

สถาปัตยกรรมคอมพิวเตอร์

Computer Architecture

การออกแบบและการวิเคราะห์
Design and Analysis

ดร. เกริก พิรอมย์สopa
Krerk Piromsopa, Ph. D.

ฉบับปรับปรุง พ.ศ. 2559

ລຳຫວ່ັບ ດູນຜກາທິພຍໍ ເຊີ້ມຫາຍດືດີ ແລະ ເຊີ້ມຄູງລາດ
ຂອບຄຸນລຳຫວ່ັບຄວາມໝາຍາອ່ງຊື່ວິຕ

คำนำ

สถาปัตยกรรมคอมพิวเตอร์ (ภาษาอังกฤษใช้คำว่า computer architecture) เป็นพื้นฐานสำคัญที่นักคอมพิวเตอร์ทุกคนต้องรู้ บอยครั้งมักจะมีผู้ติดแย้งว่า แม้จะไม่มีความเข้าใจอันดีในสถาปัตยกรรมคอมพิวเตอร์ เรายังคงเป็นนักคอมพิวเตอร์ได้ คำกล่าวนี้คงจะไม่จริงนัก เพราะทุกอย่างที่เป็นมาตรฐานของคอมพิวเตอร์ ตั้งแต่คำสั่งที่ใช้ได้ การจัดเก็บข้อมูล การประมวลผล คือสิ่งที่กำหนดโดยสถาปัตยกรรมคอมพิวเตอร์ทั้งหมด ในฐานะวิศวกร คอมพิวเตอร์ และ นักวิทยาศาสตร์คอมพิวเตอร์ นั้น การออกแบบ และ การพัฒนา ระบบสถาปัตยกรรมคอมพิวเตอร์เป็นหน้าที่หลักโดยตรง ในฐานะนักพัฒนาซอฟต์แวร์ แม้จะมีได้ออกแบบหรือพัฒนาสถาปัตยกรรมคอมพิวเตอร์โดยตรง แต่ความเข้าใจอันดีในสถาปัตยกรรมคอมพิวเตอร์ ย่อมช่วยให้สามารถพัฒนาซอฟต์แวร์ได้ดีขึ้น

เพื่อให้เห็นภาพชัดเจนยิ่งขึ้น จึงขอยกตัวอย่างง่ายๆ ก่อน ในการแก้ปัญหา เช่น ให้เราต้องการหาคำตอบของสมการ $x^2 + 2x - 3 = 0$ ให้ได้ ทางวิธีที่ง่ายที่สุดคือการทดลองและปรับเปลี่ยนค่า x จนกว่าผลลัพธ์จะเท่ากับ 0 แต่ในกรณีที่สมการนี้มีสองคำตอบ คือ $x = 1$ และ $x = -3$ ทำให้เราต้องลองทุกค่าที่อยู่ระหว่าง -3 และ 1 จนกว่าจะ找到ค่าที่เป็น 0 นี่คือการ “暴力解题” หรือ “试错法”。ในทางคณิตศาสตร์เรียกว่า “数值解法”。ในทางคอมพิวเตอร์ นี่คือการ “暴力搜索” หรือ “Brute-force search”。ในทางภาษาโปรแกรม เช่น Python นี้ เราสามารถเขียนโค้ดดังนี้:

```
def solve_quadratic(a, b, c):
    x1 = (-b + (b**2 - 4*a*c)**0.5) / (2*a)
    x2 = (-b - (b**2 - 4*a*c)**0.5) / (2*a)
    return x1, x2
```

เมื่อเรา运行 โค้ดนี้ ผลลัพธ์ที่ได้คือ $(1, -3)$ ซึ่งเป็นสองคำตอบของสมการ $x^2 + 2x - 3 = 0$ ที่เราต้องการ。

สถาปัตยกรรมคอมพิวเตอร์ที่ดีต้องมีความสามารถในการคำนวณและจัดการข้อมูลอย่างรวดเร็วและมีประสิทธิภาพ แต่ในอดีต สถาปัตยกรรมคอมพิวเตอร์มีความจำกัดอย่างมาก เช่น ความจำที่จำกัด ความเร็วในการประมวลผลที่ช้า และการใช้พลังงานที่สูง แต่ในปัจจุบัน สถาปัตยกรรมคอมพิวเตอร์มีความก้าวหน้าอย่างมาก เช่น การประมวลผลที่รวดเร็ว การจัดการพลังงานที่มีประสิทธิภาพ และ การขยายความสามารถทางด้านการคำนวณที่สามารถประมวลผลข้อมูลในปริมาณที่ใหญ่ขึ้นได้

คอมพิวเตอร์อาจไม่ลึกซึ้งมากพอ จึงไม่สามารถที่จะดึงความสามารถบางอย่างของหน่วยประมวลผลออกมายใช้ได้ ทำให้ได้สมรรถนะที่ได้แต่ก่อว่าของผู้ผลิต แน่นอนว่าผู้ผลิตมีประสบการณ์ที่คล้ายกันนี้ในหลายงานที่ผู้ผลิตเคยมีส่วนเกี่ยวข้อง สิ่งที่ยกหัวมานี้เป็นเพียงกรณีศึกษาอันหนึ่งเท่านั้น (โดยมีนิสิตมาถามผมว่าอัลกอริทึมที่เค้าใช้ก็เหมือนกับของผู้ผลิต แต่ทำไม่ของเค้าถึงข้ากว่าของผู้ผลิต ผู้ผลิตยืนในใจแล้วตอบนิสิตว่า อิกคริงหนึ่งอยู่ที่โครงสร้างข้อมูลที่เลือกใช้ และการเลือกใช้คำสั่งที่เหมาะสมด้วยครับ)

ดังนั้นผมจึงอยากรู้ว่า นักศึกษา และ นักเรียนทั้งหลาย ว่า ความเข้าใจอันดีในสถาปัตยกรรมคอมพิวเตอร์ นอกจากจะเป็นพื้นฐานสำคัญสำหรับการศึกษาด้านคอมพิวเตอร์ ยังเป็นสิ่งจำเป็นสำหรับการทำงานในสาขาวิชาชีวภาพด้านคอมพิวเตอร์ทุกสาขา ผมหวังว่า การได้อ่านหนังสือที่ดี การได้ทดลอง(ออกแบบ) รวมถึง การวิเคราะห์ที่ดี จะช่วยให้สามารถศึกษาได้อย่างถ่องแท้มากขึ้น

หนังสือเล่มนี้พยายามวางแผนพื้นฐานที่สำคัญ เริ่มจากความเข้าใจในสถาปัตยกรรมคอมพิวเตอร์ ไปถึงการวิเคราะห์สถาปัตยกรรมคอมพิวเตอร์ที่หลากหลายแบบต่าง ๆ อย่างไรก็ตาม ศาสตร์ด้านนี้มีการเปลี่ยนแปลงอยู่ตลอดเวลา และ สถาปัตยกรรมที่ถูกมองว่าดีในเวลาหนึ่ง อาจจะถูกหักล้างในเวลาถัดมาด้วยเทคโนโลยีและความต้องการที่เปลี่ยนไป แผนจึงพยายามจัดวางเนื้อหา ให้เหมาะสมกับการสอนในหนึ่งภาคการศึกษา ซึ่งสอดคล้องกับหลักวิชา เช่น Computer System Architecture ซึ่งเป็นวิชาบังคับในหลักสูตรวิศวกรรมศาสตร์บัณฑิต สาขาวิศวกรรมคอมพิวเตอร์ ของ ภาควิชาวิศวกรรมคอมพิวเตอร์ จุฬาลงกรณ์มหาวิทยาลัย (ในบางหลักสูตรอาจจะใช้ชื่อวิชาว่า Computer Organization and Architecture) ซึ่งอาจจะไม่ครอบคลุมถึงสถาปัตยกรรมทุกรูปแบบ รวมถึงเนื้อหาในบางด้าน อาจจะไม่ลงรายละเอียดในเชิงลึกมากนัก แต่จะเน้นในเชิงวิเคราะห์และการนำไปใช้เบื้องหลัก ในบางช่วงบางตอนที่ต้องการให้เห็นโครงสร้าง ก็จะทำการอธิบายด้วยแผนภาพ (ในการปรับปรุงหนังสือเล่มนี้ครั้งถัดไป ผู้เขียนวางแผนจะใช้ภาษา Verilog HDL และใช้การ Simulation เพื่อประกอบความเข้าใจ)

ทั้งนี้ขออภัยตัวว่า ผมมิได้มองว่าตัวเองเป็นผู้เชี่ยวชาญด้านสถาปัตยกรรมคอมพิวเตอร์แต่อย่างใด (แน่นอนว่าคนที่ทำงานในสาขาวิชาพินิ หากบอกว่าตัวเองเชี่ยวชาญในด้านใด คงจะเป็นความคิดที่ไม่ค่อยดีนัก เพราะศาสตร์ด้านนี้เปลี่ยนแปลงรวดเร็วทุกวันจนสิ่งที่เราเรียนรู้พร้อมจะตกรุ่นได้ทุกเวลา) แม้ผมจะมีงานวิจัยและได้เคยยื่นขอจดสิทธิบัตร (pending) ก็เท่ากับสถาปัตยกรรมคอมพิวเตอร์ที่มีความสามารถด้านความมั่นคงปลอดภัยอยู่บ้างก็ตาม หากแต่ผมเพียงอยากรู้ว่า ผู้อ่านนักคอมพิวเตอร์รุ่นใหม่ (โดยเฉพาะกลุ่มที่เป็นคนไทย) เข้าใจสถาปัตยกรรมคอมพิวเตอร์และนำไปประยุกต์ใช้ประโยชน์ เพื่อให้สามารถพัฒนาโปรแกรมที่มีสมรรถนะมากขึ้น หรืออย่างน้อย ก็สามารถที่จะเลือกใช้สถาปัตยกรรมคอมพิวเตอร์ที่เหมาะสมกับงานที่ต้องการใช้ได้

จากความชอบส่วนตัวของผม และ ประสบการณ์ที่เกี่ยวข้องกับการเรียนการสอนด้าน ยาร์ดแวร์และสถาปัตยกรรมคอมพิวเตอร์กว่าสิบปี (ผมเริ่มสอนวิชาสถาปัตยกรรม คอมพิวเตอร์ครั้งแรกในฐานะอาจารย์พิเศษที่มหาวิทยาลัยเอกชน ตั้งแต่ยังเรียนปริญญาโท) ผมจึงอยากร่วมมือด้วยทอดประสบการณ์ และ ความรู้ ที่ส่วนหนึ่งได้จากการศึกษาและการลงทุน ลงผิด ให้กับ นิสิต นักศึกษาและคนรุ่นต่อไปได้ทราบ โดยหวังว่าจะเป็นการช่วยให้ นิสิต นักศึกษา สามารถเรียนลัดได้เร็วขึ้นแล้ว ยังจะเป็นการสืบท่องค์ความรู้ด้าน สถาปัตยกรรมคอมพิวเตอร์ (ซึ่งปัจจุบัน นักพัฒนาซอฟต์แวร์รุ่นใหม่มักไม่ค่อยให้ความสนใจ หรือไม่เห็นความสำคัญ) ให้เป็นพื้นฐานสำคัญ และยังคงอยู่ต่อไป

สุดท้ายนี้ผู้เขียนขอขอบคุณท่าน รองศาสตราจารย์ ดร. สมชาย ประสิทธิ์จูตระกูล ที่ได้ให้ คำแนะนำด้านการเรียนและการจัดพิมพ์ หนังสือเล่มนี้ประสบความสำเร็จได้ส่วนหนึ่ง เพราะ คำแนะนำของท่านครับ

ดร. เกริก ภิรมย์โสภา

ผู้ช่วยศาสตราจารย์
ภาควิชาวิศวกรรมคอมพิวเตอร์
จุฬาลงกรณ์มหาวิทยาลัย

Krerk.P@chula.ac.th

1 ธันวาคม 2559

สารบัญ

1	บทนำ.....	1
1.1	รู้ก่อนเรียน.....	1
1.2	ส่วนประกอบของระบบคอมพิวเตอร์.....	2
1.3	คำจำกัดความ.....	3
1.4	สถาปัตยกรรมชุดคำสั่ง (Instruction Set Architecture).....	4
1.5	โครงสร้างคอมพิวเตอร์ (Computer Organization).....	6
1.6	การสร้างระบบคอมพิวเตอร์ (Implementation).....	7
1.7	เป้าหมายของการออกแบบ.....	8
1.8	แบบฝึกหัดท้ายบท.....	9
2	สมรรถนะของระบบคอมพิวเตอร์.....	11
2.1	การวัดสมรรถนะของระบบคอมพิวเตอร์.....	11
2.2	เกาตอบสนอง และ ปริมาณงานที่ทำได้ต่อหน่วยเวลา.....	13
2.3	เกณฑ์เปรียบเทียบสมรรถนะ (Benchmark).....	15
2.3.1	SPEC Benchmark Suites.....	16
2.3.2	เกณฑ์เปรียบเทียบสมรรถนะที่ดี.....	17
2.4	การเปรียบเทียบสมรรถนะ.....	18
2.5	CPU Time.....	21
2.6	กฎของ Amdahl.....	27
2.7	แนวทางการปรับปรุงสมรรถนะของระบบคอมพิวเตอร์.....	30
2.8	สรุป.....	31
2.9	แบบฝึกหัดท้ายบท.....	33
3	สถาปัตยกรรมชุดคำสั่ง.....	35
3.1	สถาปัตยกรรมชุดคำสั่งในอดีต.....	37
3.1.1	สถาปัตยกรรมแบบ Accumulator.....	39
3.1.2	สถาปัตยกรรมแบบ Stack.....	40
3.1.3	สถาปัตยกรรมแบบ Memory-Memory.....	42
3.1.4	สถาปัตยกรรมแบบ Register-Memory.....	43
3.1.5	สถาปัตยกรรมแบบ Register-Register.....	45
3.2	หมวดคำสั่งและชุดคำสั่ง.....	46
3.3	การจัดการหน่วยความจำ (Memory Organization).....	49
3.3.1	การจัดลำดับข้อมูล (Endian).....	50
3.3.2	การทำหนندเลขที่อยู่รีเมตั้นของข้อมูล (Memory Alignment).....	51
3.4	การอ้างอิงเลขที่อยู่หน่วยความจำ (Addressing Mode).....	54
3.5	รูปแบบคำสั่ง (Instruction Format).....	56
3.6	สถาปัตยกรรมชุดคำสั่งที่ดี.....	58
3.6.1	บทบาทของคอมไฟเลอร์ในการหาค่าเหมาะสมที่สุด.....	59

3.7 การเรียกใช้โปรแกรมย่อย และ Exception.....	62
3.7.1 การส่งผ่านและคืนค่าระหว่างโปรแกรมย่อย.....	62
3.7.2 การบันทึกค่าตัวแปรท้องถิ่นและ rejister.....	63
3.7.3 Exception และ Interrupt.....	64
3.8 สรุป.....	65
3.9 แบบฝึกหัดท้ายบท.....	67
4 การออกแบบ หน่วยประมวลผลกลาง แบบ single cycle.....	69
4.1 ขั้นตอนในการออกแบบหน่วยประมวลผลกลาง.....	70
4.1.1 สถาปัตยกรรมชุดคำสั่ง nanoLADA.....	71
4.2 องค์ประกอบภายในสำหรับสถาปัตยกรรม nanoLADA.....	72
4.2.1 ชุด rejister (Register file).....	73
4.2.2 Extender.....	75
4.2.3 หน่วยความจำ.....	76
4.3 ทางเดินข้อมูลสำหรับสถาปัตยกรรมชุดคำสั่ง nanoLADA.....	77
4.3.1 ทางเดินข้อมูลสำหรับคำสั่ง ORI และ ORUI.....	78
4.3.2 ทางเดินข้อมูลสำหรับคำสั่ง ADD.....	79
4.3.3 ทางเดินข้อมูลสำหรับคำสั่ง LW.....	81
4.3.4 ทางเดินข้อมูลสำหรับคำสั่ง SW.....	81
4.3.5 ทางเดินข้อมูลสำหรับคำสั่ง BEQ.....	82
4.3.6 ทางเดินข้อมูลของคำสั่ง JMP.....	83
4.3.7 สรุปทางเดินข้อมูลสำหรับสถาปัตยกรรมชุดคำสั่ง nanoLADA.....	84
4.4 สัญญาณควบคุม (Control Signals).....	86
4.5 วิธีวิเคราะห์และความเร็วสัญญาณนาฬิกา.....	91
4.6 แบบฝึกหัดท้ายบท.....	93
5 หน่วยประมวลผลกลางแบบ multiple cycle.....	95
5.1 สมรรถนะของหน่วยประมวลผลกลางแบบ multiple cycle.....	98
5.2 ทางเดินข้อมูลสำหรับสถาปัตยกรรม nanoLADA แบบ multiple cycle.....	100
5.3 สัญญาณควบคุมสำหรับ multiple cycle processor.....	105
5.4 ไมโครโปรแกรม (Microprogram).....	108
5.4.1 Microsequencer.....	108
5.5 แบบฝึกหัดท้ายบท.....	113
6 การเพิ่มสมรรถนะด้วย Pipeline.....	115
6.1 สมรรถนะของ Pipeline กรณีอุดมคติ.....	116
6.2 ปัญหาที่เป็นอุปสรรคต่อการทำ Pipeline.....	118
6.2.1 Structural Hazard.....	119
6.3 Data Hazard.....	122
6.3.1 การแก้ปัญหา Data Hazard ด้วยวิธีการทางซอฟต์แวร์.....	123

6.3.2 การแก้ปัญหา Data Hazard ด้วยวิธีการ Hardware Forward.....	125
6.3.3 Data Hazard แบบ RAW, WAW, WAR.....	128
6.4 Control Hazard หรือ Branch Hazard.....	130
6.4.1 การแก้ปัญหา Control Hazard ด้วยวิธีการ Branch Delay Slot.....	131
6.5 Data Path.....	133
6.6 สรุป Pipeline.....	135
6.6.1 สมรรถนะของ Pipeline ในทางปฏิบัติ.....	135
6.7 แบบฝึกหัดท้ายบท.....	137
7 สถาปัตยกรรมร่วมสมัย.....	139
7.1 อนุกรมวิธานแบบ Flynn (Flynn's Taxonomy).....	139
7.1.1 Single Instruction Stream Single Data Stream (SISD).....	139
7.1.2 Single Instruction Stream Multiple Data Stream (SIMD).....	140
7.1.3 Multiple Instruction Stream Single Data Stream (MISD).....	140
7.1.4 Multiple Instruction Stream Multiple Data Stream (MIMD).....	141
7.2 สถาปัตยกรรมแบบ เวกเตอร์.....	141
7.3 สถาปัตยกรรมแบบ SuperScalar.....	143
7.4 สถาปัตยกรรมแบบ Very Long Instruction Word (VLIW).....	144
7.5 การสนับสนุนของตัวเปลลภาษาในสถาปัตยกรรมร่วมสมัย.....	145
7.5.1 Loop unroll.....	146
7.5.2 Software pipeline.....	148
7.5.3 โครงการ LLVM.....	149
7.6 สรุปสถาปัตยกรรมร่วมสมัย.....	151
7.7 แบบฝึกหัดท้ายบท.....	153
8 การจัดการหน่วยความจำ.....	155
8.1 Cache.....	156
8.2 หลักการท้องถิ่น (Locality).....	158
8.3 การจัดการ Cache เป้าองค์.....	160
8.3.1 Direct Mapped Cache.....	161
8.4 สมรรถนะของ Cache.....	164
8.5 การลด Miss Ratio.....	167
8.6 การลด Miss Penalty.....	169
8.7 Block Size.....	171
8.8 Associativity.....	173
8.9 Replacement Algorithm.....	177
8.10 การจัดการ Cache กรณีการเขียนข้อมูล (Write Management).....	178
8.11 โปรแกรมและสมรรถนะของ Cache.....	180
8.11.1 การปรับโปรแกรมเพื่อเพิ่ม Hit.....	180

8.11.2 การปรับโปรแกรมเพื่อลด Conflict Miss.....	181
8.12 หน่วยความจำเสมือน (Virtual Memory).....	182
8.12.1 การทำงานของหน่วยความจำเสมือน.....	183
8.13 การทำงานร่วมกันของ Cache และหน่วยความจำเสมือน.....	188
8.14 การเหลือขั้นตอนของ Cache และหน่วยความจำเสมือน.....	188
8.15 สรุป.....	190
8.16 แบบฝึกหัดท้ายบท.....	191
ภาคผนวก.....	193
ภาคผนวก ก สถาปัตยกรรมชุดคำสั่ง nanoLADA.....	195
ก.1 รูปแบบคำสั่ง 3 รูปแบบ.....	195
ก.2 คำสั่งของสถาปัตยกรรม nanoLADA.....	195
ภาคผนวก ข Verilog HDL ของสถาปัตยกรรม nanoLADA.....	197
ข.1 Register Files.....	197
ข.2 Extender.....	198
ข.3 ALU.....	198
ข.4 ADDER.....	199
ข.5 Mux 2:1.....	200
ข.6 control unit.....	200
ข.7 nanoCPU.....	203
ข.8 Micro PC.....	208
ภาคผนวก ค กิจกรรมเพิ่มเติม.....	209
ค.1 กิจกรรมบทบาทของคอมไฟเลอร์.....	209
ค.2 กิจกรรมโครงสร้าง Cache และสมรรถนะ.....	213
บรรณานุกรม.....	215

โครงสร้างภายใน

โครงสร้างภายใน

รูปที่ 1.1: ส่วนประกอบของระบบคอมพิวเตอร์.....	3
รูปที่ 1.2: ตัวอย่างขั้นตอนการแปลผลโปรแกรม.....	5
รูปที่ 3.1: ทางเดินข้อมูลของสถาปัตยกรรมแบบ accumulator.....	39
รูปที่ 3.2: ทางเดินข้อมูลของสถาปัตยกรรมแบบ stack.....	41
รูปที่ 3.3: ทางเดินข้อมูลของสถาปัตยกรรมแบบ memory-memory.....	42
รูปที่ 3.4: ทางเดินข้อมูลของสถาปัตยกรรมแบบ register-memory.....	44
รูปที่ 3.5: โครงสร้างทางเดินข้อมูลของสถาปัตยกรรมแบบ register-register.....	45
รูปที่ 3.6: การเรียงข้อมูล (a) big endian (b) little endian.....	50
รูปที่ 3.7: การต่อหน่วยความจำเข้ากับหน่วยประมวลผล (a) ความกว้างสายสัญญาณเป็นหนึ่ง (b) ความกว้างสายสัญญาณเป็นสอง.....	52
รูปที่ 3.8: การอ่านค่าแบบมี restricted alignment และ แบบ unrestricted alignment.....	53
รูปที่ 3.9: โครงสร้างการจัดวางหน่วยความจำ (word addressing และในเลขฐาน 16) (a) โปรแกรมภาษาซี (b) สถาปัตยกรรมแบบ unrestricted alignment และ (c) สถาปัตยกรรมแบบ restricted alignment.....	54
รูปที่ 3.10: รูปแบบคำสั่ง (instruction format) ในสถาปัตยกรรม nanoLADA.....	58
รูปที่ 3.11: เปรียบเทียบการทำงานระหว่าง caller save และ callee save.....	64
รูปที่ 4.1: ความสัมพันธ์ระหว่างสัญญาณนาฬิกา วงจรเรซิฟสม และ วงจรเรซิล์ดับ.....	70
รูปที่ 4.2: ชุดเรจิสเตอร์ขนาด 32x32 บิต สามารถอ่านได้ 2 ชุด และเขียนได้ 1 ชุดพร้อมกัน.....	74
รูปที่ 4.3: การทำงานของ Extender (a) zero extender (b) sign extender (c) zero padding.....	75
รูปที่ 4.4: โครงสร้างหน่วยความจำ.....	76
รูปที่ 4.5: ทางเดินข้อมูลสำหรับ PC \leftarrow PC+4.....	78
รูปที่ 4.6: ทางเดินข้อมูลเมื่อเพิ่มคำสั่ง ORI และ ORUI.....	79
รูปที่ 4.7: ทางเดินข้อมูลเมื่อเพิ่มคำสั่ง ADD.....	80
รูปที่ 4.8: ทางเดินข้อมูลเมื่อเพิ่มคำสั่ง LW.....	81
รูปที่ 4.9: ทางเดินข้อมูลเมื่อเพิ่มคำสั่ง SW.....	82
รูปที่ 4.10: ทางเดินข้อมูลส่วน PC เมื่อเพิ่มคำสั่ง BEQ.....	83
รูปที่ 4.11: ทางเดินข้อมูลส่วน PC เมื่อเพิ่มคำสั่ง JMP.....	84
รูปที่ 4.12: ทางเดินข้อมูลสำหรับสถาปัตยกรรม nanoLADA.....	85
รูปที่ 5.1: เปรียบเทียบเวลา (a) คำสั่งที่ไม่ต้องขยับข้อมูลจากหน่วยความจำ และ (b) คำสั่งที่ต้องอ่านข้อมูลจากหน่วยความจำ (กำหนดให้ความกว้างแทนเวลา).....	95
รูปที่ 5.2: ตัวอย่างการแบ่งขั้นตอนการทำงาน (a) การประมวลผลแบบ single cycle (b) การประมวลผลแบบ multiple cycle.....	96
รูปที่ 5.3: การแบ่งขั้นตอนการทำงาน.....	97

รูปที่ 5.4: การแทรกเรจิสเตอร์เงาเพื่อปรับทางเดินข้อมูลให้รองรับการทำงานแบบ multiple cycle.....	101
รูปที่ 5.5: ทางเดินข้อมูลสำหรับ multiple-cycle processor.....	105
รูปที่ 5.6: state diagram แสดงการทำงานของหน่วยประมวลผลกลางแบบ multiple cycle	106
รูปที่ 5.7: โครงสร้างของ Microsequencer.....	109
รูปที่ 5.8: state diagram สำหรับไมโครโปรแกรม.....	110
รูปที่ 6.1: เปรียบเทียบเวลาการทำงานแบบมี pipeline และ ไม่มี pipeline.....	116
รูปที่ 6.2: ปัญหา Structural Hazard.....	119
รูปที่ 6.3: ปัญหา structure hazard จากการที่แต่ละคำสั่งใช้จำนวน cycle ไม่เท่ากัน.....	120
รูปที่ 6.4: การแก้ปัญหา Structural Hazard ด้วยการ stall.....	121
รูปที่ 6.5: การแก้ structural hazard โดยการปรับขั้นตอนการทำงานของคำสั่ง R-type.....	122
รูปที่ 6.6: ปัญหา data hazard.....	123
รูปที่ 6.7: การแก้ปัญหา data hazard ด้วย hardware stall.....	126
รูปที่ 6.8: ตัวอย่างการแก้ปัญหา Data Hazard ด้วย hardware forward.....	127
รูปที่ 6.9: ตัวอย่างกรณีที่ไม่สามารถ forward ได้.....	127
รูปที่ 6.10: การทำ hardware stall กรณีที่ไม่สามารถทำ hardware forward ได้.....	128
รูปที่ 6.11: data hazard แบบ RAW, WAW และ WAR.....	129
รูปที่ 6.12: ตัวอย่างการทำงานของ Pipeline เมื่อมีการ bypass เพื่อแก้ปัญหา Control Hazard.....	131
รูปที่ 6.13: การแก้ปัญหา control hazard ด้วย branch delay slot (แบบ taken) และ (not taken).....	133
รูปที่ 6.14: การแยกเรจิสเตอร์คำสั่งแยกออกจากกันในแต่ละขั้นของ pipeline.....	134
รูปที่ 6.15: การส่งผ่านสัญญาณควบคุมสำหรับ pipeline.....	134
รูปที่ 7.1: เปรียบเทียบโปรแกรมบนสถาปัตยกรรมแบบหน่วยประมวลผลปกติ และสถาปัตยกรรมแบบเวกเตอร์.....	142
รูปที่ 7.2: สถาปัตยกรรม SuperScalar.....	144
รูปที่ 7.3: สถาปัตยกรรม Very Long Instruction Word.....	145
รูปที่ 7.4: การทำ loop unroll จำนวน 4 รอบ.....	146
รูปที่ 7.5: การทำ loop unroll ในภาษาแอสเซมบลี.....	147
รูปที่ 7.6: แนวคิดการทำ software pipeline.....	148
รูปที่ 7.7: การทำ software pipeline.....	149
รูปที่ 7.8: ตัวอย่าง LLVM IR.....	150
รูปที่ 8.1: ความเร็วและปริมาณหน่วยจัดเก็บข้อมูลที่มีในระบบคอมพิวเตอร์.....	157
รูปที่ 8.2: ลำดับการเข้าถึงข้อมูล.....	157
รูปที่ 8.3: ความถี่การอ้างอิงช่วงเลขที่อยู่ของ gcc.....	159
รูปที่ 8.4: ตัวอย่างการจัดគิ้นกี้ในหน่วยประมวลผลแบบ direct mapped cache.....	162

รูปที่ 8.5: โครงสร้าง direct mapped cache.....	163
รูปที่ 8.6: โครงสร้าง direct mapped cache ที่รองรับการทำงานของหลักการท่องคิ้นเชิงพื้นที่	164
รูปที่ 8.7: กรณี cache เพียง 1 บรรทัด.....	168
รูปที่ 8.8: การทำ multilevel cache เพื่อลด miss penalty.....	170
รูปที่ 8.9: block size ต่อ miss penalty.....	172
รูปที่ 8.10: กราฟความสัมพันธ์ระหว่าง block size, miss rate และ miss penalty.....	173
รูปที่ 8.11: ตัวอย่างการเกิด conflict miss ใน 1-way set associative cache.....	174
รูปที่ 8.12: การเพิ่ม set associative เพื่อลดปัญหา conflict miss.....	175
รูปที่ 8.13: โครงสร้างของระบบ cache แบบ 4-way set associative.....	176
รูปที่ 8.14: การใช้ Write Buffer เพื่อลดเวลาในการเขียนข้อมูลแบบ Write Through.....	179
รูปที่ 8.15: การเพิ่ม level ของ cache เพื่อบรรเทาปัญหา saturate ของ write buffer.....	179
รูปที่ 8.16: การทำงานของหน่วยความจำเสมือน.....	185
รูปที่ 8.17: การแปลง virtual address เป็น physical address.....	186
รูปที่ 8.18: การทำงานของ TLB.....	187
รูปที่ 8.19: การเข้าถึงข้อมูลในหน่วยความจำเมื่อไม่มี cache และหน่วยความจำเสมือน.....	188
รูปที่ 8.20: การแปลง virtual address และการอ้างอิงเลขที่อยู่ของ cache.....	189
รูปที่ 8.21: โครงสร้างเลขที่อยู่ที่ไม่สามารถทำการเหลือมเวลาได้.....	190

บรรณนิตรารง

ตารางที่ 2.1: เปรียบเทียบเวลาได้จากเกณฑ์เปรียบเทียบสมรรถนะ.....	18
ตารางที่ 2.2: การใช้ค่าเฉลี่ยเลขคณิตต่างหน้าหักเปรียบเทียบเวลา.....	19
ตารางที่ 2.3: การเปรียบเทียบสมรรถนะด้วยอัตราส่วน พร้อมค่าเฉลี่ยเลขคณิต.....	20
ตารางที่ 2.4: การเปรียบเทียบสมรรถนะด้วยอัตราส่วน พร้อมค่าเฉลี่ยเรขาคณิต.....	20
ตารางที่ 3.1: การอ้างอิงเลขที่อยู่แบบต่าง ๆ และการเทียบเคียงกับภาระระดับสูง.....	55
ตารางที่ 4.1: การทำงานของ ALU สำหรับ nanoLADA.....	87
ตารางที่ 4.2: สัญญาณควบคุมสำหรับสถาปัตยกรรม nanoLADA.....	89
ตารางที่ 5.1: สัญญาณควบคุมสำหรับหน่วยประมวลผลแบบ multiple cycle.....	106
ตารางที่ 5.2: ไมโครอินสตรัคชันของสัญญาณควบคุมบางส่วน.....	110
ตารางที่ 8.1: สมรรถนะของ replacement algorithm แบบ LRU และ RR.....	178

บรรณนิสมการ

สมการที่ 2.1 สมรรถนะและเวลา.....	11
สมการที่ 2.2 การหา Speedup จากสมรรถนะ.....	12
สมการที่ 2.3 การหา Speedup จากเวลา.....	12
สมการที่ 2.4 Elapse Time.....	15
สมการที่ 2.5 Cycle Time.....	22
สมการที่ 2.6 average CPI.....	22
สมการที่ 2.7 CPU Time/instruction.....	23
สมการที่ 2.8 Seconds/lnstruction.....	24
สมการที่ 2.9 CPU Time เมื่อทราบเวลาต่อคำสั่ง.....	24
สมการที่ 2.10 CPU Time เมื่อทราบ Cycle ต่อคำสั่ง.....	24
สมการที่ 2.11 Execution Time.....	28
สมการที่ 2.12 Overall Speedup.....	29
สมการที่ 6.1 CPU Time (pipeline).....	118
สมการที่ 6.2 สมรรถนะของ pipeline เมื่อมี stall cycle.....	135
สมการที่ 8.1 Memory Access Time (หน่วย cycle).....	166
สมการที่ 8.2 ค่า CPI กรณีที่มี cache.....	166

1 บทนำ

1.1 รู้ก่อนเรียน

เพื่อให้การศึกษาวิชาสถาปัตยกรรมคอมพิวเตอร์ง่ายขึ้น ผู้เรียนควรมีทักษะในการออกแบบ วงจรตรรกะ หรือวงจรดิจิตอล โดยองค์ความรู้ทางวงจรดิจิตอลที่มีนั้น ควรจะเพียงพอให้สามารถออกแบบได้ทั้งวงจรแบบผสม (combinational logic circuit) และวงจรเชิงลำดับ (sequential circuit) นอกจากนี้ผู้เรียนควรจะมีความรู้พื้นฐานทางซอฟต์แวร์พสมควรเพื่อประกอบความเข้าใจ กล่าวคือ ผู้เรียนควรจะมีประสบการณ์ในการเขียนโปรแกรม (ไม่ว่าจะเป็นภาษาไทย) ทั้งนี้หากผู้เรียนมีประสบการณ์กับภาษาแอกซ์ซ์เมบลี (assembly) ไม่ว่าจะเป็นสถาปัตยกรรมใด จะช่วยให้สามารถเข้าใจเนื้อหาบางส่วนได้ง่ายและรวดเร็วยิ่งขึ้นเป็นพิเศษ แต่หากไม่มีความตุ้มเคยกับภาษาแอกซ์ซ์เมบลี อย่างน้อยผู้เรียนควรจะเข้าใจหลักการเบื้องต้น เช่น ตัวแปร การจดogn น่วยความจำ ประโยคเงื่อนไขและการเรียกใช้งานโปรแกรมย่อย ในกรณีของพื้นฐานความรู้ด้านการเขียนโปรแกรม ปัจจุบันไม่น่าจะเป็นประเด็นแต่อย่างไร เนื่องจากการเรียนการสอนทางด้านวิทยาศาสตร์และวิศวกรรมศาสตร์สมัยใหม่ มักสอดแทรกองค์ความรู้เกี่ยวกับการเขียนโปรแกรมเข้าไปอยู่แล้ว เช่น นิสิตนักศึกษาชั้นปีที่ห็นในหลายหลักสูตรทางด้านคอมพิวเตอร์ มักจะได้เรียนวิชาการทำโปรแกรมด้วยภาษาจาวา (Java programming) หรือ การทำโปรแกรมด้วยภาษาไพธอน (Python programming) และนิสิตนักศึกษาชั้นปีที่สองทางสาขาวิชาวิศวกรรมคอมพิวเตอร์ มักได้เรียนภาษาซี (C programming) และพื้นฐานการออกแบบวงจรตรรกะหรือวงจรดิจิตอล

เนื้อหาในหนังสือเล่มนี้เรียบเรียงมาจากตำราต่าง ๆ ทั้งฉบับคลาสสิก (ตำนาน) ฉบับอ้างอิงฉบับอ่านเล่น รวมถึงงานตีพิมพ์และงานวิจัยด้านต่าง ๆ เนื่องจากสถาปัตยกรรมคอมพิวเตอร์เป็นศาสตร์ที่มีการพัฒนา ปรับปรุงและเปลี่ยนแปลงอยู่เสมอ ผู้เรียนควรค้นคว้าเพิ่มเติมจากแหล่งความรู้อื่นประกอบด้วย

โครงสร้างและสถาปัตยกรรมคอมพิวเตอร์ที่อธิบายในที่นี้ จะอธิบายโดยอาศัยพื้นฐานจากสถาปัตยกรรมชุดคำสั่งสมมุติ ชื่อ nanoLADA ซึ่งเป็นฉบับย่อของ LADA (Light-weighted Architecture for Design and Analysis) เป็นหลัก โดย LADA เป็นสถาปัตยกรรมคอมพิวเตอร์ประเภท RISC แบบ 32 บิต ที่ผู้เรียนพัฒนาขึ้นเพื่อประกอบการเรียนการสอนโดยเฉพาะ (ผู้เรียนยังไม่ต้องกังวลว่า สถาปัตยกรรมคอมพิวเตอร์ประเภท RISC คืออะไร เมื่อถึงเนื้อหาส่วนที่เกี่ยวข้องก็จะเข้าใจเอง) สถาปัตยกรรม LADA มุ่งเน้นให้มีโครงสร้างเข้าใจได้ง่าย ไม่ซับซ้อน เหมาะกับการศึกษาในระดับเบื้องต้น ในส่วนที่

เป็นการกล่าวถึงเรื่องต่อ�อดต่าง ๆ อาจมีการแทรกเนื้อหาของสถาปัตยกรรมอื่นเพื่อประกอบการอธิบายเพิ่มเติมบ้าง การแทรกรายละเอียดของสถาปัตยกรรมเหล่านี้ เน้นเพื่อให้เกิดการเปรียบเทียบและเห็นข้อแตกต่าง ทั้งนี้ผู้เขียนจะไม่แทรกเนื้อหาให้มากเกินไป โดยการสอดแทรกจะขึ้นกับความต้องเนื่องและความเหมาะสมของเนื้อหาเป็นหลัก ในด้านอย่างประกอบคำอธิบาย จะพยายามใช้สถาปัตยกรรม LADA เป็นพื้นฐาน (ยกเว้นกรณีที่ต้องการเปรียบเทียบกับสถาปัตยกรรมอื่น) เพื่อให้เกิดความต้องเนื่องของเนื้อหา

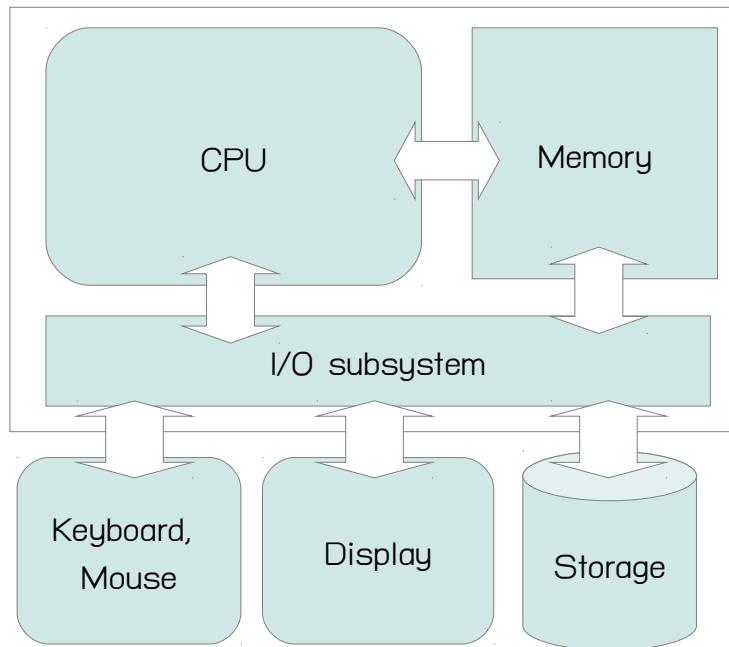
1.2 ส่วนประกอบของระบบคอมพิวเตอร์

ระบบคอมพิวเตอร์โดยทั่วไป ประกอบด้วยหน่วยประมวลผลกลาง แบ็ปปิมพ์ เม้าส์ จอภาพ เครื่องพิมพ์ ระบบสื่อประสม (multimedia) และอุปกรณ์ต่อพ่วงอีกมากมาย โครงสร้างดังกล่าวสามารถจำแนกตามการทำงานได้เป็น

- หน่วยประมวลผลกลาง (central processing unit)
- อินพุต (keyboard, mouse, etc...)
- เอ้าท์พุต (display, printer, etc...)
- หน่วยความจำ (memory)
- หน่วยจัดเก็บข้อมูล (storage, disk, cd, tape, etc)

การแบ่งในลักษณะนี้ เป็นการแบ่งส่วนประกอบของคอมพิวเตอร์ตามมุ่งมองของผู้ใช้งาน ทั่วไป ในทางสถาปัตยกรรมนิยมแบ่งส่วนประกอบของคอมพิวเตอร์เป็นสามส่วนหลักตามลักษณะของหน้าที่ดังแสดงในรูปที่ 1.1 ได้แก่ หน่วยประมวลผลกลาง หน่วยความจำ และระบบอินพุตเอ้าท์พุต จะสังเกตเห็นว่า อินพุต เอ้าท์พุต และ หน่วยจัดเก็บข้อมูล ถูกมองรวมเป็นระบบเดียวกันทั้งหมด ส่วนหนึ่งเนื่องจากในทางสถาปัตยกรรม อุปกรณ์เหล่านี้เชื่อมต่อกับหน่วยประมวลผลกลางผ่านระบบสายสัญญาณย่อยอินพุต/เอ้าท์พุต (I/O subsystem) เมื่อนอกนั้น ต่างจากหน่วยความจำที่มักจะมีสายสัญญาณเชื่อมต่อเฉพาะสำหรับหน่วยความจำ (memory bus)

หนังสือเล่มนี้มุ่งความสนใจไปที่หน่วยประมวลผลกลางเป็นหลักก่อน จากนั้นจึงกล่าวถึงองค์ประกอบรอบข้าง เช่น หน่วยความจำและส่วนต่อขยายต่าง ๆ เมื่อส่วนดังกล่าวมีผลกับสมรรถนะของระบบ จนส่งผลให้การออกแบบและการวิเคราะห์แตกต่างไป



รูปที่ 1.1: ส่วนประกอบของระบบคอมพิวเตอร์

1.3 คำจำกัดความ

สถาปัตยกรรมคอมพิวเตอร์ (computer architecture) ประกอบขึ้นจากคำสองคำคือ สถาปัตยกรรม (architecture) และ คอมพิวเตอร์ (computer) สถาปัตยกรรมหมายถึง โครงสร้าง องค์ประกอบ หรือ กรอบความคิด ของระบบใด ๆ (ซึ่งในความหมายทั่วไป มัก จะหมายถึงอาคาร หรือ สิ่งก่อสร้าง) เมื่อนำมาประยุกต์ใช้ในทางคอมพิวเตอร์ จึงหมายถึง โครงสร้างและองค์ประกอบของระบบคอมพิวเตอร์ ในทำนองเดียวกัน เรายืนยันเรียกผู้ออกแบบสถาปัตยกรรมคอมพิวเตอร์ว่า สถาปนิก (architect) เช่นเดียวกันกับสถาปนิกผู้ออกแบบอาคารเช่นกัน

เมื่อประมวลความหมายต่าง ๆ เข้าด้วยกัน สรุปความได้ว่า สถาปัตยกรรมคอมพิวเตอร์ หมายถึงโครงสร้างการทำงานของเครื่องคอมพิวเตอร์ โครงสร้างของหน่วยประมวลผลกลาง การแทนข้อมูลระดับเครื่อง สถาปัตยกรรมชุดคำสั่ง (รวมความถึง ภาษาเครื่อง หรือ ภาษาแอสเซมบลี) รวมถึงการจัดการระบบหน่วยความจำ การเชื่อมต่อ ระหว่างเครื่องคอมพิวเตอร์และอุปกรณ์ต่อพ่วง

การศึกษาวิชาสถาปัตยกรรมคอมพิวเตอร์ จึงเป็นการศึกษาที่มุ่งเน้นถึงโครงสร้างที่จำเป็นต่อ การออกแบบ และใช้งานส่วนต่าง ๆ ของระบบคอมพิวเตอร์ อย่างไรก็ตามเมื่อกล่าวถึงคำว่า สถาปัตยกรรมคอมพิวเตอร์ โดยทั่วไปผู้คนมักนึกถึงหน่วยประมวลผลกลางเป็นหลัก ซึ่งเป็น แนวคิดที่ไม่ถูกต้องนัก

หากอ้างอิงจากความหมายของสถาปัตยกรรมคอมพิวเตอร์ การศึกษาเรื่องสถาปัตยกรรม คอมพิวเตอร์ จะต้องประกอบไปด้วยอย่างน้อยสามส่วน คือ

- สถาปัตยกรรมชุดคำสั่ง
- โครงสร้างคอมพิวเตอร์
- การสร้างระบบคอมพิวเตอร์

เนื้อหาในหนังสือเล่มนี้จะครอบคลุมทั้งสามส่วน แต่จะมุ่งเน้นที่ส่วนสถาปัตยกรรมชุดคำสั่ง และโครงสร้างคอมพิวเตอร์เป็นสำคัญ ทั้งนี้เพื่อให้ผู้เรียนเข้าใจอย่างลึกซึ้ง จึงขอขยาย ความคำจำกัดความของ สถาปัตยกรรมชุดคำสั่ง โครงสร้างคอมพิวเตอร์ และ การสร้าง ระบบคอมพิวเตอร์ ดังต่อไปนี้

1.4 สถาปัตยกรรมชุดคำสั่ง (Instruction Set Architecture)

สถาปัตยกรรมชุดคำสั่ง เป็นตัวกลางในการอธิบายถึงโครงสร้างและความสามารถของหน่วย ประมวลผลกลางโดยมองจากมุมมองของผู้พัฒนาซอฟต์แวร์และผู้พัฒนาฮาร์ดแวร์ เพื่อ ประกอบความเข้าใจ เรายังทำความเข้าใจขั้นตอนการพัฒนาซอฟต์แวร์โดยทั่วไปดังนี้

การพัฒนาซอฟต์แวร์โดยทั่วไป ผู้พัฒนาหรือโปรแกรมเมอร์ มักจะพัฒนาโปรแกรมโดยใช้ ภาษาชั้นสูง (เช่น ภาษา C) จากนั้นโปรแกรมจะถูกแปลโดยคอมไพล์เลอร์ (compiler) เป็น ภาษาแอสเซมบลี ซึ่งคอมไпал์เม้นต์จะมีการแปลเป็นภาษากลาง (Intermediate code) และมีการปรับปรุงโปรแกรมด้านต่าง ๆ ให้เหมาะสม (optimization) จากนั้น โปรแกรมภาษาแอสเซมบลีเหล่านั้นจึงถูกแปลโดยแอสเซมเบลอร์ (assembler) เป็นภาษา เครื่อง (machine code) ซึ่งเป็นการเข้ารหัสแบบบิตในที่สุด ดังตัวอย่างในรูปที่ 1.2

จากลักษณะการพัฒนาซอฟต์แวร์ดังกล่าวจะพบว่า โปรแกรมที่มีต้นฉบับภาษาชั้นสูงเหมือน กันนั้น เมื่อถูกแปลโดยคอมไпал์เลอร์ และ แอสเซมเบลอร์ แล้วจะได้ภาษาเครื่อง ซึ่งแตกต่าง กันไปตามหน่วยประมวลผลกลางที่มีสถาปัตยกรรมชุดคำสั่งแตกต่างกัน ซึ่งอาจกล่าวได้ว่า ชุดคำสั่งภาษาเครื่องเหล่านี้เป็นตัวกลางในการเชื่อมต่อระหว่างฮาร์ดแวร์และซอฟต์แวร์

เพื่อใช้สั่งให้เครื่องคอมพิวเตอร์ทำงานตามโปรแกรมที่ต้องการ เราอาจเรียกชุดคำสั่งในภาษาแอ๙สเซมบลีและภาษาเครื่องเหล่านี้ว่า สถาปัตยกรรมชุดคำสั่ง ซึ่งก็คือ โครงสร้างชุดคำสั่งสำหรับสั่งให้หน่วยประมวลผลกลางที่ใช้สถาปัตยกรรมดังกล่าวทำงานตามที่ชุดคำสั่ง(โปรแกรม) กำหนด

```
int average (int a, int b) {
    return (a+b)/2;
}
```

Compiler

Intermediate/Assembly Code

```
<average>:
 0: 55          push  %rbp
 1: 48 89 e5   mov   %rsp,%rbp
 4: 89 7d fc   mov   %edi,-0x4(%rbp)
 7: 89 75 f8   mov   %esi,-0x8(%rbp)
 a: 8b 55 fc   mov   -0x4(%rbp),%edx
 d: 8b 45 f8   mov   -0x8(%rbp),%eax
 10: 01 d0     add   %edx,%eax
 12: 89 c2     mov   %eax,%edx
 14: c1 ea 1f   shr   $0x1f,%edx
 17: 01 d0     add   %edx,%eax
 19: d1 f8     sar   %eax
 1b: 5d         pop   %rbp
 1c: c3         retq
```

Assembler

Binary/Machine Code

```
5548 89e5 897d fc89 75f8 8b55 fc8b 45f8
01d0 89c2 clea 1f01 d0d1 f85d c3
```

รูปที่ 1.2: ตัวอย่างขั้นตอนการแปลผลโปรแกรม

จากมุมมองของผู้พัฒนาอาร์ดแวร์นั้น ผู้พัฒนาอาร์ดแวร์เพียงออกแบบสถาปัตยกรรมชุดคำสั่งและสร้างวงจรประมวลผลให้สามารถทำงานได้ตามสถาปัตยกรรมนั้น โดยไม่ต้องคำนึงถึงการพัฒนาซอฟต์แวร์ ทำให้ผู้พัฒนาซอฟต์แวร์เป็นอิสระจากอาร์ดแวร์

ในการพัฒนาซอฟต์แวร์โดยเฉพาะอย่างยิ่งซอฟต์แวร์ระบบ ผู้พัฒนาจึงควรมีความรู้เกี่ยวกับสถาปัตยกรรมชุดคำสั่งเหล่านี้เพื่อที่จะสามารถพัฒนาซอฟต์แวร์ได้ โดยไม่จำเป็นต้องอาศัย

ความรู้เกี่ยวกับฮาร์ดแวร์ด้านล่าง (ในระดับล่าง) อย่างไรก็ตามหน่วยประมวลผลกลางที่แตกต่างกัน อาจมีสถาปัตยกรรมชุดคำสั่งที่เหมือนกันได้ ซึ่งนั่นหมายความว่าซอฟต์แวร์ที่ทำงานบนระบบคอมพิวเตอร์เครื่องหนึ่งจะสามารถทำงานได้บนระบบคอมพิวเตอร์ทุกระบบที่มีโครงสร้างสถาปัตยกรรมชุดคำสั่งเหมือนกัน ตัวอย่างเช่น ซอฟต์แวร์ที่พัฒนาขึ้นสำหรับสถาปัตยกรรม 80486 ของ Intel สามารถทำงานได้บนสถาปัตยกรรม Pentium III ของ Intel หรือ สถาปัตยกรรม Opteron ของ AMD เป็นต้น ทั้งนี้ เพราะสถาปัตยกรรมเหล่านี้ แม้จะมีโครงสร้างภายในที่แตกต่างกัน แต่ต่างก็ใช้สถาปัตยกรรมชุดคำสั่งเดียวกัน (คล้ายคลึงกัน) จนเรียกว่าໄດ້ว่ามีความเข้ากันได้ในระดับคำสั่ง (instruction-level compatibility)

แม้ว่าการอ้างอิงถึงสถาปัตยกรรมคอมพิวเตอร์โดยใช้สถาปัตยกรรมชุดคำสั่งเป็นหลัก จะเป็นประโยชน์ในการพัฒนาซอฟต์แวร์และฮาร์ดแวร์ก็ตาม แต่บางกรณีการพัฒนาซอฟต์แวร์ และฮาร์ดแวร์โดยยึดติดกับสถาปัตยกรรมชุดคำสั่งมากเกินไป ทำให้มีความสามารถคิดค้นหรือ พัฒนาสถาปัตยกรรมคอมพิวเตอร์แบบใหม่ขึ้นมาได้ อย่างไรก็ตามการที่หน่วยประมวลผลรุ่นใหม่สามารถรองรับการทำงานของสถาปัตยกรรมแบบเดิมได้ ก็เป็นนโยบายทางการตลาดที่ดีอย่างนึง เนื่องจากทันทีที่สถาปัตยกรรมหรือหน่วยประมวลผลกลางรุ่นใหม่อกร่างกายตลาด ก็จะมีซอฟต์แวร์รองรับการทำงานทันที เช่นกัน ตัวอย่างที่เห็นได้ชัดคือ การที่หน่วยประมวลผลกลางรุ่นล่าสุดของบริษัท Intel ยังคงสามารถประมวลผลโปรแกรมที่พัฒนาบนระบบหน่วยประมวลผลรุ่น 8088 ที่พัฒนาขึ้นมากกว่า 40 ปีที่แล้วได้

1.5 โครงสร้างคอมพิวเตอร์ (Computer Organization)

โครงสร้างคอมพิวเตอร์เป็นโครงสร้างภายในของระบบคอมพิวเตอร์ ที่ทำให้หน่วยประมวลผลกลาง และส่วนประกอบต่าง ๆ สามารถทำงานร่วมกันสอดคล้องกับสถาปัตยกรรมชุดคำสั่งที่กำหนดได้

หน่วยประมวลผลกลาง เป็นศูนย์กลางในการทำงานของระบบคอมพิวเตอร์ หน่วยประมวลผลกลางต้องสามารถทำงานได้ตามสถาปัตยกรรมชุดคำสั่งที่ระบุไว้ หากเปรียบเทียบกับร่างกายของมนุษย์ หน่วยประมวลผลกลางเป็นสมองสมอง ภายในจำแนกออกเป็นองค์ประกอบต่าง ๆ ได้ดังนี้

- หน่วยประมวลผลทางคณิตศาสตร์และทางตรรกะ (arithmetic logical unit หรือ ALU)
- หน่วยควบคุม (control unit)

- เรจิสเตอร์ (register) หรือ ที่พักข้อมูลชั่วคราว
- ระบบสายสัญญาณสำหรับการเชื่อมต่อต่าง ๆ (bus)

เหตุที่หน่วยประมวลผลกลางจำเป็นจะต้องมีความสามารถในการจำนั้น เพราะในการคำนวณทุกครั้งมีความจำเป็นจะต้องใช้การจำ¹ ประกอบการคิดคำนวณเสมอ เพื่อไม่ให้สับสนกับหน่วยความจำทั่วไปประเภทอื่น ๆ จึงนิยมเรียกที่พักข้อมูลชั่วคราวภายในหน่วยประมวลผลกลางนี้ว่าเรจิสเตอร์

นอกจากนี้ ระบบคอมพิวเตอร์ยังมีระบบบอทอยู่อีกหน่วยประมวลผลกลางอีก เช่น หน่วยความจำหลัก (main memory) อุปกรณ์ต่าง ๆ (อินพุต/เอาต์พุต) และระบบสายสัญญาณต่าง ๆ การกล่าวถึงโครงสร้างคอมพิวเตอร์นั้น จะต้องครอบคลุมถึงส่วนต่าง ๆ เหล่านี้ด้วยเช่นกัน ในหนังสือเล่มนี้ จะนำเสนอรายละเอียดเฉพาะหน่วยประมวลผลกลาง และระบบหน่วยความจำนั้น

1.6 การสร้างระบบคอมพิวเตอร์ (Implementation)

การสร้างระบบคอมพิวเตอร์ในที่นี้หมายถึง การสร้างวงจรและอุปกรณ์เชื่อมต่อต่าง ๆ ให้เป็นไปตามข้อกำหนดที่ระบุไว้ในโครงสร้างคอมพิวเตอร์ องค์ความรู้ที่เกี่ยวข้องประกอบด้วย องค์ความรู้ทางวิศวกรรมคอมพิวเตอร์และองค์ความรู้ทางวิศวกรรมไฟฟ้า

องค์ความรู้ทางวิศวกรรมคอมพิวเตอร์ที่เกี่ยวข้องคือ ระบบสายสัญญาณและการเชื่อมต่อ (organization) สถาปัตยกรรมชุดคำสั่ง (instruction set architecture) โดยไม่สนใจคุณลักษณะทางไฟฟ้า² หรือโครงสร้างของวงจร บุคคลที่ทำหน้าที่ดังกล่าวเรียกว่า วิศวกรออกแบบ (design engineer)

องค์ความรู้ทางวิศวกรรมไฟฟ้าที่เกี่ยวข้องคือ การออกแบบแบบแผนผังวงจร (circuit layout) การสร้างวงจร (circuit design) และ การทวนสอบ (verification)

1 หากเปรียบเทียบกับการคิดคำนวณต่าง ๆ ของมนุษย์จะพบว่า การคิดคำนวณของมนุษย์ต้องใช้ความจำเพื่อกัน เช่น กรณีการบวกเลขแบบเด็ก ๆ หากเราต้องการนำ 2+3 เรน้ำกจะสอนให้เด็กนำสองไปในใจ ซึ่งมานามน้ำ แล้วนับต่อจนครบ ซึ่งจะพบว่ามีการใช้ความจำควบคู่กับการคำนวณด้วยเสมอ

2 คุณลักษณะทางไฟฟ้า เช่น แรงดันไฟฟ้า 5 โวลต์ (Volts) แทนตรรกะสูง (logic high)

เนื้อหาในหนังสือเล่มนี้ จะไม่ครอบคลุมถึงการสร้างระบบคอมพิวเตอร์ในเชิงลึก แต่จะบรรยายการออกแบบโดยใช้แผนภาพ (block-level diagram) และ วงจรตรรกะ (logic circuit) เพื่อประกอบความเข้าใจเป็นหลักเท่านั้น ในบางส่วนผู้เขียนจะนำภาษา Verilog HDL³ มาช่วยในการอธิบายเพิ่มเติม (อยู่ในภาคผนวก ฯ)

1.7 เป้าหมายของการออกแบบ

ประเด็นสุดท้ายที่จะกล่าวถึงในบทนี้ คือ เป้าหมายของการออกแบบ กล่าวคือ ทุกสถาปัตยกรรมและการออกแบบ จะต้องมีการแยกคุณสมบัติบางอย่างเพื่อประโยชน์บางอย่างเสมอ ข้อความนี้น่าจะสามารถใช้ได้กับการออกแบบและสถาปัตยกรรมทุกอย่าง ไม่จำกัดเฉพาะสถาปัตยกรรมคอมพิวเตอร์เท่านั้น ตัวอย่างเช่น รถยนต์ที่ออกแบบให้ประหยัดน้ำมันจะแยกด้วยความสามารถในการเร่งความเร็วเครื่องยนต์ที่ต่ำลง และมักใช้เครื่องยนต์ที่มีขนาดเล็กทำให้มีแรงม้าลดลง เป็นต้น

ในทางสถาปัตยกรรมคอมพิวเตอร์ ผู้ออกแบบระบบคอมพิวเตอร์ (โดยเฉพาะอย่างยิ่งหน่วยประมวลผลกลาง) จะเป็นจะต้องแยกเปลี่ยนคุณสมบัติบางอย่าง เพื่อให้หน่วยประมวลผลและระบบคอมพิวเตอร์สามารถทำงานอย่างได้ดี เช่น กัน เช่น หน่วยประมวลผลกลางที่สามารถทำงานด้านกราฟฟิกได้ดีมักจะกินพลังงานทำให้เกิดความร้อนสูง ในขณะที่อุปกรณ์พกพาขนาดเล็กที่ทำงานบนแบตเตอรี่มักจะไม่สามารถประมวลผลงานด้านกราฟฟิกได้มากนัก

ด้วยเงื่อนไขนี้ การเลือกใช้งานระบบคอมพิวเตอร์ซึ่งจำเป็นจะต้องพิจารณาถึงคุณสมบัติของหน่วยประมวลผลและระบบที่ใช้ ให้เหมาะสมกับงานที่ต้องการทำ เช่น กัน เพราะไม่มีสถาปัตยกรรมคอมพิวเตอร์ใดที่ดีพร้อมในทุกด้าน (ซึ่งเป็นจุดประสงค์หนึ่งของการศึกษาสถาปัตยกรรมคอมพิวเตอร์ เช่นกัน)

³ Verilog HDL เป็นภาษาสำหรับการบรรยายโครงสร้างและการออกแบบฮาร์ดแวร์ (Hardware Description Language) มีโครงสร้างคล้ายภาษาซี เป็นที่นิยมใช้สำหรับบรรยายการออกแบบในสหรัฐอเมริกา

1.8 แบบฝึกหัดท้ายบท

1. สถาปัตยกรรมคอมพิวเตอร์ คืออะไร
2. ส่วนประกอบของคอมพิวเตอร์ มีอะไรบ้าง
3. ในการพัฒนาโปรแกรมด้วยภาษาชั้นสูง เรามีขั้นตอนในการพัฒนาโปรแกรมต่าง ๆ เหล่านี้อย่างไร จึงจะทำให้โปรแกรมถูกแปลงเป็นชุดคำสั่งสำหรับทำงานบนหน่วยประมวลผลได้
4. สถาปัตยกรรมชุดคำสั่งคืออะไร เป็นประโยชน์ในการพัฒนาซอฟต์แวร์และฮาร์ดแวร์หรือไม่อย่างไร จงอธิบาย
5. (ทบทวนความรู้ด้านการออกแบบระบบจักรกล) จงสร้างวงจรบวกเลขที่จะบวกเลขขนาด 2 บิต และ ตัวทดอีก 1 บิต แล้วให้คำตอบเป็นผลลัพธ์พร้อมตัวทด
6. (ทบทวนความรู้ด้านการออกแบบระบบจักรกล) จงสร้างวงจรบวกเลขแบบอนุกรม (serial adder) แบบ Moore machine และ Mealy machine
7. จงยกตัวอย่างข้อเด่นข้อด้อยของสถาปัตยกรรมต่าง ๆ ที่พบทั่วไปในท้องตลาดมาอย่างน้อย 3 สถาปัตยกรรม (เช่น Intel Xeon, Intel Celeron, และ หน่วยประมวลผลที่มักพบในอุปกรณ์พกพาทั่วไป)

2 สมรรถนะของระบบคอมพิวเตอร์

สมรรถนะ คือ การวัดและการเปรียบเทียบการทำงาน การศึกษาเรื่องสมรรถนะของระบบคอมพิวเตอร์จะมุ่งเน้นถึงปัจจัยต่าง ๆ ที่ส่งผลกระทบต่อการทำงานของระบบคอมพิวเตอร์ โดยมุ่งให้สามารถเลือกใช้ และ รู้จักการวัดสมรรถนะของระบบ ทั้งนี้ เพื่อใช้ประกอบการศึกษาและการตัดสินใจว่า การวิเคราะห์สมรรถนะของระบบคอมพิวเตอร์ด้วยวิธีการใด ๆ ให้ผลในทางปฏิบัติที่ดีขึ้นอย่างไร โดยในที่นี้จะกล่าวถึงในส่วนของการวัดในเชิงปริมาณเป็นหลัก

เนื้อหาในบทนี้จะกล่าวถึงการวัดสมรรถนะ เกณฑ์ การเปรียบเทียบ แบบจำลองสมรรถนะ และแนวทางในการปรับปรุงสมรรถนะ

2.1 การวัดสมรรถนะของระบบคอมพิวเตอร์

การวัดสมรรถนะของระบบคอมพิวเตอร์ จะใช้เวลาเป็นหลักในการวัด โดยระบบคอมพิวเตอร์ที่ใช้เวลาในการทำงานหนึ่งน้อยกว่า จะมีสมรรถนะสูงกว่าระบบคอมพิวเตอร์ที่ใช้เวลามากกว่าในการทำงานโปรแกรมเดียวกัน หรือ กล่าวอีกนัยหนึ่งได้ว่า สมรรถนะเป็นส่วนกลับของเวลา ดังสมการที่ 2.1 ระบบที่ใช้เวลาในการประมวลผลมากจะมีสมรรถนะต่ำ ระบบที่ใช้เวลาในการประมวลผลน้อยจะมีสมรรถนะสูง

$$\text{Performance} = \frac{1}{\text{Execution Time}}$$

สมการที่ 2.1 สมรรถนะและเวลา

อีกแนวทางหนึ่งในการเปรียบเทียบสมรรถนะคือ การเปรียบเทียบสมรรถนะระหว่างระบบ เช่น การกล่าวว่า “เครื่องคอมพิวเตอร์ X เร็วกว่าเครื่อง Y เป็น g เท่า” หรือ “เครื่องคอมพิวเตอร์ X มีสมรรถนะเหนือกว่าเครื่อง Y เป็น g เท่า” ประโยชน์ดังกล่าวสามารถอธิบายในทางคณิตศาสตร์ได้ดังสมการที่ 2.2 ทั้งนี้ค่า g ที่ได้ คืออัตราส่วนที่เครื่องคอมพิวเตอร์หนึ่งมีสมรรถนะสูงกว่าอีกเครื่องหนึ่งเป็นจำนวนเท่า นิยมเรียกว่า Speedup

$$\text{Speedup}(n) = \frac{\text{Performance}_x(1/\text{seconds})}{\text{Performance}_y(1/\text{seconds})}$$

สมการที่ 2.2 การหา Speedup จากสมรรถนะ

เนื่องจากสมรรถนะเป็นส่วนกลับของเวลา ดังนั้นการเปรียบเทียบสมรรถนะ อาจคำนวณจากเวลาได้เช่นกัน โดยการรวม สมการที่ 2.1 และ สมการที่ 2.2 เข้าด้วยกัน เราจะได้สมการอธิบายการคำนวณ Speedup ดังสมการที่ 2.3

$$\text{Speedup}(n) = \frac{\text{Performance}_x}{\text{Performance}_y} = \frac{\frac{1}{\text{Execution Time}_x}}{\frac{1}{\text{Execution Time}_y}} = \frac{\text{Execution Time}_y}{\text{Execution Time}_x}$$

สมการที่ 2.3 การหา Speedup จากเวลา

ตัวอย่างที่ 2.1 แสดงการเปรียบเทียบสมรรถนะด้วยเวลา จากตัวอย่างจะเห็นว่า เครื่องที่ใช้เวลามากกว่าจะมีสมรรถนะต่ำกว่า ในตัวอย่างนี้นอกจจะสรุปได้ว่า เครื่อง A (ซึ่งใช้เวลา น้อยกว่า) มีสมรรถนะสูงกว่าเครื่อง B เป็น 1.25 เท่าแล้ว ยังสรุปได้อีกว่า เครื่อง A มีความเร็วสูงกว่าเครื่อง B อยู่ 1.25 เท่าเช่นกัน

ประเด็นช่วยจำ ผู้เรียนพยายามมักสับสนระหว่างสมรรถนะ และ เวลา หลักในการคำนึงหากสมรรถนะสูง หมายความว่า เร็ว ใช้เวลาต่ำ หรือ น้อย และหากสมรรถนะต่ำ หมายความว่า จะต้องใช้เวลาสูง หรือ มาก และหลักการนี้ ก็ช่วยให้คิดต่อໄດ้ว่า หากบอกว่า A มีสมรรถนะสูงกว่า B อยู่ n เท่า ย่อมหมายความว่า B ใช้เวลามากกว่า(มากกว่า) A ดังนั้น n ย่อมมาจากการใช้เวลาของ B (ซึ่งมีค่ามากกว่า) หารด้วยเวลาของ A (ซึ่งมีค่าน้อยกว่า)

ตัวอย่างที่ 2.1: การเปรียบเทียบสมรรถนะด้วยเวลา

จากการทดลองวัดเวลาในการประมวลผลของหน่วยประมวลผลกลางโดยใช้ benchmark program ผลลัพธ์ที่ได้เป็นดังนี้

เครื่อง A ใช้เวลาในการทำงาน 20 วินาที

เครื่อง B ใช้เวลาในการทำงาน 25 วินาที

เครื่องคอมพิวเตอร์เครื่องใดมีสมรรถนะสูงกว่ากัน และ สูงกว่ากันเป็นกี่เท่า

เครื่อง A มีสมรรถนะสูงกว่าเครื่อง B เนื่องจากสามารถประมวลผลโปรแกรมทดสอบได้เร็วกว่า

$$\text{จาก } Speedup = \frac{[Execution Time_Y]}{[Execution Time_X]}$$

$$\text{ได้ว่า } Speedup = \frac{[Execution Time_B]}{[Execution Time_A]} = \frac{25}{20} = 1.25$$

\therefore เครื่อง A มีสมรรถนะสูงกว่าเครื่อง B เป็น 1.25 เท่า

2.2 เวลาตอบสนอง และ ปริมาณงานที่ทำได้ต่อหน่วยเวลา

แม้การใช้เวลาเป็นเกณฑ์ที่จะช่วยให้การวัดสมรรถนะทำได้ง่ายขึ้น แต่เนื่องจากเวลาในการทำงานของระบบคอมพิวเตอร์ เพื่อประมวลผลโปรแกรม อาจหมายถึงเวลาในการตอบสนอง (response time) ของเครื่องคอมพิวเตอร์ได้ เช่น กัน เช่น เมื่อผู้ใช้สั่งให้โปรแกรมค้นหาข้อมูล โปรแกรมใช้เวลาานนท์ได้หลังจากเริ่มค้นหาและได้ผลลัพธ์แสดงออกมาก หรือ อาจหมายถึง ปริมาณงานที่ทำได้ต่อหน่วยเวลา (throughput) เช่น เครื่องคอมพิวเตอร์ สามารถประมวลผลโปรแกรมได้ 2 โปรแกรมภายในเวลา 1 วินาที เป็นต้น ซึ่ง เวลาตอบสนอง และ ปริมาณงานที่ทำได้ต่อหน่วยเวลา อาจจะมีได้เป็นไปในทิศทางเดียวกันเสมอไป

มีความเป็นไปได้ว่า เครื่องที่มีปริมาณงานที่ทำได้ต่อหน่วยเวลาสูง จะจะมีเวลาตอบสนองที่ต่ำได้ เช่น เครื่องคอมพิวเตอร์ประมวลผลงานเสร็จ 5 งานภายใน 10 นาที หากแต่เวลาที่ใช้ในการประมวลผลงานทั้ง 5 นั้น ก็เฉลี่ยประมาณ 10 นาที เช่นกันโดยการประมวลผลของ

ทั้ง 5 งานเกิดขึ้นพร้อม ๆ กัน ในขณะที่อีกเครื่องหนึ่ง ใช้เวลาในการประมวลผลงานละ 3 นาที หากแต่ทำได้เพียงทีละ 1 งานหมายความว่า จะต้องใช้เวลาทำงานรวมกันทั้ง 5 งาน เป็น 15 นาที เป็นต้น จึงด้วยอย่างตั้งกล่าว จะพบว่าเครื่องคอมพิวเตอร์เครื่องแรก จะมี ปริมาณงานที่ทำได้ต่อหน่วยเวลามากกว่าเครื่องที่สอง ในขณะที่เครื่องที่สองจะมีเวลาตอบสนองต่ำกว่าเครื่องแรก

ประเด็นช่วยจำ ตัวอย่างหนึ่งที่อาจจะช่วยให้เข้าใจข้อแตกต่างระหว่างเวลาตอบสนองและ ปริมาณงานที่ทำได้ต่อหน่วยเวลา คือ การโอนถ่ายข้อมูลขนาดใหญ่ ตัวอย่างเช่น หากเรา จำเป็นจะต้องโอนถ่ายข้อมูลขนาดใหญ่มาก (20 TB) ระหว่างหน่วยงานที่ห่างกัน 1 กิโลเมตร แม้ทั้งสองหน่วยงานจะเชื่อมต่อกันด้วยสายเคเบิลไลด์แก็วความเร็วสูง 1 Gb/s จะ ต้องใช้เวลากว่า 48 ชั่วโมง แต่หากเรานำข้อมูลลงกล่องไฮส์เนียร์ดไดร์ฟ (สมมุติว่าใช้เวลา เตรียม 2 ชั่วโมง) แล้วให้คนเดินส่งเอกสารนำไฮาร์ดไดร์ฟดังกล่าวไปส่งที่อีกหน่วยงานหนึ่ง คนเดินจะใช้เวลาประมาณ 15 นาที แต่บนข้อมูลได้เยื่อกว่ามาก จะพบว่าการสื่อสารผ่าน เคเบิลไลด์แก็วจะใช้เวลาตอบสนองต่อหน่วยข้อมูลสูงกว่า (ส่งตัวอักษรหนึ่งตัวไปถึงอย่าง รวดเร็ว) ในขณะที่คนเดินใช้เวลาตอบสนองช้ากว่า แต่ให้ปริมาณงานต่อหน่วยเวลาสูงกว่า

นอกจากนี้ หากวิเคราะห์การทำงานของซอฟต์แวร์ต่าง ๆ ภายในเครื่องคอมพิวเตอร์จะพบ ว่า เครื่องคอมพิวเตอร์ไม่ได้ใช้เวลาทั้งหมดไปกับการประมวลผล แต่เวลาบางส่วนอาจสูญเสียไปเนื่องจากการทำงานของไฮาร์ดแวร์ส่วนอื่น เช่น การอ่านข้อมูลจากหน่วยความจำ หรือ ระบบอินพุต/เอาต์พุต อีน ๆ ดังนั้น การวิเคราะห์เวลาในการตอบสนอง หรือ เวลาในการ ประมวลผล (execution time) จึงสามารถแบ่งย่อยออกได้เป็น

- CPU Time
- Elapse Time

CPU Time คือเวลาที่หน่วยประมวลผลกลางทำงานเพื่อประมวลผลโปรแกรมหนึ่ง โดยไม่นับรวมเวลาจากการรอคิวยการทำงานจากอุปกรณ์ต่อพ่วงที่อยู่ภายนอกหน่วยประมวลผล กลาง

Elapse time คือเวลาในการประมวลผลของโปรแกรมหนึ่งในระบบคอมพิวเตอร์ ซึ่งรวมถึง เวลาที่ใช้ในการอ่านข้อมูลจากอินพุต/เอาต์พุต (I/O Time) และ เวลาในการเข้าถึงหน่วยความจำ (memory time) ซึ่งสรุปเบื้องต้นได้ดังสมการที่ 2.4

$$\text{Elapse Time} = \text{CPU Time} + \text{I/O Time} + \text{Memory Time}$$

สมการที่ 2.4 Elapse Time

ทั้งนี้ในการวัดสมรรถนะของระบบคอมพิวเตอร์นั้น บางครั้งเรานำใจจะวัดเพียงเวลาที่ใช้ในหน่วยประมวลผลกลาง จึงวัดด้วย CPU time แทน ในกรณีที่สนใจเวลาที่ใช้ทั้งระบบ ก็จะใช้การวัดด้วย elapse time กรณีวัดสมรรถนะโดยใช้ elapse time ในการอ้างอิง ค่าของสมรรถนะที่วัดได้อาจแปรเปลี่ยนตามความเร็วของการเข้าถึงข้อมูลในหน่วยความจำ หรือความเร็วในการตอบสนองของอุปกรณ์ต่อพ่วงต่าง ๆ ทำให้บางครั้งไม่สามารถนำค่าที่วัดได้มาวิเคราะห์ในเชิงเปรียบเทียบ (จากสถิติพบว่า โปรแกรมส่วนใหญ่ใช้เวลามากกว่า 45% ในการทำงานงานของอินพุต/เอาต์พุต หรือส่วนอื่น ๆ ที่ไม่เกี่ยวข้องกับการทำงานของโปรแกรม)

2.3 เกณฑ์เปรียบเทียบสมรรถนะ (Benchmark)

เนื่องจากการวัดสมรรถนะของระบบคอมพิวเตอร์ ต้องอาศัยการวัดเวลาจากการทำงานจริงของโปรแกรม จึงจำเป็นจะต้องใช้โปรแกรมทดสอบที่เป็น เพื่อให้การวัดสามารถเปรียบเทียบได้ ทั้งนี้โปรแกรมที่จะใช้ทดสอบ อาจเป็นโปรแกรมใดก็ได้ แต่หากผู้พัฒนาแต่ละรายทำการทดสอบสมรรถนะโดยใช้โปรแกรมทดสอบของตนเอง ผลลัพธ์ที่ได้จากการทดสอบจะไม่สามารถเปรียบเทียบกันได้ หรือเปรียบเทียบกันได้ยาก

ด้วยเหตุนี้ การเลือกใช้เกณฑ์เปรียบเทียบสมรรถนะที่เหมาะสม จึงเป็นสิ่งสำคัญต่อการประเมินสมรรถนะ แนวทางในการเลือกใช้โปรแกรมสำหรับเป็นเกณฑ์เปรียบเทียบสมรรถนะ ที่เหมาะสม ได้แก่

- Real benchmark คือ การใช้โปรแกรมจริงซึ่งใช้ในการทำงานสำหรับการวัดสมรรถนะ เช่น โปรแกรมจัดงานเอกสาร word processing, คอมไฟเลอร์
- Microbenchmark คือ การใช้บางส่วนของโปรแกรมเพื่อวัดสมรรถนะ เช่น การคำนวณทางคณิตศาสตร์
- Kernel คือ การใช้โปรแกรมเฉพาะเพื่อทดสอบบางส่วนของชาร์ดแวร์
- Synthetic benchmark คือ การใช้โปรแกรมสังเคราะห์เพื่อสร้างการประมวลผลสมมุติ

- Toy benchmark คือ การใช้โปรแกรมพื้นฐานทางอัลกอริทึมในการทดสอบ เช่น การใช้โปรแกรม quick sort หรือ merge sort ในการทดสอบความเร็ว

ให้สังเกตว่า สมรรถนะที่วัดได้จะแตกต่างกันไปตามการเลือกเกณฑ์เปรียบเทียบสมรรถนะ ดังนี้ เพื่อให้การวัดสมรรถนะสามารถเปรียบเทียบกันได้โดยไม่มีการโน้มเอียงไปทางหนึ่ง กลุ่มผู้ผลิตระบบคอมพิวเตอร์จึงได้รวมตัวกันและกำหนดเกณฑ์เปรียบเทียบสมรรถนะ กลาง ที่เป็นที่ยอมรับสำหรับการทดสอบสมรรถนะขึ้น (เรียกว่า SPEC) รายละเอียดสามารถศึกษาได้ในเนื้อหาส่วนถัดไป

2.3.1 SPEC⁴ Benchmark Suites

ผู้ผลิตหน่วยประมวลผลกลางและระบบคอมพิวเตอร์ได้มีการรวมกลุ่มกันกำหนดมาตรฐานและ ชุดซอฟต์แวร์ทดสอบที่เหมาะสมสำหรับการวัดสมรรถนะของระบบคอมพิวเตอร์ เพื่อให้การทดสอบมีมาตรฐาน สามารถเปรียบเทียบกันได้ ชุดซอฟต์แวร์ทดสอบเหล่านี้ เรียกว่า SPEC Benchmark suites ซึ่งเผยแพร่และรวบรวมโดย System Performance Evaluation Cooperation ชุดซอฟต์แวร์ทดสอบแบ่งย่อยออกเป็นซอฟต์แวร์ทดสอบด้านต่าง ๆ เช่น SPECint สำหรับทดสอบการประมวลผลจำนวนเต็ม SPECweb สำหรับทดสอบระบบเบราว์เซอร์ เว็บ เป็นต้น

ตัวอย่างของชุดโปรแกรม ที่มีอยู่ใน SPEC ปัจจุบัน เช่น

- **SPEC CPU2017** สำหรับทดสอบสมรรถนะงานที่มีหน่วยประมวลผลกลางสูง ประกอบไปด้วยชุดโปรแกรมด้าน ปัญญาประดิษฐ์ อัลกอริทึม งานบีบอัดข้อมูล งานบีบอัดภาพ/วีดีโอ การประมวลผลเสียง การแปลงภาษาคอมพิวเตอร์ (คอมไฟเลอร์ และ อินเทอร์พรีเตอร์) เป็นต้น
- **SPECviewperf® 12.1** สำหรับทดสอบสมรรถนะของการงานทางด้านกราฟิกส์ ซึ่งสร้างขึ้นจากความร่วมมือของผู้ผลิตอุปกรณ์ประมวลผลด้านกราฟิกส์
- **SPECjbb2015** สำหรับทดสอบสมรรถนะของ Java Virtual Machine เป็นหลัก ซึ่งจาก Java เป็นภาษาคอมพิวเตอร์ที่ได้รับความนิยมสูงในงานองค์กร แต่ สมรรถนะของ Java ขึ้นอยู่กับความสามารถของ Java Virtual Machine เป็นหลัก ดังนั้น กลุ่มผู้พัฒนา Java Virtual Machine จึงกำหนดมาตรฐานสำหรับ การวัดสมรรถนะของ Java ขึ้น เพื่อเป็นแนวทางในการพัฒนาคุณภาพของ Java Virtual Machine

4 รายละเอียดเพิ่มเติมสามารถค้นคว้าได้จาก <http://www.spec.org/>

- **SPEC SFS2014** สำหรับทดสอบสมรรถนะในการอ่านเขียนข้อมูลจากเครื่องแม่ข่ายสำหรับจัดเก็บแฟ้มข้อมูล มีการวัดทั้งแบบเวลาตอบสนองและปริมาณงานที่ทำได้ต่อหน่วยเวลา

2.3.2 เกณฑ์เปรียบเทียบสมรรถนะที่ดี

เนื่องจากเกณฑ์เปรียบเทียบสมรรถนะถูกออกแบบเพื่อใช้เป็นตัวแทนในการวัดสมรรถนะของระบบคอมพิวเตอร์ ดังนั้น การเลือกว่าจะใช้โปรแกรมชุดใดเป็นเกณฑ์เปรียบเทียบสมรรถนะควรจะเลือกด้วยความระมัดระวัง ไม่เลือกโปรแกรมที่ส่งผลให้เกิดความเอียงในการประเมินสมรรถนะ ด้วยเงื่อนไขดังกล่าวเกณฑ์เปรียบเทียบจึงควรมีคุณสมบัติเบื้องต้นต่อไปนี้

1. เป็นโปรแกรมที่มีอยู่จริง และมีการใช้งานจริง ทั้งนี้หากนำโปรแกรมสังเคราะห์มาใช้ ให้ผลที่ได้อาจไม่สอดคล้องกับลักษณะการใช้งานจริง
2. สามารถนำมารวดได้บนฮาร์ดแวร์ที่กำหนด ในกรณีหากโปรแกรมเปรียบเทียบสมรรถนะที่เลือก ไม่สามารถนำมาแปลงและประมวลผลได้บนระบบที่ต้องการทดสอบ การทดสอบย่อมจะเกิดขึ้นไม่ได้
3. เป็นตัวแทนของงานในลักษณะคล้ายคลึงกัน และเป็นตัวแทนเทคโนโลยีที่เกี่ยวข้อง ตัวอย่างเช่น นักพัฒนาซอฟต์แวร์ทุกคน มักจะต้องทำการแปลงซอฟต์แวร์ด้วยคอมไฟเลอร์ ดังนั้นการมีตัวแทนของโปรแกรมที่จะถูกแปลงที่เหมาะสม จะช่วยให้การวัดสมรรถนะใกล้เคียงกับความเป็นจริงมากขึ้น
4. มีการคำนวณที่ชัดเจนคาดเดาได้ มีบางกรณีที่การทำงานบางอย่างของโปรแกรม ไม่สามารถคาดเดาได้ เช่น โปรแกรมมีการทำงานแบบสุ่ม ไม่สามารถประเมินจำนวนครั้งหรือรอบการประมวลผลที่แน่นอนได้ โปรแกรมในลักษณะนี้ จะทำให้การวัดผลแต่ละครั้งคลาดเคลื่อนสรุปการทำงานได้ยาก

อย่างไรก็ตาม นี้เป็นเพียงคุณสมบัติบางส่วนเท่านั้น และมีใช่ว่าเกณฑ์เปรียบเทียบสมรรถนะอันหนึ่งอันใด จะมีคุณสมบัติครบถ้วนข้อ เพราะในหลายกรณี ก็จะมีเกณฑ์เปรียบเทียบสมรรถนะที่เกิดจากการสังเคราะห์ เพื่อทดสอบฮาร์ดแวร์หรือการคำนวณแบบพิเศษเสมอ ซึ่งอาจไม่ใช้โปรแกรมที่มีการใช้งานจริงแต่อย่างใด

2.4 การเปรียบเทียบสมรรถนะ

ถึงตรงนี้ จะเห็นว่าการวัดสมรรถนะของระบบคอมพิวเตอร์ทำได้โดยการนำชุดโปรแกรมที่เหมาะสม (หรือที่ตกลงกันไว้ว่าจะใช้เป็นมาตรฐานในการวัด) มาทดสอบประมาณแล้วจับเวลา อย่างไรก็ตาม อาจมีความเป็นไปได้ว่า ผลที่ได้อาจจะไม่เป็นไปในทิศทางเดียวกันเสมอไป เช่น ในชุดโปรแกรมที่ทดสอบอาจจะมีโปรแกรมอยู่ 2 โปรแกรม ซึ่งระบบคอมพิวเตอร์หนึ่งอาจจะให้ผลบางค่าดีแต่บางค่าแย่กว่าอีกรอบหนึ่ง หากเป็นเช่นนี้ การจะสรุปว่าระบบคอมพิวเตอร์ใดมีสมรรถนะดีกว่าอย่างจะทำได้ยาก ดังตัวอย่างในตารางที่ 2.1

ตารางที่ 2.1: เปรียบเทียบเวลาได้จากเกณฑ์เปรียบเทียบสมรรถนะ

	คอมพิวเตอร์ A (วินาที)	คอมพิวเตอร์ B (วินาที)	คอมพิวเตอร์ C (วินาที)
โปรแกรม 1	10	4	5
โปรแกรม 2	2	10	5
ค่าเฉลี่ยคณิตศาสตร์	6	7	5

จากตาราง การจะตัดสินว่าคอมพิวเตอร์ระบบใดดีกว่าอย่างจะเป็นไปได้ยาก หากไม่มีข้อมูลเพิ่มเติม คำตอบที่ดูจะเหมาะสมจากค่าเฉลี่ยเลขคณิต⁵ (arithmetic mean) อาจเป็นคอมพิวเตอร์ C (เมื่อคิดว่าในระบบคอมพิวเตอร์มีการใช้งานโปรแกรม 1 และ โปรแกรม 2 ในสัดส่วนที่ใกล้เคียงกัน) แต่หากมีข้อมูลเพิ่มเติม เช่น หากการใช้งานส่วนใหญ่มักจะเป็นการใช้งานโปรแกรมที่ 2 ถึง 80% การเลือกใช้คอมพิวเตอร์ A ดูจะเป็นคำตอบที่ดีที่สุดตามตารางที่ 2.2 เป็นต้น ซึ่งการสรุปในลักษณะนี้ เป็นการใช้ ค่าเฉลี่ยเลขคณิตแบบถ่วงน้ำหนัก⁶ (weighted arithmetic mean) มาเป็นเครื่องมือช่วยในการตัดสินใจ

5 ค่าเฉลี่ยเลขคณิต (Arithmetric Mean) คำนวณได้จาก

$$\frac{\sum_{i=1}^n x_i}{n}$$

6 ค่าเฉลี่ยเลขคณิตแบบถ่วงน้ำหนัก (Weighted Arithmetic Mean) คำนวณได้จาก

$$\frac{\sum_{i=1}^n w_i x_i}{\sum_{i=1}^n w_i}$$

ตารางที่ 2.2: การใช้ค่าเฉลี่ยเลขคณิตถ่วงน้ำหนักเปรียบเทียบเวลา

	คอมพิวเตอร์ A (วินาที)	คอมพิวเตอร์ B (วินาที)	คอมพิวเตอร์ C (วินาที)
โปรแกรม 1	10	4	5
โปรแกรม 2	2	10	5
ค่าเฉลี่ยเลขคณิต แบบถ่วงน้ำหนัก	1.8	4.4	2.5

(เปรียบเทียบเมื่อโปรแกรม 2 มีการใช้งาน 80% และโปรแกรม 1 มีการใช้งาน 20%)

ในทางปฏิบัติ อาจไม่สามารถสรุปได้ว่ามีการใช้งานโปรแกรมใดในสัดส่วนเท่าไร และการบอกสมรรถนะด้วยหน่วยเวลา ยังไม่ใช่การบอกสมรรถนะในเชิงเปรียบเทียบที่เหมาะสมอีก เช่นกัน เพราะเวลาที่ข้อมูลกับปริมาณงาน (workload) ตัวอย่าง เหตุการณ์การเปรียบเทียบที่ถูกว่าดีอีก การเปรียบเทียบโดยใช้ Speedup (สมการที่ 2.3) ตัวอย่างการเปรียบเทียบแสดงได้ดัง ตารางที่ 2.3 ซึ่งการรายงานผลใช้เปรียบเทียบลักษณะนี้ จะเป็นการรายงานเปรียบเทียบ กับระบบอ้างอิง (baseline) เสมอ (กรณีตัวอย่างกำหนดให้คอมพิวเตอร์ A เป็นระบบ อ้างอิง)

อย่างไรก็ตาม ยังมีตัวเลขที่เกี่ยวข้องมากกว่าหนึ่งค่าในการเปรียบเทียบ เพราะมีโปรแกรมในชุดของเกณฑ์เปรียบเทียบสมรรถนะอยู่หลายโปรแกรมที่เกี่ยวข้อง จึงมีแนวคิดที่จะใช้ค่าเฉลี่ยในการสรุปผลแทนเพื่อให้การรายงานผลสะดวกขึ้น ทั้งนี้หากใช้ค่าเฉลี่ยเลขคณิต โดยตรง จะพบว่าค่าเฉลี่ยที่ได้จะไม่สอดคล้องกับข้อมูลที่แท้จริง (ดูตารางที่ 2.3) เพราะค่าเฉลี่ยเลขคณิต จะเหมาะสมสำหรับการหาค่าเฉลี่ยเฉพาะกรณีข้อมูลที่เปรียบเทียบมีหน่วยเดียวกัน จึงไม่เหมาะสมที่จะนำมาใช้กับข้อมูลที่เป็นอัตราส่วน (ซึ่งไม่มีหน่วย หรือหน่วยตัดกันไปแล้ว) จากตัวอย่างจะเห็นว่า หากนำ A/B มาหาค่าเฉลี่ยจะได้เป็น 1.35 ซึ่งไม่สอดคล้องกับค่าเฉลี่ย A หารด้วย ค่าเฉลี่ย B ซึ่งจะมีค่าเป็น 0.86 (คำนวณได้จาก 6 หารด้วย 7)

ตารางที่ 2.3: การเปรียบเทียบสมรรถนะด้วยอัตราส่วน พร้อมค่าเฉลี่ยเลขคณิต

	คอมพิวเตอร์ A (วินาที)	คอมพิวเตอร์ B (วินาที)	A/B	คอมพิวเตอร์ C (วินาที)	A/C
โปรแกรม 1	10	4	2.5	5	2
โปรแกรม 2	2	10	0.2	5	0.4
ค่าเฉลี่ยเลขคณิต	6	7	1.35	5	1.2

ด้วยเหตุนี้การรายงานผลที่ดีกว่าคือการใช้ค่าเฉลี่ยเรขาคณิต⁷ (geometric mean) ทั้งนี้เนื่องจากค่าเฉลี่ยเรขาคณิต จะให้ค่าที่เหมาะสมเมื่อข้อมูลอยู่ในลักษณะของอัตราส่วน (เช่น A/B ซึ่งไม่มีหน่วย หรือมีหน่วยเป็นเท่าๆ กัน) ตารางที่ 2.4 แสดงการใช้ค่าเฉลี่ยเรขาคณิตในการประมวลผล จากตัวอย่างจะเห็นได้ว่า การนำค่าเฉลี่ยเรขาคณิตมาเปรียบเทียบ (X/Y) ให้ผลลัพธ์เหมือนกับการนำเวลาไปเปรียบเทียบโดยตรง (C/B) ซึ่งค่าเฉลี่ยเรขาคณิตที่ได้จากการเปรียบเทียบก็มีค่าเท่ากันเช่นกัน

ตารางที่ 2.4: การเปรียบเทียบสมรรถนะด้วยอัตราส่วน พร้อมค่าเฉลี่ยเรขาคณิต

	A (วินาที)	B (วินาที)	A/B (X)	C (วินาที)	A/C (Y)	C/B	X/Y
โปรแกรม 1	10	4	2.5	5	2	1.25	1.25
โปรแกรม 2	2	10	0.2	5	0.4	0.5	0.5
ค่าเฉลี่ยเรขาคณิต	-	-	0.71	-	0.89	0.79	0.79

ข้อสังเกตคือ ในเรื่องของอัตราส่วนนั้น ค่าเฉลี่ยเลขคณิตของอัตราส่วน จะไม่เท่ากับอัตราส่วนของค่าเฉลี่ยเลขคณิต แต่ค่าเฉลี่ยเรขาคณิตของอัตราส่วน จะเท่ากับอัตราส่วนของค่าเฉลี่ยเรขาคณิต เราจึงใช้ค่าเฉลี่ยเรขาคณิตเมื่อต้องการเปรียบเทียบแบบสัดส่วน

⁷ ค่าเฉลี่ยเรขาคณิต (Geometric Mean) คำนวณได้จาก $\sqrt[n]{\prod^n x_i}$

การเปรียบด้วยค่าเฉลี่ยเรขาคณิตนี้ เป็นแนวทางมาตรฐานที่ใช้ในการรายงานผลสมรรถนะของ SPEC และเป็นวิธีการที่ใช้ทั่วไปเมื่ออ้างอิงถึงสมรรถนะ อย่างไรก็ตามในทางคณิตศาสตร์ยังมีการหาค่าเฉลี่ยแบบอื่นอีก เช่น ค่าเฉลี่ยชาร์มอนิก (harmonic mean) สำหรับการหาค่าเฉลี่ยกรณีที่ข้อมูลมีหน่วยเป็นอัตรา ซึ่งมีได้ก่อตัวถึงในที่นี้

2.5 CPU Time

ในทางปฏิบัตินั้น บางครั้งการวัดหรือจับเวลาการทำงานของโปรแกรมที่ใช้ในหน่วยประมวลผลกลาง ไม่สามารถทำได้โดยง่าย เนื่องจากบางครั้งเวลาที่วัดได้ดังกล่าว ไม่สามารถแยกได้ว่า เป็นเวลาที่ใช้ในหน่วยประมวลผลกลาง หรือ เป็นเวลาการทำงานที่ร dara กอain พุต/ เอาต์พุต ดังนั้นการหาเวลาที่ใช้ในหน่วยประมวลผลกลาง จึงอาจทำได้โดยการประมาณ หรือการนับจำนวนคำสั่งที่ใช้ในการประมวลผล และ หาผลรวมของเวลาที่ใช้ในการประมวลผลคำสั่งทั้งหมด ดังตัวอย่างที่ 2.2

จากตัวอย่างที่ 2.2 พบร้า เวลาที่ใช้ในการประมวลผลคำสั่งจะคำนวณได้จากผลคูณของจำนวนคำสั่งที่ทำงาน (instruction count) และเวลาเฉลี่ยในการทำงานของแต่ละคำสั่ง (average time per instruction) ในตัวอย่าง ทุกคำสั่งใช้เวลาเท่ากันคือ 10 นาโนวินาที ซึ่ง ในทางปฏิบัติ หากแต่ละคำสั่งใช้เวลาไม่เท่ากัน การประเมินความมีการถ่วงน้ำหนักเพื่อให้ได้เวลาเฉลี่ยที่ใกล้เคียงกับความเป็นจริงหรือทำการคำนวณแยกแล้วนำผลที่ได้มาบวกรวมกัน อีกครั้งหนึ่ง

ตัวอย่างที่ 2.2: การประเมิน CPU จากการคำนวณ

เครื่องคอมพิวเตอร์เครื่องหนึ่งมีเวลาในการประมวลผลคำสั่งแต่ละคำสั่งเป็น 10 นาโนวินาที หากนำโปรแกรมที่มีจำนวนคำสั่งที่ทำงาน 10 คำสั่งมาทำการประมวลผลบนเครื่องดังกล่าว จะใช้เวลาในการทำงานเป็นเท่าไร

$$\text{เวลาในการทำงานทั้งหมด} = 10 \times 10 = 100 \text{ นาโนวินาที}$$

อย่างไรก็ตาม การบอกเวลาในการประมวลผลของคำสั่งแต่ละคำสั่งนั้น ไม่นิยมบอกเป็นหน่วยเวลา (นาโนวินาที) ทั้งนี้เนื่องจากเวลาในการประมวลผลของหน่วยประมวลผลขึ้นอยู่กับความถี่สัญญาณนาฬิกา (clock rate) ที่ป้อนให้กับหน่วยประมวลผล ซึ่งห่วงเวลาที่นิยม

เรียกว่า ticks หรือ clock cycle time โดยเวลา 1 clock cycle คือเวลาหนึ่งรอบสัญญาณนาฬิกา การบอกเวลาเป็นหน่วยรอบมีข้อดีที่ทำให้การประเมินค่าไม่จำเป็นต้องยุ่งเกี่ยวกับความถี่สัญญาณนาฬิกาของระบบ ช่วยให้การนับหรือคำนวณsheduleมากขึ้น โดยเฉพาะเมื่อเครื่องที่ต้องการประเมินมีความถี่สัญญาณนาฬิกาเท่ากัน

ความสัมพันธ์ระหว่างความถี่สัญญาณนาฬิกาและรอบสัญญาณนาฬิกา (cycle time) สามารถแสดงได้ดังสมการที่ 2.5 ด้านล่างนี้ หน่วยประมาณผลที่มีความถี่สัญญาณนาฬิกาเป็น 2Ghz จะมีรอบสัญญาณนาฬิกาเป็น 0.5 นาโนวินาที ทั้งนี้ต้องยืนยันสมมุติฐานว่า ความถี่สัญญาณนาฬิกาจะถูกนำไปใช้เป็น 1 ควบเวลาของสัญญาณนาฬิกาในหน่วยประมาณผลโดยตรง⁸

$$\text{Cycle Time (seconds)} = \frac{1}{\text{Clock Rate (Hz)}}$$

สมการที่ 2.5 Cycle Time

นอกจากนี้การประมาณผลคำสั่งแต่ละคำสั่งอาจมีจำนวนรอบการประมาณผลต่อคำสั่ง (Cycle Per Instruction หรือ CPI) แตกต่างกันไป เช่นการประมาณผลคำสั่งบางอาจใช้เวลา 5 CPI และการประมาณผลคำสั่งคูณอาจใช้เวลา 8 CPI ดังนั้นหากหน่วยประมาณผลกลางมีความถี่สัญญาณนาฬิกาเป็น 2Ghz เวลาที่ใช้ในการทำคำสั่งบางจะเป็น 2.5 นาโนวินาที (5×0.5) และเวลาที่ใช้ในการทำคำสั่งคูณเท่ากับ 4.0 นาโนวินาที (8×0.5)

เนื่องจากคำสั่งใช้จำนวนรอบในการประมาณผลไม่เท่ากัน นักวิเคราะห์จึงแบ่งกลุ่มคำสั่ง (instruction class) โดยจัดคำสั่งที่มีจำนวนรอบการประมาณผลเท่ากันไว้ด้วยกัน เมื่อนำสถิติการกระจายตัวของคำสั่งในโปรแกรมเข้ามาถ่วงน้ำหนักผนวกเข้ากับการวิเคราะห์ด้วยความสามารถหาค่าเฉลี่ยจำนวนรอบการประมาณผลต่อคำสั่ง (average CPI) ได้จากการที่ 2.6 ดูตัวอย่างการหาค่าเฉลี่ยของจำนวนรอบการประมาณผลต่อคำสั่งในตัวอย่างที่ 2.3

$$\text{CPI}_{\text{average}} = \sum_{i=1}^n \text{CPI}_i \times \text{Weight}_i$$

สมการที่ 2.6 average CPI

8 ในอดีตหน่วยประมาณผลบางรุ่นจะนำความถี่สัญญาณนาฬิกามาผ่านวงจรหารความถี่หรือวงจรสร้างเฟสก่อน ทำให้มีกรณีที่หนึ่งความถี่สัญญาณนาฬิกามาไม่เท่ากับหนึ่งควบเวลาของสัญญาณนาฬิกา

ตัวอย่างที่ 2.3: การหาค่าเฉลี่ยของจำนวนรอบที่ใช้ในการประมวลผล

สถาบันกรรมคอมพิวเตอร์สมมุติ ประกอบด้วยกลุ่มคำสั่งการประมวลผลทางคณิตศาสตร์และตรรกะ (class A) กลุ่มคำสั่งควบคุม (class B) และกลุ่มคำสั่งการอ้างอิงหน่วยความจำ (class C) หากกลุ่มคำสั่งการประมวลผลทางคณิตศาสตร์และตรรกะมีจำนวนรอบการประมวลผลของคำสั่งเป็น 5 รอบต่อคำสั่ง กลุ่มคำสั่งควบคุมมีจำนวนรอบการประมวลผล 3 รอบต่อคำสั่ง และกลุ่มคำสั่งการอ้างอิงหน่วยความจำมีจำนวนรอบการประมวลผล 8 รอบต่อคำสั่ง หากโปรแกรมโดยทั่วไปมีการใช้งานคำสั่งทั้งสามกลุ่มเป็น 30% 50% และ 20% ตามลำดับ จงหาค่าเฉลี่ยของจำนวนรอบการประมวลผลต่อคำสั่งของสถาบันกรรมคอมพิวเตอร์สมมุติดังกล่าว

	CPI	สัดส่วนการใช้งาน
class A	5	30%
class B	3	50%
class C	8	20%

$$CPI_{average} = (5 \times 0.3) + (3 \times 0.5) + (8 \times 0.2)$$

$$\therefore CPI_{average} = 4.6$$

ต่อมาท้าไปอ้างถึงจำนวนรอบการประมวลผลต่อคำสั่ง (CPI) โดยมีหมายความถึง ค่าเฉลี่ยจำนวนรอบการประมวลผลต่อคำสั่ง (average CPI) ดังนั้นหากพบรหัส CPI ในสมการทั่วไปขอให้เข้าใจว่าคือค่าเฉลี่ยที่ได้จากการวิเคราะห์มาก่อนแล้ว

เมื่อนำแนวความคิดทั้งหมดมาประสานเข้าด้วยกัน สามารถสรุปเป็นสมการแสดงความสัมพันธ์ระหว่างเวลาที่ใช้ในการประมวลผล และ ความถี่สัญญาณนาฬิกาได้ดัง สมการที่ 2.7 สมการที่ 2.8 และ สมการที่ 2.9 ตามลำดับ ซึ่งเมื่อนำสมการที่ 2.7 และสมการที่ 2.9 รวมกันจะได้เป็นสมการที่ 2.10

$$CPU\ Time\ per\ instruction = CPI_{average} \times Cycle\ Time$$

สมการที่ 2.7 CPU Time/instruction

หรือ
$$\frac{\text{Seconds}}{\text{Instruction}} = \left(\frac{[\text{Clock Cycle}]}{\text{Instruction}} \right) \times (\text{Seconds per Cycle})$$

สมการที่ 2.8 Seconds/Instruction

ดังนั้นเวลาในการทำงานของ CPU (CPU Time) สำหรับ 1 โปรแกรม จะเป็น

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPU Time per Instruction}$$

สมการที่ 2.9 CPU Time เมื่อทราบเวลาต่อคำสั่ง

หรือ
$$\text{CPU Time} = \text{Instruction} \times \text{CPI}_{\text{average}} \times \text{Cycle Time}$$

สมการที่ 2.10 CPU Time เมื่อทราบ Cycle ต่อคำสั่ง

จะเห็นว่าเวลาในการประมวลผลสามารถหาได้จากการนับจำนวนคำสั่งและการคำนวณเบื้องต้น ตัวอย่างที่ 2.4 แสดงการหาเวลาในการทำงานของ CPU ด้วยการคำนวณจากจำนวนคำสั่ง ความถี่สัญญาณนาฬิกา และจำนวนรอบการประมวลผลต่อคำสั่ง

ตัวอย่างที่ 2.4: การคำนวณเวลาในการทำงานของ CPU

เครื่องคอมพิวเตอร์เครื่องหนึ่งมีความถี่สัญญาณนาฬิกา 1Ghz ใช้เวลาในการประมวลผลโปรแกรมทดสอบซึ่งทำจำนวนคำสั่งทั้งสิ้น 2000 ล้านคำสั่ง เป็นเวลาทั้งสิ้น 20 วินาที หากผู้ออกแบบสามารถพัฒนาให้หน่วยประมวลผลกลางใช้ความถี่สัญญาณนาฬิกาเป็น 2Ghz แต่ใช้ค่าเฉลี่ยจำนวนรอบการประมวลผลต่อคำสั่งมากขึ้นเป็น 1.2 เท่า เมื่อนำโปรแกรมนี้มาทดสอบในหน่วยประมวลผลที่ทำการพัฒนาใหม่ จะใช้เวลาในการประมวลผลเป็นเท่าไร

ความถี่สัญญาณนาฬิกา 1Ghz คิดเป็น

$$\text{Clock Cycle Time} = \frac{1}{(1 \times 10^9)} \text{ seconds}$$

จาก
$$\text{CPU Time (seconds)} = \text{Instruction} \times \text{CPI}_{\text{average}} \times \text{Cycle Time}$$

$$\text{CPI}_{\text{เดิม}} = \frac{(20 \text{ seconds}) \times (1 \times 10^9 \text{ seconds per cycle})}{(2000 \times 10^6 \text{ instructions})}$$

$$CPI_{\text{เดิม}} = 10$$

การพัฒนาหน่วยประมวลผล ทำให้ใช้ CPI เพิ่มขึ้นเป็น 1.2 เท่า

$$CPI_{\text{ใหม่}} = 10 \times 1.2 = 12$$

เวลาที่ใช้ในการประมวลผลโปรแกรมทดสอบหลังจากพัฒนาหน่วยประมวลผลกลาง

$$CPU TIME_{\text{ใหม่}} = \frac{(2000 \times 10^6 \times 1.2 \times 10)}{(2 \times 10^9)} \text{ seconds}$$

$$\therefore CPU TIME_{\text{ใหม่}} = 12 \text{ วินาที}$$

ในบางกรณีผู้ผลิตระบบคอมพิวเตอร์ (โดยเฉพาะเครื่องคอมพิวเตอร์สมรรถนะสูงในอดีต) นิยมที่จะกล่าวถึงความเร็วในการประมวลผลทั่วไป โดยใช้หน่วย MIPS⁹ (Millions of Instructions per second) เช่น โปรแกรมมีการทำงานทั้งสิ้น 2000 ล้านคำสั่งภายในเวลา 20 วินาทีก็มีความเร็วในการประมวลผลเป็น $2000/20 = 100$ MIPS เป็นต้น ซึ่งมีความหมายว่า หน่วยประมวลผลดังกล่าวสามารถประมวลผลได้ 100 ล้านคำสั่งภายในเวลา 1 วินาที

ในเบื้องต้นการบอกความเร็วด้วยหน่วย MIPS อาจดูสอดคล้อง (เป็นส่วนกลับของ) กับการใช้จำนวนรอบการประมวลผลต่อคำสั่งและรอบสัญญาณไฟกາ อย่างไรก็ตามหากวิเคราะห์ให้ละเอียดจะพบว่า ค่า MIPS อาจมีได้แสดงถึงสมรรถนะที่แท้จริงของระบบแต่อย่างใด เพราะค่า MIPS มีได้กล่าวถึงจำนวนคำสั่งที่เกี่ยวข้องดังแสดงตัวอย่างที่ 2.5

⁹ นอกจากหน่วย MIPS (Millions of Instructions per second) แล้ว ยังมีหน่วยที่เทียบเคียงกัน เช่น GIPS (Giga instructions per second) หรือ MOPS (Million Operations per second)

ตัวอย่างที่ 2.5: การเปรียบเทียบสมรรถนะในหน่วย MIPS

กำหนดให้ เครื่องคอมพิวเตอร์ A มีความเร็ว 10 MIPS ทำงานที่ 33 MHz และ เครื่องคอมพิวเตอร์ B มีความเร็ว 9 MIPS ทำงานที่ 20 MHz หากนำโปรแกรมทดสอบมาทำการแปลงเพื่อทดสอบบนเครื่อง A ได้จำนวนคำสั่งเป็น 12,000,000 คำสั่ง และบนเครื่อง B ได้ 10,000,000 คำสั่ง เครื่องคอมพิวเตอร์เครื่องใดมีสมรรถนะสูงกว่ากัน เมื่อเปรียบเทียบด้วยการทำงานของโปรแกรมทดสอบที่กำหนด

$$\text{จากคำจำกัดความของ MIPS จะได้ว่า } n \text{ MIPS} = \frac{n \times 10^6 (\text{instructions})}{1 (\text{second})}$$

ส่วนกลับของผลคุณของ MIPS (n) กับรอบสัญญาณนาฬิกา จะผลลัพธ์เป็นรอบการประมวลผลต่อคำสั่ง

MIPS แทนจำนวนคำสั่งใน 1 วินาที ดังนั้น ผลคุณของ MIPS และรอบสัญญาณนาฬิกาจึงได้จำนวน (ล้าน) คำสั่งใน 1 รอบสัญญาณนาฬิกา

$$CPI(\text{cycle}) = \frac{1}{\text{Number of Instructions per cycle}}$$

$$CPI(\text{cycle}) = \frac{1}{n \times 10^6 \times \text{Cycle Time}(\text{seconds})}$$

นำค่าที่ได้มาคำนวนหา CPU Time จะได้ว่า

เครื่องคอมพิวเตอร์ A

$$\text{Cycle Time}_A(\text{seconds}) = \frac{1}{\text{Clock Rate}} = \frac{1}{33 \times 10^6 (\text{Hz})}$$

$$CPU Time_A = \text{Instruction Count}_A \times CPI_A \times \text{Cycle Time}_A$$

$$CPU Time_A = 12 \times 10^6 \times \left(\frac{1}{10 \times 10^6 \times \frac{1}{33 \times 10^6}} \right) \times \frac{1}{33 \times 10^6} (\text{seconds})$$

$$CPU Time_A = \frac{12 \times 10^6}{10 \times 10^6} = 1.2 (\text{seconds})$$

เครื่องคอมพิวเตอร์ B

$$\text{Cycle Time}_B (\text{seconds}) = \frac{1}{\text{Clock Rate}} = \frac{1}{20 \times 10^6 (\text{Hz})}$$

$$\text{CPU Time}_B = 10 \times 10^6 \times \left(\frac{1}{9 \times 10^6 \times \frac{1}{20 \times 10^6}} \right) \times \frac{1}{20 \times 10^6} (\text{seconds})$$

$$\text{CPU Time}_B = \frac{10 \times 10^6}{9 \times 10^6} = 1.11 (\text{seconds})$$

จากตัวอย่างดังกล่าว จะได้ว่า เครื่องคอมพิวเตอร์ A ใช้เวลาในการประมวลผล 1.2 วินาที ในขณะที่เครื่องคอมพิวเตอร์ B ใช้เวลาในการประมวลผล 1.11 วินาที

∴ เครื่องคอมพิวเตอร์ B มีสมรรถนะสูงกว่าเครื่องคอมพิวเตอร์ A

หากจะสรุปต่อว่าเครื่องคอมพิวเตอร์ B เเร็วกว่าเครื่องคอมพิวเตอร์ A กี่เท่า สามารถทำได้โดยนำสมการที่ 2.3 มาประยุกต์ใช้ ดังนี้

$$\text{Speedup} = \frac{1.2 (\text{seconds})}{1.1 (\text{seconds})} = 1.09$$

จะเห็นได้ว่าค่า MIPS ที่สูงอาจมีได้หมายถึงสมรรถนะที่สูง เพราะสมรรถนะต้องมาจากการใช้ในการประมวลผลคำสั่งที่เกี่ยวข้องทั้งหมดในขณะที่ค่า MIPS จะบอกเพียงจำนวนคำสั่งที่ทำได้ในหน่วยเวลาที่กำหนด โดยมีได้กล่าวถึงจำนวนคำสั่งที่เกี่ยวข้องแต่อย่างใด

2.6 กฎของ Amdahl¹⁰

ในการปรับปรุงสมรรถนะส่วนใดส่วนหนึ่งของระบบคอมพิวเตอร์ มีได้หมายความว่าเมื่อนำโปรแกรมทดสอบเข้าไปทดลองประมวลผลจะได้ความเร็วตามจำนวนเท่าของการปรับ เช่น การเพิ่มสมรรถนะในการบากเลขของหน่วยประมวลผลกลางให้เร็วขึ้น 2 เท่า มีได้หมายความว่า โปรแกรมทดสอบจะได้สมรรถนะเพิ่มขึ้นเป็น 2 เท่า ทั้งนี้เนื่องจากเวลาใน

10 กฎของ Amdahl คิดคันโดย Gene M. Amdahl อธิบดีผู้ทำงานด้านระบบ Mainframe ของ IBM ซึ่งต่อมาได้ก่อตั้งบริษัท Amdahl Corporation (ปัจจุบันเป็นส่วนหนึ่งของ บริษัท Fujitsu) กฎของ Amdahl มีความสำคัญเป็นมากฐานของการพัฒนาระบบสถาปัตยกรรมคอมพิวเตอร์ในปัจจุบัน ที่มา: บางส่วนคัดมาจากการอ่านจาก wikipedia [3] (สืบคันข้อมูลเมื่อ มกราคม พ.ศ. 2557)

การประมวลผลทั้งหมด ไม่ใช่เวลาที่เกิดขึ้นจากการบวกเลข หากแต่เป็นเวลารวมของการบวกเลข การคูณเลข การร้ายข้อมูล หรือการเปรียบเทียบทางตระกูลทั้งหมดที่เกี่ยวข้อง จึงเกิดแนวคิด (กฎ) ในการวัดค่า Speedup เพื่อเป็นการประเมินแนวทางการปรับปรุงสมรรถนะของระบบคอมพิวเตอร์โดยรวมเมื่อมีการปรับปรุงสมรรถนะของส่วนย่อยบางส่วน กฎที่ช่วยในการประเมินสมรรถนะดังกล่าวนี้ว่า กฎของ Amdahl

กฎของ Amdahl (บางตำราเรียกว่า law of diminishing return) ได้กำหนดการวัด Speedup หรือสมรรถนะที่เพิ่มขึ้นว่า หากมีการพัฒนาให้ส่วนหนึ่งส่วนใดของระบบมีความสมรรถนะมากขึ้น ผลของสมรรถนะที่เพิ่มขึ้นโดยรวมจากการปรับปรุงส่วนหนึ่งส่วนใดนั้น จะเป็นสัดส่วนของส่วนที่มิได้รับการปรับปรุง รวมกับส่วนที่ได้รับการปรับปรุง

ข้อความนี้เข้าใจได้ยาก แต่หากลองพิจารณาดูสมการที่ 2.3 จะพบว่า

$$\text{Overall Speedup} = \frac{\text{Execution Time}_{\text{old}}}{\text{Execution Time}_{\text{new}}}$$

ทั้งนี้เวลาที่ใช้ในการประมวลผลโปรแกรมทดสอบหลังจากที่ได้ทำการพัฒนา หาได้จากความสัมพันธ์ของสัดส่วนเวลาที่เกี่ยวข้องดังแสดงในสมการที่ 2.11 กล่าวคือ เวลาที่ใช้ในการประมวลผลหลังปรับปรุงสมรรถนะ (Execution Time_{new}) คิดเป็น ผลรวมของเวลาประมวลผลส่วนที่ไม่ได้รับการปรับปรุง (Execution Time_{no enhancement}) กับ เวลาส่วนที่ได้รับการปรับปรุง (Execution Time_{enhancement}) หารด้วยอัตราส่วนที่ได้รับการปรับปรุง (enhanced ratio)

$$\text{Execution Time}_{\text{new}} = \text{Execution Time}_{\text{no enhancement}} + \frac{\text{Execution Time}_{\text{enhancement}}}{\text{enhanced ratio}}$$

สมการที่ 2.11 Execution Time

หากวิเคราะห์ต่อไป จะสังเกตพบว่าผลรวมของเวลาประมวลผลส่วนที่ไม่ได้รับการปรับปรุง กับเวลาส่วนที่ได้รับการปรับปรุง ความจริงก็คือเวลาที่ใช้ในการประมวลผลก่อนปรับปรุง สมรรถนะ สมมุติให้ส่วนที่ได้รับการปรับปรุงมีสัดส่วนเป็น P และ อัตราส่วนที่ได้รับการปรับปรุง (enhanced ratio) มีค่าเป็น n สมการที่ 2.3 และ สมการที่ 2.11 จะเขียนรวมกันใหม่ได้เป็นกฎของ Amdahl ดังสมการที่ 2.12

$$\text{Overall Speedup} = \frac{\text{Execution Time}_{old}}{(1-P) \times \text{Execution Time}_{old} + \frac{P \times \text{Execution Time}_{old}}{n}}$$

$$\text{Overall Speedup} = \frac{1}{(1-P) + \frac{P}{n}}$$

สมการที่ 2.12 Overall Speedup

ตัวอย่างที่ 2.6 แสดงการคำนวณค่า Speedup ตามแนวทางของ Amdahl

ตัวอย่างที่ 2.6: การคำนวณ Speedup ด้วยกฎของ Amdahl

ในการชื้ออาหารเย็นทันนั้น จะต้องเสียเวลาไปกับการเดินทางไปและกลับ 10 นาที และเสียเวลาไปกับการเลือกซื้ออาหาร และชำระเงินในร้านอาหารอีก 5 นาที หากผู้ซื้อสามารถเดินทางได้เร็วขึ้น 2 เท่า ค่า Speedup หรือ สมรรถนะที่เพิ่มขึ้นโดยรวมจะเป็นเท่าใด

เวลาที่ใช้ในการซื้อของแต่เดิม คือ 15 นาที

เวลาที่ใช้ในการซื้อของเมื่odeinเร็วขึ้น 2 เท่า คือ $5 + (10/2) = 10$ นาที

$$\text{Overall Speedup} = \frac{\text{Execution Time}_{old}}{\text{Execution Time}_{new}} = \frac{15}{10} = 1.5$$

จากตัวอย่างให้สังเกตว่า แม้จะเดินเร็วขึ้นถึง 2 เท่าแต่สมรรถนะที่ได้โดยรวมเป็นเพียงแค่ 1.5 เท่า นี้คือเหตุที่กฎของ Amdahl เรียกว่า law of diminishing return กล่าวคือ การปรับปรุงให้บางอย่างเร็วขึ้น มิได้ให้ผลที่เร็วขึ้นเท่ากับแรงที่ออกเสนอไป ลองดูตัวอย่างที่ 2.7 เพิ่มเติม ซึ่งเป็นการปรับปรุงสมรรถนะในทำนองเดียวกับตัวอย่างที่ 2.6 แต่เป็นการปรับปรุงเวลาในการซื้ออาหารและชำระเงินให้เร็วขึ้นเป็น 3 เท่าแทน

ตัวอย่างที่ 2.7: การประยุกต์ใช้กฎของ Amdahl เพื่อเป็นแนวทางในการปรับปรุงสมรรถนะ

ในการซื้ออาหารเย็นทานนั้น ประกอบไปด้วยเวลาในการเดินทางไปกลับ และ เวลาในการเลือกซื้ออาหารและชำระเงิน หากเวลาในการเลือกซื้ออาหารและชำระเงินเป็น 1 ใน 3 ของเวลาทั้งหมด จงหาค่า Speedup หากผู้ซื้อสามารถเลือกซื้ออาหารและชำระเงินได้เร็วขึ้น 3 เท่า

$$\text{Overall Speedup} = \frac{1}{\left(1 - \left(\frac{1}{3}\right)\right) + \left(\frac{3}{3}\right)} = \frac{1}{\frac{2}{3} + \frac{1}{9}} = \frac{1}{\frac{6+1}{9}} = \frac{9}{7} \approx 1.29$$

เปรียบเทียบตัวอย่างที่ 2.6 และตัวอย่างที่ 2.7 พบว่า เวลาส่วนใหญ่เสียไปกับการเดินมากกว่าการเลือกซื้ออาหารและชำระเงิน ดังนั้นหากสามารถเดินได้เร็วขึ้น 2 เท่า ย่อมให้ผลลัพธ์โดยรวมดีกว่าการเลือกซื้ออาหารและชำระเงินเร็วขึ้น 3 เท่า

หลักการเช่นนี้ ถูกนำมาใช้ในการปรับปรุงสมรรถนะของระบบคอมพิวเตอร์ เช่นกัน กล่าวคือ หากต้องเลือกปรับปรุงให้ส่วนใดส่วนหนึ่งของระบบเร็วขึ้น เราคาครเลือกส่วนที่มีการใช้งานเป็นปริมาณมาก เพื่อให้ได้สมรรถนะโดยรวมดีที่สุดแทนการเลือกส่วนที่ใช้งานน้อย

ประเด็นช่วยจำ ความจริงแล้วกฎของ Amdahl ก็มีมาจากการเทียบเวลา execution time ตามปกติ ดังนั้นหากสามารถหาที่ใช้ในการประมวลผล (สมการที่ 2.11) แล้วนำมาเทียบด้วยสมการ Speedup (สมการที่ 2.3) ตามปกติ ก็จะได้ผลเหมือนกับการคำนวณด้วยสมการ overall Speedup (สมการที่ 2.12) ตามกฎเช่นกัน

2.7 แนวทางการปรับปรุงสมรรถนะของระบบคอมพิวเตอร์

จากสมการที่ 2.3 จะเห็นว่า การปรับปรุงเวลาในการทำงานของหน่วยประมวลผลลงทำได้โดยการลดค่าจำนวนรอบการประมวลผลต่อคำสั่ง เพิ่มความถี่สัญญาณนาฬิกา หรือลดจำนวนคำสั่งลง ดังนี้

- ความถี่สัญญาณนาฬิกาขึ้นอยู่กับโครงสร้างภายในชาร์ดแวร์ และเทคโนโลยีในการผลิต transistor¹¹

¹¹ ในปี 2014 เทคโนโลยีล่าสุดในการผลิตสารกึ่งตัวนำกือที่ขนาด 14 นาโนเมตร ในปี 2004 เทคโนโลยีการผลิตอยู่ที่ 90 นาโนเมตร ในทางปฏิบัติเทคโนโลยีที่เล็กลง ทำให้ transistor มีขนาดเล็กลง ซึ่งมัก

- จำนวนรอบการประมวลผลต่อคำสั่ง ขึ้นอยู่กับโครงสร้างภายในอาร์ดแวร์ และโครงสร้างสถาปัตยกรรมชุดคำสั่ง
- จำนวนคำสั่ง ขึ้นอยู่กับโครงสร้างสถาปัตยกรรมชุดคำสั่ง ความสามารถในการแปลภาษา และ การหาค่าเหมาะสมที่สุดของตัวแปลภาษา

2.8 สรุป

พื้นฐานในการวิเคราะห์และวัดสมรรถนะของเครื่องคอมพิวเตอร์นั้น เป็นสิ่งสำคัญในการศึกษาระบบสถาปัตยกรรมคอมพิวเตอร์ ทั้งนี้เนื่องจาก เมื่อมีการออกแบบหรือปรับปรุงระบบคอมพิวเตอร์แล้ว จะต้องสามารถพิสูจน์ หรือ วิเคราะห์ได้ว่า การปรับปรุงดังกล่าวในส่วนใด ของสถาปัตยกรรมคอมพิวเตอร์ ให้ดีขึ้นได้อย่างไร ในกรณีที่เป็นผู้ใช้หรือเลือกซื้อผลิตภัณฑ์ทางคอมพิวเตอร์ ผู้ซื้อควรมีความรู้ความเข้าใจเมื่ออ่านคำแนะนำสินค้านั้น ทราบถึงข้อดีข้อเสีย วิธีทดสอบ โดยไม่หลงคล้อยตามกับคำโฆษณาชวนเชื่อ เพื่อให้สามารถเลือกใช้ระบบคอมพิวเตอร์ที่เหมาะสมกับงานของตน หน่วยประมวลผลกลางบางตัวนั้น อาจมีสมรรถนะดี เมื่อประมวลผลทางคณิตศาสตร์แบบจำนวนเต็ม แต่อาจมีสมรรถนะแย่เมื่อประมวลผลเลขคณิต รวมถึงกราฟิกส์ (graphics)

การวิเคราะห์สมรรถนะในเชิงปฏิบัตินั้น นอกจากรายงานทางคณิตศาสตร์ที่มีอยู่แล้ว ยังต้องอาศัยความรู้ทางสถิติเพื่อประกอบการวิเคราะห์ด้วย เช่น การหาค่าเฉลี่ยเลขคณิต และการหาค่าเฉลี่ยเรขาคณิต วิธีการรายงานและการเบรี่ยงเทียบผล

2.9 แบบฝึกหัดท้ายบท

1. เราชัดสมรรถนะของเครื่องคอมพิวเตอร์ได้อย่างไร
2. ท่านคิดว่า เหตุใดเราจึงเลือกใช้ CPU TIME ในการวัดสมรรถนะของระบบคอมพิวเตอร์
3. ปัจจัยใดบ้างส่งผลกระทบต่อสมรรถนะของเครื่องคอมพิวเตอร์
4. เราสามารถเพิ่มสมรรถนะของเครื่องคอมพิวเตอร์ได้อย่างไร
5. การเปลี่ยน CPU ใหม่ที่มีความเร็วสูงขึ้นใน จัดเป็นการลดเวลาตอบสนอง หรือ การเพิ่มปริมาณงานที่ทำได้ต่อหน่วยเวลา
6. ความเร็วของสายสัญญาณเครือข่าย 100 Mbps มีผลต่อเวลาตอบสนอง หรือ ปริมาณงานที่ทำได้ต่อหน่วยเวลา จงให้เหตุผลประกอบการอธิบาย
7. แม้สัญญาณเครือข่ายจะมีความเร็วสูงถึง 1Gbps อย่างไรก็ตามการขนส่งข้อมูลขนาดใหญ่ที่มีสมรรถนะที่สุดคือการนำหน่วยเก็บข้อมูลเข่น Hard Drive จำนวนมากใส่รับบรรทุกหรือบนส่งทางอากาศ กล่าวคือ การขนส่งดังกล่าว 1 ครั้ง อาจจะ ขนส่งข้อมูลได้มากกว่าระบบเครือข่ายที่มีความเร็วที่สุดในปัจจุบันหลายล้านเท่า จัดว่าการขนส่งดังกล่าวให้ปริมาณงานที่ทำได้ต่อหน่วยเวลาสูง ใช่หรือไม่ เพราะเหตุใด
8. จงยกตัวอย่างคุณสมบัติของเกณฑ์เปรียบเทียบสมรรถนะมา 4 ข้อ
9. จงเลือกชาร์ดแวร์ที่เหมาะสมสำหรับการทำ Java Application Server คำแนะนำ ดู SPEC.org (<http://www.spec.org/>)
10. ในระบบคอมพิวเตอร์หนึ่ง มีการประมวลผลที่เกี่ยวข้องกับการคูณคิดเป็น 15% ของการประมวลผลทั้งหมด ทีมวิศวกรของระบบคอมพิวเตอร์นี้ มีความเห็นเกี่ยวกับการปรับปรุงสมรรถนะของระบบคอมพิวเตอร์ดังกล่าว แต่ออกเป็น 2 กลุ่ม กลุ่มแรกเห็นว่า ควรจะปรับปรุงวงจรคูณให้มีความเร็วสูงขึ้นสักเท่า กลุ่มที่สองเห็นว่าไม่ควรปรับปรุงวงจรได้ ๆ แต่ควรไปปรับปรุงคอมโพลิเยอร์ให้ลดการใช้การคูณลง หากท่านเป็นหัวหน้าวิศวกรคอมพิวเตอร์ของระบบดังกล่าว ท่านจะเลือกลงทุนให้วิศวกรกลุ่มไหนดำเนินงาน เพราะเหตุใด จงอธิบาย

11. คอมไพล์เลอร์ 2 ชุด ทำการแปลโปรแกรมเป็นภาษาเครื่อง ได้ผลเป็นชุดคำสั่งในกลุ่ม A, B และ C ซึ่งคำสั่งในแต่ละกลุ่มมีค่า CPI เป็น 1, 2, 3 ตามลำดับ โปรแกรมที่ได้มีลักษณะดังนี้

- โปรแกรมที่ได้จากคอมไпал์เลอร์ชุดที่ 1 มีคำสั่งอยู่ในกลุ่ม A 5 ล้านคำสั่ง กลุ่ม B 1 ล้านคำสั่ง และ กลุ่ม C 1 ล้านคำสั่ง
- โปรแกรมที่ได้จากคอมไpal์เลอร์ชุดที่ 2 มีคำสั่งอยู่ในกลุ่ม A 6 ล้านคำสั่ง กลุ่ม B 2 ล้านคำสั่ง และ กลุ่ม C 1 ล้านคำสั่ง

โปรแกรมแต่ละชุดใช้เวลาเท่าใด โปรแกรมชุดใดมีสมรรถนะสูงกว่ากัน หากเปรียบเทียบในหน่วย MIPS โปรแกรมชุดใดจะให้ค่า MIPS สูงกว่า เพราะเหตุใด

12. จากสมการที่ 2.3 และ คุณลักษณะต่าง ๆ ของระบบคอมพิวเตอร์ (โปรแกรม คอมไpal์เลอร์ สถาปัตยกรรมชุดคำสั่ง โครงสร้าง เทคโนโลยี) ท่านคิดว่า คุณลักษณะเหล่านี้ มีผลกระทบต่อสมรรถนะของหน่วยประมวลผลกลางในส่วนใด (จำนวนคำสั่ง รอบการประมวลผลต่อคำสั่ง รอบสัญญาณนาฬิกา) จงอธิบาย
13. เหตุใดการประเมินสมรรถนะจึงนิยมใช้ค่าเฉลี่ยเรขาคณิตในการประเมิน จงให้เหตุผลประกอบคำอธิบาย
14. หากชุดโปรแกรมทดสอบมีการทำงานของคำสั่งที่เกี่ยวกับหน่วยความจำซึ่งมีจำนวนรอบการประมวลผลต่อคำสั่งเป็น 5 อยู่ 40% และคำสั่งที่เกี่ยวกับการ Branch ซึ่งมีจำนวนรอบการประมวลผลต่อคำสั่งเป็น 3 อยู่ 20% ที่เหลือเป็นคำสั่งเกี่ยวกับ การประมวลผลทางคณิตศาสตร์ ซึ่งมีจำนวนรอบการประมวลผลต่อคำสั่งเป็น 4 ค่าเฉลี่ยรอบการประมวลผลต่อคำสั่งของโปรแกรมทดสอบดังกล่าวจะเป็นเท่าใด

3 สถาปัตยกรรมชุดคำสั่ง

สถาปัตยกรรมชุดคำสั่ง คือสถาปัตยกรรมของภาษาเครื่องที่ใช้สำหรับหน่วยประมวลผล กลาง ทั้งนี้ภาษาเครื่องเป็นภาษาคอมพิวเตอร์ที่มีลักษณะแตกต่างจากภาษาคอมพิวเตอร์ชั้น สูงโดยทั่วไป มักไม่มีคำสั่งการควบคุมที่ซับซ้อน (เช่น while, for, loop) ในหนังสือเล่มนี้มุ่ง ประเด็นสนใจที่สถาปัตยกรรมชุดคำสั่งของ nanoLADA ซึ่งเป็นสถาปัตยกรรมชุดคำสั่ง สมมุติที่ถูกออกแบบขึ้นมาเพื่อประกอบการเรียนการสอนในหนังสือเล่มนี้โดยเฉพาะ เนื่องจาก nanoLADA เป็นสถาปัตยกรรมที่มีชุดคำสั่งน้อยเจ้าใจง่าย ในบางตอนของบทนี้ อาจยกตัวอย่างสถาปัตยกรรมอื่นขึ้นมาอ้างอิง เพื่อเบริยบเทียบการทำงาน รายละเอียดของ สถาปัตยกรรมชุดคำสั่ง nanoLADA แสดงในภาคผนวก ก.

ลองจินตนาการว่า หากมีเอกสารหนึ่งชุดที่อธิบายสถาปัตยกรรมชุดคำสั่งของระบบ คอมพิวเตอร์ สิ่งที่ต้องศึกษาและกล่าวถึงในเอกสารชุดนี้ ต้องประกอบด้วยรายละเอียดเพิ่ม ฐานที่เพียงพอสำหรับให้ผู้พัฒนาซอฟต์แวร์รวมถึงผู้พัฒนาโปรแกรมแปลงภาษาสามารถที่จะ พัฒนาซอฟต์แวร์ได้ ด้วยแนวคิดนี้สิ่งที่ต้องกล่าวถึงในสถาปัตยกรรมชุดคำสั่ง ต้องประกอบ ด้วย

- คำสั่งต่าง ๆ พร้อมคำอธิบายเรื่อง จำนวนรอบที่ใช้ และความหมายในเชิงการ ทำงานระดับการแลกเปลี่ยนข้อมูลในระดับ rejister ซึ่งนิยมอธิบายด้วยภาษา Register Transfer Language (RTL)
- rejister หรือที่พกข้อมูลชั่วคราวภายในระบบ และ flag¹² แสดงสถานะของการ ทำงาน
- ระบบการจัดการหน่วยความจำ การเรียบร้อยและอ้างอิงข้อมูล รวมถึง สถาปัตยกรรมหน่วยความจำ
- รูปแบบคำสั่ง (instruction format)
- ระบบสัญญาณขัดจังหวะ¹³ (interrupt และ exception)

12 Flag คือ ลงทะเบียนสถานะของหน่วยประมวลผลกลางที่จะต้องเม็บบันทึกไว้เพื่อนำมาใช้อ้างอิงต่อ เช่น การบันทึกตัวทดสอบที่เกิดจากการบวกกับเพื่อนำไปบวกเพิ่มในรอบถัดไป

13 สัญญาณขัดจังหวะ คือ สัญญาณแจ้งเตือนให้หน่วยประมวลผลกลางหยุดการทำงานแล้วนำ โปรแกรมพิเศษขึ้นมาประมวลผลเพื่อให้ระบบทำงานต่อได้ถูกต้อง เช่น เมื่อมีการกดแป้นพิมพ์ สัญญาณขัดจังหวะจะแจ้งให้หน่วยประมวลผลกลางเรียกโปรแกรมอ่านค่าแป้นพิมพ์

ทั้งนี้สถาปัตยกรรมชุดคำสั่งมีความจำเป็นอย่างยิ่งในการพัฒนาอาร์ดแวร์ของระบบคอมพิวเตอร์ และ ซอฟต์แวร์ระบบ เนื่องจากเป็นข้อตกลงที่ใช้ระหว่างผู้พัฒนาอาร์ดแวร์และซอฟต์แวร์ (ดังที่ได้กล่าวแล้วในบทที่ 1) อย่างไรก็ตามสถาปัตยกรรมชุดคำสั่งมีได้ครอบคลุมถึงการเขียนภาษาแอสเซมบลีทั้งหมด เพราะในภาษาแอสเซมบลีจะมีเครื่องมือต่าง ๆ ที่จะช่วยให้เขียนซอฟต์แวร์ได้ง่ายขึ้นด้วย เช่น การกำหนด macro การอ้างอิงเลขที่อยู่ในหน่วยความจำด้วย label เป็นต้น

ก่อนจะเริ่มศึกษาเรื่องสถาปัตยกรรมชุดคำสั่ง มีคำศัพท์ที่ควรทราบในเบื้องต้นดังนี้

1. Instructions หรือ Operators ในที่นี้หมายถึงคำสั่งที่ต้องการให้หน่วยประมวลผลทำงาน เช่น การสั่งให้บวกเลข จะต้องสั่งด้วยคำสั่ง **add** เป็นต้น ซึ่งส่วนของคำสั่งนี้จะถูกแปลงเป็นรหัสคำสั่ง (opcode) ต่อไป
2. Operands หรือข้อมูลสำหรับการประมวลผล เช่น การสั่งให้หน่วยประมวลผล加ลงทำ 5+7 จะได้ว่า Instruction หรือ Operator คือการบวก (add) และมี Operands เป็น 5 และ 7 เป็นต้น
3. Memory Address หรือ เลขที่อยู่ของหน่วยความจำ หมายถึงที่อยู่ของข้อมูลในหน่วยความจำ เพื่อความสะดวกขอให้มองหน่วยความจำแต่ละไบต์เป็นกลุ่มเล็ก ๆ หลายกล่องที่วางต่อกัน โดยแต่ละกล่องมีการกำหนดหมายเลขเพื่อความสะดวกในการอ้างอิง
4. Register หมายถึง ที่พักข้อมูลชั่วคราวภายในหน่วยประมวลผลกลาง ในแต่ละหน่วยประมวลผลกลางจะมีการตั้งชื่อสำหรับเรียกเรจิสเตอร์ที่แตกต่างกันไปเพื่อกันความสับสน การอ้างอิงถึงค่าเรจิสเตอร์ในสถาปัตยกรรมชุดคำสั่งจะขึ้นต้นด้วย "\$" เช่น \$r1 \$r2
5. Immediate หรือ ค่าทันที หมายถึงค่าคงที่ซึ่งมีการระบุใน Operands เช่น คำสั่ง **add \$r1, #5, #7**

ซึ่งมีความหมายว่าให้นำ 5 + 7 และนำผลลัพธ์ที่ได้เก็บที่ เรจิสเตอร์ \$r1 กรณีนี้ operands คือ \$r1, 5 และ 7 โดยค่า 5 และ 7 จะเรียกว่าค่า immediate เพราะเป็นค่าที่ถูกระยะหัวไปเลยในคำสั่ง (คำสั่งสามารถนำค่า 5 และ 7 ไปใช้งานได้โดยไม่ต้องมีการประมวลผลเพิ่มเติม)

เพื่อกันความลับสน การอ้างอิงถึงค่า *immediate* ในสถาปัตยกรรมชุดคำสั่ง จะเขียนต้นด้วย "#" เช่น #5 #7

คำศัพท์เหล่านี้ ยังเป็นมาตรฐานสำหรับอ้างอิงในสถาปัตยกรรมคอมพิวเตอร์ทั่วไปอีกด้วย ดังนั้นในหนังสือนี้ข้ออ้างอิงถึงคำศัพท์เหล่านี้โดยทับศัพท์โดยตรง เพื่อให้ผู้อ่านคุ้นเคยกับ คำต่าง ๆ

การเขียนโปรแกรมด้วยภาษาแสชเมบลีเพื่อสั่งให้เครื่องคอมพิวเตอร์ทำงาน แต่ละคำสั่งจะ ประกอบด้วย opcode และ operands โดย opcode จะบอกถึงคำสั่งที่ต้องการทำงาน และ ตัวปฏิบัติการบอกถึงหน่วยความจำหรือ เรจิสเตอร์ที่ใช้ในการทำงานคำสั่งนั้น เช่น

ADD \$r0, \$r1, \$r2

AND \$r3, \$r0, \$r1

ADD และ AND นั้นเป็น Opcode ที่บอกให้ทราบว่าคำสั่งที่ต้องการทำงานคือ add และ การ and โดย \$r0, \$r1, \$r2, \$r3 เป็น Operand ที่บอกให้ทราบว่า ให้นำค่าในเรจิสเตอร์ \$r1 และ \$r2 มาบวกกันแล้วเก็บไว้ที่ \$r0 เป็นต้น

เพื่อให้เข้าใจภาพรวมและเห็นความสำคัญของสถาปัตยกรรมชุดคำสั่ง ในบทนี้จะเริ่มต้นด้วย การกล่าวถึงสถาปัตยกรรมชุดคำสั่งในอดีต การจำแนกหมวดคำสั่งการจัดการข้อมูลในหน่วย ความจำ และรูปแบบแบบชุดคำสั่งตามลำดับ

3.1 สถาปัตยกรรมชุดคำสั่งในอดีต

ก่อนที่จะมีการกำหนดสถาปัตยกรรมชุดคำสั่งขึ้นมาหนึ่น ในอดีต หน่วยประมวลผลกลางแต่ละ ชุดแต่ละรุ่น จะไม่มีความสอดคล้องกันแต่อย่างใด หมายความว่า หน่วยประมวลผลกลางรุ่น ใหม่จะไม่มีความเข้ากันได้ (compatibility) กับหน่วยประมวลผลกลางรุ่นก่อนหน้า แม้ว่าจะ เป็นหน่วยประมวลผลกลางจากผู้ผลิตรายเดียวกันก็ตาม ผลคือทุกครั้งที่มีการเปลี่ยนรุ่นของ หน่วยประมวลผลกลาง จะต้องทำการพัฒนาซอฟต์แวร์ใหม่ทั้งหมด

ในทางกลับกันหากหน่วยประมวลผลกลางชุดใดมีสถาปัตยกรรมชุดคำสั่งเหมือนกันหรือใกล้ เดียงกัน ย่อมหมายความว่า ซอฟต์แวร์ที่พัฒนาบนระบบหนึ่ง สามารถที่จะนำมาใช้งานได้ใน อิกรอบหนึ่งทันที ตัวอย่างเช่น การที่ซอฟต์แวร์ที่พัฒนาสำหรับหน่วยประมวลผลกลาง 386 ของ Intel สามารถทำงานได้ทันทีบนหน่วยประมวลผลกลางรุ่นใหม่ล่าสุด (เช่น Intel core i7) โดยไม่ต้องทำการเปลี่ยนแปลงแก้ไขใด ๆ ทั้งนี้เพราะหน่วยประมวลผลกลางทั้ง

สองแบบ ยังคงมีสถาปัตยกรรมชุดคำสั่งเดิมไว้

การจำแนกสถาปัตยกรรมชุดคำสั่งเป็นกลุ่มอาศัยการอ้างอิงข้อมูลในการประมวลผลเป็นเกณฑ์ สามารถแบ่งออกได้เป็น สถาปัตยกรรมแบบ accumulator (อ่านว่า แอกคูมูเลเตอร์) สถาปัตยกรรมแบบ stack สถาปัตยกรรมแบบ memory-memory สถาปัตยกรรมแบบ register-memory และ สถาปัตยกรรมแบบ register-register (บางตำราจะเรียกว่า สถาปัตยกรรมแบบ load-store)

บางตำราจะนิยมอ้างอิงถึงสถาปัตยกรรมแบบ memory-memory และสถาปัตยกรรมแบบ register-memory ว่า Complex Instruction Set Computing (CISC) ซึ่งเน้นให้หน่วยประมวลผลกลางประมวลผลงานที่ซับซ้อนภายในหนึ่งคำสั่ง ทั้งนี้เนื่องจากคำสั่งมีความซับซ้อน จะช่วยให้ตัวแปลภาษาทำการแปลงได้ง่ายกว่า ในทำนองเดียวกัน หลายตำรา尼ยมเรียกสถาปัตยกรรมแบบ register-register ว่า Reduce Instruction Set Computing (RISC) ซึ่งเป็นสถาปัตยกรรมที่เน้นให้แต่ละคำสั่งมีขนาดสั้นทำงานกระชับ และการทำงานที่ซับซ้อนจะขึ้นอยู่กับคอมไพล์เลอร์ที่จะต้องทำการแปลงคำสั่งที่ซับซ้อนให้เป็นคำสั่งที่สั้นและง่ายulatory คำสั่งทำงานต่อ กัน ผลที่ได้คือโปรแกรมบน CISC จะเล็กกว่า RISC แต่หากวิเคราะห์เรื่องของสมรรถนะร่วมด้วย โปรแกรมบน RISC (ซึ่งใหญ่กว่า CISC) ทำงานได้รวดเร็วกว่ามาก ซึ่งจากล่า夙ถึงต่อไป

ปัจจุบัน เรายุค Post RISC¹⁴ กล่าวคือ หน่วยประมวลผลกลางบางแบบ (เช่น Intel core i3, i5 i7) แม้จะมีสถาปัตยกรรมชุดคำสั่งแบบ CISC (มีคำสั่งที่มีการใช้งานแบบ register-memory) แต่โครงสร้างภายในจะมีการแปลงคำสั่งเหล่านี้เป็นคำสั่งจิ๋ว (micro ops) สำหรับใช้งานภายในหน่วยประมวลผลกลางก่อน (micro ops เป็นสถาปัตยกรรมแบบ register-register) จากนั้นหน่วยประมวลผลกลางจึงจะทำงานด้วยคำสั่ง micro ops แทน จึงไม่อาจกล่าวได้ว่า หน่วยประมวลผลได้เป็น CISC หรือ RISC ได้ชัดเจนนัก

ในเบื้องต้น ขอให้ผู้เรียนทำความเข้าใจสถาปัตยกรรมแบบต่าง ๆ เพื่อเป็นแนวทางในการศึกษาและทำความเข้าใจสมรรถนะของหน่วยประมวลผลกลางต่อไป ในที่นี้จะใช้ชุดคำสั่งสมมติซึ่งประกอบไปด้วย

- LD (Load Data) หรือ PUSH สำหรับอ่านค่าหน่วยความจำมาใส่ในหน่วยประมวลผลกลาง

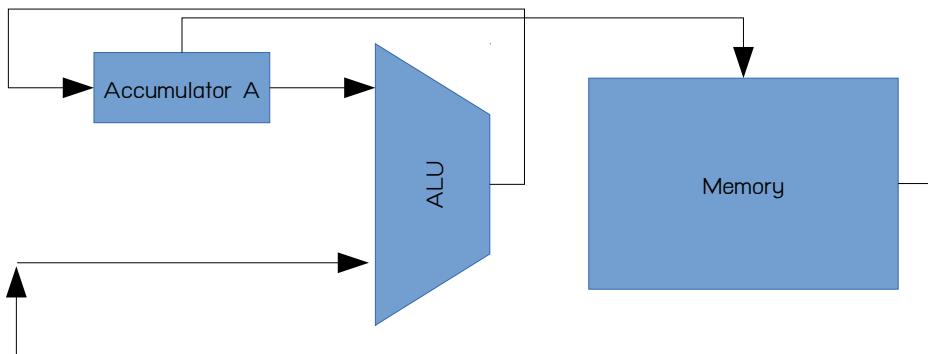
14 Beyond RISC - The Post-RISC Architecture เป็นการกล่าวถึงสถาปัตยกรรมที่อาจจะมีโครงสร้างต้านบน ดูเหมือน CISC แต่มีการทำงานภายในเป็นแบบ RISC
<http://www.cse.msu.edu/~enbody/postrisc/postrisc2.htm>

- SD (Store Data) หรือ POP สำหรับบันทึกค่าจากหน่วยประมวลผลกลางกลับลงไปยังหน่วยความจำ
- ADD (Addition) สำหรับประมวลผลการบวกข้อมูลเข้าด้วยกัน
- SUB (Subtraction) สำหรับประมวลผลการลบข้อมูลออกจากกัน
- MUL (Multiplication) สำหรับประมวลผลการคูณข้อมูลเข้าด้วยกัน

3.1.1 สถาปัตยกรรมแบบ Accumulator

สถาปัตยกรรมแบบ accumulator หรือ accumulator machine นั้นเป็นสถาปัตยกรรมในยุคแรกที่ทรานซิสเตอร์ยังมีราคาแพง ทำให้การสร้างหน่วยประมวลผลจำเป็นต้องออกแบบให้มีหน่วยความจำภายในเฉพาะเท่านั้นที่จะเป็น accumulator นั้นความจริงก็เป็นเพียงเรจิสเตอร์ตัวหนึ่งภายในระบบเท่านั้น

สถาปัตยกรรมชุดคำสั่งในลักษณะนี้ คือสถาปัตยกรรมที่มีเรจิสเตอร์สำหรับใช้ในการประมวลผลข้อมูลในหน่วยประมวลผลกลางเพียงตัวเดียว ซึ่งการคำนวณจะต้องใช้ accumulator ร่วมด้วย และผลลัพธ์ที่ได้จะเก็บใน accumulator เสมอ รูปที่ 3.1 แสดงทางเดินข้อมูลของสถาปัตยกรรมแบบ accumulator



รูปที่ 3.1: ทางเดินข้อมูลของสถาปัตยกรรมแบบ accumulator

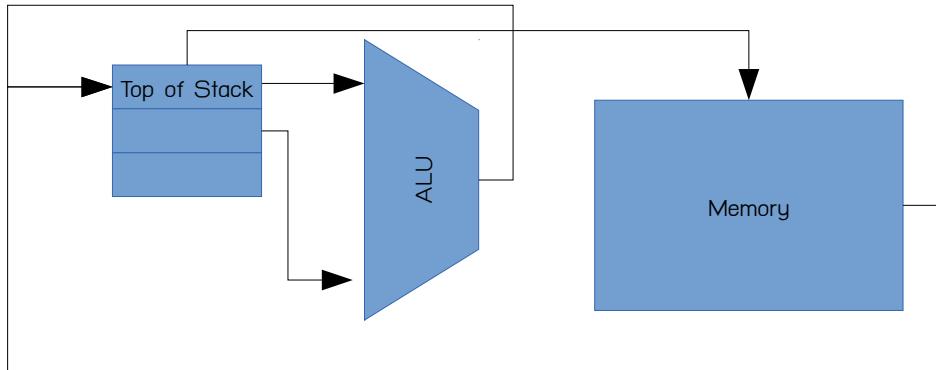
หากต้องการประมวลผล $x = (a * b) + (a - (b * c))$ เมื่อแปลงเป็นภาษาเครื่อง จะได้ลำดับการทำงานดังนี้

1.	LD	\$A, [a]	; \$A=a
2.	MUL	\$A, [b]	; \$A=a*b
3.	SD	\$A, [x]	; x=\$A
4.	LD	\$A, [b]	; \$A=b
5.	MUL	\$A, [c]	; \$A=b*c
6.	LD	\$A, [a]	; \$A=a
7.	SUB	\$A, [y]	; \$A=a-(b*c)
8.	SD	\$A, [y]	; y=\$A
9.	LD	\$A, [x]	; \$A=x
10.	ADD	\$A, [y]	; \$A=(a*b)+(a-b*c)
11.	SD	\$A, [x]	; x=\$a

จะเห็นว่าการคำนวนพื้นฐานดังกล่าว ต้องใช้ถึง 11 คำสั่ง ส่วนหนึ่งเนื่องจากมิที่พักข้อมูลให้ใช้เพียงหนึ่งที่ คือ accumulator ทำให้ทุกครั้งที่จำเป็นต้องมีการคำนวนด้วย accumulator และเมื่อค่าอื่นที่จะต้องใช้ค้างอยู่ใน accumulator จะต้องมีการย้ายข้อมูลออกมาพักในหน่วยความจำก่อนเสมอ (คำสั่งที่ 3 และ 8) ทำให้ accumulator เป็นคอขวดของการประมวลผลนั่นเอง

3.1.2 สถาปัตยกรรมแบบ Stack

สถาปัตยกรรมแบบ stack หรือ stack machine อาศัยหลักการว่า operands จะอยู่ด้านบนของ stack เสมอ สมมุติฐานนี้ทำให้แต่ละคำสั่งการประมวลผลมีขนาดเล็ก (เพราะไม่จำเป็นต้องระบุ operands) อย่างไรก็ตามปัญหาที่เห็นได้ชัดคือ stack จะกลายเป็นคอขวดของการประมวลผลในทำนองเดียวกับสถาปัตยกรรมแบบ accumulator ซึ่งทำให้มีสัดส่วนต่อการทำการประมวลผลแบบขานาน รูปที่ 3.2 แสดงทางเดินข้อมูลของสถาปัตยกรรมแบบ stack



รูปที่ 3.2: ทางเดินข้อมูลของสถาปัตยกรรมแบบ stack

หากต้องการประมาณผล $x = (a * b) + (a - (b * c))$ เมื่อแปลงเปรียบดังกล่าวเป็นภาษาเครื่อง จะได้ลำดับการทำงานดังนี้

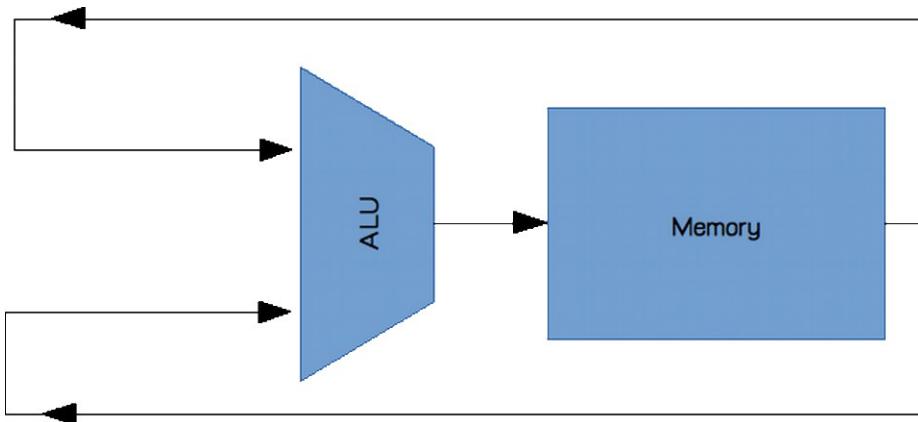
1. PUSH a ; (top of stack) a
2. PUSH b ; (top of stack) b , a
3. MUL ; (top of stack) (a*b)
4. PUSH a ; (top of stack) a, (a*b)
5. PUSH b ; (top of stack) b, a, (a*b)
6. PUSH c ; (top of stack) c, b, a, (a*b)
7. MUL ; (top of stack) (b*c), a, (a*b)
8. SUB ; (top of stack) (a - (b*c)) , (a*b)
9. ADD ; (top of stack) (a*b) + (a - (b*c))
10. POP x ;

เมื่อเปรียบเทียบกับสถาปัตยกรรมแบบ accumulator จะเห็นว่าจำนวนคำสั่งที่ใช้มีปริมาณไม่แตกต่างกัน อย่างไรก็ตาม คำสั่งของสถาปัตยกรรมแบบ stack จะสั้นกว่ามาก (เรียกว่ามี

code density ดี เพราะเจียนคำสั่งสั้นแต่ได้ความหมายสมบูรณ์ นอกจากนี้ยังสะดวกต่อการออกแบบฮาร์ดแวร์ และสะดวกต่อการทำงานของคอมไฟเลอร์ เพราะในการแปลภาษาคอมไฟเลอร์ต้องสร้างต้นไม้ภาษาสัมพันธ์¹⁵ (syntax tree) ก่อนอยู่แล้ว

3.1.3 สถาปัตยกรรมแบบ Memory-Memory

สถาปัตยกรรมแบบ memory-memory หรือ memory-memory machine เป็นรูปแบบสถาปัตยกรรมที่พับໄได้ในเครื่อง mainframe บางระบบ ซึ่งสถาปัตยกรรมในลักษณะนี้สามารถประมวลผลได้โดยไม่ต้องมี เรจิสเตอร์ หรือ accumulator ภายในระบบ กล่าวคือ การประมวลผลแต่ละคำสั่งอ่านและเขียนค่าจากหน่วยความจำโดยตรง หากจะจำแนกย่อย ต่อ ยังแบ่งออกเป็นแบบ 2 operands และ แบบ 3 operands ซึ่งแตกต่างกันที่ความยาวของคำสั่งและตำแหน่งที่ใช้เก็บผลลัพธ์ โดยแบบ 2 operands จะใช้ตัวตั้งตัวหนึ่งในการเก็บผลลัพธ์ และแบบ 3 operands จะต้องระบุทั้งตัวตั้ง ตัวประมวลผล และ ตัวเก็บผลลัพธ์ รูปที่ 3.3 แสดงทางเดินข้อมูลของสถาปัตยกรรมแบบ memory-memory



รูปที่ 3.3: ทางเดินข้อมูลของสถาปัตยกรรมแบบ memory-memory

หากต้องการประมวลผล $x = (a * b) + (a - (b * c))$ เมื่อแปลงเปรียบดังกล่าวเป็นภาษาเครื่องสำหรับสถาปัตยกรรมในแบบ 3 operands จะได้ลำดับการทำงานดังนี้

1. MUL k, a, b ; k = a*b

¹⁵ ในการแปลภาษาคอมพิวเตอร์ ต้นไม้ภาษาสัมพันธ์ (syntax tree) จะแสดงลำดับการประมวลผลก่อน หลังเพื่อให้การประมวลผลถูกต้อง รายละเอียดเพิ่มเติมศึกษาได้จากตำราคอมไฟเลอร์ที่ไว้ไป

2. MUL i, b, c ; $i = b*c$
3. SUB j, a, i ; $j = a-i = a - (b*c)$
4. ADD x, k, j ; $x = k+j = (a*b) + (a - (b*c))$

หรือในการณ์สถาปัตยกรรมแบบ 2 operands จะได้ลำดับการทำงานเป็น

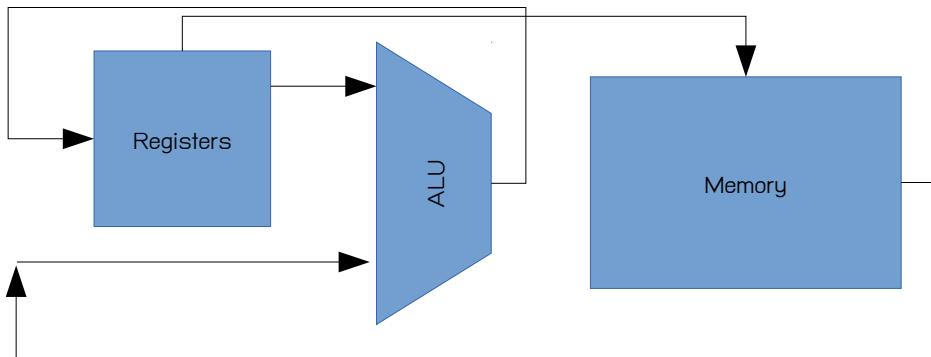
1. LD x,a ; $x = a$
2. MUL x, b ; $x = x*b = a*b$
3. LD i, b ; $i = b$
4. MUL i, c ; $i = i*c = b*c$
5. LD j, a ; $j = a$
6. SUB j, i ; $j = j - i = a - (b*c)$
7. ADD x, j ; $x = x+j = (a*b) + (a - (b*c))$

จะสังเกตเห็นว่า ไม่ว่ากรณ์ 3 operands หรือ 2 operands ต่างก็ให้ผลลัพธ์เป็นโปรแกรมที่ สั้นกว่าสถาปัตยกรรมแบบ accumulator และสถาปัตยกรรมแบบ stack อย่างไรก็ตามข้อ เสียของสถาปัตยกรรมในลักษณะนี้คือ ทำให้คำสั่งแต่ละคำสั่งมีการทำงานที่ซับซ้อน เพราะ การทำงานทุกครั้งต้องอ่านเขียนข้อมูลจากหน่วยความจำ ซึ่งทำงานช้ากว่าเรจิสเตอร์มาม (เปรียบเทียบเรจิสเตอร์เหมือนค่าตัวเลขที่จำอยู่ในสมองระหว่างการคำนวณกับ หน่วยความจำคือตัวเลขที่เขียนอยู่ในกระดาษทด ซึ่งการอ่าน/เขียนข้อมูลจากกระดาษทดย่อมช้ากว่า เสมอ)

3.1.4 สถาปัตยกรรมแบบ Register-Memory

สถาปัตยกรรมแบบ register-memory หรือ register-memory machine มี operands ตัว หนึ่งอยู่ในหน่วยความจำ และอีกตัวหนึ่งเป็นเรจิสเตอร์ สถาปัตยกรรมลักษณะนี้พัฒนาใน สถาปัตยกรรมยุค 1950, 1960, 1970 ซึ่งเป็นยุคที่นิยมให้หน่วยประมวลผลกลางทำงานที่ ซับซ้อนแทนการให้คอมไพล์เม้นท์แล้ว การทำงานที่ซับซ้อนเป็นคำสั่งย่อย ซึ่ง Intel 386 อัน เป็นที่นิยมในปัจจุบันก็จัดอยู่ในกลุ่มนี้ รูปที่ 3.4 แสดงทางเดินข้อมูลของสถาปัตยกรรมใน

ลักษณะนี้ ในเบื้องต้นโครงสร้างจะดูคล้ายกับแบบ accumulator แต่ต่างตรงที่สถาปัตยกรรมแบบนี้มีเรจิสเตอร์ให้ใช้มากกว่า 1 ตัว



รูปที่ 3.4: ทางเดินข้อมูลของสถาปัตยกรรมแบบ register-memory

หากต้องการประมวลผล $x = (a * b) + (a - (b * c))$ เมื่อแปลงประโยคดังกล่าวเป็นภาษาเครื่อง จะได้ลำดับการทำงานดังนี้

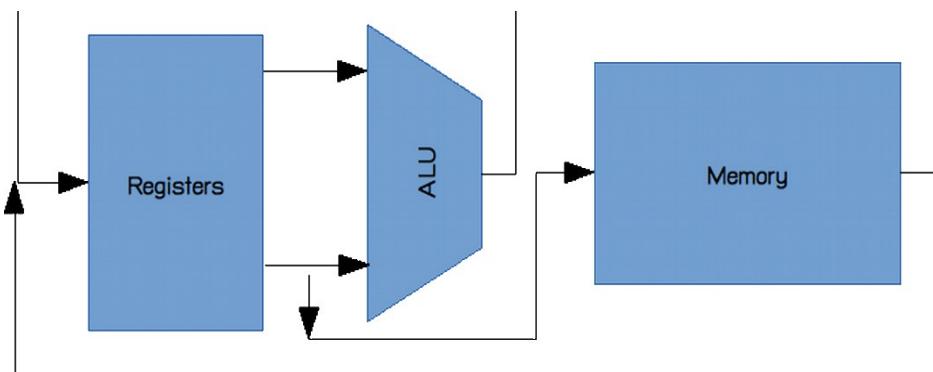
1. LD \$r1, a ; \$r1 = a
2. MUL \$r1, b ; \$r1 = \$r1 * b = a * b
3. LD \$r2, b ; \$r2 = b
4. MUL \$r2, c ; \$r2 = \$r2 * c = b * c
5. SD \$r2, j ; j = \$r2
6. LD \$r3, a ; \$r3 = a
7. SUB \$r3, j ; \$r3 = \$r3 - j = a - (b * c)
8. SD \$r3, j ; j = \$r3
9. ADD \$r1, j ; \$r1 = \$r1 - j
10. SD \$r1, x

สถาปัตยกรรมชุดคำสั่งในอดีต

หากวิเคราะห์ในเบื้องต้นจะเห็นการทำงานคล้ายคลึงกับแบบ memory-memory ผสมกับแบบ accumulator หากแต่มีเรจิสเตอร์ภายในให้ใช้มากกว่า ข้อเสียอย่างหนึ่งที่เห็นได้ชัดคือ operands แต่ละตัวจะใช้เวลาในการเข้าถึงไม่เท่ากัน ตัวหนึ่งจะอยู่ในเรจิสเตอร์ เข้าถึงได้เร็ว ในขณะที่การอ่านค่า operands ในหน่วยความจำจะช้ากว่าเสมอ นอกจากนี้การที่ operand มีการอ้างอิงไม่เหมือนกัน จึงเกิดสิ่งที่เรียกว่า poor orthogonal กล่าวคือ เกิดการอ้างอิงของ operand ไม่สอดคล้องกัน มีความเร็วในการเข้าถึงไม่เท่ากัน

3.1.5 สถาปัตยกรรมแบบ Register-Register

สถาปัตยกรรมแบบ register-register หรือ load-store machine เป็นรูปแบบของสถาปัตยกรรมที่มักพบในหน่วยประมวลผลกลางรุ่นใหม่ทั่วไป ตัวอย่างที่เด่นชัด เช่น สถาปัตยกรรมแบบ ARM ซึ่งใช้ในโทรศัพท์แบบพกพาทั่วไปในปัจจุบัน ก็เป็นสถาปัตยกรรมในลักษณะนี้ หลักการพื้นฐานของสถาปัตยกรรมในลักษณะนี้คือ การประมวลผลที่เป็นการคำนวณ จะต้องใช้ operands ที่เป็นเรจิสเตอร์เท่านั้น กล่าวคือ จะต้องทำการ load ค่าที่ต้องการใช้ในการคำนวณเข้ามาจำในเรจิสเตอร์ไว้ก่อน จากนั้นเมื่อประมวลผลเสร็จจะ store ค่าที่ได้คืนลงไปในหน่วยความจำ (การทำงานลักษณะนี้จึงเป็นที่มาของชื่อ load-store architecture)



รูปที่ 3.5: โครงสร้างทางเดินข้อมูลของสถาปัตยกรรมแบบ register-register

หากต้องการประมวลผล $x = (a^*b) + (a - (b^*c))$ เมื่อแปลงประโยคดังกล่าวเป็นภาษาเครื่อง จะได้ลำดับการทำงานดังนี้

1.	LD	\$r1, a ; \$r1 = a
2.	LD	\$r2, b ; \$r2 = b
3.	LD	\$r3, c ; \$r3 = c
4.	MUL	\$r4, \$r1, \$r2 ; \$r4 = \$r1 * \$r2 = a*b
5.	MUL	\$r5, \$r2, \$r3 ; \$r5 = \$r2 * \$r3 = b*c
6.	SUB	\$r6, \$r1, \$r5 ; \$r6 = \$r1 - \$r5 = a - (b*c)
7.	ADD	\$r7, \$r4, \$r6 ; \$r7 = \$r4 + \$r6 = (a*b) + (a - (b*c))
8.	SD	\$r7, x

ข้อดีของสถาปัตยกรรมในลักษณะนี้คือ เมื่อค่าที่ต้องการอยู่ใน寄存器แล้ว การประมวลผลจะทำให้อย่างรวดเร็ว นอกจากนี้ทุกคำสั่งยังมีความยาวของรูปแบบคำสั่งเท่ากัน (รูปแบบคำสั่งดูที่ 3.5) ทำให้การออกแบบรูปแบบคำสั่งทำได้ง่าย แต่สมรรถนะที่ได้นั้นขึ้นกับการทำงานของคอมพิวเตอร์ และในหลายกรณีจะใช้จำนวนคำสั่งมากกว่าเมื่อเปรียบเทียบกับสถาปัตยกรรมแบบ memory-memory

นอกจากนี้ ข้อเด่นที่ทำให้สถาปัตยกรรมคอมพิวเตอร์รุ่นใหม่ นิยมออกแบบเป็น register-register คือ ความสามารถในการทำ Pipeline หรือความสามารถในการประมวลผลแบบขนาน ซึ่งจะกล่าวถึงต่อไปในบทที่ 6

3.2 หมวดคำสั่งและชุดคำสั่ง

ชุดคำสั่งในหน่วยประมวลผลต่าง ๆ โดยทั่วไปสามารถจำแนกตามลักษณะการทำงาน หรือการเรียกใช้งาน ได้เป็นหมวดหมู่ดังนี้

- หมวดคำสั่งที่เกี่ยวกับการเคลื่อนย้ายข้อมูล (data movement) ใช้สำหรับการอ่านและเขียนข้อมูลในหน่วยความจำ
- หมวดคำสั่งการประมวลผลทางคณิตศาสตร์ (เช่น บวก ลบ คูณ หาร)
- หมวดคำสั่งการทำงานทางตรรกะ (เช่น and or not xor)

4. หมวดคำสั่งควบคุม เพื่อกำหนดเงื่อนไขของการทำงานคำสั่งถัดไป เช่น IF .. THEN .. ELSE รวมถึงการเรียกโปรแกรมย่อย (subprogram)

นอกจากนี้ สถาปัตยกรรมบางแบบอาจมีหมวดคำสั่งพิเศษอื่นตามโครงสร้างและการออกแบบของผู้พัฒนา เช่น ชุดคำสั่งที่มีการทำงานในลักษณะของเวกเตอร์ กล่าวคือ 1 คำสั่ง มีประมวลผลกับข้อมูลหลายชุด (มีการทำงานแบบขนาน) ซึ่งหมายความว่า การประมวลผลด้านกราฟิกส์ (graphics) และระบบสื่อประสม (multimedia) เป็นต้น

ในการประมวลผลแต่ละคำสั่งนั้น หน่วยประมวลผลกลางจะมีขั้นตอนในการดึงคำสั่งจากหน่วยความจำเข้าสู่หน่วยประมวลผลกลาง จากนั้นจึงทำการประมวลผลต่อไป (รายละเอียดเพิ่มเติมดูในบทที่ 5 และ 6) ทั้งนี้คำสั่งที่จะถูกดึงเข้ามาหนึ่งจะถูกกำหนดโดย rejister เวกเตอร์พิเศษ ซึ่งเรียกว่า Program Counter (PC) หรือ สถาปัตยกรรมบางแบบอาจจะเรียกว่า rejister เวกเตอร์พิเศษนี้ว่า Instruction Pointer¹⁶ (IP) ดังนั้นลักษณะพิเศษของคำสั่งควบคุมคือสามารถปรับเปลี่ยน PC ให้จี้ไปยังคำสั่งอื่น ซึ่งมิได้เป็นคำสั่งที่ติดกันก็ได้ เช่น

```
BEQ    $r0,$r1,test
       ADD    $r0,$r1,$r2
test:   SUB    $r0,$r1,$r2
```

คำสั่ง BEQ เป็นคำสั่งเปรียบเทียบค่าภายใน rejister \$r0 และ \$r1 ถ้าเท่ากันจะไปทำคำสั่ง SUB หากไม่เท่ากันจะทำคำสั่ง ADD ซึ่งเป็นคำสั่งถัดไปที่ติดกัน

สถาปัตยกรรมชุดคำสั่งแบบ LADA ที่ใช้อ้างอิงในหนังสือเล่มนี้ เป็นสถาปัตยกรรมแบบ register-register ดังนั้น การประมวลผลทางคณิตศาสตร์จะมี operand เป็น rejister เท่านั้น โดย operand ตัวแรกจะใช้บوك rejister ที่ใช้เก็บผลลัพธ์ และ operand ตัวมารอึก 2 ตัวจะใช้บوك operands ที่เป็นตัวตั้ง \$r1 (ในที่นี้ขอเรียกว่า source) และ ตัวประมวลผล \$r2 (ในที่นี้ขอเรียกว่า target) เช่น

```
ADD    $r0, $r1, $r2
```

¹⁶ ในสถาปัตยกรรม Intel x86 จะมีการแบ่ง Program Counter ออกเป็น 2 ส่วนคือ Code Segment (CS) และ Instruction Pointer (IP) หรือในระบบ 32 บิตจะเรียกว่า ECS และ EIP ตามลำดับ ทั้งนี้เนื่องจากสถาปัตยกรรม x86 รองรับการจัดการหน่วยความจำแบบ segmentation ด้วย (ซึ่งมีได้กล่าวถึงในหนังสือนี้)

มีความหมายคือ

$$\$r0 = \$r1 + \$r2$$

ทั้งนี้แนวทางหนึ่งอธิบายความหมายของคำสั่งคือ การใช้ (Logical) Register-Transfer Language กล่าวคือ การอธิบายความ คำสั่งทำงานอย่างไร โดยระบุว่ามีแลกเปลี่ยนข้อมูลระหว่าง rejister ใดในระบบบ้าง เช่น เมื่อต้องการอธิบายการทำงานของคำสั่ง ADD ในคู่มือของสถาปัตยกรรมชุดคำสั่ง อาจอธิบายดังนี้

คำสั่ง: ADD rd, rs, rt

RTL:

{op, rs, rt, rd } \leftarrow MEM[PC] ; ลำดับ operands จัดกับรูปแบบคำสั่ง

R[rd] \leftarrow R[rs] + R[rt]

PC \leftarrow PC + 4 ; คำสั่งมีความยาว 4 ไบต์

จากตัวอย่าง RTL อธิบายการทำงานของคำสั่ง ADD คือ (1) อ่านค่า {op, rs, rt, rd} ซึ่งเป็นค่าของคำสั่งจากหน่วยความจำ ณ เลขที่อยู่ที่ซื้อด้วย PC (ขั้นตอนนี้เรียกว่า Fetch เพราะเป็นการอ่านค่าคำสั่ง) (2) นำค่า rejister ที่ซื้อด้วย rs และ rt มาบวกกัน แล้วนำผลลัพธ์ที่ได้ไปเก็บใน rejister ที่ซื้อด้วย rd และ (3) เพิ่มค่า PC จัดอีก 4 (ในกรณีนี้ สมมุติว่าแต่ละคำสั่งมีความยาว 32 บิต หรือ 4 ไบต์)

ดังนั้นกรณีที่พัฒนาซอฟต์แวร์ด้วยภาษาชั้นสูงแล้วมีการประกาศหรือใช้ตัวแปรสำหรับการอ้างอิงค่า ตัวแปรภาษาจะต้องทำการจับคู่ (associate) ตัวแปร กับ rejister ในขณะที่มีการประมวลผลแปลงภาษา ดังแสดงในตัวอย่างที่ 3.1

เนื่องจาก nanoLADA มี rejister ให้ใช้เพียง 32 ตัวเท่านั้น ดังนั้นหากในโปรแกรมที่มีตัวแปรมากกว่า 32 ตัว คอมไพล์เตอร์ต้องจับคู่ตัวแปรกับ rejister ให้เหมาะสม เพื่อใช้ประโยชน์จากความเร็วของ rejister ให้มากที่สุด เมื่อ rejister ถูกใช้อ้างอิงกับตัวแปรจนหมด บางตัวจะต้องถูกจัดเก็บไว้ในหน่วยความจำ และมีการอ่านค่าขึ้นมาใหม่เมื่อจำเป็นต้องอ้างอิงอีก

ตัวอย่างที่ 3.1: ตัวอย่างการแปลงภาษาซีเป็น LADA code

ภาษาซี:

```
{
    int A,B,C,D,E,F;
    A = B + C + D;
    E = F - A;
}
```

หากคอมไพล์เลอร์กำหนดให้ A, B, C, D, E, F ถูกแทนค่าด้วย \$r1, \$r2, \$r3, \$r4, \$r5, \$r6 ตามลำดับ จะได้ว่า

LADA CODE:

LD	\$r2, B
LD	\$r3, C
LD	\$r4, D
LD	\$r6, F
ADD	\$r1, \$r2, \$r3
ADD	\$r1, \$r1, \$r4
SUB	\$r5, \$r6, \$r1
SD	\$r1, A
SD	\$r5, E

3.3 การจัดการหน่วยความจำ (Memory Organization)

หน่วยความจำนั้นเป็นเหมือนกับ array 1 มิติขนาดใหญ่ที่อ้างอิงด้วยเลขที่อยู่ LADA จะใช้เลขที่อยู่ขนาด 32 บิตในการอ้างอิงหน่วยความจำ ซึ่งหมายความว่า LADA มีหน่วยความจำได้ทั้งสิ้น 2^{32} ไบต์ หรือ 4 กิกะไบต์ (Gigabyte)

การอ้างอิงถึงหน่วยความจำนั้นอาจจะเป็นการอ้างถึงครั้งละ 1 ไบต์ (byte addressing) หรืออ้างอิงครั้งละ 2 ไบต์ (half-word addressing) หรืออาจจะอ้างอิงครั้งละ 4 ไบต์ (word addressing) ก็ได้ ทั้งนี้ขึ้นอยู่กับรูปแบบคำสั่ง

นอกจากขนาดของเลขที่อยู่และเรจิสเตอร์ ที่เกี่ยวข้องกับการอ้างอิงข้อมูลในหน่วยความจำ

แล้ว การออกแบบสถาปัตยกรรมชุดคำสั่งสำหรับอ้างอิงหน่วยข้อมูลในหน่วยความจำมีข้อที่ต้องคำนึงถึงอยู่อีกสองประการคือ การจัดลำดับข้อมูล (endian) และการกำหนดเลขที่อยู่เริ่มต้นของข้อมูล (memory alignment)

3.3.1 การจัดลำดับข้อมูล (Endian)

สิ่งที่เป็นประเด็นสำคัญในการอ้างอิงข้อมูลที่มีขนาดมากกว่า 1 ไบต์ (แบบ word addressing และ half-word addressing) คือ การจัดลำดับข้อมูล การจัดลำดับแบบ big endian และ การจัดลำดับแบบ little endian ซึ่งจะแตกต่างกันไปตามโครงสร้างของหน่วยประมวลผลกลาง รูปที่ 3.6 แสดงการจัดเก็บข้อมูล 0A0B0C0Dh (ฐาน 16) ลงในหน่วยความจำ

(a) ระบบ big endian

Address	00	01	02	03
	0Ah	0Bh	0Ch	0Dh

(b) ระบบ little endian

Address	00	01	02	03
	0Dh	0Ch	0Bh	0Ah

รูปที่ 3.6: การเรียงข้อมูล (a) big endian (b) little endian

จากรูปข้อต่อต่างที่เห็นได้ชัดคือ ในการจัดลำดับแบบ big endian ส่วนที่มีนัยสำคัญต่อทางขวาของ การเรียงตัวเลขแบบปกติ จะอยู่ที่เลขที่อยู่สูง ในขณะที่การจัดลำดับแบบ little endian ส่วนที่มีนัยสำคัญต่อ จะอยู่ที่เลขที่อยู่ต่ำๆ เช่นกัน

ในปัจจุบันไม่เห็นข้อต่อต่างของข้อดีข้อเสียของระบบการจัดลำดับมากนัก เนื่องจากระบบสายสัญญาณสำหรับอ่านข้อมูลมีขนาดใหญ่พอที่จะอ่านข้อมูลเข้ามาได้ทั้งหมด อย่างไรก็ตาม หากการอ่านข้อมูลหนึ่งครั้ง ไม่สามารถอ่านได้ครบทั้ง word (เช่น ต้องการอ่านข้อมูล 32 บิตแต่มีสายสัญญาณ 16 บิต) การอ่านข้อมูลจากเลขที่อยู่ต่ำไปสูงในแบบ big endian จะทำให้เราทราบเครื่องหมาย (ซึ่งอยู่ช้ายมือเวลาเรียกเลข) ได้ก่อน ช่วยให้การอ่านเลขที่ต้องมีการเปรียบเทียบค่าทำได้รวดเร็วขึ้น (เพราะเห็นเครื่องหมาย หรืออ่านลำดับสูงก่อน) ทันทีว่า ค่าใดมากกว่า (นั่นเอง) ในทำนองกลับกัน สถาปัตยกรรมที่ใช้การจัดลำดับแบบ little endian จะทำให้การประมวลผลบางแบบ (เช่นการบวกลบทางคณิตศาสตร์) ทำได้รวดเร็ว

กว่า เพราะเมื่ออ่านข้อมูลจากเลขที่อยู่ต่ำไปสูง สามารถคำนวนเลข (เช่น บวก) จากตำแหน่งที่มีนัยสำคัญต่ำไปก่อนได้ทันที โดยไม่ต้องรอให้อ่านข้อมูลได้ครบก่อน

สถาปัตยกรรม LADA ซึ่งกล่าวถึงในบทเรียนจะมีโครงสร้างการจัดลำดับแบบ big endian ในขณะที่หน่วยประมวลผลกลาง Intel ตระกูล x86 ที่พบทั่วไปจะใช้การจัดลำดับแบบ little endian ดังนั้นในการพัฒนาซอฟต์แวร์บนสถาปัตยกรรมที่มีการจัดลำดับแตกต่างกัน ผู้พัฒนาต้องใช้ความระมัดระวังเสมอ (ปัจจุบันมักเป็นหน้าที่ของคอมไฟเลอร์)

หมายเหตุ

1. ในสถาปัตยกรรมบางระบบ (เช่น ARM, PowerPC) สามารถกำหนดได้ว่าจะใช้ big endian หรือ little endian ในขณะที่ระบบเริ่มทำงาน
2. นอกจากนี้ ยังมีบางสถาปัตยกรรมที่เป็นแบบ middle endian ซึ่งมีการเรียงข้อมูล ผสมระหว่าง big endian และ little endian แต่ไม่ค่อยพบในปัจจุบัน (จึงมีได้ กล่าวถึงในที่นี้)

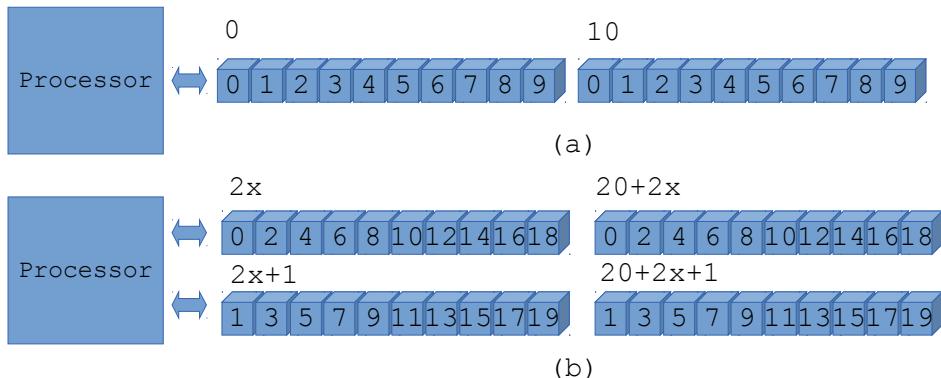
ประเด็นช่วยจำ หลักการเบื้องต้น ที่จะช่วยลดความสับสนของการจัดลำดับแบบ big endian และ little endian คือ กรณี **big endian** ข้อมูลที่มีนัยสำคัญสูงจะอยู่ที่เลขที่อยู่ต่ำส่วนกรณีของ **little endian** นั้น ข้อมูลที่มีนัยสำคัญต่ำ จะอยู่ที่เลขที่อยู่ต่ำ เช่นกัน กล่าวคือ big endian จะสลับนัยสำคัญกับเลขที่อยู่

3.3.2 การกำหนดเลขที่อยู่เริ่มต้นของข้อมูล (Memory Alignment)

หลักการเบื้องต้นของการกำหนดเลขที่อยู่เริ่มต้นของข้อมูล คือ เลขที่อยู่เริ่มต้นของข้อมูล (ที่มีการอ้างอิงแบบ word addressing หรือ half-word Addressing) จะต้องเป็นค่าลงตัว ของการหารด้วยเลขที่กำหนด ซึ่งกำหนดโดยขนาดของสายสัญญาณ เช่น หากกำหนดว่า alignment จะต้องหารด้วย 4 ลงตัว (ขนาดสายสัญญาณ 32 บิต) หมายความว่า จะต้องมีการกำหนดเลขที่อยู่เริ่มต้นเป็น 0h, 4h, 8h , Ch (ฐาน 16) การที่บางสถาปัตยกรรมมีการกำหนดเงื่อนไขในลักษณะนี้ เนื่องจากโครงสร้างของระบบ硬件และสายสัญญาณที่อยู่ภายในระบบไม่เอื้ออำนวยต่อการอ่านหรือเขียนข้อมูลแบบไม่มี alignment

ยกตัวอย่างการกำหนด alignment อ้างอิงเพื่อประกอบความเข้าใจดังนี้.... สมมุติให้กอลองแต่ละกอลองแทนหน่วยความจำ 1 ไบต์ 1 ແຕງ ในการนำกลองดังกล่าวมาต่อพ่วงกันเพื่อให้ได้หน่วยความจำที่มากขึ้นสามารถทำได้หลายแบบ (ดูรูปที่ 3.7) โดยแบบหนึ่งคือการต่อແຕງให้ยาวออกไป ซึ่งกรณีนี้ เรายังอ่านข้อมูลได้ทีละ 1 ไบต์เท่าเดิม อีกแนวทางหนึ่งคือการนำ

กล่องดังกล่าวมาต่อขานกัน เพื่อให้อ่านข้อมูลได้ครั้งละ 2 ไบต์ จะสังเกตว่า ปริมาณข้อมูลที่มียังเท่าเดิมคือ 20 ไบต์ หากแต่อ่านได้พร้อมกันครั้งละ 2 ไบต์ (ต่างจากแบบเดิมที่อ่านได้ครั้งละ 1 ไบต์)



รูปที่ 3.7: การต่อหน่วยความจำเข้ากับหน่วยประมวลผล (a) ความกว้างสายสัญญาณเป็นหนึ่ง (b) ความกว้างสายสัญญาณเป็นสอง

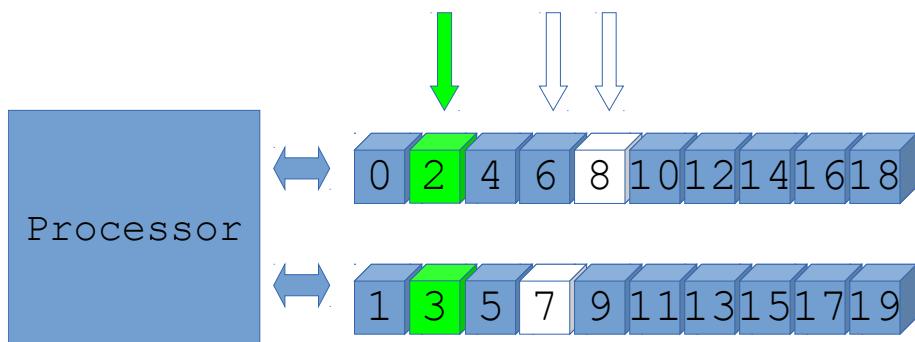
จากรูปแบบแรก (a) เป็นการต่อแบบความกว้างสายสัญญาณเป็นหนึ่ง ซึ่งกรณีนี้หากค่าที่ต้องการใช้เป็น half word (ใช้ทิล 2 ไบต์) หมายความว่าการอ่านข้อมูลจากหน่วยความจำจะต้องเกิดขึ้นสองครั้งจึงจะทำการประมวลผลได้หนึ่งครั้ง¹⁷ ในกรณีต่อไปนี้จะต้องทำ memory alignment เพราะสามารถเข้าถึงข้อมูลได้เพียงทิล 1 ไบต์ แต่ต้องอาศัยการอ่านหลายครั้งจึงจะได้ชุดข้อมูลครบตามที่เริ่มต้นต้องการ

ในแบบที่สอง (b) เป็นการขยายทางเดินข้อมูลให้กว้างขึ้นเป็นสองไบต์ หมายความว่า หน่วยประมวลผลสามารถอ่านข้อมูลได้มากขึ้นเป็นสองเท่า การอ่านข้อมูลเพื่อจะนำเข้าสู่เรจิสเตอร์ จะใช้เวลาอยู่สอง (เมื่อเทียบกับแบบ a) อย่างไรก็ตาม หากข้อมูลที่ต้องการอ่านอยู่คนละ block เช่น ต้องการอ่านข้อมูลที่เก็บอยู่ในเดาที่อยู่ 7 และ 8 เข้ามาพร้อมกันย่อมเป็นไปไม่ได้ เพราะตัวชี้เลขที่อยู่ (จากด้านบน) จะเป็นเลขคนละชุดกัน การอ่านข้อมูลในลักษณะนี้ จะ

¹⁷ ในหน่วยประมวลผลกลางรุ่นก่อนๆ แม่คircuitสร้างภายในจะสามารถประมวลผลข้อมูลได้เร็ว แต่ใช้ทางเดินข้อมูลที่แคบเพื่อให้สามารถใช้กับอุปกรณ์และ mother board รุ่นก่อนได้ เช่น Intel 80386SX ซึ่งเป็นสถาปัตยกรรมแบบ 32 บิต แต่ใช้ทางเดินข้อมูลแบบ 16 บิต เพื่อให้สามารถใช้อุปกรณ์ต่อพ่วงของ Intel 80286 ซึ่งเป็นหน่วยประมวลผลกลางรุ่นก่อนหน้าได้ ในทำนองเดียวกับ Intel 8088 ซึ่งเป็นหน่วยประมวลผลกลางแบบ 16 บิตแบบเดียวกับ Intel 8086 แต่ใช้ทางเดินข้อมูลแบบ 8 บิต เพื่อประหยัดค่าใช้จ่าย

ต้องอ่านสองครั้งและนำข้อมูลมาเรียงลำดับใหม่อยู่ดี (no alignment)

ด้วยเหตุนี้ หากสถาปัตยกรรมชุดคำสั่งบังคับให้ขอฟ์แวร์ต้องอ่านข้อมูลที่บังคับเลขที่อยู่เริ่มต้น (ทำ memory alignment) เสมอ จะช่วยให้การอ่านข้อมูลสามารถอ่านได้เร็วขึ้น เช่น ทุกครั้งที่ประกาศตัวแปรที่มีความยาว 2 ไบต์ เลขที่อยู่เริ่มต้นของตัวแปรดังกล่าวจะต้องหารด้วยสองลงตัวเพื่อให้อ่านและเขียนได้อย่างรวดเร็ว รูปที่ 3.8 แสดงการอ่านค่าแบบมีการบังคับเลขที่อยู่เริ่มต้น (restricted alignment) ในไบต์ที่ 2 และ ไบต์ที่ 3 และการอ่านค่าแบบไม่บังคับเลขที่อยู่เริ่มต้น (unrestricted alignment) ในไบต์ที่ 7 และไบต์ที่ 8



รูปที่ 3.8: การอ่านค่าแบบมี restricted alignment และ แบบ unrestricted alignment

ข้อเสียที่เห็นได้ชัดของการที่สถาปัตยกรรมบังคับให้มีการทำเลขที่อยู่เริ่มต้นคือเรื่องของ การแตกกระจาย (fragmentation) กล่าวคือ มีหน่วยความจำส่วนที่ว่างอยู่ แต่ไม่สามารถใช้ได้ เพราะจะต้องทำการบังคับให้เลขที่อยู่เริ่มต้นหารด้วยเลขที่กำหนดลงตัว ดูตัวอย่าง alignment ของคอมไพล์เตอร์สำหรับภาษาซีแบบ 32 บิตที่พับได้ทั่วไปตัวแปรแบบ int, long, float มีขนาด 4 ไบต์ ส่วน double มีขนาด 8 ไบต์ หากต้องการประกาศตัวแปร long 1 ชุด , int 2 ชุด และ double 1 ชุด ผลที่ได้ในสถาปัตยกรรมแบบบังคับเลขที่อยู่เริ่มต้นและ ไม่บังคับเลขที่อยู่เริ่มต้นจะเป็นดังรูปที่ 3.9 ข้อสังเกตคือ บนสถาปัตยกรรมที่บังคับเลขที่อยู่เริ่มต้น ข้อมูลแบบ 8 ไบต์ จะต้องเริ่มต้นที่เลขที่อยู่เริ่มต้นซึ่งหารด้วยแปดลงตัว ส่วนข้อมูลที่มีขนาด 4 ไบต์ จะต้องเริ่มต้นที่เลขที่อยู่ซึ่งหารด้วยสิบสองตัว เป็นต้น

```
long a;
int b;
int b1;
double c;
```

00	long a
04	int b
08	int b1
0C	
10	double c
14	
18	
1C	

(a)

00	long a
04	int b
08	int b1
0C	
10	
14	
18	
1C	

(b)

00	long a
04	int b
08	int b1
0C	
10	
14	double c
18	
1C	

(c)

รูปที่ 3.9: โครงสร้างการจัดวางหน่วยความจำ (word addressing แสดงในเลขฐาน 16) (a) โปรแกรมภาษาซี (b) สถาปัตยกรรมแบบ unrestricted alignment และ (c) สถาปัตยกรรมแบบ restricted alignment

3.4 การอ้างอิงเลขที่อยู่หน่วยความจำ (Addressing Mode)

การอ้างอิงเลขที่อยู่หน่วยความจำของ LADA แบ่งออกเป็นแบบ displacement (บางตำแหน่งเรียกชื่อเต็มว่า base displacement) และ Immediate เป็นหลัก ในแบบ displacement การอ้างอิงเลขที่อยู่จะประกอบด้วยจำนวนเต็มกับ rejisเตอร์เสมอ เช่น

```
LD      $r0, 32($r2)
```

32(\$r2) จะนำค่าใน rejisเตอร์ \$r2 มาบวกกับ 32 ก่อนจึงใช้อ้างอิงเลขที่อยู่ หากข้อมูลใน rejisเตอร์ ที่ \$r2 เป็น 300 คำสั่งดังกล่าวจะนำข้อมูลในเลขที่อยู่ 332 มาเก็บไว้ใน rejisเตอร์ \$r0

ส่วนการอ้างอิงแบบ Immediate คือการนำข้อมูลไปใช้ทันที โดยไม่จำเป็นต้องอ่านข้อมูลจากหน่วยความจำหรือ rejisเตอร์ (เบรียบเทียบได้กับค่าคงที่ที่ระบุในโปรแกรม) เช่น

```
ADD      $r2,$r1,#4
```

ห้ารบ #4 ในคำสั่ง ADD มีความหมายว่า ให้นำค่า 4 มาบวกกับค่าใน rejisเตอร์ที่กำหนด แล้วนำผลลัพธ์ไปเก็บที่ \$r2

ในสถาปัตยกรรมแบบอื่น อาจมีการอ้างอิงเลขที่อยู่หน่วยความจำที่ซับซ้อนมากกว่านี้ เช่นระบบ base index addressing หรือระบบ base index displacement ซึ่งสามารถอ้างอิง

ข้อมูลในหน่วยความจำได้หลากหลายรูปแบบ เพื่อประกอบความเข้าใจ ลองดูการอ้างอิงเลขที่อยู่หน่วยความจำแบบต่าง ๆ ดังแสดงในตารางที่ 3.1

ตารางที่ 3.1: การอ้างอิงเลขที่อยู่แบบต่าง ๆ และการเทียบเคียงกับภาษาเรขาคณิตสูง

Addressing Mode	ตัวอย่าง ในภาษา แอสเซมบลี	RTL	เปรียบกับ ภาษาซี
1. Immediate	LDI \$r1, #4	\$r1 \leftarrow 4	A = 4
2. Register Direct	ADD \$r1, \$r2, \$r3	\$r1 \leftarrow \$r2 + \$r3	A = B + C
3. Displacement	LD \$r1, 100(\$r2)	\$r1 \leftarrow MEM[100+\$r2]	A = D[100] หรือ A = G.age
4. Register Indirect	LD \$r1, (\$r2)	\$r1 \leftarrow MEM[\$r2]	A = *p
5. Index	LD \$r1, (\$r2 + \$r3)	\$r1 \leftarrow MEM[\$r1 + \$r2]	A = D[i]
6. Direct	LD \$r1, 1000	\$r1 \leftarrow MEM[1000]	
7. Memory Indirect	LD \$r1, @(\$r2)	\$r1 \leftarrow MEM[MEM[\$r2]]	
8. AutoIncrement	LD \$r1, (\$r2)+	\$r1 \leftarrow MEM[\$r2] \$r2 \leftarrow \$r2 + 4	A = D[i++]
9. AutoDecrement	LD \$r1, (\$r2)-	\$r1 \leftarrow MEM[\$r2] \$r2 \leftarrow \$r2 - 4	A = D[i--]
10. Scale	LD \$r1, (\$r2)[\$r3*4]	\$r1 \leftarrow MEM[\$r2+\$r3*4]	F = E[i]

กำหนดให้ A, B, C เป็นตัวแปรเป็นประเภท unsigned char, D เป็น array ของ char , E เป็น array ของ int, F เป็น int (1 int มีขนาด 4 ไบต์), p เป็น ตัวชี้ (pointer), i เป็น

index ของ array และ G เป็นโครงสร้าง (structure) ที่มีเขตข้อมูลอยู่ {char name[100]; unsigned char age;} (กรณีนี้ age จากจุดเริ่มต้น 100 ไบต์)

ทั้งนี้การอ้างอิงเลขที่อยู่หน่วยแบบที่แสดงในตารางจะไม่ค่อยพบในสถาปัตยกรรมสมัยใหม่ เนื่องจากหน่วยแบบถูกแทนที่ได้ด้วย displacement หรือ register indirect เมื่อมีการคำนวณเลขที่อยู่ที่ต้องการอ้างอิงใส่ไว้ในเรจิสเตอร์ก่อน (โปรแกรมที่ไม่มีจำนวนคำสั่งมากขึ้น เพื่อชดเชยกับรูปแบบการอ้างอิงเลขที่อยู่ที่ขาดหายไป)

นอกจากนี้ยังมีระบบอ้างอิงเลขที่อยู่หน่วยความจำแบบ PC relative ซึ่งมีการทำงานคล้ายกับ displacement แต่ใช้ PC เป็นเรจิสเตอร์อ้างอิงแทน การทำงานในแบบ PC relative นักใช้กับคำสั่งแก้คุณประโยชน์เช่น jump หรือ branch (เบร์ยบเทียบได้กับ GOTO, IF .. THEN .. ELSE, และ SWITCH)

ข้อสังเกต

- หากกำหนดค่า displacement เป็น 0 ผลที่ได้จะเหมือน register indirect เช่น LD \$r1, 0(\$r2) จะเหมือนกับ LD \$r1, (\$r2)
- หากทำการคำนวณค่าของเลขที่อยู่ที่ต้องการใส่ในเรจิสเตอร์ก่อน จะสามารถใช้การอ้างอิงเลขที่อยู่แบบ register indirect แทนได้ทุกแบบ
- ในแบบ scale เนื่องจาก E เป็น int ซึ่งมีขนาด 4 ไบต์ ดังนั้น ค่า i ซึ่งเป็น index จะต้องคูณ 4 เสมอจึงจะได้เลขที่อยู่ที่ต้องการ

แม้การมีวิธีการอ้างอิงเลขที่อยู่เพียงไม่กี่แบบ แล้วต้องอาศัยคอมไพเลอร์ และ การคำนวณที่มากขึ้น และผลที่ได้ในทางหนึ่งคือ คำสั่งมีความซับซ้อนน้อยลง และสามารถประมวลผลแบบขนานได้มากขึ้น (ซึ่งจะกล่าวถึงในเนื้อหาส่วนต่อไป)

3.5 รูปแบบคำสั่ง (Instruction Format)

คำสั่งภาษาแอสเซมบลีทุกคำสั่งจะถูกแปลเป็นภาษาเครื่องโดยภาษาเครื่องจะมีรูปแบบคำสั่งที่เป็นแบบแผนของบิต เรียกว่ารูปแบบคำสั่ง (instruction format) เนื่องจากภาษาเครื่องนั้นจะประกอบด้วยสายข้อมูลบิต ซึ่งมีเพียง 0 และ 1 มาต่อกันเท่านั้น รูปแบบคำสั่งที่แปลจากภาษาแอสเซมบลีเป็นภาษาเครื่องจะใช้การแบ่งสายข้อมูลบิตออกเป็นช่วง แล้วกำหนดความหมายให้แต่ละช่วงบิต

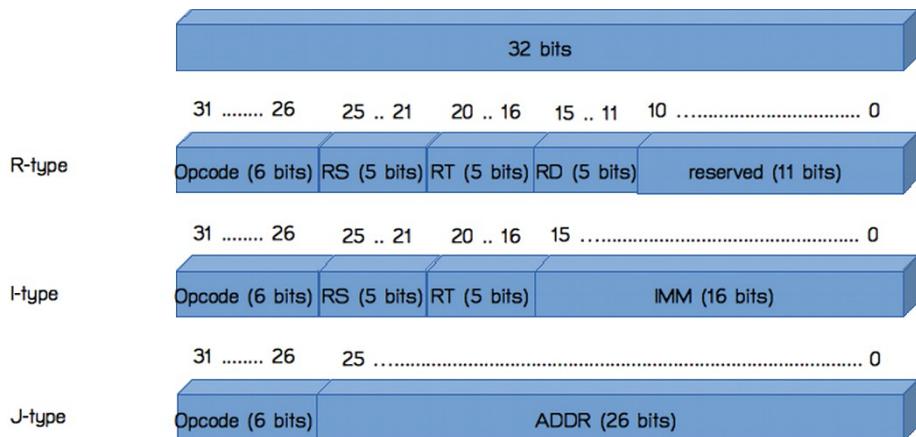
กรณีของ LADA คำสั่งทุกคำสั่งมีความยาว 32 บิตเท่ากันหมด สถาปัตยกรรมลักษณะนี้คือสถาปัตยกรรมที่มีความยาวคำสั่งแบบคงที่ (fixed instruction length¹⁸) สถาปัตยกรรมบางกลุ่ม (เช่น Intel x86) มีความยาวคำสั่งแตกต่างกันไปในแต่ละคำสั่งดังนั้น 8 – 17 ไบต์ สถาปัตยกรรมลักษณะนี้เรียกว่า สถาปัตยกรรมที่มีความยาวคำสั่งแบบผันแปร (variable instruction length¹⁹) นอกจากนี้ยังมีสถาปัตยกรรมบางรูปแบบที่มีความยาวคำสั่งแบบลูกผสม (hybrid instruction length)²⁰ โดยจะมีโหมดการทำงานพิเศษที่มีความยาวคำสั่งแตกต่างกันไป

ในสถาปัตยกรรม LADA มีรูปแบบคำสั่งทั้งสิ้น 3 รูปแบบเท่านั้น คือ แบบ R, I และ J (ขออ้างอิงถึงโดยเรียกว่า R-type, I-type และ J-type ตามลำดับ) แต่ละรูปแบบจะมีการตีความลำดับของบิตต่างกันไป ดังนี้

- R-type สำหรับการเข้ารหัสคำสั่งที่เกี่ยวกับเรจิสเตอร์มี field ที่เกี่ยวข้องได้แก่ op, rs, rt, และ rd
- I-type สำหรับการเข้ารหัสคำสั่งที่มีการอ้างอิงค่า immediate มี field ที่เกี่ยวข้องได้แก่ op, rs, rt และ immediate
- J-type สำหรับการ jump หรือ branch แบบ absolute เป็นหลัก

รายละเอียดรูปแบบคำสั่งที่พับใน nanoLADA แสดงได้ดังรูปที่ 3.10

- 18 สถาปัตยกรรมที่มีรูปแบบคำสั่งแบบคงที่ (fixed length instruction) สามารถอ่านคำสั่งได้ภายใน 1 รอบการประมวลผล ทำให้อ่านแต่ละคำสั่งในเวลาคงที่ ตัวอย่างสถาปัตยกรรมลักษณะนี้ เช่น PowerPC, MIPS, ARM, SPARC
- 19 สถาปัตยกรรมที่มีรูปแบบคำสั่งแบบผันแปร (variable length instruction) การอ่านคำสั่งแต่ละขั้นจะใช้จำนวน cycle ไม่คงที่ ส่วนแรกของคำสั่งที่อ่านได้จะมีบิตที่ใช้บอกความยาวของคำสั่งทั้งหมด ตัวอย่างสถาปัตยกรรมลักษณะนี้ เช่น Intel x86, VAX
- 20 สถาปัตยกรรมที่มีรูปแบบคำสั่งแบบผสม (hybrid length instruction) เป็นสถาปัตยกรรมแบบ fixed length instruction ที่มีโหมดการทำงานพิเศษที่รับคำสั่งที่มีความยาวเฉพาะ เช่น ARM Thumb เมื่อทำงานในโหมด thumb หน่วยประมวลผลจะทำการประมวลคำสั่งที่มีความยาวแบบ 16 บิตแทน (ปกติ ARM instruction จะมีความยาว 32 บิต)



รูปที่ 3.10: รูปแบบคำสั่ง (instruction format) ในสถาปัตยกรรม nanoLADA

ทั้งนี้การเข้ารหัสคำสั่ง จึงอยู่กับผู้ออกแบบแต่ละรายจะกำหนด (ดูรายละเอียดเพิ่มเติมได้จากคู่มือโครงสร้างชุดคำสั่งของแต่ละสถาปัตยกรรม) เช่น คำสั่ง ADD จากแทนด้วย 000001 คำสั่ง SUB จากแทนด้วย 000010 โดยหลักเกณฑ์ในการกำหนดรหัสต่างนั้น จึ้นอยู่กับแนวทางในการออกแบบจรเพื่อตีความคำสั่งและสร้างสัญญาณควบคุม

3.6 สถาปัตยกรรมชุดคำสั่งที่ดี

ปัจจุบันมีสถาปัตยกรรมชุดคำสั่งที่ใช้อยู่แพร่หลาย หากวิเคราะห์สถาปัตยกรรมชุดคำสั่งที่มีในปัจจุบัน (ปี พ.ศ. 2557) สถาปัตยกรรมชุดคำสั่งที่เห็นรอบตัว เข่น สถาปัตยกรรม Intel x86_64 ซึ่งพับในเครื่องคอมพิวเตอร์ส่วนบุคคลทั่วไปและเครื่องตระกูล Macintosh ของ Apple สถาปัตยกรรม ARM ที่มีการใช้งานแพร่หลายในโทรศัพท์เคลื่อนที่แบบ smart phone และ สถาปัตยกรรมแบบ AVR ที่มีการใช้งานใน Arduino หรือระบบคอมพิวเตอร์แบบฝังตัวส่วนใหญ่ ที่ยกตัวอย่างมาเนี้เป็นเพียงส่วนหนึ่งเท่านั้น

หากจะต้องวิเคราะห์หรือซึ้งด้วง สถาปัตยกรรมชุดคำสั่งได้มีข้อดีข้อเสีย ควรจะมีหลักเกณฑ์ในการประเมินอย่างไรบ้าง หลักในการจะบอกว่าสถาปัตยกรรมชุดคำสั่งใดเป็นสถาปัตยกรรมชุดคำสั่งที่ดีนั้น มีแนวคิดที่หลากหลาย แต่ละตำรามักให้àng มุนในการวิเคราะห์ที่แตกต่างกัน อย่างไรก็ตามจากการศึกษาจากหลายตำรา พยายจะสรุปเป็นหลักการได้ดังนี้

- มีความเข้ากันได้กับสถาปัตยกรรมรุ่นก่อนหน้า (compatibility) กล่าวคือ มีการใช้งานมานาน ตัวอย่างที่เห็นได้ชัด เช่น Intel x86 ซึ่งมีอายุเก่าแก่กว่า 40 ปี ยังคงทำงานได้บนสถาปัตยกรรมรุ่นใหม่ของ Intel
- มีความครอบคลุม (generality หรือ completeness) กล่าวคือ มีคำสั่งเพียงพอสำหรับการใช้งานทั่วไป เช่น ในยุคหนึ่งหน่วยประมวลผลกลางหรือสถาปัตยกรรมชุดคำสั่งหลักจะไม่มีคำสั่งเกี่ยวกับการคำนวณเลข浮นิยม (floating-point unit) ทำให้การทำงานที่เกี่ยวข้องกับกราฟิกส์เป็นไปได้ยาก เป็นต้น
- มีการทำงานทั่วถึง (orthogonal) กล่าวคือ มีการทำงานที่ซัดเจน ไม่มีกลุ่มเรจิสเตอร์พิเศษหรือความหมายพิเศษมากเกินจำเป็น
- ช่วยให้แปลงโปรแกรมได้ง่าย (ease of compilation) ทั้งนี้เนื่องจากสถาปัตยกรรมชุดคำสั่งคือมุ่งมองของหน่วยประมวลผลกลางจากผู้พัฒนาซอฟต์แวร์ ดังนั้นคุณสมบัติอย่างหนึ่งของหน่วยประมวลผลกลางคือ สามารถพัฒนาซอฟต์แวร์ได้ง่าย
- ช่วยให้สามารถพัฒนาฮาร์ดแวร์ได้ง่าย (ease of implementation) ในทำนองเดียวกัน เนื่องจากฮาร์ดแวร์จะต้องตอบสนองการทำงานของสถาปัตยกรรมชุดคำสั่ง ดังนั้นสถาปัตยกรรมชุดคำสั่งที่ดีจึงควรทำให้ฮาร์ดแวร์สามารถทำงานได้ง่าย ยิ่งไปกว่านั้น ปัจจุบันฮาร์ดแวร์มักมีความเป็นสถาปัตยกรรมเชิงขนาดที่สามารถทำงานได้翰翰คำสั่งพร้อมกัน ซึ่งสถาปัตยกรรมชุดคำสั่งที่ดี จะช่วยให้การประมวลผลแบบขนาน (streamlined) ทำได้ง่ายขึ้นด้วย

อย่างไรก็ตาม นี่เป็นเพียงคุณสมบัติเบื้องต้นเท่านั้น หากจะขยายความสถาปัตยกรรมที่มีความสำคัญในการพัฒนาซอฟต์แวร์ อาจจะต้องวิเคราะห์ต่อว่าคอมไพล์เตอร์ หรือตัวแปลงภาษาที่มีการทำงานอย่างไร และสถาปัตยกรรมลักษณะไหนที่จะช่วยให้ทำการแปลงได้ง่ายขึ้น ซึ่งขั้นตอนการทำงานของคอมไابل์เตอร์นั้น อยู่นอกเหนือขอบเขตของหนังสือเล่มนี้ อย่างไรก็ตาม จะขออธิบายบทบาทของคอมไابل์เตอร์เพื่อให้ผู้อ่านเข้าใจถึงประโยชน์ที่ได้จากการคอมไابل์เตอร์ (ที่ฉลาด) ในหัวข้อถัดไป

3.6.1 บทบาทของคอมไابل์เตอร์ในการหาค่าเหมาะสมที่สุด

เนื่องจากสมรรถนะของระบบคอมพิวเตอร์ส่วนหนึ่ง ขึ้นกับความสามารถในการคำนวณที่ดีที่สุด (optimization) ของคอมไابل์เตอร์ ผู้ออกแบบสถาปัตยกรรมชุดคำสั่งควรเข้าใจในบทบาทของคอมไابل์เตอร์เพื่อเป็นข้อมูลประกอบการออกแบบสถาปัตยกรรมชุดคำสั่งที่ดี เนื้อหาที่ระบุในส่วนนี้เป็นเพียงข้อมูลเบื้องต้นเพื่อประกอบการศึกษาเท่านั้น ในทางปฏิบัติการ

หาค่าที่เหมาะสมที่สุดมีวัตถุประสงค์ 2 ประการ (ให้เลือกอย่างใดอย่างหนึ่ง) คือ 1) เพื่อให้ซอฟต์แวร์ที่ได้ทำงานเร็วขึ้น หรือ 2) เพื่อให้ซอฟต์แวร์ที่ได้มีขนาดเล็กลง ซึ่งทั้งสองอย่าง มักให้ผลตรงข้ามกันเสมอ (ซอฟต์แวร์ที่เร็วขึ้นมักจะมีขนาดใหญ่ด้วย)

เพื่อให้ซอฟต์แวร์มีสมรรถนะสูงขึ้น ขั้นตอนหนึ่งในการทำงานของคอมไ派出เรอร์คือ การหาค่าที่เหมาะสมที่สุดของโปรแกรม การค่าที่เหมาะสมที่สุดแบ่งออกได้เป็นสองลักษณะคือ การค่าเหมาะสมที่สุดแบบห้องถิน (local optimization) และ การค่าเหมาะสมที่สุดแบบบรวม (global optimization)

ก่อนจะทำความเข้าใจเรื่องการหาค่าที่เหมาะสมที่สุด ต้องเข้าใจก่อนว่าในคอมไ派出เรอร์ โปรแกรมจะถูกแบ่งออกเป็นส่วนย่อยเรียกว่า block ส่วนที่เล็กสุดของการทำงานเรียกว่า basic block ซึ่งคือ block ที่มีทางเข้า(เริ่มทำงาน) เพียงทางเดียวคือด้านบน และทางออก เพียงทางเดียวคือด้านล่าง นั่นหมายความว่าทุกครั้งที่มีการเปลี่ยนลำดับการทำงาน (เช่น ด้วยประโยคเงื่อนไข IF.. THEN .. ELSE) หรือเรียกโปรแกรมย่อๆ เป็นอันสิ้นสุด basic block.

จากคำจำกัดความของ basic block ช่วยให้อธิบายความหมายของ การหาค่าที่เหมาะสมที่สุดแบบห้องถินและการค่าที่เหมาะสมที่สุดแบบบรวมได้ว่า การหาค่าที่เหมาะสมที่สุดแบบห้องถิน คือการปรับปรุงคำสั่งภายใน basic block เดียวกันเท่านั้น การหาค่าที่เหมาะสมที่สุดแบบบรวม คือการปรับปรุงคำสั่งระหว่าง basic block

ด้วยการหาค่าที่ดีที่สุดแบบห้องถิน

1. **Common expression elimination** ตัวอย่าง เช่น มี $b+c$ อยู่ในหลายคำสั่ง ดังนั้น หากคำนวน $b + c$ ไว้ก่อนอาจทำให้การคำนวนเร็วขึ้น (เช่น $a = b + c + d$; และ $y = b + c + e$; หากทำ $x = b + c$ ไว้ก่อน จะได้ว่า $a = x + d$; และ $y = x + e$; ซึ่งทำงานได้เร็วกว่า)
2. **Constant propagation** ค่าบางอย่างสามารถแทนที่ได้ด้วยค่าคงที่ ก็ควรจะแทนที่ไปในระหว่างทำการแปลทันที (เช่น $m = 60$; $h = 60 * 60$; $d = 24*m$; $days = seconds / d$; จะเห็นว่าเป็นการแปลงวินาทีเป็นเวลา หากแต่ตัวแปร m , h และ d ไม่มีการใช้งานที่อื่นอีก จึงแทนที่ประโยชน์ทั้งหมดได้ด้วย $days = seconds / 86400$; เป็นต้น)

ตัวอย่างการหาค่าที่ดีที่สุดแบบรวม (global optimization)

1. **Global common expression elimination** ทำนองเดียวกันกับ common expression elimination แบบ local หากแต่เป็นการทำระหว่าง basic block
2. **Code motion** ค่าบางอย่างอาจไม่เกิดการเปลี่ยนแปลงใน loop (เรียกว่า loop invariant) การย้ายค่าดังกล่าวไปคำนวณก่อนเข้า loop จะช่วยให้การคำนวณใน loop แต่ละรอบน้อยลง และ code ทำงานได้เร็วขึ้น

นอกจากนี้ ยังมีการหาค่าที่ดีที่สุดแบบอื่นที่เกี่ยวข้องกับเครื่องหรือสถาปัตยกรรมชุดคำสั่งที่ใช้ (machine-dependent optimization) เพื่อช่วยให้สามารถทำงานได้เร็วขึ้น เช่น การแทนที่การคูณด้วยคำสั่ง shift เป็นต้น

ประเด็นช่วยจำ ในภาคผนวก ค เป็นกิจกรรมที่จะให้ผู้เรียนได้ทดลองปรับค่าคอมไฟเลอร์ และคุณลักษณะของสถาปัตยกรรมที่ดีที่สุดแบบต่าง ๆ เมื่อได้คุณลักษณะใน กิจกรรมจะช่วยให้เข้าใจมากขึ้น

หากจะวิเคราะห์สิ่น ๆ จะได้ว่า คุณลักษณะของสถาปัตยกรรมที่ดีจากมุมมองของคอมไฟเลอร์ คือ

1. มีความเรียบง่าย (simplicity) โดยสถาปัตยกรรมมีส่วนประกอบพื้นฐาน (primitive) ให้ จำนวนน้อยสามารถทำงานได้ทั้งหมด เนื่องจากมีความสามารถในการทำงานที่ซับซ้อนได้
2. มีความครอบคลุม (orthogonal หรือ regularity) ความหมายของแต่ละ field ใน การเข้ารหัสคำสั่ง ตรงไปตรงมา ไม่มี mode พิเศษ
3. มีแนวทางในการเลือกใช้ที่ชัดเจน (simplify tradeoffs) ในบางกรณี เราสามารถจะ ทำงานให้สำเร็จได้มากกว่า 1 วิธี เช่น การกำหนดให้ตัวแปรมีค่าเป็นศูนย์ ทำได้ โดยการใช้คำสั่ง CLR หรือ XOR ข้อมูลเดิมก็ได้ ซึ่งสถาปัตยกรรมที่ดี ควรมีตัว เลือกที่ชัดเจน
4. อนุญาตให้กำหนดค่าตอนแปลภาษา (compile-time binding) ได้ เช่น ในขณะที่ แปลภาษาอาจทราบได้ทันทีว่าจะต้องนำค่าคงที่เข้าคูณกับตัวแปรเพื่อให้ได้ผลลัพธ์ แต่หากสถาปัตยกรรมไม่รองรับการทำงานของค่า immediate ย่อมเป็นไปไม่ได้ที่ จะระบุค่าคงที่ดังกล่าวเข้าไปในโปรแกรม เป็นต้น

3.7 การเรียกใช้โปรแกรมย่อย และ *Exception*

ประเดิมหนึ่งที่สำคัญแต่มิได้เป็นส่วนหนึ่งของสถาปัตยกรรมชุดคำสั่งโดยตรงคือ การเรียกใช้งานโปรแกรมย่อย ซึ่งเป็นส่วนหนึ่งของคำสั่งควบคุม ข้อสำคัญที่ทำให้การเรียกใช้โปรแกรมย่อยแตกต่างจากคำสั่งทั่วไปคือ การต้องจำเลขที่อยู่สำหรับการ `return` มาบังโปรแกรมที่เรียก ซึ่งสถาปัตยกรรมชุดคำสั่ง มักจะมีตัวช่วยในการเรียกโปรแกรมย่อยเป็นคำสั่ง `call` และ `return` ซึ่งอาจมีการทำงานดังนี้

<code>CALL addr</code>	<code>; (J-type)</code>
-----------------------------	-------------------------

ความหมาย	<code>\$r13 ← PC + 4;</code>
----------	------------------------------

PC	<code>← addr;</code>
----	----------------------

และ

<code>RET</code>	<code>; (J-type)</code>
------------------	-------------------------

ความหมาย	<code>PC ← \$r13;</code>
----------	--------------------------

ในการณีของสถาปัตยกรรมชุดคำสั่ง LADA ค่า `$r13` เป็นเรจิสเตอร์ที่ถูกกำหนดให้เก็บค่า `return address` ดังนั้นทุกครั้งที่มีการใช้คำสั่ง `CALL` ค่า `return address` จะถูกนำมารักษาไว้ในเรจิสเตอร์นี้ สถาปัตยกรรมชุดคำสั่งบางแบบอาจมีเรจิสเตอร์พิเศษอันอื่นเพื่อทำหน้าที่ดังกล่าว และในสถาปัตยกรรมบางแบบจะเลือกให้หน่วยความจำเป็นที่พัก `Return Address` แทน (เช่น Intel x86) ซึ่งการเลือกตัดสินใจว่าจะใช้แบบใด ย่อมส่งผลต่อสมรรถนะโดยรวมของระบบเข่นกัน

ข้อควรคิด ในกรณีที่ใช้เรจิสเตอร์ในการเก็บ `return address` หากต้องการเรียกโปรแกรมย่อยต่อไปหรือต้องการทำโปรแกรมเรียนเรียกตัวเอง (recursive program) จะต้องทำอย่างไรเพื่อให้สามารถจดจำเลขที่อยู่ได้เมื่อนัดมี

3.7.1 การส่งผ่านและคืนค่าระหว่างโปรแกรมย่อย

นอกจากนี้ ยังมีสิ่งที่ซอฟต์แวร์ต้องการทำในกรณีเรียกใช้โปรแกรมย่อยคือ 1) การส่งผ่านค่า arguments หรือ parameters และ 2) การคืนค่า ซึ่งมีได้มีส่วนได้ล่าไห้ในสถาปัตยกรรมชุดคำสั่งว่า จะต้องส่งผ่านค่าและคืนค่าอย่างไรให้เป็นมาตรฐาน ผู้ออกแบบสถาปัตยกรรมชุดคำสั่งจึงมักจะทำคู่มืออีกชุดหนึ่งเรียกว่า Application Binary Interface (ABI) เพื่อ

กำหนดว่าจะผ่านค่าและคืนค่าอย่างใด (เดิมมาตราฐานนี้จะกำหนดโดยระบบปฏิบัติการ และ/หรือ คอมไප์เลอร์แทน)

ลักษณะการส่งผ่านและคืนค่านั้น แบ่งเป็นกลุ่มใหญ่ ๆ ได้ดังนี้

1. การส่งผ่านด้วยเรจิสเตอร์ กรณีนี้ ABI จะต้องกำหนดมาตรฐานว่า เรจิสเตอร์ใดใช้สำหรับการคืนค่า และเรจิสเตอร์ใด (เรียงลำดับอย่างไร) ใช้สำหรับการผ่านค่า ในลำดับอย่างไร อย่างไรก็ตามเรจิสเตอร์มักมีข้อจำกัดในการส่งข้อมูลขนาดเล็ก หากต้องส่งข้อมูลขนาดใหญ่ จะต้องนำข้อมูลมาใส่ในหน่วยความจำ แล้วส่งผ่านเลขที่อยู่ของหน่วยความจำในเรจิสเตอร์แทน
2. การส่งผ่านด้วยหน่วยความจำ กำหนดว่า ก่อนการเรียกใช้โปรแกรมย่อย จะต้องทำการ push ค่า arguments ลงใน stack และเว้นที่ไว้สำหรับการรับค่าคืน ก่อนจะที่มีการเรียกโปรแกรมย่อย (ซึ่งลักษณะนี้เป็นลักษณะที่นิยมใช้โดยทั่วไปสำหรับ Intel x86)

ขอให้ผู้เรียนลองทำแบบฝึกหัดท้ายบทเพิ่มเติมเพื่อจะได้เห็นการส่งผ่านในโปรแกรมจริง

3.7.2 การบันทึกค่าตัวแปรท้องถิ่นและเรจิสเตอร์

เนื่องจากในทุกโปรแกรมย่อย จะมีการใช้งานตัวแปรท้องถิ่น (local variable) และเรจิสเตอร์ ดังนั้นเพื่อให้สามารถกลับมาทำงานได้เหมือนเดิม จึงต้องมีการจำค่าดังกล่าวไว้ด้วย ตัวอย่าง เช่น โปรแกรมหลักมีการใช้งาน \$r1 แทน a และ \$r2 แทน b อยู่ หากเรียกโปรแกรมย่อยที่ มีการใช้งาน \$r1 แทน x ก่อนเรียกโปรแกรมย่อยนี้ จะต้องบันทึกค่า a เก็บไว้ก่อน และ หลังจากโปรแกรมย่อยดีนการทำงานมาแล้วจึงจะนำค่า a กลับมาที่ \$r1

ประเด็นเพื่อการพิจารณามีอยู่ว่า หน้าที่ในการจำค่าเรจิสเตอร์ที่อาจมีการเปลี่ยนแปลงในระหว่างการทำงานนั้นเป็นของใคร ระหว่างโปรแกรมหลักหรือผู้เรียก (caller) และ โปรแกรมย่อย หรือผู้ถูกเรียก (callee) อันนี้เป็นอีกประเด็นที่ต้องระบุไว้ใน ABI เช่นกัน ยกตัวอย่างประกอบการอธิบายดังรูปที่ 3.11 เพื่อให้เห็นข้อดีข้อเสียดังนี้

จากรูปที่ 3.11 จะเห็นว่า กรณีผู้เรียกเป็นผู้บันทึกค่า (caller save) จะต้องทราบว่าโปรแกรมย่อย (SUB) ใช้เรจิสเตอร์อะไรบ้าง ซึ่งหากผู้เรียกไม่ทราบก็จะต้องบันทึกค่าเรจิสเตอร์ทุกค่า (ในกรณีนี้คือ 32 ค่า) และหลังจากกลับมาจากการ调用 SUB จะต้องทำการนำค่าทั้ง 32 ค่ากลับมา ในทางกลับกัน หากผู้ถูกเรียกเป็นผู้บันทึกค่า (callee save) ผู้ถูกเรียกจะเลือกบันทึกเพียงค่าเรจิสเตอร์ที่ตนเองใช้ และเว้นค่าอื่นที่ไม่เกี่ยวข้องไว้ได้

จะเห็นว่ากรณีผู้ถูกเรียกเป็นผู้บันทึกค่า สมรรถนะที่ได้จะดี แต่สถาปัตยกรรมหลายอันก็เลือกให้ผู้เรียกเป็นผู้บันทึกแทน ส่วนหนึ่งอาจเพราะสถาปัตยกรรมชุดคำสั่งในอดีต มักมีเครื่องเตอร์จำนานน้อย ดังนั้นการจะเก็บค่าโดยผู้เรียก จึงมีได้ส่งผลกระทบต่อสมรรถนะของระบบมากนัก

Caller Save	Callee Save
ADD \$r1, \$r1, \$r2	ADD \$r1, \$r1, \$r2
PUSH \$r1	
CALL SUB	CALL SUB
POP \$r1	ADD \$r2, \$r2, \$r3
ADD \$r2, \$r2, \$r3	...
...	
SUB:	SUB:
	PUSH \$r1
ANI \$r1, \$r1, #00	ANI \$r1, \$r1, #00
ADD \$r1, \$r7, \$r8	ADD \$r1, \$r7, \$r8
	POP \$r1
RET	RET

รูปที่ 3.11: เปรียบเทียบการทำงานระหว่าง caller save และ callee save

3.7.3 Exception และ Interrupt

exception และ interrupt คือ การทำงานที่ไม่ได้เกิดจากโปรแกรมโดยตรง กล่าวคือ เป็นสิ่งที่เกิดขึ้นจากเหตุอื่นซึ่งทำให้หน่วยประมวลผลกลางต้องหยุดการทำงานชั่วคราว หลายตำรานิยมแยกคำโดยใช้ exception แทนสิ่งที่เกิดขึ้นจากการกระทำการของซอฟต์แวร์ (เช่น การหารด้วยศูนย์ การประมวลผลคำสั่งที่ไม่รู้จัก หรือ การอ่านข้อมูลที่ไม่มีอยู่จริง เป็นต้น) และเรียก interrupt ว่าเป็นสิ่งที่เกิดจากハードแวร์ (เช่น สัญญาณนาฬิกาของระบบมีการเปลี่ยนแปลง มีผู้กดแป้นพิมพ์หรือป้อนอินพุตให้ระบบ เป็นต้น)

จะเห็นว่าการทำงานของ exception หรือ interrupt คล้ายกับการทำงานของการเรียกโปรแกรมย่อย แต่ข้อแตกต่างคือ ต้องมีการกำหนดเลขที่อยู่ของโปรแกรมย่อยนั้น (เรียกว่า handler) ล่วงหน้าไว้ก่อน และก่อนการประมวลผล handler จะต้องมีการจำค่า สถานะต่าง ๆ เพิ่มเติมจากค่า PC ปัจจุบันด้วย และการจบ exception ต้องทำด้วยคำสั่งพิเศษคือ

IRET ซึ่งจะมีการคืนค่าสถานะเพิ่มเติมจากคำสั่ง RET ด้วยเช่นกัน ในทำนองเดียวกัน การเก็บค่าสถานะเหล่านี้ อาจเก็บใน rejister พิเศษ หรือเก็บไว้ใน stack memory ก็ได้ ทั้งนี้ แล้วแต่การออกแบบของสถาปัตยกรรมชุดคำสั่ง แต่ลักษณะแบบมีข้อดีข้อเสียคล้ายกับกรณีการเก็บ return address

วิธีการระบุเลขที่อยู่ของ handler มี 3 แนวทางคือ

- vector table เป็นการสร้างตารางเพื่อบอกว่าเลขที่อยู่ของโปรแกรมย่ออยสำหรับเป็น handler อยู่ที่ใด (ตารางเก็บเลขที่อยู่)
- handler table คล้ายกับวิธีการ vector table เพียงแต่ในตารางจะเป็น handler routine เลย เช่น exception 0 ไปยังเลขที่อยู่ 0x0000, exception 1 จะไปยังเลขที่อยู่ 0x0010 เป็นต้น (1 ช่องของตารางมีขนาด 16 ไบต์) (ตารางเก็บ code)
- fix entry ไม่มีตารางสำหรับจัดเก็บ handler แต่จะกำหนดเลขที่อยู่โดยตัวที่หน่วยประมวลผลกลางจะไปทำงานเมื่อมี exception หรือ interrupt เกิดขึ้น ดังนั้นการจัดการเรียก handler จะเป็นหน้าที่ของซอฟต์แวร์ สถาปัตยกรรมแบบนี้จะใช้ rejister เศรษฐ์พิเศษเพื่อบอกแหล่งที่มาของ exception ให้กับโปรแกรมได้ทราบอีกด้วย

ข้อขวนคิด ในกรณีของ handler table ตารางที่ใช้เก็บ handler จะมีขนาดเล็ก (เพียง 16 ไบต์) ซึ่งอาจจะเก็บคำสั่งได้เพียง 4 คำสั่งเท่านั้น หาก handler มีขนาดใหญ่มาก ผู้เรียนคิดว่า เราชมีแนวทางในการแก้ปัญหาอย่างไร

3.8 สรุป

สถาปัตยกรรมชุดคำสั่งเป็นมาตรฐานที่ใช้สื่อระหว่างการพัฒนา hardware และซอฟต์แวร์ ความซับซ้อนของคำสั่งเป็นเพียงปัจจัยหนึ่งในการออกแบบเท่านั้น ทั้งนี้หากคำสั่งมีความซับซ้อนมากขึ้น อาจจะส่งผลให้คำสั่งใช้รอบการประมวลผลต่อคำสั่งมากขึ้น (เนื่องจากการจรที่มีขนาดใหญ่และมีสถานะรวมถึงขั้นตอนการทำงานมากขึ้น) ทำให้ห้องจัดเก็บล่าวไม่สามารถใช้ความถี่สัญญาณนาฬิกาที่มีความถี่สูง ปัจจุบันยังไม่มีมาตรฐานใดในการกำหนดว่า สถาปัตยกรรมชุดคำสั่งนั้นต้องการทำงานมากแค่ไหน (เนื่องจากความต้องการของผู้ใช้งาน เช่น ในบางกรณีผู้ใช้งานและผู้ออกแบบอาจจะเลือกความเข้ากันได้ เพื่อให้หน่วยประมวลผลรุ่นใหม่สามารถประมวลผลชุดคำสั่งเดิมได้ ช่วยให้ซอฟต์แวร์เดิมสามารถทำงานบนหน่วยประมวลผลรุ่นใหม่ได้ทันที ในขณะที่บางครั้งผู้ออกแบบเลือกที่จะทิ้งความเข้ากันได้ เพื่อให้ได้สมรรถนะที่ดีกว่าเป็นต้น)

3.9 แบบฝึกหัดท้ายบท

1. สถาปัตยกรรมชุดคำสั่งคืออะไร มีสิ่งใดที่ต้องกล่าวถึงบ้าง และเป็นประโยชน์ใน การพัฒนาซอฟต์แวร์ และ ฮาร์ดแวร์หรือไม่ อย่างไร
2. opcode และ operand คืออะไร
3. หากระบบคอมพิวเตอร์มีการอ้างอิงหน่วยความจำโดยใช้เลขที่อยู่ทั้งสิ้น 32 บิตจะ สามารถอ้างอิงข้อมูลแบบ word address ได้ทั้งสิ้นกี่ word
4. ระบบ big endian และ little endian มีข้อแตกต่างกันอย่างไร และแต่ละแบบมี ข้อดีข้อเสียอย่างไร
5. การที่หน่วยประมวลผลมีระบบการอ้างอิงหน่วยความจำหลากหลายรูปแบบ เป็น ประโยชน์หรือไม่ อย่างไร
6. หากสถาปัตยกรรมชุดคำสั่งที่กำหนดให้ มีเพียงระบบการอ้างอิงหน่วยความจำ แบบ register indirect และ immediate เท่านั้น เราจะสามารถใช้งาน สถาปัตยกรรมชุดคำสั่งดังกล่าวในการพัฒนาโปรแกรมที่ว่าไปได้หรือไม่ อย่างไร จง อธิบายพร้อมให้เหตุผลประกอบ
7. การที่ instruction format ทุกรูปแบบมีความยาวคำสั่งเท่ากันหมด มีข้อดีข้อเสีย อย่างไร
8. ภาษาแอกซ์เพรสบลีและสถาปัตยกรรมชุดคำสั่งมีข้อเหมือนหรือข้อแตกต่างกันหรือไม่ อย่างไร
9. จากมุมมองของคอมไพล์เยอร์ สถาปัตยกรรมชุดคำสั่งที่ดี ควรมีคุณสมบัติอย่างไร
10. จงแสดงการจัดเรียงข้อมูลตามลำดับการจองตัวแพรดิงกล่าวในภาษาซี ลงในหน่วย ความจำที่มีการบังคับ restricted alignment (กำหนดให้ char มีขนาด 1 ไบต์ short มีขนาด 2 ไบต์ int, long และ float มีขนาด 4 ไบต์ long long และ double มีขนาด 8 ไบต์)


```
char a;
short b;
int c;
```

```
char d;
long e;
double f;
```

11. เหตุใดในสถาปัตยกรรมชุดคำสั่งสมัยใหม่ จึงไม่นิยมทำเป็นแบบ accumulator จงให้เหตุผลประกอบการอธิบาย
12. ในการเรียกใช้งานโปรแกรมย่อย จงเปรียบเทียบข้อดีข้อเสียระหว่างการจำค่าโดยผู้เรียก และ การจำค่าโดยผู้ถูกเรียก กรณีเด่นๆจะทำงานได้อย่างมีสมรรถนะสูงกว่า จงอธิบายพร้อมยกตัวอย่างประกอบ
13. จากตัวอย่างโปรแกรมที่กำหนดให้ ให้ลองแปลกด้วย “gcc -C -S test.c” แล้วลองศึกษาดูว่า โปรแกรมดังกล่าว มีการส่งผ่านค่า parameters และ การคืนค่าอย่างไร (หากเป็นไปได้ ให้ลองทำในระบบปฏิบัติการและหน่วยประมวลผลกลางที่ใช้สถาปัตยกรรมชุดคำสั่งแตกต่างกันไป เพื่อจะได้เห็นความหลากหลาย)

```
test.c

int min(int a, int b) {
    int minv;
    minv=a;
    if (min>b) {
        minv=b;
    }
    return minv;
}

void test() {
    int x=min(3,2);
}
```

14. หากสถาปัตยกรรมชุดคำสั่งเลือกใช้ rejister (เข้น \$r13) ในการบันทึกค่า return address โปรแกรมย่อยจะต้องมีการจัดการอย่างไร เพื่อให้สามารถเรียกใช้งานโปรแกรมย่อยอื่นหรือทำ recursive call ได้อีก จงอธิบายพร้อมยกตัวอย่างประกอบ
15. จงอธิบายข้อแตกต่างระหว่าง vector table และ handler table พร้อมยกตัวอย่างประกอบการอธิบาย (โดยเฉพาะกรณีที่ handler ไม่มีการทำงานมีแต่เพียงการทำ IRET)

4 การออกแบบ หน่วยประมวลผลกลาง แบบ single cycle

ตามที่ได้ศึกษาถึงสถาปัตยกรรมชุดคำสั่งมาแล้ว เนื้อหาภายในบทนี้จะเป็นการสร้างหน่วยประมวลผลกลางแบบง่าย โดยการสร้างองค์ประกอบพื้นฐานและนำองค์ประกอบต่าง ๆ มาประกอบเข้าด้วยกันเป็นหน่วยประมวลผลกลาง โดยมุ่งประเด็นที่การออกแบบทางเดินข้อมูล (data path) และการสร้างสัญญาณเพื่อควบคุมองค์ประกอบภายใต้เป็นหลัก

หน่วยประมวลผลกลางประกอบขึ้นจากการจราไฟฟ้าต่าง ๆ ซึ่งหากวิเคราะห์ลักษณะของวงจรที่เป็นส่วนประกอบภายใต้หน่วยประมวลผลกลางตามลักษณะของหน้าที่ จะสามารถจำแนกวงจรได้เป็น 2 กลุ่ม คือ

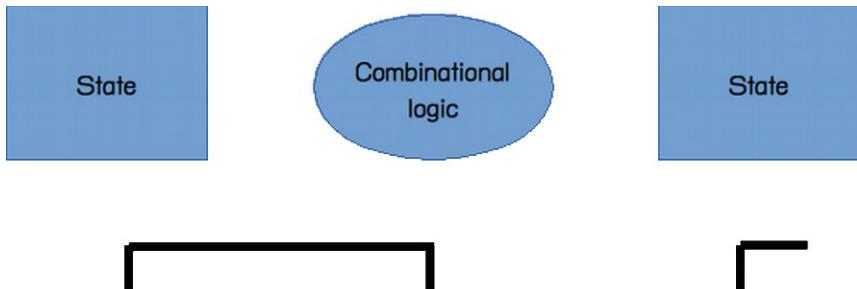
- วงจรสำหรับการประมวลผลข้อมูล หรือ วงจรเชิงผสม (combinational circuit) เป็นวงจรที่ทำหน้าที่ถอดรหัส ตีความ หรือประมวลผล
- วงจรสำหรับการประมวลผลแบบลำดับ หรือ วงจรเชิงลำดับ (sequential circuit) เป็นวงจรที่ทำหน้าที่กำหนดการทำงานของส่วนต่าง ๆ ของระบบให้ทำงานตามขั้นตอนที่เหมาะสม

สายสัญญาณที่ใช้ในการรับส่งข้อมูลระหว่างส่วนต่าง ๆ ของวงจรเรียกว่าทางเดินข้อมูล ส่วนวงจรที่ใช้ควบคุมการทำงานของทางเดินข้อมูล และกำหนดลำดับสัญญาณให้เป็นไปตามขั้นตอน (สัญญาณควบคุม) เรียกว่า หน่วยควบคุม (control unit)

เนื้อหาในบทนี้จะเริ่มต้นจากการสร้างส่วนประกอบต่าง ๆ ภายในหน่วยประมวลผลกลางก่อน จากนั้นจะนำส่วนประกอบต่าง ๆ มาต่อ กันด้วยทางเดินข้อมูล เพื่อให้ข้อมูลสามารถเดินทางได้ และสุดท้ายจึงจะเป็นการออกแบบสัญญาณควบคุม ผลที่ได้หลังจากบทนี้คือ หน่วยประมวลผลกลางแบบ single cycle (จำนวนรอบการประมวลผลเป็น 1) โดยทุกคำสั่งจะต้องรอรอบการประมวลผลที่นานพอจะให้ทำงานเสร็จ

เพื่อเป็นการบททวนความรู้เรื่อง sequential logic ลองมาทำความเข้าใจ ความสัมพันธ์ระหว่างระบบสัญญาณนาฬิกาและวงจรเชิงผสมและเชิงลำดับอีกครั้งก่อนที่จะเริ่มเข้าสู่เนื้อหาดังนี้ สัญญาณนาฬิกาเป็นสัญญาณสำคัญในการให้จังหวะการทำงาน (synchronize)

ของวงจร โดยที่ไปสัญญาณขาเข้าของนาฬิกา (positive-edge clock) จะเป็นสัญญาณให้จังหวะการเปลี่ยนสถานะของวงจรเชิงลำดับลงรูปที่ 4.1 ในรูปสัญลักษณ์สีเหลืองจะแทนสถานะซึ่งจะเปลี่ยนทุกครั้งที่เกิดสัญญาณนาฬิกาใหม่ และวงรีแทนการทำงานของวงจรเชิงผสมที่ทำงานระหว่างค่าของเวลา



รูปที่ 4.1: ความสัมพันธ์ระหว่างสัญญาณนาฬิกา วงจรเชิงผสม และ วงจรเชิงลำดับ

ในการนี้ของหน่วยประมวลผลกลางแบบ 1 รอบในบทนี้ หนึ่งคาบ (period) ของสัญญาณนาฬิกา คือเวลาในการประมวลผลหนึ่งคำสั่ง กล่าวคือ ทุกครั้งที่มีขาเข้าของสัญญาณนาฬิกา จะเป็นการเริ่มคำสั่งใหม่เสมอ นั่นหมายความว่าเวลาหนึ่งคาบจะต้องมากพอที่จะทำให้วงจรเชิงผสมที่อยู่ระหว่างกลางนั้นทำงานเสร็จได้ หากเวลาอ้อยไปอาจทำให้การประมวลผลที่ได้เกิดความผิดพลาด

4.1 ขั้นตอนในการออกแบบหน่วยประมวลผลกลาง

เพื่อความสะดวกในการศึกษาและออกแบบหน่วยประมวลผลกลาง สามารถแบ่งขั้นตอนการออกแบบเป็น 5 ขั้นตอนดังนี้

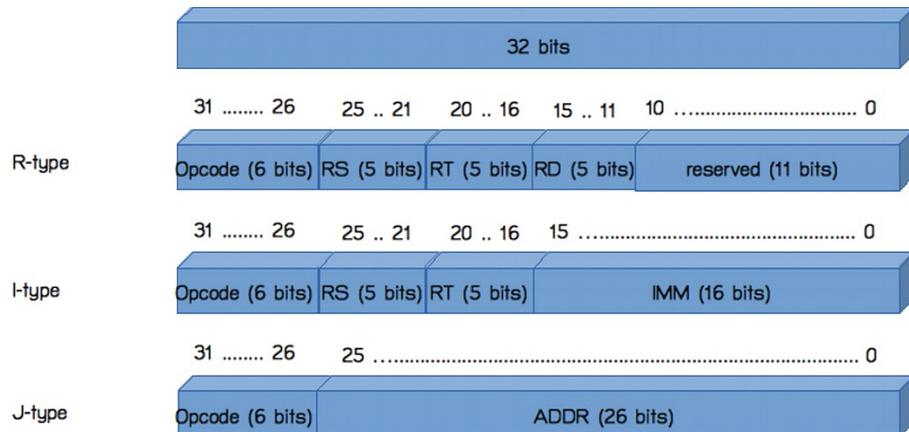
1. วิเคราะห์สถาปัตยกรรมชุดคำสั่ง เพื่อกำหนดส่วนประกอบที่เกี่ยวข้อง
2. ออกแบบส่วนประกอบที่เกี่ยวข้องเพื่อประกอบเป็นทางเดินข้อมูล
3. ประกอบทางเดินข้อมูล
4. วิเคราะห์สัญญาณควบคุมที่เกี่ยวข้อง
5. ออกแบบวงจรควบคุมเพื่อกำหนดสัญญาณควบคุม

จากขั้นตอนดังกล่าว เริ่มด้วยการศึกษาและวิเคราะห์สถาปัตยกรรมชุดคำสั่งที่จะใช้สำหรับหน่วยประมวลผลกลางก่อน เพื่อความสะดวกในการศึกษา ในบทนี้จะยกตัวอย่างการออกแบบทั้งหมดโดยใช้บางส่วนของสถาปัตยกรรม LADA มาเพื่อประกอบการอธิบายดังนี้

4.1.1 สถาปัตยกรรมชุดคำสั่ง nanoLADA

กำหนดให้สถาปัตยกรรม nanoLADA ประกอบด้วยคำสั่งจำนวน 7 คำสั่ง และมีรูปแบบคำสั่ง 3 รูปแบบ คือ R-type, I-type, และ J-type โดยมีรายละเอียดดังต่อไปนี้

รูปแบบคำสั่ง 3 รูปแบบ



ในการตีคำสั่งที่เกี่ยวข้องการหน่วยความจำ กำหนดให้

$$\text{Instruction} = \text{MEM}[\text{PC}]$$

คำสั่ง

ORI rt, rs, imm ; (I-type)

ความหมาย $R[rt] \leftarrow R[rs] \mid \text{zero_ext}(imm); \quad PC \leftarrow PC + 4$

ORUI rt, rs, imm ; (I-type)

ความหมาย $R[rt] \leftarrow R[rs] \mid \text{zero_pad}(imm); \quad PC \leftarrow PC + 4$

<i>ADD rd, rs, rt</i>	<i>; (R-type)</i>
ความหมาย $R[rd] \leftarrow R[rs] + R[rt];$	$PC \leftarrow PC + 4$
<i>LW rt, imm(rs)</i>	<i>; (I-type)</i>
ความหมาย $R[rt] \leftarrow MEM[R[rs]] + sign_ext(imm);$	$PC \leftarrow PC + 4$
<i>SW rt, imm(rs)</i>	<i>; (I-type)</i>
ความหมาย $MEM[R[rs]] + sign_ext(imm) \leftarrow R[rt];$	$PC \leftarrow PC + 4$
<i>BEQ rs, rt, imm*4</i>	<i>; (I-type)</i>
ความหมาย If $(R[rs] == R[rt])$ then $PC \leftarrow PC + 4 + (sign_ext(imm) * 4)$ else $PC \leftarrow PC + 4$	
<i>JMP addr*4</i>	<i>; (J-type)</i>
ความหมาย $PC \leftarrow (addr * 4)$	

ข้อสังเกต คำสั่งควบคุมและการทำงานที่เกี่ยวกับ PC จะเป็นการเพิ่มหรือลดทีลีช 4 ไบต์ เนื่องจากทุกคำสั่งมีความยาว 32 บิต (4 ไบต์) เมื่อนำมาเขียนเป็นเลขฐานสองจะได้เลขที่อยู่เป็น 0000 (0) 0100 (4) 1000 (8) 1100 (12) ต่อเนื่องไป ดังนั้นการทำ $PC \leftarrow PC + 4$ จะทำด้วยการตัดส่องบิตท้ายออกแล้วบวกด้วย 1 แทนก็ได้ (เขียนแทนด้วย $\{PC[31:2] + 1, 2'b00\}$ ในภาษา Verilog HDL) กรณีของคำสั่ง BEQ และ JMP จึงนำค่ามาคูณด้วย 4 ก่อนนำมาใช้เป็นเลขที่อยู่ชั้นกัน เพื่อให้สามารถอ้างอิงเลขที่อยู่ได้ใกล้ชัน 4 เท่าด้วยจำนวนบิตเท่าเดิม (เพราะส่องบิตหลังเป็น 00 เสมอ)

ในส่วนต่อไป จะเป็นการวิเคราะห์สถาปัตยกรรมชุดคำสั่งและการออกแบบค์ประกอบที่เกี่ยวข้องแต่ละส่วน

4.2 องค์ประกอบภายในสำหรับสถาปัตยกรรม nanoLADA

เนื้อหาในส่วนนี้เตรียมได้กับขั้นตอนที่สองของการออกแบบหน่วยประมวลผลกลาง จาก

สถาปัตยกรรมชุดคำสั่งที่กำหนดโดยเคราะห์ได้ว่าสถาปัตยกรรมดังกล่าวจะต้องมีองค์ประกอบที่เกี่ยวข้องได้แก่

1. ชุดเรจิสเตอร์ขนาด 32 บิตจำนวน 32 ชุด²¹ สามารถอ่านได้ 2 ชุดพร้อมกัน (*rs* และ *rt*) และสามารถเขียนข้อมูลได้ 1 ชุด (*rt* หรือ *rd*)
2. เรจิสเตอร์พิเศษสำหรับเป็น PC
3. ALU มีความสามารถในการ บวก และ ออร์ (or)
4. ADDER สำหรับเพิ่มค่า PC
5. EXTENDER มีความสามารถในการเปลี่ยนข้อมูล 16 บิต ให้เป็น 32 บิตโดยการยืดเครื่องหมาย (sign extender), เติมศูนย์ด้านหน้า (zero extender) หรือ เติมศูนย์ด้านหลัง (zero padding)

สำหรับ PC จะเป็นองค์ประกอบพื้นฐานที่สามารถออกแบบได้ด้วย D-flipflop ส่วน ADDER และ ALU สามารถสร้างได้ด้วยวงจร full adder, logic gate และ multiplexer ทั่วไป²²

4.2.1 ชุดเรจิสเตอร์ (Register file)

เมื่อกล่าวถึงเรจิสเตอร์ นักคอมพิวเตอร์มักจะนึกถึง D-flipflop ซึ่งมีความสามารถในการเก็บข้อมูล 1 บิต ดังนั้นหากต้องการให้มีเรจิสเตอร์ขนาด 32 บิต สามารถทำได้โดยการนำ D-flipflop จำนวน 32 ตัวมาต่อหนานกัน โดยใช้สัญญาณ enable, read/write ชุดเดียวกัน

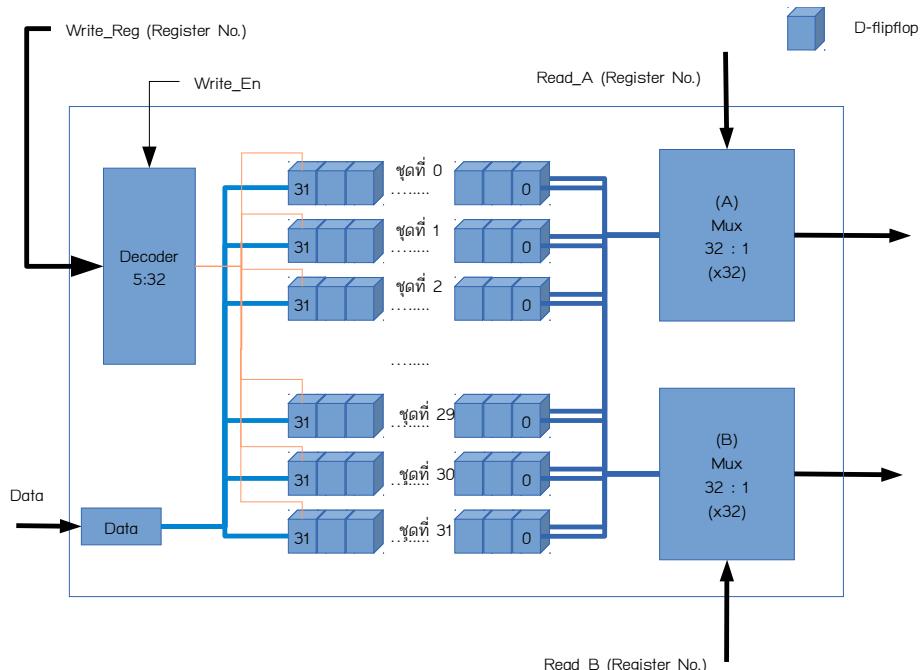
สิ่งที่ซับซ้อนขึ้นในที่นี้คือ สถาปัตยกรรมชุดคำสั่งนี้ ต้องการชุดของเรจิสเตอร์ที่ขนาด 32 บิต จำนวน 32 ชุด ที่สามารถอ่านได้ 2 ชุด และ เขียนได้ 1 ชุด พร้อมกัน การออกแบบในที่นี้ จึงต้องใช้ ของ D-flipflop จำนวน 32 ตัวมาต่อหนานกันเพื่อให้ได้ 1 ชุด และทำลักษณะเดียวกันนี้อีก 31 ชุด (รวมกันเป็น 32 ชุด) น่าวางต่อ กันโดยใช้วงจรเลือก (multiplexor) แบบ 32 เลือก 1 จำนวน 32×2 ชุด ในการเลือกว่าต้องการอ่านข้อมูลชุดใด ดังแสดงในรูปที่ 4.2 ด้านขวา (ในความเป็นจริง จำนวนสายสัญญาณจะซับซ้อนกว่านี้มาก ในที่นี้เพียง

21 เนื่องจากชุดคำสั่งที่ข้อมูล 5 บิตในการบอกว่าต้องการใช้ข้อมูลจากเรจิสเตอร์ *rs*, *rt* ด้วยเลขที่อยู่ในนา 5 บิต จึงทำให้สามารถเลือกเรจิสเตอร์ได้ทั้งหมด 32 ชุด (0..31)

22 หากอ่านถึงตรงนี้แล้ว งง ไม่เข้าใจว่า D-flipflop หรือ full adder คืออะไร คงจะต้องแนะนำให้ไปลองเข้าเรียนในวิชา Digital Computer Logic ซึ่งเป็นวิชาพื้นฐานก่อนเรียนวิชานี้ก่อน

ลักษณะของข้อมูลที่เกี่ยวข้องเพื่อแสดงให้เห็นลักษณะโครงสร้างประกอบการอธิบายเท่านั้น)

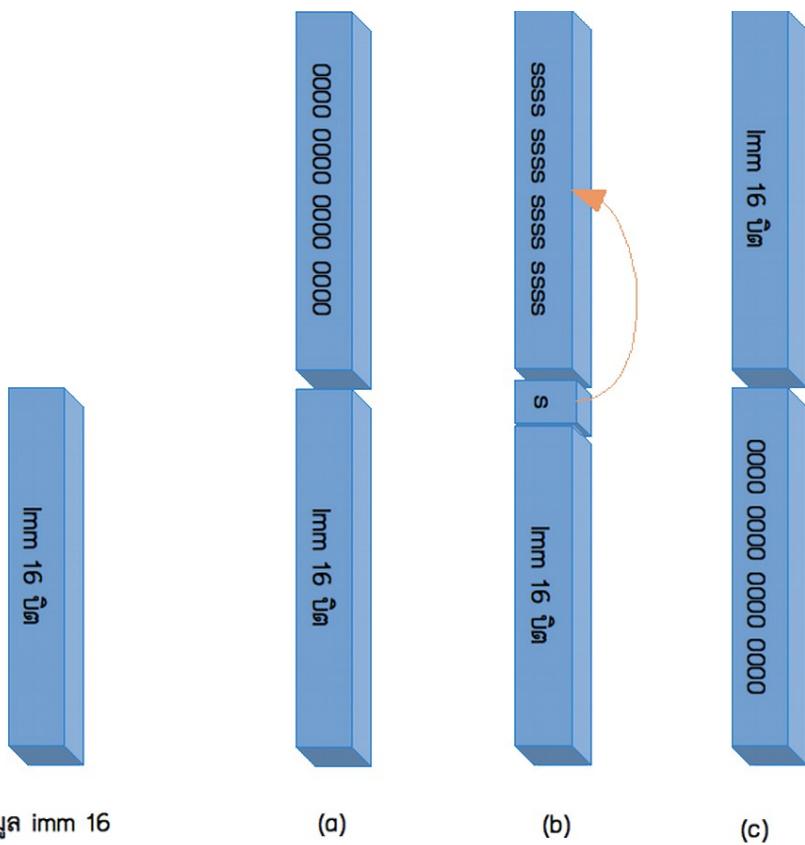
กรณีการเขียนค่าใน地址สัญญาณนาฬิกาหรือสัญญาณ write เป็นตัวกำหนดการเขียนค่าให้กับเรจิสเตอร์ร่วมกับ decoder เพื่อเลือกค่าเรจิสเตอร์ที่ต้องการเขียน ดูรูปที่ 4.2 จะเห็นว่า การเขียนค่าเรจิสเตอร์ที่ 3 ด้วยข้อมูล 2521 ทำได้โดยการป้อนค่า 3 ที่ Write_Reg และ ค่า 2521 ที่ data จากนั้นจึงให้จังหวะสัญญาณ Write_En (ซึ่งสัญญาณเหล่านี้จะสร้างจากวงจรควบคุมที่จะกล่าวถึงต่อไป)



รูปที่ 4.2: ชุดเรจิสเตอร์ขนาด 32×32 บิต สามารถอ่านได้ 2 ชุด และเขียนได้ 1 ชุดพร้อมกัน รูปที่ 4.2 ด้านซ้ายเป็นการเขียนข้อมูล ประกอบไปด้วยสัญญาณ Write_En เพื่อให้สัญญาณการเขียน (มักจะเป็น positive edge ของ สัญญาณนาฬิกา) สัญญาณ Write_Reg (5 บิต) เพื่อกำหนดชุดเรจิสเตอร์ที่ต้องการเขียนลง และ Data (32 บิต) เป็นข้อมูลที่ต้องการบันทึก ด้านขวาของรูป เป็นข้อมูลที่ต้องการอ่าน 2 ชุด ซึ่งกำหนดโดยค่า Read_A (5 บิต) และ ค่า Read_B (5 บิต) ภาคผนวก ฯ.1 แสดง Verilog HDL ของชุดเรจิสเตอร์

4.2.2 Extender

extender มีคุณสมบัติในการทำการเติมค่าให้ครบ 32 บิตทำได้ 3 แบบตามรูปที่ 4.3 กล่าวคือ (a) zero extender คือการเติม 0 ข้างหน้าจำนวน 16 บิต เพื่อแปลง immediate 16 บิตให้เป็นค่า 32 บิต (b) sign extender คือการขยายบิตเครื่องหมายจำนวน 16 บิต ให้ค่าเป็น 32 บิต และ (c) zero padding คือการเติม 0 ข้างท้ายจำนวน 16 บิต เพื่อแปลง immediate 16 ให้เป็นค่า 32 บิต



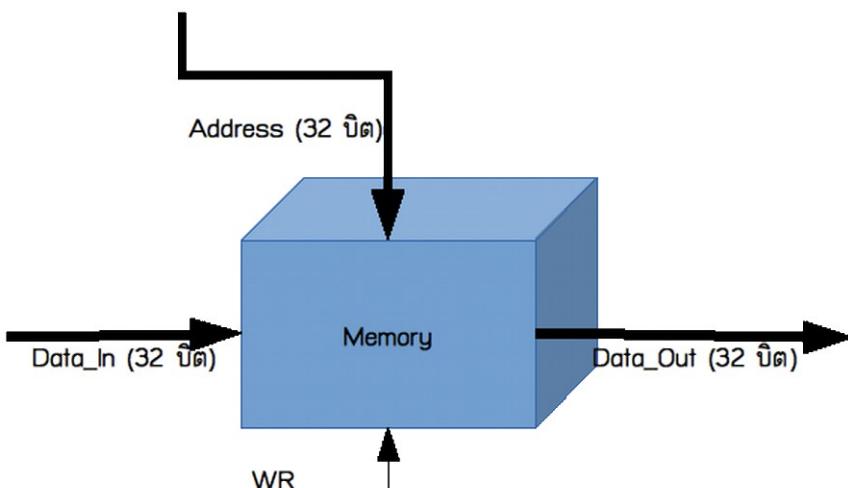
รูปที่ 4.3: การทำงานของ Extender (a) zero extender (b) sign extender (c) zero padding

การสร้าง extender แบบง่าย ๆ อาศัย multiplexor เลือกว่าต้องการผลลัพธ์ชุดใดในการแสดงผล หรืออาจใช้ multiplexor ชุดหนึ่งเลือกว่าจะทำ extender หรือ padding แล้วใช้ multiplexor อีกชุดหนึ่งเลือกว่าจะ extend ด้วย zero หรือ sign ก็ได้ ภาคผนวก ข.2 แสดง Verilog HDL ของ extender (รายละเอียดการออกแบบ ขอให้เป็นแบบฝึกหัดของผู้อ่านในแบบฝึกหัดท้ายบท)

4.2.3 หน่วยความจำ

เพื่อความสะดวก ในบทนี้สมมุติให้นำหน่วยความจำมีการทำงานที่รวดเร็วไม่จำต้องต้องรอข้อมูลเป็นเวลานาน ทั้งนี้ในความเป็นจริงหน่วยความจำที่มีขนาดใหญ่จะใช้เวลาในการเข้าถึงข้อมูลมาก ในขณะที่หน่วยความจำขนาดเล็กมักใช้เวลาในการเข้าถึงข้อมูลน้อย ซึ่งโครงสร้างและการจัดการกับหน่วยความจำจะกล่าวถึงต่อไปในบทที่ 8

ในตอนนี้ ให้มองหน่วยความจำเป็นกล่อง (เทียบได้กับ memory chip) ที่เมื่อส่งเลขที่อยู่เข้าไป จะให้ค่าที่ต้องการออกมาทาง Data_Out และ กรณีที่มีสัญญาณ WR จะบันทึกค่าจาก Data_In เข้าไปยังเลขที่อยู่ที่กำหนด รายละเอียดแสดงได้ดังรูปที่ 4.4



รูปที่ 4.4: โครงสร้างหน่วยความจำ

เพื่อความสะดวกในการออกแบบ ในขั้นนี้ จะแยกหน่วยความจำออกเป็นสองชั้นออกจากกัน คือ program memory (บางคำเรียกว่า instruction memory) และ data memory ทั้งนี้เพื่อให้สามารถอ่านคำสั่งพร้อมกับอ่านและเขียนข้อมูลได้พร้อมกัน (ในทางปฏิบัติ จะเป็น

เพียงการแยก cache หรือที่พักข้อมูลชั่วคราวเพื่อให้สามารถเข้าถึงข้อมูลจากหน่วยประมวลผลกลางได้พร้อมกันเท่านั้น ซึ่งจะกล่าวถึงต่อไปเมื่อถึงเนื้อหาในส่วนของหน่วยความจำ

4.3 ทางเดินข้อมูลสำหรับสถาปัตยกรรมชุดคำสั่ง nanoLADA

เนื้อหาในส่วนนี้ เทียบได้กับขั้นตอนที่สองของการออกแบบหน่วยประมวลผลกลาง ทางเดินข้อมูล หรือ bus ภายในหน่วยประมวลผลกลางเป็นสายสัญญาณที่ใช้ส่งผ่านข้อมูลระหว่างองค์ประกอบต่าง ๆ ภายในหน่วยประมวลผลกลาง ในการออกแบบสามารถทำได้โดยนำองค์ประกอบต่าง ๆ มาต่อพ่วงกันในรูปแบบที่เหมาะสม และใช้ multiplexor เป็นตัวเลือกทางเดินของข้อมูล กรณีที่ข้อมูลดังกล่าวมีทางเลือกที่เป็นไปได้มากกว่าหนึ่งทาง เช่น ค่า PC ซึ่งเก็บเลขที่อยู่ของคำสั่งต่อไปที่จะถูกดึงขึ้นมาทำงาน อาจจะอยู่ติดกับคำสั่งปัจจุบันคือ PC+4 หรือ อาจจะเป็นค่าอื่น ๆ ที่เกิดจากการ jump หรือ branch จึงใช้ multiplexor สำหรับการเลือก

อย่างไรก็ตาม การเลือกว่าครमี data path เชื่อมต่อระหว่างจุดใดจุดใด จะพิจารณาจากสถาปัตยกรรมชุดคำสั่งเป็นหลัก โดยพิจารณาจากวิธีการอ้างอิงเลขที่อยู่และชุดคำสั่งว่า แต่ละคำสั่งนั้นจะต้องใช้องค์ประกอบใดบ้าง และข้อมูลที่ต้องการใช้ในการประมวลคำสั่งนั้น ๆ ได้มาจากแหล่งใด เช่น คำสั่งประมวลด้วย operand ซึ่งมาจากเรจิสเตอร์ หรือมาจากคำสั่งโดยตรง จึงต้องมีทางเดินข้อมูลเพื่อนำข้อมูลจากเรจิสเตอร์ไปป้อนเป็นผลลัพธ์ให้กับ ALU และทางเดินข้อมูลที่จะนำข้อมูลในกรณีที่ operand ที่อยู่ในคำสั่งไปป้อนให้กับ ALU ด้วยเช่นกัน

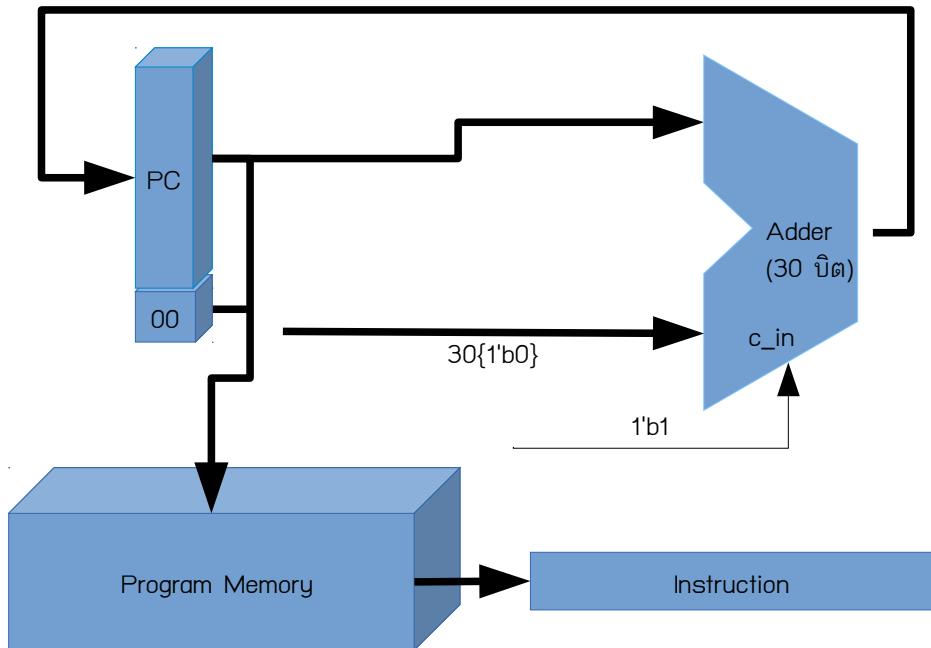
ยกข้อต้นการประกอบทางเดินข้อมูลโดยจะนำคำสั่งเข้ามาใส่ที่ลําคำสั่งเพื่อเป็นตัวอย่าง โดยเริ่มจากคำสั่ง ADD, ORI, ORUI, LW, SW, BEQ และ JMP ตามลำดับ ทั้งนี้ขอให้เทียบกับสถาปัตยกรรมชุดคำสั่งในส่วนที่ 4.1.1 ประกอบ

ก่อนอื่น ทุกคำสั่งมีโครงสร้างการทำงานที่เหมือนกันคือ การ fetch คำสั่ง โดยมี RTL ของ การ fetch คำสั่งคือ $instruction \leftarrow MEM[PC]$ และมีขั้นตอนการทำงานตอนท้ายคำสั่ง คล้ายคลึงกัน คือ $PC \leftarrow PC + 4$ จาก RTL นี้ เราสามารถออกแบบทางเดินข้อมูลได้ดังรูปที่ 4.5

เนื่องจากทุกคำสั่งมีขนาด 4 ไบต์ (32 บิต) ดังนั้น PC จะลงท้ายด้วย 2'b00 เสมอ เพื่อความสะดวก เราจึงใช้ adder ขนาด 30 บิต การบวก 1 กรณี 30 บิตก็คือการบวก 4 กรณี 32 บิต (หากคิดเป็นเลขฐาน 10 จะได้ว่า $PC \leftarrow ((PC / 4) + 1) * 4$ ซึ่ง มีค่าเหมือนกับ $PC \leftarrow PC + 4$ นั้นเอง) แต่เนื่องจากในเลขฐานสอง การคูณด้วยสิบคือการ shift left ไป

2 บิต หรือการใส่เลื่อนย้อนต่อท้ายสองตัว การหารด้วยสี่หรือคูณด้วยสี่ในลักษณะนี้ จึงทำได้ง่ายมาก ส่วนเมื่อเป็นการทำคำสั่ง branch ค่า $\{30[1'b0]\}$ ²³ นั้น จะถูกแทนที่ด้วยค่าอื่น

จากนี้จะเป็นการนำคำสั่งแต่ละคำสั่งมาวิเคราะห์ และประกอบทางเดินข้อมูลให้สมบูรณ์ กายในภาพเส้นทึบแสดงทางเดินข้อมูล ส่วนเส้นประแสดงสัญญาณควบคุม



รูปที่ 4.5: ทางเดินข้อมูลสำหรับ $PC \leftarrow PC + 4$

4.3.1 ทางเดินข้อมูลสำหรับคำสั่ง ORI และ ORUI

ORI rt, rs, imm ; (*I-type*)

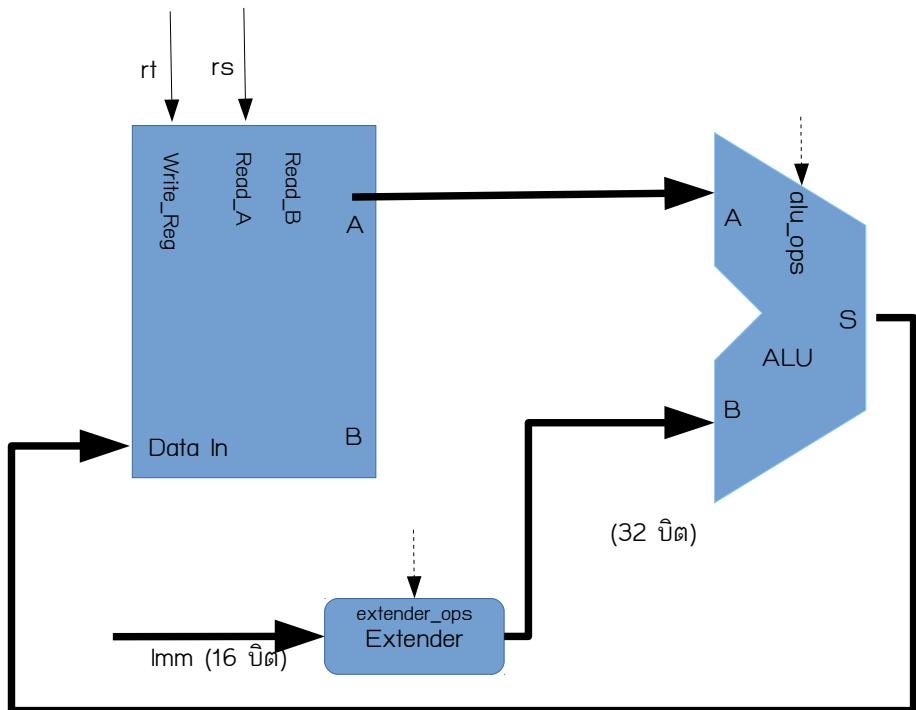
ความหมาย $R[rt] \leftarrow R[rs] \mid \text{zero_ext}(imm);$

ORUI rt, rs, imm ; (*I-type*)

ความหมาย $R[rt] \leftarrow R[rs] \mid \text{zero_pad}(imm);$

23 (บิต $\{30[1'b0]\}$ หมายถึงเลข 0 จำนวน 30 บิต เปรียนแบบภาษา Verilog HDL)

คำสั่ง ORI เป็นแบบ I-type มีการทำ extender และใช้ ALU ในการประมวลผลทางตรรกะแบบ OR จาก RTL สามารถสร้างทางเดินข้อมูลโดยทำทางเดินจาก $R[rs]$ และ $\text{zero_ext}(imm)$ ไปยัง ALU และนำผลลัพธ์ที่ได้ไปเก็บยัง $R[rt]$ ดังรูปที่ 4.6



รูปที่ 4.6: ทางเดินข้อมูลเมื่อเพิ่มคำสั่ง ORI และ ORUI

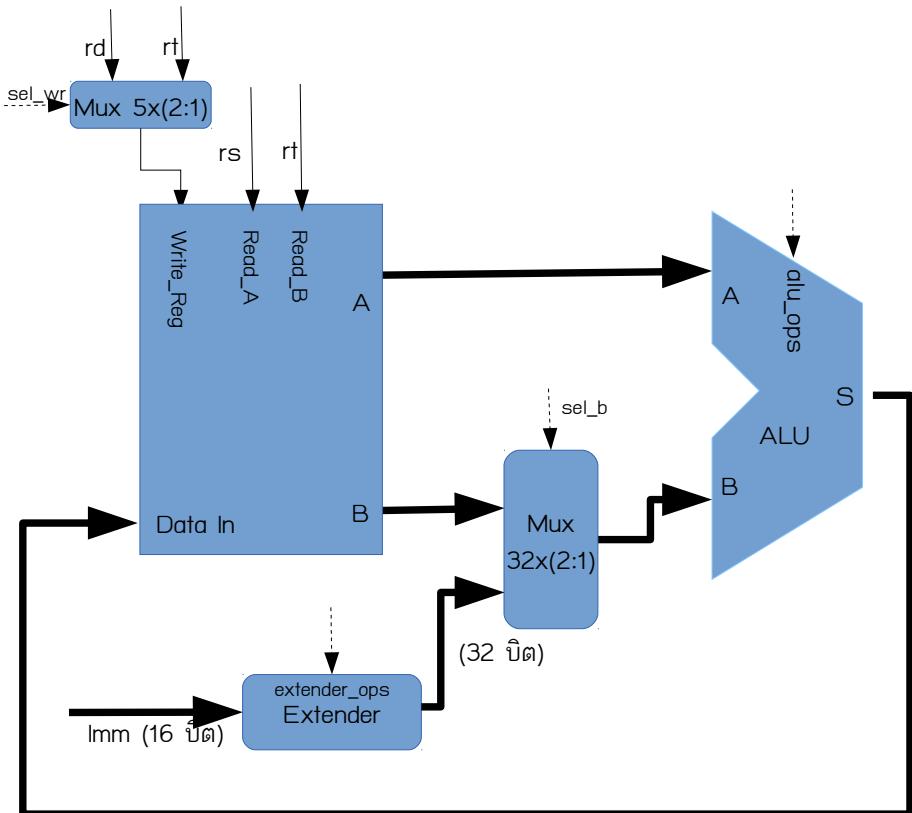
ในทำนองเดียวกันกับคำสั่ง ORI คำสั่ง ORUI มีการทำงานที่คล้ายคลึงกับคำสั่ง ORI มาก ข้อแตกต่างเพียงแค่การกำหนดให้ extender ทำการ zero padding แทนที่การทำ zero extending ดังนั้นทางเดินข้อมูลจึงไม่มีการเปลี่ยนแปลง (มีเพียงการเปลี่ยนแปลงของสัญญาณควบคุม extender_ops เท่านั้น)

4.3.2 ทางเดินข้อมูลสำหรับคำสั่ง ADD

$ADD rd, rs, rt$	$; (R-type)$
------------------	--------------

ความหมาย $R[rd] \leftarrow R[rs] + R[rt];$

คำสั่ง ADD เป็นแบบ R-type มีการนำค่า R[rs] และ R[rt] มาส่งให้ ALU ทำการบวก และนำผลลัพธ์ที่ได้ไปเก็บที่ R[rd] เพื่อให้สามารถทำคำสั่ง ADD ได้ จะต้องมีการปรับทางเดินข้อมูลเพิ่มเติมให้ค่า R[rt] มาป้อนเข้า ALU และให้สามารถเขียนค่าไปยัง R[rd] ได้ ดังรูปที่ 4.7



รูปที่ 4.7: ทางเดินข้อมูลเมื่อเพิ่มคำสั่ง ADD

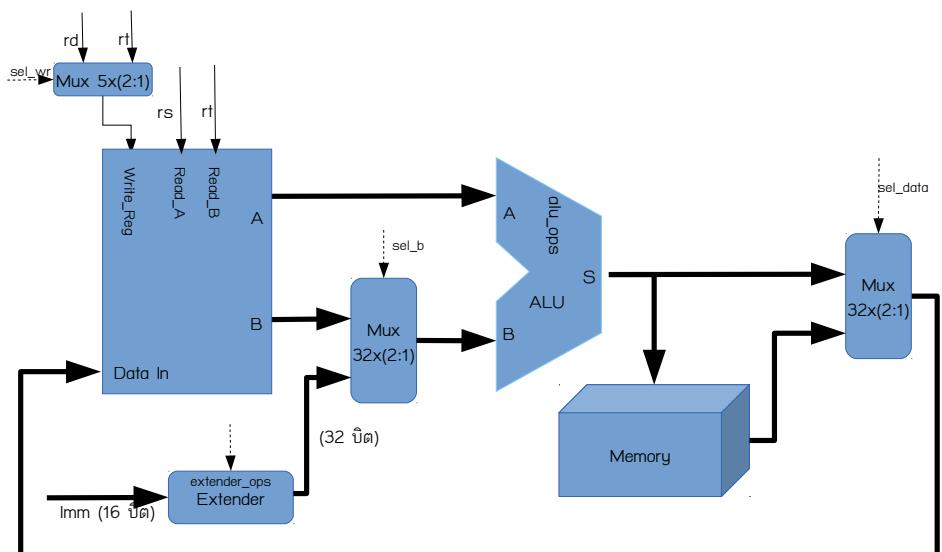
จะสังเกตเห็นว่ามี multiplexor 2 เลือก 1 เพิ่มขึ้นมา 2 ชุด ชุดแรกใช้เพื่อเลือกข้อมูลสำหรับป้อนเข้าสู่ B ของ ALU (ควบคุมด้วยสัญญาณ sel_b) และชุดที่สองใช้สำหรับเลือกเรจิสเตอร์ที่ต้องการเขียนค่า (ควบคุมด้วยสัญญาณ sel_wr)

4.3.3 ทางเดินข้อมูลสำหรับคำสั่ง LW

LW rt, rs, imm ; (I-type)

ความหมาย $R[rt] \leftarrow \text{MEM}[R[rs] + \text{sign_ext}(imm)];$

คำสั่ง LW เป็นการนำค่า $R[rs]$ มาบวกกับ immediate และนำค่าที่ได้ไปใช้เป็นเลขที่อยู่เพื่ออ่านค่าจากหน่วยความจำมาเก็บไว้ใน $R[rt]$ จะเห็นว่าทางเดินข้อมูลที่มิอยู่เดิม สามารถทำการเขียนค่าไปยัง $R[rt]$ ได้แล้ว (โดยการเลือก sel_wr ให้เป็นค่าที่เหมาะสม) และนอกจากนี้ทางเดินข้อมูลดังกล่าวยังสามารถประมวลผลค่า $R[rs] + \text{sign_ext}(imm)$ ได้แล้ว อีกเช่นกัน การนำข้อมูลจากหน่วยความจำมาบันทึกที่ $R[rt]$ จึงทำได้ง่ายเพียง (1) เพิ่ม multiplexer มาใช้เลือกระหว่าง S และ ค่าที่อ่านได้จากหน่วยความจำมาป้อนเป็น Data_in และ (2) นำค่า S มาป้อนเป็นเลขที่อยู่ให้กับหน่วยความจำ ดังแสดงได้ในรูปที่ 4.8



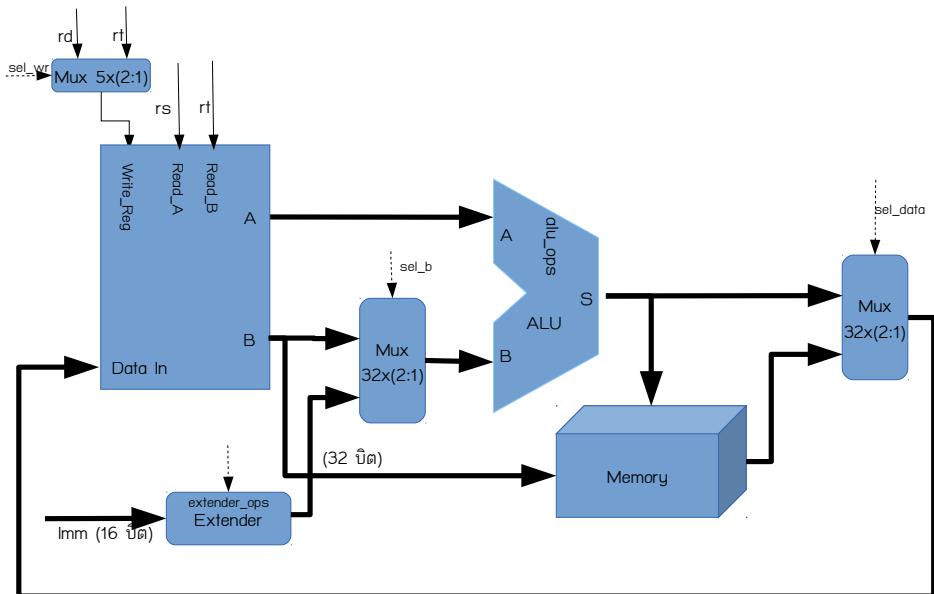
รูปที่ 4.8: ทางเดินข้อมูลเมื่อเพิ่มคำสั่ง LW

4.3.4 ทางเดินข้อมูลสำหรับคำสั่ง SW

SW rt, rs, imm ; (I-type)

ความหมาย $\text{MEM}[R[rs] + \text{sign_ext}(imm)] \leftarrow R[rt];$

คำสั่ง SW มีการทำงานตรงข้ามกับคำสั่ง LW คือเป็นการเขียนข้อมูลแทนที่จะเป็นการอ่านข้อมูล ส่วนการคำนวนเลขที่อยู่ยังคงมีลักษณะการทำงานเหมือนกัน ข้างต้นจากทางเดินข้อมูลที่มีอยู่เดิม สามารถปรับปรุงให้รองรับการทำงานของคำสั่ง SW ได้โดยการเพิ่มทางเดินจาก B ซึ่งเป็นผลลัพธ์ของเรจิสเตอร์ไปยัง Data_in ของหน่วยความจำ ดังแสดงได้ในรูปที่ 4.9



รูปที่ 4.9: ทางเดินข้อมูลเมื่อเพิ่มคำสั่ง SW

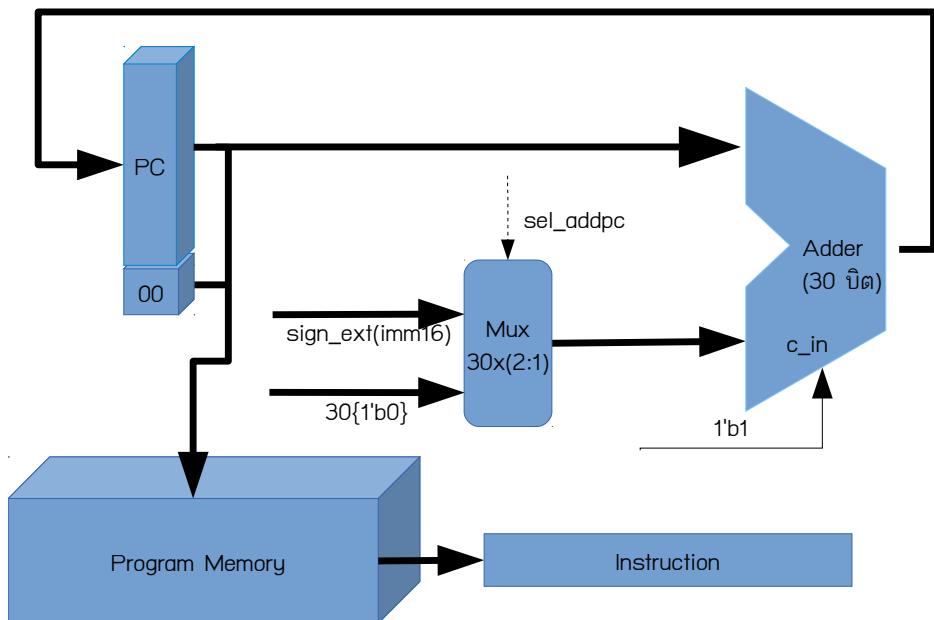
4.3.5 ทางเดินข้อมูลสำหรับคำสั่ง BEQ

*BEQ rs, rt, imm*4 ; (I-type)*

```
ความหมาย If (R[rs] == R[rt]) then
    PC ← PC + 4 + (sign_ext(imm) * 4)
else PC ← PC + 4
```

คำสั่ง BEQ เป็นการเปรียบเทียบค่าระหว่าง R[rs] และ R[rt] โดยหาก R[rs] มีค่าเท่ากัน กับ R[rt] จะบอกค่า PC ด้วย sign_ext(imm) * 4 ในกรณีนี้ ALU ให้ค่า 1 กับ flag zero ได้เมื่อค่า A และ B ซึ่งเป็น Input ของ ALU มีค่าเท่ากัน เรายังนำมามาใช้ในการเลือกได้

ว่าจะต้องบวก PC + 4 ด้วยค่า sign_ext(imm) * 4 หรือไม่จากทางเดินข้อมูลที่มีอยู่สามารถเพิ่มเติมปรับปรุงให้รองรับการทำงานของคำสั่ง BEQ ได้ดังรูปที่ 4.10 โดยในที่นี้จะแสดงเพียงทางเดินข้อมูลส่วนที่เกี่ยวข้องกับ PC เท่านั้น เพื่อประกอบความเข้าใจ ควรพิจารณารูปที่ 4.5 เทียบกับรูปที่ 4.10 จะเห็นว่าส่วนที่เพิ่มเติมขึ้นมาคือ multiplexor (ควบคุมด้วย sel_addpc) สำหรับเลือกค่าที่จะบวกเข้ากับ PC



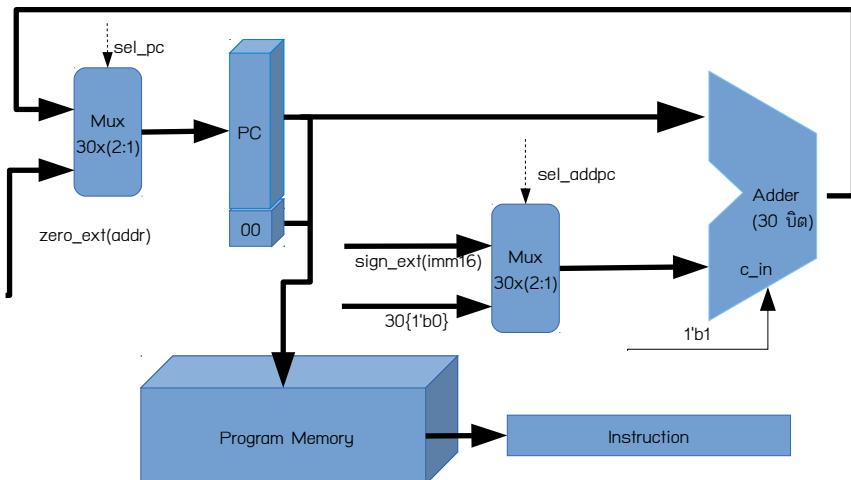
รูปที่ 4.10: ทางเดินข้อมูลส่วน PC เมื่อเพิ่มคำสั่ง BEQ

4.3.6 ทางเดินข้อมูลของคำสั่ง JMP

*JMP addr*4 ; (J-type)*

ความหมาย $PC \leftarrow (addr * 4)$

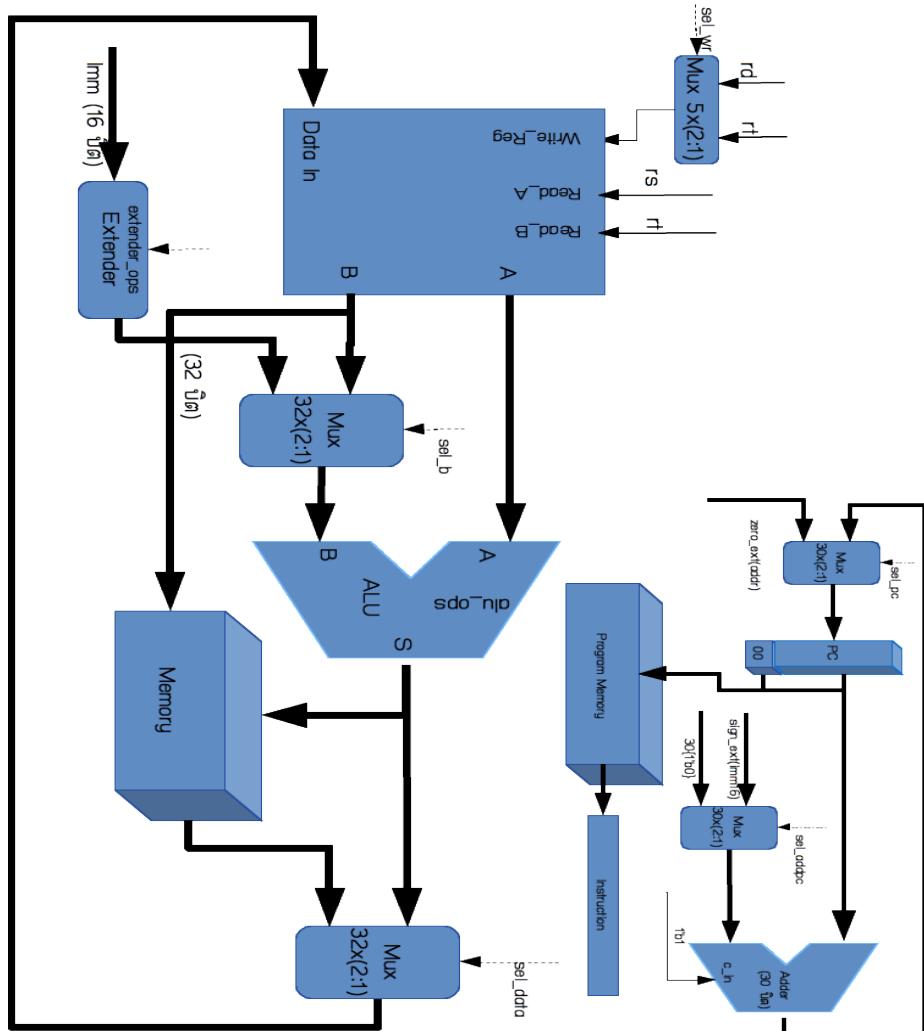
คำสั่ง JMP ทำหน้าที่ในการเปลี่ยนค่า PC เป็น ค่า addr ที่กำหนด เพื่อให้ทางเดินข้อมูลรองรับการทำงานของ JMP สามารถทำได้โดยปรับให้มี multiplexor สำหรับเลือกว่าจะให้ $PC \leftarrow PC + 4$ หรือ $PC \leftarrow addr * 4$ แทน ผลที่ได้ดังแสดงในรูปที่ 4.11



รูปที่ 4.11: ทางเดินข้อมูลส่วน PC เมื่อเพิ่มคำสั่ง JMP

4.3.7 สรุปทางเดินข้อมูลสำหรับสถาปัตยกรรมชุดคำสั่ง nanoLADA

กล่าวโดยสรุป ในการสร้างทางเดินข้อมูลทำได้โดยการนำส่วนประกอบต่าง ๆ มาวางและลากสายสัญญาณเพื่อเชื่อมระหว่างส่วนประกอบต่าง ๆ ให้สามารถประมวลผลข้อมูลได้ตามคุณสมบัติที่กำหนดใน RTL ของแต่ละคำสั่ง หลังจากได้ทางเดินข้อมูลที่สมบูรณ์ (ประมวลผลได้ทุกคำสั่งในสถาปัตยกรรมชุดคำสั่งที่กำหนด) จะเหลือเพียงสัญญาณควบคุมต่าง ๆ ซึ่งจะต้องทำการสร้างวงจรควบคุม เพื่อควบคุมสัญญาณเหล่านี้ต่อไป ซึ่งทางเดินข้อมูลที่สมบูรณ์แสดงได้ดังรูปที่ 4.12 ภาคผนวก ข.7 และ Verilog HDL ของทางเดินข้อมูล



รูปที่ 4.12: ทางเดินข้อมูลสำหรับสถาปัตยกรรม nanoLADA

4.4 สัญญาณควบคุม (Control Signals)

เนื้อหาในส่วนนี้เทียบได้กับขั้นตอนที่สื่อของการออกแบบหน่วยประมวลผลกลาง คือ การวิเคราะห์สัญญาณควบคุมที่เกี่ยวข้อง

จากรูปที่ 4.12 ทางเดินข้อมูลจะแสดงด้วยเส้นทึบ และสัญญาณควบคุมจะแสดงด้วยเส้นประ ในที่นี้สัญญาณควบคุมประกอบด้วย

- ***sel_pc*** สำหรับเลือกค่าที่ต้องการ load เข้า PC (เลือกระหว่าง addr และค่าที่คำนวนได้)
กำหนดให้ 0 – เลือกค่าที่คำนวนได้ ($PC \leftarrow PC + 4$) และ 1 – เลือก addr
- ***sel_addpc*** สำหรับค่าที่ต้องการนำเข้ากับ PC (เลือกระหว่าง 0 และ sign_ext(imm))
กำหนดให้ 0 – เลือกค่า $30\{1'b0\}$ และ 1 – เลือก sign_ext(imm)
- ***sel_wr*** สำหรับเลือกค่าเรจิสเตอร์ที่ต้องการเขียนคืน (เลือกระหว่าง rd และ rt)
กำหนดให้ 0 – เลือกค่า rd และ 1 – เลือก rt
- ***sel_b*** สำหรับเลือกข้อมูลที่จะป้อนให้กับ ALU
กำหนดให้ 0 – เลือกค่า $30\{1'b0\}$ และ 1 – เลือก sign_ext(imm)
- ***sel_data*** สำหรับเลือกข้อมูลที่จะนำไปเขียนคืนในเรจิสเตอร์
กำหนดให้ 0 – เลือกค่า S และ 1 – เลือกค่าจากหน่วยความจำ

นอกจากสัญญาณเหล่านี้ยัง ยังมีสัญญาณ reg_wr เพื่อบอกให้เรจิสเตอร์ทำการบันทึกค่า, สัญญาณ mem_wr เพื่อบอกให้หน่วยความจำทำการบันทึกค่า และสัญญาณ alu_ops ซึ่งยังไม่ได้กล่าวถึง โดยกำหนดให้

- **reg_wr** เป็นสัญญาณเพื่อบอกให้เรจิสเตอร์ทำการบันทึกค่าจาก Data_in ไปยังค่าเรจิสเตอร์ที่กำหนดโดย Write_Reg
กำหนดให้ 1 – แทน การบันทึกข้อมูลใหม่
- **mem_wr** เป็นสัญญาณเพื่อบอกให้หน่วยความจำ ทำการบันทึกค่าจาก Data_in ไปยังค่า หน่วยความจำ ที่กำหนดโดยเลขที่อยู่
กำหนดให้ 1 – แทน การบันทึกข้อมูลใหม่

- alu_ops เป็นสัญญาณบอกให้ ALU ทำการ and, or, add เป็นต้น

หากสังเกตทางเดินข้อมูลจะพบว่ามี multiplexor สำหรับเป็นตัวเลือกทางเดินข้อมูลอยู่มากหลายสายแห่ง สัญญาณที่จะใช้ป้อนให้กับ multiplexor เพื่อเลือกทางเดินข้อมูลนี้ เรียกว่า สัญญาณควบคุม ซึ่งจะแตกต่างกันไปตามแต่ละคำสั่ง ดังนั้นที่มาของสัญญาณควบคุม คือ การตีความคำสั่งที่อ่านขึ้นมาจาก program memory เพื่อใช้เลือกทางเดินข้อมูลสำหรับการทำคำสั่งนั้น

ในส่วนของ ALU สมมุติให้การทำงานของ ALU เป็นดังตารางที่ 4.1 โดยภาคผนวก ฯ.3 แสดงรายละเอียดของ ALU (วิธีการการออกแบบ ALU ขอให้ผู้เรียนบทหวานจากการเรียน วิชา Digital Computer Logic เอง)

ตารางที่ 4.1: การทำงานของ ALU สำหรับ nanoLADA

alu_ops	การทำงาน
00	$S \leftarrow A + B ; z \leftarrow (S==0)$
01	$S \leftarrow A B ; z \leftarrow (S==0)$
10	$S \leftarrow A - B ; z \leftarrow (S==0)$

เนื่องจากในหัวข้อที่ 4.1.1 ที่กำหนดมีได้กำหนดค่า opcode ของสถาปัตยกรรมชุดคำสั่ง nanoLADA มาให้ ในที่นี้จึงขอกำหนดค่า opcode ให้กับคำสั่งต่าง ๆ ดังนี้

คำสั่ง	Opcode
ORI rt, rs, imm	010000
	
ORUI rt, rs, imm	010001

คำสั่ง	Opcode
	
ADD rd, rs, rt	000001
	
LW rt, rs, imm	011000
	
SW rt, rs, imm	011100
	
BEQ rs, rt, imm*4	100100
	
JMP addr*4	110000

จากค่า opcode ที่กำหนด เราสามารถสร้างตารางค่าความจริง (truth table) เพื่อกำหนดค่าສ້າງລູານຄວບຄຸມที่เกี่ยวข้องได้โดยพิจารณา opcode (และສ້າງລູານ zero จาก ALU)

เป็น input และพิจารณาสัญญาณควบคุม เป็น output ของวงจรสร้างสัญญาณควบคุม จะได้ผลดังตารางที่ 4.2

ตารางที่ 4.2: สัญญาณควบคุมสำหรับสถาปัตยกรรม nanoLADA

คำสั่ง Opcode	Zero (ALU)	sel_pc 0 – PC 1 - addr	sel_addpc 0 – 0 1 - addr	sel_wr 0 – rd 1 – rt	sel_b 0 – R[rt] 1 - imm	sel_data 0 – S 1 - M	reg_wr 1 – wr	mem_wr 1 – wr	alu_ops
ORI 010000	x	0	0	1	1	0	1	0	01
ORUI 010001	x	0	0	1	1	0	1	0	01
ADD 000001	x	0	0	0	0	0	1	0	00
LW 011000	x	0	0	1	1	1	1	0	00
SW 011100	x	0	0	x	1	x	0	1	00
BEQ 100100	0 1	0 0	0 1	x	x	x	0	0	10
JMP 110000	x	1	x	x	x	x	0	0	00

ลองสังเกตตารางที่ 4.2 จะพบว่า alu_ops สามารถสร้างได้จาก 3 บิตหน้าของ opcode กล่าวคือ หาก opcode ขึ้นต้นด้วย 010 จะได้ค่า alu_ops เป็น 01 หาก opcode ขึ้นต้นด้วย 000 หรือ 011 ค่า alu_ops จะเป็น 00 และหาก opcode ขึ้นต้นด้วย 100 ค่า alu_ops จะเป็น 10 เป็นต้น ซึ่งหากผู้เรียนมีความเชี่ยวชาญ digital computer logic ก สามารถที่จะใช้ Karnaugh map มาทำการสร้างวงจรดังกล่าวໄได้อย่างรวดเร็ว

หากต้องสร้างวงจรเพื่อสร้างสัญญาณควบคุมขึ้นมาจากการนี้สามารถดำเนินการได้หลายวิธี เช่น การหาผลรวมของผลคูณ (sum of product) รวมถึงการใช้การลดรูปด้วย Karnaugh map และ Quine–McCluskey algorithm หรือ การใช้ PLA หรือ ROM ช่วยในการ decode ซึ่งรายละเอียดดังกล่าวไม่อธิบายในขอบเขตของหนังสือเล่มนี้ แต่เพื่อประกอบความเข้าใจ จะขอยกตัวอย่างการสร้างสัญญาณควบคุมโดยการทำ sum of product แบบไม่มีการลดรูปดังนี้

กำหนดให้ $x[5..0]$ แทนแต่ละบิตของ opcode ก่อนอื่นให้ทำการสร้างผลคูณ²⁴ (product)โดยให้

$$G1 \text{ แทนคำสั่ง ORI (010000) คือ } G1 = \bar{x}_5 \circ x_4 \circ \bar{x}_3 \circ \bar{x}_2 \circ \bar{x}_1 \circ \bar{x}_0$$

$$G2 \text{ แทนคำสั่ง ORUI (010001) คือ } G2 = \bar{x}_5 \circ x_4 \circ \bar{x}_3 \circ \bar{x}_2 \circ \bar{x}_1 \circ x_0$$

$$G3 \text{ แทนคำสั่ง ADD (000001) คือ } G3 = \bar{x}_5 \circ \bar{x}_4 \circ \bar{x}_3 \circ \bar{x}_2 \circ \bar{x}_1 \circ x_0$$

$$G4 \text{ แทนคำสั่ง LW (011000) คือ } G4 = \bar{x}_5 \circ x_4 \circ x_3 \circ \bar{x}_2 \circ \bar{x}_1 \circ \bar{x}_0$$

$$G5 \text{ แทนคำสั่ง SW (011100) คือ } G5 = \bar{x}_5 \circ x_4 \circ x_3 \circ x_2 \circ \bar{x}_1 \circ \bar{x}_0$$

$$G6 \text{ แทนคำสั่ง BEQ (100100) คือ } G6 = x_5 \circ \bar{x}_4 \circ \bar{x}_3 \circ x_2 \circ \bar{x}_1 \circ \bar{x}_0$$

$$G7 \text{ แทนคำสั่ง JMP (110000) คือ } G7 = x_5 \circ x_4 \circ \bar{x}_3 \circ \bar{x}_2 \circ \bar{x}_1 \circ \bar{x}_0$$

จากนั้นสามารถนำ G1 .. G7 มาหาผลบวก²⁵เพื่อสร้างสัญญาณควบคุมต่าง ๆ ได้ดังนี้

$$sel_pc = G7$$

$$sel_addpc = G6 + zero$$

$$sel_wr = G1 + G2 + G4$$

$$sel_b = G1 + G2 + G4 + G5$$

$$sel_data = G4$$

$$reg_wr = G1 + G2 + G3 + G4$$

$$mem_wr = G5$$

หากพิจารณาสัญญาณ sel_wr และ sel_b ในตารางที่ 4.1 จะพบว่าสามารถใช้ sel_b แทนได้เลย เนื่องจากสัญญาณ sel_wr ในตารางมีค่า x (don't care) การใช้ sel_b แทน จึงมีได้ทำให้ค่าที่ได้ผิดไปแต่อย่างใด

24 สำหรับผู้เรียนที่ไม่แน่ใจว่าจะได้ผลคูณคือการนำค่าต่าง ๆ มา AND กัน

25 ในทำนองเดียวกัน ผลบวกคือการนำค่าต่าง ๆ มา OR กัน

ปัจจุบันทางเลือกที่ดีกว่าคือ การเขียนบรรยายการทำงานด้วย Verilog HDL จากนั้นให้คอมไฟเลอร์สร้างวงจรขึ้นจาก code ที่กำหนดแทน ภาคผนวก ข.6 เป็นการใช้ Verilog HDL บรรยายพฤติกรรมของสัญญาณควบคุม

4.5 วิถีวิกฤตและความเร็วสัญญาณนาฬิกา

เมื่อเราได้ทางเดินข้อมูลและนำสัญญาณควบคุมมาต่อพ่วงรวมกันแล้ว ก็ถึงขั้นตอนของการวิเคราะห์วิถีวิกฤต (critical path) และการสร้างสัญญาณนาฬิกาให้กับหน่วยประมวลผล ในกรณีสัญญาณนาฬิกาจะต้องให้จังหวะในการทำคำสั่งให้เสร็จ ทั้งนี้ข้อควรระวัง คือ ทุกครั้งที่มีสัญญาณนาฬิกามา จะมีการเริ่มคำสั่งใหม่ ดังนั้น เวลาที่เว้นเพื่อให้วางจริง ผสมได้ทำงานจำต้องมากเพียงพอเพื่อให้ทุกอย่างทำงานเสร็จสิ้นไม่ว่าจะเป็นคำสั่งใดก็ตาม มิใช่นั้นค่าที่ควรจะเยี่ยงกลับคืนไปยัง rejister อาจจะผิดพลาดได้ การคำนวณเวลาที่ต้องรอการทำงานของแต่ละองค์ประกอบ จะอาศัยความรู้เรื่อง gate delay ซึ่งเป็นความรู้พื้นฐานเรื่อง digital computer logic ที่ผู้เรียนควรทราบมาก่อนเข่นกัน

เพื่อประกอบความเข้าใจ ลองทำการวิเคราะห์ดังนี้ จากทางเดินข้อมูลในรูปที่ 4.12 จะเห็นว่า มีเวลาที่ต้องรอให้ส่วนต่าง ๆ ของระบบทำงานเพื่อให้ได้ข้อมูลที่ต้องการดังนี้

1. ตอนเริ่มต้น Instruction $\leftarrow \text{MEM}[\text{PC}]$ จะต้องรอเวลาที่ได้ค่า PC ที่ต้องการ และรอหน่วยความจำ Program Memory อ่านค่า Instruction ที่ต้องการ (สมมุติให้เวลาที่ PC ทำงานเป็น 10 นาโนวินาที และเวลาที่หน่วยความจำทำงานเป็น 40 นาโนวินาทีตามลำดับ)
2. เมื่อได้คำสั่งแล้ว จะต้องรอเวลาที่วงจรควบคุมทำการสร้างสัญญาณควบคุมที่เกี่ยวข้อง (สมมุติให้การทำงานของวงจรควบคุมเป็น 40 นาโนวินาที)
3. รอค่าจากชุด rejister เพื่อนำค่า rejister ที่กำหนดโดย R[rs] และ/หรือ R[rt] มาแสดงผลที่ทางออก A และ B ของชุด rejister ชุด rejister ในกรณีที่เป็นแบบ I-type ที่จะต้องรอค่าที่ imm ผ่าน extender เพื่อให้ได้ค่าที่ต้องการตัวอย่างไรก็ตามเนื่องจาก extender และ ชุด rejister มีการทำงานที่ขนานกันได้ ดังนั้น เพียงรอเวลาที่มากที่สุดของข้อมูลชุดใดชุดหนึ่งเท่านั้น (สมมุติให้ rejister ทำงานที่ 20 นาโนวินาที และ extender ทำงานที่ 15 นาโนวินาที) ซึ่งในกรณี max (20 นาโนวินาที, 15 นาโนวินาที) = 20 นาโนวินาที
4. หลังจากได้ค่าที่ต้องการแล้ว ต้องให้ multiplexor ที่ควบคุมด้วยสัญญาณ sel_b ทำการเลือกค่าที่ต้องการ (10 นาโนวินาที)

5. กรณีที่มีการคำนวณค่าโดยใช้ ALU จะต้องรอให้ ALU ทำการประมวลผลจึงได้ผลลัพธ์ของมาที่ S (สมมุติให้ ALU ทำงานโดยใช้เวลา 60 นาโนวินาที)
6. ในกรณีที่เป็นการอ่านข้อมูลจาก Data Memory จะต้องรอให้หน่วยความจำทำงานเพื่อนำค่าข้อมูลที่ต้องการมาแสดงผล (สมมุติให้เวลาที่หน่วยความจำทำงานเป็น 40 นาโนวินาที)
7. หลังจากได้ค่าที่ต้องการแล้ว ต้องรอให้ multiplexor ที่ควบคุมด้วยสัญญาณ sel_data ทำการเลือกค่าที่ถูกต้องเพื่อไปเขียนคืนยัง Data_in

นอกจากนี้ยังมีเวลาที่ต้องรอ adder ทำการประมวลผล $PC \leftarrow PC + 4$ หรือกรณีอื่น ซึ่งไม่ได้กล่าวถึงในที่นี้ เพราะการทำงานอื่น ๆ สามารถทำงานไปพร้อมกับการประมวลผลขั้นตอนหลักที่กล่าวข้างต้นได้ และมักใช้เวลาอ่อนกว่าการทำงานข้างต้น

เพื่อความสะดวกสมมุติให้ multiplexor ทุกชุดทำงานที่ 10 นาโนวินาที จะพบว่าหน่วยประมวลผลกลางดังกล่าวจะใช้เวลาในการทำงาน 1 คำสั่งมากที่สุด คิดเป็น (1) $10 + 40 + (2) 40 + (3) 20 + (4) 10 + (5) 40 + (6) 20 + (7) 10$ คิดเป็นเวลา 190 นาโนวินาที

ซึ่งคิดเป็น clock rate ได้ว่า

$$\text{clock rate} = \frac{1}{(10+40+40+20+10+40+20+10)} \times 10^{-9}$$

$$\text{clock rate} = \frac{10^9}{190} = 5.26 \times 10^6 \text{ Hz}$$

หรือสรุปได้ว่า หน่วยประมวลผลกลางดังกล่าวจะทำงานได้ที่สัญญาณนาฬิกา 5.26 Mhz

อย่างไรก็ตามหากวิเคราะห์ที่การทำงานของหน่วยประมวลผลกลางในรูปข้างต้น พบว่าการประมวลผลแต่ละคำสั่งจะต้องใช้เวลาในการทำงานเท่ากันคือ 190 นาโนวินาที ทั้งที่ในบางคำสั่งนั้นสามารถทำเสร็จได้ภายในเวลาเพียง 150 นาโนวินาทีเท่านั้น (เช่น กรณีที่ไม่ต้องอ่านข้อมูลจากหน่วยความจำ) จึงได้มีความพยายามเพื่อลดเวลาในการทำงานดังกล่าวให้รอเท่าที่จำเป็น จากที่แสดงนี้ยังพบว่ามีองค์ประกอบหลายอย่างที่ทำงานช้ากัน เช่น ALU และ adder น่าจะสามารถทำงานแทนกันได้ หรือ instruction memory และ data memory น่าจะสามารถใช้งานคู่ประกอบชิ้นเดียวกันได้ การออกแบบหน่วยประมวลผลกลางจึงมีแนวคิดในรูปแบบอื่น ๆ อีกมากมาย เพื่อที่จะให้หน่วยประมวลผลกลางมีขนาดเล็กลง หรือ มีความเร็วในการประมวลผลมากขึ้น รายละเอียดและแนวทางจะกล่าวถึงในบทถัดไป

4.6 แบบฝึกหัดท้ายบท

1. จงอธิบายส่วนประกอบต่าง ๆ ภายในหน่วยประมวลผลกลาง
2. จงอธิบายขั้นตอนในการประมวลผลคำสั่งหนึ่งคำสั่งของหน่วยประมวลผลกลาง
3. จงแสดงการสร้าง extender ที่สามารถทำงานได้ตามรูปที่ 4.3
4. จงออกแบบ ALU ขนาด 32 บิตที่มีการทำงานตามตารางที่ 4.1
5. data path คืออะไร และ control คืออะไร มีความเหมือนหรือมีความสัมพันธ์กัน หรือไม่ อย่างไร
6. จากตารางที่ 4.2 จงแสดงการสร้างสัญญาณควบคุมด้วย PLA
7. จากโครงสร้างภายในของชุดเรจิสเตอร์ในรูปที่ 4.2 หากกำหนดให้ D-flipflop มีค่า delay เป็น 20 นาโนวินาที และ multiplexor และ decoder มีค่า delay เป็น 30 นาโนวินาทีแล้ว โครงสร้างของชุดเรจิสเตอร์ตั้งกล่าวคราวจะมีความเร็วในการทำงานเป็นเท่าใด
8. จากตารางที่ 4.1 จงแสดงการสร้างสัญญาณ alu_ops ด้วยวิธีการ sum of products
9. จากตัวอย่างทางเดินข้อมูลสำหรับสถาปัตยกรรม nanoLADA ในรูปที่ 4.12 หากต้องการเพิ่มคำสั่ง

SUB rd, rs, rt ; (R-type)

ความหมาย $R[rd] \leftarrow R[rs] - R[rt]$;

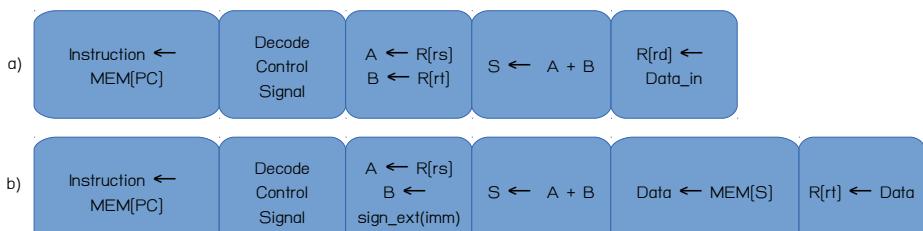
ทางเดินข้อมูลจะต้องมีการเปลี่ยนแปลงหรือไม่ อย่างไร
10. จากข้อ 9 หากกำหนด opcode ของ SUB เป็น 00_0010 จงสร้างตารางแสดงสัญญาณควบคุมที่เกี่ยวข้อง
11. จากหน่วยประมวลผลกลางแบบ single cycle ที่ได้ในบทนี้ หากโปรแกรมทดสอบมีจำนวนคำสั่งทั้งสิ้น 2 ล้านคำสั่ง หน่วยประมวลผลจะใช้เวลา CPU time คิดเป็นเท่าใด

5 หน่วยประมวลผลกลางแบบ multiple cycle

ลักษณะการออกแบบหน่วยประมวลผลกลางแบบ multiple cycle ช่วยให้เราสามารถพัฒนาหน่วยประมวลผลกลางที่ทำงานได้เร็วขึ้น และในบางกรณีอาจมีการแบ่งปันทรัพยากรบางส่วนได้ด้วย สาเหตุที่อาจจะทำงานได้เร็วขึ้นเนื่องจาก หน่วยประมวลผลกลาง จะไม่จำเป็นต้องรอการทำงานที่ยุ่งวนที่สุดในทุกกรณี ส่วนสาเหตุที่อาจแบ่งทรัพยากรใช้ร่วมกันได้นี้เนื่องจากสามารถใช้งานคู่ประกอบบางอย่างร่วมกันได้ เพราะใช้ไม่พร้อมกัน เช่น หากเปรียบเทียบ ALU และ adder กับเครื่องคิดเลข เมื่อมีคน 2 คน ต้องการใช้เครื่องคิดเลขสำหรับการประมวลผล คนหนึ่งต้องการใช้เครื่องคิดเลขสำหรับการบวก และ อีกคนหนึ่งใช้เครื่องคิดเลขสำหรับการคำนวณทั่วไป หากทั้ง 2 คนไม่มีความจำเป็นต้องใช้เครื่องคิดเลขพร้อมกัน เครื่องคิดเลขเพียงหนึ่งเครื่องก็เพียงพอสำหรับการใช้งานของคน 2 คน โดยผลัดกันใช้เครื่องคิดเลข ในทำนองเดียวกัน หน่วยประมวลผลกลางสามารถเหลือเวลาให้ใช้ ALU แทน adder สำหรับการประมวลผลค่า PC ได้

นอกจากนี้หากแบ่งการประมวลผลคำสั่งหนึ่งออกเป็นช่วงย่อย จะทำให้สามารถข้ามขั้นตอนบางขั้นที่ไม่จำเป็นในการทำงานของคำสั่งนั้นได้ ตัวอย่างเช่น รูปที่ 5.1 เมื่อให้ความกว้างแทนเวลา (a) เป็นเวลาที่แท้จริงของการทำงานของคำสั่ง ADD ส่วน (b) เป็นเวลาการทำงานของคำสั่ง LW ดังนั้นหากแบ่งการทำงานเป็นส่วนย่อยได้ จะช่วยให้หลายคำสั่งสามารถข้ามขั้นตอนการทำงานของการอ่านข้อมูลจากหน่วยความจำได้

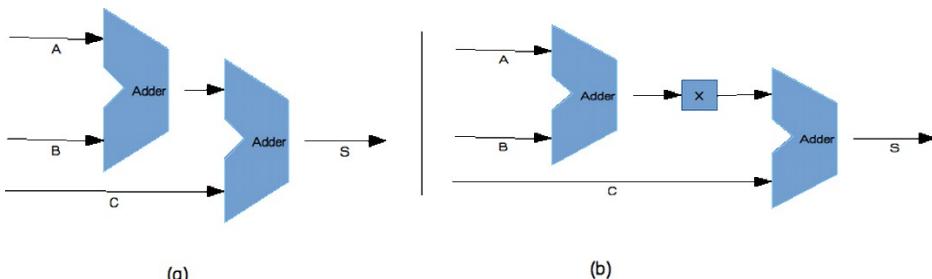
ยิ่งไปกว่านั้นการออกแบบวงจรเพื่อทำงานในปัญหาที่ซับซ้อนน้อยลง (แต่การจดใหญ่ออกเป็นวงจรที่เล็กลงแต่มีหลายขั้นตอน) ยังสามารถแก้ไขและตรวจสอบได้ง่ายกว่าการออกแบบวงจรขนาดใหญ่อีกด้วย



รูปที่ 5.1: เปรียบเทียบเวลา (a) คำสั่งที่ไม่ต้องอ่านข้อมูลจากหน่วยความจำ และ (b) คำสั่งที่ต้องอ่านข้อมูลจากหน่วยความจำ (กำหนดให้ความกว้างแทนเวลา)

จากแนวทางในการออกแบบโดยใช้การประมวลผลบางส่วนร่วมกัน และการออกแบบเพื่อให้การประมวลผลหนึ่งคำสั่งใช้หลายคาบของสัญญาณนาฬิกา จึงจำเป็นต้องมีการเพิ่มเรจิสเตอร์ภายในบางตัว เพื่อใช้ในการส่งผ่านค่าระหว่างรอบ

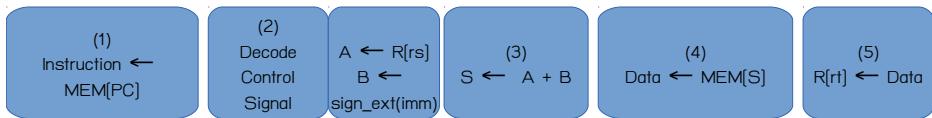
ลองพิจารณากรณีการประมวลผล $S \leftarrow A + B + C$ โดยใช้ adder 2 ชุด (รูปที่ 5.2 (a)) หากต้องการแบ่งการทำงานดังกล่าวออกเป็น 2 ช่วง สามารถทำได้โดยการเพิ่มตัวแปรเพื่อใช้จำค่าชั่วคราวก่อน เช่น ในช่วงแรก ให้ทำ $X \leftarrow A + B$ ทำงานก่อน จากนั้นจึงจะทำการประมวลผล $S \leftarrow X + C$ (รูปที่ 5.2 (b)) สมมุติเวลาของ adder แต่ละตัวมี gate delay เป็น 20 นาโนวินาที ในรูปที่ 5.2 (a) จะใช้สัญญาณนาฬิกาที่มีคาบไม่ต่ำกว่า 40 นาโนวินาที ในขณะที่รูปที่ 5.2 (b) จะใช้สัญญาณนาฬิกาที่คาบไม่ต่ำกว่า 20 นาโนวินาที จำนวน 2 รอบ แม้ว่ากรณีนี้จะมีไอดิจิตต่อตัวเดียวที่ต้องการเปลี่ยนแปลง $S \leftarrow A + B$ (ไม่มีการบวกด้วย C) รูปที่ 5.2 (b) จะสามารถลดเวลาให้ลดลงได้ถึงหนึ่ง ในขณะที่รูปที่ 5.2 (a) ถูกบังคับด้วยสัญญาณนาฬิกาให้รอถึง 40 นาโนวินาที



รูปที่ 5.2: ตัวอย่างการแบ่งขั้นตอนการทำงาน (a) การประมวลผลแบบ single cycle (b) การประมวลผลแบบ multiple cycle

หลักของการแบ่งขั้นตอนการทำงานที่ดีคือ การแบ่งให้แต่ละขั้นตอนมีเวลาในการประมวลผลใกล้เคียงกัน ในกรณีของรูปที่ 5.2 นั้น ถือได้ว่าเป็นการแบ่งที่สมบูรณ์แบบเนื่องจากทั้งสองขั้นตอน เป็นการบวกซึ่งใช้เวลาเท่ากันพอดี

ในกรณีของหน่วยประมวลผลกลางสำหรับสถาปัตยกรรม nanoLADA ที่เราสร้างขึ้น หากพิจารณาเวลาในการประมวลผล (รูปที่ 5.1) จะพบว่า สามารถแบ่งได้คร่าว ๆ เป็น 5 ขั้นตอน ดังรูปที่ 5.3



รูปที่ 5.3: การแบ่งขั้นตอนการทำงาน

จากรูป การประมวลผลคำสั่งแต่ละคำสั่นนั้นแบ่งการทำงานออกเป็นขั้นตอนอยู่ได้ 5 ขั้นตอน (ทั้งนี้ขึ้นอยู่กับโครงสร้างสถาปัตยกรรมคอมพิวเตอร์ และ สถาปัตยกรรมชุดคำสั่งที่ได้ทำการออกแบบไว้) ซึ่งในกรณีของ nanoLADA ในที่นี้ สามารถอธิบายการทำงานแต่ละขั้นตอนได้ดังนี้

1. **Instruction Fetch** คือ การประมวลผล $\text{Instruction} \leftarrow \text{MEM}[\text{PC}]$ เป็นการดึงคำสั่งที่มีเลขที่อยู่หน่วยความจำที่ซื้อด้วย PC เข้าสู่หน่วยประมวลผลกลาง พร้อมกับบอกค่า PC ให้ซื้อไปยังคำสั่งถัดไป
2. **Instruction Decode** และ **Register Fetch** คือ การตีความคำสั่งที่ถูกอ่านเข้ามาแล้วเพื่อสร้างสัญญาณควบคุม และ เตรียมข้อมูลในเรจิสเตอร์ รวมถึง extender ให้พร้อมสำหรับป้อนให้กับ ALU
3. **Execution** ในที่นี้ นอกจากจะเป็นการทำ้งานของ ALU ตามปกติแล้ว ยังรวมถึง address computation และ branch completion ด้วย ทั้งนี้การทำงานในขั้นตอน execution จะแตกต่างกันไปขึ้นอยู่กับคำสั่งที่อ่านเข้ามาได้ เช่น กรณีคำสั่งที่อ่านเข้ามาได้เป็นคำสั่งเกี่ยวกับการคำนวน ALU ก็จะคำนวนค่าตามคำสั่ง กรณีเป็นการอ่านข้อมูล ALU ก็จะคำนวนเลขที่อยู่ของหน่วยความจำที่ต้องการอ่านข้อมูล และกรณีการ branch ก็ให้ ALU คำนวนที่อยู่ของคำสั่งต่อไป (PC)
4. **Memory Access** กรณีที่คำสั่งซึ่งอ่านเข้ามาได้นั้นเป็นคำสั่งเกี่ยวกับการอ่าน เรียกข้อมูลจากหน่วยความจำ ในขั้นตอนนี้จะมีการอ่านหรือเขียนข้อมูล
5. **Write back** ซึ่งจะเกิดขึ้นเฉพาะกรณีที่ต้องมีการเขียนข้อมูลคืนกลับไปยังเรจิสเตอร์เท่านั้น (ในกรณีของ nanoLADA คือคำสั่ง ORI, ORUI, ADD, และ LW)

จากการทำงานใน 5 ขั้นตอนดังกล่าวข้างต้น จะเห็นว่าบางคำสั่ง (เช่น LI, LUI, ADD) จะใช้ 4 ขั้นตอนการทำงานโดยสามารถเข้ามายังขั้นตอน memory access ໄไปได้ ส่วนคำสั่ง SW จะใช้เพียง 4 ขั้นตอนคือ 1 – 4 โดยจะไม่มีขั้นตอนของการ write back และมีเพียง 1 คำ

สั่งเท่านั้น (LW) ที่ต้องทำงานครบทั้ง 5 ขั้นตอน ส่วนคำสั่งอื่น ๆ (BEQ และ JMP) จะทำงานได้โดยใช้เพียง 3 ขั้นตอนหรือน้อยกว่า

ทั้งนี้ในรายชื่อไม่สามารถสร้างหน่วยควบคุมได้ทันที เนื่องจากหลักการทำงานข้างต้นมิได้อธิบาย ถึงที่มาของข้อมูลแต่ละขั้นตอนโดยละเอียด เพาะระยังมิได้แสดงให้ทราบว่าจะต้องนำบิตใด ของคำสั่งที่อ่านเข้ามามาใช้อ้างอิง หรือนำข้อมูลที่ได้ไปเก็บที่เรจิสเตอร์ใดโดยละเอียด เพื่อให้ เห็นขั้นตอนการออกแบบที่ซัดเจน ลองพิจารณาขั้นตอนการออกแบบหน่วยประมวลผลกลาง ทั้ง 5 ขั้นตอน (จากหัวข้อ 4.1) อีกครั้งหนึ่ง จะพบว่าสิ่งที่เปลี่ยนไปคือ ตั้งแต่ขั้นตอนที่ 3 ซึ่งจะมีการออกแบบทางเดินข้อมูลที่เปลี่ยนไป ส่งผลให้ขั้นตอนที่ 4 และ 5 เปลี่ยนไปด้วย

เนื้อหาในบทนี้ เริ่มต้นด้วยการประเมินสมรรถนะของหน่วยประมวลผลกลางแบบ multiple cycle สำหรับสถาปัตยกรรมชุดคำสั่ง nanoLADA เปรียบเทียบกับหน่วยประมวลผลกลาง แบบ single cycle ที่ได้จากบทที่ 4 เพื่อให้เห็นแนวโน้มของสมรรถนะว่าจะได้เท่าใด จากนั้น จึงจะกลับเข้าสู่การออกแบบหน่วยประมวลผลกลางโดยเริ่มต้นจากการทำทางเดินข้อมูล และจบที่สัญญาณควบคุม

5.1 สมรรถนะของหน่วยประมวลผลกลางแบบ *multiple cycle*

หัว d นี้คือการลงทุนเพื่อปรับหน่วยประมวลผลกลางให้เป็นแบบ multiple cycle แล้ว ผลที่ได้จะเป็นเช่นไร สมรรถนะที่ได้คุ้มค่ากับการลงทุนหรือไม่ หากเวลาในการประมวลผลของหน่วยประมวลผลกลางสามารถคำนวณได้จากผลคูณของจำนวนคำสั่ง ค่าจำนวนรอบการประมวลผลต่อคำสั่ง และรอบสัญญาณนาฬิกาจากหน่วยประมวลผลกลางแบบ single cycle จะได้ว่าจำนวนรอบการประมวลผลต่อคำสั่งเป็น 1 สมมุติให้หน่วยประมวลผลกลาง แบบ single cycle มีค่ารอบสัญญาณนาฬิกาเป็น 210 นาโนวินาที ลองใช้ข้อมูลนี้ประกอบการวิเคราะห์ในตัวอย่างที่ 5.1

จากตัวอย่างจะเห็นว่า หากหน่วยประมวลผลกลางแบบ multiple cycle ที่ได้มีค่าเฉลี่ยของจำนวนรอบการประมวลผลต่อคำสั่งเป็น 4.7 จะได้สมรรถนะไม่ต่างจากเดิม อย่างไรก็ตาม หากหน่วยประมวลผลกลางที่ได้มีค่าเฉลี่ยจำนวนรอบการประมวลผลต่อคำสั่งน้อยกว่า 4.7 นั้นหมายความว่าสมรรถนะที่ได้จะดีกว่าเดิม

ตัวอย่างที่ 5.1: เปรียบเทียบสมรรถนะหน่วยประมวลผลกลางแบบ *single cycle* และ *multiple cycle*

กำหนดให้ หน่วยประมวลผลกลางแบบ *single cycle* ค่ารอบสัญญาณนาฬิกาเป็น 210 นาโนวินาที หากทำการปรับปรุงหน่วยประมวลผลกลางดังกล่าวให้เป็นแบบ *multiple cycle* แล้ว ได้ค่ารอบสัญญาณนาฬิกาใหม่เป็น 45 นาโนวินาที ค่าเฉลี่ยของการประมวลผล (average CPI) ควรจะเป็นเท่าใดเพื่อให้อย่างน้อยได้สมรรถนะเท่าเดิม

$$CPU Time = instruction count \times CPI \times Cycle Time$$

$$CPU Time_{single\ cycle} = instruction count \times 1 \times 210 (ns)$$

สมมุติว่าการปรับทางเดินข้อมูลและวงจรควบคุมทำให้เวลาของรอบสัญญาณนาฬิกาเป็น 45 นาโนวินาที (ต่อ 1 ขั้นตอนการทำงาน) จะได้ว่า

$$CPU Time_{multiple\ cycle} = instruction count \times average CPI \times 45 (ns)$$

ดังนั้นเพื่อให้ได้ สมรรถนะอย่างน้อยเท่าเดิม ค่าเฉลี่ย CPI ที่ได้ควรจะเป็น

$$CPU Time_{single\ cycle} = CPU TIME_{multiple\ cycle}$$

$$instruction count \times 1 \times 210 = instruction count \times average CPI \times 45$$

$$210 = average CPI \times 45$$

$$\therefore average CPI = \frac{210}{45} \approx 4.7$$

คราวนี้เราลองวิเคราะห์ต่อว่า หากต้องการให้ค่าเฉลี่ยของจำนวนรอบการประมวลผลต่อคำสั่งที่ 4.7 จะมีความเป็นไปได้มากน้อยแค่ไหน หากพิจารณาคำสั่งส่วนใหญ่ใช้ 4 รอบ (หรือน้อยกว่า) และมีเพียงคำสั่ง LW เท่านั้นที่ใช้ 5 รอบจะพบว่า หากมีคำสั่ง LW ไม่เกิน 20% ของคำสั่งทั้งหมด ก็จะได้ค่าเฉลี่ยจำนวนรอบการทำงานต่อคำสั่งที่ 4.2 ซึ่งต่ำกว่า 4.7 แล้ว ที่มากองค่า 20% ให้ลองพิจารณาตัวอย่างที่ 5.2

ตัวอย่างที่ 5.2: การคำนวณค่าเฉลี่ย CPI ของหน่วยประมวลผลกลางแบบ multiple cycle

กำหนดให้คำสั่ง LW มีค่าจำนวนรอบการประมวลผลเป็น 5 ส่วนคำสั่งอื่น ๆ ในระบบมีค่าจำนวนรอบการประมวลผลเป็น 4 หากต้องการให้หน่วยประมวลผลกลางมีค่าเฉลี่ยจำนวนรอบการประมวลผลเป็น 4.2 จะต้องมีคำสั่ง LW ไม่เกินกี่เฟอร์เซ็นต์ของคำสั่งทั้งหมดที่มีในระบบ

ให้จำนวนคำสั่ง LW ในระบบมีสัดส่วนเป็น k ค่าเฉลี่ยจำนวนรอบการประมวลผลคำนวนได้จาก

$$\text{average CPI} = (5 \times k) + (4 \times (1 - k))$$

$$\text{average CPI} = 5k + 4 - 4k$$

เนื่องจากค่าเฉลี่ยจำนวนรอบการประมวลผลที่ต้องการคือ 4.2 ดังนี้

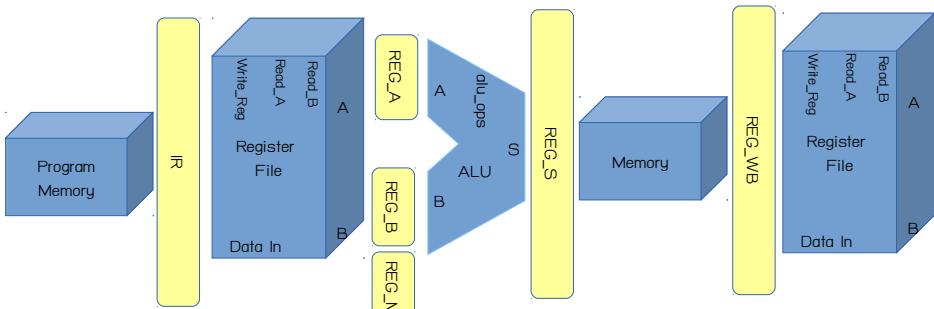
$$4.2 = 5k + 4 - 4k$$

$$4.2 - 4 = k$$

$$\therefore k = 0.2$$

5.2 ทางเดินข้อมูลสำหรับสถาปัตยกรรม nanoLADA แบบ multiple cycle

เนื่องจากในแต่ละสัญญาณนาฬิกา จะต้องมีการส่งต่อค่าที่จะใช้เป็น input ของวงจรในส่วนถัดไป ดังนั้นแนวทางในการปรับปรุงทางเดินข้อมูลให้รองรับการทำงานจึงตรงไปตรงมาคือทำการเพิ่ม(แทรก) เรจิสเตอร์สำหรับการส่งต่อค่าระหว่างแต่ละสถานะ รูปที่ 5.4 แสดงเรจิสเตอร์ที่ถูกแทรกเข้าไปเพื่อให้สามารถประมวลผลแบบ multiple cycle ได้ (ในรูปจะตัดสายสัญญาณออก เพื่อให้ดูได้ง่าย)



รูปที่ 5.4: การแทรกเรจิสเตอร์เงาเพื่อปรับทางเดินข้อมูลให้รองรับการทำงานแบบ multiple cycle

จากทางเดินข้อมูลใหม่ จะเห็นว่ามีเรจิสเตอร์ที่เกิดขึ้นมาใหม่อีก 5 ชุด (IR, REG_A, REG_B, REG_S, REG_WB) เนื่องจากเรจิสเตอร์เหล่านี้ไม่สามารถเข้าถึงได้ด้วยการเขียนโปรแกรม หลายตำแหน่งเรียกเรจิสเตอร์ ในลักษณะนี้ว่าเรจิสเตอร์เงา (shadow register) ในการออกแบบจึงนิยมอธิบายการทำงานโดยอ้างภาษา Register-Transfer language (RTL) สำหรับการทำงานแต่ละขั้น ข้อสังเกตคือ RTL ในตอนนี้จะเป็น RTL ที่แสดงการทำงานของเรจิสเตอร์ที่เกิดขึ้นใหม่เหล่านี้ ซึ่งไม่สามารถเข้าถึงจากการโปรแกรมด้วย เพื่อกันความสับสน เราจึงเรียก RTL ที่อธิบายการทำงานของคำสั่งตามสถาปัตยกรรมชุดคำสั่งว่า *Logical RTL* (ตามที่เคยแสดงในบทที่ 4) และจะเรียก RTL ที่อธิบายถึงการทำงานทุกขั้นตอนและแสดงค่าเรจิสเตอร์เงา (shadow register) เหล่านี้ว่า *physical RTL* (ตามความนิยมจะอธิบาย physical RTL แยกตามรอบการทำงานที่เกิดขึ้นด้วย)

จากสถาปัตยกรรมชุดคำสั่ง nanoLADA สามารถอธิบายการทำงานด้วย logical RTL และ physical RTL ได้ดังนี้

ORI rt, rs, imm ; (I-type)

Logical RTL:

$$R[rt] \leftarrow R[rs] \mid \text{zero_ext}(imm); \quad PC \leftarrow PC + 4$$

Cycle	Physical RTL
1	$IR \leftarrow \text{MEM}[PC]$ $PC \leftarrow PC + 4$
2	$REG_A \leftarrow R[rs]$ $REG_B \leftarrow \text{zero_ext}(imm)$
3	$REG_WR \leftarrow REG_S \leftarrow REG_A \mid REG_B$
4	$R[rt] \leftarrow REG_WR$

ORUI rt, rs, imm ; (I-type)

Logical RTL:

$$R[rt] \leftarrow R[rs] \mid \text{zero_pad}(imm); \quad PC \leftarrow PC + 4$$

Cycle	Physical RTL
1	$IR \leftarrow \text{MEM}[PC]$ $PC \leftarrow PC + 4$
2	$REG_A \leftarrow R[rs]$ $REG_B \leftarrow \text{zero_pad}(imm)$
3	$REG_WR \leftarrow REG_S \leftarrow REG_A \mid REG_B$
4	$R[rt] \leftarrow REG_WR$

ADD rd, rs, rt ; (R-type)

Logical RTL:

$$R[rd] \leftarrow R[rs] + R[rt]; \quad PC \leftarrow PC + 4$$

1	$IR \leftarrow \text{MEM}[PC]$ $PC \leftarrow PC + 4$
2	$REG_A \leftarrow R[rs]$ $REG_B \leftarrow R[rt]$

3	$\text{REG_WR} \leftarrow \text{REG_S} \leftarrow \text{REG_A} + \text{REG_B}$
4	$\text{R[rt]} \leftarrow \text{REG_WR}$

LW rt, rs, imm ; (l-type)

Logical RTL:

$$\text{R[rt]} \leftarrow \text{MEM}[\text{R[rs]} + \text{sign_ext}(\text{imm})]; \quad \text{PC} \leftarrow \text{PC} + 4$$

Cycle	Physical RTL
1	$\text{IR} \leftarrow \text{MEM}[\text{PC}]$ $\text{PC} \leftarrow \text{PC} + 4$
2	$\text{REG_A} \leftarrow \text{R[rs]}$ $\text{REG_B} \leftarrow \text{sign_ext}(\text{imm})$
3	$\text{REG_S} \leftarrow \text{REG_A} + \text{REG_B}$
4	$\text{REG_WR} \leftarrow \text{MEM}[\text{REG_S}]$
5	$\text{R[rd]} \leftarrow \text{REG_WR}$

SW rt, rs, imm ; (l-type)

Logical RTL:

$$\text{MEM}[\text{R[rs]} + \text{sign_ext}(\text{imm})] \leftarrow \text{R[rt]}; \quad \text{PC} \leftarrow \text{PC} + 4$$

Cycle	Physical RTL
1	$\text{IR} \leftarrow \text{MEM}[\text{PC}]$ $\text{PC} \leftarrow \text{PC} + 4$
2	$\text{REG_A} \leftarrow \text{R[rs]}$ $\text{REG_B} \leftarrow \text{sign_ext}(\text{imm})$ $\text{REG_M} \leftarrow \text{R[rt]}$
3	$\text{REG_S} \leftarrow \text{REG_A} + \text{REG_B}$
4	$\text{MEM}[\text{REG_S}] \leftarrow \text{REG_M}$

*BEQ rs, rt, imm*4 ; (l-type)*

Logical RTL:

```
If (R[rs] == R[rt]) then
  PC  $\leftarrow$  PC + 4 + (sign_ext(imm) * 4)
else PC  $\leftarrow$  PC + 4
```

Cycle	Physical RTL
1	$IR \leftarrow MEM[PC]$ $PC \leftarrow PC + 4$
2	(ตรวจสอบ decode สัญญาณควบคุมทำงาน)
3	If ($R[rs] == R[rt]$) THEN $PC \leftarrow PC + (\text{sign_ext}(imm) * 4)$ ENDIF

หมายเหตุ ในทางปฏิบัติสามารถดัดแปลงชุดเรจิสเตอร์ให้แสดงค่า logic 1 หรือ 0 ได้เมื่อทำการเปรียบเทียบค่า $R[rs]$ และ $R[rt]$ ได้ทันที จะช่วยให้สามารถทราบผลการเปรียบเทียบได้ภายในรอบที่สอง และไม่จำเป็นต้องใช้ REG_A และ REG_B

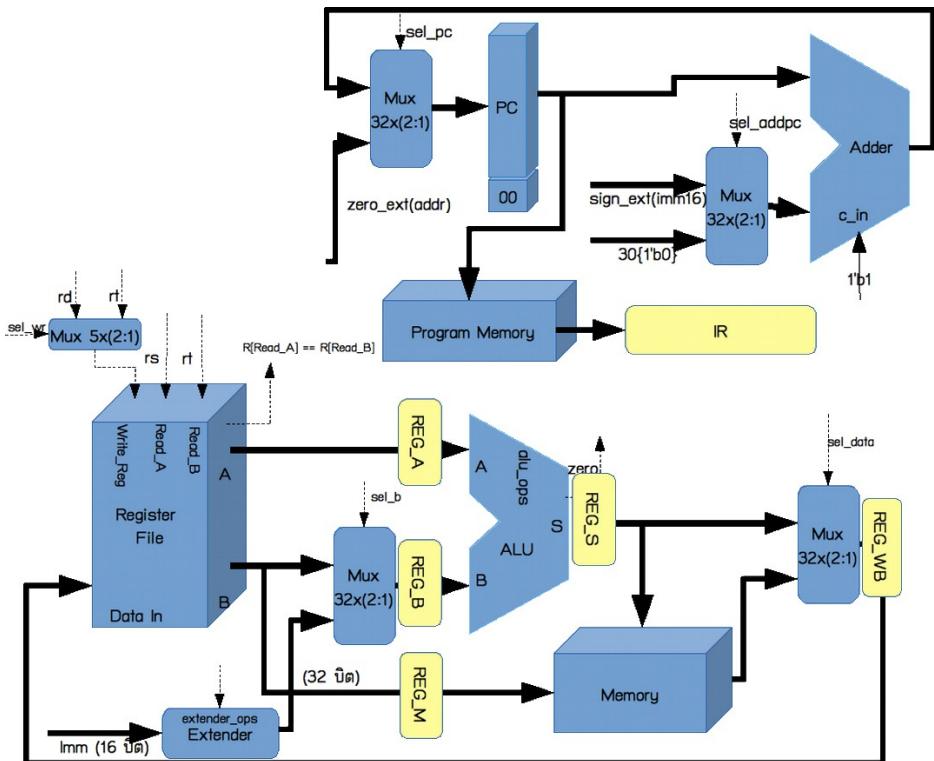
JMP addr*4 ; (J-type)

Logical RTL:

$PC \leftarrow (\text{addr} * 4)$

Cycle	Physical RTL
1	$IR \leftarrow MEM[PC]$ $PC \leftarrow PC + 4$
2	(ตรวจสอบ decode สัญญาณควบคุมทำงาน)
3	$PC \leftarrow \text{addr} * 4$

จาก physical RTL ที่ได้ เมื่อสร้าง data path ที่สมบูรณ์ จะได้ทางเดินข้อมูลดังรูปที่ 5.5 ซึ่งคล้ายกับทางเดินข้อมูลแบบ single cycle เพียงแต่มีเรจิสเตอร์งานแทรกรักษาไว้เพื่อเป็นที่พักข้อมูลสำหรับส่งให้ยังหน่วยอื่นประมวลผลในสัญญาณน้ำพิกัดไป

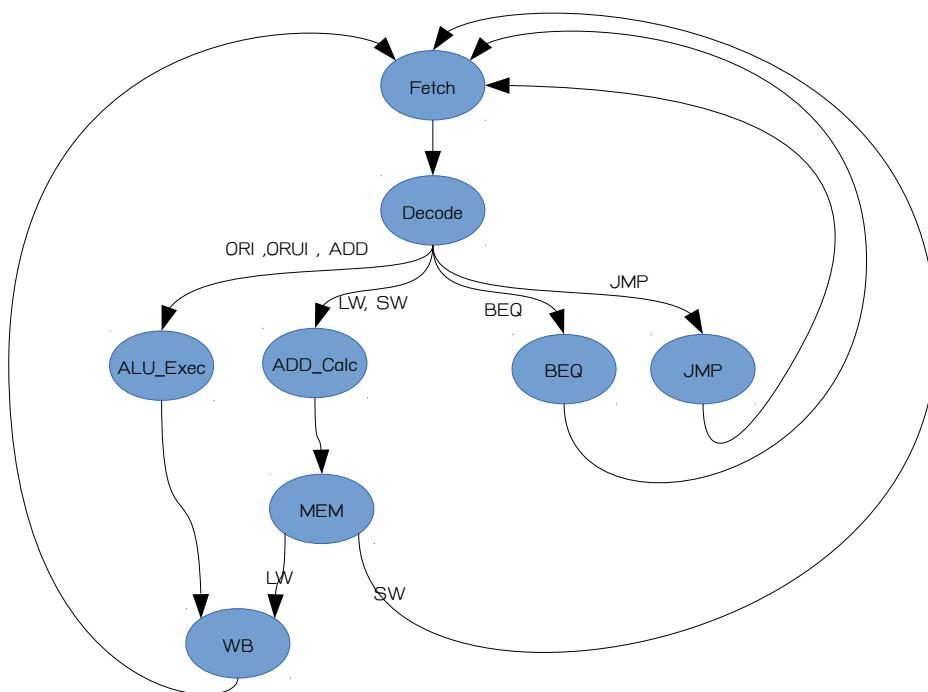


รูปที่ 5.5: ทางเดินข้อมูลสำหรับ multiple-cycle processor

5.3 สัญญาณควบคุมสำหรับ multiple cycle processor

หลังจากที่ได้ทางเดินข้อมูลเป็นอันเสร็จสิ้นการทำงานในขั้นตอนที่สามของการออกแบบ คราวนี้มาถึงขั้นตอนที่สี่และขั้นตอนที่ห้า คือ การสร้างสัญญาณควบคุม ในขั้นตอนนี้หากจะวิเคราะห์ให้ดี จะเห็นว่าสัญญาณควบคุมต่าง ๆ ยังคงเหมือนเดิม เพียงแต่การทำงานจะแยกย่อยไปแต่ละสัญญาณนาฬิกา ดังนั้นการสร้างสัญญาณควบคุมในขั้นตอนนี้จึงเป็นการสร้างวงจรเชิงลำดับ (sequential circuit) แทน

จากความรู้เดิมในการออกแบบวงจรเชิงลำดับ เราเริ่มต้นการออกแบบโดยการสร้างแผนภาพสถานะ (state diagram) หรือ การเขียน ASM chart เพื่อแสดงการเคลื่อนไหวและเงื่อนไขในการเปลี่ยนสถานะต่าง ๆ สร้างฟังก์ชันเอาท์พุต และ สร้างฟังก์ชันการเปลี่ยนสถานะ รูปที่ 5.6 แสดงแผนภาพสถานะของหน่วยประมวลผลกลางนี้



รูปที่ 5.6: state diagram แสดงการทำงานของหน่วยประมวลผลกลางแบบ multiple cycle

รูปที่ 5.6 แสดงเพียงสถานะ (ในทางปฏิบัติ อาจจะสามารถถูกรวบ state บางอันได้อีก) โดยยังไม่มีสัญญาณควบคุมที่เกี่ยวข้อง (ทั้งนี้เพื่อไม่ให้ภาพดูรก จนอ่านไม่ออก) เพื่อให้ແນກภาพสถานะสมบูรณ์จะต้องระบุสัญญาณที่เกี่ยวข้องในตารางที่ 5.1

ตารางที่ 5.1: สัญญาณควบคุมสำหรับหน่วยประมวลผลแบบ multiple cycle

สถานะ	สัญญาณที่เกี่ยวข้อง	สถานะถัดไป
Fetch	$IR \leftarrow MEM[PC]$	Decode
Decode	$IF (op[5..4] == 01) THEN ; I\text{-Type}$ $sel_wr \leftarrow 1$ $sel_b \leftarrow 1$	ขึ้นกับคำสั่ง ALU_Exec ADD_Calc

สถานะ	สัญญาณที่เกี่ยวข้อง	สถานะถัดไป
	ENDIF (wait for decoder)	BEQ JMP
ALU_Exec	IF (opcode in {ORI, ORUI}) THEN alu_ops \leftarrow OR ELSE alu_ops \leftarrow ADD ENDIF	WB
ADD_Calc	alu_ops \leftarrow ADD	MEM
MEM	sel_data \leftarrow 1 IF (op[2] == 1) THEN mem_wr \leftarrow 1 ENDIF	WB
WB	reg_wr=1	Fetch
BEQ	If (R[Read_A] == R[Read_B]) THEN sel_addpc \leftarrow 1 END IF	Fetch
JMP	sel_pc \leftarrow 1	Fetch

หมายเหตุ สัญญาณที่ไม่ระบุให้ถือว่าเป็น 0

จากแผนภาพสถานะ สามารถออกแบบวงจรได้โดยการใช้งานชิ้นเซมิคอนดักเตอร์ตัวเดียว ROM หรือ PLA ก็ได้ ทั้งนี้เนื่องจากมีการซ้อนกันของหลายควบคุมที่ได้จะมีอินพุตเป็นเรจิสเตอร์คำสั่ง และสถานะของหน่วยควบคุม ส่วนเอาท์พุตจะเป็นสัญญาณสำหรับควบคุม multiplexor และ ALU พร้อมกับการเปลี่ยนสถานะ ในที่นี้จะไม่ออกล่าวยละเอียดในการออกแบบมากนัก หากผู้อ่านสนใจเพิ่มเติมสามารถศึกษาได้จากการออกแบบจริงลำดับที่รีบไป

ทั้งนี้การออกแบบสัญญาณควบคุมที่แสดงในตารางที่ 5.1 ยังไม่ได้รวมการออกแบบเพื่อให้รองรับการเกิด exception และ interrupt ซึ่งจะทำให้โครงสร้างหน่วยประมวลผลกลางที่ได้มีความซับซ้อนยิ่งขึ้นไปอีก

5.4 ไมโครโปรแกรม (Microprogram)

ทางเลือกหนึ่งที่บริษัทผู้ผลิตหน่วยประมวลผลกลางเลือกใช้ในการสร้างสัญญาณควบคุมแทนการสร้างวงจรเชิงลำดับตามปกติคือ ไมโครโปรแกรม แนวคิดนี้ให้การออกแบบส่วนควบคุมของหน่วยประมวลผลกลางมีลักษณะคล้ายกับการเขียนโปรแกรม ซึ่งช่วยอำนวยความสะดวกในการแก้ไขปัญหาและตรวจสอบข้อผิดพลาดระหว่างการออกแบบ โดยเจียนโปรแกรมด้วยคำสั่งพิเศษที่เรียกว่าไมโครอินสตรัคชัน (microinstruction) จากนั้นทำการโหลดไมโครอินสตรัคชันลงสู่หน่วยประมวลผลกลาง การทำแบบนี้ นอกจากจะอำนวยความสะดวกในการออกแบบแล้ว ยังช่วยให้ปรับปรุงข้อบกพร่องของหน่วยประมวลผลกลาง หรือจำลองหน่วยประมวลผลกลางให้สามารถประมวลสถาปัตยกรรมชุดคำสั่งแบบอื่น ๆ ที่หลากหลายได้ (ทั้งนี้ต้องอยู่บนเงื่อนไขว่าทางเดินข้อมูลและส่วนประกอบมากพอ) ข้อดีข้อเสียของการออกแบบด้วยไมโครโปรแกรมอธิบายได้ดังด้านอย่างต่อไปนี้

IBM System/360²⁶ เป็นสถาปัตยกรรมระบบเมนเฟรมของ IBM และเป็นต้นแบบของการออกแบบด้วยไมโครโปรแกรม วางแผนครั้งแรกในปี 1964 (พ.ศ. 2507) ด้วยเทคโนโลยีไมโครโปรแกรม System/360 สามารถจำลอง (emulate) สถาปัตยกรรมชุดคำสั่งอื่นของ IBM ได้ (เช่น IBM 1400 และ IBM 7094 ซึ่งมีมาก่อน) ผู้ใช้งานจึงสามารถใช้ซอฟต์แวร์เก่าบนสถาปัตยกรรมใหม่ได้

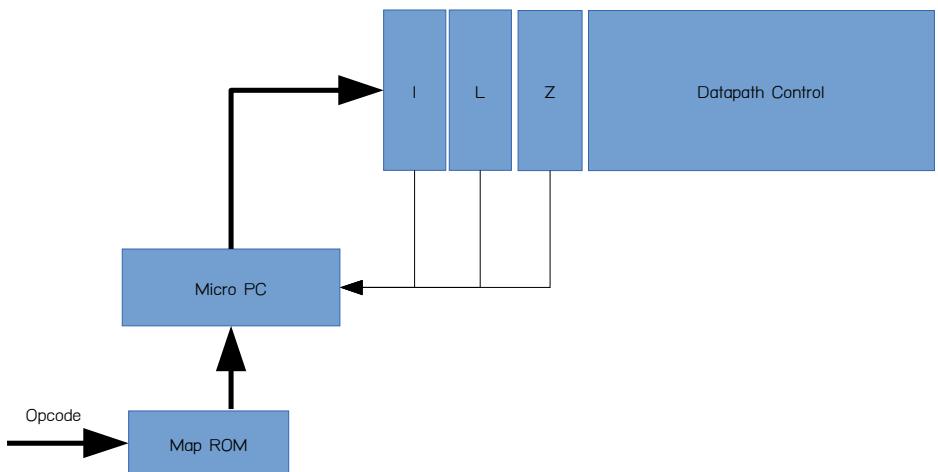
ข้อเสียของการออกแบบโดยไม่ใช้ไมโครโปรแกรม เช่น ในปี 1994 (พ.ศ. 2537) บริษัท Intel เคยพบข้อผิดพลาดจากการหารเลขคณิตด้วยคำสั่ง FDIV ในหน่วยประมวลผลกลางรุ่น Pentium²⁷ แม้ข้อผิดพลาดดังกล่าวจะไม่ได้เกิดขึ้นทุกครั้งแต่เป็นข้อผิดพลาดที่ไม่สามารถแก้ไขได้จนต้องเรียกคืน หลังจากที่ Intel ได้เปลี่ยนมาใช้ไมโครโปรแกรม Intel สามารถแก้ปัญหาของหน่วยประมวลผลกลางได้หลักครั้งต่อมาโดยการให้ผู้ใช้ปรับปรุงไมโครโปรแกรม (เช่น ผ่านการอัปเดทอัตโนมัติของระบบปฏิบัติการสมัยใหม่) โดยไม่ต้องมีการเรียกคืน

5.4.1 Microsequencer

แนวทางหนึ่งในการสร้างสัญญาณควบคุมด้วยไมโครโปรแกรมคือ การใช้ micro PC (ล็อกบีบี PC ของหน่วยประมวลผลกลาง) เป็นตัวควบคุมลำดับการทำงาน และทำการโปรแกรมสัญญาณควบคุมกับ map ROM สำหรับแปลงค่า opcode เป็นเลขที่อยู่ของไมโครโปรแกรม เมื่อนำส่วนต่าง ๆ มาประกอบกันจะเป็น microsequencer ดังแสดงในรูปที่ 5.7

26 ดูรายละเอียดเพิ่มเติมที่ https://en.wikipedia.org/wiki/IBM_System/360

27 ดูรายละเอียดเพิ่มเติมที่ https://en.wikipedia.org/wiki/Pentium_FDIV_bug

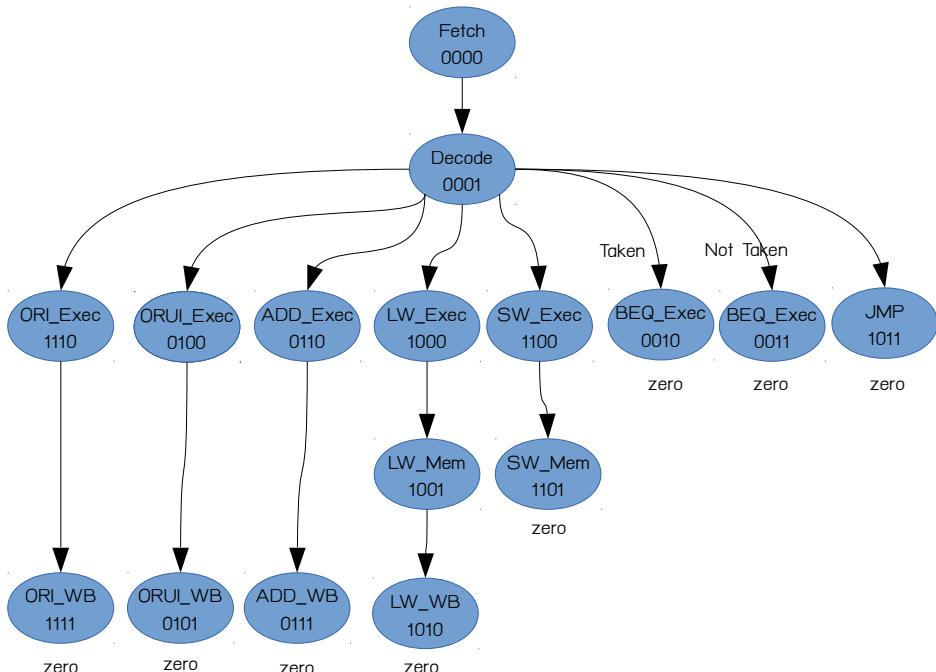


รูปที่ 5.7: โครงสร้างของ Microsequencer

จากรูป micro PC คือ ตัวนับ (counter) ที่มีสัญญาณควบคุม 3 สัญญาณได้แก่ Z, I, และ L ตามลำดับ โดย Z ใช้สำหรับการ reset ให้ค่า micro PC เป็น 0, I ใช้สำหรับการนับค่าเพิ่ม 1 และ L สำหรับการ load ค่าจาก map ROM ภาคผนวก ข.8 เป็น Verilog HDL แสดงโครงสร้างภายในของ micro PC

ส่วนของสัญญาณ datapath control²⁸ (รวมถึงสัญญาณ I, L, Z) เป็นไมโครอินสตรัคชันเพื่อสร้างวงจรเชิงลำดับและสัญญาณควบคุมที่เกี่ยวข้อง แต่ละคำสั่งจะใช้ 1 รอบสัญญาณนาพิกาในการประมวลผลและแต่ละบิตแทนสัญญาณควบคุมที่เกี่ยวข้อง รายละเอียดของลำดับการทำงานแสดงในรูปที่ 5.8 และข้อมูลสายสัญญาณใน ตารางที่ 5.2

จาก ตารางที่ 5.2 สงเกตว่าเลขที่อยู่ของคำสั่งคือเลขเข้ารหัสของสถานะในรูปที่ 5.8 เนื่องจาก micro PC เป็นตัวนับแบบหนึ่ง กรณีที่ลำดับต่อเนื่องกันจึงใช้สัญญาณ I เพื่อให้เพิ่มสถานะ และกรณีที่ลำดับไม่ได้ต่อเนื่องกัน จึงใช้สัญญาณ L เพื่อทำการอ่านค่า micro PC ใหม่จาก map ROM แทน และเมื่อต้องการกลับไปที่สถานะเริ่มต้น จึงใช้สัญญาณ Z เพื่อทำการตั้งค่าเริ่มต้นใหม่ (0000) ในแต่ละบรรทัดของ datapath control จะเป็นสัญญาณควบคุมที่เกี่ยวข้อง



รูปที่ 5.8: state diagram สำหรับไมโครโปรแกรม

ตารางที่ 5.2: ไมโครอินสตั๊กชันของสัญญาณควบคุมบางส่วน

Addr (Micro PC)	zero	I, L, Z	IR	sel_pc	sel_ addpc	sel_wr	sel_b	sel_ data	reg_wr	mem_ wr	alu_ ops
Fetch 0000	?	I	1								
Decode 0001	0 1	L I									
BEQ Taken 0010		Z			1						
BEQ -Taken 0011		Z									
ORUI EX		I					1			'01	

Addr (Micro PC)	zero	I, L, Z	IR	sel_pc	sel_addpc	sel_wr	sel_b	sel_data	reg_wr	mem_wr	alu_ops
0100											
ORUI WB 0101		Z				1			1		
ADD EX 0110		I									'00
ADD WB 0111		Z							1		
LW EX 1000		I				1					'00
LW MEM 1001		I							1		
LW WB 1010		Z				1			1		
JMP 1011		Z		1							
SW EX 1100		I				1					'01
SW MEM 1101		Z							1		
ORI EX 1110		I									'01
ORI WB 1111		Z				1			1		

ในตารางที่ 5.2 นี้ยังมีได้ระบุสัญญาณที่เกี่ยวของกับเรซเตอร์เงาอีกหลายอัน ซึ่งผู้เขียนจะขอทิ้งรายละเอียดของการทำไว้เป็นแบบผิกหัดให้ผู้เรียนได้ทำต่อไป

5.5 แบบฝึกหัดท้ายบท

- การออกแบบหน่วยประมวลผลกลางโดยใช้ single cycle และ multiple cycle มีข้อดีข้อเสียต่างกันอย่างไร จงยกตัวอย่างประกอบ
- ในการประมวลผลคำสั่งแต่ละคำสั่นนั้น หน่วยประมวลผลกลางมีขั้นตอนในการทำงานอย่างไร
- จากภาษาแอสเซมบลีของสถาปัตยกรรม nanoLADA ที่กำหนดให้ หากออกแบบหน่วยประมวลผลกลางให้ทำงานแบบ multiple cycle จงตอบค่าตามต่อไปนี้

```

LW      $r2, 0($r3)
LW      $r3, 4($r3)
BEQ    $r2, $r3, end_program #assume not taken
ADD    $r5, $r2, $r3
SW      $r5, 8($r3)
end_program:....           # end of program

```

- การทำงานในแต่ละคำสั่งจะใช้เวลาเท่ากัน
- รอบที่ 7 หน่วยประมวลผลกลางกำลังทำงานอยู่รอบ
- โปรแกรมดังกล่าวข้างต้นจะใช้เวลาในการทำงานทั้งสิ้นกี่รอบ และคิดเป็นกี่วินาที หากรอบสัญญาณนาฬิกาเป็น 40ns
- จากรูปที่ 5.6 และ ตารางที่ 5.1 จงสร้าง state machine สำหรับสร้างสัญญาณควบคุมที่กำหนด
- จาก data path ในรูปที่ 5.5 จงแสดงการปรับปรุงเพื่อให้สามารถใช้ ALU แทน adder ของการประมวลผล PC ได้
- หากต้องการให้หน่วยประมวลผลกลางตามรูปที่ 5.6 สนับสนุนการทำงานของ exception และ interrupt แล้ว ภาพของ state diagram ที่ได้ จะเปลี่ยนไปอย่างไร จงแสดงการออกแบบประกอบการอธิบาย
- จากตัวอย่าง Verilog HDL ของหน่วยประมวลผลกลางแบบ single cycle ในภาคผนวก ฯ จงปรับแต่งทางเดินข้อมูลให้รองรับการทำงานแบบ multiple cycle

พร้อมทั้งปรับหน่วยความคุมให้เป็นแบบไมโครโปรแกรมโดยอ้างสัญญาณต่าง ๆ จากตารางที่ 5.2

8. จงเปรียบเทียบข้อดีข้อเสียของไมโครโปรแกรมเมื่อเทียบกับการสร้างวงจรเชิงล้ำดับแบบปกติ

6 การเพิ่มสมรรถนะด้วย Pipeline

การทำ pipeline เป็นการเพิ่มสมรรถนะให้กับหน่วยประมวลผลกลางในลักษณะของการเพิ่ม throughput ก่อให้เกิด มีการทำงานหลายคำสั่งพร้อมกันในเวลาหนึ่ง ตัวอย่างเช่น สมมุติว่าการเติร์นข้าวสาร ประกอบด้วยขั้นตอนย่อย 3 ขั้น คือ ขั้นที่ 1 การร่อนข้าวเปลือกเพื่อคัดสิ่งแปรปรวนออกที่ป่นมากับข้าวเปลือก ขั้นที่ 2 นำข้าวเปลือกที่ได้จากขั้นแรกมาตำเพื่อให้เมล็ดข้าวหลุดออกจากเปลือกข้าว และ ขั้นที่ 3 ทำการแยกเปลือกข้าวออก เพื่อให้เหลือแต่เมล็ดข้าว หากเรามีกระถังสำหรับร่อนแยกสิ่งแปรปรวนป่นในขั้นที่ 1 อยู่เพียง 1 อัน มีครกกระเดื่องสำหรับตำให้เมล็ดข้าวแยกออกจากเปลือกเพียง 1 อัน และมีภาชนะสำหรับทำการแยกเมล็ดข้าวออกจากเปลือกข้าวเพียง 1 อัน หากมีคนทำงานเพียง 1 คน อาจจะต้องใช้เวลาในการทำงานทั้งหมด 3 ชั่วโมง แต่หากมีคนทำงาน 3 คน แบ่งกันทำงานในแต่ละขั้นตอน แล้วส่งต่อ อาจจะใช้เวลาในการทำงานเพียง 1 ชั่วโมงครึ่ง ทั้งนี้เนื่องจากหากมีคนทำงานเพียง 1 คน จะต้องแยกสิ่งแปรปรวนป่นให้เสร็จก่อน จากนั้นจึงจะทำการตำเพื่อให้เมล็ดข้าวออกจากเปลือกข้าว และจึงมาแยกทั้ง 2 ส่วนออกจากกัน แต่หากมีคนทำงาน 3 คน เมื่อคนแรกแยกสิ่งแปรปรวน แล้วส่งข้าวเปลือกที่แยกได้มาให้คนที่ 2 ตำนั้น คนแรกก็สามารถที่จะแยกข้าวเปลือกได้ต่อไป พร้อมกับคนที่ 2 กำลังตำข้างเปลือก และเมื่อคนที่ 2 ตำข้าวเปลือกเสร็จแล้วส่งให้คนที่ 3 แยกเมล็ดข้าวและเปลือกข้าว คนที่ 2 พร้อมที่จะรับข้าวเปลือกที่แยกสิ่งแปรปรวนแล้วจากคนแรกมาทำงานต่อได้ ส่งผลให้งานทุกขั้นตอนสามารถทำได้พร้อมกัน เป็นต้น

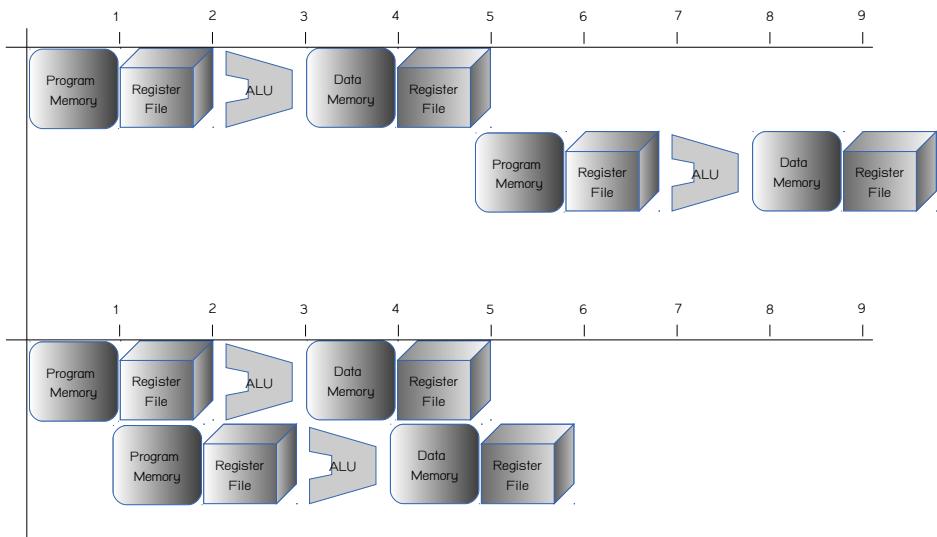
ในหน่วยประมวลผลกลางนั้น การเพิ่ม throughput โดยการทำ pipeline ก็อยู่ในลักษณะเดียวกัน คือ เมื่อคำสั่งแรกถูก fetch ขึ้น และถูกส่งไปทำการ decode ในช่วงเวลาหน่วยประมวลผลสามารถ fetch คำสั่งถัดไปขึ้นมา และเมื่อคำสั่งแรกทำการ execute คำสั่งที่ 2 ก็อยู่ระหว่างการ decode และ หน่วย fetch ก็กำลังทำการ fetch คำสั่งที่ 3 ขึ้นมา ซึ่งส่งผลให้มีคำสั่งที่ถูกทำงานโดยหน่วยประมวลผลกลางมากกว่าหนึ่งคำสั่งในเวลาเดียวกัน ดังแสดงในรูปที่ 6.1²⁹

จากรูปจะพบว่า การทำงาน 2 คำสั่งกรณีไม่มี pipeline ใช้เวลาในการทำงานถึง 20 หน่วยเวลา ในขณะที่เมื่อมีการทำ pipeline จะใช้เวลาเพียง 12 หน่วยเวลาเท่านั้น อย่างไรก็ตาม จะสังเกตได้ว่า แต่ละคำสั่งไม่สามารถ fetch หรือถูกอ่านขึ้นมาพร้อมกันได้ ทั้งนี้เนื่องจากแต่ละหน่วยของหน่วยประมวลผลกลางสามารถให้บริการได้เพียงหนึ่งคำสั่งในเวลาหนึ่ง ๆ เท่านั้น

²⁹ เพื่อการอธิบายการทำงาน ภายใต้แผนภาพให้การแรงงานเพื่อบุกร่างกายที่มีภาระแรงทางด้านชั้นจะแทนการเขียนข้อมูล ส่วนภาพที่แรงงานด้านชั้นจะแทนการอ่านข้อมูล

จากรูป พบว่ากรณี pipeline มีการทำงานย่อยภายใน 5 ขั้นตอน ซึ่งหมายถึงเมื่อทำงานเต็มรูปแบบจะมีคำสั่งทำงานพร้อมกัน 5 คำสั่ง ลักษณะนี้ ทางสถาปัตยกรรมคอมพิวเตอร์จะเรียกว่า pipeline แบบ 5 ขั้น (5-stage pipeline)

เพื่อประกอบความเข้าใจ เนื้อหาในบทนี้จะอธิบายการทำงานและการออกแบบ pipeline เป็นต้น โดยเริ่มจากการเปรียบเทียบสมรรถนะที่น่าจะได้รับ ปัญหาที่เป็นอุปสรรคต่อการทำ pipeline และการปรับปรุงหน่วยประมวลผลกลางเพื่อให้รองรับการทำงานของ pipeline ทั้งนี้อยู่บนสมมติฐานของสถาปัตยกรรม nanoLADA ที่คำสั่งทุกคำสั่งมีความยาวเท่ากัน มีรูปแบบคำสั่งที่จำกัด (3 รูปแบบ) และมีคำสั่งที่เกี่ยวข้องกับหน่วยความจำเพียง load และ store เท่านั้น ทั้งนี้หากผู้อ่านต้องการทราบถึงสถาปัตยกรรมที่ซับซ้อน สามารถหาอ่านได้ตามเอกสารอ้างอิงของผู้ผลิตหน่วยประมวลผลกลางทั่วไป



รูปที่ 6.1: เปรียบเทียบเวลาการทำงานแบบมี pipeline และ ไม่มี pipeline

เนื้อหาในบทนี้จะเริ่มต้นด้วยการกล่าวถึงสมรรถนะของ pipeline ในอุดมคติ จากนั้นถึงจะอธิบายถึงปัญหาและอุปสรรคต่าง ๆ และแนวทางการแก้ปัญหา

6.1 สมรรถนะของ Pipeline กรณีอุดมคติ

อ้างอิงถึงการวิเคราะห์สมรรถนะของหน่วยประมวลผลกลางแบบ multiple cycle ในตอนที่

5.1 พบร่วาหน่วยประมวลผลกลางแบบ multiple cycle มีแนวโน้มที่มีสมรรถนะดีกว่าหน่วยประมวลผลกลางแบบ single cycle โดยการพันแปรค่าของจำนวนรอบการประมวลผลต่อคำสั่งเทียบกับรอบสัญญาณนาฬิกาที่สั้นลง

ตัวอย่างที่ 6.1: การเปรียบเทียบสมรรถนะ Pipeline กรณีอุดมคติ

จาก pipeline แบบ 5 สถานะที่กำหนดในรูปที่ 6.1 หากมีจำนวนคำสั่งมาก (ถือว่าเป็นอนันต์ก็ได้) และสมมุติให้การทำงานทุกคำสั่งมี 5 รอบเท่ากัน การทำงานแบบ pipeline แบบอุดมคติ (ต่อเนื่อง ไม่มีการหยุด) และการทำงานแบบ multiple cycle (ไม่มี pipeline) จะให้สมรรถนะต่างกันกี่เท่า

การทำงานแบบ multiple cycle เมื่อมีจำนวนคำสั่งเป็น k จะได้ว่า

$$CPU\ TIME_{multiple\ cycle} = k \times 5 \times \text{cycle time}$$

การทำงานแบบ pipeline เมื่อมีจำนวนคำสั่งเป็น k จะได้ว่า

$$CPU\ TIME_{pipeline} = (5 + (k - 1) \times 1) \times \text{cycle time}$$

หากต้องการเปรียบเทียบสมรรถนะว่า แบบ pipeline เเร็วกว่าแบบ multiple cycle กี่เท่า จะได้ว่า

$$n = \frac{CPU\ TIME_{multiple\ cycle}}{CPU\ TIME_{pipeline}} = \frac{k \times 5 \times \text{cycle time}}{(5 + (k - 1) \times 1) \times \text{cycle time}}$$

$$n = \frac{k \times 5}{5 + (k - 1) \times 1} = \frac{5k}{5 + k - 1} = \frac{5k}{4 + k}$$

กรณีมีปริมาณคำสั่ง (k) มากจนเป็นอนันต์ จะได้ค่า n ดังนี้

$$n = \lim_{k \rightarrow \infty} \frac{5k}{5 + k - 1} = 5$$

∴ สมรรถนะของหน่วยประมวลผลแบบ pipeline จะเร็วกว่า 5 เท่า (หรือจำนวนขั้นตอนของ pipeline)

สำหรับการทำ pipeline เวลาที่ใช้ในการทำงานแต่ละคำสั่งนั้นมีได้สั้นลง หากแต่เวลาโดย

รวมของการทำงานจะสั้นลง (เป็นการปรับปรุงสมรรถนะโดยการเพิ่ม throughput ดังที่ได้กล่าวไว้แล้ว) เพื่อประกอบความเข้าใจของพิจารณารูปที่ 6.1 และ ตัวอย่างที่ 6.1

จากตัวอย่าง มีข้อสังเกตคือ เมื่อ pipeline เริ่มต้นจนคำสั่งเต็มแล้ว คำสั่งที่เหลือจะใช้เวลาเพียง 1 รอบ เพื่อให้ประมวลผลเสร็จ จำนวนรอบที่ใช้ในการทำให้ pipeline เต็มจะเท่ากับจำนวนขั้นของ pipeline (stage of pipeline) (เพื่อความเข้าใจ ขอให้ลองวางแผนภาพการทำงานของ pipeline เพิ่มเติมในแบบฝึกหัดท้ายบท) ดังนั้นสมการทั่วไปสำหรับแสดงเวลาที่ใช้ในการประมวลผลแบบ pipeline จะเป็นสมการที่ 6.1

$$CPU\ TIME_{pipeline} = [stage_{pipeline} + (count_{instructions} \times CPI)] \times T_{cycle}$$

สมการที่ 6.1 CPU Time (pipeline)

ทั้งนี้ ขอให้เนื้อก oy ย่ เสนอว่า CPU Time ดังกล่าวคิดกรณี ideal กล่าวคือ pipeline สามารถทำงานได้ต่อเนื่องไม่มีการสะดุด (จะกล่าวถึงการสะดุดของ pipeline ต่อไปในหัวข้อ 6.2)

6.2 ปัญหาที่เป็นอุปสรรคต่อการทำ Pipeline

หากวิเคราะห์การทำงานในลักษณะของ pipeline ให้ดีจะพบว่าการทำ pipeline นั้น จะกระทำได้ก็ต่อเมื่องานนั้นแต่ละขั้นตอนแยกเป็นอิสระจากกันได้ ดังนั้นในการออกแบบหน่วยประมวลผลกลางให้สามารถทำงานแบบ pipeline ได้ ปัญหาอย่างแรกคือ การทำให้งานในแต่ละขั้นตอนเป็นอิสระจากกัน เพื่อประกอบความเข้าใจจะอธิบายเปรียบเทียบปัญหาการเป็นอิสระจากกันในแต่ละขั้นตอนการทำงาน pipeline ด้วยการเตรียมข้าวสาร(ตำข้าว)ในทำนองเดียวกับที่ได้กล่าวนำก่อนเข้าสู่บทเรียน ดังนี้

ในการเตรียมข้าวสารซึ่งมี 3 ขั้นตอนได้แก่ (1) การร่อนข้าว (2) การตำข้าว (3) การแยกเปลือกข้าว หากขั้นตอนที่ 1 และขั้นตอนที่ 3 ต้องใช้เครื่องมือ(กระดัง)เหมือนกัน แต่มีกระดังเพียง 1 อันเท่านั้นที่ใช้งานได้ เช่นนี้แล้ว ก็คงไม่สามารถที่จะแบ่งการทำงานในลักษณะของ pipeline ได้ (ปัญหานี้ในลักษณะดังกล่าวเรียกว่า structural hazard ดังจะกล่าวถึงต่อไป) นอกจากนี้ยังมีปัญหาลักษณะอื่นอีก เช่น ข้าวบางประเภทอาจต้องตำและแยกเปลือกข้าวหลายรอบ ดังนั้น ถึงแม้ว่าขั้นตอนที่ 2 จะว่างอยู่ ก็อาจไม่สามารถเริ่มตำข้าวชุดใหม่ได้ เพราะต้องรอตำข้าวชุดเดิมให้เสร็จก่อน เป็นต้น

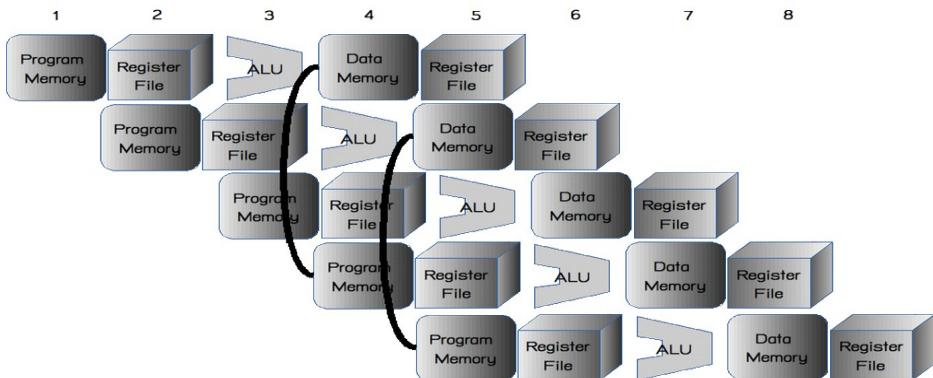
ปัญหาหลักที่เป็นอุปสรรคต่อสมรรถนะและการทำงานของ pipeline มีทั้งสิ้น 3 ประเด็นคือ ปัญหากรณีโครงสร้างของระบบหรือ resource ที่ต้องใช้งานร่วมกัน (structural

hazard) ปัญหาของการประมวลผลข้อมูลซึ่งมีความเกี่ยวเนื่องกัน (data hazard) และปัญหาการหาคำสั่งถัดไปของระบบในกรณีการ jump หรือ branch (control hazard) ซึ่งในที่นี้จะกล่าวถึงปัญหาแต่ละอันและการแก้ปัญหาเบื้องต้น โดยใช้วิธีการพื้นฐาน ดังรายละเอียดต่อไปนี้

6.2.1 Structural Hazard

structural hazard เป็นปัญหาที่เกิดจากโครงสร้างของระบบไม่รองรับการทำ pipeline เช่น กรณีของหน่วยความจำซึ่งทำหน้าที่เป็นทั้ง instruction memory และ data memory พร้อมกัน หากเราพิจารณารูปที่ 6.2 จะพบว่าที่รอบที่ 4 นั้น หน่วยความจำจะถูกใช้งานเป็น instruction memory เพื่อการ fetch และเป็น data memory เพื่อการอ่านข้อมูลพร้อมกัน ซึ่งปัญหานี้อาจแก้ได้โดยการเพิ่มจำนวนขององค์ประกอบอนั้น ๆ ให้สามารถให้บริการพร้อมกันได้ ในกรณีนี้การเพิ่มจำนวนขององค์ประกอบอาจทำได้โดยการแยก instruction memory และ data memory ให้ต่อ กับ bus ที่เป็นอิสระจากกัน หรือ แยก cache ของ instruction memory กับ data memory ออกจากกัน (ดังจะกล่าวต่อไปในบทที่ 8)

ประเด็นช่วยจำ หากปริยบเทียบ structural hazard กับตัวอย่างการดำเนินการต่อข้าวคือ กรณีที่กระดังสำหรับการร่อนข้าว (ขั้นตอนที่ 1) และ สำหรับการแยกข้าวเปลือก (ขั้นตอนที่ 3) เป็นชุดเดียวกัน จึงไม่สามารถดำเนินการพร้อมกันได้ ดังนั้นวิธีการแก้ปัญหา hazard ลักษณะนี้ที่ตรงไปตรงมาที่สุดคือ การเพิ่มจำนวนของกระดังอีก 1 ชุด

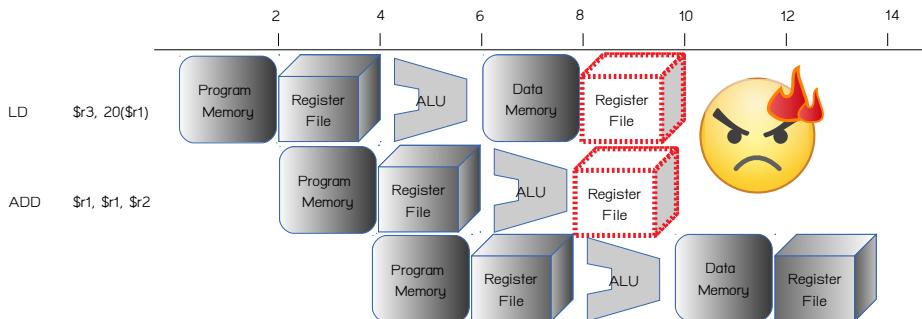


รูปที่ 6.2: ปัญหา Structural Hazard

อีกกรณีหนึ่งของ structural hazard ใน nanoLADA คือ กรณีที่หน่วยประมวลผลกลาง

ต้องใช้ ALU (adder) เพื่อบอก PC ในการทำ instruction fetch และ execute คำสั่งพร้อมกัน หากมี ALU เพียง 1 ชุด จะทำให้ไม่สามารถทำ pipeline ได้ ในทำงานองเดียวกัน วิธีแก้ไขคือการเพิ่มจำนวนทรัพยากร กรณีนี้คือเพิ่ม ALU (adder) เป็น 2 ชุด ให้ทำงานแยกเป็นอิสระจากกัน

นอกจากนี้ ยังมีปัญหา structural hazard อีกแบบหนึ่งซึ่งเกิดจากกรณีที่เวลาในการทำงานของแต่ละคำสั่งไม่เท่ากัน ลองดูตัวอย่างรูปที่ 6.3 จะเห็นว่าคำสั่ง ADD ไม่จำเป็นจะต้องทำงานในขั้นที่ 4 คือ memory access ดังนั้น จึงสามารถข้ามไปยังขั้นตอนที่ 5 คือ write back ได้เลย ผลที่ได้คือ ขั้นตอนการ write back ของคำสั่ง ADD จะตรงกับขั้นตอน write back ของคำสั่ง LD ซึ่งมาก่อนหน้าพอดี ทำให้ไม่สามารถทำงานได้ เพราะต้องใช้ resource พร้อมกัน



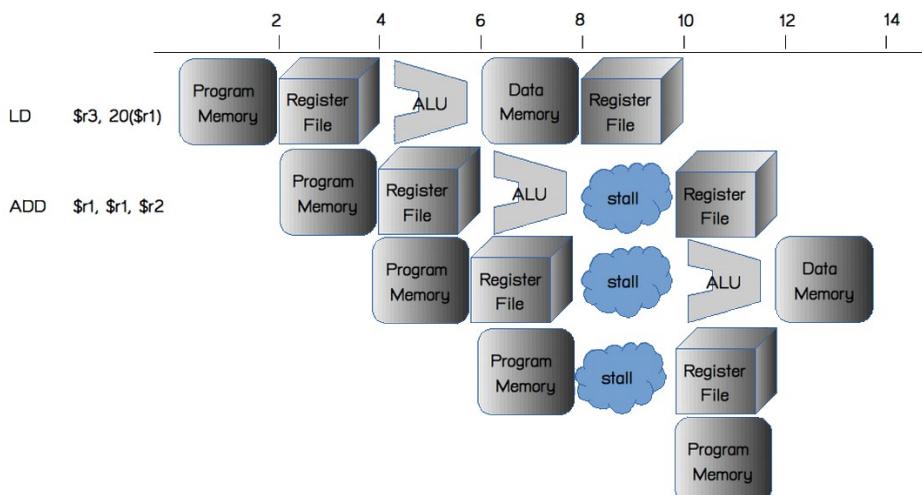
รูปที่ 6.3: ปัญหา structural hazard จากการที่แต่ละคำสั่งใช้จำนวน cycle ไม่เท่ากัน

เนื่องจากกรณีนี้ ไม่สามารถที่จะเพิ่ม instance ของ resource ได้ กล่าวคือไม่สามารถเพิ่มชุดเรจิสเตอร์เข้าไปใหม่ได้ เพราะจะเป็นเรจิสเตอร์คนละชุดที่ไม่เกี่ยวข้องกัน ดังนั้น จึงเหลือแนวทางในการแก้ปัญหาเพียงสองแนวทาง คือ การ stall (หยุดการทำงานของ pipeline ด้านล่างที่ตามมา) และ การปรับขั้นตอนการทำงานของคำสั่ง R-type (ในที่นี้คือคำสั่ง ADD) เพื่อหลีกเลี่ยงปัญหาการเกิด structural hazard

สำหรับการ stall ข้อเสียที่เห็นชัดเจนคือ pipeline ด้านล่างจะต้องหยุดทั้งหมด นั่นหมายความว่า การทำงานของ ALU, การอ่านชุดเรจิสเตอร์ และ การ fetch ข้อมูลจาก program memory จะต้องหยุดชะงักไปด้วย ดังแสดงในรูปที่ 6.4 ซึ่งหากมีการ stall ในลักษณะนี้มาก ย่อมส่งผลกระทบต่อสมรรถนะโดยรวมของ pipeline แน่นอน

ดังนั้น การปรับการทำงานคำสั่ง R-type (คำสั่ง ADD ในที่นี้) ให้เลี่ยงการเกิด structural

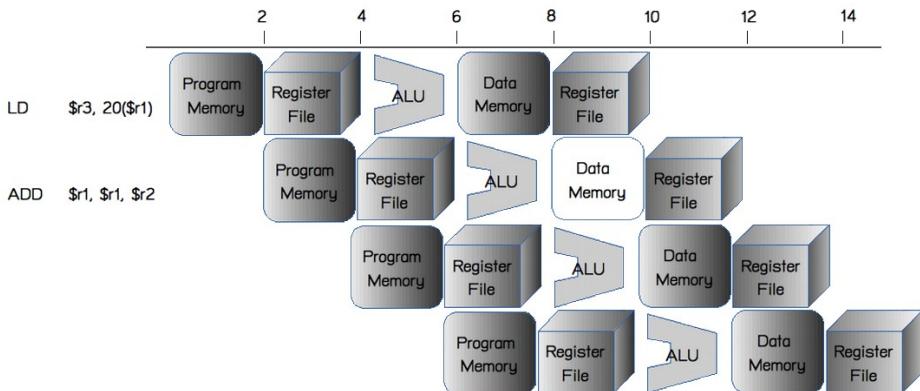
hazard ได้ จะเป็นทางออกที่ดีกว่า ในกรณีของสถาปัตยกรรม nanoLADA นี้ สามารถทำได้ง่ายโดยการเพิ่มขั้นตอน memory access ให้กับคำสั่ง R-type ทั้งหมด ข้อเสียของวิธีการนี้คือ การทำงานของคำสั่ง R-type (เช่น คำสั่ง ADD) จะเสร็จช้าลงไป 1 รอบแต่หากวิเคราะห์ภาพรวมแล้ว อาจจะมีได้ส่งผลเสียได้ต่อสมรรถนะโดยรวมของระบบ เพราะผลที่ได้คือ ไม่เกิดการ stall (ไม่มีคำสั่งใดต้องหยุดการทำงาน) และถึงแม้คำสั่งบางคำสั่งจะเสร็จช้าไป 1 รอบแต่ ทุกรอบก็ยังมีคำสั่งทำงานเสร็จอยู่เหมือนเดิม จึงมีได้ทำให้สมรรถนะแตกต่างไปแต่อย่างใด



รูปที่ 6.4: การแก้ปัญหา Structural Hazard ด้วยการ stall

เพื่อประกอบความเข้าใจ ลองวิเคราะห์รูปที่ 6.5 จากรูปจะเห็นว่าขั้นตอน memory access ของคำสั่ง ADD จะเป็นสิ่วๆ ซึ่งเป็นการเจ้าไม่มีการอ่านและไม่มีการเขียนข้อมูลเกิดขึ้นแต่อย่างใด ซึ่งการทำงานในระดับ RTL จะเป็นการส่งผ่านค่าที่ได้จาก ALU ต่อไป เพื่อรับการ write back ใน clock ถัดไป

ข้อชวนคิด อ้างอิงจากรูปที่ 5.2 ซึ่งแสดงขั้นตอนการทำงานของแต่ละคำสั่ง จะพบว่าวนอกจากคำสั่ง R-type ซึ่งมีการทำงาน 4 รอบแล้ว ยังมีคำสั่ง SW, BEQ และ JMP ซึ่งมีการทำงานเป็น 4, 4 และ 3 รอบตามลำดับ เนื่องจากการทำงานของคำสั่งดังกล่าว จึงไม่ส่งผลให้เกิดปัญหา structural hazard (ขอให้เป็นแบบฝึกหัดท้ายบทสำหรับผู้เรียนได้自行คิดต่อไป)



รูปที่ 6.5: การแก้ structural hazard โดยการปรับขั้นตอนการทำงานของคำสั่ง R-type

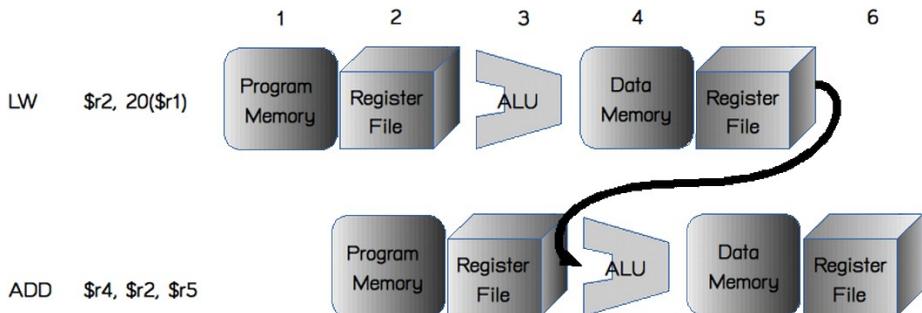
6.3 Data Hazard

data hazard นั้น เป็นปัญหาที่เกิดจากการประมวลผลคำสั่งที่มีความพึ่งพิงกัน (dependency) โดยหากวิเคราะห์จากรูปที่ 6.6 พบร่วมกับการทำงานของคำสั่ง LW นั้น ข้อมูลที่ได้จากหน่วยความจำเลขที่อยู่ $20 + \$r1$ จะถูกนำมายังใน rejister \$r2 ในรอบที่ 5 (หากผู้อ่านไม่ทราบว่าแต่ละรอบของแต่ละคำสั่งมีการทำงานอย่างไร ให้ลองกลับไปพิจารณาขั้นตอนการทำงานในบทที่ 5 อีกครั้งหนึ่ง) ทั้งนี้เนื่องจากเป็นการทำงานที่เกี่ยวข้อง กับหน่วยความจำ ซึ่งจะได้ข้อมูลออกจากหน่วยความจำในตอนจบของรอบที่ 4 ในขณะที่คำ สั่งถัดมาต้องการใช้งาน rejister \$r2 และ rejister \$r5 ในรอบที่ 2 ทั้งนี้ข้อมูล rejister เออร์ \$r2 นั้นจะได้มาจากการทำงานของคำสั่งแรกซึ่งจะเสร็จสิ้นในรอบที่ 5 ซึ่งระบบไม่ สามารถนำข้อมูลที่จะได้รับในอนาคตมาประมวลผลได้ ดังแสดงด้วยเส้นทึบในรูป ดังนั้นคำ สั่งถัดมาจึงไม่สามารถเริ่มทำงานได้ เพราะข้อมูลที่จำเป็นสำหรับการเริ่มทำงานยังไม่พร้อม ให้ใช้งานกว่าจะจบการทำงานในรอบที่ 5

เพื่อให้เห็นภาพ ขอเปรียบเทียบกับตัวอย่างโปรแกรมเบื้องต้นดังนี้

```
int a, b;
a = 7;
b = a + 5;
```

จากตัวอย่างโปรแกรมนี้ หากการทำงาน $a = 7$ เสร็จสิ้นในรอบที่ 5 แต่ เราต้องการทำ $b = a + 5$ ในรอบที่ 3 (ขั้นตอนการอ่านค่าจาก rejister) หมายความว่า หากมีการบวกเกิดขึ้น ผลลัพธ์ที่ได้จะผิด เพราะค่า a ที่ถูกนำไปใช้เป็นค่าที่ผิด (hazard)



รูปที่ 6.6: ปัญหา data hazard

จากรูปที่ 6.6 หากเรจิสเตอร์เดียวกันสามารถถูกอ่านและเขียนได้พร้อมกันการประมวลผล จะถูกต้องเมื่อการทำงานของคำสั่ง ADD ข้ามอิก 2 รอบ แต่หากจำเป็นจะต้องมีการเขียนค่าลงเรจิสเตอร์ให้เสร็จสิ้นก่อนจะอ่านค่าใหม่ออกมาได้ จะต้องทำให้คำสั่ง ADD เริ่มข้ามอิก 3 รอบ

ดังนั้นแนวทางในการแก้ปัญหาของ data hazard คือ การแก้ปัญหาด้วยวิธีการทางซอฟต์แวร์โดยการแทรกคำสั่งอื่น ๆ ที่ไม่มีการใช้ชื่อมูลที่เกี่ยวข้องกันเข้าไประหว่างคำสั่งที่ 1 และ 2 ทั้งนี้ต้องไม่ทำให้การทำงานของโปรแกรมผิดพลาดไปจากเดิม การแก้ปัญหา hazard ด้วยวิธีการทางซอฟต์แวร์ในลักษณะนี้เรียกว่า rescheduling

ข้อเสียของการแก้ปัญหา hazard ด้วยวิธีการทางซอฟต์แวร์คือ ทุกครั้งที่มีสถาปัตยกรรมรุ่นใหม่ออกมา สถาปัตยกรรมชุดคำสั่งอาจจะเปลี่ยนไปด้วย ผลคือต้องแปลงซอฟต์แวร์ใหม่เพื่อแก้ปัญหา hazard อีกครั้ง ดังนั้นหากสามารถปรับการแก้ปัญหา hazard ให้จัดการด้วยวิธีการทางhardwareได้ จะช่วยให้สถาปัตยกรรมมีความเข้ากันได้ระดับภาษาเครื่อง (binary compatibility) ซึ่งโปรแกรมเดิมจะทำงานบนสถาปัตยกรรมใหม่ได้โดยไม่ต้องมีการแก้ไข

ในหัวข้ออยู่ จะเป็นการอธิบายวิธีการแก้ปัญหา data hazard ด้วยแนวทางทางทางซอฟต์แวร์ และ hardware ดังนี้

6.3.1 การแก้ปัญหา Data Hazard ด้วยวิธีการทางซอฟต์แวร์

การแก้ปัญหา data hazard ด้วยวิธีการทางซอฟต์แวร์คือ การทำให้ซอฟต์แวร์รับรองว่า จะไม่เกิดการอ้างอิงชื่อมูลในลักษณะที่ติดกันเกินไปจนทำให้เกิดปัญหา เนื่องจากในปัจจุบันซอฟต์แวร์มักพัฒนาด้วยภาษาระดับสูง ด้วยเหตุนี้คอมไไฟเลอร์จึงได้มีการพัฒนาให้สามารถ

ทำการหาค่าที่เหมาะสมที่สุดด้วยวิธีการ code motion³⁰ เพื่อเรียงคำสั่งให้เหมาะสมกับการทำงานของหน่วยประมวลผลกลางที่มี pipeline

ในกรณีของรูปที่ 6.6 เนื่องจากโปรแกรมที่ให้มามีเพียงสองคำสั่งจึงไม่สามารถ reschedule ด้วยการเลือกคำสั่งที่อยู่ก่อนหน้าหรือตามหลังมาแทรกแทนได้ แต่หากจำเป็นต้องแทรกคำสั่งเพื่อให้ไม่เกิด hazard ตัวเลือกของคำสั่งที่เหมาะสมที่สุดคือ NOP (No Operation) ซึ่งมีความหมายว่า ไม่มีการประมวลผลใด ๆ หรือ ให้อยู่เฉย ๆ ดูตัวอย่างที่ 6.2 ประกอบคำอธิบาย

ตัวอย่างที่ 6.2: การแทรกคำสั่ง NOP เพื่อแก้ปัญหา data hazard

จากตัวอย่างการทำงานของ pipeline ซึ่งมีปัญหา data hazard ในรูปที่ 6.6 จะแทรกคำสั่ง NOP (มีการทำงาน 1 cycle) เพื่อให้หน่วยประมวลผลกลางสามารถทำงานได้โดยไม่มีปัญหา hazard

จากตัวอย่าง code ที่กำหนดให้มีเพียงสองคำสั่ง คือ

LW	\$r2, 20(\$r1)
AND	\$r4, \$r2, \$r5

หากไม่ต้องการให้เกิด hazard ต้องแก้ไขให้ขั้นตอนการอ่าน \$r2 อยู่ที่ cycle ที่ 6 (สมมุติว่าจะต้องเขียนค่าเรจิสเตอร์ใหม่ให้เสร็จก่อน จึงจะอ่านค่าที่ถูกต้อง ดังนั้น จะต้องดันให้ AND ข้ามอกไปอีก 3 cycle ดังนี้

LW	\$r2, 20(\$r1)
NOP	
NOP	
NOP	
AND	\$r4, \$r2, \$r5

อย่างไรก็ตามข้อเสียของการแทรกคำสั่ง NOP เข้าไปนั้น คือ ทำให้โปรแกรมมีขนาดใหญ่ขึ้น และเสื่อมประสิทธิภาพในการทำงานมากขึ้น ดังนั้นแนวทางที่ดีกว่าคือ การจัดลำดับของคำสั่งใหม่ โดยยังให้ผลลัพธ์จากการทำงานถูกต้องเหมือนเดิม

³⁰ เทคนิคเรื่อง code motion เดยกล่าวถึงแล้วในหัวข้อที่ 3.6.1 อย่างไรก็ตามหน่วยสถาปัตยกรรมชุดคำสั่งบางแบบ บังคับว่าคอมpileอร์จะต้องทำ code motion เพื่อแก้ปัญหา hazard เสมอ ทั้งนี้อุปสรรคของการทำ code motion คือมักติดอยู่ใน basic block เดียวกัน มักไม่สามารถทำงานข้าม block ได้

การทำ reschedule (ดังตัวอย่างที่ 6.3) จะช่วยให้สมรรถนะที่ได้ไม่ลดลง (เพราะไม่มีการแทรกคำสั่งเพิ่ม เป็นเพียงการเรียงลำดับคำสั่งใหม่) โดยต้องระวังไม่สลับลำดับจนทำให้ผลลัพธ์ที่ได้ผิดไป

ตัวอย่างที่ 6.3: การทำ rescheduling เพื่อให้ไม่เกิดปัญหา data hazard

จงเรียงลำดับโปรแกรมข้างล่างนี้โดยไม่มีการแทรกคำสั่งอื่น (เช่น NOP) เพิ่มเติม เพื่อให้ไม่มีปัญหา data hazard และมีผลลัพธ์จากการทำงานเหมือนเดิม

- 1) LW \$r2,20(\$r1)
- 2) AND \$r4,\$r2,\$r5
- 3) OR \$r5,\$r5,\$r1
- 4) ADD \$r6,\$r6,\$r1
- 5) XOR \$r9,\$r9,\$r9
- 6) ADD \$r1,\$r1,\$r3
- 7) LW \$r3,40(\$r6)

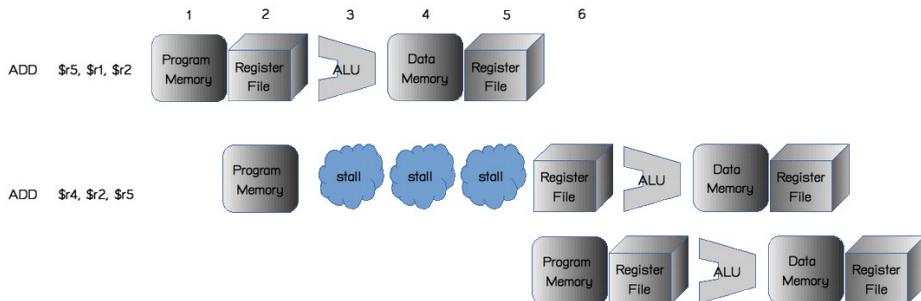
จาก code ที่กำหนดให้ หากวิเคราะห์ลำดับการทำงานจะพบว่า 1 ต้องทำก่อน 2, 2 ต้องทำก่อน 3, 3 และ 4 ต้องทำก่อน 6, และ 4 และ 6 ต้องทำก่อน 7 เป้าหมายที่ต้องการคือ ทำให้ 1 ทำเสร็จก่อน 2 จะทำการอ่านค่าจาก寄存器 ดังนั้นอาจเรียงลำดับคำสั่งใหม่ได้โดยนำ 4, 5 และ 7 ขึ้นมาแทรกก่อน 2 ได้เป็น

- 1) LW \$r2,20(\$r1)
- 4) ADD \$r6,\$r6,\$r1
- 5) XOR \$r9,\$r9,\$r9
- 2) AND \$r4,\$r2,\$r5
- 3) OR \$r5,\$r5,\$r1
- 7) LW \$r3,40(\$r6)
- 6) ADD \$r1,\$r1,\$r3

6.3.2 การแก้ปัญหา Data Hazard ด้วยวิธีการ Hardware Forward

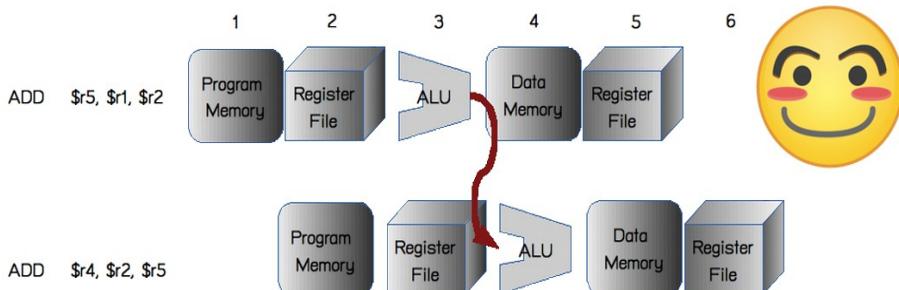
แนวทางการแก้ปัญหา data hazard อีกทางหนึ่งคือ การทำ hardware ให้สามารถแก้ปัญหาข้อมูลที่มีความเกี่ยวเนื่อง เพื่อจะได้ไม่ต้องแก้ไขโปรแกรม ข้อดีคือซอฟต์แวร์ซึ่งแต่เดิมเคยทำงานได้ถูกต้องบนหน่วยประมวลผลกลางที่ไม่มี pipeline สามารถนำประมวลผลบนหน่วยประมวลผลรุ่นใหม่ที่มี pipeline ได้ทันทีโดยไม่ต้องแก้ไขหรือแปลงใหม่แต่อย่างใด

การทำฮาร์ดแวร์เพื่อแก้ปัญหา data hazard วิธีการหนึ่งคือ การออกแบบฮาร์ดแวร์ให้ทำการ stall หรือหยุดการทำงานของคำสั่งก่อนหน้าที่จำเป็นต้องรอการประมวลผลให้เสร็จก่อนทำการประมวลผลต่อ (แสดงในรูปที่ 6.7) ซึ่งการแก้ปัญหานี้จะเกี่ยวกับการแทรกคำสั่ง NOP เข้าไปเพื่อถ่วงเวลาเริ่มต้นการทำงานของคำสั่งถัดมา ในทำนองเดียวกันข้อเสียของการทำ stall คือเวลาที่เสียไปอย่างไม่เกิดประโยชน์ของการประมวลผล



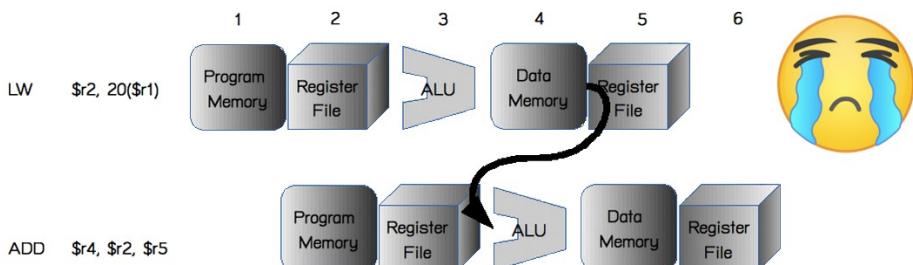
รูปที่ 6.7: การแก้ปัญหา data hazard ด้วย hardware stall

เพื่อไม่ให้เสียเวลาไปโดยเปล่าประโยชน์ อีกแนวทางหนึ่งในการแก้ปัญหา data hazard คือการทำ hardware forward³¹ หลักการของการ forward คือการดึงค่าที่ประมวลผลเสร็จออกมาใช้ก่อน แม้ค่านั้นจะยังไม่ถูกเขียนกลับไปรีบตาม เช่น กรณีคำสั่ง ADD \$r5, \$r1, \$r2 ตามด้วยคำสั่ง ADD \$r4, \$r2, \$r5 ซึ่งข้อมูลของ \$r5 ที่ประมวลผลเสร็จแล้วนั้น จะถูกนำไปเก็บที่ rejister ในรอบที่ 5 แต่หากวิเคราะห์ให้ดี จะเห็นว่าการประมวลผลทำเสร็จตั้งแต่รอบที่ 3 นั้นหมายความว่า ฮาร์ดแวร์สามารถที่จะส่งต่อค่าที่ได้จากการประมวลผลออกมายังรอบที่ 4 โดยไม่จำเป็นต้องรอให้ข้อมูลถูกนำมาเก็บใน rejister \$r5 ก่อน ซึ่งฮาร์ดแวร์ที่ทำงานดังกล่าว (เรียกว่า forward unit) จะต้องมี multiplexor สำหรับการสลับค่านำเสนอผลลัพธ์จากค่าในคำสั่งก่อนกลับมาป้อนให้กับ ALU อีกครั้งหนึ่ง ดังแสดงด้วยเส้นที่บีบในรูปที่ 6.8 จากรูปด้วย hardware forward หน่วยประมวลผลทางก็ไม่จำเป็นต้องเสียเวลารอเพื่อนำค่าของ rejister \$r5 ไปใช้หลังรอบที่ 5 แต่อย่างไร



รูปที่ 6.8: ตัวอย่างการแก้ปัญหา Data Hazard ด้วย hardware forward

อย่างไรก็ตามมีหลายกรณีที่หน่วยประมวลผลกลางไม่สามารถทำ hardware forward ได้ ตัวอย่างที่ชัดเจนที่สุด เช่น กรณีข้อมูลที่ต้องการใช้ได้มาในรอบที่ 4 จากคำสั่ง LW (แต่ต่างจากคำสั่งที่มา晚ไปที่มักจะให้ผลลัพธ์ในรอบที่ 3) ซึ่งกรณีคำสั่ง LW (ครุภูที่ 6.9) จะต้องอ่านค่าจากหน่วยความจำเข้ามาเก็บในเรจิสเตอร์ นั่นหมายความว่า หากจะต้องทำ hardware forward จะต้อง forward จากรอบที่ 4 ไปยังรอบที่ 5 แทน (เราไม่สามารถ forward จากปลายรอบที่ 4 มายังต้นรอบที่ 4 ได้ ยกเว้นมีเครื่องย้อนเวลา)

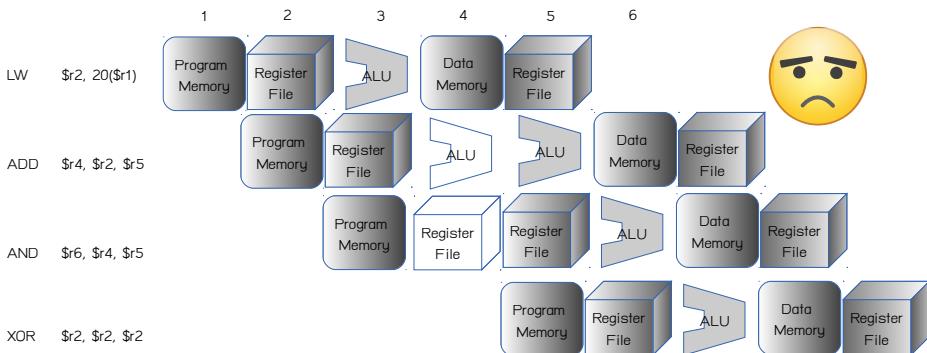


รูปที่ 6.9: ตัวอย่างกรณีที่ไม่สามารถ forward ได้

เนื่องจากไม่สามารถทำการแก้ปัญหา hazard ได้ด้วยวิธีการอื่น จึงเหลือเพียงวิธีการเดียวคือ การแก้ปัญหา hazard ด้วยการหยุดรอ หรือ stall รูปที่ 6.10 แสดงการทำ hardware stall³² โดยจะต้องมีการแทรกการการทำงานเข้าขั้นตอนเดิมในคำสั่งที่ตามมากรณีที่ไม่สามารถ forward ได้ แม้วินิจฉัยแก้ปัญหาได้แต่ในแง่ของสมรรถนะก็ยังถือว่าไม่ดี เพราะมีการทำงานที่ไม่ก่อให้เกิดประโยชน์เกิดขึ้น

32 ในทางปฏิบัติแนวทางหนึ่งในการทำ hardware stall คือ การไม่เปลี่ยนสถานะการทำงาน หรือการให้ทำ stage เดิม ข้ามอีก 1 cycle

หากวิเคราะห์ต่อจะพบว่า แม้จะมี hardware forward เพื่อช่วยแก้ปัญหา data hazard แต่ การจัดลำดับของคำสั่งที่เหมาะสมก็ยังคงช่วยให้ไม่เกิดการ stall ของคำสั่งภายในระบบได้ เช่นกัน ดังนั้นการป้องกันการเกิด data hazard ที่ดีที่สุดยังคงเป็นการปรับการทำงานของ ซอฟต์แวร์ในขั้นตอนของคอมไฟเลอร์ หรือการทำงานร่วมกันระหว่างฮาร์ดแวร์และ ซอฟต์แวร์อยู่นั่นเอง

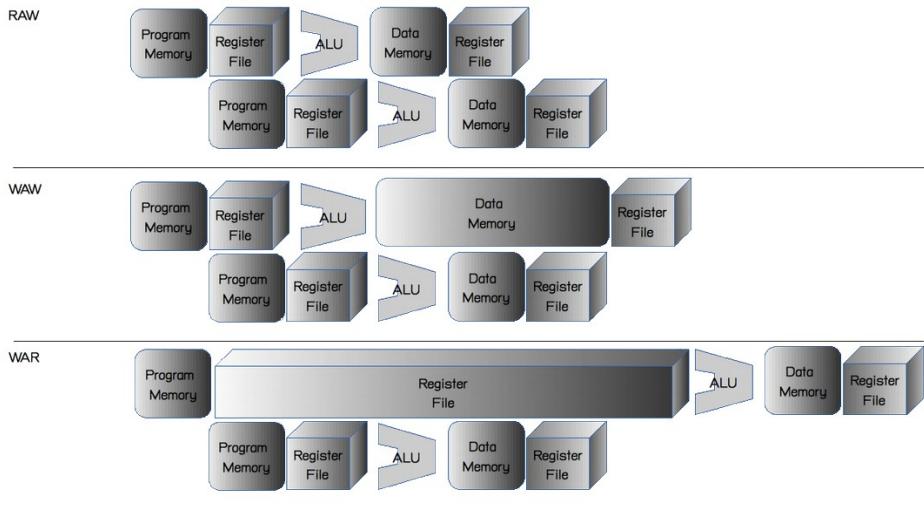


รูปที่ 6.10: การทำ hardware stall กรณีที่ไม่สามารถทำ hardware forward ได้

6.3.3 Data Hazard แบบ RAW, WAW, WAR

เพื่อประกอบความเข้าใจภาพรวมของ data hazard ทั้งหมด จะขอทำการวิเคราะห์ปัญหา ต่อไปให้ลักษณะมากขึ้น (ซึ่ง hazard บางแบบเหล่านี้ จะไม่พบในสถาปัตยกรรม nanoLADA เนื่องจากไม่มีคำสั่งที่มีการทำงานซับซ้อนจนทำให้เกิด hazard ในลักษณะดังกล่าว) โดยจะจำแนกปัญหา data hazard ออกเป็น 3 กลุ่ม คือ Read-after-Write hazard (RAW), Write-after-Write hazard, (WAW) และ Write-after-Read hazard (WAR) ดังนี้ (ดูรูปที่ 6.11 ประกอบการอธิบาย)

- Read-after-Write hazard เกิดจากค่าที่ต้องการใช้ ยังไม่พร้อมให้ใช้งาน (เช่น การใช้ค่าที่เป็นผลลัพธ์จากการคำนวณในคำสั่งก่อนหน้า) ซึ่ง hazard ลักษณะนี้ อาจแก้ได้ด้วยการ forward (รูปที่ 6.8) ดังตัวอย่างที่เคยอธิบายก่อนหน้านี้



รูปที่ 6.11: data hazard แบบ RAW, WAW และ WAR

- Write-after-Write hazard เกิดจากการที่การประมวลผลคำสั่งก่อนหน้า ทำงานเสร็จช้ากว่าคำสั่งที่ตามมา เพื่อให้เห็นภาพ ลองพิจารณา code ต่อไปนี้

```
a = data[1000];
a = 10;
b = a + 3;
```

จะเห็นว่า มีความเป็นไปได้ที่ $a \leftarrow \text{data}[1000]$ ซึ่งต้องเป็นการอ่านค่าจากหน่วยความจำ อาจจะใช้เวลาในการประมวลผลนาน หากหน่วยความจำทำงานช้า จนส่งผลให้ $a \leftarrow 10$ ที่ทำงานได้รวดเร็วกว่าเสร็จก่อน หากเกิดเหตุการณ์เข่นนี้ขึ้น จะได้ค่า a ที่ผิดไป และต่อเนื่องให้ค่า b ผิดไปด้วย

- Write-after-Read hazard เป็นแบบที่พบได้น้อยในสถาปัตยกรรมสมัยใหม่ แต่มีพบได้ทั่วไปในสถาปัตยกรรมที่เป็น variable instruction length เพราะเวลาที่ใช้ในการ decode คำสั่งบางอัน อาจจะนานกว่าที่ควรจะเป็น ทำให้ค่า operand ที่อ่านได้ อาจจะเป็นค่าใหม่ที่ผิดเพี้ยนไปแล้ว เพื่อให้เห็นภาพมากขึ้น ลองพิจารณา code ต่อไปนี้

```
a = sin(b);
b = 10;
```

หากการเตรียมการเพื่อทำการคำนวน $\sin(b)$ ใช้เวลามากกว่าที่ควรคำสั่ง $b=10$ จะทำงานเสร็จก่อนที่การอ่านค่า b ในคำสั่งแรก ทำให้ผลการคำนวนของคำสั่ง $a = \sin(b)$ ผิดไป

ด้วยการออกแบบที่เหมาจะ เราสามารถแก้ไข Write-after-Write hazard และ Write-after-Read hazard ได้ไม่ยาก กล่าวคือ กรณีของ Write-after-Write hazard จะต้องยืนยันให้การคำนวนเสร็จตามลำดับเสมอ (in-order completion) ส่วนกรณีของ Write-after-Read hazard แก้ไขได้โดยการอ่านค่า operand ตั้งแต่ต้นรอบก่อน

จากลักษณะของ data hazard ดังกล่าว ในตำแหน่งสถาปัตยกรรมคอมพิวเตอร์สมัยใหม่ จึงมักมุ่งประเด็นของ data hazard ไปที่ read-after-write เป็นหลัก

ข้อชวนคิด อ้างอิงจากรูปที่ 6.11 จะเห็นว่าไม่มี data hazard แบบ Read-after-Read hazard อย่างทราบว่า เพราะเหตุใด

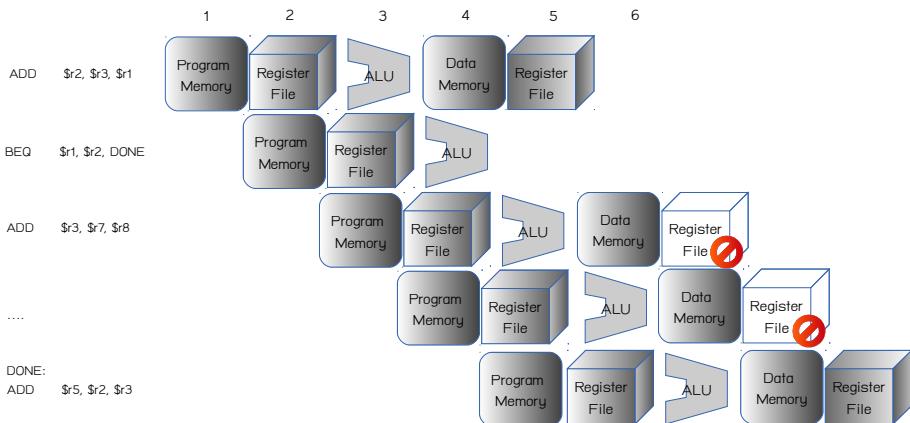
6.4 Control Hazard หรือ Branch Hazard

control hazard เป็นปัญหาในการทำ pipeline ที่เกิดจากการ jump หรือการ branch ของชอฟต์แวร์ เช่น กรณีการทำคำสั่ง BEQ จะทราบที่อยู่ของคำสั่งถัดไป (ทราบว่าจะต้อง branch หรือไม่) เมื่อได้ผ่านการประมวลผลคำสั่งนั้นไปแล้ว 3 รอบ แต่จากโครงสร้างของ pipeline ซึ่งจะต้องคำสั่งใหม่ในทุกรอบทำให้การดึงคำสั่งถัดไปขึ้นมาใน pipeline อาจเกิดปัญหาว่าเป็นคำสั่งที่ไม่ต้องการใช้งาน (เกิดการ branch)

การแก้ปัญหานี้ของ control hazard สามารถกระทำได้หลายแนวทาง เช่น การสร้างวงจรพิเศษเพื่อประมวลผลคำสั่งประเภท branch ให้เสร็จสิ้นภายใน 1 รอบเพื่อ pipeline จะได้เริ่มคำสั่งถัดไปที่ถูกต้องขึ้นมาทำงานได้ทันที ซึ่งกรณีนี้ทำให้ต้องออกแบบบางจุดที่ซับซ้อน ดังนั้นอีกทางเลือกหนึ่งคือ การไม่เขียนค่าคืนไปยัง rejistort (bypass) โดยไม่ทำขั้นตอน write back ซึ่งผลที่เกิดขึ้นจะเหมือนกับว่าคำสั่งนั้นจะไม่ถูกทำงาน รูปที่ 6.12 เป็นตัวอย่างการทำงานของ pipeline เมื่อมีการทำ bypass

อีกแนวทางในการลดผลกระทบของ control hazard ซึ่งทำได้ไม่ยากคือ การเดา (branch prediction) หลักการง่าย ๆ คือ ให้เดาว่าจะเกิดการ branch หรือไม่ และถ้าเดาถูก รอบของคำสั่งที่ทำไปก็ใช้ได้ (ไม่ต้องทำการ flush) ทั้งนี้การทำงานของ branch prediction

และ hardware ที่เกี่ยวข้องอยู่ nok เหนือขอบเขตของหนังสือเล่มนี้ แต่หลักการทำงานเป็นอย่างต้นคือ (1) เด่าว่า branch เสมอ (2) เดาว่าไม่ branch เสมอ หรือ ซับซ้อนยิ่งขึ้นด้วย (3) การเก็บสถิติการ branch ล่าสุดไว้ หากครั้งล่าสุดมีการ branch ให้เดาว่าครั้งต่อไปมีการ branch เป็นต้น (ยังมีการทำงานของ branch prediction แบบที่ซับซ้อนกว่านี้ ซึ่งมีได้ก็แล้วถึงในที่นี้)



รูปที่ 6.12: ตัวอย่างการทำงานของ Pipeline เมื่อมีการ bypass เพื่อแก้ปัญหา Control Hazard

ข้อสังเกตคือ ไม่ว่าจะแก้ปัญหา control hazard ด้วยแนวทางใด ก็มีแนวโน้มที่จะเกิดรอบที่เสียเปล่า (stall cycle) ไม่เกิดประโยชน์ใด ๆ ได้ จึงมีความพยายามที่จะให้เทคนิคทางซอฟต์แวร์ช่วยเพื่อแก้ปัญหารอบที่เปล่านี้ ดังจะกล่าวถึงในตอนต่อไป

6.4.1 การแก้ปัญหา Control Hazard ด้วยวิธีการ Branch Delay Slot

ถึงตรงนี้ ข้อเสียของการแก้ปัญหา control hazard ไม่ว่าจะแบบใดก็ตามที่ได้ก็ล้าวมาแล้ว จะมีการดึงคำสั่งที่อาจจะผิดเข้ามา ทำให้ต้องมีรอบที่ไม่เกิดประโยชน์เกิดขึ้นอยู่ดี จึงมีการเสนอแนวทางการแก้ปัญหา control hazard โดยใช้วิธีการทางซอฟต์แวร์เข้าร่วมเรียกว่า branch delay slot หรือ delayed branch

เนื่องจากทุกครั้งที่มีการ branch จะไม่ทราบว่าจะต้อง branch หรือไม่จนกว่าจะผ่านไปถึงรอบที่ 3 นั่นหมายความว่า คำสั่งใดก็ตามที่ถูกดึงเข้ามา ก่อนจะทราบผลการ branch ต่างก็

มีแนวโน้มที่จะเสียเวลา สมมุติว่ารอบที่ 3 หลังจากการทำ branch completion แล้ว สามารถดึงคำสั่งใหม่ (fetch) ที่ถูกต้องได้ทันที ดังนั้น หากสถาปัตยกรรมชุดคำสั่งปรับให้รอบตัดماของคำสั่ง branch ไม่เข้ากับการ branch โดยหน่วงการ branch ออกไป 1 รอบ และให้รอบที่ตามมาเป็นการทำงานของคำสั่งปกติ จะได้ว่า ไม่ว่าจะมีการ branch เกิดขึ้น หรือไม่ cycle ที่ตามมา จะเป็นการทำงานของคำสั่งที่ต้องมีการทำงานตามปกติเสมอ

เพื่อความเข้าใจเรื่อง delayed branch ขอให้ลองพิจารณา code ซึ่งไม่มี delayed branch และมี delayed branch ดังต่อไปนี้

```
(1) ADD      $r1, $r2, $r3
(2) BEQ      $r2, $r3, BRANCH
(3) ORI      $r4, $r5, #100
BRANCH:
(9) SW      $r1, 100($r3)
```

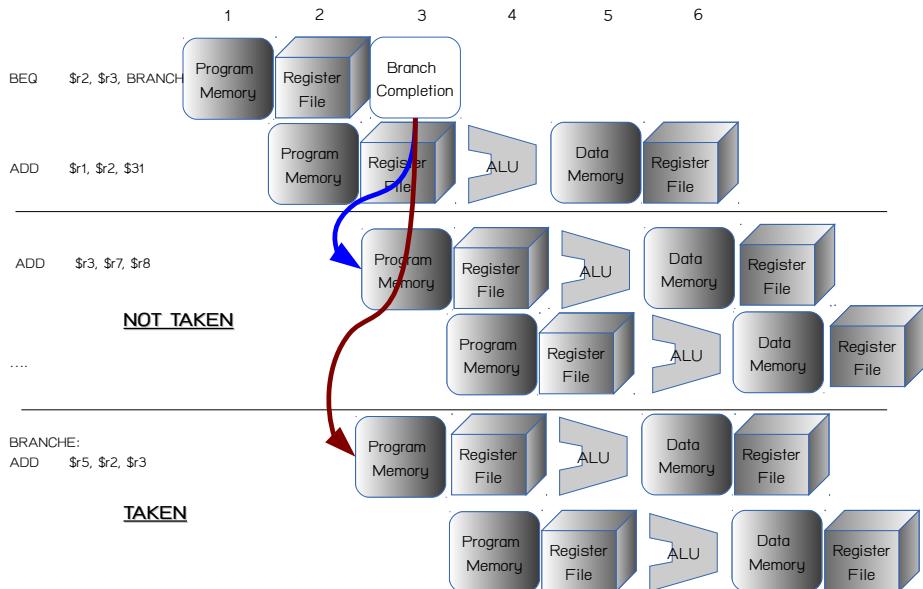
จาก code หากเป็นการทำงานของคำสั่งทั่วไป กรณีที่ branch จะทำงานคำสั่งที่ (9) และ กรณีที่ไม่ branch จะต้องทำงานคำสั่งที่ (3) ในกรณีที่มี delayed branch คำสั่งที่ตามหลัง มา จะไม่มีผลเกี่ยวกับ branch ดังนั้น หากทำการ reschedule code ดังกล่าวสำหรับ รองรับการทำงาน delayed branch จะได้เป็น

```
(2) BEQ      $r2, $r3, BRANCH
(1) ADD      $r1, $r2, $r3
(3) ORI      $r4, $r5, #100
BRANCH:
(9) SW      $r1, 100($r3)
```

เนื่องจากไม่ว่าจะเกิดการ branch หรือไม่ คำสั่งที่ตามมา จะต้องมีการทำงานเสมอ ดังนั้น การย้ายคำสั่ง (1) มาไว้หลัง (2) ทำให้ผลที่ได้คือ ไม่มีการ stall เกิดขึ้นอย่างแน่นอน การทำงานของ delayed branch สามารถอธิบายเป็นแผนภาพได้ดังรูปที่ 6.13

จากรูปที่ 6.13 จะเห็นว่าไม่ว่าจะมีการ branch หรือไม่ คำสั่งที่ตามมา จะมีการประมวลผล เสมอ และผลของ branch not taken จะถูก delay ออกไป 1 คำสั่ง (ซึ่งเป็นที่มาของชื่อ branch delay slot) และกรณี branch taken ก็สามารถอ่านคำสั่งที่เกิดจาก branch ขึ้นมา ได้ทันทีเช่นกัน

ข้อสังเกต ด้วยวิธีการทางฮาร์ดแวร์เพียงอย่างเดียว จะไม่สามารถทำ branch delayed slot ได้ เพราะ ฮาร์ดแวร์ไม่สามารถเลือกหรือสลับคำสั่งที่มาก่อน branch เข้ามาแทนที่ใน delay slot ได้

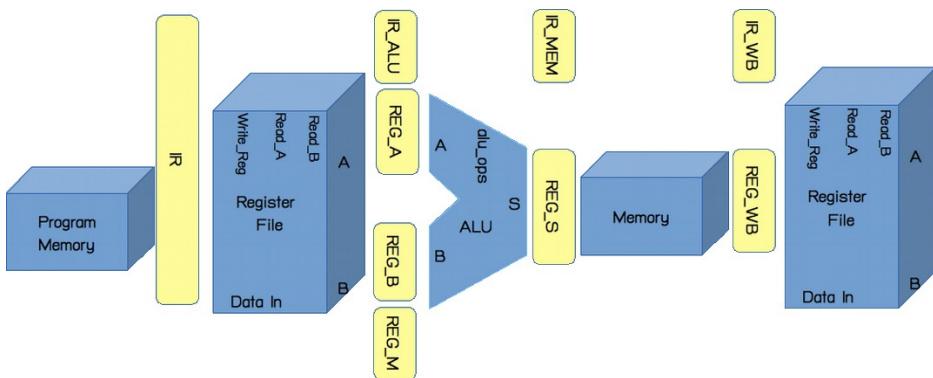


รูปที่ 6.13: การแก้ปัญหา control hazard ด้วย branch delay slot (แบบ taken) และ (not taken)

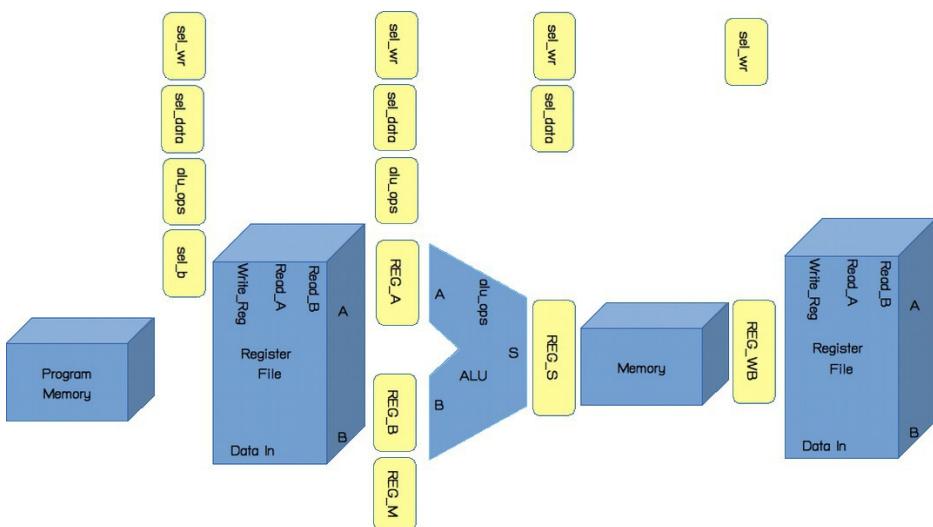
6.5 Data Path

ปัจจัยแรกที่จะทำให้สามารถเริ่มต้นคำสั่งใหม่ได้ทุกรอบคือ การแยกให้แต่ละรอบทำงานได้เป็นอิสระจากกัน มีคำสั่งที่เกี่ยวข้องเป็นของตนเอง ทำให้โครงสร้างภายในของ CPU ที่จะทำ pipeline นั้นมีส่วนประกอบที่มากขึ้นและบางส่วนอาจทำงานช้าขอนอก (ซึ่งแนวทางการใช้อุปกรณ์ที่เข้าช้อนกันนี้ จะขัดแย้งกับแนวทางในการออกแบบหน่วยประมวลผลกลางแบบ multiple cycle ในบทที่ 5 บางส่วนซึ่งเป็นการลดอุปกรณ์เข้าช้อนเพื่อให้ CPU มีขนาดเล็ก) ทั้งนี้เนื่องจากอุปกรณ์ทุกชิ้นต้องทำงานพร้อมกัน ดังแสดงในรูปที่ 6.14

เพื่อให้การทำงานแต่ละขั้นเป็นอิสระจากกัน วิธีการที่ง่ายที่สุดคือการใส่ buffer หรือ เรจิสเตอร์พิเศษ เพื่อช่วยส่งต่อสัญญาณที่ต้องใช้ต่อไป จากรูปที่ 6.14 จะเห็นว่ามีการเพิ่มเรจิสเตอร์คำสั่ง (IR_ALU, IR_MEM, IR_WB) เพื่อแยกว่าคำสั่งที่ทำงานในแต่ละขั้นตอนของ pipeline นั้น เป็นอะไรสักอย่าง (ในทำนองเดียวกัน เราอาจจะส่งต่อสัญญาณควบคุมแทนก็ได้ เพื่อแสดงว่า สัญญาณควบคุมที่ต้องใช้ในขั้นตอนถัดก้าว และขั้นตอนถัดไปเป็นอะไรบ้าง ดังแสดงในรูปที่ 6.15)



รูปที่ 6.14: การแยกเรจิสเตอร์คำสั่งแยกออกจากกันในแต่ละขั้นของ pipeline



รูปที่ 6.15: การส่งผ่านสัญญาณควบคุมสำหรับ pipeline

เนื้อหาในส่วนนี้จะไม่ลงรายละเอียดเกี่ยวกับตัวสายสัญญาณที่เกิดขึ้นจริง เพราะภาพที่ได้จะซับซ้อนจนดูเข้าใจยาก แต่จะขอทิ้งแนวคิดบางส่วนไว้เป็นแบบฝึกหัดท้ายบทแทน

6.6 สรุป Pipeline

การทำ pipeline เป็นการเพิ่ม throughput ให้กับหน่วยประมวลผลกลางที่สามารถทำได้ง่ายเมื่อมีหน่วยต่าง ๆ อยู่แล้ว เพียงแต่ทำการปรับเปลี่ยนสายสัญญาณและวงจรควบคุมให้เหมาะสม หน่วยประมวลผลกลางก็จะสามารถทำงานแบบ pipeline

สถาปัตยกรรมชุดคำสั่งที่เหมาะสมในลักษณะของ register-register architecture มักช่วยอำนวยความสะดวกให้การทำ pipeline ทำได้ง่ายขึ้น

อย่างไรก็ตามการทำ pipeline นั้น มีปัญหาที่เป็นอุปสรรคอยู่ 3 ประการ คือ structural hazard, data hazard, และ control hazard ซึ่งในหนังสือนี้ได้กล่าวถึงแนวทางการแก้ไขแบบพื้นฐานเท่านั้น ในปัจจุบันมีสถาปัตยกรรมที่มีโครงสร้างและการแก้ปัญหา hazard ที่ซับซ้อนมากกว่านี้ เช่น การทำระบบ branch prediction, branch delay slot หรือ การทำ dynamic schedule เป็นต้น นอกจากนี้ ยังมีปัจจัยหนึ่งที่ไม่ได้กล่าวถึงในที่นี้คือเรื่องของ exception ซึ่งจะทำให้การ pipeline ยุ่งยากมากขึ้น

แนวทางหนึ่งซึ่งเป็นที่นิยมใช้ในการแก้ปัญหา hazard ของสถาปัตยกรรมชุดคำสั่งสมัยใหม่ คือ การทำให้ปัญหาดังกล่าวม่องเห็นได้ด้วยซอฟต์แวร์ (หรือคอมไฟเลอร์) และบังคับให้ซอฟต์แวร์ จัดลำดับการทำงานของคำสั่งเพื่อหลีกเลี่ยงการเกิด hazard แทน วิธีการนี้ ทำให้ซอฟต์แวร์ที่ได้มีความซับซ้อนลดลง แต่เพิ่มเงื่อนไขให้กับคอมไฟเลอร์ หรือผู้พัฒนาซอฟต์แวร์แทน

6.6.1 สมรรถนะของ Pipeline ในทางปฏิบัติ

เนื่องจากในทางปฏิบัติ pipeline จะต้องมีการ stall เนื่องจากปัญหา hazard แบบต่าง ๆ อยู่บ้าง ดังนั้นหากทำการวิเคราะห์สมการที่ 6.1 ใหม่ เราจะได้ว่า สมรรถนะของ pipeline ที่แท้จริงจะต้องมีการบวกค่ารอบที่เสียเปล่าเข้าไปด้วย สมการนี้จึงเขียนใหม่ได้ดังสมการที่ 6.2

$$CPU\ TIME_{pipeline} = ([stage_{pipeline} + (count_{instructions} \times CPI)] + stall_{cycle}) \times T_{cycle}$$

สมการที่ 6.2 สมรรถนะของ pipeline เมื่อมี stall cycle

อย่างไรก็ตามหลายตำนานิยมที่จะคิดรอบที่เสียเปล่าเข้าเป็นส่วนหนึ่งของรอบการประมวลผลต่อคำสั่ง ซึ่งเฉลี่ยวรวมกรณีที่มีการ stall เข้าไปด้วย และมักจะค่าคงที่ (no. of pipeline stage) ซึ่งเป็นรอบเริ่มต้นที่ใช้สำหรับการเติม pipeline ให้เต็มตอนเริ่ม จึงทำให้ยังได้สมการ CPU time คงเดิมเหมือนดังสมการที่ 2.10

6.7 แบบฝึกหัดท้ายบท

- การทำ pipeline ช่วยในการเพิ่ม throughput ได้อย่างไร
- จงวาดแผนภาพแสดงการทำงานของ pipeline (แบบรูปที่ 6.1) เมื่อมีจำนวนคำสั่งเป็น 10 คำสั่ง และทุกคำสั่งมีการทำงาน 5 cycle เท่ากัน
- จากข้อ 2 จงหาความสัมพันธ์ระหว่างจำนวน cycle ที่ใช้และจำนวนคำสั่ง
- จงอธิบายปัญหาซึ่งเป็นอุปสรรคต่อการทำ pipeline และเสนอแนวทางแก้ไข
- เหตุใดคำสั่ง SW, BEQ, และ JMP จึงไม่ก่อให้เกิดปัญหา structural hazard แบบคำสั่ง ADD, ORI และ ORUI จงวาดแผนภาพและให้เหตุผลประกอบการอธิบาย
- จากการแสดงการทำงานของ pipeline ดังกล่าว จงแสดงภาพการทำงานของ pipeline เพื่อประมวลผลคำสั่งต่อไปนี้ พร้อมทั้งแสดงการแรเงาที่ถูกต้อง

```

ADD      $r10,$r0,$r1
LW       $r12,20($r5)
ADD      $r3,$r12,$r12
SW       $r3,30($r10)
ADD      $r10,$r10,$r1

```

- จากการที่ได้ในข้อ 2 หากหน่วยประมวลผลกลางดึงกล่าวมี hazard ซึ่งสามารถแก้ได้ด้วยการ stall และ forward จงหาเส้นแสดงเลขที่อยู่ที่ข้อมูลมีการเกี่ยวเนื่องกัน (dependency) และแสดงการเรียงคำสั่งใหม่เพื่อทำการแก้ไขปัญหา data hazard ด้วยซอฟต์แวร์
- เหตุใดการทำงานของหน่วยประมวลผลกลางแบบไม่มี pipeline จึงไม่เกิดปัญหา data hazard จงอธิบาย
- จงแสดงการสร้างทางเดินข้อมูลของสถาปัตยกรรมชุดคำสั่ง nanoLADA เพื่อให้รองรับการทำงานแบบ pipeline ได้
- จากรสถาปัตยกรรมชุดคำสั่ง nanoLADA ที่กำหนด จงแสดงการเปรียบเทียบค่าจำนวนรอบการประมวลผลต่อคำสั่งและรอบสัญญาณนาฬิกาของ หน่วยประมวลผลกลางแบบ single cycle แบบ multiple cycle และแบบ pipeline พร้อมทั้ง

วิเคราะห์ Speedup สูงสุดที่เป็นไปได้

11. จงอธิบายปัญหา WAR และ WAW พร้อมเขียนแผนภาพประกอบคำอธิบาย
12. จงปรับปรุง Verilog HDL ของหน่วยประมวลผลกลางในภาคผนวก ฯ ให้สามารถทำงานแบบ pipeline ได้

7 สถาปัตยกรรมร่วมสมัย

ปัจจุบัน สถาปัตยกรรมคอมพิวเตอร์ได้มีการพัฒนาต่ออยู่ด้วย จนมีความซับซ้อนและมีประสิทธิภาพมากขึ้นเป็นอย่างมาก หากจะนำทฤษฎีแนวคิดมาอธิบายโดยละเอียด หนังสือเล่มนี้ อาจจะต้องมีกว่าพันหน้าจึงจะอธิบายได้ทั้งหมด เพื่อประกอบความเข้าใจในการศึกษาต่อ ยอดเกี่ยวกับสถาปัตยกรรมสมัยใหม่ เนื้อหาในบทนี้ จึงจะเน้นอธิบายแนวคิดพื้นฐานที่อยู่เบื้องหลังการพัฒนาสถาปัตยกรรมแบบต่าง ๆ โดยอาศัยการจำแนกและจัดกลุ่ม โดยจะพยายามอ้างอิงถึงชุดคำสั่งให้น้อยที่สุด

เนื้อหาในบทนี้ จะเริ่มต้นด้วยการทำความเข้าใจอนุกรมวิธานแบบ Flynn เพื่อให้เข้าใจภาพรวมรูปแบบของสถาปัตยกรรมแบบต่าง ๆ จากนั้นจึงจะเชื่อมโยงไปยังรูปแบบสถาปัตยกรรมแบบต่าง ๆ ต่อไป

7.1 อนุกรมวิธานแบบ Flynn³³ (*Flynn's Taxonomy*)

อนุกรมวิธานแบบ Flynn เป็นการแบ่งประเภทของสถาปัตยกรรมคอมพิวเตอร์ โดยแบ่งตามปริมาณของกระแสของคำสั่ง (Instruction Stream) ที่กระทำต่อกระแสของข้อมูล Data Stream) ที่เกิดขึ้น ณ เวลาหนึ่ง ซึ่งจะแบ่งได้เป็น 4 รูปแบบดังนี้

1. Single Instruction Stream Single Data Stream (SISD)
2. Single Instruction Stream Multiple Data Stream (SIMD)
3. Multiple Instruction Stream Single Data Stream (MISD)
4. Multiple Instruction Stream Multiple Data Stream (MIMD)

7.1.1 Single Instruction Stream Single Data Stream (SISD)

แนวคิดแบบ Single Instruction Stream Single Data Stream คือการที่มีชุดคำสั่งหนึ่งชุด ทำงานกับข้อมูลหนึ่งชุด ซึ่งการทำงานของหน่วยประมวลผลกลางที่กล่าวถึงในบทที่ 4

³³ อนุกรมวิธานแบบ Flynn เป็นการจัดแบ่งประเภทของสถาปัตยกรรมคอมพิวเตอร์ที่เสนอโดย Michael J. Flynn ในปี ค.ศ. 1966 ซึ่งแม้จะเป็นวิธีการแบ่งที่ค่อนข้างเก่า แต่ก็ยังสามารถใช้ในการอ้างอิงถึงกลุ่มรูปแบบสถาปัตยกรรมได้อย่างดี

บทที่ 5 และบทที่ 6 ในหนังสือนี้ ล้วนแต่เป็นการทำงานแบบ Single Instruction Stream Single Data Stream ทั้งสิ้น แม้ในสถาปัตยกรรมแบบ pipeline รวมถึงสถาปัตยกรรมแบบมีหลายหน่วยประมวลผล (SuperScalar) (ซึ่งจะอธิบายในส่วนที่ 7.3) ซึ่งแม้ภายในหน่วยประมวลผลกลางจะมีการประมวลผลคำสั่งหลายคำสั่งพร้อมกัน แต่ก็จัดอยู่ในกลุ่มนี้เช่นกัน เนื่องจากในการพัฒนาซอฟต์แวร์สำหรับสถาปัตยกรรมในกลุ่มนี้ ผู้พัฒนายังคงเขียนเพียงหนึ่งชุดคำสั่งเพื่อทำงานกับข้อมูลหนึ่งชุดแล้วหน่วยประมวลผลกลางจะทำการประมวลผลแบบขนานภายในแต่ให้ผลลัพธ์เหมือนกับการทำงานตามลำดับตามปกติ

7.1.2 Single Instruction Stream Multiple Data Stream (SIMD)

แนวคิดแบบ Single Instruction Stream Multiple Data Stream คือการที่มีชุดคำสั่งหนึ่งทำงานกับข้อมูลหลายชุด สถาปัตยกรรมในลักษณะนี้ที่พัฒนาในหน่วยประมวลผลแบบเวกเตอร์ ซึ่งอยู่ภายใต้หน่วยประมวลผลด้านกราฟิกส์ (GPU) เพราะการทำงานด้านภาพและเสียง มักจะใช้ชุดคำสั่งเดียวกันทำงานกับข้อมูลขนาดใหญ่ เช่น วิธีการหนึ่งในการเปลี่ยนสีภาพซึ่งแสดงสีของแต่ละจุดด้วยข้อมูลขนาด 24 บิต สามารถทำได้โดยการทำ exclusive or กับทุกไบต์ของข้อมูล แต่เนื่องจากภาพขนาดใหญ่ อาจจะมีจำนวนจุดเป็นล้านจุด การใช้หน่วยประมวลผลที่มีความสามารถแบบเวกเตอร์นี้ จะช่วยให้การประมวลผลตั้งกล่าวทำได้เร็วขึ้น นอกจากนี้ การประมวลผลด้านการเข้ารหัสและลดอัตราหักเม็ดที่คล้ายคลึงกัน

ในปัจจุบันหน่วยประมวลผลกลางมักจะมีส่วนต่อขยายที่มีความสามารถด้านเวกเตอร์ และผู้ผลิตบางราย เช่น Intel ก็จะใช้ชื่อ SIMD³⁴ เป็นชื่อเรียกส่วนต่อขยายดังกล่าวด้วย นอกจากนี้ยังมี AltiVec ของ Motorola ซึ่งถูกนำมาใช้ในสถาปัตยกรรม PowerPC ด้วย รายละเอียดเพิ่มเติม ดูได้ในส่วนที่ 7.2

7.1.3 Multiple Instruction Stream Single Data Stream (MISD)

แนวคิดแบบ Multiple Instruction Stream Single Data Stream นั้น เป็นแนวคิดของการประมวลผลที่ไม่ค่อยพบในระบบคอมพิวเตอร์ทั่วไป แต่พบได้ทั่วไปในการพัฒนาโปรแกรมของระบบคอมพิวเตอร์แบบฝังตัว (embedded systems) ตัวอย่างที่พบเห็นอันหนึ่งคือระบบคอมพิวเตอร์ในระบบขับเคลื่อนของเครื่องบิน ซึ่งมักจะมีระบบคอมพิวเตอร์ย่อย 3 ระบบทำงานกับข้อมูลชุดเดียวกัน แล้วนำผลที่ได้มาร่วมกันประมวลผล เพื่อประกอบการตัดสินใจขั้นสุดท้ายในลักษณะของเสียงข้างมาก (majority vote) หากผู้อ่านสนใจ

³⁴ Intel ได้นำแนวคิดแบบเวกเตอร์ มาใช้กับหน่วยประมวลผลกลางทั่วไปตั้งแต่ระบบ MMX, SSE, และ AVX รายละเอียดเพิ่มเติมดูได้ที่ <https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions>

สถาปัตยกรรมในลักษณะนี้ สามารถศึกษาเพิ่มเติมได้จากวิชา fault-tolerant computer systems

7.1.4 Multiple Instruction Stream Multiple Data Stream (MIMD)

แนวคิดแบบ Multiple Instruction Stream Multiple Data Stream เป็นแนวคิดของสถาปัตยกรรมคอมพิวเตอร์ที่สามารถประมวลได้หลายโปรแกรมพร้อมกันอย่างเป็นอิสระ จากกัน บ่อยครั้งโปรแกรมเหล่านั้นจะซวยกันประมวลผลงานขนาดใหญ่ (task-level parallelism) ในปัจจุบันที่เรามักใช้คอมพิวเตอร์แบบหลายหน่วยประมวลผลในหนึ่งชิป (multi-core processor Chip) ซึ่งก็จัดเป็นสถาปัตยกรรมในลักษณะนี้ เช่นเดียวกัน บางตำรา จะเรียกสถาปัตยกรรมแบบนี้ว่า สถาปัตยกรรมการประมวลผลแบบกระจาย (Distributed Computer Systems)

นอกจากนี้ ยังมีสถาปัตยกรรมแบบ Very Long Instruction Word (VLIW) ซึ่งเวลาพัฒนา ซอฟต์แวร์จะต้องคิดว่าจะทำคำสั่งอะไรบ้างพร้อมกันภายใต้สัญญาณนาฬิกาเดียวกันกับชุด ข้อมูลหลายชุด เช่น หากเป็น Very Long Instruction Word แบบสามหน่วย ซอฟต์แวร์จะ ต้องกำหนดว่าจะทำคำสั่งอะไรสามคำสั่งกับข้อมูลสามชุดภายใต้สัญญาณนาฬิกา หาก พิจารณาตามนิยามแล้ว จะเห็นว่าสถาปัตยกรรมลักษณะนี้ จัดเป็น Multiple Instruction Stream Multiple Data Stream แบบหนึ่ง รายละเอียดเพิ่มเติมของสถาปัตยกรรมแบบ Very Long Instruction Word ถูกลากไว้ในส่วนที่ 7.4

7.2 สถาปัตยกรรมแบบเวกเตอร์

สถาปัตยกรรมแบบเวกเตอร์ จัดเป็น Single Instruction Stream Multiple Data Stream ตามแนวคิดของอนุกรมวิธานแบบ Flynn หลักการของเวกเตอร์คือ หนึ่งคำสั่งทำงานกับ ข้อมูลหลายชุด เพื่อให้เห็นข้อแตกต่างจากเปรียบเทียบโปรแกรมที่ทำงานลักษณะเดียวกัน บนสถาปัตยกรรมแบบหน่วยประมวลผลปกติ และสถาปัตยกรรมแบบเวกเตอร์ในรูปที่ 7.1

สถาปัตยกรรมแบบหน่วยประมวลผลเดี่ยว	สถาปัตยกรรมแบบเวกเตอร์
<pre>int A[16]; int B[16]; int C[16]; for (int i=0;i<16;i++) { C[i] = A[i] + B[i]; }</pre>	<pre>int A[16]; int B[16]; int C[16]; C = A + B;</pre>

รูปที่ 7.1: เปรียบเทียบโปรแกรมบนสถาปัตยกรรมแบบหน่วยประมวลผลปกติ และ สถาปัตยกรรมแบบเวกเตอร์

จากตัวอย่างจะเห็นว่า การพัฒนาโปรแกรมบนหน่วยประมวลผลปกตินั้น ผู้พัฒนาซอฟต์แวร์ จะต้องคิดเรื่องการวนรอบ เพื่อให้น่วยประมวลผลคลังทำการประมวลผลข้อมูลทีลีช่อง เอง ในขณะที่สถาปัตยกรรมแบบเวกเตอร์นั้น ผู้พัฒนาซอฟต์แวร์จะเพียงแค่ระบุว่าต้องการ นำผลลัพธ์จาก A และ B แล้วเก็บผลลัพธ์ไว้ที่ C ซึ่งน่วยประมวลผลคลังจึงใช้ชุดการ นำแบบเวกเตอร์ภายในทำการบวกข้อมูลหลายชุดพร้อมกัน ซึ่งถ้าหากน่วยประมวลผล เวกเตอร์ที่ใช้มีหน่วยบวกเลขของข้อมูลประเภทหนึ่งน้อยกว่า 16 หน่วย การประมวลผลคำ สั่งถูกกล่าว จะใช้เพียง 1 รอบการทำงานเท่านั้น เปรียบเทียบกับการประมวลผลของหน่วย ประมวลผลเดี่ยวซึ่งเมื่อแต่ละรอบออกมาน้ำจะใช้มากถึง 5 รอบการทำงานต่อการวน รอบ คุณกับจำนวนรอบอีก 16 รอบ ตัวอย่างนี้อาจจะใช้มากถึง 80 รอบการทำงานของคำ สั่งเทียบกับสถาปัตยกรรมแบบเวกเตอร์จะพบว่า Speedup มากถึง 80 เท่า

ในปัจจุบัน แนวคิดแบบเวกเตอร์นี้ ถูกประยุกต์ให้ทำงานร่วมกับหน่วยประมวลผลแบบปกติ ในลักษณะของหน่วยประมวลผลร่วม (co-processor) โดยถูกนำมาเสริมในคอมพิวเตอร์ใน รูปแบบของประมวลผลด้านกราฟิก ซึ่งแม้ซึ่งจะเป็นหน่วยประมวลผลด้านกราฟิก แต่ได้มี การพัฒนาแนวคิดให้นำมาช่วยในการพัฒนาซอฟต์แวร์ทั่วไปอย่างแพร่หลาย แนวคิดนี้ปรากฏในรูปแบบของการพัฒนาซอฟต์แวร์ด้วย CUDA³⁵ และ OpenCL³⁶

35 CUDA เป็นเทคโนโลยีของบริษัท NVIDIA ในการนำหน่วยประมวลผลทางกราฟิกมาประยุกต์ใช้กับ การประมวลผลทั่วไป ภายในประกอบด้วยชุดคำสั่งสำหรับสนับสนุนการเขียนโปรแกรมแบบเวกเตอร์ มากมาย รายละเอียดเพิ่มเติมดูได้จาก http://www.nvidia.com/object/cuda_home_new.html

36 OpenCL เป็นแนวคิดในการพัฒนามาตรฐานกลางสำหรับการเขียนโปรแกรมแบบเวกเตอร์ เพื่อใช้งาน บนหน่วยประมวลผลกราฟิก ฯ ริเริ่มโดยบริษัท Apple รายละเอียดเพิ่มเติมดูได้จาก <https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf>

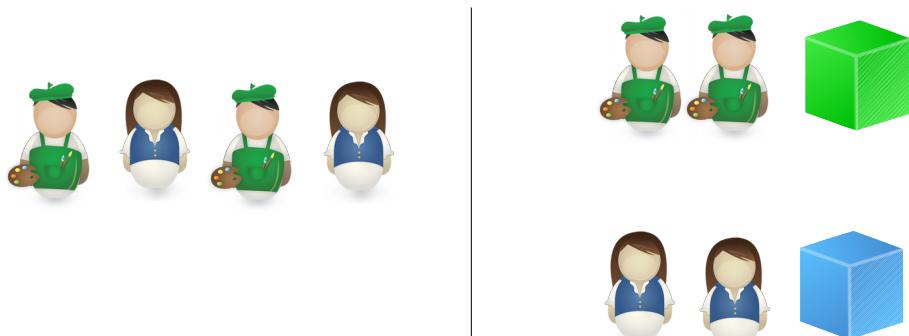
7.3 สถาปัตยกรรมแบบ SuperScalar

สถาปัตยกรรมแบบ SuperScalar จัดเป็น Single Instruction Stream Single Data Stream ตามแนวคิดของอนุกรมวิธานแบบ Flynn หลักการของ SuperScalar คือ ภายในหน่วยประมวลผลกลาง จะมีหน่วยสำหรับการคำนวณและประมวลผลที่ทำงานได้พร้อมกันอยู่หลายหน่วย แต่ผู้พัฒนาซอฟต์แวร์ยังคงพัฒนาซอฟต์แวร์ตามปกติ จากนั้นหน่วยประมวลผลกลางจะทำการแปลงซอฟต์แวร์ให้มีการกระจายคำสั่งต่าง ๆ ไปทำงานยังหน่วยประมวลผลย่อยที่เกี่ยวข้องได้หลายคำสั่งพร้อมกัน

ด้วยแนวคิดที่ให้ซอฟต์แวร์ที่พัฒนาในลักษณะเดิม สามารถแทรกการทำงาน ทำหลายคำสั่งพร้อมกันได้ในหน่วยประมวลผลกลาง หลายตำราจึงเรียกการทำงานของ SuperScalar ว่า เป็นการจัดตารางการทำงานแบบพลวัต (dynamic scheduling) ซึ่งแนวคิดนี้ได้ถูกอธิบายในรายละเอียดด้วยอัลกอริทึมของ Tomasulo (รายละเอียดของอัลกอริทึมดังกล่าว อยู่นอกขอบเขตของหนังสือเล่มนี้) มีแนวคิดเบื้องต้นคือ การให้แต่ละหน่วยประมวลผลย่อยทำงาน เมื่อมีข้อมูลพร้อม ผลที่ได้คือหน่วยประมวลผลกลางไม่จำเป็นต้องทำการประมวลผลโปรแกรมตามลำดับคำสั่งของโปรแกรมที่มีอยู่ (out-of-order execution) แต่ยังคงยืนยันให้โปรแกรมทำงานถูกต้องตามลำดับ (in-order completion)

ในรูปที่ 7.2 เป็นการจำลองสถานการณ์ของการทำงานแบบ SuperScalar โดยสมมุติว่ามีกลุ่มคนอยู่สองกลุ่มเข้าแถว กันเพื่อซื้อสินค้า กลุ่มหนึ่งต้องการซื้อสินค้าเกษตร อีกกลุ่มต้องการซื้อเครื่องใช้สำนักงาน ในการเข้าแถว (เปรียบเทียบกับซอฟต์แวร์) ยังคงเป็นการเข้าแถวตอนเรียงหนึ่ง เพื่อเข้ามาภายในพื้นที่จำหน่ายสินค้า ทันทีที่เข้ามาในพื้นที่จำหน่าย หากภายในพื้นที่แบ่งส่วนการให้บริการของสินค้าเกษตรและเครื่องใช้สำนักงานออกจากกัน จะสามารถให้บริการผู้ที่ต้องการซื้อสินค้าเกษตรพร้อมกับผู้ที่ต้องการซื้อเครื่องใช้สำนักงานได้พร้อมกัน

เมื่อวิเคราะห์เพิ่มเติม จะเห็นว่าเวลาที่ใช้ในการประมวลผลจะลดลง ซึ่งเวลาโดยรวมจะเท่ากับเวลาที่ให้บริการของส่วนที่ชาที่สุด เช่น หากการให้บริการลูกค้าที่ซื้อสินค้าเกษตรจะใช้เวลา 2 นาทีต่อราย และ การให้บริการลูกค้าที่ซื้อเครื่องใช้สำนักงานจะใช้เวลา 1 นาทีต่อราย หากเรามีลูกค้าทั้งสองส่วนเฉลี่ยเท่ากัน เวลาที่ใช้ในการประมวลผล จะเท่ากับเวลาที่ให้บริการลูกค้าที่ซื้อสินค้าเกษตรทั้งหมด (เพราะลูกค้าสำนักงานจะรับบริการเสร็จก่อนเสมอ)



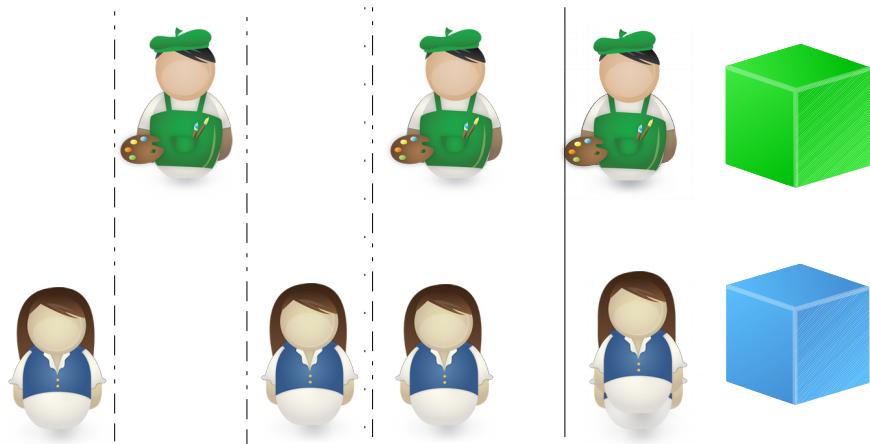
รูปที่ 7.2: สถานการณ์จำลองการทำงานแบบ SuperScalar

ในท่านองเดียวกัน หากหน่วยประมวลผลกลางแยกการประมวลผลแบบเลขจำนวนเต็ม และการประมวลผลแบบเลขศนิยมออกจากกัน จะช่วยให้สามารถประมวลผลเลขจำนวนเต็ม และเลขศนิยมพร้อมกันได้ ช่วยให้ความเร็วโดยรวมมากขึ้น หากหน่วยประมวลผลกลาง แยกการทำงานเป็นสองส่วนได้พร้อมกันในลักษณะนี้ เราจะเรียกว่าหน่วยประมวลผลกลาง แบบ SuperScalar ดังกล่าวมีดีกรีสอง (เรียกจำนวนดีกรีตามจำนวนหน่วยประมวลผลย่อย กายใน)

ในแห่งของสมรรถนะ สถาปัตยกรรมแบบ SuperScalar ช่วยให้เราสามารถมีค่าจำนวนรอบ การประมวลผลต่อคำสั่งน้อยกว่า 1 ได้ แต่เมื่อจากสถาปัตยกรรมแบบนี้จะมีคำสั่งที่ถูก ประมวลผลและทำงานเสร็จมากกว่า 1 คำสั่งต่อรอบการทำงาน ซึ่งหากทุกรอบมีคำสั่งที่ ทำงานเสร็จสองคำสั่ง (SuperScalar ดีกรีสอง) หมายความว่า ค่าจำนวนรอบการประมวล ผลต่อคำสั่งจะเป็น 0.5

7.4 สถาปัตยกรรมแบบ Very Long Instruction Word (VLIW)

สถาปัตยกรรมแบบ Very Long Instruction Word จัดเป็น Multiple Instruction Stream Multiple Data Stream แบบหนึ่ง ตามแนวคิดของอนุกรมวิธานแบบ Flynn โดยหลักการ แล้ว สถาปัตยกรรมแบบนี้ ให้ผลการทำงานในแห่งสมรรถนะเทียบเท่ากับสถาปัตยกรรมแบบ SuperScalar เพียงแต่เปลี่ยนการเรียงลำดับของคำสั่งแบบพลวัตที่ทำในหน่วยประมวลผล กลาง เป็นการเรียงลำดับแบบสถิติโดยผู้พัฒนาซอฟต์แวร์หรือคอมไพล์เตอร์ดังแสดงในรูปที่ 7.3



รูปที่ 7.3: สถาปัตยกรรมจำลองการทำงานแบบ Very Long Instruction Word

ข้อสังเกตคือ ในสถาปัตยกรรมแบบ Very Long Instruction Word นั้น ระบบซอฟต์แวร์ จะต้องกำหนดตั้งแต่ตัวว่าในลำดับถัดไป จะต้องส่งคำสั่งใดเข้าไปในหน่วยประมวลผล ต่างจากแบบ SuperScalar ที่ซอฟต์แวร์ไม่จำเป็นจะต้องทำการแก้ไขสิ่งใด ด้วยเหตุนี้ สถาปัตยกรรมแบบ Very Long Instruction Word ที่มีโครงสร้างต่างกันจึงไม่สามารถใช้งานซอฟต์แวร์ที่มีอยู่เดิมได้ ตัวอย่างเช่น สถาปัตยกรรม Intel Itanium และ Intel Itanium II แม้จะมีความคล้ายคลึงกันและเป็นผู้ผลิตเดียวกัน แต่จะไม่สามารถใช้ซอฟต์แวร์ร่วมกันได้ ต้องทำการแปลงซอฟต์แวร์ด้วยคอมไฟเลอร์ใหม่ๆ เสมอ

ในอดีตมีการศึกษาและคาดเดาว่าการที่คอมไฟเลอร์เป็นผู้จัดลำดับคำสั่งนั้น น่าจะใช้พลอย่างของสมรรถนะเดิก่าว่าการให้หน่วยประมวลผลกลางจัดลำดับ เพราะคอมไฟเลอร์จะเห็นภาพรวมของโปรแกรมทั้งหมดในขณะที่หน่วยประมวลผลกลางจะเห็นเพียงบางส่วนของชุดคำสั่งเท่านั้น แต่เมื่อเวลาผ่านไปพบว่าการสร้างคอมไฟเลอร์เพื่อให้เรียงคำสั่งได้ดีนั้นเป็นสิ่งที่ทำได้ยาก(มาก) ปัจจุบันความนิยมในสถาปัตยกรรมลักษณะนี้จึงลดลง เหลือการใช้งานอยู่เพียงในเครื่องแม่ข่ายขนาดใหญ่เท่านั้น

7.5 การสนับสนุนของตัวแปรภาษาในสถาปัตยกรรมร่วมสมัย

ในสถาปัตยกรรมสมัยใหม่ ระบบไฮาร์ดแวร์มักจะมีข้อจำกัดในการจัดลำดับการทำงานของ

โปรแกรม จึงเกิดแนวคิดในการออกแบบให้ระบบฮาร์ดแวร์ทำงานร่วมกับระบบซอฟต์แวร์ แปลภาษา (hardware/software co-design) เพื่อให้การทำงานโดยรวมมีสมรรถนะมากขึ้น

แนวทางหนึ่งที่จะช่วยให้ฮาร์ดแวร์สามารถที่จะประมวลผลซอฟต์แวร์ได้อย่างมีสมรรถนะมากขึ้นคือ การขยายให้ basic block มีขนาดใหญ่ขึ้น ส่วนหนึ่งเนื่องจากการทำงานของฮาร์ดแวร์และซอฟต์แวร์ในการจัดลำดับคำสั่งจะถูกจำกัดอยู่ภายใน block เดียวกันเท่านั้น แนวทางในการขยาย basic block ที่นิยมใช้กันมีสองแนวทาง คือ loop unroll และ software pipeline ซึ่งจะอธิบายในส่วนที่ 7.5.1 และ 7.5.2 ตามลำดับ ปัจจุบันยังมีการพัฒนาระบบ LLVM เพื่อเป็นภาษากลางที่ช่วยให้การแปลภาษาทำได้ดีขึ้น รายละเอียดจะกล่าวในส่วนที่ 7.5.3 ต่อไป

7.5.1 Loop unroll

การทำ loop unroll เป็นการทำให้เนื้อโปรแกรมในการทำงาน 1 รอบมีขนาดใหญ่ขึ้น แม้โดยพื้นฐานแล้ว การทำงานของโปรแกรมมีได้แตกต่างจากเดิมแต่อย่างไร และคุณผู้อ่านยังทำให้โปรแกรมในส่วนของเนื้อร้องรอบมีขนาดใหญ่ขึ้นด้วย แต่ผลทางอ้อมที่ได้คือ การที่มีตัวเลือกของคำสั่งในการจัดลำดับมากขึ้น ทำให้สามารถลดการเกิด stall ได้อย่างมีประสิทธิภาพ รูปที่ 7.4 แสดงโปรแกรมในภาษาซีเมื่อมีการทำ loop unroll จำนวน 4 รอบ (ในรูปสมมุติให้จำนวนการวนรอบหารด้วย 4 ลงตัว)

การวนรอบปกติ	การทำ loop unroll 4 รอบ
<pre>for (i=MAX-1; i>=0; i--) { A[i] = A[i] + C; }</pre>	<pre>for (i=MAX-1; i>=3; i-=4) { A[i] = A[i] + C; A[i-1] = A[i-1] + C; A[i-2] = A[i-2] + C; A[i-3] = A[i-3] + C; }</pre>

รูปที่ 7.4: การทำ loop unroll จำนวน 4 รอบ

เมื่อแปลโปรแกรมในรูปที่ 7.4 เป็นภาษาแอสเซมบลีจะพบว่าโปรแกรมที่ได้จากการทำ loop unroll จะมีคำสั่งมากขึ้น ทำให้สามารถเรียงลำดับใหม่จนไม่เกิด stall ได้ง่ายกว่าโปรแกรมที่มีการวนรอบปกติตั้งแต่ในรูปที่ 7.5

ข้อสังเกตคือ การทำ loop unroll แม้จะทำให้ผลโปรแกรมที่ได้เร็วขึ้น แต่ก็แลกมาด้วย

โปรแกรมที่มีขนาดใหญ่ขึ้น จากรูปที่ 7.5 โปรแกรมที่ไม่มีการทำ loop unroll จะต้องใช้จำนวน 8 รอบสัญญาณนาฬิกา เพื่อทำงาน 1 วนรอบ ในขณะที่โปรแกรมที่มีการทำ loop unroll จะใช้จำนวน 14 รอบสัญญาณนาฬิกา เพื่อทำงาน 4 วนรอบ หรือคิดเป็น 3.5 รอบ สัญญาณนาฬิกาต่อการทำงาน 1 วนรอบ หากคิดว่าจำนวนรอบ(ค่า MAX)เยอะมากจะได้ว่า ค่า Speedup คิดเป็น 2.29 เท่า โดยคำนวณจาก

$$\text{Speedup} = \lim_{n \rightarrow \infty} \frac{8 \times n}{3.5 \times n} = 2.29$$

(เมื่อ n แทนจำนวนรอบซึ่งมีค่าเยอะมากจนเทียบเป็นอนันต์)

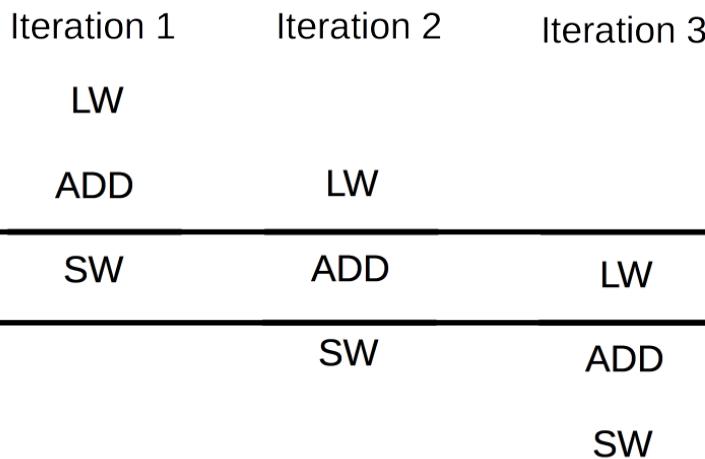
การวนรอบปกติ	การทำ loop unroll 4 รอบ (ก่อนเรียงลำดับ)	การทำ loop unroll 4 รอบ (หลังเรียงลำดับ)
<pre> LOOP: LW \$f0, \$r1 (stall) ADD \$f2, \$f0,\$f1 (stall) (stall) SW \$f2, \$r1 SUB \$r1, \$r1,#4 BNE \$r1, LOOP </pre>	<pre> LOOP: LW \$f0, 0(\$r1) (stall) ADD \$f2, \$f0, \$f1 (stall) (stall) SW \$f2, 0(\$r1) LW \$f3, 4(\$r1) (stall) ADD \$f4, \$f3, \$f1 (stall) (stall) SW \$f4, 4(\$r1) LW \$f5, 8(\$r1) (stall) ADD \$f6, \$f5, \$f1 (stall) (stall) SW \$f6, 8(\$r1) LW \$f7, 12(\$r1) (stall) ADD \$f8, \$f6, \$f1 (stall) (stall) SW \$f8, 12(\$r1) SUB \$r1, \$r1,#16 BNE \$r1, LOOP </pre>	<pre> LOOP: LW \$f0, 0(\$r1) LW \$f3, 4(\$r1) LW \$f5, 8(\$r1) LW \$f7, 12(\$r1) ADD \$f2, \$f0, \$f1 ADD \$f4, \$f3, \$f1 ADD \$f6, \$f5, \$f1 ADD \$f8, \$f6, \$f1 SW \$f2, 0(\$r1) SW \$f4, 4(\$r1) SW \$f6, 8(\$r1) SW \$f8, 12(\$r1) SUB \$r1, \$r1,#16 BNE \$r1, LOOP </pre>

รูปที่ 7.5: การทำ loop unroll ในภาษาแօสเซมบลี

ข้อชวนคิด จะสังเกตว่าการทำ loop unroll จะต้องใช้เรจิสเตอร์เพิ่มขึ้นเสมอ (หลักการนี้เรียกว่า register renaming) ดังนั้นทรัพยากรที่จำกัดจำนวนรอบของการทำ loop unroll คือ จำนวนเรจิสเตอร์ที่มีในระบบ อย่างไรก็ตามในความเป็นจริงความสามารถใช้เรจิสเตอร์ช้า ได้เมื่ออยู่นอกขอบเขตที่มีการอ้างอิง ดังนั้นหากจะดำเนินการจริง จะเห็นว่าความสามารถของ unroll จนถึงจุดที่การวนรอบหายไปเลยก็ได้ ทั้งนี้อย่างให้ทำการเปรียบเทียบจำนวนของโปรแกรมที่ใหญ่ขึ้น คุ้มค่ากับความเร็วที่ได้มากขึ้นหรือไม่

7.5.2 Software pipeline

เพื่อแก้ปัญหาโปรแกรมที่มีขนาดใหญ่ขึ้น จึงเกิดแนวคิดการทำ software pipeline โดยหลักการคือ ให้เนื้อโปรแกรมหลักมาจากการรอบที่ต่างกัน กล่าวคือในเนื้อโปรแกรมส่วนวนรอบจะเป็นการเก็บค่าที่ได้คำนวณแล้วในรอบก่อน การคำนวณข้อมูลที่ได้จากการโหลดในรอบก่อน และเป็นการโหลดข้อมูลเพื่อจะใช้ในรอบถัดไป เพื่อประกอบความเข้าใจ รูปที่ 7.6 เป็นแนวคิดการทำ software pipeline ส่วน รูปที่ 7.7 แสดงการทำ software pipeline เมื่อเนื้อหลักของโปรแกรมมี 3 รอบ



รูปที่ 7.6: แนวคิดการทำ software pipeline

การทำ loop unroll 3 รอบ	การทำ software pipeline
<pre> LOOP: LW \$f0, 0(\$r1) LW \$f3, 4(\$r1) LW \$f5, 8(\$r1) ADD \$f2, \$f0, \$f1 ADD \$f4, \$f3, \$f1 ADD \$f6, \$f5, \$f1 SW \$f2, 0(\$r1) SW \$f4, 4(\$r1) SW \$f6, 8(\$r1) SUB \$r1, \$r1, #12 BNE \$r1, LOOP </pre>	<pre> SUB \$r1, \$r1, #4 LW \$f0, 4(\$r1) ADD \$f2, \$f0, \$f1 LW \$f0, 0(\$r1) LOOP: SW \$f2, 0(\$r1) ADD \$f2, \$f0, \$f1 LW \$f0, 8(\$r1) SUB \$r1, \$r1, #4 BNE \$r1, LOOP SW \$f2, 4(\$r1) ADD \$f2, \$f0, \$f1 SW \$f2, 0(\$r1) </pre>

รูปที่ 7.7: การทำ software pipeline

ในรายละเอียด การแปลงโปรแกรมเป็น software pipeline คอมไไฟเลอร์จะดำเนินการโดย อัตโนมัติ จึงไม่ขอลายละเอียดในที่นี้

7.5.3 โครงการ LLVM

โครงการ LLVM³⁷ เป็นการจัดทำชุดคอมไไฟเลอร์และชุดเครื่องมือที่เกี่ยวข้องกับการพัฒนาซอฟต์แวร์ โครงการวิจัยดังกล่าวเริ่มต้นโดย University of Illinois เพื่อพัฒนาเครื่องมือสนับสนุนการทำงานของคอมไไฟเลอร์สำหรับภาษาทั่วไป ในโครงการนี้ประกอบด้วยโครงการอยู่อย่างมากมาย เช่น

- **LLVM Core** เป็นการสร้างระบบการแสดงผลโปรแกรมแบบไม่เข้ากับภาษา คอมพิวเตอร์และสถาปัตยกรรม โดยใช้ LLVM intermediate representation (LLVM IR) เป็นชุดคำสั่งในการอธิบายการทำงานของโปรแกรม ด้วยความสมบูรณ์และเข้าใจง่ายของชุดคำสั่งตั้งกล่าว ทำให้การสร้างภาษา ตัวแปลงภาษา การหาค่าที่ดีที่สุดและแปลเป็นภาษาเครื่องทำได้ง่าย
- **Clang** เป็นตัวแปลงภาษา C, C++ และ Objective-C ที่ใช้ LLVM เป็นแกนสำหรับการแปลงภาษา ในช่วงต้นของการพัฒนานั้นโปรแกรมที่ได้จากการแปลงภาษาของ Clang มีสมรรถนะสูงกว่าโปรแกรมเดิมกันที่แปลงด้วย

37 ดูรายละเอียดเพิ่มเติมที่ <https://www.llvm.org/>

gcc ลีน 3 เท่า รูปที่ 7.8 เป็นตัวอย่างการแปลงโปรแกรมภาษาซี เป็น LLVM IR ของ Clang

- OpenMP เป็นแนวคิดในการสร้างโปรแกรมที่รองรับการทำงานของหน่วยประมวลผลกลางแบบหลายแกนจากภาษาซีโดยใช้ preprocessor และ LLVM IR ช่วยแปลงโปรแกรมที่ทำงานบนหน่วยประมวลผลแบบหนึ่งแกนให้ทำงานบนหน่วยประมวลผลแบบหลายแกนได้อัตโนมัติ

ภาษาซี	LLVM IR
<pre>#include <stdio.h> int main (int argc, char *argv[]) { printf("Hello, World\n"); }</pre>	<pre>.str = private unnamed_addr constant [14 x i8] c"Hello, World\0A\00", align 1 ; Function Attrs: noinline nounwind ssp uwtable define i32 @main(i32, i8**) #0 { %3 = alloca i32, align 4 %4 = alloca i8**, align 8 store i32 %0, i32* %3, align 4 store i8** %1, i8*** %4, align 8 %5 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i32 0, i32 0)) ret i32 0 } declare i32 @printf(i8*, ...) #1</pre>

รูปที่ 7.8: ตัวอย่าง LLVM IR

จากรูปที่ 7.8 จะเห็นว่า LLVM IR ไม่เชื่อมสถาปัตยกรรมและยังคงรายละเอียดของขนาดข้อมูลไว้เพื่ออำนวยความสะดวกในการแปลงเป็นภาษาเครื่องต่อไป

ยังมีโครงการอยู่ในชุดโครงการ LLVM อิกามากมายที่ช่วยสนับสนุนการแปลงภาษาในสถาปัตยกรรมร่วมสมัยซึ่งมีได้ก่อนลีนในที่นี้ และ ในปัจจุบันคอมไพเลอร์ของบริษัท Apple ที่ใช้ในชุดพัฒนาซอฟต์แวร์สำหรับระบบปฏิบัติการ Mac OS X (สถาปัตยกรรม Intel x86) และ iOS (สถาปัตยกรรม ARM) ใช้ระบบของ Clang และ LLVM เป็นพื้นฐาน ซึ่งโปรแกรมที่ได้บางส่วนจะถูกบันทึกเป็น LLVM IR และจะถูกแปลงให้เหมาะสมกับสถาปัตยกรรมที่โปรแกรมกำลังทำงานอยู่ เช่น เมื่อตรวจสอบว่าเครื่องคอมพิวเตอร์ที่กำลังประมวลผลมีหน่วยประมวลผลแบบ GPU อยู่ โปรแกรมก็จะถูกแปลงเพื่อให้สามารถใช้ความสามารถของ GPU ได้ทันที สิ่งนี้แสดงให้เห็นถึงความสามารถของ LLVM ในการพัฒนาซอฟต์แวร์ข้ามสถาปัตยกรรมได้เป็นอย่างดี

ส่วนประกอบของ LLVM ยังถูกนำไปใช้ในการพัฒนาระบบทางานของภาษาอื่น ๆ อีกมากมาย ตัวอย่างเช่น Ruby, Python, Haskell, Java, D, PHP, Pure และ Lua ซึ่งแสดงถึงความเป็นภาษาอเนกประสงค์ที่มีความยืดหยุ่นและสามารถนำส่วนต่าง ๆ กลับมาใช้ได้ใหม่ของ LLVM เช่น การพัฒนาคอมไไฟเลอร์และระบบการทำงานของภาษาสมัยใหม่จึงนิยมใช้ LLVM เป็นแกน

7.6 สรุปสถาปัตยกรรมร่วมสมัย

นอกจากนี้ยังมีสถาปัตยกรรมแบบ SuperPipeline³⁸ ที่การทำงานเหมือนกับสถาปัตยกรรมแบบ pipeline เพียงแต่ใช้สัญญาณนาฬิกาที่ละเอียดมาก ในแต่ละรอบจะให้ผลที่ใกล้เคียงกับสถาปัตยกรรมแบบ SuperScalar จึงมีได้ล่าวถึงในทัน

ในปัจจุบันสถาปัตยกรรมที่นิยมใช้และมีจำเป็นในท้องตลาด จะเป็นแบบ multi-core chip กล่าวคือเป็นสถาปัตยกรรมที่มีหน่วยประมวลผลแบบ pipeline และ SuperScalar อยู่หลายหน่วยภายใน และยังสนับสนุนการทำงานแบบ Hyper Thread (มีการทำงานจากหลายโปรแกรม อยู่ภายใต้หน่วยประมวลผลพร้อมกัน) เพื่อเพิ่มการใช้งานหน่วยประมวลผลให้คุ้มค่า เช่น Intel Core i3 รุ่น 7320³⁹ จะมีหน่วยประมวลผลอยู่ 2 หน่วย แต่รองรับการทำงานได้จาก 4 โปรแกรมพร้อมกัน จึงมีการทำงานที่เทียบเท่ากับสถาปัตยกรรมแบบ SuperScalar ที่มีเดิร์กี 4

มีสถาปัตยกรรมสมัยใหม่และเทคโนโลยีการแปลงภาษาอยู่อีกมากมาย ซึ่งมีได้ยกตัวอย่างหรือกล่าวถึงทั้งหมดในที่นี้ ขอให้ผู้อ่านศึกษาเพิ่มเติมตามองจากแหล่งข้อมูลทั่วไป

38 ตารางละเอียดเพิ่มเติมได้ใน N. P. Jouppi, "The nonuniform distribution of instruction-level and machine parallelism and its effect on performance," IEEE Trans. Comput., vol. 38, no. 12, pp. 1645–1658, Dec. 1989.

39 อ้างอิง จาก <https://www.intel.com/content/www/us/en/products/processors/core/i3-processors/i3-7320.html>

7.7 แบบฝึกหัดท้ายบท

- ในสถาปัตยกรรมแบบเวกเตอร์ ท่านคิดว่าประสิทธิภาพของหน่วยประมวลผลจะเป็นอย่างไร เมื่อเปรียบเทียบกับประสิทธิภาพของหน่วยประมวลผลกลางแบบ pipeline
- สถาปัตยกรรมแบบ SuperScalar และ สถาปัตยกรรมแบบ Very Long Instruction Word ที่ดีกว่ากัน จะมีสมรรถนะที่เหมือนหรือแตกต่างกันอย่างไร จงแสดงการวิเคราะห์
- กำหนดให้คำสั่งที่ทำงานกับเรจิสเตอร์ที่ขึ้นต้นด้วย F ใช้หน่วยประมวลผลย่อยเลขทศนิยม และคำสั่งที่ทำงานกับเรจิสเตอร์ที่ขึ้นต้นด้วย R ใช้หน่วยประมวลผลย่อยเลขจำนวนเต็ม ซึ่งทั้งสองหน่วยสามารถทำงานได้พร้อมกันบนสถาปัตยกรรมแบบ SuperScalar จากโปรแกรมที่กำหนดให้ หากคำสั่ง LD ต้องห่างจากคำสั่ง ADD อย่างน้อย 1 รอบและคำสั่ง ADD ต้องห่างจากคำสั่ง SD อย่างน้อย 2 รอบจึงจะไม่เกิดการ stall จงแสดงการเรียงลำดับโปรแกรมตั้งกล่าวเพื่อให้โปรแกรมทำงานได้เร็วที่สุด ทั้งนี้ขอให้แสดงให้ชัดเจนว่าคำสั่งใดต้องทำงานในส่วนประมวลผลย่อยทางขวาที่ทางเลขทศนิยม และคำสั่งใดต้องทำงานในส่วนประมวลผลย่อยทางซ้ายจำนวนเต็ม

Loop:

```

LW      $F0, 0($R1)
ADD    $F4, $F0, $F2
SW      0($R1), $F4
SUB    $R1, $R1, #8
BNE   $R1, Loop

```

- จากข้อสาม หากสามารถทำ loop unroll ได้ 4 รอบ จะช่วยให้การจัดลำดับเพื่อให้เกิด stall น้อยลงง่ายขึ้นหรือไม่ หากง่ายขึ้น จงแสดงสมรรถนะที่ดีขึ้นเปรียบเทียบกับสมรรถนะของโปรแกรมในข้อสาม
- หากสามารถทำ loop unroll ได้ไม่จำกัด ขนาดของโปรแกรมที่ใหญ่ขึ้นจะเป็นเท่าใด และประสิทธิภาพที่ได้มากขึ้นจะคุ้มค่าหรือไม่ ขอให้แสดงการคำนวณประกอบการวิเคราะห์

6. ให้ผู้เรียนลองแปลภาษาซีของโปรแกรม MyLoop.c ด้วย clang โดยใช้คำสั่ง clang -S -emit-llvm MyLoop.c เพื่อแสดงผลลัพธ์ที่เป็น LLVM IR และลองทำความเข้าใจผลลัพธ์ดู (clang มักมีอยู่ในระบบปฏิบัติการ Linux และ Mac OS X กรณี Windows ให้ติดตั้งผ่าน cygwin⁴⁰)

```
/* MyLoop.c */
#include <stdio.h>

int main (int argc, char *argv[] ) {
    int i,j;
    int a[100][100];
    int c;
    for(i=0;i<10;i++) {
        for(j=0;j<10;j++) {
            c=10;
            a[i][j]=i*j +c;
        }
    }
}
```

40 ดูรายละเอียดเพิ่มเติมที่ <http://www.cygwin.com/>

8 การจัดการหน่วยความจำ

ปัญหาหลักของระบบหน่วยความจำ แบ่งออกได้เป็นสองลักษณะคือ 1) ปัญหารี่องสมรรถนะ หรือ ความเร็ว และ 2) ปัญหารี่องขนาดของหน่วยความจำที่สามารถใช้งานได้ การปรับปรุงอย่างใดอย่างหนึ่งในสองอย่างนี้ มักส่งผลเสียต่อคุณสมบัติอีกอย่างหนึ่งเสมอ กล่าวคือ หน่วยความจำขนาดใหญ่จะมีการทำงานที่ช้า ในขณะที่หน่วยความจำที่มีขนาดเล็ก จะมีการทำงานที่รวดเร็ว หากจะทำให้ระบบความคิดถูกขึ้นมาอีกนิด จะมีปัญหาเพิ่มอีกเรื่องหนึ่ง คือ 3) ปัญหารี่องราคา แน่นอนว่า หน่วยความจำที่มีความเร็วสูงมักจะมีราคาแพงตามไปด้วย (ทำให้มีได้น้อย) ส่วนหน่วยความจำที่มีความเร็วต่ำ มักมีราคาถูก (ทำให้มีได้มาก)

เรื่องความเร็วของหน่วยความจำและขนาดความจุนี้ มีคำอธิบายง่าย ๆ ตามหลักการทำงานพิสิกส์ทั่วไป คือ หน่วยความจำขนาดใหญ่เบรียบเสมือนห้องที่เก็บของมาก การจะหาของหรือจัดเก็บของในห้อง ย่อมต้องใช้เวลามากกว่าการหาของหรือจัดเก็บของบนโต๊ะขนาดเล็ก สำหรับหน่วยความจำเก่าเข่นเดียวกัน

จากความไม่สอดคล้องกันของความเร็วและขนาด ความยากและท้าทายของเนื้อหาในส่วนนี้ คือ การทำให้รู้สึกเหมือนกับว่ามีหน่วยความจำที่มีความเร็วสูง มีขนาดใหญ่ และราคาไม่แพง ผู้เรียนอาจจะรู้สึกว่าเป็นการท้าทายที่ดูจะโลก관า แต่หากวิเคราะห์ให้ดี บทเรียนที่ผ่านมาก่อนหน้านี้ ล้วนแต่มองว่าหน่วยความจำมี 4 GB (อ้างอิงได้ด้วยเลขที่อยู่ใน 32 บิต) และ มีความเร็วสูง (อ่านหรือเขียนได้ใน 1 รอบการทำงาน)

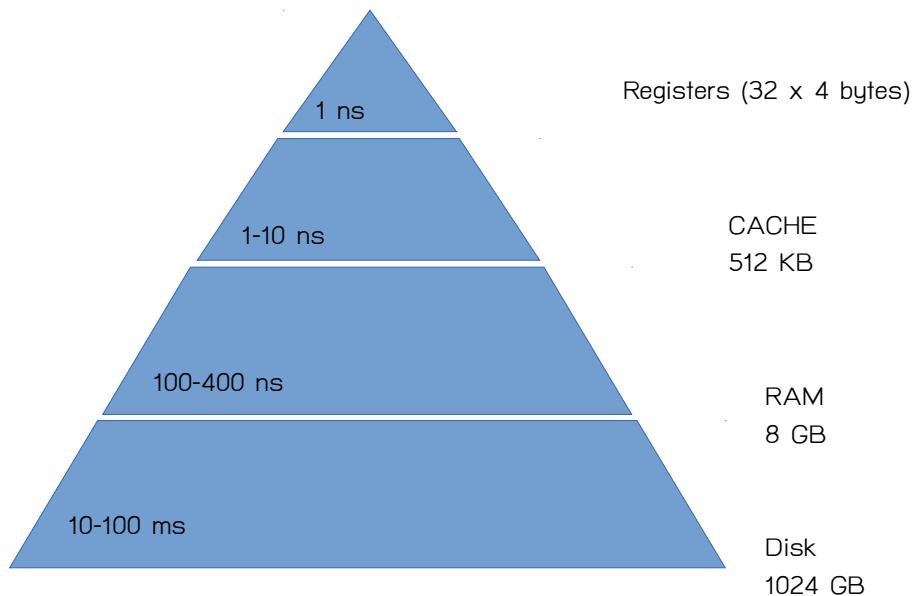
ในทางสถาปัตยกรรม มีแนวทางการแก้ปัญหารี่องความเร็วและขนาดของหน่วยความจำโดยใช้ 2 แนวทางร่วมกันคือ (1) การใช้ระบบ cache ซึ่งเป็นการนำหลักการทำงานท้องถิ่นมาใช้ เพื่อให้รู้สึกว่ามีหน่วยความจำที่เร็ว และ (2) การใช้ระบบหน่วยความจำเสมือน เพื่อให้รู้สึกเสมือนว่ามีหน่วยความจำอยู่เยอะ กล่าวคือ ระบบ cache เป็นการเพิ่มสมรรถนะในการเข้าถึงหน่วยความจำให้กับหน่วยประมวลผลกลาง (โดยเพิ่มความเร็วในการอ่านข้อมูลจากหน่วยความจำ) ส่วนหน่วยความจำเสมือนนั้นช่วยให้การอ้างอิงหน่วยความจำของผู้พัฒนาซอฟต์แวร์และระบบ硬件เป็นอิสระจากกัน ทำให้ผู้พัฒนาซอฟต์แวร์รู้สึกอยู่เสมอว่ามีหน่วยความจำอยู่ 4 GB ตลอดเวลา

เนื้อหาในบทนี้จะเน้นที่พื้นฐานการจัดการระบบ cache และระบบหน่วยความจำเสมือนแบบ paging เท่านั้น อย่างไรก็ตามยังมีระบบหน่วยความจำเสมือนอื่นอีก (เช่น ระบบ segmentation) ซึ่งมีได้เน้นในที่นี้ ดังนั้นหากผู้อ่านต้องการศึกษาเรื่อง segmentation ควรจะศึกษาเพิ่มเติมจากเอกสารหรือตำราอื่นด้วย

8.1 Cache

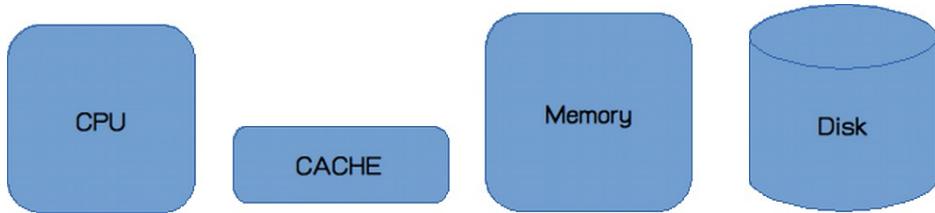
cache นั้นเป็นการเพิ่มสมรรถนะในการอ่านและเขียนข้อมูลจากหน่วยประมวลผลกลางให้สามารถกระทำได้รวดเร็วขึ้น โดยอาศัยหลักการท้องถิน (ซึ่งจะอธิบายต่อไป) คือ การทำให้ข้อมูลที่ต้องการอ่านเป็นประจำนั้นสามารถอ่านได้อย่างรวดเร็ว ด้วยการนำสิ่งที่จำเป็น จะต้องใช้งานบ่อย ๆ มาไว้ใกล้ตัว เช่น กรณีที่จำเป็นจะต้องใช้ปากกาบบ่อย ๆ หากปากกา วางอยู่ในที่ซึ่งเรายิบได้ง่าย การทำงานของเรารู้ความสามารถทำได้รวดเร็วซึ่งหากปากกาอยู่ไกลตัว (เช่นอยู่ในกระเบ้าเอกสาร) จะทำให้การทำงานมีสมรรถนะต่ำอย่าง เพราะต้องเสียเวลา กับการหยิบปากกานาน ในหน่วยประมวลผลกลางก็จะเสียเวลากัน หากการดึงข้อมูลจากหน่วยความจำ (ไม่ว่าจะเป็นการดึงข้อมูลจากหน่วยความจำไปrogram หรือหน่วยความจำข้อมูลก์ตาม) สามารถกระทำได้รวดเร็ว อาจจะช่วยให้หน่วยประมวลผลกลางสามารถทำงานได้ที่ความถี่สัญญาณนาฬิกาที่สูงขึ้น

เนื่องจากหน่วยความจำที่มีความเร็วสูงมีราคาแพง หากระบบคอมพิวเตอร์ทำงานด้วยหน่วยความจำที่มีความเร็วสูงทั้งหมด จะให้เครื่องคอมพิวเตอร์มีราคาสูงมาก แต่หากไม่มีหน่วยความจำความเร็วสูงเหล่านี้อยู่เลย ก็จะทำให้การประมวลผลนั้นช้ามากเกินกัน ด้วยเหตุนี้ ระบบคอมพิวเตอร์จึงควรมีหน่วยความจำความเร็วสูงและความเร็วต่ำผสมกันไปเป็นระดับในปริมาณที่แตกต่างกันไป หน่วยความจำประเภทใดทำงานช้ามักจะราคาถูก ก็มีได้ในปริมาณที่สูง ในขณะที่หน่วยความจำประเภทที่มีความเร็วสูงราคาแพง ก็จะมีในปริมาณน้อย ซึ่ง cache อันเป็นหน่วยความจำความเร็วสูงอาจจะมีเพียง 512 KB ส่วนหน่วยความจำหลัก (RAM) อาจจะมีมากถึง 8 GB และ Hard disk ซึ่งราคาต่อบนหน่วยค่อนข้างต่ำอาจจะมีมากถึง 1024 GB เป็นต้น ความเร็ว และ ปริมาณหน่วยจัดเก็บข้อมูลที่มีในระบบคอมพิวเตอร์ แสดงได้ดังรูปที่ 8.1



รูปที่ 8.1: ความเร็วและปริมาณหน่วยจดเก็บข้อมูลที่มีในระบบคอมพิวเตอร์

ในหน่วยประมวลผลสมัยใหม่ ยังแบ่งการทำงานของ cache ออกเป็นหลายระดับ โดยเรียก ระดับที่อยู่ใกล้เรซิสเตอร์มากที่สุดว่า Level 0 (ซึ่งมักอยู่ในหน่วยประมวลผลกลาง) และ เรียกระดับต่อไปว่า Level 1, Level 2 ออกไปเรื่อยๆ จนถึง RAM ซึ่งในการอ้างอิงข้อมูลใน หน่วยความจำ จะอ้างอิงโดยเริ่มจากการหาข้อมูลใน Cache ระดับที่ใกล้กับหน่วยประมวล ผลกลางก่อน แล้วจึงໄล่ตามระดับออกไป (รูปที่ 8.2)



รูปที่ 8.2: ลำดับการเข้าถึงข้อมูล

อย่างไรก็ตาม การมีหน่วยความจำที่มีความเร็วสูงนั้น อาจจะไม่ทำให้สมรรถนะในการอ่าน และเขียนข้อมูลเร็วขึ้นเสมอไป หากขาดระบบการจัดการที่เหมาะสม ตัวอย่างเช่น หากมีโต๊ะ อุ่ง 1 ตัว ขนาด 1x1 ตารางเมตร (ซึ่งเปรียบเทียบได้กับ cache) และข้อมูลที่ต้องทำงาน

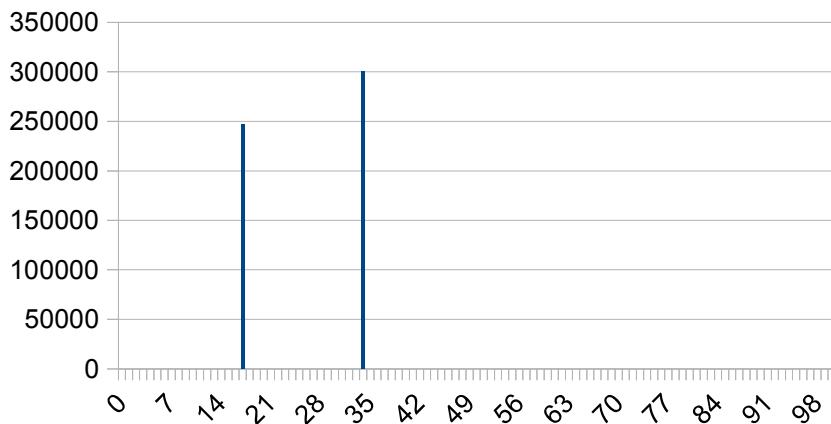
ด้วยเป็นกล่องขนาด 1x1 ตารางเมตร จำนวนหลายกล่อง (เปรียบเทียบได้กับหน่วยความจำหลัก) ซึ่งทำงานของเราระเกียรติข้องกับกล่องหล่ายไป (และตามหลักของ cache นั้น ข้อมูลที่เราต้องการใช้งานต้องอยู่ใน cache ก่อนจึงจะเริ่มอ่านและเขียนข้อมูล) หากวิเคราะห์ให้ดี จะพบว่า เวลาที่เสียในการยกกล่องขึ้นมาวางและออกจากโต๊ะอาจมากกว่าการเดินไปทำงานที่กล่องเหล่านั้นโดยตรง (เช่น กรณีการทำงานของเราต้องนำของจากกล่องที่ 1 มาเก็บในกล่องที่ 2 และนำของจากกล่องที่ 2 มาเก็บในกล่องที่ 1 ตามลำดับ จะพบว่าเมื่อเราต้องการทำงานกับกล่องที่ 1 นั้น กล่องใบที่ 1 ก็ยังไม่อยู่บนโต๊ะ เราต้องเสียเวลายกกล่องไปที่ 1 ขึ้นมาวางบนโต๊ะก่อน และเมื่อเราทำงานกับกล่องที่ 1 เสร็จแล้วต้องการทำกับกล่องใบที่ 2 ก็ต้องเสียเวลายกกล่องใบที่ 1 ลง แล้วยกกล่องใบที่ 2 ขึ้นมาวางแทน จนเมื่อทำงานกับกล่องใบที่ 2 เสร็จ ก็ต้องยกกล่องใบที่ 2 ออก แล้วยกกล่องใบที่ 1 ขึ้นมาวางแทน เป็นต้น) ดังนั้นการมี cache จึงต้องมีขนาดที่เหมาะสม และมีระบบการจัดการที่ดีควบคู่กันไปด้วย

เพื่อให้เข้าใจเรื่องการบริหารจัดการ cache ก่อนอื่นจะขออธิบายถึงหลักการทำงานของหลักการท้องถิ่นก่อน

8.2 หลักการท้องถิ่น (Locality)

หลักการท้องถิ่น คือการทำให้สิ่งที่น่าจะได้ใช้หรือถูกใช้งานบ่อย ๆ มากอยู่ใกล้ตัว ความจริงแล้ว ในชีวิตประจำวันมักพบเห็นการใช้งานของหลักการนี้อยู่ทั่วไป เช่น ผู้ที่ต้องเขียนหนังสือหรือใช้ปากกาบ่อย ๆ มักพกปากกาไว้ที่กระเบ้าเสมอ เพื่อช่วยให้หยิบจับมาเขียนได้อย่างรวดเร็ว หรือ สมุดและหนังสือที่ต้องใช้บ่อย มักวางอยู่บนโต๊ะทำงาน เพื่อให้เข้าถึงข้อมูลได้อย่างทันทีทันใด เป็นต้น ดังนั้น คงจะไม่ต้องขยายความเรื่องความหมายของหลักการท้องถิ่นมากนัก เพราะว่าเป็นเรื่องใกล้ตัว

ในซอฟต์แวร์คอมพิวเตอร์ทั่วไป หากนำค่าของเลขที่อยู่ที่มีการอ้างอิงในโปรแกรมมาทำการวิเคราะห์ จะได้ว่าโปรแกรมได้ ณ จังหวะของเลขที่อยู่ที่มีการใช้งานสูงเพียงส่วนหนึ่งเท่านั้น รูปที่ 8.3 เป็นการนำเลขที่อยู่ของโปรแกรม gcc จากการแปลงโปรแกรมขนาดใหญ่มาทำการวิเคราะห์ และ normalize ให้อยู่ในช่วง 0-99 จะเห็นว่าโปรแกรม gcc ใช้เวลาในการทำงานส่วนใหญ่อยู่เพียง 2 ช่วงเลขที่อยู่เท่านั้น (แกน X คือ เลขที่อยู่ และ แกน Y คือ ค่าความถี่) ดังนั้นหากต้องการให้โปรแกรมนี้ทำงานอย่างรวดเร็ว สมรรถนะสูง เพียงแค่จัดหา cache ที่มีขนาดใหญ่พอสำหรับเก็บข้อมูลในสองช่วงนี้ไว้ให้เข้าถึงได้อย่างรวดเร็วก็พอแล้ว ไม่จำเป็นต้องมีหน่วยความจำความเร็วสูง ราคาแพงมากเกินความจำเป็น



รูปที่ 8.3: ความถี่ของการอ้างอิงช่วงเลขที่อยู่ของ gcc

แม้ในรูปจะเป็นเพียงตัวอย่างจากโปรแกรม gcc ก็ตาม หากทำการทดลองหรือสังเกตเพิ่มเติมจะพบว่า ซอฟต์แวร์ทั่วไปก็มักจะมีการอ้างอิงถึงข้อมูลทั้งสองรูปแบบนี้เสมอ เช่น เพื่อมีการใช้ตัวแปร *a* มักจะมีแนวโน้มว่า จะต้องอ้างอิงถึงตัวแปร *a* อีกในอนาคตอันใกล้ หรือกรณีเป็นข้อมูลแบบ array เมื่ออ้างอิง *b[10]* แนวโน้มคือ อาจจะอ้างอิงถึง *b[11]* ต่อไป ด้วยเหตุนี้การบริหารจัดการ cache ที่ดี จะช่วยให้ข้อมูลที่มีการอ้างอิงทั้งสองรูปแบบสามารถเข้าถึงได้อย่างมีสมรรถนะ

อย่างไรก็ตาม คำจำกัดความของหลักการท้องถิ่น สามารถแบ่งออกได้เป็น 2 แบบย่อยคือ

1. หลักการท้องถิ่นเชิงเวลา (Temporal Locality) กล่าวคือ สิ่งที่ถูกใช้งานบ่อย ภายในช่วงเวลาล่าสุด ควรจะอยู่ใน locality (เวลาใช้อะไรแล้วมักจะใช้อีก) เช่น ในโทรศัพท์มือถือมักแสดงเบอร์โทรศัพท์ที่เพิ่งมีการโทรใช้งานล่าสุด เพื่อให้สามารถโทรใหม่ได้ง่าย นี้เป็นตัวอย่างหนึ่งของ Temporal Locality
2. หลักการท้องถิ่นเชิงพื้นที่ (Spatial Locality) กล่าวคือ ของบางอย่างมักมีการใช้งานใกล้กัน (ใช้สิ่งเดียวกันให้สิ่งที่ใกล้กับสิ่งนั้นด้วย) เช่น ดินสอที่มักถูกใช้งานคู่กับยางลบ ดังนั้น หากจะหยิบดินสอขึ้นมาวางบนโต๊ะ ก็ควรที่จะหยิบยางลบขึ้นมาด้วย เป็นต้น

ดังนั้นการออกแบบ cache ที่ดี จะต้องคำนึงถึงคุณสมบัติของหลักการท้องถิ่นทั้ง 2 ประการ นี้พร้อมกัน

ประเด็นช่วยจำ เวลาผมสอนหนังสือถึงเรื่องนี้ ผมชอบยกตัวอย่างเพื่อประกอบความเข้าใจ เรื่องหลักการท้องถิ่นเชิงเวลา และ หลักการท้องถิ่นเชิงพื้นที่ ดังนี้ ตัวอย่างหลักการท้องถิ่น เชิงเวลา คือ หากชายหนุ่มเพิ่งโทรศัพท์ติดต่อหญิงสาว หมายความว่า เค้าอาจสนใจตัว เชอ ดังนั้นควรวางแผนของเราว่า ก็จะได้เข้าถึงได้หรือไม่ ใจได้ง่าย แต่หากการ ที่ชายหนุ่มโทรศัพท์ติดต่อหญิงสาว เพราหมายจะใช้เรือเป็นทางผ่านไปยังเพื่อนสาวของเรือ การที่หิบเบอร์ของหญิงสาวขึ้นมา ก็ควรจะหิบเบอร์ของเพื่อนสาวที่ชายหนุ่มหมายปองขึ้น มาด้วย เพื่อจะได้หิบโทรศัพท์หาได้ง่าย

8.3 การจัดการ Cache เป็นชั้น

ก่อนจะทำความเข้าใจการจัดการ cache ขออนิยามศัพท์เทคนิคที่ใช้บ่อย ดังนี้

- **Block** คือ การแบ่งขนาดของหน่วยความจำและ cache เป็นส่วนย่อย ๆ ที่มีขนาด เท่ากัน เพื่อความสะดวกในการจัดการ
- **Hit** หมายถึง ข้อมูลที่เราต้องการใช้งานนั้นอยู่ใน cache แล้ว หากเป็นกรณีการ อ่านข้อมูลจะเรียกว่า read hit และกรณีการเขียนข้อมูลจะเรียกว่า write hit
- **Hit Rate** หมายถึง อัตราการ hit ของการอ้างอิงถึงข้อมูล
- **Hit Time** หมายถึง เวลาที่ใช้ในการเข้าถึงข้อมูลกรณี hit มักมีค่าเป็น 1 รอบการ ประมวลผล
- **Miss** หมายถึง ข้อมูลที่เราต้องการจะใช้งานนั้นไม่อยู่ใน Cache เราเรียกการ miss ในกรณีการอ่านข้อมูลว่า read miss และ เรียกการ miss ในกรณีการเขียน ข้อมูลว่า write miss
- **Miss Rate** (หรือ miss ratio) หมายถึง อัตราการ miss ของการอ้างอิงถึง ข้อมูล (มีค่าเป็น $1 - \text{hit rate}$) มักมีหน่วยเป็น ค่า miss ต่อจำนวนคำสั่ง (โอกาส ที่ 1 คำสั่งจะเกิด miss)
- **Miss Penalty** (หากแปลตามคำศัพท์จะเรียกว่า ค่าปรับกรณี fail) หมายถึง เวลาที่ต้องใช้ในการนำข้อมูลจากหน่วยความจำล้ำดับกลับไปเข้ามาไว้ใน cache มัก

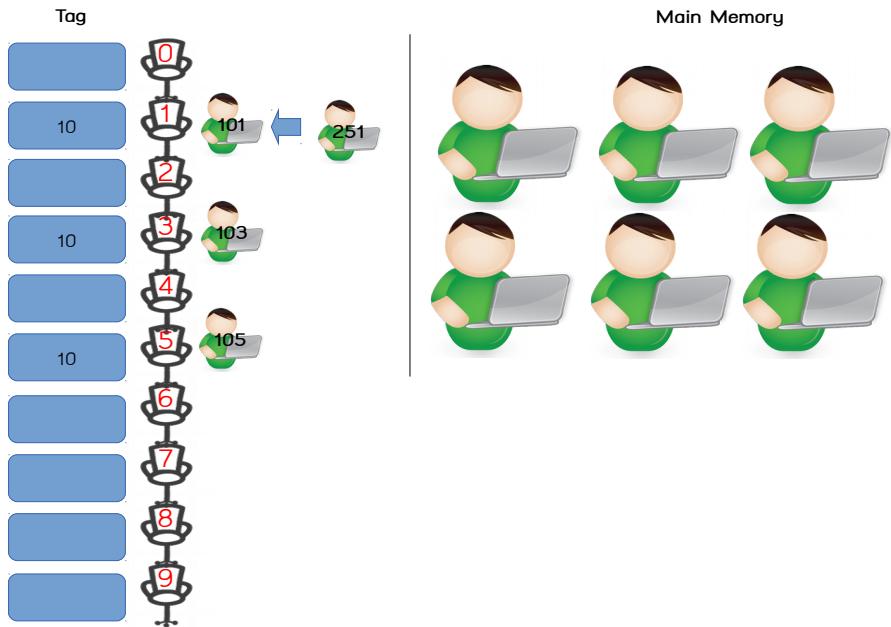
มีหน่วยเป็นจำนวนรอบการประมวลผล

ทั้งนี้การแบ่งหน่วยความจำและ cache ออกเป็น block นั้นช่วยให้เราสามารถนำข้อมูลเข้าและออกจาก cache ได้ง่าย ซึ่งหากเรามองว่า cache เป็นหน่วยความจำใหญ่เพียงก้อนเดียว โอกาสที่จะเกิดการ miss และเวลาในการนำเข้าและเก็บข้อมูลของ cache ย่อมเปลี่ยนแปลงตามขนาดของ cache ทำให้เราจัดการได้ยาก ประกอบกับลักษณะการทำงานของโปรแกรมโดยทั่วไปมักเกี่ยวข้องกับข้อมูลที่มีลักษณะโดด (ไม่ติดกัน) ซึ่งการแบ่งข้อมูลเป็น block นี้ช่วยลดโอกาส miss ของการทำงานกันหน่วยความจำในลักษณะดังกล่าวด้วย ดังจะแสดงได้จากระบบการจัดการ cache แบบ direct mapped ต่อไป

8.3.1 Direct Mapped Cache

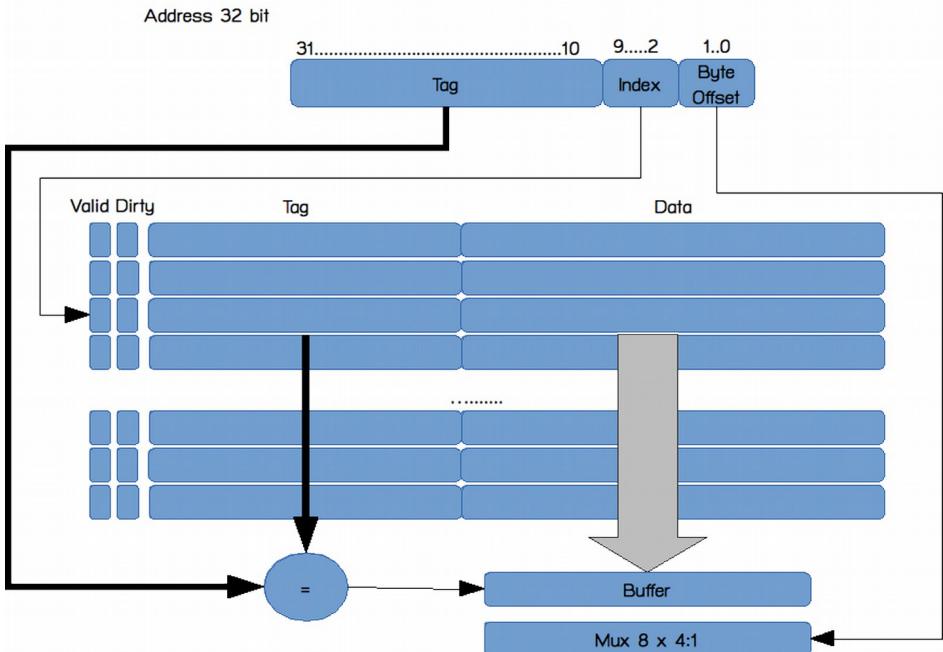
การจัดการกับ cache แบบ direct mapped เป็นระบบการจัดการพื้นฐานที่ง่ายสุด มีลักษณะโดยสรุปคือ เลขที่อยู่ในหน่วยความจำจะมีที่อยู่ใน cache เพียง 1 ตำแหน่งเท่านั้น โดยอยกด้วยตัวอย่างประกอบการอธิบายดังนี้

จากรูปที่ 8.4 หากเปรียบเทียบ cache เป็นสมือนก้าวอีนห้องทำงานซึ่งมีอยู่ 10 ตัว (ติดหมายเลขตั้งแต่ตัวที่ 0 – 9) แต่มีผู้ใช้งาน 1000 คน (แต่ละคนมีเลขประจำตัวของตนเองที่ไม่ซ้ำกัน ตั้งแต่ 000 – 999) โดยสมมุติให้ทั้ง 1000 คนนั่งรออยู่ในห้องโถงขนาดใหญ่ (ในรูปเรียกว่า Main Memory) การเข้ามาหนึ่งในห้องทำงาน ผู้ใช้งานหมายเลข K จะนั่งเก้าอี้หมายเลขที่ $K \bmod 10$ เท่านั้น ดังนั้นมือต้องการทำงานกับคนที่มีเลขประจำตัว 101, 103, 105 จะได้ว่าแต่ละคนจะนั่งเก้าอี้ตัวที่ 1, 3 และ 5 ตามลำดับ หากต้องการทำงานกับคนที่มีเลขประจำตัวเป็น 251 ต่อจากนั้นก็จะต้องให้คนที่มีเลขประจำตัว 101 (นั่งเก้าอี้ตัวที่ 1 อยู่เดิม) ออกไปก่อน แล้วจึงให้คนที่มีเลขประจำตัว 251 นั่งแทนที่ได้ จะสังเกตว่าตำแหน่งใน cache จะขึ้นอยู่กับเลขหลักสุดท้ายของเลขประจำตัว ไม่ว่าจะเป็นคนที่มีเลขประจำตัวเป็น 201, 211, 101, 151 (หรือ อีกหลายเบอร์ที่ลงท้ายด้วย 1) ต่างก็ต้องนั่งที่เก้าอี้ตัวที่ 1 เมื่อนั่น หากพบว่ามีคนนั่งอยู่ที่เก้าอี้ตัวที่ 1 แล้วเราจะทราบได้อย่างไรว่าคน ที่นั่งอยู่นั้นเป็นใคร (มีเลขประจำตัวอะไร) ด้วยเหตุนี้จึงต้องมีการติดป้าย (tag) เพื่อให้ทราบว่าคนที่นั่งอยู่นั้นมีเลขประจำตัวเป็นอะไร การทำงานของ direct mapped cache ก็จะอยู่ในลักษณะเดียวกัน กับตัวอย่างนี้



รูปที่ 8.4: ตัวอย่างการจัดคณเข้าสู่เก้าอี้ในห้องแบบ direct mapped cache

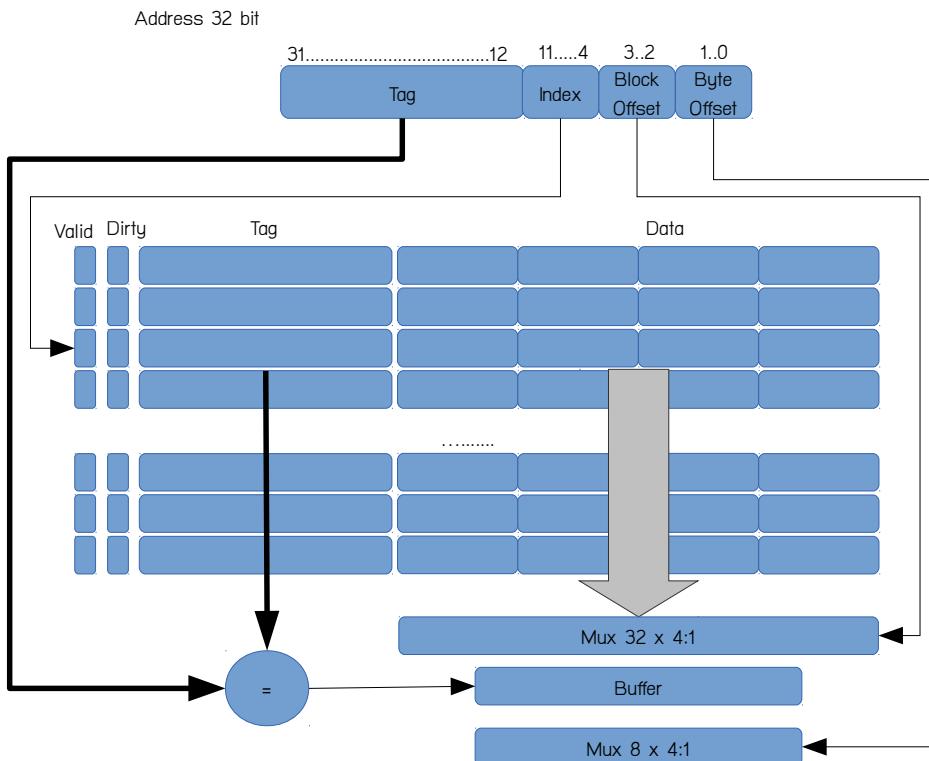
นอกจากนั้นเนื่องจาก cache ก็เป็นหน่วยความจำขนาดหนึ่ง เพื่อแยกแยะให้ทราบว่าข้อมูลในหน่วยความจำนั้นเคยถูกใช้งานมาก่อนหรือไม่ จึงจำเป็นต้องมีการตรวจสอบด้วย valid bit เพื่อแสดงว่าข้อมูลใน cache นั้นมีข้อมูลอยู่หรือไม่ (เพื่อให้ทราบว่าเมื่อใดนั่งอยู่หรือไม่ กรณีที่พบว่ามีป้ายติดอยู่เป็น 000 หรือ กรณีเริ่มต้นระบบ) นอกจากนี้กรณีเป็น cache ของข้อมูล ซึ่งมีการเขียนข้อมูลได้ มักจะมีการเพิ่ม dirty bit เข้าไป เพื่อแสดงว่าข้อมูลใน cache บรรทัด (entry) ดังกล่าวมีการเขียนข้อมูลใหม่ลงไปหรือไม่ หากมีการเขียน (มีการตั้งค่า dirty bit เป็น 1) และแสดงว่ามีการเขียนข้อมูล และต้องมีการนำข้อมูลออกไปเขียนคืนก่อนมีการแทนที่ด้วยข้อมูลชุดใหม่ จากคุณลักษณะดังกล่าวโครงสร้างที่แท้จริงของระบบ cache จึงมีลักษณะดังรูปที่ 8.5



รูปที่ 8.5: โครงสร้าง direct mapped cache

จากภาพจะพบว่าระบบ cache ตั้งกล่าวมี block ขนาด 4 ไบต์ และมี cache ทั้งสิ้น 256 block (อ้างอิงด้วยจำนวน 8 บิต คือ 9..2) รวมกันแล้วคิดเป็น cache ขนาด 1 กิโลไบต์ (Kilobyte) การตรวจสอบว่าข้อมูลที่ต้องการอยู่ใน cache หรือไม่ ทำได้โดยการตรวจป้ายของเลขที่อยู่หน่วยความจำของข้อมูลที่ต้องการอ่านกับ tag ของ cache ในตำแหน่งที่ถูก map ด้วย index หากตรงกัน และ valid บิตเป็นหนึ่งแสดงว่าข้อมูลที่ต้องการอยู่ใน cache (หรือ hit นั่นเอง)

นอกจากนี้ ยังมีการประยุกต์ให้การจัดการข้อมูลในระบบ cache มีการแบ่ง block ที่ละเอียดมากขึ้นโดยอาศัยหลักการห้องถินเชิงพื้นที่ ทำให้ cache 1 บรรทัดมีหลาย block อยู่ช่องเดียวกันโดย block index ตั้งแสดงในรูปที่ 8.6



รูปที่ 8.6: โครงสร้าง direct mapped cache ที่รองรับการทำงานของหลักการห้องถิน์แขิงพื้นที่ จากรูปจะเห็นว่า โครงสร้างดังกล่าวมี block ขนาด 16 ไบต์ โดยแบ่งออกเป็น 4 block ย่อยในแต่ละ block ย่อย มี 4 ไบต์ ทำการเลือกด้วย block offset ขนาด 2 บิต คิดเป็น cache ขนาด 4 กิโลไบต์

8.4 สมรรถนะของ Cache

ตามคำจำกัดความ การที่ cache จะมีสมรรถนะที่ดี ขึ้นอยู่กับว่าข้อมูลที่ต้องการใช้จะมีการ hit มากน้อยเพียงใด หาก hit สูง (miss ต่ำ) หมายถึงข้อมูลที่ต้องการใช้อยู่ใน cache ย่อมใช้งานได้เร็ว มีสมรรถนะดี เนื่องจาก $hit\ rate = 1 - miss\ rate$ ดังนั้นการพูดว่าต้องการลด miss rate หรือเพิ่ม hit rate จึงเป็นเรื่องเดียวกัน

อย่างไรก็ตาม การ hit และการ miss นั้นหมายถึง กรณีที่ข้อมูลตำแหน่งที่หน่วยประมวลผลต้องการใช้งานอยู่ใน cache หรือไม่อยู่ใน cache โดยไม่แยกแยะว่าเป็นการอ่านหรือ การเขียนก็ตาม การ hit และการ miss สำหรับการอ่านและการเขียนข้อมูลนั้น ก็ส่งผลต่อสมรรถนะและระบบจัดการที่แตกต่างกันไป ดังนี้

- **Read Hit** หมายถึง การ hit ข้อมูลใน cache ในกรณีที่หน่วยประมวลผลต้องการอ่านข้อมูลซึ่งสิ่งที่หน่วยประมวลผลกลางจะทำคือ การอ่านข้อมูลจาก cache ไปใช้ประมวลผล
- **Read Miss** หมายถึง การ miss ข้อมูลที่หน่วยประมวลผลต้องการอ่าน (ไม่พบใน cache) ซึ่งสิ่งที่หน่วยประมวลผลต้องการทำคือ ต้องหยุดรอ (จ่ายค่าปรับเป็นเวลา) เพื่อให้ระบบจัดการ cache ดึงข้อมูลจากหน่วยความจำหลักเข้าสู่ cache ก่อน (ตามคำสั่งพัทก์ก่อนหน้านี้ เรายังที่หน่วยประมวลผลต้องหยุดรอนี้เรียกว่า miss penalty) จากนั้นเมื่อระบบจัดการ cache นำข้อมูลจากหน่วยความจำหลักเข้ามาเก็บใน cache เรียบร้อยแล้ว จึงดำเนินการตามปกติเมื่อ read hit ต่อไป
- **Write Hit** หมายถึง การ hit ข้อมูลที่หน่วยประมวลผลต้องการเขียนใน cache ซึ่งมีแนวทางในการปฏิบัติได้ 2 ลักษณะ (ทั้งนี้ขึ้นอยู่กับผู้ออกแบบ) คือ (1) การ write back โดยการเขียนข้อมูลลงภายใน cache เท่านั้น และ (2) การ write through คือ การเขียนข้อมูลลงภายใน cache และ หน่วยความจำหลักพร้อมกัน
- **Write Miss** หมายถึง การ miss ข้อมูลที่หน่วยประมวลผลต้องการเขียนใน cache ซึ่งหน่วยประมวลผลกลางจะมี 2 ทางเลือกคือ (1) หยุดรอเพื่อให้ระบบจัดการ cache นำข้อมูลจากหน่วยความจำหลักเข้ามาเก็บใน cache ก่อน จากนั้น จึงทำการประมวลผลในลักษณะเดียวกับการ write hit ข้างต้นต่อไป ลักษณะนี้เรียกว่า write allocate และ (2) ทำการเขียนข้อมูลต่อไปเลย เมื่อไอนกิดการเขียนตามปกติ (เพราะไม่มีเหตุผลที่จะต้องนำข้อมูลเข้ามา เพื่อเขียนทับ) ลักษณะนี้เรียกว่า write not allocate

ในกรณีของการ write hit จะเห็นว่า ทั้ง 2 แนวทางในการจัดการนี้ มีข้อดีข้อเสียต่างกัน และส่งผลต่อสมรรถนะต่างกัน กล่าวคือ write back นั้นจะทำงานได้เร็วกว่า เพราะจะเขียนข้อมูลที่ต้องการลงใน cache และข้อมูลใน cache จะไม่ตรงกับข้อมูลในหน่วยความจำหลัก และเมื่อต้องนำข้อมูลใน block นั้นออกจาก cache จึงจะทำการเขียนข้อมูลดังกล่าวคืนไปยังหน่วยความจำหลัก สำหรับกรณีของการ write through⁴¹ หน่วยประมวลผลกลางจะใช้เวลา

41 ข้อสังเกต การเลือกการทำงานแบบ write through ในโครงสร้างของ cache (รูปที่ 8.5) จะไม่จำเป็น

ในการทำงานมากกว่าคือ จะต้องมีการเขียนข้อมูลทั้งใน cache และหน่วยความจำหลัก ซึ่งจะทำให้ข้อมูลในหน่วยความจำหลักมีค่าตรงกับค่าของข้อมูลใน cache เสมอ ด้วย ลักษณะดังกล่าว ระบบ write back จึงไม่เหมาะสมที่จะใช้งานบนระบบที่หน่วยประมวลผล กลางหลายตัว

ในทำนองเดียวกัน ในกรณีของการ write miss การตัดสินใจว่าจะทำ write allocate หรือ write not allocate ย่อมส่งผลต่อสมรรถนะเขียนเดียวกัน อย่างไรก็ตาม บ่อยครั้งที่สถาปนิก (ผู้ออกแบบ) มักเลือกใช้ระบบ write allocate เพราะเวลาเขียนข้อมูลอาจมีได้เจียนเต็มทุก block ของ cache ดังนั้น หากไม่นำข้อมูลเข้ามา ก่อน อาจทำให้ข้อมูลใน cache ผิดพลาด ไปได้

อย่างไรก็ตาม ในปัจจุบันมีการเสนอสถาปัตยกรรมการจัดการ cache แบบมีระบบ sub block กล่าวคือ มี valid bit และ dirty bit แยกกันในแต่ละ block เพื่อจะได้ทำการ allocate และกัน และไม่มีผลกระทบกรณีเขียนข้อมูลไม่เต็ม block แต่ลักษณะการอักบบ ดังกล่าวอยู่นอกขอบเขตของหนังสือเล่มนี้ หากผู้เรียนสนใจสามารถค้นคว้าเพิ่มเติมได้

จากลักษณะการทำงานของ cache ดังกล่าว จะพบว่า cache ที่มีสมรรถนะนั้นคือ cache ที่มีระบบการจัดการซึ่งทำให้ miss ratio มีค่าต่ำ และเวลาที่ต้องหยุดเพื่อให้ระบบจัดการ cache นำข้อมูลจากหน่วยความจำหลักเข้ามา สู่ cache (miss penalty) มีเวลาอยู่ที่สุด ซึ่งจากลักษณะดังกล่าวทำให้เวลาที่ใช้ในการประมวลผลของหน่วยประมวลผลกลางนั้นมี ความสัมพันธ์กับเวลาที่ต้องหยุดรอเพื่อให้ระบบจัดการ cache ทำงาน ดังแสดงได้ดังสมการ ที่ 8.1

$$\text{Memory Access Time} = \text{Hit Time} + (\text{miss ratio} \times \text{miss penalty})$$

สมการที่ 8.1 Memory Access Time (หน่วย cycle)

ซึ่งหากวิเคราะห์ต่อไป โดยนำสมการดังกล่าวไปผนวกรวมกับสมการเพื่อหาค่า CPU Time จะได้ว่า ค่า memory access time จะถูกผนวกเข้าไปเป็นส่วนหนึ่งของจำนวนรอบการ ประมวลผลต่อคำ สังdam แสดงได้ในสมการที่ 8.2

$$CPI = \text{ideal CPI} + (\text{miss ratio} \times \text{miss penalty})$$

สมการที่ 8.2 ค่า CPI กรณีที่มี cache

ต้องมี dirty bit เนื่องจากข้อมูลที่เขียนจะมีค่าเหมือนกับข้อมูลใน cache เพราะมีการเขียนข้อมูลทั้งสองที่

ทั้งนี้ให้ลองนำค่าจำนวนรอบการประมวลผลต่อคำสั่งจากสมการที่ 8.2 กลับเข้าไปแทนค่าในสมการที่ 2.10 พบว่า เมื่อมองผิวเผิน CPU Time ในกรณีที่มี cache นั้นมากกว่า CPU Time ในกรณีที่ไม่มี เพราะจะมีรอบการ stall ซึ่งเกิดจาก miss ratio x miss penalty เพิ่มขึ้นมา แต่ในความเป็นจริงนั้นการมี cache ช่วยให้ CPU Time มีค่าน้อยลง โดยจะขอเปรียบเทียบดังนี้

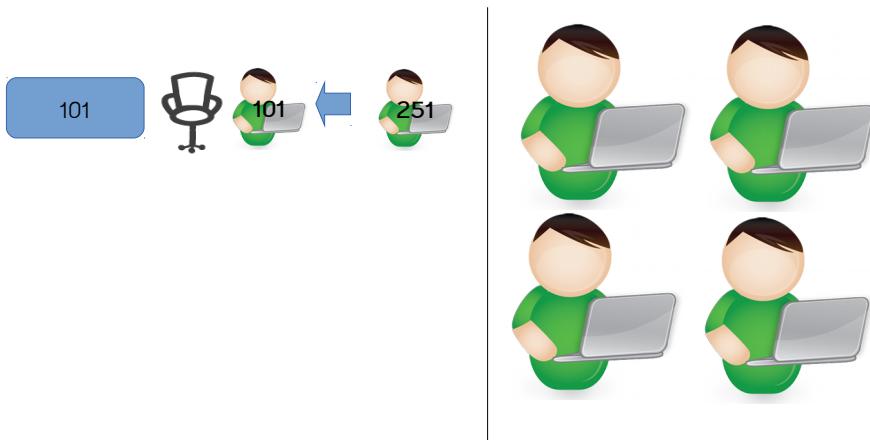
ในกรณีหน่วยประมวลผลกลางแบบ multiple cycle หากไม่มีระบบ cache จะทำให้การเข้าถึงหน่วยความจำทุกครั้ง ต้องเป็นการเข้าถึงหน่วยความจำที่มีความเร็วต่ำ ซึ่งเวลาในการเข้าถึงนี้ จะเทียบได้กับ miss penalty หรือกล่าวว่าได้อีกนัยหนึ่งว่า miss ratio มีค่าเป็น 1 เสมอ (miss ทุกครั้ง เพราะไม่มีพื้นที่ห้องถินให้ค้นหาข้อมูล)

จากสมการข้างต้นพบว่า การปรับปรุงสมรรถนะของระบบ cache นั้นสามารถทำได้โดยการลด miss ratio และ/หรือ ลด miss penalty ตามแนวทางที่จะนำเสนอต่อไปนี้

8.5 การลด Miss Ratio

การลด miss ratio คือ การทำให้ข้อมูลที่หน่วยประมวลผลต้องการใช้ มีโอกาสที่จะถูกพบใน cache มากที่สุด ทั้งนี้ปัจจัยส่วนหนึ่งคือ การกำหนดให้ cache มีขนาดเหมาะสม และมีขนาดของ block size ที่เหมาะสม การจะตอบว่าขนาดของ cache และ block size ที่เหมาะสมเป็นเท่าใดนั้น เป็นสิ่งที่ประเมินได้ยาก เพราะแต่ละโปรแกรมย่อมมีลักษณะของหลักการทำงานที่แตกต่างกันไป จากการทดสอบพบว่า จะมีค่าที่เหมาะสมอยู่ค่าหนึ่ง ๆ สำหรับหน่วยประมวลผลกลางแต่ละตัว ทั้งนี้หาก block size มีขนาดใหญ่เกินไป (เมื่อกำหนดให้ cache มีขนาดเท่าเดิม) ในบางกรณี cache จะจะมี miss ratio มากขึ้นก็ได้ ทั้งนี้เนื่องจากจำนวนที่ลดลงจนไม่เพียงพอสำหรับหลักการทำงานท้องถินเชิงเวลา

เพื่อความเข้าใจ ลองวิเคราะห์ตัวอย่างกรณีสุดขั้วด้านหนึ่งคือ กรณีมี cache เพียง 1 บรรทัด ($\text{block size} = \text{cache size}$) ซึ่งเปรียบเทียบแล้ว จะเหมือนกับว่า เราเมื่อเข้าถึงสำหรับให้คณเข้ามาในห้องเพียงที่เดียว ดังแสดงในรูปที่ 8.7 จากภาพจะเห็นว่าทุกครั้งที่เราต้องทำงาน จะต้องเชิญคณเข้าออกตลอดเวลา จนกล่าวได้ว่าเวลาส่วนใหญ่จะเสียไปกับการให้คณเข้าและออก จนไม่สามารถที่จะทำงานไร้ให้เสร็จได้ ปัญหานี้เรียกว่า ping-pong effect



รูปที่ 8.7: กรณี cache เพียง 1 บรรทัด

เพื่อให้สามารถแก้ปัญหาระเรื่อง miss ratio ได้ถูกต้อง ลองวิเคราะห์ต่ออีกนิดหนึ่งว่า miss เกิดจากสาเหตุใดได้บ้าง เมื่อการ miss เกิดจากการหาข้อมูลแล้วไม่พบข้อมูลที่ต้องการใช้ใน cache ดังนั้น miss จึงเกิดขึ้นได้จาก 3 สาเหตุด้วยกัน ได้แก่

1. **Compulsory Miss** หรือ **Cold Miss** คือ miss ที่เกิดจากการใช้งานครั้งแรก ซึ่งอาจจะไม่สามารถแก้ไขอะไรได้มากนัก (เปรียบเสมือนกับเริ่มมาใหม่โดยทั่วไป ยังไม่ได้หยิบของออกจากกระเบ้าตามตั้งบันโน๊ต ย่อมต้อง miss เพื่อยิบของครั้งแรก) มีแนวทางที่แก้ไขได้ทางหนึ่งคือการทำ prefetch กล่าวคือ หยิบของขึ้นมาไว้ก่อนไม่ว่าจะใช้หรือไม่ก็ตาม (รายละเอียดมีได้ล่าร์วถึงในที่นี้)
2. **Capacity Miss** คือ miss ที่เกิดจาก cache มีขนาดเล็กไป ตัวอย่างเช่น กรณี ping-pong effect ในรูปที่ 8.7 ซึ่งมี cache น้อยเกินไป ไม่พอกับขนาดพื้นที่ท่องถิน ที่จำเป็นต้องใช้ จึงต้องมีการนำข้อมูลเข้าออก วิธีแก้ที่ตรงไปตรงมาที่สุดคือ การเพิ่มขนาดของ cache (cache size⁴²)
3. **Conflict Miss** คือ miss ที่เกิดขึ้นแม้จะยังมีที่วางเหลืออยู่ใน cache แต่เนื่องจากระบบบริหารจัดการ ทำให้ไม่สามารถนำข้อมูลนั้นไปวางยัง cache ที่ว่าง

42 เนื่องจากขนาดของ cache ที่เหมาะสมจะแตกต่างกันไปในแต่ละโปรแกรม ดังนั้นจึงขอทิ้งคำเตือนเรื่องขนาดที่เหมาะสมนี้ให้เป็นแบบฝึกหัดของผู้เรียน

อยู่ได้ ลองวิเคราะห์รูปที่ 8.4 จะเห็นว่าแม้จะมีที่ว่างอยู่ แต่คุณหมายเลข 251 ก็ไม่สามารถนั่งได้ เพราะหมายเลข 251 นั้นถูกกำหนดต้องให้นั่งที่เก้าอี้หมายเลข 1 ซึ่งมีคนนั่งอยู่ก่อนแล้วเท่านั้น

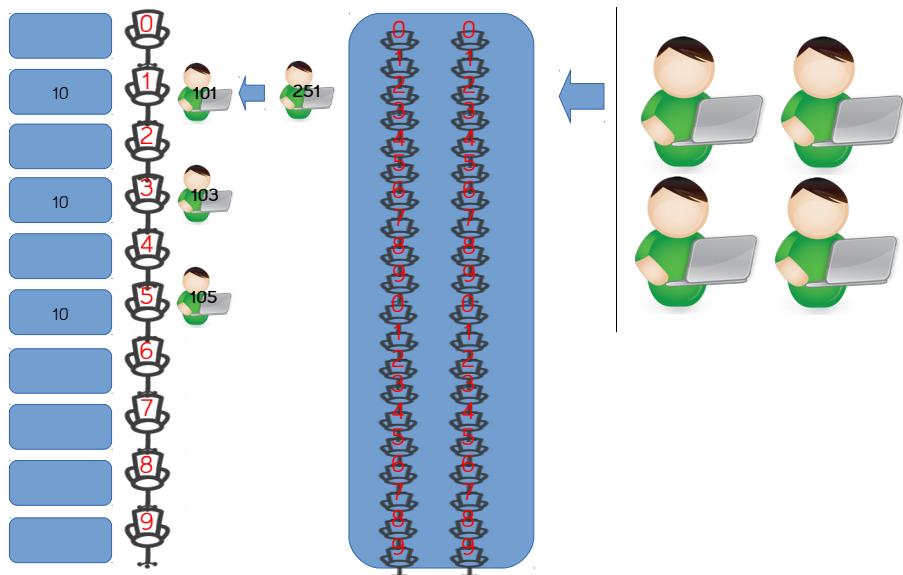
หากสนใจที่ conflict miss เป็นหลัก ปัจจัยที่ส่งผลกระทบ miss ratio ประกอบด้วย

1. Block Size
2. Associativity
3. Replacement Algorithm
4. Write Manage (Write Buffer)

ดังนั้นการลดค่าของ miss ratio ที่ดีที่สุดคือ การเลือกใช้ระบบจัดการ cache ที่มี cache size, block size, replacement algorithm และ associativity ที่เหมาะสม (เมื่อศึกษาเนื้อหาในส่วนต่อไปเรื่อยๆ จะพบว่า การหาค่าที่เหมาะสมเป็นไปได้ยากมาก)

8.6 การลด Miss Penalty

การลด miss penalty คือ การทำให้การโนนถ่ายข้อมูลระหว่าง cache และ หน่วยความจำหลักสามารถทำได้อย่างรวดเร็ว เนื่องจากในทางปฏิบัติเป็นไปได้ยากที่จะทำให้หน่วยความจำขนาดใหญ่มีความเร็วเพิ่มขึ้น ดังนั้นหลักการที่ดีที่สุดในการลด miss penalty คือ การสร้าง cache เพื่อรับรับการทำงานหลายระดับ (multilevel Cache) โดยเมื่อ หน่วยประมวลผลกลางต้องการหาข้อมูลนั้น หาก miss ในระดับแรกอาจจะต้องเสีย miss penalty ถึง 100 รอบเพื่อดึงข้อมูลจากหน่วยความจำหลัก แต่หากมีระบบ cache หลายระดับ เมื่อไม่พบข้อมูลในระดับแรก (miss) อาจจะพบข้อมูลใน cache ระดับถัดไปก็ได้ ทำให้ระบบไม่จำเป็นจะต้องเสีย miss penalty ถึง 100 รอบทุกครั้งที่มีการ miss เกิดขึ้น หากวิเคราะห์ให้ดีจะพบว่า การลด miss penalty โดยการทำ multilevel cache นั้น คือ การพยายามทำให้ข้อมูลบางส่วนสามารถหยิบใช้งานได้เร็วขึ้น มีได้เป็นการลดค่าตัวเลขโดยตรง ลองดูรูปที่ 8.8 ประกอบคำอธิบายดังนี้



รูปที่ 8.8: การทำ multilevel cache เพื่อลด miss penalty

รูปที่ 8.8 เป็นการทำ multilevel cache โดย ให้หันน่ำความจำ (RAM) เปรียบเสมือนคนที่นั่งอยู่ในห้องประชุมขนาดใหญ่จำนวน 1,000 คน กรณีที่ต้องการทำงานกับกลุ่มคนดังกล่าว จะมี cache level 0 อยู่ในห้องทำงานจำนวน 10 ที่นั่ง หากทุกครั้งที่ต้องการเชิญคนจากห้องประชุม เข้ามาในห้องทำงานจะต้องใช้เวลา 100 วินาที หากสามารถทำห้องพักระหว่างกลาง ซึ่งได้ 40 ที่นั่งไว้เป็นที่พักก่อน และการเชิญคนจากห้องพักนี้มายังห้องทำงานจะใช้เวลา 50 วินาที นั่นหมายความว่า มีโอกาสเป็นไปได้ที่ จะพบคนคนนั้นในห้องพัก ช่วยให้สามารถทำงานได้เร็วขึ้น เพราะไม่ใช่ทุกครั้งที่หาไม่เจอกันในห้องทำงาน จะต้องวิ่งไปที่ห้องประชุมเสมอไป ลองคุยกาวิเคราะห์ในเชิงตัวเลขดูในตัวอย่างที่ 8.1

ตัวอย่างที่ 8.1: สมรรถนะของ multilevel cache

จากตัวอย่างในรูปที่ 8.8 สมมุติว่า miss rate ใน level 0 เป็น 30% และ miss rate ใน level 1 เป็น 25% ให้ hit time ของ level 0 เป็น 5 นาโนวินาที และ miss penalty จาก level 0 ไปยัง level 1 เป็น 50ns และ miss penalty จาก level 0 และ level 1 ไปยังห้องประชุม คือ 100ns จงแสดง Speedup ที่ได้ จากการเพิ่ม level 1 เข้าไปยังหน่วยประมวลผลกลาง

กรณีไม่มี Level 1 จะได้ว่า Access Time เป็น

$$Access\ Time_{1level} = 5 + 0.30 \times 100\ (ns)$$

กรณีมี Level 1 จะได้ว่า Access Time เป็น

$$Access\ Time_{2level} = 5 + 0.30 \times (50 + 0.25 \times 100)\ (ns)$$

คิด Speedup จะได้เป็น

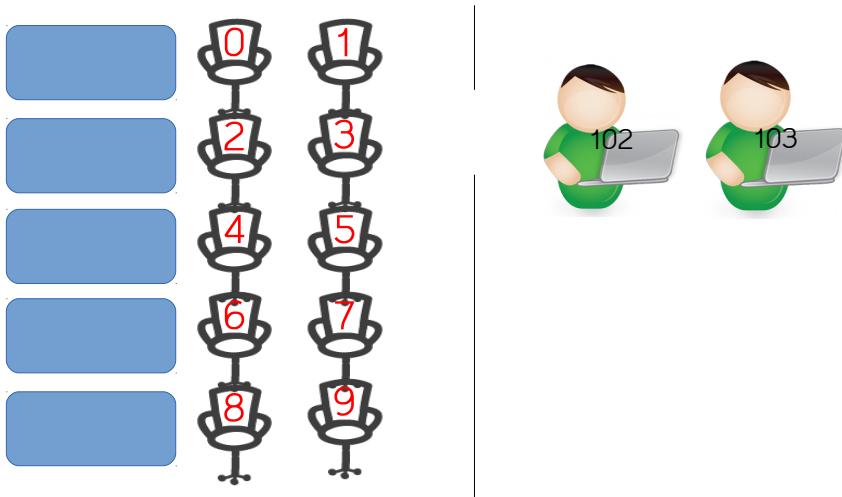
$$Speedup = \frac{Access\ Time_{1level}}{Access\ Time_{2level}} = \frac{5 + 0.30 \times 100}{5 + 0.30 \times (50 + 0.25 \times 100)}$$

$$Speedup = \frac{35}{27.5} = 1.27$$

จากตัวอย่างจะเห็นว่าได้สมรรถนะเพิ่มขึ้นถึง 27% (ลองปรับตัวเลข miss rate จะพบว่าไม่ว่าจะปรับ miss rate เป็นเท่าใด ก็ยังคงได้สมรรถนะที่ดีขึ้นอยู่ดี)

8.7 Block Size

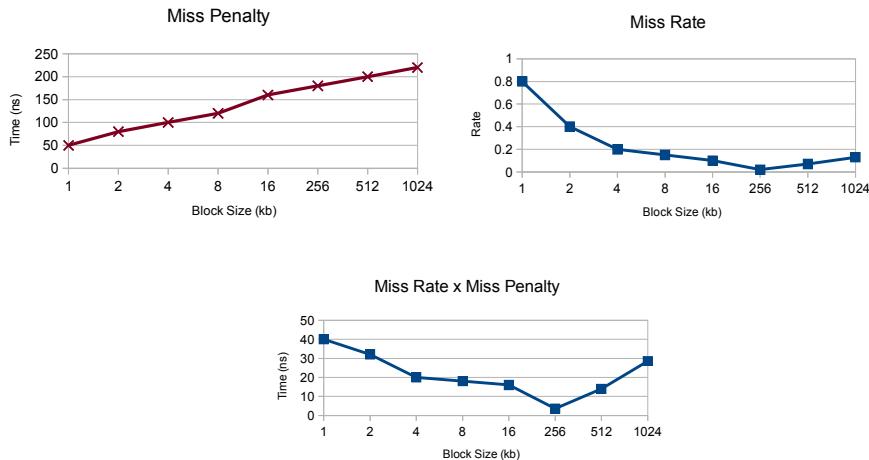
หลักการของ block size ที่มีขนาดใหญ่ขึ้นคือ การรองรับหลักการท้องถิ่นเชิงพื้นที่มากขึ้น แต่ผลในทางตรงข้ามคือ ทุกครั้งที่มีการ miss ระบบอาจจะมี miss penalty ที่มากขึ้นไป ด้วย เนื่องจากในทางปฏิบัติ ทางเดินข้อมูลมีขนาดจำกัด การนำข้อมูลจำนวนมากมา เเข้ามาภายใน ส่งผลให้จะต้องใช้เวลามากขึ้นไปด้วย ลองพิจารณารูปที่ 8.9 จะเห็นว่า มีการเพิ่มขนาดของ block size เป็น 2 แต่เนื่องจากทางเดินข้อมูล (ประตู) ยังคงเข้าได้ทีละ 1 ทำให้ miss penalty มากขึ้น



รูปที่ 8.9: block size ต่อ miss penalty

เมื่อนำผลที่ได้มารวมกัน(ดังตัวอย่างกราฟในรูปที่ 8.10) จะเห็นว่า การเพิ่มค่า block size ไปเรื่อยๆ จะทำให้ miss penalty เพิ่มขึ้นตามไปเรื่อยๆ แต่จะทำให้ miss rate ลดลง เนื่องจากองรับหลักการท้องถิ่นเชิงพื้นที่มากขึ้น จนถึงระดับที่ทำให้หลักการท้องถิ่นเชิงเวลา น้อยเกินไป ค่า miss rate จึงเพิ่มขึ้น แต่เนื่องจาก เวลาในการเข้าถึงโดยรวม เป็นผลคูณ ของ miss rate และ miss penalty ดังนั้น การเลือกขนาดของ block size ที่เหมาะสมจึงควรใช้ความระมัดระวัง

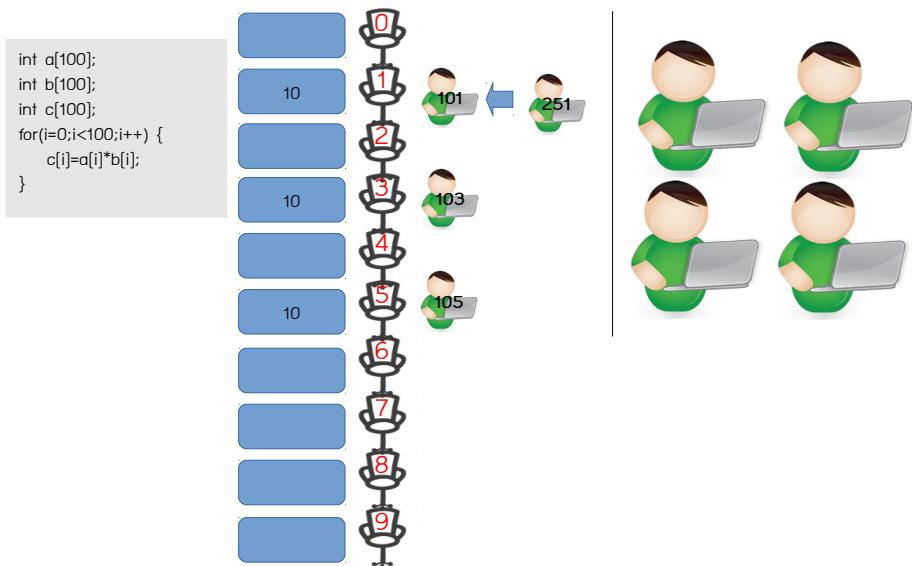
หมายเหตุ การออกแบบในปัจจุบัน จะใช้การทดสอบใน simulator เพื่อตรวจสอบผลลัพธ์ ก่อนการสร้างจริง ทั้งนี้ค่าที่แสดงในรูปที่ 8.10 เป็นเพียงค่าสมมุติเพื่อประกอบการอธิบาย เท่านั้น



รูปที่ 8.10: กราฟความสัมพันธ์ระหว่าง block size, miss rate และ miss penalty

8.8 Associativity

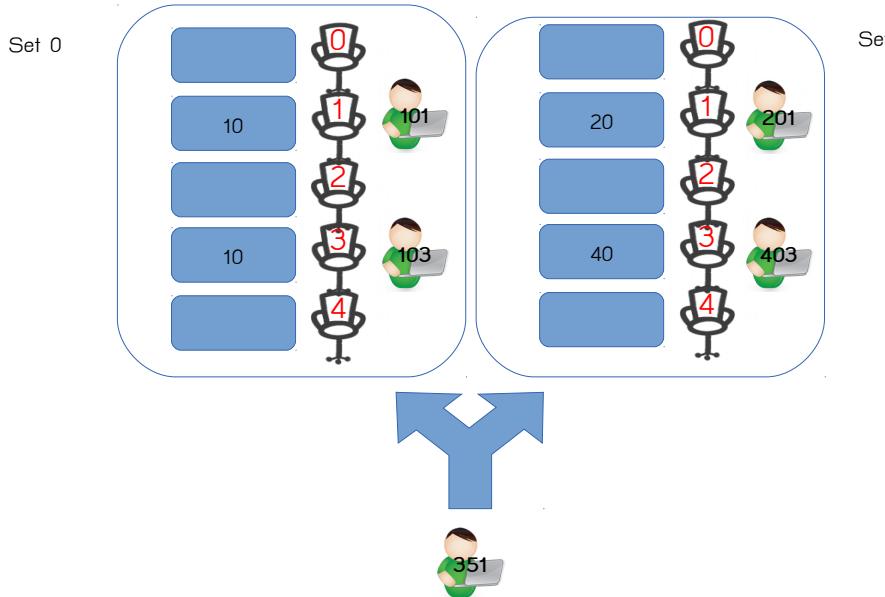
เบื้องต้นเราได้ศึกษาถึงระบบการจัดการ cache แบบ direct mapped ซึ่งอาจเรียกว่า เป็นการจัดการแบบ one-way set associative cache หรือ การกำหนดให้ 1 เลขที่อยู่ของข้อมูลในหน่วยความจำหลักนั้นมีที่อยู่ใน cache ได้เพียง 1 ที่เท่านั้น ปัญหาที่พบคือ มีโอกาสที่จะเกิด miss ratio จาก conflict miss สูงมาก รูปที่ 8.11 แสดงตัวอย่างการเกิด conflict miss



รูปที่ 8.11: ตัวอย่างการเกิด conflict miss ใน 1-way set associative cache

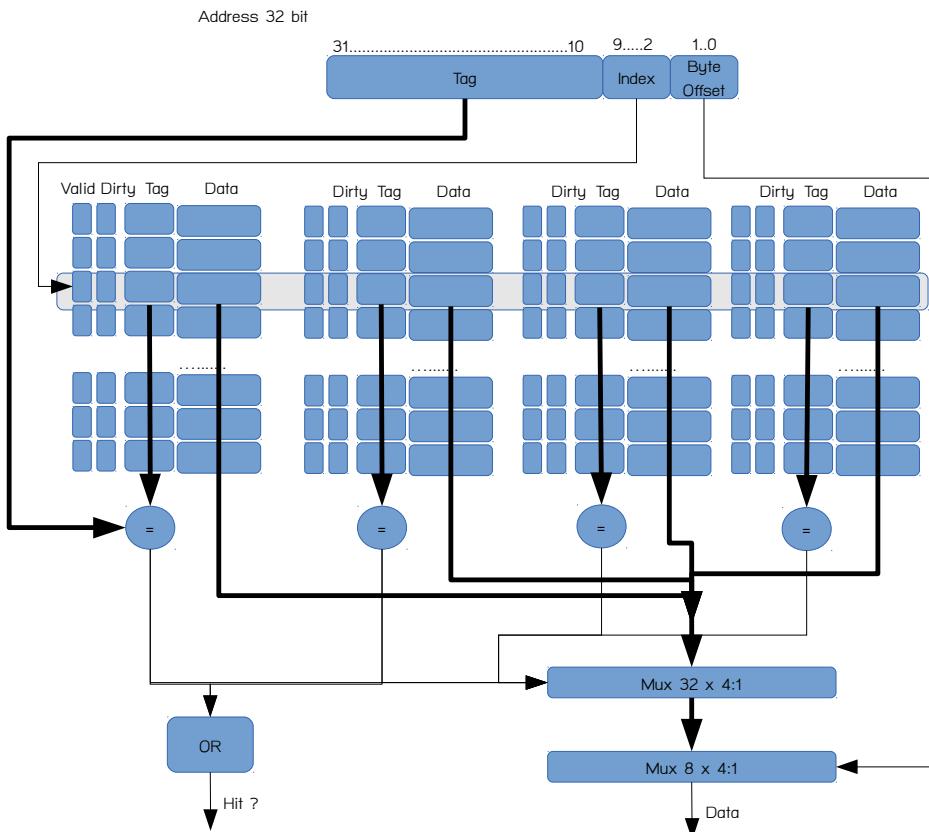
จากรูป หาก a , b และ c เริ่มเก็บข้อมูลที่ตำแหน่ง 100, 500 และ 900 ตามลำดับ จะเห็นว่า ทุกครั้งที่มีการอ้างอิงข้อมูล $a[i]$, $b[i]$ และ $c[i]$ จะเกิด conflict miss ขึ้นทุกครั้ง เพราะทั้ง 3 ตัวแปรจะอ้างอิงลงที่ตำแหน่งเดียวกันใน cache เสมอ ดังนั้น หากสามารถทำการสร้างให้มีทางเลือกมากขึ้น (associativity ที่สูงขึ้น) โอกาสจะเกิด conflict ก็ลดลง

รูปที่ 8.12 เป็นการแก้ปัญหาจากรูปที่ 8.11 โดยการเพิ่มทางเลือกเป็นสองทาง (เรียกว่า two-way set associative) จะเห็นได้ว่า cache size โดยรวมมีขนาดเท่าเดิม block size มีขนาดเท่าเดิม แต่แบ่งออกเป็น 2 ชุดย่อย ดังนั้น ข้อมูลชุด 101 และ 201 ซึ่งเดิม จะต้องอยู่ที่เดียวกัน คราวนี้จะอยู่พร้อมกันได้ เพราะมีตัวเลือกให้ 2 ที่ อย่างไรก็ตาม หากมี 301 เข้ามา ก็จะต้องทำการเลือกว่า จะเอาข้อมูลชุดใดออก ซึ่งกล่าวถึงต่อไปในส่วนของ replacement algorithm



รูปที่ 8.12: การเพิ่ม set associative เพื่อลดปัญหา conflict miss

อย่างไรก็ตาม แม้ว่าระบบการจัดการแบบมี associative way มา กกว่า 1 ทางจะช่วยลด miss ratio จากปัญหา conflict miss ได้ดี แต่การจัดการระบบเพื่อตรวจสอบการ hit และ miss ยังยากขึ้นไปด้วย เพื่อประกอบการอธิบาย รูปที่ 8.13 แสดงการทำงานของระบบ cache ในแบบ four-way set associative ซึ่งมีลักษณะคล้ายกับ direct mapped cache จำนวน 4 ชุดต่อหนาน กัน การตรวจสอบการ hit นั้นต้องเปรียบเทียบ tag กับทั้ง 4 block และเมื่อมีการ hit แล้ว ก็จะต้องผ่าน multiplexor เพื่อเลือกข้อมูลที่ต้องการออกมาใช้ จะเห็นได้ว่า วงจรโดยรวมมีขนาดใหญ่และมีความซับซ้อนยิ่งขึ้น เพราะ การ hit อาจจะเกิดจากข้อมูลชุดใดก็ได้ใน 4 ชุด และหากมีการอ้างอิงข้อมูลแบบ byte offset อีก ก็จะต้องรอให้การเลือกข้อมูลจากชุด associative เสร็จก่อน จึงจะทำการเลือกข้อมูลใบต์ที่ต้องการได้ทั้งหมดนี้ ส่งผลให้วงจรโดยรวมทำงานช้าลง ดังนั้น อาจสรุปในเบื้องต้นได้ว่า **associativity** ส่งผลเสียทำให้ **hit time** ใช้เวลามากขึ้น การทดสอบในสถาปัตยกรรมส่วนใหญ่จะเห็นว่า การเพิ่ม associative way ช่วยให้ miss rate มีค่าลดลงในลักษณะเกือบจะเป็นเชิงเส้น



รูปที่ 8.13: โครงสร้างของระบบ cache แบบ 4-way set associative

ทางเลือกสุดต่ำเพื่อทำให้ conflict miss จะเป็น 0 คือ การเลือกที่ได้ที่ว่างอยู่ (เรียกว่า fully associative) กล่าวคือ เมื่อเราเดินเข้ามาในห้องเก้าอี้ตัวไหนว่างอยู่ก็นั่งได้เลย วิธีนี้จะไม่มีปัญหาการแย่งที่นั่งกัน แต่แลกด้วยวงจรที่ซับซ้อนขึ้น เนื่องจากทุกครั้งที่ต้องการข้อมูลใน cache จะต้องทำการเปรียบเทียบ tag ของทุกบรรทัด (รายละเอียดการออกแบบขอให้ผู้เรียนฝึกทำจากแบบฝึกหัดท้ายบท)

8.9 Replacement Algorithm

อีกปัจจัยหนึ่งที่ส่งผลกระทบต่อ conflict miss คือการเลือกบรรทัดที่ต้องนำออก กรณีเกิด conflict ซึ่งจะมีผลก่อให้เมื่อมี associativity มากกว่า 1 (เนื่องจาก direct mapped cache มีที่สำหรับอ้างอิงเพียงหนึ่งที่ ดังนั้นจึงไม่สามารถมีตัวเลือกให้เลือกได้)

ปัจจุบันมีอักษรที่มีชื่อว่า replacement อยู่หลายรูปแบบ แต่โดยสรุปมักจะเป็นรูปแบบที่คล้ายคลึงหรือตัดแปลงมาจากสองรูปแบบหลักคือ Least Recently Used และ Round Robin Replacement เป็นหลัก ดังนี้

Least Recently Used (นิยมเรียกโดยย่อว่า LRU) เป็น replacement algorithm ที่ใช้การนับว่า ข้อมูลใน cache ตัวไหนมีการใช้งานล่าสุดน้อยที่สุด ตัวนั้นควรจะเป็นเหยื่อ (victim) ที่จะต้องถูกเลือกออกไปจาก cache ดังนั้นในการทำ LRU จะมีต้องมีการเก็บอายุของ cache โดยทุกครั้งที่มีการอ้างอิงถึงข้อมูลตัวใดตัวหนึ่ง อายุของ cache ตัวอื่นจะต้องเพิ่มขึ้นด้วย ด้วยเหตุนี้การทำ LRU จึงต้องมีบิตสำหรับเก็บอายุของ cache⁴³ เพิ่มขึ้น ทำให้การทำงานของ cache ยิ่งซับซ้อนยิ่งขึ้น โดยเฉพาะในกรณีที่มี associativity หมายทางดังนั้นสถาปัตยกรรมหลายแบบจึงเลือกใช้ approximate LRU แบบต่าง ๆ เพื่อลดความซับซ้อนในทางฮาร์ดแวร์แทน

Round Robin⁴⁴ ด้วยความซับซ้อนทางฮาร์ดแวร์ อีกแนวทางหนึ่งที่นิยมใช้กันคือ การเรียน เทียน ก่าวรีคือ ทุกครั้งที่เกิด conflict และต้องการ victim ในครั้งแรก ก็จะทำการเลือก ข้อมูลใน set ที่ 1 เป็น victim ในครั้งต่อไปที่เกิด conflict ก็จะเลือก victim เป็นข้อมูลใน set ที่ 2 วิธีการนี้ช่วยให้ฮาร์ดแวร์มีความซับซ้อนน้อยลง

ตารางที่ 8.1 (ข้อมูลจาก [2]) เปรียบเทียบสมรรถนะการทำงานของ LRU และ RR เมื่อมีโครงสร้าง cache เมื่อนอกจากตารางจะเห็นว่า ค่าที่ได้มีความใกล้เคียงกัน โดย LRU จะให้ผลที่ดีกว่า (ด้วยฮาร์ดแวร์ที่ซับซ้อนกว่ามาก)

43 เพื่อให้เห็นถึงความซับซ้อน ขอแนะนำให้ผู้เรียน ลองทำความเข้าใจ LRU และลองเขียนโปรแกรมง่าย ๆ เพื่อจำลองการทำงานของ LRU cache ดู

44 Round Robin เป็น Replacement Algorithm ที่นิยมใช้ในสถาปัตยกรรม ARM ซึ่งจากการทดสอบพบว่าให้ผลใกล้เคียงหรือไม่ด้อยไปกว่า LRU มากนัก

ตารางที่ 8.1: สมรรถนะของ replacement algorithm แบบ LRU และ RR

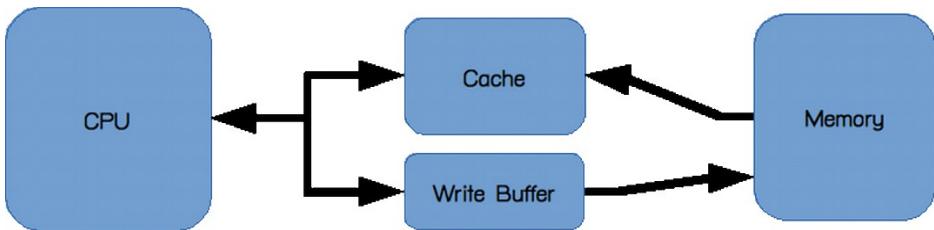
Rep.	bench	Data	
		Miss	Miss rate
RR	gcc	36805	0.018
LRU	gcc	33126	0.017
RR	go	7255	0.005
LRU	go	9073	0.006

ในภาคผนวก ค.2 เป็นกิจกรรมการจำลองการทำงานของ cache ที่มีโครงสร้างแตกต่างกัน ขอให้ผู้เรียนทดลองศึกษาเพิ่มเติมด้วย

8.10 การจัดการ Cache กรณีการเขียนข้อมูล (Write Management)

ตามที่ได้เกริ่นไว้แล้วก่อนหน้านี้ว่า การเขียนข้อมูลสามารถเลือกได้ว่าจะทำ Write Back หรือ write through และเลือกได้ว่าจะทำ write allocate หรือ write not allocate ซึ่งได้อธิบายไว้ก่อนหน้านี้ว่า write through เป็นทางเลือกที่เหมาะสมมากกว่า ในการนี้ที่เป็นการประมวลผลแบบ parallel หรือมีหน่วยประมวลผลหลายหน่วย (รวมถึง multicore) ทำงานกับหน่วยความจำชุดเดียวกัน เพื่อสามารถที่จะบริหารจัดการได้่ายากกว่า ในทำงานองเดียวกับการเลือก write allocate จะช่วยให้การบริหารจัดการง่ายกว่า กรณีที่การเขียนเกิดขึ้นเพียงบางส่วนของ block ใน cache

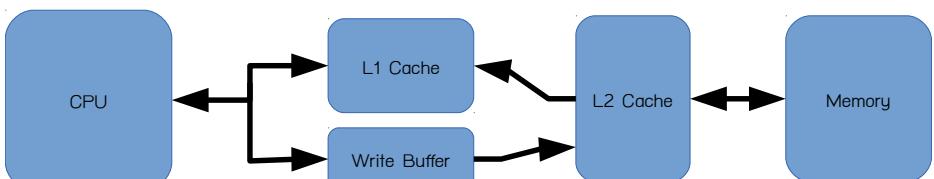
คราวนี้สิ่งที่เป็นปัญหาใหญ่ที่ต้องสมรรถนะของ cache คือ การทำ write through นั้น จะทำงานได้ช้ากว่า write back เพราะจะต้องเขียนข้อมูลขึ้นทั้งสองแห่งคือ ใน cache เองและในหน่วยความจำหลักตั้งนั้น เพื่อให้การเขียนข้อมูลแบบ write through เร็วขึ้น จึงมีผู้เสนอให้ทำ write buffer เพื่อเป็นวงจรพิเศษสำหรับช่วยเขียนข้อมูลคืนไปยังหน่วยความจำหลัก รายละเอียดสามารถแสดงได้ในรูปที่ 8.14



รูปที่ 8.14: การใช้ Write Buffer เพื่อลดเวลาในการเขียนข้อมูลแบบ Write Through

โดยหลักการคือ ทุกครั้งที่มีการเขียน ให้เขียนข้อมูลลงที่ cache แล้วฝาก write buffer ให้ นำข้อมูลไปเขียนคืนที่หน่วยความจำหลักให้ภายหลัง แต่ปัญหาที่ตามมาคือ ไม่ว่า write buffer จะมีขนาดเท่าใด ก็จะพบว่า write buffer จะเต็ม (saturate) ทำงานไม่ทันการเขียน จากหน่วยประมวลผลกลางเสมอ ทั้งนี้เนื่องจาก การสื่อสารระหว่างหน่วยประมวลผลกลางและ cache มีความเร็วสูงกว่าการสื่อสารระหว่าง cache และหน่วยความจำมาก

ดังนั้นเพื่อยังคงสามารถใช้ write through และแก้ (บรรเทา) ปัญหาดังกล่าว จึงใส่ multilevel cache เพิ่มเข้าไป กล่าวคือแบ่งช่วงการทำงานโดยไม่จำเป็นต้องให้ write buffer เขียนข้อมูลไปยังหน่วยความจำหลักโดยตรง เพราะจะใช้เวลามากเกินไป แต่ให้ write buffer เขียนข้อมูลลงใน cache L2 ซึ่งทำงานเร็วกว่า write buffer ก็มีโอกาสที่จะเต็มช้าลง (ดูรูปที่ 8.15 ประกอบคำอธิบาย)



รูปที่ 8.15: การเพิ่ม level ของ cache เพื่อบรรเทาปัญหา saturate ของ write buffer

ประเด็นช่วยจำ ลองจินตนาการถาวร หากต้องขนของบนโต๊ะทำงาน 40 กล่อง ไปจิ้นรถบรรทุก หากคนหนึ่งมีหน้าที่ยกของลงจากโต๊ะ (เขียนข้อมูลใส่ cache และ write buffer) และอีกคนหนึ่งมีหน้าที่นำกล่องจาก write buffer ใส่รถเข็นลงลิฟต์ไปใส่ในรถบรรทุก ต่อให้มีคนนำของใส่รถเข็นแล้ววิ่งไปที่รถบรรทุก 10 คน ก็คงจะทำงานไม่ทันคนที่ยกกล่องลงจากโต๊ะทำงานอยู่ดี เพราะการทำงานของส่วนแรกและส่วนที่สองใช้เวลาต่างกันมาก แนวการแก้ปัญหาคือ การแบ่งระยะเวลาให้สั้นลง และมีคนทำงานเป็นช่วง ก็จะช่วยลดปัญหาการทำงานไม่ทันหรือ saturate ลงได้

8.11 โปรแกรมและสมรรถนะของ Cache

ในทางปฏิบัติ โปรแกรมจะได้อ่านสัญญาณสมรรถนะจาก cache อยู่แล้ว โดยไม่จำเป็นต้องปรับปรุงอะไร กล่าวคือ ด้วยโปรแกรมเดิมเพียงเปลี่ยนหน่วยประมวลผลกลางที่มี cache เพิ่มขึ้น โปรแกรมที่ได้ก็จะทำงานเร็วขึ้นโดยอัตโนมัติ

อย่างไรก็ตาม หากผู้พัฒนาซอฟต์แวร์เข้าใจโครงสร้างของ cache ดีพอ ก็สามารถที่จะปรับปรุง code เพื่อให้ได้ประโยชน์จาก cache มากขึ้นได้ โดยจะขอแยกเป็น ส่วนกรณีคือ กรณีการเขียนโปรแกรมเพื่อใช้ประโยชน์จากการหลักการห้องถินเชิงพื้นที่ และ กรณีการเขียนโปรแกรมเพื่อลด conflict miss ดังนี้

8.11.1 การปรับโปรแกรมเพื่อเพิ่ม Hit

เนื่องจากโครงสร้าง cache สมัยใหม่ มักมี block size ที่ใหญ่พอจะเก็บตัวแปรที่อ้างอิงในโปรแกรมทั่วไปได้หลายตัวพร้อมกัน ดังนั้น หากสามารถเขียนการวนรอบ เพื่อให้ได้ประโยชน์จากการทำงานแบบ spatial cache การอ้างอิงข้อมูลจาก array แบบ 2 มิติควรจะอ้างอิงข้อมูลแบบใช้ชี้อ้อมูลที่ละเอียด (แนวนอน) เพราะภาษาโปรแกรมทั่วไปจัดเก็บ array 2 มิติให้ชี้อ้อมูลในแต่ละແควาติดกัน

ตัวอย่างที่ 8.2 แสดงการเขียนโปรแกรมที่ให้ผลเหมือนกัน 2 แบบ อย่างไรก็ตาม หากลองวัดสมรรถนะ จะพบว่าเมื่อ data มีขนาดใหญ่ขึ้น การเขียนแบบ A จะทำงานเร็วกว่า การเขียนแบบ B มาก ใน การเขียนแบบ A จะอ้างอิงถึง $data[i][j]$ และ $data[i][j+1]$ ต่อเนื่องกัน ซึ่งหาก cache มีหลักการห้องถินเชิงพื้นที่มากพอ (block size มีขนาดใหญ่พอ) มักจะทำให้ $data[i][j+1]$ เป็นการ read hit หากมีการ read miss ที่ $data[i][j]$ ในทางตรงกันข้าม การเขียนแบบ B จะอ้างอิง $data[j][i]$ ต่อตัวย $data[j+1][i]$ ซึ่งไม่ได้อยู่ใกล้เคียงกัน ทำให้มีได้ประโยชน์จากพื้นที่แต่อย่างใด

การเขียนแบบ A	การเขียนแบบ B
<pre>int data [100][100]; int sum; for (int r=0; r<100; r++) { for (int c=0; c<100; c++) { sum += data[r][c]; } }</pre>	<pre>int data [100][100]; int sum; for (int c=0; c<100; c++) { for (int r=0; r<100; r++) { sum += data[r][c]; } }</pre>

ตัวอย่างที่ 8.2: การเขียนโปรแกรมเพื่อใช้ประโยชน์จากหลักการท้องถิ่นเชิงพื้นที่

นี่เป็นเพียงตัวอย่างแบบหนึ่งเท่านั้น ยังมีการเขียนโปรแกรมในลักษณะอื่น เพื่อให้ได้ประโยชน์จากพื้นที่ท้องถิ่นอีกซึ่งมีได้กล่าวถึงในที่นี้

8.11.2 การปรับโปรแกรมเพื่อลด Conflict Miss

ในการวางแผนสร้างข้อมูลบางรูปแบบ จะมีโอกาสทำให้เกิด conflict miss ได้ง่าย เช่นการใช้ตัวแปรแบบ array ที่มีความยาวใกล้เคียงกันหลายชุด เพื่อแทนข้อมูล โอกาสที่ $A[i]$, $B[i]$ และ $C[i]$ จะเกิด conflict miss มักจะสูงกว่าการอ้างอิงตัวแปรที่อยู่ติดกัน (เนื่องจากมีหลักการท้องถิ่นเชิงพื้นที่) ดังนั้น หากสามารถปรับเปลี่ยนโครงสร้างข้อมูลได้ จะช่วยลดปัญหา conflict miss ที่เกิดจากลักษณะนี้ได้

ลองพิจารณาตัวอย่างที่ 8.3 จะเห็นว่าการเขียนแบบ Y ให้ผลการทำงานเหมือนการเขียนแบบ X เพียงแต่ในการเขียนแบบ X จะมีโอกาสเกิด conflict miss สูงกว่า สำหรับการเขียนแบบ Y ทำการรวม a, b และ c ไว้ใน structure ช่วยให้เกิดหลักการท้องถิ่นเชิงพื้นที่ของ a, b, c ขณะทำการประมวลผล ซึ่งนอกจากจะเป็นการเลี่ยง conflict miss แล้ว ยังสนับสนุนให้เกิด hit จากการใช้ข้อมูลหลักการท้องถิ่นเชิงพื้นที่ด้วย

การเขียนแบบ X	การเขียนแบบ Y
<pre>int a[100]; int b[100]; int c[100]; for (int i=0; i<100; i++) { c[i] = a[i] + b[i]; }</pre>	<pre>struct Data { int a; int b; int c; } data[100]; for (int i=0; i<100; i++) { data[i].c = data[i].a + data[i].b; }</pre>

ตัวอย่างที่ 8.3: การเขียนโปรแกรมเพื่อเลี่ยงการเกิด conflict miss

ยังมีแนวทางการใช้ data structure อีกหลายแบบที่จะช่วยให้ conflict miss ลดลง รวมถึงเพิ่มหลักการห้องดินนีเชิงพื้นที่ได้ ซึ่งสามารถศึกษาเพิ่มเติมได้ทั่วไป

8.12 หน่วยความจำเสมือน (Virtual Memory)

หน่วยความจำเสมือน ถูกออกแบบมาเพื่อแก้ปัญหาด้านการอ้างอิงหน่วยความจำอย่างน้อย 3 ประการ ได้แก่ 1) การให้ซอฟต์แวร์สามารถอ้างอิงหน่วยความจำได้มากกว่าหน่วยความจำที่มีอยู่จริง (ลดค่าใช้จ่ายด้านอาร์ดแวร์) 2) relocatable คือ การทำให้ซอฟต์แวร์สามารถ load ไปไว้ที่หนาแน่นี้ไม่จำเป็นต้องเป็นที่เดิมเหมือนกับตอนที่พัฒนาซอฟต์แวร์ 3) ระบบ security คือ การทำให้การอ้างอิงหน่วยความจำของแต่ละโปรแกรมเป็นอิสระจากกัน

ความสามารถในการอ้างอิงหน่วยความจำ ในบริมาณมากเกินกว่าหน่วยความจริงที่มีอยู่จริงในระบบเครื่องคอมพิวเตอร์ เกิดขึ้นได้จากการจัดการที่มีการยืมเนื้อที่จากหน่วยจัดเก็บข้อมูลอื่น (เช่น disk) มาทำการสลับค่าข้อมูลบางส่วนออก ทำให้ผู้ใช้รู้สึกเสมือนว่าอ้างอิงข้อมูลได้มากกว่าหน่วยความจำที่มีอยู่จริง

คุณสมบัติแบบ relocatable ตามความหมายดังเดิมคือ การทำให้โปรแกรมดังกล่าวสามารถ load ไปวางที่เลขที่อยู่ใหม่ได้ โดยไม่จำเป็นต้องเป็นเลขที่อยู่เดิมกับตอนที่พัฒนาซอฟต์แวร์ เช่น ในขณะที่พัฒนาโปรแกรมมีการกำหนดค่า เลขที่อยู่ 200 เป็นที่สำหรับเก็บข้อมูลตัวแปร แต่หากขณะที่มีการใช้งาน เลขที่อยู่ 200 มีการใช้เพื่อกำกับข้อมูลอื่น อาจส่งผลให้โปรแกรมดังกล่าวไม่สามารถทำงานได้ การทำ relocation ด้วยหน่วยความจำเสมือน ช่วยแก้ปัญหานี้ได้โดยการทำให้ไม่มีการเปลี่ยนแปลงเลขที่อยู่ระหว่างการพัฒนา และการใช้งานจริง เพราะ เลขที่อยู่ที่อ้างอิงจะเป็นเลขที่อยู่เสมือน (virtual address) เสมอ

ทำให้โปรแกรมสามารถอ้างอิงได้ด้วยเลขที่อยู่เดิมตลอดเวลา

ในเรื่องของ security เป็นกรณีต่อเนื่องจาก relocatable คือ ทำอย่างไรให้หน่วยความจำของโปรแกรม A และ B เป็นอิสระจากกัน กล่าวคือ หน่วยความจำเลขที่อยู่ 100 ที่อ้างอิงในโปรแกรม A จะต้องไม่เกี่ยวข้องกับ หน่วยความจำตำแหน่ง 100 ที่อ้างอิงในโปรแกรม B มิใช่นั้น การทำงานของโปรแกรม B พร้อมโปรแกรม A อาจทำให้การประมวลผลผิดพลาดได้ ด้วยระบบหน่วยความจำเสมือนทำให้เลขที่อยู่ของ A และ B เป็นอิสระจากกัน และแก้ปัญหาโดยปริยาย

8.12.1 การทำงานของหน่วยความจำเสมือน

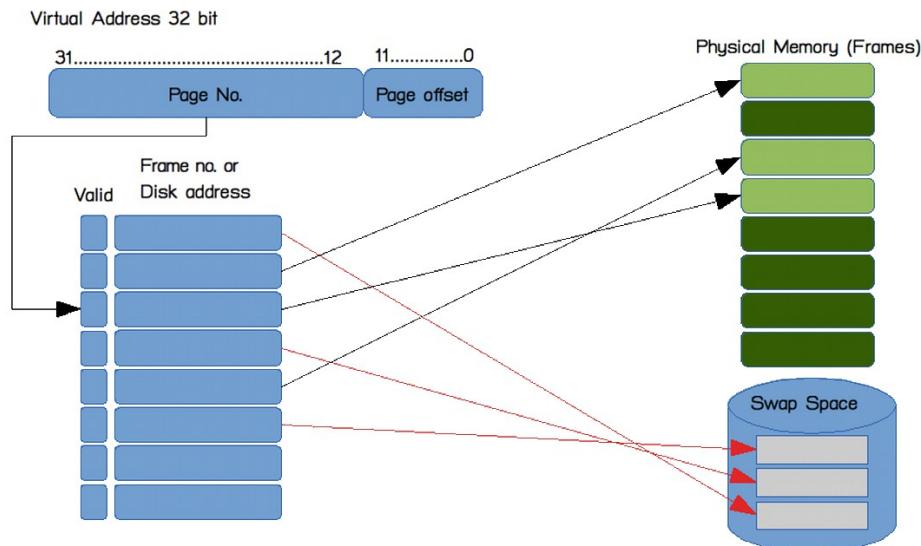
หลักการทำงานของหน่วยความจำเสมือน มีการทำงานที่คล้ายคลึงกับ cache บางส่วน เพียงแต่เปลี่ยนคำเรียกให้แตกต่างกัน เพื่อป้องกันความสับสน ข้อแตกต่างสองประการที่เห็นได้ชัดเจนคือ 1) ในทางโครงสร้าง cache จะเก็บข้อมูลในตารางโดยตรงเนื่องจากข้อมูลมีขนาดเล็กในขณะที่หน่วยความจำเสมือนจะมีเพียงเลขที่อยู่สำหรับอ้างอิงข้อมูลกำหนดอยู่ในตารางเท่านั้น 2) cache ทำงานโดยอาศัยแรร์ทั้งหมด ในขณะที่หน่วยความจำเสมือนจะต้องให้ระบบปฏิบัติการช่วยในการนำข้อมูลเข้าและออกจากระหว่างหน่วยความจำและ storage (disk) ที่ใช้จัดเก็บข้อมูล

เพื่อความสะดวกในการอธิบาย ขอเริ่มอธิบายคำศัพท์ที่แตกต่างกันระหว่างระบบ cache และระบบหน่วยความจำเสมือนโดยมีรายละเอียดดังนี้

- **Frame และ Page** เทียบได้กับ block ในระบบ cache โดย Frame หมายถึง Physical Page หรือ Page ที่มีข้อมูลพร้อมสำหรับการอ้างอิงอยู่ในหน่วยความจำหลัก และ Page หมายถึงข้อมูลอ้างอิงในหน่วยความจำเสมือน
- **Physical Address** คือ เลขที่อยู่สำหรับอ้างอิงหน่วยความจำหลัก
- **Virtual Address** คือ เลขที่อยู่สำหรับอ้างอิงหน่วยความจำที่ใช้ในโปรแกรม
- **Page Table** เทียบได้กับตารางเก็บค่า tag, valid bit และ dirty bit ของ cache เพื่อช่วยในการค้นหาข้อมูลที่ทำหน้าหลักในการแปลง virtual address เป็น physical address
- **Page Number** เทียบได้กับค่า tag ที่ใช้ในการบริหารจัดการ tag

- **Frame Number** (หรือ physical page number) คือค่า page สำหรับอ้างอิง frame ที่อยู่ในหน่วยความจำหลัก
- **Swap Space** อันนี้ จะไม่มีตัวเทียบเคียงใน cache หมายถึงที่จัดเก็บข้อมูลที่ยืมมาจากที่อื่น (โดยทั่วไปมักเป็นฮาร์ดไดส์ก) สำหรับเป็นที่พักข้อมูลที่นำออกจากหน่วยความจำหลัก (ซึ่งจะอธิบายต่อไป)

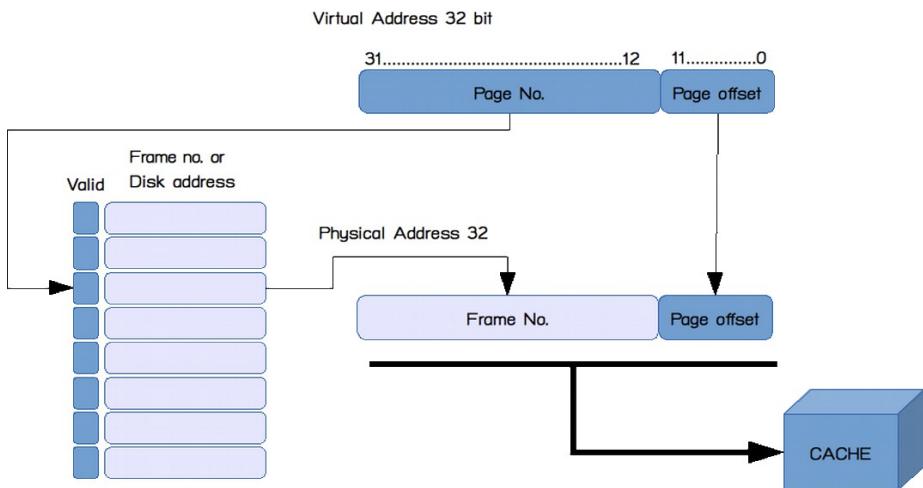
เพื่อให้เห็นภาพการทำงานของหน่วยความจำเสมือน ลองวิเคราะห์การที่โปรแกรมซึ่งต้องการใช้หน่วยความจำเป็นจำนวนมาก แต่สามารถทำงานได้บนเครื่องที่มีหน่วยความจำน้อยໄດ້ (เช่น โปรแกรมที่ต้องการหน่วยความจำ 40 MB ระหว่างการประมวลผล อาจจะทำงานได้บนเครื่องที่มีหน่วยความจำเพียง 16 MB เป็นต้น) ซึ่งการทำงาน สามารถทำได้โดย การใช้งานหน่วยความจำหลักให้ทำหน้าที่คล้ายกับ cache ทุกครั้งที่มีการอ้างอิงข้อมูล และนำข้อมูลทั้งหมดที่ต้องการอ้างอิง (ในกรณีนี้คือ 40 MB) มาตัดเป็นชุดขนาดเท่ากัน (page) ทำการกำหนดเลขหน้า (page number) และเก็บไว้ใน swap space (storage) ทุกครั้งที่ต้องการอ้างอิงข้อมูล (virtual address ที่ต้องการอ่านหรือเขียน) ก็จะเปิด page table เพื่อดูว่าข้อมูลที่ต้องการอยู่ที่ส่วนไหนของ disk (ดูที่ page number) และนำข้อมูลนั้น ขึ้นมาใส่ในของหน่วยความจำหลัก (frame) ซึ่งอ้างอิงได้ด้วย physical address ในการอ้างอิงข้อมูลครั้งต่อไป จะทำการเปิดตารางเพื่อค้นหากรอบว่าข้อมูลอยู่ใน frame หรืออยู่ใน disk หากอยู่ใน frame ก็สามารถอ้างอิงด้วย physical address ที่ได้จาก page table ได้เลย การทำงานสามารถแสดงได้ดังรูปที่ 8.16



รูปที่ 8.16: การทำงานของหน่วยความจำเสมือน

จากรูป จะสังเกตว่าหน่วยความจำเสมือน มีขนาดใหญ่กว่าหน่วยความจำจริงซึ่งบางส่วนจะเก็บอยู่ในหน่วยความจำจริง และ บางส่วนจะเก็บอยู่ใน storage (เครื่องคอมพิวเตอร์ขนาดใหญ่สมรรถนะสูงบางเครื่อง อาจมีหน่วยความจำมากพอ จนไม่ต้องมีการทำ swap เลย ก็ได้) และ valid บิต มีเพื่อแสดงว่าข้อมูลดังกล่าว อยู่ใน swap space หรือหน่วยความจำหลัก (ในระบบจะมีข้อมูลอื่น ๆ เพิ่มเติมจาก valid bit อีก เช่น dirty bit เพื่อบอกว่ามีการเขียนหรือไม่ หรือมี permission เพื่อบอกว่า page ดังกล่าว สามารถใช้เพื่ออะไรได้บ้าง)

หากจะขยายความขั้นตอนการทำงานของการแปลง virtual address เป็น physical address ใน รูปที่ 8.16 จะได้ดังรูปที่ 8.17 ซึ่งมีขั้นตอนการทำงานคือ นำค่า page no. ไปมองหาข้อมูลในบรรทัดที่เกี่ยวข้องเพื่อให้ได้ค่า frame number จากนั้น นำค่าดังกล่าวมารวมกับ offset เพื่อใช้เป็น physical address สำหรับการอ่านหรือเขียนข้อมูลจาก cache ต่อไป



รูปที่ 8.17: การแปลง virtual address เป็น physical address

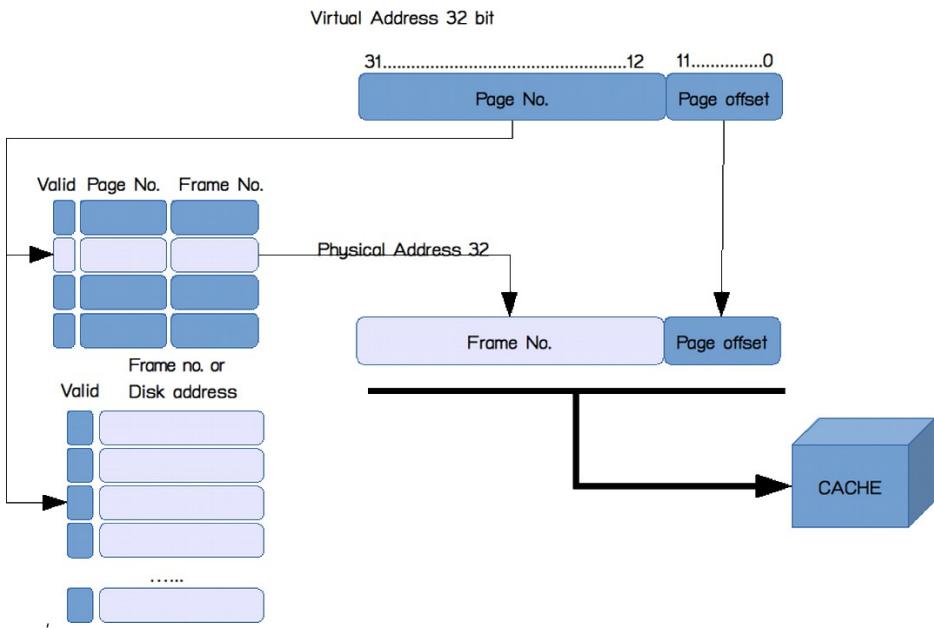
ในการณ์ที่ข้อมูลที่ต้องการใช้งานอยู่ใน swap space หน่วยประมวลผลกลางจะทำการ interrupt เพื่อให้ระบบปฏิบัติการนำข้อมูลเข้าออกจาก frame ก่อน เนื่องจากในขั้นตอนการนำข้อมูลเข้าและออกจากหัวว่างหน่วยความจำจริงและ storage จะต้องถูกจัดการโดยระบบปฏิบัติการ เนื่องจากความต้องการในการใช้หน่วยความจำของแต่ละระบบ และการบริหารจัดการ storage ของระบบปฏิบัติการมีความแตกต่างกัน หน่วยประมวลผลกลางจึงมีเพียงเรจิสเตอร์พิเศษ ที่จะซึ่งไปยังหน่วยความจำเลขที่อยู่ที่เก็บ page table และจะรชี้ที่หน้าที่ตรวจสอบว่าข้อมูลที่ต้องการอยู่ในหน่วยความจำหลักหรือไม่ และกรณีที่เกิด page fault (ข้อมูลที่ต้องการมิได้อยู่ในหน่วยความจำหลัก) หน่วยประมวลผลกลางจะสร้างสัญญาณ interrupt เพื่อให้ซอฟต์แวร์ทำงาน ซึ่งรายละเอียดเพิ่มเติมสามารถศึกษาได้จากวิชาระบบปฏิบัติการ (Operating System)

หากโครงสร้างของระบบ virtual address และ physical address เป็นดังรูปที่ 8.17 จะพบว่า page table ต้องมีจำนวนบรรทัด (record(s)) ทั้งสิ้น 2^{20} บรรทัด (คิดจาก 2^{31-11}) ซึ่งคิดเป็น 1024×1024 บรรทัด หรือ 1 เมกะบรรทัด และ แต่ละบรรทัดนั้นจะประกอบด้วย (31-11) + 1 บิต (เนื่องจาก physical page number บอกด้วยบิตที่ 12-31 ของ physical address และจะต้องเก็บ valid bit) ซึ่งเมื่อแปลงข้อมูลเป็นหน่วยไบต์จะต้องทำการปัดเศษขึ้นเพื่อให้เป็นจำนวนเต็มได้ประมาณ 3 ไบต์ ซึ่งในทางปฏิบัติ จะมีการเก็บ permission และข้อมูลสำหรับอธิบาย page เพิ่มเติมด้วย จึงอาจได้ผลเป็น 4 ไบต์

ดังนั้นในกรณีนี้ page table จะมีขนาดใหญ่ถึง $4 \times 1024 \times 1024$ ไบต์ หรือ 4 เมกะไบต์

และเนื่องจาก page table นี้ก็ถูกเก็บไว้ในหน่วยความจำหลักเช่นกัน การหาข้อมูลจากตารางขนาด 4 เมกะไบต์ย่อมส่งผลให้การแปลงเลขดังกล่าวทำงานได้ช้า

เพื่อให้การแปลงเลข virtual page number ไปเป็น physical page number กระทำได้รวดเร็วยิ่งขึ้น จึงได้มีการประยุกต์ใช้ระบบ cache กับ page table ด้วยเช่นกัน เพื่อไม่ให้สับสน จึงเรียก cache ของ page table นี้ว่า Translation Look-aside Buffer (TLB) ดังแสดงในรูปที่ 8.18



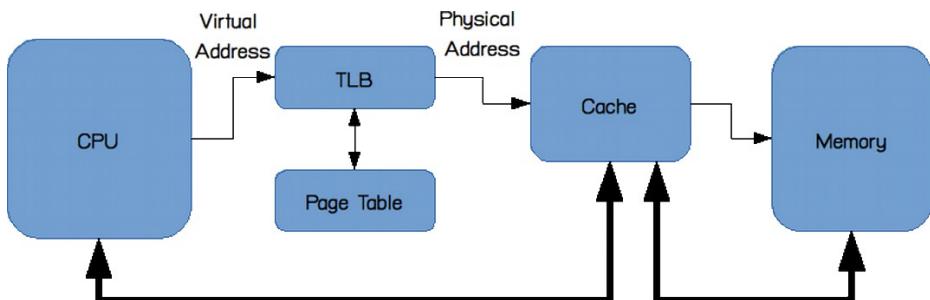
รูปที่ 8.18: การทำงานของ TLB

เนื่องจาก TLB มักมีขนาดเล็ก (ไม่เกิน 128 บรรทัด) เมื่อเทียบกับ page table ซึ่งมักมี 1 เมกะบรรทัดหน่วยประมาณผลกลางส่วนใหญ่จึงเลือกที่จะออกแบบให้ TLB ทำงานแบบ fully associate (มี conflict miss เป็น 0)

อีกประการหนึ่งคือ การ write through ในกรณีของ page นั้นจะมีค่าใช้จ่ายที่สูงมาก (เนื่องจาก storage ทำงานช้ามากเมื่อเทียบกับหน่วยความจำ) หน่วยประมวลผลกลางส่วนใหญ่จึงเลือกใช้วิธี write back เป็นหลัก

8.13 การทำงานร่วมกันของ Cache และหน่วยความจำเสมือน

เมื่อระบบหน่วยความจำเสมือนและระบบ cache ทำงานพร้อมกันทั้งหมดภายในหน่วยประมวลผลกลาง ก็จะทำให้ขั้นตอนในการประมวลผลข้อมูลจากของหน่วยประมวลผลกลาง นั้นชัดขึ้นยิ่งขึ้น โดยเริ่มตั้งแต่น่วยประมวลผลกลางได้รับ virtual address จากชอฟต์แวร์ ก็จะทำการแปลงข้อมูลให้เป็น physical address โดยค้นหาจาก page table ซึ่งการค้นหานี้จะเริ่มต้นจาก TLB ก่อน หากไม่พบก็จะทำการดึงข้อมูลจาก page table ตำแหน่งที่ต้องการเข้ามาไว้ใน TLB เมื่อได้ physical address แล้ว หากข้อมูลที่ต้องการไม่อยู่ภายในหน่วยความจำหลักก็จะสร้างสัญญาณ page fault เพื่อให้ซอฟต์แวร์ระบบปฏิบัติการจัดการดึงข้อมูลเข้ามาไว้ในหน่วยความจำหลักต่อไป เมื่อได้ physical address แล้ว หน่วยประมวลผลกลางจะค้นหาข้อมูลตำแหน่งดังกล่าวใน cache ของข้อมูล และทำการขั้นตอนของการ write hit หรือ read hit ทั้งนี้ขึ้นอยู่กับว่าคำสั่งที่ประมวลผลนั้นเป็นการอ่านหรือเขียนข้อมูล ส่วนกรณีการ miss นั้น ระบบจัดการ cache ข้อมูลจะทำการ stall หน่วยประมวลผลกลางแล้วดึงข้อมูลจากหน่วยความจำหลักเข้ามาเก็บใน cache เพื่อประมวลผลตามขั้นตอน การทำงานแสดงได้ดังรูปที่ 8.19



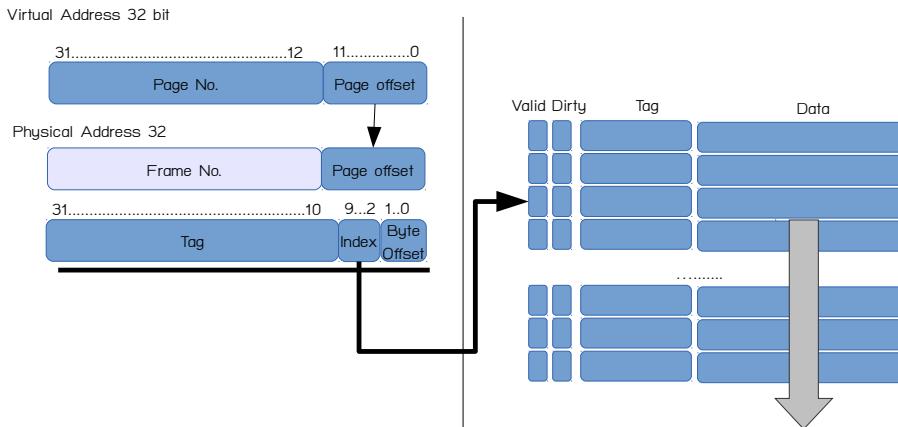
รูปที่ 8.19: การเข้าถึงข้อมูลในหน่วยความจำเมื่อมี cache และหน่วยความจำเสมือน

8.14 การเหลือมขั้นตอนของ Cache และหน่วยความจำเสมือน

จากรูปที่ 8.19 จะพบว่าขั้นตอนการทำงานในการอ่านหรือเขียนข้อมูลนั้นค่อนข้างซับซ้อน และกินเวลา ดังนั้น จึงมีแนวคิดว่า เป็นไปได้หรือไม่ที่จะเหลือมขั้นตอนการทำงานของ cache และ TLB (รวมถึง Page Table) ให้เร็วขึ้น คำตอบที่ได้คือ เป็นไปได้หากมีเงื่อนไขบางอย่างเป็นจริง ดังนี้

ลองพิจารณาการแปลงเลขที่อยู่และการนำค่าที่ได้ไปอ้างอิงข้อมูลใน cache ตามรูปที่ 8.20 หากการเปลี่ยนแปลง จาก virtual address ไปเป็น physical address ไม่ทำให้ค่า index

ที่ใช้ในการค้นหาข้อมูลใน cache เปลี่ยนแปลงไป (ค่า cache index อยู่ภายใต้ขอบเขตของ page offset) ยอมเป็นไปได้ที่การหาข้อมูลใน cache จะสามารถนำข้อมูล page offset จาก virtual address ไปใช้ได้ก่อน โดยไม่จำเป็นต้องรอให้มีการคำนวณค่า physical address (เพราะค่าที่ได้จะเหมือนเดิม)

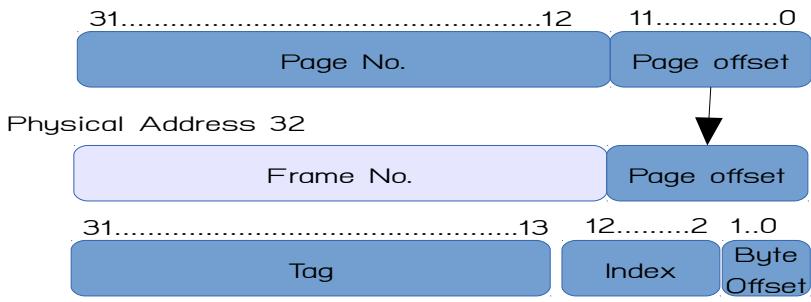


รูปที่ 8.20: การแปลง virtual address และการอ้างอิงเลขที่อยู่ของ cache

อย่างไรก็ตาม ประเด็นหนึ่งที่ตามมาคือ หากต้องการเพิ่ม cache ให้มีขนาดใหญ่มากขึ้นจนจำานวนบิตที่ได้เกินขอบเขตที่กำหนด จะทำอย่างไร จากตัวอย่าง (รูปที่ 8.21) จะเห็นว่าการเปลี่ยนแปลงจาก page number เป็น frame number อาจส่งผลให้ค่า cache index เปลี่ยนแปลงไปได้ กรณีหากยังคงต้องการใช้โครงสร้าง cache ในลักษณะนี้ อาจจะต้องแก้ปัญหาด้วยวิธีการทางซอฟต์แวร์แทน เนื่องจากระบบปฏิบัติการเป็นผู้กำหนด frame number ให้แต่ละ page ดังนั้น หากสามารถบังคับให้การเปลี่ยนแปลงจาก page number เป็น frame number ไม่มีการเปลี่ยนแปลงนี้เกิดขึ้น ก็จะยังคงทำการเหลือเวลาการเข้าถึงข้อมูลได้เหมือนเดิม

ทั้งนี้ การบังคับด้วยซอฟต์แวร์ อาจจะไม่ใช่ทางออกที่ดีนัก ดังนั้น หากต้องการเพิ่มขนาดของ cache โดยให้โครงสร้างเลขที่อยู่ในการอ้างอิงข้อมูลใน cache ไม่เปลี่ยนแปลงไปจากเดิม ทางเลือกที่เหมาะสมกว่าคือ การเพิ่มด้วยการขยาย associativity แทน (เช่น เพิ่มจาก 2-way set เป็น 4-way set) ด้วยวิธีการนี้ จะได้ cache มากขึ้น มี conflict miss น้อยลง และยังได้สมรรถนะในการค้นหาข้อมูลโดยใช้ offset จาก virtual address ได้อีกด้วย

Virtual Address 32 bit



รูปที่ 8.21: โครงสร้างเลขที่อยู่ที่ไม่สามารถทำการเหลือมเวลาได้

8.15 สรุป

ระบบหน่วยความจำเสมือนและระบบ cache เป็นระบบที่ช่วยเสริมสมรรถนะของหน่วยประมวลผลกลางในการทำงานที่เกี่ยวข้องกับหน่วยความจำ โดยระบบ cache จะช่วยอำนวยความสะดวกให้หน่วยประมวลผลกลางสามารถอ่านข้อมูลจากหน่วยความจำหลักได้เร็วขึ้น ซึ่งผลที่ได้คือหน่วยประมวลผลกลางจะสามารถทำงานได้ที่ความต้องสัญญาณพิเศษที่สูงขึ้น เพราะไม่ต้องรอการทำงานของหน่วยความจำ อย่างไรก็ตามสมรรถนะของระบบนั้นจะดีหรือไม่ขึ้นอยู่กับค่า miss ratio กับค่า miss penalty ด้วย สำหรับระบบ virtual memory นั้น ช่วยให้โปรแกรมเมอร์สามารถพัฒนาซอฟต์แวร์ได้เป็นอิสระโดยไม่ต้องสนใจระบบหน่วยความจำของเครื่องนั้น ๆ และยังให้ซอฟต์แวร์ที่ต้องการหน่วยความจำสูงสามารถทำงานบนเครื่องที่มีหน่วยความจำหลักน้อยได้

ในด้านการทำงาน cache เป็นการทำงานที่ไม่ขึ้นกับซอฟต์แวร์ (ทำงานได้โดยไม่ต้องมีซอฟต์แวร์ช่วย) ในขณะที่หน่วยความจำเสมือนจะต้องอาศัยการทำงานของระบบปฏิบัติการ และหน่วยประมวลผลกลางร่วมด้วย

ในหนังสือเล่มนี้ไม่ได้กล่าวถึงหน่วยความจำเสมือนในลักษณะอื่น (เช่น segmentation หรือ overlay) ซึ่งเป็นการจัดการที่ไม่ได้เป็นที่นิยมใช้แพร่หลายทั่วไป แต่หากผู้เรียนสนใจ ขอแนะนำให้ศึกษาเพิ่มเติมยิ่งขึ้นไปอีก

8.16 แบบฝึกหัดท้ายบท

1. หลักการท้องถิน ช่วยให้หน่วยประมวลผลกลางสามารถทำงานเร็วขึ้นได้อย่างไร
2. จงอธิบายหลักการทำงานของหลักการท้องถินเชิงเวลา และ หลักการท้องถินเชิงพื้นที่ พร้อมยกตัวอย่างประกอบคำอธิบาย
3. หากท่านเป็นผู้ออกแบบ cache ท่านจะมีหลักเกณฑ์ในการเลือกใช้ระบบ write back และ write through อย่างไร จงให้เหตุผลประกอบคำอธิบาย
4. มีปัจจัยใดบ้างที่มีผลต่อสมรรถนะของ cache และเราสามารถปรับปรุงสมรรถนะของ cache ได้อย่างไรบ้าง จงอธิบาย
5. โปรแกรมทดสอบอันหนึ่งมีชุดคำสั่งทั้งสิ้น 2 ล้านคำสั่ง หาก miss rate เป็น 10% และ miss penalty เป็น 6 รอบการประมวลผล หากหน่วยประมวลผลกลางดังกล่าวมี CPI เป็น 1 และมีความต่ำสัญญาณนาฬิกาเป็น 200Mhz จงแสดงการคำนวณเวลาในการประมวลผลโปรแกรมทดสอบของหน่วยประมวลผลกลางดังกล่าว
6. จงแสดงการออกแบบโครงสร้างภายในของ cache เพื่อให้รองรับการทำงานแบบ fully associative
7. จงแสดงการออกแบบโครงสร้างภายในของ cache เพื่อให้รองรับการทำงานแบบ 8-way associative
8. นักออกแบบท่านหนึ่งได้ให้ความเห็นว่า การทำ multilevel cache เป็นการ tradeoff ระหว่าง hit time และ miss penalty ท่านเห็นด้วยหรือไม่ อย่างไร
9. โดยปกติการทำงานของ cache จะไม่เข้าอยู่กับโปรแกรม (กล่าวคือ โปรแกรมสามารถได้ประโยชน์จากการทำงานของ cache โดยไม่ต้องทำการแก้ไขอะไร) อย่างไรก็ตาม หากนักพัฒนาซอฟต์แวร์ทราบถึงโครงสร้างการทำงานของ cache จะสามารถพัฒนาโปรแกรมเพื่อให้ใช้ประโยชน์จาก cache ได้ดีขึ้นหรือไม่ อย่างไร จงยกตัวอย่างประกอบคำอธิบาย
10. หากกำหนดโปรแกรมให้ ท่านมีแนวทางในการวิเคราะห์ขนาดของ cache ขั้นต่ำ ที่เหมาะสมกับการทำงานของโปรแกรมนี้อย่างไร

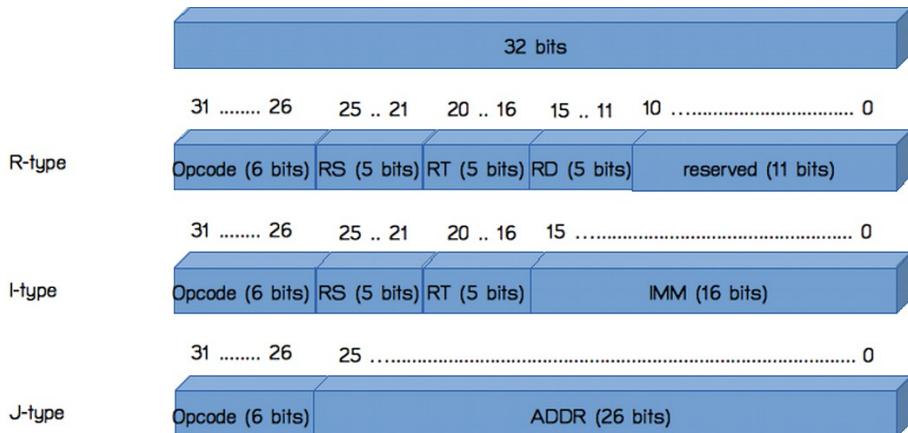
11. หากเครื่องคอมพิวเตอร์ระบบหนึ่งซึ่งมี cache ทำงานที่ความเร็ว 2 ns และหน่วยความจำหลักทำงานที่ 200 ns แล้ว การเพิ่ม cache อีกรอบหนึ่งซึ่งมีความเร็ว 20 ns แทรกเข้าไป จะช่วยเพิ่มสมรรถนะได้หรือไม่ อย่างไร จงแสดงการคำนวณประกอบการอธิบาย
12. จงเขียนโปรแกรม (ด้วยภาษาอะไรก็ได้) เพื่อจำลองการทำงานของ cache ที่มี Replacement Algorithm เป็นแบบ LRU และ RR จากนั้น ให้ลองนำ address trace จากโปรแกรมที่ว่าไป (หา download ได้ที่ว่าไป) ลองทำการประเมินสมรรถนะของ replacement algorithm ทั้งสองแบบ ว่าแบบใดให้ผลลัพธ์กว่ากัน
13. การมีระบบหน่วยความจำเสมือนนั้น เป็นประโยชน์สำหรับผู้พัฒนาซอฟต์แวร์หรือไม่ อย่างไร
14. ในหน่วยประมวลผลกลางหนึ่ง มี page ขนาด 4 กิโลไบต์ ผู้ออกแบบหน่วยประมวลผลกลางต้องการสร้าง cache ขนาด 32 กิโลไบต์ เพื่อช่วยให้การอ้างอิงข้อมูลเร็วขึ้น หากต้องการออกแบบให้มีการเหลือมการทำงานของ cache และการแปลง virtual address เป็น physical address เพื่อเพิ่มสมรรถนะด้วย จะเป็นไปได้หรือไม่อย่างไร จงแสดงการออกแบบ ประกอบการให้เหตุผล

ภาคผนวก

ภาคผนวก ก สถาปัตยกรรมชุดคำสั่ง nanoLADA

สถาปัตยกรรม nanoLADA ประกอบด้วยคำสั่งจำนวน 7 คำสั่ง และมีจำนวนรูปแบบคำสั่ง (Instruction format) 3 แบบ คือ R-type, I-type, และ J-type เรจิสเตอร์ขนาด 8 บิต จำนวน 32 ชุด (\$r0 ถึง \$r31) โดยมีรายละเอียดดังต่อไปนี้

n.1 รูปแบบคำสั่ง 3 รูปแบบ



n.2 คำสั่งของสถาปัตยกรรม nanoLADA

ORI rt, rs, imm	; (I-type)	Opcode 010000
ความหมาย	$R[rt] \leftarrow R[rs] \mid \text{zero_ext}(imm);$	$PC \leftarrow PC + 4$
ORUI rt, rs, imm	; (I-type)	Opcode 010001
ความหมาย	$R[rt] \leftarrow R[rs] \mid \text{zero_pad}(imm);$	$PC \leftarrow PC + 4$
ADD rd, rs, rt	; (R-type)	Opcode 000001

ความหมาย $R[rd] \leftarrow R[rs] + R[rt];$ $PC \leftarrow PC + 4$

LW rt, rs, imm ; (I-type) **Opcode 011000**

ความหมาย $R[rt] \leftarrow MEM[R[rs] + \text{sign_ext}(imm)];$ $PC \leftarrow PC + 4$

SW rt, rs, imm ; (I-type) **Opcode 011100**

ความหมาย $MEM[R[rs] + \text{sign_ext}(imm)] \leftarrow R[rt];$ $PC \leftarrow PC + 4$

BEQ rs, rt, imm*4 ; (I-type) **Opcode 100100**

ความหมาย If ($R[rs] == R[rt]$) then

$PC \leftarrow PC + 4 + (\text{sign_ext}(imm) * 4)$

else $PC \leftarrow PC + 4$

JMP addr*4 ; (J-type) **Opcode 110000**

ความหมาย $PC \leftarrow (\text{addr} * 4)$

ภาคผนวก ฯ Verilog HDL ของ สถาปัตยกรรม nanoLADA

Sample code can be downloaded from

<https://www.cp.eng.chula.ac.th/~krerk/books/Computer%20Architecture/nanoLADA/>

၇.၁ Register Files

```
`timescale 10ns/10ns
//-----
// File name      : regfile.v
// Title         : Register Files.
// Library        : nanoLADA
// Purpose        : Computer Architecture, Design and Verification
// Developers     : Krerk Piromsopa, Ph. D.
//                  : Chulalongkorn University.
module regfile(A, B, data, ra, rb, rw, nwr, clock);
output      [31:0]      A;
output      [31:0]      B;
input       [31:0]      data;
input       [4:0] ra, rb, rw;
input          nwr;
input          clock;
reg         [31:0] regs[31:0];

assign A = regs[ra];
assign B = regs[rb];

// initial data to reset registers (for simulation)
integer i;
initial
begin
    for(i=0;i<32;i++)
    begin
        regs[i]=0;
    end
end

always @ (posedge clock)
begin
    $display("%10d - A(REG[%d]) - %h, B(REG[%d]) - %h\n",
            i, ra, A, rb, B);
end
endmodule
```

```
$time, ra,A,rb,B);
  if (nwr==0)
begin
    regs[rw] = data;
    $display("%10d - REG[%d] <- %h",$time, rw,
data);
end
end

endmodule
```

v.2 Extender

```
`timescale 1ns / 1ps
//-----
// File name      : extender.v
// Title         : Extender.
// Library       : nanoLADA
// Purpose        : Computer Architecture
// Developers     : Krerk Piromsopa, Ph. D.
//                  : Chulalongkorn University.

module extender(data32, data16,ext_ops);
output reg [31:0] data32;
input [15:0] data16;
input [1:0] ext_ops;

always@(ext_ops or data16)
begin
  case(ext_ops)
    2'b01: data32 = {{16{data16[15]}},data16};
    2'b10: data32 = {data16,{16{1'b0}}};
    default: data32 = {{16{1'b0}},data16};
  endcase
end
endmodule
```

v.3 ALU

```
`timescale 1ns / 1ps
//-----
// File name      : alu.v
// Title         : ALU.
// Library       : nanoLADA
// Purpose        : Computer Architecture
// Developers     : Krerk Piromsopa, Ph. D.
//                  : Chulalongkorn University.
```

```

module alu(S,z,Cout,A,B,Cin,alu_ops);
output reg [31:0] S;
output z;
output reg Cout;
input [31:0] A;
input [31:0] B;
input Cin;
input [1:0] alu_ops;

assign z=~|S;

always @ (A or B or alu_ops)
begin
    case (alu_ops)
        2'b01: begin S=A|B; Cout=0; end
        2'b10: {Cout,S}=A-B;
        default: {Cout,S}=A+B+Cin;
    endcase
end
endmodule

```

v.4 ADDER

```

`timescale 1ns / 1ps
//-----
// File name      : adder.v
// Title         : ADDER.
// Library       : nanoLADA
// Purpose       : Computer Architecture
// Developers    : Krerk Piromsopa, Ph. D.
//                  : Chulalongkorn University.
module adder(S,Cout,A,B,Cin);
parameter WIDTH=31;
output      [WIDTH-1:0]      S;
output      Cout;
input       [WIDTH-1:0]      A;
input       [WIDTH-1:0]      B;
input       Cin;

assign {Cout,S}=A+B+Cin;
endmodule

```

7.5 Mux 2:1

```
`timescale 1ns / 1ps
//-----
// File name      : mux.v
// Title         : MUX.
// Library        : nanoLADA
// Purpose        : Computer Architecture
// Developers     : Krerk Piromsopa, Ph. D.
//                  : Chulalongkorn University.
module mux2_1(out,in0,in1,sel);
parameter WIDTH=32;
output      [WIDTH-1:0]      out;
input       [WIDTH-1:0]      in0;
input       [WIDTH-1:0]      in1;
input                           sel;

assign out=sel?in1:in0;

endmodule
```

7.6 control unit

```
`timescale 1ns / 1ps
//-----
// File name      : control.v
// Title         : Control Unit.
// Library        : nanoLADA
// Purpose        : Computer Architecture, Design and Verification
// Developers     : Krerk Piromsopa, Ph. D.
//                  : Chulalongkorn University.

module control(
    sel_pc,
    sel_addpc,
    sel_wr,
    sel_b,
    sel_data,
    reg_wr,
    mem_wr,
    ext_ops,
    alu_ops,
    opcode,
    z_flag
);
output reg sel_pc;
output reg sel_addpc;
```

```
output reg sel_wr;
output reg sel_b;
output reg sel_data;
output reg reg_wr;
output reg mem_wr;
output reg [1:0] ext_ops;
output reg [1:0] alu_ops;
input [5:0] opcode;
input z_flag;

localparam ORI=6'b010000;
localparam ORUI=6'b010001;
localparam ADD=6'b000001;
localparam LW=6'b011000;
localparam SW=6'b011100;
localparam BEQ=6'b101000;
localparam JMP=6'b110000;

// sel_pc
always @ (opcode)
begin
    case (opcode)
        JMP : sel_pc=1;
        default : sel_pc=0;
    endcase
end

// sel_addpc
always @ (opcode or z_flag)
begin
    case (opcode)
        BEQ : sel_addpc=z_flag;
        default : sel_addpc=0;
    endcase
end

// sel_wr
always @ (opcode)
begin
    case (opcode)
        ORI : sel_wr=1;
        ORUI : sel_wr=1;
        LW : sel_wr=1;
        default : sel_wr=0;
    endcase
end

// sel_b
always @ (opcode)
```

```
begin
    case (opcode)
        ORI : sel_b=1;
        ORUI : sel_b=1;
        LW : sel_b=1;
        SW : sel_b=1;
        default : sel_b=0;
    endcase
end

// sel_data
always @(opcode)
begin
    case (opcode)
        LW : sel_data=1;
        default : sel_data=0;
    endcase
end

// reg_wr
always @(opcode)
begin
    case (opcode)
        ORI : reg_wr=1;
        ORUI : reg_wr=1;
        ADD : reg_wr=1;
        LW : reg_wr=1;
        default : reg_wr=0;
    endcase
end

// mem_wr
always @(opcode)
begin
    case (opcode)
        SW : mem_wr=1;
        default : mem_wr=0;
    endcase
end

// ext_ops
always @(opcode)
begin
    case (opcode)
        ORUI : ext_ops=2'b10;
        LW : ext_ops=2'b01;
        SW : ext_ops=2'b01;
        BEQ : ext_ops=2'b01;
        default : ext_ops=2'b00;
    endcase
end
```

```
        endcase
    end

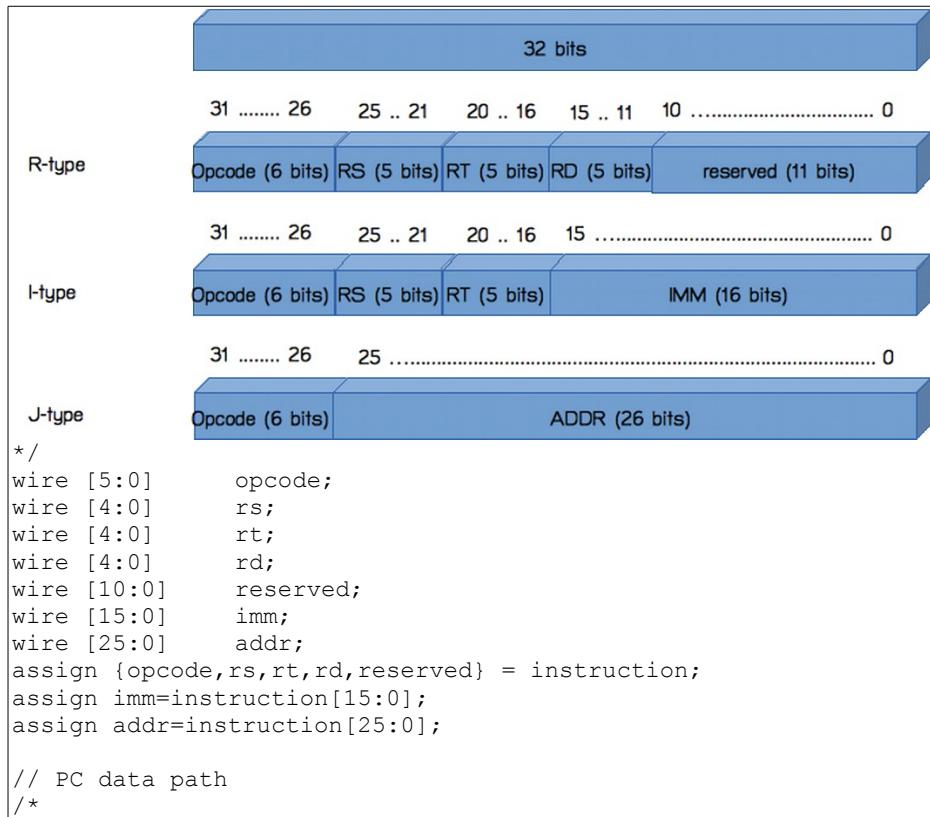
// alu_ops
always @ (opcode)
begin
    case (opcode)
        ORI : alu_ops=2'b01;
        ORUI : alu_ops=2'b01;
        BEQ : alu_ops=2'b10;
        default : alu_ops=2'b00;
    endcase
end

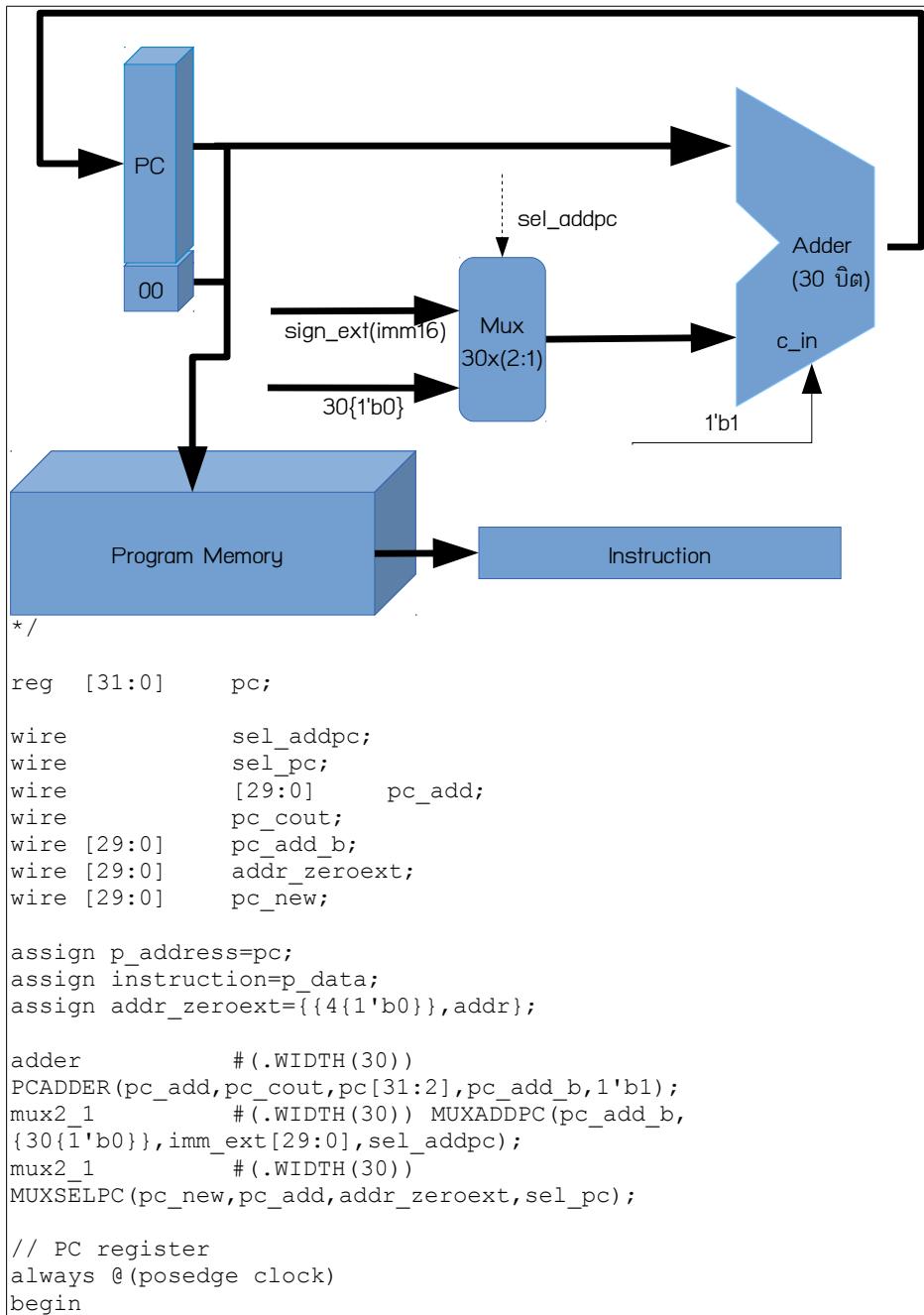
endmodule
```

7.7 nanoCPU

```
-----
// File name      : nanocpu.v
// Title         : nanoCPU.
// Library       : nanoLADA
// Purpose        : Computer Architecture
// Developers     : Krerk Piromsopa, Ph. D.
//                  : Chulalongkorn University.
module
nanocpu(p_address,p_data,d_address,d_data,mem_wr,clock,nreset);
output      [31:0]      p_address;
input       [31:0]      p_data;
output      [31:0]      d_address;
output          mem_wr;
inout      [31:0]      d_data;
input           clock;
input           nreset;

wire [31:0]      instruction;
// instruction fields
```



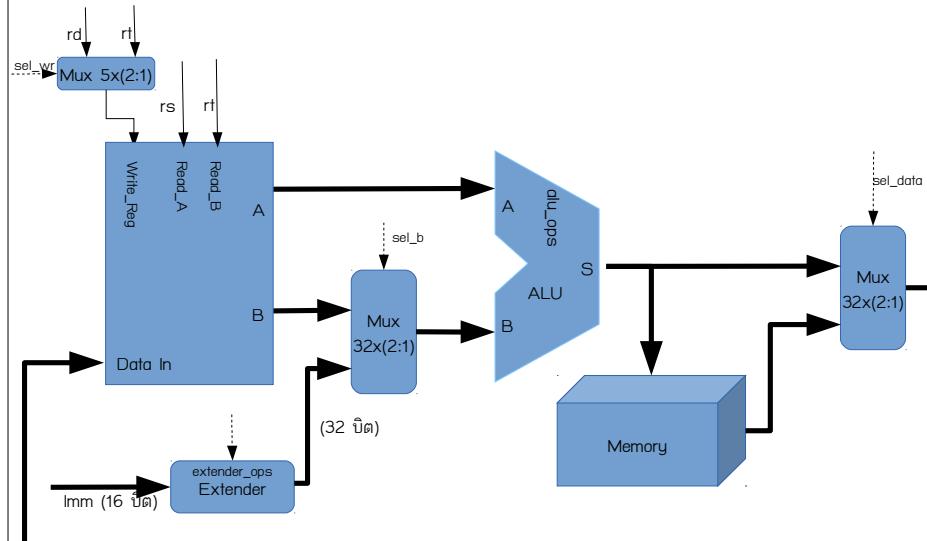


```

if (nreset==0)
begin
    pc=0;
end
else
begin
    pc={pc_new,2'b00};
end
end

```

```
// Main data path
/*
```



```

*/
wire [31:0]      imm_ext;
wire [1:0]       ext_ops;
extender        EXTENDER(imm_ext, imm, ext_ops);

reg             z_flag;
reg             c_flag;

wire reg_wr;
wire sel_wr;
wire sel_data;
wire sel_b;
wire z_new;
wire c_new;
wire [1:0]      alu_ops;

wire [31:0]      A;

```

```
wire [31:0]      B;
wire [31:0]      data_selected;
wire [31:0]      data_S;
wire [31:0]      data_M;
wire [31:0]      B_selected;
wire [4:0]       rw;

assign d_address=data_S;
assign data_M=d_data;
assign d_data=(mem_wr==1)?B:32'bzz;

mux2_1           #(.WIDTH(5)) MUXRW(rw,rd,rt,sel_wr);
regfile          REGFILE(A, B, data_selected, rs, rt, rw,
~reg_wr, clock);
mux2_1           MUXDATA(data_selected,data_S,data_M,sel_data);
mux2_1           MUXB(B_selected,B,imm_ext,sel_b);
alu              ALU(data_S,z_new,c_new,A,B_selected,c_flag,alu_ops);

control          CONTROLUNIT(
    sel_pc,
    sel_addpc,
    sel_wr,
    sel_b,
    sel_data,
    reg_wr,
    mem_wr,
    ext_ops,
    alu_ops,
    opcode,
    z_new
);

// flag
always @ (posedge clock)
begin
    if (nreset==0)
    begin
        z_flag=0;
        c_flag=0;
    end
    else
    begin
        z_flag=z_new;
        c_flag=c_new;
    end
end
```

```
endmodule
```

7.8 Micro PC

```
`timescale 1ns / 1ps
//-----
// File name      : micropc.v
// Title         : Micro PC.
// Library        : nanoLADA - microsequencer
// Purpose        : Computer Architecture, Design and Verification
// Developers     : Krerk Piromsopa, Ph. D.
//                   : Chulalongkorn University.
module micropc(mpc,lpc,z,i,l,clock);
output reg      [3:0] mpc;
input           [3:0] lpc;
input          z;
input          i;
input          l;
input          clock;

reg [3:0] mpc_new;

always @(posedge clock)
begin
    mpc=mpc_new;
end

always @(z or i or l)
begin
    if (z==1)
        mpc_new=4'b0000;
    else if (i==1)
        mpc_new=mpc+1;
    else if (l==1)
        mpc_new=lpc;
end
endmodule
```

ภาคผนวก ค กิจกรรมเพิ่มเติม

ค.1 กิจกรรมบทบาทของคอมพิวเตอร์

This activity will get you familiar with performance measurement, instruction set architecture, and compiler. It should give you a general idea on how a compiler optimize software. In particular, you will learn how architect measure performance of a system.

Prologue

(If you do not use Windows, you may skip this part.)

For Windows, Cygwin, a linux-like environment for Windows, can be used. If you want to use Cygwin, please go to <http://www.cygwin.com/> and download the installation software (setup.exe). Note that you need an internet connection for the installation.

To install Cygwin, run "setup.exe". The program will ask for a mirror site for installation, pick an appropriate choice (e.g. a mirror in Japan or Asia). There are only two main components required for this exercise: **gcc** and **bin-utils**. Other choices are optional. I recommend that you install pico or vi for using as your text editor. The installation may take hours (depending on your internet connection).

After the installation, you will find a program group "cygwin" in your Windows start menu. Run the "cygwin".

Now, let's test that gcc is working by compiling a simple program. This is "hello.c". Use your favorite editor (e.g. notepad) to write this program. Note that all cygwin path are relative to c:\cygwin\ (i.e. /home in cygwin will be c:\cygwin\home in Windows).

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    printf ("Hello, World\n");
}
```

run **gcc -o hello hello.c**

The program will give hello (or hello.exe on Windows). Type **./hello** (or **./hello.exe** on Cygwin) to run your program. To disassembler, try **objdump -d hello.exe**. (For Mac OS X, try **otool -tV hello**) I know that assembly may be geek to you, but that is not an issue (at least for now).

To compile a program with optimization level (between 0 and 3), use **-O[level]** as a parameter. (e.g. **gcc -O3 -o hello hello.c**.) Please note that clang-based compiler (e.g. X-code on Mac OS X) only supports level 1 (-O1) optimization.

To compile a program with immediate assembly result, use **-S** as a parameter. (e.g. **gcc -S hello.c**) The assembly will have **.s** as extension (e.g. **hello.s**.)

If you got it all working, you are ready for the exercise.

Exercises

1. (Instruction Analysis) This exercise will familiar you with several aspects of instruction set and the fundamental of compiler. Given **max.c** (below), please use “**gcc -S max.c**” to compile the code into assembly code. (The result will be in **max.s**.) From the result, answer the following questions.

```
int max1(int a, int b) {
    return (a>b)?a:b;
}
int max2(int a, int b) {
    int isaGTb=a>b;
```

```

int max;
if (isaGTb)
    max=a;
else
    max=b;
return max;
}

```

- What does the code hint about the kind of instruction set? (e.g. Accumulator, Register Memory, Memory Memory, Register Register) Please justify your answer.
- Can you tell whether the architecture is either **Restricted Alignment** or **Unrestricted Alignment**? Please explain how do you come up with your answer.
- Create a new function (e.g. testMax) to call *max1*. Generate new assembly code. What does the result suggest regarding the register saving (caller save vs. callee save)? Please provide your analysis.

```

void testMax() {
    int maxNum=0;
    maxNum=max1(20, 40);
}

```

- How does the arguments be passed and the return value returned from a function? Please explain the code.
- Find the part of code (snippet) that do comparison and conditional branch. Explain how does it work.
- If *max.c* is complied with optimization turned on (using “*gcc -O2 -S max.c*”), what are the differences that you may observe from the result (comparing to that without optimization). Please provide your analysis
- Please estimate the CPU Time required by the *max1* function (using the equation $CPU\ TIME = IC \times CPI \times T_c$). If possible, create a main function to calculate *max1* and use the **time** command to measure the performance. Compare the measurement to your estimation. What do you think are

the factors that cause the difference? Please provide your analysis.
(You may find references online regarding the CPI of each instruction.)

2. (Optimization) We will use simple fibonacci calculation as a benchmark. Please measure the execution time (using the `time` command) of this given program when compiling with optimization level 0 (no optimization), level 1, level 2 and level 3. (Note that some compilers do similar optimization for all level 1, level 2 and level 3. If that is the case, you will see no difference after level 1.) You may want to run each program for few times and use the average value as a result.

```
#include <stdio.h>
long fibo(long a) {
    if (a<=0L) {
        return 0L;
    }
    if (a==1L) {
        return 1L;
    }
    return fibo(a-1L)+fibo(a-2L);
}
int main (int argc,char *argv[]) {
    for (long i=1L;i<45L;i++) {
        long f=fibo(i);
        printf("fibo of %ld is %ld\n",i,f);
    }
}
```

3. (Analysis) As suggesting by the result in Exercise 2, what kinds of optimization used by the compiler in each level in order to make the program faster? To answer this question, use “gcc -S” to generate the assembly code for each level (e.g. “gcc -S -O2 fibo.c”) and use this result as a basis for your analysis. (Depending on your version of compiler, the result may vary.)

ค.2 กิจกรรมโครงสร้าง Cache และสมรรถนะ

In this exercise, we will use the cache simulator for studying the factors that affects the performance of cache accesses. The cache simulator will read an address trace from gcc_ld_trace.txt or go_ld_trace.txt. The simulator will show us a number of cache hit, a number of cache miss, miss rate, and access time. Please use only one address trace and modify the codes of simulator for calculating the **miss rate**. Please fill your results and plot graph of each table.

1. Block Size Tradeoff on direct mapped cache

Direct mapped				
Block Size (Bytes)	Cache Size (KB)			
	4	8	16	32
4				
8				
16				
32				

2. N-way associative cache with replacement algorithms: Least recently used (LRU), and Round Robin (RR).

Cache Size (KB)	Associativity			
	Two-way		Four-way	
LRU	RR	LRU	RR	
1				
4				
8				
32				
512				
1024				

Note that code and traces can be downloaded from

<https://www.cp.eng.chula.ac.th/~krerk/books/Computer%20Architecture/CacheSim>

បរណានុក្រម

- [1] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5 edition. Waltham, MA: Morgan Kaufmann, 2011.
- [2] K. Piromsopa and R. J. Enbody, "Architecting security: A secure implementation of hardware buffer-overflow protection," presented at the Proceedings of the 3rd IASTED International Conference on Advances in Computer Science and Technology, ACST 2007, 2007, pp. 17–22.
- [3] "Amdahl's law," *Wikipedia, the free encyclopedia*. 15-Jun-2014.
- [4] R. R. Schaller, "Moore's law: past, present and future," *IEEE Spectr.*, vol. 34, no. 6, pp. 52–59, Jun. 1997.
- [5] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th Edition, 4 edition. Amsterdam; Boston: Morgan Kaufmann, 2006.
- [6] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 2 edition. San Francisco, Calif: Morgan Kaufmann, 1997.
- [7] N. P. Jouppi, "The nonuniform distribution of instruction-level and machine parallelism and its effect on performance," *IEEE Trans. Comput.*, vol. 38, no. 12, pp. 1645–1658, Dec. 1989.
- [8] P. S. Oberoi and G. S. Sohi, "Parallelism in the front-end," in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings*, 2003, pp. 230–240.
- [9] S. Borkar, N. P. Jouppi, and P. Stenstrom, "Microprocessors in the Era of Terascale Integration," in *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, 2007, pp. 1–6.
- [10] "Intel® 64 and IA-32 Architectures Software Developer Manuals," *Intel*. [Online]. Available: <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>. [Accessed: 19-Jun-2014].
- [11] "ARM architecture," *Wikipedia, the free encyclopedia*. 18-Jun-2014.

- [12] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August, "Compiler Technology for Future Microprocessors," in *Proceedings of the IEEE*, 1995, pp. 1625–1640.
- [13] "Verilog," *Wikipedia, the free encyclopedia*. 18-Jun-2014.
- [14] M. D. Hill and A. J. Smith, "Evaluating associativity in CPU caches," *IEEE Trans. Compu.*, vol. 38, no. 12, pp. 1612–1630, Dec. 1989.
- [15] R. H. Katz and G. Borriello, *Contemporary Logic Design*, 2 edition. Upper Saddle River, N.J: Prentice Hall, 2004
- [16] Brehob, M., Doom, T. E., Enbody, R., Moore, W. H., Moore, S. Q., Sass, R., & Severance, C. (1996). Beyond RISC - The Post-RISC Architecture. . <http://corescholar.libraries.wright.edu/cse/294>
- [17] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, Sep. 1972.
- [18] Chris Lomont, "Introduction to Intel® Advanced Vector Extensions," May. 2011., <https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions>
- [19] Alex Boud, "The OpenCL Specification," <https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf>
- [20] http://www.nvidia.com/object/cuda_home_new.html
- [21] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, Jan. 1967.
- [22] M. Milkes, "The Genesis of Microprogramming," in *IEEE Annals of the History of Computing*, vol. 8, no. , pp. 116-126, 1986.
- [23] I. H. Witten and J. G. Cleary, "An introduction to the architecture of the Intel iAPX 432," *Software Microsystems*, vol. 2, no. 2, pp. 29–34, Apr. 1983.
- [24] J. A. Fisher, "Very Long Instruction Word Architectures and the ELI-512," in *Proceedings of the 10th Annual International Symposium on Computer Architecture*, New York, NY, USA, 1983, pp. 140–150.
- [25] R. Rangan, N. Vachharajani, G. Ottoni, and D. I. August, "Performance Scalability of Decoupled Software Pipelining," *ACM Trans. Archit. Code*

- Optim., vol. 5, no. 2, p. 8:1–8:25, Sep. 2008.
- [26] “The LLVM Compiler Infrastructure Project.” [Online]. Available: <http://llvm.org/>. [Accessed: 03-Mar-2014].
 - [27] “clang’ C Language Family Frontend for LLVM.” [Online]. Available: <https://clang.llvm.org/>. [Accessed: 03-Mar-2014].
 - [28] “OpenMP*: Support for the OpenMP language.” [Online]. Available: <http://openmp.llvm.org/>. [Accessed: 03-Mar-2018].
 - [29] “SPEC - Standard Performance Evaluation Corporation.” [Online]. Available: <http://spec.org/>. [Accessed: 03-Mar-2014].
 - [30] “CUDA,” Wikipedia. 01-Mar-2014.
 - [31] “OpenCL,” Wikipedia. 07-Feb-2014.