สิ่งเหล่านี้มีไลบรารี่ที่ครอบคลุมการทำงานอย่างหลากหลาย


**รูปที่ 2.20** Logo ของโปรแกรม python

## 2.10. FIR

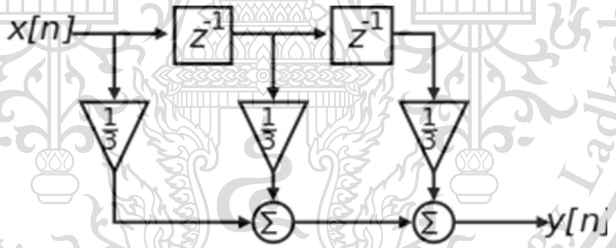
ในทาง signal processing นั้น Finite Impulse Response (FIR) filter คือฟิลเตอร์ที่มีการตอบสนองต่อสัญญาณ impulse หรือสัญญาณที่มีความยาวจำกัดจะตรงข้ามกับ Infinite Impulse Response(IRR) filters เพราะเนื่องจากสุดท้ายแล้วสัญญาณจะกลายเป็น 0 แต่ IRR อาจจะมี Feedback และอาจจะทำงานต่อไปเรื่อย ๆ

**ข้อดีของ** FIR

1.มีความเสถียรกว่า IIR

2.ไม่ต้องการ Feedback

3.สามารถออกแบบเป็น linear phase ได้ง่าย




**รูปที่ 2.21** block diagram ของ FIR

## 2.11. Window

เป็นการแบ่งช่วงของเวลาที่เลือกไว้ให้สมมาตรกัน โดยช่วงเวลาปกติจะอยู่ใกล้จุดสูงสุดตรงกลางและจะลดลงเรื่อย ๆ ในทางคณิตศาสตร์เมื่อฟังก์ชันหรือรูปคลื่นสัญญาณถูกคูณด้วยฟังก์ชัน window นอกช่วง Window จะมีค่าเป็น 0 ส่วนที่เหลืออยู่คือส่วนที่ซ้อนทับกับตัว Window

**Hann and Hamming windows**

เป็นฟังก์ชัน window ที่ใช้ในการทดลองเรื่องเสียงจะมีสมการเป็น

$$w[n] = \sum_{k=0}^{K} (-1)^k a_k \cos\left(\frac{2\pi kn}{N}\right), 0 \le n \le N \qquad (2.1)$$

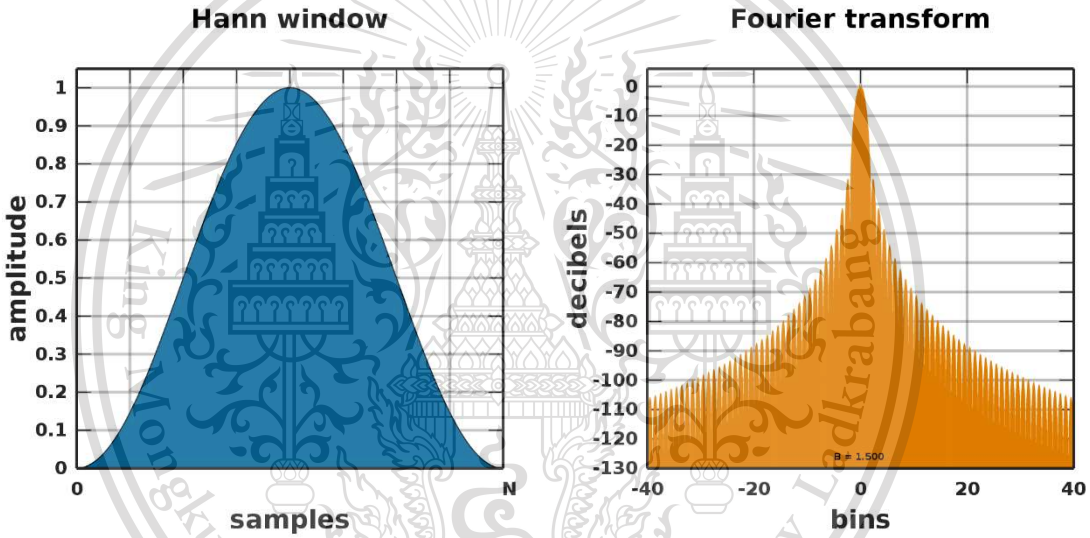
โดยปกติสมการ 2.1 คือสมการ cosine-sum window สำหรับกรณีที่ K=1 แต่สมการเฉพาะสำหรับ Hann and Hamming windows คือ

$$w[n] = a_0 - (1 - a_0) \cdot \cos\left(\frac{2\pi n}{N}\right), 0 \le n \le N \tag{2.2}$$

แต่ในรูปอย่างง่ายกรณีที่เฟสเป็น 0 จะได้

$$w_0[n] = w[n + \frac{N}{2}]$$

$$= a_0 + a_1 \cdot \cos\left(\frac{2\pi n}{N}\right), -\frac{N}{2} \le n \le \frac{N}{2} \tag{2.3}$$

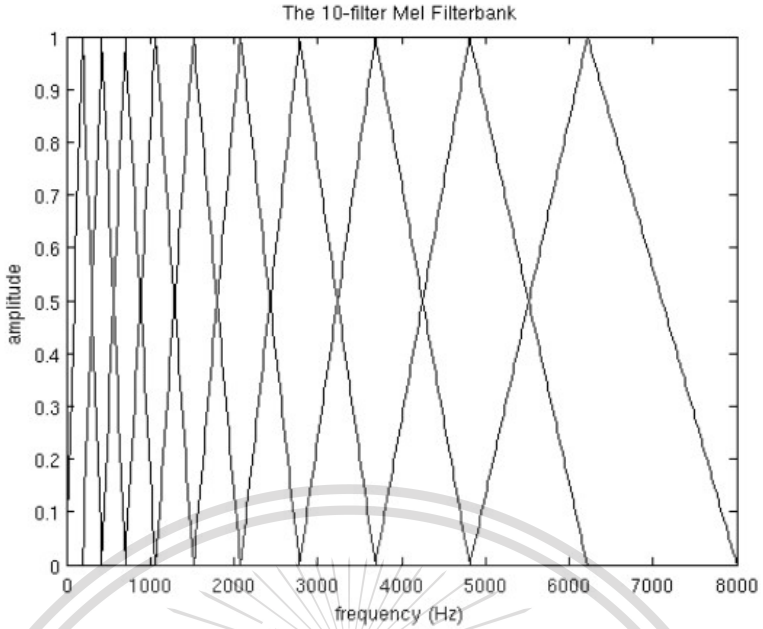



**รูปที่ 2.22** Hann window

## 2.12. Mel Scale

เนื่องจากมนุษย์ไม่มีความสามารถรับรู้เสียงแบบเส้นตรงได้และเรามีความสามารถในการตรวจจับหรือแยกแยะเสียงเสียงความถี่ต่ำได้ดีกว่าความถี่สูง เช่น เราสามารถแยกความต่างของเสียงที่ความถี่ 500 Hz และ 1000 Hz ได้อย่างง่ายๆ แต่ยากที่จะแยกความแตกต่างความถี่ระหว่าง 10,000 และ 10,500 Hz ได้จึงต้องแปลงเสียงย่านความถี่ต่าง ๆ ให้ไปอยู่ใน Mel scale โดยใช้ Mel – frequency filter bank

Mel-frequency filter bank เป็นการหาค่าสัมประสิทธิ์เซปสตรัมบนสเกลเมล เริ่มต้นจากการนำสัญญาณเสียงมาผ่านการประมวลผลสัญญาณเสียงหลังจากนั้นส่งสัญญาณไปผ่านชุดตัวกรองฟิลเตอร์แบงค์ (filter bank) เพื่อเน้นความสำคัญของความถี่ที่อยู่ในช่วงกลางของชุดตัวกรองแต่ละตัวกรอง ชุดตัวกรองฟิลเตอร์แบงค์มีลักษณะ




**รูปที่ 2.23** Mel filter bank

โดยที่ความถี่กลางของตัวกรองแต่ละชุดนั้นเกิดจากการแปลงค่าความถี่ปรกติ ($f$) ให้อยู่บนสเกลเมล ($fmel$)

$$f_{mel} = 2595log_{10}(1 + \frac{f}{700}) \quad (2.4)$$

## 2.13. Cooley-Tukey FFT

FFT ได้รับการคิดค้นโดยนักคณิตศาสตร์ 2 ท่านชื่อ Cooley และ Tukey โดยทั้งสองท่อนได้นำเสนอการคิดค้นดังกล่าวต่อสาธารณะในปี ค.ศ. 1965 โดยสมการเป็นดังนี้

$$X(m) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi n \ /N} = \sum_{n=0}^{N-1} x(n)W_N^{mn} \quad (2.5)$$

แล้วถ้าแบ่งอินพุตเป็นสองส่วนโดยอีกส่วนหนึ่งเป็นเลขคู่ และอีกส่วนเป็นเลขคี่

$$X(m) = \sum_{n=0}^{N/2-1} x(2n)(W_{N/2}^2)^{mn} + \sum_{n=0}^{N/2-1} x(2n+1)(W_{N/2}^2)^{mn} \quad (2.6)$$

จะเห็นได้ว่าส่วนที่เป็นเลขคู่และเลขคี่ คือการคำนวณ (N/2)-point DFT ดังนั้นวิธีการนี้ได้แบ่งการคำนวณ N-point DFT ออกเป็นการคำนวณ (N/2)-point DFT จำนวน 2 ส่วนโดยนำเอา $W_N^m$ ซึ่งเป็นเลขเชิงซ้อนเรียกว่า Twiddle factor ไปคูณกับผลที่ได้จาก 2 ส่วนแล้วนำไปบวกกับ

ส่วนที่ 1 สังเกตว่าเราได้ทำการแบ่ง N ออกเป็น 2 ส่วนทำนองเดียวกันเราสามารถแบ่งตัวชี้ n ออกเป็น 2 ส่วนเช่นเดียวกันคือ m = 0,1,2,.....,N/2 – 1 ซึ่งจะใช้ตามสมการ (2.5) และส่วนที่สองคือ m = N/2, ....., N-1 ซึ่งเท่ากับตัวชี้ส่วนนี้จะเป็น m + N/2 เมื่อ m เป็นตัวชี้ในส่วนแรก พิจารณา X(m+N/2) ตามสมการดังนี้

$$X(m+N/2) = \sum_{n=0}^{N/2-1} x(2n)W_{N/2}^{mn} - W_N^m \sum_{n=0}^{N/2-1} x(2n+1)W_{N/2}^{mn} \quad (2.7)$$

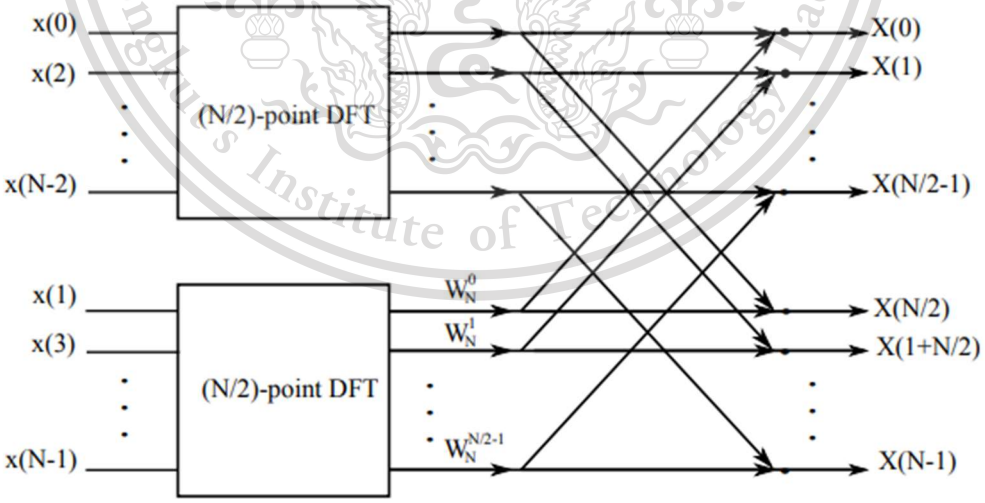
จะเห็นได้ว่าในการคำนวณ X(m + N/2) นั้นต่างกับการคำนวณ X(m) เพียงการนำส่วนหลังไปลบออกจากส่วนหน้าแทนการไปบวก ด้วยวิธีนี้สามารถทำ DFT โดย

1. ให้ m = 0
2. คำนวณ (N/2)-point DFT ที่จุด m ของอินพุต x(n) ส่วนที่ n เป็นเลขคู่ให้ผลลัพธ์คือ $X_0(m)$
3. คำนวณ (N/2)-point DFT ที่จุด m ของอินพุต x(n) ส่วนที่ n เป็นเลขคู่ให้ผลลัพธ์คือ $X_1(m)$
4. คูณ $X_1(m)$ กับ twiddle factor $W_N^m$ ได้ผลลัพธ์ $W_N^m X_1(m)$
5. คำนวณ X(m) และ X(m +N+2) ดังนี้

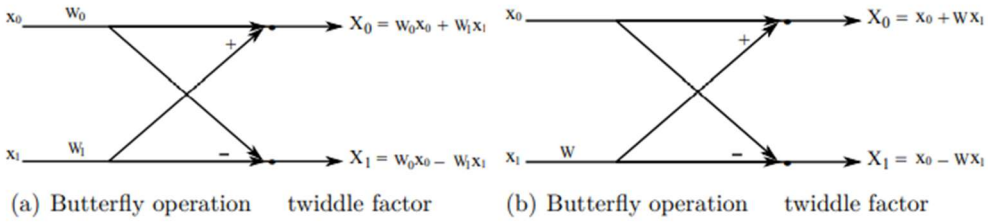
$$X(m) = X_0(m) + W_N^m X_1(m) \quad (2.8)$$

$$X(m+N/2) = X_0(m) - W_N^m X_1(m) \quad (2.9)$$

6. m = m +1
7. ถ้า m = N/2 จบไปทำขั้นตอนที่ 2




**รูปที่ 2.24** ไดอะแกรมการไหลของการคำนวณ DFT ตามสมการ (2.7) และ (2.8)




**รูปที่ 2.25** Butterfly operation

(a) แบบ twiddle factor ด้านเดียว และ (b) แบบ twiddle factor สองด้าน

หัวใจของการคำนวณตามอัลกอริทึมข้างบนคือ เราสามารถใช้วิธีนี้กับการคำนวณ (N/2)-point DFT ซึ่งแบ่งเป็นการคำนวณ (N/4)-point DFT ทั้งหมด 4 ส่วน และถ้า $\frac{N}{4} \neq 2$ เราสามารถแบ่งต่อไปจนกว่าจะได้การคำนวณแต่ละส่วน เป็น 2-point DF

# บทที่ 3

# วิธีการดำเนินงาน

## 3.1 กำหนด opcode ที่จะมาใช้การทำงาน

การกำหนด opcode ที่จะนำมาใช้นั้นจะต้องขึ้นอยู่กับผู้ออกแบบ Assembler และผู้ออกแบบ Microprocessor ทำการตกลงกันเองโดยให้สอดคล้องกันทั้งสองคนโดยจะกำหนดดังตารางต่อไปนี้

**ตารางที่ 3.1** แสดงรหัสคำสั่ง Operation code

| | | | | | |
|---|---|---|---|---|---|
| | | | R0-R7 | 000 | R0-R7 |
| MOV | 0000 | 00 | 000-111 | 000 | 000-111 |
| | | | R0-R7 | 000 | Number |
| | 0001 | 00 | 000-111 | 000 | #imm16 |
| | | | | | |
| | | | | | |
| NormE | 0000 | 01 | | | |
| | | | | | |
| | | | | | |
| Fexp | 0000 | 10 | | | |
| | | | | | |
| | | Number | Number | Number | Number |
| WIN | 000101 | #imm4 | #imm11 | #imm4 | #imm4 |
| | | | | | |
| | | | Number | Number | |
| FIR | 0001 | 10 | #imm4 | #imm4 | |
| | | | | | |
| | | | Number | Number | |
| FADD2 | 0010 | 01 | #imm4 | #imm4 | |
| | | | | | |
| | | | Number | Number | |
| FSUB2 | 0010 | 10 | #imm4 | #imm4 | |
| | | | | | |
| | | | R0-R7 | R0-R7 | Number |
| ADD | 0010 | 00 | 000-111 | 000-111 | #imm16 |
| | | | R0-R7 | R0-R7 | R0-R7 |
| | 0011 | 00 | 000-111 | 000-111 | 000-111 |
| | | | Number | Number | Number |

| | | | | | |
|---|---|---|---|---|---|
| MAC | 0100 | 01 | #imm4 | #imm4 | #imm4 |
| | | | | | |
| | | | Number | Number | |
| SQR1 | 0100 | 10 | #imm4 | #imm4 | |
| | | | | | |
| | | | Number | Number | |
| SQR2 | 0100 | 11 | #imm4 | #imm4 | |
| | | | | | |
| | | | R0-R7 | R0-R7 | Number |
| SUB | 0100 | 00 | 000-111 | 000-111 | #imm16 |
| | | | R0-R7 | R0-R7 | R0-R7 |
| | 0101 | 00 | 000-111 | 000-111 | 000-111 |
| | | | | | |
| | | | Number | Number | |
| LOOPM | 0110 | 10 | #imm16 | #imm10 | |
| | | | | | |
| | | | R0-R7 | R0-R7 | R0-R7 |
| FADD | 0110 | 00 | 000-111 | 000-111 | 000-111 |
| | | | | | |
| | | | | | |
| FDIVN | 0110 | 11 | | | |
| | | | | | |
| | | | R0-R7 | R0-R7 | R0-R7 |
| FMUL | 0110 | 01 | 000-111 | 000-111 | 000-111 |
| | | | | | |
| | | | | | |
| FDIV | 0111 | 00 | | | |
| | | | | | |
| | | | Number | | Number |
| LOOPL | 0111 | 01 | #imm16 | | #imm10 |
| | | | | | |
| | | | R0-R7 | R0-R7 | R0-R7 |
| FSUB | 0111 | 11 | 000-111 | 000-111 | 000-111 |
| | | | | | |
| | | | Number | | Number |
| LOOPA | 1000 | 01 | #imm16 | | #imm10 |
| | | | | | |
| | | | Number | | Number |

| | | | | | |
|---|---|---|---|---|---|
| LOOPB | 1000 | 10 | #imm16 | | #imm10 |
| | | | | | |
| | | | R0-R7 | R0-R7 | Number |
| ORR | 1000 | 00 | 000-111 | 000-111 | #imm16 |
| | | | R0-R7 | R0-R7 | R0-R7 |
| | 1000 | 11 | 000-111 | 000-111 | 000-111 |
| | | | | | |
| | | | R0-R7 | R0-R7 | Number |
| AND | 1001 | 00 | 000-111 | 000-111 | #imm16 |
| | | | R0-R7 | R0-R7 | R0-R7 |
| | 1001 | 11 | 000-111 | 000-111 | 000-111 |
| | | | | | |
| | | | Number | | |
| REOR | 1001 | 01 | #4 | | |
| | | | | | |
| | | | | | |
| INT | 1010 | 10 | | | |
| | | | | | |
| | | | R0-R7 | R0-R7 | Number |
| INV | 1010 | 00 | 000-111 | 000-111 | #imm16 |
| | | | R0-R7 | R0-R7 | R0-R7 |
| | 1010 | 11 | 000-111 | 000-111 | 000-111 |
| | | | | | |
| | | | R0-R7 | R0-R7 | Number |
| XOR | 1011 | 00 | 000-111 | 000-111 | #imm16 |
| | | | R0-R7 | R0-R7 | R0-R7 |
| | 1011 | 11 | 000-111 | 000-111 | 000-111 |
| | | | | | |
| | | | | | |
| NormL | 1011 | 01 | | | |
| | | | | | |
| | | | | | |
| Flog | 1011 | 10 | | | |
| | | | | | |
| | | | R0-R7 | R0-R7 | Number |
| MUL | 1100 | 00 | 000-111 | 000-111 | #imm16 |
| | | | R0-R7 | R0-R7 | R0-R7 |
| | 1100 | 11 | 000-111 | 000-111 | 000-111 |

| | | | | | |
|---|---|---|---|---|---|
| | | | Number | | |
| CALL | 1100 | 010 | #imm16 | | |
| | | | | | |
| | | | | | |
| RET | 1100 | 011 | 0000 | 0000 | |
| | | | | | |
| | | | R0-R7 | [R0-R7] | Number |
| LDR | 1101 | 00 | 000-111 | [000-111] | #imm4 |
| | | | | | |
| | | | R0-R7 | [R0-R7] | Number |
| STR | 1101 | 11 | 000-111 | [000-111] | #imm4 |
| | | | | | |
| | | | R0-R7 | [R0-R7] | Number |
| LDR2 | 1101 | 01 | 000-111 | [000-111] | #imm4 |
| | | | | | |
| | | | R0-R7 | [R0-R7] | Number |
| STR2 | 1101 | 10 | 000-111 | [000-111] | #imm4 |
| | | | | | |
| | | | Number | | |
| B | 1110 | 00 | #imm16 | | |
| | | | | | |
| | | | R0-R7 | R0-R7 | Number |
| BEQ | 1110 | 10 | 000-111 | 000-111 | #imm16 |
| | | | R0-R7 | Number | Number |
| | 1110 | 11 | 000-111 | #imm4 | #imm16 |
| | | | | | |
| | | | R0-R7 | R0-R7 | Number |
| BNE | 1111 | 00 | 000-111 | 000-111 | #imm16 |
| | | | R0-R7 | Number | Number |
| | 1111 | 11 | 000-111 | #imm4 | #imm16 |
| | | | | | |
| | | | R0-R7 | | |
| POP | 1111 | 01 | 000-111 | | |
| | | | | | |
| | | | R0-R7 | | |
| PUSH | 1111 | 10 | 000-111 | | |

## 3.2 ออกแบบ Assembler

การออกแบบ Assembler หรือการออกแบบรหัสคำสั่งแต่ละคำสั่งมีความหมายดังนี้

**1.กลุ่มคำนวณ**

1.1.MOV คือ คำสั่ง MOVE โดยการย้ายค่าไปยัง register หรือการย้ายจาก register หนึ่งไปยังอีก register หนึ่ง เช่น

MOV R1,R2 คือ การย้ายค่าให้ R1 = R2

MOV R1,#2 คือ การให้ค่า R1 = 2

1.2.ADD คือ คำสั่ง Addition โดยการนำค่า register มาบวกกับ register หรือการนำค่า register มาบวกกับตัวเลข เช่น

ADD R1,R2,R3 คือ การนำ R2+R3 หลังจากการบวกแล้วให้เก็บค่าที่ R1

ADD R1,R3,#5 คือ การนำ R3+5 หลังจากการบวกแล้วให้เก็บค่าที่ R1

1.3.SUB คือ คำสั่ง subtraction โดยการนำค่า register มาลบกับ register หรือการนำค่า register มาลบกับตัวเลข เช่น

SUB R2,R3,R4 คือ การนำ R3-R4 หลังจากการลบแล้วให้เก็บค่าที่ R2

SUB R4,R2,#5 คือ การนำ R2-5 หลังจากการลบแล้วให้เก็บค่าไว้ที่ R4

1.4.FADD คือ คำสั่งเดียวกับ ADD แต่เป็นการบวกในรูปแบบ Floating point 16 bit เช่น

FADD R1,R2,R3 คือ การนำ R2+R3 หลังจากการบวกแล้วให้เก็บค่าไว้ที่ R1

1.5.MUL คือ การทำคำสั่ง Multiplication โดยการนำค่า register หนึ่งมาคูณกับอีก register หนึ่งหรือการนำ register มาคูณกับตัวเลข เช่น

MUL R2,R1,R3 คือ การนำ R1*R3 หลังจากการคูณแล้วให้เก็บค่าไว้ที่ R2

1.6.FMUL คือ คำสั่งการ Multiplication ในรูปแบบ Floating point 16 bit

FMUL R2,R3,R5 คือ การนำ R3 * R5 หลังจากการคูณแล้วให้เก็บค่าไว้ที่ R2

1.7.FSUB คือ คำสั่งเดียวกับ SUB แต่เป็นการลบในรูปแบบ Floating point 16 bit

FSUB R1,R2,R1 คือ การนำ R2-R1 หลังจากการลบแล้วให้เก็บค่าไว้ที่ R1

1.8.NormE คือ การ normalize ตัวเลขก่อนจะทำ exponential ให้อยู่ระหว่าง 0-1 โดย input ที่จะทำ exponential อยู่ที่ R0 และเมื่อทำ normalize เสร็จผลลัพธ์จะอยู่ที่ R1

1.9.FEXP คือการทำ expo หลังจากตัวเลขถูก normalize โดยจะนำไปคูณกับค่าคงที่จน R0 มีค่ามากกว่าหรือเท่ากับ 0.998 ผลลัพธ์ที่ได้จะเป็น exponential ฐาน 2 เก็บไว้ใน R1

1.10.FDIVN คือการ normalize ก่อนทำการหาร input ตัวเศษอยู่ที่ R1 และตัวส่วนอยู่ที่ R0 โดยจะทำการปรับให้ค่า R0 อยู่ระหว่าง 0-1

1.11.FDIV คือคำสั่งทำการหารหลังจาก normalize แล้ว โดยจะนำค่าคงที่ไปคูณให้ตัวส่วนมีค่ามากกว่า 0.998 และคำตอบจะอยู่ที่ R1

1.12.NormL คือ การ normalize ตัวเลขก่อนจะทำ logarithm ให้อยู่ระหว่าง 0-1 โดย input ที่จะทำ log อยู่ที่ R0 และเมื่อทำ normalize เสร็จผลลัพธ์จะอยู่ที่ R1

1.13.Flog คือการทำ log หลังจากตัวเลขถูก normalize โดยจะนำไปคูณกับค่าคงที่จน R0 มีค่ามากกว่าหรือเท่ากับ 0.998 ผลลัพธ์ที่ได้จะเป็น log ฐาน 2 เก็บไว้ใน R1

**2.กลุ่ม logic**

2.1.ORR คือ การทำคำสั่งทางตรรกศาสตร์ โดยเป็น Operation OR เช่น

ORR R0,R2,R3 คือ การนำค่า R2 และ R3 มา OR กันหลังจากจบกระบวนการจะเก็บใน R0

ORR R0,R2,#5 คือ การนำค่า R2 และ 5 มา OR กันหลังจากจบกระบวนการจะเก็บใน R0

2.2.AND คือ การทำคำสั่งทางตรรกศาสตร์ โดยเป็น Operation AND เช่น

AND R2,R3,R1 คือการนำค่า R3 และ R1 มา AND กันหลังจากจบกระบวนการจะเก็บที่ R0

AND R1,R5,#5 คือ การนำค่า R5 และ 5 มา AND กันหลังจากจบกระบวนการจะเก็บใน R0

2.3.INV คือ การทำคำสั่งทางตรรกศาสตร์ โดยเป็น Operation invert เช่น

INV R2,R3 คือการนำค่า R3 มา INV หลังจากจบกระบวนการจะเก็บไว้ที่ R2

2.4.XOR คือ การทำคำสั่งทางตรรกศาสตร์ โดยเป็น Operation Exclusive OR เช่น

XOR R2,R1,R4 คือ การนำค่า R1 และ R4 มา XOR กันหลังจากจบกระบวนการจะเก็บที่ R2

XOR R2,R1,#2 คือ การนำค่า R1 และ 2 มา XOR กันหลังจากจบกระบวนการจะเก็บที่ R2

**3.กลุ่ม Load Store memory**

3.1.LDR คือ การ LOAD ค่าใน RAM มาเก็บไว้ใน register อีกตัวหนึ่ง เช่น

LDR R5,[R1],#1 คือ โหลดค่า Address จาก R1 มาเก็บใน R5 แล้วบวกค่า Address ไป 1

3.2.STR คือ การ Storage ค่าจาก Register ใส่ไว้ใน RAM เช่น

STR R2,[R3],#1 คือ นำค่า Address R3 ใส่ไว้ใน R2 แล้วบวกค่า Address ไป 1

3.3.LDR2 คือ การ LOAD ค่าใน Dmem ตัวที่ 2 มาเก็บไว้ใน register อีกตัวหนึ่ง เช่น

LDR2 R5,[R1],#1 คือ โหลดค่า Address จาก R1 มาเก็บใน R5 แล้วบวกค่า Address ไป 1

3.4.STR2 คือ การ Storage ค่าจาก Register ใส่ไว้ใน Dmem ตัวที่ 2 เช่น

STR2 R2,[R3],#1 คือ นำค่า Address R3 ใส่ไว้ใน R2 แล้วบวกค่า Address ไป 1

**4.กลุ่ม Branch**

4.1.B คือ Branch หมายถึงการกระโดดไปที่ Address นั้นๆ เช่น

B #12 คือ การกระโดดไปจากเดิม 12 ข้อความหรืออาจหมายถึงการกระโดดไปอีก 12 คำสั่ง

4.2.BEQ คือ Branch if Equal คือ การนำค่า register 2 ตัวมาเปรียบเทียบกันถ้าหากมีค่าเท่ากันจะทำการกระโดดไปที่ Address ถัดไป เช่น

BEQ R2,R1,#2 คือ การนำ R2 เทียบกับ R1 ถ้าหากเท่ากันจะไปที่คำสั่ง PC+2

4.3.BNE คือ Branch if Not Equal คือการนำค่า register 2 ตัวมาเปรียบเทียบกันถ้าหากมีค่าไม่เท่ากันจะทำการกระโดดไปที่ Address ถัดไป เช่น

BNE R1,R3,#2 คือ การนำ R1 เทียบกับ R3 ถ้าหากไม่เท่ากันจะไปที่คำสั่ง PC+2

**5.กลุ่ม stack**

5.1.POP คือ การนำค่าใน stack pointer ล่าสุดไปใส่ Register เช่น

POP R2 คือ การนำค่าใน stack pointer ปัจจุบันไปใส่ที่ R2

5.2.PUSH คือ การนำค่าใน register ใส่ไว้ใน stack pointer แล้วลบ address ไป 1

PUSH R2 คือ การนำค่า R2 ใส่ไว้ใน stack pointer ปัจจุบันแล้วลบ address ไป 1

5.3.CALL คือ การเก็บค่า PC ปัจจุบันแล้วจะกระโดดไปทำ Address นั้นๆ เช่น

CALL #13 คือ การเก็บค่า PC ปัจจุบัน แล้วกระโดดไปทำ Address 13

5.4.RET คือ การนำค่า PC ปัจจุบัน จากการ Call มาใช้

**6.กลุ่ม คำสั่งพิเศษ (instant loop)**

6.1.LOOPA คือ การทำคำสั่งภายในกี่บรรทัด ซ้ำเป็นจำนวนกี่รอบ เช่น

LOOPA #3,#4 คือ การทำซ้ำ 4 บรรทัด เป็นจำนวนทั้งหมด 3 รอบ

6.2.LOOPB คือ การทำคำสั่งภายในกี่บรรทัด ซ้ำเป็นจำนวนกี่รอบ เช่น

LOOPB #4,#5 คือ การทำซ้ำ 5 บรรทัด เป็นจำนวนทั้งหมด 4 รอบ

6.3.LOOPM คือการทำคำสั่งไปเรื่อย ๆ จนค่าที่นำมาเทียบมีค่ามากกว่าหรือเท่ากับค่าที่ตั้งไว้ เช่น LOOPM #0.998,#1 คือการทำไปเรื่อย ๆ จนกว่าค่าที่นำมาเทียบจะมีค่ามากกว่าหรือเท่ากับ 0.998 และภายใน loop มีคำสั่งอยู่ 1 คำสั่ง

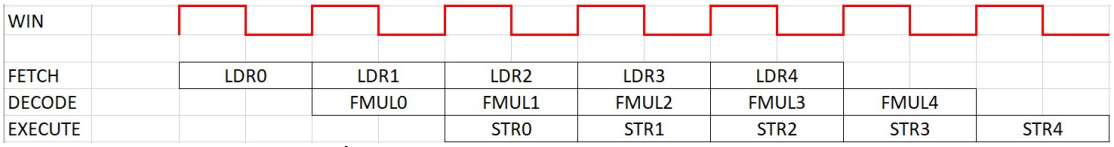
6.4.LOOPL คือการทำคำสั่งไปเรื่อย ๆ จนค่าที่นำมาเทียบมีค่าน้อยกว่าหรือเท่ากับค่าที่ตั้งไว้ เช่น LOOPL #0.998,#1 คือการทำไปเรื่อย ๆ จนกว่าค่าที่นำมาเทียบจะมีค่าน้อยกว่าหรือเท่ากับ 0.998 และภายใน loop มีคำสั่งอยู่ 1 คำสั่ง

**7.Operation เฉพาะ**

เป็นการเอาคำสั่งพื้นฐาน 5 คำสั่ง(LDR LDR2 STR FMUL FADD)มาทำใน clk เดียว

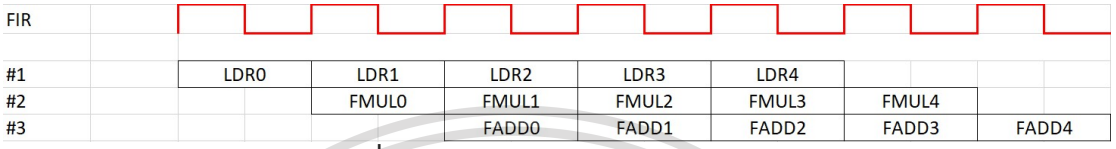
7.1.MAC คือ การรวม Operation ระหว่างการ LDR LDR2 FADD MUL STR

7.2.WIN คือ โดยที่เริ่มจากการ Load ข้อมูลจาก Dmem1 และ Dmem2 มาทำการคูณกันแล้วนำไปเก็บไว้ที่ Dmem1 โดยทุกคำสั่งนี้ทำภายใน 1 cycle



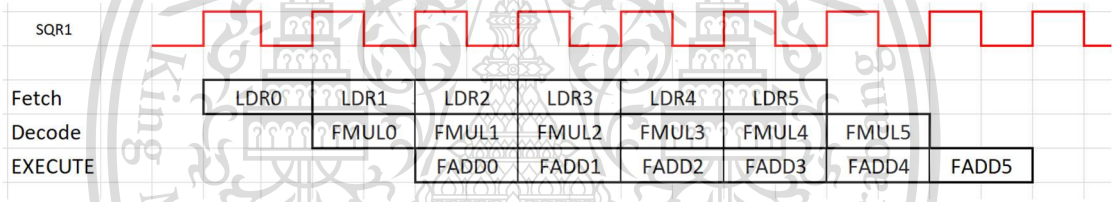

**รูปที่ 3.1** Pipeline ของ Operation Window

7.3.FIR คือ โดยที่เริ่มจากการ Load ข้อมูลจาก Dmem1 และ Dmem2 มาทำการคูณกันแล้วนำไปบวกกับค่าของผลคูณเรื่อย ๆ โดยทุกคำสั่งนี้ทำภายใน 1 cycle



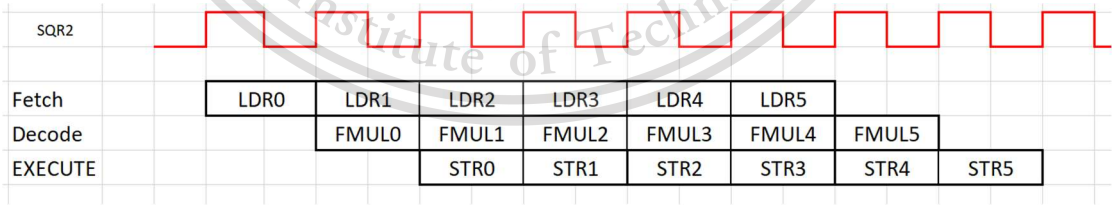

**รูปที่ 3.2** Pipeline ของ Operation FIR

7.4.SQR1 คือ คำสั่งหนึ่งของการทำ POWER จะใช้คู่กับ SQR2 ตัวอย่างการใช้ เช่น SQR1 #1 โดยจะทำการโหลดค่ามาเก็บใน R2 แล้วบวก address ไป 1 ทำการ FMUL R4,R2,R2 และ FADD R5,R4,R7 โดยทั้งหมดนี้จะทำภายใน 1 cycle




**รูปที่ 3.3** Pipeline ของ Operation SQR1

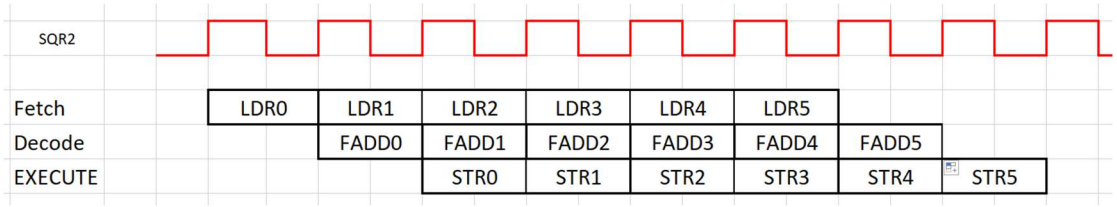
7.5.SQR2 คือ คำสั่งในการทำ Power ใช้คู่กับ SQ1 ตัวอย่างการใช้งานเช่น SQR2 #1,#1 คือ การโหลดค่ามาเก็บใน R3 แล้วบวก address ไป 1 ทำการ FMUL R7,R3,R3 และทำการ STR R5,[R6],1




**รูปที่ 3.4** Pipeline ของ Operation SQR2

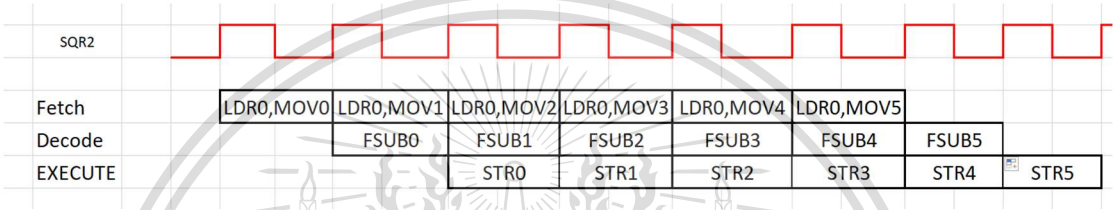
7.6 FADD2 คือส่วนหนึ่งของการทำ FFT จะใช้คู่กับ FSUB2 วิธีการใช้งานเช่น FADD2 #1,#1 โดยจะเริ่มโหลดข้อมูลมาเก็บไว้ที่ R8 และทำการ FADD R4,R2,R3 และเก็บค่าไปใน address ที่อยู่

ใน R6 โดยทุกคำสั่งนี้จะทำใน 1 cycle




**รูปที่ 3.5** Pipeline ของ Operation FADD2

7.7 FSUB2 คือส่วนหนึ่งของการทำ FFT ใช้คู่กับ FADD2 วิธีการใช้เช่น FSUB2 #1,#1 โดยจะเริ่มจาการโหลดค่ามาเก็บใน R3 ทำการย้ายค่าจาก R8 มาไว้ใน R2 และทำการ FSUB R7,R2,R3 สุดท้ายจะเก็บค่าใน R7 ไว้ใน address R5 ทุกคำสั่งนี้จะทำภายใน 1 cycle



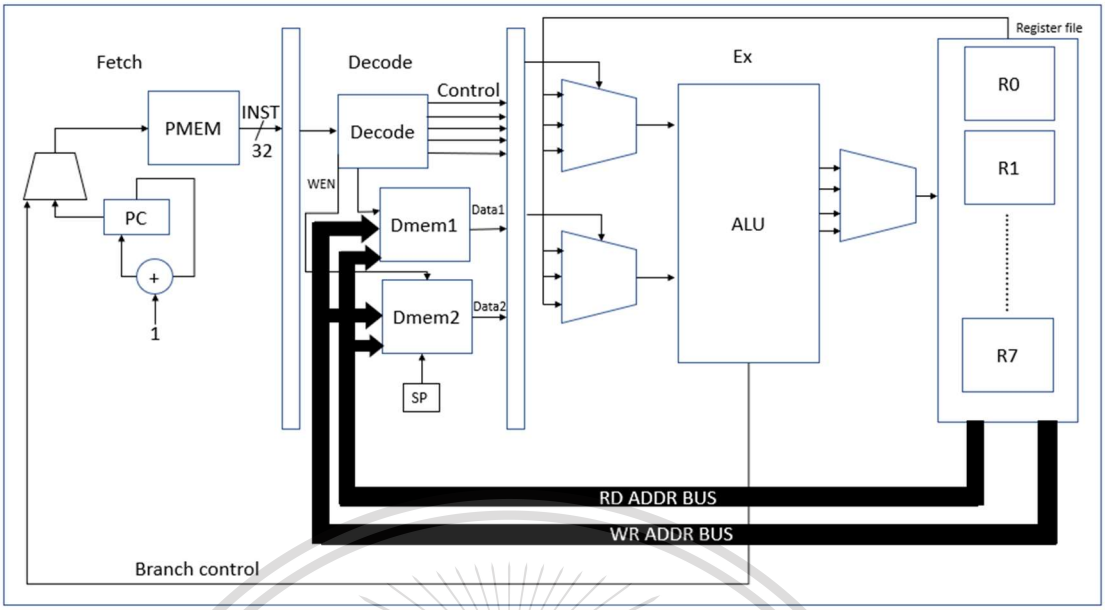

**รูปที่ 3.6** Pipeline ของ Operation FSUB2

7.8 REOR คือส่วนหนึ่งของการทำ FFT คือการจัดเรียงข้อมูลใหม่ โดยการสลับตำแหน่งข้อมูลใน memory ด้วยการกลับบิตโดยมี 4 ชุดคำสั่งแบ่งตามขนาดของจำนวน FFT

## 3.3 ออกแบบ Microprocessor ด้วย FPGA

### 3.3.1. CPU_TOP

สถาปัตยกรรมโปรเซสเซอร์ที่เลือกมาเป็นแบบในการออกแบบคือ สถาปัตยกรรมแบบ Harvard คือหน่วยความจำโปรแกรมและหน่วยความจำข้อมูลแยกออกจากกันและเลือกใช้คำสั่งแบบ RISC คือขนาดของแต่ละคำสั่งจะมีขนาดเท่ากันทำให้ไปป์ไลน์สามารถช่วยเพิ่มประสิทธิภาพในการทำงานของไมโครโปรเซสเซอร์ได้ดีขึ้น การออกแบบเบื้องต้นได้กำหนดให้ CPU มีขนาด 16-bit ขนาด Instruction set เท่ากับ 32-bit ทำให้ PMEM มีขนาดประมาณ 256KB และออกแบบหน่วยความจำข้อมูล 2 ตัวคือ DMEM1 และ DMEM2 โดยข้อมูลที่ถูกเก็บเข้าไปจะมีขนาด 16-bit ทำให้หน่วยความจำทั้งสองมีขนาดเท่ากับ 128KB และตัว DMEM2 มีการแชร์ Memory คือเป็นทั้ง Stack-pointer และจัดเก็บ Constant Value




**รูปที่ 3.7** ภาพรวมของระบบ Processor

### 3.3.2. Fetch & Decode

เมื่อได้รับ INST set จาก Fetch มาแล้วระบบจะทำการ Decode เพื่อส่งสัญญาณไปบอกอุปกรณ์ส่วนอื่นๆให้ทำงานได้ถูกต้อง การ Decode จะแบ่งเป็น 4 ส่วนหลักๆคือ

1.OPCODE หมายถึง คำสั่งที่ต้องการให้ระบบทำงาน ส่วนใหญ่จะมีขนาด 6-bit

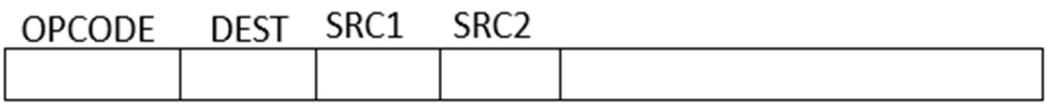
2.DEST หมายถึง Register จุดหมายที่จะเอาค่าผลลัพธ์นั้นมาใส่ในบางคำสั่งอาจจะนำค่า Dest มาใช้งานด้วย มีขนาด 3-bit

3.SRC1 และ SRC2 หมายถึง Register เป้าหมายตัวที่ 1 และ 2 ตามลำดับ ที่จะนำไปทำใน Operation นั้นมีขนาดตัวละ 3-bit

4. IMM ค่าตัวเลขที่จะมาทำการ Operation เช่นการบวกค่าหรือลบค่าใน Register

ชุด Instuction set จะแบ่งออกเป็น 14 แบบ โดยแต่ละแบบมีความหมายดังนี้

**แบบที่ 1**

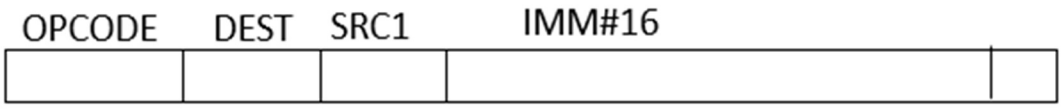



**รูปที่ 3.8** Instruction set แบบที่ 1

ในชุดคำสั่งแบบที่ 1 คำสั่งในชุดนี้ส่วนมากจะเป็นการกระทำกันระหว่าง Register 2 ตัว และนำไปเก็บใน Register เป้าหมาย เมื่อทำการถอดรหัสออกมา Decoder จะทำการส่งสัญญาณว่า SRC1, SRC2 มีค่าเท่ากับเท่าไหร่เพื่อไปเลือก Register ใน Execute มีการส่งสัญญาณ DEST เพื่อ

เลือก Register ที่ต้องการจัดเก็บและเนื่องจากใน Exe มีวงจรคำนวณมากมายจึงได้มีการส่งสัญญาณ INST เพื่อไปเลือกเอาผลลัพธ์ที่ต้องการมาเก็บใน Register ได้อย่างถูกต้อง

**แบบที่ 2**




**รูปที่ 3.9** Instruction set แบบที่ 2

ชุดคำสั่งแบบที่ 2 คำสั่งในชุดนี้จะแบ่งออกเป็น 2 แบบใหญ่ ๆ คือ

1.การนำ IMM มาบวกกับ Register SRC1 และจัดเก็บใน Register DEST

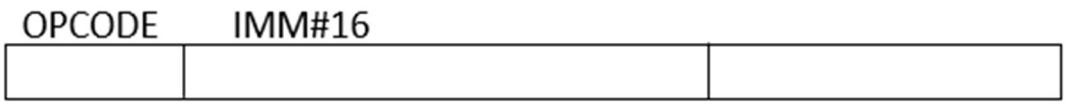
2.การทำ Branch Control ตัว Decoder จะส่งสัญญาณไปบอก Exe ให้นำค่า Register ทั้ง 2 ตัวมาเทียบกันแล้วนำค่า IMM ไปบวกหรือลบ ตามคำสั่งนั้นๆ

**แบบที่ 3**


**รูปที่ 3.10** Instruction set แบบที่ 3

คำสั่งแบบที่ 3 เป็นคำสั่งในการเก็บค่าหรืออ่านค่าใน DMEM ถ้าหากตีความว่าเขียนข้อมูลลงไปจะทำการส่งสัญญาณ WEN ไปที่ตัว DMEM และไปอ่านค่าใน Register SRC1 เพื่อนำค่าที่อยู่ที่ต้องการจะเขียนกลับมาและไปดึงค่า Register DEST เพื่อนำค่าที่จะต้องการจะเขียนจาก Exe มาทำการเขียนลงใน DMEM จากนั้นจะทำการบวกค่า IMM ที่ตัว Register SRC1 และถ้าตีความว่าอ่านข้อมูล Decoder จะไปอ่านค่าใน Register SRC1 ในทันทีและจะส่งสัญญาณ DEST ไปเลือกเพื่อจะนำข้อมูลที่ถูกอ่านออกมาเข้าไปเก็บใน Register DEST นั้นและทำการบวกค่า IMM เหมือนกับการเขียนข้อมูล

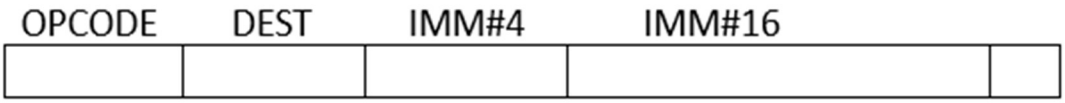
**แบบที่ 4**




**รูปที่ 3.11** Instruction set แบบที่ 4

เป็นคำสั่ง Branch โดยไม่มีเงื่อนไขเมื่อตีความได้คำสั่งนี้ Decoder จะส่งค่า IMM ไปบวกกับค่า PC โดยอัตโนมัติทันทีและจะส่งสัญญาณ NOP ไปที่ตัว Exe เพื่อให้ค่า Register ไม่เปลี่ยนแปลง

**แบบที่ 5**




**รูปที่ 3.12** Instruction set แบบที่ 5

เป็นอีกชุดคำสั่งหนึ่งของ Branch control เมื่อทำการ Decode เสร็จ จะส่งสัญญาณ DEST เพื่อไปดึงค่าใน Register ที่ต้องการมาเทียบกับค่า IMM ในชุดแรกและทำการส่งค่า IMM ชุดที่ 2 ไปบวกลบค่า PC ตามคำสั่งที่ได้สั่งมา

**แบบที่ 6**


**รูปที่ 3.13** Instruction set แบบที่ 6

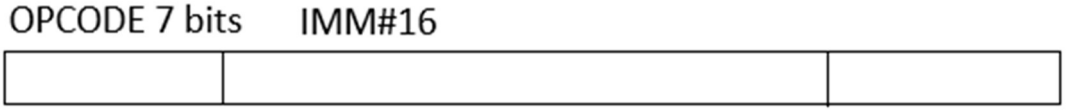
เป็นชุดคำสั่ง POP Push เมื่อตีความได้คำสั่ง Push ตัว Decode จะส่งสัญญาณ DEST เพื่อไปดึงค่าจาก Register มาและทำการส่งสัญญาณ WEN ไปที่ตัว DMEM2 และส่งสัญญาณไปเลือกที่อยู่ว่าให้เก็บแบบ Stack pointer และจะส่งสัญญาณ NOP ไปที่ตัว Exe เพื่อให้ค่า Register ไม่เปลี่ยนแปลง แต่เมื่อตีได้คำสั่ง Push จะไปทำการอ่านค่าใน DMEM2 ที่ ADDR แบบ Stack และจะส่งสัญญาณ DEST เพื่อเลือก Register ที่จะทำการเก็บค่าที่อ่านมานั้น

**แบบที่ 7**


**รูปที่ 3.14** Instruction set แบบที่ 7

เป็นชุดคำสั่งของ Return เมื่อตีความได้ Return Decoder จะทำการส่งสัญญาณไปยัง DMEM2 เพื่อทำการอ่านข้อมูลในนั้นจาก ADDR ล่าสุดแบบ Stack และทำการส่งไปยัง Fetch เพื่อเป็นค่า PC ในสัญญาณนาฬิกาถัดไป

**แบบที่ 8**




**รูปที่ 3.15** Instruction set แบบที่ 8

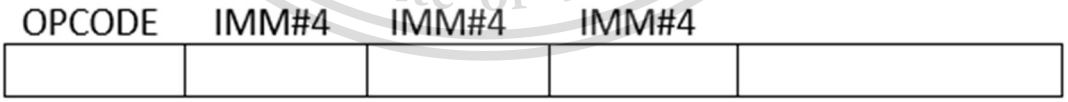
เป็นชุดคำสั่งของ Call เมื่อ Decode ตีความว่าเป็น Call จะส่งสัญญาณ WEN ไปที่ DMEM2 และจะทำการเลือก ADDR แบบ Stack Decode จะนำค่า PC ปัจจุบันมาเก็บไว้ใน DMEM2 และจะทำการเปลี่ยนค่า PC ให้เท่ากับค่า IMM นั้นเพื่อเป็น ADDR ของชุดคำสั่งต่อไปใน clock ถัดไปและจะส่งสัญญาณ NOP ไปที่ตัว Exe เพื่อให้ค่า Register ไม่เปลี่ยนแปลง

**แบบที่ 9**


**รูปที่ 3.16** Instruction set แบบที่ 9

เป็นชุดคำสั่ง LOOP เมื่อตีความได้ความหมายนี้ Decode จะทำการส่งสัญญาณไปที่ Fetch เพื่อให้เก็บค่าจำนวนลูปที่จะทำและจำนวนคำสั่งภายใน loop นั้น จากนั้น Fetch จะสั่งค่า PC ให้ทำกระบวนการวนลูปจนครบและหลุดลูปมาโดยอัตโนมัติและจะส่งสัญญาณ NOP ไปที่ตัว Exe เพื่อให้ค่า Register ไม่เปลี่ยนแปลง

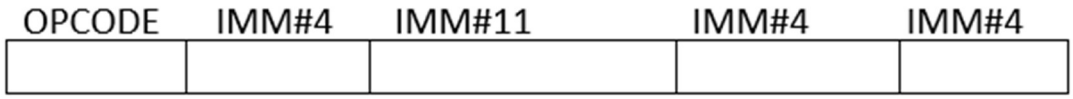
**แบบที่ 10**




**รูปที่ 3.17** Instruction set แบบที่ 10

เป็นชุดคำสั่งสำหรับการทำ Filter Decode จะทำการเก็บค่า IMM ไว้ 3 ชุด เพื่อรอนำไปบวกกับค่า Register ที่ถูก Fix ไว้ตามคำสั่งและส่งสัญญาณ INST ไปบอกที่ตัว Exe เพื่อให้ ALU กระทำการ บวก คูณ เลขระหว่าง Register หรือส่งค่าข้อมูลกลับมาเพื่อทำการอ่านค่าใน DMEM ตามที่คำสั่งได้กำหนดไว้

**แบบที่ 11**




**รูปที่ 3.18** Instruction set แบบที่ 11

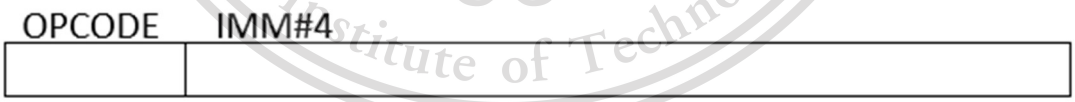
เป็นชุดคำสั่งสำหรับการทำ Window Decode จะทำการเก็บค่า IMM ไว้ 4 ชุด เพื่อรอนำไปบวกกับค่า Register ที่ถูก Fix ไว้ตามคำสั่งและส่งสัญญาณ INST ไปบอกที่ตัว Exe เพื่อให้ ALU กระทำการ บวก คูณ เลขระหว่าง Register หรือส่งค่าข้อมูลกลับมาเพื่อทำการอ่านค่าใน DMEM ตามที่คำสั่งได้กำหนดไว้

**แบบที่ 12**


**รูปที่ 3.19** Instruction set แบบที่ 12

เป็นชุดคำสั่งสำหรับการทำ FFT และส่วนหนึ่งของคำสั่งการทำ Power บางคำสั่ง Decode จะทำการเก็บค่า IMM ไว้ 2 ชุด เพื่อรอนำไปบวกกับค่า Register ที่ถูก Fix ไว้ตามคำสั่ง และส่งสัญญาณ INST ไปบอกที่ตัว Exe เพื่อให้ ALU กระทำการ บวก คูณ เลขระหว่าง Register หรือส่งค่าข้อมูลกลับมาเพื่อทำการอ่านค่าใน DMEM ตามที่คำสั่งได้กำหนดไว้

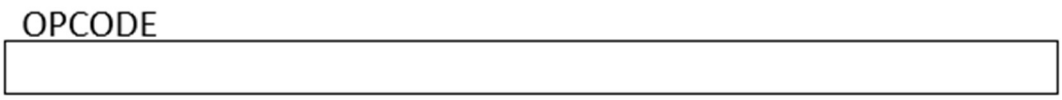
**แบบที่ 13**




**รูปที่ 3.20** Instruction set แบบที่ 13

เป็นส่วนหนึ่งของคำสั่งการทำ Power บางคำสั่ง Decode จะทำการเก็บค่า IMM ไว้ 1 ชุด เพื่อรอนำไปบวกกับค่า Register ที่ถูก Fix ไว้ตามคำสั่งและส่งสัญญาณ INST ไปบอกที่ตัว Exe เพื่อให้ ALU กระทำการ บวก คูณ เลขระหว่าง Register ตามที่คำสั่งได้กำหนดไว้

**แบบที่ 14**




**รูปที่ 3.21** Instruction set แบบที่ 14

เป็นชุดคำสั่งของพวกคำสั่ง Normalize หรือพวกคำสั่งทางคณิตศาสตร์ เช่น การหาร,การทำ logarithm และการทำ exponential โดยเมื่อ CPU fetch คำสั่งนี้ออกมา Decode จะส่งสัญญาณไปบอก Exe ให้นำค่าจาก register ไปคูณกับค่าคงที่ต่างๆ ตามคำสั่งที่อ่านมาได้

นอกจากนี้ในตัว Decode จะมี Data memory อยู่ 2 ตัวตามที่กล่าวไป

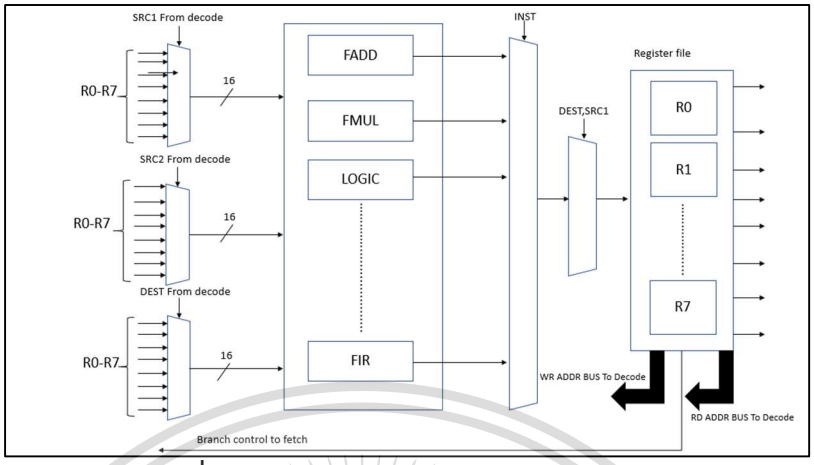

**รูปที่ 3.22** DMEM ในตัว Decode

DMEM ทั้ง 2 ตัว ถูกออกแบบให้เป็นแบบ 2-port memory หมายความว่าสามารถอ่านและเขียนได้ในเวลาเดียวกัน DMEM2 ต่างจาก DMEM ตัวแรกคือจะมีการแชร์ Memory กัน โดยที่ 100 ADDR แรกจะเป็นแบบ Stack pointer คือ ADDRแรกที่ถูกเขียนจะเป็น ADDR ที่ 99 และ ADDR จะลบ 1 ไปเองโดยอัตโนมัติ และเมื่อทำการอ่านค่าออก ADDR จะบวก 1 ให้เองแบบอัตโนมัติ พูดอย่างง่ายๆก็คือตัวแรกที่เก็บเข้าไปก่อนจะถูกอ่านค่ามาเป็นลำดับสุดท้ายและหลังจาก 100 ADDR แรกเป็นต้นไป จะเป็นการเก็บค่าคงที่ปกติเหมือน DMEM1

### 3.3.3. Execute

ในตัว Execute จะมีหน้าที่ทำ Operation นั้น ๆ ที่ได้จากการทำ Decode มา ภายในนี้จะมี Register เพื่อรองรับผลลัพธ์จากการ Operation อยู่ 8 ตัว แต่ละตัวสามารถเก็บข้อมูลได้ 2 byte การกระทำหลักๆภายในนี้ยกตัวอย่างเช่น การบวก, ลบ, คูณแบบจำนวนเต็ม, การกระทำแบบ Logic หรือการบวก, ลบ, คูณแบบทศนิยมขนาด 16-bit และ Register ทั้ง 8 ตัวสามารถส่งข้อมูลกลับไปยัง Decode เพื่อเป็นค่าที่อยู่หรือค่าข้อมูลที่ต้องการจะเก็บของ DMEM หรือส่งกลับไปยัง Fetch เพื่อทำการ Branch control ได้อีกด้วย โดยการทำงานอย่างคร่าวๆคือเมื่อ Decode ตีความคำสั่งเสร็จแล้วจะนำสัญญาณต่างๆเช่น DEST, SRC1 SRC2 มาเข้า Multiplexer เพื่อเลือก Register ที่จะนำค่า

ข้อมูลในนั้นไปเข้า ALU เมื่อได้ผลลัพธ์ออกมาจะนำสัญญาณ INST มาเลือกว่าต้องการ Output ไหน มาเก็บใน Register ปลายทาง




**รูปที่ 3.23** Block Diagram ภายใน Execute

# บทที่ 4

# ผลการดำเนินงาน

## 4.1. ผลการคำนวณทางคณิตศาสตร์

ค่าที่ได้เกิดจากการกำหนดค่าจากการสุ่มค่าเพื่อที่จะสามารถครอบคลุมการทำงานของ Operation ได้อย่างถูกต้อง

**ตารางที่ 4.1** ผลการคำนวณทางคณิตศาสตร์

| Decimal | Hexadecimal | FMUL | Decimal | Hexadecimal |
|---|---|---|---|---|
| 5.2 | 0x4533 | 0x49B8 | 2.2 | 0x4066 |
| -0.096191406 | 0xAE28 | 0xAA8E | 0.53241234 | 0x3842 |
| -0.102844238 | 0xAE95 | 0xAAC3 | 0.513689907 | 0x381C |
| -0.104980469 | 0xAEB8 | 0xAD1A | 0.759381235 | 0x3A13 |
| -0.108581543 | 0xAEF3 | 0x268E | -0.235810689 | 0xB38C |
| -0.115844727 | 0xAF6A | 0x2339 | -0.121757098 | 0xAFCB |
| -0.120422363 | 0xAFB5 | 0x2296 | -0.106784054 | 0xAED6 |
| -0.123413086 | 0xAFE6 | 0xAD6F | 0.688059685 | 0x3981 |
| -0.127563477 | 0xB015 | 0xA9A6 | 0.345836954 | 0x3589 |
| -0.13269043 | 0xB03F | 0xA995 | 0.328607056 | 0x3542 |
| -0.13659668 | 0xB05F | 0xA94B | 0.302806796 | 0x34D8 |
| -0.140014648 | 0xB07B | 0xAE59 | 0.708595844 | 0x39AB |
| -0.146362305 | 0xB0AF | 0xB09E | 0.986064333 | 0x3BE3 |
| -0.154785156 | 0xB0F4 | 0xAD05 | 0.50684033 | 0x380E |
| -0.158203125 | 0xB110 | 0xA831 | 0.206870767 | 0x329F |
| -0.158691406 | 0xB114 | 0x2C48 | -0.421455044 | 0xB6BE |
| Decimal | Hexadecimal | FADD | Decimal | Hexadecimal |
| 5.2 | 0x4533 | 0x4766 | 2.2 | 0x4066 |
| -0.096191406 | 0xAE28 | 0x36FA | 0.53241234 | 0x3842 |
| -0.102844238 | 0xAE95 | 0x3693 | 0.513689907 | 0x381C |

| | | | | |
|---|---|---|---|---|
| -0.104980469 | 0xAEB8 | 0x393C | 0.759381235 | 0x3A13 |
| -0.108581543 | 0xAEF3 | 0xB583 | -0.235810689 | 0xB38C |
| -0.115844727 | 0xAF6A | 0xB39A | -0.121757098 | 0xAFCB |
| -0.120422363 | 0xAFB5 | 0xB346 | -0.106784054 | 0xAED6 |
| -0.123413086 | 0xAFE6 | 0x3884 | 0.688059685 | 0x3981 |
| -0.127563477 | 0xB015 | 0x32FD | 0.345836954 | 0x3589 |
| -0.13269043 | 0xB03F | 0x3245 | 0.328607056 | 0x3542 |
| -0.13659668 | 0xB05F | 0x3151 | 0.302806796 | 0x34D8 |
| -0.140014648 | 0xB07B | 0x388C | 0.708595844 | 0x39AB |
| -0.146362305 | 0xB0AF | 0x3AB7 | 0.986064333 | 0x3BE3 |
| -0.154785156 | 0xB0F4 | 0x35A2 | 0.50684033 | 0x380E |
| -0.158203125 | 0xB110 | 0x2A3C | 0.206870767 | 0x329F |
| -0.158691406 | 0xB114 | 0xB8A4 | -0.421455044 | 0xB6BE |
| Decimal | Hexadecimal | FSUB | Decimal | Hexadecimal |
| 5.2 | 0x4533 | 0x4200 | 2.2 | 0x4066 |
| -0.096191406 | 0xAE28 | 0xB907 | 0.53241234 | 0x3842 |
| -0.102844238 | 0xAE95 | 0xB8EF | 0.513689907 | 0x381C |
| -0.104980469 | 0xAEB8 | 0xBAEA | 0.759381235 | 0x3A13 |
| -0.108581543 | 0xAEF3 | 0x3012 | -0.235810689 | 0xB38C |
| -0.115844727 | 0xAF6A | 0x1E10 | -0.121757098 | 0xAFCB |
| -0.120422363 | 0xAFB5 | 0xA2F8 | -0.106784054 | 0xAED6 |
| -0.123413086 | 0xAFE6 | 0xBA7E | 0.688059685 | 0x3981 |
| -0.127563477 | 0xB015 | 0xB794 | 0.345836954 | 0x3589 |
| -0.13269043 | 0xB03F | 0xB762 | 0.328607056 | 0x3542 |
| -0.13659668 | 0xB05F | 0xB708 | 0.302806796 | 0x34D8 |
| -0.140014648 | 0xB07B | 0xBACA | 0.708595844 | 0x39AB |
| -0.146362305 | 0xB0AF | 0xBC87 | 0.986064333 | 0x3BE3 |
| -0.154785156 | 0xB0F4 | 0xB94B | 0.50684033 | 0x380E |
| -0.158203125 | 0xB110 | 0xB5D8 | 0.206870767 | 0x329F |
| -0.158691406 | 0xB114 | 0x3434 | -0.421455044 | 0xB6BE |

## 4.2 การทดสอบความเร็วระหว่างการใช้ operation ปกติกับ operation ที่ปรับแต่ง

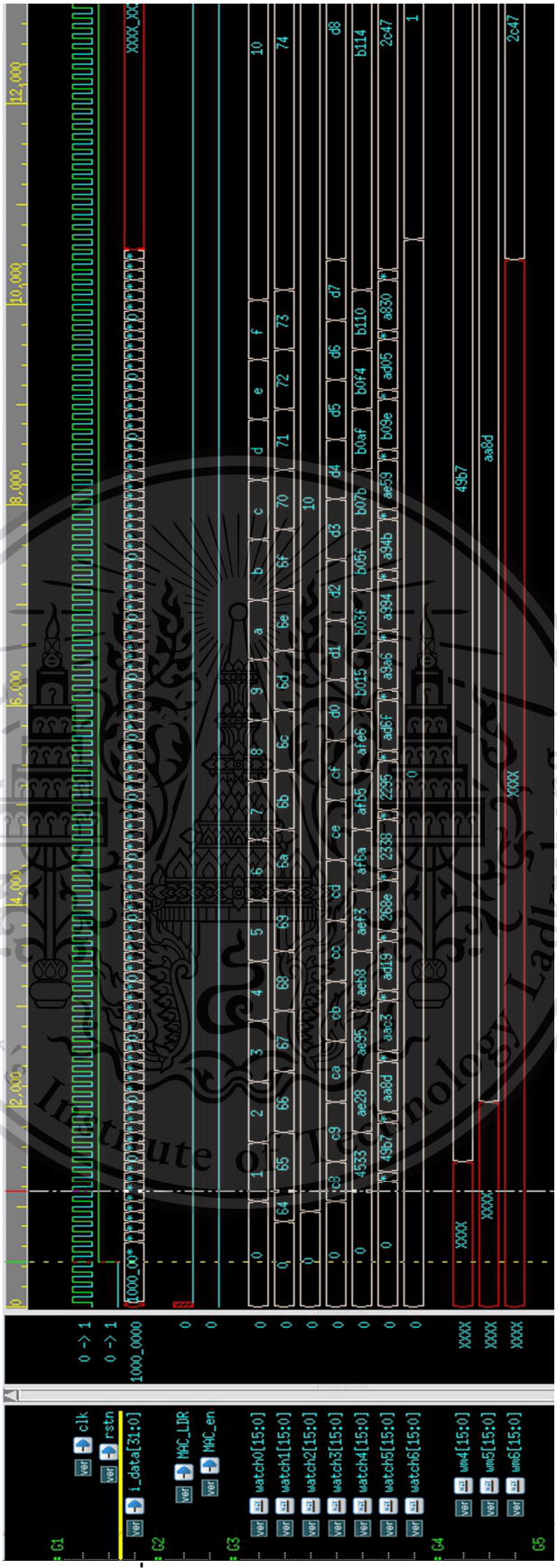
เป็นการทดสอบระหว่างการใช้ operation BNE เทียบกับ Operation LOOPA

### 4.2.1 Operation MUL โดยใช้ BNE

```
MOV R0,#0                          คือ การเก็บค่า 0 ที่ R0
MOV R1,#100                        คือ การเก็บค่า 100 ที่ R1
MOV R2,#16                         คือ การเก็บค่า 16 ที่ R2
MOV R3,#200                        คือ การเก็บค่า 200 ที่ R3
LABELA:   LDR R4,[R0],#1           คือ การนำค่า R0 ใส่ใน R4 แล้วบวก address 1
          LDR R5,[R1],#1           คือ การนำค่า R1 ใส่ใน R5 แล้วบวก address 1
          FMUL R5,R4,R5            คือ การนำ R4*R5 แล้วเก็บไว้ที่ R5
          STR R5,[R3],#1           คือ การนำค่า R3 ใส่ใน R5 แล้วบวก address 1
          BNE R0,R2,#LABELA        คือ การเทียบ R0 และ R2 ถ้าไม่เท่าให้กลับไป
                                   LABELA-PC(4-8)
MOV R6,#1                          คือ การเก็บค่า 1 ที่ R6
```

**ผลจากการแปลงเป็นเลขฐาน 16 จำนวน 8 bit**

```
10000000 // MOV R0,#0 ;
10800640 // MOV R1,#100 ;
11000100 // MOV R2,#16  ;
11800c80 // MOV R3,#200 ;
d2000010 // LDR R4,[R0],#1 ;
d2900010 // LDR R5,[R1],#1 ;
66ca0000 // FMUL R5,R4,R5 ;
deb00010 // STR R5,[R3],#1 ;
f02fffc0 // BNE R0,R2,#LABELA ;
13000010 // MOV R6,#1 ;
```



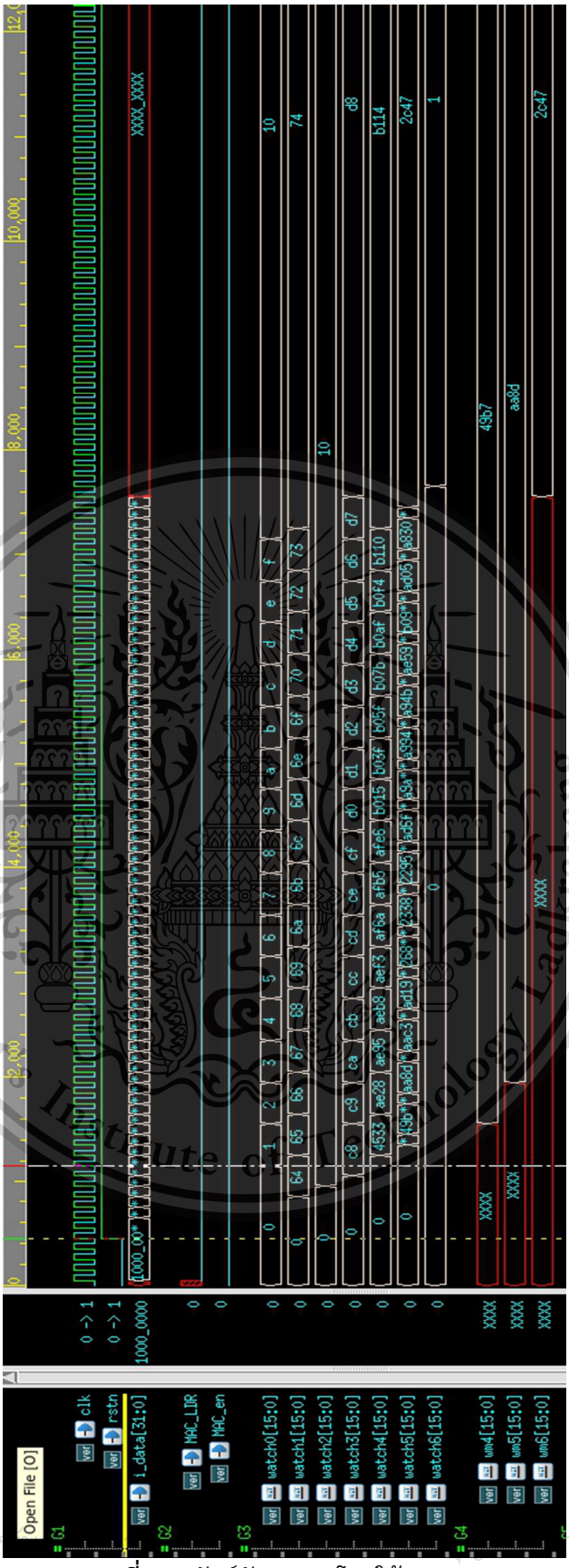
**รูปที่ 4.1** ฟังก์ชัน MUL โดยใช้ BNE

**4.2.2. Operation MUL โดยใช้ LOOPA**

| | |
|---|---|
| MOV R0,#0 | คือ การเก็บค่า 0 ที่ R0 |
| MOV R1,#100 | คือ การเก็บค่า 100 ที่ R1 |
| MOV R2,#16 | คือ การเก็บค่า 16 ที่ R2 |
| MOV R3,#200 | คือ การเก็บค่า 200 ที่ R3 |
| LOOPA #16,#4 | คือ การทำซ้ำ 16 รอบจำนวน 4 บรรทัด |
| LDR R4,[R0],#1 | คือ การนำค่า R0 ใส่ใน R4 แล้วบวก address 1 |
| LDR R5,[R1],#1 | คือ การนำค่า R1 ใส่ใน R5 แล้วบวก address 1 |
| FMUL R5,R4,R5 | คือ การนำ R4*R5 แล้วเก็บไว้ที่ R5 |
| STR R5,[R3],#1 | คือ การนำค่า R3 ใส่ใน R5 แล้วบวก address 1 |
| MOV R6,#1 | คือ การเก็บค่า 1 ที่ R6 |

**ผลจากการแปลงเป็นเลขฐาน 16 จำนวน 8 bit**

10000000 // MOV R0,#0 ;
10800640 // MOV R1,#100 ;
11000100 // MOV R2,#16 ;
11800c80 // MOV R3,#200 ;
84004004 // LOOPA #16,#4 ;
d2000010 // LDR R4,[R0],#1 ;
d2900010 // LDR R5,[R1],#1 ;
66ca0000 // FMUL R5,R4,R5 ;
deb00010 // STR R5,[R3],#1 ;
13000010 // MOV R6,#1 ;



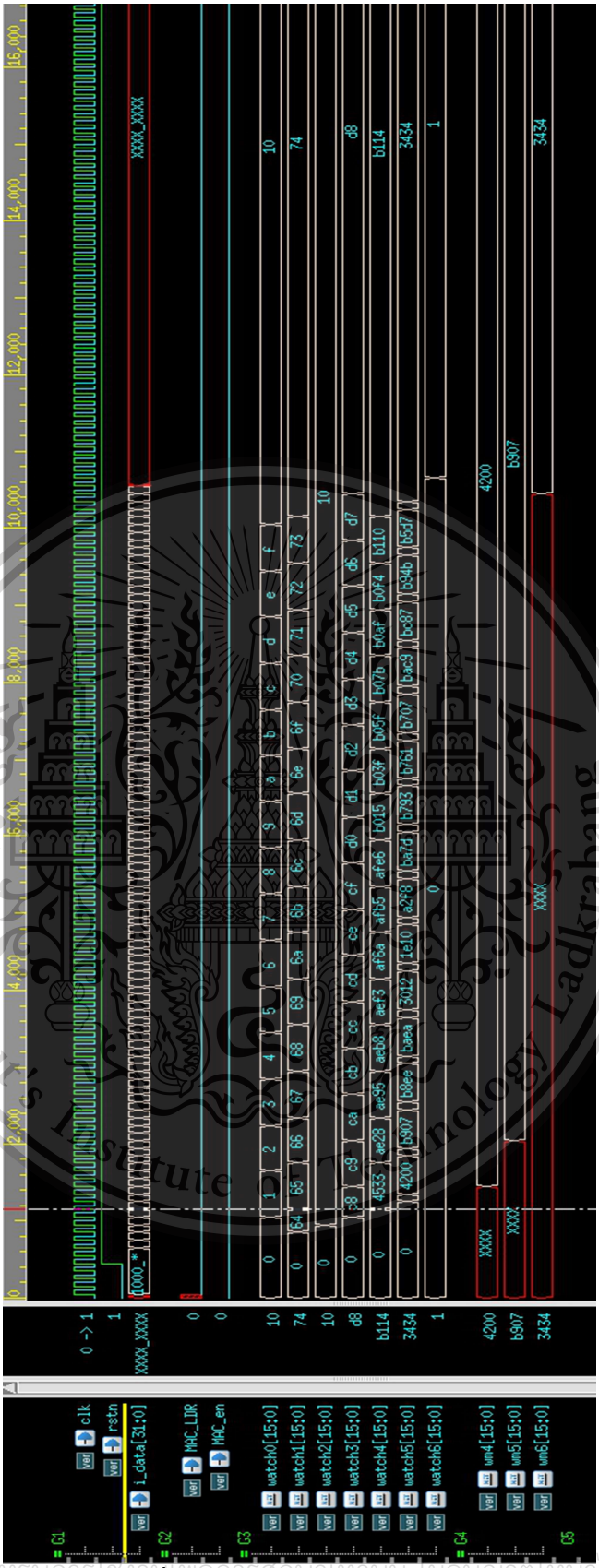

**รูปที่ 4.2** ฟังก์ชัน MUL โดยใช้ LOOP

**4.2.3 Operation SUB โดยใช้ BNE**

```
MOV R0,#0                          คือ การเก็บค่า 0 ที่ R0
MOV R1,#100                        คือ การเก็บค่า 100 ที่ R1
MOV R2,#16                         คือ การเก็บค่า 16 ที่ R2
MOV R3,#200                        คือ การเก็บค่า 200 ที่ R3
LABELA:   LDR R4,[R0],#1           คือ การนำค่า R0 ใส่ใน R4 แล้วบวก address 1
          LDR R5,[R1],#1           คือ การนำค่า R1 ใส่ใน R5 แล้วบวก address 1
          FSUB R5,R4,R5            คือ การนำ R4-R5 แล้วเก็บไว้ที่ R5
          STR R5,[R3],#1           คือ การนำค่า R3 ใส่ใน R5 แล้วบวก address 1
          BNE R0,R2,#LABELA        คือ การเทียบ R0 และ R2 ถ้าไม่เท่าให้กลับไป
                                   LABELA-PC(4-8)
MOV R6,#1                          คือ การเก็บค่า 1 ที่ R6
```

**ผลจากการแปลงเป็นเลขฐาน 16 จำนวน 8 bit**

```
10000000 // MOV R0,#0 ;
10800640 // MOV R1,#100 ;
11000100 // MOV R2,#16  ;
11800c80 // MOV R3,#200 ;
d2000010 // LDR R4,[R0],#1 ;
d2900010 // LDR R5,[R1],#1 ;
7eca0000 // FSUB R5,R4,R5 ;
deb00010 // STR R5,[R3],#1 ;
f02fffc0 // BNE R0,R2,#LABELA ;
13000010 // MOV R6,#1 ;
```



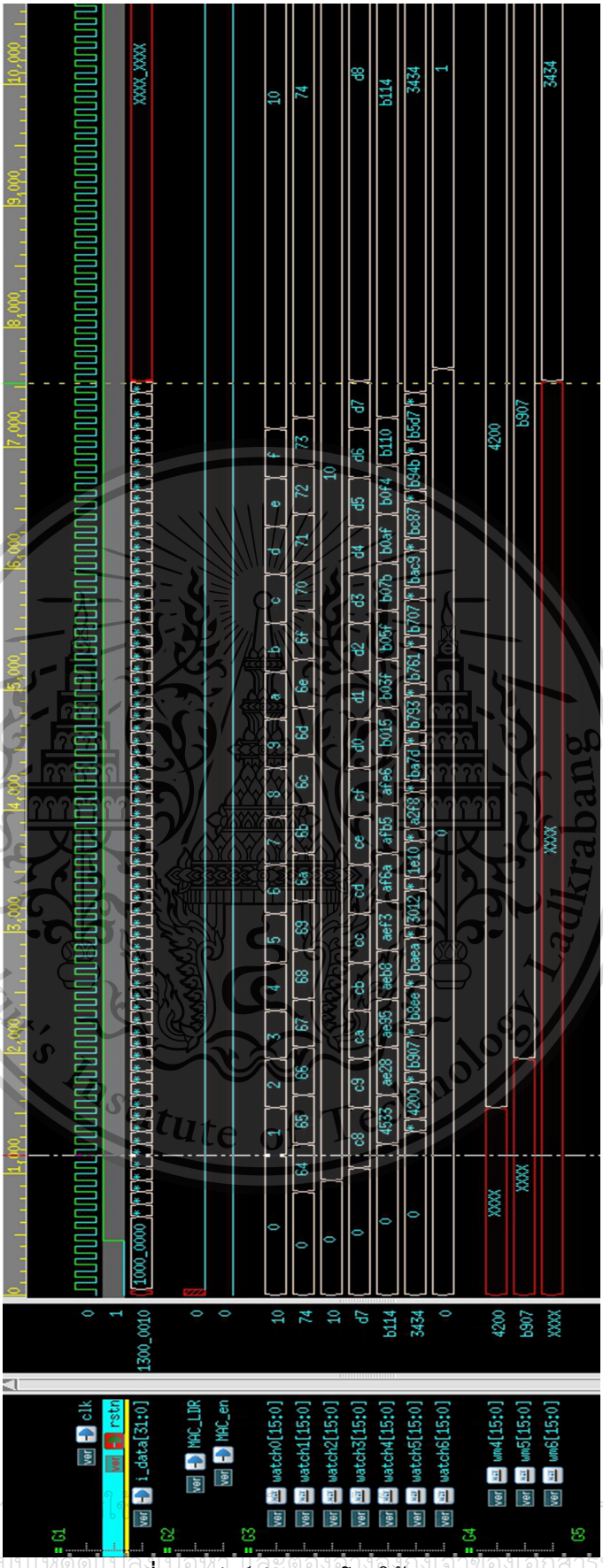

**รูปที่ 4.3** ฟังก์ชัน SUB โดยใช้ BNE

**4.2.4. Operation SUB โดยใช้ LOOPA**

| | |
|---|---|
| MOV R0,#0 | คือ การเก็บค่า 0 ที่ R0 |
| MOV R1,#100 | คือ การเก็บค่า 100 ที่ R1 |
| MOV R2,#16 | คือ การเก็บค่า 16 ที่ R2 |
| MOV R3,#200 | คือ การเก็บค่า 200 ที่ R3 |
| LOOPA #16,#4 | คือ การทำซ้ำ 16 รอบจำนวน 4 บรรทัด |
| LDR R4,[R0],#1 | คือ การนำค่า R0 ใส่ใน R4 แล้วบวก address 1 |
| LDR R5,[R1],#1 | คือ การนำค่า R1 ใส่ใน R5 แล้วบวก address 1 |
| FSUB R5,R4,R5 | คือ การนำ R4-R5 แล้วเก็บไว้ที่ R5 |
| STR R5,[R3],#1 | คือ การนำค่า R3 ใส่ใน R5 แล้วบวก address 1 |
| MOV R6,#1 | คือ การเก็บค่า 1 ที่ R6 |

**ผลจากการแปลงเป็นเลขฐาน 16 จำนวน 8 bit**

```
10000000 // MOV R0,#0 ;
10800640 // MOV R1,#100 ;
11000100 // MOV R2,#16  ;
11800c80 // MOV R3,#200 ;
84004004 // LOOPA #16,#4 ;
d2000010 // LDR R4,[R0],#1 ;
d2900010 // LDR R5,[R1],#1 ;
7eca0000 // FSUB R5,R4,R5 ;
deb00010 // STR R5,[R3],#1 ;
13000010 // MOV R6,#1 ;
```



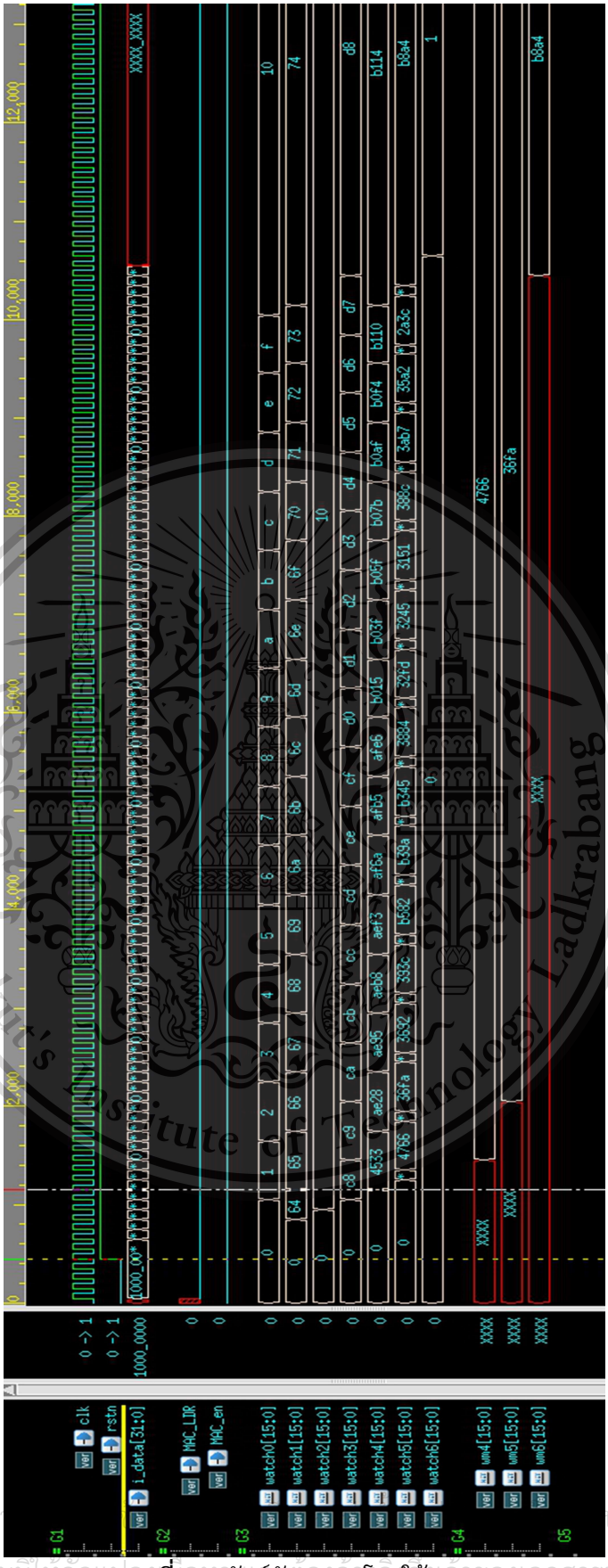

**รูปที่ 4.4** ฟังก์ชัน SUB โดยใช้ LOOPA

**4.2.5. Operation ADD โดยใช้ BNE**

```
MOV R0,#0                          คือ การเก็บค่า 0 ที่ R0
MOV R1,#100                        คือ การเก็บค่า 100 ที่ R1
MOV R2,#16                         คือ การเก็บค่า 16 ที่ R2
MOV R3,#200                        คือ การเก็บค่า 200 ที่ R3
LABELA:  LDR R4,[R0],#1            คือ การนำค่า R0 ใส่ใน R4 แล้วบวก address 1
         LDR R5,[R1],#1            คือ การนำค่า R1 ใส่ใน R5 แล้วบวก address 1
         FADD R5,R4,R5             คือ การนำ R4+R5 แล้วเก็บไว้ที่ R5
         STR R5,[R3],#1            คือ การนำค่า R3 ใส่ใน R5 แล้วบวก address 1
         BNE R0,R2,#LABELA         คือ การเทียบ R0 และ R2 ถ้าไม่เท่าให้กลับไป
                                   LABELA-PC(4-8)
MOV R6,#1                          คือ การเก็บค่า 1 ที่ R6
```

**ผลจากการแปลงเป็นเลขฐาน 16 จำนวน 8 bit**

```
10000000 // MOV R0,#0 ;
10800640 // MOV R1,#100 ;
11000100 // MOV R2,#16  ;
11800c80 // MOV R3,#200 ;
d2000010 // LDR R4,[R0],#1 ;
d2900010 // LDR R5,[R1],#1 ;
62ca0000 // FADD R5,R4,R5 ;
deb00010 // STR R5,[R3],#1 ;
f02fffc0 // BNE R0,R2,#LABELA ;
13000010 // MOV R6,#1 ;
```



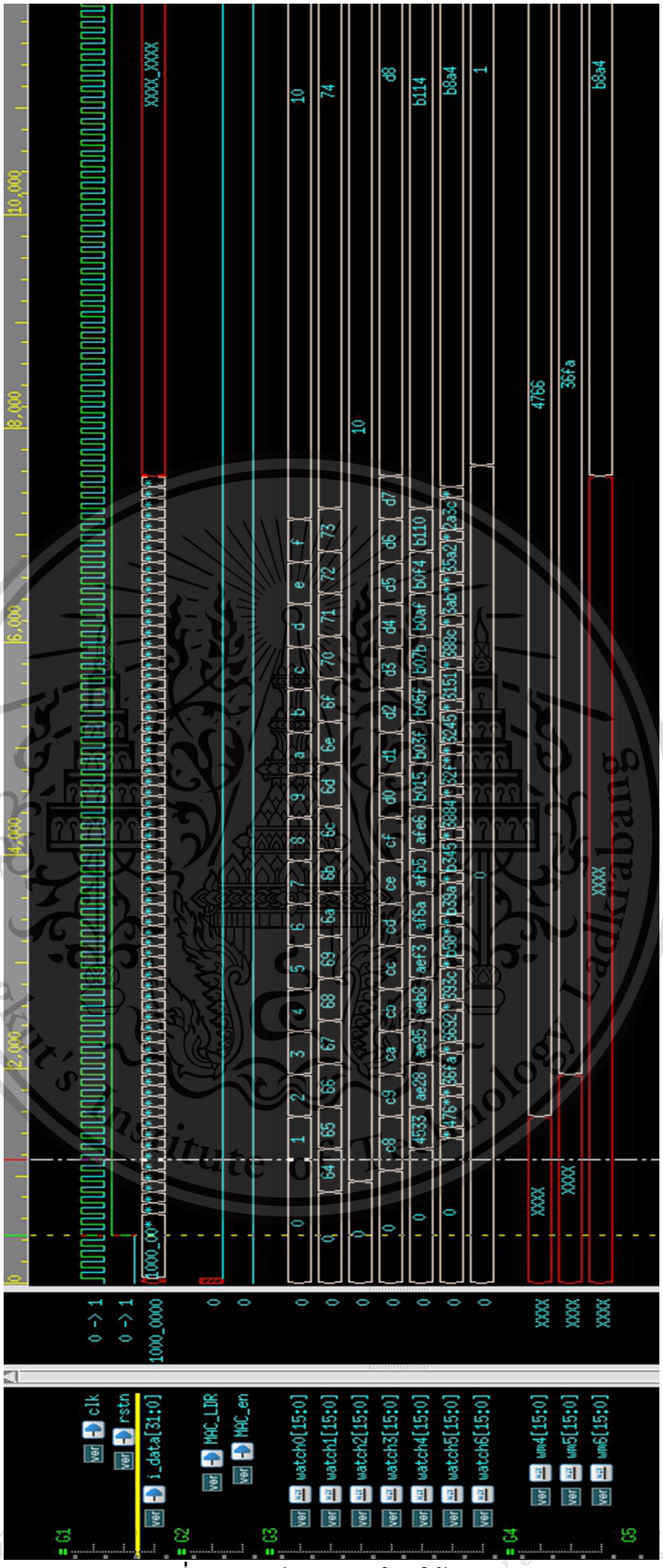

**รูปที่ 4.5** ฟังก์ชัน ADD โดยใช้ BNE

**4.2.6.Operation ADD โดยใช้ LOOPA**

| | |
|---|---|
| MOV R0,#0 | คือ การเก็บค่า 0 ที่ R0 |
| MOV R1,#100 | คือ การเก็บค่า 100 ที่ R1 |
| MOV R2,#16 | คือ การเก็บค่า 16 ที่ R2 |
| MOV R3,#200 | คือ การเก็บค่า 200 ที่ R3 |
| LOOPA #16,#4 | คือ การทำซ้ำ 16 รอบจำนวน 4 บรรทัด |
| LDR R4,[R0],#1 | คือ การนำค่า R0 ใส่ใน R4 แล้วบวก address 1 |
| LDR R5,[R1],#1 | คือ การนำค่า R1 ใส่ใน R5 แล้วบวก address 1 |
| FADD R5,R4,R5 | คือ การนำ R4+R5 แล้วเก็บไว้ที่ R5 |
| STR R5,[R3],#1 | คือ การนำค่า R3 ใส่ใน R5 แล้วบวก address 1 |
| MOV R6,#1 | คือ การเก็บค่า 1 ที่ R6 |

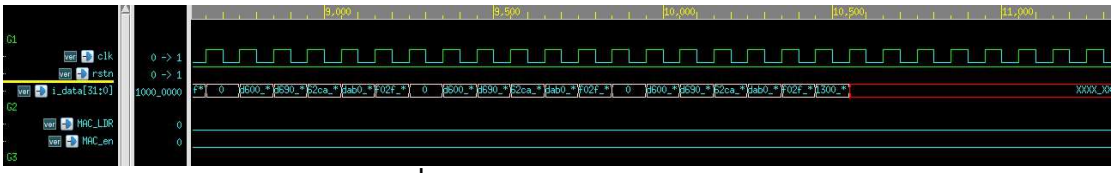
**ผลจากการแปลงเป็นเลขฐาน 16 จำนวน 8 bit**

10000000 // MOV R0,#0 ;
10800640 // MOV R1,#100 ;
11000100 // MOV R2,#16 ;
11800c80 // MOV R3,#200 ;
84004004 // LOOPA #16,#4 ;
d2000010 // LDR R4,[R0],#1 ;
d2900010 // LDR R5,[R1],#1 ;
62ca0000 // FADD R5,R4,R5 ;
deb00010 // STR R5,[R3],#1 ;
13000010 // MOV R6,#1 ;

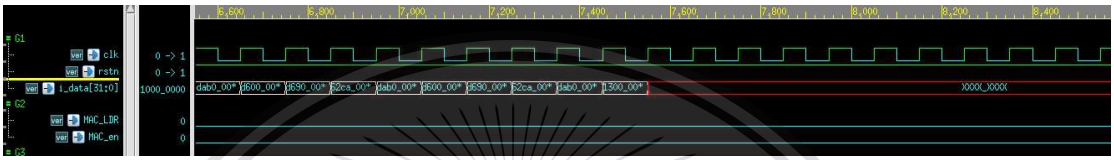



**รูปที่ 4.6** ฟังก์ชัน ADD โดยใช้ LOOP

ถ้าหากทำการขยายดูผลการทำงานของทั้งสองกระบวนการ จะเห็นได้ว่ากรณีที่เป็น LOOPA จะใช้จำนวนสัญญาณนาฬิกาน้อยกว่าการใช้แบบ BNE ดังรูป จะเห็นได้ว่า LOOPA ใช้สัญญาณนาฬิกาประมาณ 76 สัญญาณนาฬิกาและ BNE ใช้สัญญาณนาฬิกาประมาณ 106 สัญญาณนาฬิกา



**รูปที่ 4.7** ผลการทำงานของ BNE



**รูปที่ 4.8** ผลการทำงานของ LOOPA

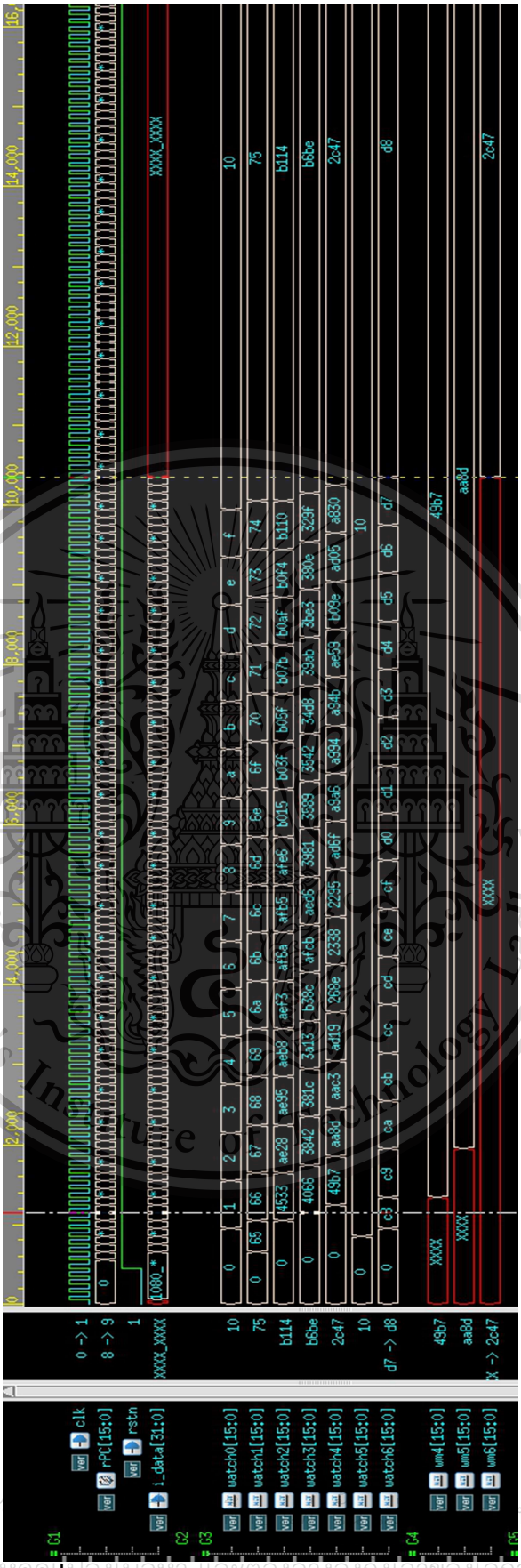
## 4.3. การใช้ operation ในการ window ข้อมูล

### 4.3.1. การใช้ Operation BNE ในการ Window

```
MOV R1,#101                คือ การเก็บค่า 101 ที่ R1
MOV R5,#16                 คือ การเก็บค่า 16 ที่ R5
MOV R6,#200                คือ การเก็บค่า 200 ที่ R6
LABELA:   LDR R2,[R0],#1   คือ การนำค่า R0 ใส่ใน R2 แล้วบวก address 1
          LDR2 R3,[R1],#1  การนำค่า R1 ใน spmem ใส่ใน R3 แล้วบวก
                           address 1
          FMUL R4,R2,R3    การนำ R2*R3 แล้วเก็บไว้ที่ R4
          STR R4,[R6],#1   คือ การนำค่า R6 ใส่ใน R4 แล้วบวก address 1
          BNE R5,R0,#LABELA คือ การเทียบ R5 และ R0 ถ้าไม่เท่าให้กลับไป
                           LABELA
```

**ผลจากการแปลงเป็นเลขฐาน 16 จำนวน 8 bit**

```
10800650 // MOV R1,#101  ;
12800100 // MOV R5,#16 ;
13000c80 // MOV R6,#200 ;
d1000010 // LDR R2,[R0],#1 ;
d5900010 // LDR2 R3,[R1],#1 ;
66260000 // FMUL R4,R2,R3 ;
de600010 // STR R4,[R6],#1 ;
f28fffc0 // BNE R5,R0,#LABELA ;
```



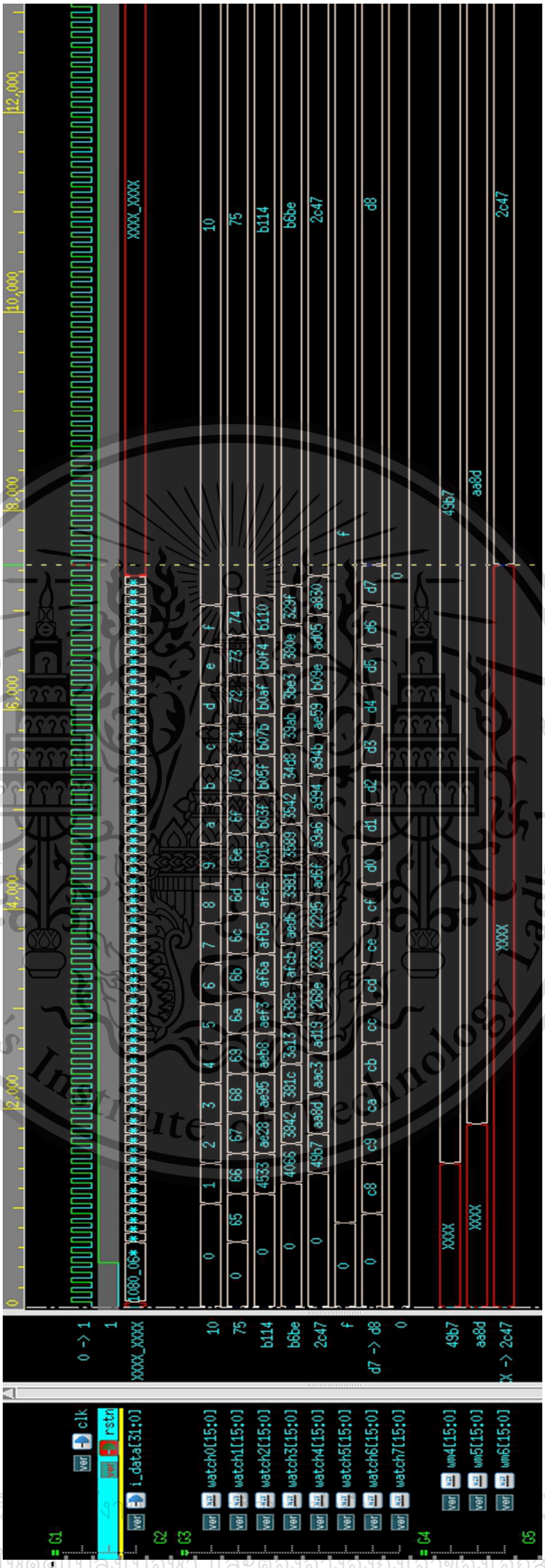

**รูปที่ 4.9** Operation BNE สำหรับทำ WINDOW

**4.3.2. การใช้ Operation LOOPA ในการ Window**

| | |
|---|---|
| MOV R1,#101 | คือ การเก็บค่า 101 ที่ R1 |
| MOV R5,#16 | คือ การเก็บค่า 16 ที่ R5 |
| MOV R6,#200 | คือ การเก็บค่า 200 ที่ R6 |
| LOOPA #16,#4 | คือ การทำซ้ำ 16 รอบจำนวน 4 บรรทัด |
| LDR R2,[R0],#1 | คือ การนำค่า R0 ใส่ใน R2 แล้วบวก address 1 |
| LDR2 R3,[R1],#1 | คือ การนำค่า R1 ใน spmem ใส่ใน R3 แล้วบวก address 1 |
| FMUL R4,R2,R3 | คือ การนำ R2*R3 แล้วเก็บไว้ที่ R4 |
| STR R4,[R6],#1 | คือ การนำค่า R6 ใส่ใน R4 แล้วบวก address 1 |

**ผลจากการแปลงเป็นเลขฐาน 16 จำนวน 8 bit**

10800650 // MOV R1,#101 ;
12800100 // MOV R5,#16 ;
13000c80 // MOV R6,#200 ;
84004004 // LOOPA #16,#4 ;
d1000010 // LDR R2,[R0],#1 ;
d5900010 // LDR2 R3,[R1],#1 ;
66260000 // FMUL R4,R2,R3 ;
de600010 // STR R4,[R6],#1 ;



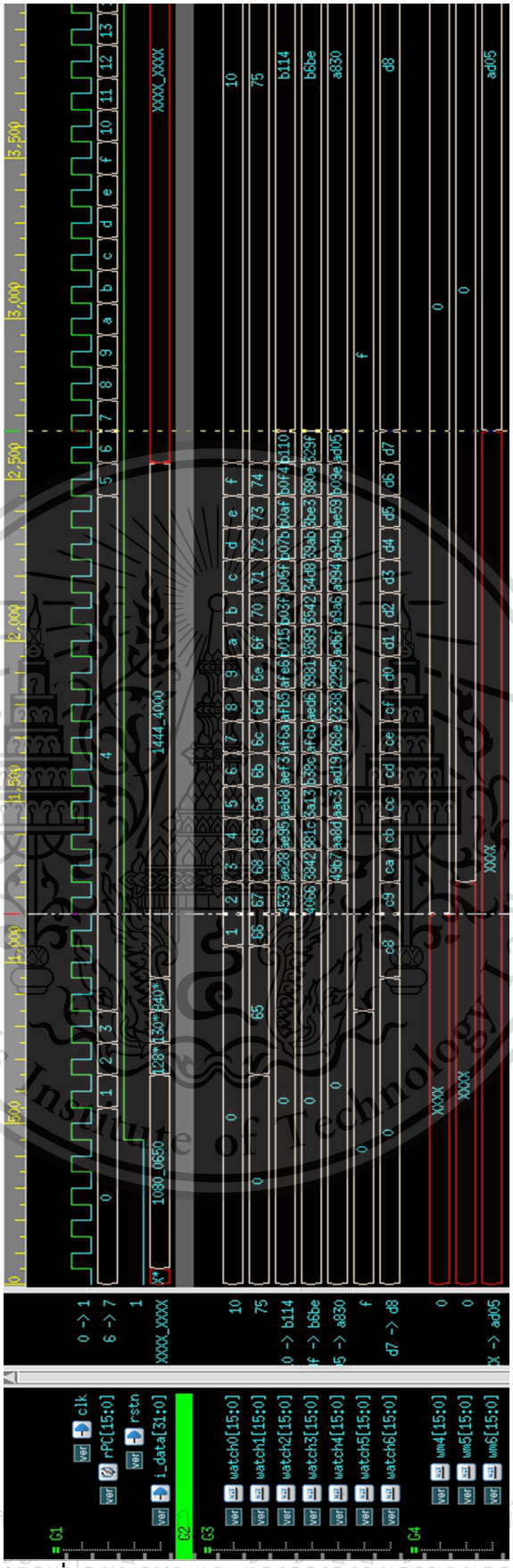
**รูปที่ 4.10** Operation LOOP สำหรับทำ WINDOW

**4.3.3. การใช้ Operation ใหม่ในการ Window**

| | |
|---|---|
| MOV R1,#101 | คือ การเก็บค่า 101 ที่ R1 |
| MOV R5,#16 | คือ การเก็บค่า 16 ที่ R5 |
| MOV R6,#200 | คือ การเก็บค่า 200 ที่ R6 |
| LOOPA #18,#1 | คือ การทำซ้ำ 18 รอบจำนวน 1 บรรทัด |
| WIN #1,#1,#1 | คือ การทำคำสั่ง window |

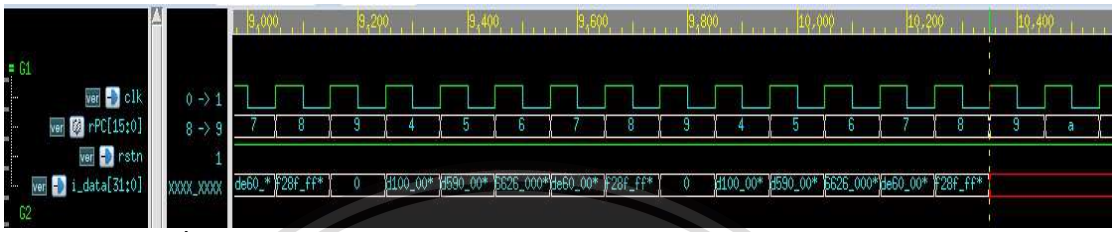
**ผลจากการแปลงเป็นเลขฐาน 16 จำนวน 8 bit**

```
10800650 // MOV R1,#101  ;
12800100 // MOV R5,#16 ;
13000c80 // MOV R6,#200 ;
84004801 // LOOPA #18,#1 ;
14444000 // WIN #1,#1,#1 ;
```



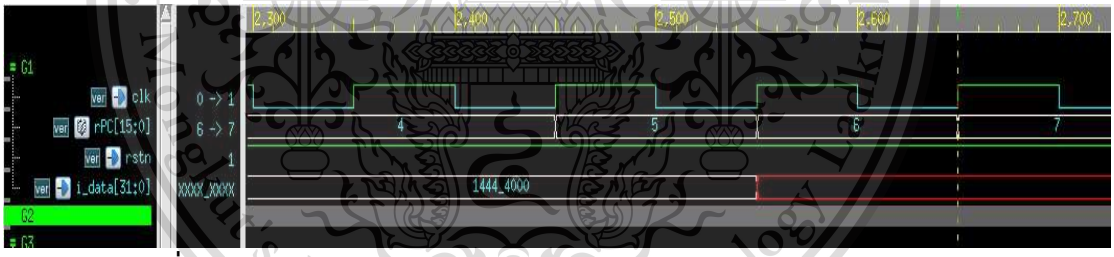

**รูปที่ 4.11** Operation เฉพาะสำหรับ Window

สรุปผลระหว่าง Operation 3 ชนิดได้แก่ การใช้ BNE สำหรับทำ Window จะใช้สัญญาณนาฬิกาประมาณ 104 สัญญาณนาฬิกา ซึ่งใช้จำนวนมากที่สุดต่อมาการใช้ LOOP สำหรับทำ Window จะใช้สัญญาณนาฬิกาประมาณ 74 สัญญาณนาฬิกาซึ่งมากเป็นอันดับที่ 2 และสุดท้ายการใช้ Operation เฉพาะทางสำหรับการทำ Window จะใช้สัญญาณนาฬิกาแค่ 26 สัญญาณนาฬิกา ซึ่งถือว่าลดลงมาหลายเท่า แต่ก็แลกมาด้วยการที่ต้องบังคับค่า Register เฉพาะ




**รูปที่ 4.12** สัญญาณนาฬิกาสุดท้ายของ Operation BNE สำหรับ WINDOW


**รูปที่ 4.13** สัญญาณนาฬิกาสุดท้ายของ Operation LOOPA สำหรับ WINDOW




**รูปที่ 4.14** สัญญาณนาฬิกาสุดท้ายของ Operation เฉพาะสำหรับ WINDOW

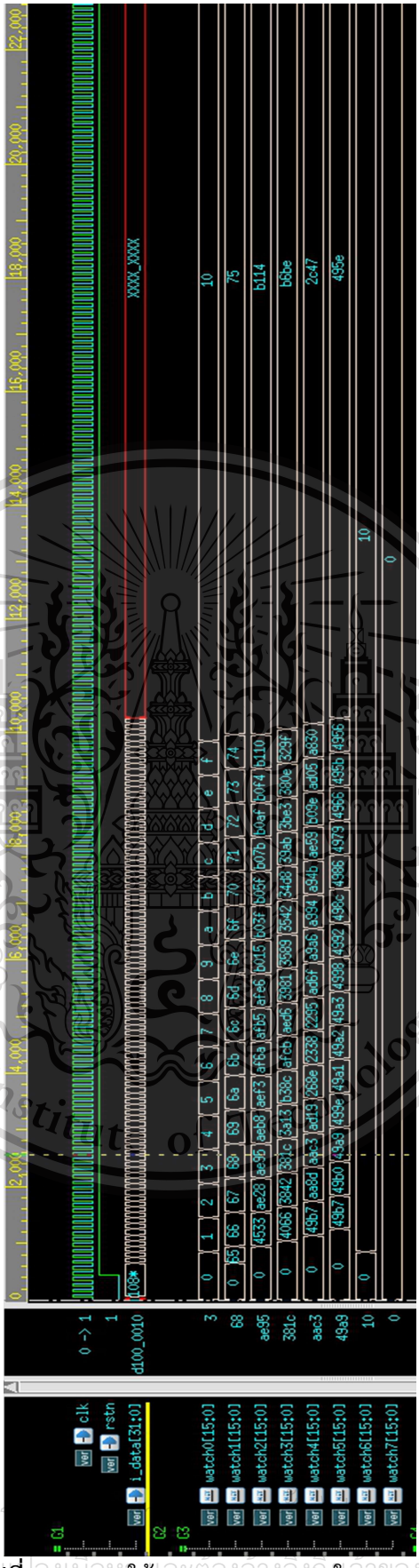
## 4.4. การใช้ operation ในการ FIR

### 4.4.1. การใช้ Operation BNE ในการ FIR

| | | |
|---|---|---|
| MOV R1,#101 | | คือ การเก็บค่า 101 ที่ R1 |
| MOV R6,#16 | | คือ การเก็บค่า 16 ที่ R6 |
| LABELA: | LDR R2,[R0],#1 | คือ การนำค่า R0 ใส่ใน R2 แล้วบวก address 1 |
| | LDR2 R3,[R1],#1 | คือ การนำค่า R1 ใน spmem ใส่ใน R3 แล้วบวก address 1 |
| | FMUL R4,R2,R3 | คือ การนำ R2*R3 แล้วเก็บไว้ที่ R4 |
| | FADD R5,R5,R4 | คือ การนำ R4+R5 แล้วเก็บไว้ที่ R5 |
| | BNE R6,R0,#LABELA | คือ การเทียบ R0 และ R6 ถ้าไม่เท่าให้กลับไป LABELA |

**ผลจากการแปลงเป็นเลขฐาน 16 จำนวน 8 bit**

```
10800650 // MOV R1,#101  ;
13000100 // MOV R6,#16 ;
d1000010 // LDR R2,[R0],#1 ;
d5900010 // LDR2 R3,[R1],#1 ;
66260000 // FMUL R4,R2,R3 ;
62d80000 // FADD R5,R5,R4  ;
f30fffc0 // BNE R6,R0,#LABELA ;
```



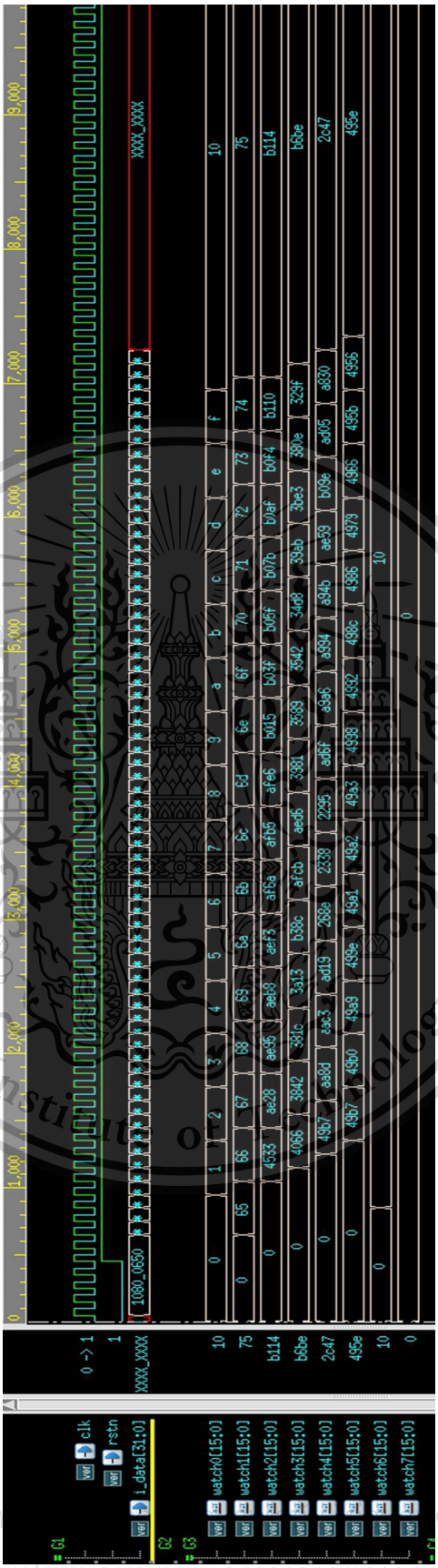

**รูปที่ 4.15** การใช้ Operation BNE ในการ FIR

**4.4.2. การใช้ Operation LOOPA ในการ FIR**

| | |
|---|---|
| MOV R1,#101 | คือ การเก็บค่า 101 ที่ R1 |
| MOV R6,#16 | คือ การเก็บค่า 16 ที่ R1 |
| LOOPA #16,#4 | คือ การทำซ้ำ 16 รอบจำนวน 4 บรรทัด |
| LDR R2,[R0],#1 | คือ การนำค่า R0 ใส่ใน R2 แล้วบวก address 1 |
| LDR2 R3,[R1],#1 | คือ การนำค่า R1 ใน spmem ใส่ใน R3 แล้วบวก address 1 |
| FMUL R4,R2,R3 | คือ การนำ R2*R3 แล้วเก็บไว้ที่ R4 |
| FADD R5,R5,R4 | คือ การนำ R4+R5 แล้วเก็บไว้ที่ R5 |

**ผลจากการแปลงเป็นเลขฐาน 16 จำนวน 8 bit**

```
10800650 // MOV R1,#101  ;
13000100 // MOV R6,#16 ;
84004004 // LOOPA #16,#4 ;
d1000010 // LDR R2,[R0],#1 ;
d5900010 // LDR2 R3,[R1],#1 ;
66260000 // FMUL R4,R2,R3 ;
62d80000 // FADD R5,R5,R4;
```




**รูปที่ 4.16** การใช้ Operation LOOP ในการ FIR

**4.4.3. การใช้ Operation ใหม่ในการ FIR**

| | |
|---|---|
| MOV R1,#101 | คือ การเก็บค่า 101 ที่ R1 |
| MOV R6,#16 | คือ การเก็บค่า 16 ที่ R6 |
| LOOPA #17,#1 | คือ การทำซ้ำ 17 รอบจำนวน 1 บรรทัด |
| FIR #1,#1 | คือ การทำฟังก์ชัน FIR |

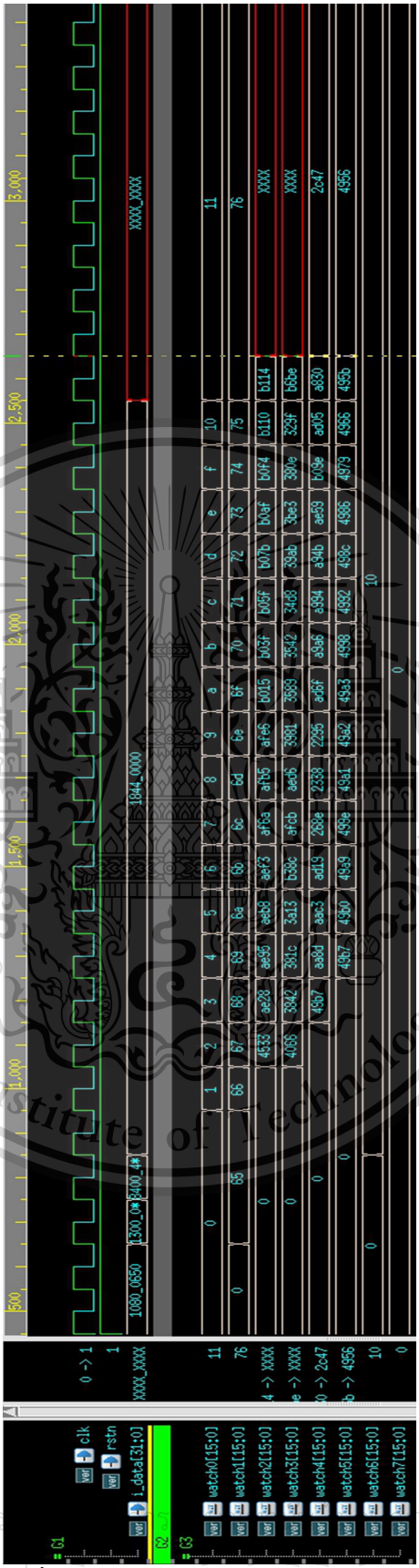
**ผลจากการแปลงเป็นเลขฐาน 16 จำนวน 8 bit**

10800650 // MOV R1,#101 ;

13000100 // MOV R6,#16 ;

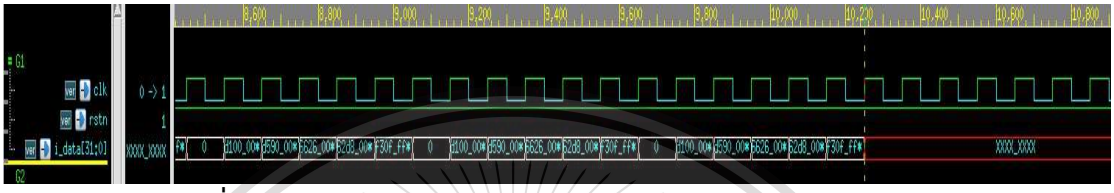
84004401 // LOOPA #17,#1 ;

18440000 // FIR #1,#1 ;

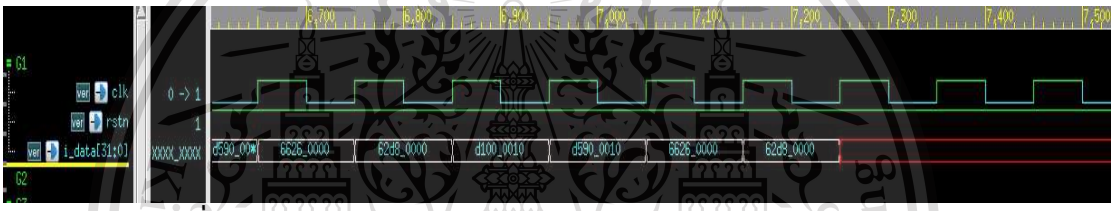


**รูปที่ 4.17** Operation เฉพาะสำหรับ FIR

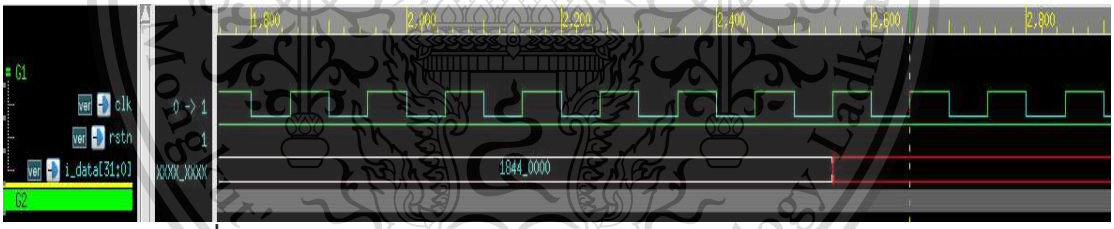
สรุปผลระหว่าง Operation 3 ชนิดได้แก่ การใช้ BNE สำหรับทำ FIR จะใช้สัญญาณนาฬิกาประมาณ 103 สัญญาณนาฬิกา ซึ่งใช้จำนวนมากที่สุด ต่อมาการใช้ LOOP สำหรับทำ FIR จะใช้สัญญาณนาฬิกาประมาณ 73 สัญญาณนาฬิกาซึ่งมากเป็นอันดับที่ 2 และสุดท้ายการใช้ Operation เฉพาะทางสำหรับการทำ FIR จะใช้สัญญาณนาฬิกาแค่ 26 สัญญาณนาฬิกาซึ่งถือว่าลดลงมาหลายเท่าแต่ก็แลกมาด้วยการที่ต้องบังคับค่า Register เฉพาะจึงทำให้ลดความ Flexible ลงแต่แทนที่ด้วยคุณภาพ



**รูปที่ 4.18** สัญญาณนาฬิกาสุดท้ายของ Operation BNE สำหรับ FIR



**รูปที่ 4.19** สัญญาณนาฬิกาสุดท้ายของ Operation LOOPA สำหรับ FIR



**รูปที่ 4.20** สัญญาณนาฬิกาสุดท้ายของ Operation เฉพาะสำหรับ FIR

## 4.5. เปรียบเทียบสัญญาณนาฬิการะหว่าง CPU กับ audio-CPU

เป็นการทำ MFCC จำนวน 1 Frame โดยมีขั้นตอนต่อไปนี้

1. Framing

เนื่องจากเสียงโดยปกติแล้วจะไม่หยุดนิ่ง การนำเสียงที่ยังไม่ผ่านการ frame นั้น จะทำให้การทำ FFT เกิดความผิดเพี้ยนได้เราจึงสมมติว่าให้เสียงที่นำมาผ่านกระบวนการนี้หยุดนิ่งและแต่ละเฟรมมีขนาดเท่ากับขนาด FFT ที่ต้องการ แต่ในการทำ MFCC ของเราเราจะนำข้อมูลเสียงที่ผ่านการ framing และ normalize มาใช้เลย

2. Window

หลังจากนำไฟล์เสียงมาเฟรมแล้วเราจะเอาเสียงนี้มาทำการ Window เราใช้ Hann window ในกระบวนการทำ MFCC นี้ โดยจะเก็บค่าคงที่ Hann window ไว้ใน Ram

3. FFT

หลังจากเสียงผ่านการ window แล้วจากนั้นจะมาทำการ FFT ในที่นี่ใช้ FFT's Butterfly 2048

4. Power

เอาเสียงจากการทำ FFT มาหากำลังเพื่อให้ค่าที่ได้เป็นจำนวนจริง

5. MEL – spaced filter bank

เอาเสียงที่ได้จากการทำ power มาผ่านฟิลเตอร์ โดยค่าคงที่ filter จะถูกเก็บไว้ใน Ram เช่นกัน

6. Logarithm

นำเสียงที่ผ่านการกรองมาทำ log เพื่อเตรียมไปเข้า DCT-Filter

7. DCT-Filter

เป็นขั้นตอนสุดท้ายที่จะได้ coefficient ที่เราต้องการโดยการเอาเสียงมาผ่าน DCT-Filter โดยค่าคงที่ DCT จะถูกเก็บไว้ใน Ram เช่นกัน

### 4.1.1 Hann Window

| | |
|---|---|
| MOV R0,#0 | คือ การเก็บค่า 0 ที่ R0 |
| MOV R1,#0 | คือ การเก็บค่า 0 ที่ R1 |
| MOV R6,#3000 | คือ การเก็บค่า 3000 ที่ R6 |
| LOOPA #2050,#1 | คือ การทำซ้ำ 2050 รอบจำนวน 1 บรรทัด |
| WIN #1,#1,#1,#0 | คือ การทำคำสั่ง window |

**ผลจากการแปลงเป็นเลขฐาน 16 จำนวน 8 bit**

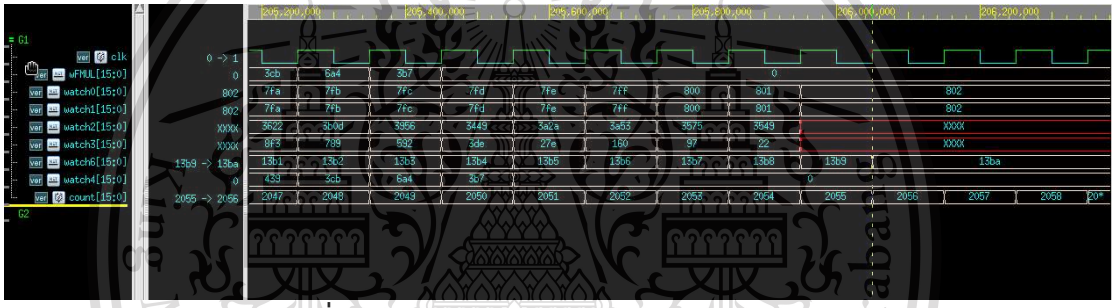
10000000 // MOV R0,#0;

10800000 // MOV R1,#0;

1300bb80 // MOV R6,#3000;

84200801 // LOOPA #2050,#1;

14400880 // WIN #1,#1,#1,#0;




**รูปที่ 4.21** timing diagram ของการทำ window

เป็นการทำ Hann window ที่มี hop size เท่ากับ 15 ms FFT size ขนาด 2048 และ sample rate เท่ากับ 44100 วิธีการทำงานของ audio CPU คือใช้ loop และคำสั่ง WIN โดยคำสั่ง WIN จะทำการโหลดค่ามา 2 ค่ามาทำการคูณกันและเก็บไว้ที่ address ที่กำหนดไว้

Input 1 frame จำนวน 2048 ตัวและค่าคงที่ Hann Window จำนวน 2048 ตัว จำนวน clk ในการทำ Window 1 frame ใช้ 2055 clk และขนาด memory ที่ใช้ 5 คำสั่ง ขนาดแต่ละคำสั่งเท่ากับ 4 byte เพราะฉะนั้นใช้ทั้งหมด 20 byte

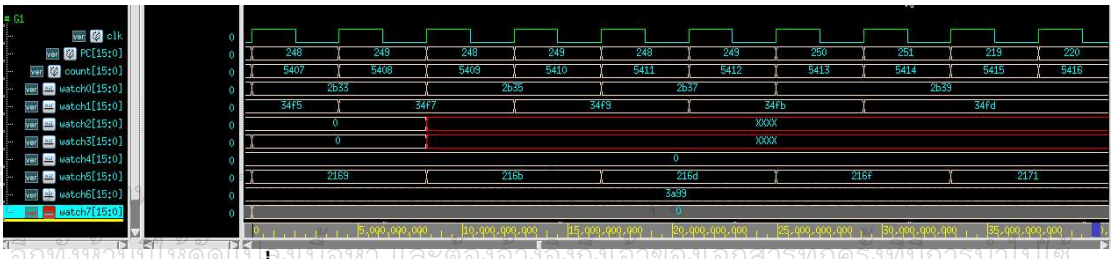
**4.1.2 FFT's Butterfly 2048**

| | |
|---|---|
| NOP | คือ การ delay เวลาให้ cpu ทำคำสั่งก่อนหน้า |
| CALL fft_even1 | คือ การเรียก function ทำ fft ที่ผู้เขียนกำหนด |
| CALL winE1 | คือ การเรียก function ทำ window ที่ผู้เขียนกำหนด |
| NOP | คือ การ delay เวลาให้ cpu ทำคำสั่งก่อนหน้า |
| LOOPA #1026,#1 | คือ การทำซ้ำ 1026 รอบจำนวน 1 บรรทัด |
| WIN #2,#1,#2,#0 | คือ การทำคำสั่ง window |
| CALL winE2 | คือ การเรียก function ทำ window ที่ผู้เขียนกำหนด |
| NOP | คือ การ delay เวลาให้ cpu ทำคำสั่งก่อนหน้า |
| LOOPA #1028,#1 | คือ การทำซ้ำ 1028 รอบจำนวน 1 บรรทัด |
| WIN #2,#1,#2,#0 | คือ การทำคำสั่ง window |
| CALL winE3 | คือ การเรียก function ทำ window ที่ผู้เขียนกำหนด |
| NOP | คือ การ delay เวลาให้ cpu ทำคำสั่งก่อนหน้า |
| LOOPA #1026,#1 | คือ การทำซ้ำ 1026 รอบจำนวน 1 บรรทัด |
| WIN #2,#1,#2,#0 | คือ การทำคำสั่ง window |
| CALL winE4 | คือ การเรียก function ทำ window ที่ผู้เขียนกำหนด |
| NOP | คือ การ delay เวลาให้ cpu ทำคำสั่งก่อนหน้า |
| LOOPA #1028,#1 | คือ การทำซ้ำ 1028 รอบจำนวน 1 บรรทัด |
| WIN #2,#1,#2,#0 | คือ การทำคำสั่ง window |
| CALL fft_even2 | คือ การเรียก function ทำ fft ที่ผู้เขียนกำหนด |
| NOP | คือ การ delay เวลาให้ cpu ทำคำสั่งก่อนหน้า |
| NOP | คือ การ delay เวลาให้ cpu ทำคำสั่งก่อนหน้า |
| CALL fft_odd1 | คือ การเรียก function ทำ fft ที่ผู้เขียนกำหนด |
| CALL winO1 | คือ การเรียก function ทำ window ที่ผู้เขียนกำหนด |
| NOP | คือ การ delay เวลาให้ cpu ทำคำสั่งก่อนหน้า |
| LOOPA #1026,#1 | คือ การทำซ้ำ 1026 รอบจำนวน 1 บรรทัด |
| WIN #2,#2,#2,#1 | คือ การทำคำสั่ง window |

| | |
|---|---|
| CALL winO2 | คือ การเรียก function ทำ window ที่ผู้เขียนกำหนด |
| NOP | คือ การ delay เวลาให้ cpu ทำคำสั่งก่อนหน้า |
| LOOPA #1028,#1 | คือ การทำซ้ำ 1028 รอบจำนวน 1 บรรทัด |
| WIN #2,#2,#2,#1 | คือ การทำคำสั่ง window |
| CALL winO3 | คือ การเรียก function ทำ window ที่ผู้เขียนกำหนด |
| NOP | คือ การ delay เวลาให้ cpu ทำคำสั่งก่อนหน้า |
| LOOPA #1026,#1 | คือ การทำซ้ำ 1026 รอบจำนวน 1 บรรทัด |
| WIN #2,#2,#2,#1 | คือ การทำคำสั่ง window |
| CALL winO4 | คือ การเรียก function ทำ window ที่ผู้เขียนกำหนด |
| NOP | คือ การ delay เวลาให้ cpu ทำคำสั่งก่อนหน้า |
| LOOPA #1028,#1 | คือ การทำซ้ำ 1028 รอบจำนวน 1 บรรทัด |
| WIN #2,#2,#2,#1 | คือ การทำคำสั่ง window |
| CALL fft_odd2 | คือ การเรียก function ทำ fft ที่ผู้เขียนกำหนด |
| NOP | คือ การ delay เวลาให้ cpu ทำคำสั่งก่อนหน้า |
| NOP | คือ การ delay เวลาให้ cpu ทำคำสั่งก่อนหน้า |
| MOV R0,#4304 | คือ การเก็บค่า 4304 ที่ R0 |
| MOV R1,#3 | คือ การเก็บค่า 3 ที่ R1 |
| LOOPA #2048,#1 | คือ การทำซ้ำ 2048 รอบจำนวน 1 บรรทัด |
| REOR #2 | คือ การสลับตำแหน่งข้อมูล 2048 ค่า |
| STR R2,[R1],#1 | คือ การนำค่า R1 ใส่ใน R2 แล้วบวก address 1 |
| MOV R0,#6504 | คือ การเก็บค่า 6504 ที่ R0 |
| MOV R1,#2203 | คือ การเก็บค่า 2203 ที่ R1 |
| LOOPA #2048,#1 | คือ การทำซ้ำ 2048 รอบจำนวน 1 บรรทัด |
| REOR #2 | คือ การสลับตำแหน่งข้อมูล 2048 ค่า |
| STR R2,[R1],#1 | คือ การนำค่า R1 ใส่ใน R2 แล้วบวก address 1 |

**ผลจากการแปลงเป็นเลขฐาน 16 จำนวน 8 bit**

00000000 // NOP;
c401ba00 // CALL fft_even1;
c4023200 // CALL winE1;
00000000 // NOP;
84100801 // LOOPA #1026,#1;
14800900 // WIN #2,#1,#2,#0;
c4023a00 // CALL winE2;
00000000 // NOP;
84101001 // LOOPA #1028,#1;
14800900 // WIN #2,#1,#2,#0;
c4024200 // CALL winE3;
00000000 // NOP;
84100801 // LOOPA #1026,#1;
14800900 // WIN #2,#1,#2,#0;
c4024a00 // CALL winE4;
00000000 // NOP;
84101001 // LOOPA #1028,#1;
14800900 // WIN #2,#1,#2,#0;
c401d800 // CALL fft_even2;
00000000 // NOP;
00000000 // NOP;
c401f600 // CALL fft_odd1;
c4025200 // CALL winO1;
00000000 // NOP;
84100801 // LOOPA #1026,#1;
14801108 // WIN #2,#2,#2,#1;
c4025a00 // CALL winO2;
00000000 // NOP;
84101001 // LOOPA #1028,#1;
14801108 // WIN #2,#2,#2,#1;
c4026200 // CALL winO3;
00000000 // NOP;
84100801 // LOOPA #1026,#1;
14801108 // WIN #2,#2,#2,#1;
c4026a00 // CALL winO4;
00000000 // NOP;
84101001 // LOOPA #1028,#1;
14801108 // WIN #2,#2,#2,#1;
c4021400 // CALL fft_odd2;
00000000 // NOP;
00000000 // NOP;
10010d00 // MOV R0,#4304
10800030 // MOV R1,#3
84200001 // LOOPA #2048,#1
95000000 // REOR #2
dd100010 // STR R2,[R1],#1
10019680 // MOV R0,#6504
108089b0 // MOV R1,#2203
84200001 // LOOPA #2048,#1
95000000 // REOR #2
dd100010 // STR R2,[R1],#1




**รูปที่ 4.22** timing diagram ของการทำ fft

เป็นการทำ FFT’s Butterfly มี input เท่ากับ 2048(real + imagine เท่ากับ 1 ชุด) output เท่ากับ 2048 ตัว ในการทำ FFT จะใช้ Window มาใช้คู่กับกับชุดคำสั่ง FADD2 และ FSUB2 ด้วยและจะทำซ้ำทั้งหมด 11 รอบต่อการทำ FFT ใน python 1 รอบ จำนวน clk ในการทำใช้เท่ากับ 12,411 clk ต่อ 1 รอบและขนาด memory ที่ใช้ 21 คำสั่ง ขนาดแต่ละคำสั่งเท่ากับ 4 byte และทำจำนวน 11 รอบ เพราะฉะนั้นใช้ทั้งหมด 924 byte

**4.1.3 Power**

| | |
|---|---|
| MOV R0,#0 | คือ การเก็บค่า 0 ที่ R0 |
| MOV R1,#2000 | คือ การเก็บค่า 2000 ที่ R1 |
| MOV R6,#4000 | คือ การเก็บค่า 4000 ที่ R6 |
| LOOPA #1028,#2 | คือ การทำซ้ำ 1028 รอบจำนวน 2 บรรทัด |
| SQR1 #1 | คือ คำสั่งทำ power ส่วนที่ 1 |
| SQR2 #1,#1 | คือ คำสั่งทำ power ส่วนที่ 2 |

**ผลจากการแปลงเป็นเลขฐาน 16 จำนวน 8 bit**

10000000 // MOV R0,#0;

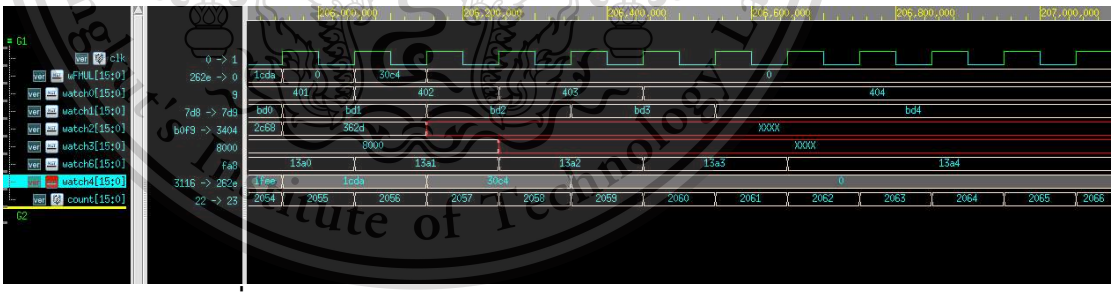
10807d00 // MOV R1,#2000;

1300fa00 // MOV R6,#4000;

84101002 // LOOPA #1028,#2;

48400000 // SQR1 #1;

4c440000 // SQR2 #1,#1;



**รูปที่ 4.23** timing diagram ของการทำ power

เป็นการหา Power จำนวน 1025 ตัว(real + imagine เท่ากับ 1 ตัว) โดยใช้คำสั่ง SQR1 ร่วมกับ SQR2 จำนวน clk ที่ใช้ต่อการทำ Power 1 frame เท่ากับ 2062 clk และ ขนาด memory ที่ใช้ 6 คำสั่ง ขนาดแต่ละคำสั่งเท่ากับ 4 byte เพราะฉะนั้นใช้ทั้งหมด 24 byte

### 4.1.4 Filter bank

| | |
|---|---|
| MOV R6,#10000 | คือ การเก็บค่า 10000 ที่ R6 |
| LOOPA #1027,1 | คือ การทำซ้ำ 1027 รอบจำนวน 1 บรรทัด |
| FIR #1,#1 | คือ การทำคำสั่ง FIR |
| STR R5,[R6],#1 | คือ การนำค่า R6 ใส่ใน R5 แล้วบวก address 1 |

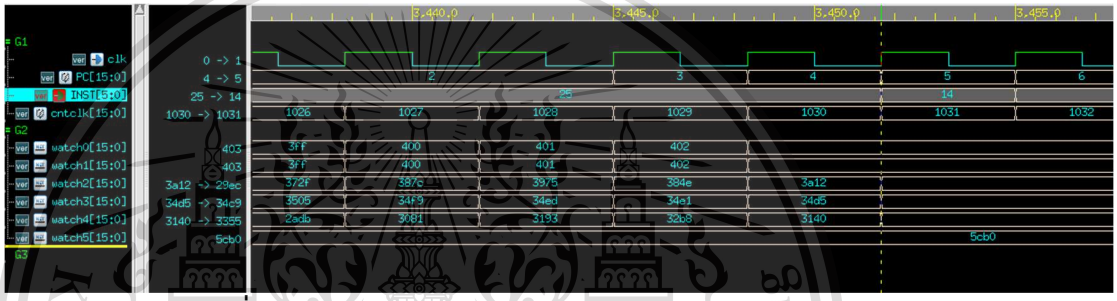
**ผลจากการแปลงเป็นเลขฐาน 16 จำนวน 8 bit**

13027100 // MOV R6,#10000;

84100c01 // LOOPA #1027,1;

18440000 // FIR #1,#1;

dee00010 // STR R5,[R6],#1;




**รูปที่ 4.24** timing diagram สำหรับการทำ Filter bank

เป็นการเอาสัญญาณเสียงมาเข้า Filter bank sample rate เท่ากับ 44100 มี input เสียงเท่ากับ 1025 ตัวไปคูณกับค่าคงที่ filter จำนวน 1025 ตัว โดยใช้คำสั่ง FIR output ที่ออกมาจะได้ 1 ค่าและทำซ้ำ 10 รอบต่อการทำ Filter 1 frame จำนวน clk ที่ใช้ต่อการทำ Filter bank 1 frame เท่ากับ 10310 clk และ ขนาด memory ที่ใช้ 4 คำสั่ง ขนาดแต่ละคำสั่งเท่ากับ 4 byte เพราะฉะนั้นใช้ทั้งหมด 16 byte

### 4.1.5 Logarithm

| | |
|---|---|
| MOV R3,#3.009 | คือ การเก็บค่า 3.009 ที่ R3 |
| MOV R5,#10000 | คือ การเก็บค่า 10000 ที่ R5 |
| LOOPA #10,#6 | คือ การทำซ้ำ 10 รอบจำนวน 6 บรรทัด |
| LDR R0,R2,#1 | คือ การนำค่า R2 ใส่ใน R0 แล้วบวก address 1 |
| NORML | คือ การทำ normalize |
| LOOPM #0.998 #1 | คือ ทำลูปจนค่ามากกว่าหรือเท่ากับ 0.998 |
| Flog | คือ การทำ logarithm |
| FMUL R1,R1,R3 | คือ การนำ R1*R3 แล้วเก็บไว้ที่ R1 |
| STR R1,[R5],#1 | คือ การนำค่า R5 ใส่ใน R1 แล้วบวก address 1 |
| MOV R7,#51966 | คือ การเก็บค่า 51966 ที่ R7 |

**ผลจากการแปลงเป็นเลขฐาน 16 จำนวน 8 bit**

11842050 // MOV R3,#3.009;

12827100 // MOV R5,#10000;

84002806 // LOOPA #10,#6;

d0200010 // LDR R0,R2,#1;

B4000000 //NORML

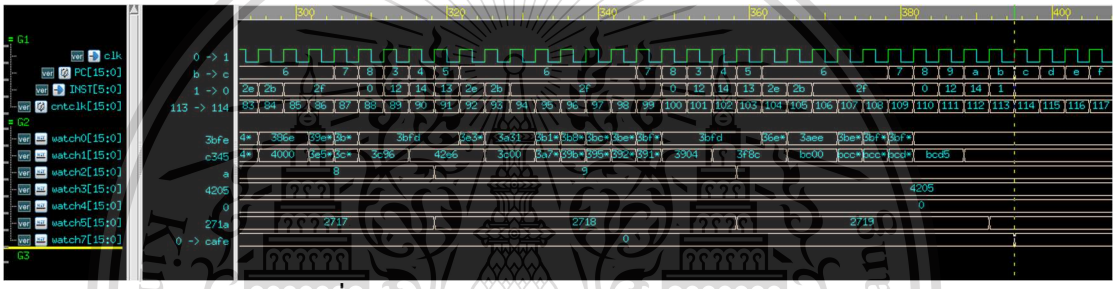
68EFF001 //LOOPM #0.998 #1

B8000000 // Flog

64960000 // FMUL R1,R1,R3; //convert to base 10 * 10

dcd00010 // STR R1,[R5],#1;

138cafe0 // MOV R7,#51966; //check finish



**รูปที่ 4.25** timing diagram สำหรับการทำ log

เป็นการเอาสัญญาณเสียงหลังจากการทำ Filter มาเข้า log โดยวิธีการทำ log จะเป็นไปตามคำสั่งข้างบนคำสั่งหลักๆในการทำ log จะเป็นการ normalize ตามด้วย LOOPM และ Flog ในที่นี้ log ที่ได้ในตอนแรกจะเป็นฐาน 2 ต้องนำไปคูณกับค่าคงที่เพื่อแปลงจากฐาน 2 เป็นฐาน 10 ตามลำดับ จำนวน clk ที่ใช้ต่อการทำ log 1 frame เท่ากับ 113 clk และขนาด memory ที่ใช้ 9 คำสั่ง ขนาดแต่ละคำสั่งเท่ากับ 4 byte เพราะฉะนั้นใช้ทั้งหมด 36 byte

### 4.1.6 DCT filter

| | |
|---|---|
| MOV R6,#10000 | คือ การเก็บค่า 10000 ที่ R6 |
| LOOPA #13,#1 | คือ การทำซ้ำ 13 รอบจำนวน 1 บรรทัด |
| FIR #1,#1 | คือ การทำคำสั่ง FIR |
| STR R5,[R6],#1 | คือ การนำค่า R6 ใส่ใน R5 แล้วบวก address 1 |

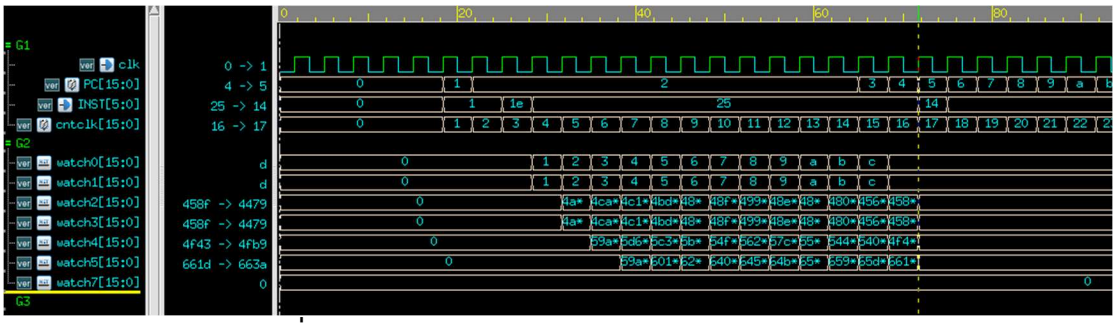
**ผลจากการแปลงเป็นเลขฐาน 16 จำนวน 8 bit**

13027100 // MOV R6,#10000;

84003401 // LOOPA #13,#1;

18440000 // FIR #1,#1;

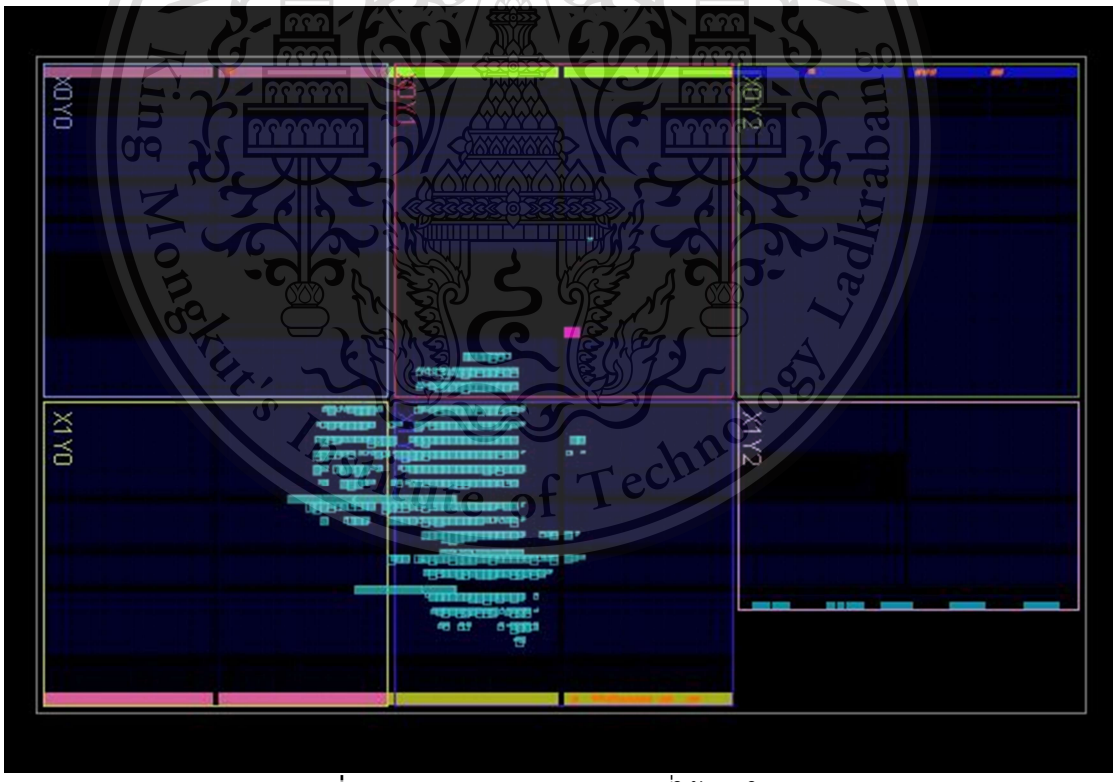
dee00010 // STR R5,[R6],#1;



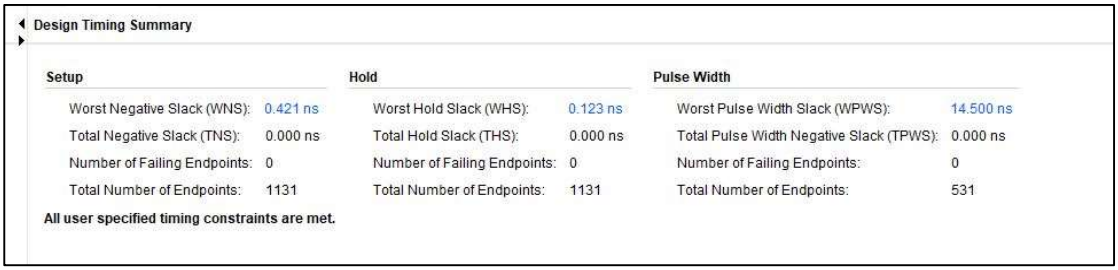

**รูปที่ 4.26** timing diagram สำหรับการทำ DCT filter

เป็นการเอาไฟล์เสียงไปทำ DCT Filter เพื่อสร้าง Coefficient ที่ต้องการโดยใช้คำสั่ง FIR การทำงานคือมี Input 10 ตัวไปคูณกับค่าคงที่ DCT 10 ตัว จะได้ coefficient 1 ตัวโดยที่ 1 frame จะใช้ DCT ทั้งหมด 40 ตัวเพราะฉะนั้น จำนวน clk ที่ใช้ต่อการทำ DCT 1 ตัวเท่ากับ 17 clk การทำ DCT 1 frame จึงใช้ทั้งหมด 680 clk และขนาด memory ที่ใช้ 4 คำสั่ง ขนาดแต่ละคำสั่งเท่ากับ 4 byte เพราะฉะนั้นใช้ทั้งหมด 16 byte

**แผนผังวงจรภายใน**




**รูปที่ 4.27** ตำแหน่ง logic gate ที่ใช้ภายใน chip FPGA



Design Timing Summary

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.421 ns | Worst Hold Slack (WHS): | 0.123 ns | Worst Pulse Width Slack (WPWS): | 14.500 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 1131 | Total Number of Endpoints: | 1131 | Total Number of Endpoints: | 531 |

All user specified timing constraints are met.

**รูปที่ 4.28** Design Timing Summary

จากรูปที่ 4.28 จะเห็นได้ว่ามีเส้นทางวงจรที่ช้าที่สุดอยู่ที่ 0.0316 ns ความเร็ว clk สูงสุดที่ใช้ประมาณ 31 MHz

1. Slice Logic

| Site Type | Used | Fixed | Available | Util% |
|---|---|---|---|---|
| Slice LUTs* | 2098 | 0 | 20800 | 10.09 |
| LUT as Logic | 2098 | 0 | 20800 | 10.09 |
| LUT as Memory | 0 | 0 | 9600 | 0.00 |
| Slice Registers | 561 | 0 | 41600 | 1.35 |
| Register as Flip Flop | 513 | 0 | 41600 | 1.23 |
| Register as Latch | 48 | 0 | 41600 | 0.12 |
| F7 Muxes | 0 | 0 | 16300 | 0.00 |
| F8 Muxes | 0 | 0 | 8150 | 0.00 |

**รูปที่ 4.29** แสดงทรัพยากรที่ใช้ในการออกแบบ

จากรูปที่ 4.29 การออกแบบใช้ combination circuit ทั้งหมด 2098 ตัวและใช้ Flip flop ทั้งหมด 561 ตัว

2. Memory

| Site Type | Used | Fixed | Available | Util% |
|---|---|---|---|---|
| Block RAM Tile | 0 | 0 | 50 | 0.00 |
| RAMB36/FIFO* | 0 | 0 | 50 | 0.00 |
| RAMB18 | 0 | 0 | 100 | 0.00 |

**รูปที่ 4.30** แสดงจำนวนหน่วยความจำที่ใช้

จากรูปที่ 4.30 แสดงว่าใช้ ram ทั้งหมด 2 ตัว

3. DSP

| Site Type | Used | Fixed | Available | Util% |
|---|---|---|---|---|
| DSPs | 1 | 0 | 90 | 1.11 |
| DSP48E1 only | 1 | | | |

**รูปที่ 4.31** แสดงจำนวนวงจร DSP ที่ใช้

ใช้วงจร DSP ในบอร์ด FPGA ไป 1.11%

**ตารางที่ 4.2** สรุปการใช้สัญญาณนาฬิการะหว่าง ARM CPU และ Audio-CPU ใน 1 frame

| Function | ARM CPU | Audio-CPU |
|---|---|---|
| Window | 34,836 clk | 2055 clk |
| FFT[4096] | 253,120 clk | 136,484 clk |
| Power[2048] | 20,518 clk | 2,062 clk |
| Dot filter[10] | 143,500 clk | 10,310 clk |
| Audio_log[10] | 1,088 clk | 113 clk |
| DCT filter[40] | 8,560 clk | 680 clk |
| Total | 461,622 clk | 151,704 clk |

จากตารางจะเห็นได้ว่า Audio CPU ใช้ clk เพียงแค่ 32.8% จาก ARM CPU

# บทที่ 5

# สรุปผลและข้อเสนอแนะ

## 5.1 สรุปผลการทำดำเนินงาน

จากผลการดำเนินงานทั้งหมดที่ได้กล่าวมาในบทก่อนหน้าพบว่าการสร้าง Operation ใหม่สำหรับการทำงานเฉพาะทางมีประสิทธิภาพดีกว่าการนำ Operation พื้นฐานมาใช้ทั้งในด้านความเร็วและหน่วยความจำที่ใช้ เมื่อเริ่มการทดลองโดยนำ ARM CPU มาทำ MFCC เทียบกับ Audio CPU เสียงที่นำมาใช้ทดลองมีขนาด 933 frames, hop size ขนาด 15 ms, Sampling rate 44100 โดยนำผลลัทธ์ค่า Mel frequency coefficient มาเทียบกันในแต่ละ frame พบว่าผลลัพธ์มีความคลาดเคลื่อนเล็กน้อยประมาณ ±6% ความคลาดเคลื่อนเกิดจากการปัดเลขทศนิยมของ Audio CPU เนื่องจากใช้ระบบการคำนวณแบบ 16 bit ความเร็วในการทำ MFCC
Audio CPU มีความเร็วกว่า ARM CPU อยู่ประมาณ 3 เท่าและขนาดหน่วยความจำที่ใช้ใช้ไปเท่ากับ 3.124 KB จากผลที่กล่าวมาข้างต้นพบว่า Audio CPU มีประสิทธิภาพเฉพาะทางที่ดีกว่าและยังใช้ทรัพยากรน้อยกว่าอีกด้วย

การสร้าง Operation พิเศษนั้นเกิดจากเรากำหนดให้ Register แต่ละตัวให้ทำงานตายตัวในแต่ละคำสั่งพิเศษซึ่งผู้ใช้จะไม่สามารถสั่งให้ Register ทำงานได้อย่างอิสระในระหว่างทำคำสั่งพิเศษซึ่งเป็นผลให้ CPU สามารถทำพร้อมกันหลายๆคำสั่งได้ภานใน 1 สัญญาณนาฬิกา

## 5.2 ปัญหาและอุปสรรค

5.2.1 การสร้าง Operation พิเศษจะมีความซับซ้อนในการสร้างมากกว่าคำสั่งปกติเนื่องจากต้องจัดการกับ Register ให้ทำงานสัมพันธ์กันและต้องคำนึงถึง pipeline สำหรับคำสั่งที่จะต้องนำมาใช้ร่วมกันด้วย

5.2.2 ในการทำการทดลองจริงต้องคำนึงถึงลูปที่ใช้กับคำสั่งพิเศษเนื่องจากในคำสั่งพิเศษการสั่งงานรอบแรกหรือรอบ 2 คำตอบอาจจะยังไม่ถูกต้องทั้งนี้ขึ้นอยู่กับ pipeline ของแต่ละคำสั่ง

5.2.3 วงงจรคูณเลข, บวกเลขทศนิยมมีความผิดพลาดบ่อยเนื่องจากการแปลงระบบ 32 บิตมาเป็น 16 บิต ค่อนข้างซับซ้อนและต้องครอบคุมค่าตัวเลขตั้งแต่น้อยมากๆ – ค่ามากๆ

## 5.3 ข้อเสนอแนะ

5.3.1 พัฒนาให้ CPU มี path delay ให้สั้นลงเพื่อเพิ่มประสิทธิภาพให้สามารถใช้สัญญาณนาฬิกาได้เร็วขึ้น

5.3.2 พัฒนาให้ CPU ใช้ทรัพยากรให้น้อยลงไปเรื่อย ๆ เพื่อให้ใช้พลังงานน้อยลง

# เอกสารอ้างอิง


[1] ไม่พบชื่อผู้เขียน(2017) : “python” สืบค้นเมื่อ 15 กันยายน 2563 จาก https://saixiii.com/python-programming/

[2] ไม่พบชื่อผู้เขียน(2017) : “Verilog HDL” สืบค้นเมื่อ 15 กันยายน 2563 จาก http://shuwernpi.blogspot.com/2017/11/verilog-hdl.html

[3] ไม่พบชื่อผู้เขียน(2018) : “FPGA อุปกรณ์โลจิกแบบโปรแกรมได้” สืบค้นเมื่อ 22 กันยายน 2563 จาก https://www.mindphp.com/บทความ/technology/5240-fpga.html

[4] ไม่พบชื่อผู้เขียน(2016) : “ระดับของภาษาคอมพิวเตอร์ ตอนที่ 2 ภาษาแอสเซมบลี” สืบค้นเมื่อ 6 ตุลาคม 2563 จาก https://milerdev.blogspot.com/2016/11/2-assembly-language.html

[5] ไม่พบชื่อผู้เขียน(2018) :“MFCC implementation and tutorial” สืบค้นเมื่อ 22 พฤศจิกายน 2563 จาก https://www.kaggle.com/ilyamich/mfcc-implementation-and-tutorial

[6] พินิจ กำหอม: “DFT และ FFT” สืบค้นเมื่อ 15 มีนาคม 2564 จาก http://webstaff.kmutt.ac.th/~thorin.the/ENE208/Lectures/dft_fft_pinit.pdf