CHAPTER 5

A Closer Look at Instruction Set Architectures

5.1 บทนำ

ในบทที่ 4 ว่าคำสั่งของเครื่องประกอบด้วย opcodes และ operand opcodes ระบุการดำเนินการถูกที่ใช้ในการประมวลผล ตัวถูกดำเนินการหรือ operand กำหนดรีจีสเตอร์หรือหน่วยความจำ ตำแหน่งของข้อมูล ทำไมเมื่อเรามีภาษาเช่น C++ Java และ Ada เป็นต้น แล้วเราต้องให้ความสำคัญกับคำสั่งเครื่องอีกหรือ? โปรแกรมในระดับสูงภาษา จะไม่ค่อยตระหนักถึงหัวข้อกล่าวถึงในบทที่ 4 (หรือในบทนี้) เพราะระดับสูงภาษาซ่อนรายละเอียดของสถาปัตยกรรมจากโปรแกรมเมอร์ แต่ก็มีงานอีกส่วนหนึ่งที่ต้องการพื้นหลังภาษาเพราะพวกเขาต้องการโปรแกรมเมอร์ภาษาแอสเซมบลี แต่เนื่องจากพวกเขาต้องการคนที่สามารถเข้าใจสถาปัตยกรรมคอมพิวเตอร์ ในการเขียนโปรแกรมมีประสิทธิภาพมากขึ้นและมีประสิทธิภาพมากขึ้น

ในบทนี้ได้ขยายหัวข้อที่นำเสนอในครั้งสุดท้าย บทที่มีวัตถุประสงค์เพื่อให้มีมากขึ้น ดูรายละเอียดที่ชุดคำสั่งเครื่อง ที่แตกต่างกัน ประเภทคำสั่งและประเภทตัวถูกดำเนินการ เช่นเดียวกับวิธีที่คำสั่งการเข้าถึงข้อมูลในหน่วยความจำ จะเห็นว่าการเปลี่ยนแปลงในชุดคำสั่งนั้นมีความสำคัญในการแยกแยะสถาปัตยกรรมคอมพิวเตอร์ที่แตกต่างกัน ทำความเข้าใจอย่างไร ชุดสั่งได้รับการออกแบบและวิธีการทำงานของคำสั่งเหล่านี้สามารถช่วยได้ เมื่อเข้าใจรายละเอียดที่ซับซ้อนยิ่งขึ้นของสถาปัตยกรรมของตัวเครื่อง

5.2 INSTRUCTION FORMATS

ชุดของคำสั่งมีความแตกต่างกันตามลักษณะต่อไปนี้

- การเก็บตัวดำเนินการ(Operand storage) ข้อมูลอาจจะถูกเก็บอยู่ในโครงสร้างสแต็กหรือรีจีสเตอร์ต่างๆ หรือทั้งสองอย่าง
- จำนวนของ operand ต่อคำสั่ง คำสั่งที่ไม่ต้องการ operand หรือต้องการ 1 หรือ 2 ตัว
- ที่เก็บ operand (Operand location) คำสั่งสามารถแบ่งเป็น register-to-register, register-to-memory, หรือ memory-to-memory
- Operations ไม่ได้จำกัดเฉพาะชนิดของ Operation แต่จะดูที่คำสั่งที่สามารถเข้าถึงหน่วยความจำ และบางคำสั่งไม่สามารถเข้าถึงได้
- ชนิดและขนาดของ Operands (เนื่องจาก Operands สามารถเป็นได้ทั้งแอดเดรส ตัวเลข หรือแม้แต่ตัวอักษร)

5.2.1 การออกแบบการตัดสินใจสำหรับชุดคำสั่ง

ในขั้นตอนการออกแบบสถาปัตยกรรมคอมพิวเตอร์ รูปแบบของชุดคำสั่งจะต้องถูกกำหนดขึ้นมาก่อน มีความยากในการเลือกรูปแบบ เนื่องจากชุดคำสั่งจะต้องเข้ากันได้กับสถาปัตยกรรม

สถาปัตยกรรมชุดคำสั่งถูกวัดโดยปัจจัยต่าง ๆ ดังนี้

- พื้นที่ที่โปรแกรมต้องการ
- ความซับซ้อนของชุดคำสั่ง โดยเฉพาะการถอดรหัสที่จำเป็นที่จะประมวลคำสั่ง และความซับซ้อนของงานที่คำสั่งนั้นทำ
- ความยาวของคำสั่ง
- จำนวนคำสั่ง

สิ่งที่ใช้พิจารณาเมื่อออกแบบชุดคำสั่งมีดังนี้

- คำสั่งสั้นๆ เป็นตัวอย่างที่ดี เนื่องจากใช้พื้นที่หน่วยความจำน้อย สามารถ fetch ได้เร็ว แต่มักถูกจำกัดด้วยจำนวนคำสั่ง เพราะต้องมีจำนวนบิตที่เพียงพอกับจำนวนคำสั่งที่จำเป็นต้องใช้
- คำสั่งที่จำกัดความยาวจะถอดรหัสได้ง่าย แต่ละเสียพื้นที่ไปมาก
- การจัดสันหน่วยความจำมีผลต่อรูปแบบของคำสั่ง เช่นถ้ามีหน่วยความจำ 16 หรือ 32 บิต และไม่สามารถจัดแอดเดรสเป็นไบต์ได้ ก็จะยากต่อการเข้าถึงข้อมูลที่เป็นชนิดหนึ่งตัวอักษร จากกรณีนี้แม้ว่าเครื่องจะมี 16, 32, หรือ 64 บิต/เวิร์ด ก็ตาม จะต้องรองรับการจัดแอดเดรสเป็นไบต์ด้วย หมายความว่า แต่ละไบต์มีแอดเดรสที่ไม่ซ้ำกัน แม้ว่าเวิร์ดจะมีขนาดมากกว่าหนึ่งไบต์
- คำสั่งใดๆ ที่มีความยาวจำกัด ไม่ได้หมายความว่าจะต้องกำหนดจำนวน operand ตายตัว เราสามารถออกแบบ ISA ที่จำกัดความยาวของคำสั่ง แต่ยอมให้จำนวนบิตของ operand ผันแปรได้ตามความจำเป็น คำสั่งในลักษณะนี้เรียกว่า expanding opcode
- มีโหมดการกำหนดแอดเดรสที่แตกต่างกันได้หลายชนิด จากบทที่ 4 MARIE ใช้สองโหมดคือ Direct และ Indirect
- ถ้าเวิร์ดมีหลายไบต์ ลำดับของไบต์ควรนำมาจัดแอดเดรสอย่างไร
- สถาปัตยกรรมควรมีรีจิสเตอร์กี่ตัว และต้องจัดรีจิสเตอร์อย่างไร มีการจัดเก็บ operand อย่างไรใน cpu

5.2.2 Little Versus Big Endian

คำว่า “endian” ในทางสถาปัตยกรรมคอมพิวเตอร์หมายถึง “byte order” เป็นวิธีที่คอมพิวเตอร์ใช้เก็บไบต์สำหรับข้อมูลที่มีหลายไบต์ สถาปัตยกรรมคอมพิวเตอร์ที่ใช้ในปัจจุบันเป็นการกำหนดแอดเดรสแบบไบต์ และจะต้องมีมาตรฐานอย่างหนึ่งสำหรับการเก็บข้อมูลที่ต้องการใช้พื้นที่มากกว่าหนึ่งไบต์ บางเครื่องเก็บเป็นจำนวนเต็มสองไบต์ เริ่มจาก LSB(Least significant byte) ก่อน แล้วจึงตามด้วย MSB เพราะฉะนั้นไบต์ใดๆ ที่มีแอดเดรสต่ำกว่าก็จะมีนัยสำคัญที่ต่ำกว่าด้วย เครื่องที่มีการจัดแอดเดรสแบบนี้เรียกว่า “little endian” ส่วนเครื่องที่จัดเก็บจำนวนเต็มสองไบต์เหมือนกัน แต่ใส่ลงใน MSB ก่อนแล้วตามด้วย LSB ก็จะถูกเรียกว่า “Big endian” เนื่องจากเครื่องจำพวกนี้จะเก็บ MSB ไว้ที่แอดเดรสที่ต่ำกว่า เครื่องที่ใช้ UNIX ส่วนใหญ่ก็จะเป็น big endian ในขณะที่เครื่อง PC จะเป็น little endian และสถาปัตยกรรม RISC รุ่นใหม่ส่วนใหญ่ก็ยังใช้ big endian

ตัวอย่าง พิจารณาการเก็บจำนวนเต็มสี่ไบต์

Byte 3	Byte 2	Byte 1	Byte 0

เครื่องที่ใช้ little endian หน่วยความจำจะมีการจัดเรียงดังนี้

```
Base Address + 0 = Byte0
Base Address + 1 = Byte1
Base Address + 2 = Byte2
Base Address + 3 = Byte3
```

เครื่องที่ใช้ big endian หน่วยความจำจะมีการจัดเรียงดังนี้

```
Base Address + 0 = Byte3
Base Address + 1 = Byte2
Base Address + 2 = Byte1
Base Address + 3 = Byte0
```

ในกรณีที่เครื่องมีการจัดหน่วยความจำแบบไบต์ ค่าตัวเลขฐานสิบหก 12345678 ถูกเก็บไว้ที่แอดเดรส 0 ตัวเลขแต่ละหลักต้องการสี่บิต ดังนั้นหนึ่งไบต์เก็บได้สองตัว ค่าฐานสิบหกนี้เมื่อนำไปเก็บในหน่วยความจำที่แสดงในรูปที่ 5.1 ที่มีการจัดหน่วยความจำทั้ง little และ big endian

Address ⟶	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

รูปที่ 5.1 ค่าตัวเลขฐานสิบหก 12345678 ที่จัดเก็บตามรูปแบบ big และ little endian

ตัวอย่างที่ 5.1 เมื่อคอมพิวเตอร์ใช้ 32 บิต สำหรับเก็บจำนวนเต็ม ให้แสดงว่าแต่ละค่าต่อไปนี้มีการจัดเก็บกันตามลำดับอย่างไร โดยมีแอดเดรสเริ่มต้นอยู่ที่ 0x200 สมมุติว่าแต่ละแอดเดรสมีขนาดหนึ่งไบต์ และค่าตัวเลขที่ต้องการเก็บคือ 0xABCD1234, 0x00FE4321, และ 0x10

Address	Byte Order Big Endian	Little Endian
0x200	AB	34
0x201	CD	12
0x202	12	CD
0x203	34	AB
0x204	00	21
0x205	FE	43
0x206	43	FE
0x207	21	00
0x208	00	10
0x209	00	00
0x20A	00	00
0x20B	10	00

สังเกตว่าเมื่อใช้ big endian เมื่อเราอ่านจากแอดเดรสเริ่มต้อง ก็สามารถอ่านได้ตามปกติคือ 0xABCD1234 นั่นคือ AB เป็นไบต์ลำดับสูงสุด ตามด้วย CD 12 และ 34 อย่างไรก็ตามการแทนด้วย little endian จำนวนที่แอดเดรสเริ่มต้น 0x200 คือ LSB หรือ 34 และสังเกตว่าจะไม่เก็บ little endian เป็น 43

21 DC และ BA ที่เป็นเช่นนี้เนื่องจาก little และ big endian เป็นการจัดลำดับของไบต์ ไม่ใช่การจัดลำดับตัวเลข ดังที่สังเกตุได้ว่ามีการเติมเต็มในค่าสุดท้ายด้วย 0 เพื่อให้ได้ตัวเลขใน 32 บิต นั่นคือ 0x00000010

จะเห็นว่าการแทนลำดับของบิต มีข้อดีข้อเสียของแต่ละวิธี และไม่มีวิธีใดปีนวิธีที่ดีที่สุด แม้ว่า big endian จะดูเป็นธรรมชาติ เมื่อเราอ่านค่าที่แทนโดยวิธีนี้ เมื่อเริ่มจาก MSB ทำให้ง่ายที่จะรู้ว่าค่านี้เป็นค่าบวกหรือลบโดยดูจากไบต์ที่เป็นแอดเดรสเริ่มต้น (เมื่อเทียบกับ little endian จะต้องรู้ก่อนว่าค่าที่เก็บยาวเท่าไร เพื่อใช้ในการค้นหาว่าไบต์ของเครื่องหมาย หรือ sign bit อยู่ที่ไหน) เครื่องที่แทนด้วย big endian จะเก็บจำนวนเต็มและสตริงในลำดับที่เหมือนกัน ทำให้ดำเนินการกับสตริงได้เร็วกว่า ภาพกราฟฟิกส์ที่เป็นบิตแม็ฟ (Bitmapped graphics) จะถูกแม็ปด้วย MSB ไว้ทางซ้ายสุด นั่นคือเมื่อจัดการกับข้อมูลแต่ละจุดข้อมูลที่มีขนาดใหญ่กว่าหนึ่งไบต์ ก็สามารถจัดการด้วยสถาปัตยกรรมของมันเอง นี้เป็นข้อจำกัดทางศักยภาพสำหรับเครื่องที่ใช้ little endian เนื่องจากข้อมูลจะต้องถูกแปลผกผัน(reverse) ลำดับของไบต์ เมื่อเก็บภาพกราฟฟิกส์ขนาดใหญ่ เมื่อถอดรหัสข้อมูลที่ถูกบีบอัด เช่นเมื่อเข้ารหัสด้วยวิธี Huffman และ LZW (ที่ได้กล่าวไว้ในบทที่ 7) รหัสของเวิร์ดหรือ codeword สามารถใช้เป็นดัชนีที่ใส่ไว้ในตารางการค้นหา ถ้ามันถูกเก็บแบบ big endian (นี่ก็จะทำได้ง่ายสำหรับการเข้ารหัส)

อย่างไรก็ตาม big Endian ก็มีข้อเสียเช่นกัน การแปลงจากแอดเดรสของจำนวนเต็ม 32 บิต ไปเป็นแอดเดรสจำนวนเต็ม 16 บิต จะต้องใช้เครื่อง big endian เพื่อทำการเติมเต็มไบต์ การคำนวณทางคณิตศาสตร์ที่มีความแม่นยำสูงในเครื่อง little endian นั้นจะเร็วและง่ายขึ้น สถาปัตยกรรมส่วนใหญ่ที่ใช้ big endian ไม่อนุญาตให้เขียนเวิร์ดลงในแอดเดรสที่ไม่ใช่เวิร์ด (เช่น ถ้าเวิร์ดมีขนาด 2 หรือ 4 ไบต์ ก็จะต้องเริ่มต้นด้วยไบต์แอดเดรสที่เป็นเลขคู่เสมอ) ทำให้เปลืองพื้นที่ สถาปัตยกรรม little endian เช่น Intel อนุญาตให้อ่านและเขียนแอดเดรสเป็นจำนวนคี่ได้ ทำให้การเขียนโปรแกรมบนเครื่องเหล่านี้ง่ายขึ้นมาก แต่หากโปรแกรมเมอร์เขียนคำสั่งให้อ่านค่าที่ไม่ใช่ศูนย์ของขนาดเวิร์ดที่ไม่ถูกต้องบนเครื่อง big endian โปรแกรมจะอ่านค่าที่ไม่ถูกต้องเสมอ แต่บนเครื่อง little endian บางครั้งก็อาจส่งผลให้ข้อมูลที่ถูกต้องถูกอ่าน (ในที่สุด Intel ได้เพิ่มคำสั่งเพื่อกลับลำดับไบต์ภายในรีจิสเตอร์)

ระบบเครือข่ายคอมพิวเตอร์ใช้ big endian หมายความว่าเมื่อคอมพิวเตอร์ little endian กำลังส่งจำนวนเต็มผ่านเครือข่าย (เช่นแอดเดรสอุปกรณ์เครือข่าย) เครื่องจำเป็นต้องแปลงเป็นลำดับไบต์ของเครือข่ายก่อน ในทำนองเดียวกันเมื่อเครื่องได้รับค่าจำนวนเต็มผ่านเครือข่าย ก็จำเป็นต้องแปลงกลับไปเป็น little endian ของเครื่องเอง

แม้ว่าอาจจะไม่คุ้นเคยกับการเปรียบเทียบ little กับ big endian แต่ก็มีแอพพลิเคชั่นหรือซอฟต์แวร์ในปัจจุบัน โปรแกรมใด ๆ ที่เขียนข้อมูลไปยังหรืออ่านข้อมูลจากไฟล์ จะต้องทราบถึงการการจัดลำดับไบต์บนเครื่อง เช่นรูปแบบกราฟิก Windows BMP ได้รับการพัฒนาบนเครื่อง little endian ดังนั้นหากต้องการดู BMP บนเครื่อง big endian แอปพลิเคชันที่ใช้ในการดูจะต้องกลับลำดับไบต์ก่อน นักออกแบบซอฟต์แวร์ที่นิยมใช้จะต้องตระหนักถึงปัญหาการจัดลำดับไบต์ เช่น Adobe Photoshop ใช้ big endian, GIF เป็น little endian, JPEG เป็น big endian, MacPaint เป็น big endian, PC Paintbrush เป็น little endian, RTF

โดย Microsoft เป็น little endian และไฟล์ Sun raster เป็น big endian บางแอปพลิเคชันรองรับทั้งสองรูปแบบ: ไฟล์ Microsoft WAV และ AVI, ไฟล์ TIF และ XWD (X Windows Dump) รองรับทั้งสองอย่างโดยทั่วไปแล้วจะเข้ารหัสตัวระบุลงในไฟล์

5.2.3 หน่วยความจำภายในในซีพียู: กองซ้อนกับรีจิสเตอร์

เมื่อกำหนดลำดับไบต์ในหน่วยความจำ ผู้ออกแบบฮาร์ดแวร์จะต้องตัดสินใจเลือกวิธีที่ CPU ใช้จัดเก็บข้อมูล วิธีพื้นฐานที่สุดที่ทำให้เห็นความแตกต่างของ ISAs มีสามตัวเลือก:

1. สถาปัตยกรรมสแต็ก
2. สถาปัตยกรรมแอคคูมูเลเตอร์
3. สถาปัตยกรรม general-purpose register (GPR)

สถาปัตยกรรมสแต็กจะใช้สแต็กเพื่อดำเนินการคำสั่ง และพบตัวถูกดำเนินการ (โดยปริยาย) ที่นำมาใส่ทางด้านบนของสแต็ก แม้ว่าเครื่องที่ใช้สแต็กจะต้องเขียนรหัสคำสั่งมาก และเป็นแบบจำลองอย่างง่ายสำหรับการประเมินนิพจน์ แต่สแต็กไม่สามารถเข้าถึงแบบสุ่มได้ ทำให้ยากต่อการสร้างรหัสที่มีประสิทธิภาพ นอกจากนี้สแต็กจะกลายเป็นคอขวดระหว่างการดำเนินการ

สถาปัตยกรรมแอคคูมูเลเตอร์เช่น MARIE มีหนึ่งตัวถูกดำเนินการโดยปริยาย สถาปัตยกรรมแอคคูมูเลเตอร์ลดความซับซ้อนภายในของเครื่องและจะมีคำสั่งสั้น ๆ แต่เนื่องจากแอคคูมูเลเตอร์เป็นเพียงที่เก็บข้อมูลชั่วคราวการรับส่งข้อมูลจากหน่วยความจำ ทำให้มีการถ่ายโอนข้อมูลสูงมาก

สถาปัตยกรรม GPR ใช้ชุดรีจิสเตอร์เอนกประสงค์ เป็นสถาปัตยกรรมที่ได้รับการยอมรับมากที่สุดสำหรับเครื่องในปัจจุบัน GPR ประกอบด้วยเซ็ตของรีจีสเตอร์ที่เร็วกว่าหน่วยความจำและง่ายสำหรับคอมไพเลอร์ที่จะจัดการ ทำให้สามารถใช้งานได้อย่างมีประสิทธิภาพ นอกจากนี้ราคาฮาร์ดแวร์ที่ลดลงอย่างมาก ทำให้สามารถเพิ่มการรีจีสเตอร์ได้เป็นจำนวนมากในราคาที่ต่ำที่สุด หากเครื่องสามารถการเข้าถึงหน่วยความจำได้เร็ว การออกแบบโดยใช้สแต็กอาจเป็นความคิดที่ดี แต่ถ้าหากหน่วยความจำช้า GPR จะดีกว่า เพราะทำให้การเขียนโปรแกรมที่สั้นลง นี่คือเหตุผลที่คอมพิวเตอร์ส่วนใหญ่ในช่วง 10 ปีที่ผ่านมามีการใช้ GPR อย่างไรก็ตามเนื่องจากตัวถูกดำเนินการทั้งหมดจะต้องตั้งชื่อ จึงทำให้ GPR มีคำสั่งที่ยาว และอาจทำให้ขั้นตอนของ Fetch และ Decode ใช้เวลานานขึ้น (เป้าหมายที่สำคัญอย่างหนึ่งสำหรับนักออกแบบ ISA คือต้องการให้คำสั่งสั้น) นักออกแบบที่เลือก ISA จะต้องตัดสินใจว่าสิ่งใดจะทำงานได้ดีที่สุดในสภาพแวดล้อมเฉพาะและตรวจสอบข้อได้เปรียบเสียเปรียบของแต่ละวิธีอย่างระมัดระวัง

สถาปัตยกรรม GPR สามารถแบ่งออกเป็นสามประเภท ขึ้นอยู่กับสถานที่ตั้งของตัวถูกดำเนินการ ประเภทแรกคือ สถาปัตยกรรม Memory-Memory อาจมีตัวถูกดำเนินการสองหรือสามตัวอยู่ในหน่วยความจำ ทำให้คำสั่งในการดำเนินการโดยไม่ต้องมีตัวถูกดำเนินการอยู่ในรีจีสเตอร์ สถาปัตยกรรม Registor-memory มีการประสมประสานกันอย่างน้อยหนึ่งตัวถูกดำเนินการอยู่ในรีจิสเตอร์ และอีกตัวหนึ่งอยู่ในหน่วยความจำ สถาปัตยกรรม Load-store ต้องการย้ายข้อมูลไปยังรีจิสเตอร์ก่อนดำเนินการใด ๆ กับข้อมูลเหล่านั้น Intel และ Motorola เป็นตัวอย่างของสถาปัตยกรรมหน่วยความจำรีจิสเตอร์ สถาปัตยกรรม Digital

Equipment’s VAX เป็นแบบ memory-memory และ SPARC, MIPS, Alpha และ PowerPC เป็นเครื่องที่ใช้ load-store

สถาปัตยกรรมในปัจจุบันส่วนใหญ่ใช้ GPR จึงตรวจสอบคุณสมบัติของชุดคำสั่งหลักสองชุดที่ใช้แบ่ง GPR คุณสมบัติทั้งสองนี้คือ จำนวนตัวถูกดำเนินการ และวิธีการที่ใช้กำหนดแอดเดรสให้กับตัวดำเนินการ ในหัวข้อ 5.2.4 จะศึกษาความยาวของคำสั่งและจำนวนตัวถูกดำเนินการที่คำสั่งสามารถมีได้ (สองหรือสามตัวถูกดำเนินการเป็นเรื่องธรรมดาที่สุดสำหรับสถาปัตยกรรม GPR และเราเปรียบเทียบสิ่งเหล่านี้กับสถาปัตยกรรมแบบศูนย์และหนึ่งตัวถูกดำเนินการ) จากนั้นจะตรวจสอบประเภทคำสั่ง สุดท้ายในส่วนที่ 5.4 เราจะตรวจสอบโหมดที่ใช้ในการกำหนดแอดเดรสต่างๆ

5.2.4 จำนวนตัวถูกดำเนินการและความยาวของคำสั่ง

วิธีการดั้งเดิมสำหรับการอธิบายสถาปัตยกรรมคอมพิวเตอร์คือ การกำหนดจำนวนตัวถูกดำเนินการ หรือแอดเดรสสูงสุดที่มีอยู่ในแต่ละคำสั่ง สิ่งนี้มีผลโดยตรงกับความยาวของคำสั่งเอง MARIE ใช้คำสั่งความยาวคงที่พร้อม opcode 4 บิตและตัวถูกดำเนินการ 12 บิต คำสั่งเกี่ยวกับสถาปัตยกรรมปัจจุบันสามารถจัดรูปแบบได้สองวิธี:

•ความยาวคงที่ - เสียพื้นที่ แต่เร็วและให้ประสิทธิภาพที่ดีขึ้น เมื่อใช้ pipelining ในระดับคำสั่งตามที่เราเห็นในส่วน 5.5

•ความยาวแปรผัน - ซับซ้อนกว่าในการถอดรหัส แต่ประหยัดพื้นที่จัดเก็บ

โดยทั่วไปมีการประนีประนอมกันเกี่ยวข้องกับการใช้ความยาวของคำสั่งสองถึงสามแบบ ซึ่งมีรูปแบบบิตที่สามารถแยกความแตกต่างและถอดรหัสได้ง่าย ความยาวคำสั่งจะต้องเปรียบเทียบกับความยาวของเวิร์ดในเครื่อง หากความยาวของคำสั่งเท่ากับความยาวของเวิร์ด แน่นอนคำสั่งจะจัดเรียงอย่างสมบูรณ์แบบ เมื่อเก็บไว้ในหน่วยความจำหลัก คำสั่งจะต้องจัดเรียงเวิร์ดตามแอดเดรส ดังนั้นคำสั่งที่มีขนาดครึ่ง หนึ่งในสี่ เป็นสองเท่าหรือสามเท่าของขนาดเวิร์ด อาจทำให้เปลืองเนื้อที่ คำสั่งความยาวแปรผันนั้นชัดเจนว่าขนาดไม่เท่ากัน และจำเป็นต้องจัดตำแหน่งเวิร์ดที่ทำให้สูญเสียพื้นที่เช่นกัน

รูปแบบคำสั่งที่พบบ่อยที่สุด ได้แก่ คำสั่งที่ไม่มีตัวถูกดำเนินการ มีหนึ่ง สอง หรือ สาม ตัว จากบทที่ 4 บางคำสั่งของ MARIE ไม่มีตัวถูกดำเนินการ ในขณะที่คำสั่งอื่นมีตัวถูกดำเนินการหนึ่งตัว การดำเนินการทางคณิตศาสตร์และตรรกะมักจะมีตัวถูกดำเนินการสองตัว แต่สามารถดำเนินการได้ด้วยตัวถูกดำเนินการหนึ่งตัว (ดังที่เราเห็นใน MARIE) เมื่อใช้ AC เราสามารถขยายแนวคิดนี้ไปยังตัวถูกดำเนินการสามตัว หากเราพิจารณาจุดหมายปลายทางสุดท้ายว่า เป็นตัวถูกดำเนินการตัวที่สาม นอกจากนี้ยังสามารถใช้สแต็กที่ไม่ต้องมีตัวถูกดำเนินการ ต่อไปนี้เป็นรูปแบบคำสั่งทั่วไป:

- OPCODE เท่านั้น (ไม่มีแอดเดรส)
- OPCODE + 1 แอดเดรส (โดยปกติจะเป็นแอดเดรสหน่วยความจำ)
- OPCODE + 2 แอดเดรส (ปกติรีจีสเตอร์สองตัว หรือรีจีสเตอร์หนึ่งและหน่วยความจำอีกหนึ่ง)
- OPCODE + 3 แอดเดรส (รีจีสเตอร์ หรือรีจีสเตอร์ร่วมกับหน่วยความจำ)

สถาปัตยกรรมมีการจำกัดจำนวนตัวถูกดำเนินการสูงสุดต่อคำสั่ง เช่นใน MARIE จำนวน operand สูงสุดคือหนึ่ง แม้ว่าบางคำสั่งจะไม่มีตัวถูกดำเนินการ (Halt และ Skipcond) โดยทั่วไปคำสั่งที่มี operand เป็นศูนย์, หนึ่ง, สองและสาม ที่พบเห็นมากที่สุด คำสั่งที่มี operand หนึ่ง - สอง – และสาม เข้าใจได้ง่าย ในตอนแรก ISA ที่สร้างขึ้นโดยใช้คำสั่งแบบ zero-operand อาจค่อนข้างสับสน

คำสั่งเครื่องที่ไม่มีตัวถูกดำเนินการต้องใช้สแต็ค เพื่อดำเนินการเหล่านั้น (โครงสร้างข้อมูลแบบเข้าก่อนออกก่อน (last-in, first-out) นำมาใช้ในบทที่ 4 และอธิบายรายละเอียดในภาคผนวก A ซึ่งการใส่และการลบจะทำจากด้านบน) โดยทั่วไปคำสั่งต้องมีหนึ่งหรือสองตัวถูกดำเนินการ (เช่น Add) แทนที่จะใช้รีจีสเตอร์วัตถุประสงค์ทั่วไป สถาปัตยกรรมแบบสแต็กจะจัดเก็บตัวถูกดำเนินการด้านบนของสแต็ก ทำให้องค์ประกอบด้านบนสามารถเข้าถึง CPU ได้ (โครงสร้างข้อมูลที่สำคัญที่สุดในสถาปัตยกรรมเครื่องคือสแต็ก ไม่เพียงแต่โครงสร้างนี้จะเป็นวิธีการที่มีประสิทธิภาพในการจัดเก็บค่าข้อมูลระดับกลาง ในระหว่างการคำนวณที่ซับซ้อนแต่ยังเป็นวิธีการที่มีประสิทธิภาพสำหรับการส่งพารามิเตอร์ เพื่อบันทึกโครงสร้างบล็อกโลคัลและกำหนดขอบเขตของตัวแปรและรูทีนย่อย)

ในสถาปัตยกรรมตามสแต็กคำสั่งส่วนใหญ่ประกอบด้วย opcodes เท่านั้น อย่างไรก็ตามมีคำสั่งพิเศษ (ที่เพิ่มองค์ประกอบไปยังและลบองค์ประกอบออกจากสแต็ก) ที่มีเพียงหนึ่งตัวถูกดำเนินการ สถาปัตยกรรมแบบสแต็กต้องการคำสั่ง push และคำสั่ง pop แต่ละคำสั่งจะมี operand ได้หนึ่งตัว Push X ใส่ข้อมูลที่ตำแหน่งหน่วยความจำ X ลงบนสแต็ก Pop X ลบองค์ประกอบด้านบนในสแต็กและเก็บไว้ที่ตำแหน่ง X เฉพาะคำสั่งบางอย่างเท่านั้น ที่ได้รับอนุญาตให้เข้าถึงหน่วยความจำ คำสั่งอื่น ๆ จะต้องใช้สแต็กสำหรับตัวถูกดำเนินการใด ๆ ที่จำเป็นในระหว่างการดำเนินการ

สำหรับการดำเนินการที่ต้องการสองตัวถูกดำเนินการจะใช้องค์ประกอบสองอันดับแรกของสแต็ก ตัวอย่างเช่นหากเราดำเนินการคำสั่งเพิ่ม CPU จะเพิ่มองค์ประกอบสองอันดับแรกของสแต็กให้เปิดทั้งสองอย่างจากนั้นจึงผลักผลรวมไปยังด้านบนสุดของสแต็ก สำหรับการดำเนินการที่ไม่ใช่การสื่อสารเช่นการลบองค์ประกอบสแต็คด้านบนจะถูกลบออกจากองค์ประกอบถัดไปไปด้านบนซึ่งทั้งคู่จะถูกตอกและผลลัพธ์จะถูกผลักลงบนสุดของสแต็ก

สแต็กมีประสิทธิภาพมากในการประเมินนิพจน์ทางคณิตศาสตร์ที่ยาว ที่เขียนอยู่ในรูป reverse Polish notation (RPN) ซึ่งเป็นสัญกรณ์คณิตศาสตร์ที่เป็นไปได้โดยนักตรรกวิทยา Jan Lukasiewicz นักคณิตศาสตร์ชาวโปแลนด์ผู้คิดค้นสัญกรณ์นี้ในปี 1924 เป็นที่รู้จักกันในนาม postfix สัญกรณ์เมื่อเทียบกับสัญกรณ์ infix ซึ่งเป็นตัวดำเนินการระหว่างตัวถูกดำเนินการ และสัญกรณ์ prefix (Polish) วางโอเปอเรเตอร์ก่อนตัวถูกดำเนินการ ตัวอย่างเช่น

X + Y is in infix notation

+ X Y is in prefix notation

X Y + is in postfix notation

เมื่อใช้เครื่องหมาย postfix (หรือ RPN) Operator อยู่หลัง Operand หากนิพจน์ใดมีมากกว่าหนึ่งการดำเนินการ Operator จะถูกกำหนดทันทีหลังตัวถูกดำเนินการที่สอง นิพจน์ infix "3 + 4" เทียบเท่ากับ postfix "3 4 +" ตัวดำเนินการ + ถูกนำไปใช้กับตัวถูกดำเนินการทั้งสอง 3 และ 4 หากนิพจน์นั้นซับซ้อนมากขึ้น ยังคงสามารถใช้แนวคิดนี้ในการแปลงจาก infix เป็น postfix เพียงแค่ต้องตรวจสอบนิพจน์และเรียงลำดับความสำคัญของ Operator

ตัวอย่าง 5.2 พิจารณานิพจน์ infix 12 / (4 + 2) เมื่อแปลงเป็น postfix ดังนี้:

นิพจน์	Explanation
12 / 4 2 +	ผลรวม 4 + 2 อยู่ในวงเล็บและมีความสำคัญกว่า แทนด้วย 4 2+
12 4 2 + /	ตัวถูกดำเนินการใหม่สองตัวคือ 12 และผลรวมของ 4 และ 2 เราวางตัวถูกดำเนินการแรกตามด้วยตัวที่สองตามด้วยตัวดำเนินการหาร

ดังนั้นนิพจน์ postfix 12 4 2 + / เทียบเท่ากับนิพจน์ infix 12 / (4 + 2) ให้สังเกตว่าไม่จำเป็นต้องเปลี่ยนลำดับของตัวถูกดำเนินการ และวงเล็บเพื่อให้ลำดับความสำคัญสำหรับตัวดำเนินการบวกจะถูกกำจัด

ตัวอย่าง 5.3 พิจารณานิพจน์ infix ต่อไปนี้ (2 + 3) - 6/3 เมื่อแปลงเป็น postfix ดังนี้:

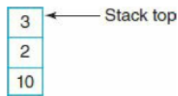
นิพจน์	Explanation
2 3 + – 6/3	ผลรวม 2 + 3 อยู่ในวงเล็บและมีความสำคัญกว่า แทนด้วย 2 3 +
2 3 + – 6 3 /	ตัวดำเนินการหารมีความสำคัญกว่า จึงแทนที่ 6/3 ด้วย 6 3 /
2 3 + 6 3 / –	เราต้องการลบผลหารของ 6/3 จากผลรวมของ 2 + 3 ดังนั้นเราจึงย้าย - ไปเป็นตัวสุดท้าย

ดังนั้นนิพจน์ postfix 2 3 + 6 3 / - เทียบเท่ากับนิพจน์ infix (2 + 3) - 6/3

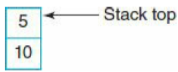
นิพจน์ทางคณิตศาสตร์ทั้งหมดสามารถเขียนได้โดยการแทนดังกล่าว อย่างไรก็ตามการแทน postfix รวมกับ stack ของ register เป็นวิธีที่มีประสิทธิภาพที่สุดในการประเมินนิพจน์ทางคณิตศาสตร์ อันที่จริงเครื่องคิดเลขอิเล็กทรอนิกส์บางตัว (เช่น Hewlett-Packard) ต้องการให้ผู้ใช้ป้อนนิพจน์ในเครื่องหมาย postfix ด้วยการฝึกฝนเล็กน้อยเกี่ยวกับเครื่องคิดเลขเหล่านี้ คุณสามารถประเมินนิพจน์ยาวที่มีวงเล็บซ้อนกันจำนวนมากได้อย่างรวดเร็วโดยไม่หยุดคิดเกี่ยวกับวิธีการจัดกลุ่ม

อัลกอริทึมในการประเมินการนิพจน์ RPN โดยใช้สแต็กค่อนข้างง่าย: จะเริ่มอ่านนิพจน์จากซ้ายไปขวาแต่ละตัวถูกดำเนินการ (ตัวแปรหรือค่าคงที่) ถูกใส่ลงบนสแต็ก เมื่อพบตัวดำเนินการแบบไบนารี ตัวถูกดำเนินการสองอันดับแรกในสแต็ก ก็จะมีการประมวลผลกับตัวถูกดำเนินการเหล่านั้น จากนั้นผลลัพธ์จะถูกใส่กลับลงไปยังสแต็ก

ตัวอย่าง 5.4 พิจารณานิพจน์ RPN 10 2 3 + / เมื่อใช้สแต็กเพื่อประเมินนิพจน์ เมื่ออ่านจากซ้ายไปขวาก่อนอื่นจะ push 10 ลงสแต็กตามด้วย 2 จากนั้น 3 ก็จะได้สแต็กดังนี้



เมื่อเจอตัวดำเนินการ “+” ก็จะ pop 3 และ 2 ออกจากสแต็ก แล้วทำการดำเนินการ (2 + 3) และใส่ผลบวก 5 ลงในสแต็ก ก็จะได้



เมื่อเจอตัวดำเนินการ “/” จะ pop 5 และ 10 ออกจากสแต็ก 10 ถูกหารด้วย 5 จากนั้นผลลัพธ์ 2 จะถูกใส่กลับลงบนสแต็ก (หมายเหตุ: สำหรับการดำเนินการที่ไม่ใช่มาตรการเช่นการลบและการหารองค์ประกอบสแต็กด้านบนจะเป็นตัวถูกดำเนินการที่สองเสมอ)

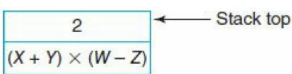
ตัวอย่าง 5.5 พิจารณานิพจน์ infix ต่อไปนี้:

(X + Y) × (W - Z) + 2

นิพจน์นี้เขียนในสัญกรณ์ RPN คือ:

X Y + W Z - × 2 +

การประเมินนิพจน์จะใช้สแต็ก push X และ Y และบวกสองค่านี้ (โดย pop จากสแต็ก) และเก็บผลรวม (X + Y) ลงสแต็ก จากนั้น push W และ Z ลบ (โดย pop จากสแต็ก) และเก็บผลต่าง (W - Z) ไว้ในสแต็ก ตัวดำเนินการ × คูณ (X + Y) ด้วย (W - Z) pop ค่าเหล่านี้ขึ้นมาจากสแต็ก แล้ว push ผลคูณลงในสแต็ก ต่อจากนั้น push 2 ลงบนสแต็ก ก็จะได้:



ตัวดำเนินการ + จะบวกค่าสองอันดับแรกของสแต็ก โดย Pop ออกมาจากสแต็ก และ push ผลบวกลงในสแต็ก ทำให้ (X + Y) × (W - Z) + 2 เก็บไว้ที่ด้านบนของสแต็ก

ตัวอย่าง 5.6 แปลงนิพจน์ RPN:

8 6 + 4 2 - /

ให้เป็น infix notation

ตัวดำเนินการแต่ละตัวตามตัวถูกดำเนินการ ดังนั้นตัวดำเนินการ “+” มีตัวถูกดำเนินการคือ 8 กับ 6 ส่วนตัวดำเนินการ “-” มีตัวถูกดำเนินการ 4 กับ 2 ตัวดำเนินการ “/” ได้จากผลบวกของ 8 กับ 6 เป็นตัวถูกดำเนินการแรกและผลต่างของ 4 กับ 2 เราจะต้องใช้วงเล็บเพื่อแทนลำดับการดำเนินการ (เพื่อให้แน่ใจว่าการบวกและการลบจะดำเนินการก่อนการหาร) ก็จะได้นิพจน์ infix คือ

(8 + 6) / (4 - 2)

เพื่อแสดงแนวคิดของตัวถูกดำเนินการศูนย์ หนึ่ง สอง และ สาม ให้เขียนโปรแกรมง่าย ๆ เพื่อประเมินนิพจน์ทางคณิตศาสตร์โดยใช้รูปแบบเหล่านี้

ตัวอย่าง 5.7 สมมติว่าเราต้องการประเมินนิพจน์ต่อไปนี้:

Z = (X × Y) + (W × U)

โดยทั่วไปเมื่อมี operand สามตัว หนึ่งในนั้นจะเป็นรีจีสเตอร์ และตัวถูกดำเนินการตัวแรกคือไว้เก็บผลลัพธ์ ใช้คำสั่งสามแอดเดรส รหัสเพื่อประเมินการนิพจน์ Z เขียนได้ดังนี้:

```
Mult    R1, X, Y
Mult    R2, W, U
Add     Z,  R2, R1
```

เมื่อใช้คำสั่งสองแอดเดรส ปกติแล้วมีหนึ่งแอดเดรสเป็นรีจีสเตอร์ (คำสั่งสองแอดเดรสไม่ค่อยอนุญาตให้ตัวถูกดำเนินการทั้งสองเป็นที่อยู่หน่วยความจำ) ตัวถูกดำเนินการอื่นอาจเป็นรีจิสเตอร์หรือแอดเดรสในหน่วยความจำ ใช้คำสั่งสองแอดเดรส จะมีรหัสดังต่อไปนี้:

```
Load    R1, X
Mult    R1, Y
Load    R2, W
Mult    R2, U
Add     R1, R2
Store   Z, R1
```

สิ่งสำคัญคือต้องรู้ว่าตัวถูกดำเนินการแรกนั้นคือต้นทางหรือปลายทาง จากคำสั่งข้างต้นตัว operand และเป็นปลายทาง (สิ่งนี้มีแนวโน้มที่จะเป็นความสับสนสำหรับโปรแกรมเมอร์ผู้ที่ต้องสลับระหว่างภาษาแอสเซมบลีของ Intel และภาษาแอสเซมบลีของโมโตโรล่า - แอสเซมบลีของ Intel ระบุตัวถูกดำเนินการแรกเป็นปลายทาง

การใช้คำสั่งหนึ่งแอดเดรสอย่าง MARIE ต้องสมมุติว่ารีจีสเตอร์ (AC) ถูกใช้เป็นปลายทางสำหรับเก็บผลลัพธ์ของคำสั่ง ดังนั้นการประเมินนิพจน์ Z สามารถเขียนได้ดังนี้

```
Load    X
Mult    Y
Store   Temp
Load    W
Mult    U
Add     Temp
Store   Z
```

จำไว้ว่าเมื่อจำนวนตัวถูกดำเนินการที่ต่อคำสั่ง จำนวนคำสั่งที่จำเป็นในการเรียกใช้งานที่ต้องเขียนจะเพิ่มขึ้นนี่เป็นตัวอย่างที่ต้องแลกกันระหว่างพื้นที่กับเวลา โดยทั่วไปการออกแบบสถาปัตยกรรม คำสั่งที่สั้นลงเราต้องเขียนโปรแกรมที่ยาวขึ้น

โปรแกรมนี้มีลักษณะอย่างไรเมื่อนำไปเขียนบนเครื่องที่ใช้สแต็ก ที่มีคำสั่งที่ไม่มีแอดเดรส สถาปัตยกรรมสแต็กไม่ใช้ตัวถูกดำเนินการสำหรับคำสั่งเช่น Add, Subt, Mult หรือหาร แต่ใช้เพียงสองฟังก์ชันของสแต็กคือ Pop กับ Push การดำเนินการที่สื่อสารกับสแต็กจะต้องมีฟิลด์แอดเดรส เพื่อใช้กำหนดแอดเดรสของตัวถูกดำเนินการที่จะ push หรือ pop ลงบนสแต็ก (การดำเนินการอื่น ๆ จะเป็น zero address) ที่ ๆ จะ push ตัวถูกดำเนินการด้านบนของสแต็ก และ pop สแต็คด้านบนออกมาใส่ลงในตัวถูกดำเนินการ สถาปัตยกรรมนี้ทำให้ต้องเขียนโปรแกรมที่ยาวที่สุดในการประเมินสมการจากตัวอย่างนี้ สมมติว่าการดำเนินการทาง

คณิตศาสตร์ ใช้ตัวถูกดำเนินการสองตัวบนสแต็กด้านบน โดย push และ pop ผลลัพธ์ของการดำเนินการกับตัวอย่างมีดังนี้:

```
Push    X
Push    Y
Mult
Push    W
Push    U
Mult
Add
Pop     Z
```

ความยาวของคำสั่งได้รับผลกระทบอย่างแน่นอนจากความยาวของรหัสและจำนวนตัวถูกดำเนินการที่ได้รับอนุญาตของคำสั่ง หากความยาว opcode คงที่การถอดรหัสจะง่ายกว่ามาก อย่างไรก็ตามเพื่อให้เข้ากันได้และมีความยืดหยุ่น opcodes สามารถมีความยาวผันแปร opcodes ความยาวแปรผันนำเสนอปัญหาเช่นเดียวกับคำสั่งตัวแปรเทียบกับความยาวคงที่ ความยืดหยุ่นนี้ก็นำไปใช้ในการออกแบบ expanding opcodes

5.2.5 การขยาย Opcodes

จะเห็นว่าจำนวนตัวถูกดำเนินการของคำสั่งเป็นอย่างไร ขึ้นอยู่กับความยาวของคำสั่ง ซึ่งจะต้องมีบิตจำนวนที่เพียงพอสำหรับ opcode และสำหรับแอดเดรสของตัวถูกดำเนินการ อย่างไรก็ตามไม่ใช่คำสั่งทั้งหมดจำเป็นต้องมีจำนวนตัวถูกดำเนินการเท่ากัน

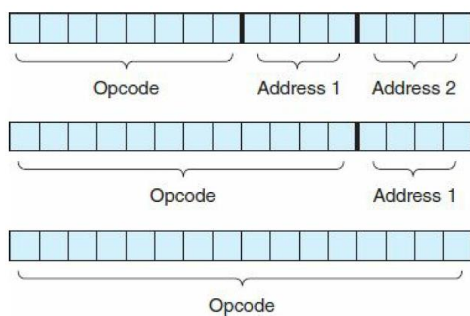
การขยาย opcodes เป็นการประนีประนอมระหว่างความต้องการที่จะมีชุดของรหัสคำสั่งจำนวนมากและเป็นรหัสที่สั้น ซึ่งทำให้คำสั่งนั้นสั้นด้วย แนวความคิดที่จะทำให้มี opcode บางส่วนสั้น และมี opcode อีกส่วนที่ยาวขึ้นตามความจำเป็น เมื่อ opcode สั้น ก็จะมีบิตเหลืออีกมากไว้เก็บ operand (ทำให้เราสามารถกำหนด operand ได้สองหรือสาม operands ต่อคำสั่ง) แต่เมื่อไม่จำเป็นต้องใช้พื้นที่ใดๆ สำหรับ operands (อย่างคำสั่ง Halt หรือเนื่องจากเครื่องใช้ stack) บิตทั้งหมดก็สามารถให้สำหรับ opcode ซึ่งทำให้สร้างคำสั่งได้มากขึ้น จะเห็นว่า expanding opcode ทำให้มีทางเลือกมากขึ้นระหว่าง opcode ที่ยาวขึ้นแต่มี operands น้อยลง และในทางกลับกันเมื่อต้องการ opcodes ที่สั้นแต่มี operands ต่อคำสั่งได้มากขึ้น

รูปที่ 5.2 รูปแบบที่เป็นไปได้สองประการสำหรับสร้างรูปแบบคำสั่ง 16 บิต

พิจารณาเครื่องที่มีคำสั่ง 16 บิตและมี 16 รีจีสเตอร์ เนื่องจากตอนนี้มีเซตของรีจีสเตอร์ แทนที่จะมีเพียง AC ตัวเดียวเหมือนใน MARIE ทำให้ต้องใช้ 4 บิตสำหรับอ้างถึงแต่ละรีจีสเตอร์ โดยใช้เข้ารหัสในสิบหกคำสั่ง

แต่ละคำสั่งใช้สามโอเพอแรนด์ที่ต้องโหลดเข้าไปที่สามรีจีสเตอร์ หรือจะใช้สี่บิตสำหรับ opcode และ 12 บิต สำหรับแอดเดรสของหน่วยความจำ (อย่างที่เคยเห็นใน MARIE ที่ทำให้มีหน่วยจำทั้งหมด 4K) อย่างไรก็ตามถ้าข้อมูลในหน่วยความจำถูกโหลดเข้ามาก่อนเพื่อเก็บไว้ในรีจีสเตอร์ใดๆ ในเซตนี้ คำสั่งสามารถเลือกข้อมูลได้เพียงสี่บิต(เมื่อมีสิบหกรีจีสเตอร์) สองทางเลือกนี้ได้แสดงไว้ในรูปที่ 5.2

แต่ทำไม ถึงจำกัด opcode ไว้ 4 บิตเท่านั้น? หากอนุญาตให้ความยาวของ opcode เปลี่ยนแปลงจำนวนนั้นจะเปลี่ยนจำนวนบิตที่เหลือ ซึ่งสามารถใช้สำหรับกำหนดแอดเดรสของตัวถูกดำเนินการ เมื่อใช้การขยาย opcodes เราสามารถทำให้ opcodes เป็น 8 บิต ตามความต้องการที่ตัวถูกดำเนินการรีจีสเตอร์สองตัว หรือเราอาจอนุญาตให้ opcodes เป็น 12 บิต ที่ทำงานกับหนึ่งรีจีสเตอร์ หรืออาจอนุญาตให้ใช้ opcodes แบบ 16 บิตเป็นคำสั่งที่ไม่ต้องการตัวถูกดำเนินการ รูปแบบเหล่านี้แสดงในรูปที่ 5.3



รูปที่ 5.3 ความเป็นไปได้อีกสามอย่างสำหรับรูปแบบออกแบบคำสั่ง 16 บิต

ปัญหาเดียวคือเราต้องการวิธีการตรวจสอบ ว่าคำสั่งควรนั้นมี opcode 4 บิต 8 บิต 12 บิตหรือ 16 บิต เคล็ดลับคือการใช้ “escape opcode” เพื่อระบุว่าควรใช้รูปแบบใด ความคิดนี้เป็นตัวอย่างที่ดีที่สุด

ตัวอย่าง 5.8 สมมติว่าต้องการเข้ารหัสคำสั่งต่อไปนี้:

- 15 คำสั่งสำหรับสามแอดเดรส
- 14 คำสั่งสำหรับสองแอดเดรส
- 31 คำสั่งสำหรับแอดเดรสเดียว
- 16 คำสั่งที่มีไม่มีแอดเดรส

เราสามารถเข้ารหัสชุดคำสั่งนี้ด้วย 16 บิตได้ไหม? คำตอบคือใช่ตราบใดที่เราใช้การขยายรหัส การเข้ารหัสมีดังนี้:

```
0000 R1   R2   R3
...                    15 three-address codes
1110 R1   R2   R3
1111 - escape opcode
1111 0000 R1   R2
...                    14 two-address codes
1111 1101 R1   R2
1111 1110 - escape opcode
1111 1110 0000 R1
...                    31 one-address codes
1111 1111 1110 R1
1111 1111 1111 - escape opcode
1111 1111 1111 0000
...                    16 zero-address codes
1111 1111 1111 1111
```

เราสามารถเห็นการใช้งาน opcode ในกลุ่มแรกของคำสั่งสามแอดเดรส เมื่อ 4 บิตแรกเป็น 1111 นั่นแสดงว่าคำสั่งนั้นไม่มีตัวถูกดำเนินการสามตัว แต่อาจจะมีสองตัวหนึ่งหรือไม่มีเลย (อันนี้ขึ้นอยู่กับกลุ่ม 4 บิตต่อไป) สำหรับกลุ่มที่สองของคำสั่งสองที่อยู่ Escape opcode คือ 11111110 (คำสั่งใด ๆ ที่มี opcode นี้หรือสูงกว่าจะไม่มีตัวถูกดำเนินการมากกว่าหนึ่งตัว) สำหรับกลุ่มที่สามของคำสั่ง one address escape opcode คือ 11111111111111 (คำสั่งที่มีลำดับ 12 บิตนี้จะไม่มีตัวถูกดำเนินการ)

แม้ว่าการอนุญาตสำหรับคำสั่งที่หลากหลายที่กว้างขึ้นชุดรูปแบบ opcode ที่ขยายนี้ ยังทำให้การถอดรหัสมีความซับซ้อนมากขึ้น แทนที่จะมองเพียงแค่รูปแบบของบิตง่ายแล้วตัดสินใจว่าเป็นคำสั่งอะไร เราต้องถอดรหัสคำสั่งดังนี้:

```
if (leftmost four bits != 1111 ) {
  Execute appropriate three-address instruction}
else if (leftmost seven bits != 1111 111 ) {
  Execute appropriate two-address instruction}
else if (leftmost twelve bits != 1111 1111 1111 ) {
  Execute appropriate one-address instruction }
else {
  Execute appropriate zero-address instruction
}
```